

Segunda edición



SISTEMAS OPERATIVOS

Diseño e implementación



ANDREW S. TANENBAUM
ALBERT S. WOODHULL

SISTEMAS OPERATIVOS:

Diseño e implementación

Segunda edición

Andrew S. Tanenbaum

*Vrije Universiteit
Amsterdam, Países Bajos*

Albert S. Woodhull

*Hampshire College
Amherst, Massachusetts*

TRADUCCIÓN:

Roberto Escalona
Traductor profesional

REVISIÓN TÉCNICA:

Raymundo Hugo Rangel Gutiérrez
Profesor de Ingeniería en Computación - UNAM



PRENTICE
HALL

MÉXICO • NUEVA YORK • BOGOTÁ • LONDRES • MADRID
MUNICH • NUEVA DELHI • PARÍS • RÍO DE JANEIRO • SINGAPUR
SYDNEY • TOKIO • TORONTO • ZURICH

EDICIÓN EN ESPAÑOL:

EDITOR:
SUPERVISORA DE TRADUCCIÓN:
SUPERVISOR DE EDICIÓN:

PABLO EDUARDO ROIG VÁZQUEZ
MA. TERESA SANZ FALCÓN
MAGDIEL GÓMEZ MARINA

EDICIÓN EN INGLÉS:

Acquisitions editor: ALAN APT
Developmental editor: SONDRÁ CHAVEZ
Production editor: ROSE KERNAN
Editor-in-chief: MARCIA HORTON
Copy editor: MARTHA WILLIAMS
Cover designer: BRUCE KENSELAAR
Director of production and manufacturing: DAVID W. RICCARDI
Managing editor: BAYANI MENDOZA DE LEÓN
Manufacturing buyer: DONNA SULLIVAN
Interior designer/Compositor: ANDREW S. TANENBAUM
Cover cartoon: JOS COLLIGNON

TANENBAUM/SISTEMAS OPERATIVOS: DISEÑO E IMPLEMENTACIÓN, 2a. ed.

Traducido del inglés de la obra: **OPERATING SYSTEMS: Design and Implementation**, 2nd ed.

All rights reserved. Authorized translation from English language edition published by Prentice Hall, Inc.
A Simon & Schuster Company.

Todos los derechos reservados. Traducción autorizada de la edición en inglés publicada por Prentice Hall, Inc.
A Simon & Schuster Company.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means,
electronic or mechanical, including photocopying, recording or by any information storage and retrieval
system, without permission in writing from the publisher.

Prohibida la reproducción total o parcial de esta obra, por cualquier medio o método sin autorización
por escrito del editor.

Derechos reservados © 1998 respecto a la segunda edición en español publicada por:
PRENTICE HALL HISPANOAMERICANA, S.A.
Calle 4, Núm. 25-2° piso, Col. Frac. Industrial Alce Blanco
53370 Naucalpan de Juárez, Edo. de México

ISBN 970-17-0165-8

Miembro de la Cámara Nacional de la Industria Editorial, Reg. Núm. 1524.

Original English Language Edition Published by Prentice Hall, Inc.
A Simon & Schuster Company
Copyright © 1997
All rights reserved

ISBN 0-13-638677-6

IMPRESO EN MÉXICO/PRINTED IN MEXICO



PROGRAMAS EDUCATIVOS, S. A. DE C.V.
CALZ. DINABACANO No. 66, LOCAL A
COL. ASTURIAS DELEG. CUAJATEMOC,
C.P. 06660, MÉXICO, D.F.

EMPRESA CERTIFICADA POR EL
INSTITUTO MEXICANO DE NORMALIZACIÓN
Y CERTIFICACIÓN A.C., BAJO LA NORMA
ISO-9002: 1994/ANSI-Z39-104: 1995
CON EL No. DE REGISTRO RSC-048



A Suzanne, Barbara, Marvin y Little Bram

- AST

A Barbara y Gordon

- ASW

CONTENIDO

PREFACIO

XV

1 INTRODUCCIÓN

1

- 1.1 ¿QUÉ ES UN SISTEMA OPERATIVO? 3
 - 1.1.1 El sistema operativo como máquina extendida 3
 - 1.1.2 El sistema operativo como administrador de recursos 4
- 1.2 HISTORIA DE LOS SISTEMAS OPERATIVOS 5
 - 1.2.1 La primera generación (1945-55): Tubos de vacío y tableros de conmutación 6
 - 1.2.2 La segunda generación (1955-65): Transistores y sistemas por lote 6
 - 1.2.3 La tercera generación (1965-1980): Circuitos integrados y multiprogramación 8
 - 1.2.4 La cuarta generación (1980-presente): Computadoras personales 12
 - 1.2.5 Historia de MINIX 13
- 1.3 CONCEPTOS DE SISTEMAS OPERATIVOS 15
 - 1.3.1 Procesos 15
 - 1.3.2 Archivos 17
 - 1.3.3 El shell 20
- 1.4 LLAMADAS AL SISTEMA 21
 - 1.4.1 Llamadas al sistema para administración de procesos 22
 - 1.4.2 Llamadas al sistema para señalización 26
 - 1.4.3 Llamadas al sistema para administración de archivos 28
 - 1.4.4 Llamadas al sistema para administración de directorios 33
 - 1.4.5 Llamadas al sistema para protección 35
 - 1.4.6 Llamadas al sistema para administración del tiempo 36

- 1.5 ESTRUCTURA DEL SISTEMA OPERATIVO 37
 - 1.5.1 Sistemas monolíticos 37
 - 1.5.2 Sistemas por capas 39
 - 1.5.3 Máquinas virtuales 40
 - 1.5.4 Modelo cliente-servidor 42
- 1.6 BOSQUEJO DEL RESTO DEL LIBRO 44
- 1.7 RESUMEN 44

2 PROCESOS

47

- 2.1 INTRODUCCIÓN A LOS PROCESOS 47
 - 2.1.1 El modelo de procesos 48
 - 2.1.2 Implementación de procesos 52
 - 2.1.3 Hilos 53
- 2.2 COMUNICACIÓN ENTRE PROCESOS 57
 - 2.2.1 Condiciones de competencia 57
 - 2.2.2 Secciones críticas 58
 - 2.2.3 Exclusión mutua con espera activa 59
 - 2.2.4 Dormir y despertar 63
 - 2.2.5 Semáforos 66
 - 2.2.6 Monitores 68
 - 2.2.7 Transferencia de mensajes 72
- 2.3 PROBLEMAS CLÁSICOS DE IPC 75
 - 2.3.1 El problema de la cena de filósofos 75
 - 2.3.2 El problema de lectores y escritores 77
 - 2.3.3 El problema del peluquero dormido 80
- 2.4 PLANIFICACIÓN DE PROCESOS 82
 - 2.4.1 Planificación *round robin* (de torneo) 84
 - 2.4.2 Planificación por prioridad 85
 - 2.4.3 Colas múltiples 86
 - 2.4.4 El primer trabajo más corto 87
 - 2.4.5 Planificación garantizada 89
 - 2.4.6 Planificación por lotería 89
 - 2.4.7 Planificación en tiempo real 90
 - 2.4.8 Planificación de dos niveles 92
 - 2.4.9 Política vs. mecanismo 93

- 2.5 PERSPECTIVA GENERAL DE PROCESOS EN MINIX 93
 - 2.5.1 La estructura interna de MINIX 93
 - 2.5.2 Administración de procesos en MINIX 95
 - 2.5.3 Comunicación entre procesos en MINIX 97
 - 2.5.4 Planificación de procesos en MINIX 98
- 2.6 IMPLEMENTACIÓN DE PROCESOS EN MINIX 98
 - 2.6.1 Organización del código fuente de MINIX 99
 - 2.6.2 Los archivos de cabecera común 102
 - 2.6.3 Los archivos de cabecera de MINIX 107
 - 2.6.4 Estructuras de datos de procesos y archivos de cabecera 112
 - 2.6.5 Autoarranque de MINIX 120
 - 2.6.6 Inicialización del sistema 122
 - 2.6.7 Manejo de interrupciones en MINIX 128
 - 2.6.8 Comunicación entre procesos en MINIX 137
 - 2.6.9 Planificación en MINIX 140
 - 2.6.10 Apoyo de kernel dependiente del hardware 142
 - 2.6.11 Rutinas de utilidad y biblioteca del kernel 145
- 2.7 RESUMEN 147

3 ENTRADA/SALIDA

153

- 3.1 PRINCIPIOS DEL HARDWARE DE E/S 154
 - 3.1.1 Dispositivos de E/S 154
 - 3.1.2 Controladores de dispositivos 155
 - 3.1.3 Acceso directo a memoria (DMA) 157
- 3.2 PRINCIPIOS DEL SOFTWARE DE E/S 159
 - 3.2.1 Objetivos del software de E/S 159
 - 3.2.2 Manejadores de interrupciones 161
 - 3.2.3 Controladores de dispositivos 161
 - 3.2.4 Software de E/S independiente del dispositivo 162
 - 3.2.5 Software de E/S de espacio de usuario 164
- 3.3 BLOQUEO MUTUO 166
 - 3.3.1 Recursos 167
 - 3.3.2 Principios del bloqueo mutuo 168
 - 3.3.3 El algoritmo del avestruz 170
 - 3.3.4 Detección y recuperación 172
 - 3.3.5 Prevención del bloqueo mutuo 173
 - 3.3.6 Evitar los bloqueos mutuos 175

3.4	GENERALIDADES DE E/S EN MINIX	179
3.4.1	Manejadores de interrupciones en MINIX	180
3.4.2	Controladores de dispositivos en MINIX	181
3.4.3	Software de E/S independiente del dispositivo en MINIX	185
3.4.4	Software de E/S de nivel de usuario en MINIX	185
3.4.5	Manejo de bloqueos mutuos en MINIX	186
3.5	DISPOSITIVOS POR BLOQUES EN MINIX	187
3.5.1	Generalidades de los controladores de dispositivos por bloques en MINIX	187
3.5.2	Software controlador de dispositivos de bloques común	190
3.5.3	La biblioteca de controladores	193
3.6	DISCOS EN RAM	195
3.6.1	Hardware y software de discos en RAM	196
3.6.2	Generalidades del controlador de disco en RAM en MINIX	197
3.6.3	Implementación del controlador de disco en RAM en MINIX	198
3.7	DISCOS	200
3.7.1	Hardware de discos	200
3.7.2	Software de discos	202
3.7.3	Generalidades del controlador de disco duro en MINIX	208
3.7.4	Implementación del controlador de disco duro en MINIX	211
3.7.5	Manejo de discos flexibles	220
3.8	RELOJES	222
3.8.1	Hardware de reloj	223
3.8.2	Software de reloj	224
3.8.3	Generalidades del controlador de reloj en MINIX	227
3.8.4	Implementación del controlador de reloj en MINIX	230
3.9	TERMINALES	235
3.9.1	Hardware de terminales	235
3.9.2	Software de terminales	240
3.9.3	Generalidades del controlador de terminales en MINIX	249
3.9.4	Implementación del controlador de terminales independiente del dispositivo	264
3.9.5	Implementación del controlador de teclado	282
3.9.6	Implementación del controlador de pantalla	288
3.10	LA TAREA DE SISTEMA EN MINIX	296
3.11	RESUMEN	304

4	ADMINISTRACIÓN DE MEMORIA	309
4.1	ADMINISTRACIÓN BÁSICA DE MEMORIA	310
4.1.1	Monoprogramación sin intercambio ni paginación	310
4.1.2	Multiprogramación con particiones fijas	311
4.2	INTERCAMBIO	313
4.2.1	Administración de memoria con mapas de bits	316
4.2.2	Administración de memoria con listas enlazadas	317
4.3	MEMORIA VIRTUAL	319
4.3.1	Paginación	319
4.3.2	Tablas de páginas	322
4.3.3	TLB—Buffers de consulta para traducción	327
4.3.4	Tablas de páginas invertidas	330
4.4	ALGORITMOS DE SUSTITUCIÓN DE PÁGINAS	331
4.4.1	El algoritmo de sustitución de páginas óptimo	331
4.4.2	El algoritmo de sustitución de páginas no usadas recientemente	332
4.4.3	El algoritmo de sustitución de páginas de primera que entra, primera que sale (FIFO)	333
4.4.4	El algoritmo de sustitución de páginas de segunda oportunidad	333
4.4.5	El algoritmo de sustitución de páginas por reloj	334
4.4.6	El algoritmo de sustitución de páginas menos recientemente usadas (LRU)	334
4.4.7	Simulación de LRU en software	336
4.5	ASPECTOS DE DISEÑO DE LOS SISTEMAS CON PAGINACIÓN	338
4.5.1	El modelo de conjunto de trabajo	338
4.5.2	Políticas de asignación local vs. global	339
4.5.3	Tamaño de página	341
4.5.4	Interfaz de memoria virtual	343
4.6	SEGMENTACIÓN	343
4.6.1	Implementación de la segmentación pura	347
4.6.2	Segmentación con paginación: MULTICS	348
4.6.3	Segmentación con paginación: El Pentium de Intel	352
4.7	GENERALIDADES DE ADMINISTRACIÓN DE MEMORIA EN MINIX	356
4.7.1	Organización de la memoria	358
4.7.2	Manejo de mensajes	361
4.7.3	Estructuras de datos y algoritmos del administrador de memoria	363
4.7.4	Las llamadas al sistema FORK, EXIT y WAIT	367
4.7.5	La llamada al sistema EXEC	368
4.7.6	La llamada al sistema BRK	371
4.7.7	Manejo de señales	372
4.7.8	Otras llamadas al sistema	378

- 4.8 IMPLEMENTACIÓN DE LA ADMINISTRACIÓN DE MEMORIA EN MINIX 379
 - 4.8.1 Los archivos de cabecera y estructuras de datos 379
 - 4.8.2 El programa principal 382
 - 4.8.3 Implementación de FORK, EXIT y WAIT 382
 - 4.8.4 Implementación de EXEC 385
 - 4.8.5 Implementación de BRK 386
 - 4.8.6 Implementación del manejo de señales 387
 - 4.8.7 Implementación de las otras llamadas al sistema 393
 - 4.8.8 Utilerías del administrador de memoria 394
- 4.9 RESUMEN 396

5 SISTEMAS DE ARCHIVOS 401

- 5.1 ARCHIVOS 402
 - 5.1.1 Nombres de archivos 402
 - 5.1.2 Estructura de archivos 404
 - 5.1.3 Tipos de archivos 405
 - 5.1.4 Acceso a archivos 407
 - 5.1.5 Atributos de archivos 408
 - 5.1.6 Operaciones con archivos 409
- 5.2 DIRECTORIOS 410
 - 5.2.1 Sistemas de directorio jerárquicos 411
 - 5.2.2 Nombres de ruta 412
 - 5.2.3 Operaciones con directorios 414
- 5.3 IMPLEMENTACIÓN DE SISTEMAS DE ARCHIVOS 415
 - 5.3.1 Implementación de archivos 415
 - 5.3.2 Implementación de directorios 419
 - 5.3.3 Administración del espacio en disco 422
 - 5.3.4 Confiabilidad del sistema de archivos 424
 - 5.3.5 Rendimiento del sistema de archivos 429
 - 5.3.6 Sistemas de archivos estructurados por diario 432
- 5.4 SEGURIDAD 434
 - 5.4.1 El entorno de seguridad 434
 - 5.4.2 Fallas de seguridad famosas 436
 - 5.4.3 Ataques genéricos contra la seguridad 439
 - 5.4.4 Principios de diseño para la seguridad 441
 - 5.4.5 Verificación de autenticidad de usuarios 442

5.5	MECANISMOS DE PROTECCIÓN	446
5.5.1	Dominios de protección	446
5.5.2	Listas de control de acceso	448
5.5.3	Capacidades	450
5.5.4	Canales encubiertos	451
5.6	GENERALIDADES DEL SISTEMA DE ARCHIVOS DE MINIX	453
5.6.1	Mensajes	454
5.6.2	Organización del sistema de archivos	454
5.6.3	Mapas de bits	458
5.6.4	Nodos-i	460
5.6.5	El caché de bloques	461
5.6.6	Directorios y rutas	463
5.6.7	Descriptores de archivos	465
5.6.8	Candados de archivos	467
5.6.9	Conductos y archivos especiales	467
5.6.10	Un ejemplo: La llamada al sistema READ	469
5.7	IMPLEMENTACIÓN DEL SISTEMA DE ARCHIVOS MINIX	470
5.7.1	Archivos de encabezado y estructuras de datos globales	470
5.7.2	Administración de tablas	474
5.7.3	El programa principal	482
5.7.4	Operaciones con archivos individuales	485
5.7.5	Directorios y rutas	493
5.7.6	Otras llamadas al sistema	498
5.7.7	La interfaz con dispositivos de E/S	501
5.7.8	Utilerías generales	503
5.8	RESUMEN	503

6 LISTA DE LECTURAS Y BIBLIOGRAFÍA **507**

6.1	SUGERENCIAS DE LECTURAS ADICIONALES	507
6.1.1	Introducción y trabajos generales	507
6.1.2	Procesos	509
6.1.3	Entrada/salida	510
6.1.4	Administración de memoria	511
6.1.5	Sistemas de archivos	511
6.2	BIBLIOGRAFÍA ALFABETIZADA	512

APÉNDICES

A	EL CÓDIGO FUENTE DE MINIX	521
B	ÍNDICE DE ARCHIVOS	905
C	ÍNDICE DE SÍMBOLOS	909
	ÍNDICE	925

PREFACIO

Gran parte de los libros sobre sistemas operativos se centran mucho en la teoría y poco en la práctica. El presente libro intenta ofrecer un mayor equilibrio entre una y otra; trata todos los principios fundamentales con gran detalle, entre ellos, los procesos, la comunicación entre procesos, semáforos, monitores, transferencia de mensajes, algoritmos de planificación, entrada/salida, bloqueos mutuos, controladores de dispositivos, administración de memoria, algoritmos de paginación, diseño de sistemas de archivos, seguridad y mecanismos de protección, pero también examina un sistema específico —MINIX, un sistema operativo compatible con UNIX— detallada mente, e incluso proporciona un listado completo del código fuente para estudiarlo. Esta organización permite al lector no sólo aprender los principios, sino también ver cómo se aplican en un sistema operativo real.

Cuando apareció la primera edición de este libro en 1987, causó una especie de revolución en la forma de impartir los cursos de sistemas operativos. Hasta entonces, la mayor parte de los cursos sólo abordaban la teoría. Con la aparición de MINIX, muchas escuelas comenzaron a ofrecer cursos de laboratorio en los que los estudiantes examinaban un sistema operativo real para ver cómo funcionaba internamente. Consideramos que esta tendencia es altamente favorable y esperamos que esta segunda edición la fortalezca.

En sus primeros 10 años, MINIX ha sufrido muchos cambios. El código original se diseñó para una IBM PC de 256 K basada en 8088 con dos unidades de disquete y sin disco duro; además, se basaba en la Versión 7 de UNIX. Con el paso del tiempo, MINIX evolucionó en muchas direcciones. Por ejemplo, la versión actual se ejecuta en máquinas desde la PC original (en modo real de 16 bits) hasta las Pentium grandes con discos duros de gigabytes (en modo protegido de 32 bits). Además, MINIX se basa ahora en el estándar internacional POSIX (IEEE 1003.1 e ISO 9945-1) en

lugar de la Versión 7. Por último, se agregaron muchas características, tal vez demasiadas en nuestra opinión, pero no suficientes según otras personas; esto dio pie a la creación de LINUX. Además, MINIX se llevó a muchas otras plataformas, incluidas Macintosh, Amiga, Atari y SPARC. Este libro sólo trata MINIX 2.0, que hasta ahora sólo se ejecuta en computadoras con una CPU 80x86, en sistemas que pueden emular tal CPU, o en SPARC.

Esta segunda edición del libro tiene muchos cambios en todas sus partes. Casi todo el material sobre principios ha sido revisado, y se ha agregado una cantidad considerable de material nuevo. Sin embargo, el cambio principal radica en la explicación del nuevo MINIX basado en POSIX, y la inclusión del nuevo código en este libro. Otra cosa nueva es la inclusión de un CD-ROM en cada libro con el código fuente completo de MINIX junto con instrucciones para instalar MINIX en una PC (véase el archivo README.TXT en el directorio principal del CD-ROM).

La configuración de MINIX en una PC 80x86, sea para uso individual o en el laboratorio, es sencilla. Primero se debe crear una partición de disco de, por lo menos, 30 MB para él; luego puede instalarse MINIX siguiendo las instrucciones del archivo README.TXT del CD-ROM. Si desea imprimir el archivo README.TXT en una PC, primero inicie MS-DOS, si no se está ejecutando ya (desde WINDOWS, haga clic en el icono MS-DOS), y luego teclee

```
copy readme.txt prn
```

para producir el listado. El archivo también puede examinarse en edit, wordpad, notepad o cualquier otro editor de texto que pueda manejar texto ASCII simple.

Para las escuelas (o individuos) que no cuentan con una PC, ahora hay otras dos opciones. Se incluyen dos simuladores en el CD-ROM. Uno, escrito por Paul Ashton, trabaja en SPARC, ejecutando MINIX como programa de usuario encima de Solaris. En consecuencia, MINIX se compila para producir un binario de SPARC y se ejecuta a toda velocidad. En esta modalidad, MINIX ya no es un sistema operativo, sino un programa de usuario, lo cual hizo necesarios ciertos cambios en el código de bajo nivel.

El otro simulador fue escrito por Kevin P. Lawton de Bochs Software Company. Este simulador interpreta el conjunto de instrucciones 80386 y suficientes aditamentos de E/S como para que MINIX pueda ejecutarse en el simulador. Desde luego, al ejecutarse encima de un intérprete el rendimiento decae, pero los estudiantes pueden realizar la depuración con mucha mayor facilidad. Este simulador tiene la ventaja de que se ejecuta en cualquier computadora que maneja el sistema X Ventana del M.I.T. Si desea mayor información acerca de los simuladores, remítase al CD-ROM,

El desarrollo de MINIX es un proyecto que continúa. El contenido de este libro y del CD-ROM son sólo una instantánea del sistema en el momento de la publicación. Si le interesa el estado actual del sistema, puede ver la página base de MINIX en la World Wide Web, <http://www.cs.vu.nl/~ast/minix.html>, Además, MINIX tiene su propio grupo de noticias USENET: comp.os.minix, al cual pueden suscribirse los lectores para averiguar qué es lo que sucede en el mundo de MINIX. Para quienes cuentan con correo electrónico, pero no con acceso a grupos de noticias, también hay una lista de correo. Escriba a listserv@listserv.nodak.edu con “subscribe minix-l <su nombre completo>” como primera y única línea del cuerpo del mensaje. A vuelta de correo electrónico recibirá mayor información.

Para uso en el aula, se pueden obtener archivos PostScript que contienen todas las ilustraciones del libro, y que son apropiados para crear diapositivas, siguiendo el enlace marcado como "Soft ware aud supplementary material" de <http://wwwcs.vu.nl/>

Hemos tenido la enorme suerte de contar con la ayuda de muchas personas durante la realización de este proyecto. Antes que nada, nos gustaría agradecer a Kees Bot el haber realizado la mayor parte del trabajo necesario para hacer que MINIX se ajustara al estándar, además de haber organizado la distribución. Sin su generosa ayuda, nunca habríamos completado el proyecto. Él mismo escribió grandes trozos del código (p. ej., la E/S de terminal de Posix), depuró otras secciones y reparó numerosos errores que se habían acumulado al pasar los años. Muchas gracias por un trabajo bien hecho.

Bruce Evans, Philip Homburg, Will Rose y Michael Teman han contribuido al desarrollo de MINIX durante muchos años, Cientos de personas más han contribuido con MINIX a través del grupo de noticias; son tantos y sus contribuciones han sido tan variadas que no podríamos siquiera comenzar a mencionarlos a todos aquí, así que lo mejor que podemos hacer es expresar un agradecimiento generalizado a todos ellos.

Varias personas leyeron partes del manuscrito e hicieron sugerencias. Nos gustaría dar las gracias especialmente a John Casey, Dale Grit y Frans Kaashoek.

Varios estudiantes de la Vrije Universiteit probaron la versión beta del CD-ROM. Ellos fueron: Ahmed Batou, Goran Dokic, Peter Gijzel, Thomer Gil, Dennis Grimbergen, Roderick Groesbeek, Wouter Haring, Guido Kollerie, Mark Lassche, Raymond Ris, Frans ter Borg, Alex van Ballegooy, Ries van der Velden, Alexander Wels y Thomas Zeeman. Nos gustaría agradecer a todos ellos lo cuidadoso de su trabajo y lo detallado de sus informes.

ASW desea agradecer, además, a varios de sus antiguos estudiantes, sobre todo a Peter W. Young de Hampshire College, y a María Isabel Sánchez y William Puddy Vargas de la Universidad Nacional Autónoma de Nicaragua el papel que su interés por MINIX tuvo en el apoyo a su esfuerzo.

Por último, nos gustaría dar las gracias a nuestras familias. Suzanne ya ha pasado por esto diez veces. A Barbara le ha tocado nueve veces. Marvin lo ha sufrido ocho veces, e incluso el pequeño Bram ha pasado por esto cuatro veces. Ya casi se está volviendo rutina, pero el amor y el apoyo siguen siendo muy apreciados. (ast)

En cuanto a la Barbara de Al, ésta es la primera vez que pasa por la experiencia, y no hubiera sido posible lograrlo sin su apoyo, paciencia y buen humor. Gordon tuvo la suerte de estar casi todo el tiempo lejos, en su universidad, pero es un deleite tener un hijo que entiende y se interesa por las mismas cosas que me fascinan. (asw)

Andrew S. Tanenbaum
Albert S. Woodhull

ACERCA DE LOS AUTORES

Andrew S. Tanenbaum tiene el título de S.B. por el M.I.T. y un Ph.D. por la University of California en Berkeley. Es actualmente profesor de ciencias de la computación en la Vrije Universiteit en Amsterdam, Países Bajos, donde encabeza el Grupo de Sistemas de Computación; también es decano de la Advanced School for Computing and Imaging, una escuela de posgrado interuniversitaria que realiza investigaciones sobre sistemas paralelos, distribuidos y de producción de imágenes avanzados. No obstante, está haciendo grandes esfuerzos por no convertirse en un burócrata.

En el pasado, el profesor Tanenbaum ha realizado investigaciones sobre compiladores, sistemas operativos, redes y sistemas distribuidos de área local. Sus investigaciones actuales se centran principalmente en el diseño de sistemas distribuidos de área extensa que atienden a millones de usuarios. Estos proyectos de investigación han dado pie a más de 70 artículos arbitrados en revistas y ponencias en conferencias, y a cinco libros.

El profesor Tanenbaum también ha producido un volumen considerable de software. Fue el arquitecto principal del Amsterdam Compiler Kit, una caja de herramientas de amplio uso para escribir compiladores portátiles, así como de MINIX. Junto con sus estudiantes de doctorado y programadores, ayudó a diseñar Amoeba, un sistema operativo distribuido de alto rendimiento basado en microkernel. MINIX y Amoeba ya están disponibles gratuitamente para educación e investigación a través de Internet.

Los estudiantes de doctorado del profesor Tanenbaum han cosechado grandes triunfos desde que obtuvieron sus grados. Él está muy orgulloso de ellos; en este sentido se asemeja a una mamá gallina.

El profesor Tanenbaum es miembro de la ACM, miembro senior del IEEE, miembro de la Academia Real de Artes y Ciencias de los Países Bajos, ganador del Karl. V. Karlstrom Outstanding Educator Award de la ACM en 1994 y del ACM/SIGCSE Award for Outstanding Contributions to Computer Science Education; además, aparece en Who 's Who in the World. Su página base en la World Wide Web puede encontrarse en el URL <http://wwwcs.vu.nl/~ast/>.

Albert S. Woodhull tiene el título de S.B. por el M.I.T. y un Ph.D. de la University of Washington. Ingresó en el M.I.T. con la intención de convertirse en ingeniero electricista, pero egresó como biólogo. Ha estado asociado a la School of Natural Science del Hampshire College en Amherst, Massachusetts, desde 1973. Como biólogo que utiliza instrumentación electrónica, comenzó a trabajar con microcomputadoras cuando se hicieron cosa común. Sus cursos de instrucción para estudiantes en ciencias evolucionaron hasta convertirse en cursos sobre interfaces con computadoras y programación en tiempo real.

El doctor Woodhull siempre ha tenido un gran interés por la enseñanza y el papel de la ciencia y la tecnología en el desarrollo. Antes de ingresar en su posgrado, impartió cursos de ciencias en educación media durante dos años en Nigeria. En fechas más recientes pasó varios años sabáticos impartiendo ciencias de la computación en la Universidad Nacional de Ingeniería de Nicaragua y en la Universidad Nacional Autónoma de Nicaragua.

El doctor Woodhull está interesado en las computadoras como sistemas electrónicos, y en las interacciones de las computadoras con otros sistemas electrónicos, y le gusta sobre todo enseñar en las áreas de arquitectura de computadoras, programación en lenguaje ensamblador, sistemas operativos y comunicaciones de computadoras. También ha trabajado como consultor en el desarrollo de instrumentación electrónica y el software relacionado.

El doctor Woodhull tiene también muchos intereses no académicos, que incluyen varios de portes al aire libre, radioaficionados y lectura. Disfruta viajando y tratando de hacerse entender en otros idiomas aparte del inglés que es su lengua materna. Su página base de la World Wide Web está situada en un sistema que ejeduta MINIX, en el URL <http://minix1.hampshire.edu/aswt>

El Doctor Albert S. Woodhull colaboró de manera entusiasta en la revisión de la traducción al español, con la ayuda de las siguientes personas: Glenda Barrios Aguirre, Universidad Nacional de Ingeniería, Managua, Nicaragua; Juan Díaz Cuadra, Universidad Nacional Autónoma de Nicaragua, Managua, Nicaragua; Shara Iskakova Baltodano, Universidad Nacional Autónoma de Nicaragua, Managua, Nicaragua; Esmilda Sáenz Artola, Universidad Nacional de Ingeniería, Managua, Nicaragua.

1

INTRODUCCIÓN

Sin su software, la computadora es básicamente un montón de metal inútil. Con su software, una computadora puede almacenar, procesar y recuperar información; exhibir documentos multimedia; realizar búsquedas en Internet; y realizar muchas otras actividades valiosas para justificar su existencia. El software de computadora puede dividirse a grandes rasgos en dos tipos: programas de sistema, que controlan la operación de la computadora misma, y programas de aplicación, que realizan las tareas reales que el usuario desea. El programa de sistema más fundamental es el **sistema operativo**, que controla todos los recursos de la computadora y establece la base sobre la que pueden escribirse los programas de aplicación.

Un sistema de computadora moderno consiste en uno o más procesadores, memoria principal (también conocida como RAM, memoria de acceso aleatorio), discos, impresoras, interfaces de red y otros dispositivos de entrada/salida. A todas luces, se trata de un sistema complejo. Escribir programas que sigan la pista a todos estos componentes y los usen correctamente, ya no digamos óptimamente, es una tarea en extremo difícil. Si todos los programadores tuvieran que ocuparse de cómo trabajan las unidades de disco, y de las docenas de cosas que pueden fallar al leer un bloque de disco, es poco probable que pudieran escribirse muchos programas.

Hace muchos años se hizo muy evidente que debía encontrarse alguna forma de proteger a los programadores de la complejidad del hardware. La solución que ha evolucionado gradualmente consiste en poner una capa de software encima del hardware solo, que se encargue de administrar todas las partes del sistema y presente al usuario una interfaz o **máquina virtual** que sea más

fácil de entender y programar. Esta capa de software es el sistema operativo, y constituye el tema de este libro.

La situación se muestra en la Fig. 1-1. En la parte inferior está el hardware que, en muchos casos, también se compone de dos o más capas. La capa más baja contiene los dispositivos físicos, que consisten en chips de circuitos integrados, alambres, fuentes de potencia, tubos de rayos catódicos y otros aparatos físicos similares. La forma en que éstos se construyen y sus principios de funcionamiento pertenecen al campo del ingeniero electricista.

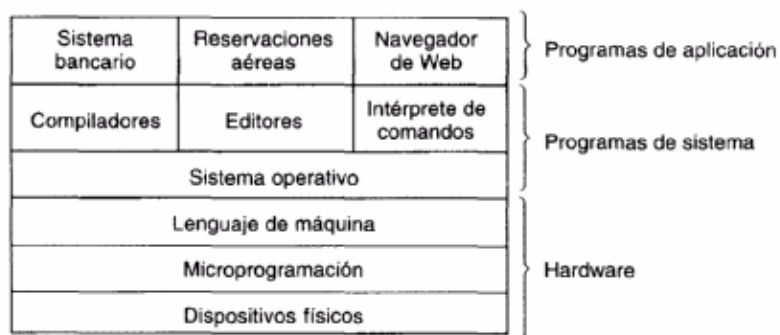


Figura 1-1. Un sistema de computadora consiste en hardware, programas de sistema y programas de aplicación.

A continuación (en algunas máquinas) viene una capa de software primitivo que controla directamente estos dispositivos y ofrece una interfaz más aseada a la siguiente capa. Este software, llamado **microprograma**, suele estar almacenado en memoria de sólo lectura. En realidad es un intérprete, que obtiene las instrucciones de lenguaje de máquina como ADD, MOVE y JUMP y las ejecuta en una serie de pasos pequeños. Por ejemplo, para ejecutar una instrucción ADD (sumar), el microprograma debe determinar dónde se encuentran los números que se van a sumar, obtenerlos, sumarlos y almacenar el resultado en algún lugar. El conjunto de instrucciones que el microprograma interpreta define el **lenguaje de máquina**, que no es realmente parte de la máquina física, aunque los fabricantes de computadoras siempre lo describen en sus manuales como tal, de modo que muchas personas piensan en él como si fuera la “máquina” real.

Algunas computadoras, llamadas **RISC (computadoras con conjunto de instrucciones reducido)**, no tienen un nivel de microprogramación. En estas máquinas, el hardware ejecuta las instrucciones de lenguaje de máquina directamente. Por ejemplo, el Motorola 680x0 tiene un nivel de microprogramación, pero el IBM PowerPC no.

El lenguaje de máquina por lo regular cuenta con entre 50 y 300 instrucciones, la mayor parte de ellas para trasladar datos dentro de la máquina, realizar operaciones aritméticas y comparar valores. En esta capa, los dispositivos de entrada/salida se controlan cargando valores **registros de dispositivo** especiales. Por ejemplo, para un disco que ejecuta una lectura, sus registros se cargan con los valores de dirección del disco, dirección de memoria principal, conteo de bytes y dirección de acceso (READ o WRITE). En la práctica se requieren muchos parámetros más, y el

estado devuelto por la unidad de disco después de una operación es muy complejo. Además, la temporización desempeña un papel importante en la programación de muchos dispositivos de E/S.

Una función importante del sistema operativo es ocultar toda esta complejidad y ofrecer al programador un conjunto de instrucciones más cómodo con el que pueda trabajar. Por ejemplo, LEER BLOQUE DE ARCHIVO es conceptualmente más sencillo que tener que preocuparse por los detalles de mover cabezas de disco, esperar que se estabilicen, etcétera.

Encima del sistema operativo está el resto del software de sistema. Aquí encontramos el intérprete de comandos (shell), sistemas de ventanas, compiladores, editores y otros programas similares independientes de la aplicación. Es importante darse cuenta de que estos programas definitivamente no forman parte del sistema operativo, a pesar de que casi siempre son provistos por el fabricante de la computadora. Éste es un punto crucial, aunque sutil. El sistema operativo es la porción del software que se ejecuta en **modo kernel o modo supervisor**, y está protegido por el hardware contra la intervención del usuario (olvidándonos por el momento de algunos de los microprocesadores más viejos que no tienen ninguna protección de hardware). Los compiladores y editores se ejecutan en **modo de usuario**. Si a un usuario no le gusta un compilador en particular, él⁺ está en libertad de escribir el suyo propio si lo desea; no está en libertad de escribir su propio manejador de interrupciones del disco, que forma parte del sistema operativo y normalmente está protegido por el hardware contra los intentos de los usuarios por modificarlo.

Por último, encima de los programas de sistema vienen los programas de aplicación. Los usuarios compran o escriben estos programas para resolver sus problemas particulares, como procesamiento de textos, hojas de cálculo, cálculos de ingeniería o juegos.

¿QUÉ ES UN SISTEMA OPERATIVO?

La mayoría de los usuarios de computadora han tenido algo de experiencia con un sistema operativo, pero no es fácil precisar con exactitud qué es un sistema operativo. Parte del problema consiste en que el sistema operativo realiza dos funciones que básicamente no están relacionadas entre sí y, dependiendo de a quién le preguntemos, por lo general se nos habla principalmente de una función o de la otra. Veamos ahora las dos.

1.1.1 El sistema operativo como máquina extendida

Como ya dijimos, la **arquitectura** (conjunto de instrucciones, organización de memoria, E/S y estructura de buses) de la mayor parte de las computadoras en el nivel de lenguaje de máquina es primitiva y difícil de programar, sobre todo para entrada/salida. A fin de hacer más concreto este punto, veamos brevemente cómo se realiza la E/S de disco flexible usando el chip controlador NEC PD765 (o su equivalente), utilizado por la mayor parte de las computadoras personales. (En todo este libro usaremos indistintamente los términos “disco flexible” y “disquete”.) El PD765

⁺“Él” debe leerse “él o ella” a lo largo de todo el libro.

tiene 16 comandos, cada uno de los cuales se especifica cargando entre 1 y 9 bytes en un registro de dispositivo. Estos comandos sirven para leer y escribir datos, mover el brazo del disco y formatear pistas, así como para inicializar, detectar, restablecer y recalibrar el controlador y las unidades de disco.

Los comandos más básicos son READ y WRITE, cada uno de los cuales requiere 13 parámetros empacados en 9 bytes. Estos parámetros especifican cosas tales como la dirección del bloque de disco que se va a leer, el número de sectores por pista, el modo de grabación empleado en el medio físico, el espaciado de la brecha entre sectores y qué hacer con una marca de “dirección de datos eliminada”. Si usted no entiende a qué nos referimos, no se preocupe; de eso se trata precisamente: es algo muy esotérico. Cuando se completa la operación, el chip controlador devuelve 23 campos de estado y error empacados en 7 bytes. Por si esto no fuera suficiente, el programador del disco flexible también debe tener presente en todo momento si el motor está encendido o apagado. Si el motor está apagado, debe encenderse (con un retardo de arranque largo) antes de que puedan leerse o escribirse datos. Empero, el motor no puede dejarse encendido demasiado tiempo, pues el disco flexible se desgastaría. Por tanto, el programador debe encontrar un equilibrio entre los retardos de arranque largos y el desgaste de los discos flexibles (y la pérdida de los datos que contienen).

Sin entrar en los verdaderos detalles, debe quedar claro que el programador ordinario seguramente no quiere intervenir de manera demasiado íntima en la programación de los discos flexibles (o de los duros, que son igualmente complejos, y muy distintos). En vez de ello, lo que el programador quiere es manejar una abstracción sencilla, de alto nivel. En el caso de los discos, una abstracción típica sería que el disco contiene una colección de archivos con nombre. Cada archivo puede abrirse para lectura o escritura, leerse o escribirse, y por último cerrarse. Los detalles de si la grabación debe usar o no modulación de frecuencia modificada y cuál es la situación actual del motor no deberán aparecer en la abstracción presentada al usuario.

El programa que oculta la verdad acerca del hardware y presenta al programador una vista sencilla y bonita de archivos con nombre que pueden leerse y escribirse es, por supuesto, el sistema operativo. Así como el sistema operativo aísla al programador del hardware del disco y presenta una interfaz sencilla orientada a archivos, también oculta muchos asuntos desagradables referentes a interrupciones, temporizadores, administración de memoria y otras funciones de bajo nivel. En cada caso, la abstracción que el sistema operativo ofrece es más sencilla y fácil de usar que el hardware subyacente.

En esta vista, la función del sistema operativo es presentar al usuario el equivalente de una **máquina extendida o máquina virtual** que es más fácil de programar que el hardware subyacente. La forma en que el sistema operativo logra este objetivo es una historia larga, que estudiaremos con detalle a lo largo del libro.

1.1.2 El sistema operativo como administrador de recursos

El concepto del sistema operativo como algo cuya función primordial es ofrecer a los usuarios una Interfaz cómoda es una visión descendente. Una visión ascendente alternativa postula que el

sistema operativo está ahí para administrar todos los componentes de un sistema complejo. Las computadoras modernas constan de procesadores, memorias, temporizadores, discos, ratones, interfaces con redes, impresoras láser y una gran variedad de otros dispositivos. En la visión alternativa, la misión del sistema operativo es asegurar un reparto ordenado y controlado de los procesadores, memorias y dispositivos de E/S entre los diferentes programas que compiten por ellos.

Imagine lo que sucedería si tres programas que se ejecutan en alguna computadora trataran de imprimir sus salidas simultáneamente en la misma impresora. Las primeras líneas del listado podrían ser del programa 1, las siguientes del programa 2, luego algunas del programa 3, y así sucesivamente. El resultado sería un caos. El sistema operativo puede poner orden en el caos potencial almacenando temporalmente en el disco todas las salidas destinadas para la impresora. Cuando un programa haya terminado, el sistema operativo podrá copiar su salida del archivo de disco donde se almacenó a la impresora, mientras que el otro programa puede continuar generando salidas, ajeno al hecho de que dichas salidas no están yendo directamente a la impresora (todavía).

Cuando una computadora (o red) tiene múltiples usuarios, la necesidad de administrar y proteger la memoria, los dispositivos de E/S y demás recursos es aún mayor, ya que de otra manera los usuarios podrían interferirse. Además, es frecuente que los usuarios tengan que compartir no sólo hardware, sino también información (archivos, bases de datos, etc.). En pocas palabras, esta es la visión del sistema operativo sostiene que su tarea primordial es seguir la pista de quién está usando cuál recurso, atender solicitudes de recursos, contabilizar la utilización y mediar entre solicitudes en conflicto provenientes de diferentes programas y usuarios.

1.2 HISTORIA DE LOS SISTEMAS OPERATIVOS

Los sistemas operativos han estado evolucionando durante muchos años. En las siguientes secciones examinaremos brevemente este desarrollo. Dado que, históricamente, los sistemas operativos han estado de manera muy estrecha vinculados con la arquitectura de las computadoras en las que se ejecutan, estudiaremos las sucesivas generaciones de computadoras para ver qué clase de sistemas operativos usaban. Esta correspondencia entre las generaciones de sistemas operativos y de computadoras es algo burda, pero establece un poco de estructura que de otra forma sería inexistente.

La primera computadora digital verdadera fue diseñada por el matemático inglés Charles Babbage (1792-1871). Aunque Babbage invirtió la mayor parte de su vida y su fortuna tratando de construir su “máquina analítica”, nunca logró que funcionara correctamente porque era totalmente mecánica, y la tecnología de su época no podía producir las ruedas, engranes y levas con la elevada precisión que él requería. Huelga decir que la máquina analítica no contaba con un sistema operativo.

Como acotación histórica interesante, diremos que Babbage se dio cuenta de que necesitaría software para su máquina analítica, así que contrató a una joven mujer, Ada Lovelace, hija del famoso poeta británico, Lord Byron, como la primera programadora de la historia. El lenguaje de programación Ada® recibió su nombre en honor a ella.

1.2.1 La primera generación (1945-55): Tubos de vacío y tableros de conmutación

Después del fracaso de los trabajos de Babbage, fueron pocos los avances que se lograron en la construcción de computadoras digitales hasta la Segunda Guerra Mundial. A mediados de la década de 1940, Howard Aiken en Harvard, John von Neumann en el Institute for Advanced Study en Princeton, J. Presper Eckert y William Mauchley en la University of Pennsylvania y Konrad Zuse en Alemania, entre otros, lograron construir máquinas calculadoras usando tubos de vacío. Estas máquinas eran enormes, y ocupaban cuartos enteros con decenas de miles de tubos de vacío, pero eran mucho más lentas que incluso las computadoras personales más baratas de la actualidad.

En esos primeros días, un solo grupo de personas diseñaba, construía, programaba, operaba y mantenía a cada máquina. Toda la programación se realizaba en lenguaje de máquina absoluto, a menudo alambrando tableros de conmutación para controlar las funciones básicas de la máquina. No existían los lenguajes de programación (ni siquiera los de ensamblador). Nadie había oído hablar de los sistemas operativos. La forma de operación usual consistía en que el programador se anotaba para recibir un bloque de tiempo en la hoja de reservaciones colgada en la pared, luego bajaba al cuarto de la máquina, insertaba su tablero de conmutación en la computadora, y pasaba las siguientes horas con la esperanza de que ninguno de los cerca de 20000 tubos de vacío se quemara durante la sesión. Prácticamente todos los problemas eran cálculos numéricos directos, como la producción de tablas de senos y cosenos.

A principios de la década de 1950, la rutina había mejorado un poco con la introducción de las tarjetas perforadas. Ahora era posible escribir programas en tarjetas e introducirlas para ser leídas, en lugar de usar tableros de conmutación; por lo demás, el procedimiento era el mismo.

1.2.2 La segunda generación (1955-65): Transistores y sistemas por lote

La introducción del transistor a mediados de la década de 1950 alteró el panorama radicalmente. Las computadoras se hicieron lo bastante confiables como para poderse fabricar y vender a clientes comerciales con la expectativa de que seguirían funcionando el tiempo suficiente para realizar algo de trabajo útil. Por primera vez, había una separación clara entre diseñadores, constructores, operadores, programadores y personal de mantenimiento.

Estas máquinas se encerraban en cuartos de computadora con acondicionamiento de aire especial, con equipos de operadores profesionales para operarias. Sólo las grandes empresas, o las principales dependencias del gobierno o universidades, podían solventar el costo de muchos millones de dólares. Para ejecutar un **trabajo** (es decir, un programa o serie de programas), un programador escribía primero el programa en papel (en FORTRAN o ensamblador) y luego lo perforaba en tarjetas. Después, llevaba el grupo de tarjetas al cuarto de entrada y lo entregaba a uno de los operadores.

Cuando la computadora terminaba el trabajo que estaba ejecutando en ese momento, un operador acudía a la impresora, separaba la salida impresa y la llevaba al cuarto de salida donde el programador podía recogerla después. Luego, el operador tomaba uno de los grupos de tarjetas

traídos del cuarto de entrada y lo introducía en el lector. Si se requería el compilador de FORTRAN, el operador tenía que traerlo de un archivero e introducirlo en el lector. Gran parte del tiempo de computadora se desperdiciaba mientras los operadores iban de un lugar a otro, en el cuarto de la máquina.

Dado el alto costo del equipo, no es sorprendente que la gente pronto buscara formas de reducir el desperdicio de tiempo. La solución que se adoptó generalmente fue el **sistema por lotes**. El principio de este modo de operación consistía en juntar una serie de trabajos en el cuarto de entrada, leerlos y grabarlos en una cinta magnética usando una computadora pequeña y (relativamente) económica, como una IBM 1401, que era muy buena para leer tarjetas, copiar cintas e imprimir salidas, pero no para realizar cálculos numéricos. Otras máquinas, mucho más costosas, como la IBM 7094, se usaban para la computación propiamente dicha. Esta situación se muestra en la Fig. 1-2.

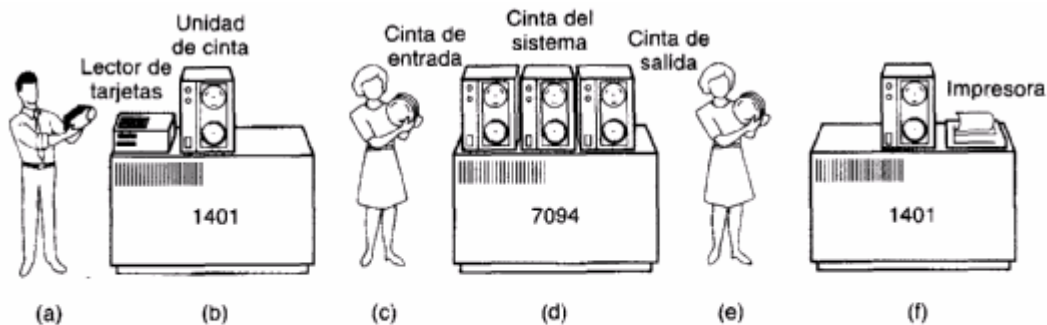


Figura 1-2. Uno de los primeros sistemas por lotes. (a) Los programadores traen tarjetas a la 1401. (b) La 1401 lee lotes de trabajos y los graba en cinta. (c) El operador lleva la cinta de entrada a la 7094. (d) La 7094 realiza la computación. (e) El operador lleva la cinta de salida a la 1401. (f) La 1401 imprime la salida.

Después de cerca de una hora de reunir un lote de trabajos, la cinta se rebobinaba y se llevaba al cuarto de la máquina, donde se montaba en una unidad de cinta. El operador cargaba entonces un programa especial (el antepasado del sistema operativo actual), que leía el primer trabajo de la cinta y lo ejecutaba. La salida se escribía en una segunda cinta, en lugar de imprimirse. Cada vez que terminaba un trabajo, el sistema operativo leía automáticamente el siguiente trabajo de la cinta y comenzaba a ejecutarlo. Una vez que estaba listo todo el lote, el operador desmontaba las cintas de entrada y salida, montaba la cinta de entrada del siguiente lote, y llevaba la cinta de salida a una 1401 para la impresión **fuera de línea** (o sea, no conectada a la computadora principal).

La estructura de un trabajo de entrada típico se muestra en la Fig. 1-3. El trabajo comenzaba con una tarjeta \$JOB, que especificaba el tiempo de ejecución máximo en minutos, el número de cuenta al que se debía cobrar el trabajo, y el nombre del programador. Luego venía una tarjeta \$FORTRAN, que ordenaba al sistema operativo leer el compilador de FORTRAN de la cinta de sistema. Esta tarjeta iba seguida del programa por compilar y por una tarjeta \$LOAD, que ordenaba al sistema operativo cargar el programa objeto recién compilado. (Los programas compilados a menudo se escribían en cintas temporales y tenían que cargarse explícitamente.) Luego venía la

tarjeta \$RUN, que ordenaba al sistema operativo ejecutar el programa con los datos que le seguían. Por último, la tarjeta \$END marcaba el final del trabajo. Estas tarjetas de control primitivas eran los precursores de los lenguajes de control de trabajos e intérpretes de comandos modernos.

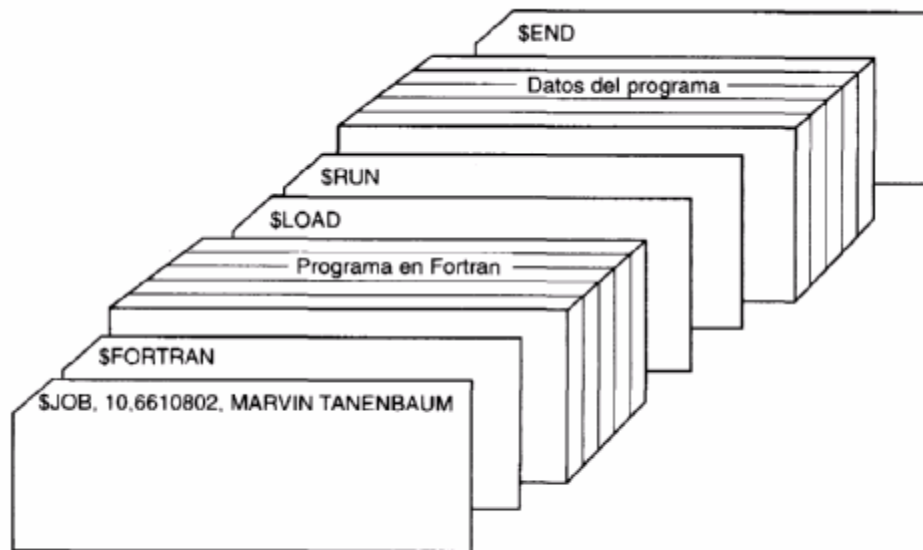


Figura 1-3. Estructura de un trabajo FMS típico.

Las computadoras grandes de la segunda generación se usaban primordialmente para cálculos científicos y de ingeniería, como la resolución de ecuaciones diferenciales parciales. Estas máquinas generalmente se programaban en FORTRAN y lenguaje ensamblador. Los sistemas operativos típicos eran FMS (el Fortran Monitor System) e IBSYS, el sistema operativo de IBM para la 7094.

1.2.3 La tercera generación (1965-1980): Circuitos integrados y multiprogramación

A principios de la década de 1960, la mayoría de los fabricantes de computadoras tenían dos líneas de producto distintas y totalmente incompatibles. Por un lado estaban las computadoras científicas a gran escala, orientadas hacia las palabras, como la 7094, que se usaban para cálculos numéricos en ciencias e ingeniería. Por el otro, estaban las computadoras comerciales orientadas hacia los caracteres, como la 1401, que los bancos y las compañías de seguros utilizaban ampliamente para ordenar e imprimir desde cinta.

La creación y mantenimiento de dos líneas de producto totalmente distintas era una situación costosa para los fabricantes. Además, muchos clientes de computadoras nuevas necesitaban inicialmente una máquina pequeña que más adelante les resultaba insuficiente, de modo que querían una máquina más grande que ejecutara todos sus viejos programas, pero más rápidamente.

IBM trató de resolver simultáneamente ambos problemas introduciendo la System/360. La 360 era una serie de máquinas de software compatible que iban desde tamaños comparables a la 1401 hasta computadoras mucho más potentes que la 7094. Las máquinas diferían sólo en el precio y el rendimiento (memoria máxima, velocidad del procesador, número de dispositivos de E/S permitidos, etc.). Puesto que todas las máquinas tenían la misma arquitectura y conjunto de instrucciones, los programas escritos para una máquina podían ejecutarse en todas las demás, al menos en teoría. Además, la 360 estaba diseñada para manejar computación tanto científica como comercial. Así, una sola familia de máquinas podía satisfacer las necesidades de todos los clientes. En años subsecuentes IBM produjo sucesoras comparables a la línea 360, usando tecnología más moderna, conocidas como series 370, 4300, 3080 y 3090.

La 360 fue la primera línea importante de computadoras en usar (a pequeña escala) circuitos integrados (IC), ofreciendo así una ventaja de precio/rendimiento considerable respecto a las máquinas de la segunda generación, que se armaban con transistores individuales. Esta línea fue un éxito inmediato, y la idea de una familia de computadoras compatibles pronto fue adoptada por todos los demás fabricantes importantes. Los descendientes de estas máquinas todavía se emplean en uno que otro centro de cómputo en la actualidad, pero su uso está en rápido declive.

La gran ventaja de la idea de “una familia” fue también su gran debilidad. La intención era que todo el software, incluido el sistema operativo, funcionara en todos los modelos. El software tenía que funcionar en sistemas pequeños, que en muchos casos simplemente sustituían a la 1401 para copiar tarjetas en cinta, y en sistemas muy grandes, que con frecuencia sustituían a las 7094 para realizar pronósticos del tiempo y otros trabajos de computación pesada. El software tenía que ser bueno en sistemas con pocos y con muchos periféricos; tenía que funcionar en entornos comerciales y científicos y, sobre todo, tenía que ser eficiente para todos estos usos distintos.

Era imposible que IBM (o alguien más) pudiera escribir un programa que satisficiera todos esos requisitos opuestos. El resultado fue un sistema operativo enorme, extraordinariamente complejo, tal vez dos o tres órdenes de magnitud mayor que FMS. Este sistema consistía en millones de líneas de lenguaje ensamblador escrito por miles de programadores, y contenía miles y miles de errores, requiriéndose un flujo continuo de nuevas versiones en un intento por corregirlos. Cada versión nueva corregía algunos errores e introducía otros nuevos, de modo que es probable que el número de errores se mantuviera constante con el tiempo.

Uno de los diseñadores de OS/360, Fred Brooks, escribió después un ingenioso e incisivo libro (Brooks, 1975) describiendo sus experiencias con el OS/360. Aunque sería imposible resumir aquí ese libro, baste con decir que la portada muestra una manada de bestias prehistóricas atascadas en un foso de brea. La portada del libro de Silberschatz y Galvin (1994) es una alusión similar.

A pesar de su enorme tamaño y de sus problemas, OS/360 y los sistemas operativos de tercera generación parecidos a él producidos por otros fabricantes de computadoras lograron satisfacer a sus clientes en un grado razonable, y también popularizaron varias técnicas clave que no existían en los sistemas operativos de la segunda generación. Tal vez la más importante de ellas haya sido la **multiprogramación**. En la 7094, cuando el trabajo actual hacía una pausa para esperar que se completara una operación de cinta u otra operación de E/S, la CPU simplemente permanecía ociosa hasta que la E/S terminaba. En los cálculos científicos, con gran uso de CPU, la E/S es poco frecuente, así que el tiempo desperdiciado no es significativo. En el procesamiento de datos

comerciales, el tiempo de espera por E/S puede ser el 80-90% del tiempo total, de modo que algo debía hacerse para evitar que la CPU estuviera ociosa tanto tiempo.

La solución a la que se llegó fue dividir la memoria en varias secciones, con un trabajo distinto en cada partición, como se muestra en la Fig. 1-4. Mientras un trabajo estaba esperando que terminara su E/S, otro podía estar usando la CPU. Si se podían tener en la memoria principal suficientes trabajos a la vez, la CPU podía mantenerse ocupada casi todo el tiempo. Tener múltiples trabajos en la memoria a la vez requiere hardware especial para proteger cada trabajo contra espionaje o por parte de los demás, pero la 360 y otros sistemas de tercera generación estaban equipados con este hardware.

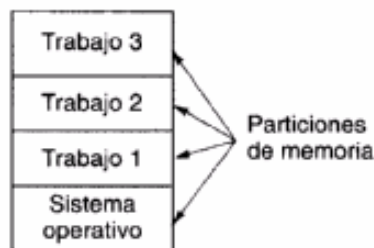


Figura 1-4. Sistema de multiprogramación con tres trabajos en la memoria.

Otra característica importante presente en los sistemas operativos de la tercera generación era la capacidad de leer trabajos de las tarjetas al disco tan pronto como se llevaban al cuarto de computadoras. Luego, cada vez que un trabajo terminaba su ejecución, el sistema operativo podía cargar uno nuevo del disco en la partición que había quedado vacía y ejecutarlo. Esta técnica se llama *spooling* (de “operación simultánea de periféricos en línea”) y también se usaba para la salida. Con *spooling*, las 1401 ya no eran necesarias, y desapareció una buena parte del transporte de cintas.

Aunque los sistemas operativos de la tercera generación se adaptaban bien a cálculos científicos extensos y sesiones masivas de procesamiento de datos comerciales, seguían siendo básicamente sistemas por lotes. Muchos programadores añoraban los días de la primera generación cuando tenían toda la máquina para ellos solos durante unas cuantas horas, lo que les permitía depurar sus programas rápidamente. Con los sistemas de la tercera generación, el tiempo entre la presentación de un trabajo y la obtención de las salidas a menudo era de varias horas, y una sola coma mal colocada podía causar el fracaso de una compilación y que el programador desperdiciara medio día.

Este deseo de respuesta rápida preparó el camino para el **tiempo compartido**, una variante de la multiprogramación, en la que cada usuario tiene una terminal en línea. En un sistema de tiempo compartido, si 20 usuarios ingresan en el sistema y 17 de ellos están pensando, hablando o tomando café, la CPU puede asignarse por turno a los tres trabajos que requieren servicio. Puesto que las personas que están depurando programas usualmente emiten comandos cortos (p. ej., compilar un procedimiento de cinco páginas) en vez de largos (p. ej., ordenar un archivo de un

millón de registros), la computadora puede proporcionar servicio rápido interactivo a varios usuarios y tal vez también trabajar con trabajos de lote grandes en segundo plano cuando la CPU está ociosa. Aunque el primer sistema serio de tiempo compartido (CTSS) fue creado en el M.I.T. en una 7094 especialmente modificada (Corbato et al., 1962), el tiempo compartido no se popularizó realmente hasta que se generalizó el uso del hardware de protección necesario durante la tercera generación.

Después del éxito del sistema CTSS, MIT, Bell Labs y General Electric (por ese entonces un fabricante importante de computadoras) decidieron emprender el desarrollo de un “servicio de computadora” una máquina que diera apoyo a cientos de usuarios de tiempo compartido simultáneos. Su modelo fue el sistema de distribución de electricidad: cuando usted necesita potencia eléctrica, simplemente enchufa una clavija en la pared y, dentro de límites razonables, obtendrá tanta electricidad como necesite. Los diseñadores de este sistema, llamado MULTICS (servicio de información y computación multiplexado), contemplaban una enorme máquina que proporcionara potencia de cómputo a todos los usuarios de Boston. La idea de que máquinas mucho más potentes que su GE-645 se vendieran como computadoras personales por unos cuantos miles de dólares sólo 30 años después no era sino ciencia ficción en ese entonces.

Para resumir un poco la historia, MULTICS introdujo muchas ideas seminales en la literatura de computación, pero su construcción fue mucho más difícil de lo que nadie había imaginado. Bell Labs abandonó el proyecto, y General Electric dejó el negocio de las computadoras por completo. Finalmente, MULTICS funcionó lo bastante bien como para usarse en un entorno de producción de MIT y en docenas de otros sitios, pero el concepto de un servicio de computadora se hizo obsoleto al desplomarse los precios de las computadoras. No obstante, MULTICS tuvo una influencia enorme sobre los sistemas subsecuentes; se le describe en (Corbato et al., 1972; Corbato y Vyssotsky, 1965; Daley y Dennis, 1968; Organick, 1972; Saltzer, 1974).

Otro avance importante durante la tercera generación fue el crecimiento fenomenal de las minicomputadoras, comenzando con la DEC PDP-1 en 1961. La PDP-1 sólo tenía 4K de palabras de 18 bits, pero a \$120 000 por máquina (menos del 5% del precio de una 7094), se vendieron como pan caliente. Para ciertos tipos de trabajos no numéricos, la PDP-1 era casi tan rápida como la 7094, e hizo nacer una industria totalmente nueva. A esta máquina pronto siguió una serie de Otras PDP (todas incompatibles, a diferencia de la familia IBM), culminando en la PDP-11.

Uno de los computólogos de Bell Labs que había trabajado en el proyecto MULTICS, Ken Thompson, encontró subsecuentemente una pequeña minicomputadora PDP-7 que nadie estaba usando y se propuso escribir una versión de MULTICS reducida al mínimo, para un solo usuario. Este trabajo posteriormente evolucionó para convertirse en el sistema operativo UNIX®, que se popularizó en el mundo académico, las dependencias del gobierno y muchas compañías.

La historia de UNIX se cuenta en otras obras (p. ej., Salus, 1994). Baste con decir que, dado que casi todo mundo podía obtener el código fuente, diversas organizaciones desarrollaron sus propias versiones (incompatibles), lo que condujo al caos. Con objeto de que fuera posible escribir programas susceptibles de ejecución en cualquier sistema UNIX, el IEEE creó un estándar para UNIX, llamado posix, que casi todas las versiones actuales de UNIX reconocen. POSIX define una interfaz mínima de llamadas al sistema que los sistemas UNIX deben reconocer. De hecho, algunos otros sistemas de programación ya reconocen la interfaz POSIX.

1.2.4 La cuarta generación (1980-presente): Computadoras personales

Con la invención de los circuitos integrados a gran escala (LSI), chips que contienen miles de transistores en un cm^2 de silicio, nació la era de la computadora personal. En términos de arquitectura, las computadoras personales no eran muy diferentes de las minicomputadoras de la clase PDP-11, pero en términos de precio sí que eran diferentes. Si bien la minicomputadora hacía posible que un departamento de una compañía o universidad tuviera su propia computadora, el chip microprocesador permitía que un solo individuo tuviera su propia computadora personal. Las computadoras personales más potentes empleadas por empresas, universidades e instalaciones del gobierno suelen llamarse **estaciones de trabajo**, pero en realidad sólo son computadoras personales grandes. Por lo regular estas máquinas están interconectadas mediante una red.

La amplia disponibilidad de la potencia de cómputo, sobre todo la potencia de cómputo altamente interactiva casi siempre acompañada por excelentes gráficos, dio pie al crecimiento de una importante industria productora de software para computadoras personales. Una buena parte de este software era **amistoso con el usuario**, lo que significa que estaba dirigido a usuarios que no sólo no sabían nada de computación, sino que además no tenían la mínima intención de aprender. Sin duda, esto representaba un cambio drástico respecto al OS/360, cuyo lenguaje de control de trabajos, JCL, era tan arcano que llegaron a escribirse libros enteros sobre él (p. ej., Cadow, 1970).

Dos sistemas operativos dominaron inicialmente el campo de las computadoras personales y las estaciones de trabajo: MS-DOS de Microsoft y UNIX. MS-DOS se usaba ampliamente en la IBM PC y otras máquinas basadas en la CPU Intel 8088 y sus sucesoras, la 80286, 80386 y 80486 (que en adelante llamaremos la 286, 386 y 486, respectivamente) y más tarde la Pentium y Pentium Pro. Aunque la versión inicial de MS-DOS era relativamente primitiva, versiones subsecuentes han incluido características más avanzadas, muchas de ellas tomadas de UNIX. El sucesor de Microsoft para MS-DOS, WINDOWS, originalmente se ejecutaba encima de MS-DOS (es decir, era más un shell que un verdadero sistema operativo), pero a partir de 1995 se produjo una versión autosuficiente de WINDOWS, WINDOWS 95®, de modo que ya no se necesita MS-DOS para apoyarlo. Otro sistema operativo de Microsoft es WINDOWS NT, que es compatible con WINDOWS 95 en cierto nivel, pero internamente se reescribió desde cero.

El otro competidor importante es UNIX, que domina en las estaciones de trabajo y otras computadoras del extremo alto, como los servidores de red. UNIX es popular sobre todo en máquinas basadas en chips RISC de alto rendimiento. Estas máquinas por lo regular tienen la potencia de cómputo de una minicomputadora, a pesar de estar dedicadas a un solo usuario, por lo que resulta lógico que estén equipadas con un sistema operativo diseñado originalmente para minicomputadoras, a saber, UNIX.

Una tendencia interesante que apareció a mediados de la década de 1980 fue el crecimiento de redes de computadoras personales en las que se ejecutan **sistemas operativos de red o sistemas operativos distribuidos** (Tanenbaum, 1995). En un sistema operativo de red los usuarios están conscientes de la existencia de múltiples computadoras y pueden ingresar en máquinas remotas y copiar archivos de una máquina a otra. Cada máquina ejecuta su propio sistema operativo local y tiene su propio usuario o usuarios locales.

Los sistemas operativos de red no son fundamentalmente distintos de aquellos para un solo procesador. Obviamente, estos sistemas necesitan un controlador de la interfaz con la red y software de bajo nivel para operarlo, así como programas para realizar inicios de sesión remotos y acceso a archivos remotos, pero estas adiciones no alteran la estructura esencial del sistema operativo.

Un sistema operativo distribuido, en cambio, presenta el mismo aspecto a los usuarios que un sistema tradicional de un solo procesador, aunque en realidad se compone de múltiples procesadores. Los usuarios no deben enterarse de en dónde se están ejecutando sus programas o almacenando sus archivos; de todo eso debe encargarse el sistema operativo automáticamente y eficientemente.

Los verdaderos sistemas operativos distribuidos requieren más que la adición de un poco más de código a un sistema operativo uniprocador, porque los sistemas distribuidos y centralizados difieren en aspectos cruciales. Los sistemas distribuidos, por ejemplo, a menudo permiten a las aplicaciones ejecutarse en varios procesadores al mismo tiempo, por lo que requieren algoritmos de planificación de más complejos a fin de optimizar el grado de paralelismo.

En muchos casos, los retardos de comunicación dentro de la red implican que éstos (y otros) algoritmos deban ejecutarse con información incompleta, caduca o incluso incorrecta. Esta situación difiere radicalmente un sistema de un solo procesador en el que el sistema operativo tiene toda la información sobre el estado del sistema.

1.2.5 Historia de MINIX

Cuando UNIX era joven (Versión 6), era fácil conseguir el código fuente, bajo licencia de AT&T, y se estudiaba mucho. John Lions, de la University of New South Wales en Australia, incluso escribió un librito que describía su operación, línea por línea (Lions, 1996). Este librito se usó (con permiso de AT&T) como texto en muchos cursos universitarios de sistemas operativos.

Cuando AT&T liberó la Versión 7, comenzó a darse cuenta de que UNIX era un producto comercial valioso, así que entregó la Versión 7 junto con una licencia que prohibía el estudio del código fuente en cursos, a fin de evitar poner en peligro su situación de secreto comercial. Muchas universidades simplemente abandonaron el estudio de UNIX e impartieron sólo teoría.

Desafortunadamente, cuando sólo se enseña teoría el estudiante adquiere una visión desbalanceada de cómo se ve realmente un sistema operativo. Los temas teóricos que suelen cubrirse con gran detalle en cursos y libros sobre sistemas operativos, como los algoritmos de planificación, en la práctica realmente no son tan importantes. Los temas que en verdad son relevantes, como E/S y sistemas de archivos, generalmente se descuidan porque no hay mucha teoría al respecto.

A fin de remediar esta situación, uno de los autores de este libro (Tanenbaum) decidió escribir un nuevo sistema operativo desde cero que fuera compatible con UNIX desde el punto de vista del usuario, pero completamente distinto en su interior. Al no utilizar ni una sola línea del código de AT&T, este sistema evita las restricciones de la licencia, así que puede usarse en clase o para estudio individual. De esta forma, los lectores pueden disectar un sistema operativo real para ver

qué hay dentro, tal como los estudiantes de biología disectan ranas. El nombre MINIX significa mini-UNIX porque es lo suficientemente pequeño como para poderlo entender a pesar de no ser un gurú.

Además de la ventaja de eliminar los problemas legales, MINIX tiene otra ventaja respecto a UNIX: se escribió una década después de UNIX y tiene una estructura más modular. El sistema de archivos de MINIX, por ejemplo, no forma parte del sistema operativo, sino que se ejecuta como programa de usuario. Otra diferencia es que UNIX se diseñó de modo que fuera eficiente; MINIX se diseñó pensando en que fuera comprensible (hasta donde puede ser comprensible cualquier programa que ocupa cientos de páginas). El código de MINIX, por ejemplo, incluye miles de comentarios.

MINIX se diseñó originalmente de modo que fuera compatible con UNIX Versión 7 (V7). Se usó como modelo esta versión a causa de su sencillez y elegancia. A veces se dice que la Versión 7 no sólo representó una mejora respecto a sus predecesores, sino también respecto a todos sus sucesores. Con la llegada de POSIX, MINIX comenzó a evolucionar hacia el nuevo estándar, al tiempo que mantenía la compatibilidad hacia atrás con los programas existentes. Este tipo de evolución es común en la industria de las computadoras, pues ningún proveedor desea introducir un sistema nuevo que ninguno de sus clientes existentes podrá usar sin grandes convulsiones. La versión de MINIX Sue se describe en este libro se basa en el estándar POSIX (a diferencia de la versión descrita en la primera edición, que se basaba en V7).

Al igual que UNIX, MINIX se escribió en el lenguaje de programación C y se pretendía que fuera fácil transportarlo a diversas computadoras. La implementación inicial fue para la IBM PC, porque esta computadora se usa ampliamente. Subsecuentemente se llevó a las computadoras Atari, Amiga, Macintosh y SPARC. Acorde con la filosofía de que “lo pequeño es hermoso”, MINIX originalmente no requería siquiera un disco duro para ejecutarse, lo que lo ponía al alcance del presupuesto de muchos estudiantes (aunque puede parecer asombroso ahora, a mediados de la década de 1980 cuando MINIX vio por primera vez la luz, los discos duros aún eran una novedad de precio elevado). Al crecer MINIX en funcionalidad y tamaño, llegó al punto en que se hizo necesario un disco duro, pero en concordancia con la filosofía de MINIX basta con una partición de 30 megabytes. En contraste, algunos sistemas UNIX comerciales ahora recomiendan una partición de disco de 200 MB como mínimo indispensable.

Para el usuario medio sentado ante una IBM PC, ejecutar MINIX es similar a ejecutar UNIX. Muchos de los programas básicos, como cat, grep, is, make y el shell están presentes y desempeñan las mismas funciones que sus contrapartes de UNIX. Al igual que el sistema operativo mismo, todos estos programas de utilería fueron reescritos completamente desde cero por el autor, sus estudiantes y algunas otras personas dedicadas.

En todo este libro se usará MINIX como ejemplo. No obstante, casi todo lo que se diga acerca de MINIX, a menos que se refiera al código en sí, también aplica a UNIX. Muchos de estos comentarios también aplican a otros sistemas. Esto debe tenerse siempre presente al leer el libro.

Como acotación, es posible que unas cuantas palabras acerca de LINUX y su relación con MINIX sean de interés para algunos lectores. Poco después de liberarse MINIX, se formó un grupo de noticias de USENET para hablar de él. En pocas semanas, este grupo tenía 40000 suscriptores, la mayor parte de los cuales quería agregar enormes cantidades de nuevas capacidades a MINIX a

fin de hacerlo más grande y mejor (bueno, al menos más grande). Cada día, varios cientos de ellos ofrecían sugerencias, ideas y fragmentos de código. El autor de MINIX resistió con éxito esta arremetida durante varios años, a fin de mantener a MINIX lo suficientemente pequeño y aseado como para que los estudiantes lo entendieran. Gradualmente, la gente comenzó a convencerse de que su posición era inamovible. Finalmente, un estudiante finlandés, Linus Torvalds, decidió escribir un clon de MINIX que pretendía ser un sistema de producción con abundantes capacidades, más que una herramienta educativa. Así fue como nació LINUX.

1.3 CONCEPTOS DE SISTEMAS OPERATIVOS

La interfaz entre el sistema operativo y los programas de usuario está definida por el conjunto de “operaciones extendidas” que el sistema operativo ofrece. Estas instrucciones se han llamado tradicionalmente **llamadas al sistema**, aunque ahora pueden implementarse de varias formas. Para entender realmente lo que los sistemas operativos hacen, debemos examinar con detenimiento esta interfaz. Las llamadas disponibles en la interfaz varían de un sistema operativo a otro (aunque los conceptos subyacentes tienden a ser similares).

Por tanto, nos vemos obligados a escoger entre (1) generalidades vagas (“los sistemas operativos tienen llamadas al sistema para leer archivos”) y (2) algún sistema específico (“MINIX tiene una llamada al sistema READ) con tres parámetros: uno para especificar el archivo, uno para indicar dónde deben colocarse los datos y uno para indicar cuántos bytes deben leerse”)

Hemos escogido el segundo enfoque. Esto implica más trabajo, pero nos permite entender mejor qué es realmente lo que hacen los sistemas operativos. En la sección 1.4 examinaremos de cerca las llamadas al sistema presentes tanto en UNIX como en MINIX. Por sencillez, sólo nos referiremos a MINIX, pero las llamadas al sistema UNIX correspondientes se basan en POSIX en la mayor parte de los casos. Sin embargo, antes de estudiar las llamadas al sistema reales, vale la pena presentar un panorama general de MINIX, a fin de tener una idea global de qué es lo que hace un sistema operativo. Este panorama aplica igualmente bien a UNIX.

Las llamadas al sistema de MINIX pertenecen a dos categorías amplias: las que se ocupan de los procesos y las que se ocupan del sistema de archivos. A continuación las examinaremos por turno.

1.3.1 Procesos

Un concepto clave en MINIX, y en todos los sistemas operativos, es el **proceso**. Un proceso es básicamente un programa en ejecución. Cada proceso tiene asociado un **espacio de direcciones**, una lista de posiciones de memoria desde algún mínimo (usualmente 0) hasta algún máximo, que el proceso puede leer y escribir. El espacio de direcciones contiene el programa ejecutable, los datos del programa, y su pila. A cada proceso también se asocia un conjunto de registros, que incluyen el contador del programa, el apuntador de la pila y otros registros de hardware, así como toda la demás información necesaria para ejecutar el programa.

Volveremos al concepto de proceso con mucho mayor detalle en el capítulo 2, pero por ahora la forma más fácil de adquirir una idea intuitiva de lo que es un proceso es pensar en los sistemas de tiempo compartido. Periódicamente, el sistema operativo decide dejar de ejecutar un proceso y comenzar a ejecutar otro, por ejemplo, porque el primero ya tuvo más tiempo de CPU del que le tocaba durante el segundo anterior.

Cuando un proceso se suspende temporalmente de esta manera, debe reiniciarse después en el mismo estado exactamente en que estaba en el momento en que se le detuvo. Esto implica que toda la información acerca del proceso se debe guardar explícitamente en algún lugar durante la suspensión. Por ejemplo, es posible que el proceso tenga varios archivos abiertos para lectura. Cada uno de estos archivos tiene asociado un apuntador que indica la posición actual (es decir, el número del byte o registro que se leerá a continuación). Cuando un proceso se suspende temporalmente, es necesario guardar todos estos apuntadores para que una llamada READ ejecutada después de reiniciarse el proceso lea los datos correctos. En muchos sistemas operativos, toda la información acerca de cada proceso, aparte del contenido de su propio espacio de direcciones, se almacena en una tabla del sistema operativo llamada **tabla de procesos**, que es un arreglo (o lista enlazada) de estructuras, una para cada proceso existente en ese momento.

Así, un proceso (suspendido) consiste en su espacio de direcciones, por lo regular llamado **imagen de núcleo** (recordando las memorias de núcleos magnéticos que se usaban en el pasado), y su entrada en la tabla de procesos, que contiene sus registros, entre otras cosas.

Las llamadas al sistema de administración para procesos clave son las que se ocupan de la creación y terminación de procesos. Consideremos un ejemplo representativo. Un proceso llamado **intérprete de comandos o shell** lee comandos de una terminal. El usuario acaba de teclear un comando solicitando la compilación de un programa. El shell debe crear ahora un proceso nuevo que ejecute el compilador. Cuando ese proceso haya terminado la compilación, ejecutará una llamada al sistema para terminarse a sí mismo.

Si un proceso puede crear uno o más procesos distintos (denominados **procesos hijos**) y éstos a su vez pueden crear procesos hijos, pronto llegamos a la estructura de árbol de procesos de la Fig. 1-5. Los procesos relacionados que están cooperando para realizar alguna tarea a menudo necesitan comunicarse entre sí y sincronizar sus actividades. Esta comunicación se llama comunicación entre procesos, y se estudiará con detalle en el capítulo 2.

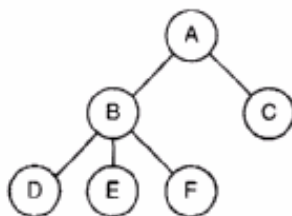


Figura 1-5. Un árbol de procesos. El proceso A creó dos procesos hijos, B y C. El proceso B creó tres procesos hijos, D, E y F.

Hay otras llamadas al sistema relacionadas con procesos que solicitan más memoria (o liberan memoria no utilizada), esperan que un proceso hijo termine, y superponen otro programa al suyo.

Ocasionalmente, se hace necesario comunicar información a un proceso en ejecución que no está simplemente esperando recibirla. Por ejemplo, un proceso que se comunica con otro proceso en una computadora distinta lo hace enviando mensajes por una red. A fin de prevenir la posibilidad de que un mensaje o su respuesta se pierda, el remitente puede solicitar que su propio sistema operativo le notifique cuando haya transcurrido cierto número de segundos, a fin de poder re-transmitir el mensaje si todavía no ha llegado un acuse de recibo. Después de establecer este temporizador, el programa puede seguir realizando otros trabajos.

Cuando ha transcurrido el número de segundos que se especificó, el sistema operativo envía una **señal** al proceso. La señal hace que el proceso suspenda temporalmente lo que estaba haciendo, guarde sus registros en la pila, y comience a ejecutar un procedimiento especial de manejo de señales, por ejemplo, para retransmitir un mensaje que al parecer se perdió. Una vez que el manejador de señales termina, el proceso en ejecución se reinicia en el estado en que estaba justo antes de la señal. Las señales son el análogo en software de las interrupciones de hardware, y pueden ser generadas por diversas causas además de la expiración de temporizadores. Muchas trampas detectadas por el hardware, como la ejecución de una instrucción no permitida o el empleo de una dirección no válida, también se convierten en señales que se envían al proceso culpable.

El administrador del sistema asigna un **uid** (identificador de usuario) a cada persona autorizada para usar MINIX. Cada proceso iniciado en MINIX tiene el uid de la persona que lo inició. Un proceso hijo tiene el mismo uid que su padre. Un uid, llamado **superusuario**, tiene facultades especiales, y puede violar muchas de las reglas de protección. En las instalaciones grandes, sólo el administrador del sistema conoce la contraseña necesaria para convertirse en superusuario, pero muchos de los usuarios ordinarios (sobre todo estudiantes) dedican un esfuerzo considerable a tratar de encontrar defectos en el sistema que les permitan convertirse en superusuarios sin contar con la contraseña.

1.3.2 Archivos

La otra categoría amplia de llamadas al sistema se relaciona con el sistema de archivos. Como ya se apuntó, una función importante del sistema operativo es ocultar las peculiaridades de los discos y otros dispositivos de E/S y presentar al programador un modelo abstracto, aseado y bonito, de archivos independientes del dispositivo. Es obvio que se necesitan llamadas al sistema para crear, eliminar, leer y escribir archivos. Antes de que un archivo pueda leerse, debe abrirse, y después de leerse debe cerrarse, así que también se incluyen llamadas para hacer estas cosas.

A fin de contar con un lugar para guardar los archivos, MINIX tiene el concepto de **directorio** como mecanismo para agrupar los archivos. Un estudiante, por ejemplo, podría tener un directorio para cada curso en el que está inscrito (donde guardaría los programas necesarios para ese curso), otro directorio para su correo electrónico, y otro más para su página base de la World Wide Web. Por tanto, se necesitan llamadas al sistema para crear y eliminar directorios. También

se incluyen llamadas para poner un archivo existente en un directorio, y para quitar un archivo de un directorio. Las entradas de directorio pueden ser archivos u otros directorios. Este modelo también da pie a una jerarquía —el sistema de archivos— como se muestra en la Fig. 1-6.

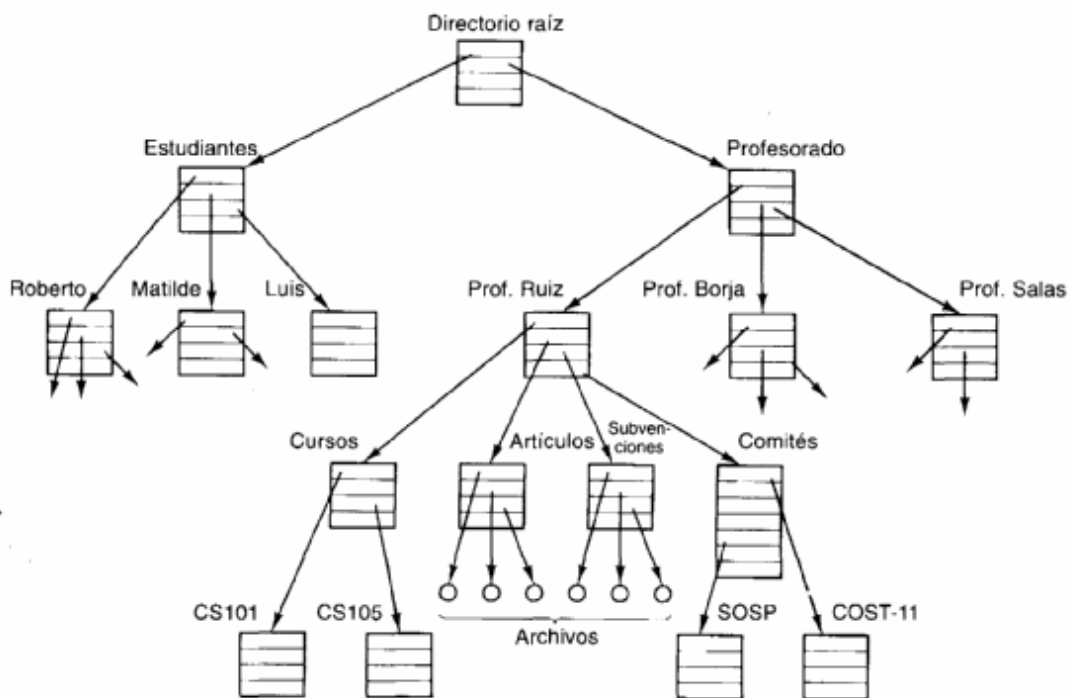


Figura 1-6. Sistema de archivos para un departamento universitario.

Las jerarquías de procesos y de archivos están organizadas como árboles, pero hasta ahí llega la similitud. Las jerarquías de procesos no suelen ser muy profundas (casi nunca tienen más de tres niveles), en tanto que las de archivos comúnmente tienen cuatro, cinco o incluso más niveles de profundidad. Las jerarquías de procesos por lo regular tienen una vida corta, generalmente de unos cuantos minutos como máximo, en tanto que la jerarquía de directorios podría existir durante años. La propiedad y protección también es diferente para los procesos y para los archivos. Típicamente, sólo un proceso padre puede controlar o incluso acceder a un proceso hijo, pero casi siempre existen mecanismos para permitir que los archivos y directorios sean leídos por un grupo más amplio que sólo el propietario.

Cada archivo dentro de la jerarquía de directorios se puede especificar dando su **nombre de ruta** a partir del tope de la jerarquía de directorios, el **directorio raíz**. Semejantes nombres de ruta absolutos consisten en la lista de directorios por los que se debe pasar partiendo del directorio raíz para llegar al archivo, separando los componentes con diagonales. En la Fig. 1-6, la ruta del archivo CSJOJ es IP rofesorado/P rof Ruiz/Cursos/CSJOJ. La diagonal inicial indica que la ruta es absoluta, es decir, que comienza en el directorio raíz.

En todo momento, cada proceso tiene un **directorio de trabajo** actual, en el cual se buscan los archivos cuyos nombres de ruta no comienzan con una diagonal. Por ejemplo, en la Fig. 1-6,

si */Profesorado/Prof. Ruiz* fuera el directorio de trabajo, el empleo del nombre de ruta *Cursos/CSJOI* se referiría al mismo archivo que el nombre de ruta absoluta dado en el párrafo anterior. Los procesos pueden cambiar de directorio de trabajo emitiendo una llamada al sistema que especifique el nuevo directorio de trabajo.

Los archivos y directorios en MINIX se protegen asignando a cada uno un código de protección binario de 9 bits. El código de protección consiste en tres campos de 3 bits, uno para el propietario, uno para otros miembros del grupo del propietario (el administrador del sistema divide a los usuarios en grupos) y uno para toda la demás gente. Cada campo tiene un bit para acceso de lectura, uno para acceso de escritura y uno para acceso de ejecución. Estos tres bits se conocen como **bits rwx**. Por ejemplo, el código de protección *rwxr-x-x* significa que el propietario puede leer, escribir o ejecutar el archivo, otros miembros del grupo pueden leer o ejecutar (pero no escribir) el archivo, y el resto de la gente puede ejecutar (pero no leer ni escribir) el archivo. En el caso de un directorio, *x* indica permiso de búsqueda. Un guión significa que el permiso correspondiente está ausente.

Antes de poder leer o escribir un archivo, es preciso abrirlo, y en ese momento se verifican los permisos. Si está permitido el acceso, el sistema devuelve un entero pequeño llamado **descriptor de archivo** que se usará en operaciones subsecuentes. Si el acceso está prohibido, se devuelve un código de error.

Otro concepto importante en MINIX es el de sistema de archivos montado. Casi todas las computadoras personales tienen una o más unidades de disco flexible en las que pueden insertarse y de las que pueden retirarse disquetes. A fin de contar con una forma congruente de manejar estos medios removibles (y también los CD-ROM, que también son removibles), MINIX permite conectar el sistema de archivos del disco flexible al árbol principal. Considere la situación de la Fig. 1-7(a). Antes de la llamada MOUNT, el disco en RAM (disco simulado en la memoria principal) contiene el **sistema de archivos raíz**, o primario, y la unidad 0 contiene un disquete que contiene otro sistema de archivos.

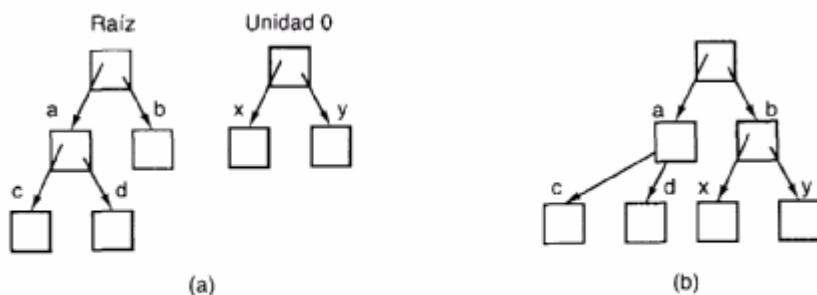


Figura 1-7. (a) Antes de montarse, los archivos de la unidad 0 no están accesibles. (b) Después de montarse, esos archivos forman parte de la jerarquía de archivos.

Sin embargo, no podemos usar el sistema de archivos de la unidad 0, porque no hay forma de especificar nombres de ruta en él. MINIX no permite anteponer a los nombres de ruta un nombre o número de unidad; ésa sería precisamente la clase de dependencia del dispositivo que los sistemas operativos deben eliminar. En vez de ello, la llamada al sistema MOUNT permite conectar el sistema

de archivos de la unidad O al sistema de archivo raíz en cualquier lugar en el que el programa quiera que esté. En la Fig. 1-7(b) el sistema de archivos de la unidad O se montó en el directorio `h`, permitiendo así el acceso a los archivos `/b/x` y `Ib/y`. Si el directorio `b` hubiera contenido archivos, éstos no habrían estado accesibles mientras estuviera montada la unidad O, ya que `Ib` se habría referido al directorio raíz de la unidad O. (No poder acceder a esos archivos no es tan grave como parece a primera vista: los sistemas de archivos casi siempre se montan en directorios vacíos.)

Otro concepto importante en MINIX es el **archivo especial**. Los archivos especiales sirven para hacer que los dispositivos de E/S semejen archivos. Así, esos dispositivos pueden leerse y escribirse usando las mismas llamadas al sistema que se usan para leer y escribir archivos. Existen dos tipos de archivos especiales: **archivos especiales por bloques** y **archivos especiales por caracteres**. Los primeros se usan para modelar dispositivos que consisten en una colección de bloques directamente direccionables, como los discos. Al abrir un archivo especial por bloques y leer, digamos, el bloque 4, un programa puede acceder directamente al bloque 4 del dispositivo, pasando por alto la estructura del sistema de archivos que contiene. De forma similar, los archivos especiales por caracteres se usan para modelar impresoras, módems y otros dispositivos que aceptan o producen flujos de caracteres.

La última característica que mencionaremos en esta reseña general se relaciona tanto con los procesos como con los archivos: los conductos. El **conducto** es una especie de pseudoarchivo que puede servir para conectar dos procesos, como se muestra en la Fig. 1-8. Cuando el proceso A desea enviar datos al proceso B, escribe en el conducto como si fuera un archivo de salida. El proceso B puede leer los datos leyendo del conducto como si fuera un archivo de entrada. Así, la comunicación entre procesos en MINIX se parece mucho a las lecturas y escrituras de archivos normales. Es más, la única forma en que un proceso puede descubrir que el archivo de salida en el que está escribiendo no es realmente un archivo, sino un conducto, es emitiendo una llamada especial al sistema.

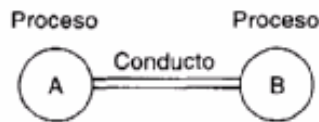


Figura 1-8. Dos procesos conectados por un conducto.

1.3.3 El shell

El sistema operativo MINIX es el código que ejecuta las llamadas al sistema. Los editores, compiladores, ensambladores, vinculadores e intérpretes de comandos definitivamente no forman parte del sistema operativo, aunque son importantes y útiles. A riesgo de confundir un poco las cosas, en esta sección examinaremos brevemente el intérprete de comandos de MINIX, llamado shell, que, si bien no es parte del sistema operativo, utiliza intensivamente muchas de las características del sistema operativo y, por tanto, es un buen ejemplo de la forma en que pueden usarse

las llamadas al sistema. El shell también es la interfaz primaria entre un usuario sentado ante su terminal y el sistema operativo.

Cuando un usuario ingresa en el sistema, se inicia un shell. El shell tiene la terminal como entrada estándar y salida estándar, y lo primero que hace es exhibir la **indicación** (*prompt*), un carácter como un signo de dólar, que le indica al usuario que el shell está esperando para aceptar un comando. Si el usuario ahora teclea

```
date
```

por ejemplo, el shell crea un proceso hijo y ejecuta el programa *date* como hijo. Mientras se está ejecutando el proceso hijo, el shell espera a que termine. Cuando el hijo termina, el shell exhibe otra vez la indicación y trata de leer la siguiente línea de entrada.

El usuario puede especificar que la salida estándar sea redirigida a un archivo, por ejemplo,

```
date >archivo
```

De forma similar, la entrada estándar puede redirigirse, como en

```
sort <archivo 1 >archivo2
```

que invoca el programa *sort* con entradas tomadas de *archivo1* y enviando las salidas a *archivo2*.

La salida de un programa puede usarse como entrada para otro programa conectándolos con un conducto Así,

```
cat archivol archivo2 archivo3 | sort >/dev/lp
```

invoca el programa *cat* para concatenar tres archivos y enviar la salida a *son* para que acomode todas las líneas en orden alfabético. La salida de *son* se redirige al archivo */dev/lp*, que es un nombre típico para el archivo especial por caracteres de la impresora. (Por convención, todos los archivos especiales se guardan en el directorio */dev*.)

Si un usuario escribe un signo *&* después de un comando, el shell no espera hasta que se completa, sino que exhibe una indicación de inmediato. Por tanto,

```
cat archivol archivo2 archivo3 | sort >/dev/lp &
```

inicia el ordenamiento como trabajo de segundo plano, permitiendo que el usuario siga trabajando normalmente mientras se está realizando el ordenamiento. El shell tiene varias otras características interesantes que no tenemos espacio para examinar aquí. Consulte cualquiera de las referencias sugeridas sobre UNIX si desea más información acerca del shell.

1.4 LLAMADAS AL SISTEMA

Armados con nuestro conocimiento general de cómo MINIX maneja los procesos y los archivos, ahora podemos comenzar a examinar la interfaz entre el sistema operativo y sus programas de aplicación, es decir, el conjunto de llamadas al sistema. Si bien esta explicación se refiere específicamente a *posix* (Norma Internacional 9945-1), y por tanto también a MINIX, casi todos

los sistemas operativos modernos tienen llamadas al sistema que realizan las mismas funciones, aunque los detalles sean diferentes. Puesto que el mecanismo real de la emisión de una llamada al sistema depende mucho de la máquina, y a menudo debe expresarse en código de ensamblador, se proporciona una biblioteca de procedimientos que permite efectuar llamadas al sistema desde programas en C.

A fin de hacer más claro el mecanismo de las llamadas al sistema, examinemos brevemente `READ` (leer). Esta llamada tiene tres parámetros: el primero especifica el archivo, el segundo especifica el buffer, y el tercero especifica el número de bytes por leer. Una llamada de `READ` desde un programa en C podría verse así:

```
cuenta = read(file, buffer, nbytes);
```

La llamada al sistema (y el procedimiento de biblioteca) devuelve en cuenta el número de bytes que realmente se leyeron. Este valor normalmente es igual a `nbytes`, pero puede ser menor, si, por ejemplo, se llega al fin del archivo durante la lectura.

Si la llamada al sistema no puede ejecutarse, ya sea a causa de un parámetro no válido o de un error de disco, se asignará el valor `-1` a cuenta, y el número del error se colocará en una variable global, *errno*. Los programas siempre deben revisar los resultados de una llamada al sistema para ver si ocurrió un error.

MINIX tiene un total de 53 llamadas al sistema, las cuales se listan en la Fig. 1-9, agrupadas por comodidad en sE/S categorías. En las siguientes secciones examinaremos brevemente cada una de estas llamadas para ver qué hacen. En gran medida, los servicios que estas llamadas ofrecen determinan la mayor parte de lo que el sistema operativo tiene que hacer, ya que la administración de recursos en las computadoras personales es mínima (al menos comparada con las máquinas grandes que tienen muchos usuarios).

Como acotación, vale la pena señalar que lo que constituye una llamada al sistema está abierto a interpretación. El estándar `POS` especifica varios procedimientos que un sistema que se ajuste a él debe proporcionar, pero no especifica si se trata de llamadas al sistema, llamadas de biblioteca o algo más. En algunos casos, los procedimientos `PO` se ofrecen como rutinas de biblioteca en MINIX. En otros, varios procedimientos requeridos son sólo variaciones menores de un procedimiento, y una llamada al sistema se encarga de todos ellos.

1.4.1 Llamadas al sistema para administración de procesos

El primer grupo de llamadas se ocupa de la administración de procesos. `FORK` (bifurcar) es un buen lugar para iniciar la explicación. `FORK` es la única forma de crear un proceso nuevo. Esta llamada crea un duplicado exacto del proceso original, incluidos todos los descriptores de archivo, registros... todo. Después del `FORK`, el proceso original y la copia (el padre y el hijo) siguen cada quien su camino. Todas las variables tienen valores idénticos en el momento del `FORK`, pero dado que los datos del padre se copian para crear el hijo, los cambios subsecuentes en uno de ellos no afectan al otro. (El texto, que es inmutable, es compartido entre padre e hijo.) La llamada `FORK` devuelve un valor, que es cero en el padre e igual al identificador de proceso o **pid** del hijo en el

Administración de procesos	<p>pid = fork() pid = waitpid(pid, &statloc, opts) s = wait(&status) s = execve(name, argv, envp) exit(status) size = brd(addr) pid = getpid() pid = getpgrp() pid = setsid() l = ptrace(req, pid, addr, data)</p>	<p>Crea un proceso hijo idéntico al padre Espera que un hijo termine Versión vieja de waitpid Sustituye la imagen en memoria de un proceso Termina la ejecución de un proceso y devolver su estado Fija el tamaño del segmento de datos Devuelve el identificador de proceso del invocador Devuelve el identificador del grupo de procesos del invocador Crea una nueva sesión y devolver su identificador de grupo de procesos Se usa para depurar</p>
Señales	<p>s = sigaction(sig, &act, &oldact) s = sigreturn(&context) s = sigprocmask(how, &set, &old) s = sigpending(set) s = sigsuspend(sigmask) s = kill(pid, sig) residual = alarm(seconds) s = pause() fd = creat(name, mode) fd = mknod(name, mode, addr) fd = open(file, how, ...) s = close(fd) n = read(fd, buffer, nbytes) n = write(fd, buffer, nbytes) pos = lseek(fd, offset, whence) s = stat(name, &buf) s = fstat(fd, &buf) fd = dup(fd) s = pipe(&fd[0]) s = ioctl(fd, request, argp) s = access(name, amode) s = rename(old, new) s = fcntl(fd, cmd, ...)</p>	<p>Define la acción a emprender al recibir señales Regresa de una señal Examina o cambia la máscara de señal Obtiene el conjunto de señales bloqueadas Sustituye la máscara de señal y suspende el proceso Envía una señal a un proceso Pone la alarma del reloj Suspende el invocador hasta la siguiente señal Forma obsoleta de crear un archivo nuevo Crea un nodo-i normal, especial o de directorio Abre un archivo para leer, escribir o ambas cosas Cierra un archivo abierto Lee datos de un archivo colocándolos en un <i>buffer</i> Escribe datos de un <i>buffer</i> a un archivo Mueve el apuntador de archivos Obtiene la información de estado de un archivo Obtiene la información de estado de un archivo Asigna un nuevo descriptor de archivo a un archivo abierto Crea un conducto Realiza operaciones especiales con un archivo Verifica la accesibilidad de un archivo Da a un archivo un nuevo nombre Bloqueo de archivos y otras operaciones</p>
Administración de archivos	<p>s = mkdir(name, mode) s = rmdir(name) s = link(name1, name2) s = unlink(name) s = mount(special, name, flag) s = umount(special) s = sync() s = chdir(dirname) s = chroot(dirname)</p>	<p>Crea un directorio nuevo Elimina un directorio vacío Crea una nueva entrada, name2, que apunta a name1 Elimina una entrada de directorio Monta un sistema de archivos Desmonta un sistema de archivos Desaloja todos los bloques en caché al disco Cambia el directorio de trabajo Cambia el directorio raíz</p>
Administración de directorios y sistemas de archivos	<p>s = chmod(name, mode) uid = getuid() gid = getgid() s = setgid(gid) s = setgid(gid) s = chown(name, owner, group) oldmask = umask(complmode)</p>	<p>Cambia los bits de protección de un archivo Obtiene el uid del invocador Obtiene el gid del invocador Establece el uid del invocador Establece el gid del invocador Cambia el propietario y el grupo de un archivo Cambia la máscara de modo</p>
Protección	<p>seconds = time(&seconds) s = stime(tp) s = utime(file, timep) s = times(buffer)</p>	<p>Obtiene el tiempo transcurrido desde el 1o. de enero de 1970 Establece el tiempo transcurrido desde el 1o. de enero de 1970 Establece el tiempo de "último acceso" de un archivo Obtiene los tiempos de usuario y sistema gastados hasta el momento</p>
Administración del tiempo		

Figura 1-9. Las llamadas al sistema de MINIX. El código de retorno *s* es -1 si ocurre un error; *fd* es un descriptor de archivo, y *n* es una cuenta de bytes. Los demás códigos de retorno son lo que el nombre sugiere.

proceso padre. Con base en el pid devuelto, los dos procesos pueden saber cuál es el proceso padre y cuál es el proceso hijo.

En la mayor parte de los casos, después de un FORK, el hijo tendrá que ejecutar código diferente del de su padre. Considere el caso del shell. Éste lee un comando de la terminal, bifurca un proceso hijo, espera que el hijo ejecute el comando, y luego lee el siguiente comando cuando el hijo termina. Para esperar a que el hijo termine, el padre ejecuta una llamada al sistema WAITPID (esperar PID), que simplemente espera hasta que el hijo termina (cualquier hijo si existe más de uno). WAITPID puede esperar un hijo específico, o cualquier hijo si se asigna -1 al primer parámetro. Cuando WAITPID termina, se asignará el valor de la situación de salida (terminación normal o anormal y valor de salida) del hijo a la dirección a la que apunta el segundo parámetro. Se ofrecen también varias otras opciones. La llamada WAITPID sustituye a la llamada anterior WAIT que ahora es obsoleta pero se incluye por razones de compatibilidad hacia atrás.

Consideremos ahora la forma en que el shell usa FORK. Cuando se teclea un comando, el shell bifurca un nuevo proceso. Este proceso hijo debe ejecutar el comando del usuario, cosa que hace utilizando la llamada al sistema EXEC (ejecutar), que hace que toda su imagen de núcleo sea sustituida por el archivo nombrado en su primer parámetro. En la Fig. 1-10 se muestra un shell muy simplificado que ilustra el uso de FORK, WAITPID y EXEC.

```

while (TRUE) {                               /* repetir siempre */
  read_command(command, parameters);         /* leer entrada de la terminal */

  if(fork() != 0) {                           /* bifurcar proceso hijo */
    /* Código del padre. */
    waitpid(-1, &status, 0);                 /* esperar que el hijo salga */
  } else {
    /* Código del hijo. */
    execve(command, parameters, 0);         /* ejecutar comando */
  }
}

```

Figura 1-10. Un shell reducido al mínimo. En todo este libro, se supone que *TRUE* es por definición 1.

En el caso más general, EXEC tiene tres parámetros: el nombre del archivo que se va a ejecutar, un apuntador al arreglo de argumentos y un apuntador al arreglo de entorno. Éstos se describirán en breve. Se proporcionan diversas rutinas de biblioteca, incluidas *execi*, *execv*, *execle* y *execve* que permiten omitir los parámetros o especificarlos de diversas formas. En todo este libro usaremos el nombre EXEC para representar la llamada al sistema invocada por todas estas rutinas.

Consideremos el caso de un comando como

```
cp archiv1 archiv2
```

que sirve para copiar *archiv1* en *archiv2*. Una vez que el shell ha bifurcado, el proceso hijo localiza y ejecuta el archivo *cp* y le pasa los nombres de los archivos de origen y destino.

El programa principal de cp (y el de casi todos los demás programas) contiene la declaración

```
main(argc, argv, envp)
```

donde *argc* es una cuenta del número de elementos que hay en la línea de comandos, incluido el nombre del programa. Para el ejemplo anterior, *argc* es 3.

El segundo parámetro, *argv*, es un apuntador a un arreglo. El elemento *i* de ese arreglo es un apuntador a la *i*-ésima cadena de la línea de comandos. En nuestro ejemplo, *argv* apuntaría a la cadena "cp". De forma similar, *argv* apuntaría a la cadena de 5 caracteres "archivo 1", y *argv*[2] apuntaría a la cadena de 5 caracteres "archivo2".

El tercer parámetro de *main*, *envp*, es un apuntador al entorno, un arreglo de cadenas que contienen asignaciones de la forma nombre = valor y que sirve para pasar a un programa información como el tipo de terminal y el nombre del directorio base. En la Fig. 1-10 no se pasa un entorno al hijo, así que el tercer parámetro de *execve* es un cero.

Si EXEC parece complicado, no se desanime; ésta es la llamada al sistema más compleja. Todas las demás son mucho más sencillas. Como ejemplo de llamada sencilla, considere EXIT (salir), que los procesos deben usar al terminar de ejecutarse. EXIT tiene un solo parámetro, el estado de salida (0 a 255), que se devuelve al padre en la variable *status* de la llamada al sistema WAIT o WAITPID. El byte de orden bajo de *status* contiene el estado de terminación, siendo 0 la terminación normal, y los demás valores, diversas condiciones de error. El byte de orden alto contiene el estado de salida del hijo (0 a 255). Por ejemplo, si un proceso padre ejecuta la instrucción

```
n = waitpid(-1, &status, options);
```

se suspenderá hasta que algún proceso hijo termine. Si el hijo sale con, digamos, 4 como parámetro de *exit*, el padre será despertado después de asignarse el *pid* del hijo a *n* y *Ox0400* a *status*. (En todo este libro se usará la convención de C de anteponer *Ox* a las constantes hexadecimales.)

Los procesos en MINIX tienen su memoria dividida en tres segmentos: el **segmento de texto** (esto es, el código de programa), el **segmento de datos** (es decir, las variables) y el **segmento de pila**. El segmento de datos crece hacia arriba y el de pila lo hace hacia abajo, como se muestra en la Fig. 1-11. Entre ellos hay una brecha de espacio de direcciones no utilizado. La pila crece hacia la brecha automáticamente, según se necesite, pero la expansión del segmento de datos se efectúa explícitamente usando la llamada al sistema BRK. BRK tiene un parámetro, que da la dirección donde debe terminar el segmento de datos. Esta dirección puede ser mayor que el valor actual (el segmento de datos está creciendo) o menor (el segmento de datos se está encogiendo). Desde luego, el parámetro debe ser menor que el apuntador de la pila, pues de otro modo los segmentos de datos y de pila se traslaparían, cosa que está prohibida.

Para comodidad del programador, se ofrece una rutina de biblioteca *sbrk* que también cambia el tamaño del segmento de datos, sólo que su parámetro es el número de bytes por agregar a dicho segmento (un parámetro negativo reduce el tamaño del segmento de datos). Esta rutina opera siguiendo la pista al tamaño actual del segmento de datos, que es el valor devuelto por BRK, calculando el nuevo tamaño, y realizando una llamada pidiendo ese número de bytes. BRK y SBRK se consideraron demasiado dependientes de la implementación y no forman parte de POSIX.

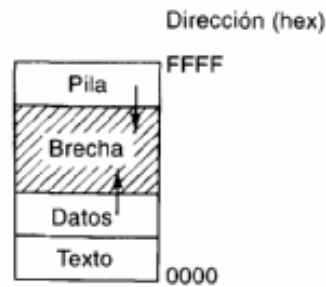


Figura 1-11. Los procesos tienen tres segmentos: texto, datos y pila. En este ejemplo, los tres están en el mismo espacio de direcciones, pero también se manejan espacios de instrucciones y datos separados.

La siguiente llamada al sistema relacionada con procesos también es la más sencilla, `GETPID`, que se limita a devolver el pid de quien la invoca. Recuerde que en `FORK`, sólo se proporcionaba el pid del hijo al padre. Si el hijo desea conocer su propio pid, deberá usar `GETPID`. La llamada `GETPGRP` devuelve el pid del grupo de procesos del invocador. `SETSID` crea una nueva sesión y asigna el pid del invocador al pid del grupo de procesos. Las sesiones están relacionadas con una característica opcional de Posix llamada **control de trabajos**, que no es apoyada por MINIX y de la cual no nos ocuparemos más.

La última llamada al sistema relacionada con procesos, `Ptrace`, es utilizada por los programas depuradores para controlar el programa que se está depurando. Esta llamada permite al depurador leer y escribir en la memoria del proceso controlado y administrarla de otras maneras.

1.4.2 Llamadas al sistema para señalización

Aunque casi todas las formas de comunicación entre procesos son planeadas, existen situaciones en las que se requiere una comunicación inesperada. Por ejemplo, si un usuario accidentalmente le pide a un editor de textos que liste todo el contenido de un archivo muy largo, y luego se percató de su error, necesita alguna forma de interrumpir el editor. En MINIX, el usuario puede pulsar la tecla `DEL` (suprimir) del teclado, la cual envía una señal al editor. El editor atrapa la señal y detiene el listado. También pueden usarse señales para informar de ciertas trampas detectadas por el hardware, como una instrucción no permitida o un desbordamiento de punto flotante. Las expiraciones de tiempo también se implementan como señales.

Cuando se envía una señal a un proceso que no ha anunciado su disposición a aceptar esa señal, el proceso simplemente se termina sin más. Para evitar esta suerte, un proceso puede usar la llamada al sistema `SIGACTION` para anunciar que está preparada para aceptar algún tipo de señal, y proporcionar la dirección del procedimiento que manejará la señal, así como un lugar para almacenar la dirección del manejador actual. Después de una llamada a `SIGACTION`, si se

genera una señal del tipo pertinente (p. ej., la tecla DEL), el estado del proceso se mete en su propia pila y luego se invoca el manejador de señales, el cual puede ejecutarse durante el tiempo que desee y emitir todas las llamadas al sistema que quiera. En la práctica, empero, los manejadores de señales suelen ser más o menos cortos. Cuando un procedimiento de manejo de señales termina, llama a SIGRETURN para que el proceso continúe donde estaba antes de la señal. La llamada SIGACTION sustituye a la antigua llamada SIGNAL, que ahora se proporciona como procedimiento de biblioteca a fin de mantener la compatibilidad hacia atrás.

Las señales pueden bloquearse en MINIX. Una señal bloqueada se mantiene pendiente hasta que se desbloquea; no se entrega, pero tampoco se pierde. La llamada SIGPROCMASK permite a un proceso definir el conjunto de señales bloqueadas presentando al kernel un mapa de bits. Un proceso también puede preguntar por el conjunto de señales que actualmente están pendientes y cuya entrega no se ha permitido porque están bloqueadas. La llamada SIGPENDING devuelve este conjunto en forma de mapa de bits. Por último, la llamada SIGSUSPEND permite a un proceso establecer atómicamente el mapa de bits de las señales bloqueadas y suspenderse a sí mismo.

En vez de proporcionar una función que atrape una señal, el programa puede especificar la constante SIG_IGN para que se haga caso omiso de todas las señales subsecuentes del tipo especificado, o SIG_DFL para restablecer la acción por omisión de la señal cuando ocurra. La acción por omisión es terminar el proceso o bien hacer caso omiso de la señal, dependiendo de la señal. Como ejemplo del uso de SIG_IGN, consideremos lo que sucede cuando el shell bifurca un proceso de segundo plano como resultado de

comando &

No sería conveniente que una señal DEL del teclado afectara el proceso de segundo plano, así que el shell hace lo siguiente después del FORK pero antes del EXEC:

```
sigaction(SIGINT, SIG_IGN, NULL);
```

y

```
sigaction(SIGQUIT, SIG_IGN, NULL);
```

para inhabilitar las señales DEL y QUIT. (La señal QUIT se genera con CTRL-\; es lo mismo que DEL excepto que, si no es atrapada o ignorada, realiza un vaciado de núcleo del proceso que se terminó.) En el caso de procesos de primer plano (sin el &), estas señales no se ignoran.

Pulsar la tecla DEL no es la única forma de enviar una señal. La llamada al sistema KILL permite a un proceso enviar una señal a otro proceso (siempre que tengan el mismo uid; los procesos no relacionados entre sí no se pueden enviar señales mutuamente). Volviendo al ejemplo del proceso de segundo plano, suponga que se inicia un proceso de segundo plano pero posteriormente se decide que se le debe terminar. SIGINT y SIGQUIT han sido inhabilitadas, así que necesitamos algún otro mecanismo. La solución es usar el programa kill, que usa la llamada al sistema KILL para enviar una señal a cualquier proceso. Si enviamos la señal 9 (SIGKILL) a un proceso de segundo plano, podremos terminarlo. SIGKILL no puede atraparse ni ignorarse.

En muchas aplicaciones de tiempo real se hace necesario interrumpir un proceso después de un intervalo de tiempo específico a fin de hacer algo, como retransmitir un paquete que tal vez se

perdió en una línea de comunicación no confiable. Se proporciona la llamada al sistema ALARM para manejar esta situación. El parámetro especifica un intervalo, en segundos, después del cual se envía una señal SIGALARM al proceso. Un proceso sólo puede tener una alarma pendiente en un momento dado. Si se emite una llamada ALARM con un parámetro de 10 segundos, y 3 segundos después se emite otra llamada ALARM con un parámetro de 20 segundos, sólo se generará una señal, 20 segundos después de la segunda llamada. La primera señal será cancelada por la segunda llamada ALARM. Si el parámetro de ALARM es cero, se cancelará cualquier señal de alarma pendiente. Si una señal de alarma no es atrapada, se emprenderá la acción por omisión y se terminará el proceso al que se envió la señal.

A veces sucede que un proceso no tiene nada que hacer en tanto no llegue una señal. Por ejemplo, consideremos un programa de instrucción asistida por computadora que está probando la velocidad de lectura y la comprensión. El programa exhibe texto en la pantalla y luego llama a ALARM para que le envíe una señal después de 30 segundos. Mientras el estudiante está leyendo el texto, el programa no tiene nada que hacer; podría quedar inerte dando vueltas en un ciclo vacío, pero eso desperdiciaría tiempo de CPU que otro proceso o usuario podría necesitar. Una solución mejor es usar PAUSE, que le ordena a MINIX que suspenda el proceso hasta la siguiente señal.

1.4.3 Llamadas al sistema para administración de archivos

Muchas llamadas al sistema se relacionan con el sistema de archivos. En esta sección examinaremos llamadas que operan sobre archivos individuales; en la siguiente veremos las que trabajan con directorios o el sistema de archivos global. Usamos la llamada CREAT para crear un nuevo archivo (la razón por la que esta llamada es CREAT y no CREATE se ha perdido en las brumas del tiempo). Los parámetros de CREAT dan el nombre del archivo y el modo de protección. Así,

```
Id = creat( "abc", 0751);
```

crea un archivo llamado abc con el modo 0751 octal (en C, un cero inicial indica que una constante está en octal). Los nueve bits de orden bajo de 0751 especifican los bits rwx para el propietario (7 significa permiso de lectura-escritura-ejecución), su grupo (5 significa permiso de lectura-ejecución) y otros (1 significa sólo ejecución).

CREAT no sólo crea un archivo nuevo sino que también lo abre para escritura, sea cual sea el modo del archivo. Se puede usar el descriptor de archivo devuelto, fd, para escribir el archivo. Si se invoca CREAT para un archivo existente, ese archivo se truncará a longitud 0, a condición, desde luego, que los permisos sean los correctos. La llamada CREAT es obsoleta, ya que ahora OPEN puede crear archivos nuevos, pero se ha incluido para asegurar la compatibilidad hacia atrás.

Los archivos especiales se crean usando MKNOD en lugar de CREAT. Una llamada típica es

```
Id = mknod("/dev/ttyc2", 020744, 0x0402);
```

que crea un archivo llamado */dev/ttyc2* (el nombre usual para la consola 2) y le asigna el modo 020744 octal (un archivo especial de caracteres con bits de protección *rwxr--r--*). El tercer parámetro

contiene el dispositivo principal (4) en el byte de orden alto y el dispositivo secundario (2) en el byte de orden bajo. El dispositivo principal podría haber sido cualquier cosa, pero un archivo llamado `/dev/ttvc2` debe ser el dispositivo secundario 2. Las llamadas a `MKNOD` fallarán si el usuario no es el superusuario.

Para leer o escribir un archivo existente, el archivo primero debe abrirse con `OPEN`. Esta llamada especifica el nombre del archivo que se va a abrir, ya sea como nombre de ruta absoluto o relativo al directorio de trabajo, y un código de `O_RDONLY`, `O_WRONLY` u `O_RDWR`, que significan abrir para lectura, escritura o ambas cosas. El descriptor de archivo devuelto puede usarse entonces para leer o escribir. Después, el archivo puede cerrarse con `CLOSE`, con lo que el descriptor de archivo queda disponible para reutilizarse en un `CREATE` u `OPEN` subsecuente.

Las llamadas más utilizadas son sin duda `READ` y `WRITE`. Ya vimos `READ` antes; `WRITE` (escribir) tiene los mismos parámetros.

Aunque casi todos los programas leen y escriben archivos secuencialmente, algunos programas de aplicación necesitan tener acceso directo a cualquier parte de un archivo. Cada archivo tiene asociado un apuntador que indica la posición actual en el archivo. Al leer (o escribir) secuencialmente, este apuntador normalmente apunta al siguiente byte que se leerá (o escribirá). La llamada `LSEEK` cambia el valor del apuntador de posición, con lo que las llamadas subsecuentes a `READ` o `WRITE` pueden comenzar en cualquier lugar del archivo, o incluso más allá de su final.

`LSEEK` tiene tres parámetros: el primero es el descriptor de archivo para el archivo, el segundo es una posición en el archivo y el tercero indica si dicha posición es relativa al principio del archivo, la posición actual o el final del archivo. El valor devuelto por `LSEEK` es la posición absoluta en el archivo una vez que se ha cambiado el apuntador.

Para cada archivo, `MINIX` se mantiene al tanto del modo del archivo (archivo regular, archivo especial, directorio, etc.), su tamaño, la hora de la última modificación y otra información. Los programas pueden pedir ver esta información usando las llamadas al sistema `STAT` y `FSTAT`. La única diferencia entre las dos es que la primera especifica el archivo por su nombre, en tanto que la segunda recibe un descriptor de archivo, lo que lo hace útil para archivos abiertos, sobre todo la entrada estándar y la salida estándar, cuyos nombres tal vez no se conozcan. Ambas llamadas requieren como segundo parámetro un apuntador a una estructura en la cual se colocará la información. La estructura se muestra en la Fig. 1-12.

Al manipular descriptores de archivo, a veces resulta útil la llamada `DUP`. Considere, por ejemplo, un programa que necesita cerrar la salida estándar (descriptor de archivo 1), sustituir otro archivo como salida estándar, invocar una función que escribe ciertas salidas en la salida estándar, y luego restablecer la situación original. Basta cerrar el descriptor de archivo 1 y luego abrir un archivo nuevo para hacer que ese archivo sea la salida estándar (suponiendo que la entrada estándar, con descriptor de archivo 0, esté en uso), pero luego será imposible restablecer la situación original.

La solución consiste en ejecutar primero la instrucción

```
fd = dup(1);
```

que usa la llamada al sistema `DUP` para asignar un nuevo descriptor de archivo, `fd`, y hacer que corresponda al mismo archivo que la salida estándar. Luego se puede cerrar la salida estándar y

```

struct stat {
    short st_dev;           /* dispositivo donde debe estar el nodo-i */
    unsigned short st_ino; /* número del nodo-i */
    unsigned short st_mode; /* palabra de modo */
    short st_nlink;        /* número de vínculos */
    short st_uid;          /* identificador del usuario */
    short st_gid;          /* identificador del grupo */
    short st_rdev;         /* dispositivo principal/secundario para archivos especiales */
    long st_size;          /* tamaño del archivo */
    long st_atime;         /* hora del último acceso */
    long st_mtime         /* hora de la última modificación */
    long st_ctime         /* hora de la última modificación del nodo-i */
};

```

Figura 1-12. La estructura empleada para devolver información para las llamadas al sistema STAT y FSTAT. En el código real se usan nombres simbólicos para algunos de los tipos.

abrir y usar un nuevo archivo. Cuando llegue el momento de restablecer la situación original, se podrá cerrar el descriptor de archivo 1 y luego ejecutarse

```
n = dup(fd);
```

para asignar el descriptor de archivo más bajo, que es 1, al mismo archivo que fd. Por último, puede cerrarse fd y estamos otra vez donde empezamos.

La llamada DUP tiene una variante que permite hacer que un descriptor de archivo arbitrario no asignado se refiera a un archivo abierto dado. Esta variante se invoca con

```
dup2(fd, fd2);
```

donde fd se refiere a un archivo abierto y fd2 es el descriptor de archivo no asignado que ahora se referirá al mismo archivo que fd. Por ejemplo, si fd se refiere a la entrada estándar (descriptor de archivo 0) y fd2 es 4, después de la llamada los descriptors de archivo 0 y 4 se referirán ambos a la entrada estándar.

La comunicación entre procesos en MINIX emplea conductos, como ya se explicó. Cuando un usuario teclea

```
cat archivol archivo2 1 sort
```

el shell crea un conducto y hace que la salida estándar del primer proceso escriba en el conducto para que la entrada estándar del segundo proceso pueda leer de él. La llamada al sistema PIPE crea un conducto y devuelve dos descriptors de archivo, uno para escribir y otro para leer. La llamada es

```
pipe(&fd[0]);
```

donde fd es un arreglo de dos enteros, y fd y fd + 1 son los descriptors de archivo para leer y para escribir, respectivamente. Por lo regular, después viene un FORK, y el padre cierra el descriptor

de archivo para leer y el hijo cierra el descriptor de archivo para escribir (o viceversa), de modo que, cuando terminan, uno de los procesos puede leer el conducto y el otro puede escribir en él.

La Fig. 1-13 muestra un esqueleto de procedimiento que crea dos procesos, enviando la salida del primero al segundo por un conducto. (Un ejemplo más realista verificaría errores y manejaría argumentos.) Primero se crea un conducto, y luego el procedimiento bifurca. El padre finalmente se convierte en el primer proceso de la tubería y el proceso hijo se convierte en el segundo. Puesto que los archivos que se van a ejecutar, `process1` y `process2` no saben que forman parte de una tubería, es indispensable que los descriptors de archivo se manipulen de modo que la salida estándar del primer proceso sea el conducto y la entrada estándar del segundo sea el conducto. El padre cierra primero el descriptor de archivo para leer del conducto, luego cierra la salida estándar y emite una llamada `DUP` que permite al descriptor de archivo 1 escribir en el conducto. Es importante tener presente que `DUP` siempre devuelve el descriptor de archivo más bajo disponible, que en este caso es 1. Luego, el programa cierra el otro descriptor de archivo del conducto.

Después de la llamada `EXEC`, el proceso iniciado tendrá los descriptors de archivo 0 y 2 sin modificación, y el 1 para escribir en el conducto. El código del hijo es análogo. El parámetro para `execl` se repite porque el primero es el archivo que se va a ejecutar y el segundo es el primer parámetro, que casi todos los programas esperan que sea el nombre del archivo.

La siguiente llamada al sistema, `IOCTL`, es potencialmente aplicable a todos los archivos especiales. Por ejemplo, es utilizada por los controladores de dispositivos de bloques como el SCSI para controlar dispositivos de cinta y CD-ROM. No obstante, su uso principal es con archivos especiales por caracteres, sobre todo las terminales. `rosix` define varias funciones que la biblioteca traduce a llamadas `IOCTL`. Las funciones de biblioteca `tcgetattr` y `tcsetattr` usan `IOCTL` para cambiar los caracteres que se usan para corregir los errores de digitación en la terminal, cambiar el modo de la terminal, etcétera.

El **modo cocido** es el modo de terminal normal, en el que los caracteres de borrar y terminar funcionan normalmente, `CTRL-S` y `CTRL-Q` pueden usarse para detener e iniciar la salida de la terminal, `CTRL-D` significa fin de archivo, `DEL` genera una señal de interrupción y `CTRL-\` genera una señal de abandonar (`quit`) para forzar un vaciado de núcleo.

En **modo crudo**, todas estas funciones están inhabilitadas; todos los caracteres se pasan directamente al programa sin ningún procesamiento especial. Además, en modo crudo una lectura de la terminal proporciona al programa todos los caracteres que se han tecleado, incluso una línea parcial, en lugar de esperar hasta que se teclea una línea completa, como en el modo cocido.

El **modo cbreak** es intermedio. Los caracteres de borrar y terminar para edición están inhabilitados, lo mismo que `CTRL-D`, pero `CTRL-S`, `CTRL-Q`, `DEL` y `CTRL-\` están habilitados. Al igual que en modo crudo, se pueden devolver líneas parciales a los programas (si se desactiva la edición intralíneas no hay necesidad de esperar hasta haber recibido una línea completa; el usuario no puede cambiar de opinión y eliminarla, como en el modo cocido).

`Posix` no utiliza los términos cocido, crudo y `cbreak`. En la terminología de `POS` el **modo canónico** corresponde al modo cocido. En este modo están definidos once caracteres especiales, y la entrada es por líneas. En el **modo no canónico** un número mínimo de caracteres por aceptar y un tiempo, especificado en unidades de décimas de segundo, determinan la forma en que se satisfará

```

#define STD_INPUT 0          /* descriptor de archivo para entrada estándar */
#define STD_OUTPUT 1       /* descriptor de archivo para salida estándar */

pipeline(process1, process2)
char *process1, *process2; /* apuntadores a nombres de programas */
{
    int fd[2];

    pipe(&fd[0]);          /* crear un conducto */
    if (fork() != 0) {
        /* El proceso padre ejecuta estas instrucciones. */
        close(fd[0]);      /* el proceso 1 no necesita leer del conducto */
        close(STD_OUTPUT); /* preparar la nueva salida estándar */
        dup(fd[1]);        /* asignar fd[1] a la salida estándar */
        close(fd[1]);      /* este descriptor de archivo ya no se necesita */
        execl(process1, process1, 0);
    } else {
        /* El proceso hijo ejecuta estas instrucciones. */
        close(fd[1]);      /* el proceso 2 no necesita escribir en el conducto */
        close(STD_INPUT); /* preparar la nueva entrada estándar */
        dup(fd[0]);        /* asignar fd[0] a la entrada estándar */
        close(fd[0]);      /* este descriptor de archivo ya no se necesita */
        execl(process2, process2, 0);
    }
}

```

Figura 1-13. Esqueleto para establecer una tubería de dos procesos.

una lectura. Bajo POSIX hay un alto grado de flexibilidad, y es posible ajustar diversas banderas para hacer que el modo no canónico se comporte como el modo cbreak o el crudo. Los términos antiguos son más descriptivos, y seguiremos usándolos informalmente.

IOCTL tiene tres parámetros; por ejemplo, una llamada a *tcsetattr* para establecer los parámetros de terminal dará lugar a:

```
ioctl(fd, TCSETS, &termios);
```

El primer parámetro especifica un archivo, el segundo especifica una operación y el tercero es la dirección de la estructura *iosix* que contiene banderas y el arreglo de caracteres de control. Otros códigos de operación pueden posponer los cambios hasta que se haya enviado toda la salida, hacer que las entradas no leídas se desechen, y devolver los valores actuales.

La llamada al sistema *ACCESS* sirve para determinar si el sistema de protección permite cierto acceso a un archivo. Esta llamada es necesaria porque algunos programas pueden ejecutarse usando el *uid* de un usuario distinto. Este mecanismo de *SETUID* se describirá posteriormente.

La llamada al sistema RENAME sirve para dar a un archivo un nuevo nombre. Los parámetros especifican los nombres viejo y nuevo.

Por último, la llamada FCNTL se usa para controlar archivos de forma un tanto análoga a IOCTL (es decir, ambas son hacks horribles). Esta llamada tiene varias opciones, la más importante de las cuales es para poner candados a archivos a discreción. Usando FCNTL, un proceso puede poner y quitar candados a partes de archivos y probar una parte de un archivo para ver si tiene candado. La llamada no impone ninguna semántica de candados; los programas deben encargarse de ello.

1.4.4 Llamadas al sistema para administración de directorios

En esta sección examinaremos algunas llamadas al sistema que más bien están relacionadas con directorios o con el sistema de archivos global, no con un archivo específico como en la sección anterior. Las dos primeras llamadas, MKDIR y RMDIR, crean y eliminan directorios vacíos, respectivamente. La siguiente llamada es LINK. Su propósito es hacer posible que el mismo archivo aparezca con dos o más nombres, a menudo en diferentes directorios. Un uso típico es permitir que varios miembros del mismo equipo de programación compartan un archivo común, y que para cada uno el archivo aparezca en su propio directorio, tal vez con un nombre distinto. Compartir un archivo no es lo mismo que dar a cada miembro del equipo una copia privada, porque al tener un archivo compartido los cambios que cualquier miembro del equipo efectúan son visibles de inmediato para los otros miembros: sólo existe un archivo. Cuando se hacen copias de un archivo, los cambios subsecuentes que se hagan a una copia no afectarán a las demás.

Para ver cómo funciona LINK, consideremos la situación de la Fig. 1-14(a). Aquí hay dos usuarios, ast y jim, cada uno de los cuales tiene sus propios directorios con varios archivos. Si ahora ast ejecuta un programa que contiene la llamada al sistema

```
link( "usr/jim/memo", "/usr/ast/note
```

el archivo memo del directorio de jim se incluye ahora en el directorio de ast con el nombre note. En adelante, /usr/jim/memo y /usr/ast/note se referirán al mismo archivo.

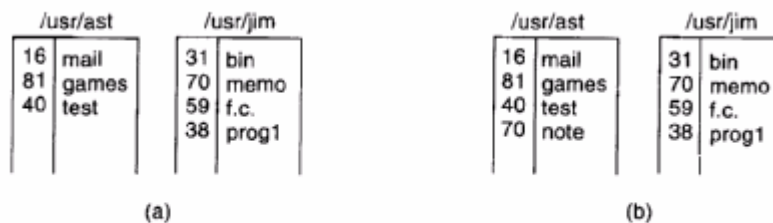


Figura 1-14. (a) Dos directorios antes de vincular /usr/jim/memo al directorio de ast.
(b) Los mismos directorios después de la vinculación.

Si entendemos cómo funciona LINK tal vez nos quedará más claro qué resultado produce. Todos los archivos en MINIX tienen un número único, su número-i, que lo identifica. Este número-i es un índice de una tabla de nodos-i, uno por archivo, que indican quién es el dueño del

archivo, dónde están sus bloques de disco, etc. Un directorio no es más que un archivo que contiene un conjunto de pares (número-i, nombre ASCII). En la Fig. 1-14, mail tiene el número-i 16, por ejemplo. Lo que LINK hace es simplemente crear una nueva entrada de directorio con un nombre (posiblemente nuevo) usando el número-i de un archivo existente. En la Fig. 1-14(b), dos entradas tienen el mismo número-i (70) y por tanto se refieren al mismo archivo. Si cualquiera de estas entradas se elimina posteriormente, usando la llamada al sistema UNLINK, la otra permanecerá. Si ambas se eliminan, MINIX verá que no existen entradas para el archivo (un campo del nodo-i indica cuántas entradas de directorio apuntan al archivo), y entonces lo eliminará del disco.

Como dijimos antes, la llamada al sistema MOUNT permite fusionar dos sistemas de archivos para formar uno solo. Una situación común es tener el **sistema de archivos raíz**, que contiene las versiones binarias (ejecutables) de los comandos comunes y otros archivos de uso intensivo, en el disco en RAM. El usuario podría después, por ejemplo, insertar en la unidad O un disco flexible con programas de usuario.

Si se ejecuta la llamada al sistema MOUNT, el sistema de archivos de la unidad O se puede conectar al sistema de archivos raíz, como se muestra en la Fig. 1-15. Una instrucción típica en C para realizar el montaje es

```
mount("/dev/fd0", "/mnt", 0);
```

donde el primer parámetro es el nombre de un archivo especial por bloques para la unidad O y el segundo parámetro es el lugar del árbol donde se debe montar.



Figura 1-15. Sistema de archivos (a) antes del montaje y (b) después del montaje.

Después de la llamada MOUNT, se puede acceder a un archivo en la unidad O usando su ruta desde el directorio raíz o desde el directorio de trabajo, sin importar en qué unidad esté. De hecho, se pueden montar una segunda, tercera y cuarta unidad en cualquier lugar del árbol. El comando MOUNT permite integrar medios removibles en una sola jerarquía de archivos integrada sin tener que preocuparse respecto a cuál es el dispositivo en el que está un archivo. Aunque en este ejemplo se habló de discos flexibles, también pueden montarse de este modo discos duros o porciones de discos duros (también llamados **particiones o dispositivos secundarios**). Si un sistema de archivos ya no se necesita, puede desmontarse con la llamada al sistema UNMOUNT.

MINIX mantiene un caché en la memoria principal con los bloques usados más recientemente a fin de no tener que leerlos del disco si se vuelven a usar en un lapso corto. Si se modifica un bloque del caché (por un WRITE a un archivo) y el sistema se cae antes de que el bloque modificado se grabe en el disco, el sistema de archivos se dañará. Para limitar el daño potencial, es importante

desalojar el caché periódicamente, de modo que la cantidad de datos que se pierdan en caso de una caída sea pequeño. La llamada al sistema SYNC le dice a MINIX que escriba en disco todos los bloques del caché que hayan sido modificados después de haberse leído. Cuando se inicia MINIX, se pone en marcha un programa llamado update (actualizar) que se ejecuta en segundo plano, invocando SYNC cada 30 segundos a fin de desalojar continuamente el caché.

Otras dos llamadas relacionadas con directorios son CHDIR y CHROOT. La primera cambia el directorio de trabajo y la segunda cambia el directorio raíz. Después de la llamada

```
chdir(“/usr/astltest”);
```

un OPEN ejecutado con el archivo xyz abrirá /usr/ast/test/xyz. CHROOT funciona de forma análoga. Una vez que un proceso le ha ordenado al sistema que cambie su directorio raíz, todos los nombres de ruta absolutos (los que comienzan con “/”) partirán de la nueva raíz. Sólo los superusuarios pueden ejecutar CHROOT, e incluso ellos no lo hacen con mucha frecuencia.

1.4.5 Llamadas al sistema para protección

En MINIX todos los archivos tienen un modo de 11 bits que se utiliza para protección. Nueve de estos bits son los de leer-escribir-ejecutar para el propietario, el grupo y otros. La llamada al sistema CHMOD hace posible cambiar el modo de un archivo. Por ejemplo, para hacer que un archivo sea sólo de lectura para todos excepto su propietario, podemos ejecutar

```
chmod(“archivo“, 0644);
```

Los otros dos bits de protección, 02000 y 04000, son los bits SETGID (establecer identificador de grupo) y SETUID (establecer identificador de usuario), respectivamente. Cuando cualquier usuario ejecuta un programa que tiene activado el bit SETUID, el uid efectivo de ese usuario se cambia al del propietario del archivo durante la ejecución del proceso. Esta capacidad se utiliza mucho para permitir a los usuarios ejecutar programas que realizan funciones exclusivas del superusuario, como crear directorios. Para crear un directorio se usa MKNOD, que es sólo para el superusuario. Si se hace que el propietario del programa mkdir sea el superusuario y que ese programa tenga el modo 04755, se podrá otorgar a los usuarios la facultad de ejecutar MKNOD pero en una forma muy restringida.

Cuando un proceso ejecuta un archivo que tiene el bit SETUID o SETGID activado en su modo, adquiere un uid o gid efectivo diferente de su uid o gid real. A veces es importante para un proceso averiguar cuáles son su uid o gid real y efectivo. Las llamadas al sistema GETUID y GETGID se incluyeron para proporcionar esa información. Cada llamada devuelve el uidlgid tanto real como efectivo, así que se necesitan cuatro rutinas de biblioteca para extraer la información apropiada: getuid, getgid, geteuid y getegid. Las primeras dos obtienen el uidlgid real, y las dos últimas, el uid/gid efectivo.

Los usuarios ordinarios no pueden cambiar su uid, excepto ejecutando programas con el bit SETUID activado, pero el superusuario tiene otra posibilidad: la llamada al sistema SETUID, que establece los uid tanto efectivo como real. SETGID establece ambos gid. El superusuario también puede cambiar el propietario de un archivo mediante la llamada al sistema CHOWN. En pocas

palabras, el superusuario tiene abundantes oportunidades para violar todas las reglas de protección, lo que explica por qué tantos estudiantes dedican tanto de su tiempo a tratar de convertirse en superusuarios.

Las últimas dos llamadas al sistema de esta categoría pueden ser ejecutadas por procesos de usuario ordinarios. La primera, `UMASK`, establece una máscara de bits interna dentro del sistema, la cual se usará para enmascarar los bits del modo cuando se cree un archivo. Después de la llamada

```
umask(022);
```

el modo proporcionado por `CREAT` y `MKNOD` tendrá los bits 022 enmascarados antes de usarse. Así, la llamada

```
creat(“archivo“, 0777);
```

asignará 0755 al modo en lugar de 0777. Puesto que la máscara de bits es heredada por los procesos hijo, si el shell ejecuta un `UMASK` justo después del inicio de sesión, ninguno de los procesos de usuario de esa sesión creará accidentalmente archivos en los que otras personas puedan escribir.

Cuando un programa propiedad de la raíz tiene el bit `SETUID` activado, puede acceder a cualquier archivo, pues su uid efectivo es el del superusuario. En muchos casos al programa le resulta útil saber si la persona que lo invocó tiene permiso para acceder a un archivo dado. Si el programa simplemente intenta acceder a él, siempre lo logrará, y no habrá averiguado nada.

Lo que se necesita es una forma de saber si el acceso está permitido para el uid real. La llamada al sistema `ACCESS` ofrece una forma de averiguarlo. El parámetro `mode` es 4 para verificar si hay acceso de lectura, 2 para acceso de escritura y 1 para acceso de ejecución. También se permiten combinaciones; por ejemplo, si `mode` es 6, la llamada devolverá 0 si el uid real tiene permiso tanto de escritura como de lectura; en caso contrario, devolverá `-1`. Si `mode` es 0, se verifica si existe el archivo y si se pueden efectuar búsquedas en los directorios que conducen a él.

1.4.6 Llamadas al sistema para administración del tiempo

MINIX tiene cuatro llamadas al sistema relacionadas con el reloj de hora del día. `TIME` simplemente devuelve el tiempo actual en segundos, donde 0 corresponde al lo. de enero de 1970 a la media noche (justo en el momento de comenzar el día, no al terminar). Desde luego, el reloj del sistema debe establecerse en algún momento para poder leerlo después, y es para ello que se ha incluido la llamada `STIME` que permite al superusuario establecer el reloj. La tercera llamada relacionada con el tiempo es `UTIME`, que permite al propietario de un archivo (o al superusuario) alterar el tiempo almacenado en el nodo-*i* de un archivo. La aplicación de esta llamada es muy limitada, pero unos cuantos programas la necesitan; por ejemplo, `touch`, que asigna el tiempo actual al tiempo del archivo.

Por último, tenemos `TIMES`, que devuelve la información de contabilidad a un proceso, a fin de que pueda saber cuánto tiempo de CPU usó directamente y cuánto tiempo de CPU gastó el sistema a su nombre (manejando sus llamadas al sistema). También se devuelven los tiempos de usuario y de sistema totales gastados por todos los hijos de ese proceso combinados.

1.5 ESTRUCTURA DEL SISTEMA OPERATIVO

Ahora que hemos visto qué aspecto tienen los sistemas operativos por fuera (es decir, la interfaz con el programador), ha llegado el momento de dar una mirada al interior. En las siguientes secciones examinaremos cuatro estructuras distintas que se han probado, a fin de tener una idea de la variedad de posibilidades. Éstas no son de ninguna manera las únicas estructuras posibles, pero nos darán una idea de algunos diseños que se han llevado a la práctica. Los cuatro diseños son los sistemas monolíticos, los sistemas por capas, las máquinas virtuales y los sistemas cliente-servidor.

1.5.1 Sistemas monolíticos

Este enfoque, que es por mucho la organización más común, bien podría subtitularse “El Gran Desorden”. La estructura consiste en que no hay estructura. El sistema operativo se escribe como una colección de procedimientos, cada uno de los cuales puede invocar a cualquiera de los otros cuando necesita hacerlo. Cuando se usa esta técnica, cada procedimiento del sistema tiene una interfaz bien definida en términos de parámetros y resultados, y cada uno está en libertad de invocar a cualquier otro, si este último realiza algún cálculo útil que el primero necesita.

Para construir el programa objeto real del sistema operativo cuando se adopta este enfoque, lo primero que se hace es compilar todos los procedimientos individuales, o archivos que contienen los procedimientos, y luego se vinculan en un solo archivo objeto usando el vinculador (linker) del sistema. En términos de ocultación de la información, prácticamente no hay tal; todos los procedimientos son visibles para todos los demás (en contraposición a una estructura que contiene módulos o paquetes, en la que gran parte de la información se oculta dentro de los módulos, y sólo se pueden invocar desde fuera del módulo los puntos de entrada designados oficialmente).

No obstante, incluso en los sistemas monolíticos es posible tener al menos un poco de estructura. Los servicios (llamadas al sistema) proporcionados por el sistema operativo se solicitan colocando los parámetros en lugares bien definidos, como en registros o en la pila, y ejecutando después una instrucción de trampa especial conocida como **llamada al kernel o llamada al supervisor**.

Esta instrucción conmuta la máquina del modo de usuario al modo de kernel y transfiere el control al sistema operativo, lo cual se muestra como evento (1) en la Fig. 1-16. (La mayor parte de las CPU tienen dos modos: modo de kernel, para el sistema operativo, en el que se permite todas las instrucciones; y modo de usuario, para programas de usuario, en el que no se permiten instrucciones de E/S y de ciertos otros tipos.)

A continuación, el sistema operativo examina los parámetros de la llamada para determinar cuál llamada al sistema se ejecutará; esto se muestra como (2) en la Fig. 1-16. Acto seguido, el sistema operativo consulta una tabla que contiene en la ranura k un apuntador al procedimiento que lleva a cabo la llamada al sistema k. Esta operación, marcada con (3) en la Fig. 1-16, identifica el procedimiento de servicio, mismo que entonces se invoca. Una vez que se ha completado el trabajo y la llamada al sistema ha terminado, se devuelve el control al programa de

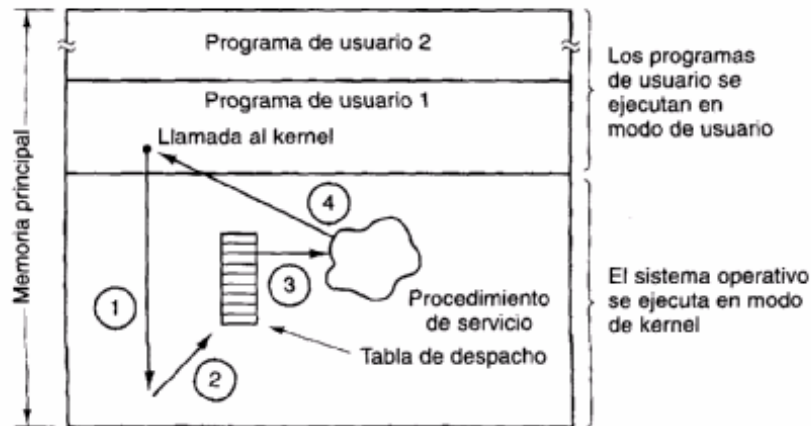


Figura 1-16. Cómo puede realizarse una llamada al sistema: (1) El programa de usuario entra en el kernel por una trampa. (2) El sistema operativo determina el número de servicio requerido. (3) El sistema operativo invoca el procedimiento de servicio. (4) Se devuelve el control al programa de usuario.

usuario (paso 4) a fin de que pueda continuar su ejecución con la instrucción que sigue a la llamada al sistema.

Esta organización sugiere una estructura básica para el sistema operativo:

1. Un programa principal que invoca el procedimiento de servicio solicitado.
2. Un conjunto de procedimientos de servicio que llevan a cabo las llamadas al sistema.
3. Un conjunto de procedimientos de utilería que ayudan a los procedimientos de servicio.

En este modelo, para cada llamada al sistema hay un procedimiento de servicio que se ocupa de ella. Los procedimientos de utilería hacen cosas que varios procedimientos de servicio necesitan, como obtener datos de los programas de usuario. Esta división de los procedimientos en tres capas se muestra en la Fig. 1-17.

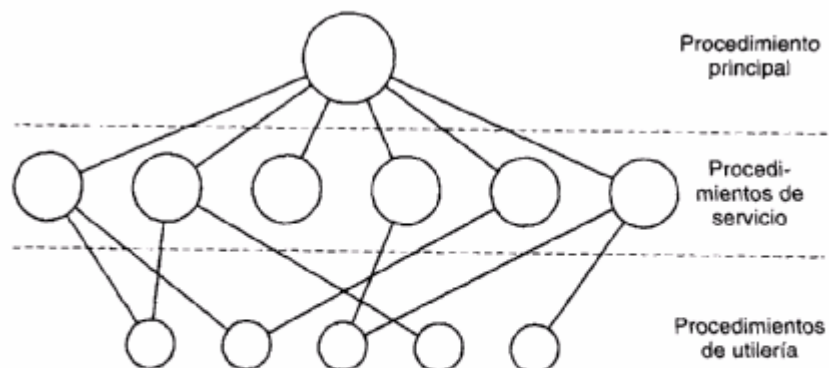


Figura 1-17. Modelo de estructuración sencillo para un sistema monolítico.

1.5.2 Sistemas por capas

Una generalización del enfoque de la Fig. 1-17 consiste en organizar el sistema operativo como una jerarquía de capas, cada una construida sobre la que está abajo de ella. El primer sistema que tuvo esta estructura fue el sistema THE construido en la Technische Hogeschool Eindhoven de los Países Bajos por E. W. Dijkstra (1968) y sus estudiantes. El sistema THE era un sencillo sistema por lotes para una computadora holandesa, la Electrologica X8, que tenía 32K de palabras de 27 bits (los bits eran costosos en esos tiempos).

El sistema tenía sE/S capas, como se muestra en la Fig. 1-18. La capa O se ocupaba del reparto del procesador, conmutando entre procesos cuando ocurrían interrupciones o expiraban temporizadores. Más arriba de la capa O, el sistema consistía en procesos secuenciales, cada uno de los cuales podía programarse sin tener que preocuparse por el hecho de que múltiples procesos se estuvieran ejecutando en un solo procesador. En otras palabras, la capa O se encargaba de la multiprogramación básica de la CPU.

Capa	Función
5	El operador
4	Programas de usuario
3	Administración de entrada/salida
2	Comunicación operador-proceso
1	Administración de memoria y tambor
0	Reparto del procesador y multiprogramación

Figura 1-18. Estructura del sistema operativo THE.

La capa 1 administraba la memoria, repartiendo espacio para los procesos en la memoria principal y en un tambor de 512K palabras que servía para contener partes de los procesos (páginas) para las que no había espacio en la memoria principal. Más arriba de la capa 1, los procesos no tenían que preocuparse por si estaban en la memoria o en el tambor; el software de esa capa se encargaba de que se colocaran en la memoria las páginas en el momento en que se necesitaban.

La capa 2 manejaba la comunicación entre cada proceso y la consola del operador. Por encima de esta capa cada proceso tenía efectivamente su propia consola de operador. La capa 3 se encargaba de administrar los dispositivos de E/S y de colocar en buffers las corrientes de información provenientes de y dirigidas a ellos. Más arriba de la capa 3 cada proceso podía tratar con dispositivos de E/S abstractos con propiedades bonitas, en lugar de dispositivos reales con muchas peculiaridades. En la capa 4 se encontraban los programas de usuario, los cuales no tenían que preocuparse por la administración de procesos, memoria, consola o E/S. El proceso del operador del sistema estaba en la capa 5.

Una forma más generalizada del concepto de capas estuvo presente en el sistema MULTICS. En vez de estar organizado en capas, MULTICS estaba organizado como una serie de anillos

concéntricos, siendo los interiores más privilegiados que los exteriores. Cuando un procedimiento de un anillo exterior quería invocar a uno de un anillo interior, tenía que emitir el equivalente de una llamada al sistema, es decir, ejecutar una instrucción TRAP cuyos parámetros se examinaban cuidadosamente para comprobar su validez antes de permitir que la llamada procediera. Aunque todo el sistema operativo formaba parte del espacio de direcciones de cada proceso de usuario en MULTICS, el hardware permitía designar procedimientos individuales (en realidad, segmentos de memoria) como protegidos contra lectura, escritura o ejecución.

En tanto el esquema por capas del sistema THE era en realidad sólo una ayuda para el diseño, ya que todas las partes del programa en última instancia se vinculaban en un solo programa objeto, en MULTICS el mecanismo de anillo estaba muy presente en el momento de la ejecución y el hardware obligaba a ajustarse a él. La ventaja del mecanismo de anillo es que fácilmente puede extenderse para estructurar los subsistemas de usuario. Por ejemplo, un profesor podría escribir un programa para probar y calificar los programas de los estudiantes y ejecutar este programa en el anillo n , con los programas de los estudiantes ejecutándose en el anillo $n + 1$ para que los estudiantes no puedan modificar sus calificaciones.

1.5.3 Máquinas virtuales

Las primeras versiones de OS/360 eran sistemas estrictamente por lotes. No obstante, muchos usuarios de 360 querían tener tiempo compartido, de modo que diversos grupos, tanto dentro como fuera de IBM, decidieron escribir sistemas de tiempo compartido para él. El sistema de tiempo compartido oficial de IBM, TSS/360, se entregó tarde, y cuando por fin llegó era tan grande y lento que pocos sitios realizaron la conversión. Finalmente, este sistema fue abandonado después de que su desarrollo había consumido unos 50 millones de dólares (Graham, 1970). Por otro lado, un grupo del Centro Científico de IBM en Cambridge, Massachusetts, produjo un sistema radicalmente diferente que IBM finalmente aceptó como producto, y que ahora se utiliza ampliamente en las macrocomputadoras IBM que quedan.

Este Sistema, llamado originalmente CP/CMS y que más adelante fue rebautizado como VM/370 (Seawright y MacKinnon, 1979), se basaba en una astuta observación: un sistema de tiempo compartido ofrece (1) multiprogramación y (2) una máquina extendida con una interfaz más cómoda que el hardware solo. La esencia de VM/370 consiste en separar por completo estas dos funciones.

El corazón del sistema, conocido como **monitor de máquina virtual**, se ejecuta en el hardware solo y realiza la multiprogramación, proporcionando no una, sino varias máquinas virtuales a la siguiente capa superior, como se muestra en la Fig. 1-19. Sin embargo, a diferencia de otros sistemas operativos, estas máquinas virtuales no son máquinas extendidas, con archivos y otras características bonitas; más bien, son copias exactas del hardware solo, incluido el modo de kernel/usuario, E/S, interrupciones y todo lo demás que la máquina real tiene.

Puesto que cada máquina virtual es idéntica al verdadero hardware, cada una puede ejecutar cualquier sistema operativo que se ejecutaría directamente en el hardware solo. Diferentes máquinas virtuales pueden, y con frecuencia lo hacen, ejecutar diferentes sistemas operativos. Algunos ejecutan uno de los descendientes de OS/360 para procesamiento por lotes o de transacciones,

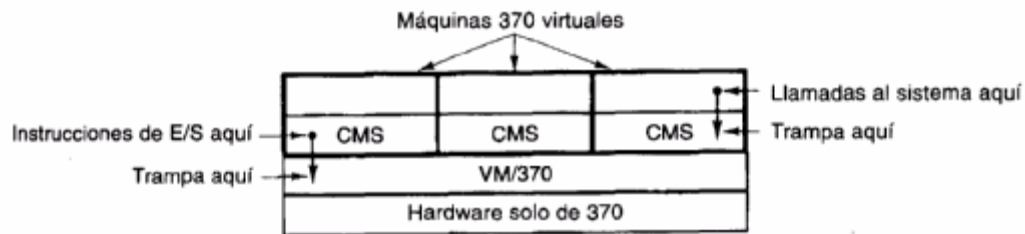


Figura 1-19. La estructura de VM/370 con CMS.

mientras que otros ejecutan un sistema interactivo monousuario llamado CMS (Sistema de Monitoreo de Conversaciones) para usuarios de tiempo compartido.

Cuando un programa de usuario ejecuta una llamada al sistema, la llamada se atrapa y se envía al sistema operativo de su propia máquina virtual, no al VM/370, tal como se haría si el programa se estuviera ejecutando en una máquina real en lugar de una virtual. A continuación, CMS emite las instrucciones de E/S del hardware normal para leer su disco virtual, o lo que sea que se necesite para llevar a cabo la llamada. Estas instrucciones de E/S son atrapadas por el VM/370, el cual, entonces, las ejecuta como parte de su simulación del hardware real. Al separar por completo las funciones de multiprogramación y de suministro de una máquina extendida, cada uno de los componentes puede ser mucho más sencillo, flexible y fácil de mantener.

El concepto de máquina virtual se usa mucho hoy día en un contexto diferente: la ejecución de viejos programas para MS-DOS en una Pentium (u otra CPU Intel de 32 bits). Al diseñar la Pentium y su software, tanto Intel como Microsoft se dieron cuenta de que habría una gran demanda por ejecutar software viejo en el nuevo hardware. Por esta razón, Intel proveyó un modo 8086 virtual en la Pentium. En este modo, la máquina actúa como una 8086 (que es idéntica a una 8088 desde la perspectiva del software) incluido direccionamiento de 16 bits con un límite de 1 MB.

Este modo es utilizado por WINDOWS, OS2 y otros sistemas operativos para ejecutar programas de MS-DOS. Estos programas se inician en modo 8086 virtual y, en tanto ejecuten instrucciones normales, operan en el hardware solo. Sin embargo, cuando un programa trata de entrar por una trampa al sistema operativo para efectuar una llamada al sistema, o trata de realizar E/S protegida directamente, ocurre una trampa al monitor de la máquina virtual.

Hay dos posibles variantes de este diseño. En la primera, se carga el MS-DOS mismo en el espacio de direcciones de la 8086 virtual, de modo que el monitor de la máquina virtual simplemente refleja la trampa de vuelta a MS-DOS, tal como ocurriría en una 8086 real. Después, cuando MS-DOS trata de efectuar la E/S él mismo, esa operación es atrapada y llevada a cabo por el monitor de la máquina virtual.

En la otra variante, el monitor de la máquina virtual simplemente atrapa la primera trampa y efectúa la E/S él mismo, ya que conoce todas las llamadas al sistema de MS-DOS y, por tanto, sabe lo que se supone que debe hacer cada trampa. Esta variante es menos pura que la primera, ya que sólo emula MS-DOS correctamente, y no otros sistemas operativos, como hace la primera. Por otro lado, esta variante es mucho más rápida, ya que no tiene que iniciar MS-DOS para llevar a cabo la

E/S. Una desventaja adicional de ejecutar realmente MS-DOS en modo 8086 virtual es que MS-DOS se mete mucho con el bit que habilita/inhabilita las interrupciones, y la emulación de esto resulta muy costosa.

Vale la pena señalar que ninguno de estos enfoques es realmente idéntico al de VM/370, ya que la máquina que está siendo emulada no es una Pentium completa, sino sólo una 8086. En el sistema VM/370 es posible ejecutar el VM/370 mismo en la máquina virtual. Con la Pentium, no es posible ejecutar, digamos, WINDOWS en la 8086 virtual porque ninguna versión de WINDOWS se ejecuta en una 8086; 286 es lo mínimo incluso para la versión más antigua, y no se proporciona emulación de 286 (y mucho menos de Pentium).

Con VM/T70, cada proceso de usuario obtiene una copia exacta de la computadora real; con el modo 8086 virtual en la Pentium, cada proceso usuario obtiene una copia exacta de una computadora distinta. Yendo un paso más allá, investigadores en el M.I.T. han construido un sistema que proporciona a cada usuario un clon de la computadora real, pero con un subconjunto de los recursos (Engler et al., 1995). Así, una máquina virtual podría obtener los bloques de disco 0 a 1023, el siguiente podría obtener los bloques 1024 a 2047, etcétera.

En la capa más baja, ejecutándose en modo de kernel, hay un programa llamado **exokernel**, cuyo trabajo consiste en repartir recursos a las máquinas virtuales y luego verificar los intentos por usarlos para asegurarse de que ninguna máquina esté tratando de usar los recursos de alguien más. Cada máquina virtual en el nivel de usuario puede ejecutar su propio sistema operativo, como en VM/370 y el modo 8086 virtual de la Pentium, excepto que cada uno sólo puede usar los recursos que ha solicitado y que le han sido asignados.

La ventaja del esquema de exokernel es que ahorra una capa de mapeo. En otros diseños, cada máquina virtual cree que tiene su propio disco, cuyos bloques van desde 0 hasta algún máximo, de modo que el monitor de máquina virtual debe mantener tablas para establecer la correspondencia con las direcciones de disco reales (y todos los demás recursos). Con el exokernel, no es necesario realizar este segundo mapeo. Lo único que el exokernel tiene que hacer es mantenerse al tanto de qué recursos se han asignado a cada máquina virtual. Este método sigue teniendo la ventaja de separar la multiprogramación (en el exokernel) y el código de sistema operativo del usuario (en el espacio del usuario), pero el gasto extra es menor, ya que todo lo que el exokernel tiene que hacer es evitar que las máquinas virtuales tomen cosas que no les pertenecen.

1.5.4 Modelo cliente-servidor

VM/370 se simplifica mucho al desplazar una parte importante del código de sistema operativo tradicional (que implementa la máquina extendida) a una capa superior, CMS. No obstante, el VM/370 en sí sigue siendo un programa complejo porque simular varias 370 virtuales no es tan fácil (sobre todo si se desea hacerlo con una eficiencia razonable).

Una tendencia en los sistemas operativos modernos es llevar aún más lejos esta idea de trasladar código a capas superiores y quitarle lo más que se pueda al sistema operativo, dejando un kernel mínimo. El enfoque usual consiste en implementar la mayor parte de las funciones del sistema operativo en procesos de usuario. Para solicitar un servicio, como leer un bloque de un

archivo, un proceso de usuario (ahora llamado proceso cliente) envía la solicitud a un **proceso servidor**, el cual realiza el trabajo y devuelve la respuesta.

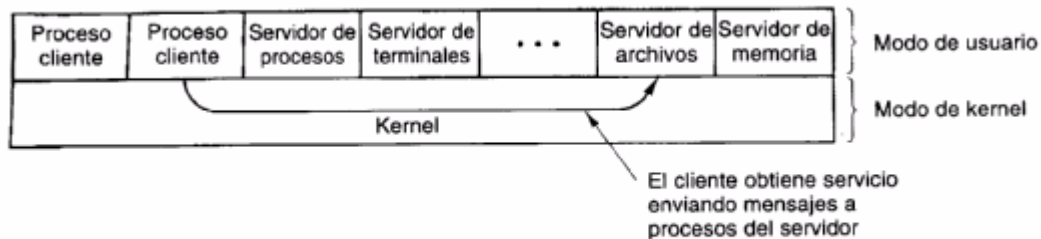


Figura 1-20. El modelo cliente-servidor.

En este modelo, que se muestra en la Fig. 1-20, lo único que el kernel hace es manejar la comunicación entre los clientes y los servidores. Al dividir el sistema operativo en partes, cada una de las cuales sólo se encarga de una faceta del sistema, como el servicio de archivos, de procesos, de terminales o de memoria, cada parte puede ser pequeña y manejable. Además, dado que todos los servidores se ejecutan como procesos en modo de usuario, y no en modo de kernel, no tienen acceso directo al hardware. Por tanto, si se activa un error en el servidor de archivos, es posible que el servicio de archivos se caiga, pero normalmente esto no hará que se caiga toda la máquina.

Otra ventaja del modelo cliente-servidor es su adaptabilidad para usarse en sistemas distribuidos (véase la Fig. 1-21). Si un cliente se comunica con un servidor enviándole mensajes, el cliente no necesita saber si el mensaje será atendido localmente en su propia máquina o si se envió a través de la red a un servidor en una máquina remota. En lo que al cliente concierne, sucede lo mismo en ambos casos: se envía una solicitud y se obtiene una respuesta.

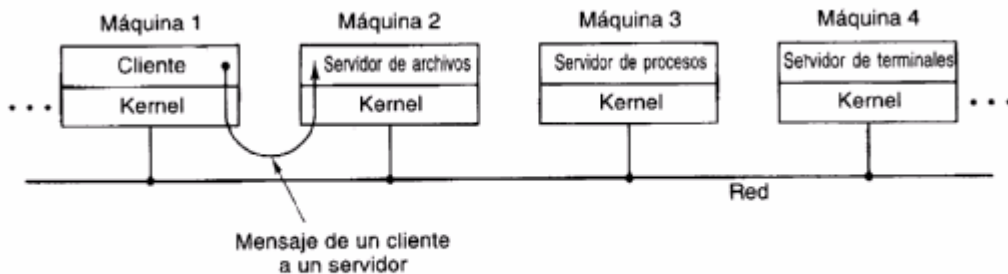


Figura 1-21. El modelo cliente-servidor en un sistema distribuido.

La imagen que acabamos de presentar de un kernel que sólo se encarga del transporte de mensajes de los clientes a los servidores y de regreso no es del todo realista. Algunas funciones del sistema operativo (como cargar comandos en los registros de los dispositivos de E/S físicos) son difíciles, si no imposibles, de efectuar desde programas del espacio de usuarios. Hay dos formas de resolver este problema. Una consiste en hacer que algunos procesos de servidor críticos

(p. ej., los controladores de dispositivos de E/S) se ejecuten realmente en modo de kernel, con acceso completo a todo el hardware, pero que se comuniquen con los demás procesos empleando el mecanismo de mensajes normal.

La otra solución consiste en incorporar una cantidad mínima de **mecanismo** en el kernel pero dejar las decisiones de **política** a los servidores en el espacio de usuarios. Por ejemplo, el kernel podría darse cuenta de que un mensaje enviado a cierta dirección especial implica que debe tomarse el contenido de ese mensaje y cargarlo en los registros de dispositivo de E/S de algún disco, a fin de iniciar una lectura de disco. En este ejemplo, el kernel ni siquiera inspeccionaría los bytes del mensaje para verificar si son válidos o significativos; sólo los copiaría ciegamente en los registros del dispositivo de disco. (Obviamente, debe utilizarse algún esquema para limitar tales mensajes sólo a los procesos autorizados.) La división entre mecanismo y política es un concepto importante; aparece una y otra vez en los sistemas operativos en diferentes contextos.

1.6 BOSQUEJO DEL RESTO DEL LIBRO

Los sistemas operativos por lo regular tienen cuatro componentes principales: administración de procesos, administración de dispositivos de E/S, administración de memoria y administración de archivos. MINIX también está dividido en estas cuatro partes. Los siguientes cuatro capítulos se ocupan de estos cuatro temas, uno por capítulo. El capítulo 6 es una lista de lecturas sugeridas y una bibliografía.

Los capítulos sobre procesos, E/S, administración de memoria y sistemas de archivos tienen la misma estructura general. Primero se presentan los principios generales del tema. Luego viene una reseña del área correspondiente de MINIX (que también aplica a UNIX). Por último, se analiza con detalle la implementación en MINIX. Los lectores a los que sólo les interesan los principios generales de los sistemas operativos y no el código de MINIX pueden pasar por alto o sólo hojear la sección de implementación sin pérdida de continuidad. Los lectores a los que sí les interesa averiguar cómo funciona un sistema operativo real (MINIX) deberán leer todas las secciones.

1.7 RESUMEN

Los sistemas operativos pueden verse desde dos perspectivas: administradores de recursos y máquinas extendidas. Desde la perspectiva de administrador de recursos, la tarea del sistema operativo es administrar con eficiencia las diferentes partes del sistema. Desde la perspectiva de máquina extendida, la tarea del sistema operativo es proporcionar a los usuarios una máquina virtual que sea más cómoda de usar que la máquina real.

Los sistemas operativos tienen una historia larga, que se remonta a los días en que sustituían al operador y llega hasta los sistemas de multiprogramación actuales.

El corazón de cualquier sistema operativo es el conjunto de llamadas al sistema que puede manejar. Éstas nos dicen qué es lo que realmente hace el sistema operativo. En el caso de MINIX, las llamadas al sistema se pueden dividir en sE/S grupos. El primer grupo tiene que ver con la

creación y terminación de procesos. El segundo grupo maneja las señales. El tercer grupo sirve para leer y escribir archivos. Un cuarto grupo se encarga de la administración de directorios. El quinto grupo protege la información, y el sexto grupo se ocupa de llevar el tiempo.

Los sistemas operativos se pueden estructurar de diversas maneras. Las más comunes son como sistema monolítico, como una jerarquía de capas, como sistema de máquina virtual y usando el modelo cliente-servidor.

PROBLEMAS

1. Señale las dos funciones principales de un sistema operativo.
2. ¿Qué es la multiprogramación?
3. ¿Qué es el *spooling*? ¿Cree usted que las computadoras personales avanzadas contarán con *spooling* como capacidad estándar en el futuro?
4. En las primeras computadoras, cada byte de datos leído o escrito era manejado directamente por la CPU (es decir, no había DMA: acceso directo a memoria). ¿Qué implicaciones tiene esta organización para la multiprogramación?
5. ¿Por qué no era muy común el tiempo compartido en las computadoras de la segunda generación?
6. ¿Cuáles de las siguientes instrucciones sólo deben permitirse en modo de kernel?
 - (a) Inhabilitar todas las interrupciones.
 - (b) Leer el reloj de hora del día.
 - (c) Establecer el reloj de hora del día.
 - (d) Cambiar el mapa de memoria.
7. Cite algunas diferencias entre los sistemas operativos de las computadoras personales y los de las macrocomputadoras.
8. Un archivo MINIX cuyo propietario tiene $uid = 12$ y $gid = 1$ tiene el modo $rwxr-x---$. Otro usuario con $uid = 6$, $gid = 1$ trata de ejecutar el archivo. ¿Qué sucede?
9. En vista del hecho de que la simple existencia de un superusuario puede dar pie a todo tipo de problemas de seguridad, ¿por qué existe tal concepto?
10. El modelo cliente-servidor es popular en los sistemas distribuidos. ¿Puede usarse también en los sistemas de una sola computadora?
11. ¿Por qué se necesita la tabla de procesos en un sistema de tiempo compartido? ¿Se necesita también en los sistemas de computadora personal, en los que sólo existe un proceso, el cual se apodera de toda la máquina hasta terminar?
12. Señale la diferencia esencial que hay entre un archivo especial por bloques y un archivo especial por caracteres.
13. En MINIX, si el usuario 2 se vincula con un archivo propiedad del usuario 1, y luego el usuario 1 elimina el archivo, ¿qué sucede cuando el usuario 2 trata de leer el archivo?

14. ¿Por qué está limitada al superusuario la llamada al sistema CHROOT? (Sugerencia: Piense en los problemas de protección.)
15. ¿Por qué tiene MINIX el programa update ejecutándose en segundo plano todo el tiempo?
16. ¿Hay algún caso en que tenga sentido ignorar la señal SIGALRM?
17. Escriba un programa (o una serie de programas) para probar todas las llamadas al sistema de MINIX. Para cada llamada, pruebe diversos conjuntos de parámetros, incluidos algunos incorrectos, para ver si son detectados.
18. Escriba un shell similar al de la Fig. 1-10 pero que contenga suficiente código como para que funcione realmente y se pueda probar. También podrían agregársele algunas capacidades como redirección de entradas y salidas, conductos y trabajos en segundo plano.

2

PROCESOS

Estamos a punto de emprender un estudio detallado de la forma en que se diseñan y construyen los sistemas operativos en general y MINIX en particular. El concepto central de cualquier sistema operativo es el proceso: una abstracción de un programa en ejecución. Todo lo demás gira alrededor de este concepto, y es importante que el diseñador (y el estudiante) de sistemas operativos sepa lo antes posible qué es un proceso.

2.1 INTRODUCCIÓN A LOS PROCESOS

Todas las computadoras modernas pueden hacer varias cosas al mismo tiempo. Mientras ejecuta un programa de usuario, una computadora también puede estar leyendo de un disco y enviando texto a una pantalla o impresora. En un sistema de multiprogramación, la CPU también conmuta de un programa a otro, ejecutando cada uno durante decenas o centenas de milisegundos. Si bien, estrictamente hablando, en un instante dado la CPU está ejecutando sólo un programa, en el curso de un segundo puede trabajar con varios programas, dando a los usuarios la ilusión de paralelismo. A veces se usa el término seudoparalelismo para referirse a esta rápida conmutación de la CPU entre programas, para distinguirla del verdadero paralelismo de hardware de los sistemas multiprocesador (que tienen dos o más CPU que comparten la misma memoria física). Para el ser humano es difícil seguir la pista a múltiples actividades paralelas. Por ello, los diseñadores de sistemas operativos han desarrollado a lo largo de los años un modelo (procesos secuenciales) que facilita el manejo del paralelismo. Ese modelo y sus usos son el tema de este capítulo.

2.1.1 El modelo de procesos

En este modelo, todo el software ejecutable de la computadora, lo que a menudo incluye al sistema operativo, está organizado en una serie de **procesos secuenciales**, o simplemente **procesos**. Un proceso no es más que un programa en ejecución, e incluye los valores actuales del contador de programa, los registros y las variables. Conceptualmente, cada uno de estos procesos tiene su propia CPU virtual. Desde luego, en la realidad la verdadera CPU conmuta de un proceso a otro, pero para entender el sistema es mucho más fácil pensar en una colección de procesos que se ejecutan en (seudo) paralelo que tratar de seguir la pista a la forma en que la CPU conmuta de un programa a otro. Esta rápida conmutación se denomina multiprogramación, como vimos en el capítulo anterior.

En la Fig. 2-1(a) vemos una computadora multiprogramando dos programas en la memoria. En la Fig. 2-1(b) vemos cuatro procesos, cada uno con su propio flujo de control (esto es, su propio contador de programa), ejecutándose con independencia de los otros. En la Fig. 2-1(c) vemos que si el intervalo de tiempo es suficientemente largo, todos los procesos avanzan, pero en un instante dado sólo un proceso se está ejecutando realmente.

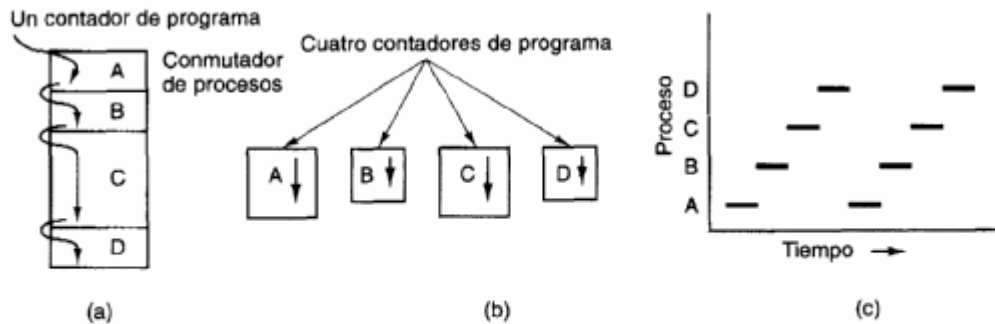


Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo un programa está activo en un instante dado.

Con la CPU conmutando entre los procesos, la rapidez con que un proceso realiza sus cálculos no es uniforme, y probablemente ni siquiera reproducible si los mismos procesos se ejecutan otra vez. Por tanto, los procesos no deben programarse basándose en supuestos acerca de los tiempos. Considere, por ejemplo, un proceso de E/S que inicia una cinta continua para restaurar archivos respaldados, ejecuta un ciclo vacío 10 000 veces para permitir que la cinta alcance la velocidad de trabajo y luego emite un comando para leer el primer registro. Si la CPU decide conmutar a otro proceso durante el ciclo vacío, es posible que los procesos de cinta no se ejecuten otra vez sino hasta después de que el primer registro haya pasado por la cabeza de lectura. Cuando un proceso tiene requisitos de tiempo real críticos como éste, es decir, cuando ciertos eventos deben

ocurrir en cierto número de milisegundos, se deben tomar medidas especiales para asegurar que así ocurran. Normalmente, empero, los procesos no resultan afectados por la multiprogramación subyacente de la CPU ni las velocidades relativas de los diferentes procesos.

La diferencia entre un programa y un proceso es sutil, pero crucial. Tal vez una analogía ayude a aclarar este punto. Consideremos un computólogo con inclinaciones gastronómicas que está preparando un pastel de cumpleaños para su hija. Él cuenta con una receta para pastel de cumpleaños y una cocina bien abastecida de las entradas necesarias: harina, huevos, azúcar, extracto de vainilla, etc. En esta analogía, la receta es el programa (es decir, un algoritmo expresado en alguna notación apropiada), el computólogo es el procesador (CPU) y los ingredientes del pastel son los datos de entrada. El proceso es la actividad de nuestro pastelero consistente en leer la receta, obtener los ingredientes y hornear el pastel.

Imaginemos ahora que el hijo del computólogo llega corriendo y llorando, diciendo que le picó una abeja. El computólogo registra el punto en que estaba en la receta (guarda el estado del proceso actual), saca un libro de primeros auxilios, y comienza a seguir las instrucciones que contiene. Aquí vemos cómo el procesador se conmuta de un proceso (hornear) a un proceso de más alta prioridad (administrar cuidados médicos), cada uno con un programa diferente (receta vs. libro de primeros auxilios). Una vez que se ha atendido la picadura de abeja, el computólogo regresa a su pastel, continuando en el punto donde había interrumpido.

La idea clave aquí es que un proceso es una actividad de algún tipo: tiene programa, entrada, salida y un estado. Se puede compartir un procesador entre varios procesos, usando algún algoritmo de planificación para determinar cuándo debe dejarse de trabajar en un proceso para atender a uno distinto.

Jerarquías de procesos

Los sistemas operativos que manejan el concepto de proceso deben contar con algún mecanismo para crear todos los procesos necesarios. En los sistemas muy sencillos, o en los diseñados para ejecutar sólo una aplicación (p. ej., controlar un dispositivo en tiempo real), es posible que, cuando el sistema se inicia, todos los procesos que puedan necesitarse estén presentes. Sin embargo, en la mayor parte de los sistemas se necesita algún mecanismo para crear y destruir procesos según sea necesario durante la operación. En MINIX, los procesos se crean con la llamada al sistema FORK (bifurcar), que crea una copia idéntica del proceso invocador. El proceso hijo también puede ejecutar FORK, así que es posible tener un árbol de procesos. En otros sistemas operativos existen llamadas al sistema para crear un proceso, cargar su memoria y ponerlo a ejecutar. Sea cual sea la naturaleza exacta de la llamada al sistema, los procesos necesitan poder crear otros procesos. Observe que cada proceso tiene un padre, pero cero, uno, dos o más hijos.

Como ejemplo sencillo del uso de los árboles de procesos, veamos la forma en que MINIX se inicializa a sí mismo cuando se pone en marcha. Un proceso especial, llamado `mit`, está presente en la imagen de arranque. Cuando este proceso comienza a ejecutarse, lee un archivo que le indica cuántas terminales hay, y luego bifurca un proceso nuevo por terminal. Estos procesos esperan a que alguien inicie una sesión. Si un inicio de sesión (`login`) tiene éxito, el proceso de

inicio de sesión ejecuta un shell para aceptar comandos. Estos comandos pueden iniciar más procesos, y así sucesivamente. Por tanto, todos los procesos del sistema pertenecen a un mismo árbol, que tiene a mit en su raíz. (El código de mit no se lista en este libro, y tampoco el del shell. Había que poner un límite en algún lado.)

Estados de procesos

Aunque cada proceso es una entidad independiente, con su propio contador de programa y estado interno, los procesos a menudo necesitan interactuar con otros procesos. Un proceso podría generar ciertas salidas que otro proceso utiliza como entradas. En el comando de shell

```
cat capítulo1 capítulo2 capítulo3 grep árbol
```

el primer proceso, que ejecuta cat, concatena y envía a la salida tres archivos. El segundo proceso, que ejecuta grep, selecciona todas las líneas que contienen la palabra “árbol”. Dependiendo de las velocidades relativas de los dos procesos (que a su vez dependen de la complejidad relativa de los programas y de cuánto tiempo de CPU ha ocupado cada uno), puede suceder que grep esté listo para ejecutarse, pero no haya entradas esperando ser procesadas por él. En tal caso, grep deberá **bloquearse** hasta que haya entradas disponibles.

Cuando un proceso se bloquea, lo hace porque le es imposible continuar lógicamente, casi siempre porque está esperando entradas que todavía no están disponibles. También puede ser que un programa que conceptualmente está listo y en condiciones de ejecutarse sea detenido porque el sistema operativo ha decidido asignar la CPU a otro proceso durante un tiempo. Estas dos condiciones son totalmente distintas. En el primer caso, la suspensión es inherente al problema (no es posible procesar la línea de comandos del usuario antes de que éste la teclee). En el segundo caso, se trata de un tecnicismo del sistema (no hay suficientes CPU para darle a cada proceso su propio procesador privado). En la Fig. 2-2 vemos un diagrama de estados que muestra los tres estados en los que un proceso puede estar:

1. Ejecutándose (usando realmente la CPU en ese instante).
2. Listo (se puede ejecutar, pero se suspendió temporalmente para dejar que otro proceso se ejecute).
3. Bloqueado (no puede ejecutarse en tanto no ocurra algún evento externo).

Lógicamente, los dos primeros estados son similares. En ambos casos el proceso está dispuesto a ejecutarse, sólo que en el segundo temporalmente no hay una CPU a su disposición. El tercer estado es diferente de los primeros dos en cuanto a que el proceso no puede ejecutarse, incluso si la CPU no tiene nada más que hacer.

Puede haber cuatro transiciones entre estos tres estados, como se muestra. La transición 1 ocurre cuando un proceso descubre que no puede continuar. En algunos sistemas el proceso debe ejecutar una llamada al sistema, BLOCK, para pasar al estado bloqueado. En otros sistemas, incluido MINIX, cuando un proceso lee de un conducto o de un archivo especial (p. ej., una terminal) y no hay entradas disponibles, se bloquea automáticamente.

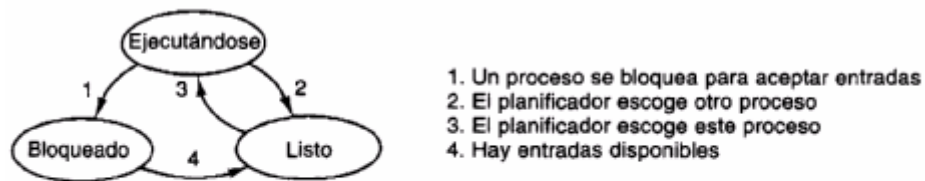


Figura 2-2. Un proceso puede estar en el estado de ejecutándose, bloqueado o listo. Las transiciones entre estos tres estados son las que se muestran.

Las transiciones 2 y 3 son causadas por el planificador de procesos, una parte del sistema operativo, sin que el proceso se entere siquiera de ellas. La transición 2 ocurre cuando el planificador decide que el proceso en ejecución ya se ejecutó durante suficiente tiempo y es hora de dejar que otros procesos tengan algo de tiempo de CPU. La transición 3 ocurre cuando todos los demás procesos han disfrutado de una porción justa y es hora de que el primer proceso reciba otra vez la CPU para ejecutarse. El tema de la planificación, es decir, de decidir cuál proceso debe ejecutarse cuándo y durante cuánto tiempo, es muy importante; lo examinaremos más adelante en este capítulo. Se han inventado muchos algoritmos para tratar de equilibrar las exigencias opuestas de eficiencia del sistema global y de equitatividad para los procesos individuales.

La transición 4 ocurre cuando acontece el suceso externo que un proceso estaba esperando (como la llegada de entradas). Si ningún otro proceso se está ejecutando en ese instante, se dispara de inmediato la transición 3, y el proceso comienza a ejecutarse. En caso contrario, el proceso tal vez tenga que esperar en el estado listo durante cierto tiempo hasta que la CPU esté disponible.

Usando el modelo de procesos, es mucho más fácil visualizar lo que está sucediendo dentro del sistema. Algunos de los procesos ejecutan programas que llevan a cabo comandos tecleados por un usuario. Otros procesos forman parte del sistema y se encargan de tareas tales como atender solicitudes de servicios de archivos o manejar los detalles de la operación de una unidad de disco o de cinta. Cuando ocurre una interrupción de disco, el sistema toma la decisión de dejar de ejecutar el proceso en curso y ejecutar el proceso de disco, que estaba bloqueado esperando dicha interrupción. Así, en lugar de pensar en interrupciones, podemos pensar en procesos de usuario, procesos de disco, procesos de terminal, etc., que se bloquean cuando están esperando que algo suceda. Cuando se ha leído el bloque de disco o se ha tecleado el carácter, el proceso que lo estaba esperando se desbloquea y es elegible para ejecutarse otra vez.

Esta perspectiva da lugar al modelo que se muestra en la Fig. 2-3. Aquí, el nivel más bajo del sistema operativo es el planificador, con diversos procesos arriba de él. Todo el manejo de interrupciones y los detalles del inicio y la detención de procesos están ocultos en el planificador, que en realidad es muy pequeño. El resto del sistema operativo está estructurado en forma de procesos. El modelo de la Fig. 2-3 se emplea en MINIX, con el entendido de que “planificador” no sólo se refiere a planificación de procesos, sino también a manejo de interrupciones y toda la comunicación entre procesos. No obstante, como una primera aproximación, sí muestra la estructura básica.

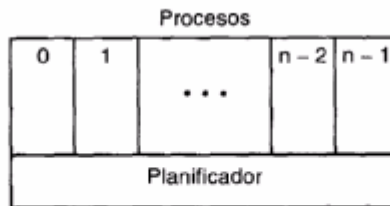


Figura 2-3. La capa más baja de un sistema operativo con estructura de procesos maneja las interrupciones y la planificación. Por encima de esa capa están los procesos secuenciales.

2.1.2 Implementación de procesos

Para implementar el modelo de procesos, el sistema operativo mantiene una tabla (un arreglo de estructuras) llamada **tabla de procesos**, con una entrada por cada proceso. Esta entrada contiene información acerca del estado del proceso, su contador de programa, el apuntador de pila, el reparto de memoria, la situación de sus archivos abiertos, su información de contabilidad y planificación y todos los demás aspectos de un proceso que se deben guardar cuando éste se conmuta del estado ejecutándose al estado listo, a fin de poder reiniciarlo después como si nunca se hubiera detenido.

En MINIX la administración de procesos, de memoria y de archivos corren a cargo de módulos individuales dentro del sistema, por lo que la tabla de procesos se divide en particiones y cada módulo mantiene los campos que necesita. En la Fig. 2-4 se muestran algunos de los campos más importantes. Los campos de la primera columna son los únicos pertinentes para este capítulo. Las otras dos columnas se incluyen sólo para dar una idea de qué información se necesita en otras partes del sistema.

Ahora que hemos examinado la tabla de procesos, podemos explicar un poco más la forma de cómo se mantiene la ilusión de múltiples procesos secuenciales en una máquina con una CPU y muchos dispositivos de E/S. Lo que sigue es técnicamente una descripción de cómo funciona el “planificador” de la Fig. 2-3 en MINIX, pero la mayor parte de los sistemas operativos modernos funcionan básicamente de la misma manera. Cada clase de dispositivo de E/S (p. ej., discos flexibles, discos duros, cronómetros, terminales) tiene asociada una posición cerca de la base de la memoria llamada **vector de interrupción** que contiene la dirección del procedimiento de servicio de interrupciones. Supongamos que el proceso de usuario 3 se está ejecutando cuando ocurre una interrupción de disco. El hardware de interrupciones mete el contador de programa, la palabra de estado del programa y quizá uno o más registros en la pila (actual). A continuación, la computadora salta a la dirección especificada en el vector de interrupciones de disco. Esto es todo lo que el hardware hace. De aquí en adelante, el software decide lo que se hace.

Lo primero que hace el procedimiento de servicio de interrupciones es guardar todos los registros de la entrada de tabla de procesos para el proceso actual. El número del proceso actual y un apuntador a su entrada de la tabla se guardan en variables globales a fin de poder encontrarlos rápidamente. Luego se saca de la pila la información depositada en ella por la interrupción, y se ajusta el apuntador de la pila de modo que apunte a una pila temporal empleada por el manejador

Administración de procesos	Administración de memoria	Administración de archivos
Registros	Apuntador al segmento de texto	Máscara UMASK
Contador de programa	Apuntador al segmento de datos	Directorio raíz
Palabra de estado del programa	Apuntador al segmento bss	Directorio de trabajo
Apuntador a la pila	Estado de salida	Descriptores de archivos
Estado del proceso	Estado de señales	Uid efectivo
Hora en que se inició el proceso	Identificador del proceso	Gid efectivo
Tiempo de CPU usado	Proceso padre	Parámetros de llamadas al sistema
Tiempo de CPU de los hijos	Grupo de procesos	Diversos bits de bandera
Hora de la siguiente alarma	Uid real	
Apuntadores a la cola de mensajes	Uid efectivo	
Bits de señales pendientes	Gid real	
Identificador del proceso	Gid efectivo	
Diversos bits de bandera	Mapas de bits para señales	
	Diversos bits de bandera	

Figura 2-4. Algunos de los campos de la tabla de procesos de MINIX.

de procesos. Las acciones como guardar los registros y ajustar el apuntador de la pila ni siquiera pueden expresarse en C, así que son realizadas por una pequeña rutina escrita en lenguaje de ensamblador. Una vez que esta rutina termina, invoca a un procedimiento en C para que se encargue del resto del trabajo.

La comunicación entre procesos en MINIX se efectúa a través de mensajes, así que el siguiente paso consiste en construir un mensaje para enviarlo al proceso de disco, el cual estará bloqueado en espera de recibirlo. El mensaje dice que ocurrió una interrupción, a fin de distinguirlo de los mensajes de usuario que solicitan la lectura de bloques de disco y cosas por el estilo. Ahora se cambia el estado del proceso de disco de bloqueado a listo y se llama al planificador. En MINIX, los diferentes procesos tienen diferentes prioridades, con objeto de dar mejor servicio a los manejadores de dispositivos de E/S que a los procesos de usuario. Si el proceso de disco es ahora el proceso ejecutable con la prioridad más alta, se planificará para ejecutarse. Si el proceso que se interrumpió tiene la misma importancia o una mayor, se le planificará para ejecutarse otra vez, y el proceso de disco tendrá que esperar un poco.

En cualquier caso, ahora regresa el procedimiento en C invocado por el código de interrupción en lenguaje de ensamblador, y este código carga los registros y el mapa de memoria para el proceso que ahora es el actual, y lo pone en marcha. El manejo de interrupciones y la planificación se resumen en la Fig. 2-5. Vale la pena señalar que los detalles varían un poco de un sistema a otro.

2.1.3 Hilos

En un proceso tradicional del tipo que acabamos de estudiar hay un solo hilo de control y un solo contador de programa en cada proceso. Sin embargo, algunos sistemas operativos modernos

1. El hardware agrega a la pila el contador de programa, etc.
2. El hardware carga un nuevo contador de programa del vector de interrupciones.
3. El procedimiento en lenguaje ensamblador guarda los registros.
4. El procedimiento en lenguaje ensamblador prepara una nueva pila.
5. Se ejecuta el servicio de interrupciones en C (por lo regular lee y coloca en buffers las entradas).
6. El planificador marca la tarea en espera como lista.
7. El planificador decide cuál proceso debe ejecutarse a continuación.
8. El procedimiento en C regresa al código en ensamblador.
9. El procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

Figura 2-5. Bosquejo de lo que el nivel más bajo del sistema operativo hace cuando ocurre una interrupción.

manejan múltiples hilos de control dentro de un proceso. Estos hilos de control normalmente se llaman sólo **hilos** u, ocasionalmente, **procesos ligeros**.

En la Fig. 2-6(a) vemos tres procesos tradicionales. Cada proceso tiene su propio espacio de direcciones y un solo hilo de control. En contraste, en la Fig. 2-6(b) vemos un solo proceso con tres hilos de control. Aunque en ambos casos tenemos tres hilos, en la Fig. 2-6(a) cada uno de ellos opera en un espacio de direcciones distinto, en tanto que en la Fig. 2-6(b) los tres comparten el mismo espacio de direcciones.

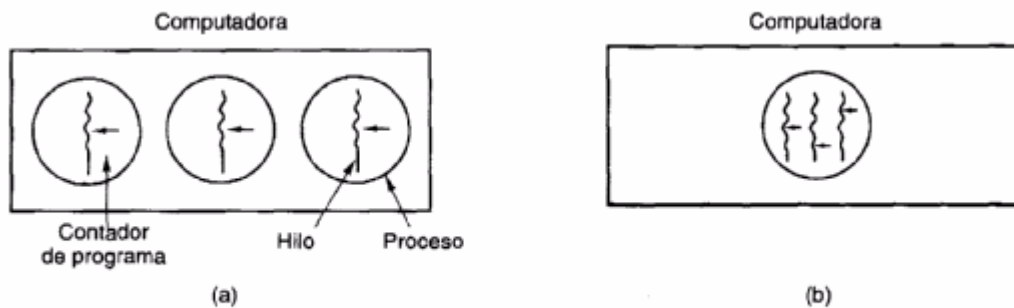


Figura 2-6. (a) Tres procesos, cada uno con un hilo. (b) Un proceso con tres hilos.

Como ejemplo de situación en la que podrían usarse múltiples hilos, consideremos un proceso servidor de archivos que recibe solicitudes para leer y escribir archivos y devuelve los datos solicitados o acepta datos actualizados. A fin de mejorar el rendimiento, el servidor mantiene un caché de archivos recientemente usados en la memoria, leyendo del caché y escribiendo en él siempre que es posible.

Esta situación se presta bien al modelo de la Fig. 2.6(b). Cuando llega una solicitud, se le entrega a un hilo para que la procese. Si ese hilo se bloquea en algún momento esperando una

transferencia de disco, los demás hilos aún pueden ejecutarse, de modo que el servidor puede seguir procesando nuevas solicitudes incluso mientras se está efectuando E/S de disco. En cambio, el modelo de la Fig. 2-6(a) no es apropiado, porque es indispensable que todos los hilos del servidor de archivos tengan acceso al mismo caché, y los tres hilos de la Fig. 2-6(a) no comparten el mismo espacio de direcciones y, por tanto, no pueden compartir el mismo caché en memoria.

Otro ejemplo de caso en el que son útiles los hilos es el de los navegadores de la World Wide Web, como Netscape y Mosaic. Muchas páginas Web contienen múltiples imágenes pequeñas. Para cada imagen de una página Web, el navegador debe establecer una conexión individual con el sitio de la página de casa y solicitar la imagen. Se desperdicia una gran cantidad de tiempo estableciendo y liberando todas estas conexiones. Si tenemos múltiples hilos dentro del navegador, podemos solicitar muchas imágenes al mismo tiempo, acelerando considerablemente el rendimiento en la mayor parte de los casos, ya que en el caso de imágenes pequeñas el tiempo de preparación es el factor limitante, no la rapidez de la línea de transmisión.

Cuando están presentes múltiples hilos en el mismo espacio de direcciones, algunos de los campos de la Fig. 2-4 no contienen información para cada proceso, sino para cada hilo, así que se necesita una tabla de hilos aparte, con una entrada por hilo. Entre los elementos que son distintos para cada hilo están el contador de programa, los registros y el estado. El contador de programa se necesita porque los hilos, al igual que los procesos, pueden suspenderse y reanudarse. Los registros se necesitan porque cuando los hilos se suspenden sus registros deben guardarse. Por último, los hilos, al igual que los procesos, pueden estar en los estados de ejecutándose, listo o bloqueado.

En algunos sistemas el sistema operativo no está consciente de la existencia de los hilos. En otras palabras, los hilos se manejan totalmente en el espacio de usuario. Por ejemplo, cuando un hilo está a punto de bloquearse, escoge e inicia a su sucesor antes de detenerse. Hay varios paquetes de hilos a nivel de usuario, incluidos los paquetes de **hilos P** de POSIX e **hilos C** de Mach.

En otros sistemas, el sistema operativo está consciente de la existencia de múltiples hilos por proceso, así que, cuando un hilo se bloquea, el sistema operativo escoge el que se ejecutará a continuación, ya sea del mismo proceso o de uno distinto. Para realizar la planificación, el kernel debe tener una tabla de hilos que liste todos los hilos del sistema, análoga a la tabla de procesos.

Aunque estas dos alternativas pueden parecer equivalentes, su rendimiento es muy distinto. La conmutación de hilos es mucho más rápida cuando la administración de hilos se efectúa en el espacio de usuario que cuando se necesita una llamada al kernel. Este hecho es un argumento convincente para realizar la administración de hilos en el espacio de usuario. Por otro lado, cuando los hilos se manejan totalmente en el espacio de usuario y uno se bloquea (p. ej., esperando E/S o que se maneje una falla de página), el kernel bloquea todo el proceso, ya que no tiene idea de que existen otros hilos. Este hecho es un argumento importante en favor de realizar la administración de hilos en el kernel. La consecuencia es que se usan ambos sistemas; además, se han propuesto diversos esquemas híbridos (Anderson et al., 1992).

Sea que los hilos sean administrados por el kernel o en el espacio de usuario, introducen una serie de problemas que deben resolverse y que modifican sustancialmente el modelo de programación. Para comenzar, consideremos los efectos de la llamada al sistema FORK. Si el proceso padre tiene múltiples hilos, ¿debe tenerlos también el hijo? Si no, es posible que el proceso no funcione correctamente, ya que es posible que todos ellos sean esenciales.

Por otro lado, si el proceso hijo obtiene tantos hilos como el padre, ¿qué sucede si un hilo estaba bloqueado en una llamada READ de, digamos, el teclado. ¿Hay ahora dos hilos bloqueados esperando el teclado? Si se tecléa una línea, ¿reciben ambos hilos una copia de ella? ¿Sólo el padre? ¿Sólo el hijo? El mismo problema ocurre con las conexiones de red abiertas.

Otra clase de problemas tiene que ver con el hecho de que los hilos comparten muchas estructuras de datos. ¿Qué sucede si un hilo cierra un archivo mientras otro todavía lo está leyendo? Supongamos que un hilo se da cuenta de que hay muy poca memoria y comienza a asignar más memoria. Luego, antes de que termine de hacerlo, ocurre una conmutación de hilo, y el nuevo hilo también se da cuenta de que hay demasiada poca memoria y comienza a asignar más memoria. ¿La asignación ocurre una sola vez o dos? En casi todos los sistemas que no se diseñaron pensando en hilos, las bibliotecas (como el procedimiento de asignación de memoria) no son reentrantes, y se caen si se emite una segunda llamada mientras la primera todavía está activa.

Otro problema se relaciona con los informes de error. En UNIX, después de una llamada al sistema, la situación de la llamada se coloca en una variable global, *errno*. ¿Qué sucede si un hilo efectúa una llamada al sistema y, antes de que pueda leer *errno*, otro hilo realiza una llamada al sistema, borrando el valor original?

Consideremos ahora las señales. Algunas señales son lógicamente específicas para cada hilo, en tanto que otras no lo son. Por ejemplo, si un hilo invoca ALARM, tiene sentido que la señal resultante se envíe al hilo que efectuó la llamada. Si el kernel está consciente de la existencia de hilos, normalmente puede asegurarse de que el hilo correcto reciba la señal. Si el kernel no sabe de los hilos, el paquete de hilos de alguna forma debe seguir la pista a las alarmas. Surge una complicación adicional en el caso de hilos a nivel de usuario cuando (como sucede en UNIX) un proceso sólo puede tener una alarma pendiente a la vez y varios hilos pueden invocar ALARM independientemente.

Otras señales, como una interrupción de teclado, no son específicas para un hilo. ¿Quién deberá atraparlas? ¿Un hilo designado? ¿Todos los hilos? ¿Un hilo recién creado? Todas estas soluciones tienen problemas. Además, ¿qué sucede si un hilo cambia los manejadores de señales sin avisar a los demás hilos?

Un último problema introducido por los hilos es la administración de la pila. En muchos sistemas, cuando hay un desbordamiento de pila, el kernel simplemente amplía la pila automáticamente. Cuando un proceso tiene varios hilos, también debe tener varias pilas. Si el kernel no tiene conocimiento de todas estas pilas, no podrá hacerlas crecer automáticamente cuando haya una falla de pila. De hecho, es posible que el kernel ni siquiera se dé cuenta de que la falla de memoria está relacionada con el crecimiento de una pila.

Estos problemas ciertamente no son insuperables, pero sí ponen de manifiesto que la simple introducción de hilos en un sistema existente sin un rediseño sustancial del sistema no funcionará. Cuando menos, es preciso redefinir la semántica de las llamadas al sistema y reescribir las bibliotecas. Además, todas estas cosas deben hacerse de modo tal que sigan siendo compatibles hacia atrás con los programas existentes para el caso limitante de un proceso con un solo hilo. Si desea información adicional acerca de los hilos, véase (Hauser et al., 1993; y Marsh et al., 1991),

2.2 COMUNICACIÓN ENTRE PROCESOS

Los procesos con frecuencia necesitan comunicarse con otros procesos. Por ejemplo, en un conducto de shell, la salida del primer proceso debe pasarse al segundo proceso, y así sucesivamente. Por tanto, es necesaria la comunicación entre procesos, de preferencia en una forma bien estructurada que no utilice interrupciones. En las siguientes secciones examinaremos algunos de los problemas relacionados con esta **comunicación entre procesos o IPC**.

En pocas palabras, tenemos tres problemas. Ya hicimos alusión al primero de ellos: ¿cómo puede un proceso pasar información a otro? El segundo tiene que ver con asegurarse de que dos o más procesos no se estorben mutuamente al efectuar actividades críticas (suponga que dos procesos tratan de apoderarse al mismo tiempo de los últimos 100K de memoria). El tercero se relaciona con la secuencia correcta cuando existen dependencias: Si el proceso A produce datos y el proceso B los imprime, B tiene que esperar hasta que A haya producido algunos datos antes de comenzar a imprimir. Examinaremos estos tres problemas a partir de la siguiente sección.

2.2.1 Condiciones de competencia

En algunos sistemas operativos, los procesos que están colaborando podrían compartir cierto almacenamiento común en el que ambos pueden leer y escribir. El almacenamiento compartido puede estar en la memoria principal o puede ser un archivo compartido; la ubicación de la memoria compartida no altera la naturaleza de la comunicación ni los problemas que surgen. Para ver cómo funciona la comunicación entre procesos en la práctica, consideremos un ejemplo sencillo pero común, un spooler de impresión. Cuando un proceso desea imprimir un archivo, introduce el nombre del archivo en un **directorio de spooler** especial. Otro proceso, el **demonio de impresión**, revisa periódicamente el directorio para ver si hay archivos por imprimir, y si los hay los imprime y luego borra sus nombres del directorio.

Imagine que nuestro directorio de spooler tiene un número elevado (potencialmente infinito) de ranuras, numeradas 0, 1, 2, ..., cada una con capacidad para un nombre de archivo. Imagine además que hay dos variables compartidas, *out*, que apuntan al siguiente archivo por imprimir, e *in*, que apunta a la siguiente ranura libre del directorio. Estas dos variables bien podrían guardarse en un archivo de dos palabras que estuviera accesible para todos los procesos. En un instante dado, las ranuras 0 a 3 están vacías (los archivos ya se imprimieron) y las ranuras 4 a 6 están llenas (con los nombres de archivos en cola para imprimirse). En forma más o menos simultánea, los procesos A y B deciden que desean poner en cola un archivo para impresión. Esta situación se muestra en la Fig. 2-7.

En las jurisdicciones en las que aplica la ley de Murphy, podría ocurrir lo siguiente. El proceso A lee *in* y almacena su valor, 7, en una variable local llamada *siguiente_ranura_libre*. Justo en ese momento ocurre una interrupción de reloj y la CPU decide que el proceso A ya se ejecutó durante suficiente tiempo, así que conmuta al proceso B. El proceso B también lee *in*, y

+ Si algo puede salir mal, saldrá mal.

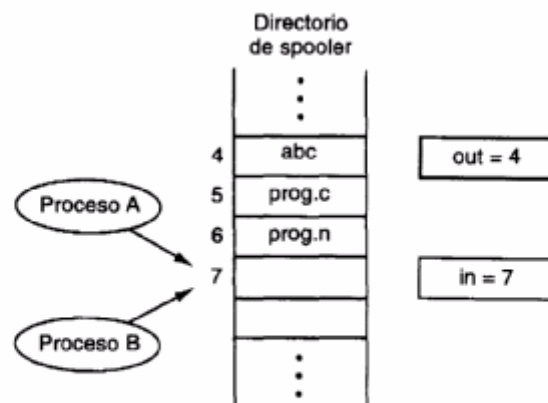


Figura 2-7. Dos procesos quieren acceder a memoria compartida al mismo tiempo.

también obtiene un 7, así que almacena el nombre de su archivo en la ranura 7 y actualiza in con el valor 8. Luego se va y hace otras cosas.

Tarde o temprano, el proceso A se ejecuta otra vez, continuando en donde se interrumpió. A examina `siguiente_ranura_libre`, encuentra 7 ahí, y escribe su nombre de archivo en la ranura 7, borrando el nombre que el proceso B acaba de poner ahí. Luego A calcula `siguiente_ranura_libre + 1`, que es 8, y asigna 8 a in. El directorio de spooler no tiene contradicciones internas, así que el demonio de impresión no notará que algo esté mal, si bien el proceso B nunca obtendrá sus salidas. Situaciones como ésta, en la que dos o más procesos leen o escriben datos compartidos y el resultado final depende de quién se ejecuta precisamente cuándo, se denominan condiciones de competencia. La depuración de programas que contienen condiciones de competencia no es nada divertida. Los resultados de la mayor parte de las pruebas son correctos, pero de vez en cuando algo raro e inexplicable sucede.

2.2.2 Secciones críticas

¿Cómo evitamos las condiciones de competencia? La clave para evitar problemas en ésta y muchas otras situaciones en las que se comparte memoria, archivos o cualquier otra cosa es encontrar una forma de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo. Dicho de otro modo, lo que necesitamos es exclusión mutua: alguna forma de asegurar que si un proceso está usando una variable o archivo compartido, los otros procesos quedarán excluidos de hacer lo mismo. El problema anterior ocurrió porque el proceso B comenzó a usar una de las variables compartidas antes de que A terminara de usarla. La selección de operaciones primitivas apropiadas para lograr la exclusión mutua es un aspecto importante del diseño de cualquier sistema operativo, y un tema que examinaremos con gran detalle en las siguientes secciones.

El problema de evitar condiciones de competencia también puede formularse de manera abstracta. Una parte del tiempo, un proceso está ocupado realizando cálculos internos y otras

cosas que no dan pie a condiciones de competencia. Sin embargo, hay veces en que un proceso está accediendo a memoria o archivos compartidos, o efectuando otras tareas críticas que pueden dar lugar a competencias. Esa parte del programa en la que se accede a la memoria compartida se denomina **región crítica o sección crítica**. Si pudiéramos organizar las cosas de modo que dos procesos nunca pudieran estar en sus regiones críticas al mismo tiempo, podríamos evitar las condiciones de competencia.

Aunque este requisito evita las condiciones de competencia, no es suficiente para lograr que los procesos paralelos cooperen de manera correcta y eficiente usando datos compartidos. Necesitamos que se cumplan cuatro condiciones para tener una buena solución:

1. Dos procesos nunca pueden estar simultáneamente dentro de sus regiones críticas.
2. No puede suponerse nada acerca de las velocidades o el número de las CPU.
3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear a otros procesos.
4. Ningún proceso deberá tener que esperar indefinidamente para entrar en su región crítica.

2.2.3 Exclusión mutua con espera activa

En esta sección examinaremos varias propuestas para lograr la exclusión mutua, de modo que cuando un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso entre en su región crítica y cause problemas.

Inhabilitación de interrupciones

La solución más sencilla es hacer que cada proceso inhabilite las interrupciones justo después de ingresar en su región crítica y vuelva a habilitarlas justo antes de salir de ella. Con las interrupciones inhabilitadas, no pueden ocurrir interrupciones de reloj. Después de todo, la CPU sólo se conmuta de un proceso a otro como resultado de interrupciones de reloj o de otro tipo, y con las interrupciones desactivadas la CPU no se conmutará a ningún otro proceso. Así, una vez que un proceso ha inhabilitado las interrupciones, puede examinar y actualizar la memoria compartida sin temor a que otro proceso intervenga.

Este enfoque casi nunca resulta atractivo porque no es prudente conferir a los procesos de usuario la facultad de desactivar las interrupciones. Supongamos que uno de ellos lo hiciera, y nunca habilitara las interrupciones otra vez. Esto podría terminar con el sistema. Además, si el sistema es multiprocesador, con dos o más CPU, la inhabilitación de las interrupciones afectaría sólo a la CPU que ejecutara la instrucción de inhabilitación; las demás seguirían ejecutándose y podrían acceder a la memoria compartida.

Por otro lado, en muchos casos es necesario que el kernel mismo inhabilite las interrupciones durante unas cuantas instrucciones mientras actualiza variables o listas. Si ocurriera una interrupción en un momento en que la lista de procesos listos, por ejemplo, está en un estado

inconsistente, ocurrirían condiciones de competencia. La conclusión es: la inhabilitación de interrupciones suele ser una técnica útil dentro del sistema operativo mismo pero no es apropiada como mecanismo de exclusión mutua general para los procesos de usuario.

Variables de candado

Veamos ahora una posible solución de software. Supongamos que tenemos una sola variable (de candado) compartida cuyo valor inicial es 0. Cuando un proceso quiere entrar en su región crítica, lo primero que hace es probar el candado. Si el candado es 0, el proceso le asigna 1 y entra en su región crítica; si es 1, el proceso espera hasta que el candado vuelve a ser 0. Así, un 0 significa que ningún proceso está en su región crítica, y un 1 significa que algún proceso está en su región crítica.

Desafortunadamente, esta idea contiene exactamente el mismo defecto fatal que vimos en el directorio de spooler. Supongamos que un proceso lee el candado y ve que es 0. Antes de que este proceso pueda asignar 1 al candado, se planifica otro proceso, el cual se ejecuta y asigna 1 al candado. Cuando el primer proceso continúa su ejecución, asignará 1 al candado, y dos procesos estarán en su región crítica al mismo tiempo.

Podríamos pensar que este problema puede superarse leyendo primero el valor del candado, y verificándolo otra vez justo antes de guardar el 1 en él, pero esto no sirve de nada. La competencia ocurriría entonces si el segundo proceso modifica el candado justo después de que el primer proceso terminó su segunda verificación.

Alternancia estricta

Una tercera estrategia para abordar el problema de la exclusión mutua se muestra en la Fig. 2-8. Este fragmento de programa, como casi todos los del libro, está escrito en C. Se escogió C aquí porque los sistemas operativos reales con frecuencia se escriben en C (u ocasionalmente en C++), pero casi nunca en lenguajes como Modula 2 o Pascal.

<pre>while (TRUE) { while(turn != 0) /* esperar */; critical_region(); turn = 1; noncritical_region(); }</pre>	<pre>while (TRUE) { while(turn != 1) /* esperar */; critical_region(); turn = 0; noncritical_region(); }</pre>
(a)	(b)

Figura 2-8. Solución propuesta para el problema de la región crítica.

En la Fig. 2-8, la variable interna turn, que inicialmente es 0, indica a quién le toca entrar en la región crítica y examinar o actualizar la memoria compartida. En un principio, el proceso 0

inspecciona turn, ve que es O, y entra en su región crítica. El proceso 1 también ve que turn es O y se mantiene en un ciclo corto probando turn continuamente para detectar el momento en que cambia a 1. Esta prueba continua de una variable hasta que adquiere algún valor se denomina **espera activa**, y normalmente debe evitarse, ya que desperdicia tiempo de CPU. La espera activa sólo debe usarse cuando exista una expectativa razonable de que la espera será corta.

Cuando el proceso O sale de la región crítica, asigna 1 a turn, a fin de que el proceso 1 pueda entrar en su región crítica. Supongamos que el proceso 1 termina su región crítica rápidamente, de modo que ambos procesos están en sus regiones no críticas, y turn vale O. Ahora el proceso O ejecuta su ciclo completo rápidamente, regresando a su región no crítica después de haber asignado 1 a turn. Luego, el proceso O termina su región no crítica y regresa al principio de su ciclo. Desafortunadamente, no puede entrar en su región crítica porque turn es 1 y el proceso 1 está ocupado en su región no crítica. Dicho de otro modo, la alternancia de turnos no es una buena idea cuando un proceso es mucho más lento que el otro.

Esta situación viola la condición 3 antes mencionada: el proceso O está siendo bloqueado por un proceso que no está en su región crítica. Volviendo al ejemplo del directorio de spooler, si ahora asociamos la región crítica a las actividades de leer y escribir el directorio de spooler, el proceso O no podría imprimir otro archivo porque el proceso 1 está haciendo alguna otra cosa.

De hecho, esta solución requiere que los dos procesos se alternen estrictamente en el ingreso a sus regiones críticas, por ejemplo, al poner archivos en spool. Ningún proceso podría poner en spool dos archivos seguidos. Si bien este algoritmo evita todas las competencias, no es en realidad un candidato serio para ser una solución porque viola la condición 3.

Solución de Peterson

Combinando la idea de tomar turnos con la de tener variables de candado y variables de advertencia, un matemático holandés, T. Dekker, inventó una solución de software para el problema de la exclusión mutua que no requiere una alternancia estricta. Si desea una explicación del algoritmo de Dekker, véase (Dijkstra, 1965).

En 1981, G. L. Peterson descubrió una forma mucho más sencilla de lograr la exclusión mutua, haciendo obsoleta la solución de Dekker. El algoritmo de Peterson se muestra en la Fig. 2-9, y consiste en dos procedimientos escritos en ANSI C, lo que implica que se deben proporcionar prototipos de función para todas las funciones que se definen y usan. Sin embargo, a fin de ahorrar espacio, no mostraremos los prototipos en este ejemplo ni en los que siguen.

Antes de usar las variables compartidas (es decir, antes de entrar en su región crítica), cada proceso invoca `enter_region` (entrar en región) con su propio número de proceso, 0 o 1, como parámetro. Esta invocación lo obligará a esperar, si es necesario, hasta que pueda entrar sin peligro. Después de haber terminado de manipular las variables compartidas, el proceso invoca `leave_region` (salir de región) para indicar que ya terminó y permitir que el otro proceso entre si lo desea.

Veamos cómo funciona esta solución. En un principio ninguno de los procesos está en su región crítica. Ahora el proceso O invoca `enter_region`, e indica su interés asignando `TRUE` a su elemento del arreglo `interested` (interesado) y asignando `turn` a O. Puesto que el proceso 1 no está interesado, `enter_region` regresa de inmediato. Si ahora el proceso 1 invoca `enter_region`,

```

#define FALSE      0
#define TRUE       1
#define N          2          /* número de procesos */

int turn;                /* ¿a quién le toca? */
int interested[N];      /* todos los valores son inicialmente 0 (FALSE) */

void enter_region(int process); /* proceso 0 o 1 */
{
    int other;           /* número del otro proceso */

    other = 1 - process; /* lo opuesto de process */
    interested[process] = TRUE; /* mostrar interés */
    turn = process;      /* establecer bandera */
    while (turn == process && interested[other] == TRUE) /* instrucción nula */;
}

void leave_region(int process) /* process: quién sale */
{
    interested[process] = FALSE; /* indicar salida de la región crítica */
}

```

Figura 2-9. Solución de Peterson para lograr la exclusión mutua.

permanecerá dando vueltas en su ciclo hasta que se asigne FALSE a *interested[0]*, cosa que sólo sucede cuando el proceso 0 invoca *leave_region* para salir de su región crítica.

Consideremos ahora el caso en que ambos procesos invocan *enter_region* casi simultáneamente. Ambos almacenarán su número de proceso en *turn*, pero el único que cuenta es el que se almacena después; el primero se pierde. Supongamos que el proceso 1 es el segundo en almacenar su número de proceso, así que *turn* vale 1. Cuando ambos procesos llegan a la instrucción *while*, el proceso 0 lo ejecuta cero veces e ingresa en su región crítica. El proceso 1 da vueltas en el ciclo y no entra en su región crítica.

La instrucción TSL

Examinemos ahora una propuesta que requiere un poco de ayuda del hardware. Muchas computadoras, sobre todo las diseñadas pensando en múltiples procesadores, tienen una instrucción TEST AND SET LOCK (TSL, probar y fijar candado) que funciona como sigue. La instrucción lee el contenido de la palabra de memoria, lo coloca en un registro y luego almacena un valor distinto de cero en esa dirección de memoria. Se garantiza que las operaciones de leer la palabra y guardar el valor en ella son indivisibles; ningún otro procesador puede acceder a la palabra de memoria en tanto la instrucción no haya terminado. La CPU que ejecuta la instrucción TSL pone un candado al bus de memoria para que ninguna otra CPU pueda acceder a la memoria en tanto no termine.

Para usar la instrucción TSL, creamos una variable compartida, lock, a fin de coordinar el acceso a la memoria compartida. Cuando lock es 0, cualquier proceso puede asignarle 1 usando la instrucción TSL y luego leer o escribir la memoria compartida. Cuando el proceso termina, asigna otra vez 0 a lock usando una instrucción MOVE ordinaria.

¿Cómo podemos usar esta instrucción para evitar que dos procesos entren simultáneamente en sus regiones críticas? La solución se da en la Fig. 2-10. Ahí se muestra una subrutina de cuatro instrucciones escrita en un lenguaje ensamblador ficticio (pero típico). La primera instrucción copia el valor antiguo de lock en el registro y luego asigna 1 a lock. Luego se compara el valor antiguo con 0. Si es distinto de cero, el candado ya estaba establecido, así que el programa simplemente vuelve al principio y lo prueba otra vez. Tarde o temprano el valor de lock será 0 (cuando el proceso que actualmente está en su región crítica termine con lo que está haciendo dentro de dicha región) y la subrutina regresará, con el candado establecido. Liberar el candado es sencillo, pues basta con almacenar 0 en lock. No se requieren instrucciones especiales.

```

enter_region:
    tsl register,lock      | copiar lock en register y asignarle 1
    cmp register,#0       | ¿era lock 0?
    jne enter_region     | si no era cero, se asignó 1 a lock, y se ejecuta el ciclo
    ret                  | volver al invocador; se entró en la región crítica

leave_region:
    move lock,#0         | guardar un 0 en lock
    ret                  | volver al invocador

```

Figura 2-10. Establecimiento y liberación de candados con TSL.

Ya tenemos una solución al problema de la región crítica que es directa. Antes de entrar en su región crítica, un proceso invoca `enter_region`, la cual realiza espera activa hasta que el candado está libre; luego adquiere el candado y regresa. Después de la región crítica el proceso invoca `leave_region`, que almacena un 0 en lock. Al igual que todas las soluciones basadas en regiones críticas, el proceso debe invocar `enter_region` y `leave_region` en los momentos correctos para que el método funcione. Si un proceso hace trampa, la exclusión mutua fallará.

2.2.4 Dormir y despertar

Tanto la solución de Peterson como la que usa TSL son correctas, pero ambas tienen el defecto de requerir espera activa. En esencia, lo que estas soluciones hacen es lo siguiente: cuando un proceso desea entrar en su región crítica verifica si está permitida la entrada; si no, el proceso simplemente repite un ciclo corto esperando hasta que lo esté.

Este enfoque no sólo desperdicia tiempo de CPU, sino que también puede tener efectos inesperados. Consideremos una computadora con dos procesos, H, de alta prioridad, y L, de baja

prioridad. Las reglas de planificación son tales que H se ejecuta siempre que está en el estado listo. En un momento dado, con L en su región crítica, H queda listo para ejecutarse (p. ej., se completa una operación de E/S). H inicia ahora la espera activa, pero dado que L nunca se planifica mientras H se está ejecutando, L nunca tiene oportunidad de salir de su región crítica, y H permanece en un ciclo infinito. Esta situación se conoce como **problema de inversión de prioridad**.

Examinemos ahora algunas primitivas de comunicación entre procesos que se bloquean en lugar de desperdiciar tiempo de CPU cuando no se les permite entrar en sus regiones críticas. Una de las más sencillas es el par SLEEP y WAKEUP. SLEEP (dormir) es una llamada al sistema que hace que el invocador se bloquee, es decir, se suspenda hasta que otro proceso lo despierte. La llamada WAKEUP (despertar) tiene un parámetro, el proceso que se debe despertar. Como alternativa, tanto SLEEP como WAKEUP pueden tener un parámetro cada uno, una dirección de memoria que sirve para enlazar los SLEEP con los WAKEUP.

El problema de productor-consumidor

Como ejemplo de uso de estas primitivas, consideremos el problema de **productor-consumidor** (también conocido como problema del **buffer limitado**). Dos procesos comparten un mismo buffer de tamaño fijo. Uno de ellos, el productor, coloca información en el buffer, y el otro, el consumidor, la saca. (También es posible generalizar el problema a m productores y n consumidores, pero sólo consideraremos el caso de un productor y un consumidor porque esto simplifica las soluciones.)

Surgen problemas cuando el productor quiere colocar un nuevo elemento en el buffer, pero éste ya está lleno. La solución es que el productor se duerma y sea despertado cuando el consumidor haya retirado uno o más elementos. De forma similar, si el consumidor desea sacar un elemento del buffer y ve que está vacío, se duerme hasta que el productor pone algo en el buffer y lo despierta.

Este enfoque parece muy sencillo, pero da lugar a los mismos tipos de condiciones de competencia que vimos antes con el directorio de spooler. Para seguir la pista al número de elementos contenidos en el buffer, necesitaremos una variable, count. Si el número máximo de elementos que el buffer puede contener es N, el código del productor primero verificará si count es igual a N. Si es así, el productor se dormirá; si no, el productor agregará un elemento e incrementará count.

El código del consumidor es similar: primero se prueba count para ver si es 0. Si así es, el consumidor se duerme; si no, el consumidor saca un elemento y decrementa el contador. Cada uno de estos procesos verifica también si el otro debería estar durmiendo, y si no es así, lo despierta. El código del productor y del consumidor se muestra en la Fig. 2-11,

Para expresar las llamadas al sistema como SLEEP y WAKEUP en C, las mostraremos como llamadas a rutinas de biblioteca. Éstas no forman parte de la biblioteca estándar de C, pero es de suponer que estarían disponibles en cualquier sistema que realmente tuviera esas llamadas al sistema. Los procedimientos `enter_item` (colocar elemento) y `remove_item` (retirar elemento), que no se muestran, se encargan de la contabilización de la colocación de elementos en el buffer y el retiro de elementos de él.

```

#define N 100                                /* número de ranuras del buffer */
int count = 0;                               /* número de elementos en el buffer */

void producer(void)
{
    while (TRUE) {                            /* repetir indefinidamente */
        produce_item();                       /* generar el siguiente elemento */
        if (count == N) sleep();             /* si el buffer está lleno, dormir */
        enter_item();                         /* colocar elemento en el buffer */
        count = count + 1;                   /* incrementar la cuenta de elementos
                                                en el buffer */

        if (count == 1) wakeup(consumer);    /* ¿estaba vacío el buffer? */
    }
}

void consumer(void)
{
    while (TRUE){                             /* repetir indefinidamente */
        if (count == 0) sleep();             /* si el buffer está vacío, dormir */
        remove_item();                       /* remover elemento del buffer */
        count = count - 1;                   /* decrementar la cuenta de elementos
                                                en el buffer */

        if (count == N-1) wakeup(producer); /* ¿estaba lleno el buffer? */
        consume_item();                      /* imprimir elemento */
    }
}

```

Figura 2-11. El problema de productor-consumidor con una condición de competencia fatal.

Volvamos ahora a la condición de competencia. Ésta puede ocurrir porque el acceso a count es inestricto, y podría presentarse la siguiente situación. El buffer está vacío y el consumidor acaba de leer count para ver si es 0. En ese instante, el planificador decide dejar de ejecutar el consumidor temporalmente y comenzar a ejecutar el productor. Éste coloca un elemento en el buffer, incrementa count, y observa que ahora vale 1. Esto implica que antes count valía 0, y por ende que el consumidor está durmiendo, así que el productor invoca wakeup para despertar al consumidor.

Desafortunadamente, el consumidor todavía no está dormido lógicamente, de modo que la señal de despertar se pierde. Cuando el consumidor reanuda su ejecución, prueba el valor de count que había leído previamente, ve que es 0 y se duerme. Tarde o temprano el productor llenará el buffer y se dormirá. Ambos seguirán durmiendo eternamente.

La esencia del problema aquí es que se perdió una llamada enviada para despertar a un proceso que (todavía) no estaba dormido. Si no se perdiera, todo funcionaría. Una compostura rápida consiste en modificar las reglas y agregar un bit de espera de despertar a la escena. Cuando se envía una llamada de despertar a un proceso que está despierto, se enciende este bit. Después, cuando el proceso trata de dormirse, si el bit de espera de despertar está encendido, se

apagará, pero el proceso seguirá despierto. El bit de espera de despertar actúa como una alcancía de señales de despertar. Aunque el bit de espera de despertar nos salva el pellejo en este ejemplo, es fácil construir ejemplos con tres o más procesos en los que un bit de espera de despertar es insuficiente. Podríamos crear otro parche y agregar un segundo bit de espera de despertar, o quizá 8 o 32, pero en principio el problema sigue ahí.

2.2.5 Semáforos

Ésta era la situación en 1965, cuando E. W. Dijkstra (1965) sugirió usar una variable entera para contar el número de señales de despertar guardadas para uso futuro. En esta propuesta se introdujo un nuevo tipo de variable, llamada **semáforo**. Un semáforo podía tener el valor 0, indicando que no había señales de despertar guardadas, o algún valor positivo si había una o más señales de despertar pendientes.

Dijkstra propuso tener dos operaciones, DOWN y UP (generalizaciones de SLEEP y WAKEUP, respectivamente). La operación DOWN (abajo) aplicada a un semáforo verifica si el valor es mayor que 0; de ser así, decrementa el valor (esto es, gasta una señal de despertar almacenada) y continúa. Si el valor es 0, el proceso se pone a dormir sin completar la operación DOWN por el momento. La verificación del valor, su modificación y la acción de dormirse, si es necesaria, se realizan como una sola **acción atómica** indivisible. Se garantiza que una vez que una operación de semáforo se ha iniciado, ningún otro proceso podrá acceder al semáforo hasta que la operación se haya completado o bloqueado. Esta atomicidad es absolutamente indispensable para resolver los problemas de sincronización y evitar condiciones de competencia.

La operación UP incrementa el valor del semáforo direccionado. Si uno o más procesos están durmiendo en espera de ese semáforo, imposibilitados de completar una operación DOWN previa, el sistema escoge uno de ellos (p. ej., al azar) y le permite completar su DOWN. Así, después de un up con un semáforo que tiene procesos durmiendo esperando, el semáforo seguirá siendo 0, pero habrá un proceso menos que se halle en fase de durmiendo esperando. La operación de incrementar el semáforo y despertar un proceso también es indivisible. Ningún proceso se bloquea durante un up, así como ningún proceso se bloquea realizando un WAKEUP en el modelo anterior.

Como acotación, en su artículo original Dijkstra usó las letras P y en lugar de DOWN y UP, respectivamente, pero en vista de que éstos no tienen significado mnemónico para quienes no hablan holandés (y apenas un significado marginal para quienes lo hablan), usaremos los términos DOWN y up en vez de éstos. DOWN y UP se introdujeron por primera vez en Algol 68.

Resolución del problema de productor-consumidor usando semáforos

Los semáforos resuelven el problema de la señal de despertar perdida, como se muestra en la Fig. 2-12. Es indispensable que se implementen de modo que sean indivisibles. El método normal consiste en implementar UP y DOWN como llamadas al sistema, para que el sistema operativo inhabilite brevemente todas las interrupciones mientras prueba el semáforo, lo actualiza y pone el proceso a dormir, si es necesario. Todas estas acciones requieren sólo unas cuantas instrucciones,

así que la inhabilitación de las interrupciones no tiene consecuencias adversas. Si se están usando múltiples CPU, cada semáforo debe estar protegido con una variable de candado, usando la instrucción TSL para asegurarse de que sólo una CPU a la vez examine el semáforo. Cerciórese de entender que el empleo de TSL para evitar que varias CPU accedan al semáforo al mismo tiempo es muy diferente de la espera activa del productor o el consumidor cuando esperan que el otro vacíe o llene el buffer. La operación del semáforo sólo toma unos cuantos microsegundos, mientras que el productor o el consumidor podrían tardar un tiempo arbitrariamente largo.

```

#define N 100                                /* número de ranuras del buffer */
typedef int semaphore;                       /* los semáforos son un tipo especial de int */
semaphore mutex = 1;                         /* controla el acceso a la región crítica */
semaphore empty = N;                         /* cuenta las ranuras de buffer vacías */
semaphore full = 0;                          /* cuenta las ranuras de buffer llenas */

void producer(void)
{
    int item;

    while (TRUE) {                            /* TRUE es la constante 1 */
        produce_item(&item);                 /* generar algo para ponerlo en el buffer */
        down(&empty);                        /* decrementar el contador empty */
        down(&mutex);                        /* entrar en la región crítica */
        enter_item(item);                    /* colocar el nuevo elemento en el buffer */
        up(&mutex);                          /* salir de la región crítica */
        up(&full);                           /* incrementar el contador de ranuras llenas */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                            /* ciclo infinito */
        down(&full);                         /* decrementar el contador full */
        down(&mutex);                        /* entrar en la región crítica */
        remove_item(&item);                 /* sacar elemento del buffer */
        up(&mutex);                          /* salir de la región crítica */
        up(&empty);                          /* incrementar el contador de ranuras vacías */
        consume_item(item);                 /* hacer algo con el elemento */
    }
}

```

Figura 2-12. El problema de productor-consumidor usando semáforos.

Esta solución usa tres semáforos: uno llamado *full* para contar el número de ranuras que están llenas, uno llamado *empty* para contar el número de ranuras que están vacías, y otro llamado *mutex* para asegurarse de que el productor y el consumidor no accedan al buffer al mismo tiempo. *Full* inicialmente vale 0, *empty* inicialmente es igual al número de ranuras del buffer y *mutex* inicialmente es 1. Los semáforos a los que se asigna 1 como valor inicial y son utilizados por dos o más procesos para asegurar que sólo uno de ellos pueda entrar en su región crítica al mismo tiempo se denominan **semáforos binarios**. Si cada proceso ejecuta *DOWN* justo antes de entrar en su región crítica, y *up* justo después de salir de ella, la exclusión mutua está garantizada.

Ahora que contamos con una buena primitiva de comunicación entre procesos, regresemos y examinemos otra vez la secuencia de interrupción de la Fig. 2-5. En un sistema que usa semáforos, la forma natural de ocultar las interrupciones es tener un semáforo, inicialmente puesto en 0, asociado a cada dispositivo de E/S. Inmediatamente después de iniciar un dispositivo de E/S, el proceso que lo administra ejecuta *DOWN* con el semáforo correspondiente, bloqueándose así de inmediato. Cuando llega la interrupción, el manejador de instrucciones ejecuta *up* con el semáforo correspondiente, haciendo que el proceso en cuestión quede otra vez listo para ejecutarse. En este modelo, el paso 6 de la Fig. 2-5 consiste en ejecutar *UP* con el semáforo del dispositivo, de modo que en el paso 7 el planificador pueda ejecutar el administrador del dispositivo. Desde luego, si ahora varios procesos están listos, el planificador puede optar por ejecutar a continuación un proceso aún más importante. Más adelante en este capítulo veremos cómo se realiza la planificación.

En el ejemplo de la Fig. 2-12, realmente usamos los semáforos de dos formas distintas. Esta diferencia es lo bastante importante como para hacerla explícita. El semáforo *mutex* se usa para exclusión mutua; está diseñado para garantizar que sólo un proceso a la vez estará leyendo o escribiendo el buffer y las variables asociadas a él. Esta exclusión mutua es necesaria para evitar el caos.

El otro uso de los semáforos es la **sincronización**. Los semáforos *full* y *empty* se necesitan para garantizar que ciertas secuencias de sucesos ocurran o no ocurran. En este caso, los semáforos aseguran que el productor dejará de ejecutarse cuando el buffer esté lleno y que el consumidor dejará de ejecutarse cuando el buffer esté vacío. Este uso es diferente de la exclusión mutua.

Aunque los semáforos se han usado desde hace más de un cuarto de siglo, todavía se siguen efectuando investigaciones sobre su uso. Por ejemplo, véase (Tai y Carver, 1996).

2.2.6 Monitores

Con los semáforos, la comunicación entre procesos parece fácil, ¿no es así? Ni por casualidad. Examine de cerca el orden de los *DOWN* antes de colocar elementos en el buffer o retirarlos de él en la Fig. 2-12. Suponga que el orden de los dos *DOWN* del código del productor se invirtiera, de modo que *mutex* se incrementara antes que *empty* en lugar de después de él. Si el buffer estuviera completamente lleno, el productor se bloquearía, con *mutex* puesto en 0. En consecuencia, la próxima vez que el consumidor tratara de acceder al buffer ejecutaría *DOWN* con *mutex*, que ahora es 0, y también se bloquearía. Ambos procesos permanecerían bloqueados indefinidamente y ya no se efectuaría más trabajo. Esta lamentable situación se llama bloqueo mutuo, y la estudiaremos con detalle en el capítulo 3.

Señalamos este problema para destacar el cuidado que debemos tener al usar semáforos. Basta un error sutil para que todo se paralice. Es como programar en lenguaje ensamblador, sólo que peor, porque los errores son condiciones de competencia, bloqueo y otras formas de comportamiento impredecible e irreproducible.

A fin de facilitar la escritura de programas correctos, Hoare (1974) y Brinch Hansen (1975) propusieron una primitiva de sincronización de nivel más alto llamada **monitor**. Sus propuestas tenían pequeñas diferencias, que describiremos más adelante. Un monitor es una colección de procedimientos, variables y estructuras de datos que se agrupan en un tipo especial de módulo o paquete. Los procesos pueden invocar los procedimientos de un monitor en el momento en que deseen, pero no pueden acceder directamente a las estructuras de datos internas del monitor desde procedimientos declarados afuera del monitor. La Fig. 2-13 ilustra un monitor escrito en un lenguaje imaginario parecido a Pascal:

```
monitor example
integer i;
condition c;

procedure producer(x);
:
:
end;

procedure consumer(x);
:
:
end;
end monitor;
```

Figura 2-13. Un monitor.

Los monitores poseen una propiedad especial que los hace útiles para lograr la exclusión mutua: sólo un proceso puede estar activo en un monitor en un momento dado. Los monitores son una construcción de lenguaje de programación, así que el compilador sabe que son especiales y puede manejar las llamadas a procedimientos de monitor de una forma diferente a como maneja otras llamadas a procedimientos. Por lo regular, cuando un proceso invoca un procedimiento de monitor, las primeras instrucciones del procedimiento verifican si hay algún otro proceso activo en ese momento dentro del monitor. Si así es, el proceso invocador se suspende hasta que el otro proceso abandona el monitor. Si ningún otro proceso está usando el monitor, el proceso invocador puede entrar.

Es responsabilidad del compilador implementar la exclusión mutua en las entradas a monitores, pero una forma común es usar un semáforo binario. Puesto que el compilador, no el programador, se está encargando de la exclusión mutua, es mucho menos probable que algo salga mal. En

cualquier caso, la persona que escribe el monitor no tiene que saber cómo el compilador logra la exclusión mutua; le basta con saber que si convierte todas las regiones críticas en procedimientos de monitor, dos procesos nunca podrán ejecutar sus regiones críticas al mismo tiempo.

Aunque los monitores ofrecen una forma fácil de lograr la exclusión mutua, esto no es suficiente, como acabamos de ver. También necesitamos un mecanismo para que los procesos se bloqueen cuando no puedan continuar. En el problema de productor-consumidor, es fácil colocar todas las pruebas para determinar si el buffer está lleno o está vacío en procedimientos de monitor, pero ¿cómo deberá bloquearse el productor cuando encuentra lleno el buffer?

La solución está en la introducción de **variables de condición**, junto con dos operaciones que se realizan con ellas, WA y SIGNAL. Cuando un procedimiento de monitor descubre que no puede continuar (p. ej., si encuentra lleno el buffer), ejecuta WAIT (esperar) con alguna variable de condición, digamos full! (lleno). Esta acción hace que el proceso invocador se bloquee, y también permite la entrada de otro proceso al que antes se le había impedido entrar en el monitor.

Este otro proceso (p. ej., el consumidor) puede despertar a su “compañero” dormido ejecutando SIGNAL (señal) con la variable de condición que su compañero está esperando. A fin de evitar la presencia de dos procesos activos en el monitor al mismo tiempo, necesitamos una regla que nos diga qué sucede después de ejecutarse SIGNAL. Hoare propuso dejar que el proceso recién despertado se ejecute, suspendiendo el otro. Brinch Hansen propuso sortear el problema exigiendo al proceso que ejecutó SIGNAL salir inmediatamente del monitor. Dicho de otro modo, una instrucción SIGNAL sólo puede aparecer como última instrucción de un procedimiento de monitor. Usaremos la propuesta de Brinch Hansen porque es conceptualmente más sencilla y también más fácil de implementar. Si se ejecuta SIGNAL con una variable de condición que varios procesos están esperando, sólo uno de ellos, el que el planificador del sistema determine, será reactivado.

Las variables de condición no son contadores; no acumulan señales para uso futuro como hacen los semáforos. Por tanto, si se ejecuta SIGNAL con una variable de condición que ningún proceso está esperando, la señal se pierde. La operación WAIT debe venir antes que SIGNAL. Esta regla simplifica mucho la implementación. En la práctica, esto no es un problema porque es fácil seguir la pista al estado de cada proceso con variables, si es necesario. Un proceso que de otra manera ejecutaría SIGNAL puede ver que esta operación no es necesaria si examina las variables.

En la Fig. 2-14 se presenta un esqueleto del problema productor-consumidor con monitores, escrito en pseudo-Pascal.

El lector tal vez esté pensando que las operaciones WAIT y SIGNAL son similares a SLEEP y WAKEUP que, como vimos antes, tenían condiciones de competencia fatales. Son muy similares, pero tienen una diferencia crucial: SLEEP y WAKEUP fallaron porque mientras un proceso intentaba dormirse, el otro estaba tratando de despertarlo. Con monitores, esto no puede suceder. La exclusión mutua automática en los procedimientos de monitor garantiza que si, por ejemplo, el productor dentro de un procedimiento de monitor descubre que el buffer T está lleno, podrá completar la operación WAIT sin tener que preocuparse por la posibilidad de que el planificador pueda conmutar al consumidor justo antes de que se complete el WAIT. El consumidor ni siquiera podrá entrar en el monitor en tanto el WAIT no se haya completado y el productor haya dejado de ser ejecutable.


```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure enter;
  begin
    if count = N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  procedure remove;
  begin
    if count = 0 then wait(empty);
    remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;

  count := 0;
end monitor;

procedure producer;
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter
  end
end;

procedure consumer;
begin
  while true do
  begin
    ProducerConsumer.remove;
    consume_item
  end
end;
end;
```

Figura 2-14. Bosquejo del problema de productor-consumidor con monitores. Sólo un procedimiento de monitor está activo a la vez. El *buffer* tiene *N* ranuras.

Al hacer automática la exclusión mutua de las regiones críticas, los monitores hacen a la programación en paralelo mucho menos propensa a errores que cuando se usan semáforos. No obstante, tienen algunas desventajas. No es por capricho que la Fig. 2-14 está escrita en un lenguaje ficticio y no en C, como otros ejemplos de este libro. Como dijimos antes, los monitores son un concepto de lenguajes de programación. El compilador debe reconocerlos y lograr de alguna manera la exclusión mutua. C, Pascal y casi todos los demás lenguajes carecen de monitores, por lo que no es razonable esperar que sus compiladores hagan cumplir reglas de exclusión mutua. De hecho, ¿cómo podría el compilador saber siquiera cuáles procedimientos están en monitores y cuáles no?

Estos mismos lenguajes tampoco tienen semáforos, pero la adición de semáforos es fácil: todo lo que se necesita es agregar dos rutinas cortas escritas en lenguaje ensamblador a la biblioteca para poder emitir las llamadas al sistema UP y DOWN. Los compiladores ni siquiera tienen que saber que existen. Desde luego, los sistemas operativos tienen que estar enterados de los semáforos, pero al menos si se cuenta con un sistema operativo basado en semáforos es posible escribir los programas de usuario para él en C o C++ (o incluso BASIC si su masoquismo llega a tanto). En el caso de los monitores, se necesita un lenguaje que los tenga incorporados. Unos cuantos lenguajes, como Concurrent Euclid (Holt, 1983) los tienen, pero son poco comunes.

Otro problema con los monitores, y también con los semáforos, es que se diseñaron con la intención de resolver el problema de la exclusión mutua en una o más CPU, todas las cuales tienen acceso a una memoria común. Al colocar los semáforos en la memoria compartida y protegerlos con instrucciones TSL, podemos evitar las competencias. Cuando pasamos a un sistema distribuido que consiste en múltiples CPU, cada una con su propia memoria privada, conectadas por una red de área local, estas primitivas ya no son aplicables. La conclusión es que los semáforos son de nivel demasiado bajo y que los monitores sólo pueden usarse con unos cuantos lenguajes de programación. Además, ninguna de las primitivas contempla el intercambio de información entre máquinas. Se necesita otra cosa.

2.2.7 Transferencia de mensajes

Esa otra cosa es la **transferencia de mensajes**. Este método de comunicación entre procesos utiliza dos primitivas SEND y RECEIVE que, al igual que los semáforos y a diferencia de los monitores, son llamadas al sistema y no construcciones del lenguaje. Como tales, es fácil colocarlas en procedimientos de biblioteca, como

```
send(destino, &mensaje);
```

y

```
receive(origen, &mensaje);
```

La primera llamada envía un mensaje a un destino dado, y la segunda recibe un mensaje de un origen dado (o de cualquiera [ANY] si al receptor no le importa). Si no hay un mensaje disponible,

el receptor podría bloquearse hasta que uno llegue. Como alternativa, podría regresar de inmediato con un código de error.

Aspectos de diseño de los sistemas de transferencia de mensajes

Los sistemas de transferencia de mensajes tienen muchos problemas y aspectos de diseño complicados que no se presentan con los semáforos ni con los monitores, sobre todo si los procesos en comunicación están en diferentes máquinas conectadas por una red. Por ejemplo, se pueden perder mensajes en la red. Para protegerse contra la pérdida de mensajes, el emisor y el receptor pueden convenir que, tan pronto como se reciba un mensaje, el receptor enviará de regreso un mensaje especial de **acuse de recibo** o **confirmación**. Si el emisor no recibe el acuse dentro de cierto intervalo de tiempo, retransmitirá el mensaje.

Consideremos ahora lo que sucede si el mensaje en sí se recibe correctamente, pero se pierde el acuse de recibo. El emisor retransmitirá el mensaje, de modo que el receptor lo recibirá dos veces. Es indispensable que el receptor pueda distinguir un mensaje nuevo de la retransmisión de uno viejo. Por lo regular, este problema se resuelve incluyendo números de secuencia consecutivos en cada mensaje original. Si el receptor recibe un mensaje que tiene el mismo número de secuencia que uno anterior, sabrá que el mensaje es un duplicado y podrá ignorarlo.

Los sistemas de mensajes también tienen que resolver la cuestión del nombre de los procesos, a fin de que el proceso especificado en una llamada SEND o RECEIVE no sea ambiguo. La **verificación de autenticidad** es otro problema en los sistemas de mensajes: ¿cómo puede el cliente saber que se está comunicando con el verdadero servidor de archivos, y no con un impostor?

En el otro extremo del espectro, hay aspectos de diseño que son importantes cuando el emisor y el receptor están en la misma máquina. Uno de éstos es el rendimiento. El copiado de mensajes de un proceso a otro siempre es más lento que efectuar una operación de semáforo o entrar en un monitor. Se ha trabajado mucho tratando de hacer eficiente la transferencia de mensajes. Cheriton (1984), por ejemplo, ha sugerido limitar el tamaño de los mensajes a lo que cabe en los registros de la máquina, y efectuar luego la transferencia de mensajes usando los registros.

El problema de productor-consumidor con transferencia de mensajes

Veamos ahora cómo puede resolverse el problema de productor-consumidor usando transferencia de mensajes y sin compartir memoria. En la Fig. 2-15 se presenta una solución. Suponemos que todos los mensajes tienen el mismo tamaño y que el sistema operativo coloca automáticamente en buffers los mensajes enviados pero aún no recibidos. En esta solución se usa un total de N mensajes, análogos a las N ranuras de un buffer en memoria compartida. El consumidor inicia enviando N mensajes vacíos al productor. Cada vez que el productor tiene un elemento que entregar al consumidor, toma un mensaje vacío y devuelve uno lleno. De este modo, el número total de mensajes en el sistema permanece constante y pueden almacenarse en una cantidad de memoria que se conoce con antelación.

Si el productor trabaja con mayor rapidez que el consumidor, todos los mensajes quedarán llenos, esperando al consumidor; el productor se bloqueará, esperando la llegada de un mensaje vacío. Si el consumidor trabaja con mayor rapidez, ocurre lo opuesto: todos los mensajes estarán vacíos esperando que el productor los llene; el consumidor estará bloqueado, esperando un mensaje lleno.

```

#define N 100                                /* número de ranuras del buffer */

void producer(void)
{
    int item;
    message m;                               /* buffer de mensaje */

    while (TRUE) {
        produce_item(&item);                 /* generar algo que poner en el buffer */
        receive(consumer, &m);              /* esperar que llegue un mensaje vacío */
        build_message(&m, item);            /* construir un mensaje para enviar */
        send(consumer, &m);                 /* enviar elemento al consumidor */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for(i = 0; i < N; i++) send(producer, &m); /* enviar N mensajes vacíos */
    while (TRUE) {
        receive(producer, &m);               /* obtener mensaje que contiene elemento */
        extract_item(&m, &item);             /* extraer elemento del mensaje */
        send(producer, &m);                  /* devolver una respuesta vacía */
        consume_item(item);                  /* hacer algo con el elemento */
    }
}

```

La transferencia de mensajes puede tener muchas variantes. Para comenzar, veamos cómo se dirigen los mensajes. Una forma es asignar a cada proceso una dirección única y hacer que los mensajes se dirijan a los procesos. Un método distinto consiste en inventar una nueva estructura de datos, llamada **buzón**. Un buzón es un lugar donde se almacena temporalmente cierta cantidad de mensajes, que normalmente se especifican cuando se crea el buzón. Si se usan buzones, los parámetros de dirección de las llamadas SEND y RECEIVE son buzones, no procesos. Cuando un

proceso trata de transmitir a un buzón que está lleno, queda suspendido hasta que se retira un mensaje de ese buzón, dejando espacio para uno nuevo.

En el caso del problema de productor-consumidor, tanto el productor como el consumidor crearían buzones con espacio suficiente para N mensajes. El productor enviaría mensajes con datos al buzón del consumidor, y éste enviaría mensajes vacíos al buzón del productor. Si se usan buzones, el mecanismo de almacenamiento temporal es claro: el buzón de destino contiene mensajes que se han enviado al proceso de destino pero todavía no han sido aceptados.

La otra forma extrema de manejar buzones es eliminar todo el almacenamiento temporal. Cuando se adopta este enfoque, si el SEND se ejecuta antes que el RECEIVE, el proceso emisor queda bloqueado hasta que ocurre el RECEIVE, y en ese momento el mensaje podrá copiarse directamente del emisor al receptor, sin buffers intermedios. De forma similar, si el RECEIVE se ejecuta primero, el receptor se bloquea hasta que ocurre el SEND. Esta estrategia se conoce como **cita** o **rendezvous**; es más fácil de implementar que un esquema de mensajes con almacenamiento temporal, pero es menos flexible, pues se obliga al emisor y al receptor a operar estrictamente sincronizados.

La comunicación entre los procesos de usuario en MINIX (y en UNIX) se efectúa a través de conductos, que efectivamente son buzones. La única diferencia real entre un sistema de mensajes con buzones y el mecanismo de conductos es que los conductos no preservan los límites de los mensajes. Dicho de otro modo, si un proceso escribe 10 mensajes de 100 bytes cada uno en un conducto y otro proceso lee 1000 bytes de ese conducto, el lector obtendrá los 10 mensajes a la vez. Con un verdadero sistema de mensajes, cada READ debería devolver sólo un mensaje. Desde luego, si los procesos convienen en leer y escribir siempre mensajes de tamaño fijo del conducto, o en terminar cada mensaje con un carácter especial (p. ej., salto de línea), no habrá problema. Los procesos que constituyen el sistema operativo MINIX mismo utilizan un verdadero esquema de mensajes de tamaño fijo para comunicarse entre sí.

2.3 PROBLEMAS CLÁSICOS DE IPC

La literatura sobre sistemas operativos abunda en problemas interesantes que han sido estudiados y analizados ampliamente. En las siguientes secciones examinaremos tres de los más conocidos.

23.1 El problema de la cena de filósofos

En 1965, Dijkstra planteó y resolvió un problema de sincronización al que llamó **problema de la cena de filósofos**. Desde entonces, quienquiera que haya inventado una primitiva de sincronización más se ha sentido obligado a demostrar lo maravillosa que es mostrando la forma tan elegante en que resuelve el problema de la cena de filósofos. El problema tiene un planteamiento muy sencillo. Cinco filósofos están sentados alrededor de una mesa circular. Cada filósofo tiene ante sí un plato de espagueti. El espagueti es tan resbaloso que un filósofo necesita dos tenedores para comerlo. Entre cada par de platos hay un tenedor. La disposición de la mesa se ilustra en la Fig. 2-16.

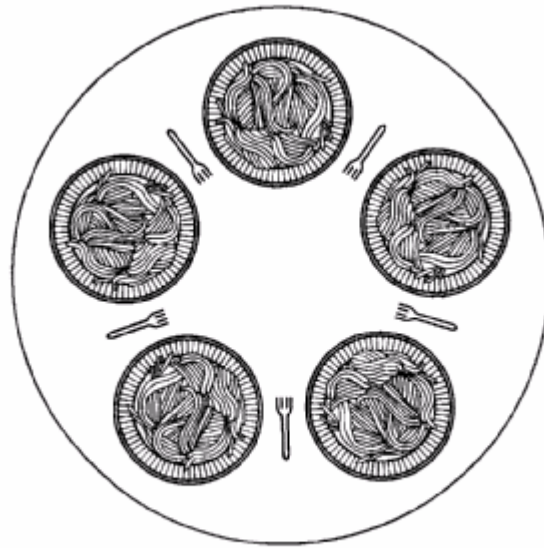


Figura 2-16. Hora de comer en el departamento de filosofía.

La vida de un filósofo consiste en periodos alternantes de comer y pensar. (Esto es una abstracción, incluso en el caso de un filósofo, pero las demás actividades no son pertinentes aquí.) Cuando un filósofo siente hambre, trata de adquirir sus tenedores izquierdo y derecho, uno a la vez, en cualquier orden. Si logra adquirir dos tenedores, comerá durante un rato, luego pondrá los tenedores en la mesa y seguirá pensando. La pregunta clave es: ¿podemos escribir un programa para cada filósofo que haga lo que se supone que debe hacer y nunca se entrampe? (Se ha señalado que el requisito de los dos tenedores es un tanto artificial; tal vez deberíamos cambiar de la comida italiana a la china, sustituyendo el espagueti por arroz y los tenedores por palillos chinos.)

La Fig. 2-17 muestra la solución obvia. El procedimiento `take_fork` (tomar tenedor) espera hasta que el tenedor especificado está disponible y luego se apodera de él. Desafortunadamente, la solución obvia está equivocada. Supongamos que todos los filósofos toman su tenedor izquierdo simultáneamente. Ninguno podrá tomar su tenedor derecho, y tendremos un bloqueo mutuo.

Podríamos modificar el programa de modo que, después de tomar el tenedor izquierdo, el programa verifique si el tenedor derecho está disponible. Si no es así, el filósofo soltará su tenedor izquierdo, esperará cierto tiempo, y repetirá el proceso. Esta propuesta también fracasa, aunque por una razón distinta. Con un poco de mala suerte, todos los filósofos podrían iniciar el algoritmo simultáneamente, tomar su tenedor izquierdo, ver que su tenedor derecho no está disponible, dejar su tenedor izquierdo, esperar, tomar su tenedor izquierdo otra vez de manera simultánea, y así eternamente. Una situación así, en la que todos los programas continúan ejecutándose de manera indefinida pero no logran avanzar se denomina **inanición** (adopta este calificativo aun cuando el problema no ocurra en un restaurante italiano o chino).

Ahora podríamos pensar: “si los filósofos esperan un tiempo aleatorio en lugar del mismo tiempo después de fracasar en su intento por disponer del tenedor derecho, la posibilidad de que

```

#define N 5                /* número de filósofos */

void philosopher(int i)    /* i: número de filósofo, de 0 a 4 */
{
    while (TRUE) {
        think();           /* el filósofo está pensando */
        take_fork(i);      /* tomar tenedor izquierdo */
        take_fork((i+1) % N); /* tomar tenedor derecho; % es el operador de residuo */
        eat();             /* delicioso espaguetti */
        put_fork(i);       /* poner el tenedor izquierdo otra vez en la mesa */
        put_fork((i+1) % N); /* poner el tenedor derecho otra vez en la mesa */
    }
}

```

Figura 2-17. Una no-solución al problema de la cena de filósofos.

sus acciones continuaran coordinadas durante siquiera una hora es excesivamente pequeña”. Esto es cierto, pero en algunas aplicaciones preferiríamos una solución que siempre funcione y que no tenga posibilidad de fallar debido a una serie improbable de números aleatorios. (Pensemos en el control de seguridad en una planta de energía nuclear.)

Una mejora de la Fig. 2-17 que no está sujeta a bloqueo ni inanición consiste en proteger las cinco instrucciones que siguen a la llamada a think (pensar) con un semáforo binario. Antes de comenzar a conseguir tenedores, un filósofo ejecutaría DOWN con mutex. Después de dejar los tenedores en la mesa, ejecutaría up con mutex. Desde un punto de vista teórico, esta solución es adecuada. En la práctica, empero, tiene un problema de rendimiento: sólo un filósofo puede estar comiendo en un instante dado. Si hay cinco tenedores disponibles, deberíamos estar en condiciones de permitir que dos filósofos comieran al mismo tiempo.

La solución que se presenta en la Fig. 2-18 es correcta y también admite un paralelismo máximo con un número arbitrario de filósofos. Se utiliza un arreglo state (estado) para mantenerse al tanto de si un filósofo está comiendo, pensando o hambriento (tratando de disponer de tenedores). Un filósofo sólo puede pasar a la situación de “comiendo” si ninguno de sus vecinos está comiendo. Los vecinos del filósofo i están definidos por las macros LEFT y RIGHT. En otras palabras, si i es 2, LEFT es 1 y RIGHT es 3.

El programa utiliza un arreglo de semáforos, uno por filósofo, de modo que los filósofos hambrientos pueden bloquearse si los tenedores que necesitan están ocupados. Observe que cada proceso ejecuta el procedimiento philosopher (filósofo) como código principal, pero los demás procedimientos, take_forks (tomar tenedores), put_forks (poner tenedores) y test (probar) son procedimientos ordinarios y no procesos aparte.

23.2 El problema de lectores y escritores

El problema de la cena de filósofos es útil para modelar procesos que compiten por tener acceso exclusivo a un número limitado de recursos, como dispositivos de E/S. Otro problema famoso es

```

#define N          5          /* número de filósofos */
#define LEFT (i-1)%N      /* número del vecino izquierdo de i */
#define RIGHT (i+1)%N    /* número del vecino derecho de i */
#define THINKING  0        /* el filósofo está pensando */
#define HUNGRY    1        /* el filósofo intenta obtener tenedores */
#define EATING    2        /* el filósofo está comiendo */

typedef int semaphore;      /* los semáforos son un tipo especial de int */
int state[N];              /* arreglo para seguir la pista al estado de todos */
semaphore mutex = 1;       /* exclusión mutua para regiones críticas */
semaphore s[N];           /* un semáforo por filósofo */
void philosopher(int i)   /* i: número de filósofo, de 0 a N-1 */
{
    while (TRUE) {        /* repetir indefinidamente */
        think();          /* el filósofo está pensando */
        take_forks(i);    /* adquirir dos tenedores o bloquearse */
        eat();            /* delicioso espagueti */
        put_forks(i);     /* dejar ambos tenedores en la mesa */
    }
}

void take_forks(int i)    /* i: número de filósofo, de 0 a N-1 */
{
    down(&mutex);         /* entrar en la región crítica */
    state[i] = HUNGRY;    /* registrar el hecho de que el filósofo i tiene hambre */
    test(i);              /* tratar de adquirir 2 tenedores */
    up(&mutex);           /* salir de la región crítica */
    down(&s[i]);          /* bloquearse si no se adquirieron los tenedores */
}

void put_forks(i)        /* i: número de filósofo, de 0 a N-1 */
{
    down(&mutex);         /* entrar en la región crítica */
    state[i] = THINKING; /* el filósofo terminó de comer */
    test(LEFT);           /* ver si el vecino izquierdo ahora puede comer */
    test(RIGHT);          /* ver si el vecino derecho ahora puede comer */
    up(&mutex);          /* salir de la región crítica */
}

void test(i)             /* i: número de filósofo, de 0 a N-1 */
{
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

Figura 2-18. Una solución al problema de la cena de filósofos.

el de los lectores y escritores (Courtois et al., 1971), que modela el acceso a una base de datos. Imaginemos, por ejemplo, un sistema de reservaciones de una línea aérea, con muchos procesos competidores que desean leerlo y escribir en él. Es aceptable tener múltiples procesos leyendo la base de datos al mismo tiempo, pero si un proceso está actualizando (escribiendo en) la base de datos, ningún otro podrá tener acceso a ella, ni siquiera los lectores. La pregunta es, ¿cómo programamos a los lectores y escritores? Una solución se muestra en la Fig. 2-19.

```

typedef int semaphore;           /* use su imaginación */
semaphore mutex = 1;           /* controla el acceso a 'rc' */
semaphore db = 1;              /* controla el acceso a la base de datos */
int rc = 0;                     /* núm. de procesos que leen o quieren leer */

void reader(void)
{
    while (TRUE) {              /* repetir indefinidamente */
        down(&mutex);           /* obtener acceso exclusivo a 'rc' */
        rc = rc + 1;            /* ahora un lector más */
        if (rc == 1) down(&db); /* si éste es el primer lector ... */
        up(&mutex);             /* liberar el acceso exclusivo a 'rc' */
        read_data_base();       /* acceder a los datos */
        down(&mutex);           /* obtener acceso exclusivo a 'rc' */
        rc = rc - 1;            /* ahora un lector menos */
        if (rc == 0) up(&db);   /* si éste es el último lector ... */
        up(&mutex);             /* liberar el acceso exclusivo a 'rc' */
        use_data_read();        /* región no crítica */
    }
}

void writer(void)
{
    while (TRUE) {              /* repetir indefinidamente */
        think_up_data();         /* región no crítica */
        down(&db);              /* obtener acceso exclusivo */
        write_data_base();       /* actualizar los datos */
        up(&db);                /* liberar el acceso exclusivo */
    }
}

```

Figura 2-19. Una solución al problema de lectores y escritores.

En esta solución, el primer lector que obtiene acceso a la base de datos ejecuta DOWN con el semáforo db. Los lectores subsiguientes se limitan a incrementar un contador, rc. Conforme los lectores salen, decrementan el contador, y el último en salir ejecuta UP con el semáforo para permitir que un escritor bloqueado, silo había, entre.

La solución que presentamos aquí contiene implícitamente una sutil decisión que vale la pena comentar. Supongamos que mientras un lector está usando la base de datos, llega otro lector. Puesto que tener dos lectores al mismo tiempo no está prohibido, se admite al segundo lector. También pueden admitirse un tercer lector y lectores subsecuentes si llegan.

Supongamos ahora que llega un escritor. El escritor no puede ser admitido en la base de datos, pues requiere acceso exclusivo, de modo que el escritor queda suspendido. Más adelante, llegan lectores adicionales. En tanto haya al menos un lector activo, se admitirán lectores subsecuentes. A consecuencia de esta estrategia, en tanto haya un suministro constante de lectores, entrarán tan pronto como lleguen. El escritor se mantendrá suspendido hasta que no haya ningún lector presente. Si llega un lector, digamos, cada 2 segundos, y cada lector tarda 5 segundos en efectuar su trabajo, el escritor nunca entrará.

Para evitar esta situación, el programa podría incluir una pequeña modificación: cuando llega un lector y un escritor está esperando, el lector queda suspendido detrás del escritor en lugar de ser admitido inmediatamente. Así, un escritor tiene que esperar hasta que terminan los lectores que estaban activos cuando llegó, pero no a que terminen los lectores que llegaron después de él. La desventaja de esta solución es que logra menor concurrencia y por tanto un menor rendimiento. Courtois et al., presentan una solución que confiere prioridad a los escritores. Si desea conocer los detalles, remítase a su artículo.

2.3.3 El problema del peluquero dormido

Otro problema de IPC clásico ocurre en una peluquería. Esta peluquería tiene un peluquero, una silla de peluquero y n sillas donde pueden sentarse los clientes que esperan, si los hay. Si no hay clientes presentes, el peluquero se sienta en la silla de peluquero y se duerme, como se ilustra en la Fig. 2-20. Cuando llega un cliente, tiene que despertar al peluquero dormido. Si llegan clientes adicionales mientras el peluquero está cortándole el pelo a un cliente, se sientan (si hay sillas vacías) o bien salen del establecimiento (si todas las sillas están ocupadas). El problema consiste en programar al peluquero y sus clientes sin entrar en condiciones de competencia.

Nuestra solución utiliza tres semáforos: `customers`, que cuenta a los clientes en espera (excluyendo al que está siendo atendido, que no está esperando), `barbers`, el número de peluqueros que están ociosos, esperando clientes (0 o 1), y `mutex`, que se usa para la exclusión mutua. También necesitamos una variable, `waiting` (esperando), que también cuenta los clientes que están esperando, y que en esencia es una copia de `customers`. Necesitamos esta variable porque no es posible leer el valor actual de un semáforo. En esta solución, un cliente que entra en la peluquería debe contar el número de clientes que esperan. Si este número es menor que el número de sillas, se queda; si no, se va.

Nuestra solución se muestra en la Fig. 2-21. Cuando el peluquero llega a trabajar en la mañana, ejecuta el procedimiento `barber` (peluquero) que lo obliga a bloquearse en espera de `customers` hasta que llegue alguien. Luego se duerme como se muestra en la Fig. 2-20.

Cuando un cliente llega, ejecuta `customer` (cliente), cuya primera instrucción es adquirir `mutex` para entrar en una región crítica. Si otro cliente llega poco tiempo después, no podrá hacer nada hasta que el primero haya liberado `mutex`. A continuación, el cliente verifica si el número de



Figura 2-20. El peluquero dormido.

clientes en espera es menor que el número de sillas. Si no es así, el cliente libera mutex y se sale sin su corte de pelo.

Si hay una silla disponible, el cliente incrementa la variable entera waiting y luego ejecuta UP con el semáforo customers, lo que despierta al peluquero. En este punto, tanto el peluquero como el cliente están despiertos. Cuando el cliente libera mutex, el peluquero lo toma, realiza algo de aseo e inicia el corte de pelo.

Una vez terminado el corte de pelo, el cliente sale del procedimiento y de la peluquería. A diferencia de los ejemplos anteriores, no hay un ciclo para el cliente porque cada uno sólo recibe un corte de pelo. El peluquero sí opera en un ciclo, tratando de atender al siguiente cliente. Si hay uno presente, el peluquero realiza otro corte de pelo; si no, se duerme.

Como acotación, vale la pena señalar que si bien los problemas de lectores y escritores y del peluquero dormido no implican transferencia de datos, pertenecen al área de IPC porque implican sincronización entre varios procesos.

```

#define CHAIRS 5                /* núm. sillas para clientes que esperan */

typedef int semaphore;         /* use su imaginación */

semaphore customers = 0;      /* núm. de clientes que esperan ser atendidos */
semaphore barbers = 0;       /* núm. de peluqueros que esperan clientes */
semaphore mutex = 1;         /* para exclusión mutua */
int waiting = 0;             /* clientes que esperan (no están siendo atendidos) */

void barber(void)
{
    while (TRUE) {
        down(customers);      /* dormirse si el núm. de clientes es 0 */
        down(mutex);         /* adquirir acceso a 'waiting' */
        waiting = waiting - 1; /* decrementar cuenta de clientes en espera */
        up(barbers);         /* un peluquero está listo para cortar el pelo */
        up(mutex);           /* liberar 'waiting' */
        cut_hair();          /* cortar el pelo (fuera de la región crítica) */
    }
}

void customer(void)
{
    down(mutex);              /* entrar en la región crítica */
    if(waiting < CHAIRS) {    /* si no hay sillas desocupadas, irse */
        waiting = waiting + 1; /* incrementar cuenta de clientes en espera */
        up(customers);        /* despertar al peluquero si es necesario */
        up(mutex);           /* liberar el acceso a 'waiting' */
        down(barbers);        /* dormirse si el núm. de peluqueros libres es 0 */
        get_haircut();        /* sentarse y ser atendido */
    } else {
        up(mutex);           /* peluquería llena; no esperar */
    }
}

```

Figura 2-21. Una solución al problema del peluquero dormido.

2.4 PLANIFICACIÓN DE PROCESOS

En los ejemplos de las secciones anteriores tuvimos varias situaciones en las que dos o más procesos (p. ej., productor y consumidor) podían ejecutarse lógicamente. Cuando hay más de un proceso ejecutable, el sistema operativo debe decidir cuál ejecutará primero. La parte del sistema operativo que toma esta decisión se denomina **planificador**; el algoritmo que usa se denomina **algoritmo de planificación**.

En la época de los sistemas por lote con entradas en forma de imágenes de tarjetas en una cinta magnética, el algoritmo de planificación era sencillo: simplemente se ejecutaba el siguiente trabajo de la cinta. En los sistemas de tiempo compartido, el algoritmo de planificación es más complejo, pues es común que haya varios usuarios en espera de ser atendidos, y también puede haber uno o más flujos por lotes (p. ej., en una compañía de seguros, para procesar reclamaciones). Incluso en las computadoras personales, puede haber varios procesos iniciados por el usuario compitiendo por la CPU, sin mencionar los trabajos de segundo plano, como los demonios de red o de correo electrónico que envían o reciben mensajes.

Antes de examinar algoritmos de planificación específicos, debemos pensar en qué está tratando de lograr el planificador. Después de todo, éste se ocupa de decidir una política, no de proveer un mecanismo. Se nos ocurren varios criterios para determinar en qué consiste un buen algoritmo de planificación. Entre las posibilidades están:

1. Equitatividad —asegurarse de que cada proceso reciba una parte justa del tiempo de CPU.
2. Eficiencia —mantener la CPU ocupada todo el tiempo.
3. Tiempo de respuesta —minimizar el tiempo de respuesta para usuarios interactivos.
4. Retorno —minimizar el tiempo que los usuarios por lotes tienen que esperar sus salidas.
5. Volumen de producción —maximizar el número de trabajos procesados por hora.

Si pensamos un poco veremos que algunos de estos objetivos son contradictorios. Si queremos minimizar el tiempo de respuesta para los usuarios interactivos, el planificador no deberá ejecutar trabajos por lotes (excepto quizá entre las 3 A.M. y las 6 A.M., cuando todos los usuarios interactivos están muy a gusto en sus camas). A los usuarios por lotes seguramente no les gustaría este algoritmo, pues viola el criterio 4. Puede demostrarse (Kleinrock, 1975) que cualquier algoritmo de planificación que dé preferencia a una clase de trabajos perjudicará a los de otras clases. Después de todo, la cantidad de tiempo de CPU disponible es finita. Para darle más a un usuario tenemos que darle menos a otro. Así es la vida.

Una complicación que deben enfrentar los planificadores es que cada proceso es único e impredecible. Algunos dedican una buena parte del tiempo a esperar E/S de archivos, mientras otros usarían la CPU durante horas si se les permitiera hacerlo. Cuando el planificador comienza a ejecutar un proceso, nunca sabe con certeza cuánto tiempo pasará antes de que dicho proceso se bloquee, sea para E/S, en espera de un semáforo o por alguna otra razón. Para asegurarse de que ningún proceso se ejecute durante demasiado tiempo, casi todas las computadoras tienen incorporado un cronómetro o reloj electrónico que genera interrupciones periódicamente. Es común que la frecuencia sea de 50060 interrupciones por segundo (equivalente a 50 o 60 **hertz**, abreviado **Hz**), pero en muchas computadoras el sistema operativo puede ajustar la frecuencia del cronómetro al valor que desee. En cada interrupción de reloj, el sistema operativo se ejecuta y decide si debe permitirse que el proceso que se está ejecutando actualmente continúe o si ya disfrutó de suficiente tiempo de CPU por el momento y debe suspenderse para otorgar a otro proceso la CPU.

La estrategia de permitir que procesos lógicamente ejecutables se suspendan temporalmente se denomina **planificación expropiativa** y contrasta con el método de **ejecución hasta terminar** de los primeros sistemas por lotes. La ejecución hasta terminar también se denomina **planificación no expropiativa**. Como hemos visto a lo largo del capítulo, un proceso puede ser suspendido en un instante arbitrario, sin advertencia, para que otro proceso pueda ejecutarse. Esto da pie a condiciones de competencia y requiere semáforos, monitores, mensajes o algún otro método avanzado para prevenirlas. Por otro lado, una política de dejar que los procesos se ejecuten durante el tiempo que quieran implicaría que un proceso que está calculando π con una precisión de mil millones de cifras podría privar de servicio a todos los demás procesos indefinidamente.

Así, aunque los algoritmos de planificación no apropiativos son sencillos y fáciles de implementar, por lo regular no son apropiados para sistemas de aplicación general con varios usuarios que compiten entre sí. Por otro lado, en un sistema dedicado como un servidor de base de datos, bien puede ser razonable que el proceso padre inicie un proceso hijo para trabajar con una solicitud y dejarlo que se ejecute hasta terminar o bloquearse. La diferencia respecto al sistema de aplicación general es que todos los procesos del sistema de bases de datos están bajo el control de un solo amo, que sabe lo que cada hijo va a hacer y cuánto va a tardar.

2.4.1 Planificación round robin (de torneo)

Examinemos ahora algunos algoritmos de planificación específicos. Uno de los más antiguos, sencillos, equitativos y ampliamente utilizados es el de **round robin**. A cada proceso se le asigna un intervalo de tiempo, llamado **cuanto**, durante el cual se le permite ejecutarse. Si el proceso todavía se está ejecutando al expirar su cuanto, el sistema operativo se apropia de la CPU y se la da a otro proceso. Si el proceso se bloquea o termina antes de expirar el cuanto, la conmutación de CPU naturalmente se efectúa cuando el proceso se bloquee. El round robin es fácil de implementar. Todo lo que el planificador tiene que hacer es mantener una lista de procesos ejecutables, como se muestra en la Fig. 2-22(a). Cuando un proceso gasta su cuanto, se le coloca al final de la lista, como se aprecia en la Fig. 2-22(b).

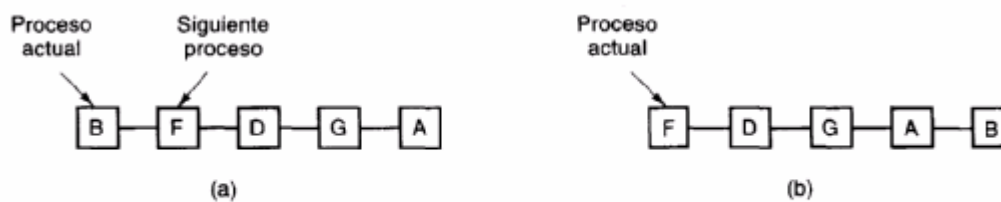


Figura 2-22. Planificación *round robin*. (a) La lista de procesos ejecutables. (b) La lista de procesos ejecutables después de que *B* gasta su cuanto.

La única cuestión interesante cuando se usa el round robin es la duración del cuanto. La conmutación de un proceso a otro requiere cierto tiempo para llevar a cabo las tareas administrativas: guardar y cargar registros y mapas de memoria, actualizar diversas tablas y listas, etc. Su pongamos que esta **conmutación de proceso** o **conmutación de contexto** requiere 5 ms. Supongamos también que usamos cuantos de 20 ms. Con estos parámetros, después de realizar trabajo

útil durante 20 ms, la CPU tendrá que ocupar 5 ms en la conmutación de procesos. Se desperdiciará el 20% del tiempo de CPU en gastos extra administrativos.

A fin de mejorar la eficiencia de la CPU, podríamos usar cuantos de, digamos, 500 ms. Ahora el tiempo desperdiciado es de menos del 1%, pero consideremos lo que sucede en un sistema de tiempo compartido si 10 usuarios interactivos pulsán la tecla de retomo de carro aproximadamente al mismo tiempo: diez procesos se pondrían en la lista de procesos ejecutables. Si la CPU está ociosa, el primero se iniciará de inmediato, el segundo podría no iniciarse hasta cerca de medio segundo después, y así sucesivamente. El pobre proceso que le haya tocado ser último podría tener que esperar 5 segundos antes de tener su oportunidad, suponiendo que los demás procesos utilizan su cuanto completo. Para casi cualquier usuario, un retardo de 5 segundos en la respuesta a un comando corto sería terrible. El mismo problema puede presentarse en una computadora personal que maneja multiprogramación.

La conclusión puede formularse así: escoger un cuanto demasiado corto causa demasiadas Conmutaciones de procesos y reduce la eficiencia de la CPU, pero escogerlo demasiado largo puede dar pie a una respuesta deficiente a solicitudes interactivas cortas. Un cuanto de cerca de 100 ms suele ser un término medio razonable.

2.4.2 Planificación por prioridad

La planificación en round robin supone implícitamente que todos los procesos son igualmente importantes. Con frecuencia, las personas que poseen y operan sistemas de computadora multiusuario tienen ideas diferentes acerca del tema. En una universidad, la jerarquía puede consistir en decanos primero, luego profesores, secretarías, conserjes y, por último, estudiantes. La necesidad de tener en cuenta factores externos da pie a la **planificación por prioridad**. La idea básica es sencilla: a cada proceso se le asigna una prioridad, y se permite que se ejecute el proceso ejecutable que tenga la prioridad más alta.

Incluso en una PC con un solo dueño, puede haber múltiples procesos, algunos más importantes que otros. Por ejemplo, un proceso demonio que envía correo electrónico en segundo plano debe tener menor prioridad que un proceso que está exhibiendo video en tiempo real en la pantalla.

A fin de evitar que los procesos de alta prioridad se ejecuten indefinidamente, el planificador puede reducir la prioridad de los procesos que actualmente se ejecutan en cada tic del reloj (esto es, en cada interrupción de reloj). Si esta acción hace que la prioridad se vuelva menor que la del siguiente proceso con más alta prioridad, ocurrirá una conmutación de procesos. Como alternativa, se podría asignar a cada proceso un cuanto máximo en el que se le permitiera tener la CPU continuamente; cuando se agota este cuanto, se da oportunidad al proceso con la siguiente prioridad más alta de ejecutarse.

Podemos asignar prioridades a los procesos estática o dinámicamente. En una computadora militar, los procesos iniciados por generales podrían comenzar con prioridad 100, los iniciados por coroneles con 90, por mayores con 80, por capitanes con 70, por tenientes con 60, etc. Como alternativa, en un centro de cómputo comercial, los procesos de alta prioridad podrían costar 100 dólares por hora, los de mediana prioridad 75 dólares por hora, y los de baja prioridad 50 dólares

por hora. El sistema UNIX tiene un comando, `nice`, que permite a un usuario reducir voluntaria mente la prioridad de su proceso, con objeto de ser amable con los demás usuarios. Nadie lo usa.

El sistema también puede asignar prioridades dinámicamente a fin de lograr ciertos objetivos del sistema. Por ejemplo, algunos procesos están limitados principalmente por E/S y pasan la mayor parte del tiempo esperando que terminen operaciones de BIS. Siempre que un proceso necesita la CPU, se le deberá otorgar de inmediato, con objeto de que pueda iniciar su siguiente solicitud de E/S, que entonces podrá proceder en paralelo con otro proceso que sí está realizando cálculos. Si hiciéramos que los procesos limitados por BIS esperaran mucho tiempo la CPU, implicaría tenerlo por ahí ocupando memoria durante un tiempo innecesariamente largo. Un algoritmo sencillo para dar buen servicio a los procesos limitados por E/S es asignarles la prioridad $1/f$, donde f es la fracción del último cuanto que un proceso utilizó. Un proceso que usó sólo 2 ms de su cuanto de 100 ms recibiría una prioridad de 50, en tanto que un proceso que se ejecutó 50 ms antes de bloquearse obtendría una prioridad de 2, y uno que ocupó todo su cuanto obtendría una prioridad de 1.

En muchos casos es conveniente agrupar los procesos en clases de prioridad y usar planificación por prioridad entre las clases pero planificación round robin dentro de cada clase. La Fig. 2-23 muestra un sistema con cuatro clases de prioridad. El algoritmo de planificación es el siguiente: en tanto haya procesos ejecutables en la clase de prioridad 4, se ejecutará cada uno durante un cuanto, con round robin, sin ocuparse de las clases de menor prioridad. Si la clase de prioridad 4 está vacía, se ejecutan los procesos de la clase 3 con round robin. Si tanto la clase 4 como la 3 están vacías, se ejecutan los procesos de clase 2 con round robin, etc. Si las prioridades no se ajustan ocasionalmente, las clases de baja prioridad podrían morir de inanición.

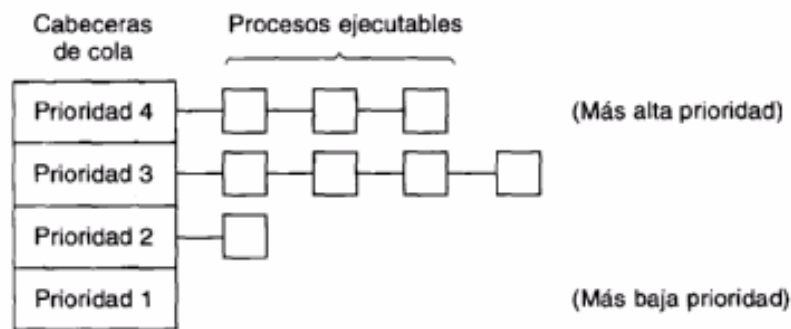


Figura 2-23. Algoritmo de planificación con cuatro clases de prioridad.

2.4.3 Colas múltiples

Uno de los primeros planificadores por prioridad se incluyó en CTSS (Corbato et al., 1962). CTSS tenía el problema de que la conmutación de procesos era muy lenta porque la 7094 sólo podía contener un proceso en la memoria. Cada conmutación implicaba escribir el proceso actual en disco y leer uno nuevo del disco. Los diseñadores de CTSS pronto se dieron cuenta de que resultaba más eficiente dar a los procesos limitados por CPU un cuanto largo de vez en cuando,

en lugar de darles cuantos pequeños muy a menudo (porque se reducía el intercambio). Por otro lado, dar a todos los procesos un cuanto largo implicaría un tiempo de respuesta deficiente, como ya hemos visto. Su solución consistió en establecer clases de prioridad. Los procesos de la clase más alta se ejecutaban durante un cuanto. Los procesos de la siguiente clase más alta se ejecutaban durante dos cuantos. Los procesos de la siguiente clase se ejecutaban durante cuatro cuantos, y así sucesivamente. Cada vez que un proceso agotaba todos los cuantos que tenía asignados, se le degradaba una clase.

Por ejemplo, consideremos un proceso que necesita calcular continuamente durante 100 cuantos; inicialmente, se le daría un cuanto, y luego se intercambiaría por otro proceso. La siguiente vez, recibiría dos cuantos antes de ser intercambiado. En ocasiones subsecuentes obtendría 4, 8, 16, 32 y 64 cuantos, aunque sólo usaría 37 de los últimos 64 cuantos para completar su trabajo. Sólo se necesitarían 7 intercambios (incluida la carga inicial) en lugar de 100 si se usara un algoritmo round robin puro. Además, al hundirse el proceso progresivamente en las colas de prioridad, se le ejecutaría cada vez con menor frecuencia, guardando la CPU para procesos interactivos cortos.

Se adoptó la siguiente política para evitar que un proceso que en el momento de iniciarse necesita ejecutarse durante un tiempo largo pero posteriormente se vuelve interactivo fuera castigado indefinidamente. Cada vez que en una terminal se pulsaba el retorno de carro, el proceso perteneciente a esa terminal se pasaba a la clase de más alta prioridad, bajo el supuesto de que estaba a punto de volverse interactivo. Un buen día un usuario con un proceso muy limitado por CPU descubrió que si se sentaba ante su terminal y pulsaba el retorno de carro al azar cada varios segundos su tiempo de respuesta mejoraba notablemente. Este usuario se lo dijo a todos sus amigos. Moraleja de la historia: acertar en la práctica es mucho más difícil que acertar en la teoría.

Se han utilizado muchos otros algoritmos para asignar procesos a clases de prioridad. Por ejemplo, el influyente sistema XDS 940 (Lampson, 1968), construido en Berkeley, tenía cuatro clases de prioridad, llamadas terminal, E/S, cuanto corto y cuanto largo. Cuando un proceso que estaba esperando entradas de la terminal finalmente se despertaba, pasaba a la clase de prioridad más alta (terminal). Cuando un proceso que estaba esperando un bloque de disco quedaba listo, pasaba a la segunda clase. Si un proceso seguía en ejecución en el momento de expirar su cuanto, se le colocaba inicialmente en la tercera clase, pero si agotaba su cuanto demasiadas veces seguidas sin bloquearse para E/S de terminal o de otro tipo, se le bajaba a la cuarta cola. Muchos otros sistemas usan algo similar para dar preferencia a los usuarios y procesos interactivos por encima de los de segundo plano.

2.4.4 El primer trabajo más corto

La mayor parte de los algoritmos anteriores se diseñaron para sistemas interactivos. Examinemos ahora uno que resulta especialmente apropiado para los trabajos por lotes cuyos tiempos de ejecución se conocen por adelantado. En una compañía de seguros, por ejemplo, es posible predecir con gran exactitud cuánto tiempo tomará ejecutar un lote de 1000 reclamaciones, pues se efectúan trabajos similares todos los días. Si hay varios trabajos de igual importancia esperando en la cola

de entrada para ser iniciados, el planificador deberá usar el criterio del **primer trabajo más corto**. Examinemos la Fig. 2-24. Aquí encontramos cuatro trabajos, A, B, C y D, con tiempos de ejecución de 8, 4, 4 y 4 minutos, respectivamente. Si los ejecutamos en ese orden, el tiempo de retomo para A será de 8 minutos, para B, de 12 minutos, para C, de 16 minutos, y para D, de 20 minutos, siendo el promedio de 14 minutos.



Figura 2-24. Ejemplo de planificación del primer trabajo más corto.

Consideremos ahora la ejecución de estos trabajos usando el primer trabajo más corto, como se muestra en la Fig. 2-24(b). Los tiempos de retomo son ahora de 4, 8, 12 y 20 minutos para un promedio de 11 minutos. Se puede demostrar que la política del primer trabajo más corto es óptima. Consideremos el caso de cuatro trabajos, con tiempos de ejecución de a , b , c y d , respectivamente. El primer trabajo termina en un tiempo a , el segundo, en $a + b$, etc. El tiempo de retomo medio es $(4a + 3b + 2c + d)/4$. Es evidente que a contribuye más al promedio que los demás tiempos, por lo que debe ser el trabajo más corto, siguiendo b , c y por último d , que es el más largo y sólo afecta su propio tiempo de retomo. El mismo argumento es aplicable a cualquier cantidad de trabajos.

Dado que la política del primer trabajo más corto produce el tiempo de respuesta medio mínimo, sería deseable poderlo usar también para procesos interactivos. Esto es posible hasta cierto punto. Los procesos interactivos generalmente siguen el patrón de esperar un comando, ejecutar el comando, esperar un comando, ejecutar el comando, etc. Si consideramos la ejecución de cada comando como un “trabajo” individual, podremos minimizar el tiempo de respuesta global ejecutando primero el trabajo más corto. El único problema es determinar cuál de los procesos ejecutables es el más corto.

Una estrategia consiste en hacer estimaciones basadas en el comportamiento histórico y ejecutar el proceso con el tiempo de ejecución estimado más corto. Supongamos que el tiempo por comando estimado para cierta terminal es T_0 . Supongamos ahora que se mide su siguiente ejecución, dando T_1 . Podríamos actualizar nuestro estimado calculando una suma ponderada de estos dos números, es decir, $aT_0 + (1 - a)T_1$. Dependiendo del valor que escojamos para a , podremos hacer que el proceso de estimación olvide las ejecuciones viejas rápidamente, o las recuerde durante mucho tiempo. Con $a = 1/2$, obtenemos estimaciones sucesivas de

$$T_0, T_0/2 + T_1/2, T_0/4 + T_1/4 + T_2/2, T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Después de tres nuevas ejecuciones, el peso de T_0 en el nuevo estimado se ha reducido a $1/8$.

La técnica de estimar el siguiente valor de una serie calculando la media ponderada del valor medido actual y el estimado previo también se conoce como **maduración**, y es aplicable a muchas situaciones en las que debe hacerse una predicción basada en valores previos. La maduración es

especialmente fácil de implementar cuando $a = 1/2$. Todo lo que se necesita es sumar el nuevo valor al estimado actual y dividir la suma entre 2 (desplazándola a la derecha un bit).

Vale la pena señalar que el algoritmo del primer trabajo más corto sólo es óptimo cuando todos los trabajos están disponibles simultáneamente. Como contraejemplo, consideremos cinco trabajos, A a E, con tiempos de ejecución de 2, 4, 1, 1 y 1, respectivamente. Sus tiempos de llegada son 0, 0, 3, 3 y 3.

Inicialmente, sólo pueden escogerse A o B, puesto que los otros tres trabajos todavía no llegan. Si ejecutamos el primer trabajo más corto, seguiremos el orden de ejecución A, B, C, D, E logrando una espera media de 4.6. Sin embargo, si los ejecutamos en el orden B, C, D, E y A la espera media será de 4.4.

2.4.5 Planificación garantizada

Una estrategia de planificación totalmente distinta consiste en hacer promesas reales al usuario en cuanto al rendimiento y después cumplirlas. Una promesa que es realista y fácil de cumplir es la siguiente: si hay n usuarios en sesión mientras usted está trabajando, usted recibirá aproximadamente $1/n$ de la capacidad de la CPU. De forma similar, en un sistema monousuario con n procesos en ejecución, si todo lo demás es igual, cada uno deberá recibir un de los ciclos de CPU.

Para poder cumplir esta promesa, el sistema debe llevar la cuenta de cuánto tiempo de CPU ha tenido cada proceso desde su creación. A continuación, el sistema calculará el tiempo de CPU al que tenía derecho cada proceso, es decir, el tiempo desde la creación dividido entre n . Puesto que también se conoce el tiempo de CPU del que cada proceso ha disfrutado realmente, es fácil calcular la relación entre el tiempo de CPU recibido y el tiempo al que se tenía derecho. Una relación de 0.5 implica que el proceso sólo ha disfrutado de la mitad del tiempo al que tenía derecho, y una relación de 2.0 implica que un proceso ha tenido dos veces más tiempo del que debería haber tenido. El algoritmo consiste entonces en ejecutar el trabajo con la relación más baja hasta que su relación haya rebasado la de su competidor más cercano.

2.4.6 Planificación por lotería

Si bien hacer promesas a los usuarios y después cumplirlas es una idea admirable, es difícil de implementar. Podemos usar otro algoritmo para obtener resultados igualmente predecibles con una implementación mucho más sencilla. El algoritmo se llama **planificación por lotería** (Waldspurger y Weihl, 1994).

La idea básica consiste en dar a los procesos boletos de lotería para los diversos recursos del sistema, como el tiempo de CPU. Cada vez que se hace necesario tomar una decisión de planificación, se escoge al azar un boleto de lotería, y el proceso poseedor de ese boleto obtiene el recurso. Cuando se aplica a la planificación del tiempo de CPU, el sistema podría realizar una lotería 50 veces por segundo, concediendo a cada ganador 20 ms de tiempo de CPU como premio.

Parafraseando a George Orwell, “todos los procesos son iguales, pero algunos son más iguales que otros”. Podemos dar más boletos a los procesos más importantes, a fin de aumentar sus posibilidades de ganar. Si hay 100 boletos pendientes, y un proceso tiene 20 de ellos, tendrá una

probabilidad del 20% de ganar cada lotería. A largo plazo, obtendrá cerca del 20% del tiempo de CPU. En contraste con los planificadores por prioridad, donde es muy difícil establecer qué significa realmente tener una prioridad de 40, aquí la regla es clara: un proceso que posee una fracción de los boletos obtendrá aproximadamente una fracción f del recurso en cuestión.

La planificación por lotería tiene varias propiedades interesantes. Por ejemplo, si aparece un proceso nuevo y se le conceden algunos boletos, en la siguiente lotería ya tendrá una probabilidad de ganar que será proporcional al número de boletos que recibió. En otras palabras, la planificación por lotería es de respuesta muy rápida.

Los procesos cooperativos pueden intercambiar boletos si así lo desean. Por ejemplo, si un proceso cliente envía un mensaje a un proceso servidor y luego se bloquea, puede regalarle todos sus boletos al servidor, a fin de incrementar la probabilidad de que el servidor se ejecute a continuación. Una vez que el servidor termina, devuelve los boletos para que el cliente pueda ejecutarse otra vez. De hecho, en ausencia de clientes los servidores no necesitan boletos.

Podemos usar la planificación por lotería para resolver problemas que son difíciles de manejar con otros métodos. Un ejemplo es un servidor de video en el que varios procesos están alimentando corrientes de video a sus clientes, pero con diferente velocidad. Supongamos que los procesos requieren cuadros a razón de 10, 20 y 25 cuadros por segundo. Si asignamos a estos procesos 10, 20 y 25 boletos, respectivamente, se repartirán automáticamente la CPU en la proporción correcta.

2.4.7 Planificación en tiempo real

Un sistema de **tiempo real** es uno en el que el tiempo desempeña un papel esencial. Por lo regular, uno o más dispositivos físicos externos a la computadora generan estímulos, y la computadora debe reaccionar a ellos de la forma apropiada dentro de un plazo fijo. Por ejemplo, la computadora de un reproductor de discos compactos recibe los bits conforme salen de la unidad de disco y los debe convertir en música dentro de un intervalo de tiempo muy estricto. Si el cálculo toma demasiado tiempo, la música sonará raro. Otros sistemas de tiempo real son los de monitoreo de pacientes en las unidades de cuidado intensivo de los hospitales, el piloto automático de un avión y los controles de seguridad de un reactor nuclear. En todos estos casos, obtener la respuesta correcta pero demasiado tarde suele ser tan malo como no obtenerla.

Los sistemas de tiempo real generalmente se clasifican como de **tiempo real estricto**, lo que implica que hay plazos absolutos que deben cumplirse a como dé lugar, y **tiempo real flexible**, lo que implica que es tolerable no cumplir ocasionalmente con un plazo. En ambos casos, el comportamiento de tiempo real se logra dividiendo el programa en varios procesos, cada uno de los cuales tiene un comportamiento predecible y conocido por adelantado. Estos procesos generalmente son de corta duración y pueden ejecutarse hasta terminar en menos de un segundo. Cuando se detecta un suceso externo, el planificador debe programar los procesos de modo tal que se cumplan todos los plazos.

Los sucesos a los que puede tener que responder un sistema de tiempo real pueden clasificarse también como **periódicos** (que ocurren a intervalos regulares) o **aperiódicos** (que ocurren de forma impredecible). Es posible que un sistema tenga que responder a múltiples corrientes de eventos periódicos. Dependiendo de cuánto tiempo requiere cada suceso para ser procesado, tal vez ni

siquiera sea posible manejarlos todos. Por ejemplo, si hay m eventos periódicos y el evento i ocurre con el periodo P_i y requiere C_i segundos de tiempo de CPU para ser manejado, la carga sólo podrá manejarse si

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Un sistema de tiempo real que satisface este criterio es **planificable**.

Por ejemplo, consideremos un sistema de tiempo real flexible con tres sucesos periódicos, con periodos de 100, 200 y 500 ms respectivamente. Si estos eventos requieren 50, 30 y 100 ms de tiempo de CPU por evento, respectivamente, el programa es planificable porque $0.5 + 0.15 + 0.2 < 1$. Si se agrega un cuarto evento con un periodo de 1 s, el sistema seguirá siendo planificable en tanto este evento no necesite más de 150 ms de tiempo de CPU por evento. Un supuesto implícito en este cálculo es que el gasto extra de la conmutación de contexto es tan pequeño que puede ignorarse.

Los algoritmos de planificación de tiempo real pueden ser dinámicos o estáticos. Los primeros toman sus decisiones de planificación en el momento de la ejecución; los segundos las toman antes de que el sistema comience a operar. Consideremos brevemente unos cuantos algoritmos de planificación de tiempo real dinámicos. El algoritmo clásico es el **algoritmo de tasa monotónica** (Liu y Layland, 1973), que asigna por adelantado a cada proceso una prioridad proporcional a la frecuencia de ocurrencia de su evento disparador. Por ejemplo, un proceso que se debe ejecutar cada 20 ms recibe una prioridad de 50, y uno que debe ejecutarse cada 100 ms recibe una prioridad de 10. En el momento de la ejecución, el planificador siempre ejecuta el proceso listo que tiene la más alta prioridad, desalojando al proceso en ejecución si es necesario. Liu y Layland demostraron que este algoritmo es óptimo.

Otro algoritmo de planificación en tiempo real muy utilizado es el del **primer plazo más próximo**. Cada vez que se detecta un evento, su proceso se agrega a la lista de procesos listos, la cual se mantiene ordenada por plazo, que en el caso de un evento periódico es la siguiente ocurrencia del evento. El algoritmo ejecuta el primer proceso de la lista, que es el que tiene el plazo más próximo.

Un tercer algoritmo calcula primero para cada proceso la cantidad de tiempo que tiene de sobra, es decir, su **holgura**. Si un proceso requiere 200 ms y debe terminar en un plazo de 250 ms, tiene una holgura de 50 ms. El algoritmo, llamado **de menor holgura**, escoge el proceso que tiene menos tiempo de sobra.

Si bien en teoría es posible convertir un sistema operativo de aplicación general en uno de tiempo real usando uno de estos algoritmos de planificación, en la práctica el gasto extra de la conmutación de contexto de los sistemas de aplicación general es tan grande que el desempeño de tiempo real sólo puede lograrse en aplicaciones con restricciones de tiempo muy holgadas. En consecuencia, en la mayor parte de los trabajos en tiempo real se usan sistemas operativos de tiempo real especiales que tienen ciertas propiedades importantes. Por lo regular, éstas incluyen un tamaño pequeño, un tiempo de interrupción rápido, una conmutación de contexto rápida, un intervalo corto durante el cual se inhabilitan las interrupciones, y la capacidad para controlar múltiples cronómetros con precisión de milisegundos o microsegundos.

2.4.8 Planificación de dos niveles

Hasta ahora más o menos hemos supuesto que todos los procesos ejecutables están en la memoria principal. Si la memoria principal disponible no es suficiente, algunos de los procesos ejecutables tendrán que mantenerse en el disco total o parcialmente. Esta situación tiene implicaciones importantes para la planificación, ya que el tiempo de conmutación de procesos cuando hay que traer los procesos del disco es varios órdenes de magnitud mayor que cuando la conmutación es a un proceso que ya está en la memoria.

Una forma más práctica de manejar los procesos intercambiados a disco es el uso de un planificador de dos niveles. Primero se carga en la memoria principal un subconjunto de los procesos ejecutables, como se muestra en la Fig. 2-25(a). Luego, el planificador se limita a escoger procesos de este subconjunto durante cierto tiempo. Periódicamente se invoca un planificador de nivel superior para eliminar los procesos que han estado en memoria suficiente tiempo y cargar procesos que han estado en el disco demasiado tiempo. Una vez efectuado el cambio, como en la Fig. 2-25(b), el planificador de bajo nivel otra vez se limita a ejecutar procesos que están en la memoria. Así, este planificador se ocupa de escoger entre los procesos ejecutables que están en la memoria en ese momento, mientras el planificador de nivel superior se ocupa de trasladar procesos entre la memoria y el disco.

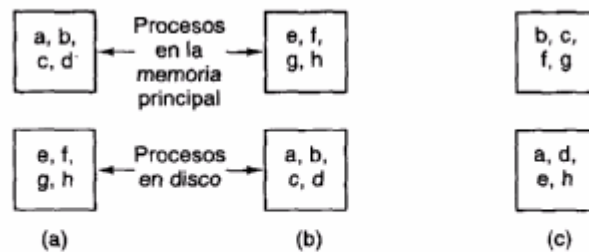


Figura 2-25. Un planificador de dos niveles debe transferir procesos entre el disco y la memoria y también escoger los procesos a ejecutar de entre los que están en la memoria. Representamos tres diferentes instantes con (a), (b) y (c).

Entre los criterios que el planificador de nivel superior podría usar para tomar sus decisiones están los siguientes:

1. ¿Cuánto tiempo hace que el proceso se intercambió del o al disco?
2. ¿Cuánto tiempo de CPU ha recibido el proceso recientemente?
3. ¿Qué tan grande es el proceso? (Los pequeños no estorban.)
4. ¿Qué tan alta es la prioridad del proceso?

Aquí también podríamos usar planificación round robin, por prioridad o por cualquiera de varios otros métodos. Los dos planificadores podrían usar el mismo algoritmo o algoritmos distintos.

2.4.9 Política vs. mecanismo

Hasta ahora, hemos supuesto tácitamente que todos los procesos del sistema pertenecen a diferentes usuarios y, por tanto, están compitiendo por la CPU. Si bien esto es conector en muchos casos, a veces sucede que un proceso tiene muchos hijos ejecutándose bajo su control. Por ejemplo, un proceso de sistema de administración de bases de datos podría tener muchos hijos. Cada hijo podría estar atendiendo una solicitud distinta, o cada uno podría tener una función específica que realizar (análisis sintáctico de consultas, acceso a disco, etc.). Es muy posible que el proceso principal tenga una idea excelente de cuáles de sus hijos son los más importantes (o para los que el tiempo es más crítico) y cuáles son los menos importantes. Desafortunadamente, ninguno de los planificadores que hemos visto aceptan entradas de los procesos de usuario relacionadas con las decisiones de planificación. Por tanto, el planificador casi nunca toma la mejor decisión.

La solución a este problema consiste en separar el **mecanismo de planificación** de la **política de planificación**. Esto significa que el algoritmo de planificación se regula de alguna manera mediante parámetros, y que estos parámetros pueden ser proporcionados por procesos de usuario. Consideremos otra vez el ejemplo de base de datos. Supongamos que el kernel usa un algoritmo de planificación por prioridad pero ofrece una llamada al sistema mediante el cual un proceso puede establecer (y modificar) las prioridades de sus hijos. De este modo, el padre puede controlar detalladamente la forma como sus hijos se planifican, aunque él en sí no realiza la planificación. Aquí el mecanismo está en el kernel pero la política es establecida por un proceso de usuario.

2.5 PERSPECTIVA GENERAL DE PROCESOS EN MINIX

Ahora que hemos completado nuestro estudio de los principios de la administración de procesos, la comunicación entre procesos y la planificación, daremos un vistazo a la forma como se aplican en MINIX. A diferencia de UNIX, cuyo kernel es un programa monolítico que no está dividido en módulos, MINIX es una colección de procesos que se comunican entre sí y con los procesos de usuario empleando una sola primitiva de comunicación entre procesos: la transferencia de mensajes. Este diseño confiere una estructura más modular y flexible y hace más fácil, por ejemplo, reemplazar todo el sistema de archivos por uno totalmente distinto, sin tener siquiera que recompilar el kernel.

2.5.1 La estructura interna de MINIX

Comencemos nuestro estudio de MINIX con una mirada a vuelo de pájaro del sistema. MINIX está estructurado en cuatro capas, cada una de las cuales realiza una función bien definida. Las cuatro capas se ilustran en la Hg. 2-26.

La capa inferior atrapa todas las interrupciones y trampas, realiza la planificación y ofrece a las capas superiores un modelo de procesos secuenciales independientes que se comunican empleando mensajes. El código de esta capa tiene dos funciones principales. La primera es atrapar las interrupciones y trampas, guardar y restaurar registros, planificar, y las demás tareas de bajo

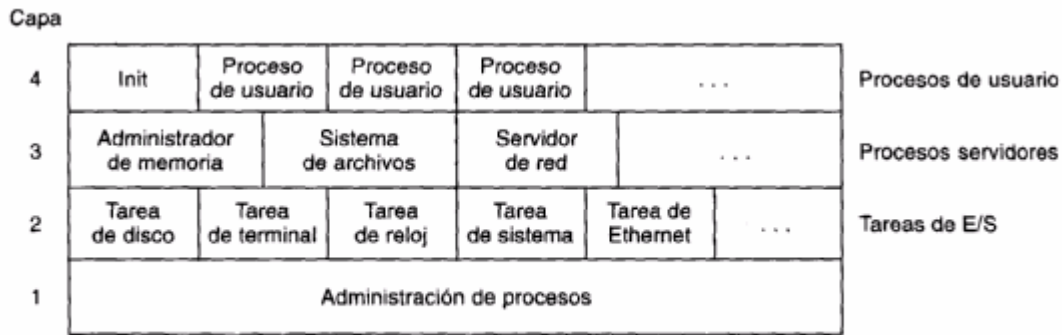


Figura 2-26. MINIX está estructurado en cuatro capas.

nivel necesarias para que funcione la abstracción de procesos que se presenta a las capas superiores. La segunda es manejar los aspectos mecánicos de los mensajes; verificar que los destinos sean válidos, ubicar los buffers de envío y recepción en la memoria física, y copiar bytes del emisor al receptor. La parte de la capa que se ocupa del nivel más bajo del manejo de interrupciones se escribe en lenguaje ensamblador. El resto de la capa y todas las demás capas de arriba se escriben en C.

La capa 2 contiene los procesos de E/S, uno por cada tipo de dispositivo. A fin de distinguirlos de los procesos de usuario ordinarios, los llamaremos tareas, pero las diferencias entre tareas y procesos son mínimas. En muchos procesos las tareas de entrada/salida se denominan manejadores de dispositivos; usaremos los términos “tarea” y “manejador de dispositivo” indistintamente. Se requiere una tarea para cada tipo de dispositivo, incluidos discos, impresoras, terminales, interfaces de red y relojes. Si están presentes otros dispositivos de E/S, se necesitará también una tarea para cada uno. Una de las tareas, la de sistema, es un poco distinta, ya que no corresponde a ningún dispositivo de E/S. Estudiaremos las tareas en el próximo capítulo.

Todas las tareas de la capa 2 y todo el código de la capa 1 se combinan para formar un solo programa binario llamado kernel. Algunas de las tareas comparten subrutinas comunes, pero por lo demás son independientes, se planifican por separado y se comunican usando mensajes. Los procesadores Intel a partir del 286 asignan uno de cuatro niveles de privilegio a cada proceso. Aunque las tareas y el kernel se compilan juntos, cuando el kernel y los manejadores de interrupciones se están ejecutando tienen más privilegios que las tareas. Así, el verdadero código de kernel puede acceder a cualquier parte de la memoria y a cualquier registro de procesador; en esencia, el kernel puede ejecutar cualquier instrucción usando datos de cualquier parte del sistema. Las tareas no pueden ejecutar todas las instrucciones de nivel de máquina, y tampoco pueden acceder a todos los registros de la CPU ni a todas las direcciones de memoria. Sin embargo, sí pueden acceder a regiones de la memoria que pertenecen a procesos menos privilegiados, con objeto de realizar E/S para ellos. Una tarea, la de sistema, no efectúa E/S en el sentido usual, sino que existe con el fin de proveer servicios, como el copiado entre diferentes regiones de la memoria, a procesos que no pueden hacer este tipo de cosas por sí mismos. Desde luego, en las máquinas que no proporcionan diferentes niveles de privilegio, como los procesadores Intel más antiguos, no es posible hacer que se cumplan estas restricciones.

La capa 3 contiene procesos que proporcionan servicios útiles a los procesos de usuario. Estos procesos servidores se ejecutan en un nivel menos privilegiado que el kernel y las tareas, y no pueden acceder directamente a los puertos de E/S. Estos procesos tampoco pueden acceder a la memoria fuera de los segmentos que les son asignados. El administrador de memoria (MM) ejecuta todas las llamadas al sistema de MINIX que intervienen en la administración de la memoria, como FORK, EXEC y BRK. El sistema de archivos (FS) ejecuta todas las llamadas al sistema relacionadas con archivos, como READ, MOUNT y CHDIR.

Como señalamos al principio del capítulo 1, los sistemas operativos hacen dos cosas: administran recursos y proporcionan una máquina extendida implementando llamadas al sistema. En MINIX la administración de recursos reside en gran medida en el kernel (capas 1 y 2), y la interpretación de las llamadas al sistema está en la capa 3. El sistema de archivos se diseñó como “servidor” de archivos y se puede trasladar a una máquina remota casi sin cambios. Esto se aplica también al administrador de memoria, aunque los servidores de memoria remotos no son tan útiles como los servidores de archivos remotos.

Pueden existir servidores adicionales en la capa 3. En la Fig. 2-26 se muestra ahí un servidor de red. Aunque MINIX, tal como se describe en este libro, no incluye el servidor de red, su código fuente forma parte de la distribución MINIX estándar; el sistema se puede recompilar fácilmente para incluirlo.

Éste es un buen lugar para mencionar que, si bien los servidores son procesos independientes, difieren de los procesos de usuario en cuanto a que se inician cuando el sistema se inicia, y nunca terminan mientras el sistema está activo. Además, aunque los servidores se ejecutan en el mismo nivel de privilegio que los procesos de usuario en términos de las instrucciones de máquina que tienen permitido ejecutar, reciben prioridad de ejecución más alta que los procesos de usuario. Si se desea incluir un nuevo servidor es necesario recompilar el kernel. El código de arranque del kernel instala los servidores en ranuras privilegiadas de la tabla de procesos antes de permitir que se ejecute cualquier proceso de usuario.

Por último, la capa 4 contiene todos los procesos de usuario: shells, editores, compiladores y programas a.out escritos por el usuario. Un sistema en ejecución por lo regular tiene algunos procesos que se inician cuando el sistema arranca y que se ejecutan indefinidamente. Por ejemplo, un demonio es un proceso de segundo plano que se ejecuta periódicamente o que espera continuamente algún evento, como la llegada de un paquete por la red. En cierto sentido, un demonio es un servidor que se inicia independientemente y se ejecuta como proceso de usuario. Sin embargo, a diferencia de los verdaderos servidores instalados en ranuras privilegiadas, tales programas no pueden recibir por parte del kernel el tratamiento especial que reciben los procesos servidores de memoria y de archivos.

23.2 Administración de procesos en MINIX

Los procesos en MINIX siguen el modelo general de procesos que describimos con cierto detalle al principio del capítulo. Los procesos pueden crear subprocesos, que a su vez pueden crear más subprocesos, produciendo un árbol de procesos. De hecho, todos los procesos de usuario del sistema forman parte de un solo árbol con `mit` (véase la Fig. 2-26) en su raíz.

¿Cómo surge esta situación? Cuando se enciende la computadora, el hardware lee el primer sector de la primera pista del disco de arranque y lo coloca en la memoria, y luego ejecuta el código que encuentra ahí. Los detalles varían dependiendo de si el disco de arranque es un disquete o el disco duro. En un disquete este sector contiene el programa de autoarranque (bootstrap), que es muy pequeño, pues debe caber en un sector. El autoarranque de MINIX carga un programa más grande, boot, que luego carga el sistema operativo propiamente dicho.

En cambio, los discos duros requieren un paso intermedio. Un disco duro está dividido en particiones, y el primer sector de un disco duro contiene un pequeño programa y la tabla de particiones del disco. Colectivamente, éstos se conocen como registro maestro de arranque. La parte de programa se ejecuta para leer la tabla de particiones y seleccionar la partición activa, la cual tiene un autoarranque en su primer sector, mismo que se carga y ejecuta para encontrar e iniciar una copia de boot en la partición, exactamente como se hace cuando se arranca con un disquete.

En ambos casos, boot busca un archivo multipartes en el disquete o la partición y carga las partes individuales en las posiciones apropiadas de la memoria. Estas partes incluyen el kernel, el administrador de memoria, el sistema de archivos e `mit`, el primer proceso de usuario. Este proceso de arranque no es una operación trivial. Las operaciones pertenecientes al ámbito de la tarea de disco y el sistema de archivos deben ser ejecutadas por boot antes de que estas partes del sistema queden activas. En una sección posterior volveremos al tema de cómo se inicia MINIX. Por ahora baste decir que una vez terminada la operación de carga el kernel comienza a ejecutarse.

Durante su fase de inicialización, el kernel inicia las tareas, y luego el administrador de memoria, el sistema de archivos y cualesquiera otros servidores que se ejecuten en la capa 3. Una vez que todos éstos se han ejecutado e inicializado a sí mismos, se bloquean, esperando algo que hacer. Una vez que todas las tareas y servidores están bloqueados, se ejecuta `mit`, el primer proceso de usuario. Este proceso ya está en la memoria principal, pero desde luego podría haberse cargado del disco como programa aparte, ya que todo está funcionando para cuando se inicia. Sin embargo, dado que `mit` se inicia sólo esta única vez y nunca se vuelve a cargar del disco, lo más fácil es incluirlo en el archivo de imagen del sistema junto con el kernel, las tareas y los servidores.

Lo primero que `init` hace es leer el archivo `/etc/ttvrab`, que lista todos los dispositivos de terminal potenciales. Los dispositivos que pueden usarse como terminales de inicio de sesión (en la distribución estándar, sólo la consola) tienen una entrada en el campo `getty` de `/etc/ttytab`, e `mit` bifurca un proceso hijo para cada una de esas terminales. Normalmente, cada hijo ejecuta `/usr/bin/getty` que exhibe un mensaje y luego espera que se teclee un nombre. Luego se invoca `lusrbini login` con ese nombre como argumento. Si una terminal en particular requiere un tratamiento especial (p. ej., una línea de marcado telefónico), `/etc/ttytab` puede especificar un comando (como `/usr/bin/stzy`) que se ejecutará para inicializar la línea antes de ejecutar `getty`.

Si se inicia la sesión con éxito, `lusrbini login` ejecuta el shell del usuario (especificado en el archivo `/etc/passwd`, y que normalmente es `/bin/lsh` o `/usr/bin/ash`). El shell espera que se tecleen comandos y luego bifurca un nuevo proceso para cada comando. De este modo, los shells son los hijos de `mit`, los procesos de usuario son los nietos de `mit`, y todos los procesos de usuario del sistema forman parte de un mismo árbol.

Las dos principales llamadas al sistema para administración de sistemas en MINIX son FORK y EXEC. FORK es la única forma de crear un proceso nuevo. EXEC permite a un proceso ejecutar un programa especificado. Cuando se ejecuta un programa, se le asigna una porción de memoria cuyo tamaño está especificado en la cabecera del archivo del programa. El programa conserva esta cantidad de memoria durante toda su ejecución, aunque la distribución entre segmento de datos, segmento de pila y espacio no utilizado puede variar durante la ejecución del proceso.

Toda la información referente a un proceso se guarda en la tabla de procesos, que se divide entre el kernel, el administrador de memoria y el sistema de archivos. Cada uno de éstos tiene los campos que necesita. Cuando nace un proceso nuevo (por FORK), o un proceso viejo termina (por EXrr o una señal), el administrador de memoria actualiza su parte de la tabla de procesos y luego envía mensajes al sistema de archivos y al kernel diciéndoles que hagan lo propio.

2.5.3 Comunicación entre procesos en MINIX

Se cuenta con tres primitivas para enviar y recibir mensajes, las cuales se invocan con los procedimientos de biblioteca de C

```
send(destino, &mensaje);
```

para enviar un mensaje al proceso destino,

```
receive(origen, &mensaje);
```

para recibir un mensaje del proceso origen (o ANY) y

```
send_rec(org_dest, &mensaje);
```

para enviar un mensaje y esperar una respuesta del mismo proceso. El segundo parámetro en todas estas llamadas es la dirección local de los datos del mensaje. El mecanismo de transferencia de mensajes del kernel copia el mensaje del emisor al receptor. La respuesta (en el caso de send_rec) sobrescribe el mensaje original. En principio, este mecanismo del kernel podría ser reemplazado por una función que copia mensajes a través de una red a una función correspondiente en otra máquina, a fin de implementar un sistema distribuido. En la práctica esto se complicaría un poco por el hecho de que el contenido de los mensajes a veces es un apuntador a una estructura de datos grande, y un sistema distribuido tendría que incluir una forma de copiar los datos mismos por la red.

Cada proceso o tarea puede enviar y recibir mensajes de procesos y tareas de su propia capa, y de aquellos en la capa que está directamente abajo. Los procesos de usuario no pueden comunicarse directamente con las tareas de E/S. El sistema obliga a cumplir esta restricción.

Cuando un proceso (lo que también incluye las tareas como caso especial) envía un mensaje aun proceso que actualmente no está esperando un mensaje, el emisor se bloquea hasta que el destino ejecuta RECEIVE. En otras palabras, MINIX usa el método de cita para evitar los problemas almacenar temporalmente los mensajes enviados que todavía no se han recibido. Aunque esto es menos flexible que un esquema con buffers, resulta ser suficiente para este sistema, y mucho más sencillo, ya que no se requiere administración de buffers.

2.5.4 Planificación de procesos en MINIX

El sistema de interrupciones es lo que mantiene funcionando a un sistema operativo con multiprogramación. Los procesos se bloquean cuando solicitan entradas, permitiendo la ejecución de otros procesos. Una vez que están disponibles las entradas, el disco, el teclado u otro hardware interrumpe el proceso que se está ejecutando. El reloj también genera interrupciones que sirven para asegurar que un proceso de usuario en ejecución que no haya solicitado entradas tarde o temprano ceda la CPU para que otros procesos tengan oportunidad de ejecutarse. Es obligación de la capa más baja de MINIX ocultar estas interrupciones convirtiéndolas en mensajes. En lo que a los procesos (y mensajes) concierne, cuando un dispositivo de E/S completa una operación envía un mensaje a algún proceso, despertándolo y haciéndolo ejecutable.

Cada vez que un proceso es interrumpido, sea por un dispositivo de E/S convencional o por el reloj, surge la oportunidad de volver a determinar cuál proceso es el que más merece una oportunidad de ejecutarse. Desde luego, esto debe hacerse también cada vez que un proceso termina, pero en un sistema como MINIX las interrupciones causadas por operaciones de E/S o el reloj ocurren con mucha mayor frecuencia que la terminación de procesos. El planificador de MINIX usa un sistema de colas multinivel con tres niveles, que corresponden a las capas 2, 3 y 4 de la Fig. 2-26. Dentro de los niveles de tareas y servidores los procesos se ejecutan hasta que se bloquean, pero los procesos de usuario se planifican round robin. Las tareas tienen la prioridad más alta, les siguen el administrador de memoria y el servidor de archivos, y los procesos de usuario vienen al final.

Al escoger el proceso que se va a ejecutar, el planificador verifica si hay alguna tarea lista. Si una o más están listas, se ejecuta la que esté a la cabeza de la cola. Si no hay tareas listas, se escoge un servidor (MM o FS), si es posible; en caso contrario, se ejecuta un proceso de usuario. Si no hay ningún proceso listo, se escoge el proceso IDLE. Éste es un ciclo que se ejecuta hasta que ocurre la siguiente interrupción.

En cada tic del reloj, se verifica si el proceso en curso es un proceso de usuario que se ha ejecutado durante más de 100 ms. Si así es, se invoca el planificador para ver si hay otro proceso de usuario esperando la CPU. Si se encuentra uno, el proceso actual se pasa al final de su cola de planificación y se ejecuta el proceso que está a la cabeza de dicha cola. Las tareas, el administrador de memoria y el sistema de archivos nunca son desalojados por el reloj, aunque se hayan estado ejecutando durante mucho tiempo.

2.6 IMPLEMENTACIÓN DE PROCESOS EN MINIX

Ya nos estamos acercando al momento en que examinaremos el código real, así que es pertinente incluir aquí unas cuantas palabras acerca de la notación que usaremos. Los términos “procedimiento”, “función” y “rutina” se usarán indistintamente. Los nombres de variables, procedimientos y archivos se escribirán en cursivas, como en *rw_flag*. Cuando un nombre de variable, procedimiento o archivo inicie una oración, se escribirá con mayúscula, pero todos los nombres reales comienzan con minúsculas. Las llamadas al sistema estarán en versalitas, como READ.

El libro y el software, que están en continua evolución, no se “fueron a la imprenta” el mismo día, así que puede haber discrepancias menores entre las referencias al código, el listado impreso y la versión en CD-ROM. Sin embargo, tales diferencias generalmente sólo afectan una o dos líneas. Además, el código fuente aquí impreso se simplificó eliminando el que se usó para compilar opciones que no se explican en el libro.

2.61 Organización del código fuente de MINIX

Lógicamente, el código fuente está organizado en dos directorios. Las rutas completas a estos directorios en un sistema MINIX estándar son `/usr/include/` y `/usr/src/` (una diagonal final en un nombre de ruta indica que se refiere a un directorio). La ubicación real de los directorios puede variar de un sistema a otro, pero normalmente la estructura de los directorios por debajo del nivel más alto es la misma en todos los sistemas. Nos referiremos a estos directorios como `include/` y `src/` en este texto.

El directorio `include/` contiene varios archivos de cabecera Posix estándar; además, tiene tres subdirectorios:

1. `sys/` — este subdirectorio contiene cabeceras Pos adicionales.
2. `minix/` — incluye archivos de cabecera utilizados por el sistema operativo.
3. `ibm/` — incluye archivos de cabecera con definiciones específicas para IBM-PC.

A fin de manejar extensiones de MINIX y programas que se ejecutan en el entorno MINIX, hay otros directorios presentes en `include/` en la versión proporcionada en el CD-ROM o por Internet. Por ejemplo, el directorio `include/neti` y su subdirectorio `include/net/gen/` apoyan las extensiones de red. Sin embargo, en este texto sólo se imprimirán y explicarán los archivos necesarios para compilar el sistema MINIX básico.

El directorio `src/` contiene tres subdirectorios importantes en los que se encuentra el código fuente del sistema operativo:

1. `kernel/` — capas 1 y 2 (procesos, mensajes y controladores).
2. `mm/` — el código del administrador de memoria.
3. `fs/` — el código del sistema de archivos.

Hay otros tres directorios de código fuente que no se imprimen ni explican en el texto, pero que son esenciales para producir un sistema funcional:

1. `src/lib/` — código fuente de los procedimientos de biblioteca (p. ej., `open`, `read`).
2. `src/tools/` — código fuente del programa `mit`, con el que se inicia MINIX.
3. `src/boot/` — código para arrancar e instalar MINIX.

La distribución estándar de MINIX incluye varios directorios fuente más. Desde luego, un sistema operativo existe para apoyar los comandos (programas) que se ejecutarán en él, así que hay un

directorio `src/commands/` de gran tamaño con el código fuente para los programas de utilidad (p. ej., `cat`, `cp`, `date`, `is`, `pwd`). Puesto que MINIX es un sistema operativo educativo modificable, existe un directorio `src/test/` con programas diseñado para probar exhaustivamente un sistema MINIX recién compilado. Por último, el directorio `/src/inet/` incluye código fuente para recompilar MINIX con apoyo de red.

Por comodidad normalmente nos referiremos a los nombres de archivo simples si por el contexto queda claro cuál es el nombre completo de la ruta. No obstante, cabe señalar que algunos nombres de archivo aparecen en más de un directorio. Por ejemplo, hay varios archivos llamados `const.h` en los que se definen constantes pertinentes a una parte específica del sistema. Los archivos de un directorio en particular se explicarán juntos, así que no debería haber confusiones. Los archivos se listan en el apéndice A en el orden en que se explican en el texto, a fin de que sea más fácil seguir las explicaciones. En este punto podría ser una buena idea conseguir un par de señadores para poder pasar fácilmente de una parte del libro a otra.

También vale la pena señalar que el apéndice B contiene la lista alfabética de todos los archivos descritos en el apéndice A, y el apéndice C contiene una lista de dónde pueden encontrarse las definiciones de macros, variables globales y procedimientos empleados en MINIX.

El código para las capas 1 y 2 está contenido en el directorio `src/kernel/`. En este capítulo estudiaremos los archivos de este directorio que apoyan la administración de procesos, la capa más baja de la estructura de MINIX que vimos en la Fig. 2-26. Esta capa incluye funciones que se encargan de la inicialización del sistema, las interrupciones, la transferencia de mensajes y la planificación de procesos. En el capítulo 3 veremos el resto de los archivos de este directorio, los cuales apoyan las diversas tareas, mismas que constituyen la segunda capa de la Fig. 2-26. En el capítulo 4 examinaremos los archivos del administrador de memoria que están en `src/mm/`, y en el capítulo 5 estudiaremos el sistema de archivos, cuyos archivos fuente se encuentran en `src/fs/`.

Una vez que MINIX se compila, todos los archivos de código fuente de `src/kernel/`, `src/mml` y `src/fs/` producen archivos objeto. Todos los archivos objeto de `src/kernel/` se enlazan para formar un solo programa ejecutable, `kernel`. Los archivos objeto de `src/mm/` también se enlazan para formar un solo programa ejecutable, `mm`, y algo análogo sucede con `fs`. Se pueden agregar extensiones incluyendo servidores adicionales; por ejemplo, el apoyo de red se agrega modificando `include/minix/config.h` a modo de habilitar la compilación de los archivos de `src/inet/`, que constituirán `inet`. Otro programa ejecutable, `in`, se construye en `src/toolst`. El programa `installboot` (cuya fuente está en `src/boot/`) agrega nombres a cada uno de estos programas, los rellena de modo que su longitud sea un múltiplo del tamaño de un sector del disco (a fin de facilitar la carga independiente de las partes) y los concatena para formar un solo archivo. Este nuevo archivo es el binario del sistema operativo y puede copiarse en el directorio raíz o en el directorio `Iminix1` de un disco flexible o de una partición del disco duro. Posteriormente, el programa monitor del arranque podrá cargar y ejecutar el sistema operativo. En la Fig. 2-27 se muestra la organización de la memoria después de que los programas concatenados se han separado y cargado. Desde luego, los detalles dependen de la configuración del sistema. El ejemplo de la figura es para un sistema MINIX configurado a modo de aprovechar una computadora equipada con varios megabytes de memoria. Esto hace posible repartir un gran número de buffers del sistema de archivos, pero el sistema de archivos resultante es demasiado grande para caber en el intervalo inferior de la memo-

ria. por debajo de los 640K. Si el número de buffers se reduce drásticamente, es posible lograr que lado el sistema quepa en menos de 640K de memoria, con espacio incluso para unos cuantos procesos de usuario.

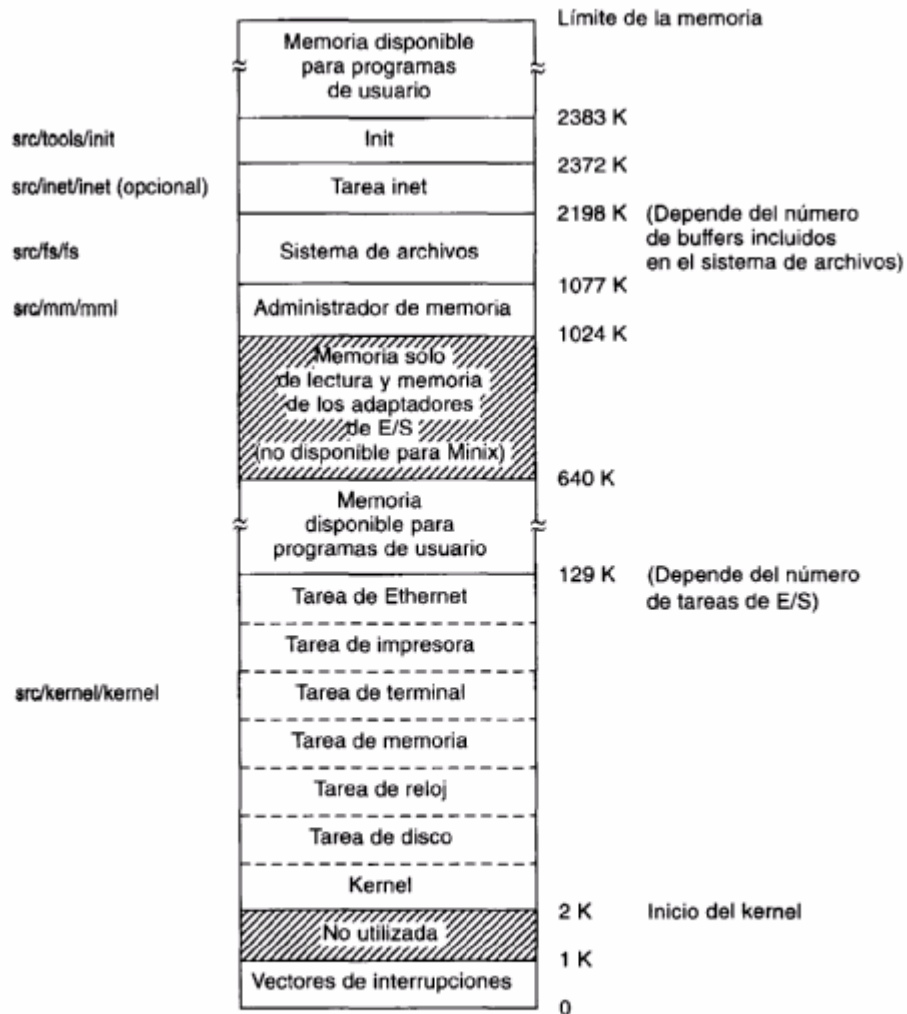


Figura 2-27. Organización de la memoria después de que MINIX se ha cargado del disco a la memoria. Las cuatro (o cinco, con el apoyo de red) partes, compiladas independientemente y enlazadas, son obviamente distintas. Los tamaños son aproximados, dependiendo de la configuración.

Es importante percatarse de que MINIX consiste en tres o más programas totalmente independientes que sólo se comunican entre sí por medio de mensajes. Un procedimiento llamado sic en src/fs/ no entra en conflicto con otro del mismo nombre en src/mm/ porque en última instancia se enlazan dentro de diferentes archivos ejecutables. Los únicos procedimientos que tienen en común los tres componentes del sistema operativo son unas cuantas de las rutinas de

biblioteca que están en `lib/`. Esta estructura modular hace que sea muy fácil modificar, digamos, el sistema de archivos sin que los cambios realizados afecten el administrador de memoria; además hace que no sea complicado quitar todo el sistema de archivos y colocarlo en una máquina distinta como servidor de archivos que se comunicará con las máquinas usuarias enviando mensajes por una red.

Como ejemplo adicional de la modularidad de MINIX, al administrador de memoria y al sistema de archivos les da exactamente lo mismo si el sistema se compila con o sin apoyo de red, y el kernel sólo resulta afectado porque la tarea de Ethernet se compila ahí, junto con el apoyo para otros dispositivos de E/S. Una vez habilitado, el servidor de red se integra al sistema MINIX como servidor con el mismo nivel de prioridad que el administrador de memoria o el servidor de archivos. Su funcionamiento puede implicar la transferencia de grandes cantidades de datos con gran rapidez, y esto requiere una prioridad más alta que la que recibiría un proceso de usuario. Sin embargo, con excepción de la tarea de Ethernet, las funciones de red podrían llevarse a cabo con procesos en el nivel de usuario. Tales funciones no son tradicionalmente funciones del sistema operativo, y una explicación detallada del código de red rebasa el alcance de este libro. En secciones y capítulos subsecuentes las explicaciones se basarán en un sistema MINIX compilado sin apoyo de red.

2.6.2 Los archivos de cabecera común

El directorio `include/` y sus subdirectorios contienen una colección de archivos que definen constantes, macros y tipos. La norma `iosix` requiere muchas de estas definiciones y especifica en qué archivos del directorio principal `include/` y de su subdirectorio `include/sys/` se encuentra cada una de las definiciones obligatorias. Los archivos de estos directorios son archivos de cabecera o de inclusión, identificados por el sufijo `.h`, y se incluyen mediante instrucciones `#include` en los archivos fuente en C. Estas instrucciones son una característica del lenguaje C. Los archivos de inclusión facilitan el mantenimiento de los sistemas grandes.

Las cabeceras que probablemente se necesitarán para compilar programas de usuario se encuentran en `include/`, mientras que `include/sys/` tradicionalmente se ha usado para archivos que sirven principalmente para compilar programas del sistema y de rutinas de utilidad. La distinción no es demasiado importante, y una compilación típica, sea de un programa de usuario o de una parte del sistema operativo, incluirá archivos de ambos directorios. Aquí veremos los archivos que se necesitan para compilar el sistema MINIX estándar, examinando primero los que están en `include/` y luego los que están en `include/syst`. En la siguiente sección hablaremos de todos los archivos de los directorios `include/minix/` e `include/ibml` que, como indican los nombres de directorio, son exclusivos de MINIX y de su implementación en computadoras tipo IBM.

Las primeras cabeceras que consideraremos son en verdad de aplicación general, al grado de que ninguno de los archivos fuente en C para el sistema MINIX hace referencia directamente a ellas; más bien, están incluidas en otros archivos de cabecera, las cabeceras maestras `src/kernel/kernel.h`, `src/mm/mm.h` y `src/fs/fs.h` para cada una de las tres partes principales del sistema MINIX, que a su vez se incluyen en todas las compilaciones. Cada cabecera maestra está adaptada a las necesidades de la parte correspondiente del sistema MINIX, pero todas comienzan con una sección como la que se muestra en la Fig. 2-28. Hablaremos más de las cabeceras maestras en otras


```

#include <minix/config.h>           /* DEBE ser el primero */
#include <ansi.h>                  /* DEBE ser el segundo */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <limits.h>
#include <errno.h>
#include <minix/syslib.h>

```

Figura 2-28. Parte de una cabecera maestra que asegura la inclusión de los archivos de cabecera que todos los archivos fuente en C necesitan.

secciones del libro. Este avance tiene como propósito subrayar el hecho de que se juntan cabeceras de varios directorios. En esta sección y en la siguiente mencionaremos cada uno de los archivos a los que se hace referencia en la Fig. 2-2 8.

Comencemos con la primera cabecera de include/, ansi.h (línea 0000). Ésta es la segunda cabecera que se procesa cada vez que se compila una parte del sistema MINIX; sólo include/minix/config.h se procesa antes. El propósito de ansi.h es probar si el compilador satisface los requisitos de Standard C, definidos por la Organización Internacional de Normas (ISO). Standard C también se llama ANSI C, puesto que el estándar fue creado originalmente por el American National Standards Institute antes de adquirir reconocimiento internacional. Un compilador de Standard C define varias macros que pueden probarse después en los programas que se compilan. `__STDC__` es una de esas macros, y un compilador estándar la define con un valor de 1, igual que si el preprocesador de C hubiera leído una línea como

```
#define __STDC__ 1
```

El compilador distribuido con las versiones actuales de MINIX se ajusta al Standard C, pero las versiones antiguas de MINIX se crearon antes de la adopción del estándar y aun es posible compilar MINIX con un compilador de C clásico (Kemighan & Ritchie). Se pretende que MINIX sea fácil de trasladar a máquinas nuevas, y permitir compiladores antiguos es parte de esta política. En las líneas 0023 a 0025 se procesa la instrucción

```
#define _ANSI
```

si se está usando un compilador de Standard C. Ansi.h define varias macros de diferentes formas, dependiendo de si está definida o no la macro `_ANSI`.

La macro más importante de este archivo es `_PROTOTYPE`. Esta macro nos permite escribir prototipos de funciones de la forma

```
_PROTOTYPE (tipo_devuelto, nombre_func, (tipo_argumento argumento,...))
```

y que el preprocesador de C los transforme en

```
tipo_devuelto nombre_func(tipo-argumento, argumento, ...)
```

si el compilador es de ANSI Standard C, o

```
tipo_devuelto nombre_func()
```

si el compilador es del tipo antiguo (Kernighan & Ritchie).

Antes de dejar `ansi.h` mencionaremos otra característica. Todo el archivo está delimitado por líneas que dicen

```
#ifndef _ANSI_H
```

```
y
```

```
#endif
```

En la línea que sigue inmediatamente a `#ifndef`, se define `_ANSI_H` misma. Un archivo de cabecera sólo debe incluirse una vez en una compilación; esta construcción asegura que se hará caso omiso del contenido del archivo si se le incluye más de una vez. Veremos que se usa esta técnica en todos los archivos de cabecera del directorio `include/`.

El segundo archivo de `include/` que se incluye indirectamente en todos los archivos fuente de MINIX es la cabecera `limits.h` (línea 0100). Este archivo define muchos tamaños básicos, tanto de tipos del lenguaje como el número de bits de un entero, así como límites del sistema operativo como la longitud de un nombre de archivo. `errno.h` (línea 0200) también es incluido por todas las cabeceras maestras; contiene los números de error devueltos a los programas de usuario en la variable global `errno` cuando una llamada al sistema falla. También se usa `errno` para identificar algunos errores internos, como un intento por enviar un mensaje a una tarea inexistente. Los números de error son negativos para marcarlos como códigos de error dentro del sistema MINIX, pero deben convertirse en positivos antes de devolverse a los programas de usuario. El truco que se usa es que cada código de error se define en una línea como

```
#define EPERM (_SIGN 1)
```

(línea 0236). La cabecera maestra de cada parte del sistema operativo define la macro `_SYSTEM`, pero ésta nunca se define cuando se compila un programa de usuario. Si se define `_SYSTEM`, entonces `_SIGN` se define como “—”; en caso contrario, se le da una definición nula.

El siguiente grupo de archivos que consideraremos no se incluye en todas las cabeceras maestras, pero de todos modos se usa en muchos archivos fuente en todas las partes del sistema MINIX. El más importante es `unistd.h` (línea 0400). Esta cabecera define muchas constantes, la mayor parte de las cuales son requeridas por `posix`. Además, `unistd.h` incluye prototipos de muchas funciones en C, incluidas todas las que se usan para acceder a las llamadas al sistema de MINIX. Otro archivo que se utiliza mucho es `string.h` (línea 0600), que incluye prototipos de muchas funciones en C que sirven para manipular cadenas. La cabecera `signal.h` (línea 0700) define los nombres de las señales estándar, y también contiene prototipos de algunas funciones relacionadas con señales. Como veremos después, en el manejo de señales intervienen todas las partes de MINIX.

`Fcntl.h` (línea 0900) define simbólicamente muchos parámetros empleados en operaciones de control de archivos. Por ejemplo, este archivo nos permite usar la macro `O_RDONLY` en lugar del valor numérico `O` como parámetro de una llamada `open`. Aunque el sistema de archivos es el que

más hace referencia a este archivo, sus definiciones también se necesitan en varios lugares del kernel y del administrador de memoria.

El resto de los archivos de `include/` no se utiliza tan ampliamente como los que ya mencionamos. `Stdlib.h` (línea 1000) define tipos, macros y prototipos de funciones que probablemente se requerirán en la compilación de todos los programas en C, con excepción de los más sencillos. Esta cabecera es una de las que se utilizan con mayor frecuencia al compilar programas de usuario, aunque dentro de la fuente del sistema MINIX sólo hacen referencia a ella unos cuantos archivos del kernel.

Como veremos cuando examinemos la capa de tareas en el capítulo 3, la interfaz con la consola y las terminales de un sistema operativo es compleja porque muchos tipos diferentes de hardware deben interactuar con el sistema operativo y los programas de usuario de una forma estandarizada. La cabecera `termios.h` (línea 1100) define constantes, macros y prototipos de funciones que se emplean para controlar dispositivos de E/S tipo terminal. La estructura más importante es `termios`, que contiene banderas para indicar diversos modos de operación, variables para fijar las velocidades de transmisión de entrada y salida, y un arreglo para contener caracteres especiales, como los caracteres `INTR` y `KILL`. `POS` exige esta estructura, lo mismo que muchas de las macros y prototipos de funciones definidos en este archivo.

Sin embargo, a pesar de que se pretende que el estándar `POS` abarque todo, no proporciona todo lo que podríamos querer, y la última parte del archivo, de la línea 1241 en adelante, provee extensiones de `POSIX`. Algunas de éstas tienen un valor obvio, como las extensiones para definir velocidades de transmisión de 57 600 baudios y superiores, y el apoyo para ventanas en la pantalla de la terminal. El estándar `OSiX` no prohíbe las extensiones, ya que ningún estándar razonable puede abarcar todo lo habido y por haber; pero al escribir un programa en el entorno MINIX que pretenda ser portátil a otros entornos, se requiere cierta cautela para evitar el uso de definiciones específicas para MINIX. Esto es fácil. En este archivo y otros que definen extensiones específicas para MINIX, el empleo de las extensiones está controlado por una instrucción

```
#ifndef _MINIX
```

Si no se ha definido `_MINIX`, el compilador ni siquiera verá las extensiones para MINIX.

El último archivo que veremos en `include/` es `a. outh` (línea 1400), una cabecera que define el formato de los archivos en los que se almacenan programas ejecutables en disco, incluida la estructura de cabecera que sirve para iniciar la ejecución de un archivo, y la estructura de tabla de símbolos producida por el compilador. Sólo el sistema de archivos hace referencia a esta cabecera.

Pasemos ahora al subdirectorio `include/sysi`. Como se muestra en la Fig. 2-28, todas las cabeceras maestras de las partes principales del sistema MINIX incluyen `sys/types.h` (línea 1600) inmediatamente después de leer `ansi.h`. Esta cabecera define muchos tipos de datos empleados por MINIX. Los errores que podrían surgir si no se entiende bien cuáles tipos de datos fundamentales se usan en una situación en particular pueden evitarse si se utilizan las definiciones provistas en esta cabecera. La Fig. 2-29 muestra la forma en que difieren los tamaños, en bits, de unos cuantos tipos definidos en este archivo cuando se compilan para procesadores de 16 o de 32 bits. Observe que los nombres de tipo terminan con “`_t`”. Esto no es sólo una convención, sino un requisito del estándar `POSIX`. Éste es un ejemplo de sufijo reservado, y no debe usarse como sufijo de ningún nombre que no sea un nombre de tipo.

Tipo	MINIX de 16 bits	MINIX de 32 bits
gid_t	8	8
dev_t	16	16
pid_t	16	32
ino_t	16	32

Figura 2-29. El tamaño, en bits, de algunos tipos en sistemas de 16 y de 32 bits.

Aunque no se utiliza tan ampliamente como para que se incluya en las cabeceras de todas las secciones, `sysioctl.h` (línea 1800) define muchas macros empleadas para operaciones de control de dispositivos. Esta cabecera también contiene el prototipo de la llamada al sistema `IOCTL`. En muchos casos, los programadores no invocan directamente esta llamada, ya que las funciones definidas por `rosix` cuyos prototipos están en `includeterms.h` han sustituido muchos usos de la antigua función de biblioteca `ioctl` para manejar terminales, consolas y dispositivos similares; no obstante, sigue siendo necesaria. De hecho, las funciones POSIX para controlar dispositivos de terminal son convertidas en llamadas `IOCTL` por la biblioteca. Además, hay un número cada vez mayor de dispositivos, todos los cuales necesitan diversos tipos de control, que pueden conectarse a un sistema de computadora moderno. Por ejemplo, cerca del final de este archivo se definen varios códigos de operación que comienzan con `DSPIO` y que sirven para controlar un procesador de señales digitales. De hecho, la diferencia principal entre MINIX tal como se describe en este libro y otras versiones es que para los fines del libro describimos un MINIX con relativamente pocos dispositivos de entrada/salida. Es posible agregar muchos más, como interfaces de red, unidades de CD-ROM y tarjetas de sonido; los códigos de control para todos éstos se definen como macros en este archivo.

Varios otros archivos de este directorio se utilizan ampliamente en el sistema MINIX. El archivo `sys/sigcontext.h` (línea 2000) define estructuras que se usan para preservar y restablecer la operación normal del sistema antes y después de la ejecución de una rutina de manejo de señales y se usa tanto en el kernel como en el administrador de memoria. MINIX ofrece apoyo para rastrear ejecutables y analizar vaciados de núcleo con un programa depurador, y `sys/ptrace.h` (línea 2200) define las diversas operaciones posibles con la llamada al sistema `PTRACE`. `sys/stat.h` (línea 2300) define la estructura que vimos en la Fig. 1-12 y que es devuelta por las llamadas al sistema `STAT` y `FSTAT`, así como los prototipos de las funciones `stat` y `fstat` y otras funciones que sirven para manipular las propiedades de los archivos. Se hace referencia a esta cabecera en varias partes del sistema de archivos y del administrador de memoria.

Los dos últimos archivos que mencionaremos en esta sección no tienen tantas referencias como los que vimos antes. `sys/dir.h` (línea 2400) define la estructura de una entrada de directorio en MINIX. Sólo se hace referencia directa a él una vez, pero esta referencia lo incluye en otra cabecera que se utiliza ampliamente en el sistema de archivos. `sys/dir.h` es importante porque, entre otras cosas, indica cuántos caracteres puede contener un nombre de archivo. Por último, la cabecera `sys/wait.h` (línea 2500) define macros utilizadas por las llamadas al sistema `WAIT` y `WAITPID`, que se implementan en el administrador de memoria.

2.6.3 Los archivos de cabecera de MINIX

Los subdirectorios `include/minixi` e `include/ibm/` contienen archivos de cabecera específicos para MINIX. Los archivos de `include/minixlse` necesitan para implementar MINIX en cualquier plataforma, aunque hay definiciones alternativas, específicas para una plataforma, dentro de algunos de ellos. Los archivos de `include/ibm/` definen estructuras y macros específicas para MINIX tal como se implementa en máquinas tipo IBM.

Comenzaremos con el directorio `mmix`. En la sección anterior señalamos que `config.h` (línea 2600) se incluye en las cabeceras maestras de todas las partes del sistema MINIX, y por tanto es el primer archivo que el compilador procesa realmente. En muchas ocasiones en las que diferencias `ene!` hardware o en la forma en que se pretende que se use el sistema operativo requieren cambios a la configuración de MINIX, lo único que se necesita es editar este archivo y recompilar el sistema. Todos los parámetros ajustables por el usuario están en la primera parte del archivo. El primero de ellos es el parámetro `MACHINE`, que puede adoptar valores como `IBM_PC`, `SUN_4`, `MACiNTOSH` u otros, dependiendo del tipo de máquina para la cual se está compilando MINIX. La mayor parte del código de MINIX es independiente del tipo de máquina, pero un sistema operativo siempre tiene algo de código dependiente del sistema. En los pocos lugares de este libro en los que hablaremos de código que se escribe de forma diferente para distintos sistemas usaremos como ejemplo código escrito para máquinas tipo IBM PC con chips procesadores avanzados (80386, 80486, Pentium, Pentium Pro) que usan palabras de 32 bits. Nos referiremos a todos éstos como procesadores Intel de 32 bits. También puede compilarse MINIX para IBM PC más viejas con tamaño de palabra de 16 bits, y las partes de MINIX que dependen de la máquina deben codificarse de forma diferente para estas máquinas. En una PC, el compilador mismo determina el tipo de máquina para la cual se compilará MINIX. El compilador estándar de MINIX para PC es el del Amsterdam Compiler Kit (ACK), el cual se identifica a sí mismo definiendo, además de la macro `__STDC__`, la macro `__ACK__`. Este compilador también define una macro `_EM_WSIZE` que es el tamaño de palabra (en bytes) de su máquina objetivo. En la línea 2626 se asigna el valor de `_EM_WSIZE` a una macro llamada `_WORD_SIZE`. Más adelante en el archivo y en diversos lugares de los demás archivos fuente de MINIX se utilizan estas definiciones. Por ejemplo, las líneas 2647 a 2650 comienzan con la prueba

```
#if (MACHINE == IBM_PC && _WORD_SIZE == 4)
```

y definen un tamaño para el caché de buffers del sistema de archivos en los sistemas de 32 bits.

Otras definiciones de `config.h` permiten personalizar otras necesidades de una instalación en particular. Por ejemplo, hay una sección que permite incluir diversos tipos de controladores de dispositivos cuando se compila el kernel de MINIX. Es probable que ésta sea la parte del código fuente de MINIX que se edite con mayor frecuencia. Esta sección comienza con:

```
#define ENABLE_NETWORKING 0
#define ENABLE_AT_WINI 1
#define ENABLE_BIOS_WINI 0
```

Si cambiamos el `0` de la primera línea a `1` podremos compilar un kernel de MINIX para una máquina

que necesita apoyo de red. Si definimos `ENABLE_AT_WINI` como 0 y `ENABLE_BIOS_WINI` como 1, podemos eliminar el código de controlador de disco duro tipo AT (es decir, IDE) y usare/ PC BIOS para el apoyo de disco duro.

El siguiente archivo es `const.h` (línea 2900) que ilustra otro uso común de los archivos de cabecera. Aquí encontramos diversas definiciones de constantes que con toda seguridad no se modificarán al compilar un nuevo kernel pero que se usan en varios lugares. Al definir las aquí se ayuda a prevenir errores que podrían ser difíciles de rastrear si se efectuaran definiciones inconsistentes en múltiples puntos. Hay otros archivos llamados `const.h` en el árbol fuente de MINIX, pero su uso es más limitado. Las definiciones que se usan sólo en el kernel se incluyen en `src/kernel/const.h`. Las definiciones que se usan sólo en el sistema de archivos se incluyen en `src/fs/const.h`. El administrador de memoria usa `src/mm/const.h` para sus definiciones locales. Sólo las definiciones que se usan en más de una parte del sistema MINIX se incluyen en `include/minix/const.h`.

Unas cuantas de las definiciones de `const.h` merecen mención especial. `EXTERN` se define como una macro que se expande para crear `extern` (línea 2906). Las variables globales que se declaran en archivos de cabecera y se incluyen en dos o más archivos se declaran `EXTERN`, como en

```
EXTERN mt who;
```

Si la variable se declarara simplemente como

```
int who;
```

y se incluyera en dos o más archivos, algunos enlazadores protestarían por haberse definido una variable más de una vez. Además, el manual de referencia de C (Kernighan y Ritchie, 1988) prohíbe explícitamente esta construcción.

A fin de evitar este problema, es necesario escribir la declaración de la siguiente manera:

```
extern in who;
```

en todos los lugares menos uno. El empleo de `EXTERN` previene este problema porque se hace que se expanda a `extern` en todos los lugares en que se incluye `const.h`, excepto después de una redefinición explícita de `EXTERN` como la cadena nula. Esto se hace en cada una de las partes de MINIX colocando las definiciones globales en un archivo especial llamado `glo.h`; por ejemplo, `src/kernel/glo.h`, que se incluye indirectamente en todas las compilaciones. Dentro de cada `glo.h` hay una secuencia

```
#ifndef _TABLE
```

```
#undef EXTERN #define EXTERN #endif
```

y en los archivos `table.c` de cada parte de MINIX hay una línea `#define _TABLE`

antes de la sección `#include`. Así, cuando los archivos de cabecera se incluyen y expanden como parte de la compilación de `table.c`, `extern` no se inserta en ningún lugar (porque `EXTERN` se

define como la cadena nula dentro de table.c) y sólo se reserva memoria para las variables globales en un lugar, en el archivo objeto table.o.

Si usted no tiene experiencia programando en C y no entiende muy bien lo que está sucediendo aquí, no se preocupe; los detalles no son realmente importantes. La inclusión múltiple de archivos de cabecera puede causar problemas a algunos enlazadores porque puede dar lugar a múltiples declaraciones de las variables incluidas. El asunto de EXTERN no es más que una forma de hacer a MINIX más portátil, de modo que pueda enlazarse en máquinas cuyos enlazadores no aceptan variables con múltiples definiciones.

PRIVATE se define como sinónimo de static. Los procedimientos y datos a los que no se hace referencia fuera de los archivos en los que se declaran siempre se declaran como PRIVATE para evitar que sus nombres sean visibles afuera del archivo en el que se declaran. Por regla general, todas las variables y procedimientos deben declararse con alcance local hasta donde sea posible. PUBLIC se define como la cadena nula. Así, la declaración

```
PUBLIC void free_zone(Dev_t dey, zone_t numb)
```

sale del preprocesador de C como

```
void free_zone(Dev_t dey, zone_t numb)
```

que, según las reglas de alcance de C, implica que el nombre free_zone se exporta del archivo y puede usarse en otros archivos. PRIVATE y PUBLIC no son necesarios, pero son intentos por contrarrestar el daño causado por las reglas de alcance de C (la acción por omisión es que los nombres se exportan afuera del archivo; debería ser exactamente lo contrario).

El resto de const.h define constantes numéricas que se usan en todo el sistema. Una sección de const.h está dedicada a las definiciones dependientes de la máquina o de la configuración. Por ejemplo, en todo el código fuente la unidad básica de tamaño de memoria es el click. El tamaño de un click depende de la arquitectura del procesador, y en las líneas 2957 a 2965 se definen alternativas para arquitecturas Intel, Motorola 68000 y Sun SPARC. Este archivo también contiene las macros MAX y MIN que nos permiten escribir

```
z = MAX(x, y);
```

para asignar el mayor de x y y a z.

Type.h (línea 3100) es otro archivo que se incluye en todas las compilaciones por medio de las cabeceras maestras; contiene varias definiciones de tipos clave, junto con valores numéricos relacionados. La definición más importante de este archivo es message en las líneas 3135 a 3146. Aunque podríamos haber definido message como un arreglo con cierto número de bytes, es mejor desde el punto de vista de la práctica de programación hacer que sea una estructura que contiene una unión de los diversos tipos de mensajes posibles. Se definen sE/S formatos de mensaje, de Nzess_1 a mess_6. Un mensaje es una estructura que contiene un campo m_source, para indicar quién envió el mensaje, un campo ,n_ltype, para indicar qué tipo de mensaje es (p. ej., GET_TIME a la tarea del reloj) y los campos de datos. Los sE/S tipos de mensajes se muestran en la Fig. 2-30. En la figura los tipos de mensajes primero y segundo parecen idénticos, iguales que el cuarto y el sexto. Esto es cierto cuando MINIX se implementa en una CPU Intel con tamaño de palabra de 32

bits, pero no sería el caso en una máquina en la que los mt, long y apuntadores tienen diferentes tamaños. La definición de sE/S formatos distintos facilita la recompilación para una arquitectura distinta.

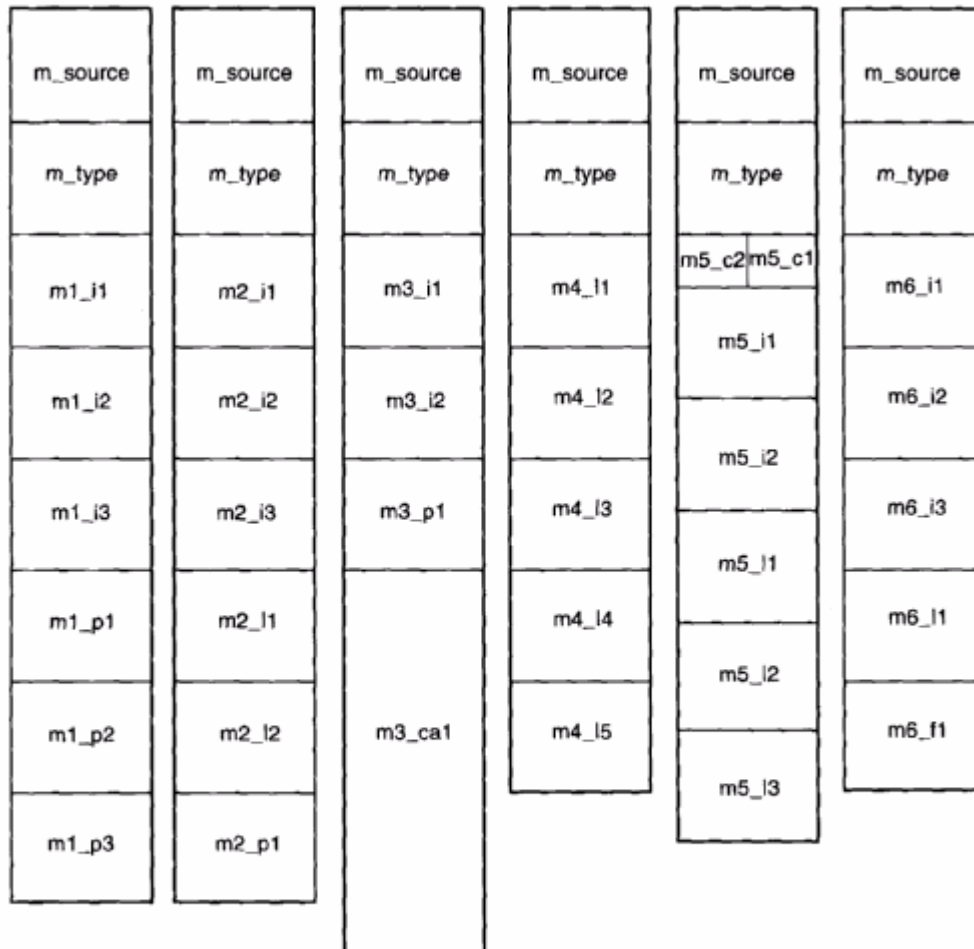


Figura 2-30. Los seis tipos de mensajes empleados en MINIX. Los tamaños de los elementos de los mensajes varían dependiendo de la arquitectura de la máquina; este diagrama ilustra los tamaños en una máquina con apuntadores de 32 bits, como la Pentium (Pro).

Si es necesario enviar un mensaje que contiene, digamos, tres enteros y tres apuntadores (o tres enteros y dos apuntadores), el formato que debe usarse es el primero de la Fig. 2-30. Lo mismo se aplica a los demás formatos. ¿Cómo asignamos un valor al primer entero en el primer formato? Supongamos que el mensaje se llama x. Entonces, x.m_u se refiere a la porción e unión de la estructura message. Si queremos referirnos a la primera de las sE/S alternativas de la unión,

usamos `x.m_u.m_ml`. Por último, para acceder al primer entero de esta struct escribimos `x.m_u.m_ml.mlil`. Esto es un poco torpe, y es por ello que se definen nombres de campos más cortos después de la definición de message misma. Así, podemos usar `x.ml_il` en lugar de `x.m_u.m_ml.mlil`. Todos los nombres cortos tienen la forma de la letra m, el número de formato, un subraya, una o dos letras que indican si el campo es un entero (i), apuntador (p), long (l), carácter (c), arreglo de caracteres (ca) o función (f), y un número de secuencia para distinguir múltiples ejemplares del mismo tipo dentro de un mensaje.

Como acotación, al hablar de los formatos de mensaje es pertinente señalar que un sistema operativo y su compilador a menudo tienen un “acuerdo” en lo tocante a cosas como la organización de las estructuras, y esto puede facilitar las cosas para el implementador. En MINIX los campos `mt` de los mensajes a veces se usan para contener tipos de datos unsigned. En algunos casos esto podría causar un desbordamiento, pero el código se escribió aprovechando el conocimiento de que el compilador de MINIX copia los tipos unsigned en ints y viceversa sin alterar los datos ni generar código para detectar un desbordamiento. Un enfoque más minucioso habría sido sustituir cada uno de los campos `mt` por una unión de un `mt` y un unsigned. Lo mismo aplica a los campos `long` de los mensajes; algunos de ellos pueden usarse para pasar datos unsigned long. ¿Estamos haciendo trampa aquí? Algunos dirían que sí, pero si queremos trasladar MINIX a una nueva plataforma, es evidente que el formato exacto de los mensajes es algo en lo que debemos poner mucha atención, y ahora estamos advertidos de que el comportamiento del compilador es otro factor al que debemos prestar atención.

Hay otro archivo en `include/minix/` que se utiliza universalmente mediante la inclusión en las cabeceras maestras. Se trata de `syslib.h` (línea 3300), que contiene prototipos de funciones de biblioteca de C que se invocan desde el interior del sistema operativo para acceder a otros servicios del sistema operativo. Las bibliotecas de C no se explican con detalle en este texto, pero muchas de ellas son estándar y están disponibles para cualquier compilador de C. Sin embargo, las funciones de C a las que hace referencia `syslib.h` son, por supuesto, específicas para MINIX y si se va a trasladar MINIX a un nuevo sistema con un compilador distinto es necesario trasladar también estas funciones de biblioteca. Por suerte, esto no es difícil, ya que estas funciones simplemente extraen el parámetro de la invocación de la función y los insertan en una estructura de mensaje, luego envían el mensaje y extraen los resultados del mensaje de respuesta. Muchas de estas funciones de biblioteca se definen en una docena de líneas de código en C o menos.

Cuando un proceso necesita ejecutar una llamada al sistema de MINIX, envía un mensaje al administrador de memoria (MM) o el sistema de archivos (FS). Cada mensaje contiene el número de la llamada deseada. Estos números se definen en el siguiente archivo, `callnrh` (línea 3400).

El archivo `com.h` (línea 3500) contiene casi exclusivamente definiciones comunes empleadas en mensajes del MM y el PS a las tareas de E/S. También se definen los números de las tareas. Para distinguirlos de los números de proceso, los números de tarea son negativos. Esta cabecera también define los tipos de mensajes (códigos de función) que se pueden enviar a cada tarea. Por ejemplo, la tarea del reloj acepta los códigos `SET_ALARM` (que sirve para establecer un cronómetro), `CLOCK_TICK` (cuando ha ocurrido una interrupción de reloj), `GET_TIME` (para solicitar el tiempo real) y `SET_TIME` (para establecer la hora del día vigente). El valor `REAL_TIME` es el tipo de mensaje para la respuesta a la solicitud `GET_TIME`.

Por último, `include/minixi` contiene varias cabeceras más especializadas. Entre éstas se encuentran `boot.h` (línea 3700), que usan tanto el kernel como el sistema de archivos para definir dispositivos y acceder a parámetros pasados al sistema por el programa `boot`. Otro ejemplo es `keymap.h` (línea 3800), que define 1 estructuras empleadas para implementar organizaciones de teclado especializadas con los conjuntos de caracteres necesarios para diferentes idiomas. Los programas que generan y cargan estas tablas también lo necesitan. Algunos archivos de este directorio, como `partition.h` (línea 4000) sólo son utilizados por el kernel, y no por el sistema de archivos ni el administrador de memoria. En una implementación con apoyo para dispositivos de E/S adicionales hay más archivos de cabecera como éste, apoyando otros dispositivos. Su colocación en este directorio requiere una explicación. Idealmente, todos los procesos de usuario accederían a los dispositivos sólo a través del sistema operativo, y los archivos como éste se colocarían en `src/kernel/`. Sin embargo, las realidades de la administración de sistemas exigen que haya algunos comandos de usuario que accedan a estructuras en el nivel de sistema, como los comandos que crean particiones de disco. Es para apoyar tales programas de rutinas de utilidad que estos archivos de cabecera especializados se colocan en el árbol de directorios `include/`.

El último directorio de cabeceras especializadas que consideraremos, `include/ibm/`, contiene dos archivos que proporcionan definiciones relacionadas con la familia de computadoras IBM PC. Uno de ellos es `diskparm.h`, utilizado por la tarea del disco flexible. Aunque esta tarea se incluye en la versión estándar de MINIX, su código fuente no se verá con detalle en este texto, ya que es muy similar a la tarea del disco duro. El otro archivo de este directorio es `partition.h` (línea 4100), que define las tablas de partición de disco y las constantes relacionadas que se usan en los sistemas compatibles con IBM. Éstas se colocan aquí para facilitar el traslado de MINIX a otra plataforma de hardware. Si se usa un hardware diferente, tendría que reemplazarse `include/ibm/partition.h`, probablemente por un `partition.h` en algún otro directorio de nombre apropiado, pero la estructura definida en el archivo `include/minixipartition.h` es interna de MINIX y debe permanecer inalterada en un MINIX albergado en una plataforma de hardware distinta.

2.6.4 Estructuras de datos de procesos y archivos de cabecera

Metámonos ahora de lleno en el código de `src/kernel/` y veamos qué aspecto tiene. En las dos secciones anteriores estructuramos nuestra explicación alrededor de un extracto de una cabecera maestra representativa; examinaremos primero la verdadera cabecera maestra del kernel, `kernel.h` (línea 4200). Lo primero que se hace es definir tres macros. La primera, `_POSIX_SOURCE` es una macro de prueba de capacidades definida por el estándar POSIX mismo. Todas las macros de este tipo deben comenzar con el carácter de subraya, `“_”`. El efecto de definir la macro `_POSIX_SOURCE` es asegurar que todos los símbolos requeridos por el estándar y todos los que se permiten explícitamente, pero que no son obligatorios, estén visibles, al tiempo que se ocultan todos los demás símbolos que son extensiones no oficiales de POSIX. Ya mencionamos las dos definiciones que siguen: la macro `_MINIX` somete el efecto de `_POSIX_SOURCE` para extensiones definidas por MINIX, y `__SYSTEM` se puede probar siempre que sea importante hacer algo de forma diferente al compilar código del sistema, en contraposición a código de usuario, como cambiar el

signo de los códigos de error. A continuación, `kernel.h` incluye otros archivos de cabecera de `include/` y sus subdirectorios `include/sys/` e `include/minixi`, incluidos todos aquellos a los que se hace referencia en la Fig. 2-28. Ya hablamos de todos estos archivos en las dos secciones anteriores. Por último, se incluyen cuatro cabeceras más del directorio local, `src/kernel/`.

Éste es un buen lugar para explicar, a quienes no tienen experiencia con el lenguaje C, la forma como se citan los nombres de archivo en una instrucción `#include`. Todo compilador de C tiene un directorio por omisión en el cual busca los archivos de inclusión. Por lo regular, éste es `/usr/include/`, como lo es en un sistema MINIX estándar. Si el nombre del archivo por incluir se encierra en paréntesis angulares ("`< ... >`") el compilador busca el archivo en el directorio de inclusión por omisión o en un subdirectorio especificado de ese directorio. Si el nombre se encierra entre comillas ordinarias ("`...`"), el archivo se busca primero en el directorio actual (o un subdirectorio especificado) y luego, si no se encuentra ahí, en el directorio por omisión.

`Kernel.h` hace posible garantizar que todos los archivos fuente compartan un gran número de definiciones importantes escribiendo la línea

```
#include "kernel.h"
```

en cada uno de los otros archivos fuente del kernel. Puesto que a veces es importante el orden de inclusión de los archivos de cabecera, `kernel.h` también se asegura de que este ordenamiento se efectúe correctamente, de una vez por todas. Esto lleva a un nivel más alto la técnica de "hazlo bien una vez y luego olvídate de los detalles" encarnada en el concepto de archivo de cabecera. Hay cabeceras maestras similares en los directorios fuente del sistema de archivos y del administrador de memoria.

Examinemos ahora los cuatro archivos de cabecera locales incluidos en `kernel.h`. Así como tenemos archivos `const.h` y `type.h` en el directorio de cabeceras comunes `include/minixi`, también tenemos archivos `const.h` y `type.h` en el directorio fuente del kernel, `src/kernel/`. Los archivos de `include/minixi` se colocan ahí porque muchas partes del sistema los necesitan, incluidos programas que se ejecutan bajo el control del sistema. Los archivos de `src/kernel/` proveen definiciones que sólo se necesitan para compilar el kernel. Los directorios fuente del PS y el MM también contienen archivos `const.h` y `type.h` para definir constantes y tipos que sólo se necesitan en esas partes del sistema. Los otros dos archivos incluidos en la cabecera maestra, `proto.h` y `glo.h`, no tienen contrapartes en los directorios `include/principales`, pero habremos de ver que también tienen contra- partes que se usan para compilar el sistema de archivos y el administrador de memoria.

`Const.h` (línea 4300) contiene varios valores dependientes de la máquina, es decir, valores que aplican a los chips de CPU Intel, pero que probablemente son diferentes cuando MIND(se compila en hardware distinto. Estos valores están delimitados por las instrucciones

```
#if(CHIP ==INTEL)
```

```
y
```

```
#endif
```

(líneas 4302 a 4396).

Al compilar MINIX para uno de los chips Intel se definen las macros CF/IP e iNTEL y se hacen iguales en `include/minix/config.h` (línea 2768); con ello se compila el código dependiente de la máquina. Cuando MINIX se trasladó a un sistema basado en el Motorola 68000, la gente que realizó el traslado agregó secciones de código delimitadas por

```
#if (CHIP==M68000)
```

```
y
```

```
#endif
```

e hizo los cambios apropiados en `include/minixlconfig.h` para que fuera efectiva una línea que decía

```
#define CHIP M68000
```

De este modo, MINIX puede manejar constantes y código que son específicos para un sistema. Esta construcción no ayuda mucho que digamos a la comprensibilidad, así que debe usarse lo menos posible. De hecho, en aras de tal comprensibilidad, hemos eliminado muchas secciones de código dependiente de la máquina para el 68000 y otros procesadores de la versión del código impresa en este texto. El código distribuido en el CD-ROM y por Internet conserva el código para otras plataformas.

Unas cuantas de las definiciones de `const.h` merecen mención especial. Algunas de ellas son dependientes de la máquina, como los vectores de interrupción importantes y los valores de campos empleados para restablecer el chip controlador de interrupciones después de cada interrupción. Cada tarea dentro del kernel tiene su propia pila, pero durante el manejo de interrupciones se usa una pila especial de tamaño `K_STACK_BYTES`, definida aquí en la línea 4304. La definición también está en la sección dependiente de la máquina, pues una arquitectura distinta podría requerir más o menos espacio de pila.

Otras definiciones son independientes de la máquina, pero muchas partes del código del kernel las necesitan. Por ejemplo, el planificador de MINIX tiene NQ (3) colas de prioridad, llama `das TASK_Q` (máxima prioridad), `SERVER_Q` (prioridad media) y `USER_Q` (prioridad más baja). Se usan estos nombres para hacer más comprensible el código fuente, pero los valores numéricos definidos por estas macros son lo que en realidad se compila en el código ejecutable. Por último, la última línea de `const.h` define `printf` como una macro cuya evaluación producirá `printk`. Esto permite al kernel exhibir mensajes, como los de error, en la consola usando un procedimiento definido dentro del kernel. Así se pasa por alto el mecanismo usual, que requiere transferir mensajes del kernel al sistema de archivos y luego del sistema de archivos a la tarea de impresora. Durante una falla del sistema esto podría no funcionar. Veremos llamadas a `printf`, alias `printk`, en un procedimiento del kernel llamado `panic` que, como tal vez habrá imaginado, se invoca cuando se detectan errores fatales.

El archivo `type.h` (línea 4500) define varios prototipos y estructuras que se emplean en cualquier implementación de MINIX. La estructura `tasktab` define la estructura de un elemento del arreglo `tasktab` y la estructura `memory` (líneas 4513 a 4516) define las dos cantidades que especifican de manera única un área de la memoria. Éste es un buen lugar para mencionar algunos conceptos

que se usan al referirse a la memoria. Un click es la unidad básica de medición de la memoria; en MiNI para procesadores Intel un click es 256 bytes. La memoria se mide como `phys_clicks`, que el kernel puede usar para acceder a cualquier elemento de memoria en cualquier lugar del sistema, o como `vir_clicks`, utilizados por otros procesos distintos del kernel. Una referencia a memoria `vir_clicks` siempre es relativa a la base de un segmento de memoria asignado a un proceso en particular, y el kernel a menudo tiene que hacer traducciones entre las dos mediciones. La necesidad de hacer esto se compensa con el hecho de que un proceso puede expresar todas sus referencias a la memoria en `vir_clicks`. Podríamos suponer que sería factible usar la misma unidad para especificar el tamaño de ambas clases de memoria, pero el uso de `vir_clicks` para especificar el tamaño de una unidad de memoria asignada a un proceso tiene la ventaja de que en cada ocasión se verifica que no se acceda a ninguna posición de memoria fuera de la que se asignó específicamente al proceso actual. Ésta es una característica importante del modo protegido de los procesadores Intel modernos, como el Pentium y el Pentium Pro. Su ausencia en los primeros procesadores 8086 y 8088 causó algunos dolores de cabeza en el diseño de versiones anteriores de fSENIX.

`Type.h` también contiene varias definiciones de tipos dependientes de la máquina, como `port_t`, `segm_t` y `reg_t` (líneas 4525 a 4527) empleados en los procesadores Intel para direccionar, respectivamente, puertos de E/S, segmentos de memoria y registros de la CPU.

También las estructuras pueden ser dependientes de la máquina. En las líneas 4537 a 4558 se define la estructura `stackframe_s`, que establece la forma en que los registros de la máquina se guardan en la pila, para los procesadores Intel. Esta estructura es extremadamente importante; se usa para guardar y restaurar el estado interno de la CPU cada vez que un proceso pasa al estado “ejecutándose” o deja de estar en ese estado (Fig. 2-2). Al definirla en una forma que se puede leer o escribir eficientemente con código en lenguaje ensamblador se reduce el tiempo requerido para una conmutación de contexto. `Segdesc_s` es otra estructura relacionada con la arquitectura de los procesadores Intel; forma parte del mecanismo de protección que impide a los procesos acceder a regiones de memoria afuera de las que les fueron asignadas.

Con objeto de ilustrar las diferencias entre plataformas, se conservaron en este archivo unas cuantas definiciones para la familia de procesadores Motorola 68000. La familia de procesadores Intel incluye algunos modelos con registros de 16 bits y otros con registros de 32 bits, así que el tipo `reg_t` básico es `unsigned` para la arquitectura Intel. Para los procesadores Motorola `reg_t` se define como el tipo `u32_t`. Estos procesadores también necesitan una estructura `stackframes` (líneas 4583 a 4603), pero la organización es diferente, a fin de hacer que las operaciones en código de ensamblador que la usan sean lo más rápidas posible. La arquitectura Motorola no necesita los tipos `port_t` ni `segm_t`, ni la estructura `segdesc_s`. Hay varias otras estructuras definidas para la arquitectura Motorola que no tienen contrapartes en Intel.

El siguiente archivo, `proto.h` (línea 4700), es el archivo de cabecera más grande que veremos. Este archivo contiene prototipos de todas las funciones que deben conocerse afuera del archivo en el que se definen. Todos se escribieron usando la macro `_PROTOTYPE` que mencionamos antes en esta misma sección, así que el kernel de MINIX se puede compilar ya sea con un compilador de C clásico (Kernighan & Ritchie), como el compilador de C de MINIX original, o un compilador moderno de ANSI Standard C, como el que forma parte de la distribución de

MINIX Versión 2. Varios de estos prototipos dependen del sistema, incluidos los manejadores de interrupciones y excepciones y las funciones escritas en lenguaje ensamblador. No se muestran los prototipos de funciones requeridas por controladores que no se mencionan en este texto. También se ha eliminado el código condicional para procesadores Motorola de éste y los demás archivos que veremos.

La última de las cabeceras del kernel incluida en la cabecera maestra es `glo.h` (línea 5000). Aquí encontramos las variables globales del kernel. El propósito de la macro `EXTERN` se describió cuando hablamos de `include/minix/const.h`. Esta macro normalmente se expande para producir `extern`. Cabe señalar que muchas definiciones de `glo.h` van precedidas por esta macro. Se obliga a que `EXTERN` no esté definida cuando este archivo se incluye en `table.c`, donde se define la macro `_TABLE`. La inclusión de `glo.h` en otros archivos fuente en C hace que las variables de `table.c` sean conocidas por los demás módulos del kernel. `head` y `head_tail` (líneas 5013 y 5014) son apuntadores a una cola de interrupciones pendientes. `Proc_ptr` (línea 5018) apunta a la entrada de la tabla de procesos correspondiente al proceso en curso. Cuando ocurre una llamada al sistema o una interrupción, este apuntador indica dónde deben almacenarse los registros y el estado del procesador. `Sig_procs` (línea 5021) cuenta el número de procesos que tienen señales pendientes que todavía no han sido enviadas al administrador de memoria para ser procesadas. Unos cuantos elementos de `glo.h` se definen con `extern` en lugar de `EXTERN`. Éstos incluyen `sizes`, un arreglo que es llenado por el monitor de arranque, la tabla de tareas, `tasktab`, y la pila de tareas, `t_stack`. Las dos últimas son variables inicializadas, una característica del lenguaje C. El empleo de la macro `EXTERN` no es compatible con la inicialización al estilo C, puesto que una variable sólo puede inicializarse una vez.

Cada tarea tiene su propia pila dentro de `t_stack`. Durante el manejo de interrupciones, el kernel usa una pila aparte, pero ésta no se declara aquí, ya que a ella sólo accede la rutina de nivel de lenguaje ensamblador que se encarga del procesamiento de interrupciones, y no tiene que ser conocida globalmente.

Hay otros dos archivos de cabecera del kernel que se utilizan ampliamente, aunque no tanto que merezcan incluirse en `kernel/`. El primero de éstos es `proc.h` (línea 5100), que define una entrada de tabla de procesos como un `struct proc` (líneas 5110 a 5148). Más adelante en el mismo archivo, se define la tabla de procesos misma como un arreglo de tales structs, `proc + NR_PROCS` (línea 5186). En el lenguaje C está permitida semejante reutilización de un nombre. La macro `NR_TASKS` se define en `include/minix/const.h` (línea 2953) y `NR_PROCS` se define en `include/minix/config.h` (línea 2639). Juntas, éstas dictan el tamaño de la tabla de procesos. `NR_PROCS` puede ser modificada para crear un sistema capaz de manejar un número grande de usuarios. Dado que el acceso a la tabla de procesos es frecuente, y el cálculo de una dirección en un arreglo requiere operaciones de multiplicación lentas, se utiliza un arreglo de apuntadores a los elementos de la tabla de procesos, `pproc_addr` (línea 5187), con objeto de lograr un acceso rápido.

Cada entrada de la tabla tiene espacio para los registros del proceso, el apuntador a la pila, el estado, el mapa de memoria, el límite de la pila, el identificador del proceso, datos de contabilización, tiempo de alarma e información de mensajes. La primera parte de cada entrada de la tabla de procesos es una estructura `stackframe_s`. Un proceso se pone en ejecución cargando su apuntador

a la pila con la dirección de su entrada en la tabla de procesos y sacando todos los registros de CPU de esta estructura.

Cuando un proceso no puede completar un SEND porque el destino no está esperando, el emisor se pone en una cola a la que apunta el campo `p_callerq` del destino (línea 5137). De este modo, cuando el destino finalmente ejecuta un RECEIVE, es fácil encontrar todos los procesos que desean enviarle algo. El campo `p_sendlink` (línea 5138) sirve para enlazar los miembros de la cola.

Cuando un proceso ejecuta RECEIVE y no hay mensajes esperándolo, se bloquea y el número del proceso del cual desea recibir algo se almacena en `p_getfrom`. La dirección del buffer de mensajes se almacena en `p_messbuf`. Los últimos tres campos de cada entrada de la tabla de procesos son `p_nextready`, `p_pending` y `p_pendcount` (líneas 5143 a 5145). El primero de éstos se usa para enlazar los procesos en las colas del planificador, y el segundo es un mapa de bits que sirve para seguir la pista a las señales que todavía no se han pasado al administrador de memoria (porque éste no está esperando un mensaje). El último campo es un contador de esas señales.

Los bits de bandera de `p_flags` definen el estado de cada entrada de la tabla. Si cualquiera de los bits es 1, el proceso no puede ejecutarse. Las diversas banderas se definen y describen en las líneas 5154 a 5160. Si la ranura no está en uso, se pone en 1 `P_SLOT_FREE`. Después de un FORK, se pone en 1 `NO_MAP` para evitar que el proceso hijo se ejecute antes de que se haya preparado su mapa de memoria. `SENDING` y `RECEIVING` indican que el proceso está bloqueado tratando de enviar o recibir un mensaje. `PENDING` y `SIG_PENDING` indican que se han recibido señales, y `P_STOP` sirve como apoyo para el rastreo durante la depuración.

La macro `proc_addr` (línea 5179) se incluye porque no es posible tener subíndices negativos en C. Lógicamente, el arreglo `proc` debería ir de `-NR_TASKS` a `+NR_PROCS`. Desafortunada mente, en C el arreglo debe comenzar en 0, así que `proc` se refiere a la tarea más negativa, y así. Para que sea más fácil calcular cuál ranura corresponde a cuál proceso, podemos escribir

```
rp = proc_addr(n);
```

para asignar a `rp` la dirección de la ranura de proceso correspondiente al proceso `n`, sea positivo o negativo.

`Bill_ptr` (línea 5191) apunta al proceso al que se le está cobrando por la CPU. Cuando un proceso de usuario invoca al sistema de archivos, y éste se está ejecutando, `proc_ptr` (en `glo.h`) apunta al proceso del sistema de archivos. Sin embargo, `bill_ptr` apunta al usuario que realizó la invocación, ya que el tiempo de CPU usado por el sistema de archivos se carga como tiempo de sistema al invocador.

Los dos arreglos `rdy_head` y `rdy_tail` sirven para mantener las colas de planificación. Por ejemplo `rdy_head` apunta al primer proceso de la cola de tareas.

Otra cabecera que se incluye en varios archivos fuente distintos es `protect.h` (línea 5200). Casi todo lo que contiene este archivo tiene que ver con detalles de la arquitectura de los procesadores Intel que manejan el modo protegido (80286, 80386, 80486, Pentium y Pentium Pro). Una descripción detallada de estos chips rebasa el alcance de este libro. Baste decir que

contienen registros internos que apuntan a tablas de descriptores en la memoria. Las tablas de descriptores definen la forma como se usan los recursos del sistema e impiden a los procesos acceder a áreas de memoria asignadas a otros procesos. Además, la arquitectura del procesador contempla cuatro niveles de privilegio, de los cuales MINIX aprovecha tres. Éstos se definen simbólicamente en las líneas 5243 a 5245. Las partes más centrales del kernel, las que se ejecutan durante las interrupciones y conmutan procesos, se ejecutan con `INTR_PRIVILEGE`. No hay ninguna parte de la memoria ni ningún registro de la CPU a los que no pueda acceder un proceso con este nivel de privilegio. Las tareas se ejecutan en el nivel `TASK_PRIVILEGE`, que les permite acceder a E/S pero no usar instrucciones que modifiquen registros especiales, como los que apuntan a las tablas de descriptores. Los servidores y los procesos de usuario se ejecutan en el nivel `USER_PRIVILEGE`. Los procesos que se ejecutan en este nivel no pueden ejecutar ciertas instrucciones, por ejemplo las que acceden a puertos de E/S, modifican asignaciones de memoria o cambian los niveles de privilegio mismos. El concepto de nivel de privilegio debe ser conocido para quienes están familiarizados con la arquitectura de las CPU modernas, pero quienes aprendieron arquitectura de computadoras estudiando el lenguaje de ensamblador de los procesadores menos potentes tal vez no se hayan topado con tales restricciones.

Hay varios otros archivos de cabecera en el directorio del kernel, pero sólo mencionaremos otros dos aquí. Primero está `sconst.h` (línea 5400), que contiene constantes utilizadas por el código de ensamblador. Todas éstas son desplazamientos dentro de la porción de estructura `stackframe_s` de una entrada de la tabla de procesos, expresados en una forma susceptible de ser utilizada por el ensamblador. Puesto que el código de ensamblador no es procesado por el compilador de C, es más sencillo tener tales definiciones en un archivo aparte. Además, puesto que todas estas definiciones dependen de la máquina, aislarlas aquí simplifica el proceso de trasladar MINIX a otro procesador que necesitará una versión distinta de `sconst.h`. Cabe señalar que muchos desplazamientos se expresan como el valor anterior más `W`, que se hace igual al tamaño de palabra en la línea 5401. Esto permite usar el mismo archivo para compilar una versión de MINIX de 16 o de 32 bits.

Aquí hay un problema potencial. Se supone que los archivos de cabecera nos permiten proveer un solo correo correcto de definiciones y luego utilizarlas en muchos lugares sin tener que volver a prestar mucha atención a los detalles. Obviamente, las definiciones duplicadas, como las de `sconst.h`, violan ese principio. Desde luego, se trata de un caso especial, pero, como tal, requiere atención especial si se llega a modificar ya sea este archivo o `proc.h`, a fin de asegurar que los dos archivos sean consistentes.

La última cabecera que mencionaremos aquí es `assert.h` (línea 5500). El estándar POSIX obliga a contar con una función `assert`, que puede servir para realizar una prueba en el momento de la ejecución y abortar un programa, exhibiendo un mensaje. De hecho, Posix exige que se proporcione una cabecera `assert.h` en el directorio `include/`, y se proporciona una allí. Entonces, ¿por qué hay otra versión aquí? La razón es que cuando algo sale mal en un proceso de usuario, se puede confiar en que el sistema operativo proporcionará servicios tales como exhibir un mensaje en la consola. Pero si algo falla en el kernel mismo, no puede contarse con los recursos normales del sistema. Por ello, el kernel provee sus propias rutinas para manejar `assert` y exhibir mensajes, con independencia de las versiones de la biblioteca normal del sistema.

Hay unos cuantos archivos de cabecera en kernel/ que no hemos visto todavía. Éstos apoyan las tareas de E/S y se describirán en el siguiente capítulo donde sean pertinentes. Antes de pasar al código ejecutable, empero, examinemos table.c (línea 5600) cuyo archivo objeto compilado contendrá todas las estructuras de datos del kernel. Ya hemos visto la definición de muchas de estas estructuras de datos, en glo.h y proc.h. En la línea 5625 se define la macro `_TABLE`, inmediatamente antes de las instrucciones `#include`. Como ya se explicó, esta definición hace que `EXTERN` quede definida como la cadena nula, y que se asigne espacio de almacenamiento a todas las declaraciones de datos precedidas por `EXTERN`. Además de las estructuras de glo.h y proc.h, también se asigna aquí memoria para unas cuantas variables globales utilizadas por la tarea de terminal, definidas en tty.h.

Además de las variables declaradas en archivos de cabecera, hay otros dos lugares donde se asigna memoria para datos globales. Algunas definiciones se realizan directamente en table.c. En las líneas 5639 a 5674 se asigna espacio de pila a cada tarea. Para cada tarea opcional, se usa la macro `ENABLE_XXX` correspondiente (definida en el archivo `include/minix./confi g. h`) para calcular el tamaño de la pila. Así, no se asigna espacio a una tarea que no está habilitada. Después de esto, se usan las diversas macros `ENABLE_XXX` para determinar si cada tarea opcional se representará o no en el arreglo `tasktab`, compuesto por estructuras `tasktab`, según se declaró antes en `src/ kernellzype.h` (líneas 5699 a 5731). Hay un elemento para cada proceso que se pone en marcha durante la inicialización del sistema, sea tarea, servidor o proceso de usuario (o sea, `mit`). El índice del arreglo establece implícitamente una correspondencia entre los números de tarea y los procedimientos de arranque asociados. `Tasktab` también especifica el espacio de pila requerido para cada proceso y provee una cadena de identificación para cada uno. `Tasktab` se colocó aquí y no en un archivo de cabecera porque el truco con `EXTERN` que se utilizó para evitar múltiples declaraciones no funciona con las variables inicializadas; es decir, no podemos escribir

```
extern int x = 3;
```

en ningún lado. Las definiciones anteriores de tamaño de pila también permiten el reparto de espacio de pila a todas las tareas en la línea 5734.

A pesar del intento por aislar toda la información de configuración ajustable por el usuario en `include/minixlconflg.h`, puede cometerse un error al igualar el tamaño del arreglo `tasktab` con `NR_TASKS`. Al final de `table.c` se realiza una prueba para detectar este error, usando un pequeño truco. Se declara aquí el arreglo ficticio `dummy_tasktab` de modo tal que su tamaño sea imposible y dé pie a un error de compilador si se cometió un error. Puesto que el arreglo ficticio se declara `extern`, no se le asigna espacio aquí (ni en ningún otro lado). Puesto que en ningún lugar del código se hace referencia a este arreglo, el compilador no objetará.

El otro lugar donde se asigna almacenamiento global es al final del archivo en lenguaje ensamblador `mpx386.s` (línea 6483). Esta asignación, en la etiqueta `_sizes`, coloca un número mágico (para identificar un kernel de MINIX válido) justo al principio del segmento de datos del kernel. Se asigna espacio adicional aquí con la seudoinstrucción `.space`. Al reservar memoria de esta manera, el programa en lenguaje ensamblador obliga a colocar físicamente el arreglo `_sizes` al principio del segmento de datos del kernel, y esto facilita la programación de boot de modo tal que coloque los datos en el lugar correcto. El monitor de arranque lee el número mágico y, si es

correcto, lo sobrescribe para inicializar el arreglo `_sizes` con los tamaños de las diferentes partes del sistema MINIX. El kernel utiliza estos datos durante la inicialización. En el momento del arranque, el kernel considera ésta como un área de datos inicializada. Sin embargo, los datos que el kernel finalmente encuentra aquí no están disponibles en el momento de la compilación. Estos datos son introducidos por el monitor de arranque justo antes de iniciarse el kernel. Todo esto es un poco inusual, pues normalmente no es necesario escribir programas que conozcan la estructura interna de otros programas. Sin embargo, el periodo después de que se enciende la máquina pero antes de que el sistema operativo esté funcionando es definitivamente inusual, y requiere técnicas poco usuales.

2.6.5 Autoarranque de MINIX

Ya casi es el momento de comenzar a examinar el código ejecutable, pero antes de hacerlo es conveniente dedicar un poco de tiempo a entender la forma como MINIX se carga en la memoria. Desde luego, MINIX se carga de un disco. La Fig. 2-31 muestra la organización de los disquetes y de los discos duros con particiones.

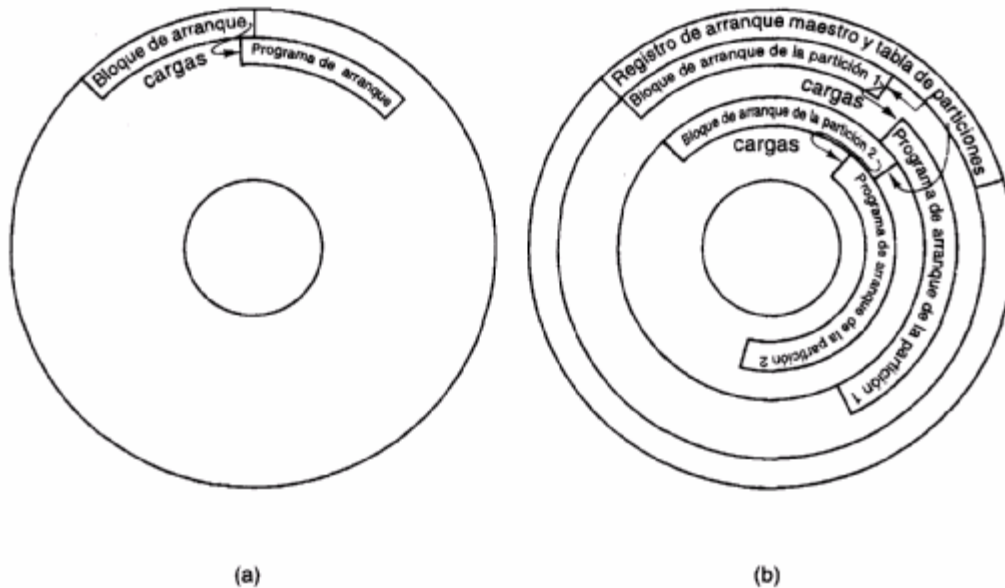


Figura 2-31. Estructuras de disco utilizadas para autoarranque. (a) Disco sin particiones. El primer sector es el bloque de arranque. (b) Disco con particiones. El primer sector es el registro de arranque maestro.

Cuando se inicia el sistema, el hardware (en realidad, un programa en ROM) lee el primer sector del disco de arranque y ejecuta el código que encuentra ahí. En un disquete de MINIX sin particiones, el primer sector es un bloque de arranque que carga el programa de arranque (boot), como se aprecia en la Hg. 2-31(a). Los discos duros tienen particiones, y el programa que está en el primer sector lee la tabla de particiones, que también está en el primer sector, y carga y ejecuta el primer sector de la partición activa, como se muestra en la Fig. 2-31(b). (Normalmente sólo una partición está marcada como activa). Una partición MINIX tiene la misma estructura que un disquete MINIX sin particiones, con un bloque de arranque que carga el programa de arranque.

La situación real puede ser un poco más complicada que lo que sugiere la figura, porque una partición puede contener subparticiones. En tal caso el primer sector de la partición es otro registro de arranque maestro que contiene la tabla de particiones para las subparticiones. No obstante, tarde o temprano se transferirá el control a un sector de arranque, el primer sector de un dispositivo que no tiene más subdivisiones. En un disquete el primer sector siempre es un sector de arranque. MINIX permite una forma de subdivisión de un disquete, pero sólo la primera partición puede ser iniciada; no existe un registro de arranque maestro aparte, y no puede haber subparticiones. Esto permite montar los disquetes con y sin particiones exactamente de la misma manera. El uso principal de un disquete con particiones es dividir un disco de instalación en una imagen raíz que se copiará en un disco en RAM y una porción montada que podrá desmontarse cuando ya no se necesite, liberando así la unidad de disquete para continuar con el proceso de instalación.

El sector de arranque de MINIX se modifica en el momento en que se escribe en el disco introduciendo los números de sector necesarios para encontrar un programa llamado boot en su partición o subpartición. Esta introducción es necesaria porque antes de que se cargue el sistema operativo no es posible usar el directorio y nombres de archivo para encontrar un archivo. Se utiliza un programa especial llamado installboot para realizar la introducción y la escritura del sector de arranque. Boot es el cargador secundario para MINIX, pero puede hacer más que simplemente cargar el sistema operativo, ya que es un programa monitor que permite al usuario modificar, establecer y guardar diversos parámetros. Boot examina el segundo sector de su partición en busca de un conjunto de parámetros que usará. MINIX, al igual que el UNIX estándar, reserva el primer bloque de 1K de todos los dispositivos de disco como bloque de arranque, pero el cargador de arranque en ROM o el sector maestro de arranque sólo carga un sector de 512 bytes, así que hay 512 bytes disponibles para guardar ajustes. Éstos controlan la operación de arranque, y también se pasan al sistema operativo mismo. Los ajustes por omisión presentan un menú con una sola opción, iniciar MINIX, pero es posible modificarlos de modo que presenten un menú más complejo que permita la iniciación de otros sistemas operativos (cargando y ejecutando sectores de arranque de otras particiones), o iniciar MINIX con diversas opciones. También podemos modificar los ajustes por omisión de modo que pasen por alto el menú e inicien MINIX de inmediato.

Boot no forma parte del sistema operativo, pero tiene la suficiente inteligencia como para utilizar las estructuras de datos del sistema de archivos para encontrar la imagen del sistema operativo real. Por omisión, boot busca un archivo llamado `¡mmix` o, si existe un directorio `/minix/`, el archivo más reciente dentro de ese directorio, pero es posible modificar los parámetros de arranque de modo que busquen un archivo con cualquier nombre. Este grado de flexibilidad es poco usual; la mayor parte de los sistemas operativos tienen un nombre de archivo predefinido

para la imagen del sistema. Por otro lado, MINIX es un sistema operativo poco usual que estimula a los usuarios para que lo modifiquen y creen versiones experimentales nuevas. La prudencia exige que los usuarios que hagan esto cuenten con un mecanismo para seleccionar múltiples versiones, y así poder regresar a la última versión que funcionó correctamente si es que llega a fallar un experimento.

La imagen de MINIX cargada por boot no es más que una concatenación de los archivos individuales producidos por el compilador cuando compila los programas del kernel, el administrador de memoria, el sistema de archivos e `mit`. Cada uno de éstos incluye una cabecera corta del tipo de las que se definen en `include/a.out.h`, y, a partir de la información contenida en la cabecera de cada parte, boot determina cuánto espacio debe reservar para datos no inicializados después de cargar el código ejecutable y los datos inicializados de cada parte, con objeto de que la siguiente parte pueda cargarse en la dirección correcta. El arreglo `_sizes` que mencionamos en la sección anterior también recibe una copia de esta información para que el kernel mismo pueda tener acceso a la ubicación y el tamaño de todos los módulos cargados por boot. Las regiones de la memoria disponibles para cargar el sector de arranque, boot mismo y MINIX dependen del hardware. Además, algunas arquitecturas de máquina pueden requerir un ajuste de las direcciones internas dentro del código ejecutable a fin de convertirlas en las direcciones reales donde se cargan los programas. La arquitectura segmentada de los procesadores Intel hace que esto sea innecesario. Puesto que los detalles del proceso de carga difieren según el tipo de máquina, y boot en sí no forma parte del sistema operativo, no lo describiremos con mayor detalle aquí. Lo importante es que, de una forma u otra, el sistema operativo se carga en la memoria. Una vez cargado, el control pasa al código ejecutable del kernel.

Como acotación, cabe señalar que los sistemas operativos no siempre se cargan de discos locales. Las estaciones de trabajo sin disco pueden cargar su sistema operativo de un disco remoto, a través de una conexión de red. Desde luego, esto requiere software de red en ROM. Aunque los detalles no son idénticos a los que hemos descrito aquí, lo más probable es que los elementos d proceso sean similares. El código en ROM debe tener la suficiente inteligencia como para ob a través de la red un archivo ejecutable que se encargará de obtener el sistema operativo completo. Si MINIX se cargara de esta forma, no sería necesario hacer muchos cambios al proceso de inicialización que ocurre una vez que el código del sistema operativo está cargado en la memoria. Desde luego, se requeriría un servidor de red y un sistema de archivos modificado capaz de acceder a los archivos a través de la red.

2.6.6 Inicialización del sistema

MINIX para máquinas tipo IBM PC se puede compilar en modo de 16 bits si se requiere compatibilidad con los procesadores Intel más antiguos, o en modo de 32 bits si se desea un mejor rendimiento en los procesadores 80386+. Se usa el mismo código fuente en C y el compilador genera la salida apropiada dependiendo de si es la versión para 16 bits o 32 bits del compilador. Una macro definida por el compilador mismo determina la definición de la macro `_WORD_SIZE` que está en `includeminixlconfig.h`. La primera parte de MINIX que se ejecuta está escrita en lenguaje ensamblador, y se deben usar diferentes archivos de código fuente para el compilador de 16 bits

y el de 32 bits. La versión del código de inicialización para 32 bits está en `mpx386.s`. La alternativa, para los sistemas de 16 bits, está en `mpx88.s`. Ambos incluyen apoyo en lenguaje ensamblador para otras operaciones del kernel de bajo nivel. La selección se efectúa automáticamente en `mpx.s`.

Este archivo es tan corto que podemos presentarlo completo en la Fig. 2-32.

```
#include <minix/config.h>
#if _WORD_SIZE == 2
#include "mpx88.s"
#else
#include "mpx386.s"
#endif
```

Figura 2-32. Forma como se seleccionan los archivos fuente en lenguaje ensamblador alternativos.

`Mpx.s` muestra un uso poco común de la instrucción `#include` del preprocesador de C. Por lo regular, `#include` sirve para incluir archivos de cabecera, pero también puede utilizarse para seleccionar una sección alterna de código fuente. Si quisiéramos hacer esto usando instrucciones `#if`, tendríamos que colocar todo el código de los archivos `mpx88.s` y `mpx386.s`, que es muy extenso, en un solo archivo. Esto no sólo sería difícil de manejar; también desperdiciaría espacio de disco, pues en una instalación dada lo más probable es que uno de los dos archivos no se utilice en absoluto, y puede archivarse o eliminarse. En la siguiente explicación usaremos el `mpx386.s` de 32 bits como ejemplo.

Puesto que ésta es la primera vez que examinamos código ejecutable, comenzaremos con unas pocas palabras acerca de cómo haremos esto a lo largo del libro. Los múltiples archivos fuente que se usan para compilar un programa grande en C pueden ser difíciles de seguir. En general, limitaremos las explicaciones a un solo archivo a la vez, y veremos los archivos en orden. Comenzaremos con el punto de entrada de cada parte del sistema MINIX, y seguiremos la línea de ejecución principal. Cuando nos topemos con una llamada a una función de apoyo, diremos unas cuantas palabras acerca del propósito de la llamada, pero normalmente no describiremos con detalle el funcionamiento interno en ese punto; dejaremos eso para cuando lleguemos a la definición de la función invocada. Las funciones subordinadas importantes por lo regular se definen en el mismo archivo en el que se invocan, después de las funciones invocadoras de nivel superior; pero las funciones pequeñas o de propósito general a veces se reúnen en archivos aparte. Además, se ha intentado colocar el código dependiente de la máquina y el independiente de la máquina en archivos distintos, a fin de facilitar el traslado a otras plataformas. Se ha hecho un esfuerzo considerable por organizar el código; de hecho, muchos archivos se reescribieron durante la redacción de este texto a fin de hacer su organización más comprensible para el lector. Sin embargo, un programa grande tiene muchas ramificaciones, y hay ocasiones en que para entender una función principal es necesario leer las funciones que invoca. Por ello, a veces es útil tener unas cuantas tiras de papel que puedan servir de marcadores en diversas partes del libro y desviarse del orden de las explicaciones para ver las cosas en un orden distinto.

Habiendo planteado la forma en que pensamos organizar nuestra explicación del código, debemos comenzar por justificar de inmediato una excepción importante. El inicio de MINIX

implica varias transferencias de control entre las rutinas en lenguaje ensamblador de `mpx386.s` y rutinas escritas en C contenidas en los archivos `start.c` y `main.c`. Describiremos estas rutinas en el orden en que se ejecutan, aunque para ello sea necesario saltar de un archivo a otro.

Una vez que el proceso de autoarranque ha cargado el sistema operativo en la memoria, se transfiere el control a la etiqueta MINIX (en `mpx386.s`, línea 6051). La primera instrucción hace que se pasen por alto unos cuantos bytes de datos; éstos incluyen las banderas del monitor de arranque (línea 6054) que el monitor usa para identificar diversas características del kernel, siendo la más importante de ellas si se trata de un sistema de 16 o de 32 bits. El monitor de arranque siempre inicia en modo de 16 bits, pero conmuta la CPU a modo de 32 bits si es necesario. Esto sucede antes de que el control pase a MINIX. El monitor también prepara una pila. El código en lenguaje ensamblador tiene una buena cantidad de trabajo que realizar aquí: preparar un soporte de pila que ofrezca un entorno apropiado para el código compilado por el compilador de C, copiar tablas empleadas por el procesador para definir los segmentos de memoria, y configurar varios registros del procesador. Una vez terminado este trabajo, el proceso de inicialización continúa con la invocación (en la línea 6109) de la función `cstart`, que está en C. Observe que el código en lenguaje ensamblador se refiere a ella como `_cstart`. La razón es que a los nombres de todas las funciones compiladas por el compilador de C se les antepone un carácter de subraya en las tablas de símbolos, y el enlazador busca esos nombres cuando enlaza módulos que se compilaban por separado. Puesto que el ensamblador no agrega tales caracteres, el escritor de un programa en lenguaje ensamblador debe agregar uno explícitamente para que el enlazador pueda encontrar un non-ibne correspondiente en el archivo objeto compilado por el compilador de C. `Cstart` invoca otra rutina para inicializar la tabla de descriptores globales, la estructura de datos central empleada por los procesadores Intel de 32 bits para supervisar la protección de memoria, y la tabla de descriptores de interrupciones, que sirve para seleccionar el código que se ejecutará al ocurrir cada tipo de interrupción. Al regresar de `cstart`, las instrucciones `Igdt` y `Iidt` (líneas 6115 y 6116) hacen efectivas estas tablas cargando los registros dedicados mediante los cuales se direccionan. La instrucción

```
jmpf CS_SELECTOR:csinit
```

parece a primera vista una operación nula, ya que transfiere el control al mismo lugar al que se transferiría si en lugar de ella hubiera una serie de instrucciones `nop`. No obstante, ésta es una parte importante del proceso de inicialización. Este salto obliga a usar las estructuras recién inicializadas. Después de manipular otro poco los registros del procesador, MINIX termina con un salto (no una llamada) en la línea 6131 al punto de entrada `main` del kernel (en `main.c`). En este punto ha terminado el código de inicialización de `mpx386.s`. El resto del archivo contiene código para iniciar o reiniciar una tarea o proceso, manejadores de interrupciones y otras rutinas de apoyo que tuvieron que escribirse en lenguaje de ensamblador para que fueran eficientes. Volveremos a ellas en la siguiente sección.

Examinemos ahora las funciones de inicialización en C de nivel más alto. La estrategia general es hacer lo más que se pueda usando código en C de alto nivel. Ya hay dos versiones del código `mpx`, como hemos visto, y todo lo que pueda descargarse a código en C elimina dos trozos de código ensamblador. Casi lo primero que hace `cstart` (en `start.c`, línea 6524) es configurar los mecanismos

de protección de la CPU y las tablas de interrupciones, mediante una invocación a `prat_mit`. Luego, `cstart` copia los parámetros de arranque en la parte de la memoria que corresponde al kernel y los convierte en valores numéricos; además, determina el tipo de pantalla de video, el tamaño de la memoria, el tipo de máquina, el modo de operación del procesador (real o protegido) y si es posible o no regresar al monitor de arranque. Toda la información se almacena en variables globales apropiadas, a fin de que cualquier parte del código del kernel pueda acceder a ella si es necesario.

`Main` (en `main.c`, línea 6721) completa la inicialización y luego inicia la ejecución normal del sistema. Esta función configura el hardware de control de interrupciones invocando a `intr_init`. Esto se hace aquí porque no puede hacerse antes de conocer el tipo de la máquina, y el procedimiento está en un archivo aparte porque depende mucho del hardware. El parámetro (1) en la llamada le dice a `intr_init` que está inicializando para MINIX. Se puede llamar con un parámetro (0) para minicializar el hardware a su estado original. La llamada a `intr_init` realiza también dos pasos para asegurar que cualquier interrupción que ocurra antes de haberse completado la inicialización no tendrá ningún efecto. Primero se escribe en cada uno de los chips controladores de interrupciones un byte que inhibe la respuesta a las entradas externas. Luego, todas las entradas de la tabla empleada para acceder a los manejadores de interrupciones específicos para cada dispositivo se llenan con la dirección de una rutina que, sin perjudicar la inicialización, exhibirá un mensaje si se recibe una interrupción espuria. Más adelante estas entradas se reemplazarán, una por una, por apuntadores a las rutinas manejadoras, conforme cada una de las tareas de E/S ejecute su propia rutina de inicialización. Entonces, cada tarea restablecerá un bit en el chip controlador de interrupciones a fin de habilitar su propia entrada de interrupciones.

A continuación se invoca `me,n_init`, que inicializa un arreglo que define la ubicación y el tamaño de cada región de memoria disponible en el sistema. Al igual que en la inicialización del hardware de interrupciones, los detalles dependen del hardware, y el aislamiento de `mem_init` como función en un archivo aparte mantiene a `main` libre de código que no pueda trasladarse a un hardware distinto.

La parte más grande del código de `main` se dedica a la preparación de la tabla de procesos, de modo que cuando se planifiquen las primeras tareas y procesos, sus mapas de memoria y registros se establezcan correctamente. Todas las ranuras de la tabla de procesos se marcan como desocupadas, y el ciclo de las líneas 6745 a 6749 inicializa el arreglo `pproc_addr` que agiliza el acceso a la tabla de procesos. El código de la línea 6748,

```
(pproc_addr + NR_TASKS) = rp;
```

podría haberse definido también como `pproc_addr + NR_TASKS] = rp;`

porque en el lenguaje C a es otra forma de escribir `*(a+i)`. Por tanto, casi da lo mismo si sumamos una constante a a o a i. Algunos compiladores de C generan código que es un poco mejor si se suma una constante al arreglo en lugar de al índice.

La parte más grande de `main`, el ciclo largo de las líneas 6762 a 6815, inicializa la tabla de procesos con la información necesaria para ejecutar las tareas, los servidores e `mit`. Todos estos procesos deben estar presentes en el momento del inicio y ninguno de ellos terminará durante el

funcionamiento normal. Al principio del ciclo, se asigna a `rp` la dirección de una entrada de la tabla de procesos (línea 6763). Puesto que `rp` es un apuntador a una estructura, se puede acceder a los elementos de dicha estructura usando una flotación como `rp—>p_name`, como se hace en la línea 6765. Esta notación se usa mucho en el código fuente de MINIX.

Por supuesto, todas las tareas se compilan en el mismo archivo que el kernel, y la información referente a sus necesidades de pila está en el arreglo `tasktab` definido en `table.c`. Puesto que las tareas se compilan junto con el kernel y pueden invocar código y acceder a datos situados en cualquier parte del espacio del kernel, el tamaño de una tarea individual no es significativo, y el campo de tamaño para cada una de ellas se llena con el tamaño del kernel mismo. El arreglo `sizes` contiene los tamaños en clicks del texto y los datos del kernel, el administrador de memoria, el sistema de archivos e `mit`. Esta información fue introducida en el área de datos del kernel por `boot` antes de que el kernel iniciara su ejecución, y desde el punto de vista del kernel es como si el compilador la hubiera proporcionado. Los dos primeros elementos de `sizes` son los tamaños del texto y los datos del kernel; los siguientes dos son los del administrador de memoria, etc. Si alguno de los cuatro programas no usa espacio `I` y `D` separado, el tamaño del texto es `O` y el texto y los datos se agrupan como datos. Al asignarse a `sizeindex` un valor de cero (línea 6775) para cada una de las tareas se asegura que se accederá al elemento número `O` de `sizes` para todas las tareas en las líneas 6783 y 6784. La asignación a `sizeindex` de la línea 6778 suministra a cada uno de los servidores y a `mit` su propio índice para acceder a `sizes`.

El diseño de la IBM PC original colocaba la memoria sólo de lectura en la parte superior del intervalo de memoria utilizable, que está limitado a 1 MB en una CPU 8088. Las máquinas modernas compatibles con PC siempre tienen más memoria que la PC original, pero por compatibilidad todavía tienen la memoria sólo de lectura en las mismas direcciones que las máquinas antiguas. Así, la memoria de leer-escribir es discontinua, con un bloque de ROM entre los 640 KB inferiores y el intervalo por encima de 1 MB. Si es posible, el monitor de arranque carga los servidores e `mit` en el intervalo de memoria situado por arriba de la ROM. Esto se hace principalmente pensando en el sistema de archivos, a fin de que pueda usar un caché de bloques muy grande sin toparse con la memoria sólo de lectura. El código condicional de las líneas 6804 a 6810 asegura que este uso del área de memoria alta quede registrado en la tabla de procesos.

Dos entradas de la tabla de procesos corresponden a procesos que no necesitan planificarse de forma ordinaria. Estos procesos son `IDLE` y `HARDWARE`. `IDLE` es un ciclo que no hace nada y que se ejecuta cuando ningún otro proceso está listo para ejecutarse. `HARDWARE` existe para propósitos de contabilización; a él se acredita el tiempo usado mientras se atiende una interrupción. El código de la línea 6811 coloca todos los demás procesos en las colas apropiadas. La función que se invoca, `lock_ready`, establece una variable de candado, `switching`, antes de modificar las colas, y luego quita el candado una vez que la cola se ha modificado. El empleo de candados no es necesario en este momento en el que nada se está ejecutando todavía, pero éste es el método estándar, y no tiene caso crear código extra que se usará sólo una vez.

El último paso para inicializar cada ranura de la tabla de procesos es una invocación a `alloc_segments`. Este procedimiento forma parte de la tarea del sistema, pero como todavía no se está ejecutando ninguna tarea se invoca como procedimiento ordinario en la línea 6814. Se trata

de una rutina dependiente de la máquina que coloca en los campos apropiados las ubicaciones, tamaños y niveles de permiso para los segmentos de memoria utilizados por cada proceso. En el caso de los procesadores Intel más antiguos que no manejan el modo protegido, esta rutina sólo define la ubicación de los segmentos, y tendría que reescribirse para manejar un tipo de procesador con un método de reparto de memoria distinto.

Una vez inicializada la tabla de procesos para todas las tareas, los servidores e `mit`, el sistema ya casi está listo para ponerse a trabajar. La variable `bill_ptr` indica a cuál proceso se le cobrará el tiempo de procesador; es necesario asignarle un valor inicial en la línea 6818, e `IDLE` es una buena opción. Más adelante podría ser modificada por la siguiente función invocada, `lock_pick_proc`. Todas las tareas están listas ya para ejecutarse, y `bill_ptr` se modificará cuando se ejecute un proceso de usuario. La otra obligación de `lock_pick_proc` es hacer que la variable `prnc_ptr` apunte a la entrada de la tabla de procesos correspondiente al siguiente proceso por ejecutar. Esta selección se efectúa examinando las colas de tareas, servidores y procesos de usuario, en ese orden. En este caso, el resultado es que `proc_ptr` apunta al punto de entrada de la tarea de la consola, que siempre es la primera en iniciarse.

Por fin ha terminado `main`. En muchos programas en C `main` es un ciclo, pero en el kernel de MINIX su trabajo está hecho una vez que se completa la inicialización. La llamada a `restart` de la línea 6822 inicia la primera tarea. El control nunca volverá a `main`.

`_Restart` es una rutina en lenguaje ensamblador que está en `mpx386.s`. De hecho, `_restart` no es una función completa; es un punto de entrada intermedio de un procedimiento mayor. Examinaremos esta rutina con detalle en la siguiente sección; por ahora sólo diremos que `_restart` causa la conmutación de contexto, así que se ejecutará el proceso al que apunta `proc_ptr`. Una vez que `_restart` se ha ejecutado por vez primera ya podemos decir que MINIX está funcionando: está ejecutando un proceso. `_Restart` se ejecuta una y otra vez conforme se da oportunidad a las tareas, servidores y procesos de usuario de ejecutarse, para luego ser suspendidos, ya sea en espera de entradas o para ceder el turno a otros procesos.

La tarea que se pone en cola primero (la que usa la ranura `O` de la tabla de procesos, es decir, la que tiene el número más negativo) siempre es la tarea de la consola, a fin de que otras tareas puedan usarla para informar de su avance o de los problemas que enfrenten una vez que se inicien. La tarea de la consola se ejecuta hasta que se bloquea tratando de recibir un mensaje. A continuación se ejecutará la siguiente tarea hasta que ella también se bloquee tratando de recibir un mensaje. Tarde o temprano todas las tareas estarán bloqueadas, y el administrador de memoria y el sistema de archivos podrán ejecutarse. Al ejecutarse por primera vez, cada uno de ellos realiza algunas operaciones de inicialización, pero llegará el momento en que ambos queden bloqueados también. Por último, `mit` bifurcará un proceso `getty` para cada terminal. Estos procesos se bloquearán hasta que se teclee alguna entrada en alguna terminal, y en ese momento el primer usuario podrá iniciar una sesión.

Ya hemos seguido el inicio de MINIX a través de tres archivos, dos escritos en C y uno en lenguaje ensamblador. El archivo en lenguaje ensamblador, `mpx386.s`, contiene código adicional que se usa para manejar interrupciones, cosa que veremos en la siguiente sección. Sin embargo, antes de continuar describiremos brevemente las rutinas restantes de los dos archivos en C. Los otros procedimientos de `start.c` son `k_atoi` (línea 6594) que convierte una cadena en un entero, y

`k_getenv` (línea 6606), que sirve para encontrar entradas en el entorno del kernel, que es una copia de los parámetros de arranque. Ambos son versiones simplificadas de funciones de biblioteca estándar que se reescriben aquí con objeto de reducir la complejidad del kernel. El único procedimiento que quedaba sin mención en `main.c` es `panic` (línea 6829), que se invoca cuando el sistema descubre una condición que le impide continuar. Las condiciones de pánico típicas son la imposibilidad de leer un bloque de disco crítico, la detección de un estado interno inconsistente, o que una parte del sistema invoque a otra con parámetros no válidos. Las llamadas a `print` aquí son en realidad llamadas a la rutina del kernel `printk`, a fin de que el kernel pueda escribir en la consola aun cuando la comunicación entre procesos normal haya quedado interrumpida.

2.6.7 Manejo de interrupciones en MINIX

Los detalles del hardware de interrupciones dependen del sistema, pero todo sistema debe contar con elementos funcionales equivalentes a los que describiremos para sistemas con una CPU Intel de 32 bits. Las interrupciones generadas por dispositivos de hardware son señales eléctricas y al principio son manejadas por un controlador de interrupciones: un circuito integrado que puede detectar varias de esas señales y generar para cada una, una disposición de datos única en el bus de datos del procesador. Esto es necesario porque el procesador en sí sólo tiene una entrada para detectar todos estos dispositivos y, por tanto, es incapaz de distinguir cuál de ellos está solicitando servicio. Las PC que usan procesadores Intel de 32 bits normalmente vienen equipadas con dos de estos chips controladores. Cada uno de ellos puede manejar ocho entradas, pero uno es un esclavo que alimenta su salida a una de las entradas del maestro, de modo que la combinación puede detectar quince dispositivos externos distintos, como se aprecia en la Fig. 2-33.

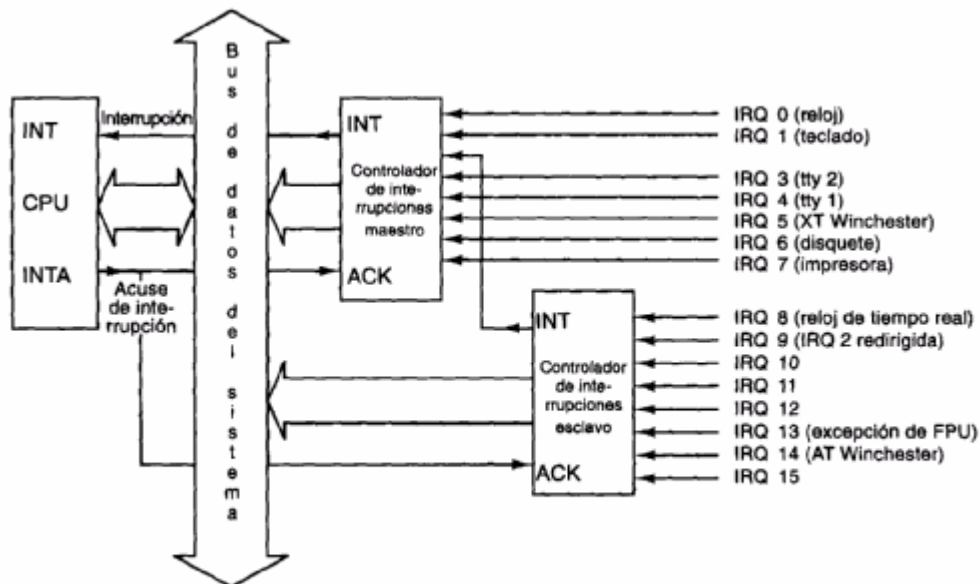


Figura 2-33. Hardware de procesamiento de interrupciones en una PC Intel de 32 bits.

En la figura, las señales de interrupción llegan por las distintas líneas JRQ *n* indicadas a la derecha. La conexión con el pin INT de la CPU le dice al procesador que ha ocurrido una interrupción. La señal INTA (acuse de interrupción) de la CPU hace que el controlador responsable de la interrupción coloque datos en el bus de datos del sistema para indicarle al procesador cuál rutina de servicio debe ejecutar. Los chips controladores de interrupciones se programan durante la inicialización del sistema, cuando main invoca a intr_init. La programación determina la salida que se envía a la CPU para cada señal recibida por cada una de las líneas de entrada, así como varios otros parámetros del funcionamiento del controlador. Los datos que se colocan en el bus son un número de 8 bits que se usa como índice de una tabla de hasta 256 elementos. La tabla de MINIX tiene 56 elementos, de los cuales sólo se usan realmente 35; los demás están reservados para utilizarse con procesadores Intel futuros o para mejoras futuras de MINIX. En los procesadores Intel de 32 bits esta tabla contiene descriptores de puertas de interrupción, cada uno de los cuales es una estructura de 8 bytes con varios campos.

Hay varios modos posibles de responder a las interrupciones; en el que se usa con MINIX, los campos que más nos interesan de los descriptores de puertas de interrupción apuntan al segmento de código ejecutable de la rutina de servicio y a la dirección inicial dentro de él. La CPU ejecuta el código al que apunta el descriptor seleccionado. El resultado es exactamente el mismo que si se ejecutara una instrucción de lenguaje ensamblador

```
int    <nnn>
```

La única diferencia es que en el caso de una interrupción de hardware la parte <nnn> se origina en un registro del chip controlador de interrupciones, y no en una instrucción que está en la memoria del programa.

El mecanismo de conmutación de tareas que entra en acción como respuesta a una interrupción en un procesador Intel de 32 bits es complejo, y la modificación del contador de programa para ejecutar otra función es sólo una parte. Cuando la CPU recibe una interrupción durante la ejecución de un proceso, establece una nueva pila para usarla mientras atiende la interrupción. La ubicación de esta pila está determinada por una entrada del segmento de estado de tareas (TSS). Ésta es una estructura única para todo el sistema, que se inicializa cuando cstart invoca prot_init y se modifica cada vez que se inicia un proceso. El efecto es que la nueva pila creada por una interrupción siempre comienza al final de la estructura stackframe_s dentro de la entrada correspondiente al proceso interrumpido en la tabla de procesos. La CPU agrega automáticamente varios registros clave en esta nueva pila, incluidos los necesarios para reinstaurar la pila del proceso interrumpido y restablecer su contador de programa. Una vez que el código del manejador de interrupciones empieza a ejecutarse, usa esta área de la tabla de procesos como su pila, pero gran parte de la información necesaria para regresar al proceso interrumpido ya se habrá almacenado. El manejador de interrupciones agrega en la pila el contenido de registros adicionales, llenando el marco de pila, y luego pasa a una pila provista por el kernel mientras hace lo necesario para atender la interrupción.

La terminación de una rutina de servicio de interrupción se efectúa pasando de la pila del kernel a un marco de pila en la tabla de procesos (pero no necesariamente el mismo que creó la última interrupción), removiendo explícitamente los registros adicionales, y ejecutando una

instrucción `iretd` (retorno de interrupción). `Iretd` restaura el estado que existía antes de una interrupción, restaurando los registros agregados por el hardware y conmutando de vuelta a una pila que se estaba usando antes de una interrupción. Así, una interrupción detiene un proceso, y la finalización del servicio de la interrupción reinicia un proceso, que podría ser distinto del que se detuvo más recientemente. A diferencia de los mecanismos de interrupción más sencillos que son el tema usual de los textos sobre programación en lenguaje de ensamblador, nada se almacena en la pila de trabajo del proceso interrumpido durante una interrupción. Además, dado que la pila se crea de nuevo en una posición conocida (determinada por el TSS) después de una interrupción, se simplifica el control de múltiples procesos. Si se desea iniciar un proceso distinto lo único que se necesita es hacer que el apuntador de pila apunte al marco de pila de otro proceso, restaurar los registros que se agregaron explícitamente, y ejecutar una instrucción `iretd`.

La CPU inhabilita todas las interrupciones cuando recibe una interrupción. Esto garantiza que nada podrá ocurrir que haga que el marco de pila dentro de una entrada de la tabla de procesos se desborde. Esto es automático, pero también existen instrucciones en el nivel de ensamblador para inhabilitar y habilitar las interrupciones. El manejador de interrupciones vuelve a habilitar las interrupciones después de pasar a la pila del kernel, situada fuera de la tabla de procesos. Desde luego, el manejador debe inhabilitar otra vez todas las interrupciones antes de pasar de nuevo a una pila dentro de la tabla de procesos, pero mientras está manejando una interrupción pueden ocurrir otras interrupciones y ser procesadas. La CPU sigue la pista a las interrupciones anidadas, y se vale de un método más sencillo de pasar a una rutina de servicio de interrupción y regresar de una cuando se interrumpe un manejador de interrupciones. Si se recibe una interrupción nueva mientras se está ejecutando un manejador (u otro código del kernel), no se crea una nueva pila. En vez de ello, la CPU agrega a la pila existente los registros esenciales necesarios para reanudar el código interrumpido. Cuando se encuentra un `iretd` durante la ejecución de código del kernel, se usa también un mecanismo de retorno más simple. El procesador puede determinar cómo debe manejar el `iretd` examinando el selector de segmento de código que se remueve de la pila como parte de la acción del `iretd`.

Los niveles de privilegio que mencionamos antes controlan las diferentes respuestas a las interrupciones que se reciben mientras se está ejecutando un proceso y mientras se está ejecutando código del kernel (incluidas las rutinas de servicio de interrupción). Se usa el mecanismo más sencillo cuando el nivel de privilegio del código interrumpido es el mismo que el del código que se ejecutará como respuesta a la interrupción. Sólo cuando el código interrumpido es menos privilegiado que el código de servicio de interrupción se usa el mecanismo más complicado, utilizando el TSS y una nueva pila. El nivel de privilegio de un segmento de código se registra en el selector de segmentos de código, y como ésta es una de las cosas que se agregan durante una interrupción, se puede examinar después de regresar de la interrupción para determinar lo que debe hacer la instrucción `iretd`. El hardware proporciona otro servicio cuando se crea una nueva pila para usarla mientras se atiende una interrupción. El hardware verifica que la nueva pila tenga el tamaño suficiente para guardar cuando menos la cantidad mínima de información que debe colocarse en ella. Esto protege al código de kernel más privilegiado contra una falla accidental (o premeditada) causada por un proceso de usuario que realiza una llamada al sistema con una pila

insuficiente. Estos mecanismos se incorporan en el procesador específicamente para utilizarse en la implementación de sistemas operativos que manejan múltiples procesos.

Este comportamiento puede causar confusión si no se conoce el funcionamiento interno de las CPU Intel de 32 bits. Normalmente tratamos de evitar la descripción de tales detalles, pero es indispensable entender qué sucede cuando ocurre una interrupción y cuando se ejecuta una instrucción `iretd` para poder entender cómo el kernel controla las transiciones de y hacia el estado “ejecutándose” de la Fig. 2-2. El hecho de que el hardware se encargue de gran parte del trabajo facilita mucho las cosas para el programador, y es de suponer que hace al sistema resultante más eficiente. Toda esta ayuda del hardware, empero, dificulta el entendimiento de lo que está sucediendo si sólo leemos el software.

Sólo una pequeñísima parte del kernel de MINIX ve realmente las interrupciones de hardware. Este código está en `mpx386.s`. Hay un punto de entrada para cada interrupción. El código fuente en cada punto de entrada, `hwint00` a `_hwint07` (líneas 6164 a 6193), parece una llamada a `hwint_master` (línea 6143), y los puntos de entrada `_hwint08` a `_hwint15` (líneas 6222 a 6251) parecen llamadas a `hwint_slave` (línea 6199). Cada punto de entrada aparentemente pasa un parámetro en la llamada, indicando cuál dispositivo requiere servicio. En realidad, éstas no son realmente llamadas, sino macros, y se ensamblan ocho copias individuales del código definido por la definición de macro de `hwint_master` que sólo difieren en el parámetro `irq`. De forma similar, se ensamblan ocho copias de la macro `hwint_slave`. Esto puede parecer extravagante, pero el código ensamblado es muy compacto. El código objeto para cada macro expandida ocupa menos de 40 bytes. Al atender una interrupción, la rapidez es crucial, y cuando se efectúa de este modo se elimina el gasto extra de ejecutar código para cargar un parámetro, invocar una subrutina y recuperar el parámetro.

Continuaremos la explicación de `hwint_master` como si en realidad fuera una sola función, y no una macro que se expande en ocho lugares distintos. Recuerde que antes de que `hwint_master` inicie su ejecución, la CPU ha creado una nueva pila en el `stackframe_s` del proceso interrumpido, dentro de su ranura de la tabla de procesos, y que ya se guardaron ahí varios registros clave. La primera acción de `hwint_master` es invocar `save` (línea 6144). Esta subrutina agrega todos los demás registros necesarios para reiniciar el proceso interrumpido. Se podría haber escrito `save` en línea como parte de la macro a fin de aumentar la rapidez, pero esto habría aumentado a más del doble el tamaño de la macro, y además hay otras funciones que invocan `save`. Como veremos, `save` hace malabarismos con la pila. Al regresar a `hwint_master`, se está usando la pila del kernel, no un marco de pila de la tabla de procesos. El siguiente paso consiste en manipular el controlador de interrupciones, a fin de evitar que reciba otra interrupción del dispositivo que generó la interrupción actual (líneas 6145 a 6147). Esta operación enmascara la capacidad del chip controlador de responder a una entrada específica; la capacidad de la CPU para responder a todas las interrupciones se inhibe internamente desde el momento en que recibe la señal de interrupción, y a estas alturas todavía no ha sido restablecida.

El código de las líneas 6148 a 6150 restablece el controlador de interrupciones y luego habilita la CPU para recibir otra vez interrupciones de otros dispositivos. A continuación, la instrucción `call` indirecta de la línea 6152 usa el número de la interrupción que está siendo atendida como

índice para acceder a una tabla de direcciones de las rutinas de bajo nivel específicas para cada dispositivo. Decimos que estas rutinas son de bajo nivel, pero están escritas en C, y por lo regular realizan operaciones tales como dar servicio a un dispositivo de entrada y transferir los datos a un buffer donde estará accesible para la tarea correspondiente cuando ésta tenga su siguiente oportunidad de ejecutarse. Puede haber una cantidad considerable de procesamiento antes de regresar de esta llamada.

Veremos ejemplos de código controlador de bajo nivel en el siguiente capítulo. No obstante, a fin de entender lo que está sucediendo aquí en `hwint_master`, mencionaremos que el código de bajo nivel puede invocar `interrupt` (`enproc.c`, que veremos en la siguiente sección), y que `interrupt` transforma la interrupción en un mensaje dirigido a la tarea que da servicio al dispositivo que causó la interrupción. Además, una llamada a `interrupt` invoca al planificador, el cual puede seleccionar esta tarea para ser ejecutada a continuación. Al regresar de la llamada al código específico para el dispositivo, se vuelve a inhabilitar la capacidad del procesador para responder a todas las interrupciones, mediante la instrucción `cli` de la línea 6154, y el controlador de interrupciones queda preparado para poder responder al dispositivo específico que causó la interrupción en curso cuando se vuelvan a habilitar todas las interrupciones (líneas 6157 a 6159). Luego, `hwint_master` termina con una instrucción `ret` (línea 6160). No es obvio que aquí esté ocurriendo algo engañoso. Si lo que se interrumpió fue un proceso, la pila que se está usando en este punto es la del kernel, no la pila dentro de la tabla de procesos que el hardware configuró antes de que se iniciara `hwint_master`. En este caso, la manipulación de la pila por `save` habrá dejado la dirección de `_restart` en la pila del kernel. Esto hará que se ejecute otra vez una tarea, servidor o proceso de usuario que podría no ser (y de hecho es poco probable que sea) el mismo proceso que se estaba ejecutando originalmente. Esto depende de si el procesamiento del mensaje creado por la rutina de servicio de interrupción específica para el dispositivo causó o no un cambio en las colas de planificación de procesos. Éste es, pues, el corazón del mecanismo que crea la ilusión de múltiples procesos que se ejecutan simultáneamente.

Para no dejar cabos sueltos, mencionemos que cuando ocurre una interrupción mientras se está ejecutando código del kernel la pila del kernel ya está en uso, y `save` deja la dirección de `restart` en la pila del kernel. En este caso, lo que sea que el kernel haya estado haciendo previamente continuará después del `ret` al final de `hwint_master`. Por tanto, las interrupciones pueden anidarse, pero una vez que han terminado todas las rutinas de servicio de bajo nivel se ejecuta finalmente `_restart`, y puede ser que se ponga en ejecución un proceso distinto del que fue interrumpido.

`Hwinr_slave` (línea 6199) es muy similar a `hwint_master`, excepto que debe volver a habilitar ambos controladores maestro y esclavo, ya que ambos quedan inhabilitados cuando el esclavo recibe una interrupción. Aquí hay unos cuantos aspectos sutiles del lenguaje ensamblador que debemos examinar. Primero, en la línea 6206 vemos

```
jmp .+2
```

que especifica un salto cuya dirección objetivo es la instrucción que le sigue inmediatamente. Esta instrucción se coloca aquí únicamente para agregar un pequeño retardo. Los autores del IBM PC BIOS original consideraron que era necesario un retardo entre instrucciones de E/S consecutivas, y estamos siguiendo su ejemplo, aunque tal vez no sea necesario en todas las computadoras

compatibles con IBM PC modernas. Esta clase de afinación es una de las razones por las que a18lmos consideran la programación de dispositivos de hardware como un arte esotérico. En la línea 6214 hay un salto condicional a una instrucción con un rótulo numérico,

```
O:    ret
```

que se encuentra en la línea 6128. Observe que la línea

```
jz    Of
```

no especifica el número de bytes que deben saltarse, como en el ejemplo anterior. El Of aquí no es un número hexadecimal. Ésta es la forma en que el ensamblador utilizado por el compilador de MINIX especifica un rótulo local; Of significa saltar hacia adelante (“forward”) al siguiente rótulo numérico O. Los nombres de rótulos ordinarios no pueden comenzar con caracteres numéricos. Otro punto interesante y tal vez confuso es que el mismo rótulo ocurre en otro lugar del mismo archivo, en la línea 6160 de hwint_master. La situación es aún más complicada de lo que parece a primera vista, ya que estos rótulos están dentro de macros y las macros se expanden antes de que el ensamblador vea este código. Por tanto, en realidad hay 16 rótulos O: en el código que el código ensamblador ve. La posible proliferación de rótulos declarados dentro de macros es, de hecho, la razón por la que el lenguaje ensamblador proporciona rótulos locales; para resolver un rótulo local el ensamblador usa el más cercano que coincide en la dirección especificada, y hace caso omiso de las demás ocurrencias del rótulo local.

Examinemos ahora save (línea 6261), al que ya hemos mencionado varias veces. Su nombre describe una de sus funciones, que es guardar el contexto del proceso interrumpido en la pila provista por la CPU. Esta pila es un marco de pila dentro de la tabla de procesos. Save usa la variable `_k_reenter` para contar y determinar el nivel de anidación de las interrupciones. Si se estaba ejecutando un proceso cuando ocurrió la interrupción actual, la instrucción

```
mov   esp, k_stktop
```

de la línea 6274 pasa a la pila del kernel, y la siguiente instrucción agrega la dirección de `_restart` (línea 6275). Si no se estaba ejecutando ningún proceso, ya se está usando la pila del kernel, y lo que se agrega es la dirección de `restart` (línea 6281). En ambos casos, una instrucción `return` ordinaria no es suficiente para regresar al invocador, pues cabe la posibilidad de que se esté usando una pila distinta de la que estaba vigente al ingresar, y de que la dirección de retorno de la rutina invocadora esté enterrada bajo los registros que acaban de agregarse. Las instrucciones

```
jmp   RETADR-P_STACKBASE(eax)
```

que dan término a los dos puntos de salida de `save`, en las líneas 6277 y 6282 respectivamente, utilizan la dirección que se agregó cuando se invocó `save`.

El siguiente procedimiento de `mpx386.s` es `_s_call`, que comienza en la línea 6288. Antes de examinar sus detalles internos, vea cómo termina. No hay ningún `ret` ni `jmp` al final. Después de inhabilitarse las interrupciones con `cli` en la línea 6315, la ejecución continúa con `_restart`. `_S_call` es la contraparte de llamada al sistema del mecanismo de manejo de interrupciones. El control llega a `_s_call` después de una interrupción de software, es decir, una ejecución de una instrucción

int nnn. Las interrupciones de software se tratan como interrupciones de hardware, excepto desde luego que el índice para acceder a la tabla de descriptores de interrupciones está codificado en la parte nnn de la instrucción `mt nnn`, en lugar de ser proporcionado por un chip controlador de interrupciones. Así, cuando se ingresa en `_s_call`, la CPU ya se pasó a una pila dentro de la tabla de procesos (suministrada por el Segmento de Estado de Tareas) y ya se han agregado varios registros en esta pila. Al continuar con `_restart`, la llamada a `_s_call` en última instancia termina con una instrucción `iretd` y, al igual que en el caso de una interrupción de hardware, esta instrucción iniciará el proceso al que apunte `proc_ptr` en ese momento. En la Fig. 2-34 se compara el manejo de una interrupción de hardware y una llamada al sistema empleando el mecanismo de interrupción de software.

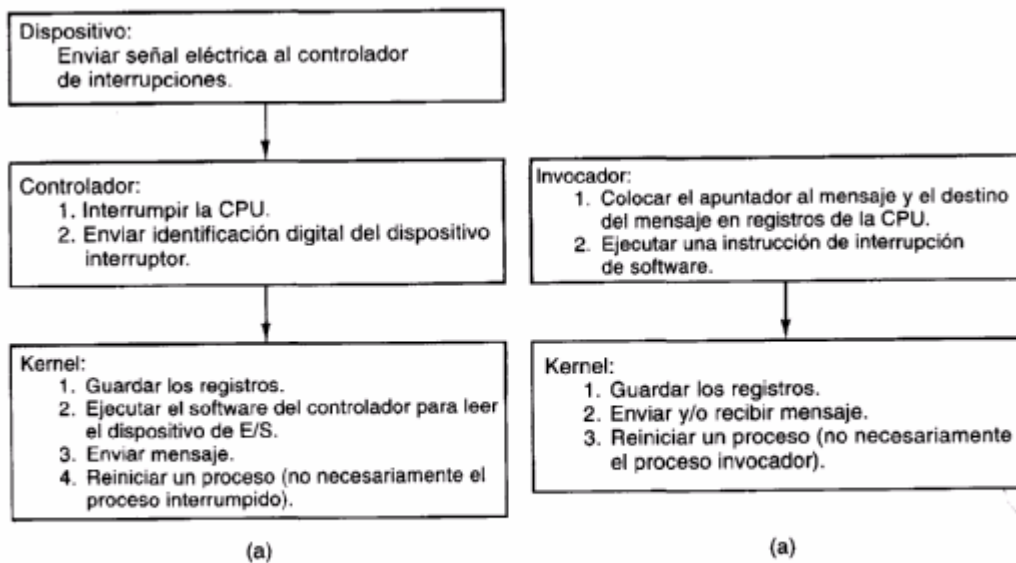


Figura 2-34. (a) Procesamiento de una interrupción de hardware. (b) Ejecución de una llamada al sistema.

Veamos ahora algunos de los detalles de `_s_call`. La etiqueta alterna, `_p_s_call`, es un vestigio de la versión de MINIX para 16 bits, que tiene rutinas individuales para operar en modo protegido y en modo real. En la versión para 32 bits todas las llamadas a cualquiera de esas etiquetas viene a dar aquí. Un programador que invoca una llamada al sistema de MINIX escribe una llamada de función en C que se ve como cualquier otra llamada de función, sea a una función definida local mente o a una rutina de la biblioteca de C. El código de biblioteca que apoya la llamada al sistema prepara un mensaje, carga la dirección del mensaje y el identificador de proceso del destino en registros de la CPU, y luego invoca una instrucción `mt SYS386_VECTOR`. Como se explicó antes, el resultado es que el control pasa al inicio de `_s_call`, y varios registros ya se han agregado en una pila dentro de la tabla de procesos.

La primera parte del código de `_s_call` se parece a una expansión en línea de `save` y guarda registros adicionales que deben preservarse. Al igual que en `save`, una instrucción

```
Mov esp, kstktop
```

Pasa entonces a la pila del kernel, y se vuelven a habilitar las interrupciones. (La similitud entre interrupción de software y una de hardware se extiende a que ambas inhabilitan todas las interrupciones.) Después de esto viene una llamada a `_sys_call`, que veremos en la siguiente i4e. Por ahora sólo diremos que esta función hace que se entregue un mensaje, y que esto a su vez hace que se ejecute el planificador. Por tanto, cuando `_sys_call` regresa, es probable que `proc...ptr` esté apuntando a un proceso distinto del que inició la llamada al sistema. Antes de que la ejecución continúe con `_restart`, una instrucción `cli` inhabilita las interrupciones a fin de proteger el marco de pila del proceso que está a punto de ser reiniciado.

Hemos visto que puede llegarse a `_restart` (línea 6322) de varias maneras:

1. Por una llamada desde `main` cuando se inicia el sistema.
2. Por un salto desde `hwint_naster` o `hwint_slave` después de una interrupción de hardware.
3. Continuando al final de `_s_call` después de una llamada al sistema.

En todos los casos las interrupciones se inhabilitan en este punto. `_Restart` llama a `unhoid` si detecta que se ha detenido alguna interrupción no atendida porque llegó mientras se estaban procesando otras interrupciones. Esto permite convertir las otras interrupciones en mensajes antes de que se reinicie cualquier proceso. Las interrupciones quedan habilitadas otra vez temporalmente, pero se inhabilitan de nuevo antes de que `unhoid` regrese. A la altura de la línea 6333 ya se ha escogido definitivamente el siguiente proceso que se ejecutará, y con las interrupciones inhabilitadas la decisión no puede cambiarse. La tabla de procesos se construyó cuidadosamente de modo que comenzara con un soporte de pila, y la instrucción de esta línea,

```
mov esp, (_proc_ptr)
```

hace que el registro apuntador a la pila de la CPU apunte al marco de pila. A continuación, la instrucción

```
lldt P_LDT_SEL(esp)
```

carga el registro de la tabla de descriptores local del procesador con un valor tomado del marco de pila. Esto prepara al procesador para usar los segmentos de memoria que pertenecen al siguiente proceso por ejecutar. La siguiente instrucción pone la dirección contenida en la entrada correspondiente al siguiente proceso en la tabla de procesos para que indique dónde se establecerá la pila para la siguiente interrupción, y la siguiente instrucción almacena dicha dirección en el TSS. La primera parte de `_restart` no es necesaria después de una interrupción que ocurre cuando se está ejecutando código del kernel (incluido código de servicio de interrupción), puesto que se estará usando la pila del kernel y la terminación del servicio de interrupción deberá permitir que el código de kernel continúe. La etiqueta `restarti` (línea 6337) marca el punto donde se reanuda la ejecución en este caso. En este momento se decrementa `k_reenter` a fin de registrar que se ha dado

cuenta de un nivel de interrupciones posiblemente anidadas, y el resto de las instrucciones restablecen el procesador al estado en el que estaba cuando el siguiente proceso se ejecutó por última vez. La penúltima instrucción modifica el apuntador a la pila de modo que se pase por alto la dirección de retorno que se agregó cuando se invocó `save`. Si la última interrupción ocurrió cuando se estaba ejecutando un proceso, la instrucción final, `iretd`, completa el retorno a la ejecución del proceso al que ahora se le va a permitir ejecutarse, restaurando sus registros restantes, incluido su segmento de pila y apuntador a la pila. Por otro lado, si este encuentro con `iretd` sucedió por la vía de `iretd`, la pila del kernel en uso no es un marco de pila sino la pila del kernel, y no se trata de un retorno a un proceso interrumpido, sino la finalización de una interrupción que ocurrió mientras se estaba ejecutando código del kernel. La CPU detecta esto cuando el descriptor del segmento de código se saca de la pila durante la ejecución del `iretd`, y la acción completa del `iretd` en este caso es mantener en uso la pila del kernel.

Hay unas cuantas cosas más que comentar en `mpx386.s`. Además de las interrupciones de hardware y software, diversas condiciones de error internas de la CPU pueden causar la iniciación de una excepción. Las excepciones no siempre son malas; pueden servir para estimular al sistema operativo para que proporcione un servicio, como suministrar más memoria a un proceso, o intercambiar hacia adentro una página de memoria que se intercambió hacia afuera, aunque tales servicios no se implementan en el MINIX estándar. Sin embargo, cuando ocurre una excepción, no debe hacerse caso omiso de ella. El mismo mecanismo que maneja las interrupciones maneja las excepciones, empleando descriptores de la tabla de descriptores de interrupciones. Estas entradas de la tabla apuntan a los 16 puntos de entrada de manejador de excepciones, comenzando con `_divide_error` y terminando con `_copr_error`, que se encuentra cerca del final de `mpx386.s`, en las líneas 6350 a 6412. Todos éstos saltan a `exception` (línea 6420) o a `errexception` (línea 6431), dependiendo de si la condición agrega a la pila un código de error o no. El manejo aquí en el código de ensamblador es similar al que ya vimos: se agregan registros y se invoca la rutina en C `_exception` (tome nota del carácter de subraya) para manejar el evento. Las consecuencias de las excepciones varían. Algunas se ignoran, otras causan pánicos, y otras más causan el envío de señales a procesos. Examinaremos `_exception` en una sección posterior.

Existe un punto de entrada más que se maneja como una interrupción, `levelO_cali` (línea 6458). Explicaremos su función en la siguiente sección, cuando veamos el código al que salta, `_levelO_func`. El punto de entrada está aquí en `mpx360.s` junto con los puntos de entrada de interrupciones y excepciones porque también se invoca mediante la ejecución de una instrucción `/nt`. Al igual que las rutinas de excepción, esta rutina invoca `save`, así que tarde o temprano el código al que se salta aquí terminará con un `ret` que conduce a `_restart`. La última función ejecutable de `mpx386.s` es `_idle_task` (línea 6465). Éste es un ciclo que no hace nada y que se ejecuta cuando no hay ningún otro proceso listo para ejecutarse.

Por último, se reserva un poco de espacio para almacenar datos al final del archivo en lenguaje ensamblador. Aquí se definen dos segmentos de datos distintos. La declaración

```
.sect .rom
```

de la línea 6478 asegura que este espacio se asignará exactamente al principio del segmento de datos del kernel. El compilador coloca un número mágico aquí para que boot pueda verificar que

el archivo que carga es una imagen de kernel válida. A continuación boot sobrescribe el número mágico y el espacio subsecuente con los datos del arreglo `_sizes`, como se explicó al hablar de las estructuras de datos del kernel. Se reserva suficiente espacio aquí para un arreglo `_sizes` con un 16 entradas, en caso de que se agreguen servidores adicionales a MINIX. La otra área de nacimiento de datos definida en la declaración

```
.sect .bss
```

(línea 6483) reserva espacio en el área de variables no inicializadas normal del kernel para la pila del kernel y para variables utilizadas por los manejadores de excepciones. Se reserva espacio de pila para los servidores y procesos ordinarios cuando se enlaza un archivo ejecutable, y éstos dependen del kernel para que ajuste correctamente el descriptor de segmento de pila y el apuntador a la pila cuando se ejecutan. El kernel tiene que hacer esto por sí mismo.

2.6.8 Comunicación entre procesos en MINIX

En MINIX los procesos se comunican con mensajes, usando el principio de cita. Cuando un proceso ejecuta un SEND, la capa más baja del kernel verifica si el destino está esperando un mensaje del emisor (o de cualquier (ANY) emisor). Si así es, el mensaje se copia del buffer del emisor al buffer del receptor, y ambos procesos se marcan como ejecutables. Si el destino no está esperando un mensaje del emisor, éste se marca como bloqueado y se coloca en una cola de procesos que esperan para enviar al receptor.

Cuando un proceso ejecuta un RECEIVE, el kernel verifica si hay algún proceso en cola tratando de enviar un mensaje al primero. Si así es, el mensaje se copia del emisor bloqueado al receptor, y ambos se marcan como ejecutables. Si no hay ningún proceso en cola tratando de enviarle un mensaje, el receptor se bloquea hasta que llega un mensaje.

El código de alto nivel para la comunicación entre procesos se encuentra en `proc.c`. Al kernel corresponde traducir una interrupción de hardware o bien de software en un mensaje. La primera es generada por hardware y la segunda es la forma en que se comunica al kernel una solicitud de servicios del sistema, es decir, una llamada al sistema. Estos casos son lo bastante similares como para que se pudieran haber manejado con una sola función, pero resultó más eficiente crear dos funciones especializadas.

Primero examinaremos `interrupt` (línea 6983), la cual es invocada por la rutina de servicio de interrupción de bajo nivel para un dispositivo después de recibirse una interrupción de hardware. La función de `interrupt` consiste en convertir la interrupción en un mensaje para la tarea que maneja el dispositivo interruptor, y por lo regular se efectúa muy poco procesamiento antes de llamar a `interrupt`. Por ejemplo, el manejador de interrupciones de bajo nivel completo para el controlador del disco duro consiste sólo en estas tres líneas:

```
w_STATUS = in_byte(w_wn—>base + REG_STATUS); / acuse de interrupción /
interrupt(WINCH ESTER);
return 1;
```

Si no fuera necesario leer un puerto de E/S en el controlador del disco duro para obtener el estado, la llamada a `interrupt` podría haber estado en `mpx386.x` y no en `at_wini.c`. Lo primero que hace

`interrupt` es verificar si ya se estaba atendiendo una interrupción cuando se recibió la interrupción actual; esto se hace examinando la variable `k_reenter` (línea 6962). En este caso la interrupción actual se pone en cola e `interrupt` regresa. La interrupción actual será atendida después, cuando se invoque `unhoid`. La siguiente acción consiste en verificar si la tarea está o no esperando una interrupción (líneas 6978 a 6981). Si la tarea no está lista para recibir, se pone en 1 su bandera `p_int_blocked`—veremos más adelante que esto permite recuperar la interrupción perdida— y no se envía ningún mensaje. Si se pasa esta prueba, se envía el mensaje. El envío de un mensaje de `HARDWARE` a una tarea es sencillo, porque las tareas y el kernel se compilan en el mismo archivo y pueden acceder a las mismas áreas de datos. El código de las líneas 6989 a 6992 envía el mensaje llenando los campos de origen y tipo del buffer de mensaje de la tarea de destino, poniendo en 0 la bandera `RECEIVING` del destino y desbloqueando la tarea. Una vez que el mensaje está listo, se planifica la tarea de destino para que se ejecute. Estudiaremos la planificación con mayor detalle en la siguiente sección, pero el código de `interrupt` entre las líneas 6997 a 7003 nos da una idea de lo que veremos: éste es un sustituto en línea del procedimiento `ready` que se invoca para poner en cola un proceso. El código aquí es sencillo, ya que los mensajes que se originan en interrupciones sólo se envían a tareas, y por tanto no hay necesidad de determinar cuál de las tres colas de procesos debe modificarse.

La siguiente función de `proc.c` es `sys_cail`, y tiene una función similar a la de `interrupt`: convierte una interrupción de software (la instrucción `mt SYS386_VECTOR` con la que se inicia una llamada al sistema) en un mensaje. Sin embargo, como en este caso hay una variedad más amplia de posibles orígenes y destinos, y dado que la llamada puede requerir ya sea el envío o la recepción, o tanto el envío como la recepción, de un mensaje, `_sys_call` tiene que efectuar más trabajo. Como suele suceder, esto implica que el código de `sys_cail` es corto y sencillo, ya que efectúa casi todo su trabajo invocando otros procedimientos. La primera de estas llamadas es a `isoksrc_dest`, una macro definida en `proc.h` (línea 5172) que incorpora una macro más, `isokprocn`, también definida en `proc.h` (línea 5171). El efecto es que se verifica si el proceso especificado como origen o destino del mensaje es válido. En la línea 7026 se realiza una prueba similar, `isuserp` (otra macro definida en `proc.h`), para asegurarse de que si la llamada proviene de un proceso de usuario, éste está pidiendo enviar un mensaje y luego recibir una respuesta, ya que ésta es la única clase de llamada que pueden efectuar los procesos de usuario. Tales errores son poco probables, pero es fácil realizar las pruebas, ya que en última instancia se compilan para producir código que realiza comparaciones con enteros pequeños. En este nivel más básico del sistema operativo es aconsejable tratar de detectar incluso los errores más inverosímiles. Es probable que este código se ejecute muchas veces durante cada segundo que está activo el sistema de computación en el que se ejecuta.

Por último, si la llamada requiere enviar un mensaje, se invoca `mini_send` (línea 7031), y si es necesario recibir un mensaje, se invoca `mini_rec` (línea 7039). Estas funciones son el corazón del mecanismo normal de transferencia de mensajes de `MINIX` y conviene estudiarlos con detenimiento.

`Mini_send` (línea 7045) tiene tres parámetros, el invocador, el proceso al que se enviará el mensaje y un apuntador al buffer donde está el mensaje. Esta función realiza varias pruebas. Primero, se asegura de que los procesos de usuario sólo traten de enviar mensajes al `FS` o al `MM`.

En la línea 7060 se prueba el parámetro `caller_ptr` con la macro `isuserp` para determinar si el invocador es un proceso de usuario, y el parámetro `dest` se prueba con una función similar, `issysentn`, para determinar si es FS o MM. Si la combinación no es una de las permitidas, `mini_send` termina con un error.

A continuación se verifica que el destino del mensaje sea un proceso activo, no una ranura vacía de la tabla de procesos (línea 7062). En las líneas 7068 a 7073 `mini_send` verifica si el mensaje cae por completo dentro del segmento de datos de usuario, el segmento de código o el espacio entre ellos. Si no es así, se devuelve un código de error.

La siguiente prueba consiste en verificar si podría haber un bloqueo mutuo. En la línea 7079 hay una prueba para asegurarse de que el destino del mensaje no esté tratando de enviar un mensaje de vuelta al invocador.

La prueba clave en `mini_send` está en las líneas 7088 a 7090. Aquí se verifica si el destino está bloqueado en un RECE! VE examinando el bit RECEIVING del campo `pj7ags` de su entrada en la tabla de procesos. Si el destino está esperando, la siguiente pregunta es “¿quién está esperando?” Si está esperando al emisor, o a cualquiera (ANY), se ejecuta `CopyMess` para copiar el mensaje, y el receptor se desbloquea poniendo en 0 su bit RECEIVING. `CopyMess` se define como una macro en la línea 6932. Esta función invoca la rutina en lenguaje ensamblador `cp_mess` que está en `klib386.s`.

Por otro lado, si el receptor no está bloqueado, o está bloqueado pero esperando un mensaje de alguien más, se ejecuta el código de las líneas 7098 a 7111 a fin de bloquear y poner en cola al emisor. Todos los procesos que desean enviar a un destino dado se colocan en una lista enlazada, y el campo `p_callerq` del destino apunta a la entrada de la tabla de procesos que corresponde al proceso que está a la cabeza de la cola. El ejemplo de la Fig. 2-35(a) muestra lo que sucede cuando el proceso 3 no puede enviar al proceso 0. Si subsecuentemente el proceso 4 tampoco puede enviar al proceso 0, tenemos la situación de la Fig. 2-35(b).

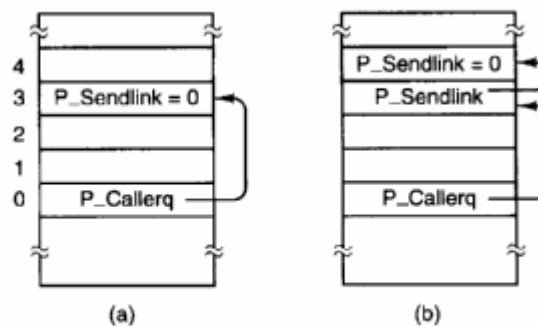


Figura 2-35. Puesta en cola de procesos que tratan de enviar al proceso 0.

`Mini_rec` (línea 6119) es invocada por `sys_cali` cuando su parámetro `function` es RECEIVE (recibir) o BOTH (ambas cosas). El ciclo de las líneas 7137 a 7151 examina todos los procesos que están en cola esperando para enviar al receptor, para ver si alguno de ellos es aceptable. Si se encuentra uno, el mensaje se copia del emisor al receptor; luego el emisor se desbloquea, se marca

como listo para ejecutarse y se quita de la cola de procesos que están tratando de enviar al receptor.

Si no se encuentra un emisor apropiado, se verifica si la bandera `p_int_blocked` del proceso receptor indica que previamente se bloqueó una interrupción para este destino (línea 7154). Si así es, se construye un mensaje en este momento; puesto que los mensajes de `HARDWARE` no tienen más contenido que `HARDWARE` en el campo de origen y `HARD_INT` en el campo de tipo, no hay necesidad de invocar `CopyMess` en este caso.

Si no se encuentra una interrupción bloqueada, las direcciones de origen y de buffer del proceso se guardan en su entrada de la tabla de procesos y el proceso se marca como bloqueado poniendo en 1 su bit `RECEIVING`. La llamada a `unready` en la línea 7165 quita al receptor de la cola de procesos ejecutables del planificador. La llamada es condicional para evitar bloquear de inmediato el proceso si hay otro bit en 1 en su `p_flags`; puede haber una señal pendiente, y el proceso debería tener otra oportunidad de ejecutarse pronto para manejar la señal.

La penúltima instrucción de `mini_rec` (líneas 7171 y 7172) tiene que ver con la forma como se manejan las señales `SIGINT`, `SIGQUIT` y `SIGALARM` generadas por el kernel. Cuando ocurre una de éstas, se envía un mensaje al administrador de memoria, si éste está esperando un mensaje de `ANY`. Si no, la señal se recuerda en el kernel hasta que el administrador de memoria por fin trata de recibir de `ANY`. Eso se prueba aquí y, si es necesario, se invoca `inform` para informarle de las señales pendientes.

2.6.9 Planificación en MINIX

MINIX emplea un algoritmo de planificación multinivel que sigue de cerca la estructura que se muestra en la Fig. 2-26. En esa figura vemos tareas de E/S en la capa 2, procesos de servidor en la capa 3 y procesos de usuario en la capa 4. El planificador mantiene tres colas de procesos ejecutables, una para cada capa, como se muestra en la Fig. 2-36. El arreglo `rdy_head` tiene una entrada por cada cola, y cada entrada apunta al proceso que está a la cabeza de la cola correspondiente. De forma similar, `rdy_tail` es un arreglo cuyas entradas apuntan al último proceso de cada cola. Ambos arreglos se definen con la macro `EXTERN` en `proc.h` (líneas 5192 y 5193).

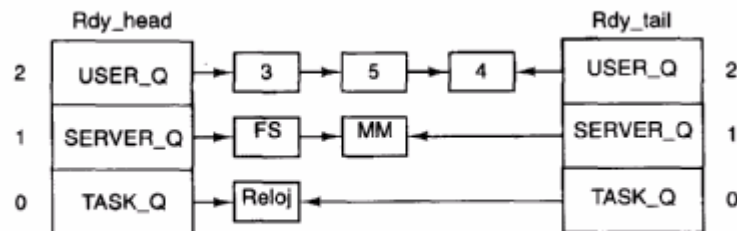


Figura 2-36. El planificador mantiene tres colas, una por cada nivel de prioridad.

Cada vez que se despierta un proceso bloqueado, se le anexa al final de su cola. La existencia del arreglo `rdy_tail` hace eficiente la acción de agregar un proceso al final de una cola. Cuando un

proceso en ejecución se bloquea, o un proceso ejecutable es terminado por una señal, ese proceso se quita de las colas del planificador. Sólo se ponen en cola los procesos ejecutables.

Dadas las estructuras de cola que acabamos de describir, el algoritmo de planificación es sencillo: encontrar la cola con más alta prioridad que no está vacía y escoger el proceso que está a la cabeza de esa cola. Si todas las colas están vacías, se ejecuta la rutina de “marcha en vacío”. En la Fig. 2-36 TASK_Q tiene la más alta prioridad. El código de planificación está en `proc.c`. La cola se escoge en `pick_proc` (línea 7179). La tarea principal de esta función es establecer `proc_ptr`. Cualquier cambio a las colas que pudiera afectar la selección del siguiente proceso por ejecutar requiere una nueva invocación de `pick_proc`. Siempre que el proceso en curso se bloquea, se invoca `pick_proc` para replanificar la CPU.

`Pick_proc` es sencilla. Hay una prueba para cada cola. Primero se prueba TASK_Q, y si un proceso de esta cola está listo, `pick_proc` establece `proc_ptr` y regresa de inmediato. A continuación se prueba SERVER_Q y, otra vez, si hay un proceso listo `pick_proc` establece `proc_ptr` y regresa. Si hay un proceso listo en la cola USER_Q, se modifica `bill_ptr` a fin de cobrar al proceso de usuario el tiempo de CPU que está a punto de concedérsele (línea 7198). Esto asegura que se cobre al último proceso de usuario ejecutado el trabajo realizado a su nombre por el sistema. Si ninguna de las colas tiene una tarea lista, la línea 7204 transfiere la facturación al proceso IDLE y lo planifica. El proceso escogido para ejecutarse no se quita de la cola en que está por el mero hecho de haber sido seleccionado.

Los procedimientos `ready` (línea 7210) y `unready` (línea 7258) se invocan para ingresar un proceso ejecutable en su cola y quitar de su cola un proceso que ya no es ejecutable, respectiva mente. `Ready` se invoca tanto desde `rmini_send` como desde `mini_rec`, como hemos visto. También podría haberse llamado desde `interrupt`, pero en aras de agilizar el procesamiento de interrupciones su equivalente funcional se incluyó en `interrupt` como código en línea. `Ready` manipula una de las tres colas de procesos, agregando directamente el proceso al final de la cola apropiada.

`Unready` también manipula las colas. Normalmente, el proceso que quita está a la cabeza de su cola, ya que un proceso necesita estar ejecutándose para poder bloquearse. En tal caso `unready` invoca `pick_proc` antes de regresar, como por ejemplo en la línea 7293. Un proceso de usuario que no se está ejecutando también puede dejar de estar listo si se le envía una señal, y si el proceso está a la cabeza de una de las colas se busca en USER_Q, eliminándose si se encuentra.

Aunque la mayor parte de las decisiones de planificación se toman cuando un proceso se bloquea o desbloquea, también debe efectuarse planificación cuando la tarea del reloj se da cuenta de que el proceso de usuario actual excedió su cuanto. En este caso la tarea del reloj invoca `sched` (línea 7311) para pasar el proceso que está a la cabeza de USER_Q al final de esa cola. Este algoritmo hace que los procesos de usuario se ejecuten por round robin simple. El sistema de archivos, el administrador de memoria y las tareas de E/S nunca se colocan al final de sus colas por haberse estado ejecutando durante demasiado tiempo; se confía en que funcionarán correctamente y se bloquearán después de haber terminado su trabajo.

Hay unas cuantas rutinas más en `proc.c` que apoyan la planificación de procesos. Cinco de ellas, `lock_mini_send`, `lock_pick_proc`, `lock_ready`, `lock_unready` y `lock_sched`, establecen un candado, usando la variable `switching` antes de invocar la función correspondiente, y liberan el candado al completar su trabajo. La última función de este archivo, `unhold` (línea 7400), se

mencionó cuando explicamos `_restart` en `mpx386.s`. Unhoid procesa cíclicamente la cola de interrupciones detenidas, invocando `interrupt` para cada una, a fin de convertir todas las interrupciones pendientes en mensajes antes de que se permita a otro proceso ejecutarse.

En síntesis, el algoritmo de planificación mantiene tres colas de prioridad, una para las tareas de E/S, otra para los procesos de servidor y otra para los procesos de usuario. Siempre se escoge para ejecutarse a continuación el primer proceso de la cola de más alta prioridad. Siempre se permite a las tareas y los servidores ejecutarse hasta bloquearse, pero la tarea del reloj vigila el tiempo utilizado por los procesos de usuario. Si un proceso de usuario agota su cuanto, se coloca al final de la cola, implementando así una planificación round robin simple entre los procesos de usuario en competencia.

2.6.10 Apoyo de kernel dependiente del hardware

Hay varias funciones de C que dependen en gran medida del hardware. Con objeto de facilitar el traslado de MINIX a otros sistemas, esas funciones se han segregado en los archivos que veremos en esta sección, `exception.c`, `i8259.c` y `protect.c`, en lugar de incluirse en los archivos en los que está el código de más alto nivel al que apoyan.

`Exception.c` contiene el manejador de excepciones, `exception` (línea 7512) que es invocado (como `_exception`) por la parte de lenguaje de ensamblador del código de manejo de excepciones de `mpx386.s`. Las excepciones que se originan en procesos de usuario se convierten en señales. Se espera que los usuarios cometan errores en sus propios programas, pero una excepción que se origina en el sistema operativo indica que algo anda en verdad mal y causa un pánico. El arreglo `ex_data` (líneas 7522 a 7540) determina el mensaje de error que se exhibirá en caso de pánico, o la señal que se enviará a un proceso de usuario para cada excepción. Los primeros procesadores Intel no generan todas las excepciones, y el tercer campo de cada entrada indica el modelo de procesador mínimo que puede generar cada una. Este arreglo ofrece un resumen interesante de la evolución de la familia de procesadores Intel en la que se implementó MINIX. En la línea 7563 se exhibe un mensaje alternativo si se produce un pánico por una interrupción que no se esperaba del procesador que se está usando.

Las tres funciones de `i8259.c` se usan durante la inicialización del sistema para asignar valores iniciales a los chips controladores de interrupciones Intel 8259. `Intr_init` (línea 7621) inicializa los controladores escribiendo datos en varias direcciones de puertos. En unas cuantas líneas se prueba una variable derivada de los parámetros de arranque para dar cabida a diferentes modelos de computadoras. Por ejemplo, en la línea 7637 se escribe en el primer puerto, con dirección determinada por el tipo de hardware. En la línea 7638, y otra vez en la línea 7644, se prueba el parámetro `mine`, escribiéndose en el puerto un valor apropiado ya sea para MINIX o para el BIOS ROM. Al salir de MINIX se puede invocar `intr_init` para restaurar los vectores de BIOS, efectuándose así una salida digna al monitor de arranque. `Mine` selecciona el modo que se usará. Para entender perfectamente lo que está sucediendo aquí sería necesario estudiar la documentación del circuito integrado 8259, así que no entraremos en detalles. Señalaremos que la llamada `out_byte` de la línea 7642 hace que el controlador maestro deje de responder a cualquier entrada excepto la que

proviene del esclavo, y la operación similar de la línea 7648 inhibe la respuesta del esclavo a sus entradas. Además, la última línea de la función precarga la dirección de `spurious_irq`, la siguiente función del archivo (línea 7657), en cada ranura de `irq` jable. Esto asegura que cualquier interrupción generada antes de que se instalen los manejadores reales no cause ningún daño.

La última función de `i8259.c` es `put_irq_handler` (línea 7673) Durante la inicialización cada que debe responder a una interrupción invoca esta función para colocar la dirección de su `irq` manejador en la tabla de interrupciones sobrescribiendo la dirección de `spurious_irq`

`Protect.c` contiene rutinas relacionadas con la operación en modo protegido de los procesadores `tel`. La tabla de descriptores globales (GDT), las tablas de descriptores locales (LDT) y la tabla de descriptores de interrupciones (IDT) todas situadas en la memoria, proporcionan acceso protegido a los recursos del sistema. Registros especiales dentro de la CPU apuntan a la GDT y a las IDT, y las entradas de la GDT apuntan a las LDT. La GDT está accesible para todos los procesos y contiene descriptores de segmento de las regiones de memoria utilizadas por el sistema operativo. Normalmente hay una LDT para cada proceso, y contiene descriptores de segmento para las regiones de memoria que el proceso usa. Los descriptores son estructuras de ocho bytes con varios componentes, pero las partes más importantes de un descriptor de segmento son los campos que describen la dirección base y el límite de una región de memoria. La IDT también se compone de descriptores de ocho bytes, siendo la parte más importante la dirección del código que se ejecutará cuando se active la interrupción correspondiente.

`Prot_init` (línea 7767) es invocada por `start.c` para establecer la GDT en las líneas 7828 a 7845. El IBM PC BIOS requiere que esta tabla esté ordenada de cierta manera, y todos los índices para acceder a ella se definen en `protect.h`. En la tabla de procesos se asigna espacio para una LDT para cada proceso. Cada LDT contiene dos descriptores, para un segmento de código y uno de datos (recuerde que aquí estamos hablando de segmentos definidos por el hardware; éstos no son los mismos segmentos administrados por el sistema operativo, el cual considera que el segmento de datos definido por el hardware está dividido en segmentos de datos y de pila). En las líneas 7851 a 7857 se construyen en la GDT descriptores para cada LDT. Las funciones `init_dataseg` e `init_codeseg` realmente construyen estos descriptores. Las entradas de las LDT mismas se inicializan cuando se modifica el mapa de memoria de un proceso (es decir, cuando se emite una llamada al sistema EXEC).

Otra estructura de datos del procesador que requiere inicialización es el segmento de estado de tareas (TSS). La estructura se define al principio de este archivo (líneas 7725 a 7753) y proporciona espacio para almacenar registros del procesador y otra información que debe guardarse cuando se efectúa una conmutación de tarea. MINIX usa sólo los campos que definen dónde debe construirse una nueva pila cuando ocurre una interrupción. La llamada a `init_dataseg` de la línea 7867 asegura que se le podrá encontrar usando la GDT.

Si queremos entender cómo funciona MINIX en el nivel más bajo, tal vez la cosa más importante sea entender la forma en que las excepciones, interrupciones de hardware o instrucciones `mt <nnn>` dan pie a la ejecución de las distintas secciones de código que se han escrito para atenderlas. Esto se logra por medio de la tabla de descriptores de puertas de interrupciones. El compilador inicializa el arreglo `gate_table` (líneas 7786 a 7818) con las direcciones de las rutinas que manejan las excepciones e interrupciones de hardware y luego se usa este arreglo en el ciclo de las líneas 7873

a 7877 para inicializar una buena parte de la tabla mencionada, usando llamadas a la función `int_gate`. Los vectores restantes, `SYS_VECTOR`, `SYS386_VECTOR` y `LE VELO_VECTOR`, requieren niveles de privilegio distintos y se inicializan después del ciclo.

Existen buenas razones para estructurar los datos de la forma como se hace en los descriptores, basándose en detalles del hardware y en la necesidad de mantener la compatibilidad entre los procesadores avanzados y el procesador 286 de 16 bits. Por fortuna, normalmente podemos dejar estos detalles a los diseñadores de procesadores de Intel. En general, el lenguaje C nos permite evitar los detalles. Sin embargo, al implementar un sistema operativo real es necesario enfrentar los detalles en algún momento. En la Fig. 2-37 se muestra la estructura interna de un tipo de descriptor de segmento. Observe que la dirección base, a las que los programas en C pueden referirse con un entero sin signo de 32 bits, se divide en tres partes, dos de las cuales están separadas por varias cantidades de 1, 2 y 4 bits. El límite es una cantidad de 20 bits almacenada como un trozo de 16 bits y uno de 4 bits. Este límite se interpreta ya sea como un número de bytes o como un número de páginas de 4096 bytes, con base en el valor del bit G (de granularidad). Otros descriptores, como los que se usan para especificar la forma de manejar las interrupciones, tienen estructuras diferentes, pero igualmente complejas. Analizaremos tales estructuras con mayor detalle en el capítulo 4.

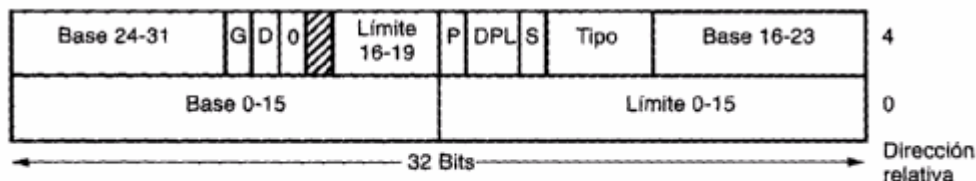


Figura 2-37. Formato de un descriptor de segmento Intel.

La mayor parte de las otras funciones definidas en `protec.c` sirven para realizar la conversión entre las variables empleadas en los programas en C y las formas más bien feas que esos datos adoptan en los descriptores legibles por la máquina como el de la Fig. 2-37. `init_codeseg` (línea 7889) e `init_dataseg` (línea 7906) tienen un funcionamiento similar y sirven para convertir los parámetros que les son pasados en descriptores de segmentos. Cada una, a su vez, invoca a la siguiente función, `sdesc` (línea 7922) para completar el trabajo. Aquí es donde se manejan los desagradables detalles de la estructura de la Fig. 2-37. `init_codeseg` e `init_data_seg` no sólo se utilizan durante la inicialización del sistema. Además, son invocadas por la tarea del sistema cada vez que se inicia un nuevo proceso, a fin de asignar los segmentos de memoria correctos que el proceso usará. `Seg2phys` (línea 7947), que sólo se invoca desde `start.c`, realiza una operación que es el inverso de la de `sdesc`, extraer de un descriptor de segmento la dirección base de un segmento. `int_gate` (línea 7969) realiza una labor similar a la de `init_codeseg` e `init_dataseg`, construyendo entradas para la tabla de descriptores de interrupciones.

La última función de `protect.c`, `enable_iop` (línea 7988) tiene una misión subrepticia. Ya hemos señalado varias veces que una de las funciones del sistema operativo es proteger los recursos del sistema, y una forma en que MINIX hace esto consiste en usar niveles de privilegio para que los procesos de usuario no puedan utilizar ciertos tipos de instrucciones. Sin embargo, también se pretende que MINIX se utilice en sistemas pequeños, que con toda probabilidad sólo tendrán un usuario o tal vez unos cuantos usuarios de confianza. En un sistema así, un usuario bien podría querer escribir un programa de aplicación que tenga acceso a los puertos de E/S, por ejemplo, para usarlos en la obtención de datos científicos. El sistema de archivos tiene incorporado un pequeño secreto: si se abren los archivos `Idevimem` o `/dev/kmem`, la tarea de memoria invoca `enable_iop`, la cual cambia el nivel de privilegio de las operaciones de E/S, permitiendo al proceso actual ejecutar instrucciones que leen los puertos de E/S y escriben en ellos. La descripción del propósito de la función es más complicada que la función misma, que simplemente pone en 1 dos bits de la palabra en la entrada del soporte de pila del proceso invocador que se cargará en el registro de estado de la CPU la próxima vez que el proceso se ejecute. No se necesita otra función que revierta esta acción, pues sólo se aplicará al proceso invocador.

2.6.11 Rutinas de utilidad y biblioteca del kernel

Por último, el kernel tiene una biblioteca de funciones de apoyo escritas en lenguaje ensamblador, que se incluyen compilando `klib.s`, y unos cuantos programas de utilidad, escritos en C, en el archivo `misc.c`. Examinemos primero los archivos en lenguaje ensamblador. `Klib.s` (línea 8000) es un archivo corto similar a `mpx.s`, que selecciona la versión apropiada, específica para la máquina, con base en la definición de `WORD_SIZE` (tamaño de palabra). El código que analizaremos está en `k11b386.s` (línea 8100), y contiene unas dos docenas de rutinas de utilidad escritas en lenguaje ensamblador ya sea por razones de eficiencia o porque no es posible escribirlas en C.

`_Monitor` (línea 8166) permite regresar al monitor de arranque. Desde el punto de vista del monitor de arranque, todo MINIX no es más que una subrutina, y cuando se inicia MINIX se deja una dirección de retorno al monitor en la pila de éste. Lo único que tiene que hacer `_monitor` es restaurar los diversos selectores de segmento y el apuntador a la pila que se guardó cuando MINIX se inició, y luego regresar como de cualquier otra subrutina.

La siguiente función, `_check_mem` (línea 8198), se usa en el momento de iniciarse MINIX para determinar el tamaño de un bloque de memoria. Esta función realiza una prueba sencilla con cada decimosexto byte, usando dos patrones que prueban cada bit con valores tanto “0” como “1”.

Aunque podría haberse usado `_phys_copy` (véase más adelante) para copiar mensajes, se ha suministrado `_cp_mess` (línea 8243) para ese fin, pues es un procedimiento especializado más rápido. Este procedimiento se invoca con

```
cp_mess(source, src_clicks, src_offset, dest_clicks, dest_offset);
```

donde `source` es el número de proceso del emisor, que se copia en el campo `m_source` del buffer del receptor. Las direcciones tanto de origen como de destino se especifican dando un número de `click`, por lo regular la base del segmento que contiene el buffer, y un desplazamiento respecto a

ese click. Esta forma de especificar el origen y el destino es más eficiente que las direcciones de 32 bits utilizadas por `_phys_copy`.

`Exit`, `— exit y —` `_exit` (líneas 8283 a 8285) se definen porque algunas rutinas de biblioteca que podrían usarse para compilar MINIX hacen llamadas a la función C estándar `exit`. Salir del kernel no es un concepto significativo, pues no hay adónde ir. La solución aquí es habilitar las interrupciones y entrar en un ciclo infinito. Tarde o temprano una operación de E/S o el reloj causará una interrupción y se reanudará el funcionamiento normal del sistema. El punto de entrada para `_main` (línea 8289) es otro intento de manejar una acción del compilador que, si bien podría tener sentido durante la compilación de un programa de usuario, no tiene ningún objeto en el kernel. Esta función apunta a una instrucción `ret` (regresar de subrutina) de lenguaje ensamblador.

`in_byte` (línea 8300), `_in` (línea 8314), `out_byte` (línea 8328) y `_out_word` (línea 8342) proporcionan acceso a puertos de E/S, que en el hardware Intel ocupan un espacio de direcciones distinto del de la memoria y usan instrucciones diferentes de las que leen y escriben en la memoria. `_Portjead` (línea 8359), `_portjead_byte` (línea 8386), `_port_write` (línea 8412) y `port_write_byte` (línea 8439) se encargan de transferir bloques de datos entre puertos de E/S y la memoria; se usan primordialmente para transferencias desde y hacia el disco que deben efectuarse con mucha mayor rapidez que la que es posible con las otras llamadas de E/S. Las versiones de bytes leen 8 bits en lugar de 16 bits en cada operación, a fin de manejar dispositivos periféricos antiguos de 8 bits.

De vez en cuando, una tarea necesitará inhabilitar todas las interrupciones de la CPU temporalmente; esto lo hace invocando `jock` (línea 8462). Cuando ya puedan habilitarse otra vez las interrupciones, la tarea puede invocar `_unlock` (línea 8474) para hacerlo. Una sola instrucción de máquina realiza cada una de estas operaciones. En contraste, el código para `Enableirq` (línea 8488) y `disableirq` (línea 8521) es más complicado. Estas funciones operan en el nivel de los chips controladores de interrupciones para habilitar e inhabilitar interrupciones de hardware individuales.

`_Phys_copy` (línea 8564) se invoca en C con

```
phys_copy(sourceaddress, destination address, bytes);
```

y copia un bloque de datos de cualquier lugar de la memoria física a cualquier otro lugar. Ambas direcciones son absolutas, es decir, la dirección `O` realmente se refiere al primer byte de todo el espacio de direcciones, y los tres parámetros son longs sin signo.

Las siguientes dos funciones cortas son muy específicas para los procesadores Intel. `_Mem_rdw` (línea 8608) devuelve una palabra de 16 bits de cualquier lugar de la memoria. El resultado se llena con ceros dentro del registro `eax` de 32 bits. La función `_reset` (línea 8623) restablece el procesador cargando el registro de la tabla de descriptores de interrupciones del procesador con un apuntador nulo y ejecutando a continuación una interrupción de software. Esto tiene el mismo efecto que un “reset” de hardware.

Las dos siguientes rutinas apoyan la exhibición en pantalla y son utilizadas por la tarea de la consola. `_Mem_vid_copy` (línea 8643) copia una cadena de palabras que contienen bytes alternos de caracteres y atributos desde la región de memoria del kernel a la memoria de pantalla, `Vid_vid_copy` (línea 8696) copia un bloque dentro de la memoria de video misma. Esto es un

poco más complicado, ya que el bloque de destino y el de origen pueden traslaparse, y la dirección del traslado es importante.

La última función de este archivo es `_level0` (línea 8773), que permite a las tareas tener el nivel de permisos más privilegiado, el nivel cero, en caso necesario. Esta función se usa para cosas tales como restablecer la CPU o acceder a las rutinas del ROM BIOS de la PC.

Las rutinas de utilidad en C contenidas en `misc.c` son especializadas. `Mem_init` (línea 8820) sólo es invocada por `main`, cuando se inicia MINIX. En una computadora compatible con la IBM PC puede haber dos o tres regiones de memoria no continuas. El BIOS informa al monitor de arranque el tamaño del intervalo más bajo, que los usuarios de PC conocen como memoria “ordinaria”, y el del intervalo de memoria que comienza arriba del área de ROM (memoria “extendida”). A su vez, el monitor pasa los valores como parámetros de arranque, que son interpretados por `start` y se escriben en `low_memsize` y `ext_memsize` durante el arranque. La tercera región es la memoria “sombra”, en la que puede copiarse el BIOS ROM para mejorar el rendimiento, ya que la memoria ROM normalmente es más lenta que la RAM. Puesto que MINIX normalmente no usa el BIOS, `mem_init` trata de localizar esta memoria y agregarla a la reserva de memoria disponible para su uso. Esto lo hace invocando `check_mem` para probar la región de memoria donde a veces se puede encontrar la memoria en cuestión.

La siguiente rutina, `env_parse` (línea 8865) también se usa durante el inicio del sistema. El monitor de arranque puede pasar cadenas arbitrarias como “DPETHO=300: 10” a MINIX en los parámetros de arranque. `Env_parse` trata de encontrar una cadena cuyo primer campo coincida con su primer argumento, `env`, y luego extraer el campo solicitado. Los comentarios del código explican el uso de la función, que se proporciona principalmente para ayudar al usuario que desea agregar nuevos controladores que tal vez requieran parámetros. El ejemplo “DPETHO” se usa para pasar información de configuración a un adaptador de Ethernet cuando se incluye apoyo de red durante la compilación de MINIX.

Las últimas dos rutinas que veremos en este capítulo son `bad_assertion` (línea 8935) y `bad_compare` (línea 8947). Éstas sólo se compilan si la macro `DEBUG` se define como `TRUE`, y apoyan las macros de `assert.h`. Aunque no se hace referencia a ellas en ninguna parte del código que describimos en el texto, pueden ser de utilidad durante la depuración para el lector que desee crear una versión modificada de MINIX.

2.7 RESUMEN

A fin de ocultar los efectos de las interrupciones, los sistemas operativos ofrecen un modelo conceptual que consiste en procesos secuenciales que se ejecutan en paralelo. Los procesos pueden comunicarse entre sí mediante primitivas de comunicación entre procesos, como son los semáforos, monitores o mensajes. Estas primitivas aseguran que dos procesos nunca estarán en sus regiones críticas al mismo tiempo. Un proceso puede estar ejecutándose, listo o bloqueado, y puede cambiar de estado cuando él u otro proceso ejecuta una primitiva de comunicación entre procesos.

Se pueden usar las primitivas de comunicación entre procesos para resolver problemas tales como el de productor-consumidor, cena de filósofos, lector-escritor y peluquero dormido. Incluso

con estas primitivas, debe tenerse cuidado para evitar errores y bloqueos. Se conocen muchos algoritmos de planificación, incluidos round robin, planificación por prioridad, colas multinivel y planificadores controlados por políticas.

MINIX maneja el concepto de procesos y proporciona mensajes para la comunicación entre procesos. Los mensajes no se guardan en buffers, así que un SEND sólo tiene éxito cuando el receptor lo está esperando. De forma similar, un RECEIVE sólo tiene éxito cuando está disponible un mensaje. Si cualquiera de estas operaciones no tiene éxito, el invocador se bloquea.

Cuando ocurre una interrupción, el nivel más bajo del kernel crea y envía un mensaje a la tarea asociada al dispositivo interruptor. Por ejemplo, la tarea de disco invoca receive y se bloquea después de escribir un comando al hardware controlador del disco pidiéndole que lea un bloque de datos. Una vez que los datos están listos, el hardware controlador hace que ocurra una interrupción. A continuación, el software de bajo nivel elabora un mensaje para la tarea de disco y la marca como ejecutable. Cuando el planificador escoge la tarea de disco para que se ejecute, obtiene y procesa el mensaje. El manejador de interrupciones también puede realizar cierto trabajo directamente, como cuando una interrupción de reloj actualiza la hora.

Después de una interrupción puede haber conmutación de tareas. Cuando un proceso se interrumpe, se crea una pila dentro de la entrada correspondiente a ese proceso en la tabla de procesos, y en esa pila se coloca toda la información necesaria para reiniciar el proceso. Cualquier proceso puede reiniciarse ajustando el apuntador de pila de modo que apunte a su entrada en la tabla de procesos e iniciando una secuencia de instrucciones para restaurar los registros de la CPU, que culmina con una instrucción iretd. El planificador decide cuál entrada de la tabla de procesos se colocará en el apuntador a la pila.

También ocurren interrupciones mientras se está ejecutando el kernel mismo. La CPU detecta esto, y se usa la pila del kernel en lugar de una pila dentro de la tabla de procesos. Esto permite que ocurran interrupciones anidadas, y cuando termina una rutina de servicio de interrupción posterior, puede llegar a su término la que está abajo de ella. Una vez atendidas todas las interrupciones, se reinicia un proceso.

El algoritmo de planificación de MINIX usa tres colas de prioridad: la más alta para tareas, la siguiente para el sistema de archivos, el administrador de memoria y otros servidores, silos hay, y la más baja para los procesos de usuario. Estos últimos se ejecutan por round robin durante un cuanto a la vez; todos los demás se ejecutan hasta bloquearse o ser desalojados.

PROBLEMAS

1. Suponga que va a diseñar una arquitectura de computadora avanzada que realizará la conmutación de procesos por hardware, en lugar de tener interrupciones. ¿Qué información necesitaría la CPU? Describa cómo podría funcionar la conmutación de procesos por hardware.
2. En todas las computadoras actuales, al menos una parte de los manejadores de interrupciones se escribe en lenguaje ensamblador. ¿Por qué?

3. En el texto se dijo que el modelo de la Fig. 2-6(a) no es apropiado para un servidor de archivos que usa un caché en memoria. ¿Por qué no? ¿Podría cada proceso tener su propio caché?

4. En un sistema con hilos, ¿hay una pila por hilo o una pila por proceso? Explique.

5. ¿Qué es una condición de competencia?

6. Escriba un guión (script) de shell que produzca un archivo de números secuenciales leyendo el último número del archivo, sumándole 1 y anexándolo al final del archivo. Ejecute un ejemplar del guión en segundo plano y otro en primer plano, con ambos accediendo al mismo archivo. ¿Cuánto tiempo tarda en manifestarse una condición de competencia? ¿Cuál es la sección crítica? Modifique el guión a modo de prevenir la competencia (sugerencia: use

In file file.lock

para poner un candado al archivo de datos).

7. Una instrucción como

In file file.lock

¿es un mecanismo de candado efectivo para un programa de usuario como los guiones empleados en el problema anterior? ¿Por qué sí o por qué no?

8. La solución de espera activa usando la variable turn (Fig. 2-8) ¿funciona cuando los dos procesos se están ejecutando en un multiprocesador de memoria compartida, es decir, dos CPU que comparten la misma memoria?

9. Considere una computadora que no cuenta con la instrucción TEST AND SET LOCK pero sí tiene una instrucción que intercambia el contenido de un registro y una palabra de memoria en una sola acción indivisible. ¿Se puede usar esta instrucción para escribir una rutina enter_region como la de la Fig. 2-10?

10. Bosqueje la forma en que un sistema operativo que puede inhabilitar interrupciones podría implementar semáforos.

11. Muestre cómo pueden implementarse semáforos contadores (es decir, semáforos capaces de contener un valor arbitrariamente grande) usando sólo semáforos binarios e instrucciones de máquina ordinarias,

12. En la sección 2.2.4 se describió una situación con un proceso de alta prioridad, H, y uno de baja prioridad, L, que condujo a la repetición infinita de H. ¿Ocurre el mismo problema si se usa planificación round robin en vez de planificación por prioridad? Explique.

13. La sincronización dentro de los monitores se logra usando variables de condición y dos operaciones especiales, WA y SIGNAL. Una forma más general de sincronización se lograría con la ayuda de una sola primitiva, WAITUNTIL (esperar hasta) que tuviera un predicado booleano arbitrario como parámetro. Así, podríamos escribir, por ejemplo,

WAITUNTIL $x < 0$ o $y + z < n$

La primitiva SIGNAL ya no se necesitaría. Queda claro que este esquema es más general que el de Hoare o el de Brinch Hansen, y sin embargo no se usa. ¿Por qué no? (Sugerencia: piense en la implementación.)

14. Un restaurante de “comida rápida” tiene cuatro tipos de (1) receptores de pedidos, que toman los pedidos de los clientes; (2) cocineros, que preparan la comida; (3) especialistas en empaçado, que meten la comida en bolsas; y (4) cajeros, que entregan las bolsas a los clientes y reciben su dinero. Cada empleado puede considerarse como un proceso secuencial en comunicación. ¿Qué forma de comunicación entre procesos utilizan? Relacione este modelo con los procesos en MINIX.

15. Suponga que tenemos un sistema de transferencia de mensajes que usa buzones. Al enviar mensajes a un buzón lleno o tratar de recibirlos de un buzón vacío, un proceso no se bloquea, sino que recibe de vuelta un código de error. El proceso responde al código de error intentándolo de nuevo, una y otra vez, hasta que tiene éxito. ¿Da este esquema lugar a condiciones de competencia?

16. En la solución al problema de la cena de filósofos (Fig. 2-20), ¿por qué se asigna HUNGRY (hambriento) a la variable de estado en el procedimiento `take_forks` (tomar tenedores)?

17. Considere el procedimiento `put_forks` (poner tenedores) de la Fig. 2-20. Suponga que se asigna el valor THINKING (pensando) a la variable de estado `state` después de las dos llamadas a `test` (probar), en lugar de antes. ¿Cómo afectaría este cambio la solución para el caso de tres filósofos? ¿Y para 100 filósofos?

18. El problema de lectores y escritores se puede formular de varias formas en lo tocante a cuál categoría de procesos puede iniciarse y cuándo. Describa minuciosamente tres variaciones diferentes del problema, cada una de las cuales favorece (o no favorece) alguna categoría de procesos. Para cada variación, explique qué sucede cuando un lector o un escritor queda listo para acceder a la base de datos, y qué sucede cuando un proceso termina de usar la base de datos.

19. Las computadoras CDC 6600 podían manejar hasta 10 procesos de E/S simultáneamente usando una forma interesante de planificación round robin llamada compartición de procesador. Ocurría una conmutación de proceso después de cada instrucción, de modo que la instrucción 1 provenía del proceso 2, la instrucción 2 provenía del proceso 2, etc. La conmutación de procesos se efectuaba mediante un hardware especial, y el gasto extra era cero. Si un proceso necesitaba T segundos para llegar a su fin en la ausencia de competidores, ¿cuánto tiempo necesitaría si se usara compartición de procesador con n procesos?

20. Los planificadores round robin normalmente mantienen una lista de todos los procesos ejecutables, y cada proceso aparece una y sólo una vez en la lista. ¿Qué sucedería si un proceso ocurriera dos veces en la lista? ¿Puede usted pensar en alguna razón para permitir esto?

21. Mediciones realizadas en cierto sistema indican que, en promedio, un proceso se ejecuta durante un tiempo T antes de bloquearse en espera de E/S. Una conmutación de procesos requiere un tiempo S , que efectivamente se desperdicia (gasto extra). Para planificación round robin con cuanto Q , deduzca una fórmula para la eficiencia de la CPU en cada uno de los siguientes casos:

(a) $Q = \infty$

(b) $Q > T$

(c) $S < Q < T$

(d) $Q = S$

(e) Q casi 0

22. Cinco trabajos están esperando para ejecutarse. Sus tiempos de ejecución esperados son 9, 6, 3, 5 y X . ¿En qué orden deben ejecutarse si se desea minimizar el tiempo medio de respuesta? (Su respuesta dependerá de X .)

23. Cinco trabajos por lotes, A a E, llegan a un centro de cómputo casi al mismo tiempo, y tienen tiempos de ejecución estimados de 10, 6, 2, 4 y 8 minutos. Sus prioridades (determinadas externamente) son 3, 5, 2, 1 y 4, respectivamente, siendo 5 la prioridad más alta. Para cada uno de los siguientes algoritmos de planificación, determine el tiempo de retorno medio de los procesos. Ignore el gasto extra por conmutación de procesos.

- (a) Round robin.
- (b) Planificación por prioridad.
- (c) Primero que llega, primero que se atiende (ejecutados en el orden 10, 6, 2, 4, 8).
- (d) El primer trabajo más corto.

Para (a), suponga que el sistema está multiprogramado, y que cada trabajo recibe una parte equitativa del tiempo de CPU. Para (b) a (d), suponga que sólo se ejecuta un trabajo a la vez, hasta terminar. Todos los trabajos están limitados únicamente por CPU.

24. Un proceso que se ejecuta en CTSS necesita 30 cuantos para llegar a su fin. ¿Cuántas veces debe intercambiarse a memoria, incluida la primera vez (antes de que se ejecute por primera vez)?
25. Se está usando el algoritmo de maduración con $a = 1/2$ para predecir los tiempos de ejecución. Las cuatro ejecuciones anteriores, de la más antigua a la más reciente, tardaron 40, 20, 40 y 15 ms. Determine la predicción para la siguiente ejecución.
26. Un sistema de tiempo real flexible tiene cuatro eventos periódicos con periodos de 50, 100, 200 y 250 ms cada uno. Suponga que los cuatro eventos requieren 35, 20, 10 y x ms de tiempo de CPU, respectivamente. Determine el valor máximo de x para el cual el sistema es planificable.
27. Explique por qué se usa comúnmente la planificación de dos niveles.
28. Durante su ejecución, MINIX mantiene una variable `proc_ptr` que apunta a la entrada para el proceso actual en la tabla de procesos, ¿por qué?
29. MINIX no guarda en buffers los mensajes. Explique por qué esta decisión de diseño causa problemas con las interrupciones del reloj y el teclado.
30. Cuando se envía un mensaje a un proceso dormido en MINIX, se invoca el procedimiento `ready` para colocar ese proceso en la cola de planificación correcta. Lo primero que hace este procedimiento es inhabilitar las interrupciones. Explique.
31. El procedimiento de MINIX `mini_rec` contiene un ciclo. Explique para qué sirve.
32. En esencia, MINIX usa el método de planificación de la Fig. 2-23, con diferentes prioridades para las clases. La clase más baja (procesos de usuario) tiene planificación round robin, pero siempre se permite que las tareas y servidores se ejecuten hasta bloquearse. ¿Es posible que procesos de la clase más baja sufran inanición? ¿Por qué sí o por qué no?
33. ¿Es MINIX apropiado para aplicaciones de tiempo real, como el registro de datos? Si no es así, ¿qué podría hacerse para que lo fuera?
34. Suponga que tiene un sistema operativo que cuenta con semáforos. Implemente un sistema de mensajes. Escriba los procedimientos para enviar y recibir mensajes.
35. Un estudiante de antropología con diplomado en ciencias de la computación se ha embarcado en un proyecto de investigación para determinar si es posible a babuinos africanos qué es un bloqueo mutuo. Localiza un cañón profundo y tiende una cuerda un lado a otro, de modo que los babuinos puedan cruzar el cañón colgándose de la cuerda y pasando un mano sobre la otra. Varios babuinos pueden cruzar al mismo tiempo, siempre que todos vayan en la misma dirección. Si en algún momento están en la cuerda al mismo tiempo babuinos que van hacia el este y otros que van hacia el oeste, ocurrirá un bloqueo (los babuinos quedarán atorados a la mitad) porque un babuino no puede pasar sobre otro mientras cuelgan por encima del cañón. Si un babuino desea cruzar el cañón, debe verificar que ningún otro babuino esté cruzando actualmente en la dirección opuesta. Escriba un programa

usando semáforos que eviten los bloqueos. No se preocupe porque una serie de babuinos dirigidos hacia el este detenga indefinidamente a los babuinos que viajan hacia el oeste.

36. Repita el problema anterior, pero ahora evite la inanición. Cuando un babuino que desea cruzar hacia el este llega a la cuerda y encuentra babuinos cruzando hacia el oeste, espera hasta que la cuerda está vacía, pero no se permite que más babuinos comiencen a cruzar hacia el oeste hasta que al menos un babuino haya cruzado en el otro sentido.

37. Resuelva el problema de la cena de filósofos usando monitores en lugar de semáforos.

38. Agregue código al kernel de MINIX para seguir la pista al número de mensajes enviados por el proceso (o tarea) *i* al proceso (o tarea) *j*. Exhiba en la pantalla esta matriz cuando se pulse la tecla F4.

39. Modifique el planificador de MINIX de modo que lleve un registro de cuánto tiempo de CPU le ha sido concedido a cada proceso de usuario recientemente. Si ninguna tarea o servidor quiere ejecutarse, el planificador escogerá el proceso que ha utilizado menos la CPU.

40. Rediseñe MINIX de modo que cada proceso tenga un campo de nivel de prioridad en su tabla de procesos que pueda servir para otorgar prioridades más altas o más bajas a procesos individuales.

41. Modifique las macros `hwint_master` y `hwint_slave` de `mpx386.s` de modo que las operaciones que ahora ejecuta la función `save` se realicen en línea. Determine el costo en términos de tamaño del código. ¿Puede usted medir el aumento en el rendimiento?

3

ENTRADA/SALIDA

Una de las principales funciones de un sistema operativo es controlar todos los dispositivos de E/S (entrada/salida) de una computadora. El sistema operativo debe enviar comandos a los dispositivos, detectar interrupciones y manejar errores; también debe proveer una interfaz entre los dispositivos y el resto del sistema que sea sencilla y fácil de usar. Se debe tratar al máximo que la interfaz sea la misma para todos los dispositivos (independencia respecto al dispositivo). El código de E/S representa una parte significativa del sistema operativo total. La forma en que el sistema operativo administra la E/S es el tema de este capítulo.

He aquí un bosquejo del capítulo. Primero examinaremos brevemente algunos de los principios del hardware de E/S, y luego estudiaremos el software de E/S en general. El software de E/S puede estructurarse en capas, cada una de las cuales tiene una tarea bien definida que realizar. Analizaremos estas capas para ver qué hacen y cómo se relacionan entre sí.

A continuación viene una sección sobre los bloqueos mutuos. Definiremos éstos de forma precisa, indicaremos sus causas, presentaremos dos mód para analizarlos y estudiaremos algunos algoritmos para prevenir su incidencia.

Después, revisaremos brevemente la E/S de MINIX. A continuación de la introducción, examinaremos cuatro dispositivos de E/S con detalle: el disco en RAM, el disco duro, el reloj y la terminal. Para cada dispositivo estudiaremos su hardware, software e implementación en MINIX. Por último, concluiremos el capítulo con una explicación corta de un pequeño componente de MINIX que se encuentra en la misma capa que las tareas de E/S pero que en sí no es una tarea de E/S. Este código proporciona algunos servicios al administrador de memoria y al sistema de archivos, como la obtención de bloques de datos de un proceso de usuario.

3.1 PRINCIPIOS DEL HARDWARE DE E/S

Los puntos de vista respecto al hardware de E/S son muy distintos para cada persona, dependiendo del campo en el que trabaje. Los ingenieros eléctricos lo ven en términos de chips, alambres, fuentes de potencia, motores y todos los demás componentes físicos que constituyen el hardware. Los programadores tienen en cuenta la interfaz en relación con el software: los comandos que el hardware acepta, las funciones que realiza y los errores que puede informar. En este libro nos interesa la programación de los dispositivos de E/S, no su diseño, construcción ni mantenimiento, así que nuestra atención se limitará a la forma como el hardware se programa, no a cómo funciona internamente. No obstante, la programación de muchos dispositivos de E/S a menudo está íntimamente ligada con su funcionamiento interno. En las tres secciones que siguen presentaremos algunos antecedentes generales del hardware de E/S que pudieran tener relación con su programación.

3.1.1 Dispositivos de E/S

Los dispositivos de E/S se pueden dividir a grandes rasgos en dos categorías: dispositivos por bloques y dispositivos por caracteres. Un dispositivo por bloques almacena información en bloques de tamaño fijo, cada uno con su propia dirección. Los tamaños de bloque comunes van desde 512 bytes hasta 32 768 bytes. La propiedad esencial de un dispositivo por bloques es que es posible leer o escribir cada bloque con independencia de los demás. Los discos son los dispositivos por bloques más comunes.

Si lo analizamos con cuidado, la frontera entre los dispositivos que son direccionables por bloques y los que no lo son no se halla bien definida. Todo el mundo coincide en que un disco es un dispositivo direccionable por bloques, pues sea donde sea que esté el brazo actualmente, siempre es posible buscar otro cilindro y luego esperar que el bloque requerido gire, hasta pasar bajo la cabeza. Consideremos ahora una unidad de cinta DAT de 8 mm empleada para realizar respaldos de disco. Estas cintas generalmente contienen bloques de tamaño fijo. Si la unidad de cinta lee el bloque N, ésta siempre podrá rebobinar la cinta y avanzarla hasta llegar al bloque N. Esta operación es análoga a una búsqueda de disco, excepto que tarda mucho más. Además, podría o no ser posible reescribir un bloque a la mitad de una cinta. Incluso si fuera posible usar las cintas como dispositivos por bloques con acceso directo, ésta no es la forma como se usan normalmente.

El otro tipo de dispositivo de E/S es el dispositivo por caracteres. Un dispositivo de este tipo suministra o acepta una corriente de caracteres, sin contemplar ninguna estructura de bloques; no es direccionable y no tiene una operación de búsqueda. Las impresoras, interfaces de red, ratones (para apuntar), ratas (para experimentos de laboratorio de psicología) y casi todos los demás dispositivos que no se parecen a los discos pueden considerarse como dispositivos por caracteres.

Este esquema de clasificación no es perfecto; algunos dispositivos simplemente no se ajustan a él. Los relojes, por ejemplo, no son direccionables por bloques, ni tampoco generan ni aceptan flujos de caracteres; lo único que hacen es generar interrupciones a intervalos bien definidos. Las pantallas mapeadas en la memoria tampoco se ajustan muy bien al modelo. No obstante, el modelo

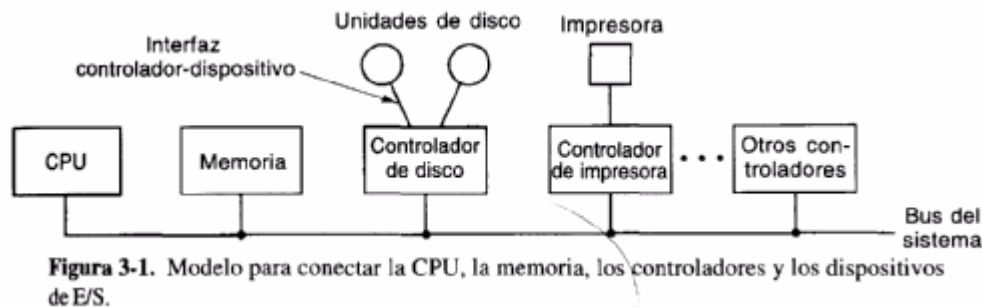
de dispositivos por bloques y por caracteres es lo bastante general como para servir de base para hacer que una parte del software del sistema operativo que se ocupa de E/S sea independiente del dispositivo. El sistema de archivos, por ejemplo, sólo maneja dispositivos por bloques abstractos y deja la parte dependiente del dispositivo a un software de más bajo nivel llamado controladores de dispositivos.

3.1.2 Controladores de dispositivos

Las unidades de E/S por lo regular consisten en un componente mecánico y otro electrónico. En muchos casos es posible separar las dos partes con objeto de tener un diseño más modular y general. El componente electrónico se llama controlador de dispositivo o adaptador. En las computadoras personales, este componente a menudo adopta la forma de una tarjeta de circuitos impresos que se puede insertar en una ranura de la tarjeta matriz de la computadora. El componente mecánico es el dispositivo mismo.

La tarjeta controladora casi siempre tiene un conector en el que puede insertarse un cable que conduce al dispositivo. Muchos controladores pueden manejar dos, cuatro o incluso ocho dispositivos idénticos. Si la interfaz entre el controlador y el dispositivo es de un tipo estándar, ya sea una norma oficial como ANSI, IEEE o ISO, o una norma defacto, las compañías pueden fabricar controladores y dispositivos que se ajustan a esa interfaz. Por ejemplo, muchas compañías producen unidades de disco que se ajustan a las interfaces de controlador de disco IDE (Integrated Drive Electronics) o SCSI (Small Computer System Interface).

Mencionamos esta distinción entre el controlador y el dispositivo porque el sistema operativo casi siempre trata con el controlador, no con el dispositivo. La mayor parte de las computadoras pequeñas usan el modelo de bus único de la Fig. 3-1 para la comunicación entre la CPU y los controladores. Las macrocomputadoras (mainframes) con frecuencia usan un modelo diferente, con múltiples buses y computadoras de E/S especializadas llamadas canales de 110 que asumen parte de la carga de la CPU principal.



La interfaz entre el controlador y el dispositivo suele ser de nivel muy bajo. Un disco, por ejemplo, podría formatearse con 16 sectores de 512 bytes en cada pista. Sin embargo, lo que realmente sale de la unidad es un flujo de bits en serie que comienza con un preámbulo seguido de los 4096 bits del sector y por último una suma de verificación, llamada también código para corrección de errores (ECC: error-correcting code). El preámbulo se escribe cuando se da formato

al disco y contiene el número de cilindro y de sector, el tamaño de sector y datos similares, así como información de sincronización.

La función del controlador consiste en convertir un flujo de bits a un bloque de bytes y realizar las acciones de corrección de errores necesarias. Generalmente, primero se arma el bloque de bytes, bit por bit, en un buffer dentro del controlador. Una vez que se ha cotejado su suma de verificación y se le declara libre de errores, el bloque puede copiarse en la memoria principal.

El controlador de una terminal de tubo de rayos catódicos (CRT) también funciona como dispositivo de bits en serie en un nivel igualmente bajo: lee de la memoria bytes que contienen los caracteres por exhibir y genera las señales que modulan el haz del CRT para hacer que escriba en la pantalla. Además, el controlador genera las señales para hacer que el haz del CRT realice un barrido horizontal una vez que ha recorrido una línea, así como las señales que hacen que el haz realice un barrido vertical después de haber recorrido toda la pantalla. Si no fuera por el controlador del CRT, el programador del sistema operativo tendría que programar explícitamente la exploración analógica del tubo. Usando un controlador, el sistema operativo inicializa el controlador con unos cuantos parámetros, como el número de caracteres por línea y el número de líneas por pantalla, y deja que el controlador se encargue de guiar realmente el haz.

Cada controlador tiene unos cuantos registros que sirven para comunicarse con la CPU. En algunas computadoras estos registros forman parte del espacio de direcciones de la memoria normal. Este esquema se denomina E/S mapeada en memoria. Por ejemplo, el 680x0 usa este método. Otras computadoras utilizan un espacio de direcciones especial para E/S, y a cada controlador se le asigna una porción. La asignación de direcciones de E/S a los dispositivos se realiza mediante lógica de descodificación del bus asociado al controlador. Algunos fabricantes de las llamadas IBM PC compatibles utilizan direcciones de E/S distintas de las que IBM usa. Además de los puertos de E/S, muchos controladores usan interrupciones para indicarle a la CPU cuándo están listos para que sus registros sean leídos o escritos. Una interrupción es, en primera instancia, un suceso eléctrico. Una línea de petición de interrupción (IRQ) de hardware es una entrada física del chip controlador de interrupciones. El número de tales entradas es limitado; las PC de tipo Pentium sólo tienen 15 entradas disponibles para dispositivos de E/S. Algunos controladores están alambrados físicamente en la tarjeta matriz del sistema, como, por ejemplo, el controlador del teclado en una IBM PC. En el caso de un controlador que se enchufa en el plano posterior, a veces pueden usarse interruptores o puentes de alambre en el controlador de dispositivo para seleccionar la IRQ que usará el dispositivo, a fin de evitar conflictos (aunque en algunas tarjetas, como Plug 'n Play, las IRQ pueden establecerse en software). El chip controlador de interrupciones establece una correspondencia entre cada entrada IRQ y un vector de interrupción, que localiza la rutina de servicio de interrupción conexas. A guisa de ejemplo, en la Fig. 3-2 se muestran las direcciones de E/S, interrupciones de hardware y los vectores de interrupción asignados a algunos de los controladores de una IBM PC. MINIX usa las mismas interrupciones de hardware, pero los vectores de interrupción de MINIX son diferentes de los que se muestran aquí para MS-DOS.

El sistema operativo realiza la E/S escribiendo comandos en los registros del controlador. Por ejemplo, el controlador de la unidad de disquete de una IBM PC acepta 15 comandos distintos, como READ (leer), WRITE (escribir), SEEK (buscar), FORMAT (formatear) y RECALIBRATE

Controlador de E/S	Dirección de E/S	IRQ de hardware	Vector de interrupción
Reloj	040 – 043	0	8
Teclado	060 – 063	1	9
Disco duro	1F0 – 1F7	14	118
RS232 secundaria	2F3 – 2FF	3	11
Impresora	378 – 37F	7	15
Disco flexible	3F0 – 3F7	6	14
RS232 primaria	3F8 – 3FF	4	12

Figura 3-2. Algunos ejemplos de controladores, sus direcciones de E/S, sus líneas de interrupción de hardware y sus vectores de interrupción en una PC típica que ejecuta MS-DOS.

(recalibrar). Muchos de los comandos tienen parámetros, que también se cargan en los registros del controlador. Una vez que un controlador ha aceptado un comando, la CPU puede dejarlo libre y ponerse a hacer otra cosa. Una vez que se ha llevado a cabo el comando, el controlador causa una interrupción para que el sistema operativo pueda recuperar el control de la CPU y probar los resultados de la operación. La CPU obtiene los resultados y el estado del dispositivo leyendo uno o más bytes de información de los registros del controlador.

3.1.3 Acceso directo a memoria (DMA)

Muchos controladores, sobre todo los de dispositivos por bloques, manejan el acceso directo a memoria o DMA. Para explicar el funcionamiento del DMA, veamos primero cómo ocurren las lecturas de disco cuando no se usa DMA. Primero el controlador lee el bloque (uno o más sectores) de la unidad en serie, bit por bit, hasta que todo el bloque está en el buffer interno del controlador. A continuación, el controlador calcula la suma de verificación para comprobar que no ocurrieron errores de lectura, y luego causa una interrupción. Cuando el sistema operativo comienza a ejecutarse, puede leer el bloque del disco del buffer del controlador byte por byte o palabra por palabra, ejecutando un ciclo, leyéndose en cada iteración un byte o una palabra de un registro del controlador y almacenándose en la memoria.

Naturalmente, un ciclo de la CPU programado para leer los bytes del controlador uno por uno desperdicia tiempo de CPU. Se inventó el DMA para liberar a la CPU de este trabajo de bajo nivel. Cuando se usa DMA, la CPU proporciona al controlador dos elementos de información, además de la dirección en disco del bloque: la dirección de memoria donde debe colocarse el bloque y el número de bytes que deben transferirse, como se muestra en la Fig. 3-3.

Una vez que el controlador ha leído todo el bloque del dispositivo, lo ha colocado en su buffer y ha calculado la suma de verificación, copia el primer byte o palabra en la memoria principal en la dirección especificada por la dirección de memoria de DMA. Luego, el controlador incrementa la dirección de DMA y decrementa la cuenta de DMA en el número de bytes que se acaban de

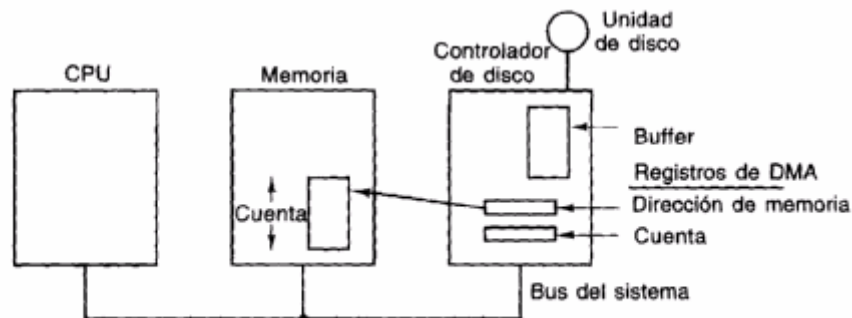


Figura 3-3. Una transferencia DMA es realizada totalmente por el controlador.

transferir. Este proceso se repite hasta que la cuenta de DMA es cero, y en ese momento el controlador causa una interrupción. Cuando el sistema operativo inicia, no tiene que copiar el bloque en la memoria; ya está ahí.

Tal vez usted se esté preguntando por qué el controlador no almacena los bytes en la memoria principal tan pronto como los recibe del disco. En otras palabras, ¿por qué necesita un buffer interno? La razón es que una vez que se ha iniciado una transferencia de disco, los bits siguen llegando del disco a velocidad constante, sea que el controlador esté listo o no para recibirlos. Si el controlador tratara de escribir los datos directamente en la memoria, tendría que hacerlo a través del bus del sistema para cada palabra transferida. Si el bus estuviera ocupado porque otro dispositivo lo está usando, el controlador tendría que esperar. Si la siguiente palabra del disco llegara antes de que la anterior se almacenara en la memoria, el controlador tendría que ponerla en algún lado. Si el bus estuviera muy ocupado, el controlador podría tener que almacenar una buena cantidad de palabras y también realizar un gran número de tareas administrativas. Si el bloque se guarda en un buffer interno, no se necesitará el bus en tanto no se inicie el DMA, y el diseño del controlador será mucho más sencillo porque la transferencia DMA a memoria no depende críticamente del tiempo. (De hecho, algunos controladores viejos sí transferían directamente a memoria con un mínimo de almacenamiento intermedio interno, pero cuando el bus estaba muy ocupado a veces era necesario terminar una transferencia con un error de desbordamiento.)

El proceso con almacenamiento intermedio de dos pasos que acabamos de describir tiene implicaciones importantes para el rendimiento de E/S. Mientras los datos están siendo transferidos del controlador a la memoria, sea por la CPU o por el controlador, el siguiente sector estará pasando bajo la cabeza del disco y los bits estarán llegando al controlador. Los controladores sencillos simplemente no pueden efectuar entrada y salida al mismo tiempo, de modo que mientras se está realizando una transferencia a la memoria el sector/que pasa bajo la cabeza del disco se pierde.

En consecuencia, el controlador sólo puede—leer bloques de manera intercalada, es decir, uno sí y uno no, de modo que la lectura de toda una pista requiere dos rotaciones completas, una para los bloques pares y otra para los impares. Si el tiempo que toma transferir un bloque del controlador a la memoria por el bus es más largo que el que toma leer un bloque del disco, puede ser necesario leer un bloque y luego saltarse dos (o más) bloques.

Saltarse bloques para dar al controlador tiempo de transferir los datos a la memoria se denomina intercalación. Cuando se da formato al disco, los bloques se numeran teniendo en cuenta el

factor de intercalación. En la Fig. 3-4(a) vemos un disco con ocho bloques por pista y cero intercalación. En la Fig. 3-4(b) vemos el mismo disco con intercalación sencilla. En la Fig. 3-4(c) ilustra la intercalación doble.

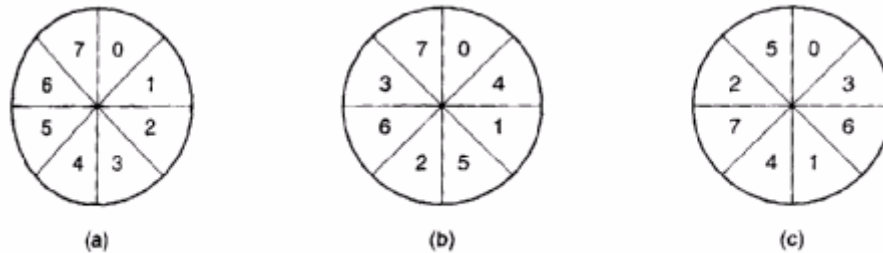


Figura 3-4. (a) Sin intercalación. (b) Intercalación sencilla. (c) Intercalación doble.

El objetivo de numerar los bloques de este modo es permitir que el sistema operativo lea bloques numerados consecutivamente y aun así logre la máxima velocidad de que el hardware es capaz. Si los bloques se numeraran como en la Fig. 3-4(a) pero el controlador sólo pudiera leer bloques alternados, un sistema operativo que se encargara de distribuir un archivo de ocho bloques en bloques de disco consecutivos requeriría ocho rotaciones de disco para leer los bloques 0 al 7 en orden. (Desde luego, si el sistema operativo se diera cuenta del problema y repartiera los bloques recibidos de forma diferente en la memoria, podría resolver el problema en software, pero es mejor dejar que el controlador se preocupe por la intercalación.)

No todas las computadoras usan DMA. El argumento en su contra es que en muchos casos la CPU principal es mucho más rápida que el controlador de DMA y puede realizar el trabajo en mucho menos tiempo (cuando el factor limitante no es la rapidez del dispositivo de E/S). Si la CPU (rápida) no tiene otra cosa que hacer, obligarla a esperar hasta que el controlador de DMA (lento) termine no tiene sentido. Además, si se omite el controlador de DMA y se deja que la CPU realice todo el trabajo, se ahorra algo de dinero.

3.2 PRINCIPIOS DEL SOFTWARE DE E/S

Dejemos ahora el hardware y examinemos la forma como está estructurado el software de E/S. Los objetivos generales del software de E/S son fáciles de plantear. La idea básica es organizar el software como una serie de capas, y que las inferiores oculten las peculiaridades del hardware para que las capas superiores no las vean. Las capas superiores se ocuparán de presentar una interfaz bonita, aseada y regular a los usuarios. En las siguientes secciones estudiaremos estos objetivos y la forma de lograrlos.

3.2.1 Objetivos del software de E/S

Un concepto clave del diseño de software de E/S se conoce como independencia del dispositivo. Esto significa que debe ser posible escribir programas que puedan leer archivos de un disquete,

un disco duro o un CD-ROM sin tener que modificar los programas para cada tipo de dispositivo distinto. Debemos poder teclear un comando como

```
sort <entrada >salida
```

y hacer que funcione con entradas provenientes de un disco flexible, un disco duro o el teclado, y enviando las salidas al disco flexible, el disco duro o incluso la pantalla. Es responsabilidad del sistema operativo resolver los problemas causados por el hecho de que estos dispositivos en realidad son distintos y requieren controladores en software muy distintos para escribir los datos en el dispositivo de salida.

Algo muy relacionado con la independencia del dispositivo es el objetivo de nombres uniformes. El nombre de un archivo o un dispositivo debe ser simplemente una cadena y un entero y no depender del dispositivo de forma alguna. En UNIX, todos los discos se pueden integrar en la jerarquía del sistema de archivos de formas arbitrarias, de modo que el usuario no necesita saber qué nombre corresponde a qué dispositivo. Por ejemplo, un disco flexible puede montarse sobre el directorio `/usr/ast/respaldo!` de modo que si copiamos un archivo en `/usr/ast/respaldo/lunes` lo estaremos copiando en el disquete. Así, todos los archivos y dispositivos se direccionan de la misma forma: con un nombre de trayectoria.

Otro aspecto importante del software de E/S es el manejo de errores. En general, los errores deben manejarse tan cerca del hardware como sea posible. Si el controlador descubre un error de lectura, debe tratar de corregirlo él mismo, si puede. Si no puede, el controlador en software debería manejarlo, tal vez tratando simplemente de leer el bloque otra vez. Muchos errores son transitorios, como los errores de lectura causados por partículas de polvo en la cabeza de lectura, y desaparecen si la operación se repite. Sólo si las capas inferiores son incapaces de resolver el problema se deberá informar de él a las capas superiores. En muchos casos, la recuperación de errores puede efectuarse de manera transparente en un nivel bajo sin que los niveles superiores se enteren siquiera de que ocurrió un error.

Otro aspecto clave es si las transferencias son síncronas (por bloqueo) o asíncronas (controladas por interrupciones). En general, la E/S física es asíncrona: la CPU inicia la transferencia y se dedica a otra cosa hasta que llega la interrupción. Los programas de usuario son mucho más fáciles de escribir si las operaciones de E/S provocan bloqueos: después de un comando READ el programa se suspende automáticamente hasta que hay datos disponibles en el buffer. Le corresponde al sistema operativo hacer que las operaciones que en realidad son controladas por interrupciones parezcan controladas por bloqueo a los programas de usuario.

El último concepto que veremos aquí es el de dispositivos de uso exclusivo y no exclusivo. Algunos dispositivos de E/S, como los discos, pueden ser utilizados por muchos usuarios al mismo tiempo. No hay problemas si varios usuarios tienen archivos abiertos en el mismo disco al mismo tiempo. Otros dispositivos, como las unidades de cinta, tienen que estar dedicados a un solo usuario hasta que éste haya terminado. Luego, otro usuario puede disponer de la unidad de cinta. Si dos o más usuarios escriben bloques entremezclados al azar en la misma cinta, se generará el caos. La introducción de dispositivos de uso exclusivo (no compartidos) da lugar a diversos problemas. Una vez más, el sistema operativo debe poder manejar dispositivos tanto compartidos como de uso exclusivo de manera tal que se eviten problemas.

Estos objetivos se pueden lograr de una forma lógica y eficiente estructurando el software de E/S en cuatro capas:

1. Manejadores de interrupciones (capa inferior)
2. Controladores de dispositivos en software.
3. Software del sistema operativo independiente del software.
4. Software de usuario (capa superior).

Estas cuatro capas son (no por coincidencia) las mismas que vimos en la Fig. 2-26. En las siguientes secciones las examinaremos una por una, comenzando por abajo. En este capítulo haremos hincapié en los controladores de dispositivos (capa 2), pero resumiremos el resto del software de E/S para mostrar cómo se relacionan las diferentes piezas del sistema de E/S.

3.2.2 Manejadores de interrupciones

Las interrupciones son desagradables pero inevitables, y deben ocultarse en las profundidades del sistema operativo, con el fin de reducir al mínimo las partes del sistema que tienen conocimiento de ellas. La mejor forma de ocultarlas es hacer que cada proceso inicie un bloqueo de operación de WS hasta que la E/S se haya llevado a cabo y la interrupción ocurra. El proceso puede bloquearse ejecutando un DOWN con un semáforo, un WA con una variable de condición o un RECEIVE con un mensaje, por ejemplo.

Cuando sucede la interrupción, el procedimiento de interrupciones hará lo que tenga que hacer para desbloquear el proceso que la originó. En algunos sistemas se ejecutará un uP con un semáforo; en otros, se ejecutará un SIGNAL con una variable de condición en un monitor. En otros más, se enviará un mensaje al proceso bloqueado. En todos los casos, el efecto neto de la interrupción será que un proceso que estaba bloqueado está ahora en condiciones de ejecutarse.

3.2.3 Controladores de dispositivos

Todo el código dependiente del dispositivo se coloca en los controladores de dispositivo. Cada controlador maneja un tipo de dispositivo o, cuando más, una clase de dispositivos similares. Por ejemplo, podría ser aconsejable tener un solo controlador de terminal, aun si el sistema maneja terminales de distintas marcas, todas con pequeñas diferencias. Por otro lado, una terminal mecánica tonta que produce salidas impresas y una terminal inteligente con gráficos de mapa de bits y ratón son tan distintas que es preciso usar diferentes controlados en software.

En una sección anterior del capítulo explicamos lo que hacen los controladores de dispositivos en hardware. Vimos que cada controlador tiene uno o más registros de dispositivo que sirven para aceptar comandos. Los controladores en software emiten estos comandos y verifican que se ejecuten correctamente. Así, el controlador de disco en software es la única parte del sistema operativo que sabe cuántos registros tiene ese controlador de disco en hardware y para qué sirven. Sólo él sabe algo de sectores, pistas, cilindros, cabezas, movimiento del brazo, factores de

intercalación, motores, tiempos de asentamiento de la cabeza y todos los demás aspectos mecánicos del funcionamiento correcto del disco.

En términos generales, la tarea de un controlador de dispositivo en software es aceptar peticiones abstractas del software independiente del dispositivo que está arriba de él y ver que dichas peticiones sean atendidas. Una petición típica sería leer el bloque *n*. Si el controlador está ocioso en el momento en que llega una petición, comenzará a atenderla de inmediato, pero si ya está ocupado con otra petición, normalmente colocará la nueva petición en una cola de peticiones pendientes que se atenderán tan pronto como sea posible.

El primer paso para atender realmente una petición de E/S, digamos para un disco, es traducirla de términos abstractos a concretos. En el caso de un controlador de disco, esto implica calcular en qué parte del disco está realmente el bloque solicitado, verificar si el motor de la unidad está funcionando, determinar si el brazo está colocado en el cilindro apropiado, etc. En pocas palabras, el controlador en software debe decidir qué operaciones del controlador son necesarias en el hardware y en qué orden.

Una vez que el controlador en software ha decidido qué comandos debe emitir al controlador en hardware, comenzará a emitirlos escribiendo en los registros de dispositivo de este último. Algunos controladores en hardware sólo pueden manejar un comando a la vez; otros están dispuestos a aceptar una lista enlazada de comandos, que luego ejecutan por su cuenta sin más ayuda del sistema operativo.

Una vez que se ha emitido el comando o comandos, se presentará una de dos situaciones. En muchos casos el controlador en software debe esperar hasta que el controlador en hardware realice cierto trabajo, así que se bloquea hasta que llega la interrupción para desbloquearlo. En otros casos, sin embargo, la operación se lleva a cabo sin dilación, y el controlador no necesita bloquearse. Como ejemplo de una situación de este tipo, el recorrido en la pantalla en algunas terminales sólo requiere la escritura de unos cuantos bytes en los registros del controlador en hardware. No se requiere movimiento mecánico, así que toda la operación puede llevarse a cabo en unos cuantos microsegundos.

En el primer caso, el controlador bloqueado será despertado por la interrupción; en el segundo, nunca se dormirá. De cualquier manera, una vez que se ha efectuado la operación el controlador debe determinar si no ocurrieron errores. Si todo está bien, el controlador puede tener datos que pasar al software independiente del dispositivo (p. ej., el bloque que acaba de leerse). Por último, el controlador devuelve información de estado para informar de errores a su invocador. Si hay otras peticiones en cola, puede seleccionarse e inicia una de ellas. Si no hay nada en la cola, el controlador se bloquea esperando la siguiente petición.

3.2.4 Software de E/S independiente del dispositivo

Aunque una parte del software de E/S es específica para cada dispositivo, una fracción considerable es independiente de él. La frontera exacta entre los controladores de dispositivos y el software independiente del dispositivo depende del sistema, porque algunas funciones que podrían realizarse de forma independiente del dispositivo podrían efectuarse realmente en los controladores en software, por razones de eficiencia o de otro tipo. Las funciones que se muestran en la Fig. 3-5 por

lo regular se realizan en el software independiente del dispositivo. En MINIX, casi todo el software independiente del dispositivo forma parte del sistema de archivos, en la capa 3 (Fig. 2-26). Aunque estudiaremos el sistema de archivos en el capítulo 5, daremos aquí un vistazo al software independiente del dispositivo, a fin de tener un panorama más completo de la E/S y poder explicar mejor donde encajan los controladores.

Interfaz uniforme para los controladores de dispositivos
Nombres de dispositivos
Protección de dispositivos
Tamaño de bloque independiente del dispositivo
Almacenamiento intermedio
Asignación de almacenamiento en dispositivos por bloques
Asignación y liberación de dispositivos dedicados
Informe de errores

Figura 3-5. Funciones del software de E/S independiente del dispositivo.

La función básica del software independiente del dispositivo es realizar las funciones de E/S comunes a todos los dispositivos y presentar una interfaz uniforme al software de nivel de usuario.

Un aspecto importante de cualquier sistema operativo es cómo se nombran los objetos tales como archivos y dispositivos de E/S. El software independiente del dispositivo se encarga de establecer la correspondencia entre los nombres simbólicos de dispositivo y el controlador apropiado. En UNIX un nombre de dispositivo, como `/dev/tty00`, especifica de manera única el nodo-*i* de un archivo especial, y este nodo-*i* contiene el número principal del dispositivo, que sirve para localizar el controlador apropiado. El nodo-*i* también contiene el número secundario del dispositivo, que se pasa como parámetro al controlador a fin de especificar la unidad que se leerá o escribirá.

Algo muy relacionado con los nombres es la protección. ¿Cómo impide el sistema que los usuarios puedan acceder a dispositivos que no están facultados para ello? En la mayor parte de los Sistemas de computadora personal no hay protección. Cualquier proceso puede hacer lo que quiera. En las macrocomputadoras, el acceso a los dispositivos de E/S por parte de los procesos de usuario está estrictamente prohibido. En UNIX se utiliza un esquema más flexible. Los archivos especiales que corresponden a los dispositivos de FIS se protegen con los bits `rxw` usuales. Así, el administrador del sistema puede establecer los permisos adecuados para cada dispositivo.

Diferentes discos pueden tener tamaños de sectores distintos. Corr, al software independiente del dispositivo ocultar este hecho y proveer un tamaño de bloque uniforme a las capas superiores, por ejemplo, tratando varios sectores como un solo bloque lógico. De esta forma, las capas superiores sólo manejan dispositivos abstractos que siempre usan el mismo tamaño de bloque lógico, sea cual sea el tamaño de los sectores físicos. De forma similar, algunos dispositivos por caracteres suministran sus datos byte por byte (como los módems) mientras que otros los suministran en unidades más grandes (como las interfaces de red). Estas diferencias también deben ocultarse.

El almacenamiento intermedio también es una cuestión importante, tanto para los dispositivos por bloques como para los de caracteres. En el primer caso, el hardware generalmente insiste en

leer y escribir bloques completos a la vez, pero los procesos de usuario están en libertad de leer y escribir en unidades arbitrarias. Si un proceso de usuario escribe medio bloque, el sistema operativo normalmente guardará los datos internamente hasta que se escriba el resto de los datos, y en ese momento se podrá grabar el bloque en el disco. En el caso de los dispositivos por caracteres, los usuarios podrían escribir los datos al sistema con mayor velocidad que aquella a la que pueden salir, lo que requeriría almacenamiento intermedio (buffers). De igual manera, la entrada del teclado que llega antes de que sea necesaria también requiere buffers.

Cuando se crea un archivo y se llena con datos, es preciso asignar nuevos bloques de disco al archivo. Para poder realizar esta asignación, el sistema operativo necesita una lista o un mapa de bits de los bloques libres del disco, pero el algoritmo para localizar un bloque libre es independiente del dispositivo y se puede efectuar en un nivel más alto que el del manejador.

Algunos dispositivos, como las grabadoras de CD-ROM, sólo pueden ser utilizados por un solo proceso en determinado momento. Corresponde al sistema operativo examinar las peticiones de uso del dispositivo y aceptarlas o rechazarlas, dependiendo de si el dispositivo solicitado está disponible o no. Una forma sencilla de manejar estas peticiones es exigir a los procesos que ejecuten instrucciones OPEN en los archivos especiales de los dispositivos de forma directa. Si el dispositivo no está disponible, el OPEN fallará. Al cerrar tal dispositivo de uso exclusivo, éste se liberaría.

El manejo de errores generalmente lo realizan los controladores. La mayor parte de los errores son altamente dependientes del dispositivo, de modo que sólo el controlador sabe qué debe hacerse (p. ej., reintentar, ignorar, situación de pánico). Un error típico es el causado por un bloque de disco que se dañó y ya no puede leerse. Después de que el controlador ha tratado de leer el bloque cierto número de veces, se da por vencido e informa de ello al software independiente del dispositivo. La forma como el error se trata de aquí en adelante es independiente del dispositivo. Si el error ocurrió mientras se estaba leyendo un archivo de usuario, puede ser suficiente informar del error al invocador. Sin embargo, si el error ocurrió durante la lectura de una estructura de datos crítica del sistema, como el bloque que contiene el mapa de bits que indica cuáles bloques están libres, puede ser que el sistema operativo no tenga más opción que imprimir un mensaje de error y terminar.

3.2.5 Software de E/S de espacio de usuario

Aunque la mayor parte del software de E/S está dentro del sistema operativo, una pequeña parte de él consiste en bibliotecas enlazadas a los programas `sl/usuario`, e incluso en programas completos que se ejecutan fuera del kernel. Las llamadas entre ellas las E/S, normalmente son efectuadas por procedimientos de biblioteca. Cuando un programa en C contiene la llamada

```
count = write(fd, buffer, nbytes);
```

el procedimiento de biblioteca `write` se enlazará al programa y estará contenido en el programa binario presente en la memoria en el momento de la ejecución. La colección de todos estos procedimientos de biblioteca evidentemente forma parte del sistema de E/S.

Si bien estos procedimientos no hacen mucho más que colocar sus parámetros en el lugar apropiado para la llamada al sistema, hay otros procedimientos de E/S que sí realizan trabajo de

verdad. En particular, el formateo de entradas y salidas se realiza mediante procedimientos de biblioteca. Un ejemplo de C es `printf`, que toma una cadena de formato y posiblemente algunas variables como entrada, construye una cadena ASCII y luego invoca `WRITE` para enviar la cadena a la salida. Un ejemplo de un procedimiento similar para la entrada lo constituye `scanf` que lee entradas y las almacena en variables descritas en una cadena de formato que tiene la misma sintaxis que `printf`. La biblioteca de E/S estándar contiene varios procedimientos que implican E/S, y todos se ejecutan como parte de programas de usuario.

No todo el software de E/S de nivel de usuario consiste en procedimientos de biblioteca. Otra categoría importante es el sistema de spool. El uso de spool es una forma de manejar los dispositivos de E/S de uso exclusivo en un sistema multiprogramado. Consideremos un dispositivo spool típico: una impresora. Aunque desde el punto de vista técnico sería fácil dejar que cualquier proceso de usuario abriera el archivo especial por caracteres de la impresora, podría suceder que un proceso lo abriera y luego pasara horas sin hacer nada. Ningún otro proceso podría imprimir nada.

En vez de ello, lo que se hace es crear un proceso especial, llamado genéricamente demonio, y un directorio especial, llamado directorio de spool. Si un proceso quiere imprimir un archivo, primero genera el archivo completo que va a imprimir y lo coloca en el directorio de spool. Corresponde al demonio, que es el único proceso que tiene permiso de usar el archivo especial de la impresora, escribir los archivos en el directorio. Al proteger el archivo especial contra el uso directo por parte de los usuarios, se elimina el problema de que alguien lo mantenga abierto durante un tiempo innecesariamente largo.

El spool no se usa sólo para impresoras; también se usa en otras situaciones. Por ejemplo, es común usar un demonio de red para transferir archivos por una red. Si un usuario desea enviar un archivo a algún lado, lo coloca en un directorio de spool de red. Más adelante, el demonio de red lo toma de ahí y lo transmite. Una aplicación especial de la transmisión de archivos por spool es el sistema de correo electrónico de Internet. Esta red consiste en millones de máquinas en todo el mundo que se comunican empleando muchas redes de computadoras. Si usted desea enviar correo a alguien, invoca un programa como `send`, que acepta la carta que se desea enviar y la deposita en un directorio de spool para ser transmitida posteriormente. Todo el sistema de correo se ejecuta fuera del sistema operativo.

En la Fig. 3-6 se resume el sistema de E/S, con todas las capas y las funciones principales de cada capa. Comenzando por abajo, las capas son el hardware, los manejadores de interrupciones, los controladores de dispositivos, el software independiente del dispositivo y por último los procesos de usuario.

Las flechas de la Fig. 3-6 indican el flujo de control. Por ejemplo, cuando un proceso de usuario trata de leer un bloque de un archivo, se invoca el sistema operativo y ejecuta la llamada. El software independiente del dispositivo busca, por ejemplo, en el caché de bloques. Si el bloque que se necesita no está ahí, ese software invoca el controlador de dispositivo para que emita la petición al hardware. A continuación el proceso se bloquea hasta que se lleva a cabo la operación de disco.

Cuando el disco termina, el hardware genera una interrupción. Se ejecuta el manejador de interrupciones para descubrir qué ha sucedido, es decir, cuál dispositivo debe atenderse en este momento. El manejador extrae entonces el estado del dispositivo y despierta al proceso dormido para que finalice la petición de E/S y permita al proceso de usuario continuar.

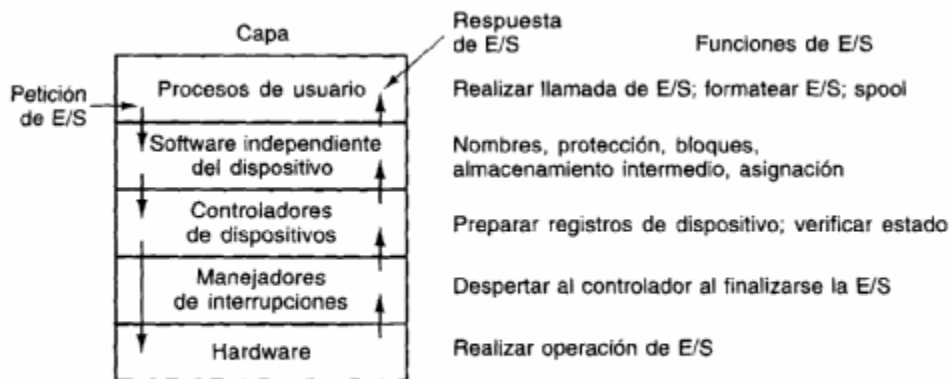


Figura 3-6. Capas del sistema de E/S y funciones principales de cada capa.

3.3 BLOQUEO MUTUO

Los sistemas de cómputo tienen muchos recursos que sólo pueden ser utilizados por un proceso a la vez. Ejemplos comunes de ellos son los graficadores de cama plana, los lectores de CD ROM, las grabadoras de CD-ROM, los sistemas de respaldo en cinta DAT de 8 mm, los formadores de imágenes y las ranuras de la tabla de procesos del sistema. Tener dos procesos escribiendo simultáneamente en una impresora produce basura. Tener dos procesos usando la misma ranura de la tabla de procesos probablemente causaría una caída del sistema. Por ello, todos los sistemas operativos tienen la capacidad de conceder (temporalmente) a un proceso acceso exclusivo a ciertos recursos.

En muchas aplicaciones, un proceso necesita acceso exclusivo no a un recurso, sino a varios. Consideremos, por ejemplo, una compañía de marketing que se especializa en preparar mapas demográficos detallados en un graficador de cama plana de 1 m de ancho. La información demográfica proviene de los CD-ROM que contienen datos censuales y de otro tipo. Supongamos que el proceso A solicita la unidad de CD-ROM y la obtiene. Un momento después, el proceso B solicita el graficador y también lo obtiene. Ahora el proceso A solicita el graficador y se bloquea esperándolo. Por último, el proceso B solicita la unidad de CD-ROM y también se bloquea. En este punto ambos procesos quedan bloqueados y permanecen así eternamente. Esta situación se denomina bloqueo mutuo. No conviene tener bloqueo un sistema.

Los bloqueos mutuos pueden ocurrir en muchas situaciones además de la petición de dispositivos de E/S de uso exclusivo. Por ejemplo, en un sistema de base de datos, un programa podría tener que poner un candado a varios registros que está usando, a fin de evitar condiciones de competencia. Si el proceso A asegura el registro R1 y el proceso B asegura el registro R2, y luego cada proceso trata de asegurar el registro del otro, también tendremos un bloqueo mutuo. Por tanto, los bloqueos mutuos pueden ocurrir con recursos de hardware o de software.

En esta sección examinaremos los bloqueos mutuos con mayor detenimiento para ver cómo surgen y cómo pueden prevenirse o evitarse. Como ejemplos, hablaremos de adquirir dispositivos

físicos como unidades de cinta, unidades de CD-ROM y graficadores, porque son fáciles de visualizar, pero los principios y algoritmos se aplican igualmente bien a otros tipos de bloqueos mutuos.

3.3.1 Recursos

Los bloqueos mutuos pueden ocurrir cuando se otorga a los procesos acceso exclusivo a dispositivos, archivos, etc. A fin de hacer la explicación de los bloqueos mutuos lo más general posible, nos referiremos a los objetos otorgados como recursos. Un recurso puede ser un dispositivo de hardware (p. ej., una unidad de cinta) o un elemento de información (p. ej., un registro con candado en una base de datos). Una computadora normalmente tiene muchos recursos distintos que se pueden adquirir. Para algunos recursos pueden estar disponibles varios ejemplares idénticos, como tres unidades de cinta. Cuando están disponibles varias copias de un recurso, cualquiera de ellas puede usarse para satisfacer cualquier petición del usuario por ese recurso. En pocas palabras, un recurso es cualquier cosa que sólo puede ser usada por un proceso en un instante dado.

Los recursos son de dos tipos: expropiables y no expropiables. Un recurso expropiable es uno que se puede arrebatar al proceso que lo tiene sin que haya efectos adversos. La memoria es un ejemplo de recurso expropiable. Consideremos, por ejemplo, un sistema con 512K de memoria de usuario, una impresora y dos procesos de 512K que quieren imprimir algo. El proceso A solicita y obtiene la impresora, y comienza a calcular los valores que va a imprimir, pero antes de que haya terminado el cálculo excede su cuanto de tiempo y es intercambiado a disco.

Ahora se ejecuta el proceso B e intenta, sin éxito, adquirir la impresora. Aquí tenemos una situación de bloqueo mutuo en potencia, porque A tiene la impresora y B tiene la memoria, y ninguno puede proceder sin el recurso que el otro tiene. Por fortuna, es posible quitarle la memoria a B (expropiarla) intercambiando B a disco e intercambiando A a la memoria. Ahora A puede ejecutarse, imprimir, y por último liberar la impresora. No hay bloqueo mutuo.

En contraste, un recurso no expropiable no puede quitársele a su poseedor actual sin hacer que el cómputo falle. Si un proceso ya comenzó a imprimir salidas y se le quita la impresora para dársela a otro proceso, se obtendrá basura como salida. Las impresoras no son expropiables.

En general, en los bloqueos mutuos intervienen recursos no expropiables. Los bloqueos mutuos en potencia en los que intervienen recursos expropiables casi siempre pueden resolverse reasignando los recursos de un proceso a otro. Por tanto, nuestro tratamiento se centrará en los recursos no apropiables.

La secuencia de sucesos que se requiere para usar un recurso es:

1. Solicitar el recurso.
2. Usar el recurso.
3. Liberar el recurso.

Si el recurso no está disponible cuando se solicita, el proceso que realiza la solicitud tiene que esperar. En algunos sistemas operativos, el proceso se bloquea automáticamente cuando una petición

de recurso falla, y se le despierta cuando el recurso está disponible. En otros sistemas, la petición falla con un código de error, y toca al proceso invocador esperar un poco e intentarlo de nuevo.

3.3.2 Principios del bloqueo mutuo

El bloqueo mutuo puede definirse formalmente como sigue:

Un conjunto de procesos está en bloqueo mutuo si cada proceso del conjunto está esperando un evento que sólo otro proceso del conjunto puede causar.

Puesto que todos los procesos están esperando, ninguno de ellos puede causar ninguno de los eventos que podrían despertar a cualquiera de los demás miembros del conjunto, y todos los procesos continúan esperando indefinidamente.

En la mayor parte de los casos, el evento que cada proceso está esperando es la liberación de algún recurso que actualmente está en poder de otro miembro del conjunto. Dicho de otro modo, cada miembro del conjunto de procesos mutuamente bloqueado está esperando un recurso que está en poder de otro proceso en bloqueo. Ninguno de los procesos puede ejecutarse, ninguno puede liberar ningún recurso, y ninguno puede ser despertado. Ni el número de los procesos ni el número y tipo de los recursos poseídos y solicitados son importantes.

Condiciones para el bloqueo mutuo

Coffman et al. (1971) demostraron que deben cumplirse cuatro condiciones para que haya un bloqueo mutuo:

1. Condición de exclusión mutua. Cada recurso está asignado únicamente a un solo proceso o está disponible.
2. Condición de retener y esperar. Los procesos que actualmente tienen recursos que les fueron otorgados previamente pueden solicitar nuevos recursos.
3. Condición de no expropiación. No es posible quitarle por la fuerza a un proceso los recursos que le fueron otorgados previamente. El proceso que los tiene debe liberarlos explícitamente.
4. Condición de espera circular. Debe haber una cadena circular de dos o más procesos, cada uno de los cuales está esperando un recurso retenido por el siguiente miembro de la cadena.

Deben estar presentes estas cuatro condiciones para que ocurra un bloqueo mutuo. Si una o más de estas condiciones está ausente, no puede haber bloqueo mutuo.

HoIt (1972) mostró cómo pueden modelarse estas cuatro condiciones usando grafos dirigidos. Los grafos tienen dos clases de nodos: procesos, que se indican con círculos, y recursos, que se indican con cuadrados. Un arco que va de un nodo de recurso (cuadrado) a uno de proceso (círculo) indica que el recurso fue solicitado previamente por el proceso, le fue concedido, y actualmente está en su poder. En la Fig. 3-7(a) el recurso R está asignado actualmente al proceso A.

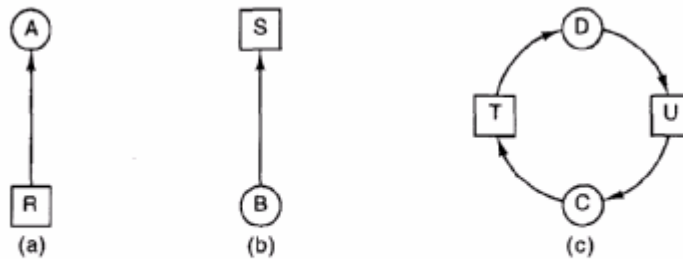


Figura 3-7. Grafos de asignación de recursos. (a) Retención de un recurso. (b) Petición de un recurso. (c) Bloqueo mutuo.

Un arco de un proceso a un recurso indica que el proceso está bloqueado esperando ese recurso. En la Fig. 3-7(b) el proceso B está esperando el recurso S. En la Fig. 3-7(c) vemos un bloqueo mutuo: el proceso C está esperando el recurso T, que actualmente está en poder del proceso D, El proceso D no va a liberar el recurso T porque está esperando el recurso U, que está en poder de C. Ambos procesos esperarán eternamente. Un ciclo en el grafo implica que hay un bloqueo mutuo en el que intervienen los procesos y recursos del ciclo. En este ejemplo, el ciclo es C-T-D-U-C.

Examinemos ahora un ejemplo de cómo pueden usarse los grafos de recursos. Imagine que tenemos tres procesos, A, B y C, y tres recursos, R, S y T. Las peticiones y liberaciones de los tres procesos se muestran en la Fig. 3-8(a)-(c). El sistema operativo está en libertad de ejecutar cualquier proceso no bloqueado en cualquier instante, de modo que podría decidir ejecutar A hasta que A terminara todo su trabajo, luego B hasta su finalización y por último C.

Este ordenamiento no da lugar a ningún bloqueo mutuo (porque no hay competencia por los recursos) pero tampoco tiene paralelismo. Además de solicitar y liberar recursos, los procesos calculan y realizan LIS. Cuando los procesos se ejecutan secuencialmente, no existe la posibilidad de que mientras un proceso está esperando LIS el otro pueda usar la CPU. Por tanto, ejecutar los procesos en forma estrictamente secuencial podría no ser óptimo. Por otro lado, si ninguno de los procesos realiza LIS, el algoritmo del primer trabajo más corto es mejor que el round robin, así que en algunas circunstancias lo mejor podría ser ejecutar todos los procesos secuencialmente.

Supongamos ahora que los procesos realizan tanto LIS como cálculos, de modo que el round robin es un algoritmo de planificación razonable. Las peticiones de recursos podrían presentarse en el orden que se indica en la Fig. 3-8(d). Si esas peticiones se llevan a cabo en ese orden, los

grafos de recursos resultantes son los que se muestran en la Fig. 3-8(e)-Q). Después de hacerse la petición 4, A se bloquea esperando S, como se aprecia en la Hg. 3-8(h). En los dos pasos B y C siguientes también se bloquean, dando lugar en última instancia a un ciclo y al bloqueo mutuo de la Fig. 3-8(j).

Sin embargo, como ya hemos mencionado, el sistema operativo no está obligado a ejecutar los procesos en un orden específico. En particular, si la concesión de una petición determinada pudiera dar pie a un bloqueo mutuo, el sistema operativo podrá simplemente suspender el proceso sin dar respuesta a la petición (o sea, no planificar el proceso) hasta que pueda hacerlo sin peligro. En la Hg. 3-8, si el sistema operativo tiene conocimiento de un bloqueo mutuo inminente, podría suspender B en lugar de otorgarle S. Al ejecutarse sólo A y C obtendríamos las peticiones y liberaciones de la Fig. 3-8(k) en lugar de las de la Hg. 3-8(d). Esta secuencia da lugar a los grafos de recursos de la Hg. 3-8(l)-(q), que no conducen a bloqueo mutuo.

Después del paso (q), ya se puede otorgar S a B porque A ya terminó y C tiene todo lo que necesita. Incluso si B llegara a bloquearse al solicitar T, no podría ocurrir un bloqueo mutuo. B simplemente esperará hasta que C termine.

Más adelante en el capítulo estudiaremos un algoritmo detallado para tomar decisiones de asignación que no conducen a bloqueos mutuos. Lo que es importante entender ahora es que los grafos de recursos son una herramienta que nos permite ver si una secuencia de petición./liberación dada conduce o no al bloqueo. Basta con indicar las peticiones y liberaciones paso por paso, determinando después de cada paso si el grafo contiene ciclos. Si los contiene, tenemos un bloqueo mutuo; si no, no hay bloqueo. Aunque nuestro tratamiento de los grafos de recursos corresponde al caso en que sólo hay un recurso de cada tipo, es posible generalizar los grafos para manejar múltiples recursos del mismo tipo (Holt, 1972).

En general, son cuatro las estrategias que se emplean para manejar el bloqueo mutuo:

1. Simplemente hacer caso omiso del problema.
2. Detección y recuperación.
3. Evitarlo de manera dinámica, mediante una asignación cuidadosa de los recursos.
4. Prevención, negando estructuralmente una de las cuatro condiciones necesarias. Examinaremos cada uno de estos métodos por turno en las siguientes cuatro secciones.

3.3.3 El algoritmo del avestruz

La estrategia más sencilla es el algoritmo del avestruz: meter la cabeza en la arena y pretender que el problema no existe. La gente reacciona a esta estrategia de diversas maneras. Los matemáticos la encuentran totalmente inaceptable y dicen que los bloqueos mutuos deben prevenirse a toda costa. Los ingenieros preguntan con qué frecuencia se espera que se presente el problema, qué tan seguido se cae el sistema por otras razones, y qué tan grave es un bloqueo mutuo. Si ocurren bloqueos mutuos una vez cada 50 años en promedio, pero las caídas del sistema debido a fallas de hardware, errores del compilador y defectos del sistema operativo ocurren una vez al mes, la mayoría de los ingenieros no estarían dispuestos a pagar un precio sustancial en términos de reducción del rendimiento o de la comodidad a fin de evitar los bloqueos mutuos.

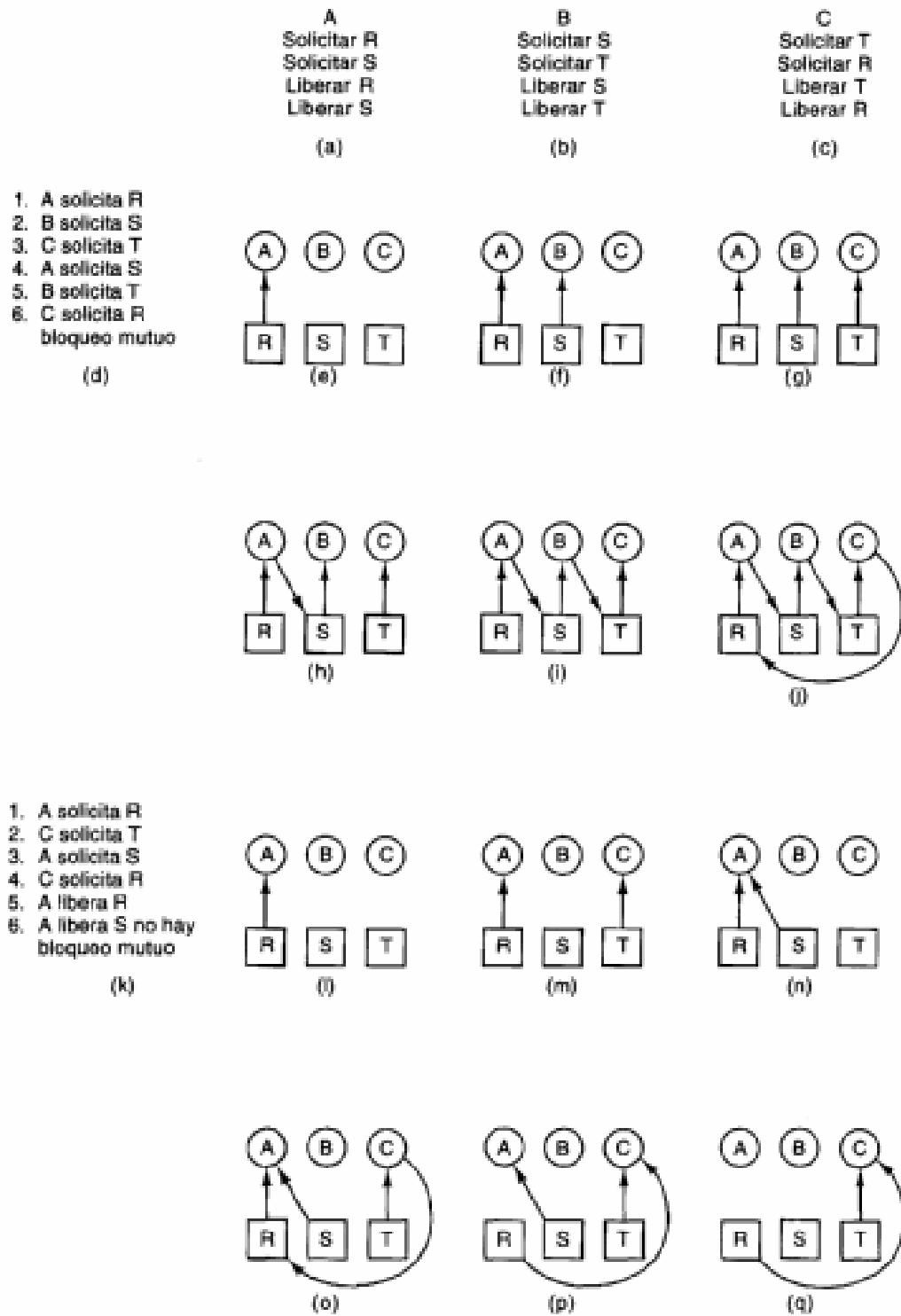


Figura 3-8. Ejemplo de cómo ocurre un bloqueo mutuo y cómo puede evitarse.

Para hacer este contraste más específico, UNIX (y MINIX) sufren potencialmente de bloqueos mutuos que ni siquiera se detectan, o mucho menos que se rompan automáticamente. El número total de procesos que hay en el sistema está determinado por el número de entradas de la tabla de procesos, así que las ranuras de la tabla de procesos son un recurso finito. Si un FORK falla porque la tabla está llena, una estrategia razonable para el programa que ejecuta el FORK sería esperar un tiempo aleatorio e intentarlo otra vez.

Supongamos ahora que un sistema UNIX tiene 100 ranuras para procesos. Se están ejecutando 10 programas, cada uno de los cuales necesita crear 12 (sub)procesos. Una vez que cada proceso ha creado 9 procesos, los 10 procesos originales y los 90 nuevos han agotado la tabla.

Ahora, cada uno de los 10 procesos originales se encuentra en un ciclo infinito en el que se bifurcan y fallan: bloqueo mutuo. La probabilidad de que esto suceda es pequeñísima, pero podría suceder. ¿Deberemos abandonar los procesos y la llamada FORK para eliminar el problema?

El número máximo de archivos abiertos está restringido de forma similar por el tamaño de la tabla de nodos-i, así que ocurre un problema similar cuando la tabla se llena. El espacio de intercambio (swapping) en el disco es otro recurso limitado. De hecho, casi todas las tablas del sistema operativo representan un recurso finito. ¿Debemos cancelar todos estos recursos porque podría suceder que una colección de n procesos solicitara un del total, y luego cada uno tratara de adquirir uno más?

La estrategia de UNIX consiste en hacer caso omiso del problema bajo el supuesto de que la mayoría de los usuarios preferirán un bloqueo mutuo ocasional en lugar de una regla que restrinja a todos los usuarios a un proceso, un archivo abierto, y una de cada cosa. Si los bloqueos mutuos pudieran eliminarse gratuitamente, no habría mucho que discutir. El problema es que el precio es alto, y principalmente consiste en poner restricciones molestas a los procesos, como veremos en breve. Así, enfrentamos un desagradable trueque entre comodidad y corrección, y mucha discusión acerca de qué es más importante.

3.3.4 Detección y recuperación

Una segunda técnica es la detección y recuperación. Cuando se usa esta técnica, el sistema no hace otra cosa que no sea vigilar las peticiones y liberaciones de recursos. Cada vez que un recurso se solicita o libera, se actualiza el grafo de recursos, y se determina si contiene algún ciclo. Si se encuentra uno, se termina uno de los procesos del ciclo. Si esto no rompe el bloqueo mutuo, se termina otro proceso, continuando así hasta romper el ciclo. Un método un tanto más burdo consiste en no mantener siquiera el grafo de recursos, y en vez de ello verificar periódicamente si hay procesos que hayan estado bloqueados continuamente durante más de, digamos, una hora. A continuación se terminan esos procesos.

La detección y recuperación es la estrategia que a menudo se usa en las macrocomputadoras, sobre todo los sistemas por lotes en los que terminar y luego reiniciar un proceso suele ser aceptable. Sin embargo, se debe tener cuidado de restaurar todos los archivos modificados a su estado original, y revertir todos los demás efectos secundarios que pudieran haber ocurrido.

3.3.5 Prevención del bloqueo mutuo

La tercera estrategia para manejar el bloqueo mutuo consiste en imponer restricciones apropiadas a los procesos de modo que el bloqueo mutuo sea estructuralmente imposible. Las cuatro condiciones planteadas por Coffman et al. (1971) señalan algunas posibles soluciones. Si podemos asegurar que al menos una de esas condiciones nunca se satisfaga, el bloqueo mutuo será imposible (Havender, 1968).

Ataquemos primero la condición de exclusión mutua. Si ningún recurso se asignara de manera exclusiva a un solo proceso, jamás tendríamos bloqueo mutuo. Sin embargo, es igualmente obvio que permitir a dos procesos escribir en la impresora al mismo tiempo conduciría al caos. Si colocamos en spool las salidas a la impresora, varios procesos podrán generar salidas al mismo tiempo. En este modelo, el único proceso que realmente solicita la impresora física es el demonio de la impresora. Puesto que el demonio nunca solicita otros recursos, podemos eliminar el bloqueo mutuo para la impresora.

Desafortunadamente, no todos los recursos se pueden manejar con spool (la tabla de procesos no se presta muy bien que digamos a ello). Además, la competencia misma por obtener espacio de disco para spool puede dar lugar al bloqueo. ¿Qué sucedería si dos procesos llenaran cada uno con sus salidas la mitad del espacio de spool disponible y ninguno terminara? Si el demonio se programara de modo que comenzara a imprimir antes de que toda la salida estuviera en spool, la impresora podría permanecer ociosa si un proceso de salida decidiera esperar varias horas después de la primera ráfaga de salida. Por esta razón, los demonios se programan de modo que sólo impriman si ya está disponible todo el archivo de salida. Ninguno de los procesos terminaría, y tendríamos un bloqueo mutuo por el disco.

La segunda de las condiciones planteadas por Coffman et al. se ve más prometedora. Si podemos evitar que los procesos que retienen recursos esperen para obtener más recursos, podremos eliminar los bloqueos mutuos. Una forma de lograr este objetivo es exigir que todos los procesos soliciten todos sus recursos antes de iniciar su ejecución. Si todo está disponible, se asignará al proceso todo lo que necesita y éste podrá ejecutarse hasta finalizar. Si uno o más recursos están ocupados, no se asignará nada y el proceso simplemente esperará.

Un problema inmediato de esta estrategia es que muchos procesos no saben cuántos recursos van a necesitar antes de iniciar su ejecución. Otro problema es que los recursos no se aprovechan de manera óptima. Tomemos como ejemplo un proceso que lee datos de una cinta de entrada, los analiza durante una hora, y luego escribe una cinta de salida y además grafica los resultados. Si todos los recursos se solicitaran por adelantado, el proceso mantendría inaccesibles la unidad de cinta de salida y el graficador durante una hora.

Una forma un poco distinta de romper la condición de retener y esperar es exigir que un proceso que solicita un recurso libere primero temporalmente todos los recursos que está reteniendo en ese momento. Sólo si la petición tiene éxito podrá el proceso recibir de vuelta los recursos originales.

Atacar la tercera condición (no expropiación) es aún menos prometedor que atacar la segunda. Si a un proceso se le asignó la impresora y apenas ha imprimido la mitad de sus salidas,

quitarle forzosamente la impresora porque un graficador que se necesita no está disponible daría lugar a un desastre.

Sólo queda una condición. Se puede eliminar la espera circular de varias formas. Una de ellas consiste sencillamente en tener una regla que diga que un proceso sólo tiene derecho a un solo recurso en un instante dado. Si el proceso necesita un segundo recurso, deberá liberar el primero. Para un proceso que necesita copiar un archivo enorme de una cinta a una impresora, esta restricción es inaceptable.

Otra forma de evitar la espera circular es crear una numeración global de todos los recursos, como se muestra en la Fig. 3-9(a). Ahora la regla es ésta: los procesos pueden solicitar recursos cuando quieran, pero todas las peticiones deben hacerse en orden numérico. Un proceso puede solicitar primero una impresora y después una unidad de cinta, pero no puede solicitar primero un graficador y luego una impresora.

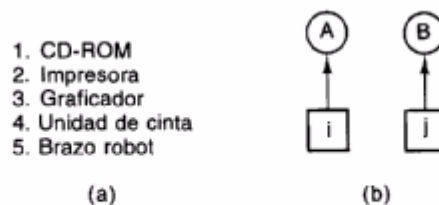


Figura 3-9. (a) Recursos ordenados numéricamente. (b) Un grafo de recursos.

Con esta regla, el grafo de asignación de recursos nunca puede tener ciclos. Veamos por qué esto se cumple para el caso de dos procesos [Fig. 3-9(b)]. Podemos tener un bloqueo mutuo sólo si A solicita el recurso j y B solicita el recurso i. Si suponemos que i y j son recursos distintos, tendrán diferente número. Si $i > j$, no se permitirá a A solicitar j. Si $i < j$, no se permitirá a B solicitar i. De cualquier manera, el bloqueo mutuo es imposible.

La misma lógica es válida para múltiples procesos. En todo instante, uno de los recursos asignados será el más alto. El proceso que esté reteniendo ese recurso nunca podrá solicitar uno que ya está asignado; o bien finalizará, o en el peor de los casos solicitará recursos con un número aún más alto, todos los cuales están disponibles. Tarde o temprano, ese proceso finalizará y liberará sus recursos. En este punto, algún otro recurso estará reteniendo el recurso más alto y también podrá finalizar. En pocas palabras, existe una situación en la que todos los procesos terminan, de modo que no hay bloqueo mutuo.

Una variación menor de este algoritmo consiste en omitir el requisito de que los recursos se adquieran en orden estrictamente creciente y sólo insistir en que ningún proceso solicite un recurso con un número menor que el del recurso que ya tiene en su poder. Si un proceso inicialmente solicita los recursos 9 y 10, y luego los libera, en efecto estará de nuevo iniciando desde el principio, y no habrá razón para prohibirle ahora que solicite el recurso 1.

Aunque el ordenamiento numérico de los recursos elimina el problema de los bloqueos mutuos, puede ser imposible encontrar un ordenamiento que satisfaga a todo mundo. Cuando los recursos incluyen ranuras de la tabla de procesos, espacio de disco para spool, registros de base de

datos con candado y otros recursos abstractos, el número de recursos potenciales y de posibles usos puede ser tan grande que ningún ordenamiento resulte práctico.

En la Fig. 3-10 se resumen las diferentes estrategias para prevenir los bloqueos mutuos.

Condición	Estrategia
Exclusión mutua	Poner todo en spool
Retener y esperar	Solicitar inicialmente todos los recursos
No expropiación	Quitar los recursos
Espera circular	Ordenar los recursos numéricamente

Figura 3-10. Resumen de estrategias para prevenir el bloqueo mutuo.

3.3.6 Evitar los bloqueos mutuos

En la Fig. 3-8 vimos que el bloqueo mutuo se evitaba no imponiendo reglas arbitrarias a los procesos sino analizando con detenimiento cada petición de recurso para ver si se puede satisfacer sin peligro. Surge la pregunta: ¿hay algún algoritmo que siempre pueda evitar el bloqueo mutuo tomando la decisión correcta en todos los casos? La respuesta es que sí se puede evitar el bloqueo mutuo, pero sólo si se cuenta con cierta información por adelantado. En esta sección examinaremos formas de evitar los bloqueos mutuos mediante una asignación cuidadosa de los recursos.

El algoritmo del banquero para un solo recurso

Un algoritmo de planificación que puede evitar el bloqueo mutuo se debe a Dijkstra (1965) y se conoce como algoritmo del banquero. Este algoritmo toma como modelo la forma en que un banquero de una ciudad pequeña podría tratar con un grupo de clientes a los que ha concedido líneas de crédito. En la Fig. 3-11(a) vemos cuatro clientes, a cada uno de los cuales se ha otorgado cierto número de unidades de crédito (p. ej., 1 unidad = 1K dólares). El banquero sabe que no todos los clientes van a necesitar su crédito máximo de inmediato, así que sólo ha reservado 10 unidades en lugar de 22 para atenderlos, (En esta analogía, los clientes son procesos, las unidades de crédito son, digamos, unidades de cinta, y el banquero es el sistema operativo.)

Los clientes atienden sus respectivos negocios, haciendo peticiones de préstamos de vez en cuando. En un momento dado, la situación es la que se muestra en la Fig. 3-11(b). Una lista de los clientes junto con el dinero que ya se les prestó (unidades de cinta que ya se les asignaron) y el crédito máximo disponible (número máximo de unidades de cinta que se necesitarán al mismo tiempo posteriormente) se denomina estado del sistema respecto a la asignación de recursos.

Se dice que un estado es seguro si existe una secuencia de otros estados que conduzca a una situación en la que todos los clientes obtienen préstamos hasta sus límites de crédito (todos los procesos obtienen todos sus recursos y finalizan). El estado de la Fig. 3-11(b) es seguro porque, quedándole dos unidades, el banquero puede posponer todas las peticiones excepto la de Miguel,

		Utilizados	Máximo
Nombre		↓	↓
Andrés	0	6	
Bárbara	0	5	
Miguel	0	4	
Susana	0	7	
Disponibles: 10			
(a)			
		Utilizados	Máximo
Nombre		↓	↓
Andrés	1	6	
Bárbara	1	5	
Miguel	2	4	
Susana	4	7	
Disponibles: 2			
(b)			
		Utilizados	Máximo
Nombre		↓	↓
Andrés	1	6	
Bárbara	2	5	
Miguel	2	4	
Susana	4	7	
Disponibles: 1			
(c)			

Figura 3-11. Tres estados de asignación de recursos: (a) Seguro. (b) Seguro. (c) Inseguro.

de modo que dejará que éste finalice y libere sus cuatro recursos. Teniendo cuatro recursos disponibles, el banquero puede otorgar a Susana o a Bárbara las unidades que necesiten, etcétera.

Consideremos lo que sucedería si se concediera a Bárbara una unidad más en la Fig. 3-11(b); tendríamos la situación de la Fig. 3-11(c) que es insegura. Si todos los clientes pidieran repentinamente sus préstamos máximos, el banquero no podría satisfacer ninguna de sus peticiones, y tendríamos un bloqueo mutuo. Un estado inseguro no tiene que dar pie al bloqueo mutuo, ya que un cliente podría no necesitar toda su línea de crédito disponible, pero el banquero no puede contar con que esto sucederá.

Por tanto, el algoritmo del banquero consiste en considerar cada petición en el momento en que se presenta y ver si su satisfacción conduce o no a un estado seguro. Si es así, se concede lo solicitado; si no, se pospone la petición. Para determinar si un estado es seguro, el banquero verifica si tiene suficientes recursos para satisfacer al cliente que está más cerca de su máximo. Si los tiene, se supone que esos préstamos ya fueron pagados, y a continuación se verifica el cliente que ahora está más cerca de su límite, y así sucesivamente. Si todos los préstamos pueden pagarse tarde o temprano, el estado es seguro y puede satisfacerse la petición inicial.

Trayectorias de recursos

El algoritmo anterior se describió en términos de una sola clase de recursos (p. ej., sólo unidades de cinta o sólo impresoras, pero no algunas de cada clase). En la Fig. 3-12 vemos un modelo para manejar dos procesos y dos recursos, por ejemplo, una impresora y un graficador. El eje horizontal representa el número de instrucciones ejecutadas por el proceso A. El eje vertical representa el número de instrucciones ejecutadas por el proceso B. En 1 A solicita una impresora; en '2' A necesita un graficador. La impresora y el graficador son liberados en 13 e 14, respectivamente. El proceso J necesita el graficador de 15 a 17, y la impresora, de '6 a 18.

Cada punto del diagrama representa un estado conjunto de los dos procesos. Inicialmente, el estado está en p, donde ningún proceso ha ejecutado todavía instrucciones. Si el planificador decide ejecutar A primero, llegamos al punto q, en el que A ya ejecutó cierto número de instrucciones, pero Ji todavía no ejecuta ninguna. En el punto q la trayectoria se vuelve vertical, lo que

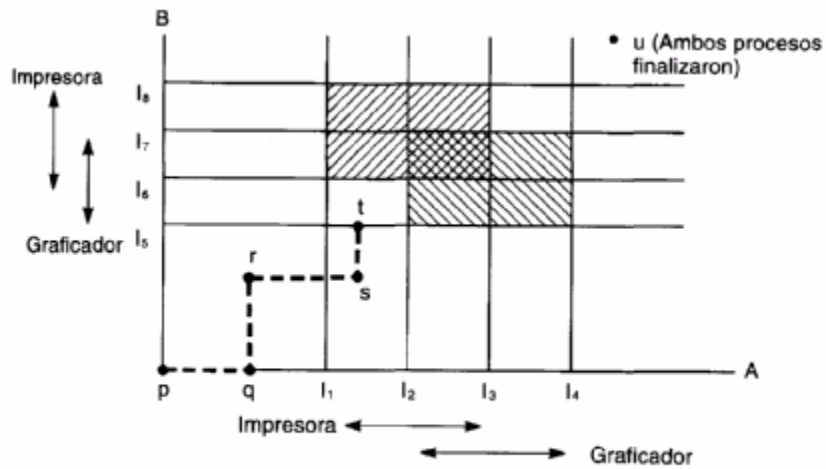


Figura 3-12. Dos trayectorias de recursos de procesos.

indica que el planificador decidió ejecutar B. Con un solo procesador, todas las trayectorias deben ser horizontales o verticales, nunca diagonales. Además, el movimiento siempre es hacia el norte o hacia el este, nunca hacia el sur o hacia el oeste (los procesos no pueden ejecutarse hacia atrás).

Cuando A cruza la línea I en la trayectoria de r a s, solicita y recibe la impresora. Cuando B llega al punto t, solicita el graficador.

Las regiones sombreadas son de especial interés. La región con líneas inclinadas del suroeste al noreste indican que ambos procesos tienen la impresora. La regla de exclusión mutua hace que sea imposible entrar en esta región. De forma similar, la región sombreada en la otra dirección indica que ambos procesos tienen el graficador, y es igualmente imposible.

Si el sistema alguna vez entra en el cuadro delimitado por 1 e '2 a los lados e 15 e '6 de arriba a abajo, nos encontraremos ante un bloqueo mutuo cuando llegue a la intersección de '2 e 16. En este punto, A está solicitando el graficador y B está solicitando la impresora, y ambos dispositivos ya están asignados. Todo el cuadro es inseguro y no debe entrarse en él. En el punto ζt lo único seguro es ejecutar el proceso A hasta que llegue a 14. De ahí en adelante, cualquier trayectoria a u es buena.

El algoritmo del banquero para múltiples recursos

Este modelo gráfico es difícil de aplicar al caso general de un número arbitrario de procesos y un número arbitrario de clases de recursos, cada una con múltiples ejemplares (p. ej., dos graficadores, tres unidades de cinta). No obstante, el algoritmo del banquero puede generalizarse para este fin. En la Fig. 3-13 se muestra cómo funciona.

En la figura vemos dos matrices. La de la izquierda muestra cuántos ejemplares de cada recurso se han asignado actualmente a cada uno de los cinco procesos. La matriz de la derecha muestra cuántos recursos necesita todavía cada proceso para poder finalizar. Al igual que en el

	Proceso	Unidades de cinta	Graficadores	Impresoras	CD-ROM
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Recursos asignados

	Proceso	Unidades de cinta	Graficadores	Impresoras	CD-ROM
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Recursos que aún se necesitan

E = (6342)
P = (5322)
A = (1020)

Figura 3-13. El algoritmo del banquero con múltiples recursos.

caso de un solo recurso, los procesos deben expresar sus necesidades de recursos totales antes de ejecutarse, a fin de que el sistema pueda calcular la matriz de la derecha en cada paso.

Los tres vectores a la derecha de la figura muestran los recursos existentes, E, los recursos poseídos, P y los recursos disponibles, A, respectivamente. Por E podemos ver que el sistema tiene seis unidades de cinta, tres graficadores, dos impresoras y dos unidades de CD-ROM. De éstos, cinco unidades de cinta, tres graficadores, dos impresoras y dos unidades de CD-ROM están asignados actualmente. Esto puede verse sumando las cuatro columnas de recursos en la matriz de la izquierda. El vector de recursos disponibles es simplemente la diferencia entre lo que el sistema tiene y lo que se está usando actualmente.

Ahora podemos plantear el algoritmo para verificar si un estado es seguro o no.

1. Busque una fila, R, cuyas peticiones de recursos no se hayan otorgado y sean en todos los casos menores que o iguales a A. Si no existe tal fila, el sistema tarde o temprano llegará a un bloqueo mutuo porque ningún proceso podrá ejecutarse hasta finalizar.

2. Suponga que el proceso de la fila elegida solicita todos los recursos que necesita (lo cual se garantiza que es posible) y finaliza. Marque ese proceso como terminado y agregue todos sus recursos al vector A.

3. Repita los pasos 1 y 2 hasta que todos los procesos se marquen como terminados, en cuyo caso el estado inicial era seguro, o hasta que ocurra un bloqueo mutuo, en cuyo caso no lo era.

Si hay varios procesos que pueden escogerse en el paso 1, no importa cuál se seleccione: la reserva de recursos aumentará o, en el peor de los casos, seguirá igual.

Volvamos ahora al ejemplo de la Fig. 3-13. El estado actual es seguro. Supongamos ahora que el proceso B solicita una impresora. Esta petición puede satisfacerse porque el estado resultante sigue siendo seguro (el proceso D puede finalizar, y luego el proceso A o E, seguidos por el resto).

Imaginemos ahora que después de otorgarle a B una de las dos impresoras restantes E quiere la última impresora. La concesión de ese recurso reduciría el vector de recursos disponibles a

(1 0 0 0), lo que conduce a un bloqueo mutuo. Es evidente que no se puede satisfacer por ahora la petición de E y deberá posponerse durante un rato.

Este algoritmo fue publicado por Dijkstra en 1965. Desde entonces, casi todos los libros sobre sistemas operativos lo han descrito detalladamente y se han escrito innumerables artículos acerca de diversos aspectos de él. Desafortunadamente, pocos autores han tenido la audacia de señalar que, si bien en teoría el algoritmo es maravilloso, en la práctica es casi inútil porque los wocesos casi nunca saben por adelantado cuáles serán sus necesidades de recursos máximas. Además, el número de procesos no es fijo, sino que varía dinámicamente conforme los usuarios inician y terminan tareas. Por añadidura, los recursos que se creía estaban disponibles pueden desaparecer repentinamente (las unidades de cinta pueden descomponerse).

En síntesis, los esquemas que hemos descrito dentro del rubro de “prevención” son demasiado restrictivos, y el algoritmo que se describió como de “evitación” requiere información que, por lo regular no está disponible. Si a usted se le ocurre un algoritmo de propósito general que realice el trabajo tanto en teoría como en la práctica, escríbalo y envíelo a su publicación local sobre ciencias de la computación.

Para aplicaciones específicas, se conocen muchos algoritmos excelentes. Por ejemplo, en muchos sistemas de bases de datos una operación que ocurre con frecuencia es la petición de candados para varios registros seguida de la actualización de todos los registros asegurados. Cuando se ejecutan varios procesos al mismo tiempo, el peligro de que ocurra un bloqueo mutuo es muy real.

La estrategia que suele usarse es la de candados de dos fases. En la primera fase, el proceso trata de asegurar todos los registros que necesita, uno por uno. Si lo logra, realiza sus actualizaciones y libera los candados. Si algún registro ya está asegurado, el proceso libera los candados que había adquirido y comienza otra vez. En cierto sentido, este enfoque es similar a solicitar todos los recursos que se necesitan por adelantado, o al menos antes de hacer algo irreversible.

Sin embargo, tal estrategia no es aplicable en general. Por ejemplo, en los sistemas de tiempo real y en los de control de procesos, no es aceptable terminar simplemente un proceso a la mitad porque un recurso no esté disponible, y comenzar otra vez desde el principio. Tampoco es aceptable comenzar otra vez si el proceso ha leído mensajes de la red o enviado mensajes por ella, si ha actualizado archivos o si ha hecho alguna otra cosa que no pueda repetirse sin peligro. El algoritmo sólo funciona en los casos en los que el programador ha dispuesto las cosas cuidadosamente de modo que el programa pueda detenerse en cualquier momento durante la primera fase y reiniciarse. Desafortunadamente, no todas las aplicaciones pueden estructurarse de esta manera.

3.4 GENERALIDADES DE E/S EN MINIX

La E/S en MINIX tiene la estructura que se muestra en la Fig. 3-6. Las cuatro capas superiores de esa figura corresponden a la estructura de cuatro capas de MINIX que se muestra en la Fig. 2-26. En las siguientes secciones daremos un vistazo a cada una de las capas, haciendo hincapié en los controladores de dispositivos. El manejo de interrupciones ya se estudió en el capítulo anterior,

y la E/S independiente del dispositivo se verá cuando lleguemos al sistema de archivos, en el capítulo 5.

3.4.1 Manejadores de interrupciones en MINIX

Muchos de los controladores de dispositivos inician algún dispositivo de E/S y luego se bloquean en espera de que llegue un mensaje. Ese mensaje por lo regular es generado por el manejador de interrupciones que corresponde a ese dispositivo. Otros controladores de dispositivos no inician una E/S física (p. ej., la lectura de un disco en RAM y la escritura en una pantalla mapeada en memoria), no usan interrupciones y no esperan un mensaje de un dispositivo de E/S. En el capítulo anterior presentamos con gran detalle el mecanismo mediante el cual las interrupciones generan mensajes y causan conmutaciones de tareas, y no hablaremos más de él aquí. Sin embargo, los manejadores de interrupciones pueden hacer algo más que generar mensajes; en muchos casos también realizan cierto procesamiento de las entradas y salidas en el nivel más bajo. Describiremos esto de forma general aquí y luego veremos los detalles cuando estudiemos el código para diversos dispositivos.

En el caso de los dispositivos de disco, las entradas y salidas generalmente implican ordenar a un dispositivo que realice su operación, y luego esperar hasta que se finaliza la operación. El controlador en hardware del disco realiza casi todo el trabajo, y poco se exige al manejador de interrupciones. Ya vimos que el manejador de interrupciones completo para la tarea del disco duro consiste en sólo tres líneas de código, y la única operación de E/S es la lectura de un solo byte para determinar el estado del controlador en hardware. Las cosas serían en verdad sencillas si todas las interrupciones pudieran manejarse con tanta facilidad.

En otros casos el manejador de bajo nivel tiene más cosas que hacer. El mecanismo de transferencia de mensajes tiene un costo. Si una interrupción puede ocurrir con frecuencia pero la cantidad de E/S que se maneja por interrupción es pequeña, puede ser costoso hacer que el manejador mismo realice un poco más de trabajo y posponga el envío de un mensaje a la tarea hasta una interrupción subsecuente, cuando la tarea tenga algo más que hacer. MINIX maneja de este modo las interrupciones del reloj. En muchos tics del reloj no hay mucho que hacer, con excepción de mantener la hora. Esto puede hacerse sin enviar un mensaje a la tarea del reloj misma. El manejador del reloj incrementa una variable llamada `pending_ticks` (tics pendientes). La hora actual es la suma de la hora registrada la última vez que se ejecutó la tarea del reloj más el valor de `pending_ticks`. Cuando la tarea del reloj recibe un mensaje y se despierta, suma `pending_ticks` a su variable principal para llevar la hora y luego pone en ceros `pending_ticks`. El manejador de interrupciones del reloj examina algunas otras variables y sólo envía un mensaje a la tarea del reloj cuando detecta que ésta tiene trabajo real que efectuar, como entregar una alarma o planificar la ejecución de un nuevo proceso. El manejador también podría enviar un mensaje a la tarea de la terminal.

En la tarea de la terminal vemos otra variación del tema de los manejadores de interrupciones. Esta tarea maneja varios tipos de hardware distintos, incluido el teclado y las líneas RS-232. Cada uno de éstos tiene su propio manejador de interrupciones. El teclado se ajusta perfectamente a la descripción de un dispositivo en el que puede haber relativamente poca E/S que realizar en respuesta

a cada interrupción. En una PC ocurre una interrupción cada vez que se oprime o se suelta una tecla. Esto incluye teclas especiales como SHIFT y CTRL, pero si nos olvidamos de ellas por el momento, podemos decir que, en promedio, se recibe medio carácter por interrupción. Puesto que no hay mucho que la terminal pueda hacer con medio carácter, parece sensato enviarle un mensaje sólo cuando pueda lograrse algo que valga la pena. Examinaremos los detalles más adelante; por ahora sólo diremos que el manejador de interrupciones del teclado realiza la lectura de bajo nivel de los datos del teclado y luego elimina los eventos que puede ignorar, como la liberación de una tecla ordinaria. (La liberación de una tecla especial, como SHIFT, no puede ignorarse.) Luego se colocan en una cola códigos que representan todos los eventos no ignorados para que sean procesados posteriormente por la tarea de la terminal misma.

El manejador de interrupciones del teclado difiere del sencillo paradigma que hemos presentado del manejador de interrupciones que envía un mensaje a la tarea a la que está asociado, porque este manejador de interrupciones no envía mensajes. En su lugar, cuando el manejador agrega un código a la cola, modifica una variable `tty_timeout`, que es leída por el manejador de interrupciones del reloj. Si una interrupción no modifica la cola, tampoco se modifica `tty_timeout`. En el siguiente tic del reloj el manejador del reloj envía un mensaje a la tarea de terminal si la cola se ha modificado. Otros manejadores de interrupciones tipo terminal, como los de las líneas RS-232, funcionan del mismo modo. La tarea de la terminal recibe un mensaje poco tiempo después de que se recibe un carácter, pero no se genera necesariamente un mensaje por cada carácter cuando los caracteres llegan rápidamente. Pueden acumularse varios caracteres y luego ser procesados como respuesta a un solo mensaje. Además, todos los dispositivos de terminal se revisan cada vez que la tarea de terminal recibe un mensaje.

3.4.2 Controladores de dispositivos en MINIX

Para cada clase de dispositivo de E/S presente en un sistema MINIX hay una tarea de E/S (controlador de dispositivo) individual. Estos controladores son procesos con todas las de la ley, cada uno con su propio estado, registros, pila, etc. Los controladores de dispositivos se comunican entre sí (si es necesario) y con el sistema de archivos usando el mecanismo de transferencia de mensajes estándar que utilizan todos los procesos de MINIX. Los controladores de dispositivos sencillos se escriben como archivos fuente únicos, por ejemplo `clock.c`. En el caso de otros controladores, como los del disco en RAM, el disco duro y el disco flexible, hay un archivo fuente para manejar cada tipo de dispositivo, además de un conjunto de rutinas comunes contenidas en `drive` que apoyan todos los tipos de hardware distintos. En cierto sentido esto divide el nivel de controladores de dispositivos de la Fig. 3-6 en dos subniveles. Esta separación de las partes del software dependiente del hardware e independiente del hardware facilita la adaptación a una diversidad de configuraciones de hardware diferentes. Aunque se usa algo de código fuente común, el controlador para cada tipo de disco se ejecuta como proceso aparte, con objeto de realizar transferencias de datos rápidas.

El código fuente del controlador de terminal está organizado de forma similar, con el código independiente del hardware en `uy.c` y el código fuente para apoyar diferentes dispositivos, como las consolas con mapa en memoria, el teclado, las líneas en serie y las seudoterminales, en archivos

individuales. Sin embargo, en este caso, un solo proceso apoya todos los diferentes tipos de dispositivos.

En el caso de grupos de dispositivos como los discos y terminales, para los cuales hay varios archivos fuente, también hay archivos de cabecera. Drive.h apoya todos los controladores de dispositivos por bloques. Tty.h provee definiciones comunes para todos los dispositivos de terminal.

La diferencia principal entre los controladores de dispositivos y otros procesos es que los primeros se enlazan juntos en el kernel, y, por tanto, todos comparten el mismo espacio de direcciones. En consecuencia, si varios controladores de dispositivos utilizan un procedimiento común, sólo se enlazará una copia en el binario de MINIX.

Este diseño es altamente modular y moderadamente eficiente; también es uno de los pocos lugares en los que MINIX difiere de UNIX de una forma esencial. En MINIX un proceso lee un archivo enviando un mensaje al proceso del sistema de archivos. Éste, a su vez, puede enviar un mensaje al controlador de dispositivo pidiéndole que lea el bloque requerido. Esta secuencia (un poco simplificada respecto a la realidad) se muestra en la Fig. 3-14(a). Al llevar a cabo estas interacciones a través del mecanismo de mensajes, obligamos a varias partes del sistema a comunicarse de forma estándar con otras partes. No obstante, al colocar todos los controladores de dispositivos en el espacio de direcciones del kernel les permitimos tener acceso fácil a la tabla de procesos y a otras estructuras de datos clave cuando es necesario.

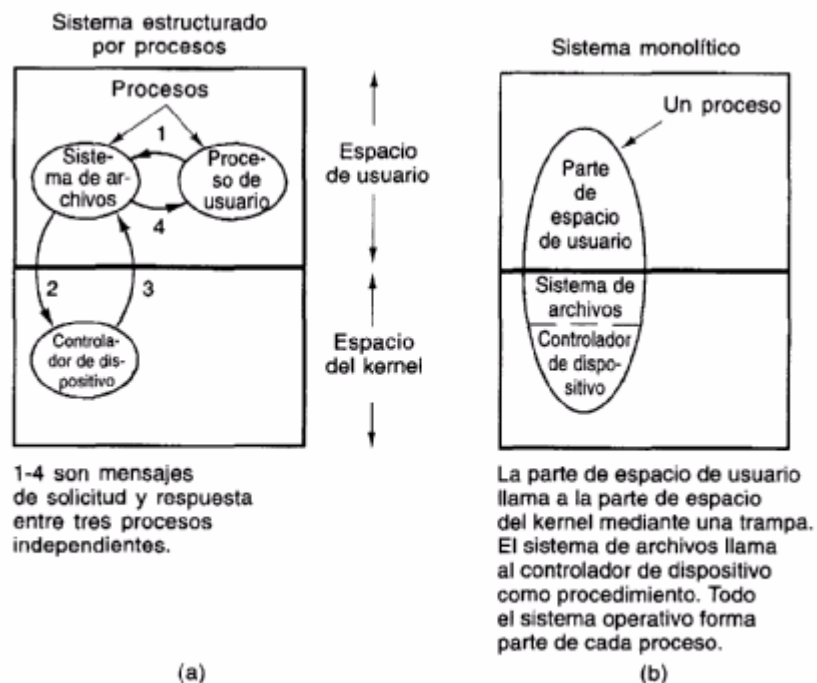


Figura 3-14. Dos formas de estructurar la comunicación usuario-sistema.

En UNIX todos los procesos tienen dos partes: una parte de espacio de usuario y una parte de espacio del kernel, como se aprecia en la Fig. 3-14(b). Cuando se efectúa una llamada al sistema,

el sistema operativo conmuta de la parte de espacio de usuario a la parte de espacio del kernel de una forma un tanto mágica. Esta estructura es una reliquia del diseño de MULTICS, en el que la conmutación era sólo una llamada ordinaria al sistema, no una trampa seguida por el almacenamiento del estado de la parte de usuario, como en UNIX.

Los controladores de dispositivos en UNIX son simplemente procedimientos del kernel invocados por la parte de espacio del kernel del proceso. Cuando un controlador necesita esperar una interrupción, invoca un procedimiento del kernel que lo pone a dormir hasta que algún manejador de interrupciones lo despierta. Observe que es el proceso de usuario mismo el que se pone a dormir aquí, porque las partes del kernel y del usuario son en realidad partes distintas del mismo proceso.

Entre los diseñadores de sistemas operativos son interminables los debates acerca de las ventajas relativas de los sistemas monolíticos, como UNIX, y los sistemas estructurados por procesos, como MINIX. El enfoque de MINIX está mejor estructurado (es más modular), tiene interfaces más limpias entre los componentes y se extiende fácilmente a los sistemas distribuidos en los que los diversos procesos se ejecutan en diferentes computadoras. El enfoque de UNIX es más eficiente, porque las llamadas a procedimientos son mucho más rápidas que el envío de mensajes. MINIX se dividió en muchos procesos porque creemos que, con la aparición de computadoras personales cada vez más potentes, se justifica hacer un poco más lento el sistema en aras de tener una estructura de software más clara. Cabe señalar que muchos diseñadores de sistemas operativos no comparten esta opinión.

En este capítulo estudiaremos los controladores para el disco en RAM, el disco duro, el reloj y la terminal. La configuración estándar de MINIX también incluye controladores para disco flexible e impresora, que no explicaremos detalladamente. La distribución de software de MINIX contiene código fuente de controladores adicionales para líneas en serie RS-232, una interfaz SCSI, CD ROM, adaptador Ethernet y tarjeta de sonido. Éstos pueden incluirse recompilando MINIX.

Todas estas tareas se comunican con otras partes del sistema MINIX de la misma forma: se envían mensajes de petición a las tareas. Los mensajes contienen diversos campos que contienen el código de operación (p. ej., READ o WRITE) y sus parámetros. La tarea intenta satisfacer una petición y devuelve un mensaje de respuesta.

En el caso de los dispositivos por bloques, los campos de los mensajes de petición y respuesta son los que se muestran en la Fig. 3-15. El mensaje de petición incluye la dirección de un área de almacenamiento intermedio que contiene los datos que se van a transmitir o en la que se esperan los datos recibidos. La respuesta incluye información de estado que permite al proceso solicitante verificar que su petición se llevó a cabo correctamente. Los campos para los dispositivos por caracteres son similares pero pueden variar un poco de una tarea a otra. Los mensajes dirigidos a la tarea del reloj, por ejemplo, contienen tiempos, y los mensajes a la tarea de terminal pueden contener la dirección de una estructura de datos que especifica todos los múltiples aspectos configurables de una terminal, como los caracteres que se usarán para las funciones de edición intralínea borrar-carácter y eliminar-línea.

La función de cada tarea es aceptar peticiones de Otros procesos, normalmente el sistema de archivos, y llevarlas a cabo. Todas las tareas de los dispositivos por bloques se escribieron de modo que obtienen un mensaje, lo llevan a cabo y envían una respuesta. Entre otras cosas, esta decisión implica que tales tareas son estrictamente secuenciales y no contienen multiprogramación interna,

Petición		
Campo	Tipo	Significado
m.m_type	int	Operación solicitada
m.DEVICE	int	Dispositivo secundario a usar
m.PROC_NR	int	Proceso que solicita la E/S
m.COUNT	int	Cuenta de bytes o código ioctl
m.POSITION	long	Posición en el dispositivo
m.ADDRESS	char*	Dirección dentro del proceso solicitante

Respuestas		
Campo	Tipo	Significado
m.m_type	int	Siempre TASK_REPLY
m.REP_PROC_NR	int	Igual que PROC_NR en la petición
m.REP_STATUS	int	Bytes transferidos o número de error

Figura 3-15. Campos de los mensajes enviados por el sistema de archivos a los controladores de dispositivos por bloques y campos de las respuestas devueltas.

a fin de mantenerlas sencillas. Cuando se emite una petición de hardware, la tarea ejecuta una operación RECEIVE especificando que sólo le interesa aceptar mensajes de interrupción, no nuevas peticiones de trabajo. Cualesquier mensajes de petición nuevos que lleguen se mantendrán esperando hasta que se haya finalizado el trabajo en curso (principio de cita). La tarea de terminal es un poco diferente, ya que una sola tarea da servicio a varios dispositivos. Así, es posible aceptar una nueva petición de entrada del teclado mientras se está atendiendo una petición de lectura de una línea serial. No obstante, para cada dispositivo se debe finalizar una petición antes de iniciar una nueva.

Estructuralmente, el programa principal para cada controlador de dispositivo por bloques es el mismo, y se bosqueja en la Fig. 3-16. Cuando se arranca el sistema, se inicia por turno cada uno de los controladores para darles la oportunidad de inicializar tablas internas y cosas por el estilo. Luego, cada tarea de controlador se bloquea tratando de obtener un mensaje. Cuando llega un mensaje, se guarda la identidad de quien lo originó y se invoca un procedimiento para llevar a cabo el trabajo, invocándose un procedimiento distinto para cada operación disponible. Una vez finalizado el trabajo, se devuelve una respuesta al originador del mensaje y la tarea regresa al principio del ciclo para esperar la siguiente petición.

Cada uno de los procedimientos dev_xxx se encarga de una de las operaciones que puede ejecutar el controlador, y devuelve un código de estado para indicar qué sucedió. Este código, que se incluye en el mensaje de respuesta como campo REP_STATUS, es el número de bytes transferidos (cero o positivo) si todo salió bien, o el número de error (negativo) si no fue así. Este conteo puede diferir del número de bytes solicitado. Si se llega al final de un archivo, el número de bytes

```

message mess;                                /* buffer del mensaje */

void io_task() {
  initialize();                               /* sólo se hace una vez, al inicializar el sistema */
  while (TRUE) {
    receive(ANY, &mess);                       /* esperar una petición de trabajo */
    caller = mess.source;                       /* proceso del que vino el mensaje */
    switch(mess.type) {
      case READ:    rcode = dev_read(&mess); break;
      case WRITE:   rcode = dev_write(&mess); break;
      /* Aquí van otros casos, incluidos OPEN, CLOSE e IOCTL */
      default:      rcode = ERROR;
    }
    mess.type = TASK_REPLY;
    mess.status = rcode;                       /* código de resultado */
    send(caller, &mess);                       /* enviar mensaje de respuesta al invocador */
  }
}

```

Figura 3-16. Bosquejo del procedimiento principal de una tarea de E/S.

disponibles puede ser menor que el número solicitado. En las terminales se devuelve como mucho una línea, aunque el número solicitado haya sido mayor.

3.4.3 Software de E/S independiente del dispositivo en MINIX

En MINIX el proceso del sistema de archivos contiene todo el código de E/S independiente del dispositivo. El sistema de E/S está tan íntimamente relacionado con el sistema de archivos que se fusionaron para formar un solo proceso. Las funciones del sistema de archivos son las que se muestran en la Fig. 3-5, excepto la petición y liberación de dispositivos de uso exclusivo, que no existen en la configuración actual de MINIX. Sin embargo, si surge la necesidad de agregarlos a los controladores de dispositivos pertinentes en el futuro, no sería difícil hacerlo.

Además de manejar la interfaz con los controladores, el almacenamiento intermedio y la asignación de bloques, el sistema de archivos también se encarga de la protección y administración de los nodos-i, directorios y sistemas de archivos montados. Cubriremos este sistema en el capítulo 5.

3.4.4 Software de E/S de nivel de usuario en MINIX

El modelo general que bosquejamos antes en este capítulo también tiene aplicación aquí. Hay procedimientos de biblioteca para realizar llamadas al sistema y para todas las funciones de C requeridas por el estándar oslx, como las funciones de entrada y salida `formatadaprintfy` `scanf`. La configuración estándar de MINIX contiene un demonio de spool, `lpd`, que coloca en spool e imprime los archivos que se le pasan con el comando `ip`. La distribución de software estándar de

MINIX contiene varios demonios que apoyan diversas funciones de red. Las operaciones de red requieren cierto apoyo del sistema operativo que no forma parte de MINIX en la configuración que se describe en este libro, pero es fácil recompilar MINIX agregando el servidor de red, el cual se ejecuta con la misma prioridad que el administrador de memoria y el sistema de archivos y, al igual que ellos, se ejecuta como proceso de usuario.

3.4.5 Manejo de bloqueos mutuos en MINIX

Haciendo honor a su herencia, MINIX sigue el mismo camino que UNIX en lo tocante a los bloqueos mutuos: simplemente hace caso omiso del problema. MINIX no contiene dispositivos de BIS de uso exclusivo, aunque si alguien quisiera conectar una unidad de cinta DAT estándar de la industria a una PC, preparar el software para ella no presentaría problemas especiales. En pocas palabras, el único lugar en el que pueden ocurrir bloqueos mutuos es en el uso de los recursos compartidos implícitos, como las ranuras de la tabla de procesos, las de la tabla de nodos-i, etc. Ninguno de los algoritmos de bloqueo mutuo conocidos puede manejar recursos como éstos que no se solicitan explícitamente.

En realidad, lo anterior no es estrictamente cierto. Aceptar el riesgo de que los procesos de usuario podrían entrar en bloqueo mutuo es una cosa, pero dentro del sistema operativo mismo hay unos cuantos lugares en los que se ha tenido mucho cuidado de evitar problemas. El principal es la interacción entre el sistema de archivos y el administrador de memoria. Este último envía mensajes al sistema de archivos para leer el archivo binario (programa ejecutable) durante una llamada al sistema EXEC, y también en otros contextos. Si el sistema de archivos no está ocioso cuando el administrador de memoria intenta enviarle mensajes, el administrador de memoria se bloqueará. Si a continuación el sistema de archivos intentara enviar un mensaje al administrador de memoria, él también descubriría que la cita falló y se bloquearía, dando lugar a un bloqueo mutuo.

Este problema se evitó construyendo el sistema de forma tal que el sistema de archivos nunca envía mensajes de solicitud al administrador de memoria, sólo respuestas, con una excepción menor. La excepción es que, durante el arranque, el sistema de archivos informa al administrador de memoria el tamaño del disco en RAM, y se garantiza que el administrador de memoria estará esperando ese mensaje.

Es posible poner candados a dispositivos y archivos incluso sin el apoyo del sistema operativo. Un nombre de archivo puede servir como variable verdaderamente global, cuya presencia o ausencia puede ser percibida por todos los demás procesos. Al igual que en la mayor parte de los sistemas UNIX, en los sistemas MINIX suele estar presente un directorio especial, /usr/spool/locks/, en el que los procesos pueden crear archivos de candado para marcar los recursos que están utilizando. El sistema de archivos de MINIX también maneja los candados de advertencia para archivos al estilo posix, pero ninguno de estos mecanismos puede hacerse obligatorio. Todo depende del buen comportamiento de los procesos, y no hay nada que impida a un programa utilizar un recurso asegurado por otro proceso. Esto no es exactamente lo mismo que expropiar el recurso, porque tampoco impide que el primer proceso intente seguir utilizando el recurso. Dicho de otro modo, no hay exclusión mutua. El resultado de semejante acción por parte de un proceso mal comportado seguramente será un desastre, pero no habrá bloqueo mutuo.

3.3 DISPOSITIVOS POR BLOQUES EN MINIX

En las siguientes secciones regresaremos a los controladores de dispositivos, el tema principal de este capítulo, y estudiaremos varios de ellos con detalle. MINIX maneja varios dispositivos por bloques distintos, así que comenzaremos por analizar los aspectos comunes de todos los dispositivos por bloques. Luego estudiaremos el disco en RAM, el disco duro y el disco flexible. Cada uno de éstos resulta interesante por una razón distinta. El disco en RAM es un buen ejemplo para estudiarlo porque tiene todas las propiedades de los dispositivos por bloques en general con excepción de la E/S real, ya que el “disco” en realidad es sólo una porción de la memoria. Esta simplificación hace que éste sea un buen lugar para comenzar. El disco duro ilustra en qué consiste un verdadero controlador de disco. Podríamos esperar que el disco flexible fuera más fácil de apoyar que el duro, pero la realidad es que no es así. No explicaremos todos los detalles del disco flexible, pero sí señalaremos varias de las complicaciones que se encuentran en su controlador.

Después de tratar los controladores por bloques, estudiaremos otras clases de controladores. El reloj es importante porque todo sistema tiene uno, y porque es totalmente distinto de los demás controladores; además, también resulta interesante como excepción de la regla de que todos los dispositivos son por bloques o por caracteres, ya que no encaja en ninguna de las dos categorías. Por último, estudiaremos el controlador de terminal, que es importante en todos los sistemas y además es un buen ejemplo de controlador de dispositivo por caracteres.

Cada una de estas secciones describe el hardware pertinente, los principios de software en que se basa el controlador, un bosquejo de la implementación y el código mismo. Esta estructura hace que la lectura de estas secciones sea de utilidad incluso para aquellos lectores a los que no les interesan los detalles del código mismo.

3.5.1 Generalidades de los controladores de dispositivos por bloques en MINIX

Ya mencionamos que los procedimientos principales de todas las tareas de E/S tienen una estructura similar. MINIX siempre tiene por lo menos tres tareas de dispositivo por bloque (el controlador del disco en RAM, el controlador del disco flexible y varios controladores de disco duro posibles) compiladas en el sistema. Además, pueden incluirse por compilación una tarea de CD-ROM y un controlador SCSI (interfaz estándar de computadora pequeña) si se requiere apoyo para tales dispositivos. Aunque el controlador de cada uno de estos dispositivos se ejecuta como proceso independiente, el hecho de que todos se compilan como parte del código ejecutable del kernel permite compartir una cantidad considerable del código, sobre todo los procedimientos de utilería.

Desde luego, cada controlador de dispositivo por bloques tiene que hacer algo de inicialización. El controlador del disco en RAM debe reservar memoria, el controlador del disco duro tiene que determinar los parámetros del hardware del disco, etc. Todos los controladores de disco se invocan individualmente para la inicialización específica para el hardware, pero después de hacer lo que sea necesario, cada controlador invoca la función que contiene el ciclo principal común. Este ciclo se ejecuta indefinidamente; no hay retorno al invocador. Dentro del ciclo principal se recibe un mensaje, se invoca una función que realice la operación requerida por cada mensaje, y se genera el mensaje de respuesta.

El ciclo principal común invocado por cada tarea de controlador de disco no es sólo una copia de una función de biblioteca compilada en cada controlador. Sólo hay una copia del código del ciclo principal en el binario de MINIX. La técnica que se emplea consiste en hacer que cada uno de los controladores individuales pase al ciclo principal un parámetro que consta de un apuntador a una tabla la cual contiene las direcciones de las funciones que el controlador usará para cada operación. Luego, se invocan indirectamente dichas funciones. Esta técnica también permite a los controladores compartir funciones. La Fig. 3-17 muestra un bosquejo del ciclo principal, en una forma similar a la de la Fig. 3-16. Instrucciones como

```
code = (*entry_points->dev_read>(&mess);
```

son invocaciones indirectas de funciones. Cada controlador invoca una función `dev_read` distinta, aunque todos ejecutan el mismo ciclo principal. Por otro lado, algunas otras operaciones, como `CLOSE`, son tan sencillas que más de un dispositivo puede invocar la misma función.

```

message mess;                               /* buffer de mensajes */

void shared_io_task(struct driver_table *entry_points) {
/* cada tarea realiza inicialización antes de invocar esta función */
  while(TRUE) {
    receive(ANY, &mess);
    caller = mess.source;
    switch(mess.type) {
      case READ:      rcode = (*entry_points->dev_read>(&mess); break;
      case WRITE:    rcode = (*entry_points->dev_write>(&mess); break;
      /* Aquí van otros casos, incluidos OPEN, CLOSE e IOCTL */
      default:       rcode = ERROR;
    }
    mess.type = TASK_REPLY;
    mess.status = rcode;                       /* código de resultado */
    send(caller, &mess);
  }
}

```

Figura 3-17. Procedimiento principal de tarea de E/S compartido usando llamadas indirectas.

Este empleo de una sola copia del ciclo es una buena ilustración del concepto de procesos que presentamos en el capítulo 1 y analizamos extensamente en el capítulo 2. En la memoria sólo hay una copia del código ejecutable para el ciclo principal de los controladores de dispositivos por bloques, pero se ejecuta como ciclo principal de tres o más procesos distintos. Cada uno de estos procesos probablemente está en un punto distinto del código en un instante dado, y cada uno opera con su propio conjunto de datos y tiene su propia pila.

Hay sE/S posibles operaciones que pueden solicitarse a cualquier controlador de dispositivo. Éstas corresponden a los posibles valores que se pueden encontrar en el campo `m.m_type` del mensaje de la Fig. 3-15, y son:

1. OPEN (abrir)
2. CLOSE (cerrar)
3. READ (leer)
4. WRITE (escribir)
5. IOCTL (control de E/S)
6. SCATTERED_IO (E/S dispersa)

Los lectores con experiencia en programación probablemente ya conocen la mayor parte de estas operaciones. En el nivel de controlador de dispositivo, casi todas las operaciones están relacionadas con llamadas al sistema que tienen el mismo nombre. Por ejemplo, el significado de READ y WRITE debe ser claro. Para cada una de estas operaciones, se transfiere un bloque de datos del dispositivo a la memoria del proceso que inició la llamada, o viceversa. Una operación READ normalmente no causa un retorno al invocador antes de que se haya completado la transferencia de datos, pero un sistema operativo podría colocar en un buffer los datos transferidos durante un WRITE para transferirlos realmente a su destino en un momento posterior, y regresar inmediatamente al invocador. Para el invocador, eso es excelente, ya que ahora está en libertad de reutilizar el buffer del cual el sistema operativo copió los datos que se escribirán. Las operaciones OPEN y CLOSE para un dispositivo tienen un significado similar al que tienen las llamadas al sistema OPEN y CLOSE aplicadas a operaciones con archivos: una operación OPEN debe verificar que el dispositivo este accesible, o devolver un mensaje de error si no lo está, y un CLOSE debe garantizar que todos los datos en buffer que fueron escritos por el invocador se transfieran por completo a su destino final en el dispositivo.

La operación IOCTL tal vez no sea tan conocida. Muchos dispositivos de E/S tienen parámetros operativos que ocasionalmente deben examinarse y tal vez modificarse. Las operaciones IOCTL se encargan de esto. Un ejemplo común es cambiar la rapidez de transmisión o la paridad de una línea de comunicación. En el caso de los dispositivos por bloques, las operaciones IOCTL son menos comunes. La consulta o modificación de la forma en que un dispositivo de disco está dividido en particiones se realiza con una operación IOCTL en MINIX (aunque podría haberse efectuado igualmente bien leyendo y escribiendo un bloque de datos).

La operación SCATTERED_IO sin duda es la menos conocida. Con la excepción de los dispositivos de disco excepcionalmente rápidos (por ejemplo, el disco en RAM), es difícil obtener un rendimiento de E/S de disco satisfactorio si todas las peticiones piden bloques individuales, uno a la vez. Una petición SCATTERED_IO permite al sistema de archivos solicitar la lectura o escritura de múltiples bloques. En el caso de una operación READ, los bloques adicionales tal vez no hayan sido solicitados por el proceso a cuyo nombre se efectúa la llamada; el sistema operativo intenta anticipar peticiones de datos futuras. En una petición de este tipo el controlador del dispositivo no tiene que conceder necesariamente todas las transferencias solicitadas. La petición de cada bloque puede modificarse con un bit de bandera que le indica al controlador que la petición

es opcional. En efecto, el sistema de archivos puede decir: “sería bueno tener todos estos datos, pero realmente no los necesito todos en este momento”. El dispositivo puede hacer lo que más le convenga. El controlador del disco flexible, por ejemplo, devolverá todos los bloques de datos que pueda leer de una misma pista, diciendo efectivamente: “te voy a dar éstos, pero tardaría mucho en pasar a otra pista; pídemelo el resto después”.

Cuando es preciso escribir datos, no puede ser opcional escribir o no un bloque dado. No obstante, el sistema operativo puede poner en buffer varias peticiones de escritura en la suposición de que la escritura de múltiples bloques pueda efectuarse de manera más eficiente que si se atiende cada petición en el momento en que llega. En una petición SCA 17'ERED_IO, sea para leer o para escribir, la lista de bloques solicitados está ordenada, y esto hace la operación más eficiente que si se atienden las peticiones al azar. Además, como sólo se hace una llamada al controlador para transferir múltiples bloques, se reduce el número de mensajes enviados dentro de MINIX.

3.5.2 Software controlador de dispositivos de bloques común

Las definiciones que todos los controladores de dispositivos por bloques necesitan están en driverh. Lo más importante de este archivo es la estructura driver, en las líneas 9010 a 9020, que todo controlador utiliza para pasar una lista de las direcciones de las funciones que usará para realizar cada parte de su trabajo. También se define aquí la estructura device (líneas 9031 a 9034) que contiene la información más importante referente a las particiones, la dirección base y el tamaño en bytes. Se escogió este formato para que no fueran necesarias conversiones al trabajar con dispositivos basados en la memoria, maximizando así la rapidez de respuesta. En el caso de los discos reales hay tantos factores adicionales que retrasan el acceso, que la conversión a sectores no implica una tardanza significativa.

El ciclo principal y las funciones compartidas de todas las tareas de controlador por bloques están en driver.c. Después de efectuar toda la inicialización específica del hardware que pudiera ser necesaria, cada controlador invoca driver_task, pasando una estructura driver como argumento de la llamada. Después de obtener la dirección del buffer que se usará para operaciones de DMA, se entra en el ciclo principal (líneas 9158 a 9199). Este ciclo se ejecuta indefinidamente; no hay retorno al invocador.

El sistema de archivos es el único proceso que se supone enviará mensajes a una tarea de controlador. El switch de las líneas 9165 a 9175 verifica esto. Se ignora una interrupción sobrante del hardware, y cualquier otro mensaje mal dirigido sólo producirá la exhibición de una advertencia en la pantalla. Esto parece inocuo, pero desde luego es muy probable que el proceso que envió el mensaje erróneo esté bloqueado permanentemente esperando una respuesta. En el switch del ciclo principal, los primeros tres tipos de mensajes, DEV_OPEN, DEV_GLOSE y DEV_IOCTL, tienen como resultado llamadas indirectas empleando direcciones que se pasaron en la estructura driver. Los mensajes DEV_READ, DEV_WRITE y SCA7TERED_IO producen llamadas directas a do o do_vrdwt. Sin embargo, todas las llamadas pasan la estructura driver como argumento desde el interior del switch, sean directas o indirectas, para que todas las rutinas invocadas puedan usarla también si es necesario.

Después de hacer lo que se solicita en el mensaje, es posible que sea necesario realizar algo de aseo, dependiendo de la naturaleza del dispositivo. En el caso de un disquette, por ejemplo, esto podría implicar iniciar un temporizador para apagar el motor de la unidad de disco si no llega pronto otra petición. También se usa una llamada indirecta para esto. Después del aseo, se cons huye un mensaje de respuesta y se envía al invocador (líneas 9194 a 9198).

Lo primero que hace cada tarea después de entrar en el ciclo principal es invocar `init_buffer` (línea 9205), que asigna un buffer para usarse en operaciones de DMA. Todas las tareas de controlador usan el mismo buffer, si es que acaso lo usan; algunos controladores no utilizan DMA. Las inicializaciones de todas las entradas después de la primera son redundantes pero no hacen daño. Sería más laborioso codificar una prueba para determinar si debe pasarse o no por alto la inicialización.

Esta inicialización sólo es necesaria debido a una peculiaridad del hardware de la IBM PC original, que exige que el buifer de DMA no cruce una frontera de 64K. Es decir, un buffer de DMA de 1K puede comenzar en 64510, pero no en 64514, porque un buffer que empieza en esta última dirección se extiende un poco más allá de la frontera de 64K que está en 65536.

Esta molesta regla existe porque la IBM PC usaba un chip de DMA viejo, el Intel 8237A, que contiene un contador de 16 bits. Se necesita un contador más grande porque el DMA usa direcciones absolutas, no relativas a un registro de segmento. En las máquinas más viejas que pueden direccionar sólo 1 M de memoria, los 16 bits de orden bajo de la dirección de DMA se cargan en el 8237A, y los 4 bits de orden superior se cargan en un latch de 4 bits. Las máquinas más nuevas usan un latch de 8 bits y pueden direccionar 1 6M. Cuando el 8237A pasa de 0xFFFF a 0x0000, no genera un bit de acarreo que se sume al latch, de modo que la dirección de DMA salta repentinamente 64K hacia abajo en la memoria.

Un programa en C portátil no puede especificar una posición de memoria absoluta para una esirtictura de datos, así que no hay forma de evitar que el compilador coloque el buffer en un lugar inadecuado. La solución es asignar memoria a un arreglo de bytes dos veces más grande que lo necesario en buffer (línea 9135), y reservar un apuntador `tmp_buf` (línea 9136) que se usará para acceder realmente a este arreglo. `init_buffer` realiza un ajuste provisional de `tmp_buf`, apuntando al principio de huffer, y luego prueba para determinar si deja suficiente espacio antes de llepar a una frontera de 64K. Si el ajuste provisional no provee suficiente espacio, se incrementa 1 bu! en el número de bytes que realmente se requieren. Así, siempre se desperdicia algo de espaci en liii c o en el otro del espacio asignado en buffer, pero nunca hay una falla debida a pie el hujier quede en una frontera de 64K.

Las computadoras más nuevas de la familia IBM PC tienen mejores controladores en hardware para DM A. este código podría simplificarse, recuperándose una pequeña cantidad de memoria, ‘.i podemos estar seguros de que nuestra máquina es inmune a este problema. Si usted está omsitierarido esto, piense cómo se manifestaría el error en caso de estar equivocado al respecto. Si Se desea un buffer de 1K para DMA, hay una probabilidad de 1 en 64 de que habrá un problema en una inñquina con el chip de DMA antiguo. Cada vez que se modifica el código fuente del kernel de modo tal que el tamaño del kernel compilado cambia, existe la misma probabilidad de que se manifieste el problema. Lo más probable es que el siguiente mes o el siguiente año, cuando ocurra la falla, se atribuya al último código que se modificó. Características inesperadas del hardware

como ésta pueden hacer que desperdiciemos semanas buscando errores extraordinariamente difíciles de rastrear (sobre todo cuando, como en este caso, el manual de referencia técnica no dice ni una palabra al respecto).

La siguiente función de driver `c` es `do_rdwt`. Ésta, a su vez, puede invocar tres funciones dependientes del dispositivo a las que apuntan los campos `dr_prepare`, `dr_schedule` y `dr_finish`. En lo que sigue usaremos la flotación del lenguaje C * apuntador a función para indicar que estamos hablando de la función a la que apunta `apuntador_a_función`.

Después de verificar que la cuenta de bytes en la petición sea positiva, `do_rdwt` invoca a `*dr_prepare`. Ésta debe tener éxito, ya que `*dr_prepare` sólo puede fallar si se especifica un dispositivo no válido en una operación OPEN. A continuación se llena una estructura `iorequest_s` estándar (definida en la línea 3194 en `include/minixltype.h`). Luego viene otra llamada indirecta, esta vez a `*drschedule`. Como veremos cuando hablemos del hardware de disco en la siguiente sección, responder a las peticiones de disco en el orden en que se reciben puede ser poco eficiente, y esta rutina permite a un dispositivo en particular manejar las peticiones en la forma que resulte óptima para el dispositivo. La indirección en este caso enmascara gran parte de las posibles variaciones en la forma como funcionan los dispositivos individuales. En el caso del disco en RAM, `dr_schedule` apunta a una rutina que realmente realiza la BIS, y la siguiente llamada indirecta, a `*drfjnsh`, es una operación que no hace nada. En el caso de un disco real, `dr_finish` apunta a una rutina que lleva a cabo todas las transferencias de datos pendientes solicitadas en todas las llamadas anteriores a `*dr_schedule` después de la última llamada a `*dr_finish`. Sin embargo, como veremos, en algunas circunstancias la llamada a `*dr_finish` podría no lograr la transferencia de todos los datos solicitados.

En cualquier llamada que sea la que realice una transferencia real de datos, se modifica el contador `io_n_bytes` de la estructura `iorequest_s`, devolviendo un número negativo si hubo un error o uno positivo para indicar la diferencia entre el número de bytes especificados en la petición original y los bytes que se transfirieron con éxito. No es necesariamente un error que no se haya transferido ningún byte; esto indica que se llegó al final del dispositivo. Al regresar al ciclo principal, el código de error negativo se devuelve en el campo `REP_STATUS` del mensaje de respuesta si hubo un error. Si no, los bytes que faltan por transferirse se restan de la petición original en el campo `COUNT` del mensaje (línea 9249) y el resultado (el número realmente transferido) se devuelve en `REP_STATUS` en el mensaje de respuesta de `driver_task`.

La siguiente función, `do_vrdwt`, maneja todas las peticiones de E/S dispersa. Un mensaje que solicita E/S dispersa utiliza el campo `ADDRESS` para apuntar a un arreglo de estructuras tipo `iorequest_s`, cada una de las cuales especifica la información que se necesita para una petición: la dirección del buffer, el desplazamiento dentro del dispositivo, el número de bytes y si se va a leer o a escribir. Las operaciones de una petición deben ser todas de lectura o todas de escritura, y estar ordenadas en orden de bloque dentro del dispositivo. Se debe trabajar más que en la lectura o escritura sencilla realizada con `do_rdwt`, ya que el arreglo de peticiones debe copiarse en el espacio de kernel, pero una vez que se ha hecho esto se efectúan las mismas tres llamadas indirectas a las rutinas dependientes del dispositivo `*dr_prepare`, `*dr_schedule` y `*dr_finish`. La diferencia es que la llamada de en medio, `*dr_schedule`, se ejecuta en un ciclo, una vez para cada petición, o hasta que ocurre un error (líneas 9288 a 9290). Una vez finalizado el ciclo, se invoca

una vez `*dr finish`, y luego se vuelve a copiar el arreglo de peticiones en el lugar de donde se obtuvo. El campo `io_nbytes` de cada elemento del arreglo se habrá modificado de modo que refleje el número de bytes transferidos, y aunque el total no se regresa directamente en el mensaje de respuesta que `driver_task` construye, el invocador puede extraer el total de este arreglo.

En una petición de lectura de E/S dispersa, no todas las transferencias solicitadas en la llamada a `*dr_schedule` se habrán efectuado forzosamente cuando se efectúe la llamada final a `*drfinish`, como explicamos en la sección anterior. El campo `io_request` de la estructura `iorequest_s` contiene un bit de bandera que le indica al controlador de dispositivo si la petición de ese bloque es opcional.

Las siguientes rutinas de drive proporcionan apoyo general a las operaciones anteriores. Se puede usar una llamada a `*dr name` para obtener el nombre de un dispositivo. Si el dispositivo no tiene un nombre específico, la función `no_name` obtiene el nombre del dispositivo de la tabla de tareas. Algunos dispositivos podrían no requerir un servicio en particular; por ejemplo, un disco en RAM no requiere que se haga nada especial para atender una petición `DEV_CLOSE` (cerrar dispositivo). En este caso la función `do_nop` atiende la petición, devolviendo diversos códigos en función del tipo de petición. Las rutinas que siguen, `nop_finish` y `nop_cleanup`, son rutinas ficticias similares para dispositivos que no requieren los servicios de `*dr finish` ni de `*dr_cleanup`.

Algunas funciones de dispositivos de disco requieren retrasos; por ejemplo, para esperar que el motor de una unidad de disquete alcance la velocidad de operación. Por ello, `driver.c` es un buen lugar para la siguiente función, `clock_mess`, que sirve para enviar mensajes a la tarea del reloj. Esta función se invoca con el número de tics de reloj que hay que esperar y la dirección de la función que debe invocarse cuando haya transcurrido el plazo.

Por último, `do_dioctl` (línea 9364) lleva a cabo peticiones `DEV_IOCTL` para un dispositivo por bloques. Es un error solicitar cualquier operación de `DEV_IOCTL` que no sea leer (`DIOGETP`) o escribir (`DJOSETP`) información de particiones. `Do_dioctl` invoca la función `*dr_prepare` del dispositivo para verificar que el dispositivo sea válido y obtener un apuntador a la estructura `device` que describe la base y el tamaño de las particiones en bytes. En una petición de lectura, `do_dioctl` invoca la función `*dr_geometry` para obtener la información de cilindro, cabeza y sector de la partición.

3.5.3 La biblioteca de controladores

Los archivos `drvlib.h` y `drvlib.c` contienen código dependiente del sistema que maneja las particiones de disco en computadoras compatibles con IBM PC.

Las particiones permiten dividir un solo dispositivo de almacenamiento en subdispositivos. Las particiones se usan comúnmente en los discos duros, pero MINIX también proporciona apoyo para la división en particiones de los discos flexibles. Entre las razones para dividir un disco en particiones podemos citar:

1. La capacidad de disco es más económica por unidad en los discos grandes. Si se usan dos o más sistemas operativos con diferentes sistemas de archivos, resulta más

económico dividir en particiones un solo disco grande que instalar varios discos de menor tamaño, uno para cada sistema operativo.

2. Los sistemas operativos pueden tener límites en cuanto al tamaño de dispositivos que pueden manejar. La versión de MINIX que estudiamos aquí puede manejar un sistema de archivos de 1 GB, pero las versiones antiguas están limitadas a 256 MB. Todo el espacio de disco adicional se desperdicia.

3. Un sistema operativo podría usar dos o más sistemas de archivos distintos. Por ejemplo, podría usarse un sistema de archivos estándar para los archivos ordinarios y uno con una estructura diferente para el espacio de intercambio de la memoria virtual.

4. Puede ser aconsejable colocar una porción de los archivos de un sistema en un dispositivo lógico independiente. Si se coloca el sistema de archivos raíz de MINIX en un dispositivo pequeño será fácil respaldarlo y copiarlo en un disco en RAM en el momento del arranque.

El manejo de las particiones de disco es específico para cada plataforma. La especificidad no tiene que ver con el hardware, pues el apoyo de particiones es independiente del dispositivo. Sin embargo, si desea ejecutarse más de un sistema operativo en un conjunto de hardware dado, todos deben acordar un formato para la tabla de particiones. En las IBM PC el estándar lo establece el comando `fdisk` de MS-DOS, y otros sistemas operativos, como MINIX, OS/2 y Linux, usan este formato para poder coexistir con MS-DOS. Si se traslada MINIX a otro tipo de máquina, tiene sentido usar un formato de tabla de particiones compatible con otros sistemas operativos instalados en el nuevo hardware. Es por ello que el código fuente de MINIX que apoya las particiones en computadoras IBM se coloca en `drvlib.c`, en lugar de incluirse en `driver.c`; eso facilita el traslado de MINIX a hardware distinto.

La estructura de datos básica heredada de los diseñadores de firmware se define en `ihm/partition.h`, que se incluye mediante una instrucción `#include` en `drvlib.h`. Dicha estructura contiene información sobre la geometría cilindro-cabeza-sector de cada partición, así como códigos que identifican el tipo de sistema de archivos de la partición y una bandera activa que indica si es arrancable. MINIX no necesita casi nada de esta información una vez que se verifica el sistema de archivos.

La función `partition` (en `drvlib.c`, línea 9521) se invoca cuando se abre por primera vez un dispositivo por bloques. Sus argumentos incluyen una estructura `driver`, para que pueda invocar funciones específicas del dispositivo, un número de dispositivo secundario inicial y un parámetro que indica si el estilo de partición es disco flexible, partición primaria o subpartición. `partition` invoca la función `*dr_prepare` específica para el dispositivo a fin de verificar que el dispositivo sea válido y colocar la dirección base y el tamaño en una estructura `device` del tipo mencionado en la sección anterior. Luego `partition` invoca `get_part_table` para determinar si está presente una tabla de particiones y, en caso dado, leerla. Si no hay tabla de particiones, la función habrá terminado su trabajo; si la hay, se calculará el número de dispositivo secundario de la primera partición usando las reglas de numeración de dispositivos secundarios que apliquen al estilo de partición

especificado en la llamada original. En el caso de particiones primarias, la tabla de particiones está ordenada de modo que el orden de las particiones sea congruente con el que usan otros sistemas operativos.

En este punto se invoca otra vez `*dr prepare`, esta vez usando el número de dispositivo `i` calculado para la primera partición. Si el subdispositivo es válido, se procesarán cíclicamente todas las entradas de la tabla, comprobando que los valores leídos de la tabla en el dispositivo no estén fuera del intervalo calculado antes para la base y el tamaño de todo el dispositivo. Si hay una discrepancia, la tabla en memoria se ajusta de modo que sea congruente. Esto puede parecer paranoico, pero dado que las tablas de particiones pueden haber sido escritas por diferentes sistemas operativos, un programador, usando otro sistema, puede haber tratado de usar ingenio samente la tabla de particiones para algo inesperado, o podría haber basura en la tabla en disco por alguna otra razón. Confiamos más en los números que calculamos usando MINIX. Más vale asegurarse que tener que arrepentirse.

Aún dentro del ciclo, para todas las particiones del dispositivo, si la partición se identifica como una partición MINIX, se invoca `partition` recursivamente a fin de obtener información sobre las subparticiones. Si la partición se identifica como una partición extendida, se invoca más bien la siguiente función del archivo, `extpartition`.

`Extpartition` (línea 9593) realmente no tiene nada que ver con el sistema operativo MINIX, así que no la explicaremos con detalle, MS-DOS usa particiones extendidas, que no son sino un meca nismo más para crear subparticiones. Si queremos apoyar los comandos de MINIX capaces de leer y escribir archivos MS-DOS, necesitamos tener conocimiento de estas subparticiones.

`Get_part_table` (línea 9642) invoca `do_rdwt` para obtener el sector de un dispositivo (o subdispositivo) donde se encuentra una tabla de particiones. El desplazamiento que se incluye como argumento es cero si la invocación es para obtener una partición primaria, y distinto de cero en el caso de una subpartición. La función busca el número mágico (0xAA55) y devuelve un estado verdadero o falso para indicar si se encontró o no una tabla de particiones válida. Si se encuentra la tabla, se copia en la dirección de tabla que se pasó como argumento.

Por último, `sort` (línea 9676) ordena las entradas de una tabla de particiones en orden ascen dente por sector. Las entradas marcadas como carentes de particiones se excluyen del ordenamiento, quedando al final aunque tengan un valor de cero en su campo de sector bajo. El ordenamiento es del tipo de burbuja; no hay necesidad de usar un algoritmo elegante para ordenar una lista de cuatro elementos.

3.6 DISCOS EN RAM

Ahora regresaremos a los controladores de dispositivos por bloques individuales y estudiaremos varios de ellos detalladamente. El primero que veremos es el controlador de disco en RAM, que puede proporcionar acceso a cualquier parte de la memoria. La aplicación principal de este controlador es reservar una parte de la memoria para ser usada como disco ordinario. Esto no quiere decir que el almacenamiento sea permanente, pero una vez que los archivos se copian en esta área el acceso a ellos puede ser extremadamente rápido.

En un sistema como MINIX, que se diseñó para funcionar en computadoras que apenas contaban con un disco flexible, el disco en RAM tiene otra ventaja. Si colocamos el dispositivo raíz en el disco en RAM, el solitario disco flexible puede montarse y desmontarse a voluntad, manejando así medios removibles. Si se colocara el dispositivo raíz en el disco flexible sería imposible guardar archivos en disquete, ya que el dispositivo raíz (el único disquete) no puede desmontarse. Además, tener el dispositivo raíz en el disco en RAM hace al sistema muy flexible; se puede montar en él cualquier combinación de discos flexibles o duros. Aunque la mayor parte de las computadoras ahora tienen disco duro, con excepción de las utilizadas en sistemas incorporados, el disco en RAM es útil durante la instalación, antes de que el disco duro esté listo para ser utilizado por MINIX, o cuando se desea usar MINIX temporalmente sin realizar una instalación completa.

3.6.1 Hardware y software de discos en RAM

La idea en que se basa el disco en RAM es sencilla. Un dispositivo por bloques es un medio de almacenamiento con dos comandos: escribir un bloque y leer un bloque. Normalmente, estos bloques se guardan en memorias giratorias, como discos flexibles o discos duros. Un disco en RAM es más sencillo: simplemente usa una porción preasignada de la memoria principal para almacenar los bloques. El disco en RAM tiene la ventaja de ofrecer acceso instantáneo (no hay búsqueda ni retraso rotacional), lo que lo hace adecuado para almacenar programas o datos a los que se accede con frecuencia.

Como acotación, vale la pena señalar brevemente una diferencia entre los sistemas que manejan sistemas de archivos montados y los que no lo hacen (como MS-DOS y WINDOWS). En los sistemas de archivos montados, el dispositivo raíz siempre está presente y en un lugar fijo, y es posible montar sistemas de archivos removibles (esto es, discos) en el árbol de archivos para formar un sistema de archivos integrado. Una vez que todo está montado, el usuario no tiene que preocuparse por saber en qué dispositivo está un archivo.

En contraste, en los sistemas como MS-DOS el usuario debe especificar la ubicación de cada archivo, ya sea explícitamente como en B:\DIR\ARCHIVO o usando ciertos valores predeterminados (dispositivo actual, directorio actual, etc.). Si sólo hay uno o dos discos flexibles, esta carga es manejable, pero en los sistemas de cómputo grandes, con docenas de discos, tener que seguir la pista continuamente a todos los dispositivos sería insoportable. Recuerde que UNIX se ejecuta en sistemas que van desde una IBM PC o una estación de trabajo hasta supercomputadoras como la Cray-2; MS-DOS sólo se ejecuta en sistemas pequeños.

En la Fig. 3-18 se ilustran los fundamentos del disco en RAM. Este disco se divide en n bloques, dependiendo de cuánta memoria se le haya asignado. Cada bloque tiene el mismo tamaño que los bloques empleados en los discos reales. Cuando el controlador recibe un mensaje indicándole que lea o escriba un disco, simplemente calcula el lugar dentro de la memoria del disco en RAM en el que está el bloque solicitado y lo lee o escribe, en lugar de hacerlo en un disco flexible o duro. La transferencia se efectúa invocando un procedimiento en lenguaje ensamblador que copia de o hacia el programa de usuario con la máxima rapidez de que el sistema es capaz.

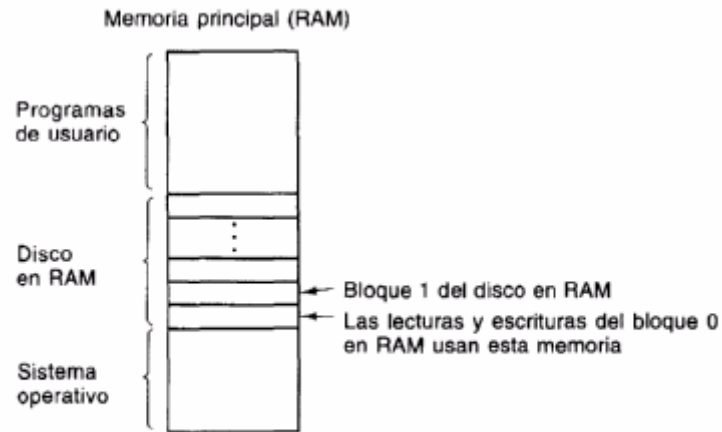


Figura 3-18. Un disco en RAM.

Un controlador de disco en RAM puede manejar varias áreas de memoria utilizadas como discos en RAM, cada una distinguida mediante un número de dispositivo secundario. Por lo regular estas áreas no se traslapan, pero en algunas situaciones puede resultar conveniente hacer que se traslapan, como veremos en la siguiente sección.

3.6.2 Generalidades del controlador de disco en RAM en MINIX

El controlador de disco en RAM es en realidad cuatro controladores en uno, todos muy relacionados entre sí. Cada mensaje que envía al controlador especifica un dispositivo secundario como sigue: O: ¡dey/ram 1: /dev/mem 2: /dev/kmem 3: /dev/null

El primero de estos archivos especiales, ¡dey/ram, es un verdadero disco en RAM. Ni su tamaño ni su origen están incorporados en el controlador; estas características las determina el sistema de archivos cuando se arranca MINIX. Por omisión, se crea un disco en RAM con el mismo tamaño que el dispositivo imagen del sistema de archivos raíz, a fin de poder copiar en él dicho sistema de archivos. Se puede usar un parámetro de arranque para especificar un disco en RAM más grande que el sistema de archivos raíz o, si no se va a copiar en RAM la raíz, se puede especificar cualquier tamaño que quepa en la memoria y que deje suficiente memoria para el funcionamiento del sistema. Una vez determinado el tamaño, se localiza un bloque de memoria suficientemente grande y se retira de la reserva de memoria aun antes de que el administrador de memoria inicie su trabajo. Tal estrategia permite aumentar o reducir el tamaño del disco en RAM sin tener que recompilar el sistema operativo.

Los dos dispositivos secundarios siguientes se utilizan para leer y escribir memoria física y memoria del kernel, respectivamente. Cuando se abre y lee /dev/mem, se obtiene el contenido de

las posiciones de memoria física comenzando por la dirección de memoria absoluta cero (los vectores de interrupción en modo real). Los programas de usuario ordinarios nunca hacen esto, pero un programa de sistema que se ocupa de depurar el sistema podría necesitar este recurso. Si se abre `/dev/mem` y se escribe en él se modifican los vectores de interrupción. Huelga decir que esto sólo debe efectuarlo un usuario experimentado que sepa exactamente lo que está haciendo, y procediendo con la mayor cautela.

El archivo especial `/dev/kmem` es parecido a `/dev/mem`, excepto que el byte 0 de este archivo es el byte 0 de la memoria de datos del kernel. La dirección absoluta de esta posición varía dependiendo del tamaño del código del kernel de MINIX. Esta posición también se usa principalmente para depuración y programas muy especiales. Observe que las áreas de disco en RAM cubiertas por estos dos dispositivos secundarios se traslapan. Si sabemos exactamente cómo está ubicado el kernel en la memoria, podemos colocarnos en el principio del área de datos del kernel y ver exactamente lo mismo que obtenemos cuando leemos desde el principio de `/dev/kmem`. Sin embargo, si recompilamos el kernel cambiando su tamaño, o si en versiones subsecuentes de MINIX el kernel se coloca en algún otro lugar de la memoria, tendremos que colocarnos en un lugar diferente de `/dev/mem` para ver lo mismo que vemos al principio de `/dev/kmem`. Estos dos archivos especiales deben protegerse para evitar que nadie que no sea el superusuario los utilice.

El último archivo de este grupo, `/dev/null`, es un archivo especial que acepta datos y los desecha. Este archivo se utiliza comúnmente en comandos del shell, cuando el programa invocado genera salidas que no sirven de nada. Por ejemplo,

```
a.out >/dev/null
```

ejecuta el programa `a.out` pero desecha sus salidas. El controlador de disco en RAM efectivamente trata este dispositivo secundario como si tuviera tamaño cero, así que jamás se copian datos en él ni de él.

El código para manejar `/dev/ram`, `/dev/mem` y `/dev/kmem` es idéntico. La única diferencia entre ellos es que cada uno corresponde a una porción diferente de la memoria, indicada por los arreglos `ram_origin` y `ram_limit`, cada uno indizado por un número de dispositivo secundario.

3.6.3 Implementación del controlador de disco en RAM en MINIX

Al igual que en otros controladores de disco, el ciclo principal del controlador de disco en RAM está en el archivo `drive,x`. El apoyo específico para dispositivos de memoria está en `memory.c`. El arreglo `m_geom` (línea 9721) contiene la base y el tamaño de cada uno de los cuatro dispositivos de memoria. La estructura `driver` de `m_dtab` en las líneas 9733 a 9743 define las llamadas de dispositivo de memoria que se harán desde el ciclo principal. Cuatro de las entradas de esta tabla son rutinas que hacen nada o casi nada en `driver.c`, indicio seguro de que el funcionamiento de un disco en RAM no es muy complicado que digamos. El procedimiento principal `mem_task` (línea 9749) invoca una función que realiza algunas inicializaciones locales, y a continuación invoca el ciclo principal, el cual recibe mensajes, ejecuta el procedimiento apropiado y devuelve respuestas. No hay retorno a `mem_task` una vez completado este ciclo.

En una operación de lectura o escritura, el ciclo principal efectúa tres llamadas: una para preparar un dispositivo, una para planificar las operaciones de E/S y una para terminar la operación. En el caso de los dispositivos de memoria, la primera de estas llamadas se hace a `m_prepare`. Esta función comprueba que se haya solicitado un dispositivo secundario válido y luego devuelve la dirección de la estructura que contiene la dirección base y el tamaño del área de RAM solicitada. La segunda llamada es a `m_schedule` (línea 9774), que es la función que realiza todo el trabajo. En el caso de los dispositivos de memoria el nombre de esta función es poco apropiado; por definición, en la memoria de acceso aleatorio (RAM) cualquier posición es tan accesible como cualquier otra, así que no hay necesidad de planificar, como sería el caso en un disco provisto de un brazo móvil.

El funcionamiento del disco en RAM es tan sencillo y rápido que nunca hay necesidad de posponer una petición, así que lo primero que hace esta función es poner en 0 el bit que podría ser puesto en 1 por una llamada de E/S dispersa para indicar que la terminación de una operación es opcional. La dirección de destino que se pasa en el mensaje apunta a una posición dentro del espacio de memoria del invocador, y el código de las líneas 9792 a 9794 convierte esta posición en una dirección absoluta en la memoria del sistema, comprobando después que se trata de una dirección válida. La transferencia de datos propiamente dicha se efectúa en la línea 9818 o en la 9820 y es un copiado directo de datos de un lugar a otro.

Los dispositivos de memoria no necesitan un tercer paso para terminar una operación de lectura o escritura, y la ranura correspondiente de `m_dtab` es una llamada a `nop_finish`.

La apertura de un dispositivo de memoria se hace con `m_do_open` (línea 9829). La acción principal se efectúa invocando `m_prepare` para verificar que se esté haciendo referencia a un dispositivo válido. En el caso de una referencia a `/dev/mem` o a `/dev/kmem`, se hace una llamada a `enable_iop` (en el archivo `protect.c`) para cambiar el nivel de privilegio actual de la CPU. Esto no es necesario para acceder a la memoria; es un truco para resolver otro problema. Recuerde que las CPU Pentium implementan cuatro niveles de privilegio. Los programas de usuario son el nivel menos privilegiado. Los procesadores Intel también tienen una característica arquitectónica que no está presente en muchos otros sistemas: un conjunto de instrucciones independiente para direccionar los puertos de E/S. En estos procesadores, los puertos de E/S se tratan aparte de la memoria. Normalmente, cuando un proceso de usuario intenta ejecutar una instrucción que direcciona un puerto de E/S, se produce una excepción de protección general. Por otro lado, hay razones válidas para que MINIX permita a los usuarios escribir programas capaces de acceder a los puertos, sobre todo en los sistemas pequeños. Por tanto, `enable_iop` cambia los bits de nivel de protección de E/S (IOPL) de la CPU a fin de permitir esto. El efecto es que un proceso que tiene permiso de abrir `/dev/mem` o `/dev/kmem` cuenta también con el privilegio adicional de acceso a los puertos de E/S. En una arquitectura en la que los dispositivos de E/S se direccionan como posiciones de memoria, los bits `rwx` de estos dispositivos cubren automáticamente el acceso a E/S. Si esta capacidad estuviera oculta, podría considerarse como un defecto de seguridad, pero ahora usted está enterado de ella. Si usted planea usar MINIX para controlar el sistema de seguridad de un banco, tal vez prefiera recompilar MINIX excluyendo esta función.

La siguiente función, `m_init` (línea 9849), sólo se invoca una vez, cuando se llama inicialmente `mem_tus'k`. Esta función establece la dirección base y el tamaño de `/dev/kmem` y también establece el tamaño de `idev/mem` en 1 MB, 16 MB o 4 GB—1, dependiendo de si MINIX se está

ejecutando en modo 8088, 80286 u 80386. Estos tamaños son los máximos que MINIX maneja y nada tienen que ver con la cantidad de RAM instalada en la máquina.

El disco en RAM apoya varias operaciones de IOCTL en mjoctl (línea 9874). MIO CRAMSIZE es un mecanismo cómodo para que el sistema de archivos fije el tamaño del disco en RAM. La operación MIOCSPSINFO es utilizada tanto por el sistema de archivos como por el administrador de memoria para establecer las direcciones de sus partes de la tabla de procesos en la tabla psinfo, de donde el programa de utilería ps puede obtenerla usando una operación MIOCGPSINFO. Ps es un programa UNIX estándar cuya implementación se complica por la estructura de microkernel de MINIX, que coloca la información de tabla de procesos que el programa necesita en varios lugares distintos. La llamada al sistema IOCTL es una forma cómoda de resolver este problema. De lo contrario, habría que compilar una versión nueva de ps cada vez que se compilara una nueva versión de MINIX.

La última función de memory.c es m_geometly (línea 9934). Los dispositivos de memoria no tienen una geometría de cilindros, pistas y sectores como las unidades de disco mecánicas, pero en caso de que se le pregunte al disco en RAM, éste simulará que la tiene.

3.7 DISCOS

El disco en RAM es una buena introducción a los controladores de disco (por ser tan sencillo), pero los discos reales tienen varios aspectos que aún no hemos tratado. En las siguientes secciones hablaremos un poco del hardware de disco y luego examinaremos los controladores de disco en general y el controlador de disco duro de MINIX en particular. No estudiaremos con detalle el controlador del disco flexible, pero sí mencionaremos algunas de las diferencias entre los controladores de disco duro y los de disco flexible.

3.7.1 Hardware de discos

Todos los discos reales están organizados en cilindros, cada uno de los cuales contiene tantas pistas como cabezas hay apiladas verticalmente. Las pistas se dividen en sectores, de los cuales por lo regular hay entre 8 y 32 por pista en los discos flexibles, llegando a varios cientos en algunos discos duros. Los diseños más sencillos tienen el mismo número de sectores en todas las pistas. Todos los sectores tienen el mismo número de bytes, aunque si pensamos un poco nos daremos cuenta de que los sectores cercanos al borde exterior del disco son físicamente más largos que los cercanos al centro. No obstante, el tiempo que toma leer o escribir un sector es el mismo. La densidad de los datos obviamente es más alta en los cilindros más interiores, y algunos diseños de disco requieren modificar la corriente de alimentación a las cabezas de lectura-escritura para las pistas interiores. El controlador en hardware del disco se encarga de esto sin que lo vea el usuario (o el implementador del sistema operativo).

La diferencia en la densidad de los datos entre las pistas interiores y exteriores implica un sacrificio de la capacidad, y existen sistemas más complejos. Se han probado diseños de disco flexible que giran con mayor rapidez cuando las cabezas están sobre las pistas exteriores. Esto

permite tener más sectores en esas pistas, incrementando la capacidad del disco. Sin embargo, ningún sistema para el cual MINIX actualmente está disponible reconoce tales discos. Los discos duros modernos de gran capacidad también tienen más sectores por pista en las pistas exteriores que en las interiores. Éstas son las unidades IDE (Integrated Drive Electronics), y el complejo procesamiento que los circuitos incorporados en la unidad realizan enmascara los detalles. Para el sistema operativo, estas unidades aparentan tener una geometría sencilla con un número constante de sectores por pista.

Los circuitos de la unidad y del controlador en hardware son tan importantes como el equipo mecánico. El principal elemento de la tarjeta controladora que se instala en el plano posterior de la computadora es un circuito integrado especializado, en realidad una microcomputadora pequeña. En el caso de un disco duro puede ser que los circuitos de la tarjeta controladora sean más sencillos que para un disco flexible, pero esto es así porque la unidad de disco duro en sí tiene un potente controlador electrónico incorporado. Una característica del dispositivo que tiene implicaciones importantes para el controlador del disco en software es la posibilidad de que el controlador en hardware realice búsquedas de sectores en dos o más unidades al mismo tiempo. Esto se conoce como búsquedas traslapadas. Mientras el controlador en hardware y en software están esperando que se lleve a cabo una búsqueda en una unidad, el controlador en hardware puede iniciar una búsqueda en otra unidad. También, muchos controladores en hardware pueden leer o escribir en una unidad mientras buscan en otra u otras, pero un controlador de disco flexible no puede leer o escribir en dos unidades al mismo tiempo. (La lectura o escritura requiere que el controlador transfiera bits en una escala de tiempo de microsegundos, por lo que una transferencia ocupa casi toda su potencia de cómputo.) La situación es diferente en el caso de los discos duros con controladores integrados en hardware, y en un sistema provisto de más de uno de estos discos duros los controladores pueden operar simultáneamente, al menos hasta el grado de transferir datos entre el disco y la memoria intermedia del controlador. Sin embargo, sólo es posible una transferencia a la vez entre el controlador y la memoria del sistema. La capacidad para realizar dos o más operaciones al mismo tiempo puede reducir considerablemente el tiempo de acceso medio.

En la Fig. 3-19 se comparan los parámetros de disquetes de doble lado y doble densidad, que eran el medio de almacenamiento estándar de la IBM PC original, y los parámetros de un disco duro de mediana capacidad como el que podría encontrarse en una computadora basada en Pentium MINIX usa bloques de 1K, así que con cualquiera de estos dos formatos de disco los bloques que el software usa consisten en dos sectores consecutivos, que siempre se leen o escriben juntos como una unidad.

Algo que debemos tener presente al estudiar las especificaciones de los discos duros modernos es que la geometría especificada y utilizada por el controlador en software puede diferir del formato físico, EJ disco duro descrito en la Fig. 3-19, por ejemplo, se especifica con “parámetros de configuración recomendados” de 1048 cilindros, 16 cabezas y 63 sectores por pista. Los circuitos del controlador montados en el disco convierten los parámetros de cabeza y sector lógicos proporcionados por el sistema operativo en los parámetros físicos empleados por el disco. Éste es otro ejemplo de un arreglo diseñado para mantener la compatibilidad con sistemas más antiguos, en este caso firmware viejo. Los diseñadores de la IBM PC original sólo apartaron

Parámetro	Disquete IBM de 360 KB	Disco duro WD de 540 MB
Número de cilindros	40	1048
Pistas por cilindro	2	4
Sectores por pista	9	252
Sectores por disco	720	1056384
Bytes por sector	512	512
Bytes por disco	368640	540868608
Tiempo de búsqueda (cilindros adyacentes)	6 ms	4 ms
Tiempo de búsqueda (caso medio)	77 ms	11 ms
Tiempo de rotación	200 ms	13 ms
Tiempo de arranque/paro del motor	250 ms	9 s
Tiempo para transferir un sector	22 ms	53 μ s

Figura 3-19. Parámetros de disco para el disquete de 360 KB de la IBM PC original y un disco duro Western Digital WD AC2540 de 540 MB.

un campo de 6 bits para contar sectores en el ROM de BIOS, y un disco que tiene más de 63 sectores por pista debe trabajar con un conjunto artificial de parámetros de disco lógicos. En este caso las especificaciones del proveedor dicen que en realidad hay cuatro cabezas, y por tanto podría parecer que realmente hay 252 sectores por pista, como se indica en la figura. Ésta es una simplificación, porque los discos de este tipo tienen más sectores en las pistas exteriores que en las interiores. El disco descrito en la figura sí tiene cuatro cabezas físicas, pero en realidad hay un poco más de 3000 cilindros. Los cilindros se agrupan en una docena de zonas que tienen desde 57 sectores por pista en las zonas más interiores hasta 105 cilindros por pista en los cilindros más exteriores. Estos números no se encuentran en las especificaciones del disco, y las traducciones realizadas por los circuitos electrónicos de la unidad nos ahorran tener que conocer tales detalles.

3.7.2 Software de discos

En esta sección veremos algunos aspectos relacionados con los controladores de disco en general. Primero, considere cuánto tiempo toma leer o escribir un bloque de disco. El tiempo requerido está determinado por tres factores:

1. El tiempo de búsqueda (el tiempo que toma mover el brazo al cilindro correcto).
2. El retraso rotacional (el tiempo que tarda el sector correcto en girar hasta quedar bajo la cabeza).
3. El tiempo real de transferencia de datos.

En casi todos los discos, el tiempo de búsqueda es mucho mayor que los otros dos, así que si reducimos el tiempo de búsqueda mejoraremos sustancialmente el rendimiento del sistema.

Los dispositivos de disco son propensos a errores. Siempre se graba junto con los datos de cada sector de un disco algún tipo de verificación de errores, una suma de verificación o una verificación de redundancia cíclica. Incluso las direcciones registradas cuando se formatea un disco cuentan con datos de verificación. El controlador en hardware de un disco flexible puede infonnar cuando se detecta un error, pero el software debe decidir qué se hará al respecto. Los controladores en hardware de los discos duros con frecuencia asumen gran parte de esta carga. Sobre todo en el caso de los discos duros, el tiempo de transferencia de sectores consecutivos dentro de la misma pista puede ser muy rápido. Por ello, la lectura de más datos de los que se solicitan y su almacenamiento en un caché de la memoria puede ser una estrategia muy eficaz para acelerar el acceso a disco.

Algoritmos de planificación del brazo del disco

Si el controlador en software del disco acepta peticiones una por una y las atiende en ese orden, es decir, el primero que llega, el primero que se atiende (FCFS: first come, first served), poco puede hacerse por optimar el tiempo de búsqueda. Sin embargo, cuando el disco está sometido a una carga pesada puede usarse otra estrategia. Es probable que, mientras el brazo está realizando una búsqueda para atender una petición, otros procesos generen otras peticiones de disco. Muchos controladores de disco mantienen una tabla, indizada por número de cilindro, con todas las peticiones pendientes para cada cilindro encadenadas en listas enlazadas cuyas cabezas son las entradas de la tabla.

Dado este tipo de estructura de datos, podemos utilizar un mejor algoritmo que el del primero que llega, primero que se atiende. Para entender este algoritmo, consideremos un disco de 40 cilindros. Llega una petición para leer un bloque que está en el cilindro 11. Mientras se está realizando la búsqueda del cilindro 11, llegan peticiones nuevas para los cilindros 1, 36, 16, 34, 9 y 12, en ese orden, y se introducen en la tabla de peticiones pendientes, con una lista enlazada individual para cada cilindro. Las peticiones se muestran en la Fig. 3-20.

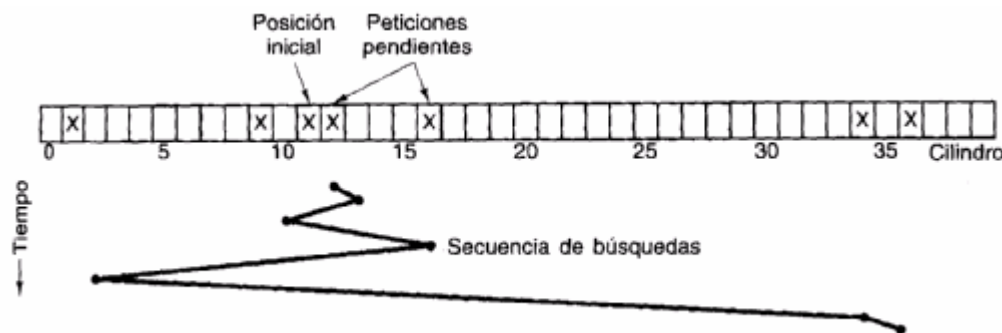


Figura 3-20. Algoritmo de planificación de la primera búsqueda más corta (SSF).

Cuando se termina de atender la petición en curso (para el cilindro 11), el controlador de disco puede escoger la petición que atenderá a continuación. Si el controlador usara FCFS, iría después al cilindro 1, luego al 36, y así sucesivamente. Este algoritmo requeriría movimientos del brazo de 10, 35, 20, 18, 25 y 3, respectivamente, para un total de 111 cilindros.

Como alternativa, el controlador podría atender a continuación la petición más cercana, a fin de minimizar el tiempo de búsqueda. Dadas las peticiones de la Fig. 3-20, la secuencia es 12, 9, 16, 1, 34 y 36, como indica la línea punteada en la parte inferior de la Fig. 3-20. Con esta secuencia, los movimientos del brazo son 1, 3, 7, 15, 33 y 2 para un total de 61 cilindros. Este algoritmo, la primera búsqueda más corta (SSF: *shortest seekfirst*), reduce el movimiento total del brazo casi a la mitad en comparación con FCFS.

Desafortunadamente, SSF tiene un problema. Suponga que siguen llegando más peticiones mientras se están procesando las peticiones de la Fig. 3-20. Por ejemplo, si, después de ir al cilindro 16 está presente una nueva petición para el cilindro 8, ésta tendrá prioridad sobre el cilindro 1. Si luego llega una petición para el cilindro 13, el brazo irá después a ese cilindro, en lugar de al 1. Si la carga del disco es elevada, el brazo tenderá a permanecer en la parte media del disco la mayor parte del tiempo, y las peticiones en los extremos tendrán que esperar hasta que una fluctuación estadística en la carga haga que no se presenten peticiones cerca de la parte media. Las peticiones alejadas de la parte media podrían recibir un servicio deficiente. Los objetivos de tiempo de respuesta mínimo y equitatividad están en conflicto aquí.

Los edificios altos también tienen que enfrentar este trueque. El problema de planificar un elevador de un edificio alto es similar al de planificar un brazo de disco. Continuamente llegan llamadas que reclaman al elevador que se dirija a otros pisos al azar. El microprocesador que controla el elevador fácilmente podría usar FCFS para seguir la pista a la secuencia en que los clientes oprimen el botón de llamada y atenderlos; también podría usar SSF.

Sin embargo, la mayor parte de los elevadores usan un algoritmo distinto para conciliar los objetivos en conflicto de eficiencia y equitatividad: continúan desplazándose en la misma dirección hasta que no hay más peticiones pendientes en esa dirección, y luego se mueven en la dirección opuesta. El algoritmo, conocido tanto en el mundo de los discos como en el de los elevadores como algoritmo del elevador, requiere que el software mantenga un bit: el bit de dirección actual, UP o DOWN. Cuando termina de atenderse una petición el controlador del disco o del elevador examina el bit. Si éste es UP, el brazo o el elevador se mueve a la petición pendiente más cercana hacia arriba. Si no hay peticiones pendientes en posiciones más altas, se invierte el bit de dirección. Si el bit está en DOWN, el movimiento es hacia la siguiente posición solicitada hacia abajo, si la hay.

La Fig. 3-21 muestra el algoritmo del elevador usando las mismas siete peticiones que en la Fig. 3-20, suponiendo que el bit de dirección inicialmente estaba en UP. El orden en que se atienden los cilindros es 12, 16, 34, 36, 9 y 1, que produce movimientos del brazo de 1, 4, 18, 2, 27 y 8, para un total de 60 cilindros. En este caso el algoritmo del elevador es un poco mejor que SSF, aunque usualmente es peor. Una propiedad agradable del algoritmo del elevador es que, dada cualquier colección de peticiones, el límite superior del movimiento total está fijo: no puede ser mayor que dos veces el número de cilindros,

Una ligera modificación de este algoritmo que tiene una menor varianza en los tiempos de respuesta (Teory, 1972) consiste en siempre barrer en la misma dirección. Una vez atendida la petición pendiente para el cilindro de número más alto, el brazo se dirige al cilindro de número más bajo que tiene una petición pendiente y continúa moviéndose hacia arriba. En efecto, se considera que el cilindro de número más bajo está justo arriba del cilindro de número más alto.

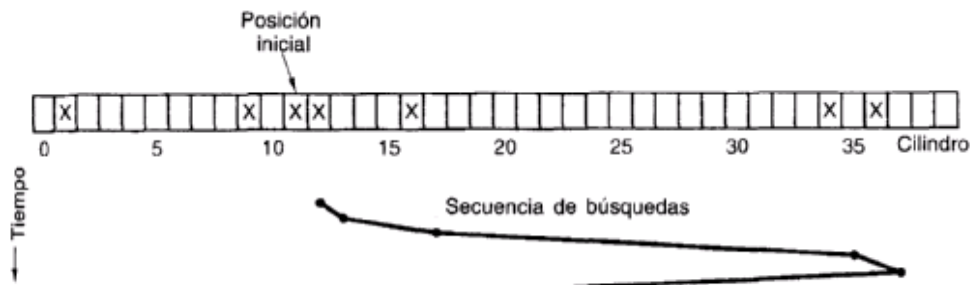


Figura 3-21. Algoritmo del elevador para planificar peticiones de disco.

Algunos controladores de disco en hardware permiten al software determinar qué sector está actualmente bajo la cabeza. Con un controlador así, es posible otra optimización. Si están pendientes dos o más peticiones para el mismo cilindro, el controlador puede solicitar que se lea a continuación el sector que va a pasar primero por la cabeza. Observe que cuando hay varias pistas en un cilindro, pueden atenderse sin retraso peticiones consecutivas para diferentes pistas. El controlador en hardware puede seleccionar cualquiera de sus cabezas instantáneamente, por que la selección de la cabeza no implica movimiento del brazo ni retraso rotacional.

Con un disco duro moderno, la tasa de transferencia de datos es tan alta en comparación con la de un disco flexible que se requiere algún tipo de almacenamiento automático en caché. Por lo regular, cualquier petición de lectura de un sector hace que se lea ese sector y también una parte del resto de la pista actual, o toda, dependiendo de cuánto espacio hay disponible en la memoria caché del controlador. El disco de 540M descrito en la Fig. 3-19 tiene un caché de 64K o 128K. El controlador en hardware determina dinámicamente cómo se utilizará el caché. En su modalidad más simple, el caché se divide en dos secciones, una para leer y la otra para escribir.

Si hay varias unidades de disco presentes, se debe mantener una tabla de peticiones pendientes aparte para cada unidad. Siempre que una unidad está ociosa, se deberá ordenar a su brazo que se mueva al cilindro donde se le necesitará a continuación (suponiendo que el controlador permite búsquedas traslapadas). Una vez que termine la transferencia en curso, se podrá verificar si alguna unidad está colocada en el cilindro correcto. Si una o más lo están, se podrá iniciar una transferencia en una unidad cuya cabeza ya está en el cilindro correcto. Si ninguno de los brazos está en el lugar debido, el controlador deberá ordenar una nueva búsqueda a la unidad que acaba de completar una transferencia y esperar hasta la siguiente interrupción para ver cuál brazo llega a su destino primero.

Manejo de errores

Los discos en RAM no tienen que preocuparse por búsquedas ni optimización rotacional: en cualquier instante todos los bloques pueden leerse o escribirse sin ningún movimiento físico. Otra área en la que los discos en RAM son más sencillos que los reales es la de manejo de errores. Los discos en RAM siempre funcionan; los reales no siempre lo hacen, pues están sujetos a una gran variedad de errores. Entre los más comunes están:

1. Error de programación (p. ej., petición de un sector inexistente).
2. Error transitorio de suma de verificación (p. ej., causado por polvo en la cabeza).
3. Error permanente de suma de verificación (p. ej., si un bloque de disco tiene un daño permanente).
4. Error de búsqueda (p. ej., el brazo se envió al cilindro 6 pero fue al 7).
5. Error de controlador en hardware (p. ej., el controlador se niega a aceptar comandos).

Corresponde al controlador de disco en software manejar todos estos errores lo mejor que pueda.

Ocurren errores de programación cuando el controlador en software le dice al controlador en hardware que busque un cilindro inexistente, lea de un sector inexistente, use una cabeza inexistente o transfiera de o hacia memoria inexistente. La mayor parte de los controladores verifica los parámetros que se les proporcionan y protestan si no son válidos. En teoría, estos errores jamás deberían ocurrir, pero, ¿qué debe hacer el controlador en software si el controlador en hardware indica que ha ocurrido uno? En el caso de un sistema hecho en casa, lo mejor es detenerse y exhibir un mensaje como “Llamar al programador” para que éste pueda rastrear el error y corregirlo. En el caso de un producto de software comercial que se usa en miles de sitios en todo el mundo, tal estrategia resulta menos atractiva. Tal vez lo único que puede hacerse es terminar la petición de disco actual con un error, y abrigar la esperanza de que no ocurra con demasiada frecuencia.

Los errores de suma de verificación transitorios son causados por partículas de polvo del aire que se meten entre la cabeza y la superficie del disco. Casi siempre, estos errores pueden eliminarse repitiendo la operación unas cuantas veces. Si el error persiste, es preciso marcar el bloque como bloque defectuoso y evitarlo.

Una forma de evitar los bloques defectuosos es escribir un programa muy especial que recibe una lista de bloques defectuosos como entrada y prepara cuidadosamente un archivo que contiene todos esos bloques. Una vez creado este archivo, el asignador de disco creará que todos esos bloques están ocupados y nunca los asignará. En tanto nadie trate de leer el archivo de bloques defectuosos, no habrá problemas.

Impedir la lectura del archivo de bloques defectuosos no es cosa trivial. Muchos discos se respaldan copiando su contenido pista por pista en una cinta o unidad de disco de respaldo. Si se sigue este procedimiento, los bloques defectuosos causarán problemas. Respalda el disco archivo por archivo es más lento pero resuelve el problema, siempre que el programa de respaldo conozca el nombre del archivo de bloques defectuosos y no lo copie.

Otro problema que no puede resolverse con un archivo de bloques defectuosos es el de un bloque defectuoso en una estructura de datos del sistema de archivos que debe estar en un lugar fijo. Casi todos los sistemas de archivos tienen por lo menos una estructura de datos cuya posición es fija, a fin de poder encontrarla fácilmente. En un sistema de archivos dividido en particiones puede ser posible redefinir las particiones evitando una pista defectuosa, pero un error permanente en los primeros sectores de un disco flexible o duro generalmente implica que el disco no puede usarse.

Los controladores “inteligentes” reservan unas cuantas pistas que normalmente no están disponibles para los programas de usuario. Cuando se da formato a una unidad de disco, el controlador en hardware determina cuáles bloques están defectuosos y automáticamente sustituye

una de las pistas de repuesto por la defectuosa. La tabla que establece la correspondencia entre las pistas defectuosas y las de repuesto se mantiene en la memoria interna del controlador y en el disco. Esta sustitución es transparente (invisible) para el controlador en software, excepto que su algoritmo de elevador puede tener un rendimiento deficiente si el controlador en hardware está usando secretamente el cilindro 800 cada vez que se solicita el cilindro 3. La tecnología de fabricación de las superficies de grabación de los discos es mejor que en el pasado, pero todavía no es perfecta. Por otro lado, la tecnología de ocultar las imperfecciones para que el usuario no las vea también ha mejorado. En los discos duros como el que se describe en la Fig 3-19, el controlador también maneja los errores nuevos que pueden aparecer con el uso, asignando permanentemente bloques sustitutos cuando determina que un error es irremediable. Con tales discos, el controlador en software casi nunca recibe una indicación de que hay bloques defectuosos.

Los errores de búsqueda son causados por problemas mecánicos en el brazo. El controlador en hardware sigue la pista a la posición del brazo internamente. Para realizar una búsqueda, el controlador envía una serie de pulsos al motor del brazo, un pulso por cilindro, a fin de desplazar el brazo al nuevo cilindro. Cuando el brazo llega a su destino, el controlador lee el número de cilindro real (que se escribió cuando se dio formato a la unidad. Si el brazo está en el lugar equivocado, habrá ocurrido un error de búsqueda.

La mayor parte de los controladores de disco duro corrige tales errores automáticamente, pero muchos controladores de disco flexible (incluidos los de las IBM PC) simplemente ponen en 1 un bit de error y dejan el resto al controlador en software. Éste maneja el error emitiendo un comando RECALIBRATE que mueve el brazo hasta su posición exterior extrema y le indica al controlador en hardware que ahora está en el cilindro 0. Normalmente, esto resuelve el problema. Si no sucede así, será necesario reparar la unidad de disco.

Como hemos visto, el controlador en hardware es en realidad una pequeña computadora especializada, completa con software, variables, buffers y, ocasionalmente, errores. A veces una secuencia inusual de sucesos, como la ocurrencia simultánea de una interrupción en una unidad y un comando RECALIBRATE para otra unidad causa un error y hace que el controlador entre en un ciclo u olvide qué es lo que estaba haciendo. Los diseñadores de controladores en hardware generalmente planean pensando en lo peor que pueda suceder e incluyen una pata en el chip que, al recibir voltaje, obliga al controlador a olvidar lo que estaba haciendo y restablecerse. Si todo lo demás falla, el controlador en software del disco puede poner en 1 un bit para invocar esta señal y restablecer el controlador en hardware. Si esto no sirve de nada, todo lo que el controlador en software puede hacer es exhibir un mensaje y darse por vencido.

Almacenamiento en caché de una pista a la vez

El tiempo requerido para llevar el brazo a un nuevo cilindro normalmente es mucho más largo que el retraso rotacional, y siempre es mucho mayor que el tiempo de transferencia. Dicho de otro modo, una vez que el controlador se ha tomado la molestia de mover el brazo a algún lugar, poco importa si se lee un sector o toda una pista. Esto es así sobre todo si el controlador en hardware cuenta con detección rotacional, de modo que el controlador en software puede saber cuál sector está actualmente bajo la cabeza y solicitar el siguiente sector, pudiendo así leer una pista en el

tiempo que tarda una rotación. (Normalmente se requiere media rotación más un tiempo de sector para leer un solo sector, en promedio.)

Algunos controladores de disco en software aprovechan esta propiedad manteniendo un caché secreto donde guardan una pista a la vez; el software independiente del dispositivo no sabe que este caché existe. Si se necesita un sector que está en el caché, no se requerirá una transferencia de disco. Una desventaja del almacenamiento de una pista a la vez en caché (además de la complejidad del software y del espacio de búfer requerido) es que las transferencias del caché al programa invocador deben ser efectuadas por la CPU usando un ciclo programado, en lugar de dejar que el hardware de DMA se encargue del trabajo.

Algunos controladores en hardware llevan este proceso aún más lejos, y guardan una pista a la vez en su propia memoria interna sin que el controlador en software tenga conocimiento de ello; así, la transferencia entre el controlador y la memoria puede utilizar DMA. Si el controlador en hardware trabaja de este modo, no tiene caso hacer que el controlador en software también lo haga. Cabe señalar que tanto el controlador en hardware como en software están en condiciones de leer y escribir pistas completas con un solo comando, pero el software independiente del dispositivo no puede hacerlo, porque considera a un disco como una secuencia lineal de bloques que no está dividida en pistas ni cilindros.

3.7.3 Generalidades del controlador de disco duro en MINIX

El controlador en software del disco duro es la primera parte de MINIX que hemos examinado que necesita tratar con una amplia variedad de tipos de hardware distintos. Antes de entrar en los detalles del controlador, consideraremos brevemente algunos de los problemas que las diferencias de hardware pueden causar. La “IBM PC” es en realidad una familia de computadoras distintas. No sólo se usan diferentes procesadores en diferentes miembros de la familia; también hay diferencias importantes en el hardware básico. Los primeros miembros de la familia, la PC original y la PC-XT, usaban un bus de 8 bits, apropiado para la interfaz externa de 8 bits del procesador 8088. La siguiente generación, la PC-AT, usaba un bus de 16 bits, diseñado ingeniosamente de modo que los periféricos viejos de 8 bits pudieran seguirse usando. Sin embargo, los periféricos más nuevos de 16 bits generalmente no pueden usarse en los sistemas PC-XT más viejos. El bus AT se diseñó originalmente para sistemas que usaban el procesador 80286, y muchos sistemas basados en 80386, 80486 y Pentium usan el bus AT. Sin embargo, dado que estos procesadores más nuevos tienen una interfaz de 32 bits, ya hay varios sistemas con bus de 32 bits disponibles, como el bus PCI de Intel.

Para cada bus hay una familia distinta de adaptadores de E/S, que se conectan a la tarjeta matriz del sistema. Todos los periféricos para un diseño de bus en particular deben ser compatibles con las normas para ese diseño pero no tienen que ser compatibles con diseños más viejos. En la familia IBM PC, al igual que en la mayor parte de los sistemas de computadora más viejos, cada diseño de bus viene acompañado de firmware en la memoria sólo de lectura del sistema básico de E/S (la BIOS ROM) diseñado para salvar la brecha entre el sistema operativo y las peculiaridades del hardware. Algunos dispositivos periféricos pueden incluir extensiones del BIOS en chips de ROM montados en las tarjetas de los mismos dispositivos. La dificultad que enfrenta un

implementador de sistemas operativos es que el BIOS en las computadoras tipo IBM (ciertamente en las primeras) estaba diseñado para un sistema operativo, MS-DOS, que no reconoce la multiprogramación y que se ejecuta en modo real de 16 bits, el mínimo común denominador de los diversos modos de operación de los miembros de la familia de CPU 80x86.

Así, el implementador de un nuevo sistema operativo para la IBM PC enfrenta varias decisiones. Una es si usará el apoyo de controladores en software para periféricos que viene en el BIOS o escribirá nuevos controladores desde cero. Esta decisión no fue difícil en el diseño original de MINIX, ya que el BIOS no era adecuado en muchos aspectos para las necesidades de MINIX. Desde luego, para iniciar, el monitor de arranque de MINIX utiliza el BIOS para efectuar la carga inicial del sistema, sea de un disco duro o un disquete, pues no existe una alternativa práctica. Una vez cargado el sistema, que incluye nuestros propios controladores de E/S, podemos hacer las cosas mucho mejor que con el BIOS.

A continuación enfrentamos la segunda decisión: sin el apoyo del BIOS, ¿cómo vamos a hacer que nuestros controladores en software se adapten a las diversas clases de hardware de los diferentes sistemas? En términos concretos, consideremos que hay por lo menos cuatro tipos fundamentalmente distintos de controladores de disco duro en hardware que podríamos encontrar en un sistema que por lo demás es adecuado para MINIX: el controlador original de 8 bits tipo XT, el controlador de 16 bits tipo AT y dos controladores diferentes para dos tipos distintos de computadoras de la serie IBM PS/2. Hay varias formas posibles de resolver esto:

1. Recompilar una versión distinta del sistema operativo para cada tipo de controlador de disco duro que tengamos que manejar.
2. Compilar varios controladores en software distintos en el kernel y hacer que éste determine automáticamente en el momento de arrancar cuál debe usarse.
3. Compilar varios controladores en software distintos en el kernel y proveer un mecanismo para que el usuario determine cuál se usará.

Como veremos, estas formas no son mutuamente exclusivas.

La primera forma es realmente la mejor a largo plazo. Si se usa en una instalación específica, no hay necesidad de gastar espacio de disco y de memoria en código para otros controladores que nunca se usarán. Sin embargo, este enfoque es una pesadilla para el distribuidor del software, pues proveer cuatro discos de arranque distintos y enseñar a los usuarios a utilizarlos es costoso y difícil. Por ello, es aconsejable usar una de las otras alternativas, al menos para la instalación inicial.

El segundo método consiste en hacer que el sistema operativo sondee los periféricos, ya sea leyendo el ROM de cada tarjeta o escribiendo en los puertos de E/S y leyéndolos a fin de identificar cada tarjeta. Esto es factible en algunos sistemas pero no funciona bien en los sistemas tipo IBM porque hay demasiados dispositivos de E/S no estándar disponibles. En algunos casos, el sondeo de los puertos de E/S para identificar un dispositivo puede activar otro dispositivo que se adueña del control e inhabilita el sistema. Este método complica el código de arranque para cada dispositivo, y aun así, no funciona muy bien. Los sistemas operativos que emplean este método generalmente tienen que proveer algún mecanismo de anulación como el que se usa en MINIX.

El tercer método, empleado en MINIX, consiste en permitir la compilación de varios controladores en software, siendo uno de ellos el predeterminado. El monitor de arranque de MINIX

permite leer diversos parámetros de arranque durante la iniciación, los cuales pueden introducirse a mano o almacenarse permanentemente en el disco. Durante el inicio, si se encuentra un parámetro de arranque de la forma

```
hd=xt
```

se usará el controlador de disco duro XT. Si no se encuentra un parámetro de arranque `hd`, se usa el controlador predeterminado.

Hay otras dos cosas que MINIX hace para tratar de minimizar los problemas causados por la presencia de múltiples controladores de disco duro. Una es proveer un controlador que sirve como interfaz entre MINIX y el apoyo de disco duro de ROM BIOS. Este controlador funciona prácticamente en todos los sistemas y se puede seleccionar usando

```
hd bios
```

como parámetro de arranque. No obstante, esto debe considerarse casi siempre como último recurso. MINIX se ejecuta en modo protegido en los sistemas con procesador 80286 o superior, pero el código del BIOS siempre se ejecuta en modo real (8086). La salida del modo protegido y el regreso posterior a él cada vez que se invoca una rutina del BIOS es algo muy lento.

La otra estrategia que MINIX usa para trabajar con los controladores en software es posponer la inicialización hasta el último momento. De este modo, si en alguna configuración de hardware ninguno de los controladores de disco duro en software funciona, aún podremos iniciar MINIX desde un disco flexible y realizar algo de trabajo útil. MINIX no tendrá problemas en tanto no se trate de acceder al disco duro. Esto tal vez no sea lo máximo en cuanto a amabilidad con el usuario, pero considere lo siguiente: si todos los controladores en software tratan de inicializarse de inmediato durante el arranque del sistema, éste puede paralizarse si algún dispositivo que ni siquiera necesitamos no está bien configurado. Al posponer la inicialización de cada controlador en software hasta que se le necesite, el sistema puede continuar con lo que sí funciona, mientras el usuario trata de resolver los problemas.

Como acotación, aprendimos esta lección e la manera difícil: las primeras versiones de MINIX trataban de inicializar el disco duro tan pronto como se arrancaba el sistema. Si no había disco duro presente, el sistema se paralizaba. Este comportamiento era doblemente molesto porque MINIX puede funcionar perfectamente en un sistema sin disco duro, si bien con capacidad de almacenamiento restringida y menor eficiencia.

En las explicaciones de esta sección y la siguiente, usaremos como modelo el controlador de disco duro tipo AT, que es el controlador en software predeterminado en la distribución de MINIX estándar. Se trata de un controlador versátil que maneja controladores en hardware desde los que se usaban en los primeros sistemas 80286 hasta los modernos controladores EIDE (Extended Integrated Drive Electronics) que manejan discos duros con capacidad de gigabytes. Los aspectos generales del funcionamiento del disco duro que veremos en esta sección se aplican también a los demás controladores en software reconocidos.

El ciclo principal de la tarea del disco duro es el mismo código compartido que ya analizamos, y pueden efectuarse las `sE/S` peticiones estándar. Una petición `DEV_OPEN` puede implicar mucho trabajo, ya que siempre hay particiones y puede haber subparticiones en un disco duro. Estas particiones deben leerse cuando se abre un dispositivo (esto es, cuando se accede a él por primera vez).

Algunos controladores en hardware de disco duro también pueden manejar unidades de CD-ROM, que tienen medios removibles, y cuando se emite una petición `DEV_OPEN` se debe verificar la presencia del medio. En un CD-ROM también tiene significado una operación `DEV_CLOSE`: exige que la puerta se abra y se expulse el disco. Existen otras complicaciones de los medios removibles que son más aplicables a los discos flexibles, así que las veremos en una sección posterior. En el caso del disco duro, la operación `DEV_IOCTL` sirve para establecer una bandera que indica que el medio deberá expulsarse cuando se emita una petición `DEV_CLOSE`. Esta capacidad es útil para los CD ROM, y también se usa para leer y escribir tablas de particiones, como señalamos antes.

Las peticiones `DEV_READ`, `DEV_WRITE` y `SCATTERED_IO` se manejan en tres fases cada una: preparar, planificar y terminar, como ya vimos. El disco duro, a diferencia de los dispositivos de memoria, distingue claramente entre las fases de planificación y terminación. El controlador en software del disco duro no usa SSF ni el algoritmo de elevador, sino que realiza una forma más limitada de planificación, recabando peticiones para sectores consecutivos. Las peticiones normalmente provienen del sistema de archivos MINIX y se refieren a múltiplos de bloques de 1024 bytes, pero el controlador puede atender peticiones de cualquier múltiplo de un sector (512 bytes). En tanto cada petición sea para un sector inicial que sigue inmediatamente al último sector solicitado, cada petición se anexará al final de una lista de peticiones. La lista se mantiene como un arreglo, y cuando se llena, o cuando se solicita un sector no consecutivo, se invoca la rutina de terminación.

En una petición `DEV_READ` o `DEV_WRITE` sencilla, podría solicitarse más de un bloque, pero cada llamada a la rutina de planificación va seguida inmediatamente de una llamada a la rutina de terminación, lo que asegura que se cumpla con la lista de peticiones vigente. En el caso de una petición `SCATTERED_IO`, puede haber múltiples llamadas a la rutina de planificación antes de invocarse la rutina de terminación. En tanto las peticiones sean para bloques de datos consecutivos, la lista se extenderá hasta que el arreglo se llene. Recuerde que en una petición `SCATTERED_IO` una bandera puede indicar que la petición de un bloque en particular es opcional. El controlador en software del disco duro, al igual que el de la memoria, hace caso omiso de la bandera `OPTIONAL` y suministra todos los datos solicitados.

La planificación rudimentaria realizada por el controlador en software del disco duro, posponiendo las transferencias reales mientras se estén solicitando bloques consecutivos, debe verse como el segundo paso de un proceso de planificación que potencialmente tiene tres pasos. El sistema de archivos mismo, al usar E/S dispersa, puede implementar algo similar a la versión de Teory del algoritmo del elevador; recuérdese que, en una petición de E/S dispersa, la lista de peticiones se ordena por número de bloque. El tercer paso de la planificación tiene lugar en el controlador en hardware de un disco duro moderno como el que se describió en la Fig. 3-19. Tales controladores son “inteligentes” y pueden almacenar en buffers grandes cantidades de datos, usando algoritmos programados internamente para recuperar los datos en el orden más eficiente, sea cual sea el orden en que se hayan recibido las peticiones.

3.7.4 Implementación del controlador de disco duro en MINIX

Los discos duros pequeños que se usan en las microcomputadoras a veces se denominan discos “winchester”. Hay varias anécdotas diferentes acerca del origen de este nombre. Al parecer, IBM

usaba este nombre como código para el proyecto que desarrolló la tecnología de disco en la que las cabezas de lectura/escritura vuelan sobre un delgado colchón de aire y se posan sobre el medio de grabación cuando el disco deja de girar. Una explicación del nombre es que uno de los primeros modelos tenía dos módulos de datos: uno fijo de 30 Mbytes y uno removible de 30 Mbytes. Supuestamente, esto recordaba a los diseñadores el rifle Winchester 30-30 que se menciona en muchas narraciones de la frontera oeste de Estados Unidos. Sea cual sea el origen del nombre, la tecnología básica sigue siendo la misma, aunque los discos duros de microcomputadora actuales son mucho más pequeños y su capacidad es mucho mayor que la de los discos de 14 pulgadas que se usaban a principios de la década de 1970 cuando se creó la tecnología winchester.

El archivo `wini.c` se encarga de ocultar del resto del kernel el controlador de disco duro real que se usará. Esto nos permite seguir la estrategia delineada en la sección anterior, compilando varios controladores de disco duro en una sola imagen del kernel y seleccionando el que se usará en el momento del arranque. Posteriormente se puede compilar una instalación a la medida que sólo incluya el controlador que realmente se necesita.

`Wini.c` contiene una definición de datos, `hdmap` (línea 10013), un arreglo que asocia un nombre con la dirección de una función. El compilador inicializa el arreglo con tantos elementos como se necesiten para el número de controladores de disco duro habilitados en `includelnix/config.h`. El arreglo es utilizado por la función `winchester_task`, que es el nombre, contenido en la tabla `task_tab`, que se usa cuando se inicializa el kernel. Cuando se invoca `winchester_task` (línea 10040), trata de encontrar una variable de entorno `hd`, usando una función del kernel que opera de forma parecida al mecanismo empleado por los programas en C ordinarios, leyendo el entorno creado por el monitor de arranque MINIX. Si no se define ningún valor `hd`, se usa la primera entrada del arreglo; de lo contrario, se busca en el arreglo un nombre concordante, y luego se invoca indirectamente la función correspondiente. En el resto de esta sección examinaremos `at_winchester_task`, que es la primera entrada del arreglo `hdmap` en la distribución estándar de MINIX.

El controlador en software para AT está en `at_wini.c` (línea 10100). Éste es un controlador complicado para un dispositivo avanzado, y hay varias páginas de definiciones de macros que especifican registros del controlador en hardware, bits de estado y comandos, estructuras de datos y prototipos. Al igual que en otros controladores de dispositivos por bloques, se inicializa una estructura `driver`, `w_dtab` (líneas 10274 a 10284) con apuntadores a las funciones que realizan el verdadero trabajo. La mayor parte de éstas se definen en `at_wini.c`, pero como el disco duro no requiere operaciones de aseo especiales, su entrada `dr_cleanup` apunta a la función común `nop_cleanup` en `drive.c`, misma que comparten otros dispositivos que no tienen necesidades de aseo especiales. La función de entrada, `at_winchester_task` (línea 10294), invoca un procedimiento que realiza la inicialización específica para el hardware y luego invoca el ciclo principal de `driver.c`. Éste se ejecuta indefinidamente, despachando llamadas a las diversas funciones a las que se apunta en la tabla `driver`,

Puesto que ahora estamos tratando con dispositivos de almacenamiento electromecánicos reales, debemos realizar una cantidad sustancial de trabajo para inicializar el controlador en software del disco duro. Diversos parámetros de los discos duros se mantienen en el arreglo `wini` definido en las líneas 10214 a 10230. Como parte de la política de posponer los pasos de inicialización que pudieran fallar hasta el momento en que se necesiten realmente, `initparams` (línea 10307),

que se invoca durante la inicialización del kernel, no hace nada que requiera acceder al dispositivo del disco mismo. Lo principal que hace es copiar cierta información referente a la configuración lógica del disco duro en el arreglo `wini`. Se trata de información que el TROM BIOS obtiene de la memoria CMOS que las computadoras clase Pentium usan para preservar los datos de configuración básicos. Las acciones del BIOS tienen lugar cuando se enciende inicialmente la computadora, antes de comenzar la primera parte del proceso de carga de MINIX. Si esta información no puede recuperarse, las consecuencias no son necesariamente fatales; si se trata de un disco moderno, la información podrá obtenerse directamente del disco.

Después de la llamada al ciclo principal común, es posible que nada suceda durante algún tiempo hasta que se intente acceder al disco duro. Entonces se recibirá un mensaje solicitando una operación `DEV_OPEN` y se invocará indirectamente `w_do_open` (línea 10355). A su vez, `w_do_open` invoca `w_prepare` para determinar si el dispositivo solicitado es válido, y luego `w_identify` para identificar el tipo de dispositivo e inicializar algunos parámetros más del arreglo `wini`. Por último, se usa un contador del arreglo `wini` para probar si ésta es la primera vez que se ha abierto el dispositivo desde que se inició MINIX. Después de examinarse, el contador se incrementa. Si se trata de la primera operación `DEV...OPEN`, se invoca la función `partition` (de `drvlib.c`).

La siguiente función, `w_prepare` (línea 10388) acepta un argumento entero, `device`, que es el número de dispositivo secundario de la unidad o partición que se usará, y devuelve un apuntador a la estructura `device` que indica la dirección base y el tamaño del dispositivo. En C el empleo de un identificador para nombrar una estructura no impide el uso del mismo identificador para nombrar una variable. Se puede determinar, a partir del número de dispositivo secundario, si un dispositivo es una unidad, una partición o una subpartición. Una vez que `w_prepare` haya completado su trabajo, ninguna de las demás funciones empleadas para leer o escribir en el disco tendrán que preocuparse por la división en particiones. Como hemos visto, se invoca `w_prepare` cuando se hace una petición `DEV_OPEN`; también es una fase del ciclo preparar/planificar/terminar empleado por todas las peticiones de transferencia de datos. En ese contexto, su inicialización de `w_count` en cero es importante.

Los discos tipo AT compatibles por software han estado en uso durante mucho tiempo, y `w_identify` (línea 10415) tiene que distinguir entre varios diseños diferentes que se han introducido a lo largo del tiempo. El primer paso consiste en comprobar que exista un puerto de E/S legible y escribible en el lugar en el que debe haber uno en todos los controladores de disco en hardware de esta familia (líneas 10435 a 10437). Si se satisface esta condición, la dirección del manejador de interrupciones de disco duro se instala en la tabla de descriptores de interrupciones y se inhabilita el controlador de interrupciones para que responda a esa interrupción. Luego se emite un comando `ATA_IDENTIFY` al controlador en hardware del disco. Si el resultado es OK, se obtienen varios elementos de información, incluida una cadena que identifica el modelo del disco y los parámetros físicos de cilindro, cabeza y sector del dispositivo. (Cabe señalar que la configuración “física” informada podría ser diferente de la configuración física verdadera, pero no tenemos más alternativa que aceptar lo que la unidad de disco asegura.) La información del disco también indica si el disco puede o no manejar direccionamiento lineal de bloques (LBA: linear block addressing). Si puede, el controlador en software puede hacer caso omiso de los parámetros de cilindro, cabeza y sector y direccionar el disco usando números de sector absolutos, lo cual es mucho más sencillo.

Como mencionamos antes, es posible que `init_params` no recupere la información de configuración lógica del disco de las tablas del BIOS. Si eso sucede, el código de las líneas 10469 a 10477 intenta crear un conjunto de parámetros apropiado con base en lo que lee de la unidad de disco misma. La idea es que los números de cilindro, cabeza y sector máximos pueden ser 1023, 255 y 63, respectivamente, en virtud del número de bits destinado a cada uno de estos campos en las estructuras de datos originales del BIOS.

Si el comando ATA JIDENTIFY falla, esto puede implicar sencillamente que el disco es un modelo viejo que no reconoce el comando. En este caso lo único que tenemos son los valores de configuración lógica que `init_params` leyó antes. Si tales parámetros son válidos, se copian en los campos de parámetros físicos de `wini`; si no, se devuelve un error y no se puede usar el disco.

Por último, MINIX emplea una variable llamada `u32_t` para contar direcciones en bytes. El tamaño de dispositivo que el controlador en software puede manejar, expresado como número de sectores, debe limitarse si el producto de cilindros x cabezas x sectores es demasiado grande (línea 10490). Aunque en el momento de escribirse este código era raro encontrar dispositivos con capacidad de 4 GB en máquinas que esperaríamos se usaran con MINIX, la experiencia ha demostrado que el software debe escribirse de modo que pruebe límites como éste, por innecesarias que tales pruebas parezcan en el momento de escribirse el código. Luego, se introducen la base y el tamaño de toda la unidad en el arreglo `wini` y se invoca `w_specify`, dos veces si es necesario, a fin de pasar los parámetros que se usarán de vuelta al controlador en software del disco. Por último, se exhiben en la consola el nombre del dispositivo (determinado por `w_name`) y la cadena de identificación obtenida mediante `identify` (si se trata de un dispositivo avanzado) o los parámetros de cilindro, cabeza y sector informados por el BIOS (si se trata de un dispositivo viejo).

`W_name` (línea 10511) devuelve un apuntador a una cadena que contiene el nombre de dispositivo, que será “at-hd0”, “at-hd5”, “at-hd10” o “at-hd15”. `W_specify` (línea 10531), además de pasar los parámetros al controlador, recalibra la unidad de disco (si se trata de un modelo viejo) buscando el cilindro cero.

Ahora estamos listos para analizar las funciones que se invocan al satisfacer una petición de transferencia de datos, `W_prepare`, que ya explicamos, es la que se invoca primero. Su inicialización de la variable `w_count` en cero es importante aquí. La siguiente función que se invoca durante una transferencia es `w_schedule` (línea 10567), la cual establece los parámetros básicos: de dónde provienen los datos, a dónde deben ir, el número de bytes por transferir (que debe ser un múltiplo del tamaño de sector, y se prueba en la línea 10584), y si la transferencia es una lectura o una escritura. El bit que puede estar presente en una petición `SCATTERED_IO` para indicar una transferencia opcional se pone en 0 en el código de operación que se pasará al controlador en hardware (línea 10595), pero obsérvese que se conserva en el campo `io_request` de la estructura `iorequest_s`. En el caso del disco duro, se intenta satisfacer todas las peticiones pero, como veremos, el controlador en software puede decidir posteriormente no hacerlo si han ocurrido errores. La última acción de la preparación es verificar que la petición no rebase el último byte del dispositivo y reducir la petición si así fuera. En este punto se puede calcular el primer sector que se leerá.

En la línea 10602 comienza el proceso de planificación propiamente dicho. Si ya hay peticiones pendientes (lo que se prueba viendo si `w_count` es mayor que cero), y si el sector que se leerá a continuación no es el que sigue al último que se solicitó, se invoca `wjinish` para completar las

peticiones previas. En caso contrario, se actualiza `w_nextblock`, que contiene el número del siguiente sector, y se ingresa en el ciclo de las líneas 10611 a 10640 para agregar nuevas peticiones de sectores al arreglo de peticiones hasta alcanzar el número máximo permisible de peticiones (línea 10614). El límite se guarda en una variable llamada `max_count` pues, como veremos más adelante, hay ocasiones en que resulta útil poder ajustar el límite. Aquí también el resultado puede ser una llamada a `w_finish`.

Como hemos visto, hay dos lugares dentro de `w_prepare` donde puede efectuarse una llamada a `w_finish`. Normalmente, `w_prepare` termina sin invocar `w_finish` pero, sea que se invoque desde `w_prepare` o no, `w_finish` (línea 10649) siempre se invoca tarde o temprano desde el ciclo principal de `driver.c`. Si acaba de invocarse otra vez, es posible que no tenga trabajo que hacer, por lo que se efectúa una prueba en la línea 10659 para comprobar esto. Si todavía hay peticiones en el arreglo de peticiones, se ingresa en la parte principal de `w_finish`.

Como era de esperar, dado que puede haber un número considerable de peticiones encoladas, la parte principal de `w_finish` es un ciclo, en las líneas 10664 a 10761. Antes de entrar en el ciclo, se preestablece la variable `r` en un valor que indica un error, a fin de obligar a la reinicialización del controlador en hardware. Si una llamada a `w_specify` tiene éxito, se inicializa la estructura `command cmd` para realizar una transferencia. Esta estructura sirve para pasar todos los parámetros necesarios a la función que opera realmente el controlador de disco en hardware. Algunas unidades utilizan el parámetro `cmd.precomp` para compensar las diferencias en el rendimiento del medio de grabación magnético cuando hay diferencias en la velocidad con que el medio pasa bajo las cabezas del disco conforme éstas se mueven de los cilindros exteriores a los interiores. Este parámetro siempre es el mismo para una unidad en particular y muchas unidades hacen caso omiso de él. `cmd.count` recibe el número de sectores por transferir, enmascarado para dar una cantidad que quepa en un byte de 8 bits, ya que éste es el tamaño de todos los registros de comandos y estado del controlador en hardware. El código de las líneas 10675 a 10689 especifica el primer sector por transferir, ya sea como un número de bloque lógico de 28 bits (líneas 10676 a 10679) o como parámetros de cilindro, cabeza y sector (líneas 10681 a 10688). En ambos casos se usan los mismos campos de la estructura `cmd`.

Por último se carga el comando mismo, leer o escribir, y se invoca `com_out` en la línea 10692 para iniciar la transferencia. La llamada a `com_out` puede fallar si el controlador en hardware no está listo o no queda listo dentro de un lapso preestablecido. En este caso se incrementa el conteo de errores y se aborta el intento si se alcanza `MAX_ERRORS`. En caso contrario, la instrucción

continúe;

de la línea 10697 hace que el ciclo se inicie otra vez en la línea 10665.

Si el controlador en hardware acepta el comando que se pasa en la llamada a `com_out`, puede pasar cierto tiempo antes de que los datos estén disponibles, así que (suponiendo que el comando es `DEV_READ`) se invoca `w_intr_wait` en la línea 10706. Estudiaremos con detalle esta función más adelante, pero por ahora bastará con tomar nota de que invoca a `receive`, de modo que en este punto la tarea de disco se bloquea.

Cierto tiempo después, largo o corto dependiendo de si fue necesario o no mover el brazo, la llamada a `w_intr_wait` regresará. Este controlador en software no utiliza DMA, aunque algunos

controladores lo reconocen. En vez de ello, se usa E/S programada. Si `w_intr_wait` no devuelve ningún error, la función en lenguaje ensamblador `port_read` tr `SECTOR_SIZE` bytes de datos del puerto de datos del controlador en hardware a su destino final, que debe ser un buffer en el caché de bloques del sistema de archivos. A continuación se ajustan diversas direcciones y contadores para registrar la transferencia realizada con éxito. Por último, si la cuenta de bytes de la petición en curso llega a cero, el apuntador al arreglo de peticiones se avanza de modo que apunte a la siguiente petición (línea 10714).

En el caso de un comando `DEV_WRITE`, la primera parte, es decir, preparar los parámetros del comando y enviar el comando al controlador en hardware, es la misma que para una lectura, excepto por el código de operación del comando. Sin embargo, el orden de los sucesos subsecuentes es diferente para una escritura. Primero hay una espera hasta que el controlador en hardware indica mediante una señal que está listo para recibir datos (línea 10724). `Waitfor` es una macro, y normalmente regresa con gran rapidez. Hablaremos más de esta macro después; por ahora sólo apuntaremos que el tiempo de espera finalmente se vence, y que se espera que las esperas largas sean extremadamente raras. Luego, los datos se transfieren de la memoria al puerto de datos del controlador en hardware usando `port_write` (línea 10729), y en este punto se invoca `w_intr_wait` y la tarea de disco se bloquea. Cuando llega la interrupción y la tarea de disco se despierta, se realiza la contabilización (líneas 10736 a 10739).

Por último, si ha habido errores al leer o escribir, deben manejarse. Si el controlador en hardware informa al controlador en software que el error fue causado por un sector defectuoso, no tiene caso intentarlo otra vez, pero otros tipos de errores sí valen la pena reintentarse, al menos hasta cierto punto, el cual se determina contando los errores y dándose por vencido se se llega a `MAX_ERRORS`. Cuando se llega a `MAX` se invoca `w_need_reset` para forzar la reinicialización cuando se efectúa el reintento. Sin embargo, si la petición originalmente era opcional (hecha con `SCATTERED_IO`), no hay reintento.

Sea que `w` termine sin errores o a causa de un error, siempre se asigna el valor de `CMD_IDLE` a `w_comrnand`. Esto permite a otras funciones determinar que el fallo no se debió a un problema mecánico o eléctrico del disco mismo, impidiendo la generación de una interrupción después de una operación intentada.

El controlador en hardware del disco se maneja mediante un conjunto de registros, que en algunos sistemas pueden tener una correspondencia con la memoria pero que, en las máquinas compatibles con IBM, aparecen como puertos de FIS. Los registros empleados por un controlador en hardware de disco duro de clase IBM-AT se muestran en la Fig. 3-22.

Éste es nuestro primer encuentro con hardware de E/S, y puede resultar útil mencionar algunas de las diferencias de comportamiento entre los puertos de E/S y las direcciones de memoria. En general, los registros de entrada y de salida que tienen la misma dirección de puerto de E/S no son el mismo registro. Por tanto, los datos escritos en una dirección específica no necesariamente pueden recuperarse con una operación de lectura subsecuente. Por ejemplo, la última dirección de registro que se muestra en la Fig. 3-22 indica el estado del controlador en hardware del disco cuando se lee y sirve para emitir comandos al controlador cuando se escribe en ella. También es común que el mero acto de leer o escribir en un registro de dispositivo de FIS cause la realización de una acción, con independencia de los detalles de los datos transferidos. Esto es cierto en el

Registro	Función de lectura	Función de escritura
0	Datos	Datos
1	Error	Precompensación de escritura
2	Cuenta de sectores	Cuenta de sectores
3	Número de sector (0-7)	Número de sector (0-7)
4	Cilindro bajo (8-15)	Cilindro bajo (8-15)
5	Cilindro alto (16-23)	Cilindro alto (16-23)
6	Seleccionar unidad/cabeza (24-27)	Seleccionar unidad/cabeza (24-27)
7	Estado	Comando

(a)

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

LBA: 0 = Modo cilindro/cabeza/sector (CHS)
 1 = Modo de direccionamiento por bloque lógico (LBA)

D: 0 = Unidad maestra
 1 = Unidad esclava

HSn: Modo CHS: Selección de cabeza en modo CHS
 Modo LBA: Bits de selección de bloque 24-27

(b)

Figura 3-22. (a) Registros de control de un controlador en hardware de disco duro IDE. Los números entre paréntesis son los bits de la dirección de bloque lógico seleccionada por cada registro en modo LBA. (b) Campos del registro de selección de unidad/cabeza.

caso del registro de comando del controlador de disco AT. En uso, se escriben datos en los registros de números más bajos para seleccionar la dirección de disco que se leerá o en la que se escribirá, y luego se escribe al final el registro de comando con un código de operación. El acto de escribir el código de operación en el registro de comando inicia la operación.

También sucede que el uso de algunos registros o campos de registros varíe con los diferentes modos de operación. En el ejemplo que se da en la figura, la escritura de un 0 o un 1 en el bit de LBA (bit 6 del registro 6) selecciona el empleo del modo CHS (cilindro-cabeza-sector) o LBA (direccionamiento de bloque lógico). Los datos que se escriben en o se leen de los registros 3, 4 y 5 y los cuatro bits bajos del registro 6 se interpretan de manera diferente dependiendo del valor del bit de LBA.

Veamos ahora cómo se envía un comando al controlador en hardware invocando `com_out` (línea 10771). Antes de modificar ningún registro, se lee el registro de estado para verificar que el controlador no esté ocupado. Esto se hace probando el bit `STATUS_BSY`. Aquí es importante la rapidez, y normalmente el controlador de disco está listo o estará listo en un tiempo corto, por lo que se emplea espera activa. En la línea 10779 se invoca `waitfor` para probar `STATUS_BSY`. A fin de maximizar la rapidez de respuesta, `waitfor` es una macro, definida en la línea 10268. Esta macro realiza la prueba requerida una vez, evitando una costosa llamada de función en la mayor parte de las invocaciones, cuando el disco está listo. En las raras ocasiones en que es necesaria una espera, la macro llama a `w_waitfor`, que ejecuta la prueba en un ciclo hasta que el resultado es verdadero o se vence un periodo de espera predefinido. Así, el valor devuelto será verdadero con el retraso mínimo posible si el controlador está listo, verdadero después de un retraso si temporalmente no está disponible, o falso si no está listo después del periodo de espera. Hablaremos más acerca del tiempo de espera cuando estudiemos `w_waitfor` en sí.

Un controlador en hardware puede manejar más de una unidad de disco, así que una vez que se determina que el controlador está listo se escribe un byte para seleccionar la unidad, la cabeza y el modo de operación (línea 10785) y luego se invoca otra vez `waitfor`. Hay ocasiones en que una unidad de disco no lleva a cabo un comando ni devuelve un código de error correcto—después de todo, es un dispositivo mecánico que se puede atorar, atascar o descomponer internamente— y como seguro se envía un mensaje a la tarea de reloj para planificar una llamada a una rutina de despertar. Después de esto, se emite el comando escribiendo primero todos los parámetros en los diversos registros y, finalmente, el código del comando mismo en el registro de comando. Este último paso y la subsecuente modificación de las variables `w_command` y `w_status` forman una sección crítica, así que toda la secuencia está flanqueada por llamadas a `lock` y `unlock` (líneas 10801 a 10805) que inhabilitan y luego habilitan las interrupciones.

Las siguientes funciones son cortas. Ya vimos que `w_need_reset` (línea 10813) es invocada por `w_finish` cuando el conteo de fallas llega a la mitad de `MAX_ERRORS`; también se invoca cuando se vencen tiempos de espera para que el disco genere una interrupción o quede listo. La acción de `w_need_reset` sólo consiste en marcar la variable `state` para cada unidad del arreglo `wini`, a fin de forzar la inicialización en el siguiente acceso.

`W_do_close` (línea 10828) tiene muy poco que hacer en el caso de un disco duro convencional. Cuando se agrega apoyo de `CD_ROM` u otros dispositivos removibles, esta rutina tiene que extenderse para generar un comando que suelte el seguro de la puerta o expulse el CD, dependiendo de lo que el hardware maneje.

Se invoca `com_simple` para emitir comandos del controlador en hardware que obligan a una terminación inmediata sin una fase de transferencia de datos. Los comandos que caen en esta categoría incluyen los que obtienen la identificación del disco, establecen algunos parámetros y recalibran.

Cuando `com_out` invoca la tarea de reloj como preparación para un posible rescate después de un fallo del controlador en hardware del disco, pasa la dirección de `w_timeout` (línea 10858) como función que la tarea del reloj debe despertar cuando expire el periodo de espera. Por lo regular, el disco completa la operación solicitada y, al vencerse el tiempo de espera, se observa que `w_command` tiene el valor `CMD_IDLE`, lo que implica que el disco completó su operación, y `w_timeout` ya puede terminar. Si el comando no se lleva a cabo y la operación es una lectura o

escritura, puede ser útil reducir el tamaño de las peticiones 4e E/S. Esto se hace en dos pasos, reduciendo primero a 8 el número máximo de sectores que es posible solicitar, y luego a 1. Siempre que se vence un tiempo de espera, se exhibe un mensaje, se invoca `w_need_reset` para forzar la reinicialización de todas las unidades en el siguiente intento de acceso, y se invoca `interrupt` para enviar un mensaje a la tarea de disco y simular la interrupción generada por hardware que debió haber ocurrido al término de la operación de disco.

Cuando se requiere un restablecimiento, se invoca `w_reset` (línea 10889). Esta función utiliza una función provista por el controlador del reloj, `milli-delay`. Después de un retraso inicial para dar a la unidad de disco tiempo de recuperarse de operaciones anteriores, se parpadea un bit del registro de control del controlador en hardware del disco; es decir, se lleva a un nivel 1 lógico durante un periodo definido y luego se regresa al nivel 0 lógico. Después de esta operación, se invoca `waitfor` para dar a la unidad un tiempo razonable para indicar mediante una señal que está lista. Si el restablecimiento no tiene éxito, se exhibe un mensaje y se devuelve un estado de error. Corresponde al invocador decidir qué se hará después.

Los comandos al disco que implican transferencia de datos normalmente terminan generando una interrupción, que envía un mensaje de vuelta a la tarea de disco. De hecho, se genera una interrupción por cada sector leído o escrito. Así, después de emitirse un comando de este tipo, siempre se invoca `w_intr_wait` (línea 10925). A su vez, `w_intr_wait` invoca `receive` en un ciclo, ignorando el contenido de cada mensaje, esperando una interrupción que asigne a `w_status` el estado de “ocioso”. Una vez recibido tal mensaje, se verifica el estado de la petición. Ésta es otra sección crítica, así que se usan `lock` y `unlock` para garantizar que no ocurrirá una nueva interrupción que modifique `w_status` antes de que se lleven a cabo los diversos pasos requeridos.

Hemos visto varios lugares en los que se invoca la macro `wai` para realizar espera activa hasta que cambia un bit en el registro de estado del controlador en hardware del disco. Después de la prueba inicial, la macro `waitfor` invoca `w_wai` (línea 10955), que invoca `milli_start` para iniciar un temporizador y luego ingresar en un ciclo que alternadamente verifica el registro de estado y el temporizador. Si se vence un tiempo de espera, se invoca `w_need_reset` a fin de preparar las cosas para un restablecimiento del controlador en hardware del disco la próxima vez que se soliciten sus servicios.

El parámetro `TIMEOUT` utilizado por `w_waitfor` se define en la línea 10206 como 32 segundos. Un parámetro similar, `WAKEUP` (línea 10193) empleado para planificar las acciones de despertar de la tarea de reloj, se ajusta a 31 segundos. Éstos son periodos muy largos para permanecer en espera activa, si consideramos que un proceso ordinario sólo recibe una tajada de tiempo de 100 ms para ejecutarse antes de ser expulsado. Sin embargo, estos números se basan en la norma publicada para conectar dispositivos de disco a computadoras de clase AT, que indica que se deben contemplar hasta 31 segundos para que un disco se acelere hasta adquirir la velocidad correcta. Desde luego, esta especificación contempla el peor caso, y además en la mayor parte de los sistemas el disco sólo se acelera cuando la computadora se enciende o después de periodos largos de inactividad. MINIX todavía se está desarrollando. Es posible que se requiera una nueva forma de manejar los tiempos de espera cuando se agregue apoyo para `CD_ROM` (u otros dispositivos que deben acelerarse con frecuencia).

`W_handler` (línea 10976) es el manejador de interrupciones. `W_identify` coloca la dirección de esta función en la tabla de descriptores de interrupciones cuando se activa inicialmente la tarea

del disco duro. Cuando ocurre una interrupción de disco, el registro de estado del controlador en hardware del disco se copia en `w_status` y luego se invoca la función `interrupt` del kernel para replanificar la tarea del disco duro. Claro que cuando esto ocurre la tarea del disco duro ya está bloqueada como consecuencia de una llamada previa a `receive` efectuada por `w_intr_wait` después del inicio de una operación de disco.

La última función de `at_wini.c` es `w_geometry`, la cual devuelve los valores lógicos máximos de cilindro, cabeza y sector del dispositivo de disco duro seleccionado. En este caso los números son reales, no inventados como en el caso del controlador de disco en RAM.

3.7.5 Manejo de discos flexibles

El controlador en software del disco flexible es más largo y complicado que el del disco duro. Esto podría parecer paradójico, pues podríamos pensar que los mecanismos de disco flexible son más sencillos que los de disco duro. Sin embargo, al ser más sencillo el mecanismo su controlador en hardware también es más simple y requiere más atención por parte del sistema operativo. Además, el hecho de que el medio es removible agrega ciertas complicaciones. En esta sección describiremos algunas de las cosas que un implementador debe considerar al manejar discos flexibles, pero no entraremos en los detalles del código del controlador de disco flexible en MINIX. Las partes más importantes son similares a las del disco duro.

Una de las cosas por las que no necesitamos preocuparnos en el caso del controlador en software de disco flexible son los múltiples tipos de controladores en hardware que tuvimos que apoyar en el caso del controlador de disco duro. Aunque los discos flexibles de alta densidad que se usan actualmente no se contemplaban cuando se diseñó la IBM PC original, los controladores en hardware de disco flexible de todas las computadoras de la familia IBM PC son apoyados por un solo controlador en software. El contraste con la situación del disco duro probablemente se debe a la falta de presión por aumentar el rendimiento de los discos flexibles. Los disquetes casi nunca se usan como almacenamiento de trabajo durante la operación de un sistema de computadora; su rapidez y capacidad de datos son demasiado limitadas en comparación con las de los discos duros. Los discos flexibles siguen siendo importantes para la distribución de software nuevo y para respaldo, y por ello casi todos los sistemas de computadora pequeños están equipados con, por lo menos, una unidad de disco flexible.

El controlador en software de disco flexible no usa SSF ni el algoritmo del elevador; es estrictamente secuencial. El controlador acepta una petición y la ejecuta antes de siquiera aceptar otra petición. Al diseñar originalmente MINIX se pensó que, como el sistema operativo estaba destinado a usarse en computadoras personales, la mayor parte del tiempo sólo habría un proceso activo, y la posibilidad de que llegara otra petición de disco mientras se estaba ejecutando una era pequeña. Por tanto, no se justificaba el considerable aumento en la complejidad del software que se requeriría para poner en cola las peticiones. Hoy día tal aumento es menos justificable aún, pues los discos flexibles raras veces se usan para otra cosa que no sea la transferencia de datos desde o hacia un sistema provisto con un disco duro.

Habiendo dicho esto, a pesar de que el controlador en software no ofrece apoyo para reordenar las peticiones, el controlador de disquete, como cualquier otro controlador por bloques, puede

atender una petición de E/S dispersa y, al igual que el controlador de disco duro, acumula las peticiones en un arreglo y las seguirá acumulando en tanto se sigan solicitando sectores secuenciales. Sin embargo, en el caso del controlador de disquete el arreglo de peticiones es más pequeño que para el disco duro, y está limitado al número máximo de sectores por pista de un disquete. Además, el controlador de disquete examina la bandera OPTIONAL en las peticiones de E/S dispersa y no procede a una nueva pista si todas las peticiones vigentes son opcionales.

La sencillez del hardware del disco flexible es la causa de algunas de las complicaciones del controlador en software del disquete. Las unidades de disquete económicas, lentas y de baja capacidad no justifican los complejos controladores en hardware integrados que forman parte de las unidades de disco duro modernas, así que el controlador en software debe manejar explícitamente aspectos del funcionamiento del disco que están ocultos durante la operación de un disco duro. Como ejemplo de complicación causada por la sencillez de las unidades de disquete, considere la colocación de la cabeza de lectura/escritura sobre una pista específica durante una operación SEEK (de búsqueda). Ningún disco duro exige al controlador en software invocar específicamente SEEK. En el caso de un disco duro, la geometría de cilindro, cabeza y sector visible para el programador podría no corresponder a la geometría física y, de hecho, es posible que la geometría física sea muy complicada, con más sectores en los cilindros exteriores que en los interiores. Sin embargo, el usuario no percibe esto. Los discos duros pueden aceptar direccionamiento por bloques lógicos (LBA), usando el número de sector absoluto del disco, como alternativa al direccionamiento por cilindro, cabeza y sector. Incluso si el direccionamiento se efectúa por cilindro, cabeza y sector, se puede usar cualquier geometría que no haga referencia a sectores inexistentes, ya que el controlador en hardware integrado al disco calcula la posición a la que deben moverse las cabezas de lectura/escritura y realiza una operación de búsqueda si es necesario.

En cambio, en el caso de un disco flexible, es necesario programar explícitamente las operaciones SEEK. En caso de fallar un SEEK, es preciso contar con una rutina que realice una operación RECALIBRATE que obligue a las cabezas a colocarse en el cilindro 0. Esto permite al controlador en hardware llevarlas a una posición de pista deseada avanzando las cabezas cierto número de pasos. Desde luego, se requieren operaciones similares para el disco duro, pero el controlador en hardware de la unidad se encarga de ellas sin que el controlador en software tenga que guiarlo detalladamente.

Entre las características de las unidades de disco flexible que hacen que su controlador en software sea complicado están:

1. Medios removibles.
2. Múltiples formatos de disco.
3. Control del motor.

Algunos controladores en hardware de disco duro contemplan medios removibles (por ejemplo, en una unidad de CD-ROM), pero en general el controlador en hardware de la unidad puede manejar cualquier complicación sin mucho apoyo por parte del controlador en software del dispositivo. En el caso del disco flexible, en cambio, el apoyo integrado no existe, a pesar de que se le necesita aún más. Algunos de los usos más comunes de los discos flexibles —instalar software

nuevo o respaldar archivos— suelen requerir el cambio de un disco a otro en la unidad. Puede haber problemas graves si los datos que debían ir en un disquete se escriben en otro. El controlador en software de la unidad debe hacer todo lo que pueda para evitar esto, pero en muchos casos no es posible hacer mucho, ya que el hardware no permite determinar si la puerta de la unidad se abrió o no desde el último acceso. Otro problema con los medios removibles es que el sistema puede paralizarse si intenta acceder a una unidad de disco flexible que actualmente no tiene un disquete insertado. Esto puede resolverse si es posible detectar que la puerta está abierta, pero como éste no siempre es el caso, es necesario tomar medidas para detectar la expiración de un tiempo de espera y devolver un error si una operación de disquete no termina en un tiempo razonable.

Los medios removibles pueden ser sustituidos por otros medios, y en el caso de los discos flexibles pueden tener muchos formatos distintos. El hardware en que se usa MINIX reconoce unidades de disco tanto de 3.5 pulgadas como de 5.25 pulgadas, y los disquetes pueden formatearse de diversas maneras para contener desde 360 KB hasta 1.2 MB (en un disquete de 5.25 pulgadas) o 1.44 MB (en un disquete de 3.5 pulgadas). MINIX reconoce siete formatos de disco flexible diferentes. Hay dos posibles soluciones al problema que esto causa, y MINIX contempla ambas. Una forma es referirse a cada posible formato como una unidad distinta y proveer múltiples dispositivos secundarios. MINIX hace esto, y en el directorio de dispositivos encontramos definiciones de 14 dispositivos distintos que van desde `/dev/pco`, un disquete de 5.25 pulgadas de 360K en la primera unidad hasta `/dev/psi`, un disquete de 3.5 pulgadas de 1.44M en la segunda unidad. No es fácil recordar las diferentes combinaciones y, por ello, se ofrece una alternativa. Cuando nos referimos a la primera unidad de disco flexible como `/dev/fdo`, o a la segunda como `/dev/fdl`, el controlador en software prueba el disquete que está actualmente en la unidad cuando se accede a él, a fin de determinar el formato. Algunos formatos tienen más cilindros, y otros tienen más sectores por pista que otros formatos. La determinación del formato de un disquete se efectúa intentando leer los sectores y pistas con los números más altos. Un proceso de eliminación permite determinar el formato. Desde luego, esto toma tiempo, y cabe la posibilidad de identificar erróneamente un disquete que tiene sectores defectuosos.

La última complicación del controlador en software para disco flexible consiste en el control del motor. No es posible leer disquetes o escribir en ellos si no están girando. Los discos duros están diseñados para funcionar durante miles de horas sin desgastarse, pero si mantenemos en operación los motores todo el tiempo la unidad de disquete y el disquete se desgastan rápidamente. Si el motor no está encendido ya cuando se accede a una unidad, es necesario emitir un comando que arranque la unidad y esperar cerca de medio segundo antes de intentar leer o escribir datos. El encendido o apagado del motor es lento, así que MINIX deja el motor de la unidad encendido durante varios segundos después de que se usa una unidad. Si la unidad se usa otra vez dentro de este intervalo, el temporizador se extiende otros segundos más. Si la unidad no se utiliza en dicho intervalo, el motor se apaga.

3.8 RELOJES

Los relojes (también llamados temporizadores) son esenciales para el funcionamiento de cualquier sistema de tiempo compartido por diversas razones. Entre otras cosas, los relojes mantienen la

hora del día y evitan que un proceso monopolice la CPU. El software del reloj puede adoptar la forma de un controlador de dispositivo, aunque el reloj no es un dispositivo por bloques, como un disco, ni por caracteres, como una terminal. Nuestro estudio de los relojes seguirá el mismo patrón que en las secciones anteriores: primero examinaremos el hardware y el software de reloj en general, y luego estudiaremos de cerca la forma en que tales ideas se aplican a MINIX.

3.8.1 Hardware de reloj

Se usan comúnmente dos tipos de relojes en las computadoras, y ambos son muy diferentes de los relojes que la gente usa. Los relojes más sencillos están conectados a la línea de potencia de 1100 220 volts, y causan una interrupción a cada ciclo de voltaje, a 50 o 60 Hz.

La otra clase de reloj consta de tres componentes: un oscilador de cristal, un contador y un registro de retención, como se aprecia en la Fig. 3-23. Si un cristal de cuarzo se corta correctamente y se monta sometido a tensión, puede generar una señal periódica de gran exactitud, por lo regular en el intervalo de 5 a 100 MHz dependiendo del cristal elegido. Toda computadora incluye al menos un circuito de este tipo, el cual proporciona una señal de sincronización a los diferentes circuitos de la computadora. Esta señal se alimenta al contador para hacer que realice una cuenta regresiva. Cuando el contador llega a cero, causa una interrupción de la CPU.



Figura 3-23. Reloj programable.

Los relojes programables suelen tener varios modos de operación. En el modo de una acción, cuando se inicia el reloj, éste copia el valor del registro de retención en el contador y luego decrementa el contador en cada pulso del cristal. Cuando el contador llega a cero, causa una interrupción y se detiene hasta que el software lo inicia otra vez explícitamente. En el modo de onda cuadrada, después de que el contador llega a cero y causa la interrupción, el registro de retención se copia en el contador y todo el proceso se repite indefinidamente. Estas interrupciones periódicas se denominan tics de reloj.

La ventaja del reloj programable es que su frecuencia de interrupción puede controlarse por software. Si se emplea un cristal de 1 MHz, el contador cambiará cada microsegundo. Si los registros son de 16 bits, podremos programar interrupciones que ocurran a intervalos desde 1 microsegundo hasta 65.536 ms. Los chips de reloj programable generalmente contienen dos o tres

relojes independientemente programables, además de muchas otras opciones (p. ej., contar hacia arriba en lugar de hacia abajo, interrupciones inhabilitadas, etcétera).

A fin de evitar que la hora actual se pierda cuando la computadora se apaga, la mayor parte de las computadoras tienen un reloj de respaldo de baterías, implementado con los tipos de circuitos de baja potencia empleados en los relojes de pulsera digitales. El reloj de baterías puede leerse en el momento del arranque. Si no hay reloj de respaldo presente, el software podría preguntar al usuario la hora y la fecha actuales. También hay un protocolo estándar para que un sistema conectado a una red obtenga la hora actual de un anfitrión remoto. En cualquier caso, la hora se traduce al número de tics de reloj transcurridos desde las 12 A.M., Tiempo Universal Coordinado (UTC) (antes conocido como hora del meridiano de Greenwich) del 1.º de enero de 1970, como hacen UNIX y MINIX, o a partir de otra hora de referencia. En cada tic del reloj, el tiempo real se incrementa en uno. Por lo regular se cuenta con programas de utilidad para establecer manualmente el reloj del sistema y el reloj de respaldo y para sincronizar los dos relojes.

3.8.2 Software de reloj

Todo lo que el hardware de reloj hace es generar interrupciones a intervalos conocidos. Todo lo demás relacionado con el tiempo corre por cuenta del software, el controlador del reloj. Las obligaciones exactas del controlador del reloj varían de un sistema operativo a otro, pero casi siempre incluyen las siguientes:

1. Mantener la hora del día.
2. Evitar que los procesos se ejecuten durante más tiempo del debido.
3. Contabilizar la utilización de la CPU.
4. Manejar la llamada al sistema ALARM emitida por procesos de usuario.
5. Proveer temporizadores de vigilancia a partes del sistema mismo.
6. Preparar perfiles, vigilar y recabar datos estadísticos.

La primera función del reloj, mantener la hora del día (también llamada tiempo real) no es difícil; sólo requiere incrementar un contador en cada tic del reloj, como se mencionó antes. Lo único que debe cuidarse es el número de bits que tiene el contador de la hora del día. Con una tasa de reloj de 60 Hz, un contador de 32 bits se desbordará en poco más de dos años. Es evidente que el sistema no puede almacenar en 32 bits el tiempo real como el número de tics transcurridos desde el 1.º de enero de 1970.

Se pueden adoptar tres estrategias para resolver este problema. La primera consiste en usar un contador de 64 bits, aunque esto hace que el mantenimiento del contador sea más costoso, pues tiene que modificarse muchas veces cada segundo. La segunda forma consiste en mantener la hora del día en segundos, utilizando un contador subsidiario para contar tics hasta acumular un

segundo completo. Dado que 232 segundos es más de 136 años, este método funcionará hasta bien entrado el siglo xx

La tercera estrategia es contar en tics, pero hacerlo relativo al momento en que se arrancó el sistema, no a un momento externo fijo. Cuando se lee el reloj de respaldo o el usuario introduce el tiempo real, se calcula el momento de arranque del sistema a partir del valor de hora del día actual y se almacena en la memoria en cualquier forma conveniente. Más adelante, cuando se solicite la hora del día, la hora almacenada se sumará al contador para obtener la hora del día vigente. Estos tres enfoques se muestran en la Fig. 3-24.

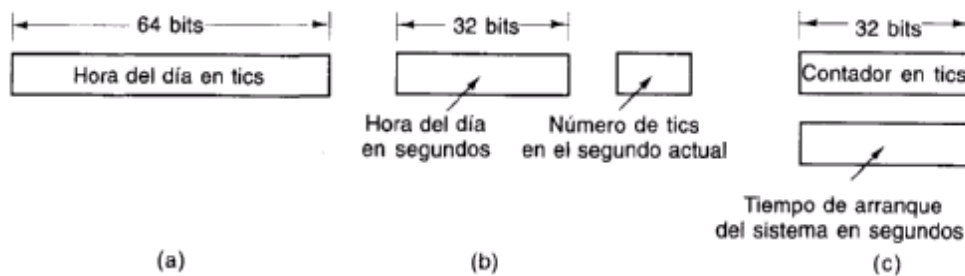


Figura 3-24. Tres formas de mantener la hora del día.

La segunda función del reloj es evitar que los procesos se ejecuten durante demasiado tiempo. Cada vez que se inicia un proceso, el planificador debe inicializar un contador con el valor del cuanto de ese proceso expresado en tics del reloj. En cada interrupción del reloj, el controlador del reloj decrementará el contador de cuanto en 1. Cuando el contador llegue a cero, el controlador del reloj llamará al planificador para que ponga en marcha otro proceso.

La tercera función del reloj es contabilizar el uso de la CPU. La forma más exacta de hacer esto es iniciar un segundo temporizador, distinto del temporizador principal del sistema, cada vez que se inicia un proceso. Cuando se detiene ese proceso, se puede leer el temporizador para determinar durante cuánto tiempo se ejecutó el proceso. Para hacer bien las cosas, el segundo temporizador debe guardarse cada vez que ocurre una interrupción, y restablecerse después.

Una forma menos exacta, pero mucho más sencilla de llevar la contabilidad es mantener en una variable global un apuntador a la entrada de la tabla de procesos correspondiente al proceso que se está ejecutando. En cada tic del reloj, se incrementa un campo de la entrada del proceso en curso. De esta forma, cada tic del reloj se "cobra" al proceso que se estaba ejecutando en el momento del tic. Un problema menor de esta estrategia es que si ocurren muchas interrupciones durante la ejecución de un proceso, de todos modos se le cobrará un tic completo, aunque no haya podido realizar mucho trabajo. La contabilización correcta de la CPU durante las interrupciones es muy costosa y nunca se efectúa.

En MINIX y muchos otros sistemas, un proceso puede solicitar que el sistema operativo le envíe un aviso después de cierto intervalo. El aviso casi siempre es una señal, interrupción, mensaje o algo similar. Una aplicación que requiere tales avisos es el trabajo en redes, en el que un paquete del cual no se ha acusado recibo dentro de cierto intervalo de tiempo debe retransmitirse.

Otra aplicación es la enseñanza asistida por computadora donde, si un estudiante no proporciona la respuesta dentro de cierto tiempo, recibe la respuesta del programa.

Si el controlador de reloj tuviera suficientes relojes, podría establecer un reloj individual para cada petición. Como no sucede así, es preciso simular varios relojes virtuales con un solo reloj físico. Una forma de hacerlo es mantener una tabla en la que se guarda el tiempo de señal para todos los temporizadores pendientes, además de una variable que indica el tiempo del siguiente. Cada vez que se actualiza la hora del día, el controlador determina si ya ocurrió la señal más cercana. Si así fue, se busca en la tabla la siguiente señal que ocurrirá.

Si se esperan muchas señales, resulta más eficiente simular múltiples relojes encadenando todas las peticiones de reloj pendientes, ordenadas por tiempo, en una lista enlazada, como se muestra en la Fig. 3-25. Cada entrada de la lista indica cuántos tics del reloj hay que esperar después de la anterior antes de causar una señal. En este ejemplo, hay señales pendientes para 4203, 4207, 4213, 4215 y 4216.

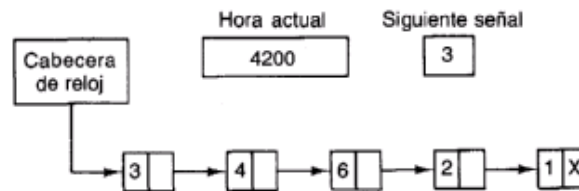


Figura 3-25. Simulación de múltiples temporizadores con un solo reloj.

En la Fig. 3-25, la siguiente interrupción ocurrirá en 3 tics. En cada tic, se decrementa la siguiente señal. Cuando ésta llega a cero, se causa la señal correspondiente al primer elemento de la lista, y dicho elemento se elimina de la lista. A continuación se asigna a Siguiente señal el valor de la entrada que ahora está a la cabeza de la lista, que en este ejemplo es 4.

Observe que durante una interrupción de reloj el controlador de reloj tiene varias cosas que hacer: incrementar el tiempo real, decrementar el cuanto y verificar si es 0, realizar la contabilización de la CPU y decrementar el contador de la alarma. No obstante, cada una de estas operaciones se ha dispuesto cuidadosamente de modo que sea muy rápida, ya que deben repetirse muchas veces cada segundo.

Algunas partes del sistema operativo también necesitan establecer temporizadores llamados temporizadores de vigilancia. Al estudiar el controlador del disco duro vimos que se planifica una llamada de despertar cada vez que se envía un comando al controlador en hardware del disco, de modo que pueda intentarse la recuperación si el comando falta por completo. También mencionamos que los controladores en software de disco flexible deben esperar hasta que el motor del disco adquiere la velocidad correcta, y deben apagar el motor si no ocurre actividad durante cierto tiempo. Algunas impresoras provistas de cabeza de impresión móvil pueden imprimir 120 caracteres por segundo (8.3 ms/carácter), pero no pueden regresar la cabeza de impresión al margen izquierdo en 8.3 ms, así que el controlador de la terminal debe esperar después de que se tecléa un retomo de carro.

El mecanismo empleado por el controlador del reloj para manejar los temporizadores de vigilancia es el mismo que se emplea para las señales de usuario. La única diferencia es que cuando un

temporizador termina, el controlador del reloj, en lugar de causar una señal, invoca un procedimiento proporcionado por el invocador. El procedimiento forma parte del código del invocador, pero dado que todos los controladores están en el mismo espacio de direcciones, el controlador del reloj también puede invocarlo. El procedimiento invocado puede hacer todo lo necesario, incluso causar una interrupción, aunque dentro del kernel las interrupciones no suelen ser recomendables y las señales no existen. Es por esto que se proporciona el mecanismo de vigilancia.

El último elemento de nuestra lista es la preparación de perfiles. Algunos sistemas operativos cuentan con un mecanismo mediante el cual un programa de usuario puede hacer que el sistema prepare un histograma de su contador de programa, a fin de ver a qué está dedicando su tiempo. Cuando se van a preparar perfiles, el controlador verifica en cada tic si se está preparando un perfil del proceso actual y, de ser así, calcula el número de gaveta (un intervalo de direcciones) correspondiente al contador de programa actual. A continuación, el controlador incrementa esa gaveta en uno. Este mecanismo también puede servir para preparar perfiles del sistema mismo.

3.8.3 Generalidades del controlador de reloj en MINIX

El controlador de reloj de MINIX está contenido en el archivo `clock.c`. La tarea de reloj acepta estos sE/S tipos de mensajes, con los parámetros que se indican:

1. `HARD_INT`
2. `GET_UPTIME`
3. `GET_TIME`
4. `SET_TIME` (nuevo tiempo en segundos)
5. `SET_ALARM` (número de proceso, procedimiento por invocar, retraso)
6. `SET_SYN_AL` (número de proceso, retraso)

`HARD_INT` es el mensaje que se envía al controlador cuando ocurre una interrupción de reloj y hay trabajo que realizar, como cuando debe enviarse una alarma o un proceso se ha ejecutado durante demasiado tiempo.

`GETUP TIME` sirve para obtener el tiempo en tics desde el momento del arranque. `GET_TIME` devuelve el tiempo real actual como el número de segundos transcurridos desde el 1.º de enero de 1970 a las 12 A.M., y `SETTIME` fija el tiempo real. Este mensaje sólo puede ser enviado por el Superusuario.

Dentro del controlador del reloj, se lleva el tiempo usando el método de la Fig. 3-24(c). Cuando se fija la hora, el controlador calcula el tiempo desde el arranque del sistema. El controlador puede efectuar este cálculo porque tiene el tiempo real actual y sabe durante cuántos tics el sistema ha estado funcionando. El sistema almacena el tiempo real del arranque en una variable. Después, cuando se invoca `GET_TIME`, el sistema convierte el valor actual del contador de tics a segundos y lo suma al tiempo desde el arranque que tiene almacenado.

`SET_ALARM` permite a un proceso establecer un temporizador que “suena” en cierto número de tics del reloj. Cuando un proceso de usuario realiza una llamada `ALARM`, envía un mensaje al

administrador de memoria, el cual a su vez envía ese mensaje al controlador de reloj. Cuando la alarma “suena”, el controlador del reloj envía un mensaje de vuelta al administrador de memoria, que entonces se encarga de que se envíe la señal.

SET_ALARM también es utilizado por tareas que necesitan iniciar un temporizador de vigi lancia. Cuando el temporizador se vence, simplemente se invoca el procedimiento proporcionado. El controlador del reloj no tiene idea de qué hace el procedimiento.

SET_SYN_ALARM es similar a SET_ALARM, pero se usa para establecer una alarma síncrona. Una alarma síncrona envía un mensaje a un proceso, en lugar de generar una señal o invocar un procedimiento. La tarea de alarma síncrona se encarga de enviar mensajes a los procesos que los requieren. Veremos las alarmas síncronas con mayor detalle posteriormente.

La tarea de reloj no utiliza estructuras de datos importantes, pero se emplean algunas variables para llevar el tiempo. Sólo una de ellas es una variable global, `lost_ticks`, definida en `glo.h` (línea 5031). Se incluye esta variable para que la use cualquier controlador que pudiera agregarse a MINIX en el futuro y que pudiera inhabilitar las interrupciones durante un tiempo tan largo que pudieran perderse uno o más ticks del reloj. Por ahora no se emplea esta variable, pero si se escribiera un controlador semejante el programador podría hacer que se incrementa `lost_ticks` para compensar el tiempo durante el cual se inhibieron las interrupciones de reloj.

Obviamente, las interrupciones de reloj ocurren con mucha frecuencia, y es importante manejarlas rápidamente. MINIX logra esto realizando el mínimo de procesamiento en la mayor parte de las interrupciones de reloj. Al recibir una interrupción, el manejador asigna el valor de `lost_ticks + 1` a una variable local, `ticks`, y luego usa esta cantidad para actualizar los tiempos de contabilización y `pending_ticks` (línea 11079); además, vuelve a poner `lost_ticks` en cero. `Pending_ticks` (tics pendientes) es una variable PRIVILEGE, declarada fuera de todas las definiciones de funciones pero conocida sólo para las funciones definidas en `clock.c`. Otra variable PRIVILEGE, `sched_ticks`, se decrementa en cada tick para llevar un registro del tiempo de ejecución. El manejador de interrupciones envía un mensaje a la tarea del reloj sólo si se vence una alarma o si se acaba un cuanto de ejecución. Este esquema permite que el manejador de interrupciones regrese casi de inmediato en la mayor parte de las interrupciones.

Cuando la tarea del reloj recibe un mensaje, suma `pending_ticks` a la variable `realtime` (línea 11067) y luego pone en ceros `pending_ticks`. `Realtime`, junto con la variable `boot_time` (línea 11068) permite calcular la hora del día actual. Las dos son variables PRIVILEGE, así que la única forma que tiene cualquier otra parte del sistema de obtener la hora es enviando un mensaje a la tarea del reloj. Aunque en un instante dado `realtime` puede ser inexacta, este mecanismo asegura que siempre será exacta cuando se necesite. Si nuestro reloj de pulsera marca la hora correcta cuando lo miramos, ¿qué importa si no marca la hora correcta cuando no lo estamos viendo?

Para manejar las alarmas, `next_alarm` registra el tiempo en el que la siguiente señal o llamada de vigilancia puede ocurrir. El controlador debe tener cuidado aquí, porque el proceso que solicita la señal podría llegar a su fin o ser terminado antes de que la señal suceda. Cuando llega el momento de la señal, se verifica si todavía se necesita. Si no se necesita, no se genera.

Un proceso de usuario sólo puede tener un temporizador de alarma vigente. La ejecución de una llamada `ALARM` mientras el temporizador está corriendo cancela el primer temporizador. Por tanto, una forma cómoda de almacenar los temporizadores consiste en reservar una palabra en la

entrada de tabla de procesos de cada proceso para su temporizador, si existe. En el caso de las tareas, también debe almacenarse en algún lado la función que se invocará, y para este fin se cuenta con el arreglo `watch_dog`. Un arreglo similar, `syn_table`, almacena banderas que indican, para cada proceso, si va a recibir o no una alarma síncrona.

La lógica global del controlador del reloj sigue el mismo patrón que los controladores de disco. El programa principal es un ciclo infinito que obtiene mensajes, realiza acciones dependiendo del tipo de mensajes, y luego envía una respuesta (excepto en el caso de `CLOCK_TICK`). Cada tipo de mensaje se maneja con un procedimiento distinto, siguiendo nuestra convención estándar de nombrar `doxxx` a todos los procedimientos invocados desde el ciclo principal, donde `xxx` es diferente para cada procedimiento. Como acotación, resulta desafortunado que muchos enlazadores truncan los nombres de procedimiento a siete u ocho caracteres, así que los nombres `do_set_time` y `do_setalarm` podrían causar conflictos. Por ello se cambió el nombre del segundo procedimiento a `do_setalarm`. Este problema ocurre en todo MINIX y, por lo regular, se resuelve abreviando uno de los nombres.

La tarea de alarma síncrona

Hay una segunda tarea que debemos estudiar en esta sección, la tarea de alarma síncrona. Una alarma síncrona es similar a una alarma normal, pero en vez de enviar una señal o invocar una función vigilante cuando expira el periodo de espera, la tarea de alarma síncrona envía un mensaje. Podría llegar una señal o podría invocarse una tarea vigilante sin importar qué parte de la tarea se está ejecutando, así que las alarmas de estos tipos son asíncronas. En contraste un mensaje sólo se recibe cuando el receptor ha ejecutado una llamada `receive`.

El mecanismo de alarma síncrona se agregó a MINIX a fin de apoyar el servidor de red que, al igual que el administrador de memoria y el servidor de archivos, se ejecuta como proceso individual. Es frecuente que surja la necesidad de poner un límite al tiempo que un proceso puede bloquearse mientras espera entradas. Por ejemplo, en una red, si no se recibe un acuse de recibo de un paquete de datos dentro de un periodo definido, es probable que haya habido una falla de transmisión. Un servidor de red puede establecer una alarma síncrona antes de que trate de recibir un mensaje y se bloquee. Puesto que la alarma síncrona se entrega como mensaje, desbloqueará el servidor tarde o temprano si éste no recibe un mensaje de la red. Al recibir cualquier mensaje, lo primero que el servidor debe hacer es restablecer la alarma. Luego, al examinar el tipo u origen del mensaje, podrá determinar si llegó un paquete o si fue desbloqueado porque se venció un tiempo de espera. Si sucedió lo segundo, el servidor puede intentar la recuperación, por lo regular volviendo a enviar el último paquete del cual no se usó recibo.

Una alarma síncrona es más rápida que una alarma enviada mediante una señal, lo cual requiere varios mensajes y una cantidad considerable de procesamiento. Una función vigilante es rápida, pero sólo puede usarse con tareas compiladas en el mismo espacio de direcciones que la tarea del reloj. Cuando un proceso está esperando un mensaje, una alarma síncrona es más apropiada y sencilla que las señales o las funciones de vigilancia, y se puede manejar fácilmente con poco procesamiento adicional.

El manejador de interrupciones de reloj

Como se explicó antes, cuando ocurre una interrupción de reloj, `realtime` no se actualiza de inmediato. La rutina de servicio de interrupciones mantiene el contador `pending_ticks` y realiza trabajos sencillos como cobrar el tic actual a un proceso y decrementar el temporizador de cuantos. Se envía un mensaje a la tarea del reloj sólo cuando es preciso realizar actividades más complicadas. Aunque lo ideal es que todas las tareas de MINIX se comuniquen mediante mensajes, esto es una concesión práctica ante la realidad de que dar servicio a los tics de reloj consume tiempo de CPU. En una máquina lenta se observó que al hacerse las cosas de esta manera se lograba un aumento del 15% en la rapidez del sistema relativa a una implementación que enviaba un mensaje a la tarea del reloj en cada interrupción del reloj.

Temporización de milisegundos

Como concesión adicional a la realidad, se incluyen unas cuantas rutinas en `clock.c` que ofrecen temporización con definición de milisegundos. Varios dispositivos de E/S requieren retrasos muy cortos (hasta 1 ms). No existe una forma práctica de implementar esto usando alarmas y la interfaz de transferencia de mensajes. Estas funciones están diseñadas para ser invocadas directamente por las tareas. La técnica empleada es la técnica de E/S más antigua y sencilla: la encuesta o interrogación. El contador que se emplea para generar las interrupciones de reloj se lee directamente, con la mayor rapidez posible, y la cuenta se convierte a milisegundos. El invocador hace esto repetidamente hasta que transcurre el tiempo deseado.

Resumen de los servicios de reloj

En la Fig. 3-26 se resumen los diversos servicios provistos por `clock.c`. Hay varias formas de acceder al reloj, y también de atender la petición. Algunos servicios están disponibles para cualquier proceso, y los resultados se devuelven en un mensaje.

El tiempo desde el arranque (`uptime`) se puede obtener mediante una llamada de función desde el kernel o una tarea, evitando el gasto extra de un mensaje. Una alarma puede ser solicitada por un proceso de usuario, en cuyo caso el resultado final es una señal, o por una tarea, causando la activación de una función vigilante. Ninguno de estos mecanismos puede ser utilizado por un proceso servidor, pero un servidor puede pedir una alarma síncrona. Una tarea o el kernel puede solicitar un retraso empleando la función `milli_delay`, o puede incorporar llamadas a `milli_elapsed` en una rutina de escrutinio, por ejemplo, mientras espera entradas de un puerto.

3.8.4 Implementación del controlador de reloj en MINIX

Cuando se inicia MINIX, se invocan todos los controladores. La mayor parte de ellos sólo intenta obtener un mensaje y se bloquea. El controlador de reloj, `clock_task` (línea 11098), hace lo mismo, pero primero invoca `init_clock` para inicializar la frecuencia del reloj programable en 60 Hz. Cada vez que se recibe un mensaje, el controlador suma `pending_ticks` a `realtime` y luego restablece

Parámetro	Disquete IBM de 360 KB	Disco duro WD de 540 MB
Número de cilindros	40	1048
Pistas por cilindro	2	4
Sectores por pista	9	252
Sectores por disco	720	1056384
Bytes por sector	512	512
Bytes por disco	368640	540868608
Tiempo de búsqueda (cilindros adyacentes)	6 ms	4 ms
Tiempo de búsqueda (caso medio)	77 ms	11 ms
Tiempo de rotación	200 ms	13 ms
Tiempo de arranque/paro del motor	250 ms	9 s
Tiempo para transferir un sector	22 ms	53 μ s

Figura 3-19. Parámetros de disco para el disquete de 360 KB de la IBM PC original y un disco duro Western Digital WD AC2540 de 540 MB.

pending_ticks antes de hacer cualquier otra cosa. Esta operación podría entrar en conflicto con una interrupción de reloj, así que se usan llamadas a lock y unlock para evitar una competencia (líneas 11115 a 11118). Por lo demás, el ciclo principal del controlador del reloj es esencialmente igual al de los demás controladores: se recibe un mensaje, se invoca una función que realice el trabajo necesario, y se envía un mensaje de respuesta.

Do_c (línea 11140) no se invoca en cada tic del reloj, así que su nombre no es Una descripción exacta de su función; se invoca cuando el manejador de interrupciones determina que podría haber algo importante que hacer. Primero se verifica si hubo una señal o terminó un temporizador vigilante. Si fue así, se inspeccionan todas las entradas de alarma en la tabla de procesos. Dado que los tics no se procesan individualmente, varias alarmas podrían “sonar” en una pasada por la tabla. También es posible que el proceso que iba a recibir la siguiente alarma ya haya terminado. Si se encuentra un proceso cuya alarma sea menor que el tiempo actual, pero no cero, se verifica la ranura del arreglo watch_dog (vigilante) que corresponde a ese proceso. En el lenguaje de programación C un valor numérico también tiene un valor lógico, de modo que la prueba de la línea 11161 devuelve TRUE si hay una dirección válida almacenada en la ranura de watch_dog, y la función correspondiente se invoca indirectamente en la línea 11163. Si se encuentra un apuntador nulo (representado en C con un valor de cero), el resultado de la prueba es FALSE y se invoca cause_sig para enviar una señal SIGALRM. La ranura de watch_dog también se usa cuando se necesita una alarma síncrona. En ese caso la dirección almacenada es la dirección de cause_a/am no la dirección de una función de vigilancia perteneciente a una tarea en particular. Para enviar una señal podríamos haber almacenado la dirección de cause_sig, pero entonces tendríamos que haber escrito cause_sig de forma diferente, sin esperar argumentos y obteniendo el número del proceso objetivo de una variable global. Como alternativa, podríamos haber hecho que todos los procesos vigilantes esperaran un argumento que no necesitan.

Analizaremos cause_sig cuando expliquemos la tarea del sistema en una sección subsecuente. Su trabajo consiste en enviar un mensaje al administrador de memoria. Esto requiere verificar si el administrador de memoria actualmente está esperando un mensaje. Si es así, cause_sig le envía un mensaje informándole de la alarma. Si el administrador de memoria está ocupado, cause_sig toma nota de que debe informarle de la alarma en la primera oportunidad.

Mientras se recorre la tabla de procesos inspeccionando el valor de `p_alarm` para cada proceso, se actualiza `next_alarm`. Antes de iniciar el ciclo, se asigna a `next_alarm` un número muy grande (línea 11151) y luego, para cada proceso cuyo valor de alarma sea distinto de cero después de enviarse las alarmas o señales, se comparan la alarma del proceso y `next_alarm`, y se asigna a éste el más pequeño de los dos valores (líneas 11171 y 11172).

Después de procesar las alarmas, `do_clocktick` determina si ya es momento de planificar otro proceso. El cuanto de ejecución se mantiene en la variable `PRIVATE sched_ticks`, que normal mente es decrementada por el manejador de interrupciones del reloj en cada tic del reloj. Sin embargo, en los tics en los que se activa `do_clocktick`, el manejador no decrementa `sched_ticks`, pues deja que `do_clocktick` mismo lo haga y pruebe si el resultado es cero en la línea 11178. `Sched_ticks` no se restablece cada vez que se planifica un proceso nuevo (porque se permite que el sistema de archivos y el administrador de memoria se ejecuten hasta terminar). En vez de ello, se restablece después de cada `SCHED_RATE` tics. La comparación de la línea 11179 se efectúa para asegurarse de que el proceso actual se ejecutó realmente durante, por lo menos, un tic completo del planificador antes de quitarle la CPU.

El siguiente procedimiento, `do_getuptime` (línea 11189), comprende sólo una línea; coloca el valor actual de realtime (el número de tics transcurridos desde el arranque) en el campo correcto del mensaje que se devolverá. Cualquier proceso puede averiguar el tiempo transcurrido de esta manera, pero el gasto extra del mensaje puede ser excesivo para las tareas, así que se proporciona una función relacionada, `get_uptime` (línea 11200) que puede ser invocada directamente por las tareas. Puesto que `get_uptime` no se invoca mediante un mensaje a la tarea de reloj, tiene que sumar ella misma los tics pendientes al valor actual de realtime. Se necesitan `lock` y `unlock` aquí para evitar que ocurra una interrupción de reloj mientras se está accediendo a `pending_ticks`.

Para obtener el tiempo real vigente, `do_get_time` (línea 11219) utiliza `realtime` y `boot_time` (el tiempo desde el arranque del sistema en segundos). `Do_set_time` (línea 11230) es su complemento; calcula un nuevo valor para `boot_time` con base en el tiempo real vigente dado y el número de tics transcurridos desde el arranque.

Los procedimientos `do_setalarm` (línea 11242) y `do_setsyn_alm` (línea 11269), para establecer una alarma normal y una alarma síncrona, respectivamente, son tan parecidos que los describiremos juntos. Ambos extraen del mensaje los parámetros que especifican el proceso al que se enviará una señal y el tiempo que debe esperarse. `Do_setalarm` también extrae la dirección de una función que se invocará (línea 11257), aunque unas cuantas líneas más adelante sustituye este valor con un apuntador nulo si el proceso objetivo es un proceso de usuario y no una tarea. Ya hemos visto cómo más adelante se prueba este apuntador en `do_clocktick` para determinar si el objetivo debe recibir una señal o una llamada a un vigilante. Ambas funciones calculan también el tiempo que falta para “sonar” la alarma (en segundos) y lo incluyen en el mensaje de retorno. Ambas invocan entonces `common_setalarm` para finalizar sus actividades. En el caso de la llamada a `do_setsyn_almz`, el parámetro de función que se pasa a `common_setalarm` siempre es `cause_alarm`,

`Common_setalarm` (línea 11291) termina el trabajo iniciado por cualquiera de las dos funciones que acabamos de describir, y luego almacena el tiempo de la alarma en la tabla de procesos y el apuntador al procedimiento vigilante (que también podría ser un apuntador a `cause_alarm` o un

apuntador nulo) en el arreglo `watch_dog`. A continuación, `common_setalarm` revisa toda la tabla de procesos para encontrar la siguiente alarma, tal como lo hace `do_clocktick`.

`Cause_alar,n` (línea 11318) es sencilla; asigna `TRUE` a una entrada del arreglo `syn_table` que corresponde al objetivo de la alarma síncrona. Si la tarea de alarma síncrona no está viva, se le envía un mensaje para que despierte.

Implementación de la tarea de alarma síncrona

La tarea de alarma síncrona, `syn_alarm_task` (línea 11333), sigue el modelo básico de todas las tareas: inicializa y luego ingresa en un ciclo sin fin en el que recibe y envía mensajes. La inicialización consiste en declarar que está viva asignando `TRUE` a la variable `syn_al_alive` y luego declarar que no tiene nada que hacer asignando `FALSE` a todas las entradas de `syn_table`. Esta tabla tiene una entrada para cada ranura de la tabla de procesos `Syn_alarin_task` inicia su ciclo exterior declarando que ha completado su trabajo y luego ingresa en un ciclo interior donde revisa todas las ranuras de `syn_table`. Si encuentra una entrada que indique que se espera una alarma síncrona, esta tarea restablece la entrada, envía un mensaje del tipo `CLOCK_INT` al proceso apropiado, y declara que no ha completado aún su trabajo. Al final de su ciclo exterior, la tarea no se detiene a esperar nuevos mensajes a menos que su bandera `work_done` este en 1 indicando que ya completo su trabajo. No se necesita un mensaje nuevo para indicarle a la tarea que hay más trabajo que efectuar, ya que `cause_alarm` escribe directamente en `syn_table`. Sólo se necesita un mensaje para despertar la tarea despues de que se ha quedado sin trabajo El efecto es que esta tarea repite su ciclo con gran rapidez en tanto hay alarmas que entregar.

De hecho, esta tarea no se utiliza en la versión de distribución de MINIX. Sin embargo, si recompilamos MINIX para agregar apoyo de trabajo con redes, el servidor de red la usará, pues necesita exactamente este tipo de mecanismo para obligar el vencimiento rápido de tiempos de espera silos paquetes no se reciben en el plazo esperado. Además de que se requiere velocidad, no es posible enviar señales a los servidores, ya que éstos deben ejecutarse indefinidamente, y la acción por omisión de la mayor parte de las señales es terminar el proceso objetivo.

Implementación del manejador de interrupciones de reloj

El diseño del manejador de interrupciones de reloj es un término medio entre hacer muy poco (a fin de minimizar el tiempo de procesamiento) y hacer lo suficiente para que no sea necesario activar con demasiada frecuencia la tarea del reloj, lo cual resulta costoso. El manejador modifica unas cuantas variables y prueba otras. Lo primero que hace `clock_handler` (línea 11374) es realizar algo de contabilidad del sistema. MINIX sigue la pista tanto al tiempo de usuario como al tiempo de sistema. El tiempo de usuario se carga a un proceso si se está ejecutando cuando ocurre un tic del reloj. El tiempo de sistema se carga si se está ejecutando el sistema de archivos o el administrador de memoria. La variable `bill_ptr` siempre apunta al último proceso de usuario planificado (los dos servidores no cuentan). La facturación se efectúa en las líneas 11447 y 11448. Una vez realizada la facturación, se incrementa `pending_ticks`, que es la variable más importante mantenida por `clock_handler` (línea 11450). Es preciso conocer el tiempo real para

probar si `clock_handler` debe despertar o no la terminal o enviar un mensaje a la tarea de reloj, pero la actualización en sí de realtime es costosa, porque esta operación debe efectuarse usando candados. A fin de evitar esto, el manejador calcula su propia versión del tiempo real en la variable local `now`. Existe la posibilidad de que el resultado sea incorrecto de vez en cuando, pero las consecuencias de tal error no serían graves.

El resto del trabajo del manejador depende de diversas pruebas. La terminal y la impresora deben despertarse cada cierto tiempo. `Tty_timeout` es una variable global, mantenida por la tarea de la terminal, que indica cuándo deberá despertarse la terminal la próxima vez. En el caso de la impresora es necesario verificar varias variables que son PRI VATE dentro del módulo de la impresora, y se prueban en la llamada a `pr_re_start`, que regresa rápidamente incluso si la impresora está paralizada, lo que sería el peor de los casos. En las líneas 11455 a 11458 se realiza una prueba que activa la tarea del reloj si se venció una alarma o si es hora de planificar otra tarea. Esta última prueba es compleja, un AND lógico de tres pruebas más sencillas. El código

```
interrupt(CLOCK);
```

de la línea 11459 hace que se envíe un mensaje `HARD_INT` a la tarea del reloj.

Al describir `do_clocktick` señalamos que decrementa `sched_ticks` y prueba si es cero para detectar la expiración del cuanto de ejecución. Probar si `sched_ticks` es igual a uno forma parte de la compleja prueba que mencionamos antes; aunque no se active la tarea del reloj, será necesario decrementar `sched_ticks` dentro del manejador de interrupciones y, si llega a cero, restablecer el cuanto. Si ocurre esto, también es momento de indicar que el proceso actual estaba activo al iniciarse el nuevo cuanto; esto se hace asignando el valor actual de `bill_ptr` a `prev_ptr` en la línea 11466.

Utilerías de tiempo

Por último, `clock.c` contiene algunas funciones que proporcionan diversos apoyos. Muchas de éstas son específicas para el hardware y tendrán que ser reemplazadas si MINIX se traslada a hardware que no sea Intel. Sólo describiremos lo que hacen estas funciones, sin entrar en detalles de su implementación.

`Init_clock` (línea 11474) es invocada por la tarea del temporizador cuando se ejecuta por primera vez. La función establece el modo y el retraso del chip temporizador de modo que produzca interrupciones de tic de reloj 60 veces cada segundo. A pesar del hecho de que la “velocidad de CPU” que se anuncia en la publicidad de las PC ha aumentado de 4.77 MHz para la IBM PC original a más de 200 MHz en los sistemas modernos, la constante `TIMER_COUNT`, empleada para inicializar el temporizador, es la misma en todos los modelos de PC en los que se ejecuta MINIX. Toda PC compatible con IBM, sea cual sea la velocidad de su procesador, suministra una señal de 14.3 MHz que es utilizada por diversos dispositivos que necesitan una referencia de tiempo. Las líneas de comunicación en serie y la pantalla de video también necesitan esta referencia de temporización.

El complemento de `init_clock` es `clock.stop` (línea 11489). Esta función no es realmente necesaria, pero es una concesión al hecho de que los usuarios de MINIX podrían querer iniciar

otro sistema operativo de vez en cuando. `Clock_stop` simplemente restablece los parámetros del chip temporizador al modo de operación predeterminado que MS-DOS y otros sistemas operativos podrían esperar del ROM BIOS en el momento de iniciarse.

Se proporciona `milli_delay` (línea 11502) para ser utilizada por cualquier tarea que necesite retrasos muy cortos. La función está escrita en C sin referencias a hardware específico, pero utiliza una técnica que sólo esperaríamos encontrar en una rutina de bajo nivel en lenguaje ensamblador. `MillDelay` inicializa un contador en cero y luego lo encuesta rápidamente hasta que alcanza un valor deseado. En el capítulo 2 dijimos que en general debe evitarse esta técnica de espera activa, pero las exigencias de la implementación pueden requerir excepciones a las reglas generales. La inicialización del contador corre por cuenta de la siguiente función, `milli_start` (línea 11516), que simplemente pone en cero dos variables. El escrutinio se efectúa invocando la última función, `milli_elapsed` (línea 11529), que accede al hardware del temporizador. El contador que se examina es el mismo que se utiliza para la cuenta regresiva de ticks del reloj, y por tanto puede sufrir un desbordamiento negativo y recuperar su valor máximo antes de que se complete el retraso deseado. `Milli` realiza la corrección necesaria en tal caso.

3.9 TERMINALES

Todas las computadoras de propósito general tienen una o más terminales que sirven para comunicarse con ellas. Hay un número muy grande de tipos de terminales distintos, y toca al controlador de la terminal ocultar todas estas diferencias de modo que la parte del sistema operativo independiente del dispositivo y los programas de usuario no tengan que reescribirse para cada tipo de terminal. En las siguientes secciones seguiremos nuestro enfoque estándar de examinar primero el hardware de las terminales en general y luego estudiar el software de MINIX.

3.9.1 Hardware de terminales

Desde el punto de vista del sistema operativo, las terminales pueden dividirse en tres categorías amplias con base en la forma en que el sistema operativo se comunica con ellas. La primera categoría consiste en terminales con mapa en la memoria, que consisten en un teclado y una pantalla, ambas conectadas directamente a la computadora. La segunda categoría consiste en terminales que se conectan a través de una línea de comunicación en serie empleando el estándar RS-232, casi siempre usando un módem. La tercera categoría consiste en terminales que se conectan a la computadora a través de una red. Esta taxonomía se muestra en la Fig. 3-27.

Terminales con mapa en la memoria

La primera categoría amplia de terminales indicada en la Fig. 3-27 consiste en dispositivos con mapa en la memoria. Estas terminales son parte integral de las computadoras mismas. La interfaz con las terminales con mapa en la memoria se establece a través de una memoria especial llamada RAM de video que forma parte del espacio de direcciones de la computadora y es direccionada por la CPU de la misma forma que el resto de la memoria (Fig. 3-28).

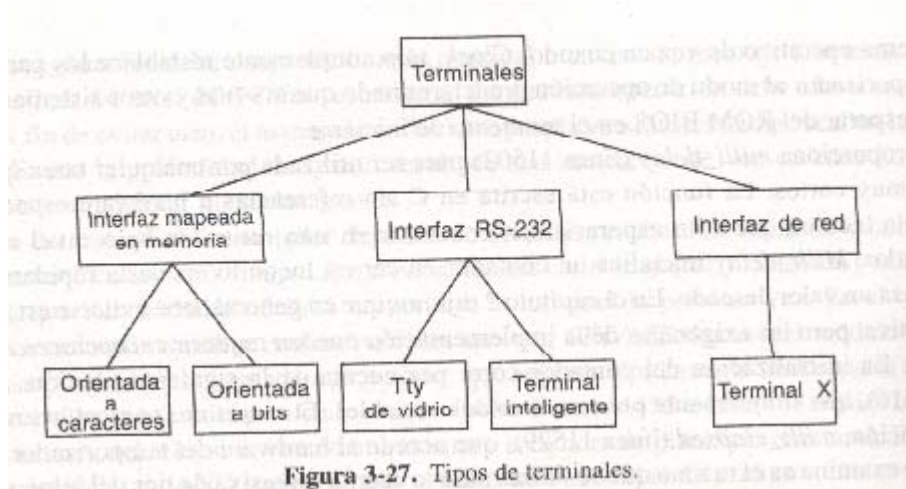


Figura 3-27. Tipos de terminales.

La tarjeta de RAM de video también contiene un chip llamado **controlador de video**. Este chip extrae códigos de caracteres de la RAM de video y genera la señal de video que maneja la pantalla (monitor). El monitor genera un haz de electrones que barre la pantalla horizontalmente, dibujando líneas sobre ella. Por lo regular, la pantalla contiene entre 480 y 1024 líneas horizontales, cada una de las cuales tiene entre 640 y 1200 puntos. Estos puntos se denominan **pixeles**. La señal del controlador de video modula el haz de electrones, determinando si un pixel dado estará iluminado u oscuro. Los monitores a color tienen tres haces, para rojo, verde y azul, que se modulan de forma independiente.

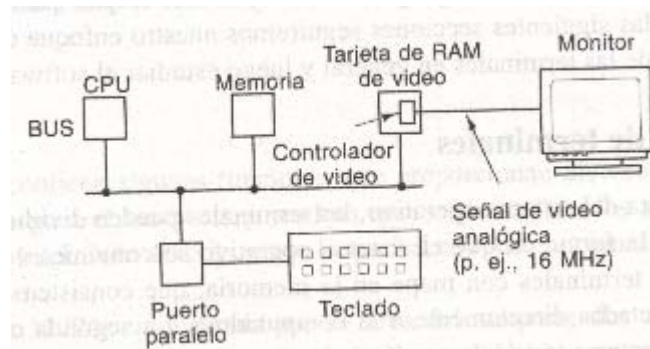


Figura 3-28. Las terminales mapeadas en memoria escriben directamente en la RAM de video.

Una pantalla monocromática sencilla podría exhibir un carácter en un cuadro de 9 pixeles de anchura y 14 pixeles de altura (incluido el espacio entre caracteres), contando así con 25 líneas de 80 caracteres cada una. En tal caso, la pantalla tendría 350 líneas de barrido con 720 pixeles cada una. Cada una de estas tramas se redibuja de 45 a 70 veces por segundo. El controlador de video podría diseñarse de modo que obtenga los primeros 80 caracteres de la RAM de video, genere 14 líneas de barrido, obtenga los siguientes 80 caracteres de la RAM de video, genere las siguientes 14 líneas de barrido, etc. De hecho, la mayor parte de los controladores obtienen los caracteres 14 veces, una por cada línea de barrido, para no tener que guardarlos en un buffer interno.

Los patrones de 9 x 14 bits para los caracteres se guardan en una ROM empleada por el controlador de video (también puede usarse RAM para manejar tipos de letra personalizados). Las direcciones de esta ROM tienen 12 bits, 8 bits para el código del carácter y 4 bits para especificar la línea de barrido. Los 8 bits de cada byte de la ROM controlan 8 pxeles; el noveno pixel entre caracteres siempre está en blanco. Por tanto, se necesitan $14 \times 80 = 1120$ referencias a la RAM de video para exhibir una línea de texto en la pantalla. Se hace un número igual de referencias a la ROM del generador de caracteres.

La IBM PC tiene varios modos para la pantalla. En el más sencillo, se usa una pantalla con mapa de caracteres para la consola. En la Fig. 3-29(a) vemos una porción de la RAM de video. Cada carácter de la pantalla de la Fig. 3-29(b) ocupa dos caracteres de la RAM. El carácter de orden bajo es el código ASCII para el carácter que se exhibe. El carácter de orden alto es el byte de atributo, que sirve para especificar el color, video inverso, parpadeo, etc. La pantalla completa de 25 por 80 caracteres requiere 4000 bytes de RAM de video en este modo.

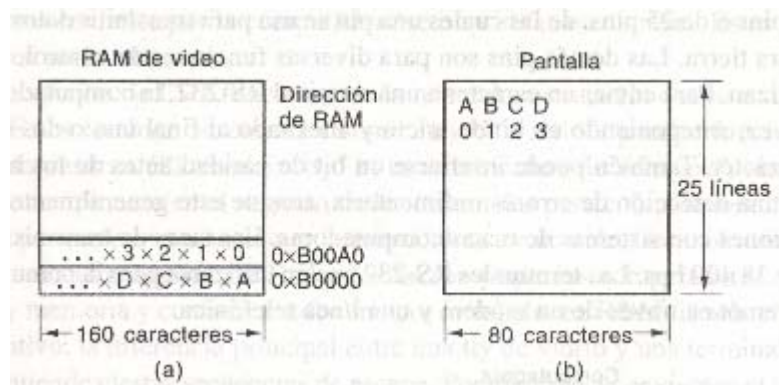


Figura 3-29. (a) Imagen de RAM de video para la pantalla monocromática IBM. (b) La pantalla correspondiente. Las x son bytes de atributos.

Las terminales de mapa de bits usan el mismo principio, excepto que se controla individualmente cada pixel de la pantalla. En la configuración más simple, para una pantalla monocromática, cada pixel tiene un bit correspondiente en la RAM de video. En el otro extremo, cada pixel se representa con un número de 24 bits, dedicando 8 bits para cada color básico: rojo, verde y azul. Una pantalla a color de 768 x 1024 con 24 bits por pixel requiere 2 MB de RAM sólo para contener la imagen.

Cuando se usa una pantalla con mapa en memoria, el teclado está totalmente desacoplado de la pantalla. La interfaz con el teclado puede ser a través de un puerto en serie o en paralelo. En cada acción de tecla, se interrumpe la CPU, y el controlador del teclado extrae el carácter digitado leyendo un puerto de E/S.

En la IBM PC, el teclado contiene un microprocesador incorporado que se comunica a través de un puerto en serie especializado con un chip controlador en la tarjeta matriz. Se genera una interrupción cada vez que se pulsa una tecla y también cuando se suelta. Además, todo lo que el hardware del teclado proporciona es el número de tecla, no el código ASCII. Cuando se pulsa la

tecla A, se coloca el código de tecla (30) en un registro de E/S. Toca al controlador en software determinar si se trata de mayúscula, minúscula, CTRL-A, ALT-A, CTRL-ALT-A o alguna otra combinación. Puesto que el controlador sabe cuáles teclas se han pulsado pero todavía no se han soltado (p. ej., la tecla de mayúsculas), cuenta con suficiente información para realizar el trabajo. Aunque la interfaz del teclado deja toda la carga al software, es extremadamente flexible. Por ejemplo, los programas de usuario pueden estar interesados en si un dígito que se acaba de pulsar provino de la fila superior del teclado principal o del subteclado numérico que está a la derecha. En principio, el controlador puede proporcionar esta información.

Terminales RS-232

Las terminales RS-232 son dispositivos provistos de un teclado y una pantalla que se comunican por medio de una interfaz en serie, bit por bit (véase la Fig. 3-30). Estas terminales usan un conector de 9 pins o de 25 pins, de las cuales una pin se usa para transmitir datos, una para recibir datos y una para tierra. Las demás pins son para diversas funciones de control, que en su mayor parte no se utilizan. Para enviar un carácter a una terminal RS-232, la computadora debe transmitir un bit a la vez, anteponiendo un bit de inicio y anexando al final uno o dos bits de paro para delimitar el carácter. También puede insertarse un bit de paridad antes de los bits de paro, si se desea realizar una detección de errores rudimentaria, aunque esto generalmente sólo se exige en las comunicaciones con sistemas de macrocomputadoras. Las tasas de transmisión comunes son 9600, 19 200 y 38 400 bps. Las terminales RS-232 suelen utilizarse para la comunicación con una computadora remota a través de un módem y una línea telefónica.

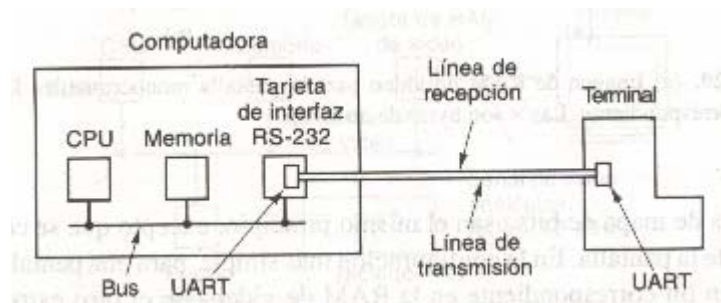


Figura 3-30. Una terminal RS-232 se comunica con una computadora a través de una línea de comunicación, bit por bit. La computadora y la terminal son totalmente independientes.

Dado que tanto las computadoras como las terminales trabajan internamente con caracteres completos pero deben comunicarse por una línea serial con un bit a la vez, se han creado chips que realizan las conversiones de carácter a serie y de serie a carácter. Estos chips se denominan UART (receptor transmisor universal asincrono) y se conectan a la computadora insertando tarjetas de interfaz RS-232 en el bus como se ilustra en la Fig. 3-30. Las terminales RS-232 se usan cada vez menos, pues están siendo sustituidas por PC y terminales X, pero todavía se encuentran en los sistemas de macrocomputadoras más antiguos, sobre todo en aplicaciones bancarias, de reservaciones de líneas aéreas y similares.

Para exhibir un carácter, el controlador en software de la terminal escribe el carácter en la tarjeta de interfaz, donde se coloca en un buffer. De ahí, el UART lo desplaza hacia la línea serial bit por bit. Incluso a 38 400 bps, toma más de 250 microsegundos enviar un carácter. A causa de esta tasa de transmisión tan lenta, el controlador generalmente envía un carácter a la tarjeta RS-232 y se bloquea, esperando la interrupción que la interfaz genera cuando termina de transmitir el carácter y el UART está en condiciones de aceptar otro. El UART puede enviar y recibir caracteres simultáneamente, como indica su nombre. También se genera una interrupción cuando se recibe un carácter, y por lo regular es posible colocar en buffers un número pequeño de caracteres de entrada. El controlador de la terminal debe examinar un registro cuando se recibe una interrupción para determinar la causa de la interrupción. Algunas tarjetas de interfaz tienen una CPU y memoria capaces de manejar múltiples líneas, y asumen una buena parte de la carga de E/S de la CPU principal.

Las terminales RS-232 se pueden subdividir en categorías, como ya se mencionó. Las más sencillas eran las terminales de impresión. Los caracteres teclados se transmitían a la computadora. Los caracteres enviados por la computadora se imprimían en el papel. Estas terminales son obsoletas y ya casi no se observan.

Las terminales tontas de CRT funcionan de la misma manera, sólo que usan una pantalla en lugar de papel. Éstas también se conocen como "tty de vidrio", porque funcionalmente son iguales a las tty impresoras. (El término "tty" es una abreviatura de Teletype®, una compañía, ya desaparecida, que fue pionera en el negocio de las terminales para computadora; "tty", o "teletipo", suele usarse para referirse a cualquier terminal.) Las tty de vidrio también son obsoletas ya.

Las terminales inteligentes de CRT en realidad son pequeñas computadoras especializadas; tienen una CPU y memoria y contienen software, por lo regular en ROM. Desde el punto de vista del sistema operativo, la diferencia principal entre una tty de vidrio y una terminal inteligente es que la segunda entiende ciertas secuencias de escape. Por ejemplo, si enviamos el carácter ASCII ESC (033) seguido por varios otros caracteres, podemos mover el cursor a cualquier posición de la pantalla, insertar texto en medio de la pantalla, y otras cosas más.

Terminales X

La última palabra en terminales inteligentes es una terminal que contiene una CPU tan potente como la de la computadora principal, junto con megabytes de memoria, un teclado y un ratón. Una terminal común de este tipo es la terminal X, que ejecuta el sistema Ventana X del M.I.T. Por lo regular, las terminales X hablan con la computadora principal a través de una Ethernet.

Una terminal X es una computadora que ejecuta el software X. Algunos productos están dedicados a ejecutar sólo X; otros son computadoras de propósito general que simplemente ejecutan X como un programa entre muchos más. En cualquier caso, una terminal X tiene una pantalla grande de mapa de bits, por lo regular con definición de 960 o 1200 o más fina aún, en blanco y negro, escala de grises o color, un teclado completo y un ratón, normalmente con tres botones.

El programa dentro de la terminal X que obtiene entradas del teclado o del ratón y acepta comandos de una computadora remota se denomina servidor X. Este servidor se comunica a

través de la red con **clientes X** que se ejecutan en algún anfitrión remoto. Puede parecer extraño tener el servidor X dentro de la terminal y los clientes en el anfitrión remoto, pero el trabajo de servidor X consiste en exhibir bits, así que resulta lógico que esté cerca del usuario. La disposición de cliente y servidor se muestra en la Fig. 3-31.

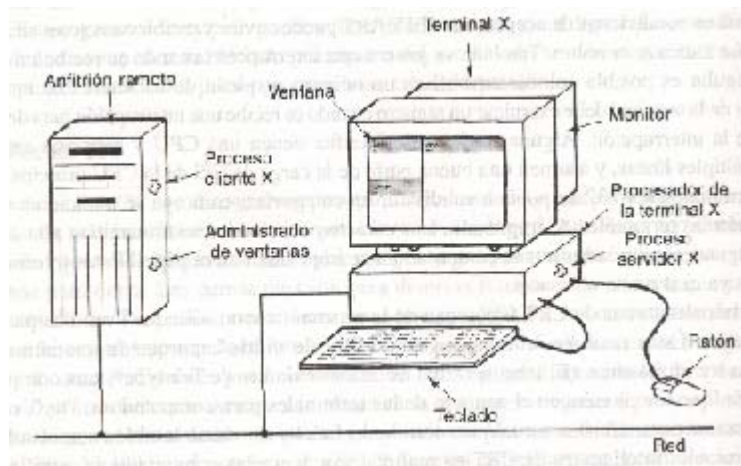


Figura 3-31. Clientes y servidores en el sistema X Ventana del M.I.T.

La pantalla de la terminal X contiene ventanas, cada una en forma de retícula rectangular de píxeles. Cada ventana por lo regular tiene una barra de título arriba, una barra de desplazamiento a la izquierda y un cuadro para redimensionar en la esquina superior derecha. Uno de los clientes X es un programa llamado **administrador de ventanas**, cuyo trabajo es controlar la creación, eliminación y desplazamiento de las ventanas en la pantalla. Para administrar las ventanas, este cliente envía comandos al servidor X indicándole qué debe hacer. Estos comandos incluyen dibujar punto, dibujar línea, dibujar rectángulo, dibujar polígono, llenar rectángulo, llenar polígono, etcétera.

El trabajo del servidor X consiste en coordinar las entradas del ratón, del teclado y de los clientes X y actualizar la pantalla de manera acorde. El servidor debe saber cuál ventana está seleccionada actualmente (donde está el puntero del ratón) para determinar a cuál cliente debí enviar las nuevas entradas del teclado.

3.9.2 Software de terminales

El teclado y la pantalla son dispositivos casi independientes, así que aquí los estudiaremos por separado. (La independencia no es total, ya que los caracteres tecleados deben exhibirse en la pantalla.) En MINIX los controladores del teclado y de la pantalla forman parte de la misma tarea; en otros sistemas pueden estar divididos en controladores distintos.

Software de entrada

El trabajo básico del controlador de teclado consiste en obtener entradas del teclado y pasarlas a los programas de usuario cuando éstos lean de la terminal. Se pueden adoptar dos posibles filosofías para el controlador. En la primera, el controlador se limita a aceptar entradas y pasarlas más arriba sin modificarlas. Un programa que lee de la terminal obtiene una secuencia en bruto de códigos ASCII. (Proporcionar a los programas de usuario los números de tecla sería demasiado primitivo, además de ser muy dependiente de la máquina.)

Esta filosofía es idónea para las necesidades de los editores de pantalla avanzados como emacs, que permiten al usuario ligar una acción arbitraria a cualquier carácter o secuencia de caracteres. Por otro lado, esta filosofía implica que si el usuario teclea `date` en lugar de `date` y luego corrige el error pulsando tres veces la tecla de retroceso y escribiendo `ate`, seguido de un retomo de carro, el programa de usuario recibirá los 11 códigos ASCII tecleados.

La mayor parte de los programas no desean tanto detalle; simplemente quieren la entrada corregida, no la secuencia exacta que la produjo. Esta observación nos lleva a la segunda filosofía: el controlador se encarga de toda la edición dentro de una línea, y entrega líneas corregidas a los programas de usuario. La primera filosofía está orientada a caracteres; la segunda está orientada a líneas. Originalmente, se llamaba a estas filosofías modo crudo y modo cocido, respectivamente. El estándar POSIX emplea el término menos pintoresco de modo canónico para describir el modo orientado a líneas. En la mayor parte de los sistemas el modo canónico se refiere a una configuración bien definida. El modo no canónico equivale al modo crudo, aunque es posible alterar muchos detalles del comportamiento de la terminal. Los sistemas compatibles con POSIX ofrecen varias funciones de biblioteca que permiten seleccionar cualquiera de los dos modos y modificar muchos aspectos de la configuración de la terminal. En MINIX, la llamada al sistema `IOCTL` apoya estas funciones.

La primera tarea del controlador del teclado consiste en obtener caracteres. Si cada digitación causa una interrupción, el controlador puede adquirir el carácter durante la interrupción. Si el software de bajo nivel convierte las interrupciones en mensajes, es posible colocar el carácter recién adquirido en el mensaje. Como alternativa, el carácter puede colocarse en un buffer pequeño en la memoria y usarse el mensaje para indicarle al controlador que llegó algo. La segunda estrategia es más segura si sólo es posible enviar mensajes a procesos que están esperando y existe la posibilidad de que el controlador del teclado todavía esté ocupado con el carácter anterior.

Una vez que el controlador ha recibido el carácter, debe comenzar a procesarlo. Si el teclado entrega números de tecla en lugar de los códigos de caracteres empleados por el software de aplicación, el controlador deberá realizar la conversión entre los códigos empleando una tabla. No todas las "compatibles con IBM" usan la numeración de teclas estándar, así que si el controlador desea apoyar tales máquinas deberá establecer una correspondencia entre los diferentes teclados y diferentes tablas. Una estrategia sencilla consiste en compilar una tabla que establece una correspondencia entre los códigos proporcionados por el teclado y los códigos ASCII (American Standard Code for Information Interchange) en el controlador del teclado, pero esto no resulta satisfactorio para usuarios de lenguajes distintos del inglés. Los teclados tienen diferente organización en los distintos países, y el conjunto de caracteres ASCII no resulta adecuado ni siquiera

para la mayoría de los usuarios del Hemisferio Occidental, donde los hablantes de español, portugués y francés requieren caracteres acentuados y signos de puntuación que no se utilizan en el inglés. A fin de responder a la necesidad de una organización de teclado flexible adaptable a diferentes idiomas, muchos sistemas operativos ofrecen mapas de teclas o páginas de código cargables, que permiten escoger la correspondencia entre los códigos del teclado y los códigos proporcionados a la aplicación, ya sea cuando se arranca el sistema o posteriormente.

Si la terminal está en modo canónico (cocido), los caracteres deben almacenarse hasta que se acumule una línea completa, porque el usuario podría decidir subsecuentemente borrar una parte. Incluso si la terminal está en modo crudo, es posible que el programa todavía no haya solicitado entradas, por lo que los caracteres deben colocarse en un buffer para permitir el tecleo adelantado. (A los diseñadores de sistemas que no permiten a los usuarios teclear por adelantado muchos caracteres se les debería untar de brea y cubrir de plumas o, peor aún, se les debería obligar a usar su propio sistema.)

Hay dos enfoques comunes para el almacenamiento en buffer de los caracteres. En el primero, el controlador cuenta con una reserva central de buffers, cada uno de los cuales puede contener tal vez 10 caracteres. Cada terminal tiene asociada una estructura de datos que contiene, entre otras cosas, un apuntador a la cadena de buffers para las entradas obtenidas de esa terminal. Al teclearse más caracteres, se adquieren más buffers y se anexan a la cadena. Cuando los caracteres se pasan a un programa de usuario, los buffers se retiran y se devuelven a la reserva central.

El otro enfoque consiste en realizar el almacenamiento intermedio en la estructura de datos de la terminal misma, sin una reserva central de buffers. Puesto que es común que los usuarios tecleen un comando que tardará cierto tiempo (digamos, una compilación) y luego tecleen unas cuantas líneas por anticipado, el controlador deberá, por seguridad, asignar algo así como 200 caracteres por terminal. En un sistema de tiempo compartido a gran escala con 100 terminales, la asignación de 20K permanentemente para el tecleo adelantado es obviamente una exageración, y en este caso seguramente bastará con una reserva central de buffers con espacio para unos 5K. Por otro lado, tener un buffer dedicado por terminal hace que el controlador sea más sencillo (pues no tendrá que manejar una lista enlazada) y es preferible en las computadoras personales con sólo una o dos terminales. En la Fig. 3-32 se muestra la diferencia entre estos dos métodos.

Si bien el teclado y la pantalla son dispositivos lógicamente independientes, muchos usuarios están acostumbrados a ver en la pantalla los caracteres que acaban de teclear. Algunas terminales (antiguas) complacen al usuario en este sentido exhibiendo automáticamente (en hardware) todo lo que se teclea. Esto es un problema no sólo cuando se están introduciendo contraseñas, sino porque limita considerablemente la flexibilidad de los editores avanzados y de otros programas. Por fortuna, la mayor parte de las terminales modernas no exhiben nada cuando se está tecleando; es obligación del software exhibir las entradas. Este proceso se denomina eco.

El eco se complica por el hecho de que un programa podría estar escribiendo en la pantalla mientras el usuario está tecleando. Como mínimo, el controlador del teclado tendrá que buscar dónde poner las entradas nuevas para que no sean sobrescritas por las salidas del programa.

El eco también se complica cuando se teclean más de 80 caracteres en una terminal que tiene líneas de 80 caracteres. Dependiendo de la aplicación, puede ser apropiado o no continuar en la

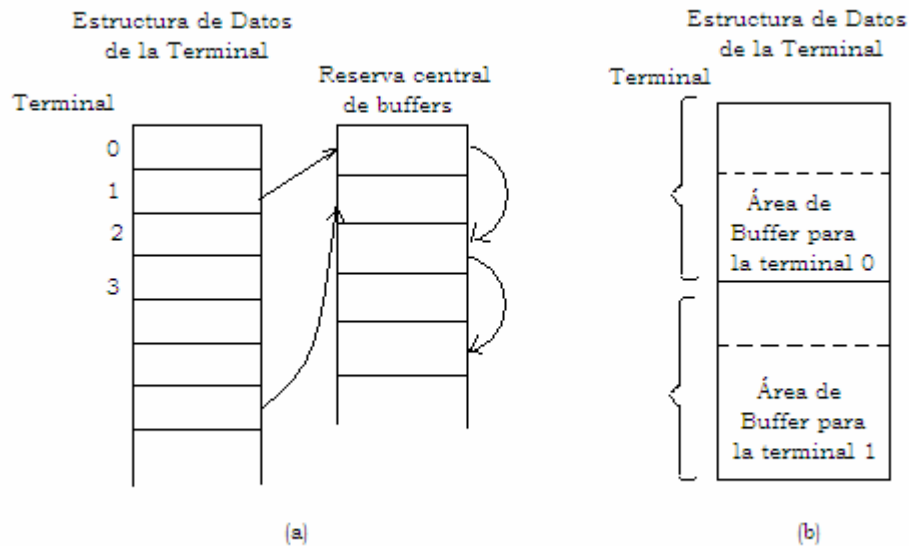


Figura 3-32. (a) Reserva central de buffers. (b) Buffer dedicado para cada terminal.

siguiente línea. Algunos controladores simplemente truncan las líneas a 80 caracteres desechando todos los caracteres más allá de la columna 80.

Otro problema es el manejo de las tabulaciones. La mayor parte de las terminales tienen una tecla de tabulación, pero pocas pueden manejar las tabulaciones en las salidas. Corresponde al controlador calcular la posición actual del cursor, teniendo en cuenta tanto las salidas de los programas como la salida del eco, y calcular el número correcto de espacios que deben exhibirse.

Ahora llegamos al problema de la equivalencia de dispositivos. Lógicamente, al final de una línea de texto habría que tener un retomo de carro, para regresar el cursor a la columna 1, y un salto de línea, para avanzar a la siguiente línea. A los usuarios seguramente no les gustaría tener que teclear ambas cosas al final de cada línea (aunque algunas terminales tienen una tecla que genera estos dos caracteres, con una probabilidad del 50% de hacerlo en el orden en que el software lo desea). Toca al controlador convertir todo lo que llega al formato interno estándar empleado por el sistema operativo.

Si la forma estándar consiste en almacenar simplemente un salto de línea (la convención en MINIX), los retomos de carro deberán convertirse en saltos de línea. Si el formato interno consiste en almacenar las dos cosas, el controlador deberá generar un salto de línea cuando reciba un retomo de carro y un retomo de carro cuando reciba un salto de línea. Sea cual sea la convención interna, es posible que la terminal requiera que se haga eco tanto de un retomo de carro como de un salto de línea para que la pantalla se actualice correctamente. Dada la posibilidad de que una computadora grande tenga una amplia variedad de terminales diferentes conectadas a ella, es obligación del controlador del teclado hacer que todas las combinaciones de retomo de carro y salto de línea se conviertan al estándar interno del sistema y que se haga eco de todo lo necesario.

Un problema relacionado es el de los tiempos de los retomos de carro y saltos de línea. En algunas terminales puede requerirse más tiempo para exhibir un retomo de carro o un salto de

línea que una letra o un número. Si el microprocesador dentro de la terminal tiene que copiar un bloque grande de texto para efectuar el desplazamiento de la pantalla, es posible que los saltos de línea sean lentos. Si es necesario regresar una cabeza de impresión mecánica al margen izquierdo del papel, los retomos de carro pueden ser lentos. En ambos casos es responsabilidad del controlador insertar caracteres de llenado (caracteres nulos ficticios) en el flujo de salida o simplemente dejar de producir salidas durante el tiempo suficiente para que la terminal se ponga al día. La duración del retraso a menudo está relacionada con la rapidez de la terminal; por ejemplo, a 4800 bps o menos tal vez no se requieran retrasos, pero a 9600 bps o más podría requerirse un carácter de relleno. Las terminales con tabulaciones en hardware, sobre todo las de copia impresa, podrían requerir también un retraso después de una tabulación.

Cuando se opera en modo canónico, varios caracteres de entrada tienen un significado especial. En la Fig. 3-33 se muestran todos los caracteres especiales requeridos por POSIX y los adicionales que MINIX reconoce. Los caracteres predeterminados son caracteres de control que no deberán causar conflictos con las entradas de texto ni los códigos utilizados por los programas, pero todos con excepción de los dos últimos pueden modificarse con el comando `stty`, si se desea. Las versiones antiguas de UNIX empleaban diferentes caracteres por omisión para muchos de éstos.

Carácter	Nombre POSIX	Comentario
CTRL-D	EOF	Fin de archivo
	EOL	Fin de línea (no definido)
CTRL-H	ERASE	Retroceder un carácter
DEL	INTR	Interrumpir proceso (SIGINT)
CTRL-U	KILL	Borrar toda la línea tecleada
CTRL-\	QUIT	Forzar vaciado de núcleo (SIQQUIT)
CTRL-Z	SUSP	Suspender (MINIX lo ignora)
CTRL-Q	START	Iniciar salidas
CTRL-S	STOP	Detener salidas
CTRL-R	REPRINT	Reexhibir entradas (extensión de MINIX)
CTRL-V	LNEXT	Sigue literal (extensión de MINIX)
CTRL-O	DISCARD	Desechar salidas (extensión de MINIX)
CTRL-M	CR	Retorno de carro (no modificable)
CTRL-J	NL	Salto de línea (no modificable)

Figura 3-33. Caracteres que se manejan de forma especial en modo canónico.

El carácter ERASE permite al usuario borrar el carácter que acaba de teclearse. En MINIX se usa para ello el retroceso (CTRL-H). Este carácter no se agrega a la cola de caracteres; más bien, quita de la cola el carácter anterior. El eco de este carácter debe consistir en una secuencia de tres caracteres, retroceso, espacio y retroceso, a fin de borrar el carácter anterior de la pantalla. Si el

carácter anterior era una tabulación, para borrarlo será necesario saber dónde estaba el cursor antes de la tabulación. En la mayor parte de los sistemas, el retroceso sólo borra caracteres de la línea actual; no borra el retomo de carro ni continúa con la línea anterior.

Cuando el usuario observa un error al principio de la línea que está tecleando, muchas veces es más cómodo borrar toda la línea y comenzar otra vez. El carácter KILL (CTRL-U en MINIX) borra toda la línea. MINIX hace que la línea borrada desaparezca de la pantalla, pero algunos sistemas hacen eco de ella junto con un retomo de carro y un salto de línea porque a algunos usuarios les gusta ver la línea vieja. Por tanto, la forma como se hace eco de KILL es cuestión de gustos. Al igual que con ERASE, normalmente no es posible regresar más allá de la línea en curso. Cuando se elimina un bloque de caracteres, podría o no valer la pena que el controlador devuelva buffers a la reserva, si se usa.

A veces los caracteres ERASE o KILL deben introducirse como datos ordinarios. El carácter LNEXT sirve como carácter de escape. En MINIX CTRL-V es lo predeterminado. Por ejemplo, los primeros sistemas UNIX a menudo usaban el signo @ para KILL, pero el sistema de correo de Internet utiliza direcciones de la forma linda@cs.washington.edu. Alguien que se sienta más cómodo con las convenciones antiguas podría redefinir KILL como @, pero después podría necesitar introducir un signo @ literalmente para especificar una dirección de correo electrónico. Esto puede hacerse tecleando CTRL-V @. El CTRL-V mismo se puede introducir literalmente tecleando CTRL-V CTRL-V. Después de ver el CTRL-V, el controlador enciende una bandera que indica que el siguiente carácter está exento de procesamiento especial. El carácter LNEXT mismo no se coloca en la cola de caracteres.

En caso de que el usuario desee detener una imagen en la pantalla para que no desaparezca por el desplazamiento, se proporcionan códigos de control para congelar la pantalla y reactivarla después. En MINIX estos códigos son STOP (CTRL-S) y START (CTRL-Q), respectivamente. Éstos no se almacenan, sino que se usan para encender y apagar (izar y bajar) una bandera en la estructura de datos de la terminal. Cada vez que se intenta producir salidas, se examina la bandera. Si la bandera está encendida, no habrá salida. Casi siempre el eco también se suprime junto con las salidas de los programas.

En muchos casos es necesario terminar un programa fuera de control que se está depuando. Los caracteres INTR (DEL) y QUIT (CTRL-\) pueden servir para este fin. En MINIX, DEL (SUPR) envía la señal SIGINT a todos los procesos iniciados en la terminal. La implementación de DEL puede ser complicada. La parte difícil es hacer que la información del controlador llegue a la parte del sistema que maneja las señales, la cual, después de todo, no ha solicitado esta información. CTRL-\ es similar a DEL, excepto que envía la señal SIGQUIT, que obliga a un vaciado de núcleo si no es atrapada o ignorada. Cuando se pulsa cualquiera de estas teclas, el controlador deberá hacer eco de un retomo de carro y un salto de línea y desechar todas las entradas acumuladas a fin de iniciar desde cero. El valor por omisión de INTR muchas veces es CTRL-C en lugar de DEL, ya que muchos programas usan DEL para edición, además de la tecla de retroceso.

Otro carácter especial es EOF (CTRL-D), que en MINIX hace que cualquier petición de lectura de la terminal pendiente se satisfaga con lo que esté disponible en el buffer, incluso si éste está vacío. Si se teclaea CTRL-D al principio de una línea, el programa obtiene una lectura de 0

bytes, lo que convencionalmente se interpreta como un fin de archivo y hace que la mayor parte de los programas actúen tal como lo harían al detectar un fin de archivo en un archivo de entrada.

Algunos controladores de terminal permiten realizar actividades de edición dentro de la línea mucho más avanzadas que las que hemos bosquejado aquí. Esos controladores cuentan con caracteres de control especiales para borrar una palabra, saltar caracteres o palabras hacia atrás o hacia adelante, ir al principio o al final de la línea que se está tecleando, etc. La adición de todas estas funciones al controlador de la terminal lo hace mucho más grande y, además, es un desperdicio cuando se usan editores de pantalla avanzados que de todas maneras trabajan en modo crudo.

Con objeto de que los programas puedan controlar los parámetros de las terminales, POSIX exige que estén disponibles varias funciones en la biblioteca estándar, siendo las más importantes `tcgetattr` y `tcssetattr`. `Tcgetattr` obtiene una copia de la estructura que se muestra en la Fig. 3-34, llamada `termios`, que contiene toda la información necesaria para cambiar caracteres especiales, establecer modos y modificar otras características de una terminal. Un programa puede examinar los valores actuales y modificarlos si lo desea. Luego, `tcssetattr` escribe la estructura otra vez en la tarea de la terminal.

```
struct termios {
    tcflag_t cIflag;           /* modos de entrada */
    tcflag_t cOflag;         /* modos de salida */
    tcflag_t cCflag;         /* modos de control */
    tcflag_t cLflag;         /* modos locales */
    speed_t cSspeed;         /* velocidad de entrada */
    speed_t cOSpeed;         /* velocidad de salida */
    cc_t c_cc[NCCS];         /* caracteres de control */
};
```

Figura 3-34. La estructura `termios`. En MINIX, `tc_flag_t` es un `short`, `speed_t` es un `int` y `cc_t` es un `char`.

POSIX no especifica si sus requisitos deben implementarse mediante funciones de biblioteca o llamadas al sistema. MINIX ofrece una llamada al sistema, `IOCTL`, invocada por

```
ioctl(descriptor_archivo, solicitud, argp);
```

que sirve para examinar y modificar las configuraciones de muchos dispositivos de E/S. Esta llamada se usa para implementar las funciones `tcgetattr` y `tcssetattr`. La variable `solicitud` especifica si se debe leer o escribir la estructura `termios` y, en caso de que se vaya a escribir, si la petición debe atenderse de inmediato o diferirse hasta que se hayan completado todas las salidas que actualmente están en cola. La variable `argp` es un apuntador a una estructura `termios` en el programa invocador. Esta forma específica de comunicación entre los programas y el controlador se escogió por su compatibilidad con UNIX, más que por su belleza inherente.

Es apropiado incluir unas cuantas notas acerca de la estructura `termios`. Las cuatro palabras de banderas ofrecen mucha flexibilidad. Los bits individuales de `c_iflag` controlan diversas formas de manejar las entradas. Por ejemplo, el bit `ICRNL` hace que los caracteres `CR` de las entradas se conviertan en `NL`. Esta bandera se enciende por omisión en MINIX. La palabra `c_oflag`

contiene bits que afectan el procesamiento de las salidas. Por ejemplo, el bit OPOST habilita el procesamiento de las salidas. Ese bit y el bit ONLCR, que hace que los caracteres NL de la salida se conviertan en una secuencia CR NL, también están encendidos por omisión en MINIX. C_cflag es la palabra de banderas de control. Los valores por omisión para MINIX permiten que una línea reciba caracteres de 8 bits y hacen que un módem cuelgue si un usuario termina una sesión en la línea. C_lflag es el campo de banderas de modo local. Un bit, ECHO, habilita el eco (y puede apagarse durante un inicio de sesión para que la introducción de la contraseña sea más segura). El bit más importante aquí es ICANON, que habilita el modo canónico. Con ICANON apagado, hay varias posibilidades. Si todos los demás bits se dejan con sus valores predeterminados, se ingresa en un modo idéntico al modo cbreak tradicional. En este modo, los caracteres se pasan al programa sin esperar hasta completar una línea, pero los caracteres INTR, QUIT, STARTy STOP conservan sus efectos. Todos éstos pueden inhabilitarse poniendo en cero los bits en las banderas, produciendo así el equivalente del modo crudo tradicional.

Los diversos caracteres especiales que pueden modificarse, incluidos los que son extensiones de MINIX, se guardan en el arreglo c_cc. Este arreglo también contiene dos parámetros que se usan en el modo no canónico. La cantidad MIN, almacenada en c_cc[VMIN}, especifica el número mínimo de caracteres que deben recibirse para satisfacer una llamada READ. La cantidad TIME de c_cc[VTIME} establece un límite de tiempo para tales llamadas. MIN y TIME interactúan como se muestra en la Fig. 3-35. Se ilustra una llamada que solicita N bytes. Con TIME = 0 y MIN = 1, el comportamiento es similar al modo crudo tradicional.

	TIME=0	TIME > 0
MIN=0	Regresa inmediatamente con lo que hay disponible, 0 a N bytes	El temporizador inicia de inmediato. Regresa con el primer byte introducido o con 0 bytes después del tiempo de espera
MIN>0	Regresa con por lo menos MIN y hasta N bytes. Posible bloqueo indefinido	El temporizador entre bytes se inicia después del primer byte. Devuelve N bytes si se han recibido cuando vence el tiempo o por lo menos un byte al vencer el tiempo. Posible bloqueo indefinido

Figura 3-35. MIN y TIME determinan cuándo una llamada de lectura regresa en modo no canónico. N es el número de bytes solicitados.

Software de salida

La salida es más sencilla que la entrada, pero los controladores para las terminales RS-232 son radicalmente diferentes de los controladores para las terminales con mapa en la memoria. El método que suele usarse en las terminales RS-232 es tener buffers de salida asociados con cada terminal. Los buffers pueden provenir de la misma reserva que los buffers de entrada, o ser dedicados, como en el caso de las entradas. Cuando los programas escriben en la terminal, la salida se copia primero en los buffers. De forma similar, las salidas del eco se copian también en los buffers. Una vez que todas las salidas se han copiado en los buffers (o que éstos están llenos),

se exhibe el primer carácter, y el controlador se duerme. Cuando llega la interrupción, se exhibe el siguiente carácter, y así sucesivamente.

En las terminales con mapa en la memoria se puede usar un esquema más sencillo. Los caracteres que se van a exhibir se extraen uno por uno del espacio de usuario y se colocan directamente en la RAM de video. En las terminales RS-232, cada carácter por exhibir se envía simplemente por la línea hacia la terminal. Cuando hay mapa en la memoria, algunos caracteres requieren un tratamiento especial, como el retroceso, el retomo de carro, el salto de línea y la alarma audible (CTRL-G). Un controlador para una terminal con mapa en la memoria debe seguir la pista en software a la posición actual en la RAM de video, a fin de poder colocar caracteres imprimibles ahí y avanzar la posición actual. El retroceso, el retomo de carro y el salto de línea requieren que dicha posición se actualice de forma apropiada.

En particular, cuando se produce un salto de línea en la línea inferior de la pantalla, es preciso hacer avanzar la pantalla. Para ver cómo funciona el desplazamiento, examine la Fig. 3-29. Si el controlador de video siempre comenzara a leer la RAM en OxBOOOO, la única forma de hacer avanzar la pantalla sería copiando 24 x 80 caracteres (cada uno de los cuales requiere dos bytes) de OxBOOAO a OxBOOOO, cosa que tardaría un tiempo apreciable.

Por fortuna, el hardware generalmente ofrece algo de ayuda aquí. La mayor parte de los controladores de video contienen un registro que determina de qué punto de la RAM de video deben comenzarse a tomar los caracteres para la línea superior de la pantalla. Si hacemos que este registro apunte a OxBOOAO en lugar de a OxBOOOO, la línea que antes era la número dos pasará a ser la primera, y toda la pantalla se desplazará hacia arriba una línea. La única otra cosa que el controlador en software debe hacer es copiar lo necesario en la nueva línea inferior. Cuando el controlador de video llega a la parte superior de la RAM, simplemente da la vuelta y continúa obteniendo bytes a partir de la dirección más baja.

Otro problema que el controlador en software debe resolver en una terminal con mapa en la memoria es la colocación del cursor. Una vez más, el hardware ayuda un poco en forma de un registro que indica dónde debe ir el cursor. Por último, está el problema de la alarma, la cual se produce enviando una onda senoidal o cuadrada al altavoz, que es una parte de la computadora totalmente independiente de la RAM de video.

Vale la pena señalar que muchos de los problemas que enfrenta el controlador de una terminal con pantalla mapeada en la memoria (desplazamiento, alarma, etc.), también los enfrenta el microprocesador de una terminal RS-232. Desde el punto de vista del microprocesador, él es el procesador principal de un sistema con pantalla mapeada en la memoria.

Los editores de pantalla y muchos otros programas avanzados necesitan poder actualizar la pantalla de formas más complejas que simplemente desplazar texto en la parte inferior de la pantalla. A fin de darles servicio, muchos controladores de terminal reconocen diversas secuencias de escape. Aunque algunas terminales reconocen conjuntos de secuencias de escape especiales, conviene tener un estándar que facilite la adaptación del software de un sistema a otro. El American National Standards Institute (ANSÍ) ha definido un conjunto de secuencias de escape estándar, y MINIX reconoce un subconjunto de las secuencias ANSÍ que se muestra en la Fig. 3-36 y que es suficiente para muchas operaciones comunes. Cuando el controlador detecta el carácter que inicia las secuencias de escape, enciende una bandera y espera hasta que llega el resto de la

secuencia de escape. Una vez que ha llegado todo, el controlador debe realizar la acción correspondiente en software. La inserción y eliminación de texto requieren la transferencia de bloques de caracteres dentro de la RAM de video. El hardware no ayuda más que con el desplazamiento de la pantalla y la exhibición del cursor.

Secuencia de escape	Significado
ESC [n A	Subir n líneas
ESC [n B	Bajar n líneas
ESC [n C	Moverse n espacios a la derecha
ESC [n D	Moverse n espacios a la izquierda
ESC [m: n H	Llevar el cursor a (m, n)
ESC [s J	Borrar la pantalla desde el cursor (0 al final, 1 desde el principio, 2 todo)
ESC [s K	Borrar la línea desde el cursor (0 al final, 1 desde el principio, 2 todo)
ESC [n L	Insertar n líneas en el cursor
ESC [n M	Borrar n líneas en el cursor
ESC [n P	Borrar n caracteres en el cursor
ESC [n @	Insertar n caracteres en el cursor
ESC [n m	Habilitar presentación n (0=normal, 4=negrita, 5=parpadeante, 7=¡nverso)
ESCM	Desplazar la pantalla hacia atrás si el cursor está en la línea superior

Figura 3-36. Las secuencias de escape ANSI aceptadas por el controlador de terminal para salida. ESC denota el carácter de escape ASCII (OxIB), y n, m y s son parámetros numéricos opcionales.

3.9.3 Generalidades del controlador de terminales en MINIX

El controlador de terminales está contenido en cuatro archivos en C (seis si está habilitado el apoyo de RS-232 y pseudoterminal) y juntos constituyen por mucho el controlador más grande de MINIX. El tamaño del controlador de terminales se explica en parte por la observación de que el controlador maneja tanto el teclado como la pantalla, cada uno de los cuales es un dispositivo complejo por derecho propio, así como otros dos tipos opcionales de terminales. No obstante, la mayoría de las personas se sorprende al enterarse de que la E/S de terminal requiere 30 veces más código que el planificador. (Esta sensación se refuerza si se examinan los numerosos libros sobre sistemas operativos que dedican un espacio 30 veces mayor a la planificación que a toda la E/S combinada.)

El controlador de terminal acepta siete tipos de mensajes:

1. Leer de la terminal (del sistema de archivos a nombre de un proceso de usuario).
2. Escribir en la terminal (del sistema de archivos a nombre de un proceso de usuario).
3. Establecer parámetros de la terminal para IOCTL (del sistema de archivos a nombre de un proceso de usuario).

4. E/S ocurrida durante el último tic del reloj (del manejador de interrupciones de reloj).
5. Cancelar la petición anterior (del sistema de archivos cuando ocurre una señal).
6. Abrir un dispositivo.
7. Cerrar un dispositivo.

Los mensajes para leer y escribir tienen el mismo formato que se mostró en la Fig. 3-15, excepto que no se necesita el campo POSITION. En el caso de un disco, el programa debe especificar cuál bloque desea leer. En el caso de una terminal, no hay opción: el programa siempre obtiene el siguiente carácter que se tecléa. Las terminales no realizan búsquedas.

Las funciones de POSIX `tcgetattr` y `tcsetattr` que sirven para examinar y modificar los atributos (propiedades) de las terminales están apoyadas por la llamada al sistema IOCTL. Según la práctica de programación recomendada, hay que usar estas funciones y otras que están en `include/termios.h` y dejar que la biblioteca de C convierta las llamadas de biblioteca a llamadas al sistema IOCTL. Sin embargo, hay algunas operaciones de control que MINIX necesita y que no están contempladas en POSIX, como cargar un mapa de teclas alterno, y para éstas el programador debe usar IOCTL explícitamente.

El mensaje enviado al controlador por una llamada al sistema IOCTL, contiene un código de petición de función y un apuntador. En el caso de la función `tcsetattr`, se realiza una llamada IOCTL con un tipo de petición TCSETS, TCSETSW o TCSETSF, y un apuntador a una estructura `termios` como la que se muestra en la Fig. 3-34. Todas estas llamadas reemplazan el conjunto vigente de atributos por un conjunto nuevo; las diferencias radican en que una petición TCSETS tiene efecto inmediato, una petición TCSETSW sólo surte efecto hasta que se han transmitido todas las salidas, y una TCSETSF espera a que las salidas terminen y desecha todas las entradas que todavía no se han leído. `Tcgetattr` se traduce a una llamada IOCTL con un tipo de petición TCGETS y devuelve al invocador una estructura `termios` llena para que pueda examinar el estado actual de un dispositivo. Las llamadas IOCTL que no corresponden a funciones definidas por POSIX, como la petición KIOCSMAP empleada para cargar un nuevo mapa de teclas, pasan apuntadores a otros tipos de estructuras, en este caso a una `keymap_t` que es una estructura de 1536 bytes (códigos de 16 bytes para 128 teclas X 6 modificadores). En la Fig. 3-43 se resume la forma en que las llamadas POSIX estándar se convierten en llamadas al sistema IOCTL.

El controlador de terminal usa una estructura de datos principal, `tty_table`, que es un arreglo de estructuras `tty`, una por terminal. Una PC estándar sólo tiene un teclado y una pantalla, pero MINIX puede reconocer hasta ocho terminales virtuales, dependiendo de la cantidad de memoria que tenga la tarjeta del adaptador de pantalla. Esto permite a la persona que usa la consola iniciar varias sesiones, conmutando la salida de la pantalla y la entrada del teclado de un "usuario" a otro. Con dos consolas virtuales, si oprimimos ALT-F2 seleccionaremos la segunda y con ALT-F1 regresaremos a la primera. También puede usarse ALT junto con la tecla de flecha. Además, las líneas en serie pueden apoyar a dos usuarios remotos conectados por un cable RS-232 o un módem, y las seudoterminales pueden apoyar a usuarios conectados a través de una red. El controlador se escribió con el fin de facilitar la adición de más terminales. La configuración estándar que se

ilustra en el código fuente del presente texto tiene dos consolas virtuales, con las líneas en serie y las seudoterminals inhabilitadas.

Cada estructura `tty` de `tty_table` sigue la pista tanto a las entradas como a las salidas. Para la entrada, la estructura contiene una cola de todos los caracteres que se han tecleado pero que el programa todavía no ha leído, información relativa a peticiones de lectura de caracteres que todavía no se han recibido e información de tiempos de espera para poder solicitar entradas sin -que la tarea se bloquee permanentemente si no se tecléa ningún carácter. Para la salida, la estructura contiene los parámetros de las peticiones de escritura que todavía no han terminado. Otros campos contienen diversas variables generales, como la estructura `termios` ya mencionada, que afectan muchas propiedades tanto de las entradas como de las salidas. También hay un campo en la estructura `tty` que apunta a información que se necesita para una clase específica de dispositivos pero que no tiene que estar en la entrada de `tty_table` para todos y cada uno de los dispositivos. Por ejemplo, la parte del controlador de la consola dependiente del hardware necesita la posición actual en la pantalla y en la RAM de video, y el byte de atributo vigente para la pantalla, pero esta información no es necesaria para apoyar una línea RS-232. Las estructuras de datos privadas para cada tipo de dispositivo también contienen los buffers que reciben entradas de las rutinas de servicio de interrupciones. Los dispositivos lentos, como el teclado, no necesitan buffers tan grandes como los que necesitan los dispositivos rápidos.

Entradas de terminales

A fin de entender mejor el funcionamiento del controlador, examinemos primero la forma como los caracteres que se teclean en la terminal se abren camino a través del sistema hasta el programa que los necesita.

Cuando un usuario inicia una sesión en la consola del sistema, se crea un shell para él con `/dev/console` como entrada estándar, salida estándar y error estándar. El shell inicia y trata de leer de la entrada estándar invocando el procedimiento de biblioteca `read`. Este procedimiento envía al sistema de archivos un mensaje que contiene el descriptor de archivo, la dirección del buffer y la cuenta. Este mensaje está rotulado como (1) en la Fig. 3-37. Después de enviar el mensaje, el shell se bloquea, esperando la respuesta. (Los procesos de usuario sólo ejecutan la primitiva `SEND_REC`, que combina un `SEND` con un `RECEIVE` del proceso al que se envió.)

El sistema de archivos recibe el mensaje y localiza el nodo `i` que corresponde al descriptor de archivo especificado. Este nodo `i` es el del archivo especial por caracteres `/dev/console` y contiene los números de dispositivo principal y secundario de la terminal. El número de dispositivo principal para las terminales es 4; para la consola, el número de dispositivo secundario es 0.

El sistema de archivos utiliza estos números como índices para buscar en su mapa de dispositivos, `dmap`, el número de la tarea de la terminal, y luego envía a dicha tarea un mensaje que «parece como (2) en la Fig. 3-37. Normalmente, el usuario todavía no habrá tecleado nada, así que el controlador de la terminal no podrá satisfacer la petición. El controlador devuelve de inmediato una respuesta para desbloquear el sistema de archivos e informar que no hay caracteres disponibles; esta respuesta aparece como (3). El sistema de archivos toma nota del hecho de que un proceso está esperando entradas de la terminal, registrándolo en la estructura de la consola en

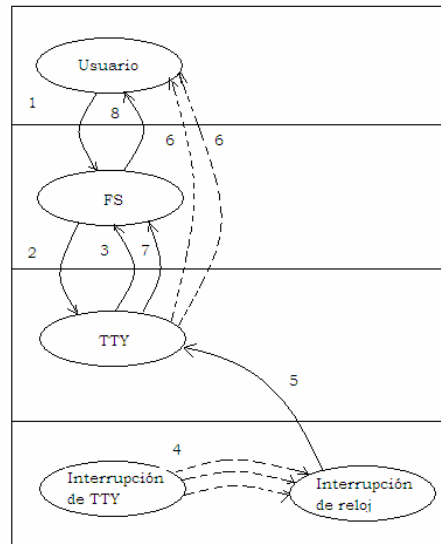


Figura 3-37. Petición de lectura de la terminal cuando no hay caracteres pendientes. FS es el sistema de archivos. TTY es la tarea de la terminal. El manejador de interrupciones de la terminal pone en cola los caracteres conforme se introducen, pero es el manejador de interrupciones del reloj el que despierta a TTY.

tty_table, y luego se pone a obtener la siguiente petición de trabajo. Desde luego, el shell del usuario permanece bloqueado hasta que llegan los caracteres solicitados.

Cuando por fin se escribe un carácter en el teclado, causa dos interrupciones: una cuando la tecla se oprime y otra cuando se suelta. Esta regla también aplica a las teclas modificadoras como CTRL y SHIFT, que no transmiten datos por sí mismas pero que de todos modos causan dos interrupciones por tecla. La interrupción del teclado es IRQ 1, y `_hwint01` en el archivo de código de ensamblador `mpx386.s` activa `kbd_hw_int` (línea 13123), que a su vez invoca `scan_keyboard` (línea 13432) para extraer el código de tecla del hardware del teclado. Si el código es el de un carácter ordinario, se coloca en la cola de entrada del teclado, `ibuf`, si la interrupción se generó al oprimirse una tecla, pero se ignora si la interrupción se generó al soltarse una tecla. Los códigos para las teclas modificadoras como CTRL y SHIFT se colocan en la cola cuando ocurren ambos tipos de interrupciones, pero pueden distinguirse después por un bit que se pone en 1 cuando se suelta una tecla. Observe que en este punto los códigos recibidos y almacenados en `ibuf` no son códigos ASCII; simplemente son los códigos de escrutinio producidos por el teclado IBM. A continuación, `kbd_hw_int` enciende una bandera, `tty_events` (parte de la sección de `tty_table` que corresponde al teclado), invoca `force_timeout` para forzar el vencimiento del tiempo de espera, y regresa

A diferencia de algunas otras rutinas de servicio de interrupciones, `kbd_hw_int` no envía IBM ensaje para despertar la tarea de la terminal. La llamada a `forceJimeout` se indica con las líneas interrumpidas marcadas con (4) en la figura. Estas no son mensajes; establecen la variable `tty_timeout` en el espacio de direcciones común a todas las rutinas de servicio de interrupciones. En la siguiente interrupción del reloj, `clock_handler` ve que `tty_timeout` indica que es el momento de invocar `tty_wakeup` (línea 11452), la cual envía un mensaje (5) a la tarea de la terminal para

despertarla. Observe que si bien el código fuente para `tty_wakeup` está en el archivo `tty.c`, se ejecuta en respuesta a la interrupción de reloj, y por esta razón decimos que la interrupción de reloj envía el mensaje a la tarea de la terminal. Si las entradas están llegando rápidamente, pueden ponerse en cola de esta forma varios códigos de carácter, y es por ello que se muestran múltiples llamadas `aforce_timeout` (4) en la figura.

Al recibir el mensaje de despertar, la tarea de terminal examina la bandera `tty_events` de cada dispositivo de terminal y, para cada dispositivo que tiene su bandera izada, invoca `handle_events` (línea 12256). La bandera `tty_events` puede indicar diversos tipos de actividades (aunque la entrada es el más probable), así que `handle_events` siempre invoca las funciones específicas para el dispositivo tanto para las entradas como para las salidas. Cuando hay entradas del teclado, el resultado es una llamada a `kb_read` (línea 13165), que sigue la pista a los códigos de teclado que indican la pulsación o liberación de las teclas CTRL, SHIFT y ALT y convierte los códigos del teclado en códigos ASCII. A su vez, `kb_read` invoca `in_process` (línea 12367), que procesa los códigos ASCII, teniendo en cuenta los caracteres especiales y las diferentes banderas que pueden estar izadas, incluida la que indica si está vigente o no el modo canónico. El efecto normal es agregar caracteres a la cola de entrada de la consola en `tty_table`, aunque algunos códigos, como por ejemplo BACKSPACE (retroceso), tienen otros efectos. Normalmente, `in_process` inicia también el eco de los códigos ASCII a la pantalla.

Una vez que han llegado suficientes caracteres, la tarea de la terminal invoca el procedimiento en lenguaje ensamblador `phys_copy` para copiar los datos en la dirección solicitada por el shell. Esta operación tampoco es una transferencia de mensajes, y por esa razón se muestra con las líneas punteadas (6) en la Fig. 3-37. Se muestra más de una línea de éstas porque pueden realizarse más de una operación de este tipo antes de que se satisfaga plenamente la petición del usuario. Una vez completada la operación, el controlador de la terminal envía un mensaje al sistema de archivos para informarle que ya se llevó a cabo el trabajo (7), y el sistema de archivos reacciona a este mensaje enviando un mensaje de vuelta al shell para desbloquearlo (8).

La decisión de cuándo es que han llegado suficientes caracteres depende del modo de la terminal. En modo canónico se considera que una petición está completa cuando se recibe un código de salto de línea, fin de línea o fin de archivo y, a fin de realizar correctamente el procesamiento de las entradas, una línea de entrada no puede exceder el tamaño de la cola de entrada. En el modo no canónico una lectura puede solicitar un número mucho mayor de caracteres, y es posible que `in_process` tenga que transferir caracteres más de una vez antes de que se devuelva un mensaje al sistema de archivos para indicar que la operación se ha completado.

Observe que el controlador de la terminal copia los caracteres reales directamente de su propio espacio de direcciones al del shell, sin pasar primero por el sistema de archivos. En el caso de E/S por bloques, los datos sí pasan por el sistema de archivos para que éste pueda mantener un buffer caché de los bloques más recientemente utilizados. Si un bloque solicitado está en el caché, el sistema de archivos podrá satisfacer directamente la petición sin tener que realizar E/S de disco.

En el caso de la E/S de terminal, no tiene sentido tener un caché. Además, una petición del sistema de archivos a un controlador en software de disco siempre puede satisfacerse en unos cientos de milisegundos, como máximo, por lo que no hay problema si se obliga al sistema de archivos a esperar. La E/S de terminal puede tardar horas en completarse, o nunca completarse

(en modo canónico la tarea de la terminal espera una línea completa, y también podría tener que esperar un tiempo largo en modo no canónico, dependiendo de los valores de MINy TIME). Por tanto, no es aceptable hacer que el sistema de archivos se bloquee hasta que se haya satisfecho una petición de entrada de la terminal.

Más adelante, puede suceder que el usuario se haya adelantado tecleando, y que los caracteres estén disponibles antes de ser solicitados, por haber ocurrido previamente los sucesos 4 y 5. En tal caso, los sucesos 1, 2, 6, 7 y 8 ocurrirán en rápida sucesión después de la petición de lectura; el suceso 3 jamás ocurrirá.

Si sucede que la tarea de terminal se está ejecutando cuando ocurre una interrupción de reloj, no se le podrá enviar ningún mensaje porque no estará esperando recibirlo. Sin embargo, a fin de mantener las entradas y salidas fluyendo continuamente cuando la tarea de la terminal está ocupada, se inspeccionan en varios momentos las banderas `tty_events` de todos los dispositivos terminales, por ejemplo, inmediatamente después de procesar un mensaje y contestarlo. Por tanto, es posible agregar caracteres a la cola de la consola sin la ayuda de un mensaje de despertar enviado por el reloj. Si ocurren dos o más interrupciones de reloj antes de que el controlador de la terminal termine lo que está haciendo, todos los caracteres se almacenarán en `ibuf`, y se encenderá repetidamente `tty_flags`. En última instancia, la tarea de la terminal recibe un mensaje; el resto se pierde. Sin embargo, como todos los caracteres están almacenados en el buffer, no se pierden entradas tecleadas. Incluso es posible que, para cuando la tarea de la terminal recibe un mensaje, la entrada está completa y ya se envió una respuesta al proceso de usuario.

El problema de qué hacer en un sistema de mensajes sin buffers (principio de cita) cuando una rutina de interrupción desea enviar un mensaje a un proceso que está ocupado es inherente a este tipo de diseño. En la mayor parte de los dispositivos, como los discos, las interrupciones sólo ocurren como respuesta a comandos emitidos por el controlador en software, así que sólo puede haber una interrupción pendiente en un momento dado. Los únicos dispositivos que generan interrupciones por sí solos son el reloj y las terminales (y, cuando está habilitada, la red). El reloj se maneja contando los tics pendientes, así que si la tarea del reloj no recibe un mensaje del manejador de interrupciones del reloj, puede compensarlo posteriormente. Las terminales se manejan haciendo que la rutina de interrupciones acumule los caracteres en un buffer y encienda una bandera para indicar que se han recibido caracteres. Si la tarea de terminal se está ejecutando, examinará todas estas banderas antes de dormirse y no se dormirá si tiene más trabajo que hacer.

La tarea de la terminal no es despertada directamente por interrupciones de terminal debido al excesivo gasto extra que ello implicaría. El reloj envía una interrupción a la tarea de la terminal en el siguiente tic después de cada interrupción de terminal. Escribiendo 100 palabras por minuto, un mecanógrafo teclaea menos de 10 caracteres por segundo. Incluso con un mecanógrafo rápido es probable que se envíe a la tarea de la terminal un mensaje de interrupción por cada carácter tecleado, aunque podría ser que se perdieran algunos de esos mensajes. Si el buffer se llena antes de ser vaciado, los caracteres en exceso se desecharán, pero la experiencia muestra que, en el caso de un teclado, basta con un buffer de 32 caracteres. En el caso de otros dispositivos de entrada pueden encontrarse tasas de datos más altas; un puerto en serie conectado a un módem de 28 800 bps puede generar tasas 1000 o más veces más rápidas que las de un mecanógrafo. A tales velocidades, el módem podría recibir aproximadamente 48 caracteres entre dos tics del reloj,

pero si se tiene en cuenta la compresión de datos en el enlace de módem el puerto en serie conectado al módem debe poder manejar por lo menos el doble de caracteres. Para las líneas en serie, MINIX proporciona un buffer de 1024 caracteres.

Pensamos que es una lástima que la tarea de la terminal no pueda implementarse sin sacrificar algunos de nuestros principios de diseño, pero el método que usamos realiza el trabajo sin aumentar demasiado la complejidad del software y sin pérdida de rendimiento. La alternativa obvia, desechar el principio de cita y hacer que el sistema guarde en buffers todos los mensajes enviados a destinos que no los están esperando, es mucho más complicada y también más lenta.

Los diseñadores de sistemas reales a menudo enfrentan la decisión de usar el caso general, que es elegante todo el tiempo pero un tanto lento, o usar técnicas más sencillas que, por lo regular, son rápidas pero que en uno o dos casos requieren un truco para funcionar correctamente. Realmente, la experiencia es la única guía para decidir cuál enfoque es mejor en circunstancias específicas. Lampson (1984) y Brooks (1975) resumen una cantidad considerable de experiencia en el diseño de sistemas operativos. Aunque estas referencias son viejas, no dejan de ser clásicas.

Concluiremos nuestra reseña de las entradas de terminal resumiendo los sucesos que ocurren cuando una petición de lectura activa inicialmente la tarea de terminal y cuando ésta se reactiva después de recibir entradas del teclado (véase la Fig. 3-38). En el primer caso, cuando llega un mensaje a la tarea de terminal solicitando caracteres del teclado, el procedimiento principal, `tty_task` (línea 11817), invoca `do_read` (línea 11891) para manejar la petición. `do_read` almacena los parámetros de la llamada en la entrada correspondiente al teclado en `tty_table`, en caso de que no haya suficientes caracteres en el buffer para satisfacer la petición.

A continuación, `do_read` invoca `in_transfer` (línea 12303) para obtener las entradas que ya estén esperando, y luego `handle_events` (línea 12256), que a su vez invoca `kb_read` (línea 13165) y luego `intransfer` otra vez, para tratar de obtener unos cuantos caracteres más del flujo de 8 entrada. `Kb_read` invoca otros procedimientos varios que no se muestran en la Fig. 3-38 para hacer su trabajo. El resultado es que todo lo que está inmediatamente disponible se copia al «uario. Si no hay nada disponible, no se copia nada. Si `in_transfer` o `handle_events` completan la lectura, se envía un mensaje al sistema de archivos una vez que se han transferido todos los caracteres, a fin de que el sistema de archivos pueda desbloquear al invocador. Si no se completó la lectura (no hubo caracteres, o no hubo suficientes caracteres), `do_read` informa de ello al sistema de archivos, indicándole si debe suspender al invocador original o, si se solicitó una lectura no bloqueadora, cancelar la lectura.

El lado derecho de la Fig. 3-38 resume los sucesos que ocurren cuando se despierta la tarea de terminal después de una interrupción del teclado. Cuando se tecldea un carácter, el procedimiento de interrupción `kb_hw_int` (línea 13123) coloca el código de carácter recibido en el buffer del teclado, enciende una bandera para indicar que el dispositivo de la consola experimentó un evento, luego hace lo necesario para que en el siguiente tic del reloj se venza un tiempo de espera. La tarea del reloj envía un mensaje a la tarea de la terminal diciéndole que algo sucedió. Al recibir este mensaje, `tty_task` examina las banderas de eventos de todos los dispositivos terminales e invoca `handle_event` para cada dispositivo que tiene la bandera izada. En el caso del teclado, `handle_event` invoca `kb_read` e `in_transfer`, tal como se hizo al recibirse la petición de lectura original. Los sucesos que se muestran en el lado derecho de la figura pueden ocurrir varias veces, hasta que se reciben

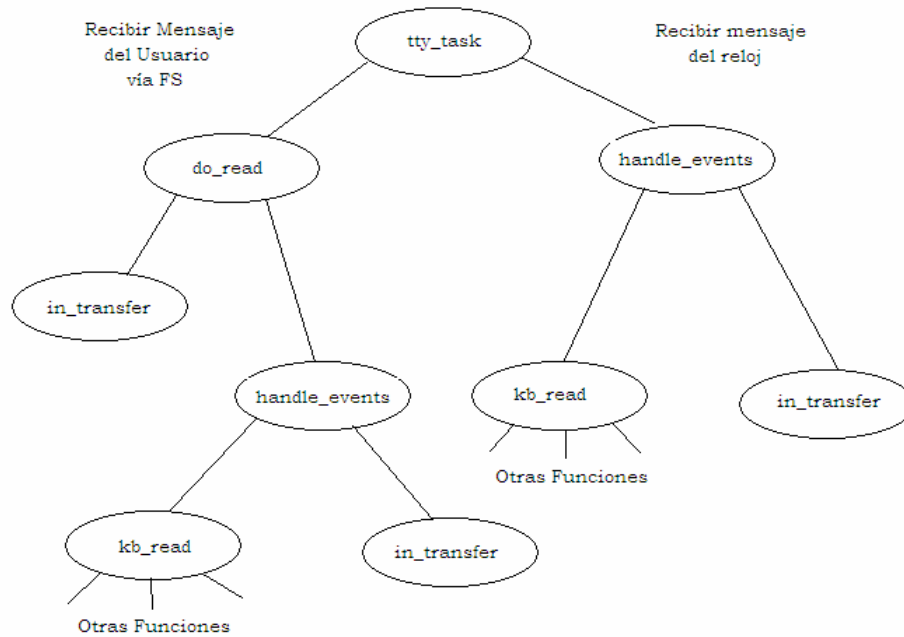


Figura 3-38. Manejo de entradas en el controlador de terminal. La rama izquierda del árbol se toma cuando se va a procesar una petición de lectura de caracteres. La rama derecha se toma cuando se envía al controlador un mensaje de "se tecleó un carácter".

suficientes caracteres para satisfacer la petición aceptada por `do_read` después del primer mensaje del sistema de archivos. Si éste trata de iniciar una petición pidiendo más caracteres al mismo dispositivo antes de completarse la primera petición, se devolverá un error. Desde luego, cada dispositivo es independiente; una petición de lectura a nombre de un usuario en una terminal remota se procesa independientemente de una hecha a nombre de un usuario en la consola.

Las funciones que no se muestran en la Fig. 3-38 y que son invocadas por `kb_read` incluye `map_key`, que convierte los códigos de tecla (códigos de detección) generados por el hardware en códigos ASCII, `make_break`, que se mantiene al tanto del estado de las teclas modificadoras como CTRL y SHIFT, e `in_process`, que maneja complicaciones tales como intentos por parte del usuario de retroceder borrando entradas tecleadas por equivocación, otros caracteres especiales y opciones disponibles en los diferentes modos de entrada. `in_process` también invoca `echo`(línea 12531) para que los caracteres tecleados aparezcan en la pantalla.

Salidas a terminales

En general, la salida de la consola es más sencilla que la entrada de terminales, porque el sistema operativo está en control y no necesita preocuparse por peticiones de salida que lleguen en momento poco propicios. Además, como la consola de MINIX es una pantalla con mapa en la memoria, la salida a ella es más sencilla aún. No se requieren interrupciones; el funcionamiento básico consiste

en copiar datos de una región de la memoria a otra. Por otro lado, todos los detalles de controlar la pantalla, incluido el manejo de secuencias de escape, corre por cuenta del controlador en software. Como hicimos para las entradas del teclado en la sección anterior, rastreamos los pasos que se siguen para enviar caracteres a la pantalla de la consola. Supondremos en este ejemplo que se está escribiendo en la pantalla activa; las complicaciones menores causadas por las consolas virtuales se analizarán más adelante.

Cuando un proceso desea exhibir algo, generalmente invoca `printf`, la cual invoca `WRITE` para enviar un mensaje al sistema de archivos. El mensaje contiene un apuntador a los caracteres por exhibir (no los caracteres mismos). A continuación, el sistema de archivos envía un mensaje al controlador de la terminal, que obtiene los caracteres y los copia en la RAM de video. La Fig. 3-39 muestra los principales procedimientos que intervienen en las salidas.

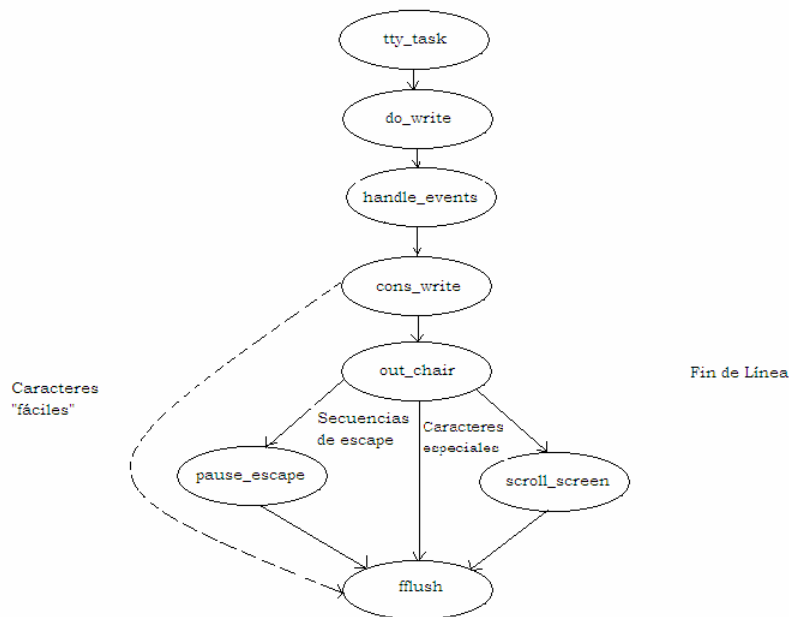


Figura 3-39. Principales procedimientos empleados para la salida a terminales. La línea punteada indica caracteres copiados directamente en ramqueue por `cons_write`.

Cuando llega un mensaje a la tarea de la terminal solicitándole que escriba en la pantalla, se invoca `do_write` (línea 11964) para almacenar los parámetros en la estructura `tty` de la consola que

está en `tty_table`. Luego se invoca `handle_events` (la misma función que se llama siempre que se encuentra izada la bandera `tty_events`). En cada llamada, esta función invoca las rutinas tanto de entrada como de salida para el dispositivo seleccionado en su argumento. En el caso de la pantalla de la consola, esto implica que primero se procesarán todas las entradas de teclado que estén esperando. Si hay entradas esperando, los caracteres de los que debe hacerse eco se agregan a los que ya estén esperando en la salida. Luego se hace una llamada a `cons_write` (línea 13729), el procedimiento de salida para las pantallas con mapa en la memoria. Este procedimiento emplea `phys_copy` para copiar bloques de caracteres del proceso de usuario a un buffer local, posiblemente repitiendo éste y los siguientes pasos varias veces, ya que al buffer local sólo le caben 64 bytes. Cuando se llena el buffer local, cada byte de 8 bits se transfiere a otro buffer, `ramqueue`, que es un arreglo de palabras de 16 bits. Cada segundo byte se llena con el valor actual del byte de atributo de la pantalla, que determina los colores de primer y segundo plano y otros atributos. Si es posible, los caracteres se transfieren directamente a `ramqueue`, pero ciertos caracteres, como los de control o los que forman parte de secuencias de escape, requieren un manejo especial. El manejo especial también es necesario cuando la posición de un carácter en la pantalla excedería la anchura de ésta, o cuando `ramqueue` se llena. En estos casos se invoca `out_char` (línea 13809) para transferir los caracteres y realizar todas las demás acciones pertinentes. Por ejemplo, se invoca `scroll_screen` (línea 13896) cuando se recibe un salto de línea mientras se está direccionando la última línea de la pantalla, y `parse_escape` maneja los caracteres durante una secuencia de escape. Por lo regular, `out_char` invoca `flush` (línea 13951), que copia el contenido de `ramqueue` en la memoria de la pantalla usando la rutina en lenguaje ensamblador `mem_vid_copy`. También se invoca `flush` después de que el último carácter se transfiere a `ramqueue` para asegurarse que se exhiban todas las salidas. El resultado final de `flush` es ordenar al chip controlador de video 6845 que exhiba el cursor en la posición correcta.

Lógicamente, los bytes obtenidos del proceso de usuario podrían escribirse en la RAM de video, uno en cada iteración del ciclo. Sin embargo, es más eficiente acumular los caracteres en `ramqueue` y luego copiar el bloque con una llamada a `mem_vid_copy` en el entorno de memoria protegida de los procesadores clase Pentium. Resulta interesante que esta técnica se introdujo en las primeras versiones de MINIX que se ejecutaba en los procesadores más viejos sin memoria protegida. El precursor de `mem_vid_copy` resolvía un problema de temporización: en las pantallas antiguas el copiado en la memoria de video tenía que efectuarse mientras la pantalla estaba en blanco durante el retrazado vertical del haz del CRT para no generar basura visual en toda la pantalla. MINIX ya no ofrece este apoyo para equipo obsoleto, pues el precio que se paga en términos de rendimiento es excesivo. Sin embargo, la versión moderna de MINIX obtiene otros beneficios de copiar `ramqueue` como bloque.

La RAM de video disponible para una consola está delimitada en la estructura `consolé` por los campos `c_start` y `c_limit`. La posición actual del cursor se almacena en los campos `c_column` y `c_row`. La coordenada (0, 0) está en la esquina superior izquierda de la pantalla, que es donde el hardware comienza a llenar la pantalla. Cada barrido de video comienza en la dirección dada por `c_org` y continúa durante 80 x 25 caracteres (4000 bytes). En otras palabras, el chip 6845 tómalas palabra que está a una distancia `c_org` del principio de la RAM de video y exhibe el byte del carácter en la esquina superior izquierda, usando el byte de atributo para controlar el color, el

parpadeo, etc. Luego, el chip obtiene la siguiente palabra y exhibe el carácter en (1, 0). Este proceso continúa hasta llegar a (79, 0), momento en el cual se inicia la segunda línea de la pantalla, en la coordenada (0, 1).

Al arranque de la computadora, la pantalla se despeja, se escriben salidas en la RAM de video comenzando en la posición `c_start`, y se asigna a `c_org` el mismo valor que tiene `c_start`. Por tanto, la primera línea aparece en la línea superior de la pantalla. Cuando es necesario exhibir salidas en una nueva línea, sea porque la primera está llena o porque `out_char` detectó un carácter de nueva línea, las salidas se escriben en la posición dada por `c_start` más 80. Tarde o temprano se llenan las 25 líneas, y es necesario desplazar, haciendo avanzar la pantalla. Algunos programas, como los editores, requieren también desplazamiento hacia abajo, cuando el cursor está en la línea superior y es necesario moverse más hacia arriba dentro del texto.

Hay dos formas de manejar el desplazamiento de la pantalla. En el desplazamiento por software, el carácter que se exhibirá en la posición (0, 0) siempre está en la primera posición de la memoria de video, la palabra O relativa a la posición a la que apunta `c_start`, y se ordena al chip controlador de video que exhiba esta posición primero colocando la misma dirección en `c_prg`. Cuando se debe desplazar la pantalla, el contenido de la posición relativa 80 de la RAM de video, el principio de la segunda línea de la pantalla, se copia en la posición relativa 0, la palabra 81 se copia en la posición relativa 1, y así sucesivamente. La secuencia de barrido no cambia, poniendo los datos que están en la posición 0 de la memoria en la posición (0,0) de la pantalla, y parece que la imagen se ha desplazado una línea hacia arriba en la pantalla. El costo es que la CPU ha movido $80 \times 24 = 1920$ palabras. En el desplazamiento por hardware los datos no cambian de lugar en la memoria; en lugar de ello, se le ordena al chip controlador de video que inicie la exhibición en un punto distinto, por ejemplo, los datos que están en la posición 80. La contabilización se efectúa sumando 80 al contenido de `c_org`, guardándolo para referencia futura, y escribiendo este valor en el registro correcto del chip controlador de video. Para ello se requiere que el chip controlador téngala suficiente inteligencia como para manejar la RAM de video en forma "circular", tomando los datos del principio de la RAM (la dirección que está en `c_start`) después de haber llegado al final (la dirección que está en `c_limif`), o bien que la RAM de video tenga más capacidad que las $80 \times 24 = 1920$ palabras necesarias para almacenar una sola pantalla. Los adaptadores de pantalla más viejos generalmente tienen una memoria más reducida pero pueden continuar del final al principio y realizar desplazamiento por hardware. Los adaptadores más nuevos generalmente tienen mucha más memoria que la necesaria para exhibir una sola pantalla de texto, pero los chips controladores no pueden continuar del final al principio. Así, un adaptador con 32768 bytes de memoria de pantalla puede contener 204 líneas completas de 160 bytes cada una, y puede realizar desplazamiento por hardware 179 veces antes de que el hecho de no poder continuar del final al principio se convierta en un problema. Tarde o temprano, se requerirá una operación de copiado de memoria para transferir los datos de las últimas 24 líneas otra vez a la posición 0 de la memoria de video. Sea cual sea el método empleado, se copia una fila de espacios en blanco en la RAM de video para asegurar que la nueva línea en la parte inferior de la pantalla esté vacía.

Cuando se configuran consolas virtuales, la memoria disponible dentro de un adaptador de video se divide equitativamente entre el número de consolas deseadas inicializando debidamente los campos `c_start` y `c_limit` para cada consola. Esto afecta el desplazamiento de pantalla. En

cualquier adaptador con el tamaño suficiente para apoyar consolas virtuales, el desplazamiento por software ocurre con cierta frecuencia, incluso si nominalmente está vigente el desplazamiento por hardware. Cuanto menor sea la cantidad de memoria disponible para cada pantalla de consola, con mayor frecuencia será necesario usar el desplazamiento por software. Se llega al límite cuando se configura el número máximo posible de consolas. Entonces, toda operación de desplazamiento tendrá que efectuarse por software.

La posición del cursor relativa al principio de la RAM de video se puede deducir de `c_column` y `c_row`, pero es más rápido almacenarla explícitamente (en `c_cur`). Cuando se va a exhibir un carácter, se coloca en la RAM de video en la posición `c_cur`, que luego se actualiza, lo mismo que `c_column`. En la Fig. 3-40 se resumen los campos de la estructura `consolé` que afectan la posición actual y el origen de la pantalla.

Campo	Significado
<code>c_start</code>	Inicio de la memoria de video para esta consola
<code>c_limit</code>	Límite de la memoria de video para esta consola
<code>c_column</code>	Columna actual (0-79) con O a la izquierda
<code>c_row</code>	Fila actual (0-24) con O arriba
<code>c_cur</code>	Distancia del cursor a partir del principio de la RAM de video
<code>c_org</code>	Posición en RAM a la que apunta el registro base del 6845

[Figura 3-40. Campos de la estructura `consolé` relacionados con la posición actual en la pantalla.

Los caracteres que afectan la posición del cursor (p. ej., salto de línea, retroceso) se manejan ajustando los valores de `c_column`, `c_row` y `c_cur`. Este trabajo se realiza al final de `flush` mediante una llamada a `set_6845`, que establece los registros del chip controlador de video. El controlador en software de la terminal reconoce secuencias de escape que permiten a los editores de pantalla y otros programas interactivos actualizar la pantalla de forma flexible. Las secuencias reconocidas son un subconjunto de un estándar ANSÍ y deberán ser suficientes para permitir que muchos programas escritos para otro hardware y otros sistemas operativos se trasladen fácilmente a MINIX. Hay dos categorías de secuencias de escape: las que nunca contienen un parámetro variable y las que pueden contener parámetros. En la primera categoría el único representante reconocido por MINIX es `ESC M`, que indiza la pantalla a la inversa, subiendo el cursor una línea y desplazando la pantalla hacia abajo si el cursor ya está en la primera línea. La otra categoría puede tener uno o dos parámetros numéricos. Todas las secuencias de este grupo comienzan con `ESC [`. El "[" es el introductor de la secuencia de control. En la Fig. 3-36 se mostró una tabla de las secuencias de escape definidas por el estándar ANSÍ y reconocidas por MINIX.

El análisis sintáctico de las secuencias `d& escape` no es trivial. Las secuencias de escape válidas en MINIX pueden tener sólo dos caracteres, como en `ESC M`, o un máximo de ocho caracteres en el caso de una secuencia que acepta dos parámetros numéricos, cada uno de los cuales puede

tener un valor de dos dígitos, como en ESC [20;60H, que mueve el cursor a la línea 20, columna 60. En una secuencia que acepta un parámetro, éste puede omitirse, y en una secuencia que acepta dos parámetros puede omitirse cualquiera de ellos, o ambos. Cuando se omite un parámetro o se usa uno que está fuera del intervalo válido, se sustituye un valor predeterminado, que es el valor permitido más bajo.

Considere las siguientes formas de construir una secuencia para llevar el cursor a la esquina superior izquierda de la pantalla:

1. ESC [H es aceptable, porque si no se incluyen parámetros se suponen los parámetros válidos más bajos.
2. ESC [1 ;1H envía correctamente el cursor a la fila 1 y la columna 1 (con ANSÍ, los números de fila y de columna comienzan en 1).
3. Tanto ESC [1 ;H como ESC [;1H tienen un parámetro omitido, que por omisión es 1 igual que en el primer ejemplo.
4. ESC [0;0H hace lo mismo, pues como ambos parámetros son menores que el valor mínimo permitido, se sustituye el mínimo.

Presentamos estos ejemplos no para sugerir que debemos usar deliberadamente parámetros no, válidos, sino para poner de relieve que el código que analiza sintácticamente tales secuencias no es trivial.

MINIX implementa un autómata de estados finitos para realizar este análisis sintáctico. La variable `c_esc_state` de la estructura `consolé` normalmente tiene el valor 0. Cuando `out_char` detecta un carácter ESC, cambia `c_esc_state` a 1, y los caracteres subsecuentes son procesados por `parse_e'cape` (línea 13986). Si el siguiente carácter es el introductor de secuencias de control, se ingresa en el estado 2; de lo contrario, se considera que la secuencia está completa y se invoca `do_escape` (línea 14045). En el estado 2, en tanto los caracteres que llegan sean numéricos, se calculará un parámetro multiplicando el valor anterior del parámetro (que inicialmente es 0) por 10 y sumándole el valor numérico del carácter actual. Los valores de los parámetros se guardan en un arreglo, y cuando se detecta un signo de punto y coma el procesamiento pasa a la siguiente celda del arreglo. (El arreglo en MINIX sólo tiene dos elementos, pero el principio es el mismo.) Cuando se encuentra un carácter no numérico que no es un signo de punto y coma se considera que la secuencia está completa, y se vuelve a invocar `do_escape`. El carácter vigente en el momento de entrar en `do_escape` se usa para seleccionar exactamente la acción por realizar y la forma de interpretar los parámetros, ya sea los predefinidos o los introducidos en el flujo de caracteres. Esto se ilustra en la Fig. 3-48.

Mapas de teclas cargables

El teclado de IBM PC no genera códigos ASCII directamente. Cada tecla se identifica con un número, comenzando por las teclas situadas arriba a la izquierda en el teclado de PC original: 1 para la tecla "ESC", 2 para el "I", y así sucesivamente. A cada tecla se asigna un número,

incluidas las teclas modificadoras como las teclas SHIFT izquierda y SHIFT derecha, números 42 y 54. Cuando se oprime una tecla, MINIX recibe el número de tecla como código de escrutinio o detección. También se genera un código de detección cuando se suelta una tecla, pero en este caso el bit más significativo es 1 (lo que equivale a sumar 128 al número de tecla). Esto permite distinguir entre la pulsación y la liberación de una tecla. Si se sabe cuáles teclas modificadoras se han presionado pero todavía no se han soltado, se puede obtener un gran número de combinaciones. Desde luego, para fines ordinarios las combinaciones de dos dedos, como SHIFT-A o CTRL-D, son las más fáciles de digitar para quienes escriben con las dos manos, pero en ocasiones especiales pueden usarse combinaciones de tres (o más) teclas, como CTRL-SHIFT-A o la conocida combinación CTRL-ALT-DEL que los usuarios de PC reconocen como la forma de restablecer y reiniciar el sistema.

La complejidad del teclado de la PC ofrece una gran flexibilidad de uso. Un teclado estándar tiene definidas 47 teclas de caracteres ordinarios (26 alfabéticos, 10 numéricos y 11 de puntuación). Si estamos dispuestos a usar combinaciones de teclas modificadoras de tres dedos, como CTRL-ALT-SHIFT, podemos reconocer un conjunto de caracteres de 376 (8 x 47) miembros. Esto de ninguna manera es el límite de lo posible, pero por ahora supongamos que no nos interesa distinguir entre las teclas modificadoras de la mano izquierda y de la derecha, ni usar las teclas de funciones ni las del subteclado numérico. De hecho, no estamos limitados a usar sólo las teclas CTRL, ALT y SHIFT como modificadores; podríamos retirar algunas teclas del conjunto de teclas ordinarias y usarlas como modificadores si quisiéramos escribir un controlador que apoyara tal sistema.

Los sistemas operativos que emplean tales teclados se valen de un mapa de teclas para determinar qué código de carácter deben pasar a un programa con base en la tecla que se está pulsando y los modificadores que están activos. El mapa de teclas de MINIX lógicamente es un arreglo de 128 filas, que representan los posibles valores de código de detección (se escogió este tamaño para poder manejar teclados japoneses; los teclados estadounidenses y europeos no tienen tantas teclas) y seis columnas. Las columnas representan ningún modificador, la tecla SHIFT, la tecla CTRL, la tecla ALT izquierda, la tecla ALT derecha y una combinación de cualquier tecla ALT con la tecla SHIFT. Por tanto, hay 720 ((128-6) x 6) códigos de carácter que se pueden generar con este esquema si se cuenta con un teclado adecuado. Esto requiere que cada entrada de la tabla sea una cantidad de 16 bits. En el caso de los teclados estadounidenses, las columnas ALT y ALT2 son idénticas. ALT2 se llama ALTGR en teclados para otros idiomas, y muchos de estos mapas de teclas reconocen teclas con tres símbolos usando esta tecla como modificador.

Un mapa de teclas estándar (determinado por la línea

```
#include keymaps/us-std.src
```

de keyboard.c) se incorpora en el kernel de MINIX en el momento de la compilación, pero se puede usar una llamada

```
ioctl(0, KIOCSMAP, keymap)
```

para cargar un mapa diferente en el kernel en la dirección keymap. Un mapa de teclas completo ocupa 1536 bytes (128 x 6 x 2). Los mapas de teclas extra se almacenan en una forma comprimi-

da. Se usa un programa llamado `genmap` para crear un nuevo mapa de teclas comprimido. Una vez compilado, `genmap` incluye el código `keymap.src` para un mapa de teclas en particular, así que el mapa se compila dentro de `genmap`. Normalmente, `genmap` se ejecuta inmediatamente después de compilarse, y en ese momento escribe la versión comprimida en un archivo y se borra el código binario de `genmap`. El comando `loadkeys` lee un mapa de teclas comprimido, lo expande internamente, y luego invoca `IOCTL` para transferir el mapa de teclas a la memoria del kernel. MINIX puede ejecutar `loadkeys` automáticamente al iniciarse, y además el usuario puede invocar este programa en el momento que desee.

Código de detección	Carácter	Normal	SHIFT	ALT1	ALT2	ALT+SHIFT	CTRL
00	ninguno	0	0	0	0	0	0
01	ESC	C('T')	C('T')	CA('T')	CA('T')	CA('T')	C('T')
02	'1'	'1'	'1'	A('1')	A('1')	A('1')	C('A')
13	'='	'='	'+'	A('=')	A('=')	A('+')	C('@')
16	'q'	L('q')	'Q'	A('q')	A('q')	A('Q')	C('Q')
28	CR/LF	C('M')	C('M')	CA('M')	CA('M')	CA('M')	C('J')
29	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL
59	F1	F1	SF1	AF1	AF1	ASF1	CF1
127	???	0	0	0	0	0	0

Figura 3-41. Unas cuantas entradas de un archivo fuente de mapa de teclas.

El código fuente de un mapa de teclas define un arreglo inicializado grande, y a fin de ahorrar espacio no se ha imprimido el archivo del mapa de teclas junto con el código fuente. La Fig. 3-41 muestra en forma tabular el contenido de unas cuantas líneas de `src/kernel/keymaps/usstd.src` que ilustran varios aspectos de los mapas de teclas. El teclado de la IBM PC no cuenta con una tecla que genere un código de detección de 0. La entrada para el código 1, la tecla ESC, muestra que el valor devuelto no se modifica cuando se oprimen SHIFT o CTRL, pero sí se devuelve un código distinto cuando se oprime una tecla ALT al mismo tiempo que ESC. Los valores compilados en las diferentes columnas se determinan mediante macros definidas en `include/minix/keymap.h`:

```
#define C(c)          ((c) & 0x1 F)          /* Mapear a código de control */
#define A(c)          ((c) | 0x80)          /* Encender bit ocho (ALT) */
#define CA(c)         A(C(c))              /* CTRL-ALT */
#define L(c)          ((c) | HASCAPS)      /* Agregar atributo de "Bloq Mayús tiene efecto"
atribute*/
```

Las primeras tres de estas macros manipulan bits en el código del carácter citado para producir el código necesario que se devolverá a la aplicación. La última pone en 1 el bit HASCAPS en el byte alto del valor de 16 bits. Éste es una bandera que indica que hay que verificar el estado de la tecla de bloqueo de mayúsculas (Caps Lock) y tal vez modificar el código antes de devolverlo. En la figura, las entradas para los códigos de detección 2, 13 y 16 muestran la forma como se manejan

las teclas numéricas, de puntuación y alfabéticas típicas. En el caso del código 28 se observa una característica especial: normalmente la tecla ENTER produce un código CR (0x0D), representado aquí por C('M'). Dado que el carácter de nueva línea en los archivos UNIX es el código LF (0x0A), y a veces es necesario introducir éste directamente, este mapa del teclado contempla una combinación CTRL-ENTER, que produce este código, C('J').

El código de detección 29 es uno de los códigos modificadores y se le debe reconocer sin importar cuál otra tecla esté oprimida, así que se devuelve el valor CTRL aunque esté presionada alguna otra tecla. Las teclas de función no devuelven valores ASCII ordinarios, y la fila para el código de detección 59 muestra simbólicamente los valores (definidos en `include/minix/keymap.h`) que se devuelven para la tecla F1 en combinación con otros modificadores. Estos valores son F1: 0x0110, SF1: 0x1010, AF1, 0x0810, ASF1: 0x0C10 y CF1: 0x0210. La última entrada que se muestra en la figura, para el código de detección 127, es representativa de muchas entradas cerca del final del arreglo. En muchos teclados, y ciertamente en aquellos que se usan en Europa y América, no hay suficientes teclas para generar todos los códigos posibles, y estas entradas de la tabla se llenan con ceros.

Tipos de letra cargables

Las primeras PC tenían los valores para generar caracteres en una pantalla de video almacenados sólo en ROM, pero las pantallas que se usan en los sistemas modernos cuentan con RAM en los adaptadores de video en la que se pueden cargar patrones personalizados de generación de caracteres. Esto se maneja en MINIX con una operación

```
ioctl(0, TIOCSFON, font)
```

de IOCTL. MINIX maneja un modo de video de 80 columnas x 25 filas, y los archivos de tipos de letra contienen 4096 bytes. Cada byte representa una línea de 8 pixeles que se iluminan si el valor del bit correspondiente es 1, y se necesitan 16 de esas líneas para mapear cada carácter. Sin embargo, el adaptador de video emplea 32 bytes para mapear cada carácter, a fin de ofrecer una definición más alta en modos que MINIX todavía no reconoce. Se incluye el comando `loadfont` para convertir estos archivos a la estructura `font` de 8192 bytes a la que hace referencia la llamada `[OCTL` y para usar dicha llamada con el fin de cargar el tipo de letra. Al igual que con los mapas de teclas, se puede cargar un tipo de letra durante el arranque, o en cualquier otro momento durante el funcionamiento normal. Por otro lado, cada adaptador de video tiene un tipo de letra incorporado en su ROM que está disponible por omisión. No hay necesidad de compilar un tipo de letra en MINIX mismo, y el único apoyo de tipo de letra necesario en el kernel es el código para llevar a cabo la operación `IOCTL TIOCSFON`.

3.9.4 Implementación del controlador de terminales independiente del dispositivo

En esta sección comenzaremos a examinar el código fuente del controlador de terminal con detalle. Cuando estudiamos los dispositivos por bloques vimos que varias tareas que apoyan varios dispositivos distintos pueden compartir una base de software común. El caso de los dispositivos

de terminal es similar, pero con la diferencia de que hay una tarea de terminal que apoya varios tipos distintos de dispositivos terminales. Aquí comenzaremos con el código independiente del dispositivo. En secciones posteriores examinaremos el código dependiente del dispositivo para el teclado y la pantalla de la consola con mapa en la memoria.

Estructuras de datos de la tarea de terminal

El archivo `tty.h` contiene definiciones utilizadas por los archivos en C que implementan los controladores de terminal. La mayor parte de las variables declaradas en este archivo se identifican con el prefijo `tty_`. También se declara una de esas variables en `glo.h` como `EXTERN`. Se trata de `tty_timeout`, utilizada por los manejadores de interrupciones tanto del reloj como de la terminal.

Dentro de `tty.h`, las definiciones de las banderas `O_NOCTTY` y `O_NONBLOCK` (que son argumentos opcionales de la llamada `OPEN`) son duplicados de definiciones contenidas en `include/fcntl.h`, pero se repiten aquí para no tener que incluir otro archivo. Los tipos `devfun_t` y `devfunarg_t` (líneas 11611 y 11612) sirven para definir apuntadores a funciones, a fin de poder manejar llamadas indirectas usando un mecanismo similar al que vimos en el código para el ciclo principal de los controladores de disco.

La más importante definición contenida en `tty.h` es la estructura `tty` (líneas 11614 a 11668). Hay una de estas estructuras para cada dispositivo terminal (la pantalla de la consola y el teclado juntos cuentan como una sola terminal). La primera variable de la estructura `tty`, `tty_events`, es la bandera que se enciende cuando una interrupción causa un cambio que requiere que la tarea de la terminal atienda el dispositivo. Cuando se enciende esta bandera, también se manipula la variable global `tty_timeout` para indicarle al manejador de interrupciones del reloj que debe despertar a la tarea de la terminal en el siguiente tic del reloj.

El resto de la estructura `tty` se organiza de modo tal que se agrupan las variables que se ocupan de entradas, salidas, estado e información referente a operaciones incompletas. En la sección de entrada, `tty_inhead` y `tty_intail` definen la cola que se usa como buffer para los caracteres recibidos. `Tty_incount` cuenta el número de caracteres contenidos en esta cola, y `fty_eotct` cuenta líneas o caracteres, como se explica más adelante. Todas las llamadas específicas para un dispositivo se realizan indirectamente, con excepción de las rutinas que inicializan las terminales, que se invocan para establecer los apuntadores empleados en las llamadas indirectas. Los campos `tty_devread` y `tty_icancel` contienen apuntadores a código específico para cada dispositivo que realizan las operaciones de lectura y de cancelación de entradas. `Tty_min` se usa en comparaciones con `tty_eotct`. Cuando esta última se vuelve igual a la primera, es que se completó una operación de entrada. Durante las entradas canónicas, `tty_min` se hace igual a 1 y `tty_eotct` cuenta las líneas introducidas. Durante la entrada no canónica, `tty_eotct` cuenta caracteres y a `tty_min` se asigna el contenido del campo `MIN` de la estructura `termios`. Así, la comparación de las dos variables indica cuándo está lista una línea o cuándo se llega a la cuenta mínima de caracteres, dependiendo del modo.

`Tty_time` contiene el valor de temporizador que determina cuándo el manejador de interrupciones del reloj debe despertar a la tarea de la terminal, y `tty_timenext` es un apuntador empleado para encadenar los campos `tty_time` activos formando una lista enlazada. La lista se ordena cada

vez que se establece un temporizado!", así que el manejador de interrupciones sólo tiene que ver la primera entrada. MINIX puede apoyar muchas terminales remotas, de las cuales sólo unas cuantas podrían tener temporizadores establecidos en un momento dado. La lista de temporizadores activos hace que el trabajo del manejador del reloj sea más fácil de lo que sería si tuviera que examinar cada entrada de `tty_table`.

Dado que la puesta en cola de las salidas corre por cuenta del código específico para cada dispositivo, la sección de salida de `tty` no declara variables y consiste exclusivamente en apuntadores a funciones específicas para cada dispositivo que escriben, hacen eco, envían una señal de paro y cancelan salidas. En la sección de estado, las banderas `tty_reprint`, `tty_escaped` y `tty_inhibited` indican que el último carácter detectado tiene un significado especial; por ejemplo, cuando se detecta un carácter CTRL-V (LNEXT), se asigna 1 a `tty_escaped` para indicar que se debe hacer caso omiso de cualquier significado especial que pudiera tener el siguiente carácter.

La siguiente parte de la estructura contiene datos relativos a las operaciones `DEV_READ`, `DEV_WRITE` y `DEV_IOCTL` que se están efectuando. Dos procesos participan en cada una de estas operaciones. El servidor que administra la llamada al sistema (normalmente FS), se identifica en `tty_incaller` (línea 11644). El servidor invoca la tarea de terminal a nombre de otro proceso que necesita realizar una operación de E/S, y este cliente se identifica en `tty_inproc` (línea 11645). Tal como se describió en la Fig. 3-37, durante un `READ`, los caracteres se transfieren directamente de la tarea de terminal a un buffer dentro del espacio de memoria del invocador original. `TtyInproc` y `tty_in_vir` ubican este buffer. Las dos variables siguientes, `tty_inleft` y `tty_incum`, cuentan los caracteres que todavía se necesitan y los que ya se transfirieron. Se requieren conjuntos similares de variables para una llamada al sistema `WRITE`. En el caso de `IOCTL`, puede haber una transferencia inmediata de datos entre el proceso solicitante y la tarea, por lo que se necesita una dirección virtual, pero no son necesarias variables para marcar el avance de una operación. Una petición `IOCTL` podría posponerse, por ejemplo, hasta que se completen las salidas en curso, pero en el momento oportuno la petición se llevará a cabo en una sola operación. Por último, la estructura `tty` incluye algunas variables que no pertenecen a ninguna otra categoría, incluidos apuntadores a las funciones que manejan las operaciones `DEV_IOCTL` y `DEV_CLOSE` en el nivel de dispositivo, una estructura `termios` al estilo POSIX y una estructura `winsize` que proporciona apoyo a las pantallas orientadas a ventanas. La última parte de la estructura sirve para almacenar la cola de entrada misma en el arreglo `tty_inbuf`. Cabe señalar que éste es un arreglo de `ul6_t`, no de caracteres `char` de 8 bits. Aunque las aplicaciones y dispositivos usen códigos de 8 bits para los caracteres, el lenguaje C requiere que la función de entrada `getchar` trabaje con un tipo de datos más grande a fin de poder devolver un valor EOF simbólico además de los 256 valores de byte posibles.

La tabla `tty_table`, un arreglo de estructuras `tty`, se declara usando la macro `EXTERN` (línea 11670). Hay un elemento para cada terminal habilitada por las definiciones `NR_CONS`, `NR_RS_LINES` y `NR_PTYS` de `include/minix/config.h`. Para la configuración que estudiamos este libro, se habilitan dos consolas, pero se puede recompilar MINIX para agregar hasta dos líneas en serie y hasta 64 seudotermiales.

Hay otra definición `EXTERN` en `tty.h`. `Tty_timelist` (línea 11690) es un apuntador que temporizador usa para contener la cabeza de la lista enlazada de campos `ttyJtime`. El archivo cabecera `tty.h` se incluye en muchos archivos, y la memoria para `tty_table` y `lty_timelist` se así

durante la compilación de `table.c`, de la misma manera que las variables `EXTERN` que se definen en el archivo de cabecera `glo.h`.

Al final de `tty.h` se definen dos macros, `bufen` y `bufend`. Éstas se utilizan a menudo en el código de la tarea de terminal, que copia muchos datos de `y` en buffers.

El controlador de terminal independiente del dispositivo

La tarea de terminal principal y todas las funciones de apoyo independientes del dispositivo están en `tty.c`. Puesto que la tarea apoya muchos dispositivos distintos, se deben usar los números de dispositivo secundarios para distinguir cuál dispositivo se está apoyando en una llamada en particular, y dichos números se definen en las líneas 11760 a 11764. Después vienen varias definiciones de macros. Si un dispositivo no está inicializado, los apuntadores a las funciones específicas para ese dispositivo contienen ceros, colocados ahí por el compilador de C. Esto permite definir la macro `tty_active` (línea 11774) que devuelve `FALSE` si encuentra un apuntador nulo. Desde luego, no es posible acceder indirectamente al código de inicialización de un dispositivo si parte de su trabajo consiste en inicializar los apuntadores que hacen posible el acceso indirecto. Las líneas 11777 a 11783 contienen definiciones de macros condicionales que igualan las llamadas de inicialización para dispositivos RS-232 o de seudoterminal a una función nula cuando estos dispositivos no están configurados. En esta sección se puede inhabilitar `do_pty` de forma similar. Esto permite omitir por completo el código para estos dispositivos si no se necesita.

Puesto que hay un gran número de parámetros configurables para cada terminal, y puede haber muchas terminales en un sistema conectado en red, se declara una estructura `termios_defaults` y se inicializa con valores por omisión (todos los cuales se definen en `include/termios.h`) en las líneas 11803 a 11810. Esta estructura se copia en la entrada de `tty_table` para una terminal cada vez que es necesario inicializarla o reinicializarla. Los valores por omisión para los caracteres especiales se mostraron en la Fig. 3-33. En la Fig. 3-42 se muestran los valores por omisión para las diversas banderas. En la siguiente línea se declara de forma similar la estructura `winsize_defaults` y se deja que el compilador de C la inicialice con solamente ceros. Ésta es la acción por omisión apropiada; está diciendo "se desconoce el tamaño de la ventana, úsese `/etc/termcap`".

Campo	Valores por omisión
<code>c_iflag</code>	BRKINT ICRNL IXON IXANY
<code>c_oflag</code>	OPOST ONLCR
<code>c_cflag</code>	CREAD CS8 HUPCL
<code>c_lflag</code>	ISIG IEXTEN ICANON ECHO ECHOE

Figura 3-42. Valores por omisión de las banderas de termios.

El punto de entrada para la tarea de terminal es `tty_task` (línea 11817). Antes de entrar en el ciclo principal, se invoca `tty_init` para cada terminal configurada (en el ciclo de la línea 11826), y

luego se exhibe el mensaje de inicio de MINIX (líneas 11829 a 11831). Aunque el código fuente muestra una llamada a `printf`, cuando se compila este código está activa la macro que conviene las llamadas a la rutina de biblioteca `printf` en llamadas a `printk`. `Printk` usa una rutina llamada `putk` dentro del controlador de la consola, así que no interviene el sistema de archivos. Este mensaje sólo se envía a la pantalla de la consola primaria y no puede redirigirse.

El ciclo principal de las líneas 11833 a 11884 es, en principio, igual al ciclo principal cualquier tarea; recibe un mensaje, ejecuta un `switch` con el tipo de mensaje para invocar la función apropiada, y luego genera un mensaje de retomo. Sin embargo, hay algunas complicación. Primero, una buena parte del trabajo es realizada por rutinas de interrupción de bajo nivel, sobre todo al manejar las entradas de la terminal. En la sección anterior vimos que se aceptan caracteres individuales del teclado y se colocan en un buffer sin enviar un mensaje a la tarea de terminal por cada carácter. Por tanto, antes de intentar recibir un mensaje, el ciclo principal siempre recorre toda la `tty_table`, inspeccionando la bandera `tp->tty_events` de cada terminal e invocando `handle_events` si es necesario (líneas 11835 a 11837) para ocuparse de todos los asuntos inconclusos. Sólo si no hay nada que exija atención inmediata se efectúa una llamada a `RECEIVE`. Si el mensaje recibido proviene del hardware, un enunciado `continue` permite reiniciar de inmediato el ciclo y volver a comprobar si han ocurrido eventos.

Segundo, esta tarea da servicio a varios dispositivos. Si un mensaje recibido proviene de una interrupción de hardware, se identifica el dispositivo o dispositivos que necesitan servicio examinando las banderas `tp->tty_events`. Si la interrupción no es de hardware, se usa el campo `TTY_LINE` del mensaje para determinar cuál dispositivo debe responder al mensaje. El número de dispositivo secundario se descodifica mediante una serie de comparaciones que permiten hacer que `tp` apunte a la entrada correcta de `tty_table` (líneas 11845 a 11864). Si el dispositivo es una seudoterminal, se invoca `do_pty` (`empty.c`) y se reinicia el ciclo principal. En este caso `do_pty` genera el mensaje de respuesta. Desde luego, si no están habilitadas las seudoterminales, la llamada a `do_pty` usa la macro ficticia que se definió antes. Es de esperar que no ocurrirán intentos de acceder a dispositivos inexistentes, pero siempre es más fácil agregar otra verificación que comprobar que no haya errores en alguna otra parte del sistema. Si el dispositivo no existe o no está configurado, se genera un mensaje de respuesta con el mensaje de error `ENXIO` y, una vez más, el control regresa al inicio del ciclo.

El resto de la tarea se parece a lo que hemos visto en el ciclo principal de otras tareas, un `switch` con base en el tipo de mensaje (líneas 11874 a 11883). Se invoca la función apropiada para el tipo de petición, `do_read`, `do_write`, etc. En cada caso la función invocada genera el mensaje de respuesta, en lugar de devolver al ciclo principal la información necesaria para construir el mensaje. Sólo se genera un mensaje de respuesta al final del ciclo principal si no se recibió un tipo de mensaje válido. En tal caso se envía un mensaje de error `EINVAL`. En vista de que se envían mensajes de respuesta desde muchos lugares distintos dentro de la tarea de terminal, se invoca una rutina común, `tty_reply`, para que se encargue de los detalles de la construcción de los mensajes de respuesta.

Si el mensaje recibido por `tty_task` es de un tipo válido, no el resultado de una interrupción, y no proviene de una seudoterminal, el `switch` al final del ciclo principal invoca una de las funciones `do_read`, `do_write`, `doJ.ocñ`, `do_open`, `do_close` o `do_cancel`. Los argumentos de todas estas lla-

madras son `tp`, un apuntador a una estructura `tty` y la dirección del mensaje. Antes de examinar cada función, mencionaremos unas cuantas consideraciones generales. Dado que `tty_task` puede dar servicio a múltiples dispositivos de terminal, estas funciones deben regresar rápidamente para que el ciclo principal pueda continuar. Sin embargo, cabe la posibilidad de que `do_read`, `do_write` y `doJocti` no puedan completar inmediatamente todo el trabajo solicitado. A fin de que el sistema de archivos (FS) pueda atender otras llamadas, se requiere una respuesta inmediata. Si la petición no puede llevarse a cabo de inmediato, se devuelve el código `SUSPEND` en el campo de estado del mensaje de respuesta. Esto corresponde al mensaje marcado con (3) en la Fig. 3-37 y suspende el proceso que inició la llamada, al tiempo que desbloquea FS. Los mensajes correspondientes a (7) y (8) en la figura se enviarán posteriormente cuando la operación pueda completarse. Si la petición puede satisfacerse plenamente, o si ocurre un error, se devuelve la cuenta de bytes transferidos o bien el código de error en el campo de estado del mensaje de retomo al FS. En este caso el FS enviará un mensaje inmediatamente al proceso que efectuó la llamada original, para despertarlo.

La lectura de una terminal es fundamentalmente diferente de la lectura de un dispositivo de disco. El controlador en software del disco emite un comando al hardware del disco y tarde o temprano recibe datos, a menos que haya habido una falla mecánica o eléctrica. La computadora puede exhibir una indicación en la pantalla, pero no tiene manera de obligar a la persona que está sentada ante la terminal a que comience a teclear. De hecho, no hay garantía de que habrá una persona sentada ahí. A fin de efectuar el pronto regreso que se exige, `do_read` (línea 11891) comienza por almacenar información que permitirá completar la petición posteriormente, cuando lleguen las entradas, si es que llegan. Primero se deben verificar algunos errores. Es un error que el dispositivo todavía esté esperando entradas para satisfacer una petición anterior, o que los parámetros del mensaje no sean válidos (líneas 11901 a 11908). Si se pasan estas pruebas, la información referente a la petición se copia en los campos apropiados de la entrada `tp->tty_table` correspondiente al dispositivo en las líneas 11911 a 11915. El último paso, asignar a `tp->tty_inleft` el número de caracteres solicitado, es importante. Esta variable sirve para determinar si ya se satisfizo la petición de lectura. En modo canónico `tp->tty_inleft` se decrementa en uno por cada carácter devuelto hasta recibir un final de línea; momento en el cual la variable se pone en cero de inmediato. En modo no canónico el manejo es diferente, pero la variable también se pone en cero en el momento en que la llamada se satisface, sea por vencimiento de un tiempo de espera o por la recepción de cuando menos el número mínimo de bytes solicitados. Cuando `tp->tty_inleft` llega a cero, se envía un mensaje de respuesta. Como veremos, los mensajes de respuesta se pueden generar en varios lugares. A veces es necesario verificar si un proceso de lectura todavía espera una respuesta; un valor de `tp->tty_inleft` distinto de cero sirve como bandera para este fin.

En modo canónico un dispositivo terminal espera entradas hasta haber recibido el número de caracteres solicitado en la llamada o hasta llegar al final de una línea o del archivo. En la línea 11917 se prueba el bit `ICANON` de la estructura `termios` para ver si está vigente el modo canónico para la terminal. Si dicho bit no es 1, se examinan los valores `MIN` y `TIME` de `termios` para determinar qué acción debe emprenderse.

En la Fig. 3-35 vimos cómo interactúan `MIN` y `TIME` para determinar los diferentes comportamientos de una llamada de lectura. `TIME` se prueba en la línea 11918. Un valor de cero corres-

ponde a la columna de la izquierda de la Fig. 3-35, y en este caso no se requieren pruebas adicionales aquí. Si TIME es distinto de cero, se prueba MIN. Si MIN es cero, se invoca settimer para poner en marcha el temporizador que terminará la petición DEV_READ después de un retraso, incluso si no se han recibido bytes. Aquí se pone `tp->tty_min` en 1 para que la llamada termine de inmediato si se reciben uno o más bytes antes de vencerse el tiempo de espera. En este punto todavía no se ha verificado la posibilidad de que haya entradas, así que varios caracteres podrían estar esperando ya para satisfacer la petición. En tal caso se devolverán tantos caracteres como estén listos, hasta el número especificado en la llamada READ, tan pronto como se encuentren las entradas. Si tanto TIME como MIN son distintos de cero, el temporizador tendrá otro significado. En este caso el temporizador se usa como temporizador entre caracteres; se pone en marcha solo después de recibirse el primer carácter y se reinicia después de cada carácter subsecuente. `tp->tty_eotet` cuenta caracteres en el modo no canónico, y si es cero en la línea 11931 querra decir que todavía no se han recibido caracteres y que el temporizador entre bytes está inhibido. Se usan lock y unlock para proteger estas dos llamadas a settimer, a fin de impedir las interrupciones de reloj mientras se está ejecutando settimer.

De cualquier modo, en la línea 11941 se invoca `in_transfer` para transferir cualesquier bytes que estén ya en la cola de entrada directamente al proceso lector. A continuación hay una llamada a `handle_events`, que puede colocar más datos en la cola de entrada y que invoca otra vez a `in_transfer`. Esta aparente duplicación de llamadas requiere una explicación. Aunque hasta ahora nuestra descripción ha sido en términos de entradas del teclado, `do_read` está en la parte código independiente del dispositivo y también da servicio a entradas de terminales remotas conectadas mediante líneas en serie. Es posible que entradas previas hayan llenado el buffer de entrada RS-232 a tal punto que se han inhibido las entradas. La primera llamada a `in_transfer` no reinicia el flujo, pero la llamada a `handle_events` puede tener este efecto. El hecho de que a continuación cause una segunda llamada a `in_transfer` es sólo algo extra; lo importante es asegurarse de que la terminal remota pueda transmitir otra vez. Cualquiera de estas llamadas puede lograr que se satisfaga la petición y que se envíe el mensaje de respuesta al FS. Se usa `tp->tty_inleft` como bandera para determinar si ya se envió la respuesta; si todavía es distinta de cero en la línea 11944, `do_read` genera y envía ella misma el mensaje de respuesta. Esto se hace en las líneas 11949 a 11957. Si la petición original especificó una lectura no bloqueadora, se le indica al FS que devuelva un código de error EAGAIN al invocador original. Si la llamada es una lectura bloqueadora ordinaria, el FS recibirá un código SUSPEND que lo desbloqueará, pero indicándole que deje bloqueado al invocador original. En este caso se asigna el valor REVIVE al campo `tp->tty_inrecode` de la terminal. Cuando se satisfaga posteriormente la llamada READ, si es que se hace, este código se colocará en el mensaje de respuesta al FS para indicarle que el invocador original se puso a dormir y que se le debe revivir.

`Do_write` (línea 11964) es similar a `do_read`, pero más sencilla, porque hay menos opciones por las cuales preocuparse al manejar una llamada al sistema WRITE. Se realizan verificaciones similares a las que `do_read` efectúa, para comprobar que no se esté efectuando todavía una escritura previa y que los parámetros del mensaje sean válidos, y luego se copian los parámetros de la petición en la estructura `tty`. Luego se invoca `handle_events`, y se verifica `tp->tty_outleft` para ver si se realizó el trabajo (líneas 11991 y 11992). Si fue así, `handle_events` ya envió un mensaje de

respuesta y no falta nada que hacer. Si no, se genera un mensaje de respuesta cuyos parámetros dependen de si la llamada WRITE original se hizo en modo bloqueador o no.

Función de POSIX	Operación de POSIX	Tipo de IOCTL	Parámetro de IOCTL
tcdrain	(ninguna)	TCDRAIN	(ninguno)
tcflow	TCOOFF	TCFLOW	int=TCOOFF
tcflow	TCOON	TCFLOW	int=TCOON
tcflow	TCIOFF	TCFLOW	int=TCIOFF
tcflow	TCION	TCFLOW	int=TCION
tcflush	TCIFLUSH	TCFLSH	int=TCIFLUSH
tcflush	TCOFLUSH	TCFLSH	int=TCOFLUSH
tcflush	TCIOFLUSH	TCFLSH	int=TCIOFLUSH
tcgetattr	(ninguna)	TCGETS	termios
tcsetattr	TCSANOW	TCGETS	termios
tcsetattr	TCSADRAIN	TCSETSF	termios
tcsetattr	TCSAFLUSH	TCSETSF	termios
tcsendbreak	(ninguna)	TCSBRK	int=duration

Figura 3-43. Llamadas de POSIX y operaciones IOCTL.

La siguiente función, `do_ioctl` (línea 12012), es larga, pero no difícil de entender. El cuerpo de `do_ioctl` consiste en dos enunciados `switch`. El primero determina el tamaño del parámetro al que apunta el apuntador del mensaje de petición (líneas 12033 a 12064). Si el tamaño no es cero, se prueba la validez del parámetro. No es posible probar el contenido aquí, pero lo que sí puede probarse es si una estructura del tamaño requerido, comenzando en la dirección especificada, cabe en el segmento en el que se especificó que debía estar. El resto de la función es otro `switch` basado en el tipo de la operación IOCTL solicitada (líneas 12075 a 12161). Desafortunadamente, la decisión de apoyar las operaciones exigidas por POSIX con la llamada IOCTL implicó la necesidad de inventar nombres para las operaciones IOCTL que sugirieran, pero no duplicaran, los nombres requeridos por POSIX. En la Fig. 3-43 se muestra la relación entre los nombres de petición POSIX y los nombres empleados por la llamada IOCTL de MINIX. Una operación `TCGETS` da servicio a una llamada `tcgetattr` hecha por el usuario y simplemente devuelve una copia de la estructura `tp-->tty_termios` del dispositivo terminal. Los siguientes cuatro tipos de petición comparten código. Los tipos de petición `TCSETSW`, `TCSETSF` y `TCSETS` corresponden a llamadas del usuario a la función `tcsetattr` definida por POSIX, y todas realizan la acción básica de copiar una nueva estructura `termios` en la estructura `tty` de una terminal. El copiado se realiza de inmediato en el caso de las llamadas `TCSETS` y puede efectuarse para las llamadas `TCSETSW` y `TCSETSF` si se completaron las salidas, mediante una llamada a `phys_copy`, para obtener los datos del usuario,

seguida de una llamada a `setattr`, en las líneas 12098 y 12099. Si `tcsetattr` se invocó con un modificador que solicita posponer la acción hasta completarse las salidas actuales, los parámetros de la petición se colocan en la estructura `tty` de la terminal para ser procesados posteriormente si la prueba de `tp->tty_outleft` de la línea 12084 revela que no se han completado las salidas. `Tcdrain` suspende un programa hasta completarse las salidas y se traduce a una llamada `IOCTL` de tipo `TCDRAIN`. Si ya se completaron las salidas, la función no tiene más que hacer; si no, también debe dejar información en la estructura `tty`.

La función `tcflush` de POSIX desecha las entradas no leídas y/o los datos de salida no enviados, según su argumento, y la traducción a `IOCTL` es directa, pues consiste en una llamada a la función `tty_icancel` que da servicio a todas las terminales y/o a la función específica para el dispositivo a la que apunta `tp->tty_icancel` (líneas 12102 a 12109). `Tcflow` se traduce de forma igualmente directa a una llamada `IOCTL`. Para suspender o reiniciar las salidas, esta función coloca un valor `TRUE` o `FALSE` en `tp->tty_inhibited` y luego enciende la bandera `tp->tty_events`. Para suspender o reiniciar las salidas, `tcflow` envía el código `STOP` (normalmente `CTRL-S`) o `START` (`CTRL-Q`) apropiado a la terminal remota, usando la rutina de eco específica para el dispositivo a la que apunta `tp->tty_echo` (líneas 12120 a 12125).

Casi todas las demás operaciones de las que `do_ioctl` se encarga se efectúan con una línea de código, invocando la función apropiada. En los casos de las operaciones `KIOCSMAP` (cargar mapa de teclas) y `TIOCSFON` (cargar tipo de letra), se realiza una prueba para asegurarse de que el dispositivo realmente es una consola, ya que estas operaciones no aplican a otras terminales. Si se están usando terminales virtuales, el mismo mapa de teclas y tipo de letra aplican a todas las consolas; el hardware no ofrece una forma sencilla de modificar esto. Las operaciones de tamaño de ventana copian una estructura `winsize` entre el proceso de usuario y la tarea de terminal. No obstante, tome nota del comentario que está bajo el código para la operación `TIOCSWINSZ`. Cuando un proceso cambia su tamaño de ventana, en algunas versiones de UNIX se supone que el kernel enviará una señal `SIGWINCH` al grupo de procesos. El estándar POSIX no exige dicha señal, pero si usted piensa emplear esas estructuras considere la adición de código aquí para iniciar la señal.

Los últimos dos casos en `do_ioctl` apoyan las funciones `tcgetpgrp` y `tcsetpgrp` requeridas por POSIX. En estos casos no hay una acción asociada, y siempre se devuelve un error. Esto nada tiene de malo. Estas funciones apoyan el control de trabajos, la capacidad para suspender y reiniciar un proceso desde el teclado. POSIX no exige control de trabajos, y MINIX no lo apoya. No obstante, POSIX requiere dichas funciones, aunque no se apoye el control de trabajos, para asegurar la transportabilidad de los programas.

`Do_open` (línea 12171) realiza una acción básica sencilla: incrementa la variable `tp->tty_opencnt` para el dispositivo a fin de poder verificar que está abierto. Sin embargo, antes hay que realizar ciertas pruebas. POSIX especifica que, en el caso de las terminales ordinarias, el primer proceso que abre una terminal es el líder de sesión, y cuando un líder de sesión muere, se revoca el acceso a la terminal por parte de los demás procesos de su grupo. Los demonios deben tener la posibilidad de escribir mensajes de error, y si su salida de error no se redirige a un archivo, deberá enviarse a un dispositivo de exhibición que no pueda cerrarse. Éste es el propósito del dispositivo llamado `/dev/log` de MINIX. Físicamente, éste es el mismo dispositivo que `/dev/console`, pero se le direcciona

empleando un número de dispositivo secundario diferente y se le trata de otra manera. Se trata de un dispositivo sólo de escritura, así que `do_open` devuelve un error `EACCESS` si se intenta abrirlo para lectura (línea 12183). La otra prueba que `do_open` realiza es verificar la bandera `O_NOCTTY`. Si la bandera está en `O` y el dispositivo no es `/dev/log`, la terminal se convierte en la terminal controladora de un grupo de procesos. Esto se hace colocando el número de proceso del invocador en el campo `tp->tty_grp` de la entrada correspondiente de `tty_table`. Después de esto, se incrementa la variable `tp->tty_openct` y se envía el mensaje de respuesta.

Es posible abrir más de una vez un dispositivo terminal, y la siguiente función, `do_close` (línea 12198) no tiene otra cosa que hacer más que decrementar `tp->tty_openct`. La prueba de la línea 12204 frustra un intento por cerrar el dispositivo si éste es `/dev/log`. Si esta operación es la última de cierre, se cancelan las entradas invocando `tp->tty_icancel`. También se invocan las rutinas específicas para el dispositivo a las que apuntan `tp->tty_ocancel` y `tp->tty_close`. Luego se restablecen a sus valores predeterminados diversos campos de la estructura `tty` para el dispositivo y se envía el mensaje de respuesta.

El último manejador de un tipo de mensajes es `do_cancel` (línea 12220), el cual se invoca cuando se recibe una señal para un proceso que está bloqueado tratando de leer o escribir. Hay tres estados que deben verificarse:

1. El proceso puede haber estado leyendo cuando se le obligó a terminar.
2. El proceso puede haber estado escribiendo cuando se le obligó a terminar.
3. El proceso puede haber sido suspendido por `tcdrain` hasta completarse sus salidas.

Se prueba cada uno de estos casos y se invoca la rutina general `tp->tty_icancel` o bien la rutina específica para el dispositivo a la que apunta `tp->tty_ocancel`, según sea necesario. En el segundo caso la única acción requerida es poner en `O` la bandera `tp->tty_ioreq`, para indicar que ya se completó la operación `IOCTL`. Por último, se pone en `1` la bandera `tp->tty_events` y se envía un mensaje de respuesta.

Código de apoyo del controlador de terminal

Habiendo examinado las funciones de nivel superior invocadas en el ciclo principal de `tty_task`, ha llegado el momento de estudiar el código que las apoya. Comenzaremos con `handle_events` (línea 12256). Como ya se mencionó, en cada repetición del ciclo principal de la tarea de terminal se examina la bandera `tp->tty_events` para cada dispositivo terminal y se invoca `handle_events` si la bandera indica que una terminal dada requiere atención. `do_read` y `do_write` también invocan `handle_events`. Esta rutina debe trabajar con rapidez; pone en cero la bandera `tp->tty_events` y luego invoca rutinas específicas para cada dispositivo para leer y escribir, usando los apuntadores a las funciones `tp->tty_devread` y `tp->tty_devwrite` (líneas 12279 a 12282). Éstas se invocan `mcondicionalmente`, porque no hay forma de probar si fue una lectura o una escritura lo que

causó que se izara la bandera. Aquí se tomó una decisión de diseño, pues examinar dos banderas para cada dispositivo sería más costoso que hacer dos llamadas cada vez que un dispositivo está activo. Además, casi siempre que se recibe un carácter de una terminal se debe hacer eco de él, y en tal caso son necesarias ambas llamadas. Como se apuntó en la explicación del manejo de las llamadas `tsetattr` por `do_ioctl`, POSIX puede posponer las operaciones de control en los dispositivos hasta completarse las salidas actuales, así que el momento inmediatamente posterior a la invocación de la función `tty_devwrite` específica para el dispositivo es ideal para ocuparse de las operaciones de control de E/S. Esto se hace en la línea 12285, donde se invoca `dev_ioctl` si hay una petición de control pendiente.

Puesto que la bandera `tp->tty_events` es izada por interrupciones, y podría ser que llegaran caracteres en un flujo precipitado desde un dispositivo rápido, cabe la posibilidad de que, para cuando se hayan completado `dev_ioctl` y las llamadas a las rutinas de lectura y escritura específicas para el dispositivo, otra interrupción haya izado la bandera de nuevo. Es muy importante que las entradas se transfieran del buffer donde la rutina de interrupción las colocó inicialmente; por tanto, `handle_events` repite las llamadas a las rutinas específicas para el dispositivo en tanto se detecte que la bandera `tp->tty_events` está izada al final del ciclo (línea 12286). Cuando se detiene el flujo de entrada (también podría ser de salida, pero es más probable que sean entradas las que efectúen semejantes demandas repetidas), se invoca `in_transfer` para transferir caracteres de la cola de entrada al buffer dentro del proceso que solicitó una operación de lectura. `in_transfer` misma envía un mensaje de respuesta si la transferencia completa la petición, ya sea transfiriendo el número máximo de caracteres solicitados o llegando al final de una línea (en modo canónico). En este caso, `tp->tty_left` será cero en el regreso a `handle_events`. Aquí se efectúa una prueba adicional y se envía un mensaje de respuesta si el número de caracteres transferidos llegó al número mínimo solicitado. La prueba de `tp->tty_inleft` evita el envío de un mensaje repetido.

A continuación examinaremos `in_transfer` (línea 12303), que se encarga de transferir datos de la cola de entrada en el espacio de memoria de la tarea al buffer del proceso de usuario que solicitó las entradas. Sin embargo, no es posible realizar un copiado de bloque directo. La cola de entrada es un buffer circular y es preciso revisar los caracteres para determinar que no se llegó al fin del archivo o, si está vigente el modo canónico, que la transferencia sólo continúe hasta el final de una línea. Además, la cola de entrada contiene cantidades de 16 bits, pero el buffer del destinatario es un arreglo de caracteres de 8 bits. Por ello, se emplea un buffer local intermedio. Los caracteres se revisan uno por uno en el momento de colocarse en el buffer local y, cuando éste se llena o cuando se ha vaciado la cola de entrada, se invoca `phys_copy` para transferir el contenido del buffer local al buffer del proceso receptor (líneas 12319 a 12345).

Se usan tres variables de la estructura `tty`, `tp->tty_inleft`, `tp->tty_eotct` y `tp->tty_min`, para decidir si `in_transfer` tiene trabajo que hacer o no, y las dos primeras controlan su ciclo principal. Como se mencionó antes, inicialmente se asigna a `tp->tty_inleft` un valor igual al número de caracteres solicitados por una llamada `READ`. Normalmente, esta variable se decrementa en uno cada vez que se transfiere un carácter, pero puede disminuirse a cero abruptamente si se detecta, una condición que indique el final de las entradas. Cada vez que `tp->tty_inleft` llega a cero, se genera un mensaje de respuesta para el lector, así que la variable también cumple su función como bandera para indicar si se ha enviado o no un mensaje. Por tanto, en la prueba de la línea 12314,

detectar que `tp->tty_inleft` vale cero es suficiente razón para abortar la ejecución de `in_transfer` sin enviar una respuesta.

En la siguiente parte de la prueba se comparan `tp->tty_eotct` y `tp->tty_min`. En el modo canónico, ambas variables se refieren a líneas de entrada completas, y en modo no canónico se refieren a caracteres. `tp->tty_eotct` se incrementa cada vez que se coloca un "salto de línea" o un byte en la cola de entrada, y es decrementada por `in_transfer` cada vez que una línea o un byte se retira de la cola. Por tanto, esta variable cuenta el número de líneas o bytes que han sido recibidos por la tarea de la terminal pero que todavía no se han pasado a un lector. `tp->tty_min` indica el número mínimo de líneas (en modo canónico) o caracteres (en modo no canónico) que se deben transferir para completar una petición de lectura; su valor siempre es 1 en modo canónico y puede tener cualquier valor entre 0 y `MAX_INPUT` (255 en MINIX) inclusive en modo no canónico. La segunda mitad de la prueba de la línea 12314 hace que `in_transfer` regrese inmediatamente en modo canónico si todavía no se ha recibido una línea completa. La transferencia no se efectúa hasta completarse una línea a fin de poder modificar el contenido de la cola si, por ejemplo, el usuario teclea subsecuentemente un carácter ERASE o KILL antes de pulsar la tecla ENTER. En modo no canónico ocurre un retorno inmediato si todavía no está disponible el número mínimo de caracteres.

Unas cuantas líneas después, se utilizan `tp->tty_inleft` y `tp->tty_eotct` para controlar el ciclo principal de `in_transfer`. En modo canónico la transferencia continúa hasta que ya no quedan líneas completas en la cola. En modo no canónico `tp->tty_eotct` es la cuenta de caracteres pendientes. `tp->tty_min` controla el ingreso al ciclo, pero no interviene en la determinación de cuándo debe detenerse. Una vez que se entra en el ciclo, se transfieren ya sea todos los caracteres disponibles o el número de caracteres solicitados en la llamada original, lo que sea menor.

0	V	D	N	c	C	c	c	7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

V: `IN_ESC`, escapado por `LNEXT` (CTRL-V)

D: `IN_EOF`, fin de archivo (CTRL-D)

N: `IN_EOT`, salto de línea (NL y otros)

cccc: cuenta de caracteres de los que se hizo eco

7: Bit 7, puede ponerse en cero si `ISTRIP` está encendido

6-0: Bits 0-6, código ASCII

Figura 3-44. Los campos de un código de carácter tal como se coloca en la cola de entrada.

Los caracteres son cantidades de 16 bits en la cola de entrada. El código de carácter real que se transfiere al proceso de usuario está en los 8 bits bajos. En la Fig. 3-44 se muestra cómo se usan los bits altos. Tres sirven como banderas para indicar si el carácter ha escapado (mediante CTRL-V), si marca el fin del archivo o si representa uno de varios códigos que indican que ya está completa

una línea. Cuatro bits se usan para contar cuánto espacio de pantalla se usa cuando se hace eco del carácter. La prueba de la línea 12322 verifica si está encendido el bit INJEOF (D en la figura). Esta prueba se hace al principio del ciclo interior porque un fin de archivo (CTRL-D) no se transfiere al lector ni se cuenta en el conteo de caracteres. Al transferirse cada carácter, se aplica una máscara para poner en cero los ocho bits superiores, y sólo se transfiere al buffer local el valor ASCII contenido en los ocho bits inferiores (línea 12324).

Hay más de una forma de indicar el fin de las entradas, pero se espera que la rutina de entrada específica para el dispositivo determine si un carácter recibido es un salto de línea, CTRL-D u otro carácter de este tipo, y marcar todos esos caracteres. `in_transfer` sólo necesita probar si está presente esta marca, el bit `IN_EOT` (N en la Fig. 3-44), en la línea 12340. Si se detecta la marca, se decrementa `tp->tty_eotct`. En el modo no canónico todos los caracteres se cuentan de esta forma en el momento de colocarse en la cola de entrada, y también se marcan todos los caracteres con el bit `IN_EOT` en ese momento, de modo que `tp->tty_eotct` cuenta los caracteres que todavía no se retiran de la cola. La única diferencia en el funcionamiento del ciclo principal de `in_transfer` en los dos modos, canónico y no canónico, está en la línea 12343. Aquí se pone en cero `tp->tty_inleft` cuando se encuentra un carácter marcado como salto de línea, pero sólo si está vigente el modo canónico. Por tanto, cuando el control regresa al principio del ciclo, éste termina correctamente después de un salto de línea en modo canónico, pero en el modo no canónico se hace caso omiso de los saltos de línea.

Cuando el ciclo termina por lo regular se tiene un buffer local parcialmente lleno que es necesario transferir (líneas 12347 a 12353). Luego se envía un mensaje de respuesta si `tp->tty_inleft` llegó a cero. Esto siempre sucede en el modo canónico, pero si está vigente el modo no canónico y el número de caracteres transferidos es menor que el número solicitado, no se envía la respuesta. Esto puede parecer extraño si tenemos una memoria para los detalles suficientemente buena como para recordar que, siempre que hemos encontrado llamadas a `in_transfer` (en `do_read` y `handle_events`), el código que sigue a la llamada a `in_transfer` envía un mensaje de respuesta si `in_transfer` regresa después de haber transferido más que la cantidad especificada en `tp->tty_min`, lo que ciertamente será el caso aquí. La razón por la que no se envía incondicionalmente una respuesta desde `in_transfer` será evidente cuando expliquemos la siguiente función, que invoca `in_transfer` en circunstancias distintas.

Esa siguiente función es `in_process` (línea 12367), que se invoca desde el software específico para el dispositivo a fin de manejar las tareas de procesamiento comunes que deben efectuarse con todas las entradas. Sus parámetros son un apuntador a la estructura `tty` del dispositivo de origen, un apuntador al arreglo de caracteres de ocho bits que se van a procesar, y una cuenta. La cuenta se devuelve al invocador. `in_process` es una función larga, pero sus acciones no son complicadas. Lo que hace es agregar caracteres de 16 bits a la cola de entrada que posteriormente será procesada por `in_transfer`.

`in_transfer` efectúa varias categorías de tratamiento:

1. Los caracteres normales se agregan a la cola de entrada, extendidos a 16 bits.
2. Los caracteres que afectan el procesamiento posterior modifican banderas para indicar el efecto, pero no se colocan en la cola.

3. Los caracteres que controlan el eco causan una acción inmediata y no son colocados en la cola.
4. A los caracteres con significado especial se les agregan códigos como el bit EOT a su byte alto en el momento de colocarse en la cola de entrada.

Veamos primero una situación totalmente normal: un carácter ordinario, como "x" (código ASCII 0x78), tecleado a la mitad de una línea corta, sin secuencia de escape vigente, en una terminal configurada con las propiedades predeterminadas estándar de MINIX. Tal como se recibe del dispositivo de entrada, este carácter ocupa los bits del 0 al 7 de la Fig. 3-44. En la línea 12385 su bit más significativo, el bit 7, se pondría en cero si el bit ISTRIP estuviera encendido, pero la acción predeterminada en MINIX es no borrar ese bit, lo que permite introducir códigos completos de 8 bits. De todos modos, esto no afectaría a nuestro "x". La acción predeterminada de MINIX es permitir el procesamiento extendido de las entradas, así que la prueba del bit IEXTEN de `tp->tty_termios.c_lflag` (línea 12388) pasa, pero las siguientes pruebas fallan en las condiciones que hemos postulado: no hay escape de caracteres vigente (línea 12391), esta entrada no es ella misma el carácter de escape de caracteres (línea 12397), y esta entrada no es el carácter REPRINT (línea 12405).

Las pruebas de las siguientes líneas determinan que el carácter de entrada no es el carácter especial `_POSIX_VDISABLE`, y que tampoco es un CR ni un NL. Por fin, un resultado positivo: está vigente el modo canónico, que es lo predeterminado (línea 12424). Sin embargo, nuestro "x" no es el carácter ERASE, y tampoco es KILL, EOF (CTRL-D), NL ni EOL, así que a la altura de la línea 12457 todavía no le habrá pasado nada. En esta línea se determina que el bit 1XON está encendido, por omisión, lo que habilita el empleo de los caracteres STOP (CTRL-S) y START (CTRL-Q), pero en las pruebas subsecuentes para detectar estos caracteres el resultado es negativo. En la línea 12478 se observa que el bit ISIG, que habilita el empleo de los caracteres INTR y QUIT, está encendido por omisión, pero otra vez nuestro carácter no coincide con ninguno de ellos.

De hecho, la primera cosa interesante que podría ocurrirle a un carácter ordinario sucede en la línea 12491, donde se realiza una prueba para determinar si la cola de entrada y a está llena. Si tal fuera el caso, el carácter se desearía en este punto, pues está vigente el modo canónico, y el usuario no vería su eco en la pantalla. (El enunciado `continué` desecha el carácter, ya que hace que reinicie el ciclo exterior.) Sin embargo, dado que hemos postulado condiciones absolutamente normales para esta ilustración, vamos a suponer que el buffer todavía no está lleno. La siguiente prueba, para ver si se requiere procesamiento especial en modo no canónico (línea 12497), falla, usando un salto hacia adelante hasta la línea 12512. Aquí se invoca `echo` para exhibir el carácter en la pantalla, pues el bit ECHO de `tp->tty_termios.c_lflag` está encendido por omisión.

Por último, en las líneas 12515 a 12519, se dispone del carácter colocándolo en la cola de entrada. En este momento se incrementa `tp->tty_incount`, pero como se trata de un carácter ordinario que no está marcado con el bit EOT, no se modifica `tp->tty_eotct`.

La última línea del ciclo invoca a `in_transfer` si el carácter que se acaba de transferir a la cola la llena. Sin embargo, en las condiciones ordinarias que hemos postulado para este ejemplo, `in_transfer` no haría nada, incluso si se invocara, porque (suponiendo que se dio servicio normal-

mente a la cola y que las entradas anteriores fueron aceptadas cuando se completó la línea de entrada previa) `tp->tty_eotct` es cero, `tp->tty_min` es uno y la prueba al principio de `in_Transf`. (línea 12314) causa un retomo inmediato.

Habiendo pasado por `in_process` con un carácter ordinario en condiciones ordinarias, regresemos al principio de esta función y veamos qué sucede en circunstancias menos ordinarias. Primero examinaremos el escape de caracteres, que permite pasar directamente al proceso de usuario un carácter que normalmente tendría un efecto especial. Si está vigente un escape de caracteres, está encendida la bandera `tp->tty_escaped`, cuando se detecta esto en la línea 12391, la bandera se apaga inmediatamente y se agrega el bit `IN_ESC` (bit V en la Fig. 3-44) al carácter actual. Esto da pie a un procesamiento especial cuando se hace eco del carácter: los caracteres de control escapados se exhiben como "" más el carácter visible correspondiente. El bit `IN_ESC` también evita que el carácter sea reconocido por las pruebas que detectan caracteres especiales. Las siguientes líneas procesan el carácter de escape mismo, el carácter `LNEXT` (`CTRL-V` por omisión). Cuando se detecta el código `LNEXT`, se enciende la bandera `tp->tty_escaped` y se invoca `rawecho` dos veces para exhibir un "" seguido de un retroceso. Esto recuerda al usuario del teclado que está vigente `n` escape, y cuando se hace eco del siguiente carácter éste sobrescribe el "" carácter `LNEXT` es un ejemplo de carácter que afecta caracteres posteriores (en este caso, sólo el carácter que sigue inmediatamente); no se coloca en la cola, y el ciclo se reinicia después de las dos llamadas a `rawecho`. El orden de estas dos pruebas es importante, pues permite teclear el carácter `LNEXT` dos veces seguidas, a fin de pasar la segunda copia a un proceso.

El siguiente carácter especial que `in_process` procesa es el carácter `REPRINT` (`CTRL-R`). Cuando se encuentra este carácter, se invoca `reprint` (línea 12406), y se vuelven a exhibir las salidas de las que se ha hecho eco actualmente. Después se desecha el `REPRINT` mismo sin afectar la cola de entrada.

Sería tedioso describir con detalle el manejo de todos y cada uno de los caracteres especiales y el código fuente de `in_process` es sencillo. Sólo mencionaremos unos cuantos puntos más. Un es que el empleo de bits especiales en el byte alto del valor de 16 bits que se coloca en la cola de entrada facilita la identificación de una clase de caracteres que tienen efectos similares. Así, los caracteres que marcan el fin de una línea, `EOT` (`CTRL-D`), `LF` y el carácter alternativo `EOL` (que por omisión no está definido), se marcan con el bit `EOT` (bit D en la Fig. 3-44) (líneas 12447 a 12453) para facilitar su reconocimiento posterior. Por último, justificaremos el comportamiento peculiar de `in_transfer` que señalamos antes. No se genera una respuesta cada vez que esta función termina, aunque en las llamadas a `in_transfer` que hemos visto anteriormente parecía que siempre se generaría una respuesta al regresar. Recuerde que la llamada a `in_transfer` efectuada por `in_process` cuando la cola de entrada está llena (línea 12522) no tiene efecto cuando este vigente el modo canónico. Sin embargo, si se desea procesamiento no canónico, cada carácter marca con el bit `EOT` en la línea 12499, lo que hace que `tp->tty_eotct` lo cuente en la línea 12519. A su vez, esto hace que se ingrese en el ciclo principal de `in_transfer` cuando se invoca porque la cola de entrada está llena en modo no canónico. En tales ocasiones no debe enviarse ningún mensaje al terminar `in_transfer`, porque es más probable que se lean más caracteres después de regresar a `in_process`. De hecho, aunque en modo canónico las entradas de un solo `READ` están limitadas por el tamaño de la cola de entrada (255 caracteres en `MINIX`), en modo no canónico una

llamada READ debe poder entregar el número de caracteres requerido por `mSK,_POSIX_SSIZE_MAX`, cuyo valor en MINIX es 32767.

Las siguientes funciones de `tty.c` apoyan la entrada de caracteres. Echo (línea 12531) trata unos cuantos caracteres de forma especial, pero en general simplemente los exhibe en el lado de salida del mismo dispositivo que se está usando para las entradas. Es posible que salidas de un proceso estén siendo enviadas a un dispositivo al mismo tiempo que se está haciendo eco de las entradas, y el resultado podría ser confuso si el usuario del teclado pulsa la tecla de retroceso. Para resolver este problema, las rutinas de salida específicas para el dispositivo siempre asignan TRUE a la bandera `tp->tty_reprint` cuando se producen salidas normales, para que la función invocada para manejar el retroceso sepa que se están produciendo salidas mezcladas. Puesto que echo también usa las rutinas de salida específicas para el dispositivo, el valor actual de `tp->tty_reprint` se conserva mientras se hace eco usando la variable local `rp` (líneas 12552 a 12585). Sin embargo, si acaba de iniciarse una nueva línea de entrada, se asigna FALSE a `rp` en lugar de asumir el valor antiguo, a fin de asegurar que `tp->tty_reprint` se pondrá en cero cuando echo termine.

Tal vez haya usted notado que echo devuelve un valor, por ejemplo, en la llamada de la línea 12512 de `in_process`:

```
ch = echo(tp, ch)
```

El valor devuelto por echo contiene el número de espacios que se usarán en la pantalla para exhibir el eco, que puede ser hasta ocho si el carácter es TAB. Esta cuenta se coloca en el campo `cccc` de la Fig. 3-44. Los caracteres ordinarios ocupan un espacio en la pantalla, pero si se hace eco de un carácter de control (excepto TAB, NL o CR o un DEL (0x7F), se exhibe como "" más un carácter ASCII imprimible y ocupa dos posiciones en la pantalla. Por otro lado, un NL o un CR ocupa cero espacios. Desde luego, el eco en sí debe ser efectuado por una rutina específica para el | dispositivo, y cada vez que se debe pasar un carácter al dispositivo se realiza una llamada indirecta utilizando `tp->tty_echo`, como ocurre en la línea 12580 para los caracteres ordinarios.

La siguiente función, `rawecho`, sirve para pasar por alto el manejo especial que efectúa echo. `rawecho` verifica si la bandera ECHO está encendida y, si lo está, envía el carácter a la rutina `fp->tty_echo` específica para el dispositivo sin procesamiento especial alguno. Aquí se usa una variable local, `rp`, para evitar que cuando `rawecho` invoque la rutina de salida se modifique el valor de `tp->tty_reprint`.

Cuando `in_process` encuentra un retroceso, invoca la siguiente función, `backover` (línea 12607). Ésta manipula la cola de entrada para eliminar el carácter que estaba en su cabeza, si esto es posible; si la cola está vacía o si el último carácter es un salto de línea no es posible retroceder. Aquí se prueba la bandera `tp->tty_reprint` que mencionamos al describir echo y `rawecho`. Si la bandera es TRUE, se invoca `reprint` (línea 12618) para colocar en la pantalla una copia limpia de la línea de salida. Luego se consulta el campo `len` del último carácter exhibido (el campo `cccc` en la P pg, 3-44) para averiguar cuántos caracteres será necesario borrar en la pantalla, y para cada caracter se envía una secuencia de caracteres retroceso-espacio-retroceso a través de `rawecho` para eliminar de la pantalla el carácter no deseado.

Reprint es la siguiente función. Además de ser invocada por backover, reprint puede ser invocada por el usuario oprimiendo la tecla REPRINT (CTRL-R). El ciclo de las líneas 12651 a 12656 recorre hacia atrás la cola de entrada en busca del último salto de línea. Si dicho salto se encuentra en la última posición que se llenó, no hay nada que hacer y reprint regresa. En caso contrario reprint hace eco del CTRL-R, que aparece en la pantalla como la secuencia de dos caracteres "R", y luego avanza a la siguiente línea y vuelve a exhibir la cola de entrada desde el último salto de línea hasta el final.

Hemos llegado ya a `out_process` (línea 12677). Al igual que `in_process`, esta función es invocada por las rutinas de salida específicas para el dispositivo, pero es más sencilla. `out_process` es invocada por las rutinas de salida específicas para dispositivos RS-232 y pseudoterminals, pero no por la rutina de la consola. Esta función opera sobre un buffer circular de bytes pero no retira los bytes del buffer. El único cambio que `out_process` hace al arreglo consiste en insertar un carácter CR adelante de un carácter NL en el buffer si están encendidos los bits OPOST (habilitar procesamiento de salida) y ONLCR (sustituir NL por CR-NL) de `tp->tty_termios.oflag`. Ambos bits están encendidos por omisión en MINIX. La misión de `out_process` es mantener actualizada la variable `tp->tty_position` de la estructura `tty` del dispositivo. Las tabulaciones y retrocesos complican la vida.

La siguiente rutina es `dev_ioctl` (línea 12763), que apoya a `do_ioctl` en la ejecución de las funciones `tcdrain` y `tcsetattr` cuando se le invoca con las opciones `TCSADRAIN` o `TCSAFLUSH`. En estos casos, `do_ioctl` no puede completar la acción inmediatamente si no se han completado las salidas, así que se guarda información referente a la petición en las partes de la estructura `tty` reservadas para las operaciones IOCTL diferidas. Cada vez que `handle_events` se ejecuta, examina el campo `tp->tty_ioreq` después de invocar la rutina de salida específica para el dispositivo, e invoca `dev_ioctl` si hay una operación pendiente. `DevJioctl` prueba `tp->tty_outleft` para ver si ya se completaron las salidas y, de ser así, realiza las mismas acciones que `do_ioctl` habría realizado inmediatamente si no hubiera habido retraso. Para dar servicio a `tcdrain`, lo único que hay que hacer es poner en cero el campo `tp->tty_ioreq` y enviar el mensaje de respuesta al ES, pidiéndole que despierte el proceso que efectuó la llamada original. La variante `TCSAFLUSH` de `tcsetattr` invoca `tty_icancl` para cancelar las entradas. En ambas variantes de `tcsetattr`, la estructura `termios` cuya dirección se pasó en la llamada original a IOCTL se copia en la estructura `tp->tty_termios` del dispositivo. A continuación se invoca `setattr` después, tal como se hizo con `tcdrain`, se envía mi mensaje de respuesta para despertar al invocador original bloqueado.

`setattr` (línea 12789) es el siguiente procedimiento. Como hemos visto, `setattr` es invocado por `do_ioctl` o `dev_ioctl` para modificar los atributos de un dispositivo terminal, y por `do_close` para restablecer los atributos a sus valores predeterminados. `setattr` siempre se invoca después de copiar una nueva estructura `termios` en la estructura `tty` de un dispositivo, porque no basta con copiar sólo los parámetros. Si el dispositivo que se está controlando ahora está en modo no canónico, la primera acción consiste en marcar con el bit `IN_EOT` todos los caracteres que actualmente están en la cola de entrada, como se habría hecho al colocar originalmente dichos caracteres en la cola si entonces hubiera estado vigente el modo no canónico. Es más fácil hacer esto (en las líneas 12803 a 12809) que probar si los caracteres ya tienen encendido ese bit. No hay forma de saber cuáles atributos acaban de ser modificados y cuáles conservan sus valores antiguos.

La siguiente acción es verificar los valores de MIN y TIME. En modo canónico, `tp->tty_min` siempre es 1, pues se enciende en la línea 12818. En modo no canónico la combinación de los dos valores permite cuatro modos de operación distintos, como vimos en la Fig. 3-35. En las líneas 12823 a 12825, primero se asigna a `tp->tty_min` el valor que se pasó en `tp->tty_termios.cc[VMIN]`, el cual entonces se modifica si es cero y `tp->tty_termios.cc[VTIME]` no es cero.

Por último, `setattr` se asegura de que las salidas no serán detenidas si está inhabilitado el control XON/XOFF, envía una señal SIGHUP si la velocidad de las salidas se pone en cero, y hace una llamada indirecta a la rutina específica para el dispositivo a la que apunta `tp->tty_ioctl` a fin de efectuar lo que sólo puede hacerse en el nivel de dispositivo.

La siguiente función, `tty_reply` (línea 12845) se ha mencionado varias veces en esta explicación. Su acción es sencilla: construir un mensaje y enviarlo. Si por alguna razón falla la respuesta, sobreviene un acceso de pánico. Las siguientes funciones son igualmente simples. `Sigchar` (línea 12866) le pide al administrador de memoria que envíe una señal. Si la bandera NOFLSH no está encendida, las entradas puestas en cola se eliminan: la cuenta de caracteres o líneas recibidas se pone en cero y los apuntadores a la cola y la cabeza de la cola se igualan. Ésta es la acción predeterminada. Si se debe atrapar una señal SIGHUP, se puede encender NOFLSH para permitir que las entradas y salidas se reanuden después de atraparse la señal. `Tty_icancel` (línea 12891) desecha incondicionalmente las salidas pendientes de la manera que describimos para `sigchar`, y además invoca la función específica para el dispositivo a la que apunta `tp->tty_icancel`, a fin de cancelar las entradas que puedan existir en el dispositivo mismo o estar en un buffer del código de bajo nivel.

`Tty_init` (línea 12905) se invoca una vez para cada dispositivo cuando se inicia por primera vez `tty_task`, y establece los valores predeterminados. Inicialmente se coloca en las variables `tp->tty_icancel`, `tp->tty_ocancel`, `tp->tty_ioctl` y `tp->tty_close` un apuntador a `tty_devnop`, una función ficticia que no hace nada. A continuación, `tty_init` invoca funciones de inicialización específicas para el dispositivo para la categoría apropiada de terminal (consola, línea serial o seudoterminal). Éstas establecen los apuntadores reales para las funciones específicas para el dispositivo que se invocan indirectamente. Recuerde que si no hay dispositivos configurados en una categoría en particular, se crea una macro que regresa inmediatamente, para que no se tenga que compilar ninguna parte del código para un dispositivo no configurado. La llamada a `scr_init` inicializa el controlador de la consola y también invoca la rutina de inicialización para el teclado.

`Tty_wakeup` (línea 12929), aunque corta, es extremadamente importante para el funcionamiento de la tarea de terminal. Cada vez que se ejecuta el manejador de interrupciones del reloj, es decir, en cada tic del reloj, se revisa la variable global `tty_timeout` (definida en `glo.h` en la línea 5032) para ver si contiene un valor menor que el tiempo actual. Si es así, se invoca `tty_wakeup`. Las rutinas de servicio de interrupciones para los controladores de terminal ponen en cero `\tty_fimeout`, lo que obliga a la ejecución de `wakeup` en el siguiente tic del reloj después de cualquier interrupción de dispositivo terminal. `Tty_timeout` también es alterada por `settimer` cuando un dispositivo terminal está atendiendo una llamada READ en modo no canónico y necesita establecer un tiempo de espera, como veremos en breve. Cuando `tty_wakeup` se ejecuta, primero inhabilita la siguiente acción de despertar asignando `TIME_NEVER`, un valor muy lejano en el futuro, a `tty_timeout`. Luego, `tty_wakeup` examina la lista enlazada de valores de temporizador, que está

ordenada con los primeros despertares programados más próximos, hasta llegar a uno que posterior al tiempo actual. Éste es el siguiente despertar, y se coloca en `tty_timeout`. `Tty_wake` también pone en cero `tp->tty_min` para ese dispositivo, lo que asegura que la siguiente lectura tendrá éxito incluso si no se han recibido bytes, enciende la bandera `tp->tty_events` para el dispositivo a fin de asegurar que reciba atención la siguiente vez que se ejecute la tarea de terminal y retira el dispositivo de la lista de temporizadores. Por último, `tty_wakeup` invoca `interrupt` para enviar el mensaje de despertar a la tarea. Como se mencionó en la explicación de la tarea de reloj `tty_wakeup` forma lógicamente parte del código de servicio de interrupciones del reloj, ya que sólo se invoca desde ahí.

La siguiente función, `settimer` (línea 12958), establece temporizadores para determinar cuándo debe regresarse de una llamada `READ` en modo no canónico; se ejecuta con parámetros de `tp` un apuntador a una estructura `tty`, y ora, un entero que representa `TRUE` o `FALSE`. Primero se revisa la lista enlazada de estructuras `tty` a la que `timelist` apunta, buscando una entrada existente que coincida con el parámetro `tp`. Si se encuentra una, se retira de la lista (líneas 12968 a 12973). Si se invoca `settimer` para eliminar un temporizador, esto es todo lo que debe hacer; si se invoca para establecer un temporizador, se asigna al elemento `tp->tty_time` de la estructura `tty` del dispositivo el tiempo actual más el incremento en décimas de segundo especificado por el valor `TIME` que está en la estructura `termios` del dispositivo. Luego se coloca la entrada en la lista, que se mantiene en orden creciente. Por último, el tiempo de espera recién incluido en la lista se compara con el valor de la variable global `tty_timeout`, y ésta se reemplaza si el nuevo tiempo de espera se va a vencer antes.

La última definición de `tty.c` es `tty_devnops` (línea 12992), una función de "no hay operación" que se direcciona indirectamente en los casos en que un dispositivo no necesita un servicio. Hemos visto que `tty_devnops` se usa en `tty_init` como valor predeterminado que se coloca varios apuntadores de función antes de invocar la rutina de inicialización para un dispositivo.

3.9.5 Implementación del controlador de teclado

Ahora pasamos al código dependiente del dispositivo que apoya la consola de MINIX, que consiste en un teclado IBM PC y una pantalla con mapa en la memoria. Los dispositivos físicos que apoyan a éstos son totalmente independientes: en un sistema de escritorio estándar la pantalla emplea una tarjeta adaptadora (de las cuales hay por lo menos media docena de tipos básico) insertada en el plano posterior, mientras que el teclado se apoya en circuitos incorporados en la tarjeta matriz que tienen una interfaz con una computadora de un solo chip de 8 bits dentro de la unidad del teclado. Los dos subdispositivos requieren apoyo de software totalmente independiente, que se encuentra en los archivos `keyboard.c` y `console.c`.

El sistema operativo ve el teclado y la consola como partes del mismo dispositivo, `/dev/consolé`. Si hay suficiente memoria disponible en el adaptador de la pantalla, puede compilarse el apoyo de consola virtual, y además de `/dev/console` podría haber dispositivos lógicos adicionales, `/dev/ttycl`, `/dev/ttyc2`, etc. Sólo las salidas de uno se envían a la pantalla en un momento dado y sólo hay un teclado para las entradas a la consola activa. Lógicamente, el teclado está supeditado a la consola, pero esto sólo se manifiesta en dos aspectos relativamente menores. Primero, `tty_table` contiene una estructura `tty` para la consola, y en los puntos en los que se proporcional

campos distintos para entrada y salida, por ejemplo, los campos `tty_devread` y `tty_devwrite`, se establecen apuntadores a funciones en `keyboard.c` y `console.c` en el momento del inicio. Sin embargo, sólo hay un campo `tty_priv`, y éste apunta sólo a las estructuras de datos de la consola. Segundo, antes de ingresar en su ciclo principal, `tty_task` invoca cada dispositivo lógico una vez para inicializarlo. La rutina invocada `psssi/dev/console` está en `console.c`, y el código de inicialización para el teclado se invoca desde ahí. No obstante, la jerarquía implícita bien podría haberse invertido. Siempre hemos examinado las entradas antes que las salidas al tratar con dispositivos de E/S y seguiremos con ese patrón, analizando `keyboard.c` en esta sección y dejando la explicación de `console.c` para la siguiente sección.

`Keyboard.c` comienza, al igual que la mayor parte de los archivos fuente que hemos visto, con varias instrucciones `#include`. Una de éstas, sin embargo, es inusual. El archivo `keymaps/us-std.src` (incluido en la línea 13104) no es una cabecera ordinaria; es un archivo fuente en C que da pie a la compilación del mapa de teclas por omisión dentro de `keyboard.o` como un arreglo inicializado. El archivo fuente del mapa de teclas no se incluye en los listados al final del libro por causa de su tamaño, pero algunas entradas representativas se ilustraron en la Fig. 3-41. Después de las instrucciones `#include` vienen macros que definen diversas constantes. Las del primer grupo se usan en la interacción de bajo nivel con el controlador en hardware del teclado. Muchas de éstas son direcciones de puertos de E/S o combinaciones de bits que tienen significado en estas interacciones. El siguiente grupo incluye nombres simbólicos para teclas especiales. La macro `kb_addr` (línea 13041) siempre devuelve un apuntador al primer elemento del arreglo `kb_lines`, puesto que el hardware de IBM sólo reconoce un teclado. En la siguiente línea se define simbólicamente el tamaño del buffer de entrada del teclado como `KB_IN_BYTES`, con un valor de 32. Las siguientes 11 variables sirven para contener diversos estados que se deben recordar a fin de interpretar correctamente una digitación. Estas variables se usan de diferentes maneras. Por ejemplo, el valor de la bandera `capslock` (línea 13046) se conmuta entre `TRUE` y `FALSE` cada vez que se pulsa la tecla Bloq Mayús (Caps Lock). La bandera `shift` (línea 13054) se pone en `TRUE` cuando se oprime la tecla `SHIFT` y en `FALSE` cuando se suelta dicha tecla. La variable `esc` se pone en 1 cuando se recibe un escape de código de detección, y siempre se pone en 0 después de recibirse el siguiente carácter.

La estructura `kb_s` de las líneas 13060 a 13065 sirve para seguir la pista a los códigos de detección conforme se introducen. Dentro de esta estructura, los códigos se mantienen en un buffer circular, en el arreglo `ibuf`, cuyo tamaño es `KB_IN_BYTES`. Se declara `kb_lines[NR_CONS]`, un arreglo de esas estructuras, una por cada consola, pero en realidad sólo se usa la primera, ya que siempre se utiliza la macro `kbaddr` para determinar la dirección de la `kb_s` vigente. Sin embargo, por lo regular nos referimos a las variables contenidas en `kb_lines[0]` empleando un apuntador a la estructura, por ejemplo, `kb->ihead`, para mantener la consistencia con la forma como tratamos otros dispositivos y para hacer que las referencias del texto sean congruentes con las contenidas en el listado del código fuente. Desde luego, se desperdicia una pequeña cantidad de memoria a causa de los elementos de arreglo no utilizados. Sin embargo, si alguien fabrica una PC con apoyo & hardware para múltiples teclados, MINIX está listo; basta con modificar la macro `kbaddr`.

`Map_key0` (línea 13084) se define como una macro; devuelve el código ASCII que corresponde a un código de detección, sin tener en cuenta los modificadores. Esto equivale a la primera

columna (sin SHIFT) del arreglo del mapa de teclas. Su hermano mayor es `map_key` (línea 13091), que realiza la transformación completa de un código de detección en un código ASCII incluida la consideración de las (múltiples) teclas modificadoras que estén oprimidas al mismo tiempo que las teclas ordinarias.

La rutina de servicio de interrupciones del teclado es `kbd_hw_int` (línea 13123), invocado cada vez que se oprime o suelta una tecla. Esta rutina invoca `scan_keyboard` para obtener el código de detección del chip controlador del teclado. El bit más significativo del código de detección se pone en 1 cuando la interrupción se debe a que se soltó una tecla, y en este caso se hace caso omiso de la tecla a menos que se trate de una de las teclas modificadoras. Si la interrupción se debe a que se oprimió una tecla, o a que se soltó una tecla modificadora, el código de detección si bruto se coloca en el buffer circular si hay espacio, se iza la bandera `tp->tty_events` para la consola actual (línea 13154) y luego se invoca `force_timeout` para asegurarse de que la tarea del reloj iniciará la tarea de la terminal en el siguiente tic del reloj. En la Fig. 3-45 se muestran los códigos de detección en el buffer para una línea corta de entrada que contiene dos caracteres en mayúsculas, cada uno precedido por el código de detección de oprimir una tecla SHIFT y seguido de código de liberación de dicha tecla.

42	35	170	18	38	38	24	57	54	17	182	24	19	38	32	28
L+	h	L-	e	l	L	o	SP	R+	W	R-	o	r	l	d	CR

Figura 3-45. Códigos de detección en el buffer de entrada, con las digitaciones correspondientes abajo, para una línea de texto introducida mediante el teclado. L+, L-, R+ y R- representan, respectivamente, oprimir y soltar las teclas SHIFT izquierda y derecha. El código de la liberación de una tecla es 128 más el código correspondiente a la pulsación de la misma tecla.

Cuando ocurre la interrupción de reloj, se ejecuta la tarea de terminal misma, y al encontrar encendida la bandera `tp->tty_events` para el dispositivo de la consola, invoca `kb_read` (línea 13165), la rutina específica para el dispositivo, empleando el apuntador contenido en el campo `tp->tty_devread` de la estructura `tty` de la consola. `Kb_read` toma códigos de detección del buffer circular del teclado y coloca códigos ASCII en su buffer local, que tiene el tamaño suficiente para contener las secuencias de escape que deben generarse como respuesta a ciertos códigos de detección del subteclado numérico. A continuación `kb_read` invoca `in_process` en el código independiente del dispositivo para colocar los caracteres en la cola de entrada. En las líneas 13181 a 13183 se usan `lock` y `unlock` para proteger el decremento de `kb->icount` contra la posible llegada simultánea de una interrupción del teclado. La llamada a `makeJbreak` devuelve el código ASCT como un entero. Las teclas especiales, como las del subteclado numérico y las de función, tienen valores mayores que `0xFF` en este punto. Los códigos en el intervalo de HOME a INSRT (`0x101` a `0x10C`, definidos en `include/minix/keymap.h`) son el resultado de oprimir las teclas del subteclado numérico, y se convierten en los caracteres de escape de tres caracteres que se muestran en la Fig. 3-46 usando el arreglo `numpad_map`. A continuación, las secuencias se pasan a `in_process` (líneas 13196 a 13201). Los códigos más altos no se pasan a `in_process`, sino que se intenta detectarlos.

códigos ALT-FLECHA-IZQUIERDA, ALT-FLECHA-DERECHA o ALT-F1 a ALT-F12, y si se encuentra uno de éstos se invoca `select_console` para intercambiar consolas virtuales.

Tecla	Código de Detección	"ASCII"	Secuencia de Escape
Inicio	71	0x101	ESC [H
Flecha Arriba	72	0x103	ESC [A
Adel. Pág	73	0x107	ESC [V
-	74	0x10A	ESC [S
Flecha Izq.	75	0x105	ESC [D
5	76	0x109	ESC [G
Flecha Der.	77	0x106	ESC [C
+	78	0x10B	ESC [T
Fin	79	0x102	ESC [Y
Flecha Abajo	80	0x104	ESC [B
Retr. Pág.	81	0x108	ESC [U
Ins	82	0x10C	ESC [@

Figura 3-46. Códigos de escape generados por el subteclado numérico. Cuando los códigos de detección de las teclas ordinarias se traducen a códigos ASCII, se asigna a las teclas especiales códigos "seudo-ASCII" con valores mayores que 0xFF.

`MakeJbreak` (línea 13222) convierte códigos de detección a ASCII y luego actualiza las variables que siguen la pista al estado de las teclas modificadoras. Antes, empero, `make_break` intenta detectar la combinación mágica CTRL-ALT-DEL que todos los usuarios de PC conocen como la forma de reiniciar bajo MS-DOS. Sin embargo, es deseable una clausura ordenada, así que, en lugar de tratar de iniciar las rutinas de PC BIOS se envía una señal SIGABRT a `init`, el proceso padre de todos los demás procesos. Se espera que `init` atraparé esta señal y la interpretará como un comando para iniciar un proceso ordenado de terminar operaciones, antes de causar un retorno al monitor de arranque, desde el cual se podrá ordenar un reinicio completo del sistema o un reinicio de MINIX. Desde luego, no es razonable esperar que esto funcione siempre. La mayor parte de los usuarios entiende los peligros de una clausura abrupta y no oprimen CTRL-ALT-DEL si no está ocurriendo algo en verdad terrible y se ha hecho imposible controlar normalmente el sistema. En este punto es factible que el sistema esté tan desorganizado que sea imposible enviar ordenadamente una señal a otro proceso. Ésta es la razón de incluir una variable static llamada `CAD_count` en `make_break`. La mayor parte de las caídas del sistema dejan el sistema de interrupciones funcionando, así que todavía es posible recibir entradas del teclado y la tarea del reloj puede mantener operando la tarea de terminal. Aquí MINIX aprovecha el comportamiento esperado

de los usuarios de computadora, que muchas veces se ponen a golpear las teclas repetidamente cuando sienten que algo no está funcionando correctamente. Si el intento de enviar la señal SIGABRT a `init` falla y el usuario oprime CTRL-ALT-DEL otras dos veces, se efectúa una llamada directa a `wreboot`, causando un retomo al monitor sin pasar por la llamada a `init`.

La parte principal de `makeJbreak` no es difícil de seguir. La variable `make` registra si el código de detección fue generado por la pulsación de una tecla o por su liberación, y entonces la llamada a `map_key` devuelve el código ASCII en `ch`. A continuación hay un `switch` con base en `ch` (líneas 13248 a 13294). Consideremos dos casos, una tecla ordinaria y una tecla especial. Para una tecla ordinaria, ninguno de los casos coincide, y no deberá suceder nada tampoco en el caso por omisión (línea 13292), ya que se supone que los códigos de las teclas ordinarias sólo se aceptan en la fase de pulsación de la pulsación y liberación de una tecla. Si por alguna razón un código de tecla ordinaria es aceptado al liberarse una tecla, se sustituirá aquí un valor de -1, y el invocado (`kb_read`) hará caso omiso de este valor. Las teclas especiales, como CTRL, se identifican en el punto apropiado del `switch`, en este caso en la línea 13249. La variable correspondiente, en este caso `control`, registra el estado de `make` y se sustituye -1 por el código de carácter que se devolverá (e ignorará). El manejo de las teclas ALT, CALOCK, NLOCK y SLOCK es más complicado, pero para todas estas teclas especiales el efecto es similar: una variable registra el estado actual (en el caso de las teclas que sólo surten efecto mientras están oprimidas) o bien conmuta el estado previo (en el caso de las teclas de cierre ("LOCK")).

Hay un caso más que considerar, el de la tecla EXTKEY y la variable `ese`. Ésta no debe confundirse con la tecla ESC del teclado, que devuelve el código ASCII OxIB. No hay forma de generar el código EXTKEY por sí solo oprimiendo cualquier tecla o combinación de teclas; se trata del prefijo de tecla extendida del teclado de PC, el primer byte de un código de detección de dos bytes que indica que se presionó una tecla que no formaba parte del teclado de la PC original pero que tiene el mismo código de detección. En muchos casos, el software trata las dos teclas de forma idéntica. Por ejemplo, esto casi siempre sucede con la tecla "/" normal y la tecla "/" gris del subteclado numérico. En otros casos, nos gustaría distinguir entre dichas teclas. Por ejemplo, muchas organizaciones de teclado para idiomas distintos del inglés tratan de forma distinta las teclas ALT izquierda y derecha, a fin de apoyar teclas que deben generar tres códigos de carácter distintos. Ambas teclas ALT generan el mismo código de detección (56), pero el código EXTKEY precede a éste cuando se oprime la tecla ALT derecha. Cuando se devuelve el código EXTKEY, se enciende la bandera `ese`. En este caso, `make_break` regresa de dentro del `switch`, pasando así por alto el último paso antes del retomo normal, que asigna cero a `ese` en todos los demás casos (línea 13295). Esto tiene el efecto de hacer a `ese` efectiva sólo para el siguiente código que se reciba. Si el lector está familiarizado con las complejidades del teclado de PC en su uso ordinario, esto le resultará conocido y a la vez un poco extraño, porque el PC BIOS no permite leer el código de detección de una tecla ALT y devuelve un valor diferente para el código extendido del que MINIX devuelve.

`Set_leds` (línea 13303) enciende y apaga las luces que indican si se han oprimido las teclas Bloq Núm (Num Lock), Bloq Mayús (Caps Lock) o Bloq Despi (Scroll Lock) de un teclado de PC. Se escribe un byte de control, `LED_CODE`, en un puerto de salida para indicar al teclado que el siguiente byte que se escriba en ese puerto será para el control de las luces, y el estado de las

tres luces se codifica en tres bits de ese siguiente byte. Las siguientes dos funciones apoyan esta operación. `Kb_wait` (línea 13327) se invoca para determinar que el teclado está listo para recibir una secuencia de comandos, y `kb_ack` (línea 13343) se invoca para verificar que se ha acusado recibo del comando. Ambos comandos utilizan espera activa, leyendo continuamente hasta que se detecta el código deseado. Ésta no es una técnica recomendable para manejar la mayor parte de las operaciones de E/S, pero el encendido y apagado de las luces del teclado no se efectúa con mucha frecuencia, así que no se desperdicia mucho tiempo si se hace de manera ineficiente. Advierta también que tanto `kb_wait` como `kb_ack` podrían fallar, y podemos determinar si sucedió esto examinando el código de retomo. Sin embargo, ajustar las luces del teclado no tiene la suficiente importancia como para justificar la verificación del valor devuelto por cualquiera de las dos llamadas, y `set_leds` se limita a proceder ciegamente.

Puesto que el teclado forma parte de la consola, su rutina de inicialización, `kb_init` (línea 13359) se invoca desde `scr_init` en `consol.c`, no directamente desde `tty_init` en `tty.c`. Si están habilitadas las consolas virtuales (es decir, `NR_CONS` en `include/minix/config.h` es mayor que 1), se invoca `kb_init` una vez para cada consola lógica. Después de la primera vez, la única parte de `kb_init` que es indispensable para las consolas adicionales es colocar la dirección de `kb_read` en `tp->tty_devread` (línea 13367), pero no hay problema si se repite el resto de la función. El resto de `kb_init` inicializa algunas variables, ajusta las luces del teclado e inspecciona el teclado para asegurarse de que no se leerá ninguna digitación remanente. Una vez que todo está listo, `kb_init` invoca `put_irq_handler` y luego `enable_irq`, de modo que se ejecute `kbd_hw_int` cada vez que se oprima o se suelte una tecla.

Las siguientes tres funciones son más bien sencillas. `Kbd_loadmap` (línea 13392) es casi trivial; es invocada por `do_ioctl` en `tty.c` para copiar un mapa de teclas del espacio de usuario sobrescribiendo el mapa de teclas por omisión compilado al incluirse un archivo fuente de mapa de teclas al principio de `keyboard.c`.

`Func_key` (línea 13405) se invoca desde `kb_read` para ver si se oprimió una tecla especial que se deba procesar localmente. En la Fig. 3-47 se resumen estas teclas y sus efectos. El código (invocado se encuentra en varios archivos). Los códigos F1 y F2 activan código en `dmp.c`, que estudiaremos en la siguiente sección. El código F3 activa `toggle_scroll`, que está en `consol.c`, y que también se explicará en la siguiente sección. Los códigos CF7, CF8 y CF9 causan llamadas `ttsigchar`, en `tty.c`. Cuando se agrega trabajo con redes a MINIX, se añade un caso adicional para detectar el código F5, a fin de exhibir datos estadísticos de Ethernet. Está disponible un gran número de códigos de detección adicionales que podrían usarse para disparar otros mensajes de depuración o eventos especiales de la consola.

`Scan_keyboard` (línea 13432) opera en el nivel de interfaz de hardware, leyendo y escribiendo rtes en los puertos de E/S. Se informa al controlador en hardware del teclado que un carácter fue leído por la secuencia de las líneas 13440 a 13442, que lee un byte, lo escribe otra vez con el bit más significativo puesto en 1, y luego lo reescribe con el mismo bit puesto en 0. Esto evita que se lean los mismos datos en una lectura subsecuente. No hay verificación de estado al leer el teclado, pero de cualquier manera no deberá haber problemas, ya que `scan_keyboard` sólo se invoca como respuesta a una interrupción, con excepción de la llamada desde `kb_init` para despejar cualquier basura que pudiera haber.

Tecla	Propósito
F1	Exhibir tablas de procesos
F2	Exhibir los detalles de uso de memoria de procesos
F3	Conmutar entre desplazamiento por hardware y por software
F5	Mostrar datos estadísticos de Ethernet (si se compiló apoyo de red)
CF7	Enviar SIGQUIT, mismo efecto que ctrl.-\
CF8	Enviar SIGINT, mismo efecto que DEL
CF9	Enviar SIGKILL, mismo efecto que ctrl.-U

Figura 3-47. Las teclas de función detectadas por `func_key()`.

La última función de `keyboard.c` es `wreboot` (línea 13450). Si se invoca como resultado de un pánico del sistema, `wreboot` ofrece al usuario la oportunidad de usar las teclas de función para exhibir información de depuración. El ciclo de las líneas 13478 a 13487 es otro ejemplo de espera activa. El teclado se lee repetidamente hasta que se tecléa ESC. Ciertamente nadie podrá asegurar que se necesite una técnica más eficiente después de una caída, mientras se espera un comando de rearranque. Dentro del ciclo, se invoca `func_key` para ofrecer la posibilidad de obtener información que podría ayudar a analizar la causa de una caída. No entraremos en detalles del retomo al monitor, pues son muy específicos para el hardware y no tienen mucho que ver con el sistema operativo.

3.9.6 Implementación del controlador de pantalla

La pantalla de la IBM PC se puede configurar como varias terminales virtuales si hay suficiente memoria disponible. En esta sección examinaremos el código dependiente del dispositivo para la consola, y también veremos las rutinas de vaciado de depuración que emplean servicios de bajo nivel del teclado y la pantalla. Éstas ofrecen apoyo para una interacción limitada con el usuario en la consola, incluso si otras partes del sistema MINIX no están funcionando, y pueden proporcionar información útil aun después de una caída casi total del sistema.

El apoyo específico del hardware para las salidas a la consola PC con pantalla mapeada en la memoria está en `consolé, c`. La estructura `consolé` se define en las líneas 13677 a 13693. En cierto sentido, esta estructura es una extensión de la estructura `tty` definida en `tty.c`. Durante la inicialización se asigna al campo `tp->tty_priv` de la estructura `tty` de una consola un apuntador a su propia estructura `consolé`. El primer elemento de la estructura `consolé` es un apuntador de regreso a la estructura `tty` correspondiente. Los componentes de una estructura `consolé` son lo que esperaríamos para una pantalla de video: variables para registrar la fila y la columna de la posición del

cursor, las direcciones de memoria del principio y el límite de la memoria empleada para la pantalla, la dirección de memoria a la que apunta el apuntador base del chip controlador, y la dirección actual del cursor. Se usan otras variables para manejar las secuencias de escape. Puesto que los caracteres inicialmente se reciben como bytes de ocho bits y deben combinarse con bytes de atributo y transferirse como palabras de 16 bits a la memoria de video, se construye el bloque por transferir en `c_ramqueue`, un arreglo con el tamaño suficiente para contener una fila completa de 80 columnas de pares carácter-atributo de 16 bits. Cada consola virtual necesita una estructura `consolé`, y la memoria para ella se asigna en el arreglo `cons_table` (línea 13696). Como hicimos con las estructuras `tty` y `kb_s`, usualmente nos referiremos a los elementos de una estructura `consolé` usando un apuntador; por ejemplo, `cons->c_tty`.

La función cuya dirección se almacena en la entrada `tp->tty_devwrite` de cada consola es `cons_write` (línea 13729). Esta función es invocada desde un solo lugar, `handle_events` en `tty.c`. La mayor parte del resto de las funciones de `consolé`, `c` existen para apoyar esta función. Cuando `cons_write` se invoca por primera vez después de que un proceso cliente efectúa una llamada `WRITE`, los datos por exhibir están en el buffer del cliente, que puede encontrarse usando los campos `tp->tty_outproc` y `tp->tty_virde` la estructura `tty`. El campo `tp->tty_outleft` indica cuántos caracteres deben transferirse, y el campo `tp->tty_outcum` inicialmente es cero, lo que indica que todavía no se ha transferido ninguno. Ésta es la situación usual cuando se ingresa en `cons_write`, porque normalmente, una vez invocada, transfiere todos los datos solicitados en la llamada original. Sin embargo, si el usuario quiere hacer más lento el proceso para poder revisar los datos en la pantalla, puede introducir un carácter `STOP` (`CTRL-S`) en el teclado, izando así la bandera `tp->tty_inhibited`. `Cons_write` regresa de inmediato cuando está izada esta bandera, aunque todavía no se haya completado el `WRITE`. En tal caso, `handle_events` continuará invocando `cons_write`, y cuando `tp->tty_inhibited` finalmente se arríe en el momento en que el usuario introduzca un carácter `START` (`CTRL-Q`), `cons_write` continuará con la transferencia interrumpida.

El único argumento de `cons_write` es un apuntador a la estructura `tty` de la consola en cuestión, así que lo primero que debe hacerse es inicializar `cons`, el apuntador a la estructura `consolé` de esa consola (línea 13741). Luego, dado que `handle_events` invoca `cons_write` cada vez que se ejecuta, la primera acción es una prueba para determinar si realmente hay trabajo que hacer. Se regresa rápidamente si no hay trabajo (línea 13746). Después de esto, se ingresa en el ciclo principal de las líneas 13751 a 13778. La estructura de este ciclo es muy similar a la del ciclo principal de `in_transfer` en `tty.c`. Se llena un buffer local que puede contener 64 caracteres invocando `phys_copy` para obtener los datos del buffer del cliente, se actualizan el apuntador al origen y las cuentas, y luego se transfiere cada uno de los caracteres del buffer local al arreglo `cons->c_ramqueue`, junto con un byte de atributo, para ser transferidos posteriormente a la pantalla por `flush`. Hay más de una forma de efectuar esta transferencia, como vimos en la Fig. 3-39. Se puede invocar `out_char` para que haga esto con cada carácter, pero es evidente que ninguno de los servicios especiales de `out_char` se necesitará si el carácter es un carácter visible, si no se está formando una secuencia de escape, si no se ha excedido el ancho de la pantalla y si `cons->c_ramqueue` no está llena. Si no se requiere el servicio completo de `out_char`, el carácter se coloca directamente en `cons->c_ramqueue` junto con el byte de atributo (obtenido de `cons->c_attr`), y se incrementan `cons->c_rwords` (el índice de la cola), `cons->c_column` (que sigue la pista a la columna en la pantalla) y `tbuf` (el apuntador

al buffer). Esta colocación directa de caracteres en `cons->c_ramqueue` corresponde a la línea interrumpida del lado izquierdo de la Fig. 3-39. Si es necesario, se invoca `out_char` (líneas 13766 a 13777). Éste se encarga de la contabilización y además invoca a `flush`, que realiza la transferencia final a la memoria de pantalla, si es necesario. La transferencia del buffer de usuario al buffer local y a la cola se repite en tanto `tp->tty_outleft` indica que aún hay caracteres que transferir y no se ha izado la bandera `tp->tty_inhibited`. Cuando se detiene la transferencia, sea porque se completó la operación `WRITE` o porque se izó `tp->tty_inhibited`, se invoca otra vez `flush` para transferir los últimos caracteres de la cola a la memoria de pantalla. Si la operación se completó (lo que se prueba viendo si `tp->tty_outleft` es cero), se envía un mensaje de respuesta invocando `tty_reply` (líneas 13784 y 13785).

Además de las llamadas a `cons_write` desde `handle_events`, los caracteres que se van a exhibir también son enviados a la consola por `echo` y `rawecho` en la parte independiente del hardware de la tarea de terminal. Si la consola es el dispositivo de salida vigente, las llamadas a través del apuntador `tp->tty_echo` se dirigen a la siguiente función, `cons_echo` (línea 13794). `Cons_echo` efectúa todo su trabajo invocando `out_char` y luego `flush`. Las entradas del teclado llegan carácter por carácter y la persona que está tecleando desea ver el eco sin un retraso perceptible, así que la colocación de los caracteres en la línea de salida no sería satisfactoria.

Ahora llegamos a `out_char` (línea 13809), la cual realiza una prueba para determinar si se esta formando una secuencia de escape, invocando `parse_escape` y luego regresando de inmediato si tal es el caso (líneas 13814 a 13816). En caso contrario, se ingresa en un `switch` para verificar los casos especiales: nulo, retroceso, el carácter de alarma, etc. El manejo de casi todos éstos es fácil de seguir. El salto de línea y la tabulación son los más complicados, ya que implican cambios complejos a la posición del cursor en la pantalla y también podrían requerir desplazamiento. La última prueba es para detectar el código ESC. Si se encuentra, se iza la bandera `cons->c_esc_stah` (línea 13871) y las llamadas futuras a `out_char` se desvían a `parse_escape` hasta que la secuencia está completa. Al final, se toma el valor predeterminado para los caracteres imprimibles. Si se ha excedido la anchura de la pantalla, es posible que sea necesario desplazar la pantalla, y se invoca `flush`. Antes de colocar un carácter en la cola de entrada, se realiza una prueba para determinar a la cola está llena, y se invoca `flush` si lo está. La colocación de un carácter en la cola requiere la misma contabilización que vimos antes en `cons_write`.

La siguiente función es `scroll_screen` (línea 13896). Esta función maneja tanto el desplazamiento de la pantalla hacia arriba, que es la situación normal que debe manejarse cada vez que se llena la línea inferior de la pantalla, como el desplazamiento hacia abajo, que ocurre cuando los comandos de movimiento del cursor intentan colocarlo más arriba de la línea superior de la pantalla. Para cada dirección de desplazamiento hay tres métodos posibles. Éstos son necesarios para apoyar diferentes clases de tarjetas de video.

Examinaremos el caso del desplazamiento hacia arriba. Para comenzar, se asigna a `chars` tamaño de la pantalla menos una línea. El desplazamiento por software se logra con una sola llamada a `vid_vid_copy` para mover `chars` caracteres más abajo en la memoria; el tamaño del bloque que se mueve es el número de caracteres en una línea. `Vid_vid_copy` puede continuar del final al principio; es decir, si se le pide mover un bloque de memoria que desborde el extremo superior del bloque asignado a la pantalla de video, obtiene la porción de desbordamiento del

extremo inferior del bloque de memoria y lo traslada a una dirección más alta que la parte que se mueve más abajo, tratando todo el bloque como un arreglo circular. La sencillez de la llamada oculta una operación relativamente lenta. Aunque `vid_vid_copy` es una rutina en lenguaje ensamblador definida en `klib386.s`, esta llamada requiere que la CPU mueva 3840 bytes, lo cual es un trabajo considerable incluso en lenguaje ensamblador.

El método de desplazamiento por software nunca es el predeterminado; se supone que el operador sólo lo seleccionará si el desplazamiento por hardware no funciona o no se desea por alguna razón. Una razón podría ser el deseo de usar el comando `screendump` para guardar la memoria de pantalla en un archivo. Cuando está vigente el desplazamiento por hardware, es común que `screendump` produzca resultados inesperados, porque es poco probable que el principio de la memoria de pantalla coincida con el principio de la pantalla visible.

En la línea 13917 se prueba la variable `wrap` como primera parte de una prueba compuesta. `Wrap` es `TRUE` en pantallas más viejas que pueden manejar el desplazamiento por hardware, y si la prueba falla ocurre un desplazamiento sencillo por hardware en la línea 13921, donde el apuntador al origen empleado por el chip controlador de video, `cons->c_org`, se actualiza de modo que apunte al primer carácter que se exhibirá en la esquina superior izquierda de la pantalla. Si `wrap` es `FALSE`, la prueba compuesta continuará con una prueba de si el bloque que se va a subir en la operación de desplazamiento desborda los límites del bloque de memoria designado para esta consola. Si es así, se invoca otra vez `vid_vid_copy` para realizar un traslado con continuidad del bloque al principio de la memoria asignada a la consola, y se actualiza el apuntador al origen. Si no hay traslapo, el control pasa al método simple de desplazamiento por hardware que siempre usan los controladores en hardware de video más viejos. Éste consiste en ajustar `cons->c_org` y luego colocar el nuevo origen en el registro correcto del chip controlador. La llamada para efectuar esto se emite posteriormente, lo mismo que una llamada para poner en blanco la línea inferior de la pantalla.

El código para el desplazamiento hacia abajo es muy similar al del desplazamiento hacia arriba. Por último, se llama `mem_vid_copy` para poner en blanco la línea inferior (o superior) direccionada por `new_line`. Luego se invoca `set_6845` para copiar el nuevo origen de `cons->c_org` a los registros apropiados, y `flush` se asegura de que todos los cambios se vean en la pantalla. Hemos mencionado `&flush` (línea 13951) varias veces. Esta función transfiere los caracteres que están en la cola a la memoria de video usando `mem_vid_copy`, actualiza ciertas variables y luego se asegura de que los números de fila y columna sean razonables, ajustándolos si, por ejemplo, una secuencia de escape ha tratado de colocar el cursor en una posición de columna negativa. Por último, se calcula dónde debe estar el cursor y se compara esta posición con `cons->c_cur`. Si estos valores no coinciden, y si la memoria de video que se está manejando actualmente pertenece a la consola virtual vigente, se efectúa una llamada a `set_6845` para estacar el valor correcto en el registro del cursor del controlador.

En la Fig. 3-48 se muestra cómo se puede representar el manejo de secuencias de escape no máquina de estados finitos. Esto se implementa con `parse_escape` (línea 13986) que se roca al principio de `out_char` si `cons->c_esc_state` es distinto de cero. Un `ESC`, en sí, es detectado por `out_char` y hace que `cons->c_esc_state` sea igual a 1. Cuando se recibe el siguiente carácter `parse_escape` se prepara para el procesamiento ulterior colocando `'\0'` en `cons->c_esc_intro`

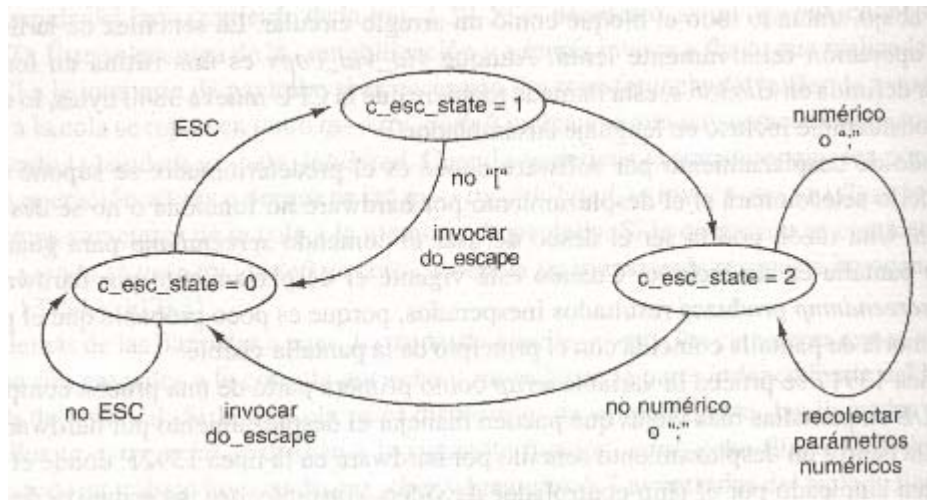


Figura 3-48. Máquina de estados finitos para procesar secuencias de escape.

(un apuntador al principio del arreglo de parámetros), `cons->c_esc_parmv[0]` en `cons->esc_parmp` y ceros en el arreglo de parámetros mismo. Luego se examina el primer carácter después del ESC; los valores válidos son "[" o bien "M". En el primer caso, el "[" se copia en `cons->c_intro` y el estado se avanza a 2. En el segundo caso, se invoca `do_escape` para efectuar la acción, y el estado de escape se restablece a cero. Si el primer carácter después del ESC no es uno de los válidos, se hace caso omiso de él y los caracteres subsecuentes se exhiben normalmente otra vez.

Si se detectó una secuencia ESC [, el siguiente carácter introducido es procesado por el código del estado de escape 2. En este punto hay tres posibilidades. Si el carácter es numérico, se extrae su valor y se suma a 10 veces el valor existente en la posición a la que actualmente apunta `cons->c_esc_parmp`, inicialmente `cons->c_esc_parmv[0]` (que se inicializó en cero). El estado de escape no cambia. Esto hace posible introducir una serie de dígitos decimales y acumular un parámetro numérico grande, aunque el valor máximo que MINIX reconoce actualmente es 80, empleado por la secuencia que mueve el cursor a una posición arbitraria (líneas 14027 a 14029). Si el carácter es un signo de punto y coma, se avanza el apuntador a la cadena de parámetros, a fin de que los valores numéricos subsecuentes puedan acumularse en el segundo parámetro (líneas 14031 a 14033). Si se cambiara `MAX_ESC_PARAMS` con objeto de reservar un arreglo; más grande para los parámetros, no sería necesario alterar este código para guardar valores numéricos adicionales después de la introducción de parámetros adicionales. Por último, si el carácter no es un dígito numérico ni un signo de punto y coma, se invoca `do_escape`.

`Do_escape` (línea 14045) es una de las funciones más largas del código fuente del sistema MINIX, aunque la cantidad de secuencias de escape que MINIX reconoce es relativamente modesta.

Pese a su longitud, el lector no deberá tener problemas para seguir el código. Después de una llamada inicial a flush para asegurarse de que la pantalla de video esté totalmente actualizada, hay una decisión if sencilla, dependiendo de si el carácter que sigue de inmediato al carácter ESC es el introductor de secuencias de control especiales o no. Si no, sólo hay una acción válida, subir el cursor una línea si la secuencia fue ESC M. Cabe señalar que la prueba de la "M" se efectuó dentro de un switch que tiene una acción por omisión, como verificación de validez y anticipando la adición de otras secuencias que no utilicen el formato ESC [. La acción es representativa de muchas secuencias de escape: se examina la variable `cons->c_row` para determinar si es necesario desplazar la pantalla. Si el cursor ya está en la fila 0, se emite una llamada `SCROLL_DOWN` a `scroll_screen`, si no, se sube el cursor una línea. Esto último se logra decrementando `cons->c_row` e invocando flush. Si se encuentra un introductor de secuencias de control, se ejecuta el código que sigue al else en la línea 14069. Se realiza una prueba para "[", el único introductor de secuencias de control que MINIX reconoce actualmente. Si la secuencia es válida, se asigna a `valué` el primer parámetro encontrado en la secuencia de escape, o cero si no se introdujo ningún parámetro numérico (línea 14072). Si la secuencia no es válida, lo único que sucede es que el switch grande que sigue (líneas 14073 a 14272) se pasa por alto y el estado de escape se restablece a cero antes de regresar de `do_escape`. En el caso más interesante de que la secuencia sea válida, se ingresa en el switch. No analizaremos todos los casos; sólo mencionaremos varios de ellos que son representativos de los tipos de acciones gobernadas por las secuencias de escape.

Las primeras cinco secuencias son generadas, sin argumentos numéricos, por las cuatro teclas de "flecha" y la tecla Inicio (Home) del teclado de la IBM PC. Las primeras dos, ESC [A y ESC [B, son similares a ESC M, excepto que pueden aceptar un parámetro numérico y subir y bajar más de una línea, y no desplazan la pantalla si el parámetro especifica un movimiento que excede los límites de la pantalla. En tales casos, flush atrapa las peticiones que piden salirse de los límites y limita el movimiento a la última fila o a la primera, según resulte apropiado. Las siguientes dos secuencias, ESC [C y ESC [D, que mueven el cursor a la derecha y a la izquierda, están igualmente limitadas por flush. Cuando estas secuencias son generadas por las teclas de flecha, no hay argumento numérico, y por tanto ocurre el movimiento por omisión de una línea o una columna.

La siguiente secuencia, ESC [H, puede tomar dos parámetros numéricos, por ejemplo, ESC [20;60H. Los parámetros especifican una posición absoluta, no una relativa a la posición actual, y se convierten de números basados en 1 a números basados en 0 para su correcta interpretación. La tecla Inicio genera la secuencia predeterminada (sin parámetros) que lleva el cursor a la posición (1, 1).

Las siguientes dos secuencias, ESC [sJ y ESC [sK, despejan una parte ya sea de la pantalla completa o de la línea actual, dependiendo del parámetro introducido. En cada caso se calcula una cuenta de caracteres. Por ejemplo, en el caso de ESC [IJ, `count` recibe el número de caracteres desde el principio de la pantalla hasta la posición del cursor, y la cuenta y un parámetro de posición, `dst`, que puede ser el principio de la pantalla, `cons->c_org`, o la posición actual del cursor, `cons->c_cur`, se usan como parámetros para una llamada a `mem_vid_copy`. Este procedimiento se invoca con un parámetro que hace que llene la región especificada con el color de segundo plano vigente.

Las siguientes cuatro secuencias insertan y eliminan líneas y espacios en la posición del cursor y sus acciones no requieren una explicación detallada. El último caso, ESC [nm (tome nota de que la n representa un parámetro numérico, pero la "m" es un carácter literal) afecta cons->c_attr, el byte de atributo que se intercala entre los códigos de carácter cuando se escriben en la memoria de video.

La siguiente función, set_6845 (línea 14280) se usa cada vez que es necesario actualizar el chip controlador de video. El 6845 tiene registros internos de 16 bits que se programan 8 bit a la vez, y la escritura de un solo registro requiere cuatro operaciones de escritura en un puerto de E/S. Se usan llamadas lock y unlock para inhabilitar las interrupciones, que pueden causar problemas si se permite que alteren la secuencia. En la Fig. 3-49 se muestran algunos de los registros del chip controlador de video 6845.

Registros	Función
10-11	Tamaño del cursor
12-13	Dirección donde se comenzará a dibujar la pantalla
14-15	Posición del cursor.

Figura 3-49. Algunos de los registros del 6845.

La función beep (línea 14300) se invoca cuando es preciso enviar a la salida un cara CTRL-G. Esta función aprovecha el apoyo integrado que la PC proporciona para emitir sonidos y enviar una onda cuadrada al altavoz. El sonido se inicia con más de las manipulaciones mágicas de los puertos de E/S que sólo los programadores en lenguaje ensamblador pueden amar, otra vez cuidando hasta cierto punto que una parte crítica del proceso quede protegida contra interrupciones. La parte más interesante del código es el empleo de la capacidad de la tarea del reloj para fijar una alarma, que puede servir para iniciar una función. La siguiente rutina, stop_beep (línea 14329), es aquella cuya dirección se pone en el mensaje que se envía a la tarea del reloj. Stop_beep detiene el sonido una vez que ha transcurrido el tiempo designado y también pone en cero la bandera beeping que sirve para evitar que las llamadas superfluas a la rutina de la alarma audible tengan algún efecto.

Scr_init (línea 14343) es invocada NR_CONS veces por tty_init. En cada ocasión, su argumentó es un apuntador a una estructura tty, un elemento de tty_table. En las líneas 14354 y 14355 se calcula Une, que se usará como índice del arreglo cons_table, se prueba su validez y, si es válida, se usa para inicializar cons, el apuntador a la entrada de la consola actual en la tabla. En este punto se puede inicializar el campo cons->c_tty con el apuntador a la estructura tty principal para el dispositivo y, a su vez, se puede hacer que tp->tty_priv apunte a la estructura console_t de este dispositivo. A continuación se invoca kb_init para inicializar el teclado, y luego se establecen los apuntadores a rutinas específicas para los dispositivos, haciendo que tp->tty_devwrite apunte a cons_write y que tp->tty_echo apunte a cons_echo. En las líneas 14368 a 14378 se obtiene la dirección de E/S del registro base del controlador en hardware del CRT y se determina la dirección y el tamaño de la memoria de video, y se enciende la bandera wrap (empleada para determinar cómo desplazar la pantalla) según la clase de controlador en hardware de video que se esta

usando. En las líneas 14382 a 14384 se inicializa el descriptor de segmento para la memoria de video en la tabla de descriptores globales.

A continuación viene la inicialización de las consolas virtuales. Cada vez que se invoca `scr_init`, el argumento es un valor diferente de `tp`, y por tanto se usan un `Une` y un `cons` diferentes en las líneas 14393 a 14396 para proporcionar a cada consola virtual su porción correspondiente de la memoria de video disponible. Luego se pone en blanco cada una de las pantallas, en preparación para el inicio, y por último se selecciona la consola `O` como primera consola activa.

El resto de las rutinas de consolé, `c` son cortas y simples y las repasaremos rápidamente. Ya mencionamos a `putk` (línea 14408). Esta rutina exhibe un carácter a nombre de cualquier código enlazado a la imagen del kernel que requiera el servicio, sin pasar por el sistema de archivos. `Toggle_scroll` (línea 14429) hace lo que indica su nombre, conmuta el valor de la bandera que determina si se emplea desplazamiento por hardware o por software; además, exhibe un mensaje en la posición actual del cursor para identificar el modo seleccionado. `Cons_stop` (línea 14442) reinicializa la consola en el estado que espera el monitor de arranque, antes de una clausura o un rearranque. `Cons_org0` (línea 14456) sólo se usa cuando se obliga a cambiar de modo de desplazamiento con la tecla `F3`, o durante la preparación para una clausura. `Select_console` (línea 14482) selecciona una consola virtual; se invoca con el nuevo índice y a su vez invoca `set_6845` dos veces para hacer que el controlador de video exhiba la parte correcta de la memoria de video.

Las últimas dos rutinas dependen en alto grado del hardware. `ConJ.oadfnt` (línea 14497) carga un tipo de letra en un adaptador de gráficos para apoyar la operación de `IOCTL TIOCSFON`. Además, invoca `ga_program` (línea 14540) para que realice una serie de escrituras mágicas en un puerto de E/S que hacen que la memoria de tipo de letra del adaptador de video, que la CPU normalmente no puede direccionar, sea visible. Luego se invoca `phys_copy` para copiar los datos de tipo de letra en esta área de memoria, y se invoca otra secuencia mágica para regresar el adaptador de gráficos a su modo de operación normal.

Vaciados para depuración

El grupo final de procedimientos que examinaremos en la tarea de la terminal se diseñaron originalmente para ser utilizados sólo temporalmente al depurar MINIX. Éstos se pueden eliminar si no se necesita su ayuda, pero muchos usuarios los encuentran útiles y los dejan en su lugar, pues son especialmente eficaces cuando se desea modificar MINIX.

Como hemos visto, `func_key` se invoca al principio de `kb_read` para detectar los códigos de detección empleados para control y depuración. Las rutinas de vaciado que se invocan cuando se detectan las teclas `F1` y `F2` están en `dmp.c`. La primera, `p_dmp` (línea 14613) exhibe información básica de todos los procesos, incluida cierta información sobre uso de memoria, cuando se oprime la tecla `F1`. La segunda, `map_dmp` (línea 14660) proporciona información más detallada sobre el uso de la memoria cuando se oprime `F2`. `Proc_name` (línea 14690) apoya `p_dmp` buscando los nombres de los procesos.

Puesto que este código está totalmente contenido dentro del código binario del kernel mismo y no se ejecuta como proceso de usuario ni como tarea, es frecuente que siga funcionando clámente incluso después de una caída seria del sistema. Desde luego, sólo se puede acce-

der a estas rutinas desde la consola. La información provista por las rutinas de vaciado no puede redirigir a un archivo ni a ningún otro dispositivo, así que una copia impresa o el empleo a través de una conexión de red quedan excluidos.

Sugerimos que el primer paso al tratar de agregar cualquier mejora a MINIX sea la extensión de las rutinas de vaciado a modo de proporcionar más información sobre el aspecto del sistema que se desea mejorar.

3.10 LA TAREA DE SISTEMA EN MENIX

Una consecuencia de hacer que el sistema de archivos y el administrador de memoria sean procesos servidores externos al kernel es que ocasionalmente tienen información que el kernel necesita. Sin embargo, esta estructura les impide escribirla simplemente en una tabla del kernel. Por ejemplo, la llamada al sistema FORK es manejada por el administrador de memoria. Cuando se crea un proceso nuevo, el kernel debe enterarse de ello para poder planificarlo. ¿Cómo puede el administrador de memoria decirselo al kernel?

La solución a este problema es tener una tarea del kernel que se comuniquen con el sistema de archivos y el administrador de memoria por medio del mecanismo de mensajes estándar y que también tenga acceso a todas las tablas del kernel. Esta tarea, llamada tarea del sistema, está en la capa 2 de la Fig. 2-26, y funciona igual que las otras tareas que hemos estudiado en este capítulo. La única diferencia es que la tarea del sistema no controla ningún dispositivo de E/S; en vez de ello, al igual que las tareas de E/S, implementa una interfaz que en este caso no es con el mundo externo sino con la parte más interna del sistema. Esta tarea tiene los mismos privilegios que las tareas de E/S y se compila junto con ellos en la imagen del kernel, así que es más lógico estudiarla aquí que en otro capítulo.

La tarea del sistema acepta 19 tipos de mensajes, los cuales se muestran en la Fig. 3-50. El programa principal de la tarea del sistema, `sys_task` (línea 14837), está estructurado igual que las otras tareas: recibe un mensaje, invoca el procedimiento de servicio apropiado, y luego envía una respuesta. A continuación examinaremos cada uno de estos mensajes y su procedimiento de servicio.

El administrador de memoria utiliza el mensaje `SYS_FORK` para decirle al kernel que acaba de nacer un nuevo proceso. El kernel necesita saber esto para poder planificarlo. El mensaje contiene los números de ranura dentro de la tabla de procesos que corresponden al padre y al hijo. El administrador de memoria y el sistema de archivos también tienen tablas de procesos, y en las tres la entrada `k` se refiere al mismo proceso. De este modo, el administrador de memoria puede especificar sólo los números de ranura del padre y del hijo y el kernel sabrá a qué procesos se refieren.

El procedimiento `do_fork` (línea 14877) primero verifica (línea 14886) si el administrador de memoria está alimentando basura al kernel. La prueba utiliza una macro, `isokusern`, definida en `proc.h`, para determinar si las entradas para el padre y el hijo en la tabla de procesos son válidas. La mayor parte de los procedimientos de servicio de `system.c` efectúan pruebas similares. Esto es pura paranoia, pero un poco de verificación de la consistencia interna no hace daño. Luego `do_fork`

Tipo de mensaje	De	Significado
SYSFORK	MM	Un proceso bifurcó
SYSNEWMAP	MM	MM instala mapa de memoria para un nuevo proceso
SYSGETMAP	MM	MM quiere el mapa de memoria de un proceso
SYSEXEC	MM	Establece el apuntador a la pila después de llamar a EXEC
SYSXIT	MM	Un proceso salió
SYSGETSP	MM	MM quiere el apuntador a la pila de un proceso
SYSTIMES	FS	FS quiere los tiempos de ejecución de un proceso
SYSABORT	Ambos	Pánico: MINIX no puede continuar
SYSSENDSIG	MM	Envía una señal a un proceso
SYSSIGRETURN	MM	Aseo después de completarse una señal
SYSKILL	FS	Enviar una señal a un proceso después de la llamada KILL
SYSENDSIG	MM	Aseo después de una señal del kernel
SYS COPY	Ambos	Copia datos entre procesos
SYSVCOPY	Ambos	Copia múltiples bloques de datos entre procesos
SYSGBOOT	FS	Obtiene parámetros de arranque
SYSTEMEM	MM	MM quiere el siguiente trozo libre de memoria física
SYSUMAP	FS	Convierte direcciones virtuales en físicas
SYS TRACE	MM	Realiza una operación de la llamada PTRACE

Figura 3-50. Los tipos de mensajes aceptados por la tarea del sistema.

copia la entrada de tabla del proceso padre en la ranura del hijo. Aquí es necesario ajustar algunas cosas. El hijo queda liberado de cualquier señal pendiente para el padre, y no hereda la situación de rastreo del padre. Y, desde luego, toda la información de contabilidad del hijo se pone en cero.

Después de un FORK, el administrador de memoria asigna memoria al hijo. El kernel debe saber dónde está situado el hijo en la memoria a fin de poder establecer debidamente los registros de segmento al ejecutar el hijo. El mensaje SYS_NEWMAP permite al administrador de memoria dar al kernel el mapa de memoria de cualquier proceso. Este mensaje también puede usarse después de que una llamada al sistema BRK modifica el mapa.

El mensaje es `man&jaAo por do_newmap <\ínea 1491V)`, que primero áebe copiar e\ muevo mapa del espacio de direcciones del administrador de memoria. El mapa no está contenido en el mensaje mismo porque es demasiado grande. En teoría, el administrador de memoria podría decirle al kernel que el mapa está en la dirección `m`, donde `m` es una dirección no permitida. No se supone que el administrador de memoria haga esto, pero el kernel de todos modos lo verifica. El

mapa se copia directamente en el campo `p_map` de la entrada de tabla de procesos correspondiente al proceso que está obteniendo el nuevo mapa. La llamada a `alloc_segments` extrae información del mapa y la carga en los campos `p_reg` que contienen los registros de segmento. Esto no es complicado, pero los detalles dependen del procesador y por esta razón se segregan en una función aparte.

El mensaje `SYS_NEWMAP` se usa mucho en el funcionamiento normal de un sistema MINIX. Un mensaje similar, `SYS_GETMAP` se usa sólo cuando el sistema de archivos arranca inicialmente. Este mensaje solicita una transferencia de la información de mapa de proceso en la dirección opuesta, del kernel al administrador de memoria. La transferencia es efectuada por `do_getmap` (línea 14957). El código de las dos funciones es similar, y difiere principalmente en el intercambio de los argumentos de origen y destino de la llamada a `phys_copy` utilizada por cada función.

Cuando un proceso realiza una llamada al sistema `EXEC`, el administrador de memoria establece una nueva pila para él que contiene los argumentos y el entorno, y pasa el apuntador de pila resultante al kernel usando `SYS_EXEC`, que es manejado por `do_exec` (línea 14990). Después de la verificación usual de la validez del proceso, hay una prueba del campo `PROC2` del mensaje. Este campo se utiliza aquí como bandera para indicar si el proceso está siendo rastreado y nada tiene que ver con la identificación de los procesos. Si está vigente el rastreo, se invoca `cause_sig` para enviar una señal `SIGTRAP` al proceso. Esto no tiene las consecuencias usuales de esta señal, que normalmente terminaría el proceso de destino y causaría un vaciado de núcleo. En el administrador de memoria todas las señales enviadas a un proceso rastreado, con excepción de `SIGKILL`, son interceptadas y hacen que el proceso al que se envió la señal se detenga para que un programa depurador pueda controlar su posterior ejecución a partir de ese momento.

La llamada `EXEC` causa una pequeña anomalía. El proceso que invoca la llamada envía un mensaje al administrador de memoria y se bloquea. Con otras llamadas al sistema, la respuesta resultante lo desbloquea. Con `EXEC` no hay respuesta, porque la imagen de núcleo recién cargada no está esperando una respuesta. Por tanto, `do_exec` desbloquea por cuenta propia el proceso en la línea 15009. La siguiente línea prepara la nueva imagen para ejecutarse, usando la función `lock_ready` que protege contra una posible condición de competencia. Por último, la cadena de comandos se guarda para que el proceso se pueda identificar cuando el usuario oprima la tecla `F1` a fin de exhibir la situación de todos los procesos.

Los procesos pueden salir en MINIX ya sea ejecutando una llamada al sistema `EXIT`, que envía un mensaje al administrador de memoria, o siendo terminados por una señal. En ambos casos, el administrador de memoria le informa al kernel usando el sistema `SYS_XIT`. El trabajo es efectuado por `do_xit` (línea 15027), que es más complicada de lo que podríamos esperar. Encargarse de la información de contabilidad es sencillo. El temporizador de alarma, si lo hay, se anula almacenando un cero encima de él. Es por esta razón que la tarea del reloj siempre verifica cuándo se ha vencido un temporizador para ver si todavía hay alguien interesado. La parte rebuscada de `do_xit` es que el proceso podría haber estado en cola tratando de enviar o recibir en el momento en que fue terminado. El código de las líneas 15056 a 15076 prueba esta posibilidad. Si el proceso que sale se encuentra en la cola de mensajes de cualquier otro proceso, se le retira con cuidado.

En contraste con el mensaje anterior, que es un tanto complicado, `SYS_GETSP` es absolutamente trivial. El administrador de memoria lo usa para averiguar el valor del apuntador de pila

actual de algún proceso. Este valor se necesita para las llamadas al sistema BRK y SBRK, con objeto de ver si el segmento de datos y el segmento de pila han chocado. El código está en `do_getsp` (línea 15089).

Ahora llegamos a uno de los pocos tipos de mensaje utilizados exclusivamente por el sistema de archivos, `SYS_TIMES`. Éste se necesita para implementar la llamada al sistema `TIMES`, que devuelve los tiempos de contabilización al invocador. Todo lo que `do_times` (línea 15106) hace es colocar los tiempos solicitados en el mensaje de respuesta. Se usan llamadas a `lock` y `unlock` para proteger contra una posible competencia mientras se accede a los contadores de tiempo.

Puede ocurrir que el administrador de memoria o bien el sistema de archivos descubra un error que haga imposible continuar operando. Por ejemplo, si al arrancar inicialmente el sistema de archivos se percata de que el superbloque en el dispositivo raíz se ha corrompido irremediablemente, entra en pánico y envía un mensaje `SYS_ABORT` al kernel. También es posible que el superusuario fuerce un retomo al monitor de arranque y/o un rearranque, empleando el comando `reboot`, que a su vez emite la llamada al sistema `REBOOT`. En cualquiera de estos casos, la tarea del sistema ejecuta `do_abort` (línea 15131), que copia instrucciones al monitor, si es necesario, y luego invoca `wreboot` para completar el proceso.

La mayor parte del trabajo de manejo de señales es efectuada por el administrador de memoria, que verifica si el proceso al que se va a enviar una señal está habilitado para atrapar o ignorar la señal, si quien envía la señal tiene derecho a hacerlo, etc. Sin embargo, el administrador de memoria no puede él mismo causar la señal, lo que requiere meter cierta información en la pila del proceso al que se envía la señal.

El manejo de señales antes de POSIX era problemático porque el hecho de atrapar una señal restauraba la respuesta predeterminada a las señales. Si se requería continuar con el manejo especial de las señales subsecuentes, el programador no podía garantizar la confiabilidad. Las señales son asincronas, y bien podía ser que llegara una segunda señal antes de volverse a habilitar el manejo. El manejo de señales al estilo POSIX resuelve este problema, pero el precio es un mecanismo más complicado. El sistema operativo implementaba el manejo de señales al estilo antiguo metiendo cierta información en la pila del proceso destinatario de la señal, de forma similar a como una interrupción agrega información. Entonces, el programador escribía un nejadador que terminaba con una instrucción de retomo, removiendo la información necesaria a continuar con la ejecución. Cuando se recibe una señal, POSIX guarda más información de la que puede manejarse sin problemas de esta manera. Después, se debe efectuar trabajo adicional, antes de que el proceso que recibió la señal pueda reanudar lo que estaba haciendo. Por tanto, el administrador de memoria tiene que enviar dos mensajes a la tarea del sistema para procesar una señal. La recompensa por este esfuerzo es un manejo más confiable de las señales.

Cuando se debe enviar una señal a un proceso, se envía el mensaje `SYS_SENDSIG` a la tarea del sistema, y el trabajo es efectuado por `do_sendsig` (línea 15157). La información necesaria para manejar señales al estilo POSIX está en una estructura `sigcontext`, en la que se guarda el contenido del registro del procesador, y una estructura `sigframe`, que contiene información sobre la forma en que el proceso debe manejar las señales. Ambas estructuras requieren cierta inicialización, pero el trabajo básico de `do_sendsig` consiste sólo en colocar la información requerida en la pila del proceso destinatario de la señal y ajustar el contador de programa y al apuntador de pila de

dicho proceso de modo que el código de manejo de señales se ejecute la siguiente vez que el planificador permita que el proceso se ejecute.

Cuando un manejador de señales al estilo POSIX termina su trabajo, no remueve la dirección donde se reanuda la ejecución del proceso interrumpido, como sucede con las señales al estilo antiguo. El programador que escribe el manejador incluye una instrucción `return` (o su equivalente en un lenguaje de alto nivel), pero la manipulación de la pila por parte de la llamada `SENDSIG` hace que la instrucción `return` cause una llamada al sistema `SIGRETURN`. El administrador de memoria envía entonces un mensaje `SYS_SIGRETURN` a la tarea del sistema. Este mensaje es manejado por `do_sigretum` (línea 15221), que copia la estructura `sigcontext` de vuelta en el espacio del kernel y luego restaura los registros del proceso destinatario de la señal. La próxima vez que el planificador permita que el proceso interrumpido se ejecute, éste reanudará su ejecución en el punto en el que fue interrumpido, conservando cualquier manejo especial de señales que se haya establecido previamente.

La llamada al sistema `SIGRETURN`, a diferencia de la mayor parte de las otras llamadas que describimos en esta sección, no es requisito de POSIX. Se trata de una invención de MINIX, una forma cómoda de iniciar el procesamiento necesario cuando finaliza el manejador de señales. No es conveniente que los programadores usen esta señal, pues no será reconocida por otros sistemas operativos, y en cualquier caso no hay necesidad de hacer referencia explícita a ella.

Algunas señales provienen del interior de la imagen del kernel, o son manejadas por el kernel antes de pasar al administrador de memoria. Éstas incluyen las señales que se originan en las tareas, como las alarmas de la tarea del reloj, o las digitaciones que causan señales y que son detectadas por la tarea de la terminal, así como las señales causadas por excepciones (como una división entre cero o instrucciones no permitidas) detectadas por la CPU. Las señales que se originan en el sistema de archivos también son manejadas primero por el kernel. El sistema de archivos usa el mensaje `SYS_KILL` para solicitar la generación de una señal de este tipo. El nombre tal vez sea un tanto engañoso, pues esto nada tiene que ver con el manejo de la llamada al sistema `KILL`, empleada por los procesos ordinarios para enviar señales. Este mensaje es manejado por `do_kill` (línea 15276), que realiza la verificación usual de la validez del origen del mensaje y luego invoca `cause_sig` para pasar realmente la señal al proceso. Las señales que se originan en el kernel también se pasan al proceso mediante una llamada a esta función, que inicia las señales enviando un mensaje `KSIG` al administrador de memoria.

Cuando el administrador de memoria termina con una de estas señales tipo `KSIG`, envía un mensaje `SYS_ENDSIG` de vuelta a la tarea del sistema. Este mensaje es manejado por `do_endsig` (línea 15294), que decrementa la cuenta de señales pendientes y, si llega a cero, apaga el bit `SIG_PENDING` para el proceso destinatario de la señal. Si no hay otras banderas encendidas que indiquen razones por las que el proceso no debe ejecutarse, se invocará `lock_ready` para permitir que el proceso se ejecute otra vez.

El mensaje `SYS_COPY` es el que más se usa, pues se necesita para que el sistema de archivos y el administrador de memoria puedan copiar información de y a los procesos de usuario.

Cuando un usuario ejecuta una llamada `READ`, el sistema de archivos examina su caché para ver si tiene el bloque que se necesita. Si no es así, el FS envía un mensaje a la tarea de disco apropiada para cargarlo en el caché, y luego envía otro mensaje a la tarea del sistema para decirle

le copie el bloque en el espacio de direcciones del proceso de usuario. En el peor de los casos se necesitan siete mensajes para leer un bloque; en el mejor de los casos se necesitan cuatro mensajes. Ambos casos se muestran en la Fig. 3-51. Estos mensajes son una fuente importante de gasto extra en MINIX y son el precio que se paga por un diseño altamente modular.

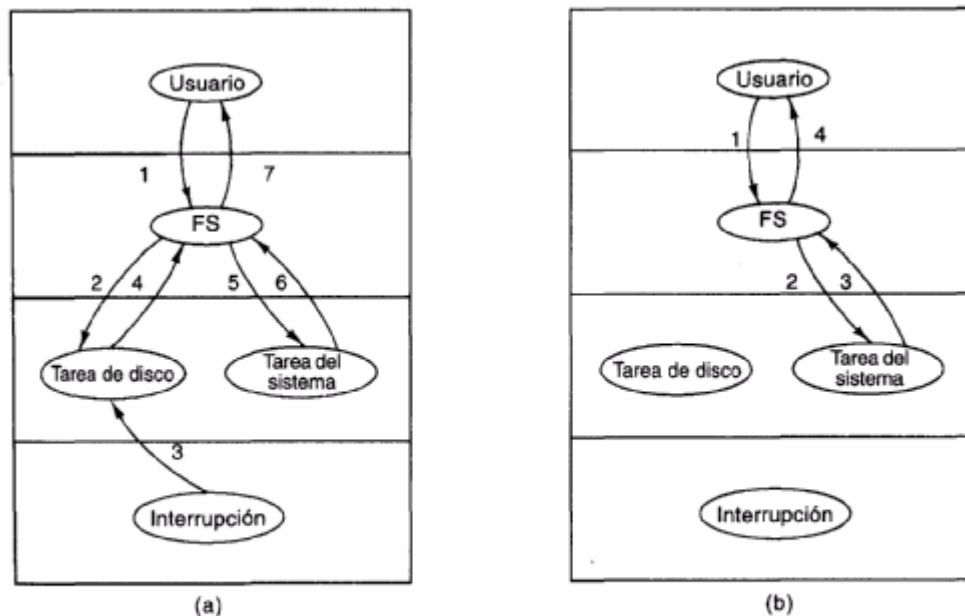


Figura 3-51. (a) En el peor de los casos se requieren siete mensajes para leer un bloque. (b) En el mejor de los casos sólo se requieren cuatro mensajes.

Como acotación, en el 8088, que no tenía protección, habría sido fácil hacer trampa y dejar que el sistema de archivos copiara los datos en el espacio de direcciones del invocador, pero esto habría violado el principio de diseño. Cualquier persona que tenga acceso a una máquina tan antigua y que esté interesada en mejorar el rendimiento de MINIX debería examinar detenidamente este mecanismo para ver cuánto comportamiento inapropiado se puede tolerar a fin de obtener ganancia en el rendimiento y cuánta sería ésta. Desde luego, esta forma de mejoramiento no está disponible en las máquinas de clase Pentium, que tienen mecanismos de protección.

El manejo de una petición SYS_COPY es directo; corre por cuenta de `do_copy` (línea 15316) y consiste casi exclusivamente en la extracción de los parámetros del mensaje y una invocación de `pis_copy`.

Una forma de reducir la ineficiencia del mecanismo de transferencia de mensajes es empaquetar múltiples peticiones en un solo mensaje. El mensaje SYS_VCOPY se encarga de esto. El contenido de este mensaje es un apuntador a un vector que especifica múltiples bloques que deberán copiarse entre posiciones de memoria. La función `do_vcopy` (línea 15364) ejecuta un ciclo, extrayendo las direcciones de origen y de destino y las longitudes de los bloques e invocando

phys_copy repetidamente hasta finalizar todas las operaciones de copiado. Esto es similar a la capacidad de los dispositivos de disco para manejar múltiples transferencias con base en una sola petición.

La tarea del sistema recibe varios tipos de mensajes más, casi todos muy sencillos. Dos de éstos normalmente sólo se usan durante el inicio del sistema. El sistema de archivos envía un mensaje SYS_GBOOT para solicitar los parámetros de arranque. Ésta es una estructura, bparam_s declarada en include/minix/boot.h, que permite al programa monitor del arranque especificar diversos aspectos de la configuración del sistema antes de iniciarse MINIX. La función do_gboot (línea 15403) lleva a cabo esta operación, que no es más que un copiado de una parte de la memoria a otra. También durante el inicio, el administrador de memoria envía a la tarea del sistema una serie de mensajes SYS_MEM para solicitar la base y el tamaño de los trozos de memoria disponibles. Do_mem (línea 15424) maneja esta petición.

El mensaje SYS_UMAP es utilizado por procesos fuera del kernel para solicitar el cálculo de la dirección de memoria física para una dirección virtual dada. Do_umap (línea 15445) realiza esto invocando umap, que es la función que se invoca desde dentro del kernel para efectuar esta conversión.

El último tipo de mensaje que examinaremos es SYS_TRACE, que apoya la llamada al sistema PTRACE, la cual se utiliza para depuración. La depuración no es una función fundamental de sistema operativo, pero el apoyo del sistema operativo puede simplificarla considerablemente. Con la ayuda del sistema operativo, un depurador puede inspeccionar y modificar la memoria utilizada por un proceso sometido a prueba, así como el contenido de los registros del procesador, los cuales se almacenan en la tabla de procesos mientras el programa que se está depurando no está en ejecución.

Normalmente, un proceso se ejecuta hasta que se bloquea para esperar E/S o hasta que agota un cuanto de tiempo. Casi todos los diseños de CPU ofrecen también un mecanismo que permite limitar a los procesos a la ejecución de una sola instrucción, o hacer que los procesos se ejecuten sólo hasta llegar a una instrucción dada, estableciendo un punto interruptor. El aprovechamiento de estas posibilidades favorece un análisis detallado de los programas.

Hay once operaciones que pueden llevarse a cabo usando PTRACE. En la ejecución de unas cuantas únicamente interviene el administrador de memoria, pero para las demás el administrador de memoria envía un mensaje SYS_TRACE a la tarea del sistema, la cual entonces invoca do_trace (línea 15467). Esta función implementa un switch basado en el código de la operación de rastreo. Las operaciones generalmente son sencillas. MINIX utiliza un bit P_STOP en la tabla de procesos para reconocer que se está efectuando depuración, y es asignado por el comando para detener el proceso (caso T_STOP) o reasignado para reiniciarlo (caso TJRESUME). La depuración depende del apoyo de hardware, y en los procesadores Intel está bajo el control de un bit del registro de banderas de la CPU. Cuando ese bit está asignado, el procesador sólo ejecuta una instrucción, y luego genera una excepción SIGTRAP. Como ya se mencionó, el administrador de memoria detiene el programa que se está rastreando cuando se le envía una señal. Este TRACEBIT es manipulado por los comandos T_STOP y TJSTEP. Hay dos formas de establecer puntos interruptores: usando el comando T_SETINS para reemplazar una instrucción por un código especial que genera una SIGTRAP, o bien empleando el comando T_SETUSER para modificar los registros especiales

de punto interruptor. En cualquier clase de sistema al cual se pueda trasladar MINIX seguramente será posible implementar un depurador valiéndose de técnicas similares, pero el traslado de estas funciones requiere estudiar el hardware de que se trata.

La mayor parte de los comandos ejecutados por dotrace o bien regresan o modifican valores en el espacio de texto o de datos, o en la entrada de tabla de procesos, del proceso rastreado, y el código es sencillo. Es peligroso permitir la alteración de ciertos registros y ciertos bits de las banderas de la CPU, así que el código incluye muchas verificaciones que manejan el comando T_SETUSER a fin de evitar tales operaciones.

Al final de system.c hay varios procedimientos de utilería que se emplean en diversos lugares del kernel. Cuando una tarea necesita causar una señal (p. ej., la tarea del reloj necesita causar una señal SIGALRM, o la tarea de la terminal necesita causar una señal SIGINT), invoca `cause_sig` (línea 15586). Este procedimiento enciende un bit en el campo `p_pending` de la entrada de tabla de procesos correspondiente al proceso destinatario de la señal y luego verifica si el administrador de memoria actualmente está esperando un mensaje de ANY, es decir, si está ocioso y esperando la siguiente petición de procesamiento. Si el administrador de memoria está ocioso, se invoca `inform` para decirle que se encargue de la señal.

`Inform` (línea 15627) se invoca sólo después de verificar que el administrador de memoria no está ocupado, como acaba de explicarse. Además de la llamada desde `cause_sig`, se le llama desde `mm_rec` (`enproc.c`) cada vez que el administrador de memoria se bloquea y hay señales del kernel pendientes. `Inform` construye un mensaje del tipo KSIG y lo envía al administrador de memoria. La tarea o proceso que invoca `cause_sig` continúa su ejecución tan pronto como el mensaje se copia en el buffer de recepción del administrador de memoria; no espera hasta que se ejecuta el administrador de memoria, como sería el caso si se utilizara el mecanismo de envío normal, que hace que el remitente se bloquee. Sin embargo, antes de regresar, `inform` invoca `lock_pick_proc`, que planifica el administrador de memoria para que se ejecute. Puesto que las tareas tienen una prioridad mayor que los servidores, el administrador de memoria no se ejecutará en tanto no se satisfagan todas las tareas. Cuando la tarea señalizadora termine, se ingresará en el planificador. Si el administrador de memoria es el proceso ejecutable con la más alta prioridad, se ejecutará y procesará la señal.

El procedimiento `umap` (línea 15658) es de utilidad general y establece la correspondencia entre una dirección virtual y una física. Como hemos señalado, `do_map`, que da servicio al mensaje `SYS_UMAP`, invoca este procedimiento. Sus parámetros son un apuntador a la entrada de tabla de procesos del proceso o tarea con cuyo espacio de direcciones virtuales se establecerá la correspondencia, una bandera que especifica el segmento de texto, de datos o de pila, la dirección virtual misma y una cuenta de bytes. Esta última es útil porque `umap` verifica que todo el buffer que comienza en la dirección virtual esté dentro del espacio de direcciones del proceso. Para ello, `umap` debe conocer el tamaño del buffer. La cuenta de bytes no se utiliza para el mapeo sí, sólo para esta verificación. Todas las tareas que copian datos de o hacia el espacio de usuarios calculan la dirección física del buffer usando `umap`. En el caso de los controladores de dispositivos, resulta práctico poder obtener los servicios de `umap` a partir del número del proceso en lugar de un apuntador a una entrada de la tabla de procesos. `Numap` (línea 15697) se encarga de esto invocando `proc_addr` para convertir su primer argumento e invocando después `umap`.

La última función definida en `system.c` es `alloc_segments` (línea 15715), la cual es invocada por `do_newmap`, así como por la rutina `main` del kernel durante la inicialización. Esta definición depende en gran medida del hardware; recibe las asignaciones de segmentos que están registrada en una entrada de la tabla de procesos y manipula los registros y descriptores que el procesador Pentium usa para apoyar los segmentos protegidos en el nivel de hardware.

3.11 RESUMEN

La entrada/salida es un tema importante que muchas veces se descuida. Una fracción sustancial de todo sistema operativo se ocupa de E/S. Comenzamos por examinar el hardware de E/S y 1, relación entre los dispositivos de E/S y los controladores en hardware de E/S, que son los que deben manejar el software. Luego vimos los cuatro niveles de software de E/S: las rutinas de interrupción, los controladores de dispositivos, el software de E/S independiente del dispositivo y las bibliotecas de E/S y los spoolers que se ejecutan en el espacio de usuario.

A continuación estudiamos el problema del bloqueo mutuo y la forma de enfrentarlo. Ocurre un bloqueo mutuo cuando se concede a cada uno de un grupo de procesos acceso exclusivo a ciertos recursos, y cada uno quiere además otro recurso que pertenece a otro proceso del grupo. Todo ellos se bloquean y ninguno puede ejecutarse jamás. Se puede evitar el bloqueo mutuo estructurando el sistema de modo que nunca pueda ocurrir, por ejemplo, decretando que un proceso sólo puede adueñarse de un recurso a la vez. También puede evitarse el bloqueo mutuo examinando todas las peticiones de recursos para determinar si dan pie a una situación en la que es posible el bloqueo mutuo (un estado inseguro) y rechazar o diferir las peticiones que pudieran causar problemas.

Los controladores de dispositivos de MINIX se implementan como procesos incorporados en el kernel. Examinamos el controlador del disco en RAM, del disco duro, del reloj y de la terminal. La tarea de alarma sincrónica y la tarea del sistema no son controladores de dispositivos pero su estructura es muy similar. Cada una de estas tareas tiene un ciclo principal que obtiene peticiones y las procesa, devolviendo tarde o temprano respuestas para informar qué sucedió. Todas las tareas están ubicadas en el mismo espacio de direcciones. Las tareas controladoras del disco en RAM, el disco duro y el disquete utilizan una sola copia del mismo ciclo principal y también comparten funciones. No obstante, cada una es un proceso independiente. Una sola tarea de terminal apoya varias terminales diferentes, usando la consola del sistema, las líneas en serie y las conexiones de red.

Los controladores de dispositivos tienen relaciones variables con el sistema de interrupciones. Los dispositivos que pueden completar su trabajo rápidamente, como el disco en RAM y la pantalla mapeados en la memoria, ni siquiera usan interrupciones. La tarea controladora del disco duro realiza la mayor parte de su trabajo en el código de la tarea mismo, y los manejadores de interrupciones se limitan a devolver información de estado. El manejador de interrupciones del reloj realiza varias operaciones de contabilidad por sí mismo y sólo envía un mensaje a la tarea del reloj cuando hay trabajo del que el manejador no se puede ocupar. El manejador de interrupciones del teclado guarda en buffers las entradas y nunca envía mensajes a su tarea; más bien, modifica una variable inspeccionada por el manejador de interrupciones del reloj, el cual envía posteriormente un mensaje a la tarea de la terminal.

PROBLEMAS

1. Imagine que los avances en la tecnología de los chips hacen posible colocar todo un controlador en hardware, incluida toda la lógica de acceso a buses, en un chip de bajo costo. ¿Cómo afectaría eso el modelo de la Fig. 3-1?

2. Si un controlador en hardware de disco escribe en la memoria los bytes que recibe del disco tan rápidamente como los recibe, sin usar buffers internos, ¿tendría alguna utilidad concebible la intercalación? Explique.

3. Con base en la velocidad de rotación y la geometría de los discos, determine las tasas de bits para las transferencias entre el disco mismo y el buffer de controlador en hardware para un disco flexible y para un disco duro. Compare estas tasas con las de otras formas de E/S (líneas en serie y redes).

4. Un disco tiene doble intercalación, como en la Fig. 3-4(c), con ocho sectores de 512 bytes en cada pista, y una velocidad de rotación de 300 rpm. ¿Cuánto tiempo toma leer todos los sectores de una pista en orden, suponiendo que el brazo ya está en la posición correcta y que se necesita media rotación para lograr que el sector 0 esté bajo la cabeza? Calcule la tasa de datos. Repita ahora el problema para un disco sin intercalación con las mismas características. ¿Cuánto se degrada la tasa de datos a causa de la intercalación?

5. El multiplexor de terminales DM-11, que se usó en la PDP-11 hace muchos, muchos años, muestreaba cada línea de terminal (semidúplex) a siete veces la tasa en bauds para ver si el bit entrante era un 0 o un 1. El muestreo de la línea tomaba 5.7 microsegundos. ¿Cuántas líneas de 1200 bauds podía manejar el DM-II?

6. Una red de área local se usa como sigue. El usuario emite una llamada al sistema para escribir paquetes de datos en la red. A continuación, el sistema operativo copia los datos en un buffer del kernel, y luego los copia en la tarjeta del controlador en hardware de la red. Una vez que todos los bytes están seguros dentro del controlador, se envían por la red a razón de 10 megabits/s. El controlador de red receptor almacena cada bit un microsegundo después de que se envía. Cuando llega el último bit, la CPU de destino se interrumpe, y el kernel copia el paquete recién llegado en un buffer del kernel para inspeccionarlo. Una vez que el kernel ha determinado a qué usuario va dirigido el paquete, copia los datos en el espacio del usuario. Si suponemos que cada interrupción y su procesamiento toma 1 microsegundo, que los paquetes tienen 1024 bytes (no tome en cuenta las cabeceras) y que se requiere 1 microsegundo para copiar un byte, ¿con qué velocidad máxima puede un proceso enviar datos a otro? Suponga que el transmisor se bloquea hasta que el procesamiento termina en el lado del receptor y regresa un acuse de recibo. Por sencillez, suponga que el tiempo que toma recibir el acuse de recibo es tan pequeño que puede ignorarse.

7. ¿Qué significa "independencia del dispositivo"?

8. ¿En cuál de las cuatro capas de software de E/S se realiza cada una de las siguientes actividades?

- (a) Calcular la pista, sector y cabeza para una lectura de disco.
- (b) Mantener un caché de los bloques recientemente utilizados.
- (c) Escribir comandos en los registros de los dispositivos.
- (d) Verificar que el usuario tenga permiso de usar el dispositivo.
- (e) Convertir enteros binarios a ASCII para imprimirlos.

9. ¿Por qué los archivos de salida para la impresora normalmente se colocan en un spool de disco antes de imprimirse?

- 10.** Considere la Fig. 3-8. Suponga que en el paso (o) C solicitara S en lugar de solicitar R. ¿Daría lugar esto a un bloqueo mutuo? ¿Y si solicitara tanto S como R?
- 11.** Examine con detenimiento la Fig. 3-11(b). Si Susana pide una unidad más ¿lleva esto a un estado seguro o a uno inseguro? ¿Y si la petición proviniera de Miguel en lugar de Susana?
- 12.** Todas las trayectorias de la Fig. 3-12 son horizontales o verticales. ¿Puede usted imaginar una circunstancia en la que también pueda haber trayectorias diagonales?
- 13.** Suponga que el proceso A de la Fig. 3-13 solicita la última unidad de cinta. ¿Esta acción da pie a un bloqueo mutuo?
- 14.** Una computadora tiene seis unidades de cinta, y n procesos compiten por ellas. Cada proceso puede necesitar dos unidades. ¿Con qué valores de n el sistema está libre de bloqueos mutuos?
- 15.** ¿Puede un sistema estar en un estado que no es un bloqueo mutuo ni un estado seguro? De ser así, cite un ejemplo. Si no, demuestre que todos los estados son bloqueos mutuos o bien estados seguros.
- 16.** Un sistema distribuido que emplea buzones tiene dos primitivas IPC, SEND y RECEIVE. La segunda primitiva especifica el proceso del que se desea recibir, y se bloquea si no hay un mensaje disponible de ese proceso, aunque haya mensajes esperando de otros procesos. No hay recursos compartidos, pero los procesos necesitan comunicarse con frecuencia respecto a otros asuntos. ¿Son posibles los bloqueos mutuos? Explique.
- 17.** En un sistema de transferencia electrónica de fondos hay cientos de procesos idénticos que funcionan como sigue. Cada proceso lee una línea de entrada que especifica una cantidad de dinero, la cuenta a la que debe abonarse y la cuenta a la que debe cargarse. Luego el proceso pone un candado a ambas cuentas y transfiere el dinero, liberando los candados cuando termina. Si hay muchos procesos ejecutándose en paralelo, existe el peligro, muy real, de que habiendo puesto un candado a la cuenta x , un proceso no pueda poner un candado a y porque otro proceso ya le puso un candado y ahora está esperando para poner un candado a x . Invente un esquema que evite los bloqueos mutuos. No libere el registro de una cuenta antes de haber completado las transacciones. (Dicho de otro modo, no se permiten soluciones que ponen un candado a una cuenta y luego lo liberan de inmediato si la otra tiene un candado.)
- 18.** Se está usando el algoritmo del banquero en un sistema con w clases de recursos y n procesos. En el límite de m y n grandes, el número de operaciones que deben realizarse para verificar la seguridad de un estado es proporcional a $m^a n^b$. ¿Qué valores tienen a y b ?
- 19.** Cenicienta y el Príncipe se están divorciando. Para dividir sus posesiones, han acordado el siguiente algoritmo. Cada mañana, cada cónyuge puede enviar una carta al abogado del otro solicitando uno de sus bienes. Puesto que la entrega de una carta tarda un día, han convenido en que, si ambos descubren que han solicitado la misma cosa el mismo día, al día siguiente enviarán una carta cancelando la reclamación. Entre sus bienes está su perro, Woof*, la perrera de Woof, su canario, Tweeter, y la jaula de Tweeter. Los animales aman sus hogares, así que se ha acordado que cualquier división de bienes que separe un animal de su casa no es válida, y requiere que toda la división se reinicie desde el principio. Tanto Cenicienta como el Príncipe desean desesperadamente quedarse con Woof. A fin de poder irse de vacaciones (por separado), cada uno de los cónyuges ha programado una computadora personal que se encargue de la negociación. Al regresar de sus vacaciones, las computadoras siguen negociando.

* Como nombres de los animales se han elegido Woof y Tweeter, términos con los que, en inglés, se designa los parlantes de un sistema de estéreo; woof significa "altavoz para frecuencias bajas" y tweeter "altavoz para altas frecuencias". (N. del autor, A. S. W.)

¿Por qué? ¿Puede haberse creado un bloqueo mutuo? ¿Es posible que se haya creado inanición (una espera eterna)? Explique.

20. Se emplea el formato de mensajes de la Fig. 3-15 para enviar mensajes de petición a los controladores en software de dispositivos por bloques. ¿Cuáles campos, si acaso, podrían omitirse en los mensajes destinados a dispositivos por caracteres?

21. Un controlador en software de disco duro recibe peticiones para los cilindros 10, 22, 20, 2, 40, 6 y 38, en ese orden. El brazo tarda 6 ms en moverse de un cilindro al siguiente. ¿Cuánto tiempo de búsqueda se requiere para que

- (a) El primero que llegue, primero que se atienda?
- (b) Siga el cilindro más cercano?
- (c) Se aplique el algoritmo del elevador (moviéndose inicialmente hacia arriba)?

En todos los casos, el brazo está inicialmente en el cilindro 20.

22. Un vendedor de computadoras personales, en una visita a una universidad del suroeste de Amsterdam, comentó durante su "discurso" de venta que su compañía se había esforzado mucho por hacer que su versión de UNIX fuera muy rápida. Como ejemplo, citó que su controlador en software de disco usaba el algoritmo del elevador y también ponía en cola múltiples peticiones para el mismo cilindro en orden por sector. Un estudiante, Harry Hacker, quedó impresionado y compró una computadora, la llevó a casa, y escribió un programa que leía directamente 10 000 bloques dispersos por todo el disco. Para su asombro, el rendimiento que midió fue idéntico al que se esperaría si se utilizara el régimen de primero que llega, primero que se atiende. ¿Estaba mintiendo el vendedor?

23. Un proceso UNIX tiene dos partes, la parte del usuario y la parte del kernel. ¿La parte del kernel es como una subrutina o como una corrutina?

24. El manejador de interrupciones de reloj de cierta computadora requiere 2 ms (incluido el gasto extra de conmutación de procesos) por tic del reloj. El reloj opera a 60 Hz. ¿Qué fracción de la CPU está dedicada al reloj?

25. En el texto se dieron dos ejemplos de temporizador de vigilancia: temporizar el arranque del motor del disco flexible y dar tiempo para el retomo de carro en las terminales de copia impresa. Cite un tercer ejemplo.

26. Las terminales RS-232 se controlan por interrupciones, no así las terminales mapeadas en la memoria. ¿Por qué?

27. Considere el funcionamiento de una terminal. El controlador envía a la salida un carácter y luego se bloquea. Una vez que se ha exhibido el carácter, ocurre una interrupción y se envía un mensaje al controlador bloqueado, que envía a la salida el siguiente carácter y se bloquea otra vez. Si el tiempo necesario para pasar un mensaje, enviar un carácter a la salida y bloquearse es de 4 ms, ¿funciona bien este método en líneas de 110 bauds? ¿Y en líneas de 4800 bauds?

28. Una terminal de mapa de bits contiene 1200 por 800 píxeles. Para desplazar una ventana, la CPU (o el controlador en hardware) debe mover todas las líneas de texto hacia arriba copiando sus bits de una parte de la RAM de video a otra. Si una ventana dada tiene 66 líneas de altura y 80 caracteres de ancho (5280 caracteres en total) y el espacio que un carácter ocupa tiene 8 píxeles de ancho por 12 píxeles de altura, ¿qué tiempo tarda en desplazarse toda la ventana con una rapidez de copiado de 500 ns por byte? Si todas las líneas tienen 80 caracteres de longitud, calcule la tasa de datos equivalente de la terminal en bauds. Se requieren 50 microsegundos para colocar un carácter en la pantalla. Calcule ahora la tasa en bauds para la misma terminal en color, con 4 bits/píxel. (Ahora se requieren 200 microsegundos para colocar un carácter en la pantalla.)

- 29.** ¿Por qué los sistemas operativos proporcionan caracteres de escape, como CTRL-V en MINIX?
- 30.** Después de recibir un carácter DEL (SIGINT), el controlador de MINIX desecha todas las salidas que actualmente están en cola para esa terminal. ¿Por qué?
- 31.** Muchas terminales RS-232 tienen secuencias de escape para borrar la línea actual y subir una línea todas las líneas que están abajo de ella. ¿Cómo cree usted que esta capacidad se implemente dentro de la terminal?
- 32.** En la pantalla a color de la IBM PC original, la escritura en la RAM de video en cualquier momento que no fuera durante el retrazo vertical del haz del CRT causaba la aparición de feas manchas por toda la pantalla. Una imagen de pantalla tenía 25 por 80 caracteres, cada uno de los cuales cabía en un cuadro de 8 por 8 pixeles. Cada fila de 640 pixeles se dibujaba durante un solo barrido horizontal del haz, que tardaba 63.6 microsegundos, incluido el retrazo horizontal. La pantalla se redibujaba 60 veces por segundo, y en cada ocasión se requería un periodo de retrazo vertical para colocar el haz otra vez al principio de la pantalla. ¿Qué fracción del tiempo está la RAM de video disponible para escribir en ella?
- 33.** Escriba un controlador de gráficos para la pantalla IBM a color, o alguna otra pantalla de mapa de bits adecuada. El controlador deberá aceptar comandos para encender y apagar pixeles individuales, desplazar rectángulos dentro de la pantalla y cualesquier otras capacidades que le resulten interesantes. Los programas de usuario se comunican con el controlador abriendo `/dev/graphics` y escribiendo comandos en él.
- 34.** Modifique el controlador en software de disquete de MINIX de modo que guarde en caché una pista a la vez.
- 35.** Implemente un controlador de disco flexible que opere como dispositivo por caracteres, no de bloques, a fin de pasar por alto el caché de bloques del sistema de archivos. De este modo, los usuarios pueden leer grandes porciones de datos del disco, que se envían por DMA directamente al espacio del usuario, mejorando considerablemente el rendimiento. Este controlador sería de interés primordialmente para los programas que necesitan leer los bits en bruto del disco, sin tener en cuenta el sistema de archivos. Los verificadores de sistemas de archivos pertenecen a esta categoría.
- 36.** Implemente la llamada al sistema PROFIL de UNIX, que no está presente en MINIX.
- 37.** Modifique el controlador de la terminal de modo que, además de tener una tecla especial para borrar el carácter anterior, cuente con una tecla para borrar la palabra anterior.
- 38.** Se agregó a un sistema MINIX un nuevo dispositivo de disco duro con medios removibles. Este dispositivo debe acelerarse hasta adquirir la velocidad de operación cada vez que se cambian los medios, y el tiempo que toma esto es considerable. Se anticipa que los medios se cambiarán con frecuencia mientras el sistema está operando. Esto hace que la rutina waitfor de `at_wini.c` ya no sea satisfactoria. Diseñe una nueva rutina waitfor en la que, si el patrón de bits que se espera no se encuentra después de un segundo de espera activa, se ingrese en una fase en la que la tarea de disco dormirá durante 1 s, probará el puerto y se dormirá de nuevo durante otro segundo hasta que se encuentre el patrón buscado o expire el periodo de espera preestablecido TIMEOUT.

4

ADMINISTRACIÓN DE MEMORIA

La memoria es un recurso importante que se debe administrar con cuidado. Si bien hoy día la computadora casera media tiene cincuenta veces más memoria que la IBM 7094, que era la computadora más grande en el mundo a principios de los años sesenta, los programas están aumentando de tamaño con tanta rapidez como las memorias. Parafraseando la ley de Parkinson, "los programas se expanden hasta llenar la memoria disponible para contenerlos". En este capítulo estudiaremos la forma en que los sistemas operativos administran su memoria.

Idealmente, lo que a todo programador le gustaría es una memoria infinitamente grande y rápida que además no sea volátil, es decir, que no pierda su contenido cuando se interrumpa la alimentación eléctrica. Ya entrados en materia, ¿por qué no pedimos también que sea económica? Desafortunadamente, la tecnología no proporciona tales memorias. En consecuencia, la mayor parte de las computadoras tienen una jerarquía de memoria, con una cantidad pequeña de memoria caché muy rápida, costosa y volátil, algunos megabytes de memoria principal (RAM) volátil de mediana velocidad y mediano precio, y cientos o miles de megabytes de almacenamiento en disco lento, económico y no volátil. Corresponde al sistema operativo coordinar el uso de estas memorias.

La parte del sistema operativo que administra la jerarquía de memoria se denomina administrador de memoria. Su trabajo consiste en mantenerse al tanto de qué partes de la memoria están en uso y cuáles no lo están, asignar memoria a los procesos cuando la necesitan y recuperarla cuando terminan, y controlar el intercambio entre la memoria principal y el disco cuando la memoria es demasiado pequeña para contener todos los procesos.

En este capítulo investigaremos varios esquemas de manejo de memoria distintos, que van desde los muy sencillos hasta los muy avanzados. Comenzaremos por el principio y examinare-

mos primero el sistema de administración de memoria más sencillo posible, para avanzar gradualmente a sistemas cada vez más complicados.

4.1 ADMINISTRACIÓN BÁSICA DE MEMORIA

Los sistemas de administración de memoria se pueden dividir en dos clases, los que trasladan procesos entre la memoria y el disco durante la ejecución (intercambio y paginación) y los que no lo hacen. Estos últimos son más sencillos, así que los estudiaremos primero. Más adelante en el capítulo examinaremos el intercambio y la paginación. A lo largo de todo este capítulo, el lector debe tener presente que el intercambio y la paginación son en buena medida situaciones causadas por la falta de suficiente memoria principal para contener todos los programas a la vez. Al bajar el costo de la memoria principal, los argumentos a favor de un tipo de esquema de administración de memoria u otro pueden hacerse obsoletos, a menos que los programas crezcan con mayor rapidez que las memorias.

4.1.1 MONOPROGRAMACIÓN SIN INTERCAMBIO NI PAGINACIÓN

El esquema de administración de memoria más sencillo posible es ejecutar sólo un programa a la vez, compartiendo la memoria entre ese programa y el sistema operativo. En la Fig. 4-1 se muestran tres variaciones sobre este tema. El sistema operativo puede estar en la base de la memoria en RAM (memoria de acceso aleatorio), como se muestra en la Fig. 4-1 (a), o puede estar en ROM (memoria sólo de lectura) en la parte superior de la memoria, como en la Fig. 4-1(b), o los controladores de dispositivos pueden estar en la parte superior de la memoria en una ROM con el resto del sistema en RAM hasta abajo, como se muestra en la Fig. 4-1(c). Este último modelo es utilizado por los sistemas MS-DOS pequeños, por ejemplo. En las IBM PC, la porción del sistema que está en ROM se llama BIOS (Basic Input Output System, sistema básico de entrada salida).



Figura 4-1. Tres formas sencillas de organizar la memoria con un sistema operativo y un proceso de usuario. También existen otras posibilidades.

Si el sistema está organizado de esta manera, sólo puede ejecutarse un proceso a la vez. Tan pronto como el usuario teclea un comando, el sistema operativo copia el programa solicitador

disco a la memoria y lo ejecuta. Cuando el proceso termina, el sistema operativo exhibe un carácter de indicación y espera un nuevo comando. Cuando el sistema operativo lo recibe, carga un nuevo programa en la memoria, sobrescribiendo el primero.

4.1.2 Multiprogramación con particiones fijas

Aunque a veces se usa la monoprogamación en computadoras pequeñas con sistemas operativos sencillos, hay muchos casos en que es deseable permitir la ejecución de múltiples procesos a la vez. En los sistemas de tiempo compartido, tener varios procesos en la memoria a la vez implica que cuando un proceso está bloqueado esperando que termine una E/S, otro puede usar la CPU. Así, la multiprogramación aumenta el aprovechamiento de la CPU. Sin embargo, incluso en computadoras personales hay ocasiones en las que resulta útil poder ejecutar dos o más programas a la vez.

La forma más fácil de lograr la multiprogramación consiste simplemente en dividir la memoria en n particiones, posiblemente desiguales. Esta división puede, por ejemplo, efectuarse manualmente cuando se inicia el sistema.

Cuando llega un trabajo, se le puede colocar en la cola de entrada de la partición pequeña que puede contenerlo. Puesto que las particiones están fijas en este esquema, cualquier espacio de una partición que un trabajo no utilice se desperdiciará. En la Fig. 4-2(a) vemos el aspecto que tiene este sistema de particiones fijas y colas de entrada individuales.

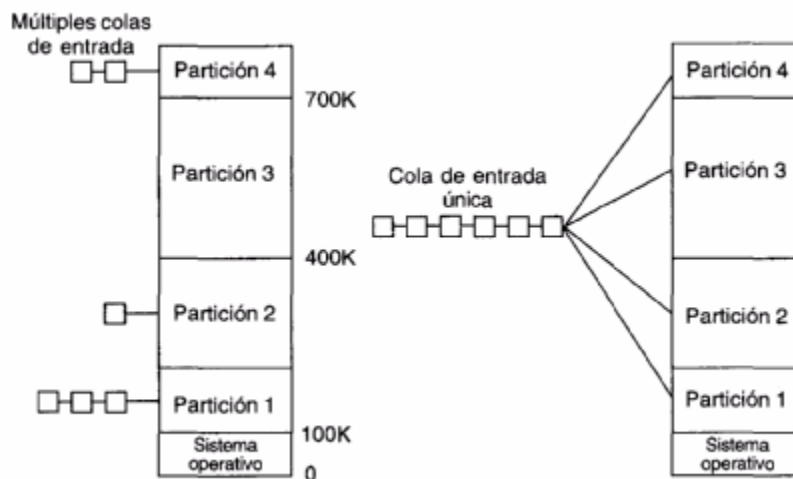


Figura 4-2. (a) Particiones de memoria fijas con colas de entrada individuales para cada partición, (b) Particiones de memoria fija con una sola cola de entrada.

La desventaja de repartir los trabajos entrantes en colas distintas se hace evidente cuando la cola de una partición grande está vacía pero la cola de una partición pequeña está llena, como es

el caso de las particiones 1 y 3 en la Fig. 4-2(a). Una organización alternativa sería mantener una sola cola, como en la Fig. 4-2(b). Cada vez que se libera una partición, se selecciona el trabajo más cercano a la cabeza de la cola que cabe en esa partición, se carga en dicha partición y ejecuta. Puesto que no es deseable desperdiciar una partición grande en un trabajo pequeño, una estrategia diferente consiste en examinar toda la cola de entrada cada vez que se libera una partición y escoger el trabajo más grande que cabe en ella. Observe que este último algoritmo discrimina contra los trabajos pequeños, considerándolos indignos de recibir toda la partición, en tanto que usualmente es deseable dar el mejor servicio, y no el peor, a los trabajos más pequeños (que supuestamente son interactivos).

Una salida consiste en tener por lo menos una partición pequeña disponible. Una partición a permitirá que se ejecuten los trabajos pequeños sin tener que asignar una partición grande para ello. Otro enfoque sería adoptar la regla de que un trabajo que es elegible para ejecutarse no puede pasarse por alto más de k veces. Cada vez que se pase por alto, el trabajo recibirá un punto, cuando haya adquirido k puntos, no se le podrá pasar por alto otra vez.

Este sistema, con particiones fijas establecidas por el operador en la mañana y que no se modificaban posteriormente, fue utilizado por OS/360 en macrocomputadoras de IBM durante muchos años. Se le llamaba MFT (multiprogramación con un número fijo de tareas, u OS/MFT). Este sistema es fácil de entender e igualmente sencillo de implementar: los trabajos entrantes se ponen en cola hasta que está disponible una partición apropiada. En ese momento, el trabajo se carga en esa partición y se ejecuta hasta terminar. Hoy día son pocos los sistemas operativos que dan soporte a este modelo, si es que todavía existe alguno.

Relocalización y protección

La multiprogramación introduce dos problemas esenciales que es preciso resolver: la relocalización y la protección. Examinemos la Fig. 4-2. Es evidente que diferentes trabajos se ejecutarán en diferentes direcciones. Cuando se vincula un programa (es decir, cuando el programa principal, los procedimientos escritos por el usuario y los procedimientos de biblioteca se combinan en un solo espacio de direcciones), el vinculador necesita saber en qué dirección de la memoria va a comenzar el programa.

Por ejemplo, supongamos que la primera instrucción es una llamada a un procedimiento que está en la dirección absoluta 100 dentro del archivo binario producido por el vinculador. Si este programa se carga en la partición 1, esa instrucción saltará a la dirección absoluta 100, que está, dentro del sistema operativo. Lo que se necesita es una llamada a $100K + 100$. Si el programa se le carga en la partición 2, deberá efectuarse como una llamada a $200K + 100$, etc. Este problema se denomina problema de relocalización.

Una posible solución es modificar realmente las instrucciones en el momento en que el programa se carga en la memoria. Se suma $100K$ a cada una de las direcciones de un programa que se carga en la partición 1, se suma $200K$ a las direcciones de los programas cargados en la partición 2, etc. A fin de efectuar la relocalización durante la carga de esta manera, el vinculador debe incluir en el programa binario una lista o mapa de bits que indique cuáles palabras del programa son direcciones que deben relocalizarse y cuáles con códigos de operación, constantes

u otros elementos que no deben relocarse. OS/MFT funcionaba de este modo. Algunas microcomputadoras también trabajan así.

La relocación durante la carga no resuelve el problema de la protección. Un programa mal intencionado siempre puede construir una nueva instrucción y saltar a ella. Dado que los programas en este sistema usan direcciones de memoria absolutas en lugar de direcciones relativas a un registro, no hay forma de impedir que un programa construya una instrucción que lea o escriba cualquier palabra de la memoria. En los sistemas multiusuario, no es conveniente permitir que los procesos lean y escriban en memoria que pertenece a otros usuarios.

La solución que IBM escogió para proteger el 360 fue dividir la memoria en bloques de 2K bytes y asignar un código de protección de cuatro bits a cada bloque. La PSW contenía una clave de cuatro bits. El hardware del 360 atrapaba cualquier intento, por parte de un proceso en ejecución, de acceder a memoria cuyo código de protección difería de la clave de la PSW. Puesto que sólo el sistema operativo podía cambiar los códigos de protección y la clave, los procesos de usuario no podían interferirse ni interferir el sistema operativo mismo.

Una solución alternativa tanto al problema de relocación como al de protección consiste en equipar la máquina con dos registros especiales en hardware, llamados registros de base y de límite. Cuando se calendariza un proceso, el registro de base se carga con la dirección del principio de su partición, y el registro de límite se carga con la longitud de la partición. A cada dirección de memoria generada se le suma automáticamente el contenido del registro de base antes de enviarse a la memoria. De este modo, si el registro de base es de 100K, una instrucción CALL 100 se convierte efectivamente en una instrucción CALL 100K + 100, sin que la instrucción en sí se modifique. Además, las direcciones se cotejan con el registro de límite para asegurar que no intenten acceder a memoria fuera de la partición actual. El hardware protege los registros de base y de límite para evitar que los programas de usuario los modifiquen.

La CDC 6600 —la primera supercomputadora que hubo en el mundo— utilizaba este esquema. La CPU Intel 8088 empleada para la IBM PC original utilizaba una versión más débil de este esquema: registros de base, pero sin registros de límite. A partir de la 286, se adoptó un mejor esquema.

4.2 INTERCAMBIO

En un sistema por lotes, la organización de la memoria en particiones fijas es sencilla y efectiva. Cada trabajo se carga en una partición cuando llega al frente de la cola, y permanece en la memoria hasta terminar. En tanto sea posible mantener en la memoria suficientes trabajos para mantener ocupada a la CPU todo el tiempo, no hay por qué usar algo más complicado.

En los sistemas de tiempo compartido o las computadoras personales orientadas a gráficos, la situación es diferente. A veces no hay bastante memoria principal para contener todos los procesos que están activos actualmente, y los procesos en exceso deben mantenerse en disco y traerse dinámicamente para que se ejecuten.

Podemos usar dos enfoques de administración de memoria generales, dependiendo (en parte) del hardware disponible. La estrategia más sencilla, llamada intercambio, consiste en traer a la

memoria cada proceso en su totalidad, ejecutarlo durante un tiempo, y después colocarlo otra vez en el disco. La otra estrategia, llamada memoria virtual, permite a los programas ejecutarse aunque sólo estén parcialmente en la memoria principal. A continuación estudiaremos el intercambio; en la sección 4.3 examinaremos la memoria virtual.

El funcionamiento de un sistema con intercambio se ilustra en la Fig. 4-3. Inicialmente, sólo el proceso A está en la memoria. Luego se crean o se traen del disco los procesos B y C. En la Fig. 4-3(d) A termina o se intercambia al disco. Luego llega D y B sale. Por último, entra E.

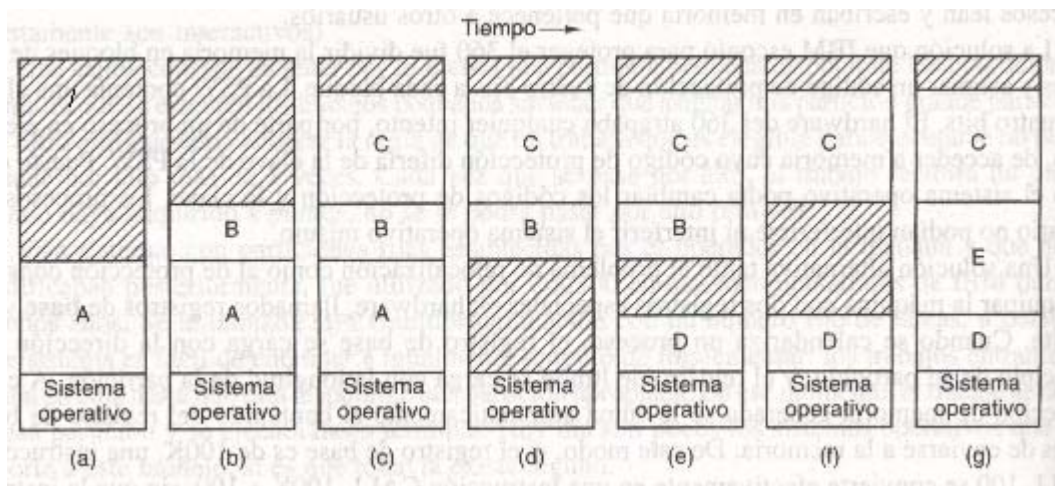


Figura 4-3. La asignación de memoria cambia conforme los procesos entran en la memoria y salen de ella. Las regiones sombreadas son memoria no utilizada.

La diferencia principal entre las particiones fijas de la Fig. 4-2 y las particiones variables de la Fig. 4-3 es que el número, ubicación y tamaño de las particiones varían dinámicamente en el segundo caso conforme los procesos vienen y van, mientras que en el primer caso están fijas. La flexibilidad de no estar atado a un número fijo de particiones que podrían ser demasiado grandes o demasiado pequeñas mejora el aprovechamiento de la memoria, pero también complica la asignación y liberación de memoria, así como su contabilización.

Si el intercambio crea múltiples agujeros en la memoria, es posible combinarlos todos para formar uno grande desplazando todos los procesos hacia abajo hasta donde sea posible. Esta técnica se conoce como compactación de memoria, y pocas veces se practica porque requiere mucho tiempo de CPU. Por ejemplo, en una máquina de 32MB que puede copiar 16 bytes por microsegundo tomaría 2 segundos compactar toda la memoria.

Un punto que cabe destacar se refiere a la cantidad de memoria que debe asignarse a un proceso cuando se crea o se trae a la memoria. Si los procesos se crean con un tamaño fijo que nunca cambia, la asignación es sencilla: se asigna exactamente lo que se necesita, ni más ni menos.

En cambio, si los segmentos de datos de los procesos pueden crecer, por ejemplo, mediante asignación dinámica de memoria desde un montículo, como en muchos lenguajes de programación, ocurrirá un problema cada vez que un proceso trate de crecer. Si hay un agujero adyacente

al proceso, se le puede asignar y el proceso podrá crecer hacia el agujero. Por otro lado, si el proceso está adyacente a otro proceso, el proceso en crecimiento tendrá que ser transferido a un agujero en la memoria que tenga el tamaño suficiente para contenerlo, o bien uno o más procesos tendrán que ser intercambiados a disco para crear un agujero del tamaño necesario. Si un proceso no puede crecer en la memoria, y el área de intercambio en el disco está llena, el proceso tendrá que esperar o morir.

Si se espera que la mayor parte de los procesos crezcan durante su ejecución, probablemente será conveniente asignar un poco de memoria adicional cada vez que se traiga un proceso a la memoria o se le cambie de lugar, a fin de reducir el gasto extra asociado al traslado o intercambio de procesos que ya no caben en la memoria que se les asignó. Sin embargo, al intercambiar los procesos al disco, sólo debe intercambiarse la memoria que se está utilizando realmente; sería un desperdicio intercambiar también la memoria adicional. En la Fig. 4-4(a) vemos una configuración de memoria en la que se asignó a dos procesos espacio para crecer.

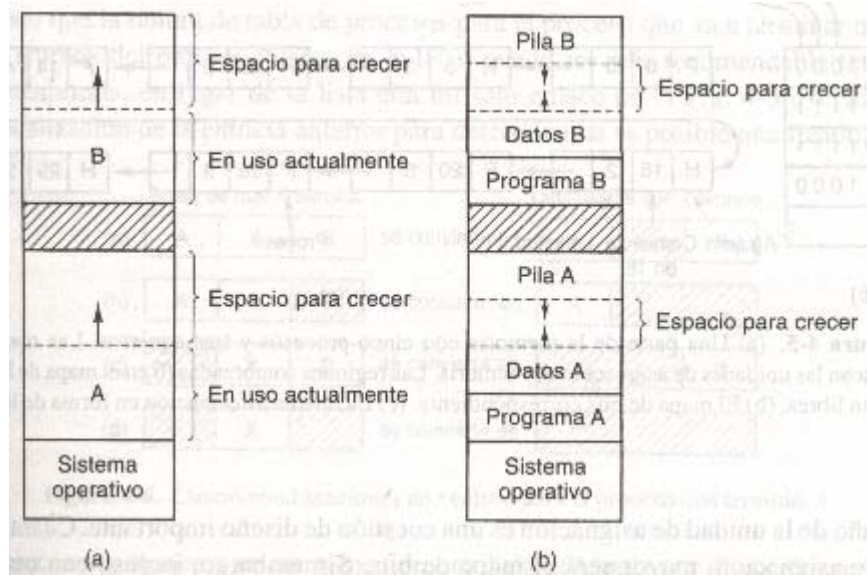


Figura 4-4. (a) Asignación de espacio para un segmento de datos que crece, (b) Asignación de espacio para una pila que crece y un segmento de datos que crece.

Si los procesos pueden tener dos procesos en crecimiento, por ejemplo, el segmento de datos que se está usando como montículo para variables que se asignan y liberan dinámicamente, y un segmento de pila para las variables locales normales y las direcciones de retomo, podemos tener una organización alternativa, que se ilustra en la Fig. 4-4(b). En esta figura vemos que cada proceso tiene una pila en la parte superior de su memoria asignada que está creciendo hacia abajo, y un segmento de datos justo después del texto del programa, creciendo hacia arriba. La memoria que está entre los dos segmentos se puede destinar a cualquiera de ellos. Si el espacio se agota, el proceso tendrá que ser transferido a un agujero con suficiente espacio, intercambiarse a disco hasta que pueda crearse un agujero del tamaño suficiente, o terminarse.

4.2.1 Administración de memoria con mapas de bits

Cuando la memoria se asigna dinámicamente, el sistema operativo debe administrarla. En términos generales, hay dos formas de contabilizar la utilización de memoria: mapas de bits y listas libres. En esta sección y en la siguiente examinaremos estos dos métodos por turno.

Con un mapa de bits, la memoria se divide en unidades de asignación, tal vez sólo de unas cuantas palabras o quizá de varios kilobytes. A cada unidad de asignación corresponde un bit del mapa de bits, que es 0 si la unidad está libre y 1 si está ocupada (o viceversa). En la Fig. 4-5 se muestra una parte de la memoria y el mapa de bits correspondiente.

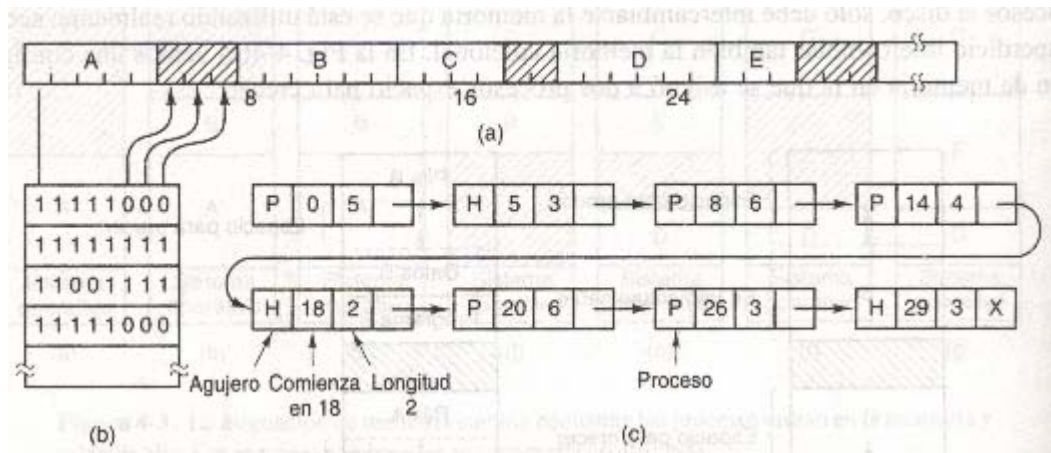


Figura 4-5. (a) Una parte de la memoria con cinco procesos y tres agujeros. Las marcas indican las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están libres, (b) El mapa de bits correspondiente, (c) La misma información en forma de lista.

El tamaño de la unidad de asignación es una cuestión de diseño importante. Cuanto menor sea la unidad de asignación, mayor será el mapa de bits. Sin embargo, incluso con una unidad de asignación de sólo cuatro bits, 32 bits de memoria sólo requerirán un bit del mapa. Una memoria de $32n$ bits usará n bits de mapa, y el mapa sólo ocupará $1/33$ de la memoria. Si se escoge una unidad de asignación grande, el mapa de bits será más pequeño, pero podría desperdiciarse una cantidad apreciable de memoria en la última unidad si el tamaño del proceso no es un múltiplo exacto de la unidad de asignación.

Un mapa de bits ofrece un método sencillo para contabilizar las palabras en una cantidad fija de memoria, porque el tamaño del mapa de bits depende sólo del tamaño de la memoria y del tamaño de la unidad de asignación. El problema principal que presenta es que una vez que se ha decidido traer a la memoria un proceso de k unidades, el administrador de memoria debe buscar en el mapa de bits una serie de k bits en O consecutivos. La búsqueda de series de una longitud dada en un mapa de bits es una operación lenta (porque la serie puede cruzar fronteras de palabra en el mapa); éste es un argumento en contra de los mapas de bits.

4.2.2 Administración de memoria con listas enlazadas

Otra forma de contabilizar la memoria es mantener una lista enlazada de segmentos de memoria libres y asignados, donde un segmento es un proceso o bien un agujero entre dos procesos. La memoria de la Fig. 4-5(a) se representa en la Fig. 4-5(c) como lista enlazada de segmentos. Cada entrada de la lista especifica un agujero (H) o un proceso (P), la dirección en la que principia, la longitud y un apuntador a la siguiente entrada.

En este ejemplo, la lista de segmentos se mantiene ordenada por dirección. Este ordenamiento tiene la ventaja de que cuando un proceso termina o es intercambiado a disco, es fácil actualizar la lista. Un proceso que termina normalmente tiene dos vecinos (excepto cuando está en el tope o la base de la memoria). Éstos pueden ser procesos o agujeros, dando lugar a las cuatro combinaciones de la Fig. 4-6. En la Fig. 4-6(a) la actualización de la lista requiere la sustitución de una P por una H. En las Figs. 4-6(b) y 4-6(c) dos entradas se funden para dar una sola, y el tamaño de la lista se reduce en una entrada. En la Fig. 4-6(d) tres entradas se funden y dos elementos se eliminan de la lista. Puesto que la ranura de tabla de procesos para el proceso que va a terminar normalmente apunta a la entrada del proceso mismo en la lista, puede ser más recomendable tener una lista doblemente enlazada, en lugar de la lista con un solo enlace de la Fig. 4-5(c). Esta estructura facilita la localización de la entrada anterior para determinar si es posible una fusión.

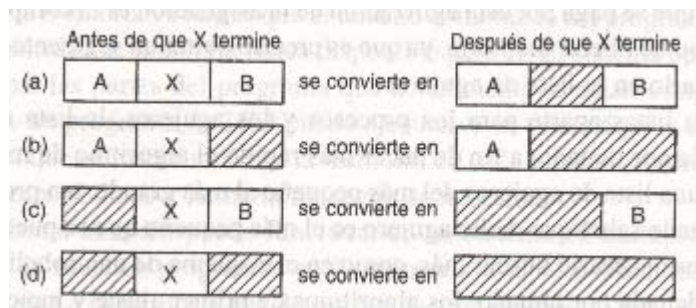


Figura 4-6. Cuatro combinaciones de vecinos para el proceso que termina, X.

Si los procesos y agujeros se mantienen en una lista ordenada por dirección, se pueden usar varios algoritmos para asignar memoria a un proceso recién creado o traído a la memoria. Suponemos que el administrador de memoria sabe cuánta memoria debe asignar. El algoritmo más sencillo es el de primer ajuste. El administrador de memoria examina la lista de segmentos hasta encontrar un agujero con el tamaño suficiente. A continuación, el agujero se divide en dos fragmentos, uno para el proceso y otro para la memoria desocupada, excepto en el poco probable caso de que el ajuste sea exacto. El algoritmo de primer ajuste es rápido porque la búsqueda es la más corta posible.

Una variante menor del primer ajuste es el siguiente ajuste. Este algoritmo funciona igual que el de primer ajuste, excepto que toma nota de dónde está cada vez que encuentra un agujero apropiado. La siguiente vez que se invoque, el algoritmo comenzará a buscar en la lista a partir del lugar donde se quedó la última vez, en lugar de comenzar por el principio, como hace el primer ajuste. Simulaciones ejecutadas por Bays (1977) demuestran que el siguiente ajuste ofrece un rendimiento ligeramente peor que el primer ajuste.

Otro algoritmo bien conocido es el de mejor ajuste, que examina toda la lista y toma el agujero más pequeño que es adecuado. En lugar de partir un agujero grande que podría necesitarse después, el mejor ajuste trata de encontrar un agujero con un tamaño cercano al que se necesita.

Como ejemplo de primer ajuste y mejor ajuste, consideremos otra vez la Fig. 4-5. Si se requiere un bloque de tamaño 2, el primer ajuste asignará el agujero que está en 5, pero el mejor ajuste asignará el agujero que está en 18.

El mejor ajuste es más lento que el primer ajuste porque debe examinar toda la lista cada vez que se invoca. Lo que resulta sorprendente es que también desperdicia más memoria que el primer ajuste o el siguiente ajuste porque tiende a llenar la memoria de pequeños agujeros inútiles. En promedio, el primer ajuste genera agujeros más grandes.

A fin de sortear el problema de partir un agujero con un tamaño casi igual al requerido para obtener un área asignada al proceso y un agujero diminuto, podríamos considerar el peor ajuste, es decir, tomar siempre el agujero más grande disponible, de modo que el agujero sobrante tenga un tamaño suficiente para ser útil. Las simulaciones han demostrado que el peor ajuste tampoco es una idea muy buena.

Los cuatro algoritmos pueden agilizarse manteniendo listas separadas de procesos y agujeros. De este modo, los algoritmos dedican toda su energía a inspeccionar agujeros, no procesos. El inevitable precio que se paga por esta agilización de la asignación es la complejidad adicional y la lentitud del proceso de liberar memoria, ya que es preciso quitar un segmento liberado de la lista de procesos e insertarlo en la lista de agujeros.

Si se mantienen listas aparte para los procesos y los agujeros, la lista de agujeros puede mantenerse ordenada por tamaño, a fin de hacer más rápido el algoritmo de mejor ajuste. Si este algoritmo examina una lista de agujeros del más pequeño al más grande, tan pronto como encuentre un agujero adecuado sabrá que dicho agujero es el más pequeño que se puede usar, y por tanto el mejor ajuste. No es necesario buscar más, como en el esquema de una sola lista. Si se tiene una lista de agujeros ordenada por tamaño, los algoritmos de primer ajuste y mejor ajuste son igualmente rápidos, y el de siguiente ajuste no tiene caso.

Si los agujeros se mantienen en listas aparte de las de procesos, es posible una pequeña optimización. En lugar de tener un conjunto aparte de estructuras de datos para mantener la lista de agujeros, como se hace en la Fig. 4-5(c), se pueden usar los agujeros mismos. La primera palabra de cada agujero podría ser el tamaño del agujero, y la segunda, un apuntador a la siguiente entrada. Los nodos de la lista de la Fig. 4-5(c), que requieren tres palabras y un bit (P/H), ya no son necesarios.

Otro algoritmo de asignación más es el de ajuste rápido, que mantiene listas por separado para algunos de los tamaños más comunes solicitados. Por ejemplo, se podría tener una tabla con n entradas, en la que la primera entrada es un apuntador a una lista de agujeros de 4K, la segunda entrada es un apuntador a una lista de agujeros de 8K, la tercera es un apuntador a una lista de agujeros de 12K, y así sucesivamente. Los agujeros de, digamos, 21K, podrían colocarse en la lista de 20K o en una lista especial de agujeros con tamaños poco comunes. Si se usa el ajuste rápido, la localización de un agujero del tamaño requerido es extremadamente rápida, pero se tiene la misma desventaja que tienen todos los esquemas que ordenan por tamaño de agujero, a saber, que cuando un proceso termina o es intercambiado a disco la localización de sus vecinos

determinar si es posible una fusión es costosa. Si no se efectúan fusiones, la memoria pronto se fragmentará en un gran número de agujeros pequeños en los que no cabrá ningún proceso.

4.3 MEMORIA VIRTUAL

Hace muchos años las personas enfrentaron por primera vez programas que eran demasiado grandes para caber en la memoria disponible. La solución que normalmente se adoptaba era dividir el programa en fragmentos, llamados superposiciones. La superposición O era la primera que se ejecutaba. Al terminar, esta superposición llamaba a otra. Algunos sistemas de superposición eran muy complejos, pues permitían varias superposiciones en la memoria a la vez. Las superposiciones se mantenían en disco y el sistema operativo las intercambiaba con la memoria dinámicamente, según fuera necesario.

Aunque el trabajo real de intercambiar las superposiciones corría por cuenta del sistema, la tarea de dividir el programa en fragmentos tenía que ser efectuada por el programador. La división de programas grandes en fragmentos modulares pequeños consumía tiempo y era tediosa. No pasó mucho tiempo antes de que a alguien se le ocurriera una forma de dejar todo el trabajo a la computadora.

El método que se inventó (Fotheringham, 1961) se conoce ahora como memoria virtual. La idea en que se basa la memoria virtual es que el tamaño combinado del programa, los datos y la pila puede exceder la cantidad de memoria física disponible para él. El sistema operativo mantiene en la memoria principal las partes del programa que actualmente se están usando, y el resto en el disco. Por ejemplo, un programa de 16M puede ejecutarse en una máquina de 4M si se escogen con cuidado los 4M que se mantendrán en la memoria en cada instante, intercambiando segmentos del programa entre el disco y la memoria según se necesite.

La memoria virtual también puede funcionar en un sistema de multiprogramación, manteniendo segmentos de muchos programas en la memoria a la vez. Mientras un programa está esperando que se traiga a la memoria una de sus partes, está esperando E/S y no puede ejecutarse, así que puede otorgarse la CPU a otro proceso, lo mismo que en cualquier otro sistema de multiprogramación.

4.3.1 Paginación

La mayor parte de los sistemas de memoria virtual emplean una técnica llamada paginación, que describiremos a continuación. En cualquier computadora, existe un conjunto de direcciones de memoria que los programas pueden producir. Cuando un programa ejecuta una instrucción como

```
MOVE REG,1000
```

está copiando el contenido de la dirección de memoria 1000 en REG (o viceversa, dependiendo de la computadora). Las direcciones pueden generarse usando indización, registros de base, registros de segmento y otras técnicas.

Estas direcciones generadas por programas se denominan direcciones virtuales y constituyen el espacio de direcciones virtual. En las computadoras sin memoria virtual, la dirección

virtual se coloca directamente en el bus de memoria y hace que se lea o escriba la palabra de memoria física que tiene la misma dirección. Cuando se usa memoria virtual, las direcciones virtuales no pasan directamente al bus de memoria; en vez de ello, se envían a una unidad de administración de memoria (MMU), un chip o colección de chips que transforma las direcciones virtuales en direcciones de memoria física como se ilustra en la Fig. 4-7.

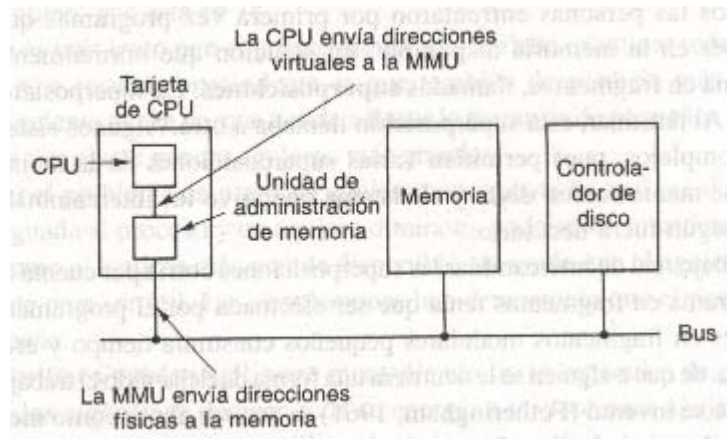


Figura 4-7. La posición y función de la MMU.

En la Fig. 4-8 se muestra un ejemplo muy sencillo de cómo funciona esta transformación. En este ejemplo, tenemos una computadora que puede generar direcciones de 16 bits, desde 0 hasta 64K. Éstas son las direcciones virtuales. Esta computadora, empero, sólo tiene 32K de memoria física, así que si bien es posible escribir programas de 64K, no pueden cargarse enteros en la memoria y ejecutarse. Sin embargo, debe estar presente en el disco una copia completa de la imagen de núcleo del programa, que puede ser de hasta 64K, para poder traer a la memoria fragmento de ella según sea necesario.

El espacio de direcciones virtual se divide en unidades llamadas páginas. Las unidades correspondientes en la memoria física se denominan marcos de página. Las páginas y los marcos de página siempre tienen exactamente el mismo tamaño. En este ejemplo, ese tamaño es 4K, pero es común usar tamaños de página desde 512 bytes hasta 64K en los sistemas existentes. Con 64K de espacio de direcciones virtual y 32K de memoria física, tenemos 16 páginas virtuales y ocho marcos de página. Las transferencias entre la memoria y el disco siempre se efectúan en unidades de una página.

Cuando el programa trata de acceder a la dirección 0, por ejemplo, usando la instrucción

```
MOVE REG,0
```

la dirección virtual 0 se envía a la MMU. La MMU ve que esta dirección virtual queda en la página 0 (0 a 4095) que, según su transformación, es el marco de página 2 (8192 a 12287). Por tanto, la MMU transforma la dirección a 8192 y la coloca en el bus. La tarjeta de memoria nada sabe de la MMU y simplemente ve una solicitud de leer o escribir en la dirección 8192, lo cual hace.

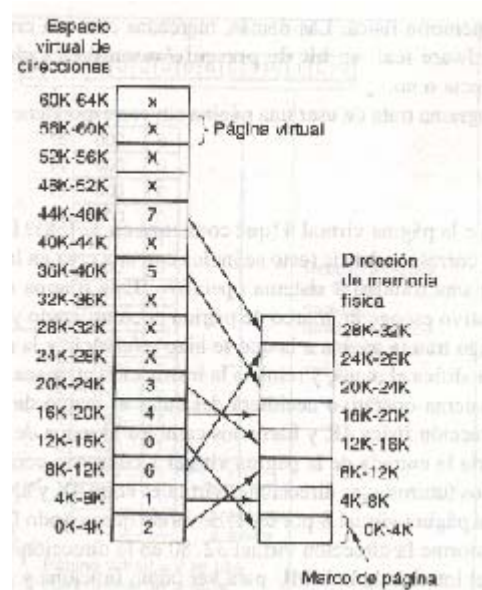


Figura 4-8. La relación entre las direcciones virtuales y las direcciones de la memoria física está dada por la tabla de páginas.

Así, la MMU ha transformado efectivamente todas las direcciones virtuales entre 0 y 4095 en las direcciones físicas 8192 a 12287.

De forma similar, una instrucción

```
MOVEREG,8192
```

se transforma efectivamente en

```
MOVE REG,24576
```

porque la dirección virtual 8192 está en la página virtual 2 y esta página corresponde al marco de página 6 (direcciones físicas 24576 a 28671). Como tercer ejemplo, la dirección virtual 20500 está 20 bytes después del inicio de la página virtual 5 (direcciones virtuales 20480 a 24575) y se transforma en la dirección física $12288 + 20 = 12308$.

En sí, esta capacidad para transformar las 16 páginas virtuales en cualquiera de los ocho marcos de página ajustando el mapa de la MMU de la forma apropiada no resuelve el problema de que el espacio de direcciones virtual sea más grande que la memoria física. Puesto que sólo tenemos ocho marcos de página físicos, sólo ocho de las páginas virtuales de la Fig. 4-8 tendrán

correspondencia con la memoria física. Las demás, marcadas con una cruz en la figura, no se transformarán. En el hardware real, un bit de presente/ausente en cada entrada indica si la página tiene correspondencia o no.

¿Qué sucede si el programa trata de usar una página sin correspondencia, por ejemplo, usando la instrucción

```
MOVE REG,32780
```

que es el byte 12 dentro de la página virtual 8 (que comienza en 32768)? La MMU se da cuenta de que la página no tiene correspondencia (esto se indica con una cruz en la figura) y hace que la CPU pase el control por una trampa al sistema operativo. Esta trampa se denomina falla de página. El sistema operativo escoge un marco de página poco utilizado y escribe su contenido de vuelta en el disco; luego trae la página a la que se hizo referencia y la coloca en el marco de página recién liberado, modifica el mapa, y reinicia la instrucción atrapada.

Por ejemplo, si el sistema operativo decidiera desalojar el marco de página 1, cargaría la página virtual 8 en la dirección física 4K y haría dos cambios al mapa de la MMU. Primero, el sistema operativo marcaría la entrada de la página virtual 1 como sin correspondencia, a fin de atrapar cualesquier accesos futuros a las direcciones virtuales entre 4K y 8K; luego, reemplazaría la cruz de la entrada de la página virtual 8 por un 1, de modo que cuando la instrucción atrapada se vuelva a ejecutar transforme la dirección virtual 32780 en la dirección física 4108.

Examinemos ahora el interior de la MMU para ver cómo funciona y por qué escogimos un tamaño de página que es una potencia de 2. En la Fig. 4-9 vemos un ejemplo de dirección virtual, 8196 (001000000000100 en binario), que se transforma usando el mapa de MMU de la Fig. 4-8. La dirección virtual de 16 bits entrante se divide en un número de página de cuatro bits y una distancia de 12 bits. Con cuatro bits para el número de página podemos representar 16 páginas, y con 12 bits para la distancia podemos direccionar los 4096 bytes contenidos en una página.

El número de página se utiliza como índice de la tabla de páginas, produciendo el número del marco de página que corresponde a esa página virtual. Si el bit Presente/ausente es 0, se genera una trampa al sistema operativo. Si el bit es 1, el número de marco de página que se encuentra en la tabla de páginas se copia en los tres bits de orden alto del registro de salida, junto con la distancia de 12 bits, que se copia sin modificación de la dirección virtual entrante. Juntas, estas dos partes forman una dirección física de 15 bits. A continuación el registro de salida se coloca en el bus de memoria como dirección de memoria física.

4.3.2 Tablas de páginas

En teoría, la transformación de direcciones virtuales en físicas se efectúa tal como lo hemos descrito. La dirección virtual se divide en un número de página virtual (bits de orden alto) y una distancia (bits de orden bajo). El número de página virtual sirve como índice para consultar la tabla de páginas y encontrar la entrada correspondiente a esa página virtual. En esa entrada se encuentra el número de marco de página, si lo hay, y este número se anexa al extremo de orden alto de la distancia, sustituyendo al número de página virtual y formando una dirección física que se puede enviar a la memoria.

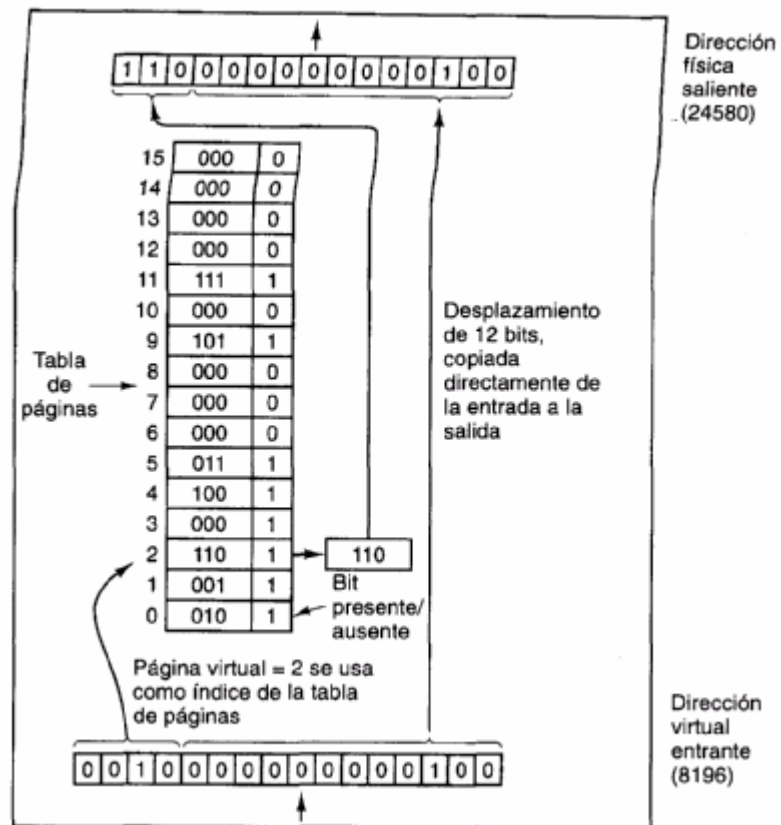


Figura 4-9. Funcionamiento interno de la MMU con 16 páginas de 4K.

El propósito de la tabla de páginas es transformar páginas virtuales en marcos de página. En términos matemáticos, la tabla de página es una función, con el número de página virtual como argumento y el número de marco físico como resultado. Usando el resultado de esta función, el campo de página virtual de una dirección virtual se puede sustituir por un campo de marco de página, formando así una dirección de memoria física.

A pesar de lo sencillo de esta descripción, hay que resolver dos problemas importantes:

1. La tabla de páginas puede ser extremadamente grande.
2. La transformación debe ser rápida.

El primer punto se sigue del hecho de que las computadoras modernas utilizan direcciones virtuales de por lo menos 32 bits. Con un tamaño de página de, digamos, 4K, un espacio de direcciones de 32 bits tiene un millón de páginas, y un espacio de direcciones de 64 bits tiene más de las que nos

podamos imaginar. Con un millón de páginas en el espacio de direcciones virtual, la tabla páginas debe tener un millón de entradas. Y recordemos que cada proceso necesita su propia tabla de páginas.

El segundo punto es consecuencia del hecho de que la transformación de virtual a física se debe efectuar en cada referencia a la memoria. Una instrucción típica tiene una palabra de instrucción, y con frecuencia también un operando en memoria. Por tanto, es necesario efectuar una, dos y a veces más referencias a la tabla de página por cada instrucción. Si una instrucción tarda, digamos, 10 ns, la búsqueda en la tabla de páginas debe efectuarse en unos cuantos nanosegundos para evitar que se convierta en un cuello de botella importante.

La necesidad de transformar páginas grandes rápidamente es una restricción significativa sobre la forma de construir computadoras. Aunque el problema es más grave en el caso de las máquinas de mayor capacidad, también es importante en el extremo inferior, donde el costo y la relación precio/rendimiento son críticos. En esta sección y en las siguientes examinaremos el diseño de tablas de páginas con detalle y presentaremos varias soluciones de hardware que se han empleado en computadoras reales.

El diseño más sencillo (al menos en lo conceptual) es tener una sola tabla de páginas que consiste en un arreglo de registros en hardware rápidos, con una entrada para página virtual, indizado por número de página virtual. Cuando se inicia un proceso, el sistema operativo carga los registros con la tabla de páginas del proceso, tomada de una copia que se mantiene en la memoria principal. Durante la ejecución del proceso, no se requieren más referencias a la memoria para la tabla de páginas. Las ventajas de este método es que es sencillo y no requiere referencias a la memoria durante la transformación. Una desventaja es que el costo puede ser alto (si la tabla de páginas es grande). Tener que cargar la tabla de páginas en cada conmutación de contexto también puede perjudicar el rendimiento.

En el otro extremo, la tabla de páginas puede estar toda en la memoria principal. En tal caso, todo lo que el hardware necesita es un solo registro que apunte al principio de la tabla. Este diseño permite modificar el mapa de memoria en una conmutación de contexto volviendo a cargar un registro. Desde luego, la desventaja es que se requiere una o más referencias a la memoria para leer las entradas de la tabla de página durante la ejecución de cada instrucción. Por esta razón este enfoque casi nunca se usa en su forma más pura, pero a continuación estudiaremos algunas variaciones que tienen un rendimiento muy superior.

Tablas de páginas multinivel

Con objeto de superar el problema de tener tablas de páginas enormes en la memoria todo el tiempo muchas computadoras usan una tabla de páginas multinivel. En la Fig. 4-10 se muestra un ejemplo sencillo. En la Fig. 4-10(a) tenemos una dirección virtual de 32 bits que se divide en un campo PT1 de 10 bits, un campo PT2 de 10 bits y un campo Offset de 12 bits. Puesto que las distancias o desplazamientos (offsets) tienen 12 bits, las páginas son de 4K y hay un total de 2^{20} de ellas.

El secreto del método de tabla de páginas multinivel es evitar mantener todas las tablas de páginas en la memoria todo el tiempo, en particular, las tablas que no se necesiten no deben estar ahí. Supongamos, por ejemplo, que un proceso necesita 12 megabytes, los cuatro megabytes

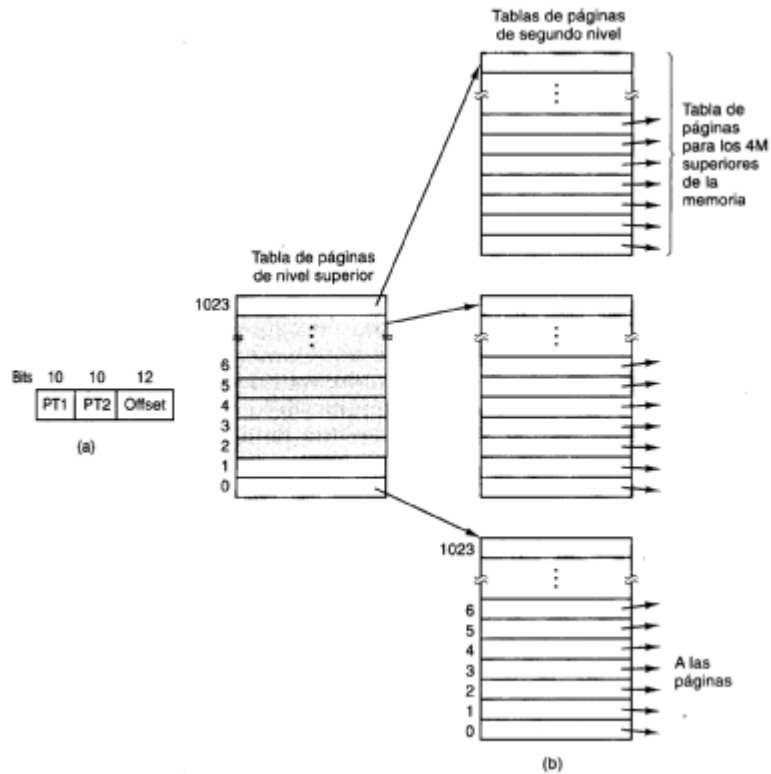


Figura 4-10. (a) Dirección de 32 bits con dos campos de tabla de páginas, (b) Tablas de páginas de dos niveles.

inferiores de la memoria para el texto del programa, los siguientes cuatro megabytes para datos y los cuatro megabytes superiores para la pila. Entre la parte superior de los datos y la base de la pila hay un agujero gigantesco que no se usa.

En la Fig. 4-10(b) vemos cómo la tabla de páginas de dos niveles funciona en este ejemplo. A la izquierda tenemos la tabla de páginas de nivel superior, con 1024 entradas, que corresponden al campo PT1 de 10 bits. Cuando se presenta una dirección virtual a la MMU, lo primero que hace es extraer el campo PT1 y usar este valor como índice para consultar la tabla de páginas de nivel superior. Cada una de estas 1024 entradas representa 4M porque todo el espacio de direcciones virtual de cuatro gigabytes (esto es, de 32 bits) se ha dividido en porciones de 1024 bytes.

La entrada que se ubica indizando en la tabla de páginas de nivel superior contiene la dirección o el número de marco de página de una tabla de páginas de segundo nivel. La entrada 0 de la tabla de páginas de nivel superior apunta a la tabla de páginas para el texto del programa, la entrada 1 apunta a la tabla de páginas para los datos, y la entrada 1023 apunta a la tabla de páginas para la pila. Las otras entradas (sombreadas) no se usan. Ahora se usa el campo PT2 como índice para consultar la tabla de páginas de segundo nivel seleccionada y encontrar el número de marco de página que corresponde a la página misma.

Por ejemplo, consideremos la dirección virtual de 32 bits 0x00403004 (4 206 596 decimal), que está a una distancia de 12 292 bytes del principio de los datos. Esta dirección corresponde a PT1 = 1, PT2 == 3 y Offset = 4. La MMU primero usa PT1 para consultar la tabla de páginas de nivel superior y obtener la entrada 1, que corresponde a las direcciones de 4M a 8M; luego, utiliza PT2 para consultar la tabla de páginas de segundo nivel que acaba de localizarse y extraer la entrada 3, que corresponde a las direcciones 12288 a 16383 dentro de su trozo de 4M (o sea, las direcciones absolutas 4 206 592 a 4 210 687). Esta entrada contiene el número de marco de página de la página que contiene la dirección virtual 0x00403004. Si esa página no está en la memoria, el bit Presente/ausente de la entrada de la tabla de páginas será cero, y causará una falla de páginas. Si la página está en la memoria, el número de marco de página tomado de la tabla de páginas de segundo nivel se combina con la distancia (4) para construir una dirección física. Esta dirección se coloca en el bus y se envía a la memoria.

El aspecto interesante de la Fig. 4-10 es que a pesar de que el espacio de direcciones contiene más de un millón de páginas, sólo se necesitan en realidad cuatro tablas de páginas: la tabla de nivel superior y las tablas de segundo nivel correspondientes a 0 a 4M, 4M a 8M y los 4M superiores. Los bits Presente/ausente de 1021 entradas de la tabla de páginas de nivel superior se ponen en 0, forzando una falla de página si se accede a ellas. Si llega a ocurrir esto, el sistema operativo se dará cuenta de que el proceso está tratando de hacer referencia a memoria a la que no tiene autorización de acceder, y tomará las medidas apropiadas, como enviarle una señal o terminarlo. En este ejemplo escogimos números redondos para los diferentes tamaños y escogimos PT1 igual a PT2, pero desde luego en la práctica pueden usarse otros valores.

El sistema de tablas de páginas de dos niveles de la Fig. 4-10 puede ampliarse a tres, cuatro o más niveles. Una mayor cantidad de niveles confiere más flexibilidad, pero es dudoso que la complejidad adicional se justifique más allá de los tres niveles.

Pasemos ahora de la estructura macroscópica de las tablas de páginas a los detalles de una entrada de tabla de páginas. La organización exacta de una entrada depende en gran medida de la máquina, pero la clase de información presente es aproximadamente la misma en todas las máquinas. En la Fig. 4-11 presentamos un ejemplo de entrada de tabla de páginas. El tamaño varía de una computadora a otra, pero un tamaño común es de 32 bits. El campo más importante es el Número de marco de página. Después de todo, el objetivo de la transformación de páginas es localizar este valor. Junto a él tenemos el bit Presente/ausente. Si este bit es 1, la entrada es válida y puede usarse; si es 0, la página virtual a la que la entrada pertenece no está actualmente en la memoria. El acceso a una entrada de tabla de páginas que tiene este bit en 0 causa una falla de página.

Los bits de Protección indican qué clases de acceso están permitidas. En la forma más sencilla, este campo contiene un bit, y 0 significa lectura/escritura mientras que 1 significa sólo lectura.

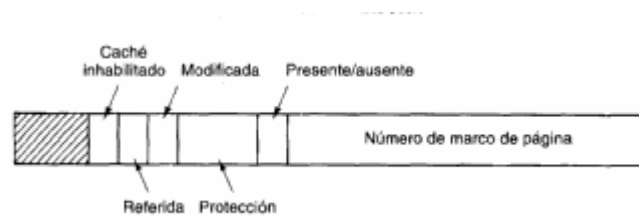


Figura 4-11. Entrada de tabla de páginas representativa.

Una organización más compleja tendría tres bits, para habilitar la lectura, escritura y ejecución, respectivamente.

Los bits *Modificada* y *Referida* siguen la pista al empleo de la página. Cuando se escribe en una página, el hardware automáticamente enciende el bit *Modificada*. Este bit es útil cuando el sistema operativo decide reutilizar un marco de página. Si la página que contiene ha sido modificada (es decir, está "sucia"), se deberá escribir de vuelta en el disco. Si la página no ha sido modificada (si está "limpia"), puede abandonarse, ya que la copia que está en el disco aún es válida. Este bit también se conoce como bit sucio, pues refleja el estado de la página.

El bit *Referida* se enciende cada vez que se hace referencia a una página, sea para lectura o escritura. Su propósito es ayudar al sistema operativo a escoger la página que desalojará cuando ocurra una falla de página. Las páginas que no se están usando son mejores candidatos que las que están en uso, y este bit desempeña un papel importante en varios de los algoritmos de reemplazo de páginas que estudiaremos más adelante en este capítulo.

El último bit permite inhabilitar la colocación en caché de la página. Esta capacidad es importante en el caso de páginas que corresponden a registros de dispositivos en lugar de memoria. Si el sistema operativo se queda dando vueltas en un ciclo esperando que un dispositivo de E/S responda a un comando que se le acaba de dar, es indispensable que el hardware continúe obteniendo la palabra del dispositivo y no use una copia vieja guardada en caché. Con este bit, puede desactivarse la colocación en caché. Las máquinas que tienen un espacio de E/S independiente y no usan E/S con mapa en la memoria no necesitan este bit.

Adviértase que la dirección en disco en la que está guardada la página cuando no está en memoria no forma parte de la tabla de páginas. La razón es sencilla. La tabla de páginas contiene sólo la información que el hardware necesita para traducir una dirección virtual en una dirección física. La información que el sistema operativo necesita para manejar las fallas de página se mantiene en tablas de software dentro del sistema operativo.

4.3.3 TLB — Buffers de consulta para traducción

En la mayor parte de los esquemas de paginación, las tablas de páginas se mantienen en la memoria, debido a su gran tamaño. Esto puede tener un impacto enorme sobre el rendimiento. Consideremos, por ejemplo, una instrucción que copia un registro en otro. Si no hay paginación, esta instrucción sólo hace referencia a la memoria una vez, para obtener la instrucción. Si hay

paginación, se requieren referencias adicionales a la memoria para acceder a la tabla de páginas. Puesto que la velocidad de ejecución generalmente está limitada por la rapidez con que la CPU puede obtener instrucciones y datos de la memoria, tener que referirse dos veces a la tabla de páginas por cada referencia a la memoria reduce el rendimiento a una tercera parte. En estas condiciones, nadie usaría paginación.

Los diseñadores de computadoras han estado al tanto de este problema desde hace años y han encontrado una solución. Su solución se basa en la observación de que muchos programas tienden a efectuar un gran número de referencias a un número pequeño de páginas, y no al revés. Así, sólo se lee mucho una fracción pequeña de las entradas de la tabla de páginas; el resto apenas si se usa.

La solución que se encontró consiste en equipar las computadoras con un pequeño dispositivo de hardware para transformar las direcciones virtuales en físicas sin pasar por la tabla de páginas. El dispositivo, llamado TLB (buffer de consulta para traducción, en inglés, Translation Lookaside Buffer) o también memoria asociativa, se ilustra en la Fig. 4-12. Generalmente, el TLB está dentro de la MMU y consiste en un pequeño número de entradas, ocho en este ejemplo, pero pocas veces más que 64. Cada entrada contiene información acerca de una página; en particular, el número de página virtual, un bit que se enciende cuando la página se modifica, el código de protección (permisos de leer/escribir/ejecutar) y el marco de página físico en el que se encuentra la página. Estos campos tienen una correspondencia uno a uno con los campos de la tabla de páginas. Otro bit indica si la entrada es válida (si está en uso) o no.

Válida	Página virtual	Modificada	Protección	Marco de página
1	140	1	RW	31
1	20	0	RX	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	RX	50
1	21	0	RX	45
1	860	1	RW	14
1	861	1	RW	75

Figura 4-12. Un TLB para agilizar la paginación

Un ejemplo que podría generar el TLB de la Fig. 4-12 es un proceso en un ciclo que abarca las páginas virtuales 19, 20 y 21, así que estas entradas del TLB tienen códigos de protección para leer y ejecutar. Los principales datos que se están usando (digamos, un arreglo que se está procesando) están en las páginas 129 y 130. La página 140 contiene los índices empleados en los cálculos del arreglo. Por último, la pila está en las páginas 860 y 861.

Veamos ahora cómo funciona el TLB. Cuando se presenta una dirección virtual a la MMU para ser traducida, lo primero que hace el hardware es verificar si su número de página virtual

está presente en el TLB, comparándolo con todas las entradas simultáneamente (es decir, en paralelo). Si hay una concordancia válida y el acceso no viola los bits de protección, el marco de página se toma directamente del TLB, sin recurrir a la tabla de páginas. Si el número de página virtual está presente en el TLB pero la instrucción está tratando de escribir en una página sólo de lectura, se generará una falla de protección, tal como sucedería si se usara la tabla de páginas.

El caso interesante es qué sucede si el número de página virtual no está en el TLB. La MMU detecta la falta de concordancia y realiza una búsqueda ordinaria en la tabla de páginas. A continuación desaloja una de las entradas del TLB y la sustituye por la entrada de tabla de páginas que encontró. Por tanto, si esa página se usa pronto otra vez, se le encontrará en el TLB. Cuando una entrada se desaloja del TLB, el bit de modificada se copia en su entrada de tabla de páginas en la memoria. Los demás valores ya están ahí. Cuando el TLB se carga de la tabla de páginas, se toman todos los campos de la memoria.

Administración de TLB por software

Hasta aquí, hemos supuesto que todas las máquinas con memoria virtual paginada tienen tablas de páginas reconocidas por el hardware, más un TLB. En este diseño, la administración del TLB y el manejo de fallas de TLB corren por cuenta exclusivamente del hardware de la MMU. Las trampas al sistema operativo sólo ocurren cuando una página no está en la memoria.

En el pasado, este supuesto era cierto, pero algunas máquinas RISC modernas, incluidas MIPS, Alpha y HP PA, realizan casi toda esta administración de páginas en software. En estas máquinas, el sistema operativo carga explícitamente las entradas del TLB. Cuando no se encuentra una página virtual en el TLB, en lugar de que la MMU se remita simplemente a las tablas de páginas para encontrar y obtener la referencia de página requerida, genera una falla de TLB y deja que el sistema operativo se encargue del problema. Éste debe encontrar la página, eliminar una entrada del TLB, introducir la nueva y reiniciar la instrucción que falló. Y, desde luego, todo esto debe hacerse con unas cuantas instrucciones, porque las fallas de TLB podrían ocurrir con mucha mayor frecuencia que las fallas de página.

Resulta sorprendente que, si el TLB tiene un tamaño razonable (digamos 64 entradas) para reducir la tasa de fallas, la administración por software del TLB es muy eficiente. La ganancia principal aquí es que la MMU es mucho más sencilla, lo que libera una cantidad considerable de área en el chip de la CPU para caches y otras características que pueden mejorar el rendimiento. Uhlig et al. (1994) estudian detalladamente la administración de TLB en software.

Se han inventado diversas estrategias para mejorar el rendimiento en máquinas que realizan la administración de TLB en software. Un enfoque ataca tanto la reducción de las fallas de TLB como la reducción del costo de una falla de TLB cuando ocurre (Bala et al., 1994). A fin de reducir las fallas de TLB, el sistema operativo a veces puede usar su intuición para averiguar cuáles S páginas es probable que se usen a continuación y precargar sus entradas en el TLB. Por ejemplo, I cuando un proceso cliente efectúa un RPC para un proceso servidor en la misma máquina, es I muy probable que el servidor tenga que ejecutarse pronto. Sabiendo esto, al tiempo que procesa { la trampa para efectuar el RPC, el sistema puede averiguar dónde están las páginas de código, datos y pila del servidor, y traer sus entradas al TLB antes de que puedan causar fallas de TLB.

La forma normal de procesar una falla de TLB, sea en hardware o en software, es ir a la tabla de páginas y realizar las operaciones de indización necesarias para localizar la página a la que se hizo referencia. Lo malo de realizar esta búsqueda en software es que las páginas que contienen la tabla de páginas podrían no estar en el TLB, lo que causaría fallas de TLB adicionales durante el procesamiento. Estas fallas pueden reducirse manteniendo un caché grande (p. ej., 4K) en software de entradas de TLB en un lugar fijo cuya página siempre se mantiene en el TLB. Si primero se busca en el caché en software, el sistema operativo puede reducir sustancialmente las fallas de TLB.

4.3.4 Tablas de páginas invertidas

Las tablas de páginas tradicionales del tipo que hemos descrito hasta ahora requieren una entrada por cada página virtual, ya que están indizadas por número de página virtual. Si el espacio de direcciones consta de 232 bytes, con 4096 bytes por página, se necesitará más de un millón de entradas de tabla. Como mínimo, la tabla de páginas tendrá que ocupar 4 megabytes. En los sistemas grandes, este tamaño probablemente será manejable.

Sin embargo, al hacerse cada vez más comunes las computadoras de 64 bits, la situación cambia drásticamente. Si el espacio de direcciones ahora tiene 264 bytes, con páginas de 4K, necesitaremos más de 1015 bytes para la tabla de páginas. Sacar de circulación un millón de gigabytes sólo para la tabla de páginas no es razonable, ni ahora, ni durante varias décadas más, sí es que alguna vez lo es. Por tanto, se requiere una solución distinta para los espacios de direcciones virtuales de 64 bits paginados.

Una solución es la tabla de páginas invertida. En este diseño, hay una entrada por marco de página de la memoria real, no por cada página del espacio de direcciones virtual. Por ejemplo, con direcciones virtuales de 64 bits, páginas de 4K y 32MB de RAM, una tabla de páginas invertida sólo requiere 8192 entradas. La entrada indica cuál (proceso, página virtual) está en ese marco de página.

Aunque las tablas de páginas invertidas ahorran enormes cantidades de espacio, al menos cuando el espacio de direcciones virtual es mucho más grande que la memoria física, tienen una desventaja importante: la traducción de virtual a física se vuelve mucho más difícil. Cuando el proceso n hace referencia a la página virtual p , el hardware ya no puede encontrar la página física usando p como índice de la tabla de páginas. En vez de ello, debe buscar en toda la tabla de páginas invertida una entrada (n, p) . Además, esta búsqueda debe efectuarse en cada referencia a la memoria, no sólo cuando hay una falla de página. Examinar una tabla de 8K cada vez que se hace referencia a la memoria no es la forma de hacer que una máquina sea vertiginosamente rápida.

La solución a este dilema es usar el TLB. Si el TLB puede contener todas las páginas de uso pesado, la traducción puede efectuarse con tanta rapidez como con las tablas de páginas normales. Sin embargo, si ocurre una falla de TLB, habrá que examinar la tabla de páginas invertida. Si se usa una tabla de dispersión como índice de la tabla de páginas invertida, esta búsqueda puede hacerse razonablemente rápida. Ya se están usando tablas de páginas invertidas en algunas estaciones de trabajo IBM y Hewlett-Packard, y se harán más comunes conforme se generalice el uso de máquinas de 64 bits.

Se pueden encontrar otros enfoques para manejar memorias virtuales grandes en (Huck y Hays, 1993; Talluri y Hill, 1994; y Talluri et al., 1995).

4.4 ALGORITMOS DE SUSTITUCIÓN DE PÁGINAS

Cuando ocurre una falla de página, el sistema operativo tiene que escoger la página que sacará de la memoria para que pueda entrar la nueva página. Si la página que se eliminará fue modificada mientras estaba en la memoria, se debe reescribir en el disco a fin de actualizar la copia del disco, pero si no fue así (p. ej., si la página contenía texto de programa), la copia en disco ya estará actualizada y no será necesario reescribirla. La nueva página simplemente sobrescribe la que está siendo desalojada.

Si bien sería posible escoger una página al azar para ser desalojada cuando ocurre una falla de página, el rendimiento del sistema es mucho mejor si se escoge una página que no se usa mucho. Si se elimina una página de mucho uso, probablemente tendrá que traerse pronto a la memoria otra vez, aumentando el gasto extra. Se ha trabajado mucho sobre el tema de los algoritmos de reemplazo de páginas, tanto teórica como experimentalmente. A continuación describimos algunos de los algoritmos más importantes.

4.4.1 El algoritmo de sustitución de páginas óptimo

El mejor algoritmo de reemplazo de páginas posible es fácil de describir pero imposible de implementar. En el momento en que ocurre una falla de páginas, algún conjunto de páginas está en la memoria. A una de estas páginas se hará referencia en la siguiente instrucción (la página que contiene esa instrucción). Otras páginas podrían no necesitarse sino hasta 10, 100 o tal vez 1000 instrucciones después. Cada página puede rotularse con el número de instrucciones que se ejecutarán antes de que se haga referencia a esa página.

El algoritmo de reemplazo de páginas óptimo simplemente dice que se debe eliminar la página que tenga el rótulo más alto. Si una página no se va a usar sino hasta después de 8 millones de instrucciones y otra página no se usará sino hasta después de 6 millones de instrucciones, el desalojo de la primera postergará la falla de página que la traerá de nuevo a la memoria lo más lejos hacia el futuro que es posible. Las computadoras, al igual que las personas, tratan de aplazar los sucesos desagradables el mayor tiempo que se puede.

El único problema con este algoritmo es que no se puede poner en práctica. En el momento en que ocurre la falla de página, el sistema operativo no tiene manera de saber cuándo se hará referencia a cada una de las páginas. (Vimos una situación similar antes con el algoritmo de planificación del primer trabajo más corto; ¿cómo puede el sistema saber cuál trabajo es el más corto?) No obstante, si se ejecuta un programa en un simulador y se toma nota de todas las referencias a páginas, es posible implementar el reemplazo de páginas óptimo en la segunda ejecución utilizando la información recabada durante la primera.

De este modo, es posible comparar el rendimiento de los algoritmos realizables con el mejor posible. Si un sistema operativo logra un rendimiento, digamos, sólo 1 % peor que el algoritmo óptimo, los esfuerzos dedicados a buscar un mejor algoritmo redundarán cuando más en una mejora del 1%.

Para evitar confusiones, debemos dejar sentado que esta bitácora de referencias a páginas sólo es válido para el programa que acaba de medirse. El algoritmo de reemplazo de páginas derivado de él es específico para éste y sólo ese programa. Aunque este método es útil para evaluar los algoritmos de reemplazo de páginas, no sirve de nada en los sistemas prácticos. A continuación estudiaremos algoritmos que sí son útiles en los sistemas reales.

4.4.2 El algoritmo de sustitución de páginas no usadas recientemente

Para que el sistema operativo pueda recabar datos estadísticos útiles sobre cuáles páginas se están usando y cuáles no, la mayor parte de las computadoras con memoria virtual tienen dos bits de situación asociados a cada página. R se enciende cada vez que se hace referencia a la página (lectura o escritura). M se enciende cuando se escribe la página (es decir, se modifica). Los bits están contenidos en cada entrada de la tabla de páginas, como se muestra en la Fig. 4-11. Es importante darse cuenta de que estos bits se deben actualizar en cada referencia a la memoria, así que es vital que sea el hardware quien los ajuste. Una vez puesto en 1 un bit, seguirá siendo 1 hasta que el sistema operativo lo ponga en 0 en software.

Si el hardware no tiene estos bits, pueden simularse como sigue. Cuando se inicia un proceso, todas sus entradas de la tabla de páginas se marcan como que no están en la memoria. Tan pronto como se haga referencia a cualquier página, ocurrirá una falla de página. Entonces, el sistema operativo enciende el bit R (en sus tablas internas), modifica la entrada de la tabla de páginas de modo que apunte a la página correcta, con el modo SÓLO LECTURA, y reinicia la instrucción. Si subsecuentemente se escribe en la página, ocurrirá otra falla de página, lo que permitirá el sistema operativo encender el bit M y cambiar el modo de la página a LECTURA/ESCRITURA.

Los bits R y M pueden servir para construir un algoritmo de paginación sencillo como sigue. Cuando se inicia un proceso, el sistema operativo pone en 0 los dos bits de todas sus páginas. Periódicamente (p. ej., en cada interrupción de reloj), se apaga el bit R, a fin de distinguir páginas a las que no se ha hecho referencia recientemente de las que sí se han leído.

Cuando ocurre una falla de página, el sistema operativo examina todas las páginas y las divide en cuatro categorías con base en los valores actuales de sus bits R y M:

Clase 0: no referida, no modificada.

Clase 1: no referida, modificada.

Clase 2: referida, no modificada.

Clase 3: referida, modificada.

Aunque a primera vista las páginas de clase 1 parecen imposibles, ocurren cuando una interrupción del reloj apaga el bit R de una página de clase 3. Las interrupciones de reloj no borran el bit M esta información se necesita para determinar si hay que reescribir en disco la página o no.

El algoritmo **NRU (no utilizada recientemente)** elimina una página al azar de la clase no vacía que tiene el número más bajo. Este algoritmo supone que es mejor eliminar una página modificada a la que no se ha hecho referencia en por lo menos un tic del reloj (por lo regular 20 ms) que una página limpia que no se está usando mucho. El atractivo principal de NRU es que es fácil de entender, eficiente de implementar y tiene un rendimiento que, si bien ciertamente no es óptimo, a menudo es adecuado.

4.4.3 El algoritmo de sustitución de páginas de primera que entra, primera que sale (FIFO)

Otro algoritmo de paginación con bajo gasto extra es el algoritmo **FIFO (primera que entra, primera que sale)**. Para ilustrar su funcionamiento, consideremos un supermercado que tiene suficientes anaqueles para exhibir exactamente k productos distintos. Un día, alguna compañía introduce un nuevo alimento: yogurt orgánico liofilizado instantáneo que puede reconstituirse en un homo de microondas. Su éxito es inmediato, así que nuestro supermercado finito tiene que deshacerse de un producto viejo para poder tener el nuevo en exhibición.

Una posibilidad consiste en encontrar el producto que el supermercado ha tenido en exhibición durante más tiempo (es decir, algo que comenzó a vender hace 120 años) y deshacerse de él bajo el supuesto de que a nadie le interesa ya. En efecto, el supermercado mantiene una lista enlazada de todos los productos que vende en el orden en que se introdujeron. El nuevo se coloca al final de la lista; el que está a la cabeza de la lista se elimina.

Como algoritmo de reemplazo de páginas, puede aplicarse la misma idea. El sistema operativo mantiene una lista de todas las páginas que están en la memoria, siendo la página que está a la cabeza de la lista la más vieja, y la del final, la más reciente. Cuando hay una falla de página, se elimina la página que está a la cabeza de la lista y se agrega la nueva página al final. Cuando FIFO se aplica a una tienda, el producto eliminado podría ser cera para el bigote, pero también podría ser harina, sal o mantequilla. Cuando FIFO se aplica a computadoras, surge el mismo problema. Por esta razón, casi nunca se usa FIFO en su forma pura.

4.4.4 El algoritmo de sustitución de páginas de segunda oportunidad

Una modificación sencilla de FIFO que evita el problema de desalojar una página muy utilizada consiste en inspeccionar el bit R de la página más vieja. Si es O , sabremos que la página, además de ser vieja, no ha sido utilizada recientemente, así que la reemplazamos de inmediato. Si el bit R es 1 , se apaga el bit, se coloca la página al final de la lista de páginas, y se actualiza su tiempo de carga como si acabara de ser traída a la memoria. Luego continúa la búsqueda.

El funcionamiento de este algoritmo, llamado de **segunda oportunidad**, se muestra en la Fig. 4-13. En la Fig. 4-13(a) vemos que se mantienen las páginas de la A a la H en una lista enlazada ordenadas según el momento en que se trajeron a la memoria.

Supongamos que ocurre una falla de página en el tiempo 20 . La página más antigua es A , que legó en el tiempo 0 , cuando se inició el proceso. Si A tiene el bit R apagado, es desalojada de la memoria, ya sea escribiéndose en el disco (si está sucia) o simplemente abandonándose (si está limpia). En cambio,

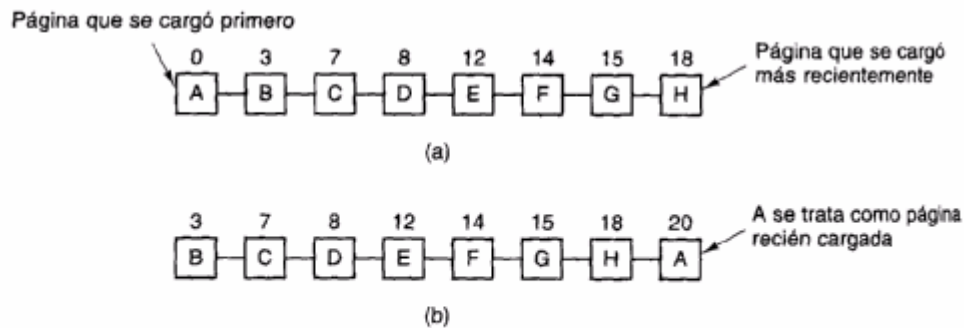


Figura 4-13. Funcionamiento del algoritmo de segunda oportunidad. (a) Páginas ordenadas en orden FIFO. (b) Lista de páginas si ocurre una falla de página en el tiempo 20 y A tiene su bit R encendido.

si el bit R está encendido, A se coloca al final de la lista y su "tiempo de carga" se ajusta al tiempo actual (20). También se apaga su bit R. La búsqueda de una página apropiada continúa con B.

Lo que hace el algoritmo de segunda oportunidad es buscar una página vieja a la que no se, haya hecho referencia en el intervalo de reloj anterior. Si se ha hecho referencia a todas las páginas, este algoritmo pasa a ser FIFO puro. Específicamente, imaginemos que todas las páginas de la Fig. 4-13(a) tienen su bit R encendido. Una por una, el sistema operativo pasará páginas al final de la lista, apagando el bit R cada vez que anexa una página al final de la lista. Tarde o temprano, regresará a la página A, que ahora tiene su bit R apagado. En este punto, A será desalojada. Así, el algoritmo siempre termina.

4.4.5 El algoritmo de sustitución de páginas por reloj

Aunque el algoritmo de segunda oportunidad es razonable, es innecesariamente eficiente porque constantemente cambia de lugar páginas dentro de su lista. Un enfoque mejor consiste en mantener todas las páginas en una lista circular con forma de reloj, como se muestra en la Fig. 4-14. Una manecilla apunta a la página más vieja.

Cuando ocurre una falla de página, se inspecciona la página a la que apunta la manecilla. Si su bit R es 0, se desaloja la página, se inserta la nueva página en el reloj en su lugar, y la manecilla avanza una posición. Si R es 1, se pone en 0 y la manecilla se avanza a la siguiente página. Este proceso se repite hasta encontrar una página con R = 0. No resulta sorprendente que este algoritmo se llame de **reloj**. La única diferencia respecto al de segunda oportunidad es la implementación.

4.4.6 El algoritmo de sustitución de páginas menos recientemente usadas (LRU)

Una buena aproximación al algoritmo óptimo se basa en la observación de que las páginas que han usado mucho en las últimas instrucciones probablemente se usarán mucho en las siguientes. Por otro lado, las páginas que hace mucho no se usan probablemente seguirán sin usarse durante largo tiempo. Esta idea sugiere un algoritmo factible: cuando ocurra una falla de página, se desalojará

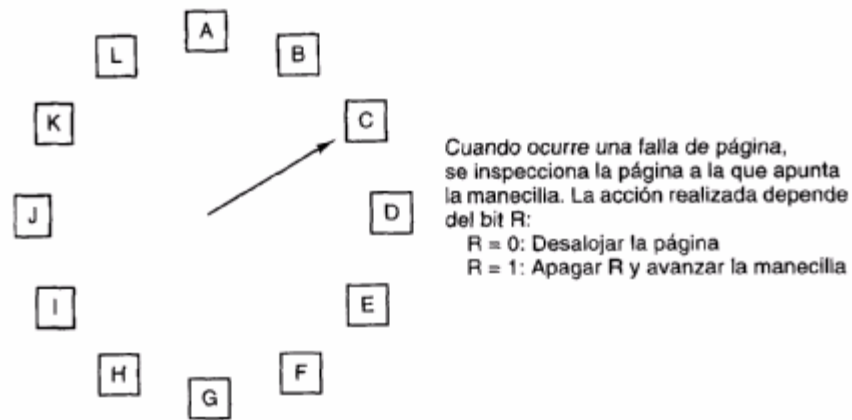


Figura 4-14. Algoritmo de reemplazo de páginas por reloj.

la página que haya estado más tiempo sin usarse. Esta estrategia se denomina paginación **LRN (menos recientemente utilizada)**.

Aunque LRU es factible en teoría, no es barato. Si queremos implementar LRU plenamente, necesitamos mantener una lista enlazada de todas las páginas que están en la memoria, con la página más recientemente utilizada al frente y la menos recientemente utilizada al final. El problema es que hay que actualizar la lista en cada referencia a la memoria. Encontrar una página en la lista, eliminarla y luego pasarla al frente es una operación que consume mucho tiempo, incluso en hardware (suponiendo que pudiera construirse tal hardware).

Sin embargo, hay otras formas de implementar LRU con hardware especial. Consideremos primero la forma más sencilla. Este método requiere equipar el hardware con un contador de 64 bits, C , que se incrementa automáticamente después de cada instrucción. Además, cada entrada de la tabla de páginas debe tener un campo con el tamaño suficiente para contener el contador. Después de cada referencia a la memoria, el valor actual de C se almacena en la entrada correspondiente a la página a la que se acaba de hacer referencia. Cuando ocurre una falla de página, el sistema operativo examina todos los contadores de la tabla de páginas hasta encontrar el más bajo. Esa página es la menos recientemente utilizada.

Examinemos ahora un segundo algoritmo LRU en hardware. Para una máquina con n marcos de página, el hardware de LRU puede mantener una matriz de $n \times n$ bits, que inicialmente son cero. Cada vez que se hace referencia al marco de página k , el hardware pone primero en 1 todos los bits de la fila k , y luego pone en 0 todos los bits de la columna k . En un instante dado, la fila cuyo valor binario sea el más bajo, será la de la página menos recientemente utilizada; la fila cuyo valor sea el siguiente más bajo será la de la siguiente página menos recientemente utilizada, etc. El funcionamiento de este algoritmo se muestra en la Fig. 4-15 para cuatro marcos de página y referencias a páginas en el orden

0123210323

Después de que se hace referencia a la página 0 tenemos la situación de la Fig. 4-15(a), y así sucesivamente.

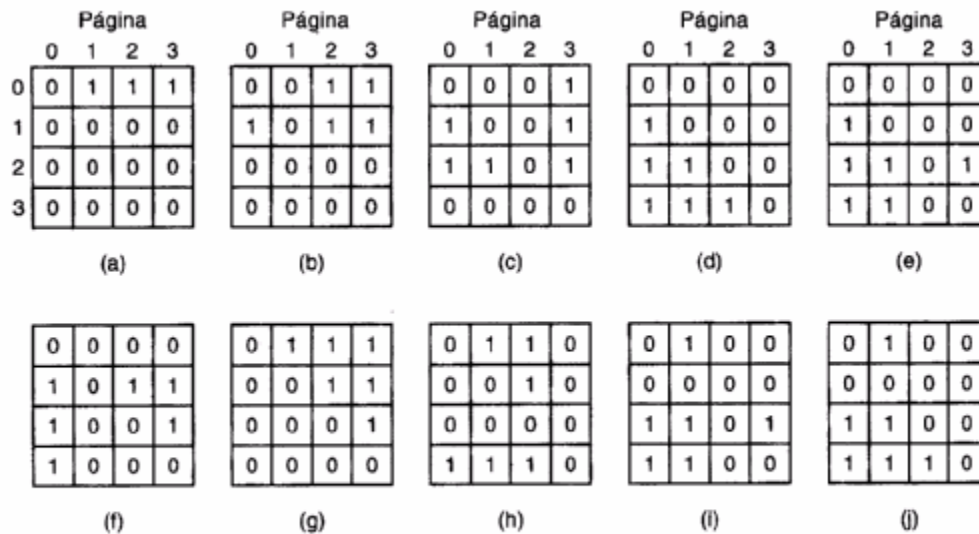


Figura 4-15. LRU empleando una matriz.

4.4.7 Simulación de LRU en software

Aunque los dos algoritmos LRU anteriores son factibles en principio, pocas máquinas, y tal ninguna, cuentan con este hardware, así que no sirven de mucho al diseñador de sistemas operados que está creando un sistema para una máquina que no tiene este hardware. Se necesita solución que pueda implementarse en software. Una posibilidad es el algoritmo **NFU (no utiliza frecuentemente)**, el cual requiere un contador en software asociado a cada página y que inicialmente vale 0. En cada interrupción del reloj, el sistema operativo examina todas las páginas que estañe memoria. Para cada página, el bit R, que es 0 o 1, se suma al contador. En efecto, los contados son un intento por contabilizar la frecuencia con que se hace referencia a cada página. Cuando ocurre una falla de página, se escoge para reemplazar la página que tiene el contador más bajo.

El problema con NFU es que nunca olvida nada. Por ejemplo, en un compilador de múltiples pasadas, las páginas que se usaron mucho durante la pasada 1 podrían tener todavía una cuenta alta durante varias pasadas posteriores. De hecho, si sucede que la pasada 1 tiene el tiempo ejecución más largo de todas, las páginas que contienen el código de pasadas subsecuentes podrían tener siempre cuentas más bajas que las de la primera pasada. Por tanto, el sistema operativo desalojará páginas útiles en lugar de páginas que ya no se están usando.

Por fortuna, una pequeña modificación de NFU hace que pueda simular LRU de forma satisfactoria. La modificación tiene dos partes. Primera, todos los contadores se desplazan a la derecha un bit antes de sumarles el bit R. Segunda, el bit R se suma al bit de la extrema izquierda al de la extrema derecha.

En la Fig. 4-16 se ilustra el funcionamiento del algoritmo modificado, llamado de **maduración**. Supongamos que después del primer tic del reloj los bits R de las páginas 0 a 5 tienen los

valores 1, 0, 1, 0, 1 y 1 respectivamente (la página 0 es 1, la página 1 es 0, la página 2 es 1, etc.). Dicho de otro modo, entre el tic 0 y el tic 1 se hizo referencia a las páginas 0,2,4 y 5, así que sus bits R se pusieron en 1, mientras que los demás siguieron en 0. Después de que los seis contadores correspondientes se desplazan a la derecha y se les inserta el bit R por la izquierda, tienen los valores que se muestran en la Fig. 4-16(a). Las cuatro columnas restantes muestran los seis contadores después de los siguientes cuatro tics del reloj.

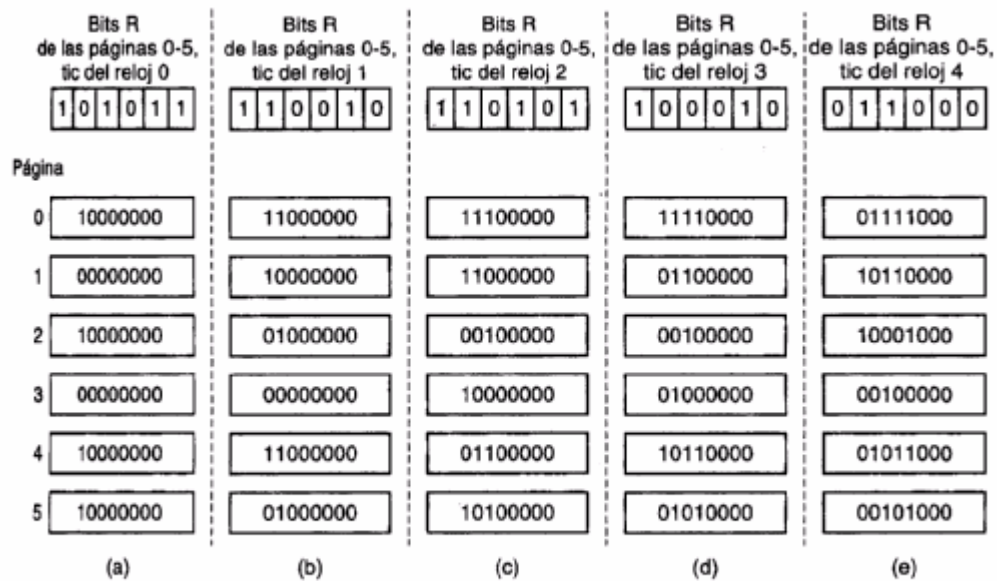


Figura 4-16. El algoritmo de maduración simula LRU en software. Se muestran seis páginas durante cinco tics del reloj. Los cinco tics se representan con las Figs. (a) a (e).

Cuando ocurre una falla de página, se elimina la página con el contador más bajo. Es evidente que una página a la que no se ha hecho referencia durante, digamos, cuatro tics del reloj tiene cuatro ceros a la izquierda en su contador, y por tanto tiene un valor más bajo que el contador de una página a la que no se ha hecho referencia durante tres tics del reloj.

Este algoritmo difiere de LRU en dos aspectos. Consideremos las páginas 3 y 5 en la Fig. 4-16(e). A ninguna de las dos se ha hecho referencia durante dos tics del reloj; a las dos se hizo referencia en el tic anterior. Según LRU, si es preciso reemplazar una página, deberíamos escoger una de estas dos. El problema es que no sabemos cuál de ellas es a la que se hizo referencia por última vez en el intervalo entre el tic 1 y el 2. Al registrar sólo un bit por intervalo de tiempo, hemos perdido la capacidad para distinguir entre las referencias al principio del intervalo de reloj y aquellas que ocurren posteriormente. Lo único que podemos hacer es eliminar la página 3, porque la página 5 también tuvo una referencia dos tics antes, y la página 3 no.

La segunda diferencia entre LRU y maduración es que en esta última los contadores tienen un número finito de bits, ocho en nuestro ejemplo. Supongamos que dos páginas tienen un valor de cero en el contador. Lo único que podemos hacer es escoger una de ellas al azar. En realidad, bien

puede ser que a una de ellas se haya hecho referencia por última vez hace nueve tics y que otra se haya hecho referencia por última vez hace 1000 tics. No tenemos forma de saber esto no obstante, en la práctica generalmente bastan ocho bits si un tic de reloj tiene alrededor de 20 Si no se ha hecho referencia a una página en 160 ms, probablemente no es muy importante.

4.5 ASPECTOS DE DISEÑO DE LOS SISTEMAS CON PAGINACIÓN

En las secciones anteriores hemos explicado cómo funciona la paginación y hemos presentado algunos de los algoritmos de reemplazo de páginas básicos. Sin embargo, no basta con conocer los aspectos mecánicos del funcionamiento. Para diseñar un sistema, necesitamos saber mucho más si queremos lograr que funcione bien. La diferencia es similar a la que existe entre saber cómo se mueven la torre, el caballo, el alfil y las demás piezas de ajedrez, y ser un buen jugador En las siguientes secciones examinaremos otros aspectos que los diseñadores de sistemas operad deben considerar detenidamente si quieren obtener un buen rendimiento de un sistema de paginación

4.5.1 El modelo de conjunto de trabajo

En la forma más pura de paginación, los procesos se inician con ninguna de sus páginas en la memoria. Tan pronto como la CPU trata de obtener la primera instrucción, detecta una falla página que hace que el sistema operativo traiga a la memoria la página que contiene dicha instrucción. Por lo regular, pronto ocurren otras fallas de página al necesitarse variables globales y la pila. Después de un tiempo, el proceso tiene la mayor parte de las páginas que necesita y se dedica a ejecutarse con relativamente pocas fallas de página. Tal estrategia se denomina **paginación por demanda** porque las páginas se cargan sólo cuando se piden, no por adelantado.

Desde luego, es fácil escribir un programa de prueba que lea sistemáticamente todas páginas de un espacio de direcciones grande, causando tantas fallas de página que no habrá suficiente memoria para contenerlas todas. Por fortuna, la mayor parte de los procesos no funcionan así; exhiben una **localidad de referencia**, lo que significa que durante cualquier fase de ejecución, el proceso sólo hace referencia a una fracción relativamente pequeña de sus páginas. Cada pasada de un compilador de múltiples pasadas, por ejemplo, sólo hace referencia a fracción de todas las páginas, que es diferente en cada pasada.

El conjunto de páginas que un proceso está usando actualmente es su **conjunto de trabajo** (Denning, 1968a; Denning, 1980). Si todo el conjunto de trabajo está en la memoria, el proceso ejecutará sin causar muchas fallas hasta que pase a otra fase de su ejecución (p. ej., la siguiente pasada de un compilador). Si la memoria disponible es demasiado pequeña para contener todo conjunto de trabajo, el proceso causará muchas fallas de página y se ejecutará lentamente, ya que la ejecución de una instrucción normalmente toma unos cuantos nanosegundos, y la lectura; una página de disco suele tomar decenas de milisegundos. Si ejecuta sólo una o dos instrucción cada 20 milisegundos, el proceso tardará muchísimo en terminar. Se dice que un programa q causa fallas de página repetidamente después de unas cuantas instrucciones se está "sacudiendo (en inglés, thrashing, que es el término que utilizaremos) (Denning, 1968b). 'i

En un sistema de tiempo compartido, los procesos a menudo se transfieren al disco (esto es, todas sus páginas se eliminan de la memoria) para que otro proceso tenga oportunidad de usar la CPU. Surge la pregunta de qué hacer cuando se vuelve a traer un proceso a la memoria. Técnicamente, no hay que hacer nada. El proceso simplemente causará fallas de página hasta que haya cargado su conjunto de trabajo. El problema es que tener 20, 50 o incluso 100 fallas de página cada vez que se carga un proceso hace lento el funcionamiento y además desperdicia mucho tiempo de CPU, pues el sistema operativo requiere unos cuantos milisegundos de tiempo de CPU para procesar una falla de página.

Por ello, muchos sistemas de paginación tratan de seguir la pista al conjunto de trabajo de cada proceso y se aseguran de que esté en la memoria antes de dejar que el proceso se ejecute. Este enfoque se denomina **modelo de conjunto de trabajo** (Denning, 1970) y está diseñado para reducir considerablemente la tasa de fallas de página. La carga de las páginas antes de dejar que los procesos se ejecuten también se denomina **prepaginación**.

A fin de implementar el modelo de conjunto de trabajo, el sistema operativo tiene que saber cuáles páginas están en el conjunto de trabajo. Una forma de seguir la pista a esta información es usar el algoritmo de maduración que acabamos de explicar. Cualquier página que contenga un bit 1 entre los n bits de orden alto del contador se considera miembro del conjunto de trabajo. Si no se ha hecho referencia a una página en n tics de reloj consecutivos, se omite del conjunto de trabajo. El parámetro n se debe determinar experimentalmente para cada sistema, pero por lo regular el rendimiento del sistema no es muy sensible al valor exacto.

La información relativa al conjunto de trabajo puede servir para mejorar el rendimiento del algoritmo del reloj. Normalmente, cuando la manecilla apunta a una página cuyo bit R es O , esa página se desaloja. La mejora consiste en verificar si esa página forma parte del conjunto de trabajo del proceso en curso. Si lo es, se le perdona la vida. Este algoritmo se conoce como **wslock**.

4.5.2 Políticas de asignación local vs. global

En las secciones anteriores hemos descrito varios algoritmos para escoger la página que será sustituida cuando ocurra una falla de página. Un problema importante asociado a esta decisión (que tuvimos cuidado de no mencionar hasta ahora) es la forma como debe repartirse la memoria entre los procesos ejecutables competidores.

Dé una mirada a la Fig. 4-17(a). En esta figura, tres procesos, A, B y C, constituyen el conjunto de los procesos ejecutables. Supongamos que A causa una falla de página. ¿El algoritmo de sustitución de páginas deberá tratar de encontrar la página menos recientemente utilizada considerando sólo las seis páginas que actualmente están asignadas a A o deberá considerar todas las páginas que están en la memoria? Si se examinan sólo las páginas de A, la página con el valor de edad más bajo será A5, y tendremos la situación de la Fig. 4-17(b).

Por otro lado, si se desaloja la página con el valor de edad más bajo sin tener en cuenta a quién pertenece, se escogerá la página B3 y tendremos la situación de la Fig. 4-17(c). El algoritmo de la Fig. 4-17(b) es un algoritmo de reemplazo de páginas **local**, en tanto que el de la Fig. 4-17(c) es **global**. Los algoritmos locales corresponden efectivamente a asignar a cada proceso una fracción fija de la memoria. Los algoritmos globales reparten dinámicamente marcos de página entre los

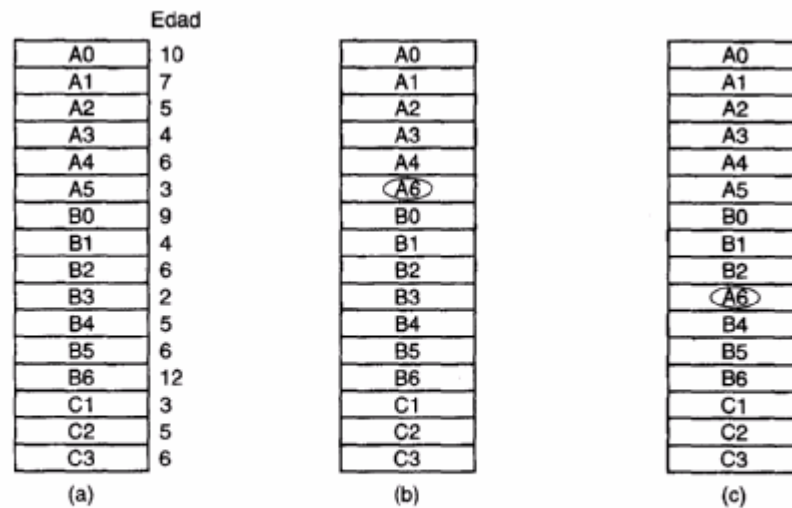


Figura 4-17. Reemplazo de páginas local vs. global. (a) Configuración original. (b) Reemplazo de páginas local. (c) Reemplazo de páginas global.

procesos ejecutables. Por tanto, el número de marcos de página asignados a cada proceso varía con el tiempo.

En general, los algoritmos globales funcionan mejor, sobre todo cuando el tamaño del conjunto de trabajo puede variar durante la vida de un proceso. Si se emplea un algoritmo local y el conjunto de trabajo crece, habrá thrashing, incluso si hay muchos marcos de página libres. Si el conjunto de trabajo se reduce, los algoritmos locales desperdician memoria. Si se emplea un algoritmo global, el sistema debe decidir continuamente cuántos marcos de página habrá de asignar a cada proceso. Una forma de vigilar el tamaño del conjunto de trabajo es examinar los bits de maduración, pero este enfoque no necesariamente evita el thrashing. El tamaño del conjunto de trabajo puede cambiar en microsegundos, en tanto que los bits de maduración son una medida burda que abarca varios tics del reloj.

Otro enfoque consiste en tener un algoritmo para asignar marcos de página a los procesos. Una posibilidad es determinar periódicamente el número de procesos en ejecución y asignar a cada uno una porción equitativa. Así, con 475 marcos de página disponibles (es decir, no ocupar dos por el sistema operativo) y 10 procesos, le tocarían 47 marcos a cada proceso. Los cinco restantes constituirían una reserva que se usaría en caso de ocurrir fallas de página.

Aunque este método parece justo, no tiene mucha lógica dar porciones iguales de la memoria un proceso de 10K y a uno de 300K. En vez de ello, se pueden asignar las páginas en proporción al tamaño total de cada proceso, de modo que un proceso de 300K reciba 30 veces más memoria que uno de 10K. Tal vez sea prudente dar a cada proceso alguna cantidad mínima para que pueda ejecutarse por más pequeño que sea. En algunas máquinas, por ejemplo, una sola instrucción podría requerir hasta seis páginas porque la instrucción misma, el operando de origen y el operando de destino podrían estar cada uno exactamente en la frontera entre dos páginas, con una parte

en cada lado. Con una asignación de sólo cinco páginas, un programa que incluyera una instrucción semejante no podría ejecutarse.

Ni el método de reparto equitativo ni el de reparto proporcional resuelven directamente el problema del thrashing. Una forma más directa de controlarlo consiste en usar el algoritmo de asignación de **frecuencia de fallas de página**, o **PFF**. En una clase amplia de algoritmos de sustitución de páginas, incluido LRU, se sabe que la tasa de fallas disminuye al asignarse más páginas, como ya explicamos. Esta propiedad se ilustra en la Fig. 4-18.



Figura 4-18. La tasa de fallas de página en función del número de marcos de página asignados.

La línea punteada A corresponde a una tasa de fallas de página inaceptablemente alta, así que se asignan más marcos de página al proceso que las está generando a fin de reducir dicha tasa. La línea punteada B corresponde a una tasa de fallas de página tan baja que podemos concluir que el proceso tiene demasiada memoria. En este caso, podemos quitarle marcos de página. Así, PFF trata de mantener la tasa de paginación dentro de límites aceptables.

Si el algoritmo descubre que hay tantos procesos en la memoria que no es posible mantenerlos a todos por debajo de A, se eliminará algún proceso de la memoria y sus marcos de página se repartirán entre los procesos restantes o se colocarán en una reserva de páginas disponibles que podrán usarse en fallas de página subsecuentes. La decisión de sacar a un proceso de la memoria es una forma de control de carga. Esto pone de manifiesto que, incluso con paginación, todavía es necesario el intercambio, sólo que ahora se usa para reducir la demanda potencial de memoria, no para recuperar sus bloques y usarlos de inmediato.

4.5.3 Tamaño de página

En muchos casos, el tamaño de página es un parámetro que el sistema operativo puede escoger. Incluso si el hardware se diseñó con páginas de, por ejemplo, 512 bytes, el sistema operativo fácilmente puede considerar las páginas 0 y 1, 2 y 3, 4 y 5, etc., como páginas de 1K asignando siempre dos marcos de página de 512 bytes consecutivos a ellas.

La determinación del tamaño de página óptimo requiere balancear varios factores opuestos. Por principio, un segmento de texto, datos o pila escogido al azar no llena un número entero de

páginas. En promedio, la mitad de la página final estará vacía. El espacio extra de esa página se desperdicia, y este desperdicio se denomina fragmentación interna. Con n segmentos en memoria y un tamaño de página de p bytes, se desperdiciarán $np/2$ bytes por fragmentación interna. Este razonamiento es un argumento a favor de las páginas pequeñas.

Otro argumento a favor de un tamaño de página pequeño se hace evidente si pensamos en un programa que consiste en ocho fases secuenciales de 4K cada una. Con un tamaño de página 32K, habría que asignar al programa 32K todo el tiempo. Con un tamaño de página de 16K programa sólo necesitaría 16K. Con un tamaño de página de 4K o menor, el programa se requeriría 4K en un momento dado. En general, un tamaño de página grande hará que haya proporción mayor de programa no utilizado en la memoria que si se usan páginas pequeñas.

Por otro lado, el empleo de páginas pequeñas implica que los programas necesitarán muchas páginas, y la tabla de páginas será grande. Un programa de 32K sólo necesita cuatro páginas 8K, pero 64 páginas de 512 bytes. Las transferencias entre la memoria y el disco generalmente son de páginas completas, y la mayor parte del tiempo se invierte en la búsqueda y el retardo rotacional, de modo que la transferencia de una página pequeña toma casi tanto tiempo como en una página grande. Podrían requerirse 64×15 ms para cargar 64 páginas de 512 bytes, pero sólo 4×25 ms para cargar cuatro páginas de 8K.

En algunas máquinas, la tabla de páginas debe cargarse en registros de hardware cada que la CPU conmuta de un proceso a otro. En estas máquinas, tener un tamaño de página pequeña implica que el tiempo requerido para cargar los registros de páginas aumenta conforme el tamaño de página disminuye. Además, el espacio ocupado por la tabla de páginas aumenta al disminuir el tamaño de página.

Este último punto puede analizarse matemáticamente. Sea s el tamaño de proceso promedio en bytes y p el tamaño de página en bytes. Además, suponga que cada entrada de página requiere s/p bytes. El número apropiado de páginas requeridas por proceso es entonces s/p , que ocupan p bytes en la tabla de páginas. La memoria desperdiciada en la última página del proceso debido a la fragmentación interna es $p/2$. Por tanto, el gasto extra total debido a la tabla de páginas y a la pérdida por fragmentación interna está dada por

$$\text{gasto extra} = selp + p/2$$

El primer término (tamaño de la tabla de páginas) es grande cuando el tamaño de página pequeño. El segundo término (fragmentación interna) es grande cuando el tamaño de página es grande. El tamaño óptimo debe estar en algún punto intermedio. Si obtenemos la primera derivada respecto a p y la igualamos a cero, tenemos la ecuación

$$-selp^2 + 1/2=0$$

De esta ecuación podemos deducir una fórmula que da el tamaño de página óptimo (considerando sólo la memoria desperdiciada por fragmentación y el tamaño de la tabla de páginas). El resultado

$$p = \sqrt{2se}$$

Para $s = 128K$ y $e = 8$ bytes por entrada de tabla de páginas, el tamaño de página óptimo es de 1448 bytes. En la práctica se usaría 1K o 2K, dependiendo de los demás factores (p. ej., rapidez del disco). La mayor parte de las computadoras comerciales emplean tamaños de página que van desde 512 bytes hasta 64K.

4.5.4 Interfaz de memoria virtual

Hasta ahora, hemos supuesto que la memoria virtual es transparente para los procesos y los programadores. Esto es, todo lo que ellos ven es un espacio de direcciones virtual grande en una computadora con una memoria física (más) pequeña. En muchos sistemas, esto es cierto, pero en algunos sistemas avanzados los programadores tienen cierto control sobre el mapa de memoria y lo pueden utilizar de formas no tradicionales. En esta sección veremos brevemente algunas de ellas.

Una razón para dar a los programadores control sobre su mapa de memoria es permitir que dos o más procesos compartan la misma memoria. Si los programadores pueden nombrar regiones de su memoria, un proceso puede darle a otro el nombre de una región de memoria para que pueda incluirla en su mapa. Cuando dos (o más) procesos comparten las mismas páginas, se hace posible la compartición de alto ancho de banda: un proceso escribe en la memoria compartida y el otro la lee.

La compartición de páginas también puede servir para implementar un sistema de transferencia de páginas de alto rendimiento. Normalmente, cuando se transfieren mensajes, los datos se copian de un espacio de direcciones a otro, pagando un precio considerable. Si los procesos pueden controlar su mapa de páginas, se puede transferir un mensaje haciendo que el proceso transmisor excluya de su mapa la página o páginas que contienen el mensaje, y que el proceso receptor las incluya en su mapa. Aquí sólo tendrían que copiarse los nombres de página, no todos los datos.

Otra técnica de administración avanzada de memoria es la **memoria compartida distribuida** (Feeley et al., 1995; Li y Hudak, 1989; Zekauskas et al., 1994). De lo que se trata aquí es de permitir que múltiples procesos en una red compartan un conjunto de páginas, posiblemente, aunque no necesariamente, como un solo espacio de direcciones lineal compartido. Cuando un proceso hace referencia a una página que actualmente no está en su mapa, genera una falla de página. El manejador de fallas de página, que puede estar en el kernel o en el espacio de usuario, localiza entonces la máquina que tiene la página y le envía un mensaje pidiéndole que la excluya de su mapa y la envíe por la red. Cuando llega la página, se incluye en el mapa y se reinicia la instrucción que falló.

4.6 SEGMENTACIÓN

La memoria virtual de la que hemos hablado hasta ahora es unidimensional porque las direcciones virtuales van desde 0 hasta alguna dirección máxima, una dirección tras otra. Para muchos problemas, tener dos o más espacios de direcciones virtuales independientes puede ser mucho mejor

que tener sólo uno. Por ejemplo, un compilador tiene muchas tablas que se construyen conforme procede la compilación, y que pueden incluir

1. El texto fuente que se está guardando para el listado impreso (en sistemas por lotes).
2. La tabla de símbolos, que contiene los nombres y los atributos de las variables.
3. La tabla que contiene todas las constantes enteras y de punto flotante empleadas.
4. El árbol de análisis sintáctico, que condensa el árbol de análisis sintáctico del programa.
5. La pila empleada para llamadas a procedimientos dentro del compilador.

Las primeras cuatro tablas crecen continuamente conforme avanza la compilación. La última crece y se encoge de forma impredecible durante la compilación. En una memoria unidimensional, habría que asignar a estas cinco tablas trozos contiguos del espacio de direcciones virtual, como en la Fig.4-19.

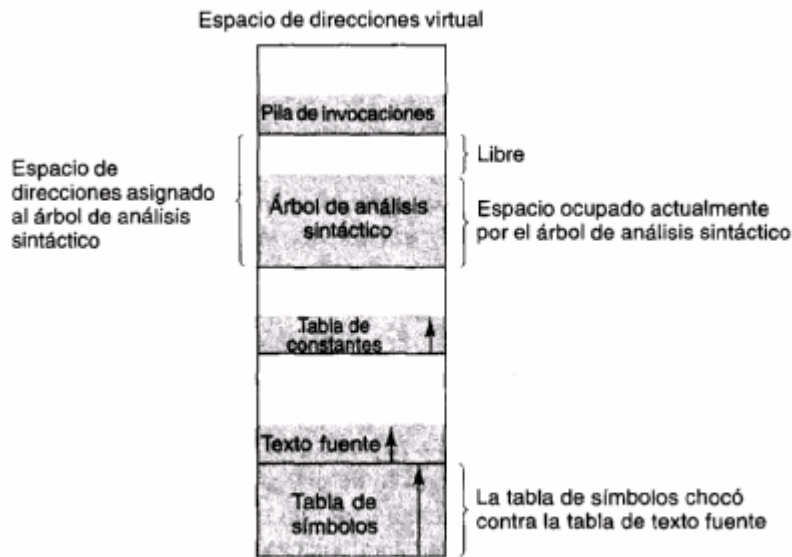


Figura 4-19. En un espacio de direcciones unidimensional con tablas crecientes, una tabla podría chocar contra otra.

Consideremos lo que sucede si un programa tiene un número excepcionalmente grande de variables. El trozo del espacio de direcciones asignado a la tabla de símbolos podría llenarse, pero podría haber espacio de sobra para otras tablas. Desde luego, el compilador podría limitarse a exhibir un mensaje diciendo que la compilación no puede continuar debido a un exceso de variables, pero esto no parece justo si hay espacio desocupado en las otras tablas.

Otra posibilidad es hacerla de Robin Hood, tomando espacio de las tablas que tienen mucho espacio y dándolo a las que tienen poco. Es posible efectuar estos movimientos, pero sería

análogo a administrar las propias superposiciones: una molestia en el mejor de los casos y una gran cantidad de trabajo tedioso y fútil en el peor.

Lo que realmente se necesita es una forma de liberar al programador de la tarea de administrar las tablas en expansión y contracción, del mismo modo como la memoria virtual elimina la preocupación de tener que organizar el programa en superposiciones.

Una solución directa y en extremo general consiste en proveer a la máquina con muchos espacios de direcciones completamente independientes, llamados segmentos. Cada segmento consiste en una secuencia lineal de direcciones, desde 0 hasta algún máximo. La longitud de cada segmento puede ser cualquiera desde 0 hasta el máximo permitido. Los diferentes segmentos pueden, y suelen, tener diferentes longitudes. Es más, la longitud de los segmentos puede cambiar durante la ejecución. La longitud de un segmento de pila puede aumentarse cada vez que algo se empila y reducirse cada vez que algo se desempila.

Puesto que cada segmento constituye un espacio de direcciones aparte, los distintos segmentos pueden crecer o encogerse de forma independiente, sin afectarse entre sí. Si una pila de cierto segmento necesita más espacio de direcciones para crecer, se le puede conceder, porque no hay nada más en su espacio de direcciones con lo que pueda chocar. Desde luego, podría llenarse un segmento, pero los segmentos suelen ser muy grandes, así que esta ocurrencia es poco común. Para especificar una dirección en esta memoria segmentada o bidimensional, el programa debe proporcionar una dirección de dos partes: un número de segmento y una dirección dentro de ese segmento. En la Fig. 4-20 se ilustra una memoria segmentada empleada para las tablas de compilador que vimos antes.

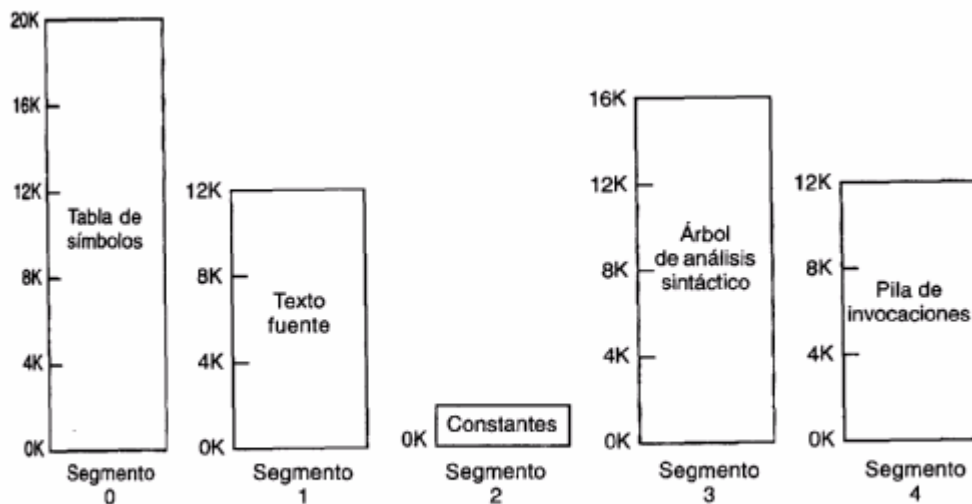


Figura 4-20. Una memoria segmentada permite a cada tabla crecer o encogerse con independencia de las demás tablas.

Subrayamos que un segmento es una entidad lógica, de la cual el programador está consciente y que utiliza como entidad lógica. Un segmento podría contener un procedimiento, un arreglo,

una pila o una colección de variables escalares, pero por lo regular no contiene una mezcla diferentes cosas.

Las memorias segmentadas tienen otras ventajas además de simplificar el manejo de estructuras de datos que están creciendo o encogiéndose. Si cada procedimiento ocupa un segmento aparte, con la dirección 0 como dirección inicial, se simplifica considerablemente la vinculación de procedimientos compilados por separado. Una vez que se han compilado y vinculado todos 1 procedimientos que constituyen un programa, una llamada al procedimiento del segmento n usa la dirección de dos partes (n, 0) para direccionar la palabra 0 (el punto de entrada).

Si el procedimiento que está en el segmento n se modifica y recompila subsecuentemente, i hay necesidad de alterar ningún otro procedimiento (porque no se modificó ninguna dirección i inicio), incluso si la nueva versión es más grande que la anterior. Con una memoria unidimensional los procedimientos se empaacan uno junto a otro, sin espacio de direcciones entre ellos. En consecuencia, la modificación del tamaño de un procedimiento puede afectar la dirección de inicio i otros procedimientos que no tienen relación con él. Esto, a su vez, requiere la modificación de todos los procedimientos que invocan a cualquiera de los procedimientos que se moviere incorporando sus nuevas direcciones de inicio. Si un programa contiene cientos de procedimiento este proceso puede ser costoso.

La segmentación también facilita compartir procedimientos o datos entre varios procese Un ejemplo común es la biblioteca compartida. Las estaciones de trabajo modernas que ejecute sistemas de ventanas avanzados suelen tener bibliotecas de gráficos extremadamente grandes compiladas en casi todos sus programas. En un sistema segmentado, la biblioteca de gráficos puede colocar en un segmento compartido por múltiples procesos, eliminando la necesidad \ tenerla en el espacio de direcciones de cada proceso. Si bien también es posible tener bibliotecas compartidas en los sistemas con paginación pura, resulta mucho más complicado. De hecho esos sistemas lo logran simulando segmentación.

Dado que cada segmento constituye una entidad lógica de la cual el programador está consciente, como un procedimiento, un arreglo o una pila, los diferentes segmentos pueden tener diferentes tipos de protección. Un segmento de procedimientos se puede especificar como so de ejecución, prohibiendo los intentos por leerlo o escribir en él. Un arreglo de punto flotante puede especificar como de lectura/escritura pero no de ejecución, y los intentos por saltar a serán atrapados. Semejante protección es útil para detectar errores de programación.

Conviene entender por qué la protección tiene sentido en una memoria segmentada pero i en una memoria paginada unidimensional. En una memoria segmentada el usuario sabe qué hay (cada segmento. Normalmente, un segmento no contendría un procedimiento y una pila, p ejemplo, sino una cosa o la otra. Puesto que cada segmento contiene sólo una clase de objetos, segmento puede tener la protección apropiada para ese tipo en particular. La paginación y la segmentación se comparan en la Fig. 4-21.

El contenido de una página es, en cierto sentido, accidental. El programador no está consciente siquiera del hecho de que está ocurriendo la paginación. Aunque sería posible colocar unos cuantos bits en cada entrada de la tabla de páginas para especificar el acceso permitido, para aprovechar esta capacidad el programador tendría que saber en dónde están las fronteras de página dentro de su espacio de direcciones. Ésta es precisamente la clase de administración que

Consideración	Paginación	Segmentación
¿El programador necesita estar consciente de que se está usando esta técnica?	No	Sí
¿Cuántos espacios de direcciones lineales hay?	1	Muchos
¿El espacio de direcciones total puede exceder el tamaño de la memoria física?	Sí	Sí
¿Se pueden distinguir los procedimientos de los datos y protegerse de forma independiente?	No	Sí
¿Se pueden manejar fácilmente tablas cuyo tamaño fluctúa?	No	Sí
¿Se facilita la compartición de procedimientos entre usuarios?	No	Sí
¿Por qué se inventó esta técnica?	Para tener un espacio de direcciones lineal grande sin tener que adquirir más memoria física.	Para poder dividir los programas y los datos en espacios de direcciones lógicamente independientes y facilitar la compartición y la protección.

Figura 4-21. Comparación de paginación y segmentación.

se buscaba eliminar al inventar la paginación. Como el usuario de una memoria segmentada tiene la ilusión de que todos los segmentos están en la memoria principal todo el tiempo —es decir, puede direccionarlos como si así fuera— puede proteger cada segmento individualmente, sin tener que preocuparse por la administración que implica superponerlos.

4.6.1 Implementación de la segmentación pura

La implementación de la segmentación difiere de la de la paginación en un aspecto esencial: el tamaño de las páginas es fijo pero el de los segmentos no. En la Fig. 4-22(a) se muestra un ejemplo de memoria física que inicialmente contiene cinco segmentos. Consideremos ahora qué sucede si se desaloja el segmento 1 y se coloca en su lugar el segmento 7, que es más pequeño. Llegamos a la configuración de memoria de la Fig. 4-22(b). Entre el segmento 7 y el segmento 2 hay un área desocupada, es decir, un agujero. Luego el segmento 4 es sustituido por el segmento 5, como en la Fig. 4-22(c), y el segmento 3 es reemplazado por el segmento 6, como en la Fig. 4-22(d). Después de cierto tiempo de ejecución del sistema, la memoria estará dividida en varios trozos, unos con segmentos y otros con agujeros. Este fenómeno, llamado cuadrícula o fragmentación externa, desperdicia memoria en los agujeros. El remedio es la compactación, como se muestra en la Fig. 4-22(e).

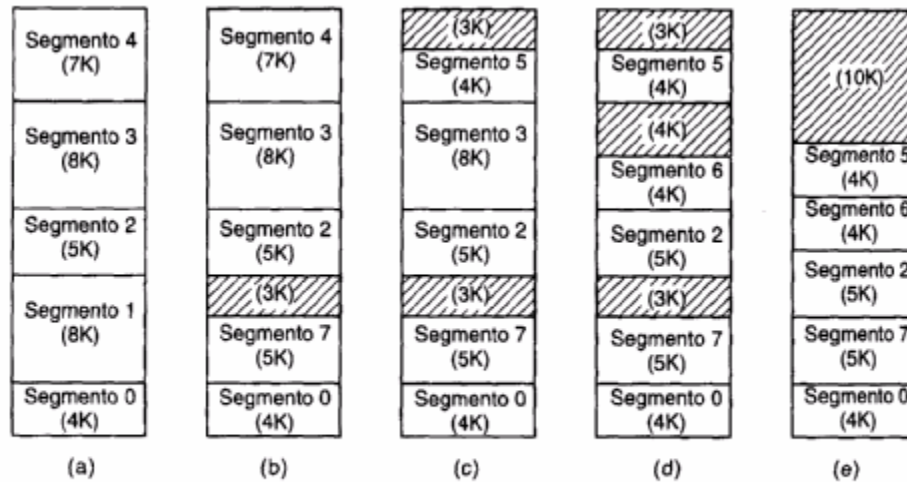


Figura 4-22. (a)-(d) Desarrollo de la cuadrícula. (e) Eliminación de la cuadrícula por compactación.

4.6.2 Segmentación con paginación: MULTICS

Si los segmentos son grandes, puede ser problemático, o incluso imposible, mantenerlos completos en la memoria principal. Esto nos lleva a la idea de paginarlos, de modo que sólo se conserven las páginas que realmente se necesitan. Varios sistemas importantes han manejado segmentos paginados. En esta sección describiremos el primero: MULTICS. En la siguiente veremos uno más reciente: Pentium de Intel.

MULTICS se ejecutaba en las máquinas Honeywell 6000 y sus descendientes y proporcionaba a cada programa una memoria virtual de hasta 218 segmentos (más de 250 000), cada uno de los cuales podía tener una longitud de hasta 65 536 palabras (de 36 bits). Para implementar esto, los diseñadores de MULTICS decidieron tratar cada segmento como una memoria virtual y paginarlo, combinando las ventajas de la paginación (tamaño de página uniforme y no tener que mantener todo el segmento en la memoria si sólo se está usando una parte) con las de la segmentación (facilidad de programación, modularidad, protección y compartición).

Cada programa MULTICS tiene una tabla de segmentos, con un descriptor por segmento. Puesto que puede haber más de un cuarto de millón de entradas en la tabla, la tabla de segmentos es en sí un segmento y se pagina. Un descriptor de segmento contiene una indicación de si el segmento está en la memoria principal o no. Si cualquier parte del segmento está en la memoria, se considera que el segmento está en la memoria, y su tabla de páginas estará en la memoria. Si el segmento está en la memoria, su descriptor contiene un apuntador de 18 bits a su tabla de páginas [véase la Fig. 4-23(a)]. Dado que las direcciones físicas tienen 24 bits y las páginas están alineadas según fronteras de 64 bytes (lo que implica que los seis bits de orden bajo de las direcciones de página son 000000), sólo se requieren 18 bits en el descriptor para almacenar una dirección de tabla de páginas. El descriptor también contiene el tamaño del segmento, los bits de protección y

unas cuantas cosas más. La Fig. 4-23(b) ilustra un descriptor de segmento MULTICS. La dirección del segmento en la memoria secundaria no está en el descriptor de segmento sino en otra tabla utilizada por el manejador de fallas de segmento.

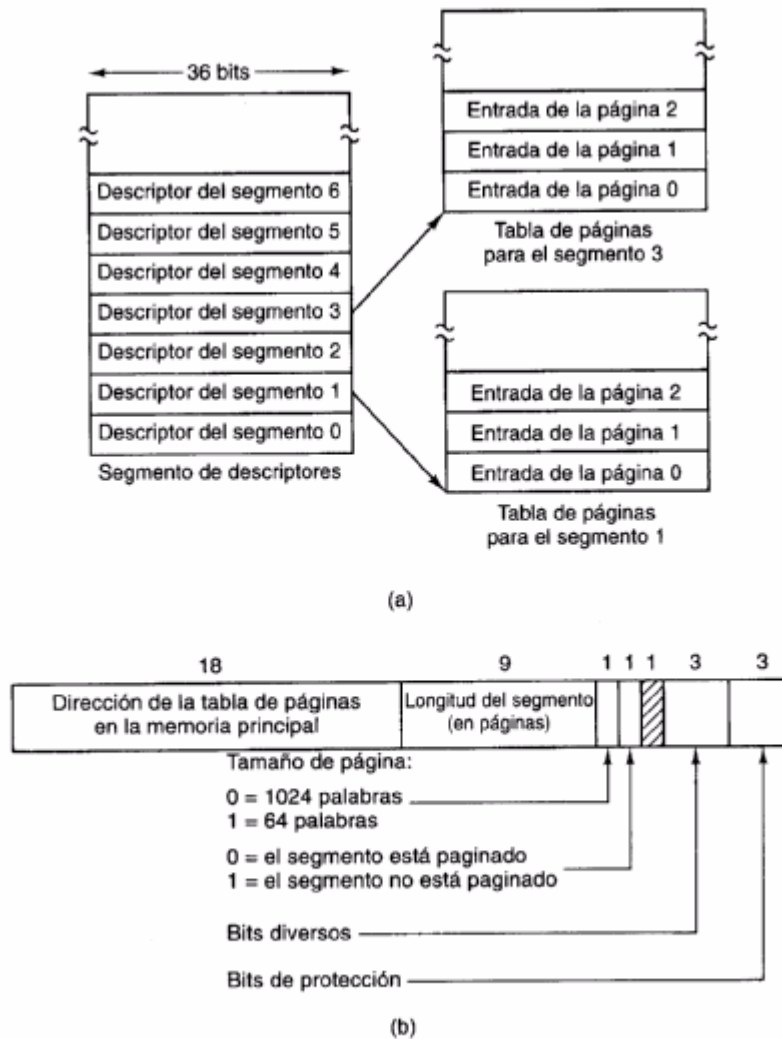


Figura 4-23. La memoria virtual MULTICS. (a) El segmento de descriptores apunta a las tablas de páginas. (b) Un descriptor de segmento. Los números son longitudes de campo.

Cada segmento es un espacio de direcciones virtual ordinario y se pagina del mismo modo que la memoria paginada no segmentada que se describió en una sección anterior de este capítulo. El tamaño de página normal es de 1024 palabras (aunque unos cuantos segmentos pequeños que utiliza MULTICS mismo no están paginados o se pagan en unidades de 64 palabras a fin de ahorrar memoria física).

Una dirección en MULTICS consta de dos partes: el segmento y la dirección dentro del segmento. La dirección dentro del segmento se subdivide en un número de página y una palabra dentro de la página, como se muestra en la Fig. 4-24. Cuando ocurre una referencia a la memoria, se lleva a cabo el siguiente algoritmo.

1. Se usa el número de segmento para encontrar el descriptor de segmento.
2. Se verifica si la tabla de páginas del segmento está en la memoria. Si es así, se le localiza; si no, ocurre una falla de segmento. Si hay una violación de la protección, ocurre una falla (trampa).
3. Se examina la entrada de tabla de páginas que corresponde a la página virtual solicitada. Si la página no está en la memoria, ocurre una falla de página; si está en la memoria, se extrae de la entrada de la tabla de páginas la dirección de principio de la página en la memoria principal.
4. Se suma la distancia al origen de la página para obtener la dirección en la memoria principal donde se encuentra la palabra.
5. Finalmente se efectúa la lectura o el almacenamiento.

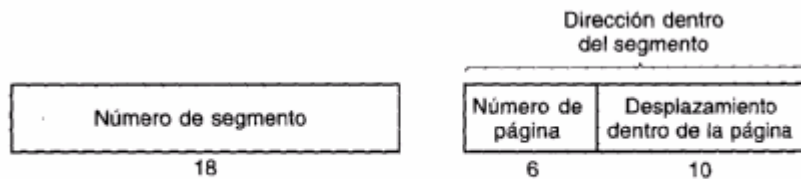


Figura 4-24. Dirección virtual MULTICS de 34 bits.

Este proceso se ilustra en la Fig. 4-25. Por sencillez, se omite el hecho de que el segmento de descriptores mismo también está paginado. Lo que realmente sucede es que se usa un registro (el registro base de descriptores) para localizar la tabla de páginas del segmento de descriptores, la cual, a su vez, apunta a las páginas del segmento de descriptores. Una vez encontrado el descriptor del segmento requerido, el direccionamiento procede tal como se muestra en la Fig. 4-25.

Como sin duda el lector ya adivinó, si el sistema operativo ejecutara el algoritmo anterior en cada instrucción, los programas no se ejecutarían con mucha rapidez. En realidad, el hardware de MULTICS contiene un TLB de alta velocidad de 16 palabras que puede buscar una clave dada en paralelo en todas sus entradas. Este TLB se ilustra en la Fig. 4-26. Cuando se presenta una dirección a la computadora, lo primero que hace el hardware de direccionamiento es ver si la dirección virtual está en el TLB. Si es así, obtiene el número de marco de página directamente del TLB y forma la dirección real de la palabra referida sin tener que examinar el segmento de descriptores ni la tabla de páginas.

Las direcciones de las 16 páginas a las que se hizo referencia más recientemente se mantienen en el TLB. Los programas cuyo conjunto de trabajo es menor que el tamaño del TLB alcanzan un equilibrio con todo el conjunto de trabajo en el TLB y por tanto se ejecutan de forma muy

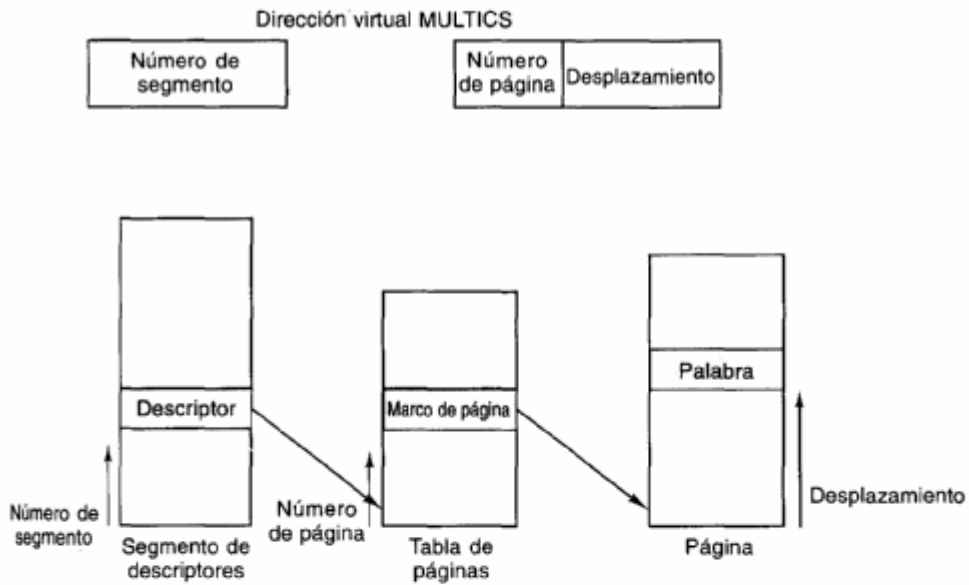


Figura 4-25. Conversión de una dirección MULTICS de dos partes en una dirección de memoria principal.

Campo de comparación				Protección	Edad	¿Se usa esta entrada?
Número de segmento	Página virtual	Marco de página				
4	1	7	Leer/escibir	13	1	
6	0	2	Sólo leer	10	1	
12	3	1	Leer/escibir	2	1	
					0	
2	1	0	Sólo ejecutar	7	1	
2	2	12	Sólo ejecutar	9	1	

Figura 4-26. Versión simplificada del TLB de MULTICS. La existencia de dos tamaños de página hace que el TLB real sea más complicado.

eficiente. Si la página no está en el TLB, se hace referencia a las tablas de descriptores y de páginas para encontrar la dirección del marco de página, y el TLB se actualiza de modo que: incluya esta página, desalojando la página menos recientemente utilizada. El campo de edad indica I cuál entrada es la que se usó menos recientemente. La razón por la que se usa un TLB es poder | comparar el número de segmento y el número de página de todas las entradas en paralelo.

4.6.3 Segmentación con paginación: El Pentium de Intel

En muchos sentidos, la memoria virtual del Pentium (y Pentium Pro) se parece a la de MULTIC incluida la presencia tanto de segmentación como de paginación. Mientras que MULTICS tiene 256K segmentos independientes, cada uno con hasta 64K palabras de 36 bits, el Pentium tiene 16 segmentos independientes, cada uno con hasta mil millones de palabras de 32 bits. Aunque hi menos segmentos, el tamaño tan grande de los segmentos es mucho más importante, ya que pocos programas necesitan más de 1000 segmentos, pero muchos programas requieren segmentos con capacidad de megabytes.

El corazón de la memoria virtual Pentium consiste en dos tablas, **la LDT (tabla de descriptor local)** y **la GDT (tabla de descriptores global)**. Cada programa tiene su propia LDT, pero sol hay una GDT, compartida por todos los programas de la computadora. La LDT describe k segmentos que son locales para cada programa, incluidos su código, datos, pila, etc., en tanto que la GDT describe los segmentos del sistema, incluido el sistema operativo mismo.

Para acceder a un segmento, lo primero que hace un programa Pentium es cargar un selecto para ese segmento en uno de los seis registros de segmento de la máquina. Durante la ejecución el registro CS contiene el selector para el segmento de código y el registro DS contiene el selecto para el segmento de datos. Los otros registros de segmento son menos importantes. Cada selector es un número de 16 bits, como se muestra en la Fig. 4-27.

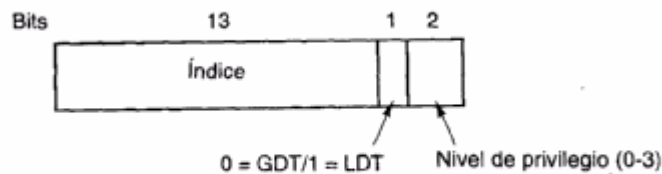


Figura 4-27. Un selector Pentium.

Uno de los bits selectores indica si el segmento es local o global (es decir, si está en la LDT o en la GDT). Trece bits más especifican el número de entrada en la LDT o la GDT, de modo que estas tablas están restringidas a contener cada una 8K descriptores de segmento. Los otros dos bits tienen que ver con la protección, y se describirán posteriormente. El descriptor 0 está prohibido; puede cargarse sin peligro en un registro de segmento para indicar que ese registro no esta disponible actualmente, pero causa una trampa si se usa.

En el momento en que un selector se carga en un registro de segmento, el descriptor correspondiente se trae de la LDT o GDT y se almacena en registros de microprograma, a fin de poder acceder a él rápidamente. Un descriptor consiste en ocho bytes, que incluyen la dirección base del segmento, el tamaño y otra información, como se ilustra en la Fig. 4-28.

El formato del selector se escogió ingeniosamente de modo que facilitara la localización del descriptor. Primero se selecciona ya sea la LDT o la GDT, con base en el bit 2 del selector. Luego se copia el selector en un registro temporal interno y se ponen en cero los tres bits de orden bajo Por último, se le suma la dirección de la tabla LDT o GDT, para dar un apuntador directo al

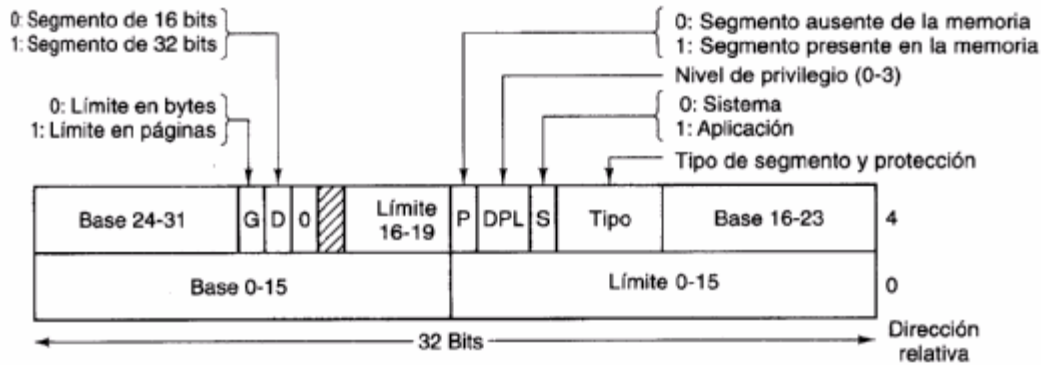


Figura 4-28. Descriptor de segmento de código Pentium. Los de segmentos de datos son un poco diferentes.

descriptor. Por ejemplo, el selector 72 se refiere a la entrada 9 de la GDT, que se encuentra en la dirección $GDT + 72$.

Sigamos uno a uno los pasos mediante los cuales un par (selector, distancia) se convierte en una dirección física. Tan pronto como el microprograma sabe cuál registro de segmento se está usando, puede encontrar el descriptor completo que corresponde a ese selector en sus registros internos. Si el segmento no existe (selector 0), o sus páginas no están actualmente en la memoria, ocurre una trampa.

Luego se verifica si la distancia rebasa el final del segmento, en cuyo caso también ocurre una trampa. Lógicamente, debería haber un campo de 32 bits en el descriptor para indicar el tamaño del segmento, pero sólo hay 20 bits disponibles, así que se emplea un esquema distinto. Si el campo Gbit (granularidad) es 0, el campo Límite es el tamaño exacto del segmento, hasta 1 MB; si es 1, el campo Límite da el tamaño de segmento en páginas en lugar de bytes. El tamaño de página de Pentium está fijo en 4K bytes, así que bastan 20 bits para segmentos de hasta 232 bytes.

Suponiendo que el segmento está en la memoria y que la distancia está dentro del intervalo, el Pentium suma entonces el campo Base de 32 bits del descriptor a la distancia para formar lo que se denomina **dirección lineal**, como se muestra en la Fig. 4-29. El campo Base se divide en tres partes que se dispersan por el descriptor para mantener la compatibilidad con el 286, en el cual 1: Base sólo tiene 24 bits. En efecto, el campo Base permite que cada segmento comience en un y lugar arbitrario dentro del espacio de direcciones lineal de 32 bits.

Si se inhabilita la paginación (con un bit de un registro de control global), la dirección lineal se interpreta como la dirección física y se envía a la memoria para la lectura o escritura. Así, con la paginación inhabilitada, tenemos un esquema de segmentación puro, con la dirección base de cada segmento dada en su descriptor. Por cierto, se permite que los segmentos se traslapen, probablemente porque implicaría demasiado trabajo y pérdida de tiempo verificar que todos sean mutuamente exclusivos.

Por otro lado, si está habilitada la paginación, la dirección lineal se interpreta como dirección virtual y se transforma en la dirección física usando tablas de páginas, más o menos como en los ejemplos anteriores. La única complicación real es que con una dirección virtual de 32

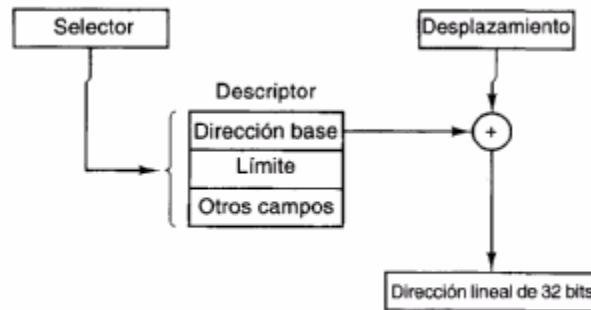


Figura 4-29. Conversión de un par (selector, desplazamiento) en una dirección lineal.

bits y páginas de 4K, un segmento podría contener un millón de páginas, así que se usa transformación de dos niveles para reducir el tamaño de las tablas de páginas cuando los segmentos son pequeños. Cada programa en ejecución tiene un **directorio de páginas** que consiste en 1024 entradas de 32 bits cada una. Este directorio está en una dirección a la que apunta un registro global. Cada entrada de este directorio apunta a una tabla de páginas que también contiene 1024 entradas de 32 bits. Las entradas de la tabla de páginas apuntan a marcos de páginas. El esquema se muestra en la Fig. 4-30

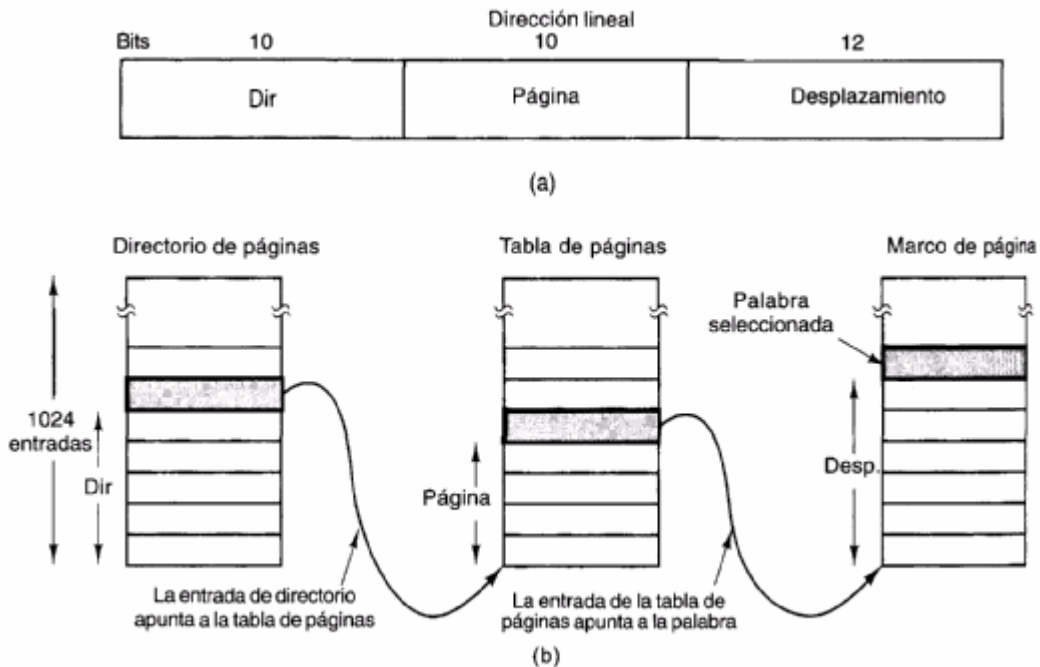


Figura 4-30. Transformación de una dirección lineal en una dirección física.

En la Fig. 4-30(a) vemos una dirección lineal dividida en tres campos, Dir, Página y DiSK campo Dir sirve como índice del directorio de páginas para encontrar un apuntador a la tabhj páginas apropiada. Luego se usa el campo Página como índice de la tabla de páginas para encontrar

la dirección física del marco de página. Por último, se suma Dist a la dirección de marco de página para obtener la dirección física del byte o palabra requerido.

Las entradas de la tabla de páginas tienen 32 bits cada una, 20 de los cuales contienen un número de marco de página. Los bits restantes contienen bits de acceso y sucios, encendidos por el hardware para beneficio del sistema operativo, bits de protección y otros bits de utilería.

Cada tabla de páginas tiene entradas para 1024 marcos de página de 4K, así que una sola tabla de páginas maneja 4 megabytes de memoria. Un segmento de menos de 4M tendrá un directorio de páginas con una sola entrada, un apuntador a su única tabla de páginas. De este modo, el gasto extra para los segmentos cortos es de sólo dos páginas, en lugar del millón de páginas que se requerirían en una tabla de páginas de un solo nivel.

A fin de evitar hacer referencias repetidas a la memoria, el Pentium, al igual que MULTICS, tiene un pequeño TLB que transforma directamente las combinaciones Dir-Página más recientemente utilizadas en la dirección física del marco de página. Sólo si la combinación actual no está presente en el TLB se aplica el mecanismo de la Fig. 4-30 y se actualiza el TLB.

Si pensamos un poco veremos que, si se usa paginación, realmente no tiene caso disponer de un campo Base distinto de cero en el descriptor. Todo lo que Base hace es indicar una distancia pequeña para usar una entrada en medio del directorio de páginas, en lugar de al principio. La verdadera razón para incluir Base es permitir la segmentación pura (no paginada), y por compatibilidad con el 286, que siempre tiene la paginación inhabilitada (es decir, el 286 tiene sólo segmentación pura, sin paginación).

También vale la pena señalar que si alguna aplicación no necesita segmentación y está satisfecha con un solo espacio de direcciones de 32 bits paginado, es posible emplear ese modelo. Todos los registros de segmento se pueden llenar con el mismo selector, cuyo descriptor tiene Base = 0 y Límite igual al máximo. La distancia de instrucción será entonces la dirección lineal, utilizando un solo espacio de direcciones; en efecto, paginación normal.

Considerando todo, hay que felicitar a los diseñadores de Pentium. Dados los objetivos opuestos de implementar paginación pura, segmentación pura y segmentos paginados, y al mismo tiempo mantener la compatibilidad con el 286, y además hacer todo esto de manera eficiente, el diseño resultante es sorprendentemente sencillo y aseado.

Aunque hemos cubierto la arquitectura completa de la memoria virtual Pentium, si bien brevemente, vale la pena decir unas palabras acerca de la protección, ya que este tema está íntimamente relacionado con la memoria virtual. Así como el esquema de memoria virtual sigue de cerca el modelo MULTICS, lo mismo sucede con el sistema de protección. El Pentium maneja cuatro niveles de protección, siendo el 0 el más privilegiado y el 3 el menos privilegiado. Éstos se muestran en la Fig. 4-31. En cada instante, un programa en ejecución está en un nivel dado, indicado por un campo de dos bits en su PSW. Cada segmento del sistema también tiene un nivel.

En tanto un programa se limite a usar segmentos en su propio nivel, todo funcionará perfectamente. Se permiten intentos por acceder a datos en un nivel superior, pero los intentos por acceder a datos en un nivel inferior son ilegales y causan trampas. Los intentos por invocar procedimientos en un nivel distinto (más alto o más bajo) están permitidos, pero de una forma cuidadosamente controlada. Para efectuar una llamada intemivel, la instrucción CALL debe contener un selector en lugar de una dirección. Este selector designa un descriptor llamado puerta de

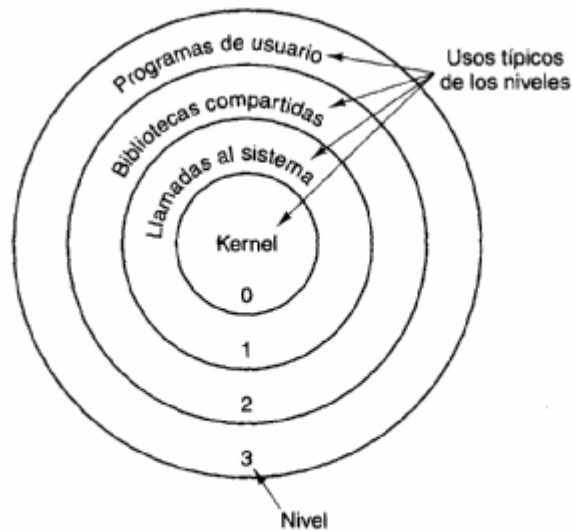


Figura 4-31. Protección en el Pentium.

llamada que da la dirección del procedimiento por invocar. Por tanto, no es posible saltar a mitad de un segmento de código arbitrario en un nivel distinto. Sólo pueden usarse los puntos de entrada oficiales. Los conceptos de niveles de protección y puertas de llamada se aplicaron por primera vez en MULTICS, donde se veían como **anillos de protección**.

Una aplicación típica de este mecanismo se sugiere en la Fig. 4-31. En el nivel 0, encontramos el kernel del sistema operativo, que se encarga de E/S, administración de memoria y otras cuestiones críticas. En el nivel 1 está presente el manejador de llamadas al sistema. Los programas usuario pueden invocar procedimientos aquí para que se ejecuten llamadas al sistema, pero si puede invocarse una lista específica y protegida de procedimientos. El nivel 2 contiene procedimientos de biblioteca, posiblemente compartidos entre muchos programas en ejecución. Los programas usuario pueden invocar estos procedimientos y leer sus datos, pero no pueden modificarlos por último, los programas de usuario se ejecutan en el nivel 3, que tiene la menor protección.

Las trampas y las interrupciones emplean un mecanismo similar a las puertas de llamada; el también hacen referencia a descriptores, en lugar de direcciones absolutas, y estos descriptores apuntan a procedimientos específicos que han de ejecutarse. El campo Tipo de la Fig. 4- distingue entre segmentos de código, segmentos de datos y las diversas clases de puertas.

4.7 GENERALIDADES DE ADMINISTRACIÓN DE MEMORIA EN MINIX

La administración de memoria en MINIX es sencilla: no se emplea ni paginación ni intercambio. El administrador de memoria mantiene una lista de agujeros ordenados en orden por dirección memoria. Cuando se necesita memoria, sea a causa de una llamada al sistema FORK o EXBC,i

examina la lista de agujeros empleando primer ajuste en busca de un agujero con el tamaño suficiente. Una vez que un proceso se ha colocado en la memoria, permanece en el mismo lugar hasta terminar; nunca se intercambia a disco y nunca se traslada a otro lugar de la memoria. Además, el área asignada nunca crece ni se encoge.

Esta estrategia amerita una explicación. Son tres los factores que la justifican: (1) la idea de que MINIX es para computadoras personales, no para sistemas grandes de tiempo compartido, (2) el deseo de hacer que MINIX funcione en todas las IBM PC y (3) el deseo de facilitar la implementación del sistema en otras computadoras pequeñas.

El primer factor implica que, en promedio, el número de procesos en ejecución será pequeño, y por lo regular habrá suficiente memoria para contener todos los procesos, con espacio de sobra. En tal caso no se necesitaría intercambiar. Dado que el intercambio hace más complejo el sistema, su ausencia da lugar a código más sencillo.

El deseo de hacer que MINIX se ejecute en todas las computadoras compatibles con IBM PC también tuvo un impacto considerable sobre el diseño de la administración de memoria. Los sistemas más sencillos de esta familia usan el procesador 8088, cuya arquitectura de administración de memoria es muy primitiva. Este procesador no reconoce la memoria virtual en ninguna de sus formas y ni siquiera detecta un desbordamiento de pila, defecto que tiene implicaciones importantes para la forma en que se organizan los procesos en la memoria. Estas limitaciones no existen en diseños posteriores que usan los procesadores 80386, 80486 o Pentium. Sin embargo, si aprovechara estas capacidades MINIX sería incompatible con muchas máquinas del extremo inferior que todavía están en uso.

El aspecto de transportabilidad es un argumento en favor de un esquema de administración de memoria lo más sencillo posible. Si MINIX usara paginación o segmentación, sería difícil, si no imposible, trasladarlo a máquinas que no contaran con estas capacidades. Al postular un mínimo de supuestos respecto a lo que el hardware puede hacer, se multiplica el número de las máquinas a las que MINIX puede trasladarse.

Otro aspecto inusual de MINIX es la forma como se implementa la administración de memoria. Esta función no forma parte del kernel, sino que corre por cuenta del proceso administrador de memoria, que se ejecuta en el espacio de usuario y se comunica con el kernel mediante el mecanismo de mensajes estándar. La posición del administrador de memoria en el nivel de servidores se muestra en la Fig. 2-26.

Sacar el administrador de memoria del kernel es un ejemplo de la separación de **política** y **mecanismo**. Las decisiones acerca de cuál proceso se colocará en qué lugar de la memoria (política) son tomadas por el administrador de memoria. El establecimiento de mapas de memoria reales para los procesos (mecanismo) es efectuado por la tarea del sistema, dentro del kernel. Esta división hace que sea relativamente fácil modificar la política de administración de memoria (algoritmos, etc.), sin tener que alterar las capas inferiores del sistema operativo.

La mayor parte del código del administrador de memoria se dedica a manejar las llamadas al sistema de MINIX que implican administración de memoria, principalmente FORK y EXEC, no sólo a manipular listas de procesos y agujeros. En la siguiente sección examinaremos la organización de la memoria, y en secciones subsecuentes veremos a grandes rasgos cómo las llamadas al sistema e implican administración de memoria son procesadas por el administrador de memoria.

4.7.1 Organización de la memoria

Los procesos MINIX sencillos usan espacios I y D combinados, en los que todas las partes de proceso (texto, datos y pila) comparten un bloque de memoria que se asigna y libera como una unidad. También es posible compilar los procesos de modo que usen espacios I y D independientes. Por claridad, se explicará primero la asignación de memoria para el modelo más sencillo. ¡O! procesos que usan espacios I y D independientes pueden aprovechar la memoria con mayor eficiencia, pero las cosas se complican. Analizaremos las complicaciones después de bosquejar caso más sencillo.

Se asigna memoria en MINIX en dos ocasiones. Primero, cuando se bifurca un proceso, se asigna la cantidad de memoria que el hijo necesita. Segundo, cuando un proceso cambia su imagen de memoria mediante la llamada al sistema EXEC, la imagen antigua se devuelve a la lista libre como un agujero, y se asigna memoria para la nueva imagen, la cual puede estar en una parte de memoria distinta de la memoria liberada. La posición dependerá de dónde se encuentre un agujero apropiado. También se libera memoria cuando un proceso termina, ya sea porque salió (porque fue cancelado por una señal.

La Fig. 4-32 muestra ambas formas de asignar memoria. En la Fig. 4-32(a) vemos dos procesos, A y B, en la memoria. Si A bifurca, obtenemos la situación de la Fig. 4-32(b). El hijo a una copia exacta de A. Si el hijo ahora ejecuta el archivo C, la memoria se verá como en la Fig 4-32(c). La imagen del hijo es sustituida por C.

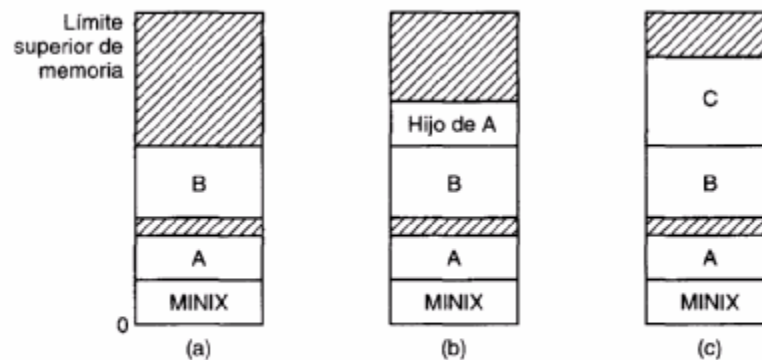


Figura 4-32. Reparto de memoria. (a) Originalmente. (b) Después de un FORK. (c) Después de que el hijo realiza un EXEC. Las regiones sombreadas indican memoria desocupada. El proceso es un proceso de I&D común.

Cabe señalar que la memoria que ocupaba el hijo es liberada antes de que se asigne la nueva memoria para C, así que C puede usar la memoria del hijo. De esta forma, una serie de pares FORB y EXEC (como cuando el shell establece un conducto o tubería) tiene como resultado que todos los procesos estén adyacentes, sin agujeros entre ellos, como habría sucedido si la nueva memoria se asignara antes de liberarse la memoria anterior.

Cuando se asigna memoria, ya sea por la llamada al sistema FORK o por EXEC, una parte de ella es ocupada por el nuevo proceso. En el primer caso, la cantidad que se toma es idéntica a la que

tiene el proceso padre. En el segundo caso, el administrador de memoria toma la cantidad especificada en la cabecera del archivo que se ejecuta. Una vez efectuada esta asignación, jamás se le asigna al proceso más memoria total.

Lo que hemos dicho hasta ahora aplica a programas que han sido compilados con espacios I y D combinados. Los programas con espacio I y D separado aprovechan un modo ampliado de administración de memoria llamado **texto compartido**. Cuando un proceso de este tipo ejecuta FORK, sólo se asigna la cantidad de memoria requerida para una copia de los datos y la pila del nuevo proceso. Tanto el padre como el hijo comparten el código ejecutable que el padre ya está usando. Cuando semejante proceso realiza un EXEC, se examina la tabla de procesos para determinar si otro proceso ya está usando el código ejecutable requerido. Si alguno lo está haciendo, sólo se asigna memoria para los datos y pila nuevos, y se comparte el texto que ya está en la memoria. La compartición de texto complica la terminación de un proceso. Cuando un proceso termina siempre libera la memoria ocupada por sus datos y su pila, pero sólo libera la memoria ocupada por su segmento de texto después de que una búsqueda en la tabla de procesos revela que ningún otro proceso vigente está compartiendo esa memoria. Así, podría asignarse más memoria a un proceso cuando inicia que la que se libera cuando termina, si carga su propio texto al iniciar pero dicho texto está siendo compartido por uno o más procesos aparte cuando el primer proceso termina.

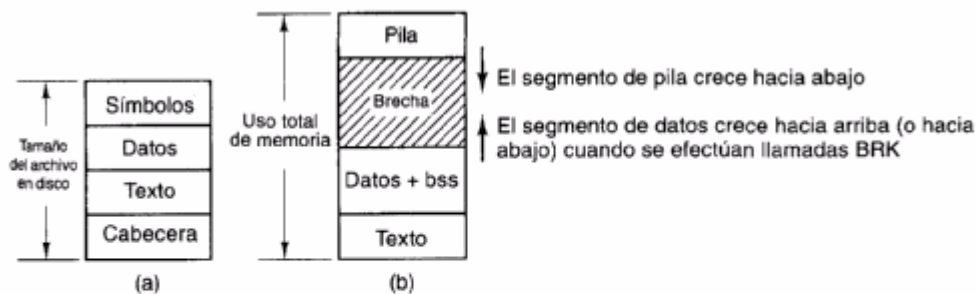


Figura 4-33. (a) Un programa tal como está almacenado en un archivo de disco. (b) Organización de memoria interna para un solo proceso. En ambas partes de la figura la dirección más baja del disco o de la memoria está abajo y la dirección más alta está arriba.

La Fig. 4-33 muestra cómo se almacena un programa en forma de archivo en disco y cómo éste se transfiere a la organización de memoria interna de un proceso MINIX. La cabecera del archivo contiene información acerca de los tamaños de las diferentes partes de la imagen, así como el tamaño total. En la cabecera de un programa con espacio I y D común, un campo especifica el tamaño total de las partes de texto y de datos; estas partes se copian directamente en la imagen de memoria. La parte de datos de la imagen se amplía en la cantidad especificada en el campo los de la cabecera. Esta área se despeja de modo que sólo contenga ceros y se utiliza para datos estáticos no inicializados. La cantidad total de memoria por asignar se especifica con el

campo total de la cabecera. Si, por ejemplo, un programa tiene 4K de texto, 2K de datos más bsf y 1K de pila, y la cabecera indica asignar 40K en total, el espacio de memoria desocupada entre el segmento de datos y el segmento de pila será de 33K. Un archivo de programa en disco también puede contener una tabla de símbolos. Ésta sirve para fines de depuración y no se copia en la memoria.

Si el programador sabe que la memoria total requerida para el crecimiento combinado de los segmentos de datos y de pila para el archivo a.out es de como máximo 10K, puede emitir el comando

```
chmem = 10240 a.out
```

que modifica el campo de cabecera de modo que cuando se emita EXEC el administrador de memoria asigne un espacio de 10240 bytes más grande que la suma de los segmentos de texto y de datos iniciales. En el ejemplo anterior, se asignaría un total de 16K en todos los EXEC subsecuentes del archivo. De esta cantidad, el 1K superior se usará para la pila, y habrá 9K en el espacio, donde podrá utilizarse para el crecimiento de la pila, del área de datos o de ambas cosas.

En el caso de un programa con espacio I y D separado (lo que se indica con un bit en la cabecera que el enlazador enciende), el campo total de la cabecera aplica sólo al espacio combinado de datos y pila. Un programa con 4K de texto, 2K de datos, 1K de pila y un tamaño total de 64K recibirá 68K (4K de espacio para instrucciones y 64K de espacio para datos), dejando 61K que el segmento de datos y la pila pueden consumir durante la ejecución. La frontera del segmento de datos sólo puede desplazarse con la llamada al sistema BRK. Lo único que hace BRK es determinar si el nuevo segmento de datos choca o no contra el apuntador a la pila actual y, si no choca, anota el cambio en ciertas tablas internas. Esto se efectúa totalmente dentro de la memoria originalmente asignada al proceso; el sistema operativo no asigna memoria adicional. Si el nuevo segmento de datos choca contra la pila, la llamada fracasa.

Se escogió tal estrategia con objeto de poder ejecutar MINIX en una IBM PC con un procesador 8088, que no vigila el desbordamiento de la pila en hardware. Un programa de usuario puede meter cuantas palabras quiera en la pila sin que el sistema operativo se dé cuenta de ello. En computadoras con hardware de administración de memoria más avanzado, se asigna inicialmente a la pila cierta cantidad de memoria. Si la pila trata de crecer más allá de este espacio, ocurre una trampa al sistema operativo y el sistema asigna otra porción de memoria a la pila, si es posible. Esta trampa no existe en el 8088, lo que hace peligroso tener a la pila adyacente a otra cosa que no sea un trozo grande de memoria desocupada, pues la pila puede crecer rápidamente y sin previo aviso. MINIX se diseñó de modo que cuando se implemente en una computadora con mejor administración de memoria no sea difícil modificar el administrador de memoria.

Ésta es una buena ocasión para mencionar un posible problema de semántica. Cuando usamos la palabra "segmento" nos referimos a un área de memoria definida por el sistema operativo. Los procesadores Intel 80x86 tienen un conjunto de "registros de segmento" internos y (en los procesadores más avanzados) "tablas de descriptores de segmentos" que proporcionan apoyo de hardware para los "segmentos". El concepto de "segmento" de los diseñadores de hardware Intel es similar, pero no siempre idéntico, a los segmentos empleados y definidos por MINIX. Todas las referencias a segmentos en este texto deben interpretarse como referencias a áreas de memoria

delineadas por estructuras de datos de MINIX. Nos referiremos explícitamente a los registros de segmento o descriptores de segmento cuando hablemos acerca del hardware.

Podemos generalizar esta advertencia. Los diseñadores de hardware a menudo tratan de proporcionar apoyo para los sistemas operativos que piensan que se usarán en sus máquinas, y la terminología empleada para describir registros y otros aspectos de la arquitectura de un procesador suele reflejar la idea que se tiene de cómo se utilizarán las capacidades. Tales capacidades con frecuencia resultan útiles para el implementador de un sistema operativo, pero no siempre se utilizan de la forma prevista por el diseñador de hardware. Esto puede dar pie a malentendidos cuando la misma palabra tiene diferente significado si se usa para describir un aspecto de un sistema operativo o del hardware subyacente.

4.7.2 Manejo de mensajes

Al igual que los demás componentes de MINIX, el administrador de memoria se controla mediante mensajes. Una vez que el sistema se ha inicializado, el administrador de memoria ingresa en su ciclo principal, que consiste en esperar un mensaje, atender la solicitud contenida en el mensaje y enviar una respuesta. En la Fig. 4-34 se muestra la lista de tipos de mensaje permitidos, sus parámetros de entrada y el valor que se devuelve en el mensaje de respuesta.

Es evidente que FORK, EXIT, WAIT, WAITPID, BRK y EXEC están muy relacionadas con la asignación y liberación de memoria. Las llamadas KILL, ALARM y PAUSE están relacionadas con las señales, lo mismo que SIGACTION, SIGSUSPEND, SIGPENDING, SIGMASK y SIGRETURN. Éstas también pueden afectar lo que está en la memoria porque cuando una señal mata un proceso la memoria que éste ocupaba se libera. REBOOT tiene efectos en todo el sistema operativo, pero su primera tarea es enviar señales para terminar todos los procesos de forma controlada, así que el administrador de memoria es un buen lugar para tenerlo. Las siete llamadas GET/SET nada tienen que ver con la administración de memoria, y tampoco con el sistema de archivos, pero tenían que estar ya sea en el sistema de archivos o en el administrador de memoria, pues todas las llamadas al sistema son manejadas por uno o por el otro. Dichas llamadas se colocaron aquí porque el sistema de archivos ya era muy grande de por sí. PTRACE, que se utiliza en depuración, está aquí por la misma razón.

El último mensaje, KSIG, no es una llamada al sistema; es el tipo de mensaje empleado por el kernel para informar al administrador de memoria de una señal que se origina en el kernel, como SIGINT, SIGQUIT o SIGALRM.

Aunque hay una rutina de biblioteca sbrk, no existe una llamada al sistema SBRK. La rutina de biblioteca calcula la cantidad de memoria requerida sumando al tamaño actual el incremento o decremento especificado como parámetro, y luego efectúa una llamada BRK para fijar el tamaño. De forma similar, no existen llamadas al sistema individuales para geteuid y getegid. Las llamadas GETUID y GETGID devuelven los identificadores tanto efectivo como real. De forma similar, GETPID devuelve el pid tanto del proceso invocador como de su padre.

Una estructura de datos clave empleada para el procesamiento de mensajes es la tabla `call_vec` declarada en `table.c` (línea 16515), la cual contiene apuntadores a los procedimientos que manejan los diversos tipos de mensajes. Cuando llega un mensaje al administrador de memoria, el ciclo

Tipo de mensaje	Parámetros de entrada	Valor de respuesta
FORK	(ninguno)	Pid del hijo (al hijo: 0)
EXIT	Situación de salida	(No hay respuesta si tiene éxito)
WAIT	(ninguno)	Estado
WAITPID	(ninguno)	Estado
BRK	Nuevo tamaño	Nuevo tamaño
EXEC	Apuntador a pila inicial	(No hay respuesta si tiene éxito)
KILL	Identificador de proceso y señal	Estado
ALARM	Número de segundos a esperar	Tiempo residual
PAUSE	(ninguno)	(No hay respuesta si tiene éxito)
SIGACTION	Núm. señal, acción, acción vieja	Estado
SIGSUSPEND	Máscara de señal	(No hay respuesta si tiene éxito)
SIGPENDING	(ninguno)	Estado
SIGMASK	Cómo, conjunto, conjunto viejo	Estado
SIGRETURN	Contexto	Estado
GETUID	(ninguno)	Uid, uid efectivo
GETGID	(ninguno)	Gid, gid efectivo
GETPID	(ninguno)	Pid, pid del padre
SETUID	Uid nuevo	Estado
SETGID	Gid nuevo	Estado
SETSID	Sid nuevo	Grupo de procesos
GETPGRP	Gid nuevo	Grupo de procesos
Ptrace	Solicitud, pid, dirección, datos	Estado
REBOOT	Cómo (parar, rearrancar o pánico)	(No hay respuesta si tiene éxito)
KSIG	Ranura de proceso y señales	(No hay respuesta)

Figura 4-34. Los tipos de mensajes, parámetros de entrada y valores de respuesta empleados para comunicarse con el administrador de memoria.

principal extrae el tipo de mensaje y lo coloca en la variable global `mm_call`. Este valor se `u||` después como índice para entrar en `callvec` y encontrar el apuntador al procedimiento que `nuffel` mensaje recién llegado. A continuación se invoca ese procedimiento para ejecutar la llamada al sistema. El valor que se devuelve se envía de vuelta al invocador en el mensaje de respuesta) informar del éxito o el fracaso de la llamada. Este mecanismo es similar al de la Fig. 1-16, sólo| en el espacio de usuario en lugar del `kemel`.

∴

4.7.3 Estructuras de datos y algoritmos del administrador de memoria

El administrador de memoria tiene dos estructuras de datos clave: la tabla de procesos y la tabla de agujeros. A continuación las veremos por turno.

En la Fig. 2-4 vimos que algunos campos de la tabla de procesos son necesarios para la administración de procesos, otros para la administración de memoria y otros más para el sistema de archivos. En MINIX, cada una de estas tres partes del sistema operativo tiene su propia tabla de procesos, misma que contiene sólo los campos que esa parte necesita. Las entradas corresponden exactamente, a fin de simplificar las cosas. Así, la ranura *k* de la tabla del administrador de memoria se refiere al mismo proceso que la ranura *k* de la tabla del sistema de archivos. Cuando se crea o se destruye un proceso, las tres partes actualizan sus tablas de modo que reflejen la nueva situación, a fin de mantenerlas sincronizadas.

La tabla de procesos del administrador de memoria se llama *mproc*; su definición está en `usr/src/mm/mproc.h`. Esta tabla contiene todos los campos relacionados con la asignación de memoria e un proceso, así como ciertos elementos adicionales. El campo más importante es el arreglo *p_seg*, que tiene tres entradas, para los segmentos de texto, datos y pila, respectivamente. Cada entrada es una estructura que contiene la dirección virtual, la dirección física y la longitud del segmento, todas medidas en clics, no en bytes. El tamaño de un clic depende de la implementación; para el MINIX estándar es 256 bytes. Todos los segmentos deben comenzar en una frontera de clic y ocupar un número entero de clics.

El método empleado para registrar la asignación de memoria se muestra en la Fig. 4-35. En esta figura tenemos un proceso con 3K de texto, 4K de datos, un espacio de 1K y luego una pila de 2K, para una asignación total de memoria de 10K. En la Fig. 4-35(b) vemos en qué consisten los campos virtual, físico y de longitud para cada uno de los tres segmentos, suponiendo que el proceso no tiene espacio *I* y *D* separado. En este modelo, el segmento de texto siempre está vacío, y el segmento de datos contiene tanto texto como datos. Cuando un proceso hace referencia a la dirección virtual *O*, ya sea para saltar a ella o para leerla (es decir, como espacio de instrucción o como espacio de datos), se usa la dirección física 0x32000 (en decimal, 200K). Esta dirección está en el clic 0x320.

Cabe señalar que la dirección virtual en la que comienza la pila depende inicialmente de la cantidad total de memoria asignada al proceso. Si se usara el comando *chmem* para modificar a cabecera de archivo a fin de contar con un área de asignación dinámica más grande (un espacio mayor entre los segmentos de datos y de pila), la siguiente vez que se ejecutara el archivo la pila comenzaría en una dirección virtual más alta. Si la pila se alarga en un clic, la entrada correspondiente a la pila debería cambiar de la tripleta (0x20, 0x340, 0x8) a la tripleta (0x1F, 0x33F, 0x9).

El hardware del 8088 no tiene una trampa de límite de pila, y MINIX define la pila de modo tal que no dispare la trampa en los procesadores de 32 bits hasta que la pila ya haya sobrescrito el segmento de datos. Por tanto, este cambio no se efectuará sino hasta la siguiente llamada al sistema *BRK*, y entonces el sistema operativo leerá explícitamente *SP* y volverá a calcular las direcciones de los segmentos. En una máquina con trampa de pila, la entrada del segmento de pila se tenía que actualizar tan pronto como la longitud de la pila excediera la de su segmento. MINIX no lo hace en los procesadores Intel de 32 bits, por razones que veremos a continuación.

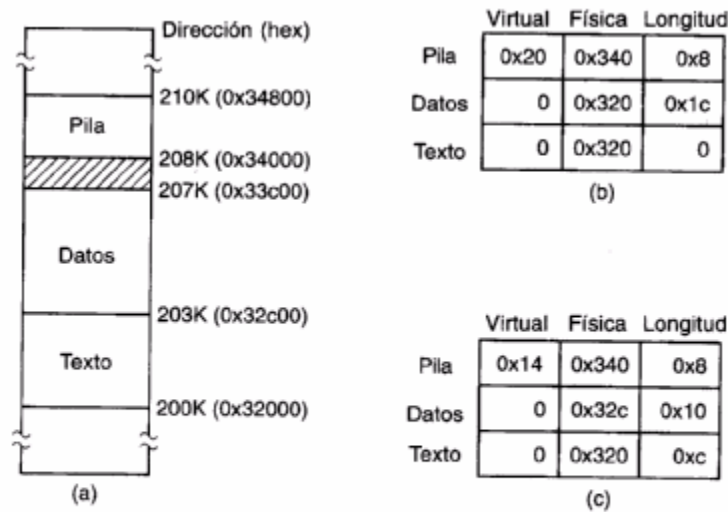


Figura 4-35. (a) Un proceso en la memoria. (b) Su representación de memoria con espacio y D no separado. (c) Su representación con espacio I y D separado.

Ya mencionamos que las labores de los diseñadores de hardware no siempre producen lo que el diseñador de software necesita. Incluso en modo protegido en un Pentium, MINIX no atrapa; cuando la pila se desborda de su segmento al crecer. Aunque en modo protegido el hardware Intel detecta los intentos por acceder a memoria fuera de un segmento (definido por un descriptor de segmento como el de la Fig. 4-28), en MINIX el descriptor de segmento de datos y el descriptor de segmento de pila siempre son idénticos. Los segmentos de datos y de pila definidos por Minff utilizan cada uno una parte de este espacio, de modo que cualquiera de ellos, o ambos, se pueden expandir hacia la brecha que los separa. Sin embargo, sólo MINIX puede administrar esto. La CPU; no tiene forma de detectar errores relativos a la brecha, ya que en lo que al hardware concierne dicha brecha es una parte válida tanto del área de datos como del área de pila. Desde luego, el hardware puede detectar un error muy grande, como un intento por acceder a memoria fuera del área combinada de datos-hueco-pila. Esto protege a un proceso de los errores de otro proceso pero no basta para proteger a un proceso de sí mismo.

Aquí se tomó una decisión de diseño. Reconocemos que podría haber argumentos válidos! favor de abandonar el segmento compartido definido por hardware que permite a MINIX reasignar dinámicamente el área de la brecha. La alternativa, usar el hardware para definir segmentos de datos y de pila que no se traslapen, ofrecería un poco más de protección contra ciertos errores pero haría que MINIX estuviera más hambriento de memoria. El código fuente está disponible para cualquier persona que desee evaluar el otro enfoque.

La Fig. 4-35(c) muestra las entradas de segmento para la organización de memoria de la Fig. 4-35(a) con espacio I y D separado. Aquí los segmentos tanto de texto como de datos tienen longitud mayor que cero. El arreglo `mp_seg` que se muestra en la Fig. 4-35(b) o (c) se utilizó primordialmente para transformar direcciones virtuales en direcciones de memoria física. ~

una dirección virtual y el espacio al que pertenece, resulta sencillo determinar si la dirección virtual es válida o no (es decir, si queda dentro de un segmento) y, si es válida, a qué dirección física corresponde. El procedimiento de kernel umap realiza esta transformación para las tareas de E/S y para el copiado en y desde el espacio de usuario, por ejemplo.

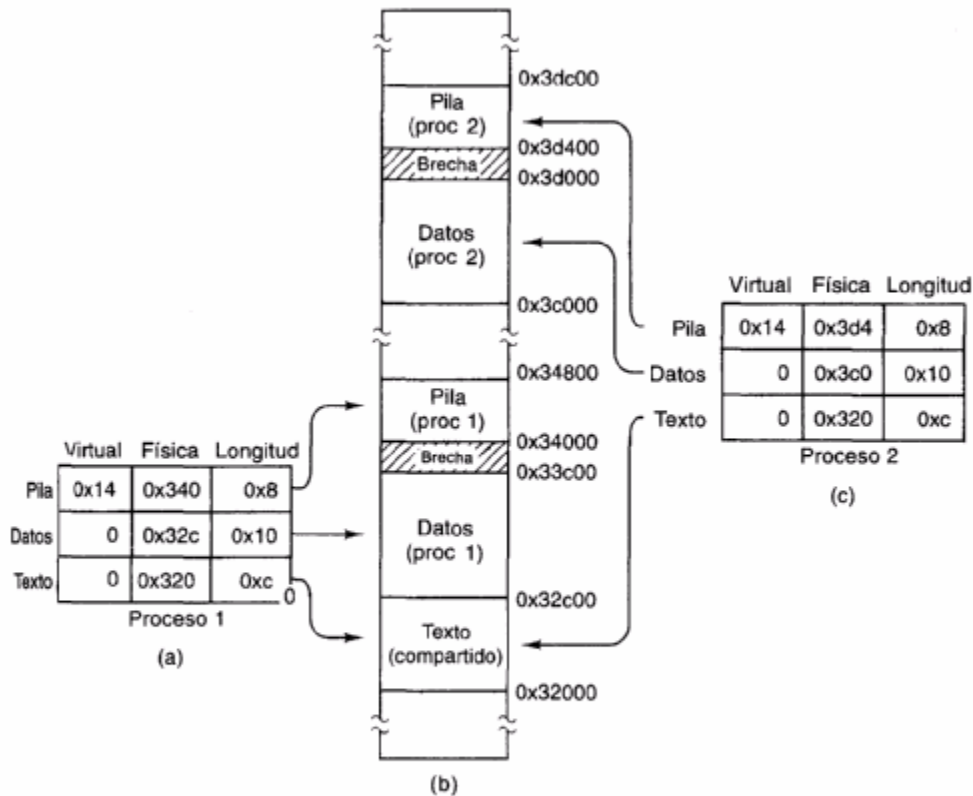


Figura 4-36. (a) El mapa de memoria de un proceso con espacio I y D separado, como en la figura anterior. (b) La organización en la memoria después de que se inicia un segundo proceso, ejecutando la misma imagen de programa con texto compartido. (c) El mapa de memoria del segundo proceso.

El contenido de las áreas de datos y de pila pertenecientes a un proceso puede cambiar durante la ejecución de éste, pero el texto no cambia. Es común que varios procesos estén ejecutando copias del mismo programa; por ejemplo, varios usuarios podrían estar ejecutando el mismo shell. La eficiencia de memoria mejora si se emplea texto compartido. Cuando EXEC está a punto de cargar un proceso, abre el archivo que contiene la imagen en disco del programa que se va a cargar, y lee la cabecera del archivo. Si el proceso emplea espacio I y D separado, se examinan los campos mp_dev, mp_ino y mp_ctime de cada ranura de mproc. Estos contienen los números de dispositivo y de nodo-i y los tiempos de cambio de situación de las imágenes que están siendo ejecutadas por otros procesos. Si se determina que un proceso que ya se cargó está ejecutando el

mismo programa que está a punto de cargarse, no habrá necesidad de asignar memoria para otra copia del texto. En vez de ello, se inicializa la porción `mp_seg[T]` del mapa de memoria del nuevo proceso de modo que apunte al mismo lugar en el que ya está cargado el segmento de texto, y sólo se establecen las porciones de datos y de pila en una nueva asignación de memoria. Esto se muestra en la Fig. 4-36. Si el programa usa espacio I y D combinado o no se encuentra un proceso que esté ejecutando el mismo programa, la memoria se reparte como se muestra en la Fig. 4-35 y el texto y los datos para el nuevo proceso se copian en ella desde el disco.

Además de la información sobre segmentos, `mproc` también contiene el identificador de proceso (`pid`) del proceso mismo y de su padre, los `uids` y `gids` (tanto reales como efectivos), información referente a las señales y la situación de salida, si el proceso ya terminó pero su padre todavía no efectúa un `WAIT` por él.

La otra tabla importante del administrador de memoria es la de agujeros, `hole`, definida en `alloc.c`, que lista todos los agujeros de memoria en orden de dirección creciente. Los espacios entre los segmentos de datos y de pila no se consideran agujeros; ya han sido asignados a procesos, así que no están contenidos en la lista de agujeros libres. Cada entrada de la lista de agujeros tiene tres campos: la dirección base del agujero en clics, la longitud del agujero en clics y un apuntador a la siguiente entrada de la lista. La lista sólo tiene enlaces sencillos, así que es fácil encontrar el siguiente agujero a partir de cualquier agujero dado, pero para encontrar el agujero anterior hay que examinar toda la lista desde el principio hasta llegar al agujero en cuestión.

La razón por la que se registra todo acerca de los segmentos y agujeros en clics en vez de bytes es sencilla: resulta mucho más eficiente. En modo de 16 bits, se usan enteros de 16 bits para registrar direcciones de memoria, así que con clics de 256 bytes se puede manejar hasta 16 MB de memoria. En modo de 32 bits, los campos de dirección pueden hacer referencia a hasta 240 bytes, es decir, 1024 gigabytes.

Las principales operaciones de la lista de agujeros son asignar una porción de memoria de cierto tamaño y devolver una asignación existente. Para asignar memoria, se realiza una búsqueda en la lista de agujeros, comenzando por el agujero con la dirección más baja, hasta encontrar un agujero con el tamaño suficiente (primer ajuste). A continuación se asigna el segmento reduciendo el agujero en la cantidad necesaria para el segmento o, en los raros casos en que el ajuste es exacto, eliminando el agujero de la lista. Este esquema es rápido y sencillo pero adolece tanto de una pequeña cantidad de fragmentación interna (se pueden desperdiciar hasta 255 bytes del clisé final, ya que siempre se toma un número entero de clics) como de fragmentación externa.

Cuando un proceso termina y se realiza el aseo, su memoria de datos y de pila se coloca vuelta en la lista libre. Si el proceso usa I y D común, con esto se libera toda su memoria, ya que nunca tienen una asignación de memoria aparte para texto. Si el programa también se devolvió la memoria asignada a texto. Puesto que en el texto compartido las regiones de texto y de datos no necesariamente son contiguas, se devuelven dos regiones de memoria. Para cada región devuelta, si cualquiera de los vecinos, o ambos, son agujeros, se fusionan, de modo que nunca hay agujeros adyacentes. número, ubicación y tamaño de los agujeros varían continuamente durante la operación del sistema. Si todos los procesos de usuario han terminado, toda la memoria disponible estará

lista para ser asignada. Esto no implica necesariamente que sólo habrá un agujero, pues la memoria física puede estar interrumpida por regiones que el sistema operativo no puede utilizar, como en los sistemas compatibles con IBM en los que la memoria sólo de lectura (ROM) y la memoria reservada para transferencias de E/S separan la memoria utilizable por debajo de la dirección 640K de la memoria que está por arriba de 1K.

4.7.4 Las llamadas al sistema FORK, EXIT y WAIT

Cuando se crean o destruyen procesos, hay que asignar o liberar memoria. Además, es preciso actualizar la tabla de procesos, incluidas las partes que mantienen el kernel y el sistema de archivos. El administrador de memoria coordina toda esta actividad. La creación de procesos se efectúa con FORK, y consta de la serie de pasos que se muestran en la Fig. 4-37.

1. Verifica permisos —¿es ejecutable el archivo?
2. Lee la cabecera para obtener los tamaños de segmento y total.
3. Obtiene los argumentos y el entorno del invocador.
4. Asigna nueva memoria y libera la memoria vieja que no se necesite.
5. Copia la pila en la nueva imagen de memoria.
6. Copia el segmento de datos (y tal vez el de texto) en la nueva imagen de memoria.
7. Verifica y maneja los bits setuid y setgid.
8. Corrige la entrada de la tabla de procesos.
9. Informa al kernel que el proceso es ejecutable.

Figura 4-38. Los pasos requeridos para llevar a cabo la llamada al sistema EXEC.

Es difícil y poco recomendable detener una llamada FORK antes de terminar, así que el administrador de memoria mantiene en todo momento una cuenta del número de procesos existentes a fin de poder determinar fácilmente si hay una ranura disponible en la tabla de procesos. Si la tabla no está llena, se intenta asignar memoria para el hijo. Si el programa tiene espacio I y D separado, sólo se solicita suficiente memoria para las nuevas asignaciones de datos y pila. Si este paso también tiene éxito, está garantizado que el FORK funcionará. A continuación se llena la memoria recién asignada, se localiza una ranura de proceso y se llena, se escoge un pid y se informa a las demás partes del sistema que se creó un proceso nuevo.

Un proceso termina por completo una vez que han ocurrido dos sucesos: (1) el proceso en sí (lió (o fue cancelado por una señal) y (2) su padre ejecutó una llamada al sistema WAIT para averiguar qué sucedió. Un proceso que ha salido o que ha sido cancelado, pero cuyo padre todavía no ha realizado un WAIT por él, queda en una especie de animación suspendida, a veces EDonúnada estado zombi. El proceso no puede ser planificado y tiene apagado su temporizador ^alarma (si estaba encendido), pero no se retira de la tabla de procesos. Su memoria queda

liberada. El estado zombi es temporal y casi nunca dura mucho. Cuando el padre finalmente ejecuta el WAIT, la ranura de la tabla de procesos se libera y se informa de ello al sistema de archivos y al kernel.

Surge un problema si el padre del proceso que está saliendo ya está muerto. Si no se realiza alguna acción especial, el proceso saliente seguiría como zombi indefinidamente. En vez de ellos las tablas se modifican para convertirlo en hijo del proceso init. Cuando el sistema arranca, lee el archivo /etc/ttytab para obtener una lista de todas las terminales, y después bifurca un proceso de inicio de sesión para manejar cada uno. A continuación, init se bloquea, esperando que los procesos terminen. De esta forma, los zombis huérfanos se eliminan rápidamente.

4.7.5 La llamada al sistema EXEC

Cuando se teclea un comando en una terminal, el shell bifurca un nuevo proceso, que entonces ejecuta el comando solicitado. Podría haber sido posible tener una sola llamada al sistema que efectuara tanto FORK como EXEC a un tiempo, pero se prefirió tener dos llamadas por una razón excelente: facilitar la implementación de la redirección de E/S. Cuando el shell bifurca, si se redirige la entrada estándar, el hijo cierra la entrada estándar y luego abre la nueva entrada estándar antes de ejecutar el comando. De este modo, el proceso recién iniciado hereda la entrada estándar redirigida. La salida estándar se maneja de la misma forma.

EXEC es la llamada al sistema más compleja de MINIX, pues debe sustituir la imagen de memoria actual por una nueva, lo que incluye el establecimiento de una nueva pila. EXEC realiza este trabajo en una serie de pasos, los cuales se listan en la Fig. 4-38.

1. Verifica permisos —¿es ejecutable el archivo?
2. Lee la cabecera para obtener los tamaños de segmento y total.
3. Obtiene los argumentos y el entorno del invocador.
4. Asigna nueva memoria y libera la memoria vieja que no se necesite.
5. Copia la pila en la nueva imagen de memoria.
6. Copia el segmento de datos (y tal vez el de texto) en la nueva imagen de memoria.
7. Verifica y maneja los bits setuid y setgid.
8. Corrige la entrada de la tabla de procesos.
9. Informa al kernel que el proceso es ejecutable.

Figura 4-38. Los pasos requeridos para llevar a cabo la llamada al sistema EXEC.

Cada paso consiste, a su vez, en pasos aún más pequeños, algunos de los cuales pueden fracasar. Por ejemplo, podría ser que no hubiera suficiente memoria disponible. El orden en que se efectúan las pruebas se escogió con cuidado a fin de asegurarse de que la nueva imagen de memoria no se liberará antes de tener la certeza de que el EXEC tendrá éxito, con objeto de evitar una embarazosa situación de no poder establecer una nueva imagen de memoria y tampoco tenerla.

antigua para regresar a ella. Normalmente, EXEC no regresa, pero si falla el proceso invocador debe obtener de nuevo el control, con una indicación de error.

Hay unos cuantos pasos de la Fig. 4-38 que merecen un comentario más amplio. Primero está la cuestión de si hay o no espacio suficiente. Después de determinar cuánta memoria se necesita, lo que requiere determinar si se puede compartir o no la memoria de texto de otro proceso, se examina la lista de agujeros para comprobar si hay suficiente memoria física antes de liberar la memoria anterior. Si la memoria vieja se liberara primero y no hubiera suficiente memoria, sería difícil recuperar la imagen anterior.

Sin embargo, esta prueba es demasiado estricta; a veces rechaza llamadas EXEC que, de hecho, podrían tener éxito. Supongamos, por ejemplo, que el proceso que efectúa la llamada EXEC ocupa 20K y que su texto no es compartido por ningún otro proceso. Supongamos, además, que hay un agujero de 30K disponible y que la nueva imagen requiere 50K. Al efectuar la prueba antes de liberar, descubriremos que sólo hay 30K disponibles y rechazaremos la llamada. Si hubiéramos liberado primero, podríamos haber tenido éxito, dependiendo de si el nuevo agujero de 20K está adyacente a, y por tanto ahora fusionado con, el agujero de 30K o no. Una implementación más avanzada podría manejar esta situación un poco mejor.

Otra mejora posible sería buscar dos agujeros, uno para el segmento de texto y otro para el segmento de datos, si el proceso que se llamó con EXEC tiene espacio I y D separado. No es necesario que los segmentos sean contiguos.

Un problema más sutil es si el archivo ejecutable cabe o no en el espacio de direcciones virtual. El problema es que la memoria se asigna no en bytes, sino en clics de 256 bytes. Cada clic debe pertenecer a un solo segmento, y no puede ser, por ejemplo, mitad datos y mitad pila, porque toda la administración de memoria se efectúa en clics.

Para ver cómo esta restricción puede causar problemas, observemos que el espacio de direcciones en los sistemas de 16 bits (8088 y 80286) está limitado a 64K, el cual puede dividirse en 256 clics. Supongamos que un programa con espacio I y D separado tiene 40 000 bytes de texto, 32 770 bytes de datos y 32 760 bytes de pila. El segmento de datos ocupa 129 clics, de los cuales el último sólo se utiliza parcialmente; de todos modos, el clic completo forma parte del segmento de datos. El segmento de pila tiene 128 clics. Juntos exceden 256 clics, y por tanto no pueden coexistir, a pesar de que el número de bytes requeridos sí cabe (apenas) en el espacio de direcciones virtual. En teoría, este problema existe en todas las máquinas cuyo tamaño de clic es mayor que un byte, pero en la práctica casi nunca ocurre en procesadores clase Pentium, ya que éstos permiten segmentos grandes (de 4 GB).

Otra cuestión importante es la configuración de la pila inicial. La llamada de biblioteca que normalmente se usa para invocar EXEC con argumentos y un entorno es

```
execve(name, argv, envp);
```

donde name es un apuntador al nombre del archivo por ejecutar, argv es un apuntador a un arreglo de apuntadores, cada uno de los cuales apunta a un argumento, y envp es un apuntador a un arreglo de apuntadores, cada uno de los cuales apunta a una cadena de entorno.

Sería muy fácil implementar EXEC colocando simplemente los tres apuntadores en el mensaje al administrador de memoria y dejando que él obtenga el nombre de archivo y los dos arreglos por

sí solo. Entonces, el administrador de memoria tendría que obtener cada argumento y cada cadena uno por uno. Hacerlo de esta forma requeriría al menos un mensaje a la tarea del sistema por cada argumento o cadena, y probablemente más, ya que el administrador de memoria no tiene forma de saber anticipadamente qué tan grande es cada uno.

Para evitar el gasto extra de una gran cantidad de mensajes para leer todos estos componentes, se ha escogido una estrategia totalmente distinta. El procedimiento de biblioteca `execve` construye toda la pila inicial dentro de sí mismo y pasa su dirección base y tamaño al administrador de memoria. La construcción de la nueva pila dentro del espacio de usuario es muy eficiente, porque las referencias a los argumentos y cadenas sólo son referencias a memoria local, no a un espacio de direcciones diferente.

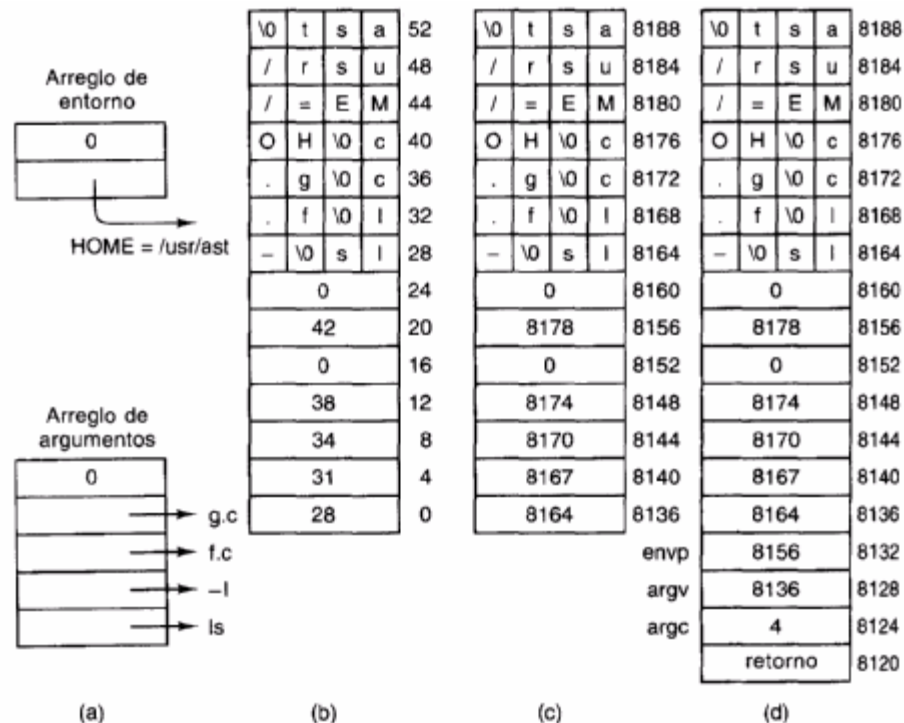


Figura 4-39. (a) Los arreglos que se pasan a `execve`. (b) La pila construida por `execve`. (c) La pila después de ser reubicada por el administrador de memoria. (d) La pila tal como `main` la ve al principio de la ejecución.

A fin de hacer más claro este mecanismo, consideremos un ejemplo. Cuando un usuario teclea `ls -l f.c g.c` para el shell, éste interpreta el comando y luego efectúa la llamada `execve("/bin/ls", argv, envp);` al procedimiento de biblioteca. El contenido de los dos arreglos de apuntadores se muestra en la Fig. 4-39(a). Ahora, el procedimiento `execve`, dentro del espacio de direcciones del shell, construye

la pila inicial, como se muestra en la Fig. 4-39(b). Esta pila se copia posteriormente sin modificación al administrador de memoria durante el procesamiento de la llamada EXEC.

Cuando la pila se copie finalmente al proceso de usuario, no se colocará en la dirección virtual 0. En vez de ello, se colocará al final de la asignación de memoria, determinado por el campo de tamaño de memoria total de la cabecera del archivo ejecutable. Por ejemplo, supongamos arbitrariamente que el tamaño total es de 8192 bytes, de modo que el último byte disponible para el programa está en la dirección 8191. Es obligación del administrador de memoria reubicar los apuntadores dentro de la pila de modo que, cuando se le deposite en la nueva dirección, la pila tenga el aspecto que se muestra en la Fig. 4-39(c).

Cuando se completa la llamada EXEC y el programa comienza a ejecutarse, la pila sí tendrá exactamente el aspecto de la Fig. 4-39(c), con el valor 8136 en el apuntador a la pila. Sin embargo, falta resolver otro problema. El programa principal del archivo ejecutado probablemente se declara con algo similar a esto:

```
main(argc, argv, envp);
```

En lo que al compilador de C concierne, main no es más que otra función. El compilador no sabe que main es especial, así que compila código para acceder a los tres parámetros bajo el supuesto de que éstos se pasarán empleando la convención de llamada estándar de C, con el último parámetro primero. Al ser un entero y dos apuntadores, se espera que los tres parámetros ocuparán las tres palabras que están justo antes de la dirección de retomo. Desde luego, la pila de la Fig. 4-39(c) no tiene para nada ese aspecto.

La solución es que los programas no comiencen con main. En vez de ello, siempre se vincula en la dirección de texto O una pequeña rutina en lenguaje ensamblador llamada crtso, el procedimiento de inicio de tiempo de ejecución de C, con objeto de que sea la primera en obtener el control. La misión de crtso es meter tres palabras más en la pila y luego invocar main usando la instrucción de llamada estándar. Esto produce la pila de la Fig. 4-39(d) en el momento en que main empieza a ejecutarse. Así, se engaña a main haciéndole creer que se le invocó de la manera usual (en realidad, no es un engaño; sí se le invoca de esa manera).

Si el programador se olvida de invocar exit al final de main, el control regresará a la rutina de inicio de tiempo de ejecución de C cuando main termine. Una vez más, el compilador sólo ve a main como un procedimiento ordinario y genera el código acostumbrado para regresar de él después de la última instrucción. Así, main regresa a su invocador, la rutina de inicio de tiempo de ejecución de C, que luego invoca ella misma exit. La mayor parte del código del crtso de 32 bits se muestra en la Fig. 4-40. Los comentarios deberán hacer obvio su funcionamiento. Lo único que se omitió es el código que carga los registros que se agregan a la pila y unas cuantas líneas que izan una bandera que indica si está presente un coprocesador de punto flotante o no.

4.7.6 La llamada al sistema BRK

Los procedimientos de biblioteca brk y sbrk sirven para ajustar el límite superior del segmento de datos. El primero toma un tamaño absoluto (en bytes) e invoca BRK. El segundo toma un incremento

```

push ecx      ! poner en la pila entorno
push edx      ! poner en la pila argv
push eax      ! poner en la pila argccali _main      ! main(argc, argv, envp)
push eax      ! poner en la pila estado de salida
call _exit
hit           ! forzar a una trampa si exit falla

```

Figura 4-40. La pane clave de la rutina de inicio de tiempo de ejecución de C.

positivo o negativo para el tamaño actual, calcula el nuevo tamaño de segmento de datos, y luego invoca BRK. No existe una llamada al sistema SBRK.

Una pregunta interesante es: "¿Cómo se mantiene sbrk al tanto del tamaño actual, para poder calcular el nuevo tamaño?" La respuesta es que una variable, brksize, siempre contiene el tamaño actual para que sbrk pueda encontrarlo. Esta variable se inicializa con un símbolo generado por el compilador que indica el tamaño inicial del texto más los datos (I y D no separados) o sólo los datos (I y D separados). El nombre y, de hecho, la existencia misma de tal símbolo depende del compilador y, por tanto, no lo veremos definido en ningún archivo de cabecera de los directorios de archivos fuente. El símbolo se define en la biblioteca, en el archivo hrksize.s. El lugar exacto donde se encuentra este archivo depende del sistema, pero siempre está en el mismo directorio que crtso.s.

Para el administrador de memoria es fácil ejecutar BRK. Lo único que hay que hacer es verificar que todo quepa aún en el espacio de direcciones, ajustar las tablas e informarle al kernel.

4.7.7 Manejo de señales

En el capítulo 1 describimos las señales como un mecanismo para comunicar información a un proceso que no necesariamente está esperando entradas. Existe un conjunto definido de señales, y cada señal tiene una acción por omisión: ya sea terminar el proceso al que va dirigida o hacer caso omiso de la señal. El procesamiento de señales sería fácil de entender e implementar si estas fueran las únicas alternativas. Sin embargo, los procesos pueden usar llamadas al sistema que alteran estas respuestas. Un proceso puede solicitar que se haga caso omiso de todas las señales (excepto la señal especial SIGKILL). Además, un proceso puede prepararse para **atrapar** una señal solicitando que se active un procedimiento manejador de señal interno al proceso, en lugar de la acción predeterminada para cualquier señal (con la excepción, otra vez, de SIGKILL). Así, desde tu punto de vista del programador, sólo hay dos momentos bien definidos en los que el sistema operativo maneja señales: una fase de preparación durante la cual un proceso puede modificar su respuesta a una señal futura y una fase de respuesta en la que se genera una señal y se realiza la acción correspondiente. La acción puede ser la ejecución de un manejador de señales personalizado;

En realidad, existe una tercera fase. Cuando un manejador escrito por el usuario termina, una llamada al sistema especial efectúa limpieza y restaura el funcionamiento normal del proceso al que se envió la señal. El programador no necesita estar enterado de esta tercera fase. Él escribe un manejador de señales igual que cualquier otra función. El sistema operativo se encarga de los detalles de invocar y terminar el manejador y administrar la pila.

En la fase de preparación hay varias llamadas al sistema que un proceso puede ejecutar en cualquier momento para modificar su respuesta a una señal. La más general de éstas es SIGACTION, que puede especificar que el proceso haga caso omiso de alguna señal, atrape alguna señal (sustituyendo la acción predeterminada por la ejecución de código de manejo de señales definido por el usuario dentro del proceso), o restaure la respuesta predeterminada a alguna señal. Otra llamada al sistema, SIGPROCmask, puede bloquear una señal, causando que se ponga en cola y que sólo se efectúe la acción correspondiente cuando el proceso desbloquee esa señal en particular en algún momento posterior, si es que llega a hacerlo. Estas llamadas pueden efectuarse en cualquier momento, incluso desde el interior del código que maneja una señal atrapada. En MINIX la fase de preparación del procesamiento de señales corre totalmente por cuenta del administrador de memoria, ya que todas las estructuras de datos necesarias están en la parte de la tabla de procesos que le corresponde. Para cada proceso hay cierto número de variables sigset_t, en las que cada señal posible está representada por un bit. Una de esas variables define un conjunto de señales a las que no se debe hacer caso, otra define un conjunto que debe atraparse, etc. Para cada proceso hay también un arreglo de estructuras sigaction, una para cada señal. Cada elemento de la estructura sigaction contiene una variable en la que se coloca la dirección de un manejador personalizado para esa señal y una variable sigset_t adicional para indicar las señales que se bloquearán mientras se está ejecutando ese manejador. El campo empleado para la dirección del manejador puede contener en vez de ella valores especiales que indiquen que la señal debe ignorarse o debe manejarse de la forma predefinida para esa señal.

Cuando se genera una señal, pueden participar varias partes del sistema MINIX. La respuesta se inicia en el administrador de memoria, que determina cuál proceso debe recibir la señal utilizando las estructuras de datos que acabamos de mencionar. Si la señal debe atraparse, debe entregarse al proceso objetivo. Esto requiere guardar información acerca del estado del proceso, a fin de poder reanudar la ejecución normal. La información se almacena en la pila del proceso que recibió la señal, y debe verificarse que haya suficiente espacio en ella. El administrador de memoria se encarga de esta verificación, ya que le corresponde, y luego invoca la tarea del sistema en el kernel para que coloque la información en la pila. La tarea del sistema también manipula el contador de programa del proceso, de modo que el proceso pueda ejecutar el código del manejador. Cuando el manejador termina, se efectúa una llamada al sistema SIGRETURN. Por medio de esta llamada, tanto el administrador de memoria como el kernel participan en el restablecimiento del contexto de señales y los registros del proceso para que pueda reanudar su ejecución normal. Si la señal no se atrapa, se emprende la acción predeterminada, que puede implicar invocar el sistema de archivos para producir un **vaciado de núcleo** (escribir la imagen del proceso en un archivo que se puede examinar con un depurador), así como terminar el proceso, lo que requiere la participación del administrador de memoria, el sistema de archivos y el kernel. Por último, el administrador de memoria podría ordenar una o más repeticiones de estas acciones, ya que puede ser necesario entregar una misma señal a un grupo de procesos.

Las señales que MINIX reconoce están definidas en /usr/include/signal.h, un archivo requerido por el estándar POSIX. Dichas señales se listan en la Fig. 4-41. Todas las señales requeridas por POSIX están definidas en MINIX, pero no todas ellas se reconocen actualmente. Por ejemplo, POSIX requiere varias señales relacionadas con el control de trabajos, la capacidad para poner en segundo

plano un programa en ejecución y traerlo otra vez a primer plano. MINIX no apoya el control de trabajos, pero los programas que podrían generar tales señales pueden trasladarse a MINIX. Se hará caso omiso de estas señales si se generan. MINIX también define algunas señales que no son de POSIX y algunos sinónimos de nombres de POSIX por compatibilidad con código fuente viejo.

Señal	Descripción	Generada por
SIGHUP	Cuelga	Llamada al sistema KILL
SIGINT	Interrumpe	Kernel
SIGQUIT	Abandona	Kernel
SIGILL	Instrucción no permitida	Kernel (*)
SIGTRAP	Trampa de rastreo	Kernel (M)
SIGABRT	Terminación anormal	Kernel
SIGFPE	Excepción de punto flotante	Kernel (*)
SIGKILL	Kill (no puede atraparse ni ignorarse)	Llamada al sistema KILL
SIGUSR1	Señal definida por el usuario # 1	No se reconoce
SIGSEGV	Violación de segmentación	Kernel (*)
SIGUSR2	Señal definida por el usuario # 2	No se reconoce
SIGPIPE	Escribe en un conducto sin nadie que lea	Kernel
SIGALRM	Reloj de alarma, tiempo de espera	Kernel
SIGTERM	Señal de terminación de software desde kill	Llamada al sistema KILL
SIGCHLD	Proceso hijo terminado o detenido	No se reconoce
SIGCONT	Continúa si está detenido	No se reconoce
SIGSTOP	Señal de detenerse	No se reconoce
SIGTSTP	Señal de detenerse interactiva	No se reconoce
SIGTTIN	Proceso de segundo plano desea leer	No se reconoce
SIGTTOU	Proceso de segundo plano desea escribir	No se reconoce

Figura 4-41. Señales definidas por POSIX y MINIX. Las señales marcadas con (*) dependen del apoyo de hardware. Las señales marcadas con (M) no están definidas por POSIX, pero sí por MINIX para fines de compatibilidad con programas viejos. Varios nombres y sinónimos obsoletos no aparecen en esta lista.

Hay dos formas de generar señales: con la llamada al sistema KILL y desde el kernel. Las señales generadas por el kernel de MINIX siempre incluyen SIGINT, SIGQUIT y SIGALRM. Otras señales del kernel dependen del apoyo de hardware. Por ejemplo, los procesadores 8086 y 8088 no apoyan la detección de códigos de operación no válidos en las instrucciones, pero esta capacidad está disponible en el 286 y superiores, que atrapan un intento por ejecutar un código de operación no permitido. Este servicio es proporcionado por el hardware. El implementador del sistema operativo debe incluir código para generar una señal en respuesta a la trampa. Ya vimos en el capítulo!

que `kernel/exception.c` contiene código para hacer precisamente esto en varias condiciones diferentes. Así, puede generarse una señal `SIGKILL` como respuesta a una instrucción no permitida cuando MINIX se ejecuta en un procesador 286 o superior, pero esta señal nunca se verá cuando MINIX se ejecute en un 8088.

El hecho de que el hardware pueda atrapar cierta condición no implica que el implementador del sistema operativo pueda aprovechar plenamente esta capacidad. Por ejemplo, varios tipos de violaciones de la integridad de la memoria producen excepciones en todos los procesadores Intel a partir del 286. El código de `kernel/exception.c` traduce estas excepciones a señales `SIGSEGV`. Se generan excepciones distintas si se violan los límites del segmento de pila definido por hardware y de otros segmentos, ya que estos casos podrían requerir un tratamiento diferente. Sin embargo, dada la forma como MINIX utiliza la memoria, el hardware no puede detectar todos los errores que podrían ocurrir. El hardware define una base y un límite para cada segmento. La base del segmento de datos definido por el usuario es la misma que la base del segmento de datos de MINIX, pero el límite del segmento de datos definido por el hardware es más alto que el límite que MINIX hace respetar en software. En otras palabras, el hardware define el segmento de datos como la cantidad máxima de memoria que MINIX podría llegar a usar para los datos, si de alguna manera la pila pudiera encogerse hasta desaparecer. De forma similar, el hardware define la pila como la cantidad máxima de memoria que la pila de MINIX podría usar si el área de datos pudiera encogerse hasta desaparecer. Aunque el hardware puede detectar ciertas violaciones, no puede detectar la violación de pila más probable, el crecimiento de la pila al interior del área de datos, ya que en lo que a los registros y tablas de descriptores del hardware concierne el área de datos y el área de pila se traslapan.

Sería concebible agregar código al kernel que verificara los registros de cada proceso cada vez que el proceso tiene oportunidad de ejecutarse, y generara una señal `SIGSEGV` en caso de detectar una violación de la integridad de las áreas de datos o de pila definidas por MINIX. Lo que no está claro es si valdría la pena; las trampas de hardware pueden detectar una violación de inmediato, pero una verificación en software tal vez no tendría oportunidad de hacer su trabajo sino hasta después de ejecutarse muchos miles de instrucciones adicionales, y a esas alturas probablemente sería muy poco lo que un manejador de señales podría hacer en un intento por recuperarse.

Sea cual sea el origen de las señales, el administrador de memoria las procesa todas del mismo modo. Para cada proceso al cual se va a enviar una señal, se realiza una serie de verificaciones para determinar si la señal es factible. Un proceso puede enviar una señal a otro si es el superusuario o si su `uid` real o efectivo es igual al `uid` real o efectivo del proceso que recibe la señal. No obstante, hay varias condiciones que pueden impedir el envío de una señal. No es posible enviar una señal a un zombi, por ejemplo. No se puede enviar una señal a un proceso si éste invocó específicamente `SIGACTION` para ignorar la señal o `SIGPROCMASK` para bloquearla. Bloquear una señal no es lo mismo que ignorarla; la recepción de una señal bloqueada se recuerda, y la señal se entrega cuando el proceso destinatario quita el bloque, si es que alguna vez lo hace. Por último, si el espacio de pila del proceso destinatario de la señal no es suficiente, el proceso se cancela.

Si se satisfacen todas las condiciones, se puede enviar la señal. Si el proceso no tomó medidas para atrapar la señal, no es necesario pasar información al proceso. En este caso el adminis-

trador de memoria ejecuta la acción predeterminada para la señal, que por lo regular es terminar el proceso, posiblemente produciendo también un vaciado de núcleo. En el caso de unas cuantas señales la acción predeterminada es hacer caso omiso de la señal. Las señales marcadas como "No se reconoce" en la Fig. 4-41 se definen porque POSIX así lo exige, pero MINIX las ignora.

Atrapar una señal significa ejecutar el código de manejo de señal personalizado del proceso. La dirección de este código está almacenada en una estructura sigaction en la tabla de procesos. En el capítulo 2 vimos cómo el marco de pila de un proceso dentro de su entrada de la tabla de procesos recibe la información necesaria para iniciar el proceso cuando éste es interrumpido. Si se modifica el marco de pila del proceso al que se va a enviar una señal, puede lograrse que la próxima vez que se permita al proceso ejecutarse se ejecutará el manejador de señales. Mediante la modificación de la pila propia del proceso en el espacio de usuario, se puede lograr que cuando el manejador de señales termine se emita la llamada al sistema SIGRETURN. Esta llamada al sistema nunca es invocada por código escrito por el usuario; se ejecuta después de que el kernel coloca su dirección en la pila de tal manera que su dirección se convierte en la dirección de retomo que se saca de la pila cuando un manejador de señales termina. SIGRETURN restaura el marco de pila original del proceso al que se envió la señal, a fin de que pueda reanudar su ejecución en el punto en que fue interrumpido por la señal.

Aunque la última etapa del envío de una señal corre por cuenta de la tarea del sistema, éste es un buen lugar para resumir cómo se hace, ya que es el administrador de memoria el que pasa al kernel los datos que se emplean. El atrapamiento de una señal requiere algo muy similar a la conmutación de contexto que ocurre cuando se suspende la ejecución de un proceso y se pone a ejecutar otro proceso, ya que cuando el manejador termine el proceso deberá poder continuar como si no hubiera sucedido nada. Sin embargo, sólo hay un lugar de la tabla de procesos donde se puede guardar el contenido de todos los registros de la CPU que se necesitan para restaurar el proceso a su estado original. La solución de este problema se muestra en la Fig. 4-42. La parte (a) de la figura es una vista simplificada de la pila de un proceso y de parte de su entrada en la tabla de procesos inmediatamente después de que se ha suspendido la ejecución de un proceso por una interrupción. En el momento de la suspensión el contenido de todos los registros de CPU se copia en la estructura de marco de pila de la entrada de] proceso en la parte de la tabla de procesos que corresponde al kernel. Ésta será la situación en el momento en que se genere una señal, ya que la ; señal es generada por un proceso o tarea diferente del destinatario de la señal. I

Como preparación para manejar la señal, el marco de pila de la tabla de procesos se copia en la pila propia del proceso como una estructura sigcontext, a fin de conservarlo. Luego se coloca I una estructura sigfmme en la pila. Esta estructura contiene información que será utilizada por | SIGRETURN cuando el manejador termine, y también contiene la dirección del procedimiento de | biblioteca que invoca SIGRETURN misma, ret addr1, y otra dirección de retomo, ret addr2, que es' la dirección donde se reanudará la ejecución del programa interrumpido. Sin embargo, como habremos de ver, la segunda dirección no se usa durante la ejecución normal.

Aunque el programador escribe el manejador como un procedimiento ordinario, no se invoca con una instrucción CALL. El campo de apuntad r de instrucción (contador de programa) del marco de pila en la tabla de procesos se altera de modo que haga que el manejador de señales comience a ejecutarse cuando restan coloque otra vez en ejecución el proceso que recibió la señal. La Fig. 4-42(b)

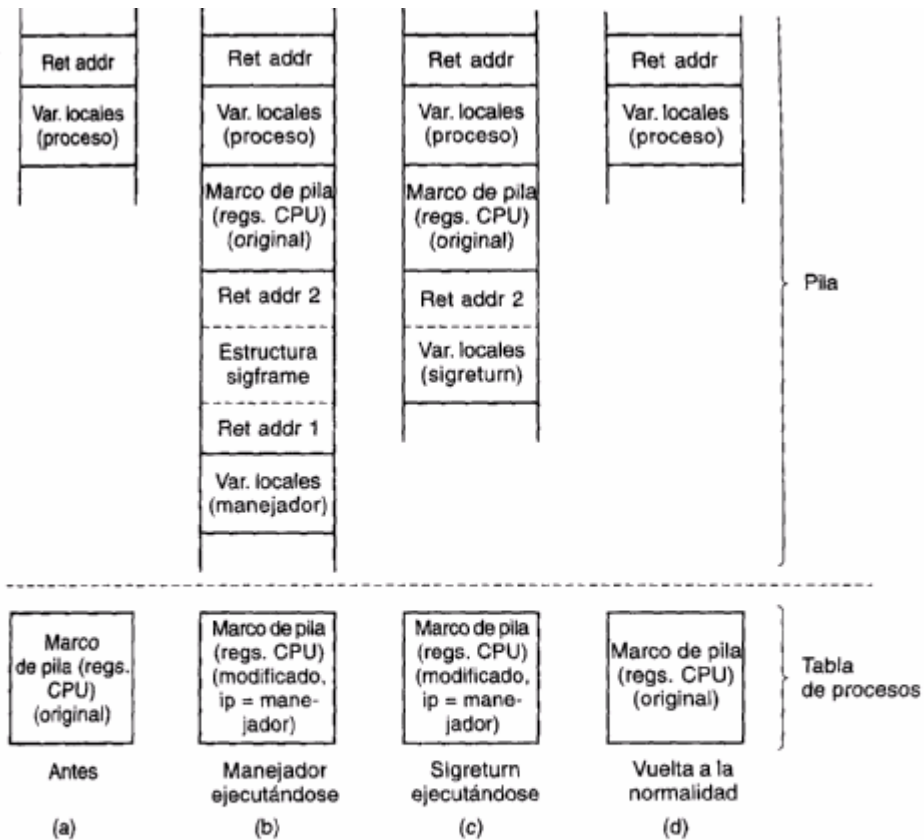


Figura 4-42. La pila de un proceso (arriba) y su marco de pila en la tabla de procesos (abajo) en las diferentes fases de manejo de una señal. (a) Estado cuando se suspende la ejecución del proceso. (b) Estado al iniciarse la ejecución del manejador. (c) Estado mientras se está ejecutando SIGRETURN. (d) Estado después de que termina de ejecutarse SIGRETURN.

también muestra la situación después de que se ha completado esta preparación y durante la ejecución del manejador de señales. Recuérdese que el manejador de señales es un procedimiento ordinario, así que cuando termina se remueve de la pila ret addrí y se ejecuta SIGRETURN.

La parte (c) muestra la situación mientras se está ejecutando SIGRETURN. El resto de la estructura sigframe consta ahora de las variables locales de SIGRETURN. Parte de la acción de SIGRETURN es ajustar su propio apuntador de pila de modo que, si llegara a terminar como una función ordinaria, utilizaría ret addr2 como dirección de retomo. Sin embargo, SIGRETURN realmente no termina de este modo; termina igual que otras llamadas al sistema, permitiendo al planificador del kernel decidir cuál proceso habrá de reiniciar. Tarde o temprano, el proceso que recibió la señal será replanificado y reiniciará en esta dirección, porque la dirección también está en el marco de pila original del proceso. La razón por la que esta dirección está en la pila es que un usuario podría querer rastrear un programa empleando un depurador, y esto engaña al depurador

haciéndolo que efectúe una interpretación razonable de la pila mientras se está rastreando un manejador de señales. En cada fase, la pila semeja la de un proceso ordinario, con variables locales encima de una dirección de retorno.

La verdadera labor de SIGRETURN es restaurar las cosas al estado en que estaban antes de que se recibiera la señal, y realizar aseo. Lo más importante es que el marco de pila en la tabla de procesos se restaura a su estado original, empleando la copia que se guardó en la pila del proceso que recibió la señal. Cuando SIGRETURN termina, la situación será como en la Fig. 4-42(d), que muestra al proceso esperando volver a entrar en ejecución en el mismo estado en el que estaba cuando fue interrumpido.

En el caso de la mayor parte de las señales, la acción predeterminada consiste en terminar el proceso al que se envía la señal. El administrador de memoria se encarga de esto para cualquier señal que no es ignorada por omisión, siempre que el proceso receptor no haya sido habilitado para manejar, bloquear o ignorar la señal. Si el padre del proceso que es terminado por la señal está esperándolo, el proceso terminado se asea y retira de la tabla de procesos. Si el padre no está esperando, el proceso terminado se convierte en zombi. En el caso de ciertos números de señal (p. ej., SIGQUIT), el administrador de memoria también escribe un vaciado de núcleo del proceso en el directorio actual.

Puede suceder fácilmente que se envíe una señal a un proceso que actualmente está bloqueado esperando un READ en una terminal para la cual no hay entradas disponibles. Si el proceso no especificó que la señal debe atraparse, simplemente se termina de la forma acostumbrada. En cambio, si la señal es atrapada, surge la cuestión de qué debe hacerse una vez que se ha procesado la interrupción de la señal. ¿El proceso debe seguir esperando o debe continuar con la siguiente instrucción?

Lo que MINIX hace es esto: la llamada al sistema se termina de tal manera que devuelve el; código de error EINTR, con lo que el proceso puede darse cuenta de que la llamada fue interrumpida por una señal. No es del todo trivial determinar si un proceso al que se envió una señal estaba bloqueado en una llamada al sistema. El administrador de memoria debe interrogar al sistema de archivos para verificarlo.

POSIX sugiere, pero no exige, este comportamiento, que también permite a un READ devolver el número de bytes leídos hasta el momento en que se recibió la señal. La devolución de EINTR hace posible establecer una alarma y atrapar SIGALRM. Ésta es una forma sencilla de implementar un tiempo de espera, por ejemplo para terminar login y colgar una línea de módem si un usuario no responde dentro de cierto periodo. Se puede usar la tarea de reloj síncrona para hacer lo mismo; con menos gasto extra, pero dicha tarea es una invención de MINIX y no es portátil como el empleo de señales. Además, esa tarea sólo está disponible para los procesos servidores, no para los procesos de usuario ordinarios.

4.7.8 Otras llamadas al sistema

El administrador de memoria maneja otras pocas llamadas al sistema sencillas. Las funciones biblioteca `getuid` y `geteuid` invocan la llamada al sistema `GETUID`, que devuelve ambos valores su mensaje de retorno. De forma análoga, la llamada al sistema `GETGID` devuelve los valores `i` y `e` que utilizan las funciones `getgid` y `getegid`. `GETPID` funciona del mismo modo]

devolver tanto el identificador del proceso como el identificador de su padre, y SETUID y SETGID pueden establecer los valores tanto real como efectivo en una llamada. Hay otras dos llamadas al sistema en este grupo, GETPGRP y SETSID. La primera devuelve el identificador del grupo de procesos, y la segunda le asigna el valor del pid actual. Estas siete llamadas son las llamadas al sistema más sencillas de MINIX.

El administrador de memoria también maneja las llamadas al sistema PTRACE y REBOOT. La primera apoya la depuración de los programas. La segunda afecta muchos aspectos del sistema; resulta apropiado colocarla en el administrador de memoria porque su primera acción es enviar señales para terminar todos los procesos con excepción de init. Luego, REBOOT recurre al sistema de archivos y a la tarea del sistema para completar su trabajo.

4.8 IMPLEMENTACIÓN DE LA ADMINISTRACIÓN DE MEMORIA EN MINIX

Ahora que sabemos en términos generales cómo funciona el administrador de memoria, pasemos al código mismo. El administrador de memoria está escrito totalmente en C, es fácil de entender y contiene cantidades sustanciales de comentarios en el código mismo, así que nuestro tratamiento de casi todas sus partes no tiene por qué ser largo ni rebuscado. Primero examinaremos brevemente los archivos de cabecera, luego el programa principal y por último los archivos para los diversos grupos de llamadas al sistema que describimos anteriormente.

4.8.1 Los archivos de cabecera y estructuras de datos

Varios archivos de cabecera en el directorio fuente del administrador de memoria tienen los mismos nombres que otros archivos en el directorio del kernel, y veremos otra vez estos nombres en el sistema de archivos. Los archivos correspondientes tienen funciones similares en sus propios contextos. Tal estructura paralela tiene como propósito facilitar el entendimiento de la organización de todo el sistema MINIX. El administrador de memoria tiene además varias cabeceras con nombres únicos. Al igual que en otras partes del sistema, se reserva espacio para las variables globales cuando se compila la versión de table.c del administrador de memoria. En esta sección estudiaremos todos los archivos de cabecera, además de table.c.

Al igual que en las otras partes principales de MINIX, el administrador de memoria tiene un archivo de cabecera maestro, mm.h (línea 15800). Éste se incluye en todas las compilaciones, y a su vez incluye todos los archivos de cabecera en el nivel de sistema de /usr/include y sus subdirectorios, mismos que necesitan todos los módulos objeto. La mayor parte de los archivos que se incluyen en kernel/kernel.h también se incluyen aquí. Además, el administrador de memoria necesita las definiciones contenidas en include/fcntl.h e include/unistd.h. También se incluyen las versiones de const.h, type.h, proto.h y glo.h propias del administrador de memoria.

Const.h (línea 15900) define algunas constantes utilizadas por el administrador de memoria, sobre todo cuando se compila para máquinas de 16 bits. La línea

```
#define printf printk
```

está contenida aquí con objeto de que las llamadas a `printf` se compilen como llamadas a la función `printf`. La función es similar a la que vimos en el kernel, y se define por una razón parecida para que el administrador de memoria pueda exhibir mensajes de error y depuración sin pe ayuda al sistema de archivos.

`Type.h` no se utiliza actualmente y existe en forma de esqueleto para que los archivos (administrador de memoria) tengan la misma organización que las demás partes de MINIX. `Proh` (línea 16100) reúne en un solo lugar prototipos de funciones que se requieren a todo lo largo del administrador de memoria.

Las variables globales del administrador de memoria se declaran en `glo.h` (línea 16200). usa aquí el mismo truco que se usó en el kernel con `EXTERN`, a saber, `EXTERN` normalmente una macro que se expande a `extern`, excepto en el archivo `table.c`. Ahí, se convierte en la cadena nula a fin de poder reservar espacio para las variables declaradas como `EXTERN`.

La primera de estas variables, `mp`, es un apuntador a una estructura `mproc`, la parte de la tabla de procesos del administrador de memoria correspondiente al proceso cuya llamada al sistema está procesando. La segunda variable, `dont_reply` (no contestar), se inicializa en `FALSE` cuando llega una solicitud nueva pero se le puede asignar `TRUE` durante la llamada si se descubre que no debe enviarse ningún mensaje de respuesta. Por ejemplo, no se envían respuestas cuando `i EXEC` tiene éxito. La tercera variable, `procs_in_use`, lleva la cuenta de cuántas ranuras de proceso se están usando actualmente, lo que facilita determinar si una llamada `FORK` es factible.

Los buffers de mensajes `mm_in` y `mm_out` son para los mensajes de solicitud y de respuesta respectivamente. `who` es el índice del proceso en curso; está relacionado con `mp` mediante

```
mp = &mproc[who];
```

Cuando llega un mensaje, se extrae de él el número de llamada al sistema y se coloca en `mm_caj`; Las tres variables `err_code`, `resultl` y `res_ptr` sirven para contener valores que se devuelven al invocador en el mensaje de respuesta. La más importante de estas variables es `err_code`, a 1 que se asigna `OK` si la llamada se lleva a cabo sin error. Las últimas dos variables se usan si surge un problema. `MINIX` escribe una imagen de un proceso en un archivo de núcleo cuando el proceso termina anormalmente. `Core_name` define el nombre que tendrá este archivo, y `core_sset` es un mapa de bits que define cuáles señales deben producirse vaciados de núcleo.

La parte de la tabla de procesos que corresponde al administrador de memoria está en el siguiente archivo, `mproc.h` (línea 16300). La mayor parte de los campos están descritos con claridad por sus comentarios. Varios campos tienen que ver con el manejo de señales. `mp_ignore`, `mp_catch`, `mp_sigmask`, `mp_sigmask2` y `mp_sigpending` son mapas de bits en los que cada bit representa una de las señales que se pueden enviar a un proceso. El tipo `sigset_t` es un entero de 32 bits, así que `MINIX` podría reconocer fácilmente hasta 32 señales, aunque actualmente sólo están definidas 16 correspondiendo a la señal 1 el bit menos significativo (el de la extrema derecha). En cualquier caso `POSIX` requiere funciones estándar para agregar o eliminar miembros de los conjuntos de señales representados por estos mapas de bits, así que todas las manipulaciones necesarias pueden efectuarse sin que el programador esté consciente de estos detalles. El arreglo `mp_sigact` es importante para manejar señales. Hay un elemento para cada tipo de señal, y cada elemento es una estructura `sigactim` (definida en `include/signal.h`). Cada estructura `sigaction` consta de tres campos:

1. El campo `sa_handler` define si la señal debe manejarse de la forma predeterminada, ignorarse o manejarse con un manejador especial.

2. El campo `sa_mask` es un `sigset_t` que define cuáles señales deben bloquearse cuando la señal esté siendo manejada por un manejador personalizado.

3. El campo `sa_flags` es un conjunto de banderas que se aplican a la señal.

Este arreglo hace posible una gran flexibilidad en el manejo de señales.

El campo `mp_flags` sirve para contener una colección diversa de bits, como se indica al final Del archivo. Este campo es un entero sin signo, de 16 bits en las CPU antiguas o de 32 bits en una 386 o superior. Hay mucho espacio para expansión aquí, incluso en una 8088, ya que sólo se usan nueve bits.

El último campo de la tabla de procesos es `mp_procargs`. Cuando se inicia un proceso nuevo, se construye una pila como la que se muestra en la Fig. 4-39, y se almacena aquí un apuntador al principio del arreglo `argv` del nuevo proceso. El comando `ps` usa este apuntador. Por ejemplo, en el caso que se ilustra en la Fig. 4-39, se guardaría aquí el valor 8164, con lo que se podría exhibir la línea de comandos

```
Is -l f.c g.c
```

si se ejecuta mientras está activo el comando `Is`.

El siguiente archivo es `param.h` (línea 16400), que contiene macros para muchos de los parámetros de llamadas al sistema contenidos en el mensaje de solicitud. Este archivo también contiene cuatro macros para campos del mensaje de respuesta. Cuando aparece el enunciado

```
k = pid;
```

en cualquier archivo en el que se ha incluido `param.h`, el preprocesador lo convierte en

```
k=mm_in.ml_i1;
```

antes de alimentarlo al compilador propiamente dicho.

Antes de continuar con el código ejecutable, examinemos `table.c` (línea 16500). Su compilación reserva espacio para las diversas variables y estructuras `EXTERN` que hemos visto en `glo.h` y `mproc.h`. El enunciado

```
#define TABLE
```

hace que `EXTERN` se convierta en la cadena nula. Este es el mismo mecanismo que vimos en el código del kernel.

La otra característica importante de `table.c` es el arreglo `call_vec` (línea 16515). Cuando llega un mensaje de solicitud, se extrae de él el número de llamada al sistema y se usa como índice para encontrar en `call_vec` el procedimiento que lleva a cabo esa llamada al sistema. Todos los números de llamada al sistema que no son válidos invocan a `no_sys`, que se limita a devolver un código de error. Adviértase que aunque se usa la macro `JPROTOTYPE` para definir `call_vec`, no se trata de la declaración de un prototipo; es la definición de un arreglo inicializado. No obstante, `call_vec` es

un arreglo de funciones, y el uso de `_PROTOTYPE` es la forma más fácil de hacer esto que es compatible tanto con el C clásico (Kemighan & Ritchie) como con Standard C.

4.8.2 El programa principal

El administrador de memoria (MM) se compila y enlaza de forma independiente respecto al kernel y el sistema de archivos; en consecuencia, tiene su propio programa principal, que se inicia una vez que el kernel ha terminado de inicializarse a sí mismo. El programa principal está en `main.c`, en la línea 16627. Después de efectuar su propia inicialización llamando a `mm_init`, el administrador de memoria ingresa en su ciclo en la línea 16636. En este ciclo, el MM invoca `get_work` para esperar la llegada de un mensaje de solicitud, luego invoca uno de sus procedimientos `do_XXX` a través de la tabla `call_vec` para atender la solicitud, y por último envía una respuesta, si es necesario. Esta construcción ya debe ser conocida a estas alturas: es la misma que emplean las tareas de E/S.

Los procedimientos `get_work` (línea 16663) y `reply` (línea 16676) se encargan de la recepción y el envío, respectivamente.

El último procedimiento de este archivo es `mm_init`, que inicializa el administrador de memoria y que no se usa después de que el sistema ha entrado en funcionamiento. La llamada a `sys_getmap` en la línea 16730 obtiene información acerca del uso de memoria del kernel. El ciclo de las líneas 16734 a 16741 inicializa todas las entradas de la tabla de procesos para las tareas y los servidores, y las siguientes líneas preparan la entrada de `init` en la tabla de procesos. En la línea 16749 el MM espera que el sistema de archivos (FS) le envíe un mensaje. Como se mencionó cuando hablamos del manejo de bloqueos en MINIX, éste es el único momento en que el ES envía un mensaje de solicitud al administrador de memoria. El mensaje indica cuánta memoria se está usando para el disco en RAM. La llamada a `mem_init` en la línea 16755 inicializa la lista de agujeros invocando la tarea del sistema. Después de esto ya puede comenzar la administración normal de la memoria. Esta llamada también asigna valores a las variables `total_clicks` y `free_clicks`, completando la información que `mm_init` necesita para exhibir un mensaje indicando la memoria total, el uso de memoria por el kernel, el tamaño del disco en RAM y la memoria libre. Después de exhibir el mensaje se envía una respuesta al FS (línea 16764) para permitirle que continúe. Por último, se proporciona a la tarea de memoria la dirección de la parte de la tabla de procesos que corresponde al MM, a fin de que el comando `ps` cuente con esa información.

4.8.3 Implementación de FORK, EXIT y WAIT

Las llamadas al sistema `FORK`, `EXIT` y `WAIT` se implementan con los procedimientos `do_fork`, `do_mm_exitydo_wait` que están en el archivo `forkexit.c`. El procedimiento `do_fork` (línea 16832) sigue los pasos que se muestran en la Fig. 4-37. Observe que la segunda llamada a `procs_m_use` (línea 16847) reserva unas cuantas ranuras al final de la tabla de procesos para el superusuario. Al calcular cuánta memoria necesita el hijo, se incluye el espacio entre los segmentos de datos y de pila, pero no se incluye el segmento de texto. O bien se comparte el texto del padre, o, si el

proceso tiene espacio 1 y D común, su segmento de texto tiene longitud cero. Después de efectuar el cálculo, se invoca `alloc_mem` para obtener la memoria. Si esta asignación tiene éxito, las direcciones base del hijo y del padre se convierten de clics a bytes absolutos, y se invoca `sys_copy` para enviar un mensaje a la tarea del sistema, la cual se encargará del copiado.

Ahora se encuentra una ranura en la tabla de procesos. La prueba previa en la que intervino `procs_in_use` garantiza que habrá una ranura. Una vez localizada ésta, se llena copiando primero la ranura del padre ahí y luego actualizando los campos `mp_parent`, `mp_flags`, `mp_seg`, `mp_exitstatus` y `mp_sigstatus`. Algunos de estos campos requieren un manejo especial. El bit TRACED del campo `mp_flags` se pone en cero, ya que un hijo no hereda la situación de rastreo. El campo `mp_seg` es un arreglo que contiene elementos para los segmentos de texto, datos y pila, y se deja que la porción de texto siga apuntando al segmento de texto del padre si las banderas indican que se trata de un programa con I y D separado que puede compartir el texto.

El siguiente paso es asignar un pid al hijo. La variable `next_pid` contiene el siguiente pid por asignar. Sin embargo, es concebible que ocurra el siguiente problema. Después de asignarse, digamos, el pid 20 a un proceso de muy larga vida, podrían crearse y destruirse 30 000 procesos más, y `next_pid` podría regresar otra vez a 20. La asignación de un pid que todavía se está usando sería un desastre (supongamos que alguien trata de enviar una señal al proceso 20), así que efectuamos una búsqueda en toda la tabla de procesos para asegurarnos de que el pid que se va a asignar no está todavía en uso.

Las llamadas a `sys_fork` y `tell_fs` informan al kernel y al sistema de archivos, respectivamente, que se ha creado un proceso nuevo, a fin de que puedan actualizar sus tablas de procesos. (Todos los procedimientos que comienzan con `sys_` son rutinas de biblioteca que envían un mensaje a la tarea del sistema en el kernel para solicitar uno de los servicios de la Fig. 3-50). La creación y destrucción de procesos siempre son iniciadas por el administrador de memoria y luego se propagan al kernel y al sistema de archivos una vez que se completan.

El mensaje de respuesta al hijo se envía explícitamente al final de `do_fork`. La respuesta al padre, que contiene el pid del hijo, es enviada por el ciclo de `main`, como respuesta normal a una solicitud.

La siguiente llamada al sistema que el administrador de memoria maneja es `EXIT`. El procedimiento `do_mm_exit` (línea 16912) acepta la llamada, pero casi todo el trabajo corre por cuenta de la llamada a `mm_exit`, unas cuantas líneas más abajo. La razón por la que el trabajo se divide de esta forma es que `mm_exit` también se invoca para encargarse de los procesos que han sido terminados por una señal. El trabajo es el mismo, pero los parámetros son diferentes, así que resulta conveniente dividir las cosas de esta manera.

Lo primero que `mm_exit` hace es detener el temporizador, si el proceso tiene uno en funcionamiento. A continuación, se notifica al kernel y al sistema de archivos que el proceso ya no es ejecutable (líneas 16949 y 16950). La llamada al procedimiento de biblioteca `sys_xit` envía un mensaje a la tarea del sistema diciéndole que marque el proceso como no ejecutable, a fin de que yano se le calendarice. Acto seguido, se libera la memoria. Una llamada `Sifind_share` determina si el segmento de texto está siendo compartido por otro proceso y, si no es así, el segmento de texto ; se libera con una llamada `Sifree_mem`. Esto va seguido de otra llamada al mismo procedimiento para liberar los datos y la pila. No vale la pena decidir si toda la memoria podría liberarse con una

sola llamada `afree_mem` o no. Si el padre está esperando, se invoca `cleanup` para liberar la ranura de la tabla de procesos; si no, el proceso se convierte en un zombi, lo cual se indica con el bit `HANGING` de la palabra `mp_flags`. Sea que el proceso se elimine por completo o se convierta en zombi, la acción final de `mm_exit` es recorrer la tabla de procesos en busca de los hijos del proceso que acaba de terminarse (líneas 16975 a 16982). Si se encuentra algún hijo, se le deshace, convirtiéndolo en hijo de `init`. Si `init` está esperando y un hijo está pendiente, se invoca `cleanup` para ese hijo. Esto resuelve las situaciones como las de la Fig. 4-43(a). En esta figura vemos que el proceso 12 está a punto de salir, y que su padre, 7, está esperándolo. Se invocará `cleanup` para deshacerse de 12, con lo cual 52 y 53 se convierten en hijos de `init`, como se muestra en la Fig. 4-43(b). Ahora tenemos una situación en la que 53, que ya salió, es el hijo de un proceso que está ejecutando `WAIT`; por tanto, también puede someterse a aseo con `cleanup`.

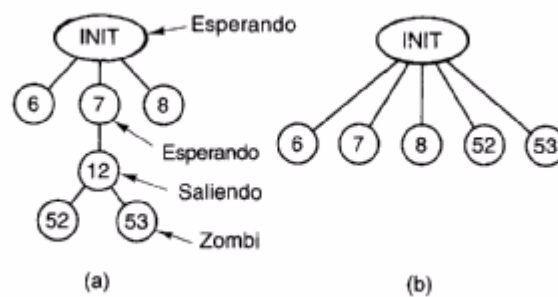


Figura 4-43. (a) La situación cuando el proceso 12 está a punto de salir. (b) La situación una vez que ha salido

Cuando el proceso padre ejecuta `WAIT` o `WAITPID`, el control pasa al procedimiento `do_waitpid` en la línea 16992. Los parámetros que proporcionan las dos llamadas son diferentes, y las acciones esperadas también son diferentes, pero la preparación efectuada en las líneas 17009 a 17011 establece variables internas de modo que `do_waitpid` pueda realizar las acciones de cualquiera de las dos llamadas. El ciclo de las líneas 17019 a 17041 examina toda la tabla de procesos para ver si el proceso tiene hijos y, de ser así, verifica si cualesquiera de ellos son zombis que puedan someterse a aseo. Si se encuentra un zombi (línea 17026), se somete a aseo y `do_waitpid` regresa. La bandera `dont_reply` se enciende porque la respuesta al padre se envía desde el interior de `cleanup`, no desde el ciclo de `main`. Si se encuentra un hijo rastreado, se envía una respuesta indicando que el proceso está detenido, y `do_waitpid` regresa. También se enciende `dont_reply` para evitar que `main` envíe una segunda respuesta.

Si el proceso que está ejecutando `WAIT` no tiene hijos, simplemente regresa con un error (línea 17053). Si el proceso tiene hijos, pero ninguno de ellos es un zombi o está siendo rastreado, se efectúa una prueba para ver si `do_waitpid` se invocó con un bit encendido indicando que el padre no quería esperar. Si no fue así (el caso normal), se enciende un bit en la línea 17047 para indicar que el padre está esperando, y éste se suspende hasta que el hijo termina.

Cuando un proceso ya salió y su padre lo está esperando, sea cual sea el orden en que ocurran ; estos sucesos, se invoca el procedimiento `cleanup` (línea 17061) para administrar la extremaim-

ción. No hay mucho que pueda hacerse a estas alturas. Se despierta al padre de su llamada WAIT o WAITPID y se le proporciona el pid del hijo que terminó, así como su situación de salida y de señales. El sistema de archivos ya liberó la memoria del hijo, y el kernel ya suspendió su planificación, así que lo único que tiene que hacer ahora el kernel es liberar la ranura del hijo en la tabla de procesos.

4.8.4 Implementación de EXEC

El código para EXEC se ajusta a la descripción resumida de la Fig. 4-38 y está contenido en el procedimiento `do_exec` (línea 17140). Después de efectuar unas cuantas verificaciones de validez sencillas, el administrador de memoria obtiene del espacio de usuario el nombre del archivo por ejecutar. En la línea 17172 el MM envía un mensaje especial al sistema de archivos para cambiar al directorio de usuario, con objeto de que la trayectoria que acaba de obtenerse se interprete en relación con el directorio de trabajo del usuario, no al del MM.

Si el archivo está presente y es ejecutable, el administrador de memoria lee la cabecera para extraer los tamaños de los segmentos. A continuación, el MM obtiene la pila del espacio de usuario (líneas 17188 y 17189), verifica si el nuevo proceso puede compartir el texto con un proceso que ya se está ejecutando (línea 17196), asigna memoria para la nueva imagen (línea 17199), repara los apuntadores [véanse las diferencias entre la Fig. 4-39(b) y (c)] y lee del disco el segmento de texto (si es necesario) y el segmento de datos (líneas 17221 a 17226). Por último, el MM procesa los bits `setuid` y `setgid`, actualiza la entrada de la tabla de procesos y le informa al kernel que ya terminó, a fin de que el proceso pueda calendarizarse otra vez.

Aunque el control de todos los pasos está en `do_exec`, muchos de los detalles corren por cuenta de procedimientos subsidiarios dentro de `exec.c`. `Read_header` (línea 17272), por ejemplo, no sólo lee la cabecera y devuelve los tamaños de segmento, sino también verifica que el archivo sea un ejecutable válido de MINIX para el mismo tipo de CPU para el cual está compilado el sistema operativo. Esto se hace mediante una compilación condicional de la prueba apropiada en el momento en que se compila el administrador de memoria (líneas 17322 a 17327). `Read_header` también verifica que todos los segmentos quepan en el espacio de direcciones virtual.

El procedimiento `new_mem` (línea 17366) comprueba que haya suficiente memoria disponible para la nueva imagen de memoria. Este procedimiento busca un agujero con el tamaño suficientes para los datos y la pila únicamente si el texto está siendo compartido; en caso contrario, busca un agujero único en el que puedan caber el texto, los datos y la pila combinados. Una posible mejora aquí sería buscar dos agujeros distintos, uno para el texto y otro para los datos y la pila, ya que no hay necesidad de que estas áreas sean contiguas. En versiones anteriores de MINIX esto era obligatorio. Si se encuentra suficiente memoria, se libera la memoria anterior y se adquiere la nueva. Si no hay suficiente memoria disponible, la llamada EXEC falla. Una vez que se ha asignado la nueva memoria, `new_mem` actualiza el mapa de memoria (en `mp_seg`) e informa de ello al kernel invocando el procedimiento de biblioteca `sys_newmap`.

El resto de `new_mem` se ocupa de poner en ceros el segmento `bss`, el espacio y el segmento de pila. (El segmento `bss` es la parte del segmento de datos que contiene todas las variables globales no inicializadas.) Muchos compiladores generan código explícito para poner en ceros el

segmento bss, pero al hacer esto aquí se logra que MINIX funcione incluso con compiladores que no lo hacen. El espacio entre los segmentos de datos y de pila también se pone en ceros, de modo que cuando BRK extienda el segmento de datos la memoria recién adquirida contenga ceros. Esto no sólo es cómodo para el programador, quien puede estar seguro de que las nuevas variables tendrán un valor inicial de cero, también es una característica de seguridad en un sistema operativo multiusuario, en el que un proceso que utilizó previamente esta memoria podría haber estado usando datos que otros procesos no deben ver.

El siguiente procedimiento es `patch_ptr` (línea 17465), que se encarga de reubicar los apuntadores de la Fig. 4-39(b) a la forma de la Fig. 4-39(c). El trabajo es sencillo: se examina la pila hasta encontrar todos los apuntadores y se suma la dirección base a cada uno.

El procedimiento `load_seg` (línea 17498) se invoca una o dos veces en cada EXEC, posiblemente para cargar el segmento de texto y siempre para cargar el segmento de datos. En lugar de limitarse a leer el archivo bloque por bloque y luego copiar los bloques al usuario, se emplea un truco para permitir que el sistema de archivos cargue todo el segmento directamente en el espacio de usuario. En efecto, la llamada es descodificada por el sistema de archivos de una forma un tanto especial con objeto de que parezca ser una lectura del segmento completo por parte del proceso de usuario mismo. Sólo unas cuantas líneas al principio de la rutina de lectura del sistema de archivos saben que aquí está ocurriendo algo sospechoso. Esta maniobra agiliza considerablemente la carga.

El procedimiento final de `exec.c` es `esfind_share` (línea 17535), el cual busca un proceso capaz de compartir texto comparando el `nodo-i`, el dispositivo y los tiempos de modificación del archivo por ejecutar con los de los procesos existentes. Se trata de una búsqueda directa de los campos apropiados de `mproc`. Desde luego, no debe incluirse en la búsqueda el proceso a nombre del cual se está efectuando.

4.8.5 Implementación de BRK

Como acabamos de ver, el modelo de memoria que MINIX emplea es muy sencillo: cada proceso recibe una sola asignación de memoria contigua para sus datos y su pila cuando se crea. El proceso nunca se cambia de lugar en la memoria, nunca se intercambia a disco, nunca crece y nunca se encoge. Lo único que puede suceder es que el segmento de datos reduzca el espacio desde su extremo inferior y que la pila lo reduzca desde su extremo superior. En estas circunstancias, la implementación de la llamada BRK en `break.c` es muy fácil; consiste en verificar que los nuevos tamaños sean factibles y actualizar después las tablas de modo que los reflejen.

El procedimiento de nivel más alto es `do_brk` (línea 17628), pero la mayor parte del trabajo se efectúa en `adjust` (línea 17661). Este último verifica si los segmentos de datos y de pila chocan. Si así fue, no es posible llevar a cabo el BRK, pero el proceso no se termina de inmediato. Un factor de seguridad, `SAFETY_BYTES`, se agrega a la parte superior del segmento de datos antes de realizar la prueba, de modo que (con suerte) la decisión de que la pila ha crecido demasiado puede tomarse mientras todavía hay suficiente espacio en la pila para que el proceso continúe durante un tiempo corto. El proceso recupera el control (recibiendo un mensaje de error) para que pueda imprimir los mensajes apropiados y después termine de forma ordenada.

Observe que SAFETYJBYTES se define empleando una instrucción #define a la mitad del procedimiento (línea 17693). Este uso es poco común; normalmente tales definiciones aparecen al principio de los archivos, o en archivos de cabecera aparte. El comentario correspondiente revela que al programador se le hizo difícil decidir qué tamaño debía tener el factor de seguridad. No hay duda de que la definición se realizó de esta manera con objeto de atraer la atención y, tal vez, estimular experimentos adicionales.

La base del segmento de datos es constante, así que si adjust se ve obligado a ajustar el segmento de datos lo único que hace es actualizar el campo de longitud. La pila crece hacia abajo desde un extremo fijo, así que si adjust también se da cuenta de que el apuntador de la pila, que recibe como parámetro, ha crecido más allá del segmento de pila (a una dirección situada más abajo), actualiza tanto el origen como la longitud.

El último procedimiento de este archivo, si7.e_ok (línea 17736) efectúa una prueba para determinar si los tamaños de los segmentos caben dentro del espacio de direcciones, tanto en clics como en bytes. Se conservó en el listado el código condicional para máquinas de 16 bits con el propósito de mostrar por qué se escribió en forma de función individual. No tendría mucho caso tener este código como función individual para MINIX de 32 bits. Este procedimiento se invoca en sólo dos lugares, y la sustitución de la línea 17765 en lugar de las llamadas produciría código más compacto, ya que las llamadas pasan varios argumentos que no se usan en la implementación de 32 bits.

4.8.6 Implementación del manejo de señales

Hay ocho llamadas al sistema relacionadas con señales, las cuales se resumen en la Fig. 4-44. Estas llamadas, así como las señales mismas, se procesan en el archivo signal.c. Este archivo maneja también una llamada al sistema adicional, REBOOT, ya que emplea señales para terminar todos los procesos.

Llamada al sistema	Propósito
SIGACTION	Modifica la respuesta a una señal futura
SIGPROCMASK	Cambia el conjunto de señales bloqueadas
KILL	Envía una señal a otro proceso
ALARM	Se envía a sí misma la llamada ALRM después de un retardo
PAUSE	Se suspende a sí misma hasta una señal futura
SIGSUSPEND	Cambia el conjunto de señales bloqueadas, luego PAUSE
SIGPENDING	Examina el conjunto de señales pendientes (bloqueadas)
SIGRETURN	Hace aseo al terminar el manejador de señales

Figura 4-44. Llamadas al sistema relacionadas con señales.

La llamada SIGACTION apoya las funciones sigaction y signal, que permiten a un proceso alterar la forma como responde a las señales. POSIX exige sigaction y es la llamada preferida en casi todos los casos, pero Standard C requiere la función de biblioteca signal, y los programas que deben ser trasladables a sistemas no POSIX deben escribirse usándola. El código de do_sigaction (línea 17845) comienza con verificaciones de la validez del número de señal y de que la llamada no es un intento por modificar la respuesta a una señal SIGKILL (líneas 17851 y 17852). (No se permite ignorar, atrapar o bloquear SIGKILL. SIGKILL es el mecanismo definitivo mediante el cual un usuario puede controlar sus procesos y un administrador del sistema puede controlar a sus usuarios.) SIGACTION se invoca con apuntadores a una estructura sigaction, sig_osa, que recibe los antiguos atributos de señal que estaban vigentes antes de la llamada, y a una estructura del mismo tipo, sig_nsa, que contiene un nuevo conjunto de atributos.

El primer paso es invocar la tarea del sistema para que copie los atributos actuales en la estructura a la que sig_osa apunta. Se puede invocar SIGACTION con un apuntador NULL en sig_nsa si lo que se desea es examinar los antiguos atributos de señal sin modificarlos. En este caso do_sigaction regresa de inmediato (línea 17860). Si sig_nsa no es NULL, la estructura que define la nueva forma de reaccionar a las señales se copia en el espacio del administrador de memoria. El código de las líneas 17867 a 17877 modifica los mapas de bits mp_catch, mpIgnore y mp_sigpending de modo que reflejen la nueva acción, que puede ser ignorar la señal, usar el manejador predeterminado o atrapar la señal. Se emplean las funciones de biblioteca sigaddset y sigdelset, aunque las acciones son operaciones de manipulación de bit directas que podrían haberse implementado con macros sencillas. Sin embargo, el estándar POSIX requiere estas funciones a fin de asegurar que los programas que las usen puedan trasladarse fácilmente, incluso a sistemas en los que el número de señales exceda el número de bits disponibles en un entero. El empleo de las funciones de biblioteca ayuda a lograr que MINIX sea fácilmente transportable a diferentes arquitecturas.

Por último, se llenan los demás campos relacionados con señales en la parte de la tabla de procesos que corresponde al administrador de memoria. Para cada posible señal hay un mapa de bits, sa_mask, que define cuáles señales deben bloquearse mientras se está ejecutando un manejador de esa señal. También hay un apuntador, sa_handler, para cada posible señal, el cual puede contener un apuntador a la función de manejo, o valores especiales para indicar que la señal debe ignorarse o manejarse de la forma predeterminada. La dirección de la rutina de biblioteca que invoca SIGRETURN cuando el manejador termina está almacenada en mp_sigreturn. Esta dirección es uno de los campos del mensaje que el administrador de memoria recibe.

POSIX permite que un proceso manipule su propio manejo de señales, incluso estando dentro de un manejador de señales. Esto puede servir para modificar la respuesta a señales subsecuentes mientras se está procesando una señal, y luego restablecer el conjunto normal de respuestas. El siguiente grupo de llamadas al sistema apoya estas capacidades de manipulación de señales. SIGPENDING se maneja con do_sigpending (línea 17889), que devuelve el mapa de bits mp_sigpending para que un proceso pueda determinar si tiene señales pendientes. SIGPROCMAK, que se maneja con do_sigprocmask, devuelve el conjunto de señales que actualmente están bloqueadas, y también puede servir para cambiar el estado de una sola señal del conjunto, o sustituir todo el conjunto por uno nuevo. El momento en el que una señal se desbloquea es el punto apropiado para

verificar si hay señales pendientes, y esto se hace con llamadas a `check_pending` en la línea 17927 y en la 17933. `Do_sigsuspend` (línea 17949) ejecuta la llamada al sistema `SIGSUSPEND`. Esta llamada suspende un proceso hasta que se recibe una señal. Al igual que las otras funciones que hemos visto aquí, ésta manipula mapas de bits, y además enciende el bit `SIGSUSPENDED` de `mp_flags`, que es lo único que se necesita para impedir la ejecución de un proceso. Una vez más, éste es un buen momento para invocar `check_pending`. Por último, `do_sigreturn` maneja `SIGRETURN`, que sirve para regresar de un manejador personalizado. Esta función restablece el contexto de señales que existía cuando se ingresó en el manejador, y también invoca `check_pending` en la línea 17980.

Algunas señales, como `SIGINT`, se originan en el kernel mismo. Tales señales se manejan de forma similar a las señales generadas por un proceso de usuario que invoca `KILL`. Los dos procedimientos, `do_kill` (línea 17983) y `do_ksig` (línea 17994) son conceptualmente similares; ambos hacen que el administrador de memoria envíe una señal. Una sola llamada a `KILL` puede requerir el envío de señales a un grupo de procesos, y `do_kill` se limita a invocar `check_sig`, que examina toda la tabla de procesos en busca de destinatarios elegibles. Se invoca `do_ksig` cuando llega un mensaje del kernel. El mensaje contiene un mapa de bits, lo que permite al kernel generar múltiples señales con un solo mensaje. Al igual que con `KILL`, cualquiera de éstas podría tenerse que entregar a un grupo de procesos. El mapa de bits se procesa bit por bit en el ciclo de las líneas 18026 a 18048. Algunas señales del kernel requieren atención especial: en algunos casos se cambia el identificador de proceso para hacer que la señal se entregue a un grupo de procesos (líneas 18030 a 18033), y se hace caso omiso de un `SIGALRM` si no fue solicitado. Con esa excepción, cada bit encendido causa una llamada a `check_sig`, tal como sucede en `do_kill`.

La llamada al sistema `ALARM` está bajo el control de `do_alarm` (línea 18056). Ésta invoca la siguiente función, `set_alarm`, que envía un mensaje a la tarea del reloj pidiéndole que ponga en marcha el temporizador. `Set_alarm` (línea 18067) es una función aparte porque también se usa para apagar el temporizador cuando un proceso sale con el temporizador encendido aún. Cuando el temporizador llega a cero, el kernel anuncia el hecho enviando al administrador de memoria un mensaje del tipo `KSIG`, que causa la ejecución de `do_ksig`, como ya se explicó. La acción predeterminada de la señal `SIGALRM` es terminar el proceso si no es atrapada. Si se desea atrapar `SIGALRM`, `SIGACTION` debe instalar un manejador. La secuencia completa de sucesos para una señal `SIGALRM` con un manejador personalizado se muestra en la Fig. 4-45. Hay tres secuencias de mensajes aquí. En los mensajes (1), (2) y (3) el usuario ejecuta una llamada `ALARM` mediante un mensaje al administrador de memoria, el cual envía una solicitud al reloj, de la cual el reloj acusa recibo. En los mensajes (4), (5) y (6), la tarea del reloj envía la alarma al administrador de memoria, éste llama a la tarea del sistema con el fin de que prepare la pila del proceso de usuario para la ejecución del manejador de señales (como en la Fig. 4-42(b)), y la tarea del sistema responde. El mensaje (7) es la llamada a `SIGRETURN` que ocurre cuando el manejador termina de ejecutarse. Como respuesta, el administrador de memoria envía el mensaje (8) a la tarea del sistema pidiéndole que complete el aseo, y la tarea del sistema responde con el mensaje (9). El mensaje (6) por sí solo no hace que se ejecute el manejador, pero la secuencia se mantendrá porque la tarea del sistema, al ser una tarea, podrá completar su trabajo gracias al algoritmo de planificación por prioridad que `MINK` emplea. El manejador forma parte del proceso de usuario y se ejecutará sólo después de que la tarea del sistema haya completado su trabajo.

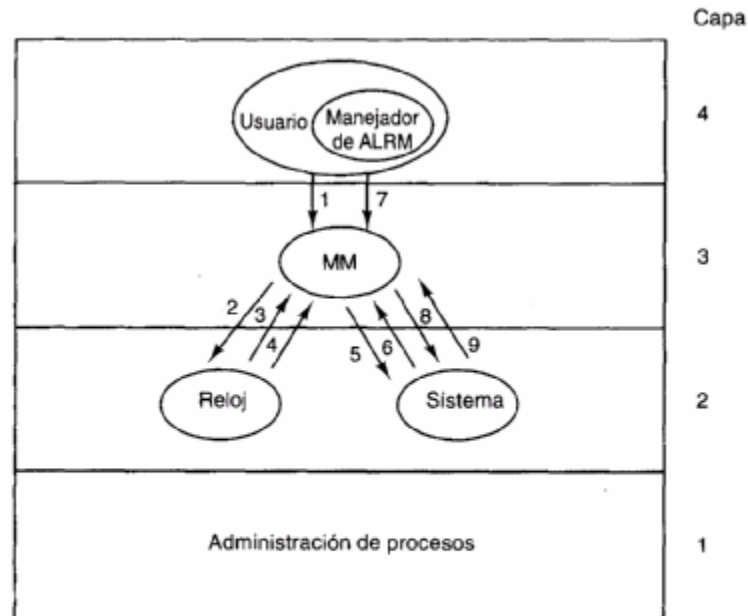


Figura 4-45. Mensajes para una alarma. Los más importantes son: (1) El usuario ejecuta ALARM. (4) Una vez transcurrido el tiempo establecido, llega la señal. (7) El manejador termina con una llamada a SIGRETURN. Consúltense los detalles en el texto.

Do_pause se ocupa de la llamada al sistema PAUSE (línea 18115). Lo único que se necesita es encender un bit y abstenerse de responder, manteniendo así al invocador bloqueado. Ni siquiera es necesario informar al kernel, ya que él sabe que el invocador está bloqueado.

La última llamada al sistema que se maneja en signal.c es REBOOT (línea 18128). Esta llamada, la emplean sólo programas especializados ejecutables por el superusuario, pero realiza una función importante: asegura que todos los procesos terminen de forma ordenada y que el sistema de archivos esté sincronizado antes de que se invoque la tarea del sistema en el kernel para concluir operaciones. La terminación de los procesos se efectúa utilizando check_sig para enviar un SIGKILL a todos los procesos excepto init. Es por esta razón que REBOOT se incluyó en este archivo.

Hemos mencionado de paso varias funciones de apoyo de signal.c. Ahora las examinaremos de forma más detallada. Por mucho, la más importante es sig_proc (línea 18168), que es la que realmente envía las señales. Primero se efectúa una serie de pruebas. Los intentos por enviar señales a procesos muertos (líneas 18190 a 18192) o zombis (líneas 18194 a 18196) son problemas graves que causan un pánico del sistema. Un proceso que se está rastreando se detiene cuando recibe una señal (líneas 18198a 18202). Si hay que ignorar la señal, el trabajo de sig_prwn termina en la línea 18204. Ésta es la acción predeterminada para algunas señales, por ejemplo: aquellas que POSIX exige pero que MINIX no apoya. Si la señal se bloquea, la única acción que se debe emprender es encender un bit en el mapa de bits mp_sigpending del proceso en cuestión. La

prueba clave (línea 18213) consiste en distinguir entre los procesos que están habilitados para atrapar señales y los que no lo están. A estas alturas ya se han eliminado todas las demás consideraciones especiales y si un proceso no puede atrapar la señal será terminado.

Las señales que son elegibles para atraparse se procesan en las líneas 18214 a 18249. Se construye un mensaje que se enviará al kernel; partes de dicho mensaje son copias de información que está en la parte de la tabla de procesos que corresponde al administrador de memoria. Si el proceso al que se enviará la señal se había suspendido previamente con SIGSUSPEND, se incluye en el mensaje la máscara de señales que se guardó en el momento de la suspensión; si no, se incluye la máscara de señal actual (líneas 18213 a 18217). Otros elementos que se incluyen en el mensaje son varias direcciones en el espacio del proceso al que se envía la señal: el manejador de señales, la dirección de la rutina de biblioteca sigreturn que se invocará cuando el manejador termine, y el apuntador a la pila actual.

A continuación, se asigna espacio en la pila del proceso. En la Fig. 4-46 se muestra la estructura que se coloca en la pila. La porción sigcontext se coloca en la pila a fin de conservarla para una restauración posterior, ya que la estructura correspondiente en la tabla de procesos misma se altera como preparación para la ejecución del manejador de señales. La parte sigframe proporciona una dirección de retomo para el manejador de señales y datos que SIGRETURN necesita para completar la restauración del estado del proceso cuando el manejador acaba. La dirección de retomo y el apuntador al marco no son utilizados realmente por ninguna parte de MINIX; están ahí para engañar a un depurador por si alguien intentara rastrear la ejecución de un manejador de señales.

La estructura que se colocará en la pila del proceso destinatario de la señal es más o menos grande. El código de las líneas 18225 y 18226 reserva espacio para ella, después de lo cual una llamada a adjust prueba si hay suficiente espacio en la pila del proceso. Si no hay suficiente espacio de pila, se termina el proceso mediante un salto al rótulo determinate empleando el pocas veces usado goto de C (líneas 18228 y 18229).

Hay un problema potencial con la llamada a adjust. Recuerde que cuando explicamos la implementación de BRK dijimos que adjust devuelve un error si la pila tiene menos de SAFETY_BYTES de toparse con el segmento de datos. Se proporciona el margen de error adicional porque el software sólo puede verificar ocasionalmente la validez de la pila. Dicho margen de error probablemente sea excesivo en el presente caso, ya que se sabe exactamente cuánto espacio se necesita en la pila para la señal, y sólo se requiere espacio adicional para el manejador de señal, que supuestamente es una función relativamente sencilla. Es posible que algunos procesos se terminen innecesariamente porque falla la llamada a adjust. Sin duda, esto es mejor que experimentar fallas misteriosas de los programas en otras ocasiones, pero podría ser posible afinar un poco más estas pruebas.

Si hay suficiente espacio en la pila, se verifican otras dos banderas. La bandera SA_NODEFER indica si el proceso al que se envía la señal debe bloquear las señales subsecuentes del mismo tipo mientras maneja una señal. La bandera SA_RESETHAND indica si el manejador de señales debe restablecerse al recibir esta señal. (Esto ofrece una fiel emulación de la antigua llamada signal. Aunque esta "capacidad" suele considerarse como un defecto de la antigua llamada, el apoyo de capacidades anteriores también implica apoyar sus defectos.) A continuación se notifica al kernel, empleando la rutina de biblioteca sys_sendsig (línea 18242). Por último, se pone en cero el bit que

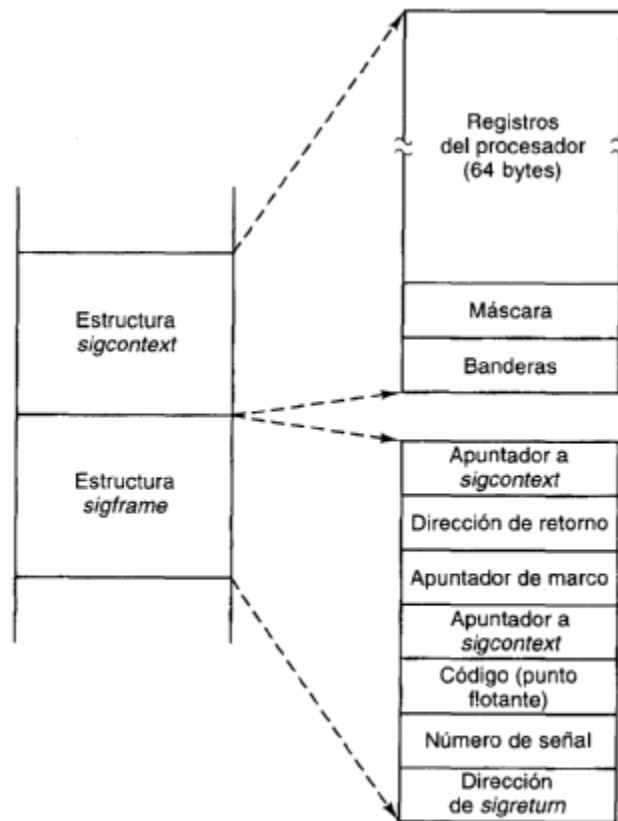


Figura 4-46. Las estructuras *sigcontext* y *sigframe* que se incorporan a la pila como preparación para la ejecución de un manejador de señales. Los registros del procesador son una copia del marco de pila empleado durante una conmutación de contexto.

indica que hay una señal pendiente y se invoca un `pause` para terminar cualquier llamada al sistema que el proceso pueda estar esperando. La próxima vez que se ejecute el proceso al que se envió la señal, se ejecutará el manejador de señales.

Examinemos ahora el código de terminación, marcado por el rótulo `determine` (línea 18250). El rótulo y un `goto` son la forma más fácil de manejar la posible falla de la llamada a `adjust`. Aquí se procesan las señales que por una razón u otra no se pueden o no se deben atrapar. La acción podría incluir un vaciado de núcleo, si resulta apropiado para la señal, y siempre concluye con la terminación del proceso como si éste hubiera salido, a través de una llamada a `mm_exit` (línea 18258).

`Check_sig` (línea 18265) es donde el administrador de memoria verifica si es posible enviar una señal. La llamada

```
kill(0, sig);
```

hace que se envíe la señal indicada a todos los procesos del grupo del usuario (es decir, todos los

procesos iniciados desde la misma terminal). Las señales que se originan en el kernel y REBOOT también podrían afectar múltiples procesos. Por esta razón, `check_sig` ejecuta un ciclo en las líneas 18288 a 18318 para examinar la tabla de procesos en busca de todos los procesos a los que se debe enviar una señal. El ciclo contiene un gran número de pruebas. Sólo si se satisfacen todas las condiciones se envía la señal, invocando `sig_proc` en la línea 18315.

`Check_pending` (línea 18330) es otra función que se invoca varias veces en el código que acabamos de explicar. Ésta ejecuta un ciclo para examinar todos los bits del mapa de bits `mp_sigpending` correspondiente al proceso al que hacen referencia `do_sigmask`, `do_sigreturn` o `do_sigsuspend`, con objeto de ver si alguna señal bloqueada ha quedado desbloqueada. `Check_pending` invoca `sig_proc` para enviar la primera señal pendiente desbloqueada que encuentra. Puesto que todos los manejadores de señales tarde o temprano causan la ejecución de `do_sigreturn`, esto basta para finalmente entregar todas las señales no enmascaradas pendientes.

El procedimiento `unpause` (línea 18359) tiene que ver con las señales que se envían a procesos suspendidos en llamadas `PAUSE`, `WAIT`, `READ`, `WRITE` o `SIGSUSPEND`. `PAUSE`, `WAIT` y `SIGSUSPEND` pueden verificarse consultando la parte de la tabla de procesos que corresponde al administrador de memoria, pero si no se encuentra ninguna de éstas será necesario pedir al sistema de archivos que use su propia función `do_unpause` para verificar si hay una posible espera por `READ` o `WRITE`.

En todos los casos la acción es la misma: se envía una respuesta de error a la llamada que espera y se pone en cero el bit de bandera que corresponde a la causa de la espera para que el proceso pueda reanudar su ejecución y procesar la señal.

El último procedimiento de este archivo es `dump_core` (línea 18402), que escribe vaciados de núcleo en el disco. Un vaciado de núcleo consiste en una cabecera con información relativa al tamaño de los segmentos ocupados por un proceso, una copia de toda la información de estado del proceso, obtenida copiando la información de tabla de procesos del kernel correspondiente a ese proceso, y la imagen de memoria de cada uno de los segmentos. Un depurador puede interpretar esta información y ayudar al programador a determinar qué error hubo durante la ejecución del proceso. El código para escribir el archivo es sencillo. El problema potencial que mencionamos en la sección anterior vuelve a asomar la cabeza, pero en una forma un tanto diferente. Para asegurar que el segmento de pila que se incluirá en el vaciado de núcleo está actualizado, se invoca `adjust` en la línea 18428. Esta llamada podría fallar a causa del margen de seguridad incorporado en ella. `Dump_core` no comprueba que la llamada haya tenido éxito, así que el vaciado de núcleo se escribirá de todos modos, pero cabe la posibilidad que dentro del archivo la información relativa a la pila sea incorrecta.

4.8.7 Implementación de las otras llamadas al sistema

El archivo `getset.c` contiene un procedimiento, `do_getset` (línea 18515), que ejecuta las siete llamadas restantes del administrador de memoria. Éstas se muestran en la Fig. 4-47. Todas son tan sencillas que no merecen un procedimiento completo para cada una. Las llamadas `GETUID` y `GETGID` welven el `uid` o `gid` tanto real como efectivo.

El establecimiento del `uid` o el `gid` es un poco más complejo que su mera lectura. Se debe comprobar si el invocador está autorizado para establecer el `uid` o `gid`. Si el invocador pasa la

Llamada al sistema	Descripción
GETUID	Devuelve uid real y efectivo
GETGID	Devuelve gid real y efectivo
GETPID	Devuelve pids del proceso y su padre
SETUID	Establece el uid real y efectivo del invocador
SETGID	Establece el gid real y efectivo del invocador
SETSID	Crea nueva sesión, devolver pid
GETPGRP	Devuelve identificador del grupo de procesos

Figura 4-47. Las llamadas al sistema apoyadas en *mm/getset.c*.

prueba, se debe informar al sistema de archivos del nuevo uid o gid, ya que la protección de los archivos depende de ello. La llamada SETSID crea una nueva sesión, y un proceso que ya es líder de un grupo de procesos no está autorizado a llevar esto a cabo. La prueba de la línea 18561 verifica esto. El sistema de archivos completa el trabajo de convertir un proceso en líder de sesión sin terminal controladora.

El apoyo mínimo para la depuración, mediante la llamada al sistema PTRACE, está contenido en el archivo *trace.c*. Hay once comandos que pueden proporcionarse como parámetro a la llamada al sistema PTRACE. Éstos se muestran en la Fig. 4-48. En el administrador de memoria, *doJrace* procesa cuatro de ellos: *enable* (habilitar), *exit* (salir), *resume* (reanudar), *step* (pasos). Las solicitudes de habilitar el rastreo o salir de él se atienden aquí. Todos los demás comandos se pasan a la tarea del sistema, que tiene acceso a la parte del kernel de la tabla de procesos. Esto se hace con la llamada a la función de biblioteca *sys_trace* en la línea 18669. Al final de *trace.c* se incluyen dos funciones de apoyo para el rastreo. *Stop_proc* sirve para detener un proceso rastreado cuando se le envía una señal, y *findproc* apoya *do_trace* buscando en la tabla de procesos el proceso que se va a rastrear.

4.8.8 Utilerías del administrador de memoria

El resto de los archivos contienen rutinas de utilería y tablas. El archivo *alloc.c* es donde el sistema toma nota de qué partes de la memoria están en uso y cuáles están libres. Hay cuatro puntos de entrada:

1. *alloc_mem* - solicitar un bloque de memoria de cierto tamaño.
2. *free_mem* - devolver memoria que ya no se necesita.
3. *max_hole* - calcular el tamaño del agujero más grande disponible.
4. *mem_init* - inicializar la lista libre cuando el administrador de memoria comienza a ejecutarse.

Como ya dijimos, *alloc_mem* (línea 18840) utiliza primer ajuste con una lista de agujeros ordenados según su dirección en la memoria. Si *alloc_mem* encuentra un fragmento demasía

Comando	Descripción
T_STOP	Detiene el proceso
T_OK	Habilita el rastreo por el padre para este proceso
T_GETINS	Devuelve valor del espacio de texto (instrucciones)
T_GETDATA	Devuelve valor del espacio de datos
T_GETUSER	Devuelve valor de la tabla de procesos de usuario
T_SETINS	Establece valor en espacio de instrucciones
T_SETDATA	Establece valor en espacio de datos
T_SETUSER	Establece valor en tabla de procesos de usuario
T_RESUME	Reanuda ejecución
T_EXIT	Sale
T_STEP	Establece bit de rastreo

Figura 4-48. Comandos de depuración apoyados por *mm/trace.c*.

grande, toma lo que necesita y deja el resto en la lista libre, pero reduciendo su tamaño en la cantidad tomada. Si se requiere un agujero completo, se invoca `del_slot` (línea 18926) para quitar la entrada de la lista libre.

El trabajo `defree_mem` consiste en verificar si una porción de memoria recién liberada se puede fusionar con agujeros adyacentes. Si se puede, se invoca `merge` (línea 18949) para unir los agujeros y actualizar las listas.

`MaxJiole` (línea 18985) examina la lista de agujeros y devuelve el elemento más grande que encuentra. `Mem_init` (línea 19005) construye la lista libre inicial, que incluye toda la memoria disponible.

El siguiente archivo es `utility.c`, que contiene unos cuantos procedimientos diversos empleados en distintos puntos del administrador de memoria. El procedimiento `allowed` (línea 19120) verifica si se permite un acceso dado a un archivo. Por ejemplo, `do_exec` necesita saber si un archivo es ejecutable.

El procedimiento `no_sys` (línea 19161) nunca debe invocarse. Se incluye sólo por si un usuario llega a invocar el administrador de memoria con un número de llamada al sistema que no es válido o que no está entre los que el administrador de memoria maneja.

`Panic` (línea 19172) se invoca sólo cuando el administrador de memoria detecta un error del cual no se puede recuperar. Este procedimiento informa del error a la tarea del sistema, la cual entonces detiene MINIX de inmediato. `Panic` sólo se invoca en situaciones graves.

La última función de `utility.c` es `tell_fs`, que construye un mensaje y lo envía al sistema de I/O cuando se debe informar a éste de sucesos manejados por el administrador de memoria. Los dos procedimientos del archivo `putk.c` también son utilerías, aunque de índole muy distinta de los anteriores. De vez en cuando se insertan llamadas a `printf` en el administrador de

memoria, principalmente para fines de depuración. Además, `panic` invoca `printf`. Como se mencionó antes, el nombre `printf` es en realidad una macro definida como `printk`, así que las llamadas a `printfno` usan el procedimiento de biblioteca de E/S estándar que envía mensajes al sistema de archivos. `Printk` invoca `putk` para comunicarse directamente con la tarea de la terminal, algo que está prohibido para los usuarios ordinarios. Vimos una rutina con el mismo nombre en el código del kernel.

4.9 RESUMEN

En este capítulo examinamos la administración de memoria, tanto en general como en MINIX. Vimos que los sistemas más sencillos no realizan intercambios ni paginación. Una vez que un programa se carga en la memoria, permanece ahí hasta terminar. Algunos sistemas operativos sólo permiten un proceso a la vez en la memoria, mientras que otros apoyan la multiprogramación.

El siguiente paso hacia arriba es el intercambio. Cuando se usa intercambio, el sistema puede manejar más procesos que los que cabrían en la memoria. Los procesos para los cuales no hay espacio se intercambian al disco. Puede seguirse la pista al espacio libre en la memoria y en el disco mediante un mapa de bits o una lista de agujeros.

Las computadoras más avanzadas suelen tener alguna forma de memoria virtual. En la forma más simple, el espacio de direcciones de cada proceso se divide en bloques de tamaño uniforme llamados páginas, que se pueden colocar en cualquier marco de página que esté disponible en la memoria. Hay muchos algoritmos de reemplazo de páginas; dos de los más conocidos son el de segunda oportunidad y el de maduración. No basta escoger un algoritmo para lograr que los sistemas de paginación funcionen bien; deben cuidarse aspectos tales como la determinación del conjunto de trabajo, la política de asignación de memoria y el tamaño de las páginas.

La segmentación ayuda a manejar estructuras de datos que cambian de tamaño durante la ejecución, y simplifica la vinculación y la compartición. Además, cuando se usan segmentos más fáciles proporcionar distintos niveles de protección a los diferentes segmentos. En algunos casos se combina la segmentación con paginación para crear una memoria virtual bidimensional. El sistema MULTICS y el Pentium de Intel apoyan la segmentación y la paginación.

La administración de memoria en MINIX es sencilla. Se asigna memoria cuando un proceso ejecuta una llamada al sistema `FORK` o `EXEC`. La memoria así asignada nunca se incrementa, se decrementa durante la vida del proceso. En los procesadores Intel, MINIX emplea dos modelos de memoria. Los programas pequeños pueden tener sus instrucciones y sus datos en el mismo segmento de memoria. Los programas más grandes usan espacio de instrucciones y de datos separado (I y D separado). Los procesos con espacio I y D separado pueden compartir la porción de texto de su memoria, de modo que sólo es necesario asignar memoria para los datos y la durante un `FORK`. Esto también podría ser cierto durante un `EXEC` si otro proceso ya está usando el texto que el nuevo programa necesita.

La mayor parte del trabajo del administrador de memoria no tiene que ver con mantener tanto de la memoria libre, lo cual hace empleando una lista de agujeros y el algoritmo de ajuste, sino con ejecutar las llamadas al sistema relacionadas con administración de memoria.

Varias llamadas al sistema reconocen señales estilo POSIX, y puesto que la acción predeterminada de la mayor parte de las señales es terminar el proceso al que se envió la señal, resulta apropiado manejarlas en el administrador de memoria, que inicia la terminación de todos los procesos. El administrador de memoria también maneja varias llamadas al sistema que no están relacionadas directamente con la memoria, principalmente porque es más pequeño que el sistema de archivos, así que resultó más cómodo colocarlas ahí.

PROBLEMAS

1. Un sistema de computadora tiene suficiente espacio para contener cuatro programas en su memoria principal. La mitad del tiempo, estos programas están ociosos esperando E/S. ¿Qué fracción del tiempo de CPU se desperdicia?
2. Considere un sistema de intercambio en el que la memoria tiene agujeros con los siguientes tamaños en orden según su posición en la memoria: 10K, 4K, 20K, 18K, 7K, 9K, 12K y 15K. ¿Cuál agujero se toma cuando hay solicitudes de segmento sucesivas de
 - (a) 12K
 - (b) 10K
 - (c) 9Ksi se usa primer ajuste? Repil;). usando mejor ajuste, peor ajuste y siguiente ajuste.
3. ¿Qué diferencia hay entre una dirección física y una dirección virtual?
4. Empleando la tabla de páginas de la Fig. 4-8, dé la dirección física que corresponde a cada una de las siguientes direcciones virtuales:
 - (a) 20
 - (b) 4100
 - (c) 8300
5. El procesador Intel 8086 no apoya la memoria virtual. No obstante, algunas compañías vendieron previamente sistemas que contenían una CPU 8086 no modificada y realizaban paginación. Utilice lo que sabe para sugerir cómo lo hicieron. (Sugerencia: piense en la ubicación lógica de la MMU.)
6. Si una instrucción tarda 1 microsegundo y una falla de página tarda n microsegundos adicionales, deduzca una fórmula para el tiempo de instrucción efectivo si ocurren fallas de página cada k instrucciones.
7. Una máquina tiene un espacio de direcciones de 32 bits y páginas de 8K. La tabla de páginas está totalmente en hardware, con una palabra de 32 bits por cada entrada. Cuando un proceso inicia, la tabla de páginas se copia en el hardware desde la memoria, a razón de una palabra cada 100 ns. Si cada proceso se ejecuta durante 100 ms (incluido el tiempo que toma cargar la tabla de páginas), ¿qué fracción del tiempo de CPU se dedica a cargar las tablas de páginas?
8. Una computadora con direcciones de 32 bits usa una tabla de páginas de dos niveles. Las direcciones virtuales se dividen en un campo de tabla de páginas de nivel superior de nueve bits y un campo de tabla de páginas de segundo nivel de once bits, además de una distancia. ¿Qué tamaño tienen las páginas y cuántas de ellas hay en el espacio de direcciones?

9. A continuación se lista un programa corto en lenguaje ensamblador para una computadora con página; de 512 bytes. El programa reside en la dirección 1020, y su apuntador a la pila está en 8192 (la pila crece hacia 0). Dé la cadena de referencia a página generada por este programa. Cada instrucción ocupa cuatro bytes (una palabra), y las referencias tanto a instrucciones como a datos cuentan en 1. cadena de referencia.

Cargar la palabra 6144 en el registro O

Meter el registro O en la pila

Invocar un procedimiento en 5120, agregando la dirección de retomo a la pila

Restar la constante inmediata 16 del apuntador a la pila

Comparar el parámetro real con la constante inmediata 4

Saltar si es igual a 5152

10. Suponga que una dirección virtual de 32 bits se divide en cuatro campos, a, b, c y d. Los primeros tres se utilizan para un sistema de tablas de páginas de tres niveles. El cuarto campo, d, es la distancia. ¿El número de páginas depende de los tamaños de los cuatro campos? Si no es así, ¿cuáles importan y cuáles no?

11. Una computadora cuyos procesos tienen 1024 páginas en sus espacios de direcciones mantiene sus tablas de páginas en la memoria. El gasto extra requerido para leer una palabra de la tabla de páginas es de 500 ns. Con objeto de reducir este gasto extra, la computadora tiene un TLB, que contiene 32 pares (página virtual, marco de página físico) y puede realizar una consulta en 100 ns. ¿Qué tasa de aciertos se necesita para reducir el gasto medio a 200 ns?

12. El TLB del VAX no contiene un bit R. ¿Por qué?

13. Una máquina tiene direcciones virtuales de 48 bits y direcciones físicas de 32 bits. Las páginas son de 8K. ¿Cuántas entradas debe tener la tabla de páginas?

14. Una computadora tiene cuatro marcos de página. A continuación se muestran el tiempo de carga, el tiempo de último acceso y los bits R y M para cada página (los tiempos están en tics del reloj):

Página	Cargada	Última ref.	R	M
O	126	279	O	O
1	230	260	1	O
2	120	272	1	1
3	160	280	1	1

(a) ¿Cuál página se reemplazará si se usa NRU?

(b) ¿Cuál página se reemplazará si se usa FIFO?

(c) ¿Cuál página se reemplazará si se usa LRU?

(d) ¿Cuál página se reemplazará si se usa segunda oportunidad?

15. Si se emplea reemplazo de páginas FIFO con cuatro marcos de página y ocho páginas, ¿cuántas fallas de página ocurrirán con la cadena de referencia 0172327103 si los cuatro marcos inicialmente están ; vacíos? Repita el problema suponiendo que se usa LRU.

16. Una computadora pequeña tiene cuatro marcos de página. En el primer tic del reloj, los bits R son 111 (la página O es O, el resto son 1). En tics del reloj subsecuentes, los valores son 1011,1010,1101,0010, 1010, 1100 y 0001. Si se emplea el algoritmo de maduración con un contador de ocho bits, indique los valores de los cuatro contadores después del último tic.

17. ¿Qué tiempo toma cargar un programa de 64K de un disco cuyo tiempo de búsqueda medio es de 30 ms, cuyo tiempo de rotación es de 20 ms y cuyas pistas contienen 32K

(a) si se usan páginas de 2K?

(b) si se usan páginas de 4K?

Las páginas están distribuidas aleatoriamente en el disco.

18. Una de las primeras máquinas de tiempo compartido, la PDP-1, tenía una memoria de 4K palabras de 18 bits, y mantenía en memoria sólo un proceso a la vez. Cuando el planificador decidía ejecutar otro proceso, el proceso que estaba en la memoria se escribía en un tambor de paginación, con 4K palabras de 18 bits alrededor de la circunferencia del tambor. El tambor podía comenzar a escribir (o leer) en cualquier palabra, no forzosamente en la palabra 0. ¿Por qué supone usted que se escogió este tambor?

19. Una computadora proporciona a cada proceso 65 536 bytes de espacio de direcciones dividido en páginas de 4096 bytes. Cierta programa tiene un tamaño de texto de 32 768 bytes, un tamaño de datos de 16 386 bytes y un tamaño de pila de 15 870 bytes. ¿Cabría este programa en el espacio de direcciones? Si el tamaño de página fuera de 512 bytes, ¿cabría el programa? Recuerde que una página no puede contener partes de dos segmentos distintos.

20. Se ha observado que el número de instrucciones ejecutadas entre dos fallas de página consecutivas es directamente proporcional al número de marcos de página asignados a un programa. Si se duplica la memoria disponible, el intervalo medio entre fallas de página también se duplica. Suponga que una instrucción normal tarda 1 microsegundo, pero si ocurre una falla de página tarda 2001 microsegundos (es decir, se requiere 2 ms para manejar la falla). Si un programa tarda 60 s en ejecutarse, y durante este tiempo experimenta 15 000 fallas de página, ¿cuánto tardaría en ejecutarse si hubiera el doble de memoria disponible?

21. Un grupo de diseñadores de sistemas operativos para la Compañía de Computadoras Económicas está buscando formas de reducir la cantidad de almacenamiento de respaldo que requiere su nuevo sistema operativo. El gurú principal acaba de sugerir que no vale la pena molestarse por guardar el texto de programa en el área de intercambio; en vez de ello, se puede colocar directamente en la memoria por paginación desde el archivo binario cada vez que se necesite. ¿Tiene algún problema tal estrategia?

22. Explique la diferencia entre fragmentación interna y externa. ¿Cuál de ellas ocurre en los sistemas con paginación? ¿Cuál ocurre en los sistemas que emplean segmentación pura?

23. Cuando se usa tanto segmentación como paginación, como en MULTICS, primero hay que consultar el descriptor de segmento y luego el descriptor de página. ¿El TLB funciona también de esta manera, con dos niveles de consulta?

24. ¿Por qué el esquema de administración de memoria de MINIX obliga a tener un programa como hmemr!

25. Modifique MINIX de modo que libere la memoria ocupada por un zombi tan pronto como el proceso pase al estado zombi, en lugar de esperar hasta que el padre ejecute WAIT por él.

26. En la implementación actual de MINIX, cuando se efectúa una llamada al sistema EXEC, el administrador de memoria verifica si está disponible un agujero con el tamaño suficiente para contener la nueva imagen de memoria. Si no lo hay, la llamada se rechaza. Un mejor algoritmo sería determinar si habrá un agujero con el tamaño requerido después de que se libere la imagen de memoria actual. Implemente este algoritmo.

27. Al realizar una llamada al sistema EXEC, MINIX emplea un truco para hacer que el sistema de archivos lea segmentos enteros en una sola operación. Invente e implemente un truco similar para permitir que los vaciados de núcleo se escriban de la misma forma.

28. Modifique MINIX de modo que realice intercambio.
29. En la sección 4.7.5 se señaló que, al efectuarse una llamada EXEC, el hecho de que la prueba para encontrar un agujero apropiado se realice antes de liberar la memoria del proceso actual da lugar a una implementación subóptima. Reprograme este algoritmo para que funcione mejor.
30. En la sección 4.8.4 se apuntó que sería mejor buscar agujeros para los segmentos de texto y de datos por separado. Implemente esta mejora.
31. Rediseñe adjust de modo que evite el problema de la terminación innecesaria de procesos destinatarios de señales a causa de una prueba demasiado estricta para determinar si hay suficiente espacio de pila.

5

SISTEMAS DE ARCHIVOS

Todas las aplicaciones de computadora necesitan almacenar y recuperar información. Mientras un proceso se está ejecutando, puede almacenar una cantidad de información limitada dentro de su propio espacio de direcciones. Sin embargo, la capacidad de almacenamiento está restringida al tamaño del espacio de direcciones virtual. En el caso de algunas aplicaciones, este tamaño es adecuado, pero en el de otras, como las de reservaciones de líneas aéreas, las aplicaciones bancadas o las bases de datos corporativas, dicho tamaño resulta excesivamente pequeño.

Un segundo problema de mantener la información dentro del espacio de direcciones de un proceso es que cuando el proceso termina la información se pierde. En muchas aplicaciones (como las bases de datos), la información debe retenerse durante semanas, meses, o incluso eternamente. No es aceptable que la información desaparezca cuando el proceso que la está usando termina. Además, la información no debe perderse cuando una caída de la computadora termina el proceso.

Un tercer problema es que en muchos casos es necesario que múltiples procesos accedan a (partes de) la información al mismo tiempo. Si tenemos un directorio telefónico en línea almacenado dentro del espacio de direcciones de un solo proceso, únicamente ese proceso tendrá acceso a él. La forma de resolver este problema es hacer que la información misma sea independiente de cualquier proceso individual.

Por tanto, tenemos tres requisitos esenciales para el almacenamiento de información a largo plazo:

1. Debe ser posible almacenar una gran cantidad de información.
2. La información debe sobrevivir a la terminación del proceso que la usa.
3. Múltiples procesos deben poder acceder a la información de forma concurrente.

La solución usual a todas estas exigencias es almacenar la información en discos y otros medios externos en unidades llamadas archivos. Así, los procesos pueden leerlos y escribir archivos

nuevos si es necesario. La información almacenada en archivos debe ser persistente, es decir, no ser afectada por la creación y terminación de procesos. Un archivo sólo debe desaparecer cuando su propietario lo elimina explícitamente.

Los archivos son administrados por el sistema operativo. La forma como se estructuran, nombran, acceden, usan, protegen e implementan son temas importantes en el diseño de sistemas operativos. Globalmente, la parte del sistema operativo que se ocupa de los archivos se denomina **sistema de archivos** y es el tema de este capítulo.

Desde el punto de vista del usuario, el aspecto más importante de un sistema de archivos es la cara que presenta ante él, es decir, qué constituye un archivo, cómo se nombran y protegen los archivos, qué operaciones pueden efectuarse con los archivos, etc. Los detalles de si se usan listas enlazadas o mapas de bits para mantenerse al tanto del espacio de almacenamiento disponible y de cuántos sectores hay en un bloque lógico tienen menos interés, aunque son de gran relevancia para los diseñadores del sistema de archivos. Por esta razón, hemos estructurado el capítulo en varias secciones. Las dos primeras se ocupan de la interfaz entre el usuario y los archivos y directorios, respectivamente. Luego viene una explicación detallada de cómo se implementa el sistema de archivos. Después, examinaremos la seguridad y los mecanismos de protección de los sistemas de archivos. Por último estudiaremos el sistema de archivos de MINIX.

5.1 ARCHIVOS

En esta sección consideraremos los archivos desde el punto de vista del usuario, es decir, cómo se usan y qué propiedades tienen.

5.1.1 Nombres de archivos

Los archivos son un mecanismo de abstracción; proporcionan una forma de almacenar información en el disco y leerla después. Esto debe hacerse de tal manera que el usuario no tenga que ocuparse de los detalles de cómo y dónde se almacena la información, ni de cómo funcionan realmente los discos.

Tal vez la característica más importante de cualquier mecanismo de abstracción sea la forma como se nombran los objetos que se manejan, así que iniciaremos nuestro estudio de los sistemas de archivos con el tema de los nombres de archivos. Cuando un proceso crea un archivo, le asigna un nombre. Cuando el proceso termina, el archivo sigue existiendo y otros procesos pueden acceder a él utilizando su nombre.

Las reglas exactas para nombrar archivos varían un tanto de un sistema a otro, pero todos los sistemas operativos permiten cadenas de uno a ocho caracteres como nombres de archivo válidos. Así, andrea, braulio y caria son posibles nombres de archivo. En muchos casos se permiten: también dígitos y caracteres especiales, haciendo válidos también nombres como 2, urgente! y,| Fig.2-14. Muchos sistemas de archivos reconocen nombres de hasta 255 caracteres.

Algunos sistemas de archivos distinguen entre las letras mayúsculas y minúsculas, y otros no. UNIX pertenece a la primera categoría; MS-DOS cae en la segunda. Por tanto, un sistema UNIX

puede tener todos los siguientes como archivos distintos: barbara. Barbara, BARBARA, BARbara y BarBaRa. En MS-DOS todos designan el mismo archivo.

Muchos sistemas operativos reconocen nombres de archivo de dos partes, las cuales se separan con un punto, como en prog.c. La parte que sigue al punto se denomina extensión de archivo y, por lo regular, indica algo acerca del archivo. En MS-DOS, por ejemplo, los nombres de archivo tienen de 1 a 8 caracteres, más una extensión opcional de 1 a 3 caracteres. En UNIX, el tamaño de la extensión, si la hay, es decisión del usuario, y un archivo incluso puede tener dos o más extensiones, como enprog.c.Z, donde .Z se emplea comúnmente para indicar que el archivo (prog.c) se compiló utilizando el algoritmo de compresión Ziv.Lempel. En la Fig. 5-1 se presentan algunas de las extensiones de archivo más comunes y su significado.

Extensión	Significado
archivo.bak	Archivo de respaldo
archivo.c	Programa fuente en C
archivo.f77	Programa en Fortran 77
archivo.gif	Formato de intercambio de gráficos de Compuserve
archivo.hlp	Archivo de ayuda
archivo.html	Documento en Lenguaje de Marcado de HiperTexto de la World Wide Web
archivo.mpg	Cine codificado con la norma MPEG
archivo.o	Archivo objeto (salida del compilador, antes de enlazar)
archivo.ps	Archivo PostScript
archivo.tex	Entrada para el programa de formateo TEX
archivo.txt	Archivo de texto general
archivo.zip	Archivo comprimido

Figura 5-1. Algunas extensiones de archivo típicas.

En algunos casos, las extensiones de archivo no son más que convenciones y su uso no es obligatorio. Un archivo llamado archivo.txt con toda probabilidad es algún tipo de archivo de texto, pero ese nombre sirve más como recordatorio para el propietario que para comunicar alguna información específica a la computadora. Por otro lado, un compilador de C podría exigir ; que los archivos que va a compilar terminen con .c, y podría negarse a compilarlos si no es así.

Las convenciones como ésta tienen especial utilidad cuando el mismo programa puede manejar varios tipos de archivo distintos. El compilador de C, por ejemplo, puede recibir una lista de varios archivos para compilar y enlazar, algunos de ellos archivos en C y otros archivos en lenguaje ensamblador. En este caso la extensión se vuelve esencial para que el compilador distinga jentre los archivos en C, los archivos en lenguaje ensamblador y los archivos de otro tipo.

5.1.2 Estructura de archivos

Los archivos pueden estructurarse de varias maneras. Tres posibilidades comunes se representan en la Fig. 5-2. El archivo de la Fig. 5-2(a) es una secuencia no estructurada de bytes. En efecto, el sistema operativo no sabe (ni le importa) qué contiene el archivo; lo único que ve es bytes. Cualquier significado que tenga deberán imponerlo los programas en el nivel de usuario. Tanto UNIX como MS-DOS adoptan este enfoque. Como acotación, WINDOWS 95 usa básicamente el sistema de archivos de MS-DOS, agregándole un poco de azúcar sintáctica (p. ej., nombres de archivo largos), así que casi todo lo que digamos en este capítulo acerca de MS-DOS también aplica a WINDOWS 95. En cambio, WINDOWS NT es totalmente distinto.

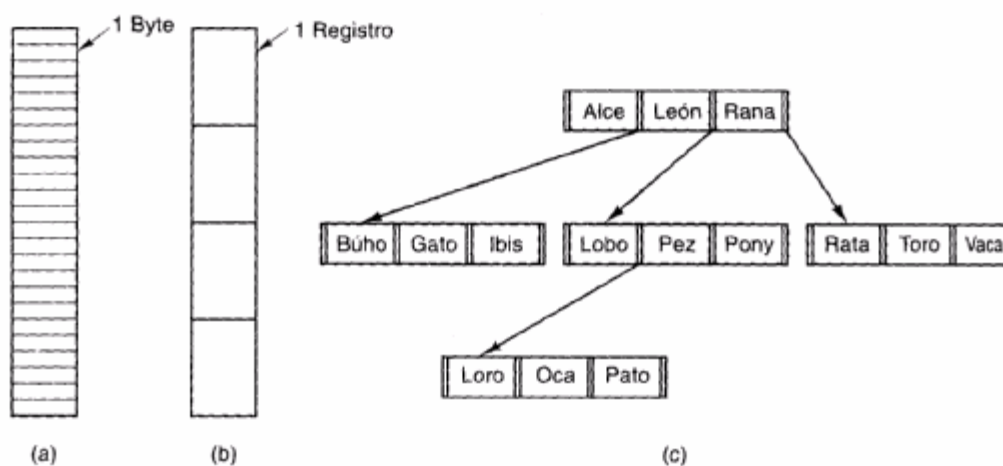


Figura 5-2. Tres clases de archivos. (a) Secuencia de bytes. (b) Secuencia de registros. (c) Árbol.

Hacer que el sistema operativo vea los archivos únicamente como secuencias de bytes ofrece el máximo de flexibilidad. Los programas de usuario pueden poner cualquier cosa que deseen los archivos y darles cualquier nombre que crean conveniente. El sistema operativo no ayuda, pero tampoco estorba. Para los usuarios que desean hacer cosas fuera de lo común, esto último es muy importante.

El primer paso hacia arriba en cuestión de estructura se muestra en la Fig. 5-2(b). En este modelo, un archivo es una secuencia de registros de longitud fija, cada uno con cierta estructura interna. La idea de que un archivo es una secuencia de registros se apoya en el concepto de que la operación de lectura devuelve un registro y que la operación de escritura sobrescribe o anexa un registro. En el pasado, cuando la tarjeta perforada de 80 columnas era la reina y señora, muchos sistemas operativos basaban sus sistemas de archivos en archivos compuestos por registros de 80 caracteres, que, en efecto, eran imágenes de tarjetas. Estos sistemas también daban soporte a archivos con registros de 132 caracteres, pensados para la impresora de líneas (que en esos días era una impresora grande de cadenas con 132 columnas). Los programas leían las entradas en unidades de 80 caracteres y las escribían en unidades de 132 caracteres aunque, desde luego, los últimos 52 podían ser espacios.

Un sistema (antiguo) que consideraba los archivos como secuencias de registros de longitud fija era CP/M. Éste utilizaba registros de 128 caracteres. Hoy día, la idea de un archivo como una secuencia de registros de longitud fija es prácticamente cosa del pasado, aunque en otra época fue la norma.

El tercer tipo de estructura de archivo se muestra en la Fig. 5-2(c). En esta organización, un archivo consiste en un árbol de registros, no necesariamente todos de la misma longitud, cada uno de los cuales contiene un campo de **llave** en una posición fija dentro del registro. El árbol está ordenado según el campo de llave, a fin de poder buscar rápidamente una llave en particular.

La operación básica aquí no es obtener el "siguiente" registro, aunque eso también es posible, sino obtener el registro que tiene una llave dada. En el caso del archivo de zoológico de la Fig. 5-2(c), podríamos pedirle al sistema que obtenga el registro cuya llave es pony, por ejemplo, sin tener que preocuparnos por su posición exacta en el archivo. Además, es posible agregar nuevos registros al archivo dejando que el sistema operativo, y no el usuario, decida dónde deben colocarse. Este tipo de archivo obviamente es muy distinto de los flujos de bytes no estructurados que se emplean en UNDC y MS-DOS, pero se utiliza ampliamente en las macrocomputadoras que todavía se usan en algunas aplicaciones comerciales de procesamiento de datos.

5.1.3 Tipos de archivos

Muchos sistemas operativos reconocen varios tipos de archivos. UNIX y MS-DOS, por ejemplo, tienen archivos normales y directorios. UNIX también tiene archivos especiales por caracteres y por bloques. Los **archivos regulares** son los que contienen información del usuario. Todos los archivos de la Fig. 5-2 son archivos normales. Los **directorios** son archivos de sistema que sirven para mantener la estructura del sistema de archivos. Estudiaremos los directorios más adelante. Los archivos especiales por caracteres están relacionados con entrada/salida y sirven para modelar dispositivos E/S en serie como las terminales, impresoras y redes. **Los archivos especiales por bloques** sirven para modelar discos. En este capítulo nos interesarán primordialmente los archivos normales.

Los archivos normales generalmente son archivos ASCII o bien archivos binarios. Los archivos ASCII consisten en líneas de texto. En algunos sistemas cada línea termina con un carácter de retorno de carro; en otros se emplea el carácter de salto de línea. Ocasionalmente se requieren ambos. Las líneas no tienen que tener todas la misma longitud.

La gran ventaja de los archivos ASCII es que pueden exhibirse e imprimirse tal como están, y se pueden editar con un editor de textos normal. Además, si una gran cantidad de programas usan archivos ASCII como entradas y salidas, es fácil conectar la salida de un programa a la entrada del otro, como en los conductos de shell. (La interconexión entre procesos no es más fácil, pero ciertamente lo es la interpretación de la información si se emplea una convención estándar, como ASCII, para expresarla.)

Otros archivos son binarios, lo que simplemente significa que no son archivos ASCII. Si listamos estos archivos en una impresora, obtendremos un listado incomprensible lleno de lo que parece ser basura. Por lo regular, estos archivos tienen alguna estructura interna.

Por ejemplo, en la Fig. 5-3(a) vemos un archivo binario ejecutable sencillo tomado de una de las primeras versiones de UNIX. Aunque técnicamente el archivo no es sino una secuencia de bytes, el sistema operativo sólo ejecuta un archivo si éste tiene el formato correcto. El ejemplo tiene

cinco secciones: encabezado, texto, datos, bits de reubicación y tabla de símbolos. El encabezado comienza con lo que se conoce como un **número mágico**, que identifica el archivo como ejecutable (a fin de evitar la ejecución accidental de un archivo que no tiene este formato). Luego vienen enteros de 16 bits que indican los tamaños de los distintos componentes del archivo, la dirección en la que se inicia la ejecución y algunos bits de bandera. Después del encabezado vienen el texto y los datos del programa mismo. Éstos se cargan en la memoria y se reubican empleando los bits de reubicación. La tabla de símbolos sirve para la depuración.

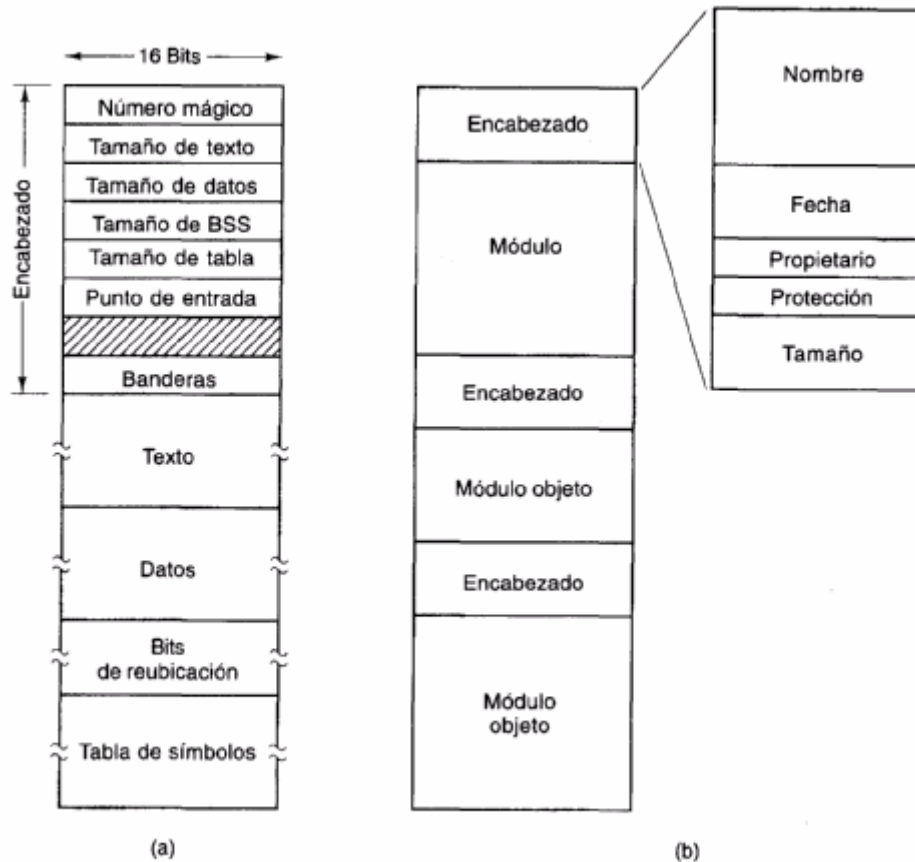


Figura 5-3. (a) Un archivo ejecutable. (b) Un archivo.

Nuestro segundo ejemplo de archivo binario es un archivo de archivado, también de UNKs| Este archivo consiste en una colección de procedimientos de biblioteca (módulos) compilado»;! pero no enlazados. Cada módulo va precedido por un encabezado que indica su nombre, fecha de creación, propietario, código de protección y tamaño. Al igual que en el archivo ejecutable, los encabezados de módulo están llenos de números binarios; si los copiáramos en una impresora obtendríamos un listado ininteligible.

Todos los sistemas operativos deben reconocer un tipo de archivo, su propio archivo ejecutable, pero algunos reconocen más. El antiguo sistema TOPS-20 llegó al extremo de examinará

tiempo de creación de todo archivo por ejecutar; luego localizaba el archivo fuente y veía si éste se había modificado después de crearse el binario. Si tal era el caso, el sistema recompilaba automáticamente el archivo fuente. En términos de UNIX, se había incorporado el programa make en el shell. Las extensiones de archivo eran obligatorias para que el sistema operativo pudiera saber cuál programa binario se derivaba de cuál fuente.

Algo por el estilo es lo que hace WINDOWS cuando un usuario hace doble clic sobre el nombre de un archivo: pone en marcha un programa apropiado con el nombre de archivo como parámetro. El sistema operativo determina cuál programa debe ejecutar basándose en la extensión del archivo.

Tener archivos con tipificación estricta como éstos causa problemas siempre que el usuario hace algo que los diseñadores del sistema no esperaban. Consideremos, por ejemplo, un sistema en el que los archivos de salida de los programas tienen el tipo dat (archivos de datos). Si un usuario escribe un formateador de programas que lee un archivo .pas, lo transforma (p. ej., convirtiéndolo a una organización con sangrías estándar) y luego envía el archivo transformado a la salida, el archivo de salida será de tipo .dat. Si el usuario trata de ofrecer este archivo al compilador de Pascal para que lo compile, el sistema no lo permitirá porque el archivo no tiene la extensión correcta. El sistema rechazará los intentos por copiar archivo.dat en archivo.pas por considerarlos no válidos (tratando de proteger al usuario contra errores).

Si bien este tipo de "amabilidad con el usuario" podría ayudar a los novatos, causa exasperación en los usuarios experimentados porque tienen que dedicar mucho trabajo a sortear la idea que se tiene del sistema operativo respecto a lo que es razonable y lo que no lo es.

5.1.4 Acceso a archivos

Los primeros sistemas operativos sólo ofrecían un tipo de acceso a los archivos: **acceso secuencial**. En estos sistemas, un proceso podía leer todos los bytes o registros de un archivo en orden, comenzando por el principio, pero no podía saltar de un lado a otro y leerlos en desorden. Los archivos secuenciales pueden "rebobinarse", así que pueden leerse tantas veces como sea necesario. Los archivos secuenciales son apropiados cuando el medio de almacenamiento es cinta magnética, no disco.

Cuando comenzaron a usarse los discos para almacenar archivos, se hizo posible leer los bytes o registros de un archivo en desorden, o acceder a los registros por llave, no por posición. Los archivos cuyos bytes o registros se pueden leer en cualquier orden se denominan **archivos de acceso aleatorio**.

Los archivos de acceso directo son esenciales para muchas aplicaciones, por ejemplo, los sistemas de bases de datos. Si un cliente de una línea aérea llama por teléfono y quiere reservar un asiento en un vuelo dado, el programa de reservación debe poder acceder al registro de ese vuelo sin tener que leer primero los registros de varios miles de otros vuelos.

Se emplean dos métodos para especificar el punto donde debe iniciarse la lectura. En el primero, cada operación READ indica la posición dentro del archivo en la que se debe comenzar a leer. En el segundo, se cuenta con una operación especial, SEEK, para establecer la posición actual. Después de un SEEK, el archivo se puede leer secuencialmente a partir de la posición que ahora es la actual.

En algunos sistemas operativos de macrocomputadoras antiguas, los archivos se clasifican como secuenciales o de acceso directo en el momento en que se crean. Esto permite al sistema usar técnicas de almacenamiento diferentes para las dos clases. Los sistemas operativos modernos no hacen esta distinción; todos sus archivos son automáticamente de acceso directo.

5.1.5 Atributos de archivos

Todo archivo tiene un nombre y ciertos datos. Además, todos los sistemas operativos asocian información adicional a cada archivo, por ejemplo, la fecha y hora de creación del archivo, y su tamaño. Llamamos a estos datos adicionales **atributos** del archivo. La lista de atributos varía considerablemente de un sistema a otro. La tabla de la Fig. 5-4 muestra algunas de las posibilidades, pero existen otras. Ningún sistema existente tiene todos estos atributos, pero cada uno está presente en algún sistema.

Campo	Significado
Protección	Quién puede acceder al archivo y de qué modo
Contraseña	Contraseña requerida para acceder al archivo
Creador	Identificador de la persona que creó el archivo
Propietario	Propietario actual
Bandera de sólo lectura	0 = lectura/escritura; 1 = sólo lectura
Bandera de oculto	0 = normal; 1 = no mostrar en los listados
Bandera de sistema	0 = archivo normal; 1 = archivo de sistema
Bandera de archivado	0 = ya se respaldó; 1 = necesita respaldarse
Bandera ASCII/binario	0 = archivo ASCII; 1 = archivo binario
Bandera de acceso aleatorio	0 = sólo acceso secuencial; 1 = acceso aleatorio
Bandera de temporal	0 = normal; 1 = eliminar al salir el proceso
Banderas de candado	0 = sin candado; diferente de cero = con candado
Longitud de registro	Número de bytes en cada registro
Posición de la llave	Distancia de la llave desde el principio de cada registro
Longitud de la llave	Número de bytes del campo de la llave
Tiempo de creación	Fecha y hora de creación del archivo
Tiempo de último acceso	Fecha y hora del último acceso al archivo
Tiempo de último cambio	Fecha y hora en que se modificó por última vez el archivo
Tamaño actual	Número de bytes del archivo
Tamaño máximo	Número de bytes que puede llegar a tener el archivo

Figura 5-4. Algunos posibles atributos de archivo.

Los primeros cuatro atributos tienen que ver con la protección del archivo e indican quién puede acceder a él y quién no. Se puede utilizar todo tipo de esquemas, algunos de los cuales

estudiaremos más adelante. En algunos sistemas el usuario debe presentar una contraseña para acceder a un archivo, en cuyo caso la contraseña debe ser uno de los atributos.

Las banderas son bits o campos cortos que controlan o habilitan alguna propiedad específica. Los archivos ocultos, por ejemplo, no aparecen en los listados de todos los archivos. La bandera de archivado es un bit que indica si el archivo ya se respaldó o no. El programa de respaldo lo pone en cero, y el sistema operativo lo pone en 1 cada vez que el archivo sufre alguna modificación. De este modo, el programa de respaldo puede saber cuáles archivos hay que respaldar. La bandera de temporal permite marcar un archivo para que sea borrado automáticamente cuando el proceso que lo creó termina.

Los campos de longitud de registro, posición de la llave y longitud de la llave sólo están presentes en los archivos cuyos registros pueden consultarse empleando una llave. Dichos campos proporcionan la información requerida para encontrar las llaves.

Los diversos tiempos mantienen un registro de cuándo se creó el archivo, cuándo se accedió a él por última vez y cuándo se modificó por última vez. Estos tiempos pueden servir para diversos fines. Por ejemplo, un archivo fuente que se modificó después de la creación del archivo objeto correspondiente debe recompilarse. Estos campos proporcionan la información necesaria.

El tamaño actual indica qué tan grande es el archivo actualmente. Algunos sistemas operativos de macrocomputadoras exigen que se especifique un tamaño máximo en el momento de crearse un archivo, a fin de poder reservar la cantidad máxima de almacenamiento por adelantado. Los sistemas operativos de las estaciones de trabajo y las computadoras personales tienen la suficiente inteligencia como para prescindir de esta información.

5.1.6 Operaciones con archivos

Los archivos existen para almacenar información que posteriormente se pueda recuperar. Los diferentes sistemas ofrecen distintas operaciones de almacenamiento y recuperación. A continuación reseñamos las llamadas al sistema más comunes relacionadas con archivos.

1. **CRÉATE.** El archivo se crea sin datos. El propósito de la llamada es anunciar que va a haber un archivo y establecer algunos de los atributos.

2. **DELETE.** Cuando el archivo ya no se necesita, es preciso eliminarlo para desocupar el espacio en disco. Siempre hay una llamada al sistema para este fin.

3. **OPEN.** Antes de usar un archivo, un proceso debe abrirlo. El propósito de la llamada **OPEN** es permitir al sistema que obtenga los atributos y la lista de direcciones de disco y los coloque en la memoria principal a fin de agilizar el acceso en llamadas posteriores.

4. **CLOSE.** Una vez concluidos todos los accesos, los atributos y las direcciones de disco ya no son necesarios, por lo que se debe cerrar el archivo para liberar el espacio correspondiente en las tablas internas. Muchos sistemas fomentan esto limitando a los procesos a un número máximo de archivos abiertos. Las escrituras en disco son por bloques, y el cierre de un archivo obliga a escribir el último bloque del archivo, aunque todavía no esté totalmente lleno.

5. READ. Se leen datos del archivo. Por lo regular, los bytes provienen de la posición actual. El invocador debe especificar cuántos datos se necesitan y también debe suministrar un buffer para colocarlos.

6. WRITE. Se escriben datos en el archivo, también, por lo regular, en la posición actual. Si dicha posición es el final del archivo, el tamaño del archivo aumenta. Si la posición actual está a la mitad del archivo, se sobrescribe en los datos existentes, que se pierden irremediamente.

7. APPEND. Esta llamada es una forma restringida de WRITE que sólo puede agregar datos al final del archivo. Los sistemas que ofrecen un juego mínimo de llamadas al sistema generalmente no cuentan con APPEND, pero muchos sistemas ofrecen varias formas de hacer una misma cosa, y a veces incluyen APPEND.

8. SEEK. En el caso de archivos de acceso aleatorio, se requiere un método para especificar el lugar del que deben tomarse los datos. Un enfoque común es tener una llamada al sistema, SEEK, que ajuste el apuntador a la posición actual haciéndolo que apunte a un lugar específico del archivo. Una vez efectuada esta llamada, se pueden leer datos de esa posición o escribirlos en ella.

9. GET ATTRIBUTES. Es frecuente que los procesos necesiten leer los atributos de un archivo para realizar su trabajo. Por ejemplo, el programa make de UNIX se usa comúnmente para administrar proyectos de desarrollo de software que constan de muchos archivos fuente. Cuando se invoca make, examina los tiempos de modificación de todos los archivos fuente y objeto y organiza el número mínimo de compilaciones necesarias para que todo esté actualizado. Para efectuar su trabajo, este comando necesita examinar algunos atributos, a saber, los tiempos de modificación.

10. SET ATTRIBUTES. Algunos de los atributos pueden ser establecidos por el usuario y modificarse después de que se creó el archivo. Esta llamada al sistema hace posible esto. La información de modo de protección es un ejemplo obvio. La mayor parte de las banderas también pertenecen a esta categoría.

11. RENAME. Es común que un usuario necesite cambiar el nombre de un archivo existente. Esta llamada permite hacerlo, aunque no siempre es indispensable, ya que el archivo por lo regular puede copiarse en un archivo nuevo con el nuevo nombre, eliminando después el archivo viejo.

5.2 DIRECTORIOS

A fin de organizar los archivos, los sistemas de archivos casi siempre tienen **directorios** que en muchos sistemas, son también archivos. En esta sección hablaremos de los directorios, su organización, sus propiedades y las operaciones que pueden efectuarse con ellos.

5.2.1 Sistemas de directorio jerárquicos

Un directorio normalmente contiene varias entradas, una por archivo. Una posibilidad se muestra en la Fig. 5-5(a), donde cada entrada contiene el nombre del archivo, los atributos del archivo, y la dirección de disco donde están almacenados los datos. Otra posibilidad se muestra en la Fig. 5-5(b). Aquí una entrada de directorio contiene el nombre del archivo y un apuntador a otra estructura de datos en la que pueden encontrarse los atributos y las direcciones en disco. Ambos sistemas son de uso generalizado.

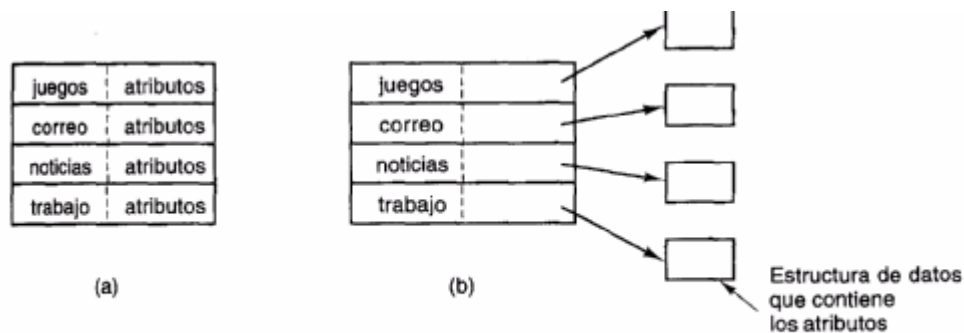


Figura 5-5. (a) Atributos en la entrada de directorio. (b) Atributos en otro lugar.

Cuando se abre un archivo, el sistema operativo examina su directorio hasta encontrar el nombre del archivo por abrir, y luego extrae los atributos y las direcciones en disco, ya sea directamente de la entrada de directorio o de la estructura de datos a la que ésta apunta, y los coloca en una tabla en la memoria principal. Todas las referencias subsecuentes al archivo utilizan la información que está en la memoria principal.

El número de direcciones varía de un sistema a otro. El diseño más sencillo es aquel en el que el sistema mantiene un solo directorio que contiene todos los archivos de todos los usuarios, como se ilustra en la Fig. 5-6(a). Si hay muchos usuarios, y éstos escogen los mismos nombres de archivo (p. ej., correo y juegos), los conflictos y la confusión resultante pronto harán inmanejable el sistema. Este modelo se utilizó en los primeros sistemas operativos de microcomputadoras, pero ya casi nunca se encuentra.

Una mejora de la idea de tener un solo directorio para todos los archivos del sistema es tener un directorio por usuario [véase la Fig. 5-6(b)]. Este diseño elimina los conflictos de nombres entre usuarios pero no es satisfactorio para quienes tienen un gran número de archivos. Es muy común que los usuarios quieran agrupar sus archivos atendiendo a patrones lógicos. Un profesor, por ejemplo, podría tener una colección de archivos que juntos constituyan un libro que está escribiendo para un curso, una segunda colección de archivos que contenga programas que los estudiantes de otro curso hayan presentado, un tercer grupo de archivos que contenga el código de un sistema avanzado de escritura de compiladores que él esté construyendo, un cuarto grupo de archivos que contenga propuestas para obtener subvenciones, así como otros archivos para correo electrónico, minutas de reuniones, artículos que esté escribiendo, juegos, etc. Se requiere algún método para agrupar estos archivos en formas flexibles elegidas por el usuario.

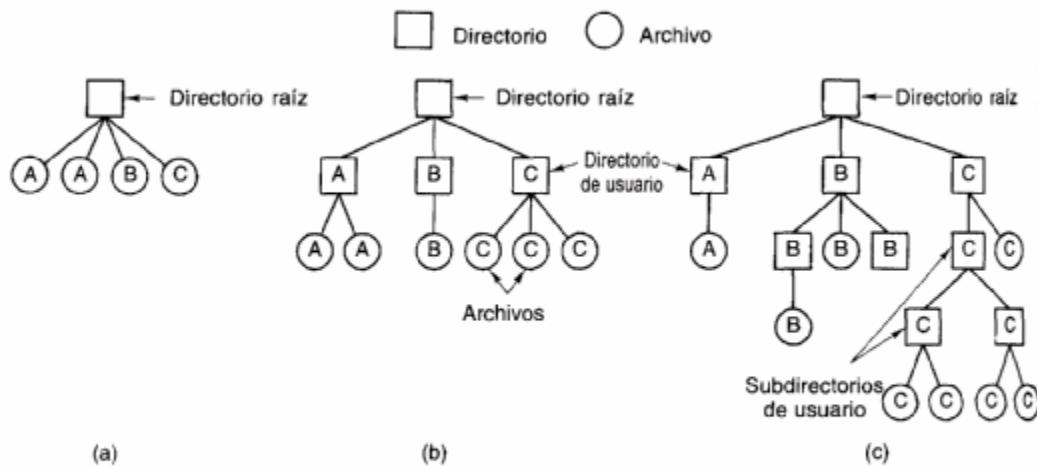


Figura 5-6. Tres diseños de sistema de archivos. (a) Un solo directorio compartido por todos los usuarios. (b) Un directorio por usuario. (c) Árbol arbitrario por usuario. Las letras indican el propietario del directorio o archivo.

Lo que hace falta es una jerarquía general (es decir, un árbol de directorios). Con este enfoque, cada usuario puede tener tantos directorios como necesite para poder agrupar sus archivos según patrones naturales. Este enfoque se muestra en la Fig. 5-6(c). Aquí, los directorios A, B y C, contenidos en el directorio raíz, pertenecen cada uno a un usuario distinto. Dos de estos usuarios han creado subdirectorios para proyectos en los que están trabajando.

5.2.2 Nombres de ruta

Cuando el sistema de archivos se organiza en forma de árbol de directorios, se necesita un método para especificar los nombres de los archivos. Hay dos métodos diferentes en uso con el primero, se asigna a cada archivo un **nombre de ruta absoluto** que consiste en la ruta desde el directorio raíz hasta el archivo. Por ejemplo, la ruta `/usr/ast/mailbox` indica que el directorio raíz contiene un subdirectorio `usr`, que a su vez contiene un subdirectorio `ast`, el cual contiene el archivo `mailbox`. Los nombres de ruta absolutos siempre parten del directorio raíz y los únicos. En UNIX los componentes de la ruta se separan con `/`. En MS-DOS el separador es `;`; en MULTICS es `>`. Sea cual sea el carácter empleado, si el primer carácter del nombre de ruta es el separador, la ruta es absoluta.

El otro tipo de nombre es el **nombre de ruta relativo**, el cual se utiliza en combinación con el concepto de directorio de trabajo (también llamado directorio actual). Un usuario puede tener un directorio como directorio de trabajo actual, en cuyo caso todos los nombres de archivo que comiencen en el directorio raíz se tomarán en relación con el directorio de trabajo. Por ejemplo, si el directorio de trabajo actual es `/usr/ast`, entonces se podrá hacer referencia al archivo cuya ruta

absoluta es `/usr/ast/mailbox` simplemente como `mailbox`. En otras palabras, el comando de UNIX `cp/usr/ast/mailbox/usr/ast/mailbox.bat`

y el comando

```
cp mailbox mailbox.bat
```

harán exactamente lo mismo si el directorio de trabajo es `/usr/ast`. La forma relativa suele ser más cómoda, pero hace lo mismo que la forma absoluta.

Algunos programas necesitan acceder a un archivo específico sin considerar el directorio de trabajo. En tal caso, siempre deben usar nombres de ruta absolutos. Por ejemplo, un revisor ortográfico podría necesitar leer `/usr/lib/dictionary` para efectuar su trabajo. En este caso, el programa deberá usar el nombre de ruta absoluto completo porque no sabe cuál será el directorio de trabajo cuando sea invocado. El nombre de ruta absoluta siempre funciona, sea cual sea el directorio de trabajo.

Desde luego, si el revisor de ortografía necesita un gran número de archivos de `/usr/lib`, una estrategia alternativa es que emita una llamada al sistema para cambiar su directorio de trabajo a `/usr/lib`, y luego utilice sólo `dictionary` como primer parámetro de `open`. Al cambiar explícitamente su directorio de trabajo, el programa sabe con certeza en qué lugar del árbol de directorios está, y puede usar rutas relativas.

En la mayor parte de los sistemas, cada proceso tiene su propio directorio de trabajo, así que cuando un proceso cambia su directorio de trabajo y luego sale, ningún otro proceso resulta afectado y no quedan rastros del cambio en el sistema de archivos. De esta forma, un proceso siempre puede cambiar sin peligro su directorio de trabajo cada vez que le resulte cómodo hacerlo. Por otro lado, si un procedimiento de biblioteca cambia el directorio de trabajo y al terminar no lo cambia otra vez al que estaba vigente antes, es posible que el resto del programa no funcione, ya que ahora podría estar en un directorio distinto del que cree que está. Por esta razón, los procedimientos de biblioteca casi nunca cambian el directorio de trabajo y, cuando necesitan hacerlo, siempre lo restauran antes de regresar.

La mayor parte de los sistemas operativos que manejan un sistema de directorios jerárquico tiene dos entradas especiales en cada directorio, `"."` y `".."`, generalmente pronunciados "punto" y "punto punto". Punto se refiere al directorio actual; punto punto se refiere a su padre. Para ver cómo se utilizan éstos, consideremos el árbol de archivos UNIX de la Fig. 5-7. Cierta proceso tiene `/usr/ast` como directorio de trabajo, y puede usar `..` para subir por el árbol. Por ejemplo, el proceso en cuestión puede copiar el archivo `/usr/lib/dictionary` en su propio directorio empleando el comando de shell

```
cp ../lib/dictionary .
```

La primera ruta hace que el sistema que suba (al directorio `usr`) y luego baje al directorio `lib` para encontrar el archivo `dictionary`.

El segundo argumento indica el directorio actual. Cuando proporcionamos al comando `cp` un nombre de directorio (incluido punto) como último argumento, copia todos los archivos ahí. Desde luego, una forma más normal de efectuar el copiado sería escribir

```
¡ cp /usr/lib/dictionary.
```

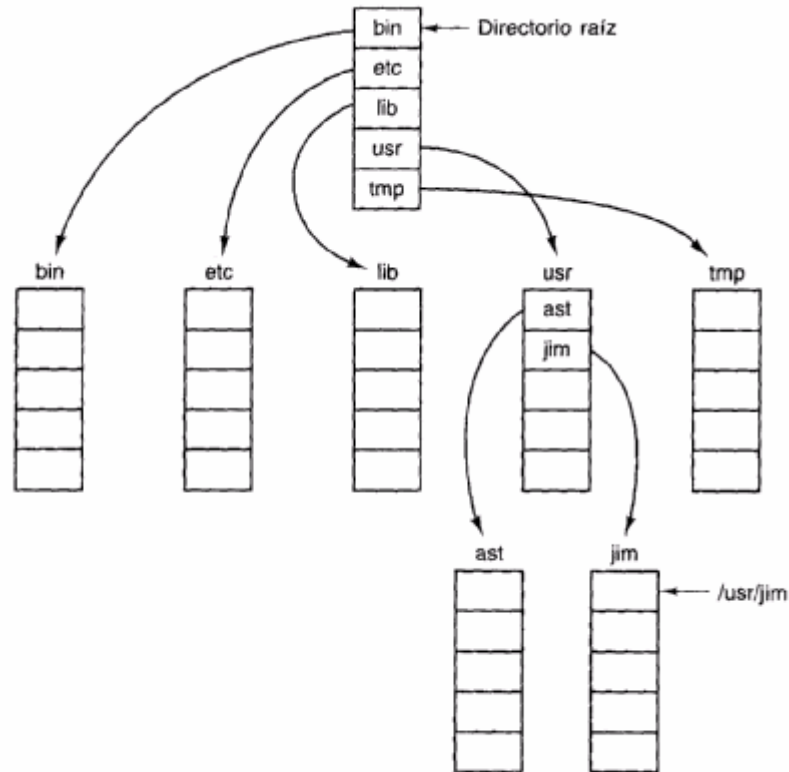


Figura 5-7. Un árbol de directorios UNIX.

Aquí el empleo de punto ahorra al usuario el trabajo de escribir dictionary otra vez.

5.2.3 Operaciones con directorios

Las llamadas al sistema permitidas para administrar directorios muestran variaciones más amp. de un sistema a otro que las llamadas para archivos. A fin de dar una idea de la naturaleza modo de operar de esas llamadas, presentaremos una muestra (tomada de UNIX).

1. **CRÉATE.** Se crea un directorio, que está vacío con la excepción de punto y punto, punto, que el sistema (o, en unos pocos casos, el programa mkdir) coloca ahí automáticamente.

2. **DELETE.** Se elimina un directorio. Sólo puede eliminarse un directorio vacío. Un directorio que sólo contiene punto y punto punto se considera vacío, ya que éstos normalmente no pueden eliminarse.

3. **OPENDIR.** Los directorios pueden leerse. Por ejemplo, para listar todos los archivo de un directorio, un programa para emitir listados abre el directorio y lee los nombres

de los archivos que contiene. Antes de poder leer un directorio, es preciso abrirlo, de forma análoga a como se abren y leen los archivos.

4. CLOSER. Una vez que se ha leído un directorio, debe cerrarse para liberar espacio de tablas internas.

5. READDIR. Esta llamada devuelve la siguiente entrada de un directorio abierto. Antes, era posible leer directorios empleando la llamada al sistema READ normal, pero ese enfoque tiene la desventaja de obligar al programador a conocer y manejar la estructura interna de los directorios. En contraste, READDIR siempre devuelve una entrada en un formato estándar, sin importar cuál de las posibles estructuras de directorio se esté usando.

6. RENAME. En muchos sentidos, los directorios son iguales que los archivos y podemos cambiar su nombre tal como hacemos con los archivos.

7. LINK. El enlace (linking) es una técnica que permite a un archivo aparecer en más de un directorio. Esta llamada al sistema especifica un archivo existente y un nombre de ruta, y crea un enlace del archivo existente al nombre especificado por la ruta. De este modo, el mismo archivo puede aparecer en múltiples directorios.

8. UNLINK. Se elimina una entrada de directorio. Si el archivo que está siendo desvinculado sólo está presente en un directorio (el caso normal), se eliminará del sistema de archivos. Si el archivo está presente en varios directorios, sólo se eliminará el nombre de ruta especificado; los demás seguirán existiendo. En UNIX, la llamada al sistema para eliminar archivos (que ya vimos antes) es, de hecho, UNLINK.

La lista anterior incluye las llamadas más importantes, pero hay otras, por ejemplo, para administrar la información de protección asociada con un directorio.

5.3 IMPLEMENTACIÓN DE SISTEMAS DE ARCHIVOS

Ya es momento de pasar de la perspectiva del usuario del sistema de archivos a la perspectiva del implementador. A los usuarios les interesa la forma de nombrar los archivos, las operaciones que pueden efectuarse con ellos, el aspecto que tiene el árbol de directorios y cuestiones de interfaz por el estilo. A los implementadores les interesa cómo están almacenados los archivos y directorios, cómo se administra el espacio en disco y cómo puede hacerse que todo funcione de forma eficiente y confiable. En las secciones siguientes examinaremos varias de estas áreas para conocer los problemas y las concesiones.

5.3.1 Implementación de archivos

Tal vez el aspecto más importante de la implementación del almacenamiento en archivos sea poder relacionar bloques de disco con archivos. Se emplean diversos métodos en los diferentes sistemas operativos. En esta sección examinaremos algunos de ellos.

Asignación contigua

El esquema de asignación más sencillo es almacenar cada archivo como un bloque contiguo de datos en el disco. Así, en un disco con bloques de 1K, a un archivo de 50K se le asignarían 50 bloques consecutivos. Este esquema tiene dos ventajas importantes. Primera, la implementación es sencilla porque para saber dónde están los bloques de un archivo basta con recordar un número, la dirección en disco del primer bloque. Segunda, el rendimiento es excelente porque es posible leer todo el archivo del disco en una sola operación.

Desafortunadamente, la asignación contigua tiene también dos ventajas igualmente importantes. Primera, no es factible si no se conoce el tamaño máximo del archivo en el momento en que se crea el archivo. Sin esta información, el sistema operativo no sabrá cuánto espacio en disco debe reservar. Sin embargo, en los sistemas en los que los archivos deben escribirse de un solo golpe, el método puede usarse con gran provecho.

La segunda desventaja es la fragmentación del disco que resulta de esta política de asignación. Se desperdicia espacio que de otra forma podría haberse aprovechado. La compactación del disco suele tener un costo prohibitivo, aunque tal vez podría efectuarse de noche cuando el sistema estaría ocioso.

Asignación por lista enlazada

El segundo método para almacenar archivos es guardar cada uno como una lista enlazada de bloques de disco, como se muestra en la Fig. 5-8. La primera palabra de cada bloque se emplea como apuntador al siguiente. El resto del bloque se destina a datos.

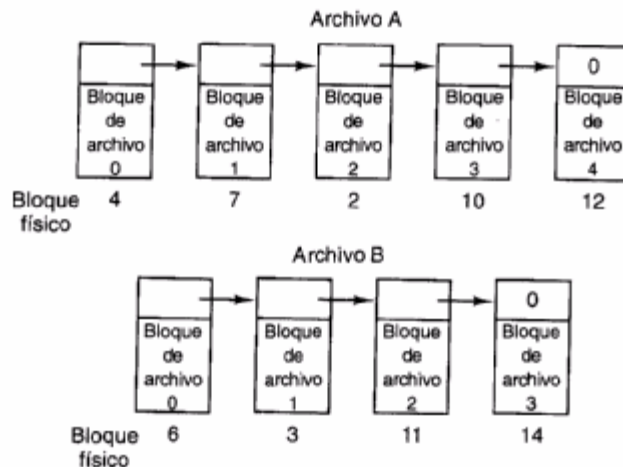


Figura 5-8. Almacenamiento de un archivo como lista enlazada de bloques de disco.

A diferencia de la asignación contigua, con este método es posible utilizar todos los bloques. No se pierde espacio por fragmentación del disco (excepto por fragmentación interna en el último |

bloque). Además, basta con que en la entrada de directorio se almacene la dirección en disco del primer bloque. El resto del archivo puede encontrarse siguiendo los enlaces.

Por otro lado, aunque la lectura secuencial de un archivo es sencilla, el acceso aleatorio es extremadamente lento. Además, la cantidad de almacenamiento de datos en un bloque ya no es una potencia de dos porque el apuntador ocupa unos cuantos bytes. Si bien tener un tamaño peculiar no es fatal, resulta menos eficiente porque muchos programas leen y escriben en bloques cuyo tamaño es una potencia de dos.

Asignación por lista enlazada empleando un índice

Las dos desventajas de la asignación por lista enlazada pueden eliminarse si se toma la palabra de apuntador de cada bloque y se le coloca en una tabla o índice en la memoria. La Fig. 5-9 muestra el aspecto que la tabla tendría para el ejemplo de la Fig. 5-8. En ambas figuras, tenemos dos archivos. El archivo A usa los bloques de disco 4,7,2,10 y 12, en ese orden, y el archivo B usa los bloques de disco 6, 3, 11 y 14, en ese orden. Si usamos la tabla de la Fig. 5-9, podemos comenzar en el bloque 4 y seguir la cadena hasta el final. Lo mismo puede hacerse comenzando en el bloque 6.

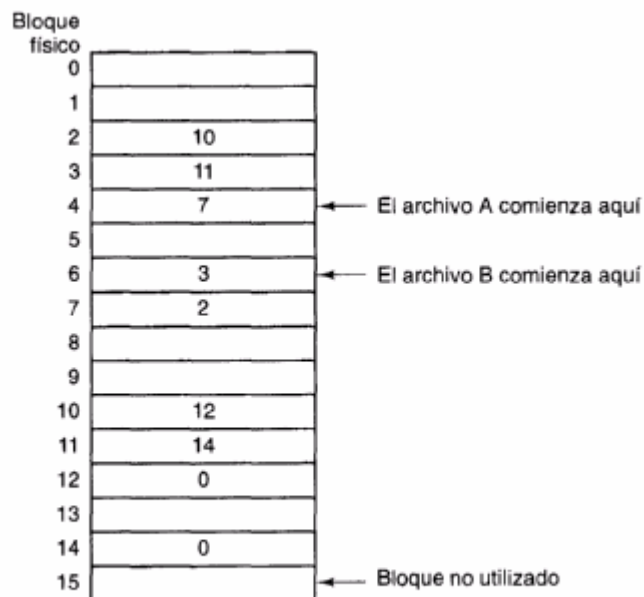


Figura 5-9. Asignación por lista enlazada empleando una tabla en la memoria principal.

Si se emplea esta organización, todo el bloque está disponible para datos. Además, el acceso directo es mucho más fácil. Aunque todavía hay que seguir la cadena para encontrar una distancia dada dentro de un archivo, la cadena está por completo en la memoria, y puede seguirse sin tener que consultar el disco. Al igual que con el método anterior, basta con guardar un solo entero (el

número del bloque inicial) en la entrada de directorio para poder localizar todos los bloques, por más grande que sea el archivo. MS-DOS emplea este método para la asignación en disco.

La desventaja primordial de este método es que toda la tabla debe estar en la memoria todo el tiempo para que funcione. En el caso de un disco grande con, digamos, 500 000 bloques de 1K (500M), la tabla tendrá 500 000 entradas, cada una de las cuales tendrá que tener un mínimo de 3 bytes. Si se desea acelerar las búsquedas, se necesitarán 4 bytes. Así, la tabla ocupará de 1.5 a 2 megabytes todo el tiempo, dependiendo de si el sistema se optimiza en cuanto al espacio o en cuanto al tiempo. Aunque MS-DOS emplea este mecanismo, evita manejar tablas muy grandes empleando bloques grandes (de hasta 32K) en los discos de gran tamaño.

Nodos-i

Nuestro último método para saber cuáles bloques pertenecen a cuál archivo consiste en asociar a cada archivo una pequeña tabla llamada **nodo-i (nodo-índice)**, que lista los atributos y las direcciones en disco de los bloques del archivo, como se muestra en la Fig. 5-10

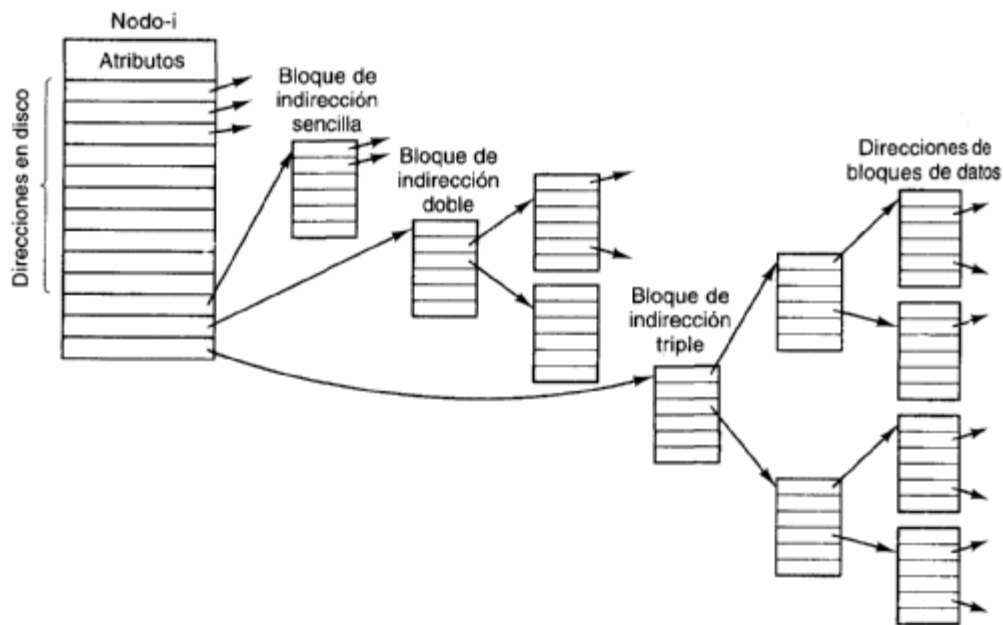


Figura 5-10. Un nodo-i.

Las primeras pocas direcciones de disco se almacenan en el nodo-i mismo, así que en el caso de archivos pequeños toda la información está contenida en el nodo-i, que se trae del disco a la memoria principal cuando se abre el archivo. En el caso de archivos más grandes, una de las direcciones del nodo-i es la dirección de un bloque de disco llamado bloque de indirección sencilla. Este bloque contiene direcciones de disco adicionales. Si esto todavía no es suficiente, otra dirección del nodo-i, llamada bloque de indirección doble, contiene la dirección de un

bloque que contiene una lista de bloques de indirección sencilla. Cada uno de estos bloques de indirección sencilla apunta a unos cuantos cientos de bloques de datos. Si ni siquiera con esto basta, se puede usar también un **bloque de dirección triple**. UNIX emplea este esquema.

5.3.2 Implementación de directorios

Antes de poder leer un archivo, hay que abrirlo. Cuando se abre un archivo, el sistema operativo usa el nombre de ruta proporcionado por el usuario para localizar la entrada de directorio. Esta entrada proporciona la información necesaria para encontrar los bloques de disco. Dependiendo del sistema, esta información puede ser la dirección en disco de todo el archivo (asignación contigua), el número del primer bloque (ambos esquemas de lista enlazada) o el número del nodo-*i*. En todos los casos, la función principal del sistema de directorios es transformar el nombre ASCII del archivo en la información necesaria para localizar los datos.

Un problema muy relacionado con el anterior es dónde deben almacenarse los atributos. Una posibilidad obvia es almacenarlos directamente en la entrada de directorio. Muchos sistemas hacen precisamente esto. En el caso de sistemas que usan nodos-*i*, otra posibilidad es almacenar los atributos en el nodo-*i*, en lugar de en la entrada de directorio. Como veremos más adelante, este método tiene ciertas ventajas respecto a la colocación de los atributos en la entrada de directorio.

Directorios en CP/M

Iniciemos nuestro estudio de los directorios con un ejemplo especialmente sencillo, el de CP/M (Golden y Pechura, 1986), ilustrado en la Fig. 5-11. En este sistema, sólo hay un directorio, así que todo lo que el sistema de archivos tiene que hacer para consultar un nombre de archivo es buscarlo en el único directorio. Una vez que encuentra la entrada, también tiene los números de bloque en el disco, ya que están almacenados ahí mismo, en la entrada, lo mismo que todos los atributos. Si el archivo ocupa más bloques de disco de los que caben en una entrada, se asignan al archivo entradas de directorio adicionales.

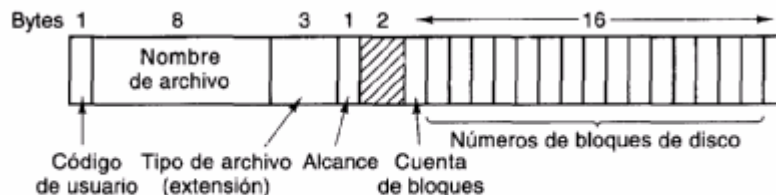


Figura 5-11. Entrada de directorio que contiene los números de bloque de disco para cada archivo.

Los campos de la Fig. 5-11 tienen los siguientes significados. El campo de código de usuario indica qué usuario es el propietario del archivo. Durante una búsqueda, sólo se examinan las entradas que pertenecen al usuario que está actualmente en sesión. Los siguientes dos campos dan el nombre y la extensión del archivo. El campo alcance (extent) es necesario porque un archivo

con más de 16 bloques ocupa múltiples entradas de directorio. Este campo sirve para saber cuál entrada es primero, cuál segunda, etc. El campo de cuenta de bloques indica cuántas de las 16 posibles entradas de bloque se están usando. Los últimos 16 campos contienen los números de bloque de disco mismos. Es posible que el último bloque no esté lleno, así que el sistema no tiene forma de conocer el tamaño exacto de un archivo hasta el último byte (es decir, registra los tamaños de archivo en bloques, no en bytes).

Directorios en MS-DOS

Consideremos ahora algunos ejemplos de sistemas con árboles de directorios jerárquicos. La Fig. 5-12 muestra una entrada de directorio de MS-DOS, la cual tiene 32 bytes de longitud y contiene el nombre de archivo, los atributos y el número del primer bloque de disco. El primer número de bloque se emplea como índice de una tabla del tipo de la de la Fig. 5-9. Siguiendo la cadena, se pueden encontrar todos los bloques.

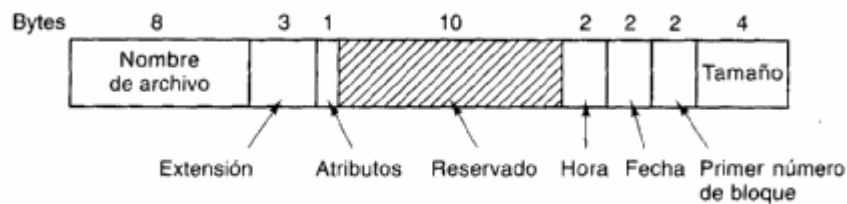


Figura 5-12. La entrada de directorio de MS-DOS.

En MS-DOS, los directorios pueden contener otros directorios, dando lugar a un sistema de archivos jerárquico. En este sistema operativo es común que los diferentes programas de aplicación comiencen por crear un directorio en el directorio raíz y pongan ahí todos sus archivos, con objeto de que no haya conflictos entre las aplicaciones.

Directorios en UNIX

La estructura de directorios que se usa tradicionalmente en UNIX es en extremo sencilla, como se aprecia en la Fig. 5-13. Cada entrada contiene sólo un nombre de archivo y su número de nodo-i. Toda la información acerca del tipo, tamaño, tiempos, propietario y bloques de disco está contenida en el nodo-i. Algunos sistemas UNIX tienen una organización distinta, pero en todos los casos una entrada de directorio contiene en última instancia sólo una cadena ASCII y un número de nodo-i.

Cuando se abre un archivo, el sistema de archivos debe tomar el nombre que se le proporciona y localizar sus bloques de disco. Consideremos cómo se busca el nombre de ruta /usr/ast/mbox. Usaremos UNIX como ejemplo, pero el algoritmo es básicamente el mismo para todos los sistemas de directorios jerárquicos. Lo primero que hace el sistema de archivos es localizará directorio raíz. En UNIX su nodo-i está situado en un lugar fijo del disco.

A continuación, el sistema de archivos busca el primer componente de la ruta, usr, en el directorio raíz para encontrar el número de nodo-i del archivo /usr. La localización de un nodo-itj

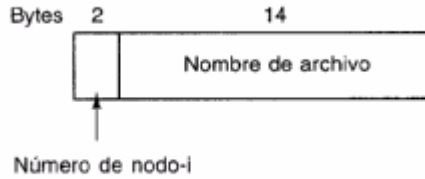


Figura 5-13. Una entrada de directorio UNIX.

una vez que se tiene su número es directa, ya que cada uno tiene una posición fija en el disco. Con la ayuda de este nodo-i, el sistema localiza el directorio correspondiente a./usr y busca el siguiente componente, ast, en él. Una vez que el sistema encuentra la entrada para ast, obtiene el nodo-i del directorio /usr/ast. Con este nodo, el sistema puede encontrar el directorio mismo y buscar mbox. A continuación se trae a la memoria el nodo-i de este archivo y se mantiene ahí hasta que se cierra el archivo. El proceso de búsqueda se ilustra en la F'g. 5-14.

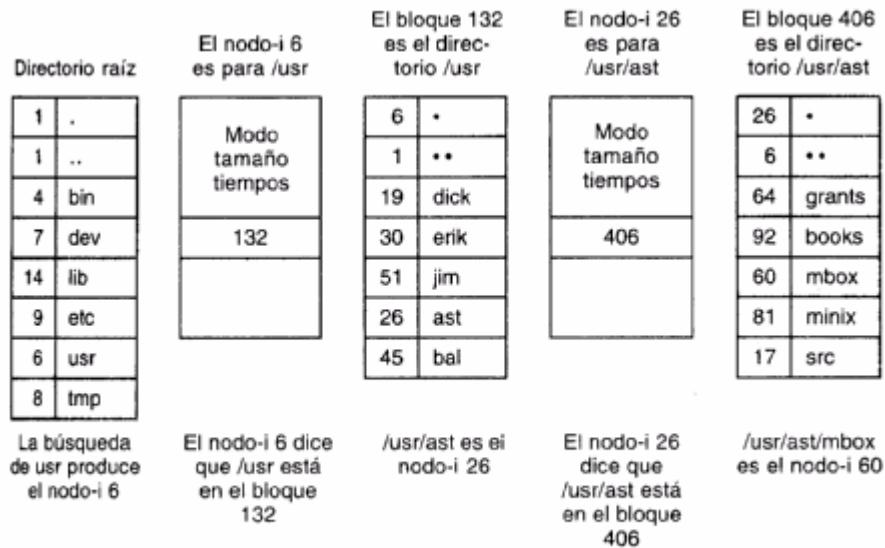


Figura 5-14. Pasos para buscar /usr/ast/mbox.

Los nombres de ruta relativos se buscan de la misma forma que los absolutos, sólo que el punto de partida es el directorio de trabajo en lugar del directorio raíz. Cada directorio tiene entradas para . y .., que se colocan ahí cuando se crea el directorio. La entrada . tiene el número de nodo-i del directorio actual, y la entrada .. tiene el número de nodo-i del directorio padre. Así, un procedimiento que busque ../dick/prog.c simplemente buscará.. en el directorio de trabajo, encontrará el número de nodo-i de su directorio padre y buscará dick en ese directorio. No se requiere ningún mecanismo especial para manejar estos nombres. En lo que al sistema de directorios concierne, se trata de cadenas ASCII ordinarias, lo mismo que los demás nombres.

5.3.3 Administración del espacio en disco

Los archivos normalmente se almacenan en disco, así que la administración del espacio en disco es de interés primordial para los diseñadores de sistemas de archivos. Hay dos posibles estrategias para almacenar un archivo de n bytes: asignar n bytes consecutivos de espacio en disco, o dividir el archivo en varios bloques (no necesariamente) contiguos. Un trueque similar (entre segmentación pura y paginación) está presente en los sistemas de administración de memoria.

El almacenamiento de un archivo como secuencia contigua de bytes tiene el problema obvio de que, si el archivo crece, probablemente tendrá que pasarse a otro lugar del disco. El mismo problema ocurre con los segmentos en la memoria, excepto que el traslado de un segmento en la memoria es una operación relativamente rápida comparada con el traslado de un archivo de una posición en el disco a otra. Por esta razón, casi todos los sistemas de archivos dividen los archivos en bloques de tamaño fijo que no necesitan estar adyacentes.

Tamaño de bloque

Una vez que se ha decidido almacenar archivos en bloques de tamaño fijo, surge la pregunta de qué tamaño deben tener los bloques. Dada la forma como están organizados los discos, el sector, la pista y el cilindro son candidatos obvios para utilizarse como unidad de asignación. En un sistema con paginación, el tamaño de página también es un contendiente importante.

Tener una unidad de asignación grande, como un cilindro, implica que cada archivo, incluso un archivo de un byte, ocupará todo un cilindro. Estudios realizados (Mullender y Tanenbaum, 1984) han demostrado que la mediana del tamaño de los archivos en los entornos UNIX es de alrededor de 1K, así que asignar un cilindro de 32K a cada archivo implicaría un desperdicio de $31/32 = 97\%$ del espacio en disco total. Por otro lado, el empleo de una unidad de asignación pequeña implica que cada archivo consistirá en muchos bloques. La lectura de cada bloque normalmente requiere una búsqueda y un retardo rotacional, así que la lectura de un archivo que consta de muchos bloques pequeños será lenta.

Por ejemplo, consideremos un disco con 32 768 bytes por pista, tiempo de rotación de 16.67 ms y tiempo de búsqueda medio de 30 ms. El tiempo en milisegundos requerido para leer un bloque de k bytes es la suma de los tiempos de búsqueda, retardo rotacional y transferencia:

$$30 + 8.3 + (k/32768) \times 16.67$$

La curva continua de la Fig. 5-15 muestra la tasa de datos para un disco con estas características en función del tamaño de bloque. Si utilizamos el supuesto burdo de que todos los bloques son de 1K (la mediana de tamaño medida), la curva de guiones de la Fig. 5-15 indicará la eficiencia de espacio del disco. La mala noticia es que una buena utilización del espacio (tamaño de bloque $< 2K$) implica tasas de datos bajas y viceversa. La eficiencia de tiempo y la eficiencia de espacio están inherentemente en conflicto.

El término medio usual es escoger un tamaño de bloque de 512, 1K o 2K bytes. Si se escoge un tamaño de bloque de 1K en un disco cuyos sectores son de 512 bytes, el sistema de archivos siempre leerá o escribirá dos sectores consecutivos y los tratará como una sola unidad, indivisi-

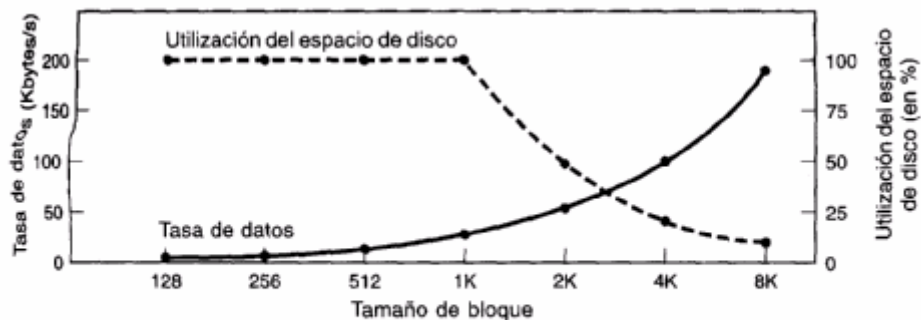


Figura 5-15. La curva continua (escala de la izquierda) da la tasa de datos de un disco. La curva de guiones (escala de la derecha) da la eficiencia de espacio de disco. Todos los archivos son de 1K.

ble. Sea cual sea la decisión que se tome, probablemente convendrá reevaluarla periódicamente, ya que, al igual que sucede con todos los aspectos de la tecnología de computadoras, los usuarios aprovechan los recursos más abundantes exigiendo todavía más. Un administrador de sistema informa que el tamaño medio de los archivos en el sistema universitario que administra ha crecido lentamente con el paso de los años, y que en 1997 el tamaño medio de los archivos ha crecido a 12K para los estudiantes y 15K para los profesores.

Administración de bloques libres

Una vez que se ha escogido el tamaño de bloque, el siguiente problema es cómo seguir la pista a los bloques libres. Se utilizan ampliamente dos métodos, mismos que se ilustran en la Fig. 5-16. El primero consiste en usar una lista enlazada de bloques de disco, en la que cada bloque contiene tantos números de bloques de disco libres como quepan en él. Con bloques de 1K y números de bloque de 32 bits, cada bloque de la lista libre contiene los números de 255 bloques libres. (Se requiere una ranura para el apuntador al siguiente bloque.) Un disco de 200M necesita una lista libre de, como máximo, 804 bloques para contener los 200K números de bloque. Es común que se usen bloques libres para contener la lista libre.

La otra técnica de administración del espacio libre es el mapa de bits. Un disco con n bloques requiere un mapa de bits con n bits. Los bloques libres se representan con unos en el mapa, y los bloques asignados con ceros (o viceversa). Un disco de 200M requiere 200K bits para el mapa, mismos que ocupan sólo 25 bloques. No es sorprendente que el mapa de bits requiera menos espacio, ya que sólo usa un bit por bloque, en vez de 32 como en el modelo de lista enlazada. Sólo si el disco está casi lleno el esquema de lista enlazada requerirá menos bloques que el mapa de bits.

Si hay suficiente memoria principal para contener el mapa de bits, este método generalmente es preferible. En cambio, si sólo se puede dedicar un bloque de memoria para seguir la pista a los bloques de disco libres, y el disco está casi lleno, la lista enlazada podría ser mejor. Con sólo un bloque del mapa de bits en la memoria, podría ser imposible encontrar bloques libres en él, causando accesos a disco para leer el resto del mapa de bits. Cuando un bloque nuevo de la lista enlazada se carga en la memoria, es posible asignar 255 bloques de disco antes de tener que traer del disco el siguiente bloque de la lista.

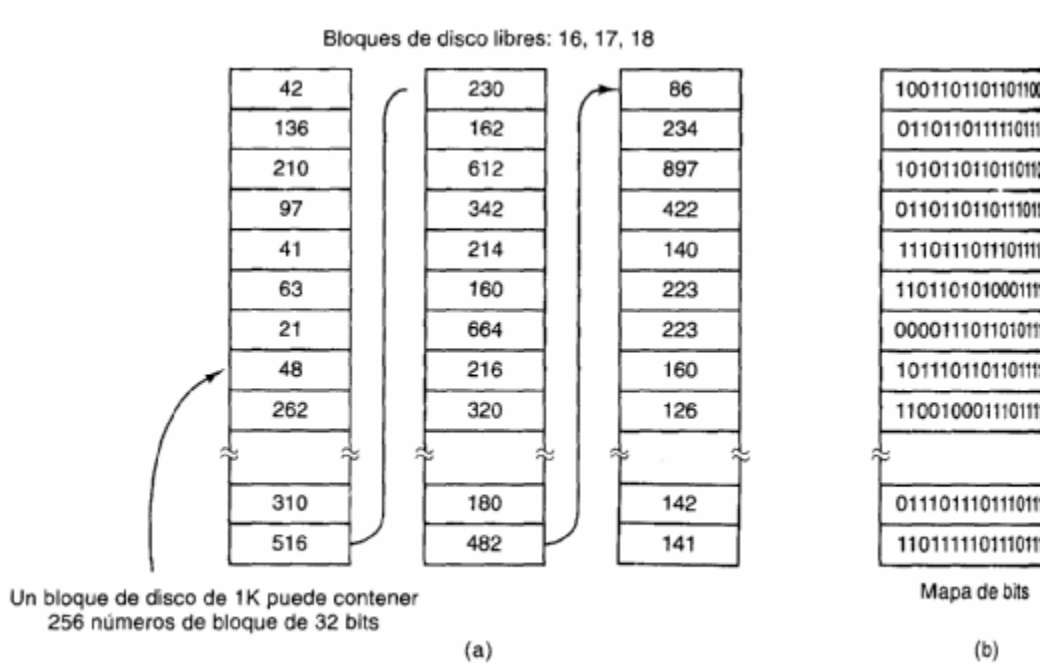


Figura 5-16. (a) Almacenamiento de la lista libre en una lista enlazada. (b) Mapa de bits.

5.3.4 Confiabilidad del sistema de archivos

La destrucción de un sistema de archivos suele ser un desastre mucho peor que la destrucción una computadora. Si una computadora se destruye por un incendio, picos de descarga eléctrica o una taza de café vertida sobre el teclado, esto implica una molestia y cuesta dinero, pero generalmente se puede adquirir un sustituto con un mínimo de problemas. Las computadoras personales económicas pueden reemplazarse en unas cuantas horas con sólo visitar una tienda (excepto las universidades, donde la aprobación de una orden de compra requiere tres comités, con firmas y 90 días).

Si el sistema de archivos de una computadora se pierde irremisiblemente, sea por causa (hardware, del software, o ratas que mordisquearon los discos flexibles, la restauración de toda información es difícil, tardada y, en muchos casos, imposible. Para las personas cuyos programas, documentos, archivos de clientes, registros fiscales, bases de datos, planes de mercadeo otros datos han dejado de existir, las consecuencias pueden ser catastróficas. Si bien el sistema archivos no puede ofrecer protección contra la destrucción física del equipo y los medios, puede ayudar a proteger la información. En esta sección examinaremos algunas cuestiones que intervienen en la salvaguarda del sistema de archivos.

Los discos pueden tener bloques defectuosos, como se apuntó en el capítulo 3. Los discos flexibles normalmente son perfectos cuando salen de la fábrica, pero pueden adquirir defectos durante el uso. Los discos winchester con frecuencia tienen bloques defectuosos aun siendo nuevos; simplemente cuesta demasiado fabricarlos completamente libres de todo defecto. De hecho, antes los discos duros solían entregarse con una lista de los bloques defectuosos descu-

tuertos por las pruebas del fabricante. En tales discos se reserva un sector para contener una lista de bloques defectuosos. Cuando se inicializa originalmente el controlador en hardware, éste lee la lista de bloques defectuosos y escoge un bloque (o una pista) de repuesto para sustituir los defectuosos, registrando la correspondencia en la lista de bloques defectuosos. A partir de entonces, todas las solicitudes que pidan el bloque defectuoso usarán el de repuesto. Cada vez que se descubren nuevos errores se actualiza esta lista como parte de un formato de bajo nivel.

Ha habido una mejoría constante en las técnicas de fabricación, de modo que los bloques defectuosos son menos comunes que antaño; sin embargo, no han desaparecido. El controlador en hardware de una unidad de disco moderna es muy complejo, como se vio en el capítulo 3. En estos discos, las pistas tienen por lo menos un sector más que el número de sectores necesario, con objeto de que al menos un punto defectuoso pueda pasarse por alto dejándolo en un hueco entre dos sectores consecutivos. También hay unos cuantos sectores de repuesto en cada cilindro con objeto de que el controlador pueda establecer una nueva correspondencia de sectores si se percata de que un sector requiere más de cierto número de reintentos para leerse o escribirse. Así, el usuario casi nunca se da cuenta de la existencia de bloques defectuosos ni de su administración. No obstante, cuando un disco IDE o SCSI moderno falla, casi siempre falla gravemente, porque se ha quedado sin sectores de repuesto. Los discos SCSI informan de un "error recuperado" cuando alteran la correspondencia de un bloque. Si el controlador en software se da cuenta de esto y exhibe un mensaje en la pantalla, el usuario sabrá que es momento de comprar un nuevo disco cuando estos mensajes comiencen a aparecer con frecuencia.

Existe una solución de software sencilla al problema de los bloques defectuosos, apropiado para usarse en los discos menos modernos. La estrategia requiere que el usuario del sistema de archivos construya cuidadosamente un archivo que contenga todos los bloques defectuosos. Esta técnica retira dichos bloques de la lista libre, así que nunca se usarán para archivos de datos. En tanto nunca se lea ni escriba el archivo de bloques defectuosos, no habrá problemas. Se debe tener cuidado durante los respaldos de disco para evitar la lectura de este archivo.

Respaldos

Incluso teniendo una estrategia ingeniosa para manejar los bloques defectuosos, es importante respaldar los archivos con frecuencia. Después de todo, cambiar automáticamente a una pista de repuesto después de que se ha arruinado un bloque que contenía datos cruciales es algo parecido a cerrar la puerta de la caballeriza después de que ha escapado un valioso caballo de carreras.

Los sistemas de archivos en disco flexible se pueden respaldar con sólo copiar todo el disco en uno en blanco. Los sistemas de archivos en discos winchester pequeños se pueden respaldar vaciando todo el disco en cinta magnética. Entre las tecnologías actuales están los cartuchos de cinta de 150M y las cintas Exabyte o DAT de 8G.

En el caso de winchester grandes (p. ej., 10GB), el respaldo de toda la unidad en cinta es tedioso y tardado. Una estrategia que es fácil de implementar pero desperdicia la mitad del espacio del almacenamiento es instalar en cada computadora dos unidades de disco en lugar de una. Ambas unidades se dividen en dos mitades: datos y respaldo. Cada noche la porción de datos de la unidad 0 se copia en la porción de respaldo de la unidad 1, y viceversa, como se muestra en la Fig. 5-17. De esta forma, si una unidad se arruina por completo, no se perderá información.

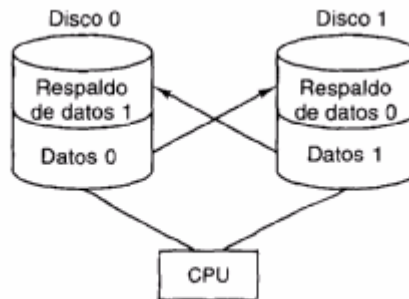


Figura 5-17. El respaldo de cada unidad en la otra desperdicia la mitad del espacio de almacenamiento.

Una alternativa al vaciado diario de todo el sistema de archivos es realizar vaciados incrementales. La forma más sencilla de vaciado incremental consiste en efectuar un vaciado completo periódicamente, digamos cada semana o cada mes, y realizar un vaciado diario de sólo aquellos archivos que han sido modificados después del último vaciado completo. Un esquema mejor sería vaciar sólo aquellos archivos que han cambiado desde que se vaciaron por última vez.

Para implementar este método, se debe mantener en disco una lista de los tiempos de vaciado para cada archivo. El programa de vaciado revisa cada archivo del disco. Si un archivo fue modificado después de la última vez que se vació, se vacía otra vez y su tiempo de último vaciado se cambia al tiempo actual. Si esto se hace en un ciclo mensual, el método requiere 31 cintas de vaciado diarias, una por día, más suficientes cintas para contener un vaciado completo, que se efectúa una vez al mes. También se usan otros esquemas más complejos que emplean menos cintas.

También están en uso métodos automáticos que emplean múltiples discos. Por ejemplo, **la creación de espejos** emplea dos discos. Las escrituras se efectúan en ambos discos, y las lecturas provienen de uno solo. La escritura en el disco espejo se retrasa un poco, efectuándose cuando el sistema está ocioso. Un sistema así puede seguir funcionando en "modo degradado" si un disco falla, y esto permite reemplazar el disco que falló y recuperar los datos sin tener que parar el sistema.

Consistencia del sistema de archivos

Otra área en la que la confiabilidad es importante es la consistencia del sistema de archivos. Muchos sistemas de archivos leen bloques, los modifican y los vuelven a escribir después. Si el sistema se cae antes de que todos los bloques modificados se hayan escrito en disco, el sistema de archivos puede quedar en un estado inconsistente. Este problema se vuelve crítico si algunos de los bloques que no se han escrito contienen nodos-i, entradas de directorio o la lista libre.

Para enfrentar el problema de un sistema de archivos inconsistente, la mayor parte de las computadoras cuentan con un programa de utilería que verifica la consistencia del sistema de archivos. Este programa puede ejecutarse cuando el sistema se arranca, sobre todo después de una caída. La descripción que sigue se refiere al funcionamiento de tal utilería en UNIX y MINIX; otros sistemas tienen algo similar. Estos verificadores del sistema de archivos examinan cada sistema de archivos (disco) con independencia de los demás.

Se pueden realizar dos tipos de verificaciones de consistencia: de bloques y de archivos. Para comprobar la consistencia de los bloques, el programa construye dos tablas, cada una de las cuales contiene un contador para cada bloque, que inicialmente vale 0. Los contadores de la primera

tabla llevan la cuenta de cuántas veces un bloque está presente en un archivo; los contadores de la segunda tabla registran cuántas veces está presente cada bloque en la lista libre (o en el mapa de bits de bloques libres).

A continuación, el programa lee todos los nodos-*i*. Partiendo de un nodo-*i*, es posible construir una lista de todos los números de bloque empleados en el archivo correspondiente. Al leerse cada número de bloque, su contador en la primera tabla se incrementa. Después, el programa examina la lista o mapa de bits de bloques libres, para encontrar todos los bloques que no están en uso. Cada ocurrencia de un bloque en la lista libre hace que su contador en la segunda tabla se incremente.

Si el sistema de archivos es consistente, cada bloque tendrá un 1 ya sea en la primera tabla o en la segunda, como se ilustra en la Fig. 5-18(a). Sin embargo, después de una caída las tablas podrían tener el aspecto de la Fig. 5-18(b), en la que el bloque 2 no ocurre en ninguna de las dos tablas. Este bloque se informará como bloque fallante. Aunque los bloques fallantes no representan un daño real, desperdician espacio y, por tanto, reducen la capacidad del disco. La solución en el caso de haber bloques fallantes es directa: el verificador del sistema de archivos simplemente los agrega a la lista libre.

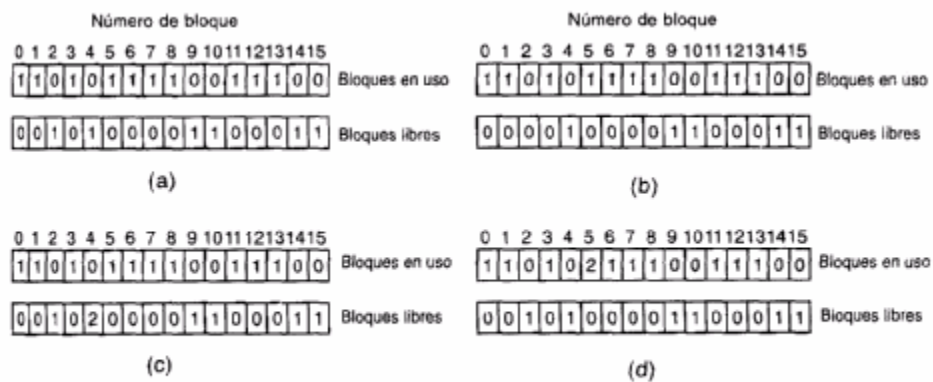


Figura 5-18. Estados del sistema de archivos. (a) Consistente. (b) Bloque faltante. (c) Bloque duplicado en la lista libre. (d) Bloque de datos duplicado.

Otra situación que podría ocurrir es la de la Fig. 5-18(c). Aquí vemos un bloque, el número 4, que ocurre dos veces en la lista libre. (Sólo puede haber duplicados si la lista libre es realmente una lista; si es un mapa de bits esto es imposible.) La solución en este caso también es simple: reconstruir la lista libre.

Lo peor que puede suceder es que el mismo bloque de datos esté presente en dos o más archivos, como se ilustra en la Fig. 5-18(d) con el bloque 5. Si cualquiera de estos archivos se elimina, el bloque 5 se colocará en la lista libre, dando lugar a una situación en la que el mismo bloque está en uso y libre al mismo tiempo. Si se eliminan ambos archivos, el bloque se colocará en la lista libre dos veces.

La acción apropiada para el verificador del sistema de archivos es asignar un bloque libre, copiaren él el contenido del bloque 5, e insertar la copia en uno de los archivos. De este modo, el contenido de información de los archivos no cambiará (aunque casi con toda seguridad estará

revuelto), y la estructura del sistema de archivos al menos será consistente. Se deberá informar del error, para que el usuario pueda inspeccionar los daños.

Además de verificar que cada bloque esté donde debe estar, el verificador del sistema de archivos también revisa el sistema de directorios. En este caso también se usa una tabla de contadores, pero ahora uno por cada archivo, no por cada bloque. El programa parte del directorio raíz y desciende recursivamente el árbol, inspeccionando cada directorio del sistema de archivo; Para cada archivo de cada directorio, el verificador incrementa el contador correspondiente í nodo-i de ese archivo (véase en la Fig. 5-13 la organización de una entrada de directorio).

Una vez que termina la revisión, el verificador tiene una lista, indizada por número de nodo que indica cuántos directorios apuntan a ese nodo-i. Luego, el programa compara estos números con los conteos de enlace almacenados en los nodos-i mismos. En un sistema de archivos consistente, ambos conteos coinciden. Sin embargo, pueden ocurrir dos tipos de errores: el conteo de enlaces del nodo-i puede ser demasiado alto o demasiado bajo.

Si el conteo de enlaces es mayor que el número de entradas de directorio, entonces aunque si borrarán todos los archivos de todos los directorios el conteo seguiría siendo mayor que cero y e nodo-i no se eliminaría. Este error no es grave, pero desperdicia espacio en el disco con archivo: que no están en ningún directorio, así que debe corregirse asignando el valor correcto al conteo de enlaces del nodo-i.

El otro error puede ser catastrófico. Si dos entradas de directorio están enlazadas a un archivo, pero el nodo-i dice que sólo hay una, cuando se elimine cualquiera de las entradas de directorio, el conteo del nodo-i se convertirá en cero. Cuando esto suceda, el sistema de archivos le marcará como desocupado y liberará todos sus bloques. El resultado de esta acción es que uno de los directorios ahora apunta a un nodo-i desocupado, cuyos bloques pronto pueden ser asignados a otros archivos. Una vez más, la solución es hacer que el conteo de enlaces del nodo-i sea igual al número real de entradas de directorio.

Estas dos operaciones, verificar bloques y verificar directorios, a menudo se integran por razones de eficiencia (p. ej., sólo se requiere una pasada por los nodos-i). También pueden realizarse otras verificaciones heurísticas. Por ejemplo, los directorios tienen un formato definido, con números de nodo-i y nombres ASCII. Si un número de nodo-i es mayor que el número de nodos-i que hay en el disco, es señal de que el directorio ha sido dañado.

Además, cada nodo-i tiene un modo, y algunos de estos modos son permitidos pero extraños, como el 0007, que no permite que el propietario ni su grupo tengan acceso, pero sí permite a terceros leer, escribir y ejecutar el archivo. Podría ser útil al menos informar de la existencia de archivos que dan más derechos a terceros que al propietario. Los directorios que tienen más de 1000 entradas también son sospechosos. Los archivos situados en directorios de usuarios, pero que son propiedad del superusuario y tienen encendido el bit SETUID, implican posibles problemas de seguridad. Con un poco de esfuerzo, es posible hacer una lista más o menos larga de situaciones permitidas pero peculiares que valdría la pena informar.

Los párrafos anteriores abordaron el problema de proteger al usuario contra las caídas del sistema. Algunos sistemas de archivos también se preocupan por proteger al usuario contra sí mismo. Si la intención del usuario era teclear

rm *.o

para eliminar todos los archivos que terminan con .o (archivos objeto generados por el compilador), pero accidentalmente teclea `rm *` (adviértase el espacio después del asterisco), `rm` eliminará todos los archivos del directorio actual y después se quejará de que no puede encontrar a .o. En MS-DOS y algunos otros sistemas, cuando un archivo se elimina lo único que sucede es que se enciende un bit en el directorio o nodo-i para marcar ese archivo como eliminado. No se devuelven bloques de disco a la lista libre en tanto no se necesitan realmente. Por tanto, si el usuario descubre el error de inmediato, es posible ejecutar un programa de utilería especial que "deselimina" (es decir, restaura) los archivos eliminados. En WINDOWS 95, los archivos eliminados se colocan en un directorio de reciclado especial, del cual pueden recuperarse posteriormente si es necesario. Desde luego, no se recupera espacio de almacenamiento en tanto los archivos no se borran de dicho directorio especial.

5.3.5 Rendimiento del sistema de archivos

El acceso a un disco es mucho más lento que el acceso a la memoria. La lectura de una palabra de memoria por lo regular toma decenas de nanosegundos. La lectura de un bloque de un disco duro puede tardar 50 microsegundos, lo que implica que es cuatro veces más lenta por palabra de 32 bits, pero a esto deben sumarse de 10 a 20 milisegundos para mover la cabeza de lectura a la pista correcta y luego esperar a que el sector deseado se coloque debajo de la cabeza. Si sólo se necesita una palabra, el acceso a la memoria será del orden de 100 000 veces más rápido que al disco. En vista de esta diferencia en el tiempo de acceso, muchos sistemas de archivos se han diseñado pensando en reducir el número de accesos a disco requeridos.

La técnica más común empleada para reducir los accesos a disco es el **caché de bloques** o el **caché de buffer** (la palabra caché proviene del verbo francés *cacher*, que significa esconder). En este contexto, un caché es una colección de bloques que lógicamente pertenecen al disco pero que se están manteniendo en la memoria por razones de rendimiento.

Se pueden usar diversos algoritmos para administrar el caché, pero uno de los más comunes es inspeccionar todas las solicitudes de lectura para ver si el bloque requerido está en el caché. Si es así, la solicitud de lectura se puede satisfacer sin un acceso a disco. Si el bloque no está en el caché, primero se lee del disco y se coloca en el caché, y luego se copia al lugar donde se necesita. Las solicitudes subsecuentes del mismo bloque se pueden satisfacer desde el caché.

Si es necesario cargar un bloque en el caché y éste está lleno, es preciso deshacerse de un bloque y escribirlo en el disco si ha sido modificado desde que se trajo del disco. Esta situación es muy similar a la paginación, y todos los algoritmos de paginación usuales que vimos en el capítulo 4, como FIFO, segunda oportunidad y LRU, son aplicables. Una diferencia agradable entre la paginación y el manejo de caché es que las referencias al caché son relativamente poco frecuentes, por lo que se hace factible mantener todos los bloques en orden LRU exacto empleando listas enlazadas.

Desafortunadamente, hay un pequeño detalle. Ahora que tenemos una situación en la que es posible usar LRU exacto, resulta que este algoritmo es indeseable. El problema tiene que ver con

las caídas y la consistencia del sistema de archivos que vimos en la sección anterior. Si un bloque crítico, digamos un bloque de nodo-*i*, se coloca en el caché y se modifica, pero no se reescribe en el disco, una caída dejaría al sistema de archivos en un estado inconsistente. Si el bloque de nodo-*i* se coloca al final de la cadena de LRU, puede pasar un buen rato antes de que llegue *i* principio y se reescriba en el disco.

Además, a algunos bloques, como los de doble indirección, casi nunca se hace referencia dos veces dentro de un tiempo corto. Estas consideraciones dan lugar a un esquema LRU modificado teniendo en cuenta dos factores:

1. ¿Es probable que el bloque se necesite pronto otra vez?
2. ¿Es indispensable el bloque para la consistencia del sistema de archivos?

Para poder contestar ambas preguntas, los bloques pueden dividirse en categorías, como bloque de nodo-*i*, bloques de indirección, bloques de directorio, bloques de datos llenos y bloques de datos parcialmente llenos. Los bloques que probablemente no se necesitarán pronto otra vez se colocan al frente, no al final, de la lista de LRU, con objeto de que sus buffers se reutilice rápidamente. Los bloques que podrían necesitarse pronto otra vez, como un bloque parcialmente lleno que se está escribiendo, se colocan al final de la lista, para que estén en ella un buen tiempo.

La segunda pregunta es independiente de la primera. Si el bloque es indispensable para la consistencia del sistema de archivos (básicamente, todo excepto los bloques de datos) y ha sido modificado, se deberá escribir de inmediato en el disco, sin importar en qué extremo de la lista LRU se coloque. Al escribir los bloques críticos rápidamente, reducimos en buena medida la probabilidad de que una caída arruine el sistema de archivos.

Incluso con esta medida para mantener intacta la integridad del sistema de archivos, no es deseable mantener los bloques de datos demasiado tiempo en el caché antes de escribirlos en el disco. Consideremos la situación de una persona que está usando una computadora personal para escribir un disco. Incluso si nuestro escritor le dice periódicamente al editor que grabe en el disco el archivo que está editando, hay una buena probabilidad de que todo estará aún en el caché y nada en el disco. Si el sistema se cae, la estructura del sistema de archivos no estará corrompida, pero se habrá perdido todo un día de trabajo.

No hace falta que ocurra esta situación muchas veces para que tengamos un usuario muy descontento. Los sistemas adoptan dos enfoques para manejar el problema. Lo que hace UNIX tener una llamada al sistema, SYNC, que obliga la escritura inmediata en disco de todos los bloques modificados. Cuando el sistema se inicia, se pone en marcha un programa de segundo plano normalmente llamado `up_date` que da vueltas en un ciclo infinito emitiendo llamadas SYNC y durmiendo durante 30 segundos entre una llamada y otra. Así, nunca se pierden más de 30 segundos a causa de una caída.

Lo que hace MS-DOS es grabar todos los bloques modificados en el disco tan pronto como los modifican. Los caches en los que todos los bloques modificados se escriben en el disco de inmediato se denominan **caches de escritura inmediata** de disco que los de otro tipo. La diferencia entre estos dos enfoques puede verse cuando una grama escribe un bloque de 1 K carácter por carácter. UNIX acumula todos los caracteres en el

caché y escribe el bloque en disco una vez cada 30 segundos, o cuando el bloque se quita del caché. MS-DOS efectúa un acceso a disco por cada carácter que se escribe. Desde luego, la mayor parte de los programas manejan buffers internos, así que normalmente no escriben un carácter, sino una línea o una unidad más grande en cada llamada al sistema WRITE.

Una consecuencia de esta diferencia en la estrategia de caché es que si retiramos un disco (flexible) de un sistema UNIX sin efectuar SYNC casi siempre perderemos datos, y en algunos casos corromperemos también el sistema de archivos. En MS-DOS no hay problema. Se escogieron estas diferentes estrategias porque UNIX se desarrolló en un entorno en el que todos los discos eran duros y no removibles, mientras que MS-DOS se inició en el mundo de los disquetes. Al generalizarse el uso de discos duros, incluso en las microcomputadoras pequeñas, el enfoque de UNIX, al ser más eficiente, definitivamente será el del futuro.

El uso de caché no es la única forma de aumentar el rendimiento de un sistema de archivos. Otra técnica importante es reducir la cantidad de movimiento del brazo del disco colocando cerca unos de otros, de preferencia en el mismo cilindro, bloques a los que probablemente se accederá en secuencia. Cuando se escribe un archivo de salida, el sistema de archivos tiene que asignar los i bloques uno por uno, conforme se necesitan. Si los bloques libres están registrados en un mapa j de bits, y el mapa de bits completo está en la memoria principal, no es difícil escoger un bloque i libre lo más cercano posible al bloque anterior. En el caso de una lista libre, ubicada parcialmente i en disco, es mucho más difícil asignar bloques cercanos unos a otros.

No obstante, incluso con una lista libre, es posible lograr cierto agrupamiento de bloques. El truco consiste en seguir la pista al espacio de almacenamiento en disco no por bloques, sino por grupos de bloques consecutivos. Si una pista consta de 64 sectores de 512 bytes, el sistema podría usar bloques de 1 K (dos sectores) pero asignar espacio de almacenamiento en disco en unidades de dos bloques (cuatro sectores). Esto no es lo mismo que tener bloques de disco de 2K, ya que el caché seguiría usando bloques de 1K y las transferencias de disco también serían de 1K, pero la lectura secuencial de un archivo en un sistema que por lo demás está ocioso reduciría el número de búsquedas a la mitad, mejorando considerablemente el rendimiento.

Una variación del mismo tema es tener en cuenta el posicionamiento rotacional. Al asignar pues, el sistema intenta colocar bloques consecutivos de un archivo en el mismo cilindro, pero intercalados a fin de obtener un rendimiento máximo. Por tanto, si un disco tiene un tiempo de rotación de 16.67 ms y un proceso de usuario tarda unos 4 ms para solicitar y obtener un bloque disco, cada bloque deberá colocarse al menos un cuarto de giro más allá que su predecesor.

Otro cuello de botella de rendimiento en los sistemas que usan nodos- i o algo equivalente es 6 la lectura de incluso un archivo corto requiere dos accesos a disco: uno para el nodo- i y el o para el bloque. En la Fig. 5-19(a) se muestra la colocación usual de los nodos- i . Aquí todos i nodos- i están cerca del principio del disco, así que la distancia media entre un nodo- i y sus bloques es cerca de la mitad del número de cilindros, lo que implica movimientos largos del brazo.

Una forma fácil de mejorar el rendimiento es colocar los nodos- i en la parte media del disco, en principio, reduciendo así a la mitad el movimiento medio del brazo para desplazarse del i al primer bloque. Otra idea, que se muestra en la Fig. 5-19(b), es dividir el disco en grupos todos, cada uno con sus propios nodos- i , bloques y lista libre (McKusick et al., 1984). Al un archivo nuevo se puede escoger cualquier nodo- i , pero se intenta encontrar un bloque

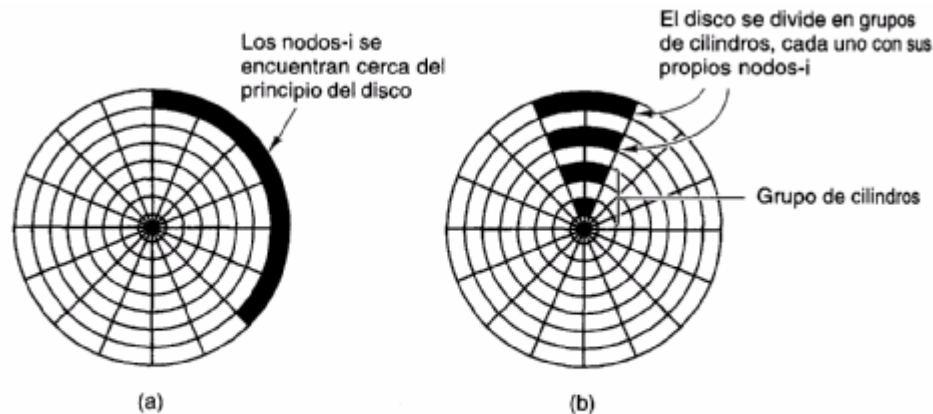


Figura 5-19. (a) Nodos-i colocados al principio del disco. (b) Disco dividido en grupos de cilindros, cada uno con sus propios bloques y nodos-i.

que esté en el mismo grupo de cilindros que el nodo-i. Si no hay uno disponible, se usa un bloque de un cilindro cercano.

5.3.6 Sistemas de archivos estructurados por diario

Los cambios tecnológicos están ejerciendo presión sobre los sistemas de archivos actuales. En particular, las CPU cada vez son más rápidas, los discos cada vez son más grandes y económicos (pero no mucho más rápidos), y el tamaño de las memorias está creciendo exponencialmente. El único parámetro que no está mejorando a pasos agigantados es el tiempo de búsqueda en disco. La combinación de estos factores implica que en muchos sistemas de archivos está apareciendo un cuello de botella del rendimiento. Investigaciones efectuadas en Berkeley intentaron aliviar este problema diseñando un tipo totalmente nuevo de sistema de archivos, LFS (el sistema de archivos estructurado por diario, log-structured file system). En esta sección describiremos brevemente cómo funciona LFS. Si el lector desea un tratamiento más detallado, puede consultar (Rosenblum y Ousterhout, 1991).

La idea en que se basa el diseño LFS es que conforme las CPU se hacen más rápidas y las memorias RAM se hacen más grandes, los caches de disco están aumentando aceleradamente. En consecuencia, ya es posible satisfacer una fracción sustancial de todas las solicitudes de lectura directamente del caché del sistema de archivos, sin tener que acceder al disco. De esta observación se desprende que, en el futuro, la mayor parte de los accesos a disco serán escrituras, y el mecanismo de lectura anticipada que se usa en algunos sistemas de archivos para obtener bloques antes de que se necesiten ya no representará una ganancia significativa en cuanto al rendimiento.

Para empeorar las cosas, en la mayor parte de los sistemas de archivos las escrituras se efectúan en fragmentos muy pequeños. Las escrituras pequeñas son muy ineficientes, ya que una escritura en disco de 50 microsegundos típicamente va precedida por una búsqueda de 10 ms y un retardo rotacional de 6 ms. Con estos parámetros, la eficiencia del disco decae a una fracción de 1%.

A fin de ver de dónde provienen todas estas escrituras pequeñas, consideremos la creación de un archivo nuevo en un sistema UNIX. Para escribir este archivo, es preciso escribir el nodo-i del directorio, el bloque de directorio, el nodo-i del archivo y el archivo mismo. Si bien estas escrituras podrían diferirse, hacerlo expone el sistema de archivos a problemas de consistencia graves en caso de ocurrir una caída antes de llevarse a cabo las escrituras. Por esta razón, las escrituras de nodos-i generalmente se efectúan de inmediato.

Siguiendo este razonamiento, los diseñadores de LFS decidieron reimplementar el sistema de archivos de UNIX de forma tal que se aprovechara todo el ancho de banda del disco, incluso en casos en los que la carga de trabajo consiste en su mayor parte en escrituras aleatorias pequeñas. La idea fundamental consiste en estructurar todo el disco como un diario. Periódicamente, y cuando surge una necesidad especial, todas las escrituras pendientes que se han ido almacenando en la memoria se juntan y se escriben en el disco en forma de un solo segmento contiguo al final del diario. Así, un mismo segmento puede contener nodos-i, bloques de directorio y bloques de datos, todos revueltos. Al principio de cada segmento hay un resumen del segmento, que indica el contenido del mismo. Si puede hacerse que en promedio el segmento ocupe 1 MB, se podrá aprovechar casi todo el ancho de banda del disco.

En este diseño, siguen existiendo nodos-i y tienen la misma estructura que en UNIX, pero ahora están dispersos por todo el diario, en lugar de estar en una posición fija en el disco. No obstante, cuando se localiza un nodo-i, la localización de los bloques se efectúa de la forma acostumbrada. Desde luego, ahora es mucho más difícil encontrar un nodo-i, ya que su dirección no se puede calcular simplemente a partir de su número, como en UNIX. Para poder encontrar los nodos-i, se mantiene un mapa de ellos, indizado por número de nodo-i. La entrada i de este mapa apunta al nodo-i i en el disco. El mapa se mantiene en disco, pero también en caché, así que las partes de uso más intenso están en la memoria casi todo el tiempo.

Resumiendo lo que hemos dicho hasta aquí, todas las escrituras se colocan inicialmente en buffers en la memoria, y periódicamente se escriben en el disco en un solo segmento, al final del diario. Cuando se desea abrir un archivo, se utiliza el mapa para encontrar el nodo-i de ese archivo. Una vez localizado este nodo, se pueden obtener de él las direcciones de los bloques. Todos los bloques mismos están en segmentos en algún lugar del diario.

Si los discos fueran infinitamente grandes, la descripción anterior ya lo habría dicho todo. Sin embargo, los discos reales son finitos, y tarde o temprano el diario ocupará todo el disco, y ya no será posible escribir más segmentos en el diario. Por fortuna, es posible que muchos segmentos existentes tengan bloques que ya no se necesitan. Por ejemplo, si se sobrescribe un archivo, su nodo -i apuntará ahora a los nuevos bloques, pero los antiguos todavía estarán ocupando espacio en segmentos previamente escritos.

A fin de resolver estos dos problemas, LFS cuenta con un hilo limpiador que dedica su tiempo a explorar circularmente el diario y compactarlo. Lo primero que hace el hilo es leer el resumen del primer segmento del diario para ver qué nodos-i y archivos contiene. Luego examina el mapa de nodos-i actual para ver si los nodos-i todavía están vigentes y si los bloques de archivo todavía están en uso. Si no es así, la información correspondiente se desecha. Los nodos-i y bloques que todavía están en uso se pasan a la memoria para ser escritos en el siguiente segmento. A continuación, el segmento original se marca como libre a fin de que el diario pueda usarlo para

datos nuevos. De esta forma, el limpiador avanza por el diario, eliminando segmentos antiguos de la parte de atrás y colocando cualesquier datos aún vigentes que encuentre en la memoria para ser reescritos en el siguiente segmento. Así, el disco es un gran buffer circular, con el hilo escritor agregando nuevos segmentos al frente y el hilo limpiador quitando los viejos de la parte de atrás.

La contabilización aquí no es trivial, ya que cuando un bloque de archivo se escribe en un nuevo segmento es preciso localizar el nodo-i del archivo (en algún lugar del diario), actualizarlo y colocarlo en la memoria para escribirlo en el siguiente segmento. Después, debe actualizarse el mapa de nodos-i de modo que apunte a la nueva copia. No obstante, toda esta administración es factible, y los resultados de rendimiento demuestran que la complejidad vale la pena. Las mediciones que se presentan en los artículos antes citados indican que LFS tiene un rendimiento mejor en un orden de magnitud que el de UNIX cuando las escrituras son pequeñas, y un rendimiento igual o mejor que el de UNIX en el caso de escrituras grandes y lecturas.

5.4 SEGURIDAD

Los sistemas de archivos a menudo contienen información que es muy valiosa para sus usuarios. Por tanto, la protección de esta información contra el uso no autorizado es una función importante de todos los sistemas de archivos. En las siguientes secciones examinaremos diversos problemas relacionados con la seguridad y la protección. Estas cuestiones se aplican tanto a los sistemas tiempo compartido como a las redes de computadoras personales conectadas a servidores compartidos a través de redes de área local.

5.4.1 El entorno de seguridad

Los términos "seguridad" y "protección" con frecuencia se usan indistintamente. No obstante, suele ser útil hacer una distinción entre los problemas generales que debemos resolver para asegurar que los archivos no sean leídos ni modificados por personas no autorizadas, lo que incluye cuestiones técnicas, gerenciales, legales y políticas, por un lado, y los mecanismos específicos del sistema operativo que proporcionan seguridad, por el otro. A fin de evitar confusiones, usaremos el término seguridad para referirnos al problema global, y el término mecanismos de protección para referirnos a los mecanismos específicos del sistema operativo que sirven para salvaguardar la información en la computadora. Sin embargo, la frontera entre las dos cosas no está bien definida. Primero examinaremos la seguridad; más adelante nos ocuparemos de la protección.

La seguridad tiene muchas facetas. Dos de las más importantes son la pérdida de datos y los intrusos. Algunas de las causas comunes de la pérdida de datos son:

1. Actos divinos: incendios, inundaciones, terremotos, guerras, motines o ratas que mordisquean cintas o disquetes.
2. Errores de hardware o software: fallas de CPU, discos o cintas ilegibles, errores de telecomunicación, errores en programas.
3. Errores humanos: captura incorrecta de datos, montar la cinta o disco equivocado,

ejecutar un programa indebido, perder un disco o una cinta, o alguna otra equivocación.

La mayor parte de estos problemas puede superarse manteniendo respaldos adecuados, de preferencia lejos de los datos originales.

Un problema más interesante es qué hacer respecto a los intrusos. Hay dos clases de estos especímenes. Los intrusos pasivos sólo desean leer archivos que no están autorizados para leer. Los intrusos activos tienen peores intenciones: quieren efectuar cambios no autorizados a los datos. Al diseñar un sistema de modo que sea seguro frente a los intrusos, es importante tener presente la clase de intruso contra la que se está tratando de proteger el sistema. He aquí algunas categorías comunes:

1. Curioso casual por parte de usuarios no técnicos. Muchas personas tienen en su escritorio terminales conectadas a sistemas de tiempo compartido o computadoras personales conectadas a redes y, al ser la naturaleza humana como es, algunas de ellas leerán el correo electrónico y otros archivos de otras personas si no se les ponen barreras. En la mayor parte de los sistemas UNIX, por ejemplo, todos los archivos están abiertos al público por omisión.

2. Intromisión por parte de gente de adentro. Los estudiantes, programadores de sistemas, operadores y demás personal técnico con frecuencia consideran como un reto personal violar la seguridad del sistema de computadora local. Es común que estas personas estén altamente capacitadas y dispuestas a dedicar una cantidad sustancial de tiempo a esta labor.

3. Intento decidido por hacer dinero. Algunos programadores bancarios han intentado introducirse en un sistema bancario para robar. Los ardides han variado desde modificar el software para truncar en lugar de redondear los intereses, guardándose la fracción de centavo para sí, hasta extraer fondos de cuentas que no se han usado en varios años, hasta chantaje ("Páguenme o destruiré todos los registros del banco").

4. Espionaje comercial o militar. Por espionaje se entiende el intento serio y bien financiado, por parte de un competidor o un país extranjero, por robar programas, secretos comerciales, patentes, tecnología, diseños de circuitos, planes de marketing, etc. En muchos casos este intento implica intervención de líneas o incluso erigir antenas dirigidas hacia la computadora a fin de captar su radiación electromagnética.

Debe ser obvio que tratar de evitar que un gobierno extranjero hostil robe secretos militares es una cuestión muy diferente a tratar de impedir que los estudiantes inserten un "mensaje del día" humorístico en el sistema. La cantidad de esfuerzo que se invierte en la seguridad y la protección naturalmente dependerá de quién se piensa que es el enemigo.

Otro aspecto del problema de la seguridad es la confidencialidad: proteger a los usuarios contra el uso indebido de la información referente a ellos. Aquí intervienen muchas cuestiones legales y morales. ¿Se justifica que el gobierno compile expedientes de todo mundo con el objeto de atrapar a quienes no cumplen con sus impuestos u otras obligaciones? ¿Necesita la policía

poder consultar cualquier dato sobre cualquier persona para poner un alto al crimen organizado? ¿Tienen derechos los patrones y las compañías de seguros? ¿Qué sucede cuando estos derechos chocan contra los derechos individuales? Todas estas cuestiones son extremadamente importantes pero rebasan el alcance del presente libro.

5.4.2 Fallas de seguridad famosas

Así como la industria del transporte tiene el Titanic y el Hindenburg, los expertos en seguridad de computadoras tienen algunas cosas que preferirían olvidar. En esta sección examinaremos algunos problemas de seguridad interesantes que han ocurrido en tres sistemas operativos distintos: UNIX, TENEX y os/360.

La utilidad de UNIX `lpr`, que imprime un archivo en la impresora de líneas, tiene una opción para eliminar el archivo una vez que se ha impreso. En las primeras versiones de UNIX cualquiera podía usar `lpr` para imprimir el archivo de contraseñas y luego hacer que el sistema lo eliminara.

Otra forma de introducirse en UNIX era enlazar un archivo llamado `core` situado en el directorio de trabajo con el archivo de contraseñas. A continuación, el intruso forzaba un vaciado de núcleo de un programa `SETUID`, mismo que el sistema escribía en el archivo `core`, esto es, encima del archivo de contraseñas. De este modo, un usuario podía reemplazar el archivo de contraseñas por uno que contenía unas cuantas cadenas escogidas por él (p. ej., argumentos de comandos).

Otro defecto sutil de UNIX tenía que ver con el comando

```
mkdir foo
```

`Mkdir`, que era un programa `SETUID` propiedad de la raíz, primero creaba el nodo `i` para el directorio `foo` con la llamada al sistema `MKNOD` y luego cambiaba (con la llamada `CHOWN`) el propietario de `foo` de su `uid` efectivo (esto es, la raíz) a su `uid` real (el `uid` del usuario). Si el sistema era lento, a veces el usuario podía eliminar rápidamente el nodo `i` del directorio y crear un enlace con el archivo de contraseñas bajo el nombre `foo` después de la llamada `MKNOD` pero antes de `CHOWN`. Cuando `mkdir` ejecutaba `CHOWN`, convertía al usuario en propietario del archivo de contraseñas. Si se colocaban los comandos necesarios en un guión de shell, se podían intentar una y otra vez hasta que el truco funcionara.

El sistema operativo TENEX solía ser muy popular en las computadoras DEC-10. Este sistema ya no se usa, pero vivirá eternamente en los anales de la seguridad de computadoras gracias al siguiente error de diseño. TENEX apoyaba la paginación. Con objeto de que los usuarios pudieran vigilar el comportamiento de sus programas, era posible ordenar al sistema que invocara una función de usuario cada vez que ocurriera una falla de página.

TENEX también usaba contraseñas para proteger los archivos. Cuando un programa quería acceder a un archivo, tenía que presentar la contraseña apropiada. El sistema operativo verificaba las contraseñas carácter por carácter, deteniéndose tan pronto como se daba cuenta de que la contraseña no era la correcta. Para introducirse en TENEX, un intruso colocaba cuidadosamente una contraseña como se muestra en la Fig. 5-20(a), con el primer carácter al final de una página y el resto al principio de la siguiente página.

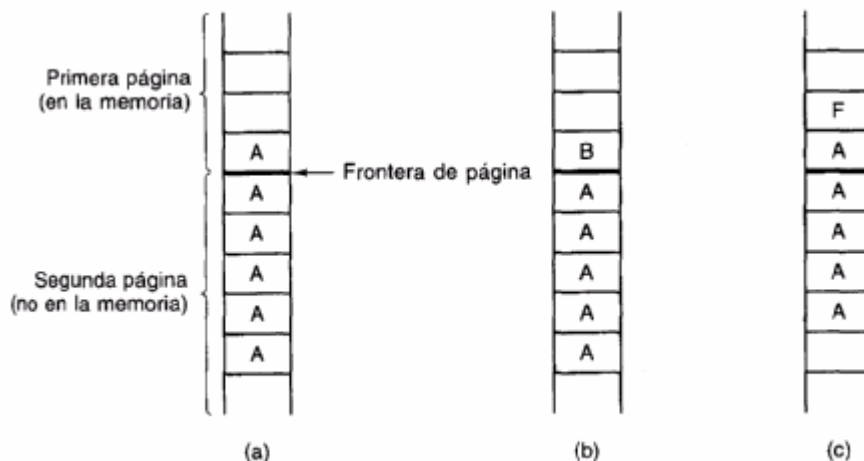


Figura 5-20. El problema de contraseña TENEX.

El siguiente paso era asegurarse de que la siguiente página no estuviera en la memoria. Esto se hacía, por ejemplo, haciendo referencia a tantas páginas que la segunda página tenía que ser desalojada para que aquéllas pudieran caber. Ahora el programa trataba de abrir el archivo de la víctima, usando la contraseña cuidadosamente alineada. Si el primer carácter de la contraseña real era distinto de A, el sistema dejaba de verificarla en el primer carácter y regresaba con un mensaje de ILLEGAL PASSWORD (contraseña no válida). En cambio, si la contraseña real comenzaba con A, el sistema seguía leyendo y generaba una falla de página, de lo cual se informaba al intruso.

Si la contraseña no comenzaba con A, el intruso cambiaba la contraseña a la de la Fig. 5-20(b) y repetía todo el proceso para ver si comenzaba con B. Se requerían como máximo 128 intentos para probar todo el conjunto de caracteres ASCII y así determinar el primer carácter.

Supongamos que el primer carácter era F. La disposición de memoria de la Fig. 5-20(c) permitía al intruso probar cadenas de la forma FA, FB, etc. Con esta estrategia, se requerían como mucho $128n$ intentos para adivinar una contraseña ASCII de n caracteres, en lugar de 128^n .

Nuestro último ejemplo de defecto se refiere a os/360. La descripción que sigue se ha simplificado un poco pero conserva la esencia del defecto. En este sistema era posible iniciar una lectura de cinta y luego seguir realizando cálculos mientras la unidad de cinta transfería datos al espacio del usuario. El truco aquí consistía en iniciar con cuidado una lectura de cinta y luego efectuar una llamada al sistema que requería una estructura de datos de usuario, por ejemplo, un retuvo para leer y su contraseña.

Lo primero que hacía el sistema operativo era verificar que la contraseña fuera la correcta ir el archivo especificado, y luego regresaba y leía el nombre del archivo otra vez para efectuar el acceso real (podía haber guardado el nombre internamente, pero no lo hacía). Desafortunadamente, justo antes de que el sistema tratara de obtener el nombre del archivo por segunda vez, la unidad de cinta lo sobrescribía. A continuación, el sistema leía el nuevo archivo, para el cual no se había presentado contraseña. Se requería práctica para sincronizar bien las acciones, pero no era cosa del otro mundo. Además, si hay algo que las computadoras hacen bien es repetir la misma operación una y otra vez, ad nauseam.

Además de estos ejemplos, ha habido muchos otros problemas de seguridad y ataques en el pasado. Uno que ha aparecido en muchos contextos es el caballo de Troya, en el que un programa al parecer inocente que se distribuye ampliamente también realiza alguna función inesperada e indeseable, como robar datos y enviarlos por correo electrónico a un sitio distante donde posteriormente se recogen.

Otro problema de seguridad en estos tiempos de inseguridad en el empleo es la bomba lógica. Este dispositivo es un fragmento de código escrito por uno de los programadores empleados actualmente por una compañía e insertado secretamente en el sistema operativo de producción. En tanto el programador le suministra su contraseña diaria, la bomba no hace nada. Sin embargo, si el programador es repentinamente despedido y retirado físicamente del lugar sin advertencia, no suministrará la contraseña a la bomba el día siguiente, y la bomba se activará.

Tal activación podría incluir borrar el disco, borrar archivos al azar, efectuar cuidadosamente cambios difíciles de detectar a programas clave, o cifrar archivos indispensables. En este último caso, la compañía enfrenta la difícil decisión de llamar a la policía (que podría o no redundar en una condena judicial muchos meses después) o ceder ante el chantaje y recontratar al ex-programador como "consultor" por una suma astronómica para que corrija el problema (y abrigar la esperanza de que no plante más bombas lógicas mientras lo hace).]

Tal vez la violación de seguridad de computadoras más grande de todos los tiempos se inició la noche del 2 de noviembre de 1988 cuando un estudiante de posgrado de Cornell, Robert Tappan Morris, liberó un programa gusano en la Internet que causó la caída de miles de máquinas en todo el mundo.

El gusano consistía en dos programas, el autoarranque y el gusano propiamente dicho. El autoarranque comprendía 99 líneas escritas en C y se llamaba ll.c. Este programa se compilaba y ejecutaba en el sistema objetivo del ataque. Una vez que estaba funcionando, el autoarranque se conectaba a la máquina de la que provenía, cargaba en la máquina objetivo el gusano principal y lo ejecutaba. Después de tomar ciertas medidas para ocultar su existencia, el gusano examinaba la tabla de enrutamiento de su nuevo anfitrión para ver con qué máquinas estaba conectado éste y trataba de propagar el autoarranque a esas máquinas.

Una vez establecido en una máquina, el gusano trataba de romper las contraseñas de los usuarios, Morris no tuvo que investigar mucho para averiguar cómo podía hacer esto. Lo único que tuvo que hacer fue pedir a su padre, un experto en seguridad de la National Security Agency, el organismo encargado de descifrar códigos en el máximo nivel de secreto del gobierno estadounidense, una reimpresión del artículo clásico sobre el tema que él (su padre) y Ken Thompson habían escrito diez años atrás en Bell Labs (Morris y Thompson, 1979). Cada contraseña rota permitía al gusano iniciar una sesión en todas las máquinas en las que el propietario de la contraseña tenía cuentas.

Morris fue capturado cuando uno de sus amigos habló con el reportero de computación del New York Times, John Markoff, y trató de convencerlo de que el incidente había sido un accidente, de que el gusano era inofensivo y de que el autor se arrepentía de lo que había hecho. El día siguiente la historia apareció en primera plana, relegando a segundo término la elección presidencial que se iba a celebrar tres días después. Morris fue juzgado y condenado en una corte federal, sentenciándosele a pagar una multa de 10 000 dólares, tres años de libertad condicional y 400 horas de servicio a la comunidad. Es probable que sus costos legales hayan excedido los 150 000 dólares.

Esta sentencia generó muchas controversias. Muchos miembros de la comunidad de la computación pensaban que Morris era un brillante estudiante de posgrado cuya travesura inofensiva se había salido fuera de control. Ningún aspecto del gusano sugirió que Morris estaba tratando de robar o dañar nada. Otros consideraron que se trataba de un delincuente peligroso que debía de haber ido a la cárcel.

Un efecto permanente de este incidente fue el establecimiento del CERT (Equipo de Respuesta a Emergencias en Computadoras) que ofrece un sitio central donde se puede informar de intentos de ingreso indebido, y un grupo de expertos para analizar problemas de seguridad y diseñar remedios. Si bien esta acción fue sin duda un paso hacia adelante, tiene su lado adverso. El CERT recaba información acerca de los defectos de los sistemas que pueden atacarse y cómo corregirlos. Necesariamente, el CERT circula ampliamente esta información a miles de administradores de sistemas de la Internet, lo que implica que "los malos" también pueden obtenerla y aprovechar los puntos débiles en las horas (o incluso días) que inevitablemente transcurren antes de subsanarse.

5.4.3 Ataques genéricos contra la seguridad

Los defectos que hemos descrito han sido corregidos, pero los sistemas operativos ordinarios siguen teniendo más agujeros que un cedazo. La forma usual de probar la seguridad de un sistema consiste en contratar un grupo de expertos, denominados equipos tigre o equipos de penetración, para ver si pueden violarla. Hebbard et al. (1980) intentó lo mismo con estudiantes de posgrado. A lo largo de los años, estos equipos de penetración han descubierto varias áreas en las que los sistemas probablemente tienen puntos débiles. A continuación listamos algunos de los ataques más comunes que suelen tener éxito. Al diseñar un sistema, asegúrese de que puede resistir ataques como éstos.

1. Solicite páginas de memoria, espacio en disco o cintas y simplemente léalas. Muchos sistemas no las borran antes de asignarlas, y pueden estar llenas de información interesante escrita por su propietario anterior.

2. Intente llamadas al sistema no válidas, o llamadas al sistema válidas con parámetros no válidos, o incluso llamadas al sistema válidas con parámetros válidos pero poco razonables. Muchos sistemas pueden confundirse fácilmente.

3. Inicie una sesión y luego oprima DEL, RUBOUT o BREAK a la mitad de la secuencia de inicio de sesión. En algunos sistemas, el programa de verificación de contraseñas será terminado y la sesión se iniciará con éxito.

4. Trate de modificar estructuras complejas del sistema operativo mantenidas en el espacio de usuario (si las hay). En algunos sistemas (sobre todo en macrocomputadoras), para abrir un archivo, el programa construye una estructura de datos grande que contiene el nombre del archivo y muchos otros parámetros y la pasa al sistema. Mientras el archivo se lee y escribe, el sistema a veces actualiza la estructura misma. La modificación de estos campos puede dar al traste con la seguridad.

5. Burle al usuario escribiendo un programa que escriba "login:" en la pantalla y luego desaparezca. Muchos usuarios se sentarán frente a la terminal y proporcionarán sin reparo su nombre de inicio de sesión y su contraseña, que el programa registrará cuidadosamente para su malvado amo.

6. Busque manuales que adiestren a "No hacer X". Intente tantas variaciones de X como le sean posibles.

7. Convenza a un programador del sistema que modifique éste de modo que pase por alto ciertas verificaciones de seguridad cruciales para cualquier usuario que proporcione su nombre de inicio de sesión. Este ataque se conoce como trampa (trapdoor).

8. Si falla todo lo demás, el invasor podría buscar al secretario del director del centro de cómputo y ofrecerle un soborno sustancial. Es probable que el secretario tenga acceso a todo tipo de información útil, y generalmente recibe un salario bajo. No subestime los problemas causados por el personal.

Éstos y otros ataques se analizan en Linde (1975).

Virus

Una categoría de ataque especial es el virus de computadora, que se ha convertido en un problema importante para muchos usuarios. Un virus es un fragmento de programa que se anexa a un programa legítimo con la intención de infectar otros programas. La única diferencia respecto a un gusano es que el virus "se monta" en un programa existente, en tanto que el gusano es un programa completo por sí solo. Tanto los virus como los gusanos intentan propagarse y pueden causar daños graves.

Un virus representativo funciona como sigue. La persona que escribe el virus primero produce un programa nuevo útil, que a menudo es un juego para MS-DOS. Este programa contiene código del virus oculto en su interior. A continuación el juego se carga en un sistema de tablero de boletines público o se ofrece gratuitamente o por un precio módico en un disquete. Luego se hace publicidad al programa, y la gente comienza a descargarlo y usarlo. La construcción de un virus no es sencilla, así que las personas que lo llevan a cabo siempre son brillantes, y la calidad del juego u otro programa a menudo es excelente.

Cuando se pone en marcha el programa, de inmediato comienza a examinar todos los programas binarios del disco duro para ver si ya están infectados. Cuando se encuentra un programa no infectado, éste queda infectado anexando el código del virus al final del archivo, y sustituyendo la primera instrucción por un salto al virus. Cuando el código del virus termina de ejecutarse ejecuta la instrucción que antes era la primera y luego salta a la segunda instrucción. De esta forma, cada vez que se ejecute un programa infectado, tratará de infectar más programas.

Además de infectar otros programas, el virus puede hacer otras cosas, como borrar, modificar o cifrar archivos. Un virus llegó a exhibir un mensaje de extorsión en la pantalla, exigiendo al usuario enviar 500 dólares en efectivo a un apartado postal en Panamá o resignarse a la pérdida permanente de sus datos y a daños al hardware.

También es posible que un virus infecte el sector de arranque del disco duro, haciendo imposible arrancar la computadora. Un virus así podría pedir una contraseña, que el escritor del virus podría ofrecer a cambio de unos cuantos billetes de baja denominación sin marcas.

Los problemas de virus son más fáciles de prevenir que de curar. El proceder más seguro consiste en comprar sólo software debidamente empacado en establecimientos respetables. Cargar software gratuito de los tableros de boletines u obtener copias pirata en disquetes es buscar problemas. Existen paquetes comerciales antivirus, pero algunos de ellos sólo buscan virus conocidos específicos.

Una estrategia más general consiste en formatear totalmente el disco duro, incluido el sector de arranque. A continuación, se instala todo el software confiable y se calcula una suma de verificación para cada archivo. No importa qué algoritmo se use, en tanto tenga suficientes bits (al menos 32). Luego debe almacenarse la lista de pares (archivo, suma de verificación) en un lugar seguro, ya sea fuera de línea en un disco flexible o en línea, pero cifrado. A partir de ese momento, cada vez que el sistema se arranque, se deberán recalcular todas las sumas de verificación y compararse con la lista segura de sumas originales. Cualquier archivo cuya suma actual difiera de la original puede estar infectado. Si bien esta estrategia no impide la infección, al menos permite detectarla en una etapa temprana.

Puede hacerse más difícil la infección si se impide que los usuarios ordinarios escriban en el directorio donde residen los programas binarios. Esta técnica obstaculiza la infección de otros archivos binarios por parte del virus. Aunque esto es factible en UNIX, no puede aplicarse a MS- DOS porque en este sistema no es posible restringir la escritura de directorios.

5.4.4 Principios de diseño para la seguridad

Los virus ocurren en su mayor parte en los sistemas de escritorio. En los sistemas más grandes se presentan otros problemas y se requieren otros métodos para enfrentarlos. Saitzer y Schroeder (1975) han identificado varios principios generales que pueden servir como guía para diseñar sistemas seguros. A continuación resumiremos sus ideas, que se basaron en la experiencia con MULTICS.

Primero, el diseño del sistema debe ser público. Suponer que el intruso no sabe cómo funciona el sistema sólo hace que los diseñadores se engañen a sí mismos.

Segundo, la política por omisión debe ser no otorgar acceso. Los errores en los que se rehúsa un acceso legítimo se informarán con mucha mayor rapidez que los errores en los que se permite un acceso no autorizado.

Tercero, se debe verificar la vigencia de la autorización. El sistema no debe comprobar si se tiene permiso, determinar que el acceso está permitido y luego guardar esta información para uso subsecuente. Muchos sistemas verifican el permiso cuando se abre un archivo, pero no después. Esto implica que un usuario que abre un archivo y lo mantiene unido durante semanas seguirá teniendo acceso, incluso si el propietario ha modificado ya la protección del archivo.

Cuarto, debe darse a cada proceso el mínimo privilegio posible. Si un editor sólo tiene autorización para acceder al archivo que se va a modificar (y que se especifica cuando se invoca el

editor), los editores con caballos de Troya no podrán causar mucho daño. Este principio implica un esquema de protección de grano fino. Trataremos tales esquemas más adelante en este capítulo.

Quinto, el mecanismo de protección debe ser sencillo, uniforme, y estar incorporado en las capas más bajas del sistema. Tratar de añadir seguridad a un sistema inseguro existente es casi imposible; la seguridad, al igual que la exactitud, no es una característica que se pueda agregar.

Sexto, el esquema escogido debe ser psicológicamente aceptable. Si los usuarios piensan que proteger sus archivos implica demasiado trabajo, simplemente no lo harán. Sin embargo, se quejarán a voz en cuello si algo sale mal. Las respuestas del tipo "usted fue el culpable" no suelen ser bien recibidas.

5.4.5 Verificación de autenticidad de usuarios

Muchos esquemas de protección se basan en el supuesto de que el sistema conoce la identidad de cada usuario. El problema de identificar los usuarios cuando inician una sesión se denomina verificación de autenticidad de usuarios. La mayor parte de los métodos de verificación de autenticidad se basan en identificar algo que el usuario conoce, tiene o es.

Contraseñas

La forma de verificación de autenticidad más ampliamente utilizada es pedir al usuario que teclee una contraseña. La protección mediante contraseña es fácil de entender y de implementar. En UNIX el esquema funciona como sigue. El programa de inicio de sesión pide al usuario que teclee su nombre y contraseña. De inmediato, la contraseña se cifra. Luego, el programa de inicio de sesión lee el archivo de contraseñas, que es una serie de líneas ASCII, una por usuario, hasta encontrar la que contiene el nombre de inicio de sesión del usuario. Si la contraseña (cifrada) contenida en esta línea concuerda con la contraseña cifrada que se acaba de calcular, se permite el inicio de la sesión; de lo contrario, se rechaza.

La verificación de autenticidad de las contraseñas es fácil de vencer. Es frecuente leer acerca de grupos de estudiantes de preparatoria, o incluso de secundaria, que, con la ayuda de sus computadoras caseras, se introducen en algún sistema de secreto máximo, propiedad de una corporación gigantesca o una dependencia del gobierno. Prácticamente en todos los casos la intrusión se efectúa adivinando una combinación de nombre de usuario y contraseña.

Aunque se han efectuado estudios más recientes (p. ej., Klein, 1990), el trabajo clásico sobre seguridad de contraseñas sigue siendo el realizado por Morris y Thompson (1979) sobre sistemas UNIX. Ellos compilaron una lista de probables contraseñas: nombres de pila y apellidos, nombre de calle o de ciudad, palabras de un diccionario de tamaño moderado (también palabras escritas al revés), números de placas de automóvil, y cadenas cortas de caracteres aleatorios.

A continuación, ellos cifraron todas estas claves empleando el algoritmo de cifrado de contraseñas conocido y verificaron si alguna de las contraseñas cifradas coincidía con una entrada de su lista. Más del 86% de todas las contraseñas resultaron estar en su lista.

Si todas las contraseñas consistieran en siete caracteres escogidos al azar de entre los 95 caracteres ASCII imprimibles, el espacio de búsqueda es de 95⁷, o sea, cerca de 7 x 10¹³. A razón

de 1000 cifrados por segundo, se requerirían 2000 años para construir la lista contra la cual se cotejaría el archivo de contraseñas. Además, la lista llenaría 20 millones de cintas magnéticas. Aun el requisito de que las contraseñas contengan por lo menos una letra minúscula, una mayúscula y un carácter especial y tengan por lo menos siete u ocho caracteres de longitud representaría una mejora considerable respecto a dejar que los usuarios escojan a su antojo una contraseña.

Incluso si se considera políticamente imposible exigir que los usuarios escojan contraseñas razonables, Morris y Thompson han descrito una técnica que hace que su propio ataque (cifrar una gran cantidad de contraseñas por adelantado) sea casi inútil. Su idea es asociar un número aleatorio de n bits a cada contraseña. El número aleatorio se cambia cada vez que se cambia la contraseña, y se almacena en el archivo de contraseñas sin cifrarse, de modo que todo mundo pueda leerlo. En vez de almacenar la contraseña cifrada en el archivo de contraseñas, la contraseña y el número aleatorio primero se concatenan y luego se cifran juntos. Este resultado cifrado se almacena en el archivo de contraseñas.

Consideremos ahora las implicaciones para un intruso que desea construir una lista de contraseñas probables, cifrarlas, y guardar el resultado en un archivo ordenado, /, con objeto de poder buscar las contraseñas cifradas rápidamente. Si un intruso sospecha que Marilyn podría ser una contraseña, no basta ya con cifrar Marilyn y colocar el resultado en/; es necesario cifrar 2^n cadenas, como MarilynOOOO, MarilynOOO1, Marilyn.0002, etc., y colocarlas todas en/. Esta técnica aumenta el tamaño de/en 2^n . UNIX utiliza este método con $n = 12$. El método se conoce como la acción de saltar el archivo de contraseñas. Algunas versiones de UNIX hacen que el archivo de contraseñas mismo no pueda leerse, y proporcionan un programa que consulta entradas cuando se le solicita, incorporando un pequeño retardo suficiente para reducir drásticamente la rapidez con que puede actuar cualquier atacante.

Si bien este método ofrece protección contra intrusos que tratan de precalcular una lista larga de contraseñas cifradas, no hace mucho para proteger a un usuario llamado David cuya contraseña también sea David. Una forma de fomentar la selección de mejores contraseñas por parte de los usuarios es hacer que la computadora los asesore. Algunas computadoras tienen un programa que genera palabras aleatorias sin significado pero fáciles de pronunciar, como mamuseo, raflije o trupimpa que se pueden usar como contraseña (de preferencia incorporando mayúsculas y wacteres especiales).

Otras computadoras obligan a los usuarios a cambiar sus contraseñas con regularidad, a fin e limitar los daños en caso de una fuga de contraseñas. La forma más extrema de este enfoque es la contraseña de una sola vez. Cuando se usa este tipo de contraseñas, el usuario recibe un libro que contiene una lista de contraseñas. En cada inicio de sesión se usa la siguiente contraseña de lá lista. Si un intruso llega a descubrir una contraseña, de nada le servirá, ya que en la siguiente ocasión debe usarse una contraseña distinta. Se sugiere al usuario tratar de no perder el libro de contraseñas.

Huelga decir que, mientras se está tecleando una contraseña, la computadora no debe exhibir los caracteres tecleados, a fin de protegerlos de ojos curiosos en las cercanías de la terminal. Lo que es menos obvio es que las contraseñas nunca deben almacenarse en la computadora en forma no cifrada. Además, ni siquiera los administradores del centro de cómputo deben poseer copias no cifradas. Mantener contraseñas no cifradas en cualquier sitio es buscarse problemas.

Una variación del concepto de contraseña es hacer que cada usuario nuevo proporcione una larga lista de preguntas y respuestas que después se guardan en la computadora en forma cifrada. Las preguntas deben ser de tal naturaleza que el usuario no necesite anotarlas. Por ejemplo,

1. ¿Quién es el hermano de Marines?
2. ¿En qué calle estaba mi escuela primaria?
3. ¿Qué clase impartía la Sra. Fajardo?

Al iniciarse la sesión, la computadora hace una de estas preguntas al azar y verifica la respuesta.

Otra variación es la de reto-respuesta. Cuando se emplea este enfoque, la persona escoge un algoritmo al inscribirse como usuario, digamos x2. Cuando se inicia una sesión, la computadora teclea un argumento, digamos 7, en cuyo caso el usuario teclea 49. El algoritmo puede ser diferente en la mañana y en la tarde, en diferentes días de la semana, desde diferentes terminales, etcétera.

Identificación física

Un enfoque de autorización totalmente distinto es verificar si el usuario tiene algún objeto, normalmente una tarjeta de plástico con una tira magnética. La tarjeta se inserta en la terminal, la cual entonces determina quién es el dueño. Este método se puede combinar con una contraseña, de modo que el usuario sólo pueda iniciar una sesión si (1) tiene la tarjeta y (2) conoce la contraseña. Las máquinas de entrega de efectivo automáticas suelen funcionar de esta manera.

Otra estrategia consiste en medir características físicas que sean difíciles de falsificar. Por ejemplo, un lector de huellas digitales o de patrón de voz en la terminal podría verificar la identidad del usuario. (La búsqueda se agiliza si el usuario le dice a la computadora quién es, en lugar de hacer que la computadora compare la huella digital con toda la base de datos.) El reconocimiento visual directo todavía no es factible pero algún día podría serlo.

Otra técnica es el análisis de firmas. El usuario firma su nombre con una pluma especial conectada a la terminal, y la computadora la compara con un espécimen conocido almacenado en línea. Mejor aún es no comparar la firma, sino comparar los movimientos de la pluma mientras se firma. Un buen falsificador podría ser capaz de copiar la firma, pero no sabrá en qué orden se efectuaron exactamente los trazos.

El análisis de la longitud de los dedos resulta sorprendentemente práctico. Cuando se emplea esto, cada terminal tiene un dispositivo como el de la Fig. 5-21. El usuario inserta su mano en él, y se mide la longitud de todos sus dedos y se coteja con la base de datos.

Podríamos seguir con más y más ejemplos, pero bastarán otros dos para dejar asentado un punto importante. Los gatos y otros animales marcan su territorio orinando a lo largo de superímetro. Al parecer, los gatos pueden identificarse unos a otros de esta manera. Supónganla) que alguien inventa un dispositivo pequeño capaz de efectuar un análisis de orina instantáneo, con lo cual se lograría una identificación infalsificable. Cada terminal podría equiparse con uno de estos dispositivos, junto con un letrero discreto que dijera "Para iniciar una sesión, sírvase depo-

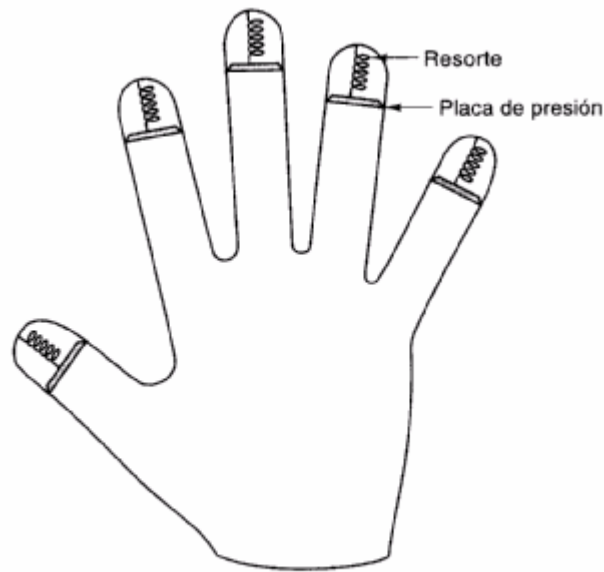


Figura 5-21. Dispositivo para medir la longitud de los dedos.

sitar una muestra aquí". Éste podría ser un sistema inviolable, pero probablemente enfrentaría un problema grave de falta de aceptación por parte de los usuarios.

Lo mismo podría decirse de un sistema consistente en una chinche o chincheta y un pequeño espectrógrafo. Se pediría al usuario que oprimiera su pulgar contra la chincheta, extrayendo así una gota de sangre para análisis espectrográfico. Lo que tratamos de decir es que cualquier esquema de verificación de autenticidad debe ser psicológicamente aceptable para la comunidad de usuarios. Las mediciones de longitud de los dedos probablemente no causarán problemas, pero incluso algo tan poco intrusivo como almacenar huellas digitales en línea podría ser inaceptable para muchas personas.

Medidas preventivas

Las instalaciones de computadoras que toman realmente en serio la seguridad, cosa que con frecuencia sucede al día siguiente de que un intruso se metiera en el sistema, causando daños importantes, suelen adoptar medidas para dificultar mucho más el ingreso no autorizado. Por ejemplo podría permitirse a cada usuario iniciar una sesión sólo desde una terminal específica, y solo durante ciertos días de la semana y ciertas horas del día.

Podría hacerse que las líneas telefónicas de marcado para ingreso funcionen como sigue. cualquiera puede marcar e iniciar una sesión, pero de inmediato el sistema rompe la conexión y llama al usuario a un número convenido con anterioridad. Esta medida implica que un intruso no puede simplemente tratar de introducirse en el sistema desde cualquier línea telefónica; sólo se

puede usar la línea telefónica (de la casa) del usuario. En cualquier caso, sea que llame de vuelta al usuario o no, el sistema debe tardar por lo menos 10 segundos en verificar cualquier contraseña que se introduzca por una línea de marcado, y deberá alargar este lapso después de varios intentos fallidos de iniciar la sesión, a fin de reducir la rapidez con que los intrusos pueden efectuar sus intentos. Después de tres intentos de iniciar la sesión sin éxito, la línea deberá desconectarse durante 10 minutos y notificarse al personal de seguridad.

Todos los inicios de sesión deben registrarse. Cuando un usuario inicia una sesión, el sistema debe informar el tiempo y la terminal del inicio previo, con objeto de que él pueda detectar posibles intrusiones.

El siguiente nivel hacia arriba es poner trampas con carnada para atrapar intrusos. Un esquema sencillo es tener un nombre de inicio de sesión especial con una contraseña fácil (p. Ej., nombre de entrada: guest, contraseña: guest). Siempre que alguien inicia una sesión empleando este nombre, se notifica de inmediato a los especialistas de seguridad del sistema. Otras trampas pueden ser errores fáciles de encontrar en el sistema operativo y cosas similares, diseñadas para atrapar a los intrusos con las manos en la masa. Stoll (1989) ha escrito un relato entretenido acerca de las trampas que puso para rastrear un espía que se introdujo en una computadora universitaria en busca de secretos militares.

5.5 MECANISMOS DE PROTECCIÓN

En las secciones anteriores hemos examinado muchos problemas potenciales, algunos técnicos otros no. En las siguientes secciones nos concentraremos en algunos de los métodos técnicos detallados que se emplean en los sistemas operativos para proteger los archivos y otras cosas. Todas estas técnicas establecen una distinción clara entre política (contra quién se van a proteger los datos de quién) y mecanismo (cómo hace el sistema que se cumpla la política). La separación de política y mecanismo se trata en (Levin et al., 1975). Nosotros haremos hincapié en el mecanismo, no en la política. Si se desea consultar material más avanzado, véase (Sandhu, 1993).

En algunos sistemas, la protección se impone mediante un programa llamado monitor de referencias. Cada vez que se intenta un acceso a un recurso que pudiera estar protegido, el sistema pide primero al monitor de referencias que verifique que tal acceso está permitido. El monitor examina entonces sus tablas de política y toma una decisión. A continuación describiremos el entorno en el que opera el monitor de referencias.

5.5.1 Dominios de protección

Un sistema de computadora contiene muchos "objetos" que deben protegerse. Estos objetos pueden ser hardware (p. ej., CPU, segmentos de memoria, unidades de disco o impresora) software (p. ej., procesos, archivos, bases de datos o semáforos).

Cada objeto tiene un nombre único con el cual se hace referencia a él, y un conjunto finita de operaciones que los procesos pueden efectuar con él. READ y WRITE son operaciones apropiadas para un archivo; UP y DOWN tienen sentido con un semáforo.

Es evidente que se requiere un mecanismo para prohibir a los procesos que accedan a objetos que no están autorizados para usar. Además, este mecanismo debe permitir restringir los procesos a un subconjunto de las operaciones permitidas cuando sea necesario. Por ejemplo, el proceso A podría tener derecho a leer, pero no escribir, el archivo F.

A fin de analizar los diferentes mecanismos de protección, resulta útil introducir el concepto de dominio. Un dominio es un conjunto de pares (objeto, derechos). Cada par especifica un objeto y algún subconjunto de las operaciones que se pueden efectuar con él. Un derecho en este contexto se refiere al permiso para efectuar una de las operaciones.

En la Fig. 5-22 se pueden ver tres dominios, y se muestran los objetos de cada dominio así como los derechos [leer (R), escribir (W), ejecutar (X)] disponibles para cada objeto. Obsérvese que Impresora 1 está en dos dominios al mismo tiempo. Aunque no se muestra en este ejemplo, es posible que el mismo objeto esté en múltiples dominios con diferentes derechos en cada uno.

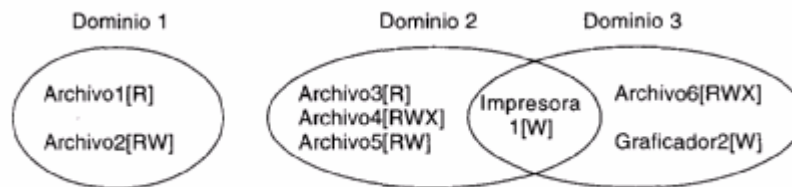


Figura 5-22. Tres dominios de protección.

, En cada instante, cada proceso se ejecuta en algún dominio de protección. En otras palabras, existe una colección de objetos a los que puede acceder, y para cada objeto tiene ciertos derechos. Los procesos pueden cambiar de un dominio a otro durante su ejecución. Las reglas para la conmutación de dominio dependen mucho del sistema del que se trate.

A fin de hacer más concreta la idea de dominio de protección, examinemos UNIX. En UNIX, el dominio de un proceso está definido por su uid y su gid. Dada una combinación (uid, gid), es posible preparar una lista completa de todos los objetos (archivos, incluidos los dispositivos de E/S presentados por archivos especiales, etc.), a los que se puede acceder, y si se puede acceder a ellos para leer, escribir o ejecutar. Dos procesos con la misma combinación (uid, gid) tienen acceso a exactamente el mismo conjunto de objetos. Dos procesos con diferentes valores (uid, gid) tienen acceso a diferentes conjuntos de archivos, aunque en la mayor parte de los casos el traslape es considerable.

Además, cada proceso en UNIX tiene dos mitades: la parte del usuario y la parte del kernel. Cuando un proceso emite una llamada al sistema, conmuta de la parte del usuario a la parte del kernel. Esta última tiene acceso a un conjunto de objetos distinto del de la parte del usuario. Por ejemplo, el kernel puede acceder a todas las páginas de la memoria física, a todo el disco y a todos los demás recursos protegidos. Por tanto, una llamada al sistema causa una conmutación de dominio.

Cuando un proceso efectúa EXEC con un archivo que tiene el bit SETUID o SETGID encendido adquiere un nuevo uid o gid efectivo. Al tener una combinación (uid, gid) diferente, el proceso tiene a su disposición un conjunto distinto de archivos y operaciones. La ejecución de un programa con SETUID o SETGID también es una conmutación de dominio, ya que cambian los derechos que se tienen.

Una pregunta importante es, ¿cómo hace el sistema para saber cuál objeto pertenece a cuál dominio? Conceptualmente, al menos, podemos imaginar una matriz grande cuyas filas son los dominios y cuyas columnas son los objetos. Cada cuadro indica los derechos, en su caso, que el dominio contiene para el objeto. En la Fig. 5-23 se muestra la matriz para la Fig. 5-22. Dada esta matriz y el número de dominio actual, el sistema puede saber si se permite cierto tipo de acceso a un objeto dado desde el dominio especificado.

Dominio	Objeto							
	Archivo1	Archivo2	Archivo3	Archivo4	Archivo5	Archivo6	Impresora1	Graficador2
1	Leer	Leer Escribir						
2			Leer	Leer Escribir Ejecutar	Leer Escribir		Escribir	
3						Leer Escribir Ejecutar	Escribir	Escribir

Figura 5-23. Una matriz de protección.

La conmutación de dominio misma se puede incluir fácilmente en el modelo de matriz si nos damos cuenta de que un dominio es también un objeto, con la operación ENTER (entrar). La Fig. 5-24 muestra otra vez la matriz de la Fig. 5-23, sólo que ahora incluye los tres dominios mismos como objetos. Los procesos del dominio 1 pueden conmutar al dominio 2 pero, una vez ahí, ya no pueden regresar. Esta situación modela la ejecución de un programa SETUID en UNIX. No se permiten otras conmutaciones de dominio en este ejemplo.

Dominio	Objeto											
	Archivo1	Archivo2	Archivo3	Archivo4	Archivo5	Archivo6	Impresora1	Graficador2	Dominio1	Dominio2	Dominio3	
1	Leer	Leer Escribir									Entrar	
2			Leer	Leer Escribir Ejecutar	Leer Escribir		Escribir					
3						Leer Escribir Ejecutar	Escribir	Escribir				

Figura 5-24. Matriz de protección con dominios como objetos.

5.5.2 Listas de control de acceso

En la práctica casi nunca se almacena una matriz como la de la Fig. 5-24 porque es grande y no muy poblada. La mayor parte de los dominios no tienen acceso a la mayor parte de los objetos, así que almacenar una matriz muy grande en su mayor parte vacía es un desperdicio de espacio en disco. Dos métodos que sí son prácticos son almacenar la matriz por filas o por columnas, y

luego almacenar sólo los elementos no vacíos. Los dos enfoques son sorprendentemente distintos. En esta sección examinaremos el almacenamiento por columna; en la siguiente veremos el almacenamiento por fila.

La primera técnica consiste en asociar a cada objeto una lista (ordenada) que contiene todos los dominios que pueden acceder al objeto, y cómo pueden hacerlo. Esta lista se denomina lista de control de acceso, o ACL. Si esto se implementara en UNIX, la forma más fácil sería colocar la ACL para cada archivo en un bloque de disco aparte e incluir el número de bloque en el nodo-*i* del archivo. Ya que sólo se almacenan las entradas no vacías de la matriz, el almacenamiento total requerido para todas las ACL combinadas es mucho menor que el que se necesitaría para toda la matriz.

Como ejemplo del funcionamiento de las ACL, sigamos imaginando que se usaran en UNIX, donde un dominio se especifica con un par (uid, gid). De hecho, las ACL se usaron en el precursor de UNIX, MULTICS, más o menos de la forma que vamos a describir, así que el ejemplo no es tan hipotético.

Supongamos ahora que tenemos cuatro usuarios (o sea, uids) Juan, Eva, Rita y Maite, que pertenecen a los grupos sistema, personal, estudiante y estudiante, respectivamente. Supongamos también que ciertos archivos tienen las siguientes ACL:

ArchivoO: (Juan, *, RWX)

Archivo 1: (Juan, sistema, RWX)

Archivo!: (Juan, *, RW-), (Eva, personal, RW-), (Maite, *, RW-)

Archivo3: (*, estudiante, R—)

Archivo4: (Rita, *, —), (*, estudiante, R—)

Cada entrada de ACL, entre paréntesis, especifica un uid, un gid y los accesos permitidos (R = leer, W = escribir, X = ejecutar). Un asterisco indica todos los uids o gids. ArchivoO puede ser leído, escrito o ejecutado por cualquier proceso cuyo uid sea Juan, sea cual sea su gid. Sólo pueden acceder a Archivo! los procesos con uid = Juan y gid = sistema. Un proceso que tiene uid = Juan y gid = personal puede acceder a ArchivoO pero no a Archivo!. Archivo! puede ser leído o escrito por procesos con uid = Juan y cualquier gid, ser leído por procesos con uid = Eva y gid = personal, o por procesos con uid = Maite y cualquier gid. Archivo3 puede ser leído por cualquier estudiante. Archivo4 resulta especialmente interesante. Su ACL dice que cualquiera con uid IB .Rito, en cualquier grupo, no tiene ningún acceso, pero que todos los demás estudiantes pueden leerlo. Mediante las ACL es posible prohibir a uids o gids específicos el acceso a un objeto, permitiéndolo a todos los demás de la misma clase.

Hasta aquí lo que UNIX no hace. Veamos ahora lo que «hace. UNIX proporciona tres bits, rwx, por archivo para el propietario, el grupo del propietario y otros. Este esquema es parecido a la ACL, pero comprimido a nueve bits. Se trata de una lista asociada con el objeto que indica quién puede acceder a él y cómo. Si bien el esquema UNIX de nueve bits es obviamente menos general que un sistema ACL con todas las de la ley, en la práctica es adecuado, y su implementación es mucho más sencilla y económica.

El propietario de un objeto puede cambiar su ACL en cualquier instante, así que es fácil prohibir accesos que antes se permitían. El único problema es que el cambio a la ACL probable-

mente no afectará a los usuarios que actualmente estén utilizando el objeto (p. ej., que tengan abierto el archivo).

5.5.3 Capacidades

La otra forma de "rebanar" la matriz de la Fig. 5-24 es por filas. Cuando se emplea este método, cada proceso tiene asociada una lista de los objetos a los que puede acceder, junto con una indicación de cuáles operaciones puede efectuar con cada uno; en otras palabras, su dominio. Esta lista se denomina lista de capacidades, y los elementos individuales que contiene se llaman capacidades (Dennis y Van Hom, 1966; Fabry, 1974).

En la Fig. 5-25 se muestra una lista de capacidades típica. Cada capacidad tiene un campo Tipo, que indica la clase de objeto de que se trata, un campo Derechos, que es un mapa de bits que indica cuáles de las operaciones que pueden efectuarse con este tipo de objeto están permitidas, y un campo Objeto, que es un apuntador al objeto mismo (p. ej., su número de nodo-i). Las listas de capacidades son ellas mismas objetos y se puede apuntar a ellas desde otras listas de capacidades, facilitando así el compartimiento de subdominios. Es común hacer referencia a las capacidades por su posición en la lista. Un proceso podría decir: "leer 1K del archivo al que apuntala capacidad 2". Esta forma de direccionamiento es similar al empleo de descriptores de archivo en UNIX.

#	Tipo	Derechos	Objeto
0	Archivo	R—	Apuntador a Archivos
1	Archivo	RWX	Apuntador a Archivo4
2	Archivo	RW-	Apuntador a Archivos
3	Apuntador	-W-	Apuntador a Impresora"

Figura 5-25. La lista de capacidades para el dominio 2 de la Fig. 5-23.

Es obvio que las listas de capacidad, o listas C, como también se les llama, deben estar protegidas contra alteración por parte del usuario. Se han propuesto tres métodos para protegerlas. El primero requiere una arquitectura etiquetada, un diseño de hardware en el que cada palabra de la memoria tiene un bit extra (o etiqueta) que indica si la palabra contiene una capacidad o no. El bit de etiqueta no se utiliza en las instrucciones ordinarias de aritmética, comparación u otro tipo, y sólo puede ser modificado por programas que se ejecuten en modo de kernel (es decir, el sistema operativo).

El segundo método consiste en mantener la lista C dentro del sistema operativo y hacer que los procesos hagan referencia a las capacidades sólo por su número de ranura, como se mencionó antes. Hydra (Wulf et al., 1974) funcionaba de esta forma.

El tercer método es mantener la lista C en el espacio de usuario, pero cifrar cada capacidad con una clave secreta que el usuario desconoce. Este enfoque es muy apropiado para sistemas distribuidos, y es utilizado ampliamente por Amoeba (Tanenbaum et al., 1990).

Además de los derechos específicos que dependen del objeto, como leer y ejecutar, las capacidades suelen tener derechos genéricos que son aplicables a todos los objetos. Como ejemplos de derechos genéricos podemos citar:

1. Copiar capacidad: crear una nueva capacidad para el mismo objeto.
2. Copiar objeto: crear un objeto duplicado con una nueva capacidad.
3. Quitar capacidad: eliminar una entrada de la lista C sin afectar el objeto.
4. Destruir objeto: eliminar permanentemente un objeto y una capacidad.

Un último comentario que vale la pena hacer acerca de los sistemas de capacidades es que es muy difícil revocar el acceso a un objeto. No es fácil para el sistema encontrar todas las capacidades vigentes para cualquier objeto y revocarlas, ya que pueden estar almacenadas en listas C por todo el disco. Una estrategia es hacer que cada capacidad apunte a un objeto indirecto, no al objeto mismo. Si el sistema hace que el objeto indirecto apunte al objeto real, siempre podrá romper esa conexión, invalidando así las capacidades. (Más tarde, cuando se presente al sistema una capacidad para el objeto indirecto, el usuario descubrirá que el objeto indirecto ahora apunta a un objeto nulo.)

Otra forma de lograr la revocación es el esquema empleado en Amoeba. Cada objeto contiene un número aleatorio largo, que también está presente en la capacidad. Cuando se presenta una | capacidad para usarse, se comparan ambos números, y sólo se permite la operación si los números coinciden. El propietario de un objeto puede solicitar que se cambie el número aleatorio del objeto, cancelando así la validez de las capacidades existentes. Ninguno de estos esquemas permite la revocación selectiva, es decir, revocar el permiso de José, pero de nadie más.

P.5.4 Canales encubiertos

Aun con listas de control de acceso y capacidades, puede haber puntos débiles en la seguridad. En (sta sección estudiaremos una clase de problema. Las ideas aquí expuestas se deben a Lampson (1973).

El modelo de Lampson implica tres procesos y aplica primordialmente a sistemas de tiempo impartido grandes. El primer proceso es el cliente, que desea que el segundo proceso, el servi», efectúe cierto trabajo. El cliente y el servidor no se tienen confianza ciega. Por ejemplo, la legación del servidor es ayudar a los clientes a llenar sus formas fiscales. Los clientes temen Sel servidor registre en secreto sus datos financieros, por ejemplo, manteniendo una lista de ién gana cuánto, y que luego venda la lista. El servidor teme que los clientes tratarán de robar el lioso programa fiscal.

El tercer proceso es el colaborador, que está conspirando con el servidor para robar en efecto los confidenciales del cliente. El colaborador y el servidor por lo regular son propiedad de la misma persona. Los tres procesos se muestran en la Fig. 5-26. El objeto de este ejercicio es diseñar un sistema en el que sea imposible que el servidor filtre al colaborador la información que recibió legítimamente del cliente. Lampson llamó a éste el problema de confinamiento.

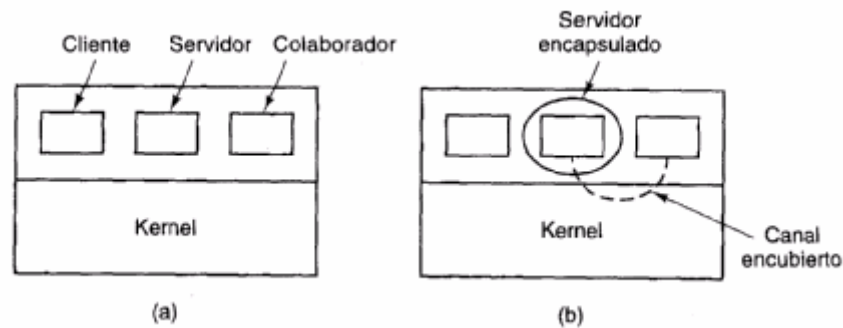


Figura 5-26. (a) Los procesos cliente, servidor y colaborador, (b) El servidor encapsulado aún puede filtrar información al colaborador a través de canales encubiertos.

Desde el punto de vista del diseñador del sistema, el objetivo es encapsular o confinar el servidor de tal manera que no pueda pasar información al colaborador. Si usamos un esquema de matriz de protección fácilmente podremos garantizar que el servidor no podrá comunicarse con el colaborador escribiendo un archivo al que el colaborador tenga acceso de lectura. Probablemente también podremos asegurar que el servidor no podrá comunicarse con el colaborador empleando el mecanismo de comunicación entre procesos del sistema.

Por desgracia, puede haber canales de comunicación más sutiles. Por ejemplo, el servidor puede tratar de comunicar una comente de bits binaria como sigue. Para enviar un bit, el servidor calcula a máxima velocidad durante un intervalo de tiempo fijo. Para enviar un bit 0, el servidor se duerme durante un lapso idéntico.

El colaborador puede tratar de detectar la corriente de bits vigilando cuidadosamente su tiempo de respuesta. En general, se obtendrá una mejor respuesta cuando el servidor está enviando un 0 que cuando está enviando un 1. Este canal de comunicación se denomina canal encubierto y se ilustra en la Fig. 5-26(b).

Desde luego, el canal encubierto es ruidoso, pues contiene mucha información ajena, pero es posible transmitir información de forma confiable a través de un canal ruidoso empleando un código de corrección de errores (p. ej., un código de Hamming, o incluso algo más complejo). El empleo de un código de corrección de errores reduce aún más el ancho de banda, ya de por sí pequeño, del canal encubierto, pero podría ser suficiente para filtrar una cantidad sustancial de información. Es evidente que ningún modelo de protección basado en una matriz de objetos y dominios va a prevenir este tipo de fugas.

La modulación de la utilización de la CPU no es el único canal encubierto. La tasa de paginación también puede modularse (muchas fallas de página para un 1, ninguna falla para un 0). De hecho, casi cualquier forma de degradar el rendimiento del sistema de forma sincronizada es un candidato. Si el sistema cuenta con un mecanismo de candados para los archivos, el servidor puede poner un candado a un archivo para indicar un 1, y quitar el candado para indicar un 0. En algunos sistemas, un proceso puede detectar la situación de un candado incluso en un archivo al que no puede acceder.

La adquisición y liberación de recursos dedicados (unidades de cinta, graficadores, etc.), también puede aprovecharse para enviar señales. El servidor adquiere el recurso para enviar un 1 y lo libera para enviar un 0. En UNIX, el servidor podría crear un archivo para indicar un 1 y eliminarlo para indicar un 0; el colaborador podría usar la llamada al sistema ACCESS para ver si existe el archivo. Esta llamada funciona incluso si el colaborador no tiene permiso para usar el archivo. Por desgracia, existen muchos otros canales encubiertos.

Lampson también menciona una forma de filtrar información al propietario (humano) del proceso servidor. Es de suponer que dicho proceso podrá informar a su dueño de cuánto trabajo efectuó para el cliente, a fin de poder cobrarle la cantidad correspondiente. Si el cargo por el servicio es de, digamos, 100 dólares y los ingresos del cliente ascienden a 53K dólares, el servidor podría informar el cargo como 100.53 dólares a su propietario.

Tan sólo encontrar todos los canales encubiertos, no digamos bloquearlos, es extremadamente difícil. En la práctica no hay mucho que se pueda hacer. La introducción de un proceso que cause fallas de página al azar, o dedique su tiempo a degradar el rendimiento del sistema de otras formas a fin de reducir el ancho de banda de los canales encubiertos no es una propuesta atractiva.

5.6 GENERALIDADES DEL SISTEMA DE ARCHIVOS DE MINIX

Al igual que cualquier otro sistema de archivos, el de MINIX debe enfrentar todas las cuestiones que acabamos de estudiar: debe asignar y liberar espacio para los archivos, seguir la pista a los bloques de disco y al espacio libre, proteger de alguna forma a los archivos contra el uso no autorizado, etc. En el resto del capítulo examinaremos de cerca el sistema de archivos de MINIX para ver cómo alcanza estos objetivos.

En la primera parte del capítulo hicimos referencia una y otra vez a UNIX y no a MINIX en aras de la generalidad, aunque la interfaz externa de los dos es casi idéntica. Ahora nos concentraremos en el diseño interno de MINIX. Si el lector desea información acerca del diseño interno de UNIX puede consultar Thompson (1978), Bach (1987), Lions (1996) y Vahalia (1996).

El sistema de archivos de MINIX es sólo un programa en C grande que se ejecuta en el espacio de usuario (véase la Fig. 2-26). Para leer y escribir archivos, los procesos de usuario envían mensajes al sistema de archivos indicándole lo que quieren que haga. El sistema de archivos efectúa el trabajo y luego devuelve una respuesta. De hecho, el sistema de archivos es un servidor de archivos en red que por casualidad se ejecuta en la misma máquina que el invocador.

Este diseño tiene ciertas implicaciones importantes. Por un lado, podemos modificar el sistema de archivos, experimentar con él y probarlo con independencia casi total del resto de MINIX. Por el otro, es muy fácil trasladar todo el sistema de archivos a cualquier computadora que tenga un compilador de C, compilarlo ahí, y utilizarlo como servidor de archivos remoto autónomo tipo UNIX. Los únicos cambios que habría que efectuar serían en el área del mecanismo de envío y recepción de mensajes, que vana de un sistema a otro.

Días siguientes secciones presentaremos una reseña de muchas de las áreas clave del diseño del sistema de archivos. Específicamente, examinaremos los mensajes, la organización del sistema de archivos, los mapas de bits, los nodos-i, el caché de bloques, los directorios y rutas, los

descriptores de archivo, los candados de archivo y los archivos especiales (más los conductos). Después de estudiar todos estos temas, mostraremos un ejemplo sencillo de cómo embonan las piezas rastreando lo que sucede cuando un proceso de usuario ejecuta la llamada al sistema READ.

5.6.1 Mensajes

El sistema de archivos acepta 39 tipos de mensajes que solicitan trabajo. Todos, menos dos, son para llamadas al sistema MINIX. Las dos excepciones son mensajes generados por otras partes de MINIX. De las llamadas al sistema, 31 se aceptan de procesos de usuario. Seis mensajes son para llamadas al sistema que son manejadas primero por el administrador de memoria, que luego llama al sistema de archivos para que efectúe una parte del trabajo. El sistema de archivos procesa también otros dos mensajes. Todos los mensajes se muestran en la Fig. 5-27.

La estructura del sistema de archivos es básicamente la misma que la del administrador de memoria y todas las tareas de E/S. Hay un ciclo principal que espera la llegada de un mensaje. Cuando llega un mensaje, se extrae su tipo y se usa como índice para consultar una tabla que contiene apuntadores a los procedimientos dentro del sistema de archivos que se encargan de cada tipo de mensaje. Luego se invoca el procedimiento apropiado, el cual efectúa el trabajo y devuelve un valor de situación. A continuación, el sistema de archivos envía una respuesta de vuelta al invocador y regresa al principio de su ciclo para esperar el siguiente mensaje.

5.6.2 Organización del sistema de archivos

Un sistema de archivos MINIX es una entidad lógica y autónoma con nodos-i, directorios y bloques de datos. Este sistema puede almacenarse en cualquier dispositivo por bloques, como un disco flexible o (una porción de) un disco duro. En todos los casos, la organización del sistema de archivos tiene la misma estructura. En la Fig. 5-28 se muestra esta organización para un disquete de 360K con 128 nodos-i y un tamaño de bloque de 1K. Los sistemas de archivos más grandes, o aquellos con más o menos nodos-i o un tamaño de bloque distinto, tienen los mismos seis, componentes en el mismo orden, pero sus tamaños relativos podrían ser distintos.

Todo sistema de archivos comienza con un bloque de arranque, el cual contiene código ejecutable. Cuando se enciende la computadora, el hardware lee el bloque de arranque del dispositivo de arranque y lo coloca en la memoria, salta a él y comienza a ejecutar su código. Este código inicia el proceso de cargar el sistema operativo mismo. Una vez puesto en marcha el sistema, no se usa más el bloque de arranque. No todas las unidades de disco pueden usarse como dispositivos de arranque, pero a fin de mantener uniforme la estructura, cada dispositivo de bloques lleva un bloque reservado para el código de arranque. En el peor de los casos, esta estrategia desperdicia un bloque. Con objeto de evitar que el hardware trate de arrancar con un dispositivo no de arranque, se coloca un número mágico en cierto lugar conocido del bloque de arranque si y sólo si el código ejecutable está escrito en el dispositivo. Al arrancar desde un dispositivo, el hardware (en realidad, el código BIOS) se negará a cargar de un dispositivo que

Mensajes de usuarios	Parámetros de entrada	Valor de respuesta
ACCESS	Nombre de archivo, modo de acceso	Estado
CHDIR	Nombre del nuevo directorio de trabajo	Estado
CHMOD	Nombre de archivo, nuevo modo	Estado
CHOWN	Nombre de archivo, nuevo dueño, grupo	Estado
CHROOT	Nombre del nuevo directorio raíz	Estado
CLOSE	Descriptor de archivo del archivo por cerrar	Estado
CREAT	Nombre del archivo por crear, modo	Descriptor de archivo
DUP	Descriptor de archivo (para dup2, dos d.a.)	Nuevo descriptor de archivo
FCNTL	Descriptor de archivo, código de función, arg.	Depende de la función
FSTAT	Nombre de archivo, buffer	Estado
IOCTL	Descriptor de archivo, código de función, arg.	Estado
LINK	Archivo al cual enlazar, nombre de enlace	Estado
LSEEK	Descriptor de archivo, distancia, de dónde	Nueva posición
MKDIR	Nombre de archivo, modo	Estado
MKNOD	Nombre de dir. o especial, modo, dirección	Estado
MOUNT	Archivo especial, dónde montar, bandera ro	Estado
OPEN	Nombre del archivo por abrir, bandera r/w	Descriptor de archivo
PIPE	Apuntador a dos descriptores de archivo (modificado)	Estado
READ	Descriptor de archivo, buffer, cuántos bytes	Núm. bytes leídos
RENAME	Nombre de archivo, nombre de archivo	Estado
RMDIR	Nombre de archivo	Estado
STAT	Nombre de archivo, buffer de estado	Estado
STIME	Apuntador a hora actual	Estado
SYNC	(Ninguno)	Siempre OK
TIME	Apuntador al lugar donde va la hora actual	Estado
TIMES	Apuntador a buffer para tiempos de proceso e hijo	Estado
UMASK	Complemento de máscara de modo	Siempre OK
UMOUNT	Nombre del archivo especial por desmontar	Estado
UNLINK	Nombre del archivo por desenlazar	Estado
UTIME	Nombre de archivo, tiempos de archivo	Siempre OK
WRITE	Descriptor de archivo, buffer, cuántos bytes	Núm. bytes escritos
Mensajes de MM	Parámetros de entrada	Valor de respuesta
EXEC	Pid	Estado
EXIT	Pid	Estado
FORK	Pid del padre, pid del hijo	Estado
SETGID	Pid, gid real y efectivo	Estado
SETPID	Pid	Estado
SETUID	Pid, uid real y efectivo	Estado
Otros mensajes	Parámetros de entrada	Valor de respuesta
REVIVE	Proceso por revivir	(Sin respuesta)
UNPAUSE	Proceso por verificar	(Véase texto)

Figura 5-27. Mensajes del sistema de archivos. Los parámetros de nombre de archivo siempre son apuntadores al nombre. El código "situación" como valor de respuesta significa OK o ERROR.

no cuente con número mágico. Esto evita el empleo inadvertido de basura como programa de arranque.

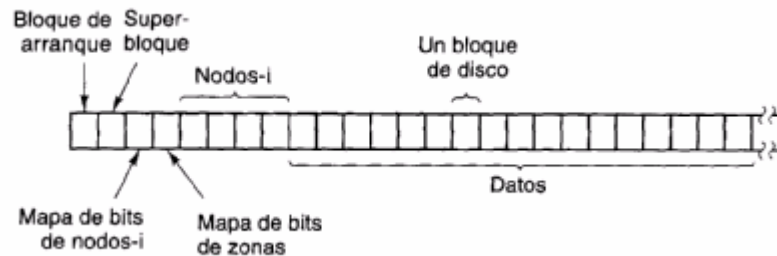


Figura 5-28. Organización del disco más sencillo: un disquete de 360K con 128 nodos-i y bloques de 1K (es decir, dos sectores consecutivos de 512 bytes se tratan como un solo bloque).

El superbloque contiene información que describe la organización del sistema de archivos, y se ilustra en la Fig. 5-29. La función principal del superbloque es indicarle al sistema de archivos el tamaño de sus diversos componentes. Dado el tamaño de bloque y el número de nodos-i, es fácil calcular el tamaño del mapa de bits de nodos-i y el número de bloques de nodos-i. Por ejemplo, si los bloques son de 1K, cada bloque de mapa de bits tiene 1K bytes (8K bits), así que puede indicar la situación de hasta 8192 nodos-i. (De hecho, el primer bloque sólo puede manejar hasta 8191 nodos-i, ya que no existe el nodo-i cero, pero de todos modos hay un bit para él en el mapa.) Si hay 10 000 nodos-i, se necesitan dos bloques para el mapa de bits. Puesto que cada nodo-i ocupa 64 bytes, un bloque de 1K contiene hasta 16 nodos-i. Con 128 nodos-i utilizables, se requieren ocho bloques de disco para contenerlos todos.

Explicaremos detalladamente la diferencia entre zonas y bloques más adelante, pero por ahora baste con decir que el espacio de almacenamiento en disco se puede asignar en unidades (zonas) de 1, 2, 4, 8 o, en general, 2^n bloques. El mapa de bits de zonas lleva la cuenta del espacio libre en términos de zonas, no de bloques. En todos los disquetes estándar empleados por MINIX el tamaño de zona y de bloque es el mismo (1K), así que como primera aproximación una zona es lo mismo que un bloque en estos dispositivos. Hasta que llegemos a los detalles de la asignación de espacio de almacenamiento más adelante en este capítulo, podemos pensar "bloque" cada vez que veamos "zona".

Cabe señalar que el número de bloques por zona no se almacena en el superbloque, ya que nunca se necesita. Lo único que se necesita es el logaritmo base 2 de la relación zona/bloque, que se usa como cuenta de desplazamiento para convertir zonas en bloques y viceversa. Por ejemplo, si hay 8 bloques en cada zona, $\log_2 8 = 3$, así que para encontrar la zona que contiene el bloque 1281 desplazamos 128 tres bits a la derecha para obtener la zona 16.

El mapa de bits de zonas sólo incluye las zonas de datos (esto es, los bloques empleados para los mapas de bits y los nodos-i no están en el mapa), designándose a la primera zona de datos zona 1 en el mapa de bits. Al igual que con el mapa de bits de nodos-i, no se utiliza el bit 0 de este mapa, de modo que el primer bloque del mapa de zonas puede seguir la pista a 8191 zonas y los bloques

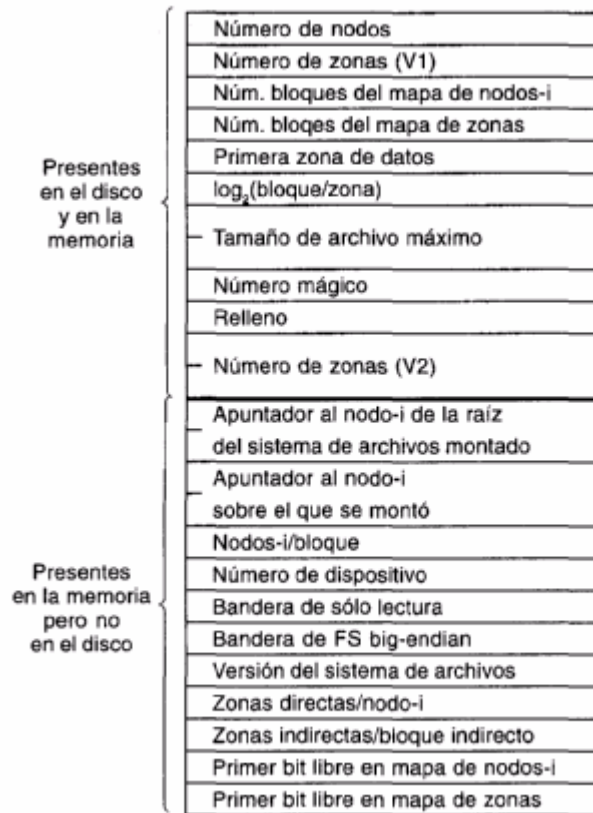


Figura 5-29. El superbloque de MINIX.

subsecuentes pueden seguir la pista a 8192 zonas cada uno. Si examinamos los mapas de bits de un disco recién formateado, veremos que tanto el mapa de bits de nodos-i como el de zonas tienen dos bits encendidos (en 1). Uno es para el nodo-i o zona cero inexistente; el otro es para el nodo-i y la zona empleados por el directorio raíz del dispositivo, que se coloca ahí cuando se crea el sistema de archivos.

La información que está en el superbloque es redundante porque a veces se necesita en una forma y a veces en otra. Con 1K dedicado al superbloque, es razonable calcular esta información en todas las formas en que se necesita, para no tener que recalcularla con frecuencia durante la ejecución. Por ejemplo, el número de zona de la primera zona de datos del disco se puede calcular a partir del tamaño de bloque, el tamaño de zona, el número de nodos-i y el número de zonas, pero es fácil guardarlo simplemente en el superbloque. De todos modos, el resto del superbloque se desperdicia, así que usar otra palabra de él no cuesta nada.

Cuando se inicia MINIX, se lee el superbloque del dispositivo raíz y se coloca en una tabla en la memoria. De forma similar, conforme se montan los demás sistemas de archivos, sus bloques se traen también a la memoria. La tabla de superbloques contiene varios campos

que no están presentes en el disco. Éstos incluyen banderas que permiten especificar que un dispositivo es sólo de lectura (ro) o que sigue una convención de orden de bytes opuesta a la estándar, así como campos para agilizar el acceso indicando puntos de los mapas de bits por debajo de los cuales todos los bits están marcados como ocupados. Además, hay un campo que describe el dispositivo del cual provino el superbloque.

Antes de que un disco pueda usarse como sistema de archivos de MINIX, se le debe imponer la estructura de la Fig. 5-28. Se incluye el programa mkfs para construir sistemas de archivos. Este programa puede invocarse ya sea con un comando como

```
mkfs/dev/fd-1 1440
```

para construir un sistema de archivos vacío de 1440 bloques en el disco flexible de la unidad 1, o se le puede proporcionar un archivo prototipo que lista los directorios y los archivos que deben incluirse en el nuevo sistema de archivos. Este comando también coloca un número mágico en el superbloque para identificar el sistema de archivos como sistema de archivos válido de MINIX. El sistema de archivos de MINIX ha evolucionado, y algunos aspectos de él (como el tamaño de los nodos-i) eran diferentes en las primeras versiones. El número mágico identifica la versión de mkfs que creó el sistema de archivos, para que puedan compensarse las diferencias. La llamada al sistema MOUNT rechazará los intentos por montar un sistema de archivos que no tenga el formato de MINIX, digamos un disquete MS-DOS. Esta llamada comprueba que el superbloque contenga un número mágico válido y otras cosas.

5.6.3 Mapas de bits

MINIX sabe cuáles nodos-i y zonas están libres gracias a dos mapas de bits (véase la Fig. 5-29). Cuando se elimina un archivo, es cosa sencilla calcular cuál bloque del mapa de bits contiene el bit que corresponde al nodo-i que se está liberando y encontrarlo empleando el mecanismo de caché normal. Una vez que se localiza ese bloque, el bit en cuestión se pone en 0. Las zonas se liberan del mapa de bits de zonas de la misma manera.

Lógicamente, cuando se va a crear un archivo el sistema de archivos debe examinar los bloques del mapa de bits uno por uno hasta encontrar el primer nodo-i libre. A continuación se asigna este nodo-i al nuevo archivo. De hecho, la copia del superbloque que está en la memoria tiene un campo que apunta al primer nodo-i libre, así que no se requiere una búsqueda sino hasta, después de que se usa un nodo, ya que entonces es necesario actualizar el apuntador de modo que, apunte al que ahora es el siguiente nodo-i libre, el cual muchas veces resulta ser el siguiente, o uno muy cercano. De forma similar, cuando se libera un nodo-i, se averigua si este nodo va antes de aquel al que actualmente se está apuntando, y se actualiza el apuntador si es necesario. Si todas las ranuras de nodo-i del disco están llenas, la rutina de búsqueda devuelve un cero, y ésta es la razón por la que no se usa el nodo-i cero (es decir, para que pueda usarse para indicar que la búsqueda fracasó). (Cuando mkfs crea un nuevo sistema de archivos, pone en ceros el nodo-i cero y pone en 1 el bit más bajo del mapa de bits, a fin de que el sistema de archivos nunca intente asignarlo.) Todo lo que hemos dicho aquí acerca de los mapas de bits de nodos-i también aplica al mapa de bits de zonas; se efectúa una búsqueda lógica de la primera zona libre cuando se necesita espacio,

pero se mantiene un apuntador a la primera zona libre con objeto de eliminar casi por completo la necesidad subsecuente de efectuar búsquedas secuenciales en el mapa de bits.

Con estos antecedentes, ya podemos explicar la diferencia entre zonas y bloques. La razón para tener zonas es ayudar a que los bloques de disco que pertenecen al mismo archivo se encuentren en el mismo cilindro, a fin de mejorar el rendimiento cuando el archivo se lee secuencialmente. La estrategia que se escogió hace posible asignar varios bloques a la vez. Por ejemplo, si el tamaño de bloque es de 1K y el tamaño de zona es de 4K, el mapa de bits de zonas sigue la pista a las zonas, no a los bloques. Un disco de 20M tiene 5K zonas de 4K, y por tanto 5K bits en su mapa de zonas.

La mayor parte del sistema de archivos funciona por bloques. Las transferencias de disco siempre se efectúan un bloque a la vez, y el caché de buffer también trabaja con bloques individuales. Sólo unas cuantas partes del sistema que se ocupan de direcciones físicas en disco (p. ej., el mapa de bits de zonas y los nodos-i) saben de la existencia de zonas.

Tuvieron que tomarse ciertas decisiones de diseño al crear el sistema de archivos de MINIX. En 1985, cuando se concibió MINIX, la capacidad de los discos era pequeña, y se esperaba que muchos usuarios sólo tendrían discos flexibles. Se tomó la decisión de restringir las direcciones en disco a 16 bits en el sistema de archivos VI, principalmente para poder almacenar muchas direcciones en los bloques indirectos. Con un número de zona de 16 bits y zonas de 1K, sólo es posible direccionar 64K zonas, lo que limita los discos a 64M. Ésta era una cantidad enorme de espacio de almacenamiento en aquellos días, y se pensaba que al aumentar el tamaño de los discos sería fácil cambiar a zonas de 2K o 4K, sin tener que cambiar el tamaño de bloque. Los números de zona de 16 bits también han facilitado mantener el tamaño de los nodos-i en 32 bytes.

Conforme se fue desarrollando MINIX, y los discos grandes se volvieron más comunes, se hizo evidente la necesidad de efectuar cambios. Muchos archivos son más pequeños que 1K, así que aumentar el tamaño de bloque implicaría desperdiciar ancho de banda del disco, leyendo y escribiendo bloques en su mayor parte vacíos y desperdiciando valiosa memoria principal al guardarlos en el caché de buffer. Se podía haber aumentado el tamaño de zona, pero esto habría implicado más espacio de disco desperdiciado, y seguía siendo deseable retener un funcionamiento eficiente en los discos pequeños. Otra alternativa razonable habría sido tener diferentes tamaños de zona en los dispositivos grandes y en los pequeños.

Al final se decidió aumentar el tamaño de los apuntadores a disco a 32 bits. Esto permite al sistema de archivos MINIX V2 manejar dispositivos de hasta 4 terabytes en términos de bloques y zonas de 1K. En parte, esta decisión fue consecuencia de otras decisiones relativas al contenido de los nodos-i que hicieron justificable aumentar el tamaño de nodo-i a 64 bytes.

Las zonas también introducen un problema inesperado, que podemos explicar mejor con la ayuda de un ejemplo sencillo en el que otra vez suponemos zonas de 4K y bloques de 1 K. Supongamos que un archivo tiene una longitud de 1K, lo que implica que se le asignó una zona. Los bloques entre 1K y 4K contienen basura (residuo del propietario anterior), pero no hay problema porque el tamaño del archivo está claramente marcado en el nodo-i como 1K. De hecho, los bloques que contienen basura no se colocan en el caché de bloques, ya que las lecturas se efectúan por bloques, no por zonas. Las lecturas más allá del final de un archivo siempre devuelven una cuenta de 0 y nada de datos.

Ahora alguien realiza un SEEK a 32768 y escribe un byte. El tamaño del archivo cambia ahora a 32769. Los SEEK subsecuentes a 1K seguidos de intentos por leer los datos podrán leer ahora el contenido previo del bloque, lo cual es una violación grave de la seguridad.

La solución es vigilar la ocurrencia de esta situación cuando se realiza una escritura más allá del final de un archivo, y poner explícitamente en cero todos los bloques que todavía no se han asignado en la zona que previamente era la última. Aunque esta situación rara vez ocurre, el código tiene que preverla, lo que lo hace un poco más complejo.

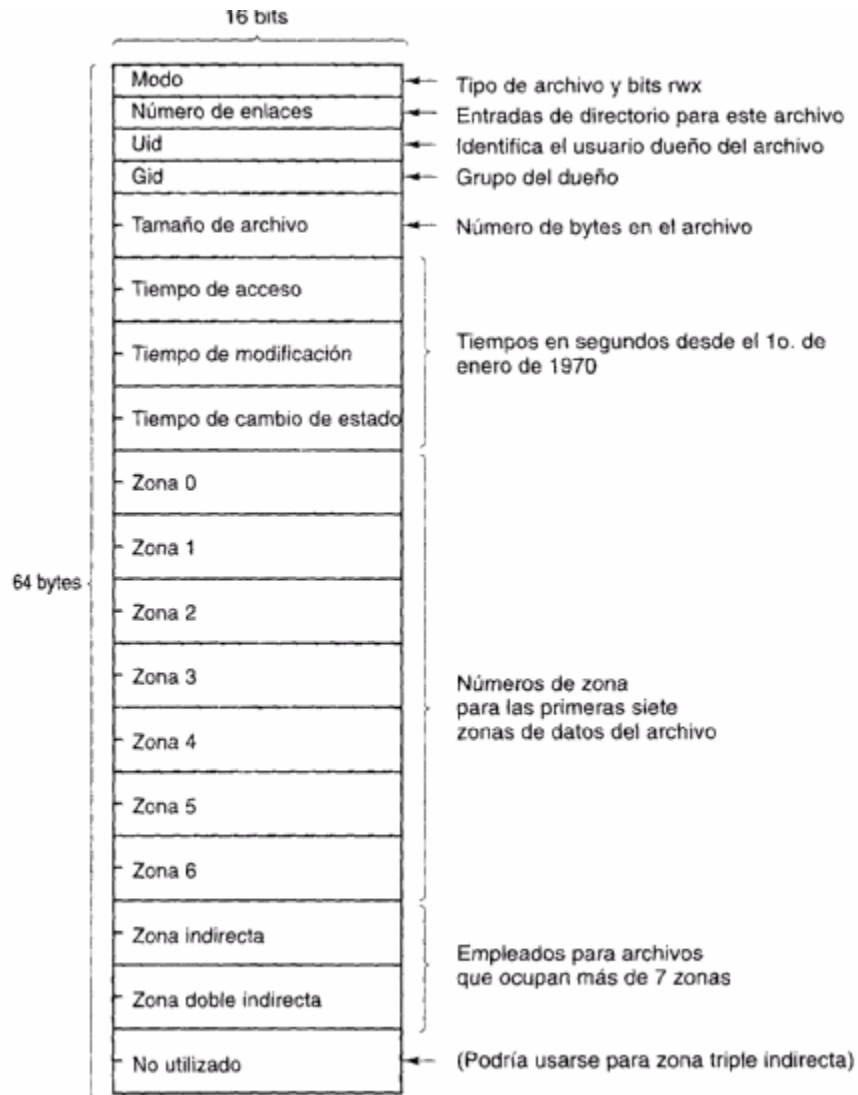
5.6.4 Nodos-i

La organización del nodo-i de MINIX se da en la Fig. 5-30, y es casi idéntica a la de un nodo-i estándar de UNIX. Los apuntadores a zonas de disco tienen 32 bits, y sólo hay nueve apuntadores, siete directos y dos indirectos. Los nodos-i de MINIX ocupan 64 bytes, lo mismo que los nodos-i UNIX estándar, y hay espacio disponible para un décimo apuntador (de triple indirección), aunque su uso no está contemplado en la versión estándar del sistema de archivos. Los tiempos de acceso a nodo-i, modificación y cambio de nodo-i en MINIX son estándar, igual que en UNIX. El último de éstos se actualiza en casi cada operación con el archivo, con excepción de la lectura.

Cuando se abre un archivo, se localiza su nodo-i y se trae a la tabla inode en la memoria, donde permanece hasta que se cierra el archivo. Esta tabla tiene unos cuantos campos adicionales que no están presentes en el disco, como el dispositivo y número del nodo-i, con objeto de que el sistema de archivos sepa dónde debe reescribirlo si se modifica mientras está en la memoria. La tabla también tiene un contador por cada nodo-i. Si el mismo archivo se abre más de una vez, sólo se guardará en la memoria una copia del nodo-i, pero el contador se incrementará cada vez que el archivo se abra y se decrementará cada vez que se cierre. Sólo si el contador llega a cero se retira el nodo-i de la tabla, Si el nodo se modificó después de que se cargó en la memoria, también se reescribe en el disco.

La función principal del nodo-i de un archivo es indicar dónde están los bloques de datos. Los primeros siete números de zona están en el nodo-i mismo. En la distribución estándar, con zonas bloques de 1K, los archivos de hasta 7K no necesitan bloques de indirección. Más allá de 7K se requieren zonas de indirección, empleando el esquema de la Fig. 5-10, excepto que sólo se usan los bloques de indirección sencilla y doble. Con bloques y zonas de 1 K y números de zona de 32 bits, un solo bloque indirecto contiene 256 entradas, lo que representa un cuarto de megabyte de almacenamiento. El bloque de doble indirección apunta a 256 bloques de indirección sencilla, dando acceso a hasta 64 megabytes. El tamaño máximo de un sistema de archivos MINIX es de 1 G, así que una modificación para usar el bloque de triple indirección o zonas más grandes podría ser útil si fuera deseable acceder a archivos muy grandes en un sistema MINIX.

El nodo-i también contiene la información de modo, que indica el tipo de archivo de que se trata (normal, de directorio, especial por bloques, especial por caracteres o conducto), y da los bits de protección y de SETUID y SETGID. El campo link (enlace) del nodo-i registra el numeral de entradas de directorio que apuntan al nodo-i, y permite al sistema de archivos saber cuándo debe liberar el espacio que ocupa el archivo. No debe confundirse este campo con el contador (presente sólo en la tabla inode en la memoria, no en el disco) que indica cuántas veces está abierto actualmente el archivo, casi siempre por procesos distintos.



5.6.5 El caché de bloques

MINIX emplea un caché de bloques para mejorar el rendimiento del sistema de archivos. El caché se implementa como arreglo de buffers, cada uno formado por un encabezado que contiene apuntadores, contadores y banderas, y un cuerpo con espacio para un bloque de disco. Todos los buffers que no se están usando se encadenan en una lista doblemente enlazada, desde el más recientemente utilizado (MRU) hasta el menos recientemente utilizado (LRU), como se ilustra en la Fig. 5-31.

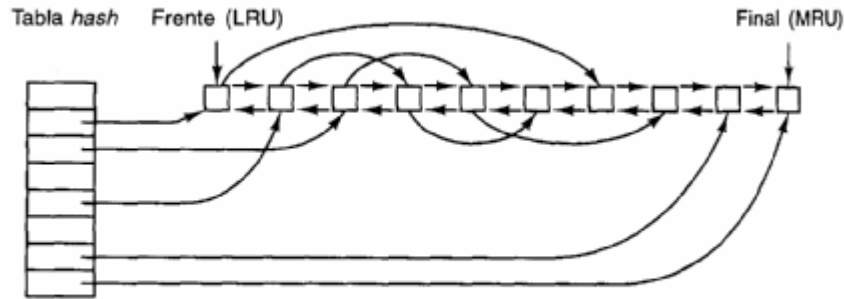


Figura 5-31. Las listas enlazadas empleadas por el caché de bloques.

Además, se emplea una tabla hash (o de cálculo de clave) para determinar con rapidez si un bloque dado está en el caché o no. Todos los buffers que contienen un bloque con el código hash k se encadenan en una lista enlazada sencilla a la que apunta la entrada k de la tabla hash. La función hash simplemente extrae los n bits de orden bajo del número de bloque, de modo que bloques de diferentes dispositivos aparecen en la misma cadena hash. Cada uno de los buffers está en alguna de estas cadenas. Desde luego, cuando se inicializa el sistema de archivos después de iniciarse MINIX, todos los buffers están desocupados, y todos están en una sola cadena a la que apunta la entrada cero de la tabla hash. En ese momento, todas las demás entradas de la tabla hash contienen un apuntador nulo, pero tan pronto como el sistema se pone en marcha se retiran buffers de la cadena cero y se construyen otras cadenas.

Cuando el sistema de archivos necesita un bloque, invoca un procedimiento, `get_block`, que calcula el código hash para ese bloque y examina la lista apropiada. Se invoca `get_block` con un número de dispositivo además de un número de bloque, y la búsqueda compara ambos números con los campos correspondientes en la cadena de buffers. Si se encuentra un buffer que contenga el bloque, se incrementa un contador en el encabezado del buffer para indicar que el bloque en uso, y se devuelve un apuntador a él. Si no se encuentra un bloque en la lista hash, se puede usar el primer buffer de la lista LRU; se garantiza que ya no se está usando, y el bloque contenido se puede desalojar para liberar el buffer.

Una vez que se escoge el bloque por desalojar, se revisa otra bandera de su encabezado para ver si el bloque ha sido modificado desde que se trajo al caché. De ser así, el bloque se reescribe en el disco. En este punto se lee del disco el bloque requerido enviando un mensaje a la tarea disco. El sistema de archivos queda suspendido hasta que llega el bloque, y entonces continúa devolviendo al invocador un apuntador al bloque.

Una vez que el procedimiento que solicitó el bloque termina su trabajo, invoca otro procedimiento, `put_block`, para liberar el bloque. Normalmente, un bloque se usará de inmediato y luego se liberará, pero como es posible que se presenten solicitudes adicionales para un bloque antes de ser liberado, `put_block` decrementa el contador de usos y coloca el buffer de vuelta en la LRU sólo cuando el contador de usos ha regresado a cero. Mientras el contador sea distinto de cero, el bloque permanecerá en el limbo.

Uno de los parámetros de `put_block` indica qué clase de bloque (p. ej., nodos-i, directorio, datos) se está liberando. Dependiendo de la clase, se toman dos decisiones clave:

1. Si se debe poner el bloque al frente o al final de la lista LRU.
2. Si se debe escribir el bloque (si se modificó) en el disco inmediatamente o no. Los bloques con baja probabilidad de que se necesiten otra vez pronto, como los superbloques, pasan al frente de la lista con objeto de que sean reclamados la próxima vez que se necesite un buffer libre. Todos los demás bloques pasan al final de la lista según el régimen LRU estricto. Un bloque modificado no se reescribe hasta que ocurre uno de dos sucesos:

1. El bloque llega al frente de la cadena LRU y es desalojado.
2. Se ejecuta una llamada al sistema SYNC.

SYNC no recorre la cadena LRU, sino que recorre con la ayuda de un índice el arreglo de buffers que están en el caché. Incluso si un buffer no se ha liberado todavía, si ha sido modificado SYNC lo encontrará y se asegurará de que se actualice la copia en disco.

Sin embargo, hay una excepción. Un superbloque modificado se escribe en el disco de inmediato. En una versión anterior de MINIX se modificaba un superbloque cuando se montaba un sistema de archivos, y el propósito de la escritura inmediata era reducir la posibilidad de corromper el sistema de archivos en caso de una caída. Los superbloques ya no se modifican, así que el código para escribirlos de inmediato es un anacronismo. En la configuración estándar, ningún otro bloque se escribe inmediatamente. Sin embargo, si se modifica la definición por omisión de `E_ROBUST` en el archivo de configuración del sistema, `include/minix/config.h`, se puede compilar el sistema de archivos de modo que marque los bloques de nodos-i, directorio, dirección y mapa de bits de modo que se escriban tan pronto como sean liberados. El propósito de esto es hacer el sistema de archivos más robusto; el precio que se paga es un funcionamiento más lento. El que esta medida sea provechosa o no es incierto. Una falla en la alimentación eléctrica en un momento en el que todavía no se han escrito todos los bloques va a causar dolores de cabeza, sea que se pierda un bloque de nodos-i o de datos.

Cabe señalar que el procedimiento dentro del sistema de archivos que solicitó y usó el bloque y el que enciende la bandera del encabezado que indica que el bloque fue modificado. Los procedimientos `get_block` y `put_block` sólo se ocupan de manipular las listas enlazadas; no tienen idea de cuál procedimiento del sistema quiere cuál bloque, o por qué.

5.6.6 Directorios y rutas

Otro subsistema importante dentro del sistema de archivos es la administración de directorios y nombres de ruta. Muchas llamadas al sistema, como `OPEN` tienen un nombre de archivo como parámetro. Lo que realmente se necesita es el nodo-i de ese archivo, así que toca al sistema de archivos buscar el archivo en el árbol de directorios y localizar su nodo-i.

Un directorio MINIX consiste en un archivo que contiene entradas de 16 bytes. Los dos primeros bytes forman un número de nodo-i de 16 bits, y los 14 bytes restantes son el nombre de

archivo. Esto es idéntico a la entrada de directorio UNIX tradicional que vimos en la Fig. 5-1 Para buscar la ruta `/usr/ast/mbox`, el sistema primero busca `usr` en el directorio raíz, luego `ast` en `/usr`, y por último busca `mbox` en `/usr/ast`. La búsqueda real se efectúa de componente componente de la ruta, como se ilustró en la Fig. 5-14.

La única complicación es qué sucede cuando se encuentra un sistema de archivos montado. La configuración usual en MINIX y muchos otros sistemas tipo UNIX es tener un sistema de archivos raíz pequeño que contiene los archivos necesarios para iniciar el sistema y efectuar tareas fundamentales de mantenimiento del sistema, y tener la mayor parte de los archivos, incluidos los directorios de usuarios, en un dispositivo aparte montado en `/usr`. Éste es un buen momento para ver cómo se monta un dispositivo. Cuando el usuario teclea el comando

```
mount /dev/hd2c /usr
```

en la terminal, el sistema de archivos contenido en la partición 2 del disco duro se monta encima de `/usr` en el sistema de archivos raíz. Los sistemas de archivos antes y después del montaje muestran en la Fig. 5-32.

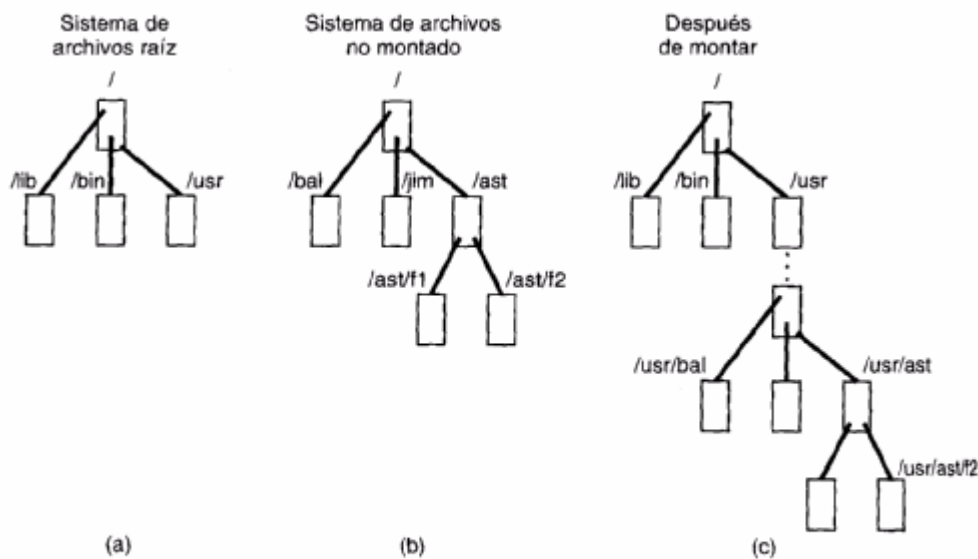


Figura 5-32. (a) Sistema de archivos raíz. (b) Un sistema de archivos no montado. (c) El resultado de montar el sistema de archivos de (b) en `/usr`.

La clave del asunto de montar dispositivos es una bandera que se enciende en la copia del nodo-`i` de `/usr` que está en la memoria después de que se logró montar con éxito el sistema de archivos. Esta bandera indica que el nodo-`i` tiene un sistema de archivos montado en él. La llamada `MOUNT` también carga el superbloque del sistema de archivos recién montado en la `tahsuper_block` y establece dos apuntadores en ella; además, coloca el nodo-`i` raíz del sistema de archivos montado en la tabla `inode`.

En la Fig. 5-29 vemos que los superbloques en la memoria contienen dos campos relacionados con los sistemas de archivos montados. El primero de éstos, *i-node-of-the-mounted-file-system*, se ajusta de modo que apunte al nodo-i raíz del sistema de archivos que se acaba demontar. El segundo, *i-node-mounted-on*, se ajusta de modo que apunte al nodo-i en el cual se montó el sistema, que en este caso es el nodo-i de */usr*. Estos dos apuntadores sirven para conectar el sistema de archivos montado a la raíz y representan el "adhesivo" que mantiene unido del sistema de archivos montado a la raíz [lo que se indica con puntos en la Fig. 5-32(c)j]. Este adhesivo es lo que hace que funcionen los sistemas de archivos montados.

Cuando se está buscando una ruta como */usr/ast/fZ*, el sistema de archivos ve una bandera en el nodo-i de */usr* y se da cuenta de que debe continuar buscando en el nodo-i raíz del sistema de archivos montado en */usr*. La pregunta es, "¿cómo encuentra este nodo-i raíz?"

La respuesta es sencilla. El sistema examina todos los superbloques que están en la memoria hasta encontrar aquel cuyo campo *i-node mounted on* apunta a */usr*. Éste debe ser el superbloque del sistema de archivos montado en */usr*. Una vez que se tiene el superbloque, es fácil seguir el otro apuntador para encontrar el nodo-i raíz del sistema de archivos montado. Ahora el sistema de archivos puede continuar la búsqueda. En este ejemplo, buscará *ast* en el directorio raíz de la partición 2 del disco duro.

5.6.7 Descriptores de archivos

Una vez que se ha abierto un archivo, se devuelve un descriptor de archivo al proceso de usuario para que lo use en las llamadas *READ* y *WRITE* subsecuentes. En esta sección veremos cómo se administran los descriptores de archivo dentro del sistema de archivos.

Al igual que el kernel y el administrador de memoria, el sistema de archivos mantiene parte de la tabla de procesos dentro de su espacio de direcciones. Tres de los campos tienen un interés especial. Los primeros dos son apuntadores a los nodos-i del directorio raíz y del directorio de trabajo. Las búsquedas de rutas, como la de la Fig. 5-14, siempre comienzan en uno o en el otro, dependiendo de si la ruta es absoluta o relativa. Estos apuntadores se modifican con las llamadas al sistema *CHROOT* y *CHDIR* de modo que apunten al nuevo directorio raíz o al nuevo directorio de trabajo, respectivamente.

El tercer campo interesante de la tabla de procesos es un arreglo indizado por número de descriptor de archivo, y sirve para localizar el archivo correcto cuando se presenta un descriptor de archivo. A primera vista, podría parecer que basta con hacer que la *k*-ésima entrada de este arreglo apunte al nodo-i del archivo que pertenece al descriptor de archivo *k*. Después de todo, el *j* nodo-i se trae a la memoria cuando se abre el archivo y se mantiene ahí hasta que el archivo se cierra, así que siempre está disponible.

Desafortunadamente, este sencillo plan falla porque los archivos se pueden compartir de formas sutiles en MINIX (lo mismo que en UNIX). El problema surge porque cada archivo tiene asociado un número de 32 bits que indica el siguiente byte que se leerá o escribirá. Es este número, llamado **posición en el archivo**, lo que se modifica con la llamada al sistema *LSEEK*. El problema puede plantearse fácilmente: "¿dónde debe guardarse el apuntador al archivo?"

La primera posibilidad es colocarlo en el nodo-*i*. Lo malo es que si dos o más procesos tienen el mismo archivo abierto al mismo tiempo, todos deben tener sus propios apuntadores al archivo, ya que difícilmente sería aceptable que un LSEEK efectuado por un proceso afectara la siguiente lectura de un proceso distinto. Conclusión: la posición en el archivo no puede colocarse en el nodo-*i*.

¿Y si la colocamos en la tabla de procesos? ¿Por qué no tener un segundo arreglo, paralelo al arreglo de descriptores de archivo, que indique la posición actual en cada archivo? Esta idea tampoco funciona, pero la razón es más sutil. Básicamente, el problema tiene su origen en la semántica de la llamada al sistema FORK. Cuando un proceso bifurca, tanto el padre como el hijo deben compartir un mismo apuntador que indique la posición actual en cada archivo abierto.

Para entender mejor el problema, consideremos el caso de un guión de shell cuya salida se ha redirigido a un archivo. Cuando el shell bifurca el primer programa, su posición en el archivo para la entrada estándar es 0. Después, el hijo hereda esta posición y escribe, digamos, 1K de salidas. Cuando el hijo termina, la posición compartida en el archivo debe ser 1K.

Ahora el shell lee otro poco del guión de shell y bifurca otro hijo. Es indispensable que el segundo hijo herede del shell una posición en el archivo de 1K, para que comience a escribir en; el lugar en el que se quedó el primer programa. Si el shell no compartiera la posición en el archivo con sus hijos, el segundo programa sobrescribiría las salidas del primero, en lugar de anexar sus salidas alas de él.

En consecuencia, no es posible colocar la posición en el archivo en la tabla de procesos; en realidad debe compartirse. La solución que se emplea en MINIX es introducir una nueva tablar compartida, ^{^//?}, que contiene todas las posiciones en los archivos. Su uso se ilustra en la Fig. 5-3Sj Al compartir realmente la posición en el archivo, la semántica de FORK se puede implementar correctamente, y los guiones de shell funcionan como es debido.

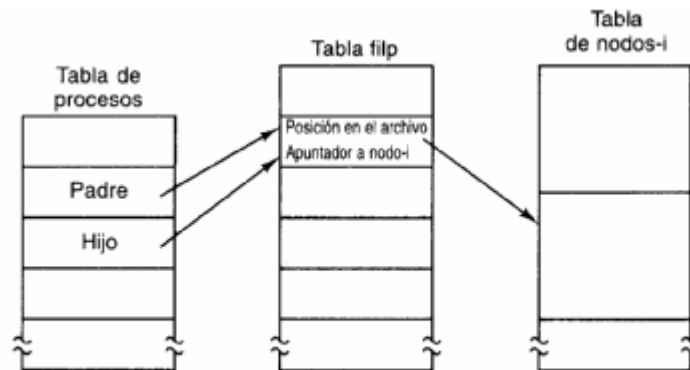


Figura 5-33. Cómo se comparten las posiciones en el archivo entre un padre y un hijo.

Aunque la única cosa que debe contener realmente la tabla filp es la posición en el *i* compartida, es aconsejable colocar también el apuntador al nodo-*i* ahí. De este modo, lo único que tiene el arreglo del descriptor de archivo en la tabla de procesos es un apuntador a una entrada

i/p. La entrada filp también contiene el modo del archivo (bits de permiso), algunas banderas que indican si el archivo se abrió en un modo especial, y una cuenta del número de procesos que lo están usando, para que el sistema de archivos pueda saber cuándo ha terminado el último proceso que estaba usando esa entrada, a fin de recuperar la ranura.

5.6.8 Candados de archivos

Hay un aspecto más de la administración de sistemas de archivos que requiere una tabla especial. Se trata de los candados de archivos. MINIX apoya el mecanismo de comunicación entre procesos POSIX de candados de archivo consultivos. Esto permite que cualquier parte, o varias partes, de un archivo se marquen como "aseguradas". El sistema operativo no obliga a respetar los candados, pero se espera que los procesos se porten bien y determinen si un archivo tiene o no candados antes de hacer algo que pudiera causar un conflicto con otro proceso.

Las razones para contar con una tabla aparte para los candados son similares a las justificaciones de la tabla filp que vimos en la sección anterior. Un solo proceso puede tener más de un candado activo, y diferentes partes de un archivo pueden ser aseguradas por más de un proceso (unque, por supuesto, los candados no pueden traslaparse), así que ni la tabla de procesos ni la iAifilp son buenos lugares para asentar los candados. Puesto que se puede poner más de un candado a un archivo, el nodo-i tampoco es un buen lugar.

MINIX emplea otra tabla, file_lock, para registrar todos los candados. Cada ranura de esta tabla tiene espacio para guardar el tipo del candado, que indica si el archivo está asegurado contra lectura o contra escritura, el identificador del proceso que posee el candado, un apuntador al i del archivo asegurado, y las distancias a las que están el primero y el último byte de la ranura asegurada.

5.9 Conductos y archivos especiales

Los conductos y los archivos especiales difieren de los archivos ordinarios en un aspecto importante. Cuando un proceso trata de leer de, o escribir en, un archivo en disco, no hay duda de que la acción se completará en unos cuantos cientos de milisegundos como máximo. En el peor de los casos, podrían necesitarse dos o tres accesos a disco, no más. Al leer de un conducto, la situación es diferente: si el conducto está vacío, el lector tendrá que esperar hasta que algún otro proceso coloque datos en él, lo cual podría tardar horas. Asimismo, al leer de una terminal, un proceso puede tener que esperar hasta que alguien teclee algo.

Por tanto, la regla normal que tiene el sistema de archivos de atender una solicitud hasta que funciona aquí. Es necesario suspender las solicitudes y reiniciarlas después. Cuando «se trata de leer de, o escribir en, un conducto, el sistema de archivos puede verificar el estado del conducto de inmediato para ver si la operación puede completarse. Si se puede, se ejecuta, pero si no, el sistema de archivos registra los parámetros de la llamada al sistema en la estructura de procesos, a fin de poder reiniciar el proceso cuando llegue el momento.

Cabe señalar que el sistema de archivos no necesita emprender acción alguna para hacer que el proceso se suspenda; lo único que tiene que hacer es no enviar una respuesta, dejando al invocador

bloqueo esperándola. Así pues, después de suspender un proceso, el sistema de archivos regresa a su ciclo principal para esperar la siguiente llamada al sistema. Tan pronto como otro proceso modifica el estado del conducto de modo tal que el proceso suspendido ya puede completar la operación, el sistema de archivos iza una bandera para que en la siguiente iteración del ciclo principal extraiga de la tabla de procesos los parámetros del proceso suspendido y ejecute la llamada.

La situación con las terminales y otros archivos especiales por caracteres es un poco distinta. El nodo-*i* de cada archivo especial contiene dos números, el dispositivo principal y el dispositivo secundario. El número de dispositivo principal indica la clase del dispositivo (p. ej., disco en RAM, disquete, disco duro, terminal) y se usa como índice para consultar una tabla del sistema de archivos que lo transforma en el número de la tarea correspondiente (es decir, controlador de E/S). En efecto, el dispositivo principal determina cuál controlador de E/S debe invocarse. El número de dispositivo secundario se pasa al controlador como parámetro, y especifica cuál dispositivo debe usarse, por ejemplo, la terminal 2 o la unidad de disco 1.

En algunos casos, sobre todo los dispositivos terminales, el número de dispositivo secundario codifica cierta información referente a una categoría de dispositivos manejados por una tarea. Por ejemplo, la consola primaria de MINIX, `/dev/console`, es el dispositivo 4, O (principal, secundario). Las consolas virtuales son manejadas por la misma parte del controlador en software. Éstas son los dispositivos `/dev/ttycl` (4, 1), `/dev/ttyc2` (4, 2), etc. Las terminales de línea serial requieren software de bajo nivel distinto, y estos dispositivos, `/dev/ttyOO` y `/dev/ttyOl` tienen asignados los números de dispositivo 4, 16 y 4, 17. Asimismo, las terminales de red emplean controladores de pseudoterminal, y también necesitan software de bajo nivel diferente. En MINIX se asigna a estos dispositivos, `ttypO`, `jttypl`, etc., números tales como 4, 128 y 4, 129. Cada uno de estos pseudodispositivos tiene asociado un dispositivo, `ptypO`, `ptypl`, etc. Los pares de número de dispositivo principal y secundario para éstos son 4,192,4,193, etc. Se escogen los números de modo que sea fácil para la tarea controladoB;invocar las funciones de bajo nivel requeridas para cada grupo de dispositivos; no se espera que lguien vaya a equipar un sistema MINIX con 192 terminales o más.

Cuando un proceso lee de un archivo especial, el sistema de archivos extrae los números de dispositivo principal y secundario del nodo-*i* del archivo, y utiliza el número de dispositivo principal como índice de una tabla propia para transformarlo en el número de tarea correspondiente. Una vez que tiene el número de tarea, el sistema de archivos envía un mensaje a esa tarea incluyendo como parámetros el número de dispositivo secundario, la operación por realizara número de proceso y dirección de buffer del invocador, y el número de bytes por transferir. El formato es el mismo que el de la Fig. 3-15, excepto que no se usa POSITION.

Si el controlador puede efectuar el trabajo de inmediato (p. ej., si ya se tecleó una línea de entrada en la terminal), copia los datos de sus propios buffers internos al usuario y envía al sistema de archivos un mensaje de respuesta indicándole que se completó el trabajo. El sistema de archivos envía entonces un mensaje de respuesta al usuario, y la llamada termina. Cabe que el controlador no copia los datos en el sistema de archivos. Los datos de dispositivo bloques pasan por el caché de bloques, pero los datos de los archivos especiales por caracte.

Por otro lado, si el controlador no puede llevar a cabo el trabajo, asienta el mensaje en sus tablas internas y de inmediato envía una respuesta al sistema de archivos (dolé que no se pudo completar la llamada. En este punto, el sistema de archivos está en la)

situación en la que está cuando descubre que alguien está tratando de leer de un conducto vacío; toma nota del hecho de que el proceso está suspendido y espera el siguiente mensaje.

Una vez que el controlador ha adquirido suficientes datos para completar la llamada, los transfiere al buffer del usuario que todavía está bloqueado y luego envía al sistema de archivos un mensaje para informarle de lo que hizo. Lo único que tiene que hacer el sistema de archivos es enviar un mensaje de respuesta al usuario para desbloquearlo e informar el número de bytes transferidos.

5.6.10 Un ejemplo: La llamada al sistema READ

Como veremos en breve, la mayor parte del código del sistema de archivos se dedica a llevar a cabo llamadas al sistema. Por tanto, es conveniente concluir esta reseña con un breve bosquejo de cómo funciona la llamada más importante, READ.

Cuando un programa de usuario ejecuta la instrucción

```
n=read(fd, buffer, nbytes);
```

para leer un archivo ordinario, se invoca el procedimiento de biblioteca read con tres parámetros. Este procedimiento construye un mensaje que contiene estos parámetros, junto con el código para READ como tipo de mensaje, envía el mensaje al sistema de archivos, y se bloquea esperando la respuesta. Cuando llega el mensaje, el sistema de archivos usa el tipo de mensaje como índice de sus tablas para obtener el nombre del procedimiento que se encarga de leer, e invocarlo.

Este procedimiento extrae el descriptor de archivo del mensaje y lo utiliza para localizar la entrada filp y luego el nodo-i del archivo que se va a leer (véase la Fig. 5-33). A continuación la solicitud se divide en fragmentos de modo que cada uno quepa en un bloque. Por ejemplo, si la | posición actual en el archivo es 600 y se solicitaron 1K bytes, la solicitud se divide en dos partes, |de600 a 1023 y de 1024 a 1623 (suponiendo bloques de 1K).

Para cada uno de estos fragmentos, se verifica si el bloque pertinente está en el caché. Si el bloque no está presente, el sistema de archivos escoge el buffer menos recientemente utilizado e no está actualmente en uso y lo toma, enviando un mensaje a la tarea del disco para que lo escriba si está sucio. Luego se pide a la tarea de disco que obtenga el bloque que se va a leer. Una vez que el bloque está en el caché, el sistema de archivos envía un mensaje a la tarea del disco pidiéndole que copie los datos en el lugar apropiado del buffer del usuario (es decir, los bytes 600 a 1023 al principio del buffer, y los bytes 1024 a 1623 a una distancia de 424 dentro del buffer). Una vez efectuado el copiado, el sistema de archivos envía un mensaje de respuesta al usuario especificando el número de bytes que se copiaron.

Cuando la respuesta llega al usuario, la función de biblioteca read extrae el código de respuesta y lo devuelve como valor de la función al invocador. Hay un paso más que no forma parte realmente de la llamada READ misma. Una vez que el sistema de archivos completa una lectura y envía una respuesta, inicia una lectura del siguiente archivo, siempre que la lectura se haya hecho en un dispositivo por bloques y que se satisfagan las condiciones adicionales. Puesto que las lecturas de archivos secuenciales son comunes, es razonable esperar que en la siguiente solicitud de lectura se pida el siguiente bloque del archivo, lo que aumenta la probabilidad de que el bloque deseado ya esté en el caché cuando se solicite.

5.7 IMPLEMENTACIÓN DEL SISTEMA DE ARCHIVOS MINIX

El sistema de archivos MINIX es relativamente grande (más de 100 páginas de C) pero muy sencillo. Llegan solicitudes para efectuar llamadas al sistema, se atienden, y se envían respuestas. En las siguientes secciones examinaremos el sistema archivo por archivo, destacando los puntos clave. El código mismo contiene muchos comentarios que ayudarán al lector.

Al examinar el código de otras partes de MINIX generalmente hemos visto primero el ciclo principal de un proceso y luego las rutinas que se encargan de los diferentes tipos de mensajes. Abordaremos el sistema de archivos de forma distinta. Primero estudiaremos los principales subsistemas (administración de caché, administración de nodos-i, etc.), y luego examinaremos el ciclo principal y las llamadas al sistema que operan con archivos. Después veremos las llamadas al sistema que operan con directorios, y por último las llamadas que no pertenecen a ninguna de esas dos categorías.

5.7.1 Archivos de encabezado y estructuras de datos globales

Al igual que con el kernel y el administrador de memoria, varias estructuras de datos y tablas utilizadas en el sistema de archivos se definen en archivos de encabezado. Algunas de estas estructuras de datos se colocan en archivos de encabezado en el nivel del sistema, en `include/` y sus subdirectorios. Por ejemplo, `include/sys/stat.h` define el formato en el cual las llamadas al sistema pueden proporcionar información de nodos-i a otros programas, y la estructura de una entrada de directorio se define en `include/sys/dir.h`. POSIX exige estos dos archivos. Varias definiciones contenidas en el archivo de configuración global `include/minix/config.h` afectan el sistema de archivos, como la macro `ROBUST` que define si las estructuras de datos importantes del sistema de archivos siempre se escribirán de inmediato o no en el disco, y `NR_BUFS` y `NR_BUF_HASH`, que controlan el tamaño del caché de bloques.

Encabezados del sistema de archivos

Los archivos de encabezado propios del sistema de archivos están en el directorio fuente del FS `src/fs/`. Muchos nombres de archivo seguramente resultarán conocidos después de estudiar otras partes del sistema MINIX. El archivo de encabezado maestro del FS, `fs.h` (línea 19400), es muy similar a `src/kernel/kernel.h` y `src/mm/mm.h`; incluye otros archivos de encabezado que todos los archivos fuente en C del sistema de archivos necesitan. Al igual que en las demás partes de MINIX el encabezado maestro del sistema de archivos incluye los archivos `const.h`, `type.h`, `proto.h` y `glo.h` propios de este sistema. Veremos éstos a continuación.

`Const.h` (línea 19500) define algunas constantes, como tamaños de tablas y banderas, que se usan en todo el sistema de archivos. MINIX ya tiene una historia. Una versión anterior tenía un sistema de archivos distinto, así que se proporciona apoyo tanto para el sistema de archivos viejo (V1) como para el actual (V2), en beneficio de los usuarios que deseen acceder a archivos escritos con la versión anterior. El superbloque de un sistema de archivos contiene un número mágico que permite al sistema operativo identificar el tipo; las constantes `SUPER_MAGIC` y

SUPER_V2 definen estos números. El apoyo de versiones antiguas no es algo que se mencione en los textos teóricos, pero siempre es importante para los implementadores de una versión nueva de cualquier software. Es preciso decidir cuánto esfuerzo se dedicará a facilitar la vida de los usuarios de la versión anterior. Veremos varios lugares del sistema de archivos en los que el apoyo de la versión antigua es importante.

Type.h (línea 19600) define las estructuras de los nodos-i tanto viejos (VI) como nuevos (V2), tal como están organizados en el disco. El nodo-i V2 es dos veces más grande que el viejo, que se diseñó pensando en ahorrar espacio en sistemas sin disco duro y disquetes de 360 KB. La nueva versión cuenta con espacio para los tres campos de tiempo que los sistemas UNIX mantienen. En el nodo-i VI había sólo un campo de tiempo, pero un STAT o un FSTAT lo "falsificaba" y devolvía una estructura *stat* que contenía los tres campos. Hay un problema menor para ofrecer apoyo a las dos versiones del sistema de archivos. El comentario de la línea 19616 subraya esto. El software MINIX viejo espera que el tipo *gid_t* sea una cantidad de ocho bits, así que *d2_gid* debe declararse como de tipo *ul6_t*.

Proto.h (línea 19700) proporciona prototipos de funciones en formas aceptables para los compiladores de C tanto antiguos (K&R) como nuevos (ANSÍ Standard). Se trata de un archivo largo, pero no muy interesante. Sin embargo, hay un punto que merece la pena destacar: en vista del gran número de llamadas al sistema distintas que maneja el sistema de archivos, y dada la forma como está organizado el sistema de archivos, las diversas funciones *do_XXX* están dispersas en varios archivos. *Proto.h* está organizado por archivo y es una forma cómoda de encontrar el archivo que se debe consultar cuando se desea ver el código que maneja una llamada al sistema en particular.

Por último, *glo.h* (línea 19900) define variables globales. Los buffers para los mensajes recibidos y de respuesta también están aquí. Se utiliza el truco que ya conocemos bien con la macro *EXTERN*, a fin de que todas las partes del sistema de archivos puedan acceder a estas variables. Al igual que en las demás partes de MINIX, el espacio de almacenamiento se reserva cuando se compila *table.c*.

La parte de la tabla de procesos que corresponde al sistema de archivos está contenida en *fproc.h* (línea 20000). El arreglo *fproc* se declara con la macro *EXTERN*. Este arreglo contiene la máscara de modo, apuntadores a los nodos-i del directorio raíz y el directorio de trabajo actuales, el arreglo de descriptores de archivo, y el uid, gid y número de terminal de cada proceso. El identificador del proceso y el del grupo de procesos también se encuentran aquí. Éstos se duplican en las partes de la tabla de procesos ubicada en el kernel y en el administrador de memoria.

Se emplean varios campos para almacenar los parámetros de las llamadas al sistema que pueden quedar suspendidas a la mitad de su ejecución, como las lecturas de un conducto vacío. Los campos *fp_suspended* y *fp_revived* en realidad sólo requieren un bit cada uno, pero casi todos los compiladores generan mejor código para caracteres que para campos de bits. También hay un campo para los bits *FD_CLOEXEC* que exige el estándar POSIX. Éstos sirven para indicar que se debe cerrar un archivo cuando se efectúa una llamada EXEC.

Ahora llegamos a los archivos que definen otras tablas mantenidas por el sistema de archivos. El primero, *buf.h* (línea 20100), define el caché de bloques. Todas las estructuras de este archivo se declaran con *EXTERN*. El arreglo *¿w/* contiene todos los buffers, cada uno de los cuales comprende

una parte de datos, *b*, y un encabezado lleno de apuntadores, banderas y contadores. La parte de datos se declara como una unión de cinco tipos (línea 20117) porque a veces es aconsejable hacer referencia al bloque como un arreglo de caracteres, a veces como un directorio, etcétera.

La forma correcta de hacer referencia a la parte de datos del buffer 3 como arreglo de caracteres es *buff[3].b.b__data* porque *buff[3].b* se refiere a la unión como un todo, de la cual se selecciona el campo *b__data*. Aunque esta sintaxis es correcta, no es fácil de usar, así que en la línea 20142 se define una macro *b__data* que nos permite escribir *buff[3].b_data* en vez de *buff[3].b.b__data*. Adviértase que *b__data* (el campo de la unión) contiene dos caracteres de subrayado, en tanto que *b_data* (la macro) sólo contiene uno, para distinguirlos. En las líneas 20143 a 20148 se definen macros para otras formas de acceder al bloque.

La tabla *hash* del buffer, *buf_hash*, se define en la línea 20150. Cada entrada apunta a una lista de buffers. Originalmente todas las listas están vacías. Las macros al final de *buf.h* (líneas 20160 a 20166) definen diferentes tipos de bloques. Cuando se devuelve un bloque al caché de buffers después de usarse, se proporciona uno de estos valores para indicarle al administrador del caché si debe colocar el bloque al frente o al final de la lista LRU, y si debe escribirlo en disco inmediatamente o no. El bit *WRITE_IMMED* indica que el bloque se debe reescribir de inmediato en disco si se modificó. El superbloque es la única estructura que se marca incondicionalmente así. ¿Qué hay con las demás estructuras que se marcan con *MAYBE_WRITE_IMMED*? Esto se define en *include/minix/config.h* como igual a *WRITE_IMMED* si *ROBUST* es verdadero, o cero en caso contrario. En la configuración estándar de MINIX, *ROBUST* se define como cero, y estos bloques se escriben cuando se escriben bloques de datos.

Por último, en la última línea (línea 20168) se define *HASH_MASK*, con base en el valor de *NR_BUF_HASH* configurado en *include/minix/config.h*. Se hace un AND de *HASH_MASK* con un número de bloque para determinar cuál entrada *hash_mask* debe usarse como punto de partida cuando se busca un buffer de bloque.

El siguiente archivo, *dev.h* (línea 20200), define la tabla *dmap*. La tabla en sí se declara en *table.c* con valores iniciales, así que esa versión no se puede incluir en varios archivos. Es por esto que se necesita *dev.h*. *Dmap* se declara aquí con *extern*, en lugar de *EXTERN*. La tabla permite transformar un número de dispositivo principal en el nombre de la tarea correspondiente.

File.h (línea 20300) contiene la tabla intermedia *filp* (declarada como *EXTERN*) que sirve para contener la posición actual en el archivo y el apuntador al nodo-i (véase la Fig. 5-33), y que también indica si el archivo se abrió para lectura, escritura o ambas cosas, y cuántos descriptores de archivo están apuntando actualmente a la entrada.

La tabla de candados de archivo, *file_lock* (declarada como *EXTERN*), está en *lock.h* (línea 20400). El tamaño del arreglo lo determina *NR_LOCKS*, que se define como 8 en *const.h*. Este número debe incrementarse si se desea implementar una base de datos multiusuario en un sistema MINIX.

En *inode.h* (línea 20500) se declara la tabla de nodos-i *inode* (empleando *EXTERN*). Esta tabla también contiene los nodos-i que están en uso actualmente. Como ya dijimos, cuando se abre un archivo su nodo-i se trae a la memoria y se mantiene ahí hasta que se cierra el archivo. La definición de la estructura *inode* incluye información que se mantiene en la memoria pero que no se escribe en el nodo-i en disco. Cabe señalar que sólo hay una versión, y que nada es específico

para una versión dada aquí. Cuando se lee el nodo-*i* del disco, se resuelven las diferencias entre los sistemas de archivos V1 y V2. El resto del sistema de archivos no necesita conocer el formato del FS en el disco, al menos en tanto llega el momento de reescribir información modificada.

La mayor parte de los campos no deberán requerir explicación a estas alturas. No obstante, *i_seek* merece algunos comentarios. Ya mencionamos que, como optimación, cuando el sistema de archivos se da cuenta de que un archivo se está leyendo secuencialmente trata de leer bloques y colocarlos en el caché antes de que sean solicitados. En el caso de archivos de acceso aleatorio no hay lectura adelantada. Cuando se efectúa una llamada LSEEK, se ajusta el campo *i_seek* de modo que inhiba la lectura anticipada.

El archivo *param.h* (línea 20600) es análogo al archivo del mismo nombre en el administrador de memoria; define nombres de campos de mensaje que contienen parámetros, de modo que el código pueda hacer referencia a, por ejemplo, *buffer*, en lugar de *m.ml_pl*, que selecciona uno de los campos del buffer de mensaje *m*.

En *super.h* (línea 20700) tenemos la declaración de la tabla de superbloques. Cuando se inicia el sistema, se carga aquí el superbloque del dispositivo raíz. A] montarse sistemas de archivos, sus superbloques también van aquí. Al igual que otras tablas, *super_block* se declara como *EXTERN*.

Asignación de espacio de almacenamiento del sistema de archivos

El último archivo que veremos en esta sección no es un encabezado. Sin embargo, tal como hicimos al estudiar el administrador de memoria, parece apropiado examinar *table.c* inmediatamente después de describir los archivos de encabezado, ya que todos ellos se incluyen cuando se compila *table.c*. La mayor parte de las estructuras de datos que hemos mencionado —el caché de bloques, la tabla/???, etc.—, se definen con la macro *EXTERN*, lo mismo que las variables globales del sistema de archivos y la parte de la tabla de procesos que corresponde al FS. Tal como hemos visto en otras partes del sistema MINIX, el espacio de almacenamiento realmente se reserva cuando se compila *table.c*. Este archivo también contiene dos arreglos importantes inicializados. *Call_vector* contiene el arreglo de apuntadores que se utiliza en el ciclo principal para determinar cuál procedimiento se encarga de cuál número de llamada al sistema. Vimos una tabla similar dentro del administrador de memoria.

Algo que sí es nuevo es la tabla *dmap* de la línea 20914. Esta tabla tiene una fila para cada dispositivo principal, comenzando por el cero. Cuando se abre, cierra, lee o escribe un dispositivo, es esta tabla la que proporciona el nombre del procedimiento que se debe invocar para efectuar la operación. Todos estos procedimientos se encuentran en el espacio de direcciones del sistema de archivos. Muchos de ellos no hacen nada, pero algunos invocan una tarea para solicitar realmente E/S. La tabla también proporciona el número de tarea que corresponde a cada dispositivo principal.

Siempre que se agrega un dispositivo principal a MINIX, se debe agregar una línea a esta tabla que indique cuál acción, en su caso, debe efectuarse cuando se abre, cierra, lee o escribe ese dispositivo. Como ejemplo sencillo, si se agrega una unidad de cinta a MINIX, cuando se abra su archivo especial el procedimiento de la tabla deberá verificar si la unidad de cinta está ya en uso o no. Para ahorrar a los usuarios el trabajo de modificar esta tabla durante una reconfiguración, se define una macro, *DT*, para automatizar el proceso (línea 20900).

Hay una línea en la tabla para cada dispositivo principal posible, y cada línea se escribe con la macro. Los dispositivos obligatorios tienen un 1 como valor del argumento *enable* de la macro. Algunas entradas no se usan, sea porque todavía no está listo un controlador proyectado o porque se eliminó un controlador viejo. Estas entradas se definen con un valor de 0 para *enable*. Las entradas para dispositivos que se pueden configurar en *include/minix/config.h* utilizan la macro habilitadora del dispositivo, por ejemplo, *ENABLE_WINI* en la línea 20920.

5.7.2 Administración de tablas

Cada una de las tablas principales de bloques, nodos-i, superbioques, etc., tiene asociado un archivo que contiene procedimientos que manejan la tabla. El resto del sistema de archivos utiliza intensamente estos procedimientos, que constituyen la principal interfaz entre las tablas y el sistema de archivos. Por esta razón, resulta apropiado iniciar nuestro estudio del código del sistema de archivos con ellos.

Administración de bloques

El caché de bloques se administra con los procedimientos del archivo *cache, c*. Este archivo contiene los nueve procedimientos que se listan en la Fig. 5-34. El primero, *getJblock* (línea 21027) es la forma estándar en que el sistema de archivos obtiene bloques de datos. Cuando un procedimiento del sistema de archivos necesita leer un bloque de datos de usuario, un bloque de directorio, un superbioque o cualquier otro tipo de bloque, invoca *get_block*, especificando el dispositivo y el número de bloque.

Procedimiento	Función
getblock	Trae un bloque para leer o escribir
putblock	Devuelve un bloque solicitado con getblock
alloczone	Asigna una nueva zona (para alargar un archivo)
freezone	Libera una zona (cuando se elimina un archivo)
rwblock	Transfiere un bloque entre el disco y el caché
invalidate	Purga todos los bloques de caché de un dispositivo
flushall	Desaloja todos los bloques sucios de un dispositivo
rwsattered	Lee o escribe datos dispersos en un dispositivo
rmlru	Quita un bloque de su cadena LRU

Figura 5-34. Procedimientos empleados para administrar bloques.

Cuando se invoca *get_block*, primero examina el caché de bloques para ver si el bloque solicitado está ahí. Si es así, *get_block* devuelve un apuntador a él; si no, tiene que leer el bloque. Los bloques del caché se encadenan en *NR_BUF_HASH* listas enlazadas. *NR_BUF_HASHes* un parámetro afinable, junto con *NR_BUFS*, el tamaño del caché de bloques. Ambos se establecen en *include/*

minix/config.h. Al final de esta sección hablaremos un poco sobre cómo optimar el tamaño del caché de bloques y la tabla *hash*. La máscara *HASH_MASK* es *NR_BUF_HASH-1*. Con 256 listas *hash*, la máscara es 255, así que todos los bloques de cada lista tienen números de bloque que terminan con la misma cadena de ocho bits, es decir, 00000000, 00000001, ..., o 11111111.

El primer paso suele ser examinar una cadena *hash* en busca de un bloque, aunque existe un caso especial, cuando se está leyendo un agujero en un archivo poco poblado, en el que se pasa por alto esta búsqueda. Ésta es la razón de la prueba de la línea 21055. En caso contrario, las dos líneas siguientes ajustan *bp* de modo que apunte al principio de la lista en la que estaría el bloque solicitado, si estuviera en el caché, aplicando *HASH_MASK* al número de bloque. El ciclo de la siguiente línea examina esta lista para ver si se puede encontrar el bloque. Si el bloque se encuentra y no está en uso, se retira de la lista LRU; si ya está en uso, no estará en la lista LRÜ de todos modos. El apuntador al bloque encontrado se devuelve al invocador en la línea 21063.

Si el bloque no está en la lista *hash*, no está en el caché, así que se toma el bloque menos recientemente utilizado de la lista LRU. El buffer escogido se retira de su cadena *hash*, ya que está a punto de adquirir un nuevo número de bloque y, por tanto, pertenece a una cadena *hash* distinta. Si el bloque está sucio, se reescribe en el disco en la línea 21095. Al hacerse esto con una llamada *aflushall* se reescriben todos los demás bloques sucios para el mismo dispositivo. Los bloques que actualmente están en uso nunca se escogen para ser desalojados, ya que no están en la cadena LRU. Sin embargo, casi nunca sucederá que los bloques estén en uso; normalmente, *put_block* libera los bloques inmediatamente después de usarlos.

Tan pronto como está disponible el buffer, todos los campos, incluido *b_dev*, se actualizan con los nuevos parámetros (líneas 21099 a 21104) y ya se puede traer el bloque del disco. No obstante, hay dos ocasiones en las que podría no ser necesario leer el bloque del disco. *GetJblock* se invoca con un parámetro *only_search* (sólo buscar). Esto puede indicar que se trata de una preobtención. Durante una preobtención se encuentra un bloque disponible, escribiendo el contenido anterior en el disco si es necesario, y se asigna un nuevo número de bloque al buffer, pero el campo *b_dev* se ajusta a *NO_DEV* para indicar que todavía no hay datos válidos en este bloque. Veremos cómo se usa esto cuando estudiemos la función *rw_scattered*. También puede usarse *only_search* para indicar que el sistema de archivos necesita un bloque sólo para reescribirlo completo. En este caso, es un desperdicio leer primero la versión antigua. En cualquiera de estos dos casos se actualizan los parámetros, pero se omite la lectura en sí del disco (líneas 21107 a 21111). Una vez que se lee el nuevo bloque, *getJblock* regresa a su invocador con un apuntador a dicho bloque.

Supongamos que el sistema de archivos necesita un bloque de directorio temporalmente, a fin de buscar un nombre de archivo. El ES invoca *getJblock* para adquirir el bloque de directorio. Una vez que ha buscado el nombre en cuestión, el ES invoca *putJblock* (línea 21119) para devolver el bloque al caché, dejando el buffer disponible en caso de que se necesite después para un bloque distinto.

Put_block se encarga de colocar el bloque recién devuelto en la lista LRU y, en algunos casos, de reescribirlo en el disco. En la línea 21144 se toma la decisión de colocarlo al frente o al final de la lista LRU, dependiendo de *block_type*, una bandera provista por el invocador que indica el tipo de bloque de que se trata. Los bloques que tal vez se necesitarán otra vez pronto se colocan al final, a fin de que permanezcan durante cierto tiempo en el caché. Los bloques con poca

probabilidad de necesitarse pronto otra vez se colocan al frente, donde se reutilizarán con rapidez. De momento, sólo los superbioques se tratan de esta manera.

Una vez que el bloque se ha reubicado en la lista LRU, se efectúa otra prueba (líneas 21172 y 21173) para determinar si el bloque se debe reescribir en el disco inmediatamente. En la configuración estándar sólo los superbioques se marcan para escritura inmediata, pero la única ocasión en que se modifica un superbioque y es necesario escribirlo es cuando se redimensiona un disco en RAM durante la inicialización del sistema. En tal caso la escritura es en el disco en RAM, y es poco probable que el superbioque de un disco en RAM tenga que volver a leerse alguna vez. Por tanto, esta capacidad casi nunca se usa. Sin embargo, es posible editar la macro *ROBUST* en *include/minix/config.h* de modo que se marquen para escritura inmediata los nodos-i, bloques de directorio y otros bloques que son indispensables para el funcionamiento correcto del sistema de archivos mismo.

Conforme un archivo crece, ocasionalmente es preciso asignar una nueva zona para contener los nuevos datos. El procedimiento *alloc_z.one* (línea 21180) se encarga de asignar zonas nuevas, cosa que hace encontrando una zona libre en el mapa de bits de zonas. No hay necesidad de buscar en el mapa de bits si ésta va a ser la primera zona de un archivo; se consulta el campo *s_zsearch* del superbioque, que siempre apunta a la primera zona disponible en el dispositivo. En los demás casos se intenta encontrar una zona cercana a la última zona existente del archivo actual, a fin de mantener juntas todas las zonas de un archivo. Esto se hace iniciando la exploración del mapa de bits en esta última zona (línea 21203). La transformación del número de bit del mapa de bits en el número de zona se efectúa en la línea 21215, donde el bit 1 corresponde a la primera zona de datos.

Cuando se elimina un archivo, sus zonas deben marcarse como desocupadas en el mapa de bits. *Free_z.one* (línea 21222) se encarga de devolver estas zonas; lo único que hace es invocar *free_bit*, pasándole el mapa de zonas y el número de bit como parámetros. *Free_bit* también sirve para devolver nodos-i, pero sólo cuando se pasa el mapa de nodos-i como primer parámetro, naturalmente.

La administración del caché requiere leer y escribir bloques. A fin de contar con una interfaz sencilla con el disco, se ha incluido el procedimiento *rwJblock* (línea 21243), que lee o escribe un bloque. De forma análoga, *rw_inode* sirve para leer y escribir nodos-i.

El siguiente procedimiento del archivo es *invalidate* (línea 21280). Este se invoca, por ejemplo, cuando se desmonta un disco, para retirar del caché todos los bloques pertenecientes al sistema de archivos que acaba de desmontarse. Si no se hiciera esto, cuando el dispositivo se volviera a utilizar (con un disquete distinto), el sistema de archivos podría encontrar los bloques antiguos en lugar de los nuevos.

Flushall (línea 21295) es invocado por la llamada al sistema SYNC para escribir en disco todos los bloques sucios pertenecientes a un dispositivo dado. Este procedimiento se invoca una vez por cada dispositivo montado, y trata el caché de buffers como arreglo lineal, de modo que se encuentran todos los buffers sucios, incluso los que se están usando actualmente y no están en la lista LRU. Se examinan todos los buffers del caché, y los que pertenecen al dispositivo que se va a actualizar y necesitan escribirse se agregan a un arreglo de apuntadores llamado *dirty*. Este arreglo se declara como *static* para mantenerlo fuera de la pila, y posteriormente se pasa a *rw_scattered*.

Rw_scattered (línea 21313) recibe un identificador de dispositivo, un apuntador a un arreglo de apuntadores a buffers, el tamaño del arreglo y una bandera que indica si se debe leer o escribir. Lo primero que hace este procedimiento es ordenar el arreglo que recibe por número de bloque, de modo que la operación de lectura o escritura se efectúe en un orden eficiente. *Rw_scattered* se invoca con la bandera *WRITING* sólo desde la función *flushall* que acabamos de describir. En este caso el origen de estos números de bloque es fácil de entender: se trata de buffers que contienen datos de bloques que se leyeron antes y que se han modificado. La única llamada a *rw_scattered* para una operación de lectura la efectúa *rabead* en *read.c*. Por ahora, sólo necesitamos saber que antes de la llamada a *rw_scattered* se invocó repetidamente *getJblock* en modo de preobtención, reservando un grupo de buffers. Dichos buffers contienen números de bloque, pero ningún parámetro de dispositivo válido. Esto no es un problema, ya que *rw_scattered* se invoca con un parámetro de dispositivo como uno de sus argumentos.

Hay una diferencia importante en la forma como un controlador de dispositivo puede responder a una solicitud de lectura (en contraposición a una de escritura) hecha por *rw_scattered*. Una solicitud de escritura de bloques *se debe* satisfacer plenamente, pero una solicitud de lectura de varios bloques puede ser manejada de diferente forma por los distintos controladores, dependiendo de lo que resulte más eficiente para el controlador en cuestión. *Rahead* a menudo invoca *rw_scattered* solicitando una lista de bloques que podrían no necesitarse realmente, así que la mejor respuesta es obtener tantos bloques como puedan leerse fácilmente, pero no saltar desesperadamente de un lado a otro de un dispositivo que podría tener un tiempo de búsqueda sustancial. Por ejemplo, el controlador de disquete podría detenerse en una frontera de pista, y muchos otros controladores sólo leen bloques consecutivos. Una vez que se completa la lectura, *rw_scattered* marca los bloques que se leyeron actualizando el campo de número de dispositivo de sus buffers de bloque.

La última función de la Fig. 5-34 es *rm.J.ru* (línea 21387). Esta función, que sirve para quitar un bloque de la lista LRU, sólo es utilizada por *getJblock* en este archivo, así que se declara *PRÍVATE* en lugar de *PUBLIC* a fin de ocultarla de procedimientos externos al archivo.

Antes de dejar el caché de bloques, diremos unas cuantas palabras acerca de su optimización. *NR_BUF_HASH* debe ser una potencia de 2; si es mayor que *NR_BUFS*, la longitud media de la cadena *hash* será menor que 1. Si hay suficiente memoria para una gran cantidad de buffers, hay espacio para un gran número de cadenas *hash*, así que la decisión usual es hacer que *NR_BUF_HASH* sea la siguiente potencia de 2 mayor que *NR_BUFS*. El listado del texto muestra valores de 512 bloques y 1024 listas *hash*. El tamaño óptimo depende de la forma como se utilice el sistema, ya que eso determina cuántos bloques deben colocarse en buffers. Empíricamente, se determinó que un aumento del número de buffers más allá de 1024 no mejoraba el rendimiento al recompilarse el sistema MINIX, así que, al parecer, esta cantidad es suficiente para contener los binarios de todas las pasadas del compilador. Para otro tipo de trabajo podría ser adecuado un tamaño más pequeño, o podría ser que uno más grande mejorara el rendimiento.

Los archivos binarios del sistema MINIX del CD-ROM se compilaron con un caché de bloques mucho más pequeño. Esto se debe a que se pretende que el binario de distribución se pueda ejecutar en el mayor número de máquinas posible. Se buscaba producir una versión de distribución de MINIX que se pudiera instalar en un sistema con sólo 2 MB de memoria RAM. Un sistema compilado con un caché de 1024 bloques requiere más de 2 MB de RAM. El binario que se

distribuye también incluye todos los posibles controladores de disco duro y de otro tipo, que podrían ser inútiles en una instalación dada. La mayoría de los usuarios probablemente querrá editar *include/minix/config.h* y recompilar el sistema poco después de la instalación, excluyendo los controladores innecesarios y agrandando el caché de bloques hasta donde sea posible.

Antes de dejar el tema del caché de bloques, mencionaremos que el límite de 64 KB para el tamaño de los segmentos de memoria en los procesadores Intel de 16 bits imposibilita el uso de un caché grande en esas máquinas. Es posible configurar el sistema de archivos de modo que utilice el disco en RAM como caché secundario para guardar los bloques que no quepan en el caché primario. No estudiaremos tal configuración aquí porque no es necesaria en un sistema Intel de 32 bits: siempre que sea factible, un caché primario grande ofrece el mejor rendimiento. No obstante, un caché secundario puede ser útil en una máquina (como una 286) que no tiene espacio para un caché primario grande dentro del espacio de direcciones virtual del sistema de archivos. Un caché secundario deberá tener un mejor rendimiento que un disco en RAM convencional. Un caché sólo contiene datos que se necesitan al menos una vez, y si tiene el tamaño suficiente puede mejorar mucho el rendimiento del sistema. No es posible definir anticipadamente "el tamaño suficiente"; sólo puede medirse determinando si un aumento adicional en el tamaño da pie a mejoras adicionales en el rendimiento. El comando *time*, que mide el tiempo empleado para ejecutar un programa, es una herramienta útil cuando se trata de optimar un sistema.

Administración de nodos-i

El caché de bloques no es la única tabla que necesita procedimientos de apoyo; la tabla de nodos-i también los necesita. Muchos de los procedimientos tienen una función similar a la que tienen los procedimientos de administración de bloques, y se listan en la Fig. 5-35.

Procedimiento	Función
Get_inode	Trae un nodo_i a la memoria
Put_inode	Devuelve un nodo_i que ya no se necesita
Alloc_inode	Asigna un nuevo nodo_i (para un archivo nuevo)
Wipe_inode	Despeja algunos campos de un nodo_i
Free_inode	Libera un nodo_i (cuando se borra un archivo)
Update_times	Actualiza campos de tiempo de un nodo_i
Rw_inode	Transfiere un nodo_i entre la memoria y el disco
Old_icopy	Convierte contenido de nodo_i para escribir en nodo_i V1
New_icopy	Convierte datos leídos de nodo_i de sistema de archivos V1
Dup_inode	Indica que alguien más está usando un nodo_i

Figura 5-35. Procedimientos empleados para administrar nodos-i.

El procedimiento *get_inode* (línea 21534) es análogo a *get_block*. Cuando una parte del sistema de archivos necesita un nodo-i, invoca *get_inode* para adquirirlo. Lo primero que hace *get_inode*

es examinar la tabla *inode* para ver si el nodo-i ya está presente. Si es así, *get_inode* incrementa el contador de usos y devuelve un apuntador al nodo. Esta búsqueda está contenida en las líneas 21546 a 21556. Si el nodo-i no está presente en la memoria, se carga invocando *rw_inode*.

Una vez que el procedimiento que requería el nodo-i termina de usarlo, el nodo se devuelve invocando el procedimiento *put_inode* (línea 21578), que decrementa el contador de usos *i_count*. Si esta cuenta llega a cero, quiere decir que el archivo ya no se está usando y es posible quitar el nodo-i de la tabla. Si el nodo está sucio, se reescribe en el disco.

Si el campo *i_link* es cero, ninguna entrada de directorio está apuntando al archivo, así que pueden liberarse todas sus zonas. Cabe señalar que la llegada a cero de la cuenta de usos y del número de enlaces son dos sucesos distintos, con diferentes causas y diferentes consecuencias. Si el nodo-i es para un conducto, es preciso liberar todas las zonas, aunque el número de enlaces no sea cero. Esto sucede cuando un proceso que lee de un conducto libera el conducto. No tiene sentido tener un conducto para un solo proceso.

Cuando se crea un archivo nuevo, se debe asignar un nodo-i con *alloc_inode* (línea 21605). MINIX permite montar dispositivos en modo de sólo lectura, así que se examina el superbloque para comprobar que se pueda escribir en el dispositivo. A diferencia de las zonas, donde se intenta mantener cercanas las zonas de un archivo, cualquier nodo-i es igualmente bueno. A fin de ahorrarse el tiempo que tomaría una búsqueda en el mapa de bits de nodos-i, se aprovecha el campo del superbloque en el que se registra el primer nodo-i desocupado.

Una vez adquirido el nodo-i, se invoca *get_inode* para colocar el nodo-i en la tabla que está en la memoria. A continuación se inicializan sus campos, una parte en línea (líneas 21641 a 21648) y una parte usando el procedimiento *wipe_mode* (línea 21664). Se escogió esta división del trabajo en particular porque *wipe_inode* también se necesita en otros puntos del sistema de archivos para despejar ciertos campos de un nodo-i (pero no todos).

Cuando se elimina un archivo, su nodo-i se libera invocando *free_inode* (línea 21684). Lo único que sucede aquí es que el bit correspondiente del mapa de bits de nodos-i se pone en O y se actualiza el campo del superbloque donde se registra el primer nodo-i desocupado.

La siguiente función, *update_times* (línea 21704), se invoca para obtener el tiempo del reloj del sistema y modificar los campos de tiempo que requieran una actualización. Las llamadas al sistema STAT y FSTAT también invocan *update_times*, así que ésta se declara *PUBLIC*.

El procedimiento *rw_inode* (línea 21731) es análogo a *rwJblock*. Su trabajo consiste en traer un nodo-i del disco, cosa que se lleva a cabo ejecutando estos pasos:

1. Calcular qué bloque contiene el nodo-i requerido.
2. Leer el bloque invocando *get_block*.
3. Extraer el nodo-i y copiarlo en la tabla *inode*.
4. Devolver el bloque invocando *putJblock*.

Rw_inode es un poco más complejo que lo que indica esta descripción básica, así que se necesitan algunas funciones adicionales. Primero, dado lo costoso que resulta obtener el tiempo actual, si se requiere modificar los campos de tiempo del nodo-i lo único que se hace es indicarlo

enciendiendo bits del campo *i_update* del nodo-*i* mientras éste está en la memoria. Si este campo es distinto de cero cuando se hace necesario escribir el nodo-*i*, se invoca *update_times* para actualizar los tiempos.

Segundo, la historia de MINIX agrega una complicación: en la versión antigua (VI) del sistema de archivos los nodos-*i* en el disco tienen una estructura diferente que en V2. Dos funciones, *old_ncpy* (línea 21774) y *newJ.copy* (línea 21821) se encargan de las conversiones. La primera efectúa la conversión entre la información de nodo-*i* que está en la memoria y el formato empleado por el sistema de archivos VI. La segunda efectúa la misma conversión para los discos del sistema de archivos V2. Ambas funciones se invocan sólo desde este archivo, así que se declaran *PRIVATE*. Cada función realiza conversiones en ambas direcciones (disco a memoria o memoria a disco). MINIX se ha implementado en sistemas que usan un orden de byte diferente del de los procesadores Intel. Toda implementación emplea el orden de byte nativo en su disco; el campo *sp->native* del superbloque identifica el orden empleado. Tanto *old_ncpy* como *newJ.copy* invocan las funciones *conv1* y *conv4* para intercambiar el orden de los bytes si es necesario.

El procedimiento *dup_inode* (línea 21865) se limita a incrementar la cuenta de usos del nodo-*i*; se invoca cuando un archivo abierto se abre otra vez. La segunda vez que se abre el archivo, no hace falta traer otra vez el nodo-*i* del disco.

Administración de superbloques

El archivo *supere* contiene procedimientos que administran el superbloque y los mapas de bits. Hay cinco procedimientos en el archivo, los cuales se listan en la Fig. 5-36.

Procedimiento	Función
<i>allocbit</i>	Asigna un bit del mapa de zonas o nodos- <i>i</i>
<i>freebit</i>	Libera un bit del mapa de zonas o nodos- <i>i</i>
<i>getsuper</i>	Busca un dispositivo en la tabla de superbloques
<i>mounted</i>	Informa si un nodo- <i>i</i> está en un FS montado (o raíz)
<i>readsuper</i>	Lee un superbloque

Figura 5-36. Procedimientos empleados para administrar el superbloque y los mapas de bits.

Cuando se necesita un nodo-*i* o una zona, se invoca *alloc_inode* o *alhc_zone*, como ya vimos. Estas dos funciones invocan *allocJbit* (línea 21926) para realizar la búsqueda real en el mapa de bits pertinente. La búsqueda implica tres ciclos anidados, a saber:

1. El exterior recorre todos los bloques de un mapa de bits.
2. El de en medio recorre todas las palabras de un bloque.
3. El interior recorre todos los bits de una palabra.

El ciclo de en medio compara la palabra con el complemento a uno de cero, esto es, una palabra completa llena de unos. Si la comparación es verdadera, es que no hay nodos-*i* o zonas libres, y se prueba la siguiente palabra. Cuando se encuentra una palabra con un valor distinto,

esto implica que debe tener por lo menos un bit 0, así que se ingresa en el ciclo interior para encontrar el bit libre (es decir, 0). Si ya se examinaron todos los bloques sin éxito, quiere decir que no hay nodos-*i* o zonas libres, y se devuelve el código *NO_BIT* (0). Las búsquedas como ésta pueden consumir una cantidad considerable de tiempo de procesador, pero el empleo de los campos del superbloque que apuntan al primer nodo-*i* desocupado y la primera zona libre, mismos que se pasan a *alloc_bit* en *origin*, ayuda a acortar dichas búsquedas.

La liberación de un bit es más sencilla que la asignación de un bit, ya que no es necesario buscar. *Free-bit* (línea 22003) calcula cuál bloque del mapa de bits contiene el bit que se debe liberar y pone en cero el bit correcto invocando *get_block*, poniendo en cero el bit en la memoria, e invocando *put_block*.

El siguiente procedimiento, *get_super* (línea 22047), sirve para buscar un dispositivo específico en la tabla de superbloques. Por ejemplo, cuando se va a montar un sistema de archivos es necesario comprobar que no esté montado ya. Esta verificación puede efectuarse pidiendo a *get_super* que encuentre el dispositivo del sistema de archivos. Si no se encuentra el dispositivo, es que el sistema de archivos no está montado.

La siguiente función, *mounted* (línea 22067), sólo se invoca cuando se cierra un dispositivo por bloques. Normalmente, todos los datos de un dispositivo que están en caché se desechan cuando el dispositivo se cierra por última vez. Sin embargo, si el dispositivo está montado, esto no es aconsejable. *Mounted* se invoca con un apuntador al nodo-*i* de un dispositivo, y simplemente devuelve *TRUE* si el dispositivo es el dispositivo raíz o es un dispositivo montado.

Por último, tenemos *read_super* (línea 22088). Esta función es en parte análoga a *rw_block* y *rw_inode*, pero sólo se invoca para leer. Durante el funcionamiento normal del sistema no es necesario escribir superbloques. *Read_super* verifica la versión del sistema de archivos del cual acaba de leerse y efectúa conversiones, si es necesario, a fin de que la copia del superbloque que está en la memoria tenga la estructura estándar aunque se haya leído de un disco con una estructura de superbloque distinta.

Administración de descriptores de archivo

MINIX contiene procedimientos especiales para administrar descriptores de archivo y la tabla *filp* (véase la Fig. 5-33), y están contenidos en el archivo *filedes.c*. Cuando se crea o abre un archivo, se requieren un descriptor de archivo libre y una ranura de *filp* desocupada. El procedimiento *get_fd* (línea 22216) sirve para encontrarlos. Sin embargo, ni el descriptor ni la ranura se marcan como "en uso", porque se deben efectuar muchas verificaciones antes de que se sepa a ciencia cierta que la llamada *CREAT* u *OPEN* va a tener éxito.

Get_filp (línea 22263) sirve para ver si un descriptor de archivo está dentro del intervalo; si es así, se devuelve su apuntador *filp*.

El último procedimiento de este archivo *esfmd_filp* (línea 22277), que se necesita para averiguar si un proceso está escribiendo en un conducto roto (es decir, un conducto que ningún otro proceso ha abierto para lectura). *Find_filp* localiza los lectores potenciales mediante una búsqueda de fuerza bruta de la tabla *filp*. Si no se puede encontrar un lector, es que el conducto está roto y la escritura fracasa.

Candados de archivos

Las funciones de candados de registros de POSIX se muestran en la Fig. 5-37. Se puede poner un candado a una parte de un archivo para lectura y escritura, o sólo para escritura, con una llamada FCNTL que especifique una solicitud *F_SETLK* o *F_SETLKW*. Podemos determinar si existe un candado para una parte de un archivo empleando la solicitud *F_GETLK*.

Operación	Significado
FSETLK	Asegura la región para lectura y escritura
FSETLKW	Asegura la región para escritura
FGETLK	Informa si la región está asegurada

Figura 5-37. Las operaciones de candados de registro consultivos de POSIX. Estas operaciones se solicitan con la llamada al sistema FCNTL.

Sólo hay dos funciones en el archivo *lock.c*. *Lock_op* (línea 22319) es invocada por la llamada al sistema FCNTL con un código para una de las operaciones que se muestran en la Fig. 5-37. *Lock_op* realiza ciertas verificaciones de error para asegurarse de que la región especificada sea válida. Cuando se está poniendo un candado, no debe haber conflicto con otro candado ya existente, y cuando se está quitando un candado no se debe partir a la mitad otro candado existente. Para quitar cualquier candado se invoca la otra función de este archivo, *lock_revive* (línea 22463), la cual despierta todos los procesos que están bloqueados esperando candados. Esta estrategia es un término medio; se requeriría código adicional para determinar exactamente cuáles procesos están esperando que se quite un candado en particular. Los procesos que todavía están esperando un archivo asegurado se bloquearán otra vez cuando se inicien. Este procedimiento se basa en el supuesto de que los candados no se usarán con mucha frecuencia. Si se fuera a construir una base de datos multiusuario de gran tamaño en un sistema MINIX, podría ser aconsejable reimplementar esto.

También se invoca *lock_revive* cuando se cierra un archivo que tiene un candado, cosa que podría ocurrir, por ejemplo, si un proceso se termina antes de que termine de usar un archivo asegurado.

5.7.3 El programa principal

El ciclo principal del sistema de archivos está contenido en el archivo *main.c*, a partir de la línea 22537. Estructuralmente, este ciclo es muy parecido al ciclo principal del administrador de memoria y de las tareas de E/S. La llamada a *get_work* espera que llegue el siguiente mensaje de solicitud (a menos que ahora pueda atenderse un proceso que estaba suspendido en un conducto o terminal). También se establece una variable global, *who*, con el número de ranura del proceso invocador en la tabla de procesos, y otra variable global, *fs_call*, con el número de la llamada al sistema que se va a ejecutar.

Ya de vuelta en el ciclo principal, se establecen tres banderas: *fp* apunta a la ranura del invocador en la tabla de procesos, *super_user* indica si el invocador es el superusuario o no, y

dont_reply se inicializa en *FALSE*. Luego viene la atracción principal: la llamada al procedimiento que lleva a cabo la llamada al sistema. El procedimiento por invocar se selecciona usando *fs_call* como índice del arreglo de apuntadores a procedimientos, *call_vector*.

Una vez que el control regresa al ciclo principal, si *dont_reply* está encendida, se inhibe la respuesta (p. ej., un proceso se bloqueó tratando de leer de un conducto vacío). En caso contrario, se envía una respuesta. La instrucción final del ciclo principal se diseñó de modo que detectara si un archivo se está leyendo secuencialmente y cargara el siguiente bloque en el caché antes de que se solicite, a fin de mejorar el rendimiento.

El procedimiento *get_work* (línea 22572) verifica si se ha revivido algún proceso que antes estaba bloqueado. De ser así, éstos tienen prioridad sobre los mensajes nuevos. Sólo si no hay trabajo interno pendiente el sistema de archivos llama al kernel para obtener un mensaje, en la línea 22598.

Después de completarse la llamada al sistema, con éxito o no, se envía una respuesta al invocador con *reply* (línea 22608). El proceso podría haber sido terminado por una señal, así que se hace caso omiso del código de situación devuelto por el kernel. De todos modos, en un caso así nada se podría hacer.

Funciones de inicialización

El resto de *main.c* consiste en funciones que se usan sólo cuando se inicia el sistema. Antes de que el sistema de archivos ingrese en su ciclo principal, se inicializa invocando *fs_init* (línea 22625), que a su vez invoca varias otras funciones para inicializar el caché de bloques, obtener los parámetros de arranque, cargar el disco en RAM si es necesario y cargar el superbloque del dispositivo raíz. El siguiente paso es inicializar la parte de la tabla de procesos que corresponde al sistema de archivos para todas las tareas y servidores, hasta *init* (líneas 22643 a 22654). Por último, se prueban varias constantes importantes, para ver si tienen sentido, y se envía un mensaje a la tarea de memoria con la dirección de la parte de la tabla de procesos que corresponde al sistema de archivos; el programa *ps* usa esta dirección.

La primera función que *fs_init* invoca es *buf_pool* (línea 22679), que construye las listas enlazadas utilizadas por el caché de bloques. En la Fig. 5-31 se muestra el estado normal del caché de bloques, en el que todos los bloques están enlazados tanto en la cadena LRU como en una cadena *hash*. Puede ser útil ver la forma en que se da la situación de la Fig. 5-31. Inmediatamente después de que *buf_pool* inicializa el caché, todos los buffers están en la cadena LRU, y todos están enlazados en la cadena *hash* cero, como en la Fig. 5-38(a). Cuando se solicita un buffer, y mientras está en uso, tenemos la situación de la Fig. 5-38(b), donde vemos que se ha quitado un bloque de la cadena LRU y ahora está en una cadena *hash* distinta. Normalmente, los bloques se liberan y devuelven a la cadena LRU de inmediato. En la Fig. 5-38(c) se muestra la situación después de que el bloque se ha devuelto a la cadena LRU. Aunque este bloque ya no está en uso, se puede acceder a él otra vez para usar los mismos datos, si es necesario, así que se mantiene en la cadena *hash*. Después de que el sistema ha estado funcionando durante algún tiempo, es de esperar que se habrán usado casi todos los bloques y estarán distribuidos aleatoriamente entre las diferentes cadenas *hash*. En ese momento la cadena LRU se ve como en la Fig. 5-31.

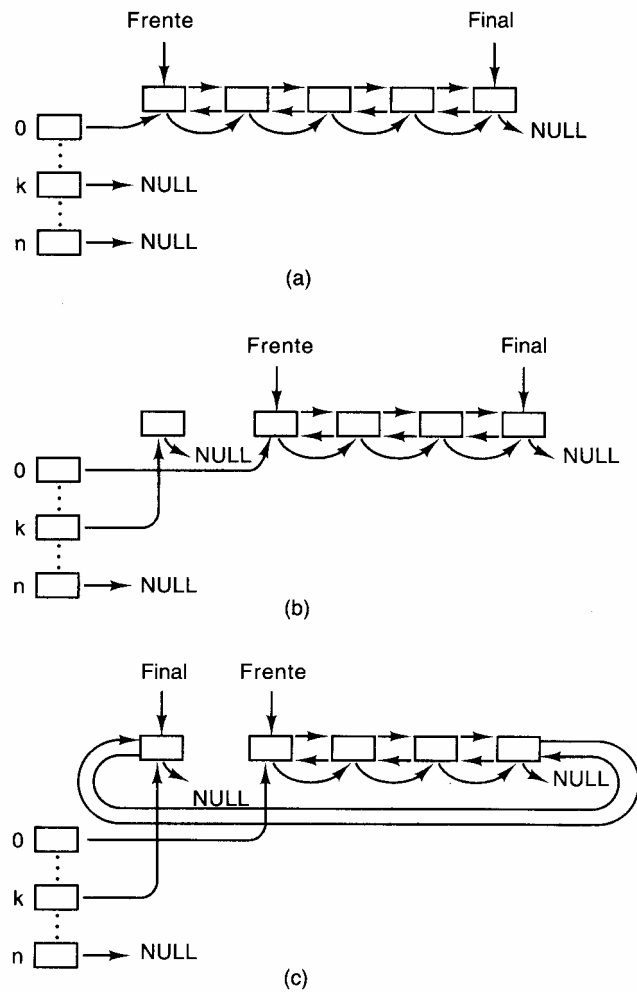


Figura 5-38. Inicialización del caché de bloques. (a) Antes de usarse buffers. (b) Después de solicitarse un bloque. (c) Después de liberarse el bloque.

La siguiente función es *get_boot_parameters* (línea 22706), que envía un mensaje a la tarea del sistema pidiéndole una copia de los parámetros de arranque. La siguiente función, *load_ram* (línea 22722), que asigna espacio para un disco en RAM, necesita esos parámetros. Si los parámetros de arranque especifican

```
rootdev =ram
```

el sistema de archivos del dispositivo raíz se copia del dispositivo nombrado por *ramimagedev* al disco en RAM bloque por bloque, comenzando por el bloque de arranque, sin interpretación de las diversas estructuras de datos del sistema de archivos. Si el parámetro de arranque *ramsize* es menor que el tamaño del sistema de archivos del dispositivo raíz, se hará que el disco en RAM

tenga el tamaño suficiente para contenerlo. Si *ramsize* especifica un tamaño mayor que el del sistema de archivos del dispositivo raíz, se asigna el tamaño especificado y el sistema de archivos del disco en RAM se ajusta de modo que utilice todo el espacio especificado (líneas 22819 a 22825). La llamada *&put_block* de la línea 22825 es el único caso en el que el sistema de archivos escribe un superbioque.

Load_ram asigna espacio para un disco en RAM vacío si se especifica un *ramsize* distinto de cero. En este caso, ya que no se copian estructuras de sistema de archivos, no será posible utilizar el dispositivo en RAM como sistema de archivos en tanto no se inicialice con el comando *mkfs*. Como alternativa, semejante disco en RAM podría utilizarse como caché secundario si en la compilación del sistema de archivos se incluye el apoyo correspondiente.

La última función de *main.c* es *load_super* (línea 22832), que inicializa la tabla de superbioques y trae a la memoria el superbioque del dispositivo raíz.

5.7.4 Operaciones con archivos individuales

En esta sección estudiaremos una por una las llamadas al sistema que operan con archivos individuales (en contraposición a, digamos, operaciones con directorios). Comenzaremos con la creación, apertura y cierre de archivos, y luego examinaremos con cierto detalle el mecanismo de lectura y escritura de archivos. Después, veremos la forma en que los conductos y sus operaciones difieren de las operaciones con archivos.

Cómo crear, abrir y cerrar archivos

El archivo *open.c* contiene el código para seis llamadas al sistema: *CREAT*, *OPEN*, *MKNOD*, *MKDIR*, *CLOSE* y *LSEEK*. Examinaremos *CREAT* y *OPEN* juntas, y luego veremos las demás.

En versiones antiguas de UNIX, las llamadas *CREAT* y *OPEN* tenían distinto propósito. Tratar de abrir un archivo que no existía era un error, y los archivos nuevos tenían que crearse con *CREAT*, que también podía servir para truncar un archivo existente a longitud cero. Sin embargo, en los sistemas POSIX ya no es necesario tener dos llamadas al sistema distintas. En POSIX, la llamada *OPEN* ahora permite crear un archivo nuevo o truncar un archivo viejo, así que la llamada *CREAT* ahora representa un subconjunto de los posibles usos de la llamada *OPEN* y en realidad sólo es necesaria por razones de compatibilidad con programas viejos. Los procedimientos que manejan *CREAT* y *OPEN* son *do_creat* (línea 22937) y *do_open* (línea 22951). (Al igual que en el administrador de memoria, en el sistema de archivos se usa la convención de que la llamada al sistema xxx se lleva a cabo con el procedimiento *do_xxx*.) La apertura o creación de un archivo implica tres pasos:

1. Encontrar el nodo-i (asignarlo e inicializarlo si el archivo es nuevo).
2. Encontrar o crear la entrada de directorio.
3. Preparar y devolver un descriptor de archivo para el archivo.

Tanto *CREAT* como *OPEN* hacen dos cosas: obtienen el nombre de un archivo y luego invocan *common_open* que se ocupa de las tareas comunes a ambas llamadas.

Lo primero que hace *common_open* (línea 22975) es asegurarse de que estén disponibles un descriptor de archivo y una ranura de tabla *filp* libres. Si la función invocadora especificó la creación de un archivo nuevo (llamando con el bit *0_CREAT* encendido), se invoca *new_node* en la línea 22998. *New_node* devuelve un apuntador a un nodo-i existente si la entrada de directorio ya existe; si no, crea tanto una entrada de directorio nueva como un nodo-i. Si no es posible crear el nodo-i, *new_node* establece la variable global *err_code*. Un código de error no siempre implica un error. Si *new_node* encuentra un archivo existente, el código de error devuelto indicará que el archivo existe, pero en este caso ese error es aceptable (línea 23001). Si el bit *0_CREAT* no está encendido, se busca el nodo-i empleando un método alternativo, la función *eat_path* de *path.c*, que comentaremos más adelante. En este punto, lo que debemos entender es que si no se encuentra un nodo-i o no se logra crear, *common_open* terminará con un error antes de llegar a la línea 23010. En los demás casos, la ejecución continuará aquí con la asignación de un descriptor de archivo y la ocupación de una ranura de la tabla *filp*. Después de esto, si se acaba de crear un archivo nuevo, se pasan por alto las líneas 23017 a 23094.

Si el archivo no es nuevo, el sistema de archivos debe determinar de qué tipo de archivo se trata, qué modo tiene, etc., para ver si lo puede abrir. La llamada *aforbidden* en la línea 23018 realiza primero una inspección general de los bits *rxw*. Si el archivo es del tipo normal y *common_open* se invocó con el bit *0_TRUNC* encendido, el archivo se trunca a longitud cero y se invoca *forbidden* otra vez (línea 23024), en esta ocasión para asegurarse de que se puede escribir el archivo. Si los permisos así lo indican, se invocan *wipe_inode* y *rw_inode* para reinicializar el nodo-i y escribirlo en el disco. Otros tipos de archivos (directorios, archivos especiales y conductos con nombre) se someten a pruebas apropiadas. En el caso de un dispositivo, en la línea 23053 se efectúa una llamada (empleando la estructura *dmap*) a la rutina apropiada para abrir el dispositivo. En el caso de un conducto con nombre, se invoca *pipe_open* (línea 23060) y se realizan varias pruebas pertinentes para los conductos.

El código de *common_open*, lo mismo que muchos otros procedimientos del sistema de archivos, contiene una gran cantidad de instrucciones que buscan detectar diversos errores y combinaciones no permitidas. Si bien no es muy elegante, este código es indispensable para tener un sistema de archivos robusto, libre de errores. Si algo anda mal, el descriptor de archivo y la ranura *filp* que se habían asignado se desocupan, y se libera el nodo-i (líneas 23098 a 23101). En este caso el valor devuelto por *common_open* será un número negativo, para indicar un error. Si no hay problemas se devuelve el descriptor de archivo, un valor positivo.

Éste es un buen sitio para analizar con mayor detalle el funcionamiento de *new_node* (línea 23111), que se encarga de la asignación del nodo-i y de la introducción del nombre de ruta en el sistema de archivos para las llamadas *CREAT* y *OPEN*. Las llamadas *MKNOD* y *MKDIR*, que todavía no estudiamos, también usan este procedimiento. La instrucción de la línea 23128 analiza sintácticamente el nombre de ruta (es decir, lo examina componente por componente) hasta llegar al directorio final; la llamada a *advance* tres líneas más adelante trata de averiguar si el componente final se puede abrir.

Por ejemplo, en la llamada

```
fd = creat(7usr/ast/foobar", 0755);
```


last_dir trata de cargar el nodo-i *parsi/usr/ast* en las tablas y devolverle un apuntador. Si el archivo no existe, necesitaremos este nodo-i en breve para agregar *foobar* al directorio. Todas las demás llamadas al sistema que agregan o eliminan archivos utilizan también *last_dir* para abrir primero el último directorio de la ruta.

Si *new_node* descubre que el archivo no existe, invoca *alloc_inode* en la línea 23134 para asignar y cargar un nodo-i nuevo, devolviéndole un apuntador. Si no quedan nodos-i libres, *new_node* falla y devuelve *NIL_INODE*.

Si es posible asignar un nodo-i, la operación continúa en la línea 23144, donde se llenan algunos de los campos, se escribe el nodo-i de vuelta en el disco y se introduce el nombre de archivo en el directorio final (en la línea 23149). Una vez más vemos que el sistema de archivos debe verificar constantemente la presencia de errores y, si encuentra uno, debe liberar cuidadosamente todos los recursos, como nodos-i y bloques, que tiene en su poder. Si estuviéramos dispuestos a dejar simplemente que MINIX entrara en un pánico cuando se agotaran, digamos, los nodos-i, en lugar de anular todos los efectos de la llamada actual y devolver un código de error al invocador, el sistema de archivos sería mucho más sencillo.

Como ya se mencionó, los conductos requieren un tratamiento especial. Si no hay por lo menos un par lector/escritor para un conducto, *pipe_open* (línea 23176) suspende el invocador;

en caso contrario, invoca *reléase*, que examina la tabla de procesos en busca de procesos que están bloqueados esperando el conducto. Si se encuentra alguno, se le revive.

La llamada MKNOD se maneja con *do_mknod* (línea 23205). Este procedimiento es similar a *do_creat*, excepto que sólo crea el nodo-i y una entrada de directorio para él. De hecho, la mayor parte del trabajo corre por cuenta de la llamada a *new_node* en la línea 23217. Si el nodo-i ya existe, se devuelve un código de error. Este es el mismo código que era un resultado aceptable de *new_node* cuando *common_open* lo invocaba; en este caso, empero, el código de error se devuelve al invocador, que es de suponer actuará de manera acorde. No necesitamos efectuar aquí el análisis caso por caso que hicimos cuando describimos *common_open*.

La llamada MKDIR se maneja con la función *do_mkdir* (línea 23226). Al igual que en las otras llamadas al sistema que hemos estudiado aquí, *new_node* desempeña un papel importante. Los directorios, a diferencia de los archivos, siempre tienen enlaces y nunca están completamente vacíos porque todo directorio debe contener dos entradas desde el momento de su creación: las entradas "." y ".." que se refieren al directorio mismo y a su directorio padre. Existe un límite para el número de enlaces que un archivo puede tener: *LINK_MAX* (definido en *include/limits.h* como 127 para el sistema MINIX estándar). Puesto que la referencia al directorio padre en un hijo es un enlace con el padre, lo primero que *do_mkdir* hace es ver si es posible crear otro enlace en el directorio padre (línea 23240). Una vez que se pasa esta prueba, se invoca *new_node*. Si *new_node* tiene éxito, se crean las entradas de directorio para "." y ".." (líneas 23261 y 23262). Todo esto es sencillo, pero podría haber fallas (por ejemplo, si el disco está lleno), y para evitar un embrollo se toman providencias para anular las etapas iniciales del proceso si no es posible completarlo.

Cerrar un archivo es más fácil que abrirlo. El trabajo corre por cuenta de *do_close* (línea 23286). Los conductos y archivos especiales requieren algo de atención, pero en el caso de los archivos normales casi lo único que hay que hacer es decrementar el contador *filp* y verificar si es cero, en cuyo caso se devolverá el nodo-i con *putJinode*. El paso final consiste en quitar

cyalesquier candados y revivir cualquier proceso que haya estado suspendido esperando que se quitara un candado del archivo.

Cabe señalar que la devolución de un nodo-i implica que su contador en la tabla *inode* se decrementa, a fin de poder quitarlo de la tabla posteriormente. Esta operación nada tiene que ver con la liberación del nodo-i (es decir, ajustar un bit en el mapa de bits para indicar que ya está disponible). El nodo-i sólo se libera cuando el archivo se elimina de todos los directorios.

El procedimiento final de *open.c* es *do_lseek* (línea 23367). Cuando se efectúa una búsqueda en un archivo, se invoca este procedimiento para asignar un nuevo valor a la posición en el archivo. En la línea 23394 se inhibe la lectura adelantada; un intento explícito de colocarse en una posición determinada en un archivo es incompatible con el acceso secuencial.

Lectura de un archivo

Una vez que se ha abierto un archivo, se puede leer o escribir. Se emplean muchas funciones durante la lectura y la escritura, y están contenidas en el archivo *read.c*. Explicaremos éstas primero y luego pasaremos al siguiente archivo, *write.c*, para examinar el código que se emplea específicamente en la escritura. Hay varias diferencias entre la lectura y la escritura, pero también suficientes similitudes como para que lo único que *do_read* (línea 23434) tenga que hacer sea invocar el procedimiento común *read_write* con una bandera puesta en *READING*. Veremos en la siguiente sección que *do_write* es igual de sencilla.

Read_write se inicia en la línea 23443. Hay cierto código especial en las líneas 23459 a 23462 que el administrador de memoria utiliza para hacer que el sistema de archivos cargue para él segmentos enteros en el espacio de usuario. Las llamadas normales se procesan a partir de la línea 23464. Luego se efectúan algunas verificaciones de validez (p. ej., leer de un archivo que se abrió sólo para escritura) y se inicializan algunas variables. Las lecturas de archivos especiales por caracteres no pasan por el caché de bloques, así que se excluyen en la línea 23498.

Las pruebas de las líneas 23507 a 23518 sólo aplican a las escrituras y tienen que ver con archivos que podrían crecer a más de la capacidad del dispositivo, o escrituras que van a cerrar un agujero en el archivo al escribir *más allá* del fin del archivo. Como vimos cuando hablamos de las generalidades de MINIX, la presencia de múltiples bloques por zona causa problemas que se deben resolver explícitamente. Los conductos también son especiales y se verifica si se trata de un conducto.

El corazón del mecanismo de lectura, al menos en el caso de los archivos ordinarios, es el ciclo que comienza en la línea 23530. Este ciclo divide la solicitud en trozos, cada uno de los cuales cabe en un solo bloque de disco. Un trozo principia en la posición actual y se extiende hasta que se satisface una de las siguientes condiciones:

1. Ya se leyeron todos los bytes.
2. Se llegó a una frontera de bloque.
3. Se llegó al fin del archivo.

Estas reglas implican que un trozo nunca requiere dos bloques de disco para ser satisfecho. En la Fig. 5-39 se muestran tres ejemplos de cómo se determina el tamaño de trozo, para tamaños de trozo de 6, 2 y 1 bytes, respectivamente. El cálculo en sí se efectúa en las líneas 23632 a 23641.

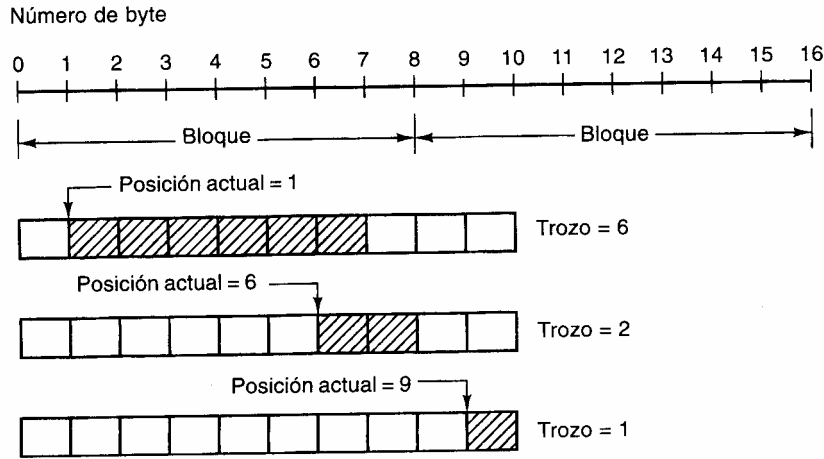


Figura 5-39. Tres ejemplos de cómo se determina el primer tamaño de trozo para un archivo de 10 bytes. El tamaño de bloque es de 8 bytes, y el número de bytes solicitados es 6. El trozo se muestra sombreado.

La lectura real del trozo corre por cuenta de *rw_chunk*. Cuando el control regresa, se incrementan varios contadores y apuntadores y se inicia la siguiente iteración. Cuando el ciclo termina, es posible que se actualicen la posición en el archivo y otras variables (p. ej., apuntadores de conductos).

Por último, si procede la lectura anticipada, el nodo-*i* del cual se va a leer y la posición en la cual se va a comenzar a leer se almacenan en variables globales para que, una vez que se haya enviado el mensaje de respuesta al usuario, el sistema de archivos pueda ponerse a trabajar en la obtención del siguiente bloque. En muchos casos el sistema de archivos se bloquea esperando el siguiente bloque de disco, y durante ese tiempo el proceso de usuario puede trabajar con los datos que ya recibió. Esta organización traslapa el procesamiento y la E/S y puede mejorar el rendimiento sustancialmente.

El procedimiento *rw_chunk* (línea 23613) se ocupa de tomar un nodo-*i* y una posición en el archivo, convertirlos en un número de bloque de disco físico, y solicitar la transferencia de ese bloque (o una porción de él) al espacio de usuario. La transformación de la posición relativa dentro del archivo en la dirección física de disco corre por cuenta de *read_map*, que tiene conocimiento de los nodos-*i* y los bloques de indirección. En el caso de un archivo ordinario, las variables *b* y *dev* de las líneas 23637 y 23638 contienen el número de bloque físico y el número de dispositivo, respectivamente. La llamada a *get_block* de la línea 23660 es donde se pide al manejador del caché que encuentre el bloque y lo lea si es necesario.

Una vez que se tiene un apuntador al bloque, la llamada a *sys_copy* de la línea 23670 se ocupa de transferir la porción requerida de él al espacio de usuario. A continuación se libera el bloque con *put_block*, de modo que pueda ser desalojado del caché posteriormente, cuando llegue el momento. (Después de ser adquirido por *get_block*, el bloque no estará en la cola LRU y no se devolverá a dicha cola en tanto el contador del encabezado del bloque indique que está en uso, así que no será susceptible de desalojo; *put_block* decrementa el contador y devuelve el bloque a la cola LRU cuando el contador llega a cero.) El código de la línea 23680 indica si una operación de escritura llenó el bloque. Sin embargo, el valor que se pasa a *put_block* en *n* no afecta la forma como el bloque se coloca en la cola; todos los bloques se colocan ahora al final de la cadena LRU.

Read_map (línea 23689) convierte una posición lógica en el archivo en un número de bloque físico inspeccionando el nodo-i. Si los bloques están lo bastante cerca del principio del archivo como para quedar dentro de una de las primeras siete zonas (las que están ahí mismo en el nodo-i), basta un cálculo sencillo para determinar qué zona se necesita, y luego qué bloque. En el caso de archivos más adelante en el archivo, podría ser necesario leer uno o más bloques de indirección.

Se invoca *rd_indir* (línea 23753) para leer un bloque de indirección. Existe un procedimiento aparte para hacer esto porque los datos pueden tener diferentes formatos en el disco, dependiendo de la versión del sistema de archivos y del hardware en el que se haya escrito el sistema de archivos. Las conversiones se efectúan aquí, si es necesario, para que el resto del sistema de archivos vea los datos en una sola forma.

Read_ahead (línea 23786) convierte la posición lógica en un número de bloque físico, invoca *get_block* para asegurarse de que el bloque esté en el caché (o se coloque en él) y luego devuelve el bloque inmediatamente. Después de todo, *read_ahead* no puede hacer nada con el bloque; sólo trata de mejorar la probabilidad de que el bloque esté en el caché si se utiliza pronto.

Cabe señalar que *read_ahead* sólo se invoca desde el ciclo principal de *main*; no se invoca como parte del procesamiento de la llamada al sistema READ. Es importante darse cuenta de que la llamada a *read_ahead* se efectúa después de haberse enviado la respuesta, con objeto de que el usuario pueda seguir ejecutándose incluso si el sistema de archivos tiene que esperar un bloque de disco mientras lee anticipadamente.

Read_ahead en sí está diseñado para pedir un solo bloque; invoca a la última función de *read.c*, *rahead*, para efectuar realmente el trabajo. *Rahead* (línea 23805) funciona según la teoría de que si un poco más es bueno, mucho más es mejor. Puesto que los discos y otros dispositivos de almacenamiento tardan un tiempo relativamente largo en localizar el primer bloque solicitado pero después pueden leer con relativa rapidez varios bloques adyacentes, podría ser posible tener muchos más bloques leídos con muy poco esfuerzo adicional. Se presenta una solicitud de preobtención a *get_block*, que prepara el caché de bloques para recibir varios bloques juntos. Luego se invoca *rw_scattered* con una lista de bloques. Ya explicamos esto antes; recuérdese que cuando *rw_scattered* invoca realmente los controladores de dispositivos, cada uno está en libertad de satisfacer sólo la parte de la solicitud que puede manejar con eficiencia. Todo esto suena muy complicado, pero las complicaciones hacen posible una agilización significativa de las aplicaciones que leen grandes cantidades de datos de disco.

En la Fig. 5-40 se muestran las relaciones entre algunos de los principales procedimientos que intervienen en la lectura de un archivo; en particular, quién invoca a quién.

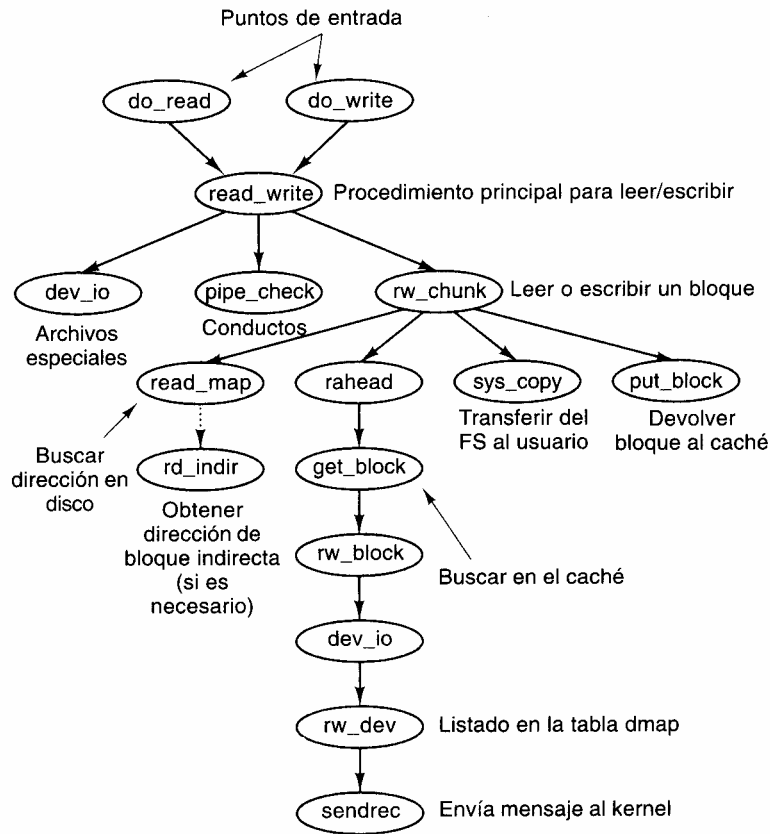


Figura 5-40. Algunos de los procedimientos que intervienen en la lectura de un archivo.

Escritura de un archivo

El código para escribir en archivos está en *write.c*. La escritura de un archivo es similar a su lectura, y *do_write* (línea 24025) simplemente invoca *read_write* con la bandera *WRITING*. Una diferencia importante entre la lectura y la escritura es que esta última requiere la asignación de nuevos bloques de disco. *Write_map* (línea 24036) es análoga a *read_map*, sólo que en lugar de buscar números de bloques físicos en el nodo-i y en sus bloques de indirección, introduce números nuevos ahí (para ser precisos, introduce números de zona, no de bloque).

El código de *write_map* es largo y detallado porque debe manejar varios casos. Si la zona por insertar está cerca del principio del archivo, simplemente se inserta en el nodo-i (línea 24058).

El peor caso es cuando un archivo excede el tamaño que se puede manejar con un bloque de indirección sencilla, y se requiere un bloque de doble indirección. A continuación, será necesario asignar un bloque de indirección sencilla y colocar su dirección en el bloque de doble indirección. Al igual que en la lectura, se invoca un procedimiento aparte, *wr_indir*. Si el bloque de doble

indirección se adquiere correctamente, pero el disco está lleno y no es posible asignar el bloque de indirección sencilla, se deberá devolver el bloque doble para evitar corromper el mapa de bits.

Una vez más, si simplemente pudiéramos tirar la toalla y abandonamos al pánico en este punto, el código sería mucho más sencillo. Sin embargo, desde el punto de vista del usuario es mucho más agradable que el agotamiento de espacio en disco simplemente devuelva un error de WRITE, en lugar de causar una caída del computador con un sistema

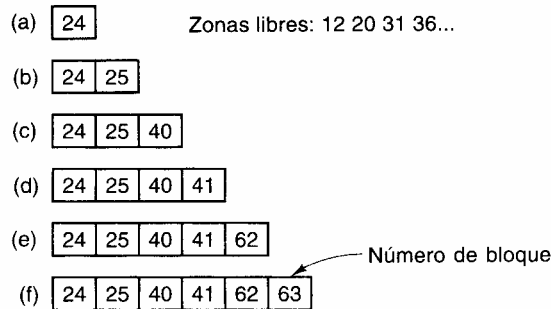


Figura 5-41. (a) - (f) La asignación sucesiva de bloques de 1K con una zona de 2K.

de archivos inconsistente.

Wr_indir (línea 24127) invoca una de las rutinas de conversión, *conv2* o *conv4*, para realizar las conversiones de datos necesarias, y coloca un nuevo número de zona en un bloque de indirección. Tenga presente que el nombre de esta función, al igual que muchas otras funciones que tienen que ver con lectura y escritura, no es literalmente correcto. La escritura real en el disco corre por cuenta de las funciones que mantienen el caché de bloques.

El siguiente procedimiento de *write.c* es *clear_z.one* (línea 24149), que se ocupa del problema de borrar bloques que repentinamente están en medio de un archivo. Esto sucede cuando se efectúa una búsqueda más allá del final de un archivo, seguida de la escritura de datos nuevos. Por fortuna, esta situación no ocurre con demasiada frecuencia.

New_block (línea 24190) es invocada por *rw_chunk* cada vez que se necesita un nuevo bloque. En la Fig. 5-41 se muestran seis etapas sucesivas del crecimiento de un archivo secuencial. El tamaño de bloque es de 1K y el de zona es de 2K en este ejemplo.

La primera vez que se invoca *new_block*, asigna la zona 12 (bloques 24 y 25). En la siguiente ocasión, *new_block* usa el bloque 25, que ya se asignó pero todavía no está en uso. En la tercera llamada se asigna la zona 20 (bloques 40 y 41), y así sucesivamente. *Zero_block* (línea 24243) despeja un bloque, borrando su contenido anterior. Esta descripción es mucho más larga que el código en sí.

Conductos

Los conductos son similares a los archivos ordinarios en muchos sentidos. En esta sección nos concentraremos en las diferencias. Todo el código que describiremos está en *pipe.c*.

En primer lugar, los conductos se crean de forma diferente, con la llamada PIPE, no con CREAT. La llamada PIPE se ejecuta con *do_pipe* (línea 24332). Lo único que *do_pipe* hace realmente es asignar un nodo-i para el conducto y devolver dos descriptores de archivo para él. Los conductos son propiedad del sistema, no del usuario, y se encuentran en el dispositivo de conductos designado (configurado en *include/minix/config.h*), que bien podría ser un disco en RAM, ya que los datos de los conductos no se tienen que conservar permanentemente.

La lectura y escritura de un conducto es un poco diferente de la lectura y escritura de un archivo, porque un conducto tiene capacidad finita. Un intento de escritura en un conducto que ya está lleno causa la suspensión del escritor. Asimismo, la lectura de un conducto vacío suspende al lector. Efectivamente, un conducto tiene dos apuntadores, la posición actual (utilizada por los lectores) y el tamaño (utilizado por los escritores), para determinar de dónde vienen o adónde van los datos.

Pipe_check (línea 24385) efectúa las diversas pruebas para comprobar si una operación con un conducto es posible. Además de las pruebas mencionadas, que pueden dar pie a la suspensión del invocador, *pipe_check* invoca *reléase* para ver si un proceso que antes se había suspendido por falta o exceso de datos ya puede revivirse. Estas reactivaciones se realizan en la línea 24413 y en la línea 24452, para escritores y lectores dormidos, respectivamente. También se detecta aquí la escritura en un conducto roto (sin lectores).

El acto de suspender un proceso corre por cuenta de *suspend* (línea 24463). Lo único que hace esta función es guardar los parámetros de la llamada en la tabla de procesos y poner la bandera *dont_reply* en *TRUE* para inhibir el mensaje de respuesta del sistema de archivos.

El procedimiento *reléase* (línea 24490) se invoca para ver si un proceso que estaba suspendido esperando un conjunto ya puede continuar. Si *reléase* encuentra uno, invoca *revive* para izar una bandera y que el ciclo principal se fije en él después. Esta función no es una llamada al sistema, pero se lista en la Fig. 5-27(c) porque usa el mecanismo de transferencia de mensajes.

El último procedimiento de *pipe. c* es *doJunpause* (línea 24560). Cuando el administrador de memoria está tratando de enviar una señal a un proceso, debe averiguar si ese proceso está suspendido esperando un conducto o un archivo especial (en cuyo caso deberá despertársele con un error *EINTR*). Puesto que el administrador de memoria nada sabe acerca de los conductos ni de los archivos especiales, envía un mensaje al sistema de archivos para preguntarle. Ese mensaje es procesado por *do_unpause*, que revive el proceso si está bloqueado. Al igual que *revive*, *do_unpause* tiene cierto parecido con las llamadas al sistema, aunque no es una.

5.7.5 Directorios y rutas

Ya terminamos de ver cómo se leen y escriben archivos. Nuestra siguiente tarea es ver cómo se manejan los nombres de ruta y los directorios.

Conversión de una ruta en un nodo-i

Muchas llamadas al sistema (p. ej., OPEN, UNLINK y MOUNT) tienen nombres de ruta (es decir, nombres de archivo) como parámetros. En su mayor parte, estas llamadas deben obtener el nodo-i del

archivo nombrado antes de que puedan ponerse a trabajar en la llamada misma. La forma en que un nombre de ruta se convierte en un nodo-i es un tema que ahora vamos a examinar con detalle. Ya vimos un bosquejo general en la Fig. 5-14.

El análisis sintáctico de los nombres de ruta se efectúa en el archivo *path.c*. El primer procedimiento, *eat_path* (línea 24727), acepta un apuntador a un nombre de ruta, lo analiza, hace que su nodo-i se cargue en la memoria y devuelve un apuntador al nodo-i. *Eat_path* efectúa su trabajo invocando *last_dir* para obtener el nodo-i del último directorio e invocando después *advance* para obtener el componente final de la ruta. Si la búsqueda fracasa, por ejemplo porque uno de los directorios de la ruta no existe, o existe pero está protegido contra búsquedas, se devuelve *NIL_INODE* en lugar de un apuntador al nodo-i.

Los nombres de ruta pueden ser absolutos o relativos y pueden tener una cantidad arbitraria de componentes, separados por diagonales. *Last_dir* (línea 24754) se ocupa de estas cuestiones; comienza (línea 24771) por examinar el primer carácter del nombre de ruta para ver si se trata de una ruta absoluta o relativa. Si la ruta es absoluta, se ajusta *rip* de modo que apunte al nodo-i raíz; si es relativa, se hace que *rip* apunte al nodo-i del directorio de trabajo actual.

En este punto, *last_dir* tiene el nombre de ruta y un apuntador al nodo-i del directorio en el que debe buscar el primer componente. Ahora la función ingresa en un ciclo en la línea 24782 para analizar sintácticamente el nombre de ruta, componente por componente. Cuando *last_dir* llega al final, devuelve un apuntador al último directorio.

Get_name (línea 24813) es un procedimiento de utilidad que extrae componentes de cadenas. Más interesante es *advance* (línea 24855), que recibe como parámetros un apuntador a un directorio y una cadena, y busca la cadena en el directorio. Si encuentra la cadena, *advance* devuelve un apuntador a su nodo-i. Los detalles de la transferencia de un sistema de archivos montado en otro se manejan aquí.

Aunque *advance* controla la búsqueda de la cadena, el cotejo en sí de la cadena contra las entradas de directorio se efectúa en *search_dir* (línea 24936), que es el único lugar del sistema de archivos en el que se examinan realmente archivos de directorio. Este procedimiento contiene dos ciclos anidados, uno para iterar sobre los bloques de un directorio y otro para iterar sobre las entradas de un bloque. También se usa *search_dir* para introducir nombres en directorios y borrarlos de ellos. En la Fig. 4-52 se muestran las relaciones entre algunos de los procedimientos más importantes que se emplean en la búsqueda de nombres de ruta.

Montado de sistemas de archivos

Dos llamadas al sistema que afectan el sistema de archivos globalmente son *MOUNT* y *UMOUNT*, las cuales permiten "pegar" sistemas de archivos independientes situados en dispositivos secundarios distintos para formar un solo árbol de nombres sin fronteras. El montado, como vimos en la Fig. 5-32, se logra efectivamente leyendo el nodo-i raíz y el superbloque del sistema de archivos que se va a montar y estableciendo dos apuntadores en su superbloque. Uno de ellos apunta al nodo-i en el que se va a montar, y el otro apunta al nodo-i raíz del sistema de archivos montado. Estos apuntadores "enganchan" los dos sistemas de archivos.

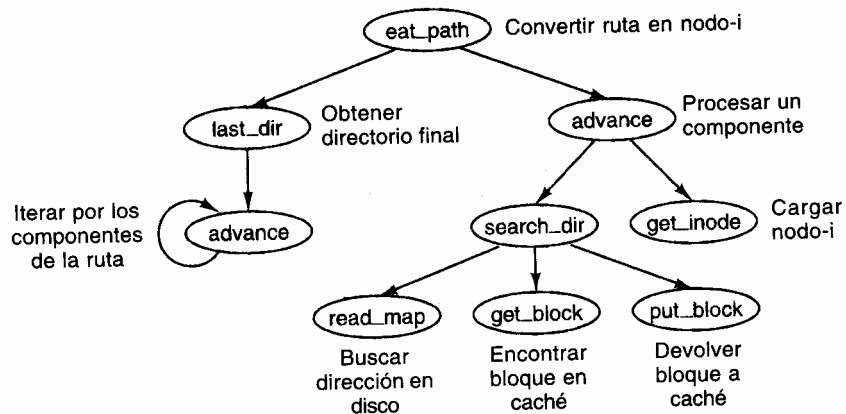


Figura 5-42. Algunos de los procedimientos que se emplean en la búsqueda de nombres de ruta.

El establecimiento de estos apuntadores se efectúa en el archivo *mount.c* mediante *do_mount* en las líneas 25231 y 25232. Las dos páginas de código que preceden al establecimiento de los apuntadores se ocupan casi exclusivamente de verificar todos los errores que pueden ocurrir mientras se monta un sistema de archivos. Entre ellos están:

1. El archivo especial dado no es un dispositivo por bloques.
2. El archivo especial dado es un dispositivo por bloques pero ya está montado.
3. El sistema de archivos por montar tiene un número mágico corrompido.
4. El sistema de archivos por montar no es válido (p. ej., no tiene nodos-i).
5. El archivo en el que se va a montar no existe o es un archivo especial.
6. No hay espacio para los mapas de bits del sistema de archivos montado.
7. No hay espacio para el superbloque del sistema de archivos montado.
8. No hay espacio para el nodo-i raíz del sistema de archivos montado.

Tal vez parezca inapropiado seguir insistiendo en este punto, pero la realidad de cualquier sistema operativo es que una fracción sustancial del código se dedica a realizar tareas menores que no son muy estimulantes intelectualmente pero que son cruciales para que el sistema se pueda usar. Si un usuario intenta montar el disco flexible equivocado por accidente, digamos, una vez al mes, y esto conduce a una caída y a un sistema de archivos inconsistente, el usuario pensará que el sistema no es confiable y culpará al diseñador, no a sí mismo.

Thomas Edison hizo alguna vez una observación que es pertinente aquí. Dijo que el "genio" es 1% inspiración y 99% transpiración. La diferencia entre un sistema bueno y uno mediocre no es lo brillante del algoritmo de planificación del primero, sino su atención a los detalles.

Desmontar un sistema de archivos es más fácil que montarlo, pues hay menos cosas que pueden salir mal. *Do_unmount* (línea 25241) se encarga de esto. La única cuestión importante aquí es asegurarse de que ningún proceso tenga archivos abiertos o directorios de trabajo en el sistema que se va a quitar. Esta verificación es directa: basta con examinar toda la tabla de nodos-i para ver si algún nodo que está en memoria pertenece al sistema de archivos que se va a desmontar (aparte del nodo-i raíz). Si lo hay, la llamada *UMOUNT* falla.

El último procedimiento de *mount.c* es *name_to_dev* (línea 25299), que toma el nombre de ruta de un archivo especial, obtiene su nodo-i y extrae sus números de dispositivo principal y secundario. Éstos se almacenan en el nodo-i mismo, en el lugar donde normalmente iría la primera zona. Esta ranura está disponible porque los archivos especiales no tienen zonas.

Enlazamiento y desenlazamiento de archivos

El siguiente archivo por considerar es *link.c*, que se ocupa de enlazar y desenlazar archivos. El procedimiento *do_link* (línea 25434) es muy parecido a *do_mount* en cuanto a que casi todo el código se ocupa de la verificación de errores. Algunos de los errores que pueden ocurrir en la llamada

```
link(tile_name, link_name);
```

se listan a continuación:

1. *Nombre_archivo* no existe o no es posible acceder a él.
2. *Nombre_archivo* ya tiene el número máximo de enlaces.
3. *Nombre_archivo* es un directorio (sólo el superusuario puede crear enlaces a él).
4. *Nombre_enlace* ya existe.
5. *Nombre_archivo* y *nombre_enlace* están en dispositivos distintos.

Si no hay errores, se crea una nueva entrada de directorio con la cadena *nombre_enlace* y el número de nodo-i de *nombre_archivo*. En el código, *namel* corresponde a *nombre_archivo* y *name2* corresponde a *nombre_enlace*. *Search_dir* crea la entrada real al ser invocado desde *do_link* en la línea 25485.

Los archivos y directorios se eliminan desenlazándolos. *Do_unlink* (línea 25504) realiza el trabajo de las dos llamadas al sistema *UNLINK* y *RMDIR*. Una vez más, es preciso efectuar varias verificaciones; la prueba de que un archivo existe y de que un directorio no es un punto de montaje son realizadas por el código común de *do_unlink*, y luego se invoca ya sea *remove_dir* (quitar directorio) o *unlink_file* (desenlazar archivo), dependiendo de la llamada al sistema que se esté manejando. Hablaremos de estas funciones en breve.

La otra llamada al sistema que se maneja en *link.c* es *RENAME* (cambiar de nombre). Los usuarios de UNIX están familiarizados con el comando de shell *mv* que en última instancia utiliza esta llamada; su nombre refleja otro aspecto de la llamada. *RENAME* no sólo puede cambiar el

nombre de un archivo dentro de un directorio, también puede transferir efectivamente un archivo de un directorio a otro, y hacer esto atómicamente, lo que evita ciertas condiciones de competencia. El trabajo corre por cuenta de *do_rename* (línea 25563). Hay muchas condiciones que deben probarse antes de que se pueda llevar a cabo este comando. Entre ellas están:

1. El archivo original debe existir (línea 25578).
2. El nombre de ruta antiguo no debe ser un directorio que esté arriba del nuevo nombre de ruta en el árbol de directorios (líneas 25596 a 25613).
3. Ni. ni.. son aceptables como nombres viejo o nuevo (líneas 25618 y 25619).
4. Ambos directorios padre deben estar en el mismo dispositivo (línea 25622).
5. En ambos directorios padre debe poderse escribir y efectuar búsquedas, y deben estar en un dispositivo en el que se pueda escribir (líneas 25625 y 25626).
6. Ni el nombre antiguo ni el nuevo pueden ser de un directorio que tenga un sistema de archivos montado en él.

Hay algunas otras condiciones que deben verificarse si el nombre nuevo ya existe; la más importante es que sea posible eliminar el archivo existente que tiene el mismo nombre.

En el código para *do_rename* hay unos cuantos ejemplos de decisiones de diseño que se tomaron para minimizar la posibilidad de ciertos problemas. El cambio de nombre de un archivo para darle un nombre que ya existe podría fallar en un disco lleno, aunque en última instancia no se utilice espacio adicional. Para que esto suceda tendría que eliminarse el archivo viejo primero, y esto es lo que se hace en las líneas 25660 a 25666. La misma lógica se emplea en la línea 25680, eliminando el nombre de archivo viejo antes de crear un nombre nuevo en el mismo directorio, a fin de evitar la posibilidad de que el directorio necesite adquirir un bloque adicional. Sin embargo, si el archivo nuevo y el viejo van a estar en diferentes directorios, no surge ese problema, y en la línea 25685 se crea un nuevo nombre de archivo (en un directorio distinto) antes de eliminar el viejo, porque desde el punto de vista de la integridad del sistema una caída que dejara dos nombres de archivo apuntando al mismo nodo-i sería mucho menos grave que una caída que dejara un nodo-i al que no apunta ninguna entrada de directorio. La probabilidad de quedarse sin espacio durante una operación de cambio de nombre es baja, y la de una caída del sistema todavía más baja, pero en estos casos nada cuesta estar preparados para lo peor.

Las funciones restantes de *link.c* apoyan las que ya hemos estudiado. Además, la primera de ellas, *truncate* (línea 25717) se invoca desde varios otros puntos del sistema de archivos. Esta función recorre un nodo-i zona por zona, liberando todas las zonas que encuentra, así como todos los bloques de indirección. *Remove_dir* (línea 25777) realiza varias pruebas adicionales para asegurarse de que el directorio se puede eliminar, y luego invoca *unlink_file* (línea 25818). Si no se encuentran errores, la entrada de directorio se borra y la cuenta de enlaces en el nodo-i se reduce en uno.

5.7.6 Otras llamadas al sistema

El último grupo de llamadas al sistema es un surtido de actividades que tienen que ver con estado, directorios, protección, tiempo y otros servicios.

Modificación de directorios y de la situación de archivos

El archivo *stadir.c* contiene el código para cuatro llamadas al sistema: CHDIR, CHROOT, STAT y PSTAT. Al estudiar *last_dir* vimos cómo la primera acción en una búsqueda de ruta es examinar el primer carácter de la ruta, para determinar si es una diagonal o no. Dependiendo del resultado, se establece un apuntador al directorio raíz o al directorio de trabajo.

Cambiar de un directorio de trabajo (o raíz) a otro sólo es cuestión de modificar estos dos apuntadores dentro de la tabla de procesos del invocador. Estos cambios se efectúan con *do_chdir* (línea 25924) y *do_chroot* (línea 25963). Las dos funciones realizan las verificaciones necesarias y luego invocan *chango* (línea 25978) para abrir el nuevo directorio y reemplazar el viejo.

En *do_chdir* el código de las líneas 25935 a 25951 no se ejecuta en las llamadas CHDIR hechas por procesos de usuario; es sólo para las llamadas efectuadas por el administrador de memoria cuando desea cambiar al directorio de un usuario con el fin de manejar llamadas EXEC. Cuando un usuario trata de ejecutar un archivo, digamos *a.out* en su directorio de trabajo, es más fácil para el administrador de memoria cambiar a ese directorio que tratar de averiguar dónde está.

Las otras dos llamadas al sistema que se manejan en este archivo, STAT y FSTAT, son básicamente iguales, excepto por la forma como se especifica el archivo. La primera da un nombre de ruta, en tanto que la segunda proporciona el descriptor de archivo de un archivo abierto. Los dos procedimientos de nivel más alto, *do_stat* (línea 26014) y *do_fstat* (línea 26035), invocan *stat_inode* para que efectúe el trabajo. Antes de llamar a *stat_inode*, *do_stat* abre el archivo para obtener su nodo-i. De esta forma, tanto *do_stat* como *do_fstat* pasan un apuntador a un nodo-i a *stat_inode*.

Todo lo que *stat_inode* (línea 26051) hace es extraer información del nodo-i y copiarla en un buffer. El buffer se debe copiar explícitamente en el espacio de usuario invocando *sys_copy* en la línea 26088 porque es demasiado grande para caber en un mensaje.

Protección

El mecanismo de protección de MINIX emplea los bits *rwX*. Hay tres juegos de bits para cada archivo: para el propietario, para su grupo y para otros. Los bits se establecen con la llamada al sistema CHMOD, que se efectúa con *do_chmod* en el archivo *protec.c* (línea 26124). Después de efectuar una serie de verificaciones de validez, el modo se cambia en la línea 26150.

La llamada al sistema CHOWN es similar a CHMOD en cuanto a que ambas modifican un campo de nodo-i interno en algún archivo. La implementación también es similar, aunque *do_chown* (línea 26163) sólo puede ser utilizada por el superusuario para cambiar el propietario. Los usuarios ordinarios pueden emplear esta llamada para cambiar el grupo de sus propios archivos.

La llamada al sistema UMASK permite al usuario establecer una máscara (almacenada en la tabla de procesos) que excluye ciertos bits en las llamadas al sistema CREAT subsecuentes. La

implementación completa sería una sola instrucción, la línea 26209, si no fuera porque la llamada debe devolver el valor antiguo de la máscara como resultado. Esta carga adicional triplica el número de líneas de código necesarias (líneas 26208 a 26210).

La llamada al sistema ACCESS permite a un proceso averiguar si puede acceder a un archivo de cierta forma (p. ej., para leerlo), y se implementa con *do_access* (línea 26217), que obtiene el nodo-i del archivo e invoca el procedimiento interno *forbidden* (línea 26242) para ver si el acceso está prohibido. *Forbidden* examina el uid y el gid, así como la información del nodo-i. Dependiendo de lo que encuentra, la función escoge uno de los tres grupos *rwx* y verifica si el acceso está permitido o prohibido.

Read_only (línea 26304) es un pequeño procedimiento interno que indica si el sistema de archivos en el que está situado el nodo-i que es su parámetro está montado sólo para lectura o para lectura y escritura. Este procedimiento es necesario para evitar la escritura en sistemas de archivos que se montaron sólo para lectura.

Tiempo

MINIX cuenta con varias llamadas al sistema que tienen que ver con el tiempo: UTIME, TIME, STIME y TIMES. Éstas se resumen en la Fig. 5-43. Aunque la mayor parte de ellas no tienen nada que ver con archivos, tiene sentido incluirlas en el sistema de archivos porque en el nodo-i de un archivo se guarda información de tiempos.

Llamada	Función
UTIME	Fija el tiempo de última modificación de un archivo
TIME	Fija el tiempo real actual en segundos
STIME	Establece el reloj de tiempo real
TIMES	Obtiene los tiempos de contabilidad de los procesos

Figura 5-43. Las cuatro llamadas al sistema relacionadas con el tiempo.

Cada archivo tiene asociados tres números de 32 bits. Dos de éstos registran los tiempos de última modificación y de último acceso de un archivo. El tercero indica cuándo fue modificado por última vez el estado del nodo-i mismo. Este tiempo cambia en casi todos los accesos a un archivo, excepto si es con READ o con EXEC. Estos tiempos se mantienen en el nodo-i. Con la llamada al sistema UTIME, el propietario del archivo o el superusuario pueden establecer los tiempos de acceso y de modificación. El procedimiento *do_utime* (línea 26422) del archivo *time.c* implementa la llamada al sistema obteniendo el nodo-i y almacenando el tiempo en él. En la línea 26450 se restablecen las banderas que indican que se requiere una actualización del tiempo, a fin de que el sistema no efectúe una costosa y redundante llamada a *clock_time*.

El sistema de archivos no mantiene el tiempo real; esto lo hace la tarea del reloj dentro del kernel. Por tanto, la única forma de obtener o establecer el tiempo real es enviar un mensaje a la

tarea del reloj. Esto es, de hecho, lo que *do_time* y *do_stime* hacen. El tiempo real está en segundos a partir del 1.º de enero de 1970.

El kernel mantiene también la información de contabilidad. En cada tic del reloj, el kernel carga un tic a algún proceso. Esta información puede obtenerse enviando un mensaje a la tarea del sistema, y esto es lo que hace *do_tims* (línea 26492). El procedimiento no se llama *do_times* porque la mayor parte de los compiladores de C anteponen un carácter de subrayado a todos los símbolos externos, y la mayor parte de los enlazadores trunca los símbolos a ocho caracteres. Esto haría imposible distinguir entre *do_time* y *do_times*.

Lo que falta

El archivo *misc.c* contiene procedimientos para unas cuantas llamadas al sistema que no embonan en otras áreas. La llamada al sistema DUP duplica un descriptor de archivo; en otras palabras, crea un nuevo descriptor de archivo que apunta al mismo archivo que su argumento. La llamada tiene una variante DUP2. Ambas versiones de la llamada se implementan con *do_dup* (línea 26632). Esta función se incluye en MINIX como apoyo para programas binarios viejos. Ambas llamadas son obsoletas. La versión actual de la biblioteca C de MINIX invoca la llamada FCNTL cuando encuentra cualquiera de ellas en un archivo fuente en C.

Operación	Significado
FDUPFD	Duplica un descriptor de archivo
FGETFD	Obtiene la bandera de cerrar-al-ejecutar
FSETFD	Establece la bandera de cerrar-al-ejecutar
FGETFL	Obtiene banderas de estado de archivo
FSETFL	Establece banderas de estado de archivo
FGETLK	Obtiene estado de candados de un archivo
FSETLK	Pone candado de lectura/escritura a un archivo
FSETLKW	Pone candado de escritura a un archivo

Figura 5-44. Los parámetros de solicitud de POSIX para la llamada al sistema FCNTL.

FCNTL, que se maneja con *do_fcntl* (línea 26670) es la forma preferida de solicitar operaciones con un archivo abierto. Los servicios se solicitan empleando las banderas definidas por POSIX que se describen en la Fig. 5-44. La llamada se invoca con un descriptor de archivo, un código de solicitud y argumentos adicionales según sea necesario para la solicitud específica. Por ejemplo, el equivalente de la antigua llamada

```
dup2(fd, fd2);
sería fcntl(fd, FJ3UPFD, fd2);
```

Varias de estas solicitudes establecen o leen una bandera; el código tiene sólo unas cuantas líneas. Por ejemplo, la solicitud *F_SETFD* enciende un bit que obliga a cerrar un archivo cuando su propietario emite un EXEC. La solicitud *FJGETFD* sirve para determinar si un archivo deberá cerrarse o no cuando se haga una llamada EXEC. Las solicitudes *FJSETFL* y *FJGETFL* permiten izar banderas para indicar que un archivo dado está disponible en modo no bloqueador o

para operaciones de anexión.

Do_fcntl también maneja el empleo de candados de archivos. Una llamada que especifica el comando *F_GETLK*, *F_SETLK* o *F_SETLKW* se traduce a una llamada a *lock_op*, que ya vimos en una sección anterior.

La siguiente llamada al sistema es SYNC, que copia en el disco todos los bloques y nodos-i que han sido modificados desde que se cargaron. La llamada se procesa con *do_sync* (línea 26730), que simplemente examina todas las tablas en busca de entradas sucias. Los nodos-i deben procesarse primero, ya que *rwJinode* deja sus resultados en el caché de bloques. Una vez que se han escrito todos los nodos-i sucios en el caché de bloques, todos los bloques sucios se escriben en el disco.

Las llamadas al sistema FORK, EXEC, EXIT y SET son en realidad llamadas del administrador de memoria, pero sus resultados deben registrarse aquí también. Cuando un proceso bifurca, es imprescindible que el kernel, el administrador de memoria y el sistema de archivos se enteren de ello. Estas "llamadas al sistema" no provienen de procesos de usuario, sino del administrador de memoria. *Do_fork*, *do_exit* y *do_set* registran la información pertinente en la parte de la tabla de procesos que corresponde al sistema de archivos. *Do_exec* busca y cierra (con *do_closé*) todos los archivos que se marcaron para cerrarse al ejecutar.

La última función de este archivo no es en realidad una llamada al sistema, pero se maneja como tal. Se trata de *do_revive* (línea 26921), que se invoca cuando una tarea que antes no había podido completar un trabajo solicitado por el sistema de archivos, como suministrar datos de entrada a un proceso de usuario, ya ha completado dicho trabajo. A continuación el sistema de archivos revive el proceso y le envía el mensaje de respuesta.

5.7.7 La interfaz con dispositivos de E/S

La E/S en MINIX se efectúa enviando mensajes a tareas que están dentro del kernel. La interfaz del sistema de archivos con dichas tareas está contenida en el archivo *device.c*. Cuando se requiere E/S real con un dispositivo, se invoca *dev_io* (línea 27033) desde *read_write* para manejar archivos especiales por caracteres, y desde *rwJblock* en el caso de archivos especiales por bloques. *Dev_io* construye un mensaje estándar (Fig. 3-15) y lo envía a la tarea especificada. Las tareas se invocan con la línea

```
(*dmap[major].dmpa_rw)(task, &dev_mess);
```

(línea 27056). Esto llama las funciones a través de apuntadores contenidos en el arreglo *dmap* que se define en *table.c*. Todas las funciones que se ocupan de esto están aquí en *device.c*. Mientras *dev_io* espera una respuesta de la tarea, el sistema de archivos espera; no tiene multiprogramación interna. Sin embargo, tales esperas suelen ser cortas (p. ej., 50 ms).

Es posible que los archivos especiales requieran un procesamiento especial cuando se abran o cierren. Lo que deba hacerse exactamente dependerá del tipo de dispositivo. La tabla *dmap* también se usa para determinar qué funciones se invocan para abrir y cerrar cada tipo de dispositivo principal. El procedimiento *dev_opcl* (línea 27071) se llama para los dispositivos de disco, sean disquetes, discos duros o dispositivos basados en memoria. La línea

```
mess_ptr->PROC_NR = fp - fproc;
```

(línea 27081) calcula el número de proceso del invocador. El trabajo real se efectúa pasando el número de tarea y un apuntador al mensaje a *call_task*, que veremos en breve. *Dev_opcl* también se usa para cerrar los mismos dispositivos. De hecho, la única diferencia entre las funciones de abrir y cerrar en el nivel de esta función está en lo que sucede después del retomo de *call_task*.

Otras funciones que se invocan a través de la estructura *dmap* incluyen *tty_open* y *tty_close*, que dan servicio a las líneas en serie, y *ctty_open* y *ctty_close*, que dan servicio a la consola. La última de éstas, *ctty_close*, es casi una rutina ficticia; lo único que hace es devolver una situación *OK* incondicionalmente.

La llamada al sistema SETSID requiere cierto trabajo por parte del sistema de archivos, trabajo que lleva a cabo *do_setsid* (línea 27164). Una llamada al sistema, IOCTL, se maneja primordialmente en *device.c*. Esta llamada se puso ahí porque está íntimamente ligada a la interfaz con las tareas. Cuando se efectúa un IOCTL, se invoca *doJioctl* para que construya un mensaje y lo envíe a la tarea correcta.

En los programas escritos de modo que se ajusten al estándar POSIX, se debe invocar una de las funciones declaradas en *include/termios.h* para controlar dispositivos de terminal. La biblioteca de C traducirá tales funciones a llamadas IOCTL. En el caso de dispositivos distintos de las terminales, IOCTL se utiliza para muchas operaciones, que en su mayor parte se estudiaron en el capítulo 3.

La siguiente función es la única función *PRÍVATE* de este archivo. Se trata *defind_dev* (línea 27228), un pequeño procedimiento auxiliar que extrae los números de dispositivos principal y secundario de un número de dispositivo completo.

La lectura y escritura reales en la mayor parte de los dispositivos pasa por *call_task* (línea 27245), que dirige un mensaje a la tarea apropiada en la imagen del kernel mediante una llamada a *sendrec*. El intento puede fracasar si la tarea está tratando de revivir un proceso como respuesta a una solicitud anterior. Con toda seguridad, éste será un proceso distinto de aquel a cuyo nombre se está efectuando la solicitud actual. *Call_task* exhibe un mensaje en la consola si se recibe un mensaje inapropiado. Es de esperar que tales mensajes no aparecerán durante el funcionamiento normal de MINIX, pero podrían presentarse si se intenta desarrollar un nuevo controlador de dispositivo.

El dispositivo */dev/tty* no existe físicamente; es una ficción a la cual cualquier usuario de un sistema multiusuario puede hacer referencia sin tener que determinar cuál de todas las terminales posibles está en uso. Cuando es necesario enviar un mensaje que hace referencia a */dev/tty*, la siguiente función, *call_ctty* (línea 27311), encuentra los dispositivos principal y secundario correctos y los sustituye en el mensaje antes de reenviarlo a través de *callJtask*.

La última función del archivo es *no_dev* (línea 27337), que se invoca desde ranuras en la tabla para las cuales no existe un dispositivo, por ejemplo cuando se hace referencia a un dispositivo

de red en una máquina sin apoyo de red. *No_dev* devuelve una situación de *ENODEV*, y evita caídas cuando se accede a dispositivos inexistentes.

5.7.8 utilerías generales

El sistema de archivos contiene unos cuantos procedimientos de utilería de propósito general que se emplean en varios lugares y que están reunidos en el archivo *utility.c*.

El primer procedimiento es *clock_time* (línea 27428), que envía mensajes a la tarea del reloj para obtener el tiempo real vigente. El siguiente procedimiento, *fetch_name* (línea 27447) es necesario porque muchas llamadas al sistema tienen un nombre de archivo como parámetro. Si el nombre de archivo es corto, se incluye en el mensaje que el usuario envía al sistema de archivos;

si es largo, se coloca en el mensaje un apuntador al nombre, el cual está en el espacio del usuario. *Fetch_name* verifica ambas posibilidades, y en todos los casos obtiene el nombre.

Dos funciones de este archivo manejan clases generales de errores. *No_sys* es el manejador de errores que se invoca cuando el sistema de archivos recibe una llamada al sistema que no es una de las suyas. *Panic* exhibe un mensaje y le dice al kernel que tire la toalla cuando algo catastrófico sucede.

Las últimas dos funciones, *conv2* y *conv4*, existen para ayudar a MINIX a resolver el problema de los órdenes de byte diferentes en los procesadores Intel y Motorola. Estas rutinas se invocan cuando se lee de o se escribe en una estructura de datos de disco, como un nodo-i o un mapa de bits. El orden de byte que usa el sistema que creó el disco está registrado en el superbloque; si es diferente del orden que el procesador local emplea, se intercambiará. El resto del sistema de archivos no necesita tener conocimiento del orden de byte en el disco.

El último archivo es *putk.c*, que contiene dos procedimientos, ambos relacionados con la exhibición de mensajes. No es posible usar los procedimientos de biblioteca estándar, porque envían mensajes al sistema de archivos. Los procedimientos *de putk.c* envían mensajes directamente a la tarea de la terminal. Vimos un par de funciones casi idénticas en la versión de este archivo incluida en el administrador de memoria.

5.8 RESUMEN

Visto desde fuera, un sistema de archivos es una colección de archivos y directorios, junto con operaciones para manejarlos. Podemos leer y escribir archivos, crear y destruir directorios, y pasar archivos de un directorio a otro. La mayor parte de los sistemas de archivos modernos utiliza un sistema de directorios jerárquico en el que los directorios pueden tener subdirectorios *ad infinitum*.

Visto desde el interior, un sistema de archivos tiene un aspecto muy distinto. Los diseñadores de un sistema de archivos deben preocuparse por la forma de asignar almacenamiento, y del mecanismo para saber siempre qué bloque corresponde a cuál archivo. Vimos además que los diferentes sistemas tienen diferentes estructuras de directorio. La confiabilidad y el rendimiento del sistema de archivos también son cuestiones importantes.

La seguridad y la protección son asuntos vitales tanto para los usuarios como para los diseñadores de un sistema de archivos. Comentamos algunas fallas de seguridad en sistemas viejos y los problemas genéricos que muchos sistemas padecen. Además, estudiamos la verificación de autenticidad, con y sin contraseñas, listas de control de acceso y capacidades, así como un modelo de matriz para razonar acerca de la protección.

Por último, estudiamos con detalle el sistema de archivos de MINIX, que es grande pero no complicado. Este sistema acepta solicitudes de trabajo de los procesos de usuario, consulta una tabla de apuntadores a procedimientos, e invoca el procedimiento apropiado para ejecutar la llamada al sistema solicitada. Gracias a su estructura modular y a su posición fuera del kernel, el sistema de archivos se puede sacar de MINIX y utilizarse como servidor de archivos autónomo con sólo efectuar algunas modificaciones menores.

Internamente, el sistema de archivos de MINIX coloca los datos en buffers de un caché de bloques y trata de leer por adelantado cuando obtiene acceso secuencial a un archivo. Si el caché tiene el tamaño suficiente, la mayor parte del texto de programa se encontrará en la memoria durante las operaciones que acceden repetidamente a un conjunto dado de programas, como en una compilación.

PROBLEMAS

1. Dé cinco nombres de ruta distintos para el archivo */etc/passwd*. (Sugerencia: no olvide las entradas de directorio "." y "..".)
2. Los sistemas que manejan archivos secuenciales siempre cuentan con una operación para "rebobinar" archivos. ¿Los sistemas que manejan archivos de acceso aleatorio también la necesitan?
3. Algunos sistemas operativos ofrecen una llamada al sistema RENAME para dar un nuevo nombre a un archivo. ¿Hay alguna diferencia entre el empleo de esta llamada para cambiar el nombre de un archivo y la táctica de copiar el archivo en un nuevo archivo con el nuevo nombre, borrando después del archivo viejo?
4. Considere el árbol de directorios de la Fig. 5-7. Si */usr/jim* es el directorio de trabajo, indique el nombre de ruta absoluto del archivo cuyo nombre de ruta relativo es *../ast/x*.
5. La asignación contigua de archivos da pie a fragmentación del disco, como se mencionó en el texto. ¿Se trata de fragmentación interna o externa? Haga una analogía con algo que se haya estudiado en el capítulo anterior.
6. Cierta sistema operativo sólo permite un directorio, pero éste puede tener un número arbitrario de archivos con nombres arbitrariamente largos. ¿Podría simularse algo parecido a un sistema de archivos jerárquico? ¿Cómo?
7. Se puede seguir la pista al espacio libre en disco empleando una lista libre o un mapa de bits. Las direcciones de disco requieren D bits. En el caso de un disco con B bloques, F de los cuales están libres, exprese la condición en la que la lista libre ocupa menos espacio que el mapa de bits. Si $D = 16$ bits, exprese su respuesta como un porcentaje del espacio en disco que debe estar libre.
8. Se ha sugerido que la primera parte de todo archivo UNIX se almacene en el mismo bloque de disco que su nodo-i. ¿De qué serviría esto?

9. El rendimiento de un sistema de archivos depende de la tasa de aciertos en caché (la fracción de los bloques requeridos que se encuentra en el caché). Si la satisfacción de una solicitud con un bloque que está en el caché tarda 1 ms, en contraste con los 40 ms que tardaría si se requiriera una lectura de disco, deduzca una fórmula para el tiempo medio de satisfacción de una solicitud si la tasa de aciertos es h . Grafique esta función para valores de h desde 0 hasta 1.0.
10. Un disco flexible tiene 40 cilindros. Una búsqueda tarda 6 ms por cada cilindro que el brazo tiene que moverse. Si no se procura colocar los bloques de un archivo cercanos entre sí, dos bloques que lógicamente son consecutivos (es decir, que van seguidos en un archivo) estarán separados en promedio 13 cilindros. Por otro lado, si el sistema operativo intenta agrupar los bloques del mismo archivo, la distancia media entre bloques se puede reducir a dos cilindros (por ejemplo). ¿Qué tiempo tomaría leer un archivo de 100 bloques en ambos casos, si la latencia rotacional es de 100 ms y el tiempo de transferencia es de 25 ms por bloque?
11. ¿Podría tener algún valor compactar periódicamente un disco? Explique.
12. ¿Cómo podría modificarse TENEX para evitar el problema de las contraseñas que se describió en el texto?
13. Después de graduarse, usted solicita el puesto de director de un centro de cómputo universitario grande que acaba de deshacerse de su antiguo sistema operativo y ha cambiado a UNIX. Usted obtiene el puesto. Quince minutos después de entrar en funciones, su asistente irrumpe en su oficina gritando:
"¡Unos estudiantes descubrieron el algoritmo que usamos para cifrar las contraseñas y lo acaban de pegar en un tablero de avisos!" ¿Qué debería hacer usted?
14. El esquema de protección Morris-Thompson con números aleatorios de n bits se diseñó para que un intruso encontrara muy difícil descubrir un gran número de contraseñas cifrando previamente cadenas comunes. ¿Este esquema ofrece también protección contra un usuario estudiante que está tratando de adivinar la contraseña del superusuario en su máquina?
15. Un departamento de ciencias de la computación tiene un gran número de máquinas UNIX en su red local. Los usuarios de cualquier máquina pueden emitir un comando de la forma
`machine4 who`
y hacer que se ejecute en *machine4* sin tener que iniciar una sesión en la máquina remota. Esta capacidad se implementa haciendo que el kernel del usuario envíe el comando y su uid a la máquina remota. ¿Es seguro este esquema si todos los kernels son confiables (p. ej., minicomputadores de tiempo compartido grandes con hardware de protección)? ¿Y si algunas de las máquinas son computadoras personales de estudiantes, sin hardware de protección?
16. Cuando se elimina un archivo, sus bloques generalmente se colocan otra vez en la lista libre, pero no se borran. ¿Cree usted que sería aconsejable hacer que el sistema operativo borre cada bloque antes de liberarlo? Considere factores tanto de seguridad como de rendimiento en su respuesta, y explique el efecto de cada uno.
17. Tres mecanismos de protección que estudiamos son capacidades, listas de control de acceso y los bits *rwx* de UNIX. Para cada uno de los siguientes problemas de protección, indique cuál de estos mecanismos puede usarse.
- (a) Carlos quiere que todo mundo pueda leer sus archivos, excepto su compañero de oficina.
 - (b) Miguel y Sergio quieren compartir algunos archivos secretos.
 - (c) Linda quiere que algunos de sus archivos sean públicos.
- En el caso de UNIX, suponga que los grupos son categorías como profesores, estudiantes, secretarías, etcétera.

18. Considere el siguiente mecanismo de protección. A cada objeto y cada proceso se asigna un número. Un proceso sólo puede acceder a un objeto si éste tiene un número más alto que el proceso. ¿Cuál de los esquemas que describimos en el texto es parecido a éste? ¿En qué aspecto fundamental este mecanismo difiere de los esquemas del texto?
19. ¿El ataque del caballo de Troya puede funcionar en un sistema protegido con capacidades?
20. Dos estudiantes de ciencias de la computación, Carolina y Eugenia, están discutiendo acerca de los nodos-i. Carolina asegura que las memorias han crecido tanto y bajado tanto de precio que, cuando se abre un archivo, resulta más sencillo y rápido traer una copia nueva del nodo-i a la tabla de nodos-i que examinar toda la tabla para ver si ya está ahí. Eugenia no está de acuerdo. ¿Quién tiene la razón?
21. ¿Qué diferencia hay entre un virus y un gusano? ¿Cómo se reproduce cada uno?
22. Los enlaces simbólicos son archivos que apuntan indirectamente a otros archivos o directorios. A diferencia de los enlaces ordinarios como los que actualmente se implementan en MINIX, un enlace simbólico tiene su propio nodo-i, que apunta a un bloque de datos. Este bloque contiene la ruta del archivo al que se está enlazando, y el nodo-i permite que el enlace tenga dueño y permisos diferentes de los del archivo al que se enlaza. Un enlace simbólico y el archivo o directorio al que apunta pueden estar en dispositivos diferentes. Los enlaces simbólicos no forman parte del estándar POSIX de 1990, pero se espera que se agregarán a POSIX en el futuro. Implemento enlaces simbólicos para MINIX.
23. Usted se da cuenta de que el límite de tamaño de 64 MB para los archivos MINIX no es suficiente para sus necesidades. Extienda el sistema de archivos de modo que aproveche el espacio no utilizado de los nodos-i para un bloque de triple indirección.
24. Indique si el establecimiento de *ROBUSThace* al sistema de archivos más o menos robusto en caso de una caída. Todavía no se ha investigado cuál es el caso en la versión actual de MINIX, así que la respuesta podría ser afirmativa o negativa. Estudie bien lo que sucede cuando un bloque modificado se desaloja del caché. Tenga en cuenta que un bloque de datos modificado puede ir acompañado de un nodo-i y mapa de bits modificados.
25. El tamaño de la *tabla filp* se define actualmente como una constante, *NR_FILPS*, en *enfs/const.h*. A fin de dar cabida a más usuarios en un sistema conectado en red, usted desea aumentar *NR_PROCS* en *include/minix/config.h*. ¿Cómo debe definirse *NR_FILPS* en función de *NR_PROCS*?
26. Diseñe un mecanismo que permita apoyar un sistema de archivos "ajeno", de modo que sea posible, por ejemplo, montar un sistema de archivos MS-DOS en un directorio del sistema de archivos MINIX.
27. Suponga que ocurre un importante avance tecnológico que permite contar con RAM no volátil, que conserva su contenido de forma confiable después de una interrupción de la alimentación eléctrica, al mismo precio y con el mismo rendimiento que la RAM convencional. ¿Qué aspectos del diseño de sistemas de archivos resultarían afectados por este avance?

6

LISTA DE LECTURAS Y BIBLIOGRAFÍA

En los cinco capítulos anteriores tratamos diversos temas. El presente capítulo pretende ser una ayuda para los lectores interesados en llevar más lejos su estudio de los sistemas operativos. La sección 6.1 es una lista de lecturas recomendadas. La sección 6.2 es una bibliografía alfabética de todos los libros y artículos citados en este libro.

Además de las referencias que se dan, los *Proceedings of the n-th ACM Symposium on Operating Systems Principles* (ACM), que se celebra cada dos años, y los *Proceedings of the n-th International Conference on Distributed Computing Systems* (IEEE), que se celebra cada año, son buenos lugares para buscar artículos recientes sobre sistemas operativos. Lo mismo puede decirse del *Symposium on Operating Systems Design and Implementation* de USENIX. Además, dos publicaciones que suelen contener artículos sobre el tema son *ACM Transactions on Computer Systems* y *Operating Systems Review*.

6.1 SUGERENCIAS DE LECTURAS ADICIONALES 6.1.1 Introducción y trabajos generales

Brooks, *The Mythical Man-Month: Essays on Software Engineering*

Libro ingenioso, divertido e informativo sobre cómo *no* escribir un sistema operativo, cuyo autor es alguien que aprendió a la fuerza. Atestado de buenos consejos.

Comer, *Operating System Design. The Xinu Approach*

Un libro acerca del sistema operativo Xinu, que se ejecuta en la computadora LSI-11; contiene una exposición detallada del código fuente, incluido un listado completo en C.

Corbató, "On Building Systems That Will Fail"

En su conferencia por el Premio Turing, el padre del tiempo compartido aborda muchas de las mismas cuestiones que Brooks trata en su libro. La conclusión de Corbató es que todos los sistemas complejos tarde o temprano fallan, y que, para tener la mínima posibilidad de éxito es absolutamente indispensable evitar la complejidad y buscar la sencillez y la elegancia en el diseño.

Deitel, *Operating Systems*, 2a. ed.

Libro de texto general sobre sistemas operativos. Además del material estándar, esta obra contiene estudios de caso de UNIX, MS-DOS, MVS, VM, OS/2 y el sistema operativo de Macintosh.

Finkel, *An Operating Systems Vade Mecum*

Otro texto general sobre sistemas operativos; tiene una orientación práctica, está bien escrito y cubre muchos de los temas tratados en este libro, por lo que es un buen sitio para buscar una forma distinta de ver el mismo tema.

IEEE, *Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*

Éste es el estándar. Algunas partes incluso son muy claras, sobre todo el Anexo B, "Rationale and Notes", que con frecuencia ilumina las razones por las que las cosas se hicieron como se hicieron. Una ventaja de referirse al documento norma es que, por definición, no hay errores. Si un error tipográfico en el nombre de una macro pasó por el proceso de edición, ya no es un error, es oficial.

Lampson, "Hints for Computer System Design"

Butler Lampson, uno de los principales diseñadores de sistemas operativos innovadores en el mundo, ha reunido muchos consejos, sugerencias y pautas de sus años de experiencia y los ha incluido en este entretenido e informativo artículo. Al igual que el libro de Brooks, se trata de una lectura obligatoria para todo aspirante a diseñador de sistemas operativos.

Lewine, *POSIX Programmer's Guide*

Este libro describe el estándar POSIX en una forma mucho más comprensible que el documento de la norma misma, e incluye explicaciones sobre cómo convertir programas anteriores a POSIX y cómo crear nuevos programas para el entorno POSIX. Hay numerosos ejemplos de código, incluidos varios programas completos. Se describen todas las funciones de biblioteca y archivos de cabecera requeridos por POSIX.

Silberschatz y Galvin, *Operating System Concepts*, 4a. ed.

Otro texto sobre sistemas operativos que abarca procesos, administración de memoria, archivos y sistemas distribuidos. Se presentan dos estudios de casos: UNIX y Mach. La portada está

llena de dinosaurios. Qué tiene que ver esto con los sistemas operativos de la década de 1990 no queda claro.

Stallings, *Operating Systems*, 2a. ed.

Un texto más sobre sistemas operativos que trata todos los temas usuales y además incluye un poco de material sobre sistemas distribuidos y un apéndice sobre teoría de colas.

Stevens, *Advanced Programming in the UNIX Environment*

Este libro explica cómo escribir programas en C que usan la interfaz de llamadas al sistema de UNIX y la biblioteca estándar de C. Los ejemplos se basan en las versiones System V Release 4 y 4.4BSD de UNIX. Se describe con detalle la relación entre estas implementaciones y POSIX.

Switzer, *Operating Systems, A Practical Approach*

Un enfoque similar al de este texto. Los conceptos teóricos se ilustran con ejemplos en pseudocódigo y una buena parte del código fuente en C de TUNIX, un sistema operativo modelo. A diferencia de MINIX, TUNIX no está pensado para ejecutarse en una máquina real; opera en una máquina virtual. Este sistema no es tan realista como MINIX en su tratamiento de los controladores de dispositivos, pero va más lejos que MINIX en otras direcciones, como en la implementación de la memoria virtual.

6.1.2 Procesos

Andrews y Schneider, "Concepts and Notations for Concurrent Programming"

Un tutorial y reseña de los procesos y la comunicación entre procesos, incluidas espera activa, semáforos, monitores, transferencia de mensajes y otras técnicas. El artículo muestra también cómo se incorporan estos conceptos en diversos lenguajes de programación.

Ben-Ari, *Principles of Concurrent Programming*

Este librito está dedicado en su totalidad a los problemas de la comunicación entre procesos. Hay capítulos sobre exclusión mutua, semáforos, monitores y el problema de la cena de filósofos, entre otros.

Dubois *et al.*, "Synchronization, Coherence and Event Ordering in Multiprocessors"

Un tutorial sobre sincronización en sistemas multiprocesador con memoria compartida. Sin embargo, algunas ideas aplican igualmente a los sistemas monoprocesador y con memoria distribuida.

Silberschatz y Galvin, *Operating System Concepts*, 4a. ed.

Los capítulos 4 a 6 tratan los procesos y la comunicación entre procesos, incluidos planificación, semáforos, monitores y problemas clásicos de comunicación entre procesos.

6.1.3 Entrada/salida

Chen *et al.*, "RAID: High Performance Reliable Secondary Storage"

El empleo de múltiples unidades de disco en paralelo para E/S rápida es una tendencia en los sistemas más grandes. Los autores analizan esta idea y examinan diferentes organizaciones en términos de rendimiento, costo y confiabilidad.

Coffman *et al.*, "System Deadlocks"

Una introducción corta a los bloqueos mutuos, sus causas y las formas de prevenirlos o detectarlos.

Finkel, *An Operating Systems Vade Mecum*, 2a. ed.

El capítulo 5 trata el hardware de E/S y los controladores de dispositivos, sobre todo para terminales y discos.

Geist y Daniel, "A Continuum of Disk Scheduling Algorithms"

Se presenta un algoritmo generalizado de planificación del brazo del disco. Se proporcionan amplios resultados de simulaciones y experimentos.

Holt, "Some Deadlock Properties of Computer Systems"

Una análisis de los bloqueos mutuos. Holt presenta un modelo de grafos dirigidos que puede servir para analizar algunas situaciones de bloqueo mutuo.

IEEE, revista *Computer*, marzo de 1994

Este número de *Computer* contiene ocho artículos sobre E/S avanzada, y cubre simulación, almacenamiento de alto rendimiento, caches, E/S para computadoras paralelas y multimedia.

Isloor y Marsiand, "The Deadlock Problem: An Overview"

Tutorial sobre bloqueos mutuos, con hincapié especial en los sistemas de bases de datos. Se tratan diversos modelos y algoritmos.

Stevens, "Heuristics for Disk Drive Positioning in 4.3BSD"

Un estudio detallado del rendimiento de disco en Berkeley UNIX. Como suele suceder en los sistemas de computadora, la realidad es más complicada de lo que predice la teoría.

Wiikes *et al.*, "The HPAutoRAID Hierarchical Storage System"

RAID (arreglo redundante de discos de bajo costo) es un importante avance en el área de los sistemas de disco de alto rendimiento. En él, un arreglo de discos pequeños colabora para producir un sistema con gran ancho de banda. Los autores describen con cierto detalle el sistema que construyeron en los laboratorios de HP.

6.1.4 Administración de memoria

Denning, "Virtual Memory"

Artículo clásico sobre muchos aspectos de la memoria virtual. Denning fue uno de los pioneros en este campo, e inventó el concepto de conjunto de trabajo.

Denning, "Work Sets Past and Present"

Una buena reseña de numerosos algoritmos de administración de memoria y paginación. Se incluye una bibliografía muy completa.

Knuth, *The Art of Computer Programming*, vol. 1

En este libro se analizan y comparan los algoritmos de administración de memoria de primer ajuste, mejor ajuste y otros.

Silberschatz y Galvin, *Operating System Concepts*, 4a. ed.

Los capítulos 8 y 9 se ocupan de la administración de memoria, incluidos intercambio, paginación y segmentación. Se mencionan diversos algoritmos de paginación.

6.1.5 Sistemas de archivos

Denning, "The United States vs. Craig Neidorf"

Cuando un joven *hacker* descubrió y publicó información sobre el funcionamiento del sistema telefónico, fue acusado de fraude por computadora. Este artículo describe el caso, que implicó muchas cuestiones fundamentales, incluida la libertad de expresión. El artículo va seguido de algunas opiniones en contra y de una refutación por parte de Denning.

Hafner y Markoff, *Cyberpunk*

El reportero de cómputo del New York Times que reveló la noticia del gusano de Internet y su esposa, también periodista, relatan tres historias absorbentes de jóvenes *hackers* que se han introducido en computadoras por todo el mundo.

Harbron, *File Systems*

Un libro sobre diseño, aplicaciones y rendimiento de sistemas de archivos. Se cubren tanto la estructura como los algoritmos.

McKusick *et al.*; "A Fast File System for UNIX"

El sistema de archivos de UNIX se reimplementó por completo para la versión 4.2 BSD. Este artículo describe el diseño del nuevo sistema, haciendo hincapié en su rendimiento.

Silberschatz y Galvin, *Operating System Concepts*, 4a. ed.

Los capítulos 10 y 11 tratan los sistemas de archivos, y cubren operaciones con archivos, métodos de acceso, semántica de consistencia, directorios, protección e implementación, entre otros temas.

Stallings, *Operating Systems*, 2a. ed.

El capítulo 14 contiene una buena cantidad de material acerca del entorno de seguridad, sobre todo en lo que se relaciona con *hackers*, virus y otras amenazas.

6.2 BIBLIOGRAFÍA ALFABETIZADA

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., and LEVY, H.M.: "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism," *ACM Trans. on Computer Systems*, vol. 10, pp. 53-79, Feb. 1992.

ANDREWS, G.R., and SCHNEIDER, F.B.: "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol. 15, pp. 3-43, March 1983.

BACH, M.J.: *The Design of the UNIX Operating System*, Englewood Cliffs, NJ: Prentice Hall, 1987.

BALA, K., KAASHOEK, M.F., WEIHL, W.: "Software Prefetching and Caching for Translation Lookaside Buffers," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 243-254, 1994.

BAYS, C.: "A Comparison of Next-Fit, First-Fit, and Best-Fit," *Commun. of the ACM*, vol. 20, pp. 191-192, March 1977.

BEN-ARI, M.: *Principles of Concurrent Programming*, Englewood Cliffs, NJ: Prentice Hall International, 1982.

BRINCH HANSEN, P.: "The Programming Language Concurrent Pascal," *IEEE Trans. on Software Engineering*, vol. SE-1, pp. 199-207, June 1975.

BROOKS, F. p., JR.: *The Mythical Man-Month: Essays on Software Engineering*, Anniversary edition, Reading, MA: Addison-Wesley, 1996.

CADOW, H.: *OS/360 Job Control Language*, Englewood Cliffs, NJ: Prentice Hall, 1970.

CHEN, P.M., LEE, E.K., GIBSON, G.A., KATZ, R.H., and PATTERSON, D.A.: "RAID: High Performance Reliable Storage," *Computing Surveys*, vol. 26, pp. 145-185, June 1994.

CHERITON, D.R.: "An Experiment Using Registers for Fast Message-Based Interprocess Communication," *Operating Systems Review*, vol. 18, pp. 12-20, Oct. 1984.

COFFMAN, E.G., ELPHICK, M.J., and SHOSHANI, A.: "System Deadlocks," *Computing Surveys*, vol. 3, pp. 67-78, June 1971.

- COMER, D.:** *Operating System Design. The Xinu Approach*, Englewood Cliffs, N.J.: Prentice Hall, 1984.
- CORBATO, F.J.:** "On Building Systems That Will Fail," *Commun. of the ACM*, vol. 34, pp. 72-81, June 1991.
- CORBATO, F.J., MERWIN-DAGGETT, M., and DALEY, R.C.:** "An Experimental Time-Sharing System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 335-344, 1962.
- CORBATO, F.J., SALTZER, J.H., and CLINGEN, C.T.:** "multics—The First Seven Years," *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS, pp. 571-583, 1972.
- CORBATO, F.J., and VYSSOTSKY, V.A.:** "Introduction and Overview of the MULTICS System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 185-196, 1965.
- COURTOIS, P.J., HEYMANS, F., and PARNAS, D.L.:** "Concurrent Control with Readers and Writers," *Commun. of the ACM*, vol. 10, pp. 667-668, Oct. 1971.
- DALEY, R.C., and DENNIS, J.B.:** "Virtual Memory, Process, and Sharing in MULTICS," *Commun. of the ACM*, vol. 11, pp. 306-312, May 1968.
- DEITEL, H.M.:** *Operating Systems*, 2nd Ed., Reading, MA: Addison-Wesley, 1990. **DENNING, D.:** "The United States vs. Craig Neidorf," *Commun. of the ACM*, vol. 34, pp. 22-43, March 1991.
- DENNING, P.J.:** "The Working Set Model for Program Behavior," *Commun. of the ACM*, vol. 11, pp. 323-333, 1968a.
- DENNING, P.J.:** "Thrashing: Its Causes and Prevention," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 915-922, 1968b.
- DENNING, P.J.:** "Virtual Memory," *Computing Surveys*, vol. 2, pp. 153-189, Sept. 1970.
- DENNING, P.J.:** "Working Sets Past and Present," *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 64-84, Jan. 1980.
- DENNIS, J.B., and VAN HORN, E.C.:** "Programming Semantics for Multiprogrammed Computations," *Commun. of the ACM*, vol. 9, pp. 143-155, March 1966.
- DIJKSTRA, E.W.:** "Co-operating Sequential Processes," in *Programming Languages*, Genuys, P. (Ed.), London: Academic Press, 1965.
- DUKSTRA, E.W.:** "The Structure of the Multiprogramming System," *Commun. of the ACM*, vol. 11, pp. 341-346, May 1968.
- DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.:** "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, vol. 21, pp. 9-21, Feb. 1988.
- ENGLER, D.R., KAASHOEK, M.F., and O'TOOLE, J. JR.:** "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. of the Fifteenth Symp. on Operating Systems Principles*. ACM, pp. 251-266, 1995.
- FABRY, R.S.:** "Capability-Based Addressing," *Commun. of the ACM*, vol. 17, pp. 403-412, July 1974.

- FEELEY, MJ., MORCAN, W.E., PIGHIN, F.H., KARLIN, A.R., LEVY, H.M., and THEKKATH, C.A.:** "Implementing Global Memory Management in a Workstation CLuster," *Proc. of the Fifteenth Symp. on Operating Systems Principies*, ACM, pp. 201-212, 1995.
- FINKEL, R.A.:** *An Operating Systems Vade Mecum*, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1988.
- FOTHERINGHAM, J.:** "Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store," *Commun. of the ACM*, vol. 4, pp. 435-436, Oct. 1961.
- GEIST, R., and DANIEL, S.:** "A Continuum of Disk Scheduling Algorithms," *ACM Trans. on Computer Systems*, vol. 5, pp. 77-92, Feb. 1987.
- GOLDEN, D., and PECHURA, M.:** "The Structure of Microcomputer File Systems," *Commun. of the ACM*, vol. 29, pp. 222-230, March 1986.
- GRAHAM, R.:** "Use of High-Level Languages for System Programming," Project MAC Report TM- 13, M.I.T., Sept. 1970.
- HAFNER, K., and MARKOFF, J.:** *Cyberpunk*, New York: Simón and Schuster, 1991. **HARBON, T.R.:** *File Systems*, Englewood Cliffs, NJ: Prentice Hall, 1988.
- HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B., and WEISER, M.:** "Using Threads in Interactive Systems: A Case Study," *Proc. of the Fourteenth Symp. on Operating Systems Principies*, ACM, pp. 94-105, 1993.
- HAVENDER, J.W.:** "Avoiding Deadlock in Multitasking Systems," *IBM Systems Journal*, vol. 7, pp. 74-84, 1968.
- HEBBARD, B. ET AL.:** "A Penetration Analysis of the Michigan Terminal System," *Operating Systems Review*, vol. 14, pp. 7-20, Jan. 1980.
- HOARE, C.A.R.:** "Monitors, An Operating System Structuring Concept," *Commun. of the ACM*, vol. 17, pp. 549-557, Oct. 1974; Erratum in *Commun. of the ACM*, vol. 18, p. 95, Feb. 1975.
- HOLT, R.C.:** "Some Deadlock Properties of Computer Systems," *Computing Surveys*, vol. 4, pp. 179-196, Sept. 1972.
- HOLT, R.C.:** *Concurrent Euclid, The unix System, and funis*, Reading, MA: Addison-Wesley, 1983.
- HUCK, J., and HAYS, J.:** "Architectural Support for Translation Table Management in Large Address Sapee Machines," *Proc. Twentieth Annual Int'l Symp. on Computer Arch.*, ACM, pp. 39-50, 1993.
- IEEE:** *Information technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, New York: Institute of Electrical and Electronics Engineers, Inc., 1990.
- ISLOOR, S.S., and MARSLAND, T.A.:** "The Deadlock Problem: An Overview," *ieee Computer*, vol. 13, pp. 58-78, Sept. 1980.
- KERNIGHAN, B.W., and RITCHIE, D.M.:** *The C Programming Language*, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1988.

- KLEIN, D.V.:** "Foiling the Cracker: A Survey of, and Improvements to, Password Security," *Proc. UNIX Security Workshop II*, USENIX, Summer 1990.
- KLEINROCK, L.:** *Queueing Systems. Vol. 1*, New York: John Wiley, 1975.
- KNUTH, D.E.:** *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 2nd Ed., Reading, MA: Addison-Wesley, 1973.
- LAMPSON, B.W.:** "A Scheduling Philosophy for Multiprogramming Systems," *Commun. of the ACM*, vol. 11, pp. 347-360, May 1968.
- LAMPSON, B.W.:** "A Note on the Confinement Problem," *Commun. of the ACM*, vol. 10, pp. 613-615, Oct. 1973.
- LAMPSON, B.W.:** "Hints for Computer System Design," *IEEE Software*, vol. 1, pp. 11-28, Jan. 1984.
- LEVIN, R., COHÉN, E.S., CORWIN, W.M., POLLACK, F.J., and WULF, W.A.:** "Policy/Mechanism Separation in Hydra," *Proc. of the Fifth Symp. on Operating Systems Principles*, ACM, pp. 132-140, 1975.
- LEWINE, D.:** *POSIX Programmer's Guide*, Sebastopol, CA: O'Reilly & Associates, 1991.
- LI, K., AND HUDAK, P.:** "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems*, vol. 7, pp. 321-359, Nov. 1989.
- LINDE, R.R.:** "Operating System Penetration," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 361-368, 1975.
- LIONS, J.:** *Lions' Commentary on Unix 6th Edition, with Source Code*, San José, CA: Peer-to-Peer Communications, 1996.
- LIU, C.L., and LAYLAND, J.W.:** "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 20, pp. 46-61, Jan. 1973.
- MARSH, B.D., SCOTT, M.L., LEBLANC, T.J., and MARKATOS, E.P.:** "First-Class User-Level Threads," *Proc. of the Thirteenth Symp. on Operating Systems Principles*, ACM, pp. 110-121, 1991.
- MCKUSICK, M.J., JOY, W.N., LEFFLER, S.J., and FABRY, R.S.:** "AFast File System for UNIX," *ACM Trans. on Computer Systems*, vol. 2, pp. 181-197, Aug. 1984.
- MORRIS, R., and THOMPSON, K.:** "Password Security: A Case History," *Commun. of the ACM*, vol. 22, pp. 594-597, Nov. 1979.
- MULLENDER, S.J., and TANENBAUM, A.S.:** "Immediate Files," *Software—Practico and Experience*, vol. 14, pp. 365-368, April 1984.
- ORGANICK, E.L.:** *The Multics System*, Cambridge, MA: M.I.T. Press, 1972.
- PETERSON, G.L.:** "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, vol. 12, pp. 115-116, June 1981.
- ROSENBLUM, M., and OUSTERHOUT, J.K.:** "The Design and Implementation of a Log-Structured File System," *Proc. Thirteenth Symp. on Operating System Principles*, ACM, pp. 1-15, 1991.

- SALTZER, J.H.:** "Protection and Control of Information Sharing in multics," *Commun. of the ACM*, vol. 17, pp. 388-402, July 1974.
- SALTZER, J.H., and SCHROEDER, M.D.:** "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, pp. 1278-1308, Sept. 1975.
- SALUS, P.H.:** "UNIX At 25," *Byte*, vol. 19, pp. 75-82, Oct. 1994. **SANDHU, R.S.:** "Lattice-Based Access Control Models," *Computer*, vol. 26, pp. 9-19, Nov. 1993.
- SEAWRIGHT, L.H., and MACKINNON, R.A.:** "VM/370—A Study of Multiplicity and Usefulness," *IBM Systems Journal*, vol. 18, pp. 4-17, 1979.
- SILBERSCHATZ, A., and GALVIN, P.B.:** *Operating System Concepts*, 4th Ed. Reading, MA: Addison-Wesley, 1994.
- STALLINGS, W.:** *Operating Systems*, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1995. **STEVENS, W.R.:** *Advanced Programming in the UNIX Environment*, Reading, MA: Addison-Wesley, 1992.
- STEVENS, W.R.:** "Heuristics for Disk Drive Partitioning in 4.3BSD," *Computing Systems*, vol. 2, pp. 251-274, Summer 1989.
- STOLL, C.:** *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*, New York: Doubleday, 1989.
- SWITZER, R.W.:** *Operating Systems, A Practical Approach*, London: Prentice Hall Int'l, 1993.
- TAI, K.C., and CARVER, R.H.:** "VP: A New Operation for Semaphores," *Operating Systems Review*, vol. 30, pp. 5-11, July 1996.
- TALLÛRI, M., and HILL, M.D.:** "Surpassing the TLB Performance of Superpages with Less Operating System Support," *Proc. Sixth Int'l Conf. on Architectural Support for Progr. Lang. and Operating Systems*, ACM, pp. 171-182, 1994.
- TALLÛRI, M., HILL, M.D., and KHALIDI, Y.A.:** "A New Page Table for 64-bit Address Spaces," *Proc. of the Fifteenth Symp. on Operating Systems Principles*, ACM, pp. 184-200, 1995.
- TANENBAUM, A.S.:** *Distributed Operating Systems*, Englewood Cliffs, NJ: Prentice Hall, 1995.
- TANENBAUM, A.S., VAN RENESSE, R., STAVEREN, H. VAN, SHARP, G.J., MULLENDER, S.J., JANSEN, J., and ROSSUM, G. VAN:** "Experiences with the Amoeba Distributed Operating System," *Commun. of the ACM*, vol. 33, pp. 46-63, Dec. 1990.
- TEORY, T.J.:** "Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 1-11, 1972.
- THOMPSON, K.:** "Unix Implementation," *Bell System Technical Journal*, vol. 57, pp. 1931-1946, July-Aug. 1978.
- UHLIG, R., NAGLE, D., STANLEY, T., MUDGE, T., SECREST, S., and BROWN, R.:** "Design Tradeoffs for Software-Managed TLBs," *ACM Trans. on Computer Systems*, vol. 12, pp. 175-205, Aug. 1994.
- VAHALIA, U.:** *UNIX Internáis—The New Frontiers*, Upper Saddle River, NJ: Prentice Hall, 1996.

- WALDSPURGER, C.A., and WEIHL, W.E.:** "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 1-12, 1994.
- WILKES, J., GOLDING, R., STAELIN, C., and SULLIVAN, T.:** "The HPAutoRAID Hierarchical Storage System," *ACMTrans. on Computer Systems*, vol. 14, pp. 108-136, Feb. 1996.
- WULF, W.A., COHÉN, E.S., CORWEM, W.M., JONES, A.K., LEVIN, R., PIERSON, C., and POLLACK, F.J.:**
"HYDRA:
The Kernel of a Multiprocessor Operating System," *Commun. of the ACM*, vol. 17, pp. 337-345, June 1974.
- ZEKAUSKAS, M.J., SAWDON, W.A., and BERSHAD, B.N.:** "Software Write Detection for a Distributed Shared Memory," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 87-100, 1994.

APÉNDICES

A

EL CÓDIGO FUENTE DE MINIX

```
++++++4include/ansi.h++++++
+++++
```

```
00000 /* La cabecera <ansi.h> intenta decidir si el compilador se ajusta lo suficiente a
00001 * Standard C para que Minix lo aproveche. Si es así, el símbolo _ANSI se define
00002 * (como 31415). Si no, _ANSI no se define aquí, pero podrían definirlo aplicaciones
00003 * que no deseen respetar estrictamente las reglas. El número mágico de la
00004 * definición es para inhibir esto si no es muy necesario. (Por consistencia
00005 * con las nuevas pruebas "#ifdef _ANSI" en las cabeceras, _ANSI realmente
00006 * debía definirse como nada, pero eso afectaría muchas rutinas de biblioteca
00007 * que usan "#if _ANSI".)
00008
00009 * Si se define _ANSI, se define la macro
00010
00011 * _PROTOTYPE(function, params)
00012 *
00013 * es definido. Esta macro se expande de diferentes formas, generando prototipos
00014 * ANSI Standard C o bien prototipos K&R (Kernighan & Ritchie) a la antigua,
00015 * según sea necesario. Algunos programas usan _CONST, _VOIDSTAR, etc., de tal
00016 * manera que son portátiles a compiladores tanto ANSI como K&R. Aquí se definen
00017 * las macros apropiadas.
00018 */
00019
00020 #ifndef _ANSI_H
00021 #define _ANSI_H
00022
00023 #if _STDC_ == 1
00024 #define _ANSI 31459 /* compilador dice ajustarse plenamente a ANSI */
00025 #endif
00026
00027 #ifndef _GNUC_
00028 #define _ANSI 31459 /* gcc se ajusta suficiente, aun en modo no ANSI */
00029 #endif
00030
00031 #ifdef _ANSI
00032
00033 /* Guardar todo para prototipos ANSI. */
00034 #define _PROTOTYPE(function, params) function params
00035 #define _ARGS(params) params
00036
00037 #define _VOIDSTAR void *
00038 #define _VOID void
00039 #define _CONST const
00040 #define _VOLATILE volatile
00041 #define _SIZET size_t
00042
00043 #else
00044
00045 /* Desechar los parámetros para prototipos K&R. */
00046 #define _PROTOTYPE(function, params) function()
00047 #define _ARGS(params) ()
00048
00049 #define _VOIDSTAR void *
00050 #define _VOID void
00051 #define _CONST
00052 #define _VOLATILE
00053 #define _SIZET int
00054
```

524

Archivo: include/ansi.h

EL CÓDIGO FUENTE DE MINIX

```
00055 #endif /* _ANSI */
00056
00057 #endif /* ANSI H */
```

+++++

include/limits.h

00100	/* La cabecera <limits.h> define algunos tamaños básicos, tanto de los
00101	* lenguaje (p. ej., cuántos bits en un entero) como del sistema
00102	* cuántos caracteres en un nombre de archivo.
00103	*/
00104	
00105	ffifnndef LIMITS H
00106	#define LIMITSH
00107	
00108	/* Definiciones de char (8 bits en MINIX, y con signo). */
00109	#define CHARBIT 8 /* núm. bits en un char */
00110	#define CHARMIN -128 /* valor mínimo de un char */
00111	ffdefine CHARMAX 127 /* valor máximo de un char */
00112	ffdefine SCHARMIN -128 /* valor mínimo de un char con signo */
00113	#define SCHARMAX 127 /* valor máximo de un char con signo */
00114	ffdefine UCHARMAX 255 /* valor máximo de un char sin signo */
00115	#define MBLLENMAX 1 /* extensión máxima de un char multibyte */
00116	
00117	/* Definiciones de short (16 bits en MINIX). */
00118	#define SHRTMIN (-32767-1) /* valor mínimo de un short */
00119	ffdefine SHRTMAX 32767 /* valor máximo de un short */
00120	#define USHRTMAX 0xFFFF /* valor máximo de un short sin signo */
00121	
00122	/* EM WSIZE es un símbolo generado por compilador que da tamaño de
00123	#if EMWSIZE " 2
00124	ffdefine INTMIN (-32767-1) /* valor mínimo de un int de 16 bits */
00125	ffdefine INTMAX 32767 /* valor máximo de un int de 16 bits */
00126	ffdefine UINT MAX 0xFFFF /* valor máximo de un int de 16 bits sin
00127	#endif
00128	
00129	#if EMWSIZE == 4
00130	ffdefine INTMIN (-2147483647-1) /* valor mínimo de un int de 32 bits
00131	ffdefine INTMAX 2147483647 /* valor máximo de un int de 32 bits */
00132	#define UINTMAX 0xFFFFFFFF /* valor máximo de un int de 32 bits sin
00133	#endif
00134	
00135	/* Definiciones de long (32 bits en MINIX). */
00136	#define LONGMIN (-2147483647L-1)/* valor minimo de un long */
00137	ffdefine LONGMAX 2147483647L /* valor máximo de un long */
00138	#define ULONGMAX 0xFFFFFFFFL /* valor máximo de un long sin signo */
00139	
00140	/* Tamaños mínimos requeridos por la norma POSIX P1003.1 (Tabla 2-3).
00141	#ifndef POSIXSOURCE /* sólo visibles para POSIX */
00142	#define POSIXARGMAX 4096 /* exec() puede tener 4K de args */
00143	#define POSIXCHILDMAX 6 /* un proceso puede tener 6 hijos */
00144	#define POSIXLINKMAX 8 /* un archivo puede tener 8 vínculos */
00145	^define POSIX MAX CANON 255 /* tamaño de cola de entrada canónica */
00146	ffdefine POSIXMAXINPUT 255 /* se puede teclear 255 car. adelantados */
00147	ffdefine POSIXNAMEMAX 14 /* nombres de archivo de hasta 14 car. */
00148	#define POSIXNGROUPSMAX 0 /* IDs de grupos suplementarios opcionales
00149	#define POSIXOPENMAX 16 /* un proceso puede tener 16 archivos abiertos

00150	ffdefine	POSIX PATH MAX 255	/* una ruta puede contener 255 car. */
00151	ffdefine	POSIX PIPE BUF 512	/* escrituras de 512 bytes en conducto atómicas */
00152	#define	POSIX STREAM MAX 8	/* al menos 8 FILE pueden estar abiertos a la vez */

00153	ffdefine	POSIXTZNAMEMAX 3	/* zona de tiempo puede tener al menos 3 car. */
00154	ffdefine	POSIXSSIZEMAX 32767	/* read() debe apoyar lecturas de 32767 bytes */
00155			
00156			/* Valores implementados por MINIX (Tablas 2-4, 2-5, 2-6 y 2-7). */
00157			/* Algunos nombres viejos deben definirse si no es POSIX. */
00158	ffdefine	NOLIMIT 100	/* núm. arbitrario; limite no obligatorio */
00159			
00160	ffdefine	NGROUPSMAX 0	/* ID de grupo suplem. no disponibles */
00161	#if EM	WSIZE > 2	
00162	#define	ARGMAX 16384	/* # de bytes de args + entorno para exec() */
00163	#else		
00164	ffdefine	ARGMAX 4096	/* args + entorno en máquinas chicas */
00165	#endif		
00166	ffdefine	CHILD MAX NO LIMIT	/* MINIX no limita número hijos */
00167	#define	OPEN MAX 20	/* # aren. abiertos que puede tener un proceso */
00168	ffdefine	LINK MAX 127	/* # vínculos que puede tener un archivo */
00169	#define	MAXCANON 255	/* tamaño de cola de entrada canónica */
00170	ffdefine	MAXINPUT 255	/* tamaño de buffer de tecleo adel. */
00171	ffdefine	ÑAME MAX 14	/* # caracteres en el nombre de un archivo */
00172	#define	PATH MAX 255	/* # caracteres en el nombre de una ruta */
00173	#define	PIPE BUF 7168	/* # bytes en escritura atómica a conducto */
00174	#define	STREAM MAX 20	/* debe ser igual a FOPENMAX en stdio.h */
00175	#define	TZNAMEMAX 3	/* máx. bytes en zona de tiempo es 3 */
00176	ffdefine	SSIZEMAX 32767	/* máx. cuenta de bytes def. para read() */
00177			
00178	#endif	/* POSIXSOURCE */	
00179			
00180	#endif	/* LIMITSH */	

+++++-----Hinclude/errno.h+++++-----1-+++++

```

00200
00201     /*La cabecera <errno.h> define los números de los diversos errores que pueden
00202     * ocurrir al ejecutarse un programa; son visibles para programas de usuario y
00203     * deben ser enteros positivos pequeños. También se usan dentro de MINIX, donde
00204     * deben ser negativos. P. ej., la llamada READ se ejecuta internamente invocando
00205     * do_read(). Ésta devuelve un núm. de error (negativo) o bien un número
00206     * (positivo) de bytes leídos realmente.
00207     *
00208     * Para resolver el problema de tener números de error negativos dentro del
00209     * sistema y positivos fuera de él, se usa el siguiente mecanismo. Todas las
00210     * definiciones son de la forma:
00211
00212         #define EPERM             (_SIGN 1)
00213
00214     * Si se define la macro _SYSTEM, _SIGN se hace "-"; si no, se hace "". Así, al
00215     * compilar el sistema operativo, se definirá la macro _SYSTEM, haciendo EPERM =
00216     * (-1), pero cuando este archivo se incluye en un programa de usuario ordinario
00217     * EPERM vale ( 1).
00218     */
00219     #ifndef ERRNO_H
00220         #define ERRNO_H
00221         /* comprobar si ya se incluyó <errno.h>

```

00220	#define ERROH		/*	está incluido; tomar nota */
00221				
00222	/* Now define SIGN as	" " Q^ "		depending on SYSTEM. */
00223	ffifdef SYSTEM			
00224	# define SIGN	-		
00225	# define OK	0		
00226	ffelse			
00227	ff define SIGN			
00228	ffendif			
00229				
00230	extern int errno;		/*	destino para los números de error */
00231				
00232	/* Éstos son los valores numéricos de			los números de error. */
00233	ffdefine NERROR	70	/*	número de errores */
00234				
00235	#define EGENERIC	(SIGN 99)	/*	error genérico */
00236	#define EPERM	(SIGN 1)	/*	operación no permitida */
00237	#define ENOENT	(SIGN 2)	/*	no existe archivo o directorio */
00238	#define ESRCH	(SIGN 3)	/*	no existe proceso */
00239	#define EINTR	(SIGN 4)	/*	llamada a función interrumpida */
00240	ffdefine EIO	(SIGN 5)	/*	error de entrada/salida */
00241	#define ENXIO	(SIGN 6)	/*	no existe dispositivo o dirección */
00242	^define E2BIG	(SIGN 7)	/*	lista de arg demasiado larga */
00243	#define ENOEXEC	(SIGN 8)	/*	error de formato de exec */
00244	#define EBADF	(SIGN 9)	/*	descriptor de archivo no válido */
00245	#define ECHILD	(SIGN 10)	/*	no hay proceso hijo */
00246	#define EAGAIN	(SIGN 11)	/*	recurso temporalmente no disponible */
00247	ffdefine ENOMEM	(SIGN 12)	/*	no hay suficiente espacio */
00248	ffdefine EACCES	(SIGN 13)	/*	permiso negado */
00249	ffdefine EFAULT	(SIGN 14)	/*	dirección no válida */
00250	#define ENOTBLK	(SIGN 15)	/*	Extensión: no es arch. especial p/bloques
00251	#define EBUSY	(SIGN 16)	/*	recurso ocupado */
00252	#define EEXIST	(SIGN 17)	/*	existe el archivo */
00253	ffdefine EXDEV	(SIGN 18)	/*	vínculo incorrecto */
00254	#define ENODEV	(SIGN 19)	/*	no existe dispositivo */
00255	ffdefine ENOTDIR	(SIGN 20)	/*	no es un directorio */
00256	ffdefine EISDIR	(SIGN 21)	/*	es un directorio */
00257	#define EINVAL	(SIGN 22)	/*	argumento no válido */
00258	#define ENFILE	(SIGN 23)	/*	demasiados archivos abiertos en sistema
00259	ffdefine EMFILE	(SIGN 24)	/*	demasiados archivos abiertos */
00260	#define ENOTTY	(SIGN 25)	/*	operación de control de E/S inapropiada
00261	ffdefine ETXTBSY	(SIGN 26)	/*	ya no se usa */
00262	ffdefine EFBIG	(SIGN 27)	/*	archivo demasiado grande */
00263	#define ENOSPC	(SIGN 28)	/*	no hay espacio en dispositivo */
00264	ffdefine ESPIPE	(SIGN 29)	/*	búsqueda no válida */
00265	#define EROFS	(SIGN 30)	/*	sistema de archivos sólo de lectura */
00266	ffdefine EMLINK	(SIGN 31)	/*	demasiados vínculos */
00267	ffdefine EPIPE	(SIGN 32)	/*	conducto roto */
00268	ffdefine EDOM	(SIGN 33)	/*	error de dominio (de ANSI C std) */
00269	#define ERANGE	(SIGN 34)	/*	resultado demasiado grande (de ANSI C
00270	ffdefine EDEADLK	(SIGN 35)	/*	bloqueo evitado */
00271	#define ENAMETOOLONG	(SIGN 36)	/*	nombre de archivo demasiado grande */
00272	ffdefine ENOLCK	(SIGN 37)	/*	no hay candados disponibles */
00273	ffdefine ENOSYS	(SIGN 38)	/*	función no implementada */
00274	ffdefine ENOTEMPTY	(SIGN 39)	/*	directorío no vacío */
00275				
00276	/* Los errores siguien	tes tienen	que	ver con trabajo en red. */
00277	#define EPACKSIZE	(SIGN 50)	/*	tamaño de paquete inválido para protocolo
00278	#define EOUTOFBUFS	(SIGN 51)	/*	no hay suficientes buffers */
00279	ffdefine EBADIOCTL	(SIGN 52)	/*	ioctl no permitido para dispositivo */

00280	#define EBADMODE	(SIGN 53)	/* modo erróneo en ioctl */
00281	ffdefine EWOULDBLOCK	(SIGN 54)	
00282	ffdefine EBADDEST	(SIGN 55)	/* dirección de destino no válida */
00283	#define EDSTNOTRCH	(SIGN 56)	/* destino no alcanzable */
00284	#define EISCONN	(SIGN 57)	/* todos listos conectados */
00285	#define EADDRINUSE	(SIGN 58)	/* dirección en uso */
00286	ffdefine	(SIGN 59)	/* conexión rechazada */
00287	ffdefine ECONNRESET	(SIGN 60)	/* conexión restablecida */
00288	ffdefine ETIMEDOUT	(SIGN 61)	/* conexión vencida */
00289	#define EURG	(SIGN 62)	/* datos urgentes presentes */
00290	ffdefine ENOURG	(SIGN 63)	/* no hay datos urgentes presentes */
00291	ffdefine ENOTCONN	(SIGN 64)	/* no hay conexión (aún o ya) */
00292	ffdefine ESHUTDOWN	(SIGN 65)	/* escritura a conexión de apagado */
00293	ffdefine ENOCONN	(SIGN 66)	/* no hay tal conexión */
00294			
00295	/* Los siguientes no	son errores	POSIX, pero pueden ocurrir. */
00296	ffdefine ELOCKED	(SIGN 101)	/* imposible enviar mensaje */
00297	ffdefine EBADCALL	(SIGN 102)	/* error al enviar/recibir */
00298			
00299	/* Los siguientes códigos de error	los genera el kernel mismo. */	
00300	ffifdef SYSTEM		
00301	#define E BAD DEST	-1001	/* dirección de destino ilegal */
00302	ffdefine E BAO SRC	-1002	/* dirección de origen ilegal */
00303	#define E TRY AGAIN	-1003	/* imposible enviar; tablas llenas */
00304	ffdefine EOVERRUN	-1004	/* interrupción para tarea que no espera */
00305	ffdefine EBADBUF	-1005	/* buf. mensaje fuera de esp. dir. del emisor */
00306	#define E TASK	-1006	/* imposible enviar a la tarea */
00307	#define E NOMESSAGE	-1007	/* RECEIVE falló: no hay mensaje */
00308	ffdefine E NO PERM	-1008	/* usuarios ordin. no pueden enviar a tareas */
00309	#define E BAD FCN	-1009	/* únicos fcn válidos son SEND, RECEIVE, BOTH */
00310	#define E BAD ADDR	-1010	/* dirección mala dada a rutina de utilería */
00311	#define E BAD PROC	-1011	/* núm. proc. erróneo dado a utilería */
00312	ffendif /* SYSTEM */		
00313			
00314	#endif /* ERRNOH */		

4-+-----include/unistd.h-----+-----H-----+-----

```

/* La cabecera <unistd.> contiene constantes diversas manifiestas. */
00400
00401 ffifndef _UNISTD_H
00402 #define _UNISTD_H
00403
00404 /* POSIX requiere size_t y ssize_t en <unistd.h> y otros lugares. */
00405 #ifndef _SIZE_T
00406 ffdefine _SIZE_T
00407 typedef unsigned int size_t;
00408 ffendif
00409 ffifndef _SSIZE_T #define
00410 _SSIZE_T
00411 typedef int ssize_t;
00412 #endif

```

00415		
00416	/* Valores usados por access().	Tabla POSIX 2-8. */
00417	ffdefine FOK 0	/* probar si existe archivo */
00418	ffdefine XOK 1	/* probar si archivo es ejecutable */
00419	ffdefine W OK 2	/* probar si archivo puede escribirse */
00420	#define ROK 4	/* probar si archivo puede leerse */
00421		
00422	/* Valores usados para whence en	lseek(fd, offset, whence). Tabla POSIX 2-9. */
00423	#define SEEK SET 0	/* offset es absoluto */
00424	#define SEEKCUR 1	/* offset es relativo a posición actual */
00425	#define SEEKEND 2	/* offset es relativo al fin del archivo */
00426		
00427	/* Este valor es requerido por	Tabla POSIX 2-10. */
00428	ffdefine POSIXVERSION 199009L	/* a cuál norma se está ajustando */
00429		
00430	/* Estas tres definiciones son	requeridas por POSIX Sec. 8.2.1.2. */
00431	ffdefine STDIN FILENO 0	/* descriptor de archivo para stdin */
00432	#define STDOUT FILENO 1	/* descriptor de archivo para stdout */
00433	#define STDERRFILENO 2	/* descriptor de archivo para stderr */
00434		
00435	#ifndef MINIX	
00436	/* Cómo salir del sistema. */	
00437	#define RBT HALT 0	
00438	ffdefine RBT REBOOT 1	
00439	ffdefine RBT PANIC 2	/* para servidores */
00440	ffdefine RBT MONITOR 3	/* dejar que el monitor haga esto */
00441	#define RBTRESET 4	/* restablecimiento duro del sistema */
00442	#endif	
00443		
00444	/* NULL debe definirse en <unis	td.h> según POSIX Sec. 2.7.1. */
00445	ffdefine NULL ((void *)0)	
00446		
00447	/* Las siguientes son variables	de sistema configurables. Tabla POSIX 4-2. */
00448	ffdefine SCARGMAX	1
00449	#define SC CHILD MAX	2
00450	ffdefine SC CLOCKS PER SEC	3
00451	#define SC CLK TCK	3
00452	ffdefine SC NGROUPS MAX	4
00453	ffdefine SC OPEN MAX	5
00454	#define SC JOB CONTROL	6
00455	ffdefine SCSAVEDIDS	7
00456	#define SC VERSIÓN	8
00457	ffdefine SC STREAMMAX	9
00458	ffdefine SCTZNAMEMAX	10
00459		
00460	/* Las siguientes son variables	de nombre de ruta configurable. Tabla POSIX 5-2. */
00461	#define PCLINKMAX	1 /* cuenta de vínculos */
00462	ffdefine PCMAXCANON	2 /* tamaño cola de entrada canónica */
00463	ffdefine PC MAX INPUT	3 /* tamaño buf. tecleo adelantado */
00464	ffdefine PC NAME MAX	4 /* tamaño de nombre de archivo */
00465	#define PC PATH MAX	5 /* tamaño de nombre de ruta */
00466	#define PC PIPE BUF	6 /* tamaño de conducto */
00467	ffdefine PCNOTRUNC	7 /* trato de componentes de nombre largo */
00468	#define PC VDISABLE	8 /* inhabilitar tty */
00469	ffdefine PCCHOWNRESTRICTED	9 /* chown restringido o no */
00470		
00471	/* POSIX define varias opciones	que pueden implementarse o no, según se desee. Este
00472	* implementador tomó las siguientes decisiones:	
00473	*	
00474	* POSIXJOBCONTROL no	definido: no hay control de trabajos

00486	ffendif		
00487			
00488	PROTOTYPE(void exit, (int status))
00489	PROTOTYPE(int access, (const char *path, int amode))
00490	PROTOTYPE(unsigned int alarm, (unsigned int seconds))
00491	PROTOTYPE(int chdir, (const char *path))
00492	PROTOTYPE(int chown, (const char * path, Uidt owner, Gidt group))
00493	PROTOTYPE(int cióse, (int fd))
00494	PROTOTYPE(char *ctermid, (char *s))
00495	PROTOTYPE(char *cuserid, (char *s))
00496	PROTOTYPE(int dup, (int fd))
00497	PROTOTYPE(int dup2, (int fd, int fd2))
00498	PROTOTYPE(int execi, (const char * path, const char *arg, ...))
00499	PROTOTYPE(int execl, (const char * path, const char *arg, ...))
00500	PROTOTYPE(int execlp, (const char *file, const char *arg, ...))
00501	PROTOTYPE(int execv, (const char *path, char *const argv[]))
00502	PROTOTYPE(int execve, (const char *path, char *const argv[],)
00503		char *const envp[]))
00504	PROTOTYPE(int execvp, (const char *file, char *const argv[]))
00505	PROTOTYPE(pid_t fork, (void))
00506	PROTOTYPE(long fpathconf, (int fd, int name))
00507	PROTOTYPE(char *getcwd, (char *buf, sizet size))
00508	PROTOTYPE(gid_t getegid, (void))
00509	PROTOTYPE(uid_t geteuid, (void))
00510	PROTOTYPE(gid_t getgid, (void))
00511	PROTOTYPE(int getgroups, (int gidsetsize, gid_t grouplist[]))
00512	PROTOTYPE(char *getlogin, (void))
00513	PROTOTYPE(pid_t getpgrp, (void))
00514	PROTOTYPE(pid_t getpid, (void))
00515	PROTOTYPE(pid_t getppid, (void))
00516	PROTOTYPE(uid_t getuid, (void))
00517	PROTOTYPE(int isatty, (int fd))
00518	PROTOTYPE(int link, (const char * existing, const char *new))
00519	PROTOTYPE(off_t lseek, (int fd, off_t offset, int whence))
00520	PROTOTYPE(long pathconf, (const char *path, int name))
00521	PROTOTYPE(int pause, (void))
00522	PROTOTYPE(int pipe, (int fildes[2]))
00523	PROTOTYPE(ssize_t read, (int fd, void *buf, sizet n))
00524	PROTOTYPE(int rmdir, (const char *path))
00525	PROTOTYPE(int setgid, (Gidt gid))
00526	PROTOTYPE(int setpgid, (pid_t pid, pid_t pgid))
00527	PROTOTYPE(pid_t setsid, (void))
00528	PROTOTYPE(int setuid, (Uid_t uid))
00529	PROTOTYPE(unsigned int sleep, (unsigned int seconds))
00530	PROTOTYPE(long sysconf, (int name))
00531	PROTOTYPE(pid_t tcgetpgrp, (int fd))
00532	PROTOTYPE(int tcsetpgrp, (int fd, pid_t pgrp))
00533	PROTOTYPE(char *ttyname, (int fd))
00534	PROTOTYPE(int unlink, (const char *path))


```

00535  _PROTOTYPE( ssize_t write, (int _fd, const void *_buf, size_t _n)
00536
00537  #ifndef _MINIX
00538  _PROTOTYPE( int brk, (char *_addr) _PROTOTYPE( int chroot, (const char *_name) _PROTOTYPE( int
00539  mknod, (const char *_name, Mode_t _PROTOTYPE( int mknod4, (const char *_name, Mode_t
00540  long _size)
00541  _PROTOTYPE( char *mktemp, (char *_template) _PROTOTYPE( int mount, (char
00542  *_spec, char *_name, int _flag) _PROTOTYPE( long mode, Dev_t _addr) ptrace, (int _req, pid_t _pid,
00543  long _addr, long _PROTOTYPE( char *sbrk, (int mode, Dev_t _addr, _incr) _PROTOTYPE( int sync,
00544  (void) _PROTOTYPE( int umount, (const char *_name) _PROTOTYPE( int reboot, (int _how, ...)
00545  _data)
00546  _PROTOTYPE( int gethostname, (char *_hostname, size_t _len) _PROTOTYPE( int getdomainname, (char
00547  *_domain, size_t _len) _PROTOTYPE( int ttysiot, (void) _PROTOTYPE( int fttysiot, (int _fd)
00548  _PROTOTYPE( char *crypt, (const char *_key, const char *_salt) #endif
00549
00550  #endif /* UNISTD H */
00551
00552
00553  ++++++include/string.h++++
00554
00555
00556  /* La cabecera <string.h> contiene prototipos para las funciones que manejan
00557  * cadenas.
00558  */
00559
00560  #ifndef STRING_H
00561  #define STRING_H
00562
00563  #define NULL ((void *)0)
00564
00565  #ifndef _SIZE_T
00566  #define _SIZE_T typedef unsigned int size_t;
00567  #endif /* _SIZE_T */
00568
00569  /* Prototipos de funciones. */ #ifndef _ANSI_H #include <ansi.h> #endif
00570
00571  _PROTOTYPE( void *inchr, (const void *_s, int _c, size_t _n) _PROTOTYPE( int memcmp, (const void
00572  *_s1, const void *_s2, size_t _n) _PROTOTYPE( void *memcpy, (void *_s1, const void *_s2, size_t
00573  _n) _PROTOTYPE( void *memmove, (void *_s1, const void *_s2, size_t _n) _PROTOTYPE( void *memset,
00574  (void *_s, int _c, size_t _n) _PROTOTYPE( char *strcat, (char *_s1, const char *_s2) _PROTOTYPE(
00575  char *strchr, (const char *_s, int _c) _PROTOTYPE( int strncmp, (const char *_s1, const char *_s2,
00576  size_t _n) _PROTOTYPE( int strcmp, (const char *_s1, const char *_s2) _PROTOTYPE( int strcoll,
00577  (const char *_s1, const char *_s2) _PROTOTYPE( char *strcpy, (char *_s1, const char *_s2)
00578
00579
00580
00581
00582
00583
00584
00585
00586
00587
00588
00589
00590
00591
00592
00593
00594
00595
00596
00597
00598
00599
00600
00601
00602
00603
00604
00605
00606
00607
00608
00609
00610
00611
00612
00613
00614
00615
00616
00617
00618
00619
00620
00621
00622
00623
00624
00625
00626
00627
00628
00629

```

00630	PROTOTYPE(size_t strcspn, (const cha	r * s1, const char	*s2))
00631	PROTOTYPE(char *strerror, (int errnuin))
00632	PROTOTYPE(size_t strlen, (const char	*s))	
00633	PROTOTYPE(char *strncat, (char *s1,	const char *s2,	size_t n))
00634	PROTOTYPE(char *strncpy, (char *s1,	const char * s2,	size_t n))
00635	PROTOTYPE(char *strpbrk, (const char	*s1, const char	*s2))
00636	PROTOTYPE(char *strrchr, (const char	*s, int c))
00637	PROTOTYPE(size_t strspn, (const char	*s1, const char	* s2))
00638	PROTOTYPE(char *strstr, (const char	*s1, const char *	S2))
00639	PROTOTYPE(char *strtok, (char * s1,	const char * s2))	
00640	PROTOTYPE(size_t strxfrm, (char *s1	i const char *s2,	size_t n))
00641					
00642	#ifdef MINIX				
00643	/* Para compatibilidad hacia atrás. */				
00644	PROTOTYPE(char *index, (const char *	s, int charwanted))
00645	PROTOTYPE(char *rindex, (const char	*s, int charwanted))
00646	PROTOTYPE(void bcopy, (const void *	src, void *dst, s	size_t length))
00647	PROTOTYPE(int bcmp, (const void *s1	i const void *s2,	size_t length))
00648	PROTOTYPE(void bzero, (void *dst, si	ze_t length))
00649	PROTOTYPE(void *memccpy, (char *dst	i const char *src	, int ucharstop	1
00650			size	t size))
00651	/* Funciones BSD */				
00652	PROTOTYPE(int strcasecmp, (const char	*s1, const char	*s2))
00653	#endif				
00654					
00655	#endif /* STRINGH */				

+++++include/signal.h+++++

```

00700 /* La cabecera <signal.h> define todas las señales ANSI y POSIX. MINIX reconoce
00701 * todas las señales requeridas por POSIX; se definen en seguida. También se
00702 * reconocen algunas señales adicionales.
00703 */
00704
00705 #ifndef _SIGNAL_H #define _SIGNAL_H
00706
00707 #ifndef _ANSI_H #include <ansi.h> #endif
00708
00709 /* Éstos son los tipos más vinculados con el manejo de señales. */ typedef int sig_atoiric_t;
00710
00711 #ifndef _POSIX_SOURCE
00712 #ifndef _SIGSET_T
00713 #define _SIGSET_T
00714 typedef unsigned long sigset_t;
00715 #endif
00716 #endif
00717 #endif
00718 16 /* número de señales empleadas */ 1 /* colgar */
00719
00720
00721 #define _NSIG
00722
00723 #define SIGHUP
00724

```

```
00725 #define SIGINT 2 /* interrupción (DEL) */
00726 #define SIGQUIT 3 /* abandonar (ASCII FS) */
00727 #define SIGILL 4 /* instrucción ilegal */
00728 #define SIGTRAP 5 /* trampa rastreo (no restab. si se atrapa) */
00729 #define SIGABRT 6 /* instrucción IOT */
00730 #define SIGIOT 6 /* SIGABRT para quienes hablan PDP-11 */
00731 #define SIGUNUSED 7 /* código "de repuesto" */
00732 #define SIGFPE 8 /* excepción de punto flotante */
00733 #define SIGKILL 9 /* kill (no puede atraparse ni ignorarse) */
00734 #define SIGUSR1 10 /* usuario definió señal # 1 */
00735 #define SIGSEGV 11 /* violación de segmentación */
00736 #define SIGUSR2 12 /* usuario definió señal # 2 */
00737 #define SIGPIPE 13 /* escritura en conducto sin lector */
00738 #define SIGALRM 14 /* reloj de alarma */
00739 #define SIGTERM 15 /* señal de term. de kill en software */
00740
00741 #define SIGEMT 7 /* obsoleto */
00742 #define SIGBUS 10 /* obsoleto */
00743
00744 /* POSIX requiere que se definan las señales siguientes, aunque no
00745 * estén apoyadas. Aquí están las definiciones, pero no están apoyadas.
00746 */
00747 #define SIGCHLD 17 /* proc. hijo terminado o detenido */
00748 #define SIGCONT 18 /* continúe si está detenido */
00749 #define SIGSTOP 19 /* señal de paro */
00750 #define SIGTSTP 20 /* señal de paro interactiva */
00751 #define SIGTTIN 21 /* proceso segundo plano quiere leer */
00752 #define SIGTTOU 22 /* proceso segundo plano quiere escr. */
00753
00754 /* No se permite el tipo sighandler_t si no se define _POSIX_SOURCE. */
00755 #ifndef _POSIX_SOURCE
00756 #define _sighandler_t sighandler_t
00757 #else
00758 typedef void (*_sighandler_t) (int);
00759 #endif
00760
00761 /* Macros empleadas como apuntadores a funciones. */
00762 #define SIG_ERR ((_sighandler_t) -1) /* devolución de error */
00763 #define SIG_DFL ((_sighandler_t) 0) /* manejo de señales por omisión */
00764 #define SIG_IGN ((_sighandler_t) 1) /* ignorar señal */
00765 #define SIG_HOLD ((_sighandler_t) 2) /* bloquear señal */
00766 #define SIG_CATCH ((_sighandler_t) 3) /* atrapar señal */
00767
00768 #ifdef _POSIX_SOURCE
00769 struct sigaction {
00770     _sighandler_t sa_handler; /* SIG_DFL, SIG_IGN o apuntador a fuñe. */
00771     sigset_t sa_mask; /* señales a bloquear durante manejador */
00772     int sa_flags; /* banderas especiales */
00773 };
00774
00775 /* Fields for sa_flags. */
00776 #define SA_ONSTACK 0x0001 /* entregar señal en pila alterna */
00777 #define SA_RESETHAND 0x0002 /* restab. manej. señales si señal atrapada */
00778 #define SA_NODEFER 0x0004 /* no bloquear señal al atraparla */
00779 #define SA_RESTART 0x0008 /* reinicio aut. de llamada al sist. */
00780 #define SA_SIGINFO 0x0010 /* manejo de señales extendido */
00781 #define SA_NOCLDWAIT 0x0020 /* no cree zombis */
00782 #define SA_NOCLDSTOP 0x0040 /* no reciba SIGCHLD cuando hijo pare */
00783
00784 /* POSIX requiere estos valores para usarlos con sigprocmask(2). */
```

```

00785 /* para bloquear señales */
00786 #define SIG_BLOCK 0
00787 #define SIG_UNBLOCK 1 /* para desbloquear señales */
00788 #define SIG_SETMASK 2 /* para establecer máscara de señal */
00789 #define SIG_INQUIRE 4 /* sólo uso interno */
00790 #endif /* POSIX SOURCE */
00791 /* Prototipos de funciones POSIX y ANSÍ. */
00792 _PROTOTYPE( int raise, (int _sig)
00793 _PROTOTYPE( _sighandler_t signal, (int _sig, _sighandler_t _func)
00794
00795 #ifndef _POSIX_SOURCE
00796 _PROTOTYPE( int kill, (pid_t _pid, int _sig)
00797 _PROTOTYPE( int sigaction,
00798 (int _sig, const struct sigaction *_act, struct sigaction *_oact) _PROTOTYPE( int sigaddset,
00799 (sigset_t *_set, int _sig) sigdelset, (sigset_t *_set, int _sig) sigemptyset, (sigset_t *_set)
00800 sigfillset, (sigset_t *_set) int sigismember, (sigset_t *_set, int _sig) int sigpending, (sigset_t
00801 *_set) int sigprocmask,
00802
00803 (int _how, const sigset_t *_set, sigset_t *_oset) int sigsuspend, (const
00804
00805 _PROTOTYPE( int sigset_t *_sigmask)
00806
00807 _PROTOTYPE( int
00808
00809 _PROTOTYPE( int
00810
00811 _PROTOTYPE( int
00812
00813 _PROTOTYPE( int
00814
00815 #endif /* SIGNAL H */
00816
00817 _PROTOTYPE( (in
00818 int
00819
00820 #endif

```

+++++include/fcntl.h <+++++

```

00900 /* La cabecera <fcntl.h> se necesita para las llamadas open() y fcntl(),
00901 * que tienen diversos parámetros y banderas, aquí descritos.
00902 * Los formatos de las llamadas son:
00903
00904 abrir un archivo obtener o fijar atrib. de archivo
00905
00906 open(path, oflag
00907 [,mode]) fcntl(fd, cmd /*
00908 [,arg]) f
00909
00910 #ifndef #define
00911
00912 FCNTL_H
00913 FCNTL_H se usan para cmd en fcntl(). Tabla POSIX 6-1. */
00914 /* Estos valores /* descriptor de archivo repetido */
00915 #define F_DUPFD /* obtener banderas de dese. arch. */
00916 #define F_GETFD /* fijar banderas de dese. arch. */
00917 #define F_SETFD /* obtener banderas de situac. arch. */
00918 #define F_GETFL /* fijar banderas de situac. arch. */
00919 #define F_SETFL /* obtener info. de candado de registro */
00920 #define F_GETLK /* fijar info. de candado de registro */
00921 #define F_SETLK /* fijar info. candado reg.; esperar si bloqueado */
00922 #define F_SETLKW /* Banderas de descriptor de archivo usadas por fcntl(). Tabla POSIX 6-2. */
00923 #define SETLKW /* Banderas de descriptor de archivo usadas por fcntl(). Tabla POSIX 6-2. */
00924 #define FD_CLOEXEC 1 /* cerrar si bandera exec para 3er arg de fcntl */

```

00925	/* Valores L type para	asegurar registros con fcntl(). Tabla POSIX 6-3. */
00926	ffdefine FRDLCK	1 /* candado compartido o de lectura */
00927	ffdefine F WRLCK	2 /* candado exclusivo o de escritura */
00928	ffdefine FUNLCK	3 /* quitar candado */
00929		
00930	/* Valores Ofiag para	open() Tabla POSIX 6-4. */
00931	#define 0 CREAT	00100 /* crear archivo si no existe */
00932	#define 0 EXCL	00200 /* bandera de uso exclusivo */
00933	ffdefine 0 NOCTTY	00400 /* no asignar terminal controladora */
00934	ffdefine 0TRUNC	01000 /* bandera de truncar */
00935		
00936	/* Banderas de situac.	de arch. para open() y fcntl(). Tabla POSIX 6-5. */
00937	#define 0 APPEND	02000 /* fijar modo de anexar */
00938	ffdefine 0NONBLOCK	04000 /* sin retardo */
00939		
00940	/* Modos de acceso a archivos	para open() y fcntl(). Tabla POSIX 6-6. */
00941	<<define 0 RDONLY	0 /* open(nombre, 0 RDONLY) abre sólo p/leer */
00942	ffdefine 0 WRONLY	1 /* open(nombre, 0WRONLY) abre sólo p/escribir */
00943	#define 0RDWR	2 /* open(nombre, 0RDWR) abre p/leer/escribir */
00944		
00945	/* Máscara para modos	de acceso a archivos. Tabla POSIX 6-7. */
00946	ffdefine 0ACCMODE	03 /* máscara para modos de acceso a arch. */
00947		
00948	/* Struct empleada para candados.	Tabla POSIX 6-8. */
00949	struct flock {	
00950	short ltype;	/* tipo: FRDLGK, FJVRLOCK 0 FJJNLCK */
00951	short l whence;	/* bandera para distancia inicial */
00952	off t l start;	/* distancia relativa en bytes */
00953	offt llen;	/* tamaño; si 0, hasta EOF */
00954	pid t l pid;	/* id de proc. del dueño del candado */
00955	};	
00956		
00957		
00958	/* Prototipos de funciones. *	/
00959	#ifndef ANSIH	
00960	#include <ansi.h>	
00961	#endif	
00962		
00963	PROTOTYPEf int creat,	(const char *path, Modet mode);
00964	PROTOTYPE(int fcntl,	(int filedes, int cmd, ...);
00965	PROTOTYPE(int open,	(const char *path, int oflag, ...);
00966		
00967	#endif /* FCNTLH */	

```
+++++include/stdlib.h+++++
```

```
01000 /* La cabecera <stdlib.h> define ciertas macros, tipos y funes, comunes.*/
01001
01002 ffifndef _STDLIB_H ffdefine _STDLIB_H
01003
01004 /* Las macros son NULL, EXIT_FAILURE, EXIT_SUCCESS, RAND_MAX y MB_CUR_MAX.*/ ffdefine NULL ((void
01005 *)0)
01006
01007 /* retorno en error estándar usando exit() */ /* retorno con éxito usando exit() */
01008 #define EXIT_FAILURE
01009 ffdefine EXIT SUCCESS
```

```
01010 32767 /* valor máx. generado por rand() */
01011 #define RAND_MAX 1 /* valor máx. de carácter multibyte en MINIX */
01012 ^define MB_CUR
01013 MAX typedef struct { int quot, rem; } div_t,
01014 typedef struct { long quot, rem; } ldiv_t;
01015 /* Los tipos son size_t, wchar_t, div_t y ldiv_t. */
01016 #ifndef _SIZE_T
01017 #define _SIZE_T
01018 typedef unsigned int size_t; /* tipo devuelto por sizeof */
01019 #endif
01020
01021 #ifndef _WCHAR_T
01022 #define _WCHAR_T typedef char wchar_t;
01023 #endif
01024 /* Prototipos de /* tipo conjunto de caracteres expandido
01025 funciones. */ #ifndef
01026 _ANSI_H
01027 #include <ansi.h>
01028 #endif
01029
01030 _PROTOTYPE( void abort, (void) _PROTOTYPE( int
01031 abs, (int _j)
01032 _PROTOTYPE( int atexit, (void (*_fune)(void)) ) _PROTOTYPE(
01033 double atof, (const char *_nptr) ) _PROTOTYPE( int atoi,
01034 (const char *_nptr) ) _PROTOTYPE( long atol, (const
01035 char *_nptr) ) _PROTOTYPE( void *calloc, (size_t _nmemb,
01036 size_t _size) ) _PROTOTYPE( div_t div, (int _numer, int _denom)
01037 ) _PROTOTYPE( void exit, (int _status) )
01038 _PROTOTYPE( void free, (void *_ptr) ) _PROTOTYPE(
01039 char *getenv, (const char *_name) ) _PROTOTYPE( long labs,
01040 (long _j) ) _PROTOTYPE( ldiv_t ldiv, (long
01041 _numer, long _denom) ) _PROTOTYPE( void *malloc, (size_t _size)
01042 ) _PROTOTYPE( int mblen, (const char *_s, size_t _n) )
01043 _PROTOTYPE( size_t mbstowcs, (wchar_t *_pwcs, const char *_s, size_t _n)) _PROTOTYPE(
01044 int mbtowc, (wchar_t *_pwc, const char *_s, size_t _n) ) _PROTOTYPE( int rand,
01045 (void) ) _PROTOTYPE( void *realloc, (void
01046 *_ptr, size_t _size) ) _PROTOTYPE( void srand, (unsigned int _seed)
01047 ) _PROTOTYPE( double strtod, (const char *_nptr, char **_endptr) )
01048 _PROTOTYPE( long strtol, (const char *_nptr, char **_endptr, int _base) ) _PROTOTYPE(
01049 int system, (const char *_string) ) _PROTOTYPE( size_t
01050 wcstombs, (char *_s, const wchar_t *_pwcs, size_t _n)) _PROTOTYPE( int wctomb, (char
01051 *_s, wchar_t _wchar) ) _PROTOTYPE( void *bsearch, (const void
01052 *_key, const void *_base,
01053 size_t _nmemb, size_t _size,
01054 int (*compar) (const void *, const void *)) ) _PROTOTYPE(
01055 void qsort, (void *_base, size_t _nmem, size_t _size,
01056 int (*compar) (const void *, const void *)) ) _PROTOTYPE(
01057 unsigned long int strtoul,
01058 (const char *_nptr, char **_endptr, int _base) )
01059
01060 #ifndef _MINIX
01061 _PROTOTYPE( int putenv, (const char *_name)
01062 _PROTOTYPE( int getopt, (int _argc, char **_argv, char *_opts));
01063 extern char *optarg;
01064 extern int optind, opterr, optopt;
01065
01066
01067
01068
01069
```

```
01070  ffendif /* _MINIX */
01071
01072  ffendif /* STDLIB H */
```

```
+++++include/termios.h+++++
```

01103	#define TERMIOSH		
01104			
01105	typedef unsigned short tcflag_t	i	
01106	typedef unsigned	cct;	
01107	typedef unsigned int	speed_t;	
01108			
01109	ffdefine NCCS	20	/* tamaño de arreglo ccc, algo de espacio extra
01110			* para extensiones. */
01111			
01112	/* Estructura de control de terminales primaria. Tabla POSIX 7-1. */		
01113	struct termios {		
01114	tcflag_t cflag;		/* modos de entrada */
01115	tcflag_t coflag;		/* modos de salida */
01116	tcflag_t ccflag;		/* modos de control */
01117	tcflag_t clflag;		/* modos locales */
01118	speed_t cispeed;		/* velocidad de entrada */
01119	speed_t cospeed;		/* velocidad de salida */
01120	ce t c ce[NCCS];		/* caracteres de control */
01121	};		
01122			
01123	/* Valores para mapa	de bits	iflag de termios. Tabla POSIX 7-2. */
01124	#define BRKINT	0x0001	/* señalar interrup. al suspender */
01125	ffdefine ICRNL	0x0002	/* convertir CR en NL en entradas */
01126	#define IGNBRK	0x0004	/* ignorar suspensión */
01127	#define IGNCR	0x0008	/* ignorar CR */
01128	ffdefine IGNPAR	0x0010	/* ignorar caracteres c/errores paridad */
01129	ffdefine INLCR	0x0020	/* convertir NL en CR en entradas */
01130	#define INPCK	0x0040	/* habilitar verif. paridad en entradas */
01131	#define ISTRIP	0x0080	/* enmascarar octavo bit */
01132	ffdefine IXOFF	0x0100	/* habilitar control entrada start/stop */
01133	#define IXON	0x0200	/* habilitar control salida start/stop */
01134	#define PARMRK	0x0400	/* marcar errores de paridad en cola entrada */
01135			
01136	/* Valores para mapa	de bits c oflag	de termios. POSIX Sec. 7.1.2.3. */
01137	ffdefine OPOST	0x0001	/* realizar procesamiento de salidas */
01138			
01139	/* Valores para mapa	de bits	cflag de termios. Tabla POSIX 7-3. */
01140	ffdefine CLOCAL	0x0001	/* ignorar líneas de estado de módem */
01141	#define CREAD	0x0002	/* habilitar receptor */
01142	#define CSIZE	0x000C	/* núm. bits por carácter */
01143	#define CS5	0x0000	/* si CSIZE es CS5, caracteres de 5 bits */
01144	#define CS6	0x0004	/* si CSIZE es CS6, caracteres de 6 bits */
01145	#define CS7	0x0008	/* si CSIZE es CS7, caracteres de 7 bits */
01146	#define CS8	0x000C	/* si CSIZE es CS8, caracteres de 8 bits */
01147	ffdefine CSTOPB	0x0010	/* 2 bits de paro si encendida, si no, 1 */
01148	^define HUPCL	0x0020	/* colgar en el último cierre */
01149	#define PARENB	0x0040	/* habilitar paridad en salidas */

01150	#define PARODD	0x0080	/* paridad impar si encendida, si no, par */	*/
01151				
01152	/* Valores para mapa de	bits c	lflag de termios. Tabla POSIX 7-4. */	
01153	ffdefine ECHO	0X0001	/* habilitar eco de carac. de entrada */	
01154	ffdefine ECHOE	0x0002	/* eco de ERASE como retroceso */	
01155	ffdefine ECHOK	0X0004	/* eco de KILL */	
01156	#define ECHONL	0x0008	/* eco de NL */	
01157	ffdefine ICANON	0X0010	/* entrada canónica (erase y kill hábil.)	*/
01158	ffdefine IEXTEN	0x0020	/* habilitar funciones extendidas */	
01159	#define ISIG	0X0040	/* habilitar señales */	
01160	#define NOFLSH	0x0080	/* inhabilitar vaciar en interrup. o salir	*/
01161	#define TOSTOP	0X0100	/* enviar SIGTTOU (ctl. trabajos, no hay)	*/
01162				
01163	/* Índices de arreglo c	ce. Pr	edeterminados entre paréntesis. Tabla POSIX	7-5. */
01164	#define VEOF	0	/* ccc[VEOF] = car. EOF ("D) */	
01165	#define VEOL	1	/* ce c[VEOL] = car. EOL (no def.) */	
01166	#define VERASE	2	/* CC C[VERASE] = car. ERASE ('H) */	
01167	#define VINTR	3	/* ccc[VINTR] = car. INTR (DEL) */	
01168	#define VKILL	4	/* ccc[VKILL] = car. KILL ('U) */	
01169	ffdefine VMIN	5	/* ce c[VMIN] = valor MIN de tempor. */	
01170	ffdefine VQUIT	6	/* ce c[VQUIT] = car. QUIT (~\) */	
01171	#define VTIME	7	/* ce c[VTIME] = valor TIME de tempor. */	
01172	#define VSUSP	8	/* ccc[VSUSP] = SUSP ("Z, se ignora) */	
01173	#define VSTART	9	/* CCC[VSTART] = car. START ("S) */	
01174	#define VSTOP	10	/* ccc[VSTOP] = car STOP ('Q)	
01175				
01176	#define POSIXVDISABLE	(CC	t)0xFF /* No puede generarse este	
01177			* carácter con teclados 'normales)
01178			* pero algunos de idioma específico	
01179			* pueden generar 0xFF.	
01180			* Parece que se usan los 256,	
01181			* así que ce t debe ser un short.	
01182			*/	
01183				
01184	/* Valores de la tasa de bauds	.	Tabla POSIX 7-6. */	
01185	#define B0	0X0000	/* colgar la línea */	
01186	#define B50	0x1000	/* 50 baud */	
01187	ffdefine B75	0x2000	/* 75 baud */	
01188	ffdefine B110	0x3000	/* 110 baud */	
01189	ffdefine B134	0x4000	/* 134.5 baud */	
01190	#define B150	0x5000	/* 150 baud */	
01191	ffdefine B200	0x6000	/* 200 baud */	
01192	#define B300	0x7000	/* 300 baud */	
01193	ffdefine B600	0x8000	/* 600 baud */	
01194	#define B1200	0x9000	/* 1200 baud */	
01195	#define B1800	0XA000	/* 1800 baud */	
01196	#define B2400	0XB000	/* 2400 baud */	
01197	#define B4800	0x0000	/* 4800 baud */	
01198	ffdefine B9600	0XD000	/* 9600 baud */	
01199	#define B19200	0XE000	/* 19200 baud */	
01200	ffdefine B38400	0XF000	/* 38400 baud */	
01201				
01202	/* Acciones opcionales	para tcsetattr(). POSIX Sec. 7.2.1.2. */		
01203	#define TCSANOW	1	/* cambios efectivos de inmediato */	
01204	ffdefine TCSADRAIN	2	/* cambios efectivos al terminar salidas */	
01205	#define TCSAFLUSH	3	/* esperar término de salidas y vaciar entradas */	
01206				
01207	/* Valores de queue selector para tcflush(). POSIX Sec. 7.2.2.2. */			
01208	ffdefine TCIFLUSH	1	/* vaciar datos entrada acumulados */	
01209	ffdefine TCOFLUSH	2	/* vaciar datos salida acumulados */	

01210	#define TCIOFLUSH 3 /* vaciar datos entrada y salida acumul.	*/
01211		
01212	/* Valores de action para tcflow(). POSIX Sec. 7.2.2.2. */	
01213	ffdefine TCOOFF 1 /* suspender salidas */	
01214	#define TCOON 2 /* reiniciar salidas suspendidas */	
01215	ffdefine TCIOFF 3 /* transmitir car. STOP por la línea */	
01216	#define TCION 4 /* transm. car. START por la línea */	
01217		
01218		
01219	/* Prototipos de funciones. */	
01220	ffifndef ANSIH	
01221	ffinclude <ansi.h>	
01222	ffendif	
01223		
01224	PROTOTYPE(int tcsendbreak, (int fildes, int duration))
01225	PROTOTYPE(int tcdrain, (int filedes))
01226	PROTOTYPE(int tcflush, (int filedes, int queue selector))
01227	PROTOTYPE(int tcflow, (int filedes, int action))
01228	PROTOTYPE(speedt cfgetspeed, (const struct termios *termiosp))
01229	PROTOTYPE(speedt cfsetospeed, (const struct termios *termiosp))
01230	PROTOTYPE(int cfsetispeed, (struct termios *termiosp, speedt speed))
01231	PROTOTYPE(int cfsetospeed, (struct termios *termiosp, speedt speed))
01232	PROTOTYPE(int tcgetattr, (int filedes, struct termios * termios p))
01233	PROTOTYPE(int tcsetattr, \	
01234	(int filedes, int optactions, const struct termios *termiosp))	
01235		
01236	ffdefine cfgetspeed(termiosp) ((termiosp)->cispeed)	
01237	#define cfgetspeed(termiosp) ((termiosp)->cospeed)	
01238	ffdefine cfsetispeed(termiosp, speed) ((termiosp)->cispeed = (speed),	0)
01239	ffdefine cfsetospeed(termiosp, speed) ((termiosp)->cospeed = (speed), 0)	
01240		
01241	#ifdef MINIX	
01242	/* Aquí están las extensiones locales al estándar POSIX para Minix.	
01243	* Los programas que se ajusten a Posix no pueden acceder a ellas,	
01244	* así que sólo se definen cuando se compila un programa Minix.	
01245	*/	
01246		
01247	/* Extensiones al mapa de bits cflag de termios. */	
01248	ffdefine IXANY 0x0800 /* cualquier tecla p/continuar salidas */	
01249		
01250	/* Extensiones al mapa de bits c oflag de termios. Sólo están activas si	
01251	* OPOST está habilitado */	
01252	#define ONLCR 0x0002 /* Convertir NL en CR-NL en salidas */	
01253	#define XTABS 0x0004 /* Expandir tabs a espacios */	
01254	#define ONOEOT 0x0008 /* desechar EOT ("D) en salidas */	
01255		
01256	/* Extensiones al mapa de bits cflag de termios. */	
01257	#define LFLUSHO 0x0200 /* Vaciar salidas */	
01258		
01259	/* Extensiones al arreglo c ce */	
01260	ffdefine VREPRINT 11 /* ce C[VREPRINT] ("R) */	
01261	ffdefine VLNEXT 12 /* ccc[VLNEXT] ("V) */	
01262	#define VDISCARD 13 /* ccc[VDISCARD] ("O) */	
01263		
01264	/* Extensiones a las tasas de bauds. */	
01265	#define B57600 0x0100 /* 57600 baud */	
01266	ffdefine BU 5200 0x0200 /* 115200 baud */	
01267		
01268	/* Éstos son los valores predeterminados que usa el kernel y 'stty sane'	*/
01269		

01274	#define TSPPEEDDEF	B9600		
01275				
01276	ffdefine TEOF DEF	'\4' /*	"D" /*	
01277	#define TEOLDEF	POS IX VDISABLE		
01278	ffdefine TERASEDEF	'\10' /*	"H" /*	
01279	ffdefine TINTRJ3EF	'\177' /*	"?" /*	
01280	ffdefine TKILL DEF	'\25' /*	"U" /*	
01281	#define TMIN DEF	1		
01282	ffdefine TQUIT DEF	'\34' /*	"\" /*	
01283	#define TSTART DEF	'\21' /*	"Q" /*	
01284	#define TSTOP DEF	'\23' /*	"S" /*	
01285	#define TSUSPDEF	'\32' /*	"Z" /*	
01286	#define TTIME DEF	0		
01287	#define TREPRINT DEF	'\22' /*	"R" /*	
01288	#define TLNEXT DEF	'\26' /*	"V" /*	
01289	ffdefine TDISCARDDEF	'\17' /*	"0" /*	
01290				
01291	/* Tamaño de ventana. Esta información se guarda en el controlador TTY pero no se usa			
01292	* Se puede usar para	aplicaciones	basadas en pantalla en un entorno de ventanas	
01293	* Se pueden usar los	ioctl TIOCGWINSZ	v TIOCSWINSZ para obtener v fijar esta	
01294	* información.			
01295	*/			
01296				
01297	struct winsize			
01298	{			
01299	unsigned short	wsrow;		/* filas, en caracteres */
01300	unsigned short	wscol;		/* columnas, en caracteres */
01301	unsigned short	wspixel;		/* tamaño horizontal, pixeles */
01302	unsigned short	wsvpixel;		/* tamaño vertical, pixeles */
01303	};			
01304	#endif /* MINIX */			
01305				
01306	ffendif /* TERMIOSH */			

```
+++++include/a.out.h+++++
```

```
01401 /* La cabecera <a.out> describe el formato de los archivos ejecutables.
01402 */
01403
01404 #ifndef _AOUT_H
01405 #define _AOUT_H
01406
01407 struct exec {
01408     unsigned char a_magic[2]; /* cabecera a.out */
01409     unsigned char a_flags; /* número mágico */
01410     unsigned char a_cpu; /* banderas, véase adelante */
01411     unsigned char a_hdrlen; /* id de cpu */
01412     unsigned char a_unused; /* longitud de cabecera */
01413     unsigned short a_version; /* reservado p/uso futuro */
01414     long a_text; /* sello de versión (no se usa) */
01415     long a_data; /* tamaño de segm. texto, bytes */
01416     long a_bss; /* tamaño de segm. sigui, datos, bytes */
01417     /* tamaño de segm. bss, bytes */

```

01415	long aentry; /* punto de entrada */
01416	long atotal; /* total de memoria asignada */
01417	long asym; /* tamaño de tabla de símbolos */
01418	
01419	/* FORMA CORTA TERMINA AQUÍ */
01420	long atrsize; /* tamaño reubicación de texto */
01421	long adrsz; /* tamaño reubicación de datos */
01422	long atbase; /* base reubicación de texto */
01423	long a dbase; /* base reubicación de datos */
01424	};
01425	
01426	ffdefine A MÁGICO (unsigned char) 0x01
01427	ffdefine A MAGIC1 (unsigned char) 0x03
01428	#define BADMAG(X) ((X) .amagic[0] !=AMAGIC0 ; i (X) .amagic[1] != AMAGIC1
01429	
01430	/* Id de CPU de máq. OBJETIVO (orden de bytes codificado en dos bits de orden bajo)
01431	ffdefine A NONE 0x00 /* desconocido */
01432	ffdefine A 18086 0x04 /* intel 18086/8088 */
01433	ffdefine AM68K 0x0B /* motorola m68000 */
01434	ffdefine ANS16K 0x00 /* national semiconductor 16032 */
01435	ffdefine AI80386 0x10 /* intel í80386 */
01436	#define ASPARC 0x17 /* Sun SPARC */
01437	
01438	#define ABLR(cputype) ((cputype&0x01)!=0) /* TRUE si bytes izq. a der. */
01439	ffdefine AWLR(cputype) ((cputype&0x02)!=0) /* TRUE si palabras izq. a der. */
01440	
01441	/* Banderas. */
01442	#define AUZP 0x01 /* página(s) cero sin mapa */
01443	ffdefine APAL 0x02 /* ejecutable alineado por página */
01444	ffdefine ANSYM 0x04 /* tabla de símbolos al estilo nuevo */
01445	#define AEXEC 0x10 /* ejecutable */
01446	^define A8EP 0x20 /* I/D separados */
01447	ffdefine APURE 0x40 /* texto puro */ /* no se usa */
01448	ffdefine ATOVLY 0x80 /* superpos. texto */ /* no se usa */
01449	
01450	/* Distancias de varias cosas. */
01451	ffdefine A MINHDR 32
01452	#define ATEXTPOS(X) ((long)(X).ahdrln)
01453	ffdefine A DATAPOS(X) (A TEXTPOS(X) + (X).a text)
01454	ffdefine A HASRELS(X) ((X).a hdrien > (unsigned char) A MINHDR)
01455	ffdefine A HASEXT(X) ((X).a hdrien > (unsigned char) (A MINHDR + 8))
01456	ffdefine A HASLNS(X) ((X).a hdrien > (unsigned char) (A MINHDR + 16))
01457	ffdefine A HASTOFF(X) ((X).a hdrien > (unsigned char) (A MINHDR + 24))
01458	ffdefine ATRELPOS(X) (ADATAPOS(X) + (X).adata)
01459	ffdefine ADRELPOS(X) (ATRELPOS(X) + (X).atrsz)
01460	#define A SYMPOS(X) (A TRELPOS(X) + (A HASRELS(X) ? \
01461	((X).atrsz + (X).adrsz) : 0)
01462	
01463	struct reloc {
01464	long rvaddr; /* dir. virtual de la referencia */
01465	unsigned short rsymndx; /* núm. seg. interno o núm. simb. externo */
01466	unsigned short r type; /* tipo de reubicación */
01467	};
01468	
01469	/* valores de r type: */
01470	ffdefine R ABBS 0
01471	ffdefine R RELLBYTE 2
01472	#define RPCRBYTE 3
01473	ffdefine R RELWORD 4
01474	ffdefine RPCRWORD 5

01484	#define SBSS	((unsigned	short)-4)
01485			
01486	struct nlist {		/* entrada de tabla de símbolos */
01487	char nname[8]	i	/* nombre de símbolo */
01488	long nvalue;		/* valor */
01489	unsigned char	nsclass;	/* clase de almacenamiento */
01490	unsigned char	nnumaux;	/* núm. entradas aux. (no se usa) */
01491	unsigned snort	nstype;	/* base lenguaje y tipo deriv. (no se usa) */
01492	};		
01493			
01494	/* Bits bajos de	clase de almacenam	iento (sección). */
01495	ffdefine NSECT	07	/* máscara de sección */
01496	#define NUNDF	00	/* no definida */
01497	#define N ABS	01	/* absoluta */
01498	ffdefine N TEXT	02	/* texto */
01499	#define N DATA	03	/* datos */
01500	ffdefine NBSS	04	/* bss */
01501	#define NCOMM	05	/* (común) */
01502			
01503	/* Bits altos de	clase de almacenam	iento. */
01504	#define N CLASS	0370	/* máscara de clase de almac. */
01505	#define C NULL		
01506	ffdefine CEXT	0020	/* símbolo externo */
01507	#define CSTAT	0030	/* estático */
01508			
01509	/* Prototipos de	funciones.	*/
01510	#ifndef ANSIH		
01511	#include <ansi.h>		
01512	#endif		
01513			
01514	PROTOTYPE(int	nlist, (char	*file , struct nlist *nl));
01515			
01516	#endif /* AOUT	H */	

+++++include/sys/types.h+++++

```

01600 /* La cabecera <sys/types.h> contiene definiciones de tipos de datos importantes.
01601 * Se considera buena práctica de programación usarlas en lugar
01602 * del tipo base subyacente. Por convención, todos los nombres
01603 * de tipo terminan con _t.
01604
01605 TYPES_H TYPES H
01606 #ifndef
01607 #   /* _ANSI se usa de alguna manera para determinar si el compilador es o no
01608 #   #define
01609 #   e

```

01610	* un compilador de 16 bits	
01611	*/	
01612	ffifndef ANSI	
01613	#include <ansi.h>	
01614	ffendif	
01615		
01616	/* El tipo sizet contiene los resultados del operador sizeof. A primera vista,	
01617	* parece obvio que debía ser unsigned int, pero no siempre es así. P. ej.,	
01618	* MINIX-ST (68000) tiene apuntadores de 32 bits y enteros de 16 bits. Si se pide	
01619	* el tamaño de un struct o array de 70K, el resultado requiere 17 bits para	
01620	* expresarse, así que size t debe ser long. ssize_t es la versión con signo	
01621	* de size t.	
01622	*/	
01623	#ifndef SIZE T	
01624	ffdefine SIZET	
01625	typedef unsigned int size t;	
01626	#endif	
01627		
01628	#ifndef SSIZE T	
01629	ffdefine SSIZET	
01630	typedef int ssize_t;	
01631	#endif	
01632		
01633	#ifndef TIME T	
01634	ffdefine TIMET	
01635	typedef long timet;	/* tiempo en seg. desde 1 ene 1970 0000 GMT */
01636	ffendif	
01637		
01638	#ifndef CLOCK T	
01639	^define CLOCKT	
01640	typedef long clock_t;	/* unidad para contabil. sistema */
01641	ffendif	
01642		
01643	ffifndef SIGSETT	
01644	#define SIGSETT	
01645	typedef unsigned long sigset t;	
01646	#endif	
01647		
01648	/* Tipos usados en estructuras de datos de disco, nodos-i, etcétera. */	
01649	typedef short dev_t; /* par dispositivo (princ.lsecund.) */	
01650	typedef char gid_t; /* id de grupo */	
01651	typedef unsigned short ino_t; /* número de nodo-i */	
01652	typedef unsigned short mode_t; /* tipo arch. y bits de permiso */	
01653	typedef char nlink_t; /* núm. de vínculos a un archivo */	
01654	typedef unsigned long off_t; /* distancia dentro de un archivo */	
01655	typedef int pid_t; /* id de proceso (debe tener signo) */	
01656	typedef short uid_t; /* id de usuario */	
01657	typedef unsigned long zone_t; /* número de zona */	
01658	typedef unsigned long block_t; /* número de bloque */	
01659	typedef unsigned long bit_t; /* núm. de bit en un mapa de bits */	
01660	typedef unsigned short zone1 t;	/* núm. zona para sistemas de arch. V1 */
01661	typedef unsigned short bitt; /* colección de bits en mapa de bits */	
01662		
01663	typedef unsigned char u8t	/* tipo de 8 bits */
01664	typedef unsigned short u16t	/* tipo de 16 bits */
01665	typedef unsigned long u32t	/* tipo de 32 bits */
01666		
01667	typedef char i8 t	/* tipo de 8 bits con signo */
01668	typedef short i16t	/* tipo de 16 bits con signo */
01669	typedef long i32t	/* tipo de 32 bits con signo */

```
01670
01671 /* Los tipos siguientes son necesarios porque MINIX usa definiciones
01672 * de función al estilo K&R (para máxima transportabilidad). Si un
01673 * short, como dev_t se pasa a una función con definición K&R, el
01674 * compilador lo promueve autom. a int. El prototipo debe contener un
01675 * int como parámetro, no un short, porque una def. de función al
01676 * estilo antiguo espera un int. Por tanto, usar dev_t en un prototipo
01677 * sería incorrecto. Bastaría con usar int en lugar de dev_t en los
01678 * prototipos, pero Dev_t es más claro.
01679 */
01680 typedef int      Dev_t;
01681 typedef int      Gid_t;
01682 typedef int      Nlink_t;
01683 typedef int      Uid_t;
01684 typedef int      U8_t;
01685 typedef unsigned long U32_t;
01686 typedef int      I8_t;
01687 typedef int      H6_t;
01688 typedef long     I32_t;
01689
01690 /* En ANSÍ C no puede escribirse limpiamente la promoción de los tipos sin signo.
01691 * Si sizeof(short) == sizeof(int), no hay promoción, y el tipo se queda sin
01692 * signo. Si el compilador no es ANSÍ, no suele perderse la carencia de signo y
01693 * usualmente no hay prototipos, así que el tipo promovido no importa. El empleo
01694 * de tipos como Ino_t es un intento por usar ints (que no se promueven) al
01695 * tiempo que se proporciona información al lector.
01696 */
01697
01698 #ifndef _ANSI_H
01699 #include <ansi.h>
01700 #endif
01701
01702 #if _EM_WSIZE == 2 ;; !defined(_ANSI)
01703 typedef unsigned int  Ino_t;
01704 typedef unsigned int  Zone1_t;
01705 typedef unsigned int  Bitchunk_t;
01706 typedef unsigned int  Ut6_t;
01707 typedef unsigned int  Mode_t;
01708
01709 #else /* _EM_WSIZE == 4, o _EM_WSIZE no definido, o _ANSI definido */
01710 typedef int           Ino_t;
01711 typedef int           Zone1_t;
01712 typedef int           Bitchunk_t;
01713 typedef int           Ul6_t;
01714 typedef int           Mode_t;
01715
01716 #endif /* _EM_WSIZE == 2, etc */
01717
01718 /* Tipo de manejador de señales, p. ej., SIG_IGN */
01719 #if defined(_ANSI)
01720 typedef void (*sighandler_t) (int);
01721 #else
01722 typedef void (*sighandler_t) ();
01723 #endif
01724
01725 #endif /* _TYPES H */
```

```

+++++include/sys/ioctl.h+++++
01800 /* La cabecera ioctl.h declara operaciones de control de dispositivos. */
01801
01802 #ifndef _IOCTL_H
01803 #define _IOCTL_H
01804
01805 #if _EM_WSIZE >= 4
01806 /* Los iocti tienen el comando codificado en la palabra de orden bajo,
01807 * y el tamaño del parámetro en la de orden alto. Los 3 bits altos de la palabra
01808 * de orden alto codifican la situación in/out/void del parámetro.
01809 */
01810 #define _IOCPARM_MASK
01811 #define _IOC_VOID          0x1FFF          ffdefine
01812 #define _IOCTYPE_MASK     0x20000000
01813 #define _IOC_IN           0XFFFF
01814 #define _IOC_OUT
01815 #define _IOC_INOUT
01816 #define _IO(x,y) (_IOC_IN _IOC_OUT) ffdefine _IOR(x,y,t)
01817 ffdefine _IOW(x,y,t) ((x << 8) ; y ; _IOC_VOID) ((x << 8) ; y ; ((sizeof(t) ffdefine _IORW(x,y,t)
01818 & _IOCPARM_MASK) << 16) ; \
01819 _IOC_OUT) ((x << 8) ; y ; ((sizeof(t) &
01820 #else
01821 #define _IOCPARM_MASK) << 16) ; \
01822 /* No hay codificación _IOC_IN) ((x << 8) ! y ; ((sizeof(t) & complicada en una
01823 máquina de 16 bits. */ _IOCPARM_MASK) << 16) ! \
01824 _IOC_INOUT) ((x << 8) ; y)
01825 #define _IO(x,y)
01826 #define _IOR(x,y,t)
01827 #define _IOW(x,y,t) ffdefine _IORW(x,y,t) ffendif
01828
01829 /* loctis de IO(x ,y)
01830 terminal. ffdefine
01831 TCGETS ffdefine '
01832 TCSETS ffdefine
01833 TCSETSW ffdefine
01834 TCSETSF ffdefine IOR( 'T', 8, struct termios)
01835 TCSBRK ffdefine IOW( 'T', 9, struct termios)
01836 TCDRAIN ffdefine IOW( 'T', 10, struct termios)
01837 TCFLOW ^define TCFLSH IOW( 'T', 11, struct termios)
01838 ffdefine TIOCGWINSZ IOW( 'T', 12, int)
01839 #define TIOCSWINSZ -10( 'T', 13)
01840 #define TIOCGPGRP IOW( 'T', 14, int) /* tcgetattr */
01841 #define TIOCSPGRP IOW( 'T', 15, int) /* tcsetattr, TCSANOW */
01842 ^define TIOCSFON IOR( 'T', 16, struct winsize) /* tcsetattr, TCSADRAIN */
01843 IOW( 'T', 17, struct winsize) /* tcsetattr, TCSAFLUSH */
01844 #define TIOCGETP IOW( 'T', 18, int) /* tcsendbreak */
01845 #define TIOCSETP IOW( 'T', 19, int) /* tcdrain */
01846 ffdefine TIOCGETC IOW( 'T', 20, u8t [8192]) /* tcflow */
01847 #define TIOCSETC /* tcflush */
01848
01849 _IOR('t', 1, struct sgtyb)
01850 _IOW('t', 2, struct sgtyb)
01851 _IOR('t', 3, struct tchars)
01852 _IOW('t', 4, struct tchars)
01853
01854

```

01855					
01856	/* loctis de red. */				
01857	ffdefine NWIOSETHOPT	IOW('n	1	16,	struct nwioethopt)
01858	ffdefine NWIOGETHOPT	IOR('n	1	17,	struct' nwioethopt)
01859	ffdefine NWIOGETHSTAT	IOR('n	'	18,	struct nwioethstat)
01860					
01861	ffdefine NWIOSIPCONF	IOW('n	'	32,	struct nwioipconf)
01862	^define NWIOGIPCONF	IOR('n)	33,	struct nwioipconf)
01863	ffdefine NWIOSIPOPT	IOW('n)	34,	struct nwioipopt)
01864	#define NWIOGIPOPT	IOR('n	1	35,	struct nwioipopt)
01865					
01866	#define NWIOIPGROUTE	IORW('	n',	40,	struct nwio route)
01867	ffdefine NWIOIPSRROUTE	IOW ('	n',	41	, struct nwio route)
01868	ffdefine NWIOIPDRROUTE	IOW ('	n',	42,	struct nwio route)
01869					
01870	ffdefine NWIOSTCPCONF	IOW('n	1	48,	struct nwiotcpconf)
01871	ffdefine NWIOGTCPCONF	IOR('n	1	49,	struct nwiotcpconf)
01872	ffdefine NWIOTCPCONN	IOW('n	i	50,	struct nwiotcpcl)
01873	ffdefine NWIOTCPLISTEN	IOW('n	1	51,	struct nwiotcpcl)
01874	ffdefine NWIOTCPATTACH	IOW('n	'	52,	struct nwiotcpatt)
01875	ffdefine NWIOTCPSHUTDOWN	10 ('n)	53)	
01876	ffdefine NWIOSTCPOPT	IOW('n	1	54,	struct nwiotcpopt)
01877	ffdefine NWIOGTCPOPT	IOR('n	1	55,	struct nwiotcpopt)
01878					
01879	ffdefine NWIOSUDPOPT	IOW('n	S	64,	struct nwioudpopt)
01880	#define NWIOGUDPOPT	IOR('n	1	65,	struct nwioudpopt)
01881					
01882	/* loctis de disco. */				
01883	#define DIOCEJECT	10 ('d	1	5)	
01884	ffdefine DIOCSETP	IOW('d	1	6,	struct partition)
01885	#define DIOCGETP	IOR('d	1	7,	struct partition)
01886					
01887	/* loctis de teclado. */				
01888	#define KIOCSMAP	IOW('k	1	3,	keymapt)
01889					
01890	/* loctis de memoria */				
01891	ffdefine MIOCRAMSIZ	IOW('in	i	3,	u32t) /* Tamaño
01892	#define MIOCSPSINFO	IOW('m	1	4,	void *)
01893	ffdefine MIOCGPSINFO	IOR('m	'	5,	struct psinfo)
01894					
01895	/* loctis de cinta magnética. *		/		
01896	ffdefine MTIOCTOP	IOW('M	1	1,	struct mtop)
01897	ffdefine MTIOCGET	IOR('M)	2,	struct mtget)
01898					
01899	/* Comando SCSI. */				
01900	ffdefine SCIOCCMD	IOW('S	J	1,	struct scsicmd)
01901					
01902	/* loctis de CD-ROM. */				
01903	ffdefine CDIOPLAYTI	IOR('c	1	1,	struct cdplaytrack)
01904	ffdefine CDIOPLAYMSS	IOR('C	1	2,	struct cdplaymss)
01905	ffdefine CDIOREADTOCHDR	IOW('c	'	3,	struct cdtocentry)
01906	#define CDIOREADTOC	IOW('c	1	4,	struct cdtocentry)
01907	ffdefine CDIOREADSUBCH	IOW('c	1	5,	struct cdtocentry)
01908	#define CDIOSTOP	10 ('c)	10)	
01909	ffdefine CDIOPAUSE	10 ('c)	11)	
01910	#define CDIORESUME	10 ('c	1	12)	
01911	ffdefine CDIOEJECT	DIOCEJECT			
01912					
01913	/* loctis DSP de tarjeta	de sonido. */			
01914	ffdefine DSPIORATE	IOR('s	1	1,	unsigned int)


```

01915 #define DSPIOSTEREO _IOR('s',      2, unsigned   int)
01916 #define DSPIOSIZE    _IOR('s',      3, unsigned   int)
01917 #define DSPIOBITS    _IOR('s',      4, unsigned   int)
01918 #define DSPIOSIGN    _IOR('s',      5, unsigned   int)
01919 #define DSPIOMAX     _IOW('s',      6, unsigned   int)
01920 #define DSPIORESET   _IO ('s', 7)
01921
01922 /* loctis de mezclador   de tarjeta       de sonido.   */
01923 #define MIXIOGETVOLUME _IORW('s', 10 , struct volume level)
01924 #define MIXIOGETINPUTLEFT _IORW('S', 11 , struct inout   ctri)
01925 #define MIXIOGETINPUTRIGHT _IORW('s', 12 , str-uct inoutctrl)
01926 #define MIXIOGETOUTPUT _IORW('si, 13 , struct inout   Ctf)
01927 #define MIXIOSETVOLUME _IORW('s', 20 , struct volume level)
01928 #define MIXIOSETINPUTLEFT _IORW('s', 21 , struct inout   ctri)
01929 #define MIXIOSETINPUTRIGHT _IORW('S', 22 , struct inout   ctri)
01930 #define MIXIOSETOUTPUT _IORW('S', 23 , struct inout   ctri)
01931
01932 #ifndef ANSI
01933 #include <ansi.h>
01934 #endif
01935
01936 PROTOTYPE( int iocti, (int fd, int request , void *data)
01937
01938 #endif /* IOCTLH */
+++++
                                include/sys/sigcontext.h
+++++

02000 #ifndef SIGGONTEXTH
02001 #define SIGCONTEXTH
02002
02003 /* La estructura sigcontext es usada por la llamada sigreturn(2).
02004 * Los programas de usuario casi nunca invocan sigreturnf), pero
02005 * el mecanismo que atrapa señales puede usarla internamente.
02006 */
02007
02008 #ifndef ANSIH
02009 #include <ansi.h>
02010 #endif
02011
02012 #ifndef CONFIGH
02013 #include <minix/config.h>
02014 #endif
02015
02016 #if !defined(CHIP)
02017 #include "error, configuration is not known"
02018 #endif
02019
02020 /* La siguiente estructura debe coincidir con la estructura stackframes empleada
02021 * por el código de conmutación de contexto del kernel. Se deben agregar registros
02022 * de punto flotante en un struct distinto.
02023 */
02024 #if (CHIP == INTEL)
02025 struct sigregs {
02026     fti-f WORDSIZE == 4
02027     short srgs;
02028     short sr fs;
02029 #endif /* WORD SIZE == 4 */

```

Falta la 547

```

02090 struct sigregs sc_regs;                                /* conjunto de registros que restaurar */
02091 };
02092
02093 #if (CHIP == INTEL)
02094 #if _WORD_SIZE == 4
02095 #define sc_gs sc_regs.sr_gs
02096 #define sc_fs sc_regs.sr_fs
02097 #endif /* _WORD_SIZE == 4 */
02098 #define sc_es segrega.sr_es
02099 #define sc_ds sc_regs.ar_ds
02100 #define sc_di sc_regs.sr_di
02101 #define sc_si sc_regs.sr_si
02102 #define sc_fp sc_regs.sr_bp
02103 #define sc_st sc_regs.sr_st      /* tope de pila -- se usa en kernel */
02104 #define sc_bx sc_regs.sr_bx
02105 #define sc_dx sc_regs.sr_dx
02106 #define sc_cx sc_regs.sr_cx
02107 #define sc_retreg sc_regs.sr_retreg
02108 #define sc_retadr sc_regs.sr_retadr /* dirección de retorno al invocador
02109                                     de save -- se usa en kernel */
02110 #define sc_pc sc_regs.sr_pc
02111 #define sc_cs sc_regs.sr_cs
02112 #define sc_psw sc_regs.sr_psw
02113 #define sc_sp sc_regs.sr_sp
02114 #define sc_ss sc_regs.sr_ss
02115 #endif /* CHIP == INTEL */
02116
02117 #if (CHIP == M68000)
02118 #define sc^retreg sc_reas.sr_retreg
02119 #define sc_dl segrega.sr_dl
02120 #define sc_d2 sc_regs.sr_d2
02121 #define sc_d3 sc_regs.sr_d3
02122 #define sc_d4 sc_regs.sr_d4
02123 #define sc_d5 sc_regs.sr_d5
02124 #define sc_d6 sc^regs.sr_d6
02125 #define sc_d7 sc_regs.sr_d7
02126 #define sc_a0 sc_regs.sr_a0
02127 #define sc_a1 sc_regs.sr_a1
02128 #define sc_a2 sc_regs.sr_a2
02129 #define sc_a3 sc_regs.sr_a3
02130 #define sc_a4 sc_regs.sr_a4
02131 #define sc_a5 sc_regs.sr_a5
02132 #define sc_fp sc__regs.sr_a6
02133 #define sc_sp sc_regs.sr_sp
02134 #define sc_pc sc_regs.sr_pc
02135 #define sc_psw sc_regs.sr_psw
02136 #endif /* CHIP == M68000 */
02137
02138 /* Valores para sc__flags. Deben coincidir con <niinis</jmp_üuf.h>. */
02139 #define SC_SIGCONTEXT 2 /* no cero si se incluye contexto de señal */
02140 #define SC_NOREGLOCALS 4 /* no cero si no se deben guardar
02141                             y restaurar los registros */
02142
02143 _PROTQTYPE( int sigreturn, (struct sigcontext * _scp) );
02144
02145 #endif /* SIGCONTEXT H */

```

```
include/sys/ptrace.h
```

```
02200 /* <sys/ptrace.h>
02201 * definiciones para ptrace(2)
02202 */
02203
02204 #ifndef _PTRACE_H
02205 #define _PTRACE_H
02206
02207 #define T_STOP          -1          /* detener el proceso */
02208 #define T_OK            0          /* habilitar rastreo del padre de este proc. */
02209 #define T_GETINS       1          /* devolver valor de espacio de instr- */
02210 #define T_GETDATA      2          /* devolver valor de espacio de datos */
02211 #define T_GETURSE      3          /* devolver valor de tabla proc. del usuario */
02212 #define T_SETINS       4          /* fijar valor de espacio de instr. */
02213 #define T_SETDATA      5          /* fijar valor de espacio de datos */
02214 #define T_SETURSE      6          /* fijar valor en tabla proc. del usuario */
02215 #define T_RESUME       7          /* reanudar ejecución */
02216 #define T_EXIT         8          /* salir */
02217 #define T_STEP         9          /* fijar bit de rastreo */
02218
02219 /* Prototipos de funciones. */
02220 #ifndef _ANSI_H
02221 #include <ansi.h>
02222 #endif
02223
02224 _PROTOTYPE( long ptrace, (int _req, pid_t _pid, long _addr, long _data) );
02225
02226 #endif /* PTRACE H */
```

```
include/sys/stat.h
```

```
92300 /* La cabecera <sys/stat.h> define un struct que se usa en las funciones stat() y
92301 * fstat. La información de este struct proviene del nodo-i de algún archivo.
92302 * Éstas son las únicas llamadas aprobadas para inspeccionar nodos-i.
92303 */
92304
92305 #ifndef STAT_H
92306 #define ^STATH
92307
92308 struct stat {
92309     dev_t stdev;          /*núm. disp. principal/segund, */
92310     ino_t stino;         /* número de nodo-i */
92311     mode_t stmode;      /* modo de aren., bits protección, etc.*/
92312     short int stnlink;  /* # vínculos; HACK TEMPORAL: debe ser nlinkt */
92313     uid_t stuid;        /* uid del dueño del archivo */
92314     short int stgid;    /* gid: HACK TEMPORAL: debe ser gidt */
92315     dev_t st^rdev;
92316     off_t stsize;      /* tamaño de archivo */
92317     time_t stmtime;    /* tiempo de último acceso */
92318     time_t stmtime;    /* tiempo de últ. modif. de datos */
92319     time_t stctime;    /* tiempo de últ. cambio de situac. aren. */
```

```

02320 };
02321
02322 /*Definiciones de máscara tradicionales para stmode. */
02323 /* Las mutaciones feas en algunas definiciones son para evitar extensiones de signo
02324 * sorprendentes como S IFREG != (mode t) S IFREG cuando los ints son de 32 bits.
02325 */
02326 #define S IFMT ((modet) 0170000) /* tipo de archivo */
02327 #define S^IFREG ((modet) 0100000) /* normal */
02328 #define SIFBLK 0060000 /* especial por bloques */
02329 #define SIFDIR 0040000 /* directorio */
02330 #define S IFCHR 0020000 /* especial por caracteres */
02331 #define SIFIFO 0010000 /* éste es un FIFO */
02332 #define SISUID 0004000 /* fijar id de usuario al ejecutar- */
02333 #define SISGID 0002000 /* fijar id de grupo al ejecutar */
02334 /* siguiente reservado para uso futuro */
02335 #define SISVTX 01000 /* guardar texto intercamü. aun desp. uso */
02336
02337 /* Máscaras POSIX para st_mode. */
02338 #define S_IRWXU 00700 /* dueño:  rwx ---- */
02339 #define S_IRUSR 00400 /* dueño:  r----- */
02340 #define S_IWUSR 00200 /* dueño:  -w----- */
02341 #define S_IXUSR 00100 /* dueño:  --x----- */
02342
02343 #define S_IRWXG 00070 /* grupo:  ---rwx--- */
02344 #define S_IRGRP 00040 /* grupo:  ---r----- */
02345 #define S_IWGRP 00020 /* grupo:  ----w---- */
02346 #define S_IXGRP 00010 /* grupo:  -----x--- */
02347
02348
02348 #define S_IRWXO 00007 /* otros:  -----rwx */
02349 #define S_IROTH 00004 /* otros:  -----r--- */
02350 #define S_IWOTH 00002 /* otros:  -----w- */
02351 #define S_IXOTH 00001 /* otros:  -----x */
02352
02353 /* Estas macros prueban st mode (de POSIX Sec. 5.6.1.1). */
02354 #define S ISREG(m) (((m) & S^IFMT) == S^IFREG) /* es aren. normal */
02355 #define S ISDIR(m) f(m) & S IFMT == S IFDIR /* es directorio */
02356 #define S ISCHR(m) (((m) & S IFMT) == S IFCHR) /* es esp. p/car. */
02357 #define S ISBLK(m) (((m) & S IFMT) == S IFBLK) /* es esp. p/bloques */
02358 #define S^ISFIFO(m) (((m) & S IFMT) == S^IFIFO) /* es conducto/FIFO */
02359
02360
02361 /* Prototipos de funciones. */
02362 #ifndef ANSIH
02363 #include <ansi.h>
02364 #endif
02365
02366 _PROTOTYPE( int chmod, (const char *path, Modet mode) );
02367 _PROTOTYPE( int fstat, (int fi-Ldes, struct stat *buf) );
02368 _PROTOTYPE( int mkdir, (const char *path, Modet mode) );
02369 _PROTOTYPE( int mkfifo, (const char *path, Mode^t mode) );
02370 _PROTOTYPE( int stat, (const char *path, struct stat *buf) );
02371 _PROTOTYPE( modet umask, (Modet cmask) );
02372
02373 #endif /* STATH */

```

```

+++++
                                     include/sys/dir.h
+++++
02400 02401  /* La cabecera <dir.h> da la organización de un directorio. */
02402 #ifndef DIR H
02403 #define DIRH
02404
02405 #define DIRBLKSIZ 512 /* tamaño de bloque de directorio */
02406
02407 #ifndef DIRSIZ
02408 #define DIRSIZ 14
02409 #endif
02410
02411 struct direct {
02412     ino_t dino;
02413     char d_name[DIRSIZ];
02414 };
02415
02416 #endif /* DIRH */

+++++
                                     include/sys/wait.h
+++++

02500 /* La cabecera <sys.wait.h> contiene macros relacionadas con wait(). El valor
02501 * devuelto por wait() y waitpid() depende de si el proceso terminó por exit(),
02502 * fue terminado por una señal o se detuvo por control de trabajos, como
02503 * sigue:
02504 *
02505 *          Byte alto      Byte bajo
02506 *          +-----+
02507 *   exit(situac.)   | situac. | 0 |
02508 *          +-----+
02509 * terminado por señal | 0 | señal |
02510 *          +-----+
02511 * detenido (ctl. trab.) | situac. | 0 |
02512 *          +-----+
02513 */
02514
02515 #ifndef WAIT H
02516 #define WAITH
02517
02518 #define LOW(V) ((V) & 0377)
02519 #define ^HIGH(V) (((V) » 8) & 0377)
02520
02521 #define WNOHANG 1 /* no esperar a que el hijo salga */
02522 #define WUNTRACED 2 /* para ctl. trab.; no implementado */
02523
02524 #define WIFEXITED(s) (_LOW(s) == 0) /* salida normal */
02525 #define WEXITSTATUS(s) (_HIGH(s)) /* situac. salida */
02526 #define WTERMSIG(s) (_LOW(S) & 0177) /* valor señal */
02527 #define WIFSIGNALED(S) (((unsigned int)(s)-1 (LOW(s) == 0177) (HIGH(s) & 0377) & 0xFFFF) < 0xFF) /* señalizado */
02528 #define WIFSTOPPED(S) (_LOW(s) == 0177) /* detenido */
02529 #define WSTOPSIG(s) (_HIGH(s) & 0377) /* señal de paro */

```

```

02530
02531 /* Prototipos de funciones. */
02532 #ifndef _ANSI_H
02533 #include <ansi.h>
02534 #endif
02535
02536 _PROTOTYPE( pid_t wait, (int * _stat_loc)
02537 _PROTOTYPE( pid_t waitpid, (pid_t _pid, int * _stat_loc, int _options)
02538
02539 #endif /* _WAIT_H */
+++++
                                     include/minix/config.h
+++++

02600 #ifndef CONFIG_H
02601 #define GONFIQH
02602
02603 /* Números de versión de Minix. */
02604 #define OS RELEASE "2.0"
02605 #define OSVERSION "0"
02606
02607 /* Este archivo fija parámetros de config, para el kernel, FS y MM de MINIX.
02608 * Se divide en 2 secciones principales. La primera contiene parámetros ajustables
02609 * por el usuario. En la segunda se fijan varios parámetros internos del sistema
02610 * con base en los parámetros ajustables por el usuario.
02611 */
02612
02613 /*=====*/
02614 *      Esta sección contiene parámetros ajustables por el usuario      *
02615 /*=====*/
02616 #define MACHINE    IBMPC          /* Debe ser uno de los nombres siguientes */
02617
02618 #define IBMPC      1              /* cualquier sistema 8088 u 80x86 */
02619 #define SUN_4      40            /* cualquier sistema Sun SPARC */
02620 #define SUN_4_60   40            /* Sun-4/60 (alias SparcStation 1 o Campus */
02621 #define ATARI      60            /* ATARI ST/STe/TT (68000/68030) */
02622 #define AMIGA      61            /* Commodore Amiga (68000) */
02623 #define MACINTOSH  62            /* Apple Macintosh (68000) */
02624
02625 /* Tamaño de palabra en bytes (constante igual a sizeof(int)). */
02626 #if _ACK_
02627 #define _WORD_SIZE    _EM_WSIZE
02628 #endif
02629
02630
02631 /* Si ROBUST es 1, se escriben en disco bloques de nodos-i, directorio e indirectos
02632 * del caché tan pronto como se modifican. Esto hace al FS más robusto pero más
02633 * lento. Si es 0, esos bloques no reciben tratamiento especial, y podría haber
02634 * problemas si se cae el sistema.
02635 */
02636 #define ROBUST 0 /* 0 para rapidez, 1 para robustez */
02637
02638 /* Núm. de ranuras de la tabla de procesos para procesos de usuario. */
02639 #define NR_PROCS    32

```

```

02640
02641 /* El caché de buffers debe hacerse tan grande como sea costeable. */
02642 #if (MACHINE == IBM_PC && _WORD_SIZE == 2)
02643 #define NR_BUFS          40 /* # núm. bloques en el caché */
02644 #define NR_BUF_HASH     64 /* tamaño tabla disp. buf.; DEBE SER POT. DE 2*/
02645 #endif
02646
02647 #if (MACHINE == IBM_PC && _WORD_SIZE == 4)
02648 #define NR^BUFS          512 /* # núm. bloques en el caché */
02649 #define NR_BUF_HASH     1024 /* tamaño tabla disp. buf.; DEBE SER POT. DE 2*/
02650 #endif
02651
02652 #if (MACHINE == SUN_4_60)
02653 #define NR_BUFS          512 /* # núm. bloques en el caché (<=1536) */
02654 #define NR_BUF_HASH     512 /* tamaño tabla disp, buf.; DEBE SER POT. DE 2*/
02655 #endif
02656
02657 #if (MACHINE == ATARI)
02658 #define NR_BUFS          1536 /* # núm. bloques en el caché (<=1536) */
02659 #define NR_BUF_HASH     2048 /* tamaño tabla disp, buf.; DEBE SER POT. DE 2*/
02660 #endif
02661
02662 /* Definiciones para la configuración del kernel. */
02663 #define AUTO_BIOS          0 /* xt_wini.c - usar BIOS autoconfig de Western */
02664 #define LINEWRAP          1 /* consolé.c - continuar líneas en col. 80 */
02665 #define ALLOW_GAP_MESSAGES 1 /* proc.c - permitir mensajes en espacio
02666                               * entre fin de bss y dir. de pila más baja */
02667
02668 /* Habilitar o no el caché de segundo nivel del FS en el disco de RAM */
02669 #define ENABLE_CACHE2     0
02670
02671 /* Incluir o excluir controladores de disp. 1 para incluir, 0 para excluir. */
02672 #define ENABLE_NETWORKING 0 /* habilitar código TCP/IP */
02673 #define ENABLE_AT_WINI    1 /* habilitar contr. winchester AT */
02674 #define ENABLE_BIOS_WINI 0 /* habilitar contr. winchester BIOS */
02675 #define ENABLE_ESDI_WINI 0 /* habilitar contr. winchester ESDI */
02676 #define ENABLE_XT_WINI   0 /* habilitar contr. winchester XT */
02677 #define ENABLE_ADAPTEC_SCSI 0 /* habilitar contr. SCSI ADAPTEC */
02678 #define ENABLE_MITSUMI_CDROM IM 0 /* habilitar contr. CD-ROM Mitsumi */
02679 #define ENABLE_SB_AUDIO 0 /* habilitar contr. audio Soundblaster
02680
02681 /* DMA_SECTORS puede aumentarse para acelerar controladores basados en DMA. */
02682 #define DUA_SECTORS      1 /* Tamaño buf. DMA (debe ser >= 1) */
02683
02684 /* incluir o excluir código para compatibilidad hacia atrás. */
02685 #define ENABLE_BINCOMPAT 0 /* p/binarios que usan llamadas obsoletas */
02686 #define ENABLE_SRCCOMPAT 0 /* p/fuentes que usan llamadas obsoletas */
02687
02688 /* Determinar qué dispositivo usar para conductos. */
02689 #define PIPE_DEV ROOT_DEV /* conductos en dispositivo raíz */
02690
02691 /* NR_CONS, NR_RS_LINES y NR_PTYS determinan el número de terminales que el
02692  * sistema puede manejar.
02693  */
02694 #define NR_CONS          2 /* # consolas de sistema (1 a 8) */
02695 #define NR_RS_LINES      0 /* # terminales rs232 (0, 1 o 2) */
02696 #define NR_PTYS          0 /* # seudoterminales (0 a 64) */
02697
02698 #if (MACHINE == ATARI)
02699 /* El sig. define dice si se tiene un ATARI ST o TT */

```



```

02700 #define ATARI_TYPE      TT
02701 #define ST              1 /* todos ST y Mega ST */
02702 #define STE              2 /* todos STE y Mega STE */
02703 #define TT              3
02704
02705 /* si SCREEN es 1 son posibles operaciones gráficas en pantalla */
02706 #define SCREEN          1
02707
02708 /* Este define dice si el teclado genera escapes VT100 o IBM_PC. */
02709 #define KEYBOARD        VT100 /* VT100 o bien IBM_PC */
02710 #define VT100           100
02711
02712 /* El siguiente define determina el tipo de partición. */
02713 #define PARTITIONING    SUFRA /* uno de los siguientes o ATARI */
02714 #define SUFRA           1 /*ICD, SUFRA y BMS son iguales */
02715 #define BMS             1
02716 #define ICD             1
02717 #define CBHD            2
02718 #define EICKMANN        3
02719
02720 /* Definir número de unidades de disco duro en el sistema. */
02721 #define NR_ACSIJ)RIVES  3 /* típicamente 0 o 1 */
02722 #define NR SCSI_DRIVES 1 /* típicamente 0 (ST, STE) o 1 (TT) */
02723
02724 /* Algunos sistemas requieren un retardo breve después de cada comando winchester;
02725 * en ellos FAST_DISK debe ser 0. Otros discos no lo necesitan, y puede ponerse
02726 * FASTJ3ISK en 1 para evitar el retardo.
02727 */
02728 #define FAST_DISK      1 /* 0 o 1 */
02729
02730 /* Mota: si quiere hacer más chico el kernel, puede hacer NR_PD_DRIVES r- 0. Aún
02731 * podrá arrancar minix.img de disquete. Sin embargo, DEBERÁ traer los sistemas de
02732 * archivos root y usr de un disco duro.
02733 */
02734
02735 /* Definir número de unidades de disquete en el sistema. */
02736 #define NR_FD_DRIVES   1 /* 0, 1, 2 */
02737
02738 /* Este define de configuración controla el código de impresora en paralelo. */
02739 #define PAR_PRINTER    1 /* inhab. (0) / hábil. (1) impr. paralelo */
02740
02741 /* Este define de configuración controla código de reloj de contr. de disco. */
02742 #define HD_CLOCK       1 /* inhab. (0) / hábil. (1) reloj disco duro */
02743
02744 #endif
02745
02746
02747 * No hay parámetros ajustables por el usuario después de esta línea *
02748 /* Fijar tipo CHIP con base en la máquina escogida. CHIP indica más que sólo la
02749 * CPU. P. ej., se espera que las máquinas con CHIP == INTEL tengan controladores
02750 * de interrupciones 8259A y demás propiedades de las máquinas tipo IBM PC/XT/AT/
02751 * 386 en general. */
02752 #define INTEL           1 /* Tipo GHIP para PC, XT, AT, 386 y clones */
02753 #define M68000          2 /* Tipo CHIP para Atari, Amiga, tiacintosh */
02754 #define SPARC           3 /* Tipo CHIP para SUN-4 (p.ej. SPARCstation)*/
02755
02756 /* Fijar tipo FP_FORMAT con base en la máquina escogida, hw o bien sw */
02757 #define FP_NONE         0 /* no apoya punto flotante */

```

```
02760 #define FP_IEEE      1    /* se ajusta a norma IEEE de pto. flot.    */
02761
02762 #if (MACHINE == IBM_PC)
02763 #define CHIP          INTEL
02764 #define SHADOWING    0
02765 #define ENABLE_WINI  (ENABLE_AT_WINI || ENABLE_BIOS_WINI || \
02766                     ENABLE_ESDI_WINI || ENABLE_XT_WINI)
02767 #define ENABLE_SCSI (ENABLE_ADAPTEG_SCSI)
02768 #define ENABLE_CDROM (ENABLE_MITSUMI_CDROM)
02769 #define ENABLE_AUDIO (ENABLE_SB_AUDIO)
02770 #endif
02771
02772 #if (MACHINE == ATARI) || (MACHINE == AMIGA) || (MACHINE == MACINTOSH)
02773 #define CHIP          M68000
02774 #define SHADOWING    1
02775 #endif
02776
02777 #if (MACHINE == SUN_4) ;; (MACHINE == SUN_4_60)
02778 #define CHIP          SPARC
02779 #define FP_FORMAT    FP_IEEE
02780 #define SHADOWING    0
02781 #endif
02782
02783 #if (MACHINE == ATARI) || (MACHINE == SUN_4)
02784 #define ASKDEV 1      /* preguntar por disp. de arranque */
02785 #define FASTLOAD 1   /* usar múltiples transf. bloque para inic. ram */
02786 #endif
02787
02788 #if (ATARI_TYPE == TT) /* y demás 68030 */
02789 #define FPP
02790 #undef SHADOWING
02791 #define SHADOWING 0
02792 #endif
02793
02794 #ifndef FP_FORMAT
02795 #define FP_FORMAT    FP_NONE
02796 #endif
02797
02798 /* El archivo buf.h usa MAYBEJWRITE_IMMED. */
02799 #if ROBUST
02800 #define MAYBE_WRITE_IMMED WRITE_IMMED /* más lento pero quizá más seguro */
02801 #else
02802 #define MAYBE_WRITE_IMMED 0           /* más rápido */
02803 #endif
02804
02805 #ifndef MACHINE
02806 error "In <minix/config.h> please define MACHINE"
02807 #endif
02808
02809 #ifndef CHIP
02810 error "In <minix/config.h> please define MACHINE to have a legal value"
02811 #endif
02812
02813 #if (MACHINE == 0)
02814 error "MACHINE has incorrect value (0)"
02815 #endif
02816
02817 #endif /* CONFIG H */
```

```

+++++
                                     include/minix/const.h
+++++

02900 /* derechos reservados © por Prentice-Hall, Inc. Se concede permiso para
02901 * redistribuir los programas binarios y fuente de este sistema para fines
02902 * educativos o de investigación. Para otro uso se requiere permiso por
02903 * escrito de Prentice-Hall .
02904 */
02905
02906 #define EXTERN      extern /* empleado en archivos *.h */
02907 #define PRÍVATE    static /* PRÍVATE x es el límite de escape de x */
02908 #de-fine PUBLIC    /* PUBLIC es lo opuesto de PRÍVATE */
02909 #define FORWARD    static /* algunos compil. requieren que sea 'static'*/
02910
02911 #define TRUE  1      /* sirve para convertir enteros en booleanos */
02912 #define FALSE 0     /* sirve para convertir enteros en booleanos */
02913
02914 #define HZ          60      /* frec. reloj (ajustable por sof. en IBM-PC */
02915 #define BLOCK^SIZE 1024    /* núm. bytes en un bloque de disco */
02916 #de-fine SUPER_USER(uidt) 0 /* uidt de superusuario */
02917
02918 #define MAJOR      8      /* disp. princ. = (dev»MAJOR) & 0377 */
02919 #define MIÑOR     0      /* disp. secund. = (dev»MINOR) & 0377 */
02920
02921 #define NULL ( (void *)0) /* apuntador nulo */
02922 #define CPVEC NR      16  /* máx. entradas en solicitud SYS VCOPY */
02923 #define NRIOREQS     MIN(NR      BUFS, 64)
02924 /* máx. entradas en solicitud de E/S */
02925
02926 #define NR SEGS      3      /* núm. segmentos por proceso */
02927 #define T            0      /* proc[i].memmap[T] es p/texto */
02928 #define D            1      /* proc[i].memmap[D] es p/datos */
02929 #define S            2      /* proc[i].memmap[S] es p/pila */
02930
02931 /* Núms. de proceso de algunos procesos importantes. */
02932 #define MM_PROC_NR  0      /* núm. proc. del adm. de memoria */
02933 #define FS_PROC_NR  1      /* núm. proc. del sist. de aren. */
02934 #define INET_PROC_NR 2     /* núm. proc. del servidor TCP/IP */
02935 #define INIT_PROC_NR (INET  PROGNR + ENABLENETWORKING)
02936 /* init -- el proceso para mu Itius Liarlo */
02937 #define LOWJJSER    (INET  PROCNR + ENABLENETWORKING)
02938 /* ler. usuario no es parte del sist. op. */
02939
02940 /* Diversos */
02941 #define BYTE 0377 /* máscara para 8 bits */
02942 #define READING 0 /* copiar datos en usuario */
02943 #define WRITING 1 /* copiar datos de usuario */
02944 #define NO NUM 0x8000 /* argumento numérico para panic() */
02945 #define NIL PTR (char *) ó /* expr. de utilidad general */
02946 #define HAVESCATTEREDIO 1 /* E/S dispersa ahora es estándar */
02947
02948 /* Macr-os. */
02949 #define MAX(a, b) ((a) > (b) ? (a) : (b))
02950 #define MIN(a, b) ((a) < (b) ? (a) : (b))
02951
02952 /* Número de tareas, */
02953 #define NRTASKS (9 + ENABLEWINI + ENABLESCSI + ENABLECDROM \
02954 + ENABLENETWORKING + 2 * ENABLEAUDIO)

```

```

02955
02956 /* La memoria se asigna en clics. */
02957 #if fCHIP == INTEL)
02958 #define CLICK SIZE 256 /* unidad de asignación de memoria */
02959 #define CLICK SHIFT 8 /* log2 de GLICK SIZE */
02960 #endif
02961
02962 #if (CHIP == SPARC) ; (CHIP == M68000)
02963 #define CLICK SIZE 4096 /* unidad de asignación de memoria */
02964 #define CLICK SHIFT 12 /* 2log de CLICK SIZE */
02965 #endif
02966
02967 #define clicl<toroundk(n) \
02968     ((unsigned) (((unsigned long) (n) « CLIGK SHIFT) + 512) / 1024))
02969 #if CLIGK SIZE < 1024
02970 #define kJtoclick(n) ((n) * (1024 / CLICKSIZE))
\ 02971 #else
02972 #define ktoclick(n) ((n) / (CLICK SIZE / 1024))
i 02973 #endif
l 02974
02975 #define ABS -999 /* este proc. implica memoria absoluta */
/ 02976
02977 /* Bits de bandera para imode en el nodo-i. */
02978 #define ITYPE          0170000 /* tipo de nodo-i */
02979 #define IRREGULAR     0100000 /* arch. normal, no dir. ni especial */
02980 #define IBLOCKSPECIAL 0060000 /* archivo especial por bloques */
02981 #define IDIRECTORY    0040000 /* el arch. es un directorio */
02982 #define ICHARSPECIAL  0020000 /* arch. especial por caracteres */
02983 #define INAMEDPIPE    0010000 /* conducto con nombre (FIFO) */
02984 #define ISETUIDBIT     0004000 /* fijar uidt efectivo en exec */
02985 #define ISETGIDBIT    0002000 /* fijar gidt efectivo en exec */
02986 #define ALL MODES     0006777 /* todos bits para usuario, grupo y otros */
02987 #define RWXMODES      0000777 /* bits de modo sólo para RWX */
02988 #define RBIT          0000004 /* bit de protección Rwx */
02989 #define WBIT          0000002 /* bit de protección rWx */
02990 #define XBIT          0080001 /* bit de protección rwX */
02991 #define INOTALLOC     0000000 /* este nodo-i está libre */
02992
02993 /* Algunos limites. */
02994 #define MAXBLOCK NR   ((block t) 07777777) /* núm. de bloque más grande */
02995 #define HIGHESTZONE  ((zonet) 07777777) /* núm. de zona más grande */
02996 #define MAX INODE NR  ((ino t) 0177777) /* núm. de nodo-i más grande */
02997 #define MAXFILEPOS   ((offt) 0377777777) /* /- máx. distancia de arch. legal */
02998
02999 #define NOBLOCK       ((block t) 0) /* ausencia de núm. de bloque */
03000 #define NOENTRY       ((inot) 0) /* ausencia de entrada de dir. */
03001 #define NO ZONE       ((zonet) 0) /* ausencia de núm. de zona */
03002 #define NODEV         ((devt) 0) /* ausencia de núm. disp. */

```

```

+++++
                                     include/minix/type.h
+++++

```

```

03100 #ifndef TYPE H
03101 #define TYPE H
03102 #ifndef MINIX TYPE H
03103 #define MINIXTYPEH
03104
03105 /* Definiciones de tipos. */
03106 typedef unsigned int virclicks; /* dir. virtuales y longitudes en clics */
03107 typedef unsigned long physbytes; /* dir. físicas y longitudes en bytes */
03108 typedef unsigned int pnysclicks; /* dir. físicas y longitudes en clics */
03109
03110 #if (CHIP == INTEL)
03111 typedef unsigned int vir üytes; /* dir. virtuales y longitudes en bytes */
03112 #endif
03113
03114 #if (CHIP == M68000)
03115 typedef unsigned long virbytes; /* dir. virtuales y longitudes en bytes */
03116 #endif
03117
03118 #if (CHIP == SPARC)
03119 typedef unsigned long vir bytes; /* dir. virtuales y longitudes en bytes */
03120 #endif
03121
03122 /* Tipos relacionados con mensajes. */
03123 #define M1 1
03124 #define M3 3
03125 #define M4 4
03126 #define M3STRING 14
03127
03128 typedef struct {int mli1, mli2, mli3; char *m1p1, *m1p2, *m1p3;} mess1;
03129 typedef struct {int m2i1, m2i2, m2i3; long m2n , m2i2; char 'tm2p1;} meS52;
03130 typedef struct {int tn3J.1, m3i2; cnar *m3p1; char m3cal [M3STRING];} mess3;
03131 typedef struct {long m4l1, m4l2, ni4l3, m4l4, m4l5;} mess4;
03132 typedef struct {char m5c1, m5c2; int in5i1, m5i2; long m5l1, m5l2, ni5l3;} mess5
03133 typedef struct {int mGi1, m6i2, m6i3; long m6l1; sighandler't ni6f1;} mess6;
03134
03135 typedef struct {
03136 int msource; /* quién envió el mensaje */
03137 int mtype; /* qué clase de mensaje es */
03138 unión {
03139 mess_1_mml;
03140 mess_2_mm2;
03141 mess_3_mm3;
03142 mess_4_mm4;
03143 fness_5_mm5;
03144 mess_6_mm6;
03145 } m_u;
03146 } message,
03147
03148 /* Los siguientes define proporcionan nombres de miembros útiles. */
03149 #define ml_i1 m_u.m_ml.mli1
03150 #define ml_i2 m_u.m_ml.mli2
03151 #define mli_3 m_u.m_ml.mli3
03152 #define ml_p1 m_u.m_ml.mlp1
03153 #define ml_p2 m_u.m_ml.mlp2
03154 #define ml_p3 m_u.m_ml.mlp3

```

```

03155
03156 #define m2_il m_u.m_m2.m2i1
03157 #define m2_i2 m_u.m_m2.m2i2
03158 #define m2_i3 m_u.m_m2.m2i3
03159 #define m2_l1 m_u.m_m2.m2l1
03160 #define m2_l2 m_u.m_m2.m2l2
03161 #define m2_pl m_u.m_m2.m2pl
03162
03163 #define m3_il m_u.m_m3.m3i1
03164 #define m3_i2 m_u.m_m3.m3i2
03165 #define m3_pl m_u.m_m3.m3pl
03166 #define m3_cal m_u.m_m3.m3cal
03167
03168 #define m4_l1 m_u.m_m4.m4l1
03169 #define m4_l2 m_u.m_m4.m4l2
03170 #define m4_l3 m_u.m_m4.m4l3
03171 #define m4_l4 m_u.m_m4.m4l4
03172 #define m4_l5 m_u.m_m4.m4l5
03173
03174 #define m5_c1 m_u.m_m5.m5c1
03175 #define m5_c2 m_u.m_m5.m5c2
03176 #define m5_i1 m_u.m_m5.m5i1
03177 #define m5_i2 m_u.m_m5.m5i2
03178 #define m5_l1 m_u.m_m5.m5l1
03179 #define m5_l2 m_u.m_m5.m5l2
03180 #define m5_l3 m_u.m_m5.m5l3
03181
03182 #define m6_il m_u.m_m6.m6i1
03183 #define m6_i2 m_u.m_m6.m6i2
03184 #define m6_i3 m_u.m_m6.m6i3
03185 #define m6_l1 m_u.m_m6.m6l1
03186 #define m6_pl m_u.m_m6.m6pl
03187
03188 struct memmap {
03189     virclicks memvir;      /* dirección virtual */
03190     physclicks memphys;   /* dirección física */
03191     vir clicks mem len;    /* longitud */
03192 };
03193
03194 struct iorequests {
03195     long ioposition; /* pos. en aren. de disp. (realmente off) */
03196     char *iobuf;     /* buffer en espacio de usuario */
03197     int io nbytes;   /* tamaño de solicitud */
03198     unsigned short io request; /* leer, escribir (opcionalmente) */
03199 };
03200 #endif /* TYPEH */
03201
03202 typedef struct {
03203     virbytes iovaddr; /* dirección de un buffer de E/s */
03204     virbytes iovsize; /* tamaño de un buffer de E/S */
03205 } iovec;
03206
03207 typedef struct {
03208     virbytes cpvsrc; /* dirección origen de datos */
03209     virbytes cpvdst; /* dirección destino de datos */
03210     virbytes cpvsize; /* tamaño de datos */
03211 } cpvect;
03212
03213 /* MM pasa la dirección de una estructura de este tipo a KERNEL cuando se invoca
03214 * dosendsig() como parte del mecanismo para atrapar señales.

```

```

03215 * La estructura contiene toda la información que KERMEL necesita para construir
03216 * la pila de señales.
03217 */
03218 struct sigmsg {
03219     int sm_signo;           /* núm. de señal que se atrapa */
03220     unsigned long sm_mask; /* máscara para restaurar al regresar */
03221     vir_bytes sm_sighandler /* dirección del manejador */
03222     vir_bytes sm_sigreturn; /* dir. de _sigreturn en biblioteca C */
03223     vir_bytes sm_stkptr;   /* apuntador a pila de usuario */
03224 };
03225
03226 #define MESS_SIZE (sizeof(message)) /* podría necesitar sizeof de fs aquí */
03227 #define NIL_MESS ((message *) 0)
03228
03229 struct psinfo { /* información para el programa ps(1) */
03230     u16_t nr_tasks, nr_procs; /* constantes NR_TASKS y NR_PROCS. */
03231     vir_bytes proc, mproc, fproc; /* dir. de las tablas de proc. princ. */
03232 };
03233
03234 #endif /* MINIX TYPE H */

```

```

+++++
include/minix/sysiib.h
+++++

```

```

03300 /* Prototipos de funciones de la biblioteca del sistema. */
03301
03302 #ifndef _SYSLIB_H
03303 #define _SYSLIB_H
03304
03305 /* Ocultar nombres para no contaminar espacio de nombres. */
03306 #define sendrec _sendrec
03307 #define receive _receive
03308 #define send _send
03309
03310 /* Biblioteca de usuario+sistema de Minix. */
03311 _PROTOTYPE( void printk, (char * _fmt, ...) );
03312 _PROTOTYPE( int sendrec, (int _src_dest, message * _m_ptr~) );
03313 _PROTOTYPE( int _taskcall, (int _who, int _syscallnr, message * _tnsgptr) );
03314
03315 /* Biblioteca de sistema de Minix. */
03316 _PROTOTYPE( int receive, (int _src, message * _m_ptr) );
03317 _PROTOTYPE( int send, (int _dest, message * _m_ptr-) );
03318
03319 _PROTOTYPE( int sys_abort, (int _how, ...) );
03320 _PROTOTYPE( int sys_adjmap, (int _pi~oc, struct mem_map * _ptr,
03321     vir_clicks_data_clicks, vir_clicks_sp) );
03322 _PROTOTYPE( int sys_copy, (int _sr-c_proc, int _src_seg, pnys_bytes _src_vir,
03323     int _dst_proc, int _dst_seg, phys_bytes _dst_vir, phyc_bytes _bytes));
03324 _PROTOTYPE( int sys_exec, (int _proc, char * _ptr, int _traced,
03325     char * _aout, vir_bytes _initpc) );
03326 _PROTOTYPE( int sys_execmap, (int _proc, struct mem_map * _ptr) );
03327 _PROTOTYPE( int sys^fork, (int _parent, int _child, int _pid,
03328     phys_clicks _shadow) );
03329 _PROTOTYPE( int sys_fresh, (int _proc, struct mem^map * _ptr,

```

```

03330         phys_clicks * _basep, phys_clicks * _sizep) );
03331 _PROTOTYPE( in-t sys_getsp, (int _proc, vir_bytes * _newsp) );
03332 _PROTOTYPE( int sys_newmap, (int _proc, struct mem_map * _ptr) );
03333 _PROTOTYPE( int sys_getmap, (int _proc, struct mem_map * _ptr) );
03334 _PROTOTYPE( int sys_sendsig (int _proc, str-uct sigmsg * _ptr) );
03335 _PROTOTYPE( int sys_oldsig, (int _proc, int _sig, sighandler_t __sighandler));
03336 _PROTOTYPE( int sys_endsig, (int _proc) );
03337 _PROTOTYPE( int sys_sigretu , (int _proc, vir_bytes _scp, int _flags) );
03338 _PROTOTYPE( int sys_trace, (int _req, int _procnr, long _addr, long "_data_p));
03339 _PROTOTYPE( int sys_xit, (int _parent, int _pr'oc, phys_clicks * _basep,
03340         phys_clicks * _sizep));
03341 _PROTOTYPE( int syskill, (int _proc, int _sig) );
03342 _PROTOTYPE( int systimes, (int _proc, clock_t _ptr[5]) );
03343
03344 #endif /* SYSLIBH */

```

```

+++++
include/minix/calnr.h
+++++

```

```

03400 #define NCALLS          77 /* núm. De llamadas al sist. Permitidas */
03401
03402 #define EXIT            1
03403 #define FORK           2
03404 #define READ           3
03405 #define WRITE          4
03406 #define OPEN           5
03407 #define GLOSE          6
03408 #define WAIT           7
03409 #define CREAT          8
03410 #define LINK           9
03411 #define UNLINK         10
03412 #define WAITPID        11
03413 #define CHDIR          12
03414 #define TIME           13
03415 #define MKNOD          14
03416 #define CHMOD          15
03417 #define CHOWN          16
03418 #define BRK            17
03419 #define STAT           18
03420 #define LSEEK          19
03421 #define GETPID         20
03422 #define MOUNT          21
03423 #define UMOUNT         22
03424 #define SETUID         23
03425 #define GETUID         24
03426 #define STIME          25
03427 #define PTRACE         26
03428 #define ALARM          27
03429 #define FSTAT          28
03430 #define PAUSE          29
03431 #define UTIME          30
03432 #define ACCESS         33
03433 #define SYNC           36
03434 #define KILL           37

```



```

03435 #define RENAME      38
03436 #define MKDIR       39
03437 #define RMDIR       40
03438 #define DUP          41
03439 #define PIPE         42
03440 #define TIMES        43
03441 #define SETGID       46
03442 #define GETGID       47
03443 #define SIGNAL       48
03444 #define IOCTL        54
03445 #define FCNTL        55
03446 #define EXEC         59
03447 #define UMASK        60
03448 #define CHROOT       61
03449 #define SETSID       62
03450 #define GETPGRP      63
03451
03452 /* Las siguientes no son llamadas al sistema, pero se procesan como tales*/
03453 #define KSIG          64 /* ker-nel detectó una señal */
03454 #define UN PAUSE     65 /* a MM o FS; comprobar EINTR */
03455 #define REVIVE       67 /* a FS: revivir proceso dormido */
03456 #define TASKJIEPLY  68 /* a FS: código de respuesta de tarea tty */
03457
03458 /* Posix signal handiing. */
03459 #define SIGACTION    71
03460 #define SIGSUSPEND  72
03461 #define SIQPENDING   73
03462 #define SIGPROGMASK  74
03463 #define SIGRETURN    75
03464
03465 #define REBOOT       76

```

+++++

include/minix/com.h

+++++

```

03500 /* llamadas al sistema, pero se procesan como tales. */
03501 #define SEND          1 /* cód. func. para enviar mensajes */
03502 #define RECEIVE      2 /* cód. func. para recibir mensajes */
03503 #define BOTH          3 /* cód. func. para SEND + RECEIVE */
03504 #define ANY (NR_PROCS+100) /* receive (ANY, buf) acepta de cualq. origen */
03505
03506 /* Números de tarea, códigos de función y códigos de respuesta. */
03507
03508 /* Los valores de varios núms. de tarea dependen de si están habilitadas
03509 * ellas u otras tareas. Se definen como (PREVIOUS_TASK - ENABLE_TASK) en general,
03510 * ENABLE_TASK es 0 o 1, así que una tarea recibe un núm. nuevo o bien el
03511 * mismo de la tarea anterior y ya no se usa más.
03512 * La tarea TTY siempre debe tener el núm. más negativo para inicializarse
03513 * primero. Muchos códigos de función TTY se comparten con otras
03514 * tareas.
03515 */
03516
03517 #define TTY          (DL_ETH - 1)
03518 /* clase de E/S de terminal */
03519 #define CANCEL      0 /* req general para que una tarea cancele */

```

```

03520 # define HARD_INT 2 /* código fcn p/todas interrup. de hard . */
03521 # define DEV_READ 3 /* código fcn p/leer de tty */
03522 # define DEV_WRITE 4 /* código fcn p/escribir en tty */
03523 # define DEV_IOCTL 5 /* código fcn p/iocti */
03524 # define DEV_OPEN 6 /* código fcn p/abrir tty */
03525 # define DEV_GLOSE 7 /* código fcn p/cerrar tty */
03526 # define SCATTERED_IO 8 /* código fcn p/múlt. lect./escrit. */
03527 # define TTY_SETPGRP 9 /* código fcn p/ setpgroup */
03528 # define TTY_EXIT 10 /* un jefe de grupo de proc. salió */
03529 # define OPTIONAL_IO 16 /* modificador- de códigos DEV*en vector */
03530 # define SUSPEND -998 /* se usa en interrup. si tty no tiene datos */
03531
03532 #define DLETH (CDROM - ENABLENETWORKING)
03533 /* tarea en red */
03534
03535 /* Tipo de mensaje para solicitudes de capa de enlace de datos. */
03536 # define DL_WRITE 3
03537 ff define DL_WRITEV 4
03538 # define DL_READ 5
03539 # define DL_READV 6
03540 # define DL_INIT 7
03541 # define DL_STOP 8
03542 # define DL_GETSTAT 9
03543
03544 I* Tipo de mensaje para respuestas de capa de enlace de datos. */
03545 # define DL_INIT_REPLY 20
03546 # define DL_TASK_REPLY 21
03547
03548 # define DL_PORT m2_il
03549 # define DL_PROC m2_i2
03550 # define DL_COUNT m2_i3
03551 # define DL_MODE m2_11
03552 # define DL_CLCK m2_12
03553 # define DL_ADDR m2_p1
03554 # define DL_STAT m_2,11
03555
03556 /* Bits de campo 'DL STAT' de respuestas de DL. */
03557 # define DL_PACK_SEND 0x01
03558 # define DL_PACK_RECV 0x02
03559 # define DL_READIP 0x04
03560
03561 /* Bits de campo 'DL MODE' de solicitudes de DL. */
03562 # define DL_NOMODE 0x0
03563 # define DL_PROMISC_REQ 0x2
03564 # define DL_MULTI_REQ 0x4
03565 # define DL_BROADREQ 0x8
03566
03567 # define NW_OPEN DEV_OPEN
03568 # define NW_CLOSE DEV_GLOSE
03569 # define NW_READ DEV_AREAD
03570 # define NW_WRITE DEV_WRITE
03571 # define NW_IOCTL DEV_IOCTL
03572 # define NW_CANCEL CANCEL
03573
03574 #define CDROM (AUDIO - ENABLECDROM)
03575 /* tarea de dispositivo CD-ROM */
03576
03577 #define AUDIO (MIXER- ENABLE AUDIO)
03578 #define MIXER (SCSI - ENABLEAUDIO)
03579 /* tareas de disp. de audio y mezclador */

```

```

03580
03581 #define SCSI          (WINCHESTER - ENABLE_SCSI)
03582                      /* tarea de dispositivo SCSI */
03583
03584 #define WINCHESTER    (SYN_ALARM_TASK - ENABLE_WINI)
03585                      /* clase de disco duro winchester */
03586
03587 #define SYN_ALARM_TASK -8 /* tarea p/enviar mensajes CLOCK_INT */
03588
03589 #define IDLE          -7 /* tarea p/ejecutar si no hay qué ejecutar */
03590
03591 #define PRINTER      -6 /* clase de E/S de impresora */
03592
03593 #define FLOPPY       -5 /* clase de disco flexible */
03594
03595 #define MEM          -4 /* clase /dev/ram, /dev/(k)mem y /dev/null */
03596 #   define NULL_MAJOR 1 /* disp. princ. para /dev/null */
03597 #   define RAM_DEV    0 /* disp. secund. para /dev/ram */
03598 #   define MEM_DEV    1 /* disp. secund. para /dev/mem */
03599 #   define KMEM_DEV   2 /* disp. secund. para /dev/kmem */
03600 #   define NULL_DEV   3 /* disp. secund. para /dev/null */
03601
03602 #define CLOCK        -3 /* clase de reloj */
03603 #   define SET_ALARM  1 /* código fcn a CLOCK, fijar alarma */
03604 #   define GET_TIME   3 /* código fcn a CLOCK, obt- tiempo real */
03605 #   define SET_TIME   4 /* código fcn a CLOCK, fij. tiempo real */
03606 #   define GET_UPTIME 5 /* código fcn a CLOCK, obt. tiempo activo */
03607 #   define SET_SYNC_AL 6 /* código fcn a CLOCK, fijar alarma que */
03608                          /* se vence con un envío */
03609 #   define REAL_TIME  1 /* resp. de CLOCK: aquí está tiempo real */
03610 #   define CLOCK_INT  HARD_INT
03611                          /* este código sólo será enviado por */
03612                          /* SYN_ALARM_TASK a una tarea que */
03613                          /* solicitó una alarma síncrona */
03614
03615 #define SYSTASK      -2 /* funciones internas */
03616 #   define SYS_XIT    1 /* código fcn para sys_xit(parent, proc) */
03617 #   define SYS_GETSP  2 /* código fcn para sys_sp(proc, &new_sp) */
03618 #   define SYS_OLDSIG 3 /* código fcn para sys_oldsig(proc, sig) */
03619 #   define SYS_FORK   4 /* código fcn para sys_fork(parent, child) */
03620 #   define SVS_NEWMAP 5 /* código fcn para sys_newmap(procno, map_ptr) */
03621 #   define SYS_COPY   6 /* código fcn para sys_copy(ptr) */
03622 #   define SYS_EXEC   7 /* código fcn para sys_exec(procno, new_sp) */
03623 #   define SYS_TIMES  8 /* código fcn para sys_times(procno, bufptr) */
03624 #   define SYS_ABORT  9 /* código fcn para sys_abort() */
03625 #   define SYS_FRESH 10 /* código fcn para sys_fresh() (sólo Atari) */
03626 #   define SYS_KILL   11 /* código fcn para sys_kill(proc, sig) */
03627 #   define SYS_GBOOT 12 /* código fcn para sys_gboot(procno, bootptr) */
03628 #   define SYS_UMAP   13 /* código fcn para sys_umap(procno, etc) */
03629 #   define SYS_MEM    14 /* código fcn para sys_mem() */
03630 #   define SYS_TRACE  15 /* código fcn para sys_trace(req, pid, addr, data) */
03631 #   define SYS_VCOPY  16 /* código fcn para sys_vcopy(src_proc, dest_proc, vcopy_s,
03632                          vcopy_ptr) */
03633 #   define SYS_SENDSIG 17 /* código fcn para sys_sendsig(&aigmsg) */
03634 #   define SYS_SIGRETURN 18 /* código fcn para sys_sigreturn(&sigmsg) */
03635 #   define SYS_ENDSIG  19 /* código fcn para sys_endsig(procno) */
03636 #   define SYS_GETMAP  20 /* código fcn para sys_getmap(procno, map_ptr) */
03637
03638 #define HARDWARE     -1 /* origen en mensajes gen. por interr. */
03639
565

```

```

03640 /* Nombres de campos de mensaje para mensajes a la tarea CLOCK. */
03641 #define DELTA TICKS m6_i1 /* intervalo de alarma en tics */
03642 #define FUNC TO CALL m6_f1 /* apunta a la función por llamar */
03643 #define NEW TIME m6_i1 /* valor p/poner reloj (SETTIME) */
03644 #define CLOCKPROCNR m6_i1 /* ¿qué pr-oc (o tarea) quiere alarma? */
03645 #define SECONDSLEFT m6_i1 /* ¿cuántos segundos quedaban? */
03646
03647 /* Campos de mensaje para mensajes a tareas por bloques y caracteres. */
03648 #define DEVICE m2_i1 /* dispositivo princ./secund. */
03649 #define PROC NR m2_i2 /* ¿Qué proc. quiere E/S? */
03650 #define COUNT m2_i3 /* núm. bytes por transferir */
03651 #define REQUEST m2_i3 /* código de solicitud iocti */
03652 #define POSITION m2_i1 /* distancia en el archivo */
03653 #define ADDRESS m2_p1 /* dirección de buffer central */
03654
03655 /* Nombres de campos de mensaje para mensajes a la tarea TTY. */
03656 #define TTY LINE DEVICE /* parám. de mensaje: línea de terminal */
03657 #define TTYREQUEST COUNT /* parám. de mensaje: cód. solicitud iocti */
03658 #define TTYSPEK POSITION /* parám. de mensaje: rapidez iocti, borrado */
03659 #define TTY FLAQS m2_i2 /* parám. de mensaje: modo tty iocti */
03660 #define TTYPRG m2_i3 /* parám. de mensaje: gpo. de procesos */
03661
03662 /* Campos de mensaje para respuest a de situac. QIC 02 de controlador de cinta */
03663 #define TAPE STAT0 m2_i1
03664 #define TAPESTAT1 m2_i2
03665
03666 /* Campos de mensaje empleados en mensajes de respuesta de tareas. */
03667 #define REP PROC NR m2_i1 /* # de proc. por los que se hizo E/S */
03668 #define REPSTATUS m2_i2 /* bytes transferidos o núm. de error */
03669
03670 /* Nombres de campos para mensaje de copia a SYSTASK. */
03671 #define SRCSPACE m5_c1 /* espacio T o D (la pila es D) */
03672 #define SRC PROC NR m5_i1 /* proceso del cual copiar */
03673 #define SRC BUFFER m5_i1 /* dir. virtual de donde vienen datos */
03674 #define DST SPACE m5_c2 /* espacio T o D (la pila es D) */
03675 #define DST PROC NR m5_i2 /* proceso al cual copiar */
03676 #define DSTBUFFER m5_i2 /* dir. virtual adonde van los datos */
03677 #define COPY_BYTES m5_i3 /* núm. de bytes por copiar */
03678
03679 /* Nombres de campos para contabilidad, SYSTASK y diversos. */
03680 #define USER TIME m4_i1 /* tiempo de usuario consumido por proc. */
03681 #define SYSTEM TIME m4_i2 /* tiempo de sistema consumido por proc. */
03682 #define CHILD UTIME m4_i3 /* tiempo usuario consum. por hijos del proc. */
03683 #define CHILD STIME m4_i4 /* tiempo sist. consum. por hijos del proc. */
03684 #define BOOTTICKS m4_i5 /* tics de reloj desde el arranque */
03685
03686 #define PROC1 m1_i1 /* indica un proceso */
03687 #define PROC2 m1_i2 /* indica un proceso */
03688 #define PID m1_i3 /* id de proceso pasado de MM a kernel */
03689 #define STACK_PTR m1_p1 /* para apunt. a pila en sysexec, sysgetsp */
03690 #define PR m6_i1 /* núm. de proc. para syssig */
03691 #define SIGNUU m6_i2 /* núm. de señal para syssig */
03692 #define FUNC m6_f1 /* apunt. a función para syssig */
03693 #define MEMPTR m1_p1 /* dónde está mapa de memoria para sysnewmap */
03694 #define NAMEPTR m1_p2 /* dónde está nombre de prog. para dmp */
03695 #define IP PTR m1_p3 /* valor inicial de ip después de exec */
03696 #define SIG PROC m2_i1 /* núm. de proc. para inform */
03697 #define SIG MAP m2_i1 /* lo usa kernel p/pasar mapa de bits de señal */
03698 #define SIGMSGPTR m1_i1 /* apunt. a info. p/crear pila p/atrap. señales */
03699 #define SIQCTXTPTR m1_p1 /* apunt. a info. p/restaurar contexto señal */

```

```

+++++
                                     include/minix/boot.h
+++++

```

```

03700 /* boot.h */
03701
03702 #ifndef BOOT H
03703 #define BOOTH
03704
03705 /* Redefinir ios dispositivos raíz e imagen de raiz como variables
03706 " Esto reduce las diferencias pero puede causar confusión futura.
03707 */
03708 #define ROOT^DEV (bootpar'ameters .bprootdev)
03709 #define IMAGEDEV (bootparanieters .bpramimagedev)
03710
03711 /* Números de dispositivo de discos RAM, flexible y duro.
03712 * h/com.h define RAM/DEV pero sólo como número secundario.
03713 */
03714 #define DEV FD0 0x200
03715 #define DEVHDD0 0x300
03716 #define DEVRAM 0x100
03717 #define DEVSCSI 0x700 /* sólo Atari TT */
03718
03719 /* Estructura para contener parámetros de arranque. */
03720 struct bparam s
03721 {
03722 devt bprootdev;
03723 dev^t bprainiagedev;
03724 unsigned short bprainsize;
03725 unsigned snort bp processor;
03726 };
03727
03728 extern struct bpar-am s boot parameters;
03729 #endif /* BOOTH */

```

```

+++++
                                     include/minix/keymap.h
+++++

```

```

03800 /* keymap .h - define mapa Autor: Marcus Hampel
03801 */
03802 #ifndef SYS KEYMAP H
03803 #define SYS KEYMAPH
03804
03805 #define C(c) ((c) & 0X1F) /* Mapear a código de control */
03806 #define A(c) ((c) | 0X80) /* Activar- bit 8 (ALT) */
03807 #define CA(C) A(C(c)) /* Control-Alt */
03808 #define L(c) ((c) : HASCAPS) /* Agregar atrib. "Bloq. Mayús. activo" */
03809
03810 #define EXT 0X0100 /* Teclas de fuñe. Normales */
03811 #define CTRL 0X0200 /* Tecla control */
03812 #define SHIFT 0X0400 /* Tecla shift */
03813 #define ALT 0x0800 /* Tecla alt */
03814 #define EXTKEY 0X1000 /* código de tecla extendido */

```

```

83815 #define HASGAPS 0x8000 /* Bloq. Mayús- activo */
83816
83817 /* Subteclado numérico */
83818 #define HOME (0x01 + EXT)
83819 #define END (0x02 + EXT)
83820 #define UP (0x03 + EXT)
83821 #define DOWN (0x04 + EXT)
83822 #define LEFT (0x05 + EXT)
83823 #define RIGHT (0x06 + EXT)
83824 #define PGUP (0x07 + EXT)
83825 #define PGDN (0x08 + EXT)
83826 #define MID (0x09 + EXT)
83827 #define NMIN (0x0A + EXT)
83828 #define PLUS (0x0B + EXT)
83829 #define INSRT (0x0G + EXT)
83830
83831 /* Alt + Subteclado numérico
83832 #define AHOME (0x01 + ALT)
83833 #define AEND (0x02 + ALT)
83834 #define AUP (0x03 + ALT)
83835 #define ADOWN (0x04 + ALT)
83836 #define ALEFT (0x05 + ALT)
83837 #define ARIGHT (0x06 + ALT)
83838 #define APGUP (0x07 + ALT)
83839 #define APGDN (0x08 + ALT)
83840 #define AMID (0x09 + ALT)
83841 #define ANMIN (0x0A + ALT)
83842 #define AFLÚS (0x0B + ALT)
83843 #define AINSRT (0x0C + ALT)
83844
83845 /* Ctri + Subteclado numérico
83846 #define CHOME (0x01 + CTRL)
83847 #define CEND (0x02 + GTRL)
83848 #define CUP (0x03 + CTRL)
83849 #define CDOWN (0x04 + CTRL)
83850 #define CLEFT (0x05 + CTRL)
83851 #define CRIGHT (0x06 + CTRL)
83852 #define CPGUP (0x07 + CTRL)
83853 #define CPGDN (0x08 + CTRL)
83854 #define CMID (0x09 + CTRL)
83855 #define CNMIN (0x0A + CTRL)
83856 #define CPLUS (0x0B + CTRL)
83857 #define CINSRT (0x0C + CTRL)
83858
83859 /* Teclas locales */
83860 #define CALOCK (0x0D + EXT) /* bloq. mayús. */
83861 #define NLOCK (Bx0E + EXT) /* bloq. núm. */
83862 #define SLOCK (0x0F + EXT) /* bloq. despl. */
83863
83864 /* Teclas de función */
83865 #define F1 (0x10 + EXT)
83866 #define F2 (0x11 + EXT)
83867 #define F3 (0x12 + EXT)
83868 #define F4 (0x13 + EXT)
83869 #define F5 (0x14 + EXT)
83870 #define F6 (0x15 + EXT)
83871 #de-fine F7 (0x16 + EXT)
83872 #define F8 (0x17 + EXT)
83873 #define F9 (0x18 + EXT)
83874 #define F10 (0x19 + EXÍ)

```

```

03875 #define F11      (0X1A + EXT)
03876 #define F12      (0X1B + EXT)
03877
03878 /* Alt+Fn */
03879 #define AF1         (0x10 + ALT)
03880 #define AF2         (0x11 + ALT)
03881 #define AF3         (0X12 + ALT)
03882 #define AF4         (0x13 + ALT)
03883 #define AF5         (0x14 + ALT)
03884 #define AF6         (0x15 + ALT)
03885 #define AF7         (0X16 + ALT)
03886 #define AF8         (0X17 + ALT)
03887 #define AF9         (0x18 + ALT)
03888 #define AF10        (0x19 + ALT)
03889 #define AF11        (0X1A + ALT)
03890 #define AF12        (0X18 + ALT)
03891
03892 /* Ctrl+Fn */
03893 #define CF1         (0x10 + CTRL)
03894 #define CF2         (0X11 + CTRL)
03895 #define CF3         (0X12 + CTRL)
03896 #define GF4         (0x13 + CTRL)
03897 #define CF5         (0x14 + CTRL)
03898 #define CF6         (0x15 + CTRL)
03899 #define CF7         (0x16 + CTRL)
03900 #define CF8         (0X17 + CTRL)
03901 #define GF9         (0x18 + CTRL)
03902 #define CF10        (0x19 + CTRL)
03903 #define CF11        (0X1A + CTRL)
03904 #define CF12        (0X1B + CTRL)
03905
03906 /* Shift+Fn */
03907 #define SF1         (0x10 + SHIFT)
03908 #define SF2         (0x11 + SHIFT)
03909 #define SF3         (0x12 + SHIFT)
03910 #define SF4         (0x13 + SHIFT)
03911 #define SF5         (0x14 + SHIFT)
03912 #define SF6         (0x15 + SHIFT)
03913 #define SF7         (0x16 + SHIFT)
03914 #define SP8         (0x17 + SHIFT)
03915 #define SF9         (0x18 + SHIFT)
03916 #define SF10        (0x19 + SHIFT)
03917 #define SF11        (0X1A + SHIFT)»
03918 #define SF12        (0x1B + SHIFT)
03919
03920 /* Alt+Shift+Fn */
03921 #define ASF1        (0X10 + ALT + SHIFT)
03922 #define ASF2        (0x11 + ALT + SHIFT)
03923 #define ASF3        (0x12 + ALT + SHIFT)
03924 #define ASF4        (0x13 + ALT + SHIFT)
03925 #define ASF5        (0x14 + ALT + SHIFT)
03926 #define ASF6        (0x15 + ALT + SHIFT)
03927 #define ASF7        (0x16 + ALT + SHIFT)
03928 #define ASF8        (0x17 + ALT + SHIFT)
03929 #define ASF9        (0x18 + ALT + SHIFT)
03930 #define ASF10       (0x19 + ALT + SHIFT)
03931 #define ASF11       (0X1A + ALT + SHIFT)
03932 #define ASF12       (0x1B + ALT + SHIFT)
03933
03934 #define MAPCOLS      6          /* Núm. de cols. en mapa de teclas */

```

```

03935 #define NR_SCAN_CODES 0x80 /* Núm. códigos detec. (filas en mapa) */
03936
03937 typedef unsigned short keymap_tt[NR_SCAN_CODES * MAP_COLS];
03938
03939 #define KEYJ1AGIC "KMAZ" /* número mágico del arch. de mapa de teclas */
03940
03941 #endif /* _SYS_KEYMAP_H */

```

```

+++++
include/minix/partition.h
+++++

```

```

04000 /* minix/partition.h Autor: Kees J. Bot
04001 * Dic. 7. 1995
04002 * Lugar de una partición en disco y la geometría de disco,
04003 * para usarse con los iocti DIOCGETP y DIOCSETP,
04004 */
04005 #ifndef _MINIX_PARTITION_H
04006 #define _MINIX_PARTITION_H
04007
04008 struct partition {
04009     u32_t base; /* distancia en bytes a inicio de partición */
04010     u32_t size; /* número de bytes en la partición */
04011     unsigned cylinders; /* geometría de disco */
04012     unsigned heads;
04013     unsigned sectors;
04014 };
04015 #endif /* MINIX_PARTITION_H */

```

```

+++++
include/ibm/partition.h
+++++

```

```

04100 /* Descripción de una entrada de la tabla de particiones */
04101 #endif /* _PARTITION_H */
04102 #define _PARTITION_H /*
04103
04104 struct partition {
04105     unsigned char bootind; /* indicador de arranque Ó/ACTIVE_FLAG */
04106     unsigned char starthead; /* valor de cabeza p/primer sector */
04107     unsigned char startsec; /* valor sector + bits cil. p/ier. sector */
04108     unsigned char startcyl; /* valor de pista p/ier. sector */
04109     unsigned char sysind; /* indicador de sistema */
04110     unsigned char lasthead; /* valor de cabeza p/últ. sector */
04111     unsigned char lastsec; /* val. sector + bits cil. p/últ. sector */
04112     unsigned char lastcyl; /* valor de pista p/últ. sector */
04113     unsigned long lowsec; /* 1er. sector lógico */
04114     unsigned long size; /* tamaño de partición en sectores */
04115 };
04116
04117 #define ACTIVE_FLAG 0x80 /* valor de activo en campo bootind (hd0) */
04118 #define NR_PARTITIONS 4 /* # entradas en tabla de partic. */
04119 #define PART_TABLE_OFF 0x1BE /* dist. de tabla de partic. en sec. arranque */

```



```

04120 /* Tipos de particiones. */
04121 #define MINIX_PART 0x81 /* Tipo de partición Minix */
04122 #define NO_PART 0x00 /* entrada no utilizada */
04123 #define OLD_MINIX_PART 0x80 /* creada antes de 1.4b, obsoleta */
04124 #define EXT_PART 0x05 /* partición extendida */
04125
04126
04127 #endif /* PARTITION H */

```

```

+++++
src/kernel/kernel.h
+++++

```

```

04200 /* Ésta es la cabecera maestra del kernel; incluye algunos otros archivos
04201 * y define las principales constantes.
04202 */
04203 #define POSIX_SOURCE 1 /* decir a cabeceras que incluyan cosas POSIX */
04204 #define MINIX 1 /* decir a cabeceras que incluyan cosas UINIX */
04205 #define SYSTEM 1 /* decir a cabeceras que éste es el kernel */
04206
04207 /* Lo siguiente es tan básico que todo archivo *.c lo obtiene automáticamente. */

04208 #include <minix/con-Fig.h> /* DEBE ser primero */
04209 #include <ansi.h> /* DEBE ser segundo */
04210 #include <sys/types.h>
04211 #include <minix/const.h>
04212 #include <minix/type.h>
04213 #include <minix/syslib.h>
04214
04215 #include <string.h>
04216 #include <limits.h>
04217 #include <err'no.h>
04218
04219 #include "const.h"
04220 #include "type.h"
04221 #include "proto.h"
04222 #include "glo.h"

```

```

+++++
src/kernel/const.h
+++++

```

```

04300 /* Constantes generales empleadas por el kernel. */
04301
04302 #if (CHIP == INTEL)
04303
04304 #define K_STACK_BYTES 1024 /* # bytes para la pila del kernel */
04305
04306 #define INIT_PSW 0x0200 /* psw inicial */
04307 #define IHIT_TASK_PSW 0x1200 /* psw inicial para tareas (con IOPL 1) */
04308 #define TRACEBIT 0x100 /* OR esto con psw en proc[] p/rastreo */
04309 #define SETPSW(rp, new) /* sólo permite activar ciertos bits */ \
04310 ((rp)->p_reg.psw = (rp)->p_reg.psw & -0xCD5 | (new) & 0xCD5)
04311
04312 /* Sp inicial para mm, fs e init.
04313 * 2 bytes para salto corto
04314 * 2 bytes no utilizados

```

```
04315 *      3 palabras para init_org[] , sólo lo usa fs
04316 *      3 palabras p/trampa de depurador en modo real (de hecho necesita 1 más)
04317 *      3 palabras para guardar y reiniciar temporales
04318 *      3 palabras para interrupción
04319 * No dejar margen, para eliminar errores pronto.
04320 */
04321 #define INII_SP (2 + 2 + 3 * 2 + 3 * 2 + 3 * 2 + 3 * 2)
04322
04323 #define HCLICK_SHIFT      4 /* 10g2 de HCLICK_SIZE */
04324 #define HCLICK_SIZE      16 /* magia de conversión de seg. de hardware */
04325 #if CLICK_SIZE >= HCLICK_SIZE
04326 #define click_to_hclick(n) ((n) << (CLICK_SHIFT -HCLICK_SHIFT))
04327 #else
04328 #define click_to_hclick(n) ((n) >> (HCLICK_SHIFT -CLICK_SHIFT))
04329 #endif
04330 #define hclick_to_physb(n) ((phys_bytes) (n) << HCLICK_SHIFT)
04331 #define physb_to_hclick(n) ((n) >> HCLICK_SHIFT)
04332
04333 /* Vectores de interrupción definidos/reservados por el procesador. */
04334 #define DIVIDE_VECTOR      0 /* error de división */
04335 #define DEBUG_VECTOR      1 /* paso a paso (rastreo) */
04336 #define NMI_VECTOR        2 /* interrupción no enmascarable */
04337 #define BREAKPOINT_VECTOR 3 /* punto de corte en software */
04338 #define OVERFLOW_VECTOR   4 /* de INTO */
04339
04340 /* Vector de llamada al sistema fijo. */
04341 #define SYS_VECTOR        32 /* llamadas se hacen con int SYSVEC */
04342 #define SYS386_VECTOR     33 /* excepto llamadas 386 usan esto */
04343 #define LEVEL0_VECTOR     34 /* para ejecutar una función en nivel 0 */
04344
04345 /* Bases irq apropiadas para interrupciones por hardware. Reprogramar 8259(S)
04346 * con predeterminados de PC BIOS ya que BIOS no respeta todos los vectores
04347 * reservados por el procesador (0 a 31).
04348 */
04349 #define BIOS_IRO0_VEC 0x08 /* base de vectores IRO0-7 empleada por BIOS */
04350 #define BIOS_IRO8_VEC 0x70 /* base de vectores IRO8-15 empleada por BIOS */
04351 #define IRO0_VECTOR      0x28 /* +/- arbitrario pero > SYS_VECTOR */
04352 #define IRQ8_VECTOR      0x30 /* juntos por sencillez */
04353
04354 /* Números de interrupciones de hardware. */
04355 #define NR_IRQ_VECTORS   16
04356 #define CLOCK_IRQ        0
04357 #define KEYBOARD_IRQ     1
04358 #define CASCADE_IRQ      2 /* habil. cascada para 20. controlador AT */
04359 #define ETHER_IRQ        3 /* vector interrup. ethernet predeterm. */
04360 #define SECONDARY_IRQ    3 /* vector interrup. RS232 p/puerto 2 */
04361 #define RS232_IRQ        4 /* vector interrup. RS232 p/puerto 1 */
04362 #define XT_WINI_IRQ      5 /* xt winchester */
04363 #define FLOPPY_IRQ       6 /* disco flexible */
04364 #define PRINTER_IRQ      7
04365 #define AT_WINI_IRQ      14 /* at winchester */
04366
04367 /* Número de interrupción a vector de hardware. */
04368 #define BIOS_VECTOR(irq) \
04369 ((irq) < 8 ? BIOS_IRO0_VEC : BIOS_IRO8_VEC) + ((irq) & 0x07)
04370 #define VECTOR(irq) \
04371 ((irq) < 8 ? IRQ0_VECTOR : IRQ8_VECTOR) + ((irq) & 0x07)
04372
04373 /* Vectores de parámetros de disco duro BIOS. */
04374 #define WINI_0_PARM_VEC 0x41
```

```

04375 #define WINI_1_PARM_VEC 0x46
04376
04377 /* Puertos del controlador de interrupciones 8259A. */
04378 #define INT_CTL          0x20    /* puerto de E/S p/contrOI. interrup. */
04379 #define INT_CTLMASK     0x21    /* activar bit s en este puerto inhabi. ints. */
04380 #define INT2_CTL        0xA0    /* puerto E/S p/20. control. interrup. */
04381 #define INT2_CTLMASK   0xA1    /* activar bits en este puerto inhabi. ints. */
04382
04383 /* Números mágicos para el controlador de interrupciones. */
04384 #define ENABLE          0x20    /* código p/rehabilitar después de una int. */
04385
04386 /* Tamaños de las tablas de memoria. */
04387 #define NR_MEMS         3        /* número de trozos de memoria */
04388
04389 /* Puertos diversos. */
04390 #define PCR              0x65    /* Registro de Control Plano */
04391 #define PORT_B          0x61    /* puerto E/S p/puerto B 8255 (tecl. alarma. ..) */
04392 #define TIMER0          0x40    /* puerto E/S p/canal 0 temporizador */
04393 #define TIMER2          0x42    /* puerto E/S p/canal 2 temporizador */
04394 #define TIMER_MODE     0x43    /* puerto E/S p/control de modo temporiz. */
04395
04396 #endif /* (CHIP == INTEL) */
04397
04398 #if (CHIP == M68000)
04399
04400 #define K_STACK_BYTES 1024 /* núm. bytes para pila del kernel */
04401
04402 /* Tamaños de tablas de memoria. */
04403 #define NR_MEMS         2        /* número de trozos de memoria */
04404
04405 /* p_reg contiene: d0-d7, a0-a6, en ese orden. */
04406 #define NR_REGS         15      /* # regs. gales. en cada ranura proc. */
04407
04408 #define TRACEBIT        0x8000   /* O esto con psw en proc[] p/rastreo */
04409 #define SETPSW(rp, new)    /* permite activar sólo ciertos bits */
04410          ((rp) ->p_reg.psw = (rp) ->p_reg.psw & -0xFF : (new) & 0xFF)
04411
04412 #define MEM_BYTES 0xffffffff /* tamaño de memoria para /dev/mem */
04413
04414 #ifndef --ACK-
04415 #define FSTRUCOPY
04416 #endif
04417
04418 #endif /* (CHIP == M68000) */
04419
04420 /* Lo siguiente pertenece a colas de planificación. */
04421 #define TASK_Q          0        /* tareas listas planif. vía cola 0 */
04422 #define SERVÉR_Q       1        /* servidores listos planif. vía cola 1 */
04423 #define USER_Q-       2        /* usuarios listos planif. vía cola 2 */
04424
04425 #if (MACHINE == ATARI)
04426 #define SHADOW_Q       3        /* procesos ejecutables pero en sombra */
04427 #define NQ              4        /* núm. colas de planificación */
04428 #else
04429 #define NQ              3        /* núm. colas de planificación */
04430 #endif
04431
04432 /* Valores devueltos por Env_parse(). */
04433 #define EP_UNSET        0        /* variable no establecida */
04434 #define EP_OFF          1        /* var = off */

```



```

04545 regt  si;      /* orden es p/concordar con pusha/popa */
04546 regt  fp;      /* bp */
04547 regt  st;      /* agujero para otra copia de sp */
04548 regt  bx;      /* */
04549 regt  dx;      /* */
04550 regt  ex;      /* */
04551 regt  retreg; /* ax y anteriores en pila por save */
04552 reat  retadr; /* dir. retorno p/save() en código ensam. */
04553 regt  pe;      /* " últ. cosa en pila por interrupción */
04554 regt  es;      /* */
04555 regt  psw;     /* */
04556 regt  sp;      /* */
04557 reg t  ss;     /* en pila por- CPU durante interrupción */
04558 };
04559
04560 struct segdescs { /* descriptor de segmento p/modo protegido */
04561     U16_t  iimitlow;
04562     ui6_t  baselow;
04563     u8_t   basemiddle;
04564     u8_t   access; /* |P|DL|1|X|E|R|A| */
04565     #if WORDSIZE == 4
04566     u8_t   granularity; /* |G|X|0|AILIMT| */
04567     u8_t   basehigh;
04568     #else
04569     u16_t  reserved;
04570     #endif
04571 };
04572
04573 typedef PROTOTYPE( int ( /*irqhandlert), (int irq) );
04574
04575 #endif /* (CHIP == INTEL) */
04576
04577 #if (CHIP == M68000)
04578 typedef PROTOTYPE( VOid (*dmaintt), (void) );
04579
04580 typedef u32t regt; /* registro de máquina */
04581
04582 /* El nombre y campos de este struct se escogieron p/compatibilidad con PC. */
04583 struct stackframe s {
04584     reg_t  retreg; /* d0 */
04585     reg_t  d1 ;
04586     reg_t  d2;
04587     reg_t  d3;
04588     reg_t  d4;
04589     reg_t  d5;
04590     reg_t  d6;
04591     reg_t  d7;
04592     reg_t  a0;
04593     reg_t  a1;
04594     reg_t  a2;
04595     reg_t  a3;
04596     reg_t  a4;
04597     reg_t  a5;
04598     reg_t  fp; /* también conocido como a6 */
04599     reg_t  sp; /* también conocido como a7 */
04600     reg_t  pe;
04601     u16_t  pSW;
04602     U16_t  dummy; /* hacer tamaño múltiplo de regt para system.c
04603 };
04604

```

```

04605     struct fsave {
04606     struct cpu_state {
04607         u16_t i_format;
04608         u32_t i_addr;
04609         u16_t i_state[4];
04610     } cpu_state;
04611     struct state_frame {
04612         u8_t frame_type;
04613         u8_t frame_size;
04614         u16_t reserved;
04615         u8_t frame[212];
04616     } state_frame;
04617     struct fpp_model {
04618         u32_t fpcr;
04619         u32_t fpsr;
04620         u32_t fpiar;
04621         struct fpN {
04622             u32_t high;
04623             u32_t low;
04624             u32_t mid;
04625         } fpN[8];
04626     } fpp_model;
04627 };
04628 #endif /* (CHIP == M68000) */
04629
04630 #endif /* TYPE_H */

```

```

+++++
src/kernel/proto.h +++++

```

```

04700 /* Prototipos de funciones. */
04701
04702 #ifndef PROTO_H
04703 #define PROTO_H
04704
04705 /* Declaraciones de struct. */
04706 struct proc;
04707 struct tty;
04708
04709 /* at_wini.c, wini.c */
04710 _PROTOTYPE( void winchester_task, (void) );
04711 _PROTOTYPE( void at_winchester_task, (void) );
04712
04713 /* clock.c */
04714 _PROTOTYPE( void clock_task, (void) );
04715 _PROTOTYPE( void clock_stop, (void) );
04716 _PROTOTYPE( clock_t get_uptime, (void) );
04717 _PROTOTYPE( void syn_alrm_task, (void) );
04718
04719 /* dmp.c */
04720 _PROTOTYPE( void map_dmp, (void) );
04721 _PROTOTYPE( void p_dmp, (void) );
04722 _PROTOTYPE( void reg_dmp, (struct proc *rp) );
04723
04724 /* dp8390.c */

```

```

04725 _PROTOTYPE( void dp8390 task, (void) );
04726 _PROTOTYPE( void dp dump, (void) );
04727 _PROTOTYPE( void dp8390_stop, (void) );
04728
04729 /* floppy.c, stfloppy.c */
04730 _PROTOTYPE( void floppy_task, (void) );
04731 _PROTOTYPE( void floppy_stop, (void) );
04732
04733 /* main.c, stmain.c */
04734 _PROTOTYPE( void main, (void) );
04735 _PROTOTYPE( void panic, (const char *s, int n) );
04736
04737 /* memory.c */
04738 _PROTOTYPE( void mem_task, (void) );
04739
04740 /* misc.c */
04741 _PROTOTYPE( int env parse, (char *env, char *fmt, int field,
04742 --long *param, long min, long max) );
04743
04744 /* printer.c, stprint.c */
04745 _PROTOTYPE( void printer_task, (void) );
04746
04747 /* proc.c */
04748 _PROTOTYPE( void interrupt, (int task) );
04749 _PROTOTYPE( int lock_mini_send, (struct proc *caller_ptr,
04750 int dest, message *m_ptr) );
04751 _PROTOTYPE( void lock_pick_proc, (void) );
04752 _PROTOTYPE( void lock_ready, (struct proc *rp) );
04753 _PROTOTYPE( void lock_sched, (void) );
04754 _PROTOTYPE( void lock_unready, (struct proc *rp) );
04755 _PROTOTYPE( int sys call, (int function, int src_dest, message *m_ptr) );
04756 _PROTOTYPE( void unhold, (void) );
04757
04758 /* rs232.c */
04759 _PROTOTYPE( void rs_init, (struct tty *tp) );
04760
04761 /* system.c */
04762 _PROTOTYPE( void cause_sig, (int proc_nr, int sig_nr) );
04763 _PROTOTYPE( void inform, (void) );
04764 _PROTOTYPE( phys_bytes numap, (int proc_nr, vir_bytes vir_addr,
04765 vir bytes bytes) );
04766 _PROTOTYPE( void sys task, (void) );
04767 _PROTOTYPE( phys bytes umap, (struct proc *rp, int seg, vir_bytes vir_addr,
04768 -vir_bytes bytes) );
04769
04770 /* tty.c */
04771 _PROTOTYPE( void handle events, (struct tty *tp) );
04772 _PROTOTYPE( void sigchar, (struct tty *tp, int sig) );
04773 _PROTOTYPE( void tty task, (void) );
04774 _PROTOTYPE( int in_process, (struct tty *tp, char *buf, int count) );
04775 _PROTOTYPE( void out process, (struct tty *tp, char *bstart, char *bpos,
04776 --char *bend, int *icount, int *ocount) );
04777 _PROTOTYPE( void tty_wakeup, (clock_t now) );
04778 _PROTOTYPE( void tty_reply, (int code, int replyee, int proc_nr,
04779 int status) );
04780 _PROTOTYPE( void tty_devnop, (struct tty *tp) );
04781
04782 /* biblioteca */
04783 _PROTOTYPE( void *memcpy, (void *_s1, const void *_s2, size_t n) );
04784

```

```
04785 #if (CHIP == INTEL)
04786
04787     /* cloc~.c */
04788     _PROTOTYPE( void milli_start, (struct milli_state *msp) ) ;
04789     _PROTOTYPE( unsigned milli_elapsed, (struct milli_state *msp) ) ;
04790     _PROTOTYPE( void milli_delay, (unsigned millisec) ) ;
04791
04792     /* console.c */
04793     _PROTOTYPE( void cons_stop, (void) ) ;
04794     _PROTOTYPE( void putk, (int c) ) ;
04795     _PROTOTYPE( void scr_init, (struct tty *tp) ) ;
04796     _PROTOTYPE( void toggle_scroll, (void) ) ;
04797     _PROTOTYPE( int con_loadfont, (phys_bytes user_phys) ) ;
04798     _PROTOTYPE( void select_console, (int cons_line) ) ;
04799
04800     /* cstart.c */
04801     _PROTOTYPE( void cstart, (U16_t cs, U16_t ds, U16_t mcs, U16_t mds,
04802     _U16_t parmoff, U16_t parmsize) ) ;
04803     _PROTOTYPE( char *k_getenv, (char *name) ) ;
04804
04805     /* exception.c */
04806     _PROTOTYPE( void exception, (unsigned vec_nr) ) ;
04807
04808     /* i8259.c */
04809     _PROTOTYPE( irq_handler_t get_irq_handler, (int irq) ) ;
04810     _PROTOTYPE( void put_irq_handler, (int irq, irq_handler_t handler) ) ;
04811     _PROTOTYPE( void intr_init, (int mine) ) ;
04812
04813     /* keyboard.c */
04814     _PROTOTYPE( void kb_init, (struct tty *tp) ) ;
04815     _PROTOTYPE( int kbd_loadmap, (phys_bytes user_phys) ) ;
04816     _PROTOTYPE( void wreboot, (int how) ) ;
04817
04818     /* klib*.s */
04819     _PROTOTYPE( void bios13, (void) ) ;
04820     _PROTOTYPE( phys_bytes check_mem, (phys_bytes base, phys_bytes size) ) ;
04821     _PROTOTYPE( void cp_mess, (int src, phys_clicks src_clicks, vir_bytes src_offset,
04822     phys_clicks dst_clicks, vir_bytes dst_offset) ) ;
04823     _PROTOTYPE( int in_byte, (port_t port) ) ;
04824     _PROTOTYPE( int in_word, (port_t port) ) ;
04825     _PROTOTYPE( void lock, (void) ) ;
04826     _PROTOTYPE( void unlock, (void) ) ;
04827     _PROTOTYPE( void enable_irq, (unsigned irq) ) ;
04828     _PROTOTYPE( int disable_irq, (unsigned irq) ) ;
04829     _PROTOTYPE( u16_t mem_rdw, (segm_t segm, vir_bytes offset) ) ;
04830     _PROTOTYPE( void out_byte, (port_t port, int value) ) ;
04831     _PROTOTYPE( void out_word, (port_t port, int value) ) ;
04832     _PROTOTYPE( void phys_copy, (phys_bytes source, phys_bytes dest,
04833     phys_bytes count) ) ;
04834     _PROTOTYPE( void port_read, (unsigned port, phys_bytes destination,
04835     unsigned bytcount) ) ;
04836     _PROTOTYPE( void port_read_byte, (unsigned port, phys_bytes destination,
04837     unsigned bytcount) ) ;
04838     _PROTOTYPE( void port_write, (unsigned port, phys_bytes source,
04839     unsigned bytcount) ) ;
04840     _PROTOTYPE( void port_write_byte, (unsigned port, phys_bytes source,
04841     unsigned bytcount) ) ;
04842     _PROTOTYPE( void reset, (void) ) ;
04843     _PROTOTYPE( void vid_vid_copy, (unsigned src, unsigned dst, unsigned count) ) ;
04844     _PROTOTYPE( void mem_vid_copy, (u16_t *src, unsigned dst, unsigned count) ;
```



```

04845 _PROTOTYPE( void leve10, (void (*func) (void)) );
04846 _PROTOTYPE( void monitor, (void) );
04847
04848 /* misc.c */
04849 _PROTOTYPE( void mem_init, (void) );
04850
04851 /* mpx*.s */
04852 _PROTOTYPE( void idle_task, (void) );
04853 _PROTOTYPE( void restart, (void) );
04854
04855 /* Las siguientes nunca se invocan desde C (procs. asm puros). */
04856
04857 /* Manejadores de excep. (modo real o protegido) en orden numérico. */
04858 void _PROTOTYPE( int00, (void) ), _PROTOTYPE( divide_error, (void) );
04859 void _PROTOTYPE( int01, (void) ), _PROTOTYPE( single_step_exception, (void) );
04860 void _PROTOTYPE( int02, (void) ), _PROTOTYPE( nmi, (void) );
04861 void _PROTOTYPE( int03, (void) ), _PROTOTYPE( breakpoint_exception, (void) );
04862 void _PROTOTYPE( int04, (void) ), _PROTOTYPE( overflow, (void) );
04863 void _PROTOTYPE( int05, (void) ), _PROTOTYPE( bounds_check, (void) );
04864 void _PROTOTYPE( int06, (void) ), _PROTOTYPE( inval_opcode, (void) );
04865 void _PROTOTYPE( int07, (void) ), _PROTOTYPE( copr_not_available, (void) );
04866 void _PROTOTYPE( double_fault, (void) );
04867 void _PROTOTYPE( copr_seg_overrun, (void) );
04868 void _PROTOTYPE( inval_tss, (void) );
04869 void _PROTOTYPE( segment_not_present, (void) );
04870 void _PROTOTYPE( stack_exception, (void) );
04871 void _PROTOTYPE( general_protection, (void) );
04872 void _PROTOTYPE( page_fault, (void) );
04873 void _PROTOTYPE( copr_error, (void) );
04874
04875 /* Manejadores de interrupciones de hardware. */
04876 _PROTOTYPE( void hwint00, (void) );
04877 _PROTOTYPE( void hwint01, (void) );
04878 _PROTOTYPE( void hwint02, (void) );
04879 _PROTOTYPE( void hwint03, (void) );
04880 _PROTOTYPE( void hwint04, (void) );
04881 _PROTOTYPE( void hwint05, (void) );
04882 _PROTOTYPE( void hwint06, (void) );
04883 _PROTOTYPE( void hwint07, (void) );
04884 _PROTOTYPE( void hwint08, (void) );
04885 _PROTOTYPE( void hwint09, (void) );
04886 _PROTOTYPE( void hwint10, (void) );
04887 _PROTOTYPE( void hwint11, (void) );
04888 _PROTOTYPE( void hwint12, (void) );
04889 _PROTOTYPE( void hwint13, (void) );
04890 _PROTOTYPE( void hwint14, (void) );
04891 _PROTOTYPE( void hwint15, (void) );
04892
04893 /* Manejadores de interrupciones de software, en orden numérico. */
04894 _PROTOTYPE( void trp, (void) );
04895 _PROTOTYPE( void s_call, (void) ), _PROTOTYPE( p_s_call, (void) );
04896 _PROTOTYPE( void leve10_call, (void) );
04897
04898 /* printer.c */
04899 _PROTOTYPE( void pr_restart, (void) );
04900
04901 /* protect.c */
04902 _PROTOTYPE( void prot_init, (void) );
04903 _PROTOTYPE( void init_codeseg, (struct segdesc_s *segdp, phys_bytes base,
04904 _phys_bytes size, int privilege) );

```

```

04905  _PROTOTYPE( void init_dataseg, (struct segdesc_s *segdp, phys_bytes base,
04906  _phys_bytes size, int privilege) );
04907  _PROTOTYPE( phys_bytes seg2phys, (U16_t seg) );
04908  _PROTOTYPE( void enable_iop, (struct proc *pp) );
04909
04910  /* pty.c */
04911  _PROTOTYPE( void do_pty, (struct tty *tp, message *m_ptr) );
04912  _PROTOTYPE( void pty_init, (struct tty *tp) );
04913
04914  /* system.c */
04915  _PROTOTYPE( void alloc_segments, (struct proc *rp) );
04916
04917  #endif /* (CHIP == INTEL) */
04918
04919  #endif /* PROTO_H */

```

+++++
src/kernel/glo.h
+++++

```

05000  /* Variables globales empleadas en el kernel. */
05001
05002  /* EXTERN se define como extern excepto en table.c. */
05003  #ifdef TA8LE
05004  #undef EXTERN
05005  #define EXTERN
05006  #endif
05007
05008  /* Memoria del kernel. */
05009  EXTERN phys_bytes code_base; /* base del código del kernel */
05010  EXTERN phys_bytes data_base; /* base de datos del kernel */
05011
05012  /* Comunicaciones de interrupciones de bajo nivel. */
05013  EXTERN struct proc *held_head; /* cabeza de cola de ints. detenidas */
05014  EXTERN struct proc *held_tail; /* final de cola de ints. detenidas */
05015  EXTERN unsigned char k_reenter; /* cuenta reingreso kernel (ingreso -1) */
05016
05017  /* Tabla de procesos. Aquí para no tener que incluir tantas veces proc.h. */
05018  EXTERN struct proc *proc_ptr; /* apunto a proc. actual. en ejecución */
05019
05020  /* Señales. */
05021  EXTERN int sig_procs; /* # procs. con p_pending != 0 */
05022
05023  /* Tamaños de memoria. */
05024  EXTERN struct memory mem[NR_MEMS]; /* base y tamaño de trozos de memo */
05025  EXTERN phys_clicks tot_mem_size; /* tamaño total de memo del sistema */
05026
05027  /* Diversas. */
05028  extern u16_t sizes[]; /* tabla llenada por monitor arranque */
05029  extern struct tasktab tasktab[]; /* inic. en table.c, así que extern aquí */
05030  extern char *t_stack[]; /* inic. en table.c, así que extern aquí */
05031  EXTERN unsigned lost_ticks; /* tics contados afuera de tarea de reloj */
05032  EXTERN clock_t tty_timeout; /* tiempo para despertar la tarea TTY */
05033  EXTERN int current; /* consola actualmente visible */
05034

```

```

05035 #if(CHIP == INTEL)
05036
05037 /* Tipo de máquina. */
05038 EXTERN int pc_at; /* interfaz de hardware PC-AT compatible */
05039 EXTERN int ps-mca; /* PS/2 con Micro Channel */
05040 EXTERN unsigned int processor; /* 86, 186, 286, 386, "" */
05041 #if WORD_SIZE == 2
05042 EXTERN int protected_mode; /* no 0 si en modo Intel protegido */
05043 #else
05044 #define protected_mode 1 /* modo 386 implica modo protegido */
05045 #endif
05046
05047 /* Tipos de tarjetas de video. */
05048 EXTERN int ega; /* no 0 si consola es EGA o VGA */
05049 EXTERN int vga; /* no 0 si consola es VGA */
05050
05051 /* Tamaños de memoria. */
05052 EXTERN unsigned ext_memsize; /* inic. por cód. arranque ensamblador */
05053 EXTERN unsigned low_memsize;
05054
05055 /* Diversas. */
05056 EXTERN irq_handler_t irq_table[NR_IRQ_VECTORS];
05057 EXTERN int irq_use; /* mapa bits de todos irq en uso */
05058 EXTERN reg_t mon_ss, mon_sp; /* pila del monitor */
05059 EXTERN int mon_return; /* true si se puede regresar a monitor */
05060 EXTERN phys_bytes reboot_code; /* programa p/monitor de arranque */
05061
05062 /* Variables inicializadas en otro lado son sólo extern aquí. */
05063 extern struct segdesc_s gdt[]; /* tabla descriptores globales p/modo prot.*/
05064
05065 EXTERN _PROTOTYPE( void (*level0_func) , (void) );
05066 #endif /* (CHIP == INTEL) */
05067
05068 #if(CHIP == M68000)
05069 /* Variables inicializadas en otro lado son sólo extern aquí. */
05070 extern int keypad; /* Bandera p/modo subteclado num. */
05071 extern int app_mode; /* Bandera p/modo aplico tecla flecha */
05072 extern int STdebKey; /* no 0 si se detecta ctl-alt-Fx */
05073 extern struct tty *cur_cons; /* cons. virtual ahora exhibida */
05074 extern unsigned char font8[]; /* tabla fuentes 8 pix. ancho (inic.) */
05075 extern unsigned char font12[]; /* tabla fuentes 12 pix. ancho (inic.) */
05076 extern unsigned char font16[]; /* tabla fuentes 16 pix. ancho (inic.) */
05077 extern unsigned short resolution; /* def. pantalla; ST_RES_LOW. .TT_RES_HIGH */
05078 #endif

```

```

05110     struct proc {
05111         struct stackframe_s p_reg;           /* regs. de proc. guardados en marco pila */
05112
05113     #if (CHIP == INTEL)
05114         reg_t p_ldt_sel;                     /* se lector en gdt da base y límite ldt */
05115         struct segdesc_s p_ldt[2];         /* descriptors locales p/cód. y datos */
05116                                           /* 2 es LDT_SIZE -evita incluir protect.h */
05117     #endif /* (CHIP == INTEL) */
05118
05119         reg_t *p_stguard;                   /* palabra guardia de la pila */
05120
05121         int p_nr;                           /* # de este proc. (p/acceso rápido) */
05122
05123         int p_int_blocked;                 /* no 0 si mens. int. bloqueado por tarea ocup.*/
05124         int p_int_held;                   /* no 0 si mens. int. detenido por llamada ocup.*/
05125         struct proc *p_nextheld;          /* sigte. en cadena de proc. int. detenidos */
05126
05127         int p_flags;                       /* P_SLOT_FREE, SENDING, RECEIVING, etc. */
05128         struct mem_map p_map[NR_SEGS];    /* mapa de memoria */
05129         pid_t p_pid;                       /* id de proceso pasado desde MM */
05130
05131         clock_t user_time;                 /* tiempo de usuario en tics */
05132         clock_t sys_time;                 /* tiempo de sistema en tics */
05133         clock_t child_utime;             /* tiempo de usuario acum. de hijos */
05134         clock_t child_stime;             /* tiempo de sist. acumulado de hijos */
05135         clock_t p_alarm;                 /* tiempo de sigte. alarma en tics, 00 */
05136
05137         struct proc *p_callerq;           /* cabeza de lista de procs. quieren enviar */
05138         struct proc *p_sendlink;         /* vínc. a sigte. proc. quiere enviar */
05139         message *p_messbuf;             /* apuntador a buffer de mensaje */
05140         int p_getfrom;                   /* ¿de quién quiere recibir el proc.? */
05141         int p_sendto;
05142
05143         struct proc *p_nextready;        /* apunto a sigte. proceso listo */
05144         sigset_t p_pending;              /* mapa bits p/señales pendientes */
05145         unsigned p_pendcount;            /* cuenta de señales pendo e inconclusas */
05146
05147         char p_name[16];                 /* nombre del proceso */
05148                                         };
05149
05150     /* Palabra guardia para pilas de tareas. */
05151     #define STACK_GUARD ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))
05152
05153     /* Bits para p_flags en proc[]. Un proc. es ejecutable si p_flags == 0. */
05154     #define P_SLOT_FREE 001 /* activado si ranura no está en uso */
05155     #define NO_MAP 002 /* evita se ejecute hijo bifurcado sin mapa */
05156     #define SENDING 004 /* enc. si proc. bloqueado tratando de enviar */
05157     #define RECEIVING 010 /* enc. si proc. bloqueado tratando de rec. */
05158     #define PENDING 020 /* enc. si inform() de señal pendiente */
05159     #define SIG_PENDING 040 /* evita se ejecute proc. por señalar */
05160     #define P_STOP 0100 /* enc. si proc. está siendo rastreado */
05161
05162     /* Direcciones mágicas de tabla de procesos. */
05163     #define BEG_PROC_ADDR (&proc[0])
05164     #define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
05165     #define END_TASK_ADDR (&proc[NR_TASKS])
05166     #define BEG_SERV_ADDR (&proc[NR_TASKS])
05167     #define BEG_USER_ADDR (&proc[NR_TASKS + LOW_USER])
05168
05169     #define NIL_PROC ((struct proc *) 0)

```

```

05170 #define isidlehardware(n) ((n) == IDLE :: (n) == HARDWARE);
05171 #define isokprocn(n) ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS);
05172 #define isoksrc_dest(n) (isOkprocn(n):: (n) == ANY);
05173 #define isokusern(n) ((unsigned) (n) < NR_PROCS);
05174 #define isokusern(n) ((unsigned) ((n) -LOW_USER) < NR_PROCS -LOW_USER);
05175 #define isrxhardware(n) ((n) == ANY :: (n) == HARDWARE);
05176 #define issysentn(n) ((n) == FS_PROC_NR :: (n) == MM_PROC_NR);
05177 #define istaskp(p) ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE));
05178 #define isuserp(p) ((p) >= 8EG_USER_ADDR);
05179 #define proc_addr(n) (pproc_addr + NR_TASKS) [(n)];
05180 #define cproc_addr(n) (&(proc + NR_TASKS)[(n)]);
05181 #define proc_number(p)((p) ->p_nr);
05182 #define proc_vir2phys(p, vir) \
05183 -(((Phys_bytes)(p)->p_map1D1.mem_phys « CLICK_SHIFT)\
05184 _ (vir_bytes\ (vir\
05185
05186 EXTERN struct proc proc[NR_TASKS + NR_PROCS]; j* tabla de procesos *;
05187 EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
05188 * apunts. a ranuras de tabla proc.; rápido porque ahora puede encontrarse
05189 una entrada indizando pproc_addr, mientras que acceder al elemento i requiere
05190 multiplicar por sizeof(struct proc) pjdeterm. la dirección *;
05191 EXTERN struct proc *bill_ptr; j* apunto a proc. al que se cobrarán tics *;
05192 EXTERN struct proc *rdy_head[NQ]; ,* apunts. a cabo listas de p. listos *;
05193 EXTERN struct proc *rdy_tail[NQ]; ,* apunts. a fin. listas de p. listos *;
05194
05195 #endif j* PROC_H *;
+++++
srckerneljprotect.h
+++++
05200 * Constantes para modo protegido. *;
05201
05202 * Tamaños de tablas. *;
05203 #define GDT_SIZE (FIRST_LDT_INDEX + NR_TASKS + NR_PROCS) j* espec. y LDT *;
05204 #define IDT_SIZE (IRQ8_VECTOR + 8) * sólo hasta vector más alto *;
05205 #define LDT_SIZE 2 * contiene sólo CS y DS *;
05206
05207 * Descriptores globales fijos. 1 a 7 prescritos por BIOS. *;
05208 #define GDT_INDEX 1 * descriptor GDT *;
05209 #define IDT_INDEX 2 * descriptor IDT *;
05210 #define DS_INDEX 3 * DS de kernel *;
05211 #define ES_INDEX 4 * ES de kernel (386: bando 4 Gb al arranque) *;
05212 #define SS_INDEX 5 * SS de kernel (386: SS monitor al arranque) *;
05213 #define CS_INDEX 6 * CS de kernel *;
05214 #define MON_CS_INDEX 7 * temp pjbIOS (386: CS monitor al arranque) *;
05215 #define TSS_INDEX 8 * TSS de kernel *;
05216 #define DS_286_INDEX 9 * segmento origen 16 bits temporal *;
05217 #define ES_286_INDEX 10 * segmento destino 16 bits temporal *;
05218 #define VIDEO_INDEX 11 * segmento de memoria de video *;
05219 #define DP_ETH0_INDEX 12 * buffer Etherplus de Western Digital *;
05220 #define DP_ETH1_INDEX 13 * buffer Etherplus de Western Digital *;
05221 #define FIRST_LDT_INDEX 14 * resto de descriptores son LDT *;
05222
05223 #define GDT_SELECTOR (GDT_INDEX * DESC_SIZE) mal para asid *;
05224 #define IDT_SELECTOR (IDT_INDEX * DESC_SIZE) *;

```

```
05225 #define DS_SELECTOR      0x18 /* (DS_INDEX * DESC_SIZE) */
05226 #define ES_SELECTOR      0x20 /* (ES_INDEX * DESC_SIZE) */
05227 #define FLAT_DS_SELECTOR 0x21 /* ES menos privilegiado */
05228 #define SS_SELECTOR      0x28 /* (SS_INDEX * DESC_SIZE) */
05229 #define CS_SELECTOR      0x30 /* (CS_INDEX * DESC_SIZE) */
05230 #define MON_CS_SELECTOR  0x38 /* (MON_CS_INDEX * DESC_SIZE) */
05231 #define TSS_SELECTOR     0x40 /* (TSS_INDEX * DESC_SIZE) */
05232 #define DS_286_SELECTOR  0x49 /* (DS_286_INDEX * DESC_SIZE + 1) */
05233 #define ES_286_SELECTOR  0x51 /* (ES_286_INDEX * DESC_SIZE + 1) */
05234 #define VIDEO_SELECTOR   0x59 /* (VIDEO_INDEX * DESC_SIZE + 1) */
05235 #define DP_ETH0_SELECTOR 0x61 /* (DP_ETH0_INDEX * DESC_SIZE) */
05236 #define DP_ETH1_SELECTOR 0x69 /* (DP_ETH1_INDEX * DESC_SIZE) */
05237
05238 /* Descriptores locales fijos. */
05239 #define CS_LDT_INDEX      0 /* CS de proceso */
05240 #define DS_LDT_INDEX     1 /* DS=ES=FS=GS=SS de proceso */
05241
05242 /* Privilegios. */
05243 #define INTR_PRIVILEGE   0 /* manejadores de kernel e interrup. */
05244 #define TASK_PRIVILEGE  1
05245 #define USER_PRIVILEGE  3
05246
05247 /* Constantes de hardware de 286. */
05248
05249 /* Números de vectores de excepción. */
05250 #define BOUNDS_VECTOR    5 /* falló verificación de límites */
05251 #define INVALID_OP_VECTOR 6 /* cód. de operación no válido */
05252 #define COPROC_NOT_VECTOR 7 /* coprocesador no disponible */
05253 #define DOUBLE_FAULT_VECTOR 8
05254 #define COPROC_SEG_VECTOR 9 /* segmento de coproc. invadido */
05255 #define INVALID_TSS_VECTOR 10 /* TSS no válido */
05256 #define SEG_NOT_VECTOR   11 /* segmento no presente */
05257 #define STACK_FAULT_VECTOR 12 /* excepción de pila */
05258 #define PRDTECTION_VECTOR 13 /* protección general */
05259
05260 /* Selector de bits. */
05261 #define TI                0x04 /* indicador de tabla */
05262 #define RPL               0x03 /* nivel de privil. de solicitante */
05263
05264 /* Distancias en estructura de descriptores. */
05265 #define DESC_BASE         2 /* a base_low */
05266 #define DESC_BASE_MIDDLE 4 /* a base_middle */
05267 #define DESC_ACCESS       5 /* al byte de acceso */
05268 #define DESC_SIZE         8 /* sizeof (struct segdesc_s) */
05269
05270 /* Tamaños de segmentos. */
05271 #define MAX_286_SEG_SIZE 0x10000L
05272
05273 /* Tamaños y desplazamientos de base y límite. */
05274 #define BASE_MIDDLE_SHIFT 16 /* despl. para base --> base_middle */
05275
05276 /* Bits de byte de acceso y byte de tipo. */
05277 #define PRESENT           0x80 /* encend. si descriptor presente */
05278 #define DPL               0x60 /* máscara nivel privil. descriptor */
05279 #define DPL_SHIFT        5
05280 #define SEGMENT           0x10 /* encend. p/descriptores tipo segmento */
05281
05282 /* Bits de byte de acceso. */
05283 #define EXECUTABLE        0x08 /* encend. p/segmento ejecutable */
05284 #define CONFORMING        0x04 /* enc. p/segmento conformante si ejecutable */
```

```

05285 #define EXPAND_DOWN 0x04 /* enc. p/segm. expand-abajo si no ejecutable */
05286 #define READABLE 0x02 /* enc. p/seg. legible si ejecutable */
05287 #define WRITEABLE 0x02 /* enc. p/seg. escribible si no ejecutable */
05288 #define TSS_BUSY 0x02 /* encend. si descriptor TSS ocupado */
05289 #define ACCESSED 0x01 /* encend. si segmento accedido */
05290
05291 /* Tipos de descriptores especiales. */
05292 #define AVL_286_TSS 1 /* disponible 286 TSS */
05293 #define LDT- -2 /* tabla de descriptores local */
05294 #define BUSY_286_TSS 3 /* enc. transparentemente al software */
05295 #define CALL_286_GATE 4 /* no utilizado */
05296 #define TASK_GATE 5 /* empleado sólo por depurador */
05297 #define INT_286_GATE 6 /* compuerta interrup., usada p/todos vectores */
05298 #define TRAP_286_GATE 7 /* no utilizado */
05299
05300 /* Constantes de hardware 386 extra. */
05301
05302 /* Números de vectores de excepción. */
05303 #define PAGE_FAULT_VECTOR 14
05304 #define COPROC_ERR=VECTOR 16 /* error de coprocesador */
05305
05306 /* Distancias en la estructura de descriptores. */
05307 #define DESC_GRANULARITY 6 /* a byte de granularidad */
05308 #define DESC_BASE_HIGH 7 /* a base_high */
05309
05310 /* Tamaños y desplazamientos de base y límite. */
05311 #define BASE_HIGH_SHIFT 24 /* despl. para base --> base_high */
05312 #define BYTE-GRAN-MAX 0xFFFFL /* tamaño máx. p/segm. con granul. byte */
05313 #define GRANULARITY_SHIFT 16 /* despl. para límite --> granularidad */
05314 #define OFFSET_HIGH_SHIFT 16 /* despl. p/díst. (comp.) -> offset_high */
05315 #define PAGE_GRAN_SHIFT 12 /* despl. extra p/límites con granul. pág. */
05316
05317 /* Bits de byte de tipo. */
05318 #define DESC_386_BIT 0x08 /* tipos 386 obtenidos haciendo OR con esto */
05319 /* LDT Y TASK_GATE no lo necesitan */
05320
05321 /* Byte de granularidad. */
05322 #define GRANULAR 0x80 /* encend. p/granularidad de 4K */
05323 #define DEFAULT 0x40 /* enc. p/predeterm. 32 bits (seg. ejec.) */
05324 #define BIG 0x40 /* enc. para "BIG" (seg. expand-abajo) */
05325 #define AVL 0x10 /* 0 para disponible */
05326 #define LIMIT_HIGH 0x0F /* máscara p/bits altos del límite */

```

```

+++++
src/kernel/sconst.h
+++++

```

```

05400 Constantes diversas empleadas en código ensamblador.
05401 W=_WORD_SIZE! Tamaño de palabra de máquina.
05402
05403 ! Distancias en struct proc. DEBEN coincidir con proc.h.
05404 P_STACKBASE = 0
05405 #if WORD_SIZE == 2
05406 ESREG -= P_STACKBASE
05407 #else
05408 GSREG = P_STACKBASE
05409 FSREG = GSREG + 2 /* 386 introduce segmentos FS y GS

```

```

05410 ESREG      = FSREG + 2
05411 #endif
05412 DSREG      = ESREG + 2
05413 DIREG      = DSREG + 2
05414 SIREG      = DIREG + W
05415 BPREG      = SIREG + W
05416 STREG      = BPREG + W      ! agujero para otro SP
05417 BXREG      = STREG + W
05418 DXREG      = BXREG + W
05419 CXREG      = DXREG + W
05420 AXREG      = CXREG + W
05421 RETADR     = AXREG + W      ! dir. de retorno p/llamada save()
05422 PCREG      = RETADR + W
05423 CSREG      = PCREG + W
05424 PSWREG     = CSREG + W
05425 SPREG      = PSWREG + W
05426 SSREG      = SPREG + W
05427 P_STACKTOP = SSREG + W
05428 P_LDT_SEL   = P_STACKTOP
05429 P_LDT       = P_LDT_SEL + W
05430
05431 #if _WORD_SIZE = 2
05432 Msize      = 12              ! tamaño de mensaje en palo de 16 bits
05433 #else
05434 Msize      = 9              ! tamaño de mensaje en palo de 32 bits
05435 #endif

```

```

+++++
src/kernel/assert.h
+++++

```

```

05500 /*
05501  assert.h
05502 */
05503 #ifndef ASSERT_H
05504 #define ASSERT_H
05505
05506 #if DEBUG
05507
05508 #define INIT_ASSERT    static char *assert_file= --FILE--;
05509
05510 void bad_assertion(char *file, int line, char *what);
05511 void bad_compare(char *file, int line, int lhs, char *what, int rhs);
05512
05513 #define assert(x)      (! (x) ? bad_assertion(assert_file, --LINE--, #x) \
05514                               : (void) 0)
05515 #define compare(a,t,b) (! ((a) t (b)) ? bad_compare(aSsert_file, --LINE--, \
05516                               (a), #a ' , #t ' , #b, (b)) : (void) 0)
05517 #else 1* !DEBUG *1
05518
05519 #define INIT_ASSERT    1* nada *1
05520
05521 #define assert(x)      (void)0
05522 #define compare(a,t,b) (void)0
05523
05524 #endif 1* !DEBUG *1

```



```

05650     #define PRINTER_STACK SMALL_STACK
05651
05652     #if (CHIP == INTEL)
05653     #define WINCH_STACK      (2 * SMALL_STACK * ENABLE_WINI)
05654     #else
05655     #define WINCH_STACK      (3 * SMALL_STACK * ENABLE_WINI)
05656     #endif
05657
05658     #if (MACHINE == ATARI)
05659     #define SCSI_STACK       (3 * SMALL_STACK)
05660     #endif
05661
05662     #if (MACHINE == IBM_PC)
05663     #define SCSI_STACK       (2 * SMALL_STACK * ENABLE_SCSI)
05664     #endif
05665
05666     #define CDROM_STACK      (4 * SMALL_STACK * ENABLE_CDROM)
05667     #define AUDIO_STACK      (4 * SMALL_STACK * ENABLE_AUDIO)
05668     #define MIXER_STACK      (4 * SMALL_STACK * ENABLE_AUDIO)
05669
05670     #define FLOP_STACK       (3 * SMALL_STACK)
05671     #define MEM_STACK        SMALL_STACK
05672     #define CLOCK_STACK      SMALL_STACK
05673     #define SYS_STACK        SMALL_STACK
05674     #define HARDWARE_STACK 0 /* tarea ficticia, usa pila del kernel */
05675
05676
05677     #define TOT_STACK_SPACE      (TTY_STACK + DP8390_STACK + SCSI_STACK + \
05678     SYN_ALARM_STACK + IDLE_STACK + HARDWARE_STACK + PRINTER_STACK + \
05679     WINCH_STACK + FLOP_STACK + MEM_STACK + CLOCK_STACK + SYS_STACK + \
05680     CDROM_STACK + AUDIO_STACK + MIXER_STACK)
05681
05682
05683     /* SCSI, CDROM y AUDIO podrían tener en el futuro diferentes opciones como
05684     * WINCHESTER, pero por ahora la opción es fija.
05685     */
05686     #define scsi_task          aha_scsi_task
05687     #define cdrom_task        mcd_task
05688     #define audio_task        dsp_task
05689
05690
05691     /*
05692     * Algunas notas respecto a la tabla siguiente:
05693     * 1) La tty_task siempre debe ser la primera para que otras tareas puedan usar
05694     *    printf si su inicialización tiene problemas.
05695     * 2) Si ud. agrega una nueva tarea de kernel, hágalo antes de la de impresora.
05696     * 3) Se usa el nombre de tarea como nombre de proceso (p_name).
05697     */
05698
05699     PUBLIC struct tasktab tasktab[] = {
05700     { tty_task,          TTY_STACK,      "TTY"          },
05701     #if ENABLE_NETWORKING
05702     { dp8390_task,      DP8390_STACK,  "DP8390",     },
05703     #endif
05704     #if ENABLE_CDROM
05705     { cdrom_task,       CDROM_STACK,  "CDROM"      },
05706     #endif
05707     #if ENABLE_AUDIO
05708     { audio_task,       AUDIO_STACK,  "AUDIO"      },
05709     { mixer_task,      MIXER_STACK,  "MIXER"      },

```



```
+++++
src/kernel/mpx386.s
+++++
05900 /*
05901 * Este archivo contiene el código de arranque en ensamblador para Minix y los 05902 ! manejadores de interrup. de 32 bits. Coopera con start.c para establecer un
05903 * buen entorno para main().
05904
05905 *Este arch. es parte de la capa más baja del kernel MINIX. La otra parte es 05906 ! "proc.c". Esta capa conmuta procesos y maneja mensajes.
05907
05908 * Toda transición al kernel pasa por este archivo. Causan transiciones el 05909 ! envío/recepción de mensajes y casi todas las interrupciones.
05910 * interrupciones RS232 pueden manejarse en "rs2.s" y rara vez ingresan en el 05911 ! kernel.)
05912
05913 * Las transiciones al kernel pueden anidarse. El ingreso inicial puede ser
05914 * con llamada al sistema, excepción o interrup. de hardware; los reingresos
05915 * sólo pueden ser por int. de hard. La cuenta de reingresos se mantiene en
05916 * "k.reenter". Es importante p/decidir si cambiar a la pila del kernel y p/
05917 * proteger el cód. de transf. de mensajes en "proc.c".
05918
05919 * Para la trampa de transf. de mens., casi todo el estado de máq. se guarda
05920 * en la tabla de proc. (Algunos regs. no tienen que guardarse.) Luego la pila
05921 * se conmuta a "k.stack" y se rehabilitan las ints. Por último se invoca el
05922 * maneja. de llamadas (en C). Al regresar, las ints. se inhabilitan otra vez y
05923 * el cód. continúa con la rutina de reinicio, para finalizar ints. Detenidas
05924 * Y ejecutar el proc. o tarea cuyo apunto está en "proc_ptr".
05925
05926 * Los manejadores de ints. de hardware hacen lo mismo, excepto que (1) todo
05927 * el estado debe guardarse. (2) Hay demasiados manejadores para hacerlo en
05928 * línea; así, se invoca save. Se ahorran unos ciclos formando en pila la dir.
05929 * de la rutina de reinic. adecuada para retorno posterior. (3) Se evita
05930 * cambio de pila si ya se cambió. (4) El contr. de ints. (maestro) 8259 se
05931 * rehabil. centralmente en save(). (5) Cada manejador enmascara su línea de
05932 * int. con el 8259 antes de habilitar otras ints. (enmascaradas), y la
05933 * desenmascara después de atender la int. Esto limita el nivel de anidación
05934 * al núm. de líneas y protege al manejador contra sí mismo.
05935
05936 * Para comunicarse con el monitor de arranque al inicio algunos datos
05937 * constantes se compilan al principio del segmento de texto. Esto facilita la
05938 * lectura de los datos al comenzar el proc. de arranque, ya que sólo hay que
05939 * leer el primer sector del archivo.
05940
05941 * También se asigna memoria a datos al final de este archivo. Estos datos
05942 * estarán al principio del segm. de datos del kernel y serán leídos y
05943 * modificados por el monitor de arranque antes de que el kernel inicie.
05944
05945 * secciones
05946
05947 sect .text
05948 #begtext:
05949 #sect .rom
05950 begrom:
05951 .sect .data
05952 begdata:
05953 .sect .bss
05954 begbss:
```

```
05955
05956 #include <minix/config.h>
05957 #include <minix/const.h>
05958 #include <minix/com.h>
05959     #include "const.h"
05960     '#include "protect.h"
05961     #include "sconst.h"
05962
05963     /* Distancias tss 386 selectas. */
05964     #define TSS3_S_SP0    4
05965
05966     /* Funciones exportadas.
05967     * Nota: en lenguaje ensamblador el enunciado .define aplicado a un nombre de
05968     * función equivale aprox. a un prototipo en código C --hace posible vincularse
05969     * con una entidad declarada en el código de ensamblador, pero no crea la
05970     * entidad. */
05971
05972     #define _idle_task
05973     #define _restart
05974     #define save
05975
05976     #define _divide_error
05977     #define _single_step_exception
05978     #define _nmi
05979     #define _breakpoint_exception
05980     #define _overflow
05981     #define _bounds_check
05982     #define _inval_opcode
05983     #define _copr_not_available
05984     #define _double_fault
05985     #define _copr_seg_overrun
05986     #define _inval_tss
05987     #define _segment_not_present
05988     #define _stack_exception
05989     #define _general_protection
05990     #define _page_fault
05991     #define _copr_error
05992
05993     #define hwint00          /* manejadores para interrupciones de hardware */
05994     #define _hwint01
05995     #define hwint02
05996     #define _hwint03
05997     #define _hwint04
05998     #define hwint05
05999     #define _hwint06
06000     #define _hwint07
06001     #define _hwint08
06002     #define _hwint09
06003     #define hwint10
06004     #define _hwint11
06005     #define _hwint12
06006     #define _hwint13
06007     #define _hwint14
06008     #define hwint15 06009
06010     #define _s_call
06011     #define _p_s_call
06012     #define _level0_call
06013
06014     /* Funciones importadas.*/
```

```

06015
06016 extern _cstart
06017 .extern _main
06018 .extern ~exception
06019 .extern _interrupt
06020 .externl _sys_call
06021 .extern _unhold
06022
06023 Variables exportadas.
06024 Nota: usado con una variable, .define no reserva memoria, hace al nombre 06025 ! visible externamente para que sea
posible vincularse a él.
06026
06027 #define begbss
06028 #define begdata
06029 #define _sizes
06030
06031 /* Variables importadas.*/
06032
06033 .extern _gdt
06034 .extern _code_base
06035 .extern _data_base
06036 .extern _held_head
06037 .extern _k_reenter
06038 .extern _pc_at
06039 .extern _proc_ptr
06040 .extern _ps_mca
06041 .extern _tss
06042 .extern _leve10_func
06043 .extern _mon_sp
06044 .extern _mon_return
06045 .extern _reboot_code
06046
06047 .sect      .text
06048
=====
06049
06050
=====
06051 MINIX:
06052     jmp      over_flags      /* punto de entrada del kernel de MINIX */
06053     .data2  CLICK_SHIFT    /* saltarse unos cuantos bytes */
06054     flags:
06055     .data2  0x002D         /* para el monitor: granularidad de memoria */
06056
06057     /* banderas del monitor de arranque: */
06058     /*     llamar en modo 386, hacer pila,*/
06059     /* cargar alto, retornará */
06060     nop
06061     over_flags:
06062     /* byte extra para sincronizar desensamblador */
06063
06061 Preparar marco de pila C en pila del monitor. (Monitor establece es y ds 06062 ! correctamente. El descrip. ss aún se refiere al segm. de datos del mon.)
06063     movzx   esp, sp         /* pila de monitor es pila de 16 bits*/
06064     push   ebp
06065     mov    ebp, esp
06066     push   esi
06067     push   edi
06068     cmp    4(ebp), 0        /* no 0 si es posible el retorno*/
06069     jz     noret
06070     inc    (_mon_return)
06071 noret:   mov    (_mon_sp) , esp /* guardar apunto a pila p/regresar luego*/
06072
06073     /* Copiar tabla de descriptores global del mono en el esp. de dir. del kernel
06074     * Y conmutar a ella. Así, prot_init() puede actualizar la con efecto inmediato. */

```



```
06135 /*                                manejadores de interrupciones                                *
06136 /*      manejadores de interrupciones para modo protegido 386 de 32 bits                                *
06137 /*=====
06138
06139 /*=====
06140 /*                                hwint00 -07                                *
06141 /*=====
06142 /* Aunque parece subrutina, ésta es una macro.*/
06143 #define hwint master(irq)
06144     call -save                                /* guardar estado proc. interrumpido */
06145     inb     INT CTLMASK                                );
06146     orb     al,-[1<<irq]                                );
06147     outb    INT CTLMASK                                /* inhabilitar irq*/
06148     movb   al,-ENABLE                                );
06149     outb    INT CTL                                /* rehabilitar 8259 maestro*/
06150     sti                                /* habilitar interrupciones*/
06151     push   irq                                /* irq*/
06152     call   (irq table + 4*irq)                /* eax = (*irq_table[irq])(irq)*/
06153     pop    ecx -;\
06154     cli                                /* inhabilitar interrupciones*/
06155     test   eax, eax                                /* ¿necesario rehabilitar irq?*/
06156     jz     0f                                );
06157     inb    INT CTLMASK                                );
06158     andb   al,-[1<<irq]                                );
06159     outb   INT CTLMASK                                /* habilitar irq*/
06160     0:    ret                                /* reiniciar (otro) proceso*/
06161
06162 /*C/U de estos ptos. de entrada es una expansión de la macro hwint_master*/
06163 align 16
06164 _hwint00:    /* Rutina de int. para irq 0 (el reloj)*/
06165 hwint_master(0)
06166
06167 align 16
06168 _hwint01:    /* Rutina de int. para irq 1 (teclado)*/
06169 hwint_master(1)
06170
06171 align 16
06172 _hwint02:    /* Rutina de int. para irq 2 (icascada!)*/
06173 hwint_master(2)
06174
06175 align 16
06176 _hwint03:    /* Rutina de int. para irq 3 (segundo serial)*/
06177 hwint_master(3)
06178
06179 align 16
06180 _hwint04:    /* Rutina de int. para irq 4 (primer serial)*/
06181 hwint_master(4)
06182
06183 align 16
06184 _hwint05:    /* Rutina de int. para irq 5 (winchester XT)*/
06185 hwint_master(5)
06186
06187 align 16
06188 _hwint06:    /* Rutina de int. para irq 6 (disquete)*/
06189 hwint_master(6)
06190
06191 align 16
06192 _hwint07:    /* Rutina de int. para irq 7 (impresora)*/
06193 hwint_master(7)
06194
```



```

06195 /*=====
06196 /*                               hwint08 -15                               *
06197 /*=====
06198     /* Aunque parece subrutina, ésta es una macro.*/
06199     #define hwint_slave(irq)
06200     call    save                               /* guardar estado proc. interrumpido */
06201     inb    INT2_CTLMASK;
06202     orb    al, [1<[irq-8]];
06203     outb   INT2_CTLMASK                       /* inhabilitar irq*/
06204     movb   al, ENA8LE;
06205     outb   INT_CTL                           /* rehabilitar 8259 maestro*/
06206     jmp    .+2                                /* retardo*/
06207     outb   INT2_CTL                           /* rehabilitar 8259 esclavo*/
06208     sti                                         /* habilitar interrupciones*/
06209     push   irq                                /* irq*/
06210     call   (_irq_table + 4*irq)              /* eax = (*irq_table[irq]) (irq)*/
06211     pop    ecx
06212     cli                                         /* inhabilitar interrupciones*/
06213     test   eax, eax                          /* ¿necesario rehabilitar irq?*/
06214     jz     0f
06215     inb    INT2_CTLMASK
06216     andb   al, -[1<[irq-8]]
06217     outb   INT2_CTLMASK                       /* habilitar irq*/
06218     0:    ret                                /* reiniciar (otro) proceso*/
06219
06220     /* C/u de estos ptas. de entrada es una expansión de la macro hwint_slave*/
06221     align  16
06222     _hwint08:                               /* Rutina de int. para irq 8 (reloj tiempo real)*/
06223     hwint_slave(8)
06224
06225     align  16
06226     _hwint09:                               /* Rutina de int. para irq 9 (irq 2 redirigida)*/
06227     hwint_slave(9)
06228
06229     align  16
06230     _hwint10:                              /* Rutina de int. para irq 10*/
06231     hwint_slave(10)
06232
06233     align  16
06234     _hwint11:                              /* Rutina de int. para irq 11*/
06235     hwint_slave(11)
06236
06237     align  16
06238     _hwint12:                              /* Rutina de int. para irq 12*/
06239     hwint_slave(12)
06240
06241     align  16
06242     _hwint13:                              /* Rutina de int. para irq 13 (excepción FPU)*/
06243     hwint_slave(13)
06244
06245     align  16
06246     _hwint14:                              /* Rutina de int. para irq 14 (winchester AT)*/
06247     hwint_slave(14)
06248
06249     align  16
06250     _hwint15:                              /* Rutina de int. para irq 15*/
06251     hwint_slave(15)
06252
06253 /*=====
06254 /*save */

```

```

06255 /*=====
06256      /* Save para modo protegido.*/
06257      /* Es mucho más simple que para modo 8086 porque la pila ya apunta a la tabla*/
06258      /* de procesos, o ya se conmutó a la pila del kernel.*/
06259
06260      align    16
06261      save:
06262      cld      /* dar valor conocido a bando dirección*/
06263      pushad   /* guardar registros "generales"*/
06264      016 push  ds      /* guardar ds*/
06265      016 push  es      /* guardar es*/
06266      016 push  fs      /* guardar fs*/
06267      016 push  gs      /* guardar gs*/
06268      mov     dx, ss    /*ss es segm. de datos de kernel*/
06269      mov     ds, dx    /*cargar resto de segm. de kernel*/
06270      mov     es, dx    /*el kernel no usa fs, gs*/
06271      mov     eax, esp  /*preparar retorno*/
06272      incb    (_k_reenter) /* de -1 si no es reingreso*/
06273      jnz     set_restart1 /*pila ya es pila del kernel*/
06274      mov     esp, k_stktop
06275      push   _restart    /* formar dir. de retorno p/manej. ints.*/
06276      xor    ebp, ebp   /* para rastreo de pila*/
06277      jmp    RETADR-P_STACKBASE(eax)
06278
06279      align    4
06280      set_restart1:
06281          push  restart1
06282          jmp  RETADR-P_STACKBASE(eax)
06283
06284 /*=====
06285 /*                                     _s_call      *
06286 /*=====
06287      align    16
06288      _s_call:
06289      _p_s_call:
06290      cld      /* dar valor conocido a bando dirección*/
06291      sub     esp, 6*4  /* saltar RETADR, eax, ecx, edx, ebx, est*/
06292      push   ebp      /* pila ya apunta a tabla de proc.*/
06293      push   esi
06294      push   edi
06295      016 push  ds
06296      016 push  es
06297      016 push  fs
06298      016 push  gs
06299      mov     dx, ss
06300      movds, dx
06301      mov     es, dx
06302      incb    (_k_reenter)
06303      mov     esi, esp  /* supone P_STACKBASE == 0*/
06304      mov     esp, k_stktop
06305      xor    ebp, ebp  /* para rastreo de pila*/
06306      /*fin de save en línea*/
06307      sti     /* permitir interrumpir SWITCHER*/
06308      /*/prep. paráms. para sys_call()*/
06309      push   ebx      /* apuntador a mensaje de usuario*/
06310      push   eax      /* src/dest*/
06311      push   ecx      /* SEND/RECEIVE/BOTH*/
06312      call   _sys_call /* sys_call(function, src_dest, m_ptr)*/
06313      /* invocador está explícitamente en proc_ptr*/
06314      mov     AXREG(esi), eax /* sys_call DEBE CONSERVAR si*/

```

```

06315         cli                               /* inhabilitar interrupciones*/
06316
06317         /* Continuar con el código para reiniciar proc/tarea en ejecución.*/
06318
06319 /*=====
06320 /*                               restart                               */
06321 /*=====
06322 _restart:
06323
06324 /* Vaciar cualesquier interrupciones detenidas. *
06325 * Esto rehabilita ints, así que el manejador de ints actual puede reingresar.*
06326 * No importa, porque el manejador actual va a salir y ningún otro puede *
06327 /* reingresar, pues el vaciado sólo se hace cuando k_reenter == 0. *
06328
06329         cmp      (_held_head) , 0 /* prueba rápida p/evitar llamada a función*/
06330         jz       over_call_unhold
06331         call    _unhold           /* poco frecuente, así que costo extra aceptable*/
06332 over call unhold:
06333         -mov    esp, (_proc_ptr) /* supondrá P_STACKBASE == 0*/
06334         lldt   P LDT_SEL(esp)   /* habilitar descript. segm. p/tarea*/
06335         lea    eax, P STACKTOP(esp) /* prep. p/siguiente interrupción*/
06336         mov    (_tsS+TSS3_S_SP0) , eax /* p/guardar estado en tabla proc.*/
06337 restart1:
06338         decb   (_k_reenter)
06339         016 pop  gs
06340         016 pop  fs
06341         016 pop  es
06342         016 pop  ds
06343 popad
06344         add    esp, 4           /* saltarse dirección de retorno*/
06345         iretd                    /* continuar proceso*/
06346
06347 /*=====
06348 /*                               manejadores de excepciones                               */
06349 /*=====
06350 _divide_error:
06351         push   DIVIDE_VECTOR
06352         jmp    exception
06353
06354 _single_step_exception:
06355         push   DEBUG_VECTOR
06356         jmp    exception
06357
06358 _nmi:
06359         push   NMI_VECTOR
06360         jmp    exception
06361
06362 _breakpoint_exception:
06363         push   BREAKPOINT_VECTOR
06364         jmp    exception
06365
06366 _overflow:
06367         push   OVERFLOW_VECTOR
06368         jmp    exception
06369
06370 _bounds_check:
06371         push   BOUNDS_VECTOR
06372         jmp    exception
06373
06374 _inval_opcode:

```



```

06435     push    eax                /* eax es registro temporal*/
06436     mov     eax, 0+4(esp)        /*viejo eip*/
06437     sseg   mov     (old_eip), eax
06438     movzx  eax, 4+4(esp)        /* viejo cs*/
06439     sseg   mov     (old_cs) , eax
06440     mov     eax, 8+4(esp)        /* viejo eflags*/
06441     sseg   mov     (old_eflags), eax
06442     pop     eax
06443     call   save
06444     push   (old_eflags)
06445     push   (old_cs)
06446     push   (old_eip)
06447     push   (trap_erno)
06448     push   (ex_number)
06449     call   _exception (ex_number, trap_erno, old_eip,
06450                                old_cs, old_eflags)
06451     add    esp, 5*4
06452     cli
06453     ret
06454
06455 /*=====
06456 /*                                     level0_call *
06457 /*=====
06458     _level0_call:
06459         call   save
06460         jmp    (_level0_func)
06461
06462 /*=====
06463 /*                                     idle_task *
06464 /*=====
06465 idle task:                /* se ejecuta si no hay trabajo*/
06466     --jmp   _idle_task      /* "hlt" antes de esto falla en modo prot.*/
06467
06468 /*=====
06469 /*                                     datos *
06470 /*=====
06471     /* Estas declaraciones aseguran que se asignará memoria al principio *
06472     /* de la sección de datos del kernel, para poder informar fácilmente*
06473     /* al monitor de arranque cómo parchar estas posiciones. El compilador *
06474     /* pone aquí el núm. mágico, pero el monitor lo lee y sobrescribe. *
06475     /* Al iniciar el kernel el arreglo de tamaños se encontrará aquí, *
06476     /* como si hubiera sido inicializado por el compilador.
06477
06478 .sect .rom                /* Antes de la tabla de cadenas, por favor*/
06479 sizes:                   /* arranque llena tamaño kernel, mm, fs*/
06480     -.data2 0x526F        /* debe ser 1a. entrada datos (# mágico)*/
06481     .space 16*2*2-2      /* monitor usa palo ant. y este espacio*/
06482                         /* espacio extra p/servidores adic.*/
06483 .sect .bss
06484 k stack:
06485 -.space K_STACK_BYTES  /* pila de kernel*/
06486 k_stktop:               /* tope de pila de kernel*/
06487     .comm   ex_number, 4
06488     .comm   trap_erno, 4
06489     .comm   old_eip, 4
06490     .comm   old_cs, 4
06491     .comm   old_eflags, 4

```

```

+++++
                                     src/kernel/start.c
+++++
06500      /* Este arch. contiene el código de inicio en C para Minix en procesadores
06501 * Intel; coopera con mpx.s para estable buen entorno para main().
06502      *
06503 * Este código se ejecuta en modo real para un kernel de 16 bits y podría
06504 * tener que cambiar a modo protegido para un 286.
06505      *
06506 * Para un kernel de 32 bits ya ejec. en modo prot., pero los selectores
06507 * aún son los dados por BIOS con ints. inhabilitadas, así que hay que
06508 * volver a cargar descriptores y crear descriptores de interrupciones.
06509      */
06510
06511      #include "kernel.h"
06512      #include <stdlib.h>
06513      #include <minix/boot.h>
06514      #include "protect.h"
06515
06516      PRIVATE char k_environ[256]; /* cadenas de entorno que pasa el cargador */
06517
06516      FORWARD_PROTOTYPE( int k_atoi, (char *s) );
06519
06520
06521 /*=====
06522 *                                     cstart
06523 *=====*/
06524      PUBLIC void cstart(cs, ds, mcs, mds, parmoff, parmsize)
06525 U16 t cs, ds;                                     /* segm. de código y datos del kernel */
06526 U16-t mcs, mds;                                   /* segm. de código y datos del monitor */
06527 U16=t parmoff, parmsize;                          /* disto y long. de paráms. de arranque */
06528 {
06529      /* Realizar inicializaciones del sistema antes de invocar main(). */
06530
06531      register char *envp;
06532      phys_bytes mcode_base, mdata_base;
06533      unsigned mon_start;
06534
06535      /* Tomar nota de dónde están el kernel y el monitor. */
06536      code_base = seg2phys(cs);
06537      data_base = seg2phys(ds);
06536      mcode_base = seg2phys(mcs);
06539      mdata_base = seg2phys(mds) ;
06540
06541      /* Inicializar descriptores de modo protegido. */
06542      prot_init();
06543
06544      /* Copiar parámetros de arranque en memoria del kernel. */
06545      if (parmsize > sizeof k_environ -2) parmsize = sizeof k_environ -2;
06546      phys_copy(mdata_base + parmoff, vir2phys(k_environ) , (phys_bytes) parmsize);
06547
06548      /* Convertir variables de entorno de arranque importantes. */
06549      boot_parameters.bp_rootdev = k_atoi(k_getenv("rootdev"));
06550      boot_parameters.bp_ramimagedev = k_atoi(k_getenv("ramimagedev"));
06551      boot_parameters.bp_ramsize = k_atoi(k_getenv("ramsize"));
06552      boot_parameters.bp_processor = k_atoi(k_getenv("processor"));
06553
06554      /* Tipo de VDU: */

```

```

06555 envp = k_getenv("video");
06556 if (strcmp(envp, "ega") == 0) ega = TRUE;
06557 if (strcmp(envp, "vga") == 0) vga = ega = TRUE;
06558 .
06559 /* Tamaños de memoria: */
06560 low memsize = k_atoi(k_getenv("memsize">>);
06561 ext=:memsize = k=:atoi(k=:getenv("emssize"»);j
06562
06563 /* ¿Procesador? */
06564 processor = boot_parameters.bp_processor; /* 86, 186, 286, 386, ...*/
06565
06566 /* ¿Bus XT, AT o MCA? */
06567 envp = k_getenv("bus");
06568 if (envp == NIL_PTR :: strcmp(envp, "at") == 0) {
06569     pc_at = TRUE;
06570 } else
06571 if (strcmp(envp, "mca") == 0) {
06572     pc_at = ps_mca = TRUE;
06573 }
06574
06575 /* Decidir si el modo es protegido. */
06576 #if _WORD_SIZE == 2
06577 protected_mode = processor >= 286;
06578 #endif
06579
06580 /* ¿Hay monitor al cual volver? En tal caso, no arriesgar. */
06581 if (!protected_mode) mon_return = 0;
06582 mon_start = mcode_base / 1024;
06583 if (mon_return &&-!low_memsize > mon_start) low_memsize = mon_start;
06584
06585 /* Volver a código ensamblador para cambiar a modo protegido (si 286),
06586 * volver a cargar selectores e invocar main().
06587 */
06588 }
06591 /*=====
06592 *                               k_atoi                               *
06593 *=====*/
06594 PRIVATE int k_atoi(s)
06595 register char *S;
06596 {
06597 /* Convertir cadena en entero. */
06598
06599 return strtol(s, (char **) NULL, 10);
06600 }
06603 /*=====
06604 *                               k_getenv                               *
06605 *=====*/
06606 PUBLIC char *k_getenv(name)
06607 char *name;
06608 {
06609 /* Obtener valor de entorno -versión del kernel de getenv par evitar
06610 * crear el arreglo de entorno usual.
06611 */
06612
06613 register char *namep;
06614 register char *envp;

```

```
06615
06616   for (envp = k_environ; *envp != 0;) {
06617       for (namep = name; *namep != 0 && *namep == *envp; namep++, envp++;)
06618
06619           if (*namep == '\0' && *envp == '=') return(envp + 1);
06620       while (*envp++ != 0)
06621
06622   }
06623       return(NIL_PTR);
06624   }
+++++
src/kernel/main.c
+++++
06700   /* Este archivo contiene el programa principal de MINIX. La rutina main()
06701   * inicializa el sistema y pone la pelota en juego estableciendo la tabla
06702   * de procesos, vectores de interrupciones y planificando la ejecución de
06703   * todas las tareas para que se inicialicen a sí mismas.
06704   *
06705   * Las entradas de este archivo son:
06706   *   main:           programa principal de MINIX
06707   *   panic:          abortar MINIX por error fatal
06708   */
06709
06710 #include "kernel.h"
06711 #include <signal.h> 06712 #include <unistd.h>
06713 #include <minix/callnr.h>
06714 #include <minix/com.h>
06715 #include "proc.h"
06716
06717
06718 /*=====
06719 *                               main                               *
06720 *=====*/
06721   PUBLIC void main()
06722   {
06723   /* Poner en juego la pelota. */
06724
06725   register struct proc *rp;
06726   register int t;
06727   int sizeindex;
06728   phys_clicks text_base;
06729   vir_clicks text_clicks;
06730   vir_clicks data_clicks;
06731   phys_bytes phys_b;
06732   reg_t ktsb;                               /* base de pila de tareas de kernel */
06733   struct memory *memp;
06734   struct tasktab *ttp;
06735
06736   /* Inicializar el controlador de interrupciones. */
06737       intr_init(1);
06738
06739   /* Interpretar tamaños de memoria. */
```



```

06740     mem_init();
06741
06742 /* Despejar la tabla de procesos.
06743 * Establecer correspondencias para macros proc_addr() y proc_number().
06744 */
06745     .for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t) {
06746         rp->p_flags = P_SLOT_FREE;
06747         rp->p_nr = t; /* núm. proc. de apuntador */
06748         (pproc_addr + NR_TASKS)[t] = rp; /* apunto a proc. de número */
06749     }
06750
06751 /* Crear entradas de tabla proc. p/tareas y servidores. Las pilas de las tareas
06752 * de kernel se inicializan a un arreglo en esp. de datos. El monitor agregó
06753 * las pilas de servidores al segmo de datos, así que el apunto a pila
06754 * se dirige al final del segm. de datos. Todos los proc. están en memoria
06755 * baja en el 8086. En el 386 sólo el kernel está en memoria baja, el resto
06756 * se carga en memoria extendida.
06757 */
06758
06759     /* Pilas de tareas. */
06760     ktsb = (reg_t) t_stack;
06761
06762     for (t = -NR_TASKS; t <= LOW_USER; ++t) {
06763         rp = proc_addr(t); /* ranura de proco de t */
06764         ttp = &tasktab[t + NR_TASKS]; /* atributos de tarea de t */
06765         strcpy(rp->p_name, ttp->name);
06766         if (t < 0) {
06767             if (ttp->stksize > 0) {
06768                 rp->p_stguard = (reg_t *) ktsb;
06769                 *rp->p_stguard = STACK_GUARD;
06770             }
06771             ktsb += ttp->stksize;
06772             rp->p_reg.sp = ktsb;
06773             text_base = code_base » CLICK_SHIFT;
06774             /* todas las tareas están en el kernel */
06775             sizeindex = 0; /* Y usan tamaños de kernel plenos */
06776             memp = &mem[0]; /* quitar de este trozo de memoria */
06777             } else {
06778                 sizeindex = 2 * t + 2; /* MM, FS, INIT tienen sus tamaños */
06779             }
06780             rp->p_reg.pc = (reg_t) ttp->initial_pc;
06781             rp->p_reg.psw = istaskp(rp) ? INIT_TASK_PSW : INIT_PSW;
06782
06783             text_clicks = sizes[sizeindex];
06784             data_clicks = sizes[sizeindex + 1];
06785             rp->p_map[T].mem_phys = text_base;
06786             rp->p_map[T].omem_len = text_clicks;
06787             rp->p_map[D].mem_phys = text_base + text_clicks;
06788             rp->p_map[D].mem_len = data_clicks;
06789             rp->p_map[S].mem_phys = text_base + text_clicks + data_clicks;
06790             rp->p_map[S].mem_vir = data_clicks; /* vacío -pila en datos */
06791             text_base += text_clicks + data_clicks; /* listo p/sigte., si servidor */
06792             memp->size -= (text_base - memp->base);
06793             memp->base = text_base; /* memoria ya no libre */
06794
06795             if (t >= 0) {
06796                 /* Inicializar apunto a pila de servidores. Bajarlo una palabra
06797                  * para dar a crtso.S algo que usar como "argc".
06798                  */
06799                 rp->p_reg.sp = (rp->p_map[S].mem_vir +

```

```
06800             rp->p_map[S].mem_len) «      CLICK_SHIFT;
06801             rp->p_reg.sp -= sizeof(reg_t);
06802             }
06803
06804     #if WORD_SIZE == 4
06805             /* Los servidores se cargan en memoria extendida en modo 386. */
06806             if (t < 0) {
06807                 memp = &mem[1];
06808                 text_base = 0x100000 »      CLICK_SHIFT;
06809             }
06810     #endif
06811             if (!isidlehardware(t) && lock_ready(rp); /* IDLE, HARDWARE neveready */
06812             rp->p_flags = 0;
06813
06814             alloc_segments(rp);
06815             }
06816
06817     proc[NR_TASKS+INIT_PROC_NR].p_pid = 1; /* INIT tiene pid 1 */
06818     bill_ptr = proc_addr(IDLE); /* tiene que apuntar a algo */
06819     lock_pick_proc();
06820
06821     /* Pasar al cód. de ensamblador p/inic. ejecución del proc. actual. */
06822     restart();
06823     }
06826
06827 *
06828
06829 *
06830 *
06831 *
06832 *
06833 *
06834 *
06835 *
06836 *
06837 *
06838
06839     PUBLIC void panic(s,n)
06840     _CONST char *s;
06841     int n;
06842     {
06843     /* El sistema experimentó un error fatal. Terminar ejecución.
06844     * Si el pánico se originó en MM o FS, la cadena está vacía
06845     * Y el FS ya está sincronizado. Si el pánico se originó
06846     * en el kernel, estamos perdidos.
06847     */
06848     if (*s != 0) {
06849         printf("\nKernel panic: %s",s);
06850         if (n != NO_NUM) printf(" %d", n);
06851         printf("\n");
06852     }
06853     wreboot(RBT_PANIC);
06854     }
```

```

+++++
                                     src/kernel/proc.c
+++++
06900      /* Este archivo contiene casi todo el manejo de procesos y mensajes.
06901 * Tiene dos puntos de entrada desde el exterior:
06902 *
06903 *   sys_call: invocada cuando un proc. o tarea hace SEND, RECEIVE o SENREC
06904 *   interrupt: invocada por rutinas de int. p/enviar mensaje a una tarea
06905 *
06906 * También tiene varios puntos de entrada secundarios:
06907 *
06908 *   lock_ready:      poner proc. en una cola de listos p/poder ejecutarse
06909 *   lock_unready:   quitar proc. de cola de listos
06910 *   lock_sched:     proc. se ejecutó mucho tiempo; planif. otro
06911 *   lock_mini_send: enviar mensaje (usado por señales de int., etc.)
06912 *   lock_pick_proc: escoger proc. p/ejecutar (usado por inicializ. sist.)
06913 *   unhold:         repetir todas las interrupciones detenidas
06914 *1
06915
06916 #include "kernel.h"
06917 #include <minix/callnr.h>
06918 #include <minix/com.h>
06919 #include "proc.h"
06920
06921 PRIVATE unsigned char switching;          /* no 0 para inhibir interrupt() *1
06922
06923 FORWARD_PROTOTYPE( int mini_send, (struct proc *caller_ptr, int dest,
06924 message *m_ptr) );
06925 FORWARD_PROTOTYPE( int mini_rec, (struct proc *caller_ptr, int src,
06926 message *m_ptr) ) j
06927 FORWARD_PROTOTYPE( void ready, (struct proc *rp) );
06928 FORWARD_PROTOTYPE( void sched, (void) )j
06929 FORWARD_PROTOTYPE( void unready, (struct proc *rp) );
06930 FORWARD_PROTOTYPE( void pick_proc, (void) );
06931
06932 #define CopyMess(s,sp,sm,dp,dm) \
06933 cp_mess(s, (sp) ->p_map[D].mem_phys, (vir_bytes)sm, (dp) ->p_map[D].mem_phys, (vir_bytes)dm)
06934
06935 /*=====
06936 *                                     interrupt                                     *
06937 *=====*/
06938 PUBLIC void interrupt(task)
06939 int task; /* # de la tarea que se iniciará */
06940 {
06941     /* Ocurrió interrupción. Planificar la tarea que la maneja. */
06942
06943     register struct proc *rp;          /* apunto a entrada de proc. de la tarea */
06944
06945     rp = proc_addr(task);
06946
06947     * Si esta llamada competiría con otras func. de conmut. de procs.,
06948     * ponerla en la cola 'detenidos' que se vaciará en el sigte. restart()
06949     * no competidor. Las condiciones competidoras son:
06950     * (1) k_reenter == (typeof k_reenter) -1:
06951     *     Llamada de nivel de tareas, por lo común de una rutina
06952     * de int. de salida. Un maneje. de ints. podría reingresar a interrupt().;
06953     * Poco frecuente; no amerita tratamiento especial.    )1
06954     * (2) k_reenter > 0:

```

```

06955 *      Llamada de mane. de ints. anidado. Un manejo de ints.
06956 *      previo podría estar dentro de interrupt() o sys_call().
06957 * (3) switching l= 0:
06958 *      Una tunco de conmut. de proc. distinta de interrupt() está siendo invocada
06959 *      del nivel de tareas, por lo común sched() de CLOCK. Un manejo
06960 *      de ints. podría invocar interrupt y pasar la prueba k_reenter.
06961 */
06962 if (k_reenter != 0 11 switching) {
06963     lock(i)
06964         if (!rp->p_int_held) {
06965             rp->p_int_held = TRUEi
06966             if (held_head != NIL_PROC)
06967                 held_tail->p_nextheld = rpi
06968         else
06969             held_head = rp;
06970             held_tail = rpi
06971             rp->p_nextheld = NIL_PROC;
06972     }
06973     unlock(i)
06974     return;
06975 }
06976
06977 /* Si la tarea no espera interrupción, registrar el bloqueo. */
06978 if ( (rp->p_flags & (RECEIVING 1 SENDING) & 1) == RECEIVING ;
06979     !isrxhardware(rp->p_getfrom)) {
06980     rp->p_int_blocked = TRUE;
06981     return;
06982 }
06983
06984 /* El destino está esperando una interrupción.
06985 * Enviarle mensaje con origen HARDWARE y tipo HARD_INT.
06986 * No puede proporcionarse de forma confiable más información porque los
06987 * mensajes de interrupción no se ponen en cola.
06988 */
06989 rp->p_messbuf->m_source = HARDWAREi
06990 rp->p_messbuf->m_type = HARD_INT;
06991 rp->p_flags &= -RECEIVING;
06992 rp->p_int_blocked = FALSE;
06993
06994 /* Hacer a rp listo y ejecutarlo si otra tarea no se está ejecutando ya. Esto
06995 * es ready(rp) en-línea para mayor velocidad.
06996 */
06997 if (rdy_head[TASK_Q] l= NIL_PROC)
06998     rdy_tail[TASK_Q] ->p_nextready = rpi
06999     else
07000     proc_ptr = rdy_head[TASK_Q] = rp;
07001 rdy_tail[TASK_Q] = rp;
07002 rp->p_nextready = NIL_PROC;
07003 }
07004
07005 /*=====
07006 *                               sys_call
07007 *=====*/
07008 PUBLIC int sys_call(function, src_dest, m_ptr);
07009                               int function; /* SEND, RECEIVE o BOTH */
07010 int src_dest; /* origen del cual recibir o dest. p/enviar */
07011 message *m_ptr; /* apuntador a mensaje */
07012 {
07013 /* Las únicas llamadas al sist. en MINIX son enviar y recibir mensajes,
07014 * Y se efectúan entrando por trampa al kernel con una instrucción INT.

```

```

07015 * La trampa se atrapa y se invoca sys_call() p/enviar o recibir un mensaje
07016 * (o ambas cosas). Proc_ptr siempre indica quién llama.
07017 */
07018
07019 register struct proc *rp;
07020 int n;
07021
07022 1* Detectar parámetros de llamada al sistema erróneos. *1
07023 if (!isoksrc_dest(src_dest) return(E_BAD_SRC);
07024 rp = proc_ptr;
07025
07026 if (isuserp(rp) && function != BOTH) return(E_NO_PERM);
07027
07028 1* Los parámetros están bien, efectuar llamada. *1
07029 if (function & SEND) {
07030     1* Function = SEND o BOTH. */
07031     n = mini_send(rp, src_dest, m_ptr);
07032     if (function == SEND || n != OK)
07033         return(n); /* hecho, o falló SEND *1
07034 }
07035
07036 /* Function = RECEIVE o BOTH.
07037 * Comprobamos que las llamadas de usuario son 80TH, y confiamos en 'function' por lo demás.
07038 */
07039 return(mini_rec(rp, src_dest, m_ptr));
07040 }
07041
07042 1*=====
07043 * mini_send :
07044 *=====1
07045 PRIVATE int mini_send(caller_ptr, dest, m_ptr)
07046 register struct proc *caller_ptr; /* ¿quién trata de enviar mensaje? *
07047 int dest; /* ¿a quién se envía mensaje? *
07048 message *m_ptr; /* apunto a buffer de mensaje *
07049 {
07050     /* Enviar mensaje de 'caller_ptr' a 'dest'. Si 'dest' está bloqueado
07051     * esperando este mensaje, copiarlo en él y desbloquear 'dest'. Si 'dest'
07052     * no espera, o espera otro origen, formar en cola 'caller_ptr'.
07053     *
07054
07055     register struct proc *dest_ptr, *next_ptr;
07056     vir_bytes vb; /* apunto a bufo mensaje como vir_bytes *1
07057     vir_clicks vlo, vhi; /* clics virtuales con mensaje por enviar °1
07058
07059     * Procs. de usuario sólo pueden enviar a FS y MM. Comprobar esto. *
07060     if (isuserp(caller_ptr) && !issysent(dest) return(E_BAD_DEST);
07061     dest_ptr = proc_addr(dest); /* apunto a entrada de proc. de dest. °1
07062     if (dest_ptr->p_flags & P_SLOT_FREE) return(E_BAD_DEST); 1° dest. muerto */
07063
07064     * Esta prueba permite que el mensaje esté en datos, pila o espacio
07065     * intermedio. Tendrá que hacerse más completa en máquinas
07066     * que no tienen el espacio mapeado.
07067     *
07068     vb = (vir_bytes) m_ptr;
07069     vlo = vb >> CLICK_SHIFT; 1* clic viro para base de mensaje *1
07070     vhi = (vb + MESS_SIZE - 1) >> CLICK_SHIFT; 1* clic viro para tope de mensaje
07071     if (vlo < caller_ptr->p_map[D].mem_vir || vlo > vhi ||
07072         vhi >= caller_ptr->p_map[S].mem_vir + caller_ptr->p_map[S].mem_len)
07073         return(EFAULT);
07074

```

```

07075  /* Detectar bloq. mortal si 'caller_ptr' y 'dest' se envían mutuamente. */
07076  if (dest_ptr->p_flags & SENDING) {
07077      next_ptr = proc_addr(dest_ptr->p_sendto);
07078      while (TRUE) {
07079          if (next_ptr == caller_ptr) return(ELOCKED);
07080          if (next_ptr->p_flags & SENDING)
07081              next_ptr = proc_addr(next_ptr->p_sendto);
07082      else
07083          break;
07084      }
07085  }
07086
07087  /* Ver si 'dest' está bloqueado esperando este mensaje. */
07088  if ( (dest_ptr->p_flags & (RECEIVING | SENDING)) == RECEIVING &&
07089      (dest_ptr->p_getfrom == ANY |
07090       dest_ptr->p_getfrom == proc_number(caller_ptr))) {
07091      /* El destino sí está esperando este mensaje. */
07092      CopYMeSS(proc_number(caller_ptr), caller_ptr, m_ptr, dest_ptr,
07093              dest_ptr->p_messbuf);
07094      dest_ptr->p_flags &= -RECEIVING;          /* desbloquear destino */
07095      if (dest_ptr->p_flags == 0) ready(dest_ptr);
07096  } else {
07097      /* El destino no espera. Bloqueo y formo en cola invocador. */
07098      caller_ptr->p_messbuf = m_ptr;
07099      if (caller_ptr->p_flags == 0) unready(caller_ptr);
07100      caller_ptr->p_flags |= SENDING;
07101      caller_ptr->p_sendto = destj
07102
07103      /* Proceso bloqueado ya. Ponerlo en la cola del destino. */
07104      if ( (next_ptr = dest_ptr->p_callerq) == NIL_PROC)
07105          dest_ptr->p_callerq = caller_ptr;
07106      else {
07107          while (next_ptr->p_sendlink != NIL_PROC)
07108              next_ptr = next_ptr->p_sendlink;
07109          next_ptr->p_sendlink = caller_ptr;
07110      }
07111      caller_ptr->p_sendlink = NIL_PROC;
07112      }
07113      return(OK);
07114  }
07116  /*=====
07117  *                               mini_rec
07118  *=====*/
07119  PRIVATE int mini_rec(caller_ptr, src, m_ptr)
07120  register struct proc *caller_ptr;          /* proc. trata de obtener mensaje */
07121  int src;                                  /* origen de mens. deseado (o ANY) */
07122  message *m_ptr;                          /* apunto a buffer de mensaje */
07123  {
07124      /* Un proceso o tarea quiere obtener un mensaje. Si hay uno en cola,
07125      * adquirirlo y desbloquear emisor. Si no hay mensaje del origen deseado,
07126      * bloquear invocador. No hay que comprobar validez de paráms. Las llamadas
07127      * de usuario siempre son sendrec(), y mini_send() ya comprobó. Se Confía
07128      * en las llamadas de las tareas, MM y FS.
07129      */
07130
07131      register struct proc *sender_ptr;
07132      register struct proc *previous_ptr;
07133
07134      /* Ver si ya está disponible un mensaje del origen deseado. */

```

```

07135 if (1 (caller_ptr->p_flags & SENDING)) {
07136     /* Verificar cola de invocador. */
07137     for (sender_ptr = caller_ptr->p_callerq; sender_ptr != NIL_PROC;
07138         previous_ptr = sender_ptr, sender_ptr = sender_ptr->p_sendlink) {
07139         if (src == ANY 11 src == proc_number(sender_ptr)) {
07140             /* Se encontró un mensaje aceptable. */
07141             CopyMess(proc_number(sender_ptr), sender_ptr,
07142                 sender_ptr->p_messbuf, caller_ptr, m_ptr);
07143             if (sender_ptr == caller_ptr->p_callerq)
07144                 caller_ptr->p_callerq = sender_ptr->p_sendlink;
07145             else
07146                 previous_ptr->p_sendlink = sender_ptr->p_sendlink;
07147             if («sender_ptr->p_flags &= -SENDING) == 0)
07148                 ready(sender_ptr);          /* desbloquear emisor */
07149             return(OK);
07150         }
07151     }
07152
07153     /* Detectar interrupción bloqueada. */
07154     if (caller_ptr->p_int_blocked && isrxhardware(src)) {
07155         m_ptr->m_source = HARDWARE;
07156         m_ptr->m_type = HARD_INT;
07157         caller_ptr->p_int_blocked = FALSE;
07158         return(OK);
07159     }
07160 }
07161
07162 /* No hay mensaje adecuado. Bloquear el proc. que intenta recibir. */
07163 caller_ptr->p_getfrom = src;
07164 caller_ptr->p_messbuf = m_ptr;
07165 if (caller_ptr->p_flags == 0) unready(caller_ptr);
07166 caller_ptr->p_flags 1= RECEIVING;
07167
07168 /* Si MM acaba de bloquearse y hay señales de kernel pendientes,
07169    * es el momento de informar de ellas al MM, pues podrá aceptar el mensaje.
07170    */
07171 if (sig_procs > 0 && proc_number(caller_ptr) == MM_PROC_NR && src == ANY)
07172     inform();
07173     return(OK);
07174 }
07175
07176 /*=====
07177 *                                     pick_proc                                     *
07178 *=====*/
07179 PRIVATE void pick_proc()
07180 {
07181     /* Decidir a quién ejecutar ahora. Se escoge un nuevo proceso fijando 'proc_ptr'.
07182     * Si se escoge un proc. de usuario nuevo (u ocioso), asentar lo en 'bill_ptr'
07183     * para que la tarea de reloj sepa a quién cobrar tiempo de sistema.
07184     */
07185
07186     register struct proc *rp;          /* proceso por ejecutar */
07187
07188     if ((rp = rdy_head[TASK_Q]) != NIL_PROC) {
07189         proc_ptr = rp;
07190         return;
07191     }
07192     if ((rp = rdy_head[SERVER_Q]) != NIL_PROC) {
07193         proc_ptr = rp;
07194         return;

```

```

07195     }
07196     if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
07197         proc_ptr = rp;
07198         bill_ptr = rp;
07199         return;
07200     }
07201     /* Nadie está listo. Ejecutar tarea ociosa. Ésta podría hacerse tarea
07202     * de usuario siempre lista para evitar este caso especial.
07203     */
07204     bill_ptr = proc_ptr = proc_addr(IDLE);
07205     }
07207 /*=====*
07208 *                                     ready                                     *
07209 *=====*/
07210 PRIVATE void ready(rp)
07211             register struct proc *rp;      /* este proc. ya es ejecutable */
07212     {
07213     /* Añadir 'rp' al final de una de las colas de procesos ejecutables.
07214     * Se mantienen tres colas:
07215     *   TASK_Q      (más alta prioridad) para tareas ejecutables
07216     *   SERVER_Q  -(prioridad media) sólo para MM y FS
07217     *   USER_Q    -(más baja prioridad) para procs. de usuario
07218     */
07219
07220     if (istaskp(rp)) {
07221         if (rdy_head[TASK_Q] != NIL_PROC)
07222             /* Agregar al final de cola no vacía. */
07223             rdy_tail[TASK_Q] ->p_nextready = rp;
07224         else {
07225             proc_ptr =                /* ejecutar tarea nueva ahora */
07226             rdy_head[TASK_Q] = rp;    /* agregar a cola vacía */
07227         }
07228         rdy_tail[TASK_Q] = rp;
07229         rp->p_nextready = NIL_PROC;    /* nueva entrada no tiene sucesor */
07230         return;
07231     }
07232     if (!isuserp(rp)) {                /* las demás son similares */
07233         if (rdy_head[SERVER_Q] != NIL_PROC)
07234             rdy_tail[SERVER_Q] ->p_nextready = rp;
07235         else
07236             rdy_head[SERVER_Q] = rp;
07237             rdy_tail[SERVER_Q] = rp;
07238             rp->p_nextready = NIL_PROC;
07239             return;
07240     }
07241     if (rdy_head[USER_Q] == NIL_PROC)
07242         rdy_tail[USER_Q] = rp;
07243         rp->p_nextready = rdy_head[USER_Q];
07244         rdy_head[USER_Q] = rp;
07245     /*
07246     if (rdy_head[USER_Q] != NIL_PROC)
07247         rdy_tail[USER_Q] ->p_nextready = rp;
07248     else
07249         rdy_head[USER_Q] = rp;
07250         rdy_tail[USER_Q] = rp;
07251         rp->p_nextready = NIL_PROC;
07252     */
07253     }

```



```

07255 /*=====
07256 *                               unready                               *
07257 *=====*/
07258 PRIVATE void unready(rp)
07259 register struct proc *rp;          /* este proc. ya no es ejecutable */
07260 {
07261 /* Un proceso se bloqueó. */
07262
07263 register struct proc *xp;
07264 register struct proc k*qtail;      /* TASK_Q, SERVER_Q o USER_Q rdy_tail */
07265
07266 if (istaskp(rp) {
07267     /* ¿pila de tareas aún ok? */
07268     if (*rp->p_stguard != STACK_GUARD)
07269         panic("stack overrun by task", proc_number(rp));
07270
07271     if ( (xp = rdy_head[TASK_Q]) == NIL_PROC) return;
07272     if (xp == rp) {
07273         /* Quitar cabeza de la cola */
07274         rdy_head[TASK_Q] = xp->p_nextready;
07275         if (rp == proc_ptr) pick_proc();
07276         return;
07277     }
07278     qtail = &rdy_tail[TASK_Q];
07279 }
07280 else if (!isuserp(rp) {
07281     if ( (xp = rdy_head[SERVER_Q]) == NIL_PROC) return;
07282     if (xp == rp) {
07283         rdy_head[SERVER_Q] = xp->p_nextready;
07284         pick_proc();
07285         return;
07286     }
07287     qtail = &rdy_tail[SERVER_Q];
07288 } else
07289 {
07290     if ( (xp = rdy_head[USER_Q]) == NIL_PROC) return;
07291     if (xp == rp) {
07292         rdy_head[USER_Q] = xp->p_nextready;
07293         pick_proc();
07294         return;
07295     }
07296     qtail = &rdy_tail[USER_Q];
07297 }
07298
07299 /* Examinar cuerpo de la cola. Puede hacerse que un proc. no esté listo
07300 * aunque no se esté ejecutando enviándole una señal que lo termine.
07301 */
07302 while (xp->p_nextready != rp)
07303     if ( (xp = xp->p_nextready) == NIL_PROC) return;
07304     xp->p_nextready = xp->p_nextready->p_nextready;
07305     if (*qtail == rp) *qtail = xp;
07306 }
07307
07308 /*=====
07309 *                               sched                               *
07310 *=====*/
07311 PRIVATE void sched()
07312 {
07313 /* El proc. actual se ejecutó demasiado tiempo. Si otro proc. de baja prioridad
07314 * (usuario) es ejecutable, colocar el actual al final de la cola de usuario,

```

```

07315 * tal vez promoviendo otro usuario a la cabeza de la cola.
07316 *
07317
07318 if (rdy_head[USER_Q] == NIL_PROC) return;
07319
07320 * Uno o más procs. de usuario en cola. *
07321 rdy_tail[US~R_Q]->p_nextready = rdy_head[USER_Q];
07322 rdy_tail[USER_Q] = rdy_head[USER_Q];
07323 rdy_head[USER_Q] = rdy_head[USER_Q] ->p_nextready;
07324 rdy_tail[USER_Q] ->p_nextready = NIL_PROC;
07325     pick_proc();
07326     }
07328 *=====
07329 *                               lock mini send                               *
07330 *=====*/
07331 PUBLIC int lock_mini_send(caller_ptr, dest, m_ptr);
07332 struct proc *caller_ptrj           * ¿quién trata de enviar mensaje? *
07333 int dest;                          * ¿a quién se envía el mensaje? *
07334 message *m_ptr;                    * apuntador al buffer de mensaje *
07335 {
07336 * Entrada segura a mini_send() para tareas. *j
07337
07338     int resultj
07339
07340     switching = TRUE;
07341     result = mini_send(caller_ptr, dest, m_ptr);
07342     switching = FALSE;
07343     return(result)j
07344 }
07346 /*=====
07347 *                               lock_pick_proc                               *
07348 *=====*/
07349 PUBLIC void lock_pick_proc()
07350 {
07351 * Entrada segura a pick_proc() para tareas. *
07352
07353     switching = TRUE;
07354     pick_proc();
07355     switching = FALSE;
07356     }
07358 *=====
07359 *                               lock_ready                               *
07360 *=====*/
07361 PUBLIC void lock_ready(rp)
07362 struct proc *rp;-                  * este proceso ya es ejecutable *
07363 {
07364 * Entrada segura a ready() para tareas. *
07365
07366     switching = TRUE;
07367     ready(rp)j
07368     switching = FALSE;
07369     }
07372 *=====
07373 *                               lock unready                               *
07374 *=====*/

```

```

07375 PUBLIC void lock_unready(rp)
07376 struct proc *rp;          /* este proc. ya no es ejecutable */
07377 {
07378     /* Entrada segura a unready() para tareas. */
07379
07380     switching = TRUE;
07381     unready(rp);
07382     switching = FALSE;
07383 }
07385 /*=====
07386 *                               lock_sched                               *
07387 *=====*/
07388     PUBLIC void lock_sched()
07389     {
07390         /* Entrada segura a sched() para tareas. */
07391
07392         switching = TRUE;
07393         sched();
07394         switching = FALSE;
07395     }
07397 /*=====
07398 *                               unhold                               *
07399 *=====*/
07400     PUBLIC void unhold()
07401     {
07402         /* Vaciar ints. detenidas. k reenter debe ser 0. held head no debe ser
07403         * NIL_PROC. Las ints. deben-estar inhabilitadas. Se habilitarán
07404         * pero cuando éstas regresen se inhabilitarán.
07405         */
07406
07407         register struct proc *rp;          /* cabeza actual de cola de detenidas */
07408
07409         if (switching) return;
07410         rp = held_head;
07411         do {
07412             if ( (held_head = rp->p_nextheld) == NIL_PROC) held_tail = NIL_PROC;
07413             rp->p_int_held = FALSE;
07414             unlock();          /* reducir latencia; cola detenidas podría cambiar! */
07415             interrupt(proc_number(rp));
07416             lock();          /* proteger otra vez cola de detenidas */
07417         }
07418         while ( (rp = held_head) != NIL_PROC );
07419     }

```

```

+++++
src/kernel/exception.c
+++++

07500      /* Este archivo contiene un manejador de excepciones sencillo. Las excepciones
07501 * en procesos de usuario se convierten en señales. Las excepciones en el kernel,
07502 * MM Y FS causan pánico.
07503  */
07504
07505      #include "kernel.h"
07506      #include <signal.h>
07507      #include "proc.h"
07508
07509 /*=====
07510 *                                  exception
07511 *=====*/
07512      PUBLIC void exception(vec_nr)
07513      unsigned vec_nrj
07514      {
07515          /* Ocurrió excepción o interrupción inesperada. */
07516
07517      struct ex_s {
07518          char *msg;
07519          int signum;
07520          int minprocessor;
07521      };
07522      static struct ex_s ex_datar[] = {
07523          "Divide error", SIGFPE, 86,
07524          "Debug exception", SIGTRAP, 86,
07525          "Nonmaskable interrupt", SIGBUS, 86,
07526          "Breakpoint", SIGEMT, 86,
07527          "Overflow", SIGFPE, 86,
07528          "Bounds check", SIGFPE, 186,
07529          "Invalid opcode", SIGILL, 186,
07530          "Coprocessor not available", SIGFPE, 186,
07531          "Double fault", SIGBUS, 286,
07532          "Coprocessor segment overrun", SIGSEGV, 286,
07533          "Invalid TSS", SIGSEGV, 286,
07534          "Segment not present", SIGSEGV, 286,
07535          "Stack exception", SIGSEGV, 286,          /* ya se usó STACK_FAULT */
07536          "General protection", SIGSEGV, 286,
07537          "page fault", SIGSEGV, 386,          /* no cerrar */
07538          NIL_PTR, SIGILL, 0,          /* quizá trampa de software */
07539          "Coprocessor error", SIGFPE, 386,
07540      };
07541      register struct ex_s *ep;
07542      struct proc *saved_pro;
07543
07544      saved_pro = proc_ptr; /* Guardar proc_ptr, porque podría modificarse
07545                          * por las instrucciones de depuración.
07546                          */
07547
07548      ep = &ex_data[vec_nr];
07549
07550      if (vec_nr == 2) {          /* NMI espuria en algunas máquinas */
07551          printf("got spurious NMI\n");
07552          return;
07553      }
07554

```

```

07555 if(k_reenter == 0 && isuserp(saved_proc)) {
07556     unlock();                               /* ésta se protege como sys_call() */
07557     cause_sig(prOc_number(saved_proc), ep->signum);
07558     return;
07559 }
07560
07561 /* No se supone que esto suceda. */
07562 if (ep->msg == NIL_PTR || processor < ep->minprocessor)
07563     printf("\nIntel-reserved exception %d\n", vec_nr);
07564     else
07565     printf("\n\n%s\n", ep->msg);
07566     printf("process number %d, pc = 0x%04x:0x%08x\n",
07567     proc_number(saved_proc) ,
07568     (unsigned) saved_proc->p_reg.cs,
07569     (unsigned) saved_proc->p_reg.pc);
07570     panic("exception in system code", NO_NUM);
07571 }

```

```

+++++

```

src/kernel/i8259.c

```

+++++

```

```

07600 /* Este arch. contiene rutinas para inicializar el cont. de ints. 8259:
07601 *   get_irq_handler: dirección de manejador para interrupción dada
07602 *   put_irq_handler: registrar un manejador de interrupciones
07603 *   intr_init: inicializar el o los controladores de interrupciones
07604 */
07605
07606 #include "kernel.h"
07607
07608 #define ICW1_AT          0x11    /* disparado p/flanco, cascada, nec. ICW4 */
07609 #define ICW1_PC          0x13    /* disp. p/flanco, no cascada, nec. ICW4 */
07610 #define ICW1_PS          0x19    /* disp. p/nivel, cascada, nec. ICW4 */
07611 #define ICW4_AT          0x01    /* no SFNM, no buffer, EOI normal, 8086 */
07612 #define ICW4_PC          0x09    /* no SFNM, buffer, EOI normal, 8086 */
07613
07614 FORWARD_PROTOTYPE( int spurious_irq, (int irq) );
07615
07616 #define set_vec(nr, addr)          ((void)0) /* kluge para modo protegido */
07617
07618 /*=====
07619 *                                     intr_init                                     *
07620 *=====*/
07621     PUBLIC void intr_init(mine)
07622     int mine;
07623     {
07624     /* Inicializar los 8259, terminando con todas ints. inhabilitadas.
07625     * Esto sólo se hace en modo protegido; en modo real no tocamos los 8259,
07626     * sino que usamos las posiciones de BIOS. Se iza "mine" si los 8259
07627     * se programarán para Minix o se restablecerán a lo que el BIOS espera.
07628     */
07629
07630     int i;
07631
07632     lock();
07633     /* Las AT y PS/2 más nuevas tienen dos controladores de ints., un maestro
07634     * Y un esclavo en IRQ2. (No tenemos que ocuparnos de la PC que

```

```

07635     * sólo tiene un controlador, porque debe ejecutarse en modo real.)
07636     */
07637 out_byte(INT_CTL, ps_mca? ICW1_PS : ICW1_AT);
07638 out_byte(INT_CTLMASK, mine? IRQ0_VECTOR : BIOS_IRQ0_VEC);
07639                                     /* ICW2 para maestro */
07640 out_byte(INT_CTLMASK, (1 « CASCADE_IRQ)); /* ICW3 informa a esclavos */
07641 out_byte(INT_CTLMASK, ICW4_AT);
07642 out_byte(INT_CTLMASK, -(1 « CASCADE_IRQ)); /* máscara IRQ 0-7 */
07643 out_byte(INT2_CTL, ps_mca? ICW1_PS : ICW1_AT);
07644 out_byte(INT2_CTLMASK, mine? IRQ8_VECTOR : BIOS_IRQ8_VEC);
07645                                     /* ICW2 para esclavo */
07646 out_byte(INT2_CTLMASK, CASCADE_IRQ); /* ICW3 es nr esclavo */
07647 out_byte(INT2_CTLMASK, ICW4_AT);
07648 out_byte(INT2_CTLMASK, -0); /* máscara IRQ 8-15 */
07649
07650 /* Inicializar tabla de manejadores de interrupciones. */
07651 for (i = 0; i < NR_IRQ_VECTORS; i++) irq_table[i] = spurious_irq;
07652     }
07654 /*=====
07655     *                               spurious_irq                               *
07656 *=====*/
07657     PRIVATE int spurious_irq(irq)
07658     int irq;
07659     {
07660     /* Manejador de interrupciones por omisión. Se queja mucho. */
07661
07662     if (irq < 0 || irq >= NR_IRQ_VECTORS)
07663         panic("invalid call to spurious_irq", irq);
07664
07665     printf("spurious irq %d\n", irq);
07666
07667     return (1); /* Rehabilitar interrupción */
07668     }
07670 /*=====
07671     *                               put_irq_handler                               *
07672 *=====*/
07673     PUBLIC void put_irq_handler(irq, handler)
07674     int irq;
07675     irq_handler_t handler;
07676     {
07677     /* Registrar un manejador de interrupciones. */
07678
07679     if (irq < 0 || irq >= NR_IRQ_VECTORS)
07680         panic("invalid call to put_irq_handler", irq);
07681
07682     if (irq_table[irq] == handler)
07683         return; /* inicialización extra */
07684
07685     if (irq_table[irq] != spurious_irq)
07686         panic("attempt to register second irq handler for irq", irq);
07687
07688     disable_irq(irq);
07689     if (!protected_mode) set_vec(BIOS_VECTOR(irq), irq_vec[irq]);
07690     irq_table[irq] = handler;
07691     irq_use |= 1 « irq;
07692     }

```

```

+++++
                                     src/kernel/protect.c
+++++
07700 /* Este archivo contiene código para inicializar el modo protegido,
07701  * los descriptores de segmentos de código y de datos, y los descriptores
07702  * globales para los descriptores locales en la tabla de procesos.
07703  */
07704
07705 #include "kernel.h"
07706 #include "proc.h"
07707 #include "protect.h"
07708
07709 #define INT_GATE_TYPE      (INT_286_GATE | DESC_386_BIT)
07710 #define TSS_TYPE          (AVL_286_TSS | DESC_386_BIT)
07711
07712 struct desctableptr_s {
07713     char limit[sizeof(u16_t)];
07714     char base[sizeof(u32_t)];                               /* realmente u24_t + pad para 286 */
07715 };
07716
07717 struct gatedesc_s {
07718     u16_t offset_low;
07719     u16_t selector;
07720     u8_t pad;                                             /* 1000\XXXXXI ig & trpg, I\XXXXXXXXXI task 9 */
07721     u8_t p_dpl_type;                                    /* IPIDLI0ITYPE */
07722     u16_t offset_high;
07723 };
07724
07725     struct tss_s {
07726     reg_t backlink;
07727     reg_t sp0;     /* apunto a pila para usar durante interrupción */
07728     reg_t ss0;     /* segmento de pila que usar durante interrupción */
07729     reg_t sp1;
07730     reg_t ss1;
07731     reg_t sp2;
07732     reg_t ss2;
07733     reg_t cr3;
07734     reg_t ip;
07735     reg_t flags;
07736     reg_t ax;
07737     reg_t cx;
07738     reg_t dx;
07739     reg_t bx;
07740     reg_t sp;
07741     reg_t bp;
07742     reg_t si;
07743     reg_t di;
07744     reg_t es;
07745     reg_t cs;
07746     reg_t ss;
07747     reg_t ds;
07748     reg_t fs;
07749     reg_t gs;
07750     reg_t ldt;
07751     u16_t trap;
07752     u16_t iobase;
07753     };
07754

```

```

07755 PUBLIC struct segdesc_s gdt[GDT_SIZE] ;
07756 PRIVATE struct gatedesc_s idt[IDT_SIZE];          /* init cero, así que no hay */
07757 PUBLIC struct tss_s tSSj                          /* init cero */
07758
07759 FORWARD _PROTOTYPE( void int_gate, (unsigned vec_nr, phys_bytes base,
07760                                unsigned dpl_type) );
07761 FORWARD _PRTOTYPE( void sdsc, (struct segdesc_s *segdp, phys_bytes base,
07762                                phys_bytes size) );
07763
07764 /*=====
07765 *                                prot_init                                *
07766 *=====*/
07767 PUBLIC void prot_init()
07768     {
07769 /* Establecer tablas para modo protegido.
07770 * Todas las ranuras GDT están alojadas en tiempo de compilación.
07771 */
07772
07773     phys_bytes code_bytes;
07774     phys_bytes data_bytes;
07775     struct gate_table_s *gtp;
07776     struct descptr_s *dtp;
07777     unsigned ldt_selector;
07778     register struct proc *rp;
07779
07780     static struct gate_table_s {
07781         _PROTOTYPE( void (*gate), (void) );
07782         unsigned char vec_nr;
07783         unsigned char privilege;
07784     }
07785     gate_table[]: {
07786         divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE.
07787         single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE,
07788         nmi, NMI_VECTOR, INTR_PRIVILEGE,
07789         breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE,
07790         overflow, OVERFLOW_VECTOR, USER_PRIVILEGE,
07791         bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE,
07792         inval_opcode, INVALID_OP_VECTOR, INTR_PRIVILEGE,
07793         copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE,
07794         double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE,
07795         copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE,
07796         inval_tss, 1     NVAL_TSS_VECTOR, INTR_PRIVILEGE,
07797         segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE,
07798         stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE,
07799         general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE,
07800         page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE,
07801         copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE,
07802         { hwint00, VECTOR( 0), INTR_PRIVILEGE };
07803         { hwint01, VECTOR( 1), INTR_PRIVILEGE };
07804         { hwint02, VECTOR( 2), INTR_PRIVILEGE };
07805         { hwint03, VECTOR( 3), INTR_PRIVILEGE };
07806         { hwint04, VECTOR( 4), INTR_PRIVILEGE };
07807         { hwint05, VECTOR( 5), INTR_PRIVILEGE };
07808         { hwint06, VECTOR( 6), INTR_PRIVILEGE };
07809         { hwint07, VECTOR( 7), INTR_PRIVILEGE };
07810         { hwint08, VECTOR( 8), INTR_PRIVILEGE };
07811         { hwint09, VECTOR( 9), INTR_PRIVILEGE };
07812         { hwint10, VECTOR(10), INTR_PRIVILEGE };
07813         { hwint11, VECTOR(11), INTR_PRIVILEGE };
07814         { hwint12, VECTOR(12), INTR_PRIVILEGE };

```



```

07815     { hwint13, VECTOR(13), INTR_PRIVILEGE };
07816     { hwint14, VECTOR(14), INTR_PRIVILEGE };
07817     { hwint15, VECTOR(15), INTR_PRIVILEGE };
07818     };
07819
07820 /* Ésta se invoca temprano y no puede usar tablas creadas por main(). */
07821 data_bytes = (phys_bytes) sizes[1] << CLICK_SHIFT;
07822 if (sizes[0] == 0)
07823     code_bytes = data_bytes;          /* I&D común */
07824 else
07825     code_bytes = (phys_bytes) sizes[0] << CLICK_SHIFT;
07826
07827 /* Construir apunts. gdt e idt en GDT donde el BIOS los espera. */
07828 dtp = (struct descstableptr_s *) &gdt[GDT_INDEX];
07829 * (u16_t *) dtp->limit = (sizeof gdt) - 1;
07830 * (u32_t *) dtp->base = vir2phys(gdt)
07831
07832 dtp = (struct descstableptr_s *) &gdt[IDT_INDEX];
07833 * (u16_t *) dtp->limit = (sizeof idt) - 1;
07834 * (u32_t *) dtp->base = vir2phys(idt);
07835
07836 /* Construir descriptores de segm. para tareas y manejo de interrupciones. */
07837 init_codeseg(&gdt[CS_INDEX], code_base, code_bytes, INTR_PRIVILEGE);
07838 init_dataseg(&gdt[DS_INDEX], data_base, data_bytes, INTR_PRIVILEGE);
07839 init_dataseg(&gdt[ES_INDEX], 0L, 0L, TASK_PRIVILEGE);
07840
07841 /* Construir descriptores desechables para funciones en klib88. */
07842 init_dataseg(&gdt[DS_286_INDEX], (phys_bytes) 0,
07843             (phys_bytes) MAX_286_SEG_SIZE, TASK_PRIVILEGE);
07844 init_dataseg(&gdt[ES_286_INDEX], (phys_bytes) 0,
07845             (phys_bytes) MAX_286_SEG_SIZE, TASK_PRIVILEGE);
07846
07847 /* Construir descriptores locales en GDT para LDT en tabla de proc.
07848  * Los LDT se asignan durante la compilación en la tabla de procesos
07849  * Y se inicializan cuando se inicializa o modifica el mapa de un proceso.
07850  */
07851 for (rp = BEG_PROC_ADDR, ldt_selector = FIRST_LDT_INDEX * DESC_SIZE;
07852      rp < END_PROC_ADDR; ++rp, ldt_selector += DESC_SIZE) {
07853     init_dataseg("gdt[ldt_selector / DESC_SIZE], vir2phys(rp->p_ldt),
07854                (phys_bytes) sizeof rp->p_ldt, INTR_PRIVILEGE);
07855     gdt[ldt_selector / DESC_SIZE].access ~ PRESENT | LDT;
07856     rp->p_ldt_sel = ldt_selector;
07857 }
07858
07859 /* Construir TSS principal.
07860  * Esto se usa sólo para registrar el apuntador a pila que se usará
07861  * después de una interrupción.
07862  * El apuntador se prepara de modo que una interrupción guarde
07863  * automáticamente los registros del proceso actual ip:cs:f:sp:ss
07864  * en la tabla de procesos.
07865  */
07866 tss.ss0 = DS_SELECTOR;
07867 init_dataseg(&gdt[TSS_INDEX], vir2phys(&tss), (phys_bytes) sizeof tss,
07868             INTR_PRIVILEGE);
07869 gdt[TSS_INDEX].access = PRESENT | (INTR_PRIVILEGE << DPL_SHIFT) | TSS_TVP;
07870 tss.iobase = sizeof tss;          /* mapa de permisos de e/s vacío */
07871
07872 /* Construir descriptores para puertas de interrupción en IDT. */
07873 for (gtp = &gate_table[0];
07874      gtp < &gate_table[sizeof gate_table / sizeof gate_table[0]]; ++gtp) {

```

```

07875         int_gate(gtp->vec_nr, (phys_bytes) (vir_bytes) gtp->gate,
07876                 PRESENT I INT_GATE_TYPE I (gtp->privilege « DPL_SHIFT));
07877     }
07878     int_gate(SYS_VECTOR, (phys_bytes) (vir_bytes) p_s_call,
07879             PRESENT I (USER_PRIVILEGE « DPL_SHIFT) I INT_GATE_TYPE);
07880     int_gate(LEVEL0_VECTOR, (phys_bytes) (vir_bytes) leve10_call,
07881             .PRESENT I (TASK_PRIVILEGE « DPL_SHIFT) I INT_GATE_TYPE);
07882     int_gate(SYS386_VECTOR, (phys_bytes) (vir_bytes) s_call,
07883             PRESENT I (USER_PRIVILEGE « DPL_SHIFT) I INT_GATE_TYPE);
07884     }
07886 /*=====
07887 *                                     init_codeseg                                     *
07888 *=====*/
07889     PUBLIC void init_codeseg(segdp, base, size, privilege)
07890     register struct segdesc_s *segdp;
07891     phys_bytes base;
07892     phys_bytes size;
07893     int privilege;
07894     {
07895         /* Construir descriptor de un segmento de código. */
07896
07897         sdesc(segdp, base, size);
07898         segdp->access = (privilege « DPL_SHIFT)
07899             I (PRESENT I SEGMENT I EXECUTABLE I READABLE);
07900         /* CONFORMING = 0, ACCESSED = 0 */
07901     }
07903 /*=====
07904 *                                     init_dataseg                                     *
07905 *=====*/
07906     PUBLIC void init_dataseg(segdp, base, size, privilege)
07907     register struct segdesc_s *segdp;
07908     phys_bytes base;
07909     phys_bytes size;
07910     int privilege;
07911     {
07912     /* Construir descriptor de un segmento de datos. */
07913
07914     sdesc(segdp, base, size);
07915     segdp->access = (privilege « DPL_SHIFT) I (PRESENT I SEGMENT I WRITEABLE);
07916     /* EXECUTABLE = 0, EXPAND_DOWN = 0, ACCESSED = 0 */
07917     }
07919 /*=====
07920 *                                     sdesc                                         *
07921 *=====*/
07922     PRIVATE void sdesc(segdp, base, size)
07923     register struct segdesc_s *segdp;
07924     phys_bytes base;
07925     phys_bytes size;
07926     {
07927         /* Llenar campos de tamaño (base, límite y granularidad) de un descriptor. */
07928
07929         segdp->base_low = base;
07930         segdp->base_middle = base » BASE_MIDDLE_SHIFT;
07931         segdp->base_high = base » BASE_HIGH_SHIFT;
07932         --size; /* convertir a límite, tamaño 0 implica 4G */
07933         if (size > BYTE_GRAN_MAX) {
07934             segdp->limit_low = size » PAGE_GRAN_SHIFT;

```

```

07935     segdp->granularity = GRANULAR 1 (size »
07936                                     (PAGE_GRAN_SHIFT + GRANULARITY_SHIFT));
07937     } else {
07938     segdp->limit_low = size;
07939     segdp->granularity = size »      GRANULARITY_SHIFTj
07940     }
07941     ,segdp->granularity 1= DEFAULTj      /* Significa BIG p/seg. datos */
07942     }
07944 /*=====*
07945 *                      seg2phys                      *
07946 *=====*/
07947 PUBLIC phys_bytes seg2phys(seg)
07948 U16_t seg;
07949 {
07950 /* Devolver la dirección base de un segmento; seg es un registro
07951 * de segmento 8086 o bien un se lector de segmento 286/386.
07952 */
07953 phys_bytes basej
07954 struct segdesc_s *segdpj
07955
07956 if (!protected_mode) {
07957     base = hclick_to_physb(seg);
07958 } else {
07959     segdp = &gdt[seg »      3]j
07960     base = segdp->base_low 1 ((u32_t) segdp->base_middle «      16);
07961     base 1= ((u32_t) segdp->base_high «      24)j
07962     }
07963 return base;
07964 }
07966 /*=====*
07967 *                      int_gate                      *
07968 *=====*/
07969 PRIVATE void int_gate(vec_nr, base, dpl_type)
07970 unsigned vec_nr;
07971 phys_bytes base;
07972 unsigned dpl_type;
07973 {
07974 /* Construir descriptor para puerta de interrupción. */
07975
07976 register struct gatedesc_s *idp;
07977
07978 idp = &idt[vec_nr];
07979 idp->offset_low = base;
07980 idp->selector = CS_SELECTOR;
07981 idp->p_dpl_type = dpl_type;
07982 idp->offset_high = base »      OFFSET_HIGH_SHIFT;
07983     }
07985 /*=====*
07986 *                      enable_iop                    *
07987 *=====*/
07&88 PUBLIC void enable_iop(pp)
07989 struct proc *pp;
07990 {
07991     /* Permitir a un proc. usuario usar instruc. de E/S. Cambiar los bits de permiso
07992 * de E/S en psw. Éstos especifican el nivel de permiso actual (CPL)
07993 * menos privilegiado que puede ejecutar instruc. de E/S. Usuarios y servidores
07994 * tienen CPL 3, que es el más bajo. Kernel tiene CPL 0, tareas CPL 1.

```



```
07995     */
07996     pp->p_reg.psw l= 0x3000;
07997     }
```

+++++
src/kernel/klib.s

```
08000 #
08001     /*! Escoge entre versiones 8086 y 386 de cód. de kernel de bajo nivel.*/
08002
08003     #include <minix/config.h>
08004     #if _WDRD_SIZE == 2
08005     #include "klib88.s"
08006     #else
08007     #include "klib386.s"
08008     #endif
```

+++++
src/kernel/klib386.s

```
08100
08101     /* sections*/
08102
08103     .sect .text; .sect .rom; .sect .data; .sect .bss
08104
08105     #include <minix/config.h>
08106     #include <minix/const.h>
08107     #include "const.h"
08108     #include "sconst.h"
08109     #include "protect.h"
08110
08111     /* Este archivo contiene varias rutinas de utilería en código ensamblador*/
08112     /* requeridas por el kernel. Son:*/
08113
08114     #define _monitor! salir de Minix y volver al monitor
08115     #define _check_mem      /* revisar bloque memoria, devolver tamaño válido */
08116     #define _cp_mess       /* copia mensajes de origen a destino */
08117     #define _exit          /* ficticia para rutinas de biblioteca */
08118     #define --exit        /* ficticia para rutinas de biblioteca */
08119     #define ---exit       /* ficticia para rutinas de biblioteca */
08120     #define ---main       /* ficticia para GCC */
08121     #define _in_byte      /* leer byte de un puerto y devolverlo */
08122     #define in_word       /* leer palabra de un puerto y devolverla */
08123     #define _out_byte     /* escribir un byte en un puerto */
08124     #define _out_word     /* escribir una palabra en un puerto */
08125     #define _port_read    /* transf. datos de puerto (cont. disco) a memoria */
08126     #define _port_read_byte /* lo mismo byte por byte */
08127     #define _port_write   /* transf. datos de memoria a puerto (cont. disco) */
08128     #define _port_write_byte /* lo mismo byte por byte */
08129     #define _lock        /* inhabilitar interrupciones */
08130     #define _unlock      /* habilitar interrupciones */
08131     #define _enable_irq   /* habilitar irq en controlador 8259 */
08132     #define _disable_irq  /* inhabilitar un irq */
08133     #define _phys_copy    /* copiar datos dentro de la memoria */
08134     #define _mem_rdw      /* copiar palabra de [segmento:distancia;] */
```

```

08135 #define _reset          /* restablecer sistema */
08136 #define _mem_vid_copy   /* copiar datos en ram de video */
08137 #define _vid_vid_copy   /* mover datos en ram de video */
08138 #define leve10          /* invocar función en nivel 0 */
08139
08140 /* Las rutinas sólo garantizan preservar los registros que el compilador
08141 * , de C espera se preserven (ebx, esi, edi, ebp, esp, registros de segmento
08142 /* Y bit de dirección en las banderas).
08143
08144 /* variables importadas*/
08145
08146 .sect      .bss
08147 .extern _mon_return, _mon_sp
08148 .extern _irq_use
08149 .extern _blank_color
08150 .extern _ext_memsize
08151 .extern _gdt
08152 .extern _low_memsize
08153 .extern _sizes
08154 .extern _vid_seg
08155 .extern _vid_size
08156 .extern _vid_mask
08157 .extern leve10_func
08158
08159 .sect      .text
08160 /*=====
08161 *                               monitor
08162 *=====*/
08163 PUBLIC void monitor();
08164 Regresar al monitor.
08165
08166 monitor:
08167 -mov     eax, (_reboot_code)      /* dir. de nuevos parámetros */
08168 mov     esp, (_mon_sp)           /* restaurar apunto a pila del monitor */
08169 016 mov dx, SS_SELECTOR          /* segmento de datos del monitor */
08170 mov     ds, dx
08171 mov     es, dx
08172 mov     fs, dx
08173 mov     gs, dx
08174 mov     ss, dx
08175 pop     edi
08176 pop     esi
08177 pop     ebp
08178 016 retf                        /* regresar al monitor*/
08179
08180 /*=====
08181 *                               check_mem
08182 *=====*/
08183 /*=====
08184 PUBLIC phys_bytes check_mem(phys_bytes base, phys_bytes size);
08185 /* Revisar bloque de memoria, devolver la cantidad válida.
08186 * Sólo se revisa cada 160. bloque.
08187 * Un tamaño inicial de 0 significa todo.
08188 /* Esto realmente debería verificar algunos alias.
08189
08190 CM_DENSITY = 16
08191 CM_LOG_DENSITY = 4
08192 TEST1PATTERN = 0x55 /* patrón de prueba de memoria 1; */
08193 TEST2PATTERN = 0xAA /* patrón de prueba de memoria 2; */
08194

```

```

08195   CHKM_ARGS   =      4 + 4 + 4      /* 4 + 4 */
08196                      ds ebx eip      /* tamaño base */
08197
08198 _check_mem:
08199     push    ebx
08200     push    ds
08201     016 mov  ax, FLAT_DS_SELECTOR
08202     mov     ds, ax
08203     mov     eax, CHKM_ARGS(esp)
08204     mov     ebx, eax
08205     mov     ecx, CHKM_ARGS+4(esp)
08206     shr    ecx, CM_LOG_DENSITY
08207 cm_loop:
08208     movb   dl, TEST1PATTERN
08209     xchgb  dl, (eax)                /* escribo pat. prueba. recordar original*/
08210     xchgb  dl, (eax)                /* restaurar original, leer pat. Prueba */
08211     cmpb   dl, TEST1PATTERN        /* coincide si memoria real buena */
08212     jnz    cm_exit                  /* si diferente, memoria inutilizable */
08213     movb   dl, TEST2PATTERN
08214     xchgb  dl, (eax)
08215     xchgb  dl, (eax)
08216     add   eax, CM_DENSITY
08217     cmpb  dl, TEST2PATTERN
08218     loopz cm_loop
08219 cm_exit:
08220     sub   eax, ebx
08221     por   ds
08222     por   ebx
08223     ret
08224
08225
08226 /*=====
08227 *                                     cp_mess                                     *
08228 *=====*/
08229 PUBLIC void cp_mess(int src, phys_clicks src_clicks, vir_bytes src_offset,
08230 /*phys_clicks dst_clicks, vir_bytes dst_offset);
08231 * Copia rápida de un mensaje de cualquier lugar del espacio de dir.
08232 * a cualquier otro. También copia la dirección de origen que es parámetro
08233 * de la llamada en la 1a. palabra del registro de destino.
08234 *
08235 * El tamaño de mensaje "Msize" está en DWORDS (no bytes) y debe
08236 * ajustarse correctamente. Cambiar la definición de un mensaje en el archivo
08237 * de tipo y no cambiarlo aquí conduce a un desastre total.
08238 */
08239 CM_ARGS =      4 + 4 + 4 + 4 + 4      /* 4 + 4 + 4 + 4 + 4 */
08240           /* es      ds edi esi eip      proc scl sof del dof */
08241
08242     .align 16
08243 _cp_mess:
08244     cld
08245 push  esi
08246 push  edi
08247 push  ds
08248 push  es
08249
08250     mov   eax, FLAT_DS_SELECTOR
08251     mov   ds, ax
08252     mov   es, ax
08253
08254     mov   esi, CM_ARGS+4(esp)        /* clics origen */

```

```

08255     shl     esi, CLICK_SHIFT
08256     add     esi, CM_ARGS+4+4(esp)      /* distancia origen*/
08257     mov     edi, CM_ARGS+4+4+4(esp)    /* clics destino*/
08258     shl     edi, CLICK_SHIFT
08259     add     edi, CM_ARGS+4+4+4(esp)    /* distancia destino*/
08260
08261     mov     eax, CM_ARGS(esp)           /* núm. proc. de emisor*/
08262     stos    /* copiar # emisor en mensaje dest.*/
08263     add     esi, 4                       /* no copiar primera palabra */
08264     mov     ecx, Msize -1               /* la primera palabra no cuenta */
08265     rep
08266     movs    /* copiar el mensaje*/
08267
08268     pop     es
08269     pop     ds
08270     pop     edi
08271     pop     esi
08272     ret                                     /* eso es todo amigos*/
08273
08274
08275 /*=====*
08276 *                                     exit                                     *
08277 *=====*/
08278     PUBLIC void exit();
08279     /* Algunas rutinas de bibl. usan exit, así que necesitamos versión ficticia.
08280     * No puede haber llamadas reales a exit en el kernel.
08281     * GNU CC gusta de llamar --main desde main() por razones no obvias.
08282
08283     _exit:
08284     --_exit:
08285     _exit:
08286     sti
08287     jmp    _exit
08288
08289     main:
08290     ret
08291
08292
08293 *=====*
08294 *                                     in_byte                                     *
08295 *=====*/
08296     PUBLIC unsigned in_byte(port_t port);
08297     Leer byte (sin signo) de puerto de E/S port y devolverlo.
08298
08299     .align 16
08300     in_byte:
08301     mov     edx, 4(esp)                   ! puerto
08302     sub     eax, eax
08303     inb    dx                             1 leer 1 byte
08304     ret
08305
08306
08307 *=====*
08308 *                                     in_word                                     *
08309 *=====*/
08310     PUBLIC unsigned in_word(port_t port);
08311     1 Leer palabra (sin signo) de puerto de E/S port y devolverla. 08312
08313     .align 16
08314     _in_word:

```



```

08315     mov     edx, 4(esp)           ! puerto
08316     sub     eax, eax
08317     016 in  dx                    1 leer una palabra
08318     ret
08319
08320
08321 *=====
08322*                                out_byte                                *
08323 *=====
08324     PUBLIC void out_byte(port_t port, u8_t value);
08325     /* Escribir valor (mutado a byte) en el puerto de E/S port.*/
08326
08327     .align   16
08328     out_byte:
08329     mov     edx, 4(esp)           /* puerto*/
08330     movb   al, 4+4(esp)         /* valor */
08331     outb   dx                    /* 1 byte a la salida */
08332     ret
08333
08334
08335 /*=====
08336*                                out_word                                *
08337 *=====
08338     PUBLIC void out_word(Port_t port, UI6_t value);
08339     /* Escribir valor (mutado a palabra) en el puerto de E/S port.*/
08340
08341     .align   16
08342     out_word:
08343     mov     edx, 4(esp)           /* puerto*/
08344     mov     eax, 4+4(esp)        /* valor*/
08345     016 outdx /* 1 palabra a la salida*/
08346     ret
08347
08348
08349 *=====
08350*                                port_read                                *
08351 *=====
08352     PUBLIC void port_read(port_t port, phys_bytes destination, unsigned bytcount);
08353     /* Transferir datos de puerto (controlador disco duro) a memoria.*
08354
08355     PR_ARGS =          4 + 4 + 4      /* 4 + 4 + 4*/
08356                   /*es edi eip      port dst len*/
08357
08358     .align   16
08359     _port_read:
08360     cld
08361     push   edi
08362     push   es
08363     mov   ecx, FLAT_DS_SELECTOR
08364     mov   es, cx
08365     mov   edx, PR_ARGS(esp)        /* puerto del cual leer*/
08366     mov   edi, PR_ARGS+4(esp)      /* dir. de destino*/
08367     mov   ecx, PR_ARGS+4+4(esp)    /* cuenta de bytes*/
08368     shr   ecx, 1                   /* cuenta de palabras*/
08369     rep   /* (hardware no maneja palo dobles) */
08370     016 ins                        /* leer todo */
08371     pop   es
08372     pop   edi
08373     ret
08374

```

```

08375
08376 /*=====
08377 *                                     port_read_byte                                     *
08378 /*=====
08379     PUBLIC void port_read_byte(port_t port, phys_bytes destination,
08380     .unsigned bytcount);
08381     /* Transferir datos de puerto a memoria.*/
08382
08383     PR_ARGS_B =    4 + 4 + 4                                     /* 4 + 4 + 4*/
08384 /*es edi eip      port dst len*/
08385
08386 _ port_read_byte:
08387     cld
08388         push    edi
08389         push    es
08390         mov     ecx, FLAT_DS_SELECTOR
08391         mov     es, cx
08392         mov     edx, PR_ARGS_B(esp)
08393         mov     edi, PR_ARGS_B+4(esp)
08394         mov     ecx, PR_ARGS_B+4+4(esp)
08395     rep
08396     insb
08397         pop     es
08398         pop     edi
08399     ret
08400
08401
08402 /*=====
08403 *                                     port_write                                     *
08404 /*=====
08405     PUBLIC void port_write(port_t port, phys_bytes source, unsigned bytcount);
08406     /* Transferir datos de memoria a puerto (controlador disco duro).*/
08407
08408     PW_ARGS =    4 + 4 + 4                                     /* 4 + 4 + 4*/
08409 /*              es edi eip      port src len */
08410
08411         .align    16
08412 _ port_write:
08413     cld 08414     push    es;
08415         push    ds
08416         mov     ecx, FLAT_DS_SELECTOR
08417         mov     ds, cx
08418         mov     edx, PW_ARGS(esp)                               /* puerto al cual escribir*/
08419         mov     esi, PW_ARGS+4(esp)                             /* dir. de origen*/
08420         mov     ecx, PW_ARGS+4+4(esp)                           /* cuenta de bytes*/
08421         shr     ecx, 1                                           /* cuenta de palabras*/
08422         rep                                           /* (hardware no maneja palo dobles)*/
08423         016 outs                                           /* escribir todo*/
08424         pop     ds
08425         pop     esi
08426         ret
08427
08428
08429 /*=====
08430 *                                     port_write_byte                             *
08431 /*=====
08432     PUBLIC void port_write_byte(port_t port, phys_bytes source,
08433     unsigned bytcount);
08434     /* Transferir datos de memoria a puerto.*/

```

```

08435
08436     PW_ARGS_B =          4 + 4 + 4      /* 4 + 4 + 4 */
08437     /*                  es edi eip      port src len */
08438
08439 _port_write_byte:
08440     cld
08441     .push  esi
08442     push  ds
08443     mov   ecx, FLAT_DS_SELECTOR
08444     mov   ds, cx
08445     mov   edx, PW_ARGS_B(esp)
08446     mov   esi, PW_ARGS_B+4(esp)
08447     mov   ecx, PW_ARGS_B+4+4(esp)
08448     rep
08449     outsb
08450     pop   ds
08451     pop   esi
08452     ret
08453
08454
08455 /*=====*
08456 *                               lock                               *
08457 *=====*
08458     PUBLIC void lock();
08459     /* Inhabilitar interrupciones de CPU.*/
08460
08461     .align 16
08462 _lock:
08463     cli                               /* inhabilitar interrupciones*/
08464     ret
08465
08466
08467 /*=====*
08468 *                               unlock                              *
08469 *=====*
08470     PUBLIC void unlock();
08471     /* Habilitar interrupciones de CPU.*/
08472
08473     .align 16
08474 _unlock:
08475     sti
08476     ret
08477
08478
08479 /*=====*
08480 *                               enable_irq                          *
08481 *=====*/
08482     PUBLIC void enable_irq(unsigned irq);
08483     /* Habilitar linea de solic. de interrupción apagando un bit de 8259.
08484     * Código equivalente para irq < 8:
08485     out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) & -(1 «irq));
08486
08487     .align 16
08488 _enable_irq:
08489     mov   ecx, 4(esp)                /* irq*/
08490     pushf
08491     cli
08492     movb ah,-1
08493     rolb ah,cl                       /* ah = -(1 « (irq % 8)»
08494     cmpb cl, 8

```

```

08495     jae     enable_8           ! habil. irq >= 8 en 8259 esclavo
08496 enable_0:
08497     inb     INT_CTLMASK
08498     andb    al, ah
08499     outb    INT_CTLMASK       ! apagar bit en 8259 maestro
08500     popf
08501     ret
08502     .align  4
08503 enable_8:
08504     inb     INT2_CTLMASK
08505     andb    al, ah
08506     outb    INT2_CTLMASK       ! apagar bit en 8259 esclavo
08507     popf
08508     ret
08509
08510
08511 /*=====
08512 *                               disable_irq                               *
08513 *=====*/
08514     PUBLIC int disable_irq(unsigned irq)
08515     /* Habilitar línea de solicitud de int. activando un bit del 8259.
08516     * Código equivalente para irq < 8:
08517     out_byte (INT_CTLMASK, in_byte(INT_CTLMASK) | (1 <<  irq));
08518     /* Devuelve true si la interrupción no estaba ya inhabilitada.*/
08519
08520     .align  16
08521 _disable_irq:
08522     mov     ecx, 4(esp)         /* irq*/
08523     pushf
08524     cli
08525     movb   ah, 1
08526     rolb  ah, cl                /* ah = (1 <<  (irq % 8))*/
08527     cmpb  cl, 8
08528     jae   disable_8           /* inhabil. irq >= 8 en 8259 esclavo
08529 disable_0:
08530     inb     INT_CTLMASK
08531     testb   al, ah
08532     jnz    dis_already        /* ¿ya inhabilitada?*/
08533     orb    al, ah
08534     outb    INT_CTLMASK       /* activar bit en 8259 maestro*/
08535     popf
08536     mov     eax, 1            /* inhabil. por esta función*/
08537     ret
08538 disable_8:
08539     inb     INT2_CTLMASK
08540     testb   al, ah
08541     jnz    dis_already        /* ¿ya inhabilitada?*/
08542     orb    al, ah
08543     outb    INT2_CTLMASK       /* activar bit en 8259 esclavo*/
08544     popf
08545     mov     eax, 1            /* inhabil. por esta función*/
08546     ret
08547 dis_already:
08548     popf
08549     xor     eax, eax         /* ya in habilitada*/
08550     ret
08551
08552
08553 !*=====
08554 *                               phys_copy                               *

```

```

08555 *=====*/
08556 /* PUBLIC void phys_copy(phys_bytes source, phys_bytes destination,
08557 /*Phys_bytes bytecount);*/
08558 /* Copiar un bloque de memoria fisica.*/
08559
08560 PC_ARGS = 4 + 4 + 4 + 4 /* 4 + 4 + 4*/
08561 /*.es edi esi eip src dst len */
08562
08563 .align 16
08564 _phys_copy:
08565 cld
08566 push esi
08567 push edi
08568 push es
08569
08570 mov eax, FLAT_DS_SELECTOR
08571 mov es, ax
08572
08573 mov esi, PC_ARGS(esp)
08574 mov edi, PC_ARGS+4(esp)
08575 mov eax, PC_ARGS+4+4(esp)
08576
08577 cmp eax, 10 /* evitar gasto alineac. p/cuentas peq.*/
08578 jb pc_small
08579 mov ecx, esi /* alinear origen, ojalá objetivo tambo*/
08580 neg ecx
08581 and ecx, 3 /* cuenta p/alineación */
08582 sub eax, ecx
08583 rep
08584 eseg movsb
08585 mov ecx, eax
08586 shr ecx, 2 /* cuenta de palabras dobles*/
08587 rep
08588 eseg movs
08589 and eax, 3
08590 pc_small:
08591 xchg ecx, eax /* residuo*/
08592 rep
08593 eseg movsb
08594
08595 pop es
08596 pop edi
08597 pop esi
08598 ret
08599
08600
08601 *=====*
08602 * mem_rdw *
08603 *=====*
08604 /* PUBLIC u16_t mem_rdw(U16_t segment, u16_t *offset);*/
08605 /* Cargar y devolver palabra en apunto lejano segmento:distancia.*/
08606
08607 .align 16
08608 _mem_rdw:
08609 mov cx, ds
08610 mov ds, 4(esp) /* segmento*/
08611 mov eax, 4+4(esp) /* distancia */
08612 movzx eax, (eax) /* palabra por devolver*/
08613 mov ds, cx
08614 ret

```

```

08615
08616
08617 /*=====*
08618 /*                      reset                      */ *
08619 /*=====*/
08620 PUBLIC void reset();
08621' /* Restablecer sistema cargando IDT con distancia 0 e interrumpiendo.*/
08622
08623 _ reset:
08624     lidt     (idt_zero)
08625     int     3          ! todo vale; al 386 no le gustará
08626 .sect     .data
08627     idt_zero: .data4 0, 0
08628 .sect     .text
08629
08630
08631 /*=====*
08632 *                      mem_vid_copy                */ *
08633 /*=====*/
08634 PUBLIC void mem_vid_copy(u16 *src, unsigned dst, unsigned count);
08635 /*
08636  * Copiar count caracteres de memoria kernel a memoria video. Src, dst
08637  * Y count son distancias y cuentas de video basadas en caracteres (pal.).
08638  * Si src es null se borra la memo pantalla llenándola con blank_color. */
08639
08640     MVC_ARGS    =      4 + 4 + 4 + 4      /* 4 + 4 + 4*/
08641     es edi esi eip      src dst ct
08642
08643 _ mem_vid_copy:
08644     push     esi
08645     push     edi
08646     push     es
08647     mov     esi, MVC_ARGS(esp)      /* origen*/
08648     mov     edi, MVC_ARGS+4(esp)    /* destino */
08649     mov     edx, MVC_ARGS+4+4(esp)  /* cuenta */
08650     mov     es, (_vid_seg)         /* destino es segm. Video */
08651     cld                               /* asegurar direc. hacia arriba */
08652 mvc_loop:
08653     and     edi, (_vid_mask)        /* continuar dir. al principio*/
08654     mov     ecx, edx                /* un trozo por copiar */
08655     mov     eax, (_vid_size)
08656     sub     eax, edi
08657     cmp     ecx, eax
08658     jbe     0f
08659     mov     ecx, eax                /* ecx = min(ecx, vid_size .edi) */
08660 0:   sub     edx, ecx                /* count = ecx */
08661     shl     edi, 1                  /* dirección de byte */
08662     test    esi, esi                /* origen == 0 indica borrar pantalla */
08663     jz     mvc_blank
08664 mvc_copy:
08665     rep                               /* copiar palabras en memo video*/
08666     016 movs
08667     jmp     mvc_test
08668 mvc_blank:
08669     mov     eax, (_blank_color)     /* ax = carácter p/borrar */
08670     rep                               /* copiar blancos en memo video */
08671     016 stos
08672     !jmp     mvc_test
08673 mvc_test:
08674     shr     edi, 1                  /* direcciones de palabras */

```

```

08675     test     edx, edx
08676     jnz     mvc_loop
08677     mvc_done:
08678     pop     es
08679     pop     edi
08680     pop     esi
08681     .ret
08682
08683
08684 /*=====*
08685 *                               vid_vid_copy                               *
08686 *=====*/
08687     PUBLIC void vid_vid_copy(unsigned src, unsigned dst, unsigned count);
08688
08689     /* Copiar count caracteres de memo video a memo video. Manejar traslapo.
08690     * Sirve para despl. pantalla, insertar o elim. líneas o caro Src, dst y count
08691     * son distancias y cuentas de video basadas en caracteres (pal.)*/
08692
08693     VVC_ARGS      =      4 + 4 + 4 + 4      /* 4 + 4 + 4 */
08694     -es edi esi eip      src dst ct
08695
08696     _vid_vid_copy:
08697     push     esi
08698     push     edi
08699     push     es
08700     mov     esi, VVC_ARGS(esp)      /* origen*/
08701     mov     edi, VVC_ARGS+4(esp)    /* destino */
08702     mov     edx, VVC_ARGS+4+4(esp)  /* cuenta */
08703     mov     es, (_vid_seg)          /* usar segmento de video */
08704     cmp     esi, edi                /* ¿copiar hacia arriba o abajo?*/
08705     jb     vvc_down
08706     vvc up:
08707     -cld                                /* dirección es hacia arriba*/
08708     vvc uploop:
08709     -andesi, (_vid_mask)              /* continuar dir. al principio */
08710     and     edi, (_vid_mask)
08711     mov     ecx, edx                /* un trozo por copiar */
08712     mov     eax, (_vid_size)
08713     sub     eax, esi
08714     cmp     ecx, eax
08715     jbe     0f
08716     mov     ecx, eax                /* ecx = min(ecx, vid_size -esi) */
08717     mov     eax, (_vid_size)
08718     sub     eax, edi
08719     cmp     ecx, eax
08720     jbe     0f
08721     mov     ecx, eax                /* ecx = min(ecx, vid_size -edi)*/
08722     sub     edx, ecx                /* count -= ecx */
08723     shl     esi, 1
08724     shl     edi, 1                /* direcciones de bytes */
08725     rep
08726     eseg 016 movs                    /* copiar palabras de video */
08727     shr     esi, 1
08728     shr     edi, 1                /* direcciones de palabras */
08729     test    edx, edx
08730     jnz     vvc_uploop              /* ¿otra vez?*/
08731     jmp     vvc_done
08732     vvc down:
08733     -std                                /* dirección es hacia abajo */
08734     lea     esi, -1 (esi) (edx*1)    /* comenzar a copiar arriba */

```

```

08735     lea     edi, -1 (edi)(edx*1)
08736     vvc downloop:
08737     -and    esi, (_vid_mask)          /* continuar dir. al principio */
08738     and     edi, (_vid_mask)
08739     mov     ecx, edx                  /* un trozo por copiar */
08740     lea     eax, 1      (esi)
08741     cmp     ecx, eax
08742     jbe     0f
08743     mov     ecx, eax                  /* ecx = min(ecx, esi + 1) */
08744 0:    lea     eax, 1 (edi)
08745     cmp     ecx, eax
08746     jbe     0f
08747     mov     ecx, eax                  /* ecx = min(ecx, edi + 1) */
08748 0:    sub     edx, ecx                  /* count -= ecx */
08749     shl     esi, 1
08750     shl     edi, 1                  /* direcciones de bytes */
08751     rep
08752     eseg 016 movs                    /* copiar palabras de video */
08753     shr     esi, 1
08754     shr     edi, 1                  /* direcciones de palabras */
08755     test    edx, edx
08756     jnz     vvc_downloop            /* ¿otra vez? */
08757     cld
08758     ljmp    vvc_done                /* compilador C espera arriba */
08759     vvc_done:
08760     pop     es
08761     pop     edi
08762     pop     esi
08763     ret
08764
08765
08766 /*=====
08767 *                                     level0                                     *
08768 *=====*/
08769     PUBLIC void level0(void (*func)(void)
08770     /* Llamar función en nivel de permiso 0. Permite a tareas de kernel
08771     * hacer cosas sólo posibles en el nivel más privilegiado de CPU.
08772     */
08773     _level0:
08774     mov     eax, 4(esp)
08775     mov     (_level0_func), eax
08776     int     LEVEL0_VECTOR
08777     ret
+++++
                                     src/kernel/misc.c
+++++
08800 /* Este archivo contiene una colección de procedimientos diversos:
08801 *     mem init:          inic. tablas de memo Algo de memo es informada por 8IOS,
08802 *     -algo se estima y verifica después
08803 *     env_parse         analizar variable de entorno.
08804 *     bad assertion     para depuración
08805 *     bad=compare       para depuración
08806 *
08807
08808 #include "kernel.h"
08809 #include "assert.h"

```



```

08810 #include <stdlib.h>
08811 #include <minix/com.h>
08812
08813 #define EM_BASE          0x100000L    /* base de memo extendida en AT */
08814 #define SHADOW_BASE 0xFA0000L    /* base de RAM sombra de ROM en algo AT */
08815 #define SHADOW_MAX      0x060000L    /* memo sombra utilizable máx. (lím. 16M) */
08816
08817 /*=====
08818 *                               mem init                               *
08819 *=====*/
08820 PUBLIC void mem_init()
08821 {
08822     /* Inic. tablas de tamaño de memoria. Esto se complica por fragmentación
08823      * Y diferentes estrato de acceso p/modo protegido. Debe haber un trozo en 0
08824      * suficiente para contener Minix prop. dicho. Para 286 y 386 puede haber memo
08825      * extendida (más de 1MB) que suele comenzar en 1MB, pero puede
08826      * haber otro trozo justo abajo de 16MB, reservada bajo DOS p/sombra
08827      * de ROM, pero disponible p/Minix si se puede re-mapear el hardware.
08828      * En modo protegido, la memo ext. está disponible si CLICK_SIZE
08829      * es lo bastante grande, pero se trata como memo ordinaria.
08830      */
08831
08832     u32_t ext_clicks;
08833     phys_clicks max_clicks;
08834
08835     /* Obtener tamaño de memoria ordinaria del BIOS. */
08836     mem[0].size = k_to_click(low_memsz);          /* base = 0 */
08837
08838     if (pc_at && protected_mode) {
08839         /* Obtener el tamaño de memo ext. del BIOS. Esto es especial excepto
08840          * en modo protegido, pero el modo prot. ahora es normal. No se puede
08841          * direccionar más de 16M en modo 286, así que hay que asegurarse
08842          * de que la dir. de memoria más alta quepa holgadamente si se cuenta
08843          * en clics.
08844          */
08845         ext_clicks = k_to_click((u32_t) ext_memsz);
08846         max_clicks = USHRT_MAX -(EM_BASE » CLICK_SHIFT);
08847         mem[1].size = MIN(ext_clicks, max_clicks);
08848         mem[1].base = EM_BASE » CLICK_SHIFT;
08849
08850         if (ext_memsz <= (unsigned) ((SHADOW_BASE -EM_BASE) / 1024)
08851             && check_mem (SHADOW_BASE , SHADOW_MAX) == SHADOW_MAX) {
08852             /* Memoria sombra de ROM. */
08853             mem[2].size = SHADOW_MAX » CLICK_SHIFT;
08854             mem[2].base = SHADOW_BASE » CLICK_SHIFT;
08855         }
08856     }
08857
08858     /* Memoria total del sistema. */
08859     tot_mem_size = mem[0].size + mem[1].size + mem[2].size;
08860 }
08861 /*=====
08862 *                               env_parse                               *
08863 *=====*/
08864 /*=====*/
08865 PUBLIC int env_parse(env, fmt, field, param, min, max)
08866 char *env;          /* variable de entorno que inspeccionar */
08867 char *fmt;          /* plantilla para analizarla */
08868 int field;          /* núm. campo del valor por devolver */
08869 long *param;        /* dir. del parámetro por obtener */

```

```

08870 long min, max;          /* valores mino y máx. para el parámetro */
08871 {
08872     /* Analizar variable de entorno, algo como "DPETH0=300:3". Pánico
08873 * si falla el análisis. Devolver EP_UNSET si la variable no está establecida,
08874 * EP_OFF si está en "off", EP_ON si-está en "on" o un campo está en
08875 * blanco, o EP_SET si se da un campo (devolver valor con *param).
08876 * Pued-n usarse comas y dos puntos en la cadena de entorno y formato,
08877 * puede haber campos vacíos en la cadena de entorno, y puede faltar
08878 * puntuación para saltarse campos. La cadena de formato contiene
08879 * caracteres 'd', 'o', 'x' y 'c' p/indicar que se usa 10, 8, 16 o 0
08880 * como último arg. de strtol.
08881 */
08882
08883 char *val, *end;
08884 long newpar;
08885 int i = 0, radix, r;
08886
08887 if ((val = k_getenv(env)) == NIL_PTR) return(EP_UNSET);
08888 if (strcmp(val, "off") == 0) return(EP_OFF);
08889 if (strcmp(val, "on") == 0) return(EP_ON);
08890
08891 r = EP_ON;
08892 for (;;) {
08893     while (*val == ' ') val++;
08894
08895     if (*val == 0) return(r);          /* punto de salida correcto */
08896
08897     if (*fmt == 0) break;            /* demasiados valores */
08898
08899     if (*val == ',' || *val == ':') {
08900         /* Momento de pasar a siguiente campo. */
08901         if (*fmt == ',' || *fmt == ':') i++;
08902         if (*fmt++ == *val) val++;
08903     } else {
08904         /* Entorno contiene un valor, obtenerlo. */
08905         switch (*fmt) {
08906             case 'd':          radix = 10;      break;
08907             case 'o':          radix = 010;     break;
08908             case 'x':          radix = 0x10;    break;
08909             case 'c':          radix = 0;       break;
08910             default:           goto badenv;
08911         }
08912         newpar = strtol(val, &end, radix);
08913
08914         if (end == val) break;          /* no es un número. */
08915         val = end;
08916
08917         if (i == field) {
08918             /* El campo solicitado. */
08919             if (newpar < min || newpar > max) break;
08920             *param = newpar;
08921             r = EP_SET;
08922         }
08923     }
08924 }
08925
08926 badenv:
08927 printf("Bad environment setting: '%s = %s'\n", env, k_getenv(env));
08928 panic("", NO_NUM);
08929 /*NOTREACHED*/
08929 }

```

```

08931 #if DEBUG
08932 /*=====
08933 *                                     bad_assertion *
08934 *=====*/
08935 PUBLIC void bad_assertion(file, line, what)
08936 char *file;
08937 int lj,ne;
08938 char *what;
08939 {
08940     printf("panic at %s(%d): assertion \"%s\" failed\n", file, line, what);
08941     panic(NULL, NO_NUM);
08942 }
08944 /*=====
08945 *                                     bad_compare *
08946 *=====*/
08947 PUBLIC void bad_compare(file, line, lhs, what, rhs)
08948 char *file;
08949 int line;
08950 int lhs;
08951 char *what;
08952 int rhs;
08953 {
08954     printf("panic at %s(%d): compare (%d) %s (%d) failed\n",
08955           file, line, lhs, what, rhs);
08956     panic(NULL, NO_NUM);
08957 }
08958 #endif /* DEBUG */
+++++
                                     src/kernel/driver.h
+++++
09000 /* Tipos y constantes compartidos entre el código de controlador de disp. genérico
09001 * Y dependiente del dispositivo.
09002 */
09003
09004 #include <minix/callnr.h>
09005 #include <minix/com.h>
09006 #include "proc.h"
09007 #include <minix/partition.h>
09008
09009 /* Info. y puntos de entrada al código dependiente del dispositivo. */
09010 struct driver {
09011     _PROTOTYPE( char *(*dr_name), (void) );
09012     _PROTOTYPE( int (*dr_open), (struct driver *dp, message *m_ptr) );
09013     _PROTOTYPE( int (*dr_close), (struct driver *dp, message *m_ptr) );
09014     _PROTOTYPE( int (*dr_ioctl) , (struct driver *dp, message *m_ptr) );
09015     _PROTOTYPE( struct device *(*dr_prepare), (int device) );
09016     _PROTOTYPE( int (*dr_schedule) , (int proc_nr, struct iorequest_s *request) );
09017     _PROTOTYPE( int (*dr_finish), (void) );
09018     _PROTOTYPE( void (*dr_cleanup), (void) );
09019     _PROTOTYPE( void (*dr_geometry), (struct partition *entry) );
09020 };
09021
09022 #if (CHIP == INTEL)
09023
09024 /* Núm. bytes accesibles por DMA antes de taparse con frontera de 64K: */

```



```

09115 * ISCATTERED_IOI disp. I # proc. I solíc. I          I ap. iov I
09116 *
09117 * IOEV_IOCTL I disp. I # proc. I cód func I      I ap. buf I
09118 *
09119 *
09120 * Este archivo contiene un archivo de entrada:
09121 *
09122 *     driver_task:      invocado por entrada de tarea dependo disp.
09123 *
09124 *
09125 * Construido 92/04/02 por Kees J. Bot con viejo AT wini y cont. disquete.
09126 */
09127
09128 #include "kernel.h"
09129 #include <sys/ioctl.h>
09130 #include "driver.h"
09131
09132 #define BUF_EXTRA
09133
09134 /* Apartar espacio para variables. */
09135 PRIVATE u8_t buffer[(unsigned) 2 * DMA_BUF_SIZE + BUF_EXTRA];
09136 u8_t *tmp buf;          /* el buffer DMA después */
09137 phys_bytes tmp_phys;   /* dir. fis. de buffer DMA */
09138
09139 FORWARD_PROTOTYPE( void init_buffer, (void) );
09140
09141 /*=====
09142 *                               driver_task                               *
09143 *=====*/
09144 PUBLIC void driver_taSk(dp)
09145 struct driver *dp;      /* Puntos de entrada dependientes del disp. */
09146 {
09147     /* Programa principal de cualquier tarea de controlador de dispositivo. */
09148
09149     int r, caller, proc_nr;
09150     message mess;
09151
09152     init_buffer();      /* Obtener un buffer de DMA. */
09153
09154     /* Éste es el ciclo principal de la tarea de disco. Espera un mensaje,
09155      * lo ejecuta y envía una respuesta.
09156      */
09157
09158     while (TRUE) {
09159         /* Primero espera solicitud de leer o escribir bloque de disco. */
09160         receive(ANY, &mess);
09161
09162         caller = mess.m_source;
09163         proc_nr = mess.PROC_NR;
09164
09165         switch (caller) {
09166             case HARDWARE:
09167                 /* Interrupción remanente. */
09168                 continue;
09169             case FS PROC NR:
09170                 /* El único invocador legítimo. */
09171                 break;
09172             default:
09173                 printf("%s: got message from %d\n", (*dp->dr_name)(), caller);
09174                 continue;

```

```

09175     }
09176
09177     /* Ahora realizar el trabajo. */
09178     switch(mess.m_type) {
09179         case DEV_OPEN:      r = (*dp->dr_open)(dp, &mess); break;
09180         case DEV_CLOSE:    r = (*dp->dr_close) (dp, &mess); break;
09181         case DEV_IOCTL:    r = (*dp->dr_ioctl)(dp, &mess); break;
09182
09183         case DEV_READ:
09184         case DEV_WRITE:    r = do_rdwt(dp, &mess);                break;
09185
09186         case SCATTERED_IO: r = do_vrdwt(dp, &mess);                break;
09187         default:           r = EINVAL;                            break;
09188     }
09189
09190     /* Aseo de estado remanente. */
09191     (*dp->dr_cleanup) ();
09192
09193     /* Por último, preparar y enviar mensaje de respuesta. */
09194     mess.m_type = TASK_REPLY;
09195     mess.REP_PROC_NR = proc_nr;
09196
09197     mess.REP_STATUS = r;      /* # bytes transferidos o cód. error */
09198     send(caller, &mess);     /* enviar respuesta a invocador */
09199     }
09200 }
09202 /*=====
09203 *                               init_buffer                               *
09204 *=====*/
09205     PRIVATE void init_buffer()
09206     {
09207         /* Escoger buffer seguro para usar en transf. DMA. También
09208         * puede usarse para leer tablas de particiones, etc. Su dirección
09209         * absoluta es 'tmp_phys', la dirección normal es 'tmp_buf'.
09210         */
09211
09212         tmp_buf = buffer;
09213         tmp_phys = vir2phys(buffer);
09214
09215         if (tmp_phys == 0) panic("no DMA buffer", NO_NUM);
09216
09217         if (dma_bytes_left(tmp_phys) < DMA_BUF_SIZE) {
09218             /* 1a. mitad de bufo cruza un limo de 64K, no acceso por DMA ahí. */
09219             tmp_buf += DMA_BUF_SIZE;
09220             tmp_phys += DMA_BUF_SIZE;
09221         }
09222     }
09224 /*=====
09225 *                               do_rdwt                               *
09226 *=====*/
09227     PUBLIC int do_rdwt(dp, m_ptr)
09228     struct driver *dp;          /* ptos. entrada dependo del disp. */
09229     message *m_ptr;            /* apuntador p/leer o escribo mensaje */
09230     {
09231         /* Ejecutar una sola solicitud de leer o escribir. */
09232         struct iorequest_s ioreq;
09233         int r;
09234

```

```

09235  if (m_ptr->COUNT <= 0) return(EINVAL);
09236
09237  if («*dp->dr_prepare) (m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
09238
09239  ioreq.io_request = m_ptr->m_type;
09240  ioreq.io_buf = m_ptr->ADDRESS;
09241  ior~q.io_position = m_ptr->POSITION;
09242  ioreq.io_nbytes = m_ptr->COUNT;
09243
09244  r = (*dp->dr_schedule) (m_ptr->PROC_NR, &ioreq);
09245
09246  if (r == OK) (void) (*dp->dr_finish)();
09247
09248  r = ioreq.io_nbytes;
09249  return(r < 0 ? r : m_ptr->COUNT -r);
09250  }
09252 /*=====
09253  *                               do_vrdwt                               *
09254 *=====*/
09255 PUBLIC int do_vrdwt(dp, m_ptr)
09256 struct driver *dp;           /* ptas. entrada dependo del disp. */
09257 message *m_ptr;           /* apunto p/leer o escribo mensaje */
09258 {
09259     /* Obtener un vector de solicitudes de E/S. Atender solicitudes una por una.
09260     * Devolver la situación en el vector.
09261     */
09262
09263     struct iorequest_s *iop;
09264     static struct iorequest_s iovec[NR_IOREOS];
09265     phys_bytes iovec_physj;
09266     unsigned nr_requests;
09267     int requestj;
09268     int r;
09269     phys_bytes user_iovec_phys;
09270
09271     nr_requests = m_ptr->COUNT;
09272
09273     if (nr_requests > sizeof iovec / sizeof iovec[0])
09274         panic("FS passed toa big an 1/0 vector", nr_requests);
09275
09276     iovec_phys = vir2phys(iovec);
09277     user_iovec_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
09278         (vir_bytes) (nr_requests * sizeof iovec[0]));
09279
09280     if (user_iovec_phys == 0)
09281         panic("FS passed abad 1/0 vector", (int) m_ptr->ADDRESS);
09282
09283     phys_copy(user_iovec_phys, iovec_phys,
09284         (phys_bytes) nr_requests * sizeof iovec[0]);
09285
09286     if («*dp->dr_prepare) (m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
09287
09288     for (request = 0, iop = iovecj request < nr_requests; request++, iop++) {
09289         if («r = (*dp->dr_schedule) (m_ptr->PROC_NR, iop») != OK) break;
09290     }
09291
09292     if (r == OK) (void) (*dp->dr_finish) ();
09293
09294     phys_copy(iovec_phys, user_iovec_phys,

```

```

09295     (phys_bytes) nr_requests * sizeof iovec[0]);
09296     return(OK);
09297     }
09299 /*=====*/
09300 *                                     no_name                                     *
09301 *=====*/
09302 PUBLIC char *no_name()
09303     {
09304     /* Si no hay nombre específico para el dispositivo. */
09305
09306     return(tasktab[proc_number(proc_ptr) + NR_TASKS].name);
09307     }
09309 /*=====*/
09310 *                                     do_nop                                     *
09311 *=====*/
09312     PUBLIC int do_nop(dp, m_ptr)
09313     struct driver *dp;
09314     message *m_ptr;
09315     {
09316     /* Nada ahí, o nada que hacer. */
09317
09318     switch (m_ptr->m_type) {
09319     case DEV_OPEN:         return(ENODEV);
09320     case DEV_CLOSE:       return(OK);
09321     case DEV_IOCTL:       return(ENOTTY);
09322     default:               return(EIO);
09323     }
09324     }
09326 /*=====*/
09327 *                                     nop_finish                                 *
09328 *=====*/
09329     PUBLIC int nop_finish()
09330     {
09331     /* Nada que terminar, dp->dr_schedule hizo todo el trabajo. */
09332     return(OK);
09333     }
09335 /*=====*/
09336 *                                     nop cleanup                               *
09337 *=====*/
09338     PUBLIC void nop_cleanup()
09339     {
09340     /* Nada que asear. */
09341     }
09343 /*=====*/
09344 *                                     clock_mess                               *
09345 *=====*/
09346     PUBLIC void clock_mess(ticks, func)
09347     int ticks;             /* cuántos tics de reloj esperar */
09348     watchdog_t func;      /* función que llamar al vencer tiempo */
09349     {
09350     /* Enviar un mensaje a la tarea del reloj. */
09351
09352     message mess;
09353
09354     mess.m_type = SET_ALARM;

```



```

09355 mess.CLOCK_PROC_NR = proc_number(proc_ptr);
09356 mess.DELTA_TICKS = (long) ticksj;
09357 mess.FUNC_TO_CALL = (sighandler_t) func;
09358 sendrec(CLOCK, &mess);
09359 }

09361 /*=====
09362 *                                     do diocntl
09363 *=====*/
09364 PUBLIC int do_diocntl(dp, m_ptr)
09365 struct driver *dp;
09366 message *m_ptr;          /* apuntador a solicitud ioctl */
09367 {
09368     /* Ejecutar solicitud de establecer/obtener partición. */
09369     struct device *dv;
09370     phys_bytes user_phys, entry_phys;
09371     struct partition entry;
09372
09373     if (m_ptr->REQUEST != DIOCSETP && m_ptr->REQUEST != DIOCGETP) return(ENOTTY);
09374
09375     /* Decodificar parámetros del mensaje. */
09376     if ((dv = (*dp->dr_prepare)(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
09377
09378     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(entrY));
09379     if (user_phys == 0) return(EFAULT);
09380
09381     entry_phys = vir2phys(&entrY);
09382
09383     if (m_ptr->REQUEST == DIOCSETP) {
09384         /* Copiar sólo esta entrada de la tabla de particiones. */
09385         phys_copy(user_phys, entry_phys, (phys_bytes) sizeof(entr));
09386         dv->dv_base = entry.base;
09387         dv->dv_size = entry.size;
09388     } else {
09389         /* Devolver una entrada de tabla de partic. y geometría de la unidad. */
09390         entry.base = dv->dv_base;
09391         entry.size = dv->dv_size;
09392         (*dp->dr_geometry)(&entrY);
09393         phys_copy(entry_phys, user_phys, (phys_bytes) sizeof(entr));
09394     }
09395     return(OK);
09396 }

```

```

+++++
src/kernel/drvlib.h
+++++

```

```

09400                                     /* Definiciones de controlador de disp. IBM.      Autor: Kees J.
Bot
09401     *                                     Dic. 7, 1995
09402     */
09403
09404     #include <ibm/partition.h>
09405
09406     _PROTOTYPE( void partition, (struct driver *dr, int device, int style) );
09407
09408     /* Organización de la tabla de parámetros de BIOS. */
09409     #define bp_cylinders(t)    (* (u16_t *) (&(t)[0]))

```

```

09410             #define bp_heads(t)(* (u8_t *) (&(t)[2]))
09411             #define bp_reduced_wr(t) (* (u16_t *) (&(t)[3]))
09412             #define bp_precomp(t) (* (u16_t *) (&(t)[5]))
09413             #define bp_max_ecc(t) (* (u8_t *) (&(t) [7]))
09414             #define bp_ctlbyte(t)(* (u8_t *) (&(t)[8]))
09415             #define bp_landingzone(t) (* (u16_t *) (&(t)[12]))
09416             #define bp_sectors(t) (* (u8_t *) (&(t)[141]))
09417
09418             /* Diversos. */
09419             #define DEV_PER_DRIVE (1 + NR_PARTITIONS)
09420             #define MINOR_hd1a 128
09421             #define MINOR_fd0a (28<<2)
09422             #define P_FLOPPY 0
09423             #define P_PRIMARY 1
09424             #define P_SUB 2

```

```

+++++
src/kernel/drvlib.c
+++++

```

```

09500/* Funciones de utilería de cont. disp. IBM.                               Autor: Kees J. Bot
09501 *                                                                                               Dic. 7, 1995
09502 * Punto de entrada:
09503 * partition: dividir un disco según tabla(s) de particiones que tiene.
09504 */
09505
09506     #include "kernel.h"
09507     #include "driver.h"
09508     #include "drvlib.h"
09509
09510
09511     FORWARD_PROTOTYPE( void extpartition, (struct driver *dp, int extdev,
09512                                         unsigned long extbase) );
09513     FORWARD_PROTOTYPE( int get_part_table, (struct driver *dp, int device,
09514                                         unsigned long offset, struct part_entry *table) );
09515     FORWARD_PROTOTYPE( void sort, (struct part_entry *table) );
09516
09517
09518 /*=====
09519 *                               partition                               *
09520 *=====*/
09521     PUBLIC void partition(dp, device, style)
09522             struct driver *dp; /* ptos. entrada dependo del disp. */
09523             int device; /* dispositivo que dividir */
09524             int style; /* estilo de partic.: disquete, primario, sub. */
09525     {
09526         /* Esta rutina se llama en 1a. apertura p/inicializar tablas
09527 * de particiones de un disp. Se asegura de que cada partición
09528 * esté dentro de los límites del disp. Dependiendo del estilo de partic.,
09529 * estamos haciendo particiones de disquete, primarias o subpartic.
09530 * Sólo las primarias están ordenadas, porque se comparten
09531 * con otros sistemas operativos que esperan esto.
09532 */
09533     struct part_entry table[NR_PARTITIONS], *pe;
09534     int disk, par;

```

```

09535     struct device *dv;
09536         unsigned long base, limit, part_limit;
09537
09538         /* Obtener geometría del dispositivo por dividir. */
09539     if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV || dv->dv_size == 0) return;
09540     base = dv->dv_base >> SECTOR_SHIFT;
09541     limit = base + (dv->dv_size >>> SECTOR_SHIFT);
09542
09543         /* Leer la tabla de particiones del dispositivo. */
09544     if (!get_part_table(dp, device, 0L, table)) return;
09545
09546         /* Calcular el núm. de dispositivo de la 1a. partición. */
09547     switch (style) {
09548     case P_FLOPPY:
09549         device += MINOR_fd0a;
09550         break;
09551     case P_PRIMARY:
09552         sort(table); /* ordenar tabla de partic. primaria */
09553         device += 1;
09554         break;
09555     case P_SUB:
09556         disk = device / DEV_PER_DRIVE;
09557         par = device % DEV_PER_DRIVE - 1;
09558         device = MINOR_hd1a + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
09559     }
09560
09561     /* Encontrar un arreglo de dispositivos. */
09562     if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV) return;
09563
09564     /* Establecer geometría de particiones con base en la tabla. */
09565     for (par = 0; par < NR_PARTITIONS; par++, dv++) {
09566         /* Encoger partición pique quepa en dispositivo. */
09567         pe = &table[par];
09568         part_limit = pe->lowsec + pe->size;
09569         if (part_limit < pe->lowsec) part_limit = limit;
09570         if (part_limit > limit) part_limit = limit;
09571         if (pe->lowsec < base) pe->lowsec = base;
09572         if (part_limit < pe->lowsec) part_limit = pe->lowsec;
09573
09574         dv->dv_base = pe->lowsec << SECTOR_SHIFT;
09575         dv->dv_size = (part_limit - pe->lowsec) << SECTOR_SHIFT;
09576
09577         if (style == P_PRIMARY) {
09578             /* Cada partic. primaria de Minix puede subdividirse. */
09579             if (pe->sysind == MINIX_PART)
09580                 partition(dp, device + par, P_SUB);
09581
09582             /* Una partic. extendida tiene partic. lógicas. */
09583             if (pe->sysind == EXT_PART)
09584                 extpartition(dp, device + par, pe->lowsec);
09585         }
09586     }
09587 }
09590 /*=====
09591 *                                     extpartition                                     *
09592 *=====*/
09593 PRIVATE void extpartition(dp, extdev, extbase)
09594 struct driver *dp; /* ptos. entrada dependo del disp. */

```

```

09595             int extdev;          /* partición extendida que explorar */
09596             unsigned long extbase; /* disto sector de la base de la partic. ext. */
09597     {
09598         /* Las partic. extendidas no pueden ignorarse porque la gente gusta de mover
09599 * archivos entre partic. de DOS. Evite leer esto, no es divertido.
09600 */
09601     struct part_entry table[NR_PARTITIONS], *pe;
09602     int subdev, disk, par;
09603     struct device *dv;
09604     unsigned long offset, nextoffset;
09605
09606     disk = extdev / DEV_PER_DRIVE;
09607     par = extdev % DEV_PER_DRIVE -1;
09608     subdev = MINOR_hd1a + (disk * NR_PARTITIONS + par) * NR_PARTITIONS ;
09609
09610     offset = 0;
09611     do {
09612         if (!get_part_table(dp, extdev, offset, table)) return;
09613         sort(table);
09614
09615         /* La tabla debe contener una partic. lógica y opcionalmente otra partic.
09616 * extendida. (Es una lista enlazada.)
09617 */
09618         nextoffset = 0 ;
09619         for (par = 0 ; par < NR_PARTITIONS ; par++) {
09620             pe = &table[par];
09621             if (pe->sysind == EXT_PART) {
09622                 nextoffset = pe->lowsec;
09623             } else
09624             if (pe->sysind != NO_PART) {
09625                 if ((dv = (*dp->dr_prepare) (subdev)) == NIL_DEV) return;
09626
09627                 dv->dv_base = (extbase + offset
09628                             + pe->lowsec) << SECTOR_SHIFT;
09629                 dv->dv_size = pe->size << SECTOR_SHIFT;
09630
09631                 /* ¿No más dispositivos? */
09632                 if ((++subdev % NR_PARTITIONS == 0) return;
09633             }
09634         }
09635     } while ((offset = nextoffset) != 0);
09636 }
09639 /*=====
09640 *                               get_part_table                               *
09641 *=====*/
09642 PRIVATE int get_part_table(dp, device, offset, table)
09643     struct driver *dp;
09644     int device;
09645             unsigned long offset; /* disto de sector a la tabla */
09646             struct part_entry *table; /* cuatro entradas */
09647 {
09648     /* Leer la tabla de particiones del dispositivo, devolver true si no
09649 * hubo errores.
09650 */
09651     message mess ;
09652
09653     mess.DEVICE = device ;
09654     mess.POSITION = offset << SECTOR_SHIFT;

```

```

09655 mess.COUNT = SECTOR_SIZE;
09656 mess.ADDRESS = (char *) tmp_buf;
09657 mess.PROC_NR = proc_number(proc_ptr);
09658 mess.m_type = DEV_READ;
09659
09660 if (do_rdwt(dp, &mess) != SECTOR_SIZE) {
09661     printf("%s: can't read partition table\n", (*dp->dr_name()));
09662     return 0;
09663 }
09664 if (tmp_buf[510] != 0x55 11 tmp_buf[511] != 0xAA) {
09665     /* Tabla de particiones no válida. */
09666     return 0;
09667 }
09668 memcpy(table, (tmp_buf + PART_TABLE_OFF), NR_PARTITIONS * sizeof(table[0]));
09669 return 1;
09670 }

09673 /*=====
09674 *                               sort                               *
09675 *=====*/
09676     PRIVATE void sort(table)
09677     struct part_entry *table;
09678     {
09679     /* Ordenar una tabla de particiones. */
09680     struct part_entry *pe, tmp;
09681     int n = NR_PARTITIONS;
09682
09683     do {
09684         for (pe = table; pe < table + NR_PARTITIONS-1; pe++) {
09685             if (pe[0].sysind == NO_PART
09686                 || (pe[0].lowsec > pe[1].lowsec
09687                     && pe[1].sysind != NO_PART)) {
09688                 tmp = pe[0]; pe[0] = pe[1]; pe[1] = tmp;
09689             }
09690         }
09691     } while (--n > 0);
09692     }

+++++
src/kernel/memory.c
+++++
09700     /* Este archivo contiene la parte dependiente del dispositivo
09701 * de los controladores para los archivos especiales siguientes:
09702 *     /dev/null         -dispositivo nulo (sumidero de datos)
09703 *     /dev/mem          -memoria absoluta
09704 *     /dev/kmem         -memoria virtual del kernel
09705 *     /dev/ram          -disco en RAM
09706 *
09707 * El archivo contiene un punto de entrada:
09708 *
09709 *     mem_task: entrada principal cuando se inicia el sistema
09710 *
09711 * Cambios:
09712 *         20 abr 1992 por Kees J. Bot: división depend/indep del disp.
09713 *
09714 */

```

```

09715     #include "kernel.h"
09716     #include "driver.h"
09717     #include <sys/ioctl.h>
09718
09719         #define NR_RAMs    4           /* núm. de dispositivos tipo RAM */
09720
09721     PRIVATE struct ~evice m_geom[NR_RAMs]; /* Base y tamaño de cada disco RAM */
09722         PRIVATE int m_devicej /* dispositivo actual */
09723
09724     FORWARD_PROTOTYPE( struct device *m_prepare, (int device) );
09725     FORWARD_PROTOTYPE( int m_schedule, (int proc_nr, struct iorequest_s *iop) );
09726     FORWARD =PROTOTYPE( int m_do_open, (struct driver *dp, message *m_ptr) );
09727         FORWARD_PROTOTYPE( void m_init, (void) );
09728     FORWARD =PROTOTYPE( int m_ioctl, (struct driver *dp, message *m_ptr) );
09729     FORWARD_PROTOTYPE( void m_geometry, (struct partition *entry) );
09730
09731
09732     /* Puntos de entrada a este controlador. */
09733     PRIVATE struct driver m_dtab = {
09734     no_name,           /* nombre del dispositivo actual */
09735     m_do_open,        /* abrir o montar */
09736     do_nop,           /* nada al cerrar */
09737     m_prepare,        /* especificar geometría de disco en RAM */
09738     m_ioctl,          /* preparar para E/S en disp. secundario dado */
09739     m_schedule,       /* realizar E/S */
09740     nop_finish,       /* schedule hace el trabajo, no hace falta pensar */
09741     nop_cleanup,      /* no hay nada sucio */
09742     m_geometry,       /* "geometría" del dispositivo de memoria */
09743     };
09744
09745
09746 /*=====
09747 *                               mem task                               *
09748 *=====*/
09749     PUBLIC void mem_task()
09750     {
09751         m_init();
09752         driver_task(&m_dtab);
09753     }
09754
09755
09756 /*=====
09757 *                               m_prepare                               *
09758 *=====*/
09759     PRIVATE struct device *m_prepare(device)
09760     int device;
09761     {
09762         /* Preparar para E/S en dispositivo. */
09763
09764         if (device < 0 || device >= NR_RAMs) return(NIL_DEV) ;
09765         m_device = device;
09766
09767         return(&m_geom[device]);
09768     }
09769
09770
09771 /*=====
09772 *                               m schedule                               *
09773 *=====*/
09774     PRIVATE int m_schedule(proc_nr, iop)

```

```

09775 int proc_nr;                /* proceso que solicita */
09776     struct iorequest_s *iop;   /* apunto a solíc. de leer o escribir */
09777     {
09778     /* Leer o escribir /dev/null, /dev/mem, /dev/kmem o /dev/ram. */
09779
09780     int device, count, opcode ;
09781     phys_bytes mem_phys, user   _phys;
09782     struct device *dv;
09783
09784     /* Tipo de solicitud. */
09785     opcode = iop->io_request & OPTIONAL_IO;
09786
09787     /* Obtener núm. de disp. secundario y verificar /dev/null. */
09788     device = m_device;
09789     dv = &m_geom[device];
09790
09791     /* Determinar dirección de origen o destino de datos. */
09792     user_phys = numap(proc_nr, (vir_bytes) iop->io_buf,
09793                               (vir_bytes) iop->io_nbytes);
09794     if (user_phys == 0) return(iop->io_nbytes = EINVAL);
09795
09796     if (device == NULL_DEV) {
09797         /* /dev/null: Agujero negro. */
09798         if (opcode == DEV_WRITE) iop->io_nbytes = 0;
09799         count = 0;
09800     } else {
09801         /* /dev/mem, /dev/kmem o /dev/ram: Verificar EOF */
09802         if (iop->io_position >= dv->dv_size) return(OK);
09803         count = iop->io_nbytes;
09804         if (iop->io_position + count > dv->dv_size)
09805             count = dv->dv_size - iop->io_position;
09806     }
09807
09808     /* Preparar 'mem_phys' para /dev/mem, /dev/kmem o /dev/ram */
09809     mem_phys = dv->dv_base + iop->io_position;
09810
09811     /* Reservar por adelantado núm. de bytes por transferir. */
09812     iop->io_nbytes -= count;
09813
09814     if (count == 0) return(OK);
09815
09816     /* Copiar los datos. */
09817     if (opcode == DEV_READ)
09818         phys_copy(mem_phys, user_phys, (phys_bytes) count);
09819     else
09820         phys_copy(user_phys, mem_phys, (phys_bytes) count);
09821
09822     return(OK);
09823 }
09826 /*=====
09827 *                               m_do_open                               *
09828 *=====
09829 PRIVATE int m_do_open(dp, m_ptr)
09830 struct driver *dp;
09831 message *m_ptr;
09832 {
09833     /* Revisar núm. de dispositivo al abrir. Dar privilegios de E/S a un proceso
09834        * que abre /dev/mem o /dev/kmem.

```

```

09835     */
09836
09837 if (m_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
09838
09839 if (m_device == MEM_DEV || m_device == KMEM_DEV)
09840     enable_iop(proc_addr(m_ptr->PROC_NR));
09841
09842     return(OK);
09843 }
09844 /*=====
09845 *                                     m_init                                     *
09846 *=====*/
09847 PRIVATE void m_init()
09848 {
09849     /* Inicializar esta tarea. */
09850     extern int _end;
09851
09852     m_geom[KMEM_DEV].dv_base = vir2phys(0);
09853     m_geom[KMEM_DEV].dv_size = vir2phys(&_end);
09854
09855     #if (CHIP == INTEL)
09856     if (!protected_mode) {
09857         m_geom[MEM_DEV].dv_size = 0x100000; /* 1M p/sistemas 8086 */
09858     } else {
09859         #if _WORD_SIZE == 2
09860         m_geom[MEM_DEV].dv_size = 0x1000000; /* 16M p/sistemas 286 */
09861         #else
09862         m_geom[MEM_DEV].dv_size = 0xFFFFFFFF; /* 4G-1 para sistemas 386 */
09863         #endif
09864     }
09865     #endif
09866 }
09867 #endif
09868 }
09869
09870 /*=====
09871 *                                     m_ioctl                                     *
09872 *=====*/
09873 PRIVATE int m_ioctl(dp, m_ptr)
09874 struct driver *dp;
09875 message *m_ptr; /* apunto para leer o escribir mensaje */
09876 {
09877     /* Establecer parámetros para uno de los discos en RAM. */
09878
09879     unsigned long bytesize;
09880     unsigned base, size;
09881     struct memory *memp;
09882     static struct psinfo psinfo = { NR_TASKS, NR_PROCS, (vir_bytes) proc, 0, 0 };
09883     phys_bytes psinfo_phys;
09884
09885     switch (m_ptr->REQUEST) {
09886     case MIOCRAMSIZ:
09887         /* FS establece el tamaño del disco en RAM. */
09888         if (m_ptr->PROC_NR != FS_PROC_NR) return(EPERM);
09889
09890         bytesize = m_ptr->POSITION * 8LOCK_SIZE;
09891         size = (bytesize + CLICK_SHIFT-1) >> CLICK_SHIFT;
09892
09893         /* Encontrar trozo de memoria en el que quepa el disco en RAM. */

```



```

09895     memp= &mem[NR_MEMS];
09896     while («-memp) ->size < size) {
09897         if (memp == mem) panic("RAM disk is toa big", NO_NUM);
09898     }
09899     base = memp->base;
09900     memp->base += size;
09901     roemp->size -= size;
09902
09903     m_geom[RAM_DEV].dv_base = (unsigned long) base << CLICK_SHIFT;
09904     m_geOm[RAM_DEV] .dv_size = bytesize;
09905     break;
09906 case MIOCSPSINFO:
09907     /* MM o FS fijan la dirección de su tabla de procesos. */
09908     if (m_ptr->PROC_NR == MM_PROC_NR) {
09909         psinfo.mproc = (vir_bytes) m_ptr->ADDRESS;
09910     } else
09911     if (m_ptr->PROC_NR == FS_PROC_NR) {
09912         psinfo.fproc = (vir_bytes) m_ptr->ADDRESS;
09913     } else {
09914         return(EPERM);
09915     }
09916     break;
09917 case MIOCGPSINFO:
09918     /* El programa ps quiere direcciones de tabla de procesos. */
09919     psinfo_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
09920                        sizeof(psinfo));
09921     if (psinfo_phys == 0) return(EFAULT);
09922     phys_copy(vir2phys(&psinfo) , psinfo_phys, (phys_bytes) sizeof(psinfo));
09923     break;
09924 default:
09925     return(do_diocntl(&m_dtab, m_ptr));
09926 }
09927 return(OK);
09928 }

09931 /*=====
09932 *                               m_geometry                               *
09933 *=====*/
09934     PRIVATE void m_geometry(entry)
09935     struct partition *entry;
09936     {
09937     /* Disp. de memoria no tienen geometría, pero el mundo exterior insiste. */
09938     entry->cylinders = (m_geom[m_device] .dv_size » SECTOR_SHIFT) / (64 * 32);
09939     entry->heads = 64;
09940     entry->sectors = 32;
09941     }

```

```

+++++
                                     src/kernel/wini.c
+++++
10000                                     /*wini.c -escoger controlador winchester      Autor: Kees J.
Bot
10001 *
10002 * Se pueden compilar varios controladores winchester
10003 * diferentes en el kernel, pero sólo uno puede ejecutarse. Ése se
10004 * escoge aquí usando la variable de arranque 'hd'.
10005 */
10006
10007     #include "kernel.h"
10008     #include "driver.h"
10009
10010     #if ENABLE_WINI
10011
10012     /* Correspondencia entre nombre de controlador y función de tarea. */
10013     struct hdmmap {
10014 char          *name;
10015 task_t       *task;
10016     } hdmmap[] = {
10017
10018     #if ENABLE_AT_WINI
10019 {"at",        at_winchester_task      },
10020 #endif
10021
10022     #if ENABLE_BIOS_WINI
10023 {"bios",     bios_winchester_task    },
10024 #endif
10025
10026     #if ENABLE_ESDI_WINI
10027 {"esdi",     esdi_winchester_task    },
10028 #endif
10029
10030     #if ENABLE_XT_WINI
10031 {"xt",       xt_winchester_task      },
10032 #endif
10033 };
10034 };
10035
10036
10037 /*=====
10038 *                                     winchester_task
10039 *=====*/
10040     PUBLIC void winchester_task()
10041     {
10042     /* Invocar tarea winchester predeterminada o escogida. */
10043     char *hd;
10044     struct hdmmap *map;
10045
10046     hd = k_getenv("hd");
10047
10048     for (map = hdmmap; map < hdmmap + sizeof(hdmmap)/sizeof(hdmmap[0]); map++) {
10049         if (hd == NULL || strcmp(hd, map->name) == 0) {
10050             /* Ejecutar tarea winchester escogida. */
10051             (*map->task) ();
10052         }
10053     }
10054     panic("no hd driver", NO_NUM);

```

```
10055 }
10056 #endif /* ENABLE_W1N1 */
```

```
+++++
src/kernel/at_wini.c
+++++
```

```
10100 /* Este archivo contiene la parte dependiente del dispositivo del controlador
10101 * en soto para el controlador en har. winchester de IBM-AT.
10102 * Escrita por Adri Koppes.
10103 *
10104 * El archivo contiene un punto de entrada:
10105 *
10106 * at_winchester_task:          entrada princ. al iniciarse el sist.
10107 *
10108 *
10109 * Cambios:
10110 *   Abr 13, 1992, por Kees J. Bot: división depend./indep. del dispositivo.
10111 */
10112
10113 #include "kernel.h"
10114 #include "driver.h"
10115 #include "drvlib.h"
10116
10117 #if ENABLE_AT_W1N1
10118
10119 /* Puertos de E/S empleados por controladores de disco winchester. */
10120
10121 /* Registros de lectura y escritura */
10122 #define REG_BASE0      0x1F0 /* reg. base de controlador en har. 0 */
10123 #define REG_BASE1      0x170 /* reg. base de controlador en har. 1 */
10124 #define REG-DATA        0 /* reg. datos (distancia del reg. base) */
10125 #define REG=PRECOMP     1 /* inicio de precompensación de escrito */
10126 #define REG COUNT      2 /* sectores por transferir */
10127 #define REG-SECTOR     3 /* número de sector */
10128 #define REG-CYL LO     4 /* byte bajo de número de cilindro */
10129 #define REG=CYL=H1     5 /* byte alto de número de cilindro */
10130 #define REG LDH        6 /* lba, unidad y cabeza */
10131 #define LDH_DEFAULT    0xA0 /* habilitar ECC 512 bytes/sector */
10132 #define LDH_LBA        0x40 /* Usar direccionamiento LBA */
10133 #define ldh_init(drive) (LDH_DEFAULT | ((drive) << 4))
10134
10135 /* Registros sólo de lectura */
10136 #define REG STATUS      7 /* situación */
10137 #define STATUS_BSY     0x80 /* controlador en har. ocupado */
10138 #define STATUS_RDY     0x40 /* unidad lista */
10139 #define STATUS_WF      0x20 /* falla de escritura */
10140 #define STATUS-SC      0x10 /* fin de búsqueda (obsoleto) */
10141 #define STATUS=DRO     0x08 /* solicitud transferencia datos */
10142 #define STATUS CRD     0x04 /* datos corregidos */
10143 #define STATUS=1DX     0x02 /* pulso índice */
10144 #define STATUS_ERR     0x01 /* error */
10145 #define REG_ERROR      1 /* código de error */
10146 #define ERROR_BB       0x80 /* bloque malo */
10147 #define ERROR ECC      0x40 /* bytes ecc malos */
10148 #define ERROR=1D      0x10 /* no se encontró id */
10149 #define ERROR_AC       0x04 /* comando abortado */
```

```

10150 #define ERROR_TK          0x02    /* error de pista cero */
10151 #define ERROR_DM          0x01    /* no hay marca de dir. de datos */
10152
10153 /* Registros sólo de escritura */
10154 #define REG_COMMAND      7    /* comando */
10155 #define CMD_IDLE         0x00    /* para w_command: unidad ociosa */
10156 #define CMD_RECALIBRATE  0x10    /* recalibrar unidad */
10157 #define CMD_READ         0x20    /* leer datos */
10158 #define CMD_WRITE        0x30    /* escribir datos */
10159 #define CMD_READVERIFY   0x40    /* verificar lectura */
10160 #define CMD_FORMAT       0x50    /* formatear pista */
10161 #define CMD_SEEK         0x70    /* buscar cilindro */
10162 #define CMD_DIAG         0x90    /* ejec. diagnóstico de disp. */
10163 #define CMD_SPECIFY      0x91    /* especificar parámetros */
10164 #define ATA_IDENTIFY     0xEC    /* identificar unidad */
10165 #define REG_CTL          0x206 /* registro de control */
10166 #define CTL_NORETRY      0x80    /* inhabil. reintento de acceso */
10167 #define CTL_NOECC        0x40    /* inhabil. reintento de ecc */
10168 #define CTL_EIGHTHEADS  0x08    /* más de 8 cabezas */
10169 #define CTL_RESET        0x04    /* restablecer controlador har. */
10170 #define CTL_INTDISABLE   0x02    /* inhabilitar interrupciones */
10171
10172 /* Líneas de solicitud de interrupción. */
10173 #define AT_IRQ0          14    /* núm. interrupción p/controlador 0 */
10174 #define AT_IRQ1          15    /* núm. interrupción p/controlador 1 */
10175
10176 /* Bloque de comando común */
10177 struct command {
10178     u8_t precomp; /* REG_PRECOMP, etc. */
10179     u8_t count;
10180     u8_t sector;
10181     u8_t cyl_lo;
10182     u8_t cyl_hi; 10183 u8_t ldh;
10184     u8_t command;
10185 };
10186
10187
10188 /* Códigos de error */
10189 #define ERR              (-1)    /* error general */
10190 #define ERR_BAD_SECTOR (-2) /* detección de bloque marcado malo */
10191
10192 /* Algunos contr. en har. no interrumpen, el reloj nos despertará. */
10193 #define WAKEUP           (32*HZ) /* unidad apagada 31 seg. máximo */
10194
10195 /* Diversos. */
10196 #define MAX_DRIVES      4    /* este contr. apoya 4 unidades (hd0-hd19) */
10197 #if _WORD_SIZE > 2
10198 #define MAX_SECS        256  /* contr. har. puede transf este # sectores */
10199 #else
10200 #define MAX_SECS        127  /* pero no a un proceso de 16 bits */
10201 #endif
10202 #define MAX_ERRORS      4    /* cuánto intentar rd/wt antes de desistir */
10203 #define NR_DEVICES      (MAX_DRIVES * DEV_PER_DRIVE)
10204 #define SUB_PER_DRIVE   (NR_PARTITIONS * NR_PARTITIONS)
10205 #define NR_SUBDEVS      (MAX_DRIVES * SUB_PER_DRIVE)
10206 #define TIMEOUT         32000 /* plazo de contr. en har. en ms */
10207 #define RECOVERYTIME    500  /* tiempo recup. contr. har. en ms */
10208 #define INITIALIZED     0x01  /* unidad inicializada */
10209 #define DEAF            0x02  /* contr. har. debe restablecerse */

```

```

10210 #define SMART          0x04 /* unidad reconoce comandos ATA */
10211
10212
10213 /* Variables. */
10214 PRIVATE struct wini { /* struct de unidad princ. 1 entrada/unidad */
10215     unsigned state; /* estado unidad: sorda, iniciada, muerta */
10216     unsigned base; /* reg. base del archivo de registros */
10217     unsigned irq; /* línea de solicitud de interrupción */
10218     unsigned lcyllinders; /* núm. lógico de cilindros (BIOS) */
10219     unsigned lheads; /* núm. lógico de cabezas */
10220     unsigned lsectors; /* núm. lógico de sectores/pista */
10221     unsigned pcyllinders; /* núm. fis. de cilindros (traducido) */
10222     unsigned pheads; /* núm. fis. de cabezas */
10223     unsigned psectors; /* núm. fis. de sectores/pista */
10224     unsigned ldhpref; /* 4 bytes altos del reg. LDH (cabeza) */
10225     unsigned precomp; /* cil. precompens. escritura /4*/
10226     unsigned max_count; /* máx. solicitudes p/esta unidad */
10227     unsigned open_ct; /* cuenta en uso */
10228     struct device part[DEV_PER_DRIVE]; /* partic. primarias: hd[0-4] */
10229     struct device subpart[SUB_PER_DRIVE]; /* subparticiones: hd[1-4][a-d] */
10230 } wini[MAX_DRIVES], *w_wn;
10231
10232 PRIVATE struct trans {
10233     struct iorequest_s *iopj /* pertenece a esta solicitud de E/S */
10234     unsigned long block; /* primer sector por transferir */
10235     unsigned count; /* cuenta de bytes */
10236     phys_bytes phys; /* dirección física del usuario */
10237     } wtrans[NR_IOREQS];
10238
10239 PRIVATE struct trans *W_tp; /* p/agregar solicitudes de transf. */
10240 PRIVATE unsigned w_count; /* núm. bytes por transferir */
10241 PRIVATE unsigned long w_nextblock; /* sigio bloque disco por transferir */
10242 PRIVATE int w_opcode; /* DEV_READ o DEV_WRITE */
10243 PRIVATE int w_command; /* comando en ejecución actual */
10244 PRIVATE int w_status; /* situac. después de interrupción */
10245 PRIVATE int w_drive; /* unidad seleccionada */
10246 PRIVATE struct device *w_dv; /* base y tamaño de dispositivo */
10247
10248 FORWARD_PROTOTYPE( void init_params, (void) );
10249 FORWARD_PROTOTYPE( int w_do_open, (struct driver *dp, message *m_ptr) );
10250 FORWARD_PROTOTYPE( struct device *w_prepare, (int device) );
10251 FORWARD_PROTOTYPE( int w_identify, (void) );
10252 FORWARD_PROTOTYPE( char *w_name, (void) );
10253 FORWARD_PROTOTYPE( int w_specify, (void) );
10254 FORWARD_PROTOTYPE( int w_schedule, (int proc_nr, struct iorequest_s *iop) );
10255 FORWARD_PROTOTYPE( int w_finish, (void) );
10256 FORWARD_PROTOTYPE( int com_out, (struct command *cmd) );
10257 FORWARD_PROTOTYPE( void w_need_reset, (void) );
10258 FORWARD_PROTOTYPE( int w_do_close, (struct driver *dp, message *m_ptr) );
10259 FORWARD_PROTOTYPE( int com_simple, (struct command *cmd) );
10260 FORWARD_PROTOTYPE( void w_timeout, (void) );
10261 FORWARD_PROTOTYPE( int w_reset, (void) );
10262 FORWARD_PROTOTYPE( int w_intr_wait, (void) );
10263 FORWARD_PROTOTYPE( int w_waitfor, (int mask, int value) );
10264 FORWARD_PROTOTYPE( int w_handler, (int irq) );
10265 FORWARD_PROTOTYPE( void w_geometry, (struct partition *entry) );
10266
10267 /* ciclo w_waitfor desenrollado una vez para agilizar. */
10268 #define waitfor(mask, value) \
10269     ((in_byte(w_wn->base + REG_STATUS) & mask) == value \

```

```

10270     || w_waitTor(mask, value))
10271
10272
10273 /* Entrada apunta a este controlador en software. */
10274 PRIVATE struct driver w_dtab = {
10275     w_name,           /* nombre del dispositivo actual */
10276     w_do_open,       /* solíc. abrir o montar, inicializar disp. */
10277     w_do_close,      /* liberar dispositivo */
10278     do_diocntl,      /* obtener o fijar geom. de 1 partición */
10279     w_prepare,       /* preparar para E/S en disp. seco dado */
10280     w_schedule,      /* precalcular cil., cabeza, sector, etc. */
10281     w_finish;        /* efectuar E/S */
10282     nop_cleanup,     /* nada que asear */
10283     w_geometry,      /* indicar geometría del disco */
10284     };
10285
10286 #if ENABLE_ATAPI
10287     #include "atapi.c" /* código extra p/ATAPI CD-ROM */
10288     #endif
10289
10290
10291 /*=====
10292 *                               at_winchester_task                               *
10293 *=====*/
10294 PUBLIC void at_winchester_task()
10295     {
10296     /* Fijar paráms. especiales de disco e invocar ciclo principal genérico. */
10297
10298     init_params();
10299
10300     driver_task(&w_dtab);
10301     }
10302
10303
10304 /*=====
10305 *                               init_params                               *
10306 *=====*/
10307 PRIVATE void init_params()
10308     {
10309     /* Esta rutina se llama al arrancar p/inicializar paráms. de la unidad. */
10310
10311     u16_t parv[2];
10312     unsigned int vector;
10313     int drive, nr_drives, i;
10314     struct wini *wn;
10315     u8_t params[16];
10316     phys_bytes param~phys = vir2phys(params);
10317
10318     /* Obtener núm. de unidades del área de datos de BIOS. */
10319     phys_coPy(0x475L, param_phys, 1L);
10320     if ((nr_drives = params[0]) > 2) nr_drives = 2;
10321
10322     for (drive = 0, wn = wini; drive < MAX_DRIVES; drive++, wn++) {
10323     if (drive < nr_drives) {
10324     /* Copiar el vector de parámetros de BIOS */
10325     vector = drive == 0 ? WINI_0_PARM_VEC : WINI_1_PARM_VEC;
10326     phys_copy(vector * 4L, vir2phys(parv), 4L);
10327
10328     /* Calcular dirección de los paráms. y copiarlos */
10329     phys_copy(hclick_to_Physb(parv[1]) + parv[0], param_phys, 16L);

```

```

10330
10331         /* Copiar paráms. en estructuras de la unidad */
10332         wn->lcyllinders = bp_cylinders(params);
10333         wn->lheads = bp_heads(params);
10334         wn->lsectors = bp_sectors(params);
10335         wn->precomp = bp_precomp(params) >> 2;
10336     }
10337     wn->ldhpref = ldh_init(drive);
10338     wn->max_count = MAX_SECS << SECTOR_SHIFT;
10339     if (drive < 2) {
10340         /* Controller 0. */
10341         wn->base = REG_BASE0;
10342         wn->irq = AT_IRQ0;
10343     } else {
10344         /* Controller 1. */
10345         wn->base = REG_BASE1;
10346         wn->irq = AT_IRQ1;
10347     }
10348 }
10349 }

10352 /*=====
10353 *                               w_do_open                               *
10354 *=====*/
10355 PRIVATE int w_do_open(dp, m_ptr)
10356     struct driver *dp;
10357     message *m_ptr;
10358     {
10359     /* Abrir dispositivo: Inic. controlador har. y leer tabla partic. */
10360
10361     int r;
10362     struct wini *wn;
10363     struct command cmd;
10364
10365     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
10366     wn = w_wn;
10367
10368     if (wn->state == 0) {
10369         /* Tratar de identificar el dispositivo. */
10370         if (w_identify() != OK) {
10371             printf("%s: probe failed\n", w_name());
10372             if (wn->state & DEAF) w_reset();
10373             wn->state = 0;
10374             return(ENXIO);
10375         }
10376     }
10377     if (wn->open_ct++ == 0) {
10378         /* Dividir el disco en particiones. */
10379         partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY);
10380     }
10381     return(OK);
10382 }

10385 /*=====
10386 *                               w_prepare                               *
10387 *=====*/
10388 PRIVATE struct device *w_prepare(device)
10389                                     int device;

```

```

10390 {
10391 /* Preparar para E/S en un dispositivo. */
10392
10393 /* Nada que transferir aún. */
10394 w_count = 0j
10395
10396 if (device < NR_DEVICES) { /* hd0, hd1, ...*/
10397     w_drive = device / DEV_PER_DRIVE; /* guardar núm. de unidad */
10398     w_wn = &wini[w_drive];
10399     w_dv = &w_wn->part[device % DEV_PER_DRIVE];
10400 } else
10401 if (<unsigned) (device -= MINOR_hd1a) < NR_SUBDEVS) { /* hd1a, hd1b, ...*/
10402     w_drive = device / SUB_PER_DRIVE;
10403     w_wn = &wini[w_drive];
10404     w_dv = &w_wn->subpart[device % SUB_PER_DRIVE];
10405 } else {
10406     return(NIL_DEV);
10407 }
10408 return(w_dv);
10409 }

10412 /*=====
10413 *                               w_identify                               *
10414 *=====*/
10415 PRIVATE int w_identify()
10416 {
10417 /* Averiguar si existe dispositivo, si es un disco AT viejo o una unidad ATA
10418 * más nueva, un dispositivo de medio removible, etc.
10419 */
10420
10421 struct wini *wn = w_wn;
10422 struct command cmd;
10423 char id_string[40];
10424 int i, r;
10425 unsigned long size;
10426 #define id_byte(n) (&tmp_buf[2 * (n)])
10427 #define id_word(n) (((u16_t) id_byte(n)[0] << 0) \
10428 | ((u16_t) id_byte(n)[1] << 8))
10429 #define id_longword(n) (((u32_t) id_byte(n)[0] << 0) \
10430 | ((u32_t) id_byte(n)[1] << 8) \
10431 | ((u32_t) id_byte(n)[2] << 16) \
10432 | ((u32_t) id_byte(n)[3] << 24))
10433
10434 /* Verificar si existe uno de los registros. */
10435 r = in_byte(wn->base + REG_CYL_LO);
10436 out_byte(wn->base + REG_CYL_LO, -r);
10437 if (in_byte(wn->base + REG_CYL_LO) == r) return(ERR);
10438
10439 /* Parece correctoj registrar IRQ e intentar comando de identificación ATA. */
10440 put_irq_handler(wn->irq, w_handler);
10441 enable_irq(wn->irq);
10442
10443 cmd.ldh = wn->ldhpref;
10444 cmd.command = ATA_IDENTIFY;
10445 if (com_simple(&cmd) == OK) {
10446     /* Es un dispositivo ATA. */
10447     wn->state |= SMART;
10448
10449     /* Información del dispositivo. */

```



```

10450     port_read(wn->base + REG_DATA, tmp_phys, SECTOR_SIZE);
10451
10452     /* ¿Por qué están intercambiados bytes de la cadena? */
10453     for (i = 0; i < 40; i++) id_string[i] = id_byte(27)[i^1];
10454
10455     /* Modo de traducción CHS preferido. */
10456     wn->pcylinders = id_word(1);
10457     wn->pheads = id_word(3);
10458     wn->psectors = id_word(6);
10459     size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
10460
10461     if ((id_byte(49)[1] & 0x02) && size > 512L*1024*2) {
10462         /* La unidad tiene capacidad LBA y tamaño suficiente para confiar
10463          * en que no causará un desastre.
10464          */
10465         wn->ldhpref 1= LDH_LBA;
10466         size = id_longword(60);
10467     }
10468
10469     if (wn->lcyinders == 0) {
10470         /* ¿No hay parámetros BIOS? Inventarlos. */
10471         wn->lcyinders = wn->pcylinders;
10472         wn->lheads = wn->pheads;
10473         wn->lsectors = wn->psectors;
10474         while (wn->lcyinders > 1024) {
10475             wn->lheads *= 2;
10476             wn->lcyinders /= 2;
10477         }
10478     }
10479 } else {
10480     /* No es dispositivo ATAJ no traducciones, no funciones especiales.
10481      * No tocarlo a menos que BIOS esté enterado.
10482      */
10483     if (wn->lcyinders == 0) return(ERR);          /* no hay paráms. BIOS */
10484     wn->pcylinders = wn->lcyinders;
10485     wn->pheads = wn->lheads;
10486     wn->psectors = wn->lsectors;
10487     size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
10488 }
10489 /* La diversión termina en 4 GB. */
10490 if (size > ((u32_t) -1) ; SECTOR_SIZE) size = ((u32_t) -1) ; SECTOR_SIZE;
10491
10492 /* Base y tamaño de toda la unidad. */
10493 wn->part[0] .dv_base = 0;
10494 wn->part[0] .dv_size = size <<     SECTOR_SHIFT;
10495
10496 if (w_specify() 1= OK && w_specify() != OK) return(ERR);
10497
10498 printf("%s: ", w_name());
10499 if (wn->state & SMART) {
10500     printf("%.40s\n", id_string);
10501 } else {
10502     printf("%0ux%0ux%0u\n", wn->pcylinders, wn->pheads, wn->psectors);
10503 }
10504     return(OK);
10505 }

10508 /*=====
10509 *                                     w_name                                     *

```

```

10510 *=====*/
10511                                     PRIVATE char *w_name()
10512 {
10513 /* Devolver un nombre para el dispositivo actual. */
10514 static char name[ ] = "at-hd15";
10515 unsigned device = w_drive * DEV_PER_DRIVE;
10516
10517 if (device < 10) {
10518     name[5] = '0' + device;
10519     name[6] = 0;
10520 } else {
10521     name[5] = '0' + device / 10;
10522     name[6] = '0' + device % 10;
10523 }
10524 return name;
10525 }

10528 *=====*/
10529 *                                     w_specify                                     *
10530 *=====*/
10531                                     PRIVATE int w_specify()
10532 {
10533 /* Inicializar unidad después del arranque o si es necesario restablecerla. */
10534
10535 struct wini *wn = w_wn;
10536 struct command cmd;
10537
10538 if ((wn->state & DEAF) && w_reset() != OK) return(ERR);
10539
10540 /* Especificar paráms.: precompensación, núm. cabezas y sectores. */
10541 cmd.precomp = wn->precomp;
10542 cmd.count = wn->psectors;
10543 cmd.ldh = w_wn->ldhpref 1 (wn->pheads -1);
10544 cmd.command = CMD_SPECIFY; /* Especificar algunos paráms. */
10545
10546 if (com_simple(&cmd) != OK) return(ERR);
10547
10548 if (! (wn->state & SMART)) {
10549     /* Calibrar un disco viejo. */
10550     cmd.sector = 0j
10551     cmd.cyl_lo = 0j
10552     cmd.cyl_hi = 0;
10553     cmd.ldh = w_wn->ldhpref;
10554     cmd.command = CMD_RECALIBRATE;
10555
10556     if (com_simple(&cmd) != OK) return(ERR);
10557 }
10558
10559 wn->state 1= INITIALIZED;
10560 return(OK);
10561 }

10564 *=====*/
10565 *                                     w schedule                                     *r
10566 *=====*/
10567 PRIVATE int w_schedule(proc_nr, iop)
10568 int proc_nr; /* proceso que solicita */
10569 struct iorequest_s *iop; /* apunto a solíc. de leer o escribir */

```

```

10570 {
10571 /* Reunir solicitudes de E/S en bloques consecutivos para poder leer/escribir
10572 * en un solo comando. (Hay suficiente tiempo para calcular la sigo solicitud
10573 * consecutiva mientras pasa un bloque no deseado.)
10574 */
10575 struct wini *wn = w_wn;
10576 int ~, opcode;
10577 unsigned long pos;
10578 unsigned nbytes, count;
10579 unsigned long block;
10580 phys_bytes user_phys;
10581
10582 /* Este número de bytes por leer/escribir */
10583 nbytes = iop->io_nbytes;
10584 if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
10585
10586 /* De/a esta posición en el dispositivo */
10587 pos = iop->io_position;
10588 if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
10589
10590 /* De/a esta dirección de usuario */
10591 user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
10592 if (user_phys == 0) return(iop->io_nbytes = EINVAL);
10593
10594 /* ¿Leer o escribir? */
10595 opcode = iop->io_request & -OPTIONAL_IO;
10596
10597 /* ¿Cuál bloque en disco y qué tan cerca de EOF? */
10598 if (pos >= w_dv->dv_size) return(OK); /* At EOF */
10599 if (pos + nbytes > w_dv->dv_size) nbytes = w_dv->dv_size - pos;
10600 block = (w_dv->dv_base + pos) >> SECTOR_SHIFT;
10601
10602 if (w_count > 0 && block != w_nextblock) {
10603 /* Esta nueva solicitud no puede encadenarse al trabajo */
10604 if ((r = w_finish()) != OK) return(r);
10605 }
10606
10607 /* El siguiente bloque consecutivo */
10608 w_nextblock = block + (nbytes >> SECTOR_SHIFT);
10609
10610 /* Mientras haya bytes "no planificados" en la solicitud: */
10611 do {
10612 count = nbytes;
10613
10614 if (w_count == wn->max_count) {
10615 /* La unidad no puede hacer más que max_count a la vez */
10616 if ((r = w_finish()) != OK) return(r);
10617 }
10618
10619 if (w_count + count > wn->max_count)
10620 count = wn->max_count - w_count;
10621
10622 if (w_count == 0) {
10623 /* La primera solicitud de una fila, inicializar. */
10624 w_opcode = opcode;
10625 w_tp = wtrans;
10626 }
10627
10628 /* Guardar parámetros de E/S */
10629 w_tp->iop = iop;

```

```

10630     w_tp->block = block;
10631     w_tp->count = count;
10632     w_tp->phys = user_phys;
10633
10634     /* Update counters */
10635     w_tp++;
10636     w_count += count;
10637     block += count >> SECTOR_SHIFT;
10638     user_phys += count;
10639     nbytes -= count;
10640 } while (nbytes > 0);
10641
10642     return(OK);
10643 }

10646 /*=====
10647 *                               w_finish                               *
10648 *=====*/
10649     PRIVATE int w_finish()
10650     {
10651     /* Ejecutar solicitudes de E/S reunidas en wtrans[] */
10652
10653     struct trans *tp = wtrans;
10654     struct wini *wn = w_wn;
10655     int r, errors;
10656     struct command cmd;
10657     unsigned cylinder, head, sector, secspcyl;
10658
10659     if (w_count == 0) return(OK); /* Finalización espuria. */
10660
10661     r = ERR; /* Disparar el primer com_out */
10662     errors = 0;
10663
10664     do {
10665         if (r != OK) {
10666             /* El controlador en har. debe (re)programarse. */
10667
10668             /* Primero ver si es preciso reinicializar. */
10669             if (1 (wn->state & INITIALIZED) && w_specify() != OK)
10670                 return(tp->iop->io_nbytes = EIO);
10671
10672             /* Decir al controlador en har. que transfiera w_count bytes */
10673             cmd.precomp = wn->precomp;
10674             cmd.count = (w_count » SECTOR_SHIFT) & BYTE;
10675             if (wn->ldhpref & LDH_LBA) {
10676                 cmd.sector = (tp->block >> 0) & 0xFF;
10677                 cmd.cyl_lo = (tp->block >> 8) & 0xFF;
10678                 cmd.cyl_hi = (tp->block >> 16) & 0xFF;
10679                 cmd.ldh = wn->ldhpref | ((tp->block >>
10680                                     } else {
10681                 secspcyl = wn->pheads * wn->psectors;
10682                 cylinder = tp->block / secspcyl;
10683                 head = (tp->block % secspcyl) / wn->psectors;
10684                 sector = tp->block % wn->psectors;
10685                 cmd.sector = sector + 1;
10686                 cmd.cyl_lo = cylinder & BYTE;
10687                 cmd.cyl_hi = (cylinder » 8) & BYTE;
10688                 cmd.ldh = wn->ldhpref | head;
10689                 }

```

```

10690         cmd.command = w_opcode == DEV_WRITE ? CMD_WRITE : CMD_READ;
10691
10692         if ((r = com_out(&cmd)) != OK) {
10693             if (++errors == MAX_ERRORS) {
10694                 w_command = CMD_IDLE;
10695                 return(tp->iop->io_nbytes = EIO);
10696             }
10697             continue;          /* Retry */
10698         }
10699     }
10700
10701     /* Para cada sector, esperar interrupción y traer datos (read)
10702      * o enviar datos al contr. har. y esperar interrupción (write).
10703      */
10704
10705     if (w_opcode == DEV_READ) {
10706         if ((r = w_intr_wait()) == OK) {
10707             /* Copiar datos del buffer del disp. al espacio de usuario. */
10708
10709             port_read(wn->base + REG_DATA, tp->phys, SECTOR_SIZE);
10710
10711             tp->phys += SECTOR_SIZE;
10712             tp->iop->io_nbytes -= SECTOR_SIZE;
10713             w_count -= SECTOR_SIZE;
10714             if ((tp->count -= SECTOR_SIZE) == 0) tp++;
10715         } else {
10716             /* ¿Hay datos defectuosos? */
10717             if (w_status & STATUS_DRQ) {
10718                 port_read(wn->base + REG_DATA, tmp_phys,
10719                     SECTOR_SIZE); 10720 }
10721         }
10722     } else {
10723         /* Esperar datos solicitados. */
10724         if (!waitfor(STATUS_DRQ, STATUS_DRQ)) {
10725             r = ERR;
10726         } else {
10727             /* Llenar el buffer de la unidad. */
10728
10729             port_write(wn->base + REG_DATA, tp->phys, SECTOR_SIZE);
10730             r = w_intr_wait();
10731         }
10732
10733         if (r == OK) {
10734             /* Reservar bytes escritos con éxito. */
10735
10736             tp->phys += SECTOR_SIZE;
10737             tp->iop->io_nbytes -= SECTOR_SIZE;
10738             w_count -= SECTOR_SIZE;
10739             if ((tp->count -= SECTOR_SIZE) == 0) tp++;
10740         }
10741     }
10742
10743     if (r != OK) {
10744         /* No reintentar si sector marcado malo o demasiados errores. */
10745         if (r == ERR_BAD_SECTOR || ++errors == MAX_ERRORS) {
10746             w_command = CMD_IDLE;
10747             return(tp->iop->io_nbytes = EIO);
10748         }
10749     }

```

```

10750          /* Restablecer si a la mitad, pero abandonar si EfS opcional. */
10751          if (errors == MAX_ERRORS / 2) {
10752              w_need_reset();
10753              if (tp->iop->io_request & OPTIONAL_IO) {
10754                  w_command = CMD_IDLE;
10755                  return(tp->iop->io_nbytes = EIO);
10756              }
10757          }
10758          continue;          /* Reintentar */
10759      }
10760      errors = 0;
10761  } while (w_count > 0);
10762
10763  w_command = CMD_IDLE;
10764  return(OK);
10765  }

10768 /*=====
10769 *                               com_out                               *
10770 *=====*/
10771  PRIVATE int com_out(cmd)
10772          struct command *cmd;          /* Bloque de comando */
10773  {
10774      /* Enviar bloque de comando a contr. har. winchester y devolver situación */
10775
10776      struct wini *wn = w_wn;
10777      unsigned base = wn->base;
10778
10779      if (!waitfor(STATUS_BSY, 0)) {
10780          printf("%s: controller not ready\n", w_name());
10781          return(ERR);
10782      }
10783
10784      /* Escoger unidad. */
10785      out_byte(base + REG_LDH, cmd->ldh);
10786
10787      if (!waitfor(STATUS_BSY, 0)) {
10788          printf("%s: drive not ready\n", w_name());
10789          return(ERR);
10790      }
10791
10792      /* Planificar llamada pfdespertar, algunos controladores no confiables. */
10793      clock_mess(WAKEUP, w_timeout);
10794
10795      out_byte(base + REG_CTL, wn->pheads >= 8 ? CTL_EIGHTHEADS : 0);
10796      out_byte(base + REG_PRECOMP, cmd->precomp);
10797      out_byte(base + REG_COUNT, cmd->count);
10798      out_byte(base + REG_SECTOR, cmd->sector);
10799      out_byte(base + REG_CYL_LO, cmd->cyl_lo);
10800      out_byte(base + REG_CYL_HI, cmd->cyl_hi);
10801      lock();
10802      out_byte(base + REG_COMMAND, cmd->command);
10803      w_command = cmd->command;
10804      w_status = STATUS_BSY;
10805      unlock();
10806      return(OK);
10807  }

```

```

10810 /*=====
10811 *                               w_need_reset                               *
10812 *=====*/
10813 PRIVATE void w_need_reset()
10814 {
10815 /* Hay que restablecer el controlador en hardware. */
10816 struct wini *wn;
10817
10818 for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
10819     wn->state |= DEAF;
10820     wn->state &= -INITIALIZED;
10821 }
10822 }

10825 /*=====
10826 *                               w_do_close                               *
10827 *=====*/
10828 PRIVATE int w_do_close(dp, m_ptr)
10829 struct driver *dp;
10830 message *m_ptr;
10831 {
10832     /* Cerrar dispositivo: Liberar un dispositivo. */
10833
10834     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
10835     w_wn->open_ct--;
10836     return(OK);
10837 }

10840 /*=====
10841 *                               com_simple                               *
10842 *=====*/
10843 PRIVATE int com_simple(cmd)
10844 struct command *cmd; /* Bloque de comando. */
10845 {
10846     /* Comando de controlador simple, sólo 1 int. sin fase de sacar datos. */
10847     int r;
10848
10849     if ((r = com_out(cmd)) == OK) r = w_intr_wait();
10850     w_command = CMD_IDLE;
10851     return(r);
10852 }

10855 /*=====
10856 *                               w_timeout                               *
10857 *=====*/
10858 PRIVATE void w_timeout()
10859 {
10860     struct wini *wn = w_wn;
10861
10862     switch (w_command) {
10863     case CMD_IDLE:
10864         break; /* perfecto */
10865     case CMD_READ:
10866     case CMD_WRITE:
10867         /* Imposible, pero no en PC: El controlador en har. no responde. */
10868
10869         /* Limitar la E/S multiseccion parece ayudar. */

```

```

10870     if (wn->max_count > 8 * SECTOR_SIZE) {
10871         wn->max_count = 8 * SECTOR_SIZE;
10872     } else {
10873         wn->max_count = SECTOR_SIZE;
10874     }
10875     /*CONTINUAR CON LA SIGUIENTE RUTINA*/
10876     default:
10877         /* Algún otro comando. */
10878         printf("%s: timeout on command %02x\n", w_name(), w_command);
10879         w_need_reset();
10880         w_status = 0;
10881         interrupt(WINCHESTER);
10882     }
10883 }

10886 /*=====
10887  *                               w_reset                               *
10888 *=====*/
10889 PRIVATE int w_reset()
10890 {
10891     /* Enviar un reset al controlador en hardware. Esto se hace después
10892     * de cualquier catástrofe, como cuando el controlador no responde.
10893     */
10894
10895     struct wini *wn;
10896     int err;
10897
10898     /* Esperar recuperación interna de la unidad. */
10899     milli_delay(RECOVERYTIME);
10900
10901     /* Bit de restablecimiento estroboscópico */
10902     out_byte(w_wn->base + REG_CTL, CTL_RESET);
10903     milli_delay(1);
10904     out_byte(w_wn->base + REG_CTL, 0);
10905     milli_delay(1);
10906
10907     /* Esperar a que el controlador esté listo */
10908     if (!w_waitfor(STATUS_BSY | STATUS_RDY, STATUS_RDY)) {
10909         printf("%s: reset failed, drive busy\n", w_name());
10910         return(ERR);
10911     }
10912
10913     /* El reg. de error debe revisarse aquí, pero algunas unidades se equivocan. */
10914
10915     for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
10916         if (wn->base == w_wn->base) wn->state &= -DEAF;
10917     }
10918     return(OK);
10919 }

10922 /*=====
10923  *                               w_intr_wait                           *
10924 *=====*/
10925 PRIVATE int w_intr_wait()
10926 {
10927     /* Esperar que una tarea termine interrupción y devuelva resultados. */
10928
10929     message mess;

```



```

10930 int r;
10931
10932 /* Esperar interrupción que inicie w_status para "no ocupado". */
10933 while (w_status & STATUS_BSY) receive (HARDWARE , &mess;
10934
10935 /* Verificar situación. */
10936     lock();
10937 if ((w_status & (STATUS_BSY | STATUS_RDY | STATUS_WF | STATUS_ERR))
10938     == STATUS_RDY) {
10939     r = OK;
10940     w_status |= STATUS_BSY; /* suponer aún ocupada con E/S */
10941 } else
10942 if ((w_status & STATUS_ERR) && (in_byte(w_wn->base + REG_ERROR) & ERROR_BB)) {
10943     r = ERR_BAD_SECTOR; /* sector malo, inútil reintentar */
10944 } else {
10945     r = ERR; /* cualquier otro error */
10946 }
10947 unlock();
10948 return(r);
10949 }

10952 /*=====
10953 *                               w_waitfor                               *
10954 *=====*/
10955     PRIVATE int w_waitfor(mask, value)
10956         int mask; /* máscara de situación */
10957         int value; /* situación requerida */
10958     {
10959 /* Esperar que controlador esté en estado requerido. Devolver 0 al vencer plazo. */
10960
10961 struct milli_state ms;
10962
10963     milli_start(&ms);
10964 do {
10965         if (in_byte(w_wn->base + REG_STATUS) & mask) == value) return 1;
10966 } while (milli_elapsed(&ms) < TIMEOUT);
10967
10968 w_need_reset(); /* El controlador en hardware está sordo. */
10969     return(0);
10970 }

10973 /*=====
10974 *                               w_handler                               *
10975 *=====*/
10976     PRIVATE int w_handler(irq)
10977     int irq;
10978     {
10979     /* Interrupción de disco, enviar mens. a tarea winch. y rehabil. ints. */
10980
10981 w_status = in_byte(w_wn->base + REG_STATUS); /* acusar interrupción */
10982     interrupt(WINCHESTER);
10983 return 1;
10984     }

10987 /*=====
10988 *                               w_geometry                               *
10989 *=====*/

```

```

10990 PRIVATE void w_geometry(entry)
10991 struct partition *entry;
10992 {
10993     entry->cylinders = w_wn->lcylinders;
10994     entry->heads = w_wn->lheads;
10995     entry->sectors = w_wn->lsectors;
10996 }
10997 #endif /* ENABLE_AT_WINI */

```

```

+++++
src/kernel/clock.c
+++++

```

```

11000 /* Este archivo contiene el código y datos para la tarea del reloj. Esta tarea
11001 * acepta seis tipos de mensajes:
11002 *
11003 * HARD_INT: ocurrió una interrupción de reloj
11004 * GET UPTIME: obtener tiempo en tics desde el arranque
11005 * GET-TIME: un proceso quiere el tiempo real en segundos
11006 * SET-TIME: un proc. quiere fijar el tiempo real en segundos
11007 * SET-ALARM: un proc. quiere que le avisen después de cierto intervalo
11008 * SET:=SYN_AL: fijar la alarma síncrona
11009 *
11010 *
11011 * El mensaje de entrada es formato m6. Los parámetros son:
11012 *
11013 * m_type CLOCK_PROC FUNC NEW_TIME
11014 * -----
11015 * |HARD_INT | | | |
11016 * |-----+-----+-----+-----|
11017 * |GET_UPTIME | | | |
11018 * |-----+-----+-----+-----|
11019 * |GET_TIME | | | |
11020 * |-----+-----+-----+-----|
11021 * |SET_TIME | | | newtime |
11022 * |-----+-----+-----+-----|
11023 * |SET_ALARM |proc_nr | f a llam. | delta |
11024 * |-----+-----+-----+-----|
11025 * |SET_SYN_AL |proc_nr | | delta |
11026 * -----
11027 * NEW_TIME, DELTA_CLICKS y SECONDS_LEFT se refieren
11028 * al mismo campo del mensaje, dependiendo del tipo de mensaje.
11029 *
11030 * Los mensajes de respuesta son de tipo OK, excepto en caso de HARD_INT,
11031 * al cual no se responde. Para los mensajes GET_* se devuelve el tiempo
11032 * en el campo NEW_TIME, y para SET_ALARM y SET_SYN_ALARM
11033 * se devuelve en ese campo el tiempo en segundos que falta para que se devuelva
11034 * la alarma.
11035 *
11036 * Al sonar una alarma, si el invocador es un proc. de usuario, se le envía
11037 * una señal SIGALRM. Si es una tarea, se invocará una func. especificada
11038 * por el invocador, la cual podría, p.ej., enviar un mensaje, pero
11039 * sólo si está segura de que la tarea estará bloqueada cuando el temporiz.
11040 * llegue a 0. Una alarma síncrona envía un mens. a la tarea de alarma
11041 * síncrona, que a su vez puede enviar un mens. a otro servidor. Ésta es la única
11042 * forma de enviar una alarma a un servidor, pues los servidores no pueden
11043 * usar el meco de llamar func. que usan las tareas y no pueden recibir señales.
11044 */

```

```

11045
11046 #include "kernel.h"
11047 #include <signal.h>
11048 #include <minix/callnr.h>
11049 #include <minix/com.h>
11050 #include "proc.h"
11051
11052 /* Definiciones de constantes. */
11053 #define MILLISEC 100 /* frec. con que se llama la planif. (ms) */
11054 #define SCHED_RATE (MILLISEC*HZ/1000) /* # tics por planificación */
11055
11056 /* parámetros del reloj. */
11057 #define COUNTER_FREQ (2*TIMER_FREQ) /* frec. del contador con onda cuadrada*/
11058 #define LATCH COUNT 0x00 /* cc00xxxx, c = canal, x = cualq. */
11059 #define SQUARE WAVE 0x36 /* ccaammmb, a = access, m = modo, b = BCD */
11060 /* 11x11, 11 = LSB luego MSB, x11 = onda c. */
11061 #define TIMER_COUNT ((unsigned) (TIMER_FREQ/HZ)) /* val. inic. contador*/
11062 #define TIMER_FREQ 1193182L /* frec. reloj p/temporiz. en PC y AT */
11063
11064 #define CLOCK_ACK_BIT 0x80 /* bit acuse interrupción de reloj PS/2 */
11065
11066 /* Variables de la tarea de reloj. */
11067 PRIVATE clock t realtime; /* reloj de tiempo real */
11068 PRIVATE time t boot time; /* segundos desde arranque del sist. */
11069 PRIVATE clock t next alarm; /* tiempo probable de sigo alarma */
11070 PRIVATE message mc; /* bufo mensaje p/entrada y salida */
11071 PRIVATE int watchdog_proc; /* contiene proc_nr en llamo de *watch_dog[ ]*/
11072 PRIVATE watchdog_t watch_dog[NR_TASKS+NR_PROCS];
11073
11074 /* Variables empleadas por las tareas de reloj y de alarma síncrona */
11075 PRIVATE int syn_al_alive= TRUE; /* no despertar syn_alm_task antes inited*/
11076 PRIVATE int syn_table[NR_TASKS+NR_PROCS]; /* qué tareas rec. CLOCK_INT*/
11077
11078 /* Variables que el manejador de interrupciones modifica */
11079 PRIVATE clock_t pending_ticks; /* tics vistos sólo por nivel bajo */
11080 PRIVATE int sched_ticks = SCHED_RATE; /* contador: si 0, llamar planif. */
11081 PRIVATE struct proc *prev_ptr; /* últ. proc. usuario ejec. por tarea reloj */ 11082
11083 FORWARD PROTOTYPE( void common_setalarm, (int proc_nr,
11084 -long delta_ticks, watchdog_t fuction) );
11085 FORWARD_PROTOTYPE( void do_clocktick, (void) );
11086 FORWARD_PROTOTYPE( void do_get_time, (void) );
11087 FORWARD PROTOTYPE( void do_getuptime, (void) );
11088 FORWARD -PROTOTYPE( void do-set time, (message *m_ptr) );
11089 FORWARD =PROTOTYPE( void do=setalarm, (message *m_ptr) );
11090 FORWARD_PROTOTYPE~E( void init_clock, (void) );
11091 FORWARD_PROTOTYPE( void cause_alarm, (void) );
11092 FORWARD_PROTOTYPE( void do_setsyn_alm, (message *m_ptr) );
11093 FORWARD_PROTOTYPE( int clock_handler, (int irq) );
11094
11095 /*=====
11096 * clock_task *
11097 *=====*/
11098 PUBLIC void clock_task()
11099 {
11100 /* Programa principal de la tarea de reloj. Corrige realtime
11101 * sumando tics pendientes vistos sólo por el servicio de ints., luego
11102 * determina cuál de las 6 posibles llamadas es examinando 'mc.m_type'.
11103 * Luego despacha.
11104 */

```

```

11105
11106 int opcode;
11107
11108 init_clock();          /* inicializ. tarea de reloj */
11109
11110 /* Ciclo principal: Obtener trabajo, procesarlo, a veces responder. */
11111 while (TRUE) {
11112     receive(ANY, &mc);    /* obtener un mensaje */
11113     opcode = mc.m_type;   /* extraer cód. de función */
11114
11115     lock();
11116     realtime += pending_ticks; /* transf. tics de mane. de bajo nivel */
11117     pending_ticks = 0;     /* plno tener que preocuparnos p/ellos */
11118     unlock();
11119
11120     switch (opcode) {
11121         case HARD_INT:    do_clocktick();          break;
11122         case GET_ÜPTIME: do_getuptime();          break;
11123         case GET_TIME:   do_get_time();           break;
11124         case SET_TIME:   do_set_time(&mc);        break;
11125         case SET_ALARM:  do_setalarm(&mc);        break;
11126         case SET_SYNC_AL:do_setsyn_alm(&mc);     break;
11127         default:         panic("clock task got bad message", mc.m_type);
11128     }
11129
11130     /* Enviar respuesta, excepto por tic de reloj. */
11131     mc.m_type = OK;
11132     if (opcode != HARD_INT) send(mc.m_source, &mc);
11133 }
11134 }

11137 /*=====
11138 *                               do_clocktick                               *
11139 *=====*/
11140 PRIVATE void do_clocktick()
11141 {
11142     /* A pesar de su nombre, esta rutina no se invoca en cada tic. Se invoca
11143 * en los tics en los que hay que efectuar mucho trabajo.
11144 */
11145
11146 register struct proc *rp;
11147 register int proc_nr;
11148
11149 if (next_alarm <= realtime) {
11150     /* Tal vez sonó una alarma, pero el proc pudo haber salido, verificar. */
11151     next_alarm = LONG_MAX; /* comenzar a calco sigio alarma */
11152     for (rp = 8EG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
11153         if (rp->p_alarm != 0) {
11154             /* Ver si se llegó a este tiempo de alarma. */
11155             if (rp->p_alarm <= realtime) {
11156                 /* Un temporiz. llegó a 0. Si es proc.
11157 * usuario, enviarle señalj si es tarea,
11158 * inv. funGo antes especific. por la tarea.
11159 */
11160                 proc_nr = proc_number(rp);
11161                 if (watch_dog[proc_nr+NR_TASKS]) {
11162                     watchdog_proc= proc_nr;
11163                     (*watch_dog[proc_nr+NR_TASKS])();
11164                 }

```

```

11165             else
11166                 cause_sig(proc_nr, SIGALRM);
11167             rp->p_alarm = 0;
11168         }
11169
11170         /* Determinar cuál alarma sigue. */
11171         if (rp->p_alarm != 0 && rp->p_alarm < next_alarm)
11172             next_alarm = rp->p_alarm;
11173     }
11174 }
11175 }
11176
11177 /* Si un proc. usuario se ha ejecutado demasiado tiempo, escoger otro. */
11178 if (--sched_ticks == 0) {
11179     if (bill_ptr == prev_ptr) lock_sched();           /* proc. se ejec. demasiado */
11180     sched_ticks = SCHED_RATE;                       /* restablecer cuanto */
11181     prev_ptr = bill_ptr;                            /* nuevo proceso anterior */
11182 }
11183 }

11186 /*=====
11187 *                               do_gettime                               *
11188 *=====*/
11189 PRIVATE void do_gettime()
11190 {
11191     /* Obtener y devolver el tiempo desde arranque en tics. */
11192
11193     mc.NEW_TIME = realtime;           /* tiempo desde arranque actual */
11194 }
11197 /*=====
11198 *                               get_uptime                               *
11199 *=====*/
11200 PUBLIC clock_t get_uptime()
11201 {
11202     /* Obt. y devolver tiempo desde arranque en tics. Func. diseñada
11203 * para ser llamada desde otras tareas para obtener el tiempo
11204 * sin el gasto que implican los mensajes. Debe tener cuidado con pending_ticks.
11205 */
11206
11207     clock_t uptime;
11208
11209     lock();
11210     uptime = realtime + pending_ticks;
11211     unlock();
11212     return(uptime);
11213 }

11216 /*=====
11217 *                               do_get_time                               *
11218 *=====*/
11219 PRIVATE void do_get_time()
11220 {
11221     /* Obtener y devolver tiempo de reloj actual en segundos. */
11222
11223     mc.NEW_TIME = boot_time + realtime/HZ;           /* tiempo real actual */
11224 }

```

```

11227 /*=====
11228 *                               do_set_time                               *
11229 *=====*/
11230     PRIVATE void do_set_time(m_ptr)
11231     message *m_ptr; /* apunto al mensaje de solicitud */
11232     {
11233     /* Poner reloj tiempo real. Sólo el superusuario puede usar esta llamada. */
11234
11235     boot_time = m_ptr->NEW_TIME -realtime/HZ;
11236     }

11239 /*=====
11240 *                               do_setalarm                               *
11241 *=====*/
11242     PRIVATE void do_setalarm(m_ptr)
11243     message *m_ptr; /* apunto al mensaje de solicitud */
11244     {
11245     /* Un proceso pide una señal de alarma o una tarea pide que se invoque
11246 * una tunco vigilante dada después de cierto intervalo.
11247     */
11248
11249     register struct proc *rp;
11250     int proc_nr; /* qué proc. quiere alarma */
11251     long delta_ticks; /* ¿en cuántos tics la quiere? */
11252     watchdog_t function; /* tunco por llamar (sólo tareas) */
11253
11254     /* Extraer los parámetros del mensaje. */
11255     proc_nr = m_ptr->CLOCK_PROC_NR; /* proc. que interrumpir después */
11256     delta_ticks = m_ptr->DELTA_TICKS; /* cuántos tics esperar */
11257     function = (watchdog_t) m_ptr->FUNC_TO_CALL;
11258     /* tunco por llamar (sólo tareas) */
11259     rp = proc_addr(proc_nr);
11260     mc.SECONDS_LEFT = (rp->p_alarm == 0 ? 0 : (rp->p_alarm -realtime)/HZ );
11261     if (listaskp(rp)) function= 0; /* procs. usu. son señalizados */
11262     common_setalarm(proc_nr, delta_ticks, function);
11263     }

11266 /*=====
11267 *                               do_setsyn_alrm                               *
11268 *=====*/
11269     PRIVATE void do_setsyn_alrm(m_ptr)
11270     message *m_ptr; /* apunto a mensaje de solicitud */
11271     {
11272     /* Un proceso pide una alarma síncrona.
11273     */
11274
11275     register struct proc *rp;
11276     int proc_nr; /* qué proc. pide la alarma */
11277     long delta_ticks; /* ¿en cuántos tics la pide? */
11278
11279     /* Extraer los parámetros del mensaje. */
11280     proc_nr = m_ptr->CLOCK_PROC_NR; /* proc. que interrumpir después */
11281     delta_ticks = m_ptr->DELTA_TICKS; /* cuántos tics esperar */
11282     rp = proc_addr(proc_nr);
11283     mc.SECONDS_LEFT = (rp->p_alarm == 0 ? 0 : (rp->p_alarm -realtime)/HZ );
11284     common_setalarm(proc_nr, delta_ticks, cause_alarm);

```

```
11285     }

11288 /*=====
11289 *                               common_setalarm                               *
11290 *=====*/
11291     PRIVATE void common_setalarm(proc_nr, delta_ticks, function)
11292 int proc_nrj                               /* qué proc. pide la alarma */
11293 long delta_tickSj                          /* ¿en cuántos tics la pide? */
11294 watchdog t function;                       /* tuncos por llamar (0 si se debe
11295                                         * llamar cause_sig) */
11296     {
11297     /* Terminar trabajo de do_set_alarm y do_setsyn_alm. Registrar solicitud
11298 * de alarma y ver si es la sigo alarma que se necesita.
11299     */
11300
11301     register struct proc *rp;
11302
11303     rp = proc_addr(proc_nr);
11304     rp->p_alarm = (delta_ticks == 0 ? 0 : realtime + delta_ticks);
11305     watch_dog[proc_nr+NR_TASKSI] = function;
11306
11307     /* ¿Cuál alarma sigue? */
11308     next_alarm = LONG_MAX;
11309     for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++)
11310         if(rp->p_alarm != 0 && rp->p_alarm < next_alarm)next_alarm=rp->p_alarm;
11311
11312     }

11315 /*=====
11316 *                               cause_alarm                               *
11317 *=====*/
11318     PRIVATE void cause_alarm()
11319     {
11320     /* Rutina invocada si un temporiz. llegó a 0 y el proc. solicitó alarma
11321 * sincrona. El núm. proc. está en la varo global watchdog_proc (HACK).
11322     */
11323     message mess;
11324
11325     syn_table[watchdog_proc + NR_TASKSI]= TRUEj
11326     if (!syn_al_alive) send (SYN_ALRM_TASK, &mess)j
11327     }

11330 /*=====
11331 *                               syn_alm_task                               *
11332 *=====*/
11333     PUBLIC void syn_alm_task()
11334     {
11335     /* Programa principal de la tarea de alarma sincrona.
11336 * Esta tarea sólo recibe mensajes de cause_alarm en la tarea de reloj.
11337 * Envía un mensaje CLOCK_INT a un proc. que solicitó una syn_alm.
11338 * Las alarmas sincronas, a diferencia de las señales o de la activación
11339 * de un vigilante, son recibidas por un proceso cuando está en una parte
11340 * conocida de su código, es decir, cuando ha emitido una llamada
11341 * para recibir un mensaje.
11342     */
11343
11344     message mess;
```

```

11345 int work_done;          /* ¿listo para dormir? */
11346 int *al_ptr;           /* apuntador en syn_table */
11347 int i;
11348
11349 syn_al_alive= TRUE;
11350 for (i= 0, al_ptr= syn_table; i<NR_TASKS+NR_PROCS; i++, al_ptr++)
11351     *al_ptr= FALSE;
11352
11353 while (TRUE) {
11354     work_done= TRUE;
11355     for (i= 0, al_ptr= syn_table; i<NR_TASKS+NR_PROCS; i++, al_ptr++)
11356         if (*al_ptr) {
11357             *al_ptr= FALSE;
11358             mess.m_type= CLOCK_INT;
11359             send (i-NR_TASKS, &mess);
11360             work_done= FALSE;
11361         }
11362     if (work_done) {
11363         syn_al_alive= FALSE;
11364         receive (CLOCK, &mess);
11365         syn_al_alive= TRUE;
11366     }
11367 }
11368
11371 /*=====*
11372 *                               clock_handler                               *
11373 *=====*/
11374PRIVATE int clock_handler(irq)
11375int irq;
11376 {
11377 /* Se ejecuta cada tic del reloj (cada vez que el chip temporiz. genera
11378  * una interrupción). Hace un poco de trabajo para que la tarea de reloj
11379  * no tenga que invocarse en cada tic.
11380  *
11381  * Conmutar contexto a do_clocktick si sonó una alarma. También
11382  * conmutar ahí para replanificar si la replanificación va a hacer algo.
11383  * Esto sucede cuando
11384  *     (1) el cuanto expiró
11385  *     (2) el proc. actual recibió cuanto completo (isegún reloj!)
11386  *     (3) algo más está listo para ejecutarse.
11387  * También llamar TTY y PRINTER Y dejar que hagan lo que sea necesario.
11388  *
11389  * Se accede a muchas variables globales y estáticas aquí. Se debe justificar
11390  * la seguridad de esto. Casi ninguna se modifica aquí:
11391  *     k_reenter:
11392  *         Ésta indica sin peligro si la int. de reloj está anidada.
11393  *     proc_ptr, bill_ptr:
11394  *         Éstas se usan p/contabil. No importa si proc.c las modifica
11395  *         siempre que sean apunto válidos, ya que en peor caso
11396  *         se cobraría al proc. anterior.
11397  *     next_alarm, realtime, sched_ticks, bill_ptr, prev_ptr,
11398  *     rdy_head[USER_Q]:
11399  *         Éstas se prueban p/decidir si invocar interrupt().
11400  *         No importa si la prueba es (raras veces) al revés por
11401  *         competencia, ya que el procesam. de ~lto nivel sólo
11402  *         se retrasará un tic, o se llamará innecesariamente el alto nivel.
11403  * Las variables que se modifican requieren más cuidado:
11404  *     rp->user_time, rp->sys_time:

```



```

11405 *           Éstas se protegen con candados explícitos en system.c.
11406 *           No se protegen debidamente en dmp.c (incremento no
11407 *           atómico), pero eso no importa.
11408 *           pending_ticks:
11409 *           Se protege con candados explícitos en clock.c.
11410 *           No actualizar realtime directamente, pues hay demasiadas
11411 *           referencias a ella para cuidar fácilmente.
11412 *           lost_ticks:
11413 *           Tics de reloj contados fuera de la tarea de reloj.
11414 *           sched_ticks, prev_ptr:
11415 *           Al actualizo éstas se compite con cód. similar en do_clocktick().
11416 *           No hace falta candado, porque si algo malo sucede (como
11417 *           sched_ticks negativo), el cód. de do_clocktick() restaurará las
11418 *           variables a valores razonables, y no importa un sched() faltante
11419 *           o extra ocasional.
11420 *
11421 * ¿Valen la pena estas complicaciones? Bueno, hacen el sistema 15%
11422 * más rápido en una 8088 de 5MHz, y facilitan mucho la depuración,
11423 * pues no hay conmutaciones de tarea en un sistema inactivo.
11424 */
11425
11426 register struct proc *rp;
11427 register unsigned ticks;
11428 clock_t now;
11429
11430 if (ps_mca) {
11431     /* Acusar recibo de la interrupción de reloj PS/2. */
11432     out_byte(PORT_B, in_byte(PORT_B) | CLOCK_ACK_BIT);
11433 }
11434
11435 /* Actualizar tiempos de usuario y sistema contabilizados.
11436 * Primero cobrar tiempo de usuario al proc. actual.
11437 * Si el proc. actual no es el facturable (p.ej. si es una tarea), cobrar
11438 * al proc. facturable tiempo de sistema también. Así, el tiempo de usuario
11439 * de la tarea no tacto es el tiempo de sist. del usuario facturable.
11440 */
11441 if (k_reenter != 0)
11442     rp = proc_addr(HARDWARE);
11443 else
11444     rp = proc_ptr;
11445 ticks = lost_ticks + 1;
11446 lost_ticks = 0;
11447 rp->user_time += ticks;
11448 if (rp != bill_ptr && rp != proc_addr(IDLE)) bill_ptr->sys_time += ticks;
11449
11450 pending_ticks += ticks;
11451 now = realtime + pending_ticks;
11452 if (tty_timeout <= now) tty_wakeup(now);           /* quizá despertar TTV */
11453 pr_restart();                                     /* quizá reinic. impresora */
11454
11455 if (next_alarm <= now ||
11456     sched_ticks == 1 &&
11457     bill_ptr == prev_ptr &&
11458     rdy_head[USER_Q] != NIL_PROC) {
11459     interrupt(CLOCK);
11460     return 1;           /* Rehabilitar interrupciones */
11461 }
11462
11463 if (--sched_ticks == 0) {
11464     /* Si bill_ptr == prev_ptr, no hay usuarios listos, no neces. sched(). */

```

```

11465     sched_ticks = SCHED_RATE;           /* restablecer cuanto */
11466     prev_ptr = bill_ptr;                  /* nuevo proceso previo */
11467 }
11468 return 1;                                /* Reenable clock interrupt */
11469 }

11471 /*=====
11472 *                               init_clock                               *
11473 *=====*/
11474 PRIVATE void init_clock()
11475 {
11476     /* Inicializar canal 0 del temporizador 8253A a p.ej. 60 Hz. */
11477
11478     out_byte(TIMER_MODE, SQUARE_WAVE); /* ejec. continua de temporiz. */
11479     out_byte(TIMER0, TIMER_COUNT);     /* cargar byte bajo temporiz. */
11480     out_byte(TIMER0, TIMER_COUNT >> 8); /* cargar byte alto temporiz. */
11481     put_irq_handler(CLOCK_IRQ, clock_handler); /* fijar manej. de ints. */
11482     enable_irq(CLOCK_IRQ);             /* listo para ints. de reloj */
11483 }

11486 /*=====
11487 *                               clock_stop                               *
11488 *=====*/
11489 PUBLIC void clock_stop()
11490 {
11491     /* Restablecer reloj a la frec. de BIOS (para rearranque) */
11492
11493     out_byte(TIMER_MODE, 0x36);
11494     out_byte(TIMER0, 0);
11495     out_byte(TIMER0, 0);
11496 }

11499 /*=====
11500 *                               milli_delay                               *
11501 *=====*/
11502 PUBLIC void milli_delay(millisec)
11503     unsigned millisec;
11504 {
11505     /* Retardo de algunos milisegundos. */
11506
11507     struct milli_state ms;
11508
11509     milli_start(&ms);
11510     while (milli_elapsed(&ms) < millisec) {}
11511 }

11513 /*=====
11514 *                               milli_start                               *
11515 *=====*/
11516 PUBLIC void milli_start(msp)
11517     struct milli_state *msp;
11518 {
11519     /* Prepararse para llamadas a milli_elapsed(). */
11520
11521     msp->prev_count = 0;
11522     msp->accum_count = 0;
11523 }

```

```

11526 I*=====
11527 *                milli_elapsed                *
11528 *=====I
11529     PUBLIC unsigned milli_elapsed(msp)
11530     struct milli_state *msp;
11531                                     {
11532     I* Devolver el número de milisegundos desde la llamada a milli_start().
11533 * Debe escrutarse rápidamente.
11534     *I
11535     unsigned count;
11536
11537 I* Leer contador de canal 0 del temporiz. 8253A. El contador
11538 * se decrementa a 2 x la frec del temporiz. (un ciclo por cada mitad
11539 * de la onda cuadrada). El contador normalmente tiene un valor
11540 * entre 0 y TIMER_COUNT, pero antes de que la tarea del reloj
11541 * se haya inicializado su valor máx. es 65535, fijado por BIOS.
11542     *I
11543     out_byte(TIMER_MODE, LATCH_COUNT);          I* hacer chip copiar count en latch *I
11544     count = in_byte(TIMER0);                    I* cuenta reg. continúa durante lectura *I
11545     count |= in_byte(TIMER0) << 8;
11546
11547 I* Sumar diferencia entre cuenta previa y nueva a menos que el contador
11548 * haya aumentado (reinic. su ciclo). Podríamos perder un tic de v~z
11549 * en cuando, pero no necesitamos precisión de ms.
11550     *I
11551     msp->accum_count += count <= msp->prev_count ? (msp->prev_count - count) : 1;
11552     msp->prev_count = count;
11553
11554     return msp->accum_count / (TIMER_FREQ / 1000);
11555     }

```

```

+++++
src/kernel/tty.h
+++++
11600     I*     tty.h - Terminales *I
11601
11602     #define TTY_IN_BYTES    256    I* tamaño cola entrada tty *I
11603     #define TAB_SIZE       8      I* distancia entre tabulaciones *I
11604     #define TAB_MASK       7      I* máscara para calco pos. de tab. *I
11605
11606     #define ESC '\33'    I* escape *I
11607
11608     #define O_NOCTTY      00400    I* de <fcntl.h>, o cc se ahogará *I
11609     #define O_NONBLOCK    04000
11610
11611     typedef _PROTOTYPE( void (*devfun_t), (struct tty *tp) );
11612     typedef _PROTOTYPE( void (*devfunarg_t), (struct tty *tp, int c) );
11613
11614     typedef struct tty {
11615     int tty_events;                I* fijar cuándo TTY debe examinar esta línea *I
11616
11617 I* Cola de entrada. Las digit. se guardan aquí mientras un prog. las lee. *I
11618     u16_t *tty_inhead;            I* apunto a donde va el sigo carácter *I
11619     u16_t *tty_intail;           I* apunto al sigo caro que se dará al prog. *I

```

```

11620 int tty_incount;          /* # cars. en cola de entrada */
11621 int tty_eotct;           /* # "saltos línea" en cola entrada */
11622 devfun_t tty_devread;    /* rutina p/leer de bufs. bajo nivel */
11623 devfun_t tty_icancel;    /* cancelar entradas de dispositivos */
11624 int tty_min;             /* # mín. cars. solíc. en cola ent. */
11625 clock_t tty_time;        /* tiempo entrada estará disponible */
11626 struct tty *tty_timenext; /* p/lista ttys con temporiz. activos */
11627
11628 /* Sección de salida. */
11629 devfun_t tty_devwrite;    /* rut. p/iniciar salida de disp. real */
11630 devfunarg_t tty_echo;    /* rut. p/hacer eco de cars. introd. */
11631 devfun_t tty_ocancel;    /* cancelar salida de disp. actual */
11632 devfun_t tty_break;      /* que el disp. envíe un corte */
11633
11634 /* parámetros y situación de la terminal. */
11635 int tty_position;         /* pos. actual en pantalla p/eco */
11636 char tty_reprint;        /* 1 si eco de entradas mal, 0 sí ok */
11637 char tty_escaped;        /* 1 si llegó LNEXT (AV), 0 si no */
11638 char tty_inhibited;      /* 1 si llegó STOP (AS) (detiene salida) */
11639 char tty_pgrp;           /* # ranura del proceso que controla */
11640 char tty_openct;         /* cuenta de aperturas de esta tty */
11641
11642 /* Aquí se guarda info. sobre solicitudes de E/S incompletas. */
11643 char tty_inrepcode;       /* cód. respuesta, TASK_REPLY o REVIVE */
11644 char tty_incaller;        /* proceso que invocó (usualmente FS) */
11645 char tty_inproc;         /* proc. que quiere leer de tty */
11646 vir_bytes tty_in_vir;    /* dirección virtual p/enviar datos */
11647 int tty_inleft;          /* ¿cuántos más cars. se necesitan? */
11648 int tty_incum;           /* # cars. introducidos hasta ahora */
11649 char tty_outrepcode;     /* cód. respuesta, TASK_REPLY o REVIVE */
11650 char tty_outcaller;      /* proceso que invocó (usualmente FS) */
11651 char tty_outproc;        /* proc. que quiere escribir en tty */
11652 vir_bytes tty_out_vir;   /* dirección virtual de origen datos */
11653 int tty_outleft;         /* # cars. aún por enviar */
11654 int tty_outcum;          /* # cars. enviados hasta ahora */
11655 char tty_iocaller;       /* proceso que invocó (usualmente FS) */
11656 char tty_ioproc;         /* proc. que quiere hacer un ioctl */
11657 int tty_ioreq;           /* código de solicitud de ioctl */
11658 vir_bytes tty_iovir;    /* dir. virtual de buffer de ioctl */
11659
11660 /* Diversos. */
11661 devfun_t tty_ioctl;       /* fijar velo línea, etc. en nivel disp. */
11662 devfun_t tty_close;       /* decir a dispositivo que tty cerrada */
11663 void *tty_priv;          /* apunto a datos privados por disp. */
11664 struct termios tty_termios; /* atributos de terminal */
11665 struct winsize tty_winsize; /* tamaño ventana (# líneas y # cols.) */
11666
11667 u16_t tty_inbuf[TTY_IN_BYTES]; /* buffer de entrada de tty */
11668 } tty_t;
11669
11670 EXTERN tty_t tty_table[NR_CONS+NR_RS_LINES+NR_PTYS];
11671
11672 /* Valores para los campos. */
11673 #define NOT_ESCAPED      0          /* caro previo no es LNEXT (AV) */
11674 #define ESCAPED          1          /* caro previo fue LNEXT (AV) */
11675 #define RUNNING          0          /* no caro STOP (AS) p/detener salida */
11676 #define STOPPED          1          /* se tecleó STOP (AS) p/detener salida */
11677
11678 /* Campos y banderas p/caracteres en la cola de entrada. */
11679 #define IN_CHAR          0x00FF     /* 8 bits bajos son el carácter mismo */

```

```

11680 #define IN_LEN      0x0F00    /* long. de caro si se hizo eco de él */
11681 #define IN_LSHIFT   8        /* longitud = (c & IN_LEN) »    IN_LSHIFT */
11682 #define IN_EOT 0x1000    /* caro es salto de línea (D, LF) */
11683 #define IN_EOF 0x2000    /* caro es EOF (D), no devolv. a usuario */
11684 #define IN_ESC 0x4000    /* escapado con LNEXT (V), no interpretar */
11685
11686 /* Tiempos y plazos. */
11687 #define TIME_NEVER ((clock_t) -1 < 0 ? (clock_t) LONG_MAX : (clock_t) -1)
11688 #define force_timeout() ((void) (tty_timeout = 0))
11689
11690 EXTERN tty_t *tty_timelist; /* lista de ttys con temporiz. activos */
11691
11692 /* Número de elementos y límite de un buffer. */
11693 #define buflen(buf) (sizeof(buf) / sizeof((buf)[0]))
11694 #define bufend(buf) ((buf) + buflen(buf))

```

```

+++++
src/kernel/tty.c
+++++

```

```

11700 /* Este archivo contiene el controlador en sof. de la terminal, tanto consola
11701 * IBM como terms. ASCII normales. Sólo maneja la parte indep. del disp.
11702 * de una TTV. Las partes dependientes están en console.c, rs232.c, etc.
11703 * El arch. tiene dos ptos. de entrada princ., tty_task() y tty_wakeup().
11704 * Y varios ptos. entrada menores para uso del cód. dependo del disp.
11705 *
11706 * La parte indep. del disp. acepta entradas de "teclado" de la parte dependo
11707 * del disp., procesa entradas (interpreta teclas especiales) y las envía
11708 * a un proc. que lee de la TTV. Las salidas a TTV se envían al cód.
11709 * dependo del disp. para procesarse y exhibo en "pantalla". El procesam.
11710 * de entradas lo hace el disp. que invoca 'in_process' con los caro de
11711 * entrada; el procesam. de salidas puede ser hecho por el disp.
11712 * mismo o llamando 'out_process'. TTV se ocupa de poner en cola
11713 * entradas, el disp. pone en cola salidas. Si un disp. recibe una señal
11714 * externa, como una interrupción, hace que la tarea CLOCK ejecute
11715 * tty_wakeup() para (adivinó) despertar la TTV y que verifique
11716 * si puede continuar la entrada o la salida.
11717 *
11718 * Los mensajes válidos y sus parámetros son:
11719 *
11720 * HARD_INT: terminó salida o llegó entrada
11721 * DEV_READ: un proceso quiere leer de una terminal
11722 * DEV_WRITE: un proceso quiere escribir en una terminal
11723 * DEV_IOCTL: un proc. quiere cambiar los parámetros de una terminal
11724 * DEV_OPEN: se abrió una línea de tty
11725 * DEV_CLOSE: se cerró una línea de tty
11726 * CANCEL: terminar inmediato llamada a sist. incompleta previa
11727 *
11728 * m_type      TTV_LINE PROC_NR  COUNT  TTV_SPEK TTV FLAGS ADDRESS
11729 * -----
11730 * | HARD_INT  |          |          |          |          |          |          |
11731 * |-----+-----+-----+-----+-----+-----+-----+
11732 * | DEV_READ  | disp sec | # proc  | cuenta  |          | O_NONBLOCK | ap buf |
11733 * |-----+-----+-----+-----+-----+-----+-----+
11734 * | DEV_WRITE | disp sec | # proc  | cuenta  |          |          | ap buf |

```

```

11735 * |-----+-----+-----+-----+-----+-----+-----|
11736 * | DEV_IOCTL  |disp sec | # proc |cod func | borra etc |banderas | |
11737 * |-----+-----+-----+-----+-----+-----+-----|
11738 * | DEV_OPEN   |disp sec | # proa | O_NOCTTY| | | |
11739 * |-----+-----+-----+-----+-----+-----+-----|
11740 * | DEV CLaSE  |disp sec | # proc | | | | |
11741' * |-----+-----+-----+-----+-----+-----+-----|
11742 * | CANCEL     |disp sec | # proc | | | | |
11743 * |-----+-----+-----+-----+-----+-----+-----|
11744   */
11745
11746   #include "kernel.h"
11747   #include <termios.h>
11748   #include <sys/ioctl.h>
11749   #include <signal.h>
11750   #include <minix/callnr.h>
11751   #include <minix/com.h>
11752   #include <minix/keymap.h>
11753   #include "tty.h"
11754   #include "proc.h"
11755
11756 /* Dirección de una estructura tty. */
11757   #define tty_addr(line) (&tty_table[line])
11758
11759 /* Primeros núms. segundo para las diversas clases de dispositivos TTY. */
11760   #define CONS_MINOR    0
11761   #define LOG_MINOR     15
11762   #define RS232_MINOR   16
11763   #define TTYPX_MINOR   128
11764   #define PTYPX_MINOR   192
11765
11766 /* Macros para tipos de tty mágicos. */
11767   #define isconsole(tp)  ((tp) < tty_addr(NR_CONS))
11768
11769 /* Macros para apuntadores a estructuras de tty mágicas. */
11770   #define FIRST_TTY     ttyaddr(0)
11771   #define END_TTY       tty=addr(sizeof(tty_table)
/ sizeof(tty_table[0]))
11772
11773   /* Existe un dispositivo si está definida por lo menos su función 'devread'. */
11774   #define tty_active(tp)  ((tp)->tty_devread != NULL)
11775
11776   /* Las líneas RS232 o pseudoterminal pueden contigo completamente. */
11777   #if NR_RS_LINES == 0
11778 #define rs_init(tp)      ((void) 0)
11779   #endif
11780   #if NR_PTYS == 0
11781 #define pty_init(tp)     ((void) 0)
11782 #define do_pty(tp, mp)   ((void) 0)
11783   #endif
11784
11785   FORWARD_PROTOTYPE( void do_cancel, (tty_t *tp, message *m_ptr) );
11786   FORWARD_PROTOTYPE( void do_ioctl, (tty_t *tp, message *m_ptr) );
11787   FORWARD_PROTOTYPE( void do_open, (tty_t *tp, message *m_ptr) );
11788   FORWARD_PROTOTYPE( void do_close, (tty_t *tp, message *m_ptr) );
11789   FORWARD_PROTOTYPE( void do_read, (tty_t *tp, message *m_ptr) );
11790   FORWARD_PROTOTYPE( void do_write, (tty_t *tp, message *m_ptr) );
11791   FORWARD_PROTOTYPE( void in_transfer, (tty_t *tp) );
11792   FORWARD_PROTOTYPE( int echo, (tty_t *tp, int ch) );
11793   FORWARD_PROTOTYPE( void rawecho, (tty_t *tp, int ch) );
11794   FORWARD_PROTOTYPE( int back_over, (tty_t *tp) );

```

```

11795 FORWARD _PROTOTYPE( void reprint, (tty_t *tp) );
11796 FORWARD _PROTOTYPE( void dev_ioctl, (tty_t *tp) );
11797 FORWARD _PROTOTYPE( void setattr, (tty_t *tp) );
11798 FORWARD _PROTOTYPE( void tty_icancel, (tty_t *tp) );
11799 FORWARD _PROTOTYPE( void tty_init, (tty_t *tp) );
11800 FORWARD _PROTOTYPE( void settimer, (tty_t *tp, int on) );
11801
11802 /* Atributos predeterminados. */
11803 PRIVATE struct termios termios_defaults = {
11804 TINPUT_DEF, TOUTPUT_DEF, TCTRL_DEF, TLOCAL_DEF, TSPEED_DEF, TSPEED_DEF,
11805 {
11806 TEOF_DEF, TEOL_DEF, TERASE_DEF, TINTR_DEF, TKILL_DEF, TMIN_DEF,
11807 TQUIT_DEF, TTIME_DEF, TSUSP_DEF, TSTART_DEF, TSTOP_DEF,
11808 TREPRINT_DEF, TLNEXT_DEF, TDISCARD_DEF,
11809 },
11810 };
11811 PRIVATE struct winsize winsize_defaults; /* = puros ceros */
11812
11813
11814 /*=====
11815 *
11816 *=====*/
11817 PUBLIC void tty_task()
11818 {
11819 /* Rutina principal de la tarea de terminal. */
11820
11821 message tty_mess; /* buffer p/todos los mensajes entrantes */
11822 register tty_t *tp;
11823 unsigned line;
11824
11825 /* Inicializar las líneas de terminal. */
11826 for (tp = FIRST_TTY; tp < END_TTY; tp++) tty_init(tp);
11827
11828 /* Exhibir letrero de inicio de Minix. */
11829 printf("Minix %s.%s Copyright 1997 Prentice-Hall, Inc.\n\n",
11830 OS_RELEASE, OS_VERSION);
11831 printf("Executing in 32-bit protected mode\n\n");
11832
11833 while (TRUE) {
11834 /* Manejar cualesquier eventos en cualquier tty. */
11835 for (tp = FIRST_TTY; tp < END_TTY; tp++) {
11836 if (tp->tty_events) handle_events(tp);
11837 }
11838
11839 receive(ANY, &tty_mess);
11840
11841 /* Una int. de hardware es una invitación a verificar si hubo eventos. */
11842 if (tty_mess.m_type == HARD_INT) continue;
11843
11844 /* Verificar el número de dispositivo secundario. */
11845 line = tty_mess.TTY_LINE;
11846 if ((line -CONS_MINOR) < NR_CONS) {
11847 tp = tty_addr(line -CONS_MINOR);
11848 } else
11849 if (line == LOG_MINOR) {
11850 tp = tty_addr(0);
11851 } else
11852 if ((line -RS232_MINOR) < NR_RS_LINES) {
11853 tp = tty_addr(line -RS232_MINOR + NR_CONS);
11854 } else

```

```

11855     if ((line -TTYPX_MINOR) < NR_PTYS) {
11856         tp = tty_addr(line -TTYPX_MINOR + NR_CONS + NR_RS_LINES);
11857     } else
11858     if ((line -PTYPX_MINOR) < NR_PTYS) {
11859         tp = tty_addr(line -PTYPX_MINOR + NR_CONS + NR_RS_LINES);
11860         do_pty(tp, &tty_mess);
11861         continue; /* es una pty, no una tty */
11862     } else {
11863         tp = NULL;
11864     }
11865
11866     /* Si el disp. no existe o no está configurado, devolver ENXIO. */
11867     if (tp == NULL || !tty_active(tp)) {
11868         tty_reply(TASK_REPLY, tty_mess.m_source,
11869                 tty_mess.PROC_NR, ENXIO);
11870         continue;
11871     }
11872
11873     /* Ejecutar la función solicitada. */
11874     switch (tty_mess.m_type) {
11875         case DEV_READ:      do_read(tp, &tty_mess);          break;
11876         case DEV_WRITE:    do_write(tp, &tty_mess);         break;
11877         case DEV_IOCTL:    do_ioctl(tp, &tty_mess);         break;
11878         case DEV_OPEN:     do_open(tp, &tty_mess);          break;
11879         case DEV_CLOSE:    do_close(tp, &tty_mess);         break;
11880         case CANCEL:       do_cancel(tp, &tty_mess);        break;
11881         default:           tty_reply(TASK_REPLY, tty_mess.m_source,
11882                                   tty_mess.PROC_NR, EINVAL);
11883     }
11884 }
11885 }
11886 /*=====
11887 *
11888 *
11889 *
11890 *=====*/
11891 PRIVATE void do_read(tp, m_ptr)
11892 register tty_t *tp; /* apuntador a struct tty */
11893 message *m_ptr; /* apunto a mens. enviado a tarea */
11894 {
11895     /* Un proceso quiere leer de una terminal. */
11896     int r;
11897
11898     /* Comprobar si ya hay un proceso suspendido por lectura, verificar
11899     * si los parámetros son correctos, efectuar E/S.
11900     */
11901     if (tp->tty_inleft > 0) {
11902         r = EIO;
11903     } else
11904     if (m_ptr->COUNT <= 0) {
11905         r = EINVAL;
11906     } else
11907     if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
11908         r = EFAULT;
11909     } else {
11910         /* Copiar info. del mensaje a la struct. tty. */
11911         tp->tty_inrepcode = TASK_REPLY;
11912         tp->tty_incaller = m_ptr->m_source;
11913         tp->tty_inproc = m_ptr->PROC_NR;
11914         tp->tty_in_vir = (vir_bytes) m_ptr->ADDRESS;

```



```

11915     tp->tty_inleft = m_ptr->COUNT;
11916
11917     if (! (tp->tty_termios.c_lflag & ICANON)
11918         && tp->tty_termios.C_Cc[VTIME] > 0) {
11919         if (tp->tty_termios.C_CC[VMIN] == 0) {
11920             /* MIN Y TIME especifican temporiz. que termina lectura
11921              * en TIME/10 seg si no hay bytes disponibles.
11922              */
11923             lock();
11924             settimer(tp, TRUE);
11925             tp->tty_min = 1;
11926             unlock();
11927         } else {
11928             /* MIN Y TIME especifican temporiz. entre bytes que podría
11929              * tener que cancelarse si no hay bytes aún.
11930              */
11931             if (tp->tty_eotct == 0) {
11932                 lock();
11933                 settimer(tp, FALSE);
11934                 unlock();
11935                 tp->tty_min = tp->tty_termios.c_CC[VMIN];
11936             }
11937         }
11938     }
11939
11940     /* ¿Algo esperando en buffer de entrada? Despejarlo... */
11941     in_transfer(tp);
11942     /* ...y luego regresar por más */
11943     handle_events(tp);
11944     if (tp->tty_inleft == 0) return; /* ya se hizo */
11945
11946     /* No había bytes disponibles en la cola, así que suspender invocador
11947      * o terminar la lectura si no es bloqueadora.
11948      */
11949     if (m_ptr->TTY_FLAGS & O_NONBLOCK) {
11950         r = EAGAIN; /* cancelar la lectura */
11951         tp->tty_inleft = tp->tty_incum = 0;
11952     } else {
11953         r = SUSPEND; /* suspender el invocador */
11954         tp->tty_inrecode = REVIVE;
11955     }
11956 }
11957 tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
11958 }

```

```

11961 /*=====
11962 *                               do_write                               *
11963 *=====*/
11964 PRIVATE void do_write(tp, m_ptr)
11965 register tty_t *tp;
11966 register message *m_ptr; /* apunto a mensaje enviado a tarea */
11967 {
11968     /* Un proceso quiere escribir en una terminal. */
11969     int r;
11970
11971     /* Comprobar si ya hay un proceso suspendido por escritura, verificar
11972      * si los parámetros son correctos, efectuar E/S.
11973      */
11974     if (tp->tty_outleft > 0) {

```

```

11975         r = EIO;
11976     } else
11977     if (m_ptr->COUNT <= 0) {
11978         r = EINVAL;
11979     } else
11980     if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
11981         r = EFAULT;
11982     } else {
11983         /* Copiar parámetros del mensaje en la estructura tty. */
11984         tp->tty_outrcode = TASK_REPLY;
11985         tp->tty_outcaller = m_ptr->m_source;
11986         tp->tty_outproc = m_ptr->PROC_NR;
11987         tp->tty_out_vir = (vir_bytes) m_ptr->ADDRESS;
11988         tp->tty_outleft = m_ptr->COUNT;
11989
11990         /* Tratar de escribir. */
11991         handle_events(tp);
11992         if (tp->tty_outleft == 0) return;           /* ya se hizo */
11993
11994         /* Imposible escribir algún o todos los bytes, así que suspender
11995          * invocador o terminar la escritura si es no bloqueadora.
11996          */
11997         if (m_ptr->TTY_FLAGS & O_NONBLOCK) {     /* cancelar escritura */
11998             r = tp->tty_outcum > 0 ? tp->tty_outcum : EAGAIN;
11999             tp->tty_outleft = tp->tty_outcum = 0;
12000         } else {
12001             r = SUSPEND;                         /* suspender invocador */
12002             tp->tty_outrcode = REVIVE;
12003         }
12004     }
12005     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
12006 }

12009 /*=====
12010 *                               do_ioctl                               *
12011 *=====*/
12012 PRIVATE void do_ioctl(tp, m_ptr)
12013 register tty_t *tp;
12014 message *m_ptr;           /* apunto a mensaje enviado a tarea */
12015 {
12016     /* Realizar IOCTL en esta terminal. La llamada al sistema
12017     * IOCTL maneja las llamadas terminos de Posix
12018     */
12019
12020     int r;
12021     union {
12022         int i;
12023         /* ya no se usan estos paráms. no Posix, pero se conserva la unión
12024         * para minimizar diferencias de código con versión compatible hacia atrás
12025         * struct sgtyb sg;
12026         * struct tchars tc;
12027         */
12028     } param;
12029     phys_bytes user_phys;
12030     size_t size;
12031
12032     /* Tamaño del parámetro de ioctl. */
12033     switch (m_ptr->TTY_REQUEST) {
12034         case TCGETS:           /* función tcgetattr de Posix */

```

```
12035 case TCSETS:          /* función tcsetattr de Posix, opción TCSANOW */
12036 case TCSETSW:         /* función tcsetattr de Posix, opción TCSADRAIN */
12037 case TCSETSF:         /* función tcsetattr de Posix, opción TCSAFLUSH */
12038     size = sizeof(struct termios);
12039     break;
12040
12041 case TCSBRK:          /* función tcsetattr de Posix */
12042 case TCFLW:          /* función tcsetattr de Posix */
12043 case TCFLSH:         /* función tcsetattr de Posix */
12044 case TIOCGPGRP:      /* función tegetpgrp de Posix */
12045 case TIOCSPPGRP:     /* función tcsetpgrp de posix */
12046     size = sizeof(int);
12047     break;
12048
12049 case TIOCGWINSZ:     /* obtener tamaño ventana (no Posix) */
12050 case TIOCSWINSZ:     /* fijar tamaño ventana (no Posix) */
12051     size = sizeof(struct winsize);
12052     break;
12053
12054 case KIOCSMAP:       /* cargar mapa de teclas (ext. de Minix) */
12055     size = sizeof(keymap_t);
12056     break;
12057
12058 case TIOCSFON:       /* cargar fuente (extensión de Minix) */
12059     size = sizeof(u8_t [8192]);
12060     break;
12061
12062 case TCDRAIN:        /* función tcdrain de Posix, sin parám. */
12063 default:             size = 0;
12064 }
12065
12066 if (size != 0) {
12067     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, size);
12068     if (user_phys == 0) {
12069         tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, EFAULT);
12070         return;
12071     }
12072 }
12073
12074 r = OK;
12075 switch (m_ptr->TTY_REQUEST) {
12076 case TCGETS:
12077     /* Obtener atributos de terminos. */
12078     phys_copy(vir2phys(&tp->tty_termios), user_phys, (phys_bytes) size);
12079     break;
12080
12081 case TCSETSW:
12082 case TCSETSF:
12083 case TCDRAIN:
12084     if (tp->tty_outleft > 0) {
12085         /* Esperar que termine procesam. actual de salidas. */
12086         tp->tty_iocaller = m_ptr->m_source;
12087         tp->tty_ioproc = m_ptr->PROC_NR;
12088         tp->tty_ioreq = m_ptr->REQUEST;
12089         tp->tty_iovir = (vir_bytes) m_ptr->ADDRESS;
12090         r = SUSPEND;
12091         break;
12092     }
12093     if (m_ptr->TTY_REQUEST == TCDRAIN) break;
12094     if (m_ptr->TTY_REQUEST == TCSETSF) tty_icancel(tp);
```

```

12095     /*SEGUIRSE A LA SIGUIENTE RUTINA*/
12096 case TCSETS:
12097     /* Fijar los atributos de terminos. */
12098     phys_copy(user_phys, vir2phys(&tp->tty_termios), (phys_bytes) size);
12099     setattr(tp);
12100     break;
12101
12102 case TCFLSH:
12103     phys_copy(user_phys, vir2phys(&param.i), (phys_bytes) size);
12104     switch (param.i) {
12105         case TCIFLUSH:      tty_icancel(tp);                break;
12106         case TCOFLUSH:     (*tp->tty_ocancel) (tp);        break;
12107         case TCIOFLUSH:    tty_icancel(tp); (*tp->tty_ocancel) (tp);break;
12108         default:           r = EINVAL;
12109     }
12110     break;
12111
12112 case TCFLOW:
12113     phys_copy(user_phys, vir2phys(&param.i), (phys_bytes) size);
12114     switch (param.i) {
12115         case TCOOFF:
12116         case TCOON:
12117             tp->tty_inhibited = (param.i == TCOOFF);
12118             tp->tty_events = 1;
12119             break;
12120         case TCIOFF:
12121             (*tp->tty_echo) (tp, tp->tty_termios.c_cc[VSTOP]);
12122             break;
12123         case TCION:
12124             (*tp->tty_echo) (tp, tp->tty_termios.c_CC[VSTARTI]);
12125             break;
12126         default:
12127             r = EINVAL;
12128     }
12129     break;
12130
12131 case TCSBRK:
12132     if (tp->tty_break != NULL) (*tp->tty_break) (tp);
12133     break;
12134
12135 case TIOCGWINSZ:
12136     phys_copy(vir2phys(&tp->tty_winsize) , user_phys, (phys_bytes) size);
12137     break;
12138
12139 case TIOCSWINSZ:
12140     phys_copy(user_phys, vir2phys(&tp->tty_winsize) , (phys_bytes) size);
12141     /* SIGWINCH... */
12142     break;
12143
12144 case KIOCSMAP:
12145     /* Cargar nuevo mapa de teclas (sólo /dev/console). */
12146     if (isconsole(tp)) r = kbd_loadmap(user_phys);
12147     break;
12148
12149 case TIOCSFON:
12150     /* Cargar fuente en una tarjeta EGA o VGA (hs@hck.hr) */
12151     if (isconsole(tp)) r = con_loadfont(user_phys);
12152     break;
12153
12154 /* Se permite que estas funciones Posix fallen si no está

```

```

12155 * definido _POSIX_JOB_CONTROL.
12156 */
12157     case TIOCGPRG:
12158     case TIOCSPGRP:
12159         default:
12160             r = ENOTTV;
12161     }
12162
12163     /* Enviar la respuesta. */
12164     tty_reply(TASK_REPLV, m_ptr->m_source, m_ptr->PROC_NR, r);
12165 }

12168 /*=====
12169 *                                     do_open                                     *
12170 *=====*/
12171     PRIVATE void do_open(tp, m_ptr)
12172     register tty_t *tp;
12173     message *m_ptr;          /* apunto a mens. enviado a tarea */
12174     {
12175     /* Se abrió 1 línea tty. Que sea la tty controladora del invocador si O_NOCTTV
12176 * está apagada y no es el dispositivo de bitácora. Se devuelve 1
12177 * si la tty se hace la controladora; si no, OK o un código de error.
12178 */
12179     int r = OK;
12180
12181     if (m_ptr->TTV_LINE == LOG_MINOR) {
12182         /* El disp. de bitácora es un disp. de diagnóstico sólo de escritura. */
12183         if (m_ptr->COUNT & R_BIT) r = EACCES;
12184     } else {
12185         if (1 (m_ptr->COUNT & O_NOCTTV)) {
12186             tp->tty_pgrp = m_ptr->PROC_NR;
12187             r = 1;
12188         }
12189         tp->tty_opencnt++;
12190     }
12191     tty_reply(TASK_REPLV, m_ptr->m_source, m_ptr->PROC_NR, r);
12192 }

12195 /*=====
12196 *                                     do_close                                     *
12197 *=====*/
12198     PRIVATE void do_close(tp, m_ptr)
12199     register tty_t *tp;
12200     message *m_ptr; /* apunto a mens. enviado a tarea */
12201     {
12202     /* Se cerró 1 línea tty. Asear línea si es último cierre. */
12203
12204     if (m_ptr->TTV_LINE != LOG_MINOR && !tp->tty_opencnt) {
12205         tp->tty_pgrp = 0;
12206         tty_icancel(tp);
12207         (*tp->tty_ocancel) (tp);
12208         (*tp->tty_close) (tp);
12209         tp->tty_termios = termios_defaults;
12210         tp->tty_winsize = winsize_defaults;
12211         setattr(tp);
12212     }
12213     tty_reply(TASK_REPLV, m_ptr->m_source, m_ptr->PROC_NR, OK);
12214 }

```

```

12217 /*=====
12218 *                               do_cancel                               *
12219 *=====*/
12220     PRIVATE void do_cancel(tp, m_ptr)
12221     register tty_t *tp;
12222     message *m_ptr; /* apunto a mensaje enviado a tarea */
12223     {
12224     /* Se envió una señal a un proceso que está suspendido tratando de leer
12225     * o escribir. Terminar inmediato lectura o escritura pendiente.
12226     */
12227
12228     int proc_nr;
12229     int mode;
12230
12231     /* Revisar paráms. con cuidado, p/evitar cancelar dos veces. */
12232     proc_nr = m_ptr->PROC_NR;
12233     mode = m_ptr->COUNT;
12234     if ((mode & R_BIT) && tp->tty_inleft != 0 && proc_nr == tp->tty_inproc) {
12235     /* El proceso leía cuando se terminó. Asear entradas. */
12236     tty_icancel(tp);
12237     tp->tty_inleft = tp->tty_incum = 0;
12238     }
12239     if ((mode & W_BIT) && tp->tty_outleft != 0 && proc_nr == tp->tty_outproc) {
12240     /* El proceso escribía cuando se terminó. Asear salidas. */
12241     (*tp->tty_ocancel)(tp);
12242     tp->tty_outleft = tp->tty_outcum = 0;
12243     }
12244     if (tp->tty_ioreq != 0 && proc_nr == tp->tty_ioprocc) {
12245     /* El proceso estaba esperando que la salida drenara. */
12246     tp->tty_ioreq = 0;
12247     }
12248     tp->tty_events = 1;
12249     tty_reply(TASK_REPLY, m_ptr->m_source, proc_nr, EINTR);
12250     }

12253 /*=====
12254 *                               handle_events                           *
12255 *=====*/
12256     PUBLIC void handle_events(tp)
12257     tty_t *tp;
12258     {
12259     /* Manejar cualesquier eventos pendientes en una TTY; por lo regular son interrupciones
12260     * de dispositivos.
12261     *
12262     * Dos clases de eventos sobresalen:
12263     *   -se recibió 1 caro de la consola o línea RS232.
12264     *   -línea RS232 completó solicitud de escrito (p/un usuario).
12265     * El manejador de ints. puede retardar el mens. de int. a discreción
12266     * p/Ino saturar la tarea TTY. Pueden sobrescribirse mensajes si las líneas
12267     * son rápidas o hay competo entre diferentes líneas, entrada y salida,
12268     * porque MINIX sólo tiene 1 buffer p/mensajes de int. (en proc.c). Esto
12269     * se maneja revisando explícitamente cada línea para detectar entradas
12270     * nuevas y salidas terminadas en cada interrupción.
12271     */
12272     char *buf;
12273     unsigned count;
12274

```

```

12275 do {
12276     tp->tty_events = 0j
12277
12278     /* Leer entradas y procesarlas. */
12279     (*tp->tty_devread)(tp);
12280
12281     /* Procesar salidas y escribirlas. */
12282     (*tp->tty_devwrite) (tp) ;
12283
12284     /* ¿Espera ioctl algún evento? */
12285     if (tp->tty_ireq != 0) dev_ioctl(tp);
12286 } while (tp->tty_events);
12287
12288 /* Transferir cargo de cola de entrada a proceso en espera. */
12289 in_transfer(tp) ;
12290
12291 /* Contestar si hay suficientes bytes disponibles. */
12292 if (tp->tty_incum >= tp->tty_min && tp->tty_inleft > 0) {
12293     tty_reply(tp->tty_inrecode, tp->tty_incaller, tp->tty_inproc,
12294                                     tp->tty_incum);
12295     tp->tty_inleft = tp->tty_incum = 0;
12296 }
12297 }

12300 /*=====
12301 *                                     in_transfer                                     *
12302 *=====*/
12303 PRIVATE void in_transfer(tp)
12304     register tty_t *tp; /* apunto a terminal de donde leer */
12305 {
12306     /* Transf. bytes de cola de entrada a 1 proc. que lee de una terminal. */
12307
12308     int ch;
12309     int count;
12310     phys_bytes buf_phys, user_base;
12311     char buf[641, *bp;
12312
12313     /* ¿Algo que hacer? */
12314     if (tp->tty_inleft == 0 || tp->tty_eotct < tp->tty_min) return;
12315
12316     buf_phys = vir2phys(buf);
12317     user_base = proc_vir2phys(proc_addr(tp->tty_inproc), 0);
12318     bp = buf;
12319     while (tp->tty_inleft > 0 && tp->tty_eotct > 0) {
12320         ch = *tp->tty_intail;
12321
12322         if (1 (ch & IN_EOF)) {
12323             /* Un carácter p/entregar al usuario. */
12324             *bp = ch & IN_CHAR;
12325             tp->tty_inleft--;
12326             if (++bp == bufend(buf)) {
12327                 /* Buf. temp. lleno, copiar en espacio usuario. */
12328                 phys_copy(buf_phys, user_base + tp->tty_in_vir,
12329                             (phys_bytes) buflen(buf));
12330                 tp->tty_in_vir += buflen(buf);
12331                 tp->tty_incum += buflen(buf);
12332                 bp = buf;
12333             }
12334         }

```

```

12335
12336     /* Quitar carácter de la cola de entrada. */
12337     if (++tp->tty_intail == bufend(tp->tty_inbuf))
12338         tp->tty_intail = tp->tty_inbuf;
12339     tp->tty_incount--;
12340     if (ch & IN_EOT) {
12341         tp->tty_eotct--j
12342         /* No leer después de salto-línea en modo canónico. */
12343         if (tp->tty_termios.c_lflag & ICANON) tp->tty_inleft = 0;
12344     }
12345 }
12346
12347 if (bp > buf) {
12348     /* Caracteres sobrantes en buffer. */
12349     count = bp - buf;
12350     phys_copy(buf_phys, user_base + tp->tty_in_vir, (phys_bytes) count);
12351     tp->tty_in_vir += count;
12352     tp->tty_incum += count;
12353 }
12354
12355 /* Usualmente contestar a lector, quizá aun si incum == 0 (EOF). */
12356 if (tp->tty_inleft == 0) {
12357     tty_reply(tp->tty_inrecode, tp->tty_incaller, tp->tty_inproc,
12358             tp->tty_incum);
12359     tp->tty_inleft = tp->tty_incum = 0;
12360 }
12361 }

12364 /*=====
12365 *                                     in_process                                     *
12366 *=====*/
12367 PUBLIC int in_process(tp, buf, count)
12368 register tty_t *tp;                /* terminal en la que llegó carácter */
12369 char *buf;                          /* buffer con caracteres de entrada */
12370 int count;                          /* núm. de caracteres de entrada */
12371 {
12372     /* Se acaban de teclear caracteres. Procesar, guardar y hacer eco. Devolver
12373     * el número de caracteres procesados.
12374     */
12375
12376     int ch, sig, ct;
12377     int timeset = FALSE;
12378     static unsigned char csize_mask[] = { 0x1F, 0x3F, 0x7F, 0xFF };
12379
12380     for (ct = 0; ct < count; ct++) {
12381         /* Tomar un carácter. */
12382         ch = *buf++ & BYTE;
12383
12384         /* ¿Reducir a siete bits? */
12385         if (tp->tty_termios.c_iflag & ISTRIP) ch &= 0x7F;
12386
12387         /* ¿Extensiones de entrada? */
12388         if (tp->tty_termios.c_lflag & IEXTEN) {
12389
12390             /* ¿Caro anterior fue de escape? */
12391             if (tp->tty_escaped) {
12392                 tp->tty_escaped = NOT_ESCAPED;
12393                 ch |= IN_ESC;        /* proteger carácter */
12394             }

```



```

12395
12396     /* ¿LNEXT (AV) p/escapar sigio carácter? */
12397     if (ch == tp->tty_termios.c_cc[VLNEXT]) {
12398         tp->tty_escaped = ESCAPED;
12399         rawecho(tp, 'A');
12400         rawecho(tp, '\b');
12401         continue;          /* no guardar el escape */
12402     }
12403
12404     /* ¿REPRINT (AR) p/reimprimir cars. de eco? */
12405     if (ch == tp->tty_termios.c_cc[VREPRINT]) {
12406         reprint(tp);
12407         continue;
12408     }
12409 }
12410
12411 /* _POSIX_VDISABLE es valor de caro normal, mejor escaparlo. */
12412 if (ch == _POSIX_VDISABLE) ch |= IN_ESC;
12413
12414 /* Convertir CR en LF, ignorar CR, o convertir LF en GR. */
12415 if (ch == '\r') {
12416     if (tp->tty_termios.c_iflag & IGNCR) continue;
12417     if (tp->tty_termios.c_iflag & ICRNL) ch = '\n';
12418 } else
12419 if (ch == '\n') {
12420     if (tp->tty_termios.c_iflag & INLCR) ch = '\r';
12421 }
12422
12423 /* ¿Modo canónico? */
12424 if (tp->tty_termios.c_lflag & ICANON) {
12425
12426     /* Procesar borrado (del último carácter). */
12427     if (ch == tp->tty_termios.c_cc[VERASE]) {
12428         (void) back_over(tp);
12429         if (!(tp->tty_termios.c_lflag & ECHOE)) {
12430             (void) echo(tp, ch);
12431         }
12432         continue;
12433     }
12434
12435     /* Terminar procesamiento (quitar línea actual). */
12436     if (ch == tp->tty_termios.c_cc[VKILL]) {
12437         while (back_over(tp)) {}
12438         if (!(tp->tty_termios.c_lflag & ECHOE)) {
12439             (void) echo(tp, ch);
12440             if (tp->tty_termios.c_lflag & ECHOK)
12441                 rawecho(tp, '\n');
12442         }
12443         continue;
12444     }
12445
12446     /* EOF (AD) es fin-de-archivo, "salto de línea" invisible. */
12447     if (ch == tp->tty_termios.c_cc[VEOF]) ch |= IN_EOT | IN_EOF;
12448
12449     /* La línea puede devolverse al usuario después de un LF. */
12450     if (ch == '\n') ch |= IN_EOT;
12451
12452     /* Lo mismo con EOL, sea lo que sea. */
12453     if (ch == tp->tty_termios.c_cc[VEOL]) ch |= IN_EOT;
12454 }

```

```

12455
12456 /* ¿Control de entradas start/stop? */
12457 if (tp->tty_termios.c_iflag & IXON) {
12458
12459     /* Las salidas paran con STOP (^S). */
12460     if (ch == tp->tty_termios.c_cc[VSTOP]) {
12461         tp->tty_inhibited = STOPPED;
12462         tp->tty_events = 1;
12463         continue;
12464     }
12465
12466     /* Salidas reinician con START (^O) o cualq. caro si IXANY. */
12467     if (tp->tty_inhibited) {
12468         if (ch == tp->tty_termios.c_cc[VSTART]
12469             || (tp->tty_termios.c_iflag & IXANY)) {
12470             tp->tty_inhibited = RUNNING;
12471             tp->tty_events = 1;
12472             if (ch == tp->tty_termios.c_cc[VSTART])
12473                 continue;
12474         }
12475     }
12476 }
12477
12478 if (tp->tty_termios.c_iflag & ISIG) {
12479     /* Detectar caracteres INTR (?) Y OUIT (\). */
12480     if (ch == tp->tty_termios.c_cc[VINTR]
12481         || ch == tp->tty_termios.c_cc[VOUIT]) {
12482         sig = SIGINT;
12483         if (ch == tp->tty_termios.c_cc[VOUIT]) sig = SIGOUIT;
12484         sigchar(tp, sig);
12485         (void) echo(tp, ch);
12486         continue;
12487     }
12488 }
12489
12490 /* ¿Hay espacio en el buffer de entrada? */
12491 if (tp->tty_incount == buflen(tp->tty_inbuf)) {
12492     /* No hay; desechar en modo canónico, guardar en modo crudo. */
12493     if (tp->tty_termios.c_iflag & ICANON) continue;
12494     break;
12495 }
12496
12497 if (! (tp->tty_termios.c_iflag & ICANON)) {
12498     /* En modo crudo todos los cars. son "saltos de linea". */
12499     ch |= IN_EOT;
12500
12501     /* ¿Poner en marcha un temporizador entre bytes? */
12502     if (!timeset && tp->tty_termios.c_cc[VMIN] > 0
12503         && tp->tty_termios.c_cc[VTIME] > 0) {
12504         lock();
12505         settimer(tp, TRUE);
12506         unlock();
12507         timeset = TRUE;
12508     }
12509 }
12510
12511 /* Realizar la intrincada función de hacer eco. */
12512 if (tp->tty_termios.c_iflag & (ECHOIECHONL)) ch = echo(tp, ch);
12513
12514 /* Guardar el carácter en la cola de entrada. */

```

```

12515     *tp->tty_inhead++ = ch;
12516     if (tp->tty_inhead == bufend(tp->tty_inbuf))
12517         tp->tty_inhead = tp->tty_inbuf;
12518         tp->tty_inoount++;
12519     if (ch & IN_EOT) tp->tty_eotct++;
12520
12521     /* Tratar de terminar entradas si la cola amenaza desbordarse.          */
12522     if (tp->tty_incount == buflen(tp->tty_inbuf)) in_transfer(tp);
12523 }
12524 return ct;
12525 }

12528 /*=====
12529 *                                     echo                                     *
12530 *=====*/
12531 PRIVATE int echo(tp, ch)
12532     register tty_t *tp; /* Terminal para el eco */
12533     register int ch; /* apunto al carácter para eco */
12534 {
12535     /* Hacer eco del carácter si el eco está activo. Algunos caracteres de control
12536 * hacen eco con su efecto normal, otros hacen eco como "AX", los caracteres
12537 * normales aparecen normalmente. Se hace eco de EOF (AD), pero seguido
12538 * de retrocesos. Devolver el carácter con la longitud del eco agregada
12539 * a sus atributos.
12540 */
12541     int len, rp;
12542
12543     ch &= -IN_LEN;
12544     if (! (tp->tty_termios.c_lflag & ECHO)) {
12545         if (ch == ('\n' | IN_EOT) && (tp->tty_termios.c_lflag
12546             & (ICANONIECHONL)) == (ICANONIECHONL))
12547             (*tp->tty_echo) (tp, '\n');
12548         return(ch);
12549     }
12550
12551     /* "Reprint" dice si otras salidas arruinaron la salida de eco. */
12552     rp = tp->tty_incount == 0 ? FALSE : tp->tty_reprint;
12553
12554     if ((ch & IN_CHAR) < ' ') {
12555         switch (ch & (IN_ESCHIN_EOFIIN_EOTIIN_CHAR)) {
12556             case '\t':
12557                 len = 0;
12558                 do {
12559                     (*tp->tty_echo) (tp, , ');
12560                     len++;
12561                 } while (len < TAB_SIZE && (tp->tty_position & TAB_MASK) != 0);
12562                 break;
12563             case '\r' | IN_EOT:
12564             case '\n' | IN_EOT:
12565                 (*tp->tty_echo) (tp, ch & IN_CHAR);
12566                 len = 0;
12567                 break;
12568             default:
12569                 (*tp->tty_echo) (tp, 'A');
12570                 (*tp->tty_echo) (tp, '@' + (ch & IN_CHAR));
12571                 len = 2;
12572         }
12573     } else
12574     if ((ch & IN_CHAR) == '\177') {

```

```

12575     /* Un DEL aparece como ""7". */
12576     (*tp->tty_echo) (tp, "");
12577     (*tp->tty_echo) (tp, '7');
12578     len = 2;
12579 } else {
12580     (*tp->tty_echo) (tp, ch & IN_CHAR);
12581     len = 1;
12582 }
12583 if (ch & IN_EOF) while (len > 0) { (*tp->tty_echo) (tp, '\b'); len--; }
12584
12585 tp->tty_reprint = rp;
12586 return(ch | (len << IN_LSHIFT));
12587 }

12590 /*=====
12591 *                               rawecho                               *
12592 *=====*/
12593     PRIVATE void rawecho(tp, ch)
12594     register tty_t *tp;
12595     int ch;
12596     {
12597     /* Eco sin interpretación si ECHO está encendido. */
12598     int rp = tp->tty_reprint;
12599     if (tp->tty_termios.c_lflag & ECHO) (*tp->tty_echo) (tp, ch);
12600     tp->tty_reprint = rp;
12601     }

12604 /*=====
12605 *                               back over                               *
12606 *=====*/
12607     PRIVATE int back_over(tp)
12608     register tty_t *tp;
12609     {
12610     /* Retroceder al carácter anterior en la pantalla y borrarlo. */
12611     u16_t *head;
12612     int len;
12613
12614     if (tp->tty_incount == 0) return(0);          /* cola vacía */
12615     head = tp->tty_inhead;
12616     if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
12617     if (*--head & IN_EOT) return(0);             /* imposible borrar "saltos" */
12618     if (tp->tty_reprint) reprint(tp);            /* reimprimir si arruinada */
12619     tp->tty_inhead = head;
12620     tp->tty_incount--;
12621     if (tp->tty_termios.c_lflag & ECHOE) {
12622         len = (*head & IN_LEN) » IN_LSHIFT;
12623         while (len > 0) {
12624             rawecho(tp, '\b');
12625             rawecho(tp, ' ');
12626             rawecho(tp, '\b');
12627             len--;
12628         }
12629     }
12630     return(1);          /* un carácter borrado */
12631     }

12634 /*=====

```

```
12635 *                               reprint                               *
12636 *=====*/
12637 PRIVATE void reprint(tp)
12638 register tty_t *tp;                /* apuntador a struct tty */
12639 {
12640 /* Restaurar el eco anterior a la pantalla si las salidas arruinaron las entradas
12641  * del usuario o si se tecleó REPRINT (AR).
12642  */
12643 int count;
12644 u16_t *head;
12645
12646 tp->tty_reprint = FALSE;
12647
12648 /* Encontrar último salto de línea en la entrada. */
12649 head = tp->tty_inhead;
12650 count = tp->tty_incount;
12651 while (count > 0) {
12652     if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
12653     if (head[-1] & IN_EOT) break;
12654     head--;
12655     count--;
12656 }
12657 if (count == tp->tty_incount) return;          /* ninguna razón p/reimprimir */
12658
12659 /* Mostrar REPRINT (AR) Y pasar a nueva línea. */
12660 (void) echo(tp, tp->tty_termios.c_cc[VREPRINT] | IN_ESC);
12661 rawecho(tp, '\r');
12662 rawecho(tp, '\n');
12663
12664 /* Reimprimir desde el último corte en adelante. */
12665 do {
12666     if (head == bufend(tp->tty_inbuf)) head = tp->tty_inbuf;
12667     *head = echo(tp, *head);
12668     head++;
12669     count++;
12670 } while (count < tp->tty_incount);
12671 }

12674 /*=====*/
12675 *                               out_process                               *
12676 *=====*/
12677 PUBLIC void out_process(tp, bstart, bpos, bend, icount, ocount)
12678 tty_t *tp;
12679 char *bstart, *bpos, *bend; /* inic/pos/fin de buffer circular */
12680 int *icount; /* # cars entrada / cars entrada usados */
12681 int *ocount; /* máx. cars salida / cars salida usados */
12682 {
12683 /* Procesar salidas en buffer circular. *icount es el núm. de bytes por procesar,
12684 * Y los bytes realmente procesados al regresar. *ocount es el espacio
12685 * disponible en entradas y el usado en salidas. (Naturalmente, *icount
12686 * < *ocount.) La posición de columna se actualiza módulo al tamaño
12687 * de TAB, porque realmente sólo la necesitamos para tabulaciones.
12688  */
12689
12690 int tablen;
12691 int ict = *icount;
12692 int oct = *ocount;
12693 int pos = tp->tty_position;
12694
```

```

12695 while (ict > 0) {
12696     switch (*bpos) {
12697     case '\7':
12698         break;
12699     case '\b':
12700         pos_o;
12701         break;
12702     case '\r':
12703         pos = 0;
12704         break;
12705     case '\n':
12706         if ((tp->tty_termios.c_oflag & (OPOSTIONLCR))
12707             == (OPOSTIONLCR)) {
12708             /* Convertir LF en CR+LF si hay espacio.
12709              * Se reescribe el siguiente carácter del buffer,
12710              * así que nos detenemos aquí.
12711              */
12712             if (oct >= 2) {
12713                 *bpos = '\r';
12714                 if (++bpos == bend) bpos = bstart;
12715                 *bpos = '\n';
12716                 pos = 0;
12717                 ict--;
12718                 oct -= 2;
12719             }
12720             goto out_done; /* no hay espacio o buffer cambió */
12721         }
12722         break;
12723     case '\t':
12724         /* Mejor estimación de la long. de tabulación. */
12725         tablen = TA8_SIZE -(pos & TAB_MASK);
12726
12727         if ((tp->tty_termios.c_oflag & (OPOSTIXTABS))
12728             == (OPOSTIXTABS)) {
12729             /* Las tabulaciones deben expandirse. */
12730             if (oct >= tablen) {
12731                 pos += tablen;
12732                 ict--;
12733                 oct -= tablen;
12734                 do {
12735                     *bpos = ' ';
12736                     if (++bpos == bend) bpos = bstart;
12737                 } while (--tablen != 0);
12738             }
12739             goto out_done;
12740         }
12741         /* Las tabulaciones se envían directamente a la salida. */
12742         pos += tablen;
12743         break;
12744     default:
12745         /* Suponer que los demás cars. se exhiben como 1 carácter. */
12746         pos++;
12747     }
12748     if (++bpos == bend) bpos = bstart;
12749     ict--;
12750     oct--;
12751 }
12752 out_done:
12753     tp->tty_position = pos & TAB_MASK;
12754

```

```

12755 *icount -= ict;          /* [io]ct son núms. de cars. no usados */
12756 *ocount -= oct;        /* *[io]count son núms. de cars. usados */
12757 }

12760 /*=====
12761 *                               dev_ioctl                               *
12762 *=====*/
12763 PRIVATE void dev_ioctl(tp)
12764     tty_t *tp;
12765 {
12766     /* TCSETSW, TCSETSF y TCDRAIN de ioctl esperan que las salidas
12767 * terminen para asegurarse que un cambio de atrib. no afecte el procesam.
12768 * de salidas actuales. Ya terminadas, el ioctl se ejecuta como en do_ioctl().
12769 */
12770     phys_bytes user_phys;
12771
12772     if (tp->tty_outleft > 0) return;          /* no han terminado las salidas */
12773
12774     if (tp->tty_ireq != TCDRAIN) {
12775         if (tp->tty_ireq == TCSETSF) tty_icancel(tp);
12776         user_phys = proc_vir2phys(proc_addr(tp->tty_ioproc), tp->tty_iovir);
12777         phys_copy(user_phys, vir2phys(&tp->tty_termios),
12778                 (phys_bytes) sizeof(tp->tty_termios));
12779         setattr(tp);
12780     }
12781     tp->tty_ireq = 0;
12782     tty_reply(REVIVE, tp->tty_iocaller, tp->tty_ioproc, OK);
12783 }

12786 /*=====
12787 *                               setattr                               *
12788 *=====*/
12789 PRIVATE void setattr(tp)
12790     tty_t *tp;
12791 {
12792     /* Aplicar nuevos atrib. de línea (crudo/canónico, velo de línea, etc.) */
12793     u16_t *inp;
12794     int count;
12795
12796     if (! (tp->tty_termios.c_lflag & ICANON)) {
12797         /* Modo crudoj agregar "salto de línea" a cada carácter de cola de entrada.
12798 * No se define qué sucede con la cola sí ICANON se apaga; los procs.
12799 * deben usar TCSAFLUSH para vaciar la cola. Sin embargo, lo correcto
12800 * es guardar la cola para preservar el tecleo adelantado cuando un proceso
12801 * usa TCSANOW para cambiar a modo crudo.
12802 */
12803         count = tp->tty_eotct = tp->tty_ccount;
12804         inp = tp->tty_inbuf;
12805         while (count > 0) {
12806             *inp |= IN_EOT;
12807             if (++inp == bufend(tp->tty_inbuf)) inp = tp->tty_inbuf;
12808             --count;
12809         }
12810     }
12811
12812     /* Examinar MIN y TIME. */
12813     lock();
12814     settimer(tp, FALSE);

```

```

12815     unlock();
12816     if (tp->tty_termios.c_lflag & ICANON) {
12817         /* No MIN ni TIME en modo canónico. */
12818         tp->tty_min = 1;
12819     } else {
12820         /* En modo crudo, MIN es núm. de caracteres deseado, y TIME cuánto
12821          * esperarlos. Con excepciones interesantes si cualquiera es cero.
12822          */
12823         tp->tty_min = tp->tty_termios.c_cc[VMIN];
12824         if (tp->tty_min == 0 && tp->tty_termios.c_cc[VTIME] > 0)
12825             tp->tty_min = 1;
12826     }
12827
12828     if (!(tp->tty_termios.c_iflag & IXON)) {
12829         /* Sin control de salidas start/stop, así que no dejar salidas inhibidas. */
12830         tp->tty_inhibited = RUNNING;
12831         tp->tty_events = 1;
12832     }
12833
12834     /* Poner en cero la velocidad de salida cuelga el teléfono. */
12835     if (tp->tty_termios.c_ospeed == 80) sigchar(tp, SIGHUP);
12836 '
12837 /* Fijar nueva velo de línea, tamaño car., etc. en nivel de dispositivo. */
12838 (*tp->tty_ioctl) (tp);
12839 }

12842 /*=====
12843 *                                     tty_reply                                     *
12844 *=====*/
12845     PUBLIC void tty_reply(code, replyee, proc_nr, status)
12846         int code;          /* TASK_REPLY o REVIVE */
12847         int replyee;       /* dir. de destino p/respuesta */
12848         int proc_nr;       /* ¿a quién debe dirigirse la resp.? */
12849         int status;        /* código de respuesta */
12850     {
12851         /* Enviar respuesta al proceso que quería leer o escribir datos. */
12852
12853         message tty_mess;
12854
12855         tty_mess.m_type = code;
12856         tty_mess.REP_PROC_NR = proc_nr;
12857         tty_mess.REP_STATUS = status;
12858         if ((status = send(replyee, &tty_mess)) != OK)
12859             panic("tty_reply failed, status\n", status);
12860     }

12863 /*=====
12864 *                                     sigchar                                     *
12865 *=====*/
12866     PUBLIC void sigchar(tp, sig)
12867         register tty_t *tp;
12868         int sig; /* SIGINT, SIGQUIT, SIGKILL o SIGHUP */
12869     {
12870         /* Procesar 1 caro SIGINT, SIGQUIT o SIGKILL del teclado o SIGHUP
12871          * de cierre de tty, "stty 0" o cuando se cuelga realmente una línea RS-232.
12872          * MM enviará la señal al grupo de procesos (INT, QUIT), a todos los procesos
12873          * (KILL) o al líder de sesión (HUP).
12874          */

```



```
12875
12876 if (tp->tty_pgrp != 0) cause_sig(tp->tty_pgrp, sig);
12877
12878 if (1 (tp->tty_termios.c_lflag & NOFLSH)) {
12879     tp->tty_incount = tp->tty_eotct = 0;          /* matar entradas anteriores */
12880     tp->tty_intail = tp->tty_inhead;
12881     (*tp->tty_ocancel) (tp);                      /* matar todas las salidas */
12882     tp->tty_inhibited = RUNNING;
12883     tp->tty_events = 1;
12884 }
12885 }

12888 /*=====
12889 *                               tty_icancel                               *
12890 *=====*/
12891 PRIVATE void tty_icancel(tp)
12892 register tty_t *tp;
12893 {
12894     /* Desechar todas las entradas pendientes, buffer tty o dispositivo. */
12895
12896     tp->tty_incount = tp->tty_eotct = 0;
12897     tp->tty_intail = tp->tty_inhead;
12898     (*tp->tty_icancel) (tp);
12899 }

12902 /*=====
12903 *                               tty_init                               *
12904 *=====*/
12905 PRIVATE void tty_init(tp)
12906     tty_t *tp;          /* Línea TTY por inicializar. */
12907 {
12908     /* Inic. estructura tty e invocar rutinas de inic. de dispositivo. */
12909
12910     tp->tty_intail = tp->tty_inhead = tp->tty_inbuf;
12911     tp->tty_min = 1;
12912     tp->tty_termios = termios_defaultS;
12913     tp->tty_icancel = tp->tty_ocancel = tp->tty_ioctl = tp->tty_close =
12914         tty_devnop;
12915     if (tp < tty_addr(NR_CONS)) {
12916         scr_init(tp);
12917     } else
12918     if (tp < tty_addr(NR_CONS+NR_RS_LINES)) {
12919         rs_init(tp);
12920     } else {
12921         pty_init(tp);
12922     }
12923 }

12926 /*=====
12927 *                               tty_wakeup                               *
12928 *=====*/
12929 PUBLIC void tty_wakeup(now)
12930     clock_t now;      /* tiempo actual */
12931 {
12932     /* Despertar TTY si algo interesante está sucediendo en una de las líneas
12933     * de terminal, como la llegada de un carácter por una línea RS232, la pulsación
12934     * de una tecla o la expiración de un temporiz. en una línea por TIME.
```

```

12935     */
12936 tty_t *tp;
12937
12938 /* Buscar en timerlist temporiz. expirados y calcular sigo plazo vencido. */
12939 tty_timeout = TIME_NEVER;
12940 while ((tp = tty_timelist) != NULL) {
12941     if (tp->tty_time > now) {
12942         tty_timeout = tp->tty_time;           /* sigue este temporizador */
12943         break;
12944     }
12945     tp->tty_min = 0;                          /* forzar lectura con éxito */
12946     tp->tty_events = 1;
12947     tty_timelist = tp->tty_timenext;
12948 }
12949
12950 /* Avisar a TTY que algo está sucediendo. */
12951 interrupt(TTY);
12952 }

12955 /*=====
12956 *                               settimer                               *
12957 *=====*/
12958 PRIVATE void settimer(tp, on)
12959     tty_t *tp;           /* línea para poner o quitar un temporiz. */
12960     int on; /* poner temporiz. si true, si no, quitar */
12961 {
12962     /* Poner o quitar un temporiz. según TIME. Esta tunco es sensible a ints. a cau
12963 * de tty_wakeup(), así que debe invocarse desde dentro de lock()/unlock().
12964 */
12965     tty_t **ptp;
12966
12967     /* Sacar tp de timerlist si está presente. */
12968     for (ptp = &tty_timelist; *ptp != NULL; ptp = &>(*ptp)->tty_timenext) {
12969         if (tp == *ptp) {
12970             *ptp = tp->tty_timenext;       /* sacar tp de la lista */
12971             break;
12972         }
12973     }
12974     if (!on) return;                       /* basta con quitarlo */
12975
12976     /* El plazo vence TIME decise segundos a partir de ahora. */
12977     tp->tty_time = get_uptime() + tp->tty_termios.c_cc[VTIME] * (HZ/10);
12978
12979     /* Encontrar un nuevo lugar en la lista. */
12980     for (ptp = &tty_timelist; *ptp != NULL; ptp = &>(*ptp)->tty_timenext) {
12981         if (tp->tty_time <= (*ptp)->tty_time) break;
12982     }
12983     tp->tty_timenext = *ptp;
12984     *ptp = tp;
12985     if (tp->tty_time < tty_timeout) tty_timeout = tp->tty_time;
12986 }

12989 /*=====
12990 *                               tty_devnop                               *
12991 *=====*/
12992 PUBLIC void tty_devnop(tp)
12993     tty_t *tp;
12994 {

```

```
12995     /* Algunas funcns. no tienen que implementarse en nivel de dispositivo. */
12996     }
```

```
+++++
src/kernel/keyboard.c
+++++
```

```
13000 /* Controlador del teclado para PC y AT.
13001  *
13002  * Modificado por Marcus Hampel      (04/02/1994)
13003  * - Mapas de teclas cargables
13004  */
13005
13006     #include "kernel.h"
13007     #include <termios.h>
13008     #include <signal.h>
13009     #include <unistd.h>
13010     #include <minix/callnr.h>
13011     #include <minix/com.h>
13012     #include <minix/keymap.h>
13013     #include "tty.h"
13014     #include "keymaps/us-std.src"
13015
13016     /* Teclado estándar y AT. (PS/2 MCA implica AT siempre.) */
13017     #define KEYBD      0x60 /* puerto de E/S p/datos del teclado */
13018
13019     /* Teclado AT. */
13020     #define KB_COMMAND 0x64 /* puerto de E/S p/comandos en AT */
13021     #define KB_GATE_A20 0x02 /* bit en puerto salida p/habil. línea A20 */
13022     #define KB_PULSE_OUTPUT 0xF0 /* base p/comandos p/pulsar puerto
salida */
13023     #define KB_RESET    0x01 /* bit en puerto salida p/restab. CPU */
13024     #define KB_STATUS   0x64 /* puerto de E/S p/situación en AT */
13025     #define KB_ACK      0xFA /* respuesta ack del teclado */
13026     #define KB_BUSY     0x02 /* bit de situac. 1 si puerto KEYBD listo */
13027     #define LED_CODE    0xED /* comando a teclado p/activar LEDs */
13028     #define MAX_KB_ACK_RETRIES 0x1000 /* máx. veces esperar
acuse de teclado */
13029     #define MAX_KB_BUSY_RETRIES 0x1000 /* máx. ciclos
mientras teclado ocupado */
13030     #define KBIT        0x80 /* bit p/acuse de cars. al teclado */
13031
13032     /* Diversos. */
13033     #define ESC_SCAN    1 /* tecla p/rearranque si hay pánico */
13034     #define SLASH_SCAN  53 /* p/reconocer diagonal numérica */
13035     #define HOME_SCAN   71 /* 1a. tecla de subteclado num. */
13036     #define DEL_SCAN    83 /* DEL p/usar en rearranque CTRL-ALT-DEL */
13037     #define CONSOLE     0 /* número de línea p/consola */
13038     #define MEMCHECK_ADR 0x472 /* dir. p/detener verif. memo en rearranque */
13039     #define MEMCHECK_MAG 0x1234 /* núm. mágico p/detener verif. de
memo */
13040
13041     #define kb_addr()    (&kb_lines[0]) /* sólo hay un teclado */
13042     #define KB_IN_BYTES 32 /* tamaño de buffer entrada de teclado */
13043
13044     PRIVATE int alt1; /* estado de tecla alt izquierda */
13045     PRIVATE int alt2; /* estado de tecla alt derecha */
13046     PRIVATE int capslock; /* estado de tecla Bloq Mayús */
13047     PRIVATE int esc; /* ¿Código de escape detectado? */
13048     PRIVATE int control; /* estado de tecla de control */
13049     PRIVATE int caps_off; /* 1 = pos. normal, 0 = oprimida */
```

```

13050             PRIVATE int numlock; /* estado de tecla Bloq Núm */
13051             PRIVATE int num_off; /* 1 = pos. normal, 0 = oprimida */
13052             PRIVATE int slock; /* estado de tecla Bloq Despl */
13053             PRIVATE int slock_off; /* 1 = pos. normal, 0 = oprimida */
13054             PRIVATE int shift; /* estado de tecla shift */
13055
13056             PRIVATE char numpad_map[ ] =
13057                 { 'H', 'Y', 'A', 'B', 'D', 'C', 'V', 'U', 'G', 'S', 'T', '@' };
13058
13059             /* Estructura de teclado, una por consola. */
13060             struct kb_s {
13061                 char *ihead; /* sigo lugar libre en buffer entrada */
13062                 char *itail; /* cód. detección p/devolver a TTY */
13063                 int icount; /* núm. de códigos en buffer */
13064                 char ibuf[KB_IN_BYTES]; /* buffer de entrada */
13065             };
13066
13067             PRIVATE struct kb_s kb_lines[NR_CDNS];
13068
13069             FORWARD_PROTOTYPE( int kb_ack, (void) );
13070             FORWARD_PROTOTYPE( int kb_wait, (void) );
13071             FORWARD_PROTOTYPE( int func_key, (int scode) );
13072             FORWARD_PROTOTYPE( int scan_keyboard, (void) );
13073             FORWARD_PROTOTYPE( unsigned make_break, (int scode) );
13074             FORWARD_PROTOTYPE( void set_leds, (void) );
13075             FORWARD_PROTOTYPE( int kbd_hw_int, (int irq) );
13076             FORWARD_PROTOTYPE( void kb_read, (struct tty *tp) );
13077             FORWARD_PROTOTYPE( unsigned map_key, (int scode) );
13078
13079
13080 /*=====
13081 *                               map_key0                               *
13082 *=====*/
13083             /* Transformar código de detección a cód. ASCII ignorando modificadores. */
13084             #define map_key0(scode);
13085                 ((unsigned) keymap[(scode) * MAP_COLS])
13086
13087
13088 /*=====
13089 *                               map_key                               *
13090 *=====*/
13091             PRIVATE unsigned map_key(scode)
13092                 int scode;
13093             {
13094                 /* Transformar un código de detección en código ASCII. */
13095
13096                 int caps, column;
13097                 ul6_t *keyrow;
13098
13099                 if (scode == SLASH_SCAN && esc) return '/'; /* no transf. diagonal num. */
13100
13101                 keyrow = &keymap[scode * MAP_COLS];
13102
13103                 caps = shift;
13104                 if (numlock && HOME_SCAN <= scode && scode <= DEL_SCAN) caps = IcapS;
13105                 if (capslock && (keyrow[0] & HASCAPS)) caps = !caps;
13106
13107                 if (alt1 || alt2) {
13108                     column = 2;
13109                     if (control 11 alt2) column = 3; /* Ctrl + Alt1 == Alt2 */
13057

```

```

13110         if (caps) column = 4;
13111     } else {
13112         column = 0;
13113         if (caps) column = 1;
13114         if (control) column = 5;
13115     }
13116     return keyrow[column] & -HASCAPS;
13117 }

13120 I=====*
13121 *                               kbd_hw_int                               *
13122 *=====*I
13123 PRIVATE int kbd_hw_int(irq)
13124 int irq;
13125 {
13126     I* Ocurrió una interrupción de teclado. Procesarla. *I
13127
13128     int code;
13129     unsigned km;
13130     register struct kb_s *kb;
13131
13132     I* Obtener el carácter del hardware del teclado y acusar recibo. *I
13133     code = scan_keyboard();
13134
13135     I* El teclado IBM interrumpe 2 veces por tecla, una al oprimirse,
13136     * una al soltarse. Filtrar la segunda, ignorando todas las teclas menos las tipo
13137     * shift. Las teclas tipo shift 29, 42, 54, 56, 58 Y 69 se procesan normalmente.
13138     *I
13139
13140     if (code & 0200) {
13141         I* Se soltó una tecla (bit alto encendido). *I
13142         km = map_key0(code & 0177);
13143         if (km != CTRL && km != SHIFT && km != ALT && km != CALOCK
13144             && km != NLOCK && km != SLOCK && km != EXTKEY)
13145             return 1;
13146     }
13147
13148     I* Guardar carácter en memoria para que la tarea pueda usa;-iO después. *I
13149     kb = kb_addr();
13150     if (kb->icount < KB_IN_BYTES) {
13151         *kb->ihead++ = code;
13152         if (kb->ihead == kb->ibuf + KB_IN_BYTES) kb->ihead = kb->ibuf;
13153         kb->icount++;
13154         tty_table[current].tty_events = 1;
13155         force_timeout();
13156     }
13157     I* Si no cabe -desecharlo. *I
13158     return 1;          I* Rehabilitar interrupción de teclado *I
13159 }

13162 I=====*
13163 *                               kb_read                               *
13164 *=====*I
13165     PRIVATE void kb_read(tp)
13166     tty_t *tp;
13167     {
13168     I* Procesar caracteres del buffer circular del teclado. *I
13169

```

```

13170 struct kb_s *kb;
13171 char buf[3];
13172 int scode;
13173 unsigned ch;
13174
13175     kb = kb_addr();
13176     tp = &tty_table[current];          /* siempre usar consola actual */
13177
13178     while (kb->icount > 0) {
13179         scode = *kb->itail++;          /* tomar un cód. detec. de tecla */
13180         if (kb->itail == kb->ibuf + KB_IN_BYTES) kb->itail = kb->ibuf;
13181         lock();
13182         kb->icount--;
13183         unlock();
13184
13185         /* Se usan teclas de función para vaciados de depuración. */
13186         if (func_key(scode)) continue;
13187
13188         /* Realizar procesamiento make/break. */
13189         ch = make_break(scode);
13190
13191         if (ch <= 0xFF) {
13192             /* Un carácter normal. */
13193             buf[0] = ch;
13194             (void) in_process(tp, buf, 1);
13195         } else
13196         if (HOME <= ch && ch <= INSRT) {
13197             /* Secuencia de escape ASCII generada por subteclado num. */
13198             buf[0] = ESC;
13199             buf[1] = '[';
13200             buf[2] = numpad_map[ch - HOME];
13201             (void) in_process(tp, buf, 3);
13202         } else
13203         if (ch == ALEFT) {
13204             /* Escoger la consola de núm. más bajo como actual. */
13205             select_consOle(current - 1);
13206         } else
13207         if (ch == ARIGHT) {
13208             /* Escoger la consola de núm. más alto como actual. */
13209             select_conSole(current + 1);
13210         } else
13211         if (AF1 <= ch && ch <= AF12) {
13212             /* Alt-F1 es consola, Alt-F2 es tty1, etc. */
13213             select_console(ch - AF1);
13214         }
13215     }
13216 }

```

```

13219 /*=====
13220 *                               make_break                               *
13221 *=====*/
13222 PRIVATE unsigned make_break(scode)
13223 int scode;          /* cód. det. de tecla oprimida o soltada */
13224 {
13225     /* Esta rutina puede manejar teclados que sólo interrumpen cuando
13226     * se oprimen teclas, y también los que interrumpen al presionar y al soltar.
13227     * Por eficiencia, la rutina de int. elimina casi todas las liberaciones de tecla.
13228     */
13229 int ch, make;

```

```

13230 static int CAD_count = 0;
13231
13232 /* Detectar CTRL-ALT-DEL y, si se encuentra, detener la computadora.
13233 * Sería mejor hacer esto en keyboard() en caso de que tty esté colgada,
13234 * excepto que control y alt se establecen en el código de alto nivel.
13235 */
13236 if (control && (alt1 || alt2) && scode == DEL_SCAN)
13237 {
13238     if (++CAD_count == 3) wreboot(RBT_HALT);
13239     cause_sig(INIT_PROC_NR, SIGABRT);
13240     return -1;
13241 }
13242
13243 /* El bit de orden alto se enciende al soltar una tecla. */
13244 make = (scode & 0200 ? 0 : 1); /* 0 = soltar, 1 = oprimir */
13245
13246 ch = map_key(scode & 0177); /* correspondencia con ASCII */
13247
13248 switch (ch) {
13249     case CTRL:
13250         control = make;
13251         ch = -1;
13252         break;
13253     case SHIFT:
13254         shift = make;
13255         ch = -1;
13256         break;
13257     case ALT:
13258         if (make) {
13259             if (esc) alt2 = 1; else alt1 = 1;
13260         } else {
13261             alt1 = alt2 = 0;
13262         }
13263         ch = -1;
13264         break;
13265     case CALOCK:
13266         if (make && caps_off) {
13267             capslock = 1 - capslock;
13268             set_leds();
13269         }
13270         caps_off = 1 - make;
13271         ch = -1;
13272         break;
13273     case NLOCK:
13274         if (make && num_off) {
13275             numlock = 1 - numlock;
13276             set_leds();
13277         }
13278         num_off = 1 - make;
13279         ch = -1;
13280         break;
13281     case SLOCK:
13282         if (make & SloCk_Off) {
13283             slock = 1 - slock;
13284             set_leds();
13285         }
13286         sloCk_Off = 1 - make;
13287         ch = -1;
13288         break;
13289     case EXTKEY:

```

```

13290             esc = 1;
13291             return(-1);
13292     default:
13293         if (!make) ch = -1;
13294     }
13295     esc = 0;
13296     return(ch)
13297 }

13300 /*=====
13301 *                               set_leds                               *
13302 *=====*/
13303 PRIVATE void set_leds()
13304 {
13305     /* Activar LEOs en teclas Bloq Mayús y Bloq Num. */
13306
13307     unsigned leds;
13308
13309     if (!pc_at) return;          /* PC/XT no tiene LEOs */
13310
13311     /* codificar bits de LEO */
13312     leds = (slock << 0) | (numlock << 1) | (capslock << 2);
13313
13314     kb_wait();                  /* esperar que el buffer esté vacío */
13315     out_byte(KEYBO, LEO_.COOE); /* preparar teclados p/aceptar valores LEO */
13316     kb_ack();                   /* esperar respuesta de acuse */
13317
13318     kb_wait();                  /* esperar que el buffer esté vacío */
13319     out_byte(KEYBO, leds);      /* dar al teclado valores LEO */
13320     kb_ack();                   /* esperar respuesta de acuse */
13321 }

13324 /*=====
13325 *                               kb_wait                               *
13326 *=====*/
13327 PRIVATE int kb_wait()
13328 {
13329     /* Esperar que el controlador esté listo; devolver 0 si vence el plazo. */
13330
13331     int retries;
13332
13333     retries = MAX_KB_BUSY_RETRIES + 1;
13334     while (--retries != 0 && in_byte(KB_STATUS) & KB_BUSY)
13335         /* esperar que no esté ocupado */
13336         return(retries);      /* no 0 si está listo */
13337 }

13340 /*=====
13341 *                               kb_ack                               *
13342 *=====*/
13343 PRIVATE int kb_ack()
13344 {
13345     /* Esperar que tecl. acuse rec. últ. comando; devolver 0 si vence plazo. */
13346
13347     int retries;
13348
13349     retries = MAX_KB_ACK_RETRIES + 1;

```



```

13350     while (--retries != 0 && in_byte(KEYBD) != KB_ACK)
13351         ;                               /* esperar acuse */
13352     return(retries);                       /* no 0 si se recibió acuse */
13353 }

13356 /*=====
13357 *                               kb init                               *
13358 *=====*/
13359 PUBLIC void kb_init(tp)
13360     tty_t *tp;
13361 {
13362     /* Inicializar el controlador en software del teclado. */
13363
13364     register struct kb_s *kb;
13365
13366     /* Función de entrada. */
13367     tp->ttY_devread = kb_read;
13368
13369     kb = kb_addr();
13370
13371     /* Preparar cola de entrada. */
13372     kb->ihead = kb->itail = kb->ibuf;
13373
13374     /* Establecer valores iniciales. */
13375     caps_off = 1;
13376     num_off = 1;
13377     sloCk_Off = 1;
13378     esc = 0;
13379
13380     set_leds();                               /* apagar led de Bloq Num */
13381
13382     scan_keyboard();                           /* detener bloqueo de digit. remanente */
13383
13384     put_irq_handler(KEYBOARD_IRO, kb_hw_int); /* estab. manejador ints. */
13385     enable_irq(KEYBOARD_IRO)j                 /* todo inicializado: no hay peligro*/
13386 }

13389 /*=====
13390 *                               kbd_loadmap                           *
13391 *=====*/
13392 PUBLIC int kbd_loadmap(user_Phys)
13393     phys_bytes user_phys;
13394 {
13395     /* Cargar un nuevo mapa de teclas. */
13396
13397     Phys_copy(user_phys, vir2phys(keymap), (phys_bytes) sizeof(keymap));
13398     return(OK);
13399 }

13402 /*=====
13403 *                               func_key                               *
13404 *=====*/
13405 PRIVATE int func_key(scode)
13406     int scode;                               /* cód. de detección de tecla de función */
13407 {
13408     /* Este procedimiento atrapa teclas de tunco p/fines de depuro y control. */
13409

```

```

13410 unsigned code;
13411
13412 code = map_key0(scode);                /* 10. ignorar modificadores */
13413 if (code < F1 || code > F12) return(FALSE); /* no nos toca */
13414
13415 switch (map_key(scode)) {                /* incluir modificadores */
13416
13417 case F1:      p_dmp(); break;             /* imprimir tabla de procesos */
13418 case F2:      map_dmp(); break;          /* imprimir mapa de memoria */
13419 case F3:      toggle_scroll(); break;    /* despl. pant. por har. vs. sof. */
13420 case CF7:     sigchar(&tty_table[CONSOLE], SIGQUIT); break;
13421 case CF8:     sigchar(&tty_table[CONSOLE], SIGINT); break;
13422 case CF9:     sigchar(&tty_table[CONSOLE], SIGKILL); break;
13423 default:     return(FALSE);
13424 }
13425 return(TRUE);
13426 }

13429 /*=====*/
13430 *                scan_keyboard                *
13431 *=====*/
13432 PRIVATE int scan_keyboard()
13433 {
13434     /* Obtener el carácter del hardware de teclado y acusar recibo. */
13435
13436     int codej;
13437     int val;
13438
13439     code = in_byte(KEYBD);                /* obt. cód. det. de tecla oprimida */
13440     val = in_byte(PORT_B);                /* strobe al teclado p/ack el caro */
13441     out_byte(PORT_B, val | KBIT); /* subir estrobosc. el bit */
13442     out_byte(PORT_B, val);                /* bajar estrobosc. el bit */
13443     return code;
13444 }

13447 /*=====*/
13448 *                wreboot                *
13449 *=====*/
13450     PUBLIC void wreboot(how)
13451                 int how;                /* 0 = alto, 1 = rearranque, 2 = pánico, ... */
13452     {
13453         /* Esperar digitaciones para imprimir info de depuración y rearrancar. */
13454
13455         int quiet, code;
13456         static u16_t magic = MEMCHECK_MAG;
13457         struct tasktab *ttp;
13458
13459         /* Enmascarar todas las interrupciones. */
13460         out_byte(INT_CTLMASK, -0);
13461
13462         /* Decir a varias tareas que se detengan. */
13463         cons_stop();
13464         floppy_stop();
13465         clock_stop();
13466
13467         if (how == RBT_HALT) {
13468             printf("System Halted\n");
13469             if (!mon_return) how = RBT_PANIC;

```

```

13470     }
13471
13472 if (how == RBT_PANIC) {
13473     /* ¡Un pánico! */
13474     printf("Hit ESC to reboot, F-keys for debug dumps\n");
13475
13476     (void) scan_keyboard(); /* acusar rec. de entradas viejas */
13477     quiet = scan_keyboard(); /* valor latente (0 en PC, últ. cód. en AT)*/
13478     for (; ;) {
13479         milli_delay(100);          /* pausa de 1 decisegundo */
13480         code = scan_keyboard();
13481         if (code != quiet) {
13482             /* Se oprimió una tecla. */
13483             if (code == ESC_SCAN) break; /* rearrancar si ESC */
13484             (void) func_key(code);      /* procesar tecla funGo */
13485             quiet = scan_keyboard();
13486         }
13487     }
13488     how = RBT_REBOOT;
13489 }
13490
13491 if (how == RBT_REBOOT) printf("Rebooting\n");
13492
13493 if (mon return && how != RBT_RESET) {
13494     /* Reinicializar controladores de ints. a valores de BIOS. */
13495     intr_init(0);
13496     out_byte(INT_CTLMASK, 0);
13497     out_byte(INT2_CTLMASK, 0);
13498
13499     /* Regresar al monitor de arranque. */
13500     if (how == RBT_HALT) {
13501         reboot_code = vir2phys("");
13502     } else
13503     if (how == RBT_REBOOT) {
13504         reboot_code = vir2phys("delaYjboot");
13505     }
13506     leve10(monitor);
13507 }
13508
13509 /* Detener prueba de memoria de BIOS. */
13510 phys copy(vir2phys(&magic), (phys_bytes) MEMCHECK_ADR,
13511          -(phys_bytes) sizeof(magic));
13512
13513 if (protected_mode) {
13514     /* Usar el controlador en hardware del teclado AT para restablecer
13515     * el procesador. La línea A20 se mantiene habilitada por si este código
13516     * se ejecuta desde memoria extendida y porque algunas máquinas
13517     * parecen subir la A20 falsa en vez de bajar la justo después del
13518     * restablecimiento, dando pie a una trampa de cód. de operación
13519     * ilegal. Este error es más problemático si está en uso la A20 falsa,
13520     * como sucedería si se usara el reset de teclado para modo real.
13521     */
13522     kb_wait();
13523     out_byte(KB_COMMAND,
13524             KB_PULSE_OUTPUT | (0x0F & ~(KB_GATE_A20 | KB_RESET)));
13525     milli_delay(10);
13526
13527     /* Si el método amable falla, restablecer. En modo protegido esto implica
13528     * un cese de funciones del procesador.
13529     */

```

```

13530         printf("Hard reset.. \n");
13531             milli~delay(250);
13532     }
13533     /* En modo real, basta con saltar a la dirección de restablecimiento. */
13534     leve10(reset);
13535 }

```

```

+++++
src/kernel/console.c
+++++

```

```

13600     /* Código y datos para el controlador de consola IBM.
13601     *
13602     * El controlador de video 6845 empleado por IBM PC comparte su memo de video
13603 * con la CPU en algún lugar del banco 0xB0000. Para el 6845 esta memoria
13604 * consta de palabras de 16 bits. Cada una tiene un cód. de carácter en el byte bajo
13605 * Y un byte de atributo en el alto. La CPU modifica directamente la memo de video
13606 * p/exhibir caracteres, y fija dos registros del 6845 que especific. el origen
13607 * del video y la pos. del cursor. El origen es el lugar de la memo
13608 * de video donde está el 1er. carácter (esquina supo izq.) Mover el origen
13609 * es una forma rápida de desplazar la pantalla. Algunos adaptadores
13610 * vinculan el final con el principio de la memo de video, y no hay límite
13611 * para el mov. del origen. En otros a veces hay que mover la memo de video para
13612 * restab. el origen. Todos los cálculos en memo de video usan direcciones
13613 * de caro (palabra) por sencillez y suponen que no hay continuidad. Las func.
13614 * de apoyo traducen las direcciones de palabra a dirs. de byte y la tunco de
13615 * despl. se preocupa por la continuidad.
13616     */
13617
13618     #include "kernel.h"
13619     #include <termios.h>
13620     #include <minix/callnr.h>
13621     #include <minix/com.h>
13622     #include "protect.h"
13623     #include "tty.h"
13624     #include "proc.h"
13625
13626     /* Definiciones empleadas por el controlador de la consola. */
13627     #define MONO_BASE      0xB0000L      /* base de memoria de video mono */
13628     #define COLOR_BASE    0xB8000L      /* base de memoria de video de color */
13629     #define MONO_SIZE     0x1000        /* 4K memoria de video mono */
13630     #define COLOR_SIZE    0x4000        /* 16K memoria de video de color */
13631     #define EGA_SIZE      0x8000        /* EGA & VGA tienen al menos 32K */
13632     #define BLANK_COLOR   0x0700        /* determina color cursor en pant. vacía */
13633     #define SCROLL_UP     0             /* desplazar hacia adelante */
13634     #define SCROLL_DOWN   1            /* desplazar hacia atrás */
13635     #define BLANK~MEM ((u16_t *) 0) /* dice a mem_vid_copy que borre pant. */
13636     #define CONS_RAM_WORDS 80           /* tamaño buffer de ram de video */
13637     #define MAX_ESC_PARMS 2            /* # de paráms. permito en seco escape */
13638
13639     /* Constantes relacionadas con los chips controladores. */
13640     #define M_6845        0x3B4 /* puerto para 6845 mono */
13641     #define C_6845        0x3D4 /* puerto para 6845 color */
13642     #define EGA           0x3C4 /* puerto para tarjeta EGA o VGA */
13643     #define INDEXT        0      /* registro índice del 6845 */
13644     #define DATAT         1      /* registro de datos del 6845 */

```

```

13645         #define VID_ORG      12    /* registro de origen del 6845 */
13646         #define CURSaR      14    /* registro de cursor del 6845 */
13647
13648         /* Alarma sonora. */
13649         #define BEEP_FREQ    0x0533 /* valor p/temporiz.: frec. de bip */
13650         #define B_TIME      3      /* long. del bip CTRL-G en tics */
13651
13652         /* definiciones empleadas en la administración de fuentes */
13653         #define GA_SEQUENCER_INDEX 0x3C4
13654         #define GA_SEQUENCER_DATA 0x3C5
13655         #define GA_GRAPHICS_INDEX 0x3CE
13656         #define GA_GRAPHICS_DATA 0x3CF
13657         #define GA_VIDEO_ADDRESS 0xA000L
13658         #define GA_FONT_SIZE 8192
13659
13660         /* Variables globales empleadas por el controlador de consola. */
13661         PUBLIC unsigned vid_seg;      /* selector ram de video (0xB0000 o
0xB8000) */
13662         PUBLIC unsigned vid_size;     /* 0x2000 p/color o 0x0800 p/mono */
13663         PUBLIC unsigned vid_mask;     /* 0x1FFF p/color o 0x07FF p/mono */
13664         PUBLIC unsigned blank_color = BLANK_COLOR; /* cód. exhibo p/blanco */
13665
13666         /* Variables privadas empleadas por el controlador de consola. */
13667         PRIVATE int vid_port;         /* puerto E/S p/acceder a 6845 */
13668         PRIVATE int wrap;            /* ¿continuidad por hardware? */
13669         PRIVATE int softscroll;      /* 1 = despl. por sof., 0 = por hardware */
13670         PRIVATE unsigned vid_base;   /* base ram video (0xB000 o 0xB800) */
13671         PRIVATE int beeping;         /* ¿altavoz hace bip? */
13672         #define scr_width80          /* núm. caracteres/línea */
13673         #define scr_lines25          /* núm. líneas/pantalla */
13674         #define scr_size (80*25) /* núm. caracteres/pantalla */
13675
13676         /* Datos por consola. */
13677         typedef struct console {
13678             tty t *c tty;             /* struct TTY asociado */
13679             int c column;             /* núm. columna actual (0-origen) */
13680             int c rrow;              /* fila actual (0 arriba pantalla) */
13681             int c rwords;            /* núm WORDS (no bytes) en outqueue */
13682             unsigned c_start;        /* inicio de memo video esta consola */
13683             unsigned c_limit;        /* límite de memo video esta consola */
13684             unsigned c_org;          /* lugar en RAM donde apunta base 6845 */
13685             unsigned c_cur;          /* pos. actual de cursor en RAM video */
13686             unsigned c_attr;         /* atributo de carácter */
13687             unsigned c_blank;        /* atributo blanco */
13688             char c esc state;         /* 0=normal, 1=ESC, 2=ESC[ */
13689             char c-esc-intro;        /* carácter distintivo sigue a ESC */
13690             int *c-esc-parmp;        /* apunto a parám. de escape actual */
13691             int c_esc_parmv[MAX_ESC_PARMS]; /* lista paráms. escape */
13692             u16_t c_ramqueue[CONS_RAM_WORDS]; /* buffer p/RAM de video */
13693         } console_t;
13694
13695         PRIVATE int nr_cons= 1; /* número real de consolas */
13696         PRIVATE console_t cons_table[NR_CONS];
13697         PRIVATE console_t *curcons; /* actualmente visible */
13698
13699         /* Color si se está usando controlador de color. */
13700         #define color (vid_port == C_6845)
13701
13702         /* Transformar colores ANSI a los atributos usados por la PC */
13703         PRIVATE int ansi_colors[8] = {0, 4, 2, 6, 1, 5, 3, 7};
13704

```

```

13705 /* Estructura empleada para administración de fuentes */
13706 struct sequence {
13707     unsigned short index;
13708     unsigned char port;
13709     unsigned char value;
13710 };
13711
13712 FORWARD PROTOTYPE( void cons write, (struct tty *tp) );
13713 FORWARD =PROTOTYPE( void cons=echo, (tty_t *tp, int c) );
13714 FORWARD _PROTOTYPE( void out_char, (console_t *cons, int c) );
13715 FORWARD PROTOTYPE( void beep, (void) );
13716 FORWARD -PROTOTYPE( void do_escape, (console_t *cons, int c) );
13717 FORWARD -PROTOTYPE( void flush, (console_t *cons) );
13718 FORWARD =PROTOTYPE( void parse_escape, (Console_t *cons, int c) );
13719 FORWARD _PROTOTYPE( void scroll_screen, (console_t *cons, int dir) );
13720 FORWARD _PROTOTYPE( void set 6845, (int reg, unsigned val) );
13721 FORWARD =PROTOTYPE( void stop_beep, (void) );
13722 FORWARD PROTOTYPE( void cons org0, (void) );
13723 FORWARD =PROTOTYPE( void ga_program, (struct sequence *seq) );
13724
13725
13726 /*=====
13727 *                               cons_write                               *
13728 *=====*/
13729 PRIVATE void cons write(tp)
13730 register struct tty *tp;          /* dice cuál terminal debe usarse */
13731 {
13732     /* Copiar tantos datos como sea posible en cola de salida, e iniciar E/S. En
13733     * terminales con mapa en memoria, como la consola I8M, la E/S también se termina y
13734     * se actualizan las cuentas. Seguir repitiendo hasta terminar toda la E/S.
13735     */
13736
13737     int count;
13738     register char *tbuf;
13739     char bUf[64];
13740     phys_bytes user_phys;
13741     console_t *cons = tp->tty_priv;
13742
13743     /* Verificar rápidamente que no haya nada que hacer, para que ésta pueda invocarse
13744     * a menudo sin pruebas no modulares en otros puntos.
13745     */
13746     if ((count = tp->tty_outleft) == 0 || tp->tY_inhibited) return;
13747
13748     /* Copiar bytes de usuario en buf[] para un direccionamiento decente. Iterar las
13749     * copias, ya que el buffer de usuario podría ser mucho mayor que buf[ ].
13750     */
13751     do {
13752         if (count > sizeof(buf)) count = sizeof(buf);
13753         user_phys = proc_vir2Phys(proc_addr(tp->tty_outproc), tp->tty_out_vir);
13754         phys_copy(user_phys, vir2phys(buf), (phys_bytes) count);
13755         tbuf = buf;
13756
13757         /* Actualizar estructura de datos de terminal. */
13758         tp->tty_out_vir += count;
13759         tp->tty_outcum += count;
13760         tp->tty_outleft -= count;
13761
13762         /* Enviar cada byte de la copia a la pantalla. Evitar invocar out_char()
13763         * para los caracteres "fáciles"; colocarlos en el buffer
13764         * directamente.

```

```

13765         */
13766     do {
13767         if ((unsigned) *tbuf < ' ', 11 cons->c_esc_state > 0
13768             || cons->c_column >= scr_width
13769             || cons->c_rwords >= BUflen(cons->c_ramqueue)»
13770         {
13771             out_char(cons, *tbuf++);
13772         } else {
13773             cons->c_ramqueue[cons->c_rwords++j =
13774                 cons->c_attr I (*tbuf++ & BYTE);
13775             cons->c_column++;
13776         }
13777     } while (--count != 0);
13778 } while ((count = tp->tty_outleft) != 0 && !tp->tty_inhibited);
13779
13780 fluSh(cons);                /* transferir todo lo del buffer a la pantalla */
13781
13782 /* Re-sponder al escritor si terminaron todas las salidas. */
13783 if (tp->tty_outleft == 0) {
13784     tty_reply(tp->tty_outrepcOde, tp->tty_outcaller, tp->tty_outproc,
13785              tp->tty_outcum);
13786     tp->tty_outcum = 0;
13787 }
13788 }

13791 /*=====
13792 *                                     cons_echo *
13793 *=====*/
13794 PRIVATE void cons_eCho(tp, c)
13795 register tty_t *tp;                /* apuntador a struct tty */
13796 int c;                            /* caro del que se hará eco */
13797 {
13798     /* Hacer eco de entradas del teclado (imprimir y vaciar). */
13799     console_t *cons = tp->tty_priv;
13800
13801     out_char(cons, c);
13802     fluSh(cons);
13803 }

13806 /*=====
13807 *                                     out_char *
13808 *=====*/
13809 PRIVATE void out_char(cons, c)
13810 register console_t *cons;         /* apuntador a struct de consola */
13811 int c;                            /* carácter por exhibir */
13812 {
13813     /* Exhibir carácter en consola. Verifo primero secuencias de escape. */
13814     if (cons->c_esc_state > 0) {
13815         parse_escape(cons, c);
13816         return;
13817     }
13818
13819     switch(c) {
13820     case 000:                       /* null suele usarse como relleno */
13821         return;                     /* mejor no hacer nada */
13822
13823     case 007:                       /* sonar la campana */
13824         fluSh(cons); /* exhibo caracteres en cola p/salida */

```

```

13825         beep();
13826         return;
13827
13828     case '\b':           /* retroceso */
13829         if (--cons->c_column < 0) {
13830             if (--cons->c_row >= 0) cons->c_column += scr_width;
13831         }
13832         flush(cons);
13833         return;
13834
13835     case '\n':           /* salto de línea */
13836         if ((cons->c_tty->tty_termios.c_oflag & (OPOSTIONLCR))
13837             == (OPOSTIONLCR)) {
13838             cons->c_column = 0;
13839         }
13840         /*CONTINUAR CON RUTINA SIGUIENTE*/
13841     case 013:           /* CTRL-K */
13842     case 014:           /* CTRL-L */
13843         if (cons->c_row == scr_lines-1) {
13844             scroll_screen(cons, SCROLL_UP);
13845         } else {
13846             cons->c_row++;
13847         }
13848         flush(cons);
13849         return;
13850
13851     case '\r':           /* retorno de carro */
13852         cons->c_column = 0;
13853         flush(cons);
13854         return;
13855
13856     case '\t':           /* tab */
13857         cons->c_column = (cons->c_column + TAB_SIZE) & ~TAB_MASK;
13858         if (cons->c_column > scr_width) {
13859             cons->c_column -= scr_width;
13860             if (cons->c_row == scr_lines-1) {
13861                 scroll_screen(cons, SCROLL_UP);
13862             } else {
13863                 cons->c_row++;
13864             }
13865         }
13866         flush(cons);
13867         return;
13868
13869     case 033:           /* ESC -inicia secuencia de escape */
13870         flush(cons);     /* exhibo cars. en cola p/salida */
13871         cons->c_esc_state = 1;     /* marcar ESC como visto */
13872         return;
13873
13874     default:           /* cars. imprimibles se guardan en ramqueue */
13875         if (cons->c_column >= scr_width) {
13876             if (!LINEWRAP) return;
13877             if (cons->c_row == scr_lines-1) {
13878                 scroll_screen(cons, SCROLL_UP);
13879             } else {
13880                 cons->c_row++;
13881             }
13882             cons->c_column = 0;
13883             flush(cons);
13884         }

```



```

13885         if (cons->c_rwords == bUflen(cons->c_ramqueue)) flush(cons);
13886         cons->c_ramqueue[cons->c_rwords++] = cons->c_attr I (c & BYTE);
13887         cons->c_column++; /* siguiente columna */
13888         return;
13889     }
13890 }

13893 /*=====
13894 *                               scroll_screen                               *
13895 *=====*/
13896 PRIVATE void scroll_screen(cons, dir)
13897 register console_t *cons; /* apuntador a struct de consola */
13898 int dir; /* SCROLL_UP o SCROLL_DOWN */
13899 {
13900     unsigned new_line, new_org, chars;
13901
13902     flush(cons);
13903     chars = scr_size - scr_width; /* 1 pantalla menos 1 linea */
13904
13905     /* Desplazar la pantalla es una lata por tantas tarjetas de video incompatibles.
13906     * Este controlador apoya despl. p/software (¿Hércules?), por hardware
13907     * (tarjs. mono y CGA) y por hardware sin continuidad (EGA y VGA).
13908     * En el tercer caso hay que asegurarse que se cumpla
13909     * c_start <= c_org && c_org + scr_size <= c_limit
13910     * porque EGA y VGA no continúan del fin al prinGo de la memo de video.
13911     */
13912     if (dir == SCROLL_UP) {
13913         /* Despl. 1 línea arriba de 3 formas: soft, evitar cont., usar origen. */
13914         if (softscroll) {
13915             Vid_Vid_copy(cons->c_start + scr_width, cons->c_start, chars);
13916         } else
13917         if (!wrap && cons->c_org + scr_size + scr_width >= cons->c_limit) {
13918             Vid_Vid_copy(cons->c_org + scr_width, cons->c_start, chars);
13919             cons->c_org = cons->c_start;
13920         } else {
13921             cons->c_org = (cons->c_org + scr_width) & vid_mask;
13922         }
13923         new_line = (cons->c_org + chars) & vid_mask;
13924     } else {
13925         /* Despl. 1 línea abajo de 3 formas: soft, evitar cont., usar origen. */
13926         if (softscroll) {
13927             Vid_Vid_copy(cons->c_start, cons->c_start + scr_width, chars);
13928         } else
13929         if (!wrap && cons->c_org < cons->c_start + scr_width) {
13930             new_org = cons->c_limit - scr_size;
13931             Vid_Vid_copy(cons->c_org, new_org + scr_width, chars);
13932             cons->c_org = new_org;
13933         } else {
13934             cons->c_org = (cons->c_org - scr_width) & vid_mask;
13935         }
13936         new_line = cons->c_org;
13937     }
13938     /* Poner en blanco la nueva línea arriba o abajo. */
13939     blank_color = cons->c_blank;
13940     mem_vid_coPY(BLANK_MEM, new_line, scr_width);
13941
13942     /* Establecer el nuevo origen de video. */
13943     if (cons == curcons) set_6845 (VID_ORG, cons->c_org);
13944     flush(cons);

```

```

13945 }

13948 I*=====
13949 *                flush                *
13950 *=====*/
13951     PRIVATE void flu'sh (cons)
13952     register console_t *cons;          I* apuntador a struct de consola */
13953     {
13954     I* Enviar caracteres en 'ramqueue' a memo de pantalla, verif. nueva
13955     * pos. de cursar, calcular nueva pos. del cursar de hardware Y fijarla.
13956     */
13957     unsigned cur;
13958     tty_t *tp = cons->c_tty;
13959
13960     I* Transferir los caracteres de 'ramqueue' a la pantalla. */
13961     if (cons->c_rwords > 0) {
13962         mem_vid_copy(cons->c_ramqueue, cons->c_cur, cons->c_rwords);
13963         cons->c_rwords = 0;
13964
13965         I* TTY gusta de conocer col. actual y si el eco se arruinó. */
13966         tp->tty_position = cons->c_column;
13967         tp->tty_reprint = TRUE;
13968     }
13969
13970     I* Verificar Y actualizar la posición del cursar. */
13971     if (cons->c_column < 0) cons->c_column = 0;
13972     if (cons->c_column > scr_width) cons->c_column = scr_width;
13973     if (cons->c_row < 0) cons->c_row = 0;
13974     if (cons->c_row >= scr_lines) cons->c_row = scr_lines - 1;
13975     cur = cons->c_org + cons->c_row * scr_width + cons->c_column;
13976     if (cur != cons->c_cur) {
13977         if (cons == curcons) set_6845 (CURSOR, cur);
13978         cons->c_cur = cur;
13979     }
13980 }

13983 I*=====
13984 *                parse_escape                *
13985 *=====*/
13986     PRIVATE void parse_escape(cons, c)
13987     register console_t *cons;          I* apuntador a struct de consola */
13988     char C;                            I* signo carácter de secuencia escape */
13989     {
13990     I* Actualmente se reconocen las secuencias de escape ANSI siguientes:
13991     * Si se omiten n y/o m, valen 1. Si se omite s, vale 0.
13992     *   ESC [nA sube n líneas
13993     *   ESC [nB baja n líneas
13994     *   ESC [nC se mueve n espacios a la derecha
13995     *   ESC [nO se mueve n espacios a la izquierda
13996     *   ESC [m;nH mueve el cursar a (m,n)
13997     *   ESC [sJ borra pantalla rel. al cursar (0 al fin, 1 del inicio, 2 todo)
13998     *   ESC [sK borra línea rel. al cursar (0 al fin, 1 del inicio, 2 todo)
13999     *   ESC [nL inserta n líneas en el cursar
14000     *   ESC [nM elimina n líneas en el cursar
14001     *   ESC [nP elimina n caracteres en el cursar
14002     *   ESC [n@ inserta n caracteres en el cursar
14003     *   ESC [nm habilita estilo n (0=normal, 1=negra, 4=subray. 5=parpad,
14004     *           7=inverso, 30..37 color 1er. plano, 40..47 color 20. plano)

```



```

14065             break;
14066
14067         default: break;
14068     }
14069 } else
14070 if (cons->c_esc_intro == '[') {
14071     /* Manejar secuencia que comienza con ESC [ y parámetros */
14072     value = cons->c_esc_parmv[0];
14073     switch (c) {
14074         case 'A':             /* ESC [nA sube n lineas */
14075             n = (value == 0 ? 1 : value);
14076             cons->c_row -= n;
14077             fluSh(cons);
14078             break;
14079
14080         case 'B':             /* ESC [nB baja n lineas */
14081             n = (value == 0 ? 1 : value);
14082             cons->c_row += n;
14083             flush(cons);
14084             break;
14085
14086         case 'C':             /* ESC [nC mueve n espacios a la derecha */
14087             n = (value == 0 ? 1 : value);
14088             cons->c_column += n;
14089             fluSh(cons);
14090             break;
14091
14092         case 'D':             /* ESC [nO mueve n espacios a la izq. */
14093             n = (value == 0 ? 1 : value);
14094             cons->c_column -= n;
14095             flush(cons);
14096             break;
14097
14098         case 'H':             /* ESC [m;nH" mueve cursor a (m,n) */
14099             cons->c_row = cons->c_esc_parmv[0] - 1 ;
14100             cons->c_column = cons->c_esc_parmv[1] - 1 ;
14101             flush(cons);
14102             break;
14103
14104         case 'J':             /* ESC [sJ despeja la pantalla */
14105             switch (value) {
14106                 case 0:         /* Borrar de cursor a fin pantalla */
14107                     count = scr_size -(cons->c_cur -cons->c_org);
14108                     dst = cons->c_cur;
14109                     break;
14110                 case 1:         /* Borrar de princ. pantalla a cursor */
14111                     count = cons->c_cur -cons->c_org;
14112                     dst = cons->c_org;
14113                     break;
14114                 case 2:         /* Borrar toda la pantalla */
14115                     count = scr_size;
14116                     dst = cons->c_org;
14117                     break;
14118                 default:         /* No hacer nada */
14119                     count = 0;
14120                     dst = cons->c_org;
14121             }
14122             blank_color = cons->c_blank;
14123             mem_vid_copy (BLANK_MEM , dst, count);
14124             break;

```

```
14125
14126     case 'K': /* ESC [sK borra línea desde el cursar */
14127         switch (value) {
14128             case 0: /* Borrar de cursar a fin de línea */
14129                 count = scr_width - cons->c_column;
14130                 dst = cons->c_cur;
14131                 break;
14132             case 1: /* Borrar de prinGo línea a cursar */
14133                 count = cons->c_column;
14134                 dst = cons->c_cur - cons->c_column;
14135                 break;
14136             case 2: /* Borrar toda la línea */
14137                 count = scr_width;
14138                 dst = cons->c_cur - cons->c_column;
14139                 break;
14140             default: /* No hacer nada */
14141                 count = 0;
14142                 dst = cons->c_cur;
14143         }
14144         blank_color = cons->c_blank;
14145         mem_vid_coPY(BLANK_MEM, dst, count);
14146         break;
14147
14148     case 'L': /* ESC [nL inserta n líneas en cursar */
14149         n = value;
14150         if (n < 1) n = 1;
14151         if (n > (scr_lines - cons->c_row))
14152             n = scr_lines - cons->c_row;
14153
14154         src = cons->c_org + cons->c_row * scr_width;
14155         dst = src + n * scr_width;
14156         count = (scr_lines - cons->c_row - n) * scr_width;
14157         vid_vid_coPY(src, dst, count);
14158         blank_color = cons->c_blank;
14159         mem_vid_coPY(BLANK_MEM, src, n * scr_width);
14160         break;
14161
14162     case 'M': /* ESC [nM borra n líneas en el cursar */
14163         n = value;
14164         if (n < 1) n = 1;
14165         if (n > (scr_lines - cons->c_row))
14166             n = scr_lines - cons->c_row;
14167
14168         dst = cons->c_org + cons->c_row * scr_width;
14169         src = dst + n * scr_width;
14170         count = (scr_lines - cons->c_row - n) * scr_width;
14171         vid_vid_copy(src, dst, count);
14172         blank_color = cons->c_blank;
14173         mem_vid_coPY(BLANK_MEM, dst + count, n * scr_width);
14174         break;
14175
14176     case '@': /* ESC [n@ inserta n cars. en cursar */
14177         n = value;
14178         if (n < 1) n = 1;
14179         if (n > (scr_width - cons->c_column))
14180             n = scr_width - cons->c_column;
14181
14182         src = cons->c_cur;
14183         dst = src + n;
14184         count = scr_width - cons->c_column - n;
```

```

14185     vid_vid_copy(src, dst, count);
14186     blank_color = cons->c_blank;
14187     mem_vid_coPY(BLANK_MEM, src, n);
14188     break;
14189
14190     case 'P':                               /* ESC [nP borra n cars. en cursar */
14191         n = value;
14192         if (n < 1) n = 1;
14193         if (n > (scr_width - cons->c_column))
14194             n = scr_width - cons->c_column;
14195
14196         dst = cons->c_cur;
14197         src = dst + n;
14198         count = scr_width - cons->c_column - n;
14199         vid_vid_copy(src, dst, count);
14200         blank_color = cons->c_blank;
14201         mem_vid_coPY(BLANK_MEM, dst + count, n);
14202         break;
14203
14204     case 'm':                               /* ESC [nm habilita estilo n */
14205         switch (value) {
14206             case 1:                         /* NEGRITAS */
14207                 if (color) {
14208                     /* No se puede, usar amarillo */
14209                     cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0E00;
14210                 } else {
14211                     /* Activar bit de intensidad */
14212                     cons->c_attr |= 0x0800;
14213                 }
14214                 break;
14215
14216             case 4:                         /* SUBRAYAR */
14217                 if (color) {
14218                     /* Usar verde claro */
14219                     cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0A00;
14220                 } else {
14221                     cons->c_attr = (cons->c_attr & 0x8900);
14222                 }
14223                 break;
14224
14225             case 5:                         /* PARPADEANTE */
14226                 if (color) {
14227                     /* Usar magenta */
14228                     cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0500;
14229                 } else {
14230                     /* Activar bit de parpadeo */
14231                     cons->c_attr |= 0x8000;
14232                 }
14233                 break;
14234
14235             case 7:                         /* INVERSO */
14236                 if (color) {
14237                     /* Intercambiar colores 1er/20 plano */
14238                     cons->c_attr =
14239                         ((cons->c_attr & 0xf000) >> 4) |
14240                         ((cons->c_attr & 0x0f00) << 4);
14241                 } else
14242                     if ((cons->c_attr & 0x7000) == 0) {
14243                         cons->c_attr = (cons->c_attr & 0x8800) | 0x7000;
14244                     } else {

```

```

14245             cons->c_attr = (cons->c_attr & 0x8800) | 0x0700;
14246         }
14247         break;
14248
14249         default: /* COLOR */
14250             if (30 <= value && value <= 37) {
14251                 cons->c_attr =
14252                     (cons->c_attr & 0xf0ff) |
14253                     (ansi_colors[(value - 30)] << 8);
14254                 cons->c_blank =
14255                     (cons->c_blank & 0xf0ff) |
14256                     (ansi_colors[(value - 30)] << 8);
14257             } else
14258             if (40 <= value && value <= 47) {
14259                 cons->c_attr =
14260                     (cons->c_attr & 0x0fff) |
14261                     (ansi_colors[(value - 40)] << 12);
14262                 cons->c_blank =
14263                     (cons->c_blank & 0x0fff) |
14264                     (ansi_colors[(value - 40)] << 12);
14265             } else {
14266                 cons->c_attr = cons->c_blank;
14267             }
14268             break;
14269         }
14270     }
14271 }
14272 }
14273 cons->c_esc_state = 0;
14274 }

14277 /*=====
14278 *                               set_6845                               *
14279 *=====*/
14280     PRIVATE void set_6845(reg, val)
14281         int reg; /* qué par de registros establecer */
14282         unsigned val; /* valor de 16 bit para establecer */
14283     {
14284         /* Establecer par de registros dentro de 6845.
14285 * Los regs. 12-13 dicen al 6845 dónde comenzar en ram de video
14286 * Los regs. 14-15 dicen al 6845 dónde poner el cursar
14287 */
14288         lock(); /* intentar detener carga h/w valor intermedio */
14289         out_byte(vid_port + INDEX, reg); /* fijar registro índice */
14290         out_byte(vid_port + DATA, (val >> 8) & BYTE); /* byte alto a salida */
14291         out_byte(vid_port + INDEX, reg + 1); /* otra vez */
14292         out_byte(vid_port + DATA, val & BYTE); /* byte bajo a salida */
14293         unlock();
14294     }

14297 /*=====
14298 *                               beep                               *
14299 *=====*/
14300     PRIVATE void beep()
14301     {
14302         /* Emitir alarma sonora por altavoz (salida de CTRL-G).
14303 * Esta rutina opera encendiendo los bits 0 Y 1 del puerto B del chip 8255
14304 * que controla el altavoz.

```

```

14305  */
14306
14307 message mess;
14308
14309 if (beeping) return;
14310 out_byte(TIMER_MODE, 0xB6); /* prep. temporiz. canal 2 (onda cuadr.) */
14311 out_byte(TIMER2, BEEP_FREO & BYTE); /* cargar bits bajos de frec. */
14312 out_byte(TIMER2, (BEEP_FREO» 8) & BYTE); /* ahora bits altos */
14313 lock(); /* proteger PORT_B del manej. int. teclado */
14314 out_byte(PORT_B, in_byte(PORT_B) | 3); /* activar bits de bip */
14315 unlock();
14316 beeping = TRUE;
14317
14318 mess.m_type = SET_ALARM;
14319 mess.CLOCK_PROC_NR = TTY;
14320 mess.DELTA_TICKS = B_TIME;
14321 mess.FUNC_TO_CALL = (sig_handler_t) stop_beep;
14322 sendrec(CLOCK, &mess);
14323 }

14326 /*=====
14327 *                               stop beep                               *
14328 *=====*/
14329 PRIVATE void stop_beep()
14330 {
14331     /* Apagar alarma sonora apagando bits 0 y 1 en PORT_B. */
14332
14333     lock(); /* proteger PORT_B del manej. int. teclado */
14334     out_byte(PORT_B, in_byte(PORT_B) & -3);
14335     beeping = FALSE;
14336     unlock();
14337 }

14340 /*=====
14341 *                               scr_init                               *
14342 *=====*/
14343 PUBLIC void scr_init(tp)
14344     tty_t *tp;
14345 {
14346     /* Inicializar controlador en software de la pantalla. */
14347     console_t *cons;
14348     phys_bytes vid_base;
14349     u16_t bios_crtbase;
14350     int line;
14351     unsigned page_size;
14352
14353     /* Asociar consola y TTY. */
14354     line = tp-&tty_table[0];
14355     if (line >= nr_cons) return;
14356     cons = &cons_table[line];
14357     cons->c_tty = tp;
14358     tp->tty_priv = cons;
14359
14360     /* Inicializar controlador en software del teclado. */
14361     kb_init(tp);
14362
14363     /* Funciones de salida. */
14364     tp->tty_devwrite = cons_write;

```



```

14365 tp->tty_echo = cons_echo;
14366
14367 /* Obtener parámetros BIOS que dan a la VDU el registro base de E/S. */
14368 phys_copy(0x463L, vir2phys(&bios_crtbase), 2L);
14369
14370 vid_port = bios_crtbase;
14371
14372 if (color) {
14373     vid_base = COLOR_BASE;
14374     vid_size = COLOR_SIZE;
14375 } else {
14376     vid_base = MONO_BASE;
14377     vid_size = MONO_SIZE;
14378 }
14379 if (ega) vid_size = EGA_SIZE; /* tanto para EGA como VGA */
14380 wrap = lega;
14381
14382 vid_seg = protected_mode ? VIDEO_SELECTOR : physb_to_hclick(vid_base);
14383 init_dataseg(&gdt[VIDEO_INDEX], vid_base, (phys_bytes) vid_size,
14384             TASK_PRIVILEGE);
14385 vid_size >>= 1; /* cuenta de palabras */
14386 vid_mask = vid_size - 1;
14387
14388 /* Puede haber tantas consolas como la memoria de video permita. */
14389 nr_cons = vid_size / scr_size;
14390 if (nr_cons > NR_CONS) nr_cons = NR_CONS;
14391 if (nr_cons > 1) wrap = 0;
14392 page_size = vid_size / nr_cons;
14393 cons->c_start = line * page_size;
14394 cons->c_limit = cons->c_start + page_size;
14395 cons->c_org = cons->c_start;
14396 cons->c_attr = cons->c_blank = BLANK_COLOR;
14397
14398 /* Despejar la pantalla. */
14399 blank_color = BLANK_COLOR;
14400 mem_vid_copy(BLANK_MEM, cons->c_start, scr_size);
14401     select_console(0);
14402 }

14405 /*=====*/
14406 *                                     putk                                     *
14407 *=====*/
14408     PUBLIC void putk(c)
14409                 int c; /* carácter por imprimir */
14410     {
14411     /* La versión de printf() vinculada con el kernel mismo usa este procedimiento.
14412 * El de la biblioteca envía un mensaje a FS, que no es lo que se necesita
14413 * para imprimir dentro del kernel. Esta versión se limita a poner en cola el
14414 * carácter e iniciar la salida.
14415 */
14416
14417     if (c != 0) {
14418         if (c == '\n') putk('\r');
14419         out_char(&cons_table[0], (int) c);
14420     } else {
14421         flush(&cons_table[0]);
14422     }
14423 }

```

```

14426 /*=====
14427 *                                toggle_scroll                                *
14428 *=====*/
14429 PUBLIC void toggle_scroll()
14430 {
14431     /* Conmutar entre desplazamiento por hardware y por software. */
14432
14433     cons_org0();
14434     softscroll = !softscroll;
14435     printf("%sware scrolling enabled.\n", softscroll ? "80ft" : "Hard");
14436 }

14439 /*=====
14440 *                                cons_stop                                    *
14441 *=====*/
14442 PUBLIC void cons_stop()
14443 {
14444     /* Preparar paro o rearranque. */
14445
14446     cons_org0();
14447     softscroll = 1;
14448     select_conSole(0);
14449     cons_table[0].c_attr = cons_table[0].c_blank = BLANK_COLOR;
14450 }

14453 /*=====
14454 *                                cons_org0                                    *
14455 *=====*/
14456 PRIVATE void cons_org0()
14457 {
14458     /* Desplazar memoria de video hacia atrás para poner el origen en 0. */
14459
14460     int cons_line;
14461     console_t *cons;
14462     unsigned n;
14463
14464     for (cons_line = 0; cons_line < nr_consj cons_line++) {
14465         cons = &cons_table[cons_line];
14466         while (cons->c_org > cons->c_start) {
14467             n = vid_size - scr_size;                /* cantidad de memo no usada */
14468             if (n > cons->c_org - cons->c_start)
14469                 n = cons->c_org - cons->c_start;
14470             vid_vid_copy(cons->c_org, cons->c_org - n, scr_size);
14471             cons->c_org -= n;
14472         }
14473         flush(cons);
14474     }
14475     select_conSole(current);
14476 }

14479 /*=====
14480 *                                select_console                                *
14481 *=====*/ :
14482
14482 PUBLIC void
select console(int con s line)
14483 {
14484     /* Asignar a la consola actual núm. de consola 'cons_line', */

```

```

14485
14486 if (cons_line < 0 || cons_line >= nr_cons) return;
14487 current = cons_line;
14488 curcons = &cons_table[cons_line];
14489 set_6845(VID_ORG, curcons->c_org);
14490 set_6845(CURSORS, curcons->c_cur);
14491 }

14494 /*=====
14495 *                               con_loadfont                               *
14496 *=====*/
14497 PUBLIC int con_loadfont(user_phys)
14498     phys_bytes user_phys;
14499 {
14500     /* Cargar una fuente en el adaptador EGA o VGA. */
14501
14502     static struct sequence seq1[7] = {
14503         { GA_SEQUENCER_INDEX, 0x00, 0x01 },
14504         { GA_SEQUENCER_INDEX, 0x02, 0x04 },
14505         { GA_SEQUENCER_INDEX, 0x04, 0x07 },
14506         { GA_SEQUENCER_INDEX, 0x00, 0x03 },
14507         { GA_GRAPHICS_INDEX, 0x04, 0x02 },
14508         { GA_GRAPHICS_INDEX, 0x05, 0x00 },
14509         { GA_GRAPHICS_INDEX, 0x06, 0x00 },
14510     };
14511     static struct sequence seq2[7] = {
14512         { GA_SEQUENCER_INDEX, 0x00, 0x01 },
14513         { GA_SEQUENCER_INDEX, 0x02, 0x03 },
14514         { GA_SEQUENCER_INDEX, 0x04, 0x03 },
14515         { GA_SEQUENCER_INDEX, 0x00, 0x03 },
14516         { GA_GRAPHICS_INDEX, 0x04, 0x00 },
14517         { GA_GRAPHICS_INDEX, 0x05, 0x10 },
14518         { GA_GRAPHICS_INDEX, 0x06, 0 },
14519     };
14520     seq2[6].value = color? 0x0E : 0x0A;
14521
14522     if (!lega) return(ENOTTY);
14523     lock();
14524     ga_program(seq1);          /* traer a la vista la memoria de fuente */
14525     phys_copy(user_phys, (phys_bytes)GA_VIDEO_ADDRESS, (phys_bytes)GA_FONT_SIZE);
14526     ga_program(seq2);          /* restaurar */
14527     unlock();
14528     return(OK);
14529 }

14537 /*=====
14538 *                               ga_program                               *
14539 *=====*/
14540 PRIVATE void ga_program(seq)
14541     struct sequence *seq;
14542 {
14543     /* función de apoyo para con_loadfont */
14544

```

```

14545 int len= 7;
14546 do {
14547     out_byte(seq->index, seq->port);
14548     out_byte(seq->index+1, seq->value);
14549     seq++;
14550 } while (-len > 0);
14551 }

```

```

+++++
src/kernel/dmp.c
+++++

```

```

14600  /* Este archivo contiene rutinas de vaciado para depuración. */
14601
14602 #include "kernel.h"
14603 #include <minix/com.h>
14604 #include "proc.h"
14605
14606 char *vargv;
14607
14608 FORWARD_PROTOYPE(char *proc_name, (int proc_nr));
14609
14610 /*=====
14611 *                               p_dmp                               *
14612 *=====*/
14613 PUBLIC void p_dmp()
14614 {
14615     /* Vaciado de tabla de procesos. */
14616
14617     register struct proc *rp;
14618     static struct proc *oldrp = BEG_PROC_ADDR;
14619     int n = 0;
14620     phys_clicks text, data, size;
14621     int proc_nr;
14622
14623     printf("\n--pid --pc- ---sp- flag -user --sys-- -text- -data- -size- -recv- command\n");
14624
14625     for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
14626         proc_nr = proc_number(rp);
14627         if (rp->p_flags & P_SLOT_FREE) continue;
14628         if (++n > 20) break;
14629         text = rp->p_map[T].mem_phys;
14630         data = rp->p_map[D].mem_phys;
14631         size = rp->p_map[T].mem_len
14632             + ((rp->p_map[S].mem_phys + rp->p_map[S].mem_len) - data);
14633         printf("%5d %5lx %6lx %2x %7U %7U %5uK %5uK %5uK ",
14634             proc_nr < 0 ? proc_nr : rp->p_pid,
14635             (unsigned long) rp->p_reg.pc,
14636             (unsigned long) rp->p_reg.sp,
14637             rp->p_flags,
14638             rp->user_time, rp->sys_time,
14639             click_to_round_k(text), click_to_round_k(data),
14640             click_to_round_k(size));
14641         if (rp->p_flags & RECEIVING) {
14642             printf("%-7.7s", proc_name(rp->p_getfrom));
14643         } else
14644         if (rp->p_flags & SENDING) {

```

```

14645         printf("S:%-5.5S", proc_name(rp->p_sendto));
14646     } else
14647     if (rp->p_flags == 0) {
14648         printf(" ");
14649     }
14650     printf("%s\n", rp->p_name);
14651 }
14652 if (rp == END_PRDC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
14653 oldrp = rp;
14654 }

14657 /*=====
14658 *                               map_dmp                               *
14659 *=====*/
14660 PUBLIC void map_dmp()
14661 {
14662     register struct proc *rp;
14663     static struct proc *oldrp = cproC_addr(HARDWARE);
14664     int n = 0;
14665     phys_clicks size;
14666
14667     printf("\nPROC NAME- ----- TEXT ----- DATA----- STACK -----SIZE-\n");
14668     for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
14669         if (rp->p_flags & P_SLOT_FREE) continue;
14670         if (++n > 20) break;
14671         size = rp->p_map[T].mem_len
14672             + ((rp->p_map[S].mem_phys + rp->p_map[S].mem_len
14673                -rp->p_map[D].mem_phys);
14674         printf("%3d %-6.6s %4x %4x %4x %4x %4x %4x %4x %4x %5uK\n",
14675             proc_number(rp),
14676             rp->p_name,
14677             rp->p_map[T].mem_vir, rp->p_map[T].mem_phys, rp->p_map[T].mem_len,
14678             rp->p_map[D].mem_vir, rp->p_map[D].mem_phys, rp->p_map[D].mem_len,
14679             rp->p_map[S].mem_vir, rp->p_map[S].mem_phys, rp->p_map[S].mem_len,
14680             click_to_round_k(size));
14681     }
14682     if (rp == END_PROC_ADDR) rp = cproC_addr(HARDWARE); else printf("--more--\r");
14683     oldrp = rp;
14684 }

14687 /*=====
14688 *                               proc_name                               *
14689 *=====*/
14690 PRIVATE char *proc_name(proc_nr)
14691     int proc_nr;
14692     {
14693     if (proc_nr == ANY) return "ANY";
14694     return proc_addr(proc_nr)->p_name;
14695     }

```

```
+++++
src/kernel/system.c
+++++
```

```
14700/* Esta tarea maneja la interfaz entre el sist. arch. y el kernel y entre el
14701 * adm. memo y el kernel. Los servicios de sistema se obtienen enviando un
14702 * mensaje ~ sys_task() especificando qué se necesita. Para ayudar al MM y FS,
14703 * hay una biblioteca con rutinas cuyos nombres tienen la forma sys_xxx, p.ej.
14704 * sys_xit envía el mensaje SYS_XIT a sys_task. Los tipos de mensajes y sus
14705 * parámetros son:
14706 *
14707 * SYS_FORK informa al kernel que un proceso bifurcó
14708 * SYS_NEWMAP permite a MM crear mapa de memoria de un proceso
14709 * SYS_GETMAP permite a MM obtener mapa de memoria de un proceso
14710 * SYS_EXEC fija contador de programa y apunto a pila después de EXEC
14711 * SYS_XIT informa al kernel que un proceso salió
14712 * SYS_GETSP invocador desea leer el apunto a pila de algún proceso
14713 * SYS_TIMES invocador desea tiempos de contabil. de un proceso
14714 * SYS_ABORT MM o FS no puede continuar; abortar MINIX
14715 * SYS_FRESH iniciar con nueva imagen de proceso durante EXEC (sólo 68000)
14716 * SYS_SENOSIG enviar una señal a un proceso (estilo POSIX)
14717 * SYS_SIGRETURN completar señalización estilo POSIX
14718 * SYS_KILL causar envío de señal via MM
14719 * SYS_ENOSIG acabar después de señal tipo SYS_KILL
14720 * SYS_COPY solicitar copiado de un bloque de datos entre procesos
14721 * SYS_VCOPY solíc. copiado de serie de bloques de datos entre procs
14722 * SYS_GBOOT copia los parámetros de arranque a un proceso
14723 * SYS_MEM devuelve el siguiente trozo de memoria física libre
14724 * SYS_UMAP calcula la dirección física de una dirección virtual dada
14725 * SYS_TRACE solicita una operación de rastreo
14726 *
14727 * Tipos de mensajes y parámetros:
14728 *
14729 * m_type PROC1 PROC2 PIO MEM_PTR
14730 * -----
14731 * |SYS_FORK | padre | hijo | pid | |
14732 * |-----+-----+-----+-----+-----|
14733 * |SYS_NEWMAP | # proc | | | ap mapa |
14734 * |-----+-----+-----+-----+-----|
14735 * |SYS_EXEC | # proc | rastro | a-p nvo |
14736 * |-----+-----+-----+-----+-----|
14737 * |SYS_XIT | padre | salió | | |
14738 * |-----+-----+-----+-----+-----|
14739 * |SYS_GETSP | # proc | || | |
14740 * |-----+-----+-----+-----+-----|
14741 * |SYS_TIMES | # proc | | | ap buf |
14742 * |-----+-----+-----+-----+-----|
14743 * |SYS_ABORT | | | | |
14744 * |-----+-----+-----+-----+-----|
14745 * |SYS_FRESH | # proc | data_cl | | |
14746 * |-----+-----+-----+-----+-----|
14747 * |SYS_GBOOT | # proc | | | ap arr |
14748 * |-----+-----+-----+-----+-----|
14749 * |SYS_GETMAP | # proc | | | ap mapa |
14750 * -----
14751 *
14752 * m_type m1_i1 m1_i2 m1_i3 m1_p1
14753 * -----+-----+-----+-----+-----|
14754 * |SYS_VCOPY | p orig | p dest | tam vec | dir vc |
```

```

14755 * |-----+-----+-----+-----+-----|
14756 * |SYS_SENDSIG |# proc |      |      |      | smp      |
14757 * |-----+--+-----+-----+-----+-----|
14758 * |SYS_SIGRETURN|# proc |      |      |      | scp      |
14759 * |-----+-----+-----+-----+-----|
14760 * |SYS_ENDSIG  |# proc |      |      |      |      |
14761 * -----
14762 *
14763 *      m_type      m2_i1      m2_i2      m2_l1      m2_l2
14764 * -----
14765 * |SYS_TRACE   |núm_proc | solic      | direc      | datos      |
14766 * -----
14767 *
14768 *
14769 *      m_type      m6_i1      m6_i2      m6_i3      m6_f1
14770 * -----
14771 * |SYS_KILL    | núm_proc | señal      |      |      |
14772 * -----
14773 *
14774 *
14775 *      m_type  m5_c1  m5_i1  m5_l1  m5_c2  m5_i2  m5_l2  m5_l3
14776 * -----
14777 * |SYS_CDPY |seg org|proc org |vir org|seg dst |proc dst |vir dst | cta bytes |
14778 * -----
14779 * |SYS_UMAP|  seg |núm_proc|dir vir |      |      |      | cta bytes |
14780 * -----
14781 *
14782 *
14783 *      m_type      m1_i1      m1_i2      m1_i3
14784 * |-----+-----+-----+-----|
14785 * |SYS_MEM | base mem | tam mem | mem tot |
14786 * -----
14787 *
14788 * Además del punto de entrada sys_task() principal, hay otros 5 puntos
14789 * de entrada secundarios:
14790 * cause_sig: tomar medidas para que ocurra una señal, tarde o temprano
14791 * inform: informar a MM de señales pendientes
14792 * numap: umap segmento D partiendo del # de proc en vez de apuntador
14793 * umap: calcular dirección física para una dirección virtual dada
14794 * alloc_segments: repartir segmentos para procesador 8088 o mayor
14795 */
14796
14797 #include "kernel.h"
14798 #include <signal.h>
14799 #include <unistd.h>
14800 #include <sys/sigcontext.h>
14801 #include <sys/ptrace.h>
14802 #include <minix/boot.h>
14803 #include <minix/callnr.h>
14804 #include <minix/com.h>
14805 #include "proc.h"
14806 #include "protect.h"
14807
14808 /* Máscaras PSW. */
14809 #define IF_MASK 0x0000200
14810 #define IOPL_MASK 0x003000
14811
14812 PRIVATE message m;
14813
14814 FORWARD_PROTOTYPE( int do_abort, (message *m_ptr) );

```

```

14815 FORWARD_PROTOTYPE( int do_copy, (message *m_ptr) );
14816 FORWARD_PROTOTYPE( int do_exec, (message *m_ptr) );
14817 FORWARD_PROTOTYPE( int do_fork, (message *m_ptr) );
14818 FORWARD_PROTOTYPE( int do_gboot, (message *m_ptr) );
14819 FORWARD_PROTOTYPE( int do_getsp, (message *m_ptr) );
14820 FORWARD_PROTOTYPE( int do_kill, (message *m_ptr) );
14821 FORWARD_PROTOTYPE( int do_mem, (message *m_ptr) );
14822 FORWARD_PROTOTYPE( int do_newmap, (message *m_ptr) );
14823 FORWARD_PROTOTYPE( int do_sendsig, (message *m_ptr) );
14824 FORWARD_PROTOTYPE( int do_sigreturn, (message *m_ptr) );
14825 FORWARD_PROTOTYPE( int do_endsig, (message *m_ptr) );
14826 FORWARD_PROTOTYPE( int do_times, (message *m_ptr) );
14827 FORWARD_PROTOTYPE( int do_trace, (message *m_ptr) );
14828 FORWARD_PROTOTYPE( int do_umap, (message *m_ptr) );
14829 FORWARD_PROTOTYPE( int do_xit, (message *m_ptr) );
14830 FORWARD_PROTOTYPE( int do_vcopy, (message *m_ptr) );
14831 FORWARD_PROTOTYPE( int do_getmap, (message *m_ptr) );
14832
14833
14834 /*=====*/
14835 *                                     sys_task                                     *
14836 /*=====*/
14837 PUBLIC void sys_task()
14838 {
14839     /* Pta. entrada principal de sys_task. Obt. mensaje y despachar según tipo. */
14840
14841     register int r;
14842
14843     while (TRUE) {
14844         receive(ANY, &m);
14845
14846         switch (m.m_type) { /* cuál llamada al sistema */
14847             case SYS_FORK:    r = do_fork(&m);        break;
14848             case SYS_NEWMAP:  r = do_newmap(&m);      break;
14849             case SYS_GETMAP:  r = do_getmap(&m);      break;
14850             case SYS_EXEC:    r = do_exec(&m);        break;
14851             case SYS_XIT:     r = do_xit(&m);         break;
14852             case SYS_GETSP:   r = do_getsp(&m);       break;
14853             case SYS_TIMES:   r = do_times(&m);       break;
14854             case SYS_ABORT:   r = do_abort(&m);       break;
14855             case SYS_SENDSIG: r = do_sendsig(&m);    break;
14856             case SYS_SIGRETURN: r = do_sigreturn(&m); break;
14857             case SYS_KILL:    r = do_kill(&m);        break;
14858             case SYS_ENDSIG:  r ~ do_endsig(&m);     break;
14859             case SYS_COPY:    r = do_copy(&m);        break;
14860             case SYS_VCOPY:   r = do_vcopy(&m);      break;
14861             case SYS_GBOOT:   r = do_gboot(&m);      break;
14862             case SYS_MEM:     r = do_mem(&m);         break;
14863             case SYS_UMAP:    r = do_umap(&m);       break;
14864             case SYS_TRACE:   r = do_trace(&m);      break;
14865             default:         r = E_BAD_FCN;
14866         }
14867
14868         m.m_type = r; /* 'r' informa situación de llamada */
14869         send(m.m_source, &m); /* enviar respuesta al invocador */
14870     }
14871 }
14874 /*=====*/

```



```

14875 *                                     do_fork                                     *
14876 *=====
14877PRIVATE int do_fork(m_ptr)
14878register message *m_ptrj          /* apuntador a mensaje de solicitud */
14879 {
14880/* Manejar sys_fork(). m_ptr->PROC1 bifurcó. El hijo es m_ptr->PROC2. */
14881
14882 reg_t old_ldt_sel;
14883 register struct proc *rpc;
14884 struct proc *rpp;
14885
14886 if (!isokusern(m_ptr->PROC1) || !isokusern(m_ptr->PROC2))
14887     return(E_BAD_PROC);
14888 rpp = proc_addr(m_ptr->PROC1);
14889 rpc = proc_addr(m_ptr->PROC2);
14890
14891 /* Copiar struct 'proc' del padre al hijo. */
14892 old_ldt_sel = rpc->p_ldt_sel; /* evitar que copia destruya esto */
14893
14894 *rpc = *rpp;          /* copiar estructura 'proc' */
14895
14896 rpc->p_ldt_sel = old_ldt_sel;
14897 rpc->p_nr = m_ptr->PROC2; /* la copia destruyó esto */
14898
14899 rpc->p_flags |= NO_MAP; /* inhibir ejecución del proceso */
14900
14901 rpc->p_flags &= ~(PENDING | SIG_PENDING | P_STOP);
14902
14903 /* Sólo 1 del grupo debe tener PENDING, hijo no hereda situac. rastreo*/
14904 sigemptyset(&rpc->p_pending);
14905 rpc->p_pendcount = 0;
14906 rpc->p_pid = m_ptr->PID; /* instalar pid del hijo */
14907 rpc->p_reg.retreg = 0; /* hijo ve pid = 0 p/saber que es hijo */
14908
14909 rpc->user_time = 0; /* poner en 0 tiempos de contabil. */
14910 rpc->sys_time = 0;
14911 rpc->child_utime = 0;
14912 rpc->child_stime = 0;
14913
14914     return(OK);
14915 }

14918 /*=====
14919*                                     do_newmap                                     *
14920 *=====
14921PRIVATE int do_newmap(m_ptr)
14922                                     message *m_ptr; /* apunto a mensaje de solicitud */
14923 {
14924 /* Manejar sys_newmap(). Traer mapa de memoria del MM. */
14925
14926     register struct proc *rp;
14927     phys_bytes src_phys;
14928 int caller; /* espacio de quien tiene nvo. mapa (norm. MM) */
14929 int k; /* proceso cuyo mapa debe cargarse */
14930 int old_flags; /* valor de banderas antes de modificación */
14931 struct mem_map *map_ptr; /* dir. virtual de mapa dentro invocador (MM) */
14932
14933 /* Extraer parámetros del mensaje y copiar nuevo mapa de memoria de MM. */
14934 caller = m_ptr->m_source;

```

```

14935 k = m_ptr->PROC1;
14936 map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
14937 if (!isokprocn(k)) return(E_BAD_PROC);
14938 rp = proc_addr(k);          /* apunto a entrada de usuario que obt. nvo. mapa */
14939
14940 /* Copiar el mapa de MM. */
14941 src_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, Sizeof(rp->p_map));
14942 if (src_phys == 0) panic("bad call to sys_newmap", NO_NUM);
14943 phys_copy(src_phys, vir2phys(rp->p_map), (phys_bytes) Sizeof(rp->p_map));
14944
14945 alloc_segments(rp);
14946 old_flags = rp->p_flags;    /* guardar valor previo de banderas */
14947 rp->p_flags &= -NO_MAP;
14948 if (old_flags != 0 && rp->p_flags == 0) lock_ready(rp);
14949
14950 return(OK);
14951 }

14954 /*=====
14955 *                               do_getmap                               *
14956 *=====*/
14957 PRIVATE int do_getmap(m_ptr)
14958 message *m_ptr;          /* apuntador a mensaje solicitud */
14959 {
14960 /* Manejar sys_getmap(). Informar mapa de memoria a MM. */
14961
14962 register struct proc *rp;
14963 phys_bytes dst_phys;
14964 int caller;              /* dónde guardar el mapa */
14965 int k;                   /* proceso cuyo mapa se carga */
14966 struct mem_map *map_ptr; /* dir. virtual de mapa dentro invocador (MM) */
14967
14968 /* Extraer parámetros del mensaje y copiar nuevo mapa de memoria en MM. */
14969 caller = m_ptr->m_source;
14970 k = m_ptr->PROC1;
14971 map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
14972
14973 if (!isokproCn(k))
14974     panic("do_getmap got bad proc: ", m_ptr->PROC1);
14975
14976 rp = proc_addr(k);      /* apunto a entrada del mapa */
14977
14978 /* Copiar el mapa en MM. */
14979 dst_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, SizeOf(rp->p_map));
14980 if (dst_phys == 0) panic("bad call to sys_getmap", NO_NUM);
14981 phys_copy(vir2phys(rp->p_map), dst_phys, sizeOf(rp->p_map));
14982
14983 return(OK);
14984 }

14987 /*=====
14988 *                               do_exec                               *
14989 *=====*/
14990 PRIVATE int do_exec(m_ptr)
14991 register message *m_ptr; /* apuntador a mensaj~ solicitud */
14992 {
14993 /* Manejar sys_exec(). Un proceso hizo EXEC con éxito. Parcharlo. */
14994

```

```

14995 register struct proc *rp;
14996 reg_t sp;                               /* nuevo apuntador a pila */
14997 phys_bytes phys_name;
14998 char *np;
14999 #define NLEN (sizeof(rp->p_name) -1)
15000
15001 if (!isoksuser(m_ptr->PROC1)) return E_BAD_PROC;
15002 /* Campo PROC2 sirve como bandera p/indicar que el proc. es rastreado. */
15003 if (m_ptr->PROC2) cause_sig(m_ptr->PROC1, SIGTRAP);
15004 sp = (reg_t) m_ptr->STACK_PTR;
15005 rp = proc_addr(m_ptr->PROC1);
15006 rp->p_reg.sp = sp;                         /* fijar apuntador a pila */
15007 rp->p_reg.pc = (reg_t) m_ptr->IP_PTR; /* fijar contador programa */
15008 rp->p_alarm = 0;                           /* restab. temporiz. alarma */
15009 rp->p_flags &= -RECEIVING;                 /* MM no contesta a llamada EXEC */
15010 if (rp->p_flags == 0) lock_ready(rp);
15011
15012 /* Guardar nombre de comando p/depuración, salida ps(1), etc. */
15013 phys_name = numap(m_ptr->m_source, (vir_bytes) m_ptr->NAME_PTR,
15014                  (vir_bytes) NLEN);
15015 if (phys_name != 0) {
15016     phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) NLEN);
15017     for (np = rp->p_name; (*np & BYTE) >= ' '; np++) {}
15018     *np = 0;
15019 }
15020 return(OK);
15021 }

15024 /*=====
15025 *                               do_xit                               *
15026 *=====*/
15027 PRIVATE int do_xit(m_ptr)
15028 message *m_ptr;                 /* apunto a mensaje solicitud */
15029 {
15030     /* Manejar sys_xit(). Un proceso salió. */
15031
15032 register struct proc *rp, *rc;
15033 struct proc *np, *xp;
15034 int parent;                       /* núm. del padre del proc. que sale */
15035 int proc_nr;                       /* núm. del proceso que sale */
15036 phys_clicks base, size;
15037
15038 parent = m_ptr->PROC1;             /* núm. ranura de proc. padre */
15039 proc_nr = m_ptr->PROC2;           /* núm. ranura de proc. que sale */
15040 if (!isoksuser(parent) || !isOksuser(proc_nr)) return(E_BAD_PROC);
15041 rp = proc_addr(parent);
15042 rc = proc_addr(proc_nr);
15043 lock();
15044 rp->child_utime += rc->user_time + rc->child_utime; /* tiempos acum. hijo */
15045 rp->child_9time += rc->sys_time + rc->child_stime;
15046 unlock();
15047 rc->p_alarm = 0;                   /* apagar temporizador de alarma */
15048 if (rc->p_flags == 0) lock_unready(rc);
15049
15050 strcpy(rc->p_name, "<noname>");     /* el proc. ya no tiene nombre */
15051
15052 /* Si el proceso terminado está formado en cola
15053 * tratando de enviar un mensaje (o sea, una señal mató al proceso,
15054 * él no hizo EXIT), deberá quitarse de las colas de mensaje.

```

```

15055     */
15056 if (rc->p_flags & SENDING) {
15057     /* Revisar todas ranuras de proc p_jver si el proc que sale está ahí. */
15058     for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
15059         if (rp->p_callerq == NIL_PROC) continue;
15060         if (rp->p_callerq == rc) {
15061             /* El proc que sale está a la cabeza de esta cola. */
15062             rp->p_callerq = rc->p_sendlink;
15063             break;
15064         } else {
15065             /* Ver si el proc que sale está en medio de cola. */
15066             np = rp->p_callerq;
15067             while ( ( xp = np->p_sendlink) != NIL_PROC)
15068                 if (xp == rc) {
15069                     np->p_sendlink = xp->p_sendlink;
15070                     break;
15071                 } else {
15072                     np = xp;
15073                 }
15074             }
15075         }
15076     }
15077
15078 if (rc->p_flags & PENDING) --sig_procs;
15079 sigemptyset(&rc->p_pending);
15080 rc->p_pendcount = 0;
15081 rc->p_flags = P_SLOT_FREE;
15082     return(OK);
15083 }

15086 /*=====
15087 *                               do_getsp                               *
15088 *=====*/
15089     PRIVATE int do_getsp(m_ptr)
15090         register message *m_ptr; /* apunto a mensaje solicitud */
15091     {
15092         /* Manejar sys_getsp(). MM quiere conocer el apuntador a la pila (sp). */
15093
15094     register struct proc *rp;
15095
15096     if (!isokusern(m_ptr->PROC1))     return(E_BAD_PROC);
15097     rp = proc_addr(m_ptr->PROC1);
15098     m_ptr->STACK_PTR = (char *) rp->p_reg.sp;          /* devolver sp aquí (tipo mal) */
15099     return(OK);
15100     }

15103 /*=====
15104 *                               do_times                               *
15105 *=====*/
15106     PRIVATE int do_times (m_ptr)
15107         register message *m_ptr; /* apunto a mensaje solicitud */
15108     {
15109         /* Manejar sys_times(). Recuperar información de contabilidad. */
15110
15111     register struct proc *rp;
15112
15113     if (!isokusern(m_ptr->PROC1))     return E_BAD_PROC;
15114     rp = proc_addr(m_ptr->PROC1);

```

```

15115
15116 /* Insertar tiempos req. por llamada al sist. TIMES en el mensaje. */
15117 loCk(); /* detener contadores de tiempo volátiles en rp */
15118 m_ptr->USER_TIME = rp->user_time;
15119 m_ptr->SYSTEM_TIME = rp->sys_time;
15120 unloCk();
15121 m_ptr->CHILD_UTIME = rp->child_utime.
15122 m_ptr->CHILD_STIME = rp->child_stime;
15123 m_ptr->BOOT_TICKS = get_uptime();
15124 return(OK);
15125 }

15128 /*=====
15129 * do_abort *
15130 *=====
15131 PRIVATE int do_abort(m_ptr)
15132 message *m_ptr; /* apunto a mensaje solicitud */
15133 {
15134 /* Manejar sys_abort(). MINIX no puede continuar. Terminar operación. */
15135 char monitor_Code[64];
15136 phys_bytes src_phys;
15137
15138 if (m_ptr->m1_i1 == RBT_MONITOR) {
15139 /* Monitor debe ejecutar instrucciones especific. por usuario. */
15140 src_phys = numap(m_ptr->m_source, (vir_bytes) m_ptr->m1_p1,
15141 (vir_bytes) sizeof(monitor_Code));
15142 if (src_phys == 0) panic("bad monitor code from", m_ptr->m_source);
15143 phys_copy(src_phys, vir2Phys(monitor_Code),
15144 (phys_bytes) sizeof(monitor_Code));
15145 reboot_code = vir2Phys(monitor_Code);
15146 }
15147 wreboot(m_ptr->m1_i1);
15148 return(OK); /* pro-forma (realmente EDISASTER) */
15149 }

15154 /*=====
15155 * do_sendsig *
15156 *=====
15157 PRIVATE int do_sendsig(m_ptr)
15158 message *m_ptr; /* apunto a mensaje solicitud */
15159 {
15160 /* Manejar Sys_sendsig, señal estilo POSIX */
15161
15162 struct sigmsg smsg;
15163 register struct proc *rp;
15164 phys_bytes src_phys, dst_phys;
15165 struct sigcontext sc, *scp;
15166 struct sigframe fr, *fip;
15167
15168 if (lisokusern(m_ptr->PROC1) return(E_BAD_PROC);
15169 rp = proc_addr(m_ptr->PROC1);
15170
15171 /* Traer estructura sigmsg a nuestro espacio de direcciones. */
15172 src_phys = umap(proc_addr(MM_PROC_NR), D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
15173 (vir_bytes) sizeof(struct sigmsg));
15174 if (src_phys == 0)

```

```

15175         panic("do_sendsig can't signal: bad sigmsg address from MM", NO_NUM);
15176 phys_copy(src_phys, vir2phys(&smsg) , (phys_bytes) sizeof(struct sigmsg));
15177
15178 /* Calcular valor de apunt a pila usuario donde se guardará sigcontext. */
15179 scp = (struct sigcontext *) smsg.sm_stkptr -1;
15180
15181 /* Copiar registros en estructura sigcontext. */
15182 memcpy(&sc.sc_regs, &rp->p_reg, sizeof(struct sigregs));
15183
15184 /* Terminar la inicialización de signcontext. */
15185 sc.sc_flags = SC_SIGCONTEXT;
15186
15187 sc.sc_mask = smsg.sm_mask;
15188
15189 /* Copiar la estructura sigcontext en la pila del usuario. */
15190 dst_phys = umap(rp, D, (vir_bytes) scp,
15191                (vir_bytes) sizeof(struct sigcontext));
15192 if (dst_phys == 0) return(EFAULT);
15193 phys_copy(vir2phys(&sc) , dst_phys, (phys_bytes) sizeof(struct sigcontext));
15194
15195 /* Inicializar la estructura sigframe. */
15196 frp = (struct sigframe *) scp -1;
15197 fr.sf_scpcopy = scp;
15198 fr.sf_retadr2= (void (*) ()) rp->p_reg.pc;
15199 fr.sf_fp = rp->p_reg.fp;
15200 rp->p_reg.fp = (reg_t) &frp->sf_fp;
15201 fr.sf_scp = scp;
15202 fr.sf_code = 0; /* XXX -debe usarse p/tipo de excepción FP */
15203 fr.sf_signo = smsg.sm_signo;
15204 fr.sf_retadr = (void (*)()) smsg.sm_sigreturn;
15205
15206 /* Copiar la estructura sigframe en la pila del usuario. */
15207 dst_phys = umap(rp, D, (vir_bytes) frp, (vir_bytes) sizeof(struct sigframe));
15208 if (dst_phys == 0) return(EFAULT);
15209 phys_copy(vir2phys(&fr), dst_phys, (phys_bytes) sizeof(struct sigframe));
15210
15211 /* Restablecer registros de usuario p/ejecutar el manejador de señales. */
15212 rp->p_reg.sp = (reg_t) frp;
15213 rp->p_reg.pc = (reg_t) smsg.sm_sighandler;
15214
15215     return(OK);
15216 }

15218 /*=====
15219 *                               do_sigreturn                               *
15220 *=====*/
15221 PRIVATE int do_sigreturn(m_ptr)
15222 register message *m_ptr;
15223 {
15224 /* Las señales estilo POSIX requieren sys_sigreturn para ordenar las cosas antes
15225 * de que el proceso señalado pueda reanudar su ejecución
15226 */
15227
15228 struct sigcontext sc;
15229 register struct proc *rp;
15230 phys_bytes src_phys;
15231
15232 if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
15233 rp = proc_addr(m_ptr->PROC1);
15234

```

```

15235 /* Copiar la estructura sigcontext. */
15236 src_phys = umap(rp, D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
15237                (vir_bytes) sizeof(struct sigcontext));
15238 if (src_phys == 0) return(EFAULT);
15239 phys_copy(src_phys, vir2phys(&sc), (phys_bytes) sizeof(struct sigcontext));
15240
15241 /* Asegurarse de que no es s610 un jmp_buf. */
15242 if ((sc.sc_flags & SC_SIGCONTEXT) == 0) return(EINVAL);
15243
15244 /* Arreglar s610 ciertos registros clave si el compilador no usa
15245  * variables de registro dentro de las funciones que contienen setjmp.
15246  */
15247 if (sc.sc_flags & SC_NOREGLOCALS) {
15248     rp->p_reg.retreg = sc.sc_retreg;
15249     rp->p_reg.fp = sc.sc_fp;
15250     rp->p_reg.pc = sc.sc_pc;
15251     rp->p_reg.sp = sc.sc_sp;
15252     return (OK);
15253 }
15254 sc.sc_psw = rp->p_reg.psw;
15255
15256 #if (CHIP == INTEL)
15257 /* No asustar al kernel si el usuario dio selecto res malos. */
15258 sc.sc_cs = rp->p_reg.cs;
15259 sc.sc_ds = rp->p_reg.ds;
15260 sc.sc_es = rp->p_reg.es;
15261 #if _WORD_SIZE == 4
15262 sc.sc_fs = rp->p_reg.fs;
15263 sc.sc_gs = rp->p_reg.gs;
15264 #endif
15265 #endif
15266
15267 /* Restaurar los registros. */
15268 memcpy(&rp->p_reg, (char *)&sc.sc_regs, sizeof(struct sigregs));
15269
15270 return(OK);
15271 }

15273 /*=====
15274 *                               do_kill                               *
15275 *=====*/
15276 PRIVATE int do_kill(m_ptr)
15277     register message *m_ptr; /* apunto a mensaje solicitud */
15278 {
15279     /* Manejar sys_kill(). Causar envío de una señal a un proceso vía MM.
15280     * Esto nada tiene que ver con la llamada al sistema kill (2),
15281     * así es como el FS (y quizá otros servidores) obtienen acceso a cause_sig
15282     * para enviar un mensaje KSIG a MM.
15283     */
15284
15285     if (!isokusern(m_ptr->PR)) return(E_BAD_PROC);
15286     cause_sig(m_ptr->PR, m_ptr->SIGNUM);
15287     return(OK);
15288 }

15291 /*=====
15292 *                               do_endsig                               *
15293 *=====*/
15294 PRIVATE int do_endsig(m_ptr)

```

```

15295 register message *m_ptr;          /* apunto a mensaje solicitud */
15296 {
15297     /* Asear después de señal tipo KSIG, causada por un mensaje SYS_KILL
15298     * o una llamada a cause_sig por parte de una tarea
15299     */
15300
15301     register .struct proc *rp;
15302
15303     if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
15304     rp = proc_addr(m_ptr->PROC1);
15305
15306     /* MM ha concluido una KSIG */
15307     if (rp->p_pendcount != 0 && --rp->p_pendcount == 0
15308         && (rp->p_flags &= -SIG_PENDING) == 0)
15309         lock_ready(rp);
15310     return(OK);
15311 }

15313 /*=====*
15314 *                               do_copy                               *
15315 *=====*/
15316     PRIVATE int do_copy(m_ptr)
15317     register message *m_ptr;          /* apunto a mensaje solicitud */
15318     {
15319     /* Manejar sys_copy(). Copiar datos para MM o FS. */
15320
15321     int src_proc, dst_proc, src_space, dst_space;
15322     vir_bytes src_vir, dst_vir;
15323     phys_bytes src_phys, dst_phys, bytes;
15324
15325     /* Desmembrar el mensaje del comando. */
15326     src_proc = m_ptr->SRC_PROC_NR;
15327     dst_proc = m_ptr->DST_PROC_NR;
15328     src_space = m_ptr->SRC_SPACE;
15329     dst_space = m_ptr->DST_SPACE;
15330     src_vir = (vir_bytes) m_ptr->SRC_BUFFER;
15331     dst_vir = (vir_bytes) m_ptr->DST_BUFFER;
15332     bytes = (phys_bytes) m_ptr->COPY_BYTES;
15333
15334     /* Calcular direcciones de origen y destino y hacer el copiado. */
15335     if (src_proc == ABS)
15336         src_phys = (phys_bytes) m_ptr->SRC_BUFFER;
15337     else {
15338         if (bytes != (vir_bytes) bytes)
15339             /* Esto sucedería para segmentos de 64K y vir_bytes de 16 bits.
15340             * Sucedería mucho con do_fork excepto que MM usa copias ABS
15341             * para ese caso.
15342             */
15343             panic("overflow in count in do_copy", NO_NUM);
15344         src_phys = umap(proc_addr(src_proc), src_space, src_vir,
15345             (vir_bytes) bytes);
15346     }
15347
15348     if (dst_proc == ABS)
15349         dst_phys = (phys_bytes) m_ptr->DST_BUFFER;
15350     else
15351         dst_phys = umap(proc_addr(dst_proc), dst_space, dst_vir,
15352             (vir_bytes) bytes);
15353
15354

```



```

15355 if(src_phys == 0 || dst_phys == 0) return(EFAULT);
15356 phys_copy(src_phys, dst_phys, bytes) j
15357     return(OK);
15358 }

15361 /*=====
15362*                                     do vcopy                                     *
15363*=====*/
15364     PRIVATE int do_vcopy(m_ptr)
15365                                     register message *m_ptr; /* apunto a mensaje solicitud */
15366 {
15367     /* Manejar sys_vcopy(). Copiar múltiples bloques de memoria */
15368
15369     int src_proc, dst_proc, vect_s, i;
15370     vir_bytes src_vir, dst_vir, vect_addr;
15371     phys_bytes src_phys, dst_phys, bytes;
15372     cpvec_t cpvec_table[CPVEC_NR];
15373
15374     /* Desmembrar el mensaje del comando. */
15375     src_proc = m_ptr->m1_i1;
15376     dst_proc = m_ptr->m1_i2;
15377     vect_s = m_ptr->m1_i3;
15378     vect_addr = (vir_bytes)m_ptr->m1_p1;
15379
15380     if (vect_s > CPVEC_NR) return EDOM;
15381
15382     src_phys = numap(m_ptr->m_source, vect_addr, vect_s * sizeof(cpvec_t));
15383     if (!src_phys) return EFAULT;
15384     phys_copy(src_phys, vir2phys(cpvec_table),
15385              (phys_bytes)(vect_s * sizeof(cpvec_t)));
15386
15387     for (i = 0; i < vect_s; i++) {
15388         src_vir = cpvec_table[i].cpv_src;
15389         dst_vir = cpvec_table[i].cpv_dst;
15390         bytes = cpvec_table[i].cpv_size;
15391         src_phys = numap(src_proc, src_vir, (vir_bytes)bytes);
15392         dst_phys = numap(dst_proc, dst_vir, (vir_bytes)bytes);
15393         if (src_phys == 0 || dst_phys == 0) return(EFAULT);
15394         phys_copy(src_phys, dst_phys, bytes);
15395     }
15396     return(OK);
15397 }

15400 /*=====
15401*                                     do_gboot                                     *
15402*=====*/
15403     PUBLIC struct bparam_s boot_parameters;
15404
15405     PRIVATE int do_gboot(m_ptr)
15406                                     message *m_ptr; /* apunto a mensaje solicitud */
15407 {
15408     /* Copiar paráms. de arranque. Normal. sólo se llama durante inic. de FS. */
15409
15410     phys_bytes dst_phys;
15411
15412     dst_phys = umap(proc_addr(m_ptr->PROC1), D, (vir_bytes) m_ptr->MEM_PTR,
15413                (vir_bytes) sizeof(boot_parameters));
15414     if (dst_phys == 0) panic("bad call to SYS_GBOOT", NO_NUM);

```

```

15415 phys_copy(vir2phys(&boot_parameters), dst_phys,
15416             (phys_bytes) sizeof(boot_parameters));
15417     return(OK);
15418 }

15421 /*=====
15422 *                               do_mem                               *
15423 *=====*/
15424     PRIVATE int do_mem(m_ptr)
15425     register message *m_ptr; /* apunto a mensaje solicitud */
15426 {
15427     /* Devolver base y tamaño del siguiente trozo de memoria. */
15428
15429     struct memory *memp;
15430
15431     for (memp = mem; mem < &mem[NR_MEMS]; ++memp) {
15432         m_ptr->m1_i1 = mem->base;
15433         m_ptr->m1_i2 = mem->size;
15434         m_ptr->m1_i3 = tot_mem_size;
15435         mem->size = 0;
15436         if (m_ptr->m1_i2 != 0) break;          /* encontró un trozo */
15437     }
15438     return(OK);
15439 }

15442 /*=====
15443 *                               do_umap                               *
15444 *=====*/
15445     PRIVATE int do_umap(m_ptr)
15446     register message *m_ptr; /* apunto a mensaje solicitud */
15447 {
15448     /* Igual que umap(), para procesos sin kernel. */
15449
15450     m_ptr->SRC_BUFFER = umap(proc_addr((int) m_ptr->SRC_PROC_NR),
15451                             (int) m_ptr->SRC_SPACE,
15452                             (vir_bytes) m_ptr->SRC_BUFFER,
15453                             (vir_bytes) m_ptr->COPY_BYTES);
15454     return(OK);
15455 }

15458 /*=====
15459 *                               do_trace                               *
15460 *=====*/
15461     #define TR_PROCNR      (m_ptr->m2_i1)
15462     #define TR_REQUEST    (m_ptr->m2_i2)
15463     #define TR_ADDR       ((vir_bytes) m_ptr->m2_i1)
15464     #define TR_DATA       (m_ptr->m2_i2)
15465     #define TR_VLSIZE     ((vir_bytes) sizeof(long))
15466
15467     PRIVATE int do_trace(m_ptr)
15468     register message *m_ptr;
15469 {
15470     /* Manejar los comandos de depuración apoyados por la llamada ptrace
15471     * Los comandos son:
15472     * T_STOP    detener el proceso
15473     * T_OK      habilitar rastreo por el padre de este proceso
15474     * T_GETINS  devolver valor del espacio de instrucciones

```

```

15475 * T_GETDATA devolver valor de espacio de datos
15476 * T_GETUSER devolver valor de tabla de procesos de usuario
15477 * T_SETINS fijar valor de espacio de instrucciones
15478 * T_SETDATA fijar valor de espacio de datos
15479 * T_SETUSER fijar valor en tabla de procesos de usuario
15480 * T_RESUME reanudar ejecución
15481 * T_EXLr salir
15482 * T_STEP activar bit de rastreo
15483 *
15484 * El administrador de memoria maneja totalmente los comandos T_OK
15485 * Y T_EXIT, todos los demás van aquí.
15486 */
15487
15488 register struct proc *rp;
15489 phys_bytes src, dst;
15490 int i;
15491
15492 rp = proc_addr(TR_PROCNR);
15493 if (rp->p_flags & P_SLOT_FREE) return(EIO);
15494 switch (TR_REQUEST) {
15495 case T_STOP: /* detener proceso */
15496     if (rp->p_flags == 0) lock_unready(rp);
15497     rp->p_flags |= P_STOP;
15498     rp->p_reg.psw &= -TRACEBIT; /* apagar bit de rastreo */
15499     return(OK);
15500
15501 case T_GETINS: /* devolver valor de espacio de ins. */
15502     if (rp->p_map[T].mem_len != 0) {
15503         if ((src = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
15504         phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
15505         break;
15506     }
15507     /* Espacio texto es realmente espacio datos -continuar. */
15508
15509 case T_GETDATA: /* devolver valor de espacio de datos */
15510     if ((src = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
15511     phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
15512     break;
15513
15514 case T_GETUSER: /* devolver valor de tabla de procs. */
15515     if ((TR_ADDR & (sizeof(long) - 1)) != 0 ||
15516         TR_ADDR > sizeof(struct proc) - sizeof(long))
15517         return(EIO);
15518     TR_DATA = *(long *) ((char *) rp + (int) TR_ADDR);
15519     break;
15520
15521 case T_SETINS: /* fijar valor en espacio de instruc. */
15522     if (rp->p_map[T].mem_len != 0) {
15523         if ((dst = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
15524         phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
15525         TR_DATA = 0;
15526         break;
15527     }
15528     /* Espacio texto es realmente espacio datos -continuar. */
15529
15530 case T_SETDATA: /* fijar valor en espacio de datos */
15531     if ((dst = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
15532     phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
15533     TR_DATA = 0;
15534     break;

```

```

15535
15536 case T_SETUSER:                                /* fijar valor en tabla de procs. */
15537     if ((TR_ADDR & (sizeof(reg_t) - 1))        != 0 ||
15538         TR_ADDR > sizeof(struct stackframe_s) - sizeof(reg_t))
15539         return(EIO);
15540     i = (int) TR_ADDR;
15541     #if (CHIP == INTEL)
15542         /* Alterar registros de segmentos podría causar una caída del kernel
15543          * cuando trate de cargarlos antes de reiniciar un proceso,
15544          * así que no debe permitirse.
15545          */
15546         if (i == (int) &((struct proc *) 0)->p_reg.cs ||
15547             i == (int) &((struct proc *) 0)->p_reg.ds ||
15548             i == (int) &((struct proc *) 0)->p_reg.es ||
15549             #if _WORD_SIZE == 4
15550                 i == (int) &((struct proc *) 0)->p_reg.gs |
15551                 i == (int) &((struct proc *) 0)->p_reg.fs |
15552             #endif
15553             i == (int) &((struct proc *) 0)->p_reg.ss)
15554             return(EIO);
15555     #endif
15556     if (i == (int) &((struct proc *) 0)->p_reg.psw)
15557         /* sólo bits seleccionados son intercambiables */
15558         SETPSW(rp, TR_DATA);
15559     else
15560         *(reg_t *) ((char *) &rp->p_reg + i) = (reg_t) TR_DATA;
15561     TR_DATA = 0;
15562     break;
15563
15564 case T_RESUME:                                    /* reanudar ejecución */
15565     rp->p_flags &= -P_STOP;
15566     if (rp->p_flags == 0) lock_ready(rp);
15567     TR_DATA = 0;
15568     break;
15569
15570 case T_STEP:                                      /* fijar bit de rastreo */
15571     rp->p_reg.psw |= TRACEBIT;
15572     rp->p_flags &= -P_STOP;
15573     if (rp->p_flags == 0) lock_ready(rp);
15574     TR_DATA = 0;
15575     break;
15576
15577 default:
15578     return(EIO);
15579 }
15580 return(OK);
15581 }

```

```

15583 /*=====
15584 *                                     cause_sig                                     *
15585 *=====*/
15586 PUBLIC void cause_sig(proc_nr, sig_nr)
15587     int proc_nr;    /* proceso por señalar */
15588     int sig_nr;    /* señal por enviar, 1 a _NSIG */
15589 {
15590     /* Una tarea quiere enviar 1 señal a un proc. Ejemplos de tales tareas son:
15591     * TTY quiere causar SIGINT al obtener un DEL
15592     * CLOCK quiere causar SIGALRM cuando temporizador expira
15593     * FS también usa esto p/Enviar una señal, vía el mensaje SYS KILL.
15594     * Las señales se manejan enviando un mensaje a MM. Las tareas no se atreven

```

```

15595  * a hacerlo directamente, por temor a lo que pasaría si MM estuviera
15596  * ocupado, así que llaman cause_sig, que enciende bits en p_pending y
15597  * luego ve con cuidado si MM está libre. Si así es, se le envía un mensaje.
15598  * Si no, cuando queda libre se envía un mensaje. El proceso señalizado se
15599  * bloquea si MM no ha visto o atendido todas las señales para él. Estas señales
15600  * se cuentan en p_pendcount, y la bandera SIG_PENDING se mantiene no 0 mientras hay
15601  * alguna.. No basta con preparar el proceso cuando se informa a MM, porque MM
15602  * puede bloquearse esperando que FS haga un vaciado de núcleo.
15603  */
15604
15605  register struct proc *rp, *mmp;
15606
15607  rp = proc_addr(proc_nr);
15608  if (sigismember(&rp->p_pending, sig_nr))
15609      return; /* esta señal ya está pendiente */
15610  sigaddset(&rp->p_pending, sig_nr);
15611  ++rp->p_pendcount; /* contar nueva señal pendiente */
15612  if (rp->p_flags & PENDING)
15613      return; /* otra señal ya pendiente */
15614  if (rp->p_flags == 0) lock_unready(rp);
15615  rp->p_flags |= PENDING | SIG_PENDING;
15616  ++sig_procs; /* contar nuevo proceso pendiente */
15617
15618  mmp = proc_addr(MM_PROC_NR);
15619  if ( ((mmp->p_flags & RECEIVING) == 0) || mmp->p_getfrom != ANY) return;
15620  inform();
15621  }

15624 /*=====
15625  *                                inform                                *
15626  *=====*/
15627  PUBLIC void inform()
15628  {
15629  /* Cuando el kernel detecta una señal (p.ej. DEL) o una tarea la genera (p.ej.
15630  * la del reloj para SIGALRM), se invoca cause_sig() para activar un bit
15631  * en el campo p_pending del proceso por señalizar. Luego se llama inform()
15632  * para ver si MM está ocioso y se le puede informar. Siempre que MM se
15633  * bloquea, se verifica si 'sig_procs' es dif. de 0; si así es, se invoca inform().
15634  */
15635
15636  register struct proc *rp;
15637
15638  /* MM espera nuevas entradas. Encontrar un proc. con señales pendientes. */
15639  for (rp = BEG_SERV_ADDR; rp < END_PROC_ADDR; rp++)
15640      if (rp->p_flags & PENDING) {
15641          m.m_type = KSIG;
15642          m.SIG_PROC = proc_number(rp);
15643          m.SIG_MAP = rp->p_pending;
15644          sig_procs--;
15645          if (lock_mini_send(proc_addr(HARDWARE), MM_PROC_NR, &m) != OK)
15646              panic("can't inform MM", NO_NUM);
15647          sigemptyset(&rp->p_pending); /* le toca a MM */
15648          rp->p_flags &= ~PENDING; /* sigue inhibido por SIG_PENDING */
15649          lock_pick_proc(); /* evitar retardo al planif. MM */
15650          return;
15651      }
15652  }

```

```

15655 /*=====
15656 *                               umap                               *
15657 *=====*/
15658 PUBLIC phys_bytes umap(rp, seg, vir_addr, bytes)
15659 register struct proc *rp;          /* apunto a entrada del proc en tab proc */
15660 int seg;                          /* segmento T, D o S */
15661 vir_bytes vir_addr;              /* dir. virtual en bytes dentro del segm */
15662 vir_bytes bytes;                 /* núm. bytes por copiar */
15663 {
15664     /* Calcular dirección memoria física p/dirección virtual dada. */
15665
15666     vir_clicks vc;                /* la dir. virtual está en clicks */
15667     phys_bytes pa;                /* variables intermedias como phys_bytes */
15668     phys_bytes seg_base;
15669
15670     /* Si 'seg' es D podría ser realmente S y viceversa. T si es T.
15671     * Si la dir. virtual cae en el espacio, causa un problema. En el 8088 quizá
15672     * es una referencia a pila legal, ya que el hardware no detecta "fallas de
15673     * pila". En los 8088, el espacio se llama S y se acepta, pero en otras máquinas
15674     * se llama D y se rechaza.
15675     * El Atari ST se comporta como el 8088.
15676     */
15677
15678     if (bytes <= 0) return( (phys_bytes) 0);
15679     vc = (vir_addr + bytes -1) »CLICK_SHIFT; /* últ. clic de datos */
15680
15681     if (seg != T)
15682         seg = (vc < rp->p_map[D] .mem_vir + rp->p_map[D] .mem_len ? D : S);
15683
15684     if((vir_addr»CLICK_SHIFT) >= rp->p_map[seg] .mem_vir+ rp->p_map[seg] .mem_len)
15685         return( (phys_bytes) 0 ).
15686         seg_base = (phys_bytes) rp->p_map[seg] .mem_phys;
15687     seg_base = seg_base «CLICK_SHIFT; /* origen de segmento en bytes */
15688     pa = (phys_bytes) vir_addr;
15689     pa -= rp->p_map[seg] .mem_vir « CLICK_SHIFT;
15690     return(seg_base + pa);
15691 }

15694 /*=====
15695 *                               numap                               *
15696 *=====*/
15697 PUBLIC phys_bytes numap(proc_nr, vir_addr, bytes)
15698 int proc_nr;                      /* núm. de proceso por mapear */
15699 vir_bytes vir_addr;              /* dir. virtual en bytes dentro seg D */
15700 vir_bytes bytesj                 /* núm. bytes requeridos en segmento */
15701 {
15702     /* Hacer umap() partiendo de un núm. de proceso en vez de un apuntador.
15703     * Los controladores de dispositivos usan esta función, así que no necesitan
15704     * saber de la tabla de procesos. Para ahorrar tiempo, no hay parámetro 'seg'.
15705     * El segmento siempre es D.
15706     */
15707
15708     return(umap(proc_addr(proc_nr), D, vir_addr, bytes));
15709 }
15711 #if (CHIP == INTEL)
15712 /*=====
15713 *                               alloc_segments                       *
15714 *=====*/

```

```

15715 PUBLIC void alloc_segments(rp)
15716     register struct proc *rp;
15717     {
15718     /* Sólo do_newmap invoca ésta, pero es una función aparte
15719     * porque muchas cosas dependen del hardware.
15720     */
15721
15722     phys_bytes code_bytes;
15723     phys_bytes data_bytes;
15724     int privilege;
15725
15726     if (protected_mOde) {
15727         data_bytes = (phys_bytes) (rp->p_map[S].mem_vir + rp->p_map[S].mem_len)
15728             « CLICK_SHIFT;
15729         if (rp->p_map[T].mem_len == 0)
15730             code_bytes = data_bytes;          /* I&D común, mala protección */
15731         else
15732             code_bytes = (phys_bytes) rp->p_map[T].mem_len « CLICK_SHIFT;
15733         privilege = istaskp(rp) ? TASK_PRIVILEGE : USER_PRIVILEGE;
15734         init_codeseg(&rp->p_ldt[CS_LDT_INDEX],
15735             (phys_bytes) rp->p_map[T].mem_phys « CLICK_SHIFT,
15736             code_bytes, privilege);
15737         init_dataseg(&rp->p_ldt[DS_LDT_INDEX],
15738             (phys_bytes) rp->p_map[D].mem_phys « CLICK_SHIFT,
15739             data_bytes, privilege);
15740         rp->p_reg.cs = (CS_LDT_INDEX * DESC_SIZE) | TI | privilege;
15741     #if _WORD_SIZE == 4
15742         rp->p_reg.gs =
15743         rp->p_reg.fs =
15744     #endif
15745         rp->p_reg.ss =
15746         rp->p_reg.es =
15747         rp->p_reg.ds = (DS_LDT_INDEX * DESC_SIZE) | TI | privilege;
15748     } else {
15749         rp->p_reg.cs = click_to_hcliCk(rp->p_map[T].mem_phys);
15750         rp->p_reg.ss =
15751         rp->p_reg.es =
15752         rp->p_reg.ds = Click_to_hcliCk(rp->p_map[D].mem_phys);
15753     }
15754 }
15755 #endif /* (CHIP == INTEL) */

```

```

+++++
src/mm/mm.h
+++++

```

```

15800 /* La cabeza maestra de MM. Incluye algunos otros archivos
15801 * Y define las constantes principales.
15802 */
15803 #define _POSIX_SOURCE 1 /* decir a cabeceras incluyan cosas POSIX */
15804 #define MINIX 1 /* decir a cabeceras incluyan cosas MINIX */
15805 #define =SYSTEM 1 /* decir a cabo que éste es el kernel */
15806
15807 /* Éstas son tan básicas que todos los arch. *.c las incluyen automát. */
15808 #include <minix/config.h> /* DEBE ser primero */
15809 #include <ansi.h> /* DEBE ser segundo */

```

```

15810 #include <sys/types.h>
15811 #include <minix/const.h>
15812 #include <minix/type.h>
15813
15814 #include <fcntl.h>
15815 #include <unistd.h>
15816 #include <min~x/syslib.h>
15817
15818 #include <limits.h>
15819 #include <errno.h>
15820
15821 #include "const.h"
15822 #include "type.h"
15823 #include "proto.h"
15824 #include "glo.h"

```

```

+++++
src/mm/const.h
+++++

```

```

15900 /* Constantes empleadas por el administrador de memoria. */
15901
15902 #define NO_MEM ((phys_clicks) 0) /* devuelto por alloc_mem() si mem opera */
15903
15904 #if (CHIP == INTEL && _WORD_SIZE == 2)
15905 /* Estas definiciones se usan en size ok y no se necesitan para el 386.
15906  * La granularidad del segmento de 386-es 1 para segmentos de menos de 1M y de 4096
15907 * para segmentos mayores.
15908  */
15909 #define PAGE_SIZE 16 /* cuántos bytes en 1 pág. (s.b.HCLICK_SIZE)*/
15910 #define MAX_PAGES 4096 /* cuántas págs. en el esp. de dir. virtual */
15911 #endif
15912
15913 #define printf printk
15914
15915 #define INIT_PID 1 /* núm. de id de proceso init */

```

```

+++++
src/mm/type.h
+++++

```

```

16000 /* Si hubiera definiciones de tipos locales al administrador de memoria,
16001 * estarían aquí. Sólo se incluye este archivo por simetría con el kernel
16002 * Y FS, que sí tienen definiciones de tipos locales.
16003  */
16004

```



```

+++++
                                     src/mm/proto.h
+++++

16100  /* Prototipos de funciones. */
16101
16102  struct mproc;          /* necesita tipos afuera de lista de paráms. ..kub */
16103  struct stat;
16104
16105  /* alloc.c */
16106  _PROTOTYPE( phys_clicks alloc_mem, (phys_cliCks cliCks) );
16107  _PROTOTYPE( void free_mem, (phys_clicks base, phys_clicks clicks) );
16108  _PROTOTYPE( phys_clicks max_hole, (void) );
16109  _PROTOTYPE( void mem_init, (phys_cliCks *total, phys_clicks *free) );
16110  _PROTOTYPE( phys_clicks mem_left, (void) );
16111  _PROTOTYPE( int do_brk3, (void) );
16112
16113  /* break.c */
16114  _PROTOTYPE( int adjust, (struct mproc *rmp,
16115                      vir_clicks data_clicks, vir_bytes sp) );
16116  _PROTOTYPE( int do_brk, (void) );
16117  _PROTOTYPE( int size_ok, (int file_type, vir_clicks tc, vir_clicks dc,
16118                      vir_clicks sc, vir_clicks dvir, vir_clicks s_vir) );
16119
16120  /* exec.c */
16121  _PROTOTYPE( int do_exec, (void) );
16122  _PROTOTYPE( struct mproc *find_share, (struct mproc *mp_ign, Ino_t ino,
16123                      Dev_t dev, time_t ctime) );
16124
16125  /* forkexit.c */
16126  _PROTOTYPE( int do_fork, (void) );
16127  _PROTOTYPE( int do_mm_exit, (void) );
16128  _PROTOTYPE( int do_waitpid, (void) );
16129  _PROTOTYPE( void mm_exit, (struct mproc *rmp, int exit_status) );
16130
16131  /* getset.c */
16132  _PROTOTYPE( int do_getset, (void) );
16133
16134  /* main.c */
16135  _PROTOTYPE( void main, (void) );
16136
16137  #if (MACHINE == MACINTOSH)
16138  _PROTOTYPE( phys_clicks start_click, (void) );
16139  #endif
16140
16141  _PROTOTYPE( void reply, (int proc_nr, int result, int res2, char *resp);
16142
16143  /* putk.c */
16144  _PROTOTYPE( void putk, (int c) );
16145
16146  /* signal.c */
16147  _PROTOTYPE( int do_alarm, (void) );
16148  _PROTOTYPE( int do_kill, (void) );
16149  _PROTOTYPE( int do_ksig, (void) );
16150  _PROTOTYPE( int do_pause, (void) );
16151  _PROTOTYPE( int set_alarm, (int proc_nr, int sec) );
16152  _PROTOTYPE( int check_sig, (pid_t proc_id, int signo) );
16153  _PROTOTYPE( void sig_proc, (struct mproc *rmp, int sig_nr) );
16154  _PROTOTYPE( int do_sigaction, (void) );

```

```

16155         _PROTOTYPE( int do_sigpending, (void) );
16156 _PROTOTYPE( int do sigprocmask, (void) );
16157         _PROTOTYPE( int do-sigreturn, (void) );
16158         _PROTOTYPE( int do-sigsuspend, (void) );
16159         _PROTOTYPE( int do=reboot, (void) );
16160
16161         /* trace.c */
16162 _PROTOTYPE( int do trace, (void) );
16163         _PROTOTYPE( void stop_proc, (struct mproc *rmp, int sig_nr) );
16164
16165         /* utility.c */
16166 _PROTOTYPE( int allowed, (char *name_buf, struct stat *s_buf, int mask) );
16167 _PROTOTYPE( int no sys, (void) );
16168         _PROTOTYPE( void panic, (char *format, int num) );
16169         _PROTOTYPE( void tell_fs, (int what, int p1, int p2, int p3) );

```

```

+++++
src/mm/glo.h
+++++

```

```

16200         /* EXTERN debe ser extern excepto en table.c
16201         #ifdef _TABLE
16202         #undef EXTERN
16203         #define EXTERN
16204 #endif
16205
16206         /* Variables globales. */
16207         EXTERN struct mproc *mp;          /* apunto a ranura 'mproc' de proc actual */
16208         EXTERN int dont_reply;          /* normalmente 0; 1 p/inhibir respuesta */
16209         EXTERN int procs_in_use;        /* cuántos procesos marcados IN_USE */
16210
16211         /* Los parámetros de la llamada se guardan aquí. */
16212         EXTERN message mm_in;          /* el mensaje entrante se guarda aquí. */
16213         EXTERN message mm_out;          /* el mens. de respuesta se crea aquí. */
16214         EXTERN int who;                 /* núm. de proceso del invocador */
16215         EXTERN int mm_call;             /* núm. de llamada al sistema */
16216
16217         /* Estas variables sirven para devolver resultados al invocador. */
16218         EXTERN int err_code;             /* almac. temporal de núm. de error */
16219         EXTERN int result2;              /* resultado secundario */
16220         EXTERN char *res_ptr;           /* resultado, si apuntador */
16221
16222         extern _PROTOTYPE( int (*call_vec[ ]), (void) ); /* manej. de llamo al sist. */
16223         extern char core name[ ];        /* archivo donde se prado imág. núcleo */
16224         EXTERN sigset_t core_sset;      /* qué señales causan imágenes de núcleo
*/

```

```
+++++
                                     src/mm/mproc.h
+++++

16300      /* Esta tabla tiene 1 ranura p/proceso. Contiene toda la info. de admón. de memoria
16301      * para cada proceso. Entre otras cosas, define los segmentos de texto, dato
16302      * y. pila, uids y gids, y varias banderas. El kernel y FS
16303      * tienen tablas indizadas también por proceso, y el contenido de las ranuras
16304      * corresp. se refiere al mismo proceso en las tres.
16305      */
16306
16307  EXTERN struct mproc {
16308      struct mem_map mp_seg[NR_SEGS]; /* apunta a texto, datos, pila */
16309      char mp_exitstatus;             /* almac. p/situac. si proceso sale */
16310      char mp_sigstatus;             /* almac. p/núm. señal p/proc. matado */
16311      pid_t mp_pid;                  /* id de proceso */
16312      pid_t mp_progrp;               /* pid de grupo proc. (p/señales) */
16313      pid_t mp_wpid;                 /* pid que este proceso espera */
16314      int mp_parent;                 /* índice de proceso padre */
16315
16316      /* Uids y gids reales y efectivos. */
16317      uid_t mp_realuid;              /* uid real del proceso */
16318      uid_t mp_effuid;               /* uid efectivo del proceso */
16319      gid_t mp_realgid;              /* gid real del proceso */
16320      gid_t mp_effgid;               /* gid efectivo del proceso */
16321
16322      /* Identificación de archivo para compartir. */
16323      ino_t mp_ino;                   /* número de nodo-i del archivo */
16324      dev_t mp_dev;                   /* núm. de dispositivo del sist. archivos */
16325      time_t mp_ctime.                /* nodo-i que cambió el tiempo */
16326
16327      /* Información para manejo de señales. */
16328      sigset_t mp_ignore;             /* 1 = ignorar señal, 0 = no */
16329      sigset_t mp_catch;              /* 1 = atrapar señal, 0 = no */
16330      sigset_t mp_sigmask;           /* señales que se bloquearán */
16331      sigset_t mp_sigmask2;          /* copia guardada de mp_sigmask */
16332      sigset_t mp_sigpending;        /* señales que se bloquean */
16333      struct sigaction mp_sigact[_NSIG + 1]; /* como en sigaction(2) */
16334      vir_bytes mp_sigreturn;        /* dir. de func --sigreturn de bibl. C */
16335
16336      /* Compatibilidad hacia atrás para las señales. */
16337      sighandler_t mp_func;           /* todas señales remito a 1 fcn usuario */
16338
16339      unsigned mp_flags;              /* bits de bandera */
16340      vir_bytes mp_procargs;          /* apunt a args de pila inic de proc */
16341      } mproc[NR_PROCS];
16342
16343      /* Valores de banderas */
16344      #define IN_USE      001          /* 1 si ranura 'mproc' en uso */
16345      #define WAITING     002          /* encend. por llamada siso WAIT */
16346      #define HANGING     004          /* encend. por llamada siso EXIT */
16347      #define PAUSED      010          /* encend. por llamada siso PAUSE */
16348      #define ALARM_ON    020          /* enc. al iniciarse tempor. SIGALRM */
16349      #define SEPARATE    040          /* 1 si arch. es espacio I & O separado */
16350      #define TRACED      0100         /* 1 si el proceso debe rastrearse */
16351      #define STOPPED     0200         /* 1 si proc. detenido p/rastreo */
16352      #define SIGSUSPENDED 0400        /* enc. por llamada siso SIGSUSPEND */
16353
16354      #define NIL_MPROC ((struct mproc *) 0)
```

```

+++++
                                     src/mm/param.h
+++++

```

```

16400      /* Estos nombres son sinónimos de las variables del mensaje de entrada. */
16401          #define addr      mm_in.m1_p1
16402          #define exec_name   mm_in.m1_p1
16403          #define exec_len   mm_in.m1_i1
16404          #define func       mm_in.m6_f1
16405          #define grpuid     (gid_t) mm_in.m1_i1
16406          #define namelen   mm_in.m1_i1
16407          #define pid       mm_in.m1_i1
16408          #define seconds   mm_in.m1_i1
16409          #define sig       mm_in.m6_i1
16410          #define stack_bytes mm_in.m1_i2
16411          #define stack_ptr mm_in.m1_p2
16412          #define status    mm_in.m1_i1
16413          #define usr_id    (uid_t) mm_in.m1_i1
16414          #define request   mm_in.m2_i2
16415          #define taddr    mm_in.m2_11
16416          #define data     mm_in.m2_12
16417          #define sig_nr   mm_in.m1_i2
16418          #define sig_nsa  mm_in.m1_p1
16419          #define sig_osa  mm_in.m1_p2
16420          #define sig_ret  mm_in.m1_p3
16421          #define sig_set  mm_in.m2_11
16422          #define sig_how mm_in.m2_i1
16423          #define sig_flags mm_in.m2_i2
16424          #define sig_context mm_in.m2_p1
16425      #ifdef _SIGMESSAGE
16426          #define sig_msg mm_in.m1_i1
16427      #endif
16428          #define reboot_flag mm_in.m1_i1
16429          #define reboot_code fmm_in.m1_p1
16430          #define reboot_size mm_in.m1_i2
16431
16432      /* Estos nombres son sinónimos de las variables del mensaje de salida. */
16433          #define reply_type mm_out.m_type
16434          #define reply_i1 mm_out.m2_i1
16435          #define reply_p1 mm_out.m2_p1
16436          #define ret_mask mm_out.m2_11
16437

```

```

+++++
                                     src/mm/table.c
+++++

```

```

16500      /* Este archivo contiene la tabla que sirve para transformar los números de llama
16501      * al sistema en las rutinas que las ejecutan.
16502      */
16503
16504      #define _TABLE
16505
16506      #include "mm.h"
16507      #include <minix/callnr.h>
16508      #include <signal.h>
16509      #include "mproc.h"

```

```
16510 #include "param.h"
16511
16512 /* Diversos. */
16513 char core_name[1 = "core";/* archivo donde se prado imág. núcleo */
16514
16515 _PROTOTYPE (int (*call_veC[NCALLS]) , (void)) = {
16516     no_sys,          /* 0 = no se usa */
16517     do_mm_exit,     /* 1 = exit */
16518     do_fork,        /* 2 = fork */
16519     no_sys,         /* 3 = read */
16520     no_sys,         /* 4 = write */
16521     no_sys,         /* 5 = open */
16522     no_sys,         /* 6 = clase */
16523     do_waitpid,     /* 7 = wait */
16524     no_sys,         /* 8 = creat */
16525     no_sys,         /* 9 = link */
16526     no_sys,         /* 10 = unlink */
16527     do_waitpid,    /* 11 = waitpid */
16528     no_sys,         /* 12 = chdir */
16529     no_sys,         /* 13 = time */
16530     no_sys,         /* 14 = mknod */
16531     no_sys,         /* 15 = chmod */
16532     no_sys,         /* 16 = chown */
16533     do_brk,         /* 17 = break */
16534     no_sys,         /* 18 = stat */
16535     no_sys,         /* 19 = lseek */
16536     do_getset,     /* 20 = getpid */
16537     no_sys,         /* 21 = mount */
16538     no_sys,         /* 22 = umount */
16539     do_getset,     /* 23 = setuid */
16540     do_getset,     /* 24 = getuid */
16541     no_sys,         /* 25 = stime */
16542     do_trace,      /* 26 = ptrace */
16543     do_alarm,      /* 27 = alarm */
16544     no_sys,         /* 28 = fstat */
16545     do_pause,      /* 29 = pause */
16546     no_sys,         /* 30 = utime */
16547     no_sys,         /* 31 = (stty) */
16548     no_sys,         /* 32 = (gtty) */
16549     no_sys,         /* 33 = access */
16550     no_sys,         /* 34 = (nice) */
16551     no_sys,         /* 35 = (ftime) */
16552     no_sys,         /* 36 = sync */
16553     do_kill,        /* 37 = kill */
16554     no_sys,         /* 38 = rename */
16555     no_sys,         /* 39 = mkdir */
16556     no_sys,         /* 40 = rmdir */
16557     no_sys,         /* 41 = dup */
16558     no_sys,         /* 42 = pipe */
16559     no_sys,         /* 43 = times */
16560     no_sys,         /* 44 = (prof) */
16561     no_sys,         /* 45 = no se usa */
16562     do_getset,     /* 46 = setgid */
16563     do_getset,     /* 47 = getgid */
16564     no_sys,         /* 48 = (signal) */
16565     no_sys,         /* 49 = no se usa */
16566     no_sys,         /* 50 = no se usa */
16567     no_sys,         /* 51 = (acct) */
16568     no_sys,         /* 52 = (phys) */
16569     no_sys,         /* 53 = (lock) */
```

```

16570     no_sys,          I* 54 = ioctl      */
16571     no_sys,          I* 55 = fcntl      */
16572     no_sys,          I* 56 = (mpx)      */
16573     no_sys,          I* 57 = no se usa */
16574     no_sys,          I* 58 = no se usa */
16575     do_exec,         I* 59 = execve     */
16576     no_sys,          I* 60 = umask      */
16577     no_sys,          I* 61 = chroot     */
16578     do_getset,       I* 62 = setsid    */
16579     do_getset,       I* 63 = getpgrp   */
16580
16581     do_ksig,         I* 64 = KSIG: señales orig. en el kernel */
16582     no_sys,          I* 65 = UNPAUSE   */
16583     no_sys,          I* 66 = no se usa */
16584     no_sys,          I* 67 = REVIVE    */
16585     no_sys,          I* 68 = TASK_REPLY */
16586     no_sys,          I* 69 = no se usa */
16587     no_sys,          I* 70 = no se usa */
16588     do_sigaction,    I* 71 = sigaction */
16589     do_sigsuspend,   I* 72 = sigsuspend */
16590     do_sigpending,   I* 73 = sigpending */
16591     do_sigprocmask,  I* 74 = sigprocmask */
16592     do_sigreturn,    I* 75 = sigreturn */
16593     do_reboot,       I* 76 = reboot    */
16594 };

```

```

+++++
src/mm/main.c.
+++++

```

```

16600     I* Este archivo contiene el prog. principal del adm. de memoria y algunos
16601     * procedimientos relacionados. Cuando MINIX arranca, el kernel se ejecuta un rato,
16602     * inicializándose a sí mismo y a sus tareas, y luego ejecuta MM y FS.
16603     * Ambos se inicializan hasta donde pueden. Luego FS llama a MM porque MM
16604     * debe esperar a que FS adquiera un disco en RAM. MM
16605     * pide al kernel toda la memo libre y comienza a atender solicitudes.
16606     *
16607     * Los puntos de entrada a este archivo son:
16608     *   main:   pone en marcha el MM
16609     *   reply: responder a un proceso que efectúa llamada al sistema de MM
16610     */
16611
16612     #include "mm.h"
16613     #include <minix/callnr.h>
16614     #include <minix/com.h>
16615     #include <signal.h>
16616     #include <fcntl.h>
16617     #include <sys/ioctl.h>
16618     #include "mproc.h"
16619     #include "param.h"
16620
16621     FORWARD     PROTOTYPE( void get work, (void)          );
16622     FORWARD _PROTOTYPE ( void mm_init, (void)          );
16623
16624 I*=====

```

```
16625 *                               main                               *
16626 *=====*/
16627 PUBLIC void main()
16628 {
16629     /* Rutina principal del administrador de memoria. */
16630
16631     int error;
16632
16633     mm_init();                               /* inicializar tablas del administrador de memoria */
16634
16635     /* Ciclo principal del MM- obtener trabajo y efectuarlo, eternamente. */
16636     while (TRUE) {
16637         /* Esperar un mensaje. */
16638         get_work();                           /* esperar llamada al sistema MM */
16639         mp = &mprOc[who];
16640
16641         /* Izar algunas banderas. */
16642         error = OK;
16643         dont_reply = FALSE;
16644         err_code = -999;
16645
16646         /* Si núm. de llamada válido, ejecutar la llamada. */
16647         if (mm_call < 0 || mm_call >= NCALLS)
16648             error = EBADCALL;
16649         else
16650             error = (*call_vec[mm_Call]) ();
16651
16652         /* Devolver resultados al usuario p/indicar que terminamos. */
16653         if (dont_reply) continue;             /* no resp. a EXIT ni WAIT */
16654         if (mm_call == EXEC && error == OK) continue;
16655         reply(who, error, result2, res_ptr);
16656     }
16657 }
16660 *=====*/
16661 *                               get_work                               *
16662 *=====*/
16663 PRIVATE void get_work()
16664 {
16665     /* Esperar sigte. mensaje y extraer información útil de él. */
16666
16667     if (receive(ANY, &mm_in) != OK) panic("MM receive error", NO_NUM);
16668     who = mm_in.m_source;                     /* quién envió el mensaje */
16669     mm_call = mm_in.m_type;                   /* núm. de llamada al sistema */
16670 }
16673 *=====*/
16674 *                               reply                               *
16675 *=====*/
16676 PUBLIC void reply(proc_nr, result, res2, respt)
16677     int proc_nr;                             /* proceso al cual responder */
16678     int result;                               /* resultado de llamada (norm. OK o # error)*/
16679     int res2;                                 /* resultado secundario */
16680     char *respt;                             /* resultado si apuntador */
16681 {
16682     /* Enviar una respuesta a un proceso de usuario. */
16683
16684     register struct mproc *proc_ptr;
```

```

16685
16686 proc_ptr = &mproc[proc_nr]
16687 /*
16688  * Para hacer a MM robusto, ver si el destino sigue vivo.
16689  * Debe omitirse esta verificación si el invocador es una tarea.
16690  */
16691 if ((who >=0) && ((proc_ptr->mp_flags&IN_USE) == 0 11
16692     (proc_ptr->mp_flags&HANGING)) returnj
16693
16694 reply_type = result;
16695 reply_i1 = res2;
16696 reply_p1 = respt;
16697 if (send(proc_nr, &mm_out) != OK) panic("MM can't reply", NO_NUM);
16698 }

16701 /*=====
16702  *                               mm_init                               *
16703  *=====*/
16704     PRIVATE void mm_init()
16705     {
16706         /* Inicializar el administrador de memoria. */
16707
16708     static char core_sigs[ ] = {
16709         SIGQUIT, SIGILL, SIGTRAP, SIGABRT,
16710         SIGEMT, SIGFPE, SIGUSR1, SIGSEGV,
16711         SIGUSR2, 0 };
16712     register int proc_nr;
16713     register struct mproc *rmp;
16714     register char *sig_ptr;
16715     phys_clicks ram_clicks, total_clicks, minix_clicks, free_clicks, dummy;
16716     message mess;
16717     struct mem_map kernel_map[NR_SEGS];
16718     int mem;
16719
16720     /* Crear el conj. de señales que causan vaciados de núcleo. Hacerlo estilo Posix,
16721      * así que no necesitamos conocer posiciones de bits.
16722      */
16723     sigemptyset(&core_sset);
16724     for (sig_ptr = core_sigs; *sig_ptr != 0; sig_ptr++)
16725         sigaddset (&core_sset , *sig_ptr);
16726
16727     /* Obtener mapa de memoria del kernel para ver cuánta memoria usa,
16728      * incluido el espacio entre la dir. 0 y el principio del kernel.
16729      */
16730     sys_getmap(SYSTASK, kernel_map);
16731     minix_clicks = kernel_map[S] .mem_phys + kernel_map[S] .mem_len;
16732
16733     /* Inicializar tablas del MM. */
16734     for (proc_nr = 0; proc_nr <= INIT_PROC_NR; proc_nr++) {
16735         rmp = &mproc[proc_nr];
16736         rmp->mp_flags |= IN_USE;
16737         sys_getmap(proc_nr, rmp->mp_seg);
16738         if (rmp->mp_seg[T] .mem_len != 0) rmp->mp_flags |= SEPARATE;
16739         minix_clicks += (rmp->mp_seg[S] .mem_phys + rmp->mp_seg[S] .mem_len)
16740             -rmp->mp_seg[T] .mem_phys;
16741     }
16742     mproc[INIT_PROC_NR] .mp_pid = INIT_PID;
16743     sigemptyset(&mproc[INIT_PROC_NR] .mp_ignore);
16744     sigemptyset(&mproc[INIT_PROC_NR] .mp_catch);

```



```

16745 procs_in_use = LOW_USER + 1;
16746
16747 /* Esperar que FS envíe un mensaje con tamo de disco RAM y entre "en línea".
16748 */
16749 if (receive(FS_PROC_NR, &mess) != OK)
16750     panic("MM can't obtain RAM disk size from FS", NO_NUM);
16751
16752 ram_clicks = mess.m1_i1;
16753
16754 /* Inicializar tablas a toda la memoria física. */
16755 mem_init(&total_clicks, &free_clicks);
16756
16757 /* Imprimir información de memoria. */
16758 printf("\nMemory size =%5dK      ", click_to_round_k(total_clicks));
16759 printf("MINIX =%4dK      ", click_to_round_k(minix_clicks));
16760 printf("RAM disk =%5dK      ", click_to_round_k(ram_clicks));
16761 printf("Available =%5dK\n\n", click_to_round_k(free_clicks));
16762
16763 /* Decir a FS que continúe. */
16764 if (send(FS_PROC_NR, &mess) != OK)
16765     panic("MM can't sync up with FS", NO_NUM);
16766
16767 /* Decir a la tarea de memo dónde está mi tabla de proc. para bien de ps(1). */
16768 if ((mem = open("/dev/mem", O_RDWR)) != -1) {
16769     ioctl(mem, MIOCSPSINFO, (void *) mproc);
16770     close(mem);
16771 }
16772 }

```

```

+++++
src/mm/forkexit.c
+++++

```

```

16800 /* Este archivo crea procesos (vía FORK) y los elimina (vía
16801 * EXIT/WAIT). Si un proceso se bifurca, se le asigna una nueva ranura en la tabla
16802 * 'mproc' y se hace 1 copia de la imagen de núcleo del padre plel hijo.
16803 * Luego se informa al kernel y a FS. Se quita un proceso de 'mproc'
16804 * cuando ocurren dos eventos: (1) salió o se mató por una señal, y
16805 * (2) el padre hizo un WAIT. S~ el proceso sale primero, sigue
16806 * ocupando su ranura hasta que el padre ejecuta WAIT.
16807 *
16808 * Los puntos de entrada a este archivo son:
16809 * do fork: realizar llamada al sistema FORK
16810 * do=mm_exit: realizar llamada al siso EXIT (invocando mm_exit»)
16811 * mm_exit: salir realmente
16812 * do_wait: realizar llamadas al sistema WAITPID o WAIT
16813 */
16814
16815
16816 #include "mm.h"
16817 #include <sys/wait.h>
16818 #include <minix/callnr.h>
16819 #include <signal.h>
16820 #include "mproc.h"
16821 #include "param.h"
16822
16823 #define LAST_FEW 2 /* últ. ranuras reservadas p/superusuario */
16824

```

```

16825 PRIVATE pid_t next_pid = INIT_PID+1;          /* sigte. pid por asignar */
16826
16827     FORWARD_PROTOTYPE (void cleanup, (register struct mproc *child) );
16828
16829 /*=====*/
16830 *                               do_fork                               *
16831 /*=====*/
16832     PUBLIC int do_fork()
16833     {
16834         /* El proceso al que apunta 'mp' bifurcó. Crear un proceso hijo. */
16835
16836         register struct mproc *rmp;          /* apuntador al padre */
16837         register struct mproc *rmc;          /* apuntador al hijo */
16838         int i, child_nr, t;
16839         phys_clicks prog_clicks, child_base = 0;
16840         phys_bytes prog_bytes, parent_abs, child_abs; /* Sólo Intel */
16841
16842         /* Si las tablas podrían llenarse durante FORK, ni siquiera comenzar
16843         * porque recuperarse a medio camino es una lata.
16844         */
16845         rmp = mp;
16846         if (procs_in_use == NR_PROCS) return(EAGAIN);
16847         if (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0) return(EAGAIN);
16848
16849         /* Determinar cuánta memoria repartir. Sólo hay que copiar datos y pila,
16850         * porque el segmento de texto es compartido o de longitud cero.
16851         */
16852         prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
16853         prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
16854         prog_bytes = (phys_bytes) prog_clicks « CLICK_SHIFT;
16855         if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(EAGAIN);
16856
16857         /* Crear copia de la imagen de núcleo del padre para el hijo. */
16858         child_abs = (phys_bytes) child_base « CLICK_SHIFT;
16859         parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys « CLICK_SHIFT;
16860         i = sys_copy(ABS, 0, parent_abs, ABS, 0, child_abs, prog_bytes);
16861         if (i < 0) panic("do_fork can't copy", i);
16862
16863         /* Encontrar ranura en 'mproc' para el hijo. Debe haber una. */
16864         for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++)
16865             if ( (rmc->mp_flags & IN_USE) == 0) break;
16866
16867         /* Preparar hijo y su mapa de memoria; copiar su ranura 'mproc' del padre. */
16868         child_nr = (int) (rmc - mproc);          /* núm. ranura del hijo */
16869         procs_in_use++;
16870         *rmc = *rmp;          /* copiar ranura de proc de padre a hijo */
16871
16872         rmc->mp_parent = who;          /* registrar padre del hijo */
16873         rmc->mp_flags &= -TRACED;      /* hijo no hereda situac. de rastreo */
16874         /* Un hijo con I&D separada conserva el segmento de texto del padre.
16875         * Los segmentos de datos y pila deben hacer reto a la nueva copia.
16876         */
16877         if (! (rmc->mp~flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
16878         rmc->mp_seg[D].mem_phys = child_base;
16879         rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
16880             (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
16881         rmc->mp_exitstatus = 0;
16882         rmc->mp_sigstatus = 0;
16883
16884         /* Encontrar pid libre para el hijo y ponerlo en la tabla. */

```

```

16885 do {
16886     t = 0;                               /* 't' = 0 indica pid aún libre */
16887     next_pid = (next_pid < 30000 ? next_pid + 1 : INIT_PID + 1);
16888     for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
16889         if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
16890             t = 1;
16891             break;
16892         }
16893     rmc->mp_pid = next_pid; /* asignar pid al hijo */
16894 } while (t);
16895
16896 /* Decir a kernel y FS acerca del FORK (que ya tuvo éxito). */
16897 sys_fork(who, child_nr, rmc->mp_pid, child_base); /* child_base es sólo 68K*/
16898 tell_fs(FORK, who, child_nr, rmc->mp_pid);
16899
16900 /* Informar al kernel el mapa de memoria del hijo. */
16901 sys_newmap(child_nr, rmc->mp_seg);
16902
16903 /* Contestar al hijo para despertarlo. */
16904 reply(child_nr, 0, 0, NIL_PTR);
16905 return(next_pid); /* pid del hijo */
16906 }

16909 /*=====
16910 *                               do_mm_exit                               *
16911 *=====*/
16912     PUBLIC int do_mm_exit()
16913     {
16914         /* Realizar llamada exit(status). El trabajo real lo efectúa mm_exit(),
16915         * que también se invoca cuando un proceso se mata por una señal.
16916         */
16917
16918     mm_exit(mp, status);
16919     dont_reply = TRUE; /* no resp. a proceso recién terminado */
16920     return(OK); /* código de retorno pro forma */
16921     }

16924 /*=====
16925 *                               mm_exit                               *
16926 *=====*/
16927     PUBLIC void mm_exit(rmp, exit_status)
16928         register struct mproc *rmp; /* apunto al proceso por terminar */
16929         int exit_status; /* situac. de salida del proceso (p/padre) */
16930     {
16931         /* Un proceso terminó. Liberar casi todas sus posesiones.
16932         * Si su padre está esperando, liberar el resto, si no, suspender.
16933         */
16934
16935     register int proc_nr;
16936     int parent_waiting, right_child;
16937     pid_t pidarg, procgrp;
16938     phys_clicks base, size, s; /* base y tamaño sólo en 68000 */
16939
16940     proc_nr = (int) (rmp - mproc); /* obtener núm. ranura del proceso */
16941
16942     /* Recordar grupo de procesos de un jefe de sesión. */
16943     procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;
16944

```

```

16945 /* Si el proc que salió tenía temporiz. pendiente, matarlo. */
16946 if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr, (unsigned) 0);
16947
16948 /* Decir a kernel y FS que el proceso ya no es ejecutable. */
16949 tell_fs(EXIT, proc_nr, 0, 0); /* FS puede liberar ranura de proc */
16950 sys_xit(rmp->mp_parent, proc_nr, &base, &size);
16951
16952 /* Liberar la memoria ocupada por el hijo. */
16953 if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
16954     /* Ningún otro proceso comparte el segmento de texto, así que liberar. */
16955     free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
16956 }
16957 /* Liberar los segmentos de datos y de pila. */
16958 free_mem(rmp->mp_seg[D].mem_phys,
16959         rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
16960
16961 /* Sólo puede liberarse la ranura si el padre ejecutó WAIT. */
16962 rmp->mp_exitstatus = (char) exit_status;
16963 pidarg = mproc[rmp->mp_parent].mp_wpid; /* ¿a quién esperan? */
16964 parent_waiting = mproc[rmp->mp_parent].mp_flags & WAITING;
16965 if (pidarg == -1 || pidarg == rmp->mp_pid || -pidarg == rmp->mp_procgrp)
16966     right_child = TRUE; /* hijo pasa una de 3 pruebas */
16967 else
16968     right_child = FALSE; /* hijo falla las 3 pruebas */
16969 if (parent_waiting && right_child)
16970     cleanup(rmp); /* decir a padre y lib. ranura hijo */
16971 else
16972     rmp->mp_flags |= HANGING; /* padre no espera, suspender hijo */
16973
16974 /* Si el proc tiene hijos, desheredarlos. INIT es el nuevo padre. */
16975 for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
16976     if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
16977         /* ahora 'rmp' apunta a un hijo para desheredarlo. */
16978         rmp->mp_parent = INIT_PROC_NR;
16979         parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
16980         if (parent_waiting && (rmp->mp_flags & HANGING)) cleanup(rmp);
16981     }
16982 }
16983
16984 /* Enviar susp. al gpo. de procs. del proceso si era jefe de sesión. */
16985 if (procgrp != 0) check_sig(-procgrp, SIGHUP);
16986 }

16989 /*=====
16990 * do_waitpid *
16991 *=====*/
16992 PUBLIC int do_waitpid()
16993 {
16994     /* Un proceso quiere esperar que un hijo termine. Si uno ya está esperando,
16995     * asearlo y dejar que termine esta llamada WAIT. Si no, esperar realmente.
16996     * Este código maneja tanto WAIT como WAITPID.
16997     */
16998
16999     register struct mproc *rp;
17000     int pidarg, options, children, res2;
17001
17002     /* Un proceso que invoca WAIT nunca recibe una respuesta normal
17003     * vía reply() en el ciclo principal (a menos que esté izada WNOHANG
17004     * o no haya un hijo que califique). Si un hijo ya salió, la rutina cleanup( )

```

```

17005     * envía la respuesta para despertar al invocador.
17006     */
17007
17008 /* Fíjar variables internas, dependiendo de si es WAIT o WAITPID. */
17009 pidarg      = (mm_call == WAIT ? -1 : pid);      /* 1er. parám de waitpid */
17010 options = (mm_call == WAIT ? 0: sig_nr);        /* 3er. parám de waitpid */
17011 if (.pidarg == 0) pidarg = -mp->mp_procgrp;      /* pidarg < 0 ==> gpo proc */
17012
17013 /* ¿Hay un hijo que espere ser recogido? Aquí, pidarg != 0:
17014     * pidarg > 0 indica pidarg es pid de un proc dado al cual esperar
17015     * pidarg == -1 indica esperar cualquier hijo
17016     * pidarg < -1 indica esperar cualq. hijo cuyo gpo. proc = -pidarg
17017     */
17018 children = 0;
17019 for (rp = &mproc[0] j rp < &mproc[NR_PROCS] j rp++) {
17020     if ((rp->mp_flags & IN_USE) && rp->mp_parent == who) {
17021         /* El valor de pidarg determina cuáles hijos califican. */
17022         if (pidarg > 0 && pidarg != rp->mp_pid) continue;
17023         if (pidarg < -1 && -pidarg != rp->mp_procgrp) continue;
17024
17025         children++; /* este hijo es aceptable */
17026         if (rp->mp_flags & HANGING) {
17027             /* Este hijo pasó prueba de pid y salió. */
17028             cleanup(rp); /* este hijo ya salió */
17029             dont_reply = TRUE;
17030             return(OK);
17031         }
17032         if ((rp->mp_flags & STOPPED) && rp->mp_sigstatus) {
17033             /* Este hijo pasó prueba pid y se rastrea. */
17034             res2 = 0177 1 (rp->mp_sigstatus << 8);
17035             reply(who, rp->mp_pid, res2, NIL_PTR);
17036             dont_reply = TRUE;
17037             rp->mp_sigstatus = 0;
17038             return(OK);
17039         }
17040     }
17041 }
17042
17043 /* No salió ningún hijo que califique. Esperar uno, si existe. */
17044 if (children > 0) {
17045     /* Existe al menos 1 hijo que pasa prueba pid, pero no ha salido. */
17046     if (options & WNOHANG) return(0); /* padre no quiere esperar */
17047     mp->mp_flags |= WAITING; /* padre quiere esperar */
17048     mp->mp_wpid = (pid_t) pidarg; /* guardar pid p/después */
17049     dont_reply = TRUE; /* pero no responder ahora */
17050     return(OK); /* sí -esperar que uno salga */
17051 } else {
17052     /* Ningún hijo pasa prueba pido Devolver error de inmediato. */
17053     return(ECHILD); /* no -padre no tiene hijos */
17054 }
17055 }

```

```

17058 /*=====
17059 *                               cleanup                               *
17060 *=====*/
17061 PRIVATE void cleanup(child)
17062     register struct mproc *child; /* dice cuál proceso está saliendo */
17063 {
17064     /* Terminar salida de un proc. El proceso salió o se mató por

```

```

17065 * una señal, y su padre está esperando.
17066 */
17067
17068 int exitstatus;
17069
17070 /* Despertar al padre. */
17071 exitstatus = (child->mp_exitstatus <<
17072              8) | (child->mp_sigstatus & 0377);
17073 reply(child->mp_parent, child->mp_pid, exitstatus, NIL_PTR);
17074 mproc[child->mp_parent].mp_flags &= -WAITING; /* padre ya no espera */
17075
17076 /* Liberar la entrada de tabla del proceso. */
17077 child->mp_flags = 0;
17078   procs_in_use--;
17079   }

```

```

+++++
src/mm/exec.c
+++++

```

```

17100 /* Este archivo maneja la llamada al sistema EXEC, y trabaja como sigue:
17101 * -ver si los permisos indican que el archivo se puede ejecutar
17102 * -leer la cabecera y extraer los tamaños
17103 * -traer argumentos iniciales y entorno del espacio de usuario
17104 * -asignar la memoria para el nuevo proceso
17105 * -copiar pila inicial de MM al proceso
17106 * -leer de disco segmentos de texto y datos y copiar en proceso
17107 * -ajustar bits de setuid y setgid
17108 * -arreglar tabla 'mproc'
17109 * -decir al kernel del EXEC
17110 * -guardar distancia a argc inicial (para ps)
17111 *
17112 * Los puntos de entrada a este archivo son:
17113 * do_exec: realizar la llamada al sistema EXEC
17114 * find_share: encontrar proceso cuyo segm. de texto pueda compartirse
17115 */
17116
17117 #include "mm.h"
17118 #include <sys/stat.h>
17119 #include <minix/callnr.h>
17120 #include <a.out.h>
17121 #include <signal.h>
17122 #include <string.h>
17123 #include "mproc.h"
17124 #include "param.h"
17125
17126 FORWARD _PROTOTYPE( void load_seg, (int fd, int seg, vir_bytes seg_bytes) );
17127 FORWARD _PROTOTYPE( int new_mem, (struct mproc *sh_mp, vir_bytes text_bytes,
17128   vir_bytes data_bytes, vir_bytes bss_bytes,
17129   vir_bytes stk_bytes, phys_bytes tot_bytes) );
17130 FORWARD _PROTOTYPE( void patch_ptr, (char stack [ARG_MAX ], vir_bytes base) );
17131 FORWARD _PROTOTYPE( int read_header, (int fd, int *ft, vir_bytes *text_bytes,
17132   vir_bytes *data_bytes, vir_bytes *bss_bytes,
17133   phys_bytes *tot_bytes, long *sym_bytes, vir_clicks sc,
17134   vir_bytes *pc) );
17135
17136
17137 /*=====
17138 * do_exec *
17139 *=====

```

```
17140 PUBLIC int do_exec()
17141 {
17142     /* Ejecutar llamada exeve(name, argv, envp). La biblioteca de usuario
17143     * construye una imagen de pila completa, incl. apuntadores, args, entorno, etc.
17144     * La pila se copia en un buffer dentro de MM, y luego a la nueva imagen de núcleo.
17145     */
17146
17147     register struct mproc *rmp;
17148     struct mproc *sh_mp;
17149     int m, r, fd, ft, sn;
17150     static char mbuf[ARG_MAX]; /* buffer para pila y ceros */
17151     static char name_buf[PATH_MAX]; /* archivo por ejecutar */
17152     char *new_sp, *basename;
17153     vir_bytes src, dst, text_bytes, data_bytes, bss_bytes, stk_bytes, vsp;
17154     phys_bytes tot_bytes; /* espacio total p/programa, incl. espacio */
17155     long sym_bytes;
17156     vir_clicks sc;
17157     struct stat s_buf;
17158     vir_bytes pc;
17159
17160     /* Realizar algunas comprobaciones de validez. */
17161     rmp = mp;
17162     stk_bytes = (vir_bytes) stack_bytes;
17163     if (stk_bytes > ARG_MAX) return(ENOMEM); /* pila demasiado grande */
17164     if (exec_len <= 0 || exec_len > PATH_MAX) return(EINVAL);
17165
17166     /* Obtener nombre de arch. por ejecutar y ver si es ejecutable. */
17167     src = (vir_bytes) exec_name;
17168     dst = (vir_bytes) name_buf;
17169     r = sys_copy(who, D, (phys_bytes) src,
17170                MM_PROC_NR, D, (phys_bytes) dst, (phys_bytes) exec_len);
17171     if (r != OK) return(r); /* nombre arch no en segm datos usuario */
17172     tell fs(CHDIR, who, FALSE, 0); /* cambiar a entorno FS usuario. */
17173     fd = -allOwed(name_buf, &s_buf, X_BIT); /* ¿archivo ejecutable? */
17174     if (fd < 0) return(fd); /* el archivo no era ejecutable */
17175
17176     /* Leer cabecera del archivo y extraer tamaños de segmentos. */
17177     sc = (stk_bytes + CLICK_SIZE - 1) » CLICK_SHIFT;
17178     m = read_header(fd, &ft, &text_bytes, &data_bytes, &bss_bytes,
17179                   &tot_bytes, &sym_bytes, sc, &pc);
17180     if (m < 0) {
17181         close(fd); /* algo está mal en la cabecera */
17182         return(ENOEXEC);
17183     }
17184
17185     /* Traer la pila del usuario antes de destruir imagen de núcleo vieja. */
17186     src = (vir_bytes) stack_ptr;
17187     dst = (vir_bytes) mbuf;
17188     r = sys_copy(who, D, (phys_bytes) src,
17189                MM_PROC_NR, D, (phys_bytes) dst, (phys_bytes)stk_bytes);
17190     if (r != OK) {
17191         close(fd); /* imposible traer pila (p.ej. dir. virtual mala) */
17192         return(EACCES);
17193     }
17194
17195     /* ¿Texto del proc puede compartirse con el de uno ya en ejecución? */
17196     sh_mp = find_share(rmp, s_buf.st_ino, s_buf.st_dev, s_buf.st_ctime);
17197
17198     /* Asignar nueva memoria y liberar vieja. Arreglar mapa y decir al kernel. */
17199     r = new_mem(sh_mp, text_bytes, data_bytes, bss_bytes, stk_bytes, tot_bytes);
```

```
17200 if (r != OK) {
17201     close(fd);                /* núcleo insuf. o prog demasiado grande */
17202     return(r);
17203 }
17204
17205 /* Guardar identificación del archivo pIque pueda compartirse. */
17206 rmp->mp_ino = s_buf.st_ino;
17207 rmp->mp_dev = s_buf.st_dev;
17208 rmp->mp_ctime = s_buf.st_ctime;
17209
17210 /* Parchar pila y copiarla de MM a nueva imagen de núcleo. */
17211 vsp = (vir_bytes) rmp->mp_seg[S].mem_vir « CLICK_SHIFT;
17212 vsp += (vir_bytes) rmp->mp_seg[S].mem_len « CLICK_SHIFT;
17213 vsp -= stk_bytes;
17214 patch_ptr(mbuf, vsp);
17215 src = (vir_bytes) mbuf;
17216 r = sys_copy(MM_PROC_NR, D, (phys_bytes) src,
17217             who, D, (phys_bytes) vsp, (phys_bytes)stk_bytes);
17218 if (r != OK) panic("do_exec stack copy err", NO_NUM);
17219
17220 /* Leer de disco segmentos de texto y datos. */
17221 if (sh mp l= NULL) {
17222     lseek(fd, (off_t) text_bytes, SEEK_CUR);          /* compartido: omitir texto */
17223 } else {
17224     load_seg(fd, T, text_bytes);
17225 }
17226 load_seg(fd, D, data_bytes);
17227
17228
17229 close(fd);                /* ya no se necesita arch. ejec. */
17230
17231 /* Cuidado con bits setuid/setgid. */
17232 if ((rmp->mp_flags & TRACED) == 0) { /* suprimir si rastreando */
17233     if (s_buf.st_mode & I_SET_UID_BIT) {
17234         rmp->mp_effuid = s_buf.st_uid;
17235         tell_fs(SETUID,who, (int)rmp->mp_realuid, (int)rmp->mp_effuid);
17236     }
17237     if (s_buf.st_mode & I_SET_GID_BIT) {
17238         rmp->mp_effgid = s_buf.st_gid;
17239         tell_fs(SETGID,who, (int)rmp->mp_realgid, (int)rmp->mp_effgid);
17240     }
17241 }
17242
17243 /* Guardar distancia a argc inicial (para ps) */
17244 rmp->mp_procargs = vsp;
17245
17246 /* Arreglar campos 'mproc', decir kernel exec terminó, reset atrapó señ. */
17247 for (sn = 1; sn <= _NSIG; sn++) {
17248     if (sigismember(&rmp->mpCatch, sn)) {
17249         sigdelset(&rmp->mpCatch, sn);
17250         rmp->mp_sigact[sn].sa_handler = SIG_DFL;
17251         sigemptyset(&rmp->mp_sigact[sn].sa_mask);
17252     }
17253 }
17254
17255 rmp->mp_flags &= -SEPARATE; /* apagar bit SEPARATE */
17256 rmp->mp_flags |= FT;        /* activar si archivos 1 & D separados */
17257 new_sp = (char *) vsp;
17258
17259 tell_fs(EXEC, who, 0, 0); /* que FS maneje archivos FD_CLOEXEC */
```



```
17260
17261 /* Sistema guardará línea comandos p/depurac., salida ps(1), etc. */
17262 basename = strrchr(name_buf, '1');
17263 if (basename == NULL) basen_ame = name_buf; else basename++;
17264 sys_exec(who, new_sp, rmp->mp_flags & TRACED, basename, pc);
17265     return(OK);
17266     }

17269 /*=====
17270 *                               read_header                               *
17271 *=====*/
17272     PRIVATE int read_header(fd, ft, text_bytes, data_bytes, bss_bytes,
17273                             tot_bytes, sym_bytes, sc, pc)
17274     int fd;                               /* desc. archivo p/leer archivo ejec. */
17275     int *ft; /* lugar p/devolver núm. ft */
17276     vir_bytes *text_bytes; /* lugar p/devolver tamaño texto */
17277     vir_bytes *data_bytes; /* lugar p/devolver tamaño datos inicial. */
17278     vir_bytes *bss_bytes; /* lugar p/devolver tamaño bss */
17279     phys_bytes *tot_bytes; /* lugar p/devolver tamaño total */
17280     long *sym_bytes; /* lugar p/devolver tamaño tabla símb. */
17281     vir_clicks sc; /* tamaño de pila en clics */
17282     vir_bytes *pC; /* punto entrada programa (PC inicial) */
17283     {
17284     /* Leer cabecera y extraer tamaños texto, datos, bss y total. */
17285
17286     int m, ct;
17287     vir_clicks tc, dc, s_vir, dvir;
17288     phys_clicks totc;
17289     struct exec hdr; /* aquí se lee cabecera a.out */
17290
17291     /* Leer cabecera y verificar el número mágico. La cabecera MINIX
17292     * estándar se define en <a.out.h> y consiste en 8 chars seguidos por 6 longs;
17293     * luego vienen 4 longs más que no se usan aquí.
17294     * Byte 0: número mágico 0x01
17295     * Byte 1: número mágico 0x03
17296     * Byte 2: normal = 0x10 (no verif, 0 es OK) I/D separado = 0x20
17297     * Byte 3: Tipo CPU, Intel 16 bits = 0x04, Intel 32 bits = 0x10,
17298     * Motorola = 0x0B, Sun SPARC = 0x17
17299     * Byte 4: Longitud de cabecera = 0x20
17300     * Bytes 5-7 no se usan.
17301     *
17302     * Ahora vienen los 6 longs
17303     * Bytes 8-11: tamaño de segmentos de texto en bytes
17304     * Bytes 12-15: tamaño de segm. de datos inicializado en bytes
17305     * Bytes 16-19: tamaño de bss en bytes
17306     * Bytes 20-23: punto de entrada al programa
17307     * Bytes 24-27: memo total asignada al programa (texto, datos + pila)
17308     * Bytes 28-31: tamaño de tabla de símbolos en bytes
17309     * Los longs se representan en un orden que depende de la máquina,
17310     * little-endian en el 8088, big-endian en el 68000.
17311     * La cabecera va seguida directamente de los segmentos de texto y
17312     * datos, y la tabla de símbolos (en su caso). Los tamaños se dan en
17313     * la cabecera. Exec s610 copia los segmentos de texto y datos en la
17314     * memoria. La cabecera se usa s610 aquí. La tabla de símbolos s610
17315     * sirve a un depurador y se hace caso omiso de ella aquí.
17316     */
17317
17318     if (read(fd, (char *) &hdr, A_MINHDR) != A_MINHDR) return(ENOEXEC);
17319
```

```

17320 I* Verificar número mágico, tipo de cpu y banderas. *I
17321 if (BADMAG(hdr)) return(ENOEXEC);
17322 #if (CHIP == INTEL && _WORD_SIZE == 2)
17323 if (hdr.a_cpu != A_I8086) return(ENOEXEC);
17324 #endif
17325 #if (CHIP == INTEL && _WORD_SIZE == 4)
17326 if (hdr.a_cpu != A_I80386) return(ENOEXEC);
17327 #endif
17328 if ((hdr.a_flags & ~(A_NSYM | A_EXEC | A_SEP)) != 0) return(ENOEXEC);
17329
17330 *ft = ( (hdr.a_flags & A_SEP) ? SEPARATE : 0); I* I & O separado o no *I
17331
17332 I* Obtener tamaños de texto y datos. *I
17333 *text_bytes = (vir_bytes) hdr.a_text; I* tamaño texto en bytes *I
17334 *data_bytes = (vir_bytes) hdr.a_data; I* tamaño datos en bytes *I
17335 *bss_bytes = (vir_bytes) hdr.a_bss; I* tamaño de bss en bytes *I
17336 *tot_bytes = hdr.a_total; I* total bytes asigo al programa *I
17337 *sym_bytes = hdr.a_syms; I* tamaño tabla símb. en bytes *I
17338 if (*tot_bytes == 0) return(ENOEXEC);
17339
17340 if (*ft != SEPARATE) {
17341
17342     I* Si espacio I & O no separado, todo se considera datos. Text=0*I
17343     *data_bytes += *text_bytes;
17344     *text_bytes = 0;
17345
17346 }
17347 *pc = hdr.a_entry; I* dirección p/iniciar ejecución *I
17348
17349 I* Ver si los tamaños de segmento son factibles. *I
17350 tc = ((unsigned long) *text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17351 dc = (*data_bytes + *bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17352 totc = (*tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17353 if (dc >= totc) return(ENOEXEC); I* pila debe ser al menos 1 clic *I
17354 dvir = (*ft == SEPARATE ? 0 : tc);
17355 s_vir = dvir + (totc - sc);
17356 m = size_ok(*ft, tc, dc, sc, dvir, s_vir);
17357 ct = hdr.a_hdrlen & BYTE; I* longitud de cabecera *I
17358 if (ct > A_MINHDR) lseek(fd, (off_t) ct, SEEK_SET); I* saltar cab no usada *I
17359 return(m);
17360 }

17363 I*=====*
17364 * new_mem *
17365 *=====*
17366 PRIVATE int new_mem(sh_mp, text_bytes, data_bytes, bss_bytes, stk_bytes, tot_bytes)
17367 struct mproc *sh_mp; I* puede compartirse texto con este proc *I
17368 vir_bytes text_bytes; I* tamaño segmento de texto en bytes *I
17369 vir_bytes data_bytes; I* tamaño datos inicializ. en bytes *I
17370 vir_bytes bss_bytes; I* tamaño de bss en bytes *I
17371 vir_bytes stk_bytes; I* tamaño segm. pila inicial en bytes *I
17372 phys_bytes tot_bytes; I* memo total p/asignar, incl. espacio *I
17373 {
17374     I* Asignar memoria nueva y liberar vieja. Cambiar mapa e informar nuevo mapa al
17375 * kernel. Poner en 0 bss, espacio y pila de nueva imagen núcleo.
17376     *I
17377
17378     register struct mproc *rmp;
17379     vir_clicks text_clicks, data_clicks, gap_clicks, stack_clicks, tot_clicks;

```

```
17380 phys_clicks new_base;
17381
17382 static char zero[1024];          /* para poner en cero bss */
17383 phys_bytes bytes, base, count, bss_offset;
17384
17385 /* No es necesario asignar texto si se puede compartir. */
17386 if (sh_mp != NULL) text_bytes = 0;
17387
17388 /* Adquirir la nueva memoria; c/u de las 4 partes: texto, (datos+bss),
17389 * espacio y pila ocupa núm. entero de clics, comenzando en frontera de clic.
17390 * Las partes de datos y bss se juntan sin espacio.
17391 */
17392
17393 text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) » CLICK_SHIFT;
17394 data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) » CLICK_SHIFT;
17395 stack_clicks = (stk_bytes + CLICK_SIZE - 1) » CLICK_SHIFT;
17396 tot_clicks = (tot_bytes + CLICK_SIZE - 1) » CLICK_SHIFT;
17397 gap_clicks = tot_clicks - data_clicks - stack_clicks;
17398 if ( (int) gap_clicks < 0) return(ENOMEM);
17399
17400 /* Ver si hay un agujero de tamaño suficiente. Si lo hay, podemos arriesgar
17401 * a liberar primero la vieja imagen de núcleo antes de asignar la nueva,
17402 * pues sabemos que se podrá. Si no hay suficiente, devolver fracaso.
17403 */
17404 if (text_clicks + tot_clicks > max_hole()) return(EAGAIN);
17405
17406 /* Hay sufic. memoria para nueva imagen núcleo. Liberar la vieja. */
17407 rmp = mp;
17408
17409 if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
17410     /* Ningun otro proc. comparte segmento de texto, así que liberarlo. */
17411     free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
17412 }
17413 /* Liberar segmentos de datos y de pila. */
17414 free_mem(rmp->mp_seg[D].mem_phys,
17415         rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
17416
17417 /* Ya no podemos echarnos atrás. Perdimos para siempre la imagen vieja.
17418 * La llamada debe llegar al final. Preparar e informar nuevo mapa.
17419 */
17420 new_base = alloc_mem(text_clicks + tot_clicks);          /* nueva imagen de núcleo */
17421 if (new_base == NO_MEM) panic("MM hole list is inconsistent", NO_NUM);
17422
17423 if (sh_mp != NULL) {
17424     /* Compartir segmento de texto. */
17425     rmp->mp_seg[T] = sh_mp->mp_seg[T];
17426 } else {
17427     rmp->mp_seg[T].mem_phys = new_base;
17428     rmp->mp_seg[T].mem_vir = 0;
17429     rmp->mp_seg[T].mem_len = text_clicks;
17430 }
17431 rmp->mp_seg[D].mem_phys = new_base + text_clicks;
17432 rmp->mp_seg[D].mem_vir = 0;
17433 rmp->mp_seg[D].mem_len = data_clicks;
17434 rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys + data_clicks + gap_clicks;
17435 rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + data_clicks + gap_clicks;
17436 rmp->mp_seg[S].mem_len = stack_clicks;
17437
17438
17439 sys_newmap(who, rmp->mp_seg);          /* informar nuevo mapa al kernel */
```

```

17440
17441 /* Poner en cero bss, espacio y segmento de pila. */
17442 bytes = (phys_bytes) (data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
17443 base = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
17444 bss_offset = (data_bytes » CLICK_SHIFT) << CLICK_SHIFT;
17445 base += bss_offset;
17446 bytes -= bss_offset;
17447
17448 while (bytes > 0) {
17449     count = MIN(bytes, (phys_bytes) sizeof(zero));
17450     if (sys_copy (MM_PROC_NR, D, (phys_bytes) zero,
17451                 ABS, 0, base, count) != OK) {
17452         panic("new_mem can't zero", NO_NUM);
17453     }
17454     base += count;
17455     bytes -= count;
17456 }
17457
17458 return(OK);
17459 }

17462 /*=====
17463 *                               patch_ptr                               *
17464 *=====*/
17465 PRIVATE void patch_ptr(stack, base)
17466     char stack[ARG_MAX]; /* apunto a imagen de pila dentro de MM */
17467     vir_bytes base; /* dir. virtual de base pila dentro de usuario */
17468 {
17469     /* Al ejecutar una llamada exec(name, argv, envp), el usuario
17470 * crea una imagen de pila con apuntes arg y env relativos al principio de la
17471 * pila. Ahora éstos deben reubicarse, pues la pila no está colocada en la dirección
17472 * 0 en el espacio de direcciones del usuario.
17473 */
17474
17475 char **ap, flag;
17476 vir_bytes v;
17477
17478 flag = 0; /* cuenta núm. de apuntes-0 vistos */
17479 ap = (char **) stack; /* apunta inicialmente a 'nargs' */
17480 ap++; /* ahora apunta a argv[0] */
17481 while (flag < 2) {
17482     if (ap >= (char **) &stack[ARG_MAX]) return; /* lástima */
17483     if (*ap != NIL_PTR) {
17484         v = (vir_bytes) *ap; /* v es apuntes relativo */
17485         v += base; /* reubicarlo */
17486         *ap = (char *) v; /* regresarlo */
17487     } else {
17488         flag++;
17489     }
17490     ap++;
17491 }
17492 }

17495 /*=====
17496 *                               load_seg                               *
17497 *=====*/
17498 PRIVATE void load_seg(fd, seg, seg_bytes)
17499     int fd; /* descriptor de arch del cual leer */

```

```

17500             int seg ;           /* T o D */
17501             vir_bytes seg_bytes; /* qué tan grande es el segmento */
17502     {
17503     /* Leer texto o datos del arch. ejecutable y copiar en nueva imagen de núcleo.
17504     * Este procedo es algo complicado. La forma lógica de cargar un segmento
17505     * sería leerlo bloque por bloque y copiar uno a la vez en el espacio de usuario.
17506     * Esto es muy lento, así que hacemos algo sucio aquí: enviar el espacio de usuario
17507     * Y la dirección virtual al FS en los 10 bits superiores del descriptor de archivo,
17508     * Y le pasamos la dir. virtual del usuario en vez de una dirección MM.
17509     * El FS extrae estos parámetros cuando recibe una llamada de lectura del MM,
17510     * que es el único proceso que tiene permitido usar este truco. A continuación,
17511     * el FS copia todo el segmento directamente en espacio de usuario,
17512     * pasando totalmente por alto el MM.
17513     */
17514
17515     int new_fd, bytes;
17516     char *ubuf_ptr;
17517
17518     new_fd = (who << 8) | (seg << 6) | fd;
17519     ubuf_ptr = (char *) ((vir_bytes)mp->mp_seg[seg].mem_vir << CLICK_SHIFT);
17520     while (seg_bytes != 0) {
17521         bytes = (INT_MAX / 8BLOCK_SIZE) * BLOCK_SIZE;
17522         if (seg_bytes < bytes)
17523             bytes = (int)seg_bytes;
17524         if (read(new_fd, ubuf_ptr, bytes) != bytes)
17525             break; /* error */
17526         ubuf_ptr += bytes;
17527         seg_bytes -= bytes;
17528     }
17529     }

17532 /*=====
17533 *                               find share                               *
17534 *=====*/
17535     PUBLIC struct mproc *find_share(mp_ign, ino, dev, ctime)
17536     struct mproc *mp_ign; /* proceso que no debe mirarse */
17537     ino_t ino;           /* paráms. que identif. de forma única un arch. */
17538     dev_t dev;
17539     time_t ctime ;
17540     {
17541     /* Buscar un proceso que sea el archivo <ino, dev, ctime> en ejecución.
17542     * No "encontrar" accidentalmente" mp_ign, porque es el proceso a nombre del cual
17543     * se hace esta llamada.
17544     */
17545     struct mproc *sh_mp;
17546
17547     for (sh_mp = &mproc[INIT_PROC_NR]; sh_mp < &mproc[NR_PROCS]; sh_mp++) {
17548         if ((sh_mp->mp_flags & (IN_USE | HANGING | SEPARATE))
17549             != (IN_USE | SEPARATE)) continue;
17550         if (sh_mp == mp_ign) continue;
17551         if (sh_mp->mp_ino != ino) continue;
17552         if (sh_mp->mp_dev != dev) continue;
17553         if (sh_mp->mp_ctime != ctime) continue;
17554         return sh_mp;
17555     }
17556     return(NULL);
17557     }

```

```

+++++
                                     src/mm/break.c
+++++

17600      /* El modelo MINIX de reparto de memoria reserva una cantidad fija para los
17601 * segmentos combinados de texto, datos y pila. La cantidad empleada para un proc.
17602 * .hijo creado por FORK es la misma que tenia el padre. Si el hijo hace EXEC
17603 * después, el nuevo tamaño se toma de la cabecera del archivo EXECutado.
17604 *
17605 * La organizo en memoria consiste en el segm. de texto, seguido del de datos,
17606 * seguido de un espacio (memoria no usada), seguido por el segm. de pila.
17607 * El segm. de datos crece hacia arriba y el de pila hacia abajo, asi que ambos
17608 * pueden tomar memoria del espacio. Si chocan, el proceso debe morir.
17609 * Los procedo de este arch. se ocupan del crecimiento de estos segmentos.
17610 *
17611 * Los puntos de entrada a este archivo son:
17612 *   do_brk:   llamadas BRK/SBRK p/crecer o encoger segm. de datos
17613 *   adjust:   ver si se permite un ajuste de segmento propuesto
17614 *   size_ok:  ver si los tamaños de los segmentos son factibles
17615 */
17616
17617 #include "mm.h"
17618 #include <signal.h>
17619 #include "mproc.h"
17620 #include "param.h"
17621
17622 #define DATA_CHANGED  1      /* valor bandera si cambió tamo segm. datos
*/
17623 #define STACK_CHANGED  2      /* valor bandera si cambió tamaño de pila */
17624
17625 /*=====
17626 *                               do_brk                               *
17627 *=====*/
17628     PUBLIC int do_brk()
17629     {
17630         /* Realizar la llamada al sistema brk(addr).
17631 *
17632 * La llamada se complica por el hecho de que en algunas máquinas (p.ej. 8088)
17633 * el apuntador a la pila puede crecer más allá de la base del segmento
17634 * de pila sin que nadie se dé cuenta.
17635 * El parámetro, 'addr', es la nueva dirección virtual en espacio D.
17636 */
17637
17638     register struct mproc *rmp;
17639     int r;
17640     vir_bytes v, new_sp;
17641     vir_clicks new_clicks;
17642
17643     rmp = mp;
17644     v = (vir_bytes) addr;
17645     new_clicks = (vir_clicks) (((long) v + CLICK_SIZE - 1) » CLICK_SHIFT);
17646     if (new_clicks < rmp->mp_seg[D].mem_vir) {
17647         res_ptr = (char *) -1;
17648         return(ENOMEM);
17649     }
17650     new_clicks -= rmp->mp_seg[D].mem_vir;
17651     sys_getsp(who, &new_sp);      /* pedir a kernel valor sp actual */
17652     r = adjust(rmp, new_clicks, new_sp);
17653     res_ptr = (r == OK ? addr : (char *) -1);
17654     return(r);      /* devolver nva. dirección o -1 */

```

```

17655             } Archivo: src/mrn/break.c
17658 /*=====
17659 *             adjust                                     *
17660 *=====*/
17661     PUBLIC int adjust(rmp, data_clicks, sp)
17662             register struct mproc *rmp;           /* ¿de quién se ajusta la memoria? */
17663             vir_clicks data_clicks;           /* ¿cuánto debe crecer segm. datos? */
17664     vir=bytes sp; /* nuevo valor de apunto a pila (sp) */
17665     {
17666     /* Ver si los segm. de datos y pila pueden coexistir, ajustándolos si es necesario.
17667     * Nunca se asigna ni libera memoria; se agrega o quita del espacio entre los segms.
17668     * de datos y de pila. Si el tamaño del espacio se vuelve negativo,
17669     * el ajuste de datos o pila falla y se devuelve ENOMEM.
17670     */
17671
17672     register struct mem_map *mem_sp, *mem_dp;
17673     vir_clicks sp_click, gap_base, lower, old_clicks;
17674     int changed, r, ft;
17675     long base_of_stack, delta;           /* longs evitan ciertos problemas */
17676
17677     mem_dp = &rmp->mp_seg[D];           /* apunto a mapa de segm. de datos */
17678     mem_sp = &rmp->mp_seg[S];           /* apunto a mapa de segm. de pila */
17679     changed = 0;           /* encend. si cambió uno de los seg. */
17680
17681     if(mem_sp->mem_len == 0) return(OK); /* no molestar a init */
17682
17683     /* Ver si tamaño pila se hizo negat. (sp demasiado cerca de 0xFFFF...) */
17684     base_of_stack = (long) mem_sp->mem_vir + (long) mem_sp->mem_len;
17685     sp_click = sp » CLICK_SHIFT; /* clic que contiene sp */
17686     if(sp_click >= base_of_stack) return(ENOMEM); /* sp demasiado alto */
17687
17688     /* Calcular tamaño de espacio entre segmentos de datos y de pila. */
17689     delta = (long) mem_sp->mem_vir .(long) sp_click;
17690     lower = (delta > 0 ? sp_click : mem_sp->mem_vir);
17691
17692     /* Agregar margen segurid. p/futuro crecim. pila. Imposible hacer bien. */
17693     #define SAFETV_BVTES (384 * sizeof(char *))
17694     #define SAFETV=CLICKS ((SAFETV_BVTES + CLICK_SIZE -1) / CLICK_SIZE)
17695     gap_base = mem_dp->mem_vir + data_clicks + SAFETV_CLICKS;
17696     if(lower < gap_base) return(ENOMEM); /* datos y pila chocaron */
17697
17698     /* Actualiz. long. datos (pero no su origen) a nombre de la llamada brk(). */
17699     old_clicks = mem_dp->mem_len;
17700     if(data_clicks != mem_dp->mem_len) {
17701         mem_dp->mem_len = data_clicks;
17702         changed |= DATA_CHANGED;
17703     }
17704
17705     /* Actualiz. longitud y origen de pila por cambio en apuntador a pila. */
17706     if(delta > 0) {
17707         mem_sp->mem_vir -= delta;
17708         mem_sp->mem_phys -= delta;
17709         mem_sp->mem_len += delta;
17710         changed |= STACK_CHANGED;
17711     }
17712
17713     /* ¿Caben en espacio de dir. nuevos tamaños de segm. de datos y de pila? */
17714     ft = (rmp->mp_flags & SEPARATE);

```

```

17715 r = size_ok(ft, rmp->mp_seg[T].mem_len, rmp->mp_seg[D].mem_len,
17716         rmp->mp_seg[S].mem_len, rmp->mp_seg[D].mem_vir, rmp->mp_seg[S].mem_vir);
17717 if (r == OK) {
17718     if (changed) sys_newmap((int)(rmp - mproc), rmp->mp_seg);
17719     return(OK);
17720 }
17721
17722 /* Nvos tamaños no caben o req demasiados regs pág/segm. Restaurar.*/
17723 if (changed & DATA_CHANGED) mem_dp->mem_len = old_clicks;
17724 if (changed & STACK_CHANGED) {
17725     mem_sp->mem_vir += delta;
17726     mem_sp->mem_phys += delta;
17727     mem_sp->mem_len -= delta;
17728 }
17729 return(ENOMEM);
17730 }
17731
17732 /*=====
17733 *                                     size_ok                                     *
17734 *=====*/
17735
17736 PUBLIC int size_ok(file_type, tc, dc, sc, dvir, s_vir)
17737     int file_type; /* SEPARATE o 0 */
17738     vir_clicks tC; /* tamaño texto en clics */
17739     vir_clicks dc; /* tamaño datos en clics */
17740     vir_clicks SC; /* tamaño pila en clics */
17741     vir_clicks dvir; /* dir. virtual p/inicio seg. datos */
17742     vir_clicks s_vir; /* dir. virtual p/inicio seg. pila */
17743 {
17744     /* Ver si tamaños son factibles y hay suficientes registros de segmentación.
17745     * En una máquina con 8 págs. de 8K, tamaños de texto, datos, pila
17746     * de (32K, 16K, 16K) caben, pero (33K, 17K, 13K) no, aunque lo primero
17747     * es mayor (64K) que lo segundo. Aun en el 8088 se necesita esta prueba,
17748     * pues los datos y pila no pueden exceder 4096 clics.
17749     */
17750
17751     #if (CHIP == INTEL && _WORD_SIZE == 2)
17752     int pt, pd, ps; /* tamaños segmentos en páginas */
17753
17754     pt = ((tc << CLICK_SHIFT) + PAGE_SIZE) / PAGE_SIZE;
17755     pd = ((dc << CLICK_SHIFT) + PAGE_SIZE) / PAGE_SIZE;
17756     ps = ((sc << CLICK_SHIFT) + PAGE_SIZE) / PAGE_SIZE;
17757
17758     if (file_type == SEPARATE) {
17759         if (pt > MAX_PAGES || pd + ps > MAX_PAGES) return(ENOMEM);
17760     } else {
17761         if (pt + pd + ps > MAX_PAGES) return(ENOMEM);
17762     }
17763     #endif
17764
17765     if (dvir + dc > s_vir) return(ENOMEM);
17766
17767     return(OK);
17768 }

```



```

+++++
                                     src/mm/signal.c
+++++

```

```

17800  /* Este archivo maneja señales, que son eventos asíncronos y en general
17801 * un asunto desagradable. Se pueden generar con la llamada KILL
17802 * o por el teclado (SIGINT) o el reloj (SIGALRM). En todos
17803 * los casos el control finalmente pasa a check_sig() para ver cuáles procesos
17804 * pueden señalizarse. La señalización en si la realiza sig_proc().
17805  *
17806 * Los puntos de entrada a este archivo son:
17807 *   do_sigaction:      realizar llamada al sistema SIGACTION
17808 *   do_sigpending:     realizar llamada al sistema SIGPENDING
17809 *   do_sigprocmask:    realizar llamada al sistema SIGPROCMASK
17810 *   do_sigreturn:      realizar llamada al sistema SIGRETURN
17811 *   do_sigsuspend:     realizar llamada al sistema SIGSUSPEND
17812 *   do_kill:           realizar llamada al sistema KILL
17813 *   do_ksig:           aceptar señal que se origina en el kernel (p.ej. SIGINT)
17814 *   do_alarm:          realizar llamada al sist. ALARM invocando set_alarm()
17815 *   set_alarm:         decir a la tarea del reloj que inicie o pare un temporiz.
17816 *   do_pause:          realizar la llamada al sistema PAUSE
17817 *   do_reboot:         matar todos los procesos y reorganizar el sistema
17818 *   sig_proc:          interrumpir o terminar un proceso señalizado
17819 *   check_sig:         verif. cuáles procesos señalizar con sig_proc()
17820  */
17821
17822  #include "mm.h"
17823  #include <sys/stat.h>
17824  #include <minix/callnr.h>
17825  #include <minix/com.h>
17826  #include <signal.h>
17827  #include <sys/sigcontext.h>
17828  #include <string.h>
17829  #include "mproc.h"
17830  #include "param.h"
17831
17832  #define CORE_MODE    0777    /* modo a usar en arch. de imagen núc. */
17833  #define DUMPED       0200    /* bit encend. cuando se vacía núcleo */
17834  #define DUMP_SIZE    ((INT_MAX / BLOCK_SIZE) * BLOCK_SIZE)
17835  /* tamaño buffer p/vaciado núcleo */
17836
17837  FORWARD _PROTOTYPE(
void check_pending, (void)
17838  );
17839  FORWARD _PROTOTYPE(
void dump_core, (struct mproc *rmp)
17840  );
17841  FORWARD _PROTOTYPE(
void unpause, (int pro)
17842  );
17843  *
17844  *=====
17845  PUBLIC int do_sigaction()
17846  {
17847  int r;
17848  struct sigaction svec;
17849  struct sigaction *SVp;
17850
17851  if (sig_nr == SIGKILL) return(OK);
17852  if (sig_nr < 1 || sig_nr > _NSIG) return (EINVAL);
17853  svp = &mp->mp_sigact[sig_nr];
17854  if ((struct sigaction *) sig_osa != (struct sigaction *) NULL) {

```

```

17855     r = sys_copy(MM_PROC_NR,D, (phys_bytes) svp,
17856                 who, D, (phys_bytes) sig_osa, (phys_bytes) sizeof(svec)) ;
17857     if (r != OK) return(r) ;
17858 }
17859
17860 if ((struct sigaction *) sig_nsa == (struct sigaction *) NULL) return(OK);
17861
17862 /* Leer la estructura sigaction. */
17863 r = sys_copy(who, D, (phys_bytes) sig_nsa,
17864             MM_PROC_NR, D, (phys_bytes) &svec, (phys_bytes) sizeof(svec));
17865 if (r != OK) return(r);
17866
17867 if (svec.sa_handler == SIG_IGN) {
17868     sigaddset(&mp->mp_ignore, sig_nr);
17869     sigdelset(&mp->mp_sigpending, sig_nr);
17870     sigdelset(&mp->mp_catch, sig_nr);
17871 } else {
17872     sigdelset(&mp->mp_ignore, sig_nr);
17873     if (svec.sa_handler == SIG_DFL)
17874         sigdelset(&mp->mp_catch, sig_nr);
17875     else
17876         sigaddset(&mp->mp_catch, sig_nr) j
17877 }
17878 mp->mp_sigact[sig_nr].sa_handler = svec.sa_handler;
17879 sigdelset(&svec.sa_mask, SIGKILL);
17880 mp->mp_sigact[sig_nr].sa_mask = svec.sa_mask;
17881 mp->mp_sigact[sig_nr].sa_flags = svec.sa_flags;
17882 mp->mp_sigreturn = (vir_bytes) sig_ret;
17883     return(OK);
17884 }

17886 /*=====
17887 *                               do_sigpending                               *
17888 *=====*/
17889 PUBLIC int do_sigpending()
17890 {
17891     ret_mask = (long) mp->mp_sigpending;
17892     return OK;
17893 }

17895 /*=====
17896 *                               do sigprocmask                               *
17897 *=====*/
17898 PUBLIC int do_sigprocmask()
17899 {
17900     /* La interfaz de biblioteca pasa la máscara real en sigmask_set,
17901     * no un apuntador a ella, para ahorrar un sys_copy. Así mismo,
17902     * la máscara vieja se pone en el mensaje de retorno que la interfaz
17903     * copia (si se le pide) en la dirección especificada por el usuario.
17904     *
17905     * La interfaz de biblioteca debe establecer SIG_INQUIRE si el argumento
17906     * 'act' es NULL.
17907     */
17908
17909     int i;
17910
17911     ret_mask = (long) mp->mp_sigmask;
17912
17913     switch (sig_how) {
17914         case SIG_BLOCK:

```

```

17915     sigdelset((sigset_t *)&sig_set, SIGKILL);
17916     for (i = 1; i < _NSIG; i++) {
17917         if (sigismember((sigset_t *)&sig_set, i))
17918             sigaddset(&mp->mp_sigmask, i);
17919     }
17920     break;
17921
17922     case SIG_UNBLOCK:
17923         for (i = 1; i < _NSIG; i++) {
17924             if (sigismember((sigset_t *)&sig_set, i))
17925                 sigdelset(&mp->mp_sigmask, i);
17926         }
17927         check_pending();
17928         break;
17929
17930     case SIG_SETMASK:
17931         sigdelset((sigset_t *)&sig_set, SIGKILL);
17932         mp->mp_sigmask = (sigset_t) sig_set;
17933         check_pending();
17934         break;
17935
17936     case SIG_INQUIRE:
17937         break;
17938
17939     default:
17940         return(EINVAL);
17941         break;
17942     }
17943     return OK;
17944 }

17946 /*=====
17947 *                               do_sigsuspend                               *
17948 *=====*/
17949     PUBLIC int do_sigsuspend()
17950     {
17951         mp->mp_sigmask2 = mp->mp_sigmask;        /* guardar la máscara vieja */
17952         mp->mp_sigmask = (sigset_t) sig_set;
17953         sigdelset(&mp->mp_sigmask, SIGKILL);
17954         mp->mp_flags |= SIGSUSPENDED;
17955         dont_reply = TRUE;
17956         check_pending();
17957         return OK;
17958     }

17961 /*=====
17962 *                               do_sigreturn                               *
17963 *=====*/
17964     PUBLIC int do_sigreturn()
17965     {
17966         /* Un manejador de señales de usuario acabó. Restaurar contexto
17967 * Y ver si no hay señales no bloqueadas pendientes.
17968 */
17969
17970     int r;
17971
17972     mp->mp_sigmask = (sigset_t) sig_set;
17973     sigdelset(&mp->mp_sigmask, SIGKILL);
17974

```

```

17975 r = sys_sigreturn(who, (vir_bytes) sig_context , sig_flags);
17976     check_pending();
17977     return(r);
17978 }

17980 /*=====
17981 *                                     do_kill                                     *
17982 *=====*/
17983 PUBLIC int do_kill()
17984 {
17985     /* Realizar llamada al sistema kill(pid, signo). */
17986
17987     return check_sig(pid, sig_nr);
17988 }

17991 /*=====
17992 *                                     do_ksig                                     *
17993 *=====*/
17994     PUBLIC int do_ksig()
17995     {
17996         /* Ciertas señales, como violaciones de segmentación y DEL se originan
17997 * en el kernel. Cuando el kernel las detecta, enciende bits en un mapa de bits.
17998 * Apenas el MM está esperando nuevo trabajo, el kernel le envía un mensaje con la
17999 * ranura de proceso y mapa de bits. El mensaje llega aquí. El FS también usa
18000 * este mecanismo para indicar escritura en conductos rotos (SIGPIPE).
18001 * */
18002
18003     register struct mproc *rmp;
18004     int i, proc_nr;
18005     pid_t proc_id, id;
18006     sigset_t sig_map;
18007
18008     /* Sólo el kernel puede hacer esta llamada. */
18009     if (who != HARDWARE) return(EPERM);
18010     dont_reply = TRUE; /* no responder al kernel */
18011     proc_nr = mm_in.SIG_PROC;
18012     rmp = &mproc[proc_nr];
18013     if ( (rmp->mp_flags & IN_USE) == 0 || (rmp->mp_flags & HANGING) ) return(OK);
18014     proc_id = rmp->mp_pid;
18015     sig_map = (sigset_t) mm_in.SIG_MAP;
18016     mp = &mproc[0]; /* fingir señales kernel son del MM */
18017     mp->mp_procgrp = rmp->mp_procgrp; /* obt. gpo. de procesos correcto */
18018
18019     /* Verificar cada bit p/ver si debe enviarse una señal. A diferencia
18020 * de kill(), el kernel puede reunir varias señales no relacionadas para un
18021 * proceso y pasarlas al MM de un golpe. De ahí la iteración sobre
18022 * el mapa de bits. Para SIGINT y SIGQUIT, usar proc_id 0 para indicar
18023 * una difusión al grupo de procesos del destinatario. Para SIGKILL,
18024 * usar proc_id -1 para indicar una difusión a todo el sistema.
18025 * */
18026     for (i = 1; i <= _NSIG; i++) {
18027         if (!Sigismember(&sig_map, i)) continue;
18028         switch (i) {
18029             case SIGINT:
18030             case SIGQUIT:
18031                 id = 0; break; /* difundir a grupo de procesos */
18032             case SIGKILL:
18033                 id = -1; break; /* difundir a todos excepto INIT */
18034             case SIGALRM:

```

```

18035          /* No hacer caso de SIGALRM cuando el proceso objetivo
18036             * no solicitó una alarma. Esto sólo aplica a una señal
18037             * generada por el kernel.
18038             */
18039          if ((rmp->mp_flags & ALARM_ON) == 0) continue;
18040          rmp->mp_flags &= -ALARM_ON;
18041          /* caer a la siguiente rutina */
18042          default:
18043              id = proc_id;
18044              break;
18045      }
18046      check_sig(id, i);
18047      sys_endsig(proc_nr);          /* decir al kernel que acabó */
18048  }
18049  return(OK);
18050  }

18053 /*=====
18054 *                               do_alarm                               *
18055 *=====*/
18056  PUBLIC int do_alarm()
18057  {
18058      /* Realizar la llamada al sistema alarm(seconds). */
18059
18060      return(set_alarm(who, seconds));
18061  }

18064 /*=====
18065 *                               set_alarm                               *
18066 *=====*/
18067  PUBLIC int set_alarm(proc_nr, sec)
18068          int proc_nr;          /* proceso que quiere la alarma */
18069          int sec; /* retardo (seg) antes de la señal */
18070  {
18071      /* Do_alarm usa esta rutina p/fijar temporiz. de alarma. También sirve p/apagar
18072 * el temporiz. cuando un proceso sale con el temporiz. aún encendido.
18073 */
18074
18075      message m_sig;
18076      int remaining;
18077
18078      if (sec != 0)
18079          mproc[proc_nr].mp_flags |= ALARM_ON;
18080      else
18081          mproc[proc_nr].mp_flags &= -ALARM_ON;
18082
18083      /* Decir a tarea de reloj envíe mensaje de señal cuando llegue el momento.
18084 *
18085 * Retardos largos causan muchos problemas. Primero, la llamada alarm
18086 * recibe una cuenta de segundos unsigned y la biblioteca la mutó a un int.
18087 * Eso quizá funcione, pero al regresar la bibl. convertirá unsigned "negativos"
18088 * en errores. Supuestamente nadie verifica estos errores, y se fuerza
18089 * a la llamada a seguir. Segundo, si unsigned y long tienen el mismo
18090 * tamaño, la conversión de segundos a tics fácilmente puede desbordarse.
18091 * Por último, el kernel tiene desbordam. similares al sumar tics.
18092 *
18093 * Corregir esto requiere muchas mutaciones feas para caber en los tipos
18094 * de interfaz equivocados y evitar trampas de desbordam. DELTA_TICKS

```

```

18095 * tiene el tipo correcto (clock_t) aunque se declara como long.
18096 * ¿Cómo pueden declararse debidamente variables como ésta sin una explosión
18097 * combinatoria de tipos de mensajes?
18098 */
18099 m_sig.m_type = SET_ALARM;
18100 m_sig.CLOCK_PROC_NR = proc_nr;
18101 m_sig.DELTA_TICKS = (clock_t) (HZ * (unsigned long) (unsigned) sec) ;
18102 if ( (unsigned long) m_sig.DELTA_TICKS / HZ != (unsigned) sec)
18103     m_sig.DELTA_TICKS = LONG_MAX; /* eternidad (real. CLOCK_T_MAX) */
18104 if (sendrec(CLOCK, &m_sig) != OK) panic("alarm er", NO_NUM);
18105 remaining = (int) m_sig.SECONDS_LEFT;
18106 if (remaining != m_sig.SECONDS_LEFT || remaining < 0)
18107     remaining = INT_MAX; /* el valor real no es representable */
18108 return(remaining) ;
18109 }

18112 /*=====
18113 *                               do_pause                               *
18114 *=====*/
18115 PUBLIC int do_pause()
18116 {
18117 /* Realizar la llamada al sistema pause(). */
18118
18119 mp->mp_flags |= PAUSED;
18120 dont_reply = TRUE;
18121 return(OK) ;
18122 }

18125 /*=====
18126 *                               do_reboot                               *
18127 *=====*/
18128 PUBLIC int do_reboot()
18129 {
18130 register struct mproc *rmp = mp;
18131 char monitor_code[64];
18132
18133 if (rmp->mp_effuid != SUPER_USER) return EPERM;
18134
18135 switch (reboot_flag) {
18136 case RBT_HALT:
18137 case RBT_REBOOT:
18138 case RBT_PANIC:
18139 case RBT_RESET:
18140     break;
18141 case RBT_MONITOR:
18142     if (reboot_size > sizeof(monitor_code)) return EINVAL;
18143     memset(monitor_code, 0, sizeof(monitor_code));
18144     if (sys_copy(who, D, (phys_bytes) reboot_code,
18145                MM_PROC_NR, D, (phys_bytes) monitor_code,
18146                (phys_bytes) reboot_size) != OK) return EFAULT;
18147     if (monitor_code[sizeof(monitor_code)-1] != 0) return EINVAL;
18148     break;
18149 default:
18150     return EINVAL;
18151 }
18152
18153 /* Matar todos los procesos excepto init. */
18154 check_sig(-1, SIGKILL) ;

```

```

18155
18156 tell_fs(EXIT, INIT_PROC_NR, 0, 0);          /* asear init */
18157
18158     tell_fs(SYNC,0,0,0);
18159
18160 sys_abort(reboot_flag, monitor_code);
18161 /* NOTREACHED */
18162 }

18165 /*=====*/
18166 *                sig_proc                *
18167 /*=====*/
18168     PUBLIC void sig_proc(rmp, signo)
18169 register struct mproc *rmp;          /* apunto al proceso por señalar */
18170 int signo;                          /* señal a enviar al proceso (1 a _NSIG) */
18171 {
18172     /* Enviar señal a un proceso. Ver si la señal debe atraparse, ignorarse
18173 * o bloquearse. Si debe atraparse, coordinar con KERNEL para meter
18174 * una struct sigcontext y una sigframe en la pila de quien atraparé.
18175 * Además, KERNEL restablecerá el contador de programa y el apuntador
18176 * de pila, para que la siguiente vez que el proceso se ejecute, ejecute
18177 * el manejador de señales. Cuando éste regrese, se invocará
18178 * sigreturn(2). Luego KERNEL restaurará el contexto de la señal
18179 * de la estructura sigcontext.
18180 *
18181 * Si no hay suficiente espacio en pila, matar el proceso.
18182 */
18183
18184     vir_bytes new_sp;
18185     int slot;
18186     int sigflags;
18187     struct sigmsg sm;
18188
18189     slot = (int) (rmp -mproc);
18190     if (! (rmp->mp_flags & IN_USE)) {
18191         printf("MM: signal %d sent to dead process %d\n", signo, slot);
18192         panic("", NO_NUM);
18193     }
18194     if (rmp->mp_flags & HANGING) {
18195         printf("MM: signal %d sent to HANGING process %d\n", signo, slot);
18196         panic("", NO_NUM);
18197     }
18198     if (rmp->mp_flags & TRACED && signo != SIGKILL) {
18199         /* Un proceso perseguido tiene manejo especial. */
18200         unpauses(slot);
18201         stop_proc(rmp, signo);          /* una señal lo hace que pare */
18202         return;
18203     }
18204     /* Algunas señales se ignoran por omisión. */
18205     if (sigismember(&rmp->mp_ignore, signo)) return;
18206
18207     if (sigismember(&rmp->mp_sigmask, signo)) {
18208         /* La señal debe bloquearse. */
18209         sigaddset(&rmp->mp_sigpending, signo);
18210         return;
18211     }
18212     sigflags = rmp->mp_sigact[signo].sa_flags;
18213     if (sigismember(&rmp->mp_catch, signo)) {
18214         if (rmp->mp_flags & SIGSUSPENDED)

```

```

18215         sm.sm_mask = rmp->mp_sigmask2;
18216     else
18217         sm.sm_mask = rmp->mp_sigmask;
18218     sm.sm_signo = signo;
18219     sm.sm_sighandler = (vir_bytes) rmp->mp_sigact[signo].sa_handler;
18220     sm.sm_sigreturn = rmp->mp_sigreturn;
18221     sys_getsp(slot, &new_sp);
18222     sm.sm_stkptr = new_sp;
18223
18224     /* Hacer espacio para las struct sigcontext y sigframe. */
18225     new_sp - = sizeof(struct sigcontext)
18226                + 3 * sizeof(char *) + 2 * sizeof(int);
18227
18228     if (adjust(rmp, rmp->mp_seg[D].mem_len, new_sp) != OK)
18229         goto doterminate;
18230
18231     rmp->mp_sigmask |= rmp->mp_sigact[signo].sa_mask;
18232     if (sigflags & SA_NODEFER)
18233         sigdelset(&rmp->mp_sigmask, signo);
18234     else
18235         sigaddset(&rmp->mp_sigmask, signo);
18236
18237     if (sigflags & SA_RESETHAND) {
18238         sigdelset(&rmp->mp_catch, signo);
18239         rmp->mp_sigact[signo].sa_handler = SIG_DFL;
18240     }
18241
18242     sys_sendsig(slot, &sm);
18243     sigdelset(&rmp->mp_sigpending, signo);
18244     /* Si el proceso está suspendido por PAUSE, WAIT, SIGSUSPEND, tty,
18245        * conducto, etc., liberarlo.
18246        */
18247     unpauses(slot);
18248     return;
18249 }
18250 doterminate:
18251 /* La señal no debe o no puede atraparse. Terminar el proceso. */
18252 rmp->mp_sigstatus = (char) signo;
18253 if (sigismember(&core_sset, signo)) {
18254     /* Conmutar al entorno de FS del usuario y vaciar núcleo. */
18255     tell_fs(CHDIR, slot, FALSE, 0);
18256     dump_core(rmp);
18257 }
18258 mm_exit(rmp, 0);          /* terminar el proceso */
18259 }

18262 /*=====
18263 *                               check_sig                               *
18264 *=====*/
18265     PUBLIC int check_sig(proc_id, signo)
18266                 pid_t proc_id; /* pid de proc a señalar, o 0 o -1 o -pgrp */
18267                 int signo;     /* señal a enviar al proc (0 a _NSIG) */
18268     {
18269         /* Ver si es posible enviar una señal. Quizá haya que enviar la señal
18270 * a un grupo de procs. Esta rutina es invocada por la llamada al sistema KILL,
18271 * Y también cuando el kernel atrapa DEL u otra señal.
18272        */
18273
18274     register struct mproc *rmp;

```



```
18275 int count;                                /* contar núm. de señales enviadas */
18276 int error_code;
18277
18278 if (signo < 0 || signo > _NSIG) return(EINVAL);
18279
18280 /* Devolver EINVAL en intentos por enviar SIGKILL sólo a INIT. */
18281 if (proc_id == INIT_PID && signo == SIGKILL) return(EINVAL);
18282
18283 /* Buscar en la tabla de procesos aquellos a los que se señalará. (Ver forkexit.c
18284  * en lo que toca a magia de pid.)
18285  */
18286 count = 0;
18287 error_code = ESRCH;
18288 for (rmp = &mproc[INIT_PROC_NR]; rmp < &mproc[NR_PROCS]; rmp++) {
18289     if ((rmp->mp_flags & IN_USE) == 0) continue;
18290     if (rmp->mp_flags & HANGING && signo != 0) continue;
18291
18292     /* Verificar selección. */
18293     if (proc_id > 0 && proc_id != rmp->mp_pid) continue;
18294     if (proc_id == 0 && mp->mp_procgrp != rmp->mp_procgrp) continue;
18295     if (proc_id == -1 && rmp->mp_pid == INIT_PID) continue;
18296     if (proc_id < -1 && rmp->mp_procgrp != -proc_id) continue;
18297
18298     /* Verificar permiso. */
18299     if (mp->mp_effuid != SUPER_USER
18300         && mp->mp_realuid != rmp->mp_realuid
18301         && mp->mp_effuid != rmp->mp_realuid
18302         && mp->mp_realuid != rmp->mp_effuid
18303         && mp->mp_effuid != rmp->mp_effuid) {
18304         error_code = EPERM;
18305         continue;
18306     }
18307
18308     count++;
18309     if (signo == 0) continue;
18310
18311     /* 'sig_proc' se encarga de procesar la señal, que puede ser atrapada,
18312     * bloqueada, ignorada o causar la terminación del proceso,
18313     * quizá con vaciado de núcleo.
18314     */
18315     sig_proc(rmp, signo);
18316
18317     if (proc_id > 0) break; /* sólo un proceso señalado */
18318 }
18319
18320 /* Si el proceso invocador se mató a si mismo, no contestar. */
18321 if ((mp->mp_flags & IN_USE) == 0 || (mp->mp_flags & HANGING))
18322     dont_reply = TRUE;
18323 return(count > 0 ? OK : error_code);
18324 }

18327 /*=====
18328                                     *                               *
18329 *=====*/
18330 PRIVATE void check_pending()
18331 {
18332     /* Ver si se ha desbloqueado alguna señal pendiente. La primera señal
18333     * de este tipo que se encuentre se entregará.
18334     */
```

```

18335  * Si se encuentran varias señales no enmascaradas pendientes,
18336  * todas se entregarán en orden.
18337  *
18338  * Hay varios lugares en este archivo donde se cambia la máscara
18339  * de señal. En cada uno, debe invocarse check_pending()
18340  * para ver si recién se desbloqueó alguna señal.
18341  */
18342
18343 int i;
18344
18345 for (i = 1; i < _NSIG; i++) {
18346     if (sigismember(&mp->mp_sigpending, i) &&
18347         !sigismember(&mp->mp_sigmask, i)) {
18348         sigdelset(&mp->mp_sigpending, i);
18349         sig_proc(mp, i);
18350         break;
18351     }
18352 }
18353 }

18356 /*=====
18357  *                               unpause                               *
18358  *=====*/
18359 PRIVATE void unpause(pro)
18360 int pro;
18361 {
18362     /* I* Se debe enviar una señal a un proceso. Si el proc está suspendido
18363     * en una llamada al sist, ésta debe terminarse con EINTR. Posibles
18364     * llamadas: PAUSE, WAIT, READ Y WRITE, las dos últimas p/conductos y ttys.
18365     * Primero ver si el proc está susp. por una llamada del MM. Si no, decirlo al FS
18366     * para que pueda detectar READS y WRITES de conductos, ttys y demás.
18367     *I
18368
18369     register struct mproc *rmp;
18370
18371     rmp = &mproc[pro];
18372
18373     /* I* Ver si el proceso está suspendido en una llamada PAUSE. *I
18374     if ( (rmp->mp_flags & PAUSED) && (rmp->mp_flags & HANGING) == 0) {
18375         rmp->mp_flags &= -PAUSED;
18376         reply(pro, EINTR, 0, NIL_PTR);
18377         return;
18378     }
18379
18380     /* I* Ver si el proceso está suspendido en una llamada WAIT. *I
18381     if ( (rmp->mp_flags & WAITING) && (rmp->mp_flags & HANGING) == 0) {
18382         rmp->mp_flags &= -WAITING;
18383         reply(pro, EINTR, 0, NIL_PTR);
18384         return;
18385     }
18386
18387     /* I* Ver si el proceso está suspendido en una llamada SIGSUSPEND. *I
18388     if ((rmp->mp_flags & SIGSUSPENDED) && (rmp->mp_flags & HANGING) == 0) {
18389         rmp->mp_flags &= -SIGSUSPENDED;
18390         reply(pro, EINTR, 0, NIL_PTR);
18391         return;
18392     }
18393
18394     /* I* El proceso no está susp. en una llamada MM. Pedir al FS que investigue. *I

```

```

18395     tell_fs(UNPAUSE, pro, 0, 0);
18396 }

18399 /*=====
18400 *                               dump_core                               *
18401 *=====*/
18402     PRIVATE void dump_core(rmp)
18403     register struct mproc *rmp; /* de quién se vaciará el núcleo */
18404     {
18405     /* Vaciar el núcleo en el archivo "core" si es posible. */
18406
18407     int fd, fake_fd, nr_written, seg, slot;
18408     char *buf;
18409     vir_bytes current_sp;
18410     phys_bytes left; /* cuidado: 64K podría desbordar vir_bytes */
18411     unsigned nr_to_write; /* unsigned p/arg a write() pero < INT_MAX */
18412     long trace_data, trace_off;
18413
18414     slot = (int) (rmp -mproc);
18415
18416     /* ¿Puede escribirse el archivo core? Estamos operando en el entorno FS del
18417     * usuario, así que no se requieren verificaciones de permisos especiales.
18418     */
18419     if (rmp->mp_realuid != rmp->mp_effuid) return;
18420     if ( (fd = creat(core_name, CORE_MODE)) < 0) return;
18421     rmp->mp_sigstatus != DUMPED;
18422
18423     /* Asegurarse que el segmento de pila esté actualizado.
18424     * No queremos que adjust() falle a menos que current_sp sea absurdo,
18425     * pero podría fallar por verif. de seguridad. Tampoco queremos
18426     * realmente que el adjust() para enviar una señal falle por verif. de seguridad.
18427     * Tal vez hacer que SAFETY_BYTES sea un parámetro.
18428     */
18429     sys_getsp(slot, &current_sp);
18430     adjust(rmp, rmp->mp_seg[D].mem_len, current_sp);
18431
18432     /* Escribir mapa de memoria de todos los segmentos p/iniciar archivo coreo */
18433     if (write(fd, (char *) rmp->mp_seg, (unsigned) sizeof rmp->mp_seg)
18434         != (unsigned) sizeof rmp->mp_seg) {
18435         close(fd);
18436         return;
18437     }
18438
18439     /* Escribir toda la entrada de tabla de procs. de kernel p/obtener registros. */
18440     trace_off = 0;
18441     while (sys_trace(3, slot, trace_off, &trace_data) == OK) {
18442         if (write(fd, (char *) &trace_data, (unsigned) sizeof (long))
18443             != (unsigned) sizeof (long)) {
18444             close(fd);
18445             return;
18446         }
18447         trace_off += sizeof (long);
18448     }
18449
18450     /* Iterar los segmentos y escribirlos. */
18451     for (seg = 0; seg < NR_SEGS; seg++) {
18452         buf = (char *) ((vir_bytes) rmp->mp_seg[seg].mem_vir << CLICK_SHIFT);
18453         left = (phys_bytes) rmp->mp_seg[seg].mem_len << CLICK_SHIFT;
18454         fake_fd = (slot << 8) | (seg << 6) | fd;

```

```

18455
18456     /* Iterar por un segmento, vaciándolo. */
18457     while (left != 0) {
18458         nr_to_write = (unsigned) MIN(left, DUMP_SIZE);
18459         if ( (nr_written = write(fake_fd, buf, nr_to_write)) < 0) {
18460             close(fd);
18461             return;
18462         }
18463         buf += nr_written;
18464         left -= nr_written;
18465     }
18466 }
18467 close(fd);
18468 }

```

```

+++++
src/mm/getset.c
+++++

```

```

18500     /* Este archivo maneja las 4 llamadas al sistema que obtienen y fijan uids y gids,
18501     * asi como getpid(), setsid() y getpgrp(). El código para c/u es tan pequeño
18502     * que dificilmente se justificaba hacer una función aparte
18503     * a cada una.
18504     */
18505
18506     #include "mm.h"
18507     #include <minix/callnr.h>
18508     #include <signal.h>
18509     #include "mproc.h"
18510     #include "param.h"
18511
18512     /*=====
18513     *                               do_getset                               *
18514     *=====*/
18515     PUBLIC int do_getset()
18516     {
18517         /* Manejar GETUID, GETGID, GETPID, GETPRGP, SETUID, SETGID, SETSID. Las 4
18518         * GETs y SETSID devuelven sus resultados primarios en 'r'. GETUID, GETGID
18519         * Y GETPID también devuelven resultados secundarios (los ID efectivos o el ID
18520         * del proceso padre) en 'result2', que se devuelve al usuario.
18521         */
18522
18523         register struct mproc *rmp = mp;
18524         register int r;
18525
18526         switch(mm_call) {
18527             case GETUID:
18528                 r = rmp->mp_realuid;
18529                 result2 = rmp->mp_effuid;
18530                 break;
18531
18532             case GETGID:
18533                 r = rmp->mp_realgid;
18534                 result2 = rmp->mp_effgid;
18535                 break;
18536
18537             case GETPID:
18538                 r = mproc[who].mp_pid;
18539                 result2 = mproc[rmp->mp_parent].mp_pid;

```

```
18540             break;
18541
18542     case SETUID:
18543         if (rmp->mp_realuid != usr_id && rmp->mp_effuid != SUPER_USER)
18544             return(EPERM);
18545         rmp->mp_realuid = usr_id;
18546         rmp->mp_effuid = usr_id;
18547         tell_fs(SETUID, Who, US_id, usr_id);
18548         r = OK;
18549         break;
18550
18551     case SETGID:
18552         if (rmp->mp_realgid != grp_id && rmp->mp_effuid != SUPER_USER)
18553             return(EPERM);
18554         rmp->mp_realgid = grp_id;
18555         rmp->mp_effgid = grp_id;
18556         tell_fs(SETGID, who, grp_id, grp_id);
18557         r = OK;
18558         break;
18559
18560     case SETSID:
18561         if (rmp->mp_progrp == rmp->mp_pid) return(EPERM);
18562         rmp->mp_progrp = rmp->mp_pid;
18563         tell_fs(SETSID, who, 0, 0);
18564         /*CONTINUAR EN LA SIGUIENTE RUTINA*/
18565
18566     case GETPGRP:
18567         r = rmp->mp_progrp;
18568         break;
18569
18570     default:
18571         r = EINVAL;
18572         break;
18573     }
18574     return(r);
18575 }
```

```
+++++
src/mm/trace.c
+++++
```

```
18600 /* Este archivo se encarga de la parte de la depuración que corresponde al MM,
18601 * empleando la llamada ptrace. Casi todos los comandos se pasan a la tarea
18602 * del sistema para que los termine.
18603 *
18604 * Los comandos de depuración disponibles son:
18605 * T_STOP    detener el proceso
18606 * T_OK      habilitar rastreo de este proceso por el padre
18607 * T_GETINS  devolver valor de espacio de instrucciones
18608 * T_GETDATA devolver valor de espacio de datos
18609 * T_GETUSER devolver valor de tabla de procesos de usuario
18610 * T_SETINS  fijar valor en espacio de instrucciones
18611 * T_SETDATA fijar valor en espacio de datos
18612 * T_SETUSER fijar valor en tabla de procesos de usuario
18613 * T_RESUME  reanudar ejecución
18614 * T_EXIT    salir
```

```

18615 * T_STEP          activar bit de rastreo
18616 *
18617 * Los comandos T_OK y T_EXIT se manejan aquí, y
18618 * T_RESUME y T_STEP se manejan parcialmente aquí pero la tarea del sistema
18619 * los completa. El resto corren totalmente por cuenta de la tarea del sistema.
18620 */
18621
18622     #include "mm.h"
18623     #include <sys/ptrace.h>
18624     #include <signal.h>
18625     #include "mproc.h"
18626     #include "param.h"
18627
18628             #define NIL_MPROC      ((struct mproc *) 0)
18629
18630     FORWARD_PROTOTYPE( struct mproc *findproc, (pid_t lpid) );
18631
18632 /*=====
18633 *                               do_trace                               *
18634 *=====*/
18635     PUBLIC int do_trace()
18636     {
18637     register struct mproc *child;
18638
18639     /* La llamada T_OK es efectuada por la bifurcación de hijo del depurador antes de
18640     * que ejecute el proceso por rastrear.
18641     */
18642     if (request == T_OK) { /* habil. rastreo de este proc por su padre */
18643         mp->mp_flags |= TRACED;
18644         mm_out.m2_12 = 0;
18645         return(OK);
18646     }
18647     if ((child = findproc(pid)) == NIL_MPROC || ! (child->mp_flags & STOPPED)) {
18648         return(ESRCH);
18649     }
18650     /* todas las demás llamadas son efectuadas por la bifurcación de padre del
18651     * depurador para controlar la ejecución del hijo
18652     */
18653     switch (request) {
18654     case T_EXIT:          /* salir */
18655         mm_exit(child, (int) data);
18656         mm_out.m2_12 = 0;
18657         return(OK);
18658     case T_RESUME:
18659     case T_STEP:         /* reanudar ejecución */
18660         if (data < 0 || data > _NSIG) return(EIO);
18661         if (data > 0) { /* emitir señal */
18662             child->mp_flags &= -TRACED; /* p!no desviar señal */
18663             sig_proc(child, (int) data);
18664             child->mp_flags != TRACED;
18665         }
18666         child->mp_flags &= -STOPPED;
18667         break;
18668     }
18669     if (sys_trace(request, (int) (child - mproc), taddr, &data) != OK)
18670         return(-errno);
18671     mm_out.m2_12 = data;
18672     return(OK);
18673     }

```

```

18675 /*=====
18676 *                               findproc                               *
18677 *=====*/
18678     PRIVATE struct mproc *findproc(lpid)
18679     pid_t lpid;
18680     {
18681     register struct mproc *rmp;
18682
18683     for (rmp = &mproc[INIT_PROC_NR + 1]; rmp < &mproc[NR_PROCS]; rmp++)
18684         if (rmp->mp_flags & IN_USE && rmp->mp_pid == lpid) return(rmp);
18685     return(NIL_MPROC);
18686     }

```

```

18688 /*=====
18689 *                               stop_proc                               *
18690 *=====*/
18691     PUBLIC void stop_proc(rmp, signo)
18692     register struct mproc *rmp;
18693     int signo;
18694     {
18695     /* Un proceso rastreado recibió una señal, así que detenerlo. */
18696
18697     register struct mproc *rpmp = mproc + rmp->mp_parent;
18698
18699     if (sys_trace(-1, (int) (rmp - mproc), 0L, (long *) 0) != OK) return;
18700     rmp->mp_flags != STOPPED;
18701     if (rpmp->mp_flags & WAITING) {
18702         rpmp->mp_flags &= -WAITING;          /* el padre ya no espera */
18703         reply(rpmp->mp_parent, rmp->mp_pid, 0177 | (signo << 8), NIL_PTR);
18704     } else {
18705         rmp->mp_sigstatus = signo;
18706     }
18707     return;
18708     }

```

```

+++++
src/mm/alloc.c
+++++

```

```

18800     /* Este archivo asigna y libera bloques de memoria física de tamaño
18801     * arbitrario a nombre de las llamadas FORK y EXEC. La estructura de datos
18802     * clave empleada es la tabla de agujeros, que mantiene una lista de agujeros
18803     * de memoria, ordenada por dir. de memoria ascendente. Las direcciones
18804     * que contiene se refieren a memo fís., comenzando en la dir. absoluta 0
18805     * (no son relativas al principio del MM). Durante la inicialización
18806     * del sistema, la parte de la memoria que contiene los vectores de int.,
18807     * el kernel y el MM se "asignan" p/marcarlos como no disponibles
18808     * Y eliminarlos de la lista de agujeros.
18809     *
18810     * Los puntos de entrada a este archivo son:
18811     *     alloc_mem:  asignar un trozo de memoria del tamaño especificado
18812     *     free_mem:   liberar un trozo de memoria previamente asignado
18813     *     mem_init:   inicializar las tablas cuando MM inicia
18814     *     max_hole:   devuelve el agujero más grande disponible
18815     */
18816
18817     #include "mm.h"
18818     #include <minix/com.h>
18819

```

```

18820         #define NR_HOLES    128    /* # máx. entradas en tabla de agujeros */
18821     #define NIL_HOLE (struct hole *) 0
18822
18823     PRIVATE struct hole {
18824     phys_clicks h_base;           /* ¿dónde comienza el agujero? */
18825     phys_clicks h_len;           /* ¿qué tamaño tiene? */
18826     struct hole *h_next;         /* apunt a sigte entrada de lista */
18827     } hole[NR_HOLES];
18828
18829
18830     PRIVATE struct hole *hole_head; /* apuntador a 1er agujero */
18831     PRIVATE struct hole *free_slots; /* apunt a lista ranuras tabla desocup */
18832
18833     FORWARD_PROTOTYPE( void del_slot, (struct hole *prev_ptr, struct hole *hp) );
18834     FORWARD_PROTOTYPE( void merge, (struct hole *hp) );
18835
18836
18837 /*=====
18838 *                               alloc_mem                               *
18839 *=====*/
18840     PUBLIC phys_clicks alloc_mem(clicks)
18841     phys_clicks clicks;           /* cantidad de memoria solicitada */
18842     {
18843     /* Asignar un bloque de memoria de la lista libre empleando primer ajuste. El bloque
18844     * es una secuencia de bytes contiguos, cuya longitud en clics está dada por
18845     * 'clicks'. Se devuelve un apuntador al bloque. El bloque siempre está en una
18846     * frontera de clic. Este procedimiento se invoca cuando se requiere memoria
18847     * para FORK o EXEC.
18848     */
18849
18850     register struct hole *hp, *prev_ptr;
18851     phys_clicks old_base;
18852
18853     hp = hole_head;
18854     while (hp != NIL_HOLE) {
18855         if (hp->h_len >= clicks) {
18856             /* Se halló un agujero sufic. grande. Usarlo. */
18857             old_base = hp->h_base; /* recordar dónde comenzaba */
18858             hp->h_base += clicks; /* arrancarle un trozo */
18859             hp->h_len -= clicks; /* ídem */
18860
18861             /* Si no se usa todo el agujero, reducir tamaño y regresar. */
18862             if (hp->h_len != 0) return(old_base);
18863
18864             /* Se usó todo el agujero. Manipular lista libre. */
18865             del_slot(prev_ptr, hp);
18866             return(old_base);
18867         }
18868
18869         prev_ptr = hp;
18870         hp = hp->h_next;
18871     }
18872     return(NO_MEM);
18873 }
18874
18875 /*=====
18876 *                               free_mem                               *
18877 *=====*/
18878 /*=====*/
18879     PUBLIC void free_mem(base, clicks)

```



```

18880 phys_clicks base;          /* dir. base del bloque por liberar */
18881 phys_clicks clicks;       /* núm. clics por liberar */
18882 {
18883     /* Devolver un bloque de mem libre a la lista de agujeros. Los paráms
18884     * dicen dónde comienza el bloque en mem fis y su tamaño. El bloque se agrega
18885     * a la lista de agujeros si es contiguo a un agujero existente por cualquier
18886     * extr&mo, se fusiona con el o los agujeros.
18887     */
18888
18889     register struct hole *hp, *new_ptr, *prev_ptr;
18890
18891     if (clicks == 0) return;
18892     if ( (new_ptr = free_slots) == NIL_HOLE) panic("Hole table full", NO_NUM);
18893     new_ptr->h_base = base;
18894     new_ptr->h_len = clicks;
18895     free_slots = new_ptr->h_next;
18896     hp = hole_head;
18897
18898     /* Si la dirección de este bloque es numéricamente menor que el agujero más bajo
18899     * disponible, o si no hay agujeros disponibles, colocar este agujero al frente
18900     * de la lista de agujeros.
18901     */
18902     if (hp == NIL_HOLE || base <= hp->h_base) {
18903         /* Bloque por liberar va al frente de lista agujeros. */
18904         new_ptr->h_next = hp;
18905         hole_head = new_ptr;
18906         merge(new_ptr);
18907         return;
18908     }
18909
18910     /* Bloque por devolver no va al frente de lista de agujeros. */
18911     while (hp != NIL_HOLE && base > hp->h_base) {
18912         prev_ptr = hp;
18913         hp = hp->h_next;
18914     }
18915
18916     /* Encontramos dónde va. Insertar bloque después de 'prev_ptr'. */
18917     new_ptr->h_next = prev_ptr->h_next;
18918     prev_ptr->h_next = new_ptr;
18919     merge(prev_ptr);          /* el orden es 'prev_ptr', 'new_ptr', 'hp' */
18920 }

18923 /*=====
18924 *                               del slot                               *
18925 *=====*/
18926 PRIVATE void del_slot(prev_ptr, hp)
18927     register struct hole *prev_ptr; /* apunt a entrada aguj. justo adel 'hp' */
18928     register struct hole *hp;     /* apunt a entrada aguj por eliminar */
18929 {
18930     /* Quitar una entrada de la lista de agujeros. Se invoca este procedo cuando
18931     * una solicitud de asignar memoria gasta todo un agujero, lo que reduce
18932     * el núm de agujeros y requiere la eliminación de una entrada
18933     * de la lista de agujeros.
18934     */
18935
18936     if (hp == hole_head)
18937         hole_head = hp->h_next;
18938     else
18939         prev_ptr->h_next = hp->h_next;

```

```
18940
18941 hp->h_next = free_slots;
18942 free_slots = hp;
18943 }

18946 /*=====
18947 *                                     merge                                     *
18948 *=====*/
18949     PRIVATE void merge(hp)
18950         register struct hole *hp; /* apunt a agujero p/fusionar con suceso */
18951     {
18952     /* Ver si hay agujeros contiguos y fusionarlos si se hallan pueden ocurrir
18953 * cuando se libera un bloque de memoria y está junto a otro agujero
18954 * por cualquier extremo, o ambos. 'hp' apunta al primero de una serie
18955 * de tres agujeros que quizá podrían fusionarse.
18956 */
18957
18958     register struct hole *next_ptr;
18959
18960     /* Si 'hp' apunta al últ. agujero, no hay fusión posible. Si no,
18961     * tratar de absorber en él su sucesor y liberar la entrada de tabla del sucesor.
18962     */
18963     if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
18964     if (hp->h_base + hp->h_len == next_ptr->h_base) {
18965         hp->h_len += next_ptr->h_len; /* El 1ero obtiene mem del 2o */
18966         del_slot(hp, next_ptr);
18967     } else {
18968         hp = next_ptr;
18969     }
18970
18971     /* Si 'hp' ahora apunta al último agujero, regresar; si no, tratar
18972     * de absorber al sucesor.
18973     */
18974     if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
18975     if (hp->h_base + hp->h_len == next_ptr->h_base) {
18976         hp->h_len += next_ptr->h_len;
18977         del_slot(hp, next_ptr);
18978     }
18979     }

18982 /*=====
18983 *                                     max_hole                                     *
18984 *=====*/
18985     PUBLIC phys_clicks max_hole()
18986     {
18987     /* Examinar lista de agujeros y devolver el más grande. */
18988
18989     register struct hole *hp;
18990     register phys_clicks max;
18991
18992     hp = hole_head;
18993     max = 0;
18994     while (hp != NIL_HOLE) {
18995         if (hp->h_len > max) max = hp->h_len;
18996         hp = hp->h_next;
18997     }
18998     return(max) ;
18999     }
```

```
19002 /*=====*/
19003 *                mem_init                *
19004 /*=====*/
19005     PUBLIC void mem_init(total, free)
19006     phys_clicks *total, *free;           /* resúmenes de tamaño de memoria */
19007     {
19008         /* Inicializar listas de agujeros. Hay dos: 'hole_head' apunta a una lista
19009 * enlazada de todos los agujeros (mem desocupada) del sistema; 'free_slots'
19010 * apunta a una lista enlazada de entradas de tabla no en uso. Al principio,
19011 * la primera lista tiene 1 entrada p/trozo de mem física, y la 2a. enlaza
19012 * las ranuras de tabla restantes. Al fragmentarse la memoria con el tiempo
19013 * (los agujeros grandes iniciales se dividen en agujeros más chicos),
19014 * se requieren más ranuras de tabla p/representarlos. Estas ranuras se toman
19015 * de la lista encabezada por 'free_slots'.
19016     */
19017
19018     register struct hole *hp;
19019     phys_clicks base;                    /* dirección base del trozo */
19020     phys_clicks size;                    /* tamaño del trozo */
19021     message mess;
19022
19023     /* Poner todos los agujeros en la lista libre. */
19024     for (hp = &hole[0]; hp < &hole[NR_HOLES]; hp++) hp->h_next = hp + 1;
19025     hole[NR_HOLES-1].h_next = NIL_HOLEj
19026     hole_head = NIL_HOLE;
19027     free_slots = &hole[0];
19028
19029     /* Pedir al kernel trozos de memoria física y asignar un agujero a c/u. La llamada
19030 * SYS_MEM responde con la base y el tamaño del siguiente trozo
19031 * Y la cantidad total de memoria.
19032     */
19033     *free = 0;
19034     for (; ;) {
19035         mess.m_type = SYS_MEM;
19036         if (sendrec(SYSTASK, &mess) != OK) panic("bad SYS_MEM?", NO_NUM);
19037         base = mess.m1_i1;
19038         size = mess.m1_i2;
19039         if (size == 0)-break;           /* ¿no hay más? */
19040
19041         free_mem(base, size);
19042         *total = mess.m1_i3;
19043         *free += size;
19044     }
19045 }
```

```

+++++
+++++                               src/mm/utility.c                               +++++
+++++
19100  /* Este archivo contiene algunas rutinas de utileria para MM.
19101  *
19102  * Los puntos de entrada son:
19103  *   allowed:   ver si se permite un acceso
19104  *   no_sys:    se llama si el núm de llamada al sistema no es válido
19105  *   panic:     MM encontró un error fatal y no puede continuar
19106  *   tell_fs:   interfaz con el sistema de archivos
19107  */
19108
19109  #include "mm.h"
19110  #include <sys/stat.h>
19111  #include <minix/callnr.h>
19112  #include <minix/com.h>
19113  #include <fcntl.h>
19114  #include <signal.h>           /* sólo porque mproc.h la necesita */
19115  #include "mproc.h"
19116
19117  /*=====
19118  *                               allowed                               *
19119  *=====*/
19120  PUBLIC int allowed(name_buf, s_buf, mask)
19121  char *name_buf;           /* apunto a nombre arch. por EXECutar */
19122  struct stat *s_buf;       /* buffer p/crear y devolver struct stat*/
19123  int mask;                 /* R_BIT, W_BIT o X_BIT */
19124  {
19125  /* Ver si puede accederse al archivo. Devolver EACCES o ENOENT si acceso prohibido.
19126  * Si es legal, abrir archivo y devolver descriptor de archivo.
19127  */
19128
19129  int fd;
19130  int save_errno;
19131
19132  /* Aprovechar que la máscara de access() es igual a la de permisos. P.ej. X_BIT
19133  * en <minix/const.h> es igual a X_OK en <unistd.h> y S_IXOTH en <sys/stat.h>.
19134  * tell_fs(DO_CHDIR, ...) igualó los ids reales de MM a los ids efectivos
19135  * del usuario, así que access() funciona bien para programas setuid.
19136  */
19137  if (access(name_buf, mask) < 0) return(-errno);
19138
19139  /* Archivo accesible pero tal vez no legible. Hacerlo legible. */
19140  tell_fs(SETUID, MM_PROC_NR, (int) SUPER_USER, (int) SUPER_USER);
19141
19142  /* Abrir arch. y aplicarle fstat. Restaurar ids pronto p/manejar errores. */
19143  fd = open(name_buf, O_RDONLY);
19144  save_errno = errno;       /* open podría fallar, p.ej. de ENFILE */
19145  tell_fs(SETUID, MM_PROC_NR, (int) mp->mp_effuid, (int) mp->mp_effuid);
19146  if (fd < 0) return(-save_errno);
19147  if (fstat(fd, s_buf) < 0) panic("allowed: fstat failed", NO_NUM);
19148
19149  /* Sólo pueden ejecutarse archivos normales. */
19150  if (mask == X_BIT && (s_buf->st_mode & I_TYPE) != I_REGULAR) {
19151      close(fd);
19152      return(EACCES);
19153  }
19154  return(fd);

```

```
19155     }
19158 /*=====
19159 *                                     no_sys                                     *
19160 *=====*/
19161     PUBLIC int no_sys()
19162     {
19163     /* Se solicitó un número de llamada al sistema no implementado por MM. */ 19164
19165     return(EINVAL);
19166     }

19169 /*=====
19170 *                                     panic                                     *
19171 *=====*/
19172     PUBLIC void panic(format, num)
19173     char *format; /* cadena de formato */
19174     int num; /* núm que va con cadena de foro */
19175     {
19176     /* Algo terrible sucedió. Se causan pánicos cuando se detecta una inconsistencia
19177     * interna, p.ej., un error de programación o un valor no permitido
19178     * de una constante definida.
19179     */
19180
19181     printf("Memory manager panic: %s ", format);
19182     if (num != NO_NUM) printf("%d", num);
19183     printf("\n");
19184     tell_fs(SYNC, 0, 0, 0); /* vaciar el caché al disco */
19185     sys_abort(RBT_PANIC);
19186     }

19189 /*=====
19190 *                                     tell_fs                                     *
19191 *=====*/
19192     PUBLIC void tell_fs(what, p1, p2, p3)
19193     int what, p1, p2, p3;
19194     {
19195     /* MM sólo usa esta rutina para informar al FS de ciertos eventos:
19196     * tell_fs(CHDIR, slot, dir, 0)
19197     * tell_fs(EXEC, proc, 0, 0)
19198     * tell_fs(EXIT, proc, 0, 0)
19199     * tell_fs(FORK, parent, child, pid)
19200     * tell_fs(SETGID, proc, realgid, effgid)
19201     * tell_fs(SETSID, proc, 0, 0)
19202     * tell_fs(SETUID, proc, realuid, effuid)
19203     * tell_fs(SYNC, 0, 0, 0)
19204     * tell_fs(UNPAUSE, proc, signr, 0)
19205     */
19206
19207     message m;
19208
19209     m.m1_i1 = p1;
19210     m.m1_i2 = p2;
19211     m.m1_i3 = p3;
19212     _taskcall (FS_PROC_NR, what, &m);
19213     }
```

```

+++++
src/mm/putk.c
+++++

19300  I* Ocasionalmente, MM debe exhibir mensajes. Usa la rutina de biblioteca
19301  * estándar printk(). (El nombre "printf" realmente es una maGro definida como
19302  * "printk".) La exhibición se logra invocando la tarea TTY directamente,
19303  * no pasando por el FS.
19304  */
19305
19306      #include "mm.h"
19307      #include <minix/com.h>
19308
19309          #define BUF_SIZE      100      I* exhibir tamaño de buffer */
19310
19311          PRIVATE int buf_count; I* # caracteres en buffer */
19312          PRIVATE char print_buf[BUF_SIZE]; I* buffer de salidas aquí */
19313          PRIVATE message putch_msg; I* sirve para mensaje a tarea TTY */
19314
19315      _PROTOTYPE( FORWARD void flush, (void) );
19316
19317 /*=====*
19318  *                               putk                               *
19319 /*=====*/
19320      PUBLIC void putk(c)
19321      int C;
19322      {
19323          /* Acumular otro carácter. Si 0 o buffer lleno, exhibirlo. */
19324
19325      if (c == 0 || buf_count == BUF_SIZE) flush();
19326      if (c == '\n') putk('\r');
19327      if (c != 0) print_buf[buf_count++] = c;
19328      }
19331 /*=====*
19332  *                               flush                               *
19333 /*=====*/
19334      PRIVATE void flush()
19335      {
19336          I* Vaciar el buffer de exhibición invocando la tarea TTY. */
19337
19338      if (buf_count == 0) returnj ;
19339      putch_msg.m_type = DEV_WRITE;
19340      putch_msg.PROC_NR = 0;
19341      putch_msg.TTY_LINE = 0;
19342      putch_msg.ADDRESS = print_buf;
19343      putch_msg.COUNT = buf count;
19344      sendrec(TTY, &putch_msg);
19345      buf count = 0;
19346      }

```

```
+++++
                                     src/fs/fs.h
+++++
19400      /* Cabecera maestra del FS. Incluye algunos otros archivos
19401      * y define las constantes principales.
19402      */
19403          #define POSIX SOURCE      1      /* decir cabeceras q/incl. cosas POSIX */
19404          #define -MINIX-           1      /* decir cabeceras q/incl. cosas MINIX */
19405          #define =SYSTEM           1      /* decir cabeceras q/éste es el kernel */
19406
19407      /* Lo siguiente es tan básico que todos los archivos *.c lo incluyen autom. */
19408          #include <minix/config.h> /* DEBE ser primero */
19409          #include <ansi.h>        /* DEBE ser segundo */
19410          #include <sys/types.h>
19411          #include <minix/const.h>
19412          #include <minix/type.h>
19413
19414          #include <limits.h>
19415          #include <errno.h>
19416
19417          #include <minix/syslib.h>
19418
19419          #include "const.h"
19420          #include "type.h"
19421      #include "proto.h"
19422      #include "glo.h"

```

```
+++++
                                     src/fs/const.h
+++++
19500      /* Tamaños de tablas */
19501          #define V1_NR_DZONES      7      /* # núms. zona directos en nodo-i V1 */
19502          #define V1_NA_TZONES      9      /* # núms. zona totales en nodo-j V1 */
19503          #define V2_NR_DZONES      7      /* # núms. zona directos en nodo-i V2 */
19504          #define V2=NR=TZONES      10     /* # núms. zona totales en nodo-i V2 */
19505
19506          #define NR_FILPS           128   /* # ranuras en tabla filp */
19507          #define NR_INODES          64    /* # rano en tabla nodos-i "en núcleo" */
19508          #define NR_SUPERS          8     /* # ranuras en tabla superbloques */
19509          #define NR_LOCKS           8     /* # rano en tabla candados de archivo */
19510
19511      /* El tipo de sizeof puede ser (unsigned) long. Use la sigte macro p/tomar
19512      * tamaños de objetos pequeños pique no haya sorpresas como el paso
19513      * de constantes (small) long a rutinas que esperan un int.
19514      */
19515      #define usizeof(t) ((unsigned) sizeof(t))
19516
19517      /* Tipos de sistemas de archivos. */
19518          #define SUPER_MAGIC        0x137F /* núm mágico contenido en superbloque */
19519          #define SUPER=REV          0x7F13 /* # mágico si disco 68000 leído en PC o vv */
19520          #define SUPER_V2          0x2468 /* núm mágico p/sistemas de archivos V2 */
19521          #define SUPEA_V2_REV       0x6824 /* V2 mágico escr. en PC leído en 68K o vv */
19522
19523          #define V1                  1     /* núm versión de sistemas de arch. V1 */
19524          #define V2                  2     /* núm versión de sistemas de arch. V2 */

```

```

19525
19526 /* Constantes diversas. */
19527     #define SU_UID      ((uid_t) 0)      /* uid_t del superusuario */
19528     #define SYS_UID     ((uid_t) 0)      /* uid_t de procesos MM e INIT */
19529     #define SYS_GID     ((gid_t) 0)      /* gid_t de procesos MM e INIT */
19530     #define NORMAL      0                /* hace que get_block lea disco */
19531     #define NO_READ     1                /* impide que get_block lea disco */
19532     #define PREFETCH    2                /* dice a get_block no lea ni marque dev */
19533
19534     #define XPIPE       (-NR_TASKS-1)    /* usado en fp_task si susp en conducto */
19535     #define XOPEN       (-NR_TASKS-2)    /* usado en fp_task si susp en abrir */
19536     #define XLOCK       (-NR_TASKS-3)    /* usado en fp_task si susp en candado */
19537     #define XPOPEN      (-NR_TASKS-4)    /* usado en fp_task si susp en abrir cond */
19538
19539     #define NO_BIT      ((bit_t) 0)      /* devuelto por alloc_bit() si fracasó */
19540
19541     #define DUP_MASK    0100            /* máscara p/distinguir dup2 y dup */
19542
19543     #define LOOK_UP     0                /* dice search_dir busque cadena */
19544     #define ENTER       1                /* dice search_dir cree entrada dir */
19545     #define DELETE      2                /* dice search_dir borre entrada */
19546     #define IS_EMPTY    3                /* dice search_dir dev. OK o ENOTEMPTY */
19547
19548     #define CLEAN       0                /* copias en disco y memoria idénticas */
19549     #define DIRTY       1                /* copias en disco y memoria difieren */
19550     #define ATIME       002             /* 1 si campo atime debe actualizarse */
19551     #define CTIME       004             /* 1 si campo ctime debe actualizarse */
19552     #define MTIME       010             /* 1 si campo mtime debe actualizarse */
19553
19554     #define BYTE_SWAP   0                /* dice a conv2/conv4 q/interc. bytes */
19555     #define DONT_SWAP   1                /* dice a conv2/conv4 q/no interc. bytes */
19556
19557     #define END_OF_FILE  (-104)          /* eof detectado */
19558
19559     #define ROOT_INODE  1                /* # nodo-i p/directorio raíz */
19560     #define BOOT_BLOCK ((block_t) 0)    /* # bloque de bloque de
arranque */
19561     #define SUPER_BLOCK ((block_t) 1)    /* # bloque de
superbloque */
19562
19563     #define DIR_ENTRY_SIZE  sizeof(struct direct) /* # bytes/entr dir */
19564     #define NR_DIR_ENTRIES (BLOCK_SIZE/DIR_ENTRY_SIZE) /* # entr
dir/bloque */
19565     #define SUPER_SIZE     sizeof(struct super_block) /* tamaño superbloq
*/
19566     #define PIPE_SIZE      (V1_NR_DZONES*BLOCK_SIZE) /* tam
conducto, bytes */
19567     #define
BITMAP_CHUNKS (BLOCK_SIZE/sizeof(bitchunk_t)) /* # trozos mapa/blq */
19568
19569 /* Tamaños derivados correspondientes al sist. archivos V1. */
19570     #define V1_ZONE_NUM_SIZE  sizeof(zone1_t) /* # bytes en zona V1 */
19571     #define V1_INODE_SIZE     sizeof(d1_inode) /* bytes en nodo-i V1 */
19572     #define V1_INDIRECTS      (BLOCK_SIZE/V1_ZONE_NUM_SIZE) /* #
zonas/blq indir */
19573     #define V1_INODES_PER_BLOCK (BLOCK_SIZE/V1_INODE_SIZE) /* # nodos-i V1/blq
19574
19575 /* Tamaños derivados correspondientes al sist. archivos V2. */
19576     #define V2_ZONE_NUM_SIZE  sizeof(zone_t) /* # bytes en zona V2 */
19577     #define V2_INODE_SIZE     sizeof(d2_inode) /* bytes en nodo-i V2 */
19578     #define V2_INDIRECTS      (BLOCK_SIZE/V2_ZONE_NUM_SIZE) /* #
zonas/blq indir */
19579     #define V2_INODES_PER_BLOCK (BLOCK_SIZE/V2_INODE_SIZE) /* # nodos-i V2/blq */
19580
19581     #define printf printk

```



```

+++++
src/fs/type.h
+++++

```

```

19600  /* Declaración del nodo-i V1 en disco (no en el núcleo). */
19601                                     typedef struct { /* nodo-i de disco V1.x */
19602  mode_t d1 mode;                       /* tipo archivo, protección, etc. */
19603  uid_t d1 uid;                          /* id usuario de dueño del archivo */
19604  off_t d1-size;                         /* tamaño archivo actual en bytes */
19605  time_t d1_mtime;                      /* último cambio a datos del arch. */
19606  gid_t d1_Qid;                          /* número de grupo */
19607  nlink_t d1_nlinks;                    /* cuántos vinculas a este archivo */
19608  U16_t d1_Zone[V1_NR_TZONESJ]; /* núms blq p/directo, ind y dob ind */
19609  } d1_inode;
19610
19611  /* Declaración del nodo-i V2 en disco (no en el núcleo). */
19612                                     typedef struct { /* nodo-i de disco V2.x */
19613  mode_t d2 mode;                       /* tipo archivo, protección, etc. */
19614  u16_t d2_nlinks;                      /* cuántos vínculos al arch. iHACK! */
19615  uid_t d2-uid;                          /* id usuario de dueño del archivo */
19616  U16_t d2-gid;                          /* número de grupo. iHACK! */
19617  off_t d2-size;                         /* tamaño archivo actual en bytes */
19618  time_t d2_atime;                      /* último acceso a datos del arch. */
19619  time_t d2_mtime;                      /* último cambio a datos del arch. */
19620  time_t d2_ctime;                      /* último cambio a datos del nodo-i */
19621  zone_t d2=zone[V2_NR_TZONESJ]; /* núms blq p/directo, ind y dob ind */
19622  } d2_inode ;

```

```

+++++
src/fs/proto.h
+++++

```

```

19700  /* Prototipos de funciones. */
19701
19702  /* Structs empleados en prototipos deben declararse como tales primero. */ 19703 struct buf
19704  struct filp.
19705  struct inode;
19706  struct super_block;
19707
19708  /* cache.c */
19709 _PROTOTYPE( zone_t alloc_zone, (Dev_t dev, zone_t z) );
19710 flushall, (Dev_t dev) );
19711 free_zone, (Dev_t dev, zone_t numb) );
19712 _PROTOTYPE( struct buf *get_block, (Dev_t dev, block_t block,int only_search»);
19713 invalidate, (Dev_t device) );
19714 _PROTOTYPE( void put_block, (struct buf *bp, int block_type) );
19715 rw_block, (struct buf *bp, int rw_flag) );
19716 _PROTOTYPE( void rw_scattered, (Dev_t dev,
19717 struct buf **bufq, int bufqsize, int rw_flag) );
19718
19719 /* device.c */
19720 _PROTOTYPE( void call_task, (int task_nr, message *mess_ptr) );
19721 dev_opcl, (int task_nr, message *mess_ptr) );
19722 _PROTOTYPE( int dev_io, (int rw_flag, int nonblock, Dev_t dev,
19723 off_t pos, int bytes, int proc, char *buff) );
19724 (void) );

```

```
19725 _PROTOTYPE( void no_dev, (int task_nr, message *m_ptr) );
19726 _PROTOTYPE( void call_ctty, (int task_nr, message *mess_ptr) );
19727 _PROTOTYPE( void tty_open, (int task_nr, message *mess_ptr) );
19728 _PROTOTYPE( void cty_close, (int task_nr, message *mess_ptr) );
19729 _PROTOTYPE( void cty_open, (int task_nr, message *mess_ptr) );
19730 _PROTOTYPE( int do_setsid, (void) );
19731 #if ENABLE_NETWORKING
19732 _PROTOTYPE( void net_open, (int task_nr, message *mess_ptr) );
19733 #else
19734 #define net_open 0
19735 #endif
19736
19737 /* filedes.c */
19738 _PROTOTYPE( struct filp *find_filp, (struct inode *rip, Mode_t bits) );
19739 _PROTOTYPE( int get_fd, (int start, Mode_t bits, int *k, struct filp **fpt) );
19740 _PROTOTYPE( struct filp *get_filp, (int fild) );
19741
19742 /* inode.c */
19743 _PROTOTYPE( struct inode *alloc_inode, (Dev_t dev, Mode_t bits) );
19744 _PROTOTYPE( void dup_inode, (struct inode *ip) );
19745 _PROTOTYPE( void free_inode, (Dev_t dev, Ino_t numb) );
19746 _PROTOTYPE( struct inode *get_inode, (Dev_t dev, int numb) );
19747 _PROTOTYPE( void put_inode, (struct inode *rip) );
19748 _PROTOTYPE( void update_times, (struct inode *rip) );
19749 _PROTOTYPE( void rw_inode, (struct inode *rip, int rw_flag) );
19750 _PROTOTYPE( void wipe_inode, (struct inode *rip) );
19751
19752 /* link.c */
19753 _PROTOTYPE( int do_link, (void) );
19754 _PROTOTYPE( int do_unlink, (void) );
19755 _PROTOTYPE( int do_rename, (void) );
19756 _PROTOTYPE( void truncate, (struct inode *rip) );
19757
19758 /* lock.c */
19759 _PROTOTYPE( int lock_op, (struct filp *f, int req) );
19760 _PROTOTYPE( void lock_revive, (void) );
19761
19762 /* main.c */
19763 _PROTOTYPE( void main, (void) );
19764 _PROTOTYPE( void reply, (int whom, int result) );
19765
19766 /* misc.c */
19767 _PROTOTYPE( int do_dup, (void) );
19768 _PROTOTYPE( int do_exit, (void) );
19769 _PROTOTYPE( int do_fcntl, (void) );
19770 _PROTOTYPE( int do_fork, (void) );
19771 _PROTOTYPE( int do_exec, (void) );
19772 _PROTOTYPE( int do_revive, (void) );
19773 _PROTOTYPE( int do_set, (void) );
19774 _PROTOTYPE( int do_sync, (void) );
19775
19776 /* mount.c */
19777 _PROTOTYPE( int do_mount, (void) );
19778 _PROTOTYPE( int do_umount, (void) );
19779
19780 /* open.c */
19781 _PROTOTYPE( int do_clase, (void) );
19782 _PROTOTYPE( int do_creat, (void) );
19783 _PROTOTYPE( int do_lseek, (void) );
19784 _PROTOTYPE( int do_mknod, (void) );
```

```

19785 _PROTOTYPE( int do_mkdir, (void) );
19786 _PROTOTYPE( int do_open, (void) );
19787
19788 /* path.c */
19789 _PROTOTYPE( struct inode *advance, (struct inode *dirp, char string[NAME_MAX]));
19790 _PROTOTYPE( int search_dir, (struct inode *ldir_ptr,
19791 char string [NAME_MAX], ino_t *numb, int flag) );
19792 _PROTOTYPE( struct inode *eat_path, (char *path) );
19793 _PROTOTYPE( struct inode *last_dir, (char *path, char string [NAME_MAX]));
19794
19795 /* pipe.c */
19796 _PROTOTYPE( int do_pipe, (void) );
19797 _PROTOTYPE( int do_unpause, (void) );
19798 _PROTOTYPE( int pipe_check, (struct inode *rip, int rw_flag,
19799 int oflags, int bytes, off_t position, int *canwrite));
19800 _PROTOTYPE( void release, (struct inode *ip, int call_nr, int count) );
19801 _PROTOTYPE( void revive, (int proc_nr, int bytes) );
19802 _PROTOTYPE( void suspend, (int task) );
19803
19804 /* protect.c */
19805 _PROTOTYPE( int do_access, (void) );
19806 _PROTOTYPE( int do_chmod, (void) );
19807 _PROTOTYPE( int do_chown, (void) );
19808 _PROTOTYPE( int do_umask, (void) );
19809 _PROTOTYPE( int forbidden, (struct inode *rip, Mode_t access_desired) );
19810 _PROTOTYPE( int read_only, (struct inode *ip) );
19811
19812 /* putk.c */
19813 _PROTOTYPE( void putk, (int c) );
19814
19815 /* read.c */
19816 _PROTOTYPE( int do_read, (Yoid) );
19817 _PROTOTYPE( struct buf *rahead, (struct inode *rip, block_t baseblock,
19818 off_t position, unsigned bytes_ahead) );
19819 _PROTOTYPE( void read_ahead, (void) );
19820 _PROTOTYPE( block_t read_map, (struct inode *rip, off_t position) );
19821 _PROTOTYPE( int read_write, (int rw_flag) );
19822 _PROTOTYPE( zone_t rd_indir, (struct buf *bp, int index) );
19823
19824 /* stadir.c */
19825 _PROTOTYPE( int do_chdir, (void) );
19826 _PROTOTYPE( int do_chroot, (void) );
19827 _PROTOTYPE( int do_fstat, (void) );
19828 _PROTOTYPE( int do_stat, (void) );
19829
19830 /* super.c */
19831 _PROTOTYPE( bit_t alloc_bit, (struct super_block *sp, int map, bit_t origin));
19832 _PROTOTYPE( void free_bit, (struct super_block *sp, int map,
19833 bit_t bit_returned) );
19834 _PROTOTYPE( struct super_block *get_super, (Dev_t dev) );
19835 _PROTOTYPE( int mounted, (struct inode *rip) );
19836 _PROTOTYPE( int read_super, (struct super_block *sp) );
19837
19838 /* time.c */
19839 _PROTOTYPE( int do_stime, (void) );
19840 _PROTOTYPE( int do_time, (void) );
19841 _PROTOTYPE( int do_tims, (void) );
19842 _PROTOTYPE( int do_utime, (void) );
19843
19844 /* utility.c */

```

```

19845 _PROTOTYPE( time t clock time, (void) );
19846 _PROTOTYPE( unsigned conv2, (int narro, int w) );
19847 _PROTOTYPE( long conv4, (int narro, long x) );
19848 _PROTOTYPE( int fetch_name, (char *path, int len, int flag) );
19849 _PROTOTYPE( int no_sys, (void) );
19850 _PROTOTYPE( void panic, (char *format, int num) );
19851
19852 /* write.c */
19853 _PROTOTYPE( void clear_zone, (struct inode *rip, off_t pos, int flag) );
19854 _PROTOTYPE( int do_write, (void) );
19855 _PROTOTYPE( struct buf *new_block, (struct inode *rip, off_t position) );
19856 _PROTOTYPE( void zero_block, (struct buf *bp) );

```

+++++

```

src/fs/glo.h
+++++
19900 /* EXTERN debe ser extern excepto para el archivo de tabla */
19901 #ifdef TA8LE
19902 #undef EXTERN
19903 #define EXTERN
19904 #endif
19905
19906 /* Variables globales del sistema de archivos */
19907 EXTERN struct fproc *fp; /* apunto a struct fproc de invocador */
19908 EXTERN int super_user; /* 1 si invocador es superusuario */
19909 EXTERN int dont_reply; /* narro. 0; 1 p/inhibir respuesta */
19910 EXTERN int susp_count; /* # procesos suspendidos en conducto */
19911 EXTERN int nr_locks; /* # candados puestos actualmente */
19912 EXTERN int reviving; /* # procesos de conducto por revivir */
19913 EXTERN off_t rdahedpos; /* posición p/lectura anticipada */
19914 EXTERN struct inode *rdahed_inode; /* apunto a nodo-i p/lect. antic. */
19915
19916 /* Los parámetros de la llamada se guardan aquí. */
19917 EXTERN message m; /* el mensaje de entrada mismo */
19918 EXTERN message m1; /* mensaje de salida p/responder */
19919 EXTERN int who; /* núm proceso del invocador */
19920 EXTERN int fs_call; /* número de llamada al sistema */
19921 EXTERN char user_path[PATH_MAX]; /* espacio p/nombre ruta usuario */
19922
19923 /* Estas variables sirven para devolver resultados al invocador. */
19924 EXTERN int err_code; /* almac. temp. p/número de error */
19925 EXTERN int rdwt_err; /* situac. de últ. solíc. E/S disco */
19926
19927 /* Datos que requieren inicialización. */
19928 extern _PROTOTYPE (int (*call vector[]), (void) ); /* tabla llamo al sist. */
19929 extern int max_major; /* dispositivo princ. máx (+ 1) */
19930 extern char dot1[2]; /* dot1 (&dot1[0]) y dot2 (&dot2[0]) tienen signif. */
19931 extern char dot2[3]; /* especial p/search_dir: no verif. perro acceso. */

```

```

+++++
                                     src/fs/fproc.h
+++++

20000  /* Ésta es la información por proceso. Se reserva 1 ranura p/cada proceso potencial.
20001 * Así, NR_PROCS debe ser igual que en el kernel. No es posible
20002 * ni necesario saber aquí si una ranura está libre.
20003  */
20004
20005
20006     EXTERN struct fproc {
20007 mode_t fp_umask;                /* máscara fijada por la llamada umask */
20008 struct inode *fp_workdir;        /* apunt al nodo-i del dir de trabajo */
20009 struct inode *fp_rootdir;        /* apunt al dir raíz actual (ver chroot) */
20010 struct filp *fp_filp[OPEN_MAX]; /* tabla de descriptores de archivo */
20011 uid_t fp_realuid;                /* id de usuario real */
20012 uid_t fp_effuid;                 /* id de usuario efectivo */
20013 gid_t fp_realgid;               /* id de grupo real */
20014 gid_t fp_effgid;                /* id de grupo efectivo */
20015 dev_t fp_tty;                   /* princ/secund de tty controladora */
20016 int fp_fd;                       /* guardar fd si rd/wr no puede acabar */
20017 char *fp_buffer;                /* guardar buf si rd/wr no puede acabar */
20018 int fp_nbytes;                  /* guard bytes si rd/wr no puede acabar */
20019 int fp_cum_io_partial;          /* cta bytes si rd/wr no puede acabar */
20020 char fp_suspended;              /* 1 si proceso suspendido esperando */
20021 char fp_revived;                /* 1 si proceso se va a revivir */
20022 char fp_task;                   /* qué tarea espera el proceso */
20023 char fp_sesldr;                 /* 1 si proceso es jefe de sesión */
20024 pid_t fp_pid;                   /* id de proceso */
20025 long fp_cloexec;                /* mapa bits p/tabla 6-2 POSIX FD_CLOEXEC */
20026     } fproc[NR_PROCS];
20027
20028     /* Valores de campos. */
20029 #define NOT_SUSPENDED    0        /* proc no suspendido en conducto o tarea */
20030 #define SUSPENDED        1        /* proc suspendido en conducto o tarea */
20031 #define NOT_REVIVING     0        /* el proc no está siendo revivido */
20032 #define REVIVING         1        /* se revive al proceso en suspensión */

```

```

+++++
                                     src/fs/buf.h
+++++

20100  /* Caché de buffers (bloques). Para adquirir un bloque, una rutina invoca
20101 * get_block(), y le dice cuál quiere. El bloque ahora está "en uso"
20102 * Y se incrementa su campo 'b_count'. Los bloques no en uso
20103 * se encadenan en una lista LRÚ; 'front' apunta al bloque menos
20104 * recientemente usado, y 'rear', al más recientemente usado.
20105 * También se mantiene una cadena inversa usando el campo b_prev. El uso
20106 * para LRU se mide por el momento en que put_block() acaba. El 20.
20107 * parámetro de put_block() puede violar el orden LRU y poner un bloque
20108 * a la cabeza de la lista, si es probable que no se necesitará pronto.
20109 * Si un bloque se modifica, la rutina que lo haga debe asignar DIRTY a b_dirt
20110 * para que el bloque se reescriba finalmente en disco.
20111 */
20112
20113     #include <sys/dir.h>                /* necesita struct direct */
20114

```

```

20115     EXTERN struct buf {
20116 /* Porción de datos del buffer */
20117 union {
20118     char b--data[BLOCK_SIZE];          /* datos de usuario ordinarios */
20119     struct direct b_dir[NR DIR ENTRIES]; /* bloque de directorio */
20120     zone1_t b_v1_ind[V1 INDIRECTS];    /* bloque de indirección V1 */
20121     zone_t b==v2=ind[V2=INDIRECTS];    /* bloque de indirección V2 */
20122     d1_inode b--v1_ino[V1_INODES_PER_BLOCK]; /* bloque de nodos-i V1 */
20123     d2_inode b--v2_ino[V2_INODES_PER_BLOCK]; /* bloque de nodos-i V2 */
20124     bichunk_t b--bitmap[BITMAP_CHUNKS]; /* bloque de mapa de bits */
20125 } b;
20126
20127 /* Porción de cabecera del buffer. */
20128 struct buf *b_next;          /* para enlazar todos los bufs libres en cadena */
20129 struct buf *b_prev;         /* para enlazar los bufs libres en el otro sentido */
20130 struct buf *b_hash;         /* para enlazar bufs en cadenas de dispersión */
20131 block_t b_blocknr;         /* # de bloque de su dispositivo (secund) */
20132 dev_t b_dev;               /* disp. princ I secund donde reside el bloque */
20133 char b_dirt;               /* CLEAN o DIRTY */
20134 char b_count;              /* núm de usuarios de este buffer */
20135 } buf[NR_BUFS];
20136
20137 /* Un bloque está libre si b_dev == NO_DEV. */
20138
20139 #define NIL_BUF ((struct buf *) 0)          /* indica ausencia de buffer */
20140
20141 /* Estas defs. permiten usar bp->b_data en vez de bp->b--data */
20142 #define b_data b.b--data
20143 #define b_dir b.b--dir
20144 #define b_v1_ind b.b--v1_ind 20145 #define b_v2_ind b.b--v2_ind
20146 #define b_v1_ino b.b--v1_ino 20147 #define b_v2_ino b.b--v2_ino 20148 #define b_bitmap b.b--bitmap 20149
20150     EXTERN struct buf *buf_hash[NR_BUF_HASH]; /* tabla de disp. de buffers */
20151
20152     EXTERN struct buf *front;          /* apunta a bloque libre menos rec. usado */
20153     EXTERN struct buf *rear;          /* apunta a bloque libre más rec. usado */
20154     EXTERN int bufs_in_use;          /* # buffers en uso (no en lista libre) */
20155
20156 /* Al liberarse un bloque, el tipo de uso se pasa a put_block(). */
20157 #define WRITE_IMMED 0100 /* bloque debe escribir en disco ahora */
20158 #define ONE_SHOT 0200 /* 1 si bloque no se necesitará pronto */
20159
20160 #define INODE_BLOCK (0 + MAYBE_WRITE_IMMED) /* bloque nodos-i */
20161 #define DIRECTORY_BLOCK (1 + MAYBE_WRITE_IMMED) /* bloque directorio */
20162 #define INDIRECT_BLOCK (2 + MAYBE_WRITE_IMMED) /* bloque apuntadores */
20163 #define MAP_BLOCK (3 + MAYBE_WRITE_IMMED) /* mapa de bits */
20164 #define ZUPER_BLOCK (4 + WRITE_IMMED + ONE_SHOT) /* superbloque */
20165 #define FULL_DATA_BLOCK 5 /* datos, tot usado */
20166 #define PARTIAL_DATA_BLOCK 6 /* datos, parc usado */
20167
20168 #define HASH_MASK (NR_BUF_HASH - 1) /* máscara p/dispersar núms bloque */

```

```
+++++
                                     src/fs/dev.h
+++++

20200      /* Tabla de dispositivos. Se indiza por núm de disp. principal.
20201 * Vincula los núms de disp principal y las rutinas que los procesan.
20202      */
20203
20204      typedef _PROTOTYPE (void (*dmap_t), (int task, message *m_ptr) );
20205
20206      extern struct dmap {
20207 dmap_t dmap_open;
20208 dmap_t dmap_rwj;
20209 dmap_t dmap_closej;
20210 int dmap_task;
20211      } dmap[ ];
20212
```

```
+++++
                                     src/fs/file.h
+++++

20300      /* Tabla filp. Es un intermediario entre los descriptores de archivo y
20301 * los nodos-i. Está libre una ranura si filp_count == 0.
20302      */
20303
20304      EXTERN struct filp {
20305 mode_t filp_modej;          /* bits RW, dicen cómo se abrió archivo */
20306 int filp_flags;           /* banderas de open y fcntl */
20307 int filp_count;          /* cuántos descrip. arch. comparten ranura*/
20308 struct inode *filp_inoj;  /* apuntador al nodo-i */
20309 off_t filp_pos;          /* posición en el archivo */
20310      } filp[NR_FILPS];
20311
20312      #define FILP_CLOSED      0      /* filp_mode: disp asociado cerrado */
20313
20314 #define NIL_FILP (struct filp *) 0      /* indica ausencia de ranura filp */
```

```
+++++
                                     src/fs/lock.h
+++++

20400      /* Tabla de candados de archivo. Como la filp, apunta a la tabla de
20401 * nodos-i, pero en este caso para poner candados asesores.
20402      */
20403      EXTERN struct file_lock {
20404 short lock_type;          /* F_RDLOCK o F_WRLOCK; 0 = ranura libre */
20405 pid_t lock_pid;          /* pid del proceso q/tiene el candado */
20406 struct inode *lock_inode; /* apuntador al nodo-i asegurado */
20407 off_t lock_first;        /* distancia del 1er byte asegurado */
20408 off_t lock_last;         /* distancia del últ byte asegurado */
20409      } file_lock[NR_LOCKS];
```

```

+++++
                                     src/fs/inode.h
+++++

20500      /* Tabla de nodos-i. Contiene nodos-i en uso. En algunos casos
20501 * se abrieron con una llamada open() o creat(); en otros, el FS mismo
20502 * necesita el nodo-i por una razón u otra, como buscar un nombre de ruta
20503 * en un directorio. La primera parte del struct contiene campos que están
20504 * en el disco; la segunda contiene campos que no lo están. La parte
20505 * de nodos-i de disco también se declara en "type.h" como
20506 * 'd1_inode' para los sistemas de archivo V1 y como 'd2_inode'
20507 * para los sistemas V2.
20508      */
20509
20510      EXTERN struct inode {
20511 mode_t i_mode;          /* tipo archivo, protección, etc. */
20512 nlink_t i_nlinks;      /* cuántos vínculos a este archivo */
20513 uid_t i_uid;           /* id de usuario del dueño del archivo */
20514 gid_t i_gid;           /* número de grupo */
20515 off_t i_size;          /* tamaño del archivo actual en bytes */
20516 time_t i_atime;        /* tiempo de último acceso (sólo V2) */
20517 time_t i_mtime;        /* última modif. de datos del archivo */
20518 time_t i_ctime;        /* últ. modif. del nodo.i mismo (sólo V2) */
20519 zone_t i_zone[V2_NR_TZONES]; /* núms zona p/directo, ind y doble ind */
20520
20521 /* Las siguientes cosas no están presentes en el disco. */
20522 dev_t i_dev;            /* dispositivo en el que está el nodo-i */
20523 ino_t i_num;            /* núm nodo-i en su dispositivo (secund) */
20524 int i_count;           /* # veces se usó nodo.ij 0 = ranura libre */
20525 int i_ndzones;         /* # zonas directas (Vx_NR_DZONES) */
20526 int i_nindirs;         /* # zonas indirec. por bloque de indirec. */
20527 struct super_block *i_sp; /* apunt a superbloque del disp del nodo-i */
20528 char i_dirt;           /* CLEAN o DIRTY */
20529 char i_pipe;           /* igual a I_PIPE si es conducto */
20530 char i_mount;          /* bit encendido si montado en archivo */
20531 char i_seek;           /* encend. por LSEEK, apago por READ/WRITE */
20532 char i_update;        /* bits ATIME, CTIME y MTIME están aquí */
20533     } inode[NR_INODES];
20534
20535
20536 #define NIL_INODE (struct inode *) 0      /* indica ausencia ranura nodo-i */
20537
20538 /* Valores de campo. CLEAN y DIRTY se definen en "const.h" */
20539 #define NO_PIPE      0      /* i_pipe = NO_PIPE si nodo-i no es conducto */
20540 #define I_PIPE       1      /* i_pipe = I_PIPE si nodo-i es conducto */
20541 #define NO_MOUNT     0      /* i_mount = NO_MOUNT si no montado en arch */
20542 #define I_MOUNT      1      /* i_mount = I_MOUNT si montado en archivo */
20543 #define NO_SEEK      0      /* i_seek = NO_SEEK si últ op no fue SEEK */
20544 #define ISEEK        1      /* i_seek = I_SEEK si últ op fue SEEK */

```



```
+++++
                                     src/fs/param.h
+++++
20600  /* Estos nombres son sinónimos de las variables del mensaje de entrada. */
20601  #define acc_time      m.m2_l1
20602  #define .addr         m.m1_i3
20603  #define buffer        m.m1_p1
20604  #define child         m.m1_i2
20605  #define co_mode       m.m1_i1
20606  #define eff_grp_id    m.m1_i3
20607  #define eff_user_id   m.m1_i3
20608  #define erki          m.m1_p1
20609  #define fd             m.m1_i1
20610  #define fd2          m.m1_i2
20611  #define ioflags      m.m1_i3
20612  #define group        m.m1_i3
20613  #define real_grp_id  m.m1_i2
20614  #define ls_fd        m.m2_i1
20615  #define mk_mode       m.m1_i2
20616  #define mode          m.m3_i2
20617  #define c_mode        m.m1_i3
20618  #define c_name        m.m1_p1
20619  #define name          m.m3_p1
20620  #define name1         m.m1_p1
20621  #define name2         m.m1_p2
20622  #define name_length   m.m3_i1
20623  #define name1_length  m.m1_i1
20624  #define name2_length  m.m1_i2
20625  #define nbytes        m.m1_i2
20626  #define offset       m.m2_l1
20627  #define owner        m.m1_i2
20628  #define parent       m.m1_i1
20629  #define pathname      m.m3_ca1
20630  #define pid           m.m1_i3
20631  #define pro           m.m1_i1
20632  #define rd_only       m.m1_i3
20633  #define real_user_id  m.m1_i2
20634  #define request       m.m1_i2
20635  #define sig           m.m1_i2
20636  #define slot1         m.m1_i1
20637  #define tp            m.m2_l1
20638  #define utime_actime   m.m2_l1
20639  #define utime_modtime  m.m2_l2
20640  #define utime_file     m.m2_p1
20641  #define utime_length   m.m2_i1
20642  #define whence        m.m2_i2
20643
20644  /* Estos nombres son sinónimos de las variables del mensaje de salida. */
20645          #define reply_type      m1.m_type
20646          #define reply_l1         m1.m2_l1
20647          #define reply_i1        m1.m1_i1
20648          #define reply_i2        m1.m1_i2
20649          #define reply_t1        m1.m4_l1
20650          #define reply_t2        m1.m4_l2
20651          #define reply_t3        m1.m4_l3
20652          #define reply_t4        m1.m4_l4
20653          #define reply_t5        m1.m4_l5
```

```

+++++
                                     src/fs/super.h
+++++

20700      /* Tabla de superbloques. El FS raíz y todos los FS montados tienen una entrada
20701 * aquí. La entrada contiene info. sobre los tamaños de los mapas de bits y nodos-i.
20702 * El campo s_ninodes da el núm de nodos-i disponibles para archivos y directorios,
20703 * incl. el directorio raíz. El nodo-i 0 está en el disco pero no se usa. Por tanto,
20704 * s_ninodes = 4 implica que se usarán 5 bits en el mapa de bits: el bit 0,
20705 * que siempre es 1 y no se usa, y los bits 1-4 para archivos y directorios.
20706 * La organización del disco es:
20707      *
20708 *      Elemento          # bloques
20709 *      bloque arranque      1
20710 *      superbloque          1
20711 *      mapa de nodos-i      s_imap_blocks
20712 *      mapa de zonas        s_zmap_blocks
20713 *      nodos-i              (s_ninodes + 'nodos.i/bloque' - 1)/'nodos-i/bloque'
20714 *      no se usa            lo necesario p/llevar la zona actual
20715 *      zonas de datos       (s_zones - s_firstdatazone) <<          s_log_zone_size
20716      *
20717 * Una ranura de superbloque está libre si s_dev == NO_DEV.
20718 */
20719
20720
20721  EXTERN struct super_block {
20722  ino_t s_ninodes;          /* # nodos-i usables en disp. secundo */
20723  zone1_t s_nzones;        /* tamaño total disp, incl mapas-bits, etc */
20724  short s_imap_blocks;     /* # bloques ocup por mapa-bits de nodos-i */
20725  short s_zmap_blocks;     /* # bloques ocup por mapa-bits de zonas */
20726  zone1_t s_firstdatazone; /* número de primera zona de datos */
20727  short s_log_zone_size;   /* 10g2 de bloques/zona */
20728  off_t s_max_size;        /* tam máx de archivo en este dispos. */
20729  short s_magic;           /* núm mágico p/reconocer superbloques */
20730  short s_pad;             /* evitar relleno dependiente del compilo */
20731  zone_t s_zones;         /* # zonas (sustituye a s_nzones en V2) */
20732
20733  /* Lo siguiente sólo se usa si el superbloque está en memoria. */
20734  struct inode *s_isup;    /* nodo-i p/dir raíz de FS montado */
20735  struct inode *s_imount;  /* nodo-i en el que está montado */
20736  unsigned s_inodes_per_block; /* precalculado a partir del núm mág */
20737  dev_t s_dev;            /* ¿de quién es este superbloque? */
20738  int s_rd_only;          /* 1 si sist. arch. montado sólo lectura */
20739  int s_native;           /* 1 si sist. arch. sin interc. bytes */
20740  int s_version;          /* versión sist. arch. 0 = mágico mal */
20741  int s_ndzones;          /* núm. zonas directas en nodo-i */
20742  int s_nindir;           /* # zonas indir. por bloque indir. */
20743  bit_t s_isearch;        /* los nodos-i bajo este # bit se usan */
20744  bit_t s_zsearch;        /* zonas bajo este # bit se usan */
20745  } super_block[NR_SUPERS];
20746
20747  #define NIL_SUPER (struct super_block *) 0
20748      #define IMAP 0          /* operando en mapa-bits de nodos-i */
20749      #define ZMAP 1         /* operando en mapa-bits de zonas */

```

```
+++++
                                     src/fs/table.h
+++++
20800  /* Este archivo contiene la tabla que transforma los números
20801  * de llamadas al sistema en las rutinas que las ejecutan.
20802  */
20803
20804      #define _TABLE
20805
20806  #include "fs.h"
20807  #include <minix/callnr.h>
20808  #include <minix/com.h>
20809      #include      "buf.h"
20810      #include "dev.h"
20811      #include "file.h"
20812      #include "fproc.h"
20813      #include "inode.h"
20814      #include "lock.h"
20815      #include "super.h"
20816
20817  PUBLIC _PROTOTYPE (int (*call_vector[NCALLS]), (void) ) = {
20818  no_sys,          /*      0 = no se usa */
20819  do_exit,        /*      1 = exit      */
20820  do_fork,        /*      2 = fork      */
20821  do_read,        /*      3 = read      */
20822  do_write,       /*      4 = write     */
20823  do_open,        /*      5 = open      */
20824  do_clase,       /*      6 = Glose     */
20825  no_sys,         /*      7 = wait      */
20826  do_creat,       /*      8 = creat     */
20827  do_link,        /*      9 = link      */
20828  do_unlink,      /*     10 = unlink   */
20829  no_sys,         /*     11 = waitpid */
20830  do_chdir,       /*     12 = chdir    */
20831  do_time,        /*     13 = time     */
20832  do_mknod,       /*     14 = mknod    */
20833  do_chmod,       /*     15 = chmod    */
20834  do_chown,       /*     16 = chown    */
20835  no_sys,         /*     17 = break    */
20836  do_stat,        /*     18 = stat     */
20837  do_lseek,       /*     19 = lseek    */
20838  no_sys,         /*     20 = getpid   */
20839  do_mount,       /*     21 = mount    */
20840  do_umount,      /*     22 = umount   */
20841  do_set,         /*     23 = setuid   */
20842  no_sys,         /*     24 = getuid   */
20843  do_stime,       /*     25 = stime    */
20844  no_sys,         /*     26 = ptrace   */
20845  no_sys,         /*     27 = alarm    */
20846  do_fstat,       /*     28 = fstat    */
20847  no_sys,         /*     29 = pause    */
20848  do_utime,       /*     30 = utime    */
20849  no_sys,         /*     31 = (stty)   */
20850  no_sys,         /*     32 = (gtty)   */
20851  do_access,      /*     33 = access   */
20852  no_sys,         /*     34 ~ (nice)   */
20853  no_sys,         /*     35 = (ftime) */
20854  do_sync,        /*     36 = sync     */

```

```

20855     no_sys,          /* 37 = kill      */
20856     do_rename,      /* 38 = rename    */
20857     do_mkdir,       /* 39 = mkdir     */
20858     do_unlink,      /* 40 = rmdir    */
20859     do_dup,         /* 41 = dup       */
20860     do_pipe,        /* 42 = pipe      */
20861     do_tiros,       /* 43 = times     */
20862     no_sys,         /* 44 = (prof)    */
20863     no_sys,         /* 45 = no se usa */
20864     do_set,         /* 46 = setgid    */
20865     no_sys,         /* 47 = getgid    */
20866     no_sys,         /* 48 = (signal) */
20867     no_sys,         /* 49 = no se usa */
20868     no_sys,         /* 50 = no se usa */
20869     no_sys,         /* 51 = (acct)    */
20870     no_sys,         /* 52 = (phys)    */
20871     no_sys,         /* 53 = (lock)    */
20872     do_ioctl,      /* 54 = ioctl     */
20873     do_fcntl,      /* 55 = fcntl     */
20874     no_sys,         /* 56 = (mpx)     */
20875     no_sys,         /* 57 = no se usa */
20876     no_sys,         /* 58 = no se usa */
20877     do_exec,        /* 59 = execve    */
20878     do_umask,       /* 60 = umask     */
20879     do_chroot,     /* 61 = chroot    */
20880     do_setsid,     /* 62 = setsid    */
20881     no_sys,         /* 63 = getpgrp */
20882
20883     no_sys,         /* 64 = KSIG: señales con origen en el kernel */
20884     do_unpause,     /* 65 = UNPAUSE */
20885     no_sys,         /* 66 = no se usa */
20886     do_revive,      /* 67 = REVIVE    */
20887     no_sys,         /* 68 = TASK_REPLY */
20888     no_sys,         /* 69 = no se usa */
20889     no_sys,         /* 70 = no se usa */
20890     no_sys,         /* 71 = SIGACTION */
20891     no_sys,         /* 72 = SIGSUSPEND */
20892     no_sys,         /* 73 = SIGPENDING */
20893     no_sys,         /* 74 = SIGPROCMAK */
20894     no_sys,         /* 75 = SIGRETURN */
20895     no_sys,         /* 76 = REBOOT */
20896     };
20897
20898
20899     /* Algunos dispositivos podrían estar o no en la siguiente tabla. */
20900 #define DT(enable, open, rw, Glose, task) \
20901     { (enable ? (open) : no_dev), (enable ? (rw) : no_dev), \
20902     (enable ? (close) : no_dev), (enable ? (task) : 0) },
20903
20904     /* El orden de las entradas aquí determina la correspondencia entre
20905     * núms de disp. principal y tareas. La 1a. entrada (disp princ 0) no se usa.
20906     * La siguiente es disp princ 1, etc. Los dispositivos por caracteres
20907     * Y por bloques pueden entremezclarse al azar. Si se cambia el orden,
20908     * los dispositivos en <include/minix/boot.h> deben cambiarse a los nuevos valores.
20909     * Los núms de disp principal empleados en /dev NO son iguales a los usados
20910     * en el kernel (definidos en <include/minix/com.h>). Si /dev/mem
20911     * se cambia de 1, deberá modificarse NULL_MAJOR en
20912     * <include/minix/com.h>.
20913     */
20914 PUBLIC struct dmap dmap[ ] = {

```

```

20915 /* ?      Abrir      Leer/Escribir Cerrar # Tarea  Disp.      Archivo
20916 -      -----
20917 DT(1, no_dev, ~o_dev, no_dev, 0) /* 0 = no se usa */
20918 DT(1, dev_opcl, call_task, dev_opcl, MEM) /* 1 = /dev/mem */
20919 DT(1, dev_opcl, call_task, dev_opcl, FLOPPY) /* 2 = /dev/fd0 */
20920 DT(ENABLE_WINI,
20921 dev_opcl, call_task, dev_opcl, WINCHESTER) /* 3 = /dev/hd0 */
20922 DT(1, tty_open, call_task, dev_opcl, TTY) /* 4 = /dev/tty00 */
20923 DT(1, ctty_open, call_ctty, ctty_close, TTY) /* 5 = /dev/tty */
20924 DT(1, dev_opcl, call_task, dev_opcl, PRINTER) /* 6 = /dev/lp */
20925
20926 #if (MACHINE == IBM_PC)
20927 DT(ENABLE_NETWORKING,
20928 net_open, call_task, dev_opcl, INET_PROC_NR) /* 7 = /dev/ip */
20929 DT(ENABLE_COROM,
20930 dev_opcl, call_task, dev_opcl, COROM) /* 8 = /dev/cd0 */
20931 DT(0, 0, 0, 0, 0) /* 9 = no se usa */
20932 DT(ENABLE_SCSI,
20933 dev_opcl, call_task, dev_opcl, SCSI) /* 10 = /dev/sd0 */
20934 DT(0, 0, 0, 0, 0) /* 11 = no se usa */
20935 DT(0, 0, 0, 0, 0) /* 12 = no se usa */
20936 DT(ENABLE_AUOIO,
20937 dev_opcl, call_task, dev_opcl, AUOIO) /* 13 = /dev/audio */
20938 DT(ENABLE_AUOIO,
20939 dev_opcl, call_task, dev_opcl, MIXER) /* 14 = /dev/mixer */
20940 #endif /* IBM_PC */
20941
20942 #if (MACHINE == ATARI)
20943 DT(ENABLE_SCSI,
20944 dev_opcl, call_task, dev_opcl, SCSI) /* 7 = /dev/hdscsi0 */
20945 #endif
20946 };
20947
20948 PUBLIC int max_major = sizeof(dmap)/sizeof(struct dmap);

```

```

+++++
src/fs/cache.c
+++++

```

```

21000 /* El sistema de archivos mantiene un caché de buffers para reducir el número
21001 * de accesos a disco necesarios. Cada vez que se lee o escribe en el disco,
21002 * se verifica si el bloque está en el caché. Este archivo administra
21003 * el caché.
21004 *
21005 * Los puntos de entrada a este archivo son:
21006 * get_block: obtener del caché un bloque para leer o escribir
21007 * put_block: devolver bloque antes solicitado con get_block
21008 * alloc_zone: asignar nueva zona (p/aumentar longitud de un archivo)
21009 * free_zone: liberar una zona (cuando se quita un archivo)
21010 * rw_block: leer o escribir un bloque en el disco mismo
21011 * invalidate: quitar todos los bloques de caché de algún dispositivo
21012 */
21013
21014 #include "fs.h"
21015 #include <minix/com.h>
21016 #include <minix/boot.h>
21017 #include "buf.h"
21018 #include "file.h"
21019 #include "fproc.h"

```

```

21020 #include "super.h"
21021
21022 FORWARD_PROTOTYPE( void rm_lru, (struct buf *bp) );
21023
21024 /*=====
21025 *                                     get_block                                     *
21026 *=====*/
21027 PUBLIC struct buf *get_block(dev, block, only_search)
21028     register dev_t dev;      /* ¿en qué disp está el bloque? */
21029     register block_t block;  /* ¿qué bloque se desea? */
21030     int only_search; /* si NO_READ, no leer; si no, normal */
21031 {
21032     /* Ver si el bloque solicitado está en el caché. Si está, devolver
21033 * apuntador a él. Si no, expulsar otro bloque y traerlo (a menos que
21034 * 'only_search' = 1). Todos los bloques del caché no en uso se enlazan
21035 * en una cadena; 'front' apunta al menos recientemente usado, y 'rear',
21036 * al más recient. usado. Si 'only_search' es 1, el bloque solicitado
21037 * se sobrescribirá por completo, así que sólo es necesario ver si está
21038 * en el caché; si no está, cualquier buffer libre sirve. No es necesario
21039 * leer el bloque de disco. Si 'only_search' es PREFETCH, no hay que
21040 * leer el bloque del disco, y el dispositivo no debe marcarse
21041 * en el bloque; así, los invocadores pueden saber si el bloque
21042 * devuelto es válido.
21043 * Además de la cadena LRU, hay una de dispersión para enlazar bloques cuyos
21044 * números terminan con la misma cadena de bits, para búsqueda rápida.
21045 */
21046
21047 int b;
21048 register struct buf *bp, *prev_ptr;
21049
21050 /* Buscar (dev, block) en la cadena de dispersión. Do_read() puede
21051 * usar get_block(NO_DEV ...) para obtener un bloque sin nombre
21052 * que llenar con ceros si alguien quiere leer de un agujero en un archivo,
21053 * en cuyo caso esta búsqueda se pasa por alto.
21054 */
21055 if (dev != NO_DEV) {
21056     b = (int) block & HASH_MASK;
21057     bp = buf_hash[b];
21058     while (bp != NIL_BUF) {
21059         if (bp->b_blocknr == block && bp->b_dev == dev) {
21060             /* Se encontró el bloque requerido. */
21061             if (bp->b_count == 0) rm_lru(bp);
21062             bp->b_count++; /* registrar bloque en uso */
21063             return(bp);
21064         } else {
21065             /* Este bloque no es el que buscábamos. */
21066             bp = bp->b_hash; /* pasar a sigio bloque de cadena disp. */
21067         }
21068     }
21069 }
21070
21071 /* Bloque deseado no está en cadena disponible. Tomar el más viejo ('front'). */
21072 if ((bp = front) == NIL_BUF) panic("all buffers in use", NR_BUFS);
21073 rm_lru(bp);
21074
21075 /* Quitar el bloque recién tomado de su cadena de dispersión. */
21076 b = (int) bp->b_blocknr & HASH_MASK;
21077 prev_ptr = buf_hash[b];
21078 if (prev_ptr == bp) {
21079     buf_hash[b] = bp->b_hash;

```

```

21080 } else {
21081     /* El bloque tomado no está al frente de su cadena de disp. */
21082     while (prev_ptr->b_hash != NIL_BUF)
21083         if (prev_ptr->b_hash == bp) {
21084             prev_ptr->b_hash = bp->b_hash;    /* lo hallamos */
21085             break;
21086         } else {
21087             prev_ptr = prev_ptr->b_hash;    /* seguir buscando */
21088         }
21089     }
21090
21091     /* Si el bloque tomado está sucio, limpiarlo escribiéndolo en disco.
21092     * Evitar histéresis desalojando los demás bloques sucios p/mismo dispositivo.
21093     */
21094     if (bp->b_dev != NO_DEV) {
21095         if (bp->b_dirt == DIRTV) flushall(bp->b_dev);
21096     }
21097
21098     /* Llenar paráms. del bloque y agregarlo a la cadena de disp. en su lugar. */
21099     bp->b_dev = dev;    /* llenar núm. dispositivo */
21100     bp->b_blocknr = block;    /* llenar número de bloque */
21101     bp->b_count++;    /* registrar bloque en uso */
21102     b = (int) bp->b_blocknr & HASH_MASK;
21103     bp->b_hash = buf_hash[b];
21104     buf_hash[b] = bp;    /* agregar a lista de dispersión */
21105
21106     /* Obtener bloque solicitado a menos que sea búsqueda o preobtención. */
21107     if (dev != NO_DEV) {
21108         if (only_search == PREFETCH) bp->b_dev = NO_DEV;
21109         else
21110             if (only_search == NORMAL) rw_block(bp, READING);
21111     }
21112     return(bp);    /* devolver bloque recién adquirido */
21113 }

```

```

21116 /*=====
21117 *                               put_block                               *
21118 *=====*/
21119     PUBLIC void put_block(bp, block_type)
21120         register struct buf *bp;    /* apunto a bloque por liberar */
21121         int block_type;    /* INODE_BLOCK, DIRECTORV_BLOCK o lo que sea */
21122     {
21123         /* Devolver un bloque a la lista de disponibles. Dependiendo de 'block_type',
21124         * puede ponerse al frente o al final de la cadena LRU. Los bloques
21125         * que se espera necesitar pronto (p.ej. bloques de datos parco llenos)
21126         * van al finalj los que quizá no se necesitarán pronto otra vez
21127         * (p.ej. bloques de datos llenos) van al frente. Los bloques cuya pérdida
21128         * podría dañar la integridad del sist. de archivos (p.ej. bloques de nodos-i)
21129         * se escriben de inmediato en disco si están sucios.
21130         */
21131
21132         if (bp == NIL_BUF) return;    /* es más fácil verif. aquí qlen invocador */
21133
21134         bp->b_count--;    /* ahora hay un uso menos */
21135         if (bp->b_count != 0) returnj /* bloque aún en uso */
21136
21137         bufs_in_use--;    /* un buffer de bloque menos en uso */
21138
21139     /* Devolver este bloque a la cadena LRU. Si el bit ONE_SHOT está encendido

```

```

21140 * en 'block_type', no es probable que el bloque se necesite pronto;
21141 * se coloca al frente de la cadena LRU para que sea el primero que se tome
21142 * cuando se necesite posteriormente un buffer libre.
21143 */
21144 if (block_type & ONE_SHOT) {
21145     /* El bloque quizá no se necesitará pronto. Ponerlo al frente de la cadena;
21146     * será el próximo bloque expulsado del caché.
21147     */
21148     bp->b_prev = NIL_BUF;
21149     bp->b_next = front;
21150     if (front == NIL_BUF)
21151         rear = bp;          /* la cadena LRU estaba vacía */
21152     else
21153         front->b_prev = bp;
21154     front = bp;
21155 } else {
21156     /* El bloque quizá se necesitará pronto. Ponerlo al final de la cadena;
21157     * tardará mucho en ser expulsado del caché.
21158     */
21159     bp->b_prev = rear;
21160     bp->b_next = NIL_BUF;
21161     if (rear == NIL_BUF)
21162         front = bp;
21163     else
21164         rear->b_next = bp;
21165     rear = bp;
21166 }
21167
21168 /* Algunos bloques son tan importantes (nodos-i, de indirección) que deben
21169 * escribirse en disco de inmediato para evitar arruinar el sistema
21170 * de archivos en caso de una caída.
21171 */
21172 if ((block_type & WRITE_IMMED) && bp->b_dirt==DIRTY && bp->b_dev != NO_DEV)
21173     rw_block(bp, WRITING);
21174 }

21177 /*=====
21178 *                               alloc_zone                               *
21179 *=====*/
21180     PUBLIC zone_t alloc_zone(dev, z)
21181         dev_t dev;          /* disp. donde se quería zona */
21182         zone_t z;          /* tratar de asignar nva. zona cerca */
21183     {
21184     /* Asignar nueva zona en el dispositivo indicado y devolver su número. */
21185
21186     int major, minor;
21187     bit_t b, bit;
21188     struct super_block *sp;
21189
21190     /* La rutina alloc_bit() devuelve 1 para la zona más baja posible,
21191     * que corresp. a sp->s_firstdatazone. Para convertir un valor
21192     * entre el número de bit, 'b', empleado por alloc_bit() y el número
21193     * de zona, 'z', almacenado en el nodo-i, use la fórmula:
21194     *     z = b + sp->s_firstdatazone - 1
21195     * Alloc_bit() nunca devuelve 0, que se usa para NO_BIT (fracaso).
21196     */
21197     sp = get_super(dev);          /* encontrar superbloque p/este disp. */
21198
21199     /* Si z es 0, saltar parte inicial del mapa que se sabe está toda en uso. */

```



```

21200 if (z == sp->s_firstdatazone) {
21201     bit = sp->s_zsearch;
21202 } else {
21203     bit = (bit_t) z -(sp->s_firstdatazone -1);
21204 }
21205 b = alloc_bit(sp, LMAP, bit);
21206 if (b T= NO_BIT) {
21207     err_code = ENOSPC;
21208     major = (int) (sp->s_dev >>     MAJOR) & BYTE;
21209     minor = (int) (sp->s_dev >>     MINOR) & BYTE;
21210     printf("No space on %sdevice %d/%d\n",
21211           sp->s_dev == ROOT_DEV ? "root ' : '", major, minor);
21212     return(NO_ZONE);
21213 }
21214 if (z == sp->s_firstdatazone) sp->s_zsearch = b;           /* p/sigte. vez */
21215 return(sp->s_firstdatazone -1 + (zone_t) b);
21216 }

21219 /*=====
21220 *                               free_zone                               *
21221 *=====*/
21222     PUBLIC void free_zone(dev, numb)
21223                               dev_t dev; /* disp. donde está zona */
21224                               zone_t numb; /* zona por devolver */
21225     {
21226         /* Devolver una zona. */
21227
21228     register struct super_block *sp;
21229     bit_t bit;
21230
21231     /* Encontrar el superbloque apropiado y devolver bit. */
21232     sp = get_super(dev);
21233     if (numb < sp->s_firstdatazone || numb >= sp->s_zones) return;
21234     bit = (bit_t) (numb -(sp->s_firstdatazone -1));
21235     free_bit(sp, ZMAP, bit);
21236     if (bit < sp->s_zsearch) sp->s_zsearch = bit;
21237     }

21240 /*=====
21241 *                               rw_block                               *
21242 *=====*/
21243     PUBLIC void rw_block(bp, rw_flag)
21244                               register struct buf *bp; /* apuntador a buffer */
21245                               int rw_flag; /* READING o WRITING */
21246     {
21247         /* Leer o escribir un bloque de disco. Ésta es la única rutina en la que se invoca
21248 * realmente E/S de disco. Si hay error, se exhibe un mensaje aquí, pero el error
21249 * no se informa al invocador. Si el error ocurrió mientras se purgaba un bloque
21250 * del caché, no queda claro qué podría hacer el invocador en todo caso.
21251 */
21252
21253     int r, op;
21254     off_t pos;
21255     dev_t dev;
21256
21257     if ((dev = bp->b_dev) != NO_DEV) {
21258         pos = (off_t) bp->b_blocknr * BLOCK_SIZE;
21259         op = (rw_flag == READING ? DEV_READ : DEV_WRITE);

```

```

21260         r = dev_io(op, FALSE, dev, pos, BLOCK_SIZE, FS_PROC_NR, bp->b_data);
21261     if (r != BLOCK_SIZE) {
21262         if (r >= 0) r = END_OF_FILE;
21263         if (r != END_OF_FILE)
21264             printf("Unrecoverable disk error on device %d/%d, block %ld\n",
21265                 (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
21266         bp->b_dev = NO_DEV;      /* invalidar bloque */
21267
21268         /* Informar errores de lectura a las partes interesadas. */
21269         if (rw_flag == READING) rdwt_err = r;
21270     }
21271 }
21272
21273 bp->b_dirt = CLEAN;
21274 }

21277 /*=====
21278 *                               invalidate                               *
21279 *=====*/
21280 PUBLIC void invalidate(device)
21281 dev_t device;          /* disp. cuyos bloques se purgarán */
21282 {
21283     /* Quitar del caché todos los bloques pertenecientes a algún dispositivo. */
21284
21285     register struct buf *bp;
21286
21287     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
21288         if (bp->b_dev == device) bp->b_dev = NO_DEV;
21289 }

21292 /*=====
21293 *                               flushall                               *
21294 *=====*/
21295     PUBLIC void flushall(dev)
21296 dev_t dev;            /* dispositivo p/desalojar */
21297 {
21298 /* Desalojar todos los bloques sucios p/un dispositivo. */
21299
21300     register struct buf *bp;
21301     static struct buf *dirty[NR_BUFS];          /* static; no está en pila */
21302     int ndirty;
21303
21304     for (bp = &buf[0], ndirty = 0; bp < &buf[NR_BUFS]; bp++)
21305         if (bp->b_dirt == DIRTY && bp->b_dev == dev) dirty[ndirty++] = bp;
21306     rw_scattered(dev, dirty, ndirty, WRITING);
21307 }

21310 /*=====
21311 *                               rw_scattered                               *
21312 *=====*/
21313     PUBLIC void rw_scattered(dev, bufq, bufqsize, rw_flag)
21314 dev_t dev;          /* núm disp principal/secundario */
21315 struct buf **bufq;  /* apuntador a matriz de buffers */
21316 int bufqsize;      /* número de buffers */
21317 int rw_flag;       /* READING o WRITING */
21318 {
21319     /* Leer o escribir datos dispersos de un dispositivo. */

```

```

21320
21321 register struct buf *bp;
21322 int gap;
21323 register int i;
21324 register struct iorequest_s *iop;
21325 static struct iorequest_s iovec[NR_IOREQS];          /* static: no en la pila*/
21326 int j;
21327
21328 /* Ordenar (shellsort) buffers por b_blocknr. */
21329 gap = 1;
21330 do
21331     gap = 3 * gap + 1;
21332 while (gap <= bufqsize);
21333 while (gap != 1) {
21334     gap /= 3;
21335     for (j = gap; j < bufqsize; j++) {
21336         for (i = j - gap;
21337              i >= 0 && bufq[i] ->b_blocknr > bufq[i + gap] ->b_blocknr;
21338              i -= gap) {
21339             bp = bufq[i];
21340             bufq[i] = bufq[i + gap];
21341             bufq[i + gap] = bp;
21342         }
21343     }
21344 }
21345
21346 /* Preparar vector de E/S y efectuar E/S. El resultado de dev_io se desecha
21347  * porque todos los resultados se devuelven en el vector. Si dev_io fracasa
21348  * totalmente, el vector no cambia y los resultados se toman como errores.
21349  */
21350 while (bufqsize > 0) {
21351     for (j = 0, iop = iovec; j < NR_IOREQS && j < bufqsize; j++, iop++) {
21352         bp = bufq[j];
21353         iop->io_position = (off_t) bp->b_blocknr * BLOCK_SIZE;
21354         iop->io_buf = bp->b_data;
21355         iop->io_nbytes = BLOCK_SIZE;
21356         iop->io_request = rw_flag == WRITING ;
21357         DEV_WRITE : DEV_READ | OPTIONAL_IO;
21358     }
21359     (void) dev_io(SCATTERED_IO, 0, dev, (off_t) 0, j, FS_PROC_NR,
21360                 (char *) iovec);
21361
21362     /* Cosechar resultados. Dejar los de lectura pique rw_block() se queje. */
21363     for (i = 0, iop = iovec; i < j; i++, iop++) {
21364         bp = bufq[i];
21365         if (rw_flag == READING) {
21366             if (iop->io_nbytes == 0)
21367                 bp->b_dev = dev;          /* validar bloque */
21368             put_block(bp, PARTIAL_DATA_BLOCK);
21369         } else {
21370             if (iop->io_nbytes != 0) {
21371                 printf("Unrecoverable write error on device %d/%d, block %ld\n",
21372                        (dev»MAJOR)&BYTE, (dev»MINOR)&BYTE, bp->b_blocknr);
21373                 bp->b_dev = NO_DEV;      /* invalidar bloque */
21374             }
21375             bp->b_dirt = CLEAN;
21376         }
21377     }
21378     bufq += j;
21379     bufqsize -= j;

```

```

21380     }
21381 }
21384 /*=====
21385 *                               rm_lru                               *
21386 *=====*/
21387 PRIVATE void rm_lru(bp)
21388 struct buf *bp;
21389 {
21390     /* Quitar un bloque de su cadena LRU. */
21391     struct buf *next_ptr, *prev_ptr;
21392     bufs_in_use++;
21393     next_ptr = bp->b_next;      /* sucesor en cadena LRU */
21394     prev_ptr = bp->b_prev;      /* predecesor en cadena LRU */
21395     if (prev_ptr != NIL_BUF)
21396         prev_ptr->b_next = next_ptr;
21397     else
21398         front = next_ptr;      /* este bloque estaba al frente */
21399     if (next_ptr != NIL_BUF)
21400         next_ptr->b_prev = prev_ptr;
21401     else
21402         rear = prev_ptr;      /* este bloque estaba al final */
21403 }
+++++
src/fs/inode.c
+++++
21500     /* Este archivo administra la tabla de nodos-i. Hay procedimientos
21501     * para asignar, quitar, adquirir, borrar y liberar nodos-i, Y leerlos
21502     * de y escribirlos en disco.
21503     *
21504     * Los puntos de entrada a este archivo son
21505     * get_inode:      buscar nodo-i dado en tabla; si no está, leerlo
21506     * put_inode:      indicar que un nodo-i ya no se necesita en memoria
21507     * alloc_inode:    asignar un nuevo nodo-i no utilizado
21508     * wipe_inode:     borrar algo campos de un nodo-i recién asignado
21509     * free_inode:     marcar nodo-i como disponible para un nuevo archivo
21510     * update_times:  actualizar atime, ctime y mtime
21511     * rw_inode:      leer bloque disco y extraer nodo-i, o escribirlo
21512     * old_icopy:     copiar a/de struct inode en núcleo y nodo-i disco (v1.x)
21513     * new_icopy:     copiar a/de struct inode en núcleo y nodo-i disco (v2.x)
21514     * dup_inode:     indicar que alguien más usa la entrada tabla nodos-i
21515     */
21516
21517     #include "fs.h"
21518     #include <minix/boot.h>
21519     #include "buf.h"
21520     #include "file.h"
21521     #include "fproc.h"
21522     #include "inode.h"
21523     #include "super.h"
21524

```

```

21525 FORWARD _PROTOTYPE( void old_icopy, (struct inode *rip, d1_inode *dip,
21526                                     int direction, int norm)) ;
21527 FORWARD _PROTOTYPE( void new_icopy, (struct inode *rip, d2_inode *dip,
21528                                     int direction, int norm));
21529
21530
21531 /*=====
21532 *                                     get_inode                                     *
21533 *=====*/
21534     PUBLIC struct inode *get_inOde(dev, numb)
21535     dev_t dev;                               /* disp. donde reside nodo-i */
21536     int numb;                               /* # nodo-i (ANSI: nunca unshort) */
21537     {
21538     /* Encontrar una ranura en la tabla de nodos-i, cargar el nodo-i especificado en ella
21539     * Y devolver un apuntador a la ranura. Si 'dev' == NO_DEV, s610 devolver una ranura libre.
21540     */
21541
21542     register struct inode *rip, *xp;
21543
21544     /* Buscar tanto (dev, numb) como 1 ranura libre en tabla de nodos-i. */
21545     xp = NIL_INODE;
21546     for (rip = &inode[0] j rip < &inode[NR_INODES] j rip++) {
21547         if (rip->i_count > 0) { /* s610 buscar en ranuras usadas */
21548             if (rip->i_dev == dev && rip->i_num == numb) {
21549                 /* Éste es el nodo que buscamos. */
21550                 rip->i_count++;
21551                 return(rip);          /* hallamos (dev, numb) */
21552             }
21553         } else {
21554             xp = rip;          /* recordar esta rano libre después */
21555         }
21556     }
21557
21558     /* Nodo-i deseado no está en uso. ¿Encontramos una ranura libre? */
21559     if (xp == NIL_INODE) {      /* tabla nodos-i totalmente llena */
21560         err_code = ENFILE;
21561         return(NIL_INODE);
21562     }
21563
21564     /* Encontramos ranura de nodo-i libre. Cargar nodo-i en ella. */
21565     xp->i_dev = dev;
21566     xp->i_num = numb;
21567     xp->i_count = 1;
21568     if (dev != NO_DEV) rw_inode(xp, READING);          /* obtener nodo de disco */
21569     xp->i_update = 0;          /* todos los tiempos inic. actualizados */
21570
21571     return(xp);
21572 }
21573 /*=====
21574 *                                     put_inode                                     *
21575 *=====*/
21576     PUBLIC void put_inode(rip)
21577     register struct inode *rip;          /* apunto a nodo-i por liberar */
21578     {
21579     /* El invocador ya no usa este nodo-i. Si nadie más lo usa
21580     * escribirlo en disco de inmediato. Si no tiene vínculos, truncarlo
21581     * Y devolverlo a la reserva de nodos-i disponibles.
21582     */
21583     }
21584

```

```

21585
21586 if (rip == NIL_INODE) return; /* más fácil verif. aquí q/en invocador */
21587 if (--rip->i_count == 0) { /* i_count == 0 implica q/nadie lo usa */
21588     if ((rip->i_nlinks & BYTE) == 0) {
21589         /* i_nlinks == 0 implica liberar el nodo-i. */
21590         truncate(rip); /* devolver todos los bloques de disco */
21591         rip->i_mode = I_NOT_ALLOC; /* borrar campo I_TYPE */
21592         rip->i_dirt = DIRTY;
21593         free_inode(rip->i_dev, rip->i_num);
21594     } else {
21595         if (rip->i_pipe == I_PIPE) truncate(rip);
21596     }
21597     rip->i_pipe = NO_PIPE; /* siempre debe borrarse */
21598     if (rip->i_dirt == DIRTY) rw_inode(rip, WRITING);
21599 }
21600 }
21601 /*=====
21602 *
21603 * alloc_inode *
21604 *=====
21605     PUBLIC struct inode *alloc_inode(dev, bits)
21606         dev_t dev; /* disp. en el cual asignar nodo-i */
21607     mode_t bits; /* modo del nodo-i */
21608     {
21609     /* Asignar nodo-i libre en 'dev' y devolver apuntador a él. */
21610
21611     register struct inode *ripj
21612     register struct super_block *sp;
21613     int majar, minar, inumb;
21614     bit_t bj
21615
21616     sp = get_super(dev); /* obtener apuntador a superbloque */
21617     if (sp->s_rd_only) { /* imposible asignar nodo en disp. sólo lectura. */
21618         err_code = EROFS;
21619         return(NIL_INODE);
21620     }
21621
21622     /* Adquirir un nodo-i del mapa de bits. */
21623     b = alloc_bit(sp, IMAP, sp->s_isearch);
21624     if (b == NO_BIT) {
21625         err_code = ENFILE;
21626         majar = (int) (sp->s_dev » MAJOR) & BYTE;
21627         minar = (int) (sp->s_dev » MINOR) & BYTE;
21628         printf("Out of i-nodes on %sdevice %d/%d\n",
21629             sp->s_dev == ROOT_DEV ? "root " : "", majar, minar);
21630         return(NIL_INODE);
21631     }
21632     sp->s_isearch = b; /* sigte vez comenzar aquí */
21633     inumb = (int) bj /* no pasar unshort como parám. */
21634
21635     /* Tratar de adquirir una ranura de la tabla de nodos-i. */
21636     if ((rip = get_inode(NO_DEV, inumb) == NIL_INODE) {
21637         /* No hay ranuras disponibles. Liberar el nodo recién asignado. */
21638         free_bit(sp, IMAP, b);
21639     } else {
21640         /* Ranura disponible. Colocar en ella el nodo-i recién asignado. */
21641         rip->i_mode = bitsj /* establecer bits RWX */
21642         rip->i_nlinks = (nlink_t) 0; /* inic. sin vínculos */
21643         rip->i_uid = fp->fp_effuid; /* uid del arch es el del dueño */
21644         rip->i_gid = fp->fp_effgid; /* ídem id de grupo */

```

```

21645     rip->i_dev = dev;                               /* marcar en cuál disp está */
21646     rip->i_ndzones = sp->s_ndzones; /* núm zonas directas */
21647     rip->i_nindirs = sp->s_nindirs; /* núm zonas indirectas/bloque*/
21648     rip->i_sp = sp;                                  /* apuntador a superbloque */
21649
21650     /* Los campos aún no borrados se borran en Wipe_inOde().
21651     * Se pusieron ahí porque truncate() necesita borrar los mismos
21652     * campos si el archivo está abierto mientras se trunca. Se ahorra
21653     * espacio si no se repite el código dos veces.
21654     */
21655     wipe_inode(rip);
21656     }
21657
21658     return(rip);
21659 }
21661 /*=====
21662 *                               wipe_inode                               *
21663 *=====
21664 PUBLIC void wipe_inode(rip)
21665 register struct inode *rip' /* el nodo-i por borrar */
21666 {
21667     /* Borrar unos campos del nodo-i. Esta función se invoca desde alloc inode()
21668     * cuando se va a asignar un nuevo nodo-i, y de truncate() cuando se-va
21669     * a truncar un nodo-i existente.
21670     */
21671
21672     register int i;
21673
21674     rip->i_size = 0;
21675     rip->i_update = ATIME I CTIME I MTIME;           /* actualizar tiempos después */
21676     rip->i_dirt = DIRTY;
21677     for (i = 0; i < V2_NR_TIONES; i++) rip->i_zone[i] = NO_IONE;
21678     }
21681 /*=====
21682 *                               free inode                               *
21683 *=====
21684 PUBLIC void free_inode(dev, inumb)
21685 dev_t dev; /* en cuál dispositivo está el nodo-i */
21686 ino_t inumb; /* núm del nodo-i por liberar */
21687 {
21688     /* Devolver un nodo-i a la reserva de nodos no asignados. */
21689
21690     register struct super_block *sp;
21691     bit_t b;
21692
21693     /* Encontrar el superbloque apropiado. */
21694     sp = get_super(dev);
21695     if (inumb <= 0 || inumb > sp->s_ninodes) return;
21696     b = inumb;
21697     free_bit(sp, IMAP, b);
21698     if (b < sp->s_isearCh) sp->s_isearCh = b;
21699 }
21701 /*=====
21702 *                               update_times                             *
21703 *=====
21704 PUBLIC void update_times(rip)

```

```

21705             register struct inode *rip; /* apunt a nodo-i por leer/escribir */
21706         {
21707             /* El estándar exige que varias llamadas al sist. actualicen atime, ctime
21708 * o mtime. Ya que para ello hay que enviar un mensaje a la tarea del reloj,
21709 * lo cual es caro, los tiempos se marcan para actualización encendiendo bits
21710 * en i_update. Al terminar una stat, fstat o sync, o al liberarse un nodo-i,
21711 * se puede invocar update_times() para asentar realmente los tiempos.
21712 */
21713
21714     time_t cur_time;
21715     struct super_block *sp;
21716
21717     sp = rip->i_sp;             /* obtener apunt a superbloque. */
21718     if (sp->s_rd_only) return; /* no actualizo en FS sólo de lectura */
21719
21720     cur_time = clock_time();
21721     if (rip->i_update & ATIME) rip->i_atime = cur_time;
21722     if (rip->i_update & CTIME) rip->i_ctime = cur_time;
21723     if (rip->i_update & MTIME) rip->i_mtime = cur_time;
21724     rip->i_update = 0;         /* ya están actualizados */
21725 }
21726
21727 /*=====
21728 *                               rw_inode                               *
21729 *=====*/
21730
21731     PUBLIC void rw_inode(rip, rw_flag)
21732             register struct inode *rip; /* apunto a nodo-i por leer/escribir */
21733             int rw_flag; /* READING o WRITING */
21734     {
21735         /* Una entrada de tabla nodos-i debe copiarse en o de disco. */
21736
21737         register struct buf *bp;
21738         register struct super_block *sp;
21739         d1_inode *dip;
21740         d2_inode *dip2;
21741         block_t b, offset;
21742
21743         /* Obtener bloque donde reside el nodo-i. */
21744         sp = get_super(rip->i_dev); /* obt. apunto a superbloque */
21745         rip->i_sp = sp;             /* nodo-i debe contener apunt a superbl */
21746         offset = sp->s_imap_blocks + sp->s_zmap_blocks + 2;
21747         b = (block_t) (rip->i_num - 1) / sp->s_inodes_per_block + offset;
21748         bp = get_block(rip->i_dev, b, NORMAL);
21749         dip = bp->b_v1_ino + (rip->i_num - 1) % V1_INODES_PER_BLOCK;
21750         dip2 = bp->b_v2_ino + (rip->i_num - 1) % V2_INODES_PER_BLOCK;
21751
21752         /* Efectuar lectura o escritura. */
21753         if (rw_flag == WRITING) {
21754             if (rip->i_update) update_times(rip); /* hay que acto tiempos */
21755             if (sp->s_rd_only == FALSE) bp->b_dirt = DIRTY;
21756         }
21757
21758         /* Copiar el nodo-i del bloque de disco a la tabla en núcleo o viceversa.
21759 * Si el cuarto parám es FALSE, los bytes se intercambian.
21760 */
21761         if (sp->s_version == V1)
21762             old_icopy(rip, dip, rw_flag, sp->s_native);
21763         else
21764             new_icopy(rip, dip2, rw_flag, sp->s_native);

```



```

21765
21766     put_block(bp, INODE_BLOCK);
21767     rip->i_dirt = CLEAN;
21768 }
21771 /*=====
21772 *                               old_icopy                               *
21773 *=====*/
21774     PRIVATE void old_icopy(rip, dip, direction, norm)
21775     register struct inode *rip; /* apunt a struct inode en núcleo */
21776     register d1_inode *dip;     /* apunt a struct inode d1_inode */
21777     int direction;              /* * READING (de disco) o WRITING (en disco) */
21778     int norm;                   /* * TRUE = no intercamb. bytes; FALSE = sí */
21779
21780     {
21781     /* El disco IBM V1.x, el disco 68000 V1.x y el disco V2 (mismo para IBM
21782     * Y 68000) tienen diferentes organizo de nodos-i. Al leerse o escribirse
21783     * un nodo esta rutina hace las conversiones para que la info de la tabla de nodos-i
21784     * sea independiente de la estructura del disco del que provino el nodo-i.
21785     * La rutina old_icopy copia a y de discos V1.
21786     */
21787
21788     int i;
21789
21790     if (direction == READING) {
21791     /* Copiar nodo-i V1.x en tabla en núcleo, interc. bytes si es preciso. */
21792     rip->i_mode = conv2(norm, (int) dip->d1_mOde);
21793     rip->i_uid = conv2(norm, (int) dip->d1_uid);
21794     rip->i_size = COnv4(norm, dip->d1_Size);
21795     rip->i_mtime = COnv4(norm, dip->d1_mtime);
21796     rip->i_atime = rj-p->i_mtime;
21797     rip->i_ctime = rip->i_mtime;
21798     rip->i_nlinks = (nlink_t) dip->d1_nlinkS; /* 1 char */
21799     rip->i_gid = (gid_t) dip->d1_gid; /* 1 char */
21800     rip->i_ndzones = = V1_NR_DIONES;
21801     rip->i_nindirs = V1_INDIRECTS;
21802     for (i = 0; i < V1_NR_TIONES; i++)
21803     rip->i_zone[i] = conv2(norm, (int) dip->d1_zone[i])
21804     } else {
21805     /* Se copia nodo-i V1.x a disco de la tabla en núcleo. */
21806     dip->d1_mode = conv2(norm, (int) rip->i_mode);
21807     dip->d1_uid = conv2(norm, (int) rip->i_uid);
21808     dip->d1_size = cOnv4(norm, rip->i_size);
21809     dip->d1_mtime = COnv4(norm, rip->i_mtime);
21810     dip->d1_nlinks = (nlink_t) rip->i_nlinks; /* 1 char */
21811     dip->d1_gid = (gid_t) rip->i_gid; /* 1 char */
21812     for (i = 0; i < V1_NR_TIONES; i++)
21813     dip->d1_zone[i] = conv2inorm, (int) rip->i_zone[i];
21814     }
21815     }
21818 /*=====
21819 *                               new_icopy                               *
21820 *=====*/
21821     PRIVATE void new_icopy(rip, dip, direction, norm)
21822     register struct inode *ripj /* apunt a struct inode en núcleo */
21823     register d2_inode *dip; /* apunt a struct d2_inode */
21824     int direction; /* * READING (de disco) o WRITING (en disco) */

```

```

21825     int norm;                /* TRUE = no intercamb. bytes; FALSE = sí */
21826
21827     {
21828     /* Mismo que old_icopy, pero a/de disco con organizo V2. */
21829
21830     int i;
21831
21832     if (direction == READING) {
21833     /* Copiar nodo-i V2.x en tabla en núcleo, interc. bytes si es preciso. */
21834     rip->i_mode      = conv2(nOrm,dip->d2_mode);
21835     rip->i_uid       = conv2(nOrm,dip->d2_uid );
21836     rip->i_nlinks    = conv2(norm, (int) dip->d2_nlinks);
21837     rip->i_gid       = conv2(norm, (int) dip->d2_gid );
21838     rip->i_size      = conv4(nOrm,dip->d2_size);
21839     rip->i_atime     = conv4(norm,dip->d2_atime);
21840     rip->i_ctime     = conv4(nOrm,dip->d2_ctime);
21841     rip->i_mtime     = conv4(nOrm,dip->d2_mtime);
21842     rip->i_ndzones   = V2_NR_DIONES;
21843     rip->i_nindirs   = V2_INDIRECTS;
21844     for (i = 0; i < V2_NR_TIONES; i++)
21845         rip->i_zone[i] = conv4(norm, (long) dip->d2_zone[i]);
21846     } else {
21847     /* Se copia nodo-i V2.x a disco de la tabla en núcleo. */
21848     dip->d2_mode     = conv2(nOrm,rip->i_mode);
21849     dip->d2_uid      = conv2(nOrm,rip->i_uid );
21850     dip->d2_nlinks   = conv2(norm,rip->i_nlinks);
21851     dip->d2_gid      = conv2(norm,rip->i_gid );
21852     dip->d2_size     = conv4(norm,rip->i_size);
21853     dip->d2_atime    = conv4(nOrm,rip->i_atime);
21854     dip->d2_ctime    = conv4(norm,rip->i_ctime);
21855     dip->d2_mtime    = conv4(nOrm,rip->i_mtime);
21856     for (i = 0; i < V2_NR_TIONES; i++)
21857         dip->d2_zone[i] = conv4(norm, (long) rip->i_zone[i]);
21858     }
21859     }
21862 /*=====
21863 *                                     dup_inode                                     *
21864 /*=====
21865     PUBLIC void dup_inode(ip)
21866     struct inode *ip;          /* El nodo-i por duplicar. */
21867     {
21868     /* Esta rutina es una forma simplificada de get_inode() para
21869     * el caso en que ya se conoce el apuntador a nodo-i.
21870     */
21871
21872     ip->i_count++;
21873     }

```

```
+++++
src/fs/super.c
+++++
21900      /* Este archivo administra la tabla de superbloques y las estruc.
21901 * de datos relacionadas: los mapas de bits que indican cuáles zonas
21902 * Y nadas están asignados/libres. Cuando se necesita un nuevo nodo-i
21903 * o zona, se busca una entrada libre en el mapa de bits apropiado.
21904 *
21905 * Los puntos de entrada a este archivo son
21906 * alloc_bit:    alguien quiere asignar 1 zona o nodo-i; hallarlo
21907 * free_bit:     indicar que 1 zona o nodo-i está disponible p/asig.
21908 * get_super:    buscar un dispositivo en tabla 'superblock'
21909 * mounted:      indica si nodo arch está en sist arch montado (o ROOT)
21910 * read_super:  leer un superbloque
21911 */
21912
21913 #include "fs.h"
21914 #include <string.h>
21915 #include <minix/boot.h>
21916 #include "buf.h"
21917 #include "inode.h"
21918 #include "super.h"
21919
21920 #define BITCHUNK_BITS      (sizeof(bitchunk_t) * CHAR_BIT)
21921 #define BITS_PER_BLOCK    (BITMAP_CHUNKS * BITCHUNK_BITS)
21922
21923 /*=====
21924 *                alloc_bit                *
21925 *=====*/
21926 PUBLIC bit_t alloc_bit(sp, map, origin)
21927     struct super_block *sp;    /* sist arch del cual asignar */
21928     int map;                   /* IMAP (mapa nadas) o ZMAP (zonas) */
21929     bit_t origin;             /* # bit donde iniciar búsqueda */
21930 {
21931     /* Asignar un bit del mapa de bits y devolver su número. */
21932
21933     block_t start_block;      /* primer bloque de bits */
21934     bit_t map_bits;           /* ¿cuántos bits en el mapa? */
21935     unsigned bit_blocks;      /* ¿cuántos bloques en el mapa? */
21936     unsigned block, word, bcount, wcount;
21937     struct buf *bp;
21938     bitchunk_t *wptr, *wlim, kj;
21939     bit_t i, b;
21940
21941     if (sp->s_rd_only)
21942         panic("can't allocate bit on read-only filesystem.", NO_NUM);
21943
21944     if (map == IMAP) {
21945         start_block = SUPER_BLOCK + 1;
21946         map_bits = sp->s_ninodes + 1j;
21947         bit_blocks = sp->s_imap_blocksj;
21948     } else {
21949         start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
21950         map_bits = sp->s_zones -(sp->s_firstdatazone -1);
21951         bit_blocks = sp->s_zmap_blocks;
21952     }
21953
21954     /* Averiguar dónde iniciar búsqueda de bit (depende de 'origin'). */
```

```

21955 if (origin >= map_bitS) origin = 0;          /* para robustez */
21956
21957 /* Localizar el punto de inicio. */
21958 block = origin / BITS_PER_BLOCK;
21959 word = (origin % BITS_PER_BLOCK) / BITCHUNK_BITS;
21960
21961 /* Iterar con todos los bloques + 1, porque comenzamos en medio. */
21962 bcount = bit_blocks + 1;
21963 do {
21964     bp = get_block(sp->s_dev, start_block + block, NORMAL);
21965     wlim = &bp->b_bitmap[BITMAP_CHUNKS];
21966
21967     /* Iterar con las palabras del bloque. */
21968     for (wptr = &bp->b_bitmap[word]; wptr < wlim; wptr++) {
21969
21970         /* ¿Contiene esta palabra un bit libre? */
21971         if (*wptr == (bitchunk_t) -0) continue;
21972
21973         /* Encontrar Y asignar el bit libre. */
21974         k = conv2(Sp->s_native, (int) *wptr);
21975         for (i = 0; (k & (1 << i) != 0); ++i) {}
21976
21977         /* Núm de bit desde el principio del mapa de bits. */
21978         b = <<bit_t> block * BITS_PER_BLOCK
21979           + (wptr - &bp->b_bitmap[0]) * BITCHUNK_BITS
21980           + i;
21981
21982         /* No asignar bits más allá del final del mapa. */
21983         if (b >= map_bits) break;
21984
21985         /* Asignar Y devolver número de bit. */
21986         k |= 1 << i;
21987         *wptr = conv2(sp->s_native, (int) k);
21988         bp->b_dirt = DIRTY;
21989         put_block(bp, MAP_BLOCK);
21990         return(b);
21991     }
21992     put_block(bp, MAP_BLOCK);
21993     if (++block >= bit_blockS) block = 0; /* últ. bloque, ir al inicio */
21994     word = 0;
21995 } while (--bcount > 0);
21996 return(NO_BIT);          /* no pudo asignarse un bit */
21997 }
22000 /*=====*/
22001 *                free_bit                *
22002 /*=====*/
22003 PUBLIC void free_bit(sp, map, bit_returned)
22004 struct super_block *sp;          /* sist arch sobre el cual operar */
22005 int map;                          /* IMAP (mapa nodos) o ZMAP (zonas) */
22006 bit_t bit_returned;             /* # bit que insertar en el mapa */
22007 {
22008     /* Regresar una zona o nodo-i apagando su bit en el mapa. */
22009
22010     unsigned block, word, bit;
22011     struct buf *bp;
22012     bitchunk_t k, mask;
22013     block_t start_block;
22014

```

```

22015 if (sp->s_rd_only)
22016     panic("can't free bit on read-only fileys.", NO_NUM);
22017
22018 if (map == IMAP) {
22019     start_block = SUPER_BLOCK + 1;
22020 } else {
22021     start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
22022 }
22023 block = bit_returned / BITS_PER_BLOCK;
22024 word = (bit_returned % BITS_PER_BLOCK) / BITCHUNK_BITS;
22025 bit = bit_returned % BITCHUNK_BITS;
22026 mask = 1 << bit;
22027
22028 bp = get_block(sp->s_dev, start_block + block, NORMAL);
22029
22030 k = conv2(sp->s_native, (int) bp->b_bitmap[word]);
22031 if (!(k & mask) {
22032     panic(map == IMAP ? "tried to free unused inode" :
22033         "tried to free unused block", NO_NUM);
22034 }
22035
22036 k &= ~mask;
22037 bp->b_bitmap[word] = conv2(sp->s_native, (int) k);
22038 b->b_dirt = DIRTY;
22039
22040 put_block(bp, MAP_BLOCK);
22041 }
22044
/*=====
22045 *                                     get_super                                     *
22046*=====
==*/
22047     PUBLIC struct super_block *get_super(dev)
22048         dev_t dev; /* núm disp cuyo superbloque se busca */
22049     {
22050 /* Buscar este dispositivo en tabla de superbloques. Debe estar ahí. */
22051
22052     register struct super_block *sp;
22053
22054     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
22055         if (sp->s_dev == dev) return(sp);
22056
22057 /* Falló la búsqueda. Algo anda mal. */
22058     panic("can't find superbloc for device (in decimal)", (int) dev);
22059
22060     return(NIL_SUPER); /* p/mantener tranquilos al compilador y lint */
22061 }
22064
/*=====
22065 *                                     mounted                                     *
22066*=====
==*/
22067     PUBLIC int mounted(rip)
22068         register struct inode *rip; /* apuntador a nodo-i */
22069     {
22070 /* Informar si el nodo-i dado está en un sist arch montado (o ROOT). */
22071
22072     register struct super_block *sp;
22073     register dev_t dev;
22074

```

```

22075 dev = (dev_t) rip->i_zone[0];
22076 if (dev == ROOT_DEV) return(TRUE);          /* nodo está en sist arch raíz */
22077
22078 for (sp = &super_block[0]j sp < &super_block[NR_SUPERS] j sp++)
22079     if (sp->s_dev == dev) return(TRUE);
22080
22081 return(FALSE);
22082 }
22085
/*=====
22086 *                               read_super                               *
22087
*=====*/
22088     PUBLIC int read_super(sp)
22089     register struct super_block *sp; /* apuntador a superbloque */
22090     {
22091     /* Leer un superbloque. */
22092
22093     register struct buf *bp;
22094     dev_t dev;
22095     int magic;
22096     int version, native;
22097
22098     dev = sp->s_dev;          /* guardar disp (copia lo sobrescribirá) */
22099     bp = get_block(sp->s_dev, SUPER_BLOCK, NORMAL);
22100     memcpy( (char *) sp, bp->b_data, (size_t) SUPER_SIZE);
22101     put_block(bp, ZUPER_BLOCK)j
22102     sp->s_dev = NO_DEV;      /* restaurar después */
22103     magic = sp->s_magic;    /* determina tipo de sistema arch */
22104
22105     /* Obtener versión y tipo del sistema de archivos. */
22106     if (magic == SUPER_MAGIC 1 j magic == conv2(BYTE_SWAP, SUPER_MAGIC) {
22107         version = V1;j
22108         native = (magic == SUPER_MAGIC);
22109     } else if (magic == SUPER_V2 11 magic == conv2(BYTE_SWAP, SUPER_V2) {
22110         version = V2;
22111         native = (magic == SUPER_V2);
22112     } else {
22113         return(EINVAL);
22114     }
22115
22116     /* Si el superbloque tiene el orden de bytes equivocado, intercambiar los campos;
22117     * el número mágico no requiere conversión. */
22118     sp->s_ninodes = conv2(native, (int) sp->s_ninodes);
22119     sp->s_nzones = conv2(native, (int) sp->s_nzones);
22120     sp->s_imap_blockS = conv2(native, (int) sp->s_imap_blockS);
22121     sp->s_zmap_blockS = conv2(native, (int) sp->s_zmap_blockS);
22122     sp->s_firstdatazone = conv2(native, (int) sp->s_firstdatazone);
22123     sp->s_log_zone_size = conv2(native, (int) sp->s_log_zone_size);
22124     sp->s_max_size = conv4(native, sp->s_max_size);
22125     sp->s_zones = conv4(native, sp->s_zones);
22126
22127     /* En V1, el tamaño de disp. se guarda en un short, s_nzones, lo que limitaba
22128     * los dispositivos a 32K zonas. En V2 se decidió guardar el tamaño
22129     * como long. Empero, cambiar s_nzones a long no funcionaría, pues entonces
22130     * la posición de s_magic en el superbloque no sería la misma en los sistemas
22131     * V1 y V2, y no habría forma de saber si un sistema de archivos recién montado
22132     * es V1 o V2. La solución fue introducir una nueva variable, s_zones,
22133     * Y copiar el tamaño ahí.
22134     *

```

Archivo: src/fs/super.c

```

22135     * Calcular aquí también algunos otros números que dependen de la versión,
22136     * para ocultar algunas de las diferencias.
22137     */
22138     if (version == V1) {
22139         sp->s_zones = sp->s_nzones;                /* sólo V1 req esta copia */
22140         sp->s_inodes_per_block = V1_INODES_PER_BLOCK;
22141         sp->s_ndzones = V1_NR_DIONES;
22142         sp->s_nindirs = V1_INDIRECTS;
22143     } else {
22144         sp->s_inodes_per_block = V2_INODES_PER_BLOCK;
22145         sp->s_ndzones = V2_NR_DIONES;
22146         sp->s_nindirs = V2_INDIRECTS;
22147     }
22148
22149     sp->s_isearch = 0;                            /* búsquedas de nodos comienzan en 0 */
22150     sp->s_zsearch = 0;                            /* búsquedas de zonas comienzan en 0 */
22151     sp->s_version = version;
22152     sp->s_native = native;
22153
22154     /* Algunas verifs. básicas p/ver si el superbloque parece razonable. */
22155     if (sp->s_imap_blocks < 1 || sp->s_zmap_blocks < 1
22156         || sp->s_ninodes < 1 || sp->s_zones < 1
22157         || (unsigned) sp->s_log_zone_size > 4) {
22158         return(EINVAL);
22159     }
22160     sp->s_dev = dev;                              /* restaurar número dispositivo */
22161     return(OK);
22162 }

```

+++++

```

src/fs/filedes.c
+++++

```

+++++

```

22200     /* Este archivo contiene los procedo que manipulan descriptores de arch.
22201     *
22202     * Los puntos de entrada al archivo son
22203     *   get_fd: buscar descriptor de archivo libre y ranuras filp libres
22204     *   get_filp: buscar entrada filp p/descriptor de archivo dado
22205     *   find_filp: encontrar ranura filp que apunte a un nodo-i dado
22206     */
22207
22208     #include "fs.h"
22209     #include "file.h"
22210     #include "fproc.h"
22211     #include "inode.h"
22212
22213

```

```

22214 *                                     get_fd                                     *
22215

```

```

22216     PUBLIC int get_fd(start, bits, k, fpt)
22217         int start;                /* inicio búsqueda (para F_DUPFD) */
22218         mode_t bits;             /* modo del arch a crear (bits RWX) */
22219         int *k;                  /* dónde devolver descriptor de arch */
22220         struct filp **fptj;      /* dónde devolver ranura filp */
22221     {
22222         /* Buscar un descriptor de archivo libre y una ranura filp libre. Llenar palabra
22223         * de modo en la última pero no apropiarse de ninguno, pues aún podría
22224         * fallar el open() o creat().

```

```

22225     */
22226
22227 register struct filp *f;
22228 register int i;
22229
22230 *k = -1;                                /* necesitamos cómo saber si se halló desc arch */
22231
22232 /* Buscar un descriptor de archivo libre en la tabla fproc fp_filp. */
22233 for (i = startj i < OPEN_MAXj i++) {
22234     if (fp->fp_filp[i] == NIL_FILP) {
22235         /* Se encontró un descriptor de archivo. */
22236         *k = i;
22237         break;
22238     }
22239 }
22240
22241 /* Ver si se encontró un descriptor de archivo. */
22242 if (*k < 0) return(EMFILE);              /* por esto hicimos k = -1 inic. */
22243
22244 /* Ahora que hallamos un descriptor de archivo, buscar ranura filp libre. */
22245 for (f = &filp[0] j f < &filp[NR_FILPS]j f++) {
22246     if (f->filp_count == 0) {
22247         f->filp_mode = bits;
22248         f->filp_pos = 0L;
22249         f->filp_flags = 0;
22250         *fpt = f;
22251         return(OK);
22252     }
22253 }
22254
22255 /* Si llegamos aquí, la tabla filp está llena. Informar de ello. */
22256 return(ENFILE);
22257 }
22260
22261 *=====*
22262                                     get_filp                                     *
22263
22264 *=====*/
22263 PUBLIC struct filp *get_filp(fild)
22264 int fild;                                /* descriptor de archivo */
22265 {
22266     /* Ver si 'fild' se ref a un desc arch válido. Si si, devolver su apunt filp. */
22267
22268     err_code = EBAOF;
22269     if (fild < 0 || fild >= OPEN_MAX) return(NIL_FILP);
22270     return(fp->fp_filp[fild]);           /* puede ser también NIL FILP ~/
22271     }
22274
22275 *=====*
22276                                     find_filp                                    *
22277
22278 *=====*/
22277 PUBLIC struct filp *find_filp(rip, bits)
22278 register struct inode *rip;              /* nodo al que se ref el filp por hallar */
22279 Mode_t bits;                             /* modo del filp por hallar (bits RWX) */
22280 {
22281     /* Encontrar una ranura filp que se refiera al nodo-i 'rip' tal como describe
22282     * el bit de modo 'bit s'. Sirve p/determinar si alguien aún está interesado
22283     * en cualquier extremo de un conducto. También se usa al abrir un FIFO para hallar
22284     * con quién compartir un campo filp (p/compartir la posición en el archivo).

```


Archivo: src/fs/filedes.c

```

22285     * Al igual que 'get_fd', realiza una búsqueda lineal de la tabla filp.
22286     */
22287
22288     register struct filp *f;
22289
22290     for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
22291         if (f->filp_count != 0 && f->filp_ino == rip && (f->filp_mode & bitS)){
22292             return(f);
22293         }
22294     }
22295
22296     /* Si llegamos aquí, no estaba ahí el filp. Informar de ello. */
22297     return(NIL_FILP);
22298 }

```

+++

src/fs/lock.c

+++

```

22300     /* Este archivo maneja candados asesores de archivos según exige POSIX.
22301     *
22302     * Los puntos de entrada a este archivo son
22303     *   10Ck_op: realizar ops de candado para llamada al sistema FCNTL
22304     *   lock_revive: revivir procesos cuando se libera un candado
22305     */
22306
22307     #include "fs.h"
22308     #include <fcntl.h>
22309     #include <unistd.h>          /* cc se queda sin memoria con unistd.h :-( */
22310     #include "file.h"
22311     #include "fproc.h"
22312     #include "inode.h"
22313     #include "lock.h"
22314     #include "param.h"
22315
22316
22317     /* ===== */
22318     /*
22319     PUBLIC int 10Ck_op(f, req)
22320     struct filp *f;
22321     int req;          /* F_SETLK o bien F_SETLKW */
22322     {
22323     /* Aplicar los candados asesores requeridos por POSIX. */
22324
22325     int r, ltype, i, conflict = 0, unlocking = 0;
22326     mode_t mo;
22327     off_t first, last;
22328     struct flock flock;
22329     vir_bytes user_flock;
22330     struct file_lock *flp, *flp2, *emptyY;
22331
22332     /* Traer la estructura flock del espacio de usuario. */
22333     user_flock = (vir_bytes) name1;
22334     r = sys_copy(who, D, (phys_bytes) user_flock,
22335                 FS_PROC_NR, D, (phys_bytes) &flock, (phys_bytes) sizeof(flock));
22336     if (r != OK) return(EINVAL);
22337
22338     /* Realizar algunas verificaciones de errores. */
22339     ltype = flock.l_type;

```



```

22340 mo = f->filp_mode;
22341 if (ltype != F_UNLCK && ltype != F_RDLCK && ltype != F_WRLCK) return(EINVAL);
22342 if (req == F_GETLK && ltype == F_UNLCK) return(EINVAL);
22343 if ((f->filp_ino->i_mode & I_TYPE) != I_REGULAR) return(EINVAL);
22344 if (req != F_GETLK && ltype == F_RDLCK && (mo & R_BIT) == 0) return(EBADF);
22345 if (req != F_GETLK && ltype == F_WRLCK && (mo & W_BIT) == 0) return(EBADF);
22346
22347 /* Calcular primero y último bytes de la región asegurada. */
22348 switch (flock.l_whence) {
22349     case SEEK_SET:    first = 0; break;
22350     case SEEK_CUR:   first = f->filp_pos; break;
22351     case SEEK=END:  first = f->filp_ino->i_size; break;
22352     default:        return(EINVAL);
22353 }
22354 /* Verificar si hay desbordamiento. */
22355 if (((long)flock.l_start > 0) && ((first + flock.l_start) < first))
22356     return(EINVAL);
22357 if (((long)flock.l_start < 0) && ((first + flock.l_start) > first))
22358     return(EINVAL);
22359 first = first + flock.l_start;
22360 last = first + flock.l_len - 1;
22361 if (flock.l_len == 0) last = MAX_FILE_POS;
22362 if (last < first) return(EINVAL);
22363
22364 /* Ver si esta región está en conflicto con algún candado existente. */
22365 empty = (struct file_lock *) 0;
22366 for (flp = &file_lock[0]; flp < &file_lock[NR_LOCKS]; flp++) {
22367     if (flp->lock_type == 0) {
22368         if (empty == (struct file_lock *) 0) empty = flp;
22369         continue; /* 0 = ranura desocupada */
22370     }
22371     if (flp->lock_inode != f->filp_ino) continue; /* diferente arch */
22372     if (last < flp->lock_first) continue; /* nuevo al frente */
22373     if (first > flp->lock_last) continue; /* nuevo está después */
22374     if (ltype == F_RDLCK && flp->lock_type == F_RDLCK) continue;
22375     if (ltype != F_UNLCK && flp->lock_pid == flp->fp_pid) continue;
22376
22377     /* podría haber conflicto. Procesarlo. */
22378     conflict = 1;
22379     if (req == F_GETLK) break;
22380
22381     /* Si tratábamos de poner un candado, recién falló. */
22382     if (ltype == F_RDLCK || ltype == F_WRLCK) {
22383         if (req == F_SETLK) {
22384             /* Para F_SETLK, sólo informar fracaso. */
22385             return(EAGAIN);
22386         } else {
22387             /* Para F_SETLKW, suspender el proceso. */
22388             suspend(XLOCK);
22389             return(0);
22390         }
22391     }
22392
22393     /* Estamos quitando un candado y encontramos algo que traslapa. */
22394     unlocking = 1;
22395     if (first <= flp->lock_first && last >= flp->lock_last) {
22396         flp->lock_type = 0; /* marcar ranura como desocupada */
22397         nr_locks--; /* núm candados ahora es 1 menos */
22398         continue;
22399     }

```

```
22400
22401     /* Parte de una región asegurada se desaseguró. */
22402     if (first <= flp->lock_first) {
22403         flp->lock_first = last + 1;
22404         continue;
22405     }
22406
22407     if (last >= flp->lock_last) {
22408         flp->lock_last = first -1;
22409         continue;
22410     }
22411
22412     /* Mala suerte. Candado partido en 2 al desasegurar en medio. */
22413     if (nr_locks == NR_LOCKS) return(ENOLCK);
22414     for (i = 0; i < NR_LOCKS; i++)
22415         if (file_lock[i].lock_type == 0) break;
22416     flp2 = &file_lock[i];
22417     flp2->lock_type = flp->lock_type;
22418     flp2->lock_pid = flp->lock_pid;
22419     flp2->lock_inode = flp->lock_inode;
22420     flp2->lock_first = last + 1;
22421     flp2->lock_last = flp->lock_last;
22422     flp->lock_last = first -1;
22423     nr_locks++;
22424 }
22425 if (unlocking) lock_revive();
22426
22427 if (req == F_GETLK) {
22428     if (conflict) {
22429         /* GETLK Y conflicto. Informar de candado en conflicto. */
22430         flock.l_type = flp->lock_type;
22431         flock.l_whence = SEEK_SET;
22432         flock.l_start = flp->lock_first;
22433         flock.l_len = flp->lock_last - flp->lock_first + 1;
22434         flock.l_pid = flp->lock_pid;
22435     } else {
22436         /* Es GETLK y no hay conflicto. */
22437         flock.l_type = F_UNLCK;
22438     }
22439 }
22440
22441     /* Copiar estructura flock de vuelta al invocador. */
22442     r = sys_copy (FS_PROC_NR , D, (phys_bytes) &flock,
22443                 who, D, (phys_bytes) user_flock, (phys_bytes) sizeof(flock));
22444     return(r);
22445 }
22446
22447 if (ltype == F_UNLCK) return(OK) ;           /* desaseguró región sin candados */
22448
22449 /* No hay conflicto. Si hay espacio, guardar nuevo candado en la tabla. */
22450 if (empty == (struct file_lock *) 0) return(ENOLCK); /* tabla llena */
22451 empty->lock_type = ltype;
22452 empty->lock_pid = fp->fp_pid;
22453 empty->lock_inode = f->filp_ino;
22454 empty->lock_first = first;
22455 empty->lock_last = last;
22456     nr_locks++;
22457 return(OK);
22458 }
```

```

22460
/*=====
22461 *                               lock_revive                               *
22462 *=====
*/
22463 PUBLIC void lock_revive()
22464 {
22465     /* Encontrar todos los procesos que están esperando por cualquier tipo
22466     * de candado y'revivirlos. Los que aún estén bloqueados se bloquearán
22467     * otra vez al ejecutarsej los demás acabarán. Esta estrategia es un trueque
22468     * espacio-tiempo. Necesitaríamos código extra para saber exactamente
22469     * cuáles desbloquear ahora, y sólo ganaríamos un poco en rendimiento
22470     * en casos extremadamente raros (a saber, si alguien usó realmente
22471     * candados.)
22472     */
22473
22474     int task;
22475     struct fproc *fptr;
22476
22477     for (fptr = &fproc[INIT_PROC_NR + 1]j fptr < &fproc[NR_PROCS]j fptr++){
22478         task = -fptr->fp_task;
22479         if (fptr->fp_suspended == SUSPENDED && task == XLOCK) {
22480             revive( (int) (fptr -fproc), 0);
22481         }
22482     }
22483 }
++++
+++
src/fs/main.c
++++
22500 /* Este archivo contiene el programa principal del Sistema de Archivos. Consiste
22501 * en un ciclo que obtiene mensajes que solicitan trabajo, realiza el trabajo y
22502 * devuelve respuestas.
22503 *
22504 * Los puntos de entrada a este archivo son
22505 *   main:          programa principal del Sistema de Archivos
22506 *   reply: enviar respuesta a un proc después de realizar trab solicitado
22507 */
22508
22509 struct super_block;          /* proto.h necesita saber esto */
22510
22511 #include "fs.h"
22512 #include <fcntl.h>
22513 #include <string.h>
22514 #include <sys/ioctl.h>
22515 #include <minix/callnr.h>
22516 #include <minix/com.h>
22517 #include <minix/boot.h>
22518 #include "buf.h"
22519 #include "dev.h"
22520 #include "file.h"
22521 #include "fproc.h"
22522 #include "inode.h"
22523 #include "param.h"
22524 #include "super.h"
22525
22526 FORWARD _PROTOTYPE( void buf_pool, (void) );
22527 FORWARD _PROTOTYPE( void fs_init, (void) );
22528 FORWARD _PROTOTYPE( void get_boot_parameters, (void) );
22529 FORWARD _PROTOTYPE( void get_work, (void) );

```

Archivo: src/fs/main.c

```

22530 FORWARD_PROTOTYPE( void load_ram, (void) );
22531 FORWARD_PROTOTYPE( void load_super, (Dev_t super_dev) );
22532
22533
22534
/*=====*/
22535 *                               main                               *
22536
/*=====*/
22537 PUBLIC void main()
22538 {
22539     /* Éste es el programa principal del sistema de archivos. El ciclo principal
22540 * tiene 3 actividades principales: obtener nuevo trabajo, procesarlo y responder.
22541 * El ciclo nunca termina en tanto se ejecuta el sistema de archivos.
22542 */
22543     int error;
22544
22545     fs_init();
22546
22547     /* Ciclo principal que obtiene trabajo, lo procesa y responde. */
22548     while (TRUE) {
22549         get_work();                /* establece who y fs_call */
22550
22551         fp = &fproc[who];          /* apunt a struct de tabla proc */
22552         super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);          /* ¿su? */
22553         dont_reply = FALSE;        /* o sea, responder es default */
22554
22555         /* Llamar la función interna que realiza el trabajo. */
22556         if (fs_call < 0 || fs_call >= NCALLS)
22557             error = EBADCALL;
22558         else
22559             error = (*call_vector[fs_call]) ();
22560
22561         /* Copiar resultados al usuario y enviar respuesta. */
22562         if (dont_reply) continue;
22563         reply(who, error);
22564         if (rdahed_inode != NIL_INODE) read_ahead(); /* leer blq anticip */
22565     }
22566 }
22569
/*=====*/
22570 *                               get_work                               *
22571
/*=====*/
22572 PRIVATE void get_work()
22573 {
22574     /* Normalmente esperar nuevas entradas, pero si 'reviving' es dif.
22575 * de cero, hay que despertar un proceso suspendido.
22576 */
22577
22578     register struct fproc *rp;
22579
22580     if (reviving != 0) {
22581         /* Revivir un proceso suspendido. */
22582         for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++)
22583             if (rp->fp_revived == REVIVING) {
22584                 who = (int)(rp - fproc);
22585                 fs_call = rp->fp_fd & BYTE;
22586                 fd = (rp->fp_fd >> 8) & BYTE;
22587                 buffer = rp->fp_buffer;
22588                 nbytes = rp->fp_nbytes;
22589                 rp->fp_suspended = NOT_SUSPENDED; /*ya no espera*/

```



```

22590             rp->fp_revived = NOT_REVIVINGj
22591             reviving--;
22592             return;
22593         }
22594         panic("get_work couldn't revive anyone", NO_NUM);
22595     }
22596
22597     /* Caso normal. Nadie a quien revivir. */
22598     if (receive(ANY, &m) != OK) panic("fs receive error", NO_NUM);
22599
22600     who = m.m_source;
22601     fs_call = m.m_type;
22602     }
22605
22606 *=====*
22607     reply
22608 *=====*/
22609     PUBLIC void reply(whom, result)
22610         int whom;      /* proceso al cual responder */
22611         int result;    /* resultado (usualmente OK o # error) */
22612     {
22613         /* Enviar respuesta a proceso de usuario. Puede fallar (si el proceso
22614          * acaba de matarse por una señal), así que no verificar el código de retorno.
22615          * Si falla el envío, hacer caso omiso.
22616          */
22617         reply_type = result;
22618         send(whom, &m1);
22619     }
22622
22623 *=====*
22624 *=====*/
22625     PRIVATE void fs_init()
22626     {
22627         /* Inicializar variables globales, tablas, etc. */
22628
22629         register struct inode *rip;
22630         int i;
22631         message mess;
22632
22633         /* Inicializaciones necesarias para que dev_opc1 tenga éxito.*/
22634         fp = (struct fproc *) NULL;
22635         who = FS_PROC_NR;
22636
22637         buf_pool();          /* inicializar reserva de buffers */
22638         get_boot_parameters(); /* obtener parámetros del menú */
22639         load_ram();          /* inic disco RAM, cargar si raíz */
22640         load_super(ROOT_DEV); /* cargar superbloque disp raíz */
22641
22642         /* Inicializar campos 'fproc' para proceso 0 ..INIT. */
22643         for (i = 0; i <= LOW_USER; i += 1) {
22644             if (i == FS_PROC_NR) continue;      /* ~ no inicializar FS */
22645             fp = &fproc[i];
22646             rip = get_inode (ROOT_DEV, ROOT_INODE);
22647             fp->fp_rootdir = rip;
22648             dup_inode(rip);
22649             fp->fp_workdir = rip;

```



```
22650     fp->fp_realuid = (uid_t) SYS_UID;
22651     fp->fp_effuid = (uid_t) SYS_UID;
22652     fp->fp_realgid = (gid_t) SYS_GID;
22653     fp->fp_effgid = (gid_t) SYS_GID;
22654     fp->fp_umask = -0;
22655 }
22656
22657 /* Deben cumplirse ciertas relaciones para que el FS funcione. */
22658 if (SUPER_SIZE > BLOCK_SIZE) panic("SUPER_SIZE > BLOCK_SIZE", NO_NUM);
22659 if (BLOCK_SIZE % V2_INODE_SIZE != 0) /* verif V1_INODE_SIZE también */
22660     panic("BLOCK_SIZE % V2_INODE_SIZE != 0", NO_NUM);
22661 if (OPEN_MAX > 127) panic("OPEN_MAX > 127", NO_NUM);
22662 if (NR_BUFS < 6) panic("NR_BUFS < 6", NO_NUM);
22663 if (V1_INODE_SIZE != 32) panic("V1 inode size != 32", NO_NUM);
22664 if (V2_INODE_SIZE != 64) panic("V2 inode size != 64", NO_NUM);
22665 if (OPEN_MAX > 8 * sizeof(long)) panic("Too few bits in fp_cloexec", NO_NUM);
22666
22667 /* Decir tarea de memoria dónde está mi tabla de proc para benef ps(1). */
22668 mess.m_type = DEV_IOCTL;
22669 mess.PROC_NR = FS_PROC_NR;
22670 mess.REQUEST = MIOCSPSINFO;
22671 mess.ADDRESS = (void *) fproc;
22672 (void) sendrec(MEM, &mess);
22673 }
22674
/*=====
22677 *                                     buf_pool *
22678 *=====
==*/
22679     PRIVATE void buf_pool()
22680     {
22681         /* Inicializar la reserva de buffers. */
22682
22683         register struct buf *bp;
22684
22685         bufs_in_use = 0;
22686         front = &buf[0];
22687         rear = &buf[NR_BUFS - 1];
22688
22689         for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) {
22690             bp->b_blocknr = NO_BLOCK;
22691             bp->b_dev = NO_DEV;
22692             bp->b_next = bp + 1;
22693             bp->b_prev = bp - 1;
22694         }
22695         buf[0].b_prev = NIL_BUF;
22696         buf[NR_BUFS - 1].b_next = NIL_BUF;
22697
22698         for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) bp->b_hash = bp->b_next;
22699         buf_hash[0] = front;
22700     }
22701
/*=====
22704 *                                     get_boot_parameters *
22705 *=====
==*/
22706     PUBLIC struct bparam_s boot_parameters;
22707
22708     PRIVATE void get_boot_parameters()
22709     {
```

```

22710  /* Pedir al kernel parámetros de arranque. */
22711
22712  m1.m_type = SYS_GBOOT;
22713  m1.PROC1 = FS_PROC_NR;
22714  m1.MEM_PTR = (char *) &boot_parameters;
22715  (void) sendrec(SYSTASK, &m1);
22716  }
22719
/*=====
22720  *                               load_ram                               *
22721  *=====
*/
22722  PRIVATE void load_ram()
22723  {
22724  /* Si el disp raíz es el disco RAM, copiar todo el disp de imagen raíz
22725  * bloque por bloque en un disco RAM con el mismo tamaño que la imagen.
22726  * Si no, asignar un disco en RAM con el tamaño dado en paráms de arranque.
22727  */
22728
22729  register struct buf *bp, *bp1;
22730  long k_loaded, lcount;
22731  u32_t ram_size, fsmax;
22732  zone_t zones;
22733  struct super_block *sp, *dsp;
22734  block_t b;
22735  int majar, task;
22736  message dev_mess;
22737
22738  ram_size = boot_parameters.bp_ramsize;
22739
22740  /* Abrir el dispositivo raíz. */
22741  majar = (ROOT_DEV » MAJOR) & BYTE; /* núm de dispositivo principal */
22742  task = dmap[majar].dmap_task; /* núm de tarea de dispositivo */
22743  dev_mess.m_type = DEV_OPEN; /* distinguir de cerrar */
22744  dev_mess.DEVICE = ROOT_DEV;
22745  dev_mess.COUNT = R_BITW_BIT;
22746  (*dmap[majar].dmap_open) (task, &dev_mess);
22747  if (dev_mess.REP_STATUS != OK) panic("Cannot open root device", NO_NUM);
22748
22749  /* Si el disp raíz es el disco RAM, llenarlo del dispositivo de imagen. */
22750  if (ROOT_DEV == DEV_RAM) {
22751      majar = (IMAGE_DEV » MAJOR) & BYTE; /* núm disp principal */
22752      task = dmap[majar].dmap_task; /* núm tarea de disp */
22753      dev_mess.m_type = DEV_OPEN; /* distinguir de cerrar */
22754      dev_mess.DEVICE = IMAGE_DEV;
22755      dev_mess.COUNT = R_BIT;
22756      (*dmap[majar].dmap_open) (task, &dev_mess);
22757      if (dev_mess.REP_STATUS != OK) panic("Cannot open root device", NO_NUM);
22758
22759      /* Obtener tamaño de disco RAM leyendo superbloque de sist arch raíz. */
22760      sp = &super_block[0];
22761      sp->s_dev = IMAGE_DEV;
22762      if (read_super(sp) != OK) panic("Bad root file system", NO_NUM);
22763
22764      lcount = sp->s_zones « sp->s_log_zone_size; /* # blqs disp raíz*/
22765
22766      /* Estirar el FS del disco en RAM al tamaño dado por los paráms de arranque,
22767      * pero no más allá de lo que permite el últ. bloque del mapa de bits de zonas.
22768      */
22769      if (ram_size < lcount) ram_size = lcount;

```



```

22830 *                               load_super                               *
22831
=====*/
22832     PRIVATE void load_super(super_dev)
22833                               dev_t super_dev;           /* de dónde obtener
superbloque */
22834     {
22835     int bad;
22836     register struct super_block *sp;
22837     register struct inode *rip;
22838
22839     /* Inicializar la tabla de superbloques. */
22840     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
22841         sp->s_dev = NO_DEV;
22842
22843     /* Leer superbloque para sistema de archivos raíz. */
22844     sp = &super_block[0];
22845     sp->s_dev = super_dev;
22846
22847     /* Verif. consistencia superbloque (¿es el disquete correcto?). */
22848     bad = (read_super(sp) != OK);
22849     if (!bad) {
22850         rip = get_inode(super_dev, ROOT_INODE); /* nodo-i de dir raíz */
22851         if ((rip->i_mode & I_TVPE) != I_DIRECTORV 11 rip->i_nlinks < 3) bad++;
22852     }
22853     if (bad)panic("Invalid root file system. Possibly wrong diskette.",NO_NUM);
22854
22855     sp->s_imount = rip;
22856     dup_inode(rip);
22857     sp->s_isup = rip;
22858     sp->s_rd_only = 0;
22859     return;
22860 }
++++
+++
src/fs/open.c
++++
+++
22900     /* Este archivo contiene los procedimientos para crear, abrir, cerrar
22901     * Y realizar búsquedas en archivos.
22902     *
22903     * Los puntos de entrada a este archivo son
22904     * do_creat:     ejecutar llamada al sistema CREAT
22905     * do_open:     ejecutar llamada al sistema OPEN
22906     * do_mknod:    ejecutar llamada al sistema MKNOD
22907     * do_mkdir:    ejecutar llamada al sistema MKDIR
22908     * do_clase:    ejecutar llamada al sistema CLaSE
22909     * do_lseek:    ejecutar llamada al sistema LSEEK
22910 */
22911
22912 #include "fs.h"
22913 #include <sys/stat.h> 22914 #include <fcntl.h>
22915 #include <minix/callnr.h>
22916 #include <minix/com.h>
22917 #include "buf.h"
22918 #include "dev.h"
22919 #include "file.h"

```



```
22920 #include "fproc.h" 22921 #include "inode.h" 22922 #include "lock.h"
22923 #include "param.h"
22924
22925 PRIVATE message dev mess;
22926 PRIVATE char mode_roap[] = {R_BIT, W_BIT, R_BITIW_BIT, 0};
22927
22928
22929     FORWARD_PROTOTYPE( int common_open, (int oflags, Mode_t omode)
22930     FORWARD_PROTOTYPE( int pipe_open, (struct inode *rip, Mode_t bits, int oflags));
22931     FORWARD_PROTOTYPE( struct inode *new_node, (char *path, Mode_t bits,
22932     zone_t z0) )j
22933
22934
22935 *
22936
22937 PUBLIC int do_creat()
22938 {
22939     /* Ejecutar la llamada al sistema creat(name, mode). */
22940     int r;
22941
22942     if (fetch_name(name, name_length, M3) != OK) return(err_code);
22943     r = common_open(O_WRONLY | O_CREAT | O_TRUNC, (mode_t) mode);
22944     return(r);
22945 }
22948/*=====
====*
22949 *
22950 *
22951 PUBLIC int do_open()
22952 {
22953     /* Realizar la llamada al sistema open(name, flags,...). */
22954
22955     int create_mode = 0; /* realmente mode_t pero da problemas */
22956     int r;
22957
22958     /* Si O_CREAT es 1, open tiene 3 parámetros, si no, 2. */
22959     if (mode & O_CREAT) {
22960         create_mode = c_mode;
22961         r = fetch_name(c_name, name1_length, M1);
22962     } else {
22963         r = fetch_name(name, name_length, M3);
22964     }
22965
22966     if (r != OK) return(err_code); /* nombre erróneo */
22967     r = common_open(mode, create_mode);
22968     return(r);
22969 }
22972/*=====
====*
22973 *
22974
22975 PRIVATE int common_open(oflags, omode)
22976     register int oflags;
22977     mode_t omode;
22978 {
22979     /* Código común de do_creat y do_apeno */
```

```

22980
22981 register struct inode *rip;
22982 int r, b, major, task, exist = TRUE;
22983 dev_t dev;
22984 mode_t bits;
22985 off_t pos;
22986 struct filp *fil_ptr, *filp2;
22987 .
22988 /* Remapear los dos bits inferiores de oflags. */
22989 bits = (mode_t) mode_map[oflags & O_ACCMODE];
22990
22991 /* Ver si están disponibles descriptor de archivo y ranuras filp. */
22992 if ( (r = get_fd(0, bits, &fd, &fil_ptr)) != OK) return(r);
22993
22994 /* Si O_CREATE es 1, tratar de crear el archivo. */
22995 if (oflags & O_CREAT) {
22996     /* Crear nodo-i nuevo invocando new_node(). */
22997     omode = I_REGULAR | (omode & ALL_MODES & fp->fp_umask);
22998     rip = new_node(user_path, omode, NO_IONE);
22999     r = err_code;
23000     if (r == OK) exist = FALSE;           /* recién creamos el archivo */
23001     else if (r != EEXIST) return(r);     /* otro error */
23002     else exist = ! (oflags & O_EXCL); /* archivo existe; si bandera O_EXCL
23003                                     izada, esto es un error */
23004 } else {
23005     /* Examinar nombre de ruta. */
23006     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
23007 }
23008
23009 /* Apartar descriptor de archivo y ranura filp y llenarlos. */
23010 fp->fp_filp[fd] = fil_ptr;
23011 fil_ptr->filp_count = 1;
23012 fil_ptr->filp_ino = rip;
23013 fil_ptr->filp_flags = oflags;
23014
23015 /* Sólo ejecutar código de open normal si no creamos el archivo. */
23016 if (exist) {
23017     /* Verificar protecciones. */
23018     if ((r = forbidden(rip, bits)) == OK) {
23019         /* Apertura archs normales, dirs y archs esp. difiere. */
23020         switch (rip->i_mode & I_TYPE) {
23021             case I_REGULAR:
23022                 /* Truncar arch normal si O_TRUNC */
23023                 if (oflags & O_TRUNC) {
23024                     if ((r = forbidden(rip, W_BIT)) != OK) break;
23025                     truncate(rip);
23026                     wipe_inode(rip);
23027                     /* Enviar nodo-i de caché de nodos-i
23028                      * al caché de bloques pique se escriba en sigte
23029                      * desalojo de caché.
23030                      */
23031                     rw_inode(rip, WRITING);
23032                 }
23033                 break;
23034
23035             case I_DIRECTORY:
23036                 /* Directorios pueden leerse pero no escribirse. */
23037                 r = (bits & W_BIT ? EISDIR : OK);
23038                 break;
23039

```

```

23040 case I_CHAR_SPECIAL:
23041 case I_BLOCK_SPECIAL:
23042     /* Invocar controlador p/procesam. especial. */
23043     dev_mess.m_type = DEV_OPEN;
23044     dev = (dev_t) rip->i_zone[0];
23045     dev_mess.DEVICE = dev;
23046     dev_mess.COUNT = bits I (oflags & -O_ACCMODE);
23047     majar = (dev » MAJOR) & BYTE; /* # disp princ */
23048     if (majar <= 0 11 majar >= max_majar) {
23049         r = ENODEV;
23050         break;
23051     }
23052     task = dmap[majar].dmap_task; /* # tarea disp */
23053     (*dmap[majar].dmap_open) (task, &dev_mess);
23054     r = dev_mess.REP_STATUS;
23055     break;
23056
23057 case I_NAMED_PIPE:
23058     oflags |= O_APPEND; /* modo anexión forzada */
23059     fil_ptr->filp_flags = oflags;
23060     r = pipe_open(rip, bits, oflags);
23061     if (r == OK) {
23062         /* Ver si alguien más lee o escribe en FIFO.
23063          * Si sí, usar su entrada filp plque la posición
23064          * de archivo se comparta automáticamente.
23065          */
23066         b = (bits & R_BIT ? R_BIT : W_BIT);
23067         fil_ptr->filp_count = 0; /* no se encuentra */
23068         if ((filp2 = find_filp(rip, b)) != NIL_FILP) {
23069             /* Coescritor o lector hallado. Usarlo. */
23070             fp->fp_filp[fd] = filp2;
23071             filp2->filp_count++;
23072             filp2->filp_ino = rip;
23073             filp2->filp_flags = oflags;
23074
23075             /* eatpath incrementó i_count
23076              * erróneamente porque no sabía
23077              * que íbamos a usar una entrada
23078              * filp existente. Corregir.
23079              */
23080             rip->i_count--;
23081         } else {
23082             /* Nadie más hallado. Restaurar f~lp. */
23083             fil_ptr->filp_count = 1;
23084             if (b == R_BIT)
23085                 pos = rip->i_zone[V2_NR_DZONES+1];
23086             else
23087                 pos = rip->i_zone[V2_NR_DZONES+2];
23088             fil_ptr->filp_pos = pos;
23089         }
23090     }
23091     break;
23092 }
23093 }
23094 }
23095
23096 /* Si hay error, liberar nodo-i. */
23097 if (r != OK) {
23098     fp->fp_filp[fd] = NIL_FILP;
23099     fil_ptr->filp_count = 0;

```



```

23100     put_inode(rip);
23101     return(r) ;
23102     }
23103
23104     return(fd);
23105     }
23108
/*=====
23109 *                               new_node                               *
23110 *=====*/
23111 PRIVATE struct inode *new_node(path, bits, z0)
23112 char *path;                               /* apunt a nombre de ruta */
23113 mode_t bitsj                               /* modo del nuevo nodo-i */
23114 zone_t z0j                               /* núm zona 0 p/nuevo nodo-i */
23115 {
23116     /* New_node() es invocado por common_open(), do_mknod() Y do_mkdir().
23117     * En todos los casos asigna un nuevo nodo-i, crea una entrada de directorio
23118     * p/él en la ruta 'path' y la inicializa. Devuelve un apunt al nodo-i
23119     * si puede hacer esto; si no, devuelve NIL_INODE. Siempre asigna
23120     * valor apropiado a 'err_code' (OK o código de error).
23121     */
23122
23123     register struct inode *rlast_dir_ptr, *rip;
23124     register int r;
23125     char string[NAME_MAX];
23126
23127     /* Ver si la ruta puede abrirse hasta el últ. directorio. */
23128     if ((rlast_dir_ptr = last_dir(path, string)) == NIL_INODE) return(NIL_INODE);
23129
23130     /* Directorio final accesible, obtener componente final de la ruta. */
23131     rip = advance(rlast_dir_ptr, string);
23132     if (rip == NIL_INODE && err_code == ENOENT) {
23133         /* Últ. comp. de ruta no existe. Crear nueva entrada directorio. */
23134         if ((rip = alloc_inode(rlast_dir_ptr->i_dev, bits)) == NIL_INODE) {
23135             /* Imposible crear nuevo nodo-i: no hay más. */
23136             put_inode(rlast_dir_ptr);
23137             return(NIL_INODE);
23138         }
23139
23140         /* Forzar nodo-i a disco antes de crear entrada de directorio
23141         * para hacer al sistema más robusto ante caídas: un nodo-i
23142         * sin entrada de directorio es mucho mejor que lo contrario.
23143         */
23144         rip->i_nlinks++;
23145         rip->i_zone[0] = z0;                               /* núms disp principal/secund */
23146         rw_inode(rip, WRITING);                          /* forzar nodo-i a disco ahora */
23147
23148         /* Nuevo nodo-i adquirido. Tratar de crear entrada directorio. */
23149         if ((r = search_dir(rlast_dir_ptr, string, &rip->i_num, ENTER)) != OK) {
23150             put_inode(rlast_dir_ptr);
23151             rip->i_nlinks--;                               /* lástima, hay que liberar nodo-i */
23152             rip->i_dirt = DIRTY;                          /* nodos-i sucios se escriben */
23153             put_inode(rip); /* esta llamada libera el nodo-i */
23154             err_code = r;
23155             return(NIL_INODE);
23156         }
23157     } else {
23158         /* El último componente existe o hay algún problema. */

```



```

23160         if (rip != NIL_INODE)
23161             r = EEXIST;
23162         else
23163             r = err_code;
23164     }
23165
23166     /* Devolver el nodo-i de directorio y salir. */
23167     put_inode(rlast_dir_ptr);
23168     err_code = r;
23169     return(rip);
23170 }
23173
/*=====
23174 *                                     pipe_open                                     *
23175*=====
= */
23176     PRIVATE int pipe_open(rip, bits, oflags)
23177     register struct inode *rip;
23178     register mode_t bits;
23179     register int oflags;
23180     {
23181     /* Esta función se llama desde common_open. Ve si hay
23182     * por lo menos un par lector/escritor para el conducto; si no,
23183     * suspende al invocador, si si, revive todos los demás procesos
23184     * bloqueados que esperan el conducto.
23185     */
23186
23187     if (find_filp(rip, bits & W_BIT ? R_BIT : W_BIT) == NIL_FILP) {
23188         if (oflags & O_NONBLOCK) {
23189             if (bits & W_BIT) return(ENXIO);
23190         } else
23191             suspend(XPOPEN); /* suspender invocador */
23192     } else if (susp_count > 0) /* revivir procesos bloqueados */
23193         release(rip, OPEN, susp_count);
23194         release(rip, CREAT, susp_count);
23195     }
23196     rip->i_pipe = I_PIPE;
23197
23198     return(OK);
23199 }
23202/*=====
==*23203 *                                     do_mknod                                     *
*
23204
*===== */
23205     PUBLIC int do_mknod()
23206     {
23207     /* Realizar la llamada al sistema mknod(name, mode, addr). */
23208
23209     register mode_t bits, mode_bits;
23210     struct inode *ip;
23211
23212     /* Sólo el superusuario puede crear cosas que no sean fifos. */
23213     mode_bits = (mode_t) m.m1_i2; /* modo del nodo-i */
23214     if (!super_user && ((mode_bits & I_TYPE) != I_NAMED_PIPE)) return(EPERM);
23215     if (fetch_name(m.m1_p1, m.m1_i1, M1) != OK) return(err_code);
23216     bits = (mode_bits & I_TYPE) | (mode_bits & ALL_MODES & fp->fp_umask);
23217     ip = new_node(uSer_path, bits, (zone_t) m.m1_i3);
23218     put_inode(ip);
23219     return(err_code);

```



```

23220 }
23223
/*=====
23224 *                               do_mkdir                               *
23225 *=====
*/

23226 PUBLIC int do_mkdir()
23227 {
23228     /* Ejecutar llamada al sistema mkdir(name, mode). */
23229
23230     int r1, r2;                               /* códigos de situación */
23231     ino_t dot, dotdot;                         /* núms nodos-i para. y ../ */
23232     mode_t bits;                               /* bits de modo p/nuevo nodo-i */
23233     char string[NAME_MAX];                     /* últ. comp de nombre ruta nuevo dir */
23234     register struct inode *rip, *ldirp;
23235
23236     /* Ver si es posible crear otro vínculo en dir padre. */
23237     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
23238     ldirp = last_dir(user_path, string);        /* apunt a padre de nuevo dir */
23239     if (ldirp == NIL_INODE) return(err_code);
23240     if ((ldirp->i_nlinks & BYTE) >= LINK_MAX) {
23241         put_inode(ldirp);                       /* devolver padre */
23242         return(EMLINK);
23243     }
23244
23245     /* Ahora crear nodo-i. Si fracasa, devolver código de error. */
23246     bits = I_DIRECTORY 1 (mode & RWX_MODES & fp->fp_umask);
23247     rip = new_node(user_path, bits, (zone_t) 0);
23248     if (rip == NIL_INODE || err_code == EEXIST) {
23249         put_inode(rip);                         /* imposible crear dir; ya existe */
23250         put_inode(ldirp);                       /* devolver padre también */
23251         return(err_code);
23252     }
23253
23254     /* Obtener núms nodo-i para. y ../colocar en directorio. */
23255     dotdot = ldirp->i_num;                       /* núm nodo-i del padre */
23256     dot = rip->i_num;                             /* núm nodo-i nuevo dir mismo */
23257
23258     /* Crear entradas dir para. y ../a menos que disco totalmente lleno. */
23259     /* Usar dot1 y dot2, así que el modo del dir no es importante. */
23260     rip->i_mode = bits;                           /* fijar modo */
23261     r1 = search_dir(rip, dot1 J &dot, ENTER);    /* poner. en nvo dir */
23262     r2 = search_dir(rip, dot2, &dotdot, ENTER); /* poner.. en nvo dir */
23263
23264     /* Si se ingresaron. y ../con éxito, increm. cuentas de vínculos. */
23265     if (r1 == OK && r2 == OK) {
23266         /* Caso normal: pudimos ingresar. y ../en nuevo dir. */
23267         rip->i_nlinks++;                          /* esto da cuenta de . */
23268         ldirp->i_nlinks++;                        /* esto da cuenta de ../ */
23269         ldirp->i_dirt = DIRTY;                   /* marcar nodo-i padre c/sucio */
23270     } else {
23271         /* No pudimos ingresar. ni ../; quizá disco lleno. */
23272         (void) search_dir(ldirp, string, (ino_t *) 0, DELETE);
23273         rip->i_nlinks--;                          /* anular incremento en new_node() */
23274     }
23275     rip->i_dirt = DIRTY;                          /* como sea, i_nlinks ha cambiado */
23276
23277     put_inode(ldirp);                            /* devolver nodo-i de dir padre */
23278     put_inode(rip);                              /* devolver nodo-i de dir nuevo */
23279     return(err_code);                           /* new_node() siempre fija .err_code' */

```

```

23280 }
23283 *=====
23284 *                               do_clase                               *
23285 *=====*/
23286     PUBLIC int do_close() ,
23287 {
23288     /* Ejecutar la llamada al sistema close(fd). */
23289
23290     register struct filp *rfilp;
23291     register struct inode *rip;
23292     struct file_lock *flp;
23293     int rw, mode_word, majar, task, lock_count;
23294     dev_t dev;
23295
23296     /* Primero encontrar nodo-i que pertenece al descriptor de archivo. */
23297     if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);
23298     rip = rfilp->filp_ino;          /* 'rip' apunta al nodo-i */
23299
23300     if (rfilp->filp_count -1 == 0 && rfilp->filp_mode != FILP_CLOSED) {
23301         /* Ver si el archivo es especial. */
23302         mode_word = rip->i_mode & I_TYPE;
23303         if (mode_word == I_CHAR_SPECIAL || mode_word == I_BLOCK_SPECIAL) {
23304             dev = (dev_t) rip->i_zone[0];
23305             if (mode_word == I_BLOCK_SPECIAL) {
23306                 /* Invalidar entradas de caché a menos que un especial
23307                  * esté montado o sea ROOT
23308                  */
23309                 if (!mounted(rip)) {
23310                     (void) do_sync();          /* purgar caché */
23311                     invalidate(dev);
23312                 }
23313             }
23314             /* Usar entrada dmap_close para realizar cualquier procesamiento
23315              * especial que se requiera.
23316              */
23317             dev_mess.m_type = DEV_CLOSE;
23318             dev_mess.DEVICE = dev;
23319             majar = (dev >> MAJOR) & BYTE;     /* núm disp principal */
23320             task = dmap[majar].dmap_task;     /* núm tarea disp */
23321             (*dmap[majar].dmap_close)(task, &dev_mess);
23322         }
23323     }
23324
23325     /* Si el nodo que se cierra es conducto, liberar a todos los que lo esperan. */
23326     if (rip->i_pipe == I_PIPE) {
23327         rw = (rfilp->filp_mode & R_BIT ? WRITE : READ);
23328         release(rip, rw, NR_PROCS);
23329     }
23330
23331     /* Si se escribió, el nodo-i ya está marcado DIRTY. */
23332     if (--rfilp->filp_count == 0) {
23333         if (rip->i_pipe == I_PIPE && rip->i_count > 1) {
23334             /* Guardar la pos. en el arch en el nodo-i en caso de que se nece
23335              * site después. Las pos. de lectura y escrit se guardan aparte.
23336              * Las últ. 3 zonas del nodo-i no se usan p/conductos (nombrados).
23337              */
23338             if (rfilp->filp_mode == R_BIT)
23339                 rip->i_zone[V2_NR_DZONES+1] = (zone_t) rfilp->filp_pos;

```

```

23340         else
23341             rip->i_zone[V2_NR_DZONES+2] = (zone_t) rfilp->filp_pos;
23342     }
23343     put_inode(rip);
23344 }
23345
23346 fp->fp_cloe~ec &= ~(1L << fd);           /* apagar bit cerrar-al-ejec */
23347 fp->fp_filp[fd] = NIL_FILP;
23348
23349 /* Ver si el archivo tiene candado. Soltar candados si los hay. */
23350 if (nr_locks == 0) return(OK);
23351 lock_count = nr_locks;           /* guardar cuenta candados */
23352 for (flp = &file_lock[0]; flp < &file_lock[NR_LOCKS]; flp++) {
23353     if (flp->lock_type == 0) continue;           /* ranura no en uso */
23354     if (flp->lock_inode == rip && flp->lock_pid == fp->fp_pid) {
23355         flp->lock_type = 0;
23356         nr_locks--;
23357     }
23358 }
23359 if (nr_locks < lock_count) lock_revive();           /* candado liberado */
23360 return(OK);
23361 }

```

```

23364 *=====
23365 *                               do lseek                               *
23366 *=====*/
23367     PUBLIC int do_lseek()
23368     {
23369     /* Ejecutar llamada al sistema lseek(ls_fd, offset, whence). */
23370
23371     register struct filp *rfilp;
23372     register off_t pos;
23373
23374     /* Ver si el descriptor de archivo es válido. */
23375     if ((rfilp = get_filp(ls_fd)) == NIL_FILP) return(err_code);
23376
23377     /* No lseek en conductos. */
23378     if (rfilp->filp_ino->i_pipe == I_PIPE) return(ESPIPE);
23379
23380     /* El valor de 'whence' determina la posición inicial. */
23381     switch(whence) {
23382     case 0: pos = 0;           break;
23383     case 1: pos = rfilp->filp_pos;           break;
23384     case 2: pos = rfilp->filp_ino->i_size;           break;
23385     default: return(EINVAL);
23386     }
23387
23388     /* Verificar desbordamiento. */
23389     if (((long)offset > 0) && ((long)(pos + offset) < (long)pos)) return(EINVAL);
23390     if (((long)offset < 0) && ((long)(pos + offset) > (long)pos)) return(EINVAL);
23391     pos = pos + offset;
23392
23393     if (pos != rfilp->filp_pos)
23394         rfilp->filp_ino->i_seek = ISEEK;           /* inhibir lect adel. */
23395     rfilp->filp_pos = pos;
23396     reply_11 = pos;           /* insertar long en mensaje salida */
23397     return(OK);
23398 }

```

```

+++++
                                     src/fs/read.c
+++++

23400  /* Este archivo contiene el corazón del mecanismo para leer (y escribir)
23401 * archivos. Las solicitudes se dividen en trozos que no cruzan
23402 * fronteras de bloque. Cada trozo se procesa por turno. también se detectan
23403 * Y manejan lecturas de archivos especiales.
23404  *
23405 * Los puntos de entrada a este archivo son
23406 *   do_read:      ejec. llamada al sistema READ invocando read_write
23407 *   read_write:   realizar trabajo real de READ y WRITE
23408 *   read_map:    dado un nodo-i y pos en arch, buscar su núm de zona
23409 *   rd_indir:    leer entrada en un bloque de indirección
23410 *   read_ahead:  encargarse de la lectura adelantada de bloques
23411  */
23412
23413     #include "fs.h"
23414     #include <fcntl.h>
23415     #include <minix/com.h>
23416     #include "buf.h"
23417     #include "file.h"
23418     #include "fproc.h"
23419     #include "inode.h"
23420     #include "param.h"
23421     #include "super.h"
23422
23423     #define FD_MASK      077      /* máx. descrip de archivo es 63 */
23424
23425     PRIVATE message umessj      /* mensaje p/pedir a SYSTASK copia usuario */
23426
23427     FORWARD PROTOTYPE( int rw_chunk, (struct inode *rip, off_t position,
23428     unsigned off, int chunk, unsigned left, int rw_flag,
23429     char *buff, int seg, int usr)
23430     );
23431  /*=====
23432  *                                     do_read                                     *
23433  *=====*/
23434     PUBLIC int do_read()
23435     {
23436     return(read_write(READING));
23437     }
23440  /*=====
23441  *                                     read_write                                 *
23442  *=====*/
23443     PUBLIC int read_write(rw_flag)
23444     int rw_flag;      /* READING o WRITING */
23445     {
23446     /* Ejecutar llamada read(fd, buffer, nbytes) o write(fd, buffer, nbytes). */
23447
23448     register struct inode *rip;
23449     register struct filp *f;
23450     off_t bytes_left, f_size, position;
23451     unsigned int off, cum_io;
23452     int op, oflags, r, chunk, usr, seg, block_spec, char_spec;
23453     int regular, partial_pipe = 0, partial_cnt = 0;
23454     dev_t dev;

```



```

23455 mode_t mode_word;
23456 struct filp *wf;
23457
23458 /* MM carga segmentos poniendo cosas raras en 10 bits altos de 'fd'. */
23459 if (who == MM_PROC_NR && (fd & (-BYTE))) {
23460     usr = (fd >> 8) & BYTE;
23461     seg = (fd >> 6) & 03;
23462     fd &= FD_MASK; /* descartar bits de usuario y segm */
23463 } else {
23464     usr = who; /* caso normal */
23465     seg = D;
23466 }
23467
23468 /* Si descriptor de archivo válido, obtener nodo-i, tamaño y modo. */
23469 if (nbytes < 0) return(EINVAL);
23470 if ((f = get_filp(fd)) == NIL_FILP) return(err_code);
23471 if (((f->filp_mode) & (rw_flag == READING ? R_BIT : W_BIT)) == 0) {
23472     return(f->filp_mode == FILP_CLOSED ? EIO : EBADF);
23473 }
23474 if (nbytes == 0) return(0); /* así archs esp de cars no tienen q/probar 0*/
23475 position = f->filp_pos;
23476 if (position > MAX_FILE_POS) return(EINVAL);
23477 if (position + nbytes < position) return(EINVAL); /* desbordo sin signo */
23478 oflags = f->filp_flags;
23479 rip = f->filp_ino;
23480 f_size = rip->i_size;
23481 r = OK;
23482 if (rip->i_pipe == I_PIPE) {
23483     /* ffp->fp_cum_io_partial no es 0 sólo al efectuar lecturas parciales */
23484     cum_io = fp->fp_cum_io_partial;
23485 } else {
23486     cum_io = 0;
23487 }
23488 op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
23489 mode_word = rip->i_mode & I_TYPE;
23490 regular = mode_word == I_REGULAR || mode_word == I_NAMED_PIPE;
23491
23492 char_spec = (mode_word == I_CHAR_SPECIAL ? 1 : 0);
23493 block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0);
23494 if (block_spec) f_size = LONG_MAX;
23495 rdwt_err = OK; /* poner en EIO si ocurre error de disco */
23496
23497 /* Verificar si archivos especiales por caracteres. */
23498 if (char_spec) {
23499     dev = (dev_t) rip->i_zone[0];
23500     r = dev_io(op, oflags & O_NONBLOCK, dev, position, nbytes, who, buffer);
23501     if (r >= 0) {
23502         cum_io = r;
23503         position += r;
23504         r = OK;
23505     }
23506 } else {
23507     if (rw_flag == WRITING && block_spec == 0) {
23508         /* Anticipar si el archivo crecerá demasiado. */
23509         if (position > rip->i_sp->s_max_size - nbytes) return(EFBIG);
23510
23511         /* Verificar bandera O_APPEND. */
23512         if (oflags & O_APPEND) position = f_size;
23513
23514         /* Borrar la zona que contiene EOF actual si se va a crear

```

```

23515         * agujero. Esto es necesario porque todos los bloques
23516         * no escritos antes del EOF deben leerse como ceros.
23517         */
23518         if (position > f_size) clear_zone(rip, f_size, 0);
23519     }
23520
23521     /* Los conductos son un poco diferentes. Verificar. */
23522     if (rip->i_pipe == I_PIPE) {
23523         r = PiPe_CheCk(rip, rw_flag, OflagS, nbytes, position, &partial_cnt);
23524         if (r <= 0) return(r);
23525     }
23526
23527     if (partial_cnt > 0) partial_pipe = 1;
23528
23529     /* Dividir transferencia en trozos que no abarquen dos bloques. */
23530     while (nbytes != 0) {
23531         off = (unsigned int) (position % BLOCK_SIZE); /* disto en blk */
23532         if (partial_pipe) { /* sólo conductos */
23533             chunk = MIN(partial_cnt, BLOCK_SIZE - off);
23534         } else
23535             chunk = MIN(nbytes, BLOCK_SIZE - off);
23536         if (chunk < 0) chunk = BLOCK_SIZE - off;
23537
23538         if (rw_flag == READING) {
23539             bytes_left = f_size - position;
23540             if (position >= f_size) break; /* rebasamos EOF */
23541             if (chunk > bytes_left) chunk = (int) bytes_left;
23542         }
23543
23544         /* Leer o escribir 'chunk' bytes. */
23545         r = rw_chunk(rip, position, off, chunk, (unsigned) nbytes,
23546                   rw_flag, buffer, seg, usr);
23547         if (r != OK) break; /* llegamos a EOF */
23548         if (rdwt_err < 0) break;
23549
23550         /* Actualizar contadores y apuntadores. */
23551         buffer += chunk; /* dirección buffer usuario */
23552         nbytes -= chunk; /* bytes aún por leer */
23553         cum_io += chunk; /* bytes ya leídos */
23554         position += chunk; /* posición en archivo */
23555
23556         if (partial_pipe) {
23557             partial_cnt -= chunk;
23558             if (partial_cnt <= 0) break;
23559         }
23560     }
23561 }
23562
23563     /* Al escribir, actualizar el tamaño de archivo y el tiempo de acceso. */
23564     if (rw_flag == WRITING) {
23565         if (regular || mode_word == I_DIRECTORY) {
23566             if (position > f_size) rip->i_size = position;
23567         }
23568     } else {
23569         if (rip->i_pipe == I_PIPE && position >= rip->i_size) {
23570             /* Restablecer apuntadores del conducto. */
23571             rip->i_size = 0; /* ya no hay datos */
23572             position = 0; /* restab. lector(es) */
23573             if ((wf = find_filp(rip, W_BIT) != NIL_FILP) wf->filp_pos = 0;
23574         }

```

```

23575     }
23576     f->filp_pos = position;
23577
23578     /* Ver si se requiere lectura adelantada; si sí, prepararla. */
23579     if (rw_flag == READING && rip->i_seek == NO_SEEK && position % BLOCK_SIZE == 0
23580         && (regular 11 mode_word == I_DIRECTORV»    {
23581         rdahed_inode = rip;
23582         rdahedpos = position;
23583     }
23584     rip->i_seek = NO_SEEK;
23585
23586     if (rdwt_err != OK) r = rdwt_err;           /* verif. error de disco */
23587     if (rdwt_err == END_OF_FILE) r = OK;
23588     if (r == OK) {
23589         if (rw_flag == READING) rip->i_update |= ATIME;
23590         if (rw_flag == WRITING) rip->i_update |= CTIME | MTIME;
23591         rip->i_dirt = DIRTY;                    /* nodo-i ya está sucio */
23592         if (partial_pipe) {
23593             partial_pipe = 0;
23594             /* escritura parcial en conducto con */
23595             /* O_NONBLOCK, devolver cuenta de escritura */
23596             if (1 (oflags & O_NONBLOCK»    {
23597                 fp->fp_cum_io_partial = cum_io;
23598                 suspend(XPIPE); /* escrito parcial en conducto con */
23599                 return(0);      /* nbyte > PIPE_SIZE -no atómica */
23600             }
23601         }
23602         fp->fp_cum_io_partial = 0;
23603         return(cum_io);
23604     } else {
23605         return(r);
23606     }
23607 }
23610/*=====
23611 *                               rw_chunk                               *
23612 *=====*/
23613 PRIVATE int rw_chunk(rip, position, off, chunk, left, rw_flag, buff, seg, usr)
23614     register struct inode *rip; /* apunt a nodo-i de arch por leer/escr */
23615     off_t position; /* posic en arch por leer o escribir */
23616     unsigned off; /* distancia en bloque actual */
23617     int chunk; /* # bytes por leer o escribir */
23618     unsigned left; /* máx bytes deseados después de pos */
23619     int rw_flag; /* READING o WRITING */
23620     char *buff; /* dir virtual del buffer de usuario */
23621     int seg; /* segmento T o D en espacio usuario */
23622     int usr; /* cuál proceso de usuario */
23623 {
23624     /* Leer o escribir (parte de) un bloque. */
23625
23626     register struct buf *bp;
23627     register int r;
23628     int n, block_spec;
23629     block_t b;
23630     dev_t dev;
23631
23632     block_spec = (rip->i_mode & I_TVPE) == I_BLOCK_SPECIAL;
23633     if (block_spec) {
23634         b = position/BLOCK_SIZE;

```

```

23635     dev = (dev_t) rip->i_zone[0];
23636 } else {
23637     b = read_map(rip, position);
23638     ctev = rip->i_dev;
23639 }
23640
23641 if (!block_spec && b == NO_BLOCK) {
23642     if (rw_flag == READING) {
23643         /* Leer de bloque inexistente. Debe leer sólo ceros.*/
23644         bp = get_block(NO_DEV, NO_BLOCK, NORMAL).      /* obt buffer */
23645         zero_block(bp);
23646     } else {
23647         /* Escribir en bloque inexis. Crear e ingresar en nodo-i. */
23648         if ((bp= new_block(rip, position)) == NIL_BUF)return(err_code);
23649 }
23650 } else if (rw_flag == READING) {
23651     /* Leer y leer adelantado si conviene. */
23652     bp = rahead(rip, b, position, left);
23653 } else {
23654     /* Normalmente un bloque existente que se sobrescribirá
23655      * parcialmente primero se lee, pero no hay que leer un bloque lleno.
23656      * Si ya está en caché, adquirirlo; si no, adq. buffer libre.
23657 */
23658     n = (chunk == BLOCK_SIZE ? NO_READ : NORMAL);
23659     if (!block_spec && off == 0 && position >= rip->i_size) n = NO_READ;
23660     bp = get_block(dev, b, n);
23661 }
23662
23663 /* En todos los casos, bp apunta ahora a un buffer válido. */
23664 if (rw_flag == WRITING && chunk != BLOCK_SIZE && !block_spec &&
23665     position >= rip->i_size && off == 0) {
23666     zero_block(bp);
23667 }
23668 if (rw_flag == READING) {
23669     /* Copiar un trozo del buffer de bloques al espacio de usuario. */
23670     r = sys_copy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off);
23671                 usr, seg, (phys_bytes) buff,
23672                 (phys_bytes) chunk);
23673 } else {
23674     /* Copiar un trozo del espacio de usuario al buffer de bloques. */
23675     r = sys_copy(usr, seg, (phys_bytes) buff,
23676                 FS_PROC_NR, D, (phys_bytes) (bp->b_data+off);
23677                 (phys_bytes) chunk);
23678     bp->b_dirt = DIRTY;
23679 }
23680 n = (off + chunk == BLOCK_SIZE ? FULL_DATA_BLOCK : PARTIAL_DATA_BLOCK);
23681 put_block(bp, n);
23682 return(r);
23683 }
23684 /*=====
23685 *                                     read_map                                     *
23686 *=====*/
23689 PUBLIC block_t read_map(rip, position)
23690     register struct inode *rip; /* apunt a nodo del cual mapear */
23691     off_t position; /* pos en arch cuyo blq se desea */
23692 {
23693     /* Dado un nodo-i y una posición dentro del arch corresp, encontrar
23694     * el núm de bloque (no de zona) en el que está la posición y devolverlo.

```

```

23695     */
23696
23697 register struct buf *bp;
23698 register zone_t Z;
23699 int scale, boff, dzones, nr_indirects, index, zind, ex;
23700 block_t b;
23701 long excess, zone, block_pos;
23702
23703 scale = rip->i_sp->s_log_zone_size;          /* p/conversión bloque-zona */
23704 block_pos = position/BLOCK_SIZE;          /* # bloque relativo en arch */
23705 zone = block_pos » scalej                /* zona de la posición */
23706 boff = (int) (block_pos -(zone «         scale) ); /* # bloque rel. en zona */
23707 dzones = rip->i_ndzones;
23708 nr_indirects = rip->i_nindirs;
23709
23710 /* ¿Se encuentra 'position' en el nodo-i mismo? */
23711 if (zone < dzones) {
23712     zind = (int) zone;                    /* indice debe ser un int */
23713     z = rip->i_zone[zind] ;
23714     if (z == NO_ZONE) return(NO_BLOCK);
23715     b = ((block_t) z « scale) + boff;
23716     return(b);
23717 }
23718
23719 /* No está en nodo-i, así que debe ser indirec. sencilla o doble. */
23720 excess = zone -dzones;                    /* 1as. Vx_NR_DZONES no cuentan */
23721
23722 if (excess < nr_indirects) {
23723     /* 'position' puede hallarse vía bloque indirec. sencilla. */
23724     z = rip->i_zone[dzones];
23725 } else {
23726     /* 'position' puede hallarse vía bloque indirec. doble. */
23727     if ( (z = rip->i_zone[dzones+1]) == NO_ZONE) return(NO_BLOCK);
23728     excess -= nr_indirectsj                /* indir. senc. no cuenta*/
23729     b = (block_t) z « scale;
23730     bp = get_block(rip->i_dev, b, NORMAL); /* obt. bloque doble indir. */
23731     index = (int) (excess/nr_indirects);
23732     z = rd_indir(bp, index);                /* z = zona p/sencilla*/
23733     put_block(bp, INDIRECT_BLOCK);         /* liberar blq doble indir. */
23734     excess = excess % nr_indirects;        /* índice en blq indir senc. */
23735 }
23736
23737 /* 'z' es núm zona p/bloque indirec. sencillaj 'excess' es índice en él. */
23738 if (z == NO_ZONE) return(NO_BLOCK);
23739 b = (block_t) z « scale;                    /* b es # blq p/ind senc */
23740 bp = get_block(rip->i_dev, b, NORMAL);      /* obt bloque indir senc */
23741 ex = (int) excess;                          /* necesitamos un entero */
23742 z = rd_indir(bp, ex);                       /* obt blq al que apunta */
23743 put_block(bp, INDIRECT_BLOCK);              /* soltar blq indir senc */
23744 if (z == NO_ZONE) return(NO_BLOCK);
23745 b = ((block_t) z « scale) + boff;
23746 return(b);
23747 }
23750 /*=====
23751 *                                     rd_indir                                     *
23752 *=====*/
23753     PUBLIC zone_t rd_indir(bp, index)
23754     struct buf *bpj /* apunto a bloque de indirección */

```

```

23755             int index;          /* índice en *bp */
23756     {
23757         /* Dado un apuntador a un bloque de indirección, leer una entrada.
23758 * Se hace una rutina aparte porque hay cuatro casos:
23759 * V1 (1BM Y 68000) Y V2 (1BM Y 68000).
23760 */
23761
23762     struct super_block *sp;
23763     zone_t zone;          /* zonas v2 son long s (shorts en V1) */
23764
23765     sp = get_super(bp->b_dev); /* neces. superbloque p/saber tipo FS */
23766
23767     /* leer una zona de un bloque de indirección */
23768     if (sp->s_version == V1)
23769         zone = (zone_t) conv2(sp->s_native, (int) bp->b_v1_ind[index]);
23770     else
23771         zone = (zone_t) conv4(sp->s_native, (long) bp->b_v2_ind[index]);
23772
23773     if (zone != NO_ZONE &&
23774         (zone < (zone_t) sp->s_firstdatazone || zone >= sp->s_zones) {
23775         printf("Illegal zone number %ld in indirect block, index %d\n",
23776             (long) zone, index);
23777         panic("check file system", NO_NUM);
23778     }
23779     return(zone);
23780 }
23783/*=====
23784 *                               read_ahead                               *
23785*=====*/
23786     PUBLIC void read_ahead()
23787     {
23788         /* Leer un bloque y colocarlo en taché antes de que se necesite. */
23789
23790     register struct inode *rip;
23791     struct buf *bp;
23792     block_t b;
23793
23794     rip = rdahed_inode; /* apunt a nodo-i desde donde leer */
23795     rdahed_inode = NIL_INODE; /* desactivar lectura adelantada */
23796     if ((b = read_map(rip, rdahedpos) == NO_BLOCK) return; /* en EOF */
23797     bp = rahead(rip, b, rdahedpos, BLOCK_SIZE);
23798     put_block(bp, PARTIAL_DATA_BLOCK);
23799 }
23802/*=====
23803 *                               rahead                               *
23804*=====*/
23805     PUBLIC struct buf *rahead(rip, baseblock, position, bytes_ahed)
23806         register struct inode *rip; /* apunt a nodo de arch por leer */
23807         block_t baseblock; /* bloque en posición actual */
23808         off_t position; /* posición en el archivo */
23809         unsigned bytes_ahed; /* bytes después pos plusa inmediato */
23810     {
23811         /* Traer un bloque del taché o dispositivo. Si se requiere lectura física,
23812 * preobtener tantos bloques más como sea cómodo. Esto suele abarcar
23813 * bytes_ahed y es al menos BLOCKS_MINIMUM. El controlador
23814 * puede decidir que es mejor dejar de leer en una frontera de cilindro

```

```

23815     * (o después de un error). Rw_scattered() pone una bandera opcional
23816     * en todas las lecturas para permitir esto.
23817     */
23818
23819     /* Número mínimo de bloques que preobtener. */
23820     # define BLOCKS_MINIMUM    (NR_BUFS < 50 ? 18 : 32)
23821
23822     int bloCk_spec, scale, read_q_Size;
23823     unsigned int blocks_ahead, fragment;
23824     block_t bloCk, blocks_left;
23825     off_t indl_pos;
23826     dev_t dev;
23827     struct buf *bp;
23828     static struct buf *read_q[NR_BUFS];
23829
23830     blOck_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
23831     if (block_spec) {
23832         dev = (dev_t) rip->i_zone[0];
23833     } else {
23834         dev = rip->i_dev;
23835     }
23836
23837     block = baseblock;
23838     bp = get_block(dev, block, PREFETCH);
23839     if (bp->b_dev != NO_DEV) return(bp);
23840
23841     /* La mejor estimación el núm de bloques que preobtener: Muchos.
23842     * Es imposible saber cómo es el dispositivo, así que ni tratamos
23843     * de adivinar su geometría; lo dejamos al controlador.
23844     *
23845     * El controlador de disquete puede leer toda una pista sin retardo rotacional,
23846     * Y evita leer pistas parciales si puede, así que es perfecto darle suficientes
23847     * buffers para leer 2 pistas. (Dos, porque algunos tipos de disquetes tienen
23848     * núm impar de sectores por pista, así que un bloque puede abarcar pistas.)
23849     *
23850     * Los controladores de disco no tratan de ser inteligentes. Con los discos
23851     * actuales es imposible saber qué geometría tienenj es mejor leer lo más
23852     * que se pueda. Con Suerte el caché de la unidad da algo de tiempo para
23853     * iniciar la siguiente lectura.
23854     *
23855     * Esta solución es casi un hack, sólo lee bloques de la Posición actual
23856     * en el archivo esperando poder encontrar más del archivo. Una mejor
23857     * solución debe examinar los apuntadores de zona y bloques de indirección
23858     * ya disponibles (ipero no invocar read_map!).
23859     */
23860
23861     fragment = Position % BLOCK_SIZE;
23862     Position -= fragment;
23863     bytes_ahead += fragment;
23864
23865     blocks_ahead = (bytes_ahead + BLOCK_SIZE - 1) / BLOCK_SIZE;
23866
23867     if (block_spec && rip->i_size == 0) {
23868         blocks_left = NR_IOREOS;
23869     } else {
23870         blocks_left = (rip->i_size - Position + BLOCK_SIZE - 1) / BLOCK_SIZE;
23871
23872         /* Ir por el primer bloque de indirección si estamos cerca. */
23873         if (!bloCk_spec) {
23874             scale = rip->i_sP->S_log_zone_size;

```

```

23875     ind1_pos = (off_t) rip->i_ndzones * (BLOCK_SIZE << scale);
23876     if (position <= ind1_pos && rip->i_size > ind1_pos) {
23877         blocks_ahead++;
23878         blocks_left++;
23879     }
23880 }
23881 }
23882
23883 /* No más que la máxima solicitud. */
23884 if (blocks_ahead > NR_IOREQS) blocks_ahead = NR_IOREQS;
23885
23886 /* Leer al menos el núm mínimo de bloques, pero no después de búsqueda. */
23887 if (blocks_ahead < BLOCKS_MINIMUM && rip->i_seek == NO SEEK)
23888     blocks_ahead = BLOCKS_MINIMUM;
23889
23890 /* No se puede rebasar el fin de archivo. */
23891 if (blocks_ahead > blocks_left) blocks_ahead = blocks_left;
23892
23893 read_q_size = 0;
23894
23895 /* Adquirir buffers de bloques. */
23896 for (j;) {
23897     read_q[read_q_size++] = bp;
23898
23899     if (--blocks_ahead == 0) break;
23900
23901     /* No arruinar el caché, dejar 4 libres. */
23902     if (bufs_in_use >= NR_BUFS -4) break;
23903
23904     block++;
23905
23906     bp = get_block(dev, block, PREFETCH);
23907     if (bp->b_dev != NO_DEV) {
23908         /* ¡Epa! Bloque ya en caché, salirse. */
23909         put_block(bp, FULL_DATA_BLOCK);
23910         break;
23911     }
23912 }
23913 rw_scattered(dev, read_q, read_q_size, READING);
23914 return(get_block(dev, baseblock, NORMAL));
23915 }

```

+++++

src/fs/write.c

```

+++++
24000 /* Este archivo es la contraparte de "read.c". Contiene el código
24001 * para escribir que no está contenido en read_write().
24002 *
24003 * Los puntos de entrada a este archivo son
24004 * do_write:   invocar read_write p/ejec llamada al sistema WRITE
24005 * clear_zone: borrar una zona en media de un archivo
24006 * new_block:  adquirir un nuevo bloque
24007 */
24008
24009 #include "fs.h"

```



```

24010     #include <string.h>
24011     #include "buf.h"
24012     #include "file.h"
24013     #include "fproc.h"
24014     #include "inode.h"
24015     #include "super.h" 24016
24017     FORWARD_PROTOTYPE( int write_map, (struct inode *rip, off_t position,
24018                                     zone_t new_zone) );
24019
24020     FORWARD_PROTOTYPE( void wr_indir, (struct buf *bp, int index, zone_t zone) ); 24021
24022 /*=====
24023 *                                     do_write                                     *
24024 *=====*/
24025     PUBLIC int do_write()
24026     {
24027     /* Ejecutar la llamada al sistema write(fd, buffer, nbytes). */
24028
24029     return(read_write(WRITING»);
24030     }
24033 /*=====
24034 *                                     write_map                                     *
24035 *=====*/
24036 PRIVATE int write_map(rip, position, new_zone)
24037     register struct inode *rip; /* apunt a nodo-i por modificar */
24038     off_t positionj /* dirección archivo por mapear */
24039     zone_t new_zone; /* núm zona por insertar */
24040     {
24041     /* Escribir una nueva zona en un nodo-i. */
24042     int scale, ind_ex, new_ind, new_dbl, zones, nr_indirects, single, zindex, ex;
24043     zone_t z, z1 j
24044     register block_t b;
24045     long excess, zone;
24046     struct buf *bp;
24047
24048     rip->i_dirt = DIRTY; /* el nodo-i se modificará */
24049     bp = NIL_BUF;
24050     scale = rip->i_sp->s_log_zone_size; /* p/conversión zona-bloque */
24051     zone = (position/BLOCK_SIZE) » scalej /* # zona rel por insertar */
24052     zones = rip->i_ndzones; /* # zonas directas en el nodo-i */
24053     nr_indirects = rip->i_nindirs; /* # zonas indirectas p/blq indir */
24054
24055     /* ¿Se encuentra 'position' en el nodo-i mismo? */
24056     if (zone < zones) {
24057         zindex = (int) zone; /* necesitamos un entero aquí */
24058         rip->i_zone[zindex] = new_zone;
24059         return(OK);
24060     }
24061
24062     /* No está en el nodo-i; debe ser indirec sencilla o doble. */
24063     excess = zone - zones; /* 1 as Vx_NR_DZONES no cuentan */
24064     new_ind = FALSE;
24065     new_dbl = FALSE;
24066
24067     if (excess < nr_indirects) {
24068         /* 'position' puede encontrarse vía bloque indirección sencilla. */
24069         z1 = rip->i_zone[zones]; /* zona indirec sencilla */

```

```

24070     single = TRUE;
24071 } else {
24072     /* 'position' puede encontrarse vía bloque doble indirección. */
24073     if ( (z = rip->i_zone[zones+1]) == NO_ZONE ) {
24074         /* Crear el bloque de doble indirección. */
24075         if ( (z = alloc_zone(rip->i_dev, rip->i_zone[0]) == NO_ZONE)
24076             return(err_code);
24077         rip->i_zone[zones+1] = Z;
24078         new_dbl = TRUE; /* izar bandera p/después */
24079     }
24080
24081     /* Como sea, 'z' es núm zona p/bloque doble indirección. */
24082     excess -= nr_indirectS; /* indirec sencilla no cuenta */
24083     ind_ex = (int) (excess / nr_indirects);
24084     excess = excess % nr_indirectS;
24085     if (ind_ex >= nr_indirects) return(EFBIG);
24086     b = (block_t) z << scale;
24087     bp = get_block(rip->i_dev, b, (new_dbl ? NO_READ : NORMAL));
24088     if (new_dbl) zero_block(bp);
24089     z1 = rd_indir(bp, ind_ex);
24090     single = FALSE;
24091 }
24092
24093 /* z1 ahora es zona indirec sencillaj 'excess' es el índice. */
24094 if (z1 == NO_ZONE) {
24095     /* Crear bloque indirec y guardar # zona en nodo-i o blq doble indir. */
24096     z1 = alloc_zone(rip->i_dev, rip->i_zone[0]);
24097     if (single)
24098         rip->i_zone[zones] = z1; /* actualizar nodo-i */
24099     else
24100         wr_indir(bp, ind_ex, z1); /* actualizar doble indir */
24101
24102     new_ind = TRUE;
24103     if (bP != NIL_BUF) bp->b_dirt = DIRTY; /* si doble indir, sucio */
24104     if (z1 == NO_ZONE) {
24105         put_block(bp, INDIRECT_BLOCK); /* liberar blq doble ind */
24106         return(err_code); /* no pudo crear ind senc */
24107     }
24108 }
24109 put_block(bp, INDIRECT_BLOCK); /* liberar bloque doble ind */
24110
24111 /* z1 es el número de zona del bloque de indirección. */
24112 b = (block_t) z1 << scale;
24113 bp = get_block(rip->i_dev, b, (new_ind ? NO_READ : NORMAL));
24114 if (new_ind) zero_block(bp);
24115 ex = (int) excess; /* necesitamos un int aquí */
24116 wr_indir(bp, ex, new_zone);
24117 bp->b_dirt = DIRTY;
24118 put_block(bp, INDIRECT_BLOCK);
24119
24120     return(OK);
24121 }
24124 /*=====
24125 *                               wr indir                               *
24126 *=====*/
24127 PRIVATE void wr_indir(bp, index, zone)
24128     struct buf *bp; /* apuntador a bloque indirección */
24129     int index; /* índice a *bp */

```

```

24130                                     zone_t zone;    /* zona por escribir */
24131     {
24132         /* Dado un apuntador a un bloque de indirección, escribir una entrada. */
24133
24134     struct super_block *sp;
24135
24136     sp = get~super(bp->b_dev);          /* nec superbloque p/hallar tipo de FS */
24137
24138     /* escribir una zona en un bloque de indirección */
24139     if (sp->s_version == V1)
24140         bp->b_v1_ind[index] = (zone_t) conv2(sp->s_native, (int) zone);
24141     else
24142         bp->b_v2_ind[index] = (zone_t) conv4(sp->s_native, (long) zone);
24143     }
24144 /*=====
24145 *                                     clear zone                                     *
24146 *=====
24147
24148     PUBLIC void clear_zone(rip, pos, flag)
24149         register struct inode *rip; /* nodo-i por borrar */
24150         off_t pos;                /* apunta a bloque por borrar */
24151         int flag;                 /* 0 si invoc p/read_write, 1 si new_block */
24152     {
24153         /* Poner en ceros una zona, quizá comenzando en la mitad. El parámetro 'pos'
24154          * da un byte en el 1er bloque por borrar. Clearzone() se invoca desde
24155          * read_write y new_block().
24156          */
24157
24158     register struct buf *bp;
24159     register block_t b, blo, bhi;
24160     register off_t next;
24161     register int scale;
24162     register zone_t zone_size;
24163
24164     /* Si tamaño de bloque y zona iguales, no necesitamos cle'ar_zone(). */
24165     scale = rip->i_sp->s_log_zone_size;
24166     if (scale == 0) return;
24167
24168     zone_size = (zone_t) BLOCK_SIZE << scale;
24169     if (flag == 1) pos = (pos/zone_size) * zone_size;
24170     next = pos + BLOCK_SIZE - 1;
24171
24172     /* Si 'pos' está en el último bloque de una zona, no borrar la zona. */
24173     if (next/zone_size != pos/zone_size) return;
24174     if ( (blo = read_map(rip, next) == NO_BLOCK) return;
24175     bhi = ( ((blo»scale)+1)« scale) - 1;
24176
24177     /* Borrar todos los bloques entre 'blo' y 'bhi'. */
24178     for (b = blo; b <= bhi; b++) {
24179         bp = get_block(rip->i_dev, b, NO_READ);
24180         zero_block(bp);
24181         put_block(bp, FULL_DATA_BLOCK);
24182     }
24183 }
24184 /*=====
24185 *                                     new_block                                     *
24186 *=====

```

```

24190 PUBLIC struct buf *new_block(rip, position)
24191 register struct inode *rip;          /* apuntador a nodo-i */
24192 off_t position;                      /* apuntador de archivo */
24193 {
24194     /* Adquirir bloque nuevo y devolver apuntador a él. Esto puede requerir
24195     * asignar una zona completa y después devolver el bloque inicial.
24196     * Por otro lado, la zona actual podría tener todavía bloques no utilizados.
24197     */
24198
24199     register struct buf *bp;
24200     block_t b, base_block;
24201     zone_t z;
24202     zone_t zone_size;
24203     int scale, r;
24204     struct super_block *sp;
24205
24206     /* ¿Está disponible otro bloque en la zona actual? */
24207     if ( (b = read_map(rip, position)) == NO_BLOCK ) {
24208         /* Escoger la primera zona si es posible. */
24209         /* Perder si el archivo no está vacío pero el primer número de zona
24210         * es NO_ZONE, que corresponde a una zona llena de ceros. Sería
24211         * mejor buscar cerca de la última zona real.
24212         */
24213         if (rip->i_zone[0] == NO_ZONE) {
24214             sp = rip->i_sp;
24215             z = sp->s_firstdatazone;
24216         } else {
24217             z = rip->i_zone[0];          /* buscar cerca de 1ra zona */
24218         }
24219         if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE ) return(NIL_BUF);
24220         if ( (r = write_map(rip, position, z)) != OK ) {
24221             free_zone(rip->i_dev, z);
24222             err_code = r;
24223             return(NIL_BUF);
24224         }
24225
24226         /* Si no escribimos en EOF, borrar zona, por seguridad. */
24227         if ( position != rip->i_size ) clear_zone(rip, position, 1);
24228         scale = rip->i_sp->S_log_zone_size;
24229         base_block = (block_t) z << scale;
24230         zone_size = (zone_t) BLOCK_SIZE << scale;
24231         b = base_block + (block_t) ((position % zone_size)/BLOCK_SIZE);
24232     }
24233
24234     bp = get_block(rip->i_dev, b, NO_READ);
24235     zero_block(bp);
24236     return(bp);
24237
24238     bp->b_dirt = DIRTY;7 }
24239
24240 /*=====
24241 *                               zero_block                               *
24242 *=====*/
24243 PUBLIC void zero_block(bp)
24244 register struct buf *bp;          /* apunt a buffer por borrar */
24245 {
24246     /* Poner en ceros un bloque. */
24247
24248     memset(bp->b_data, 0, BLOCK_SIZE);

```

```

24250 }
+++++
                                     src/fs/pipe.c
+++++
24300  /* Este archivo se ocupa de suspender y revivir procesos. Un proceso
24301  * puede suspenderse porque quiere leer de o escribir en un conducto
24302  * Y no puede, o porque quiere leer de o escribir en un archivo especial
24303  * Y no puede. Cuando un proceso no puede continuar se suspende,
24304  * Y se revive después cuando ya puede continuar.
24305  *
24306  * Los puntos de entrada a este archivo son
24307  *   do_pipe:     ejecutar llamada al sistema PIPE
24308  *   pipe_check:  ver si es factible leer o escribir en conducto
24309  *   suspend:     suspender proc que no puede efectuar lectura o escritura
24310  *   release:     ver si un proc suspendido puede liberarse y hacerlo
24311  *   revive:      marcar un proceso suspendido como ejecutable
24312  *   do_unpause:  se envió 1 señal a un procesoj ver si está suspendido
24313  */
24314
24315     #include "fs.h"
24316     #include <fcntl.h>
24317     #include <signal.h>
24318     #include <minix/boot.h>
24319     #include <minix/callnr.h>
24320     #include <minix/com.h>
24321     #include "dev.h"
24322     #include "file.h"
24323     #include "fproc.h"
24324     #include "inode.h"
24325     #include "param.h"
24326
24327     PAIVATE message mess;
24328
24329 /*=====
24330  *                                     do_pipe                                     *
24331  *=====*/
24332     PUBLIC int do_pipe()
24333     {
24334         /* Ejecutar la llamada al sistema pipe(fil_des). */
24335
24336         register struct fproc *rfp;
24337         register struct inode *rip;
24338         int      ij;
24339         struct filp *fil_ptr0, *fil_ptr1;
24340         int fil_des[2]j                /* aquí va la respuesta */
24341
24342         /* Adquirir dos descriptores de archivo. */
24343         rfp = fpj
24344         if ( (r = get_fd(0, A_BIT, &fil_des[0]                , &fil_ptr0»    1= OK) return(r);
24345         rfp->fp_filp[fil_des[0]] = fil_ptr0;
24346         fil_ptr0->filp_count = 1;
24347         if ( (r = get_fd(0, W_BIT, &fil_des[1], &fil_ptr1»    1= OK) {
24348             rfp->fp_filp[fil_des[0]] = NIL_FILP;
24349             fil_ptr0->filp_count = 0;

```



```

24410     suspend(XPIPE); /* bloquear lector */
24411
24412     /* Si necesario, activar escritores dormidos. */
24413     if (susp_count > 0) release(rip, WRITE, susp_count);
24414     }
24415     return(r);
24416     }
24417 } else {
24418     /* El proceso está escribiendo en un archivo. */
24419     /*     if (bytes > PIPE_SIZE) return(EFBIG); */
24420     if (find_filp(rip, R_BIT) == NIL_FILP) {
24421         /* Decir a kernel que genere señal SIGPIPE. */
24422         sys_kill((int)(fp - fproc), SIGPIPE);
24423         return(EPIPE);
24424     }
24425
24426     if (position + bytes > PIPE_SIZE) {
24427         if ((oflags & O_NONBLOCK) && bytes < PIPE_SIZE)
24428             return(EAGAIN);
24429         else if ((oflags & O_NONBLOCK) && bytes > PIPE_SIZE) {
24430             if ( (*canwrite = (PIPE_SIZE - position)) > 0) {
24431                 /* Realizar escritura parcial, hay que despertar al lector. */
24432                 release(rip, READ, susp_count);
24433                 return(1);
24434             } else {
24435                 return(EAGAIN);
24436             }
24437         }
24438         if (bytes > PIPE_SIZE) {
24439             if ((*canwrite = PIPE_SIZE - position) > 0) {
24440                 /* Realizar escritura parcial, hay que despertar
24441                  * al lector, ya que nos suspenderemos en read~.vrite()
24442                  */
24443                 release(rip, READ, susp_count);
24444                 return(1);
24445             }
24446         }
24447         suspend(XPIPE); /* parar escritor --conducto lleno */
24448         return(0);
24449     }
24450
24451     /* Se escribe en conducto vacío. Buscar lector suspendido. */
24452     if (position == 0) release(rip, READ, susp_count);
24453 }
24454
24455 *canwrite ~ 0;
24456     return(1);
24457 }
24460/*=====
24461 *                               suspend                               *
24462*=====
24463 PUBLIC void suspend(task)
24464 int task;                               /* ¿a quién espera proc? (PIPE = conducto) */
24465 {
24466     /* Tomar medidas p/suspender el procesam. de la llamada actual.
24467     * Guardar en tabla proc los paráms que se usarán al reanudar.
24468     * (En realidad, no se usan cuando un proceso está esperando un disp
24469     * de E/S, pero sí para conductos, y no vale la pena hacer la distinción.)

```

```

24470     */
24471
24472 if (task == XPIPE 11 task == XPOPEN) susp_count++j/* # procs susp en cond*/
24473 fp->fp_suspended = SUSPENDED;
24474 fp->fp_fd = fd <<      8 I fs_call;
24475 fp->fp_task = -task;
24476 if (task == XLOCK) {
24477     fp->fp_buffer = (char *) name1; /*          3er. arg de fcntl() */
24478     fp->fp_nbytes = requestj          /* 20. arg de fcntl() */
24479 } else {
24480     fp->fp_buffer = buffer;          /* p/lecturas y escrituras */
24481     fp->fp_nbytes = nbytes;
24482 }
24483 dont_reply = TRUE;                /* no enviar mensaje respuesta ahora */
24484 }
24487/*=====
24488 *                                re leas e                                *
24489*=====*/
24490     PUBLIC void release(ip, call_nr, count)
24491         register struct inode *ip; /* nodo-i del conducto */
24492         int call_nr;              /* READ, WRITE, OPEN o CREAT */
24493         int count;                /* núm máx de procs que liberar */
24494     {
24495     /* Ver si cualquier proceso está esperando el conducto cuyo nodo-i está en 'ip'.
24496 * Si hay uno, y estaba tratando de ejecutar la llamada indicada por 'call_nr',
24497 * liberarlo.
24498     */
24499
24500     register struct fproc *rp;
24501
24502     /* Buscar en la tabla de procesos. */
24503     for (rp = &fproc[0]; rp < &fproc[NR_PROCS] j rp++) {
24504         if (rp->fp_suspended == SUSPENDED &&
24505             rp->fp_revived == NOT_REVIVING &&
24506             (rp->fp_fd & BYTE) == call_nr &&
24507             rp->fp_filp[rp->fp_fd>8] ->filp_ino == ip) {
24508             revive((int) (rp -fproc), 0);
24509             susp_count--; /* saber quién está suspendido */
24510             if (--count == 0) return;
24511         }
24512     }
24513 }
24516/*=====
24517 *                                revive                                *
24518*=====*/
24519 PUBLIC void revive(proc_nr, bytes)
24520     int proc_nr;                  /* proceso por revivir */
24521     int bytes;                    /* si espera tarea, cuántos bytes leyó */
24522     {
24523     /* Revivir un proceso que estaba bloqueado. Si un proceso está esperando una tty,
24524 * ésta es la forma en que finalmente se libera.
24525     */
24526
24527     register struct fproc *rfp;
24528     register int task;
24529

```



```

24530 if (proc_nr < 0 || proc_nr >= NR_PROCS) panic("revive err", proc_nr);
24531 rfp = &fproc[proc_nr];
24532 if (rfp->fp_suspended == NOT_SUSPENDED || rfp->fp_revived == REVIVING) return;
24533
24534 /* La bandera 'reviving' sólo aplica a conductos. Los procesos esperando
24535    * TTY reciben un mensaje de inmediato. El proceso de revivir es diferente
24536    * para TTY y conductos. Para TTY, el trabajo ya se hizo; para conductos
24537    * no: el proceso debe reiniciarse para que pueda volver a intentarlo.
24538    */
24539 task = -rfp->fp_task;
24540 if (task == XPIPE || task == XLOCK) {
24541     /* Revivir proceso suspendido por conducto o candado. */
24542     rfp->fp_revived = REVIVING;
24543     reviving++; /* el proc esperaba un conducto o candado */
24544 } else {
24545     rfp->fp_suspended = NOT_SUSPENDED;
24546     if (task == XOPEN) /* proc bloqueado en open o creat */
24547         reply(proc_nr, rfp->fp_fd»8);
24548     else {
24549         /* Revivir proceso suspendido por TTY u otro dispositivo. */
24550         rfp->fp_nbytes = bytes; /* fingir q/quiere sólo lo q/hay */
24551         reply(proc_nr, bytes); /* desbloquear proceso */
24552     }
24553 }
24554 }
24557/*=====
24558 *                               do_unpause                               *
24559*=====*/
24560 PUBLIC int do_unpause()
24561 {
24562     /* Se envió una señal a un proceso que está en pausa en el sistema
24563     * de archivos. Abortar llamada al sistema con el mensaje de error EINTR.
24564     */
24565
24566     register struct fproc *rfp;
24567     int proc_nr, task, fild;
24568     struct filp *f;
24569     dev_t dev;
24570
24571     if (who > MM_PROC_NR) return(EPERM);
24572     proc_nr = pro;
24573     if (proc_nr < 0 || proc_nr >= NR_PROCS) panic("unpause err 1", proc_nr);
24574     rfp = &fproc[proc_nr];
24575     if (rfp->fp_suspended == NOT_SUSPENDED) return(OK);
24576     task = -rfp->fp_task;
24577
24578     switch(task) {
24579         case XPIPE: /* proc trata de leer o escribir conducto */
24580             break;
24581
24582         case XOPEN: /* proc trata de abrir archivo especial */
24583             panic("fs/do_unpause called with XOPEN\n", NO_NUM);
24584
24585         case XLOCK: /* proc trata de poner candado con FCNTL */
24586             break;
24587
24588         case XOPEN: /* proc trata de abrir un fifo */
24589             break;

```

```

24590
24591     default:                /* proc trata hacer E/S disp (p.ej. tty) */
24592         fild = (rfp->fp_fd » 8) & 8YTEj /* extraer desc archivo */
24593         if (fild < 0 11 fild >= OPEN_MAX) panic("unpause err 2",NO_NUM);
24594         f = rfp->fp_filp[fild];
24595         dev = (dev_t) f->filp_ino->i_zone[0];          /* disp esperado */
24596         mess.TTY_LINE = (dev » MINOR) & BYTE;
24597         mess.PROC_NR = proc_nr;
24598
24599         /* Decir a kernel R o W. Modo es de llamada actual, no open. */
24600         mess.COUNT = (rfp->fp_fd & BYTE) == READ ? R_BIT : W_BIT;
24601         mess.m_type = CANCEL;
24602         fp = rfp;                /* hack .call_ctty usa fp */
24603         (*dmap[(dev » MAJOR) & BYTE].dmap_rw)(task, &mess);
24604     }
24605
24606     rfp->fp_suspended = NOT_SUSPENDED;
24607     reply(proc_nr, EINTR);      /* señal que interrumpió llamada */
24608     return(OK);
24609 }

```

+++++

src/fs/path.c

+++++

```

24700     /* Este archivo contiene los procedimientos que buscan nombres de ruta en el sist
24701     * de directorios y determinan el núm nodo.i correspondiente.
24702     *
24703     *     Los puntos de entrada a este archivo son
24704     *     eat_path:  rutina 'main' del mecan de conversión ruta-a-nodo-i
24705     *     last_dir: encontrar directorio final de una ruta dada
24706     *     advance:  analizar un componente de un nombre de ruta
24707     *     search_dir: buscar cadena en directorio y devolver su núm nodo-i
24708     */
24709
24710 #include "fs.h"
24711 #include <string.h>
24712 #include <minix/callnr.h>
24713 #include "buf.h"
24714 #include "file.h"
24715 #include "fproc.h"
24716 #include "inode.h"
24717 #include "super.h"
24718
24719 PUBLIC char dot1[2] = ". ";          /* pique search_dir se salte */
24720 PUBLIC char dot2[3] = ". . ";      /* permisos de acceso para. y.. */
24721
24722 FORWARD _PROTOTYPE( char *get_name, (char *old_name, char string [NAME_MAX]) );
24723
24724 /*=====
24725     *                               eat_path                               *
24726 *===== */
24727 PUBLIC struct inode *eat_path(path)
24728 char *path;                /* nombre de ruta por analizar */
24729 {

```

```

24730      /* Analizar ruta 'path' y colocar su nodo-i en tabla. Si imposible, devolver
24731 * NIL_INODE como valor de la función y un cód. error en 'err_code'.
24732 */
24733
24734 register struct inode *ldip, *rip;
24735 char string[NAME_MAX];          /* poner 1 nombre comp de ruta aquí */
24736 ,
24737 /* Primero abrir la ruta hasta el directorio final. */
24738 if ((ldip = last_dir(path, string)) == NIL_INODE)
24739     return(NIL_INODE);          /* imposible abrir dir final */
24740
24741 /* La ruta que sólo es "1" es un caso especial, detectarlo. */
24742 if (string[0] == '\0') return(ldip);
24743
24744 /* Obtener componente final de la ruta. */
24745 rip = advance(ldip, string);
24746     put_inode(ldip);
24747     return(rip);
24748     }
24751 *=====
24752 *                               last_dir                               *
24753 *=====*/
24754     PUBLIC struct inode *last_dir(path, string)
24755         char *path;          /* nombre ruta por analizar */
24756 char string[NAME_MAX];      /* componente final se devuelve aquí */
24757 {
24758     /* Dada una ruta 'path' en el espacio de direcciones fs,
24759 * analizarla hasta el último directorio, traer el nodo-i de ese dir
24760 * a la tabla de nodos-i y devolver un apuntador al nodo.
24761 * Además, devolver el componente final de la ruta en 'string'.
24762 * Si no es posible abrir el último directorio, devolver NIL_INODE,
24763 * Y dar la razón del fracaso en 'err_code'.
24764 */
24765
24766 register struct inode *rip;
24767 register char *new_name;
24768 register struct inode *new_ip;
24769
24770 /* ¿Ruta absoluta o relativa? Inicializar 'rip' según esto. */
24771 rip = (*path == '1' ? fp->fp_rootdir : fp->fp_workdir);
24772
24773 /* Si el dir se eliminó o la ruta está vacía, devolver ENOENT. */
24774 if (rip->i_nlinks == 0 || *path == '\0') {
24775     err_code = ENOENT;
24776     return(NIL_INODE);
24777 }
24778
24779 dup_inode(rip);                /* se devolverá nodo-i con put_inode */
24780
24781 /* Examinar la ruta componente por componente. */
24782 while (TRUE) {
24783     /* Extraer un componente. */
24784     if ((new_name = get_name(path, string)) == (char*) 0) {
24785         put_inode(rip); /* ruta errónea en esp usuario */
24786         return(NIL_INODE);
24787     }
24788     if (*new_name == '\0')
24789         if ((rip->i_mode & I_TVPE) == I_DIRECTORY)

```

```

24790         return(rip);          /* salida normal */
24791     else {
24792         /* último arch de prefijo de ruta no es directorio */
24793         put_inode(rip);
24794         err_code = ENOTDIRj
24795         return(NIL_INODE)i
24796     }
24797
24798     /* Hay más ruta. Seguir analizando. */
24799     new_ip = advanCe(rip, string);
24800     put=inode(rip);          /* rip obsoleto o no aplica */
24801     if (new_ip == NIL_INODE) return(NIL_INODE) ;
24802
24803     /* Éxito en llamada advance(). Traer siguiente componente. */
24804     path = new_name;
24805     rip = new_ip;
24806 }
24807 }
24810 /*=====
24811 *                               get_name                               *
24812 *=====*/
24813 PRIVATE char *get_name(old_name, string)
24814 char *old_name;          /* nombre de ruta p/analizar */
24815     char string[NAME_MAX]; /* componente extraído de 'old_name' */
24816 {
24817     /* Dado un apuntador a un nombre de ruta en espacio fs, 'old_name',
24818     * copiar siguiente componente en 'string' y rellenar con ceros. Se devuelve
24819     * un apuntador a la parte del nombre aún no analizada. Aproximadamente,
24820     * 'get_name' = 'old_name' - 'string'.
24821     *
24822     * Esta rutina sigue la convención estándar de que /usr/ast, /usr//ast,
24823     * //usr//ast y /usr/ast/ son equivalentes.
24824     */
24825
24826     register int C;
24827     register char *np, *rnp;
24828
24829     np = string;          /* 'np' apunta a posic actual */
24830     rnp = old_name;      /* 'rnp' apunta a cadena no analiz */
24831     while ( (c = *rnp) == '/') rnp++;          /* saltar diagonales inic */
24832
24833     /* Copiar la ruta no analizada, 'old_name' en arreglo 'string'. */
24834     while ( rnp < &old_name[PATH_MAX] && c != '/' && c != '\0' ) {
24835         if (np < &string[NAME_MAX]) *np++ = C;
24836         c = *++rnp;          /* avanzar a siguiente carácter */
24837     }
24838
24839     /* P/hacer /usr/ast/ equivalente a /usr/ast, saltar diagonales finales. */
24840     while (c == '/' && rnp < &old_name[PATH_MAX]) c = *++rnp;
24841
24842     if (np < &string[NAME_MAX]) *np = '\0' ;          /* Terminar cadena */
24843
24844     if (rnp >= &old_name[PATH_MAX]) {
24845         err_code = ENAMETOOLONG;
24846         return((char *) 0);
24847     }
24848     return(rnp);
24849 }

```

```

24852 /*=====
24853 *                               advance                               *
24854 /*=====
24855     PUBLIC struct inode *advance(dirp, string)
24856     struct inode *dirp;          /* nodo-i del dir para buscar */
24857     char string[NAME_MAX];      /* componente que
buscar */
24858 {
24859     /* Dado un directorio y un componente de una ruta, buscar el componente
24860 * en el directorio, encontrar el nodo-i, abrirlo y devolver un apuntador
24861 * a su ranura de nodo-i. Si no se puede, devolver NIL_INODE.
24862 */
24863
24864 register struct inode *rip;
24865 struct inode *rip2;
24866 register struct super_block *sp;
24867 int r, inumb;
24868 dev_t mnt_dev;
24869 ino_t numb;
24870
24871 /* Si 'string' está vacía, entregar el mismo nodo-i de inmediato. */
24872 if (string[0] == '\0') return(get_inode(dirp->i_dev, (int) dirp->i_num));
24873
24874 /* Detectar NIL_INODE. */
24875 if (dirp == NIL_INODE) return(NIL_INODE);
24876
24877 /* Si 'string' no está en el directorio, indicar error. */
24878 if ((r = search_dir(dirp, string, &numb, LOOK_UP)) != OK) {
24879     err_code: r;
24880     return(NIL_INODE);
24881 }
24882
24883 /* No rebasar directorio raíz actual, a menos que la cadena sea dot2. */
24884 if (dirp == fp->fp_rootdir && strcmp(string, ".") == 0 && string != dot2)
24885     return(get_inode(dirp->i_dev, (int) dirp->i_num));
24886
24887 /* Se halló el componente en el directorio. Obtener nodo-i. */
24888 if ((rip = get_inode(dirp->i_dev, (int) numb)) == NIL_INODE)
24889     return(NIL_INODE);
24890
24891 if (rip->i_num == ROOT_INODE)
24892     if (dirp->i_num == ROOT_INODE) {
24893         if (string[1] == '.') {
24894             for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++){
24895                 if (sp->s_dev == rip->i_dev) {
24896                     /* Liberar nodo-i raíz. Sustituir por el nodo
24897 * en que está montado.
24898 */
24899                     put_inode(rip);
24900                     mnt_dev = sp->s_imount->i_dev;
24901                     inumb = (int) sp->s_imount->i_num;
24902                     rip2 = get_inode(mnt_dev, inumb);
24903                     rip = advance(rip2, string);
24904                     put_inode(rip2);
24905                     break;
24906                 }
24907             }
24908         }
24909     }

```

```

24910 if (rip == NIL_INODE) return(NIL_INODE);
24911
24912 /* Ver si el nodo-i tiene un sist arch montado. Si sí, cambiar al dir
24913 * raíz de ese sistema. El superbloque es el enlace entre el nodo-i de
24914 * montura y el directorio raíz del sistema de archivos montado.
24915 */
24916 while (rip != NIL_INODE && rip->i_mount == I_MOUNT) {
24917     /* Sí hay un sistema de archivos montado en el nodo-i. */
24918     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
24919         if (sp->s_imount == rip) {
24920             /* Liberar el nodo-i. Sustituir por el nodo-i raíz
24921              * del dispositivo montado.
24922              */
24923             put_inode(rip);
24924             rip = get_inode(sp->s_dev, ROOT_INODE);
24925             break;
24926         }
24927     }
24928 }
24929 return(rip); /* devolver apuntador a componente de nodo-i */
24930 }
24931 /*=====
24932 *
24933 *
24934 *
24935 *=====
24936     PUBLIC int search_dir(ldir_ptr, string, numb, flag)
24937 register struct inode *ldir_ptr; /* apunt a nodo del dir p/buscar */
24938     char string[NAME_MAX]; /* componente que
24939 buscar */
24940     ino_t *numbj /* apunt a número de nodo-i */
24941     int flag; /* LOOK_UP, ENTER, DELETE o
IS_EMPTY */
24942 {
24943     /* Esta función busca en el dir a cuyo nodo-i 'ldip' apunta:
24944     * si (flag == ENTER) ingresar 'string' en el dir con # nodo-i '*numbj';
24945     * si (flag == DELETE) eliminar 'string' del directorio;
24946     * si (flag == LOOK_UP) buscar 'string' y devolver # nodo-i en 'numbj';
24947     * si (flag == IS_EMPTY) devolver OK si sólo. y ..en dirj si no, ENOTEMPTY;
24948     * si 'string' es dot1 o dot2, no se verifican permisos de acceso.
24949     */
24950
24951     register struct direct *dp;
24952     register struct buf *bp;
24953     int i, r, e_hit, t, match;
24954     mode_t bits;
24955     off_t pOS;
24956     unsigned new_slots, old_slots;
24957     block_t b;
24958     struct super_block *sp;
24959     int extended = 0;
24960
24961     /* Si 'ldir_ptr' no es un apuntador a nodo-i de dir, error. */
24962     if ((ldir_ptr->i_mode & I_TYPE) != I_DIRECTORY) return(ENOTDIR);
24963
24964     r = OK;
24965
24966     if (flag != IS_EMPTY) {
24967         bits = (flag == LOOK_UP ? X_BIT: W_BIT | X_BIT);
24968
24969         if (string == dot1 || string == dot2) {

```

```

24970         if (flag != LOOK_UP) r = read_only(ldir_ptr);
24971         /* sólo se requiere un dispositivo escribible. */
24972     }
24973     else r = forbidden(ldir_ptr, bits); /* verific. permisos acceso */
24974     }
24975     if (r != OK) return(r);
24976
24977     /* Recorrer el directorio bloque por bloque. */
24978     old_slots = (unsigned) (ldir_ptr->i_size/DIR_ENTRY_SIZE);
24979     new_slots = 0;
24980     e_hit = FALSE;
24981     match = 0;                                /* 1 si concuerda la cadena */
24982
24983     for (pos = 0; pos < ldir_ptr->i_size; pos += BLOCK_SIZE) {
24984         b = read_map(ldir_ptr, pos);           /* obtener núm bloque */
24985
24986         /* Directorios no tienen agujeros, así que 'b' no puede ser NO_BLOCK. */
24987         bp = get_block(ldir_ptr->i_dev, b, NORMAL); /* obt bloque dir */
24988
24989         /* Buscar en un bloque de directorio. */
24990         for (dp = &bp->b_dir[0]; dp < &bp->b_dir[NR_DIR_ENTRIES] j dp++) {
24991             if (++new_slots > old_slots) { /* no hallado, pero cabe */
24992                 if (flag == ENTER) e_hit = TRUE;
24993                 break;
24994             }
24995
24996             /* Hay concordancia si se encuentra la cadena. */
24997             if (flag != ENTER && dp->d_ino != 0) {
24998                 if (flag == IS_EMPTY) {
24999                     /* Si pasa la prueba, dir no vacío. */
25000                     if (strcmp(dp->d_name, ".") != 0 &&
25001                         strcmp(dp->d_name, "..") != 0) match = 1;
25002                 } else {
25003                     if (strncmp(dp->d_name, string, NAME_MAX) == 0)
25004                         match = 1;
25005                 }
25006             }
25007
25008             if (match) {
25009                 /* LOOK_UP o DELETE halló lo que buscaba. */
25010                 r = OK;
25011                 if (flag == IS_EMPTY) r = ENOTEMPTY;
25012                 else if (flag == DELETE) {
25013                     /* Guardar d_ino p/recuperación. */
25014                     t = NAME_MAX - sizeof(ino_t);
25015                     *((ino_t *) &dp->d_name[t]) = dp->d_ino;
25016                     dp->d_ino = 0; /* borrar entrada */
25017                     bp->b_dirt = DIRTY;
25018                     ldir_ptr->i_update |= CTIME | MTIME;
25019                     ldir_ptr->i_dirt = DIRTY;
25020                 } else {
25021                     sp = ldir_ptr->i_sp; /* 'flag' es LOOK_UP */
25022                     *numb = conv2(sp->s_native, (int) dp->d_ino);
25023                 }
25024                 put_block(bp, DIRECTORY_BLOCK);
25025                 return(r);
25026             }
25027
25028
25029     /* Ver si ranura libre p/beneficio de ENTER. */

```



```

25030             if (flag == ENTER && dp->d_ino == 0) {
25031                 e_hit = TRUE;      /* hallamos ranura libre */
25032                 break;
25033             }
25034         }
25035
25036         /* Buscamos todo el bloque o ENTER tiene ranura libre. */
25037         if (e_hit) break;          /* e_hit es 1 si ENTER ejecutable */
25038         put_block(bp, DIRECTORY_BLOCK); /* si no, seguir buscando */
25039     }
25040
25041     /* Ya se buscó en todo el directorio. */
25042     if (flag != ENTER) return(flag == IS_EMPTY ? OK : ENOENT);
25043
25044     /* Esta llamada es para ENTER si aún no se ha encontrado una ranura libre,
25045      * tratar de extender el directorio.
25046      */
25047     if (e_hit == FALSE) { /* dir lleno y no hay espacio en últ bloque */
25048         new_slots++;      /* aumentar tamaño dir en 1 entrada */
25049         if (new_slots == 0) return(EFBIG); /* tam dir limit por # ranuras */
25050         if ((bp = new_block(ldir_ptr, ldir_ptr->i_size)) == NIL_BUF)
25051             return(err_code);
25052         dp = &bp->b_dir[0];
25053         extended = 1;
25054     }
25055
25056     /* 'bp' ahora apunta a bloque de dir con espacio, 'dp' apunta a ranura. */
25057     (void) memset(dp->d_name, 0, (size_t) NAME_MAX); /* borrar entrada */
25058     for (i = 0; string[i] && i < NAME_MAX; i++) dp->d_name[i] = string[i];
25059     sp = ldir_ptr->i_sp;
25060     dp->d_ino = conv2(sp->s_native, (int) *numb);
25061     bp->b_dirt = DIRTY;
25062     put_block(bp, DIRECTORY_BLOCK);
25063     ldir_ptr->i_update |= CTIME | MTIME; /* marcar mtime p/actualizar */
25064     ldir_ptr->i_dirt = DIRTY;
25065     if (new_slots > old_slots) {
25066         ldir_ptr->i_size = (off_t) new_slots * DIR_ENTRY_SIZE;
25067         /* Enviar cambio a disco si se extendió el directorio. */
25068         if (extended) rw_inode(ldir_ptr, WRITING);
25069     }
25070     return(OK);
25071 }

```

```

+++++
src/fs/mount.c
+++++

```

```

25100 /* Este archivo ejecuta las llamadas al sistema MOUNT y UMount.
25101  *
25102  * Los puntos de entrada a este archivo son
25103  * do_mount: ejecutar llamada al sistema MOUNT
25104  * do_umount: ejecutar llamada al sistema UMount
25105  */
25106
25107     #include "fs.h"
25108     #include <fcntl.h>
25109     #include <minix/com.h>

```

```

25110 #include <sys/stat.h>
25111 #include "buf.h" 25112 #include "dev.h"
25113 #include "file.h"
25114 #include "fproc.h" 25115 #include "inode.h" 25116 #include "param.h" 25117 #include "super.h"
25118
25119 PRIVATE message dev_mess;
25120
25121 FORWARD _PROTOTYPE( dev_t name_to_dev, (char *path)
25122 );
25123 /*=====
25124 * do_mount *
25125 *=====*/
25126 PUBLIC int do_mount()
25127 {
25128     /* Ejecutar la llamada al sistema mount(name, mfile, rd_only). */
25129
25130     register struct inode *rip, *root_ip;
25131     struct super_block *xp, *sp;
25132     dev_t dev;
25133     mode_t bits;
25134     int rdir, mdir; /* TRUE si archivo {rootmount} es dir */
25135     int r, found, majar, task;
25136
25137     /* S610 el superusuario puede ejecutar MOUNT. */
25138     if (!super_user) return(EPERM);
25139
25140     /* Si 'name' no es de un archivo especial por bloques, devolver error. */
25141     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
25142     if ((dev = name_to_dev(user_path)) == NO_DEV) return(err_code);
25143
25144     /* Examinar tabla superbloques p/ver si disp ya montado y hallar ranura*/
25145     sp = NIL_SUPER;
25146     found = FALSE;
25147     for (xp = &super_block[0]; xp < &super_block[NR_SUPERS]; xp++) {
25148         if (xp->s_dev == dev) found = TRUE; /* ¿ya está montado? */
25149         if (xp->s_dev == NO_DEV) sp = xp; /* regist ranura libre */
25150     }
25151     if (found) return(EBUSY); /* ya montado */
25152     if (sp == NIL_SUPER) return(ENFILE); /* no superbloque disponible */
25153
25154     dev_mess.m_type = DEV_OPEN; /* distinguir de cerrar */
25155     dev_mess.DEVICE = dev; /* tocar el dispositivo. */
25156     if (rd_only) dev_mess.COUNT = R_BIT;
25157     else dev_mess.COUNT = R_BITIW_BIT;
25158
25159     majar = (dev » MAJOR) & BYTEj
25160     if (majar <= 0 11 majar >= max_majar) return(ENODEV);
25161     task = dmap[majar].dmap_task; /* núm tarea de dispositivo */
25162     (*dmap[majar].dmap_open) (task, &dev_mess);
25163     if (dev_mess.REP_STATUS != OK) return(EINVAL);
25164
25165     /* Llenar el superbloque. */
25166     sp->s_dev = dev; /* read_super() nec saber cuál disp */
25167     r = read_super(sp);
25168
25169     /* ¿Se reconoce como sistema de archivos Minix? */

```

```

25170  if (r != OK) {
25171      dev_mess.m_type = DEV_CLOSE;
25172      dev_mesS.DEVICE = dev;
25173      (*dmap[major].dmap_close) (task, &dev_mess);
25174      return(r);
25175  }
25176
25177  /* Ahora obtener el nodo-i del archivo en el que se montará. */
25178  if (fetch_name(name2, name2_length, M1) != OK) {
25179      sp->s_dev = NO_DEV;
25180      dev_mess.m_type = DEV_CLOSE;
25181      dev_mesS.DEVICE = dev;
25182      (*dmap[major].dmap_close) (task, &dev_mess);
25183      return(err_code);
25184  }
25185  if ((rip = eat_path(user_path)) == NIL_INOUE) {
25186      sp->s_dev = NO_DEV;
25187      dev_mess.m_type = DEV_CLOSE;
25188      dev_mesS.DEVICE = dev;
25189      (*dmap[major].dmap_close) (task, &dev_mess);
25190      return(err_cOde);
25191  }
25192
25193  /* Podría no estar ocupado. */
25194  r = OK;
25195  if (rip->i_count > 1) r = EBUSY;
25196
25197  /* Podría no ser especial. */
25198  bits = rip->i_mode & I_TYPE;
25199  if (bits == I_BLOCK_SPECIAL || bits == I_CHAR_SPECIAL) r = ENOTDIR;
25200
25201  /* Obtener nodo-i del sistema de archivos montado. */
25202  root_ip = NIL_INODE;          /* si 'r' no OK, asegurar q/esté definido */
25203  if (r == OK) {
25204      if ((root_ip = get_inode(dev, ROOT_INODE)) == NIL_INODE) r = err_code;
25205  }
25206  if (root_ip != NIL_INODE && root_ip->i_mode == 0) r = EINVAL;
25207
25208  /* No puede haber conflicto entre tipos de arch 'rip' y 'root_ip'. */
25209  if (r == OK) {
25210      mdir = ((rip->i_mode & I_TYPE) == I_DIRECTORY); /* TRUE si dir */
25211      rdir = ((root_ip->i_mode & I_TYPE) == I_DIRECTORY);
25212      if (!mdir && rdir) r = EISDIR;
25213  }
25214
25215  /* Si error, devolver superbloque y ambos nodos-ij liberar los mapas. */
25216  if (r != OK) {
25217      put_inode(rip);
25218      put_inOde(root_ip);
25219      (void) do_sync();
25220      invalidate(dev);
25221
25222      sp->s_dev = NO_DEV;
25223      dev_mess.m_type = DEV_CLOSE;
25224      dev_mesS.DEVICE = dev;
25225      (*dmap[major].dmap_close) (task, &dev_mess);
25226      return(r);
25227  }
25228
25229  /* Nada más puede salir mal. Montar. */

```

```

25230 rip->i_mount = I_MOUNT;          /* este bit dice que montado en nodo-i */
25231 sp->s_imount = rip;
25232 sp->s_isup = root_ip;
25233 sp->s_rd_only = rd_only;
25234     return(OK);
25235 }
25238/*=====
25239 *                               do_umount                               *
25240*=====*/
25241 PUBLIC int do_umount()
25242 {
25243     /* Ejecutar la llamada al sistema umount(name). */
25244
25245     register struct inode *rip;
25246     struct super_block *sp, *sp1;
25247     dev_t dev;
25248     int count;
25249     int majar, task;
25250
25251     /* Sólo el superusuario puede ejecutar UMOUNT. */
25252     if (!super_user) return(EPERM);
25253
25254     /* Si 'name' no es de un archivo especial por bloques, devolver error. */
25255     if (fetch_name(name, name_length, M3) != OK) return(err_code);
25256     if ((dev = name_to_dev(user_path)) == NO_DEV) return(err_code);
25257
25258     /* Ver si el disp montado está ocupado. Sólo debe estar abierto
25259      * un nodo-i que lo use, el raíz, y ese nodo-i sólo una vez.
25260      */
25261     count = 0;
25262     for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++)
25263         if (rip->i_count > 0 && rip->i_dev == dev) count += rip->i_count;
25264     if (count > 1) return(EBUSY); /* imposible desmontar si ocupado */
25265
25266     /* Encontrar el superbloque. */
25267     sp = NIL_SUPER;
25268     for (sp1 = &super_block[0]; sp1 < &super_block[NR_SUPERS]; sp1++) {
25269         if (sp1->s_dev == dev) {
25270             sp = sp1;
25271             break;
25272         }
25273     }
25274
25275     /* Sincronizar el disco e invalidar el caché. */
25276     (void) do_sync();          /* sacar de memoria bloques en caché */
25277     invalidate(dev);          /* invalidar entradas caché este disp */
25278     if (sp == NIL_SUPER) return(EINVAL);
25279
25280     majar = (dev » MAJOR) & BYTE;          /* núm disp principal */
25281     task = dmap[majar].dmap_task; /* núm tarea de dispositivo */
25282     dev_mess.m_type = DEV_CLOSE;          /* distinguir de open */
25283     dev_mess.DEVICE = dev;
25284     (*dmap[majar].dmap_close)(task, &dev_mess);
25285
25286     /* Terminar de desmontar. */
25287     sp->s_imount->i_mount = NO_MOUNT;      /* nodo.i vuelve a normal */
25288     put_inode(sp->s_imount);          /* liberar nodo-i en el que se montó */
25289     put_inode(sp->s_isup);          /* liberar nodo-i raíz del fs montado */

```

```

25290     sp->s_imount = NIL_INODE;
25291     sp->s_dev = NO_DEV;
25292     return(OK);
25293 }
25296 /*=====
25297 *                                     name_to_dev                                     *
25298 *=====*/
25299     PRIVATE dev_t name_to_dev(path)
25300 char *pathj                                     /* apunt a nombre ruta */
25301 {
25302     /* Convertir el arch esp p/bloques 'path' en núm de dispositivo. si 'path'
25303 * no es arch especial p/bloques, devolver cód. error en 'err_code'.
25304 */
25305
25306     register struct inode *rip;
25307     register dev_t dev;
25308
25309     /* Si 'path' no puede abrirse, desistir de inmediato. */
25310     if ((rip = eat_path(path)) == NIL_INODE) return(NO_DEV);
25311
25312     /* Si 'path' no es archivo especial por bloques, devolver error. */
25313     if ((rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) {
25314         err_code = ENOTBLK;
25315         put_inode(rip);
25316         return(NO_DEV);
25317     }
25318
25319     /* Extraer el número de dispositivo. */
25320     dev = (dev_t) rip->i_zone[0];
25321     put_inode(rip);
25322     return(dev);
25323 }
+++++
                                     srC/fs/link.c
+++++
25400     /* Este archivo maneja las llamadas LINK y UNLINK, y también se encarga de
25401 * liberar la memoria ocupada por un archivo cuando se ejecuta el último UNLINK
25402 * a un archivo y los bloques deben devolverse a la reserva de bloques libres.
25403 *
25404 * Los puntos de entrada a este archivo son
25405 *   do_link: ejecutar llamada al sistema LINK
25406 *   do_unlink: ejecutar llamadas a sistemas UNLINK y RMDIR
25407 *   do_rename: ejecutar llamada al sistema RENAME
25408 *   truncate: liberar todos los bloques asociados a un nodo.i
25409 */
25410
25411 #include "fs.h"
25412 #include <sys/stat.h>
25413 #include <string.h>
25414 #include <minix/callnr.h>
25415 #include "buf.h"
25416 #include "file.h"
25417 #include "fproc.h"
25418 #include "inode.h"
25419 #include "param.h"

```

```

25420 #include "super.h"
25421
25422 #define SAME 1000
25423
25424 FORWARD_PROTOTYPE( int remove_dir, (struct inode *rldirp, struct inode *rip,
25425                                     char dir_name[NAME_MAX])
25426                                     );
25427 FORWARD_PROTOTYPE( int unlink_file, (struct inode *dirp, struct inode *rip,
25428                                     char file_name[NAME_MAX])
25429                                     );
25430
25431 /*=====
25432 *                                     do_link                                     *
25433 *=====*/
25434 PUBLIC int do_link()
25435 {
25436     /* Ejecutar la llamada al sistema link(name1, name2). */
25437
25438     register struct inode *ip, *rip;
25439     register int r;
25440     char string[NAME_MAX];
25441     struct inode *new_ip;
25442
25443     /* Ver si existe 'name' (archivo al que se vinculará). */
25444     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
25445     if ((rip = eat_path(user_path)) == NIL_INODE) return(err_code);
25446
25447     /* Ver si el archivo ya tiene el máximo de vínculos. */
25448     r = OK;
25449     if ((rip->i_nlinks & BYTE) >= LINK_MAX) r = EMLINK;
25450
25451     /* Sólo el superusuario puede vincular a directorios. */
25452     if (r == OK)
25453         if ((rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;
25454
25455     /* Si error con 'name', devolver el nodo-i. */
25456     if (r != OK) {
25457         put_inode(rip);
25458         return(r);
25459     }
25460
25461     /* ¿Existe el directorio final para 'name2'? */
25462     if (fetch_name(name2, name2_length, M1) != OK) {
25463         put_inode(rip);
25464         return(err_code);
25465     }
25466     if ((ip = last_dir(user_path, string)) == NIL_INODE) r = err_code;
25467
25468     /* Si 'name2' existe completo (aunque no espacio), hacer r = error. */
25469     if (r == OK) {
25470         if ((new_ip = advance(ip, string)) == NIL_INODE) {
25471             r = err_code;
25472             if (r == ENOENT) r = OK;
25473         } else {
25474             put_inode(new_ip);
25475             r = EEXIST;
25476         }
25477     }
25478
25479     /* Ver si hay vínculos entre dispositivos. */

```

```

25480 if (r == OK)
25481     if (rip->i_dev != ip->i_dev) r = EXDEV;
25482
25483 /* Tratar de vincular. */
25484 if (r == OK)
25485     r = sea_rch_dir(ip, string, &rip->i_num, ENTER);
25486
25487 /* Si se logra, registrar el vínculo. */
25488 if (r == OK) {
25489     rip->i_nlinks++;
25490     rip->i_update |= CTIME;
25491     rip->i_dirt = DIRTY;
25492 }
25493
25494 /* Hecho. Liberar ambos nodos-i. */
25495     put_inode(rip);
25496     put_inode(ip);
25497     return(r);
25498 }
25501 /*=====
25502 *                               do_unlink                               *
25503 *=====*/
25504 PUBLIC int do_unlink()
25505 {
25506     /* Ejecutar la llamada al sistema unlink(name) o rmdir(name). El código .de las dos
25507 * es casi igual, sólo difieren en ciertas pruebas de condiciones. Unlink() puede
25508 * ser utilizado por el superusuario para hacer cosas peligrosas; rmdir() no.
25509 */
25510
25511     register struct inode *rip;
25512     struct inode *rldirp;
25513     int r;
25514     char string[NAME_MAX];
25515
25516     /* Obtener el último directorio de la ruta. */
25517     if (fetch_name(name, name_length, M3) != OK) return(err_code);
25518     if ( (rldirp = last_dir(user_path, string) == NIL_INODE)
25519         return(err_code);
25520
25521     /* Existe el último directorio. ¿Existe también el archivo? */
25522     r = OK;
25523     if ( (rip = advance(rldirp, string) == NIL_INODE) r = err_code;
25524
25525     /* Si error, devolver nodo-i. */
25526     if (r != OK) {
25527         put_inode(rldirp);
25528         return(r);
25529     }
25530
25531     /* No quitar un punto de montura. */
25532     if (rip->i_num == ROOT_INODE) {
25533         put_inode(rldirp);
25534         put_inode(rip);
25535         return(EBUSY);
25536     }
25537
25538     /* Ahora ver si se permite la llamada. Distinto para unlink() y rmdir(). */
25539     if (fs_call == UNLINK) {

```

```

25540      /* Sólo su puede desvincular directorios, y puede desvincular cualquier dir. */
25541      if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;
25542
25543      /* No desvinc. archivo si es raíz de un sistema de archivos montado. */
25544      if (rip->i_num == ROOT_INODE) r = EBUSY;
25545
25546      /* Tratar realmente de desvinc. archivo; falla si padre es modo 0, etc. */
25547      if (r == OK) r = unlink_file(rldirp, rip, string);
25548
25549  } else {
25550      r = remove_dir(rldirp, rip, string); /* llamada es RMDIR */
25551  }
25552
25553  /* Si pudo desvincularse, se hizo si no, no. */
25554      put_inode(rip);
25555      put_inode(rldirp);
25556      return(r);
25557  }
25560  /*=====
25561  *                                     do_rename                                     *
25562  *=====*/
25563      PUBLIC int do_rename()
25564  {
25565      /* Ejecutar la llamada al sistema rename(name1, name2). */
25566
25567      struct inode *old_dirp, *old_ip;          /* apunts a dir viejo, nada arch */
25568      struct inode *new_dirp, *new_ip;        /* apunts a dir nvo, nada arch */
25569      struct inode *new_superdirp, *next_new_superdirp;
25570      int r = OK;                             /* bandera error; inic ninguno */
25571      int odir, ndir;                          /* TRUE si arch {viejo|nvo} es dir */
25572      int same_pdir;                          /* TRUE si dirs padres iguales */
25573      char old_name[NAME_MAX], new_name[NAME_MAX];
25574      ino_t numb;
25575      int r1;
25576
25577      /* Ver si existe 'name1' (arch existente). Obtener nadas de dir y arch. */
25578      if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
25579      if ( (old_dirp = last_dir(user_path, old_name)) == NIL_INODE) return(err_code);
25580
25581      if ( (old_ip = advance(old_dirp, old_name)) == NIL_INODE) r = err_code;
25582
25583      /* Ver si existe 'name2'. (arch nuevo). Obtener nadas de dir y arch. */
25584      if (fetch_name(name2, name2_length, M1) != OK) r = err_code;
25585      if ( (new_dirp = last_dir(user_path, new_name)) == NIL_INODE) r = err_code;
25586      new_ip = advance(new_dirp, new_name); /* no tiene que existir */
25587
25588      if (old_ip != NIL_INODE)
25589          odir = «old_ip->i_mode & I_TYPE) == I_DIRECTORY)j          /* TRUE si dir */
25590
25591      /* Si está bien, tratar de detectar diversos errores posibles. */
25592      if (r == OK) {
25593          same_pdir = (old_dirp == new_dirp);
25594
25595          /* El nodo-i viejo no debe ser superdir del nuevo últ dir. */
25596          if (odir && !same_pdir) {
25597              dup_inode(new_superdirp = new_dirp);
25598              while (TRUE) { /* puede esperar en ciclo del FS */
25599                  if (new_superdirp == old_ip) {

```



```

25600             r = EINVAL;
25601             break;
25602         }
25603         next_new_superdirp = advance(new_superdirp, dot2);
25604         put_inode(new_superdirp);
25605         if (next_new_superdirp == new_superdirp)
25606             break; /* vuelta a dir raiz del sist */
25607         new_superdirp = next_new_superdirp;
25608         if (new_superdirp == NIL_INODE) {
25609             /* Falta entrada".", suponer lo peor. */
25610             r = EINVAL;
25611             break;
25612         }
25613     }
25614     put_inode(new_superdirp);
25615 }
25616
25617 /* Nombre viejo o nuevo no debe ser. ni..*/
25618 if (strcmp(old_name, ".")==0 || strcmp(old_name, ". .")==0 ||
25619     strcmp(new_name, ".")==0 || strcmp(new_name, "..")==0) r = EINVAL;
25620
25621 /* Ambos dirs padre deben estar en el mismo dispositivo. */
25622 if (old_dirp->i_dev != new_dirp->i_dev) r = EXDEV;
25623
25624 /* Dirs padre deben ser escribibles, buscables y en un disp escribible. */
25625 if ((r1 = forbidden(old_dirp, W_BIT | X_BIT) != OK ||
25626     (r1 = forbidden(new_dirp, W_BIT | X_BIT) != OK) r = r1;
25627
25628 /* Algunas pruebas aplican sólo si existe la nva ruta. */
25629 if (new_ip == NIL_INODE) {
25630     /* no renombrar arch que tiene un FS montado. */
25631     if (old_ip->i_dev != old_dirp->i_dev) r = EXDEV;
25632     if (odir && (new_dirp->i_nlinks & BYTE) >= LINK_MAX &&
25633         !same_pdir && r == OK) r = EMLINK;
25634 } else {
25635     if (old_ip == new_ip) r = SAMEj /* viejo=nuevo */
25636
25637     /* ¿el arch viejo o nuevo tiene un sistema de archivos montado? */
25638     if (old_ip->i_dev != new_ip->i_dev) r = EXDEV;
25639
25640     ndir = ((new_ip->i_mode & I_TYPE) == I_DIRECTORY); /* ¿dir? */
25641     if (odir == TRUE && ndir == FALSE) r = ENOTDIR;
25642     if (odir == FALSE && ndir == TRUE) r = EISDIR;
25643 }
25644 }
25645
25646 /* Si un proceso tiene otro directorio raíz que el del sistema,
25647 * podríamos mover "accidentalmente" su directorio de trabajo
25648 * a un lugar donde su directorio raíz ya no sea superdirectorío suyo.
25649 * Esto puede hacer inútil la función chroot. Si se usará chroot
25650 * a menudo, tal vez debamos tratar de detectarlo aquí.
25651 */
25652
25653 /* El cambio de nombre quizá funcionará. Ya sólo puede haber 2 problemas:
25654 * 1. no poder quitar el nuevo archivo (si ya existe)
25655 * 2. no poder crear la nueva entrada de dir (arch nuevo no existe)
25656 *           [el directorio tiene que crecer en un bloque y no puede
25657 *           porque el disco está totalmente lleno].
25658 */
25659 if (r == OK) {

```

```

25660     if (new_ip != NIL_INODE) {
25661         /* Ya hay 1 entrada para 'new'. Tratar de quitarla. */
25662         if (odir)
25663             r = remove_dir(new_dirp, new_ip, new_name);
25664         else
25665             r = unlink_file(new_dirp, new_ip, new_name);
25666     }
25667     /* Si r es OK, se cambiará el nombre; ahora hay una entrada
25668     * no utilizada en el nuevo directorio padre.
25669     */
25670 }
25671
25672 if (r == OK) {
25673     /* Si el nuevo nombre estará en el mismo dir padre que el viejo,
25674     * primero quitar el viejo a fin de liberar una entrada para el nuevo
25675     * nombre. Si no, primero tratar de crear la nueva entrada de nombre
25676     * para asegurar que el cambio de nombre tenga éxito.
25677     */
25678     numb = old_ip->i_num;          /* núm nodo-i de archivo viejo */
25679
25680     if (same_pdir) {
25681         r = search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
25682         /* no debería fallar. */
25683         if (r == OK) (void) search_dir(old_dirp, new_name, &numb, ENTER);
25684     } else {
25685         r = search_dir(new_dirp, new_name, &numb, ENTER);
25686         if (r == OK)
25687             (void) search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
25688     }
25689 }
25690 /* Si r es OK, ctime y mtime de old_dirp se habrán marcado
25691 * para actualización en search_dir.
25692 */
25693
25694 if (r == OK && odir && !same_pdir) {
25695     /* Actualizar entrada.. en directorio (aún apunta a old_dirp). */
25696     numb = new_dirp->i_num;
25697     (void) unlink_file(old_ip, NIL_INODE, dot2);
25698     if (search_dir(old_ip, dot2, &numb, ENTER) == OK) {
25699         /* Nuevo vínculo creado. */
25700         new_dirp->i_nlinks++;
25701         new_dirp->i_dirt = DIRTY;
25702     }
25703 }
25704
25705 /* Liberar los nodos-i. */
25706     put_inode(old_dirp);
25707     put_inode(old_ip);
25708     put_inode(new_dirp);
25709     put_inode(new_ip);
25710     return(r == SAME ? OK : r);
25711 }
25714 /*=====
25715 *                               truncate                               *
25716 *=====*/
25717 PUBLIC void truncate(rip)
25718 register struct inode *rip;          /* apuntador a nodo-i por truncar */
25719 {

```



```

25720          /* Quitar todas las zonas del nodo-i 'rip' y marcarlo sucio. */
25721
25722     register block_t b;
25723     zone_t z, zone_size, z1 ;
25724     off_t ~ position;
25725     int i, scale, file_type, waspipe, single, nr_indirects;
25726     struct buf *bp;
25727     dev_t dev;
25728
25729     file_type = rip->i_mode & I_TYPE;          /* Ver si arch es especial */
25730     if (file_type == I_CHAR_SPECIAL || file_type == I_BLOCK_SPECIAL) return;
25731     dev = rip->i_dev;          /* disp en que reside nodo-i */
25732     scale = rip->i_sp->s_log_zone_size;
25733     zone_size = (zone_t) BLOCK_SIZE «      scale;
25734     nr_indirects = rip->i_nindirs;
25735
25736     /* Conductos pueden encogerse; ajustar tamaño p/quitar todas zonas. */
25737     waspipe = rip->i_pipe == I_PIPE;          /* TRUE si era un conducto */
25738     if (waspipe) rip->i_size = PIPE_SIZE;
25739
25740     /* Recorrer archivo zona x zona, hallando y liberando las zonas. */
25741     for (position = 0; position < rip->i_size; position += zone_size) {
25742         if ((b = read_map(rip, position)) != NO_BLOCK) {
25743             z = (zone_t) b »      scale;
25744             free_zone(dev, z);
25745         }
25746     }
25747
25748     /* Todas las zonas de datos liberadas. Liberar zonas de indirección. */
25749     rip->i_dirt = DIRTY;
25750     if (waspipe) {
25751         wipe_inode(rip);          /* despejar nodo-i p/conductos */
25752         return;          /* ranuras indirec contienen pos archivos */
25753     }
25754     single = rip->i_ndzones;
25755     free_zone(dev, rip->i_zone[single]);      /* zona indirec sencilla */
25756     if ((z = rip->i_zone[single+1]) != NO_ZONE) {
25757         /* Liberar todas las zonas de indirec senc a que apunta la doble. */
25758         b = (block_t) z «      scale;
25759         bp = get_block(dev, b, NORMAL); /* obt zona doble indirec */
25760         for (i = 0; i < nr_indirects; i++) {
25761             z1 = rd_indir(bp, i);
25762             free_zone(dev, z1);
25763         }
25764
25765         /* Ahora liberar la zona de doble indirección misma. */
25766         put_block(bp, INDIRECT_BLOCK);
25767         free_zone(dev, z);
25768     }
25769
25770     /* Dejar núms de zona p/que de(1) recupere archivo después de unlink(2). */
25771 }
25774/*=====
25775 *                               remove_dir                               *
25776 *=====*/
25777 PRIVATE int remove_dir(rldirp, rip, dir_name)
25778 struct inode *rldirp;          /* directorio padre */
25779 struct inode *rip; /* directorio por eliminar */

```

```

25780 char dir_name[NAME_MAX];                               /* nombre dir por eliminar */
25781 {
25782     /* Debe eliminarse un arch de directorio. Deben cumplirse 5 condiciones:
25783     *     -El archivo debe ser un directorio
25784     *     -El directorio debe estar vacío (excepto por. y ..)
25785     *           El componente final de la ruta no debe ser. ni ..
25786     *     -El dir no debe ser raíz de un sistema de archivos montado
25787     *     -El dir no debe ser directorio raíz/de trabajo de nadie
25788     */
25789
25790     int r;
25791     register struct fproc *rfp;
25792
25793     /* search_dir verifica que rip también es un directorio. */
25794     if ((r = search_dir(rip, "", (ino_t *) 0, IS_EMPTY)) != OK) return r;
25795
25796     if (strcmp(dir_name, ".") == 0 || strcmp(dir_name, "..") == 0) return(EINVAL);
25797     if (rip->i_num == ROOT_INODE) return(EBUSY); /* impos elim 'root' */
25798
25799     for (rfp = &fproc[INIT_PROC_NR + 1]; rfp < &fproc[NR_PROCS1]; rfp++)
25800         if (rfp->fp_workdir == rip || rfp->fp_rootdir == rip) return(EBUSY);
25801                                     /* imposible eliminar el dir de trabajo de alguien */
25802
25803     /* Tratar de desvincular archivo; falla si padre es modo 0 etc. */
25804     if ((r = unlink_file(rldirp, rip, dir_name)) != OK) return r;
25805
25806     /* Desvincular. y ..del dir. El superusuario puede vincular y desvincular
25807     * cualquier dir, así que no suponer demasiado acerca de ellos.
25808     */
25809     (void) unlink_file(rip, NIL_INODE, dot1);
25810     (void) unlink_file(rip, NIL_INODE, dot2);
25811     return(OK);
25812 }
25815 /*=====
25816 *                               unlink_file                               *
25817 *=====*/
25818 PRIVATE int unlink_file(dirp, rip, file_name)
25819 struct inode *dirp;           /* dir padre del archivo */
25820 struct inode *rip;           /* nodo-i del arch, puede ser NIL_INODE. */
25821 char file_name[NAME_MAX1];   /* nombre archivo por eliminar */
25822 {
25823     /* Desvinc. 'file_name'; rip debe ser nodo-i de 'file_name' o NIL_INODE. */
25824
25825     ino_t numb;               /* número de nodo-i */
25826     int r;
25827
25828     /* Si rip no es NIL_INODE, se usa p/acceso más rápido al nodo-i. */
25829     if (rip == NIL_INODE) {
25830         /* Buscar archivo en dir y tratar de obtener su nodo-i. */
25831         err_code = search_dir(dirp, file_name, &numb, LOOK_UP);
25832         if (err_code == OK) rip = get_inode(dirp->i_dev, (int) numb);
25833         if (err_code != OK || rip == NIL_INODE) return(err_code);
25834     } else {
25835         dup_inode(rip);       /* se devolverá nodo-i con put_inode */
25836     }
25837
25838     r = search_dir(dirp, file_name, (ino_t *) 0, DELETE);
25839

```

```

25840     if (r == OK) {
25841         rip->i_nlinks--;           /* entrada eliminada de dir padre */
25842         rip->i_update 1= CTIME;
25843         rip->i_dirt = OIRTY;
25844     }
25845
25846     put_inode(rip);
25847     return(r);
25848 }

```

 src/fs/stadir.c

```

25900 /* Este archivo contiene el código para ejecutar cuatro llamadas
25901 * al sistema relacionadas con la situación y los directorios.
25902 *
25903 * Los puntos de entrada a este archivo son
25904 * do_chdir: ejecuta la llamada al sistema CHDIR
25905 * do_chroot: ejecuta la llamada al sistema CHROOT
25906 * do_stat: ejecuta la llamada al sistema STAT
25907 * do_fstat: ejecuta la llamada al sistema FSTAT
25908 */
25909
25910 #include "fs.h"
25911 #include <sys/stat.h>
25912 #include "file.h"
25913 #include "fproc.h"
25914 #include "inode.h"
25915 #include "param.h"
25916
25917 FORWARD_PROTOYPE( int change, (struct inode **iip, char *name_ptr, int len));
25918 FORWARD_PROTOYPE( int stat_inode, (struct inode *rip, struct filp *fil_ptr,
25919                                     char *user_addr) );
25920
25921 /*=====
25922 *                                     do_chdir                                     *
25923 *=====*/
25924 PUBLIC int do_chdir()
25925 {
25926     /* Cambiar directorio. MM llama también esta función para simular un chdir
25927     * al ejecutar EXEC, etc. También cambia el directorio raíz, los uids y gids,
25928     * Y la máscara.
25929     */
25930
25931     int r;
25932     register struct fproc *rfp;
25933
25934     if (who == MM_PROC_NR) {
25935         rfp = &fproc[slot1];
25936         put_inode(fp->fp_rootdir);
25937         dup_inode(fp->fp_rootdir = rfp->fp_rootdir);
25938         put_inode(fp->fp_workdir);
25939         dup_inode(fp->fp_workdir = rfp->fp_workdir);
25940
25941         /* MM usa access() para verificar permisos. Para que funcione, suponer
25942         * que los id reales del usuario son iguales a los efectivos. Las llamadas
25943         * del FS aparte de access() no usan los id reales, así que no resultan
25944         * afectadas.

```

```

25945  */
25946      fp->fp_realuid =
25947      fp->fp_effuid = rfp->fp_effuid;
25948      fp->fp_realgid =
25949      fp->fp_effgid = rfp->fp_effgid;
25950      fp->fp_umask = rfp->fp_umask;
25951      return(OK);
25952  }
25953
25954      /* Ejecutar la llamada al sistema chdir(name). */
25955      r = change(&fp->fp_workdir, name, name_length);
25956      return(r);
25957  }
25960  /*=====*/
25961      *                               do_chroot                               *
25962  /*=====*/
25963  PUBLIC int do_chroot()
25964  {
25965      /* Ejecutar la llamada al sistema chroot(name). */
25966
25967      register int r;
25968
25969      if (!super_user) return(EPERM);          /* sólo su puede chroot() */
25970      r = change(&fp->fp_rootdir, name, name_length);
25971      return(r);
25972  }
25975  /*=====*/
25976      *                               change                               *
25977  /*=====*/
25978  PRIVATE int change(iip, name_ptr, len)
25979  struct inode **iip;          /* apunt a apunt nodo-i piel dir */
25980  char *name_ptr;             /* apunt al nombre dir a cambiar */
25981  int len;                    /* long cadena de nombre del dir */
25982  {
25983      /* Realizar el trabajo real de chdir() y chroot(). */
25984
25985      struct inode *rip;
25986      register int r;
25987
25988      /* Tratar de abrir el nuevo directorio. */
25989      if (fetch_name(name_ptr, len, M3) != OK) return(err_code);
25990      if ( (rip = eat_path(uSer_path)) == NIL_INODE) return(err_code);
25991
25992      /* Debe ser un directorio y también buscable. */
25993      if ( (rip->i_mode & I_TYPE) != I_DIRECTORY)
25994          r = ENOTDIR;
25995      else
25996          r = forbidden(rip, X_BIT);          /* ver si dir es buscable */
25997
25998      /* Si hay error, devolver nodo-i. */
25999      if (r != OK) {
26000          put_inode(rip);
26001          return(r);
26002      }
26003
26004      /* Todo bien. Hacer el cambio. */

```

```

26005     put_inode(*iip);                /* liberar directorio viejo */
26006     *iip = rip;                      /* adquirir el nuevo */
26007     return(OK);
26008 }
26011 /*=====
26012     *                                do_stat                                *
26013 *=====*/
26014     PUBLIC int do_stat()
26015     {
26016     /* ejecutar la llamada al sistema stat(name, buf). */
26017
26018     register struct inode *rip;
26019     register int r;
26020
26021     /* Stat() y fstat() usan la misma rutina para efectuar el trabajo real. Esa rutina
26022     * espera un nodo-i, así que lo adquirimos temporalmente.
26023     */
26024     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
26025     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
26026     r = stat_inode(rip, NIL_FILP, name2); /* Hacer realmente el trabajo. */
26027     put_inOde(rip);                    /* liberar nodo-i */
26028     return(r);
26029 }
26032 /*=====
26033     *                                do_fstat                             *
26034 *=====*/
26035     PUBLIC int do_fstat()
26036     {
26037     /* Ejecutar la llamada al sistema fstat(fd, buf). */
26038
26039     register struct filp *rfilp;
26040
26041     /* ¿Es válido el descriptor de archivo? */
26042     if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);
26043
26044     return(stat_inOde(rfilp->filp_ino, rfilp, buffer));
26045 }
26048 /*=====
26049     *                                stat_inode                            *
26050 *=====*/
26051     PRIVATE int stat_inode(rip, fil_ptr, user_addr)
26052     register struct inode *rip;        /* apunt a nodo-i p/stat */
26053     struct filp *fil_ptr;              /* apunt filp, proporc por 'fstat' */
26054     char *user_addr;                  /* dir espacio usuario p/buf stat */
26055     {
26056     /* Código común para las llamadas al sistema stat y fstat. */
26057
26058     struct stat statbuf;
26059     mode_t mo;
26060     int r, s;
26061
26062     /* Actualizar campos atime, ctime y mtime del nodo-i, si necesario. */
26063     if (rip->i_update) update_times(rip);
26064

```



```

26065 /* Llenar la estructura statbuf. */
26066 mo = rip->i_mode & I_TYPE;
26067 s = (mo == I_CHAR_SPECIAL || mo == I_BLOCK_SPECIAL); /* TRUE si especial */
26068 statbuf.st_dev = rip->i_dev;
26069 statbuf.st_ino = rip->i_num;
26070 statbuf.st_mode = rip->i_mode;
26071 statbuf.st_nlink = rip->i_nlinks & BYTE;
26072 statbuf.st_uid = rip->i_uid;
26073 statbuf.st_gid = rip->i_gid & BYTE;
26074 statbuf.st_rdev = (dev_t) (s ? rip->i_zone[0] : NO_DEV);
26075 statbuf.st_size = rip->i_size;
26076
26077 if (rip->i_pipe == I_PIPE) {
26078     statbuf.st_mode &= ~I_REGULAR; /* borrar bit I_REGULAR p/conductos */
26079     if (fil_ptr != NIL_FILP && fil_ptr->filp_mode & R_BIT)
26080         statbuf.st_size -= fil_ptr->filp_pos;
26081 }
26082
26083 statbuf.st_atime = rip->i_atime;
26084 statbuf.st_mtime = rip->i_mtime;
26085 statbuf.st_ctime = rip->i_ctime;
26086
26087 /* Copiar la estructura en el espacio de usuario. */
26088 r = sys_copy (FS_PROC_NR, D, (phys_bytes) &statbuf,
26089             who, D, (phys_bytes) user_addr, (phys_bytes) sizeof(statbuf));
26090 return(r);
26091 }

```

+++++

src/fs/protect.c

+++++

```

26100 /* Este archivo maneja la protección en el sistema de archivos; contiene
26101 * el código de 4 llamadas al sistema relacionadas con protección.
26102 *
26103 * Los puntos de entrada a este archivo son
26104 * do_chmod: ejecutar la llamada al sistema CHMOD
26105 * do_chown: ejecutar la llamada al sistema CHOWN
26106 * do_umask: ejecutar la llamada al sistema UMASK
26107 * do_access: ejecutar la llamada al sistema ACCESS
26108 * forbidden: ver si acceso dado permitido a un nodo-i dado
26109 */
26110
26111 #include "fs.h"
26112 #include <unistd.h>
26113 #include <minix/callnr.h>
26114 #include "buf.h"
26115 #include "file.h"
26116 #include "fproc.h"
26117 #include "inode.h"
26118 #include "param.h"
26119 #include "super.h"
26120
26121 /*=====
26122 *                                     do_chmod                                     *
26123 *=====*/
26124 PUBLIC int do_chmod()

```

```

26125 {
26126 /* Ejecutar la llamada al sistema ChmOd(name, mode). */
26127
26128     register struct inode *rip;
26129     register int r;
26130
26131     /* Abrir 'temporalmente' el archivo. */
26132     if (fetch_name(name, name_length, M3) != OK) return(err_code);
26133     if ( (rip = eat_path(user_path) == NIL_INODE) return(err_cOde);
26134
26135         /* Sólo el dueño o el su puede cambiar el modo de un archivo. Nadie puede cambiar
26136         * el modo de un arch en un sistema de archivos sólo de lectura.
26137         */
26138     if (rip->i_uid != fp->fp_effuid && !super_user)
26139         r = EPERM;
26140     else
26141         r = read_only(rip);
26142
26143     /* Si hay error, devolver nodo-i. */
26144     if (r != OK) {
26145         put_inOde(rip);
26146         return(r);
26147     }
26148
26149     /* Hacer cambio. Apagar bit setgid si arch no en grupo del invocador */
26150     rip->i_mode = (rip->i_mode & -ALL_MODES) | (mode & ALL_MODES);
26151     if (!super_user && rip->i_gid != fp->fp_effgid) rip->i_mode &= -I_SET_GID_BIT;
26152     rip->i_update |= CTIME;
26153     rip->i_dirt = DIRTY;
26154
26155     put_inOde(rip);
26156     return(OK);
26157 }
26160 /*=====
26161 *                               do chown                               *
26162 *=====
26163 PUBLIC int do_Chown()
26164 {
26165     /* Ejecutar la llamada al sistema chown(name, owner, group). */
26166
26167     register struct inode *rip;
26168     register int r;
26169
26170     /* Abrir temporalmente el archivo. */
26171     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
26172     if ( (rip = eat_path(user_path) == NIL_INODE) return(err_code);
26173
26174     /* No se permite cambiar dueño de un arch en un sist arch sólo de lectura. */
26175     r = read_only(rip);
26176     if (r == OK) {
26177         /* FS es R/W. Que se permita llamada o no depende de dueño, etc. */
26178         if (super_user) {
26179             /* El superusuario puede hacer todo. */
26180             rip->i_uid = owner;                /* otros después */
26181         } else {
26182             /* Usuarios normales sólo pueden cambiar grupos de arch propios. */
26183             if (rip->i_Uid != fp->fp_effuid) r = EPERM;
26184             if (rip->i_uid != owner) r = EPERM;        /* no regalar */

```

```

26185         if (fp->fp_effgid != group) r = EPERM;
26186     }
26187 }
26188     if (r == OK) {
26189         rip->i_gid = group;
26190         rip->i_mode &= ~(I_SET_UID_BIT | I_SET_GID_BIT);
26191         rip->i_uPdate |= CTIME;
26192         rip->i_dirt = DIRTY;
26193     }
26194
26195     put_inOde(rip);
26196     return(r);
26197 }
26200 /*=====
26201 *                                     do_umask
26202 *=====*/
26203 PUBLIC int do_umask()
26204 {
26205     /* Ejecutar la llamada al sistema Umask(co_mOde) o */
26206     register mode_t r;
26207
26208     r = -fp->fp_umask;          /* hacer 'r' = complemento máscara Vieja */
26209     fp->fp_umask = ~(Co_mode & RWX_MODES);
26210     return(r);                /* devolver complemento máscara vieja */
26211 }
26214 /*=====
26215 *                                     do_access
26216 *=====*/
26217 PUBLIC int do_access()
26218 {
26219     /* Ejecutar la llamada al Sistema access(name, mode). */
26220
26221     struct inOde *rip;
26222     register int r;
26223
26224     /* Primero ver si el modo es el Correcto. */
26225     if ( (mode & ~(R_OK | W_OK | X_OK)) != 0 && mode != F_OK)
26226         return(EINVAL);
26227
26228     /* Abrir temporalmente el archivo cuyo acceso se verificará. */
26229     if (fetch_name(name, name_length, M3) != OK) return(err_CODE);
26230     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_CODE);
26231
26232     /* Ahora verificar los permisos. */
26233     r = forbidden(rip, (mode_t) mode);
26234     put_inOde(rip);
26235     return(r);
26236 }
26239 /*=====
26240 *                                     forbidden
26241 *=====*/
26242 PUBLIC int forbidden(rip, access_desired)
26243 register struct inOde *rip;    /* apunt a nodo-i por verificar */
26244 mode_t access_desired;        /* bits RWX */

```

```

26245 {
26246 /* Dado un apuntador a un nodo-i, 'rip' y el acceso deseado, determinar
26247 * si éste se permite, y si no, por qué. La rutina busca el uid del invocador
26248 * en la tabla 'fproc'. Si se permite el acceso, se devuelve OK. Si no,
26249 * se devuelve EACCES.
26250 */
26251
26252 register struct inode *old_rip = rip;
26253 register struct super_block *sp;
26254 register mode_t bits, perm_bits;
26255 int r, shift, test_uid, test_gid;
26256
26257 if (rip->i_mount == I_MOUNT) /* Sist montado en nodo-i. */
26258     for (sp = &super_block[1]; sp < &super_block[NA_SUPEAS] j sp++)
26259         if (sp->s_imount == rip) {
26260             rip = get_inode(sp->s_dev, AOOT_INODE);
26261             break;
26262         } /* if */
26263
26264 /* Aislar los bits rwx pertinentes del modo. */
26265 bits = rip->i_mode;
26266 test_uid = (fs_call == ACCESS ? fp->fp_realuid : fp->fp_effuid);
26267 test_gid = (fs_call == ACCESS ? fp->fp_realgid : fp->fp_effgid);
26268 if (test_uid == SU_UID) {
26269     /* Conceder permiso de leer y escribir. Conceder permiso de búsqueda
26270     * directorios. Conceder permiso ejecución (plno directorios) si
26271     * Y sólo si uno de los bit s 'X' está encendido.
26272     */
26273     if ((bits & I_TYPE) == I_DIAECTOAY 11
26274         bits & ((X_BIT << 6) | (X_BIT << 3) | X_BIT))
26275         perm_bits = A_BIT | W_BIT | X_BIT;
26276     else
26277         perm_bits = A_BIT | W_BIT;
26278 } else {
26279     if (test_uid == rip->i_uid) shift = 6; /* dueño */
26280     else if (test_gid == rip->i_gid) shift = 3; /* grupo */
26281     else shift = 0; /* otro */
26282     perm_bits = (bits >> shift) & (A_BIT | W_BIT | X_BIT);
26283 }
26284
26285 /* Si el acceso deseado no es subconjunto de lo permitido, se niega. */
26286 r = OK;
26287 if ((perm_bits | access_desired) != perm_bits) r = EACCES;
26288
26289 /* Ver si alguien está tratando de escribir en un sistema de archivos que está
26290 * montado sólo para lectura.
26291 */
26292 if (r == OK)
26293     if (access_desired & W_BIT) r = read_only(rip);
26294
26295 if (rip != old_rip) put_inode(rip);
26296
26297     return(r);
26298 }
26301 /*=====
26302 *                               read_only                               *
26303 *=====*/
26304 PUBLIC int read_only(ip)

```

```

26305 struct inode *ip;                                /* apunt a nodo cuyo FS debe verif */
26306 {
26307     /* Ver si el sistema de archivos en el que el nodo-i 'ip' reside está montado
26308     * sólo p/lectura. Si si, devolver EROFS; si no, devolver OK.
26309     */
26310
26311     register struct 'super_block *sp;
26312
26313     sp = ip->i_sp;
26314     return(sp->s_rd_only? EROFS : OK);
26315 }
+++++
src/fs/time.c
+++++
26400 /* Este archivo se ocupa de las llamadas relacionadas con el tiempo.
26401  *
26402  * Los puntos de entrada a este archivo son:
26403  * do_utime: ejecutar la llamada al sistema UTIME
26404  * do_time: ejecutar la llamada al sistema TIME
26405  * do_stime: ejecutar la llamada al sistema STIME
26406  * do_tims: ejecutar la llamada al sistema TIMES
26407  */
26408
26409 #include "fs.h"
26410 #include <minix/callnr.h>
26411 #include <minix/com.h>
26412 #include "file.h"
26413 #include "fproc.h"
26414 #include "inode.h"
26415 #include "param.h"
26416
26417 PRIVATE message clock_mess;
26418
26419 /*=====
26420  * do_utime *
26421 *=====*/
26422 PUBLIC int do_utime()
26423 {
26424     /* Ejecutar la llamada al sistema utime(name, timep). */
26425
26426     register struct inode *rip;
26427     register int len, r;
26428
26429     /* Ajustar para el caso de NULL 'timep'. */
26430     len = utime_length;
26431     if (len == 0) len = m.m2_i2;
26432
26433     /* Abrir temporalmente el archivo. */
26434     if (fetch_name(utime_file, len, M1) != OK) return(err_code);
26435     if ((rip = eat_path(user_path)) == NIL_INODE) return(err_code);
26436
26437     /* Sólo dueño del archivo o superusuario pueden cambiar su tiempo. */
26438     r = OK;
26439     if (rip->i_uid != fp->fp_effuid && !super_user) r = EPERM;

```

```

26500     reply_t2 = t[1];
26501     reply_t3 = t[2];
26502     reply_t4 = t[3];
26503     reply_t5 = t[4];
26504     return(OK);
26505 }
+++++
+++++                               src/fs/misc.c                               +++++
+++++
26600     /* Este archivo contiene procedimientos diversos. Algunos de ellos ejecutan
26601     * llamadas sencillas al sistema. Otros hacen una parte de algunas llamadas
26602     * que el administrador de memoria ejecuta casi en su totalidad.
26603     *
26604     * Los puntos de entrada a este archivo son
26605     *   do_dup:      ejecutar la llamada al sistema DUP
26606     *   do_fcntl:   ejecutar la llamada al sistema FCNTL
26607     *   do_sync:    ejecutar la llamada al sistema SYNC
26608     *   do_fork:    ajustar tablas después de que MM ejecutó FORK
26609     *   do_exec:    manejar archs con FD_CLOEXEC encendido después de EXEC por MM
26610     *   do_exit:    un proceso salió; anotarlos en las tablas
26611     *   do_set:     establecer uid o gid para algún proceso
26612     *   do_revive:  revivir proceso que esperaba algo (p.ej. TTY)
26613     */
26614
26615     #include "fs.h"
26616     #include <fcntl.h>
26617     #include <unistd.h>          /* cc se queda sin memoria con unistd.h : -( */
26618     #include <minix/callnr.h>
26619     #include <minix/com.h>
26620     #include <minix/boot.h>
26621     #include "buf.h"
26622     #include "file.h"
26623     #include "fproc.h"
26624     #include "inode.h"
26625     #include "dev.h"
26626     #include "param.h"
26627
26628
26629 /*=====
26630     *                               do_dup                               *
26631 *=====*/
26632     PUBLIC int do__dup()
26633     {
26634     /* Ejecutar la llamada al sistema dup(fd) o dup2(fd,fd2). Éstas son obsoletas.
26635     * De hecho, ni siquiera podemos invocarlas con la biblioteca actual
26636     * porque las rutinas de biblioteca invocan fcntl(). Se incluyen para que
26637     * programas binarios viejos puedan seguir ejecutándose.
26638     */
26639
26640     register int rfd;
26641     register struct filp *f;
26642     struct filp *dummy;
26643     int r;
26644

```

```

26645     /* ¿Es válido el descriptor de archivo? */
26646     rfd = fd & ~DUP_MASK;          /* apagar bit dup2 si encendido */
26647     if ((f = get_filp(rfd)) == NIL_FILP) return(err_code);
26648
26649     /* Distinguir entre dup y dup2. */
26650     if (fd == rfd) {                /* bit apagado */
26651         /* dup(fd) */
26652         if ( (r = get_fd(0, 0, &fd2, &dummy)) != OK) return(r);
26653     } else {
26654         /* dup2(fd, fd2) */
26655         if (fd2 < 0 || fd2 >= OPEN_MAX) return(EBADF);
26656         if (rfd == fd2) return(fd2);          /* ignorar llamada: dup2(x, x) */
26657         fd = fd2;                            /* preparar p/cerrar fd2 */
26658         (void) do_close();                    /* no puede fallar */
26659     }
26660
26661     /* Éxito. Establecer nuevos descriptores de archivo. */
26662     f->filp_count++;
26663     fp->fp_filp[fd2] = f;
26664     return(fd2);
26665 }
26666 /*=====
26667 *                                     do_fcntl                                     *
26668 *=====*/
26669 PUBLIC int do_fcntl()
26670 {
26671     /* Ejecutar la llamada al sistema fcntl(fd, request, " "). */
26672
26673     register struct filp *f;
26674     int new_fd, r, fl;
26675     long cloexec_mask;                /* mapa bits p/bandera FD_CLOEXEC */
26676     long clo_value;                   /* FD_CLOEXEC en posición correcta */
26677     struct filp *dummy;
26678
26679     /* ¿Es válido el descriptor de archivo? */
26680     if ((f = get_filp(fd)) == NIL_FILP) return(err_code);
26681
26682     switch (request) {
26683     case F_DUPFD:
26684         /* Sustituye a la vieja llamada al sistema dup(). */
26685         if (addr < 0 || addr >= OPEN_MAX) return(EINVAL);
26686         if ((r = get_fd(addr, 0, &new_fd, &dummy)) != OK) return(r);
26687         f->filp_count++;
26688         fp->fp_filp[new_fd] = f;
26689         return(new_fd);
26690
26691     case F_GETFD:
26692         /* Obt bandera cerrar-al-ejec (FD_CLOEXEC en tabla 6-2 POSIX). */
26693         return( ((fp->fp_cloexec >> fd) & 01) ? FD_CLOEXEC : 0);
26694
26695     case F_SETFD:
26696         /* Fijar bandera cerrar-al-ejec (FD_CLOEXEC en tabla 6-2 POSIX). */
26697         cloexec_mask = 1L << fd;      /* posic conjunto solo ok */
26698         clo_value = (addr & FD_CLOEXEC ? cloexec_mask : 0L);
26699         fp->fp_cloexec = (fp->fp_cloexec & ~cloexec_mask) | clo_value;
26700         return(OK);
26701
26702     case F_GETFL:
26703         /* Obtener banderas de situación archivo (O_NONBLOCK y O_APPEND). */
26704

```

```

26705         fl = f->filp_flags & (O_NONBLOCK I O_APPEND I O_ACCMODE);
26706         return(fl);
26707
26708     case F_SETFL:
26709         /* Fijar banderas de situación archivo (O_NONBLOCK y O_APPEND). */
26710         fl = 'O_NONBLOCK I O_APPEND;
26711         f->filp_flags = (f->filp_flags & ~fl) I (addr & fl);
26712         return(OK);
26713
26714     case F_GETLK:
26715     case F_SETLK:
26716     case F_SETLKW:
26717         /* Poner o quitar un candado de archivo. */
26718         r = lock_op(f, request);
26719         return(r);
26720
26721     default:
26722         return(EINVAL);
26723     }
26724 }
26727 /*=====
26728 *                                     do_sync                                     *
26729 *=====*/
26730 PUBLIC int do_sync()
26731 {
26732     /* Ejecutar llamada al sistema sync(). Vaciar todas las tablas. */
26733
26734     register struct inode *rip;
26735     register struct buf *bp;
26736
26737     /* El orden en que se vacían las tablas es crítico. Los bloques deben
26738     * vaciarse al final, ya que rw_inode() deja sus resultados en el caché
26739     * de bloques.
26740     */
26741
26742     /* Escribir todos los nodos-i sucios en disco. */
26743     for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++)
26744         if (rip->i_count > 0 && rip->i_dirt == DIRTY) rw_inode(rip, WRITING);
26745
26746     /* Escribir todos los bloques sucios en disco, una unidad a la vez. */
26747     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
26748         if (bp->b_dev != NO_DEV && bp->b_dirt == DIRTY) flushall(bp->b_dev);
26749
26750     return(OK);          /* sync() no puede fallar */
26751 }
26754 /*=====
26755 *                                     do_fork                                     *
26756 *=====*/
26757 PUBLIC int do_fork()
26758 {
26759     /* Realizar los aspectos de la llamada fork() relacionados con archivos. En
26760     * particular, que el hijo herede los descriptores de archivo de su padre. Los
26761     * parámetros de padre e hijo dicen quién bifurcó a quién. El sistema de archivos
26762     * usa los mismos números de ranura que el kernel. Sólo MM efectúa esta llamada.
26763     */
26764

```



```
26765     register struct fproc *cp;
26766     int i;
26767
26768     /* Sólo MM puede hacer esta llamada directamente. */
26769     if (who != MM_PROC_NR) return(EGENERIC);
26770
26771     /* Copiar estructura fproc del padre al hijo. */
26772     fproc[child] = fproc[parent];
26773
26774     /* Aumentar contadores en la tabla 'filp'. */
26775     cp = &fproc[child];
26776     for (i = 0; i < OPEN_MAX; i++)
26777         if (Cp->fp_filp[i] != NIL_FILP) cp->fp_filp[i] ->filp_count++;
26778
26779     /* Llenar nuevo id de proceso. */
26780     cp->fp_pid = pid;
26781
26782     /* Un hijo no es un jefe de proceso. */
26783     cp->fp_sesldr = 0;
26784
26785     /* Registrar hecho de que dir raíz y de trabajo tienen otro usuario. */
26786     dup_inode(cp->fp_rootdir);
26787     dup_inode(cp->fp_workdir);
26788     return(OK);
26789 }
26792 /*=====
26793 *                               do_exec                               *
26794 *=====*/
26795 PUBLIC int do_exec()
26796 {
26797     /* Podemos marcar archivos con el bit FO_CLOEXEC (en fp->fp_cloexec). Cuando MM
26798     * efectúa un EXEC, llama al FS para que él pueda encontrar estos archivos y los cierre:
26799     */
26800
26801     register int i;
26802     long bitmap;
26803
26804     /* Sólo MM puede hacer esta llamada directamente. */
26805     if (who != MM_PROC_NR) return(EGENERIC);
26806
26807     /* El arreglo de bits FO_CLOEXEC está en el mapa de bits fd_cloexec. */
26808     fp = &fproc[slot1];          /* get_filp() necesita 'fp' */
26809     bitmap = fp->fp_cloexec;
26810     if (bitmap == 0) return(OK); /* caso normal, ningún FO_CLOEXEC */
26811
26812     /* Revisar descriptores de archivo 1 por 1 p/detectar FO_CLOEXEC. */
26813     for (i = 0; i < OPEN_MAX; i++) {
26814         fd = i;
26815         if ((bitmap » i) & 01) (void) do_close();
26816     }
26817
26818     return(OK);
26819 }
26822 /*=====
26823 *                               do_exit                               *
26824 *=====*/
```

```

26825 PUBLIC int do_exit()
26826 {
26827 /* Realizar la porción de la llamada exit(status) que toca al FS. */
26828
26829 register int i, exitee, task;
26830 register struct fproc *rfp;
26831 register struct filp *rfilp;
26832 register struct inode *rip;
26833 int majar;
26834 dev_t dev;
26835 message dev_mess;
26836
26837 /* Sólo el MM puede efectuar la llamada EXIT directamente. */
26838 if (who != MM_PROC_NR) return(EGENERIC);
26839
26840 * No obstante, fingir que la llamada provino del usuario. */
26841 fp = &fproc[slot1]; /* get_filp() necesita 'fp' */
26842 exitee = slot1;
26843
26844 if (fp->fp_suspended == SUSPENDED) {
26845     task = -fp->fp_task;
26846     if (task == XPIPE || task == XOPEN) susp_count--;
26847     pro = exitee;
26848     (void) do_unpause(); /* esto siempre tiene éxito p/MM */
26849     fp->fp_suspended = NOT_SUSPENDED;
26850 }
26851
26852 /* Iterar en descriptores de archivo, cerrando los abiertos. */
26853 for (i = 0; i < OPEN_MAX; i++) {
26854     fd = i;
26855     (void) do_close();
26856 }
26857
26858 /* Liberar directorios raíz y de trabajo. */
26859 put_inode(fp->fp_rootdir);
26860 put_inode(fp->fp_workdir);
26861 fp->fp_rootdir = NIL_INODE;
26862 fp->fp_workdir = NIL_INODE;
26863
26864 /* Si existe un jefe de sesión, revocar acceso a su tty controladora
26865  * desde todos los otros procesos que la usan.
26866  */
26867 if (!fp->fp_sesldr) return(OK); /* no es jefe de sesión */
26868 fp->fp_sesldr = FALSE;
26869 if (fp->fp_tty == 0) return(OK); /* no es tty controladora */
26870 dev = fp->fp_tty;
26871
26872 for (rfp = &fproc[LOW_USER]; rfp < &fproc[NR_PROCS]; rfp++) {
26873     if (rfp->fp_tty == dev) rfp->fp_tty = 0;
26874
26875     for (i = 0; i < OPEN_MAX; i++) {
26876         if ((rfilp = rfp->fp_filp[i]) == NIL_FILP) continue;
26877         if (rfilp->filp_mode == FILP_CLOSED) continue;
26878         rip = rfilp->filp_ino;
26879         if ((rip->i_mode & I_TYPE) != I_CHAR_SPECIAL) continue;
26880         if ((dev_t) rip->i_zone[0] != dev) continue;
26881         dev_mess.m_type = DEV_CLOSE;
26882         dev_mess.DEVICE = dev;
26883         majar = (dev » MAJOR) & BYTE; /* núm disp principal */
26884         task = dmap[llajor].dmap_task; /* núm tarea dispositivo */

```

```

26885             (*dmap[major].dmap_close)(task, &dev_mess);
26886             rfilp->filp_mode = FILP_CLOSED;
26887         }
26888     }
26889     return(OK);
26890 }
26893/*=====
26894     *                               do_set                               *
26895 *===== */
26896 PUBLIC int do_set()
26897 {
26898     /* Establecer campo uid_t o gid_t. */
26899
26900     register struct fproc *tfp;
26901
26902     /* Sólo MM puede hacer esta llamada directamente. */
26903     if (who != MM_PROC_NR) return(EGENERIC);
26904
26905     tfp = &fproc[slot1];
26906     if (fs_call == SETUID) {
26907         tfp->fp_realuid = (uid_t) real_user_idj
26908         tfp->fp_effuid =      (uid_t) eff_user_idj
26909     }
26910     if (fs_call == SETGID) {
26911         tfp->fp_effgid =      (gid_t) eff_grp_idj
26912         tfp->fp_realgid = (gid_t) real_grp_idj
26913     }
26914     return(OK);
26915 }
26918/*=====
26919     *                               do_revive                               *
26920 *===== */
26921 PUBLIC int do_revive()
26922 {
26923     /* Una tarea, usualmente TTY, ya obtuvo los caracteres necesarios p/una
26924     * lectura previa. El proceso no obtuvo respuesta al hacer la llamada,
26925     * sino que se suspendió. Ahora podemos enviar la respuesta para despertarlo.
26926     * Hay que tener cuidado, pues el mensaje entrante viene de una tarea
26927     * (a la que no puede enviarse respuesta), y la respuesta debe ir a un proceso
26928     * que se bloqueó antes. La respuesta al invocador se inhibe izando
26929     * la bandera 'dont_reply', y la respuesta al proceso bloqueado se efectúa
26930     * explícitamente en revive().
26931     */
26932
26933     #if IALLOW_USER_SEND
26934         if (who >= LOW_USER) return(EPERM);
26935     #endif
26936
26937     revive (m.REP_PROC_NR , m.REP_STATUS);
26938     dont_reply = TRUE; /* no responder a tarea TTY */
26939     return(OK);
26940 }

```

```

+++++
                                     src/fs/device.c
+++++
27000 /* Si un bloque requerido no está en el caché, debe traerse de disco. Los archivos
27001 * especial,es por caracteres también requieren E/S. Las rutinas para todo esto están aquí.
27002 *
27003 * Los puntos de entrada a este archivo son:
27004 *   dev_io:      leer o escribir en un disp por bloques o por caracteres
27005 *   dev_opcl:   procesam. de abrir y cerrar genérico específico p/disp
27006 *   tty_open:   procesamiento de abrir específico p/tty
27007 *   ctty_open:  procesam de abrir específico p/tty controladora
27008 *   ctty_close: procesam de cerrar específico p/tty controladora
27009 *   do_setsid:  ejecutar llamada al sistema SETSID (lado del FS)
27010 *   do_ioctl:   ejecutar llamada al sistema IOCTL
27011 *   call_task:  procedim que realmente llama las tareas del kernel
27012 *   call_ctty: procedim que realmente llama tarea para /dev/tty
27013 */
27014
27015 #include "fs.h"
27016 #include <fcntl.h>
27017 #include <minix/callnr.h>
27018 #include <minix/com.h>
27019 #include "dev.h"
27020 #include "file.h"
27021 #include "fproc.h"
27022 #include "inode.h"
27023 #include "param.h"
27024
27025 PRIVATE message dev_mess;
27026 PRIVATE majar, minar, task;
27027
27028 FORWARD_PROTOTYPE( void find_dev, (Dev_t dev)
27029
27030 /*=====
27031 *                               dev_io                               *
27032 *=====*/
27033 PUBLIC int dev_io(op, nonblock, dev, pos, bytes, proc, buff)
27034 int op;                               /* DEV_READ, DEV_WRITE, DEV_IOCTL, etc. */
27035 int nonblock;                         /* TRUE si op no bloqueadora */
27036 dev_t dev;                             /* núm disp principal-secundario */
27037 off_t pos;                             /* posición de byte */
27038 int bytes;                             /* cuántos bytes por transferir */
27039 int proc;                              /* ¿en esp dir de quién está buff? */
27040 char *buff;                            /* dirección virtual del buffer */
27041 {
27042 /* Leer de o escribir en dispositivo; 'dev' indica cuál. */
27043
27044   find_dev(dev);                       /* cargar variables majar, minar y task */
27045
27046 /* Preparar el mensaje que se pasa a la tarea. */
27047   dev_mess.m_type   = op;
27048   dev_mess.DEVICE   = (dev »  MINOR) & BYTE;
27049   dev_mess.POSITION = pos;
27050   dev_mess.PROC_NR  = proc;
27051   dev_mess.ADDRESS  = buff;
27052   dev_mess.COUNT    = bytes;
27053   dev_mess.TTY_FLAGS = nonblock; /* arreglo temporal */
27054

```

```
27055     /* Invocar la tarea. */
27056     (*dmap[major].dmap_rw)(task, &dev_mess);
27057
27058     /* Tarea acabó. Ver si la llamada acabó. */
27059     if (dev_mess.REP_STATUS == SUSPEND) {
27060         if (op == DEV_OPEN) task = XPOPEN;
27061         suspend(task);           /* suspender usuario */
27062     }
27063
27064     return(dev_mess.REP_STATUS);
27065 }
27066
27067 /*=====
27068 *                               dev_opcl                               *
27069 *=====*/
27070
27071 PUBLIC void dev_opcl(task_nr, mess_ptr)
27072 int task_nr;           /* cuál tarea */
27073 message *mess_ptr;    /* apuntador a mensaje */
27074 {
27075     /* Invocada de struct dmap en table.c al abrir o cerrar archs especiales.*/ 27076
27077     int op;
27078
27079     op = mess_ptr->m_type;           /* guardar DEV_OPEN o DEV_CLOSE p/después */
27080     mess_ptr->DEVICE = (mess_ptr->DEVICE » MINOR) & BYTE;
27081     mess_ptr->PROC_NR = fp -fproc;
27082
27083     call_task(task_nr, mess_ptr);
27084
27085     /* Tarea acabó. Ver si llamada acabó. */
27086     if (mess_ptr->REP_STATUS == SUSPEND) {
27087         if (op == DEV_OPEN) task_nr = XPOPEN;
27088         suspend(task_nr);           /* suspender usuario */
27089     }
27090 }
27091
27092 /*=====
27093 *                               tty_open                               *
27094 *=====*/
27095
27096 PUBLIC void tty_open(task_nr, mess_ptr)
27097 int task_nr;
27098 message *mess_ptr;
27099 {
27100     /* Se invoca desde la struct dmap en table.c al abrir una tty. */
27101
27102     int r;
27103     dev_t dev;
27104     int flags, proc;
27105     register struct fproc *rfp;
27106
27107     dev = (dev_t) mess_ptr->DEVICE;
27108     flags = mess_ptr->COUNT;
27109     proc = fp -fproc;
27110
27111     /* Agregar O_NOCTTY a las banderas si este proceso no es jefe
27112      * de sesión, o si ya tiene una tty controladora, o si es la tty
27113      * controladora de alguien más.
27114      */
27115     if (!fp->fp_sesldr || fp->fp_tty != 0) {
```

```

27115         flags |= O_NOCTTY;
27116     } else {
27117         for (rfp = &fproc[LOW_USER]; rfp < &fproc[NR_PROCS]; rfp++) {
27118             if (rfp->fp_tty == dev) flags |= O_NOCTTY;
27119         }
27120     }
27121
27122     r = dev_io(DEV_OPEN, mode, dev, (off_t) 0, flags, proc, NIL_PTR);
27123
27124     if (r == 1) {
27125         ffp->fp_tty = dev;
27126         r = OK;
27127     }
27128
27129     mess_ptr->REP_STATUS = r;
27130 }
27131
27132 I*=====
27133 *                                     ctty_open                                     *
27134 *=====
27135 *=====
27136 PUBLIC void ctty_open(task_nr, mess_ptr)
27137 int task_nr;
27138 message *mess_ptr;
27139 {
27140     I* Este procedimiento se invoca desde la struct dmap en table.c al abrir
27141     * Idev/tty, el disp mágico que traduce a la tty controladora.
27142     *I
27143
27144     mess_ptr->REP_STATUS = ffp->fp_tty == 0 ? ENXIO : OK;
27145 }
27146
27147 I*=====
27148 *                                     ctty close                                     *
27149 *=====
27150 *=====
27151 PUBLIC void ctty_close(task_nr, mess_ptr)
27152 int task_nr;
27153 message *mess_ptr;
27154 {
27155     I* Cerrar Idev/tty. *I
27156
27157     mess_ptr->REP_STATUS = OK;
27158 }
27159
27160 I*=====
27161 * do_setsid *
27162 *=====
27163 *=====
27164 PUBLIC int do_setsid()
27165 {
27166     I* Realizar lado de FS de la llamada SETSID, o sea, deshacerse de la terminal
27167     * controladora de un proceso Y hacer a éste jefe de sesión.
27168     *I
27169     register struct fproc *rfp;
27170
27171     I* Sólo el MM puede efectuar la llamada SETSID directamente. *I
27172     if (who != MM_PROC_NR) return(ENOSYS);
27173
27174     I* Hacer al proceso jefe de sesión sin tty controladora. *I

```

```

27175     rfp = &fproc[slot1];
27176     rfp->fp_sesldr = TRUE;
27177     rfp->fp_tty = 0;
27178 }
27181 /*=====
27182 *                               do_ioctl                               *
27183 *=====*/
27184 PUBLIC int do_ioctl()
27185 {
27186     /* Ejecutar la llamada al sistema ioctl(ls_fd, request, argx) (usa finto m2). */ 27187
27188     struct filp *f;
27189     register struct inode *rip;
27190     dev_t dev;
27191
27192     if ( (f = get_filp(ls_fd) == NIL_FILP) return(err_code);
27193     rip = f->filp_inoj) /* obtener apunt a nodo-i */
27194     if ( (rip->i_mode & I_TYPE) != I_CHAR_SPECIAL
27195         && (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) return(ENOTTY);
27196     dev = (dev_t) rip->i_zone[0];
27197     find_dev(dev);
27198
27199     dev_mess = m;
27200
27201     dev_mess.m_type = DEV_IOCTL;
27202     dev_mess.PROC_NR = who;
27203     dev_mess.TTY_LINE = minar;
27204
27205     /* Invocar la tarea. */
27206     (*dmap[major].dmap_rw)(task, &dev_mess);
27207
27208     /* La tarea acabó. Ver si la llamada acabó. */
27209     if (dev_mess.REP_STATUS == SUSPEND) {
27210         if (f->filp_flags & O_NONBLOCK) {
27211             /* No se supone que se bloquee. */
27212             dev_mess.m_type = CANCEL;
27213             dev_mess.PROC_NR = who;
27214             dev_mess.TTY_LINE = minar;
27215             (*dmap[major].dmap_rw)(task, &dev_mess);
27216             if (dev_mess.REP_STATUS == EINTR) dev_mess.REP_STATUS = EAGAIN;
27217         } else {
27218             suspend(task); /* Debe suspenderse el usuario. */
27219         }
27220     }
27221     return(dev_mess.REP_STATUS);
27222 }
27225 /*=====
27226 *                               find_dev                               *
27227 *=====*/
27228 PRIVATE void find_dev(dev)
27229 dev_t dev; /* dispositivo */
27230 {
27231     /* Extraer núm de disp principal y secundario del parámetro. */
27232
27233     majar = (dev » MAJOR) & BYTE; /* núm dispositivo principal */
27234     minar = (dev » MINOR) & BYTE; /* núm dispositivo secund */

```

```

27235     if (major >= max_major) {
27236         major = minor = 0;          /* fallará con ENODEV */
27237     }
27238     task = dmap[major].dmap_task; /* cuál tarea atiende dispositivo */
27239 }
27242 /*=====
27243 *                               call_task                               *
27244 *=====*/
27245 PUBLIC void call_task(task_nr, mess_ptr)
27246 int task_nr;          /* cuál tarea invocar */
27247 message *mess_ptr;    /* apunt a mensaje p/tarea */
27248 {
27249     /* Toda E/S del FS se reduce a E/S de pares de dispositivo principal/second.
27250     * Éstos conducen a llamadas de las rutinas sigtes vía tabla dmap.
27251     */
27252
27253     int r, proc_nr;
27254     message local_m;
27255
27256     proc_nr = mess_ptr->PROC_NR;
27257
27258     while ((r = sendrec(task_nr, mess_ptr)) == ELOCKED) {
27259         /* sendrec() falló p/evitar bloqueo mortal. La tarea 'task_nr'
27260         * está tratando de enviar mensaje REVIVE p/solicitud previa.
27261         * Manejar e intentar otra vez.
27262         */
27263         if ((r = receive(task_nr, &local_m)) != OK) break;
27264
27265         /* Si tratamos de enviar mensaje de cancelar a una tarea que acaba de enviar
27266         * respuesta de finalización, ignorar respuesta y abortar solicitud de
27267         * cancelar. El invocador revivirá el proceso.
27268         */
27269         if (mess_ptr->m_type == CANCEL && local_m.REP_PROC_NR == proc_nr)
27270             return;
27271
27272         /* Si no, debe ser un REVIVE. */
27273         if (local_m.m_type != REVIVE) {
27274             printf(
27275                 "fs: strange device reply from %d, type = %d, proc = %d\n",
27276                 local_m.m_source,
27277                 local_m.m_type, local_m.REP_PROC_NR);
27278             continue;
27279         }
27280
27281         revive (local_m.REP_PROC_NR, local_m.REP_STATUS);
27282     }
27283
27284     /* El mensaje recibido puede ser una respuesta a esta llamada, o un REVIVE
27285     * para algún otro proceso.
27286     */
27287     for (j;) {
27288         if (r != OK) panic("call_task: can't send/receive", NO_NUM);
27289
27290         /* El proc piel que hicimos sendrec(), ¿obtuvo resultado? */
27291         if (mess_ptr->REP_PROC_NR == proc_nr) break;
27292
27293         /* Si no, debe ser un REVIVE. */
27294         if (mess_ptr->m_type != REVIVE) {

```



```

27295         printf(
27296             "fs: strange device reply from %d, type = %d, proc = %d\n",
27297             mess_ptr->m_source,
27298             mess_ptr->m_type, mess_ptr->REP_PROC_NR);
27299         continue;
27300     }
27301     revive(mess_ptr->REP_PROC_NR, mess_ptr->REP_STATUS);
27302
27303     r = receive(task_nr, mess_ptr);
27304 }
27305 }
27308 /*=====
27309 *                               call_ctty                               *
27310 *=====*/
27311 PUBLIC void call_ctty(task_nr, mess_ptr)
27312 int task_nr;                               /* no se usa -p/compatib con dmap_t */
27313 message *mess_ptr;                         /* apuntador a mensaje p/tarea */
27314 {
27315     /* Esta rutina sólo se invoca p/un dispositivo, /dev/tty. Su trabajo
27316     * es cambiar el mensaje para usar la terminal controladora, en vez del par
27317     * principal/secundario para /dev/tty misma.
27318     */
27319
27320     int major_device;
27321
27322     if (fp->fp_tty == 0) {
27323         /* Ya no hay tty controladora, devolver error de E/S. */
27324         mess_ptr->REP_STATUS = EIO;
27325         return;
27326     }
27327     major_device = (fp->fp_tty » MAJOR) & BYTE;
27328     task_nr = dmap[major_device].dmap_task;           /* tarea p/tty controladora */
27329     mess_ptr->DEVICE = (fp->fp_tty » MINOR) & BYTE;
27330     call_task(task_nr, mess_ptr);
27331 }
27334 /*=====
27335 *                               no_dev                               *
27336 *=====*/
27337 PUBLIC void no_dev(task_nr, m_ptr)
27338 int task_nr;                               /* no se usa -p/compatib con dmap_t */
27339 message *m_ptr;                             /* apuntador a mensaje */
27340 {
27341     /* No hay dispositivo ahí. */
27342
27343     m_ptr->REP_STATUS = ENODEV;
27344 }

```

```

+++++
src/fs/utility.c
+++++
27400 /* Este archivo contiene algunas rutinas de utilidad general.
27401 *
27402 * Los puntos d~ entrada a este archivo son
27403 * clock_time: pedir el tiempo real a la tarea de reloj
27404 * copy:        copiar un bloque de datos
27405 * fetch name: obtener nombre ruta de espacio de usuario
27406 * no_sys:      rechazar llamada al sistema que el FS no maneja
27407 * panic:       algo terrible ocurrió; MINIX no puede continuar
27408 * conv2:       intercambiar bytes en int de 16 bits
27409 * conv4:       intercambiar bytes en long de 32 bits
27410 */
27411
27412 #include "fs.h"
27413 #include <minix/com.h>
27414 #include <minix/boot.h>
27415 #include <unistd.h>
27416 #include "buf.h"
27417 #include "file.h"
27418 #include "fproc.h"
27419 #include "inode.h"
27420 #include "param.h"
27421
27422 PRIVATE int panicking;          /* inhibe pánicos recursivos en sync */
27423 PRIVATE message clock_mess;
27424
27425 /*=====
27426 *                                clock_time                                *
27427 *=====*/
27428 PUBLIC time_t clock_time()
27429 {
27430 /* Esta rutina devuelve el tiempo en segundos desde 1.1.1970. MINIX
27431 * es un sistema astrofísicamente ignorante que supone que la Tierra
27432 * gira con rapidez constante y que no existen los segundos bisiestos.
27433 */
27434
27435     register int k;
27436
27437     clock_mess.m_type = GET_TIME;
27438     if ( (k = sendrec(CLOCK, &clock_mess)) != OK) panic("clock_time err", k);
27439
27440     return( (time_t) clock_mess.NEW_TIME);
27441 }
27442 /*=====
27443 *                                fetch name                                *
27444 *=====*/
27445 PUBLIC int fetch_name(path, len, flag)
27446 {
27447     char *path;          /* apunt a ruta en esp usuario */
27448     int len;             /* longit ruta, incl byte 0 */
27449     int flag;           /* M3 implica ruta quizá en mens */
27450
27451     {
27452 /* Obtener ruta y colocarla en 'user_path'.
27453 * Si 'flag' = M3 Y 'len' <= M3_STRING, la ruta está presente
27454 * en 'message'. Si no, copiarla del espacio de usuario.

```

```

27455  */
27456
27457     register char *rpu, *rpm;
27458     int r;
27459
27460     /* Verificar que la longitud del nombre sea válida. */
27461     if (len <= 0) {
27462         err_code = EINVAL;
27463         return(EGENERIC);
27464     }
27465
27466     if (len > PATH_MAX) {
27467         err_code = ENAMETOOLONG;
27468         return(EGENERIC);
27469     }
27470
27471     if (flag == M3 && len <= M3_STRING) {
27472         /* Sólo copiar ruta del mensaje a 'user_path'. */
27473         rpu = &user_path[0];
27474         rpm = pathname;          /* contenida en mensaje entrada */
27475         do { *rpu++ = *rpm++; } while (--len);
27476         r = OK;
27477     } else {
27478         /* Cadena no contenida en el mensaje. Obtener de espacio usuario. */
27479         r = sys_copy(who, D, (phys_bytes) path,
27480                    FS_PROC_NR, D, (phys_bytes) user_path, (phys_bytes) len);
27481     }
27482     return(r);
27483 }
27486 /*=====
27487  *                               no_sys                               *
27488 *=====*/
27489 PUBLIC int no_sys()
27490 {
27491     /* Alguien usó un número de llamada al sistema ilegal. */
27492
27493     return(EINVAL);
27494 }
27497 /*=====
27498  *                               panic                               *
27499 *=====*/
27500 PUBLIC void panic(format, num)
27501 char *format;          /* cadena de formato */
27502 int num;              /* núm para cadena de formato */
27503 {
27504     /* Algo terrible sucedió. Ocurren pánicos cuando se detecta una inconsistencia
27505     * interna, p. ej., un error de programación o un valor no permitido
27506     * de una constante definida.
27507     */
27508
27509     if (panicking) return;          /* no pánico durante sync */
27510     panicking = TRUE;              /* evitar otro pánico durante sync */
27511     printf("File system panic: %s ", format);
27512     if (num != NO_NUM) printf("%d", num);
27513     printf("\n");
27514     (void) do_sync();              /* vaciar todo al disco */

```

```

27515 sys_abort(RBT_PANIC);
27516 }
27519 I*=====
27520 *                               conv2                               *
27521 *=====I
27522 PUBLIC unsigned conv2(norm, w)
27523 int norm;                               I* TRUE si no intercambio de bytes *I
27524 int W;                                   I* promoción de pal 16 bits por interc *I
27525 {
27526 I* Quizá intercambiar palabra de 16 bits entre orden de bytes 8086 y 68000. *I 27527
27528 if (norm) return( (unsigned) W & 0xFFFF);
27529 return( ((w&BYTE) << 8) | (w>>8) & BYTE);
27530 }
27533 I*=====
27534 *                               conv4                               *
27535 *=====I
27536 PUBLIC long conv4(norm, x)
27537 int norm;                               I* TRUE si no intercambio de bytes *I
27538 long X;                                   I* long de 32 bits por intercambiar *I
27539 {
27540 I* Quizá intercambiar long de 32 bits entre orden de bytes 8086 y 68000. *I
27541
27542 unsigned lo, hi;
27543 long l;
27544
27545 if (norm) return(x) ;                   I* orden bytes ya estaba ok *I
27546 lo = conv2(FALSE, (int) x & 0xFFFF);   I* mitad orden bajo, intercamb *I
27547 hi = conv2(FALSE, (int) (x>>16) & 0xFFFF); I* mitad orden alto, interc *I
27548 l = ( (long) lo <<16) | hi;
27549 return(l);
27550 }
+++++
                               src/fs/putk.c
+++++
27600 I* A veces el FS debe exhibir mensajes. Usa la rutina estándar de biblioteca
27601 * printk(). (El nombre "printf" realmente es una maGro definida como "printk").
27602 * Se exhibe invocando la tarea TTY directamente, no a través de FS.
27603 *I
27604
27605 #include "fs.h"
27606 #include <minix/com.h>
27607
27608 #define BUF_SIZE          100  I* exhibir tamaño de buffer *I
27609
27610 PRIVATE int buf_count;        /* # caracteres en buffer *I
27611 PRIVATE char print_buf[BUF_SIZE]; I* buffer de salida aquí *I
27612 PRIVATE message putch_msg;    I* sirve para mensaje a tarea TTY *I
27613
27614 FORWARD _PROTOTYPE( void flush, (void) );

```


APÉNDICE B

ÍNDICE DE ARCHIVOS

B

INDICE DE ARCHIVOS

Directorio include

01400 a.out.h
00000 ansi.h
00200 errno.h
00900 fcntl.h
04100 ibm/partition.h
00100 limits.h
03700 minix/boot.h
03400 minix/callnr.h
03500 minix/com.h
02600 minix/config.h
02900 minix/const.h
03800 minix/keymap.h
04000 minix/partition.h
03300 minix/syslib.h
03100 minix/type.h
00700 signal.h
01000 stdlib.h
00600 string.h
02400 sys/dir.h
01800 sys/ioctl.h
02200 sys/ptrace.h

02000 sys/sigcontext.h
02300 sys/stat.h
01600 sys/types.h
02500 sys/wait.h
01100 termios.h
00400 unistd.h

Kernel

05500 assert.h
10100 at_wini.c
11000 clock.c
13600 console.c
04300 const.h
14600 dmp.c
09100 driver.c
09000 driver.h
09500 drvlib.c
09400 drvlib.h
07500 exception.c
05000 gl0.h
07600 i8259.c
04200 kernel.h

13000 keyboard.c
08000 klib.s
08100 klib386.s
06700 main.c
09700 memory.c
08800 misc.c
05800 mpx.s
05900 mpx386.s
06900 proc.c
05100 proc.h
07700 protect.c
05200 protect.h
04700 proto.h
05400 sconst.h
06500 start.c
14700 system.c
05600 table.c
11700 tty.c
11600 tty.h
04500 type.h
10000 wini.c

Administrador de memoria

18800 allloc.c
17600 break.c
15900 const.h
17100 exec.c
16800 forkexit.c
18500 getset.c
16200 gl0.h
16600 main.c
15800 mm.h
16300 mproc.h
16400 param.h
16100 proto.h
19300 putk.c
17800 signal.c
16500 table.c
18600 trace.c
16000 type.h
19100 utility.c

Sistema de archivos

20100 buf.h
21000 cache.c
19500 const.h
20200 dev.h
27000 device.c
20300 file.h
22200 filedes.c
20000 fproc.h
19400 fs.h
19900 gl0.h
21500 inode.c
20500 inode.h
25400 link.c
22300 lock.c
20400 lock.h
22500 main.c
26600 misc.c
25100 mount.c
22900 open.c
20600 param.h
24700 path.c
 24300 pipe.c
26100 protect.c
 19700 proto.h
 27600 putk.c
 23400 read.c
25900 stadir.c
21900 super.c
20700 super.h
 20800 table.c
 26400 time.c
 19600 type.h
27400 utility.c
 24000 write.c

APENDICE C

INDICE DE SIMBOLOS

C

INDICE DE SÍMBOLOS

A	3806 #define	APGDN	3839 #define	AVL_286_TSS	5292 #define
A8S	2975 #define	APGUP	3838 #define	A_BLR	1438 #define
ACCESS	3432 #define	APLUS	3842 #define	A_DATAPDS	1453 #define
ACCESSED	5289 #define	ARG_MAX	162 #define	A_DRELPDS	1459 #define
ACTIVE_FLAG	4117 #define	ARG_MA	164 #define	A_EXEC	1445 #define
ADDRESS	3653 #define	ARIGHT	3837 #define	A_HASEXT	1455 #define
ADDWN	3835 #define	ASF1	3921 #define	A_HASLNS	1456 #define
AEND	3833 #define	ASF10	3930 #define	A_HASRELS	1454 #define
AF1	3879 #define	ASF11	3931 #define	A_HASTOFF	1457 #define
AF10	3888 #define	ASF12	3932 #define	A_I80386	1435 #define
AF11	3889 #define	ASF2	3922 #define	A_I8086	1432 #define
AF12	3890 #define	ASF3	3923 #define	A_M68K	1433 #define
AF	3880 #define	ASF4	3924 #define	A_MAGIC0	1426 #define
AF	3881 #define	ASF5	3925 #define	A_MAGIC1	1427 #define
AF4	3882 #define	ASF6	3926 #define	A_MINHDR	1451 #define
AF5	3883 #define	ASF7	3927 #define	A_NONE	1431 #define
AF6	3884 #define	ASF8	3928 #define	A_NS16K	1434 #define
AF	3885 #define	ASF9	3929 #define	A_NSYM	1444 #define
AF	3886 #define	ASKDEV	2784 #define	A_PAL	1443 #define
AF9	3887 #define	ASSERT_H	5504 #define	A_PURE	1447 #define
AHOM	3832 #define	ATARI	2621 #define	A_SEP	1446 #define
AINSRT	3843 #define	ATARI_TYPE	2700 #define	A_SPARC	1436 #define
ALAR M	3428 #define	ATA_IDENTIFY	10164 #define	A_SYMPOS	1460 #define
ALAAM_ON	16348 #define	ATIME	19550 #define	A_TEXTPOS	1452 #define
ALEF	3836 #define	AT_IRQO	10173 #define	A_TOVLY	1448 #define
ALLOW_GAP_MES	2665 #define	AT_IAQ1	10174 #define	A_TRELPOS	1458 #define
ALL_MODES	2986 #define	AT_WINI_IRQ	4365 #define	A_UZP	1442 #define
ALT	3813 #define	AUDIO	3577 #define	A_WLR	1439 #define
AMID	3840 #define	AUDIO_STACK	5667 #define	B0	1185 #define
AMIGA	2622 #define	AUP	3834 #define	B110	1188 #define
ANMI	3841 #define	AUTO_BIOS	2663 #define	B115200	1266 #define
ANY	3504 #define	AVL	5325 #define	B1200	1194 #define

B134	11B9 #define	CDIOAEADTDCHD	1905 #define	CONFOAMING	5284 #define
B150	1190 #define	CDIOAESUME	1910 #define	CONSOLE	13037 #define
81800	1195 #define	CDIOSTOP	1908 #define	CONS_MINOA	11760 #define
819200	1199 #define	CDOWN	3849 #define	CONS_AAM_WOARD	13636 #define
8200	1191 #define	CDAOM	3574 #define	COPAOC_EAA_VE	5304 #define
B2400	1196 #define	CDAOM_STACK	5666 #define	COPAOC_NOT_VE	5252 #define
B300	1192 #define	CENO	3847 #define	COPAOC_SEG_VE	5254 #define
B38400	1200 #define	CF1	3893 #define	COPY_BYTES	3677 #define
B4800	1197 #define	CF10	3902 #define	COAE_MODE	17832 #define
B50	1186 #define	CF11	3903 #define	COUNT	3650 #define
B57600	1265 #define	CF12	3904 #define	COUNTEA_FAEQ	11057 #define
B600	1193 #define	CF2	3894 #define	CPGDN	3853 #define
B75	11B7 #define	CF3	3895 #define	CPGUP	3852 #define
B9600	1198 #define	CF4	3896 #define	CPLUS	3856 #define
BADMAG	142B #define	CF5	3897 #define	CPVEC_NA	2922 #define
BASE_HIGH_SHI	5311 #define	CF6	3898 #define	CAEAD	1141 #define
BASE_MIDDLE_S	5274 #define	CF7	3899 #define	CAEAT	3409 #define
BEEP_FAEQ	13649 #define	CFB	3900 #define	CAIGHT	3851 #define
BEG_PAOC_ADDA	5163 #define	CF9	3901 #define	CS5	1143 #define
BEG_SEAV_ADDA	5166 #define	CHAA_BIT	109 #define	CS6	1144 #define
FIEG_USEA_ADDA	5167 #define	CHAA_MAX	111 #define	CS7	1145 #define
BIG	5324 #define	CHAA_MIN	110 #define	CS8	1146 #define
BIOS_IAQ0_VEC	4349 #define	CHOIA	3413 #define	CSIZE	1142 #define
BIOS_IAQB_VEC	4350 #define	CHILD_MAX	166 #define	CSTOPB	1147 #define
BIOS_VECTOA	4368 #define	CHILD_STIME	3683 #define	CS_INDEX	5213 #define
BITCHUNK_BITS	21920 #define	CHILD_UTIME	3682 #define	CS_LDT_INDEX	5239 #define
BITMAP_CHUNKS	19567 #define	CHIP	2778 #define	CS_SELECTOA	5229 #define
BITS_PEA_BLOC	21921 #define	CHIP	2773 #define	CTIME	19551 #define
BLANK_COLOA	13632 #define	CHIP	2763 #define	CTL_EIGHTHEAD	10166 #define
BLANK_MEM	13635 #define	CHMOD	3416 #define	CTL_INTDISABL	10170 #define
BLOCKS_MINIMU	23820 #define	CHOME	3846 #define	CTL_NOECC	10167 #define
BLOCK_SIZE	2915 #define	CHOWN	3417 #define	CTL_NOAETAY	10166 #define
BMS	2715 #define	CHAOOT	3448 #define	CTL_AESET	10169 #define
BOOT_BLOCK	19560 #define	CINSAT	3857 #define	CTAL	3811 #define
BOOT_TICKS	3884 #define	CLEAN	19548 #define	CUP	3848 #define
BOTH	3503 #define	CLEFT	3850 #define	CUASOA	13646 #define
BOUNDS_VECTOA	5250 #define	CLICK_SHIFT	2959 #define	C_6845	13641 #define
BAEAKPOINT_VE	4337 #define	CLICK_SHIFT	2964 #define	C_EXT	1506 #define
BAK	3418 #define	CLICK_SIZE	2963 #define	C_NULL	1505 #define
BAKINT	1124 #define	CLICK_SIZE	2958 #define	C_STAT	1507 #define
BUF_EXTAA	9132 #define	CLOCAL	1140 #define	CopyMess	6932 #define
BUF_SIZE	27608 #define	CLOCK	3602 #define	O	2928 #define
BUF_SIZE	19309 #define	CLOCK_ACK_BIT	11064 #define	DATA	13644 #define
BUSY_286_TSS	5294 #define	CLOCK_INT	3610 #define	DATA_CHANGED	17622 #define
BYTE	2941 #define	CLOCK_IAQ	4356 #define	DEAF	10209 #define
BYTE_GAAN_MAX	5312 #define	CLOCK_PAOC_NA	3644 #define	DEBUG_VECTOA	4335 #define
BYTE_SWAP	19554 #define	CLOCK_STACK	5672 #define	DEFAULT	5323 #define
B_TIME	13650 #define	GLOSE	3407 #define	DELETE	19545 #define
C	3805 #define	CMD_DIAG	10162 #define	DELTA_TICKS	3641 #define
CA	3807 #define	CMD_FOAMAT	10160 #define	DEL_SCAN	13036 #define
CALL_286_GATE	5295 #define	CMD_IDLE	10155 #define	DESC_386_BIT	5318 #define
CALOCK	3860 #define	CMD_AEAD	10157 #define	DESC_ACCESS	5267 #define
CANCEL	3519 #define	CMD_AEADVEAIF	10159 #define	DESC_BASE	5265 #define
CASCADE_IAQ	4358 #define	CMD_AECALIBAA	10156 #define	DESC_BASE_HIG	5308 #define
CBHD	2717 #define	CMD_SEEK	10161 #define	DESC_BASE_MID	5266 #define
CDIOEJECT	1911 #define	CMD_SPECIFY	10163 #define	DESC_GAANULAA	5307 #define
CDIOPAUSE	1909 #define	CMD_WAIT	10158 #define	DESC_SIZE	5268 #define
CDIOPAYMSS	1904 #define	CMD	3854 #define	DEVICE	3648 #define
CDIOPAYTI	1903 #define	CNMN	3855 #define	DEV_CLOSE	3525 #define
CDIOAEADSUBCH	1907 #define	COLOA_BASE	13628 #define	DEV_FDO	3714 #define
CDIOAEADTOC	1906 #define	COLOA_SIZE	13630 #define	DEV_HDO	3715 #define

DEV_IDCTL	3523 #define	DST_SPACE	3674 #define	ENABLE_WINI	2765 #define
DEV_OPEN	3524 #define	DS_266_INDEX	5216 #define	ENABLE_XT_WIN	2676 #define
DEV_PER_DRIVE	9419 #define	DS_286_SELECT	5232 #define	ENAMETOOLONG	271 #define
DEV_RAM	37,16 #define	DS_INDEX	5210 #define	END	3819 #define
DEV_READ	3521 #define	DS_LDT_INDEX	5240 #define	END_OF_FILE	19557 #define
DEV_SCSI	3717 #define	DS_SELECTOR	5225 #define	END_PROC_ADDR	5164 #define
DEV_WRITE	3522 #define	DT	20900 #define	END_TASK_ADDR	5165 #define
DIOCEJECT	1863 #define	DUMPED	17B33 #define	END_TTY	11771 #define
DIOCGETP	1885 #define	DUMP_SIZE	17834 #define	ENFILE	258 #define
DIOCSETP	1884 #define	DUP	3438 #define	ENOCCONN	293 #define
DIRBLKSIZ	2405 #define	DUP_MAX	19541 #define	ENODEV	254 #define
DIRECTORY_BLO	20161 #define	E2BIG	242 #define	ENOENT	237 #define
DIRSIZ	2408 #define	EACCES	248 #define	ENOEXEC	243 #define
DIRTY	19549 #define	EADDRINUSE	265 #define	ENOLCK	272 #define
DIR_ENTRY_SIZ	19563 #define	EAGAIN	246 #define	ENOMEM	247 #define
DIVIDE_VECTOR	4334 #define	EBADCALL	297 #define	ENOSPC	263 #define
DL_ADDR	3553 #define	E8ADDEST	262 #define	ENOSYS	273 #define
DL_BROAD_REQ	3565 #define	EBADF	244 #define	ENOT8LK	250 #define
DL_CLCK	3552 #define	EBADIOCTL	279 #define	ENOTCONN	291 #define
DL_COUNT	3550 #define	EBADMODE	280 #define	ENOTDIR	255 #define
DL_ETH	3532 #define	EBUSY	251 #define	ENOTEMPTY	274 #define
DL_GETSTAT	3542 #define	ECHILD	245 #define	ENOTTY	260 #define
DL_INIT	3540 #define	ECHO	1153 #define	ENOURG	290 #define
DL_INIT_REPLY	3545 #define	ECHOE	1154 #define	ENTER	19544 #define
DL_MODE	3551 #define	ECHOK	1155 #define	ENXIO	241 #define
DL_MULTI_REQ	3564 #define	ECHONL	1156 #define	EOUOFBUFS	278 #define
DL_NOMODE	3562 #define	ECONNREFUSED	286 #define	EPACKSIZE	277 #define
DL_PACK_RECV	3558 #define	ECONNRESET	287 #define	EPERM	236 #define
DL_PACK_SEND	3557 #define	EDEADLK	270 #define	EPIPE	267 #define
DL_PORT	3548 #define	EDOM	268 #define	EP_OFF	4434 #define
DL_PROC	3549 #define	EDSTNOTRCH	283 #define	EP_ON	4435 #define
DL_PROMISC_RE	3563 #define	EEXIST	252 #define	EP_SET	4436 #define
DL_READ	3538 #define	EFAULT	249 #define	EP_UNSET	4433 #define
DL_READV	3539 #define	EFBIG	262 #define	ERANGE	269 #define
DL_READ_IP	3559 #define	EGA	13642 #define	EROFS	265 #define
DL_STAT	3554 #define	EGA_SIZE	13631 #define	ERR	10189 #define
DL_STOP	3541 #define	EGENERIC	235 #define	ERROR_AC	10149 #define
DL_TASK_REPLY	3546 #define	EICKMANN	2718 #define	ERROR_B8	10146 #define
DL_WRITE	3536 #define	EINTR	239 #define	ERROR_DM	10151 #define
DL_WRITEV	3537 #define	EINVAL	257 #define	ERROR_ECC	10147 #define
DMA_BUF_SIZE	9055 #define	EIO	240 #define	ERROR_ID	10148 #define
DMA_SECTORS	2682 #define	EISCONN	284 #define	ERROR_TK	10150 #define
DONT_SWAP	19555 #define	EISDIR	256 #define	ERR_BAD_SECTO	10190 #define
DOUBLE_FAULT-	5253 #define	ELOCKED	296 #define	ESC	11606 #define
DOWN	3821 #define	EMFILE	259 #define	ESCAPED	11674 #define
DP8390_STACK	5642 #define	EMLINK	266 #define	ESC_SCAN	13033 #define
DPL	5278 #define	EM_BASE	8813 #define	ESHUTDOWN	292 #define
DPL_SHIFT	5279 #define	ENABLE	4384 #define	ESPIPE	264 #define
DP_ETH0_INDEX	5219 #define	ENABLE_ADAPTE	2677 #define	ESRCH	238 #define
DP_ETH0_SELEC	5235 #define	ENABLE_AT_WIN	2673 #define	ES_286_INDEX	5217 #define
DP_ETH1_INDEX	5220 #define	ENABLE_AUDIO	2769 #define	ES_286_SELECT	5233 #define
DP_ETH1_SELEC	5236 #define	ENABLE_BINCOM	2685 #define	ES_INDEX	5211 #define
DSPIOBITS	1917 #define	ENABLE_BIOS_W	2674 #define	ES_SELECTOR	5226 #define
DSPIOMAX	1919 #define	ENABLE_CACHE2	2669 #define	ETHER_IRQ	4359 #define
DSPIORATE	1914 #define	ENABLE_CDROM	2768 #define	ETIMEDOUT	288 #define
DSPIORESET	1920 #define	ENABLE_ESDI_W	2675 #define	ETXTBSY	261 #define
DSPIOSIGN	1918 #define	ENABLE_MITSUM	2678 #define	EURG	289 #define
DSPIOSIZE	1916 #define	ENABLE_NETWORK	2672 #define	EWOULDBLOCK	281 #define
DSPIOSTEREO	1915 #define	ENABLE_SB_AUD	2679 #define	EXDEV	253 #define
DST_BUFFER	3676 #define	ENABLE_SCSI	2767 #define	EXEC	3446 #define
DST_PROC_NR	3675 #define	ENABLE_SRCCOM	2686 #define	EXECUTABLE	5283 #define

EXIT	3402 #define	F_RDLOCK	926 #define	INDEX	13643 #define
EXIT_FAILURE	1008 #define	F_SETFD	915 #define	INDIRECT_8LOC	20162 #define
EXIT_SUCCESS	1009 #define	F_SETFL	917 #define	INET_PROC_NR	2934 #define
EXPAND_DOWN	5285 #define	F_SETLK	919 #define	INITIALIZED	10208 #define
EXT	3810 #define	F_SETLKw	920 #define	INIT_ASSERT	5519 #define
EXTKEY	3814 #define	F_UNLCK	928 #define	INIT_ASSERT	5508 #define
EXT_PART	4125 #define	F_WRLCK	927 #define	INIT_PID	15915 #define
E_BAD_ADDR	310 #define	GA_FONT_SIZE	13658 #define	INIT_PROC_NR	2935 #define
E_BAD_RUF	305 #define	GA_GRAPHICS_D	13656 #define	INIT_PSW	4306 #define
E_BAD_DEST	301 #define	GA_GRAPHICS_I	13655 #define	INIT_SP	4321 #define
E_BAD_FCN	309 #define	GA_SEQUENCER	13654 #define	INIT_TASK_PSW	4307 #define
E_BAD_PROC	311 #define	GA_SEQUENCER-	13653 #define	INLCR	1129 #define
E_BAD_SRC	302 #define	GA_VIDEO_ADDR	13657 #define	INODE_8LOCK	20160 #define
E_NO_MESSAGE	307 #define	GDT_INDEX	5208 #define	INPCK	1130 #define
E_NO_PERM	308 #define	GDT_SELECTOR	5223 #define	INSRT	3829 #define
E_OVERRUN	304 #define	GDT_SIZE	5203 #define	INT2_CTL	4380 #define
E_TASK	306 #define	GETGID	3442 #define	INT2_CTLMASK	4381 #define
E_TRY_AGAIN	303 #define	GETPGRP	3450 #define	INTEL	2754 #define
F1	3865 #define	GETPID	3421 #define	INTR_PrvILEG	5243 #define
F10	3874 #define	GETUID	3425 #define	INT_286_GATE	5297 #define
F11	3875 #define	GET_TIME	3604 #define	INT_CTL	4378 #define
F12	3876 #define	GET_UPTIME	3606 #define	INT_CTLMASK	4379 #define
F2	3866 #define	GRANULAR	5322 #define	INT_GATE_TYPE	7709 #define
F3	3867 #define	GRANULARITY_S	5313 #define	INT_MAX	125 #define
F4	3868 #define	HANGING	16346 #define	INT_MAX	131 #define
F5	3869 #define	HARDWARE	3638 #define	INT_MIN	124 #define
F6	3870 #define	HARDWARE_STAC	5674 #define	INT_MIN	130 #define
F7	3871 #define	HARD_INT	3520 #define	INVAL_OP_VECT	5251 #define
F8	3872 #define	HASCAPS	3815 #define	INVAL_TSS_VEC	5255 #define
F9	3873 #define	HASH_MASK	20168 #define	IN_CHAR	11679 #define
FALSE	2912 #define	HAVE_SCATTERED	2946 #define	IN_EOF	11683 #define
FASTLOAD	2785 #define	HCLICK_SHIFT	4323 #define	IN_EOT	11682 #define
FAST_DISK	2728 #define	HCLICK_SIZE	4324 #define	IN_ESC	11684 #define
FCNTL	3445 #define	HD_CLOCK	2742 #define	IN_LEN	11680 #define
FD_CLOEXEC	923 #define	HIGHEST_ZONE	2995 #define	IN_LSHIFT	11681 #define
FD_MASK	23423 #define	HOME	3818 #define	IN_USE	16344 #define
FILP_CLOSED	20312 #define	HOME_SCAN	13035 #define	IOCTL	3444 #define
FIRST_LDT_IND	5221 #define	HUPCL	1148 #define	IOPL_MASK	14810 #define
FIRST_TTY	11770 #define	HZ	2914 #define	IP_PTR	3695 #define
FLAT_DS_SELEC	5227 #define	IBM_PC	2618 #define	IRQ0_VECTOR	4351 #define
FLOPPY	3593 #define	ICANON	1157 #define	IRQ8_VECTOR	4352 #define
FLOPPY_IRQ	4363 #define	ICD	2716 #define	ISEEK	20544 #define
FLOP_STACK	5670 #define	ICRNL	1125 #define	ISIG	1159 #define
F	3403 #define	ICW1_AT	7608 #define	ISTRIP	1131 #define
FORWARD	2909 #define	ICW1_PC	7609 #define	IS_EMPTY	19546 #define
FPP	2789 #define	ICW1_PS	7610 #define	IXANY	1248 #define
FP_FORMAT	2779 #define	ICW4_AT	7611 #define	IXOFF	1132 #define
FP_FORMAT	2795 #define	ICW4_PC	7612 #define	IXON	1133 #define
FP_IEEE	2760 #define	IDLE	3589 #define	I_8LOCK_SPECI	2980 #define
FP_NONE	2759 #define	IDLE_STACK	5645 #define	I_CHAR_SPECIA	2982 #define
FSTAT	3429 #define	IDLE_STACK	5647 #define	I_DIRECTORY	2981 #define
FSTRUCOPY	4415 #define	IDT_INDEX	5209 #define	I_MOUNT	20542 #define
FS_PROC_NR	2933 #define	IDT_SELECTOR	5224 #define	I_NAMED_PIPE	2983 #define
FULL_DATA_BLO	20165 #define	IDT_SIZE	5204 #define	I_NOT_ALLOC	2991 #define
FUNC	3692 #define	IEXTEN	1158 #define	I_PIPE	20540 #define
FUNC_TO_CALL	3642 #define	IF_MASK	4809 #define	I_REGULAR	2979 #define
F_DUPFD	913 #define	IGNBRK	1126 #define	I_SET_GID_BIT	2985 #define
F_GETFD	914 #define	IGNCR	1127 #define	I_SET_UID_BIT	2984 #define
F_GETFL	916 #define	IGNPAR	1128 #define	I_TYPE	2978 #define
F_GETLK	918 #define	IMAGE_DEV	3709 #define	K8IT	13030 #define
F_OK	417 #define	IMAP	20748 #define	KB_ACK	13025 #define

KB_BUSY	13026 #define	MAX_SECS	10200 #define	NIL_PTR	2945 #define
KB_COMMANO	13020 #define	MAX_SECS	10198 #define	NIL_SUPER	20747 #define
KB_GATE_A20	13021 #define	MAYBE_WRITE I	2800 #define	NKT	5742 #define
KB_IN_BYTES	13042 #define	MAYBE_WRITE_I	2802 #define	NLEN	14999 #define
KB_PULSE_OUTP	13022 #define	MB_CUR_MAX	1011 #define	NLOCK	3861 #define
KB_RESET	13023 #define	MB_LEN_MAX	115 #define	NMIN	3827 #define
KB_STATUS	13024 #define	MEM	3595 #define	NMI_VECTOR	4336 #define
KEYBO	13017 #define	MEMCHECK_AOR 1	3038 #define	NOFLSH	1160 #define
KEYBOARO	2709 #define	MEMCHECK_MAG 1	3039 #define	NORMAL	19530 #define
KEYBOARO IRQ	4357 #define	MEM_BYTES	4412 #define	NOT_ESCAPEO	11673 #define
KEY_MAGIC	3939 #define	MEM_DEV	3598 #define	NOT_REVIVING	20031 #define
KILL	3434 #define	MEM_PTR	3693 #define	NOT_SUSPENDED	20029 #define
KIOCSMAP	1888 #define	MEM_STACK	5671 #define	NO_BIT	19539 #define
KMEM_DEV	3599 #define	MESS_SIZE	3226 #define	NO_BLOCK	2999 #define
KSIG	3453 #define	MID	3826 #define	NO_OEV	3002 #define
K_STACK_BYTES	4400 #define	MILLISEC	11053 #define	NO_ENTRY	3000 #define
K_STACK_BYTES	4304 #define	MIN	2950 #define	NO_MAP	5155 #define
L	3808 #define	MINIX_PART	4122 #define	NO_MEM	15902 #define
LAST_FEW	16823 #define	MINOR	2919 #define	NO_MOUNT	20541 #define
LATCH_COUNT	11058 #define	MINOR_fd0a	9421 #define	NO_NUM	2944 #define
LDH_DEFAULT	10131 #define	MINOR_hdla	9420 #define	NO_PART	4123 #define
LDH_LBA	10132 #define	MIOCGPSINFO	1893 #define	NO_PIPE	20539 #define
LDT	5293 #define	MIOCRAMSIZE	1891 #define	NO_READ	19531 #define
LIYT_SIZE	5205 #define	MIOCSPSINFO	1892 #define	NO_SEEK	20543 #define
LED_COOE	13027 #define	MIXER	3578 #define	NO_ZONE	3001 #define
LEFT	3822 #define	MIXER_STACK	5668 #define	NQ	4429 #define
LEVEL0_VECTOR	4343 #define	MIXIOGETINPUT	1925 #define	NQ	4427 #define
LFUSHO	1257 #define	MIXIOGETINPUT	1924 #define	NR_ACSI_ORIVE	2721 #define
LIMIT_HIGH	5326 #define	MIXIOGETOUTPU	1926 #define	NR_BUFS	2643 #define
LINEWRAP	2664 #define	MIXIOGETVOLUM	1923 #define	NR_BUFS	2648 #define
LINK	3410 #define	MIXIOSETINPUT	1928 #define	NR_BUFS	2653 #define
LINK_MAX	168 #define	MIXIOSETINPUT	1929 #define	NR_BUFS	2658 #define
LOG_MINOR	11761 #define	MIXIOSETOUTPU	1930 #define	NR_BUF_HASH	2659 #define
LONG_MAX	137 #define	MIXIOSETVOLUM	1927 #define	NR_BUF_HASH	2654 #define
LONG_MIN	136 #define	MKOIR	3436 #define	NR_BUF_HASH	2649 #define
LOOK_UP	19543 #define	MKNOD	3415 #define	NR_BUF_HASH	2644 #define
LOW_USER	2937 #define	MM_PROC_NR	2932 #define	NR_CONS	2694 #define
LSEEK	3420 #define	MONO_BASE	13627 #define	NR_DEVICES	10203 #define
M1	3123 #define	MONO_SIZE	13629 #define	NR_DIR_ENTRIE	19564 #define
M3	3124 #define	MON_CS_INDEX	5214 #define	NR_FD_DRIVES	2736 #define
M3_STRING	3126 #define	MON_CS_SELECT	5230 #define	NR_FILPS	19506 #define
M4	3125 #define	MOUNT	3422 #define	NR_HOLES	18820 #define
M68000	2755 #define	MTIME	19552 #define	NR_INODES	19507 #define
MACHINE	2616 #define	MTIOCGET	1897 #define	NR_IOREQS	2923 #define
MACINTOSH	2623 #define	MTIOCTOP	1896 #define	NR_IRQ_VECTOR	4355 #define
MAJOR	2918 #define	M_6845	13640 #define	NR_LOCKS	19509 #define
MAP_BLOCK	20163 #define	NAME_MAX	171 #define	NR_MEMS	4387 #define
MAP_COLS	3934 #define	NAME_PTR	3694 #define	NR_MEMS	4403 #define
MAX	2949 #define	NCALLS	3400 #define	NR_PARTITIONS	4118 #define
MAX_286_SEG_S	5271 #define	NCCS	1109 #define	NR_PROCS	639 #define
MAX_BLOCK_NR	2994 #define	NEW_TIME	3643 #define	NR_PTYS	2696 #define
MAX_CANON	169 #define	NGROUPS_MAX	160 #define	NR_RAM	9719 #define
MAX_DRIVES	10196 #define	NIL_BUF	20139 #define	NR_REGS	4406 #define
MAX_ERRORS	10202 #define	NIL_DEV	9036 #define	NR_RS_LINES	2695 #define
MAX_ESC_PARMS	13637 #define	NIL_FILP	20314 #define	NR_SCAN_CODES	3935 #define
MAX_FILE_POS	2722 #define	2997 #define	NIL_HOLE	18821 #define	NR SCSI_DRIVE
MAX_INODE_NR	2996 #define	NIL_INODE	20536 #define	NR_SEGS	2926 #define
MAX_INPUT	170 #define	NIL_MESS	3227 #define	NR_SUBOEVS	10205 #define
MAX_KB_ACK_REI	3028 #define	NIL_MPROC	16354 #define	NR_SUPERS	19508 #define
MAX_KB_BUSY_R	13029 #define	NIL_MPROC	18628 #define	NR_TASKS	2953 #define
MAX_PAGES	15910 #define	NIL_PROC	5169 #define	NULL	2921 #define

NUII	1006 #define	O_RDWR	943 #define	RECOVERYTIME	10207define
NUII	607 #define	O_TRUNC	934 #define	REG_BASEO	10122 #define
NUII	445 #define	O_WRONIY	942 #define	REG_BASEI	10123 #define
NUII_DEV	3600 #define	PAGE_FAUIT_VE	5303 #define	REG_COMMAND	10154 #define
NUII_MAJOR	3596 #define	PAGE_GRAN_SHI	5315 #define	REG_COUNT	10126 #define
NWIOGETHOPT	1858 #define	PAGE_SIZE	15909 #define	REG_CTI	10165 #define
NWIOGETHSTAT	1859 #define	PARENB	1149 #define	REG_CYI_HI	10129 #define
NWIOGIPCONF	1862 #define	PARMRK	1134 #define	REG_CYI_IO	10128 #define
NWIOGIPOPT	1864 #define	PARODD	1150 #define	REG_DATA	10124 #define
NWIOGTCPCONF	1871 #define	PARTIAL_DATA-	20166 #define	REG_ERROR	10145 #define
NWIOGTCPOPT	1877 #define	PARTITIONING	2713 #define	REG_IDH	10130 #define
NWIOGUDPOPT	1880 #define	PART_TABIE_OF	4119 #define	REG_PRECOMP	10125 #define
NWIOIPDROUTE	1868 #define	PAR_PRINTER	2739 #define	REG_SECTOR	10127 #define
NWIOIPGROUTE	1866 #define	PATH_MAX	172 #define	REG_STATUS	10136 #define
NWIOIPSRROUTE	1867 #define	PAUSE	3430 #define	RENAME	3435 #define
NWIOSETHOPT	1857 #define	PAUSED	16347 #define	REP_PROC_NR	3667 #define
NWIOSIPCONF	1861 #define	PCR	4390 #define	REP_STATUS	3668 #define
NWIOSIPOPT	1863 #define	PENDING	5158 #define	REOUEST	3651 #define
NWIOSTCPCONF	1870 #define	PGDN	3825 #define	REVIVE	3455 #define
NWIOSTCPOPT	1876 #define	PGUP	3824 #define	REVIVING	20032 #define
NWIOSUDPOPT	1879 #define	PID	3688 #define	RIGHT	3823 #define
NWIOTCPATTACH	1874 #define	PIPE	3439 #define	RMDIR	3437 #define
NWIOTCPCONN	1872 #define	PIPE_BUF	173 #define	ROBUST	2636 #define
WIOTCPIISTEN	1873 #define	PIPE_DEV	2689 #define	ROOT_DEV	3708 #define
NWIOTCPSHUTDO	1875 #define	PIPE_SIZE	19566 #define	ROOT_INODE	19559 #define
NW_CANCEL	3572 #define	PLUS	3828 #define	RPI	5262 #define
NW_CIOSE	3568 #define	PORT_B	4391 #define	RS232_IRO	4361 #define
NW_IOCTI	3571 #define	POSITION	3652 #define	RS232_MINOR	11762 #define
NW_OPEN	3567 #define	PR	3690 #define	RUNNING	11675 #define
NW_READ	3569 #define	PREFETCH	19532 #define	RWX_MODES	2987 #define
NW_WRITE	3570 #define	PRESENT	5277 #define	R_ABBS	1470 #define
N_ABS	1497 #define	PRINTER	3591 #define	R_BIT	2988 #define
N_BSS	1500 #define	PRINTER_IRO	4364 #define	R_KBRANCHE	1478 #define
N_CIASS	1504 #define	PRINTER_STACK	5650 #define	R_OK	420 #define
N_COMM	1501 #define	PROC1	3686 #define	R_PCRBYTE	1472 #define
N_DATA	1499 #define	PROC2	3687 #define	R_PCRIONG	1476 #define
N_SECT	1495 #define	PROC_H	5101 #define	R_PCRWORD	1474 #define
N_TEXT	1498 #define	PROC_NR	3649 #define	R_REI3BYTE	1477 #define
N_UNDF	1496 #define	PROTECTION_VE5	258 #define	R_REI1BYTE	1471 #define
OFFSET_HIGH_S	5314 #define	PROTO_H	4703 #define	R_REI1ONG	1475 #define
OK	225 #define	PTRACE	3427 #define	R_REIWORD	1473 #define
OID_MINIX_PAR	4124 #define	PTYPX_MINOR	11764 #define	S	2929 #define
ONE_SHOT	2018 #define	P_FIOPPY	9422 #define	SAPETY_BYTES	17693 #define
ONICR	1252 #define	P_PRIMARY	9423 #define	SAFETY_CIIKS	17694 #define
ONDEar	1254 #define	P_SNOT_FREE	5154 #define	SAME	25422 #define
OPEN	3406 #define	P_STOP	5160 #define	SA_NOCIDSTOP	782 #define
OPEN_MAX	167 #define	P_SUB	9424 #define	SA_NOCIDWAIT	781 #define
OPOST	1137 #define	RAM_DEV	3597 #define	SA_NODEFER	778 #define
OPTIONAL_IO	3529 #define	RAND_MAX	1010 #define	SA_ONSTACK	776 #define
OS_RELEASE	2604 #define	RBT_HAIT	437 #define	SA_RESETHANO	777 #define
OS_VERSION	2605 #define	RBT_MONITOR	440 #define	SA_RESTART	779 #define
OVERFIOW_VECT	4338 #define	RBT_PANIC	439 #define	SA_SIGINFO	780 #define
O_ACCMODE	946 #define	RBT_RE800T	438 #define	SCATTERED_IO	3526 #define
O_APPEND	937 #define	RBT_RESET	441 #define	SCHAR_MAX	113 #define
O_CREAT	931 #define	READ	3404 #define	SCHAR_MIN	112 #define
O_EXCI	932 #define	READABIE	5286 #define	SCHED_RATE	11054 #define
O_NOCTTY	933 #define	READING	2942 #define	SCIOCCMD	1900 #define
O_NOCTTY	11608 #define	REAL_TIME	3609 #define	SCREEN	2706 #define
O_NON BLOCK	11609 #define	REBOOT	3465 #define	SCROIL_DOWN	13634 #define
O_NONBLOCK	938 #define	RECEIVE	3502 #define	SCROLL_UP	13633 #define
O_RDONLY	941 #define	RECEIVING	5157 #define	SCSI	3581 #define

SCSI_STACK	5659 #define	SIGNAL	3443 #define	STDDUT_FILENO	432 #define
SCSI_STACK	5663 #define	SIGNAL	3691 #define	STE -	2702 #define
SC_NOREGLOCAL	2140 #define	SIGPENDING	3461 #define	STIME	3426 #define
SC_SIGCONTEXT	2149 #define	SIGPIPE737 #define	STOPPED	16351 #define	
SECONDARY_IRO	4360 #define	SIGPROCMASK	3462 #define	STOPPED	11676 #define
SECONDS_LEFT	3645 #define	SIGQUIT	726 #define	STREAM_MAX	174 #define
SECTOR_MASK	9052 #define	SJGRETURN	3463 #define	SUB_PER_DRIVE	10204 #define
SECTOR_SHIFT	9051 #define	SIGSEGV735 #define	SUN_4	2619 #define	
SECTOR_SIZE	9050 #define	SIGSTOP749 #define	SUN_4_60	2620 #define	
SEEK_CUR	424 #define	SIGSUSPEND	3460 #define	SUPER_BLOCK	19581 #define
SEEK_END	425 #define	SIGSUSPENDED	1632 #define	SUPER_MAGIC /	9518 #define
SEEK_SET	423 #define	SIGTERM	739 #define	SUPER_REY	19519 #define
SEGMENT	5280 #define	SIGTRAP	728 #define	SUPER_SIZE	19565 #define
SEG_NOT_VECTO	5256 #define	SIGTSTP	750 #define	SUPER_USER	2916 #define
SEN0	3501 #define	SIGTTIN	751 #define	SUPER_V2	19520 #define
SENDING	5156 #define	SIGTTOU	752 #define	SUPER_V2_REV	19521 #define
SEPARATE	16349 #define	SIGUNUSED	731 #define	SUPRA	2714 #define
SERVER_O	4422 #define	SIGUSR1	734 #define	SUSPEND	3530 #define
SETGID	3441 #define	SIGUSR2	736 #define	SUSPENDED	20030 #define
SETPSW	4309 #define	SIG_8LOCK	785 #define	SU_UID	19527 #define
SETPSW	4409 #define	SIG_CATCH	766 #define	SYNC	3433 #define
SETSID	3449 #define	SIG_CTXT_PTR	3699 #define	SYN_ALARM_5TAC	5640 #define
SETUID	3424 #define	SIG:::DFL	763 #define	SYN_ALARM_TASK	3587 #define
SET_ALAAM	3603 #define	SIG_ERR	762 #define	SYS386_VECTOR	4342 #define
SET_SYNC_AL	3607 #define	SIG_HOLD	765 #define	SYSTASK	3615 #define
SET_TIME	3605 #define	SIG_IGN	764 #define	SYSTEM_TIME	3681 #define
SF1	3907 #define	SIG_INQUIRE	788 #define	SYS_ABORT	3624 #define
SF10	3916 #define	SIG_MAP	3697 #define	SYS_COPY	3621 #define
SF11	3917 #define	SIG_MSG_PTR	3698 #define	SYS_ENDSIG	3635 #define
SF12	3918 #define	SIG_PENDING	5159 #define	SYS_EXEC	3622 #define
SF2	3908 #define	SIG_PROC	3696 #define	SYS_FORK	3619 #define
SF3	3909 #define	SIG_SETMASK	787 #define	SYS_FRESH	3625 #define
SF4	3910 #define	SIG_UNBLOCK	786 #define	SYS_GBOOT	3627 #define
SF5	3911 #define	SLASH_SCAN	13034 #define	SYS_GETMAP	3636 #define
SF6	3912 #define	BLOCK	3862 #define	SYS_GETSP	3617 #define
SF7	3913 #define	SMALL_STACK	5637 #define	SYS_GID	19529 #define
SF8	3914 #define	SMART	10210 #define	SYS_KILL	3626 #define
SF9	3915 #define	SPARC	2756 #define	SYS_MEM	3629 #define
SHADOWING	2764 #define	SQUARE_WAVE	11059 #define	SYS_NEWMAP	3620 #define
SHADOWING	2774 #define	SRC_BUFFER	3673 #define	SYS_OLDSIG	3618 #define
SHADOWING	2780 #define	SRC_PROC_NR	3672 #define	SYS_SENDSIG	3633 #define
SHADOWING	2791 #define	SRC_SPACE	3671 #define	SYS_SIGRETURN	3634 #define
SHADOW_BASE	8814 #define	SS_IZE_MAX	176 #define	SYS_STACK	5673 #define
SHADOW_MAX	8815 #define	SS_INDEX	5212 #define	SYS_TIMES	3623 #define
SHADOW_O	4426 #define	SS_SELECTOR	5228 #define	SYS_TRACE	3630 #define
SHIFT	3812 #define	ST	2701 #define	SYS_UID	19528 #define
SHRT_MAX	119 #define	STACK_CHANGED1	7623 #define	SYS_UMAP	3628 #define
SHRT_MIN	118 #define	STACK_FAULT_V	5257 #define	SYS_VCOPY	3631 #define
SIGABRT	729 #define	STACK_GUARD	5151 #define	SYS_VECTOR	4341 #define
SIGACTION	3459 #define	STACK_PTR	3689 #define	SYS_XIT	3616 #define
SIGALRM	738 #define	STAT	3419 #define	S_ABS	1481 #define
SIGBUS	742 #define	STATUS_BSY	10137 #define	S_BSS	1484 #define
SIGCHLD	747 #define	STATUS_CRD	10142 #define	S_DATA	1483 #define
SIGCONT	748 #define	STATUS_ORO	10141 #define	S_IFBLK	2328 #define
SIGEMT	741 #define	STATUS_ERR	10144 #define	S_IFCHR	2330 #define
SIGFPE	732 #define	STATUS_IDX	10143 #define	S_IFDIR	2329 #define
SIGHUP	724 #define	STATUS_RDY	10138 #define	S_IFIFO	2331 #define
SIGILL	727 #define	STATUS_SC	10140 #define	S_IFMT	2326 #define
SIGINT	725 #define	STATUS_WF	10139 #define	S_IFREG	2327 #define
SIGIOT	730 #define	STDERR_FILENO	433 #define	S_IRGRP	2344 #define
SIGKILL	733 #define	STDIN_FILENO	431 #define	S_IROTH	2349 #define

S_IRUSR	2339 #define	TINPUT_DEF	1271 #define	T_SETINS	2212 #define
S_IRWXG	2343 #define	TINTR_DEF	1279 #define	T_SETUSER	2214 #define
S_IRWXO	2348 #define	TIOCGETC	1852 #define	T_STEP	2217 #define
S_IRWXU	2338 #define	TIOCGETP	1850 #define	T_STOP	2207 #define
S_IS8LK	2357 #define	TIOCGPGRP	1846 #define	UCHAR_MAX	114 #define
S_ISCHR	2356 #define	TIOCGWINSZ	1844 #define	UINT_MAX	132 #define
S_ISDIR	2355 #define	TIOCSETC	1853 #define	UINT_MAX	126 #define
S_ISFIFO	2358 #define	TIOCSETP	1851 #define	ULONG_MAX	138 #define
S_ISGID	2333 #define	TIOCSFON	1848 #define	UMASK	3447 #define
S_ISREG	2354 #define	TIOCSPGRP	1847 #define	UMOUNT	3423 #define
S_ISUID	2332 #define	TIOCSWINSZ	1845 #define	UNLINK	3411 #define
S_ISVTX	2335 #define	TKILL_DEF	1280 #define	UNPAUSE	3454 #define
S_IWGRP	2345 #define	TLNEXT_DEF	1288 #define	UP	3820 #define
S_IWOTH	2350 #define	TLOCAL_DEF	1273 #define	USER_PRIVILEG	5245 #define
S_IWUSR	2340 #define	TMIN_DEF	1281 #define	USER_O	4423 #define
S_IXGRP	2346 #define	TOSTOP	1161 #define	USER_TIME	3680 #define
S_IXOTH	2351 #define	TOT_STACK_SPA	5677 #define	USHRT_MAX	120 #define
S_IXUSR	2341 #define	TOUTPUT_DEF	1272 #define	UTIME	3431 #define
S_TEXT	1482 #define	TOUIT_DEF	1282 #define	V1	19523 #define
T	2927 #define	TRACE8IT	4408 #define	V1_INDIRECTS	19572 #define
TA8_MASK	11604 #define	TRACE8IT	4308 #define	V1_INODES_PER	19573 #define
TA8_SIZE	11603 #define	TRACED	16350 #define	V1_INODE_SIZE	19571 #define
TAPE_STAT0	3663 #define	TRAP_286_GATE	5298 #define	V1_NR_DZONES	19501 #define
TAPE_STAT1	3664 #define	TREPRINT_DEF	1287 #define	V1_NR_TZONES	19502 #define
TASK_GATE	5296 #define	TRUE	2911 #define	V1_ZONE_NUM_S	19570 #define
TASK_PRIVILEG5244	#define	TR_ADDR	15463 #define	V2	19524 #define
TASK_O	4421 #define	TR_DATA	15464 #define	V2_INDIRECTS	19578 #define
TASK_REPLY	3456 #define	TR-PROCNR	15461 #define	V2_INODES_PER	19579 #define
TCDRAIN	1841 #define	TR_REOUEST	15462 #define	V2_INODE_SIZE	19577 #define
TCFLOW	1842 #define	TR_VLSIZE	15465 #define	V2_NR_DZONES	19503 #define
TCFLSH	1843 #define	TSPEED_DEF	1274 #define	V2_NR_TZONES	19504 #define
TCGETS	1836 #define	TSS3_SSP0	5964 #define	V2_ZONE_NUM_S	19576 #define
TCIFLUSH	1208 #define	TSS_8USY	5288 #define	VDISCARD	1262 #define
TCIOFF	1215 #define	TSS_INDEX	5215 #define	VECTOR	4370 #define
TCIOFLUSH	1210 #define	TSS_SELECTOR	5231 #define	VEOF	1164 #define
TCION	1216 #define	TSS_TYPE	7710 #define	VEOL	1165 #define
TCOFLUSH	1209 #define	TSTART_DEF	1283 #define	VERASE	1166 #define
TCOOFF	1213 #define	TSTOP_DEF	1284 #define	VIDEO_INDEX	5218 #define
TCOON	1214 #define	TSUSP_DEF	1285 #define	VIDEO_SELECTO	5234 #define
TCSADRAIN	1204 #define	TT	2703 #define	VID_ORG	13645 #define
TCSAFLUSH	1205 #define	TTIME_DEF	1286 #define	VINTR	1167 #define
TCSANOW	1203 #define	TTY	3517 #define	VKILL	1168 #define
TCS8RK	1840 #define	TTYPX_MINOR	11763 #define	VLNEXT	1261 #define
TCSETS	1837 #define	TTY_EXIT	3528 #define	VMIN	1169 #define
TCSETSF	1839 #define	TTY_FLAGS	3659 #define	VOUIT	1170 #define
TCSETSW	1838 #define	TTY_IN_8YTES	11602 #define	VREPRINT	1260 #define
TCTRL_DEF	1270 #define	TTY_LINE	3656 #define	VSTART	1173 #define
TDISCARD_DEF1289	#define	TTY_PGRP	3660 #define	VSTOP	1174 #define
TEOF_DEF	1276 #define	TTY_REOUEST	3657 #define	VSUSP	1172 #define
TEOL_DEF	1277 #define	TTY_SETPGRP	3527 #define	VT100	2710 #define
TERASE_DEF	1278 #define	TTY_SPEK	3658 #define	VTIME	1171 #define
TI	5261 #define	TTY_STACK	5639 #define	WAIT	3408 #define
TIME	3414 #define	TYPE_H	4501 #define	WAITING	16345 #define
TIMEOUT	10206 #define	TZNAME_MAX	175 #define	WAITPID	3412 #define
TIMER0	4392 #define	T_EXIT	2216 #define	WAKEUP	10193 #define
TIMER2	4393 #define	T_GETDATA	2210 #define	WEXITSTATUS	2525 #define
TIMER_COUNT11061	#define	T_GETINS	2209 #define	WIFEXITED	2524 #define
TIMER_FREQ11062	#define	T_GETUSER	2211 #define	WIFSIGNALED	2527 #define
TIMER_MODE	4394 #define	T_OK	2208 #define	WIFSTOPPED	2528 #define
TIMES	3440 #define	T_RESUME	2215 #define	WINCHESTER	3584 #define
TIME_NEVER	11687 #define	T_SETDATA	2213 #define	WINCH_STACK	5655 #define

WINCH_STACK	5653 #define	_NO_LIMIT	158 #define	_STAT_H	2306 #define
WINI_0_PARM_V	4374 #define	_NSIG	722 #define	_STOLIB_H	1003 #define
WINI_1_PARM_V	4375 #define	_PARTITION_H	4102 #define	_STRING_H	605 #define
WNOHANG	2521 #define	_PC_CHOWN_RES	469 #define	_SYSLIB_H	3303 #define
WRITE	3405 #define	_PC_LINK_MAX	461 #define	_SYSTEM	4205 #define
WRITEABLE	5287 #define	_PC_MAX_CANON	462 #define	_SYSTEM	15805 #define
WRITE_IMMED	20157 #define	_PC_MAX_INPUT	466 #define	SYSTEM	19405 #define
WRITING	2943 #define	_PC=NAME_MAX	464 #define	_SYS_KEYMAP	3803 #define
WSTOPSIG	2529 #define	_PC_NO_TRUNC	467 #define	TABLE	5625 #define
WTERMSIG	2526 #define	_PC_PATH_MAX	465 #define	_TABLE	16504 #define
WUNTRACED	2522 #define	_PC_PIPE_BUF	466 #define	_TABLE	20804 #define
W_BIT	2989 #define	_PC_VDISABLE	468 #define	_TERMIOS_H	1103 #define
W_OK	419 #define	_POSIX_ARG_MAX	142 #define	_TIME_T	1634 #define
XLOCK	19536 #define	_POSIX_CHILD-	143 #define	_TYPES_H	1607 #define
XOPEN	19535 #define	_POSIX_CHOWN-	481 #define	_TYPE_H	3101 #define
XPIPE	19534 #define	_POSIX_LINK_M	144 #define	_UNISTD_H	403 #define
XPOPEN	19537 #define	_POSIX_MAX_CA	145 #define	_VOID	50 #define
XTABS	1253 #define	_POSIX_MAX_IN	146 #define	_VOID	38 #define
XT_WINI_IRO	4362 #define	_POSIX_NAME_M	147 #define	_VOIDSTAR	37 #define
X_BIT	2990 #define	_POSIX_NGROUP	148 #define	_VOIDSTAR	49 #define
X_OK	418 #define	_POSIX_NO_TRU	480 #define	_VOLATILE	52 #define
ZMAP	20749 #define	_POSIX_OPEN_M	149 #define	_VOLATILE	40 #define
ZUPER_BLOCK	20164 #define	_POSIX_PATH_M	150 #define	_WAIT_H	2516 #define
_ANSI	24 #define	_POSIX_PIPE_B	151 #define	_WCHAR_T	1023 #define
_ANSI	28 #define	_POSIX_SOURCE	19403 #define	_WORD_SIZE	2627 #define
_ANSI_H	21 #define	_POSIX_SOURCE	15803 #define	_sighandler -	756 #define
_AOUT_H	1403 #define	_POSIX_SOURCE	4203 #define	acc_time	20601 #define
_RGS	35 #define	_POSIX_SIZE	154 #define	addr	16401 #define
_ARGS	47 #define	_POSIX_STREAM	152 #define	addr	20602 #define
_BOOT_H	3703 #define	_POSIX_TZNAME	153 #define	adjust	17661 PUBLIC
_CLOCK_T	1639 #define	_POSIX_VDISAB	1176 #define	advance	249855 PUBLIC
_CONFIG_H	2601 #define	_POSIX_VERSID	428 #define	alloc_bit	21926 PUBLIC
_CONST	39 #define	_PROTOTYPE	34 #define	alloc_inode	21605 PUBLIC
CONST	51 #define	_PROTOTYPE	46 #define	alloc_mem	18840 PUBLIC
_DIR_H	2403 #define	_PROTOTYPE	20B17 PUBLIC	alloc_segment	15715 PUBLIC
ERRNO_H	220 #define	_PTRACE_H	2205 #define	alloc_zone	21180 PUBLIC
_FCNTL_H	910 #define	_SC_ARG_MAX	448 #define	allowed	19120 PUBLIC
_HIGH	2519 #define	_SC_CHILD_MAX	449 #define	at1	13044 PRIVATE
_101	828 #define	_SC_CLK_TCK	451 #define	alt2	13045 PRIVATE
_101	818 #define	_SC_CLOCKS_PE	450 #define	ansi_colors	13703 PRIVATE
_IOCPARM	1811 #define	_SC_JOB_CONTR	454 #define	assert	5513 #define
_IOCTL_H	183 #define	_SC_NGROUPS_M	452 #define	assert	5521 #define
_IOCTYPE	1813 #define	_SC_OPEN_MAX	453 #define	at_winchester	10294 PUBLIC
_IOC_IN	1814 #define	_SC_SAVED_IDS	455 #define	audio_task	5 688 #define
_IOC_INOUT	1816 #define	_SC_STREAM_MA	457 #define	b	20125 EXTERN
_IOC_OUT	1815 #define	_SC_TZNAME_MA	4598 #define	b_bitmap	20148 #define
_IOC_VOID	1812 #define	_SC_VERSION	456 #define	b_data	20142 #define
_IOR	1829 #define	_SIGCONTEXT_H	2001 #define	b_dir	20143 #define
_IOR	1819 #define	_SIGN	224 #define	b_vl_ind	20144 #define
_IORW	1823 #define	_SIGN	227 #define	b_vl_inO	20146 #define
_IORW	1831 #define	_SIGNAL_H	706 #define	b_v2_ind	20145 #define
_IOW	1830 #define	SIGSET_T	717 #define	b_v2_inO	20147 #define
_IOW	1821 #define	_SIGSET_T	1644 #define	back_over	12607 PRIVATE
_LIMITS_H	106 #define	_SIZET	41 #define	bad_assertion	8935 PUBLIC
_LOW	2518 #define	_SIZET	53 #define	bad_compare	8947 PUBLIC
_MINIX	4204 #define	_SIZE_T	1624 #define	beep	14300 PRIVATE
_MINIX	15804 #define	_SIZE_T	1018 #define	beeping	13671 PRIVATE
_MINIX	19404 #define	_SIZE_T	610 #define	bill_ptr	5191 EXTERN
_MINIX	3103 #define	_SIZE_T	407 #define	blank_color	13664 #define
_MINIX_PARTI	4006 #define	_SSIZE_T	412 #define	boot_paramete	15403 PUBLIC
_NERROR	233 #define	_SSIZE_T	1629 #define	boot_paramete	22706 PUBLIC

boot_time	11068 PAIVATE	cons_write	13729 PAIVATE	do_lseek	23367 PUBLIIC
bp_cilbyte	9414 #define	control	1304B PAIVATE	do_mem	15424 PAIVATE
bp_cylinders	9409 #define	conv2	27522 PUBLIIC	do_mkdir	23226 PUBLIIC
bp_heads	9410 #define	conv4	27536 PUBLIIC	do_mkknod	23205 PUBLIIC
bp_landingzon	9415 #define	core_sset	16224 EXTEAN	do_mm_exit	16912 PUBLIIC
bp_max_ecc	9413 #define	cproc_addr	5180 #define	do_mount	25126 PUBLIIC
bp_precomp	9412 #define	cstart	6524PUBLIIC	do_newmap	14921 PAIVATE
bp_reduced_wr	9411 #define	ctty_close	27151 PUBLIIC	do_nop	9312 PUBLIIC
bp_sectors	9416 #define	ctty_open	27136 PUBLIIC	do_open	12171 PAIVATE
buf_count	27610 PAIVATE	curcons	13697 PAIVATE	do_open	22951 PUBLIIC
buf_count	19311 PAIVATE	current	5033 EXTEAN	do_pause	18115 PUBLIIC
buf_hash	20150 EXTEAN	data	16416 #define	do_pipe	24332 PUBLIIC
bUf_pool	22679 PAIVATE	data_base	5010 EXTEAN	do_pty	11782 #define
bufend	11694 #define	del_slot	18926 PAIVATE	do_rdwt	9227 PUBLIIC
buffer	20603 #define	dev_io	27033 PUBLIIC	do_read	11891 PAIVATE
buffer	9135 PAIVATE	deV_ioctl	12763 PAIVATE	do_read	23434 PUBLIC
buflen	11693 #define	dev_mess	27025 PAIVATE	do_reboot	1812B PUBLIIC
bufs_in_use	20154 EXTEAN	dev_mess	25119 PAIVATE	do_rename	25563 PUBLIIC
c_mode	20617 #define	dev_mess	22925 PAIVATE	do_revive	26921 PUBLIIC
c_name	20618 #define	dev_opcl	27071 PUBLIIC	do_sendsig	15157 PAIVATE
call_ctty	27311 PUBLIIC	dma_bytes_lef	9025 #define	do_set	26B96 PUBLIIC
call_task	27245 PUBLIIC	dmap	20914 PUBLIIC	do_set_time	11230 PAIVATE
caps_off	13049 PAIVATE	do_abort	15131 PAIVATE	do_setalarm	11242 PAIVATE
capslock	13046 PAIVATE	do_access	26217 PUBLIIC	do_setsid	27164 PUBLIIC
cause_alarm	1131B PAIVATE	do_alarm	1B056 PUBLIIC	do_setsyn_alr	11269 PAIVATE
cauSe_sig	155B6 PUBLIIC	do_brk	1762B PUBLIIC	do_sigaction	17845 PUBLIIC
cdrom_task	5687 #define	do_cancel	12220 PAIVATE	do_sigpending	17889 PUBLIIC
cfgetispeed	1236 #define	do_chdir	25924 PUBLIIC	do_sigprocma	17898 PUBLIIC
cfgetospeed	1237 #define	do_chmod	26124 PUBLIIC	do_sigreturn	15221 PAIVATE
cfsetispeed	1238 #define	do_chown	26163 PUBLIIC	do_sigreturn	17964 PUBLIIC
cfsetospeed	1239 #define	do_chroot	25963 PUBLIIC	do_Sigsuspend	17949 PUBLIIC
change	25978 PAIVATE	do_clocktick	11140 PRIVATE	do_stat	26014 PUBLIIC
check_pending	18330 PAIVATE	do_clocktick	12198 PRIVATE	do_stime	26475 PUBLIIC
check_sig	18265 PUBLIIC	do_clase	23286 PUBLIIC	do_sync	26730 PUBLIIC
child	20604 #define	do_copy	15316 PRIVATE	do_time	26462 PUBLIIC
cleanup	17061 PRIVATE	do_creat	22937 PUBLIIC	do_times	15106 PRIVATE
clear_zone	24149 PUBLIIC	do_dioctl	9364 PUBLIC	do_tims	26492 PUBLIC
cliCk_tO_hcli	4328 #define	do_dup	26632 PUBLIC	do_trace	15467 PRIVATE
Click_to_hcli	4326 #define	do_endsig	15294 PRIVATE	do_trace	18635 PUBLIIC
click_to_roun	2967 #define	do_escape	14045 PRIVATE	do_umap	15445 PRIVATE
clock_handler	11374 PAIVATE	do_exec	14990 PRIVATE	do_umask	26203 PUBLIIC
clock_mess	27423 PAIVATE	do_exec	26795 PUBLIC	do_umount	25241 PUBLIC
clock_mess	26417 PAIVATE	do_exec	17140 PUBLIC	do_unlink	25504 PUBLIIC
clock_mess	9346 PUBLIC	do_exit	26825 PUBLIC	do_unpause	24560 PUBLIC
clock_stop	11489 PUBLIC	do_fcntl	26670 PUBLIC	do_utime	26422 PUBLIC
Clock_task	1109B PUBLIC	do_fork	14877 PRIVATE	do_vcopy	15364 PRIVATE
clock_time	2742B PUBLIC	do_fork	26757 PUBLIC	do_vrdwt	9255 PUBLIC
co_mode	20605 #define	do_fork	16832 PUBLIC	do_waitpid	16992 PUBLIC
code_base	5009 EXTERN	do_fstat	26035 PUBLIC	do_write	11964 PRIVATE
color	13700 #define	do_gboot	15405 PAIVATE	do_write	24025 PUBLIIC
com_out	10771 PRIVATE	do_get_time	11219 PRIVATE	do_xit	15027 PRIVATE
com_simple	10843 PRIVATE	do_getmap	14957 PAIVATE	dont_reply	16208 EXTEAN
common_open	22975 PAIVATE	do_getset	18515 PUBLIC	dont_reply	19909 EXTERN
common_setala	11291 PRIVATE	do_getsp	15089 PRIVATE	dot1	24719 PUBLIC
compare	5522 #define	do_getuptime	11189 PAIVATE	dot2	24720 PUBLIIC
compare	5515 #define	do_ioctl	12012 PAIVATE	driver_task	9144 PUBLIIC
con_loadfont	14497 PUBLIIC	do_ioctl	27184 PUBLIC	dump_core	18402 PAIVATE
cons_echo	13794 PRIVATE	do_kill	15276 PRIVATE	dup_inode	21865 PUBLIC
cons_orgQ	14456 PRIVATE	do_kill	17983 PUBLIC	eat_path	24727 PUBLIC
cons_stop	14442 PUBLIC	do_ksig	17994 PUBLIC	echo	12531 PRIVATE
cons_table	13696 PRIVATE	do_link	25434 PUBLIIC	eff_grp_id	20606 #define

eff_user_id	20607 #define	hwint_master	6143 #define	m	19917 EXTERN
ega	5046 EXTERN	hwint_slave	6199 #define	m	14812 PRIVATE
enable_iop	7988 PUBLIC	id_byte	10426 #define	m1	19918 EXTERN
env_parse	8865 PUBLIC	id_longword	10429 #define	m1_i1	3149 #define
erki	20606 #define	id_word	10427 #define	m1_i2	3150 #define
err_code	19924 EXTERN	idt	7756 PRIVATE	m1_i3	3151 #define
err_code	16218 EXTERN	in_process	12367 PUBLIC	m1_p1	3152 #define
esc	13047 PRIVATE	in_transfer	12303 PRIVATE	m1_p2	3153 #define
exception	7512 PUBLIC	inform	15627 PUBLIC	m1_p3	3154 #define
exec_len	16403 #define	init_buffer	9205 PRIVATE	m2_i1	3156 #define
exec_name	16402 #define	init_clock	11474 PRIVATE	m2_i2	3157 #define
ext_memsize	5052 EXTERN	init_codeseg	7889 PUBLIC	m2_i3	3158 #define
extpartition	9593 PRIVATE	init_dataseg	7906 PUBLIC	m2_i1	3159 #define
fd	20609 #define	init_params	10307 PRIVATE	m2_i2	3160 #define
fd2	20610 #define	inode	20533 EXTERN	m2_p1	3161 #define
fetch_name	27447 PUBLIC	int_gate	7969 PRIVATE	m3_cal	3166 #define
file_lock	20409 EXTERN	interrupt	6938 PUBLIC	m3_i1	3163 #define
filp	20310 EXTERN	intr_init	7621 PUBLIC	m3_i2	3164 #define
find_dev	27228 PRIVATE	invalidate	21280 PUBLIC	m3_p1	3165 #define
find_filp	22277 PUBLIC	ioflags	20611 #define	m4_l	3168 #define
find_share	17535 PUBLIC	irq_table	5056 EXTERN	m4_l2	3169 #define
findpr#c	18678 PRIVATE	irq_use	5057 EXTERN	m4_l3	3170 #define
flush	27633 PRIVATE	isconsole	11767 #define	m4_l4	3171 #define
flush	19334 PRIVATE	isidlehardwar	5170 #define	m4_l5	3172 #define
flush	13951 PRIVATE	isokprocn	5171 #define	m5_c1	3174 #define
flushall	21295 PUBLIC	isoksrc_dest	5172 #define	m5_c2	3175 #define
forbidden	26242 PUBLIC	isoksusern	5113 #define	m5_i1	3176 #define
force_timeout	11688 #define	isokusern	5174 #define	m5_i2	3177 #define
fp	19907 EXTERN	isrxhardware	5175 #define	m5_l1	3178 #define
fproc	20026 EXTERN	issysentn	5176 #define	m5_l2	3179 #define
free_bit	22003 PUBLIC	istaskp	5177 #define	m5_l3	3180 #define
free_inode	21684 PUBLIC	isuserp	5178 #define	m6_fl	3166 #define
free_mem	18879 PUBLIC	k_atoi	6594 PRIVATE	m6_i1	3182 #define
free_slots	18831 PRIVATE	k_environ	6516 PRIVATE	m6_i2	3183 #define
free_zone	21222 PUBLIC	k_getenv	6606 PUBLIC	m6_i3	3184 #define
front	20152 EXTERN	k_reenter	5015 EXTERN	m6_l1	3185 #define
fs_call	19920 EXTERN	k_to_click	2970 #define	m_device	9722 PRIVATE
fs_init	22625 PRIVATE	k_to_click	2972 #define	m_do_open	9829 PRIVATE
func	16404 #define	kb_ack	13343 PRIVATE	m_dtab	9733 PRIVATE
func_key	13405 PRIVATE	kb_addr	13041 #define	m_geom	9721 PRIVATE
ga_program	14540 PRIVATE	kb_init	13359 PUBLIC	m_geometry	9934 PRIVATE
gdt	7755 PUBLIC	kb_lines	13067 PRIVATE	m_init	9649 PRIVATE
get_block	21027 PUBLIC	kb_read	13165 PRIVATE	m_ioctl	9874 PRIVATE
get_boot_para	22708 PRIVATE	kb_wait	13327 PRIVATE	m_prepare	9759 PRIVATE
get_fd	22216 PUBLIC	kbd_hw_int	13123 PRIVATE	m_schedule	9774 PRIVATE
get_filp	22263 PUBLIC	kbd_loadmap	13392 PUBLIC	main	6721 PUBLIC
get_inode	21534 PUBLIC	last_dir	24754 PUBLIC	main	16627 PUBLIC
get_name	24813 PRIVATE	ldh_init	10133 #define	main	22537 PUBLIC
get_part_tabl	9642 PRIVATE	load_ram	22722 PRIVATE	make break	13222 PRIVATE
get_super	22047 PUBLIC	load_seg	17498 PRIVATE	map_dmp	14660 PUBLIC
get_uptime	11200 PUBLIC	load_super	22832 PRIVATE	map_key	13091 PRIVATE
get_work	22572 PRIVATE	lock_mini_sen	7331 PUBLIC	map_keyO	13084 #define
get_work	16663 PRIVATE	lock_op	22319 PUBLIC	max_hole	18985 PUBLIC
group	20612 #define	loCk_pick_pro	7349 PUBLIC	max_major	20948 PUBLIC
grpId	16405 #define	lock_ready	7361 PUBLIC	mc	11070 PRIVATE
handle_events	12256 PUBLIC	lock_revive	22463 PUBLIC	mem	5024 EXTERN
hclick_to_phy	4330 #define	lock_sched	7388 PUBLIC	mem_init	8820 PUBLIC
held_head	5013 EXTERN	lock_unready	7375 PUBLIC	mem_init	19005 PUBLIC
held_tail	5014 EXTERN	lost_ticks	5031 EXTERN	mem_task	9749 PUBLIC
hole	16827 PRIVATE	low_memsize	5053 EXTERN	merge	18949 PRIVATE
hole_head	16830 PRIVATE	ls_fd	20614 #define	mess	24327 PRIVATE

milli_delay	11502PUBLIC	pc_at	5038 EXTERN	receive	3307 #define
milli_elapsed	11529PUBLIC	pending_ticks	11079 PRIVATE	release	24490PUBLIC
milli_start	11516 PUBLIC	physb_to_hcli	4331 #define	remove_dir	25777 PRIVATE
mini_rec	7119 PRIVATE	pick_proc	7179 PRIVATE	reply	22608 PUBLIC
mini_send	7045 PRIVATE	pid	20630 #define	reply	16676 PUBLIC
mk_mode	20615 #define	pid	16407 #define	reply_i1	16434 #define
mm_call	16215 (XTERN	pipe_check	24385 PUBLIC	reply_i1	20647 #define
mm_exit	16927PUBLIC	pipe_open	23176 PRIVATE	reply_i2	20648 #define
mm_in	16212 EXTERN	pproc_addr	5187 EXTERN	reply_i1	20646 #define
mm_init	16704 PRIVATE	prev_ptr	11081 PRIVATE	reply_p1	16435 #define
mm_out	16213EXTERN	print_buf	19312 PRIVATE	reply_t1	20649 #define
mode	20616 #define	print_buf	27611 PRIVATE	reply_t2	20650 #define
mode_map	22926 PRIVATE	printf	4443 #define	reply_t3	20651 #define
mon_return	5059 EXTERN	printf	15913 #define	reply_t4	20652 #define
mon_sp	5058EXTERN	printf	19581 #define	reply_t5	20653 #define
mounted	22067 PUBLIC	pro	20631 #define	reply_type	16433 #define
mp	16207EXTERN	proc	5186EXTERN	reply_type	20645 #define
mproc	16341EXTERN	proc_addr	5179 #define	reprint	12637PRIVATE
name	20619 #define	proc_name	14690 PRIVATE	request	16414 #define
name1	20620 #define	proc_number	5181 #define	request	20634 #define
name1_length	20623 #define	proc_ptr	5018 EXTERN	res_ptr	16220 EXTERN
name2	20621 #define	proc_vir2phys	5182 #define	result2	16219EXTERN
name2_length	20624 #define	processor	5040EXTERN	ret_mask	16436 #define
name_length	20622 #define	procs_in_use	16209EXTERN	revive	24519PUBLIC
name_to_dev	25299 PRIVATE	prot_init	7767 PUBLIC	reviving	19912EXTERN
namelen	16406 #define	protected_mod	5044 #define	rm_lru	21387 PRIVATE
nbytes	20625 #define	protected_mod	5042 EXTERN	rs_init	11778 #define
net_open	19734 #define	ps_mca	5039EXTERN	rw_block	21243PUBLIC
new_block	24190 PUBLIC	pty_init	11781 #define	rw_chunk	23613PRIVATE
new_icopy	21821 PRIVATE	put_block	21119 PUBLIC	rw_inode	21731PUBLIC
new_mem	17366 PRIVATE	put_inode	21578 PUBLIC	rw_scattered	21313PUBLIC
new_node	23111PRIVATE	put_irq_handl	7673PUBLIC	sc_a0	2126 #define
next_alarm	11069PRIVATE	putch_msg	19313 PRIVATE	sc_a1	2127 #define
next_pid	16825PRIVATE	putch_msg	27612 PRIVATE	sc_a2	2128 #define
no_dev	27337 PUBLIC	putk	27619 PUBLIC	sc_a3	2129 #define
no_name	9302 PUBLIC	putk	19320PUBLIC	sc_a4	2130 #define
no_sys	19161 PUBLIC	putk	14408 PUBLIC	sc_a5	2131 #define
no_sys	27489 PUBLIC	rahead	23805PUBLIC	sc_bx	2104 #define
nop_cleanup	9338 PUBLIC	rawecho	12593 PRIVATE	sc_cs	2111 #define
nop_finish	9329 PUBLIC	rd_indir	23753 PUBLIC	sc_cx	2106 #define
nr_cons	13695PRIVATE	rd_only	20632 #define	sc_d1	2119 #define
nr_locks	19911 EXTERN	rdahed_inode	19914EXTERN	sc_d2	2120 #define
num_off	13051PRIVATE	rdahedpos	19913EXTERN	sc_d3	2121 #define
numap	15697 PUBLIC	rdwt_err	19925EXTERN	sc_d4	2122 #define
numlock	13050 PRIVATE	rdy_head	5192 EXTERN	sc_d5	2123 #define
numpad_map	13056PRIVATE	rdy_tail	5193 EXTERN	sc_d6	2124 #define
offset	20626 #define	read_ahead	23786 PUBLIC	sc_d7	2125 #define
old_icopy	21774 PRIVATE	read_header	17272 PRIVATE	sc_di	2100 #define
out_char	13809 PRIVATE	read_map	23689 PUBLIC	sc_ds	2099 #define
out_process	12677 PUBLIC	read_only	26304PUBLIC	sc_dx	2105 #define
owner	20627 #define	read_super	22088PUBLIC	sc_es	2098 #define
p_dmp	14613 PUBLIC	read_write	23443 PUBLIC	sc_fp	2102 #define
panic	6829 PUBLIC	ready	7210 PRIVATE	sc_fp	2132 #define
panic	19172PUBLIC	real_grp_id	20613 #define	sc_fs	2096 #define
panic	27500 PUBLIC	real_user_id	20633 #define	sc_gs	2095 #define
panicking	27422PRIVATE	realtime	11067 PRIVATE	sc_pc	2110 #define
parent	20628 #define	rear	20153EXTERN	sc_pc	2134 #define
parse_escape	13986PRIVATE	reboot_code	16429 #define	sc_psw	2112 #define
partition	9521 PUBLIC	reboot_code	5060 EXTERN	sc_psw	2135 #define
patch_ptr	17465PRIVATE	reboot_flag	16428 #define	sc_retadr	2108 #define
pathname	20629 #define	reboot_size	16430 #define	sc_retagr	2107 #define

sc_rstreg	2118 #define	syn_al_alive	11075 PRIVATE	w_name	10511 PRIVATE
sc_si	2101 #define	syn_alm_task	11333 PUBLIC	w_need reset	10813 PRIVATE
sc_sp	2113 #define	syn_table	11076 PRIVATE	w_nextblock	10241 PRIVATE
sc_sp	2133 #define	sys_call	7008 PUBLIC	w_opcode	10242 PRIVATE
so_ss	2114 #define	sys_task	14837 PUBLIC	w_prepare	10388 PRIVATE
sc_st	2103 #define	t_stack	5734 PUBLIC	w_reset	10889 PRIVATE
scan_keyboard	13432 PRIVATE	taddr	16415 #define	w_sche{!ule	10567 PRIVATE
sched	7311 PRIVATE	task	27026 PRIVATE	w_specify	10531 PRIVATE
schsd_ticks	11080 PRIVATE	tasktab	5699 PUBLIC	w_status	10244 PRIVATE
scr_init	14343 PUBLIC	tell_fs	19192 PUBLIC	w_timeout	10858 PRIVATE
scr_lines	13673 #define	termios_defau	11803 PRIVATE	w_tp	10239 PRIVATE
scr_size	13674 #define	toggle_scroll	14429 PUBLIC	w_waitfor	10955 PRIVATE
scr_width	13672 #define	tot_mem_size	5025 EXTERN	waitfor	10268 #define
scroll_screen	13896 PRIVATE	tp	20637 #define	watch_dog	11072 PRIVATE
scsi_task	5686 #define	truncate	25717 PUBLIC	watchdog_proc	11071 PRIVATE
sdesc	7922 PRIVATE	tss	7757 PUBLIC	whence	20642 #define
search_dir	24936 PUBLIC	tty_active	11774 #define	who	16214 EXTERN
seconds	16408 #define	tty_addr	11757 #define	who	19919 EXTERN
seg2phys	7947 PUBLIC	tty_devnop	12992 PUBLIC	winchester ta	10040 PUBLIC
selst_console	14482 PUBLIC	tty_icancel	12891 PRIVATE	wini	10230 PRIVATE
sand	3308 #define	tty_init	12905 PRIVATE	winsize_defau	11811 PRIVATE
ssndrec	3306 #define	tty_open	27095 PUBLIC	wipe_inode	21664 PUBLIC
set_6645	14280 PRIVATE	tty_reply	12845 PUBLIC	wr_indir	24127 PRIVATE
set_slarm	18067 PUBLIC	tty_table	11670 EXTERN	wrap	13668 PRIVATE
set_leds	13303 PRIVATE	tty_task	11817 PUBLIC	wreboot	13450 PUBLIC
set_vec	7616 #define	tty_timelist	11690 EXTERN	write_map	24036 PRIVATE
setattr	12789 PRIVATE	tty_timeout	5032 EXTERN	wtrans	10237 PRIVATE
settimer	12958 PRIVATE	tty_wakeup	12929 PUBLIC	zero_block	24243 PUBLIC
shift	13054 PRIVATE	umap	15658 PUBLIC		
sig	20635 #define	umess	23425 PRIVATE		
sig	16409 #define	unhold	7400 PUBLIC		
sig_context	16424 #define	unlink_file	25818 PRIVATE		
sig_flags	16423 #define	unpause	18359 PRIVATE		
sig_how	16422 #define	unready	7258 PRIVATE		
sig_msg	16426 #define	update_times	21704 PUBLIC		
sig_nr	16417 #define	user_path	19921 EXTERN		
sig_nsa	16418 #define	usizeof	19515 #define		
sig_osa	16419 #define	use_id	16413 #define		
sig_proc	18168 PUBLIC	utime_actime	20638 #define		
sig_procs	5021 EXTERN	utime_file	20640 #define		
sig_ret	16420 #define	utime_length	20641 #define		
sig_set	16421 #define	utime_modtime	20639 #define		
sigchar	12866 PUBLIC	vga	5049 EXTERN		
size_ok	17736 PUBLIC	vid_base	13670 PRIVATE		
slock	13052 PRIVATE	vid_mas k	13663 PUBLIC		
slock_off	13053 PRIVATE	vid_port	13667 PRIVATE		
slot1	20636 #define	vid_seg	13661 PUBLIC		
softscroll	13669 PRIVATE	vid_size	13662 PUBLIC		
sort	9676 PRIVATE	vir2phys	4441 #define		
spurious_irq	7657 PRIVATE	w_command	10243 PRIVATE		
stack_bytes	16410 #define	w_count	10240 PRIVATE		
stack_ptr	16411 #define	w_do_close	10828 PRIVATE		
stat_inode	26051 PRIVATE	w_do_open	10355 PRIVATE		
status	16412 #define	w_drive	10245 PRIVATE		
stop_beep	14329 PRIVATE	w_dtab	10274 PRIVATE		
stop_proc	18691 PUBLIC	w_dv	10246 PRIVATE		
super_block	20745 EXTERN	w_finish	10649 PRIVATE		
super_user	19908 EXTERN	w_geometry	10990 PRIVATE		
susp_count	19910 EXTERN	w_handler	10976 PRIVATE		
suspend	24463 PUBLIC	w_identify	10415 PRIVATE		
switching	6921 PRIVATE	w_intr_wait	10925 PRIVATE		

INDICE

925

INDICE

- A**
- abrazo mortal, 69, 166-179
 - algoritmo de reemplazo
 - del avestruz, 170-172
 - del banquero, 175-179
 - condiciones para, 168
 - detección y recuperación de, 172
 - en MINIX, 186
 - estado de, 175
 - estado seguro, 175-176
 - evitación de, 175-179
 - modelado de, 169-171
 - prevención de, 173-175
 - trayectoria de recursos, 176-177
 - acceso
 - a archivos, 407-408
 - acción atómica, 66
 - ACL (*véase* lista de control de acceso)
 - acuse de recibo, 73
 - adaptador
 - administración
 - de espacio en disco, 422-424
 - algoritmo
 - de ajuste rápido, 318
 - de mejor ajuste, 318
 - de peor ajuste, 318
 - de primer ajuste, 317
 - de siguiente ajuste, 317
 - de tasa monotónica, 91
 - en MINIX, 356-396
 - lista enlazada de, 317-318
 - mapa de bits, 316
 - algoritmo (*véase también* de páginas)
 - de procesos, MINIX, 95-97
 - administrador de ventanas, 240
 - Aiken, Howard, 6
 - alarma síncrona, MINIX, 228-229, 233
 - algoritmo
 - de ajuste rápido, 318
 - de calendarización, 82-84
 - de frecuencia de fallas de página, 341
 - de mejor ajuste, 318
 - de menos recientemente utilizado, 334-338, 429
 - de no utilizado frecuentemente, 336
 - de paginación de segunda oportunidad, 333-334
 - directo a memoria, 157-159 *de peor ajuste*, 318
 - de plazo más próximo primero, 91
 - de primer ajuste, 317
 - de reemplazo
 - de páginas, 331-338
 - de E/S, 208 *maduración*, 336-338
 - dispositivo, 155 *menos recientemente utilizada*, 334-338
 - no utilizada frecuentemente, 336
 - no utilizada recientemente, 332-333
 - de memoria, 309-397 *por frecuencia de fallas de página*, 341
 - por reloj, 334-335
 - primero que llega, primero que sale, 333
 - segunda oportunidad, 333-334
 - wsclock, 339
 - de siguiente ajuste, 317
 - de tasa monotónica, 91

- algoritmos
- dmp.c*, 287, 295
- driver.c*, 181, 190, 192-194, 198,212,215
 - alternación estricta, 60-61
 - amabilidad con el usuario, 407
 - anillo de protección, MULTICS, 356
 - ANSI
- exception.c*, 142, 375
- apuntador nulo, 231
- archivo, 17-20,402-410
 - aleatorio, 407
 - secuencial,407 *fproc.h*,471
- en MINIX, 468
 - por bloques, 20, 405
 - por caracteres, 20,405
- archivos
 - de cabecera
 - MINIX, 107-112, 379-382
 - POSIX, 102-106
 - a.out.h*, 105, 122
 - alloc.c*, 366, 394
 - an.~i.h*, 103-105
 - assert.h*, 118, 147
 - at_wini.c*, 137,212,220,308
 - boot.h*, 112,302
 - break.c*, 386
 - brksize.s*, 372
 - buf.h*, 471-472
 - cache.c*,474
 - callnr.h*,111
 - clock.c*, 181,227,228,230,234
 - com.h*, 111
 - config.h*, 100, 103, 107, 114, 116, 119, 122,212, 266,287,463,470,472,474-478,93,506
 - console.c*, 282, 283, 287, 288, 289, 295
 - const.h*, 100, 108, 109, 113, 114, 116,379,470, 472, 506, 108
 - crt.o.s*, 372
 - del avestruz, 170, 172 *dev.h*, 472
 - del banquero, 175-179 *device.c*, 501, 502
 - del elevador, 204 *dir:h*, 106,470
 - diskparm.h*, 112
 - globales de asignación de páginas, 339-341
 - locales de asignación de páginas, 339-341
 - driver: h*, 182, 190
 - drvlib.c*, 193, 194,213
 - drvlib.h*, 193, 194
 - errno.h*, 104
 - secuencia de escape de terminal, 249
 - Standard C, 103 *exec.c*,385-386
 - fcntl.h*, 104, 265, 379
 - file.h*,472
 - de acceso *filedes.c*,481
 - forkexit.c*, 382
 - ejecutable, 405-406 *fs.h*, 102,470
 - especial, 468 *getset.c*, 393
 - glo.h*, 108, 113, 116, 117, 119,228,265,266, 281,379,380,381,470,471
 - i8259.c*, 142, 143
 - normal, 405 *inode.h*, 472
 - ioctl.h*, 106
 - kernel.h*, 102, 112, 113, 116,379,470
 - keyboard.c*, 262, 282, 283, 287, 288
 - keymap.h*, 112,263,264,284
 - fuelle de MINIX, *keymap.~rc*, 263
 - klib.s*, 145
 - klib386.s*, 145,291
 - limits.h*, 104,487
 - link.c*, 496, 497
 - lock.c*,482
 - lock.h*,472
 - main.c*, 124, 125, 128,382,482-485
 - memory.c*, 198,200
 - minix*,111
 - misc.c*, 145, 147,500
 - mm.h*, 102,379,470
 - mount.c*, 495, 496
 - mproc.h*, 363
 - mproc.h*, 363, 380, 381
 - mpx.s*, 123, 145
 - mpx386.s*, 119, 123, 124, 127, 131, 133, 136, 137, 142,152,252
 - mpx88.s*, 123
 - open.c*, 485, 488

- param.h*, 381, 473
 - partition.h*, 112, 194
 - path.c*, 486, 494
 - pipe.c*, 492, 493
 - proc.c*, 132, 137, 138, 141,303
 - proc.h*, 116, 118, 119, 138, 140,296
 - protect.c*, 142, 143, 144, 145, 199,498
 - protect.h*, 117, 143
 - proto.h*, 113, 115,379,380,470,471
 - ptrace.h*, 106
 - Pty.c*,268
 - putk.c*, 395, 503
 - read.c*, 477, 488, 490
 - sconst.h*,118
 - screendump*,291
 - sigcontext.h*, 106
 - signal.c*, 387, 390
 - signal.h*, 104,381
 - stadil:L*,498 bibliografía
 - start.c*, 124, 125, 127, 144
 - stat.h*, 106,470
 - stdlib.h*, 104
 - string.h*, 104
 - Supel:c*,480 bit
 - Supel:h*,473
 - syslib.h*, 111
 - system.c*, 296, 303
 - table.c*, 108, 109, 116, 119, 126,266,361,379,
 - sucio*, 327
 - termios.h*, 105, 106,250,267,502
 - time.c*,499
 - traL'e.c*,394
 - ty.c*, 181,253,267,279,282,287,288,289
 - ty.h*, 119, 182,265,266,267
 - type.h*, 109, 113, 114, 115, 119, 192,379,380, fal tan te, 427
 - 470,471
 - types.h*, 105
 - unistd.h*, 104, 379
 - uS-std.src*, 263, 283
 - utility.c*, 395, 503
 - wait.h*, 106
 - wini.c*,212 351,355
 - write.c*, 488, 491,492
 - quitectura búsqueda
 - de computadoras, 3
 - etiquetada, 450
 - arranque de *MINIX*, 96,120-122
 - asignación
 - contigua de archivos, 415
 - de archivos
 - indizada,417-418
 - nodo-i*,418-419
 - por lista enlazada, 416-417
 - de páginas local vs. global, 339-343
 - atrapamiento de señales, *MINIX*, 372
 - atributo de archivo, 408-409
 - atributos de archivos, 408-409
 - autenticidad, verificación de, 73
 - usuario, 442-446
 - autoarranque,96
- B**
- Babbage, Charles, 5
 - alfabética,512-517
 - lecturas sugeridas, 507-512
 - biblioteca cOmpartida, 346
 - Bias, 209-210
 - de espera para despertar, 65
 - de modificado, 327
 - de presente/ausente, 322
 - de referencia, 327380,381,471,472,473,501
 - bloque
 - de arranque, *MINIX*, 121,454
 - de indirección, 418-419
 - doble, 419
 - sencilla, 418
 - bloqueo mortal (*véase* abrazo mortal)
 - bloques
 - libres en *disco*, 423-424
 - malos, manejo de, 206-207
 - bomba lógica, 438
 - buffer de consulta para traducción, 327-329, 350,
 - administrado en software, 329-330
 - más Corta primero, 204
 - traslapada,201
 - buzón, 74

C

- caballo de Troya, 438
- caché, 430
 - de bloques en MINIX, 429, 461-463, 474-478
 - de buffers, 429
 - de escritura a través, 429
 - de una pista a la vez, 207-208
- calendarización
 - algoritmo de tasa monótonica, 91
 - apropiativa, 83
 - de dos niveles, 92
 - de ejecutarse hasta terminar, 84
 - de menor holgura, 91
 - de múltiples colas, 86-87
 - de procesos, 82-92
 - de trabajo más corto primero, 87-99
 - del brazo del disco, 203-205
 - determinada por política, 93
 - en MINIX, 98, 140-142
 - en tiempo real, 90-91
 - garantizada, 89
 - no apropiativa, 84
 - plazo próximo primero, 91
 - política vs. mecanismo, 93
 - por lotería, 89-90
 - por prioridad, 85-86
 - por turno circular, 84-85
 - XDS 940, 87
- calendarizador, 82
- canal
 - de E/S, 155
 - encubierto, 451-453
- candados
- MINIX, 288-295
- capacidad, 450-451
- carácter de relleno, 244
- cbreak, modo, 31, 247
- CDC 6600, 313
- cena de filósofos, 75-77
- cita, 75
- clic de memoria, 363
- cliente X, 240
- cliente-servidor, sistema operativo, 42-44
- código de corrección de errores, 155
- compactación de memoria, 314
- computadora
 - de primera generación, 6
 - de segunda generación, 6-8
 - de tercera generación, 8-11
- comunicación entre procesos, 16, 57-75, 97
- cena de filósofos, 75-77
- en MINIX, 137-140
- lectores y escritores, 77-80
- monitores, 68-72
- paso de mensajes, 72-75
- peluquero dormido, 80-82
- condición
 - de competencia, 58
 - de exclusión mutua, 168
 - de retener y esperar, 168
- conducto, 20
 - en MINIX, 467-469
- confiabilidad del sistema de archivos, 424-429
- confidencialidad, 435
- conmutación
 - de contexto, 84
 - de procesos, 84
- consola virtual, MINIX, 282
- contraseña, 442-444
 - de una vez, 443
 - salar, 443
- control de trabajos, 26, 272
- controlador
 - de disco flexible, MINIX, 220-222
 - de dispositivo en hardware, 155
 - de dispositivo en software, 94, 155, 161-162
 - en MINIX, 181-185
 - aseSores para archivos en MINIX, 467
 - de exhibición,
 - de archivos, MINIX, 467, 482
 - de reloj, MINIX, 227-235
 - de dos fases, 179
 - de teclado, MINIX, 282-288
- de terminal,
 - código de apoyo, 273-282
 - en MINIX, 249-296
- de video, 236
- en hardware de dispositivos de E/S, 155-157
- CP/M, 419-420
- CTSS, 86-87
- cuadro de páginas, 320
- cuanto, 84

D

daemon 95 165

Dekker, algoritmo de, 61

derecho
a la seguridad, 447
genérico, 451
derechos genéricos, 451
descriptor
de archivo, 19,465-467,481-482

despachado (*véase* calendarización)

despertar, 64

desplazamiento de pantalla en una terminal, 259

dirección

direccionamiento de bloques lineal, 213

directorio, 17,405,410-415

en MINIX, 463, 465

en UNIX, 420-421

estructura de, 419-421

directorios jerárquicos, 411-412
djvco, 200-222

por bloques, 154
MINIX, 187-195

por caracteres, 154

SCSI, 155
secundario, 34, 163

DMA (*véase* acceso directo a memoria)

de im~re~ora, 57 domi~io de protección, 446-448

dormir y despertar, 63-64

E

E/S, 153-304
con mapa en la memoria, 156
de segmento, Pentium, 144, 353 de disco, 200-222
en RAM, 195-200
de reloj, 222-235
de terminal, 235-296
por hardware, 259 MINIX, 235-304
por software, 259 en MINIX, 179-304

ECC (*véase* código de corrección de errores)

lineal, Pentium, 353 Eckert, J. Presper, 6

virtual, 319 eco, 242-244

EIDE (*véase* Extended Integrated Drive Electronics)

entrada
actual, 412 de terminal, MINIX, 251-256
de páginas, Pentium, 354 estándar, 21
de spool, 57,165 salida (*véase* E/S)

de trabajo, 18,412 equipo
en CP/M, 419-420 de penetración, 439
de respuesta a emergencias de computadora, 439
en MS-DOS, 420 tigre, 439

error de E/S, 205-207

errores de disco, manejo de, 205-207

implementación de, 419-421 escape, carácter de, 245

jerárquico, 411-412 escotillón, 440

raíz, 18 espacio de direcciones, 15
virtual, 319-320

espejo, 426

con electrónica de unidad integrada, 201 espera
en RAM, 195-200 activa, 59-63
en MINI X, 197-200 circular, condición de, 168

hardware de, 200-202 estación de trabajo sin disco, 122

software de, 202-208 estado
winchester, 211-212 de procesos, 50-52

dispositivo zombi, 367-368, 384

de E/S, 154-155 estructura
de archivos, 404-405
de datos de terminal, MINIX, 265-267
de sistema operativo, 37-44

cliente-servidor, 42-44
 de capas, 39-40
 de máquina virtual, 40-42
 evento
 aperiódico, 90
 periódico, 90
 evitación de bloqueos mortales, 175-179
 excepción, Pentium, 136
 exclusión mutua, 59-63
 exokernel, 42
 Extended Integrated Drive Electronics, 210
 extensión de archivo, 403
 PC, 14, 107, 122, 126, 132, 156,191,194,201,

System/360, 8-9

F

falla de página, 322
 FIFO (*véase* reemplazo de páginas de primero que llega primero que sale)
 FORTRAN,6-8

externa, 347-348
 interna, 342
 fuente (tipo de letra)

inhabilitación de interrupciones, 59-60
 inicialización del sistema de archivos de MINIX, 483-485

G

gatos, método de identificación de, 444
 GDT (*véase* tabla de descriptores globales)
 GE-645, 11
 gusano, 438-439
 IPC (*véase* comunicación entre procesos)
 IRQ (*véase* solicitud de interrupción)

H

hardware de terminal, 235-240
 hertz, 83
 hilo limpiador, 433

C, 55
 P, 55

historia
 de los sistemas operativos 5-15
 de MINIX, 13-15
 holgura, 91

I

IBM

1401,6,8
 7094, 6, 8, 9

202,208,313,357,360

IDE, disco (*véase* disco con electrónica de unidad integrada)

identificación

física, 444-445
 entre gatos, 444
 por huellas digitales, 445
 fragmentación por longitud de los dedos, 444-445

imagen de núcleo, 16

implementación de procesos, 52-53

inanición, 76

cargable,264 independencia del dispositivo, 159-160
 de MINIX, 264 indicación, 21

Intel 8088, 12,41, 115, 126,208,301,313, 357

intercalación, 158

intercambio, 313-318

interfaz de memoria virtual, 343

intérprete de comandos, 16

introdutor de secuencia de control, 260

J

jefe de sesión, 272

hilos, 53-56 jerarquía
 de memoria, 309
 de procesos, 49-50

- K**
- kernel,94
- L**
- LBA (véase direcciónamiento de bloques lineal)
- LDT (véase tabla de descriptores locales)
- lenguaje
- ley de Murphy, 57
- Linux, 15
- lista
- de capacidades, 450
- de control de acceso, 448-450
- llamada
- de archivos, 28-33
- de directorios, 33-35
- de procesos, 22-26
- de tiempo, 36
- (véase también llamadas al sistema de MINI X)
- llamadas al sistema de MINIX,
- ACCESS, 32, 36, 453,499
- ALARM, 224, 227, 228, 28, 361, 389, 56
- SIGPROCmask, 27, 373, 375, 388
386-387,391,95
- CHOWN, 35, 436, 498
- DUP, 29-31, 500
- EXEC, 24-27, 31, 95, 97,143,186,298,356-361,
365,368,369,371-372,380,385-387,
399,400,447,471,498,499,501
- EXIT, 25, 97,298, 361, 367-368, 382-385, 501
- FCNTL, 33,482, 500-501
- FORK, 22-27, 30,49,55,95,97, 117, 172, 296,
297,356,357,358,359,361,367-368,
380, 382-385, 396, 466, 501
- FSTAT,29, 106,471,479,498
- GETGID, 35, 361, 378, 393
- GETPGRP, 26, 379
- GETPID, 26, 361, 378
- GETUID, 35, 361, 378, 393
- IOCTL, 31-33, 106,200,241,246,249,250,263,
266, 271-273, 280, 295, 502
- de máquina, 2
- KILL, 27, 300, 361, 374, 389
- ensamblador, 8,9,22,53-54
- LINK, 33-34,496-497
- LSEEK, 29,465,466,473,485
- MKDIR, 33, 485-487
- MKNOD, 28, 29, 35, 36,436,485-487
- MOUNT, 19,34,458,464,493-496,494,495
- OPEN, 28, 29,164,189,192,265,463,481,
485-486, 493
- al sistema, 15,21-36
- PAUSE, 28, 361, 390, 393
- al supervisor, 27
- PIPE, 30, 492
- administración
- Ptrace, 26, 106,302,361,379,394
- READ, 15, 16,22,29,56,75,95,98, 160, 183,
247,266,270,274,300, 378, 393, 407,
446,454, 465,469,488-492,499
- RENAME, 33, 496, 504
- del kernel, 37
- RMDIR, 33, 496
- protección, 35-36
- SBRK, 26, 299, 361, 371
- señalización, 26-28
- SETGID, 35, 379
- SETSID, 26, 379, 394, 502
- SETUID, 35, 379
- SIGACTION, 27,361,373,375,387,388,389
- SIGPENDING, 27, 361, 388
- BRK, 25, 26, 297, 299,360, 361, 363, 371-372,
- SIGRETURN, 27, 300, 361, 373, 376-388, 389,
CHDIR, 35, 95, 465, 498 391
- CHMOD, 35,498
- SIGSEGV, 375
- SIGSUSPEND, 27, 361, 389, 391, 393, 389
- CHROOT, 35, 465, 498
- SLEEP, 64, 66, 70
- CLaSE, 29, 188, 189,485
- STAT, 29, 106,471,479,498
- CREAT, 28, 29, 36,481,485-486,492,499
- STIME, 36, 499
- SYNC, 35, 430, 431, 463, 476, 500-501
- DUP2, 500
- TIME, 36, 499
- TIMES, 36, 299, 499
- UMASK, 36, 499
- UMOUNT, 34,494,496
- UNLINK, 34,493,496

WAIT, 24, 25, 106, 361, 366, 367-368, 382-385, 393, 70, 72

97 WRITE, 29, 34, 165, 183, 257, 266, 270, 271, 289, 290, 393, 431, 446, 465, 491-492

localidad de referencia, 338

Lord Byron, 5

Lovelace, Ada, 5

LRU (*véase* algoritmo de menos recientemente utilizado) autoarranque, 120-122

M

delkernel, 145-147

bloque de arranque, 121, 454

maduración, 336-338

manejador de interrupciones, 128-137, 161, 180-181

de MINIX, 128-137, 180-181

manejo de mensajes, MINIX, 361-362

mapa

de teclas cargable, MINIX, 261-264

máquina

virtual, 1-2, 4

mecanismo

de calendarización, 93

de protección, 434, 446-453

vs. política, 44, 93, 357

asociativa, 328

compartida distribuida, 343

virtual, 319-356

algoritmos de reemplazo de páginas, 331-338

cuestiones de diseño, 338-343

en MULTICS, 348-351

paginación, 319-331

por demanda, 338

segmentación, 343-356

UTIME, 36, 499

administración

de bloques del sistema de archivos, 474-478

de memoria, 356-396

WAITPID, 24, 25, 106, 361, 384, 385 de procesos, 95-

administrador de memoria, 95

alarma sincrónica, 228-229

archivo especial, 20, 405, 468

archivos de cabecera, 107-112, 379-382, 470-473

arranque, 96

atrapamiento de señales, 373

biblioteca

de controladores, 193-195

bloques mortales, 186

cabeceras del sistema de archivos, 470-473

caché de bloques, 461-463

de reloj, MINIX, 230

candados de archivos, 467, 482

código

de apoyo del controlador de terminal, 273-282

de bits, 316, 458-460

fuerza, 99-101

de teclas, 242

comunicación entre procesos, 137-140

conducto, 467-469, 492-493

en MINIX, 261-264

controlador

de disco

extendida, 3-4

duro, 208-222

RISC, 2

flexible, 220-222

de dispositivo, 94, 181-185

matriz de acceso, 448

por bloques, 190-193

Mauchley, William, 6

de exhibición, 288-295

de terminal independiente del dispositivo, 264-282, 267-273

del teclado, 282-288

definición EXTERN, 108-109, 265, 380, 471

memoria descriptor de archivo, 465-467, 481-482

DEV OPEN, solicitud, 210-211

DEV READ, solicitud, 211

directorios, 463-465

disco en RAM, 197-200

dispositivo por bloques, 187-195

E/S, 179-304

de terminal, 235-304

entradas de terminal, 251-256

estructura

microprograma, 2

de datos de terminal, 265-267

MINIX interna, 93-95

- estructuras procesamiento
 - del administrador de memoria, 363-367
 - del sistema de archivos, 473-482
 - fuelle cargable, 264
- 256-261
 - del controlador de disco duro, 208-211
 - implementación
 - 458,480-481
 - del controlador
 - de disco duro, 211-222
 - de reloj, 230-235
 - inicialización, 122, 128
 - interfaz con dispositivo de E/S, 501-502
 - IPC,97
 - kemel,94
 - lista de señales, 374
 - manejador de interrupciones de reloj, 230, 233-234
 - manejo
 - mapa
 - cargable,261-264
 - mapas de bits del sistema de archivos, 458-460
 - mensajes
 - monitor de arranque, 121,209
 - nodo-i, 460-461, 478-480
 - número mágico, 454
 - operaciones con archivos, 485-493
 - organización
 - parámetros de arranque, 209-210
 - planificación de procesos, 98,140-142
 - terminos, 246
 - posición en un archivo, 465
 - de datos, 112-120
 - de nombres de trayectoria, 493-494
 - de secuencias de escape, 291-294
 - programa principal
 - del administrador de memoria, 382
 - generalidades del sistema de archivos, 482-483
 - de administración de memoria, 356-379
 - salidas de terminal,
 - de E/S, 179-186
 - servicios de reloj, 230-231
 - de procesos, 93-98
 - seudoterminal, 250
 - sistema de archivos, 95, 453-503
 - del controlador de terminal, 249-251
 - montado, 494-496
 - del controlador del reloj, 227-230
 - software de E/S
 - del sistema de archivos, 453-469
 - a nivel de usuario, 185-186
 - independiente del dispositivo, 185
 - de administración de memoria, 379-396
 - superbloque,456-
 - de procesos, 98-147
 - tarea, 94
 - del sistema, 296-304
 - temporización en milisegundos, 230
 - texto compartido, 359
 - del sistema de archivos, 470-503
 - vaciado de núcleo, 373
 - vaciados para depuración, 295-296
 - del sistema de archivos, 483-485
 - variables inicializadas, 116
 - vinculación de archivos, 496-497
 - zombis, 367-368, 384
 - MMU (*véase* unidad de administración de memoria)
 - modelo
 - de conjunto de trabajo, 338-339
 - de proceso, 48-52
 - de interrupciones, 128-137, 180-181
 - modo
 - de mensajes, 361-362
 - canónico, 31-32, 241
 - de señales, 372-378, 387-393
 - cocido, 31, 241
 - crudo, 31, 241
 - de teclas, 261-264
 - de kemel, 2
 - de onda cuadrada, 223
 - de un disparo, 223
 - de usuario, 2
 - al administrador de memoria, 361-362
 - no canónico, 31-32, 241
 - al sistema de archivos, 454-455
 - protegido, 115
 - supervisor, 2
 - monitor, 68-72
 - de arranque, MINIX, 121,209
 - de máquina virtual, 40-41
 - de referencias, 446
 - de la memoria, 101,358-361
 - monoprogramación, 310-311
 - del sistema de archivos layout, 454-458
 - Morris, Robert, 438-439
 - MS-DOS, 12,41, 156-157, 195, 196,285,310,402
 - 403,412,418,420,430

- multiprocesador, 47
 75
 PDP-7, 11
 PDP-11, 11
N
 NEC PD 765, chip, 3
 nivel
 de privilegio, Pentium, 118, 130
 nodo-i, 418-419 nivel
 MINIX, 460-461, 478-480
 nombre
 de trayectoria, 18, 412-414
 absoluta, 412
 relativa, 412
 nombres
 352-353
 61-62
 en MINIX, 454
 con múltiples colas, 86-87
 no apropiativa, 84
O
 operación con archivos, 409-410
 prepaginación, 339
 prevención de bloqueos mortales, 173-175
 primitiva
P
 página, 320
 paginación problema
 por demanda, 338 de confinación, 452
 MULTICS, 11, 348-351 partición, 34
 protección de anillo, 356 de disco 96
 fija, 311~312
 multiprogramación, 9-10, 48 paso de mensajes, 72-
 con particiones fijas, 311-313 PDP-1, 11
 peluquero dormido, 80-82
 Pentium, 115
 descriptor de segmentos, 144, 353
 direcciones lineales, 353
 directorio de páginas, 354
 de protección de E/S, Pentium, 199 excepción en, 136
 de privilegio, 118, 130
 de protección de E/S, 199
 de ruta absoluta, 412 puerta de llamada, 356
 segmento de estado de tareas, 129
 tabla
 de descriptores
 de interrupciones, 124, 143
 de archivos, 402-403 globales, 124, 143,
 uniformes, 160 locales, 143, 352-353
 número tablas de descriptores, 118
 de dispositivo principal, 163 Peterson, algoritmo de,
 mágico, 406 pixel, 236
 planificación
 política
 de calendarización, 93
 vs. mecanismo, 44, 93, 357
 posición en un archivo, MINIX, 465
 operaciones POSIX, 11
 con directorios, 414-415 archivos de cabecera, 102-106
 de IOCTL, MINIX, 271 preámbulo, 155
 oscilador de cristal, 223 prefijo de clave extendida, 286
 SIGNAL, 70 SLEEP, 63-64, 70
 WAIT, 70
 de código, 242 WAKEUP, 63-64, 70
 introducción, 319-322 de buffer acotado, 64

- de inversión de prioridad, 64
- de lectores y escritores, 77-80
- de productor-consumidor, 64-75
- IPC clásico, 75-82

- lectores y escritores, 77-80

- Jroceso, 15-17
- en MINIX, 93-147
- hijo, 16
- introducción a, 47-56
- ligero, 54
- padre, 16
- secuencial (*véase* proceso)
- servidor, 43
- protección de memoria, 312-313
- prueba de capacidades, macro de, 112
- puerta de llamada, Pentium, 356
- punto de corte, 302
- seudoparalelismo, 47
- seudoterminal, 250

- R**

- identificación física, 444-445
- RAM de video, 235-236
- recurso, 167-168
- apropiable, 167

- autenticidad de usuarios, 442-446
- reemplazo de páginas

- semáforo, 66-68

- región crítica, 58-59
- registro

- procesos, 68

- reloj

- espacio en disco, 422-424
- MINIX, 222-235
- software de, 224-227
- respaldo de archivos, 425-426
- reto-respuesta, verificación de autenticidad por, 444
- reubicación de memoria, 312-313
- rótulo local, en ensamblador, 133
- R, ~-712 terminal, 238-239
- mapas de bits de, MINIX, 458-460

- S**

- salar, 443
- cena de filósofos, 75-77
- salida
- de terminal, MINIX, 256-261
- peluquero dormido, 80-82
- estándar, 21
- sección crítica, 58-59
- secuencia de escape, 291-294
- en MINIX, 249
- segmentación, 343-356
- en MULTICS, 348-351
- implementación de, 347-348
- segmento, 345
- de datos, 25-26
- de estado de tareas, Pentium, 129, 143
- de pila, 25-26
- de texto, 25-26
- seguridad, 434-446
- amenazas, 434-436
- ataques a la, 439-441
- capacidad de, 450-451
- defectos de, 436-439
- gusano, 438-439

- lista de control de acceso, 448-450
- mecanismos de protección, 446-453
- principios de diseño para, 441-442
- no apropiable, 167
- verificación de

- virus, 440-441
- de primero que llega primero que sale, 333

- no utilizadas recientemente, 332-333
- binario, 68
- óptimo, 331-332
- señal, 17
- en MINIX, 374
- servidor X, 240
- base, 313 shell, 16, 20-21
- de arranque maestro, 96
- sincronización de

- de dispositivo, 2
- sistema
- de límite, 313
- calendarizable, 91
- de archivos, 401-503
- hardware de, 223-224
- administración de

- confiabilidad de, 424-429
- consistencia de, 426-429
- de MINIX, 453-503
- directorios de, 419-421
- estructurado por bitácora, 432-434
- implementación de, 415-434

- montado, 160
 - en MINIX, 494
- raíz, 19, 34
- rendimiento de, 429-432

- como administrador de recursos, 4-5
- como máquina extendida, 3-4
- conceptos de, 13-21
- de capas, 39-40
- distribuido, 12
- en red, 12
- historia de, 5-15
- monolítico, 37-38

- sistemas de paginación, diseño de, 338-343
- software
 - a nivel de usuario, MINIX, 185-186

 - independiente del dispositivo, 162-164
 - independiente del dispositivo, MINIX, 185
 - objetivos del, 159-161

 - de terminal: de entrada, 241-247
 - de terminal: de salida, 247-249
 - solicitud ~t: interrupción, 156-157
 - spooling, 10, 165
 - SSF (*véase* búsqueda más corta primero)
 - sufijo reservado, 105
 - superbloque, MINIX, 456-458, 480-481
 - superposición, 319
 - superusuario, 17

- T**

- tabla
 - de descriptores
 - de interrupciones, Pentium, 124, 143
 - globales, Pentium, 124, 143, 352
 - locales, Pentium, 143, 352-353
 - de páginas:, 322
 - de procesos, 16, 52
 - invertida, 330-331
 - multinivel, 324-327
 - tablas de descriptores, Pentium, 118, 352
 - tablero de ajedrez, 347-348
 - tamaño
 - en tiempo real, 90-91, 224
 - de bloque, 422-423
 - operativo de página, 341-343
 - tarea
 - de alarma síncrona, 228-229, 233
 - del sistema, MINIX, 296-304
 - en MINIX, 94
 - tarjeta matriz, 155
 - tecla de función, MINIX, 287
 - temporizador, 222
 - de vigilancia, 226
 - por lotes, 7-8 TENEX, 437-438
 - terminal
 - con mapa en la memoria, 235-238
 - de E/S, 159-165
 - de mapa de bits, 237
 - inteligente, 239
 - en espacio de usuario, 164-166 RS-232, 238-239
 - X, 239-240
 - texto compartido, MINIX, 359
 - thrashing, 338
 - de entrada, 241-247 tic de reloj, 223
 - de salida, 247-249 tiempo
 - de terminal:, 240-249 compartido, 10-11
 - coordinado universal, 224
 - tipo de archivo, 405-407
 - TLB (*véase* buffer de consulta para traducción)
 - trabajo, 6-8
 - trayectoria de recursos, 176-177
 - TSL, instrucción, 62-63

- U**

- Vid, 17
- unidad de administración de memoria, 320
- VNIX,
 - archivo especial, 163
 - bloque de arranque, 121
 - bloqueo mortal, 172, 186
 - comunicación entre procesos, 75
 - controlador de dispositivo, 183
 - directorio, 413-414, 420-421
 - estructura de procesos, 182
 - hilos, 56

informe de errores, 56
 seguridad, 436, 442-443

UTC (*véase* tiempo coordinado universal)

V

vaciado
 de nú]cleo, MINIX, 373
 incrementa], 425
 vaciados para depuración, MINIX, 295-296
 variab]e
 de candado, 60
 de condición, 70

historia, 11 -13 vector de interrupción, 52
 verificación de autenticidad de usuarios, 442-446
 vincu]ación con un archivo, 33-34, 496-497
 sistema de archivos, 402-407, 4]8, 433 virus, 440-44]

X

XDS 940, 87

Z

zona, 456-460
 Zuse Konrad 6



Segunda edición

SISTEMAS OPERATIVOS

Diseño e implementación

ANDREW S. TANENBAUM • ALBERT S. WOODHULL

La segunda edición de este popular texto introductorio sobre sistemas operativos es el único libro de texto que conjuga con éxito teoría y práctica. Los autores logran este importante objetivo tratando primero todos los conceptos fundamentales de sistemas operativos como procesos, comunicación entre procesos, entrada/salida, memoria virtual, sistemas de archivos y seguridad. A continuación, estos principios se ilustran con la ayuda de un sistema operativo tipo UNIX, llamado MINIX, pequeño, pero real, que permite al estudiante poner a prueba sus conocimientos en proyectos de diseño de sistemas prácticos. El libro incluye un CD-ROM que contiene el código fuente completo de MINIX y dos simuladores para ejecutar MINIX en diversos computadores.

Características

- El único texto de sistemas operativos que primero explica los principios importantes y luego muestra cómo se aplican a los sistemas reales usando MINIX como ejemplo detallado.
- Las secciones pertinentes del código de MINIX se describen con detalle en la mayor parte de los capítulos.
- Cada capítulo incluye una sección de problemas al final.
- Como ayuda para el profesor se incluye la disponibilidad de todas las ilustraciones en Web en un formato apropiado para crear diapositivas.

Novedades de esta edición

- El material sobre principios se actualizó para reflejar los nuevos avances en el campo.
- El sistema MINIX revisado, basado en POSIX, ahora se ejecuta en máquinas basadas en Pentium y en todas sus predecesoras hasta la 8088 original.
- Código MINIX para trabajo con redes basado en Ethernet y TCP/IP.
- El CD-ROM incluido en el libro contiene:
 - el código fuente completo de MINIX descrito en el texto
 - el código fuente completo para controladores opcionales no tratados en el texto, incluidas interfaces para CD-ROM, Ethernet, RS-232 y SCSI
 - el código fuente completo de la implementación TCP/IP de MINIX
 - instrucciones para instalar MINIX en una PC
 - simuladores para ejecutar MINIX en estaciones de trabajo UNIX y otros sistemas

