6

SISTEMAS DE FICHEROS

Todas las aplicaciones necesitan almacenar y recuperar información. Mientras un proceso está ejecutándose puede almacenar cierta cantidad de información dentro de su propio espacio de direcciones. Sin embargo, esa capacidad de almacenamiento está limitada por el tamaño del espacio de direcciones virtual. Para algunas aplicaciones ese tamaño es adecuado, pero para otras, tales como la reserva de billetes de avión, la banca o el registro de las operaciones realizadas por una empresa, resulta demasiado pequeño.

Un segundo problema con el que nos encontramos al guardar la información dentro del espacio de direccionamiento de un proceso es que cuando el proceso termina, la información se pierde. Para muchas aplicaciones (por ejemplo para las bases de datos) la información debe ser retenida durante semanas, meses o incluso para siempre. Es inaceptable permitir que la información se desvanezca cuando termina el proceso que la utiliza. Además, tampoco debe perderse aunque el proceso se destruya repentinamente debido a una caída del sistema.

Un tercer problema es que frecuentemente es necesario que múltiples procesos accedan a (partes de) la información al mismo tiempo. Si disponemos de una guía telefónica almacenada dentro del espacio de direccionamiento de un único proceso, sólo ese proceso va a poder acceder a ella. La manera de resolver este problema es hacer que la información sea ella misma independiente de cualquier proceso.

Entonces tenemos ya tres requerimientos esenciales para el almacenamiento a largo plazo de la información:

1. Debe poder almacenarse una cantidad de información muy grande.
2. La información debe permanecer tras la terminación del proceso que la usa.
3. Debe permitir que múltiples procesos puedan acceder a la información concurrentemente

La solución usual a todos estos problemas es almacenar la información sobre discos y otros medios externos en unidades denominadas **ficheros**. Los procesos pueden entonces leerlos y crear nuevos ficheros si es necesario. La información almacenada en los ficheros debe ser persistente, esto es, no debe verse afectada por la creación y terminación de los procesos. Un fichero sólo puede desaparecer cuando su propietario lo borre de forma explícita.

Los ficheros están gestionados por el sistema operativo. La forma en la cual están estructurados, cómo se nombran, se acceden, se utilizan, se protegen e implementan son temas principales en el diseño de los sistemas operativos. Globalmente, a esa parte del sistema operativo que trata los ficheros se la conoce como el **sistema de ficheros** y es el tema de este capítulo.

Desde el punto de vista de los usuarios, el aspecto más importante de un sistema de ficheros es su apariencia, es decir, qué constituye un fichero, como se nombran y se protegen los ficheros, qué operaciones se permiten, etc. Los detalles de si para seguir la pista de la memoria libre se utilizan listas enlazadas o mapas de bits, o el detalle de cuántos sectores hay en un bloque lógico, son cuestiones de menos interés, aunque son de gran importancia para los

diseñadores del sistema de ficheros. Por esa razón, hemos estructurado el capítulo en varias secciones. Las dos primeras secciones tienen que ver con la interfaz del usuario con los ficheros y con los directorios, respectivamente. A continuación se discutirá en detalle la forma en la cual se implementa el sistema de ficheros. Finalmente, daremos algunos ejemplos de sistemas de ficheros reales.

6.1 FICHEROS

En las páginas siguientes examinaremos los ficheros desde la perspectiva del usuario; es decir, cómo se utilizan y qué propiedades tienen.

6.1.1 Nombres de fichero

Los ficheros son un mecanismo de abstracción que permite almacenar información en el disco y leerla después. Esto debe hacerse de tal modo que el usuario no tenga que enterarse de los detalles de cómo y dónde está almacenada la información, y de cómo funcionan en realidad los discos.

Tal vez la característica más importante de cualquier mecanismo de abstracción es la forma en la que se da nombre a los objetos que se manejan, así que comenzaremos nuestro estudio de los sistemas de ficheros con el tema de los nombres de fichero. Cuando un proceso crea un fichero, le asigna un nombre. Cuando el proceso termina, el fichero sigue existiendo y otros programas pueden tener acceso a él utilizando su nombre.

Las reglas exactas para nombrar ficheros varían un tanto de un sistema a otro, pero todos los sistemas operativos actuales permiten usar cadenas de una a ocho letras como nombres de fichero válidos. Así *andrea*, *bruce* y *cathy* son posibles nombres de fichero. Es común que se permitan también dígitos y caracteres especiales, de modo que nombres como *2*, *urgent!* y *Fig.2-14* también son válidos en muchos casos. Muchos sistemas de ficheros reconocen nombres de hasta 255 caracteres de longitud.

Algunos sistemas de ficheros distinguen entre mayúsculas y minúsculas, pero otros no. UNIX pertenece a esta primera categoría; MS-DOS, a la segunda. Por tanto, en un sistema UNIX los siguientes nombres corresponden a tres ficheros distintos: *maria*, *Maria* y *MARIA*. En MS-DOS, todos esos nombres se refieren al mismo fichero.

Quizá valga la pena hacer aquí una pequeña digresión en lo tocante a los nombres de fichero. Tanto Windows 95 como Windows 98 utilizan el sistema de ficheros de MS-DOS, y por lo tanto heredaron muchas de sus propiedades, como la forma de construir nombres de fichero. Además, Windows NT y Windows 2000 reconocen el sistema de ficheros de MS-DOS y por ende también heredan sus propiedades. Sin embargo, los últimos dos sistemas también tienen un sistema de ficheros nativo (NTFS) que tiene diferentes propiedades (como nombres de fichero en Unicode). En este capítulo, cuando nos refiramos al sistema de ficheros de Windows, estaremos hablando del sistema de ficheros de MS-DOS, que es el único que reconocen todas las versiones de Windows. Trataremos el sistema de ficheros nativo de Windows 2000 en el capítulo 11.

Muchos sistemas de ficheros manejan nombres de fichero con dos partes, separadas por un punto, como en *prog.c*. La parte que sigue al punto se denomina **extensión del fichero**, y normalmente indica algo acerca del fichero. En MS-DOS, por ejemplo, los nombres de fichero tienen de uno a ocho caracteres, más una extensión opcional de uno a tres caracteres. En UNIX, el tamaño de la extensión, si la hay, se deja al criterio del usuario, y un fichero podría incluso tener dos o más extensiones, como en *prog.c.Z*, donde *.Z* se utiliza por lo común para indicar que el fichero (*prog.c*) se comprimió utilizando el algoritmo de compresión Ziv-Lempel. En la Figura 6-1 se presentan algunas de las extensiones de fichero más comunes y su significado.

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Figura 6-1. Algunas extensiones de fichero comunes.

En algunos sistemas (como UNIX) las extensiones de fichero son sólo un convenio y el sistema operativo no vigila que se utilicen de alguna manera específica. Un fichero llamado *fichero.txt* puede ser algún tipo de fichero de texto, pero el nombre sirve más para recordar ese hecho a su propietario que para comunicar alguna información real al ordenador. En cambio, un compilador de C podría insistir en que cualquier fichero que vaya a compilar termine en *.c*, y podría negarse a compilarlo de no ser así.

Los convenios de este tipo son útiles en especial cuando el mismo programa puede manejar varios tipos de ficheros distintos. Por ejemplo, podría suministrarse al compilador de C una lista de varios ficheros que debe compilar y enlazar, algunos de ellos en C y otros en lenguaje ensamblador. En tal caso, la extensión se vuelve indispensable para que el compilador sepa cuáles son los ficheros en C, cuáles están en lenguaje ensamblador y cuáles son de otro tipo.

En contraste, Windows tiene conocimiento de las extensiones y les asigna un significado. Los usuarios (o procesos) pueden registrar extensiones de cara al sistema operativo y especificar, para cada una, qué programa es el "dueño" de la extensión. Cuando un usuario hace doble clic sobre un nombre de fichero, se inicia el programa asociado a su extensión de fichero, con el nombre de fichero como parámetro. Por ejemplo, si se hace doble clic en *fichero.doc*, se iniciará el programa Word de Microsoft, y éste abrirá fichero.doc como primer documento a editar.

6.1.2 Estructura de los ficheros

Los ficheros pueden estructurarse de varias maneras. En la Figura 6-2 se ilustran tres posibilidades comunes. El fichero de la Figura 6-2(a) es una sucesión no estructurada de bytes. En efecto, el sistema operativo no sabe qué contiene el fichero, ni le interesa; lo único que ve son bytes. Cualquier significado que tenga el fichero deberán atribuírselo los programas en el nivel de usuario. Tanto UNIX como Windows utilizan este enfoque.

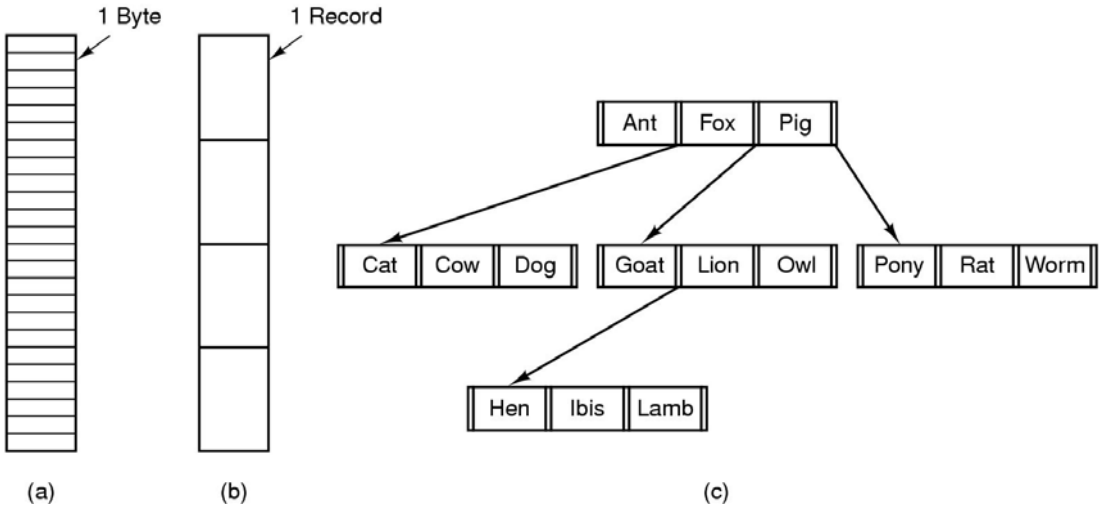

Figura 6-2. Tres tipos de ficheros. (a) Sucesión de bytes. (b) sucesión de registros. (c) Árbol.

Hacer que el sistema operativo vea los ficheros únicamente como sucesiones de bytes ofrece el máximo de fexibilidad. Los programas de usuario pueden colocar lo que deseen en sus ficheros y darles el nombre que les convenga. El sistema operativo no ayuda, pero tampoco estorba. Esto es muy importante para los usuarios que desean hacer cosas fuera de lo común.

El primer paso de estructuración se muestra en la Figura 6-2(b). En este modelo, un fichero es una sucesión de registros de longitud fija, cada uno de los cuales tiene cierta estructura interna. Un aspecto fundamental de la idea de que un fichero es una sucesión de registros es la idea de que la operación de lectura devuelve un registro y que la operación de escritura sobrescribe o añade un registro. Como nota histórica, en las décadas pasadas, cuando reinaba la tarjeta perforada de 80 columnas, muchos sistemas operativos de mainframe basaban sus sistemas de ficheros en ficheros formados por registros de 80 caracteres, que de hecho eran imágenes de tarjetas. Esos sistemas también reconocían ficheros de registros de 132 caracteres, destinados a impresoras (que en esa época eran grandes impresoras de cadena que imprimían 132 columnas). Los programas leían las entradas en unidades de 80 caracteres y escribían sus salidas en unidades de 132 caracteres, aunque los últimos 52 podían ser espacios, claro. Ningún sistema actual de uso general funciona ya así.

El tercer tipo de estructura de fichero se muestra en la Figura 6-2(c). En esta organización un fichero consiste en un árbol de registros, no todos necesariamente de la misma longitud , cada uno de los cuales contiene un campo **clave** en una posición fija del registro. El árbol está ordenado según el campo clave, con objeto de poder hallar con rapidez una clave en particular.

La operación básica aquí no es obtener el "siguiente" registro, aunque también puede hacerse, sino obtener el que tenga una clave dada. En el caso del fichero del zoológico de la Figura 6-2(c), se le podría pedir al sistema que obtenga el registro cuya clave es *Pony*, por ejemplo, sin preocuparse por su posición exacta en el fichero. Además, es posible añadir registros nuevos al fichero y dejar que sea el sistema operativo, no el usuario, quien decida dónde colocarlos. Es obvio que este tipo de fichero es muy distinto de los flujos de bytes no estructurados que se usan en UNIX y Windows, pero se utiliza en forma amplia en los grandes ordenadores mainframe que todavía se emplean en el procesamiento comercial de datos.

6.1.3 Tipos de ficheros

Muchos sistemas operativos reconocen varios tipos de ficheros. UNIX y Windows, por ejemplo, tienen ficheros regulares y directorios. UNIX también tiene ficheros especiales de bloques y de caracteres. Los **ficheros regulares** son los que contienen información del usuario. Todos los ficheros de la Figura 6-2 son ficheros regulares. Los **directorios** son ficheros del sistema que sirven para mantener la estructura del sistema de ficheros, y los estudiaremos más adelante. Los **ficheros especiales de caracteres** tienen que ver con la entrada/salida, y sirven para modelar dispositivos de E/S de tipo serie como terminales, impresoras y redes. Los **ficheros especiales de bloques** sirven para modelar discos. En este capítulo nos referiremos primordialmente a los ficheros regulares.

Los ficheros regulares son normalmente ficheros ASCII o ficheros binarios. Los ficheros ASCII consisten en líneas de texto. En algunos sistemas, cada línea termina con un carácter de retorno de carro; en otros se usa el carácter de salto de línea. Algunos sistemas (como MS-DOS) utilizan ambos. No es necesario que todas las líneas sea de la misma longitud.

La gran ventaja de los ficheros ASCII es que pueden visualizarse e imprimirse tal cual, y pueden editarse con cualquier editor de texto. Además, si un gran número de programas utiliza ficheros ASCII como su entrada y su salida, es fácil conectar la salida de un programa con la entrada de otro, como en las tuberías del shell. (La fontanería entre procesos no es nada fácil, pero interpretar la información ciertamente sí que lo es si se utiliza un convenio estándar para expresarla, tal como ASCII.)

Otros ficheros son binarios, lo que significa simplemente que no son ficheros ASCII. Si se escriben en una impresora se produce un listado incomprensible que parece estar lleno de basura. Normalmente, estos ficheros tienen alguna estructura interna conocida por los programas que los usan.

Por ejemplo en la Figura 6-3(a) vemos un fichero binario ejecutable sencillo tomado de una versión de UNIX. Aunque desde el punto de vista técnico el fichero no es más que una sucesión de bytes, el sistema operativo sólo puede ejecutar un fichero si éste tiene el formato correcto. Este fichero 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**, el cual identifica el fichero como ejecutable (para evitar la ejecución accidental de un fichero que no tenga este formato). Luego vienen los tamaños de los diversos componentes del fichero, la dirección de la primera instrucción a ejecutar y algunos bits que actúan como indicadores. Después del encabezado vienen el texto y los datos del programa propiamente dicho. Éstos se cargan en la memoria y se reubican empleando los bits de reubicación. La tabla de símbolos sirve para depurar el programa.

Nuestro segundo ejemplo de fichero binario es también un fichero de UNIX. Consiste de una colección de procedimientos de biblioteca (módulos) compilados pero sin enlazar. Cada procedimiento va precedido por un encabezado que indica su nombre, la fecha en que se creó, el propietario, un código de protección y el tamaño. Al igual que en el fichero ejecutable, los encabezados de módulo están llenos de números binarios. Si se escribieran por una impresora se obtendría algo ilegible.

Todo sistema operativo debe reconocer al menos un tipo de fichero: sus propios ficheros ejecutables, pero algunos reconocen más. El viejo sistema TOPS-20 (para el sistema DEC 20) llegaba al extremo de examinar la hora de creación de cualquier fichero a ejecutar. Luego buscaba el fichero fuente y veía si había sido modificado desde la hora en que se había creado el binario. En tal caso, recompilaba automáticamente el fichero fuente. En términos de UNIX, sería como si el programa *make* se hubiera integrado en el shell. Las extensiones de fichero eran

obligatorias para que el sistema pudiera saber qué programa binario se había obtenido de qué fichero fuente.

Tener ficheros fuertemente tipados de esta forma provoca problemas cuando el usuario hace algo que los diseñadores del sistema no contemplaron. Consideremos, por ejemplo, un sistema en el que los ficheros de salida de los programas tienen la extensión *.dat* (ficheros de datos). Si un usuario escribe un formateador de programas que lee un fichero *.c* (programa en C), lo convierte (por ejemplo, a un esquema de sangrado convencional) y luego escribe el programa convertido como salida, el fichero de salida será de tipo *.dat*. Si el usuario trata de compilar este fichero con el compilador de C, el sistema se negará porque no tiene la extensión correcta. El sistema rechazará también cualquier intento por copiar *fichero.dat* en *fichero.c* por considerarlo incorrecto (para proteger al usuario contra equivocaciones).

Aunque esta "amigabilidad con el usuario" podría ayudar a los novatos, exaspera a los usuarios experimentados, porque deben dedicar un tiempo considerable a buscar formas de sustraerse al concepto que tiene el sistema operativo de lo que es razonable y lo que no lo es.

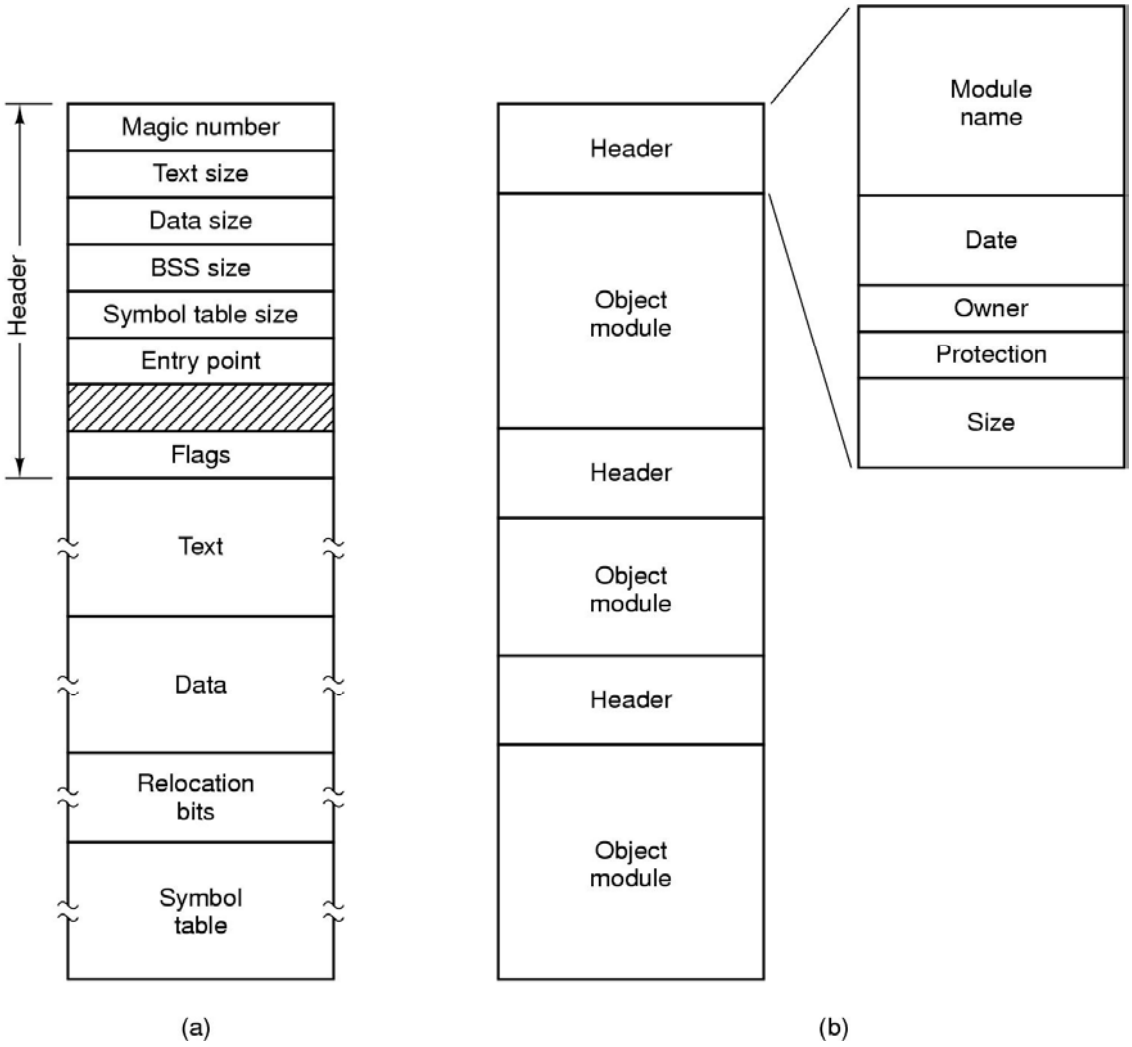

Figura 6-3. (a) Fichero ejecutable. (b) Un fichero de biblioteca.

6.1.4 Acceso a ficheros

Los primeros sistemas operativos sólo permitían un tipo de acceso a los ficheros: **acceso secuencial**. En aquellos sistemas, un proceso podía leer todos los bytes o registros de un fichero por orden, comenzando por el principio, pero no podía efectuar saltos para leerlos en otro orden. Lo que sí podía hacerse con los ficheros secuenciales era "rebobinarlos" para poder leerlos tantas veces como se quisiera. Los ficheros secuenciales eran apropiados cuando el medio de almacenamiento era la cinta magnética, no el disco.

Cuando comenzaron a usarse discos para almacenar ficheros se hizo posible leer los bytes o registros de un fichero sin un orden específico, o tener acceso a los registros por clave, no por posición. Los ficheros cuyos bytes o registros pueden leerse en cualquier orden se denominan ficheros de acceso aleatorio, y muchas aplicaciones los necesitan.

Los ficheros de acceso aleatorio son indispensables en muchas aplicaciones, como los sistemas de bases de datos. Si un cliente de una línea aérea llama para reservar un asiento en un vuelo dado, el programa de reservas deberá contar con la capacidad de acceder al registro de ese vuelo sin tener que leer primero los demás miles de vuelos existentes.

Se utilizan dos métodos para especificar dónde debe comenzar la lectura. En el primero, cada operación read da la posición en el fichero dónde debe comenzarse a leer. En el segundo, se cuenta con una operación especial, seek, para establecer la posición actual. Después del seek, el fichero podrá leerse de forma secuencial a partir de la posición que ahora es la actual.

En algunos sistemas operativos de mainframe antiguos, los ficheros se clasifican como secuenciales o de acceso aleatorio en el momento en que se crean. Esto permite al sistema emplear técnicas de almacenamiento distintas para las dos clases. Los sistemas operativos modernos no hacen esta distinción; todos los ficheros son de acceso aleatorio de forma automática.

6.1.5 Atributos de los ficheros

Todo fichero tiene un nombre y datos. Además, todos los sistemas operativos asocian otra información a cada fichero, como la fecha y la hora en que se creó, y su tamaño. Llamaremos a esta información adicional **atributos** del fichero. La lista de atributos varía de manera considerable de un sistema a otro. La tabla de la Figura 6-4 muestra algunas de las posibilidades, pero existen otras. Ningún sistema actual maneja todos estos atributos, pero todos están presentes en algún sistema.

Los primeros cuatro atributos tienen que ver con la protección del fichero e indican quién puede tener acceso a él y quien no. Es posible usar todo tipo de esquema, algunos de los cuales estudiaremos más adelante. En algunos sistemas el usuario debe presentar una contraseña para el acceso a un fichero, en cuyo caso la contraseña deberá ser uno de los atributos.

Los indicadores son bits o campos cortos que controlan o habilitan alguna propiedad específica. Los ficheros ocultos, por ejemplo, no aparecen en los listados de todos los ficheros. El indicador de archivado es un bit que indica si el fichero ya se respaldó o no. El programa de respaldo lo establece a 0 y el sistema lo pone a 1 cada vez que se modifica el fichero. Así, el programa de respaldo sabe qué ficheros deben respaldarse. El indicador temporal permite marcar un fichero para que se borre de forma automática cuando termine el proceso que lo creó.

Los campos de longitud del registro, posición de la clave y longitud de la clave sólo están presentes en ficheros cuyos registros pueden consultarse empleando una clave. Dichos campos proporcionan la información necesaria para hallar las claves.

Las diversas horas llevan el control de cuándo se creó el fichero, cuándo fue la última vez que se tuvo acceso a él y cuando fue la última vez que se modificó. Son útiles para varias cosas. Por ejemplo, si un fichero fuente se modificó después de crear el fichero objeto correspondiente, será necesario recompilarlo. Estos campos proporcionan la información necesaria.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figura 6-4. Algunos posibles atributos de un fichero.

El tamaño actual indica lo grande que es un fichero. Algunos sistemas operativos de mainframe antiguos exigen que se especifique el tamaño máximo cuando se crea un fichero, para poder reservar por adelantado la cantidad máxima de espacio de almacenamiento. Los sistemas operativos de estaciones de trabajo y ordenadores personales son lo bastante inteligentes como para prescindir de esa información.

6.1.6 Operaciones con ficheros

Los ficheros existen para guardar información y poder recuperarla después. Los distintos sistemas ofrecen diferentes operaciones de almacenamiento y recuperación. A continuación estudiaremos las llamadas al sistema más comunes relacionadas con los ficheros.

1. **Create**. Se crea el fichero sin datos. El objetivo de la llamada es anunciar que va a haber un fichero nuevo y establecer algunos de sus atributos.

2. **Delete**. Si ya no se necesita un fichero, conviene borrarlo para desocupar el espacio en disco. Siempre hay una llamada al sistema para ese fin.

3. **Open**. Antes de usar un fichero, un proceso debe abrirlo. El propósito de la llamada open es que el sistema obtenga los atributos y la lista de direcciones de disco y los

coloque en la memoria principal para tener acceso a ellos rápidamente en llamadas posteriores.

4. **Close**. Una vez que han terminado todos los accesos, ya no se necesitarán los atributos y direcciones en disco, por lo que es recomendable cerrar el fichero para desocupar espacio en las tablas internas. Muchos sistemas fomentan esto imponiendo un límite para el número de fichero que pueden tener abiertos los procesos. Los discos se escriben en bloques, y el cierre de un fichero hace que se escriba el último bloque del fichero, aunque no esté lleno por completo.

5. **Read**. Se leen datos de un fichero. Normalmente, los bytes provienen de la posición actual. Quien efectúa la llamada debe especificar cuántos datos necesita, y el búfer donde deben colocarse.

6. **Write**. Se escriben datos en un fichero, también, normalmente, en la posición actual. Si la posición actual es el fin del fichero, aumenta el tamaño del fichero. Si la posición actual está en un punto intermedio del fichero, los datos existentes se sobrescriben y se perderán sin remedio.

7. **Append**. Esta llamada es una forma restringida de write; con ella sólo se puede agregar datos al final del fichero. Los sistemas que ofrecen un número mínimo de llamadas al sistema por lo general no tienen append, pero muchos sistemas ofrecen varias formas de hacer lo mismo, y en algunos casos cuentan con append.

8. **Seek**. En el caso de ficheros de acceso aleatorio, se requiere alguna forma de especificar el punto del fichero de donde se tomarán los datos. Un método común es usar una llamada al sistema, seek, que sitúe el puntero del fichero en un lugar específico del fichero. Una vez ejecutada esta llamada, podrán leerse datos de esa posición, o escribir en ella.

9. **Get attributes**. Muchas veces los procesos necesitan leer los atributos de un fichero para efectuar su trabajo. Por ejemplo, el programa *make* de UNIX se usa por lo común para administrar proyectos de desarrollo de software que contienen muchos ficheros fuente. Cuando se invoca a *make* se examinan los tiempos de modificación de todos los ficheros fuente y objeto y se determina el número mínimo de compilaciones necesarias para que todo esté actualizado. Para efectuar su trabajo, el sistema debe examinar atributos, a saber, las horas de modificación.

10. **Set attributes**. El usuario puede establecer algunos de los atributos, o modificarlos después de que se creó el fichero, y eso se logra con esta llamada al sistema. La información de modo de protección es un ejemplo obvio. Casi todos los indicadores pertenecen también a esa categoría.

11. **Rename**. Es común que un usuario necesite cambiar el nombre de un fichero existente. Esta llamada al sistema lo hace posible. No siempre es estrictamente necesaria, pues por lo general el fichero puede copiarse en un fichero nuevo con el nuevo nombre, borrando después el fichero viejo.

6.1.7 Ejemplo de programa que utiliza llamadas al sistema de ficheros

En esta sección examinaremos un sencillo programa en UNIX que copia un fichero de su fichero origen a su fichero de destino. El listado aparece en la Figura 6-5. El programa tiene una funcionalidad mínima y sus informes de errores son más rudimentarios todavía, pero da una

idea razonable de cómo funcionan algunas de las llamadas al sistema relacionadas con los ficheros.

El programa, *copyfile*, puede invocarse, por ejemplo, con el siguiente comando

```
copyfile abc xyz
```

para copiar el fichero *abc* en *xyz*. Si *xyz* ya existe, se sobrescribe; si no, se crea. El programa debe invocarse exactamente con dos argumentos, ambos nombres de fichero válidos.

Las cuatro directivas *#include* cerca de la parte superior del programa hacen que se incluya un gran número de definiciones y prototipos de función en el programa. Son necesarios para que el programa se ajuste a las normas internacionales pertinentes, pero no nos ocuparemos más de ellos. La siguiente línea es un prototipo de función para *main*, algo que exige el ANSI C, pero que tampoco es importante para nuestros fines.

La primera directiva *#define* es una definición de macro que especifica la cadena *BUF_SIZE* como una macro que se expande al número 4096. El programa leerá y escribirá en porciones de 4096 bytes. Se considera una buena práctica de programación asignar nombres a las constantes de este tipo y usar los nombres en lugar de las constantes. Este convenio no sólo facilita la lectura de los programas, sino que facilita también su mantenimiento. La segunda directiva *#define* determina quién puede tener acceso al fichero de salida.

El programa principal se llama *main* y tiene dos argumentos, arc y argv. Éstos los proporciona el sistema operativo cuando se invoca al programa. El primero indica cuantas cadenas estaban presentes en la línea del comando introducido para ejecutar el programa, incluido el propio nombre del programa. En este caso deberá ser 3. El segundo parámetro es un array de punteros a los argumentos. En el ejemplo de llamada dado aquí, los elementos de dicho array contendrían punteros a los siguientes valores:

argv[0] = “copyfile”
argv[0] = “abc”
argv[0] = “xyz”

Por medio de este array el programa tiene acceso a sus argumentos.

Se declaran cinco variables. Las dos primeras, *in_fd* y *out_fd*, contienen los **descriptores de fichero**, número enteros pequeños que se devuelven cuando se abre un fichero. Las dos siguientes, *rd_count* y *wt_count*, son los contadores de bytes devueltos por las llamadas al sistema *read* y *write*, respectivamente. La última, *buffer*, es el búfer empleado para contener los datos leídos y suministrar los datos a escribir.

La primera instrucción propiamente dicha verifica si *argc* es 3. Si no, termina el programa con el código de estado 1. Cualquier código de estado distinto de 0 implica que hubo un error. El código de estado es el único informe de errores que produce este programa. Una versión de producción normalmente debería imprimir también los correspondientes mensajes de error.

```
/* Programa de copia de ficheros. */
/* Con una mínima comprobación y comunicación de errores */

#include <sys/types.h>       /* se incluyen los ficheros de cabecera necesarios */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int main ( int argc, char * argv[]) ;  /* prototipo ANSI */
#define BUF_SIZE 4096              /* tamaño de bufer de 4096 bytes */
#define OUTPUT_MODE 0700     /* bits de proteccion para el fichero de salida */
int main ( int argc, char * argv[])
{
        int in_fd, out_fd, rd_count, wt_count ;
        char buffer[BUF_SIZE] ;

        if (argc != 3) exit(1) ;         /* error de sintaxis si argc no es 3 */

        /* Abre el fichero de entrada y crea el fichero de salida */
        in_fd = open(argv[1], O_RONLY) ;            /* abre el fichero de origen */
        if (in_fd < 0) exit(2) ;                     /* si no se puede terminar */
        out_fd = creat(argv[2], OUTPUT_MODE) ; /* crea el fichero de destino */
        if (out_df < 0) exit(3) ;                    /* si no se puede terminar */

        /* bucle de copia */
        while (TRUE) {
                rd_count = read(in_fd, buffer, BUF_SIZE) ; /* lee bloque de datos */
        if (rd_count <= 0) break ;           /* si fin de fichero o error, salir del bucle */
                wr_count = write(out_fd, buffer, rd_count) ;          /* escribe datos */
                if (wt_count <= 0) exit(4) ;                /* wt_count <= 0 es un error */
        }

        /* Cierra los ficheros */
        close(in_fd) ;
        close(out_fd) ;
        if (rd_count) == 0)
                exit(0) ;                   /* no hubo error en la ultima lectura */
        else
                exit(5) ;                   /* hubo un error en la ultima lectura */
}
```

Figura 6-5. Programa sencillo para copiar un fichero.

Luego se intenta abrir el fichero de origen y crear el de destino. Si se logra abrir el fichero de origen, el sistema asigna un entero pequeño a *in_df* para identificar el fichero. Las llamadas subsiguientes deberán incluir este entero para que es sistema sepa qué fichero quieren. De forma similar, si se logra crear el fichero de destino, *out_fd* recibe un valor que lo identifica. El segundo argumento de *creat* establece el modo de protección. Si fracasa la apertura o bien la creación, se asigna -1 al descriptor del fichero correspondiente, y el programa termina con un código de error.

Ahora viene el bucle de copiado. Lo primero que hace es tratar de leer 4 KB de datos y colocarlos en el búfer. Esto lo hace invocando el procedimiento de biblioteca read, que es el que emite la llamada al sistema *read*. El primer parámetro identifica el fichero, el segundo identifica el búfer y el tercero indica cuántos bytes hay que leer. El valor asignado a *rd_count* es el número de bytes que se leyeron en realidad. Comúnmente, este número será 4096, a menos que queden menos bytes en el fichero. Cuando se llegue al final del fichero, el valor será cero. Si *rd_count* llega a ser cero o negativo, la copia no podrá continuar, así que se ejecutará la instrucción *break* para salir del bucle (que de otro modo sería infinito).

La llamada a *write* envía el contenido del búfer al fichero de destino. El primer parámetro identifica el fichero, el segundo da el búfer y el tercero indica cuántos bytes hay que

escribir, de forma análoga a *read*. Cabe señalar que el contador de bytes es el número que se leyó en realidad, no *BUF_SIZE*. Este punto es importante porque la última lectura no devolverá 4096, a menos que el tamaño del fichero sea un múltiplo de 4 KB.

Una vez que se haya procesado todo el fichero, la primera llamada que rebase el fin del fichero asignará 0 a *rd_count*, con lo cual se saldrá del bucle. En este punto los dos ficheros se cierran y el programa termina con un código de estado que indica que lo hizo de forma normal.

Aunque las llamadas al sistema de Windows son diferentes de las de UNIX, la estructura general de un programa de Windows de línea de comandos para copiar un fichero es más o menos similar al de la Figura 6-5. Examinaremos las llamadas de Windows 2000 en el capítulo 11.

6.1.8 Ficheros con correspondencia en memoria

Muchos programadores opinan que tener acceso a los ficheros como acabamos de ver es un método torpe y poco recomendable, sobre todo si se le compara con el acceso a la memoria ordinaria. Por ese motivo, algunos sistemas operativos, comenzando por MULTICS, han incluido un mecanismo para establecer una correspondencia entre los ficheros y el espacio de direcciones de un proceso en ejecución. Desde el punto de vista conceptual, podemos imaginar la existencia de dos nuevas llamadas al sistema, map y unmap. La primera proporciona un nombre de fichero y una dirección virtual, y hace que el sistema operativo establezca una correspondencia del fichero con el espacio de direcciones a partir de la dirección virtual.

Por ejemplo, supongamos que se establece una correspondencia entre un fichero, f, cuya longitud es 64 KB, y el espacio de direcciones virtual a partir de la dirección 512 K. Entonces cualquier instrucción de máquina que lea el contenido del byte que está en 512 K obtendrá el byte 0 del fichero, y así de forma sucesiva. Del mismo modo, una escritura en la dirección 512 K + 21000 modificará el byte 21000 del fichero. Cuando termina el proceso, el fichero modificado queda en el disco, como si hubiera sido modificado por una combinación de llamadas al sistema seek y write.

Lo que sucede en verdad es que las tablas internas del sistema se modifican de modo que el fichero se convierta en el almacén de respaldo para la región de memoria que está entre 512 K y 576 K. Así, una lectura desde 512 K genera una falta de página y hace que se traiga la página 0 del fichero. De forma similar, una escritura en 512 K + 1100 provoca un fallo de página que trae a la memoria la página que contiene esa dirección, después de lo cual puede efectuarse la escritura en memoria. Si esa página llega a ser desalojada por el algoritmo de sustitución de páginas se escribe en el lugar correcto del fichero. Cuando termina el proceso, todas las páginas con correspondencia que se hayan modificado se escriben de vuelta en sus ficheros.

La correspondencia de ficheros funciona de manera óptima en un sistema que maneja segmentación. En un sistema así, cada fichero puede establecer una correspondencia con su propio segmento para que el byte *k* del fichero sea también el byte *k* del segmento. En la Figura 6-6(a) vemos un proceso que tiene dos segmentos, uno de código y otro de datos. Supongamos que este proceso copia ficheros, como el programa de la Figura 6-5. Primero establece una correspondencia entre el fichero de origen, digamos *abc*, y un segmento. Luego crea un segmento vacío y establece una correspondencia entre éste y el fichero de destino, que en nuestro caso es *xyz*. Estas operaciones producen la situación que se muestra en la Figura 6-6(b).

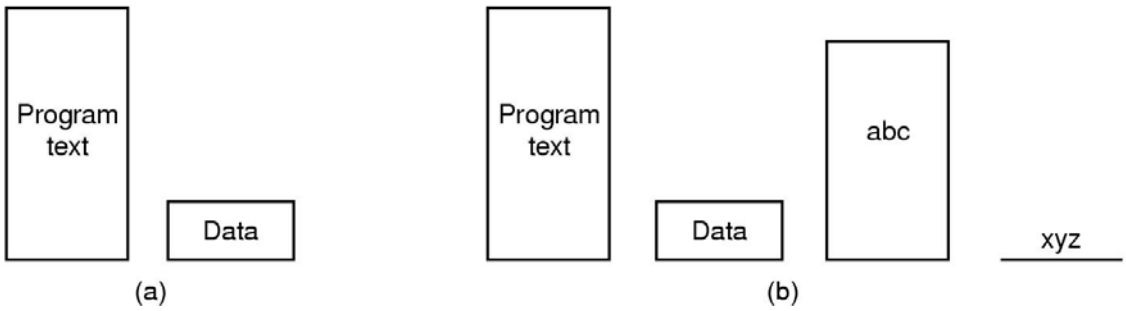

Figura 6-6. (a) Proceso segmentado antes de establecer las correspondencias entre los ficheros y su espacio de direcciones. (b) El proceso después de establecer una correspondencia entre un fichero existente *abc* y un segmento, y crear un segmento nuevo para el fichero *xyx*.

Ahora el proceso puede copiar el segmento de origen en el de destino, utilizando un bucle de copia ordinario. No se necesitan llamadas al sistema read ni write. Una vez que lo ha hecho, el proceso puede ejecutar la llamada al sistema unmap para eliminar los ficheros del espacio de direcciones y luego terminar. Ahora ya existe el fichero de salida, *xyz*, como si se hubiera creado de la forma convencional.

Aunque la correspondencia de ficheros elimina la necesidad de E/S y, por tanto, facilita la programación, presenta algunos problemas. En primer lugar, para el sistema es difícil conocer la longitud exacta del fichero de salida, *xyz* en nuestro ejemplo. Es fácil averiguar el número de la página más grande que se escribió, pero no hay forma de saber cuántos bytes de esa página se escribieron. Supongamos que el programa sólo utiliza la página 0, y que después de la ejecución todos los bytes siguen siendo 0 (su valor inicial). Tal vez *xyz* es un fichero que consta de 10 ceros. Tal vez es un fichero que consta de 100 ceros. Tal vez es un fichero que consta de 1000 ceros. ¿Quién sabe? El sistema operativo no. Lo único que puede hacerse es crear un fichero cuya longitud sea igual al tamaño de página.

Puede presentarse un segundo problema (potencialmente) si un proceso hace corresponder un fichero y otro lo abre para leerlo de la forma convencional. Si el primer proceso modifica una página, ese cambio no se reflejará en el fichero en disco hasta que la página sea desalojada. El sistema debe tener mucho cuidado para asegurar que los dos procesos no vean versiones incongruentes del fichero.

Un tercer problema del establecimiento de la correspondencia es que un fichero podría ser mayor que un segmento, o incluso mayor que todo el espacio de direcciones virtual. La única salida es que la llamada al sistema map pueda hacer corresponder sólo una porción de fichero, no todo el fichero. Aunque esto funciona, es a todas luces menos satisfactorio que hacer corresponder el fichero entero.

6.2 DIRECTORIOS

Para llevar el control de los ficheros, los sistemas de ficheros suelen tener **directorios** o **carpetas** que, en muchos sistemas son a su vez ficheros. En esta sección veremos los directorios, su organización, sus propiedades y las operaciones que pueden realizarse con ellos.

6.2.1 Sistemas de directorios a un solo nivel

La forma más sencilla de sistema de directorios es que un directorio contenga todos los ficheros. A veces se le llama **directorio raíz**, pero dado que es el único, el nombre no importa mucho. En los primeros ordenadores personales este sistema era muy común, en parte porque sólo había un usuario. Resulta interesante que el primer superordenador del mundo, el CDC 6600, también tenía un único directorio para todos los ficheros, aunque la utilizaban muchos usuarios a la vez. Esta decisión sin duda se tomó para que el diseño del software fuera lo más sencillo posible.

En la Figura 6-7 se muestra un ejemplo de sistema con un único directorio. Aquí el directorio contiene cuatro ficheros. En la figura se muestran los *propietarios* de los ficheros, no los *nombres* de los ficheros (porque los propietarios son importantes para lo que vamos a decir). Las ventajas de este esquema son su sencillez y la capacidad para localizar ficheros con rapidez; después de todo sólo pueden estar en un lugar.

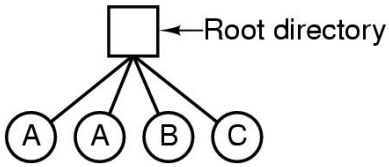

Figura 6-7. Sistema de directorios a un solo nivel que contiene cuatro ficheros, propiedad de tres personas A, B y C.

El problema de tener sólo un directorio en un sistema con múltiples usuarios es que diferentes usuarios podrían usar por accidente los mismos nombres para sus ficheros. Por ejemplo, si el usuario *A* crea un fichero llamado *correo*, y luego el usuario *B* crea también un fichero llamado *correo*, el fichero de *B* sobrescribirá al de *A*. Por ello, este esquema ya no se usa en los sistemas multiusuario, pero podría usarse en un sistema empotrado pequeño, como un sistema en un automóvil, diseñado para almacenar perfiles de un número reducido de conductores.

6.2.2 Sistemas de directorios a dos niveles

Para evitar conflictos cuando dos usuarios escogen el mismo nombre para sus propios ficheros, el siguiente escalón sería dar a cada usuario un directorio privado. Así, los nombres escogidos por un usuario no chocarán con los escogidos por otro, y no habrá problemas si el mismo nombre aparece en dos o más directorios. Este diseño lleva al sistema de la Figura 6-8. Podría usarse, por ejemplo, en un ordenador multiusuario o en una red simple de ordenadores personales que comparten un servidor de ficheros en una red local.

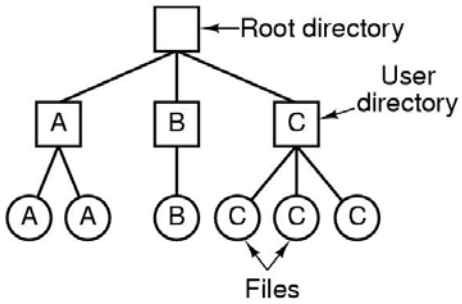

Figura 6-8. Sistema de directorios a dos niveles. Las letras indican los propietarios de los directorios y de los ficheros.

Algo implícito en este diseño es que cuando un usuario trata de abrir un fichero, el sistema necesita saber de qué usuario se trata para saber en qué directorio buscar. Por ello, se requiere algún tipo de procedimiento de inicio de sesión (*login*) en el que el usuario especifica un nombre o una identificación de inicio de sesión, algo que no era necesario en los sistemas de directorios de un nivel.

Cuando se implementa este sistema en su forma más básica, los usuarios sólo pueden tener acceso a ficheros de sus propios directorios. Sin embargo, una extensión del esquema podría permitir a los usuarios el acceso a ficheros de otros usuarios, si indican de quién es el fichero que desean abrir. Por ejemplo,

```
open("x")
```

podría ser la llamada para abrir un fichero llamado *x* en el directorio del usuario, y

```
open("nancy/x")
```

podría ser la llamada para abrir un fichero llamado *x* que está en el directorio de otro usuario, Nancy.

Una situación en la que los usuarios necesitan tener acceso a ficheros distintos de los propios es la ejecución de programas binarios del sistema. Es obvio que sería poco eficiente tener copias de todos los programas de utilidad en cada uno de los directorios. Como mínimo, se necesita un directorio del sistema que contenga los programas binarios ejecutables.

6.2.3 Sistemas de directorios jerárquicos

La jerarquía de dos niveles elimina los conflictos de nombres entre usuarios pero no es satisfactoria para usuarios que tienen un gran número de ficheros. Resulta inadecuada incluso en un ordenador personal con un único usuario. Es muy común que los usuarios quieran agrupar sus ficheros de forma lógica. Un profesor, por ejemplo, podría tener una serie de fichero que juntos integran un libro que está escribiendo para un curso, una segunda colección de ficheros formada por programas que han presentado los estudiantes para otro curso, un tercer grupo de ficheros que contienen el código de un sistema avanzado para escribir compiladores que está desarrollando, un cuarto grupo de ficheros que contienen propuestas de becas, así como otros ficheros de correo electrónico, minutas de reuniones, artículos que está escribiendo, juegos, etc. Se necesita alguna forma de agrupar estos ficheros dentro de esquemas flexibles determinados por el usuario.

Lo que se necesita es una jerarquía general (es decir, un árbol de directorios). Con este enfoque, cada usuario puede tener tantos directorios como necesite para agrupar sus ficheros en categorías naturales. El enfoque se muestra en la Figura 6-9. Aquí, los directorios *A*, *B* y *C* contenidos en el directorio raíz pertenecen cada uno a un usuario distinto, dos de los cuales han creado un subdirectorios para los proyectos en los que están trabajando.

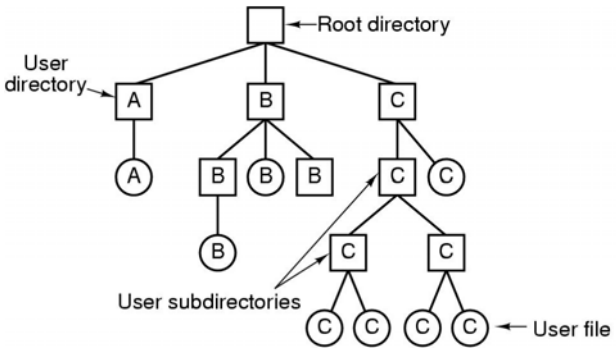

Figura 6-9. Sistema de directorios jerárquico.

6.2.4 Nombres de camino

Cuando un sistema de ficheros está organizado como un árbol de directorios, se necesita un mecanismo para especificar los nombres de fichero. Por lo común se utilizan dos métodos. En el primero, cada fichero recibe un **nombre de camino absoluto** que consiste en el camino que debe seguirse para llegar del directorio raíz hasta el fichero. Por ejemplo, el camino */usr/ast/correo* nos indica que el directorio raíz contiene un subdirectorio, *usr*, que a su vez contiene un subdirectorio, *ast*, que contiene el fichero correo. Los nombres de camino absolutos siempre parten del directorio raíz y son únicos. En UNIX los componentes del camino se separan con /. En Windows el separador es \. En MULTICS era >. Así el mismo nombre de camino se escribiría como sigue en esos tres sistemas:

```
Windows     \usr\ast\correo
UNIX        /usr/ast/correo
MULTICS     >usr>ast>correo
```

Sea cual sea el carácter empleado, si el primer carácter del nombre de camino es el separador, el camino será absoluto.

El otro tipo de nombre es el **nombre de camino relativo**. Éste se utiliza junto con el concepto de **directorio de trabajo** (también llamado **directorio actual**). Un usuario puede designar un directorio como su directorio de trabajo actual, en cuyo caso todos los nombres de camino que no comiencen en el directorio raíz se considerarán relativos al directorio de trabajo. Por ejemplo, si el directorio de trabajo actual es */usr/ast*, podrá hacerse referencia al fichero cuyo camino absoluto es */usr/ast/correo* simplemente mediante correo. Dicho de otro modo, en UNIX el comando

```
cp /usr/ast/correo /usr/ast/correo.bak
```

y el comando

```
cp correo correo.bak
```

hacen exactamente lo mismo si el directorio de trabajo es */usr/ast*. La forma relativa suele ser más conveniente, pero hace lo mismo que la forma absoluta.

Algunos programas necesitan tener acceso a un fichero específico sin importar cuál sea el directorio de trabajo; en tal caso, siempre deberán utilizar nombres de camino absolutos. Por ejemplo, un corrector ortográfico podría tener que leer */usr/lib/diccionario* para realizar su trabajo. En este caso deberá utilizar el nombre de camino absoluto completo porque no sabe

cuál será el directorio de trabajo en el momento en que se necesite el diccionario. El nombre de camino absoluto siempre funciona, sea cual sea el directorio de trabajo.

Desde luego, si el corrector ortográfico necesita un gran número de ficheros de */usr/lib*, una estrategia alternativa sería emitir una llamada al sistema para cambiar su directorio de trabajo a */usr/lib*, y luego utilizar simplemente diccionario como primer parámetro de open. Al cambiar en forma explícita el directorio de trabajo, el programa sabe con certeza en qué parte del árbol de directorios está, y puede utilizar caminos relativos.

Cada proceso tienen su propio directorio de trabajo, así que cuando un proceso cambia su directorio de trabajo y después termina, ningún otro proceso se ve afectado y no quedan rastros del cambio en el sistema de ficheros. De esta forma siempre es perfectamente seguro para un proceso cambiar su directorio de trabajo cuando le convenga. Por otra parte, si un *procedimiento de biblioteca* cambia el directorio de trabajo del programa y al terminar no regresa a donde estaba, es posible que el resto del programa no funciones porque la que se supone su ubicación podría no serlo. Por este motivo, los procedimientos de biblioteca casi nunca cambian el directorio de trabajo, y si tienen que hacerlo, siempre lo restauran antes de terminar.

La mayoría de los sistemas operativos que disponen de sistema de directorio jerárquico tienen dos entradas especiales en cada directorio, "." Y ".." que normalmente se pronuncian "punto" y "punto punto". Punto se refiere al directorio actual; punto punto se refiere a su padre. Para ver cómo se utilizan esas entradas, consideremos el árbol de ficheros UNIX de la Figura 6-10. Cierto proceso tiene */usr/ast* como su directorio de trabajo. Ese proceso puede utilizar ".." para subir por el árbol. Por ejemplo, el proceso puede copiar el fichero */usr/ast/diccionario* a su propio directorio emitiendo el comando

```
cp ../lib/diccionario .
```

El primer camino le indica al sistema que suba en la jerarquía (al directorio *usr*) y luego que baje al directorio *lib* para hallar el fichero *diccionario*.

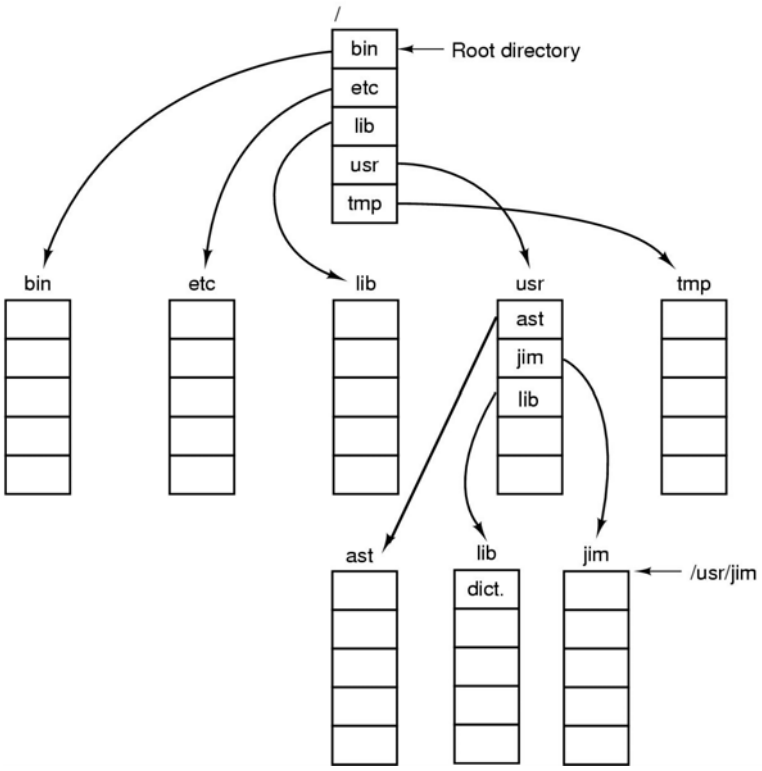

Figura 6-10. Un árbol de directorios UNIX.

El segundo argumento (punto) nombra el directorio actual. Cuando el comando *cp* recibe un nombre de directorio (que puede ser punto) como segundo argumento, copia todos los ficheros ahí. Desde luego, una forma más normal de efectuar la copia sería teclear

```
cp /usr/lib/diccionario .
```

Aquí el uso de punto ahorra al usuario el trabajo de teclear otra vez diccionario. No obstante, teclear

```
cp /usr/lib/diccionario diccionario
```

también funciona a la perfección, lo mismo que

```
cp /usr/lib/diccionario /usr/ast/diccionario
```

Todos estos comandos hacen exactamente lo mismo.

6.2.5 Operaciones con directorios

Las llamadas al sistema que pueden emitirse para administrar los directorios presentan más variaciones entre los diferentes sistemas que las llamadas para administrar los ficheros. Para dar una idea de cuáles son y cómo funcionan, damos la siguiente muestra (tomada de UNIX).

1. **Create**. Se crea un directorio, el cual está vacío excepto por punto y punto punto, que el sistema coloca ahí (o, en algunos casos, el programa *mkdir*).

2. **Delete**. Se elimina un directorio. Sólo es posible eliminar un directorio vacío. Se considera vacío un directorio que sólo contiene los directorios punto y punto punto. Por lo general no pueden borrarse esas entradas.

3. **Opendir**. Los directorios pueden leerse. Por ejemplo, si se desea visualizar la lista de todos los ficheros contenidos en un directorio, el programa que los visualiza debe abrir el directorio para leer los nombres de todos los ficheros que contiene. Para poder leer un directorio es necesario abrirlo antes, de forma análoga a como se abre y se lee un fichero.

4. **Closedir**. Una vez que se ha terminado de leer un directorio, debe cerrarse para desocupar espacio en las tablas internas.

5. **Readdir**. Esta llamada devuelve la siguiente entrada de un directorio abierto. Antes era posible leer directorios utilizando la llamada al sistema read normal, pero eso tenía la desventaja de que obligaba al programador a conocer y tener en cuenta la estructura interna de los directorios. En cambio, readdir siempre devuelve una entrada en el formato estándar, sin importar cuál de las posibles estructuras de directorio se estén utilizando.

6. **Rename**. En muchos sentidos, los directorios son como ficheros y se les puede cambiar el nombre igual que a los ficheros.

7. **Link**. El enlazado es una técnica que permite a un fichero aparecer en más de un directorio. Esta llamada al sistema especifica un fichero existente y un nombre de camino, y crea un enlace entre el fichero existente y el nombre especificado por el camino. Así el mismo fichero podría aparecer en múltiples directorios. Un enlace de este tipo, que incrementa el contador en el i-nodo del fichero (para llevar la cuenta

del número de entradas de directorio que contienen al fichero), se conoce como **enlace duro**.

8. **Unlink**. Se elimina una entrada de directorio. Si el fichero que se está desenlazando sólo está presente en un directorio (que es lo más común), se elimina del sistema de ficheros. Si está presente en varios directorios, sólo se elimina el nombre de camino especificado; los demás permanecerán. En UNIX, la llamada al sistema para borrar ficheros (que vimos antes) en realidad es unlink.

La lista anterior incluye las llamadas más importantes, pero hay unas pocas más, como las que administran la información de protección asociada con un directorio.

6.3 IMPLEMENTACIÓN DEL SISTEMA DE FICHEROS

Ha llegado el momento de pasar a la perspectiva que tiene el usuario acerca del sistema de ficheros, a la perspectiva del implementador. A los usuarios les preocupa qué nombres tienen los ficheros, qué operaciones pueden ejecutarse con ellos, qué aspecto tiene el árbol de directorios, y cuestiones de interfaz similares. A los implementadores les preocupa la forma en que se almacenan los ficheros y directorios, cómo se administra el espacio en disco y cómo puede hacerse que todo funcione de manera eficiente y confiable. En las secciones que siguen examinaremos varias de esas áreas para ver qué problemas y sacrificios implican.

6.3.1 Organización del sistema de ficheros

Los sistemas de ficheros se almacenan en discos. Casi todos los discos pueden dividirse en una o más particiones, con sistemas de ficheros independientes en cada partición. El sector 0 del disco se llama **registro maestro de arranque** (**MBR**; *Master Boot Record*) y sirve para arrancar el ordenador. El final del MBR contiene la tabla de particiones. Esta tabla contiene las direcciones inicial y final de cada partición. Una de las particiones de la tabla está marcada como activa. Cuando se enciende el ordenador, el BIOS lee el MBR del disco y lo ejecuta. Lo primero que hace el programa del MBR es localizar la partición activa, leer su primer bloque, llamado **bloque de arranque**, y ejecutarlo. El programa del bloque de arranque carga el sistema operativo contenido en esa partición. Por uniformidad, cada partición, comienza con un bloque de arranque aun que no contenga un sistema operativo arrancable. De cualquier modo, ese bloque podría contener uno en el futuro, por lo que es una buena idea reservarlo.

Aparte de comenzar con un bloque de arranque, la organización de una partición del disco varía de forma considerable de un sistema de ficheros a otro. Es común que el sistema de ficheros contenga algunos de los elementos que se muestran en la Figura 6-11. El primero es el **superbloque**, que contiene todos los parámetros clave acerca del sistema de ficheros y se transfiere del disco a la memoria cuando se arranca el ordenador o cuando se toca por primera vez el sistema de ficheros. La información que suele contener un superbloque incluye un número mágico para identificar el tipo de sistema de ficheros, el número de bloques que hay en el sistema de ficheros y otra información administrativa crucial.

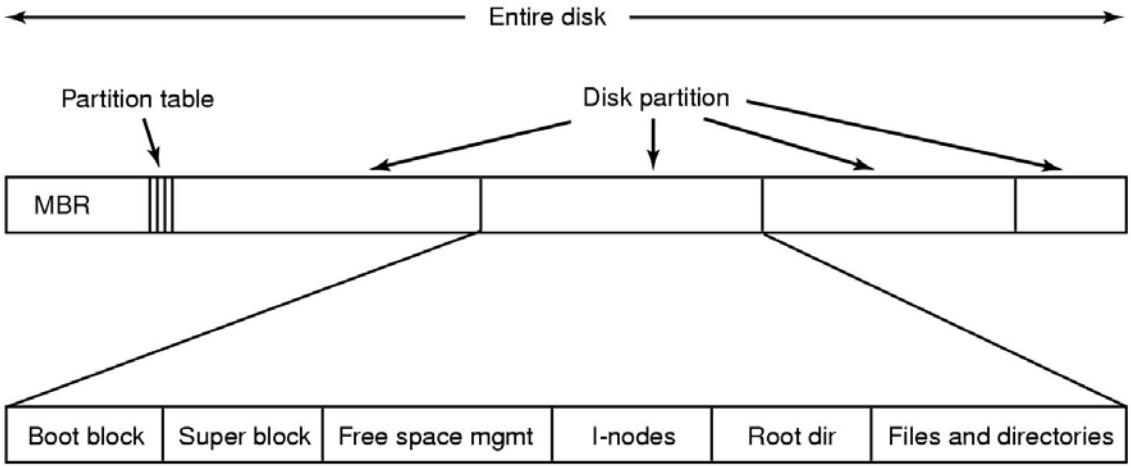

Figura 6-11. Una posible organización del sistema de ficheros.

A continuación podría haber información acerca de bloques libres en el sistema de ficheros, por ejemplo, en forma de mapa de bits o de una lista de punteros. Luego podrían estar los i-nodos, un array de estructuras, una por fichero, que proporciona todas las características del fichero. Después podría venir el directorio raíz, que contiene la parte más alta del árbol del sistema de ficheros. Por último, el resto del disco suele contener los demás directorios y ficheros.

6.3.2 Implementación de los ficheros

Tal vez el aspecto más importante de la implementación del almacenamiento de los ficheros sea llevar el control de qué bloques de disco corresponden a qué fichero. Se emplean diversos métodos en los distintos sistemas operativos. En esta sección examinaremos algunos de ellos.

Asignación contigua

El esquema de asignación más simple es almacenar cada fichero en una serie contigua de bloques de disco. Así en un disco con bloques de 1 KB, se asignarían 50 bloques consecutivos a un fichero de 50 KB. Si los bloques fueran de 2 KB, se le asignarían 25 bloques consecutivos .

Vemos un ejemplo de asignación de almacenamiento contiguo en la Figura 6-12(a). Ahí se muestran los primeros 40 bloques de disco, comenzando con el bloque 0 a la izquierda . En un principio, el disco estaba vacío. Luego se escribió en el disco un fichero *A* con una longitud de cuatro bloques a partir del principio (bloque 0). Después se escribió un fichero de seis bloques, *B*, inmediatamente después de fichero *A*. Cabe señalar que cada fichero comienza al principio de un bloque nuevo, de modo que si el fichero *A* en realidad ocupara 3,5 bloques, se desperdiciaría algo de espacio al final del último bloque. En la figura se muestra un total de siete ficheros, cada uno comenzando en el bloque que sigue al último bloque del fichero anterior. Se utiliza sombreado para que sea más fácil distinguir los ficheros.

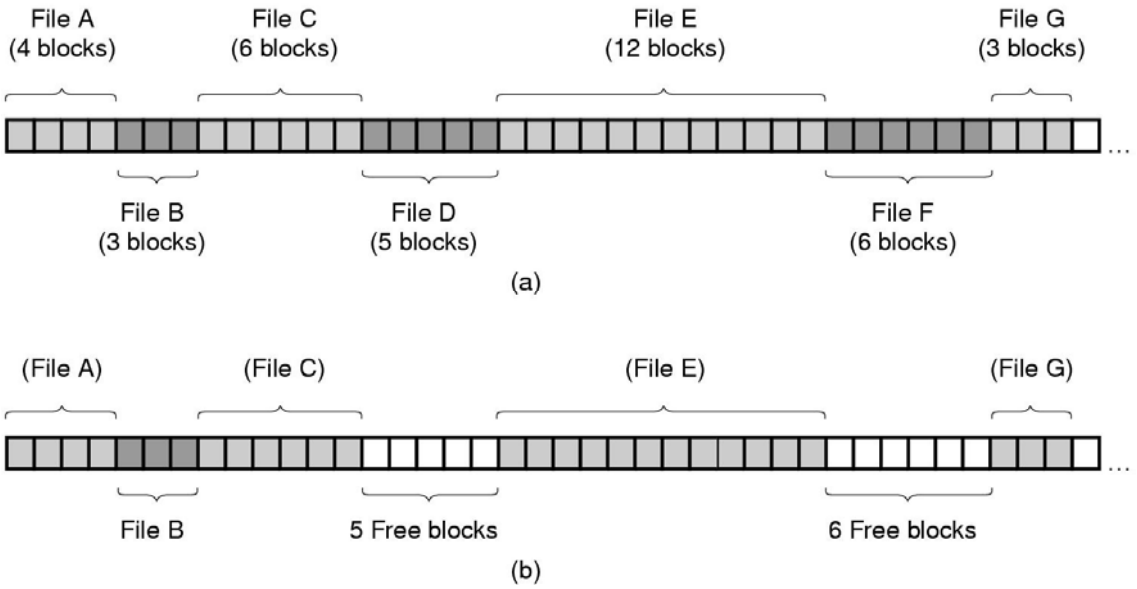

Figura 6-12. (a) Asignación contigua de espacio en disco para siete ficheros. (b) El estado del disco después de borrar los ficheros D y F.

La asignación de espacio contiguo en disco tiene dos ventajas importantes. La primera es que su implementación es sencilla porque para llevar el control de dónde están los bloques de un fichero basta con recordar dos números: la dirección en disco del primer bloque y el número de bloques del fichero. Dado el número del primer bloque, se podrá hallar el número de cualquier otro bloque mediante una simple suma.

La segunda ventaja es que la eficiencia en lectura es excelente porque puede leerse todo el fichero del disco en una sola operación. Sólo se necesita un desplazamiento del brazo (al primer bloque). Después no se requiere más desplazamientos ni retrasos rotacionales, y los datos se transfieren con el ancho de banda máximo que permite el disco. Así la asignación contigua es fácil de implementar y tiene un alto rendimiento.

Por desgracia, la asignación contigua también tiene una importante desventaja: con el tiempo, el disco se fragmenta. Para ver cómo sucede este, examinaremos la Figura 6-12(b). Aquí se han borrado dos ficheros *C* y *F*. Cuando se elimina un fichero, sus bloques se liberan, dejando una serie de bloques libres en el disco. El disco no se compacta de inmediato para tener un único hueco grande, pues eso requeriría copiar todos los bloques que están después del hueco, de los cuales podría haber millones. El resultado es que al final el disco se compone de ficheros y huecos, como se ilustra en la figura.

En un principio, esta fragmentación no es un problema porque es posible escribir cada fichero nuevo al final del disco, después del anterior. Sin embargo, tarde o temprano el disco se llenará y será necesario compactarlo, lo cual tiene un coste prohibitivo, o reutilizar el espacio desocupado (los huecos). Para ello es necesario mantener una lista de huecos, lo cual es factible. Sin embargo, cuando se va a crear un fichero nuevo se hace necesario conocer su tamaño final para escoger un hueco del tamaño correcto en el cual colocarlo.

Imaginemos las consecuencias de tal diseño. El usuario inicial un editor de texto o procesador de texto para escribir un documento. Lo primero que pregunta el programa es cuántos bytes va a tener el fichero final, negándose a continuar si no se contesta a esa pregunta. Si el número dado al final resulta demasiado pequeño, el programa tendrá que terminar de forma prematura porque el hueco en el disco está lleno y no hay lugar para colocar el resto del fichero.

Si el usuario trata de evitar este problema dando como tamaño final una cifra muy grande, poco realista, digamos 100 MB, el editor podría ser incapaz de hallar un hueco tan grande, en cuyo caso anunciaría que no puede crearse el fichero. Desde luego, el usuario podría reiniciar el programa y contestar 50 MB, y seguir así hasta hallar un hueco apropiado. De cualquier manera, es poco probable que este esquema satisfaga a los usuarios.

No obstante, existe una situación en la que la asignación contigua es factible y, de hecho, muy utilizada: en los CD-ROMs. Aquí se conoce con antelación el tamaño de todos los ficheros, y esos tamaños no cambiarán durante el uso posterior del sistema de ficheros del CD-ROM. En una sección posterior del capítulo estudiaremos el sistema de ficheros más común utilizado en los CD-ROMs.

Como mencionamos en el capítulo 1, la historia a menudo se repite en el campo de la informática, conforme surgen nuevas generaciones de tecnología. La asignación contigua se utilizó en los sistemas de ficheros de disco magnético hace años debido a su sencillez y su gran rapidez (la amabilidad para con el usuario no contaba mucho entonces). Luego se desechó la idea por la molestia de tener que especificar el tamaño final de los ficheros en el momento de crearlos. Sin embargo, con la llegada de los CD-ROMs, DVDs. y otros medios ópticos en los que se escribe una sola vez, de repente los ficheros contiguos vuelven a ser una buena idea. Por ello, es importante estudiar los sistemas e ideas antiguos que eran claros y sencillos desde el punto de vista conceptual, por que podrían ser aplicables a sistemas futuros de formas sorprendentes.

Asignación por lista enlazada

El segundo método para almacenar ficheros consiste en mantener cada uno como una lista enlazada de bloques de disco, como se muestra en la Figura 6-13. La primera palabra de cada bloque se utiliza como puntero al siguiente bloque del fichero. El resto del bloque es para datos.

A diferencia de la asignación contigua, con este método pueden utilizarse todos los bloques del disco. No se pierde espacio por fragmentación del disco (sólo por fragmentación interna en el último bloque). Además basta que la entrada del directorio correspondiente al fichero guarde la dirección de disco del primer bloque. El resto de bloques pueden localizarse a partir de ese punto.

Por otra parte, aunque la lectura secuencial de un fichero es directa, el acceso aleatorio es lento en el extremo. Para llegar al bloque n, el sistema operativo tienen que comenzar por el principio y leer los $n - 1$ bloques que lo preceden, uno por uno. Es evidente que tantas lecturas hacen demasiado lento el acceso.

Además, la cantidad de datos almacenados en un bloque ya no es una potencia de 2 porque el puntero ocupa unos cuantos bytes. Aunque no es fatal tener un tamaño peculiar, merma la eficiencia porque muchos programas leer y escriben en bloques cuyo tamaño es una potencia de 2. Si los primeros bytes de cada bloque están ocupados por un puntero al siguiente bloque, las lecturas de bloques enteros requieren obtener y concatenar información de dos bloques de disco, lo cual genera un gasto adicional debido a la copia.

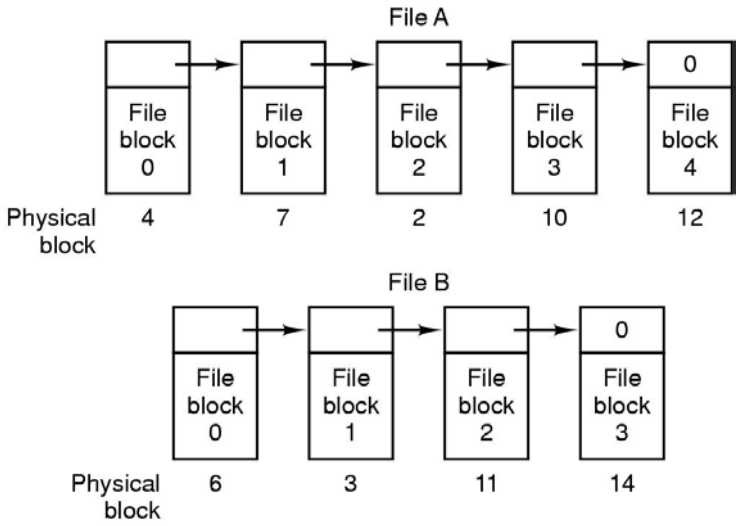

Figura 6-13. Almacenamiento de un fichero como una lista enlazada de bloques del disco.

Las dos desventajas de la asignación por lista enlazada pueden eliminarse sacando el puntero de cada bloque del disco y colocándolo en una tabla en la memoria. La Figura 6-14 muestra cómo se vería la tabla para el ejemplo de la Figura 6-13. En ambas figuras tenemos dos ficheros. El fichero *A* ocupa los bloques de disco 4, 7, 2, 10 y 12, en ese orden, y el fichero *B* ocupa los bloques 6, 3, 11 y 14, en ese orden. Con la tabla de la Figura 6-14, podemos partir del bloque 4 y seguir la cadena hasta el final. Lo mismo puede hacerse partiendo del bloque 6. Ambas cadenas terminan con un marcador especial (por ejemplo –1) que no es un número de bloque válido. Una tabla así en la memoria principal se denomina una **FAT** (**File Allocation Table**; tabla de asignación de ficheros).

Con esta organización los bloques pueden llenarse ahora completamente con datos. Además el acceso aleatorio es mucho más fácil. Aunque todavía es necesario seguir la cadena para hallar un desplazamiento dado dentro del fichero, la cadena está por completo en la memoria, así que puede seguirse sin tener que leer el disco. Al igual que con el método anterior, basta con que la entrada del directorio guarde un único entero (el número del primer bloque) para poder localizar todos los bloques, sin importar qué tamaño tenga el fichero.

La desventaja primordial de este método es que, para que funcione, toda la tabla debe estar en la memoria todo el tiempo. Con un disco de 20 GB y bloques de 1 KB, la tabla necesita 20 millones de entradas, una para cada uno de los 20 millones de bloques del disco. Cada entrada debe tener un mínimo de 3 bytes, y si se desea agilizar la consulta se necesitan 4 bytes. Por tanto, la tabla ocupará 60 u 80 MB de memoria principal todo el tiempo, dependiendo de si el sistema está optimizado desde el punto de vista del espacio o del tiempo. Es concebible colocar la tabla en memoria paginable, pero de todos modos ocuparía una gran cantidad de memoria virtual y de espacio en disco, además de generar tráfico de paginación adicional.

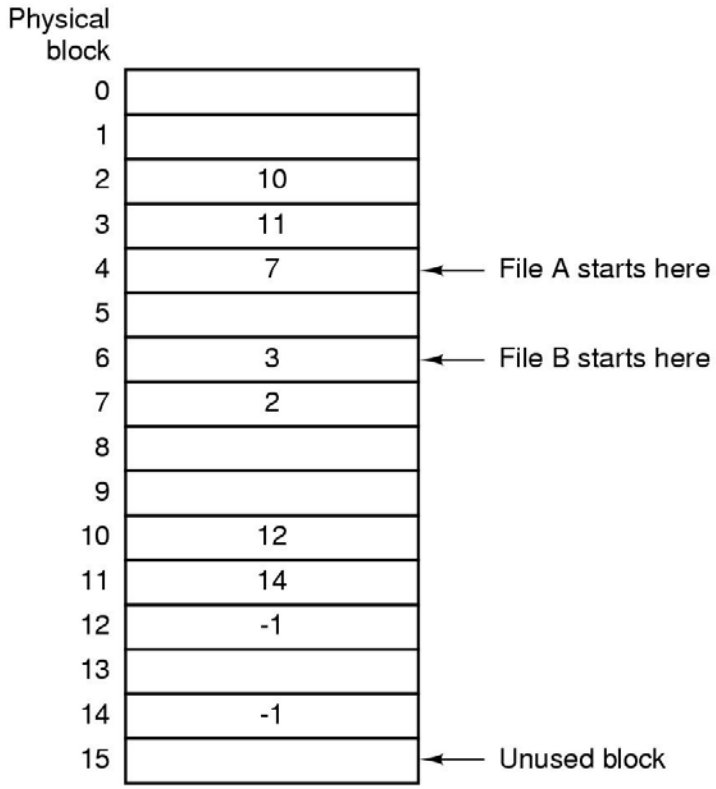

Figura 6-14. Asignación enlazada empleando una tabla de asignación de ficheros en la memoria principal.

i-nodos

Nuestro último método para llevar el control de qué bloques pertenecen a qué ficheros consiste en asociar a cada fichero una estructura de datos llamada **i-nodo** (**nodo de índice**), que contiene los atributos y direcciones en disco de los bloques del fichero. En la Figura 6-15 se muestra un ejemplo sencillo. Dado el i-nodo, es posible hallar todos los bloques del fichero. La gran ventaja de este esquema respecto de las listas enlazadas empleando una tabla en la memoria es que el i-nodo sólo tiene que estar en memoria cuando el fichero correspondiente está abierto. Si cada i-nodo ocupa *n* bytes y no puede haber más de *k* ficheros abiertos al mismo tiempo, la memoria total ocupada por el array que contiene los i-nodos de los ficheros abiertos es de sólo *kn* bytes. Únicamente es necesario reservar esa cantidad de espacio.

Este array suele ser mucho más pequeño que el espacio ocupado por la tabla de ficheros que describimos en la sección anterior. La razón es sencilla. La tabla para contener la lista enlazada de todos los bloques del disco tiene un tamaño proporcional al disco mismo. Si el disco tiene *n* bloques, la tabla necesita *n* entradas. A medida que aumenta el tamaño de los discos, el tamaño de esta tabla crece en proporción lineal. En contraste, el esquema de i-nodos requiere un array en la memoria cuyo tamaño sea proporcional al número máximo de ficheros que pueden estar abiertos a la vez. No importa si el disco es de 1, de 10 o de 100 GB.

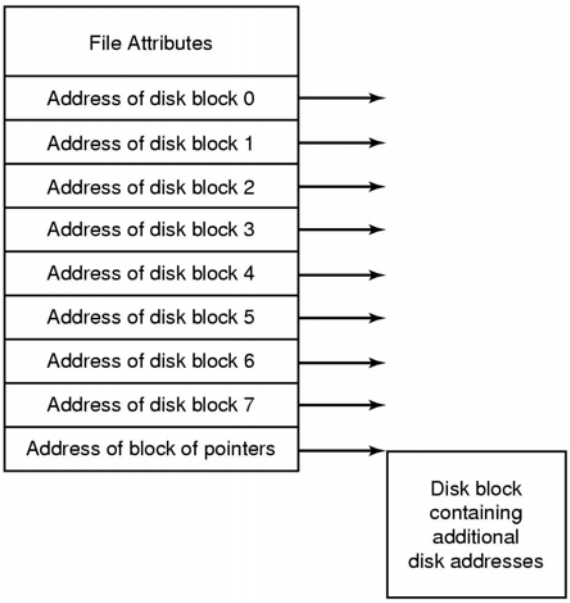

Figura 6-15. Ejemplo de i-nodo.

Un problema de los i-nodos es que si cada uno tiene espacio para un número fijo de direcciones de disco, ¿qué sucede cuando un fichero crece más allá de ese límite? Una solución es reservar la última dirección de disco no para un bloque de datos, sino para la dirección de un bloque que contenga más direcciones de bloques de disco, como se muestra en la Figura 6-15. Algo aún más avanzado sería tener dos o más de esos bloques llenos de direcciones en disco o incluso bloques de disco que apunten a otros bloque de disco llenos de direcciones. Volveremos a ver los i-nodos cuando estudiemos UNIX más adelante.

6.3.3 Implementación de directorios

Para poder leer de un fichero, es preciso abrirlo primero. Cuando se abre un fichero, el sistema operativo utiliza el nombre de camino proporcionado por el usuario para localizar la entrada de directorio. Ésta proporciona la información necesaria para hallar los bloques de disco. Dependiendo del sistema, esta información podría ser la dirección en disco de todo el fichero (asignación contigua), el número del primer bloque (ambos esquemas de lista enlazada) o el número del i-nodo. En todos los casos. La función principal del sistema de directorios es establecer una correspondencia entre el nombre de fichero ASCII y la información necesaria para localizar los datos.

Un aspecto estrechamente relacionado es dónde deben guardarse los atributos. Todo sistema de ficheros mantiene atributos de los ficheros, como su propietario y tiempo de creación, y deben almacenarse en algún lado. Una posibilidad obvia es guardarlos directamente en la entrada de directorio. Muchos sistemas hacen precisamente esto. En la Figura 6-16(a) se muestra esta opción. En este sencillo diseño, un directorio es una lista de entradas de tamaño fijo, una por fichero, que contiene un nombre de fichero (de longitud fija), una estructura con los atributos del fichero y una o más direcciones en disco (hasta algún máximo) que indican dónde están los bloques de disco.

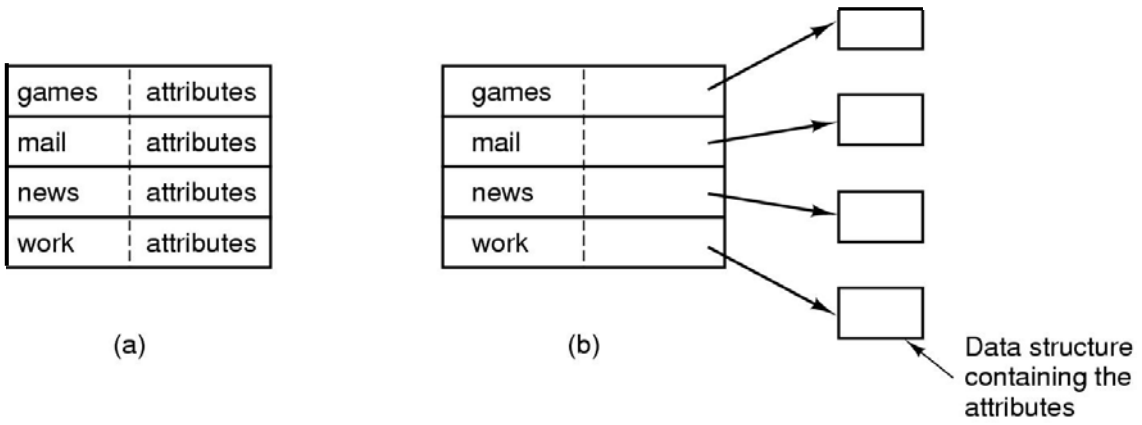

Figura 6-16. (a) Directorio sencillo que contiene entradas de tamaño fijo con las direcciones en disco y los atributos de cada fichero. (b) Directorio en el que cada entrada sólo hace referencia a un i-nodo.

En los sistemas que utilizan i-nodos, otra posibilidad para almacenar los atributos es en los i-nodos, en lugar de en las entradas de directorio. En este caso, la entrada de directorio puede ser más corta: tan sólo un nombre de fichero y un número de i-nodo. Este enfoque se ilustra en la Figura 6-16(b). Como veremos más adelante, este método tiene ciertas ventajas respecto a colocar los atributos en la entrada del directorio. Los dos enfoques que se muestran en la Figura 6-16 corresponden a MS-DOS/Windows y UNIX, respectivamente, como veremos en una sección posterior del capítulo.

Hasta ahora hemos supuesto que los ficheros tienen nombres cortos de longitud fija. En MS-DOS los ficheros tienen un nombre base de uno a ocho caracteres y una extensión opcional de uno a tres caracteres. En UNIX versión 7, los nombres de fichero tenían de uno a 14 caracteres, incluyendo cualquier extensión. Sin embargo casi todos los sistemas operativos modernos reconocen nombres de fichero más largos, de longitud variable. ¿Cómo pueden implementarse?

El método más sencillo es fijar un límite para la longitud del nombre de fichero, por lo regular 255 caracteres, y luego utilizar uno de los diseños de la Figura 6-16 con 255 caracteres reservados para cada nombre de fichero. Este método es sencillo, pero desperdicia mucho espacio de directorio, porque pocos ficheros tienen nombres tan largos. Por razones de eficiencia, conviene utilizar una estructura diferente.

Una alternativa es abandonar la idea de que todas las entradas de directorio tienen el mismo tamaño. Con este método, cada entrada de directorio contiene una porción fija, que por lo regular comienza con la longitud de la entrada, seguida de datos con un formato fijo, que normalmente incluyen al propietario, la hora en que se creó, información de protección y otros atributos. Este encabezado de longitud fija va seguido del nombre del fichero en sí, que puede tener cualquier longitud, como se muestra en la Figura 6-17(a) en formato big-endian (por ejemplo SPARC). En este ejemplo tenemos tres ficheros, *project-budget*, *personnel* y *foo*. Cada nombre de fichero termina con un carácter especial (normalmente el carácter que tiene número ASCII 0) que se representa en la figura con una cruz encerrada en un cuadrado. Para que cada entrada de directorio pueda comenzar en una frontera de palabra, cada nombre de fichero se rellena hasta un número entero de palabras, lo cual se indica con rectángulos sombreados en la figura.

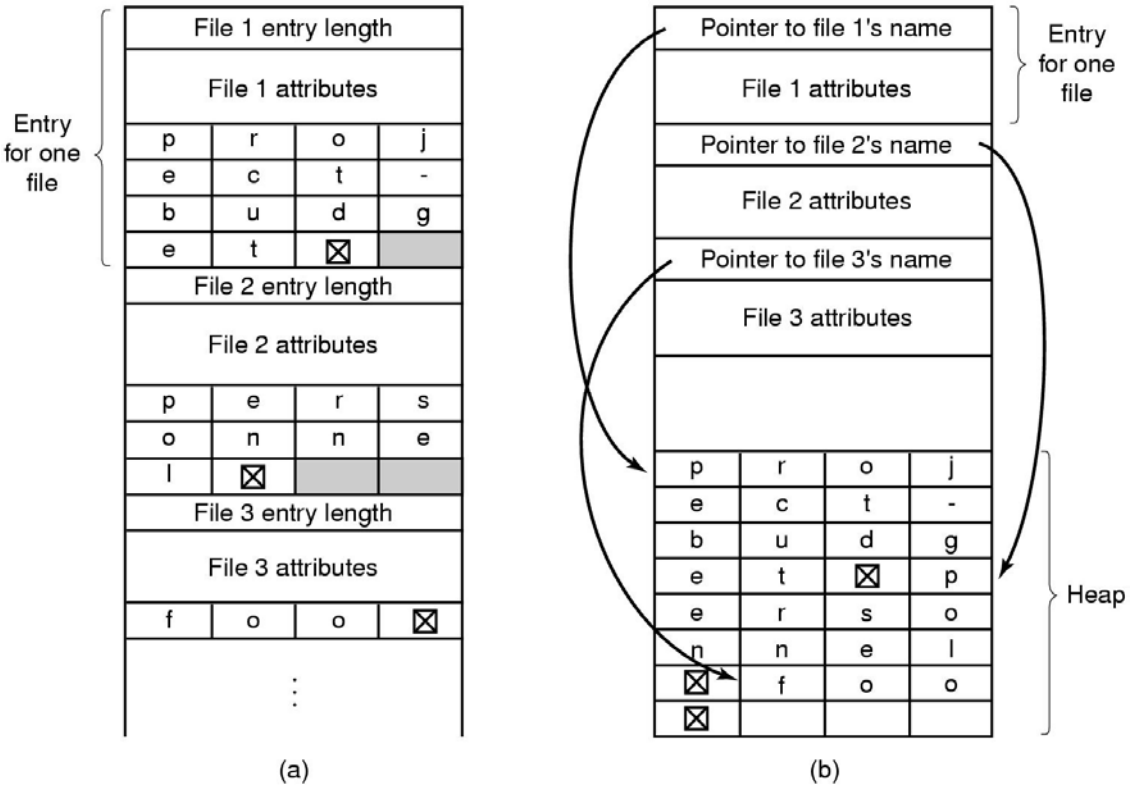

Figura 6-17. Dos formas de manejar nombres de fichero largos en un directorio. (a) Incorporados. (b) en un heap.

Una desventaja de este método es que cuando se elimina un fichero queda en el directorio un hueco de tamaño variable en el que tal vez no quepa la entrada del siguiente fichero que se cree. Este problema es el mismo que vimos con los ficheros contiguos en disco, sólo que ahora sí es factible compactar el directorio porque está por completo en la memoria. Otro problema es que una sola entrada de directorio podría cruzar fronteras de página, por lo que podría presentarse una falta de página durante la lectura de un nombre de fichero.

Otra forma de manejar los nombres de longitud variable es hacer que todas las entradas de directorio propiamente dichas sean de longitud fija y mantener los nombres de fichero juntos en un heap al final del directorio, como se muestra en la Figura 6-17(b). Este método tienen la ventaja de que cuando se elimina una entrada siempre cabrá ahí la del siguiente fichero que se cree. Desde luego, hay que administrar el heap, y sigue existiendo la posibilidad de que se presenten fallos de página durante el procesamiento de nombres de fichero. Una ventaja menor aquí es que ya no es necesario que los nombres de fichero comiencen en fronteras de palabra, de modo que no se requieren caracteres de relleno en la Figura 6-17(b), como sucedía en la Figura 6-17(a).

En todos los diseños que hemos visto hasta ahora, los directorios se exploran mediante una búsqueda lineal, desde el principio hasta el final, cada vez que se busca un nombre de fichero. En el caso de directorios extremadamente largos, las búsquedas lineales pueden ser lentas. Una forma de acelerar la búsqueda es utilizar una tabla hash en cada directorio. Digamos que el tamaño de la tabla es n. Para introducir un nombre de fichero, el nombre se transforma en un valor entre 0 y $n - 1$, por ejemplo, dividiéndolo entre n y utilizando el resto. O bien, las palabras que integran el nombre del fichero pueden sumarse y dividir la cantidad obtenida entre n, o algo similar.

De cualquier modo, se examina la entrada de la tabla correspondiente al código hash. Si está desocupada, se coloca en ella un puntero a la entrada del fichero. Las entradas con los atributos de los ficheros se encuentran después de la tabla hash. Si ya se está utilizando esa entrada en la tabla hash, se construye una lista enlazada, poniendo como primer elemento esa entrada de la tabla, que encadena todas las entradas que tienen el mismo valor hash.

La consulta de un fichero sigue el mismo procedimiento. El nombre del fichero se transforma en un índice para seleccionar una entrada de la tabla hash. Se examinan todas las entradas de la cadena cuya primer elemento está en esa entrada, para ver si está presente el nombre del fichero. Si el nombre no está en la cadena, quiere decir que el fichero no está en el directorio.

La utilización de una tabla hash tiene la ventaja de que las búsquedas son mucho más rápidas, pero tiene la desventaja de que la administración es más complicada. En realidad sólo es lógico utilizar una tabla hash en los sistemas en los que se espera que los directorios normalmente contengan cientos o miles de ficheros.

Una forma completamente distinta de acelerar las búsquedas en directorios grandes es poner en una caché los resultados de la búsqueda. Antes de iniciar una búsqueda, se verifica si el nombre de fichero está en la caché. Si está, podrá localizarse rápido, evitando así una búsqueda larga. Claro que la utilización de caché sólo resulta útil si el número de ficheros que se buscan más es relativamente reducido.

6.3.4 Ficheros compartidos

Cuando varios usuarios colaboran en un proyecto, es normal que necesiten compartir ficheros. Por ello, en muchos casos conviene que un fichero compartido aparezca al mismo tiempo en diferentes directorios que pertenecen a usuarios distintos. La Figura 6-18 muestra otra vez el sistema de ficheros de la Figura 6-9, sólo que ahora uno de los ficheros de *C* también está presente en uno de los directorios de *B*. La conexión entre el directorio de *B* y el fichero compartido se denomina un **enlace** (*link*). El sistema de ficheros en sí es ahora un **DAG** (*Direct Acyclic Graph*; grafo acíclico dirigido), no un árbol.

Compartir ficheros es conveniente, pero también presenta problemas. Para empezar, si los directorios contienen de verdad direcciones de disco, tendrá que crearse una copia de ellas en el directorio de *B* cuando se enlace el fichero. Si después *B* o *C* hacen crecer al fichero, los nuevos bloques aparecerán sólo en el directorio del usuario que modificó el fichero. Los cambios no serán visibles para el otro usuario, frustrando así el propósito de compartir los ficheros.

Hay dos formas de resolver este problema. En la primera, los bloques de disco no se listan en los directorios, sino en una pequeña estructura de datos asociada con el fichero mismo. Los directorios apuntarían entonces sólo a la pequeña estructura de datos. Éste el es método que se utiliza en UNIX (donde la pequeña estructura de datos es el i-nodo).

En la segunda solución, *B* se enlaza con uno de los ficheros de *C* pidiendo al sistema que cree un fichero nuevo, de tipo LINK, y lo introduzca en el directorio de *B*. El nuevo fichero sólo contiene el nombre de camino del fichero con el cual está enlazado. Cuando *B* lee del fichero enlazado, el sistema operativo percibe que el fichero que se está leyendo es de tipo LINK, busca el nombre del fichero y lo lee. Este método se llama **enlace simbólico**.

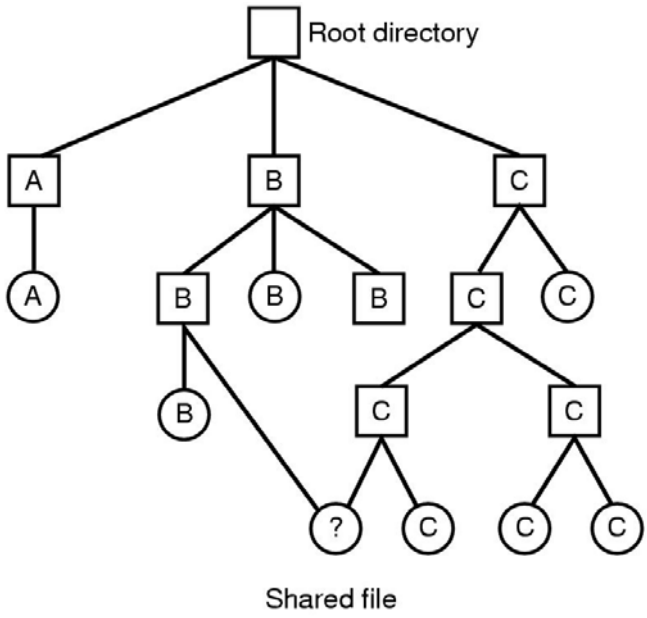

Figura 6-18. Sistema de ficheros que contiene un fichero compartido.

Todos estos métodos tienen desventajas. Con el primero, en el momento en que *B* se enlaza con el fichero compartido, el i-nodo indica que el propietario del fichero es *C*. La creación de un enlace no hace que el fichero cambie de dueño (vea la Figura 6-19), pero sí incrementa el contador de enlaces en el i-nodo para que el sistema sepa cuantas entradas de directorio están apuntando en la actualidad al fichero.

Si más adelante *C* trata de eliminar el fichero, el sistema se enfrenta a un problema. Si elimina el fichero y libera el i-nodo, *B* tendrá una entrada de directorio que apunta a un i-nodo que no es válido. Si después el i-nodo se reasigna a otro fichero, el enlace de *B* apuntará al fichero equivocado. El sistema puede ver, por la cuenta contenida en el i-nodo, que el fichero todavía está en uso, pero no tiene forma de hallar todas las entradas de directorio para ese fichero a fin de borrarlas. No es posible almacenar punteros a los directorios en el i-nodo porque podría haber un número ilimitado de directorios.

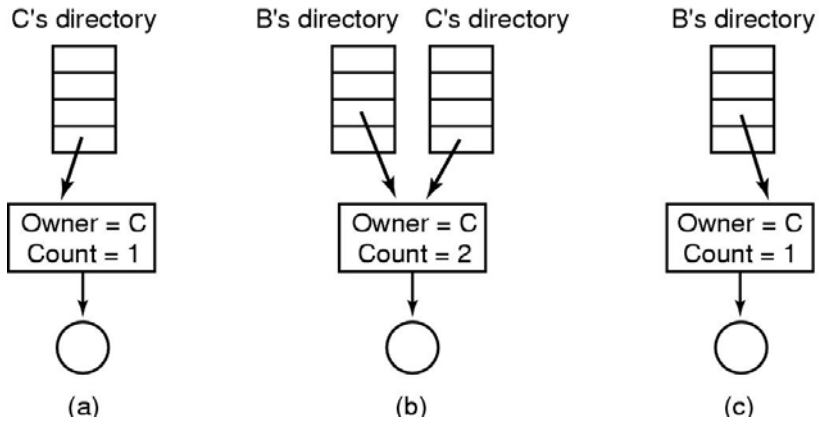

Figura 6-19. (a) La situación antes del enlace. (b) Una vez que se ha creado el enlace. (c) Después de que el propietario original elimina el fichero.

Lo único que puede hacerse es eliminar la entrada de directorio de *C*, pero dejar intacto el i-nodo, con su contador a 1, como se muestra en la Figura 6-19(c). Ahora tenemos una

situación en la que *B* es el único usuario que tiene una entrada de directorio para un fichero propiedad de *C*. Si el sistema realiza contabilidad o tiene cuotas, se seguirá cobrando a *C* por el fichero hasta que B decida borrarlo, si alguna vez lo hace, pues en ese momento el contador se hará cero y el fichero se borrará.

Con enlaces simbólicos no se presenta este problema porque sólo el verdadero propietario tiene un puntero al i-nodo. Los usuarios que se han enlazado con el fichero sólo tienen nombres de camino, no punteros a i-nodos. Cuando el propietario elimina el fichero, éste se destruye. Los intentos subsiguientes de utilizar el fichero por medio de un enlace simbólico fracasarán cuando el sistema no pueda localizar el fichero. La eliminación de un enlace simbólico no afecta en absoluto al fichero.

El problema con los enlaces simbólicos es el procesamiento adicional requerido. Es preciso leer el fichero que contiene el camino, el cual debe entonces analizarse y seguirse, componente por componente, hasta llegar al i-nodo. Toda esta actividad podría requerir una cantidad considerable de accesos adicionales al disco. Además, se necesita un i-nodo adicional por cada enlace simbólico, y también un bloque de disco adicional para almacenar el camino, aunque si el nombre de camino es corto, el sistema podría almacenarlo en el mismo i-nodo, como optimización. Los enlaces simbólicos tienen la ventaja de que pueden servir para enlazarse con ficheros de máquinas que puede estar en cualquier lugar del mundo, con sólo proporcionar la dirección de red de la máquina en la que reside el fichero, además de su camino en esa máquina.

Hay otro problema que provocan los enlaces, simbólicos o no. Si se permiten enlaces, los ficheros pueden tener dos o más caminos. Los programas que en el arranque se sitúan en un directorio dado y buscan todos los ficheros en ese directorio y sus subdirectorios, localizarán un fichero enlazado múltiples veces. Por ejemplo, un programa de backup en cinta de todos los ficheros de un directorio y sus subdirectorios podría crear varias copias de un fichero enlazado. Además, si la cinta se copia después en otra máquina, a menos que el programa de backup sea inteligente, el fichero enlazado se copiará dos veces en el disco, en lugar de estar enlazado.

6.3.5 Administración del espacio en disco

Los ficheros generalmente se almacenan en disco, por lo que la administración del espacio en disco es de primordial interés para los diseñadores de sistemas de ficheros. Pueden adoptarse dos estrategias generales para almacenar un fichero de *n* bytes: asignar *n* bytes consecutivos de espacio en disco, o dividir el fichero en varios bloques (no necesariamente) contiguos. Las ventajas y desventajas son las mismas que en los sistemas de administración de memoria, entre segmentación pura y paginación.

Como hemos visto, almacenar un fichero como una sucesión contigua de bytes tiene el problema obvio de que, si un fichero crece, es probable que tenga que pasarse a otro lugar del disco. El mismo problema se tiene con los segmentos en la memoria, sólo que cambiar de lugar un segmento en la memoria es una operación relativamente rápida, en comparación con cambiar un fichero de una posición a otra en el disco. Por ello, casi todos los sistemas de ficheros dividen los ficheros en bloques de tamaño fijo que no tienen que ser adyacentes.

Tamaño de bloque

Una vez que se ha decidido almacenar ficheros en bloques de tamaño fijo, surge la pregunta de qué tamaño debe de tener un bloque. Dada la forma en que está organizados los discos, el sector, la pista y el cilindro son candidatos obvios para ser la unidad de asignación (aunque todos estos tamaños dependen del dispositivo, lo cual es una desventaja). En un sistema con paginación, el tamaño de página también es un contendiente importante.

Tener una unidad de asignación grande, digamos un cilindro, implica que todos los ficheros, incluso aunque sólo tengan un byte, ocuparán cilindros enteros. Estudios efectuados (Mullender y Tanenbaum, 1984) han mostrado que la mediana del tamaño de los ficheros en los entornos UNIX es de aproximadamente 1 KB, así que asignar un bloque de 32 KB a cada fichero desperdiciaría 31/32 = 97% del espacio total en el disco.

Por otra parte, el uso de una unidad de asignación pequeña implica que cada fichero va a constar de varios bloques. Leer cada bloque requiere por lo regular un desplazamiento del brazo y una latencia rotacional, por lo que resulta lenta la lectura de un fichero integrado por muchos bloques pequeños.

Por ejemplo, consideremos un disco que tiene 131072 bytes por pista, un tiempo de rotación de 8,33 ms y un tiempo medio de desplazamiento del brazo de 10 ms. El tiempo requerido en milisegundos para leer un bloque de *k* bytes será entonces la suma de los tiempos de desplazamiento, latencia rotacional y transferencia:

$$10 + 4{,}165 + (k/131072) \times 8.33$$

La curva continua de la Figura 6-20 muestra la tasa de datos de un disco así en función del tamaño del bloque. Para calcular el aprovechamiento del espacio, necesitamos suponer algo acerca del tamaño medio de los ficheros. Una medición reciente efectuada en el departamento académico del autor, que tiene 1000 usuarios y más de un millón de ficheros UNIX en disco, da una mediana del tamaño de los ficheros de 1680 bytes, lo que implica que la mitad de los ficheros tienen menos de 1680 bytes, y la otra mitad tiene un tamaño mayor. Por cierto, la mediana es una métrica mejor que la media porque un número muy pequeño de ficheros puede influir de forma considerable en la media, pero no en la mediana. (De hecho, la media es de 10845 bytes, debido en parte a unos cuantos manuales de hardware de 100 MB que por casualidad están en línea.) Por sencillez, supongamos que todos los ficheros son de 2 KB, lo cual da pie a la curva de trazos punteados de la Figura 6-20 que describe la eficiencia espacial del disco.

Las dos curvas pueden interpretarse como sigue. El tiempo para tener acceso a un bloque está dominado por completo por el tiempo de posicionamiento del brazo y la latencia rotacional. Entonces, dado que va a costar 14 ms tener acceso a un bloque, cuantos más datos se obtengan, mejor. Por tanto la tasa de datos crece al aumentar el tamaño del bloque (hasta que las transferencias tardan tanto que el tiempo de transferencia comienza a dominar). Con bloques pequeños que son potencias de 2 y ficheros de 2 KB, no se desperdicia espacio en los bloques, en cambio, con ficheros de 2 KB y bloques de 4 KB o mayores, se desperdicia algo de espacio de disco. En realidad, pocos ficheros son múltiplos del tamaño de bloque de disco, por lo que siempre se desperdicia algún espacio en el último bloque de un fichero.

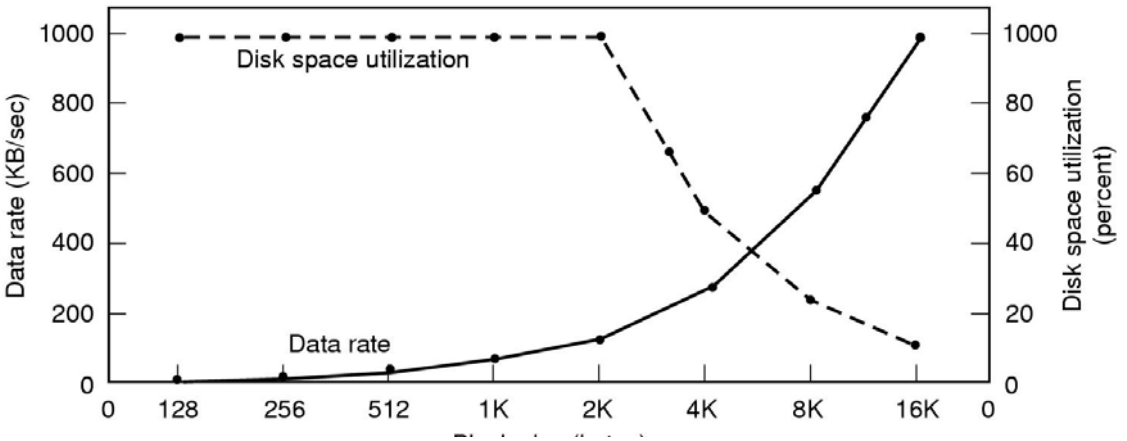

Figura 6-20. La curva (escala de la izquierda) da la tasa de datos de un disco. La curva discontinua (escala de la derecha) da la eficiencia espacial del disco. Todos los ficheros son de 2 KB.

Lo que muestran las curvas, empero, es que el rendimiento y el aprovechamiento del espacio están en conflicto de manera inherente. Los bloques pequeños son malos para el rendimiento pero buenos para el aprovechamiento del espacio en disco. Se requiere un tamaño intermedio de compromiso. Con estos datos, 4 KB podría ser una buena opción, pero algunos sistemas operativos tomaron su decisión hace mucho tiempo, cuando los parámetros de disco y los tamaños de fichero eran diferentes. En el caso de UNIX, es común utilizar 1 KB. En el caso de MS-DOS, el tamaño de bloque puede ser cualquier potencia de 2, desde 512 bytes hasta 32 KB, pero está determinado por el tamaño del disco y por factores que no tienen relación con estos argumentos. (El número máximo de bloques en una partición de disco es 2^{16}, lo que obliga a usar bloques grandes en discos grandes.)

En un experimento para ver si el uso de ficheros en Windows NT mostraba diferencias apreciables respecto al uso de ficheros en UNIX, Vogels hizo mediciones de ficheros en la Universidad de Cornell (Vogels, 1999). Observó que la utilización de los ficheros en NT es más complicada que en UNIX, y escribió:

> *Si tecleamos unos cuantos caracteres en el editor de texto notepad, al guardar esto en un fichero se generan 26 llamadas al sistema, incluyendo tres intentos fallidos de apertura, 1 sobrescritura de fichero y 4 secuencias de apertura y cierre adicionales.*

No obstante, observó una mediana (ponderada por el uso) del tamaño de los ficheros recién escritos de 2,3 KB y de ficheros leídos y escritos de 4,2 KB. Considerando el hecho de que Cornell realiza computación científica a mucha mayor escala que la institución del autor, y la diferencia en la técnica de medición (estática frente a dinámica), los resultados son razonablemente congruentes con una mediana del tamaño de fichero alrededor de 2 KB.

Control de bloques libres

Una vez escogido un tamaño de bloque, la siguiente cuestión es cómo llevar el control de los bloques libres. Se usan dos métodos principalmente, los cuales se muestran en la Figura 2-21. El primero consiste en usar una lista enlazada de bloques de disco, en cada uno de los cuales se guardan tantos números de bloques de disco como quepan. Con bloques de 1 KB y números de bloque de 32 bits, cada bloque de la lista de bloques libres contendrá los números de 255 bloques libres (se necesita una entrada para el puntero al siguiente bloque). Un disco de 16 GB necesitará una lista de bloques libres de 16794 bloques como máximo para contener los 2^{24} números de bloque. Es común que se utilicen bloques libres para almacenar la lista libre.

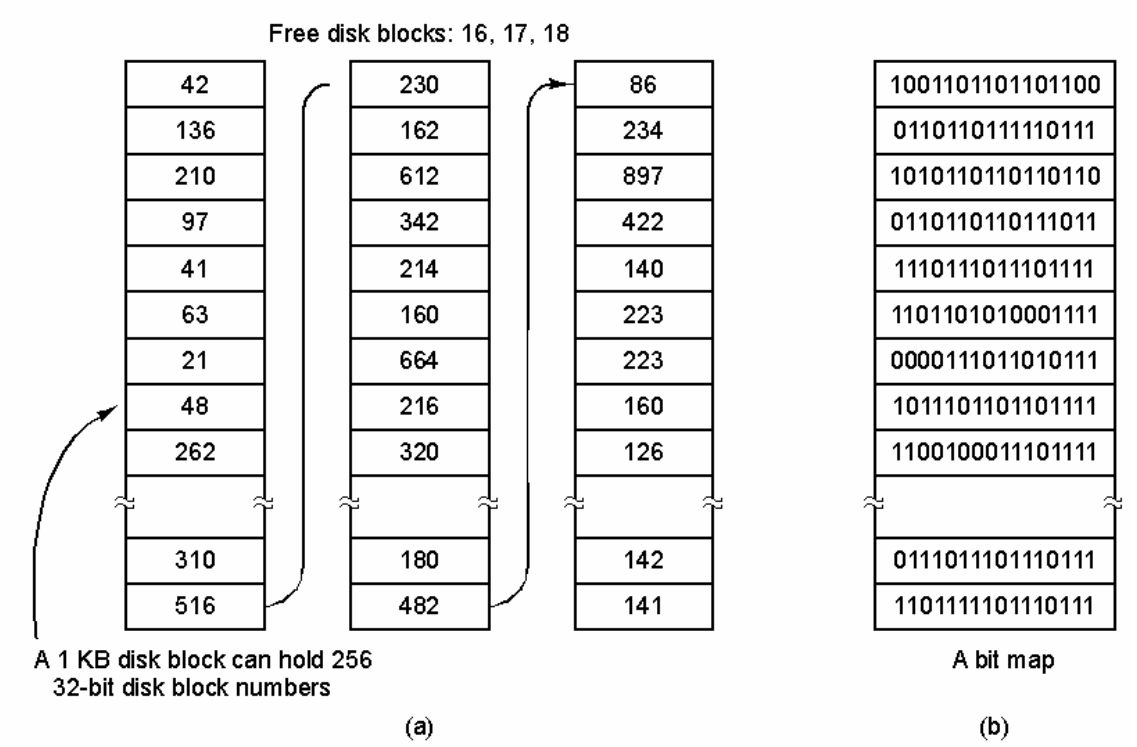

Figura 6-21. (a) Almacenamiento de la lista de bloques libres en una lista enlazada. (b) Un mapa de bits.

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 16 GB tiene 2^{24} bloques de 1KB y por tanto requiere 2^{24} bits para el mapa, lo cual ocupa 2048 bloques. No es sorprendente que el mapa de bits requiera menos espacio, puesto que utiliza 1 bit por bloque, en comparación con 32 bits si se utiliza el modelo de la lista enlazada. Sólo si el disco está lleno (es decir, si tiene pocos bloques libres) el esquema de lista enlazada requerirá menos bloques que el mapa de bits. Por otra parte, si hay muchos bloques libres, podrán pedirse prestados algunos de ellos para almacenar la lista libre, sin pérdida de capacidad del disco.

Si se usa el método de la lista de bloques libres, sólo es preciso mantener un bloque de punteros en la memoria principal. Cuando se crea un fichero, los bloques que necesita se toman del bloque de punteros. Cuando este bloque se agota, se lee del disco un nuevo bloque de punteros. De forma similar, cuando se borra un fichero, sus bloques se liberan y se añaden al bloque de punteros que está en la memoria principal. Si este bloque se llena, se escribe en el disco.

En ciertas circunstancias, este método da pie a una cantidad innecesarias de operaciones de E/S al disco. Consideremos la situación de la Figura 6-22(a), donde el bloque de punteros que está en la memoria sólo tiene espacio para dos entradas más. Si se libera un fichero de tres bloques, el bloque de punteros se desbordará y tendrá que escribirse en el disco, produciendo la situación de la Figura 6-22(b). Si ahora se escribe un fichero de tres bloques, será necesario traer otra vez a la memoria el bloque lleno de punteros, y volveremos a la situación de la Figura 6-22(a). Si el fichero de tres bloques recién escrito era un fichero temporal, cuando se liberen sus bloques se requerirá otra escritura en disco para guardar el bloque lleno de punteros. En síntesis, cuando el bloque de punteros está casi vacío, una serie de ficheros temporales efímeros puede generar mucha E/S de disco.

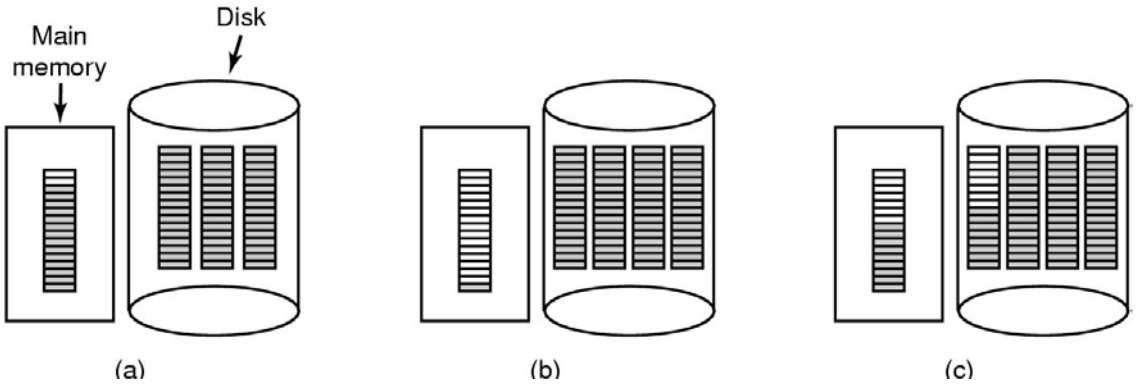

Figura 6-22. (a) Un bloque en la memoria casi lleno de punteros a bloques de disco libres y tres bloques de punteros en el disco. (b) Resultado de liberar un fichero de tres bloques. (c) Estrategia alternativa para manejar los tres bloques liberados. Las entradas sombreadas representan punteros a bloques de disco libres.

Un método alternativo que evita la mayor parte de esa E/S de disco consiste en dividir el bloque de punteros lleno. Así, en lugar de pasar de la Figura 6-22(a) a la Figura 6-22(b), pasamos de la Figura 6-22(a) a la Figura 6-22(c) cuando se liberan tres bloques. Ahora el sistema puede manejar una serie de ficheros temporales sin tener que efectuar E/S de disco. Si se llena el bloque que está en la memoria, se escribe en el disco y el bloque a medio llenar que estaba en el disco se pasa a memoria. Lo que se busca aquí es mantener llenos la mayoría de los bloques de punteros que están en el disco (para reducir al mínimo el consumo de disco), pero mantener medio lleno el que está en la memoria para poder manejar la creación y eliminación de ficheros sin que la administración de la lista libre requiera E/S de disco.

Con un mapa de bits también es posible mantener un solo bloque en la memoria, acudiendo al disco para obtener otro, sólo si el que está en la memoria se llena o se vacía. Una ventaja adicional de este método es que si se efectúa toda la asignación de bloques a ficheros a partir de un solo bloque del mapa de bits, los bloques de disco de un fichero dado estarán cercanos unos a otros, con lo que se reducirá al mínimo el movimiento del brazo del disco. Puesto que el mapa de bits es una estructura de datos fija, si el núcleo se pagina (en forma parcial), el mapa de bits podrá colocarse en la memoria virtual y sus páginas se intercambiarán a la memoria cuando se necesiten.

Cuotas de disco

Para evitar que las personas acaparen demasiado espacio de disco, los sistemas operativos multiusuario a menudo tienen un mecanismo para imponer cuotas de disco. La idea consiste en que el administrador del sistema asigne a cada usuario una porción máxima de ficheros y bloques, y que el sistema operativo cuide que los usuarios no excedan su cuota. A continuación describiremos un mecanismo típico.

Cuando un usuario abre un fichero, se localizan los atributos y direcciones de disco y se colocan en una tabla de ficheros abiertos en la memoria principal. Entre los atributos hay una entrada que indica quién es el propietario del fichero. Cualquier aumento en el tamaño del fichero se cargará a la cuota del propietario.

Una segunda tabla contiene los registros de cuota de cada uno de los usuarios que tienen un fichero abierto en ese momento, aunque alguien más haya abierto ese fichero. Esta tabla se muestra en la Figura 6-23. La tabla es un extracto de un fichero de cuotas en disco para los

usuarios cuyos ficheros están abiertos en la actualidad. Cuando se cierran todos los ficheros, el registro se escribe en el fichero de cuotas.

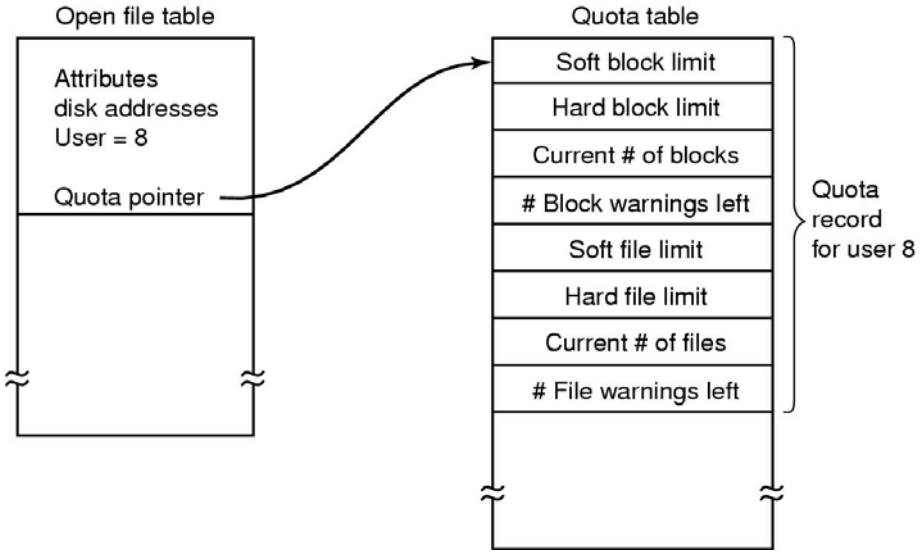

Figura 6-23. Se lleva el control de las cuotas por usuario en una tabla de cuotas.

Cuando se inserta una nueva entrada en la tabla de ficheros abiertos, se incluye en ella un puntero al registro de cuota del dueño de ese fichero, para poder hallar los diversos límites. Cada vez que se añade un bloque a un fichero, se incrementa el número total de bloques cargados al dueño, y se coteja con ambos límites, el estricto y el no estricto. El límite no estricto puede sobrepasarse, pero el estricto no. Un intento de aumentar el tamaño de un fichero cuando se ha llegado al límite de bloques estricto producirá un error. Se hacen verificaciones análogas para el número de ficheros.

Cuando un usuario intenta iniciar la sesión, el sistema examina el fichero de cuotas para ver si ese usuario ha excedido el límite no estricto de número de bloques o el de número de ficheros. Si se ha violado cualquiera de esos límites, se da una advertencia y se reduce en uno el contador de advertencias restantes. Si ese contador llega a cero, significa que el usuario ha hecho caso omiso de la advertencia demasiadas veces, y no se le permitirá iniciar la sesión. El usuario tendrá que hablar con el administrador del sistema para que se le vuelva a dar permiso para iniciar una sesión.

Este método tiene la propiedad de que los usuarios podrían rebasar sus límites no estrictos durante una sesión, siempre que se deshagan del excedente antes de terminarla. Los límites estrictos nunca pueden excederse.

6.3.6 Fiabilidad del sistema de ficheros

La destrucción de un sistema de ficheros suele ser un desastre mucho más grave que la destrucción de un ordenador. Si un ordenador queda destruido por un incendio, un rayo o una taza de café vertida en un teclado, esto es algo molesto y que nos costará mucho dinero, pero normalmente es posible comprar una máquina nueva para reemplazarlo con un mínimo de problemas. Los ordenadores personales de bajo costo pueden reemplazarse en menos de una hora con sólo acudir a una tienda (excepto en las universidades, donde la aprobación de una orden de compra requiere pasar tres comisiones, cinco firmas y 90 días).

Si se pierde de forma irremisible el sistema de ficheros de un ordenador, sea por culpa del hardware, del software o de ratas que royeron los disquetes; la recuperación de toda la información será difícil, lenta y, en muchos casos, imposible. Para las personas cuyos programas, documentos, ficheros de clientes, registros fiscales, bases de datos, planes de marketing u otros datos se han esfumado, las consecuencias pueden ser catastróficas. Aunque el sistema de ficheros no puede ofrecer protección contra la destrucción física del equipo y los medios, sí puede ayudar a proteger la información. En esta sección examinaremos algunos de los aspectos de protección del sistema de ficheros.

Los disquetes suelen ser perfectos cuando salen de la fábrica, pero algunos de sus bloques pueden estropearse durante su uso. Los discos duros a menudo tienen bloques defectuosos desde el principio: simplemente cuesta demasiado fabricarlos sin defectos. Como vimos en el capítulo 5, el controlador por lo general maneja los bloques defectuosos, sustituyendo los sectores correspondientes por sectores de repuesto que se proveen para ese fin. A pesar de esta técnica hay otros problemas de fiabilidad que consideraremos a continuación.

Copias de seguridad (Backups) (Respaldos)

La mayoría de las personas no piensa que valga la pena hacer un backup de sus ficheros debido al tiempo y al esfuerzo que requiere, hasta que un buen día su disco muere de repente y se vuelven creyentes en su lecho de muerte. En cambio, las compañías (normalmente) entienden muy bien el valor de sus datos y por lo general hacen una copia de seguridad por lo menos una vez al día, casi siempre en cinta. Las cintas modernas contienen decenas o a veces centenares de gigabytes y cuestan unos cuantos centavos de dólar por gigabyte. No obstante, hacer copias de seguridad no es tan trivial como suena, así que examinaremos ahora algunos de los problemas que ello implica.

Los backups en cinta generalmente tienen por objeto resolver uno de dos problemas potenciales:

1. Recuperarse de desastres.
2. Recuperarse de la estupidez.

Lo primero comprende hacer que el ordenador funcione otra vez después de un fallo del disco, inundación u otra catástrofe natural. En la práctica, estos sucesos no se presentan con mucha frecuencia, y es por eso por lo que muchas personas no se molestan en hacer copias de respaldo. Esas mismas personas casi nunca tienen pólizas de seguro contra incendios para sus hogares por la misma razón.

El segundo problema es que muchas veces los usuarios borran por accidente ficheros que van a necesitar después. Este problema se presenta con tanta frecuencia que cuando un fichero se "borra" en Windows, en realidad no se borra, sino que se pasa a un directorio especial, la **papelera de reciclaje**, para poder encontrarlo y restaurarlo con facilidad después. Los respaldos llevan este principio más lejos aún y permiten restaurar, a partir de cintas de respaldo viejas, ficheros que se borraron días o incluso semanas atrás.

Hacer un respaldo toma mucho tiempo y ocupa una gran cantidad de espacio, por lo que es importante hacerlo de forma eficiente y conveniente. Estas consideraciones dan pie a varias preguntas. En primer lugar hay que plantearse la pregunta: ¿debe respaldarse todo el sistema de ficheros o sólo una parte? En muchas instalaciones, los programas ejecutables (binarios) se mantienen en una parte concreta del sistema de ficheros. No es necesario respaldar esos ficheros si pueden reinstalarse utilizando los CD-ROMs del fabricante. Además, casi todos los sistemas tienen un directorio para ficheros temporales. Casi nunca tiene objeto respaldar estos ficheros. En UNIX, todos los ficheros especiales (de dispositivos de E/S) están en el directorio */dev*. No sólo no es necesario respaldar ese directorio, sino que resultaría peligroso intentarlo porque el

programa de respaldo se bloquearía si intentase leer todos esos ficheros de principio a fin. En pocas palabras, casi siempre es recomendable respaldar sólo directorios específicos y todo su contenido, en lugar de respaldar todo el sistema de ficheros.

En segundo lugar, resulta una pérdida de tiempo respaldar ficheros que no han sufrido cambios desde la última vez en que se respaldaron, lo cual conduce a la idea de los volcados incrementales. La forma más sencilla de volcado incremental es efectuar un volcado (respaldo) completo de forma periódica, digamos cada semana o cada mes, y hacer un volcado diario sólo de los ficheros que se han modificado desde el último volcado completo. Algo mejor aún es volcar sólo los ficheros que se han modificado desde la última vez que se respaldaron. Si bien este esquema reduce al mínimo el tiempo de volcado, complica la recuperación; porque primero es preciso restaurar el último volcado completo, seguido de todos los volcados incrementales en orden inverso. Es común utilizar esquemas de volcado incremental más elaborados para facilitar la recuperación.

En tercer lugar, puesto que casi siempre se respaldan cantidades enormes de datos, podría ser una buena idea comprimirlos antes de escribirlos en cinta. Sin embargo, con muchos algoritmos de compresión un único pequeño defecto en la cinta de respaldo puede interferir con el algoritmo de descompresión e impedir la lectura de todo un fichero o incluso de toda una cinta. Por ello, hay que estudiar con detenimiento la decisión de comprimir o no el flujo de respaldo.

En cuarto lugar, es difícil respaldar un sistema de ficheros activo. Si se están añadiendo, borrando y modificando archivos y directorios durante el proceso de respaldo, la copia de seguridad resultante podría ser inconsistente. Sin embargo, como el respaldo podría tardar horas, podría ser necesario poner el sistema fuera de línea durante una buena parte de la noche para hacerlo, algo que no siempre es aceptable. Por ello, se han ideado algoritmos que toman "instantáneas" del estado del sistema de ficheros, copiando estructuras de datos cruciales, y obligando a que los cambios futuros en los ficheros y directorios copien los bloques en lugar de actualizarlos en su lugar (Hutchinson y otros, 1999). Así, el sistema de ficheros "se congela" efectivamente en el momento en que se toma la instantánea, y se puede respaldar con calma después.

En quinto y último lugar, la producción de respaldos presenta muchos problemas no técnicos en una organización. El mejor sistema de seguridad en línea del mundo podría ser inútil si el administrador del sistema guarda todas las cintas de respaldo en su oficina y la deja abierta y sin vigilancia cada vez que sale al pasillo para retirar salidas de la impresora. Lo único que tiene que hacer un espía es entrar un momento en la oficina, poner una pequeña cinta en su bolsillo y salir caminando como si nada. Adiós seguridad. Además hacer un respaldo diario no sirve de mucho si el incendio que destruye los ordenadores quema también todas las cintas de respaldo. Por ello dichas cintas deben conservarse en otro edificio, aunque esto representa más riesgos de seguridad. En Nemeth y otros (2000) se presenta un tratamiento exhaustivo de estos y otros problemas prácticos de administración. A continuación examinaremos sólo los aspectos técnicos de la generación de respaldos del sistema de ficheros.

Pueden adoptarse dos estrategias para volcar un disco en cinta: un volcado físico o un volcado lógico. Un **volcado físico** comienza por el bloque 0 del disco, escribe en orden todos los bloques de disco en la cinta de salida y se detiene cuando ha copiado el último bloque. Un programa así es tan simple que es probable que sea posible eliminar por completo los errores, algo que tal vez no pueda decirse acerca de ningún otro programa útil.

No obstante, vale la pena hacer varios comentarios acerca de los volcados físicos. Para empezar, no tiene sentido respaldar bloques de disco desocupados. Si el programa de volcado puede obtener acceso a la estructura de datos de bloques libres, podrá evitar el volcado de esos bloques. Sin embargo, el hecho de saltarse los bloques no utilizados obliga a escribir el número

de bloque (o su equivalente) antes de cada bloque, pues el bloque k de la cinta ya no es el bloque k en el disco.

Una segunda inquietud es el volcado de bloques defectuosos. Si el controlador de disco remapea todos los bloques defectuosos y los oculta del sistema operativo como describimos en la sección 5.4.4, el volcado físico funcionará sin problemas. Pero si el sistema operativo puede ver esos bloques y los mantiene en uno o más "fichero de bloques defectuosos" o mapas de bits, es absolutamente indispensable que el programa de volcado físico tenga acceso a esta información y evite vaciar esos bloques para evitar interminables errores de lectura de disco durante el proceso de vaciado.

Las ventajas principales del volcado físico son la sencillez y la rapidez (básicamente porque puede efectuarse a la velocidad del disco). Las desventajas principales son la imposibilidad de saltarse directorios específicos, de efectuar volcados incrementales y de restaurar ficheros individuales si se solicita. Por estas razones, casi todas las instalaciones efectúan volcados lógicos.

Un **volcado lógico** comienza en uno o más directorios específicos y respalda de forma recursivo todos los ficheros y directorios que se encuentran ahí y que hallan sido modificados desde alguna fecha base dada (por ejemplo, el último respaldo en el caso de un volcado incremental o la fecha de instalación del sistema en el caso de un volcado completo). Así, en un volcado lógico la cinta recibe una serie de directorios y ficheros identificados meticulosamente, lo cual facilita la restauración de un fichero o directorio específico cuando se solicite.

Puesto que el volcado lógico es la forma más común, examinaremos los pormenores de un algoritmo común utilizando el ejemplo de la Figura 6-24 como guía. Casi todos los sistemas UNIX utilizan este algoritmo. En la figura vemos un árbol de ficheros con directorios (cuadrados) y ficheros (círculos). El sombreado indica lo que se ha modificado desde la fecha base y por tanto debe volcarse. Los elementos sin sombrear no tienen que volcarse.

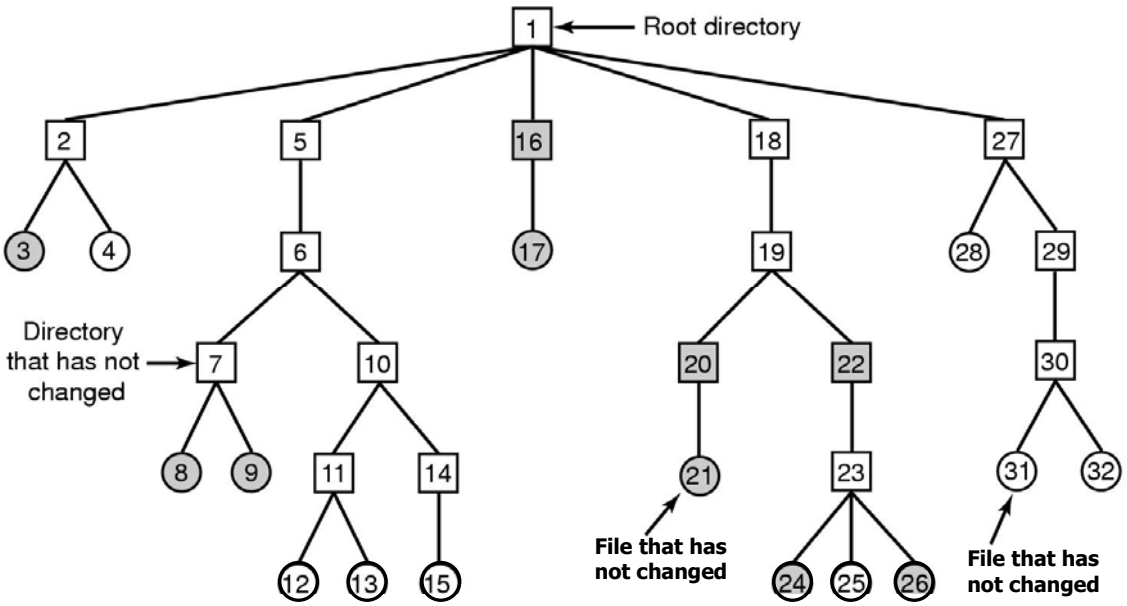

Figura 6-24. Sistema de ficheros que debe volcarse. Los cuadrados son directorios y los círculos son ficheros. Los elementos sombreados se modificaron después del último volcado. Cada directorio y fichero está rotulado con su número de i-nodo.

Este algoritmo también vuelca todos los directorios (aunque no se hayan modificado) que están en el camino que va hasta un fichero o directorio modificado, por dos razones. La primera es para que sea posible restaurar los ficheros y directorios volcados en un sistema de ficheros nuevo en otro ordenador. Así, pueden utilizarse los programas de volcado y restauración para transportar sistemas de ficheros completos de un ordenador a otro.

El segundo motivo para volcar los directorios no modificados que están por encima de ficheros modificados es hacer posible la restauración incremental de un único fichero (tal vez para recuperarse de una estupidez). Supongamos que se efectúa un volcado completo del sistema de ficheros el domingo por la noche y un volcado incremental el lunes por la noche. El martes se elimina el directorio */usr/jhs/proa/nr3*, junto con todos los directorios y ficheros que están más abajo. El miércoles temprano el usuario quiere restaurar el fichero */usr/jhs/proa/nr3/plans/sumary*. Sin embargo no es posible restaurar solamente el fichero resumen porque no hay donde ponerlo. Primero hay que restaurar los directorios *nr3* y *plans*. Para que sus propietarios, modos, tiempos, etcétera, sean los correctos, esos directorios deberán estar presentes en la cinta de volcado aunque no se hayan modificado desde el volcado completo anterior.

El algoritmo de volcado mantiene un mapa de bits indexado por el número de i-nodo, con varios bits por i-nodo. Se activan y desactivan bits en este mapa a medida que avanza el algoritmo. Éste último opera en cuatro fases. La fase 1 comienza en el directorio inicial (la raíz, en este ejemplo) y examina todas las entradas que contiene. Por cada fichero modificado, su i-nodo se marca en el mapa de bits. También se marca cada uno de los directorios (se haya modificado o no) y luego se inspecciona de forma recursiva.

Al término de la fase 1, todos los ficheros modificados y todos los directorios se han marcado en el mapa de bits, como se muestra (con sombreado) en la Figura 6-25(a). Conceptualmente, la fase 2 recorre el árbol otra vez de forma recursiva, quitando la marca a todos los directorios que no tengan (o en los que no haya debajo) ficheros o directorios modificados. Esta fase deja el mapa como se muestra en la Figura 6-25(b). Obsérvese que los directorios 10, 11, 14, 27, 29 y 30 ya no están marcados, porque debajo de ellos no hay nada que se haya modificado. Estos directorios no se vaciarán. En contraste, los directorios 5 y 6 sí se vacían aunque en sí no se hayan modificado, porque se necesitarán para restaurar los cambios de hoy en otra máquina. Por eficiencia, las fases 1 y 2 pueden combinarse en un único recorrido del árbol.

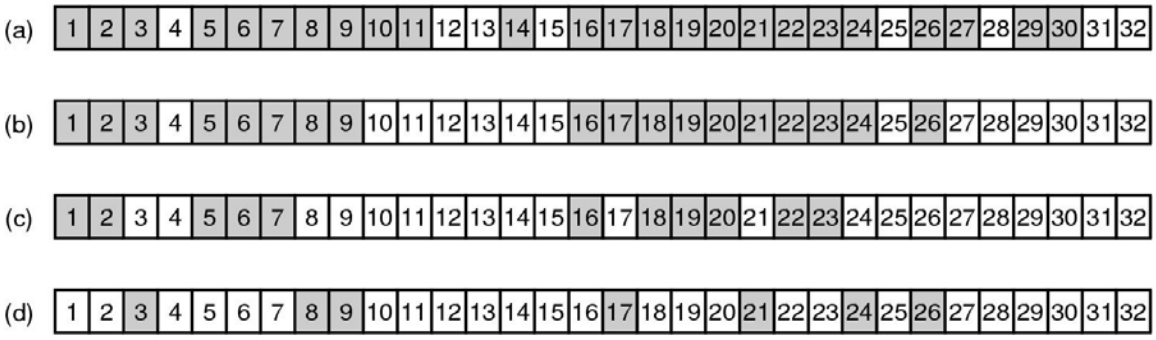

Figura 6-25. Mapas de bits utilizados por el algoritmo de volcado lógico.

Ahora se sabe qué directorios y ficheros deben volcarse. Son los que se marcaron en la Figura 6-25(b). En la fase 3 se exploran los i-nodos en orden numérico y se vuelcan todos los directorios marcados para ello. Éstos se muestran en la Figura 6-25(c). A cada directorio se le anteponen sus atributos (propietario, tiempos, etcétera) para que pueda restaurarse. Por último,

en la fase 4 se respaldan también los ficheros marcados en la Figura 6-25(d), anteponiéndoles también sus atributos. Con esto termina el volcado.

La restauración de un sistema de ficheros a partir de las cintas de volcado es directa. Lo primero que se hace es crear un sistema de ficheros vacío en el disco. Luego se restaura el volcado completo más reciente. Puesto que los directorios aparecen primero en la cinta, se restauran todos primero, para dar un esquema del sistema de ficheros. Luego se restauran los ficheros mismos. Después se repite este proceso con el primer volcado incremental efectuado después del volcado completo, seguido del siguiente, y así en forma sucesiva.

Aunque los volcados lógicos no son complicados, deben cuidarse algunos aspectos. Uno de ellos es que la lista de bloques libres no es un fichero, de modo que no se respalda y habrá que reconstruirla desde cero una vez que se hayan restaurado todos los volcados. Esto siempre es posible porque el conjunto de bloques libres no es más que el conjunto complementario del conjunto de bloques contenidos en todos los ficheros combinados.

Otro problema es el de los enlaces. Si un fichero está enlazado a dos o más directorios, es importante que se restaure sólo una vez y que todos los directorios que deben apuntar a él lo hagan.

Un tercer problema es el hecho de que los ficheros UNIX pueden contener huecos. Está permitido abrir un fichero, escribir unos cuantos bytes, luego posicionarse dentro del fichero a una gran distancia del principio y escribir otros pocos bytes. Los bloques intermedios no forman parte del fichero y no deben vaciarse ni restaurarse. Los ficheros del núcleo suelen tener un gran hueco entre el segmento de datos y la pila. Si esto no se maneja de forma correcta, cada uno de los ficheros de núcleo que se restauren tendrá esta área llena de ceros y por tanto será del mismo tamaño que el espacio de direcciones virtual (por ejemplo 2^{32} bytes o, peor aún, 2^{64} bytes).

Por último, los ficheros especiales, las tuberías con nombre y cosas así jamás deben vaciarse, estén en el directorio que estén (no es forzoso que estén en */dev*). Puede encontrarse más información acerca de los respaldos de sistemas de ficheros en Chervanak y otros (1998) y en Zwicky (1991).

Consistencia del sistema de ficheros

Otro área en la que la fiabilidad es un problema es en la consistencia de los sistemas de ficheros. Muchos sistemas de ficheros leen bloques, los modifican y los vuelven a escribir después. Si hay un fallo antes de que se escriban en el disco todos los bloques modificados, el sistema de ficheros podría quedar en un estado inconsistente. Este problema es grave sobre todo si algunos de los bloques que todavía no se han escrito en disco son bloques de i-nodo, bloques de directorio o bloques que contienen la lista libre.

Para resolver el problema de la inconsistencia del sistema de ficheros, casi todos los ordenadores tienen un programa de utilidad que verifica la consistencia. Por ejemplo, UNIX tiene *fsck* y Windows tiene *scandisk*. Este programa puede ejecutarse cada vez que se arranca el sistema, en especial después de una caída del sistema. La descripción que sigue es del funcionamiento de *fsck*. *Scandisk* es un tanto diferente porque opera con un sistema de ficheros distinto, pero el principio general de aprovechar la redundancia inherente del sistema de ficheros para repararlo sigue siendo válido. Todos los verificadores de sistemas de ficheros examinan cada sistema de ficheros (partición de disco) independientemente de los demás.

Es posible efectuar dos tipos de verificaciones de consistencia: de bloques y de ficheros. Para verificar la consistencia de los bloques, el programa construye dos tablas, cada una de las cuales contiene un contador para cada bloque, que en un principio se establecen a 0. Los contadores de la primera tabla cuentan las veces que cada bloque está presente en un fichero.

Los de la segunda tabla cuentan las veces que cada bloque está presente en la lista de bloques libres (o en el mapa de bits de bloques libres).

A continuación, el programa lee todos los i-nodos. Partiendo de un i-nodo, es posible construir una lista de todos los números de bloque empleados en el fichero correspondiente. A medida que se lee cada número de bloque, se incrementa su contador en la primera tabla. Luego el programa examina la lista libre o el mapa de bits para hallar todos los bloques que no se están utilizando. Cada aparición de un bloque en la lista de bloques libres hace que se incremente su contador en la segunda tabla.

Si el sistema de fichero es consistente, todos los bloques tendrán un 1 en la primera tabla o bien en la segunda, pero no en las dos, como se ilustra en la Figura 6-26(a). Sin embargo, como resultados de un fallo en las tablas podrían quedar como en la Figura 6-26(b), donde el bloque 2 está ausente de ambas tablas. Esto debe de ser indicado por el programa como un **bloque perdido** (*missing blok*). Aunque los bloques perdidos realmente no representan ningún peligro, desperdician espacio y por tanto reducen la capacidad del disco. La solución en este caso es sencilla: el verificador del sistema de ficheros simplemente los añade a la lista de bloques libres.

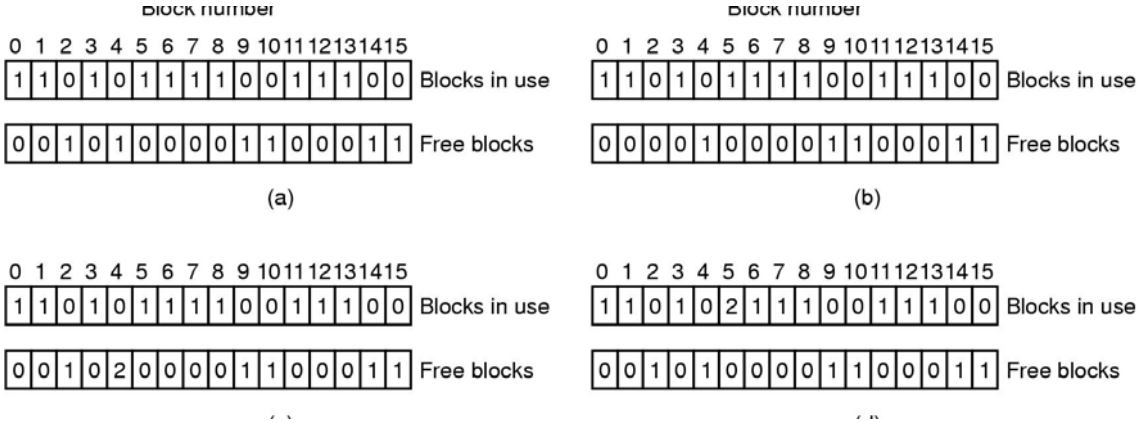

Figura 6-26. Estados del sistema de ficheros. (a) Consistente. (b) Bloque perdido. (c) Bloque repetido en la lista de bloques libres. (d) Bloque de datos repetido.

Otra situación que podría presentarse es la de la Figura 6-26(c). Aquí vemos un bloque, el número 4, que aparece dos veces en la lista de bloques libres. (Sólo puede haber bloques repetidos si la lista de bloques libres está realmente implementada como una lista; con un mapa de bits es imposible.) La solución en este caso es también sencilla: reconstruir la lista de bloques libres.

Lo peor que puede suceder es que el mismo bloque de datos esté presente en dos o más ficheros, como en el bloque 5 en la Figura 6-26(d). Si se borra cualquiera de estos dos ficheros, el bloque 5 se colocará en la lista de bloques libres y dará pie a una situación en la que el mismo bloque esté en uso y libre al mismo tiempo. Si se borran ambos ficheros, el bloque se colocará dos veces en la lista de bloques libres.

La acción apropiada del verificador del sistema de ficheros es asignar un bloque libre, copiar en él el contenido del bloque 5 e insertar la copia en uno de los ficheros. Así, el contenido de información de los ficheros no cambia (aunque con toda seguridad uno de ellos no será correcto), pero al menos el sistema de ficheros recupera la consistencia. Debe informarse del error, para que el usuario pueda inspeccionar el daño.

Además de comprobar que se conozca dónde están todos los bloques, el verificador del sistema de ficheros examina también el sistema de directorios. Aquí también se utiliza una tabla de contadores, pero estos son de ficheros, no de bloques. Se parte del directorio raíz, descendiendo de forma recursiva por el árbol, inspeccionando cada directorio del sistema de ficheros. Para cada fichero de cada directorio, se incrementa un contador por cada uso de ese fichero. Recordemos que, debido a los enlaces duros, un fichero podría aparecer en dos o más directorios. Los enlaces simbólicos no cuentan y no hacen que se incremente el contador del fichero en cuestión.

Al terminar el verificador, tiene una lista indexada por el número de i-nodo, que indica cuántos directorios contiene cada fichero. Luego compara estas cifras con los contadores de enlaces almacenados en los mismos i-nodos. Estos contadores son 1 cuando se crea el fichero y se incrementan cada vez que se establece un enlace (duro) con el fichero. En un sistema de ficheros consistente, ambos contadores coinciden. Sin embargo, pueden presentarse dos tipos de errores: el contador de enlaces en el i-nodo puede ser demasiado alto o demasiado bajo.

Si el contador de enlaces es mayor que el número de entradas en los directorios, entonces aunque se eliminen todos los ficheros de esos directorios el contador seguirá siendo mayor que cero y el i-nodo no se eliminará. Este error no es grave, pero desperdicia espacio en el disco con ficheros que no están en ningún directorio. Debe corregirse asignando el valor correcto al contador de enlaces en el i-nodo.

El otro error podría ser catastrófico. Si dos entradas de directorio están enlazadas en un fichero pero el i-nodo dice que sólo hay una, cuando se borre cualquiera de las dos entradas de directorio el contador del i-nodo pasará a valer cero. Cuando esto sucede, el sistema de ficheros marca el i-nodo como no utilizado y libera todos sus bloques. Esta acción hará que uno de los directorios apunte a un i-nodo que no se utiliza y cuyos bloques tal vez pronto se asignen a otros ficheros. Una vez más, la solución es simplemente asignar el número real de entradas de directorio al contador de enlaces en el i-nodo.

Estas dos operaciones, la verificación de bloques y la verificación de directorios, suelen integrarse para mejorar la eficiencia (ya que sólo habrá que dar una pasada por los i-nodos). También pueden efectuarse otras verificaciones. Por ejemplo, los directorios tienen un formato definido, con números de i-nodo y nombres ASCII. Si un número de i-nodo es mayor que el número de i-nodos que hay en disco, quiere decir que el directorio está dañado.

Además, cada i-nodo tiene un modo, algunos de los cuales son válidos pero extraños, como el 0007, que prohibe al propietario del fichero y a su grupo todo tipo de acceso, pero permite a cualquier otra persona leer, escribir y ejecutar el fichero. Podría ser útil informar por lo menos sobre cualquier fichero que otorga a extraños más privilegios que al propietario. Los directorios con más de, digamos, 1000 entradas son también sospechosos. Los ficheros situados en directorios de usuario pero que son propiedad del superusuario y tienen activado el bit SETUID, son problemas de seguridad potenciales porque permiten a cualquier usuario que los ejecute adquiera las facultades del superusuario. Con un poco de esfuerzo, es posible armar un lista relativamente larga de situaciones permitidas desde el punto de vista técnico, pero que no dejan de ser peculiares y de las que podría ser recomendable informar.

En los párrafos anteriores analizamos el problema de proteger al usuario contra las caídas del sistema. Algunos sistemas de ficheros se preocupan también por proteger al usuario contra sí mismo. Si el usuario quiere teclear

```
rm *.o
```

para eliminar todos los ficheros que terminan por *.o* (ficheros objeto generados por el compilador) pero por accidente teclea

```
rm * .o
```

(observe el espacio después del asterisco), *rm* borrará todos los ficheros del directorio actual y luego se quejará de que no puede encontrar *.o*. En MS-DOS y algunos otros sistemas, cuando se borra un fichero, lo único que sucede es que se activa un bit en el directorio o en el i-nodo para marcar el fichero como borrado. No se devuelven bloques libres a la lista de bloques libres mientras no se necesiten de verdad. Por tanto, si el usuario descubre el error de inmediato, es posible ejecutar un programa de utilidad especial que restaura los ficheros borrados. En Windows, los ficheros que se borran se colocan en la papelera de reciclaje, de donde se pueden recuperar sin es necesario. Por supuesto, no se libera el espacio que ocupan hasta que los ficheros no se borran realmente de la papelera de reciclaje.

6.3.7 Rendimiento del sistema de ficheros

El acceso a un disco es mucho más lento que el acceso a la memoria. La lectura de una palabra de memoria podría tardar 10 ns. La lectura de un disco duro podría efectuarse a 10 MB/s. lo cual es 40 veces más lento por palabra de 32 bits, pero a esto se le debe añadir de 5 a 10 ms para situar el brazo del disco hasta la pista y luego esperar a que el sector deseado esté debajo de la cabeza lectora. Si sólo se necesita una palabra, el acceso a la memoria es del orden de un millón de veces más rápido que el acceso al disco. En vista de esta diferencia en el tiempo de acceso, muchos sistemas de ficheros se han diseñado con diversas optimizaciones para mejorar su rendimiento. En esta sección cubriremos tres de ellas.

Uso de caché

La técnica empleada más comúnmente para reducir los accesos al disco es la **caché de bloques** o **caché de búfer**. (La palabra caché proviene del verbo francés cacher, que significa esconder.) En este contexto, una caché es una colección de bloques que lógicamente debían de estar en el disco pero que se están manteniendo en la memoria por cuestiones de eficiencia.

Pueden utilizarse diversos algoritmos para administrar la caché, pero uno muy común es mirar, en cada solicitud de lectura, si el bloque deseado está en la caché. Si está ahí, la solicitud de lectura podrá satisfacerse sin acceder al disco. Si el bloque no está en la caché, primero se lee del disco y se coloca en la caché y luego se copia donde se necesita. Las solicitudes posteriores que pidan el mismo bloque podrán satisfacerse desde la caché.

El funcionamiento de la caché se ilustra en la Figura 6-27. Puesto que hay muchos bloques en la caché (a menudo miles), se necesita algún mecanismo para determinar con rapidez si un bloque está presente o no. El mecanismo común consiste en dispersar (hash) el dispositvo y la dirección del disco, y buscar el resultado en una tabla hash. Todos los bloques con el mismo valor de hash están encadenados en una lista enlazada para poder seguir la cadena de colisiones.

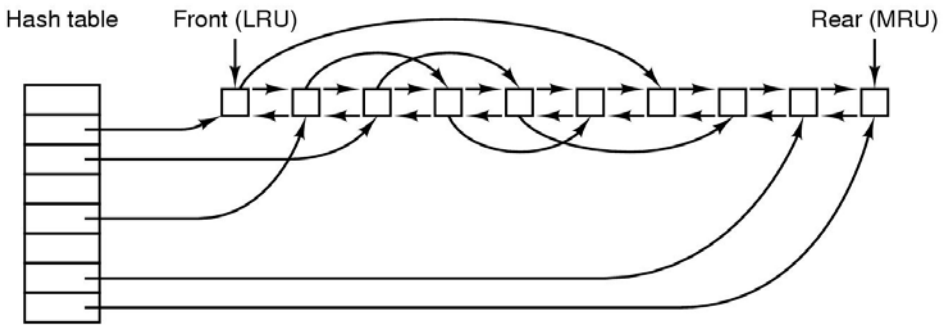

Figura 6-27. Estructuras de datos del caché de búfer.

Cuando es necesario cargar un bloque en una caché llena, hay que liberar algún bloque (y reescribirlo en el disco si ha sido modificado desde que se colocó allí). Esta situación se parece mucho a la paginación, y son aplicables todos los algoritmos de sustitución de páginas usuales que describimos en el capítulo 3, como FIFO, segunda oportunidad y LRU. Una diferencia agradable entre la paginación y el uso de caché es que la referencias a la caché no son demasiado frecuentes, por lo que es factible mantener todos los bloques en orden LRU exacto con listas enlazadas.

En la Figura 6-27 vemos que, además de las cadenas de colisiones que comienzan en la tabla hash, también hay una lista doblemente enlazada con todos los bloques ordenados según el orden de utilización, con el bloque utilizado menos recientemente al frente de la lista y el bloque utilizado más recientemente al final. Cuando se hace referencia a un bloque, se puede quitar de su posición en la lista bidireccional y colocarse en el extremo. De esta manera, puede mantenerse orden LRU exacto.

Por desgracia hay un pequeño problema. Ahora que tenemos una situación en la que es posible utilizar LRU exacto, resulta que no es recomendable. El problema tiene que ver con las caídas del sistema y la consistencia del sistema de ficheros que examinamos en la sección anterior. Si se lee del disco un bloque crucial, digamos un bloque de i-nodos, se coloca en la caché y se modifica, pero no se reescribe en el disco, un fallo dejaría al sistema de ficheros en un estado inconsistente. Si el bloque de i-nodos se coloca en el extremo de la cadena LRU, podría pasar un buen tiempo antes de que llegue al frente y se reescriba en el disco.

Además, es raro que se haga referencia a algunos bloques, como los de i-nodos, dos veces dentro de un intervalo de tiempo corto. Estas consideraciones nos llevan a un esquema LRU modificado que toma en cuenta dos factores:

1. ¿Es probable que se vuelva a necesitar pronto el bloque?
2. ¿El bloque es indispensable para la consistencia del sistema de ficheros?

Para contestar a ambas preguntas, los bloques pueden dividirse en categorías como bloques de i-nodos, bloques indirectos, bloques de directorio, bloques de datos llenos y bloques de datos parcialmente llenos. Los bloques que es probable que no se vayan a necesitar pronto se colocan al frente de la lista LRU, no al final, para que sus búferes se reciclen rápido. Los bloques que podrían volver a necesitarse pronto, como un bloque parcialmente lleno en el que se está escribiendo, se colocan al final de la lista para que permanezcan en la caché largo tiempo.

La segunda pregunta es independiente de la primera. Si el bloque es indispensable para la consistencia del sistema de ficheros (básicamente todo menos los bloques de datos) y se ha modificado, se deberá escribir en disco de inmediato, sin importar en qué extremo de la lista LRU se coloque. Al escribir rápidamente los bloques cruciales, se reduce mucho la probabilidad de que una caída del sistema estropee el sistema de ficheros. Un usuario podría molestarse si uno de sus ficheros se estropeara por un fallo, pero seguramente estaría mucho más molesto si se pierde todo el sistema de ficheros.

Incluso con esta precaución para mantener la integridad del sistema de ficheros, no es conveniente mantener los bloques de datos en la caché demasiado tiempo antes de escribirlos en el disco. Consideremos la situación de una persona que está utilizando un ordenador personal para escribir un libro. Incluso si el escritor le ordena de forma periódica al programa editor que grabe en el disco el fichero que está editando, existe la posibilidad de que todo siga en la caché y que no se escriba nada en el disco. Si el sistema falla, la estructura del sistema de ficheros no se corrompe, pero se pierde todo un día de trabajo.

Esta situación no necesita presentarse muy a manudo para que tengamos un usuario muy descontento. Los sistemas adoptan dos métodos para resolver este problema. Lo que hace UNIX es ofrecer una llamada al sistema, sync, que escribe de inmediato en disco todos los bloques modificados. Cuando se inicia el sistema, se pone en marcha un programa en segundo plano, por lo normal llamado update, que da vueltas en un bucle infinito emitiendo llamadas sync, durmiéndose durante 30 segundos entre llamadas. Gracias a esto, no se pierden más de 30 segundos de trabajo por una caída.

Lo que hace MS-DOS es escribir en disco todos los bloques modificados tan pronto como se modifican. Las cachés en las que todos los bloques modificados se escriben inmediatamente en el disco se denominan **cachés de escritura directa**, y requieren más E/S de disco que las cahés de otro tipo. La diferencia entre estos dos métodos puede observarse cuando un programa escribe un bloque de 1 KB hasta llenarlo, carácter a carácter. UNIX reúne todos los caracteres que están en la cahé y escribe el bloque en disco una vez cada 30 segundos, o cuando el bloque se desaloje de la caché. MS-DOS efectúa un acceso al disco por cada carácter que se escribe. Desde luego la mayoría de los programas utiliza búferes internos, así que normalmente no se escribe un carácter, sino una línea o unidad más grande en cada llamada al sistema write.

Una consecuencia de esta diferencia en la estrategia de almacenamiento en caché es que el simple hecho de sacar un disco (flexible) de un sistema UNIX sin emitir una llamada sync casi siempre da como resultado una pérdida de datos, y en muchos casos también un sistema de ficheros corrupto. Con MS-DOS no hay problema. Se escogieron estas diferentes estrategias porque UNIX se desarrolló en un entorno en el que los discos eran discos duros y no removibles, mientras que MS-DOS nació en el mundo de los disquetes. A medida que los discos duros se convirtieron en la norma, el método UNIX, al ser más eficiente, se convirtió también en la norma y se usa ahora en Windows con los discos duros.

Lectura adelantada de bloques

Una segunda técnica para mejorar el rendimiento aparente del sistema de ficheros es tratar de colocar bloques en la caché antes de que se necesiten, a fin de mejorar la tasa de aciertos. En particular, muchos ficheros se leen de forma secuencial. Cuando se pide al sistema de ficheros entregar el bloque k de un fichero, lo hace, pero cuando termina examina de manera subrepticia la caché para ver si ya está ahí el bloque $k + 1$. Si no está, se planifica la lectura de ese bloque con la esperanza de que, cuando se necesite, ya haya llegado a la caché o que, cuando menos, ya venga de camino.

Desde luego, tal estrategia de lectura adelantada sólo funciona en el caso de ficheros que se están leyendo de forma secuencial. Si el acceso a un fichero es aleatorio, la lectura adelantada no ayuda; de hecho, perjudica porque ocupa parte del ancho de banda del disco en la lectura de bloques que no se utilizarán y hace que se expulsen de la caché bloques que tal vez sí sean útiles (y quizá ocupa más ancho de banda de disco, aún si los bloques expulsados estaban modificados y fue necesario escribirlos primero en disco). Para ver si vale la pena la lectura adelantada, el sistema de ficheros puede determinar los patrones de acceso a los ficheros abiertos. Por ejemplo, un bit asociado con cada fichero puede indicar si está en "modo de acceso secuencial" o en "modo de acceso aleatorio". En un principio, se da por hecho que el fichero está en modo de acceso secuencial, pero cada vez que se efectúa un posicionamiento del brazo del disco, el bit se desactiva. Si luego vuelven a presentarse lecturas secuenciales, el bit se activa de nuevo. De este modo, el sistema de ficheros puede hacer conjeturas razonables respecto a si le conviene leer por adelantado o no. Si se equivoca de vez en cuando, no será ningún desastre; sólo se desperdiciará un poco del ancho de banda del disco.

Reducción del movimiento del brazo del disco

La utilización de caché y la lectura adelantada no son las únicas formas de mejorar el rendimiento del sistema. Otra técnica importante consiste en reducir los movimientos del brazo del disco colocando juntos, preferentemente en el mismo cilindro, los bloques a los que tal vez se tendrá acceso de forma secuencial. Cuando se escribe en un fichero de salida, el sistema de ficheros tiene que asignar los bloques uno por uno, conforme se van necesitando. Si el control de los bloques libres se lleva con un mapa de bits, y todo el mapa de bits está en la memoria principal, no será difícil escoger un bloque libre lo más cercano posible al bloque anterior. Si se utiliza una lista de bloques libres, parte de la cual están en el disco, será mucho más difícil asignar bloques cercanos entre sí.

No obstante, incluso con una lista de bloques libres puede lograrse cierta agregación de bloques. El truco es llevar el control del almacenamiento en el disco no por bloques, sino por grupos de bloques consecutivos. Si los sectores constan de 512 bytes, el sistema podría usar bloques de 1 KB (2 sectores), pero asignar el almacenamiento en disco en unidades de 2 bloques (4 sectores). Esto no es lo mismo que tener bloques de disco de 2 KB porque la caché de todos modos utilizará bloques de 1 KB y las transferencias de disco seguirán siendo de 1 KB, pero la lectura secuencial de un fichero en un sistema de otra manera ocioso reducirá el número de desplazamientos del brazo del disco a la mitad, mejorando el rendimiento de manera considerable. Una variación del mismo tema es tomar en cuenta el posicionamiento rotacional. Al asignar bloques, el sistema intenta colocar los bloques consecutivos de un fichero en el mismo cilindro.

Otro cuello de botella en lo tocante al rendimiento en sistemas que utilizan i-nodos o algún equivalente es que la lectura de cualquier fichero, por pequeño que sea, requiere dos accesos al disco: uno para el i-nodo y otro para el bloque. En la Figura 6-28(a) se muestra la colocación usual de los i-nodos. Aquí todos los i-nodos están cerca del principio del disco, por lo que la distancia media entre un i-nodo y sus bloques será alrededor de la mitad del número de cilindros, y requerirá largos desplazamientos del brazo.

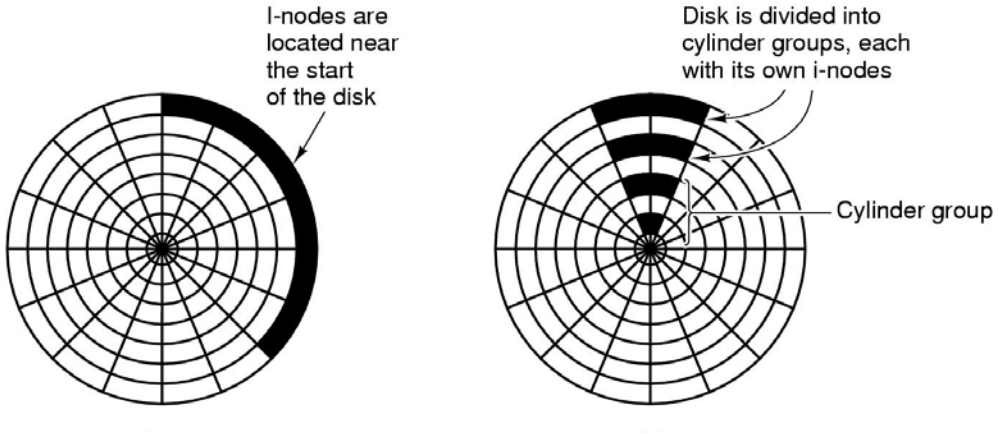

Figura 6-28. (a) i-nodos colocados al principio del disco. (b) Disco dividido en grupos de cilindros, cada uno con sus propios bloques e i-nodos.

Una forma fácil de mejorar el rendimiento es colocar los i-nodos a la mitad del disco, no al principio, con lo que se reduce a la mitad el desplazamiento medio del brazo entre el i-nodo y el primer bloque. Otra idea, que se muestra en la Figura 6-28(b), es dividir el disco en grupos de cilindros, cada uno con sus propios i-nodos, bloques y lista libre (McKusick y otros, 1984). Al crear un fichero nuevo puede escogerse cualquier i-nodo, pero se procura hallar un bloque en el mismo grupo de cilindros en el que está el i-nodo. Si no hay ninguno disponible, se utiliza un bloque de un grupo de cilindros cercano.

6.3.8 Sistemas de ficheros con estructura de registro

Los cambios tecnológicos están sometiendo a mucha presión a los sistemas de ficheros actuales. En particular, las CPUs cada día son más rápidas, los discos cada día son más grandes y de más bajo coste (pero no mucho más rápidos), y el tamaño de las memorias está creciendo de forma exponencial. El único parámetro que no está mejorando a ritmo vertiginoso es el tiempo de desplazamiento del brazo del disco. La combinación de estos factores implica que en muchos sistemas de ficheros esté apareciendo un cuello de botella en lo tocante a la eficiencia. Algunas investigaciones efectuadas en Berkeley intentaron aliviar este problema diseñando un tipo de sistemas de ficheros totalmente nuevo, el **sistema de ficheros con estructura de diálogo** (**LFS**; *Log-structured File System*). En esta sección describiremos de forma breve el funcionamiento del LFS. Puede encontarse un tratamiento más completo en Rosenblum y Ousterhout (1991).

La idea que impulsó el diseño del LFS es que conforme se vuelven más rápidas las CPUs y las memorias RAM aumentan de tamaño, las cachés de disco también están creciendo con rapidez. Por ello, ahora es posible satisfacer una fracción considerable de todas las solicitudes de lectura directamente de la caché del sistema de archivos, sin necesidad de accesos a disco. Una consecuencia de esta observación es que, en el futuro, casi todos los accesos al disco escrituras, así que el mecanismo de lectura adelantada empleado en algunos sistemas de ficheros para traer bloques antes de que se necesiten ya no mejora mucho el rendimiento.

Para empeorar las cosas, en la mayoría de los sistemas de ficheros las escrituras se efectúan en trozos muy pequeños. Las escrituras pequeñas son muy ineficientes, porque una escritura en disco de 50 μs muchas veces va precedida por un desplazamiento del brazo que tarda 10 ms y por un retraso rotacional de 4 ms. Con estos parámetros, la eficiencia del disco baja a una fracción de 1%.

Para ver de dónde provienen todas esas escrituras pequeñas, consideremos la creación de un fichero nuevo en un sistema UNIX. Para escribir este fichero, es preciso escribir el i-nodo del directorio, el bloque del directorio, el i-nodo del fichero y el fichero en sí. Aunque todas estas escrituras pueden aplazarse, eso expone al sistema de ficheros a problemas de consistencia graves si se presenta un fallo antes de que se efectúen las escrituras. Por ello, las escrituras de i-nodos generalmente se hacen de inmediato.

Con base en este razonamiento, los diseñadores del LFS decidieron reimplementar el sistema de ficheros de UNIX a modo de lograr el ancho de banda máximo del disco, aun cuando la carga de trabajo consista en su mayor parte en pequeñas escrituras aleatorias. La idea básica es estructurar el disco como un registro. De manera periódica, y cuando surge una necesidad especial de hacerlo, todas las escrituras pendientes que están en búferes en la memoria se reúnen en un único segmento y se escriben en el disco como un segmento contiguo único al final del registro. Así, un único segmento podría contener i-nodos, bloques de directorio y bloques de datos, todos revueltos. Al principio de cada segmento hay un resumen del segmento que indica su contenido. Si puede hacerse que el tamaño medio de tales segmentos sea cercano a 1MB, podrá aprovecharse casi todo el ancho de banda del disco.

En este diseño siguen existiendo i-nodos y tienen la misma estructura que en UNIX, pero ahora están dispersos por todo el registro, en lugar de estar en una posición fija en el disco. No obstante, cuando se localiza un i-nodo, la localización de los bloques se efectúa de la manera acostumbrada. Claro que ahora es mucho más difícil hallar un i-nodo, porque su dirección no puede calcularse simplemente a partir de su número-i, como en UNIX. Para que sea posible hallar i-nodos, se mantiene un mapa de ellos, indexado por número-i. La entrada *i* de este mapa apunta al i-nodo *i* en el disco. El mapa se mantiene en el disco, pero también en la caché, así que las partes más utilizadas estarán en la memoria la mayor parte del tiempo.

Para resumir lo que hemos dicho hasta ahora, todas las escrituras se colocan al principio en búferes en la memoria, y en forma periódica todas las escrituras almacenadas en el búfer se escriben en disco en un único segmento, al final del registro. La apertura de un fichero consiste ahora en utilizar el mapa para localizar el i-nodo del fichero. Una vez hallado ese i-nodo, proporciona las direcciones de los bloques. Todos los bloques estarán también en segmentos, en algún lugar del registro.

Si los discos fueran infinitamente grandes, la descripción anterior sería todo. Sin embargo, los discos reales son finitos, por lo que tarde o temprano el registro ocupará todo el disco, y ya no será posible escribir segmentos nuevos en él. Por fortuna, muchos segmentos existentes podrían tener bloques que ya no se necesitan. Por ejemplo, si un fichero se sobrescribió, su i-nodo apuntará ahora a los nuevos bloques, pero los viejos seguirán ocupando espacio en segmentos escritos antes.

Para resolver este problema, LFS tiene un proceso **limpiador** que dedica su tiempo a explorar el registro de forma circular para compactarla. Lo primero que hace es leer el resumen del primer segmento del registro para ver qué i-nodos y ficheros están ahí. Luego examina el mapa de i-nodos actual para ver si siguen vigentes y si los bloques de fichero todavía están en uso. Si no, esa información se desecha. Los i-nodos y bloques que todavía están en uso se pasan a la memoria para que se escriban en disco en el siguiente segmento. Ahora el segmento original se marca como libre, y el registro podrá utilizarlo para grabar datos nuevos. De este modo, el limpiador avanza por el registro, eliminando segmentos viejos de la parte de atrás y colocando los datos vigentes en la memoria para que se reescriban en el siguiente segmento. El resultado es que el disco actúa como un gran búfer circular, con el subprograma escritor añadiendo segmentos nuevos al frente, y el subprograma limpiador eliminando segmentos viejos en la parte posterior.

La compatibilización en este sistema no es trivial, pues cuando un bloque de ficheros se escribe en un nuevo segmento, es preciso localizar el i-nodo del fichero (en algún lugar del registro), actualizarlo y colocarlo en la memoria para que se escriba en el disco en el siguiente segmento. Luego hay que actualizar el mapa de i-nodos para que apunte a la nueva copia. No obstante, tal administración es factible, y las mediciones de rendimiento demuestran que toda esta complejidad vale la pena. Las mediciones presentadas en los artículos antes citados muestran que LFS mejora el rendimiento de UNIX en un orden de magnitud cuando las escrituras son pequeñas, y tiene un rendimiento tan bueno como el de UNIX, o mejor, en las lecturas y en las escrituras grandes.

6.4 EJEMPLOS DE SISTEMAS DE FICHEROS

En las secciones que siguen analizaremos varios ejemplos de sistemas de ficheros, que van desde muy sencillos hasta muy avanzados. Puesto que los sistemas de ficheros UNIX modernos y el sistema de ficheros nativo de Windows 2000 se tratan en los capítulos sobre UNIX (capitulo 10) y sobre Windows 2000 (capítulo 11), no veremos esos sistemas aquí. Lo que sí haremos a continuación será examinar a sus predecesores.

6.4.1 Sistemas de ficheros en CD-ROM

Como primer ejemplo de sistema de ficheros, consideremos los sistemas de ficheros que se utilizan en los CD-ROMs. Estos sistemas son notablemente sencillos porque se diseñaron para medios en los que sólo se escribe una vez. Por ejemplo, entre otras cosas, no consideran un control de bloques libres porque en un CD-ROM los bloques no pueden ni liberarse ni añadirse después de que se ha fabricado el disco. A continuación examinaremos el principal tipo de sistema de ficheros para CD-ROM y dos de sus extensiones.

El sistema de ficheros ISO 9660

El estándar más común para sistemas de ficheros en CD-ROM se adoptó como Estándar Internacional en 1998 bajo el nombre de **ISO 9660**. Casi todos los CD-ROMs que están en el mercado en la actualidad son compatibles con esta norma, a veces con las extensiones que analizaremos más adelante. Una de las metas de esa norma fue lograr que todo CD-ROM pudiera leerse en cualquier ordenador, independientemente de la ordenación de bytes y del sistema operativo empleados. Por ello, se impusieron algunas limitaciones al sistema de ficheros para que los sistemas operativos más débiles utilizados en ese entonces (como MS-DOS) pudieran leerlo.

Los CD-ROMs no tienen cilindros concéntricos como los discos magnéticos. En su lugar hay una única espiral continua que contienen los bits en sucesión lineal (aunque es posible hacer desplazamientos en dirección radial). Los bits a lo largo de la espiral se dividen en bloques lógicos (también llamados sectores lógicos) de 2352 bytes. Algunos de éstos son para preámbulos, corrección de errores y demás gasto adicional. La porción de "carga útil" de cada bloque lógico es de 2048 bytes. Cuando los CDs se utilizan para grabar música, tienen introducciones, terminaciones y espacios entre pistas, pero estos elementos no se usan en los CD-ROMs de datos. En muchos casos la posición de un bloque a lo largo de la espiral se especifica en minutos y segundos, pero se puede convertir en un número de bloque lineal aplicando el factor de conversión de 1 s = 75 bloques.

ISO 9660 reconoce conjuntos de CD-ROM con hasta $2^{16} - 1$ CDs en cada conjunto. Los CD-ROMs individuales también pueden dividirse en volúmenes lógicos (particiones). Sin embargo, en lo que sigue nos concentraremos es ISO 9660 para un solo CD-ROM sin particiones.

Cada CD-ROM comienza con 16 bloques cuya función no está definida por el estándar ISO 9660. Un fabricante de CD-ROMs podría utilizar esa área para grabar un programa de autoarranque que permita arrancar el ordenador desde el CD-ROM, o para algún otro fin. Luego viene un bloque que contiene el descriptor de volumen primario, el cual contiene información general acerca del CD-ROM. Dicha información incluye el identificador del sistema (32 bytes), el identificador de volumen (32 bytes), el identificador del productor (128 bytes) y el identificador del preparador de datos (128 bytes). El fabricante puede llenar esos campos como desee, pero sólo puede usar letras mayúsculas, dígitos y un número muy reducido de signos de puntuación, a fin de garantizar la compatibilidad entre plataformas.

El descriptor de volumen primario contiene también los nombres de tres ficheros, los cuales podrían contener el resumen, información sobre derechos de autor (*copyright*) e información bibliográfica, respectivamente. Además, están presentes ciertos números clave que incluyen el tamaño de bloque lógico (por lo normal 2048, pero se permiten 4096, 8192 y potencias superiores de 2 en ciertos casos), el número de bloques que tiene el CD-ROM y sus fechas de creación y expiración. Por último, el descriptor de volumen primario contiene también una entrada de directorio que indica dónde hallar el directorio raíz en el CD-ROM (es decir, en qué bloque comienza). Desde este directorio puede localizarse el resto del sistema de ficheros.

Además del descriptor de volumen primario, un CD-ROM puede contener un descriptor de volumen suplementario que contiene información similar al primario, pero no nos ocuparemos aquí de este descriptor.

El directorio raíz y todos los demás directorios, constan de un número variable de entradas, la última de las cuales contiene un bit que la marca como entrada final. Las entradas de directorio en sí también son de longitud variable. Cada una consta de 10 a 12 campos, algunos de los cuales están en ASCII mientras que otros son campos numéricos en binario. Los campos binarios están codificados dos veces, una vez en formato *little endian* (empleado por

ejemplo en el Pentium) y una en formato *big endian* (empleado por ejemplo en el SPARC). Así, un número de 16 bits ocupa 4 bytes y un número de 32 bits ocupa 8 bytes. Fue necesario utilizar esta codificación redundante para evitar lastimar los sentimientos de alguien durante el desarrollo del estándar. Si el estándar hubiera estipulado *little endian*, la gente de compañías con productos *big endian* se habría sentido menospreciada y no lo habría aceptado. Así, el contenido emocional de un CD-ROM puede cuantificarse y medirse con exactitud en kilobytes/hora de espacio desperdiciado.

El formato de una entrada de directorio ISO 9660 se ilustra en la Figura 6-29. Puesto que las entradas de directorio son de longitud variable, el primer campo es un byte que da la longitud de la entrada. Ese byte se define con el bit de orden alto a la izquierda a fin de evitar ambigüedades.

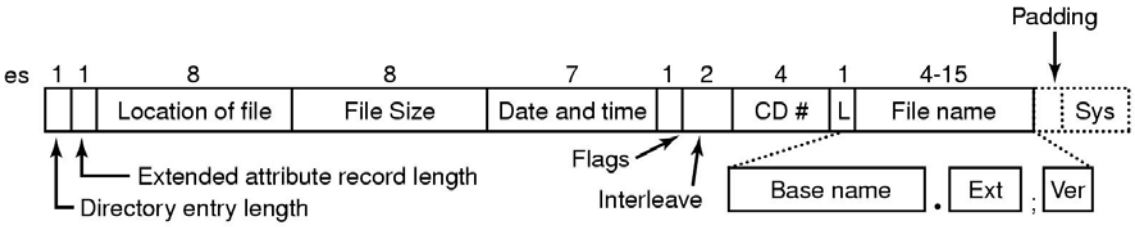

Figura 6-29. La entrada de directorio ISO 9660.

Las entradas de directorio pueden tener, de manera opcional, atributos extendidos. En tal caso, el segundo byte da la longitud del registro de atributos extendidos.

Luego viene la posición del bloque inicial del fichero mismo. Los atributos se almacenan como series contiguas de bloques, por lo que su ubicación queda especificada por completo con el bloque inicial y el tamaño, que viene en el siguiente campo.

La fecha y la hora en que se grabó el CD-ROM están en el campo que sigue, con bytes individuales para el año, mes día, hora, minuto, segundo y huso horario. Los años se comenzaron a contar en 1900, lo que implica que los CD-ROMs van a tener un problema en el año 2156 por que el año siguiente al 2155 será el 1900. Este problema podría haberse aplazado definiendo el origen del tiempo como 1988 (el año en el que se adoptó el estándar). Si se hubiera hecho así, el problema se habría pospuesto hasta 2244. Otros 88 años de respiro son muy buenos.

El campo *Marcadores* contiene algunos bits de uso diverso, incluyendo uno para ocultar esa entrada en los listados (algo que se copió de MS-DOS), uno para distinguir una entrada de un fichero de una que es un directorio, una para habilitar el uso de los atributos extendidos y una para marcar la última entrada de un directorio. Hay otros bits en este campo pero no nos ocuparemos de ellos aquí. El siguiente campo se ocupa de la intercalación de fragmentos de ficheros en una forma que no se usa en la versión más simple de ISO 9660, así que no hablaremos más de ella.

El campo que sigue indica en qué CD-ROM está el fichero. Es válido que una entrada de directorio en un CD-ROM corresponda a un fichero situado en otro CD-ROM corresponda a un fichero situado en otro CD-ROM del conjunto. Esto permite construir un directorio maestro en el primer CD-ROM, con una lista de todos los ficheros de todos los CD-ROMs del conjunto completo.

El campo marcado con una L en la Figura 6-29 da el tamaño del nombre de fichero en bytes, y va seguido del nombre de fichero mismo. Un nombre de fichero consta de un nombre base, un punto, una extensión, un signo de punto y coma y un número de versión binario (1 o 2 bytes). El nombre base y la extensión pueden contener letras mayúsculas, los dígitos del 0 al 9 y el carácter de subrayado. Todos los demás caracteres están prohibidos para garantizar que todos los ordenadores puedan manejar todos los nombres de fichero. El nombre base puede tener hasta ocho caracteres; la extensión puede tener hasta tres caracteres. Estas decisiones fueron obligadas por la necesidad de compatibilidad con MS-DOS. Un nombre de fichero dado puede estar presente en un directorio varias veces, mientras cada vez tenga un número de versión distinto.

Los últimos dos campos no siempre están presentes. El campo *Relleno* sirve para hacer que toda entrada de directorio tenga un número par de bytes, a fin de alinear los campos numéricos de entradas subsiguientes en fronteras de dos bytes. Si se necesita relleno, se utiliza un byte 0. Por último, tenemos el campo de *uso del sistema*. Su función y tamaño no están definidos, excepto que debe tener un número par de bytes. Los distintos sistemas lo utilizan de diferente forma. El Macintosh guarda ahí indicadores Finder, por ejemplo.

Dentro de un directorio, las entradas aparecen en orden alfabético con excepción de las dos primeras. La primera entrada es para el directorio en sí. La segunda es para su padre. En este sentido, las dos entradas son similares a las entradas de directorio . y .. de UNIX. Los ficheros no tienen que estar en orden por directorio.

No hay un límite explícito para el número de entradas de un directorio, pero sí para la profundidad de anidamiento: la profundidad máxima es ocho.

ISO 9660 define tres niveles. El nivel 1 es el más restrictivo y especifica que los nombres de fichero están limitados a 8 + 3 caracteres sin extensiones. La utilización de ese nivel ofrece la máxima garantía de que un CD-ROM podrá leerse en cualquier ordenador.

El nivel 2 relaja la restricción de longitud: permite que los ficheros y directorios tengan nombres de hasta 31 caracteres, pero el conjunto de caracteres permitidos sigue siendo el mismo.

El nivel 3 utiliza los mismos límites que el nivel 2 en cuanto a los nombres, pero relaja en parte el requisitote que los ficheros tienen que ser contiguos. Con este nivel, un fichero puede constar de varias secciones, cada una de las cuales es una serie contigua de bloques. La misma serie podría aparecer varias veces en un fichero y también podría aparecer en dos o más ficheros. Si en varios ficheros se repiten grandes porciones de datos, el nivel 3 permite optimizar un poco el espacio al no exigir que los datos estén presentes varias veces.

Extensiones Rock Ridge

Como hemos visto, ISO 9660 es muy restrictivo en varios sentidos. Poco después de que salió, algunos miembros de la comunidad UNIX comenzaron a trabajar en una extensión para poder representar sistemas de ficheros UNIX en un CD-ROM. Las extensiones se llamaron Rock Ridge, en memoria de un pueblo ficticio que aparece en la película de Gene Wilder, *Blazing Saddles*, quizá porque a uno de los miembros de la comunidad le gustaba la película.

Las extensiones aprovechan el campo *Uso del sistema* para hacer que los CD-ROMs Rock Ridge puedan leerse en cualquier ordenador. Todos los demás campos conservan el significado que tienen en ISO 9660. Si un sistema no reconoce las extensiones Rock Ridge, tan solo harña caso omiso de ellas y verá un CD-ROM normal.

Las extensiones se dividen en los campos siguientes:

1. PX – Atributos POSIX.
2. PN – Números de dispositivo principal y secundario.
3. SL – Enlace simbólico.
4. NM – Nombre alterno.
5. CL – Ubicación de hijo.
6. PL – Ubicación de padre.
7. RE – Reubicación.
8. TF – Sellos de tiempo.

El campo *PX* contiene los bits de autorización *rwxrwxrwx* estándar de UNIX para el dueño, grupo y otros. También contiene los demás bits contenidos en la palabra de modo, como los bits SETUID y SETGID, etcétera.

El campo *PN* está presente para poder representar dispositivos puros en un CD-ROM. Éste contiene los números de dispositivo principal y secundario asociados con el fichero. De este modo, podrá escribirse el contenido del directorio */dev* en un CD-ROM, pudiendo después reconstruirlo correctamente en el sistema de destino.

El campo *SL* es para enlaces simbólicos, permite que un fichero de un sistema de ficheros se refiera a un fichero en otro sistema de ficheros distinto.

Tal vez el campo más importante sea *NM*, que permite asociar un segundo nombre al fichero. Este nombre no está sujeto a las restricciones de ISO 9660 en cuanto a conjunto de caracteres y longitud, y permite expresar nombre de fichero UNIX arbitrarios en un CD-ROM.

Los tres campos que siguen se utilizan juntos para evitar el límite impuesto por ISO 9660 de solo poder anidar directorios hasta una profundidad de ocho. Si se utilizan, es posible especificar que un directorio debe reubicarse e indicar en qué lugar de la jerarquía debe ir. Esta es una manera artificial de superar la limitación en la profundidad de los directorios.

Por último, el campo *TF* contiene los tres sellos de tiempo incluidos en cada nodo-i de UNIX, a saber, la hora de creación, la hora de la última modificación y la del último acceso. Juntas estas extensiones permiten copiar un sistema de ficheros UNIX en un CD-ROM y luego restaurarlo por completo en otro sistema.

Extensiones Joliet

La comunidad UNIX no fue el único grupo que quería una forma de extender ISO 9660. A Microsoft también le pareció demasiado restrictivo (aunque fue precisamente el propio MS-DOS de Microsoft el que obligó a imponer la mayoría de las restricciones). Por tanto, Microsoft inventó ciertas extensiones que recibieron el nombre de **Joliet**. Se diseñaron para poder copiar sistemas de ficheros Windows en CD-ROM y luego restaurarlos, exactamente de la misma forma que se diseñó Rock Ridge para UNIX. Casi todos los programas que se ejecutan bajo Windows y utilizan CD-ROM reconocen Joliet, incluyendo los programas que queman CD-R. Por lo regular estos programas permiten escoger entre los distintos niveles de ISO 9660 y Joliet.

Las principales extensiones que ofrece Joliet son:

1. Nombre largos de fichero.
2. Conjunto de caracteres Unicode.
3. Anidación de directorios a más de ocho niveles.
4. Nombre de directorio con extensiones.

La primera extensión permite nombre de fichero de hasta 64 caracteres. La segunda permite utilizar el conjunto de caracteres Unicode en los nombre de fichero. Esta extensión es

importante para software destinado a utilizarse en países que no utilizan el alfabeto latino, como Japón, Israel y Grecia. Puesto que los caracteres Unicode ocupan dos bytes, la longitud máxima de un nombre de fichero en Joliet es de 128 bytes.

Al igual que Rock Ridge, Joliet elimina la limitación respecto a anidamiento de directorios. Los directorios pueden anidarse hasta cualquier profundidad que se requiera. Por último. Los nombres de directorio pueden tener extensiones. No queda claro por qué se incluyó esta extensión, pues en Windows los directorios casi nunca llevan extensiones, pero quizá algún día lo harán.

6.4.2 El sistema de ficheros de CP/M

Los primeros ordenadores personales (entonces llamados microordenadores) salieron a principios de la década de 1980. De esas primeros microordenadores, uno de los más populares utilizaba la CPU 8080 de Intel, de 8 bits, tenía 4 KB de RAM y contaba con un solo disco flexible de 8 pulgadas, con capacidad para 180 KB. Versiones posteriores utilizaron la CPU Zilog Z80, un poco más elaborada (pero todavía de 8 bits), tenían hasta 64 KB de RAM y utilizaban disquetes con la descomunal capacidad de 720 KB como dispositivo de almacenamiento masivo. A pesar de la baja velocidad y la pequeña cantidad de RAM, casi todas esas máquinas ejecutaban un sistema operativo basado en disco de una potencia sorprendente, llamado **CP/M** (Programa de Control para Microordenadores) (Goleen y Pechura, 1986). Este sistema dominó su época tanto como MS-DOS y después Windows dominaron el mundo de los PCs de IBM. Dos décadas después, ha desaparecido sin dejar huella (con la excepción de un reducido grupo de aficionados de hueso colorado), lo que permite pensar que los sistemas que ahora dominan el mundo podrían ser casi desconocidos cuando los bebés actuales se conviertan en estudiantes universitarios (¿Windows qué?).

Vale la pena echar un vistazo a CP/M por varias razones. La primera es que desde una perspectiva histórica, fue un sistema muy importante y fue el antepasado directo de MS-DOS. La segunda es que es probable que los diseñadores de sistemas operativos actuales y futuros que piensan que un ordenador necesita 32 MB tan solo para arrancar el sistema operativo podrían aprender mucho en cuanto a sencillez de un sistema que operaba de forma muy satisfactoria en 16 KB de RAM. La tercera es que, en las décadas por venir, van a ser muy comunes los sistemas empotrados. Debido a las restricciones de coste, espacio, peso y consumo de electricidad, los sistemas operativos empleados en, por ejemplo, relojes, cámaras, radios y teléfonos celulares van a tener que ser esbeltos y ágiles, como CP/M. Claro que esos sistemas no tienen disquetes de 8 pulgadas, pero bien podrían tener discos electrónicos que utilizan memoria flash, y es sencillo construir un sistema de ficheros de tipo CP/M en un dispositivo así.

La organización de CP/M en la memoria se muestra en la Figura 6-30. En la parte más alta de la memoria principal (en RAM) está el BIOS, que contiene una biblioteca básica de 17 llamadas de E/S utilizadas por CP/M (en esta sección describiremos CP/M 2.2, que fue la versión estándar cuando CP/M estaba en el cénit de su popularidad). Estas llamadas leen y escriben en el teclado, la pantalla y el disquete.

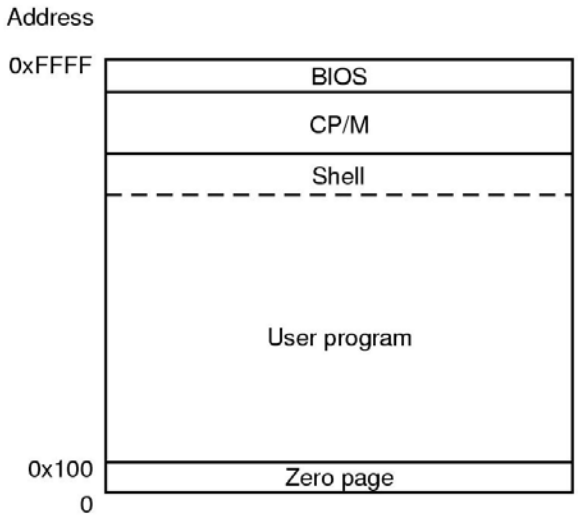

Figura 6-30. Organización de la memoria en CP/M.

Justo debajo del BIOS está el sistema operativo propiamente dicho. El tamaño del sistema operativo en CP/M 2.2 es de 3584 bytes. Increíble pero cierto: un sistema operativo completo en menos de 4 KB. Debajo del sistema operativo está el shell (procesador de la línea de comandos), que ocupa otros 2 KB. El resto de la memoria es para programas de usuario, con excepción de los 256 bytes de hasta abajo, que se reservan para los vectores de interrupción del hardware, unas cuantas variables y un búfer para la línea de comandos actual, con el fin de que los programas de usuario tengan acceso a ella.

El motivo para separar el BIOS de CP/M en sí (aunque ambos están en la RAM) fue la portabilidad. CP/M sólo interactúa con el hardware emitiendo llamadas al BIOS. Para portar CP/M a otra máquina, sólo es necesario trasladar ahí el BIOS. Una vez hecho eso, podrá instalarse CP/M sin modificación.

Un sistema CP/M sólo tiene un directorio, que contiene entradas de tamaño fijo (32 bytes). El tamaño del directorio, aunque fijo para una implementación dada, podría ser distinto en otras implementaciones de CP/M. Todos los ficheros del sistema aparecen en este directorio. Después de que CP/M arranca, lee el directorio y calcula un mapa de bits que contiene los bloques de disco libres, viendo qué bloques no están en ningún fichero. Este mapa de bits, que sólo ocupa 23 bytes para un disco de 180 KB, se mantiene en la memoria durante la ejecución. En el momento de apagar el sistema el mapa se desecha; es decir, no se escribe en el disco. Esto elimina la necesidad de un verificador de consistencia del disco (como *fsck*) y ahorra un bloque en el disco (equivalente en porcentaje al ahorro de 90 MB en un disco moderno de 16 GB).

Cuando el usuario teclea un comando, lo primero que hace el *shell* es copiarlo en un búfer en los 256 bytes más bajos de la memoria. Luego busca el programa a ejecutar, lo carga en la memoria en la dirección 256 (a continuación de los vectores de interrupción) y salta a él. Luego comienza la ejecución del programa, el cual descubre sus argumentos examinando el búfer de la línea de comandos. El programa puede sobrescribir el *shell* si necesita la memoria. Cuando termina el programa, emite una llamada al sistema CP/M para pedirle que vuelva a cargar el *shell* (si lo sobrescribió) y lo ejecute. En pocas palabras, así es como funciona CP/M.

Además de cargar programas, CP/M ofrece 38 llamadas al sistema, en su mayoría servicios de ficheros, para los programas de usuario. Las llamadas más importantes son las que leen y escriben ficheros. Para poder leer de un fichero es necesario abrirlo. Cuando CP/M recibe una llamada al sistema open, tiene que leer el único directorio y buscar en él el fichero. El

directorio no se mantiene en la memoria todo el tiempo, para ahorrar la escasa RAM. Cuando CP/M encuentra la entrada, tiene de inmediato los números de bloques del disco, porque están almacenados ahí mismo en la entrada, al igual que todos los atributos. En la Figura 6-31 se muestra el formato de una entrada de directorio.

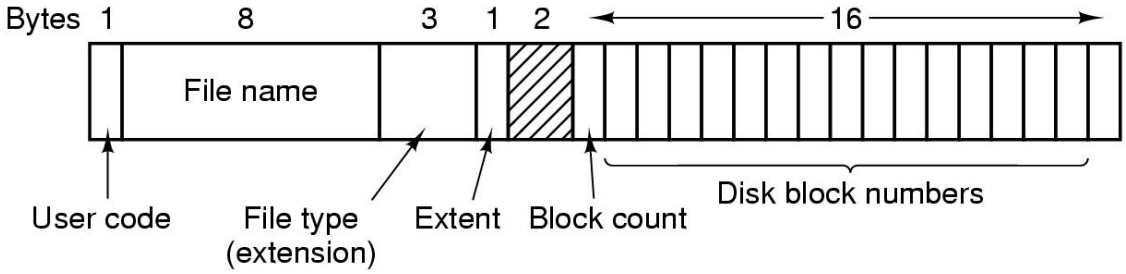

Figura 6-31. Formato de una entrada de directorio en CP/M.

Los campos de la Figura 6-31 se utilizan para lo siguiente: el campo *Código de usuario* indica quién es el dueño del fichero. Aunque sólo una persona puede trabajar con CP/M en un momento dado, el sistema reconoce múltiples usuarios que se turnan para utilizar el sistema. Al buscar un nombre de fichero, sólo se eliminan las entradas que pertenecen al usuario actual. En efecto, cada usuario tiene un directorio virtual sin el gasto adicional de administrar múltiples directorios.

Los dos campos que siguen dan el nombre y la extensión del fichero. El nombre base tiene hasta ocho caracteres; puede haber una extensión opcional de hasta tres caracteres. Sólo se permiten letras mayúsculas, dígitos y un número reducido de caracteres especiales en los nombres de fichero. Este esquema de 8 + 3 empleando sólo mayúsculas fue adoptado después por MS-DOS.

El campo *Número de bloques* indica cuántos bytes tiene el fichero medido en unidades de 128 bytes (porque la E/S se efectúa en sectores físicos de 128 bytes). El último bloque de 1 KB podría no estar lleno, así que el sistema no tiene forma de determinar el tamaño exacto de un fichero. Corresponde al usuario colocar un marcador de FIN de FICHERO si lo desea. Los últimos 16 campos contienen los números de bloques de disco en sí. Cada bloque ocupa 1 KB así que el tamaño máximo del fichero es de 16 KB. Cabe señalar que la E/S física se efectúa en sectores de 128 bytes y se lleva el control del tamaño en sectores, pero los bloques de los ficheros se asignan en unidades de 1 KB (ocho sectores a la vez) para evitar que la entrada de directorio sea demasiado grande.

No obstante, los diseñadores de CP/M se dieron cuenta de que algunos ficheros, incluso en un disquete de 180 KB, podrían exceder 16 KB, por lo que se incorporó una forma de soslayar el límite de 16 KB. Un fichero que tiene entre 16 KB y 32 KB no utiliza una entrada de directorio, sino dos. La primera contiene los primeros 16 bloques; la segunda contiene los otros 16. Más allá de 32 KB, se utiliza una tercera entrada de directorio, y así de forma sucesiva. El campo *Extensión* lleva el control del orden de las entradas de directorio para que el sistema sepa qué 16 KB vienen primero, cuáles vienen después, etcétera.

Después de una llamada open, se conocen las direcciones de todos los bloques de disco, así que la lectura es directa. La llamada write es también sencilla. Sólo se requiere asignar un bloque libre del mapa de bits que está en la memoria y luego escribir el bloque. Los bloques consecutivos de un fichero no se colocan en bloques consecutivos en el disco, porque el 8080 no puede procesar una interrupción y comenzar a leer el siguiente bloque a tiempo. Por ello, se utiliza intercalamiento para poder leer varios bloques en una única rotación.

Es evidente que CP/M no es el último grito de la moda en cuanto a los sistemas de ficheros avanzados, pero es sencillo, rápido y un programador competente puede implementarlo en menos de una semana. Para muchas aplicaciones integradas, bien podría ser todo lo que se necesita.

6.4.3 El sistema de ficheros de MS-DOS

Como primera aproximación, MS-DOS es una versión mejor y más grande de CP/M. Sólo se ejecuta en plataformas Intel, no maneja multiprogramación y sólo opera en el modo real del PC (que en un principio era el único modo). El *shell* tiene más funciones y hay más llamadas al sistema, pero la función básica del sistema operativo sigue siendo cargar programas, manejar el teclado y la pantalla, y administrar el sistema de ficheros. Esta última funcionalidad es la que nos interesa aquí.

El sistema de ficheros de MS-DOS sigue de cerca el patrón del sistema de CP/M, incluyendo la utilización de nombres de fichero de 8 + 3 caracteres (mayúsculas). La primera versión (MS-DOS 1.0) estaba limitada a un único directorio, igual que CP/M. Sin embargo, a partir de MS-DOS 2.0 se expandió de manera considerable la funcionalidad del sistema de ficheros. La principal mejora fue la inclusión de un sistema de ficheros jerárquico en el que los directorios podían anidarse hasta una profundidad arbitraria. Esto implicaba que el directorio raíz (que seguía teniendo un tamaño máximo fijo) podía contener subdirectorios, y estos podían contener otros subdirectorios, *ad infinitum*. No se permitían enlaces al estilo UNIX, así que el sistema de ficheros formaba un árbol a partir del directorio raíz.

Es común que los programas de aplicación creen un subdirectorio en el directorio raíz y coloquen allí todos sus ficheros (o en subdirectorios de ese subdirectorio), para evitar conflictos entre las distintas aplicaciones. Puesto que los directorios mismos se almacenan como ficheros, no hay límite para el número de directorios o ficheros que es posible crear. Sin embargo, a diferencia de CP/M, no existe el concepto de usuarios distintos en MS-DOS. Por ello, el usuario, que comenzó la sesión tiene acceso a todos los ficheros.

Para leer un fichero, un programa MS-DOS debe emitir primero una llamada al sistema open para obtener un identificador de fichero. La llamada especifica un camino, que podría ser absoluto o relativo al directorio de trabajo actual. El camino se examina componente a componente hasta que se encuentra el directorio final y se carga en la memoria. Luego se busca en él el fichero que se abrirá.

Aunque los directorios en MS-DOS tienen tamaño variable, igual que en CP/M, sus entradas son de tamaño fijo, 32 bytes. El formato de una entrada de directorio en MS-DOS se muestra en la Figura 6-32, contiene el nombre del fichero, sus atributos, la fecha y la hora en que se creó, el bloque inicial y el tamaño exacto del fichero. Los nombres de fichero de menos de 8 + 3 caracteres se ajustan a la izquierda y se rellenan con espacios a la derecha, por separado para cada campo. El campo *Atributos* es nuevo y contiene bits para indicar que el fichero es de solo lectura, que necesita salvaguardarse, que está oculto o que es un fichero del sistema. Los ficheros de sólo lectura no pueden escribirse. Esto se hace para protegerlos frente a daños accidentales. El bit de salvaguarda no tiene otra función real dentro del sistema operativo (es decir, MS-DOS no lo examina ni modifica su valor). La intención es que los programas a nivel de usuario puedan ponerlo a 0 tras hacer un copia de seguridad del fichero y que otros programas lo pongan a 1 cuando modifiquen el fichero. De este modo, un programa de backup puede examinar ese bit de atributo en todos los ficheros para ver cuáles son los ficheros que hay que copiar. El bit de fichero oculto puede ponerse a 1 para evitar que el fichero aparezca en los listados de directorio. Su principal uso es evitar que los usuarios novatos se confundan con ficheros que quizás no entiendan. Por último, el bit del sistema también oculta los ficheros.

Además, los ficheros del sistema no pueden borrarse por accidente con el comando *del*. Los principales componentes de MS-DOS tienen puesto ese bit a 1.

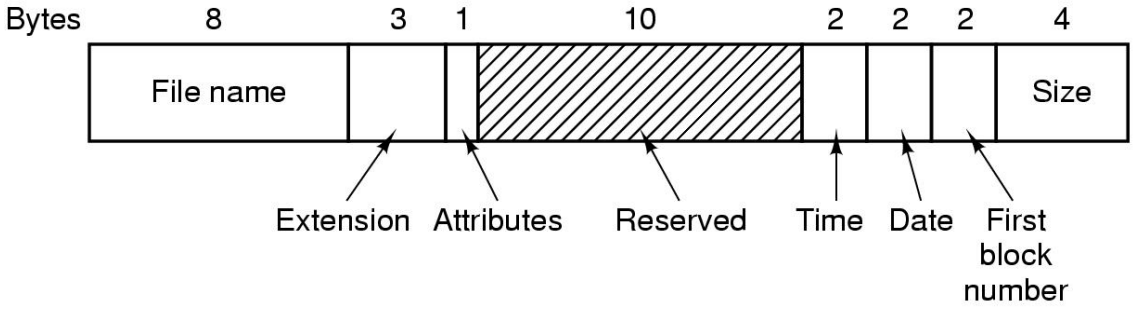

Figura 6-32. Una entrada del directorio de MS-DOS.

La entrada de directorio contiene también la hora y la fecha de creación o de última modificación del fichero. La hora tiene exactitud de ±2 segundos porque se almacena en un campo de dos bytes, que sólo puede contener 65536 valores únicos (un día contiene 86400 segundos distintos). El campo de hora se subdivide en segundos (5 bits), minutos (6 bits) y horas (5 bits). La fecha cuenta en días, utilizando tres subcampos: día (5 bits), mes (4 bits) y año−1980 (7 bits). Con un número de siete bits para el año y el tiempo comenzando en 1980, el año más alto que se puede expresar es 2107. Por tanto, MS-DOS tiene incorporado un problema del año 2108. Para evitar una catástrofe, los usuarios de MS-DOS deben tomar medidas lo antes posible para que sus programas tomen eso en cuenta. Si MS-DOS hubiera utilizado los campos de fecha y hora combinados en forma de un contador de segundos de 32 bits, podría haber representado cada segundo con exactitud y haber aplazado la catástrofe hasta 2116.

A diferencia de CP/M, que no almacena el tamaño exacto del fichero, MS-DOS sí lo hace. Puesto que se utiliza un número de 32 bits para el tamaño, en teoría los ficheros pueden tener hasta 4 GB. Sin embargo, otros límites (que describiremos a continuación) restringen el tamaño máximo de los ficheros a 2 GB o menos. Una porción sorprendentemente grande de la entrada (10 bytes) no se utiliza.

Otra diferencia entre MS-DOS y CP/M es que MS-DOS no almacena las direcciones de disco de un fichero en su entrada de directorio, tal vez porque los diseñadores se dieron cuenta de que los discos duros grandes (comunes entonces en miniordenadores) llegarían algún día al mundo de MS-DOS. En vez de eso, MS-DOS lleva el control de los bloques de fichero mediante una tabla de asignación de ficheros (FAT) en la memoria principal. La entrada de directorio contiene el número del primer bloque del fichero. Este número se utiliza como un índice para consultar una FAT de 64K entradas en la memoria principal. Siguiendo la cadena, es posible localizar todos los bloques. El funcionamiento de la FAT se ilustra en la Figura 6-14.

El sistema de ficheros FAT viene en tres versiones para MS-DOS: FAT-12, FAT-16 y FAT 32, dependiendo del número de bits que tenga una dirección de disco. En realidad, FAT-32 es un nombre engañoso porque sólo se utilizan los 28 bits de menor peso de las direcciones de disco. Se debería haber llamado FAT-28, pero las potencias de 2 suenan mucho más elegantes.

En todas las FATs, el bloque de disco puede definirse como algún múltiplo de 512 bytes (y puede ser diferente para cada partición), y el conjunto de tamaños de bloque permitidos (llamados **tamaños de *cluster*** por Microsoft) es diferente para cada variante. La primera versión de MS-DOS utilizaba FAT-12 con bloques de 512 bytes, lo que daba un tamaño de partición máximo de $2^{12} \times 512$ bytes (en realidad, sólo 4086×512, porque 10 de las direcciones de disco se utilizaban como marcadores especiales: fin de fichero, bloque defectuoso, etcétera).

Con estos parámetros, el tamaño máximo de una partición de disco era de aproximadamente 2 MB, y el tamaño de la FAT en la memoria era de 4096 entradas de dos bytes cada una. La utilización de una entrada de tabla de 12 bits habría resultado demasiado lento.

Este sistema funcionaba bien con discos flexibles, pero cuando salieron los discos duros se convirtió en un problema, que Microsoft resolvió permitiendo tamaños de bloque adicionales de 1, 2 y 4 KB. Este cambio conservó la estructura y el tamaño de la FAT-12, pero permitió particiones de disco de hasta 16 MB.

Puesto que MS-DOS reconocía cuatro particiones por unidad de disco, el nuevo sistema de ficheros FAT-12 funcionaba con discos de hasta 64 MB. Más allá de esa capacidad, algo tenía que ceder. Lo que sucedió fue la introducción de FAT-16, con punteros de disco de 16 bits. Además, se permitieron tamaños de bloque de 8, 16 y 32 KB. (32768 es la potencia de 2 más grande que puede representarse con 16 bits) La tabla FAT-16 ocupaba ahora 128 KB de la memoria principal todo el tiempo, pero como ya había memorias más grandes, entonces se empezó a utilizar más y pronto sustituyó al sistema de ficheros FAT-12. La partición de disco más grande que se puede manejar con FAT-16 es de 2 GB (64 K entradas de 32 KB cada una) y el disco más grande es de 8 GB, o sea, cuatro particiones de 2 GB cada una.

Para cartas de negocios, ese límite no representa ningún problema, pero para almacenar vídeo digital empleando el estándar DV, un fichero de 2 GB apenas contiene nueve minutos de vídeo. Una consecuencia del hecho de que un disco de PC sólo puede manejar cuatro particiones, es que el vídeo más largo que puede almacenarse en un disco tiene una duración de cerca de 38 minutos, por más grande que sea el disco. Este límite también implica que el vídeo más grande que puede editarse en línea es de menos de 19 minutos, ya que se necesita tanto un fichero de entrada como uno de salida.

A partir de la segunda versión de Windows 95, se introdujo el sistema de ficheros FAT-32, con sus direcciones de disco de 28 bits, y la versión de MS-DOS que era la base de Windows 95 se adaptó para manejar FAT-32. En este sistema, las particiones podían ser de en teoría de $2^{28} \times 2^{15}$ bytes, pero en realidad están limitadas a 2 TB (2048 GB) porque internamente el sistema lleva el control de los tamaños de las particiones en sectores de 512 bytes empleando un número de 32 bits, y $2^9 \times 2^{32}$ es 2 TB. En la Figura 6-33 se muestra el tamaño máximo de las particiones con diferentes tamaños de bloque para los tres tipos de FAT.

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

Figura 6-33. Tamaño máximo de las particiones con diferentes tamaños de bloque. Los cuadros vacíos representan combinaciones prohibidas.

Además de manejar discos más grandes, el sistema de ficheros FAT-32 tiene otras dos ventajas respecto a FAT-16. Primera, un disco de 8 GB que utiliza FAT-32 puede tener una sola partición. Si se utiliza FAT-16 tiene que haber cuatro particiones, las cuales se presentan al usuario de Windows como las unidades de disco lógicas C:, D:, E: y F:. Corresponde al usuario decidir qué ficheros colocará en qué discos, y llevar el control de dónde está cada cosa.

La otra ventaja de FAT-32 respecto a FAT-16 es que, para un tamaño de partición dado, puede utilizarse un tamaño de bloque más pequeño. Por ejemplo, con particiones de disco de 2 GB, FAT-16 tiene que utilizar bloques de 32 KB, pues de lo contrario no podría cubrir toda la partición con las 64K direcciones de disco de que dispone. En contraste, FAT-32 puede usar, por ejemplo, bloques de 4 KB en una partición de 2 GB. La ventaja de utilizar un tamaño de bloque más pequeño es que la mayoría de los ficheros ocupa mucho menos de 32 KB. Si el tamaño de bloque es de 32 KB, un fichero de 10 bytes ocupará 32 KB de espacio en disco. Si el fichero medio tiene, digamos, 8 KB, y se utilizan bloques de 32 KB, se desperdiciarán 3/4 partes del disco, lo cual no es una forma muy eficiente de utilizarlo. Con ficheros de 8 KB y bloques de 4 KB, no habrá desaprovechamiento del disco, pero el precio que se paga es más RAM ocupada por la FAT. Con bloques de 4 KB y particiones de disco de 2 GB, habrá 512K bloques, así que la FAT deberá tener 512K entradas en la memoria (que ocupan 2 MB de RAM).

MS-DOS utiliza la FAT para llevar el control de los bloques de disco libres. Cualquier bloque no asignado se marca con un código especial. Cuando MS-DOS necesita un nuevo bloque de disco, busca en la FAT una entrada que contenga ese código. Por tanto, no se requiere mapa de bits ni lista de bloques libres.

6.4.4 El sistema de ficheros de Windows 98

La versión original de Windows 95 utilizaba el sistema de ficheros de MS-DOS, incluyendo nombres de fichero de 8 + 3 caracteres y los sistemas de ficheros FAT-12 y FAT-16. A partir de la segunda versión de Windows 95 se permitieron nombres de fichero de más de 8 + 3 caracteres. Además, se introdujo la FAT-32, sobre todo para poder tener particiones de disco de más de 2 GB y discos de más de 8 GB, que ya habían salido a la venta. Tanto los nombres de fichero largos como la FAT-32 se utilizaron en Windows 98 de la misma forma que en la segunda versión de Windows 95. A continuación describiremos estas características del sistema de ficheros de Windows 98, que se han llevado también a Windows Me.

Puesto que los nombres de fichero largos son más emocionantes para los usuarios que la estructura de la FAT, los examinaremos primero. Una forma de introducir nombres de fichero largos habría sido inventar una nueva estructura de directorio. El problema con ese método es que, si Microsoft lo hubiera hecho, quienes todavía estaban en proceso de convertir Windows 3 a Windows 95 o Windows 98 no hubieran podido tener acceso a sus ficheros desde ambos sistemas. Se tomó una decisión política dentro de Microsoft de que los nombres creados utilizando Windows 98 debían ser también accesibles desde Windows 3 (para las máquinas de arranque doble). Esta restricción obligó a adoptar un diseño para manejar nombres de fichero largos, que fuera compatible con el viejo sistema de nombres 8 + 3 de MS-DOS. Puesto que tales restricciones de compatibilidad hacia atrás no son inusitadas en la industria de los ordenadores, vale la pena ver los pormenores de la forma en la que Microsoft logró su objetivo.

El efecto de esta decisión de ser compatible hacia atrás implicó que la estructura de directorios de Windows 98 tenía que ser compatible con la de MS-DOS. Como vimos, dicha estructura no es más que una lista de entradas de 32 bytes, como se muestra en la Figura 6-32. Este formato se tomó directamente de CP/M (que se escribió para el 8080), lo cual demuestra que las estructuras (obsoletas) pueden persistir durante mucho tiempo en el mundo de los ordenadores.

Sin embargo, ahora era posible utilizar los 10 bytes reservados de las entradas de la Figura 6-32, y eso fue lo que se hizo, como se aprecia en la Figura 6-34. Este cambio nada tiene que ver con los nombres largos, pero se utiliza en Windows 98, así que vale la pena entenderlo.

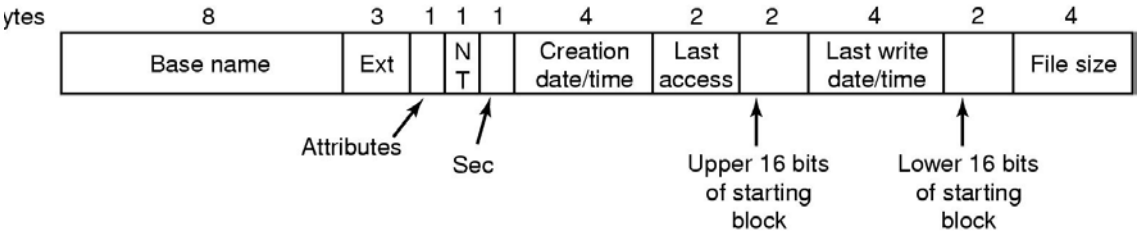

Figura 6-34. La entrada de directorio MS-DOS extendida empleada en Windows 98.

Los cambios consisten en la adición de cinco campos nuevos donde solían estar los 10 bytes desocupados. El campo *NT* sirve sobre todo para asegurar cierta compatibilidad con Windows NT, en el sentido de desplegar los nombres de fichero en el caso correcto (en MS-DOS, todos los nombres de fichero están en mayúsculas). El campo *Seg* resuelve el problema de que no es posible almacenar la hora del día en un campo de 16 bits: proporciona bits adicionales para que el nuevo ampo *Fecha/hora de creación* tenga una precisión de 10 ms. Otro campo nuevo es *Último acceso*, que almacena la fecha (pero no la hora) del último acceso al fichero. Por último, el cambio al sistema de ficheros FAT-32 implica que los números de bloque ahora son de 32 bits, así que se necesita un campo adicional de 16 bits para almacenar los 16 bits superiores del número de bloque inicial.

Ahora llegamos al corazón del sistema de ficheros Windows 98: cómo se representan los nombres de fichero largos de modo que sean compatibles con MS-DOS. La solución escogida consiste en asignar dos nombres a cada fichero: uno (potencialmente) largo (en Unicode, por compatibilidad con Windows NT) y un nombre 8 + 3 por compatibilidad con MS-DOS. Se puede tener acceso a los ficheros con cualquiera de los dos nombres. Cuando se crea un fichero cuyo nombre no obedece a las reglas de MS-DOS (longitud 8 + 3, nada de Unicode, conjunto de caracteres limitado, nada de espacios, etcétera), Windows 98 inventa un nombre MS-DOS para el fichero según cierto algoritmo. La idea básica es tomar los primeros seis caracteres del nombre convertirlos a mayúsculas, si es necesario, y añadir ~1 para formar el nombre base. Si ese nombre ya existe, se utiliza el sufijo ~2, y así de forma sucesiva. Además, se eliminan los espacios y puntos adicionales y ciertos caracteres especiales se convierten en caracteres de subrayado. Por ejemplo, a un fichero de nombre *Lista de libros comprados* se le asigna el nombre MS-DOS *LISTAD~1*. Si después se crea un fichero con el nombre Lista de libros vendidos, se le asignará el nombre MS-DOS *LISTAD~2*, y así en forma sucesiva.

Todo fichero tiene un nombre MS-DOS que se almacena empleando el formato de directorio de la Figura 6-34. Si el fichero también tiene un nombre largo, ese nombre se almacena en una o más entradas de directorio que preceden inmediatamente al nombre de fichero MS-DOS. Cada entrada de nombre largo contiene hasta 13 caracteres (Unicode). Las entradas se almacenan en orden inverso, con el principio del nombre justo delante de la entrada MS-DOS y los fragmentos subsiguientes antes de ella. El formato de cada entrada de nombre largo se muestra en la Figura 6-35.

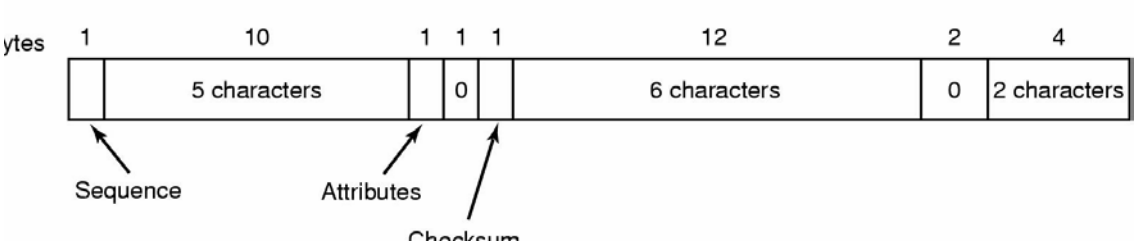

Figura 6-35. Entrada de (parte de) un nombre de fichero largo en Windows 98.

Una pregunta obvia es: "¿Cómo sabe Windows 98 si una entrada de directorio contiene un nombre de fichero MS-DOS o un (fragmento de un) nombre de fichero largo?" La respuesta está en el campo *Atributos*. En el caso de una entrada de nombre largo, este campo tiene el valor 0x0F, que representa una combinación imposible de atributos. Los programas MS-DOS antiguos que lean el directorio tan solo harán caso omiso de esa entrada, por considerarla no válida. ¡Si supieran! Los fragmentos del nombre se arman en forma consecutiva con base en el primer byte de la entrada. La última parte del nombre largo (la primera entrada de la secuencia) se marca sumando 64 al número consecutivo. Puesto que sólo se utilizan 6 bits para el número consecutivo, en teoría el tamaño máximo de los nombres de fichero es $63 \times 13 = 819$ caracteres. De hecho, los nombres están limitados a 260 caracteres por razones históricas.

Cada entrada de nombre largo contiene un campo de *Suma de comprobación* para evitar el siguiente problema. Primero, un programa de Windows 98 crea un fichero con nombre largo. Luego, se rearranca el ordenador para ejecutar Windows 3 o MS-DOS. Después, un programa viejo en ese entorno elimina el nombre de fichero MS-DOS del directorio pero no elimina el nombre largo que le precede (porque no sabe que existe). Por último, algún programa crea un fichero nuevo que reutiliza la entrada de directorio recién desocupada. Ahora tenemos una secuencia válida de entradas de nombre largo, justo antes de una entrada de fichero MS-DOS que nada tiene que ver con el nombre largo. El campo *Suma de comprobación* permite a Windows detectar esta situación, verificando que el nombre de fichero MS-DOS que sigue a un nombre largo en verdad corresponde a él. Desde luego, como sólo se utiliza un byte hay una probabilidad de 1/256 de que Windows 98 non se dé cuenta de las sustitución de ficheros.

Para ver un ejemplo de cómo funcionan los nombres largos, consideremos el ejemplo de la Figura 6-36. Aquí tenemos un fichero llamado *Carta que les escribo a mis queridos hijitos*. Con 44 caracteres, ciertamente cumple con los requisitos para ser un nombre de fichero largo. El nombre MS-DOS que se construye a partir de él es *CARTAQ~1* y se almacena en la última entrada.

La estructura de directorio incorpora cierta redundancia para ayudar a detectar problemas en caso de que un programa antiguo de Windows 3 haya hecho cosas que no debía con el directorio. El byte consecutivo en realidad no se necesita porque el byte 0x40 marca la primera entrada, pero es un ejemplo de redundancia incluida de forma deliberada. Además, el campo *Inferiores* de la Figura 6-36 (la mitad inferior del número de clúster inicial) es 0 en todas las entradas salvo la última, también para evitar que los programas antiguos lo interpreten mal y den al trate con el sistema de ficheros. El byte *NT* de la Figura 6-36 se utiliza en NT y no se toma en cuenta en Windows 98. El byte A contiene los atributos.

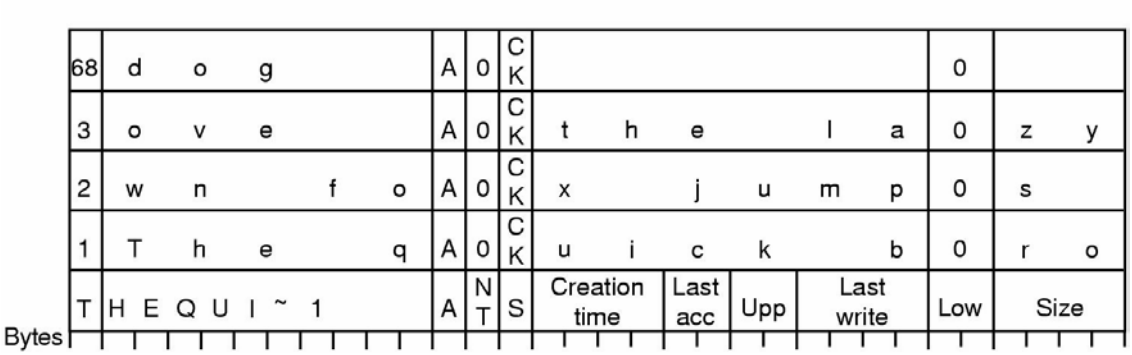

Figura 6-36. Ejemplo de cómo se almacena un nombre largo en Windows 98.

Desde el punto de vista conceptual, la implementación del sistema de ficheros FAT-32 es similar a la del sistema de ficheros FAT-16. Sin embargo, en lugar de una tabla de 65536 entradas, hay tantas entradas como se necesiten para cubrir la parte del disco que contiene datos. Si se utiliza el primer millón de bloques, desde una perspectiva conceptual la tabla tiene un millón de entradas. Para evitar la necesidad de tenerlas todas en memoria a la vez, Windows 98 mantiene una "ventana" que ve hacia la tabla, y sólo mantiene una parte de ella en la memoria en un momento dado.

6.4.5 El sistema de ficheros de UNIX V7

Aún las primeras versiones de UNIX tenían un sistema de ficheros multiusuario relativamente elaborado, pues se derivó de MULTICS. A continuación trataremos el sitema de ficheros V7, utilizado en la PDP-11 y que hizo famoso a UNIX. Examinaremos versiones modernas en el capítulo 10.

El sistema de ficheros tiene la forma de un árbol que nace en el directorio raíz, con la adición de enlaces para formar un grafo acíclico dirigido (DAG). Los nombres de fichero tienen hasta 14 caracteres y pueden contener cualquier carácter ASCII con excepción de / (porque és es el separador de los componentes de un camino) y NUL (porque sirve para rellenar los nombres de menos de 14 caracteres). NUL tiene el valor numérico 0.

Una entrada de directorio UNIX contiene una entrada para cada fichero de ese directorio. Las entradas son simples en extremo porque UNIX utiliza el esquema de i-nodos que ilustramos en la Figura 6-15. Una entrada de directorio contiene solamente dos campos: el nombre del fichero (14 bytes) y el número del i-nodo correspondiente a ese fichero (2 bytes), como se muestra en la Figura 6-37. Estos parámetros limitan el número de ficheros por sistema de ficheros a 64K.

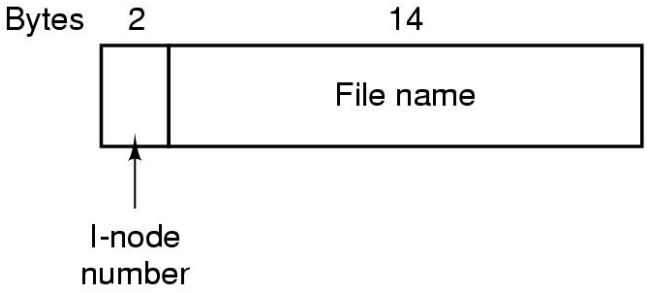

Figura 6-37. Entrada de directorio en UNIX V7.

Al igual que el i-nodo de la Figura 6-15, los i-nodos de UNIX contienen algunos atributos. Éstos incluyen el tamaño del fichero, la hora de creación, la del último acceso y la de la última modificación), propietario, grupo, información de protección y un contador del número de entradas de directorio que apuntan al i-nodo. Éste último campo es necesario para los enlaces. Cada vez que se crea un enlace nuevo con un i-nodo, se incrementa el contador en el i-nodo. Cuando se elimina un enlace, el contador se decrementa. Cuando el contador llega a 0, el i-nodo se recicla y los bloques de disco se colocan en la lista libre.

El control de los bloques de disco se lleva utilizando una generalización de la Figura 6-15 para manejar ficheros muy grandes. Las primeras 10 direcciones de disco se almacenan en el mismo i-nodo, así que en el caso de ficheros pequeños toda la información necesaria está justo en el i-nodo, que pasa del disco a la memoria principal cuando se abre el fichero. Si los ficheros son algo más grandes, una de las direcciones que están en el i-nodo es la dirección de un bloque de disco llamado **bloque indirecto simple** o **bloque indirecto** a secas. Este bloque contiene más direcciones de disco. Si todavía no son suficientes, otra dirección en el i-nodo, denominada **bloque indirecto doble**, contiene la dirección de un bloque que contiene una lista de bloques indirectos. Si ni siquiera esto es suficiente, puede utilizarse también un **bloque indirecto triple**. El panorama completo se presenta en la Figura 6-38.

Cuando se abre un fichero, el sistema de ficheros debe tomar el nombre del fichero proporcionado y localizar sus bloques de disco. Consideremos cómo se busca el nombre de camino */usr/ast/correo*. Utilizaremos UNIX como ejemplo, pero el algoritmo es básicamente el mismo en todos los sistemas de directorios jerárquicos. Primero el sistema de ficheros localiza el directorio raíz. En UNIX, el i-nodo del directorio raíz está en un lugar fijo del disco. A partir de este i-nodo, se localiza el directorio raíz, que puede estar en cualquier lugar del disco, pero digamos que en este caso está en el bloque 1.

Luego se busca el primer componente del camino, *usr*, en el directorio raíz para hallar el número de i-nodo del fichero */usr*. Localizar un i-nodo teniendo su número es fácil, porque todos tienen una posición fija en el disco. A partir de ese i-nodo, el sistema localiza el directorio de */usr* y busca en él el siguiente componente, *ast*. Al encontrar la entrada de *ast*, tendrá el i-nodo del directorio */usr/ast*. A partir de ese i-nodo se encuentra el directorio en sí y se busca correo. Luego se lee el i-nodo de ese fichero y se coloca en la memoria, donde se mantiene hasta que se cierre el fichero. El proceso de búsqueda se ilustra en la Figura 6-39.

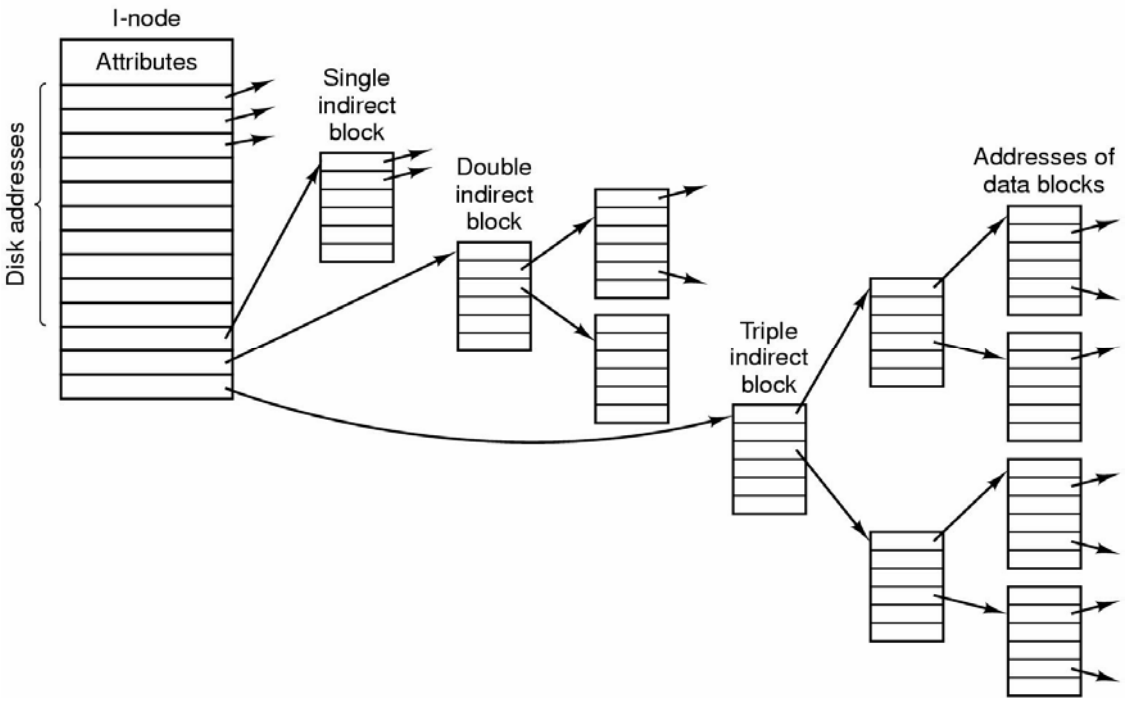

Figura 6-38. i-nodo de UNIX.

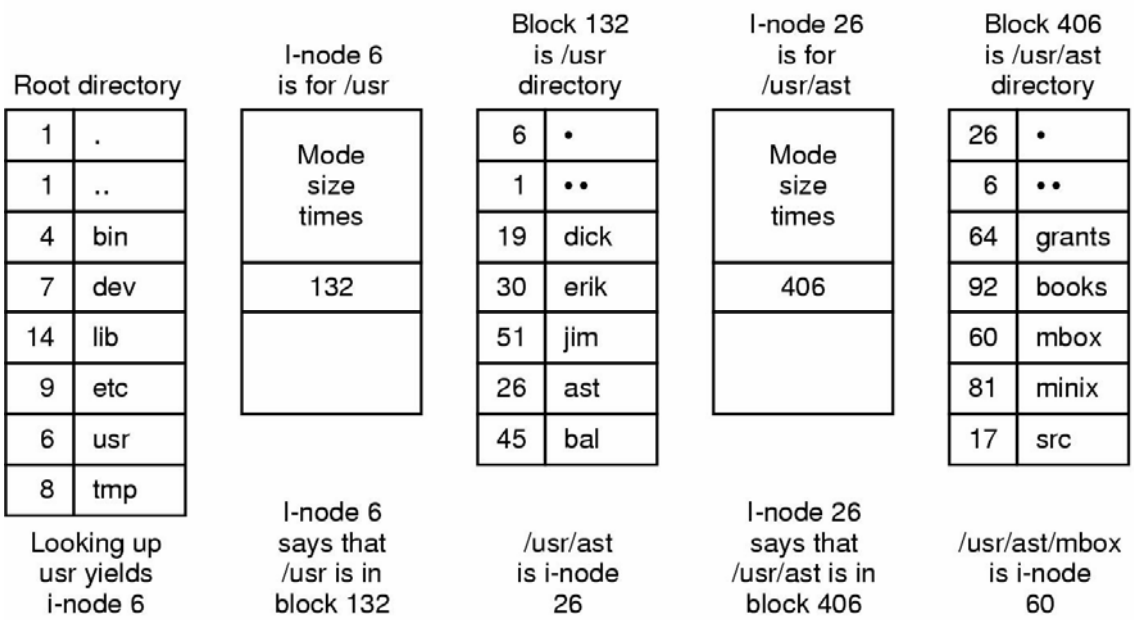

Figura 6-39. Pasos para buscar */usr/ast/correo*.

Los nombres de camino relativos se buscan igual que los absolutos, sólo que partiendo del directorio de trabajo en lugar del directorio raíz. Todo directorio tiene entradas para . y .., que se colocan ahí cuando se crea el directorio. La entrada . tiene el número de i-nodo del directorio actual, y la entrada .. tiene el número de i-nodo del directorio padre. Así, un procedimiento que busca *../luis/prog.c* simplemente consulta .. en el directorio de trabajo, halla el número de i-nodo del directorio padre y busca *luis* en ese directorio. No se necesita ningún mecanismo especial para manejar estos nombres. En lo que concierne al sistema de directorios, son sólo cadenas ASCII ordinarias, como cualquier otro nombre.

6.5 INVESTIGACIÓN SOBRE SISTEMAS DE FICHEROS

Los sistemas de ficheros han atraído siempre más investigadores que otras partes del sistema operativo, y sigue siendo así. Algunas de las investigaciones tienen que ver con la estructura de los sistemas de ficheros. Los sistemas de ficheros con estructura de registro y temas afines son populares (Matthews y otros, 1997 y Wang y otros, 1999). El disco lógico divide el sistema de ficheros en dos capas distintas: el sistema de ficheros y el sistema de disco (De Jorge y otros, 1993). La construcción de un sistema de ficheros a partir de capas apilables también es tema de investigación (Heidemann y Popek, 1994).

Los sistemas de ficheros extensibles son análogos a los kernels extensibles que vimos en el capítulo 1. Con ellos es posible añadir nuevas funciones al sistema de ficheros sin tener que rediseñarlo desde cero (Karpovich y otros, 1994 y Khalidi y Nelson, 1993).

Otro tema de investigación que ha alcanzado cierta popularidad es la medición del contenido y el uso de los sistemas de ficheros. Se ha medido la distribución de tamaños de los ficheros, la longevidad de los ficheros, el acceso equitativo a todos los ficheros, la comparación entre lecturas y escrituras, y muchos otros parámetros (Douceur y Bolosky, 1999; Gill y otros, 1994; Roselli y Lorch, 2000, y Vogels, 1999).

Otros investigadores han examinado el rendimiento de los sistemas de ficheros y la forma de mejorarlo utilizando preextracción, cachés, menos copiado y otras técnicas. Normalmente, estos investigadores realizan mediciones, averiguan dón de están los cuellos de botella, eliminan por lo menos uno de ellos y luego realizan las mediciones en el sistema mejorado para validar sus resultados (Cao y otros, 1995; Pai y otros, 2000, y Patterson y otros, 1995).

Un tema en el que pocos piensan hasta que sucede un desastre es el de los backups y la recuperación de los sistemas de ficheros. Aquí también han surgido algunas ideas nuevas acerca de cómo hacer mejor las cosas (Chen y otros, 1996; Devarakonda y otros, 1996, y Hutchinson y otros, 1999). Un tanto relacionada con este tema está la cuestión de qué hacer cuando un usuario borra un fichero: ¿eliminarlo o ocultarlo? El sistema de ficheros Elephant, por ejemplo, nunca olvida (Santry y otros, 1999a y Santry y otros, 1999b).

6.6 RESUMEN

Visto desde fuera, un sistema de ficheros es una colección de ficheros y directorios, además de operaciones con ellos. Los ficheros pueden leerse y escribirse, los directorios pueden crearse y destruirse, y los ficheros pueden cambiarse de un directorio a otro. Casi todos los sistemas de ficheros modernos manejan un sistema de directorios jerárquico en el que los directorios pueden tener subdirectorios y éstos pueden tener subdirectorios, *ad infinitum*.

Visto desde dentro, un sistema de ficheros es muy diferente. Los diseñadores del sistema de ficheros tienen que decidir cómo se asigna el espacio de almacenamiento y cómo se mantiene el sistema al tanto de qué bloque corresponde a qué fichero. Entre las posibilidades están los ficheros contiguos, las listas enlazadas, las tablas de asignación de ficheros y los i-nodos. Los distintos sistemas tienen diferentes estructuras de directorio. Los atributos pueden colocarse en los directorios o en otro lado (por ejemplo en un i-nodo). El espacio de disco puede administrarse utilizando listas libres o mapas de bits. La fiabilidad del sistema de ficheros aumenta si se realizan volcados incrementales y si se utiliza un programa para reparar sistemas de ficheros dañados. El rendimiento de los sistemas de ficheros es importante y hay varias formas de mejorarlo, entre ellas la utilización de cachés, la lectura adelantada y la colocación cuidadosa de los bloques de un fichero cercanos entre sí en el disco. Los sistemas de ficheros con estructura de registro también mejoran el rendimiento porque realizan las escrituras en unidades grandes.

Como ejemplos de sistemas de ficheros podemos citar ISO 9660, CP/M, MS-DOS, Windows 98 y UNIX. Hay muchas diferencias entre ellos, que incluyen la forma de llevar el control de qué bloques corresponden a qué ficheros, la estructura de directorios y la administración del espacio libre en disco.