3

ADMINISTRACIÓN DE MEMORIA

La memoria principal (RAM) es un importante recurso que debe administrarse con cuidado. Aunque actualmente una computadora doméstica promedio tiene 10,000 veces más memoria que la IBM 7094, la computadora más grande en el mundo a principios de la década de 1960, los programas están creciendo con más rapidez que las memorias. Parafraseando la ley de Parkinson diría que "los programas se expanden para llenar la memoria disponible para contenerlos". En este capítulo estudiaremos la forma en que los sistemas operativos crean abstracciones de la memoria y cómo las administran.

Lo que todo programador quisiera es una memoria privada, de tamaño y rapidez infinitas, que también sea no volátil, es decir, que no pierda su contenido cuando se desconecta de la potencia eléctrica. Y ya que estamos en ello, ¿por qué no hacerla barata también? Por desgracia, la tecnología no proporciona tales memorias en estos momentos. Tal vez usted descubra cómo hacerlo.

¿Cuál es la segunda opción? A través de los años se ha elaborado el concepto de **jerarquía de memoria**, de acuerdo con el cual, las computadoras tienen unos cuantos megabytes de memoria caché, muy rápida, costosa y volátil, unos cuantos gigabytes de memoria principal, de mediana velocidad, a precio mediano y volátil, unos cuantos terabytes de almacenamiento en disco lento, económico y no volátil, y el almacenamiento removible, como los DVDs y las memorias USB. El trabajo del sistema operativo es abstraer esta jerarquía en un modelo útil y después administrarla.

La parte del sistema operativo que administra (parte de) la jerarquía de memoria se conoce como **administrador de memoria**. Su trabajo es administrar la memoria con eficiencia: llevar el registro de cuáles partes de la memoria están en uso, asignar memoria a los procesos cuando la necesiten y desasignarla cuando terminen.

En este capítulo investigaremos varios esquemas distintos de administración de memoria, que varían desde muy simples hasta muy sofisticados. Como generalmente el hardware es el que se encarga de administrar el nivel más bajo de memoria caché, en este capítulo nos concentraremos en el modelo del programador de la memoria principal y en cómo se puede administrar bien. Las abstracciones y la administración del almacenamiento permanente, el disco, son el tema del siguiente capítulo. Analizaremos primero los esquemas más simples posibles y después progresaremos de manera gradual hacia esquemas cada vez más elaborados.

3.1 SIN ABSTRACCIÓN DE MEMORIA

La abstracción más simple de memoria es ninguna abstracción. Las primeras computadoras mainframe (antes de 1960), las primeras minicomputadoras (antes de 1970) y las primeras computadoras personales (antes de 1980) no tenían abstracción de memoria. Cada programa veía simplemente la memoria física. Cuando un programa ejecutaba una instrucción como

```
MOV REGISTRO1, 1000
```

la computadora sólo movía el contenido de la ubicación de memoria física 1000 a *REGISTRO1*. Así, el modelo de programación que se presentaba al programador era simplemente la memoria física, un conjunto de direcciones desde 0 hasta cierto valor máximo, en donde cada dirección correspondía a una celda que contenía cierto número de bits, comúnmente ocho.

Bajo estas condiciones, no era posible tener dos programas ejecutándose en memoria al mismo tiempo. Si el primer programa escribía un nuevo valor en, por ejemplo, la ubicación 2000, esto borraría cualquier valor que el segundo programa estuviera almacenando ahí. Ambos programas fallarían de inmediato.

Incluso cuando el modelo de memoria consiste en sólo la memoria física hay varias opciones posibles. En la figura 3-1 se muestran tres variaciones. El sistema operativo puede estar en la parte inferior de la memoria en la RAM (*Random Access Memory*, Memoria de acceso aleatorio), como se muestra en la figura 3-1(a), puede estar en la ROM (*Read Only Memory*, Memoria de sólo lectura) en la parte superior de la memoria, como se muestra en la figura 3-1(b), o los controladores de dispositivos pueden estar en la parte superior de la memoria en una ROM y el resto del sistema en RAM más abajo, como se muestra en la figura 3-1(c). El primer modelo se utilizó antes en las mainframes y minicomputadoras, pero actualmente casi no se emplea. El segundo modelo se utiliza en algunas computadoras de bolsillo y sistemas integrados. El tercer modelo fue utilizado por las primeras computadoras personales (por ejemplo, las que ejecutaban MS-DOS), donde la porción del sistema en la ROM se conoce como **BIOS** (*Basic Input Output System*, Sistema básico de entrada y salida). Los modelos (a) y (c) tienen la desventaja de que un error en el programa de usuario puede borrar el sistema operativo, posiblemente con resultados desastrosos (la información del disco podría quedar ininteligible).

Cuando el sistema se organiza de esta forma, por lo general se puede ejecutar sólo un proceso a la vez. Tan pronto como el usuario teclea un comando, el sistema operativo copia el programa solicitado del disco a la memoria y lo ejecuta. Cuando termina el proceso, el sistema operativo mues-

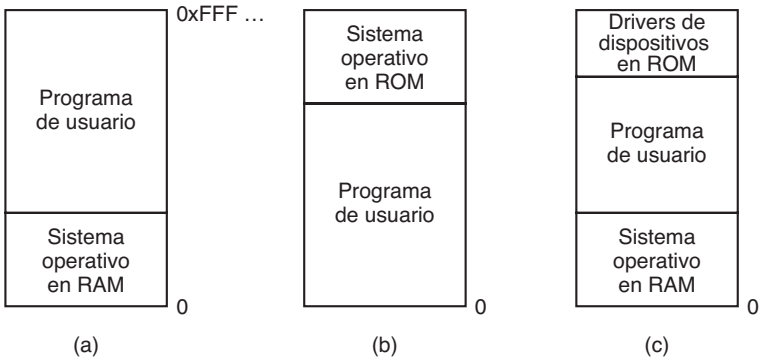

Figura 3-1. Tres formas simples de organizar la memoria con un sistema operativo y un proceso de usuario. También existen otras posibilidades.

tra un carácter indicador de comando y espera un nuevo comando. Cuando recibe el comando, carga un nuevo programa en memoria, sobrescribiendo el primero.

Una forma de obtener cierto grado de paralelismo en un sistema, sin abstracción de memoria, es programar con múltiples hilos. Como se supone que todos los hilos en un proceso ven la misma imagen de memoria, el hecho de que se vean obligados a hacerlo no es un problema. Aunque esta idea funciona, es de uso limitado ya que lo que las personas desean a menudo es que los programas *no relacionados* se ejecuten al mismo tiempo, algo que la abstracción de los hilos no provee. Además, es muy poco probable que un sistema tan primitivo como para no proporcionar una abstracción de memoria proporcione una abstracción de hilos.

Ejecución de múltiple programas sin una abstracción de memoria

No obstante, aun sin abstracción de memoria es posible ejecutar varios programas al mismo tiempo. Lo que el sistema operativo debe hacer es guardar todo el contenido de la memoria en un archivo en disco, para después traer y ejecutar el siguiente programa. Mientras sólo haya un programa a la vez en la memoria no hay conflictos. Este concepto, el intercambio, se analiza a continuación.

Con la adición de cierto hardware especial es posible ejecutar múltiples programas concurrentemente, aun sin intercambio. Los primeros modelos de la IBM 360 resolvieron el problema de la siguiente manera: la memoria estaba dividida en bloques de 2 KB y a cada uno se le asignaba una llave de protección de 4 bits, guardada en registros especiales dentro de la CPU. Un equipo con una memoria de 1 MB sólo necesitaba 512 de estos registros de 4 bits para totalizar 256 bytes de almacenamiento de la llave. El registro **PSW** (*Program Status Word*, Palabra de estado del programa) también contenía una llave de 4 bits. El hardware de la 360 controlaba mediante un trap cualquier intento por parte de un proceso en ejecución de acceder a la memoria con un código de protección distinto del de la llave del PSW. Como sólo el sistema operativo podía modificar las llaves de protección, los procesos de usuario fueron controlados para que no interfirieran unos con otros, ni con el mismo sistema operativo.

Sin embargo, esta solución tenía una gran desventaja, que se ilustra en la figura 3-2. Como muestran las figuras 3-2(a) y (b), se tienen dos programas, cada uno con un tamaño de 16 KB. El primero está sombreado para indicar que tiene una llave de memoria diferente a la del segundo y empieza saltando a la dirección 24, que contiene una instrucción `MOV`; el segundo, saltando a la dirección 28, que contiene una instrucción `CMP`. Las instrucciones que no son relevantes para este análisis no se muestran. Cuando los dos programas se cargan consecutivamente en la memoria, empezando en la dirección 0, tenemos la situación de la figura 3-2(c). Para este ejemplo, suponemos que el sistema operativo está en la parte alta de la memoria y no se muestra.

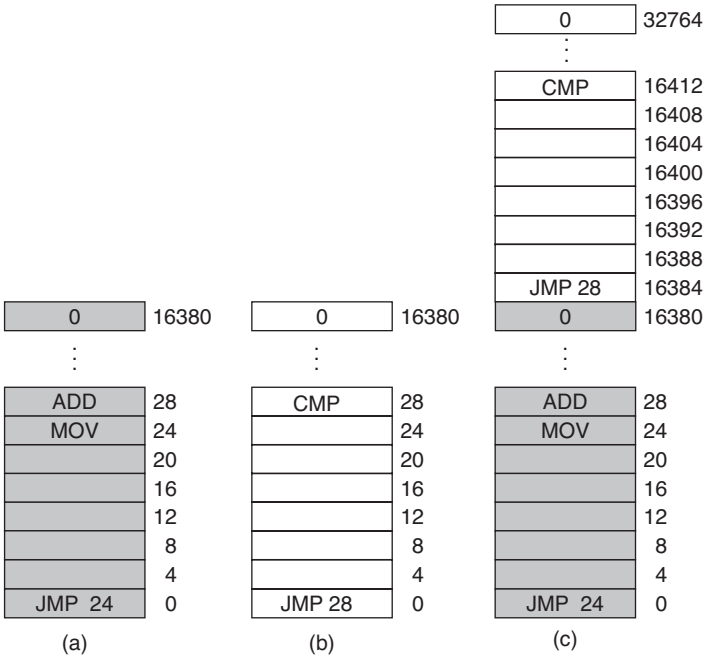

Figura 3-2. Ilustración del problema de reubicación. (a) Un programa de 16 KB. (b) Otro programa de 16 KB. (c) Los dos programas cargados consecutivamente en la memoria.

Una vez que los programas se cargan se pueden ejecutar. Como tienen distintas llaves de memoria, ninguno de los dos puede dañar al otro. Pero el problema es de una naturaleza distinta. Cuando se inicia el primer programa, ejecuta la instrucción `JMP 24`, que salta a la instrucción, como se espera. Este programa funciona de manera normal.

Sin embargo, después de que el primer programa se ha ejecutado el tiempo suficiente, el sistema operativo tal vez decida ejecutar el segundo programa, que se carga encima del primero, en la dirección 16,384. La primera instrucción ejecutada es `JMP 28`, que salta a la instrucción `ADD` en el primer programa, y no a la instrucción `CMP` a la que se supone debe saltar. Es muy probable que el programa falle en menos de 1 segundo.

El problema central aquí es que los dos programas hacen referencia a la memoria física absoluta. Eso, definitivamente, no es lo que queremos; deseamos que cada programa haga referencia a un conjunto privado de direcciones locales para él. En breve le mostraremos cómo se logra esto. Lo que la IBM 360 hacía como solución para salir del paso era modificar el segundo programa al instante, a medida que se cargaba en la memoria, usando una técnica conocida como **reubicación estática**. Esta técnica funcionaba así: cuando se cargaba un programa en la dirección 16,384, se sumaba el valor constante 16,384 a todas las direcciones del programa durante el proceso de carga. Aunque este mecanismo funciona si se lleva a cabo en la forma correcta, no es una solución muy general y reduce la velocidad de la carga. Lo que es más, se requiere información adicional en todos los programas ejecutables para indicar cuáles palabras contienen direcciones (reubicables) y cuáles no. Después de todo, el "28" en la figura 3-2(b) tiene que reubicarse, pero una instrucción como

```
MOV REGISTRO1, 28
```

que mueve el número 28 a *REGISTRO1* no se debe reubicar. El cargador necesita cierta forma de saber qué es una dirección y qué es una constante.

Por último, como recalcamos en el capítulo 1, la historia tiende a repetirse en el mundo de las computadoras. Mientras que el direccionamiento directo de la memoria física está muy lejos de poder aplicarse en las mainframes, minicomputadoras, computadoras de escritorio y notebooks, la falta de una abstracción de memoria sigue siendo común en los sistemas integrados y de tarjeta inteligente. En la actualidad, los dispositivos como las radios, las lavadoras y los hornos de microondas están llenos de software (en ROM), y en la mayoría de los casos el software direcciona memoria absoluta. Esto funciona debido a que todos los programas se conocen de antemano, y los usuarios no tienen la libertad de ejecutar su propio software en su tostador.

Mientras que los sistemas integrados de alta tecnología (como los teléfonos celulares) tienen sistemas operativos elaborados, los más simples no. En algunos casos hay un sistema operativo, pero es sólo una biblioteca ligada con el programa de aplicación y proporciona llamadas al sistema para realizar operaciones de E/S y otras tareas comunes. El popular sistema operativo **e-cos** es un ejemplo común de un sistema operativo como biblioteca.

3.2 UNA ABSTRACCIÓN DE MEMORIA: ESPACIOS DE DIRECCIONES

Con todo, exponer la memoria física a los procesos tiene varias desventajas. En primer lugar, si los programas de usuario pueden direccionar cada byte de memoria, pueden estropear el sistema operativo con facilidad, ya sea intencional o accidentalmente, con lo cual el sistema se detendría en forma súbita (a menos que haya hardware especial como el esquema de bloqueo y llaves de la IBM 360). Este problema existe aun cuando sólo haya un programa de usuario (aplicación) en ejecución. En segundo lugar, con este modelo es difícil tener varios programas en ejecución a la vez (tomando turnos, si sólo hay una CPU). En las computadoras personales es común tener varios programas abiertos a la vez (un procesador de palabras, un programa de correo electrónico y un navegador Web, donde uno de ellos tiene el enfoque actual, pero los demás se reactivan con el clic de un ratón). Como esta situación es difícil de lograr cuando no hay una abstracción de la memoria física, se tuvo que hacer algo.

3.2.1 La noción de un espacio de direcciones

Hay que resolver dos problemas para permitir que haya varias aplicaciones en memoria al mismo tiempo sin que interfieran entre sí: protección y reubicación. Ya analizamos una solución primitiva al primer problema en la IBM 360: etiquetar trozos de memoria con una llave de protección y comparar la llave del proceso en ejecución con la de cada palabra de memoria obtenida por la CPU. Sin embargo, este método por sí solo no resuelve el segundo problema, aunque se puede resolver mediante la reubicación de los programas al momento de cargarlos, pero ésta es una solución lenta y complicada.

Una mejor solución es inventar una nueva abstracción para la memoria: el espacio de direcciones. Así como el concepto del proceso crea un tipo de CPU abstracta para ejecutar programas, el espacio de direcciones crea un tipo de memoria abstracta para que los programas vivan ahí. Un **espacio de direcciones** (*address space*) es el conjunto de direcciones que puede utilizar un proceso para direccionar la memoria. Cada proceso tiene su propio espacio de direcciones, independiente de los que pertenecen a otros procesos (excepto en ciertas circunstancias especiales en donde los procesos desean compartir sus espacios de direcciones).

El concepto de un espacio de direcciones es muy general y ocurre en muchos contextos. Considere los números telefónicos. En los Estados Unidos y en muchos otros países, un número de teléfono local es comúnmente un número de 7 dígitos. En consecuencia, el espacio de direcciones para los números telefónicos varía desde 0,000,000 hasta 9,999,999, aunque algunos números, como los que empiezan con 000, no se utilizan. Con el crecimiento de los teléfonos celulares, módems y máquinas de fax, este espacio se está volviendo demasiado pequeño, en cuyo caso habrá qué utilizar más dígitos. El espacio de direcciones para los puertos de E/S en el Pentium varía desde 0 hasta 16383. Las direcciones IPv4 son números de 32 bits, por lo que su espacio de direcciones varía desde 0 hasta $2^{32} - 1$ (de nuevo, con ciertos números reservados).

Los espacios de direcciones no tienen que ser numéricos. El conjunto de dominios *.com* de Internet es también un espacio de direcciones. Este espacio de direcciones consiste de todas las cadenas de longitud de 2 a 63 caracteres que se puedan formar utilizando letras, números y guiones cortos, seguidas de *.com*. Para estos momentos usted debe de hacerse hecho ya la idea. Es bastante simple.

Algo un poco más difícil es proporcionar a cada programa su propio espacio de direcciones, de manera que la dirección 28 en un programa indique una ubicación física distinta de la dirección 28 en otro programa. A continuación analizaremos una forma simple que solía ser común, pero ha caído en desuso debido a la habilidad de poner esquemas mucho más complicados (y mejores) en los chips de CPU modernos.

Registros base y límite

La solución sencilla utiliza una versión muy simple de la **reubicación dinámica**. Lo que hace es asociar el espacio de direcciones de cada proceso sobre una parte distinta de la memoria física, de una manera simple. La solución clásica, que se utilizaba en máquinas desde la CDC 6600 (la primera supercomputadora del mundo) hasta el Intel 8088 (el corazón de la IBM PC original), es equipar cada CPU con dos registros de hardware especiales, conocidos comúnmente como los registros

base y **límite**. Cuando se utilizan estos registros, los programas se cargan en ubicaciones consecutivas de memoria en donde haya espacio y sin reubicación durante la carga, como se muestra en la figura 3-2(c). Cuando se ejecuta un proceso, el registro base se carga con la dirección física donde empieza el programa en memoria y el registro límite se carga con la longitud del programa. En la figura 3-2(c), los valores base y límite que se cargarían en estos registros de hardware al ejecutar el primer programa son 0 y 16,384, respectivamente. Los valores utilizados cuando se ejecuta el segundo programa son 16,384 y 16,384, respectivamente. Si se cargara un tercer programa de 16 KB directamente por encima del segundo y se ejecutara, los registros base y límite serían 32,768 y 16,384.

Cada vez que un proceso hace referencia a la memoria, ya sea para obtener una instrucción de memoria, para leer o escribir una palabra de datos, el hardware de la CPU suma de manera automática el valor base a la dirección generada por el proceso antes de enviar la dirección al bus de memoria. Al mismo tiempo comprueba si la dirección ofrecida es igual o mayor que el valor resultante de sumar los valores de los registros límite y base, en cuyo caso se genera un fallo y se aborta el acceso. Por ende, en el caso de la primera instrucción del segundo programa en la figura 3-2(c), el proceso ejecuta una instrucción

```
JMP 28
```

pero el hardware la trata como si fuera

```
JMP 16412
```

por lo que llega a la instrucción `CMP`, como se esperaba. Los valores de los registros base y límite durante la ejecución del segundo programa de la figura 3-2(c) se muestran en la figura 3-3.

El uso de registros base y límite es una manera fácil de proporcionar a cada proceso su propio espacio de direcciones privado, ya que a cada dirección de memoria que se genera en forma automática se le suma el contenido del registro base antes de enviarla a memoria. En muchas implementaciones, los registros base y límite están protegidos de tal forma que sólo el sistema operativo puede modificarlos. Éste fue el caso en la CDC 6600, pero no el del Intel 8088, que ni siquiera tenía el registro límite. Sin embargo, tenía varios registros base, lo cual permitía que se reubicaran texto y datos de manera independiente, pero no ofrecía una protección contra las referencias a memoria fuera de rango.

Una desventaja de la reubicación usando los registros base y límite es la necesidad de realizar una suma y una comparación en cada referencia a memoria. Las comparaciones se pueden hacer con rapidez, pero las sumas son lentas debido al tiempo de propagación del acarreo, a menos que se utilicen circuitos especiales de suma.

3.2.2 Intercambio

Si la memoria física de la computadora es lo bastante grande como para contener todos los procesos, los esquemas descritos hasta ahora funcionarán en forma más o menos correcta. Pero en la práctica, la cantidad total de RAM que requieren todos los procesos es a menudo mucho mayor de lo que puede acomodarse en memoria. En un sistema Windows o Linux común, se pueden iniciar

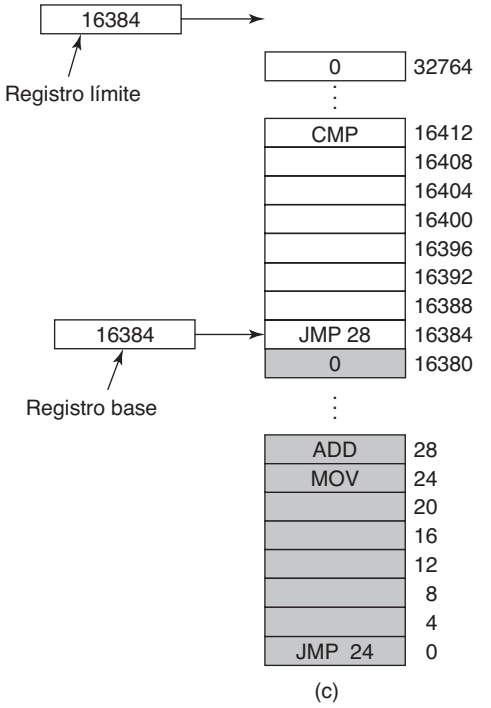

Figura 3-3. Se pueden utilizar registros base y límite para dar a cada proceso un espacio de direcciones separado.

entre 40 y 60 procesos o más cada vez que se inicia la computadora. Por ejemplo, cuando se instala una aplicación Windows, a menudo emite comandos de manera que en los inicios subsiguientes del sistema se inicie un proceso que no haga nada, excepto buscar actualizaciones para la aplicación. Dicho proceso puede ocupar fácilmente de 5 a 10 MB de memoria. Otros procesos en segundo plano comprueban el correo entrante, las conexiones de red entrantes y muchas otras cosas. Y todo esto se hace antes de que se inicie el primer programa de usuario. Hoy en día, los programas de aplicaciones de usuario serios pueden ejecutarse ocupando fácilmente desde 50 a 200 MB y más. En consecuencia, para mantener todos los procesos en memoria todo el tiempo se requiere una gran cantidad de memoria y no puede hacerse si no hay memoria suficiente.

A través de los años se han desarrollado dos esquemas generales para lidiar con la sobrecarga de memoria. La estrategia más simple, conocida como **intercambio**, consiste en llevar cada proceso completo a memoria, ejecutarlo durante cierto tiempo y después regresarlo al disco. Los procesos inactivos mayormente son almacenados en disco, de tal manera que no ocupan memoria cuando no se están ejecutando (aunque algunos de ellos se despiertan periódicamente para realizar su trabajo y después vuelven a quedar inactivos). La otra estrategia, conocida como **memoria virtual**, permite que los programas se ejecuten incluso cuando sólo se encuentran en forma parcial en la memoria. A continuación estudiaremos el intercambio; en la sección 3.3 examinaremos la memoria virtual.

La operación de un sistema de intercambio se ilustra en la figura 3-4. Al principio, sólo el proceso *A* está en la memoria. Después los procesos *B* y *C* se crean o se intercambian desde disco. En la figura 3-4(d), *A* se intercambia al disco. Después llega *D* y sale *B*. Por último, *A* entra de nuevo. Como *A* está ahora en una ubicación distinta, las direcciones que contiene se deben reubicar, ya sea mediante software cuando se intercambia o (más probablemente) mediante hardware durante la ejecución del programa. Por ejemplo, los registros base y límite funcionarían bien en este caso.

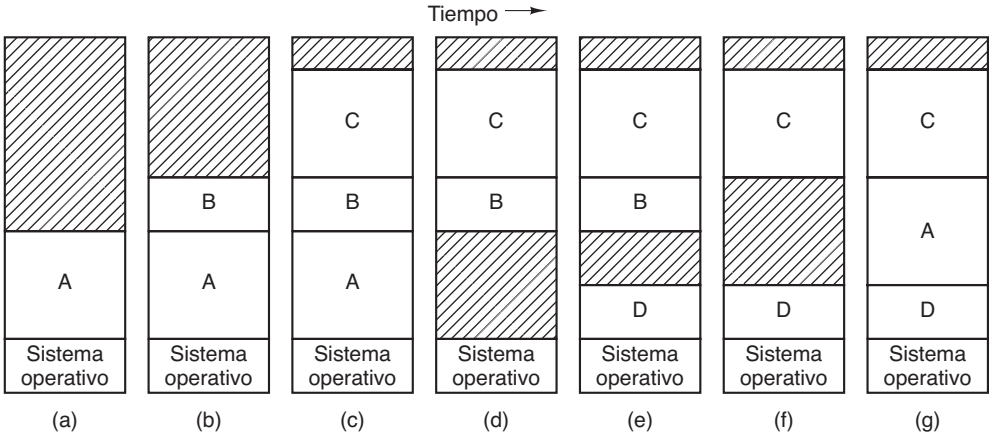

Figura 3-4. La asignación de la memoria cambia a medida que llegan procesos a la memoria y salen de ésta. Las regiones sombreadas son la memoria sin usar.

Cuando el intercambio crea varios huecos en la memoria, es posible combinarlos todos en uno grande desplazando los procesos lo más hacia abajo que sea posible. Esta técnica se conoce como **compactación de memoria**. Por lo general no se realiza debido a que requiere mucho tiempo de la CPU. Por ejemplo, en una máquina con 1 GB que pueda copiar 4 bytes en 20 nseg, se requerirían aproximadamente 5 segundos para compactar toda la memoria.

Un aspecto que vale la pena mencionar es la cantidad de memoria que debe asignarse a un proceso cuando éste se crea o se intercambia. Si los procesos se crean con un tamaño fijo que nunca cambia, entonces la asignación es sencilla: el sistema operativo asigna exactamente lo necesario, ni más ni menos.

No obstante, si los segmentos de datos de los procesos pueden crecer, por ejemplo mediante la asignación dinámica de memoria proveniente de un heap, como en muchos lenguajes de programación, ocurre un problema cuando un proceso trata de crecer. Si hay un hueco adyacente al proceso, puede asignarse y se permite al proceso crecer en el hueco; si el proceso está adyacente a otro proceso, el proceso en crecimiento tendrá que moverse a un hueco en memoria que sea lo bastante grande como para alojarlo, o habrá que intercambiar uno o más procesos para crear un hueco con el tamaño suficiente. Si un proceso no puede crecer en memoria y el área de intercambio en el disco está llena, el proceso tendrá que suspenderse hasta que se libere algo de espacio (o se puede eliminar).

Si se espera que la mayoría de los procesos crezcan a medida que se ejecuten, probablemente sea conveniente asignar un poco de memoria adicional cada vez que se intercambia o se mueve un proceso, para reducir la sobrecarga asociada con la acción de mover o intercambiar procesos que ya no caben en su memoria asignada. No obstante, al intercambiar procesos al disco, se debe intercambiar sólo la memoria que se encuentre en uso; es un desperdicio intercambiar también la memoria adicional. En la figura 3-5(a) podemos ver una configuración de memoria en la cual se ha asignado espacio para que crezcan dos procesos.

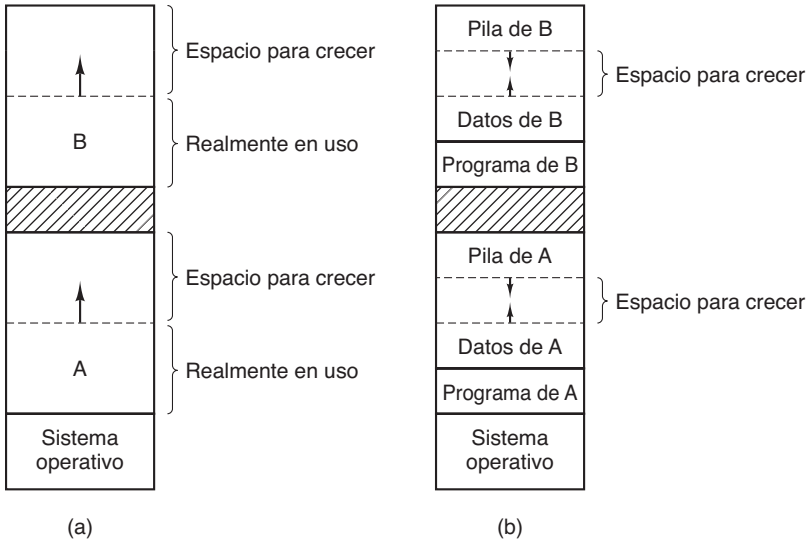

Figura 3-5. (a) Asignación de espacio para un segmento de datos en crecimiento. (b) Asignación de espacio para una pila en crecimiento y un segmento de datos en crecimiento.

Si los procesos pueden tener dos segmentos en crecimiento, por ejemplo, cuando el segmento de datos se utiliza como heap para las variables que se asignan y liberan en forma dinámica y un segmento de pila para las variables locales normales y las direcciones de retorno, un arreglo alternativo se sugiere por sí mismo, a saber, el de la figura 3-5(b). En esta figura podemos ver que cada proceso ilustrado tiene una pila en la parte superior de su memoria asignada, la cual está creciendo hacia abajo, y un segmento de datos justo debajo del texto del programa, que está creciendo hacia arriba. La memoria entre estos segmentos se puede utilizar para cualquiera de los dos. Si se agota, el proceso tendrá que moverse a un hueco con suficiente espacio, intercambiarse fuera de la memoria hasta que se pueda crear un hueco lo suficientemente grande, o eliminarse.

3.2.3 Administración de memoria libre

Cuando la memoria se asigna en forma dinámica, el sistema operativo debe administrarla. En términos generales, hay dos formas de llevar el registro del uso de la memoria: mapas de bits y listas libres. En esta sección y en la siguiente analizaremos estos dos métodos.

Administración de memoria con mapas de bits

Con un mapa de bits, la memoria se divide en unidades de asignación tan pequeñas como unas cuantas palabras y tan grandes como varios kilobytes. Para cada unidad de asignación hay un bit correspondiente en el mapa de bits, que es 0 si la unidad está libre y 1 si está ocupada (o viceversa). La figura 3-6 muestra parte de la memoria y el mapa de bits correspondiente.

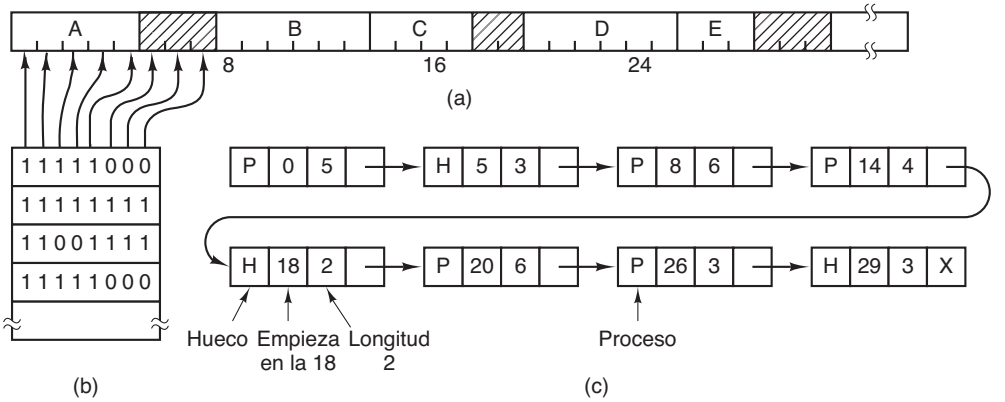

Figura 3-6. (a) Una parte de la memoria con cinco procesos y tres huecos. Las marcas de graduación muestran las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están libres. (b) El mapa de bits correspondiente. (c) La misma información en forma de lista.

El tamaño de la unidad de asignación es una importante cuestión de diseño. Entre más pequeña sea la unidad de asignación, mayor será el mapa de bits. Sin embargo, aun con una unidad de asignación tan pequeña como 4 bytes, 32 bits de memoria sólo requerirán 1 bit del mapa. Una memoria de 32*n* bits utilizará *n* bits del mapa, por lo que el mapa de bits ocupará sólo 1/33 de la memoria. Si la unidad de asignación se elige de manera que sea grande, el mapa de bits será más pequeño pero se puede desperdiciar una cantidad considerable de memoria en la última unidad del proceso si su tamaño no es un múltiplo exacto de la unidad de asignación.

Un mapa de bits proporciona una manera simple de llevar el registro de las palabras de memoria en una cantidad fija de memoria, debido a que el tamaño del mapa de bits sólo depende del tamaño de la memoria y el tamaño de la unidad de asignación. El problema principal es que, cuando se ha decidido llevar un proceso de *k* unidades a la memoria, el administrador de memoria debe buscar en el mapa para encontrar una serie de *k* bits consecutivos con el valor 0 en el mapa de bits. El proceso de buscar en un mapa de bits una serie de cierta longitud es una operación lenta (debido a que la serie puede cruzar los límites entre las palabras en el mapa); éste es un argumento contra los mapas de bits.

Administración de memoria con listas ligadas

Otra manera de llevar el registro de la memoria es mantener una lista ligada de segmentos de memoria asignados y libres, en donde un segmento contiene un proceso o es un hueco vacío entre dos

procesos. La memoria de la figura 3-6(a) se representa en la figura 3-6(c) como una lista ligada de segmentos. Cada entrada en la lista especifica un hueco (H) o un proceso (P), la dirección en la que inicia, la longitud y un apuntador a la siguiente entrada.

En este ejemplo, la lista de segmentos se mantiene ordenada por dirección. Al ordenarla de esta manera, tenemos la ventaja de que cuando termina un proceso o se intercambia, el proceso de actualizar la lista es simple. Un proceso en terminación por lo general tiene dos vecinos (excepto cuando se encuentra en la parte superior o inferior de la memoria). Éstos pueden ser procesos o huecos, lo cual conduce a las cuatro combinaciones de la figura 3-7. En la figura 3-7(a), para actualizar la lista se requiere reemplazar una P por una H. En las figuras 3-7(b) y 3-7(c) dos entradas se fusionan en una sola y la lista se reduce en una entrada. En la figura 3-7(d), se fusionan las tres entradas y dos elementos son eliminados de la lista.

Como la ranura de la tabla de procesos para el proceso en terminación en general apuntará a la entrada en la lista para el mismo proceso, puede ser más conveniente tener la lista como una lista doblemente ligada, en vez de la lista simplemente ligada de la figura 3-6(c). Esta estructura facilita encontrar la entrada anterior y ver si es posible una combinación.

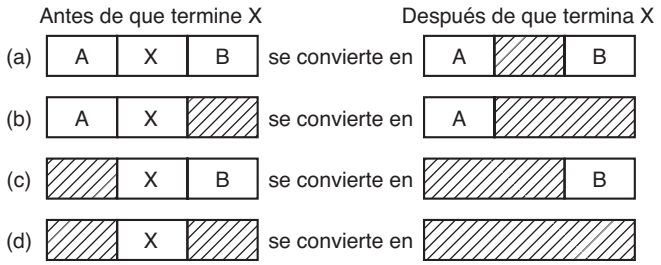

Figura 3-7. Cuatro combinaciones de los vecinos para el proceso en terminación, *X*.

Cuando los procesos y huecos se mantienen en una lista ordenada por dirección, se pueden utilizar varios algoritmos para asignar memoria a un proceso creado (o a un proceso existente que se intercambie del disco). Suponemos que el administrador de memoria sabe cuánta debe asignar. El algoritmo más simple es el de **primer ajuste**: el administrador de memoria explora la lista de segmentos hasta encontrar un hueco que sea lo bastante grande. Después el hueco se divide en dos partes, una para el proceso y otra para la memoria sin utilizar, excepto en el estadísticamente improbable caso de un ajuste exacto. El algoritmo del primer ajuste es rápido debido a que busca lo menos posible.

Una pequeña variación del algoritmo del primer ajuste es el algoritmo del **siguiente ajuste**. Funciona de la misma manera que el primer ajuste, excepto porque lleva un registro de dónde se encuentra cada vez que descubre un hueco adecuado. La siguiente vez que es llamado para buscar un hueco, empieza a buscar en la lista desde el lugar en el que se quedó la última vez, en vez de empezar siempre desde el principio, como el algoritmo del primer ajuste. Las simulaciones realizadas por Bays (1977) muestran que el algoritmo del siguiente ajuste tiene un rendimiento ligeramente peor que el del primer ajuste.

Otro algoritmo muy conocido y ampliamente utilizado es el del **mejor ajuste**. Este algoritmo busca en toda la lista, de principio a fin y toma el hueco más pequeño que sea adecuado. En vez de dividir un gran hueco que podría necesitarse después, el algoritmo del mejor ajuste trata de buscar

un hueco que esté cerca del tamaño actual necesario, que coincida mejor con la solicitud y los huecos disponibles.

Como ejemplo de los algoritmos de primer ajuste y de mejor ajuste, considere de nuevo la figura 3-6. Si se necesita un bloque de tamaño 2, el algoritmo del primer ajuste asignará el hueco en la 5, pero el del mejor ajuste asignará el hueco en la 18.

El algoritmo del mejor ajuste es más lento que el del primer ajuste, ya que debe buscar en toda la lista cada vez que se le llama. De manera sorprendente, también provoca más desperdicio de memoria que los algoritmos del primer ajuste o del siguiente ajuste, debido a que tiende a llenar la memoria con huecos pequeños e inutilizables. El algoritmo del primer ajuste genera huecos más grandes en promedio.

Para resolver el problema de dividir las coincidencias casi exactas en un proceso y en un pequeño hueco, podríamos considerar el algoritmo del **peor ajuste**, es decir, tomar siempre el hueco más grande disponible, de manera que el nuevo hueco sea lo bastante grande como para ser útil. La simulación ha demostrado que el algoritmo del peor ajuste no es muy buena idea tampoco.

Los cuatro algoritmos pueden ser acelerados manteniendo listas separadas para los procesos y los huecos. De esta forma, todos ellos dedican toda su energía a inspeccionar los huecos, no los procesos. El precio inevitable que se paga por esta aceleración en la asignación es la complejidad adicional y la lentitud al desasignar la memoria, ya que un segmento liberado tiene que eliminarse de la lista de procesos e insertarse en la lista de huecos.

Si se mantienen distintas listas para los procesos y los huecos, la lista de huecos se puede mantener ordenada por el tamaño, para que el algoritmo del mejor ajuste sea más rápido. Cuando el algoritmo del mejor ajuste busca en una lista de huecos, del más pequeño al más grande, tan pronto como encuentre un hueco que ajuste, sabrá que el hueco es el más pequeño que se puede utilizar, de aquí que se le denomine el mejor ajuste. No es necesario buscar más, como con el esquema de una sola lista. Con una lista de huecos ordenada por tamaño, los algoritmos del primer ajuste y del mejor ajuste son igual de rápidos, y hace innecesario usar el del siguiente ajuste.

Cuando los huecos se mantienen en listas separadas de los procesos, una pequeña optimización es posible. En vez de tener un conjunto separado de estructuras de datos para mantener la lista de huecos, como se hizo en la figura 3-6(c), la información se puede almacenar en los huecos. La primera palabra de cada hueco podría ser el tamaño del mismo y la segunda palabra un apuntador a la siguiente entrada. Los nodos de la lista de la figura 3-6(c), que requieren tres palabras y un bit (P/H), ya no son necesarios.

Un algoritmo de asignación más es el denominado de **ajuste rápido**, el cual mantiene listas separadas para algunos de los tamaños más comunes solicitados. Por ejemplo, podría tener una tabla con *n* entradas, en donde la primera entrada es un apuntador a la cabeza de una lista de huecos de 4 KB, la segunda entrada es un apuntador a una lista de huecos de 8 KB, la tercera entrada un apuntador a huecos de 12 KB y así sucesivamente. Por ejemplo, los huecos de 21 KB podrían colocarse en la lista de 20 KB o en una lista especial de huecos de tamaño inusual.

Con el algoritmo del ajuste rápido, buscar un hueco del tamaño requerido es extremadamente rápido, pero tiene la misma desventaja que todos los esquemas que se ordenan por el tamaño del hueco: cuando un proceso termina o es intercambiado, buscar en sus vecinos para ver si es posible una fusión es un proceso costoso. Si no se realiza la fusión, la memoria se fragmentará rápidamente en un gran número de pequeños huecos en los que no cabrá ningún proceso.

3.3 MEMORIA VIRTUAL

Mientras que los registros base y límite se pueden utilizar para crear la abstracción de los espacios de direcciones, hay otro problema que se tiene que resolver: la administración del agrandamiento del software (*bloatware*). Aunque el tamaño de las memorias se incrementa con cierta rapidez, el del software aumenta con una mucha mayor. En la década de 1980, muchas universidades operaban un sistema de tiempo compartido con docenas de usuarios (más o menos satisfechos) trabajando simultáneamente en una VAX de 4 MB. Ahora Microsoft recomienda tener por lo menos 512 MB para que un sistema Vista de un solo usuario ejecute aplicaciones simples y 1 GB si el usuario va a realizar algún trabajo serio. La tendencia hacia la multimedia impone aún mayores exigencias sobre la memoria.

Como consecuencia de estos desarrollos, existe la necesidad de ejecutar programas que son demasiado grandes como para caber en la memoria y sin duda existe también la necesidad de tener sistemas que puedan soportar varios programas ejecutándose al mismo tiempo, cada uno de los cuales cabe en memoria, pero que en forma colectiva exceden el tamaño de la misma. El intercambio no es una opción atractiva, ya que un disco SATA ordinario tiene una velocidad de transferencia pico de 100 MB/segundo a lo más, lo cual significa que requiere por lo menos 10 segundos para intercambiar un programa de 1 GB de memoria a disco y otros 10 segundos para intercambiar un programa de 1 GB del disco a memoria.

El problema de que los programas sean más grandes que la memoria ha estado presente desde los inicios de la computación, en áreas limitadas como la ciencia y la ingeniería (para simular la creación del universo o, incluso, para simular una nueva aeronave, se requiere mucha memoria). Una solución que se adoptó en la década de 1960 fue dividir los programas en pequeñas partes, conocidas como **sobrepuestos** (***overlays***). Cuando empezaba un programa, todo lo que se cargaba en memoria era el administrador de sobrepuestos, que de inmediato cargaba y ejecutaba el sobrepuesto 0; cuando éste terminaba, indicaba al administrador de sobrepuestos que cargara el 1 encima del sobrepuesto 0 en la memoria (si había espacio) o encima del mismo (si no había espacio). Algunos sistemas de sobrepuestos eran muy complejos, ya que permitían varios sobrepuestos en memoria a la vez. Los sobrepuestos se mantenían en el disco, intercambiándolos primero hacia adentro de la memoria y después hacia afuera de la memoria mediante el administrador de sobrepuestos.

Aunque el trabajo real de intercambiar sobrepuestos hacia adentro y hacia afuera de la memoria lo realizaba el sistema operativo, el de dividir el programa en partes tenía que realizarlo el programador en forma manual. El proceso de dividir programas grandes en partes modulares más pequeñas consumía mucho tiempo, y era aburrido y propenso a errores. Pocos programadores eran buenos para esto. No pasó mucho tiempo antes de que alguien ideara una forma de pasar todo el trabajo a la computadora.

El método ideado (Fotheringham, 1961) se conoce actualmente como **memoria virtual**. La idea básica detrás de la memoria virtual es que cada programa tiene su propio espacio de direcciones, el cual se divide en trozos llamados **páginas**. Cada página es un rango contiguo de direcciones. Estas páginas se asocian a la memoria física, pero no todas tienen que estar en la memoria física para poder ejecutar el programa. Cuando el programa hace referencia a una parte de su espacio de direcciones que está en la memoria física, el hardware realiza la asociación necesaria al instante. Cuando el programa hace referencia a una parte de su espacio de direcciones que *no* está en la memoria física, el sistema operativo recibe una alerta para buscar la parte faltante y volver a ejecutar la instrucción que falló.

En cierto sentido, la memoria virtual es una generalización de la idea de los registros base y límite. El procesador 8088 tenía registros base separados (pero no tenía registros límite) para texto y datos. Con la memoria virtual, en vez de tener una reubicación por separado para los segmentos de texto y de datos, todo el espacio de direcciones se puede asociar a la memoria física en unidades pequeñas equitativas. A continuación le mostraremos cómo se implementa la memoria virtual.

La memoria virtual funciona muy bien en un sistema de multiprogramación, con bits y partes de muchos programas en memoria a la vez. Mientras un programa espera a que una parte del mismo se lea y coloque en la memoria, la CPU puede otorgarse a otro proceso.

3.3.1 Paginación

La mayor parte de los sistemas de memoria virtual utilizan una técnica llamada **paginación**, que describiremos a continuación. En cualquier computadora, los programas hacen referencia a un conjunto de direcciones de memoria. Cuando un programa ejecuta una instrucción como

```
MOV REG, 1000
```

lo hace para copiar el contenido de la dirección de memoria 1000 a `REG` (o viceversa, dependiendo de la computadora). Las direcciones se pueden generar usando indexado, registros base, registros de segmentos y otras formas más.

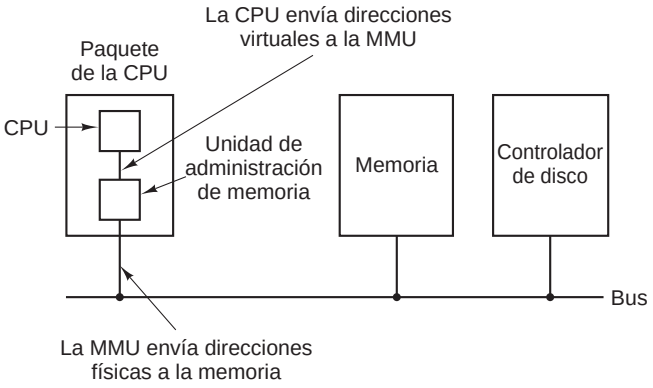

Figura 3-8. La posición y función de la MMU. Aquí la MMU se muestra como parte del chip de CPU, debido a que es común esta configuración en la actualidad. Sin embargo, lógicamente podría ser un chip separado y lo era hace años.

Estas direcciones generadas por el programa se conocen como **direcciones virtuales** y forman el **espacio de direcciones virtuales**. En las computadoras sin memoria virtual, la dirección física se coloca directamente en el bus de memoria y hace que se lea o escriba la palabra de memoria física con la misma dirección. Cuando se utiliza memoria virtual, las direcciones virtuales no van directamente al bus de memoria. En vez de ello, van a una **MMU** (*Memory Managemente Unit*,

Unidad de administración de memoria) que asocia las direcciones virtuales a las direcciones de memoria físicas, como se ilustra en la figura 3-8.

En la figura 3-9 se muestra un ejemplo muy simple de cómo funciona esta asociación. En este ejemplo, tenemos una computadora que genera direcciones de 16 bits, desde 0 hasta 64 K. Éstas son las direcciones virtuales. Sin embargo, esta computadora sólo tiene 32 KB de memoria física. Así, aunque se pueden escribir programas de 64 KB, no se pueden cargar completos en memoria y ejecutarse. No obstante, una copia completa de la imagen básica de un programa, de hasta 64 KB, debe estar presente en el disco para que las partes se puedan traer a la memoria según sea necesario.

El espacio de direcciones virtuales se divide en unidades de tamaño fijo llamadas **páginas**. Las unidades correspondientes en la memoria física se llaman **marcos de página**. Las páginas y los marcos de página por lo general son del mismo tamaño. En este ejemplo son de 4 KB, pero en sistemas reales se han utilizado tamaños de página desde 512 bytes hasta 64 KB. Con 64 KB de espacio de direcciones virtuales y 32 KB de memoria física obtenemos 16 páginas virtuales y 8 marcos de página. Las transferencias entre la RAM y el disco siempre son en páginas completas.

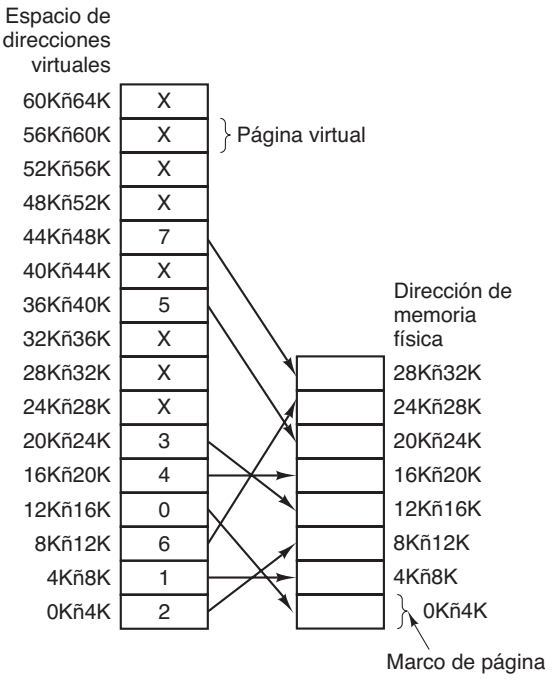

Figura 3-9. La relación entre las direcciones virtuales y las direcciones de memoria física está dada por la tabla de páginas. Cada página empieza en un múltiplo de 4096 y termina 4095 direcciones más arriba, por lo que de 4 K a 8 K en realidad significa de 4096 a 8191 y de 8 K a 12 K significa de 8192 a 12287.

La notación en la figura 3-9 es la siguiente. El rango marcado de 0K a 4 K significa que las direcciones virtuales o físicas en esa página son de 0 a 4095. El rango de 4 K a 8 K se refiere a las

direcciones de 4096 a 8191 y así en lo sucesivo. Cada página contiene exactamente 4096 direcciones que empiezan en un múltiplo de 4096 y terminan uno antes del múltiplo de 4096.

Por ejemplo, cuando el programa trata de acceder a la dirección 0 usando la instrucción

```
MOV REG,0
```

la dirección virtual 0 se envía a la MMU. La MMU ve que esta dirección virtual está en la página 0 (0 a 4095), que de acuerdo con su asociación es el marco de página 2 (8192 a 12287). Así, transforma la dirección en 8192 y envía la dirección 8192 al bus. La memoria no sabe nada acerca de la MMU y sólo ve una petición para leer o escribir en la dirección 8192, la cual cumple. De esta manera, la MMU ha asociado efectivamente todas las direcciones virtuales entre 0 y 4095 sobre las direcciones físicas de 8192 a 12287.

De manera similar, la instrucción

```
MOV REG,8192
```

se transforma efectivamente en

```
MOV REG,24576
```

debido a que la dirección virtual 8192 (en la página virtual 2) se asocia con la dirección 24576 (en el marco de página físico 6). Como tercer ejemplo, la dirección virtual 20500 está a 20 bytes del inicio de la página virtual 5 (direcciones virtuales 20480 a 24575) y la asocia con la dirección física $12288 + 20 = 12308$.

Por sí sola, la habilidad de asociar 16 páginas virtuales a cualquiera de los ocho marcos de página mediante la configuración de la apropiada asociación de la MMU no resuelve el problema de que el espacio de direcciones virtuales sea más grande que la memoria física. Como sólo tenemos ocho marcos de página físicos, sólo ocho de las páginas virtuales en la figura 3-9 se asocian a la memoria física. Las demás, que se muestran con una cruz en la figura, no están asociadas. En el hardware real, un **bit de presente/ausente** lleva el registro de cuáles páginas están físicamente presentes en la memoria.

¿Qué ocurre si el programa hace referencia a direcciones no asociadas, por ejemplo, mediante el uso de la instrucción

```
MOV REG,32780
```

que es el byte 12 dentro de la página virtual 8 (empezando en 32768)? La MMU detecta que la página no está asociada (lo cual se indica mediante una cruz en la figura) y hace que la CPU haga un trap al sistema operativo. A este trap se le llama **fallo de página**. El sistema operativo selecciona un marco de página que se utilice poco y escribe su contenido de vuelta al disco (si no es que ya está ahí). Después obtiene la página que se acaba de referenciar en el marco de página que se acaba de liberar, cambia la asociación y reinicia la instrucción que originó el trap.

Por ejemplo, si el sistema operativo decidiera desalojar el marco de página 1, cargaría la página virtual 8 en la dirección física 8192 y realizaría dos cambios en la asociación de la MMU. Primero, marcaría la entrada de la página virtual 1 como no asociada, para hacer un trap por cualquier acceso a las direcciones virtuales entre 4096 y 8191. Después reemplazaría la cruz en la entrada de

la página virtual 8 con un 1, de manera que al ejecutar la instrucción que originó el trap, asocie la dirección virtual 32780 a la dirección física 4108 (4096 + 12).

Ahora veamos dentro de la MMU para analizar su funcionamiento y por qué hemos optado por utilizar un tamaño de página que sea una potencia de 2. En la figura 3-10 vemos un ejemplo de una dirección virtual, 8196 (0010000000000100 en binario), que se va a asociar usando la asociación de la MMU en la figura 3-9. La dirección virtual entrante de 16 bits se divide en un número de página de 4 bits y en un desplazamiento de 12 bits. Con 4 bits para el número de página, podemos tener 16 páginas y con los 12 bits para el desplazamiento, podemos direccionar todos los 4096 bytes dentro de una página.

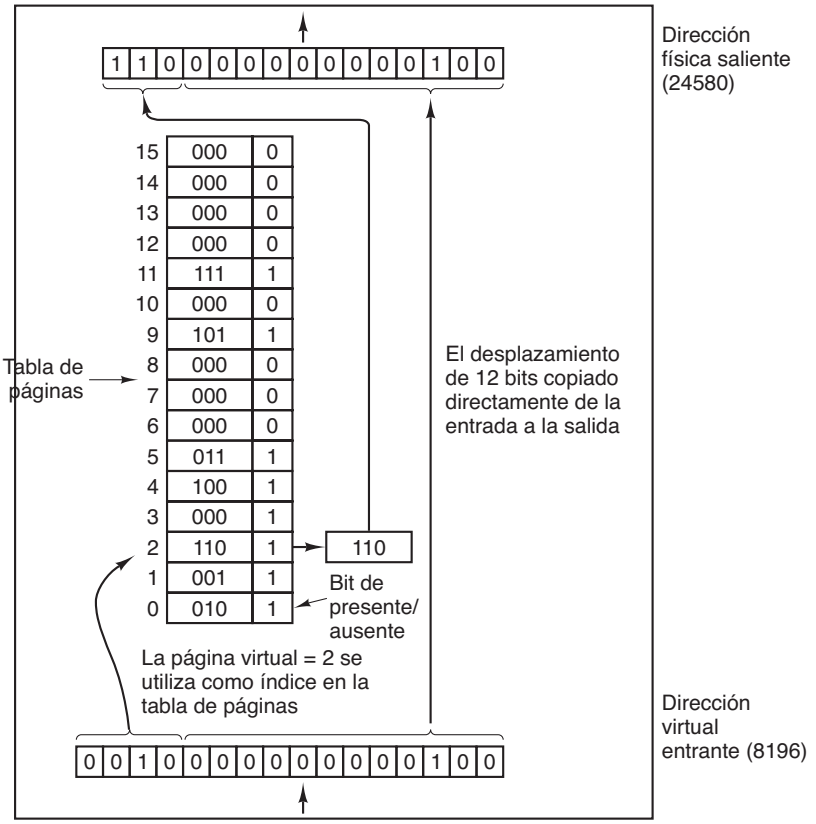

Figura 3-10. La operación interna de la MMU con 16 páginas de 4 KB.

El número de página se utiliza como índice en la **tabla de páginas**, conduciendo al número del marco de página que corresponde a esa página virtual. Si el bit de *presente/ausente* es 0, se provoca un trap al sistema operativo. Si el bit es 1, el número del marco de página encontrado en la tabla de páginas se copia a los 3 bits de mayor orden del registro de salida, junto con el desplazamiento de 12 bits, que se copia sin modificación de la dirección virtual entrante. En conjunto forman una dirección física de 15 bits. Después, el registro de salida se coloca en el bus de memoria como la dirección de memoria física.

3.3.2 Tablas de páginas

En una implementación simple, la asociación de direcciones virtuales a direcciones físicas se puede resumir de la siguiente manera: la dirección virtual se divide en un número de página virtual (bits de mayor orden) y en un desplazamiento (bits de menor orden). Por ejemplo, con una dirección de 16 bits y un tamaño de página de 4 KB, los 4 bits superiores podrían especificar una de las 16 páginas virtuales y los 12 bits inferiores podrían entonces especificar el desplazamiento de bytes (0 a 4095) dentro de la página seleccionada. Sin embargo, también es posible una división con 3, 5 u otro número de bits para la página. Las distintas divisiones implican diferentes tamaños de página.

El número de página virtual se utiliza como índice en la tabla de páginas para buscar la entrada para esa página virtual. En la entrada en la tabla de páginas, se encuentra el número de marco de página (si lo hay). El número del marco de página se adjunta al extremo de mayor orden del desplazamiento, reemplazando el número de página virtual, para formar una dirección física que se pueda enviar a la memoria.

Por ende, el propósito de la tabla de páginas es asociar páginas virtuales a los marcos de página. Hablando en sentido matemático, la tabla de páginas es una función donde el número de página virtual es un argumento y el número de marco físico es un resultado. Utilizando el resultado de esta función, el campo de la página virtual en una dirección virtual se puede reemplazar por un campo de marco de página, formando así una dirección de memoria física.

Estructura de una entrada en la tabla de páginas

Ahora vamos a pasar de la estructura de las tablas de páginas en general a los detalles de una sola entrada en la tabla de páginas. La distribución exacta de una entrada depende en gran parte de la máquina, pero el tipo de información presente es aproximadamente el mismo de una máquina a otra. En la figura 3-11 proporcionamos un ejemplo de una entrada en la tabla de páginas. El tamaño varía de una computadora a otra, pero 32 bits es un tamaño común. El campo más importante es el *número de marco de página*. Después de todo, el objetivo de la asociación de páginas es mostrar este valor. Enseguida de este campo tenemos el bit de *presente/ausente*. Si este bit es 1, la entrada es válida y se puede utilizar. Si es 0, la página virtual a la que pertenece la entrada no se encuentra actualmente en la memoria. Al acceder a una entrada en la tabla de página con este bit puesto en 0 se produce un fallo de página.

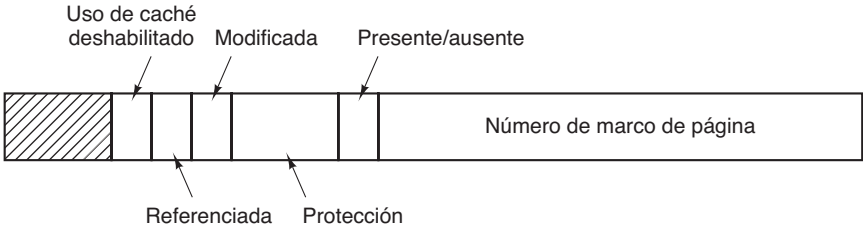

Figura 3-11. Una típica entrada en la tabla de páginas.

Los bits de *protección* indican qué tipo de acceso está permitido. En su forma más simple, este campo contiene 1 bit, con 0 para lectura/escritura y 1 para sólo lectura. Un arreglo más sofisticado es tener 3 bits: uno para habilitar la lectura, otro para la escritura y el tercero para ejecutar la página.

Los bits de *modificada* y *referenciada* llevan el registro del uso de páginas. Cuando se escribe en una página, el hardware establece de manera automática el bit de *modificada*. Este bit es valioso cuando el sistema operativo decide reclamar un marco de página. Si la página en él ha sido modificada (es decir, está "sucia"), debe escribirse de vuelta en el disco. Si no se ha modificado (es decir, está "limpia") sólo se puede abandonar, debido a que la copia del disco es aun válida. A este bit se le conoce algunas veces como **bit sucio**, ya que refleja el estado de la página.

El bit de *referenciada* se establece cada vez que una página es referenciada, ya sea para leer o escribir. Su función es ayudar al sistema operativo a elegir una página para desalojarla cuando ocurre un fallo de página. Las páginas que no se estén utilizando son mejores candidatos que las páginas que se están utilizando y este bit desempeña un importante papel en varios de los algoritmos de reemplazo de páginas que estudiaremos más adelante en este capítulo.

Finalmente, el último bit permite deshabilitar el uso de caché para la página. Esta característica es importante para las páginas que se asocian con los registros de dispositivos en vez de la memoria. Si el sistema operativo está esperando en un ciclo de espera hermético a que cierto dispositivo de E/S responda a un comando que acaba de recibir, es esencial que el hardware siga obteniendo la palabra del dispositivo y no utilice una copia antigua en la caché. Con este bit, el uso de la caché se puede desactivar. Las máquinas que tienen un espacio de E/S separado y no utilizan E/S asociada a la memoria no necesitan este bit.

Observe que la dirección del disco utilizado para guardar la página cuando no está en memoria no forma parte de la tabla de páginas. La razón es simple: la tabla de páginas sólo guarda la información que el hardware necesita para traducir una dirección virtual en una dirección física. La información que necesita el sistema operativo para manejar los fallos de página se mantiene en tablas de software dentro del sistema operativo. El hardware no la necesita.

Antes de analizar más aspectos de la implementación, vale la pena recalcar de nuevo que lo que la memoria virtual hace es crear una abstracción —el espacio de direcciones— que es una abstracción de la memoria física, al igual que un proceso es una abstracción del procesador físico (CPU). Para implementar la memoria virtual, hay que descomponer el espacio de direcciones virtuales en páginas y asociar cada una a cierto marco de página de memoria física o dejarla (temporalmente) sin asociar. Por ende, este capítulo trata fundamentalmente de una abstracción creada por el sistema operativo y la forma en que se administra esa abstracción.

3.3.3 Aceleración de la paginación

Acabamos de ver los fundamentos de la memoria virtual y la paginación. Ahora es tiempo de entrar en más detalle acerca de las posibles implementaciones. En cualquier sistema de paginación hay que abordar dos cuestiones principales:

1. La asociación de una dirección virtual a una dirección física debe ser rápida.

2. Si el espacio de direcciones virtuales es grande, la tabla de páginas será grande.

El primer punto es una consecuencia del hecho de que la asociación virtual-a-física debe realizarse en cada referencia de memoria. Todas las instrucciones deben provenir finalmente de la memoria y muchas de ellas hacen referencias a operandos en memoria también. En consecuencia, es necesario hacer una, dos o algunas veces más referencias a la tabla de páginas por instrucción. Si la ejecución de una instrucción tarda, por ejemplo 1 nseg, la búsqueda en la tabla de páginas debe realizarse en menos de 0.2 nseg para evitar que la asociación se convierta en un cuello de botella importante.

El segundo punto se deriva del hecho de que todas las computadoras modernas utilizan direcciones virtuales de por lo menos 32 bits, donde 64 bits se vuelven cada vez más comunes. Por decir, con un tamaño de página de 4 KB, un espacio de direcciones de 32 bits tiene 1 millón de páginas y un espacio de direcciones de 64 bits tiene más de las que desearíamos contemplar. Con 1 millón de páginas en el espacio de direcciones virtual, la tabla de páginas debe tener 1 millón de entradas. Y recuerde que cada proceso necesita su propia tabla de páginas (debido a que tiene su propio espacio de direcciones virtuales).

La necesidad de una asociación de páginas extensa y rápida es una restricción considerable en cuanto a la manera en que se construyen las computadoras. El diseño más simple (por lo menos en concepto) es tener una sola tabla de páginas que consista en un arreglo de registros de hardware veloces, con una entrada para cada página virtual, indexada por número de página virtual, como se muestra en la figura 3-10. Cuando se inicia un proceso, el sistema operativo carga los registros con la tabla de páginas del proceso, tomada de una copia que se mantiene en la memoria principal. Durante la ejecución del proceso no se necesitan más referencias a memoria para la tabla de páginas. Las ventajas de este método son que es simple y no requiere referencias a memoria durante la asociación. Una desventaja es que es extremadamente costoso que la tabla de páginas sea extensa; otra es que tener que cargar la tabla de páginas completa en cada conmutación de contexto ve afectado el rendimiento.

En el otro extremo, toda la tabla de páginas puede estar en la memoria principal. Así, todo lo que el hardware necesita es un solo registro que apunte al inicio de la tabla de páginas. Este diseño permite cambiar la asociación de direcciones virtuales a direcciones físicas al momento de una conmutación de contexto con sólo recargar un registro. Desde luego, tiene la desventaja de requerir una o más referencias a memoria para leer las entradas en la tabla de páginas durante la ejecución de cada instrucción, con lo cual se hace muy lenta.

Búferes de traducción adelantada

Ahora veamos esquemas implementados ampliamente para acelerar la paginación y manejar espacios de direcciones virtuales extensos, empezando con la aceleración de la paginación. El punto inicial de la mayor parte de las técnicas de optimización es que la tabla de páginas está en la memoria. Potencialmente, este diseño tiene un enorme impacto sobre el rendimiento. Por ejemplo, considere una instrucción de 1 byte que copia un registro a otro. A falta de paginación, esta instrucción hace

sólo una referencia a memoria para obtener la instrucción. Con la paginación se requiere al menos una referencia adicional a memoria para acceder a la tabla de páginas. Como la velocidad de ejecución está comúnmente limitada por la proporción a la que la CPU puede obtener instrucciones y datos de la memoria, al tener que hacer dos referencias a memoria por cada una de ellas se reduce el rendimiento a la mitad. Bajo estas condiciones, nadie utilizaría la paginación.

Los diseñadores de computadoras han sabido acerca de este problema durante años y han ideado una solución que está basada en la observación de que la mayor parte de los programas tienden a hacer un gran número de referencias a un pequeño número de páginas y no viceversa. Por ende, sólo se lee con mucha frecuencia una pequeña fracción de las entradas en la tabla de páginas; el resto se utiliza muy pocas veces.

La solución que se ha ideado es equipar a las computadoras con un pequeño dispositivo de hardware para asociar direcciones virtuales a direcciones físicas sin pasar por la tabla de páginas. El dispositivo, llamado **TLB** (*Translation Lookaside Buffer*, Búfer de traducción adelantada) o algunas veces **memoria asociativa**, se ilustra en la figura 3-12. Por lo general se encuentra dentro de la MMU y consiste en un pequeño número de entradas, ocho en este ejemplo, pero raras veces más de 64. Cada entrada contiene información acerca de una página, incluyendo el número de página virtual, un bit que se establece cuando se modifica la página, el código de protección (permisos de lectura/escritura/ejecución) y el marco de página físico en el que se encuentra la página. Estos campos tienen una correspondencia de uno a uno con los campos en la tabla de páginas, excepto por el número de página virtual, que no se necesita en la tabla de páginas. Otro bit indica si la entrada es válida (es decir, si está en uso) o no.

Válida	Página virtual	Modificada	Protección	Marco de página
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figura 3-12. Un TLB para acelerar la paginación.

Un ejemplo que podría generar el TLB de la figura 3-12 es un proceso en un ciclo que abarque las páginas virtuales 19, 20 y 21, de manera que estas entradas en el TLB tengan códigos de protección para lectura y ejecución. Los datos principales que se están utilizando (por decir, un arreglo que se esté procesando) se encuentran en las páginas 129 y 130. La página 140 contiene los índices utilizados en los cálculos del arreglo. Al final, la pila está en las páginas 860 y 861.

Ahora veamos cómo funciona el TLB. Cuando se presenta una dirección virtual a la MMU para que la traduzca, el hardware primero comprueba si su número de página virtual está presente en

el TLB al compararla con todas las entradas en forma simultánea (es decir, en paralelo). Si se encuentra una coincidencia válida y el acceso no viola los bits de protección, el marco de página se toma directamente del TLB, sin pasar por la tabla de páginas. Si el número de página virtual está presente en el TLB, pero la instrucción está tratando de escribir en una página de sólo lectura, se genera un fallo por protección.

El caso interesante es lo que ocurre cuando el número de página virtual no está en el TLB. La MMU detecta que no está y realiza una búsqueda ordinaria en la tabla de páginas. Después desaloja una de las entradas del TLB y la reemplaza con la entrada en la tabla de páginas que acaba de buscar. De esta forma, si esa página se utiliza pronto otra vez, la segunda vez se producirá un acierto en el TLB en vez de un fracaso. Cuando se purga una entrada del TLB, el bit modificado se copia de vuelta a la entrada en la tabla de páginas en memoria. Los otros valores ya están ahí, excepto el bit de referencia. Cuando el TLB se carga de la tabla de páginas, todos los campos se toman de la memoria.

Administración del TLB mediante software

Hasta ahora hemos supuesto que toda máquina con memoria virtual paginada tiene tablas de páginas reconocidas por el hardware, más un TLB. En este diseño, la administración y el manejo de fallas del TLB se realiza por completo mediante el hardware de la MMU. Las traps, o trampas, para el sistema operativo ocurren sólo cuando una página no se encuentra en memoria.

En el pasado, esta suposición era correcta. Sin embargo, muchas máquinas RISC modernas (incluyendo SPARC, MIPS y HP PA) hacen casi toda esta administración de páginas mediante software. En estas máquinas, las entradas del TLB se cargan de manera explícita mediante el sistema operativo. Cuando no se encuentra una coincidencia en el TLB, en vez de que la MMU vaya a las tablas de páginas para buscar y obtener la referencia a la página que se necesita, sólo genera un fallo del TLB y pasa el problema al sistema operativo. El sistema debe buscar la página, eliminar una entrada del TLB, introducir la nueva página y reiniciar la instrucción que originó el fallo. Y, desde luego, todo esto se debe realizar en unas cuantas instrucciones, ya que los fallos del TLB ocurren con mucha mayor frecuencia que los fallos de página.

De manera sorprendente, si el TLB es razonablemente grande (por ejemplo, de 64 entradas) para reducir la proporción de fallos, la administración del TLB mediante software resulta tener una eficiencia aceptable. El beneficio principal aquí es una MMU mucho más simple, lo cual libera una cantidad considerable de área en el chip de CPU para cachés y otras características que pueden mejorar el rendimiento. Uhlig y colaboradores (1994) describen la administración del TLB mediante software.

Se han desarrollado varias estrategias para mejorar el rendimiento en equipos que realizan la administración del TLB mediante software. Un método se enfoca en reducir los fallos del TLB y el costo de un fallo del TLB cuando llega a ocurrir (Bala y colaboradores, 1994). Para reducir los fallos del TLB, algunas veces el sistema operativo averigua de modo “intuitivo” cuáles páginas tienen más probabilidad de ser utilizadas a continuación y precarga entradas para ellas en el TLB. Por ejemplo, cuando un proceso cliente envía un mensaje a un proceso servidor en el mismo equipo, es muy probable que el servidor tenga que ejecutarse pronto. Sabiendo esto al tiempo que procesa el trap para realizar la operación `send`, el sistema también puede comprobar en dónde están las páginas de código, datos y pila del servidor, asociándolas antes de que tengan la oportunidad de producir fallos del TLB.

La forma normal de procesar un fallo del TLB, ya sea en hardware o en software, es ir a la tabla de páginas y realizar las operaciones de indexado para localizar la página referenciada. El problema al realizar esta búsqueda en software es que las páginas que contienen la tabla de páginas tal vez no estén en el TLB, lo cual producirá fallos adicionales en el TLB durante el procesamiento. Estos fallos se pueden reducir al mantener una caché grande en software (por ejemplo, de 4 KB) de entradas en el TLB en una ubicación fija, cuya página siempre se mantenga en el TLB. Al comprobar primero la caché de software, el sistema operativo puede reducir de manera substancial los fallos del TLB.

Cuando se utiliza la administración del TLB mediante software, es esencial comprender la diferencia entre los dos tipos de fallos. Un **fallo suave** ocurre cuando la página referenciada no está en el TLB, sino en memoria. Todo lo que se necesita aquí es que el TLB se actualice. No se necesita E/S de disco. Por lo general, un fallo suave requiere de 10 a 20 instrucciones de máquina y se puede completar en unos cuantos nanosegundos. Por el contrario, un **fallo duro** ocurre cuando la misma página no está en memoria (y desde luego, tampoco en el TLB). Se requiere un acceso al disco para traer la página, lo cual tarda varios milisegundos. Un fallo duro es en definitiva un millón de veces más lento que un fallo suave.

3.3.4 Tablas de páginas para memorias extensas

Los TLBs se pueden utilizar para acelerar las traducciones de direcciones virtuales a direcciones físicas sobre el esquema original de la tabla de páginas en memoria. Pero ése no es el único problema que debemos combatir. Otro problema es cómo lidiar con espacios de direcciones virtuales muy extensos. A continuación veremos dos maneras de hacerlo.

Tablas de páginas multinivel

Como primer método, considere el uso de una **tabla de páginas multinivel**. En la figura 3-13 se muestra un ejemplo simple. En la figura 3-13(a) tenemos una dirección virtual de 32 bits que se particiona en un campo *TP1* de 10 bits, un campo *TP2* de 10 bits y un campo *Desplazamiento* de 12 bits. Como los desplazamientos son de 12 bits, las páginas son de 4 KB y hay un total de 2^{20}.

El secreto del método de la tabla de páginas multinivel es evitar mantenerlas en memoria todo el tiempo, y en especial, aquellas que no se necesitan. Por ejemplo, suponga que un proceso necesita 12 megabytes: los 4 megabytes inferiores de memoria para el texto del programa, lo siguientes 4 megabytes para datos y los 4 megabytes superiores para la pila. Entre la parte superior de los datos y la parte inferior de la pila hay un hueco gigantesco que no se utiliza.

En la figura 3-13(b) podemos ver cómo funciona la tabla de página de dos niveles en este ejemplo. A la izquierda tenemos la tabla de páginas de nivel superior, con 1024 entradas, que corresponden al campo *TP1* de 10 bits. Cuando se presenta una dirección virtual a la MMU, primero extrae el campo *TP1* y utiliza este valor como índice en la tabla de páginas de nivel superior. Cada una de estas 1024 entradas representa 4 M, debido a que todo el espacio de direcciones virtuales de 4 gigabytes (es decir, de 32 bits) se ha dividido en trozos de 4096 bytes.

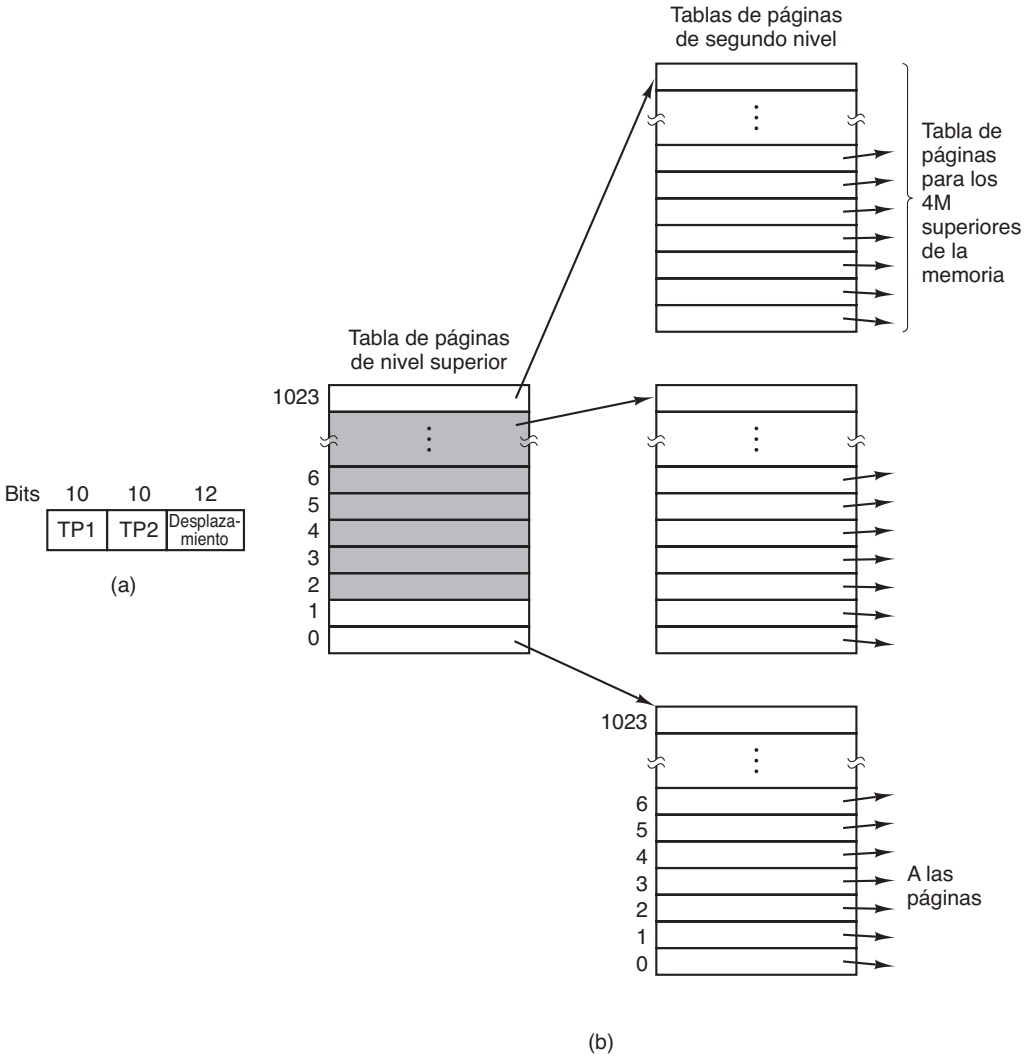

Figura 3-13. (a) Una dirección de 32 bits con dos campos de tablas de páginas. (b) Tablas de páginas de dos niveles.

La entrada que se localiza al indexar en la tabla de páginas de nivel superior produce la dirección (o número de marco de página) de una tabla de páginas de segundo nivel. La entrada 0 de la tabla de páginas de nivel superior apunta a la tabla de páginas para el texto del programa, la entrada 1 apunta a la tabla de páginas para los datos y la entrada 1023 apunta a la tabla de páginas para la pila. Las otras entradas (sombreadas) no se utilizan. Ahora el campo *TP2* se utiliza como índice en la tabla de páginas de segundo nivel seleccionada para buscar el número de marco de página para esta página en sí.

Como ejemplo, considere la dirección virtual de 32 bits 0x00403004 (4,206,596 decimal), que se encuentra 12,292 bytes dentro de los datos. Esta dirección virtual corresponde a *TP1* = 1,

TP2 = 2 y *Desplazamiento* = 4. La MMU utiliza primero a *TP1* para indexar en la tabla de páginas de nivel superior y obtener la entrada 1, que corresponde a las direcciones de 4M a 8M. Después utiliza *PT2* para indexar en la tabla de páginas de segundo nivel que acaba de encontrar y extrae la entrada 3, que corresponde a las direcciones de 12288 a 16383 dentro de su trozo de 4M (es decir, las direcciones absolutas de 4,206,592 a 4,210,687). Esta entrada contiene el número de marco de la página que contiene la dirección virtual 0x00403004. Si esa página no está en la memoria, el bit de *presente/ausente* en la entrada de la tabla de páginas será cero, con lo cual se producirá un fallo de página. Si la página está en la memoria, el número del marco de página que se obtiene de la tabla de páginas de segundo nivel se combina con el desplazamiento (4) para construir la dirección física. Esta dirección se coloca en el bus y se envía a la memoria.

Lo interesante acerca de la figura 3-13 es que, aunque el espacio de direcciones contiene más de un millón de páginas, en realidad sólo cuatro tablas de páginas son requeridas: la tabla de nivel superior y las tablas de segundo nivel de 0 a 4M (para el texto del programa), de 4M a 8M (para los datos) y los 4M superiores (para la pila). Los bits de *presente/ausente* en 1021 entradas de la tabla de páginas de nivel superior están en 0 y obligan a que se produzca un fallo de página si se tratan de utilizar alguna vez. En caso de que esto ocurra, el sistema operativo observará que el proceso está tratando de referenciar memoria que no debe y tomará la acción apropiada, como enviar una señal o eliminarlo. En este ejemplo hemos elegido números redondos para los diversos tamaños y que *TP1* sea igual a *TP2*, pero en la práctica también es posible elegir otros valores.

El sistema de tablas de páginas de dos niveles de la figura 3-13 se puede expandir a tres, cuatro o más niveles. Entre más niveles se obtiene una mayor flexibilidad, pero es improbable que la complejidad adicional sea de utilidad por encima de tres niveles.

Tablas de páginas invertidas

Para los espacios de direcciones virtuales de 32 bits, la tabla de páginas multinivel funciona bastante bien. Sin embargo, a medida que las computadoras de 64 bits se hacen más comunes, la situación cambia de manera drástica. Si el espacio de direcciones es de 2^{64} bytes, con páginas de 4 KB, necesitamos una tabla de páginas con 2^{52} entradas. Si cada entrada es de 8 bytes, la tabla es de más de 30 millones de gigabyes (30 PB). Ocupar 30 millones de gigabytes sólo para la tabla de páginas no es una buena idea por ahora y probablemente tampoco lo sea para el próximo año. En consecuencia, se necesita una solución diferente para los espacios de direcciones virtuales paginados de 64 bits.

Una de esas soluciones es la **tabla de páginas invertida**. En este diseño hay una entrada por cada marco de página en la memoria real, en vez de tener una entrada por página de espacio de direcciones virtuales. Por ejemplo, con direcciones virtuales de 64 bits, una página de 4 KB y 1 GB de RAM, una tabla de páginas invertida sólo requiere 262,144 entradas. La entrada lleva el registro de quién (proceso, página virtual) se encuentra en el marco de página.

Aunque las tablas de página invertidas ahorran grandes cantidades de espacio, al menos cuando el espacio de direcciones virtuales es mucho mayor que la memoria física, tienen una seria desventaja: la traducción de dirección virtual a dirección física se hace mucho más difícil. Cuando el proceso *n* hace referencia a la página virtual *p*, el hardware ya no puede buscar la página física usando *p* como índice en la tabla de páginas. En vez de ello, debe buscar una entrada (*n*, *p*) en toda la

tabla de páginas invertida. Lo que es peor: esta búsqueda se debe realizar en cada referencia a memoria, no sólo en los fallos de página. Buscar en una tabla de 256 K en cada referencia a memoria no es la manera de hacer que la máquina sea deslumbrantemente rápida.

La forma de salir de este dilema es utilizar el TLB. Si el TLB puede contener todas las páginas de uso frecuente, la traducción puede ocurrir con igual rapidez que con las tablas de páginas regulares. Sin embargo, en un fallo de TLB la tabla de páginas invertida tiene que buscarse mediante software. Una manera factible de realizar esta búsqueda es tener una tabla de hash arreglada según el hash de la dirección virtual. Todas las páginas virtuales que se encuentren en memoria y tengan el mismo valor de hash se encadenan en conjunto, como se muestra en la figura 3-14. Si la tabla de hash tiene tantas ranuras como las páginas físicas de la máquina, la cadena promedio tendrá sólo una entrada, con lo cual se acelera de manera considerable la asociación. Una vez que se ha encontrado el número de marco de página, se introduce el nuevo par (virtual, física) en el TLB.

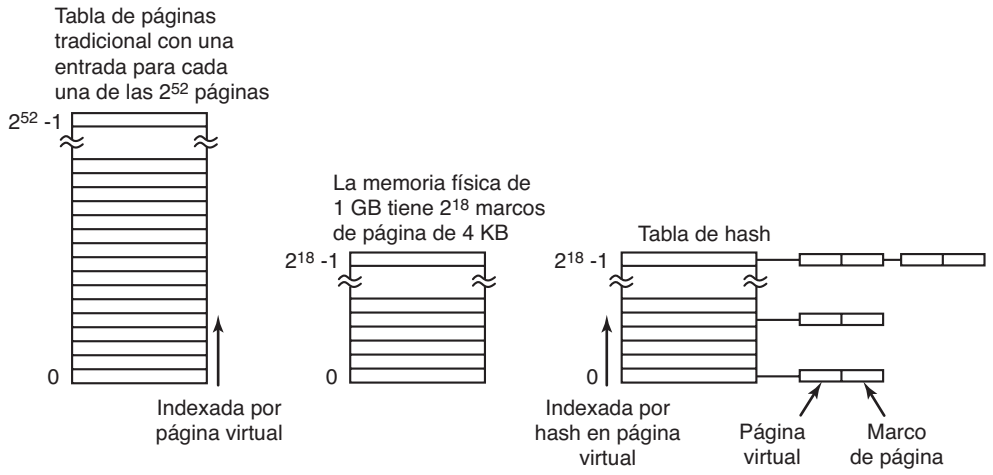

Figura 3-14. Comparación de una tabla de páginas tradicional con una tabla de páginas invertida.

Las tablas de páginas invertidas son comunes en las máquinas de 64 bits ya que incluso con un tamaño de página muy grande, el número de entradas en la tabla de páginas es enorme. Por ejemplo, con páginas de 4 MB y direcciones virtuales de 64 bits, se necesitan 2^{42} entradas en la tabla de páginas. Otros métodos para manejar memorias virtuales extensas se pueden encontrar en Talluri y colaboradores (1995).

3.4 ALGORITMOS DE REEMPLAZO DE PÁGINAS

Cuando ocurre un fallo de página, el sistema operativo tiene que elegir una página para desalojarla (eliminarla de memoria) y hacer espacio para la página entrante. Si la página a eliminar se modificó

mientras estaba en memoria, debe volver a escribirse en el disco para actualizar la copia del mismo. No obstante, si la página no se ha modificado (por ejemplo, si contiene el texto del programa), la copia ya está actualizada y no se necesita rescribir. La página que se va a leer sólo sobrescribe en la página que se va a desalojar.

Aunque sería posible elegir una página al azar para desalojarla en cada fallo de página, el rendimiento del sistema es mucho mayor si se selecciona una página que no sea de uso frecuente. Si se elimina una página de uso frecuente, tal vez tenga que traerse de vuelta rápidamente, lo cual produce una sobrecarga adicional. Se ha realizado mucho trabajo, tanto teórico como experimental en el tema de los algoritmos de reemplazo de páginas. A continuación describiremos algunos de los algoritmos más importantes.

Vale la pena observar que el problema del "reemplazo de páginas" ocurre también en otras áreas del diseño computacional. Por ejemplo, la mayoría de las computadoras tienen una o más memorias cachés que consisten en bloques de memoria de 32 bytes o 64 bytes de uso reciente. Cuando la caché está llena, hay que elegir cierto bloque para eliminarlo. Este problema es precisamente el mismo que la reemplazo de páginas, sólo que en una escala de tiempo menor (tiene que realizarse en unos cuantos nanosegundos, no milisegundos como en el reemplazo de páginas). La razón de tener una menor escala de tiempo es que los fallos de bloques de caché se satisfacen desde la memoria principal, que no tiene tiempo de búsqueda ni latencia rotacional.

Un segundo ejemplo es en un servidor Web. El servidor puede mantener cierto número de páginas Web de uso muy frecuente en su memoria caché. Sin embargo, cuando la memoria caché está llena y se hace referencia a una nueva página, hay que decidir cuál página Web se debe desalojar. Las consideraciones son similares a las de las páginas de memoria virtual, excepto por el hecho de que las páginas Web nunca se modifican en la caché, por lo cual siempre hay una copia fresca "en el disco". En un sistema de memoria virtual, las páginas en memoria principal pueden estar sucias o limpias.

En todos los algoritmos de reemplazo de páginas que analizaremos a continuación, surge cierta cuestión: al desalojar una página de la memoria, ¿tiene que ser una de las propias páginas del proceso fallido o puede ser una página que pertenezca a otro proceso? En el primer caso, estamos limitando en efecto cada proceso a un número fijo de páginas; en el segundo caso no. Ambas son posibilidades. En la sección 3-5.1 volveremos a ver este punto.

3.4.1 El algoritmo de reemplazo de páginas óptimo

El mejor algoritmo de reemplazo de páginas posible es fácil de describir, pero imposible de implementar: al momento en que ocurre un fallo de página, hay cierto conjunto de páginas en memoria y una de éstas se referenciará en la siguiente instrucción (la página que contiene la instrucción); otras páginas tal vez no se referencien sino hasta 10, 100 o tal vez 1000 instrucciones después. Cada página se puede etiquetar con el número de instrucciones que se ejecutarán antes de que se haga referencia por primera vez a esa página.

El algoritmo óptimo de reemplazo de páginas establece que la página con la etiqueta más alta debe eliminarse. Si una página no se va a utilizar durante 8 millones de instrucciones y otra no se va a utilizar durante 6 millones de instrucciones, al eliminar la primera se enviará el fallo de página que la obtendrá de vuelta lo más lejos posible en el futuro. Al igual que las personas, las computadoras tratan de posponer los eventos indeseables el mayor tiempo posible.

El único problema con este algoritmo es que no se puede realizar. Al momento del fallo de página, el sistema operativo no tiene forma de saber cuándo será la próxima referencia a cada una de las páginas (vimos una situación similar antes, con el algoritmo de planificación del trabajo más corto primero: ¿cómo puede saber el sistema cuál trabajo es más corto?). Aún así, al ejecutar un programa en un simulador y llevar la cuenta de todas las referencias a páginas, es posible implementar un algoritmo óptimo de reemplazo de página en la *segunda* corrida, al utilizar la información de referencia de páginas recolectada durante la *primera* corrida.

De esta manera, se puede comparar el rendimiento de los algoritmos realizables con el mejor posible. Si un sistema operativo logra un rendimiento de, por ejemplo, sólo 1 por ciento peor que el algoritmo óptimo, el esfuerzo invertido en buscar un mejor algoritmo producirá cuando mucho una mejora del 1 por ciento.

Para evitar cualquier posible confusión, hay que aclarar que este registro de referencias de páginas se refiere sólo al programa que se acaba de medir y sólo con una entrada específica. Por lo tanto, el algoritmo de reemplazo de páginas que se derive de esto es específico para ese programa y esos datos de entrada. Aunque este método es útil para evaluar los algoritmos de reemplazo de páginas, no es útil en sistemas prácticos. A continuación estudiaremos algoritmos que *son* útiles en sistemas reales.

3.4.2 El algoritmo de reemplazo de páginas: no usadas recientemente

Para permitir que el sistema operativo recolecte estadísticas útiles sobre el uso de páginas, la mayor parte de las computadoras con memoria virtual tienen dos bits de estado asociados a cada página. *R* se establece cada vez que se hace referencia a la página (lectura o escritura); *M* se establece cuando se escribe en la página (es decir, se modifica). Los bits están contenidos en cada entrada de la tabla de páginas, como se muestra en la figura 3-11. Es importante tener en cuenta que estos bits se deben actualizar en cada referencia a la memoria, por lo que es imprescindible que se establezcan mediante el hardware. Una vez que se establece un bit en 1, permanece así hasta que el sistema operativo lo restablece.

Si el hardware no tiene estos bits, pueden simularse de la siguiente forma. Cuando se inicia un proceso, todas sus entradas en la tabla de páginas se marcan como que no están en memoria. Tan pronto como se haga referencia a una página, ocurrirá un fallo de página. Entonces, el sistema operativo establece el bit *R* (en sus tablas internas), cambia la entrada en la tabla de páginas para que apunte a la página correcta, con modo de SÓLO LECTURA, y reinicia la instrucción. Si la página se modifica después, ocurrirá otro fallo de página que permite al sistema operativo establecer el bit *M* y cambiar el modo de la página a LECTURA/ESCRITURA.

Los bits *R* y *M* se pueden utilizar para construir un algoritmo simple de paginación de la siguiente manera. Cuando se inicia un proceso, ambos bits de página para todas sus páginas se establecen en 0 mediante el sistema operativo. El bit *R* se borra en forma periódica (en cada interrupción de reloj) para diferenciar las páginas a las que no se ha hecho referencia recientemente de las que si se han referenciado.

Cuando ocurre un fallo de página, el sistema operativo inspecciona todas las páginas y las divide en 4 categorías con base en los valores actuales de sus bits *R* y *M*:

Clase 0: no ha sido referenciada, no ha sido modificada.

Clase 1: no ha sido referenciada, ha sido modificada.

Clase 2: ha sido referenciada, no ha sido modificada.

Clase 3: ha sido referenciada, ha sido modificada.

Aunque las páginas de la clase 1 parecen a primera instancia imposibles, ocurren cuando una interrupción de reloj borra el bit *R* de una página de la clase 3. Las interrupciones de reloj no borran el bit *M* debido a que esta información se necesita para saber si la página se ha vuelto a escribir en el disco o no. Al borrar *R* pero no *M* se obtiene una página de clase 1.

El algoritmo **NRU** (*Not Recently Used*, No usada recientemente) elimina una página al azar de la clase de menor numeración que no esté vacía. En este algoritmo está implícita la idea de que es mejor eliminar una página modificada a la que no se haya hecho referencia en al menos un pulso de reloj (por lo general, unos 20 mseg) que una página limpia de uso frecuente. La principal atracción del NRU es que es fácil de comprender, moderadamente eficiente de implementar y proporciona un rendimiento que, aunque no es óptimo, puede ser adecuado.

3.4.3 El algoritmo de reemplazo de páginas: Primera en entrar, primera en salir (FIFO)

Otro algoritmo de paginación con baja sobrecarga es el de **Primera en entrar, primera en salir** (*First-In, First-Out*, **FIFO**). Para ilustrar cómo funciona, considere un supermercado con suficientes repisas como para mostrar exactamente *k* productos distintos. Un día, cierta empresa introduce un nuevo alimento de conveniencia: yogurt orgánico instantáneo liofilizado que se puede reconstituir en un horno de microondas. Es un éxito inmediato, por lo que nuestro supermercado finito se tiene que deshacer de un producto antiguo para tenerlo en existencia.

Una posibilidad es buscar el producto que el supermercado haya tenido en existencia por más tiempo (por ejemplo, algo que se empezó a vender hace 120 años) y deshacerse de él por la razón de que nadie está interesado ya. En efecto, el supermercado mantiene una lista ligada de todos los productos que vende actualmente en el orden en el que se introdujeron. El nuevo pasa a la parte final de la lista; el que está al frente de la lista se elimina.

Como un algoritmo de reemplazo de páginas, se aplica la misma idea. El sistema operativo mantiene una lista de todas las páginas actualmente en memoria, en donde la llegada más reciente está en la parte final y la menos reciente en la parte frontal. En un fallo de página, se elimina la página que está en la parte frontal y la nueva página se agrega a la parte final de la lista. Cuando se aplica en las tiendas, FIFO podría eliminar la gomina para el bigote, pero también podría eliminar harina, sal o mantequilla. Cuando se aplica a las computadoras, surge el mismo problema. Por esta razón es raro que se utilice FIFO en su forma pura.

3.4.4 El algoritmo de reemplazo de páginas: segunda oportunidad

Una modificación simple al algoritmo FIFO que evita el problema de descartar una página de uso frecuente es inspeccionar el bit *R* de la página más antigua. Si es 0, la página es antigua y no se ha

utilizado, por lo que se sustituye de inmediato. Si el bit *R* es 1, el bit se borra, la página se pone al final de la lista de páginas y su tiempo de carga se actualiza, como si acabara de llegar a la memoria. Después la búsqueda continúa.

La operación de este algoritmo, conocido como **segunda oportunidad**, se muestra en la figura 3-15. En la figura 3-15(a) vemos que las páginas de la *A* a la *H* se mantienen en una lista ligada y se ordenan con base en el tiempo en que llegaron a la memoria.

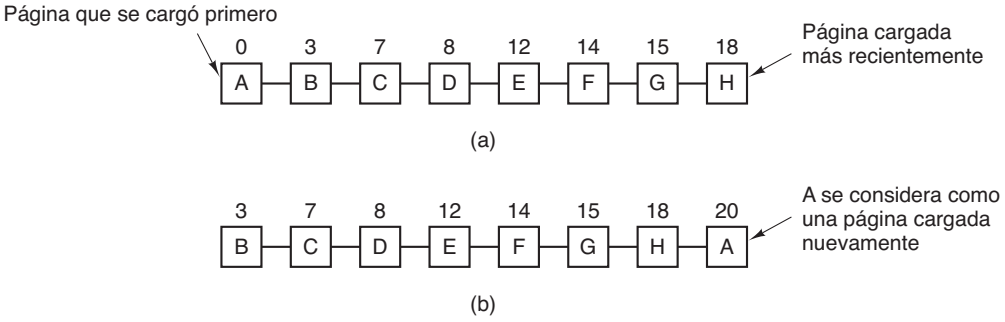

Figura 3-15. Operación del algoritmo de la segunda oportunidad. (a) Páginas ordenadas con base en FIFO. (b) Lista de las páginas si ocurre un fallo de página en el tiempo 20 y *A* tiene su bit *R* activado. Los números encima de las páginas son sus tiempos de carga.

Suponga que ocurre un fallo de página en el tiempo 20. La página más antigua es *A*, que llegó en el tiempo 0, cuando se inició el proceso. Si el bit *R* de *A* esta desactivado, se desaloja de la memoria, ya sea escribiéndola en el disco (si está sucia) o sólo se abandona (si está limpia). Por otro lado, si el bit *R* está activado, *A* se coloca al final de la lista y su "tiempo de carga" se restablece al tiempo actual (20). El bit *R* también esta desactivado. La búsqueda de una página adecuada continúa con *B*.

Lo que el algoritmo de segunda oportunidad está buscando es una página antigua a la que no se haya hecho referencia en el intervalo de reloj más reciente. Si se ha hecho referencia a todas las páginas, el algoritmo segunda oportunidad se degenera y se convierte en FIFO puro. De manera específica, imagine que todas las páginas en la figura 3-15(a) tienen sus bits *R* activados. Una por una, el sistema operativo mueve las páginas al final de la lista, desactivando el bit *R* cada vez que adjunta una página al final de la lista. En un momento dado regresará a la página *A*, que ahora tiene su bit *R* desactivado. En este punto, *A* se desaloja. Por ende, el algoritmo siempre termina.

3.4.5 El algoritmo de reemplazo de páginas: reloj

Aunque el algoritmo segunda oportunidad es razonable, también es innecesariamente ineficiente debido a que está moviendo constantemente páginas en su lista. Un mejor método sería mantener todos los marcos de página en una lista circular en forma de reloj, como se muestra en la figura 3-16. La manecilla apunta a la página más antigua.

Cuando ocurre un fallo de página, la página a la que apunta la manecilla se inspecciona. Si el bit *R* es 0, la página se desaloja, se inserta la nueva página en el reloj en su lugar y la manecilla se avanza

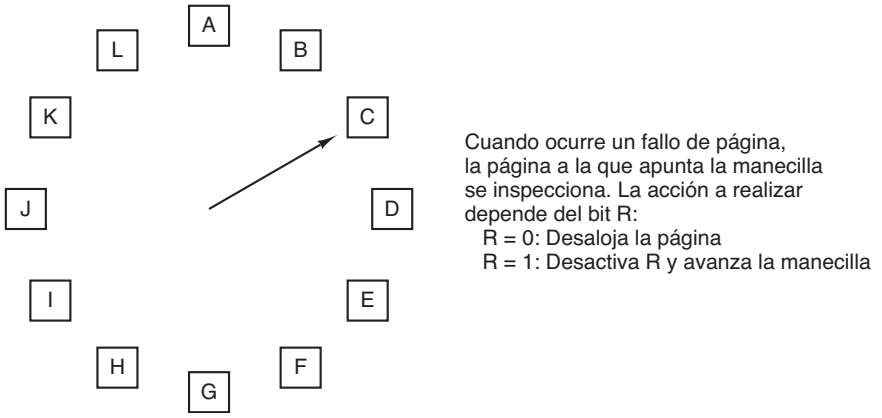

Figura 3-16. El algoritmo de reemplazo de páginas en reloj.

una posición. Si *R* es 1, se borra y la manecilla se avanza a la siguiente página. Este proceso se repite hasta encontrar una página con $R = 0$. No es de sorprender que a este algoritmo se le llame en **reloj**.

3.4.6 El algoritmo de reemplazo de páginas: menos usadas recientemente (LRU)

Una buena aproximación al algoritmo óptimo se basa en la observación de que las páginas que se hayan utilizado con frecuencia en las últimas instrucciones, tal vez se volverán a usar con frecuencia en las siguientes. Por el contrario, las páginas que no se hayan utilizado por mucho tiempo probablemente seguirán sin utilizarse por mucho tiempo más. Esta idea sugiere un algoritmo factible: cuando ocurra un fallo de página, hay que descartar la página que no se haya utilizado durante la mayor longitud de tiempo. A esta estrategia se le conoce como paginación **LRU** (*Least Recently Used*, Menos usadas recientemente).

Aunque en teoría el algoritmo LRU es realizable, no es barato. Para implementar el LRU por completo, es necesario mantener una lista enlazada de todas las páginas en memoria, con la página de uso más reciente en la parte frontal y la de uso menos reciente en la parte final. La dificultad es que la lista debe actualizarse en cada referencia a memoria. Buscar una página en la lista, eliminarla y después pasarla al frente es una operación que consume mucho tiempo, incluso mediante hardware (suponiendo que pudiera construirse dicho hardware).

Sin embargo, hay otras formas de implementar el algoritmo LRU con hardware especial. Consideremos primero la forma más simple. Este método requiere equipar el hardware con un contador de 64 bits, llamado *C*, el cual se incrementa de manera automática después de cada instrucción. Además, cada entrada en la tabla de páginas debe tener también un campo lo bastante grande como para poder guardar el contador. Después de cada referencia a memoria, el valor actual de *C* se almacena en la entrada en la tabla de páginas para la página que se acaba de referenciar. Cuando ocurre un fallo de página, el sistema operativo examina todos los contadores en la tabla de páginas para encontrar el menor. Esa página es la de uso menos reciente.

Ahora veamos un segundo algoritmo LRU mediante hardware. Para una máquina con *n* marcos de página, el hardware del LRU puede mantener una matriz de *n* x *n* bits (inicialmente, todos son 0). Cada vez que se hace referencia a la página *k*, el hardware primero establece todos los bits de la fila *k* en 1 y después todos los bits de la columna k en 0. En cualquier instante, la fila cuyo valor binario sea menor es la de uso menos reciente, la fila cuyo valor sea el siguiente más bajo es la del siguiente uso menos reciente, y así en lo sucesivo. El funcionamiento de este algoritmo se muestra en la figura 3-17 para cuatro marcos de página y referencias a páginas en el siguiente orden:

0 1 2 3 2 1 0 3 2 3

Después de hacer referencia a la página 0, tenemos la situación de la figura 3-17(a). Después de hacer referencia a la página 1, tenemos la situación de la figura 3-17(b), y así en lo sucesivo.

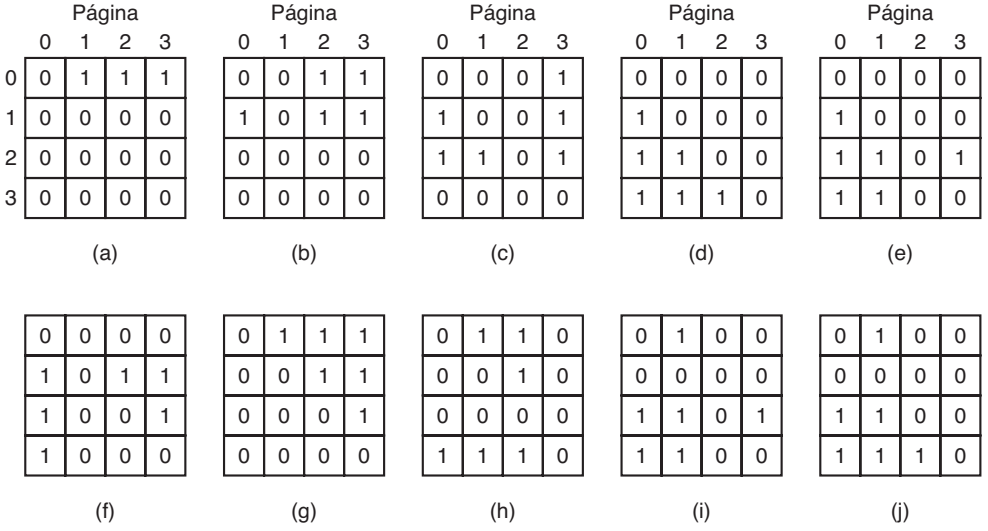

Figura 3-17. LRU usando una matriz cuando se hace referencia a las páginas en el orden 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

3.4.7 Simulación de LRU en software

Aunque los dos algoritmos LRU anteriores son (en principio) realizables, muy pocas máquinas (si acaso) tienen el hardware requerido. En vez de ello, se necesita una solución que puede implementarse en software. Una posibilidad es el algoritmo **NFU** (*Not Frequently Used*, No utilizadas frecuentemente). Este algoritmo requiere un contador de software asociado con cada página, que al principio es cero. En cada interrupción de reloj, el sistema operativo explora todas las páginas en memoria. Para cada página se agrega el bit *R*, que es 0 o 1, al contador. Los contadores llevan la cuenta aproximada de la frecuencia con que se hace referencia a cada página. Cuando ocurre un fallo de página, se selecciona la página con el contador que sea menor para sustituirla.

El principal problema con el algoritmo NFU es que nunca olvida nada. Por ejemplo, en un compilador con varias pasadas, las páginas que se utilizaron con frecuencia durante la pasada 1 podrían

tener todavía una cuenta alta en las siguientes pasadas. De hecho, si la pasada 1 tiene el tiempo de ejecución más largo de todas las pasadas, las páginas que contienen el código para las pasadas subsiguientes pueden tener siempre cuentas menores que las páginas de la pasada 1. En consecuencia, el sistema operativo eliminará páginas útiles, en vez de páginas que ya no se hayan utilizado.

Por fortuna, una pequeña modificación al algoritmo NFU le permite simular el algoritmo LRU muy bien. La modificación consta de dos partes. Primero, cada uno de los contadores se desplaza a la derecha 1 bit antes de agregar el bit *R*. Después, el bit *R* se agrega al bit de más a la izquierda, en lugar de sumarse al bit de más a la derecha.

La figura 3-18 ilustra cómo funciona el algoritmo modificado, conocido como **envejecimiento** (*aging*). Suponga que después del primer pulso de reloj los bits *R* para las páginas de la 0 a la 5 tienen los valores 1, 0, 1, 0, 1 y 1, respectivamente (la página 0 es 1, la página 1 es 0, la página 2 es 1, etc.). En otras palabras, entre los pulsos 0 y 1 se hizo referencia a las páginas 0, 2, 4 y 5, estableciendo sus bits *R* en 1, mientras que los de las otras páginas se quedaron en 0. Después de que se han desplazado los seis contadores correspondientes y el bit *R* se ha insertado a la izquierda, tienen los valores que se muestran en la figura 3-18(a). Las cuatro columnas restantes muestran los seis contadores después de los siguientes cuatro pulsos de reloj.

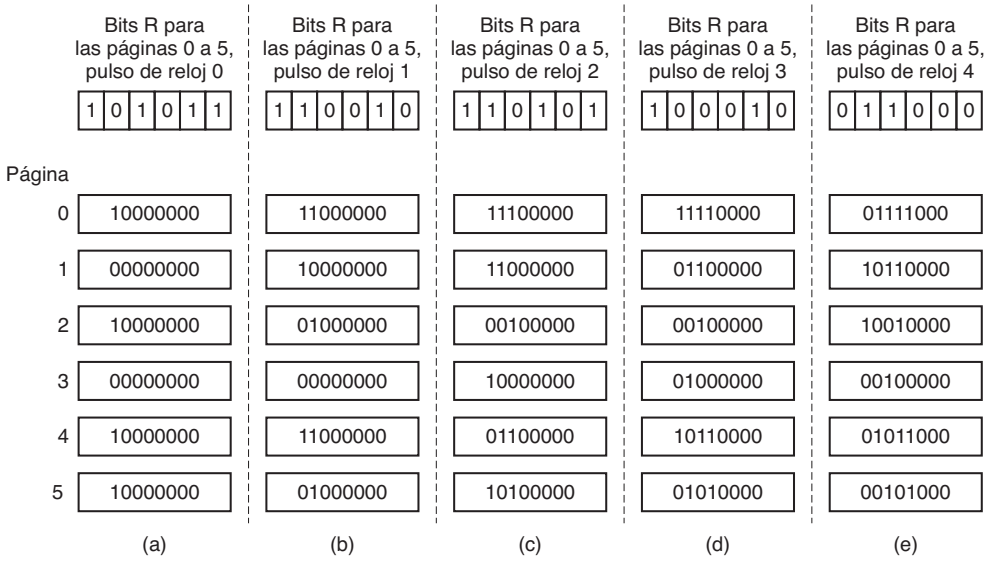

Figura 3-18. El algoritmo de envejecimiento simula el LRU en software. Aquí se muestran seis páginas para cinco pulsos de reloj. Los cinco pulsos de reloj se representan mediante los incisos (a) a (e).

Cuando ocurre un fallo de página, se elimina la página cuyo contador sea el menor. Está claro que una página a la que no se haya hecho referencia durante, por ejemplo, cuatro pulsos de reloj, tendrá cuatro ceros a la izquierda en su contador, y por ende tendrá un valor menor que un contador al que no se haya hecho referencia durante tres pulsos de reloj.

Este algoritmo difiere del de LRU en dos formas. Considere las páginas 3 y 5 en la figura 3-18(e). Ninguna se ha referenciado durante dos pulsos de reloj; ambas se referenciaron en el

pulso anterior a esos dos. De acuerdo con el algoritmo LRU, si hay que sustituir una página, debemos elegir una de estas dos. El problema es que no sabemos a cuál de ellas se hizo referencia por última vez en el intervalo entre el pulso 1 y el 2. Al registrar sólo un bit por cada intervalo de tiempo, hemos perdido la capacidad de diferenciar las referencias anteriores en el intervalo de reloj de las que ocurrieron después. Todo lo que podemos hacer es eliminar la página 3, debido a que también se hizo referencia a la página 5 dos pulsos de reloj antes, y a la página 3 no.

La segunda diferencia entre los algoritmos de LRU y envejecimiento es que en este último los contadores tienen un número finito de bits (8 en este ejemplo), lo cual limita su horizonte pasado. Suponga que dos páginas tienen cada una un valor de 0 en su contador; todo lo que podemos hacer es elegir una de ellas al azar. En realidad, bien podría ser una de las páginas a las que se haya hecho referencia nueve pulsos atrás, y la otra 1000 pulsos atrás. No tenemos forma de determinarlo. Sin embargo, en la práctica bastan 8 bits si un pulso de reloj es de aproximadamente 20 milisegundos. Si no se ha hecho referencia a una página en 160 milisegundos, tal vez no sea tan importante.

3.4.8 El algoritmo de reemplazo de páginas: conjunto de trabajo

En la forma más pura de paginación, los procesos inician sin ninguna de sus páginas en la memoria. Tan pronto como la CPU trata de obtener la primera instrucción, recibe un fallo de página, causando que el sistema operativo tenga que traer la página que contiene la primera instrucción. Por lo general a este fallo le siguen otros fallos de página por las variables globales y la pila. Después de cierto tiempo, el proceso tiene la mayoría de las páginas que necesita y se establece para ejecutarse con relativamente pocos fallos de página. A esta estrategia se le conoce como **paginación bajo demanda**, debido a que las páginas se cargan sólo según la demanda, no por adelantado.

Desde luego que es fácil escribir un programa de prueba que lea de manera sistemática todas las páginas en un espacio de direcciones extenso, produciendo tantos fallos de página que no haya suficiente memoria como para contenerlos todos. Por fortuna, la mayoría de los procesos no trabajan de esta manera. Exhiben una **localidad de referencia**, lo cual significa que durante cualquier fase de ejecución el proceso hace referencia sólo a una fracción relativamente pequeña de sus páginas. Por ejemplo, cada pasada de un compilador con varias pasadas hace referencia sólo a una fracción de todas las páginas y cada vez es una fracción distinta.

El conjunto de páginas que utiliza un proceso en un momento dado se conoce como su **conjunto de trabajo** (Denning, 1968a; Denning, 1980). Si todo el conjunto de trabajo está en memoria, el proceso se ejecutará sin producir muchos fallos hasta que avance a otra fase de ejecución (por ejemplo, la siguiente pasada del compilador). Si la memoria disponible es demasiado pequeña como para contener todo el conjunto de trabajo completo, el proceso producirá muchos fallos de página y se ejecutará lentamente, ya que para ejecutar una instrucción se requieren unos cuantos nanosegundos y para leer una página del disco se requieren en general 10 milisegundos. A una proporción de una o dos instrucciones por cada 10 milisegundos, se requeriría una eternidad para terminar. Se dice que un programa que produce fallos de página cada pocas instrucciones está **sobrepaginando** (***thrashing***) (Denning, 1968b).

En un sistema de multiprogramación, los procesos se mueven con frecuencia al disco (es decir, todas sus páginas se eliminan de la memoria) para dejar que otros procesos tengan oportunidad de

usar la CPU. Aquí surge la pregunta de qué hacer cuando un proceso se regresa una y otra vez. Técnicamente, no hay que hacer nada. El proceso producirá fallos de página sólo hasta que se haya cargado su conjunto de trabajo. El problema es que tener 20, 100 o incluso 1000 fallos de página cada vez que se carga un proceso es algo lento, además de que se desperdicia un tiempo considerable de la CPU, ya que el sistema operativo tarda unos cuantos milisegundos de tiempo de la CPU en procesar un fallo de página.

Por lo tanto, muchos sistemas de paginación tratan de llevar la cuenta del conjunto de trabajo de cada proceso y se aseguran que esté en memoria antes de permitir que el proceso se ejecute. Este método se conoce como **modelo del conjunto de trabajo** (Denning, 1970). Está diseñado para reducir en gran parte la proporción de fallos de página. Al proceso de cargar las páginas *antes* de permitir que se ejecuten los procesos también se le conoce como **prepaginación**. Tenga en cuenta que el conjunto de trabajo cambia con el tiempo.

Desde hace mucho se sabe que la mayor parte de los programas no hacen referencia a su espacio de direcciones de manera uniforme, sino que las referencias tienden a agruparse en un pequeño número de páginas. Una referencia a memoria puede obtener una instrucción, puede obtener datos o puede almacenar datos. En cualquier instante de tiempo *t*, existe un conjunto consistente de todas las páginas utilizadas por las *k* referencias a memoria más recientes. Este conjunto $w(k, t)$, es el de trabajo. Debido a que las $k = 1$ referencias más recientes deben haber utilizado todas las páginas utilizadas por las $k > 1$ referencias más recientes y tal vez otras, $w(k, t)$ es una función de *k* monotónicamente no decreciente. El límite de $w(k, t)$ a medida que *k* crece es finito, debido a que un programa no puede hacer referencia a más páginas de las que contiene su espacio de direcciones, y pocos programas utilizarán cada una de las páginas. La figura 3-19 muestra el tamaño del conjunto de trabajo como una función de *k*.

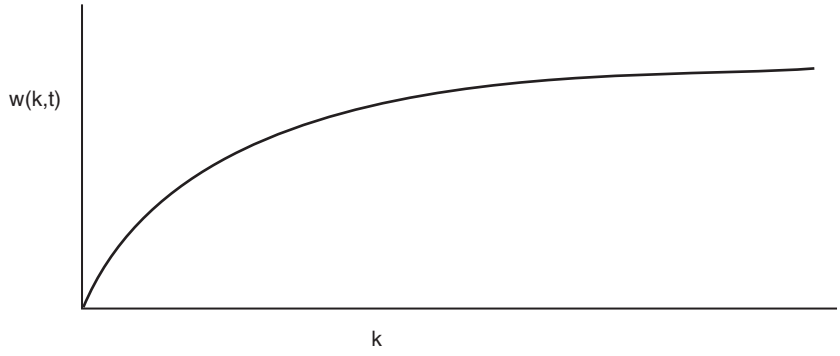

Figura 3-19. El conjunto de trabajo es el conjunto de páginas utilizadas por las *k* referencias a memoria más recientes. La función $w(k, t)$ es el tamaño del conjunto de trabajo en el tiempo *t*.

El hecho de que la mayoría de los programas acceden de manera aleatoria a un pequeño número de páginas, pero que este conjunto cambia lentamente con el tiempo, explica la rápida elevación inicial de la curva y después la lenta elevación para una *k* grande. Por ejemplo, un programa que ejecuta un ciclo que ocupa dos páginas utilizando datos en cuatro páginas podría hacer referencia a las seis páginas cada 1000 instrucciones, pero la referencia más reciente a alguna otra página po-

dría ser un millón de instrucciones antes, durante la fase de inicialización. Debido a este comportamiento asintótico, el contenido del conjunto de trabajo no es sensible al valor elegido de *k*. Dicho en forma distinta, existe un amplio rango de valores *k* para los cuales el conjunto de trabajo no cambia. Debido a que el conjunto de trabajo varía lentamente con el tiempo, es posible realizar una predicción razonable en cuanto a qué páginas se necesitarán cuando el programa se reinicie, con base en su conjunto de trabajo la última vez que se detuvo. La prepaginación consiste en cargar estas páginas antes de reanudar el proceso.

Para implementar el modelo del conjunto de trabajo, es necesario que el sistema operativo lleve la cuenta de cuáles páginas están en el conjunto de trabajo. Tener esta información también nos conduce de inmediato a un posible algoritmo de reemplazo de páginas: cuando ocurra un fallo de página, hay que buscar una página que no se encuentre en el conjunto de trabajo y desalojarla. Para implementar dicho algoritmo se requiere una manera precisa de determinar cuáles páginas están en el conjunto de trabajo. Por definición, el conjunto de trabajo es el conjunto de páginas utilizadas en las *k* referencias a memoria más recientes (algunos autores utilizan las *k* referencias a páginas más recientes, pero la elección es arbitraria). Para implementar cualquier algoritmo de conjunto de trabajo se debe elegir por adelantado cierto valor de *k*. Una vez que se ha seleccionado cierto valor, después de cada referencia a memoria, el conjunto de páginas utilizadas por las *k* referencias a memoria más recientes se determina en forma única.

Desde luego que tener una definición operacional del conjunto de trabajo no significa que haya una forma eficiente de calcularlo durante la ejecución de un programa. Uno podría imaginar un registro de desplazamiento de longitud *k*, donde cada referencia a memoria desplazará el registro una posición a la izquierda e insertará el número de página de referencia más reciente a la derecha. El conjunto de todos los *k* números de página en el registro de desplazamiento sería el conjunto de trabajo. En teoría, en un fallo de página, el contenido del registro de desplazamiento se podría extraer y ordenar; las páginas duplicadas entonces pueden ser eliminadas. El resultado sería el conjunto de trabajo. Sin embargo, el proceso de mantener el registro de desplazamiento y procesarlo en un fallo de página sería en extremo costoso, por lo que esta técnica nunca se utiliza.

En vez de ello, se utilizan varias aproximaciones. Una de uso común es desechar la idea de contar hacia atrás *k* referencias de memoria y usar en su defecto el tiempo de ejecución. Por ejemplo, en vez de definir el conjunto de trabajo como las páginas utilizadas durante los 10 millones de referencias a memoria anteriores, podemos definirlo como el conjunto de páginas utilizadas durante los últimos 100 milisegundos de tiempo de ejecución. En la práctica, dicha definición es igual de conveniente y es mucho más fácil trabajar con ella. Observe que para cada proceso sólo cuenta su propio tiempo de ejecución. Así, si un proceso empieza su ejecución en el tiempo T y ha tenido 40 milisegundos de tiempo de la CPU a un tiempo real de $T + 100$ milisegundos, para los fines del conjunto de trabajo su tiempo es de 40 mseg. La cantidad de tiempo de la CPU que ha utilizado en realidad un proceso desde que empezó se conoce comúnmente como su **tiempo virtual actual**. Con esta aproximación, el conjunto de trabajo de un proceso es el conjunto de páginas a las que se ha hecho referencia durante los últimos τ segundos de tiempo virtual.

Ahora veamos un algoritmo de reemplazo de páginas basado en el conjunto de trabajo. La idea básica es buscar una página que no esté en el conjunto de trabajo y desalojarla. En la figura 3-20 vemos una porción de una tabla de páginas para cierta máquina. Ya que sólo las páginas que están en memoria se consideran como candidatos para el desalojo, este algoritmo ignora las páginas que

no están en la memoria. Cada entrada contiene (al menos) dos elementos clave de información: el tiempo (aproximado) que se utilizó la página por última vez y el bit *R* (referenciada). El rectángulo vacío de color blanco simboliza los demás campos que no necesita este algoritmo, como el número de marco de página, los bits de protección y el bit *M* (modificado).

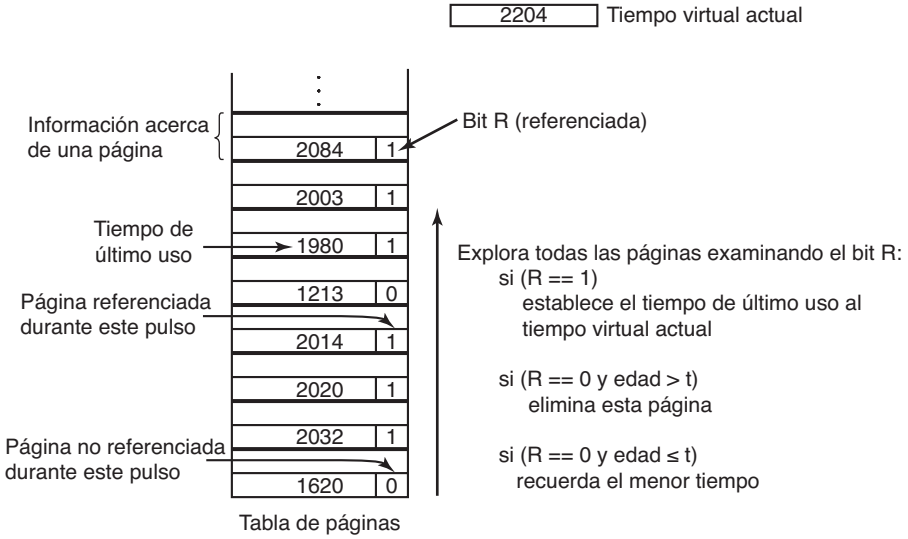

Figura 3-20. El algoritmo del conjunto de trabajo.

El algoritmo funciona de la siguiente manera. Se supone que el hardware debe establecer los bits *R* y *M*, como vimos antes. De manera similar, se supone que una interrupción periódica de reloj hace que se ejecute software para borrar el bit *Referenciada* en cada pulso de reloj. En cada fallo de página se explora la tabla de páginas en busca de una página adecuada para desalojarla.

A medida que se procesa cada entrada, se examina el bit *R*. Si es 1, el tiempo virtual actual se escribe en el campo *Tiempo de último uso* en la tabla de páginas, indicando que la página estaba en uso al momento en que ocurrió el fallo de página. Como se hizo referencia a la página durante el pulso de reloj actual, es evidente que está en el conjunto de trabajo y no es candidata para la eliminación (se supone que τ abarca varios pulsos de reloj).

Si *R* es 0, no se ha hecho referencia a la página durante el pulso de reloj actual y puede ser candidata para la eliminación. Para ver si debe o no eliminarse, se calcula su edad (el tiempo virtual actual menos su *Tiempo de último uso*) y se compara con τ. Si la edad es mayor que τ, la página ya no se encuentra en el conjunto de trabajo y la nueva página la sustituye. La exploración continúa actualizando las entradas restantes.

No obstante, si *R* es 0 pero la edad es menor o igual que τ, la página está todavía en el conjunto de trabajo. La página se reserva temporalmente, pero se apunta la página con la mayor edad (el menor valor de *Tiempo de último uso*). Si toda la tabla completa se explora sin encontrar un candidato para desalojar, eso significa que todas las páginas están en el conjunto de trabajo. En ese caso, si se encontraron una o más páginas con $R = 0$, se desaloja la más antigua. En el peor caso, se ha hecho

referencia a todas las páginas durante el pulso de reloj actual (y por ende, todas tienen $R = 1$), por lo que se selecciona una al azar para eliminarla, de preferencia una página limpia, si es que existe.

3.4.9 El algoritmo de reemplazo de páginas WSClock

Al algoritmo básico del conjunto de trabajo es incómodo ya que exige explorar toda la tabla de páginas en cada fallo de página hasta localizar un candidato adecuado. Un algoritmo mejorado, basado en el algoritmo de reloj pero que también utiliza la información del conjunto de trabajo, se conoce como **WSClock** (Carr y Hennessey, 1981). Debido a su simplicidad de implementación y buen rendimiento, es muy utilizado en la práctica.

La estructura de datos necesaria es una lista circular de marcos de página, como en el algoritmo de reloj, mostrada en la figura 3-21(a). Al principio, esta lista está vacía. Cuando se carga la primera página, se agrega a la lista. A medida que se agregan más páginas, pasan a la lista para formar un anillo. Cada entrada contiene el campo *Tiempo de último uso* del algoritmo básico del conjunto de trabajo, así como el bit *R* (mostrado) y el bit *M* (no mostrado).

Al igual que con el algoritmo de reloj, en cada fallo de página se examina primero la página a la que apunta la manecilla. Si el bit *R* es 1, la página se ha utilizado durante el pulso actual, por lo que no es candidata ideal para la eliminación. Después el bit *R* se establece en 0, la manecilla se avanza a la siguiente página y se repite el algoritmo para esa página. El estado después de esta secuencia de eventos se muestra en la figura 3-21(b).

Ahora considere lo que ocurre si la página a la que apunta la manecilla tiene $R = 0$, como se muestra en la figura 3-21(c). Si la edad es mayor que τ y la página está limpia, significa que no se encuentra en el conjunto de trabajo y existe una copia válida en el disco. El marco de página simplemente se reclama y la nueva página se coloca ahí, como se muestra en la figura 3-21(d). Por otro lado, si la página está sucia no se puede reclamar de inmediato, ya que no hay una copia válida presente en el disco. Para evitar una conmutación de procesos, la escritura al disco se planifica pero la manecilla avanza y el algoritmo continúa con la siguiente página. Después de todo, podría haber una página antigua y limpia más allá de la línea que se pueda utilizar de inmediato.

En principio, todas las páginas se podrían planificar para la E/S de disco en un ciclo alrededor del reloj. Para reducir el tráfico de disco se podría establecer un límite para permitir que se escriban de vuelta un máximo de *n* páginas. Una vez que se llega a este límite, no se planifican nuevas escrituras.

¿Qué ocurre si la manecilla llega otra vez a su punto inicial? Hay dos casos a considerar:

1. Se ha planificado por lo menos una escritura.
2. No se han planificado escrituras.

En el primer caso, la manecilla sólo sigue moviéndose, buscando una página limpia. Como se han planificado una o más escrituras, en algún momento se completará alguna escritura y su página se marcará como limpia. La primera página limpia que se encuentre se desaloja. Esta página no es necesariamente la primera escritura planificada, ya que el controlador de disco puede reordenar escrituras para poder optimizar el rendimiento del disco.

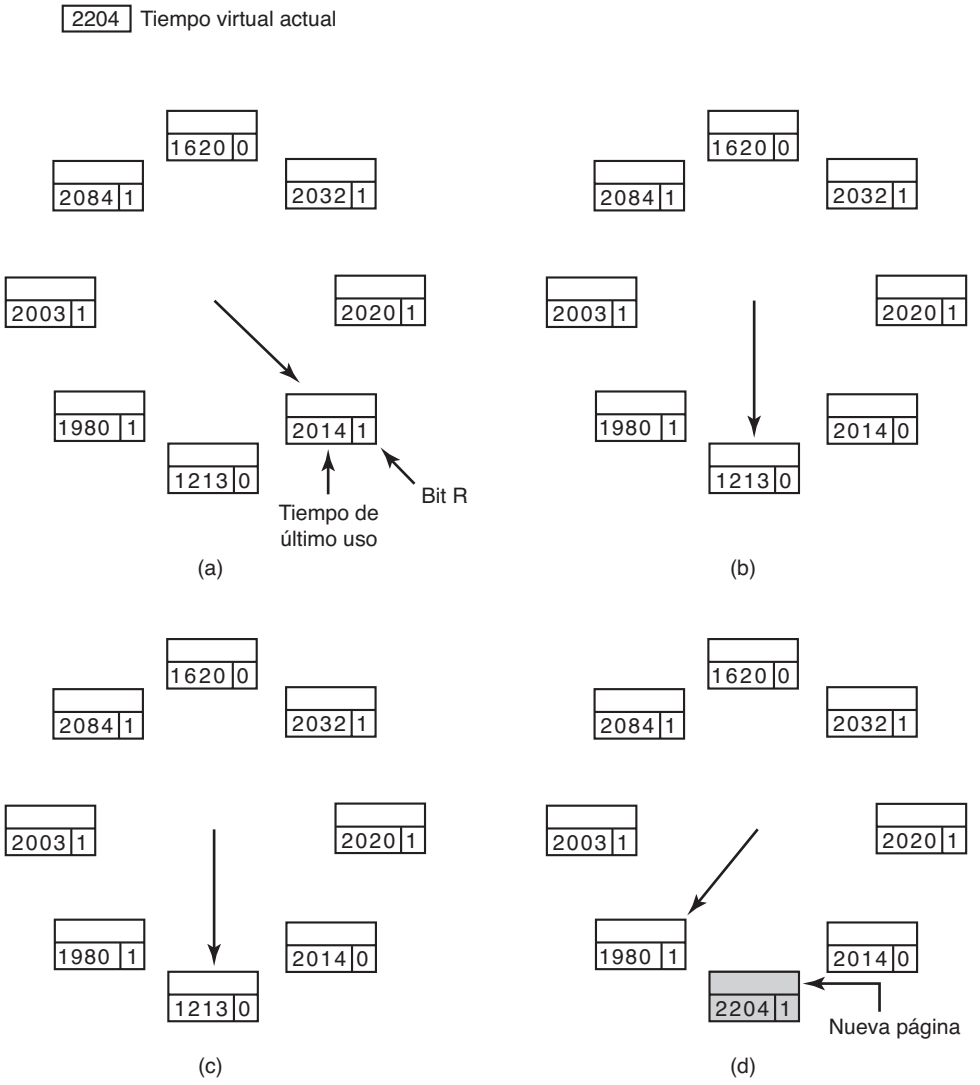

Figura 3-21. Operación del algoritmo WSClock. (a) y (b) dan un ejemplo de lo que ocurre cuando $R = 1$. (c) y (d) dan un ejemplo de cuando $R = 0$.

En el segundo caso, todas las páginas están en el conjunto de trabajo, de otra manera se hubiera planificado por lo menos una escritura. Sin información adicional, lo más simple por hacer es reclamar cualquier página limpia y usarla. La ubicación de una página limpia podría rastrearse durante el barrido. Si no existen páginas limpias, entonces se selecciona la página actual como la víctima y se escribe de vuelta al disco.

3.4.10 Resumen de los algoritmos de reemplazo de páginas

Ahora hemos visto una variedad de algoritmos de reemplazo de páginas. En esta sección mostraremos un breve resumen de ellos. La lista de algoritmos descritos se proporciona en la figura 3-22.

Algoritmo	Comentario
Óptimo	No se puede implementar, pero es útil como punto de comparación
NRU (No usadas recientemente)	Una aproximación muy burda del LRU
FIFO (primera en entrar, primera en salir)	Podría descartar páginas importantes
Segunda oportunidad	Gran mejora sobre FIFO
Reloj	Realista
LRU (menos usadas recientemente)	Excelente, pero difícil de implementar con exactitud
NFU (no utilizadas frecuentemente)	Aproximación a LRU bastante burda
Envejecimiento	Algoritmo eficiente que se aproxima bien a LRU
Conjunto de trabajo	Muy costoso de implementar
WSClock	Algoritmo eficientemente bueno

Figura 3-22. Algoritmos de reemplazo de páginas descritos en el texto.

El algoritmo óptimo desaloja la página a la que se hará referencia en el futuro más lejano. Por desgracia, no hay forma de determinar cuál página es, por lo que en la práctica no se puede utilizar este algoritmo. Sin embargo, es útil como punto de comparación para los demás algoritmos.

El algoritmo NRU divide las páginas en cuatro clases dependiendo del estado de los bits *R* y *M*. Se selecciona una página aleatoria de la clase con menor numeración. Este algoritmo es fácil de implementar, pero muy burdo. Existen mejores.

FIFO lleva la cuenta del orden en el que se cargaron las páginas en memoria al mantenerlas en una lista enlazada. Eliminar la página más antigua se vuelve entonces un proceso trivial, pero como esa página podría estar todavía en uso, FIFO es una mala opción.

El algoritmo de segunda oportunidad es una modificación de FIFO que comprueba si hay una página en uso antes de eliminarla. Si lo está, la página se reserva. Esta modificación mejora de manera considerable el rendimiento. El algoritmo de reloj es simplemente una implementación distinta del algoritmo de segunda oportunidad. Tiene las mismas propiedades de rendimiento, pero toma un poco menos de tiempo para ejecutar el algoritmo.

LRU es un algoritmo excelente, pero no se puede implementar sin hardware especial. NFU es un burdo intento por aproximarse a LRU; sin embargo, el algoritmo de envejecimiento es una mucho mejor aproximación a LRU y se puede implementar con eficiencia. Es una buena opción.

Los últimos dos algoritmos utilizan el conjunto de trabajo. El algoritmo del conjunto de trabajo ofrece un rendimiento razonable, pero es un poco costoso de implementar. WSClock es una variante que no sólo da un buen rendimiento, sino que también es eficiente al implementar.

Con todo, los dos mejores algoritmos son el de envejecimiento y WSClock. Se basan en LRU y el conjunto de trabajo, respectivamente. Ambos dan un buen rendimiento en la paginación y pueden implementarse con eficiencia. Existen varios algoritmos más, pero estos dos son tal vez los más importantes en la práctica.

3.5 CUESTIONES DE DISEÑO PARA LOS SISTEMAS DE PAGINACIÓN

En las secciones anteriores hemos explicado cómo funciona la paginación y hemos analizado unos cuantos de los algoritmos básicos de reemplazo de páginas, además de mostrar cómo modelarlos. Pero no basta con conocer la mecánica básica. Para diseñar un sistema hay que saber mucho más para hacer que funcione bien. Es como la diferencia que hay entre saber mover las piezas de ajedrez y ser un buen jugador. En la siguientes secciones analizaremos otras cuestiones que deben considerar los diseñadores de sistemas operativos para poder obtener un buen rendimiento de un sistema de paginación.

3.5.1 Políticas de asignación local contra las de asignación global

En las secciones anteriores hemos analizado varios algoritmos para seleccionar una página y sustituirla al ocurrir un fallo. Una cuestión importante asociada con esta elección (que hemos dejado de lado cuidadosamente hasta ahora) es cómo se debe asignar la memoria entre los procesos ejecutables en competencia.

Dé un vistazo a la figura 3-23(a). En esta figura, tres procesos (*A*, *B* y *C*) componen el conjunto de procesos ejecutables. Suponga que *A* obtiene un fallo de página. ¿Debe el algoritmo de reemplazo de páginas tratar de encontrar la página de uso menos reciente, considerando sólo las seis páginas que están actualmente asignadas a *A* o debe considerar todas las páginas en memoria? Si sólo examina las páginas asignadas a *A*, la página con el menor valor de edad es *A5*, por lo que obtenemos la situación de la figura 3-23(b).

Por otro lado, si se va a eliminar la página con el menor valor de edad sin importar de quién sea, se elegirá la página *B3* y obtendremos la situación de la figura 3-23(c). Al algoritmo de la figura 3-23(b) se le considera de reemplazo de páginas **local**, mientras que al de la figura 3-23(c), un algoritmo **global**. Los algoritmos locales corresponden de manera efectiva a asignar a cada proceso una fracción fija de la memoria. Los algoritmos globales asignan marcos de página de manera dinámica entre los procesos ejecutables. Así, el número de marcos de página asignados a cada proceso varía en el tiempo.

En general, los algoritmos globales funcionan mejor, en especial cuando el tamaño del conjunto de trabajo puede variar durante el tiempo de vida de un proceso. Si se utiliza un algoritmo local y el conjunto de trabajo crece, se producirá una sobrepaginación, aun cuando haya muchos marcos de página libres. Si el conjunto de trabajo se reduce, los algoritmos locales desperdician memoria. Si se utiliza un algoritmo global, el sistema debe decidir en forma continua cuántos marcos de página asignar a cada proceso. Una manera es supervisar el tamaño del conjunto de trabajo, según lo

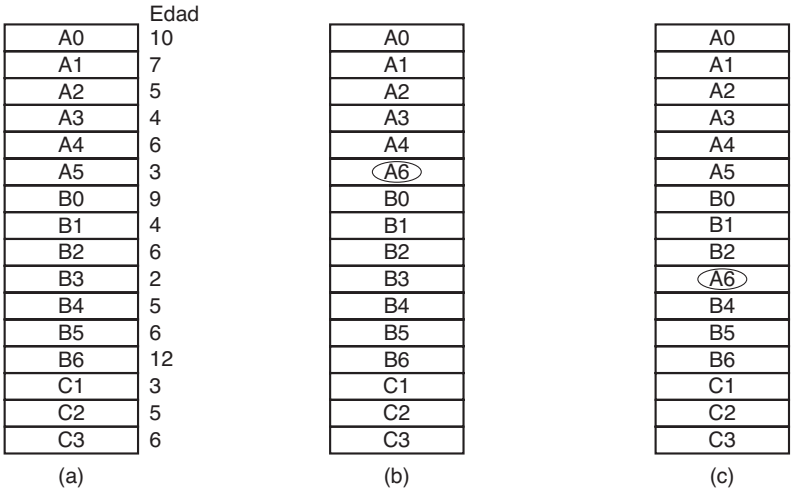

Figura 3-23. Comparación entre el reemplazo de páginas local y el global. (a) Configuración original. (b) Reemplazo de páginas local. (c) Reemplazo de páginas global.

indicado por los bits de envejecimiento, pero este método no necesariamente evita la sobrepaginación. El conjunto de trabajo puede cambiar de tamaño en microsegundos, mientras que los bits de envejecimiento son una medida burda, esparcida a través de varios pulsos de reloj.

Otro método es tener un algoritmo para asignar marcos de página a los procesos. Una manera es determinar periódicamente el número de procesos en ejecución y asignar a cada proceso una parte igual. Así, con 12,416 marcos de página disponibles (es decir, que no son del sistema operativo) y 10 procesos, cada proceso obtiene 1241 marcos. Los seis restantes pasan a una reserva, para utilizarlos cuando ocurran fallos de página.

Aunque este método parece equitativo, tiene poco sentido otorgar partes iguales de la memoria a un proceso de 10 KB y un proceso de 300 KB. En vez de ello, se pueden asignar páginas en proporción al tamaño total de cada proceso, donde un proceso de 300 KB obtiene 30 veces la asignación de un proceso de 10 KB. Probablemente sea prudente dar a cada proceso cierto número mínimo, de manera que se pueda ejecutar sin importar qué tan pequeño sea. Por ejemplo, en algunas máquinas una sola instrucción de dos operandos puede necesitar hasta seis páginas, debido a que la misma instrucción, el operando de origen y el operando de destino pueden extenderse a través de los límites de las páginas. Con una asignación de sólo cinco páginas, los programas que contengan tales instrucciones no se podrán ejecutar.

Si se utiliza un algoritmo global, es posible empezar cada proceso con cierto número de páginas proporcional al tamaño del proceso, pero la asignación se tiene que actualizar dinámicamente a medida que se ejecuten los procesos. Una manera de administrar la asignación es utilizando el algoritmo **PFF** (*Page Fault Frequency*, Frecuencia de fallo de páginas). Este algoritmo indica cuándo se debe incrementar o decrementar la asignación de páginas a un proceso, pero no dice nada acerca de cuál página se debe sustituir en un fallo. Sólo controla el tamaño del conjunto de asignación.

Para una clase extensa de algoritmos de reemplazo de páginas, incluyendo LRU, se sabe que la proporción de fallos disminuye a medida que se asignan más páginas, como lo describimos anteriormente. Ésta es la suposición detrás del algoritmo PFF. Esta propiedad se ilustra en la figura 3-24.

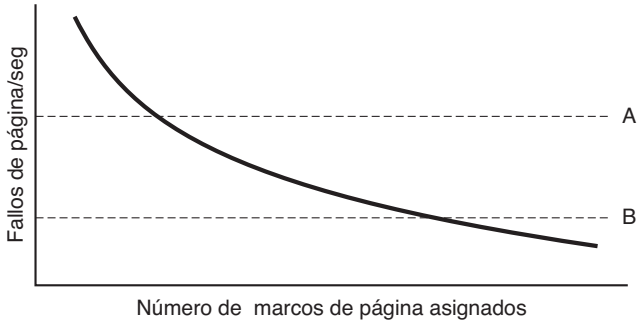

Figura 3-24. Proporción de fallos de página como una función del número de marcos de página asignados.

Medir la proporción de fallos de página es un proceso simple: sólo se cuenta el número de fallos por segundo, posiblemente tomando una media móvil sobre los segundos transcurridos también. Una manera sencilla de hacer esto es sumar el número de fallos de página durante el segundo inmediatamente anterior a la media móvil actual y dividir entre dos. La línea punteada marcada como *A* corresponde a una proporción de fallos de página que es demasiado alta, por lo que el proceso que emitió el fallo recibe más marcos de página para reducir la proporción de fallos. La línea punteada marcada como *B* corresponde a una proporción de fallos de página tan baja que podemos suponer que el proceso tiene demasiada memoria. En este caso se le pueden quitar marcos de página. Así, el algoritmo PFF trata de mantener la proporción de paginación para cada proceso dentro de límites aceptables.

Es importante recalcar que ciertos algoritmos de reemplazo de páginas pueden funcionar con una política de sustitución local o con una global. Por ejemplo, FIFO puede sustituir la página más antigua en toda la memoria (algoritmo global) o la más antigua que posea el proceso actual (algoritmo local). De manera similar, el algoritmo LRU o alguna aproximación a éste puede reemplazar la página usada menos recientemente en toda la memoria (algoritmo global) o la página menos usada recientemente poseída por el proceso actual (algoritmo local). La elección entre local y global es independiente del algoritmo en algunos casos.

Por otro lado, para los demás algoritmos de sustitución de página, sólo tiene sentido una estrategia local. En especial, los algoritmos del conjunto de trabajo y WSClock se refieren a un proceso específico y deben aplicarse en este contexto. En realidad no hay un conjunto de trabajo para la máquina como un todo, y al tratar de usar la unión de todos los conjuntos de trabajo se perdería la propiedad de localidad, y no funcionaría bien.

3.5.2 Control de carga

Aun con el mejor algoritmo de reemplazo de páginas y una asignación global óptima de marcos de página a los procesos, puede ocurrir que el sistema se sobrepagine. De hecho, cada vez que los con-

juntos de trabajo combinados de todos los procesos exceden a la capacidad de la memoria, se puede esperar la sobrepaginación. Un síntoma de esta situación es que el algoritmo PFF indica que algunos procesos necesitan más memoria, pero ningún proceso necesita menos memoria. En este caso no hay forma de proporcionar más memoria a esos procesos que la necesitan sin lastimar a algún otro proceso. La única solución real es deshacerse temporalmente de algunos procesos.

Una buena forma de reducir el número de procesos que compiten por la memoria es intercambiar algunos de ellos enviándolos al disco y liberar todas las páginas que ellos mantienen. Por ejemplo, un proceso puede intercambiarse al disco y sus marcos de página dividirse entre otros procesos que están sobrepaginando. Si el sobrepaginado se detiene, el sistema puede operar de esta forma por un tiempo. Si no se detiene hay que intercambiar otro proceso y así en lo sucesivo hasta que se detenga el sobrepaginado. Por ende, incluso hasta con la paginación se sigue necesitando el intercambio, sólo que ahora se utiliza para reducir la demanda potencial de memoria, en vez de reclamar páginas.

El proceso de intercambiar procesos para liberar la carga en la memoria es semejante a la planificación de dos niveles, donde ciertos procesos se colocan en disco y se utiliza un planificador de corto plazo para planificar los procesos restantes. Sin duda se pueden combinar las dos ideas donde se intercambien, fuera de memoria, sólo los procesos suficientes para hacer que la proporción de fallo de páginas sea aceptable. Periódicamente, ciertos procesos se traen del disco y otros se intercambian hacia el mismo.

Sin embargo, otro factor a considerar es el grado de multiprogramación. Cuando el número de procesos en la memoria principal es demasiado bajo, la CPU puede estar inactiva durante largos periodos. Esta consideración sostiene que no sólo se debe tomar en cuenta el tamaño del proceso y la proporción de paginación al decidir qué proceso se debe intercambiar, sino también sus características, tal como si está ligado a la CPU o a la E/S, así como las características que tienen los procesos restantes.

3.5.3 Tamaño de página

El tamaño de página es un parámetro que a menudo el sistema operativo puede elegir. Incluso si el hardware se ha diseñado, por ejemplo, con páginas de 512 bytes, el sistema operativo puede considerar fácilmente los pares de páginas 0 y 1, 2 y 3, 4 y 5, y así en lo sucesivo, como páginas de 1 KB al asignar siempre dos marcos de página de 512 bytes consecutivos para ellas.

Para determinar el mejor tamaño de página se requiere balancear varios factores competitivos. Como resultado, no hay un tamaño óptimo en general. Para empezar, hay dos factores que están a favor de un tamaño de página pequeño. Un segmento de texto, datos o pila elegido al azar no llenará un número integral de páginas. En promedio, la mitad de la página final estará vacía. El espacio adicional en esa página se desperdicia. A este desperdicio se le conoce como **fragmentación interna**. Con n segmentos en memoria y un tamaño de página de p bytes, se desperdiciarán $np/2$ bytes en fragmentación interna. Este razonamiento está a favor de un tamaño de página pequeño.

Otro argumento para un tamaño de página pequeño se hace aparente si consideramos que un programa consiste de ocho fases secuenciales de 4 KB cada una. Con un tamaño de página de 32 KB, se deben asignar 32 KB al programa todo el tiempo. Con un tamaño de página de 16 KB,

sólo necesita 16 KB. Con un tamaño de página de 4 KB o menor, sólo requiere 4 KB en cualquier instante. En general, un tamaño de página grande hará que haya una parte más grande no utilizada del programa que un tamaño de página pequeño.

Por otro lado, tener páginas pequeñas implica que los programas necesitarán muchas páginas, lo que sugiere la necesidad de una tabla de páginas grande. Un programa de 32 KB necesita sólo cuatro páginas de 8 KB, pero 64 páginas de 512 bytes. Las transferencias hacia y desde el disco son por lo general de una página a la vez, y la mayor parte del tiempo se debe al retraso de búsqueda y al retraso rotacional, por lo que para transferir una página pequeña se requiere casi el mismo tiempo que para transferir una página grande. Se podran requerir 64 x 10 mseg para cargar 64 páginas de 512 bytes, pero sólo 4 x 12 mseg para cargar cuatro páginas de 8 KB.

En algunas máquinas, la tabla de páginas se debe cargar en registros de hardware cada vez que la CPU cambia de un proceso a otro. En estas máquinas, tener un tamaño pequeño de página significa que el tiempo requerido para cargar sus registros aumenta a medida que se hace más pequeña. Además, el espacio ocupado por la tabla de páginas aumenta a medida que se reduce el tamaño de las páginas.

Este último punto se puede analizar matemáticamente. Digamos que el tamaño promedio de un proceso es de s bytes y que el tamaño de página es de p bytes. Además supone que cada entrada de página requiere e bytes. El número aproximado de páginas necesarias por proceso es entonces s/p, ocupando se/p bytes de espacio en la tabla de páginas. La memoria desperdiciada en la última página del proceso debido a la fragmentación interna es $p/2$. Así, la sobrecarga total debido a la tabla de páginas y a la pérdida por fragmentación interna se obtiene mediante la suma de estos dos términos:

$$\text{sobrecarga} = se/p + p/2$$

El primer término (tamaño de la tabla de páginas) es grande cuando el tamaño de página es pequeño. El segundo término (fragmentación interna) es grande cuando el tamaño de página es grande. El valor óptimo debe estar entre estos dos. Al sacar la primera derivada con respecto a p e igualarla a cero, obtenemos la ecuación

$$-se/p^2 + 1/2 = 0$$

De esta ecuación podemos derivar una fórmula que proporcione el tamaño de página óptimo (considerando sólo la memoria gastada en la fragmentación y el tamaño de la tabla de páginas). El resultado es:

$$p = \sqrt{2se}$$

Para $s = 1$ MB y $e = 8$ bytes por cada entrada en la tabla de páginas, el tamaño de página óptimo es de 4 KB. Las computadoras disponibles en forma comercial han utilizado tamaños de página que varían desde 512 bytes hasta 64 KB. Un valor común solía ser 1 KB, pero hoy en día es más común tener 4 KB u 8 KB. A medida que las memorias aumentan su tamaño, el tamaño de página tiende a crecer también (pero no en forma lineal). Cuadruplicar el tamaño de la RAM rara vez duplica siquiera el tamaño de página.

3.5.4 Espacios separados de instrucciones y de datos

La mayor parte de las computadoras tienen un solo espacio de direcciones que contiene tanto programas como datos, como se muestra en la figura 3-25(a). Si este espacio de direcciones es lo bastante grande, todo funciona bien. No obstante, a menudo es demasiado pequeño, lo cual obliga a los programadores a pararse de cabeza tratando de ajustar todo en el espacio de direcciones.

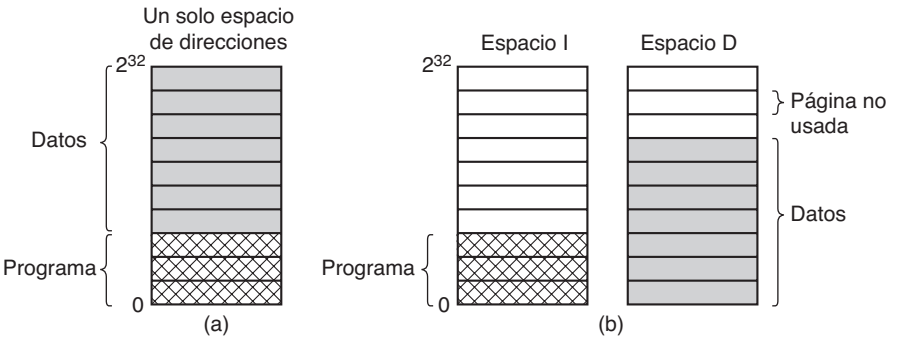

Figura 3-25. (a) Un espacio de direcciones. (b) Espacios I y D separados.

Una solución utilizada por primera vez en la PDP-11 (de 16 bits) es tener espacios de direcciones separados para las instrucciones (texto del programa) y los datos, llamados **espacio I** y **espacio D**, respectivamente, como se muestra en la figura 3-25(b). Cada espacio de direcciones empieza desde 0 hasta cierto valor máximo, por lo general de $2^{16} - 1$ o de $2^{32} - 1$. El enlazador debe saber cuándo se utilizan espacios I y D separados, ya que cuando esto ocurre los datos se reubican a la dirección virtual 0, en vez de empezar después del programa.

En una computadora con este diseño, ambos espacios de direcciones se pueden paginar de manera independiente. Cada uno tiene su propia tabla de páginas, con su propia asignación de páginas virtuales a marcos de páginas físicas. Cuando el hardware desea obtener una instrucción, sabe que debe utilizar el espacio I y la tabla de páginas de este espacio. De manera similar, las referencias a los datos deben pasar a través de la tabla de páginas del espacio D. Aparte de esta distinción, tener espacios I y D separados no introduce ninguna complicación especial y sí duplica el espacio de direcciones disponible.

3.5.5 Páginas compartidas

Otra cuestión de diseño es la compartición. En un sistema de multiprogramación grande, es común que varios usuarios ejecuten el mismo programa a la vez. Evidentemente es más eficiente compartir las páginas para evitar tener dos copias de la misma página en memoria al mismo tiempo. Un problema es que no todas las páginas se pueden compartir. En especial, sólo pueden compartirse las páginas que son de sólo lectura como el texto del programa, pero las páginas de datos no.

Si se admiten espacios I y D separados, es relativamente simple compartir los programas al hacer que dos o más procesos utilicen la misma tabla de páginas para su espacio I pero distintas

tablas de páginas para sus espacios D. Por lo general en una implementación que soporta la compartición de esta forma, las tablas de páginas son estructuras de datos independientes de la tabla de procesos. Entonces, cada proceso tiene dos apuntadores en su tabla de procesos: uno para la tabla de páginas del espacio I y otro para la tabla de páginas del espacio D, como se muestra en la figura 3-26. Cuando el planificador selecciona un proceso para ejecutarlo, utiliza estos apuntadores para localizar las tablas de páginas apropiadas y establece la MMU para que los utilice. Aun sin espacios I y D separados, los procesos pueden compartir programas (o algunas veces bibliotecas) pero el mecanismo es más complicado.

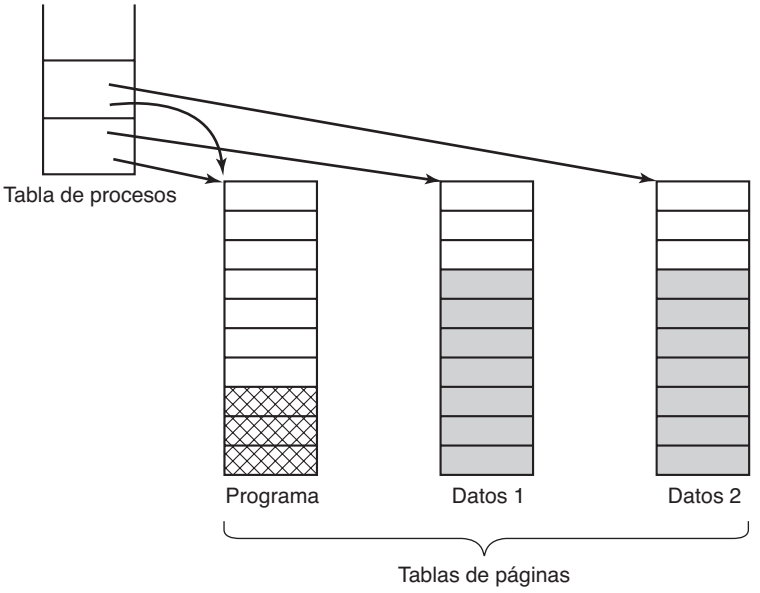

Figura 3-26. Dos procesos comparten el mismo programa compartiendo su tabla de páginas.

Cuando dos o más procesos comparten cierto código, ocurre un problema con las páginas compartidas. Suponga que los procesos *A* y *B* están ejecutando el editor y comparten sus páginas. Si el planificador decide eliminar *A* de la memoria, desalojando todas sus páginas y llenando los marcos de página vacíos con otro programa, causará que *B* genere un gran número de fallos de página para traerlas de vuelta.

De manera similar, cuando *A* termina, es esencial poder descubrir que las páginas aún están en uso, de manera que su espacio en el disco no se libere por accidente. Buscar en todas las tablas de páginas para ver si una página es compartida, frecuentemente es muy caro, por lo que se necesitan estructuras de datos especiales para llevar la cuenta de las páginas compartidas, en especial si la unidad de compartición es la página individual (o serie de páginas) en vez de toda una tabla de páginas completa.

Compartir datos es más complicado que compartir código, pero no imposible. En especial, en UNIX después de una llamada al sistema `fork`, el padre y el hijo tienen que compartir tanto el tex-

to del programa como el texto de los datos. En un sistema paginado, lo que se hace a menudo es dar a cada uno de estos procesos su propia tabla de páginas y hacer que ambos apunten al mismo conjunto de páginas. Así, no se realiza una copia de páginas al momento de la operación `fork`. Sin embargo, todas las páginas de datos son asociadas en ambos procesos de SÓLO LECTURA (READ ONLY).

Mientras que ambos procesos sólo lean sus datos, sin modificarlos, esta situación puede continuar. Tan pronto como cualquiera de los procesos actualiza una palabra de memoria, la violación de la protección de sólo lectura produce un trap al sistema operativo. Después se hace una copia de la página ofensora, para que cada proceso tenga ahora su propia copia privada. Ambas copias se establecen ahora como LECTURA-ESCRITURA (READ-WRITE), para que las siguientes operaciones de escritura en cualquiera de las copias continúen sin lanzar un trap. Esta estrategia significa que aquellas páginas que nunca se modifican (incluyendo todas las del programa) no se necesitan copiar. Este método, conocido como **copiar en escritura**, mejora el rendimiento al reducir el copiado.

3.5.6 Bibliotecas compartidas

La compartición se puede realizar en otros elementos además de las páginas individuales. Si un programa se inicia dos veces, la mayor parte de los sistemas operativos compartirán de manera automática todas las páginas de texto, quedando sólo una copia en la memoria. Las páginas de texto siempre son de sólo lectura, por lo que aquí no hay problema. Dependiendo del sistema operativo, cada proceso puede obtener su propia copia privada de las páginas de datos o se pueden compartir y marcar como de sólo lectura. Si cualquier proceso modifica una página de datos, se realizará una copia privada para éste, esto es, se aplicará la copia en escritura.

En los sistemas modernos hay muchas bibliotecas extensas utilizadas por muchos procesos, por ejemplo, la biblioteca que maneja el diálogo para explorar por archivos que se desean abrir y varias bibliotecas de gráficos. Si se enlazaran en forma estática estas bibliotecas con cada programa ejecutable en el disco se agrandarían aún más.

En vez de ello, una técnica común es utilizar **bibliotecas compartidas** (que se conocen como **DLL**s o **Bibliotecas de enlaces dinámicos** en Windows). Para aclarar la idea de una biblioteca compartida, primero considere el enlazamiento tradicional. Cuando un programa se enlaza, se nombra uno o más archivos de código objeto y posiblemente algunas bibliotecas en el comando para el enlazador, como el siguiente comando de UNIX:

```
ld *.o –lc –lm
```

el cual enlaza todos los *.o* (objeto) en el directorio actual y después explora dos bibliotecas, */usr/lib/libc.a* y */usr/lib/libm.a*. Las funciones a las que se llame en los archivos objeto pero que no estén ahí (por ejemplo, *printf*) se conocen como **externas indefinidas** y se buscan en las bibliotecas. Si se encuentran, se incluyen en el binario ejecutable. Cualquier función a la que llamen pero que no esté aún presente también se convierte en externa indefinida. Por ejemplo, *printf* necesita a *write*, por lo que si *write* no está ya incluida, el enlazador la buscará y la incluirá cuando la encuentre.

Cuando el enlazador termina, se escribe un archivo binario ejecutable en el disco que contiene todas las funciones necesarias. Las funciones presentes en la biblioteca, pero que no se llamaron, no se incluyen. Cuando el programa se carga en memoria y se ejecuta, todas las funciones que necesita están ahí.

Ahora suponga que los programas comunes usan de 20 a 50 MB de funciones de gráficos y de interfaz de usuario. Si se enlazaran de manera estática cientos de programas con todas estas bibliotecas se desperdiciaría una tremenda cantidad de espacio en el disco, además de desperdiciar espacio en la RAM a la hora de cargarlas, ya que el sistema no tendría forma de saber si puede compartirlas. Aquí es donde entran las bibliotecas compartidas. Cuando un programa se vincula con bibliotecas compartidas (que son ligeramente diferentes a las estáticas), en vez de incluir la función a la que se llamó, el vinculador incluye una pequeña rutina auxiliar que se enlaza a la función llamada en tiempo de ejecución. Dependiendo del sistema y los detalles de configuración, las bibliotecas compartidas se cargan cuando se carga el programa o cuando las funciones en ellas se llaman por primera vez. Desde luego que si otro programa ya ha cargado la biblioteca compartida, no hay necesidad de volver a cargarla; éste es el objetivo. Observe que cuando se carga o utiliza una biblioteca compartida, no se lee toda la biblioteca en memoria de un solo golpe. Se pagina una página a la vez según sea necesario, de manera que las funciones que no sean llamadas no se carguen en la RAM.

Además de reducir el tamaño de los archivos ejecutables y ahorrar espacio en memoria, las bibliotecas compartidas tienen otra ventaja: si una función en una biblioteca compartida se actualiza para eliminar un error, no es necesario recompilar los programas que la llaman, pues los antiguos binarios siguen funcionando. Esta característica es en especial importante para el software comercial, donde el código fuente no se distribuye al cliente. Por ejemplo, si Microsoft descubre y corrige un error de seguridad en alguna DLL estándar, *Windows Update* descargará la nueva DLL y sustituirá la anterior, y todos los programas que utilicen la DLL podrán usar de manera automática la nueva versión la próxima vez que se inicien.

Sin embargo, las bibliotecas compartidas tienen un pequeño problema que hay que resolver. El problema se ilustra en la figura 3-27. Aquí vemos dos procesos compartiendo una biblioteca de 20 KB en tamaño (suponiendo que cada cuadro sea de 4 KB). No obstante, la biblioteca está ubicada en una dirección distinta en cada proceso, tal vez debido a que los programas en sí no son del mismo tamaño. En el proceso 1, la biblioteca empieza en la dirección 36K; en el proceso 2 empieza en la 12K. Suponga que lo primero que hace la primera función en la biblioteca es saltar a la dirección 16 en la biblioteca. Si la biblioteca no fuera compartida, podría reubicarse al instante al momento de cargarla, de manera que el salto (en el proceso 1) pudiera ser a la dirección virtual 36K + 16. Observe que la dirección física en la RAM en donde se encuentra la biblioteca no importa, ya que todas las páginas son asociadas de direcciones virtuales a direcciones físicas por el hardware de la MMU.

Pero como la biblioteca es compartida, la reubicación instantánea no funcionará. Después de todo, cuando el proceso 2 llama a la primera función (en la dirección 12K), la instrucción de salto tiene que ir a 12K + 16, no a 36K + 16. Éste es el pequeño problema. Una manera de resolverlo es utilizar la copia en escritura y crear páginas para cada proceso que comparta la biblioteca, reubicándolas instantánemente a medida que se crean, pero es evidente que este esquema no sigue el propósito de compartir la biblioteca.

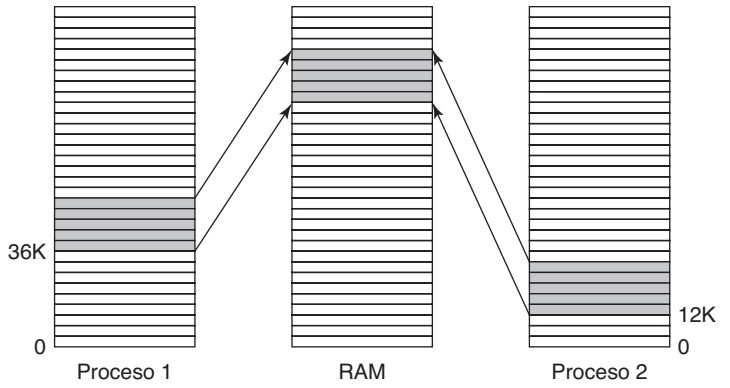

Figura 3-27. Una biblioteca compartida utilizada por dos procesos.

Una mejor solución es compilar las bibliotecas compartidas con una bandera de compilador especial, para indicar al compilador que no debe producir instrucciones que utilicen direcciones absolutas. En vez de ello, sólo se utilizan instrucciones con direcciones relativas. Por ejemplo, hay casi siempre una instrucción para saltar hacia adelante (o hacia atrás) por *n* bytes (en contraste a una instrucción que proporciona una dirección específica a la que se debe saltar). Esta instrucción funciona correctamente sin importar en dónde se coloque la biblioteca compartida en el espacio de direcciones virtuales. Al evitar direcciones absolutas, el problema se puede resolver. El código que utiliza sólo desplazamientos relativos se conoce como **código independiente de la posición**.

3.5.7 Archivos asociados

Las bibliotecas compartidas son realmente un caso de una herramienta más general, conocida como **archivos asociados a memoria**. La idea aquí es que un proceso puede emitir una llamada al sistema para asociar un archivo a una porción de su espacio de direcciones virtuales. En la mayor parte de las implementaciones no se traen páginas al momento de la asociación, sino que a medida que se usan las páginas, se paginan bajo demanda una a la vez, usando el archivo de disco como el almacén de respaldo. Cuando el proceso termina o desasocia en forma explícita el archivo, todas las páginas modificadas se escriben de vuelta en el archivo.

Los archivos asociados proporcionan un modelo alternativo para la E/S. En vez de realizar lecturas y escrituras, el archivo se puede accesar como un gran arreglo de caracteres en la memoria. En algunas situaciones, los programadores encuentran que este modelo es más conveniente.

Si dos o más procesos se asocian al mismo archivo y al mismo tiempo, se pueden comunicar a través de la memoria compartida. Las escrituras realizadas por un proceso en la memoria compartida son inmediatamente visibles cuando el otro lee de la parte de su espacio de direcciones virtuales asociado al archivo. Por lo tanto, este mecanismo proporciona un canal con un gran ancho de banda entre los procesos, y a menudo se utiliza como tal (incluso al grado de asociar un archivo temporal). Ahora debe estar claro que si hay disponibles archivos asociados a memoria, las bibliotecas compartidas pueden usar este mecanismo.

3.5.8 Política de limpieza

La paginación funciona mejor cuando hay muchos marcos de página libres que se pueden reclamar al momento en que ocurran fallos de página. Si cada marco de página está lleno y además modificado, antes de que se pueda traer una nueva página se debe escribir una página anterior en el disco. Para asegurar una provisión abundante de marcos de página libres, muchos sistemas de paginación tienen un proceso en segundo plano conocido como **demonio de paginación**, que está inactivo la mayor parte del tiempo pero se despierta en forma periódica para inspeccionar el estado de la memoria. Si hay muy pocos marcos de página libres, el demonio de paginación empieza a seleccionar páginas para desalojarlas mediante cierto algoritmo de reemplazo de páginas. Si estas páginas han sido modificadas después de haberse cargado, se escriben en el disco.

En cualquier caso se recuerda el contenido anterior de la página. En caso de que una de las páginas desalojadas se necesite otra vez antes de que se sobrescriba su marco, puede reclamarse si se elimina de la reserva de marcos de página libres. Al mantener una provisión de marcos de página a la mano se obtiene un mejor rendimiento que al utilizar toda la memoria y después tratar de encontrar un marco al momento de necesitarlo. Cuando menos el demonio de paginación asegura que todos los marcos libres estén limpios, por lo que no se necesitan escribir en el disco en un apuro a la hora de ser requeridos.

Una manera de implementar esta política de limpieza es mediante un reloj con dos manecillas. La manecilla principal es controlada por el demonio de paginación. Cuando apunta a una página sucia, esa página se escribe de vuelta al disco y la manecilla principal se avanza. Cuando apunta a una página limpia, sólo se avanza. La manecilla secundaria se utiliza para reemplazar páginas, como en el algoritmo de reloj estándar. Sólo que ahora, la probabilidad de que la manecilla secundaria llegue a una página limpia se incrementa debido al trabajo del demonio de paginación.

3.5.9 Interfaz de memoria virtual

Hasta ahora, en todo nuestro análisis hemos supuesto que la memoria virtual es transparente para los procesos y los programadores; es decir, todo lo que ven es un gran espacio de direcciones virtuales en una computadora con una memoria física (más) pequeña. Con muchos sistemas esto es cierto pero, en algunos sistemas avanzados, los programadores tienen cierto control sobre el mapa de memoria y pueden utilizarlo de maneras no tradicionales para mejorar el comportamiento de un programa. En esta sección analizaremos unas cuantas de estas formas.

Una razón por la que se otorga a los programadores el control sobre su mapa de memoria es para permitir que dos o más procesos compartan la misma memoria. Si los programadores pueden nombrar regiones de su memoria, tal vez sea posible para un proceso dar a otro proceso el nombre de una región de memoria, de manera que el proceso también pueda asociarla. Con dos (o más) procesos compartiendo las mismas páginas, la compartición con mucho ancho de banda se hace posible: un proceso escribe en la memoria compartida y otro proceso lee de ella.

La compartición de páginas también se puede utilizar para implementar un sistema de transmisión de mensajes de alto rendimiento. Por lo general, cuando se pasan mensajes los datos se copian de un espacio de direcciones a otro, a un costo considerable. Si los procesos pueden controlar su mapa de páginas, se puede pasar un mensaje al hacer que el proceso emisor desasocie la(s) pági-

na(s) que contiene(n) el mensaje, y el proceso receptor la(s) asocia. Aquí sólo se tienen que copiar los nombres de las páginas, en vez de todos los datos.

Otra técnica más de administración avanzada de memoria es la **memoria compartida distribuida** (Feeley y colaboradores, 1995; Li, 1986; Li y Hudak, 1989; y Zekauskas y colaboradores, 1994). La idea aquí es permitir que varios procesos compartan a través de la red un conjunto de páginas, posiblemente (pero no es necesario) como un solo espacio de direcciones lineal compartido. Cuando un proceso hace referencia a una página que no está asociada, obtiene un fallo de página. El manejador de fallos de página, que puede estar en espacio de kernel o de usuario, localiza entonces la máquina que contiene la página y le envía un mensaje pidiéndole que la desasocie y la envíe a través de la red. Cuando llega la página, se asocia y la instrucción que falló se reinicia. En el capítulo 8 examinaremos la memoria compartida distribuida con más detalle.

3.6 CUESTIONES DE IMPLEMENTACIÓN

Los implementadores de los sistemas de memoria virtual tienen que elegir entre los principales algoritmos teóricos: entre el algoritmo de segunda oportunidad y el de envejecimiento, entre la asignación de páginas local o global, y entre la paginación bajo demanda o la prepaginación. Pero también tienen que estar al tanto de varias cuestiones prácticas de implementación. En esta sección daremos un vistazo a unos cuantos de los problemas comunes y ciertas soluciones.

3.6.1 Participación del sistema operativo en la paginación

Hay cuatro ocasiones en las que el sistema operativo tiene que realizar trabajo relacionado con la paginación: al crear un proceso, al ejecutar un proceso, al ocurrir un fallo de página y al terminar un proceso. Ahora examinaremos brevemente cada una de estas ocasiones para ver qué se tiene que hacer.

Cuando se crea un proceso en un sistema de paginación, el sistema operativo tiene que determinar qué tan grandes serán el programa y los datos (al principio), y crear una tabla de páginas para ellos. Se debe asignar espacio en memoria para la tabla de páginas y se tiene que inicializar. La tabla de páginas no necesita estar residente cuando el proceso se intercambia hacia fuera, pero tiene que estar en memoria cuando el proceso se está ejecutando. Además, se debe asignar espacio en el área de intercambio en el disco, para que cuando se intercambie una página, tenga un lugar a donde ir. El área de intercambio también se tiene que inicializar con el texto del programa y los datos, para que cuando el nuevo proceso empiece a recibir fallos de página, las páginas se puedan traer. Algunos sistemas paginan el texto del programa directamente del archivo ejecutable, con lo cual se ahorra espacio en disco y tiempo de inicialización. Por último, la información acerca de la tabla de páginas y el área de intercambio en el disco se debe registrar en la tabla de procesos.

Cuando un proceso se planifica para ejecución, la MMU se tiene que restablecer para el nuevo proceso y el TLB se vacía para deshacerse de los restos del proceso que se estaba ejecutando antes. La tabla de páginas del nuevo proceso se tiene que actualizar, por lo general copiándola o mediante un apuntador a éste hacia cierto(s) registro(s) de hardware. De manera opcional, algunas o

todas las páginas del proceso se pueden traer a memoria para reducir el número de fallos de página al principio (por ejemplo, es evidente que será necesaria la página a la que apunta la PC).

Cuando ocurre un fallo de página, el sistema operativo tiene que leer los registros de hardware para determinar cuál dirección virtual produjo el fallo. Con base en esta información debe calcular qué página se necesita y localizarla en el disco. Después debe buscar un marco de página disponible para colocar la nueva página, desalojando alguna página anterior si es necesario. Luego debe leer la página necesaria y colocarla en el marco de páginas. Por último, debe respaldar el contador de programa para hacer que apunte a la instrucción que falló y dejar que la instrucción se ejecute de nuevo.

Cuando un proceso termina, el sistema operativo debe liberar su tabla de páginas, sus páginas y el espacio en disco que ocupan las páginas cuando están en disco. Si alguna de las páginas están compartidas con otros procesos, las páginas en memoria y en disco sólo pueden liberarse cuando el último proceso que las utilice haya terminado.

3.6.2 Manejo de fallos de página

Finalmente, estamos en una posición para describir con detalle lo que ocurre en un fallo de página. La secuencia de eventos es la siguiente:

1. El hardware hace un trap al kernel, guardando el contador de programa en la pila. En la mayor parte de las máquinas, se guarda cierta información acerca del estado de la instrucción actual en registros especiales de la CPU.

2. Se inicia una rutina en código ensamblador para guardar los registros generales y demás información volátil, para evitar que el sistema operativo la destruya. Esta rutina llama al sistema operativo como un procedimiento.

3. El sistema operativo descubre que ha ocurrido un fallo de página y trata de descubrir cuál página virtual se necesita. A menudo, uno de los registros de hardware contiene esta información. De no ser así, el sistema operativo debe obtener el contador de programa, obtener la instrucción y analizarla en software para averiguar lo que estaba haciendo cuando ocurrió el fallo.

4. Una vez que se conoce la dirección virtual que produjo el fallo, el sistema comprueba si esta dirección es válida y si la protección es consistente con el acceso. De no ser así, el proceso recibe una señal o es eliminado. Si la dirección es válida y no ha ocurrido un fallo de página, el sistema comprueba si hay un marco de página disponible. Si no hay marcos disponibles, se ejecuta el algoritmo de reemplazo de páginas para seleccionar una víctima.

5. Si el marco de página seleccionado está sucio, la página se planifica para transferirla al disco y se realiza una conmutación de contexto, suspendiendo el proceso fallido y dejando que se ejecute otro hasta que se haya completado la transferencia al disco. En cualquier caso, el marco se marca como ocupado para evitar que se utilice para otro propósito.

6. Tan pronto como el marco de página esté limpio (ya sea de inmediato, o después de escribirlo en el disco), el sistema operativo busca la dirección de disco en donde se encuentra la página necesaria, y planifica una operación de disco para llevarla a memoria. Mientras se está cargando la página, el proceso fallido sigue suspendido y se ejecuta otro proceso de usuario, si hay uno disponible.

7. Cuando la interrupción de disco indica que la página ha llegado, las tablas de páginas se actualizan para reflejar su posición y el marco se marca como en estado normal.

8. La instrucción fallida se respalda al estado en que tenía cuando empezó, y el contador de programa se restablece para apuntar a esa instrucción.

9. El proceso fallido se planifica y el sistema operativo regresa a la rutina (en lenguaje ensamblador) que lo llamó.

10. Esta rutina recarga los registros y demás información de estado, regresando al espacio de usuario para continuar la ejecución, como si no hubiera ocurrido el fallo.

3.6.3 Respaldo de instrucción

Cuando un programa hace referencia a una página que no está en memoria, la instrucción que produjo el fallo se detiene parcialmente y ocurre un trap al sistema operativo. Una vez que el sistema operativo obtiene la página necesaria, debe reiniciar la instrucción que produjo el trap. Es más fácil decir esto que hacerlo.

Para ver la naturaleza del problema en el peor de los casos, considere una CPU que tiene instrucciones con dos direcciones, como el procesador Motorola 680x0, utilizado ampliamente en sistemas integrados. Por ejemplo, la instrucción

```
MOV.L #6(A1),2(A0)
```

es de 6 bytes (vea la figura 3-28). Para poder reiniciar la instrucción, el sistema operativo debe determinar en dónde se encuentra el primer byte de la instrucción. El valor del contador de programa al momento en que ocurre el trap depende de cuál fue el operando que falló y cómo se ha implementado el microcódigo de la CPU.

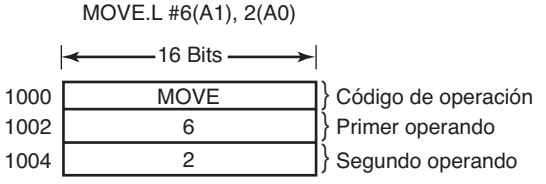

Figura 3-28. Una instrucción que produce un fallo de página.

En la figura 3-28, tenemos una instrucción que empieza en la dirección 1000 y que hace tres referencias a memoria: la palabra de la instrucción en sí y dos desplazamientos para los operandos.

Dependiendo de cuál de estas tres referencias a memoria haya ocasionado el fallo de página, el contador de programa podría ser 1000, 1002 o 1004 al momento del fallo. Con frecuencia es imposible que el sistema operativo determine sin ambigüedad en dónde empezó la instrucción. Si el contador de programa es 1002 al momento del fallo, el sistema operativo no tiene manera de saber si la palabra en 1002 es una dirección de memoria asociada con una instrucción en 1000 (por ejemplo, la ubicación de un operando) o el código de operación de una instrucción.

Tan mal como podría estar este problema, podría ser aún peor. Algunos modos de direccionamiento del 680x0 utilizan el autoincremento, lo cual significa que un efecto secundario de ejecutar la instrucción es incrementar uno o más registros. Las instrucciones que utilizan el modo de autoincremento también pueden fallar. Dependiendo de los detalles del microcódigo, el incremento se puede realizar antes de la referencia a memoria, en cuyo caso el sistema operativo debe decrementar el registro en el software antes de reiniciar la instrucción. O el autoincremento se puede realizar después de la referencia a memoria, en cuyo caso no se habrá realizado al momento del trap y el sistema operativo no deberá deshacerlo. También existe el modo de autodecremento y produce un problema similar. Los detalles precisos de si se han o no realizado los autoincrementos o autodecrementos antes de las correspondientes referencias a memoria pueden diferir de una instrucción a otra, y de un modelo de CPU a otro.

Por fortuna, en algunas máquinas los diseñadores de la CPU proporcionan una solución, por lo general en la forma de un registro interno oculto, en el que se copia el contador de programa justo antes de ejecutar cada instrucción. Estas máquinas también pueden tener un segundo registro que indique cuáles registros se han ya autoincrementado o autodecrementado y por cuánto. Dada esta información, el sistema operativo puede deshacer sin ambigüedad todos los efectos de la instrucción fallida, de manera que se pueda reiniciar. Si esta información no está disponible, el sistema operativo tiene que hacer peripecias para averiguar qué ocurrió y cómo puede repararlo. Es como si los diseñadores del hardware no pudieran resolver el problema y pasaran esa responsabilidad a los escritores del sistema operativo.

3.6.4 Bloqueo de páginas en memoria

Aunque no hemos hablado mucho sobre la E/S en este capítulo, el hecho de que una computadora tenga memoria virtual no significa que estén ausentes las operaciones de E/S. La memoria virtual y la E/S interactúan en formas sutiles. Considere un proceso que acaba de emitir una llamada al sistema para leer algún archivo o dispositivo y colocarlo en un búfer dentro de su espacio de direcciones. Mientras espera a que se complete la E/S, el proceso se suspende y se permite a otro proceso ejecutarse. Este otro proceso recibe un fallo de página.

Si el algoritmo de paginación es global, hay una pequeña probabilidad (distinta de cero) de que la página que contiene el búfer de E/S sea seleccionada para eliminarla de la memoria. Si un dispositivo de E/S se encuentra en el proceso de realizar una transferencia por DMA a esa página, al eliminarla parte de los datos se escribirán en el búfer al que pertenecen y parte sobre la página que se acaba de cargar. Una solución a este problema es bloquear las páginas involucradas en operaciones de E/S en memoria, de manera que no se eliminen. Bloquear una página se conoce a menudo

como **fijada** (*pinning*) en la memoria. Otra solución es enviar todas las operaciones de E/S a búferes del kernel y después copiar los datos a las páginas de usuario.

3.6.5 Almacén de respaldo

En nuestro análisis de los algoritmos de reemplazo de páginas, vimos cómo se selecciona una página para eliminarla. No hemos dicho mucho con respecto a dónde se coloca en el disco cuando se pagina hacia fuera de la memoria. Ahora vamos a describir algunas cuestiones relacionadas con la administración del disco.

El algoritmo más simple para asignar espacio de página en el disco es tener una partición de intercambio especial en el disco o aún mejor es tenerla en un disco separado del sistema operativo (para balancear la carga de E/S). La mayor parte de los sistemas UNIX funcionan así. Esta partición no tiene un sistema de archivos normal, lo cual elimina la sobrecarga de convertir desplazamientos en archivos a direcciones de bloque. En vez de ello, se utilizan números de bloque relativos al inicio de la partición.

Cuando se inicia el sistema, esta partición de intercambio está vacía y se representa en memoria como una sola entrada que proporciona su origen y tamaño. En el esquema más simple, cuando se inicia el primer proceso, se reserva un trozo del área de la partición del tamaño del primer proceso y se reduce el área restante por esa cantidad. A medida que se inician nuevos procesos, se les asigna trozos de la partición de intercambio con un tamaño equivalente al de sus imágenes de núcleo. Al terminar, se libera su espacio en disco. La partición de intercambio se administra como una lista de trozos libres. En el capítulo 10 analizaremos mejores algoritmos.

Con cada proceso está asociada la dirección de disco de su área de intercambio; es decir, en qué parte de la partición de intercambio se mantiene su imagen. Esta información se mantiene en la tabla de procesos. El cálculo la dirección en la que se va a escribir una página es simple: sólo se suma el desplazamiento de la página dentro del espacio de direcciones virtual al inicio del área de intercambio. Sin embargo, antes de que un proceso pueda empezar se debe inicializar el área de intercambio. Una forma de hacerlo es copiar toda la imagen del proceso al área de intercambio, de manera que se pueda traer y colocar *en* la memoria según sea necesario. La otra es cargar todo el proceso en memoria y dejar que se pagine *hacia fuera* según sea necesario.

Sin embargo, este simple modelo tiene un problema: los procesos pueden incrementar su tamaño antes de empezar. Aunque el texto del programa por lo general es fijo, el área de los datos puede crecer algunas veces, y la pila siempre puede crecer. En consecuencia, podría ser mejor reservar áreas de intercambio separadas para el texto, los datos y la pila, permitiendo que cada una de estas áreas consista de más de un trozo en el disco.

El otro extremo es no asignar nada por adelantado y asignar espacio en el disco para cada página cuando ésta se intercambie hacia fuera de la memoria y desasignarlo cuando se vuelva a intercambiar hacia la memoria. De esta forma, los procesos en memoria no acaparan espacio de intercambio. La desventaja es que se necesita una dirección de disco en la memoria para llevar la cuenta de cada página en el disco. En otras palabras, debe haber una tabla por cada proceso que indique en dónde se encuentra cada página en el disco. Las dos alternativas se muestran en la figura 3-29.

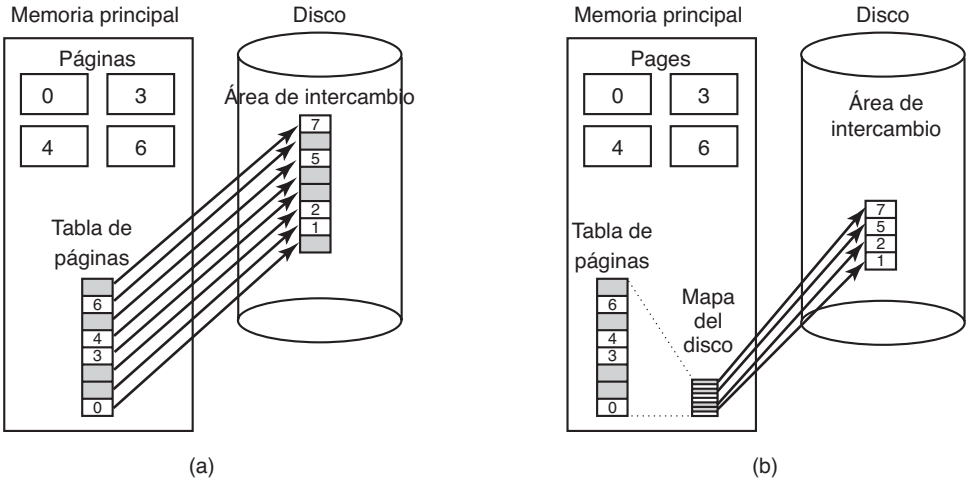

Figura 3-29. (a) Paginación a un área de intercambio estática. (b) Respaldo de páginas en forma dinámica.

En la figura 3-29(a) se ilustra una tabla de páginas con ocho páginas. Las páginas 0, 3, 4 y 6 están en memoria. Las páginas 1, 2, 5 y 7 están en disco. El área de intercambio en el disco es tan grande como el espacio de direcciones virtuales del proceso (ocho páginas), donde cada página tiene una ubicación fija a la cual se escribe cuando se desaloja de la memoria principal. Para calcular esta dirección sólo se requiere saber dónde empieza el área de paginación del proceso, ya que las páginas se almacenan en ella de manera contigua, ordenadas por su número de página virtual. Una página que está en memoria siempre tiene una copia sombra en el disco, pero esta copia puede estar obsoleta si la página se modificó después de haberla cargado. Las páginas sombreadas en la memoria indican páginas que no están presentes en memoria. Las páginas sombreadas en el disco son (en principio) suplantadas por las copias en memoria, aunque si una página de memoria se tiene que intercambiar de vuelta a disco y no se ha modificado desde que se cargó, se utilizará la copia del disco (sombreada).

En la figura 3-29(b), las páginas no tienen direcciones fijas en el disco. Cuando se intercambia una página hacia fuera de la memoria, se selecciona una página vacía en el disco al momento y el mapa de disco (que tiene espacio para una dirección de disco por página virtual) se actualiza de manera acorde. Una página en memoria no tiene copia en el disco. Sus entradas en el mapa de disco contienen una dirección de disco inválida o un bit que las marca como que no están en uso.

No siempre es posible tener una partición de intercambio fija. Por ejemplo, tal vez no haya particiones de disco disponibles. En este caso se pueden utilizar uno o más archivos previamente asignados dentro del sistema de archivos normal. Windows utiliza este método. Sin embargo, aquí se puede utilizar una optimización para reducir la cantidad de espacio en disco necesaria. Como el texto del programa de cada proceso proviene de algún archivo (ejecutable) en el sistema de archivos, este archivo ejecutable se puede utilizar como el área de intercambio. Mejor aún, ya que el texto del programa generalmente es de sólo lectura, cuando la memoria es escasa y se tienen que desalojar páginas del programa de la memoria, sólo se descartan y se vuelven a leer del archivo ejecutable cuando se necesiten. Las bibliotecas compartidas también pueden trabajar de esta forma.

3.6.6 Separación de política y mecanismo

Una importante herramienta para administrar la complejidad de cualquier sistema es separar la política del mecanismo. Este principio se puede aplicar a la administración de la memoria, al hacer que la mayor parte del administrador de memoria se ejecute como un proceso a nivel usuario. Dicha separación se realizó por primera vez en Mach (Young y colaboradores, 1987). El siguiente análisis se basa de manera general en Mach.

En la figura 3-30 se muestra un ejemplo simple de cómo se pueden separar la política y el mecanismo. Aquí, el sistema de administración de memoria se divide en dos partes:

1. Un manejador de la MMU de bajo nivel.
2. Un manejador de fallos de página que forma parte del kernel.
3. Un paginador externo que se ejecuta en espacio de usuario.

Todos los detalles acerca del funcionamiento de la MMU están encapsulados en el manejador de la MMU, que es código dependiente de la máquina y tiene que volver a escribirse para cada nueva plataforma a la que se porte el sistema operativo. El manejador de fallos de página es código independiente de la máquina y contiene la mayor parte del mecanismo para la paginación. La política se determina en gran parte mediante el paginador externo, que se ejecuta como un proceso de usuario.

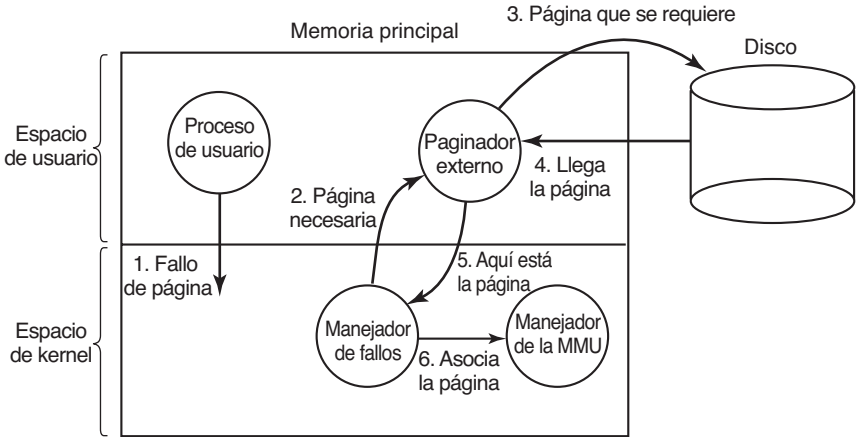

Figura 3-30. Manejo de fallos de página con un paginador externo.

Cuando se inicia un proceso, se notifica al paginador externo para poder establecer el mapa de páginas del proceso y asignar el almacenamiento de respaldo en el disco, si es necesario. A medida que el proceso se ejecuta, puede asignar nuevos objetos en su espacio de direcciones, por lo que se notifica de nuevo al paginador externo.

Una vez que el proceso empieza a ejecutarse, puede obtener un fallo de página. El manejador de fallos averigua cuál página virtual se necesita y envía un mensaje al paginador externo, indicán-

dole el problema. Después el paginador externo lee la página necesaria del disco y la copia a una porción de su propio espacio de direcciones. Después le indica al manejador de fallos en dónde está la página. Luego, el manejador de fallos desasigna la página del espacio de direcciones del paginador externo y pide al manejador de la MMU que la coloque en el espacio de direcciones del usuario, en el lugar correcto. Entonces se puede reiniciar el proceso de usuario.

Esta implementación no deja establecido dónde se va a colocar el algoritmo de reemplazo de páginas. Sería más limpio tenerlo en el paginador externo, pero hay ciertos problemas con este método. El problema principal es que el paginador externo no tiene acceso a los bits *R* y *M* de todas las páginas. Estos bits desempeñan un papel en muchos de los algoritmos de paginación. Por ende, se necesita algún mecanismo para pasar esta información al paginador externo o el algoritmo de reemplazo de páginas debe ir en el kernel. En el último caso, el manejador de fallos indica al paginador externo cuál página ha seleccionado para desalojarla y proporciona los datos, ya sea asignándola al espacio de direcciones eterno del paginador o incluyéndola en un mensaje. De cualquier forma, el paginador externo escribe los datos en el disco.

La principal ventaja de esta implementación es que se obtiene un código más modular y una mayor flexibilidad. La principal desventaja es la sobrecarga adicional de cruzar el límite entre usuario y kernel varias veces, y la sobrecarga de los diversos mensajes que se envían entre las partes del sistema. En estos momentos el tema es muy controversial, pero a medida que las computadoras se hacen cada vez más rápidas, y el software se hace cada vez más complejo, a la larga sacrificar cierto rendimiento por un software más confiable probablemente sea algo aceptable para la mayoría de los implementadores.

3.7 SEGMENTACIÓN

La memoria virtual que hemos analizado hasta ahora es unidimensional, debido a que las direcciones virtuales van desde 0 hasta cierta dirección máxima, una dirección después de la otra. Para muchos problemas, tener dos o más espacios de direcciones virtuales separados puede ser mucho mejor que tener sólo uno. Por ejemplo, un compilador tiene muchas tablas que se generan a medida que procede la compilación, las cuales posiblemente incluyen:

1. El texto del código fuente que se guarda para el listado impreso (en sistemas de procesamiento por lotes).
2. La tabla de símbolos, que contiene los nombres y atributos de las variables.
3. La tabla que contiene todas las constantes enteras y de punto flotante utilizadas.
4. El árbol de análisis sintáctico, que contiene el análisis sintáctico del programa.
5. La pila utilizada para las llamadas a procedimientos dentro del compilador.

Cada una de las primeras cuatro tablas crece en forma continua a medida que procede la compilación. La última crece y se reduce de maneras impredecibles durante la compilación. En una memoria unidimensional, a estas cinco tablas se les tendría que asignar trozos contiguos de espacio de direcciones virtuales, como en la figura 3-31.

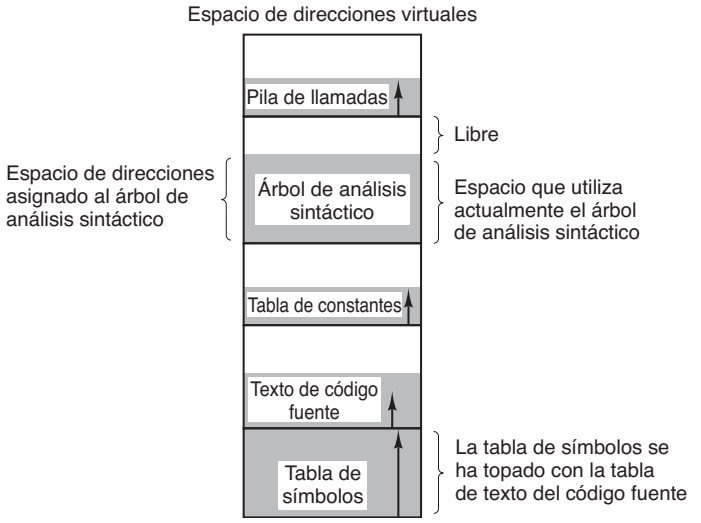

Figura 3-31. En un espacio de direcciones unidimensional con tablas que aumentan de tamaño, una tabla puede toparse con otra.

Considere lo que ocurre si un programa tiene un número variables mucho mayor de lo usual, pero una cantidad normal de todo lo demás. El trozo de espacio de direcciones asignado para la tabla de símbolos se puede llenar, pero puede haber mucho espacio en las otras tablas. Desde luego que el compilador podría simplemente emitir un mensaje indicando que la compilación no puede continuar debido a que hay demasiadas variables, pero esto no parece muy deportivo cuando se deja espacio sin usar en las otras tablas.

Otra posibilidad es jugar a Robin Hood, tomando espacio de las tablas con exceso y dándolo a las tablas con poco espacio. Esta revoltura puede hacerse, pero es similar a cuando uno administra sus propios sobrepuestos; una molestia como mínimo y, en el peor de los casos, mucho trabajo tedioso sin recompensas.

Lo que se necesita realmente es una forma de liberar al programador de tener que administrar las tablas en expansión y contracción, de la misma forma que la memoria virtual elimina la preocupación de tener que organizar el programa en sobrepuestos.

Una solución simple y en extremado general es proporcionar la máquina con muchos espacios de direcciones por completo independientes, llamados **segmentos**. Cada segmento consiste en una secuencia lineal de direcciones, desde 0 hasta cierto valor máximo. La longitud de cada segmento puede ser cualquier valor desde 0 hasta el máximo permitido. Los distintos segmentos pueden tener distintas longitudes (y por lo general así es). Además las longitudes de los segmentos pueden cambiar durante la ejecución. La longitud de un segmento de pila puede incrementarse cada vez que se meta algo a la pila y decrementarse cada vez que se saque algo.

Debido a que cada segmento constituye un espacio de direcciones separado, los distintos segmentos pueden crecer o reducirse de manera independiente, sin afectar unos a otros. Si una pila en

cierto segmento necesita más espacio de direcciones para crecer, puede tenerlo, ya que no hay nada más en su espacio de direcciones con lo que se pueda topar. Desde luego que un segmento se puede llenar, pero por lo general los segmentos son muy grandes, por lo que esta ocurrencia es rara. Para especificar una dirección en esta memoria segmentada o bidimensional, el programa debe suministrar una dirección en dos partes, un número de segmento y una dirección dentro del segmento. La figura 3-32 ilustra el uso de una memoria segmentada para las tablas del compilador que vimos antes. Aquí se muestran cinco segmentos independientes.

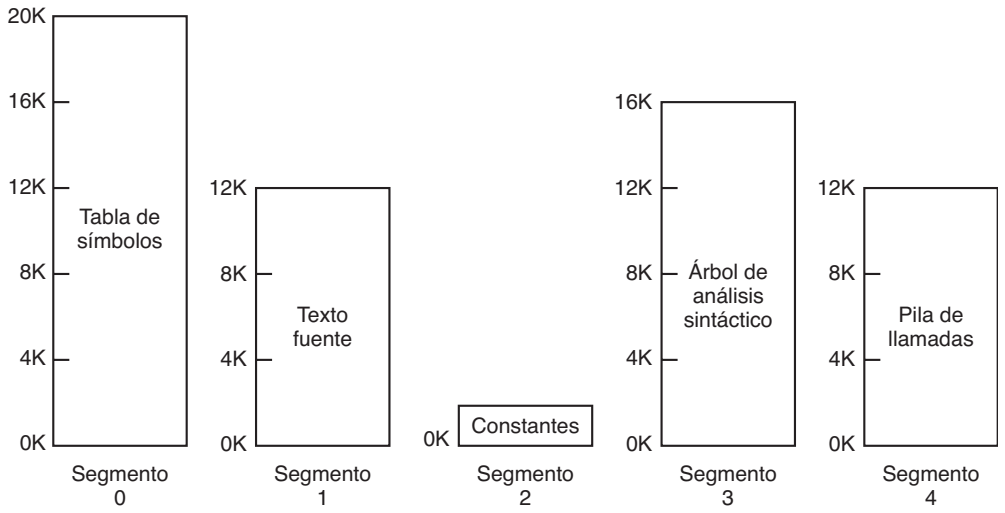

Figura 3-32. Una memoria segmentada permite que cada tabla crezca o se reduzca de manera independiente a las otras tablas.

Enfatizamos que un segmento es una entidad lógica, de la cual el programador está consciente y la utiliza como entidad lógica. Un segmento podría contener un procedimiento, o un arreglo, o una pila, o una colección de variables escalares, pero por lo general no contiene una mezcla de distintos tipos.

Una memoria segmentada tiene otras ventajas además de simplificar el manejo de estructuras de datos que aumentan o reducen su tamaño. Si cada procedimiento ocupa un segmento separado, con la dirección 0 como su dirección inicial, la vinculación de procedimientos que se compilan por separado se simplifica de manera considerable. Después de que se han compilado y vinculado todos los procedimientos que constituyen un programa, una llamada al procedimiento en el segmento *n* utilizará la dirección en dos partes (*n*, 0) para direccionar la palabra 0 (el punto de entrada).

Si el procedimiento en el segmento *n* se modifica y recompila posteriormente, no hay necesidad de cambiar los demás procedimientos (ya que no se han modificado direcciones iniciales), aun si la nueva versión es más grande que la anterior. Con una memoria unidimensional, los procedimientos se empaquetan estrechamente, uno al lado del otro, sin espacio de direcciones entre ellos. En consecuencia, al cambiar el tamaño de un procedimiento se puede afectar la dirección inicial de otros procedimientos (no relacionados). Esto a su vez requiere la modificación de todos los proce-

dimientos que llamen a cualquiera de los procedimientos que se movieron, para poder incorporar sus nuevas direcciones iniciales. Si un programa contiene cientos de procedimientos, este proceso puede ser costoso.

La segmentación también facilita la compartición de procedimientos o datos entre varios procesos. Un ejemplo común es la biblioteca compartida. Las estaciones de trabajo modernas que operan sistemas de ventanas avanzados tienen a menudo bibliotecas gráficas en extremo extensas que se compilan en casi todos los programas. En un sistema segmentado, la biblioteca gráfica se puede colocar en un segmento y varios procesos pueden compartirla, eliminando la necesidad de tenerla en el espacio de direcciones de cada proceso. Aunque también es posible tener bibliotecas compartidas en sistemas de paginación puros, es más complicado. En efecto, estos sistemas lo hacen mediante la simulación de la segmentación.

Como cada segmento forma una entidad lógica de la que el programador está consciente, como un procedimiento, un arreglo o una pila, los distintos segmentos pueden tener diferentes tipos de protección. Un segmento de procedimiento se puede especificar como de sólo ejecución, para prohibir los intentos de leer de él o almacenar en él. Un arreglo de punto flotante se puede especificar como de lectura/escritura pero no como de ejecución, y los intentos de saltar a él se atraparán. Dicha protección es útil para atrapar errores de programación.

El lector debe tratar de comprender por qué la protección es sensible en una memoria segmentada, pero no en una memoria paginada unidimensionalmente. En una memoria segmentada, el usuario está consciente de lo que hay en cada segmento. Por lo general, un segmento no contendría un procedimiento y una pila, por ejemplo, sino uno o el otro, no ambos. Como cada segmento contiene sólo un tipo de objeto, puede tener la protección apropiada para ese tipo específico. La paginación y la segmentación se comparan en la figura 3-33.

El contenido de una página es, en cierto sentido, accidental. El programador ni siquiera está consciente del hecho de que está ocurriendo la paginación. Aunque sería posible poner unos cuantos bits en cada entrada de la tabla de páginas para especificar el acceso permitido, para utilizar esta característica el programador tendría que llevar registro del lugar en el que se encontraran los límites de página en su espacio de direcciones. La paginación se inventó para eliminar precisamente ese tipo de administración. Como el usuario de una memoria segmentada tiene la ilusión de que todos los segmentos se encuentran en memoria principal todo el tiempo (es decir, que puede direccionarlos como si estuvieran) puede proteger cada segmento por separado sin tener que preocuparse con la administración por tener que superponerlos.

3.7.1 Implementación de segmentación pura

La implementación de segmentación difiere de la paginación de una manera esencial: las páginas tienen un tamaño fijo y los segmentos no. La figura 3-34(a) muestra un ejemplo de una memoria física que al principio contiene cinco segmentos. Ahora considere lo que ocurre si el segmento 1 se desaloja y el segmento 7, que es más pequeño, se coloca en su lugar. Obtendremos la configuración de memoria de la figura 3-34(b). Entre el segmento 7 y el 2 hay un área sin uso; es decir, un hueco. Después el segmento 4 se reemplaza por el segmento 5, como en la figura 3-34(c), y el segmento 3 se reemplaza por el segmento 6, como en la figura 3-34(d).

Consideración	Paginación	Segmentación
¿Necesita el programador estar consciente de que se está utilizando esta técnica?	No	Sí
¿Cuántos espacios de direcciones lineales hay?	1	Muchos
¿Puede el espacio de direcciones total exceder al tamaño de la memoria física?	Sí	Sí
¿Pueden los procedimientos y los datos diferenciarse y protegerse por separado?	No	Sí
¿Pueden las tablas cuyo tamaño fluctúa acomodarse con facilidad?	No	Sí
¿Se facilita la compartición de procedimientos entre usuarios?	No	Sí
¿Por qué se inventó esta técnica?	Para obtener un gran espacio de direcciones lineal sin tener que comprar más memoria física	Para permitir a los programas y datos dividirse en espacios de direcciones lógicamente independientes, ayudando a la compartición y la protección

Figura 3-33. Comparación de la paginación y la segmentación.

Una vez que el sistema haya estado en ejecución por un tiempo, la memoria se dividirá en un número de trozos, de los cuales algunos contendrán segmentos y otros huecos. Este fenómeno, llamado **efecto de tablero de ajedrez** o **fragmentación externa**, desperdicia memoria en los huecos. Se puede manejar mediante la compactación, como veremos en la figura 3-34(c).

3.7.2 Segmentación con paginación: MULTICS

Si los segmentos son extensos, puede ser inconveniente (o incluso imposible) mantenerlos completos en memoria principal. Esto nos lleva a la idea de paginarlos, de manera que sólo las páginas que realmente se necesiten tengan que estar presentes. Varios sistemas importantes han soportado segmentos de páginas. En esta sección describiremos el primero: MULTICS. En la siguiente analizaremos uno más reciente: el Intel Pentium.

MULTICS operaba en las máquinas Honeywell 6000 y sus descendientes; proveía a cada programa con una memoria virtual de hasta 2^{18} segmentos (más de 250,000), cada uno de los cuales podría ser de hasta 65,536 palabras (36 bits) de longitud. Para implementar esto, los diseñadores de MULTICS optaron por considerar a cada segmento como una memoria virtual y la paginaron, com-

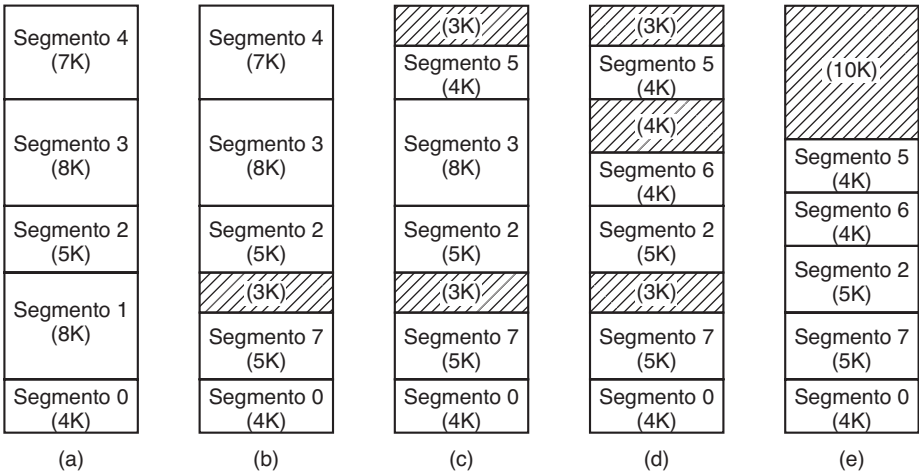

Figura 3-34. (a)-(d) Desarrollo del efecto de tablero de ajedrez. (e) Eliminación del efecto de tablero de ajedrez mediante la compactación.

binando las ventajas de la paginación (tamaño de página uniforme y no tener que mantener todo el segmento en la memoria, si sólo se está utilizando parte de él) con las ventajas de la segmentación (facilidad de programación, modularidad, protección, compartición).

Cada programa de MULTICS tiene una tabla de segmentos, con un descriptor por segmento. Como hay en potencia más de un cuarto de millón de entradas en la tabla, la tabla de segmentos es en sí un segmento y se pagina. Un descriptor de segmentos contiene una indicación acerca de si el segmento está en memoria principal o no. Si cualquier parte del segmento está en memoria, se considera que el segmento está en memoria y su tabla de páginas estará en memoria. Si el segmento está en memoria, su descriptor contiene un apuntador de 18 bits a su tabla de páginas, como en la figura 3-35(a). Como las direcciones físicas son de 24 bits y las páginas se alinean en límites de 64 bytes (implicando que los 6 bits de menor orden de las direcciones de página son 000000), sólo se necesitan 18 bits en el descriptor para almacenar una dirección de la tabla de páginas. El descriptor también contiene el tamaño del segmento, los bits de protección y unos cuantos elementos más. La figura 3-35(b) ilustra un descriptor de segmento de MULTICS. La dirección del segmento en la memoria secundaria no está en el descriptor de segmentos, sino en otra tabla utilizada por el manejador de fallos de segmento.

Cada segmento es un espacio de direcciones virtual ordinario y se pagina de la misma forma que la memoria paginada no segmentada descrita anteriormente en este capítulo. El tamaño normal de página es de 1024 palabras (aunque unos cuantos segmentos utilizados por MULTICS en sí no están paginados o están paginados en unidades de 64 palabras para ahorrar memoria física).

Una dirección en MULTICS consiste en dos partes: el segmento y la dirección dentro del segmento. La dirección dentro del segmento se divide aún más en un número de página y en una palabra dentro de la página, como se muestra en la figura 3-36. Cuando ocurre una referencia a memoria, se lleva a cabo el siguiente algoritmo.

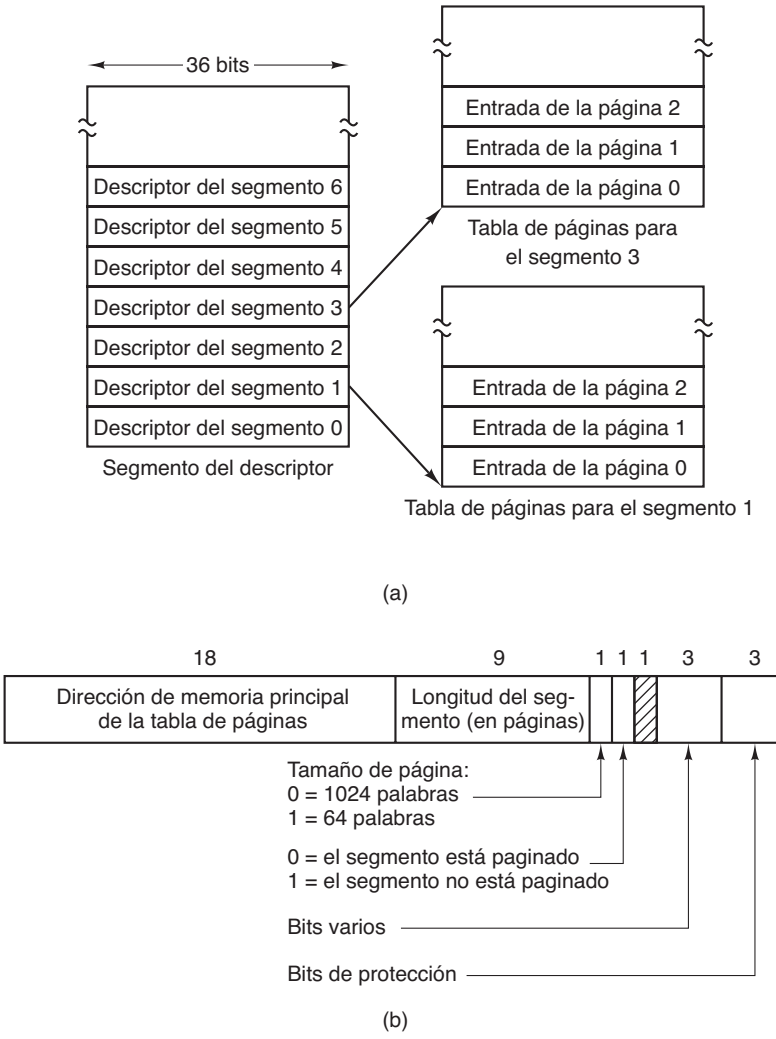

Figura 3-35. La memoria virtual de MULTICS. (a) El segmento del descriptor apunta a las tablas de páginas. (b) Un descriptor de segmento. Los números son las longitudes de los campos.

1. El número de segmento se utiliza para encontrar el descriptor de segmentos.

2. Se realiza una comprobación para ver si la tabla de páginas del segmento está en la memoria. Si la tabla de páginas está en memoria, se localiza. Si no, ocurre un fallo de segmento. Si hay una violación a la protección, ocurre un fallo (trap).

3. La entrada en la tabla de páginas para la página virtual solicitada se examina. Si la página en sí no está en memoria, se dispara un fallo de página. Si está en memoria, la direc-

ción de la memoria principal del inicio de la página se extrae de la entrada en la tabla de páginas.

4. El desplazamiento se agrega al origen de la página para obtener la dirección de memoria principal en donde se encuentra la palabra.

5. Finalmente se lleva a cabo la operación de lectura o almacenamiento.

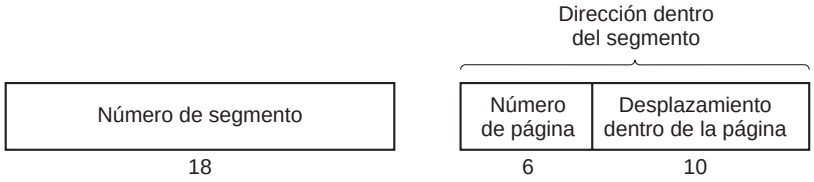

Figura 3-36. Una dirección virtual de MULTICS de 34 bits.

Este proceso se ilustra en la figura 3-37. Por simpleza, omitimos el hecho de que el mismo segmento del descriptor está paginado. Lo que ocurre en realidad es que se utiliza un registro (el registro base del descriptor) para localizar la tabla de páginas del segmento del descriptor, que a su vez apunta a las páginas del segmento del descriptor. Una vez que se ha encontrado el descriptor para el segmento necesario, el direccionamiento procede como se muestra en la figura 3-37.

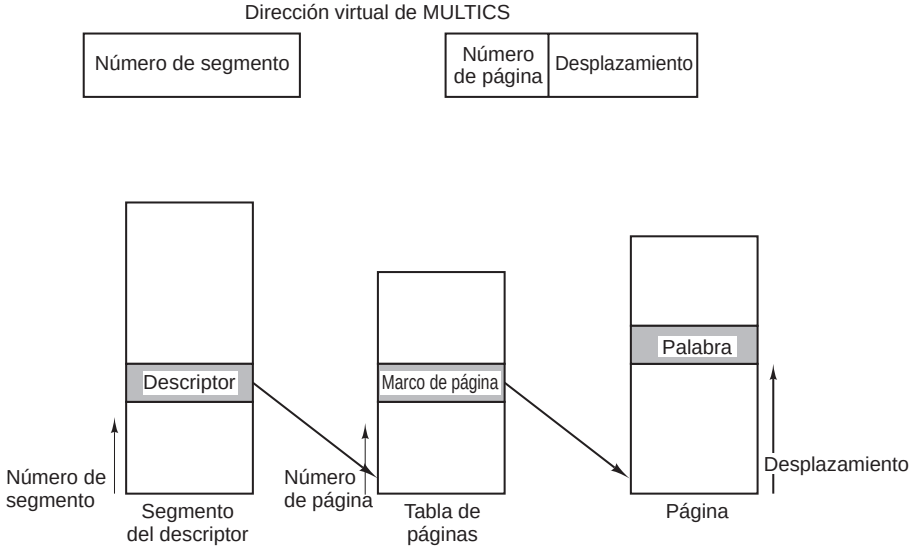

Figura 3-37. Conversión de una dirección MULTICS de dos partes en una dirección de memoria principal.

Como sin duda habrá adivinado para estos momentos, si el sistema operativo llevara a cabo el algoritmo anterior en cada instrucción, los programas no se ejecutarían con mucha rapidez. En realidad, el hardware de MULTICS contiene un TLB de 16 palabras de alta velocidad que puede buscar una llave dada en todas sus entradas en paralelo. Esto se ilustra en la figura 3-38. Cuando se

presenta una dirección a la computadora, el hardware de direccionamiento primero comprueba que la dirección virtual esté en el TLB. De ser así, obtiene el número del marco de página directamente del TLB y forma la dirección actual de la palabra referenciada sin tener que buscar en el segmento del descriptor o en la tabla de páginas.

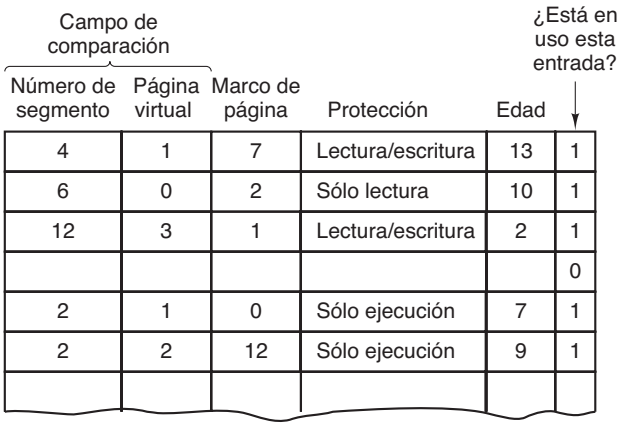

Figura 3-38. Una versión simplificada del TLB de MULTICS. La existencia de dos tamaños de página hace que el TLB real sea más complicado.

Las direcciones de las 16 páginas con referencia más reciente se mantienen en el TLB. Los programas cuyo conjunto de trabajo sea menor que el tamaño del TLB se equilibrarán con las direcciones de todo el conjunto de trabajo en el TLB, y por lo tanto se ejecutarán con eficiencia. Si la página no está en el TLB, se hace referencia al descriptor y las tablas de páginas para encontrar la dirección del marco, el TLB se actualiza para incluir esta página, y la página de uso menos reciente se descarta. El campo edad lleva el registro de cuál entrada es la de uso menos reciente. La razón de usar un TLB es para comparar los números de segmento y de página de todas las entradas en paralelo.

3.7.3 Segmentación con paginación: Intel Pentium

La memoria virtual en el Pentium se asemeja en muchas formas a MULTICS, incluyendo la presencia de segmentación y paginación. Mientras que MULTICS tiene 256K segmentos independientes, cada uno con hasta 64K palabras de 36 bits, el Pentium tiene 16K segmentos independientes, cada uno de los cuales contiene hasta un mil millones de palabras de 32 bits. Aunque hay menos segmentos, entre mayor sea el tamaño del segmento será más importante, ya que pocos programas necesitan más de 1000 segmentos, pero muchos programas necesitan segmentos extensos.

El corazón de la memoria virtual del Pentium consiste en dos tablas, llamadas **LDT** (*Local Descriptor Table*, Tabla de descriptores locales) y **GDT** (*Global Descriptor Table*, Tabla de descrip-

tores globales). Cada programa tiene su propia LDT, pero hay una sola GDT compartida por todos los programas en la computadora. La LDT describe los segmentos locales para cada programa, incluyendo su código, datos, pila, etcétera, mientras que la GDT describe los segmentos del sistema, incluyendo el sistema operativo en sí.

Para acceder a un segmento, un programa del Pentium primero carga un selector para ese segmento en uno de los seis registros de segmento de la máquina. Durante la ejecución, el registro CS contiene el selector para el segmento de código y el registro DS contiene el selector para el segmento de datos. Los demás registros de segmento son menos importantes. Cada selector es un número de 16 bits, como se muestra en la figura 3-39.

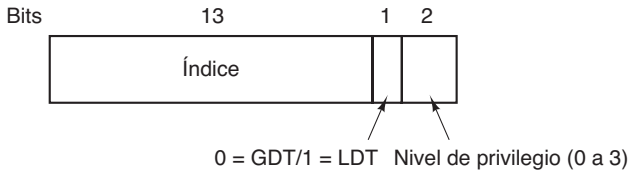

Figura 3-39. Un selector del Pentium.

Uno de los bits del selector indica si el segmento es local o global (es decir, si está en la LDT o en la GDT). Otros trece bits especifican el número de entrada en la LDT o GDT, por lo que estas tablas están restringidas a contener cada una 8K descriptores de segmento. Los otros 2 bits que se relacionan con la protección se describen más adelante. El descriptor 0 está prohibido. Puede cargarse de manera segura en un registro de segmento para indicar que el registro de segmento no está disponible en un momento dado. Produce un trap si se utiliza.

Al momento en que se carga un selector en un registro de segmento, el descriptor correspondiente se obtiene de la LDT o GDT y se almacena en registros de microprograma, por lo que se puede acceder con rapidez. Como se ilustra en la figura 3-40, un descriptor consiste de 8 bytes, incluyendo la dirección base del segmento, su tamaño y demás información.

El formato del selector se ha elegido con inteligencia para facilitar la localización del descriptor. Primero se selecciona la LDT o GDT, con base en el bit 2 del selector. Después el selector se copia a un registro temporal interno, y los 3 bits de menor orden se establecen en 0. Por último, se le suma la dirección de la tabla LDT o GDT para proporcionar un apuntador al descriptor. Por ejemplo, el selector 72 se refiere a la entrada 9 en la GDT, que se encuentra en la dirección GDT + 72.

Vamos a rastrear los pasos mediante los cuales un par (selector, desplazamiento) se convierte en una dirección física. Tan pronto como el microprograma sabe cuál registro de segmento se está utilizando, puede encontrar el descriptor completo que corresponde a ese selector en sus registros internos. Si el segmento no existe (selector 0) o en ese momento se paginó para sacarlo de memoria, se produce un trap.

Después, el hardware utiliza el campo *Límite* para comprobar si el desplazamiento está más allá del final del segmento, en cuyo caso también se produce un trap. Lógicamente debería haber un campo de 32 bits en el descriptor para proporcionar el tamaño del segmento, pero sólo hay 20 bits

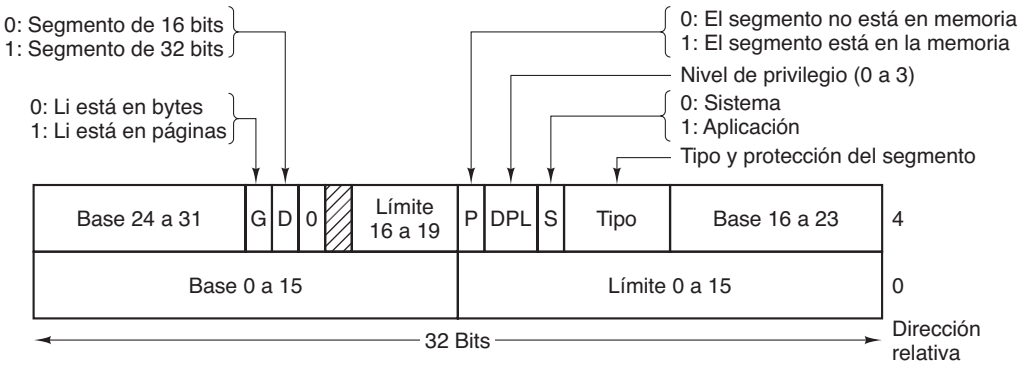

Figura 3-40. Descriptor del segmento de código del Pentium. Los segmentos de datos difieren un poco.

disponibles, por lo que se utiliza un esquema distinto. Si el campo *Gbit* (granularidad) es 0, el campo *Límite* es el tamaño de segmento exacto, hasta 1 MB. Si es 1, el campo *Límite* proporciona el tamaño del segmento en páginas, en vez de bytes. El tamaño de página del Pentium está fijo en 4 KB, por lo que 20 bits son suficientes para segmentos de hasta 2^{32} bytes.

Suponiendo que el segmento está en memoria y que el desplazamiento está en el rango, el Pentium suma el campo *Base* de 32 bits en el descriptor al desplazamiento para formar lo que se conoce como una **dirección lineal**, como se muestra en la figura 3-41. El campo *Base* se divide en tres partes y se esparce por todo el descriptor para tener compatibilidad con el 286, en donde el campo *Base* sólo tiene 24 bits. En efecto, el campo *Base* permite que cada segmento empiece en un lugar arbitrario dentro del espacio de direcciones lineal de 32 bits.

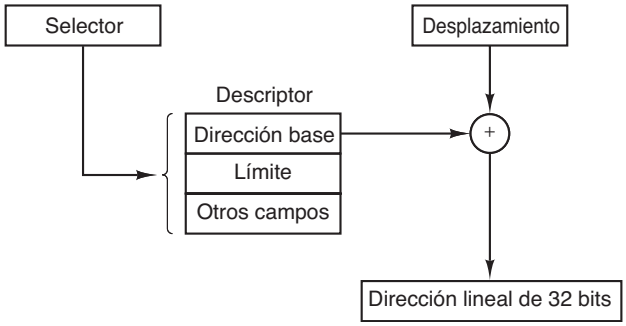

Figura 3-41. Conversión de un par (selector, desplazamiento) en una dirección lineal.

Si se deshabilita la paginación (mediante un bit en un registro de control global), la dirección lineal se interpreta como la dirección física y se envía a la memoria para la lectura o escritura. Por ende, con la paginación deshabilitada tenemos un esquema de segmentación pura, en donde la dirección base de cada segmento se proporciona en su descriptor. No se evita que los segmentos se

traslapen, probablemente debido a que sería demasiado problema y se requeriría mucho tiempo para verificar que todos estuvieran disjuntos.

Por otro lado, si la paginación está habilitada, la dirección lineal se interpreta como una dirección virtual y se asigna a la dirección física usando tablas de páginas, en forma muy parecida a los ejemplos anteriores. La única complicación verdadera es que con una dirección virtual de 32 bits y una página de 4 KB, un segmento podría contener 1 millón de páginas, por lo que se utiliza una asignación de dos niveles para reducir el tamaño de la tabla de páginas para segmentos pequeños.

Cada programa en ejecución tiene un **directorio de páginas** que consiste de 1024 entradas de 32 bits. Se encuentra en una dirección a la que apunta un registro global. Cada entrada en este directorio apunta a una tabla de páginas que también contiene 1024 entradas de 32 bits. Las entradas en la tabla de páginas apuntan a marcos de página. El esquema se muestra en la figura 3-42.

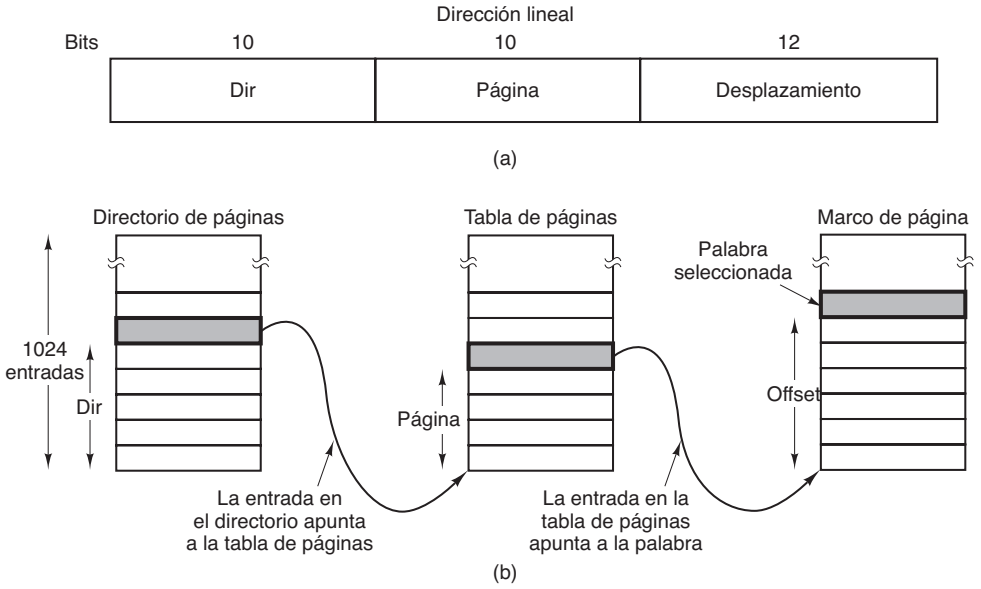

Figura 3-42. Asociación de una dirección lineal a una dirección física.

En la figura 3-42(a) podemos ver una dirección lineal dividida en tres campos: *Dir*, *Página* y *Desplazamiento*. El campo *Dir* se utiliza para indexar en el directorio de páginas y localizar un apuntador a la tabla de páginas apropiada. Después se utiliza el campo *Página* como un índice en la tabla de páginas para buscar la dirección física del marco de página. Por último, *Desplazamiento* se suma a la dirección del marco de página para obtener la dirección física del byte o palabra requerida.

Las entradas en la tabla de páginas son de 32 bits cada una, 20 de las cuales contienen un número de marco de página. Los bits restantes contienen bits de acceso y bits sucios, establecidos por el hardware para el beneficio del sistema operativo, bits de protección y otros bits utilitarios.

Cada tabla de páginas tiene entradas para 1024 marcos de página de 4 KB, por lo que una sola tabla de páginas maneja 4 megabytes de memoria. Un segmento menor de 4M tendrá un directo-

rio de páginas con una sola entrada: un apuntador a su única tabla de páginas. De esta manera, la sobrecarga por los segmentos cortos es sólo de dos páginas, en vez del millón de páginas que se necesitarían en una tabla de páginas de un nivel.

Para evitar realizar referencias repetidas a memoria, el Pentium (al igual que MULTICS) tiene un pequeño TLB que asigna directamente las combinaciones *Dir-Página* de uso más reciente a la dirección física del marco de página. Sólo cuando la combinación actual no está presente en el TLB es cuando se lleva a cabo el mecanismo de la figura 3-42 y se actualiza el TLB. Mientras los fracasos del TLB sean raros, el rendimiento será bueno.

También vale la pena observar que si alguna aplicación no necesita segmentación pero está contenta con un solo espacio de direcciones paginado de 32 bits, ese modelo es posible. Todos los registros de segmento se pueden establecer con el mismo selector, cuyo descriptor tiene *Base* = 0 y *Límite* establecido al máximo. Entonces el desplazamiento de la instrucción será la dirección lineal, con sólo un espacio de direcciones único utilizado, en efecto, una paginación normal. De hecho, todos los sistemas operativos actuales para el Pentium funcionan de esta manera. OS/2 fue el único que utilizó todo el poder de la arquitectura Intel MMU.

Con todo, hay que dar crédito a los diseñadores del Pentium. Dadas las metas conflictivas por implementar la paginación pura, la segmentación pura y los segmentos paginados, al tiempo que debía ser compatible con el 286, y todo esto había que hacerlo con eficiencia. El diseño resultante es sorprendentemente simple y limpio.

Aunque hemos cubierto en forma breve la arquitectura completa de la memoria virtual del Pentium, vale la pena decir unas cuantas palabras acerca de la protección, ya que este tema está estrechamente relacionado con la memoria virtual. Al igual que el esquema de memoria virtual está modelado en forma muy parecida a MULTICS, el sistema de protección también lo está. El Pentium admite cuatro niveles de protección, donde el nivel 0 es el más privilegiado y el 3 el menos privilegiado. Éstos se muestran en la figura 3-43. En cada instante, un programa en ejecución se encuentra en cierto nivel, indicado por un campo de 2 bits en su PSW. Cada segmento en el sistema también tiene un nivel.

Mientras que un programa se restrinja a sí mismo a utilizar segmentos en su propio nivel, todo funcionará bien. Se permiten los intentos de acceder a los datos en un nivel más alto, pero los de acceder a los datos en un nivel inferior son ilegales y producen traps. Los intentos de llamar procedimientos en un nivel distinto (mayor o menor) se permiten, pero de una manera controlada cuidadosamente. Para realizar una llamada entre niveles, la instrucción CALL debe contener un selector en vez de una dirección. Este selector designa a un descriptor llamado **compuerta de llamada**, el cual proporciona la dirección del procedimiento al que se va a llamar. Por ende, no es posible saltar en medio de un segmento de código arbitrario en un nivel distinto. Sólo se pueden utilizar puntos de entrada oficiales. Los conceptos de los niveles de protección y las compuertas de llamada se utilizaron por primera vez en MULTICS, en donde se denominaron **anillos de protección**.

Un uso común para este mecanismo se sugiere en la figura 3-43. En el nivel 0 encontramos el kernel del sistema operativo, que se encarga de la E/S, la administración de memoria y otras cuestiones críticas. En el nivel 1 está presente el manejador de llamadas al sistema. Los programas de usuario pueden llamar procedimientos aquí para llevar a cabo las llamadas al sistema, pero sólo se puede llamar a una lista de procedimientos específicos y protegidos. El nivel 2 contiene procedimientos de biblioteca, posiblemente compartida entre muchos programas en ejecución. Los progra-

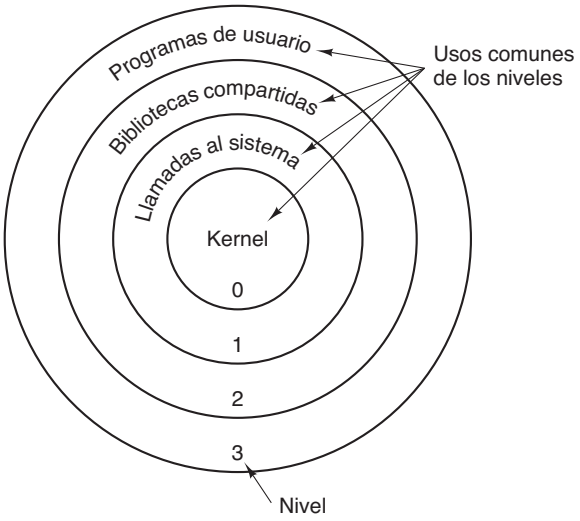

Figura 3-43. La protección en el Pentium.

mas de usuario pueden llamar a estos procedimientos y leer sus datos, pero no pueden modificarlos. Por último, los programas de usuario se ejecutan en nivel 3, que tiene la menor protección.

Los traps y las interrupciones utilizan un mecanismo similar a las compuertas de llamadas. También hacen referencia a los descriptores en vez de direcciones absolutas, y estos descriptores apuntan a los procedimientos específicos que se van a ejecutar. El campo *Tipo* en la figura 3-40 indica las diferencias entre los segmentos de código, los segmentos de datos y los diversos tipos de compuertas.

3.8 INVESTIGACIÓN ACERCA DE LA ADMINISTRACIÓN DE MEMORIA

La administración de memoria, en especial los algoritmos de paginación, fue alguna vez un área fructífera para la investigación, pero la mayor parte de eso parece haber desaparecido desde hace mucho tiempo, por lo menos para los sistemas de propósito general. La mayoría de los sistemas reales tienden a utilizar cierta variación sobre el reloj, debido a que es fácil de implementar y relativamente efectivo. Sin embargo, una excepción reciente es un rediseño del sistema de memoria virtual de BSD 4.4 (Cranor y Prulkar, 1999).

Sin embargo, aún se están realizando investigaciones sobre la paginación en los tipos más recientes de sistemas. Por ejemplo, los teléfonos celulares y los PDAs se han convertido en pequeñas PCs y muchps de ellas paginan la RAM al “disco”, sólo que el disco en un teléfono celular es la memoria flash, que tiene propiedades distintas a las de un disco magnético giratorio. Cierto trabajo reciente se reporta (In y colaboradores, 2007; Joo y colaboradores, 2006; y Park y colaboradores, 2004a). Park y colaboradores (2004b) también han analizado la paginación por demanda consciente de la energía en los dispositivos móviles.

También se están realizando investigaciones acerca del modelado del rendimiento de la paginación (Albers y colaboradores, 2002; Burton y Kelly, 2003; Cascaval y colaboradores, 2005; Panagiotou y Souza, 2006; y Peserico, 2003). También es de interés la administración de memoria para los sistemas multimedia (Dasigenis y colaboradores, 2001; Hand, 1999) y los sistemas de tiempo real (Pizlo y Vitek, 2006).

3.9 RESUMEN

En este capítulo examinamos la administración de memoria. Vimos que los sistemas más simples no realizan intercambios ni paginaciones. Una vez que se carga un programa en la memoria, permanece ahí hasta que termina. Algunos sistemas operativos sólo permiten un proceso a la vez en la memoria, mientras que otros soportan la multiprogramación.

El siguiente paso es el intercambio. Cuando se utiliza esta técnica, el sistema puede manejar más procesos de los que puede alojar en la memoria. Los procesos para los cuales no haya espacio se intercambian hacia el disco. Se puede llevar el registro del espacio libre en la memoria y en el disco con un mapa de bits o una lista de huecos.

A menudo, las computadoras modernas tienen cierta forma de memoria virtual. En su forma más simple, el espacio de direcciones de cada proceso se divide en bloques de tamaño uniforme llamados páginas, que pueden colocarse en cualquier marco de página disponible en la memoria. Hay muchos algoritmos de reemplazo de páginas; dos de los mejores algoritmos son el de envejecimiento y WSClock.

Para modelar los sistemas de paginación hay que abstraer la cadena de referencia de página del programa y utilizar la misma cadena de referencia con distintos algoritmos. Estos modelos se pueden utilizar para hacer algunas predicciones acerca del comportamiento de la paginación.

Para hacer que los sistemas de paginación funcionen bien, no basta con elegir un algoritmo; hay que poner atención en cuestiones tales como determinar el conjunto de trabajo, la política de asignación de memoria y el tamaño de página.

La segmentación ayuda a manejar las estructuras de datos que pueden cambiar de tamaño durante la ejecución, y simplifica la vinculación y la compartición. También facilita la provisión de distintos tipos de protección para distintos segmentos. Algunas veces la segmentación y la paginación se combinan para proporcionar una memoria virtual bidimensional. El sistema MULTICS y el Intel Pentium soportan segmentación y paginación.

PROBLEMAS

1. En la figura 3-3 los registros base y límite contienen el mismo valor, 16,384. ¿Es esto un accidente, o siempre son iguales? Si esto es sólo un accidente, ¿por qué son iguales en este ejemplo?

2. Un sistema de intercambio elimina huecos mediante la compactación. Suponiendo una distribución aleatoria de muchos huecos y muchos segmentos de datos y un tiempo de lectura o escritura en una palabra de memoria de 32 bits de 10 nseg, ¿aproximadamente cuánto tiempo se requiere para compactar 128 MB? Para simplificar, suponga que la palabra 0 es parte de un hueco y que la palabra más alta en la memoria contiene datos válidos.

3. En este problema tiene que comparar el almacenamiento necesario para llevar la cuenta de la memoria libre, utilizando un mapa de bits contra el uso de una lista ligada. La memoria de 128 MB se asigna en unidades de *n* bytes. Para la lista enlazada, suponga que la memoria consiste en una secuencia alternante de segmentos y huecos, cada uno de 64 KB. Suponga también que cada nodo en la lista enlazada necesita una dirección de memoria de 32 bits, una longitud de 16 bits y un campo para el siguiente nodo de 16 bits. ¿Cuánto bytes de almacenamiento se requieren para cada método? ¿Cuál es mejor?

4. Considere un sistema de intercambio en el que la memoria consiste en los siguientes tamaños de hueco, por orden de memoria: 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB y 15 KB. ¿Cuál hueco se toma para las siguientes solicitudes de segmento sucesivas:

   a) 12 KB
   b) 10 KB
   c) 9 KB

   para el algoritmo del primer ajuste? Ahora repita la pregunta para el mejor ajuste, peor ajuste y siguiente ajuste.

5. Para cada una de las siguientes direcciones virtuales decimales, calcule el número de página virtual y desplazamiento para una página de 4 KB y para una página de 8 KB: 20000, 32768, 60000.

6. El procesador Intel 8086 no admite memoria virtual. Sin embargo, algunas compañías vendían anteriormente sistemas que contenían una CPU 8086 sin modificaciones y realizaba la paginación. Trate de llegar a una conclusión lógica acerca de cómo lo hicieron. *Sugerencia:* piense acerca de la ubicación lógica de la MMU.

7. Considere el siguiente programa en C:

   ```
   int X[N];
   int paso = M; // M es una constante predefinida
   for (int i = 0; i < N; i += paso) X[i] = X[i] + 1;
   ```

   a) Si este programa se ejecuta en una máquina con un tamaño de página de 4 KB y un TLB con 64 entradas, ¿qué valores de M y N harán que un TLB falle para cada ejecución del ciclo interno?

   b) ¿Sería distinta su respuesta al inciso a) si el ciclo se repitiera muchas veces? Explique.

8. La cantidad de espacio en disco que debe estar disponible para el almacenamiento de páginas está relacionada con el número máximo de procesos *n,*, el número de bytes en el espacio de direcciones virtual *v,* así como con el número de bytes de RAM *r*. Proporcione una expresión para los requerimientos de espacio en disco en el peor de los casos. ¿Qué tan realista es esa cantidad?

9. Una máquina tiene un espacio de direcciones de 32 bits y una página de 8 KB. La tabla de páginas está completamente en el hardware, con una palabra de 32 bits por cada entrada. Cuando se inicia un proceso, la tabla de páginas se copia al hardware desde la memoria, una palabra por cada 100 nseg. Si cada proceso se ejecuta durante 100 mseg (incluyendo el tiempo para cargar la tabla de páginas), ¿qué fracción del tiempo de la CPU se dedica a cargar las tablas de páginas?

10. Suponga que una máquina tiene direcciones virtuales de 48 bits y direcciones físicas de 32 bits.

   a) Si las páginas son de 4 KB, ¿Cuántas entradas hay en la tabla de páginas si sólo hay un nivel? Explique.

b) Suponga que el mismo sistema tiene un TLB (Búfer de traducción adelantada) con 32 entradas. Además, suponga que un programa contiene instrucciones que caben en una página y lee secuencialmente elementos enteros largos de un arreglo que abarca miles de páginas. ¿Qué tan efectivo será el TLB para este caso?

11. Suponga que una máquina tiene direcciones virtuales de 38 bits y direcciones físicas de 32 bits.

a) ¿Cuál es la principal ventaja de una tabla de páginas de multinivel sobre una tabla de páginas de un solo nivel?

b) Con una tabla de páginas de dos niveles, páginas de 16 KB y entradas de 4 bytes, ¿cuántos bits se deben asignar para el campo de la tabla de páginas de nivel superior y cuántos para el campo de la tabla de páginas del siguiente nivel? Explique.

12. Una computadora con una dirección de 32 bits utiliza una tabla de páginas de dos niveles. Las direcciones virtuales se dividen en un campo de la tabla de páginas de nivel superior de 9 bits, un campo de la tabla de páginas de segundo nivel de 11 bits y un desplazamiento. ¿Qué tan grandes son las páginas y cuántas hay en el espacio de direcciones?

13. Suponga que una dirección virtual de 32 bits se divide en cuatro campos: *a*, *b*, *c* y *d*. Los primeros tres se utilizan para un sistema de tablas de páginas de tres niveles. El cuarto campo (*d*) es el desplazamiento. ¿Depende el número de páginas de los tamaños de los cuatro campos? Si no es así, ¿cuáles importan y cuáles no?

14. Una computadora tiene direcciones virtuales de 32 bits y páginas de 4 KB. El programa y los datos caben juntos en la página más baja (0 a 4095). La pila cabe en la página más alta. ¿Cuántas entradas se necesitan en la tabla de páginas si se utiliza la paginación tradicional (un nivel)? ¿Cuántas entradas en la tabla de páginas se necesitan para la paginación de dos niveles, con 10 bits en cada parte?

15. Una computadora cuyos procesos tienen 1024 páginas en sus espacios de direcciones mantiene sus tablas de páginas en memoria. La sobrecarga requerida para leer una palabra de la tabla de páginas es 5 nseg. Para reducir esta sobrecarga, la computadora tiene un TLB que contiene 32 pares (página virtual, marco de página física) y puede realizar una búsqueda en 1 nseg. ¿Qué proporción de aciertos necesita para reducir la sobrecarga promedio a 2 nseg?

16. El TLB en la VAX no contiene un bit *R*. ¿Por qué?

17. ¿Cómo puede implementarse en hardware el dispositivo de memoria asociativa necesario para implementar una TLB y cuáles son las implicaciones de dicho diseño para que sea expandible?

18. Una máquina tiene direcciones virtuales de 48 bits y direcciones físicas de 32 bits. Las páginas son de 8 KB. ¿Cuántas entradas se necesitan para la tabla de páginas?

19. Una computadora con una página de 8 KB, una memoria principal de 256 KB y un espacio de direcciones virtuales de 64 GB utiliza una tabla de páginas invertida para implementar su memoria virtual. ¿Qué tan grande debe ser la tabla de hash para asegurar una cadena de hash de una longitud promedio menor a 1? Suponga que el tamaño de la tabla de hash es una potencia de dos.

20. Un estudiante en un curso de diseño de compiladores propone al profesor un proyecto de escribir un compilador que produzca una lista de referencias a páginas que se puedan utilizar para implementar el algoritmo de reemplazo de páginas óptimo. ¿Es esto posible? ¿Por qué sí o por qué no? ¿Hay algo que pudiera hacerse para mejorar la eficiencia de la paginación en tiempo de ejecución?

21. Suponga que el flujo de referencia de páginas virtuales contiene repeticiones de largas secuencias de referencias a páginas, seguidas ocasionalmente por una referencia a una página aleatoria. Por ejemplo, la secuencia 0, 1, …, 511, 431, 0, 1, …, 511, 332, 0, 1, … consiste en repeticiones de la secuencia 0, 1, …, 511 seguida de una referencia aleatoria a las páginas 431 y 332.

a) ¿Por qué no serían efectivos los algoritmos de sustitución estándar (LRU, FIFO, Reloj) al manejar esta carga de trabajo para una asignación de página que sea menor que la longitud de la secuencia?
b) Si a este programa se le asignaran 500 marcos de página, describa un método de sustitución de página que tenga un rendimiento mucho mejor que los algoritmos LRU, FIFO o Reloj.

22. Si se utiliza el algoritmo FIFO de reemplazo de páginas con cuatro marcos de página y ocho páginas, ¿cuántos fallos de página ocurrirán con la cadena de referencia 0172327103 si los cuatro marcos están vacíos al principio? Ahora repita este problema para el algoritmo LRU.

23. Considere la secuencia de páginas de la figura 3-15(b). Suponga que los bits *R* para las páginas de la *B* a la *A* son 11011011, respectivamente, ¿Cuál página eliminará el algoritmo de segunda oportunidad?

24. Una pequeña computadora tiene cuatro marcos de página. En el primer pulso de reloj, los bits *R* son 0111 (la página 0 es 0, el resto son 1). En los siguientes pulsos de reloj, los valores son 1011, 1010, 1101, 0010, 1010, 1100 y 0001. Si se utiliza el algoritmo de envejecimiento con un contador de 8 bits, proporcione los valores de los cuatro contadores después del último pulso.

25. Dé un ejemplo simple de una secuencia de referencias a páginas en donde la primera página seleccionada para la sustitución sea diferente para los algoritmos de reemplazo de páginas de reloj y LRU. Suponga que a un proceso se le asignan 3 marcos y que la cadena de referencia contiene números de página del conjunto 0, 1, 2, 3.

26. En el algoritmo WSClock de la figura 3-21(c), la manecilla apunta a una página con $R = 0$. Si $\tau = 400$, ¿se eliminará esta página? ¿Qué pasa si $\tau = 1000$?

27. ¿Cuánto tiempo se requiere para cargar un programa de 64 KB de un disco cuyo tiempo de búsqueda promedio es de 10 mseg, cuyo tiempo de rotación es de 10 mseg y cuyas pistas contienen 32 KB

a) para un tamaño de página de 2 KB?
b) para un tamaño de página de 4 KB?

Las páginas están esparcidas de manera aleatoria alrededor del disco y el número de cilindros es tan grande que la probabilidad de que dos páginas se encuentren en el mismo cilindro es insignificante.

28. Una computadora tiene cuatro marcos de página. El tiempo de carga, tiempo del último acceso y los bits *R* y *M* para cada página se muestran a continuación (los tiempos están en pulsos de reloj):

Página	Cargada	Última referencia	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

a) ¿Cuál página reemplazará el algoritmo NRU?
b) ¿Cuál página reemplazará el algoritmo FIFO?
c) ¿Cuál página reemplazará el algoritmo LRU?
d) ¿Cuál página reemplazará el algoritmo de segunda oportunidad?

29. Considere el siguiente arreglo bidimensional:

```
int X[64][64];
```

Suponga que un sistema tiene cuatro marcos de página y que cada marco es de 128 palabras (un entero ocupa una palabra). Los programas que manipulan el arreglo X caben exactamente en una página y siempre ocupan la página 0. Los datos se intercambian hacia dentro y hacia fuera de los otros tres marcos. El arreglo *X* se almacena en orden de importancia por filas (es decir, *X*[0][1] va después de *X*[0][0] en la memoria). ¿Cuál de los dos fragmentos de código que se muestran a continuación generarán el menor número de fallos de página? Explique y calcule el número total de fallos de página.

Fragmento A

```
for (int j = 0; j < 64; j++)
  for (int i = 0; i < 64; i++) X[i][j] = 0;
```

Fragmento B

```
for (int i = 0; i < 64; i++)
  for (int j = 0; j < 64; j++) X[i][j] = 0;
```

30. Una de las primeras máquinas de tiempo compartido (la PDP-1) tenía una memoria de 4K palabras de 18 bits. Contenía un proceso a la vez en la memoria. Cuando el planificador de proceso decidía ejecutar otro proceso, el proceso en memoria se escribía en un tambor de paginación, con 4K palabras de 18 bits alrededor de la circunferencia del tambor, el cual podía empezar a escribir (o leer) en cualquier palabra, en vez de hacerlo sólo en la palabra 0. ¿Supone usted que este tambor fue seleccionado?

31. Una computadora proporciona a cada proceso 65,536 bytes de espacio de direcciones, dividido en páginas de 4096 bytes. Un programa específico tiene un tamaño de texto de 32,768 bytes, un tamaño de datos de 16,386 bytes y un tamaño de pila de 15,870 bytes. ¿Cabrá este programa en el espacio de direcciones? Si el tamaño de página fuera de 512 bytes, ¿cabría? Recuerde que una página no puede contener partes de dos segmentos distintos.

32. ¿Puede una página estar en dos conjuntos de trabajo al mismo tiempo? Explique.

33. Se ha observado que el número de instrucciones ejecutadas entre fallos de página es directamente proporcional al número de marcos de página asignados a un programa. Si la memoria disponible se duplica, el intervalo promedio entre los fallos de página también se duplica. Suponga que una instrucción normal requiere 1 microsegundo, pero si ocurre un fallo de página, requiere 2001 μseg (es decir, 2 mseg para hacerse cargo del fallo). Si un programa requiere 60 segundos para ejecutarse, tiempo durante el cual obtiene 15,000 fallos de página, ¿cuánto tiempo requeriría para ejecutarse si hubiera disponible el doble de memoria?

34. Un grupo de diseñadores de sistemas operativos para la Compañía de Computadoras Frugal están ideando maneras de reducir la cantidad de almacenamiento de respaldo necesario en su nuevo sistema operativo. El jefe de ellos ha sugerido que no se deben preocupar por guardar el texto del programa en el área de intercambio, sino sólo paginarla directamente desde el archivo binario cada vez

que se necesite. ¿Bajo qué condiciones, si las hay, funciona esta idea para el texto del programa? ¿Bajo qué condiciones, si las hay, funciona para los datos?

35. Una instrucción en lenguaje máquina para cargar una palabra de 32 bits en un registro contiene la dirección de 32 bits de la palabra que se va a cargar. ¿Cuál es el número máximo de fallos de página que puede provocar esta instrucción?

36. Cuando se utilizan la segmentación y la paginación, como en MULTICS, primero se debe buscar el descriptor del segmento y después el descriptor de página. ¿Funciona el TLB también de esta manera, con dos niveles de búsqueda?

37. Consideremos un programa que tiene los dos segmentos que se muestran a continuación, los cuales consisten de instrucciones en el segmento 0 y datos de lectura/escritura en el segmento 1. El segmento 0 tiene protección de lectura/ejecución y el segmento 1 tiene protección de lectura/escritura. El sistema de memoria es un sistema de memoria virtual con paginación bajo demanda, con direcciones virtuales que tienen un número de página de 4 bits y un desplazamiento de 10 bits. Las tablas de páginas y la protección son las siguientes (todos los números en la tabla están en decimal):

Segmento 0		**Segmento 1**	
Lectura/ejecución		**Lectura/escritura**	
# de página virtual	**# de marco de página**	**# delpágina virtual**	**# de marco de página**
0	2	0	En disco
1	En disco	1	14
2	11	2	9
3	5	3	6
4	En disco	4	En disco
5	En disco	5	13
6	4	6	8
7	3	7	12

Para cada uno de los siguientes casos, proporcione la dirección de memoria real (actual) que resulta de la traducción de direcciones dinámicas o identifique el tipo de fallo que ocurre (ya sea fallo de página o de protección).

a) Obtener del segmento 1, página 1, desplazamiento 3
b) Almacenar en segmento 0, página 0, desplazamiento 16
c) Obtener del segmento 1, página 4, desplazamiento 28
d) Saltar a la ubicación en el segmento 1, página 3, desplazamiento 32

38. ¿Puede pensar en alguna situación en donde el soporte de la memoria virtual fuera una mala idea y qué se ganaría al no tener que soportar la memoria virtual? Explique.

39. Trace un histograma y calcule la media y la mediana de los tamaños de los archivos binarios ejecutables en una computadora a la que tenga acceso. En un sistema Windows, analice todos los archivos .exe y .dll; en un sistema UNIX analice todos los archivos ejecutables en */bin*, */usr/bin* y

/local/bin que no sean secuencias de comandos (o utilice la herramienta *file* para buscar todos los ejecutables). Determine el tamaño de página óptimo para esta computadora, considerando sólo el código (no los datos). Considere la fragmentación interna y el tamaño de la tabla de páginas, haciendo alguna suposición razonable acerca del tamaño de una entrada en la tabla de páginas. Suponga que todos los programas tienen la misma probabilidad de ejecutarse, y por ende deben considerarse con el mismo peso.

40. Los pequeños programas para MS-DOS se pueden compilar como archivos *.COM*. Estos archivos siempre se cargan en la dirección 0x100 en un solo segmento de memoria que se utilice para código, datos y pila. Las instrucciones que transfieren el control de la ejecución, como **JMP** y **CALL**, o que acceden a datos estáticos desde direcciones fijas hacen que las instrucciones se compilen en el código objeto. Escriba un programa que pueda reubicar dicho archivo de programa para ejecutarlo empezando en una dirección arbitraria. Su programa debe explorar el código en busca de códigos objeto para instrucciones que hagan referencia a direcciones de memoria fijas, después debe modificar esas direcciones que apunten a ubicaciones de memoria dentro del rango a reubicar. Encontrará los códigos objeto en un libro de programación en lenguaje ensamblador. Tenga en cuenta que hacer esto perfectamente sin información adicional es, en general, una tarea imposible debido a que ciertas palabras de datos pueden tener valores similares a los códigos objeto de las instrucciones.

41. Escriba un programa que simule un sistema de paginación utilizando el algoritmo de envejecimiento. El número de marcos de página es un parámetro. La secuencia de referencias a páginas debe leerse de un archivo. Para un archivo de entrada dado, dibuje el número de fallos de página por cada 1000 referencias a memoria como una función del número de marcos de página disponibles.

42. Escriba un programa para demostrar el efecto de los fallos del TLB en el tiempo de acceso efectivo a la memoria, midiendo el tiempo por cada acceso que se requiere para recorrer un arreglo extenso.

a) Explique los conceptos principales detrás del programa y describa lo que espera que muestre la salida para alguna arquitectura de memoria virtual práctica.
b) Ejecute el programa en una computadora y explique qué tan bien se ajustan los datos a sus expectativas.
c) Repita la parte b) pero para una computadora más antigua con una arquitectura distinta y explique cualquier diferencia importante en la salida.

43. Escriba un programa que demuestre la diferencia entre el uso de una política de reemplazo de páginas local y una global para el caso simple de dos procesos. Necesitará una rutina que pueda generar una cadena de referencias a páginas basado en un modelo estadístico. Este modelo tiene *N* estados enumerados de 0 a N-1, los cuales representan cada una de las posibles referencias a páginas y una probabilidad p_1 asociada con cada estado *i* que represente la probabilidad de que la siguiente referencia sea a la misma página. En caso contrario, la siguiente referencia a una página será a una de las otras páginas con igual probabilidad.

a) Demuestre que la rutina de generación de la cadena de referencias a páginas se comporta en forma apropiada para cierta *N* pequeña.
b) Calcule la proporción de fallos de página para un pequeño ejemplo en el que hay un proceso y un número fijo de marcos de página. Explique por qué es correcto el comportamiento.
c) Repita la parte b) con dos procesos con secuencias de referencias a páginas independientes, y el doble de marcos de página que en la parte (b).
d) Repita la parte c) utilizando una política global en vez de una local. Además, compare la proporción de fallos de página por proceso con la del método de política local.

4

SISTEMAS DE ARCHIVOS

Todas las aplicaciones de computadora requieren almacenar y recuperar información. Mientras un proceso está en ejecución, puede almacenar una cantidad limitada de información dentro de su propio espacio de direcciones. Sin embargo, la capacidad de almacenamiento está restringida por el tamaño del espacio de direcciones virtuales. Para algunas aplicaciones este tamaño es adecuado; para otras, tales como las de reservaciones en aerolíneas, las bancarias o las de contabilidad corporativa, puede ser demasiado pequeño.

Un segundo problema relacionado con el mantenimiento de la información dentro del espacio de direcciones de un proceso es que cuando el proceso termina, la información se pierde. Para muchas aplicaciones (por ejemplo, una base de datos) la información se debe retener durante semanas, meses o incluso indefinidamente. Es inaceptable que esta información se desvanezca cuando el proceso que la utiliza termine. Además, no debe desaparecer si una falla en la computadora acaba con el proceso.

Un tercer problema es que frecuentemente es necesario que varios procesos accedan a (partes de) la información al mismo tiempo. Si tenemos un directorio telefónico en línea almacenado dentro del espacio de direcciones de un solo proceso, sólo ese proceso puede tener acceso al directorio. La manera de resolver este problema es hacer que la información en sí sea independiente de cualquier proceso.

En consecuencia, tenemos tres requerimientos esenciales para el almacenamiento de información a largo plazo:

1. Debe ser posible almacenar una cantidad muy grande de información.
2. La información debe sobrevivir a la terminación del proceso que la utilice.
3. Múltiples procesos deben ser capaces de acceder a la información concurrentemente.

Durante muchos años se han utilizado discos magnéticos para este almacenamiento de largo plazo, así como cintas y discos ópticos, aunque con un rendimiento mucho menor. En el capítulo 5 estudiaremos más sobre los discos, pero por el momento basta con pensar en un disco como una secuencia lineal de bloques de tamaño fijo que admite dos operaciones:

1. Leer el bloque *k*.
2. Escribir el bloque *k*.

En realidad hay más, pero con estas dos operaciones podríamos (en principio) resolver el problema del almacenamiento a largo plazo.

Sin embargo, éstas son operaciones muy inconvenientes, en especial en sistemas extensos utilizados por muchas aplicaciones y tal vez varios usuarios (por ejemplo, en un servidor). Unas cuantas de las preguntas que surgen rápidamente son:

1. ¿Cómo encontramos la información?
2. ¿Cómo evitamos que un usuario lea los datos de otro usuario?
3. ¿Cómo sabemos cuáles bloques están libres?

y hay muchas más.

Así como vimos la manera en que el sistema operativo abstrajo el concepto del procesador para crear la abstracción de un proceso y el concepto de la memoria física para ofrecer a los procesos espacios de direcciones (virtuales), podemos resolver este problema con una nueva abstracción: el archivo. En conjunto, las abstracciones de los procesos (e hilos), espacios de direcciones y archivos son los conceptos más importantes en relación con los sistemas operativos. Si realmente comprende estos tres conceptos de principio a fin, estará preparado para convertirse en un experto en sistemas operativos.

Los **archivos** son unidades lógicas de información creada por los procesos. En general, un disco contiene miles o incluso millones de archivos independientes. De hecho, si concibe a cada archivo como un tipo de espacio de direcciones, no estará tan alejado de la verdad, excepto porque se utilizan para modelar el disco en vez de modelar la RAM.

Los procesos pueden leer los archivos existentes y crear otros si es necesario. La información que se almacena en los archivos debe ser **persistente**, es decir, no debe ser afectada por la creación y terminación de los procesos. Un archivo debe desaparecer sólo cuando su propietario lo remueve de manera explícita. Aunque las operaciones para leer y escribir en archivos son las más comunes, existen muchas otras, algunas de las cuales examinaremos a continuación.

Los archivos son administrados por el sistema operativo. La manera en que se estructuran, denominan, abren, utilizan, protegen, implementan y administran son tópicos fundamentales en el diseño de sistemas operativos. La parte del sistema operativo que trata con los archivos se conoce como **sistema de archivos** y es el tema de este capítulo.

Desde el punto de vista del usuario, el aspecto más importante de un sistema de archivos es su apariencia; es decir, qué constituye un archivo, cómo se denominan y protegen los archivos qué operaciones se permiten con ellos, etcétera. Los detalles acerca de si se utilizan listas enlazadas (ligadas) o mapas de bits para llevar la cuenta del almacenamiento libre y cuántos sectores hay en un bloque de disco lógico no son de interés, aunque sí de gran importancia para los diseñadores del sis-

tema de archivos. Por esta razón hemos estructurado el capítulo en varias secciones: las primeras dos están relacionadas con la interfaz del usuario para los archivos y directorios, respectivamente; después incluimos un análisis detallado acerca de cómo se implementa y administra el sistema de archivos; por último, daremos algunos ejemplos de sistemas de archivos reales.

4.1 ARCHIVOS

En las siguientes páginas analizaremos los archivos desde el punto de vista del usuario; es decir, cómo se utilizan y qué propiedades tienen.

4.1.1 Nomenclatura de archivos

Los archivos son un mecanismo de abstracción. Proporcionan una manera de almacenar información en el disco y leerla después. Esto se debe hacer de tal forma que se proteja al usuario de los detalles acerca de cómo y dónde se almacena la información y cómo funcionan los discos en realidad.

Probablemente, la característica más importante de cualquier mecanismo de abstracción sea la manera en que los objetos administrados son denominados, por lo que empezaremos nuestro examen de los sistemas de archivos con el tema de la nomenclatura de los archivos. Cuando un proceso crea un archivo le proporciona un nombre. Cuando el proceso termina, el archivo continúa existiendo y puede ser utilizado por otros procesos mediante su nombre.

Las reglas exactas para denominar archivos varían un poco de un sistema a otro, pero todos los sistemas operativos actuales permiten cadenas de una a ocho letras como nombres de archivos legales. Por ende, *andrea*, *bruce* y *cathy* son posibles nombres de archivos. Con frecuencia también se permiten dígitos y caracteres especiales, por lo que nombres como *2*, *urgente!* y *Fig.2-14* son a menudo válidos también. Muchos sistemas de archivos admiten nombres de hasta 255 caracteres.

Algunos sistemas de archivos diferencian las letras mayúsculas de las minúsculas, mientras que otros no. UNIX cae en la primera categoría; MS-DOS en la segunda. Así, un sistema UNIX puede tener los siguientes nombres como tres archivos distintos: *maria*, *Maria* y *MARIA*. En MS-DOS, todos estos nombres se refieren al mismo archivo.

Tal vez sea adecuado hacer en este momento un paréntesis sobre sistemas de archivos. Windows 95 y Windows 98 utilizan el sistema de archivos de MS-DOS conocido como **FAT-16** y por ende heredan muchas de sus propiedades, como la forma en que se construyen sus nombres. Windows 98 introdujo algunas extensiones a FAT-16, lo cual condujo a **FAT-32**, pero estos dos sistemas son bastante similares. Además, Windows NT, Windows 2000, Windows XP y .WV admiten ambos sistemas de archivos FAT, que en realidad ya son obsoletos. Estos cuatro sistemas operativos basados en NT tienen un sistema de archivos nativo (NTFS) con diferentes propiedades (como los nombres de archivos en Unicode). En este capítulo, cuando hagamos referencia a los sistemas de archivos MS-DOS o FAT, estaremos hablando de FAT-16 y FAT-32 como se utilizan en Windows, a menos que se especifique lo contrario. Más adelante en este capítulo analizaremos los sistemas de archivos FAT y en el capítulo 11 examinaremos el sistema de archivos NTFS, donde analizaremos Windows Vista con detalle.

Muchos sistemas operativos aceptan nombres de archivos en dos partes, separadas por un punto, como en *prog.c*. La parte que va después del punto se conoce como la **extensión del archivo** y por lo general indica algo acerca de su naturaleza. Por ejemplo, en MS-DOS, los nombres de archivos son de 1 a 8 caracteres, más una extensión opcional de 1 a 3 caracteres. En UNIX el tamaño de la extensión (si la hay) es a elección del usuario y un archivo puede incluso tener dos o más extensiones, como en *paginainicio.html.zip*, donde *.html* indica una página Web en HTML y *.zip* indica que el archivo se ha comprimido mediante el programa *zip*. Algunas de las extensiones de archivos más comunes y sus significados se muestran en la figura 4-1.

Extensión	Significado
archivo.bak	Archivo de respaldo
archivo.c	Programa fuente en C
archivo.gif	Imagen en Formato de Intercambio de Gráficos de CompuServe
archivo.hlp	Archivo de ayuda
archivo.html	Documento en el Lenguaje de Marcación de Hipertexto de World Wide Web
archivo.jpg	Imagen fija codificada con el estándar JPEG
archivo.mp3	Música codificada en formato de audio MPEG capa 3
archivo.mpg	Película codificada con el estándar MPEG
archivo.o	Archivo objeto (producido por el compilador, no se ha enlazado todavía)
archivo.pdf	Archivo en Formato de Documento Portable
archivo.ps	Archivo de PostScript
archivo.tex	Entrada para el programa formateador TEX
archivo.txt	Archivo de texto general
archivo.zip	Archivo comprimido

Figura 4-1. Algunas extensiones de archivos comunes.

En algunos sistemas (como UNIX) las extensiones de archivo son sólo convenciones y no son impuestas por los sistemas operativos. Un archivo llamado *archivo.txt* podría ser algún tipo de archivo de texto, pero ese nombre es más un recordatorio para el propietario que un medio para transportar información a la computadora. Por otro lado, un compilador de C podría insistir que los archivos que va a compilar terminen con *.c* y podría rehusarse a compilarlos si no tienen esa terminación.

Las convenciones como ésta son especialmente útiles cuando el mismo programa puede manejar diferentes tipos de archivos. Por ejemplo, el compilador C puede recibir una lista de varios archivos para compilarlos y enlazarlos, algunos de ellos archivos de C y otros archivos de lenguaje ensamblador. Entonces, la extensión se vuelve esencial para que el compilador sepa cuáles son archivos de C, cuáles son archivos de lenguaje ensamblador y cuáles son archivos de otro tipo.

Por el contrario, Windows está consciente de las extensiones y les asigna significado. Los usuarios (o procesos) pueden registrar extensiones con el sistema operativo y especificar para cada una cuál programa “posee” esa extensión. Cuando un usuario hace doble clic sobre un nombre de archivo, el programa asignado a su extensión de archivo se inicia con el archivo como parámetro. Por

ejemplo, al hacer doble clic en *archivo.doc* se inicia Microsoft *Word* con *archivo.doc* como el archivo inicial a editar.

4.1.2 Estructura de archivos

Los archivos se pueden estructurar en una de varias formas. Tres posibilidades comunes se describen en la figura 4-2. El archivo en la figura 4-2(a) es una secuencia de bytes sin estructura: el sistema operativo no sabe, ni le importa, qué hay en el archivo. Todo lo que ve son bytes. Cualquier significado debe ser impuesto por los programas a nivel usuario. Tanto UNIX como Windows utilizan esta metodología.

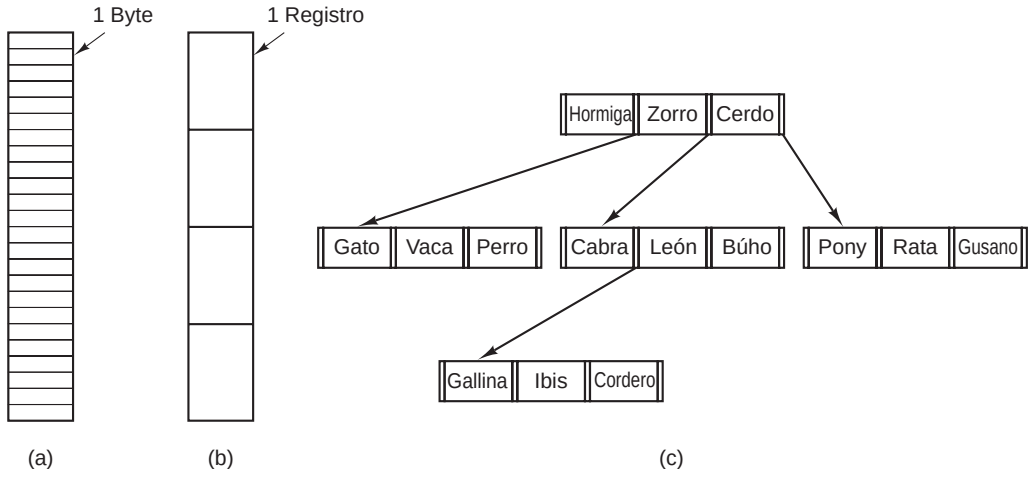

Figura 4-2. Tres tipos de archivos. (a) Secuencia de bytes. (b) Secuencia de registros. (c) Árbol.

Hacer que el sistema operativo considere los archivos sólo como secuencias de bytes provee la máxima flexibilidad. Los programas de usuario pueden colocar cualquier cosa que quieran en sus archivos y denominarlos de cualquier manera conveniente. El sistema operativo no ayuda, pero tampoco estorba. Para los usuarios que desean realizar cosas inusuales, esto último puede ser muy importante. Todas las versiones de UNIX, MS-DOS y Windows utilizan este modelo de archivos.

La primera configuración en la estructura se muestra en la figura 4-2(b). En este modelo, un archivo es una secuencia de registros de longitud fija, cada uno con cierta estructura interna. El concepto central para la idea de que un archivo sea una secuencia de registros es la idea de que la operación de lectura devuelva un registro y la operación de escritura sobrescriba o agregue un registro. Como nota histórica, hace algunas décadas, cuando reinaba la tarjeta perforada de 80 columnas, muchos sistemas operativos de mainframes basaban sus sistemas de archivos en archivos consistentes de registros de 80 caracteres; es decir, en imágenes de la tarjeta. Estos sistemas también admitían archivos con registros de 132 caracteres, que fueron destinados para la impresora de

línea (que en esos días eran grandes impresoras de cadena con 132 columnas). Los programas leían la entrada en unidades de 80 caracteres y la escribían en unidades de 132 caracteres, aunque los últimos 52 podían ser espacios, desde luego. Ningún sistema de propósito general de la actualidad utiliza ya este modelo como su sistema de archivos primario, pero en aquellos días de las tarjetas perforadas de 80 columnas y del papel de impresora de línea de 132 caracteres, éste era un modelo común en las computadoras mainframe.

El tercer tipo de estructura de archivo se muestra en la figura 4-2(c). En esta organización, un archivo consiste de un árbol de registros, donde no todos son necesariamente de la misma longitud; cada uno de ellos contiene un campo **llave** en una posición fija dentro del registro. El árbol se ordena con base en el campo llave para permitir una búsqueda rápida por una llave específica.

La operación básica aquí no es obtener el "siguiente" registro, aunque eso también es posible, sino obtener el registro con una llave específica. Para el archivo del zoológico de la figura 4-2(c), podríamos pedir al sistema que, por ejemplo, obtenga el registro cuya llave sea *pony*, sin preocuparnos acerca de su posición exacta en el archivo. Además, se pueden agregar nuevos registros al archivo, con el sistema operativo, y no el usuario, decidiendo dónde colocarlos. Evidentemente, este tipo de archivos es bastante distinto de los flujos de bytes sin estructura que se usan en UNIX y Windows, pero se utiliza de manera amplia en las grandes computadoras mainframe que aún se emplean en algún procesamiento de datos comerciales.

4.1.3 Tipos de archivos

Muchos sistemas operativos soportan varios tipos de archivos. Por ejemplo, UNIX y Windows tienen archivos y directorios regulares. UNIX también tiene archivos especiales de caracteres y de bloques. Los **archivos regulares** son los que contienen información del usuario. Todos los archivos de la figura 4-2 son archivos regulares. Los **directorios** son sistemas de archivos para mantener la estructura del sistema de archivos. Estudiaremos los directorios un poco más adelante. Los **archivos especiales de caracteres** se relacionan con la entrada/salida y se utilizan para modelar dispositivos de E/S en serie, tales como terminales, impresoras y redes. Los **archivos especiales de bloques** se utilizan para modelar discos. En este capítulo estaremos interesados principalmente en los archivos regulares.

Por lo general, los archivos regulares son archivos ASCII o binarios. Los archivos ASCII consisten en líneas de texto. En algunos sistemas, cada línea se termina con un carácter de retorno de carro. En otros se utiliza el carácter de avance de línea. Algunos sistemas (por ejemplo, MS-DOS) utilizan ambos. No todas las líneas necesitan ser de la misma longitud.

La gran ventaja de los archivos ASCII es que se pueden mostrar e imprimir como están, y se pueden editar con cualquier editor de texto. Además, si muchos programas utilizan archivos ASCII para entrada y salida, es fácil conectar la salida de un programa con la entrada de otro, como en las canalizaciones de shell. (La plomería entre procesos no es más fácil, pero la interpretación de la información lo es si una convención estándar, tal como ASCII, se utiliza para expresarla).

Otros archivos son binarios, lo cual sólo significa que no son archivos ASCII. Al listarlos en la impresora aparece un listado incomprensible de caracteres. Por lo general tienen cierta estructura interna conocida para los programas que los utilizan.

Por ejemplo, en la figura 4-3(a) vemos un archivo binario ejecutable simple tomado de una de las primeras versiones de UNIX. Aunque técnicamente el archivo es sólo una secuencia de bytes, el sistema operativo sólo ejecutará un archivo si tiene el formato apropiado. Este archivo tiene cinco secciones: encabezado, texto, datos, bits de reubicación y tabla de símbolos. El encabezado empieza con un supuesto **número mágico**, que identifica al archivo como un archivo ejecutable (para evitar la ejecución accidental de un archivo que no tenga este formato). Después vienen los tamaños de las diversas partes del archivo, la dirección en la que empieza la ejecución y algunos bits de bandera. Después del encabezado están el texto y los datos del programa en sí. Éstos se cargan en memoria y se reubican usando los bits de reubicación. La tabla de símbolos se utiliza para depurar.

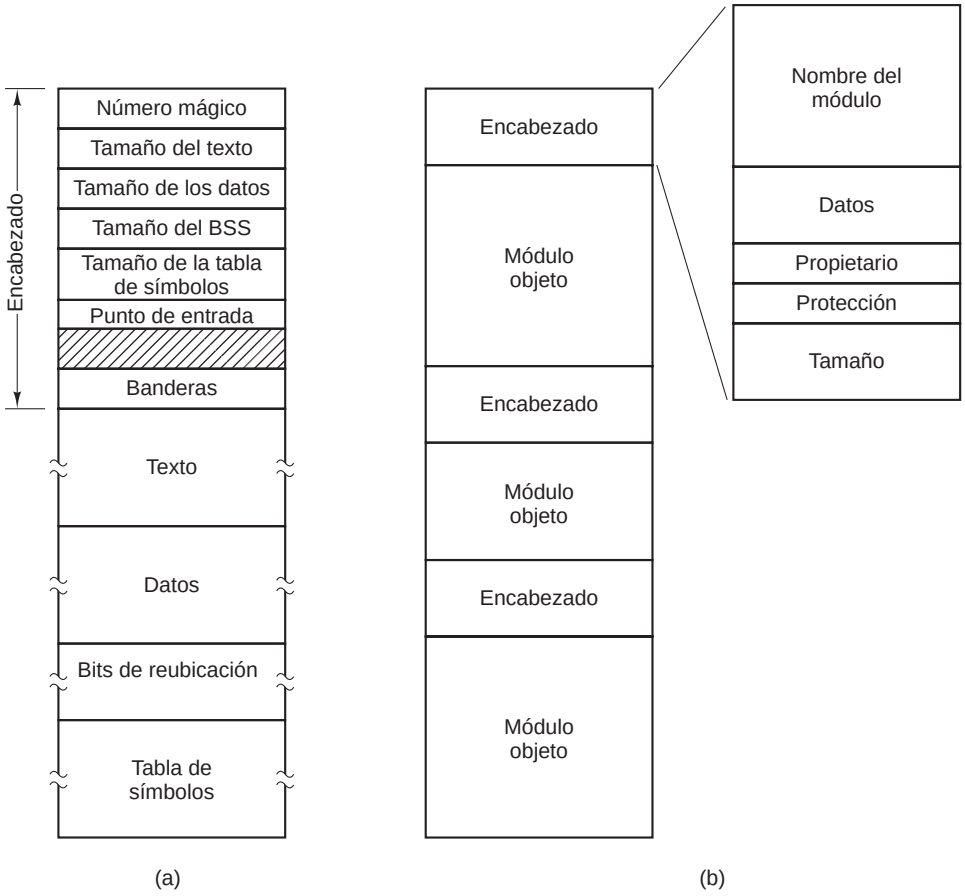

Figura 4-3. (a) Un archivo ejecutable. (b) Un archivo.

Nuestro segundo ejemplo de un archivo binario es un archivo, también de UNIX. Consiste en una colección de procedimientos (módulos) de biblioteca compilados, pero no enlazados. A cada uno se le antepone un encabezado que indica su nombre, fecha de creación, propietario, código de

protección y tamaño. Al igual que en el caso del archivo ejecutable, los encabezados de los módulos están llenos de números binarios. Al copiarlos a la impresora se produciría basura como salida.

Cada sistema operativo debe reconocer por lo menos un tipo de archivo —su propio archivo ejecutable— y algunos reconocen más. El antiguo sistema TOPS-20 (para el DECsystem 20) hacía algo más, ya que examinaba la hora de creación de cualquier archivo a ejecutar. Después localizaba el archivo de código fuente y veía si éste se había modificado desde la última vez que se creó el binario. Si así era, recompilaba automáticamente el código fuente. En términos de UNIX, el programa *make* había sido integrado al shell. Las extensiones de archivo eran obligatorias, por lo que el sistema operativo podía saber cuál programa binario se derivaba de cuál fuente.

Tener archivos fuertemente tipificados como éstos ocasiona problemas cada vez que el usuario hace algo que los diseñadores del sistema no esperaban. Por ejemplo, considere un sistema en el que los archivos de salida de un programa tienen la extensión *.dat* (archivos de datos). Si un usuario escribe un programa formateador que lea un archivo *.c* (programa de C), lo transforma (por ejemplo, convirtiéndolo a un esquema de sangría estándar) y después escribe el archivo transformado como salida, el archivo de salida será de tipo *.dat*. Si el usuario trata de ofrecer esto al compilador de C para que lo compile, el sistema se rehusará debido a que tienen la extensión incorrecta. Los intentos de copiar *archivo.dat* a *archivo.c* serán rechazados por el sistema como inválidos (para proteger al usuario contra los errores).

Mientras que este tipo de "amabilidad con el usuario" puede ayudar a los novatos, vuelve locos a los usuarios experimentados debido a que tienen que dedicar un esfuerzo considerable para sortear la idea del sistema operativo en cuanto a lo que es razonable y lo que no.

4.1.4 Acceso a archivos

Los primeros sistemas operativos proporcionaban sólo un tipo de acceso: **acceso secuencial**. En estos sistemas, un proceso podía leer todos los bytes o registros en un archivo en orden, empezando desde el principio, pero no podía saltar algunos y leerlos fuera de orden. Sin embargo, los archivos secuenciales podían rebobinarse para poder leerlos todas las veces que fuera necesario. Los archivos secuenciales eran convenientes cuando el medio de almacenamiento era cinta magnética en vez de disco.

Cuando se empezó a usar discos para almacenar archivos, se hizo posible leer los bytes o registros de un archivo fuera de orden, pudiendo acceder a los registros por llave en vez de posición. Los archivos cuyos bytes o registros se pueden leer en cualquier orden se llaman **archivos de acceso aleatorio**. Son requeridos por muchas aplicaciones.

Los archivos de acceso aleatorio son esenciales para muchas aplicaciones, como los sistemas de bases de datos. Si el cliente de una aerolínea llama y desea reservar un asiento en un vuelo específico, el programa de reservación debe poder tener acceso al registro para ese vuelo sin tener que leer primero los miles de registros de otros vuelos.

Es posible utilizar dos métodos para especificar dónde se debe empezar a leer. En el primero, cada operación `read` da la posición en el archivo en la que se va a empezar a leer. En el segundo se provee una operación especial (`seek`) para establecer la posición actual. Después de una operación `seek`, el archivo se puede leer de manera secuencial desde la posición actual. Este último método se utiliza en UNIX y Windows.

4.1.5 Atributos de archivos

Todo archivo tiene un nombre y sus datos. Además, todos los sistemas operativos asocian otra información con cada archivo; por ejemplo, la fecha y hora de la última modificación del archivo y su tamaño. A estos elementos adicionales les llamaremos **atributos** del archivo. Algunas personas los llaman **metadatos**. La lista de atributos varía considerablemente de un sistema a otro. La tabla de la figura 4-4 muestra algunas de las posibilidades, pero existen otras. Ningún sistema existente tiene todos, pero cada uno de ellos está presente en algún sistema.

Atributo	Significado
Protección	Quién puede acceso al archivo y en qué forma
Contraseña	Contraseña necesaria para acceder al archivo
Creador	ID de la persona que creó el archivo
Propietario	El propietario actual
Bandera de sólo lectura	0 para lectura/escritura; 1 para sólo lectura
Bandera oculto	0 para normal; 1 para que no aparezca en los listados
Bandera del sistema	0 para archivos normales; 1 para archivo del sistema
Bandera de archivo	0 si ha sido respaldado; 1 si necesita respaldarse
Bandera ASCII/binario	0 para archivo ASCII; 1 para archivo binario
Bandera de acceso aleatorio	0 para sólo acceso secuencial; 1 para acceso aleatorio
Bandera temporal	0 para normal; 1 para eliminar archivo al salir del proceso
Banderas de bloqueo	0 para desbloqueado; distinto de cero para bloqueado
Longitud de registro	Número de bytes en un registro
Posición de la llave	Desplazamiento de la llave dentro de cada registro
Longitud de la llave	Número de bytes en el campo llave
Hora de creación	Fecha y hora en que se creó el archivo
Hora del último acceso	Fecha y hora en que se accedió al archivo por última vez
Hora de la última modificación	Fecha y hora en que se modificó por última vez el archivo
Tamaño actual	Número de bytes en el archivo
Tamaño máximo	Número de bytes hasta donde puede crecer el archivo

Figura 4-4. Algunos posibles atributos de archivos.

Los primeros cuatro atributos se relacionan con la protección del archivo e indican quién puede acceder a él y quién no. Todos los tipos de esquemas son posibles, algunos de los cuales estudiaremos más adelante. En algunos sistemas, el usuario debe presentar una contraseña para acceder a un archivo, en cuyo caso la contraseña debe ser uno de los atributos.

Las banderas son bits o campos cortos que controlan o habilitan cierta propiedad específica. Por ejemplo, los archivos ocultos no aparecen en los listados de todos los archivos. La bandera de archivo es un bit que lleva el registro de si el archivo se ha respaldado recientemente. El programa

de respaldo lo desactiva y el sistema operativo lo activa cada vez que se modifica un archivo. De esta forma, el programa de respaldo puede indicar qué archivos necesitan respaldarse. La bandera temporal permite marcar un archivo para la eliminación automática cuando el proceso que lo creó termina.

Los campos longitud de registro, posición de llave y longitud de llave sólo están presentes en los archivos en cuyos registros se pueden realizar búsquedas mediante el uso de una llave. Ellos proporcionan la información requerida para buscar las llaves.

Los diversos tiempos llevan la cuenta de cuándo se creó el archivo, su acceso y su modificación más recientes. Éstos son útiles para una variedad de propósitos. Por ejemplo, un archivo de código fuente que se ha modificado después de la creación del archivo de código objeto correspondiente necesita volver a compilarse. Estos campos proporcionan la información necesaria.

El tamaño actual indica qué tan grande es el archivo en el presente. Algunos sistemas operativos de computadoras mainframe antiguas requieren que se especifique el tamaño máximo a la hora de crear el archivo, para poder permitir que el sistema operativo reserve la cantidad máxima de almacenamiento de antemano. Los sistemas operativos de estaciones de trabajo y computadoras personales son lo bastante inteligentes como para arreglárselas sin esta característica.

4.1.6 Operaciones de archivos

Los archivos existen para almacenar información y permitir que se recupere posteriormente. Distintos sistemas proveen diferentes operaciones para permitir el almacenamiento y la recuperación. A continuación se muestra un análisis de las llamadas al sistema más comunes relacionadas con los archivos.

1. `Create`. El archivo se crea sin datos. El propósito de la llamada es anunciar la llegada del archivo y establecer algunos de sus atributos.

2. `Delete`. Cuando el archivo ya no se necesita, se tiene que eliminar para liberar espacio en el disco. Siempre hay una llamada al sistema para este propósito.

3. `Open`. Antes de usar un archivo, un proceso debe abrirlo. El propósito de la llamada a `open` es permitir que el sistema lleve los atributos y la lista de direcciones de disco a memoria principal para tener un acceso rápido a estos datos en llamadas posteriores.

4. `Close`. Cuando terminan todos los accesos, los atributos y las direcciones de disco ya no son necesarias, por lo que el archivo se debe cerrar para liberar espacio en la tabla interna. Muchos sistemas fomentan esto al imponer un número máximo de archivos abiertos en los proceso. Un disco se escribe en bloques y al cerrar un archivo se obliga a escribir el último bloque del archivo, incluso aunque ese bloque no esté lleno todavía.

5. `Read`. Los datos se leen del archivo. Por lo general, los bytes provienen de la posición actual. El llamador debe especificar cuántos datos se necesitan y también debe proporcionar un búfer para colocarlos.

6. `Write`. Los datos se escriben en el archivo otra vez, por lo general en la posición actual. Si la posición actual es al final del archivo, aumenta su tamaño. Si la posición actual está en medio del archivo, los datos existentes se sobrescriben y se pierden para siempre.
7. `Append`. Esta llamada es una forma restringida de `write`. Sólo puede agregar datos al final del archivo. Los sistemas que proveen un conjunto mínimo de llamadas al sistema por lo general no tienen `append`; otros muchos sistemas proveen varias formas de realizar la misma acción y algunas veces ésos tienen `append`.
8. `Seek`. Para los archivos de acceso aleatorio, se necesita un método para especificar de dónde se van a tomar los datos. Una aproximación común es una llamada al sistema de nombre `seek`, la cual reposiciona el apuntador del archivo en una posición específica del archivo. Una vez que se completa esta llamada, se pueden leer o escribir datos en esa posición.
9. `Get attributes`. A menudo, los procesos necesitan leer los atributos de un archivo para realizar su trabajo. Por ejemplo, el programa *make* de UNIX se utiliza con frecuencia para administrar proyectos de desarrollo de software que consisten en muchos archivos fuente. Cuando se llama a *make*, este programa examina los tiempos de modificación de todos los archivos fuente y objeto, con los que calcula el mínimo número de compilaciones requeridas para tener todo actualizado. Para hacer su trabajo, debe analizar los atributos, a saber, los tiempos de modificación.
10. `Set attributes`. Algunos de los atributos puede establecerlos el usuario y se pueden modificar después de haber creado el archivo. Esta llamada al sistema hace eso posible. La información del modo de protección es un ejemplo obvio. La mayoría de las banderas también caen en esta categoría.
11. `Rename`. Con frecuencia ocurre que un usuario necesita cambiar el nombre de un archivo existente. Esta llamada al sistema lo hace posible. No siempre es estrictamente necesaria, debido a que el archivo por lo general se puede copiar en un nuevo archivo con el nuevo nombre, eliminando después el archivo anterior.

4.1.7 Un programa de ejemplo que utiliza llamadas al sistema de archivos

En esta sección examinaremos un programa simple de UNIX que copia un archivo de su archivo fuente a un archivo destino. Su listado se muestra en la figura 4-5. El programa tiene una mínima funcionalidad y un reporte de errores aún peor, pero nos da una idea razonable acerca de cómo funcionan algunas de las llamadas al sistema relacionadas con archivos.

Por ejemplo, el programa *copyfile* se puede llamar mediante la línea de comandos

```
copyfile abc xyz
```

para copiar el archivo *abc* a *xyz*. Si *xyz* ya existe, se sobrescribirá. En caso contrario, se creará.

```
/* Programa para copiar archivos. La verificación y el reporte de errores son mínimos. */

#include <sys/types.h>                  /* incluye los archivos de encabezado necesarios *
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* prototipo ANSI */

#define TAM_BUF 4096                    /* usa un tamaño de búfer de 4096 bytes */
#define MODO_SALIDA 0700                /* bits de protección para el archivo de salida */

int main(int argc, char *argv[])
{
        int ent_da, sal_da, leer_cuenta, escribir_cuenta:
        char bufer[TAM_BUF];

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

        /* Abre el archivo de entrada y crea el archivo de salida */
        ent_da = open(argv[1], O_RDONLY);       /* abre el archivo fuente */
        if (ent_da < 0) exit(2);                /* si no se puede abrir, termina */
        sal_da = creat(argv[2], MODO_SALIDA);   /* crea el archivo de destino */
        if (sal_da < 0) exit(3);                /* si no se puede crear, termina */

        /* Ciclo de copia */
        while (TRUE) {
                leer_cuenta = read(ent_da,bufer,TAM_BUF); /* lee un bloque de datos */
        if (lee_cuenta <= 0) break;             /* si llega al fin de archivo o hay un error, sale del ciclo */
                escribe_cuenta = write(sal_da,bufer,lee_cuenta); /* escribe los datos */
                if (escribe_cuenta <= 0) exit(4);       /* escribe_cuenta <= 0 es un error */
        }

        /* Cierra los archivos */
        close(ent_da);
        close(sal_da);
        if (lee_cuenta==0)                      /* no hubo error en la última lectura */
                exit(0);
        else
                exit(5);                        /* hubo error en la última lectura */
}
```

Figura 4-5. Un programa simple para copiar un archivo.

El programa debe llamarse con exactamente dos argumentos, ambos nombres de archivo válidos. El primero es el archivo fuente, el segundo es el de salida.

Las cuatro instrucciones *#include* iniciales hacen que se incluya una gran cantidad de definiciones y prototipos de funciones en el programa. Esto es necesario para hacer el programa conforme a los estándares internacionales relevantes, pero no nos ocuparemos más de ello. La siguiente

línea es un prototipo de función para *main*, algo requerido por ANSI C, pero que tampoco es importante para nuestros fines.

La primera instrucción *#define* es una definición de macro que define la cadena de caracteres *TAM_BUF* como una macro que se expande en el número 4096. El programa leerá y escribirá en trozos de 4096 bytes. Se considera una buena práctica de programación dar nombres a las constantes como ésta y utilizar los nombres en vez de las constantes. Esta convención no sólo facilita que los programas sean fáciles de leer, sino también su mantenimiento. La segunda instrucción *#define* determina quién puede acceder al archivo de salida.

El programa principal se llama *main* y tiene dos argumentos: *argc* y *argv*. El sistema operativo suministra estos argumentos cuando se hace una llamada al programa. El primero indica cuántas cadenas estaban presentes en la línea de comandos que invocó al programa, incluyendo su nombre. Debe ser 3. El segundo es un arreglo de apuntadores a los argumentos. En la llamada de ejemplo anterior, los elementos de este arreglo contienen apuntadores a los siguientes valores:

argv[0] = “copyfile”
argv[1] = “abc”
argv[2] = “xyz”

Es mediante este arreglo que el programa tiene acceso a sus argumentos.

Se declaran cinco variables. Las primeras dos, *ent_da* y *sal_da*, contienen los **descriptores de archivos**: pequeños enteros que se devuelven cuando se abre un archivo. Los siguientes dos, *leer_cuenta* y *escribir_cuenta*, son las cuentas de bytes que devuelven las llamadas al sistema `read` y `write`, respectivamente. La última variable, *bufer*, es el búfer que se utiliza para guardar los datos leídos y suministrar los datos que se van a escribir.

La primera instrucción verifica *argc* para ver si es 3. Si no, termina con el código de estado 1. Cualquier código de estado distinto de 0 indica que ocurrió un error. El código de estado es el único modo de reportar errores en este programa. Una versión de producción por lo general también imprime mensajes de error.

Después tratamos de abrir el archivo fuente y crear el archivo de destino. Si el archivo fuente se abre con éxito, el sistema asigna un pequeño entero a *ent_da*, para identificar el archivo. Las llamadas siguientes deben incluir este entero, de manera que el sistema sepa qué archivo desea. De manera similar, si el archivo de destino se crea satisfactoriamente, *sal_da* recibe un valor para identificarlo. El segundo argumento para *creat* establece el modo de protección. Si falla la apertura o la creación de los archivos, el descriptor de archivo correspondiente se establece en -1 y el programa termina con un código de error.

Ahora viene el ciclo de copia. Empieza tratando de leer 4 KB de datos al *bufer*. Para ello hace una llamada al procedimiento de biblioteca *read*, que en realidad invoca la llamada al sistema `read`. El primer parámetro identifica al archivo, el segundo proporciona el búfer y el tercero indica cuántos bytes se deben leer. El valor asignado a *leer_cuenta* indica el número de bytes leídos. Por lo general este número es 4096, excepto si en el archivo quedan menos bytes. Cuando se haya alcanzado el fin del archivo, será 0. Si alguna vez *leer_cuenta* es cero o negativo, el proceso de copia no puede continuar, ejecutándose la instrucción *break* para terminar el ciclo (que de otra manera no tendría fin).

La llamada a *write* envía el búfer al archivo de destino. El primer parámetro identifica al archivo, el segundo da el bufer y el tercero indica cuántos bytes se deben escribir, en forma análoga a *read*. Observe que la cuenta de bytes es el número de bytes leídos, no *TAM_BUF*. Este punto es importante, debido a que la última lectura no devolverá 4096 a menos que el archivo sea un múltiplo de 4 KB.

Cuando se haya procesado todo el archivo, la primera llamada más allá del final del archivo regresará 0 a *leer_cuenta*, lo cual hará que salga del ciclo. En este punto se cierran los dos archivos y el programa termina con un estado que indica la terminación 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 archivo es moderadamente similar al de la figura 4-5. En el capítulo 11 examinaremos las llamadas a Windows Vista.

4.2 DIRECTORIOS

Para llevar el registro de los archivos, los sistemas de archivos por lo general tienen **directorios** o **carpetas**, que en muchos sistemas son también archivos. En esta sección hablaremos sobre los directorios, su organización, sus propiedades y las operaciones que pueden realizarse con ellos.

4.2.1 Sistemas de directorios de un solo nivel

La forma más simple de un sistema de directorios es tener un directorio que contenga todos los archivos. Algunas veces se le llama **directorio raíz**, pero como es el único, el nombre no importa mucho. En las primeras computadoras personales, este sistema era común, en parte debido a que sólo había un usuario. Como dato interesante, la primera supercomputadora del mundo (CDC 6600) también tenía un solo directorio para todos los archivos, incluso cuando era utilizada por muchos usuarios a la vez. Esta decisión sin duda se hizo para mantener simple el diseño del software.

En la figura 4-6 se muestra un ejemplo de un sistema con un directorio. Aquí el directorio contiene cuatro archivos. Las ventajas de este esquema son su simpleza y la habilidad de localizar archivos con rapidez; después de todo, sólo hay un lugar en dónde buscar. A menudo se utiliza en dispositivos incrustados simples como teléfonos, cámaras digitales y algunos reproductores de música portátiles.

4.2.2 Sistemas de directorios jerárquicos

Tener un solo nivel es adecuado para aplicaciones dedicadas simples (e incluso se utilizaba en las primeras computadoras personales), pero para los usuarios modernos con miles de archivos, sería imposible encontrar algo si todos los archivos estuvieran en un solo directorio.

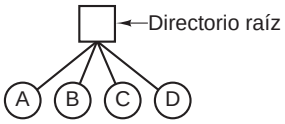

Figura 4-6. Un sistema de directorio de un solo nivel que contiene cuatro archivos.

En consecuencia, se necesita una forma de agrupar los archivos relacionados. Por ejemplo, un profesor podría tener una colección de archivos que en conjunto formen un libro que está escribiendo para un curso, una segunda colección de archivos que contienen programas enviados por los estudiantes para otro curso, un tercer grupo de archivos que contenga el código de un sistema de escritura de compiladores avanzado que está construyendo, un cuarto grupo de archivos que contienen proposiciones de becas, así como otros archivos para correo electrónico, minutas de reuniones, artículos que está escribiendo, juegos, etcétera.

Lo que se necesita es una jerarquía (es decir, un árbol de directorios). Con este esquema, puede haber tantos directorios como se necesite para agrupar los archivos en formas naturales. Además, si varios usuarios comparten un servidor de archivos común, como se da el caso en muchas redes de empresas, cada usuario puede tener un directorio raíz privado para su propia jerarquía. Este esquema se muestra en la figura 4-7. Aquí, cada uno de los directorios *A*, *B* y *C* contenidos en el directorio raíz pertenecen a un usuario distinto, dos de los cuales han creado subdirectorios para proyectos en los que están trabajando.

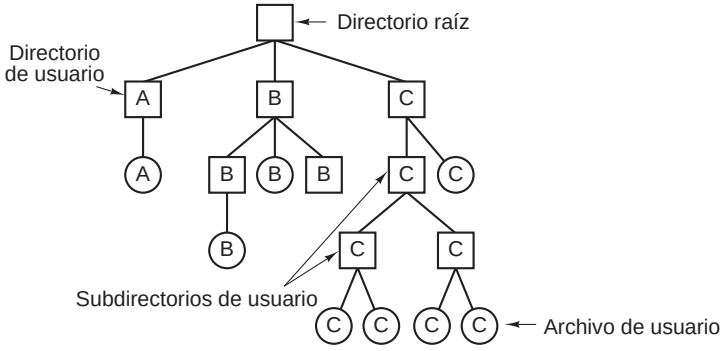

Figura 4-7. Un sistema de directorios jerárquico.

La capacidad de los usuarios para crear un número arbitrario de subdirectorios provee una poderosa herramienta de estructuración para que los usuarios organicen su trabajo. Por esta razón, casi todos los sistemas de archivos modernos se organizan de esta manera.

4.2.3 Nombres de rutas

Cuando el sistema de archivos está organizado como un árbol de directorios, se necesita cierta forma de especificar los nombres de los archivos. Por lo general se utilizan dos métodos distintos. En el primer método, cada archivo recibe un **nombre de ruta absoluto** que consiste en la ruta desde

el directorio raíz al archivo. Como ejemplo, la ruta */usr/ast/mailbox* significa que el directorio raíz contiene un subdirectorio llamado *usr*, que a su vez contiene un subdirectorio *ast*, el cual contiene el archivo *mailbox*. Los nombres de ruta absolutos siempre empiezan en el directorio raíz y son únicos. En UNIX, los componentes de la ruta van separados por /. En Windows el separador es \. En MULTICS era >. Así, el mismo nombre de ruta se escribiría de la siguiente manera en estos tres sistemas:

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

Sin importar cuál carácter se utilice, si el primer carácter del nombre de la ruta es el separador, entonces la ruta es absoluta.

El otro tipo de nombre es el **nombre de ruta relativa**. Éste se utiliza en conjunto con el concepto del **directorio de trabajo** (también llamado **directorio actual**). Un usuario puede designar un directorio como el directorio de trabajo actual, en cuyo caso todos los nombres de las rutas que no empiecen en el directorio raíz se toman en forma relativa al directorio de trabajo. Por ejemplo, si el directorio de trabajo actual es */usr/ast*, entonces el archivo cuya ruta absoluta sea */usr/ast/mailbox* se puede referenciar simplemente como *mailbox*. En otras palabras, el comando de UNIX

```
cp  /usr/ast/mailbox  /usr/ast/mailbox.bak
```

y el comando

```
cp mailbox  mailbox.bak
```

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

Algunos programas necesitan acceder a un archivo específico sin importar cuál sea el directorio de trabajo. En ese caso, siempre deben utilizar nombres de rutas absolutas. Por ejemplo, un corrector ortográfico podría necesitar leer */usr/lib/dictionary* para realizar su trabajo. Debe utilizar el nombre de la ruta absoluta completo en este caso, ya que no sabe cuál será el directorio de trabajo cuando sea llamado. El nombre de la ruta absoluta siempre funcionará, sin importar cuál sea el directorio de trabajo.

Desde luego que, si el corrector ortográfico necesita un gran número de archivos de */usr/lib*, un esquema alternativo es que emita una llamada al sistema para cambiar su directorio de trabajo a */usr/lib* y que después utilice *dictionary* como el primer parámetro para `open`. Al cambiar en forma explícita el directorio de trabajo, sabe con certeza dónde se encuentra en el árbol de directorios, para poder entonces usar rutas relativas.

Cada proceso tiene su propio directorio de trabajo, por lo que cuando éste cambia y después termina ningún otro proceso se ve afectado y no quedan rastros del cambio en el sistema de archivos. De esta forma, siempre es perfectamente seguro para un proceso cambiar su directorio de trabajo cada vez que sea conveniente. Por otro lado, si un *procedimiento de biblioteca* cambia el directorio de trabajo y no lo devuelve a su valor original cuando termina, el resto del programa tal

vez no funcione, ya que el supuesto directorio de trabajo que debería tener ahora podría ser inválido. Por esta razón, los procedimientos de biblioteca raras veces cambian el directorio de trabajo y cuando deben hacerlo siempre lo devuelven a su posición original antes de regresar.

La mayoría de los sistemas operativos que proporcionan un sistema de directorios jerárquico tienen dos entradas especiales en cada directorio: "." y "..", que por lo general se pronuncian "punto" y "puntopunto". Punto se refiere al directorio actual; puntopunto se refiere a su padre (excepto en el directorio raíz, donde se refiere a sí mismo). Para ver cómo se utilizan estas entradas especiales, considere el árbol de archivos de UNIX en la figura 4-8. Cierto proceso tiene a */usr/ast* como su directorio de trabajo. Puede utilizar .. para subir en el árbol. Por ejemplo, puede copiar el archivo */usr/lib/dictionary* a su propio directorio mediante el comando

```
cp ../lib/dictionary .
```

La primera ruta instruye al sistema que vaya hacia arriba (al directorio *usr*) y después baje al directorio *lib* para encontrar el archivo *dictionary*.

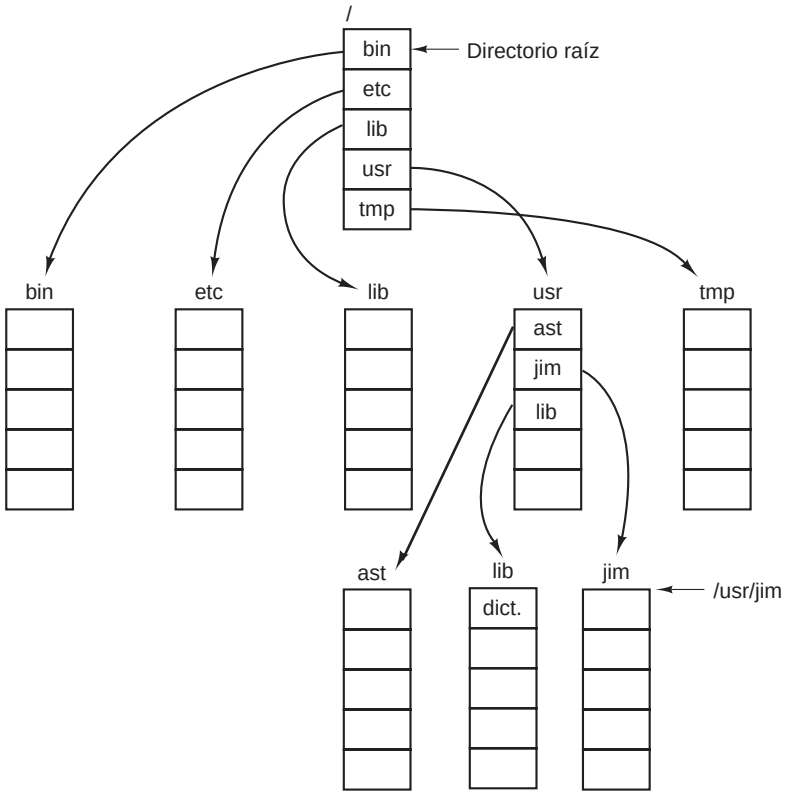

Figura 4-8. Un árbol de directorios de UNIX.

El segundo argumento (punto) nombra el directorio actual. Cuando el comando *cp* obtiene un nombre de directorio (incluyendo punto) como su último argumento, copia todos los archivos a ese

directorio. Desde luego que una forma más normal de realizar la copia sería utilizar el nombre de ruta absoluta completo del archivo de origen:

```
cp /usr/lib/dictionary .
```

Aquí el uso de punto ahorra al usuario la molestia de escribir *dictionary* una segunda vez. Sin embargo, también funciona escribir

```
cp /usr/lib/dictionary dictionary
```

al igual que

```
cp /usr/lib/dictionary  /usr/ast/dictionary
```

Todos estos comandos realizan exactamente lo mismo.

4.2.4 Operaciones de directorios

Las llamadas al sistema permitidas para administrar directorios exhiben más variación de un sistema a otro que las llamadas al sistema para los archivos. Para dar una impresión de lo que son y cómo funcionan, daremos un ejemplo (tomado de UNIX).

1. `Create`. Se crea un directorio. Está vacío, excepto por punto y puntopunto, que el sistema coloca ahí de manera automática (o en unos cuantos casos lo hace el programa *mkdir*).
2. `Delete`. Se elimina un directorio. Se puede eliminar sólo un directorio vacío. Un directorio que sólo contiene a punto y puntopunto se considera vacío, ya que por lo general éstos no se pueden eliminar.
3. `Opendir`. Los directorios se pueden leer. Por ejemplo, para listar todos los archivos en un directorio, un programa de listado abre el directorio para leer los nombres de todos los archivos que contiene. Antes de poder leer un directorio se debe abrir, en forma análoga al proceso de abrir y leer un archivo.
4. `Closedir`. Cuando se ha leído un directorio, se debe cerrar para liberar espacio en la tabla interna.
5. `Readdir`. Esta llamada devuelve la siguiente entrada en un directorio abierto. Antes era posible leer directorios utilizando la llamada al sistema `read` común, pero ese método tiene la desventaja de forzar al programador a conocer y tratar con la estructura interna de los directorios. En contraste, `readdir` siempre devuelve una entrada en formato estándar, sin importar cuál de las posibles estructuras de directorio se utilice.
6. `Rename`. En muchos aspectos, los directorios son sólo como archivos y se les puede cambiar le nombre de la misma forma que a los archivos.
7. `Link`. La vinculación (ligado) es una técnica que permite a un archivo aparecer en más de un directorio. Esta llamada al sistema especifica un archivo existente y el nombre de una ruta, creando un vínculo desde el archivo existente hasta el nombre especificado por la ruta.

De esta forma, el mismo archivo puede aparecer en varios directorios. A un vínculo de este tipo, que incrementa el contador en el nodo-i del archivo (para llevar la cuenta del número de entradas en el directorio que contienen el archivo), se le llama algunas veces **vínculo duro** (o **liga dura)**.

8. `Unlink`. Se elimina una entrada de directorio. Si el archivo que se va a desvincular sólo está presente en un directorio (el caso normal), se quita del sistema de archivos. Si está presente en varios directorios, se elimina sólo el nombre de ruta especificado. Los demás permanecen. En UNIX, la llamada al sistema para eliminar archivos (que vimos antes) es, de hecho, `unlink`.

La lista anterior contiene las llamadas más importantes, pero hay unas cuantas más; por ejemplo, para administrar la información de protección asociada con un directorio.

Una variante sobre la idea de vincular archivos es el **vínculo simbólico** (**liga simbólica)**. En vez de tener dos nombres que apunten a la misma estructura de datos interna que representa un archivo, se puede crear un nombre que apunte a un pequeño archivo que nombre a otro. Cuando se utiliza el primer archivo, por ejemplo, abierto, el sistema de archivos sigue la ruta y busca el nombre al final. Después empieza el proceso de búsqueda otra vez, utilizando el nuevo nombre. Los vínculos simbólicos tienen la ventaja de que pueden traspasar los límites de los discos e incluso nombrar archivos en computadoras remotas. Sin embargo, su implementación es poco menos eficiente que los vínculos duros.

4.3 IMPLEMENTACIÓN DE SISTEMAS DE ARCHIVOS

Ahora es el momento de cambiar del punto de vista que tiene el usuario acerca del sistema de archivos, al punto de vista del que lo implementa. Los usuarios se preocupan acerca de cómo nombrar los archivos, qué operaciones se permiten en ellos, cuál es la apariencia del árbol de directorios y cuestiones similares de la interfaz. Los implementadores están interesados en la forma en que se almacenan los archivos y directorios, cómo se administra el espacio en el disco y cómo hacer que todo funcione con eficiencia y confiabilidad. En las siguientes secciones examinaremos varias de estas áreas para ver cuáles son los problemas y las concesiones que se deben hacer.

4.3.1 Distribución del sistema de archivos

Los sistemas de archivos se almacenan en discos. La mayoría de los discos se pueden dividir en una o más particiones, con sistemas de archivos independientes en cada partición. El sector 0 del disco se conoce como el **MBR** (*Master Boot Record*; Registro maestro de arranque) y se utiliza para arrancar la computadora. El final del MBR contiene la tabla de particiones, la cual proporciona las direcciones de inicio y fin de cada partición. Una de las particiones en la tabla se marca como activa. Cuando se arranca la computadora, el BIOS lee y ejecuta el MBR. Lo primero que hace el programa MBR es localizar la partición activa, leer su primer bloque, conocido como **bloque de arranque**, y ejecutarlo. El programa en el bloque de arranque carga el sistema operativo contenido

en esa partición. Por cuestión de uniformidad, cada partición inicia con un bloque de arranque no contenga un sistema operativo que se pueda arrancar. Además, podría contener uno en el futuro.

Además de empezar con un bloque de arranque, la distribución de una partición de disco varía mucho de un sistema de archivos a otro. A menudo el sistema de archivos contendrá algunos de los elementos que se muestran en la figura 4-9. El primero es el **superbloque**. Contiene todos los parámetros clave acerca del sistema de archivos y se lee en la memoria cuando se arranca la computadora o se entra en contacto con el sistema de archivos por primera vez. La información típica en el superbloque incluye un número mágico para identificar el tipo del sistema de archivos, el número de bloques que contiene el sistema de archivos y otra información administrativa clave.

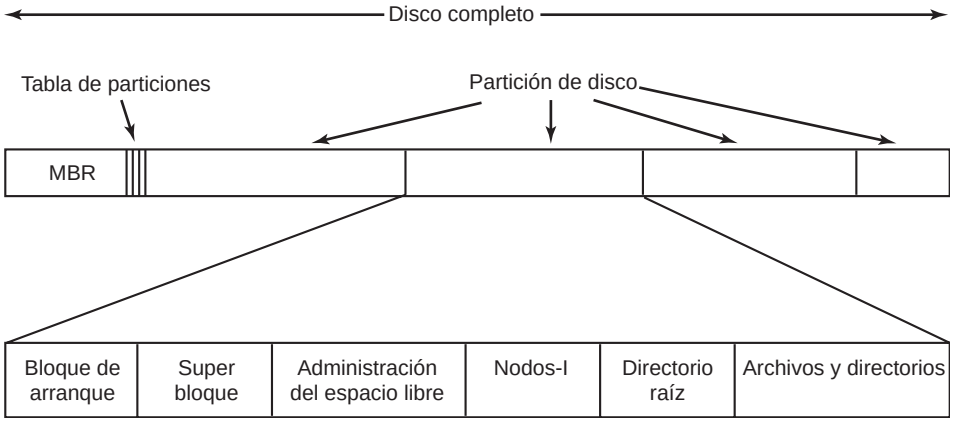

Figura 4-9. Una posible distribución del sistema de archivos.

A continuación podría venir información acerca de los bloques libres en el sistema de archivos, por ejemplo en la forma de un mapa de bits o una lista de apuntadores. Ésta podría ir seguida de los nodos-i, un arreglo de estructuras de datos, uno por archivo, que indica todo acerca del archivo. Después de eso podría venir el directorio raíz, que contiene la parte superior del árbol del sistema de archivos. Por último, el resto del disco contiene todos los otros directorios y archivos.

4.3.2 Implementación de archivos

Probablemente la cuestión más importante al implementar el almacenamiento de archivos sea mantener un registro acerca de qué bloques de disco van con cuál archivo. Se utilizan varios métodos en distintos sistemas operativos. En esta sección examinaremos unos cuantos de ellos.

Asignación contigua

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

En la figura 4-10(a) podemos ver un ejemplo de asignación de almacenamiento contigua. Aquí se muestran los primeros 40 bloques de disco, empezando con el bloque 0, a la izquierda. Al principio el disco estaba vacío, después se escribió un archivo *A* de cuatro bloques de longitud al disco, empezando desde el principio (bloque 0). Posteriormente se escribió un archivo de seis bloques llamado *B*, empezando justo después del archivo *A*.

Observe que cada archivo empieza al inicio de un nuevo bloque, por lo que si el archivo *A* fuera realmente de $3^1/_2$ bloques, se desperdiciaría algo de espacio al final del último bloque. En la figura se muestra un total de siete archivos, cada uno empezando en el bloque que va después del final del archivo anterior. Se utiliza sombreado sólo para facilitar la distinción de cada archivo. No tiene un significado real en términos de almacenamiento.

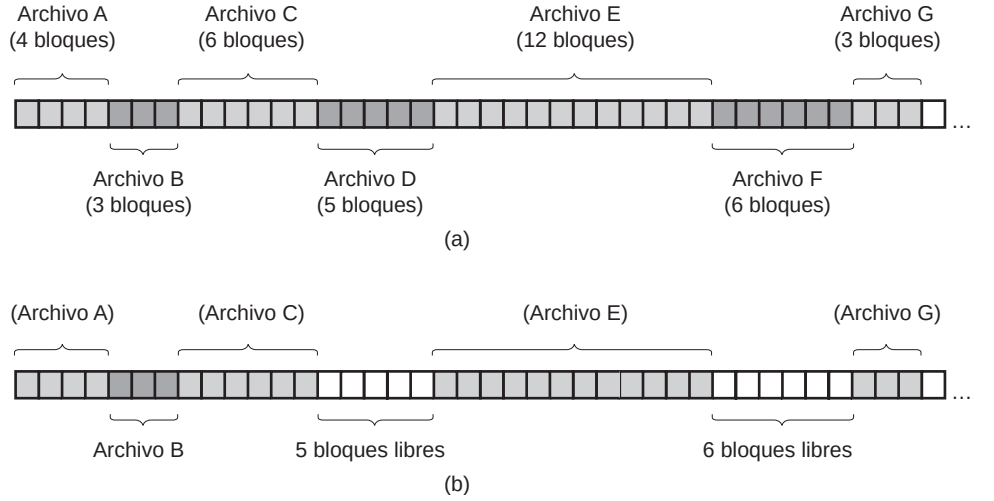

Figura 4-10. (a) Asignación contigua de espacio de disco para siete archivos. (b) El estado del disco después de haber removido los archivos *D* y *F*.

La asignación de espacio en disco contiguo tiene dos ventajas significativas. En primer lugar es simple de implementar, ya que llevar un registro de la ubicación de los bloques de un archivo se reduce a recordar dos números: la dirección de disco del primer bloque y el número de bloques en el archivo. Dado el número del primer bloque, se puede encontrar el número de cualquier otro bloque con una simple suma.

En segundo lugar, el rendimiento de lectura es excelente debido a que el archivo completo se puede leer del disco en una sola operación. Sólo se necesita una búsqueda (para el primer bloque). Después de eso, no son necesarias más búsquedas ni retrasos por rotación, por lo que los datos llegan con el ancho de banda completa del disco. Por ende, la asignación contigua es simple de implementar y tiene un alto rendimiento.

Por desgracia, la asignación contigua también tiene una desventaja ligeramente significante: con el transcurso del tiempo, los discos se fragmentan. Para ver cómo ocurre esto, examine la figura 4-10(b).

Aquí se han eliminado dos archivos, *D* y *F*. Cuando se quita un archivo, sus bloques se liberan naturalmente, dejando una serie de bloques libres en el disco. El disco no se compacta al momento para quitar el hueco, ya que eso implicaría tener que copiar todos los bloques que van después del hueco, que podrían ser millones. Como resultado, el disco al final consiste de archivos y huecos, como se ilustra en la figura.

Al principio esta fragmentación no es un problema, ya que cada nuevo archivo se puede escribir al final del disco, después del anterior. Sin embargo, en un momento dado el disco se llenará y será necesario compactarlo, lo cual es en extremo costoso o habrá que reutilizar el espacio libre de los huecos. Para reutilizar el espacio hay que mantener una lista de huecos, lo cual se puede hacer. Sin embargo, cuando se cree un nuevo archivo será necesario conocer su tamaño final para poder elegir un hueco del tamaño correcto y colocarlo.

Imagine las consecuencias de tal diseño. El usuario empieza un editor de texto o procesador de palabras para poder escribir un documento. Lo primero que pide el programa es cuántos bytes tendrá el documento final. Esta pregunta se debe responder o el programa no continuará. Si el número dado finalmente es demasiado pequeño, el programa tiene que terminar prematuramente debido a que el hueco de disco está lleno y no hay lugar para colocar el resto del archivo. Si el usuario trata de evitar este problema al proporcionar un número demasiado grande como tamaño final, por decir 100 MB, tal vez el editor no pueda encontrar un hueco tan grande y anuncie que el archivo no se puede crear. Desde luego que el usuario tiene la libertad de iniciar de nuevo el programa diciendo 50 MB esta vez y así en lo sucesivo hasta que se encuentre un hueco adecuado. Aún así, no es probable que este esquema haga que los usuarios estén felices.

Sin embargo, hay una situación en la que es factible la asignación contigua y de hecho, se utiliza ampliamente: en los CD-ROMs. Aquí todos los tamaños de los archivos se conocen de antemano y nunca cambiarán durante el uso subsiguiente del sistema de archivos del CD-ROM. Más adelante en este capítulo estudiaremos el sistema de archivos del CD-ROM más común.

La situación con los DVDs es un poco más complicada. En principio, una película de 90 minutos se podría decodificar como un solo archivo de una longitud aproximada de 4.5 GB, pero el sistema de archivos utilizado, conocido como **UDF** (*Universal Disk Format,* Formato de disco universal), utiliza un número de 30 bits para representar la longitud de un archivo, limitando el tamaño de los archivos a 1 GB. Como consecuencia, las películas en DVD se almacenan generalmente como tres o cuatro archivos de 1 GB, cada uno de los cuales es contiguo. Estas partes físicas del único archivo lógico (la película) se conocen como **fragmentos**.

Como mencionamos en el capítulo 1, la historia se repite a menudo en las ciencias computacionales, a medida que van surgiendo nuevas generaciones de tecnología. En realidad, la asignación contigua se utilizó en los sistemas de archivos en discos magnéticos hace años, debido a su simpleza y alto rendimiento (la amabilidad con el usuario no contaba mucho entonces). Después se descartó la idea debido a la molestia de tener que especificar el tamaño final del archivo al momento de su creación. Pero con la llegada de los CD-ROMs, los DVDs y otros medios ópticos en los que se puede escribir sólo una vez, los archivos contiguos son repentinamente una buena idea otra vez. Por lo tanto, es importante estudiar los antiguos sistemas y las ideas, que conceptualmente eran claras y simples, debido a que pueden aplicarse a los sistemas futuros en formas sorprendentes.

Asignación de lista enlazada (ligada)

El segundo método para almacenar archivos es mantener cada uno como una lista enlazada de bloques de disco, como se muestra en la figura 4-11. La primera palabra de cada bloque se utiliza como apuntador al siguiente. El resto del bloque es para los datos.

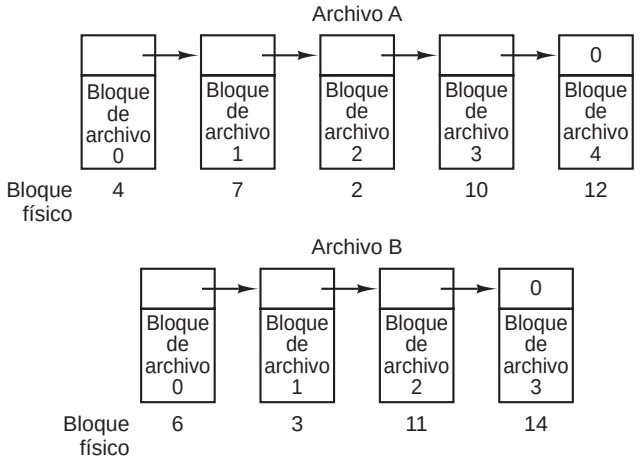

Figura 4-11. Almacenamiento de un archivo como una lista enlazada de bloques de disco.

A diferencia de la asignación contigua, en este método se puede utilizar cada bloque del disco. No se pierde espacio debido a la fragmentación del disco (excepto por la fragmentación interna en el último bloque). Además, para la entrada del directorio sólo le basta con almacenar la dirección de disco del primer bloque. El resto se puede encontrar a partir de ella.

Por otro lado, aunque la lectura secuencial un archivo es directa, el acceso aleatorio es en extremo lento. Para llegar al bloque *n*, el sistema operativo tiene que empezar desde el principio y leer los $n - 1$ bloques anteriores, uno a la vez. Es clarto que tantas lecturas serán demasiado lentas.

Además, la cantidad de almacenamiento de datos en un bloque ya no es una potencia de dos, debido a que el apuntador ocupa unos cuantos bytes. Aunque no es fatal, tener un tamaño peculiar es menos eficiente debido a que muchos programas leen y escriben en bloques, cuyo tamaño es una potencia de dos. Con los primeros bytes de cada bloque ocupados por un apuntador al siguiente bloque, leer el tamaño del bloque completo requiere adquirir y concatenar información de dos bloques de disco, lo cual genera un gasto adicional de procesamiento debido a la copia.

Asignación de lista enlazada utilizando una tabla en memoria

Ambas desventajas de la asignación de lista enlazada se pueden eliminar si tomamos la palabra del apuntador de cada bloque de disco y la colocamos en una tabla en memoria. La figura 4-12 mues-

tra cuál es la apariencia de la tabla para el ejemplo de la figura 4-11. En ambas figuras tenemos dos archivos. El archivo *A* utiliza los bloques de disco 4, 7, 2, 10 y 12, en ese orden y el archivo *B* utiliza los bloques de disco 6, 3, 11 y 14, en ese orden. Utilizando la tabla de la figura 4-12, podemos empezar con el bloque 4 y seguir toda la cadena hasta el final. Lo mismo se puede hacer empezando con el bloque 6. Ambas cadenas se terminan con un marcador especial (por ejemplo, −1) que no sea un número de bloque válido. Dicha tabla en memoria principal se conoce como **FAT** (*File Allocation Table,* Tabla de asignación de archivos).

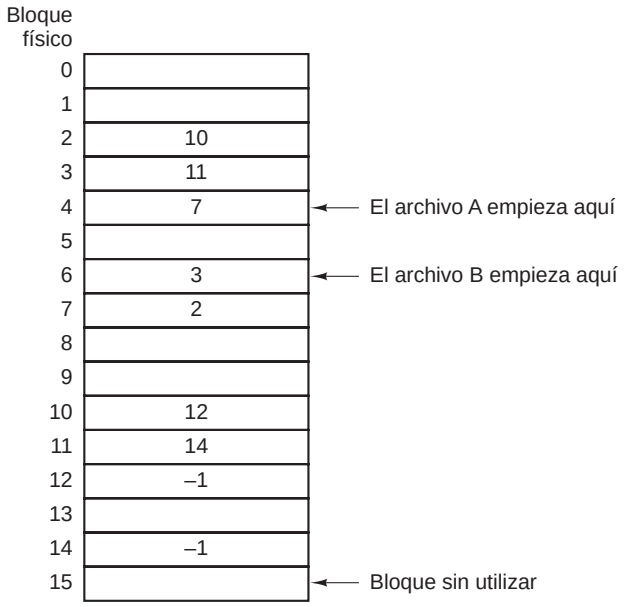

Figura 4-12. Asignación de lista enlazada que utiliza una tabla de asignación de archivos en la memoria principal.

Utilizando esta organización, el bloque completo está disponible para los datos. Además, el acceso aleatorio es mucho más sencillo. Aunque aún se debe seguir la cadena para encontrar un desplazamiento dado dentro del archivo, la cadena está completamente en memoria y se puede seguir sin necesidad de hacer referencias al disco. Al igual que el método anterior, la entrada de directorio necesita mantener sólo un entero (el número de bloque inicial) y aún así puede localizar todos los bloques, sin importar qué tan grande sea el archivo.

La principal desventaja de este método es que toda la tabla debe estar en memoria todo el tiempo para que funcione. Con un disco de 200 GB y un tamaño de bloque de 1 KB, la tabla necesita 200 millones de entradas, una para cada uno de los 200 millones de bloques de disco. Cada entrada debe tener un mínimo de 3 bytes. Para que la búsqueda sea rápida, deben tener 4 bytes. Así, la tabla ocupará 600 MB u 800 MB de memoria principal todo el tiempo, dependiendo de si el sistema está optimizado para espacio o tiempo. Esto no es muy práctico. Es claro que la idea de la FAT no se escala muy bien en los discos grandes.

Nodos-i

Nuestro último método para llevar un registro de qué bloques pertenecen a cuál archivo es asociar con cada archivo una estructura de datos conocida como **nodo-i** (**nodo-índice**), la cual lista los atributos y las direcciones de disco de los bloques del archivo. En la figura 4-13 se muestra un ejemplo simple. Dado el nodo-i, entonces es posible encontrar todos los bloques del archivo. La gran ventaja de este esquema, en comparación con los archivos vinculados que utilizan una tabla en memoria, es que el nodo-i necesita estar en memoria sólo cuando está abierto el archivo correspondiente. Si cada nodo-i ocupa *n* bytes y puede haber un máximo de *k* archivos abiertos a la vez, la memoria total ocupada por el arreglo que contiene los nodos-i para los archivos abiertos es de sólo *kn* bytes. Sólo hay que reservar este espacio por adelantado.

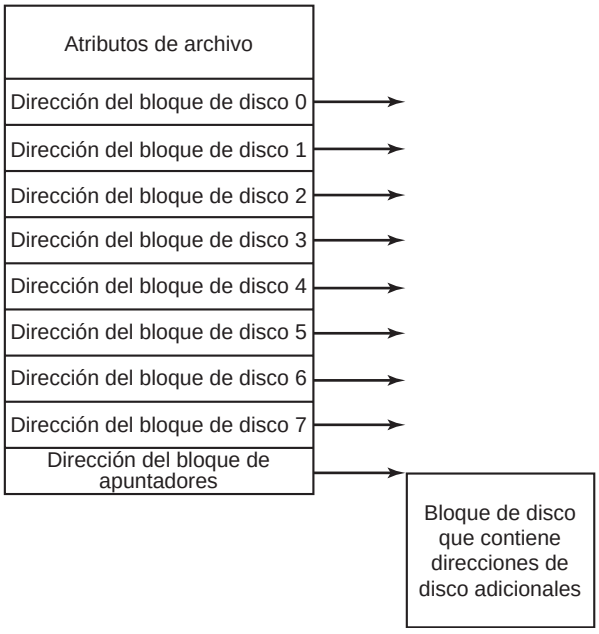

Figura 4-13. Un nodo-i de ejemplo.

Por lo general, este arreglo es mucho más pequeño que el espacio ocupado por la tabla de archivos descrita en la sección anterior. La razón es simple: la tabla para contener la lista enlazada de todos los bloques de disco es proporcional en tamaño al disco en sí. Si el disco tiene *n* bloques, la tabla necesita *n* entradas. A medida que aumenta el tamaño de los discos, esta tabla aumenta linealmente con ellos. En contraste, el esquema del nodo-i requiere un arreglo en memoria cuyo tamaño sea proporcional al número máximo de archivos que pueden estar abiertos a la vez. No importa si el disco es de 10 GB, de 100 GB ó de 1000 GB.

Un problema con los nodos-i es que si cada uno tiene espacio para un número fijo de direcciones de disco, ¿qué ocurre cuando un archivo crece más allá de este límite? Una solución es reser-

var 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 4-13. Algo aun más avanzado sería que dos o más de esos bloques contuvieran direcciones de disco o incluso bloques de disco apuntando a otros bloques de disco llenos de direcciones. Más adelante volveremos a ver los nodos-i, al estudiar UNIX.

4.3.3 Implementación de directorios

Antes de poder leer un archivo, éste debe abrirse. Cuando se abre un archivo, el sistema operativo utiliza el nombre de la ruta suministrado por el usuario para localizar la entrada de directorio. Esta entrada provee la información necesaria para encontrar los bloques de disco. Dependiendo del sistema, esta información puede ser la dirección de disco de todo el archivo (con asignación contigua), el número del primer bloque (ambos esquemas de lista enlazada) o el número del nodo-i. En todos los casos, la función principal del sistema de directorios es asociar el nombre ASCII del archivo a la información necesaria para localizar los datos.

Una cuestión muy relacionada es dónde deben almacenarse los atributos. Cada sistema de archivos mantiene atributos de archivo, como el propietario y la hora de creación de cada archivo, debiendo almacenarse en alguna parte. Una posibilidad obvia es almacenarlos directamente en la entrada de directorio. Muchos sistemas hacen eso. Esta opción se muestra en la figura 4-14(a). En este diseño simple, un directorio consiste en una lista de entradas de tamaño fijo, una por archivo, que contienen un nombre de archivo (de longitud fija), una estructura de los atributos del archivo y una o más direcciones de disco (hasta cierto máximo) que indique en dónde se encuentran los bloques de disco.

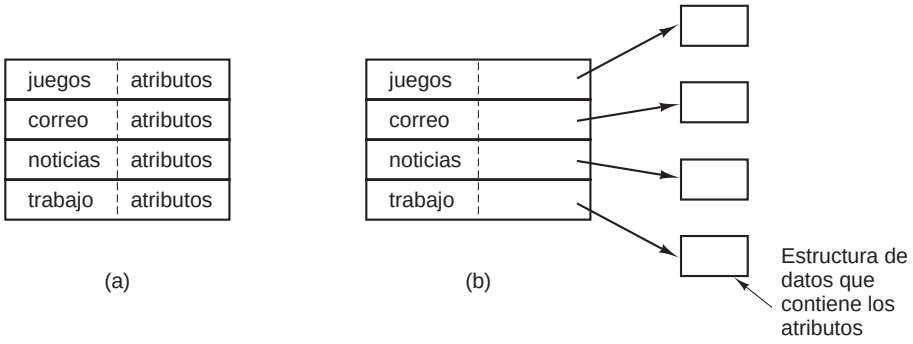

Figura 4-14. (a) Un directorio simple que contiene entradas de tamaño fijo, con las direcciones de disco y los atributos en la entrada de directorio. (b) Un directorio en el que cada entrada sólo hace referencia a un nodo-i.

Para los sistemas que utilizan nodos-i, existe otra posibilidad para almacenar los atributos en los nodos-i, en vez de hacerlo en las entradas de directorio. En ese caso, la entrada de directorio puede ser más corta: sólo un nombre de archivo y un número de nodo-i. Este método se ilustra en

la figura 4-14(b). Como veremos más adelante, este método tiene ciertas ventajas sobre el método de colocarlos en la entrada de directorio. Los dos esquemas que se muestran en la figura 4-14 corresponden a Windows y UNIX, respectivamente, como veremos más adelante.

Hasta ahora hemos hecho la suposición de que los archivos tienen nombres cortos con longitud fija. En MS-DOS, los archivos tienen un nombre base de 1 a 8 caracteres y una extensión opcional de 1 a 3 caracteres. En UNIX Version 7, los nombres de los archivos eran de 1 a 14 caracteres, incluyendo cualquier extensión. Sin embargo, casi todos los sistemas operativos modernos aceptan nombres de archivos más largos, con longitud variable. ¿Cómo se pueden implementar éstos?

El esquema más simple es establecer un límite en la longitud del nombre de archivo, que por lo general es de 255 caracteres y después utilizar uno de los diseños de la figura 4-14 con 255 caracteres reservados para cada nombre de archivo. Este esquema es simple, pero desperdicia mucho espacio de directorio, ya que pocos archivos tienen nombres tan largos. Por cuestiones de eficiencia, es deseable una estructura distinta.

Una alternativa es renunciar a la idea de que todas las entradas de directorio sean del mismo tamaño. Con este método, cada entrada de directorio contiene una porción fija, que por lo general empieza con la longitud de la entrada y después va seguida de datos con un formato fijo, que comúnmente incluyen el propietario, la hora de creación, información de protección y demás atributos. Este encabezado de longitud fija va seguido por el nombre del archivo, sin importar la longitud que tenga, como se muestra en la figura 4-15(a) en formato big-endian (por ejemplo, SPARC). En este ejemplo tenemos tres archivos, *sexto-proyecto*, *personaje* y *ave*. Cada nombre de archivo se termina con un carácter especial (por lo general 0), el cual se representa en la figura mediante un cuadro con una cruz. Para permitir que cada entrada de directorio empiece en un límite de palabra, cada nombre de archivo se rellena a un número entero de palabras, que se muestran como cuadros sombreados en la figura.

Una desventaja de este método es que cuando se elimina un archivo, en su lugar queda un hueco de tamaño variable dentro del directorio, dentro del cual el siguiente archivo a introducir puede que no quepa. Este problema es el mismo que vimos con los archivos de disco contiguos, sólo que ahora es factible compactar el directorio ya que se encuentra por completo en la memoria. Otro problema es que una sola entrada en el directorio puede abarcar varias páginas, por lo que puede ocurrir un fallo de páginas al leer el nombre de un archivo.

Otra manera de manejar los nombres de longitud variable es hacer que las mismas entradas de directorio sean de longitud fija y mantener los nombres de los archivos juntos en un heap al final del directorio, como se muestra en la figura 4-15(b). Este método tiene la ventaja de que cuando se remueva una entrada, el siguiente archivo a introducir siempre cabrá ahí. Desde luego que el heap se debe administrar y todavía pueden ocurrir fallos de página al procesar los nombres de los archivos. Una pequeña ganancia aquí es que ya no hay una verdadera necesidad de que los nombres de archivos empiecen en límites de palabras, por lo que no se necesitan caracteres de relleno después de los nombres de archivos en la figura 4-15(b), como se hizo en la figura 4-15(a).

En todos los diseños mostrados hasta ahora se realizan búsquedas lineales en los directorios de principio a fin cuando hay que buscar el nombre de un archivo. Para los directorios en extremo largos, la búsqueda lineal puede ser lenta. Una manera de acelerar la búsqueda es utilizar una tabla de hash en cada directorio. Llamemos n al tamaño de la tabla. Para introducir un nombre de archivo, se codifica en hash con un valor entre 0 y $n - 1$, por ejemplo, dividiéndolo entre n y tomando el

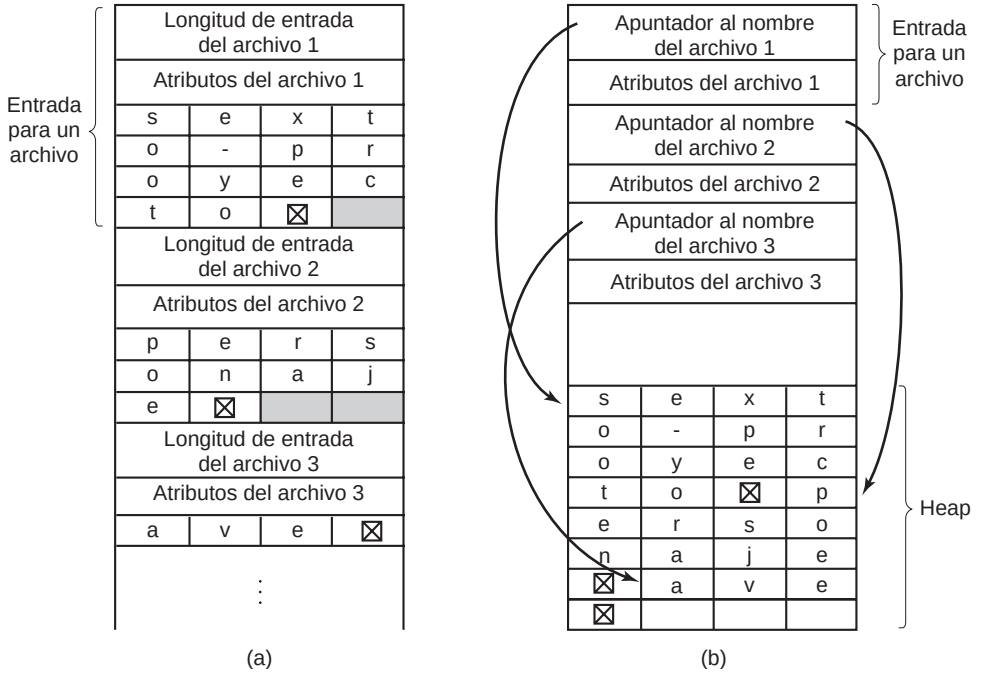

Figura 4-15. Dos maneras de manejar nombres de archivos largos en un directorio. (a) En línea. (b) En un heap.

residuo. De manera alternativa, las palabras que componen el nombre del archivo se pueden sumar y esta cantidad se divide entre *n*, o algo similar.

De cualquier forma, se inspecciona la entrada de la tabla correspondiente al código de hash. Si no está en uso, un apuntador se coloca ahí a la entrada del archivo. Las entradas de archivos siguen la tabla de hash. Si esa ranura ya está en uso, se construye una lista enlazada, que encabeza la entrada de la tabla y que encadena todas las entradas con el mismo valor de hash.

Para buscar un archivo se sigue el mismo procedimiento. El nombre de archivo se codifica en hash para seleccionar una entrada en la tabla hash. Todas las entradas en la cadena que encabeza esa ranura se verifican para ver si el nombre de archivo está presente. Si el nombre no se encuentra en la cadena, el archivo no está presente en el directorio.

El uso de una tabla de hash tiene la ventaja de que la búsqueda es mucho más rápida, pero la desventaja de una administración más compleja. En realidad es sólo un candidato serio en los sistemas donde se espera que los directorios rutinariamente contengan cientos o miles de archivos.

Una manera distinta de acelerar la búsqueda en directorios extensos es colocar en caché los resultados de las búsquedas. Antes de iniciar una búsqueda, primero se realiza una verificación para ver si el nombre del archivo está en la caché. De ser así, se puede localizar de inmediato. Desde luego, el uso de la caché sólo funciona si un número relativamente pequeño de archivos abarcan la mayoría de las búsquedas.

4.3.4 Archivos compartidos

Cuando hay varios usuarios trabajando en conjunto en un proyecto, a menudo necesitan compartir archivos. Como resultado, con frecuencia es conveniente que aparezca un archivo compartido en forma simultánea en distintos directorios que pertenezcan a distintos usuarios. La figura 4-16 muestra el sistema de archivos de la figura 4-7 de nuevo, sólo que con uno de los archivos de *C* ahora presentes en uno de los directorios de *B* también. La conexión entre el directorio de *B* y el archivo compartido se conoce como un **vínculo** (liga). El sistema de archivos en sí es ahora un **Gráfico acíclico dirigido** (*Directed Acyclic Graph,* **DAG**) en vez de un árbol.

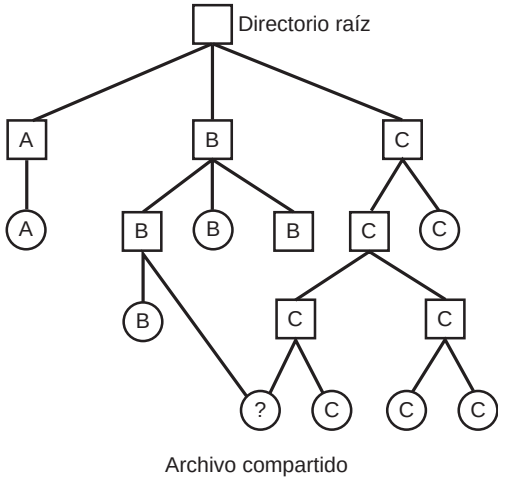

Figura 4-16. Sistema de archivos que contiene un archivo compartido.

Compartir archivos es conveniente, pero también introduce ciertos problemas. Para empezar, si los directorios en realidad contienen direcciones de disco, entonces habrá que realizar una copia de las direcciones de disco en el directorio de *B* cuando se ligue el archivo. Si *B* o *C* agregan posteriormente al archivo, los nuevos bloques se listarán sólo en el directorio del usuario que agregó los datos. Los cambios no estarán visibles para el otro usuario, con lo cual fracasa el propósito de la compartición.

Este problema se puede resolver de dos formas. En la primera solución, los bloques de disco no se listan en los directorios, sino en una pequeña estructura de datos asociada con el archivo en sí. Entonces, los directorios apuntarían sólo a la pequeña estructura de datos. Éste es el esquema que se utiliza en UNIX (donde la pequeña estructura de datos es el nodo-i).

En la segunda solución, *B* se vincula a uno de los archivos de *C* haciendo que el sistema cree un archivo, de tipo LINK e introduciendo ese archivo en el directorio de *B*. El nuevo archivo contiene sólo el nombre de la ruta del archivo al cual está vinculado. Cuando *B* lee del archivo vinculado, el sistema operativo ve que el archivo del que se están leyendo datos es de tipo LINK, busca el nombre del archivo y lee el archivo. A este esquema se le conoce como **vínculo simbólico (liga simbólica)**, para contrastarlo con el tradicional vínculo (duro).

Cada uno de estos métodos tiene sus desventajas. En el primer método, al momento en que *B* se vincula al archivo compartido, el nodo-i registra al propietario del archivo como *C*. Al crear un vínculo no se cambia la propiedad (vea la figura 4-17), sino incrementa la cuenta de vínculos en el nodo-i, por lo que el sistema sabe cuántas entradas de directorio actualmente apuntan al archivo.

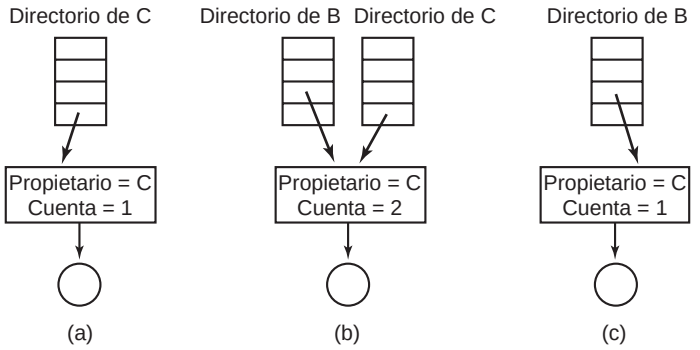

Figura 4-17. (a) Situación previa a la vinculación. (b) Después de crear el vínculo. (c) Después de que el propietario original elimina el archivo.

Si *C* posteriormente trata de eliminar el archivo, el sistema se enfrenta a un problema. Si elimina el archivo y limpia el nodo-i, *B* tendrá una entrada de directorio que apunte a un nodo-i inválido. Si el nodo-i se reasigna más tarde a otro archivo, el vínculo de *B* apuntará al archivo incorrecto. El sistema puede ver de la cuenta en el nodo-i que el archivo sigue todavía en uso, pero no hay una manera sencilla de que encuentre todas las entradas de directorio para el archivo, para que pueda borrarlas. Los apuntadores a los directorios no se pueden almacenar en el nodo-i, debido a que puede haber un número ilimitado de directorios.

Lo único por hacer es eliminar la entrada de directorio de *C*, pero dejar el nodo-i intacto, con la cuenta establecida en 1, como se muestra en la figura 4-17(c). Ahora tenemos una situación en la que *B* es el único usuario que tiene una entrada de directorio para un archivo que pertenece a *C*. Si el sistema realiza la contabilidad o tiene cuotas, seguirá cobrando a *C* por el archivo hasta que *B* decida eliminarlo, si acaso lo hace, momento en el cual la cuenta será 0 y el archivo se eliminará.

Con los vínculos simbólicos no se produce este problema, debido a que sólo el verdadero propietario tiene un apuntador al nodo-i. Los usuarios que han creado vínculos al archivo sólo tienen nombres de ruta, no apuntadoras a nodos-i. Cuando el *propietario* elimina el archivo, éste se destruye. Los intentos posteriores por utilizar el archivo vía un vínculo simbólico fallarán cuando el sistema no pueda localizar el archivo. Al eliminar un vínculo simbólico, el archivo no se ve afectado.

El problema con los vínculos simbólicos es el gasto adicional de procesamiento requerido. Se debe leer el archivo que contiene la ruta, después ésta se debe analizar sintácticamente y seguir, componente por componente, hasta llegar al nodo-i. Toda esta actividad puede requerir una cantidad considerable de accesos adicionales al disco. Además, se necesita un nodo-i adicional para cada vínculo simbólico, al igual que un bloque de disco adicional para almacenar la ruta, aunque si el nombre de la ruta es corto, el sistema podría almacenarlo en el mismo nodo-i, como un tipo de

optimización. Los vínculos simbólicos tienen la ventaja de que se pueden utilizar para vincular archivos en máquinas dondequiera en todo el mundo, con sólo proporcionar la dirección de red de la máquina donde reside el archivo, además de su ruta en esa máquina.

También hay otro problema que introducen los vínculos, ya sean simbólicos o de cualquier otro tipo. Cuando se permiten los vínculos, los archivos pueden tener dos o más rutas. Los programas, que empiezan en un directorio dado, encontrando todos los archivos en ese directorio y en sus subdirectorios, localizarán un archivo vinculado varias veces. Por ejemplo, un programa que vacía todos los archivos en un directorio y sus subdirectorios en una cinta, podría realizar varias copias de un archivo vinculado. Lo que es más, si la cinta se lee en otra máquina, a menos que el programa de vaciado sea inteligente, el archivo vinculado se copiará dos veces en el disco, en vez de ser vinculado.

4.3.5 Sistemas de archivos estructurados por registro

Los cambios en la tecnología están ejerciendo presión sobre los sistemas de archivos actuales. En especial, las CPUs se hacen más veloces, los discos se están haciendo más grandes y baratos (pero no mucho más rápidos) y las memorias están aumentando su tamaño en forma exponencial. El único parámetro que no está teniendo grandes avances es el tiempo de búsqueda de disco. La combinación de estos factores indica que un cuello de botella de funcionamiento está creciendo en muchos sistemas de archivos. Las investigaciones realizadas en Berkeley trataron de aliviar este problema al diseñar un tipo de sistema de archivos completamente nuevo, llamado **LFS** (*Log-structured File System*, Sistema de archivos estructurado por registro). En esta sección describiremos brevemente cómo funciona LFS. Consulte un tratamiento más completo en Rosenblum y Ousterhout, 1991.

La idea que impulsó el diseño del LFS es que, a medida que las CPUs se vuelven más rápidas y las memorias RAM se hacen más grandes, las cachés de disco también se incrementan con rapidez. En consecuencia, ahora es posible satisfacer una fracción muy considerable de todas las peticiones de lectura directamente del caché del sistema de archivos, sin necesitar accesos al disco. De esta observación podemos deducir que en el futuro la mayoría de los accesos al disco serán escrituras, por lo que el mecanismo de lectura adelantada utilizado en algunos sistemas de archivos para obtener bloques antes de necesitarlos ya no gana mucho en rendimiento.

Para empeorar las cosas, en la mayoría de los sistemas de archivos las escrituras se realizan en trozos muy pequeños. Las pequeñas escrituras son altamente ineficientes, ya que una escritura en disco de 50 µseg a menudo va precedida de una búsqueda de 10 mseg y de un retraso rotacional de 4 mseg. Con estos parámetros, la eficiencia en disco disminuye a una fracción de 1%.

Para ver de dónde provienen todas las pequeñas escrituras, considere la creación de un nuevo archivo en un sistema UNIX. Para escribir este archivo, se deben escribir el nodo-i para el directorio, el bloque de directorio, el nodo-i para el archivo y el mismo archivo. Aunque estas escrituras se pueden retrasar, hacerlo expone al sistema de archivos a problemas de consistencia graves si ocurre una falla antes de realizar las escrituras. Por esta razón, las escrituras de nodos-i generalmente se realizan de inmediato.

A partir de este razonamiento, los diseñadores del LFS decidieron reimplementar el sistema de archivos de UNIX a fin de que alcance el ancho de banda completo del disco, incluso ante una carga de trabajo que consistiera en gran parte de pequeñas escrituras al azar. La idea básica es estructurar

todo el disco como un registro. De manera periódica, y cuando haya una necesidad especial para ello, todas las escrituras pendientes que se colocaron en un búfer en memoria se recolectan en un solo segmento y se escriben en el disco como un solo segmento continuo al final del registro. Por lo tanto, un solo segmento puede contener nodos-i, bloques de directorio y bloques de datos, todos mezclados entre sí. Al inicio de cada segmento hay un resumen de segmento, que indica lo que se puede encontrar en el segmento. Si se puede hacer que el segmento promedio tenga un tamaño aproximado a 1 MB, entonces se puede utilizar casi todo el ancho de banda del disco.

En este diseño, los nodos-i aún existen y tienen la misma estructura que en UNIX, pero ahora están esparcidos por todo el registro, en vez de estar en una posición fija en el disco. Sin embargo, cuando se localiza un nodo-i, la localización de los bloques se realiza de la manera usual. Desde luego que ahora es mucho más difícil buscar un nodo-i, ya que su dirección simplemente no se puede calcular a partir de su número-i, como en UNIX. Para que sea posible encontrar nodos-i, se mantiene un mapa de nodos-i, indexados por número-i. La entrada *i* en este mapa apunta al nodo-i *i* en el disco. El mapa se mantiene en el disco, pero también se coloca en la caché, de manera que las partes más utilizadas estén en memoria la mayor parte del tiempo.

Para resumir lo que hemos dicho hasta ahora, al principio todas las escrituras se colocan en un búfer en memoria y periódicamente todas las escrituras en búfer se escriben en el disco en un solo segmento, al final del registro. Para abrir un archivo, ahora se utiliza el mapa para localizar el nodo-i para ese archivo. Una vez localizado el nodo-i, se pueden encontrar las direcciones de los bloques a partir de él. Todos los bloques estarán en segmentos, en alguna parte del registro.

Si los discos fueran infinitamente extensos, la descripción anterior sería todo. Sin embargo, los discos reales son finitos, por lo que en un momento dado el registro ocupará todo el disco y en ese momento no se podrán escribir nuevos segmentos en el registro. Por fortuna muchos segmentos existentes pueden tener bloques que ya no sean necesarios; por ejemplo, si se sobrescribe un archivo, su nodo-i apuntará ahora a los nuevos bloques, pero los anteriores seguirán ocupando espacio en los segmentos escritos anteriormente.

Para lidiar con este problema, el LFS tiene un hilo **limpiador** que pasa su tiempo explorando el registro circularmente para compactarlo. Empieza leyendo el resumen del primer segmento en el registro para ver qué nodos-i y archivos están ahí. Después verifica el mapa de nodos-i actual para ver si los nodos-i están actualizados y si los bloques de archivos están todavía en uso. De no ser así, se descarta esa información. Los nodos-i y los bloques que aún están en uso van a la memoria para escribirse en el siguiente segmento. El segmento original se marca entonces como libre, de manera que el registro pueda utilizarlo para nuevos datos. De esta manera, el limpiador se desplaza por el registro, removiendo los segmentos antiguos de la parte final y colocando cualquier información viva en la memoria para volver a escribirla en el siguiente segmento. En consecuencia, el disco es un gran búfer circular, donde el hilo escritor agrega nuevos segmentos al frente y el hilo limpiador remueve los segmentos viejos de la parte final.

Llevar la contabilidad aquí no es trivial, ya que cuando un bloque de archivo se escribe de vuelta en un nuevo segmento, se debe localizar el nodo-i del archivo (en alguna parte del registro), actualizarlo y colocarlo en memoria para escribirse en el siguiente segmento. El mapa de nodos-i debe entonces actualizarse para apuntar a la nueva copia. Sin embargo, es posible realizar la administración y los resultados de rendimiento muestran que toda esta complejidad vale la pena. Las mediciones proporcionadas en los artículos antes citados muestran que LFS supera a UNIX en rendimiento

por una orden de magnitud en las escrituras pequeñas, mientras que tiene un rendimiento tan bueno o mejor que UNIX para las lecturas o las escrituras extensas.

4.3.6 Sistemas de archivos por bitácora

Aunque los sistemas de archivos estructurados por registro son una idea interesante, no se utilizan ampliamente, debido en parte a su alta incompatibilidad con los sistemas de archivos existentes. Sin embargo, una de las ideas inherentes en ellos, la robustez frente a las fallas, se puede aplicar con facilidad a sistemas de archivos más convencionales. La idea básica aquí es mantener un registro de lo que va a realizar el sistema de archivos antes de hacerlo, por lo que si el sistema falla antes de poder realizar su trabajo planeado, al momento de re-arrancar el sistema puede buscar en el registro para ver lo que estaba ocurriendo al momento de la falla y terminar el trabajo. Dichos sistemas de archivos, conocidos como **sistemas de archivos por bitácora** (*Journaling files system*, JFS), se encuentran en uso actualmente. El sistema de archivos NTFS de Microsoft, así como los sistemas ext3 y ReiserFS de Linux son todos por bitácora. A continuación daremos una breve introducción a este tema.

Para ver la naturaleza del problema, considere una operación simple que ocurre todo el tiempo: remover un archivo. Esta operación (en UNIX) requiere tres pasos:

1. Quitar el archivo de su directorio.
2. Liberar el nodo-i y pasarlo a la reserva de nodos-i libres.
3. Devolver todos los bloques de disco a la reserva de bloques de disco libres.

En Windows se requieren pasos similares. En la ausencia de fallas del sistema, el orden en el que se realizan estos pasos no importa; en la presencia de fallas, sí. Suponga que se completa el primer paso y después el sistema falla. El nodo-i y los bloques de archivo no estarán accesibles desde ningún archivo, pero tampoco estarán disponibles para ser reasignados; sólo se encuentran en alguna parte del limbo, disminuyendo los recursos disponibles. Si la falla ocurre después del siguiente paso, sólo se pierden los bloques.

Si el orden de las operaciones se cambia y el nodo-i se libera primero, entonces después de rearrancar, el nodo-i se puede reasignar pero la entrada de directorio anterior seguirá apuntando a él y por ende al archivo incorrecto. Si los bloques se liberan primero, entonces una falla antes de limpiar el nodo-i indicará que una entrada de directorio válida apunta a un nodo-i que lista los bloques que ahora se encuentran en la reserva de almacenamiento libre y que probablemente se reutilicen en breve, produciendo dos o más archivos que compartan al azar los mismos bloques. Ninguno de estos resultados es bueno.

Lo que hace el sistema de archivos por bitácora es escribir primero una entrada de registro que liste las tres acciones a completar. Después la entrada de registro se escribe en el disco (y como buena medida, posiblemente se lea otra vez del disco para verificar su integridad). Sólo hasta que se ha escrito la entrada de registro es cuando empiezan las diversas operaciones. Una vez que las operaciones se completan con éxito, se borra la entrada de registro. Si ahora el sistema falla, al momen-

to de recuperarse el sistema de archivos puede verificar el registro para ver si había operaciones pendientes. De ser así, todas ellas se pueden volver a ejecutar (múltiples veces, en caso de fallas repetidas) hasta que el archivo se elimine en forma correcta.

Para que funcione el sistema por bitácora, las operaciones registradas deben ser **idempotentes**, lo cual significa que pueden repetirse todas las veces que sea necesario sin peligro. Las operaciones como "Actualizar el mapa de bits para marcar el nodo-i *k* o el bloque *n* como libre" se pueden repetir hasta que las todas las operaciones se completen sin peligro. De manera similar, las operaciones de buscar en un directorio y eliminar una entrada llamada *foobar* también son idempotentes. Por otro lado, la operación de agregar los bloques recién liberados del nodo-i *K* al final de la lista libre no es idempotente, debido a que éstos tal vez ya se encuentren ahí. La operación más costosa "Buscar en la lista de bloques libres y agregarle el bloque *n* si no está ya presente", también es idempotente. Los sistemas de archivos por bitácora tienen que organizar sus estructuras de datos y operaciones que pueden registrarse, de manera que todas ellas sean idempotentes. Bajo estas condiciones, la recuperación de errores puede ser rápida y segura.

Para una mayor confiabilidad, un sistema de archivos puede introducir el concepto de las bases de datos conocido como **transacción atómica.** Cuando se utiliza este concepto, varias acciones se pueden agrupar mediante las operaciones `begin transaction` y `end transaction`. Así, el sistema de archivos sabe que debe completar todas las operaciones agrupadas o ninguna de ellas, pero ninguna otra combinación.

NTFS tiene un sistema por bitácora extenso y su estructura rara vez se corrompe debido a las fallas en el sistema. Ha estado en desarrollo desde la primera vez que se liberó con Windows NT en 1993. El primer sistema de archivos por bitácora de Linux fue ReiserFS, pero su popularidad se vio impedida por el hecho de que era incompatible con el sistema de archivos ext2, que entonces era el estándar. En contraste, ext3, que es un proyecto menos ambicioso que ReiserFS, también se hace por bitácora, al tiempo que mantiene la compatibilidad con el sistema ext2 anterior.

4.3.7 Sistemas de archivos virtuales

Hay muchos sistemas de archivos distintos en uso, a menudo en la misma computadora, incluso para el mismo sistema operativo. Un sistema Windows puede tener un sistema de archivos NTFS principal, pero también una unidad o partición, FAT-32 o FAT-16, heredada que contenga datos antiguos, pero aún necesarios y de vez en cuando también se puede requerir un CD-ROM o DVD (cada uno con su propio sistema de archivos). Para manejar estos sistemas de archivos dispares, Windows identifica a cada uno con una letra de unidad distinta, como en *C:*, *D:*, etcétera. Cuando un proceso abre un archivo, la letra de la unidad está presente en forma explícita o, implícita de manera que Windows sepa a cuál sistema de archivos debe pasar la solicitud. No hay un intento por integrar sistemas de archivos heterogéneos en un todo unificado.

En contraste, todos los sistemas UNIX modernos tratan con mucha seriedad de integrar varios sistemas de archivos en una sola estructura. Un sistema Linux podría tener a ext2 como el sistema de archivos raíz, con una partición ext3 montada en */usr* y un segundo disco duro con un sistema de archivos ReiserFS montado en */home*, así como un CD-ROM ISO 9660 montado temporalmente en

/mnt. Desde el punto de vista del usuario, hay una sola jerarquía de sistemas de archivos. Lo que sucede al abarcar múltiples sistemas de archivos (incompatibles) no es visible para los usuarios o procesos.

Sin embargo, la presencia de múltiples sistemas de archivos está muy visible definitivamente para la implementación y desde el trabajo encabezado por Sun Microsystems (Kleiman, 1986), la mayoría de los sistemas UNIX han utilizado el concepto de **VFS** (*virtual file system*, Sistema de archivos virtual) para tratar de integrar múltiples sistemas de archivos en una estructura ordenada. La idea clave es abstraer la parte del sistema de archivos que es común para todos los sistemas de archivos y poner ese código en una capa separada que llame a los sistemas de archivos concretos subyacentes para administrar los datos. La estructura general se ilustra en la figura 4-18. El siguiente análisis no es específico para Linux o FreeBSD, ni cualquier otra versión de UNIX, sino que proporciona un panorama general acerca de cómo funcionan los sistemas de archivos virtuales en los sistemas UNIX.

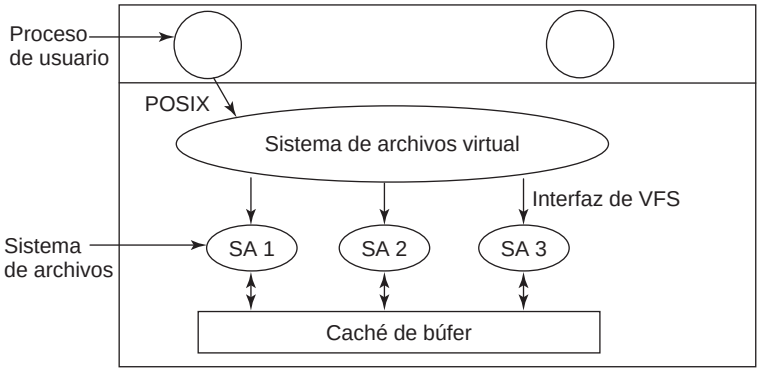

Figura 4-18. Posición del sistema de archivos virtual.

Todas las llamadas al sistema relacionadas con archivos se dirigen al sistema de archivos virtual para su procesamiento inicial. Estas llamadas, que provienen de procesos de usuario, son las llamadas de POSIX estándar tales como open, read, write, lseek, etcétera. Por ende, el VFS tiene una interfaz "superior" para los procesos de usuario y es la muy conocida interfaz de POSIX.

El VFS también tiene una interfaz "inferior" para los sistemas de archivos concretos, etiquetada como **Interfaz de VFS** en la figura 4-18. Esta interfaz consiste de varias docenas de llamadas a funciones que el VFS puede hacer a cada sistema de archivos para realizar el trabajo. Así, para crear un nuevo sistema de archivos que trabaje con el VFS, los diseñadores del nuevo sistema de archivos se deben asegurar que suministre las llamadas a funciones que requiere el VFS. Un ejemplo obvio de dicha función es una que lee un bloque específico del disco, lo coloca en la caché de búfer del sistema de archivos y devuelve un apuntador a ese bloque. En consecuencia, el VFS tiene dos interfaces distintas: la superior para los procesos de usuario y la inferior para los sistemas de archivos concretos.

Aunque la mayoría de los sistemas bajo el VFS representan particiones en un disco local, éste no es siempre el caso. De hecho, la motivación original de Sun para construir el VFS era dar soporte a

los sistemas de archivos remotos mediante el uso del protocolo **NFS** (*Network File System*, Sistema de archivos de red). El diseño del VFS es tal que mientras el sistema de archivos concreto suministre las funciones que requiere el VFS, éste no sabe ni se preocupa por saber en dónde se almacenan los datos o como cuál es el sistema de archivos subyacente.

En el interior, la mayoría de las implementaciones de VFS son en esencia orientadas a objetos, aun si están escritas en C, en vez de C++. Hay varios tipos clave de objetos que se soportan normalmente. Éstos incluyen el superbloque (que describe a un sistema de archivos), el nodo-v (que describe a un archivo) y el directorio (que describe a un directorio del sistema de archivos). Cada uno de éstos tiene operaciones (métodos) asociadas que deben soportar los sistemas de archivos concretos. Además, el VFS tiene ciertas estructuras de datos internas para su propio uso, incluyendo la tabla de montaje y un arreglo de descriptores de archivos para llevar la cuenta de todos los archivos abiertos en los procesos de usuario.

Para comprender cómo funciona el VFS, veamos un ejemplo en forma cronológica. Cuando se arranca el sistema, el sistema de archivos raíz se registra con el VFS. Además, cuando se montan otros sistemas de archivos (ya sea en tiempo de arranque o durante la operación), éstos también se deben registrar con el VFS. Cuando un sistema de archivos se registra, lo que hace básicamente es proveer una lista de las direcciones de las funciones que requiere la VFS, ya sea como un vector (tabla) de llamadas extenso o como varios de ellos, uno por cada objeto VFS, según lo demande el VFS. Así, una vez que se ha registrado un sistema de archivos con el VFS, éste sabe cómo leer un bloque de ese sistema, por ejemplo: simplemente llama a la cuarta (o cualquier otra) función en el vector suministrado por el sistema de archivos. De manera similar, el VFS sabe entonces también cómo llevar a cabo cada una de las demás funciones que debe suministrar el sistema de archivos concreto: sólo llama a la función cuya dirección se suministró cuando se registró el sistema de archivos.

Una vez que se ha montado un sistema de archivos, se puede utilizar. Por ejemplo, si se ha montado un sistema de archivos en */usr* y un proceso realiza la llamada

```
open("/usr/include/unistd.h", O_RDONLY)
```

al analizar sintácticamente la ruta, el VFS ve que se ha montado un nuevo sistema de archivos en */usr* y localiza su superbloque, para lo cual busca en la lista de superbloques de los sistemas de archivos montados. Habiendo realizado esto, puede encontrar el directorio raíz del sistema de archivos montado y buscar la ruta *include/unistd.h* ahí. Entonces, el VFS crea un nodo-v y hace una llamada al sistema de archivos concreto para que devuelva toda la información en el nodo-i del archivo. Esta información se copia al nodo-v (en RAM) junto con otra información, siendo la más importante el apuntador a la tabla de funciones a llamar para las operaciones con los nodos-v, como `read`, `write`, `close`, etcétera.

Una vez que se ha creado el nodo-v, el VFS crea una entrada en la tabla de descriptores de archivos para el proceso que está haciendo la llamada y la establece para que apunte al nuevo nodo-v (para los puristas, el descriptor de archivo en realidad apunta a otra estructura de datos que contiene la posición actual en el archivo y un apuntador al nodo-v, pero este detalle no es importante para nuestros propósitos aquí). Por último, el VFS devuelve el descriptor de archivo al llamador, de manera que lo pueda utilizar para leer, escribir y cerrar el archivo.

Posteriormente, cuando el proceso realiza una operación `read` utilizando el descriptor de archivo, el VFS localiza el nodo-v del proceso y las tablas de descriptores de archivos, siguiendo el apuntador hasta la tabla de funciones, todas las cuales son direcciones dentro del sistema de archivos concreto en el que reside el archivo solicitado. La función que maneja a `read` se llama ahora y el código dentro del sistema de archivos concreto obtiene el bloque solicitado. El VFS no tiene idea acerca de si los datos provienen del disco local, de un sistema de archivos remoto a través de la red, un CD-ROM, una memoria USB o de algo distinto. Las estructuras de datos involucradas se muestran en la figura 4-19. Empezando con el número de proceso del llamador y el descriptor de archivo, se localizan en forma sucesiva el nodo-v, el apuntador de la función `read` y la función de acceso dentro del sistema de archivos concreto.

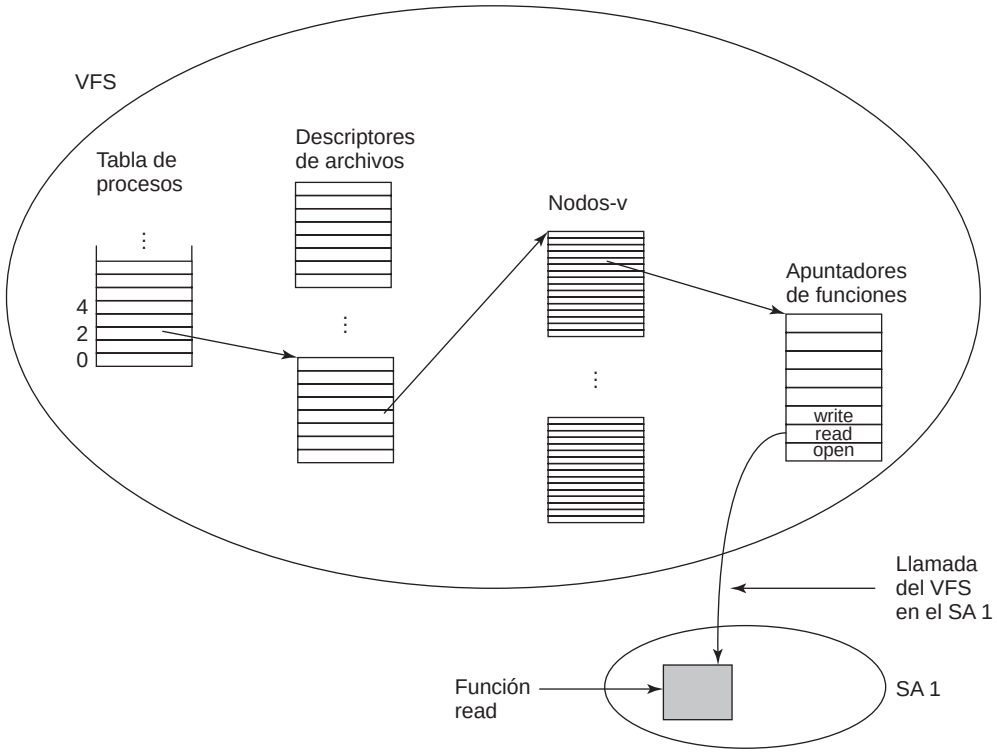

Figura 4-19. Una vista simplificada de las estructuras de datos y el código utilizados por el VFS y el sistema de archivos concreto para realizar una `read`.

De esta forma, es relativamente directo agregar nuevos sistemas de archivos. Para hacer uno, los diseñadores primero obtienen una lista de llamadas a funciones que espera el VFS y después escriben su sistema de archivos para proveer todas. De manera alternativa, si el sistema de archivos ya existe, entonces tienen que proveer funciones de envoltura que hagan lo que el VFS necesita, por lo general realizando una o más llamadas nativas al sistema de archivos concreto.

4.4 ADMINISTRACIÓN Y OPTIMIZACIÓN DE SISTEMAS DE ARCHIVOS

Hacer que el sistema de archivos funcione es una cosa; hacerlo que funcione de manera eficiente y robusta en la vida real es algo muy distinto. En las siguientes secciones analizaremos algunas de las cuestiones involucradas en la administración de discos.

4.4.1 Administración del espacio en disco

Por lo general los archivos se almacenan en disco, así que la administración del espacio en disco es una cuestión importante para los diseñadores de sistemas de archivos. Hay dos estrategias generales posibles para almacenar un archivo de *n* bytes: se asignan *n* bytes consecutivos de espacio en disco o el archivo se divide en varios bloques (no necesariamente) contiguos. La misma concesión está presente en los sistemas de administración de memoria, entre la segmentación pura y la paginación.

Como hemos visto, almacenar un archivo como una secuencia contigua de bytes tiene el problema obvio de que si un archivo crece, probablemente tendrá que moverse en el disco. El mismo problema se aplica a los segmentos en memoria, excepto que la operación de mover un segmento en memoria es rápida, en comparación con la operación de mover un archivo de una posición en el disco a otra. Por esta razón, casi todos los sistemas de archivos dividen los archivos en bloques de tamaño fijo que no necesitan ser adyacentes.

Tamaño de bloque

Una vez que se ha decidido almacenar archivos en bloques de tamaño fijo, surge la pregunta acerca de qué tan grande debe ser el bloque. Dada la forma en que están organizados los discos, el sector, la pista y el cilindro son candidatos obvios para la unidad de asignación (aunque todos ellos son dependientes del dispositivo, lo cual es una desventaja). En un sistema de paginación, el tamaño de la página también es uno de los principales contendientes.

Tener un tamaño de bloque grande significa que cada archivo (incluso un archivo de 1 byte) ocupa un cilindro completo. También significa que los pequeños archivos desperdician una gran cantidad de espacio en disco. Por otro lado, un tamaño de bloque pequeño significa que la mayoría de los archivos abarcarán varios bloques y por ende, necesitan varias búsquedas y retrasos rotacionales para leerlos, lo cual reduce el rendimiento. Por ende, si la unidad de asignación es demasiado grande, desperdiciamos espacio; si es demasiado pequeña, desperdiciamos tiempo.

Para hacer una buena elección hay que tener cierta información sobre la distribución de los tamaños de archivo. Tanenbaum y colaboradores (2006) estudiaron la distribución de los tamaños de archivo en el Departamento de Ciencias Computacionales de una gran universidad de investigación (la VU) en 1984 y después en el 2005, así como en un servidor Web comercial que hospedaba a un sitio Web político (*www.electoral-vote.com*). Los resultados se muestran en la figura 4-20, donde para cada tamaño de archivo de una potencia de dos, se lista el porcentaje de todos los archivos menores o iguales a éste para cada uno de los tres conjuntos de datos. Por ejemplo, en el 2005 el 59.13% de todos los archivos en la VU eran de 4 KB o menores y el 90.84% de todos los archivos

Longitud	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

Longitud	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

Figura 4-20. Porcentaje de archivos menores que un tamaño dado (en bytes).

eran de 64 KB o menores. El tamaño de archivo promedio era de 2475 bytes. Tal vez a algunas personas este tamaño tan pequeño les parezca sorprendente.

¿Qué conclusiones podemos obtener de estos datos? Por una parte, con un tamaño de bloque de 1 KB sólo entre 30 y 50% de todos los archivos caben en un solo bloque, mientras que con un bloque de 4 KB, el porcentaje de archivos que caben en un bloque aumenta hasta el rango entre 60 y 70%. Los demás datos en el artículo muestran que con un bloque de 4 KB, 93% de los bloques de disco son utilizados por 10% de los archivos más grandes. Esto significa que el desperdicio de espacio al final de cada pequeño archivo no es muy importante, debido a que el disco se llena por una gran cantidad de archivos grandes (videos) y la cantidad total de espacio ocupado por los pequeños archivos es insignificante. Incluso si se duplica el espacio que 90% de los archivos más pequeños ocuparían, sería insignificante.

Por otro lado, utilizar un bloque pequeño significa que cada archivo consistirá de muchos bloques. Para leer cada bloque normalmente se requiere una búsqueda y un retraso rotacional, por lo que la acción de leer un archivo que consista de muchos bloques pequeños será lenta.

Como ejemplo, considere un disco con 1 MB por pista, un tiempo de rotación de 8.33 mseg y un tiempo de búsqueda promedio de 5 mseg. El tiempo en milisegundos para leer un bloque de k bytes es entonces la suma de los tiempos de búsqueda, de retraso rotacional y de transferencia:

$$5 + 4.165 + (k/1000000) \times 8.33$$

La curva sólida de la figura 4-21 muestra la velocidad de los datos para dicho disco como una función del tamaño de bloque. Para calcular la eficiencia del espacio, necesitamos hacer una suposición acerca del tamaño de archivo promedio. Por simplicidad, vamos a suponer que todos los archivos son de 4 KB. Aunque este número es un poco más grande que los datos medidos en la VU, es probable

que los estudiantes tengan más archivos pequeños de los que estarían presentes en un centro de datos corporativo, por lo que podría ser una mejor aproximación en general. La curva punteada de la figura 4-21 muestra la eficiencia del espacio como una función del tamaño de bloque.

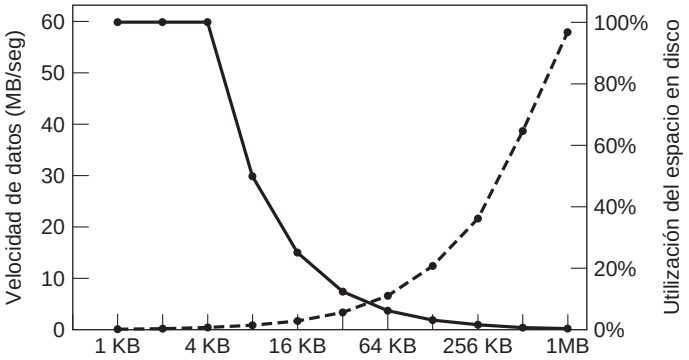

Figura 4-21. La curva sólida (escala del lado izquierdo) da la velocidad de datos del disco. La curva punteada (escala del lado derecho) da la eficiencia del espacio de disco. Todos los archivos son de 4 KB.

Las dos curvas se pueden comprender de la siguiente manera. El tiempo de acceso para un bloque está completamente dominado por el tiempo de búsqueda y el retraso rotacional, dado que se van a requerir 9 mseg para acceder a un bloque, entre más datos se obtengan será mejor. En consecuencia, la velocidad de datos aumenta casi en forma lineal con el tamaño de bloque (hasta que las transferencias tardan tanto que el tiempo de transferencia empieza a ser importante).

Ahora considere la eficiencia del espacio. Con archivos de 4 KB y bloques de 1 KB, 2 KB o 4 KB, los archivos utilizan el 4, 2 y 1 bloques, respectivamente, sin desperdicio. Con un bloque de 8 KB y archivos de 4 KB, la eficiencia del espacio disminuye hasta el 50% y con un bloque de 16 KB disminuye hasta 25%. En realidad, pocos archivos son un múltiplo exacto del tamaño de bloque del disco, por lo que siempre se desperdicia espacio en el último bloque de un archivo.

Sin embargo, lo que muestran las curvas es que el rendimiento y el uso del espacio se encuentran inherentemente en conflicto. Los bloques pequeños son malos para el rendimiento pero buenos para el uso del espacio en el disco. Para estos datos no hay un compromiso razonable disponible. El tamaño más cercano al punto en el que se cruzan las dos curvas es 64 KB, pero la velocidad de datos es de sólo 6.6 MB/seg y la eficiencia del espacio es de aproximadamente 7%; ninguno de estos dos valores son buenos. Históricamente, los sistemas de archivos han elegido tamaños en el rango de 1 KB a 4 KB, pero con discos que ahora exceden a 1 TB, podría ser mejor incrementar el tamaño de bloque a 64 KB y aceptar el espacio en disco desperdiciado. El espacio en disco ya no escasea en estos días.

En un experimento por ver si el uso de archivos en Windows NT era muy distinto al de UNIX, Vogels hizo mediciones en los archivos de la universidad Cornell University (Vogels, 1999). Él observó que el uso de archivos en NT es más complicado que en UNIX. Escribió lo siguiente:

> *Cuando escribimos unos cuantos caracteres en el editor de texto* bloc de notas (notepad)*, al guardar esto a un archivo se activarán 26 llamadas al sistema, incluyendo 3 intentos de apertura fallidos, 1 sobrescritura de archivo y 4 secuencias adicionales de abrir y cerrar.*

Sin embargo, observó un tamaño promedio (ponderado por uso) de archivos sólo leídos de 1 KB, de archivos sólo escritos de 2.3 KB y de archivos leídos y escritos de 4.2 KB. Dadas las distintas técnicas de medición de los conjuntos de datos y el año, estos resultados son sin duda compatibles con los resultados de la VU.

Registro de bloques libres

Una vez que se ha elegido un tamaño de bloque, la siguiente cuestión es cómo llevar registro de los bloques libres. Hay dos métodos utilizados ampliamente, como se muestra en la figura 4-22. El primero consiste en utilizar una lista enlazada de bloques de disco, donde cada bloque contiene tantos números de bloques de disco libres como pueda. Con un bloque de 1 KB y un número de bloque de disco de 32 bits, cada bloque en la lista de bloques libres contiene los números de 255 bloques libres (se requiere una ranura para el apuntador al siguiente bloque). Considere un disco de 500 GB, que tiene aproximadamente 488 millones de bloques de disco. Para almacenar todas estas direcciones a 255 por bloque, se requiere una cantidad aproximada de 1.9 millones de bloques. En general se utilizan bloques libres para mantener la lista de bloques libres, por lo que en esencia el almacenamiento es gratuito.

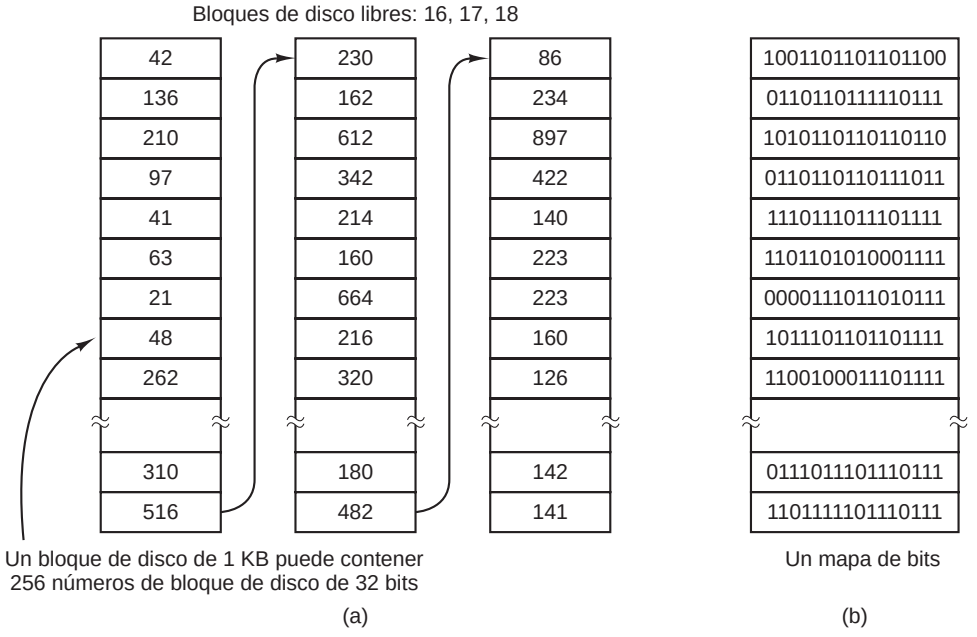

Figura 4-22. (a) Almacenamiento de la lista de bloques libres en una lista enlazada. (b) Un mapa de bits.

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 mediante 1s en el mapa, los bloques asignados mediante 0s (o viceversa). Para nuestro disco de 500 GB de ejemplo, necesitamos

488 millones de bits para el mapa, que requieren poco menos de 60,000 bloques de 1 KB para su almacenamiento. No es sorpresa que el mapa de bits requiera menos espacio, ya que utiliza 1 bit por bloque, en comparación con 32 bits en el modelo de la lista enlazada. Sólo si el disco está casi lleno (es decir, que tenga pocos bloques libres) es cuando el esquema de la lista enlazada requiere menos bloques que el mapa de bits.

Si los bloques libres tienden a presentarse en series largas de bloques consecutivos, el sistema de la lista de bloques libres se puede modificar para llevar la cuenta de las series de bloques, en vez de bloques individuales. Una cuenta de 8, 16 o 32 bits se podría asociar con cada bloque dando el número de bloques libres consecutivos. En el mejor caso, un disco prácticamente vacío podría representarse mediante dos números: la dirección del primer bloque libre seguida de la cuenta de bloques libres. Por otro lado, si el disco se fragmenta demasiado, es menos eficiente llevar el registro de las series que de los bloques individuales, debido a que no sólo se debe almacenar la dirección, sino también la cuenta.

Esta cuestión ilustra un problema con el que los diseñadores de sistemas operativos se enfrentan a menudo. Hay varias estructuras de datos y algoritmos que se pueden utilizar para resolver un problema, pero para elegir el mejor se requieren datos que los diseñadores no tienen y no tendrán sino hasta que el sistema se opere y se utilice con frecuencia. E incluso entonces, tal vez los datos no estén disponibles. Por ejemplo, nuestras propias mediciones de los tamaños de archivo en la VU en 1984 y 1995, los datos del sitio Web y los datos de Cornell son sólo cuatro muestras. Aunque es mejor que nada, tenemos una idea muy vaga acerca de si también son representativos de las computadoras domésticas, las computadoras empresariales, las computadoras gubernamentales y otras. Con algo de esfuerzo podríamos haber obtenido un par de muestras de otros tipos de computadoras, pero incluso así sería imprudente extrapolar a todas las computadoras de los tipos que se midieron.

Regresando por un momento al método de la lista de bloques libres, sólo hay que mantener un bloque de apuntadores en la memoria principal. Al crear un archivo, los bloques necesarios se toman del bloque de apuntadores. Cuando este bloque se agota, se lee un nuevo bloque de apuntadores del disco. De manera similar, cuando se elimina un archivo se liberan sus bloques y se agregan al bloque de apuntadores en la memoria principal. Cuando este bloque se llena, se escribe en el disco.

Bajo ciertas circunstancias, este método produce operaciones de E/S de disco innecesarias. Considere la situación de la figura 4-23(a), donde el bloque de apuntadores en memoria tiene espacio para sólo dos entradas más. Si se libera un archivo de tres bloques, el bloque de apuntadores se desborda y tiene que escribirse en el disco, con lo cual se produce la situación de la figura 4.23(b). Si ahora se escribe un archivo de tres bloques, se tiene que volver a leer el bloque completo de apuntadores del disco, lo cual nos regresa a la figura 4-23(a). Si el archivo de tres bloques que se acaba de escribir era temporal, al liberarlo se necesitará otra escritura de disco para escribir el bloque completo de apuntadores de vuelta en el disco. En resumen, cuando el bloque de apuntadores está casi vacío, una serie de archivos temporales de un tiempo corto de vida puede producir muchas operaciones de E/S de disco.

Un método alternativo que evita la mayor parte de estas operaciones de E/S de disco es dividir el bloque completo de apuntadores. Así, en vez de pasar de la figura 4-23(a) a la figura 4-23(b), pasamos de la figura 4-23(a) a la figura 4-23(c) cuando se liberan tres bloques. Ahora el sistema puede manejar una serie de archivos temporales sin realizar ninguna operación de E/S. Si el bloque en memoria se llena, se escribe en el disco y se lee el bloque medio lleno del disco. La idea aquí es mantener la

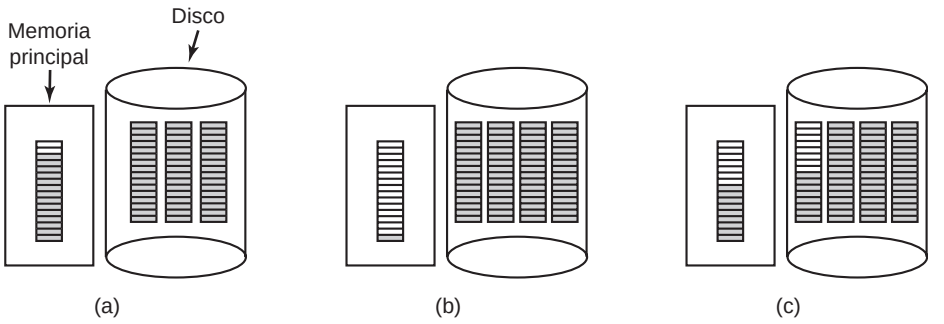

Figura 4-23. (a) Un bloque casi lleno de apuntadoras a bloques de disco libres en la memoria y tres bloques de apuntadores en el disco. (b) Resultado de liberar un archivo de tres bloques. (c) Una estrategia alternativa para manejar los tres bloques libres. Las entradas sombreadas representan apuntadores a bloques de disco libres.

mayor parte de los bloques de apuntadores en el disco llenos (para minimizar el uso del disco), pero mantener el que está en memoria lleno a la mitad, para que pueda manejar tanto la creación como la remoción de archivos sin necesidad de operaciones de E/S de disco en la lista de bloques libres.

Con un mapa de bits también es posible mantener sólo un bloque en memoria, usando el disco para obtener otro sólo cuando el primero se llena o se vacía. Un beneficio adicional de este método es que al realizar toda la asignación de un solo bloque del mapa de bits, los bloques de disco estarán cerca uno del otro, con lo cual se minimiza el movimiento del brazo del disco. Como el mapa de bits es una estructura de datos de tamaño fijo, si el kernel está paginado (parcialmente), el mapa de bits puede colocarse en memoria virtual y hacer que se paginen sus páginas según se requiera.

Cuotas de disco

Para evitar que los usuarios ocupen demasiado espacio en disco, los sistemas operativos multiusuario proporcionan un mecanismo para imponer las cuotas de disco. La idea es que el administrador del sistema asigne a cada usuario una cantidad máxima de archivos y bloques y que el sistema operativo se asegure de que los usuarios no excedan sus cuotas. A continuación describiremos un mecanismo común.

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

Una segunda tabla contiene el registro de cuotas para cada usuario con un archivo actualmente abierto, aun si el archivo fue abierto por alguien más. Esta tabla se muestra en la figura 4-24. Es un extracto de un archivo de cuotas en el disco para los usurarios cuyos archivos están actualmente abiertos. Cuando todos los archivos se cierran, el registro se escribe de vuelta en el archivo de cuotas.

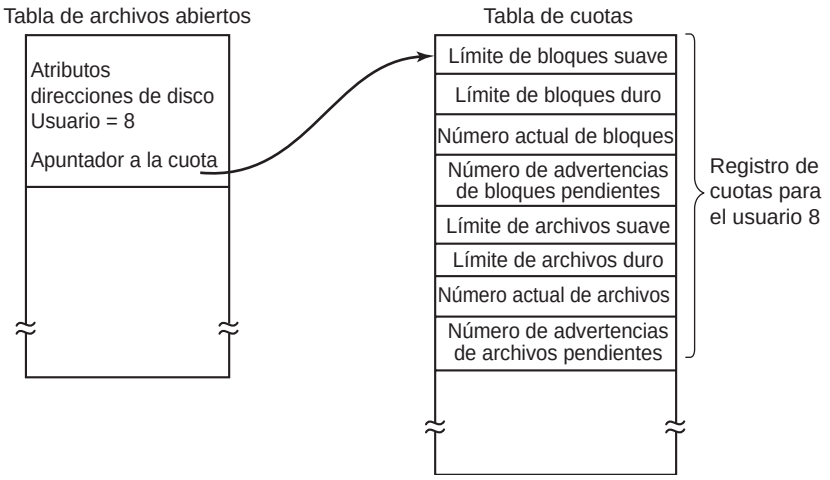

Figura 4-24. El registro de cuotas se lleva en una tabla de cuotas, usuario por usuario.

Cuando se crea una nueva entrada en la tabla de archivos abiertos, se introduce en ella un apuntador al registro de cuotas del propietario, para facilitar la búsqueda de los diversos límites. Cada vez que se agrega un bloque a un archivo se incrementa el número total de bloques que se cargan al propietario y se realiza una verificación con los límites duro y suave. Se puede exceder el límite suave, pero el límite duro no. Un intento de agregar datos a un archivo cuando se ha llegado al límite de bloques duro producirá un error. También existen verificaciones similares para el número de archivos.

Cuando un usuario trata de iniciar sesión, el sistema examina el archivo de cuotas para ver si el usuario ha excedido el límite suave para el número de archivos o el número de bloques de disco. Si se ha violado uno de los límites, se muestra una advertencia y la cuenta de advertencias restantes se reduce en uno. Si la cuenta llega a cero en algún momento, el usuario ha ignorado demasiadas veces la advertencia y no se le permite iniciar sesión. Para obtener permiso de iniciar sesión otra vez tendrá que hablar con el administrador del sistema.

Este método tiene la propiedad de que los usuarios pueden sobrepasar sus límites suaves durante una sesión, siempre y cuando eliminen el exceso cuando cierren su sesión. Los límites duros nunca se pueden exceder.

4.4.2 Respaldos del sistema de archivos

La destrucción de un sistema de archivos es a menudo un desastre aún mayor que la destrucción de una computadora. Si una computadora se destruye debido a un incendio, tormentas eléctricas o si se derrama una taza de café en el teclado, es molesto y costará dinero, pero en general se puede comprar un reemplazo sin muchos problemas. Las computadoras personales económicas se pueden reemplazar incluso en un lapso no mayor a una hora, con sólo ir a una tienda de computadoras

(excepto en las universidades, donde para emitir una orden de compra se requieren tres comités, cinco firmas y 90 días).

Si el sistema de archivos de una computadora se pierde de manera irrevocable, ya sea debido a hardware o software, será difícil restaurar toda la información, llevará mucho tiempo y en muchos casos, podrá ser imposible. Para las personas cuyos programas, documentos, registros fiscales, archivos de clientes, bases de datos, planes de comercialización o demás datos se pierden para siempre, las consecuencias pueden ser catastróficas. Aunque el sistema de archivos no puede ofrecer protección contra la destrucción física del equipo y los medios, sí puede ayudar a proteger la información. Es muy simple: realizar respaldos. Pero eso no es tan sencillo como parece. Demos un vistazo.

La mayoría de las personas no creen que realizar respaldos de sus archivos valga la pena en cuanto al tiempo y esfuerzo; hasta que un día su disco duro deja de funcionar en forma repentina, momento en el cual la mayoría de esas personas cambian de opinión demasiado tarde. Sin embargo, las empresas (por lo general) comprenden bien el valor de sus datos y en general realizan un respaldo por lo menos una vez al día, casi siempre en cinta. Las cintas modernas contienen cientos de gigabytes a un costo de varios centavos por gigabyte. Sin embargo, realizar respaldos no es tan trivial como se oye, por lo que a continuación analizaremos algunas de las cuestiones relacionadas.

Por lo general se realizan respaldos en cinta para manejar uno de dos problemas potenciales:

1. Recuperarse de un desastre.
2. Recuperarse de la estupidez.

El primer problema trata acerca de cómo hacer que la computadora vuelva a funcionar después de una falla general en el disco, un incendio, una inundación o cualquier otra catástrofe natural. En la práctica estas cosas no ocurren muy a menudo, razón por la cual muchas personas no se preocupan por realizar respaldos. Estas personas además tienden a no tener seguros contra incendios en sus hogares por la misma razón.

La segunda razón es que a menudo los usuarios remueven de manera accidental archivos, que más tarde vuelven a necesitar. Este problema ocurre con tanta frecuencia que cuando se “elimina” un archivo en Windows, no se elimina del todo, sino que sólo se mueve a un directorio especial, conocido como **Papelera de reciclaje**, para que pueda restaurarse con facilidad en un momento posterior. Los respaldos llevan este principio más allá y permiten que los archivos que se removieron hace días, o incluso semanas, se restauren desde las cintas de respaldo antiguas.

Para realizar un respaldo se requiere mucho tiempo y se ocupa una gran cantidad de espacio, por lo que es importante hacerlo con eficiencia y conveniencia. Estas consideraciones dan pie a las siguientes cuestiones. En primer lugar, ¿debe respaldarse el sistema completo o sólo parte de él? En muchas instalaciones, los programas ejecutables (binarios) se mantienen en una parte limitada del árbol del sistema de archivos. No es necesario respaldar esos archivos si pueden volver a instalarse de los CD-ROM de los fabricantes. Además, la mayoría de los sistemas tienen un directorio para los archivos temporales. Por lo general no hay razón para respaldarlos tampoco. En UNIX, todos los archivos especiales (dispositivos de E/S) se mantienen en un directorio */dev*. No sólo es innecesario respaldar este directorio, sino que es bastante peligroso debido a que el programa de respaldo se quedaría paralizado para siempre si tratara de leer cada uno de estos archivos hasta que se completaran. En resumen, por lo general es conveniente respaldar sólo directorios específicos y todo lo que contengan, en vez de respaldar todo el sistema de archivos.

En segundo lugar, es un desperdicio respaldar archivos que no han cambiado desde el último respaldo, lo cual nos lleva a la idea de los **vaciados incrementales**. La forma más simple de vaciado incremental es realizar un vaciado completo (respaldo) en forma periódica, por decir cada semana o cada mes, y realizar un vaciado diario de sólo aquellos archivos que se hayan modificado desde el último vaciado completo. Mejor aún es vaciar sólo los archivos que han cambiado desde la última vez que se vaciaron. Aunque este esquema minimiza el tiempo de vaciado, complica más la recuperación debido a que se tiene que restaurar el vaciado completo más reciente, seguido de todos los vaciados incrementales en orden inverso. Para facilitar la recuperación se utilizan con frecuencia esquemas de vaciado incremental más sofisticados.

En tercer lugar, como por lo general se vacían inmensas cantidades de datos, puede ser conveniente comprimir los datos antes de escribirlos en cinta. Sin embargo, con muchos algoritmos de compresión un solo punto malo en la cinta de respaldo puede frustrar el algoritmo de descompresión y hacer que todo un archivo o incluso toda una cinta, sea ilegible. Por ende, la decisión de comprimir el flujo de respaldo se debe considerar con cuidado.

En cuarto lugar, es difícil llevar a cabo un respaldo en un sistema de archivos activo. Si se están agregando, eliminando y modificando archivos y directorios durante el proceso de vaciado, el vaciado resultante puede ser inconsistente. Sin embargo, como para realizar un vaciado se pueden requerir horas, tal vez sea necesario desconectar el sistema de la red durante gran parte de la noche para realizar el respaldo, algo que no siempre es aceptable. Por esta razón se han ideado algoritmos para tomar instantáneas rápidas del sistema de archivos al copiar las estructuras de datos críticas y después requerir que los futuros cambios a los archivos y directorios copien los bloques en vez de actualizarlos al instante (Hutchinson y colaboradores, 1999). De esta forma, el sistema de archivos en efecto se congela al momento de la instantánea, por lo que se puede respaldar a conveniencia después de ello.

En quinto y último lugar, al realizar respaldos se introducen en una organización muchos problemas que no son técnicos. El mejor sistema de seguridad en línea del mundo sería inútil si el administrador del sistema mantiene todas las cintas de respaldo en su oficina y la deja abierta y desprotegida cada vez que se dirige por el pasillo a recoger los resultados de la impresora. Todo lo que un espía tiene que hacer es entrar por un segundo, colocar una pequeña cinta en su bolsillo y salir con tranquilidad y confianza. Adiós a la seguridad. Además, realizar un respaldo diario es de poca utilidad si el incendio que quemó a las computadoras también quema todas las cintas de respaldo. Por esta razón, las cintas de respaldo se deben mantener fuera del sitio, pero eso introduce más riesgos de seguridad (debido a que ahora se deben asegurar dos sitios). Para ver un análisis detallado de éstas y otras cuestiones de administración prácticas, consulte (Nemeth y colaboradores, 2000). A continuación sólo analizaremos las cuestiones técnicas implicadas en realizar respaldos del sistema de archivos.

Se pueden utilizar dos estrategias para vaciar un disco en la cinta: un vaciado físico o un vaciado lógico. Un **vaciado físico** empieza en el bloque 0 del disco, escribe todos los bloques del disco en la cinta de salida en orden y se detiene cuando acaba de copiar el último. Dicho programa es tan simple que probablemente pueda hacerse 100% libre de errores, algo que probablemente no se pueda decir acerca de cualquier otro programa útil.

Sin embargo, vale la pena hacer varios comentarios acerca del vaciado físico. Por una parte, no hay ningún valor en respaldar bloques de disco sin utilizar. Si el programa de vaciado puede obte-

ner acceso a la estructura de datos de bloques libres, puede evitar vaciar los bloques no utilizados. Sin embargo, para omitir los bloques no utilizados hay que escribir el número de cada bloque en frente de éste (o su equivalente), ya que en este caso no se aplica que el bloque *k* en la cinta sea el bloque *k* en el disco.

Una segunda preocupación es la de vaciar bloques defectuosos. Es casi imposible fabricar discos grandes sin ningún defecto. Siempre hay algunos bloques defectuosos presentes. Algunas veces cuando se realiza un formato de bajo nivel se detectan los bloques defectuosos, se marcan y se reemplazan por bloques reservados al final de cada pista para tales emergencias. En muchos casos, el controlador del disco se encarga del reemplazo de bloques defectuosos de manera transparente, sin que el sistema operativo sepa siquiera acerca de ello.

Sin embargo, algunas veces los bloques se vuelven defectuosos después del formato, en cuyo caso el sistema operativo los detectará en un momento dado. Por lo general, resuelve el problema creando un "archivo" consistente en todos los bloques malos, sólo para asegurarse que nunca aparezcan en la reserva de bloques libres y que nunca se asignen. Este archivo, sobra decirlo, es completamente ilegible.

Si todos los bloques defectuosos son reasociados por el controlador de disco y se ocultan del sistema operativo como acabamos de describir, el vaciado físico funciona bien. Por otro lado, si están visibles para el sistema operativo y se mantienen en uno o más archivos de bloques defectuosos o mapas de bits, es en lo absoluto esencial que el programa de vaciado físico obtenga acceso a esta información y evite vaciarlos, para evitar interminables errores de lectura del disco al tratar de respaldar el archivo de bloques defectuosos.

Las principales ventajas del vaciado físico son la simplicidad y una gran velocidad (básicamente, puede operar a la velocidad del disco). Las principales desventajas son la incapacidad de omitir directorios seleccionados, realizar vaciados incrementales y restaurar archivos individuales a petición del usuario. Por estas razones, la mayoría de las instalaciones realizan vaciados lógicos.

Un **vaciado lógico** empieza en uno o más directorios especificados y vacía en forma recursiva todos los archivos y directorios que se encuentran ahí que hayan sido modificados desde cierta fecha base dada (por ejemplo, el último respaldo para un vaciado incremental o la instalación del sistema para un vaciado completo). Así, en un vaciado lógico la cinta de vaciado obtiene una serie de directorios y archivos cuidadosamente identificados, lo cual facilita la restauración de un archivo o directorio específico a petición del usuario.

Como el vaciado lógico es la forma más común, vamos a examinar un algoritmo común en forma detallada, utilizando el ejemplo de la figura 4-25 para guiarnos. La mayoría de los sistemas UNIX utilizan este algoritmo. En la figura podemos ver un árbol de archivos con directorios (cuadros) y archivos (círculos). Los elementos sombreados se han modificado desde la fecha base y por ende necesitan vaciarse. Los que no están sombreados no necesitan vaciarse.

Este algoritmo también vacía todos los directorios (incluso los que no se han modificado) que se encuentran en la ruta hacia un archivo o directorio modificado por dos razones. En primer lugar, para que sea posible restaurar los archivos y directorios vaciados a un sistema de archivos fresco en una computadora distinta. De esta forma, los programas de vaciado y restauración se pueden utilizar para transportar sistemas de archivos completos entre computadoras.

La segunda razón de vaciar directorios no modificados que estén arriba de archivos modificados es para que sea posible restaurar un solo archivo en forma incremental (tal vez para manejar

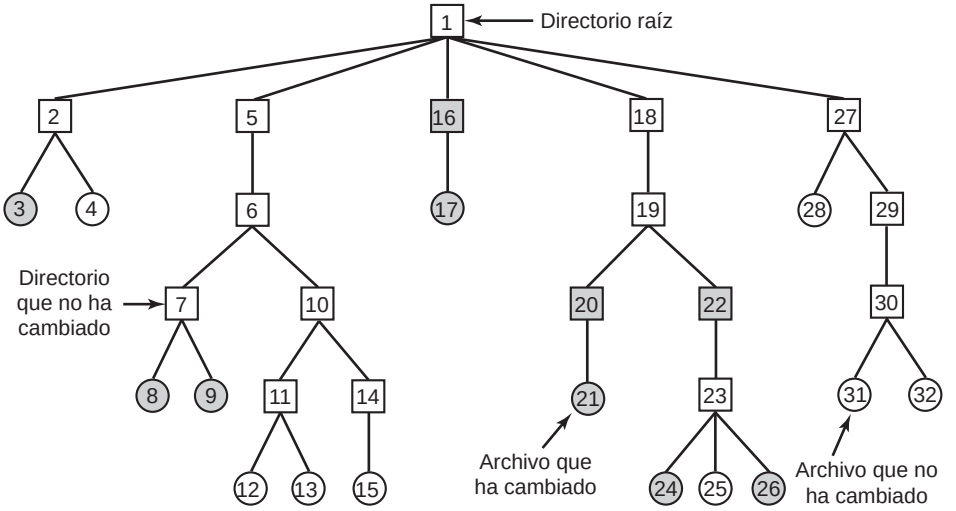

Figura 4-25. Un sistema de archivos que se va a vaciar. Los cuadros son directorios y los círculos son archivos. Los elementos sombreados han sido modificados desde el último vaciado. Cada directorio y archivo está etiquetado con base en su número de nodo-i.

la recuperación por causa de estupidez). Suponga que se realiza un vaciado completo del sistema de archivos el domingo por la tarde y un vaciado incremental el lunes por la tarde. El martes se remueve el directorio */usr/jhs/proy/nr3*, junto con todos los directorios y archivos debajo de él. El miércoles en la mañana, el usuario desea restaurar el archivo */usr/jhs/proy/nr3/planes/resumen*. Sin embargo, no es posible restaurar sólo el archivo *resumen* debido a que no hay lugar en dónde ponerlo. Los directorios *nr3* y *planes* se deben restaurar primero. Para obtener los datos correctos sobre sus propietarios, modos, horas y demás, estos directorios deben estar presentes en la cinta de vaciado, aun cuando no hayan sido modificados desde la última vez que ocurrió un vaciado completo.

El algoritmo de vaciado mantiene un mapa de bits indexado por número de nodo-i con varios bits por nodo-i. Los bits se activarán y borrarán en el mapa, a medida que el algoritmo realice su trabajo. El algoritmo opera en cuatro fases. La fase 1 empieza en el directorio inicial (el directorio raíz en este ejemplo) y examina todas las entradas que contiene. Para cada archivo modificado, se marca su nodo-i en el mapa de bits. Cada directorio también se marca (se haya modificado o no) y después se inspecciona de manera recursiva.

Al final de la fase 1, se han marcado todos los archivos y directorios modificados en el mapa de bits, como se muestra (mediante el sombreado) en la figura 4-26(a). En concepto, la fase 2 recorre en forma recursiva el árbol de nuevo, desmarcando los directorios que no tengan archivos o directorios modificados en ellos, o bajo ellos. Esta fase deja el mapa de bits que se muestra en la figura 4-26(b). Observe que los directorios 10, 11, 14, 27, 29 y 30 ahora están desmarcados, ya que no contienen nada bajo ellos que se haya modificado. Estos directorios no se vaciarán. Por el contrario, los directorios 5 y 6 se vaciarán incluso aunque en sí no se hayan modificado, ya que se ne-

cesitarán para restaurar los cambios de hoy en una máquina nueva. Por cuestión de eficiencia, las fases 1 y 2 se pueden combinar en un solo recorrido del árbol.

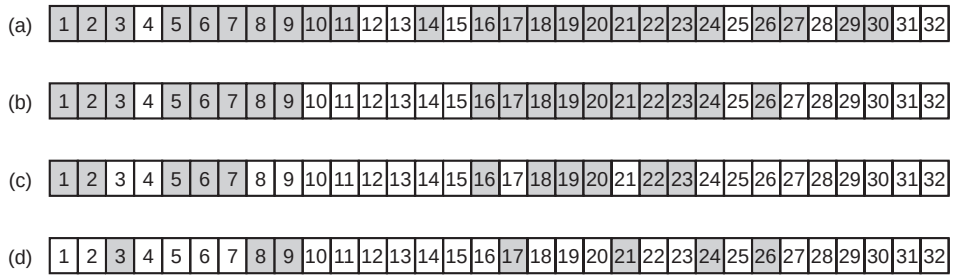

Figura 4-26. Mapas de bits utilizados por el algoritmo de vaciado lógico.

En este punto se sabe qué directorios y archivos se deben vaciar. Éstos son los que están marcados en la figura 4-26(b). La fase 3 consiste en explorar los nodos-i en orden numérico y vaciar todos los directorios marcados para vaciado. Éstos se muestran en la figura 4-26(c). Cada directorio tiene como prefijo sus atributos (propietario, horas, etc.) de manera que se pueda restaurar. Por último, en la fase 4 también se vacían los archivos marcados en la figura 4.26(d), que de nuevo tienen como prefijo sus atributos. Esto completa el vaciado.

Restaurar un sistema de archivos a partir de las cintas de vaciado es un proceso directo. Para empezar, se crea un sistema de archivos vacío en el disco. Después se restaura el vaciado completo más reciente. Como los directorios aparecen primero en la cinta, se restauran todos primero, con lo cual se obtiene un esqueleto del sistema de archivos. Después se restauran los archivos en sí. Este proceso se repite con el primer vaciado incremental realizado después del vaciado completo, después el siguiente y así en lo sucesivo.

Aunque el vaciado lógico es directo, hay unas cuantas cuestiones engañosas. Por ejemplo, como la lista de bloques libres no es un archivo, no se vacía y por ende se debe reconstruir desde cero, una vez que se han restaurado todos los vaciados. Esto es siempre posible, ya que el conjunto de bloques libres es sólo el complemento del conjunto de bloques contenidos en todos los archivos combinados.

Los vínculos son otra de las cuestiones. Si un archivo se vincula a uno o más directorios, es importante que el archivo se restaure sólo una vez y que todos los directorios que deben apuntar a él lo hagan.

Otra de las cuestiones es el hecho de que los archivos de UNIX pueden contener huecos. Es legal abrir un archivo, escribir unos cuantos bytes, después realizar una búsqueda hacia un desplazamiento de archivo distante y escribir unos cuantos bytes más. Los bloques en el medio no forman parte del archivo, por lo que no deben vaciarse ni restaurarse. A menudo los archivos básicos tienen un hueco de cientos de megabytes entre el segmento de datos y la pila. Si no se manejan en forma apropiada, cada archivo básico restaurado llenará esta área con ceros y por ende,

será del mismo tamaño que el espacio de direcciones virtuales (por ejemplo, de 2^{32} bytes, o peor aún, de 2^{64} bytes).

Por último, los archivos especiales (llamados canales) y sus equivalentes nunca deben vaciarse, sin importar en qué directorio se encuentren (no necesitan confinarse a */dev*). Para obtener más información acerca de los respaldos del sistema de archivos, consulte (Chervenak y colaboradores, 1998; y Zwicky, 1991).

Las densidades de las cintas no están mejorando con tanta rapidez como las densidades de los discos. Esto conlleva gradualmente a una situación en la que tal vez el respaldo de un disco muy grande requiera de varias cintas. Aunque hay robots disponibles para cambiar las cintas de manera automática, si esta tendencia continúa, en un momento dado las cintas serán demasiado pequeñas como para utilizarlas como medio de respaldo. En ese caso, la única forma de respaldar un disco será en otro disco. Aunque utilizar el método de reflejar cada disco con un repuesto es una posibilidad, en el capítulo 5 analizaremos esquemas más sofisticados, conocidos como RAIDs.

4.4.3 Consistencia del sistema de archivos

Otra área donde la confiabilidad es una cuestión importante es la de consistencia del sistema de archivos. Muchos sistemas de archivos leen bloques, los modifican y los escriben posteriormente. Si el sistema falla antes de escribir todos los bloques modificados, el sistema de archivos puede quedar en un estado inconsistente. Este problema es muy crítico si algunos de los bloques que no se han escrito son bloques de nodos-i, bloques de directorios o bloques que contienen la lista de bloques libres.

Para lidiar con el problema de los sistemas de archivos inconsistentes, la mayoría de las computadoras tienen un programa utilitario que verifica la consistencia del sistema de archivos. Por ejemplo, UNIX tiene a *fsck* y Windows tiene a *scandisk*. Esta herramienta se puede ejecutar cada vez que se arranca el sistema, en especial después de una falla. La descripción de más adelante nos indica cómo funciona *fsck*. *Scandisk* es un poco distinto, ya que opera en un sistema de archivos diferente, pero el principio general de utilizar la redundancia inherente del sistema de archivos para repararlo todavía es válido. Todos los verificadores de sistema de archivos comprueban cada sistema de archivos (partición de disco) de manera independiente de los demás.

Se pueden realizar dos tipos de verificaciones de consistencia: archivos y bloques. Para comprobar la consistencia de los bloques, el programa crea dos tablas, cada una de las cuales contiene un contador para cada bloque, que al principio se establece en 0. Los contadores en la primera tabla llevan el registro de cuántas veces está presente cada bloque en un archivo; los contadores en la segunda tabla registran con qué frecuencia está presente cada bloque en la lista de bloques libres (o en el mapa de bits de bloques libres).

Después el programa lee todos los nodos-i utilizando un dispositivo puro, el cual ignora la estructura de los archivos y sólo devuelve todos los bloques de disco que empiezan en 0. Si partimos de un nodo-i, es posible construir una lista de todos los números de bloque utilizados en el archivo correspondiente. A medida que se lee cada número de bloque, se incrementa su contador en la primera tabla. Después el programa examina la lista o mapa de bits de bloques libres para encontrar todos los bloques que no estén en uso. Cada ocurrencia de un bloque en la lista de bloques libres hace que se incremente su contador en la segunda tabla.

Si el sistema de archivos es consistente, cada bloque tendrá un 1 en la primera tabla o en la segunda, como se ilustra en la figura 4-27(a). Sin embargo, como resultado de una falla, las tablas podrían verse como en la figura 4-27(b), donde el bloque 2 no ocurre en ninguna de las dos tablas. Se reportará como un **bloque faltante**. Aunque los bloques faltantes no hacen un daño real, desperdician espacio y por ende reducen la capacidad del disco. La solución a los bloques faltantes es directa: el verificador del sistema de archivos sólo los agrega a la lista de bloques libres.

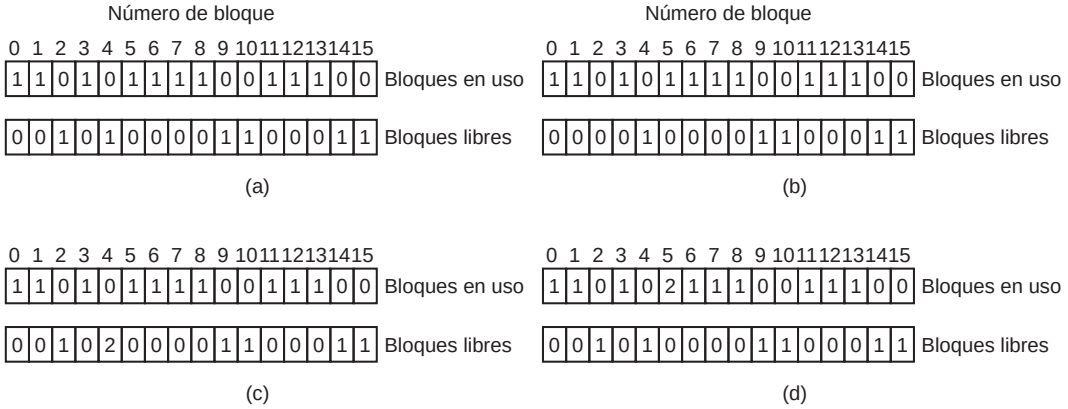

Figura 4-27. Estados del sistema de archivos. (a) Consistente. (b) Bloque faltante. (c) Bloque duplicado en la lista de bloques libres. (d) Bloque de datos duplicado.

Otra situación que podría ocurrir es la de la figura 4-27(c). Aquí podemos ver un bloque, el número 4, que ocurre dos veces en la lista de bloques libres (puede haber duplicados sólo si la lista de bloques libres es en realidad una lista; con un mapa de bits es imposible). La solución aquí también es simple: reconstruir la lista de bloques libres.

Lo peor que puede ocurrir es que el mismo bloque de datos esté presente en dos o más archivos, como se muestra en la figura 4.27(d) con el bloque 5. Si se remueve alguno de esos archivos, el bloque 5 se colocará en la lista de bloques libres, lo cual producirá una situación en la que el mismo bloque, al mismo tiempo, esté en uso y sea libre. Si se remueven ambos archivos, el bloque se colocará dos veces en la lista de bloques libres.

La acción apropiada que debe tomar el verificador del sistema de archivos es asignar un bloque libre, copiar el contenido del bloque 5 en él e insertar la copia en uno de los archivos. De esta forma, el contenido de información en los archivos no se modifica (aunque es muy probable que uno ellos termine con basura), pero por lo menos la estructura del sistema de archivos se hace consistente. El error se debe reportar para permitir que el usuario inspeccione los daños.

Además de verificar que cada bloque se haya contabilizado apropiadamente, el verificador del sistema de archivos también verifica el sistema de directorios. Utiliza también una tabla de contadores, pero éstos son por archivo, no por bloque. Empieza en el directorio raíz y desciende recursivamente por el árbol, inspeccionando cada directorio en el sistema de archivos. Para cada nodo-i en cada directorio, incrementa un contador para la cuenta de uso de ese archivo. Recuerde que debido

a los vínculos duros, un archivo puede aparecer en dos o más directorios. Los vínculos simbólicos no cuentan y no hacen que se incremente el contador para el archivo objetivo.

Cuando el verificador termina, tiene una lista indexada por número de nodo-i, que indica cuántos directorios contienen cada archivo. Después compara estos números con las cuentas de vínculos almacenadas en los mismos nodos-i. Estas cuentas empiezan en 1 cuando se crea un archivo y se incrementan cada vez que se crea un vínculo (duro) al archivo. En un sistema de archivos consistentes, ambas cuentas concordarán. Sin embargo, pueden ocurrir dos tipos de errores: que la cuenta de vínculos en el nodo-i sea demasiado alta o demasiado baja.

Si la cuenta de vínculos es mayor que el número de entradas en el directorio, entonces aun si se remueven todos los archivos de los directorios, la cuenta seguirá siendo distinta de cero y el nodo-i no se removerá. Este error no es grave, pero desperdicia espacio en el disco con archivos que no están en ningún directorio. Para corregirlo, se debe establecer la cuenta de vínculos en el nodo-i al valor correcto.

El otro error es potencialmente catastrófico. Si dos entradas en el directorio están vinculados a un archivo, pero el nodo-i dice que sólo hay una, cuando se elimine una de las dos entradas del directorio, la cuenta de nodos-i será cero. Cuando una cuenta de nodos-i queda en cero, el sistema de archivos la marca como no utilizada y libera todos sus bloques. Esta acción hará que uno de los directorios apunte ahora a un nodo-i sin utilizar, cuyos bloques pueden asignarse pronto a otros archivos. De nuevo, la solución es tan sólo obligar a que la cuenta de vínculos en el nodo-i sea igual al número actual de entradas del directorio.

Estas dos operaciones, verificar bloques y verificar directorios, se integran a menudo por cuestiones de eficiencia (es decir, sólo se requiere una pasada sobre los nodos-i). También son posibles otras comprobaciones. Por ejemplo, los directorios tienen un formato definido, con números de nodos-i y nombres ASCII. Si un número de nodo-i es más grande que el número de nodos-i en el disco, el directorio se ha dañado.

Además, cada nodo-i tiene un modo, algunos de los cuales son legales pero extraños, como 0007, que no permite ningún tipo de acceso al propietario y su grupo, pero permite a los usuarios externos leer, escribir y ejecutar el archivo. Podría ser útil por lo menos reportar los archivos que dan a los usuarios externos más derechos que al propietario. Los directorios con más de, por decir, 1000 entradas, son también sospechosos. Los archivos localizados en directorios de usuario, pero que son propiedad del superusuario y tienen el bit SETUID activado, son problemas potenciales de seguridad debido a que dichos archivos adquieren los poderes del superusuario cuando son ejecutados por cualquier usuario. Con un poco de esfuerzo, podemos reunir una lista bastante extensa de situaciones técnicamente legales, pero aún así peculiares, que podría valer la pena reportar.

En los párrafos anteriores hemos analizado el problema de proteger al usuario contra las fallas. Algunos sistemas de archivos también se preocupan por proteger al usuario contra sí mismo. Si el usuario trata de escribir

```
rm *.o
```

para quitar todos los archivos que terminen con *.o* (archivos objeto generados por el compilador), pero escribe por accidente

```
rm * .o
```

(observe el espacio después del asterisco), *rm* eliminará todos los archivos en el directorio actual y después se quejará de que no puede encontrar *.o*. En MS-DOS y algunos otros sistemas, cuando se remueve un archivo todo lo que ocurre es que se establece un bit en el directorio o nodo-i que marca el archivo como removido. No se devuelven bloques de disco a la lista de bloques libres sino hasta que realmente se necesiten. Así, si el usuario descubre el error de inmediato, es posible ejecutar un programa de utilería especial que "des-remueve" (es decir, restaura) los archivos removidos. En Windows, los archivos que se remueven se colocan en la papelera de reciclaje (un directorio especial), desde donde se pueden recuperar más tarde, si se da la necesidad. Desde luego que no se reclama el espacio de almacenamiento sino hasta que realmente se remuevan de este directorio.

4.4.4 Rendimiento del sistema de archivos

El acceso al disco es mucho más lento que el acceso a la memoria. Para leer una palabra de memoria de 32 bits se podrían requerir 10 nseg. La lectura de un disco duro se podría realizar a 100 MB/seg, que es cuatro veces más lenta que la de la palabra de 32 bits, pero a esto se le debe agregar de 5 a 10 mseg para realizar una búsqueda hasta la pista y después esperar a que se coloque el sector deseado bajo la cabeza de lectura. Si se necesita sólo una palabra, el acceso a memoria está en el orden de un millón de veces más rápido que el acceso al disco. Como resultado de esta diferencia en el tiempo de acceso, muchos sistemas de archivos se han diseñado con varias optimizaciones para mejorar el rendimiento. En esta sección hablaremos sobre tres de ellas.

Uso de caché

La técnica más común utilizada para reducir los accesos al disco es la **caché de bloques** o **caché de búfer** (caché se deriva del francés *cacher*, que significa ocultar). En este contexto, una caché es una colección de bloques que pertenecen lógicamente al disco, pero se mantienen en memoria por cuestiones de rendimiento.

Se pueden utilizar varios algoritmos para administrar la caché, pero un algoritmo común es verificar todas las peticiones de lectura para ver si el bloque necesario está en la caché. Si está, la petición de lectura se puede satisfacer sin necesidad de acceder al disco. Si el bloque no está en la caché, primero se lee en la caché y después se copia a donde sea necesario. Las peticiones posteriores de ese mismo bloque se pueden satisfacer desde la caché.

La operación de la caché se ilustra en la figura 4-28. Como hay muchos bloques (a menudo miles) en la caché, se necesita cierta forma de determinar con rapidez si cierto bloque está presente. La forma usual es codificar en hash la dirección de dispositivo y de disco, buscando el resultado en una tabla de hash. Todos los bloques con el mismo valor de hash se encadenan en una lista enlazada, de manera que se pueda seguir la cadena de colisiones.

Cuando se tiene que cargar un bloque en una caché llena, hay que eliminar cierto bloque (y volver a escribirlo en el disco, si se ha modificado desde la última vez que se trajo). Esta situación es muy parecida a la paginación y se pueden aplicar todos los algoritmos de reemplazo de página usuales descritos en el capítulo 3, como FIFO, segunda oportunidad y LRU. Una diferencia placentera entre la paginación y el uso de la caché es que las referencias a la caché son relativamente

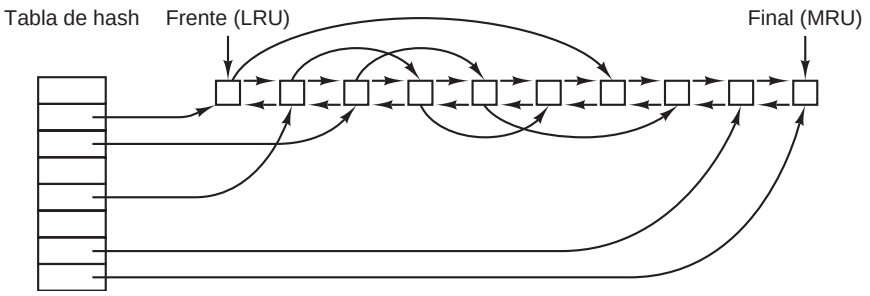

Figura 4-28. Estructuras de datos de la caché de búfer.

infrecuentes, por lo que es factible mantener todos los bloques en orden LRU exacto con listas enlazadas.

En la figura 4-28 podemos ver que además de las cadenas de colisión que empiezan en la tabla de hash, también hay una lista bidireccional que pasa por todos los bloques en el orden de uso, donde el bloque de uso menos reciente está al frente de esta lista y el bloque de uso más reciente está al final. Cuando se hace referencia a un bloque, se puede quitar de su posición en la lista bidireccional y colocarse al final. De esta forma, se puede mantener el orden LRU exacto.

Por desgracia hay una desventaja. Ahora que tenemos una situación en la que es posible el LRU exacto, resulta ser que esto no es deseable. El problema tiene que ver con las fallas y la consistencia del sistema de archivos que vimos en la sección anterior. Si un bloque crítico, como un bloque de nodos-i, se lee en la caché y se modifica pero no se vuelve a escribir en el disco, una falla dejará al sistema de archivos en un estado inconsistente. Si el bloque de nodos-i se coloca al final de la cadena LRU, puede pasar un buen tiempo antes de que llegue al frente y se vuelva a escribir en el disco.

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

1. ¿Es probable que el bloque se necesite pronto otra vez?
2. ¿Es el bloque esencial para la consistencia del sistema de archivos?

Para ambas preguntas, los bloques se pueden dividir en categorías como bloques de nodos-i, bloques indirectos, bloques de directorios, bloques de datos llenos y bloques de datos parcialmente llenos. Los bloques que tal vez no se necesiten de nuevo pronto pasan al frente, en vez de pasar al final de la lista LRU, por lo que sus búferes se reutilizarán con rapidez. Los bloques que podrían necesitarse pronto, como un bloque parcialmente lleno que se está escribiendo, pasan al final de la lista, por lo que permanecerán ahí por mucho tiempo.

La segunda pregunta es independiente de la primera. Si el bloque es esencial para la consistencia del sistema de archivos (básicamente, todo excepto los bloques de datos) y ha sido modificado, debe escribirse de inmediato en el disco, sin importar en cuál extremo de la lista LRU esté coloca-

do. Al escribir los bloques críticos con rapidez, reducimos en forma considerable la probabilidad de que una falla estropee todo el sistema de archivos. Aunque un usuario puede estar inconforme si uno de sus archivos se arruina en una falla, es probable que esté más inconforme si se pierde todo el sistema de archivos.

Incluso con esta medida para mantener intacta la integridad del sistema de archivos, es indeseable mantener los bloques de datos en la caché por mucho tiempo antes de escribirlos. Considere la situación apremiante de alguien que utiliza una computadora personal para escribir un libro. Aun si nuestro escritor indica en forma periódica al editor que escriba en el disco el archivo que se está editando, hay una buena probabilidad de que todo esté todavía en la caché y no haya nada en el disco. Si el sistema falla, la estructura del sistema de archivos no se volverá corrupta, pero se perderá todo un día de trabajo.

Esta situación no necesita ocurrir con mucha frecuencia para que un usuario esté bastante molesto. Los sistemas utilizan dos esquemas para lidiar con ella. La forma utilizada por UNIX es hacer una llamada al sistema, `sync`, que obligue a que se escriban todos los bloques modificados en el disco de inmediato. Cuando el sistema se arranca, un programa conocido comúnmente como *update* se inicia en segundo plano para permanecer en un ciclo sin fin emitiendo llamadas a `sync`, permaneciendo inactivo durante 30 segundos entre una llamada y otra. Como resultado, no se pierden más que 30 segundos de trabajo debido a una falla.

Aunque Windows tiene ahora una llamada al sistema equivalente a `sync`, conocida como `FlushFileBuffers`, en el pasado no tenía nada. En vez de ello, tenía una estrategia diferente que en ciertos aspectos era mejor que el esquema de UNIX (y en otros aspectos era peor). Lo que hacía era escribir cada bloque modificado en el disco tan pronto como se había escrito en la caché. Las cachés en las que todos los bloques modificados se escriben de inmediato en el disco se conocen como **cachés de escritura inmediata**. Requieren más operaciones de E/S de disco que las cachés que no son de escritura inmediata.

La diferencia entre estos dos esquemas se puede ver cuando un programa escribe un bloque lleno de 1 KB, un carácter a la vez. UNIX recolectará todos los caracteres en la caché y escribirá el bloque en el disco una vez cada 30 segundos o cuando el bloque se elimine de la caché. Con una caché de escritura inmediata, hay un acceso al disco por cada carácter escrito. Desde luego que la mayoría de los programas utilizan un búfer interno, por lo que generalmente no escriben un carácter, sino una línea o una unidad más grande en cada llamada al sistema `write`.

Una consecuencia de esta diferencia en la estrategia de uso de cachés es que con sólo sacar un disco (flexible) de un sistema UNIX sin realizar una llamada a `sync`, casi siempre se perderán datos y con frecuencia se obtendrá un sistema de archivos corrupto también. Con la caché de escritura inmediata no surge ningún problema. Estas distintas estrategias se eligieron debido a que UNIX se desarrolló en un entorno en el que todos los discos eran discos duros y no removibles, mientras que el primer sistema de archivos de Windows se heredó de MS-DOS, que empezó en el mundo del disco flexible. A medida que los discos duros se convirtieron en la norma, el método de UNIX, con la mejor eficiencia (pero la peor confiabilidad) se convirtió en la norma y también se utiliza ahora en Windows para los discos duros. Sin embargo, NTFS toma otras medidas (por bitácora) para mejorar la confiabilidad, como vimos antes.

Algunos sistemas operativos integran la caché de búfer con la caché de páginas. Esto es en especial atractivo cuando se soportan archivos asociados a memoria. Si un archivo se asocia a la

memoria, entonces algunas de sus páginas pueden estar en memoria debido a que se paginaron bajo demanda. Dichas páginas no son muy distintas a los bloques de archivos en la caché de búfer. En este caso se pueden tratar de la misma forma, con una sola caché tanto para bloques de archivos como para páginas.

Lectura adelantada de bloque

Una segunda técnica para mejorar el rendimiento percibido por el sistema de archivos es tratar de colocar bloques en la caché antes de que se necesiten, para incrementar la proporción de aciertos. En especial, muchos archivos se leen en forma secuencial. Cuando se pide al sistema de archivos que produzca el bloque k en un archivo, hace eso, pero cuando termina realiza una verificación disimulada en la caché para ver si el bloque $k + 1$ ya está ahí. Si no está, planifica una lectura para el bloque $k + 1$ con la esperanza de que cuando se necesite, ya haya llegado a la caché. Cuando menos, estará en camino.

Desde luego que esta estrategia de lectura adelantada sólo funciona para los archivos que se leen en forma secuencial. Si se accede a un archivo en forma aleatoria, la lectura adelantada no ayuda. De hecho afecta, ya que ocupa ancho de banda del disco al leer bloques inútiles y eliminar bloques potencialmente útiles de la caché (y tal vez ocupando más ancho de banda del disco al escribirlos de vuelta al disco si están sucios). Para ver si vale la pena realizar la lectura adelantada, el sistema de archivos puede llevar un registro de los patrones de acceso a cada archivo abierto. Por ejemplo, un bit asociado con cada archivo puede llevar la cuenta de si el archivo está en "modo de acceso secuencial" o en "modo de acceso aleatorio". Al principio, el archivo recibe el beneficio de la duda y se coloca en modo de acceso secuencial. Sin embargo, cada vez que se realiza una búsqueda, el bit se desactiva. Si empiezan de nuevo las lecturas secuenciales, el bit se activa otra vez. De esta manera, el sistema de archivos puede hacer una estimación razonable acerca de si debe utilizar o no la lectura adelantada. Si se equivoca de vez en cuando, no es un desastre, sólo un poco de ancho de banda de disco desperdiciado.

Reducción del movimiento del brazo del disco

El uso de caché y la lectura adelantada no son las únicas formas de incrementar el rendimiento del sistema de archivos. Otra técnica importante es reducir la cantidad de movimiento del brazo del disco, al colocar los bloques que tengan una buena probabilidad de utilizarse en secuencia cerca unos de otros, de preferencia en el mismo cilindro. Cuando se escribe un archivo de salida, el sistema de archivos tiene que asignar los bloques uno a la vez, bajo demanda. Si los bloques libres se registran en un mapa de bits y todo el mapa de bits se encuentra en la memoria principal, es bastante sencillo elegir un bloque libre lo más cerca posible del bloque anterior. Con una lista de bloques libres, parte de la cual está en el disco, es mucho más difícil asignar bloques que estén cerca unos de otros.

Sin embargo, incluso con una lista de bloques libres, se puede llevar a cabo cierta agrupación de bloques. El truco es llevar la cuenta del almacenamiento de disco no en bloques, sino en grupos de bloques consecutivos. Si todos los sectores consisten de 512 bytes, el sistema podría utilizar bloques

de 1 KB (2 sectores) pero asignar almacenamiento del disco en unidades de 2 bloques (4 sectores). Esto no es lo mismo que tener bloques de disco de 2 KB, ya que la caché seguiría utilizando bloques de 1 KB y las transferencias de disco seguirían siendo de 1 KB, pero al leer un archivo en forma secuencial en un sistema que de otra manera estaría inactivo, se reduciría el número de búsquedas por un factor de dos, lo cual mejoraría el rendimiento en forma considerable. Una variación en el mismo tema es tomar en cuenta el posicionamiento rotacional. Al asignar bloques, el sistema intenta colocar bloques consecutivos en un archivo en el mismo cilindro.

Otro factor que reduce el rendimiento en los sistemas que utilizan nodos-i o cualquier cosa como ellos es que al leer incluso hasta un archivo corto, se requieren dos accesos: uno para el nodo-i y otro para el bloque. La colocación común de nodos-i se muestra en la figura 4-29(a). Aquí todos los nodos están cerca del principio del disco, por lo que la distancia promedio entre un nodo-i y sus bloques será de aproximadamente la mitad del número de cilindros, con lo cual se requieren búsquedas extensas.

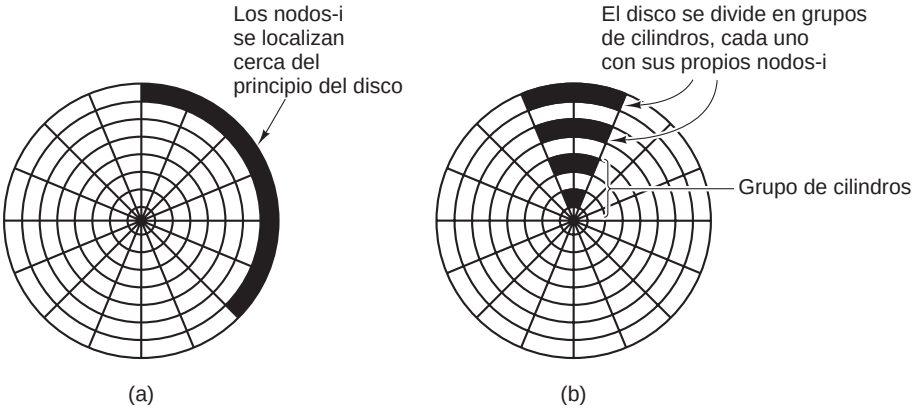

Figura 4-29. (a) Los nodos-i colocados al principio del disco. (b) Disco dividido en grupos de cilindros, cada uno con sus propios bloques y nodos-i.

Una mejora fácil en el rendimiento es colocar los nodos-i en medio del disco, en vez de hacerlo al principio, con lo cual se reduce la búsqueda promedio entre el nodo-i y el primer bloque por un factor de dos. Otra idea, que se muestra en la figura 4-29(b), es dividir el disco en grupos de cilindros, cada uno con sus propios nodos-i, bloques y lista de bloques libres (McKusick y colaboradores, 1984). Al crear un nuevo archivo, se puede elegir cualquier nodo-i, pero se hace un intento por encontrar un bloque en el mismo grupo de cilindros que el nodo-i. Si no hay uno disponible, entonces se utiliza un bloque en un grupo de cilindros cercano.

4.4.5 Desfragmentación de discos

Cuando el sistema operativo se instala por primera vez, los programas y archivos que necesita se instalan en forma consecutiva, empezando al principio del disco, cada uno siguiendo directamente del anterior. Todo el espacio libre en el disco está en una sola unidad contigua que va después

de los archivos instalados. Sin embargo, a medida que transcurre el tiempo se crean y eliminan archivos, generalmente el disco se fragmenta mucho, con archivos y huecos esparcidos por todas partes. Como consecuencia, cuando se crea un nuevo archivo, los bloques que se utilizan para éste pueden estar esparcidos por todo el disco, lo cual produce un rendimiento pobre.

El rendimiento se puede restaurar moviendo archivos para hacerlos contiguos y colocando todo el espacio libre (o al menos la mayoría) en una o más regiones contiguas en el disco. Windows tiene un programa llamado *defrag*, el cual realiza esto. Los usuarios de Windows deben ejecutarlo en forma regular.

La desfragmentación funciona mejor en los sistemas de archivos que tienen una buena cantidad de espacio libre en una región continua al final de la partición. Este espacio permite al programa de desfragmentación seleccionar archivos fragmentados cerca del inicio de la partición, copiando todos sus bloques al espacio libre. Esta acción libera un bloque contiguo de espacio cerca del inicio de la partición en la que se pueden colocar los archivos originales u otros en forma contigua. Después, el proceso se puede repetir con el siguiente pedazo de espacio en el disco y así en lo sucesivo.

Algunos archivos no se pueden mover, incluyendo el archivo de paginación, el de hibernación y el registro por bitácora, ya que la administración requerida para ello es más problemática de lo que ayuda. En algunos sistemas, éstas son áreas contiguas de tamaño fijo de todas formas, por lo que no se tienen que desfragmentar. La única vez cuando su falta de movilidad representa un problema es cuando se encuentran cerca del final de la partición y el usuario desea reducir su tamaño. La única manera de resolver este problema es quitar todos los archivos, cambiar el tamaño de la partición y después recrearlos.

Los sistemas de archivos de Linux (en especial ext2 y ext3) por lo general sufren menos por la desfragmentación que los sistemas Windows debido a la forma en que se seleccionan los bloques de disco, por lo que raras veces se requiere una desfragmentación manual.

4.5 EJEMPLOS DE SISTEMAS DE ARCHIVOS

En las siguientes secciones analizaremos varios sistemas de archivos de ejemplo, que varían desde los muy simples hasta algunos más sofisticados. Como los sistemas modernos de UNIX y el sistema de archivos nativo de Windows Vista se cubren en el capítulo acerca de UNIX (capítulo 10) y en el capítulo acerca de Windows Vista (capítulo 11), no cubriremos estos sistemas aquí. Sin embargo, a continuación examinaremos sus predecesores.

4.5.1 Sistemas de archivos de CD-ROM

Como nuestro primer ejemplo de un sistema de archivos, vamos a considerar los sistemas de archivos utilizados en CD-ROMs. Estos sistemas son particularmente simples, debido a que fueron diseñados para medios en los que sólo se puede escribir una vez. Entre otras cosas, por ejemplo, no tienen provisión para llevar el registro de los bloques libres, debido a que en un CD-ROM no se

pueden liberar ni agregar archivos después de fabricar el disco. A continuación analizaremos el tipo principal de sistema de archivos de CD-ROM y dos extensiones del mismo.

Algunos años después de que el CD-ROM hizo su debut, se introdujo el CD-R (CD regrabable). A diferencia del CD-ROM, es posible agregar archivos después del quemado inicial, pero éstos simplemente se adjuntan al final del CD-R. Los archivos nunca se eliminan (aunque el directorio se puede actualizar para ocultar los archivos existentes). Como consecuencia de este sistema de archivos de "sólo adjuntar", no se alteran las propiedades fundamentales. En especial, todo el espacio se encuentra en un trozo contiguo al final del CD.

El sistema de archivos ISO 9660

El estándar más común para los sistemas de archivos de CD-ROM se adoptó como un Estándar Internacional en 1998, bajo el nombre **ISO 9660**. Casi cualquier CD-ROM que se encuentra actualmente en el mercado es compatible con este estándar, algunas veces con las extensiones que analizaremos a continuación. Uno de los objetivos de este estándar era hacer que cada CD-ROM se pudiera leer en todas las computadoras, independientemente del orden de bytes utilizado y del sistema operativo que se emplee. Como consecuencia, se impusieron algunas limitaciones en el sistema de archivos para que los sistemas operativos más débiles que estaban en uso (como MS-DOS) pudieran leerlo.

Los CD-ROMs no tienen cilindros concéntricos como los discos magnéticos. En vez de ello hay una sola espiral continua que contiene los bits en una secuencia lineal (aunque son posibles las búsquedas a través de la espiral). Los bits a lo largo de la espiral se dividen en bloques lógicos (también conocidos como sectores lógicos) de 2352 bytes. Algunos de éstos son para preámbulos, corrección de errores y demás gasto adicional. La porción de carga de cada bloque lógico es de 2048 bytes. Cuando se utilizan para música, los CDs tienen ranuras de introducción, ranuras de salida y espacios entre las pistas, pero éstos no se utilizan para los CD-ROMs de datos. A menudo la posición de un bloque a lo largo de la espiral se expresa en minutos y segundos. Se puede convertir en un número de bloque lineal utilizando el factor de conversión de 1 seg $=$ 75 bloques.

ISO 9660 soporta conjuntos de CD-ROMs con hasta $2^{16} - 1$ CDs en el conjunto. Los CD-ROMs individuales también se pueden particionar en volúmenes lógicos (particiones). Sin embargo, a continuación nos concentraremos en ISO 9660 para un solo CD-ROM sin particiones.

Cada CD-ROM empieza con 16 bloques cuya función no está definida por el estándar ISO 9660. Un fabricante de CD-ROMs podría utilizar esta área para proporcionar un programa de arranque para permitir que la computadora se inicie desde el CD-ROM o para algún otro propósito. A continuación le sigue un bloque que contiene el **descriptor de volumen primario**, que contiene cierta información general acerca del CD-ROM. Esta información incluye el identificador del sistema (32 bytes), el identificador del volumen (32 bytes), el identificador del publicador (128 bytes) y el identificador del preparador de datos (128 bytes). El fabricante puede llenar estos campos de cualquier forma que lo desee, excepto que sólo se pueden utilizar letras mayúsculas, dígitos y un número muy reducido de signos de puntuación para asegurar la compatibilidad entre plataformas.

El descriptor de volumen primario también contiene los nombres de tres archivos, que pueden contener el resumen, el aviso de copyright e información bibliográfica, respectivamente. Además también hay varios números clave presentes, incluyendo el tamaño del bloque lógico (por lo general de 2048, pero se permiten 4096, 8192 y potencias de dos grandes en ciertos casos), el número de bloques en el CD-ROM, las fechas de creación y expiración del CD-ROM. Por último, el descriptor de volumen primario también contiene una entrada de directorio para el directorio raíz, que indica en dónde se puede encontrar en el CD-ROM (es decir, en qué bloque empieza). A partir de este directorio se puede localizar el resto del sistema de archivos.

Además del descriptor de volumen primario, un CD-ROM puede contener un descriptor de volumen suplementario. Contiene información similar a la del primario, pero eso no es de importancia para nosotros aquí.

El directorio raíz, con todos los demás directorios relacionados, consiste de un número variable de entradas, la última de las cuales contiene un bit que la marca como la última. Las mismas entradas de directorio son también de longitud variable. Cada entrada de directorio puede tener de 10 a 12 campos, algunos de los cuales están en ASCII y otros son campos numéricos en binario. Los campos binarios están codificados dos veces, una vez en formato *little endian* (que se utiliza en Pentiums, por ejemplo) y otra vez en formato *big endian* (que se utiliza en SPARCs, por ejemplo). Así, un número de 16 bits utiliza 4 bytes y un número de 32 bits utiliza 8 bytes.

El uso de esta codificación redundante fue necesario para evitar lastimar los sentimientos de cualquiera cuando se desarrolló el estándar. Si el estándar hubiera dictado el formato *little endian*, entonces las personas de empresas cuyos productos tuvieran el formato *big endian* se hubieran sentido como ciudadanos de segunda clase y no hubieran aceptado el estándar. El contenido emocional de un CD-ROM puede entonces cuantificarse y medirse exactamente en kilobytes/hora de espacio desperdiciado.

El formato de una entrada de directorio ISO 9660 se ilustra en la figura 4-30. Como las entradas de directorios tienen longitudes variables, el primer campo es un byte que indica qué tan larga es la entrada. Este byte está definido para tener el bit de mayor orden a la izquierda, para evitar cualquier ambigüedad.

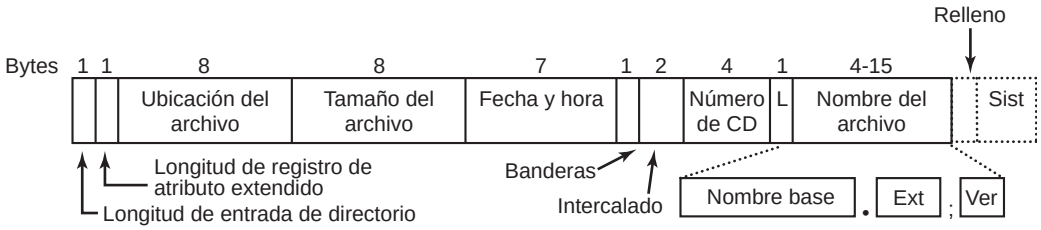

Figura 4-30. La entrada de directorio ISO 9660.

Las entradas de directorio pueden tener de manera opcional atributos extendidos. Si se utiliza esta característica, el segundo byte indica qué tan largos son los atributos extendidos.

A continuación viene el bloque inicial del archivo en sí. Los archivos se almacenan como series contiguas de bloques, por lo que la ubicación de un archivo está completamente especificada por el bloque inicial y el tamaño, que está contenido en el siguiente campo.

La fecha y hora en que se grabó el CD-ROM se almacena en el siguiente campo, con bytes separados para el año, mes, día, hora, minuto, segundo y zona horaria. Los años empiezan a contar desde 1900, lo cual significa que los CD-ROM sufrirán un problema en el año 2156, debido a que el año después del 2155 será 1900. Este problema se podría haber retrasado al definir el origen del tiempo como 1988 (el año en que se adoptó el estándar). Si se hubiera hecho eso, el problema se hubiera pospuesto hasta el 2244. Los 88 años extra son útiles.

El campo *Banderas* contiene unos cuantos bits misceláneos, incluyendo uno para ocultar la entrada en los listados (una característica que se copió de MS-DOS): uno para diferenciar una entrada que es un archivo de una entrada que es un directorio, una para permitir el uso de los atributos extendidos y una para marcar la última entrada en un directorio. También hay unos cuantos bits presentes en este campo, pero no son de importancia aquí. El siguiente campo trata sobre la intercalación de partes de archivos en una manera que no se utiliza en la versión más simple de ISO 9660, por lo que no veremos con más detalle.

El siguiente campo indica en qué CD-ROM se encuentra el archivo. Se permite que una entrada de directorio en un CD-ROM haga referencia a un archivo ubicado en otro CD-ROM en el conjunto. De esta forma, es posible crear un directorio maestro en el primer CD-ROM que liste todos los archivos en todos los CD-ROM del conjunto completo.

El campo marcado como *L* en la figura 4-30 proporciona el tamaño del nombre del archivo en bytes. Va seguido del nombre del archivo en sí. Un nombre de archivo consiste en un nombre base, un punto, una extensión, un punto y coma y un número de versión binario (1 o 2 bytes). El nombre base y la extensión pueden utilizar letras mayúsculas, los dígitos del 0 al 9 y el carácter de guión bajo. Todos los demás caracteres están prohibidos, para asegurar que cada computadora pueda manejar cada nombre de archivo. El nombre base puede ser de hasta ocho caracteres; la extensión puede ser de hasta tres caracteres. Estas opciones se impusieron debido a la necesidad de tener compatibilidad con MS-DOS. Un nombre de archivo puede estar presente en un directorio varias veces, siempre y cuando cada uno tenga un número de versión distinto.

Los últimos dos campos no siempre están presentes. El campo *Relleno* se utiliza para forzar a que cada entrada de directorio sea un número par de bytes, para alinear los campos numéricos de las entradas subsiguientes en límites de 2 bytes. Si se necesita relleno, se utiliza un byte 0. Por último tenemos el campo *Uso del sistema, sist*. Su función y tamaño no están definidos, excepto que debe ser un número par de bytes. Los distintos sistemas lo utilizan en formas diferentes. Por ejemplo, la Macintosh mantiene las banderas Finder aquí.

Las entradas dentro de un directorio se listan en orden alfabético, excepto por las primeras dos. La primera entrada es para el directorio en sí. La segunda es para su padre. En este aspecto, estas son similares a las entradas de directorio . y .. de UNIX. Los archivos en sí no necesitan estar en orden de directorio.

No hay un límite explícito en cuanto al número de entradas en un directorio. Sin embargo, hay un límite en cuanto a la profundidad de anidamiento. La máxima profundidad de anidamiento de directorios es ocho. Este límite se estableció en forma arbitraria para simplificar las implementaciones.

ISO 9660 define lo que se conoce como tres niveles. El nivel 1 es el más restrictivo y especifica que los nombres de archivos están limitados a 8 + 3 caracteres; también requiere que todos los archivos sean contiguos, como hemos visto antes. Además, especifica que los nombres de directorio deben estar limitados a ocho caracteres sin extensiones. El uso de este nivel maximiza las probabilidades de que un CD-ROM se pueda leer en todas las computadoras.

El nivel 2 reduce la restricción de longitud. Permite que los archivos y directorios tengan nombres de hasta 31 caracteres, pero siguen teniendo el mismo conjunto de caracteres.

El nivel 3 utiliza los mismos límites de nombres que el nivel 2, pero reduce parcialmente la restricción de que los archivos tienen que ser contiguos. Con este nivel, un archivo puede consistir de varias secciones (fragmentos), cada una de las cuales es una serie contigua de bloques. La misma serie puede ocurrir varias veces en un archivo y también puede ocurrir en dos o más archivos. Si se repiten grandes trozos de datos en varios archivos, el nivel 3 ofrece cierta optimización del espacio al no requerir que los datos estén presentes varias veces.

Extensiones Rock Ridge

Como hemos visto, ISO 9660 es altamente restrictivo en varias formas. Poco después de que salió al mercado, las personas en la comunidad UNIX empezaron a trabajar en una extensión para que fuera posible representar los sistemas de archivos de UNIX en un CD-ROM. Estas extensiones se llamaron Rock Ridge, en honor a una ciudad de la película *Blazing Saddles* de Gene Wilder, probablemente debido a que a uno de los miembros del comité le gustó el filme.

Las extensiones utilizan el campo *Uso del sistema* para que los CD-ROMs Rock Ridge se puedan leer en cualquier computadora. Todos los demás campos retienen su significado normal de ISO 9660. Cualquier sistema que no esté al tanto de las extensiones Rock Ridge sólo las ignora y ve un CD-ROM normal.

Las extensiones se dividen en los siguientes campos:

1. PX – Atributos POSIX.
2. PN – Números de dispositivos mayores y menores.
3. SL – Vínculo simbólico.
4. NM – Nombre alternativo.
5. CL – Ubicación del hijo.
6. PL – Ubicación del padre.
7. RE – Reubicación.
8. TF – Estampado de tiempo.

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

Para permitir que los dispositivos puros se representen en un CD-ROM, está presente el campo *PN*, el cual contiene los números de dispositivos mayores y menores asociados con el archivo. De esta forma, el contenido del directorio */dev* se puede escribir en un CD-ROM y después reconstruirse de manera correcta en el sistema de destino.

El campo *SL* es para los vínculos simbólicos. Permite que un archivo en un sistema de archivos haga referencia a un archivo en un sistema de archivos distinto.

Tal vez el campo más importante sea *NM*. Este campo permite asociar un segundo nombre con el archivo. Este nombre no está sujeto a las restricciones del conjunto de caracteres o de longitud de ISO 9660, con lo cual es posible expresar nombres de archivos de UNIX arbitrarios en un CD-ROM.

Los siguientes tres campos se utilizan en conjunto para sobrepasar el límite de ISO 9660 de los directorios que sólo pueden tener ocho niveles de anidamiento. Mediante su uso es posible especificar que un directorio se va a reubicar e indicar cuál es su posición en la jerarquía. Es en efecto una manera de solucionar el límite de profundidad artificial.

Por último, el campo *TF* contiene los tres estampados de tiempo incluidos en cada nodo-i de UNIX, a saber la hora en que se creó el archivo, la hora en que se modificó por última vez, y la hora en la que se accedió por última vez. En conjunto, estas extensiones hacen que sea posible copiar un sistema de archivos UNIX a un CD-ROM y después restaurarlo de manera correcta en un sistema diferente.

Extensiones Joliet

La comunidad de UNIX no fue el único grupo que quería una manera de extender ISO 9660. A Microsoft también le pareció demasiado restrictivo (aunque fue el propio MS-DOS de Microsoft quien causó la mayor parte de las restricciones en primer lugar). Por lo tanto, Microsoft inventó ciertas extensiones, a las cuales llamó **Joliet**. Estas extensiones estaban diseñadas para permitir que los sistemas de archivos de Windows se copiaran a CD-ROM y después se restauraran, precisamente en la misma forma que se diseñó Rock Ridge para UNIX. Casi todos los programas que se ejecutan en Windows y utilizan CD-ROMs tienen soporte para Joliet, incluyendo los programas que queman CDs grabables. Por lo general, estos programas ofrecen una elección entre los diversos niveles de ISO 9660 y Joliet.

Las principales extensiones que proporciona Joliet son:

1. Nombres de archivos largos.
2. Conjunto de caracteres Unicode.
3. Más de ocho niveles de anidamiento de directorios.
4. Nombres de directorios con extensiones.

La primera extensión permite nombres de archivos de hasta 64 caracteres. La segunda extensión permite el uso del conjunto de caracteres Unicode para los nombres de archivos. Esta extensión es importante para el software destinado a los países que no utilizan el alfabeto en latín, como Japón, Israel y Grecia. Como los caracteres Unicode son de 2 bytes, el nombre de archivo más grande en Joliet ocupa 128 bytes.

Al igual que Rock Ridge, Joliet elimina la limitación en el anidamiento de directorios. Los directorios se pueden anidar con tantos niveles como sea necesario. Por último, los nombres de los directorios pueden tener extensiones. No está claro por qué se incluyó esta extensión, ya que raras veces los directorios de Windows utilizan extensiones, pero tal vez algún día lo hagan.

4.5.2 El sistema de archivos MS-DOS

El sistema de archivos MS-DOS es uno de los primeros sistemas que se incluyeron con la IBM PC. Era el sistema principal hasta Windows 98 y Windows ME. Aún cuenta con soporte en Windows 2000, Windows XP y Windows Vista, aunque ya no es estándar en las nuevas PCs, excepto por los discos flexibles. Sin embargo, este sistema y una extensión de él (FAT-32) están teniendo mucho uso en varios sistemas incrustados. La mayoría de las cámaras digitales lo utilizan. Muchos reproductores de MP3 lo utilizan de manera exclusiva. El popular iPod de Apple lo utiliza como el sistema de archivos predeterminado, aunque los hackers conocedores pueden volver a dar formato al iPod e instalar un sistema de archivos distinto. Así, el número de dispositivos electrónicos que utilizan el sistema de archivos MS-DOS es inmensamente grande en la actualidad, en comparación con cualquier momento en el pasado, y sin duda es mucho mayor que el número de dispositivos que utilizan el sistema de archivos NTFS más moderno. Por esta única razón, vale la pena analizarlo con cierto detalle.

Para leer un archivo, un programa de MS-DOS primero debe realizar una llamada al sistema open para obtener un manejador para el archivo. La llamada al sistema open especifica una ruta, que puede ser absoluta o relativa al directorio de trabajo actual. Se realiza una búsqueda de la ruta, componente por componente, hasta que se localiza el directorio final y se lee en la memoria. Después se busca el archivo que se desea abrir.

Aunque los directorios de MS-DOS tienen tamaños variables, utilizan una entrada de directorio de tamaño fijo de 32 bytes. El formato de una entrada de directorio de MS-DOS se muestra en la figura 4-31. Contiene el nombre de archivo, los atributos, la fecha y hora de creación, el bloque inicial y el tamaño exacto del archivo. Los nombres de archivo menores de 8 + 3 caracteres se justifican a la izquierda y se rellenan con espacios a la derecha, en cada campo por separado. El campo *Atributos* es nuevo y contiene bits para indicar que un archivo es de sólo lectura, que necesita archivarse, está oculto o que es un archivo del sistema. No se puede escribir en los archivos de sólo lectura. Esto es para protegerlos de daños accidentales. El bit archivado no tiene una función del sistema operativo actual (es decir, MS-DOS no lo examina ni lo establece). La intención es permitir que los programas de archivos a nivel de usuario lo desactiven al archivar un archivo y que otros programas lo activen cuando modifiquen un archivo. De esta manera, un programa de respaldo sólo tiene que examinar este bit de atributo en cada archivo para ver cuáles archivos debe respaldar. El bit oculto se puede establecer para evitar que un archivo aparezca en los listados de directorios. Su principal uso es para evitar confundir a los usuarios novatos con los archivos que tal vez no entiendan. Por último, el bit del sistema también oculta archivos. Además, los archivos del sistema no pueden eliminarse de manera accidental mediante el uso del comando *del*. Los componentes principales de MS-DOS tienen activado este bit.

La entrada de directorio también contiene la fecha y hora de creación del archivo o su última modificación. La hora sólo tiene una precisión de hasta ±2 segundos, debido a que se almacena en un campo de 2 bytes, que sólo puede almacenar 65,536 valores únicos (un día contiene 86,400 segundos).

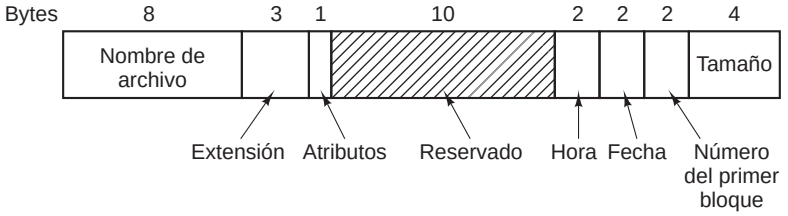

Figura 4-31. La entrada de directorio de MS-DOS.

El campo del tiempo se subdivide en segundos (5 bits), minutos (6 bits) y horas (5 bits). La fecha se cuenta en días usando tres subcampos: día (5 bits), mes (4 bits) y año desde 1980 (7 bits). Con un número de 7 bits para el año y la hora que empiezan en 1980, el mayor año que puede expresarse es 2107. Por lo tanto, MS-DOS tiene integrado un problema para el año 2108. Para evitar una catástrofe, los usuarios de MS-DOS deben empezar con la conformidad con el año 2108 lo más pronto posible. Si MS-DOS hubiera utilizado los campos combinados de fecha y hora como un contador de segundos de 32 bits, podría haber representado cada segundo con exactitud y hubiera retrasado su catástrofe hasta el 2116.

MS-DOS almacena el tamaño de archivo como un número de 32 bits, por lo que en teoría los archivos pueden ser tan grandes como 4 GB. Sin embargo, otros límites (que se describen a continuación) restringen el tamaño máximo de un archivo a 2 GB o menos. De manera sorprendente, no se utiliza una gran parte de la entrada (10 bytes).

MS-DOS lleva la cuenta de los bloques de los archivos mediante una tabla de asignación de archivos (FAT) en la memoria principal. La entrada de directorio contiene el número del primer bloque del archivo. Este número se utiliza como un índice en una FAT con entradas de 64 KB en la memoria principal. Al seguir la cadena, se pueden encontrar todos los bloques. La operación de la FAT se ilustra en la figura 4-12.

El sistema de archivos FAT viene en tres versiones: FAT-12, FAT-16 y FAT-32, dependiendo de cuántos bits contenga una dirección de disco. En realidad, FAT-32 es algo así como un término equivocado, ya que sólo se utilizan los 28 bits de menor orden de las direcciones de disco. Debería haberse llamado FAT-28, pero las potencias de dos se oyen mucho mejor.

Para todas las FATs, el bloque de disco se puede establecer en algún múltiplo de 512 bytes (posiblemente distinto para cada partición), donde el conjunto de tamaños de bloque permitidos (llamados **tamaños de grupo** por Microsoft) es distinto para cada variante. La primera versión de MS-DOS utilizaba FAT-12 con bloques de 512 bytes, para proporcionar un tamaño de partición máximo de $2^{12} \times 512$ bytes (en realidad sólo son 4086 x 512 bytes, debido a que 10 de las direcciones de disco se utilizaban como marcadores especiales, como el de fin de archivo, el de bloque defectuoso, etc.). Con estos parámetros, el tamaño máximo de partición de disco era de aproximadamente 2 MB y el tamaño de la tabla FAT en memoria era de 4096 entradas de 2 bytes cada una. Utilizar una entrada de tabla de 12 bits hubiera sido demasiado lento.

Este sistema funcionaba bien para los discos flexibles, pero cuando salieron al mercado los discos duros se convirtió en un problema. Microsoft resolvió el problema al permitir tamaños de bloque adicionales de 1 KB, 2 KB y 4 KB. Este cambio preservó la estructura y el tamaño de la tabla FAT-12, pero permitía particiones de disco de hasta 16 MB.

Ya que MS-DOS soportaba cuatro particiones de disco por cada unidad de disco, el nuevo sistema de archivos FAT-12 podía operar discos de hasta 64 MB. Más allá de esa capacidad, algo podría fallar. Lo que ocurrió fue la introducción de FAT-16, con apuntadores de disco de 16 bits. Además se permitieron tamaños de bloque de 8 KB, 16 KB y 32 KB (32,768 es la potencia más grande de dos que se puede representar en 16 bits). La tabla FAT-16 ahora ocupaba 128 KB de memoria principal todo el tiempo, pero con las memorias más grandes que para entonces había disponibles, se utilizó ampliamente y reemplazó con rapidez el sistema de archivos FAT-12. La partición de disco más grande que podía soportar FAT-16 era de 2 GB (64K entradas de 32 KB cada una) y el disco más grande, 8 GB, a saber de cuatro particiones de 2 GB cada una.

Para las cartas comerciales este límite no es un problema, pero para almacenar video digital utilizando el estándar DV, un archivo de 2 GB apenas si contiene 9 minutos de video. Como consecuencia del hecho de que un disco de PC sólo puede soportar cuatro particiones, el video más grande que se puede almacenar en un disco es de aproximadamente 36 minutos, sin importar qué tan grande sea el disco. Este límite también significa que el video más grande que se puede editar en línea es de menos de 18 minutos, ya que se necesitan archivos de entrada y de salida.

Empezando con la segunda versión de Windows 95, se introdujo el sistema de archivos FAT-32 con sus direcciones de disco de 28 bits, donde la versión de MS-DOS que operaba bajo Windows 95 se adaptó para dar soporte a FAT-32. En este sistema las particiones podrían ser en teoría de $2^{28} \times 2^{15}$ bytes, pero en realidad están limitadas a 2 TB (2048 GB) debido a que el sistema lleva de manera interna la cuenta de los tamaños de las particiones en sectores de 512 bytes utilizando un número de 32 bits, y $2^9 \times 2^{32}$ es 2 TB. El tamaño máximo de partición para los diversos tamaños de bloque y los tres tipos de FAT se muestran en la figura 4-32.

Tamaño de bloque	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 4-32. Tamaño máximo de partición para los distintos tamaños de bloque. Los cuadros vacíos representan combinaciones prohibidas.

Además de soportar discos más grandes, el sistema de archivos FAT-32 tiene otras dos ventajas en comparación con FAT-16. En primer lugar, un disco de 8 GB que utiliza FAT-32 puede ser de una sola partición. Si utiliza FAT-16 tiene que tener cuatro particiones, lo cual aparece para el usuario de Windows como las unidades de disco lógicas *C:*, *D:*, *E:* y *F:*. Depende del usuario decidir qué archivo colocar en cuál unidad y llevar el registro de dónde se encuentran las cosas.

La otra ventaja de FAT-32 sobre FAT-16 es que para una partición de disco de cierto tamaño, se puede utilizar un tamaño de bloque más pequeño. Por ejemplo, para una partición de disco de 2 GB,

FAT-16 debe utilizar bloques de 32 KB; de lo contrario, con solo 64K direcciones de disco disponibles, no podrá cubrir toda la partición. Por el contrario, FAT-32 puede utilizar, por ejemplo, bloques de 4 KB para una partición de disco de 2 GB. La ventaja del tamaño de bloque más pequeño es que la mayoría de los archivos son mucho menores de 32 KB. Si el tamaño de bloque es de 32 KB, un archivo de 10 bytes ocupa 32 KB de espacio en el disco. Si el archivo promedio es, por decir, de 8 KB, entonces con un bloque de 32 KB se desperdiciarán $^3/_4$ partes del disco, lo cual no es una manera muy eficiente de utilizarlo. Con un archivo de 8 KB y un bloque de 4 KB no hay desperdicio del disco, pero el precio a pagar es que la FAT ocupa más RAM. Con un bloque de 4 KB y una partición de disco de 2 GB hay 512K bloques, por lo que la FAT debe tener 512K entradas en la memoria (que ocupan 2 MB de RAM).

MS-DOS utiliza la FAT para llevar la cuenta de los bloques de disco libres. Cualquier bloque que no esté asignado en un momento dado se marca con un código especial. Cuando MS-DOS necesita un nuevo bloque de disco, busca en la FAT una entrada que contenga este código. Por lo tanto, no se requiere un mapa de bits o una lista de bloques libres.

4.5.3 El sistema de archivos V7 de UNIX

Incluso las primeras versiones de UNIX tenian un sistema de archivos multiusuario bastante sofisticado, ya que se derivaba de MULTICS. A continuación analizaremos el sistema de archivos V7, diseñado para la PDP-11 que famoso hizo a UNIX. En el capítulo 10 examinaremos un sistema de archivos de UNIX moderno, en el contexto de Linux.

El sistema de archivos está en la forma de un árbol que empieza en el directorio raíz, con la adición de vínculos para formar un gráfico acíclico dirigido. Los nombres de archivos tienen hasta 14 caracteres y pueden contener cualquier carácter ASCII excepto / (debido a que es el separador entre los componentes en una ruta) y NUL (debido a que se utiliza para rellenar los nombres menores de 14 caracteres). NUL tiene el valor numérico de 0.

Una entrada de directorio de UNIX contiene una entrada para cada archivo en ese directorio. Cada entrada es en extremo simple, ya que UNIX utiliza el esquema de nodos-i ilustrado en la figura 4-13. Una entrada de directorio sólo contiene dos campos: el nombre del archivo (14 bytes) y el número del nodo-i para ese archivo (2 bytes), como se muestra en la figura 4-33. Estos parámetros limitan el número de archivos por cada sistema de archivos a 64 K.

Al igual que el nodo-i de la figura 4-13, los nodos-i de UNIX contienen ciertos atributos, los cuales contienen el tamaño del archivo, tres tiempos (hora de creación, de último acceso y de última modificación), el propietario, el grupo, información de protección y una cuenta del número de entradas de directorio que apuntan al nodo-i. El último campo se necesita debido a los vínculos. Cada vez que se crea un nuevo vínculo a un nodo-i, la cuenta en ese nodo-i se incrementa. Cuando se elimina un vínculo, la cuenta se decrementa. Cuando llega a 0, el nodo-i se reclama y los bloques de disco se devuelven a la lista de bloques libres.

Para llevar la cuenta de los bloques de disco se utiliza una generalización de la figura 4-13, para poder manejar archivos muy grandes. Las primeras 10 direcciones de disco se almacenan en el mismo nodo-i, por lo que para los archivos pequeños toda la información necesaria se encuentra

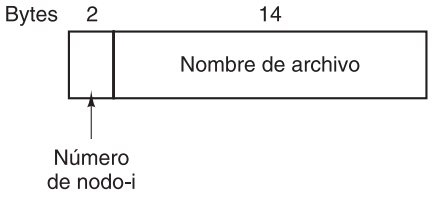

Figura 4-33. Una entrada de directorio de UNIX V7.

justo en el nodo-i, que se obtiene del disco a la memoria principal al momento de abrir el archivo. Para archivos un poco más grandes, una de las direcciones en el nodo-i es la dirección de un bloque de disco llamado **bloque indirecto sencillo**. Este bloque contiene direcciones de disco adicionales. Si esto no es suficiente, hay otra dirección en el nodo-i, llamada **bloque indirecto doble**, que contiene la dirección de un bloque que contiene una lista de bloques indirectos sencillos. Cada uno de estos bloques indirectos sencillos apunta a unos cuantos cientos de bloques de datos. Si esto aún no es suficiente, también se puede utilizar un **bloque indirecto triple**. El panorama completo se muestra en la figura 4-34.

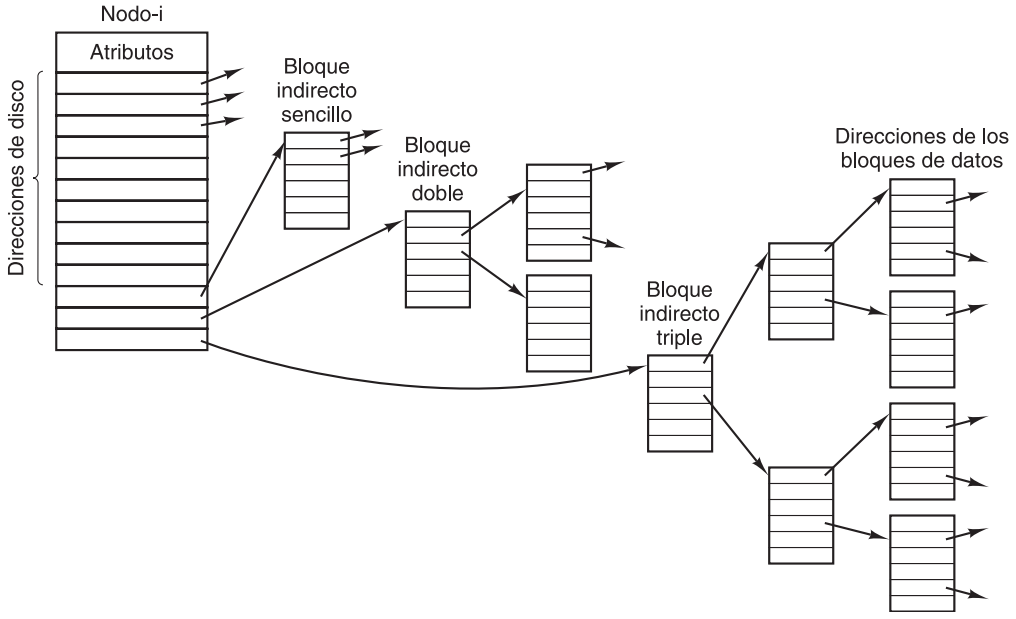

Figura 4-34. Un nodo-i de UNIX.

Cuando se abre un archivo, el sistema de archivos debe tomar el nombre de archivo suministrado y localizar sus bloques de disco. Ahora consideremos cómo se busca el nombre de la ruta

/usr/ast/mbox. Utilizaremos UNIX como un ejemplo, pero el algoritmo es básicamente el mismo para todos los sistemas de directorios jerárquicos. En primer lugar, el sistema de archivos localiza el directorio raíz. En UNIX, su nodo-i se localiza en un lugar fijo en el disco. De este nodo-i localiza el directorio raíz, que puede estar en cualquier parte del disco, pero digamos que está en el bloque 1.

Después lee el directorio raíz y busca el primer componente de la ruta, *usr*, en el directorio raíz para encontrar el número de nodo-i del archivo */usr*. La acción de localizar un nodo-i a partir de su número es simple, ya que cada uno tiene una ubicación fija en el disco. A partir de este nodo-i, el sistema localiza el directorio para */usr* y busca el siguiente componente, *ast*, en él. Cuando encuentra la entrada para *ast*, tiene el nodo-i para el directorio */usr/ast*. A partir de este nodo-i puede buscar *mbox* en el mismo directorio. Después, el nodo-i para este archivo se lee en memoria y se mantiene ahí hasta que se cierra el archivo. El proceso de búsqueda se ilustra en la figura 4-35.

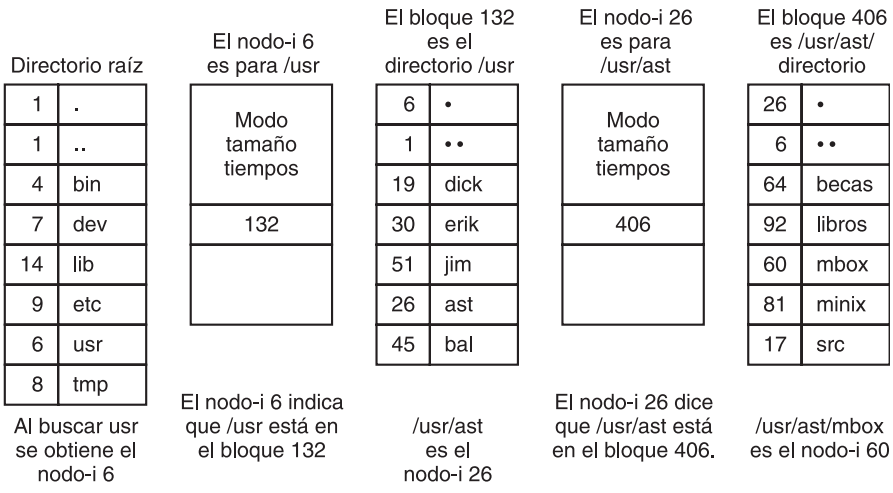

Figura 4-35. Los pasos para buscar */usr/ast/mbox*.

Los nombres de rutas relativas se buscan de la misma forma que las absolutas, sólo que se empieza desde el directorio de trabajo en vez de empezar del directorio raíz. Cada directorio tiene entradas para . y.., que se colocan ahí cuando se crea el directorio. La entrada . tiene el número del nodo-i para el directorio actual y la entrada .. tiene el número del nodo-i para el directorio padre. Así, un procedimiento que busca *../dick/prog.c* simplemente busca .. en el directorio de trabajo, encuentra el número del nodo-i para el directorio padre y busca *dick* en ese directorio. No se necesita un mecanismo especial para manejar estos nombres. En cuanto lo que al sistema de directorios concierne, sólo son cadenas ASCII ordinarias, igual que cualquier otro nombre. El único truco aquí es que .. en el directorio raíz se apunta a sí mismo.

4.6 INVESTIGACIÓN ACERCA DE LOS SISTEMAS DE ARCHIVOS

Los sistemas de archivos siempre han atraído más investigación que las demás partes del sistema operativo y éste sigue siendo el caso. Aunque los sistemas de archivos estándar están muy bien comprendidos, aún hay un poco de investigación acerca de cómo optimizar la administración de la caché de búfer (Burnett y colaboradores, 2002; Ding y colaboradores, 2007; Gnaidy y colaboradores, 2004; Kroeger y Long, 2001; Pai y colaboradores, 2000; y Zhou y colaboradores, 2001). Se está realizando trabajo acerca de los nuevos tipos de sistemas de archivos, como los sistemas de archivos a nivel de usuario (Mazières, 2001), los sistemas de archivos flash (Gal y colaboradores, 2005), los sistemas de archivos por bitácora (Prabhakaran y colaboradores, 2005; y Stein y colaboradores, 2001), los sistemas de archivos con control de versiones (Cornell y colaboradores, 2004), los sistemas de archivos de igual a igual (Muthitacharoen y colaboradores, 2002) y otros. El sistema de archivos de Google es también inusual debido a su gran tolerancia a errores (Ghemawat y colaboradores, 2003). También son de interés las distintas formas de buscar cosas en los sistemas de archivos (Padioleau y Ridoux, 2003).

Otra área que ha estado obteniendo atención es la procedencia: llevar un registro del historial de los datos, incluyendo de dónde provinieron, quién es el propietario y cómo se han transformado (Muniswarmy-Reddy y colaboradores, 2006; y Shah y colaboradores, 2007). Esta información se puede utilizar de varias formas. La realización de respaldos también está recibiendo algo de atención (Cox y colaboradores, 2002; y Rycroft, 2006), al igual que el tema relacionado de la recuperación (Keeton y colaboradores, 2006). Algo también relacionado con los respaldos es el proceso de mantener los datos disponibles y útiles durante décadas (Baker y colaboradores, 2006; Maniatis y colaboradores, 2003). La confiabilidad y la seguridad también están muy lejos de ser problemas solucionados (Greenan y Miller, 2006; Wires y Feeley, 2007; Wright y colaboradores, 2007; y Yang y colaboradores, 2006). Y por último, el rendimiento siempre ha sido un tema de investigación y lo sigue siendo (Caudill y Gavrikovska, 2006; Chiang y Huang, 2007; Stein, 2006; Wang y colaboradores, 2006a; y Zhang y Ghose, 2007).

4.7 RESUMEN

Visto desde el exterior, un sistema de archivos es una colección de archivos y directorios, más las operaciones que se realizan con ellos. Los archivos se pueden leer y escribir, los directorios se pueden crear y destruir, y los archivos se pueden mover de un directorio a otro. La mayoría de los sistemas de archivos modernos soportan un sistema de directorios jerárquico en el cual los directorios pueden tener subdirectorios y éstos pueden tener subdirectorios en forma infinita.

Visto desde el interior, un sistema de archivos tiene una apariencia distinta. Los diseñadores del sistema de archivos se tienen que preocupar acerca de la forma en que se asigna el almacenamiento y cómo el sistema lleva el registro de qué bloque va con cuál archivo. Las posibilidades incluyen archivos contiguos, listas enlazadas (ligadas), tablas de asignación de archivos y nodos-i. Los distintos sistemas tienen diferentes estructuras de directorios. Los atributos pueden ir en los directorios o en alguna otra parte (por ejemplo, en un nodo-i). El espacio en el disco se puede administrar

mediante el uso de listas de bloques libres o mapas de bits. La confiabilidad del sistema aumenta al realizar vaciados incrementales y tener un programa que pueda reparar sistemas de archivos enfermos. El rendimiento del sistema de archivos es importante y puede mejorarse de varias formas, incluyendo el uso de cachés, la lectura adelantada y la colocación cuidadosa de los bloques de un archivo cerca unos de otros. Los sistemas de archivos estructurados por registro también mejoran el rendimiento, al realizar escrituras en unidades grandes.

Algunos ejemplos de sistemas de archivos son ISO 9660, MS-DOS y UNIX. Éstos difieren en varias formas, incluyendo la manera en que llevan el registro de qué bloques van con cuál archivo, la estructura de los directorios y la administración del espacio libre en el disco.

PROBLEMAS

1. En los primeros sistemas UNIX, los archivos ejecutables (archivos *a.out*) empezaban con un número mágico muy específico, no uno elegido al azar. Estos archivos empezaban con un encabezado, seguido de los segmentos de texto y de datos. ¿Por qué cree usted que se eligió un número muy específico para los archivos ejecutables, mientras que otros tipos de archivos tenían un número mágico más o menos aleatorio como la primera palabra?

2. En la figura 4-4, uno de los atributos es la longitud del registro. ¿Por qué se preocupa el sistema operativo por esto?

3. ¿Es absolutamente esencial la llamada al sistema `open` en UNIX? ¿Cuáles serían las consecuencias de no tenerla?

4. Los sistemas que soportan archivos secuenciales siempre tienen una operación para rebobinar los archivos. ¿Los sistemas que soportan archivos de acceso aleatorio la necesitan también?

5. Algunos sistemas operativos proporcionan una llamada al sistema `rename` para dar a un archivo un nuevo nombre. ¿Hay acaso alguna diferencia entre utilizar esta llamada para cambiar el nombre de un archivo y sólo copiar el archivo a uno nuevo con el nuevo nombre, eliminando después el archivo anterior?

6. En algunos sistemas es posible asociar parte de un archivo a la memoria. ¿Qué restricciones deben imponer dichos sistemas? ¿Cómo se implementa esta asociación parcial?

7. Un sistema operativo simple sólo soporta un directorio, pero permite que ese directorio tenga arbitrariamente muchos archivos con nombres de archivos arbitrariamente largos. ¿Puede simularse algo aproximado a un sistema de archivos jerárquico? ¿Cómo?

8. En UNIX y Windows, el acceso aleatorio se realiza al tener una llamada especial al sistema que mueve el apuntador de la "posición actual" asociado con un archivo a un byte específico en el archivo. Proponga una manera alternativa de realizar un acceso aleatorio sin tener esta llamada al sistema.

9. Considere el árbol de directorios de la figura 4-8. Si */usr/jim* es el directorio de trabajo, ¿cuál es el nombre de ruta absoluto para el archivo cuyo nombre de ruta relativo es *../ast/x*?

10. La asignación contigua de archivos produce la fragmentación del disco, como se menciona en el texto, debido a que cierto espacio en el último bloque del disco se desperdiciará en los archivos cuya

longitud no sea un número entero de bloques. ¿Es esta fragmentación interna o externa? Haga una analogía con algo que se haya descrito en el capítulo anterior.

11. Una manera de utilizar la asignación contigua del disco y no sufrir de huecos es compactar el disco cada vez que se elimina un archivo. Como todos los archivos son contiguos, para copiar un archivo se requiere una búsqueda y un retraso rotacional para leerlo, seguidos de la transferencia a toda velocidad. Para escribir de vuelta el archivo se requiere el mismo trabajo. Suponiendo un tiempo de búsqueda de 5 mseg, un retraso rotacional de 4 mseg, una velocidad de transferencia de 8 MB/seg y un tamaño de archivo promedio de 8 KB, ¿cuánto tiempo se requiere para leer un archivo en la memoria principal y después escribirlo de vuelta al disco en una nueva ubicación? Utilizando estos números, ¿cuánto tiempo se requeriría para compactar la mitad de un disco de 16 GB?

12. A luz de la respuesta de la pregunta anterior, ¿tiene algún sentido compactar el disco?

13. Algunos dispositivos digitales para el consumidor necesitan almacenar datos, por ejemplo como archivos. Nombre un dispositivo moderno que requiera almacenamiento de archivos y para el que la asignación contigua sería una excelente idea.

14. ¿Cómo implementa MS-DOS el acceso aleatorio a los archivos?

15. Considere el nodo-i que se muestra en la figura 4-13. Si contiene 10 direcciones directas de 4 bytes cada una y todos los bloques de disco son de 1024 KB, ¿cuál es el archivo más grande posible?

16. Se ha sugerido que la eficiencia se podría mejorar y el espacio en disco se podría ahorrar al almacenar los datos de un archivo corto dentro del nodo-i. Para el nodo-i de la figura 4-13, ¿cuántos bytes de datos podrían almacenarse dentro del nodo-i?

17. Dos estudiantes de ciencias computacionales, Carolyn y Elinor, están hablando acerca de los nodos-i. Carolyn sostiene que el tamaño de las memorias ha aumentado tanto y se han vuelto tan económicas que cuando se abre un archivo, es más simple y rápido obtener una nueva copia del nodo-i y colocarla en la tabla de nodos-i, en lugar de buscar en toda la tabla para ver si ya está ahí. Elinor está en desacuerdo. ¿Quién tiene la razón?

18. Nombre una ventaja que tienen los vínculos duros (las ligas duras) sobre los vínculos simbólicos (las ligas simbólicas) y una ventaja que tienen los vínculos simbólicos sobre los vínculos duros.

19. El espacio libre en el disco se puede contabilizar mediante el uso de una lista de bloques libres o un mapa de bits. Las direcciones de disco requieren D bits. Para un disco con B bloques, F de los cuales son libres, indique la condición bajo la cual la lista de bloques libres utiliza menos espacio que el mapa de bits. Si D tiene el valor de 16 bits, exprese su respuesta como un porcentaje del espacio en el disco que debe estar libre.

20. El inicio de un mapa de bits de espacio libre tiene la siguiente apariencia después de que se da formato por primera vez a la partición de disco: 1000 0000 0000 0000 (el primer bloque es utilizado por el directorio raíz). El sistema siempre busca bloques libres empezando en el bloque de menor numeración, por lo que después de escribir el archivo *A*, que utiliza seis bloques, el mapa de bits se ve así: 1111 1110 0000 0000. Muestre el mapa de bits después de cada una de las siguientes acciones adicionales:

(a) Se escribe el archivo *B*, utilizando cinco bloques
(b) Se elimina el archivo *A*
(c) Se escribe el archivo *C*, utilizando ocho bloques
(d) Se elimina el archivo *B*.

21. ¿Qué ocurriría si el mapa de bits o la lista de bloques libres que contiene la información acerca de los bloques de disco libres se perdiera por completo debido a una falla? ¿Hay alguna forma de recuperarse de este desastre o definitivamente hay que decir adiós al disco duro? Analice sus respuestas para los sistemas de archivos UNIX y FAT-16 por separado.

22. El trabajo nocturno de Oliver Owl en el centro de cómputo de la universidad es cambiar las cintas que se utilizan para los respaldos de datos nocturnos. Mientras espera a que se complete cada cinta, trabaja en la escritura de su tesis que demuestra que las obras de Shakespeare fueron escritas por visitantes extraterrestres. Su procesador de texto se ejecuta en el sistema que se está respaldando, ya que es el único que tienen. ¿Hay algún problema con este arreglo?

23. En el texto analizamos con cierto detalle la forma de realizar vaciados incrementales. En Windows es fácil saber cuándo vaciar un archivo, debido a que cada archivo tiene un bit para archivar. Este bit no está en UNIX. ¿Cómo saben los programas de respaldo de UNIX qué archivos vaciar?

24. Suponga que el archivo 21 en la figura 4-25 no se modificó desde la última vez que hubo un vaciado. ¿En qué forma deberían ser distintos los cuatro mapas de bits de la figura 4-26?

25. Se ha sugerido que la primera parte de cada archivo de UNIX debe mantenerse en el mismo bloque de disco que su nodo-i. ¿Qué beneficio traería esto?

26. Considere la figura 4-27. ¿Es posible que para cierto número de bloque específico los contadores en *ambas* listas tengan el valor de 2? ¿Cómo debería corregirse este problema?

27. El rendimiento de un sistema de archivos depende de la proporción de aciertos de la caché (fracción de bloques encontrados en la caché). Si se requiere 1 mseg para satisfacer una solicitud de la caché, pero 40 mseg para satisfacer una solicitud si se necesita una lectura de disco, proporcione una fórmula para el tiempo promedio requerido para satisfacer una solicitud si la proporción de aciertos es *h*. Grafique esta función para los valores de *h* que varían de 0 hasta 1.0.

28. Considere la idea detrás de la figura 4-21, pero ahora para un disco con un tiempo de búsqueda promedio de 8 mseg, una velocidad rotacional de 15,000 rpm y 262,144 bytes por pista. ¿Cuáles son las velocidades de datos para los tamaños de bloque de 1 KB, 2 KB y 4 KB, respectivamente?

29. Cierto sistema de archivos utiliza bloques de disco de 2 KB. El tamaño de archivo promedio es de 1 KB. Si todos los archivos fueran exactamente de 1 KB, ¿qué fracción del espacio en el disco se desperdiciaría? ¿Piensa usted que el desperdicio de un sistema de archivos real será mayor que este número o menor? Explique su respuesta.

30. La tabla FAT-16 de MS-DOS contiene 64K entradas. Suponga que uno de los bits se necesita para algún otro propósito, y que la tabla contiene exactamente 32,768 entradas en vez de 64 K. Sin ningún otro cambio, ¿cuál sería el archivo de MS-DOS más grande bajo esta condición?

31. Los archivos en MS-DOS tienen que competir por el espacio en la tabla FAT-16 en memoria. Si un archivo utiliza *k* entradas, es decir, *k* entradas que no están disponibles para ningún otro archivo, ¿qué restricción impone esto sobre la longitud total de todos los archivos combinados?

32. Un sistema de archivos UNIX tiene bloques de 1 KB y direcciones de disco de 4 bytes. ¿Cuál es el tamaño de archivo máximo si los nodos-i contienen 10 entradas directas y una entrada indirecta sencilla, una doble y una triple?

33. ¿Cuántas operaciones de disco se necesitan para obtener el nodo-i para el archivo */usr/ast/cursos/so-/folleto.t*? Suponga que el nodo-i para el directorio raíz está en la memoria, pero no hay nada más a

lo largo de la ruta en memoria. Suponga también que todos los directorios caben en un bloque de disco.

34. En muchos sistemas UNIX, los nodos-i se mantienen al inicio del disco. Un diseño alternativo es asignar un nodo-i cuando se crea un archivo y colocar el nodo-i al inicio del primer bloque del archivo. Hable sobre las ventajas y desventajas de esta alternativa.

35. Escriba un programa que invierta los bytes de un archivo, de manera que el último byte sea ahora el primero y el primero sea ahora el último. Debe funcionar con un archivo arbitrariamente largo, pero trate de hacerlo eficiente de una manera razonable.

36. Escriba un programa que inicie en un directorio dado y recorra el árbol de archivos desde ese punto hacia abajo, registrando los tamaños de todos los archivos que encuentre. Cuando termine, deberá imprimir un histograma de los tamaños de archivos, utilizando una anchura de contenedor especificada como parámetro (por ejemplo, con 1024, los tamaños de archivos de 0 a 1023 van en un contenedor, de 1024 a 2047 van en el siguiente contenedor, etcétera).

37. Escriba un programa que explore todos los directorios en un sistema de archivos UNIX, que busque y localice todos los nodos-i con una cuenta de vínculos duros de dos o más. Para cada uno de esos archivos debe listar en conjunto todos los nombres de archivos que apunten a ese archivo.

38. Escriba una nueva versión del programa *ls* de UNIX. Esta versión debe tomar como argumento uno o más nombres de directorios y para cada directorio debe listar todos los archivos que contiene, una línea por archivo. Cada campo debe tener un formato razonable, conn base en su tipo. Liste sólo las primeras direcciones de disco, si las hay.

5

ENTRADA/SALIDA

Además de proporcionar abstracciones como los procesos (e hilos), espacios de direcciones y archivos, un sistema operativo también controla todos los dispositivos de E/S (Entrada/Salida) de la computadora. Debe emitir comandos para los dispositivos, captar interrupciones y manejar errores. Adicionalmente debe proporcionar una interfaz —simple y fácil de usar— entre los dispositivos y el resto del sistema. Hasta donde sea posible, la interfaz debe ser igual para todos los dispositivos (independencia de dispositivos). El código de E/S representa una fracción considerable del sistema operativo total. El tema de este capítulo es la forma en que el sistema operativo administra la E/S.

Este capítulo se organiza de la siguiente manera. Primero veremos algunos de los principios del hardware de E/S y después analizaremos el software de E/S en general. El software de E/S se puede estructurar en niveles, cada uno de los cuales tiene una tarea bien definida. Analizaremos estos niveles para describir qué hacen y cómo trabajan en conjunto.

Después de esa introducción analizaremos detalladamente hardware y software de varios dispositivos de E/S: discos, relojes, teclados y pantallas. Por último, consideraremos la administración de la energía.

5.1 PRINCIPIOS DEL HARDWARE DE E/S

Distintas personas ven el hardware de E/S de diferentes maneras. Los ingenieros eléctricos lo ven en términos de chips, cables, fuentes de poder, motores y todos los demás componentes físicos que constituyen el hardware. Los programadores ven la interfaz que se presenta al software: los comandos

que acepta el hardware, las funciones que lleva a cabo y los errores que se pueden reportar. En este libro nos enfocaremos en la programación de los dispositivos de E/S y no en el diseño, la construcción o el mantenimiento de los mismos, por lo que nuestro interés se limitará a ver cómo se programa el hardware, no cómo funciona por dentro. Sin embargo, la programación de muchos dispositivos de E/S a menudo está íntimamente conectada con su operación interna. En las siguientes tres secciones proveeremos un poco de historia general acerca del hardware de E/S y cómo se relaciona con la programación. Puede considerarse como un repaso y expansión del material introductorio de la sección 1.4.

5.1.1 Dispositivos de E/S

Los dispositivos de E/S se pueden dividir básicamente en dos categorías: **dispositivos de bloque** y **dispositivos de carácter**. Un dispositivo de bloque almacena información en bloques de tamaño fijo, cada uno con su propia dirección. Los tamaños de bloque comunes varían desde 512 bytes hasta 32,768 bytes. Todas las transferencias se realizan en unidades de uno o más bloques completos (consecutivos). La propiedad esencial de un dispositivo de bloque es que es posible leer o escribir cada bloque de manera independiente de los demás. Los discos duros, CD-ROMs y memorias USBs son dispositivos de bloque comunes.

Como resultado de un análisis más detallado, se puede concluir que no está bien definido el límite entre los dispositivos que pueden direccionarse por bloques y los que no se pueden direccionar así. Todos concuerdan en que un disco es un dispositivo direccionable por bloques, debido a que no importa dónde se encuentre el brazo en un momento dado, siempre será posible buscar en otro cilindro y después esperar a que el bloque requerido gire debajo de la cabeza. Ahora considere una unidad de cinta utilizada para realizar respaldos de disco. Las cintas contienen una secuencia de bloques. Si la unidad de cinta recibe un comando para leer el bloque *N*, siempre puede rebobinar la cinta y avanzar hasta llegar al bloque *N*. Esta operación es similar a un disco realizando una búsqueda, sólo que requiere mucho más tiempo. Además, puede o no ser posible volver a escribir un bloque a mitad de la cinta. Aun si fuera posible utilizar las cintas como dispositivos de bloque de acceso aleatorio, es algo que se sale de lo normal: las cintas no se utilizan de esa manera.

El otro tipo de dispositivo de E/S es el dispositivo de carácter. Un dispositivo de carácter envía o acepta un flujo de caracteres, sin importar la estructura del bloque. No es direccionable y no tiene ninguna operación de búsqueda. Las impresoras, las interfaces de red, los ratones (para señalar), las ratas (para los experimentos de laboratorio de psicología) y la mayoría de los demás dispositivos que no son parecidos al disco se pueden considerar como dispositivos de carácter.

Este esquema de clasificación no es perfecto. Algunos dispositivos simplemente no se adaptan. Por ejemplo, los relojes no son direccionables por bloques. Tampoco generan ni aceptan flujos de caracteres. Todo lo que hacen es producir interrupciones a intervalos bien definidos. Las pantallas por asignación de memoria tampoco se adaptan bien al modelo. Aún así, el modelo de dispositivos de bloque y de carácter es lo bastante general como para poder utilizarlo como base para hacer que parte del sistema operativo que lidia con los dispositivos de E/S sea independiente. Por ejemplo, el sistema de archivos sólo se encarga de los dispositivos de bloque abstractos y deja la parte dependiente de los dispositivos al software de bajo nivel.

Los dispositivos de E/S cubren un amplio rango de velocidades, lo cual impone una presión considerable en el software para obtener un buen desempeño sobre muchos órdenes de magnitud en las velocidades de transferencia de datos. La figura 5.1 muestra las velocidades de transferencia de datos de algunos dispositivos comunes. La mayoría de estos dispositivos tienden a hacerse más rápidos a medida que pasa el tiempo.

Dispositivo	Velocidad de transferencia de datos
Teclado	10 bytes/seg
Ratón	100 bytes/seg
Módem de 56K	7 KB/seg
Escáner	400 KB/seg
Cámara de video digital	3.5 MB/seg
802.11g inalámbrico	6.75 MB/seg
CD-ROM de 52X	7.8 MB/seg
Fast Ethernet	12.5 MB/seg
Tarjeta Compact Flash	40 MB/seg
FireWire (IEEE 1394)	50 MB/seg
USB 2.0	60 MB/seg
Red SONET OC-12	78 MB/seg
Disco SCSI Ultra 2	80 MB/seg
Gigabit Ethernet	125 MB/seg
Unidad de disco SATA	300 MB/seg
Cinta de Ultrium	320 MB/seg
Bus PCI	528 MB/seg

Figura 5-1. Velocidades de transferencia de datos comunes de algunos dispositivos, redes y buses.

5.1.2 Controladores de dispositivos

Por lo general, las unidades de E/S consisten en un componente mecánico y un componente electrónico. A menudo es posible separar las dos porciones para proveer un diseño más modular y general. El componente electrónico se llama **controlador de dispositivo** o **adaptador**. En las computadoras personales, comúnmente tiene la forma de un chip en la tarjeta principal o una tarjeta de circuito integrado que se puede insertar en una ranura de expansión (PCI). El componente mecánico es el dispositivo en sí. Este arreglo se muestra en la figura 1-6.

La tarjeta controladora por lo general contiene un conector, en el que se puede conectar un cable que conduce al dispositivo en sí. Muchos controladores pueden manejar dos, cuatro o inclusive ocho dispositivos idénticos. Si la interfaz entre el controlador y el dispositivo es estándar, ya sea un estándar oficial ANSI, IEEE o ISO, o un estándar de facto, entonces las empresas

pueden fabricar controladores o dispositivos que se adapten a esa interfaz. Por ejemplo, muchas empresas fabrican unidades de disco que coinciden con la interfaz IDE, SATA, SCSI, USB o FireWire (IEEE 1394).

La interfaz entre el controlador y el dispositivo es a menudo de muy bajo nivel. Por ejemplo, se podría dar formato a un disco con 10,000 sectores de 512 bytes por pista. Sin embargo, lo que en realidad sale del disco es un flujo serial de bits, empezando con un **preámbulo**, después los 4096 bits en un sector y por último una suma de comprobación, también conocida como **Código de Corrección de Errores (ECC)**. El preámbulo se escribe cuando se da formato al disco y contiene el cilindro y número de sector, el tamaño del sector y datos similares, así como información de sincronización.

El trabajo del controlador es convertir el flujo de bits serial en un bloque de bytes y realizar cualquier corrección de errores necesaria. Por lo general, primero se ensambla el bloque de bytes, bit por bit, en un búfer dentro del controlador. Después de haber verificado su suma de comprobación y de que el bloque se haya declarado libre de errores, puede copiarse a la memoria principal.

El controlador para un monitor también funciona como un dispositivo de bits en serie a un nivel igual de bajo. Lee los bytes que contienen los caracteres a mostrar de la memoria y genera las señales utilizadas para modular el haz del CRT para hacer que escriba en la pantalla. El controlador también genera las señales para que el haz del CRT realice un retrazado horizontal después de haber terminado una línea de exploración, así como las señales para hacer que realice un retrazado vertical después de haber explorado toda la pantalla. Si no fuera por el controlador del CRT, el programador del sistema operativo tendría que programar de manera explícita la exploración análoga del tubo. Con el controlador, el sistema operativo inicializa el controlador con unos cuantos parámetros, como el número de caracteres o píxeles por línea y el número de líneas por pantalla, y deja que el controlador se encargue de manejar el haz. Las pantallas TFT planas funcionan de manera diferente, pero son igualmente complicadas.

5.1.3 E/S por asignación de memoria

Cada controlador tiene unos cuantos registros que se utilizan para comunicarse con la CPU. Al escribir en ellos, el sistema operativo puede hacer que el dispositivo envíe o acepte datos, se encienda o se apague, o realice cualquier otra acción. Al leer de estos registros, el sistema operativo puede conocer el estado del dispositivo, si está preparado o no para aceptar un nuevo comando, y sigue procediendo de esa manera.

Además de los registros de control, muchos dispositivos tienen un búfer de datos que el sistema operativo puede leer y escribir. Por ejemplo, una manera común para que las computadoras muestren píxeles en la pantalla es tener una RAM de video, la cual es básicamente sólo un búfer de datos disponible para que los programas o el sistema operativo escriban en él.

De todo esto surge la cuestión acerca de cómo se comunica la CPU con los registros de control y los búferes de datos de los dispositivos. Existen dos alternativas. En el primer método, a cada registro de control se le asigna un número de **puerto de E/S**, un entero de 8 o 16 bits. El conjunto de todos los puertos de E/S forma el **espacio de puertos de E/S** y está protegido de manera que los

programas de usuario ordinarios no puedan utilizarlo (sólo el sistema operativo puede). Mediante el uso de una instrucción especial de E/S tal como

```
IN REG,PUERTO,
```

la CPU puede leer el registro de control `PUERTO` y almacenar el resultado en el registro de la CPU llamado `REG`. De manera similar, mediante el uso de

```
OUT PUERTO, REG
```

la CPU puede escribir el contenido de `REG` en un registro de control. La mayoría de las primeras computadoras, incluyendo a casi todas las mainframes, como la IBM 360 y todas sus sucesoras, trabajaban de esta manera.

En este esquema, los espacios de direcciones para la memoria y la E/S son distintos, como se muestra en la figura 5-2(a). Las instrucciones

```
IN R0,4
```

y

```
MOV R0,4
```

son completamente distintas en este diseño. La primera lee el contenido del puerto 4 de E/S y lo coloca en `R0`, mientras que la segunda lee el contenido de la palabra de memoria 4 y lo coloca en `R0`. Los 4s en estos ejemplos se refieren a espacios de direcciones distintos y que no están relacionados.

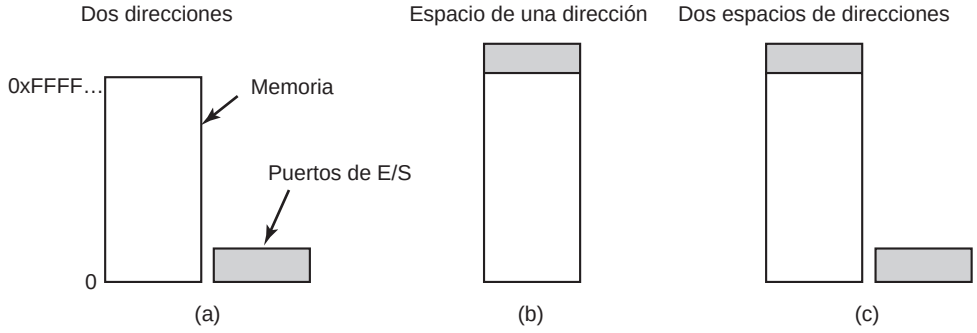

Figura 5-2. (a) Espacio separado de E/S y memoria. (b) E/S por asignación de memoria. (c) Híbrido.

El segundo método, que se introdujo con la PDP-11, es asignar todos los registros de control al espacio de memoria, como se muestra en la figura 5-2(b). A cada registro de control se le asigna una dirección de memoria única a la cual no hay memoria asignada. Este sistema se conoce como **E/S con asignación de memoria** (*mapped-memory*). Por lo general, las direcciones asignadas se encuentran en la parte superior del espacio de direcciones. En la figura 5-2(c) se muestra un esquema híbrido, con búferes de datos de E/S por asignación de memoria y puertos de E/S separados para los registros de control. El Pentium utiliza esta arquitectura, donde las direcciones de 640K a 1M se reservan para los búferes de datos de los dispositivos en las IBM PC compatibles, además de los puertos de E/S de 0 a 64K.

¿Cómo funcionan estos esquemas? En todos los casos, cuando la CPU desea leer una palabra, ya sea de la memoria o de un puerto de E/S, coloca la dirección que necesita en las líneas de dirección del bus y después impone una señal READ en una línea de control del bus. Se utiliza una segunda línea de señal para indicar si se necesita espacio de E/S o de memoria. Si es espacio de memoria, ésta responde a la petición. Si es espacio de E/S, el dispositivo de E/S responde a la petición. Si sólo hay espacio de memoria [como en la figura 5-2(b)], todos los módulos de memoria y todos los dispositivos de E/S comparan las líneas de dirección con el rango de direcciones a las que dan servicio. Si la dirección está en su rango, responde a la petición. Como ninguna dirección se asigna tanto a la memoria como a un dispositivo de E/S, no hay ambigüedad ni conflicto.

Los dos esquemas para direccionar los controladores tienen distintos puntos fuertes y débiles. Vamos a empezar con las ventajas de la E/S por asignación de memoria. En primer lugar, si se necesitan instrucciones de E/S especiales para leer y escribir en los registros de control, para acceder a ellos se requiere el uso de código ensamblador, ya que no hay forma de ejecutar una instrucción IN o OUT en C o C++. Al llamar a dicho procedimiento se agrega sobrecarga para controlar la E/S. En contraste, con la E/S por asignación de memoria los registros de control de dispositivos son sólo variables en memoria y se pueden direccionar en C de la misma forma que cualquier otra variable. Así, con la E/S por asignación de memoria, un controlador de dispositivo de E/S puede escribirse completamente en C. Sin la E/S por asignación de memoria se requiere cierto código ensamblador.

En segundo lugar, con la E/S por asignación de memoria no se requiere un mecanismo de protección especial para evitar que los procesos realicen operaciones de E/S. Todo lo que el sistema operativo tiene que hacer es abstenerse de colocar esa porción del espacio de direcciones que contiene los registros de control en el espacio de direcciones virtuales de cualquier usuario. Mejor aún, si cada dispositivo tiene sus registros de control en una página distinta del espacio de direcciones, el sistema operativo puede proporcionar a un usuario el control sobre dispositivos específicos pero no el de los demás, con sólo incluir las páginas deseadas en su tabla de páginas. Dicho esquema permite que se coloquen distintos controladores de dispositivos en diferentes espacios de direcciones, lo cual no sólo reduce el tamaño del kernel, sino que también evita que un controlador interfiera con los demás.

En tercer lugar, con la E/S por asignación de memoria, cada instrucción que puede hacer referencia a la memoria también puede hacerla a los registros de control. Por ejemplo, si hay una instrucción llamada TEST que evalúe si una palabra de memoria es 0, también se puede utilizar para evaluar si un registro de control es 0, lo cual podría ser la señal de que el dispositivo está inactivo y en posición de aceptar un nuevo comando. El código en lenguaje ensamblador podría ser así:

```
CICLO:   TEST PUERTO_4      // comprueba si el puerto 4 es 0
         BEQ LISTO          // si es 0, ir a listo
         BRANCH CICLO       // en caso contrario continúa la prueba
LISTO:
```

Si la E/S por asignación de memoria no está presente, el registro de control se debe leer primero en la CPU y después se debe evaluar, para lo cual se requieren dos instrucciones en vez de una. En el caso del ciclo antes mencionado, se tiene que agregar una cuarta instrucción, con lo cual se reduce ligeramente la capacidad de respuesta al detectar un dispositivo inactivo.

En el diseño de computadoras, casi todo implica tener que hacer concesiones, y aquí también es el caso. La E/S por asignación de memoria también tiene sus desventajas. En primer lugar, la mayoría de las computadoras actuales tienen alguna forma de colocar en caché las palabras de memoria. Sería desastroso colocar en caché un registro de control de dispositivos. Considere el ciclo de código en lenguaje ensamblador antes mostrado en presencia de caché. La primera referencia a `PUERTO_4` haría que se colocara en la caché. Las referencias subsiguientes sólo tomarían el valor de la caché sin siquiera preguntar al dispositivo. Después, cuando el dispositivo por fin estuviera listo, el software no tendría manera de averiguarlo. En vez de ello, el ciclo continuaría para siempre.

Para evitar esta situación relacionada con la E/S por asignación de memoria, el hardware debe ser capaz de deshabilitar la caché en forma selectiva; por ejemplo, por página. Esta característica agrega una complejidad adicional al hardware y al sistema operativo, que tiene que encargarse de la caché selectiva.

En segundo lugar, si sólo hay un espacio de direcciones, entonces todos los módulos de memoria y todos los dispositivos de E/S deben examinar todas las referencias a memoria para ver a cuáles debe responder. Si la computadora tiene un solo bus, como en la figura 5-3(a), es simple hacer que todos analicen cada dirección.

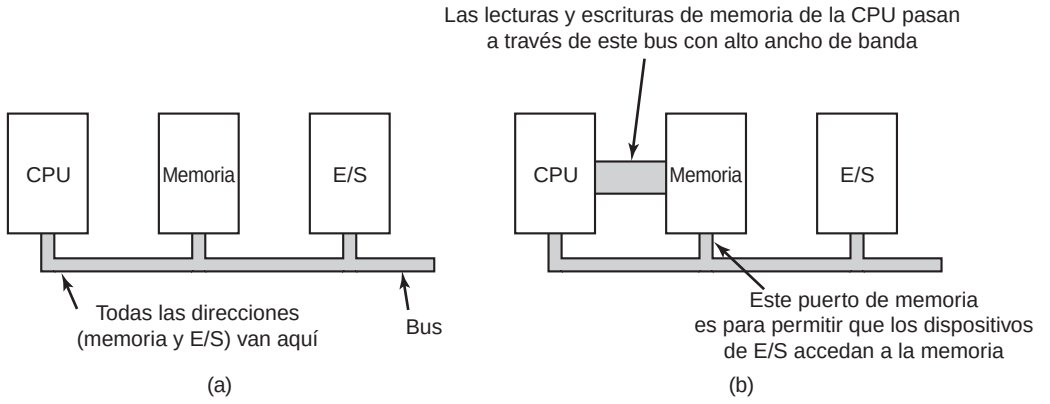

Figura 5-3. (a) Arquitectura con un solo bus. (b) Arquitectura con bus dual.

Sin embargo, la tendencia en las computadoras personales modernas es tener un bus de memoria dedicado de alta velocidad, como se muestra en la figura 5-3(b), una propiedad que también se encuentra incidentalmente en las mainframes. Este bus está ajustado para optimizar el rendimiento de la memoria sin sacrificarse por el bienestar de los dispositivos de E/S lentos. Los sistemas Pentium pueden tener varios buses (memoria, PCI, SCSI, USB, ISA), como se muestra en la figura 1-12.

El problema de tener un bus de memoria separado en equipos con asignación de memoria es que los dispositivos no tienen forma de ver las direcciones de memoria a medida que recorren el bus de memoria, por lo que no tienen manera de responderles. De nuevo es necesario tomar medidas especiales para hacer que la E/S por asignación de memoria funcione en un sistema con varios buses. Una posibilidad es enviar primero todas las referencias de memoria a la memoria; si ésta no responde, entonces la CPU prueba los otros buses. Este diseño puede funcionar, pero requiere una complejidad adicional del hardware.

Un segundo posible diseño es colocar un dispositivo husmeador en el bus de memoria para pasar todas las direcciones presentadas a los dispositivos de E/S potencialmente interesados. El problema aquí es que los dispositivos de E/S tal vez no puedan procesar las peticiones a la velocidad que puede procesarlas la memoria.

Un tercer posible diseño, que es el utilizado en la configuración del Pentium de la figura 1-12, es filtrar las direcciones en el chip del puente PCI. Este chip contiene registros de rango que se cargan previamente en tiempo de inicio del sistema. Por ejemplo, el rango de 640K a 1M se podría marcar como un rango que no tiene memoria; las direcciones ubicadas dentro de uno de los rangos así marcados se reenvían al bus PCI, en vez de reenviarse a la memoria. La desventaja de este esquema es la necesidad de averiguar en tiempo de inicio del sistema qué direcciones de memoria no lo son en realidad. Por ende, cada esquema tiene argumentos a favor y en contra, por lo que los sacrificios y las concesiones son inevitables.

5.1.4 Acceso directo a memoria (DMA)

Sin importar que una CPU tenga o no E/S por asignación de memoria, necesita direccionar los controladores de dispositivos para intercambiar datos con ellos. La CPU puede solicitar datos de un controlador de E/S un bit a la vez, pero al hacerlo se desperdicia el tiempo de la CPU, por lo que a menudo se utiliza un esquema distinto, conocido como **DMA** (**Acceso Directo a Memoria**). El sistema operativo sólo puede utilizar DMA si el hardware tiene un controlador de DMA, que la mayoría de los sistemas tienen. Algunas veces este controlador está integrado a los controladores de disco y otros controladores, pero dicho diseño requiere un controlador de DMA separado para cada dispositivo. Lo más común es que haya un solo controlador de DMA disponible (por ejemplo, en la tarjeta principal) para regular las transferencias a varios dispositivos, a menudo en forma concurrente.

Sin importar cuál sea su ubicación física, el controlador de DMA tiene acceso al bus del sistema de manera independiente de la CPU, como se muestra en la figura 5-4. Contiene varios registros en los que la CPU puede escribir y leer; éstos incluyen un registro de dirección de memoria, un registro contador de bytes y uno o más registros de control. Los registros de control especifican el puerto de E/S a utilizar, la dirección de la transferencia (si se va a leer del dispositivo de E/S o se va a escribir en él), la unidad de transferencia (un byte a la vez o una palabra a la vez), y el número de bytes a transferir en una ráfaga.

Para explicar el funcionamiento del DMA, veamos primero cómo ocurren las lecturas de disco cuando no se utiliza DMA. Primero, el controlador de disco lee el bloque (uno o más sectores) de la unidad en forma serial, bit por bit, hasta que se coloca todo el bloque completo en el búfer interno del controlador. Después calcula la suma de comprobación para verificar que no hayan ocurrido errores de lectura. Después, el controlador produce una interrupción. Cuando el sistema operativo empieza a ejecutarse, puede leer el bloque de disco del búfer del controlador, un byte o una palabra a la vez, mediante la ejecución de un ciclo en el que cada iteración se lee un byte o palabra de un registro de dispositivo controlador y se almacena en la memoria principal.

Cuando se utiliza DMA, el procedimiento es distinto. Primero, la CPU programa el controlador de DMA, para lo cual establece sus registros de manera que sepa qué debe transferir y a dónde

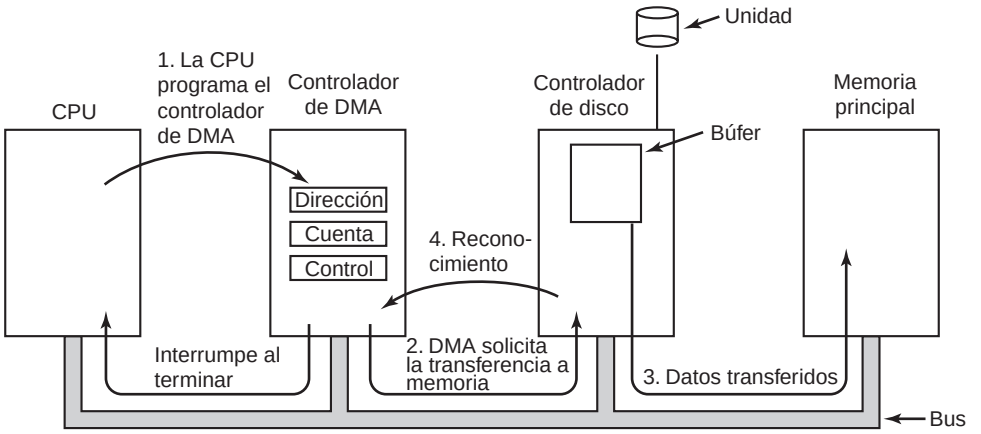

Figura 5-4. Operación de una transferencia de DMA.

(paso 1 en la figura 5-4). También emite un comando al controlador de disco para indicarle que debe leer datos del disco en su búfer interno y verificar la suma de comprobación. Cuando hay datos válidos en el búfer del controlador de disco, puede empezar el DMA.

El controlador de DMA inicia la transferencia enviando una petición de lectura al controlador de disco mediante el bus (paso 2). Esta petición de lectura se ve como cualquier otra petición de lectura, por lo que el controlador de disco no sabe ni le importa si vino de la CPU o de un controlador de DMA. Por lo general, la dirección de memoria en la que se va a escribir está en las líneas de dirección del bus, por lo que cuando el controlador de disco obtiene la siguiente palabra de su búfer interno, sabe dónde escribir. La escritura en memoria es otro ciclo de bus estándar (paso 3). Cuando se completa la escritura, el controlador de disco envía una señal de reconocimiento al controlador de DMA, también a través del bus (paso 4). El controlador de DMA incrementa a continuación la dirección de memoria a utilizar y disminuye la cuenta de bytes. Si la cuenta de bytes es aún mayor que 0, se repiten los pasos del 2 al 4 hasta que la cuenta llega a 0. En ese momento, el controlador de DMA interrumpe la CPU para hacerle saber que la transferencia está completa. Cuando el sistema operativo se inicia, no tiene que copiar el bloque de disco en la memoria; ya se encuentra ahí.

Los controladores de DMA varían considerablemente en cuanto a su sofisticación. Los más simples manejan una transferencia a la vez, como se describió antes. Los más complejos se pueden programar para manejar varias transferencias a la vez. Dichos controladores tienen varios conjuntos de registros internos, uno para cada canal. La CPU empieza por cargar cada conjunto de registros con los parámetros relevantes para su transferencia. Cada transferencia debe utilizar un controlador de dispositivo distinto. Después de transferir cada palabra (los pasos 2 al 4) en la figura 5-4, el controlador de DMA decide a cuál dispositivo va a dar servicio a continuación. Se puede configurar para utilizar un algoritmo por turno rotatorio (*round-robin*), o puede tener un diseño de esquema prioritario para favorecer a unos dispositivos sobre otros. Puede haber varias peticiones a distintos controladores de dispositivos pendientes al mismo tiempo, siempre y cuando haya

una manera ambigua de diferenciar las señales de reconocimiento. A menudo se utiliza una línea de reconocimiento distinta en el bus para cada canal de DMA por esta razón.

Muchos buses pueden operar en dos modos: el modo de una palabra a la vez y el modo de bloque. Algunos controladores de DMA también pueden operar en cualquier modo. En el modo anterior, la operación es como se describe antes: el controlador de DMA solicita la transferencia de una palabra y la obtiene; si la CPU también desea el bus, tiene que esperar. El mecanismo se llama **robo de ciclo** debido a que el controlador de dispositivo se acerca de manera sigilosa y roba un ciclo de bus a la CPU de vez en cuando, retrasándola ligeramente. En el modo de bloque, el controlador de DMA indica al dispositivo que debe adquirir el bus, emitir una serie de transferencias y después liberar el bus. Esta forma de operación se conoce como **modo de ráfaga**. Es más eficiente que el robo de ciclo, debido a que la adquisición del bus requiere tiempo y se pueden transferir varias palabras con sólo una adquisición de bus. La desventaja en comparación con el modo de ráfaga es que puede bloquear la CPU y otros dispositivos por un periodo considerable, si se está transfiriendo una ráfaga extensa.

En el modelo que hemos estado analizando, algunas veces conocido como **modo "fly-by"**, el controlador de DMA indica al controlador de dispositivo que transfiera los datos directamente a la memoria principal. Un modo alternativo que utilizan algunos controladores de DMA es hacer que el controlador de dispositivo envíe la palabra al controlador de DMA, el cual emite después una segunda solicitud de bus para escribir la palabra a donde se supone que debe ir. Este esquema requiere un ciclo de bus adicional por cada palabra transferida, pero es más flexible en cuanto a que puede realizar también copias de dispositivo a dispositivo, e incluso copias de memoria a memoria (para lo cual primero emite una lectura a memoria y después una escritura a memoria en una dirección distinta).

La mayoría de los controladores de DMA utilizan direcciones físicas de memoria para sus transferencias. El uso de direcciones físicas requiere que el sistema operativo convierta la dirección virtual del búfer de memoria deseado a una dirección física, y que escriba esta dirección física en el registro de dirección del controlador de DMA. Un esquema alternativo que se utiliza en unos cuantos controladores de DMA es escribir direcciones virtuales en el controlador de DMA. Así, el controlador de DMA debe utilizar la MMU para realizar la traducción de dirección virtual a física. Sólo en el caso en que la MMU sea parte de la memoria (es posible, pero raro) en vez de formar parte de la CPU, las direcciones virtuales se podrán colocar en el bus.

Como lo mencionamos anteriormente, antes de que pueda iniciar el DMA, el disco lee primero los datos en su búfer interno. Tal vez el lector se pregunte por qué el controlador no sólo almacena los bytes en memoria tan pronto como los obtiene del disco. En otras palabras, ¿para qué necesita un búfer interno? Hay dos razones: en primer lugar, al utilizar el búfer interno, el controlador de disco puede verificar la suma de comprobación antes de iniciar una transferencia y si la suma de comprobación es incorrecta se señala un error y no se realiza la transferencia; la segunda es que, una vez que se inicia una transferencia de disco, los bits siguen llegando a una velocidad constante, esté listo o no el controlador para ellos. Si el controlador tratara de escribir datos directamente en la memoria, tendría que pasar por el bus del sistema por cada palabra transferida. Si el bus estuviera ocupado debido a que algún otro dispositivo lo estuviera utilizando (por ejemplo, en modo ráfaga), el controlador tendría que esperar. Si la siguiente palabra del disco llegar antes de que se almacenara la anterior, el controlador tendría que almacenarla en algún otro lado. Si el bus estu-

viera muy ocupado, el controlador podría terminar almacenando muy pocas palabras y tendría que realizar también mucho trabajo de administración. Cuando el bloque se coloca en un búfer interno, el bus no se necesita sino hasta que empieza el DMA, por lo que el diseño del controlador es mucho más simple debido a que la transferencia de DMA a la memoria no es muy estricta en cuanto al tiempo (de hecho, algunos controladores antiguos van directamente a la memoria con sólo una pequeña cantidad de uso del búfer interno, pero cuando el bus está muy ocupado, tal vez haya que terminar una transferencia con un error de desbordamiento).

No todas las computadoras utilizan DMA. El argumento en contra es que la CPU principal es a menudo más rápida que el controlador de DMA y puede realizar el trabajo con mucha mayor facilidad (cuando el factor limitante no es la velocidad del dispositivo de E/S). Si no hay otro trabajo que realizar, hacer que la CPU (rápida) tenga que esperar al controlador de DMA (lento) para terminar no tiene caso. Además, al deshacernos del controlador de DMA y hacer que la CPU realice todo el trabajo en software se ahorra dinero, lo cual es importante en las computadoras de bajo rendimiento (incrustadas o embebidas).

5.1.5 Repaso de las interrupciones

En la sección 1.4.5 vimos una introducción breve a las interrupciones, pero hay más que decir. En un sistema de computadora personal común, la estructura de las interrupciones es como se muestra en la figura 5-5. A nivel de hardware, las interrupciones funcionan de la siguiente manera. Cuando un dispositivo de E/S ha terminado el trabajo que se le asignó, produce una interrupción (suponiendo que el sistema operativo haya habilitado las interrupciones). Para ello, impone una señal en una línea de bus que se le haya asignado. Esta señal es detectada por el chip controlador de interrupciones en la tarjeta principal, que después decide lo que debe hacer.

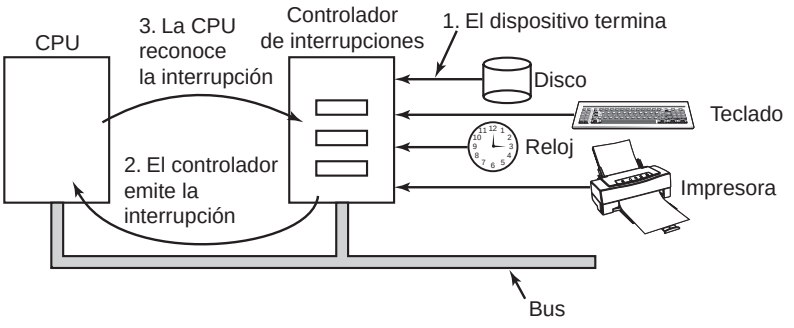

Figura 5-5. Cómo ocurre una interrupción. Las conexiones entre los dispositivos y el controlador de interrupciones en realidad utilizan líneas de interrupción en el bus, en vez de cables dedicados.

Si no hay otras interrupciones pendientes, el controlador de interrupciones procesa la interrupción de inmediato. Si hay otra en progreso, o si otro dispositivo ha realizado una petición simultánea en una línea de petición de interrupción de mayor prioridad en el bus, el dispositivo sólo se

ignora por el momento. En este caso, continúa imponiendo una señal de interrupción en el bus hasta que la CPU la atiende.

Para manejar la interrupción, el controlador coloca un número en las líneas de dirección que especifican cuál dispositivo desea atención e impone una señal para interrumpir a la CPU.

La señal de interrupción hace que la CPU deje lo que está haciendo y empiece a hacer otra cosa. El número en las líneas de dirección se utiliza como índice en una tabla llamada **vector de interrupciones** para obtener un nuevo contador del programa. Este contador del programa apunta al inicio del procedimiento de servicio de interrupciones correspondiente. Por lo general, las trampas e interrupciones utilizan el mismo mecanismo desde este punto en adelante, y con frecuencia comparten el mismo vector de interrupción. La ubicación del vector de interrupción se puede determinar de manera estática (*hardwired*) en la máquina, o puede estar en cualquier parte de la memoria, con un registro de la CPU (cargado por el sistema operativo) apuntando a su origen.

Poco después de que se empieza a ejecutar, el procedimiento de servicio de interrupciones reconoce la interrupción al escribir cierto valor en uno de los puertos de E/S del controlador de interrupciones. Este reconocimiento indica al controlador que puede emitir otra interrupción. Al hacer que la CPU retrase este reconocimiento hasta que esté lista para manejar la siguiente interrupción, se pueden evitar las condiciones de competencia que involucran varias interrupciones (casi simultáneas). Por otra parte, algunas computadoras (antiguas) no tienen un controlador de interrupciones centralizado, por lo que cada controlador de dispositivo solicita sus propias interrupciones.

El hardware siempre guarda cierta información antes de iniciar el procedimiento de servicio. La información que se guarda y el lugar en donde se guarda varía de manera considerable, entre una CPU y otra. Como mínimo se debe guardar el contador del programa para que se pueda reiniciar el proceso interrumpido. En el otro extremo, todos los registros visibles y un gran número de registros internos se pueden guardar también.

Una cuestión es dónde guardar esta información. Una opción es colocarla en registros internos que el sistema operativo pueda leer según sea necesario. Un problema derivado de este método es que entonces el controlador de interrupciones no puede recibir una señal de reconocimiento sino hasta que se haya leído toda la información potencialmente relevante, por temor a que una segunda interrupción sobrescriba los registros internos que guardan el estado. Esta estrategia conlleva tiempos inactivos más largos cuando se deshabilitan las interrupciones, y posiblemente interrupciones y datos perdidos.

En consecuencia, la mayoría de las CPUs guardan la información en la pila. Sin embargo, este método también tiene problemas. Para empezar, ¿la pila de quién? Si se utiliza la pila actual, podría muy bien ser una pila de un proceso de usuario. El apuntador a la pila tal vez ni siquiera sea legal, lo cual produciría un error fatal cuando el hardware tratara de escribir ciertas palabras en la dirección a la que estuviera apuntando. Además, podría apuntar al final de una página. Después de varias escrituras en memoria, se podría exceder el límite de página y generar un fallo de página. Al ocurrir un fallo de página durante el procesamiento de la interrupción de hardware se crea un mayor problema: ¿Dónde se debe guardar el estado para manejar el fallo de página?

Si se utiliza la pila del kernel, hay una probabilidad mucho mayor de que el apuntador de la pila sea legal y que apunte a una página marcada. Sin embargo, para cambiar al modo kernel tal vez se requiera cambiar de contextos de la MMU, y probablemente se invalide la mayoría de (o toda) la caché y el TLB. Si se vuelven a cargar todos estos, ya sea en forma estática o dinámica, aumenta el tiempo requerido para procesar una interrupción, y por ende se desperdiciará tiempo de la CPU.

Interrupciones precisas e imprecisas

Otro problema es ocasionado por el hecho de que la mayoría de las CPUs modernas tienen muchas líneas de tuberías y a menudo son superescalares (paralelas en su interior). En los sistemas antiguos, después de que cada instrucción terminaba de ejecutarse, el microprograma o hardware comprobaba si había una interrupción pendiente. De ser así, el contador del programa y el PSW se metían en la pila y empezaba la secuencia de interrupción. Una vez que se ejecutaba el manejador de interrupciones, se llevaba a cabo el proceso inverso, en donde el PSW antiguo y el contador del programa se sacaban de la pila y continuaba el proceso anterior.

Este modelo hace la suposición implícita de que si ocurre una interrupción justo después de cierta instrucción, todas las instrucciones hasta (e incluyendo) esa instrucción se han ejecutado por completo, y no se ha ejecutado ninguna instrucción después de ésta. En equipos antiguos, esta suposición siempre era válida. En los modernos tal vez no sea así.

Para empezar, considere el modelo de línea de tubería de la figura 1-7(a).¿Qué pasa si ocurre una interrupción mientras la línea de tubería está llena (el caso usual)? Muchas instrucciones se encuentran en varias etapas de ejecución. Cuando ocurre la interrupción, el valor del contador del programa tal vez no refleje el límite correcto entre las instrucciones ejecutadas y las no ejecutadas. De hecho, muchas instrucciones pueden haberse ejecutado en forma parcial, y distintas instrucciones podrían estar más o menos completas. En esta situación, el contador del programa refleja con más probabilidad la dirección de la siguiente instrucción a obtener y meter en la línea de tubería, en vez de la dirección de la instrucción que acaba de ser procesada por la unidad de ejecución.

En una máquina superescalar, como la de la figura 1-7(b), las cosas empeoran. Las instrucciones se pueden descomponer en microoperaciones y éstas se pueden ejecutar en desorden, dependiendo de la disponibilidad de los recursos internos como las unidades funcionales y los registros. Al momento de una interrupción, algunas instrucciones iniciadas hace tiempo tal vez no hayan empezado, y otras que hayan empezado más recientemente pueden estar casi completas. Al momento en que se señala una interrupción, puede haber muchas instrucciones en varios estados de avance, con menos relación entre ellas y el contador del programa.

Una interrupción que deja al equipo en un estado bien definido se conoce como **interrupción precisa** (Walker y Cragon, 1995). Dicha interrupción tiene cuatro propiedades:

1. El contador del programa (PC) se guarda en un lugar conocido.
2. Todas las instrucciones antes de la instrucción a la que apunta el PC se han ejecutado por completo.
3. Ninguna instrucción más allá de la instrucción a la que apunta el PC se ha ejecutado.
4. Se conoce el estado de ejecución de la instrucción a la que apunta el PC.

Observe que no hay prohibición sobre las instrucciones más allá de la instrucción a la que apunta el PC acerca de si pueden iniciar o no. Es sólo que los cambios que realizan a los registros o la memoria se deben deshacer antes de que ocurra la interrupción. Se permite ejecutar la instrucción a la que se apunta. También se permite que no se haya ejecutado. Sin embargo, debe estar claro cuál es el caso que se aplica. A menudo, si la interrupción es de E/S, todavía no habrá empezado. No

obstante, si la interrupción es en realidad una trampa o fallo de página, entonces el PC generalmente apunta a la instrucción que ocasionó el fallo, por lo que puede reiniciarse después. La situación de la figura 5-6(a) ilustra una interrupción precisa. Todas las instrucciones hasta el contador del programa (316) se han completado, y ninguna más allá de éstas ha iniciado (o se ha rechazado para deshacer sus efectos).

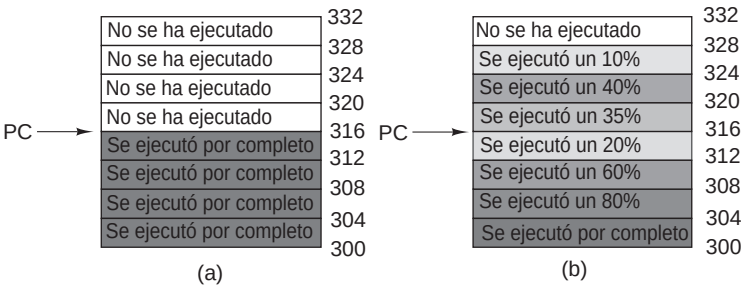

Figura 5-6. (a) Una interrupción precisa. (b) Una interrupción imprecisa.

Una interrupción que no cumple con estos requerimientos se conoce como **interrupción imprecisa** y hace la vida muy incómoda para el escritor del sistema operativo, que entonces tiene que averiguar lo que ocurrió y lo que aún está por ocurrir. La figura 5-6(b) muestra una interrupción imprecisa, en donde hay distintas instrucciones cerca del contador del programa en distintos estados de avance, y las más antiguas no necesariamente están más completas que las más recientes. Las máquinas con interrupciones imprecisas por lo general vuelcan una gran cantidad de estado interno en la pila, para dar al sistema operativo la posibilidad de averiguar qué está pasando. El código necesario para reiniciar la máquina es por lo general muy complicado. Además, al guardar una cantidad extensa de información para la memoria en cada interrupción se reduce aún más la velocidad de las interrupciones y la recuperación. Esto conlleva a una irónica situación en la que se tienen CPUs superescalares muy veloces, que algunas veces no son adecuadas para el trabajo real debido a la lentitud de las interrupciones.

Algunas computadoras están diseñadas de manera que ciertos tipos de interrupciones y trampas (traps) sean precisas y otras no. Por ejemplo, hacer que las interrupciones de E/S sean precisas, pero que las trampas producidas por errores de programación fatales sean imprecisas no es tan malo, ya que no hay necesidad de intentar reiniciar un proceso en ejecución una vez que ha dividido entre cero. Algunas máquinas tienen un bit que se puede establecer para forzar a que todas las interrupciones sean precisas. La desventaja de establecer este bit es que obliga a la CPU a registrar cuidadosamente todo lo que está haciendo, y a mantener copias “sombra” de los registros para que pueda generar una interrupción precisa en cualquier instante. Toda esta sobrecarga tiene un gran impacto sobre el rendimiento.

Algunas máquinas superescalares como la serie Pentium tienen interrupciones precisas para permitir que el software antiguo funcione correctamente. El precio a pagar por las interrupciones precisas es una lógica de interrupciones en extremo compleja dentro de la CPU necesaria para asegurar que cuando el controlador de interrupciones indique que desea ocasionar una interrupción,

todas las instrucciones hasta cierto punto puedan terminar y ninguna más allá de ese punto pueda tener un efecto considerable sobre el estado de la máquina. Aquí el precio no se paga con tiempo, sino en área del chip y en la complejidad del diseño. Si no se requirieran interrupciones precisas para fines de compatibilidad con aplicaciones antiguas, esta área del chip estaría disponible para cachés en chip más grandes, haciendo a la CPU más veloz. Por otra parte, las interrupciones hacen que el sistema operativo sea mucho más complicado y lento, por lo que es difícil saber qué método es mejor en realidad.

5.2 FUNDAMENTOS DEL SOFTWARE DE E/S

Ahora vamos a alejarnos del hardware de E/S para analizar el software de E/S. Primero analizaremos los objetivos del software de E/S y después las distintas formas en que se puede realizar la E/S desde el punto de vista del sistema operativo.

5.2.1 Objetivos del software de E/S

Un concepto clave en el diseño del software de E/S se conoce como **independencia de dispositivos**. Lo que significa es que debe ser posible escribir programas que puedan acceder a cualquier dispositivo de E/S sin tener que especificar el dispositivo por adelantado. Por ejemplo, un programa que lee un archivo como entrada debe tener la capacidad de leer un archivo en el disco duro, un CD-ROM, un DVD o una memoria USB sin tener que modificar el programa para cada dispositivo distinto. De manera similar, debe ser posible escribir un comando tal como

```
sort <entrada> salida
```

y hacer que funcione con datos de entrada provenientes de cualquier tipo de disco o del teclado, y que los datos de salida vayan a cualquier tipo de disco o a la pantalla. Depende del sistema operativo encargarse de los problemas producidos por el hecho de que estos dispositivos en realidad son diferentes y requieren secuencias de comandos muy distintas para leer o escribir.

Un objetivo muy relacionado con la independencia de los dispositivos es la **denominación uniforme**. El nombre de un archivo o dispositivo simplemente debe ser una cadena o un entero sin depender del dispositivo de ninguna forma. En UNIX, todos los discos se pueden integrar en la jerarquía del sistema de archivos de maneras arbitrarias, por lo que el usuario no necesita estar al tanto de cuál nombre corresponde a cuál dispositivo. Por ejemplo, una memoria USB se puede **montar** encima del directorio */usr/ast/respaldo*, de manera que al copiar un archivo a */usr/ast/respaldo/lunes*, este archivo se copie a la memoria USB. De esta forma, todos los archivos y dispositivos se direccionan de la misma forma: mediante el nombre de una ruta.

Otra cuestión importante relacionada con el software de E/S es el **manejo de errores**. En general, los errores se deben manejar lo más cerca del hardware que sea posible. Si el controlador descubre un error de lectura, debe tratar de corregir el error por sí mismo. Si no puede, entonces el software controlador del dispositivo debe manejarlo, tal vez con sólo tratar de leer el bloque de nuevo. Muchos errores son transitorios, como los errores de lectura ocasionados por pizcas de polvo en

la cabeza de lectura, y comúnmente desaparecen si se repite la operación. Sólo si los niveles inferiores no pueden lidiar con el problema, los niveles superiores deben saber acerca de ello. En muchos casos, la recuperación de los errores se puede hacer de manera transparente a un nivel bajo, sin que los niveles superiores se enteren siquiera sobre el error.

Otra cuestión clave es la de las transferencias **síncronas** (de bloqueo) contra las **asíncronas** (controladas por interrupciones). La mayoría de las operaciones de E/S son asíncronas: la CPU inicia la transferencia y se va a hacer algo más hasta que llega la interrupción. Los programas de usuario son mucho más fáciles de escribir si las operaciones de E/S son de bloqueo: después de una llamada al sistema `read`, el programa se suspende de manera automática hasta que haya datos disponibles en el búfer. Depende del sistema operativo hacer que las operaciones que en realidad son controladas por interrupciones parezcan de bloqueo para los programas de usuario.

Otra cuestión relacionada con el software de E/S es el **uso de búfer**. A menudo los datos que provienen de un dispositivo no se pueden almacenar directamente en su destino final. Por ejemplo, cuando un paquete llega de la red, el sistema operativo no sabe dónde colocarlo hasta que ha almacenado el paquete en alguna parte y lo examina. Además, algunos dispositivos tienen severas restricciones en tiempo real (por ejemplo, los dispositivos de audio digital), por lo que los datos se deben colocar en un búfer de salida por adelantado para desacoplar la velocidad a la que se llena el búfer, de la velocidad a la que se vacía, de manera que se eviten sub-desbordamientos de búfer. El uso de búfer involucra una cantidad considerable de copiado y a menudo tiene un importante impacto en el rendimiento de la E/S.

El concepto final que mencionaremos aquí es la comparación entre los dispositivos compartidos y los dispositivos dedicados. Algunos dispositivos de E/S, como los discos, pueden ser utilizados por muchos usuarios a la vez. No se producen problemas debido a que varios usuarios tengan archivos abiertos en el mismo disco al mismo tiempo. Otros dispositivos, como las unidades de cinta, tienen que estar dedicados a un solo usuario hasta que éste termine. Después, otro usuario puede tener la unidad de cinta. Si dos o más usuarios escriben bloques entremezclados al azar en la misma cinta, definitivamente no funcionará. Al introducir los dispositivos dedicados (no compartidos) también se introduce una variedad de problemas, como los interbloqueos. De nuevo, el sistema operativo debe ser capaz de manejar los dispositivos compartidos y dedicados de una manera que evite problemas.

5.2.2 E/S programada

Hay tres maneras fundamentalmente distintas en que se puede llevar a cabo la E/S. En esta sección analizaremos la primera (E/S programada). En las siguientes dos secciones examinaremos las otras (E/S controlada por interrupciones y E/S mediante el uso de DMA). La forma más simple de E/S es cuando la CPU hace todo el trabajo. A este método se le conoce como **E/S programada**.

Es más simple ilustrar la E/S programada por medio de un ejemplo. Considere un proceso de usuario que desea imprimir la cadena de ocho caracteres “ABCDEFGH” en la impresora. Primero ensambla la cadena en un búfer en espacio de usuario, como se muestra en la figura 5-7(a).

Después el proceso de usuario adquiere la impresora para escribir, haciendo una llamada al sistema para abrirla. Si la impresora está actualmente siendo utilizada por otro proceso, esta llamada

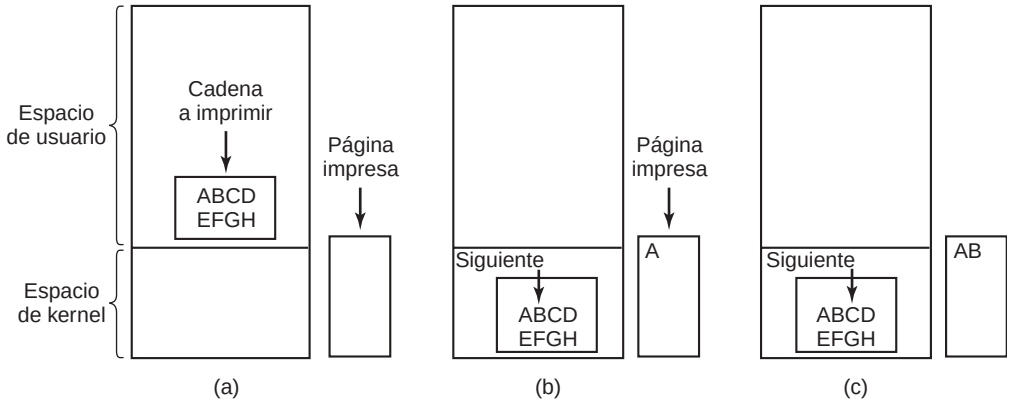

Figura 5-7. Pasos para imprimir una cadena.

fallará y devolverá un código de error o se bloqueará hasta que la impresora esté disponible, dependiendo del sistema operativo y los parámetros de la llamada. Una vez que obtiene la impresora, el proceso de usuario hace una llamada al sistema para indicar al sistema operativo que imprima la cadena en la impresora.

Después, el sistema operativo por lo general copia el búfer con la cadena a un arreglo, por ejemplo, *p* en espacio de kernel, donde se puede utilizar con más facilidad (debido a que el kernel tal vez tenga que modificar el mapa de memoria para tener acceso al espacio de usuario). Después comprueba si la impresora está disponible en ese momento. Si no lo está, espera hasta que lo esté. Tan pronto como la impresora está disponible, el sistema operativo copia el primer carácter al registro de datos de la impresora, en este ejemplo mediante el uso de E/S por asignación de memoria. Esta acción activa la impresora. El carácter tal vez no aparezca todavía, debido a que algunas impresoras colocan en búfer una línea o una página antes de imprimir algo. No obstante, en la figura 5-7(b) podemos ver que se ha impreso el primer carácter y que el sistema ha marcado a “B” como el siguiente carácter a imprimir.

Tan pronto como copia el primer carácter a la impresora, el sistema operativo comprueba si la impresora está lista para aceptar otro. En general la impresora tiene un segundo registro, que proporciona su estado. El acto de escribir en el registro de datos hace que el estado se convierta en “no está lista”. Cuando el controlador de la impresora ha procesado el carácter actual, indica su disponibilidad estableciendo cierto bit en su registro de estado, o colocando algún valor en él.

En este punto el sistema operativo espera a que la impresora vuelva a estar lista. Cuando eso ocurre, imprime el siguiente carácter, como se muestra en la figura 5-7(c). Este ciclo continúa hasta que se ha impreso toda la cadena. Después el control regresa al proceso de usuario.

Las acciones realizadas por el sistema operativo se sintetizan en la figura 5-8. Primero se copian los datos en el kernel. Después el sistema operativo entra a un ciclo estrecho, imprimiendo los caracteres uno a la vez. El aspecto esencial de la E/S programada, que se ilustra con claridad en esta figura, es que después de imprimir un carácter, la CPU sondea en forma continua el dispositivo

para ver si está listo para aceptar otro. Este comportamiento se conoce comúnmente como **sondeo** u **ocupado en espera**.

```
copiar_del_usuario(bufer, p, cuenta);                    /* p es el búfer del kernel */
for (i=0; i<cuenta; i++) {                               /* itera en cada carácter */
        while (*reg_estado_impresora != READY);          /* itera hasta que esté lista */
        *registro_datos_impresora = p[i];                /* imprime un carácter */
}
regresar_al_usuario();
```

Figura 5-8. Cómo escribir una cadena en la impresora usando E/S programada.

La E/S programada es simple, pero tiene la desventaja de ocupar la CPU tiempo completo hasta que se completen todas las operaciones de E/S. Si el tiempo para "imprimir" un carácter es muy corto (debido a que todo lo que hace la impresora es copiar el nuevo carácter a un búfer interno), entonces está bien usar ocupado en espera. Además, en un sistema incrustado o embebido, donde la CPU no tiene nada más que hacer, ocupado en espera es razonable. Sin embargo, en sistemas más complejos en donde la CPU tiene otros trabajos que realizar, ocupado en espera es ineficiente. Se necesita un mejor método de E/S.

5.2.3 E/S controlada por interrupciones

Ahora vamos a considerar el caso de imprimir en una impresora que no coloca los caracteres en un búfer, sino que imprime cada uno a medida que va llegando. Si la impresora puede imprimir (por ejemplo,) 100 caracteres/seg, cada carácter requiere 10 mseg para imprimirse. Esto significa que después de escribir cada carácter en el registro de datos de la impresora, la CPU estará en un ciclo de inactividad durante 10 mseg, esperando a que se le permita imprimir el siguiente carácter. Este tiempo es más que suficiente para realizar un cambio de contexto y ejecutar algún otro proceso durante los 10 mseg que, de otra manera, se desperdiciarían.

La forma de permitir que la CPU haga algo más mientras espera a que la impresora esté lista es utilizar interrupciones. Cuando se realiza la llamada al sistema para imprimir la cadena, el búfer se copia en espacio de kernel (como vimos antes) y el primer carácter se copia a la impresora, tan pronto como esté dispuesta para aceptar un carácter. En ese momento, la CPU llama al planificador y se ejecuta algún otro proceso. El proceso que pidió imprimir la cadena se bloquea hasta que se haya impreso toda la cadena. El trabajo realizado en la llamada al sistema se muestra en la figura 5-9(a).

Cuando la impresora ha impreso el carácter, y está preparada para aceptar el siguiente, genera una interrupción. Esta interrupción detiene el proceso actual y guarda su estado. Después se ejecuta el procedimiento de servicio de interrupciones de la impresora. Una versión cruda de este código se muestra en la figura 5-9(b). Si no hay más caracteres por imprimir, el manejador de interrupciones realiza cierta acción para desbloquear al usuario. En caso contrario, imprime el siguiente carácter, reconoce la interrupción y regresa al proceso que se estaba ejecutando justo antes de la interrupción, que continúa desde donde se quedó.

```
copiar_del_usuario(bufer, p, cuenta);
habilitar_interrupciones();
while (*reg_estado_impresora != READY);
*registro_datos_impresora = p[0];
planificador();
```

(a)

```
if (cuenta==0) {
   desbloquear_usuario();
} else {
   *registro_datos_impresora = p[i];
   cuenta=cuenta – 1;
   i = i + 1;
}
reconocer_interrupcion();
regresar_de_interrupcion();
```

(b)

Figura 5-9. Cómo escribir una cadena a la impresora, usando E/S controlada por interrupciones. (a) Código que se ejecuta al momento en que se hace una llamada al sistema para imprimir. (b) Procedimiento de servicio de interrupciones para la impresora.

5.2.4 E/S mediante el uso de DMA

Una obvia desventaja de la E/S controlada por interrupciones es que ocurre una interrupción en cada carácter. Las interrupciones requieren tiempo, por lo que este esquema desperdicia cierta cantidad de tiempo de la CPU. Una solución es utilizar DMA. Aquí la idea es permitir que el controlador de DMA alimente los caracteres a la impresora uno a la vez, sin que la CPU se moleste. En esencia, el DMA es E/S programada, sólo que el controlador de DMA realiza todo el trabajo en vez de la CPU principal. Esta estrategia requiere hardware especial (el controlador de DMA) pero libera la CPU durante la E/S para realizar otro trabajo. En la figura 5-10 se muestra un esquema del código.

```
copiar_del_usuario(bufer, p, cuenta);
establecer_controlador_DMA();
planificador();
```

```
reconocer_interrupcion();
desbloquear_usuario();
regresar_de_interrupcion();
```

Figura 5-10. Cómo imprimir una cadena mediante el uso de DMA. (a) Código que se ejecuta cuando se hace la llamada al sistema para imprimir. (b) Procedimiento de servicio de interrupciones.

La gran ganancia con DMA es reducir el número de interrupciones, de una por cada carácter a una por cada búfer impreso. Si hay muchos caracteres y las interrupciones son lentas, esto puede ser una gran mejora. Por otra parte, el controlador de DMA es comúnmente más lento que la CPU principal. Si el controlador de DMA no puede controlar el dispositivo a toda su velocidad, o si la CPU por lo general no tiene nada que hacer mientras espera la interrupción de DMA, entonces puede ser mejor utilizar la E/S controlada por interrupción o incluso la E/S programada. De todas formas, la mayor parte del tiempo vale la pena usar DMA.

5.3 CAPAS DEL SOFTWARE DE E/S

Por lo general, el software de E/S se organiza en cuatro capas, como se muestra en la figura 5-11. Cada capa tiene una función bien definida que realizar, y una interfaz bien definida para los niveles adyacentes. La funcionalidad y las interfaces difieren de un sistema a otro, por lo que el análisis que veremos a continuación, que examina todas las capas empezando desde el inferior, no es específico de una sola máquina.

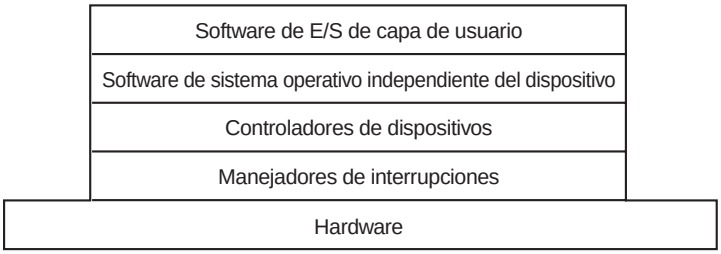

Figura 5-11. Capas del sistema de software de E/S.

5.3.1 Manejadores de interrupciones

Aunque la E/S programada es útil algunas veces, para la mayor parte de las operaciones de E/S las interrupciones son un hecho incómodo de la vida y no se pueden evitar. Deben ocultarse en la profundidad de las entrañas del sistema operativo, de manera que éste sepa lo menos posible de ellas. La mejor manera de ocultarlas es hacer que el controlador que inicia una operación de E/S se bloquee hasta que se haya completado la E/S y ocurra la interrupción. El controlador se puede bloquear a sí mismo realizando una llamada a `down` en un semáforo, una llamada a `wait` en una variable de condición, una llamada a `receive` en un mensaje o algo similar, por ejemplo.

Cuando ocurre la interrupción, el procedimiento de interrupciones hace todo lo necesario para poder manejarla. Después puede desbloquear el controlador que la inició. En algunos casos sólo completará `up` en un semáforo. En otros casos realizará una llamada a `signal` en una variable de condición en un monitor. En otros más enviará un mensaje al controlador bloqueado. En todos los casos, el efecto neto de la interrupción será que un controlador que estaba bloqueado podrá ejecutarse ahora. Este modelo funciona mejor si los controladores están estructurados como procesos del kernel, con sus propios estados, pilas y contadores del programa.

Desde luego que en realidad esto no es tan simple. Procesar una interrupción no es cuestión de sólo tomar la interrupción, llamar a `up` en algún semáforo y después ejecutar una instrucción `IRET` para regresar de la interrupción al proceso anterior. Hay mucho más trabajo involucrado para el sistema operativo. Ahora veremos un esquema de este trabajo como una serie de pasos que se deben llevar a cabo en el software, una vez que se haya completado la interrupción de hardware. Hay que recalcar que los detalles dependen mucho del sistema, por lo que algunos de los pasos que se listan

a continuación tal vez no sean necesarios en una máquina específica, y tal vez se requieran otros que no estén listados. Además, los pasos que se llevan a cabo pueden estar en distinto orden en algunas máquinas.

1. Guardar los registros (incluyendo el PSW) que no han sido guardados por el hardware de la interrupción.
2. Establecer un contexto para el procedimiento de servicio de interrupciones. Para ello tal vez sea necesario establecer el TLB, la MMU y una tabla de páginas.
3. Establecer una pila para el procedimiento de servicio de interrupciones.
4. Reconocer el controlador de interrupciones. Si no hay un controlador de interrupciones centralizado, rehabilitar las interrupciones.
5. Copiar los registros desde donde se guardaron (posiblemente en alguna pila) a la tabla de procesos.
6. Ejecutar el procedimiento de servicio de interrupciones. Éste extraerá información de los registros del controlador de dispositivos que provocó la interrupción.
7. Elegir cuál proceso ejecutar a continuación. Si la interrupción ha ocasionado que cierto proceso de alta prioridad que estaba bloqueado cambie al estado listo, puede elegirse para ejecutarlo en ese momento.
8. Establecer el contexto de la MMU para el proceso que se va a ejecutar a continuación. También puede ser necesario establecer un TLB.
9. Cargar los registros del nuevo proceso, incluyendo su PSW.
10. Empezar a ejecutar el nuevo proceso.

Como se puede ver, el procesamiento de interrupciones no carece de importancia. También ocupa un considerable número de instrucciones de la CPU, en especial en máquinas en las que hay memoria virtual y es necesario establecer tablas de páginas, o se tiene que guardar el estado de la MMU (por ejemplo, los bits *R* y *M*). En algunas máquinas, el TLB y la caché de la CPU tal vez también tengan que manejarse al cambiar entre los modos de usuario y de kernel, lo cual requiere ciclos de máquina adicionales.

5.3.2 Drivers de dispositivos

Al principio de este capítulo analizamos lo que hacen los drivers. Vimos que cada controlador tiene ciertos registros de dispositivos que se utilizan para darle comandos o ciertos registros de dispositivos que se utilizan para leer su estado, o ambos. El número de registros de dispositivos y la naturaleza de los comandos varían radicalmente de un dispositivo a otro. Por ejemplo, un driver de ratón tiene que aceptar información del ratón que le indica qué tanto se ha desplazado y cuáles botones están oprimidos en un momento dado. Por el contrario, un driver de disco tal vez tenga que saber todo acerca de los sectores, pistas, cilindros, cabezas, movimiento del brazo, los propulsores

del motor, los tiempos de asentamiento de las cabezas y todos los demás mecanismos para hacer que el disco funcione en forma apropiada. Obviamente, estos drivers serán muy distintos.

Como consecuencia, cada dispositivo de E/S conectado a una computadora necesita cierto código específico para controlarlo. Este código, conocido como **driver**, es escrito por el fabricante del dispositivo y se incluye junto con el mismo. Como cada sistema operativo necesita sus propios drivers, los fabricantes de dispositivos por lo común los proporcionan para varios sistemas operativos populares.

Cada driver maneja un tipo de dispositivo o, a lo más, una clase de dispositivos estrechamente relacionados. Por ejemplo, un driver de disco SCSI puede manejar por lo general varios discos SCSI de distintos tamaños y velocidades, y tal vez un CD-ROM SCSI también. Por otro lado, un ratón y una palanca de mandos son tan distintos que por lo general se requieren controladores diferentes. Sin embargo, no hay una restricción técnica en cuanto a que un driver controle varios dispositivos no relacionados. Simplemente no es una buena idea.

Para poder utilizar el hardware del dispositivo (es decir, los registros del controlador físico), el driver por lo general tiene que formar parte del kernel del sistema operativo, cuando menos en las arquitecturas actuales. En realidad es posible construir controladores que se ejecuten en el espacio de usuario, con llamadas al sistema para leer y escribir en los registros del dispositivo. Este diseño aísla al kernel de los controladores, y a un controlador de otro, eliminando una fuente importante de fallas en el sistema: controladores con errores que interfieren con el kernel de una manera u otra. Para construir sistemas altamente confiables, ésta es, en definitiva, la mejor manera de hacerlo. Un ejemplo de un sistema donde los controladores de dispositivos se ejecutan como procesos de usuario es MINIX 3. Sin embargo, como la mayoría de los demás sistemas operativos de escritorio esperan que los controladores se ejecuten en el kernel, éste es el modelo que consideraremos aquí.

Como los diseñadores de cada sistema operativo saben qué piezas de código (drivers) escritas por terceros se instalarán en él, necesita tener una arquitectura que permita dicha instalación. Esto implica tener un modelo bien definido de lo que hace un driver y la forma en que interactúa con el resto del sistema operativo. Por lo general, los controladores de dispositivos se posicionan debajo del resto del sistema operativo, como se ilustra en la figura 5-12.

Generalmente los sistemas operativos clasifican los controladores en una de un pequeño número de categorías. Las categorías más comunes son los **dispositivos de bloque** como los discos, que contienen varios bloques de datos que se pueden direccionar de manera independiente, y los **dispositivos de carácter** como los teclados y las impresoras, que generan o aceptan un flujo de caracteres.

La mayoría de los sistemas operativos definen una interfaz estándar que todos los controladores de bloque deben aceptar. Estas interfaces consisten en varios procedimientos que el resto del sistema operativo puede llamar para hacer que el controlador realice un trabajo para él. Los procedimientos ordinarios son los que se utilizan para leer un bloque (dispositivo de bloque) o escribir una cadena de caracteres (dispositivo de carácter).

En algunos sistemas, el sistema operativo es un solo programa binario que contiene compilados en él todos los controladores que requerirá. Este esquema fue la norma durante años con los sistemas UNIX, debido a que eran ejecutados por centros de computadoras y los dispositivos de E/S cambiaban pocas veces. Si se agregaba un nuevo dispositivo, el administrador del sistema simplemente recompilaba el kernel con el nuevo controlador para crear un nuevo binario.

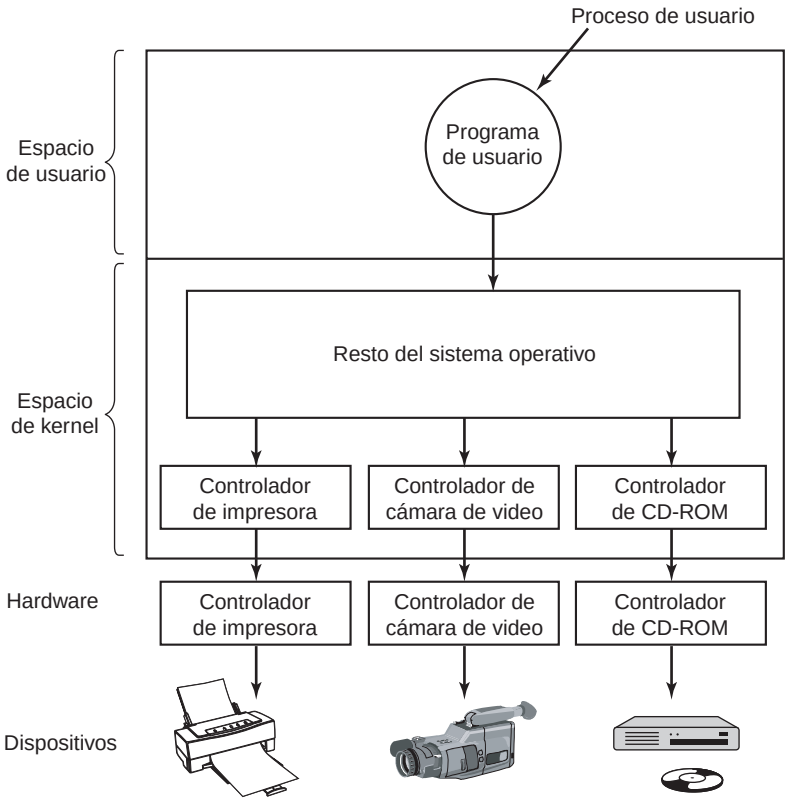

Figura 5-12. Posicionamiento lógico del software controlador de dispositivos. En realidad toda la comunicación entre el software controlador y los controladores de dispositivos pasa a través del bus.

Con la llegada de las computadoras personales y la multitud de dispositivos de E/S, este modelo ya no era funcional. Pocos usuarios son capaces de volver a compilar o vincular el kernel, aun si tienen el código fuente o los módulos de código objeto, que no siempre es el caso. En vez de ello, los sistemas operativos (empezando con MS-DOS) se inclinaron por un modelo en el que el controlador se carga en forma dinámica en el sistema durante la ejecución. Los diferentes sistemas manejan la carga de controladores en distintas formas.

Un controlador de dispositivo tiene varias funciones. La más obvia es aceptar peticiones abstractas de lectura y escritura del software independiente del dispositivo que está por encima de él, y ver que se lleven a cabo. Pero también hay otras tantas funciones que deben realizar. Por ejemplo, el controlador debe inicializar el dispositivo, si es necesario. También puede tener que administrar sus propios requerimientos y eventos del registro.

Muchos tipos de controladores de dispositivos tienen una estructura general similar. Un controlador ordinario empieza por comprobar los parámetros de entrada para ver si son válidos. De no ser así se devuelve un error. Si son válidos tal vez sea necesaria una traducción de términos

abstractos a concretos. Para un controlador de disco, esto puede significar tener que convertir un número de bloque lineal en los números de cabeza, pista, sector y cilindro para la geometría del disco.

A continuación, el controlador puede comprobar si el dispositivo se encuentra en uso. De ser así, la petición se pondrá en cola para procesarla después. Si el dispositivo está inactivo, el estado del hardware se examinará para ver si la petición se puede manejar en ese momento. Tal vez sea necesario encender el dispositivo o iniciar un motor para que puedan empezar las transferencias. Una vez que el dispositivo esté encendido y listo para trabajar, puede empezar el control en sí.

Controlar el dispositivo significa enviarle una secuencia de comandos. El controlador es el lugar en donde se determina la secuencia de comandos, dependiendo de lo que se deba hacer. Una vez que el controlador sabe qué comandos va a emitir, empieza a escribirlos en los registros de dispositivo del controlador. Después de escribir cada comando para el controlador, tal vez sea necesario comprobar si el controlador aceptó el comando y está preparado para aceptar el siguiente. Esta secuencia continúa hasta que se hayan emitido todos los comandos. Algunos controladores pueden recibir una lista de comandos (en memoria) y se les puede indicar que los procesen todos por sí mismos, sin necesidad de ayuda del sistema operativo.

Una vez que se han emitido los comandos, se dará una de dos situaciones. En muchos casos el controlador de dispositivos debe esperar hasta que el controlador realice cierto trabajo para él, por lo que se bloquea a sí mismo hasta que llegue la interrupción para desbloquearlo. Sin embargo, en otros casos la operación termina sin retraso, por lo que el controlador no necesita bloquearse. Como ejemplo de esta última situación, para desplazar la pantalla en modo de carácter se requiere escribir sólo unos cuantos bytes en los registros del controlador. No es necesario ningún tipo de movimiento mecánico, por lo que toda la operación se puede completar en nanosegundos.

En el primer caso, el controlador bloqueado será despertado por la interrupción. En el segundo caso nunca pasará al estado inactivo. De cualquier forma, una vez que se ha completado la operación, el controlador debe comprobar si hay errores. Si todo está bien, el controlador puede hacer que pasen datos al software independiente del dispositivo (por ejemplo, un bloque que se acaba de leer). Por último, devuelve cierta información de estado para reportar los errores de vuelta al que lo llamó. Si hay otras peticiones en la cola, ahora se puede seleccionar e iniciar una de ellas. Si no hay nada en la cola, el controlador se bloquea en espera de la siguiente petición.

Este modelo simple es sólo una aproximación a la realidad. Muchos factores hacen el código mucho más complicado. Por una parte, un dispositivo de E/S puede completar su operación mientras haya un controlador en ejecución, con lo cual se interrumpe el controlador. La interrupción puede hacer que se ejecute un controlador de dispositivo. De hecho, puede hacer que el controlador actual se ejecute. Por ejemplo, mientras el controlador de red está procesando un paquete entrante, puede llegar otro paquete. En consecuencia, un controlador tiene que ser **reentrante**, lo cual significa que un controlador en ejecución tiene que esperar a ser llamado una segunda vez antes de que se haya completado la primera llamada.

En un sistema con “conexión en caliente” es posible agregar o eliminar dispositivos mientras la computadora está en ejecución. Como resultado, mientras un controlador está ocupado leyendo de algún dispositivo, el sistema puede informarle que el usuario ha quitado de manera repentina ese dispositivo del sistema. No sólo se debe abortar la transferencia actual de E/S sin dañar ninguna estructura de datos del kernel, sino que cualquier petición pendiente del ahora desaparecido disposi-

tivo debe eliminarse también, con cuidado, del sistema, avisando a los que hicieron la llamada. Además, la adición inesperada de nuevos dispositivos puede hacer que el kernel haga malabares con los recursos (por ejemplo, las líneas de petición de interrupciones), quitando los anteriores al controlador y dándole nuevos recursos en vez de los otros.

El controlador no puede hacer llamadas al sistema, pero a menudo necesita interactuar con el resto del kernel. Por lo general se permiten llamadas a ciertos procedimientos del kernel. Por ejemplo, comúnmente hay llamadas para asignar y desasignar páginas fijas de memoria para usarlas como búferes. Otras llamadas útiles se necesitan para administrar la MMU, los temporizadores, el controlador de DMA, el controlador de interrupciones, etcétera.

5.3.3 Software de E/S independiente del dispositivo

Aunque parte del software de E/S es específico para cada dispositivo, otras partes de éste son independientes de los dispositivos. El límite exacto entre los controladores y el software independiente del dispositivo depende del sistema (y del dispositivo), debido a que ciertas funciones que podrían realizarse de una manera independiente al dispositivo pueden realizarse en los controladores, por eficiencia u otras razones. Las funciones que se muestran en la figura 5-13 se realizan comúnmente en el software independiente del dispositivo.

Interfaz uniforme para controladores de dispositivos
Uso de búfer
Reporte de errores
Asignar y liberar dispositivos dedicados
Proporcionar un tamaño de bloque independiente del dispositivo

Figura 5-13. Funciones del software de E/S independiente del dispositivo.

La función básica del software independiente del dispositivo es realizar las funciones de E/S que son comunes para todos los dispositivos y proveer una interfaz uniforme para el software a nivel de usuario. A continuación analizaremos las cuestiones antes mencionadas con más detalle.

Interfaz uniforme para los controladores de software de dispositivos

Una importante cuestión en un sistema operativo es cómo hacer que todos los dispositivos de E/S y sus controladores se vean más o menos iguales. Si los discos, las impresoras, los teclados, etc., se conectan de distintas maneras, cada vez que llegue un nuevo dispositivo el sistema operativo deberá modificarse para éste. No es conveniente tener que modificar el sistema operativo para cada nuevo dispositivo.

Un aspecto de esta cuestión es la interfaz entre los controladores de dispositivos y el resto del sistema operativo. En la figura 5-14(a) ilustramos una situación en la que cada controlador de dis-

positivo tiene una interfaz distinta con el sistema operativo. Lo que esto significa es que las funciones de controlador disponibles para que el sistema pueda llamarlas difieren de un controlador a otro. Además, podría significar que las funciones de kernel que el controlador necesita también difieren de controlador en controlador. En conjunto, quiere decir que la interfaz para cada nuevo controlador requiere mucho esfuerzo de programación.

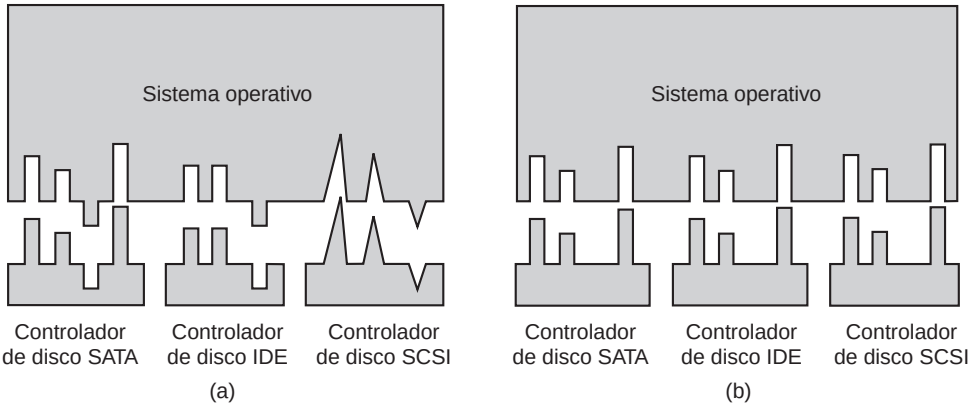

Figura 5-14. (a) Sin una interfaz de controlador estándar. (b) Con una interfaz de controlador estándar.

Por el contrario, en la figura 5-14(b) podemos ver un diseño distinto, en el que todos los controladores tienen la misma interfaz. Ahora es mucho más fácil conectar un nuevo controlador, siempre y cuando sea compatible con su interfaz. Esto también significa que los escritores de los controladores saben lo que se espera de ellos. En la práctica no todos los dispositivos son absolutamente idénticos, pero por lo general hay sólo un pequeño número de tipos de dispositivos, e incluso éstos en general son casi iguales.

La manera en que funciona es la siguiente. Para cada clase de dispositivos, como los discos o las impresoras, el sistema operativo define un conjunto de funciones que el controlador debe proporcionar. Para un disco, estas funciones incluyen naturalmente la lectura y la escritura, pero también encender y apagar la unidad, aplicar formato y otras cosas relacionadas con los discos. A menudo, el controlador contiene una tabla con apuntadores a sí mismo para estas funciones. Cuando se carga el controlador, el sistema operativo registra la dirección de esta tabla de apuntadores a funciones, por lo que cuando necesita llamar a una de las funciones, puede hacer una llamada indirecta a través de esta tabla. Esta tabla de apuntadores a funciones define la interfaz entre el controlador y el resto del sistema operativo. Todos los dispositivos de una clase dada (discos, impresoras, etc.) deben obedecerla.

Otro aspecto de tener una interfaz uniforme es la forma en que se nombran los dispositivos. El software independiente del dispositivo se hace cargo de asignar nombres de dispositivo simbólicos al controlador apropiado. Por ejemplo, en UNIX el nombre de un dispositivo como */dev/disk0* especifica de manera única el nodo-i para un archivo especial, y este nodo-i contiene el **número mayor de dispositivo**, que se utiliza para localizar el controlador apropiado. El nodo-i también contiene

el **número menor de dispositivo**, que se pasa como un parámetro al controlador para poder especificar la unidad que se va a leer o escribir. Todos los dispositivos tienen números mayores y menores, y para acceder a todos los controladores se utiliza el número mayor de dispositivo para seleccionar el controlador.

La protección está muy relacionada con la denominación. ¿Cómo evita el sistema que los usuarios accedan a dispositivos que tienen prohibido utilizar? Tanto en UNIX como en Windows, los dispositivos aparecen en el sistema de archivos como objetos con nombre, lo cual significa que las reglas de protección ordinarias para los archivos también se aplican a los dispositivos de E/S. Así, el administrador del sistema puede establecer los permisos apropiados para cada dispositivo.

Uso de búfer

El uso de búfer es otra cuestión, tanto para los dispositivos de bloque como los de carácter, por una variedad de razones. Para ver una de ellas, considere un proceso que desea leer datos de un módem. Una posible estrategia para lidiar con los caracteres entrantes es hacer que el proceso de usuario realice una llamada al sistema `read` y se bloquee en espera de un carácter. Cada carácter que llega produce una interrupción. El procedimiento de servicio de interrupciones entrega el carácter al proceso de usuario y lo desbloquea. Después de colocar el carácter en alguna parte, el proceso lee otro carácter y se bloquea de nuevo. Este modelo se indica en la figura 5-15(a).

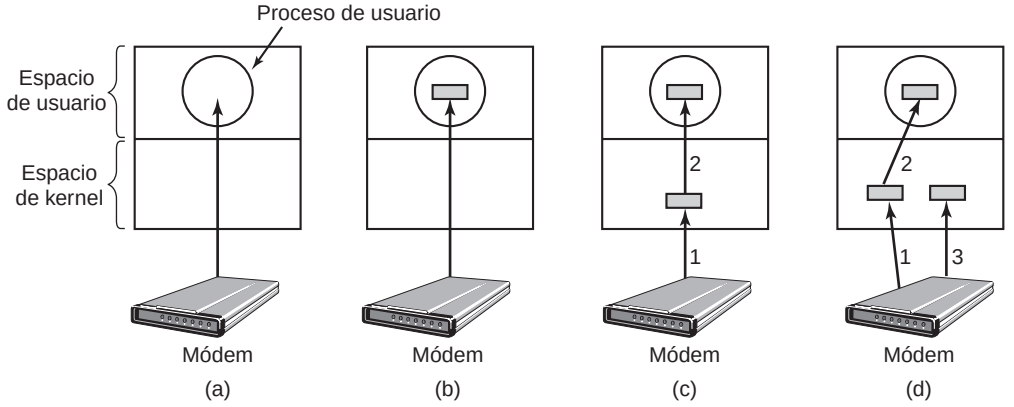

Figura 5-15. (a) Entrada sin búfer. (b) Uso de búfer en espacio de usuario. (c) Uso de búfer en el kernel, seguido de la acción de copiar al espacio de usuario. (d) Uso de doble búfer en el kernel.

El problema con esta forma de hacer las cosas es que el proceso de usuario tiene que iniciarse por cada carácter entrante. Permitir que un proceso se ejecute muchas veces durante tiempos cortos de ejecución es ineficiente, por lo que este diseño no es bueno.

En la figura 5-15(b) se muestra una mejora. Aquí el proceso de usuario proporciona un búfer de *n* caracteres en espacio de usuario y realiza una lectura de *n* caracteres. El procedimiento de servicio

de interrupciones coloca los caracteres entrantes en este búfer hasta que se llena. Después despierta al proceso de usuario. Este esquema es mucho más eficiente que el anterior, pero tiene una desventaja: ¿qué ocurre si el búfer se pagina cuando llega un carácter? El búfer se podría bloquear en la memoria, pero si muchos procesos empiezan a bloquear páginas en memoria, se reducirá la reserva de páginas disponibles y se degradará el rendimiento.

Otro método más es crear un búfer dentro del kernel y hacer que el manejador de interrupciones coloque ahí los caracteres, como se muestra en la figura 5-15(c). Cuando este búfer está lleno, se trae la página con el búfer de usuario en caso de ser necesario, y el búfer se copia ahí en una operación. Este esquema es mucho más eficiente.

Sin embargo, incluso este esquema presenta un inconveniente: ¿Qué ocurre con los caracteres que llegan mientras se está trayendo del disco la página con el búfer de usuario? Como el búfer está lleno, no hay lugar en dónde ponerlos. Una solución es tener un segundo búfer de kernel. Una vez que se llena el primero, pero antes de que se haya vaciado, se utiliza el segundo como se muestra en la figura 5-15(d). Cuando se llena el segundo búfer, está disponible para copiarlo al usuario (suponiendo que el usuario lo haya pedido). Mientras el segundo búfer se copia al espacio de usuario, el primero se puede utilizar para nuevos caracteres. De esta forma, los dos búferes toman turnos: mientras uno se copia al espacio de usuario, el otro está acumulando nuevos datos de entrada. Un esquema de uso de búferes como éste se conoce como **uso de doble búfer**.

Otra forma de uso de búfer que se utiliza ampliamente es el **búfer circular**. Consiste en una región de memoria y dos apuntadores. Un apuntador apunta a la siguiente palabra libre, en donde se pueden colocar nuevos datos. El otro apuntador apunta a la primera palabra de datos en el búfer que todavía no se ha removido. En muchas situaciones, el hardware avanza el primer apuntador a medida que agrega nuevos datos (por ejemplo, los que acaban de llegar de la red) y el sistema operativo avanza el segundo apuntador a medida que elimina y procesa datos. Ambos apuntadores regresan a la parte final después de llegar a la parte superior.

El uso de búfer es también importante en la salida. Por ejemplo, considere cómo se envían datos al módem sin uso de búfer mediante el modelo de la figura 5-15(b). El proceso de usuario ejecuta una llamada al sistema `write` para imprimir *n* caracteres. El sistema tiene dos opciones en este punto. Puede bloquear el usuario hasta que se hayan escrito todos los caracteres, pero se podría requerir mucho tiempo a través de una línea telefónica lenta. También podría liberar de inmediato al usuario y realizar la operación de E/S mientras el usuario realiza unos cálculos más, pero esto produce un problema aún peor: ¿Cómo sabe el proceso de usuario que se ha completado la salida y puede reutilizar el búfer? El sistema podría generar una señal o interrupción de software, pero ese estilo de programación es difícil y está propenso a condiciones de competencia. Una mucho mejor solución es que el kernel copie los datos a un búfer, de manera similar a la figura 5-15(c) (pero en sentido opuesto), y que desbloquee al que hizo la llamada de inmediato. Entonces no importa cuándo se haya completado la operación de E/S. El usuario es libre de reutilizar el búfer al instante en que se desbloquea.

Los búferes constituyen una técnica muy utilizada; ésta también tiene una desventaja. Si los datos se colocan en búfer demasiadas veces, el rendimiento se reduce. Por ejemplo, considere la red de la figura 5-16. Aquí un usuario realiza una llamada al sistema para escribir en la red. El kernel copia el paquete a un búfer de kernel para permitir que el usuario proceda de inmediato (paso 1). En este punto, el programa de usuario puede reutilizar el búfer.

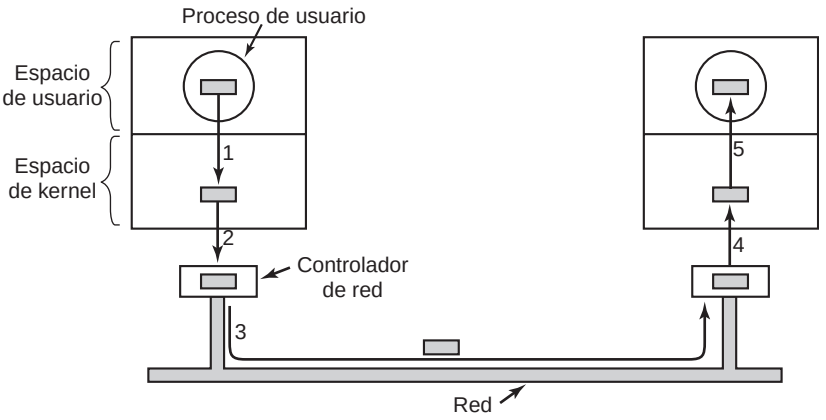

Figura 5-16. El trabajo en red puede involucrar muchas copias de un paquete.

Cuando se hace la llamada al controlador, copia el paquete al controlador para enviarlo como salida (paso 2). La razón por la que no envía directamente al cable desde la memoria kernel es que una vez iniciada la transmisión del paquete, debe continuar a una velocidad uniforme. El controlador no puede garantizar que pueda llegar a la memoria a una velocidad uniforme, debido a que los canales de DMA y otros dispositivos de E/S pueden estar robando muchos ciclos. Si no se obtiene una palabra a tiempo se podría arruinar el paquete. Al colocar en búfer el paquete dentro del controlador se evita este problema.

Una vez que se ha copiado el paquete en el búfer interno del controlador, se copia hacia la red (paso 3). Los bits llegan al receptor poco después de haber sido enviados, por lo que justo después de enviar el último bit, éste llega al receptor, donde el paquete se ha colocado en un búfer en el controlador. A continuación, el paquete se copia al búfer del kernel del receptor (paso 4). Por último se copia al búfer del proceso receptor (paso 5); generalmente, el receptor envía después un reconocimiento. Cuando el emisor recibe el reconocimiento, puede enviar el segundo paquete. Sin embargo, debe quedar claro que todo este proceso de copia reduce la velocidad de transmisión de manera considerable, ya que todos los pasos se deben llevar a cabo en forma secuencial.

Reporte de errores

Los errores son mucho más comunes en el contexto de la E/S que en otros. Cuando ocurren, el sistema operativo debe manejarlos de la mejor manera posible. Muchos errores son específicos de cada dispositivo y el controlador apropiado debe manejarlos, pero el marco de trabajo para el manejo de errores es independiente del dispositivo.

Los errores de programación son una clase de errores de E/S. Éstos ocurren cuando un proceso pide algo imposible, como escribir en un dispositivo de entrada (teclado, escáner, ratón, etc.) o leer de un dispositivo de salida (una impresora o un plotter, por ejemplo). Otros errores son propor-

cionar una dirección de búfer inválida o algún otro parámetro, y especificar un dispositivo inválido (por ejemplo, el disco 3 cuando el sistema sólo tiene dos), entre otros. La acción a tomar en estos errores es simple: sólo se reporta un código de error al que hizo la llamada.

Otra clase de errores son los de E/S reales; por ejemplo, tratar de escribir un bloque de disco dañado o tratar de leer de una cámara de video que está apagada. En estas circunstancias depende del controlador determinar qué hacer. Si el controlador no sabe qué hacer, puede pasar el problema de vuelta al software independiente del dispositivo.

Lo que hace este software depende del entorno y la naturaleza del error. Si se trata de un error simple de lectura y hay un usuario interactivo disponible, puede mostrar un cuadro de diálogo pidiendo al usuario lo que debe hacer. Las opciones pueden incluir volver a intentar cierto número de veces, ignorar el error o eliminar el proceso que hizo la llamada. Si no hay usuario disponible, tal vez la única opción real sea hacer que la llamada al sistema falle con un código de error.

Sin embargo, algunos errores no se pueden manejar de esta forma. Por ejemplo, una estructura de datos crítica, como el directorio raíz o la lista de bloques libres, puede haberse destruido. En este caso, el sistema tal vez tenga que mostrar un mensaje de error y terminar.

Asignación y liberación de dispositivos dedicados

Algunos dispositivos, como los grabadores de CD-ROM, sólo pueden ser utilizados por un solo proceso en un momento dado. Es responsabilidad del sistema operativo examinar las peticiones de uso de los dispositivos y aceptarlas o rechazarlas, dependiendo de si el dispositivo solicitado está o no disponible. Una manera simple de manejar estas peticiones es requerir que los procesos realicen llamadas a `open` en los archivos especiales para los dispositivos directamente. Si el dispositivo no está disponible, la llamada a `open` falla. Al cerrar un dispositivo dedicado de este tipo se libera.

Un método alternativo es tener mecanismos especiales para solicitar y liberar dispositivos dedicados. Un intento por adquirir un dispositivo que no está disponible bloquea al proceso que hizo la llamada, en vez de fallar. Los procesos bloqueados se ponen en una cola. Tarde o temprano, el dispositivo solicitado estará disponible y el primer proceso en la cola podrá adquirirlo para continuar su ejecución.

Tamaño de bloque independiente del dispositivo

Los distintos discos pueden tener diferentes tamaños de sectores. Es responsabilidad del software independiente del dispositivo ocultar este hecho y proporcionar un tamaño de bloque uniforme a los niveles superiores; por ejemplo, al tratar varios sectores como un solo bloque lógico. De esta forma, los niveles superiores lidian sólo con dispositivos abstractos que utilizan todos el mismo tamaño de bloque lógico, sin importar el tamaño del sector físico. De manera similar, algunos dispositivos de carácter envían sus datos un byte a la vez (como los módems), mientras que otros envían sus datos en unidades más grandes (como las interfaces de red). Estas diferencias también se pueden ocultar.

5.3.4 Software de E/S en espacio de usuario

Aunque la mayor parte del software de E/S está dentro del sistema operativo, una pequeña porción de éste consiste en bibliotecas vinculadas entre sí con programas de usuario, e incluso programas enteros que se ejecutan desde el exterior del kernel. Las llamadas al sistema, incluyendo las llamadas al sistema de E/S, se realizan comúnmente mediante procedimientos de biblioteca. Cuando un programa en C contiene la llamada

```
cuenta = write(da, bufer, nbytes);
```

el procedimiento de biblioteca *write* se vinculará con el programa y se incluirá en el programa binario presente en memoria en tiempo de ejecución. La colección de todos estos procedimientos de biblioteca es sin duda parte del sistema de E/S.

Aunque estos procedimientos hacen algo más que colocar sus parámetros en el lugar apropiado para la llamada al sistema, hay otros procedimientos de E/S que en realidad realizan un trabajo real. En especial, el formato de la entrada y la salida se lleva a cabo mediante procedimientos de biblioteca. Un ejemplo de C es *printf*, que toma una cadena de formato y posiblemente unas variables como entrada, construye una cadena ASCII y después llama al sistema `write` para imprimir la cadena. Como ejemplo de *printf*, considere la instrucción

```
printf("El cuadrado de %3d es %6d\n", i, i*i);
```

Esta instrucción da formato a una cadena que consiste en la cadena de 14 caracteres "El cuadrado de" seguida por el valor *i* como una cadena de 3 caracteres, después la cadena de 4 caracteres "es", luego i^2 como seis caracteres, y por último un salto de línea.

Un ejemplo de un procedimiento similar para la entrada es *scanf*, que lee los datos de entrada y los almacena en variables descritas en una cadena de formato que utiliza la misma sintaxis que *printf*. La biblioteca de E/S estándar contiene varios procedimientos que involucran operaciones de E/S y todos se ejecutan como parte los programas de usuario.

No todo el software de E/S de bajo nivel consiste en procedimientos de biblioteca. Otra categoría importante es el sistema de colas. El **uso de colas** (*spooling*) es una manera de lidiar con los dispositivos de E/S dedicados en un sistema de multiprogramación. Considere un dispositivo común que utiliza colas: una impresora. Aunque sería técnicamente sencillo dejar que cualquier proceso de usuario abriera el archivo de caracteres especial para la impresora, suponga que un proceso lo abriera y no hiciera nada durante horas. Ningún otro proceso podría imprimir nada.

En vez de ello, lo que se hace es crear un proceso especial, conocido como **demonio**, y un directorio especial llamado **directorio de cola de impresión**. Para imprimir un archivo, un proceso genera primero todo el archivo que va a imprimir y lo coloca en el directorio de la cola de impresión. Es responsabilidad del demonio, que es el único proceso que tiene permiso para usar el archivo especial de la impresora, imprimir los archivos en el directorio. Al proteger el archivo especial contra el uso directo por parte de los usuarios, se elimina el problema de que alguien lo mantenga abierto por un tiempo innecesariamente extenso.

El uso de colas no es exclusivo de las impresoras. También se utiliza en otras situaciones de E/S. Por ejemplo, la transferencia de archivos a través de una red utiliza con frecuencia un demonio de red. Para enviar un archivo a cierta parte, un usuario lo coloca en un directorio de la cola de

red. Más adelante, el demonio de red lo toma y lo transmite. Un uso específico de la transmisión de archivos mediante el uso de una cola es el sistema de noticias USENET. Esta red consiste en millones de máquinas en todo el mundo, que se comunican mediante Internet. Existen miles de grupos de noticias sobre muchos temas. Para publicar un mensaje, el usuario invoca a un programa de noticias, el cual acepta el mensaje a publicar y luego lo deposita en un directorio de cola para transmitirlo a las otras máquinas más adelante. Todo el sistema de noticias se ejecuta fuera del sistema operativo.

En la figura 5-17 se resume el sistema de E/S, donde se muestran todos los niveles y las funciones principales de cada nivel. Empezando desde la parte inferior, los niveles son el hardware, los manejadores de interrupciones, los controladores de dispositivos, el software independiente del dispositivo y, por último, los procesos de usuario.

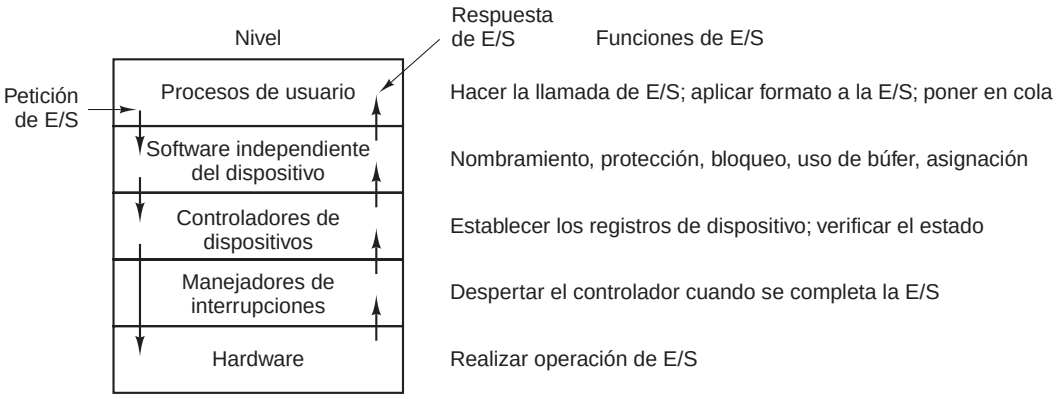

Figura 5-17. Niveles del sistema de E/S y las funciones principales de cada nivel.

Las flechas en la figura 5-17 muestran el flujo de control. Por ejemplo, cuando un programa de usuario trata de leer un bloque de un archivo, se invoca el sistema operativo para llevar a cabo la llamada. El software independiente del dispositivo busca el bloque en la caché del búfer, por ejemplo. Si el bloque necesario no está ahí, llama al controlador del dispositivo para enviar la petición al hardware y obtenerlo del disco. Después, el proceso se bloquea hasta que se haya completado la operación de disco.

Cuando termina el disco, el hardware genera una interrupción. El manejador de interrupciones se ejecuta para descubrir qué ocurrió; es decir, qué dispositivo desea atención en ese momento. Después extrae el estado del dispositivo y despierta al proceso inactivo para que termine la petición de E/S y deje que el proceso de usuario continúe.

5.4 DISCOS

Ahora vamos a estudiar algunos dispositivos de E/S reales. Empezaremos con los discos, que en concepto son simples, pero muy importantes. Después examinaremos los relojes, los teclados y las pantallas.

5.4.1 Hardware de disco

Los discos son de varios tipos. Los más comunes son los discos magnéticos (discos duros y flexibles). Se caracterizan por el hecho de que las operaciones de lectura y escritura son igual de rápidas, lo que los hace ideales como memoria secundaria (como paginación o sistemas de archivos, por ejemplo). Algunas veces se utilizan arreglos de estos discos para ofrecer un almacenamiento altamente confiable. Para la distribución de programas, datos y películas, son también importantes varios tipos de discos ópticos (CD-ROMs, CD-grabable y DVD). En las siguientes secciones describiremos primero el hardware y luego el software para estos dispositivos.

Discos magnéticos

Los discos magnéticos se organizan en cilindros, cada uno de los cuales contiene tantas pistas como cabezas apiladas en forma vertical. Las pistas se dividen en sectores. El número de sectores alrededor de la circunferencia es por lo general de 8 a 32 en los discos flexibles, y hasta varios cientos en los discos duros. El número de cabezas varía entre 1 y 16.

Los discos antiguos tienen pocos componentes electrónicos y sólo producen un flujo de bits serial simple. En estos discos el controlador realiza la mayor parte del trabajo. En otros discos, en especial los discos **IDE** (**Electrónica de Unidad Integrada**) y **SATA** (**ATA Serial**), la unidad de disco contiene un microcontrolador que realiza un trabajo considerable y permite al controlador real emitir un conjunto de comandos de nivel superior. A menudo el controlador coloca las pistas en caché, reasigna los bloques defectuosos y mucho más.

Una característica de dispositivo que tiene implicaciones importantes para el software controlador del disco es la posibilidad de que un controlador realice búsquedas en dos o más unidades al mismo tiempo. Éstas se conocen como **búsquedas traslapadas**. Mientras el controlador y el software esperan a que se complete una búsqueda en una unidad, el controlador puede iniciar una búsqueda en otra unidad. Muchos controladores también pueden leer o escribir en una unidad mientras buscan en otra u otras unidades, pero un controlador de disco flexible no puede leer o escribir en dos unidades al mismo tiempo (para leer o escribir, el controlador tiene que desplazar bits en una escala de tiempo en microsegundos, por lo que una transferencia ocupa la mayor parte de su poder de cómputo). Esta situación es distinta para los discos duros con controladores integrados, y en un sistema con más de una de estas unidades de disco duro pueden operar de manera simultánea, al menos en cuanto a la transferencia de datos entre el disco y la memoria de búfer del controlador. Sin embargo, sólo es posible una transferencia entre el controlador y la memoria principal. La capacidad de realizar dos o más operaciones al mismo tiempo puede reducir el tiempo de acceso promedio de manera considerable.

En la figura 5-18 se comparan los parámetros del medio de almacenamiento estándar para la IBM PC original con los parámetros de un disco fabricado 20 años después, para mostrar cuánto han cambiado los discos en 20 años. Es interesante observar que no todos los parámetros han mejorado tanto. El tiempo de búsqueda promedio es siete veces mejor de lo que era antes, la velocidad de transferencia es 1300 veces mejor, mientras que la capacidad aumentó por un factor de

50,000. Este patrón está relacionado con las mejoras relativamente graduales en las piezas móviles, y densidades de bits mucho mayores en las superficies de grabación.

Parámetro	Disco flexible IBM de 360-KB	Disco duro WD 18300
Número de cilindros	40	10601
Pistas por cilindro	2	12
Sectores por pista	9	281 (promedio)
Sectores por disco	720	35742000
Bytes por sector	512	512
Capacidad del disco	360 KB	18.3 GB
Tiempo de búsqueda (cilindros adyacentes)	6 mseg	0.8 mseg
Tiempo de búsqueda (caso promedio)	77 mseg	6.9 mseg
Tiempo de rotación	200 mseg	8.33 mseg
Tiempo de arranque/paro del motor	250 mseg	20 seg
Tiempo para transferir 1 sector	22 mseg	17 µseg

Figura 5-18. Parámetros de disco para el disco flexible IBM PC 360 KB original y un disco duro Western Digital WD 18300.

Algo que debemos tener en cuenta al analizar las especificaciones de los discos duros modernos es que la geometría especificada y utilizada por el controlador es casi siempre distinta a la del formato físico. En los discos antiguos, el número de sectores por pista era el mismo para todos los cilindros. Los discos modernos se dividen en zonas, con más sectores en las zonas exteriores que en las interiores. La figura 5-19(a) ilustra un pequeño disco con dos zonas. La zona exterior tiene 32 sectores por pista; la interior tiene 16 sectores por pista. Un disco real, como el WD 18300, tiene por lo general 16 o más zonas, y el número de sectores se incrementa aproximadamente 4% por zona, a medida que se avanza desde la zona más interior hasta la más exterior.

Para ocultar los detalles sobre cuántos sectores tiene cada pista, la mayoría de los discos modernos tienen una geometría virtual que se presenta al sistema operativo. Se instruye al software para que actúe como si hubiera *x* cilindros, *y* cabezas y *z* sectores por pista. Después el controlador reasigna una petición para (*x*, *y*, *z*) a los valores reales de cilindro, cabeza y sector. Una posible geometría virtual para el disco físico de la figura 5-19(a) se muestra en la figura 5-19(b). En ambos casos el disco tiene 192 sectores, sólo que el arreglo publicado es distinto del real.

Para las PCs, los valores máximos para estos parámetros son a menudo 5535, 16 y 63, debido a la necesidad de tener compatibilidad hacia atrás con las limitaciones de la IBM PC original. En esta máquina se utilizaron campos de 16, 4 y 6 bits para especificar estos números, donde los cilindros y sectores enumerados empiezan en 1 y las cabezas enumeradas empiezan en 0. Con estos parámetros y 512 bytes por sector, el disco más grande posible es de 31.5 GB. Para sobrepasar este límite, todos los discos modernos aceptan ahora un sistema llamado **direccionamiento de bloques**

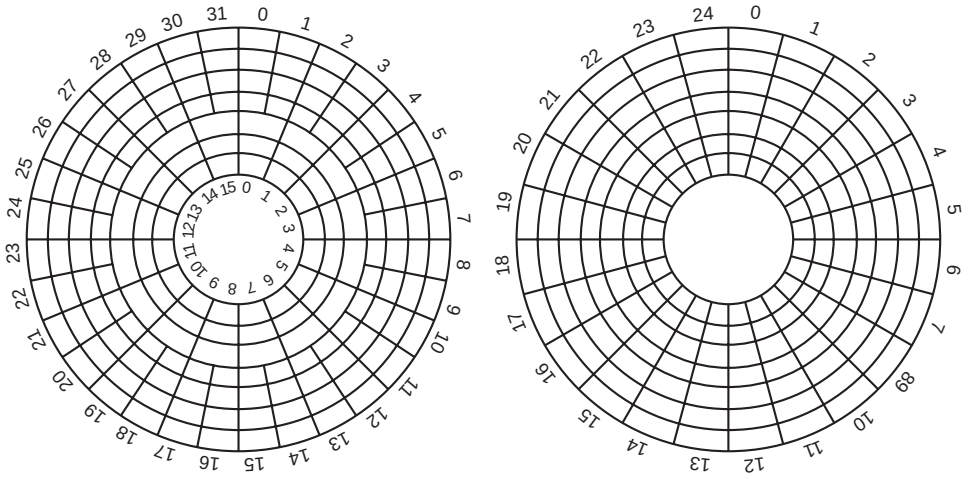

Figura 5-19. (a) Geometría física de un disco con dos zonas. (b) Una posible geometría virtual para este disco.

lógicos, en el que los sectores de disco sólo se enumeran en forma consecutiva empezando en 0, sin importar la geometría del disco.

RAID

El rendimiento de la CPU ha crecido en forma exponencial durante la década pasada, y se duplica aproximadamente cada 18 meses. No pasa lo mismo con el rendimiento del disco. En la década de 1970, los tiempos de búsqueda promedio en los discos de minicomputadoras eran de 50 a 100 mseg; ahora están ligeramente por debajo de 10 mseg. En la mayoría de las industrias técnicas (por ejemplo, automóviles o aviación), un factor del 5 a 10 en cuanto a la mejora del rendimiento en dos décadas serían grandes noticias (imagine autos de 300 millas por galón), pero en la industria de las computadoras es una vergüenza. Por ende, la brecha entre el rendimiento de la CPU y el rendimiento del disco se ha vuelto mucho mayor con el tiempo.

Como hemos visto, el procesamiento en paralelo se utiliza cada vez más para agilizar el rendimiento de la CPU. Con el paso de los años, a varias personas se les ha ocurrido que la E/S en paralelo podría ser una buena idea también. En su artículo de 1988, Patterson y colaboradores sugirieron seis organizaciones de discos específicas que se podrían utilizar para mejorar el rendimiento del disco, su confiabilidad o ambas características (Patterson y colaboradores, 1988). Estas ideas fueron adoptadas de inmediato por la industria y han conllevado a una nueva clase de dispositivo de E/S conocido como **RAID**. Patterson y sus colaboradores definieron RAID como **Arreglo Redundante de Discos Económicos** (*Redundant Array of Inexpensive Disks*), pero la industria redefinió la I para que indicara “Independiente” en vez de “económico” (¿tal vez para que pudieran cobrar más?). Como también se necesitaba un villano (como en RISC contra CISC, también debido a Patterson), el tipo malo aquí era **SLED** (*Single Large Expensive Disk*, Un solo disco grande y costoso).

La idea básica detrás de un RAID es instalar una caja llena de discos a un lado de la computadora (que por lo general es un servidor grande), reemplazar la tarjeta controladora de discos con un controlador RAID, copiar los datos al RAID y después continuar la operación normal. En otras palabras, un RAID se debe ver como un SLED para el sistema operativo, pero con mejor rendimiento y confiabilidad. Como los discos SCSI tienen un buen rendimiento, bajo costo y la capacidad de tener hasta siete unidades en un solo controlador (15 para SCSI amplio), es natural que la mayoría de los RAIDs consistan en un controlador RAID SCSI más una caja de discos SCSI que el sistema operativo considere como un solo disco grande. De esta forma no se requieren cambios en el software para utilizar el RAID; un gran punto de venta para muchos administradores de sistemas.

Además de aparecer como un solo disco para el software, todos los RAIDs tienen la propiedad de que los datos se distribuyen entre las unidades, para permitir la operación en paralelo. Patterson y sus colaboradores definieron varios esquemas distintos para hacer esto, y ahora se conocen como RAID nivel 0 hasta RAID nivel 5. Además hay unos cuantos niveles menores que no analizaremos. El término "nivel" es algo equivocado, debido a que no hay una jerarquía involucrada; simplemente son seis organizaciones distintas posibles.

El nivel RAID 0 se ilustra en la figura 5-20(a). Consiste en ver el disco virtual simulado por el RAID como si estuviera dividido en bandas de k sectores cada una, en donde los sectores 0 a $k - 1$ son la banda 0, los sectores k a $2k - 1$ son la banda 1, y así en lo sucesivo. Para $k = 1$, cada banda es un sector; para $k = 2$, una banda es de dos sectores, etc. La organización RAID de nivel 0 escribe bandas consecutivas sobre las unidades utilizando el método por turno rotatorio (*round-robin*), como se muestra en la figura 5-20(a) para un RAID con cuatro unidades de disco.

Al proceso de distribuir datos sobre varias unidades de esta forma se le conoce como **reparto de bloques** (*striping*). Para cada ejemplo, si el software emite un comando para leer un bloque de datos de cuatro bloques consecutivos que empieza en un límite de bloque, el controlador RAID descompondrá este comando en cuatro comandos separados, uno para cada uno de los cuatro discos, y hará que operen en paralelo. Así, tenemos E/S en paralelo sin que el software lo sepa.

El nivel 0 de RAID funciona mejor con las peticiones grandes; entre mayores sean, mejor. Si una petición es mayor que el número de unidades multiplicado por el tamaño de los bloques, algunas unidades obtendrán varias peticiones, por lo que cuando terminen la primera petición empezarán la segunda. Es responsabilidad del controlador dividir la petición y alimentar los comandos apropiados a los discos apropiados en la secuencia correcta, y después ensamblar correctamente los resultados en la memoria. El rendimiento es excelente y la implementación sencilla.

El nivel 0 de AID funciona peor con los sistemas operativos que habitualmente piden datos un sector a la vez. Los resultados serán correctos pero no hay paralelismo, y por ende no aumenta el rendimiento. Otra desventaja de esta organización es que la confiabilidad es potencialmente peor que tener un SLED. Si RAID consiste de cuatro discos, cada uno con un tiempo de falla promedio de 20,000 horas, una vez cada 5,000 fallará una unidad y los datos se perderán por completo. Un SLED con un tiempo promedio de falla de 20,000 horas será cuatro veces más confiable. Como no hay redundancia presente en este diseño, en realidad no es un verdadero RAID.

La siguiente opción, el nivel 1 de RAID que se muestra en la figura 5-20(b), es un RAID verdadero. Duplica todos los discos, por lo que hay cuatro discos primarios y cuatro discos de respal-

do. En una operación de escritura cada bloque se escribe dos veces. En una lectura se puede utilizar cualquiera de las copias, con lo que se distribuye la carga entre más unidades. En consecuencia, el rendimiento de escritura no es mejor que para una sola unidad, pero el rendimiento de lectura puede ser de hasta el doble. La tolerancia a fallas es excelente: si una unidad falla, simplemente se utiliza la copia. La recuperación consiste tan sólo en instalar una nueva unidad y copiar toda la unidad de respaldo en ella.

A diferencia de los niveles 0 y 1, que funcionan con bandas de sectores, el nivel 2 de RAID funciona por palabra, tal vez hasta por byte. Imagine dividir cada byte del disco virtual en un par de medios bits (*nibbles* de 4 bits), y después agregar un código de Hamming a cada uno para formar una palabra de 7 bits, de la cual los bits 1, 2 y 4 son bits de paridad. Imagine que las siete unidades de la figura 5-20(c) se sincronizan en términos de la posición del brazo y de la posición rotacional. Entonces sería posible escribir la palabra con código Hamming de 7 bits sobre las siete unidades, un byte por unidad.

La computadora Thinking Machines CM-2 utilizaba este esquema. Tomaba palabras de datos de 32 bits y agregaba 6 bits de paridad para formar una palabra de Hamming de 38 bits, más un bit adicional para la paridad de palabra, y esparcía cada palabra sobre 39 unidades de disco. La tasa de transferencia total era inmensa, ya que en un tiempo de sector podía escribir 32 sectores de datos. Además, al perder un disco no había problemas, ya que significaba perder 1 bit en cada lectura de palabra de 39 bits, algo que el código de Hamming solucionaría al instante.

Por el lado negativo, este esquema requiere que todas las unidades se sincronicen en forma rotacional, y sólo tiene sentido con un número considerable de unidades (incluso con 32 unidades de datos y 6 unidades de paridad, la sobrecarga es de 19%). También exige mucho al controlador, ya que debe realizar una suma de comprobación de Hamming por cada tiempo de bit.

El nivel 3 de RAID es una versión simplificada del nivel 2 de RAID. Se ilustra en la figura 5-20(d). Aquí se calcula un solo bit de paridad para cada palabra de datos y se escribe en una unidad de paridad. Al igual que en el nivel 2 de RAID, las unidades deben tener una sincronización exacta, ya que las palabras de datos individuales están distribuidas a través de varias unidades.

En primera instancia, podría parecer que un solo bit de paridad proporciona sólo detección, y no corrección de errores. Para el caso de los errores aleatorios no detectados, esta observación es verdadera. Sin embargo, para el caso de la falla de una unidad, proporciona corrección de error de 1 bit debido a que la posición del bit defectuoso se conoce. Si una unidad falla, el controlador sólo simula que todos sus bits son 0; si una palabra tiene un error de paridad, el bit de la unidad defectuosa debe haber sido 1, por lo que se corrige. Aunque los niveles 2 y 3 de RAID ofrecen velocidades de transferencia de datos muy altas, el número de peticiones de E/S separadas que pueden manejar por segundo no es mayor que para una sola unidad.

Los niveles 4 y 5 de RAID funcionan con bloques otra vez, no con palabras individuales con paridad, y no requieren unidades sincronizadas. El nivel 4 de RAID [véase la figura 5-20(e)] es como el nivel 0, en donde se escribe una paridad banda por banda en una unidad adicional. Por ejemplo, si cada banda es de k bytes, se aplica un OR EXCLUSIVO a todos los bloques y se obtiene como resultado un bloque de paridad de k bytes. Si una unidad falla, los bytes perdidos se pueden recalcular a partir del bit de paridad, leyendo el conjunto completo de unidades.

Este diseño protege contra la pérdida de una unidad, pero tiene un desempeño pobre para las actualizaciones pequeñas. Si se cambia un sector, es necesario leer todas las unidades para poder re-

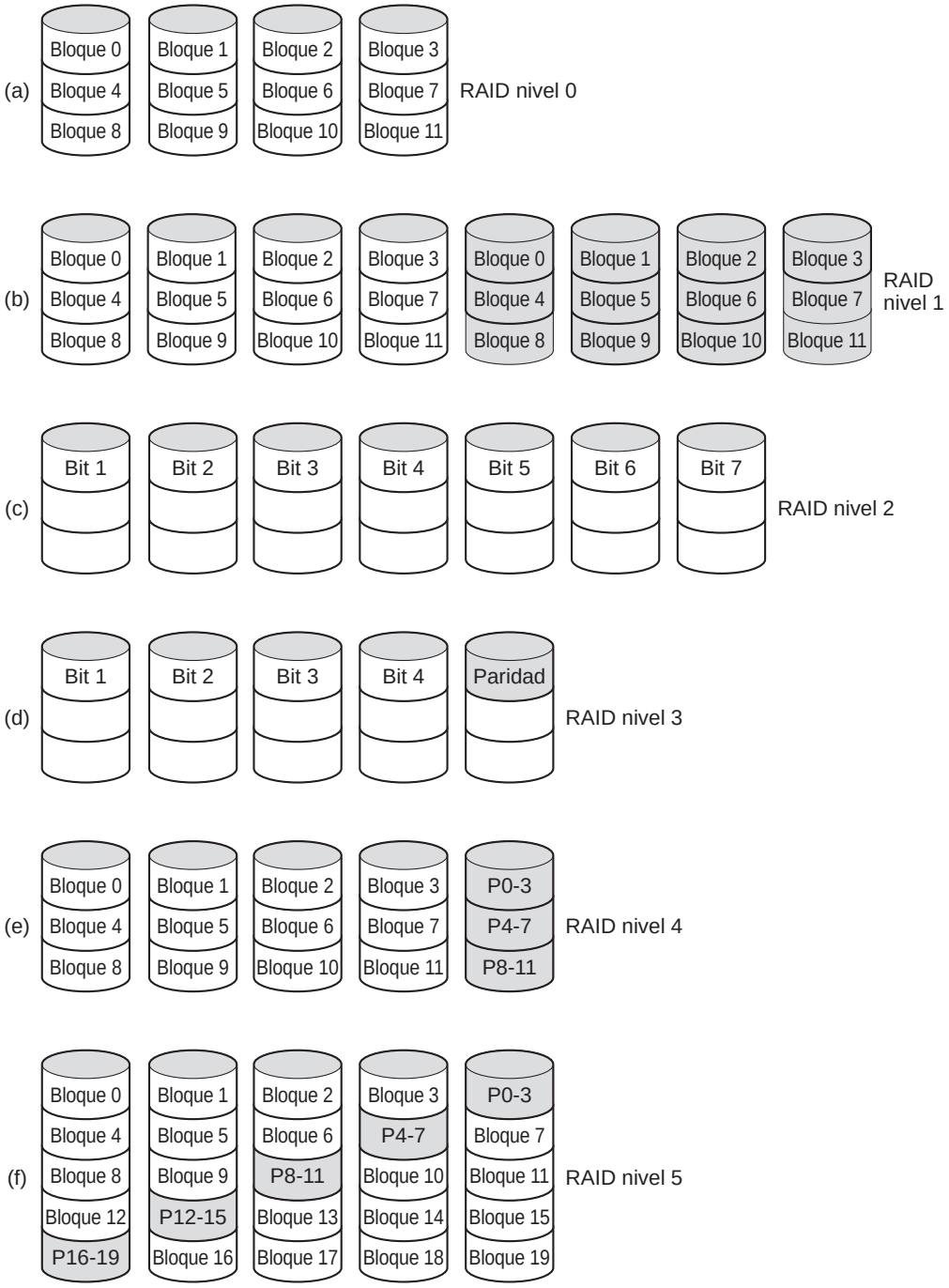

Figura 5-20. Niveles 0 a 5 de RAID. Las unidades de respaldo y de paridad se muestran sombreadas.

calcular la paridad, que entonces se debe volver a escribir. De manera alternativa, puede leer los datos antiguos del usuario y los datos antiguos de paridad, y recalcular la nueva paridad a partir de ellos. Incluso con esta optimización, una pequeña actualización requiere dos lecturas y dos escrituras

Como consecuencia de la pesada carga en la unidad de paridad, puede convertirse en un cuello de botella. Éste se elimina en el nivel 5 de RAID al distribuir los bits de paridad de manera uniforme sobre todas las unidades, por turno rotatorio (*round-robin*), como se muestra en la figura 5-20(f). Sin embargo, en caso de una falla de unidad, la reconstrucción del contenido de la unidad fallida es un proceso complejo.

CD-ROMs

En años recientes se han empezado a utilizar los discos ópticos (en contraste a los magnéticos). Estos discos tienen densidades de grabación mucho más altas que los discos magnéticos convencionales. Los discos ópticos se desarrollaron en un principio para grabar programas de televisión, pero se les puede dar un uso más estético como dispositivos de almacenamiento de computadora. Debido a su capacidad potencialmente enorme, los discos ópticos han sido tema de una gran cantidad de investigación y han pasado por una evolución increíblemente rápida.

Los discos ópticos de primera generación fueron inventados por el conglomerado de electrónica holandés Philips, para contener películas. Tenían 30 centímetros de diámetro y se comercializaron bajo el nombre LaserVision, pero no tuvieron mucha popularidad, excepto en Japón.

En 1980, Philips y Sony desarrollaron el CD (Disco Compacto), que sustituyó rápidamente al disco de vinilo de 33 1/3 RPM que se utilizaba para música (excepto entre los conocedores, que aún preferían el vinilo). Los detalles técnicos precisos para el CD se publicaron en un Estándar Internacional oficial (IS 10149) conocido popularmente como el **Libro rojo** debido al color de su portada (los Estándares Internacionales son emitidos por la Organización Internacional de Estándares, que es la contraparte internacional de los grupos de estándares nacionales como ANSI, DIN, etc. Cada uno tiene un número IS). El punto de publicar las especificaciones de los discos y las unidades como un estándar internacional tiene como fin permitir que los CDs de distintas compañías disqueras y los reproductores de distintos fabricantes electrónicos puedan funcionar en conjunto. Todos los CDs tienen 120 mm de diámetro y 1.2 mm de grosor, con un hoyo de 15 mm en medio. El CD de audio fue el primer medio de almacenamiento digital masivo en el mercado. Se supone que deben durar 100 años. Por favor consulte de nuevo en el 2080 para saber cómo le fue al primer lote.

Un CD se prepara en varios pasos. El primero consiste en utilizar un láser infrarrojo de alto poder para quemar hoyos de 0.8 micrones de diámetro en un disco maestro con cubierta de vidrio. A partir de este disco maestro se fabrica un molde, con protuberancias en lugar de los hoyos del láser. En este molde se inyecta resina de policarbonato fundido para formar un CD con el mismo patrón de hoyos que el disco maestro de vidrio. Después se deposita una capa muy delgada de aluminio reflectivo en el policarbonato, cubierta por una laca protectora y finalmente una etiqueta. Las depresiones en el sustrato de policarbonato se llaman **hoyos** (*pits*); las áreas no quemadas entre los hoyos se llaman **áreas lisas** (*lands*).

Cuando se reproduce, un diodo láser de baja energía emite luz infrarroja con una longitud de onda de 0.78 micrones sobre los hoyos y áreas lisas a medida que van pasando. El láser está del lado del policarbonato, por lo que los hoyos salen hacia el láser como protuberancias en la superficie de área lisa. Como los hoyos tienen una altura de un cuarto de la longitud de onda de la luz del láser, la luz que se refleja de un hoyo está desfasada por media longitud de onda con la luz que se refleja de la superficie circundante. Como resultado, las dos partes interfieren en forma destructiva y devuelven menos luz al fotodetector del reproductor que la luz que rebota de un área lisa. Así es como el reproductor puede diferenciar un hoyo de un área lisa. Aunque podría parecer más simple utilizar un hoyo para grabar un 0 y un área lisa para grabar un 1, es más confiable utilizar una transición de hoyo a área lisa o de área lisa a un hoyo para un 1 y su ausencia como un 0, por lo que se utiliza este esquema.

Los hoyos y las áreas lisas se escriben en una sola espiral continua, que empieza cerca del hoyo y recorre una distancia de 32 mm hacia el borde. La espiral realiza 22,188 revoluciones alrededor del disco (aproximadamente 600 por milímetro). Si se desenredara, tendría 5.6 km de largo. La espiral se ilustra en la figura 5-21.

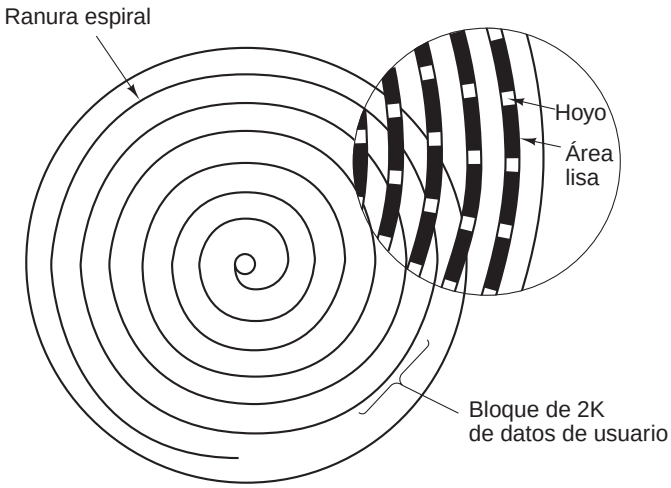

Figura 5-21. Estructura de grabación de un disco compacto o CD-ROM.

Para reproducir la música a una velocidad uniforme, es necesario un flujo continuo de hoyos y áreas a una velocidad lineal constante. En consecuencia, la velocidad de rotación del CD se debe reducir en forma continua a medida que la cabeza de lectura se desplaza desde la parte interna del CD hacia la parte externa. En el interior la velocidad de rotación es de 530 RPM para lograr la velocidad de flujo continuo deseada de 120 cm/seg; en el exterior tiene que reducirse a 200 RMP para proporcionar la misma velocidad lineal en la cabeza. Una unidad de velocidad lineal constante es muy distinta a una unidad de disco magnético, que opera a una velocidad angular constante, sin importar dónde se encuentre la cabeza en un momento dado. Además, 530 RPM están muy lejos de las 3600 a 7200 RPM a las que giran la mayoría de los discos magnéticos.

En 1984, Philips y Sony se dieron cuenta del potencial de utilizar CDs para almacenar datos de computadora, por lo cual publicaron el **Libro amarillo** que define un estándar preciso para lo que

se conoce ahora como **CD-ROM**s (*Compact disk-read only memory,* Disco compacto–memoria de sólo lectura). Para apoyarse en el mercado de CDs de audio, que para ese entonces ya era considerable, los CD-ROMs tenían que ser del mismo tamaño físico que los CDs de audio, ser compatibles en sentido mecánico y óptico con ellos, y se debían producir utilizando las mismas máquinas de moldeado por inyección de policarbonato. Las consecuencias de esta decisión fueron que no sólo se requerían motores lentos de velocidad variable, sino también que el costo de fabricación de un CD-ROM estaría muy por debajo de un dólar en un volumen moderado.

Lo que definió el Libro amarillo fue el formato de los datos de computadora. También mejoró las habilidades de corrección de errores del sistema, un paso esencial pues aunque a los amantes de la música no les importaba perder un poco aquí y allá, los amantes de las computadoras tendían a ser Muy Exigentes en cuanto a eso. El formato básico de un CD-ROM consiste en codificar cada byte en un símbolo de 14 bits, que es suficiente como para usar el código de Hamming en un byte de 8 bits con 2 bits de sobra. De hecho, se utiliza un sistema de codificación más potente. La asignación de 14 a 8 para la lectura se realiza en el hardware mediante la búsqueda en una tabla.

En el siguiente nivel hacia arriba, un grupo de 42 símbolos consecutivos forma una **estructura** de 588 bits. Cada estructura contiene 192 bits de datos (24 bytes). Los 396 bits restantes se utilizan para corrección y control de errores. De éstos, 252 son los bits de corrección de errores en los símbolos de 14 bits, y 144 se llevan en las transmisiones de símbolos de 8 bits. Hasta ahora, este esquema es idéntico para los CDs de audio y los CD-ROMs.

Lo que agrega el Libro amarillo es el agrupamiento de 98 estructuras en un **sector de CD-ROM**, como se muestra en la figura 5-22. Cada sector del CD-ROM empieza con un preámbulo de 16 bytes, del cual los primeros 12 son 00FFFFFFFFFFFFFFFFFFFF00 (hexadecimal) para permitir que el reproductor reconozca el inicio de un sector de CD-ROM. Los siguientes 3 bytes contienen el número de sector, que se requiere debido a que es mucho más difícil realizar búsquedas en un CD-ROM con una sola espiral de datos que en un disco magnético, con sus pistas concéntricas uniformes. Para buscar, el software en la unidad calcula aproximadamente a dónde debe ir, desplaza ahí la cabeza y después empieza a buscar un preámbulo para ver qué tan buena fue su aproximación. El último byte del preámbulo es el modo.

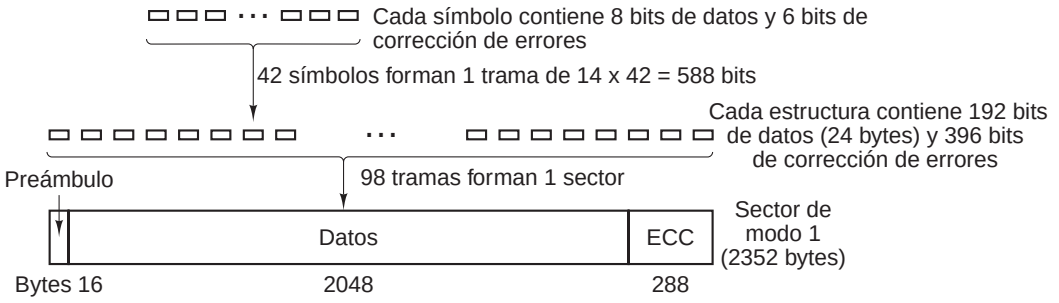

Figura 5-22. Distribución lógica de los datos en un CD-ROM.

El Libro amarillo define dos modos. El modo 1 utiliza la distribución de la figura 5-22. Con un preámbulo de 16 bytes, 2048 bytes de datos y un código de corrección de errores de 288 bytes

(un código Reed-Solomon cruzado entrelazado). El modo 2 combina los campos de datos y ECC en un campo de datos de 2336 bytes para aquellas aplicaciones que no necesitan (o que no pueden darse el tiempo de llevar a cabo) la corrección de errores, como el audio y el video. Hay que tener en cuenta que para ofrecer una excelente confiabilidad, se utilizan tres esquemas separados de corrección de errores: dentro de un símbolo, dentro de una estructura y dentro de un sector del CD-ROM. Los errores de un solo bit se corrigen en el nivel más bajo, los errores de ráfagas cortas se corrigen en el nivel de estructura y los errores residuales se atrapan en el nivel de sector. El precio a pagar por esta confiabilidad es que se requieren 98 estructuras de 588 bits (7203 bytes) para llevar a cabo una sola transferencia de 2048 bytes, una eficiencia de sólo 28%.

Las unidades de CD-ROM de una sola velocidad operan a 75 sectores/seg, con lo cual se obtiene una velocidad de transferencia de datos de 153,600 bytes/seg en modo 1 y de 175,200 bytes/seg en modo 2. Las unidades de doble velocidad son el doble de rápidas, y así en lo sucesivo hasta la velocidad más alta. Por ende, una unidad de 40x puede transferir datos a una velocidad de $40 \times 153{,}600$ bytes/seg, suponiendo que la interfaz de la unidad, el bus y el sistema operativo puedan manejar esta velocidad de transferencia de datos. Un CD de audio estándar tiene espacio para 74 minutos de música que, si se utilizan para datos en modo 1, logran una capacidad de 681,984,000 bytes. Esta cifra se reporta comúnmente como 650 MB, debido a que 1 MB equivale a 2^{20} bytes (1,048,576 bytes) y no a 1,000,000 bytes.

Tenga en cuenta que ni siquiera una unidad de CD-ROM de 32x (4,915,200 bytes/s) le hace frente a una unidad de disco magnético SCSI rápida a 10 MB/seg, aunque muchas unidades de CD-ROM utilizan la interfaz SCSI (también existen las unidades IDE de CD-ROM). Cuando se da uno cuenta de que el tiempo de búsqueda generalmente es de varios cientos de milisegundos, queda claro que las unidades de CD-ROM no están en la misma categoría de rendimiento que las unidades de disco magnético, a pesar de su gran capacidad.

En 1986, Philips publicó el **Libro verde**, agregando gráficos y la habilidad de entrelazar audio, video y datos en el mismo sector, una característica esencial para los CD-ROMs de multimedia.

La última pieza del rompecabezas del CD-ROM es el sistema de archivos. Para que fuera posible usar el mismo CD-ROM en distintas computadoras, era necesario un acuerdo sobre los sistemas de archivos de CD-ROM. Para llegar a este acuerdo, se reunieron los representantes de muchas compañías de computadoras en Lake Tahoe, en la región High Sierra del límite entre California y Nevada, e idearon un sistema de archivos al que llamaron **High Sierra**. Después evolucionó en un Estándar Internacional (IS 9660). Tiene tres niveles. El nivel 1 utiliza nombres de archivo de hasta 8 caracteres, seguidos opcionalmente de una extensión de hasta 3 caracteres (la convención de denominación de archivos de MS-DOS). Los nombres de archivo pueden contener sólo letras mayúsculas, dígitos y el guión bajo. Los directorios se pueden anidar hasta ocho niveles, pero los nombres de los directorios no pueden contener extensiones. El nivel 1 requiere que todos los archivos sean contiguos, lo cual no es problema en un medio en el que se escribe sólo una vez. Cualquier CD-ROM que cumpla con el nivel 1 del IS 9660 se puede leer utilizando MS-DOS, una computadora Apple, una computadora UNIX o casi cualquier otra computadora. Los editores de CD-ROMs consideran que esta propiedad es una gran ventaja.

El nivel 2 del IS 9660 permite nombres de hasta 32 caracteres, y el nivel 3 permite archivos no contiguos. Las extensiones Rock Ridge (denominadas en honor a la ciudad en la que se desarrolla el filme de Gene Wilder llamado *Blazing Saddles*) permiten nombres muy largos (para UNIX),

UIDs, GIDs y vínculos simbólicos, pero los CD-ROMs que no cumplan con el nivel 1 no podrán leerse en todas las computadoras.

Los CD-ROMs se han vuelto en extremo populares para publicar juegos, películas, enciclopedias, atlas y trabajos de referencia de todo tipo. La mayoría del software comercial viene ahora en CD-ROM. Su combinación de gran capacidad y bajo costo de fabricación los hace adecuados para innumerables aplicaciones.

CD-Grabables

En un principio, el equipo necesario para producir un CD-ROM maestro (o CD de audio, para esa cuestión) era extremadamente costoso. Pero como siempre en la industria de las computadoras, nada permanece costoso por mucho tiempo. A mediados de la década de 1990, los grabadores de CDs no más grandes que un reproductor de CD eran un periférico común disponible en la mayoría de las tiendas de computadoras. Estos dispositivos seguían siendo distintos de los discos magnéticos, porque una vez que se escribía información en ellos no podía borrarse. Sin embargo, rápidamente encontraron un nicho como medio de respaldo para discos duros grandes y también permitieron que individuos o empresas que iniciaban operaciones fabricaran sus propios CD-ROMs de distribución limitada, o crear CDs maestros para entregarlos a plantas de duplicación de CDs comerciales de alto volumen. Estas unidades se conocen como **CD-R**s (**CD-Grabables**).

Físicamente, los CD-Rs empiezan con discos en blanco de policarbonato de 120 mm, que son similares a los CD-ROMs, excepto porque contienen una ranura de 0.6 mm de ancho para guiar el láser para la escritura. La ranura tiene una excursión sinoidal de 0.3 mm a una frecuencia exacta de 22.05 kHz para proveer una retroalimentación continua, de manera que la velocidad de rotación se pueda supervisar y ajustar con precisión, en caso de que sea necesario. Los CD-Rs tienen una apariencia similar a los CD-ROMs, excepto porque tienen una parte superior dorada, en lugar de una plateada. El color dorado proviene del uso de verdadero oro en vez de aluminio para la capa reflectora. A diferencia de los CDs plateados que contienen depresiones físicas, en los CD-Rs la distinta reflectividad de hoyos y áreas lisas se tiene que simular. Para ello hay que agregar una capa de colorante entre el policarbonato y la capa de oro reflectiva, como se muestra en la figura 5-23. Se utilizan dos tipos de colorante: la cianina, que es verde, y la ptalocianina, que es de color naranja con amarillo. Los químicos pueden argumentar indefinidamente sobre cuál es mejor. Estos colorantes son similares a los que se utilizan en la fotografía, lo cual explica por qué Eastman Kodak y Fuji son los principales fabricantes de CD-Rs en blanco.

En su estado inicial, la capa de colorante es transparente y permite que la luz del láser pase a través de ella y se refleje en la capa dorada. Para escribir, el láser del CD-R se pone en alto poder (8 a 16 mW). Cuando el haz golpea en un punto del colorante, se calienta y quebranta un lazo químico. Este cambio en la estructura molecular crea un punto oscuro. Cuando se lee de vuelta (a 0.5 mW), el fotodetector ve una diferencia entre los puntos oscuros en donde se ha golpeado el colorante y las áreas transparentes donde está intacto. Esta diferencia se interpreta como la diferencia entre los hoyos y las áreas lisas, aun y cuando se lea nuevamente en un lector de CD-ROM regular, o incluso en un reproductor de CD de audio.

Ningún nuevo tipo de CD podría andar con la frente en alto sin un libro de colores, por lo que el CD-R tiene el **Libro naranja**, publicado en 1989. Este documento define el CD-R y también un

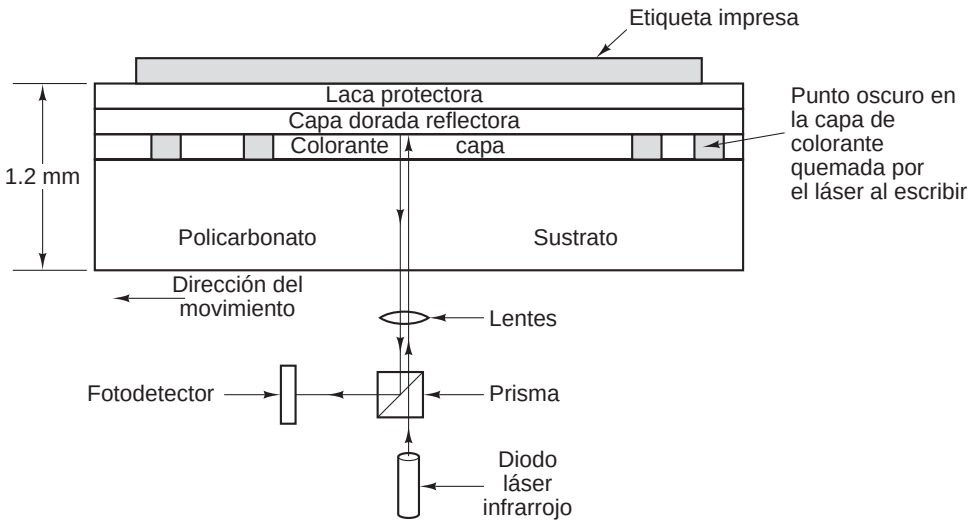

Figura 5-23. Sección transversal de un disco CD-R y el láser (no está a escala). Un CD-ROM plateado tiene una estructura similar, excepto sin la capa de colorante y con una capa de aluminio picada en vez de una capa dorada.

nuevo formato, el **CD-ROM XA**, que permite escribir CD-Rs en forma incremental; unos cuantos sectores hoy, otros pocos mañana, y unos cuantos el siguiente mes. A un grupo de sectores consecutivos que se escriben a la vez se le llama **pista de CD-ROM**.

Uno de los primeros usos del CD-R fue para el PhotoCD de Kodak. En este sistema, el cliente trae un rollo de película expuesta y su viejo PhotoCD al procesador de fotografías, y obtiene de vuelta el mismo PhotoCD, al que se le agregan las nuevas fotografías después de las anteriores. El nuevo lote, que se crea al explorar los negativos, se escribe en el PhotoCD como una pista separada en el CD-ROM. Se requería la escritura incremental debido a que cuando se introdujo este producto, los CD-R en blanco eran demasiado costosos como para ofrecer uno nuevo por cada rollo de película.

Sin embargo, la escritura incremental crea otro problema. Antes del Libro naranja, todos los CD-ROMs tenían una sola **VTOC** (*Volume Table of Contents*, Tabla de contenido del volumen) al principio. Ese esquema no funciona con las escrituras incrementales (es decir de varias pistas). La solución del Libro naranja es proporcionar a cada pista de CD-ROM su propia VTOC. Los archivos que se listan en la VTOC pueden incluir algunos o todos los archivos de las pistas anteriores. Una vez que se inserta el CD-R en la unidad, el sistema operativo busca a través de todas las pistas de CD-ROM para localizar la VTOC más reciente, que proporciona el estado actual del disco. Al incluir algunos, pero no todos los archivos de las pistas anteriores en la VTOC actual, es posible dar la ilusión de que se han eliminado archivos. Las pistas se pueden agrupar en **sesiones**, lo cual conlleva a los CD-ROMs de **multisesión**. Los reproductores de CDs de audio estándar no pueden manejar los CDs de multisesión, ya que esperan una sola VTOC al principio. Sin embargo, algunas aplicaciones de computadora pueden manejarlos.

El CD-R hace posible que los individuos y las empresas copien fácilmente CD-ROMs (y CDs de audio), por lo general violando los derechos del editor. Se han ideado varios esquemas para dificultar la piratería y para que sea difícil leer un CD-ROM utilizando otra cosa que no sea el software del editor. Uno de ellos implica grabar todas las longitudes de los archivos en el CD-ROM como de varios gigabytes, para frustrar cualquier intento de copiar los archivos en disco duro mediante el uso de software de copiado estándar. Las verdaderas longitudes se incrustan en el software del editor o se ocultan (posiblemente cifradas) en el CD-ROM, en un lugar inesperado. Otro esquema utiliza ECCs incorrectos de manera intencional en sectores seleccionados, esperando que el software de copiado de CDs "corrija" esos errores. El software de aplicación comprueba los ECCs por sí mismo, rehusándose a funcionar si están correctos. El uso de huecos no estándar entre las pistas y otros "defectos" físicos también es posible.

CD-Regrabables

Aunque las personas están acostumbradas a otros medios de escritura de sólo una vez como el papel y la película fotográfica, hay una demanda por el CD-ROM regrabable. Una tecnología que ahora está disponible es la del **CD-RW** (**CD-Regrabable**), que utiliza medios del mismo tamaño que el CD-R. Sin embargo, en vez de colorante de cianina o ptalocianina, el CD-RW utiliza una aleación de plata, indio, antimonio y telurio para la capa de grabación. Esta aleación tiene dos estados estables: cristalino y amorfo, con distintas reflectividades.

Las unidades de CD-RW utilizan láseres con tres potencias: en la posición de alta energía, el láser funde la aleación y la convierte del estado cristalino de alta reflectividad al estado amorfo de baja reflectividad para representar un hoyo; en la posición de energía media, la aleación se funde y se vuelve a formar en su estado cristalino natural para convertirse en un área lisa nuevamente; en baja energía se detecta el estado del material (para la lectura), pero no ocurre una transición de estado.

La razón por la que el CD-RW no ha sustituido al CD-R es que los CD-RW en blanco son más costosos. Además, para las aplicaciones que consisten en respaldar discos duros, el hecho de que una vez escrito el CD-R no se pueda borrar accidentalmente es una gran ventaja.

DVD

El formato básico de CD/CD-ROM ha estado en uso desde 1980. La tecnología ha mejorado desde entonces, por lo que ahora los discos ópticos de mayor capacidad son económicamente viables y hay una gran demanda por ellos. Hollywood estaría encantado de eliminar las cintas de video análogas a favor de los discos digitales, ya que los discos tienen una mayor calidad, son más económicos de fabricar, duran más tiempo, ocupan menos espacio en las repisas de las tiendas de video y no tienen que rebobinarse. Las empresas de electrónica para el consumidor siempre están buscando un nuevo producto que tenga un gran éxito, y muchas empresas de computadoras desean agregar características de multimedia a su software.

Esta combinación de tecnología y demanda por tres industrias inmensamente ricas y poderosas conllevó al **DVD**, que originalmente era un acrónimo para **Video disco digital (***Digital Video Disk***)**,

pero que ahora se conoce oficialmente como **Disco versátil digital (***Digital Versatile Disk***)**. Los DVDs utilizan el mismo diseño general que los CDs, con discos de policarbonato moldeado por inyección de 120 mm que contienen hoyos y áreas lisas, que se iluminan mediante un diodo láser y se leen mediante un fotodetector. Lo nuevo es el uso de

1. Hoyos más pequeños (0.4 micrones, en comparación con 0.8 micrones para los CDs).
2. Una espiral más estrecha (0.74 micrones entre pistas, en comparación con 1.6 micrones para los CDs).
3. Un láser rojo (a 0.65 micrones, en comparación con 0.78 micrones para los CDs).

En conjunto, estas mejoras elevan la capacidad siete veces, hasta 4.7 GB. Una unidad de DVD 1x opera a 1.4 MB/seg (en comparación con los 150 KB/seg de los CDs). Por desgracia, el cambio a los láseres rojos utilizados en los supermercados implica que los reproductores de DVD requieren un segundo láser o una óptica compleja de conversión para poder leer los CDs y CD-ROMs existentes. Pero con la disminución en el precio de los láseres, la mayoría de los reproductores de DVD tienen ahora ambos tipos de láser para leer ambos tipos de medios.

¿Es 4.7 GB suficiente? Tal vez. Mediante el uso de la compresión MPEG-2 (estandarizada en el IS 13346), un disco DVD de 4.7 GB puede contener 133 minutos de video de pantalla y movimiento completo en alta resolución (720 x 480), así como pistas de sonido en hasta ocho lenguajes y subtítulos en 32 más. Cerca de 92% de todas las películas que se hayan realizado en Hollywood son de menos de 133 minutos. Sin embargo, algunas aplicaciones como los juegos multimedia o las obras de consulta pueden requerir más, y a Hollywood le gustaría colocar varias películas en el mismo disco, por lo que se han definido cuatro formatos:

1. Un solo lado, una sola capa (4.7 GB).
2. Un solo lado, doble capa (8.5 GB).
3. Doble lado, una sola capa (9.4 GB).
4. Doble lado, doble capa (17 GB).

¿Por qué tantos formatos? En una palabra: política. Philips y Sony querían discos de un solo lado, doble capa para la versión de alta capacidad, pero Toshiba y Time Warner querían discos de doble lado, una sola capa. Philips y Sony no creyeron que la gente estuviera dispuesta a voltear los discos, y Time Warner no creía que colocar dos capas en un lado podía funcionar. El resultado: todas las combinaciones; el mercado será quien defina cuáles sobrevivirán.

La tecnología de doble capa tiene una capa reflectora en la parte inferior, con una capa semireflectora encima. Dependiendo del lugar en el que se enfoque el láser, rebota de una capa o de la otra. La capa inferior necesita hoyos y áreas lisas ligeramente más grandes para poder leer de manera confiable, por lo que su capacidad es un poco menor que la de la capa superior.

Los discos de doble lado se fabrican tomando dos discos de un solo lado de 0.6 mm y pegándolos de su parte posterior. Para que el grosor de todas las versiones sea el mismo, un disco de un solo lado consiste en un disco de 0.6 mm pegado a un sustrato en blanco (o tal vez en el futuro, uno

que consista en 133 minutos de publicidad, con la esperanza de que la gente tenga curiosidad sobre lo que pueda contener). La estructura del disco de doble lado, doble capa se ilustra en la figura 5-24.

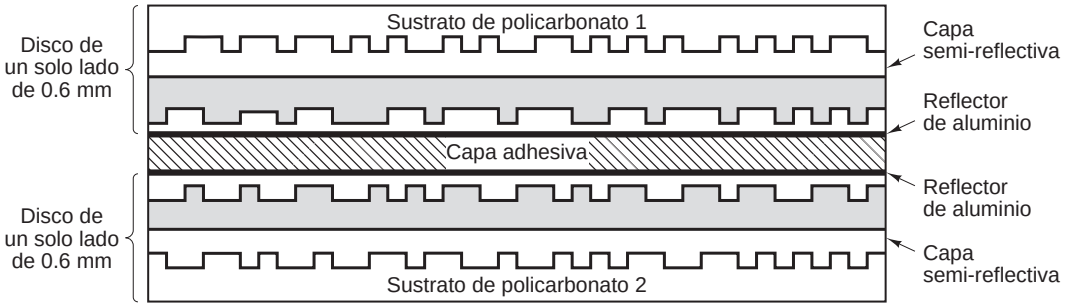

Figura 5-24. Un disco DBD de doble lado y doble capa.

El DVD fue ideado por un consorcio de 10 empresas de aparatos electrónicos para el hogar, siete de ellas japonesas, en estrecha cooperación con los principales estudios de Hollywood (algunos de los cuales son propiedad de las empresas de electrónica japonesas que están en el consorcio). Las industrias de las computadoras y las telecomunicaciones no fueron invitadas al picnic, y el enfoque resultante estuvo en utilizar el DVD para exposiciones de renta y venta de películas. Por ejemplo, las características estándar incluyen la omisión en tiempo real de escenas sucias (para permitir que los padres conviertan una película con clasificación NC17 en una segura para los bebés), sonido de seis canales y soporte para Pan-and-Scan. Esta última característica permite al reproductor del DVD decidir en forma dinámica cómo cortar los bordes izquierdo y derecho de las película (cuya proporción de anchura:altura es 3:2) para adaptarlas a los televisores actuales (cuya proporción de aspecto es 4:3).

Otra cuestión que la industria de las computadoras probablemente no hubiera considerado es una incompatibilidad intencional entre los discos destinados para los Estados Unidos y los discos destinados para Europa, y otros estándares más para los otros continentes. Hollywood exigía esta "característica" debido a que las nuevas películas siempre se estrenan primero en los Estados Unidos y después se envían a Europa, cuando los videos salen en los Estados Unidos. La idea era hacer que las tiendas de video europeas no pudieran comprar videos en los EE.UU. con demasiada anticipación haciendo disminuir las ventas de boletos para las nuevas películas en los cines europeos. Si Hollywood hubiera operado la industria de las computadoras, tendríamos discos flexibles de 3.5 pulgadas en los Estados Unidos y discos flexibles de 9 cm en Europa.

Las personas que idearon los DVDs de un solo/doble lado y una sola/doble capa están ideando nuevos descubrimientos otra vez. La siguiente generación también carece de un solo estándar debido a las disputas políticas por parte de los participantes en la industria. Uno de los nuevos dispositivos es **Blu-ray**, que utiliza un láser de 0.405 micrones (azul) para empaquetar 25 GB en un disco de una sola capa y 50 GB en un disco de doble capa. El otro es **HD DVD**, que utiliza el mismo láser azul pero tiene una capacidad de sólo 15 GB (una sola capa) y 30 GB (doble capa). Esta guerra de los formatos ha dividido a los estudios de películas, los fabricantes de computadoras y

las compañías de software. Como resultado de la falta de estandarización, esta generación está despegando con bastante lentitud, a medida que los consumidores esperan a que se asiente el polvo para ver qué formato ganará. Esta falta de sensibilidad por parte de la industria nos trae a la mente el famoso comentario de George Santayana: “Aquellos que no pueden aprender de la historia están destinados a repetirla”.

5.4.2 Formato de disco

Un disco duro consiste en una pila de platos de aluminio, aleación de acero o vidrio, de 5.25 o 3.5 pulgadas de diámetro (o incluso más pequeños en las computadoras notebook). En cada plato se deposita un óxido de metal delgado magnetizable. Después de su fabricación, no hay información de ninguna clase en el disco.

Antes de poder utilizar el disco, cada plato debe recibir un **formato de bajo nivel** mediante software. El formato consiste en una serie de pistas concéntricas, cada una de las cuales contiene cierto número de sectores, con huecos cortos entre los sectores. El formato de un sector se muestra en la figura 5-25.

Figura 5-25. Un sector de disco.

El preámbulo empieza con cierto patrón de bits que permite al hardware reconocer el inicio del sector. También contiene los números de cilindro y sector, junto con cierta información adicional. El tamaño de la porción de datos se determina con base en el programa de formato de bajo nivel. La mayoría de los discos utilizan sectores de 512 bytes. El campo ECC contiene información redundante que se puede utilizar para recuperarse de los errores de lectura. El tamaño y contenido de este campo varía de un fabricante a otro, dependiendo de cuánto espacio de disco esté dispuesto a perder el diseñador por obtener una mayor confiabilidad, y de qué tan complejo sea el código ECC que pueda manejar el controlador. Un campo ECC de 16 bytes no es poco común. Además, todos los discos duros tienen cierta cantidad de sectores adicionales que se asignan para utilizarlos como reemplazo de los sectores con un defecto de fabricación.

La posición del sector 0 en cada pista está desfasada de la pista anterior cuando se aplica el formato de bajo nivel. Este desplazamiento, conocido como **desajuste de cilindros**, se realiza para mejorar el rendimiento. La idea es permitir que el disco lea varias pistas en una operación continua sin perder datos. La naturaleza de este problema se puede ver en la figura 5-19(a). Suponga que una petición necesita 18 sectores, empezando en el sector 0 de la pista más interna. Para leer los primeros 16 sectores se requiere una rotación de disco, pero se necesita una búsqueda para desplazarse una pista hacia fuera para llegar al sector 17. Para cuando la cabeza se ha desplazado una pista, el sector 0 ha girado más allá de la cabeza, de manera que se necesita una rotación para que vuelva a pasar por debajo de la cabeza. Este problema se elimina al desviar los sectores como se muestra en la figura 5-26.

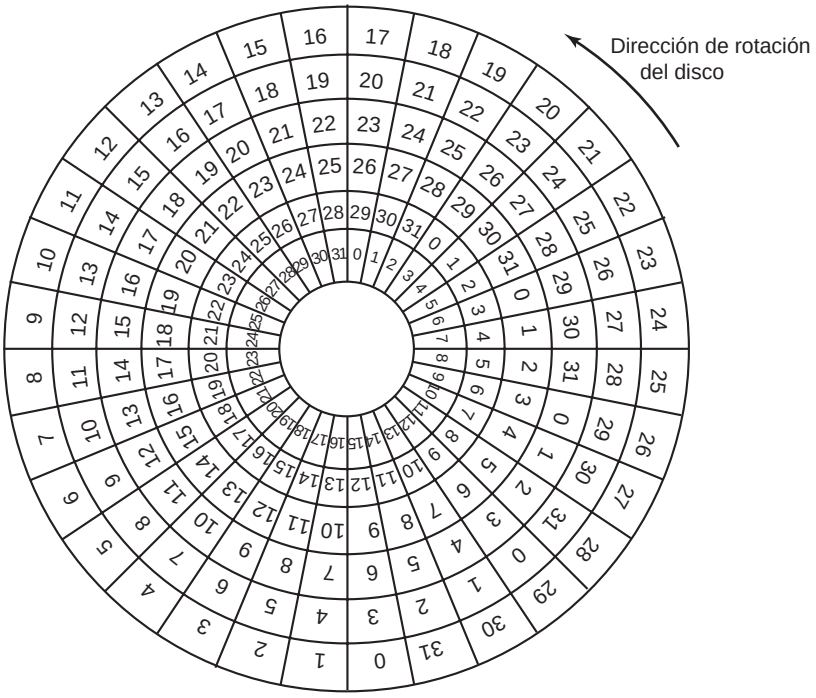

Figura 5-26. Una ilustración de la desviación de cilindros.

La cantidad de desviación de los cilindros depende de la geometría del disco. Por ejemplo, una unidad de 10,000 RPM gira en 6 mseg. Si una pista contiene 300 sectores, un nuevo sector pasa debajo de la cabeza cada 20 µseg. Si el tiempo de búsqueda de pista a pista es de 800 µseg, pasarán 40 sectores durante la búsqueda, por lo que la desviación de los cilindros debe ser de 40 sectores en vez de los tres sectores que se muestran en la figura 5-26. Vale la pena mencionar que el cambio entre una cabeza y otra también requiere un tiempo finito, por lo que hay también **desajuste de cabezas** al igual que desajuste de cilindros, pero el desajuste de las cabezas no es muy grande.

Como resultado del formato de bajo nivel se reduce la capacidad del disco, dependiendo de los tamaños del preámbulo, el hueco entre sectores y el ECC, así como el número de sectores adicionales reservados. A menudo la capacidad con formato es 20% menor que la capacidad sin formato. Los sectores adicionales no cuentan para la capacidad con formato, por lo que todos los discos de un tipo dado tienen exactamente la misma capacidad cuando se envían, sin importar cuántos sectores defectuosos tengan en realidad (si el número de sectores defectuosos excede al número de adicionales, la unidad se rechazará y no se enviará).

Hay una considerable confusión acerca de la capacidad del disco, debido a que ciertos fabricantes anunciaban la capacidad sin formato para que sus unidades parecieran más grandes de lo que realmente son. Por ejemplo, considere una unidad cuya capacidad sin formato es de 200×10^9 bytes. Éste se podría vender como un disco de 200 GB. Sin embargo, después del formato tal vez sólo haya 170×10^9 bytes para los datos. Para aumentar la confusión, el sistema operativo proba-

blemente reportará esta capacidad como 158 GB, y no como 170 GB debido a que el software considera que una memoria de 1 GB es de 2^{30} (1,073,741,824) bytes, no 10^9 (1,000,000,000) bytes.

Para empeorar las cosas, en el mundo de las comunicaciones de datos, 1 Gbps significa 1,000,000,000 bits/seg, ya que el prefijo *giga* en realidad significa 10^9 (un kilómetro equivale a 1000 metros, no 1024 metros, después de todo). Sólo con los tamaños de las memorias y los discos kilo, mega, giga y tera significan 2^{10},2^{20}, 2^{30} y 2^{40}, respectivamente.

El formato también afecta al rendimiento. Si un disco de 10,000 RPM tiene 300 sectores por pista de 512 bytes cada uno, se requieren 6 mseg para leer los 153,600 bytes en una pista para una velocidad de transferencia de datos de 25,600,000 bytes/seg, o 24.4 MB/seg. No es posible ir más rápido que esto, sin importar qué tipo de interfaz esté presente, aun si es una interfaz SCSI a 80 MB/seg o 160 MB/seg.

En realidad, para leer de manera continua a esta velocidad se requiere un búfer extenso en el controlador. Por ejemplo, considere un controlador con un búfer de un sector que ha recibido un comando para leer dos sectores consecutivos. Después de leer el primer sector del disco y de realizar el cálculo del ECC, los datos se deben transferir a la memoria principal. Mientras se está llevando a cabo esta transferencia, el siguiente sector pasará por la cabeza. Cuando se complete la copia a la memoria, el controlador tendrá que esperar casi un tiempo de rotación completo para que el segundo sector vuelva a pasar por la cabeza.

Este problema se puede eliminar al enumerar los sectores en forma entrelazada al aplicar formato al disco. En la figura 5-27(a) podemos ver el patrón de enumeración usual (ignorando aquí el desajuste de cilindros). La figura 5-27(b) muestra el **entrelazado simple**, que proporciona al controlador cierto espacio libre entre los sectores consecutivos para poder copiar el búfer a la memoria principal.

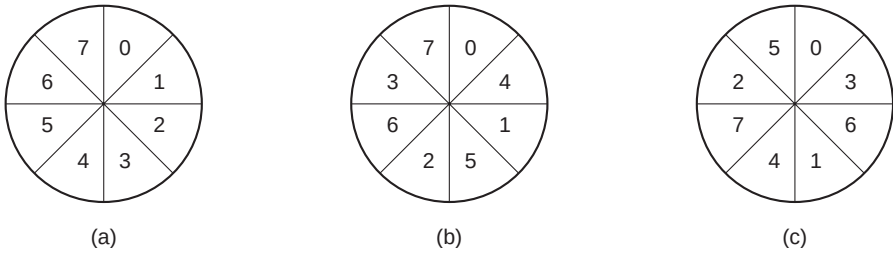

Figura 5-27. (a) Sin entrelazado. (b) Entrelazado simple. (c) Entrelazado doble.

Si el proceso de copia es muy lento, tal vez sea necesario el **entrelazado doble** de la figura 5-27(c). Si el controlador tiene un búfer de un sector solamente, no importa si la copia del búfer a la memoria principal se realiza mediante el controlador, la CPU principal o un chip de DMA; de todas formas requiere cierto tiempo. Para evitar la necesidad del entrelazado, el controlador debe ser capaz de colocar en el búfer una pista completa. Muchos controladores modernos pueden hacer esto.

Una vez que se completa el formato de bajo nivel, el disco se particiona. En sentido lógico, cada partición es como un disco separado. Las particiones son necesarias para permitir que coexistan varios sistemas operativos. Además, en algunos casos se puede utilizar una partición para el intercambio. En el Pentium y la mayoría de las otras computadoras, el sector 0 contiene el **registro de inicio**

maestro (MBR, por sus siglas en inglés), el cual contiene cierto código de inicio además de la tabla de particiones al final. La tabla de particiones proporciona el sector inicial y el tamaño de cada partición. En el Pentium, la tabla de particiones tiene espacio para cuatro particiones. Si todas ellas son para Windows, se llamarán C:, D:, E: y F:, y se tratarán como unidades separadas. Si tres de ellas son para Windows y una es para UNIX, entonces Windows llamará a sus particiones C:, D: y E:. El primer CD-ROM será entonces F:. Para poder iniciar del disco duro, una partición se debe marcar como activa en la tabla de particiones.

El paso final de preparación de un disco para utilizarlo es realizar un **formato de alto nivel** de cada partición (por separado). Esta operación establece un bloque de inicio, la administración del espacio de almacenamiento libre (lista de bloques libres o mapa de bits), el directorio raíz y un sistema de archivos vacío. También coloca un código en la entrada de la tabla de particiones para indicarle qué sistema de archivos se va a utilizar en la partición, debido a que muchos sistemas operativos soportan varios sistemas de archivos incompatibles (por cuestiones históricas). En este punto se puede iniciar el sistema.

Cuando se enciende la máquina, el BIOS se ejecuta al inicio, después lee el registro de inicio maestro y salta al mismo. Este programa de inicio comprueba a su vez cuál partición está activa. Después lee el sector de inicio de esa partición y lo ejecuta. El sector de inicio contiene un pequeño programa que por lo general carga un programa de inicio más grande, el cual busca en el sistema de archivos el kernel del sistema operativo. Ese programa se carga en memoria y se ejecuta.

5.4.3 Algoritmos de programación del brazo del disco

En esta sección analizaremos ciertas cuestiones relacionadas con los controladores de disco en general. En primer lugar hay que considerar cuánto tiempo se requiere para leer o escribir en un bloque de disco. El tiempo requerido se determina en base a tres factores:

1. Tiempo de búsqueda (el tiempo para desplazar el brazo al cilindro apropiado).
2. Retraso rotacional (el tiempo para que el sector apropiado se coloque debajo de la cabeza).
3. Tiempo de transferencia de datos actual.

Para la mayoría de los discos, el tiempo de búsqueda domina los otros dos tiempos, por lo que al reducir el tiempo de búsqueda promedio se puede mejorar el rendimiento del sistema de manera considerable. Si el software controlador de disco acepta peticiones una a la vez, y las lleva a cabo en ese orden, es decir, **Primero en llegar, primero en ser atendido** (*First-Come, Firsts-Served,* FCFS), no se puede hacer mucho para optimizar el tiempo de búsqueda. Sin embargo, hay otra estrategia posible cuando el disco está muy cargado. Es probable que mientras el brazo esté realizando una búsqueda a favor de una petición, otros procesos puedan generar otras peticiones de disco. Muchos controladores de disco mantienen una tabla, indexada por número de cilindro, con todas las peticiones pendientes para cada cilindro encadenadas en una lista enlazada, encabezada por las entradas en la tabla.

Dado este tipo de estructura de datos, podemos realizar mejorías con base en el algoritmo de programación del primero en llegar, primero en ser atendido. Para ver cómo, considere un disco imaginario con 40 cilindros. Llega una petición para leer un bloque en el cilindro 11; mientras la

búsqueda en el cilindro 11 está en progreso, llegan nuevas peticiones para los cilindros 1, 36, 16, 34, 9 y 12, en ese orden, y se introducen en la tabla de peticiones pendientes, con una lista enlazada separada para cada cilindro. Las peticiones se muestran en la figura 5-28.

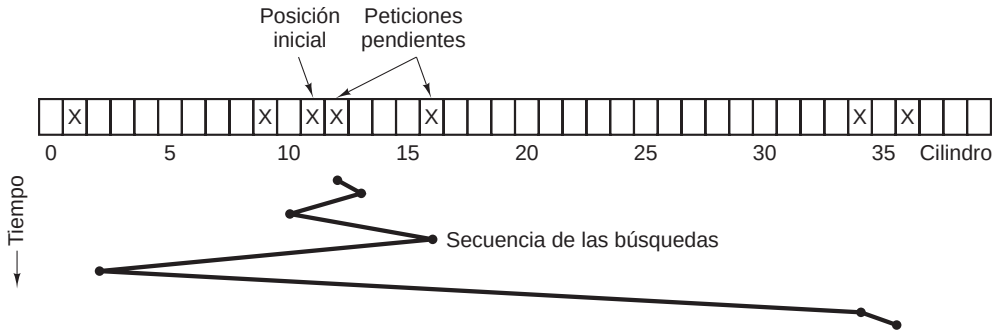

Figura 5-28. Algoritmo de planificación de disco de La búsqueda del trabajo más corto primero (SSF).

Cuando termina la petición actual (para el cilindro 11), el software controlador de disco puede elegir qué petición manejar a continuación. Utilizando FCFS, el siguiente cilindro sería el 1, luego el 36, y así en lo sucesivo. Este algoritmo requiere movimientos del brazo de 10, 35, 20, 18, 25 y 3, respectivamente, para un total de 111 cilindros.

De manera alternativa, siempre podría manejar la petición más cercana primero, para minimizar el tiempo de búsqueda. Dadas las peticiones de la figura 5-28, la secuencia es 12, 9, 16, 1, 34 y 36, y se muestra mediante la línea dentada en la parte inferior de la figura 5-28. Con esta secuencia, los movimientos del brazo son 1, 3, 7, 15, 33 y 2 para un total de 61 cilindros. Este algoritmo de **La búsqueda del trabajo más corto primero** (*Shortest Seek First,* SSF) recorta el movimiento total del brazo casi a la mitad, en comparación con FCFS.

Por desgracia, SSF tiene un problema. Suponga que siguen llegando más peticiones mientras se están procesando las peticiones de la figura 5-28. Por ejemplo, si después de ir al cilindro 16 hay una nueva petición para el cilindro 8, esa petición tendrá prioridad sobre el cilindro 1. Si después llega una petición para el cilindro 13, el brazo irá a continuación a 13, en vez de ir a 1. Con un disco que contenga mucha carga, el brazo tenderá a permanecer en la parte media del disco la mayor parte del tiempo, por lo que las peticiones en cualquier extremo tendrán que esperar hasta que una fluctuación estadística en la carga ocasione que ya no haya peticiones cerca de la parte media. Las peticiones alejadas de la parte media pueden llegar a obtener un mal servicio. Los objetivos del tiempo de respuesta mínimo y la equidad están en conflicto aquí.

Los edificios altos también tienen que lidiar con esta concesión. El problema de planificar un elevador en un edificio alto es similar al de planificar un brazo de disco. Las peticiones llegan en forma continua, llamando al elevador a los pisos (cilindros) al azar. La computadora que opera el elevador podría llevar fácilmente la secuencia en la que los clientes oprimieron el botón de llamada, y atenderlos utilizando FCFS o SSF.

Sin embargo, la mayoría de los elevadores utilizan un algoritmo distinto para poder reconciliar las metas mutuamente conflictivas de eficiencia y equidad. Siguen avanzando en la misma direc-

ción hasta que no haya más peticiones pendientes en esa dirección, y después cambian de direcciones. Este algoritmo, conocido como **algoritmo del elevador** tanto en el mundo de los discos como en el mundo de los elevadores, requiere que el software mantenga 1 bit: el bit de dirección actual, *ARRIBA* o *ABAJO*. Cuando termina una petición, el software controlador del disco o del elevador comprueba el bit. Si es *ARRIBA*, el brazo o cabina se desplaza a la siguiente petición pendiente de mayor prioridad. Si no hay peticiones pendientes en posiciones mayores, el bit de dirección se invierte. Cuando el bit se establece en *ABAJO*, el movimiento es a la siguiente posición de petición con menor prioridad, si la hay.

En la figura 5-29 se muestra el algoritmo del elevador usando las mismas siete peticiones que en la figura 5-28, suponiendo que el bit de dirección haya sido *ARRIBA* en un principio. El orden en el que se da servicio a los cilindros es 12, 16, 34, 36, 9 y 1, que produce los movimientos del brazo de 1, 4, 18, 2, 27 y 8, para un total de 60 cilindros. En este caso, el algoritmo del elevador es ligeramente mejor que SSF, aunque por lo general es peor. Una buena propiedad que tiene el algoritmo del elevador es que dada cierta colección de peticiones, el límite superior en el movimiento total es fijo: es sólo el doble del número de cilindros.

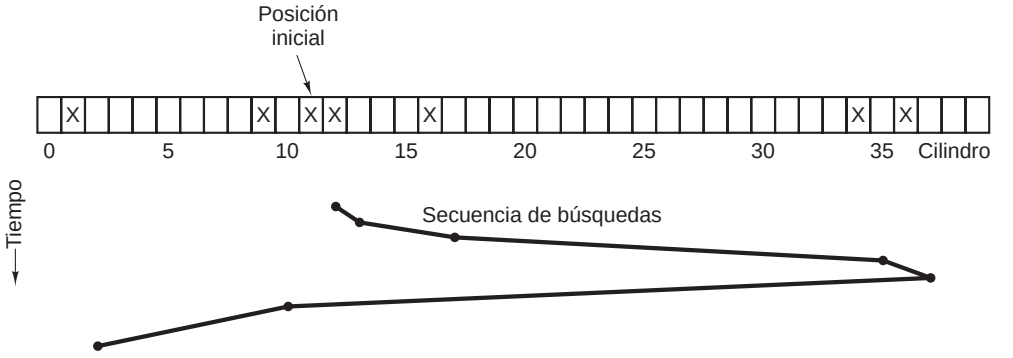

Figura 5-29. El algoritmo del elevador para planificar las peticiones de disco.

Una ligera modificación de este algoritmo que tiene una variación menor en tiempos de respuesta (Teory, 1972) es explorar siempre en la misma dirección. Cuando el cilindro de mayor numeración con una petición pendiente ha sido atendido, el brazo pasa al cilindro de menor numeración con una petición pendiente y después continúa desplazándose en dirección hacia arriba. En efecto, se considera que el cilindro de menor numeración está justo por encima del cilindro de mayor numeración.

Algunos controladores de disco proveen una forma para que el software inspeccione el número de sector actual bajo la cabeza. Con dicho controlador es posible otra optimización. Si hay dos o más peticiones para el mismo cilindro pendientes, el controlador puede emitir una petición para el sector que pasará bajo la cabeza a continuación. Observe que cuando hay varias pistas presentes en un cilindro, las peticiones consecutivas pueden ser para distintas pistas sin castigo. El controlador puede seleccionar cualquiera de sus cabezas casi en forma instantánea (la selección de cabezas no implica un movimiento del brazo ni un retraso rotacional).

Si el disco tiene la propiedad de que el tiempo de búsqueda es mucho más rápido que el retraso rotacional, entonces se debe utilizar una optimización distinta. Las peticiones pendientes deben ordenarse por número de sector, y tan pronto como el siguiente sector esté a punto de pasar por debajo de la cabeza, el brazo debe moverse rápidamente a la pista correcta para leer o escribir.

Con un disco duro moderno, los retrasos de búsqueda y rotacional dominan tanto el rendimiento que es muy ineficiente leer uno o dos sectores a la vez. Por esta razón, muchos controladores de disco siempre leen y colocan en la caché varios sectores, aún y cuando sólo se solicite uno. Por lo general, cualquier solicitud para leer un sector hará que se lea ese sector y gran parte de (o toda) la pista actual, dependiendo de qué tanto espacio haya disponible en la memoria caché del controlador. El disco descrito en la figura 5-18 tiene una caché de 4 MB, por ejemplo. El uso de la caché se determina en forma dinámica mediante el controlador. En su modo más simple, la caché se divide en dos secciones, una para lecturas y una para escrituras. Si una lectura subsiguiente se puede satisfacer de la caché del controlador, puede devolver los datos solicitados de inmediato.

Vale la pena recalcar que la caché del controlador de disco es completamente independiente de la caché del sistema operativo. Por lo general, la caché del controlador contiene bloques que en realidad no se han solicitado, pero que era conveniente leer debido a que pasaron por debajo de la cabeza como efecto secundario de alguna otra lectura. Por el contrario, cualquier caché mantenida por el sistema operativo consistirá en bloques que se hayan leído de manera explícita y que el sistema operativo considere que podrían ser necesarios de nuevo en un futuro cercano (por ejemplo, un bloque de disco que contiene un bloque de directorio).

Cuando hay varias unidades presentes en el mismo controlador, el sistema operativo debe mantener una tabla de peticiones pendiente para cada unidad por separado. Cada vez que una unidad está inactiva, debe emitirse una búsqueda para desplazar su brazo al cilindro en donde se necesitará a continuación (suponiendo que el controlador permita búsquedas traslapadas). Cuando termine la transferencia actual, se puede realizar una comprobación para ver si hay unidades posicionadas en el cilindro correcto. Si hay una o más, la siguiente transferencia se puede iniciar en una unidad que ya se encuentre en el cilindro correcto. Si ningún brazo está en la posición correcta, el software controlador deberá emitir una nueva búsqueda en la unidad que haya terminado una transferencia y deberá esperar hasta la siguiente interrupción para ver cuál brazo llega primero a su destino.

Es importante tener en cuenta que todos los algoritmos de planificación de disco anteriores asumen de manera tácita que la geometría de disco real es igual que la geometría virtual. Si no es así, entonces no tiene sentido planificar las peticiones de disco, ya que el sistema operativo en realidad no puede saber si el cilindro 40 o el cilindro 200 está más cerca del cilindro 39. Por otra parte, si el controlador de disco puede aceptar varias peticiones pendientes, puede utilizar estos algoritmos de planificación en forma interna. En ese caso, los algoritmos siguen siendo válidos, pero un nivel más abajo, dentro del controlador.

5.4.4 Manejo de errores

Los fabricantes de disco constantemente exceden los límites de la tecnología al incrementar las densidades de bits lineales. Una pista a la mitad de un disco de 5.25 pulgadas tiene una circunferencia de 300 mm aproximadamente. Si la pista contiene 300 sectores de 512 bytes, la densidad

de grabación lineal puede ser de aproximadamente 5000 bits/mm, tomando en consideración el hecho de que se pierde cierto espacio debido a los preámbulos, ECCs y huecos entre sectores. Para grabar 5000 bits/mm se requiere un sustrato en extremo uniforme y una capa de óxido muy fina. Por desgracia, no es posible fabricar un disco con tales especificaciones sin defectos. Tan pronto como la tecnología de fabricación haya mejorado hasta el grado en que sea posible operar sin errores a dichas densidades, los diseñadores de discos buscarán densidades más altas para incrementar la capacidad. Al hacer esto, probablemente vuelvan a introducir defectos.

Los defectos de fabricación incluyen sectores defectuosos; es decir, sectores que no leen correctamente el valor que se acaba de escribir en ellos. Si el defecto es muy pequeño, por ejemplo, de unos cuantos bits, es posible utilizar el sector defectuoso y sólo dejar que el ECC corrija los errores cada vez. Si el defecto es mayor, el error no se puede enmascarar.

Hay dos métodos generales para los bloques defectuosos: lidiar con ellos en el controlador o lidiar con ellos en el sistema operativo. En el primer método, antes de que el disco salga de la fábrica, se prueba y se escribe una lista de sectores defectuosos en el disco. Cada sector defectuoso se sustituye por uno de los sectores adicionales.

Hay dos formas de realizar esta sustitución. En la figura 5-30(a) podemos ver una sola pista de disco con 30 sectores de datos y dos sectores adicionales. El sector 7 está defectuoso. Lo que el controlador puede hacer es reasignar uno de los sectores adicionales como el sector 7, como se muestra en la figura 5-30(b). La otra forma es desplazar todos los sectores una posición hacia arriba, como se muestra en la figura 5-30(c). En ambos casos, el controlador tiene que saber cuál sector es cuál. Puede llevar el registro de esta información mediante tablas internas (una por pista), o volviendo a escribir los preámbulos para proporcionar los números de los sectores reasignados. Si se vuelven a escribir los preámbulos, el método de la figura 5-30(c) presenta más dificultad (debido a que se tienen que volver a escribir 23 preámbulos) pero en última instancia ofrece un mejor rendimiento, ya que de todas formas se puede leer una pista completa en una rotación.

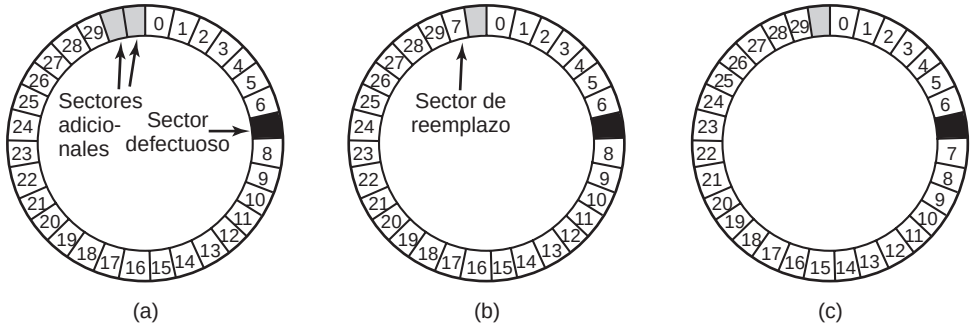

Figura 5-30. (a) Una pista de disco con un sector defectuoso. (b) Sustitución de un sector adicional por un sector defectuoso. (c) Desplazar todos los sectores para omitir el defectuoso.

También se pueden desarrollar errores durante la operación normal, una vez que se haya instalado la unidad. La primera línea de defensa al obtener un error que el ECC no puede manejar es tratar de leerlo otra vez. Algunos errores de lectura son transitorios; es decir, son producidos por

partículas de polvo bajo la cabeza y desaparecerán en un segundo intento. Si el controlador detecta que está obteniendo errores repetidos en cierto sector, puede cambiar a un sector de reemplazo antes de que el sector defectuoso falle para siempre. De esta manera no se pierden datos; el sistema operativo y el usuario ni siquiera se enteran del problema. Por lo general, se tiene que utilizar el método de la figura 5-30(b) debido a que los otros sectores podrían contener ahora datos. Para utilizar el método de la figura 5-30(c) no sólo habría que volver a escribir los preámbulos, sino también copiar todos los datos.

Anteriormente dijimos que había dos métodos generales para manejar errores: manejarlos en el controlador o en el sistema operativo. Si el controlador no tiene la capacidad de reasignar sectores de manera transparente como hemos visto, el sistema operativo tiene que hacer lo mismo en software. Esto significa que primero debe adquirir una lista de sectores defectuosos, ya sea leyéndolos del disco, o simplemente probando todo el disco en sí. Una vez que sepa cuáles sectores están defectuosos, puede construir tablas de reasignación. Si el sistema operativo desea utilizar el método de la figura 5-30(c), debe cambiar los datos en los sectores del 7 al 29, un sector hacia arriba.

Si el sistema operativo se está haciendo cargo de la reasignación, debe asegurarse que no haya sectores malos en ningún archivo, y que tampoco haya en la lista libre o mapa de bits. Una manera de hacer esto es crear un archivo que consista de todos los sectores malos. Si este archivo no se introduce en el sistema de archivos, los usuarios no lo leerán por accidente (o peor aún, lo liberarán).

Sin embargo, hay otro problema: los respaldos. Si el disco se respalda archivo por archivo, es importante que la herramienta de respaldo no trate de copiar el archivo con el bloque defectuoso. Para evitar esto, el sistema operativo tiene que ocultar el archivo con el bloque defectuoso tan bien que ni siquiera una herramienta de respaldo pueda encontrarlo. Si el disco se respalda sector por sector, en vez de archivo por archivo, será difícil (si no imposible) evitar errores de lectura durante el respaldo. La única esperanza es que el programa de respaldo tenga suficiente inteligencia para rendirse después de 10 lecturas fallidas y continúe con el siguiente sector.

Los sectores defectuosos no son la única fuente de errores. También ocurren errores de búsqueda producidos por problemas mecánicos en el brazo. El controlador lleva el registro de la posición del brazo de manera interna. Para realizar una búsqueda, emite una serie de pulsos en el motor del brazo, un pulso por cilindro, para desplazarlo al nuevo cilindro. Cuando el brazo llega a su destino, el controlador lee el número del cilindro actual del preámbulo del siguiente sector. Si el brazo está en la posición incorrecta, ha ocurrido un error de búsqueda.

La mayoría de los controladores de disco duro corrigen los errores de búsqueda de manera automática, pero la mayoría de los controladores de disco flexible (incluyendo el del Pentium) sólo establecen un bit de error y dejan el resto al controlador, que a su vez maneja este error emitiendo un comando `recalibrate`, para alejar el brazo lo más que sea posible y restablecer la idea interna del controlador del cilindro actual a 0. Por lo general, esto resuelve el problema. Si no es así, la unidad se debe reparar.

Como hemos visto, el controlador es en realidad una pequeña computadora especializada, completa con software, variables, búferes y, en algunas ocasiones, errores. Algunas veces una secuencia inusual de eventos, como una interrupción en una unidad que ocurre de manera simultánea con un comando `recalibrate` para otra unidad desencadenará un error y hará que el controlador entre en un ciclo o que pierda la cuenta de lo que estaba haciendo. Los diseñadores de los contro-

ladores por lo general planean para lo peor y proporcionan una terminal en el chip que, cuando se le aplica una señal, obliga al controlador a olvidar lo que estaba haciendo y se restablece a sí mismo. Si todo lo demás falla, el controlador de disco puede establecer un bit para invocar esta señal y restablecer el controlador. Si eso no ayuda, todo lo que puede hacer el software controlado es imprimir un mensaje y darse por vencido.

Al recalibrar un disco éste hace un ruido extraño, pero que por lo general no es molesto. Sin embargo, hay una situación en la que la recalibración es un grave problema: los sistemas con restricciones de tiempo real. Cuando se está reproduciendo un video de un disco duro, o cuando se queman archivos de un disco duro en un CD-ROM, es esencial que los bits salgan del disco duro a una velocidad uniforme. Bajo estas circunstancias, las recalibraciones insertan huecos en el flujo de bits y por lo tanto son inaceptables. Para tales aplicaciones hay disponibles unidades especiales, conocidas como **discos AV** (**discos Audio Visuales**), que nunca se tienen que recalibrar.

5.4.5 Almacenamiento estable

Como hemos visto, los discos algunas veces cometen errores. Los sectores buenos pueden de pronto volverse defectuosos. Discos completos pueden dejar de funcionar inesperadamente. Los RAIDs protegen contra el hecho de que unos cuantos sectores se vuelvan defectuosos, o incluso que falle una unidad. Sin embargo, no protegen contra los errores de escritura que establecen datos defectuosos en primer lugar. Tampoco protegen contra las fallas durante las escrituras que corrompen los datos originales sin reemplazarlos con datos nuevos.

Para algunas aplicaciones es esencial que los datos nunca se pierdan o corrompan, incluso frente a errores de disco o de CPU. En teoría, un disco simplemente debe trabajar todo el tiempo sin errores. Por desgracia, eso no se puede lograr. Lo que se puede lograr es un subsistema de disco que tenga la siguiente propiedad: cuando se emita una escritura, el disco debe escribir correctamente los datos o no hacer nada, dejando los datos existentes intactos. A dicho sistema se le conoce como **almacenamiento estable** y se implementa en software (Lampson y Sturgis, 1979). El objetivo es mantener el disco consistente a toda costa. A continuación analizaremos una ligera variante de la idea original.

Antes de describir el algoritmo, es importante tener un modelo claro de los posibles errores. El modelo asume que cuando un disco escribe un bloque (uno o más sectores), la escritura es correcta o incorrecta y este error se puede detectar en una lectura subsiguiente, al examinar los valores de los campos ECC. En principio, no es posible una detección de errores garantizada debido a que, por ejemplo, con un campo ECC de 16 bytes que protege a un sector de 512 bytes, hay 2^{4096} valores de datos y sólo 2^{144} valores de ECC. Por ende, si se daña la información de un bloque durante la escritura pero el ECC no, hay miles de millones sobre miles de millones de combinaciones incorrectas que producen el mismo ECC. Si ocurre cualquiera de ellas, el error no se detectará. En general, la probabilidad de que datos aleatorios tengan el ECC de 16 bytes apropiado es de casi 2^{-144}, lo suficientemente pequeño como para poder llamarlo cero, aunque en realidad no lo es.

El modelo también supone que un sector escrito correctamente puede volverse defectuoso de manera repentina, quedando ilegible. Sin embargo, la suposición es que dichos eventos son tan raros que la probabilidad de que el mismo sector se vuelva defectuoso en una segunda unidad

(independiente) durante un intervalo de tiempo razonable (por ejemplo, 1 día) es lo bastante pequeña como para ignorarla.

El modelo supone además que la CPU puede fallar, en cuyo caso sólo se detiene. Cualquier escritura de disco en progreso durante la falla también se detiene, lo cual produce datos incorrectos en un sector y un ECC incorrecto que se puede detectar posteriormente. Bajo todas estas condiciones, el almacenamiento estable puede ser 100% confiable en el sentido de que las operaciones de escritura funcionen correctamente o de lo contrario se dejarán los datos antiguos en su lugar. Desde luego que no protege contra desastres físicos, como la ocurrencia de un terremoto y que la computadora caiga por una fisura de 100 metros y aterrice en un charco de magma hirviente. Es difícil recuperarse de esta condición mediante software.

El almacenamiento estable utiliza un par de discos idénticos, en donde los bloques correspondientes funcionan en conjunto para formar un bloque libre de errores. En la ausencia de errores, los bloques correspondientes en ambas unidades son iguales. Cualquiera se puede leer para obtener el mismo resultado. Para lograr este objetivo, se definen las siguientes tres operaciones:

1. **Escrituras estables**. Una escritura estable consiste en primero escribir el bloque en la unidad 1, y después volver a leerlo para verificar que se haya escrito correctamente. Si no se escribió en forma correcta, las operaciones de escribir y volver a leer se realizan de nuevo, hasta *n* veces hasta que funcione. Después de *n* fallos consecutivos, el bloque se reasigna a un bloque adicional y la operación se repite hasta tener éxito, sin importar cuántos bloques adicionales haya que probar. Una vez que tiene éxito la escritura en la unidad 1, se escribe el bloque correspondiente en la unidad 2 y se vuelve a leer, repetidas veces si es necesario, hasta que también tiene éxito esta operación. En la ausencia de fallas de la CPU, cuando se completa una escritura estable, el bloque se ha escrito correctamente en ambas unidades y se ha verificado también en ambas.

2. **Lecturas estables**. En una lectura estable primero se lee el bloque de la unidad 1. Si esto produce un ECC incorrecto, la operación de lectura se vuelve a intentar hasta *n* veces. Si todas estas operaciones producen ECCs defectuosos, se lee el bloque correspondiente de la unidad 2. Dado el hecho de que una escritura estable exitosa deja dos copias correctas del bloque, y en base a nuestra suposición de que la probabilidad de que el mismo bloque se vuelva defectuoso de manera repentina en ambas unidades en un intervalo de tiempo razonable es insignificante, siempre se obtiene una lectura estable.

3. **Recuperación de fallas**. Después de una falla, un programa de recuperación explora ambos discos y compara los bloques correspondientes. Si un par de bloques están bien y son iguales, no se hace nada. Si uno de ellos tiene un error de ECC, el bloque defectuoso se sobrescribe con el bloque bueno correspondiente. Si un par de bloques están bien pero son distintos, el bloque de la unidad 1 se escribe en la unidad 2.

En ausencia de fallas de la CPU, este esquema funciona debido a que las escrituras estables siempre escriben dos copias válidas de cada bloque, y se asume que los errores espontáneos nunca ocurrirán en ambos bloques correspondientes al mismo tiempo. ¿Y qué tal en la presencia de fallas

de la CPU durante escrituras estables? Depende del momento preciso en el que ocurra la falla. Hay cinco posibilidades, como se muestra en la figura 5.31.

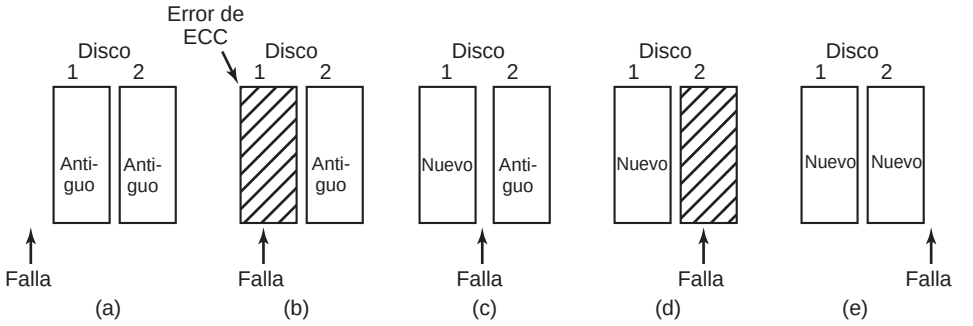

Figura 5-31. Análisis de la influencia de fallas en escrituras estables.

En la figura 5-31(a), la falla en la CPU ocurre antes de escribir cualquiera de las dos copias del bloque. Durante la recuperación no se cambiará ninguno de los dos bloques y el valor anterior seguirá existiendo, lo cual está permitido.

En la figura 5-31(b), la CPU falla durante la escritura en la unidad 1, destruyendo el contenido del bloque. Sin embargo, el programa de recuperación detecta este error y restaura el bloque en la unidad 1 con el bloque en la unidad 2. Por ende, el efecto de la falla se borra y el estado anterior se restaura por completo.

En la figura 5-31(c), la falla de la CPU ocurre una vez que se escribe en la unidad 1, pero antes de escribir en la unidad 2. Aquí se ha traspasado el punto donde no hay retorno: el programa de recuperación copia el bloque de la unidad 1 a la unidad 2. La escritura tiene éxito.

La figura 5-31(d) es como la figura 5-31(b): durante la recuperación, el bloque bueno sobrescribe al bloque malo. De nuevo, el valor final de ambos bloques es el nuevo.

Por último, en la figura 5-31(e) el programa de recuperación ve que ambos bloques son iguales, por lo que no se cambia ninguno y la escritura tiene éxito también.

Hay varias optimizaciones y mejoras posibles para este esquema. Para empezar, la acción de comparar todos los bloques en pares después de una falla es posible de hacer, pero costosa. Una enorme mejora es llevar la cuenta de cuál bloque se estaba escribiendo durante una escritura estable, de manera que se tenga que comprobar sólo un bloque durante la recuperación. Algunas computadoras tienen una pequeña cantidad de **RAM no volátil**, la cual es una memoria CMOS especial, energizada por una batería de litio. Dichas baterías duran años, posiblemente durante toda la vida de la computadora. A diferencia de la memoria principal, que se pierde después de una falla, no se pierde la RAM no volátil durante una falla. La hora del día se mantiene aquí por lo general (y se incrementa mediante un circuito especial), lo cual explica por qué las computadoras siguen sabiendo qué hora es, aún después de que se hayan desconectado.

Suponga que hay unos cuantos bytes de RAM no volátil disponibles para fines del sistema operativo. La escritura estable puede colocar el número del bloque que está a punto de actualizar en la RAM no volátil antes de empezar la escritura. Una vez que se completa con éxito la escritura estable,

el número de bloque en la RAM no volátil se sobrescribe con un número de bloque inválido; por ejemplo, −1. Bajo estas condiciones, después de una falla el programa de recuperación puede verificar la RAM no volátil para ver si había una escritura estable en progreso durante la falla, y de ser así, cuál bloque se estaba escribiendo cuando ocurrió la falla. Después se puede comprobar que las dos copias del bloque sean correctas y consistentes.

Si la RAM no volátil no está disponible, se puede simular de la siguiente manera. Al inicio de una escritura estable, se sobrescribe un bloque de disco fijo en la unidad 1 con el número del bloque en el que se va a realizar la escritura estable. Después este bloque se vuelve a leer para verificarlo. Después de obtener el bloque correcto, se escribe y verifica el bloque correspondiente en la unidad 2. Cuando la escritura estable se completa correctamente, ambos bloques se sobrescriben con un número de bloque inválido y se verifican. De nuevo, después de una falla es fácil determinar si había o no una escritura estable en progreso durante la falla. Desde luego que esta técnica requiere ocho operaciones de disco adicionales para escribir un bloque estable, por lo que debería utilizarse sólo en casos muy necesarios.

Vale la pena mencionar un último punto. Hicimos la suposición de que sólo ocurre un cambio espontáneo de un bloque bueno a un bloque defectuoso por cada bloque a diario. Si pasan suficientes días, el otro también podría volverse defectuoso. Por lo tanto, una vez al día se debe realizar una exploración completa de ambos discos para reparar cualquier daño. De esta forma, cada mañana ambos discos siempre son idénticos. Incluso si ambos bloques en un par se vuelven defectuosos dentro de un periodo de unos cuantos días, todos los errores se reparan en forma correcta.

5.5 RELOJES

Los **relojes** (también conocidos como **temporizadores**) son esenciales para la operación de cualquier sistema de multiprogramación, por una variedad de razones. Mantienen la hora del día y evitan que un proceso monopolice la CPU, entre otras cosas. El software de reloj puede tomar la forma de un software controlador de dispositivo, aun y cuando un reloj no es un dispositivo de bloque (como un disco) ni un dispositivo de carácter (como un ratón). Nuestro análisis de los relojes seguirá el mismo patrón que en la sección anterior: primero un vistazo al hardware del reloj y después un vistazo a su software.

5.5.1 Hardware de reloj

Hay dos tipos de relojes de uso común en las computadoras, y ambos son bastante distintos de los relojes que utilizan las personas. Los relojes más simples están enlazados a la línea de energía de 110 o 220 voltios y producen una interrupción en cada ciclo de voltaje, a 50 o 60 Hz. Estos relojes solían dominar el mercado, pero ahora son raros.

El otro tipo de reloj se construye a partir de tres componentes: un oscilador de cristal, un contador y un registro contenedor, como se muestra en la figura 5-32. Cuando una pieza de cristal de cuarzo se corta en forma apropiada y se monta bajo tensión, puede generar una señal periódica con una precisión muy grande, por lo general en el rango de varios cientos de megahertz, dependiendo del cristal elegido. Mediante el uso de componentes electrónicos, esta señal base puede multiplicar-

se por un pequeño entero para obtener frecuencias de hasta 1000 MHz o incluso más. Por lo menos uno de esos circuitos se encuentra comúnmente en cualquier computadora, el cual proporciona una señal de sincronización para los diversos circuitos de la misma. Esta señal se alimenta al contador para hacer que cuente en forma descendente hasta cero. Cuando el contador llega a cero, produce una interrupción de la CPU.

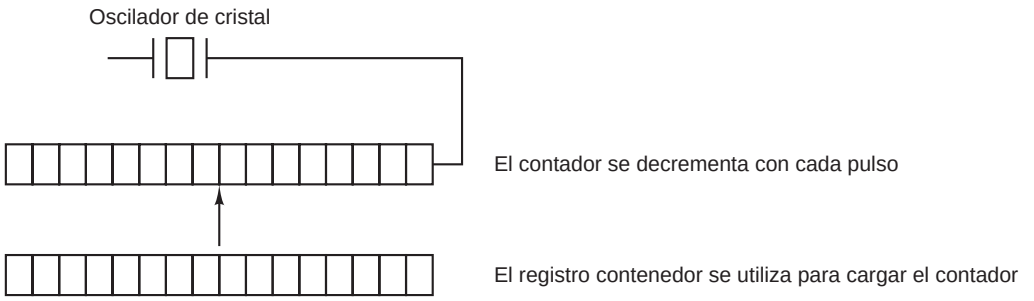

Figura 5-32. Un reloj programable.

Por lo general, los relojes programables tienen varios modos de operación. En el **modo de un solo disparo**, cuando se inicia el reloj copia el valor del registro contenedor en el contador y después decrementa el contador en cada pulso del cristal. Cuando el contador llega a cero, produce una interrupción y se detiene hasta que vuelve a ser iniciado en forma explícita mediante el software. En el **modo de onda cuadrada**, después de llegar a cero y producir la interrupción, el registro contenedor se copia automáticamente en el contador y todo el proceso se repite de nuevo en forma indefinida. Estas interrupciones periódicas se conocen como **pulsos de reloj**.

La ventaja del reloj programable es que su frecuencia de interrupción se puede controlar mediante software. Si se utiliza un cristal de 500 MHz, entonces se aplica un pulso al contador cada 2 nseg. Con registros de 32 bits (sin signo), se pueden programar interrupciones para que ocurran a intervalos de 2 nseg hasta 8.6 seg. Los chips de reloj programables por lo general contienen dos o tres relojes que pueden programarse de manera independiente, y tienen muchas otras opciones también (por ejemplo, contar en forma ascendente o descendente, deshabilitar interrupciones, y más).

Para evitar que se pierda la hora actual cuando se apaga la computadora, la mayoría cuentan con un reloj de respaldo energizado por batería, implementado con el tipo de circuitos de baja energía que se utilizan en los relojes digitales. El reloj de batería puede leerse al iniciar el sistema. Si no está presente, el software puede pedir al usuario la fecha y hora actuales. También hay una forma estándar para que un sistema obtenga la hora actual de un host remoto. En cualquier caso, la hora se traduce en el número de pulsos de reloj desde las 12 A.M. **UTC** (*Universal Coordinated Time*, Tiempo coordenado universal) (anteriormente conocido como Tiempo del meridiano de Greenwich) el 1 de enero de 1970, como lo hace UNIX, o a partir de algún otro momento de referencia. El origen del tiempo para Windows es enero 1, 1980. En cada pulso de reloj, el tiempo real se incrementa por un conteo. Por lo general se proporcionan programas utilitarios para

establecer el reloj del sistema y el reloj de respaldo en forma manual, y para sincronizar los dos relojes.

5.5.2 Software de reloj

Todo lo que hace el hardware del reloj es generar interrupciones a intervalos conocidos. Todo lo demás que se relacione con el tiempo debe ser realizado por el software controlador del reloj. Las tareas exactas del controlador del reloj varían de un sistema operativo a otro, pero por lo general incluyen la mayoría de las siguientes tareas:

1. Mantener la hora del día.
2. Evitar que los procesos se ejecuten por más tiempo del que tienen permitido.
3. Contabilizar el uso de la CPU.
4. Manejar la llamada al sistema `alarm` que realizan los procesos de usuario.
5. Proveer temporizadores guardianes (watchdogs) para ciertas partes del mismo sistema.
6. Realizar perfilamiento, supervisión y recopilación de estadísticas.

La primera función del reloj, mantener la hora del día (también conocida como **tiempo real**), no es difícil. Sólo requiere incrementar un contador en cada pulso de reloj, como se dijo antes. Lo único por lo que debemos estar al pendiente es el número de bits en el contador de la hora del día. Con una velocidad de reloj de 60 Hz, un contador de 32 bits se desbordará aproximadamente en 2 años. Es evidente que el sistema no puede almacenar el tiempo real como el número de pulsos del reloj desde enero 1, 1970 en 32 bits.

Se pueden usar tres métodos para resolver este problema. El primero es utilizar un contador de 64 bits, aunque para ello se vuelve más complejo dar mantenimiento al contador, ya que se tiene que realizar muchas veces por segundo. El segundo es mantener la hora del día en segundos, en vez de pulsos, usando un contador subsidiario para contar pulsos hasta que se haya acumulado un segundo completo. Como 2^{32} segundos equivale a más de 136 años, este método funcionará hasta el siglo XXII.

El tercer método es contar en pulsos, pero hacerlo de manera relativa a la hora en que se inició el sistema, en vez de un momento externo fijo. Cuando el reloj de respaldo se lee o el usuario escribe la hora real, el tiempo de inicio del sistema se calcula a partir del valor de la hora del día actual y se almacena en memoria, en cualquier forma conveniente. Más adelante, cuando se solicita la hora del día, se suma al contador la hora almacenada del día para obtener la hora actual. Los tres métodos se muestran en la figura 5-33.

La segunda función del reloj es evitar que los procesos se ejecuten en demasiado tiempo. Cada vez que se inicia un proceso, el planificador inicializa un contador para el valor del quantum de ese proceso, en pulsos de reloj. En cada interrupción del reloj, el software controlador del mismo decrementa el contador del quantum en 1. Cuando llega a cero, el software controlador del reloj llama al planificador para establecer otro proceso.

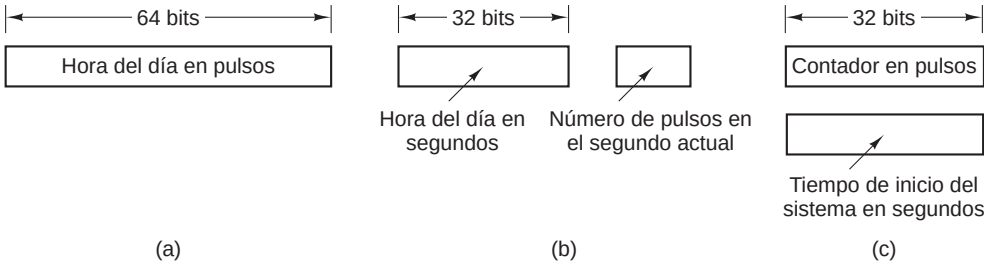

Figura 5-33. Tres formas de mantener la hora del día.

La tercera función del reloj es contabilizar el uso de la CPU. La forma más precisa de hacerlo es iniciar un segundo temporizador, distinto del temporizador principal del sistema, cada vez que se inicia un proceso. Cuando ese proceso se detiene, el temporizador se puede leer para saber cuánto tiempo se ha ejecutado el proceso. Para hacer lo correcto, el segundo temporizador se debe guardar cuando ocurre una interrupción y se debe restaurar cuando ésta termine.

Una manera menos precisa pero más simple de realizar la contabilidad es mantener en una variable global un apuntador a la entrada en la tabla de procesos para el proceso actual en ejecución. En cada pulso de reloj se incrementa un campo en la entrada del proceso actual. De esta forma, cada pulso de reloj se "carga" al proceso en ejecución al momento del pulso. Un problema menor con esta estrategia es que, si ocurren muchas interrupciones durante la ejecución de un proceso, de todas formas se le carga un pulso completo, aun cuando no haya podido realizar mucho trabajo. La contabilidad apropiada para la CPU durante las interrupciones es demasiado complicada y se realiza en raras ocasiones.

En muchos sistemas, un proceso puede solicitar que el sistema operativo le proporcione una advertencia después de cierto intervalo. La advertencia es por lo general una señal, interrupción, mensaje o algo similar. Una aplicación que requiere dichas advertencias es el trabajo en red, donde un paquete que no se reconozca durante cierto intervalo de tiempo se debe volver a transmitir. Otra aplicación es la instrucción asistida por computadora, donde un estudiante que no provee una respuesta dentro de cierto tiempo la recibe automáticamente.

Si el software controlador de reloj tuviera suficientes relojes, podría establecer un reloj separado para cada petición. Como éste no es el caso, debe simular varios relojes virtuales con un solo reloj físico. Una forma de hacer esto es mantener una tabla en la que se mantenga el tiempo de la señal de todos los temporizadores pendientes, así como una variable que proporcione el tiempo de la señal del siguiente. Cada vez que se actualiza la hora del día, el controlador comprueba que haya ocurrido la señal más cercana. Si así fue, se busca en la tabla la siguiente entrada que está por ocurrir.

Si se esperan muchas señales, es más eficiente simular varios relojes al encadenar todas las peticiones de reloj pendientes, ordenadas con base en el tiempo, en una lista enlazada como se muestra en la figura 5.34. Cada entrada en la lista indica cuántos pulsos de reloj después del anterior hay que esperar para poder producir una señal. En este ejemplo, hay señales pendientes para 4203, 4207, 4213, 4215 y 4216.

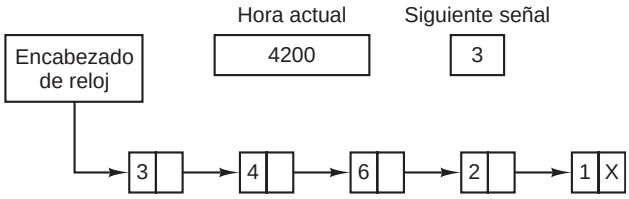

Figura 5-34. Simulación de varios temporizadores con un solo reloj.

En la figura 5-34, la siguiente interrupción ocurre en 3 pulsos. En cada pulso se decrementa *Siguiente señal*. Cuando llega a 0 se produce la señal correspondiente al primer elemento en la lista, y ese elemento se elimina de la misma. Después *Siguiente señal* se establece al valor en la entrada que se encuentra ahora en la cabeza de la lista, que en este ejemplo es 4.

Observe que durante una interrupción de reloj, el software controlador del mismo tiene varias cosas por hacer: incrementar el tiempo real, decrementar el quantum y comprobar que sea 0, realizar la contabilidad del tiempo de la CPU y decrementar el contador de la alarma. Sin embargo, cada una de estas operaciones se han diseñado con cuidado para que sean muy rápidas, debido a que tienen que repetirse muchas veces por segundo.

Hay partes del sistema operativo que también necesitan establecer temporizadores. A éstos se les conoce como **temporizadores guardianes** (*watchdogs*). Por ejemplo, los discos flexibles no giran cuando no están en uso, para evitar desgaste en el medio y la cabeza del disco. Cuando se necesitan datos de un disco flexible, primero se debe iniciar el motor. Sólo cuando el disco flexible está girando a toda velocidad se puede iniciar la operación de E/S. Cuando un proceso intenta leer de un disco flexible inactivo, el software controlador del mismo inicia el motor y después establece un temporizador guardián para que produzca una interrupción cuando haya pasado un intervalo de tiempo suficientemente extenso (debido a que no hay un interruptor de máxima velocidad en el disco flexible).

El mecanismo utilizado por el software controlador del reloj para manejar temporizadores guardianes es igual que para las señales de usuario. La única diferencia es que cuando un temporizador se desactiva, en vez de producir una señal, el software controlador de reloj llama a un procedimiento proporcionado por el que hizo la llamada. El procedimiento es parte del código del proceso que hizo la llamada. El procedimiento al que se llamó puede hacer lo que sea necesario, incluso hasta producir una interrupción, aunque dentro del kernel las interrupciones son a menudo inconvenientes y las señales no existen. Esta es la razón por la que se proporciona el mecanismo del guardián. Vale la pena observar que este mecanismo sólo funciona cuando el software controlador de reloj y el procedimiento que se va a llamar se encuentran en el mismo espacio de direcciones.

Lo último en nuestra lista es el perfilamiento. Algunos sistemas operativos proporcionan un mecanismo mediante el cual un programa de usuario puede hacer que el sistema construya un histograma de su contador del programa, para poder ver en dónde pasa su tiempo. Cuando el perfilamiento es una posibilidad, en cada pulso el controlador comprueba si se está perfilando el proceso actual y, de ser así, calcula el número de receptáculo (un rango de direcciones) que corresponde al contador del programa actual. Después incrementa ese receptáculo en uno. Este mecanismo también se puede utilizar para perfilar al mismo sistema.

5.5.3 Temporizadores de software

La mayoría de las computadoras tienen un segundo reloj programable que se puede establecer para producir interrupciones del temporizador, a cualquier velocidad que requiera un programa. Este temporizador es aparte del temporizador principal del sistema, cuyas funciones se describieron en la sección anterior. Mientras que la frecuencia de interrupción sea baja, no habrá problema al usar este segundo temporizador para fines específicos de la aplicación. El problema surge cuando la frecuencia del temporizador específico de la aplicación es muy alta. A continuación analizaremos de manera breve un esquema de un temporizador basado en software que funciona bien bajo ciertas circunstancias, inclusive a frecuencias mucho mayores. La idea se debe a Aron y Druschel (1999). Para obtener más detalles, consulte su artículo.

En general hay dos formas de manejar las interrupciones de E/S y el sondeo. Las interrupciones tienen baja latencia; es decir, ocurren justo después del evento en sí, con muy poco o nada de retraso. Por otra parte, con las CPUs modernas las interrupciones tienen una sobrecarga considerable, debido a la necesidad del cambio de contexto y su influencia en la canalización, el TLB y la caché.

La alternativa para las interrupciones es hacer que la aplicación sondee el evento que espera por sí misma. Al hacer esto se evitan las interrupciones, pero puede haber una latencia considerable debido a que puede ocurrir un evento directamente después de un sondeo, en cuyo caso espera casi todo un intervalo de sondeo. En promedio, la latencia equivale a la mitad del intervalo de sondeo.

Para ciertas aplicaciones, no es aceptable la sobrecarga de las interrupciones ni la latencia del sondeo. Por ejemplo, considere una red de alto rendimiento como Gigabit Ethernet. Esta red es capaz de aceptar o enviar un paquete de tamaño completo cada 12 µseg. Para ejecutarse con un rendimiento óptimo en la salida, debe enviarse un paquete cada 12 µseg.

Una forma de lograr esta velocidad es hacer que al completarse la transmisión de un paquete, se produzca una interrupción o se establezca el segundo temporizador para que produzca una interrupción cada 12 µseg. El problema es que al medir esta interrupción en un Pentium II de 300 MHz, tarda 4.45 µseg (Aron y Druschel, 1999). Esta sobrecarga apenas si es mejor que la de las computadoras en la década de 1970. Por ejemplo, en la mayoría de las minicomputadoras una interrupción requería cuatro ciclos: para poner el contador del programa y el PSW en la pila y para cargar un nuevo contador de programa y PSW. Actualmente, al lidiar con la línea de tuberías, la MMU, el TLB y la caché, aumenta la sobrecarga en forma considerable. Es probable que estos efectos empeoren en vez de mejorar con el tiempo, cancelando así las velocidades mayores de reloj.

Los **temporizadores de software** evitan las interrupciones. En vez de ello, cada vez que el kernel se ejecuta por alguna otra razón, justo antes de regresar al modo de usuario comprueba el reloj de tiempo real para ver si ha expirado un temporizador de software. Si el temporizador ha expirado, se realiza el evento programado (por ejemplo, transmitir paquetes o comprobar si llegó un paquete), sin necesidad de cambiar al modo de kernel debido a que el sistema ya se encuentra ahí. Una vez realizado el trabajo, el temporizador de software se restablece para empezar de nuevo. Todo lo que hay que hacer es copiar el valor actual del reloj en el temporizador y sumarle el intervalo de tiempo de inactividad.

Los temporizadores de software marcan o descienden con la velocidad a la que se realizan entradas en el kernel por otras razones. Estas razones son:

1. Llamadas al sistema.
2. Fallos del TLB.
3. Fallos de página.
4. Interrupciones de E/S.
5. La CPU queda inactiva.

Para ver con qué frecuencia ocurrían estos eventos, Aron y Druschel realizaron mediciones con varias cargas de la CPU, incluyendo un servidor Web con carga completa, un servidor Web con un trabajo en segundo plano para realizar cálculos, la reproducción de audio en tiempo real en Internet, y la recompilación del kernel de UNIX. La velocidad de entrada promedio en el kernel tuvo una variación de 2 µseg a 18 µseg, donde casi la mitad de estas entradas eran llamadas al sistema. Así, para una aproximación de primer orden, es viable hacer que un temporizador de software se active cada 12 µseg, aunque fallando en un tiempo límite en ocasiones. Para las aplicaciones como el envío de paquetes o el sondeo de paquetes entrantes, tener un retraso de 10 µseg de vez en cuando es mejor que tener interrupciones que ocupen 35% de la CPU.

Desde luego que habrá periodos en los que no haya llamadas al sistema, fallos del TLB o de página, en cuyo caso no se activarán temporizadores. Para colocar un límite superior en estos intervalos, se puede establecer el segundo temporizador de hardware para que se active cada 1 mseg, por ejemplo. Si la aplicación puede sobrevivir con sólo 1000 paquetes/seg por intervalos ocasionales, entonces la combinación de temporizadores de software y un temporizador de hardware de baja frecuencia puede ser mejor que la E/S controlada sólo por interrupciones o por sondeo.

5.6 INTERFACES DE USUARIO: TECLADO, RATÓN, MONITOR

Toda computadora de propósito general tiene un teclado y un monitor (y por lo general un ratón) para permitir que las personas interactúen con ella. Aunque el teclado y el monitor son dispositivos técnicamente separados, trabajan muy de cerca. En los mainframes con frecuencia hay muchos usuarios remotos, cada uno con un dispositivo que contiene un teclado y una pantalla conectados como una unidad. Estos dispositivos se conocen históricamente como **terminales**. Con frecuencia, las personas siguen utilizando ese término, aun cuando hablan sobre los teclados y monitores de las computadoras personales (en gran parte debido a que no hay otro mejor).

5.6.1 Software de entrada

La entrada del usuario proviene principalmente del teclado y del ratón; analicemos estos dispositivos. En una computadora personal, el teclado contiene un microprocesador integrado que por lo general se comunica, a través de un puerto serial especializado, con un chip controlador en la tarjeta principal (aunque cada vez con más frecuencia, los teclados se conectan a un puerto USB). Se ge-

nera una interrupción cada vez que se oprime una tecla, y se genera una segunda interrupción cada vez que se suelta. En cada una de estas interrupciones de teclado, el software controlador del mismo extrae la información acerca de lo que ocurre desde el puerto de E/S asociado con el teclado. Todo lo demás ocurre en el software y es muy independiente del hardware.

La mayoría del resto de esta sección se puede comprender mejor si se considera que se escriben comandos en una ventana de shell (interfaz de línea de comandos). Así es como trabajan comúnmente los programadores. A continuación analizaremos las interfaces gráficas.

Software de teclado

El número en el puerto de E/S es el número de tecla, conocido como **código de exploración**, no el código ASCII. Los teclados tienen menos de 128 teclas, por lo que sólo se necesitan 7 bits para representar el número de tecla. El octavo bit se establece en 0 cuando se oprime una tecla, y en 1 cuando se suelta. Es responsabilidad del controlador llevar un registro del estado de cada tecla (oprimida o suelta).

Por ejemplo, cuando se oprime la tecla *A* el código de exploración (30) se coloca en un registro de E/S. Es responsabilidad del controlador determinar si es minúscula, mayúscula, CTRL-A, ALT-A, CTRL-ALT-A o alguna otra combinación. Como el controlador puede saber qué teclas se han oprimido y todavía no se han liberado (por ejemplo, MAYÚS), tiene suficiente información para realizar el trabajo.

Por ejemplo, la secuencia de teclas

OPRIMIR MAYÚS, OPRIMIR A, SOLTAR A, SOLTAR MAYÚS

indica una A mayúscula. Sin embargo, la secuencia de teclas

OPRIMIR MAYÚS, OPRIMIR A, SOLTAR MAYÚS, SOLTAR A

indica también una A mayúscula. Aunque esta interfaz de teclado pone toda la carga en el software, es en extremo flexible. Por ejemplo, los programas de usuario pueden querer saber si un dígito que se acaba de escribir provino de la fila superior de teclas, o del teclado numérico lateral. En principio, el controlador puede proporcionar esta información.

Se pueden adoptar dos filosofías posibles para el controlador. En la primera, el trabajo del controlador es sólo aceptar la entrada y pasarla hacia arriba sin modificarla. Un programa que lee del teclado obtiene una secuencia pura de códigos ASCII (dar a los programas de usuario los códigos de exploración es demasiado primitivo, y se genera una alta dependencia del teclado).

Esta filosofía se adapta bien a las necesidades de los editores de pantalla sofisticados como *emacs*, que permite al usuario enlazar una acción arbitraria a cualquier carácter o secuencia de caracteres. Sin embargo, significa que si el usuario escribe *dste* en vez de *date* y luego corrige el error oprimiendo tres veces la tecla retroceso y *ate*, seguido de un retorno de carro, el programa de usuario recibirá los 11 códigos ASCII que se teclearon, de la siguiente manera:

d s t e _ _ _ a t e CR

No todos los programas desean tanto detalle. A menudo sólo quieren la entrada corregida y no la secuencia exacta sobre cómo se produjo. Esta observación conlleva a la segunda filosofía: el con-

trolador maneja toda la edición entre líneas, y envía sólo las líneas corregidas a los programas de usuario. La primera filosofía está orientada a caracteres; la segunda está orientada a líneas. En un principio se conocieron como **modo crudo** y **modo cocido**, respectivamente. El estándar POSIX utiliza el término **modo canónico** para describir el modo orientado a líneas. El **modo no canónico** es equivalente al modo crudo, aunque es posible cambiar muchos detalles del comportamiento. Los sistemas compatibles con POSIX proporcionan varias funciones de biblioteca posibilitan la selección de uno de esos modos y se pueden modificar muchos parámetros.

Si el teclado está en modo canónico (cocido), los caracteres se deben almacenar hasta que se haya acumulado una línea completa, debido a que tal vez el usuario decida posteriormente borrar parte de ella. Incluso si el teclado está en modo crudo, el programa tal vez no haya solicitado entrada todavía, por lo que los caracteres se deben colocar en un búfer para permitir la escritura adelantada. Se puede utilizar un búfer dedicado o se pueden asignar búferes de una reserva. El primer método impone un límite fijo en la escritura adelantada; el segundo no. Esta cuestión surge con más fuerza cuando el usuario escribe en una ventana de shell (ventana de línea de comandos en Windows) y acaba de emitir un comando (como una compilación) que no se ha completado todavía. Los siguientes caracteres que se escriban tienen que colocarse en un búfer, debido a que el shell no está listo para leerlos. Los diseñadores de sistemas que no permiten a los usuarios escribir por adelantado deberían ser bañados con brea y emplumados, o peor aún, ser obligados a utilizar su propio sistema.

Aunque el teclado y el monitor son dispositivos lógicamente separados, muchos usuarios han crecido acostumbrados a ver los caracteres que acaban de escribir aparecer en la pantalla. A este proceso se le conoce como **producir eco (*echo*)**.

La producción del eco se complica debido al hecho de que un programa puede estar escribiendo en la pantalla mientras el usuario teclea (de nuevo, piense en escribir en una ventana de shell). Cuando menos, el controlador del teclado tiene que averiguar dónde colocar la nueva entrada sin que sea sobrescrita por la salida del programa.

La producción de eco también se complica cuando hay que mostrar más de 80 caracteres en una ventana con líneas de 80 caracteres (o algún otro número). Dependiendo de la aplicación, puede ser apropiado que los caracteres se ajusten a la siguiente línea. Algunos controladores simplemente truncan las líneas a 80 caracteres, al descartar todos los caracteres más allá de la columna 80.

Otro problema es el manejo de los tabuladores. Por lo general es responsabilidad del controlador calcular la posición del cursor, tomando en cuenta tanto la salida de los programas como la salida debido al eco, y calcular el número apropiado de espacios que se van a imprimir con eco.

Ahora llegamos al problema de la equivalencia de dispositivos. Lógicamente, al final de una línea de texto queremos un retorno de carro, para desplazar el cursor de vuelta a la columna 1, y un avance de línea para pasar a la siguiente línea. No sería muy atractivo requerir que los usuarios escribieran ambos caracteres al final de cada línea. Es responsabilidad del controlador de dispositivo convertir lo que llegue como entrada en el formato utilizado por el sistema operativo. En UNIX, la tecla INTRO se convierte en un avance de página para su almacenamiento interno; en Windows se convierte en un retorno de carro seguido de un avance de línea.

Si la forma estándar es sólo almacenar un avance de línea (la convención de UNIX), entonces los retornos de carro (creados por la tecla Intro) se deben convertir en avances de línea. Si el for-

mato interno es almacenar ambos (la convención de Windows), entonces el controlador debe generar un avance de línea cuando reciba un retorno de carro, y un retorno de carro cuando obtenga un avance de línea. Sin importar cuál sea la convención interna, el monitor puede requerir que se produzca eco tanto de un avance de línea como un retorno de carro para poder actualizar la pantalla de manera apropiada. En los sistemas multiusuario como las mainframes, distintos usuarios pueden tener distintos tipos de terminales conectadas y es responsabilidad del controlador del teclado convertir todas las distintas combinaciones de retorno de carro/avance de línea en el estándar interno del sistema, y encargarse de que todo el eco se produzca correctamente.

Cuando se opera en modo canónico, algunos caracteres de entrada tienen significado especial. La figura 5-35 muestra todos los caracteres especiales requeridos por POSIX. Los valores predeterminados son todos caracteres de control que no deben estar en conflicto con la entrada de texto o los códigos utilizados por los programas; todos excepto los últimos dos se pueden cambiar bajo el control del programa.

Carácter	Nombre POSIX	Comentario
CTRL-H	ERASE	Retrocede un carácter
CTRL-U	KILL	Borra toda la línea que se está escribiendo
CTRL-V	LNEXT	Interpreta el siguiente carácter literalmente
CTRL-S	STOP	Detiene la salida
CTRL-Q	START	Inicia la salida
DEL	INTR	Proceso de interrupción (SIGINT)
CTRL-\	QUIT	Obliga un vaciado de núcleo (SIGQUIT)
CTRL-D	EOF	Fin de archivo
CTRL-M	CR	Retorno de carro (no se puede modificar)
CTRL-J	NL	Avance de línea (no se puede modificar)

Figura 5-35. Caracteres que se manejan de manera especial en modo canónico.

El carácter *ERASE* permite al usuario borrar el carácter que acaba de escribir. Por lo general es el carácter de retroceso (CTRL-H). No se agrega a la cola de caracteres, sino que elimina el carácter anterior de la cola. Su eco se debe producir como una secuencia de tres caracteres —retroceso, espacio y retroceso— para poder eliminar el carácter anterior de la pantalla. Si el carácter anterior era un tabulador, poder borrarlo depende de la manera en que se procesó cuando se escribió. Si se expande de inmediato en espacios, se necesita cierta información adicional para determinar cuánto hay que retroceder. Si el mismo tabulador se almacena en la cola de entrada, se puede eliminar y sólo se imprime toda la línea completa de nuevo. En la mayoría de los sistemas, la tecla retroceso sólo borra caracteres en la línea actual. No borrará un retorno de carro y retrocederá hasta la línea anterior.

Cuando el usuario detecta un error al principio de la línea que está escribiendo, a menudo es conveniente borrar toda la línea y empezar de nuevo. El carácter *KILL* borra toda la línea. La ma-

yoría de los sistemas hacen que la línea borrada desaparezca de la pantalla, pero unos cuantos más antiguos producen su eco además de un retorno de carro y un avance de línea, ya que a algunos usuarios les gusta ver la línea anterior. En consecuencia, la forma de producir el eco de *KILL* es cuestión de gusto. Al igual que con *ERASE*, por lo general no es posible regresar más allá de la línea actual. Cuando se elimina un bloque de caracteres, tal vez sea conveniente o no que el software controlador devuelva los búferes a la reserva, si es que se utilizan.

Algunas veces los caracteres *ERASE* o *KILL* se deben introducir como datos ordinarios. El carácter *LNEXT* sirve como un **carácter de escape**. En UNIX, CTRL-V es el predeterminado. Como ejemplo, los sistemas UNIX anteriores utilizaban a menudo el signo @ para *KILL*, pero el sistema de correo de Internet utiliza direcciones de la forma *linda@cs.washington.edu*. Alguien que se sienta más cómodo con las convenciones antiguas podría redefinir a *KILL* como @, pero entonces tendría que introducir literalmente un signo @ para utilizar el correo electrónico. Para ello se puede escribir CTRL-V @. El carácter CTRL-V en sí se puede introducir literalmente al escribir CTRL-V CTRL-V. Después de ver un CTRL-V, el software controlador establece una bandera que indica que el siguiente carácter está exento de un procesamiento especial. El carácter *LNEXT* en sí no se introduce en la cola de caracteres.

Para permitir que los usuarios eviten que una imagen de pantalla se salga de la vista, se proporcionan códigos de control para congelar la pantalla y reiniciarla posteriormente. En UNIX, estos códigos son *STOP*, (CTRL-S) y *START*, (CTRL-Q), respectivamente. No se almacenan pero se utilizan para establecer y borrar una bandera en la estructura de datos del teclado. Cada vez que se intenta una operación de salida, se inspecciona la bandera. Si está activada, no se lleva a cabo la operación de salida. Por lo general, el eco también se suprime junto con la salida del programa.

A menudo es necesario eliminar un programa fugitivo que se está depurando. Para este fin se pueden utilizar los caracteres *INTR* (SUPR) y *QUIT* (CTRL-\). En UNIX, SUPR envía la señal SIGINT a todos los procesos que inician a partir de ese teclado. Implementar SUPR puede ser algo engañoso, ya que UNIX se diseñó desde un principio para manejar varios usuarios a la vez. Por ende, en el caso general puede haber muchos procesos ejecutándose a beneficio de muchos usuarios, y la tecla SUPR sólo debe señalar a los procesos de ese usuario. La parte difícil es llevar la información del controlador a la parte del sistema que se encarga de las señales, la que después de todo, no ha pedido esta información.

CTRL-\ es similar a SUPR, excepto porque envía la señal SIGQUIT, que obliga a realizar un vaciado del núcleo si no se atrapa o se ignora. Cuando se oprime una de estas teclas, el controlador debe producir el eco de un retorno de carro y un avance de línea, y descartar toda la entrada acumulada para permitir empezar desde cero. El valor predeterminado para *INTR* es a menudo CTRL-C en vez de SUPR, ya que muchos programas utilizan SUPR en lugar de retroceso para editar.

Otro carácter especial es *EOF* (CTRL-D), que en UNIX hace que se cumpla cualquier petición de lectura pendiente para la terminal con los datos que estén disponibles en el búfer, incluso aunque esté vacío. Al escribir CTRL-D al inicio de una línea el programa obtiene una lectura de 0 bytes, que por convención se interpreta como fin de archivo y ocasiona que la mayoría de los programas actúen como si hubieran visto el fin de archivo en un archivo de entrada.

Software de ratón

La mayoría de las PCs tienen un ratón, o algunas veces un *trackball*, que sencillamente es un ratón boca arriba. Un tipo común de ratón tiene una bola de goma en su interior que se asoma por un hoyo en la parte inferior y gira, a medida que el ratón se desplaza por una superficie dura, frotándose contra unos rodillos posicionados en ejes ortogonales. El movimiento en la dirección este-oeste hace que gire el eje paralelo al eje y; el movimiento en la dirección norte-sur hace que gire el eje paralelo al eje x.

Otro tipo popular de ratón es el óptico, que está equipado con uno o más diodos emisores de luz y fotodetectores en su parte inferior. Los primeros tenían que operar sobre un tapete especial grabado con una rejilla rectangular, de manera que el ratón pudiera contar las líneas que cruzaba. Los ratones ópticos modernos tienen un chip de procesamiento de imágenes en ellos y sacan fotos continuas de baja resolución de la superficie debajo de ellos, buscando cambios de imagen en imagen.

Cada vez que un ratón se ha desplazado cierta distancia mínima en cualquier dirección, o cuando se oprime o suelta un botón, se envía un mensaje a la computadora. La distancia mínima es de aproximadamente 0.1 mm (aunque se puede establecer mediante software). Algunas personas llaman a esta unidad **mickey**. Los ratones pueden tener uno, dos o tres botones, dependiendo de la estimación de los diseñadores en cuanto a la habilidad intelectual de los usuarios para llevar la cuenta de más de un botón. Algunos ratones tienen ruedas que pueden enviar datos adicionales a la computadora. Los ratones inalámbricos son iguales a los alámbricos, excepto que en vez de devolver sus datos a la computadora a través de un cable, utilizan radios de baja energía, por ejemplo mediante el uso del estándar **Bluetooth**.

El mensaje para la computadora contiene tres elementos: Δx, Δy, botones. El primer elemento es el cambio en la posición *x* desde el último mensaje. Después viene el cambio en la posición *y* desde el último mensaje. Por último, se incluye el estado de los botones. El formato del mensaje depende del sistema y del número de botones que tenga el ratón. Por lo general, requiere de 3 bytes. La mayoría de los ratones se reportan con la computadora un máximo de 40 veces/seg, por lo que el ratón se puede haber desplazado varios mickeys desde el último reporte.

Observe que el ratón indica sólo los cambios en la posición, no la posición absoluta en sí. Si se recoge el ratón y se coloca de nuevo en su posición gentilmente sin hacer que la bola gire, no se enviarán mensajes.

Algunas GUIs diferencian un solo clic y un doble clic de un botón del ratón. Si dos clics están lo bastante cerca en el espacio (mickeys) y también en el tiempo (milisegundos), se señala un doble clic. El valor máximo para “lo bastante cerca” depende del software, y por lo general el usuario puede ajustar ambos parámetros.

5.6.2 Software de salida

Ahora consideremos el software de salida. Primero analizaremos una salida de ejemplo a una ventana de texto, que es lo que los programadores prefieren utilizar comúnmente. Después consideraremos las interfaces gráficas de usuario, que otros usuarios prefieren a menudo.

Ventanas de texto

La salida es más simple que la entrada cuando se envía secuencialmente en un solo tipo de letra, tamaño y color. En su mayor parte, el programa envía caracteres a la ventana en uso y se muestran ahí. Por lo general, un bloque de caracteres (por ejemplo, una línea) se escribe en una llamada al sistema.

Los editores de pantalla y muchos otros programas sofisticados necesitan la capacidad de actualizar la pantalla en formas complejas, como sustituir una línea a mitad de la pantalla. Para satisfacer esta necesidad, la mayoría de los controladores de software de salida proporcionan una serie de comandos para desplazar el cursor, insertar y eliminar caracteres o líneas en el cursor, entre otras tareas. A menudo estos comandos se conocen como **secuencias de escape**. En los días de la terminal "tonta" ASCII de 25 x 80 había cientos de tipos de terminales, cada una con sus propias secuencias de escape. Como consecuencia, era difícil escribir software que funcionara en más de un tipo terminal.

Una solución que se introdujo en Berkeley UNIX fue una base de datos de terminales conocida como **termcap**. Este paquete de software definía una variedad de acciones básicas, como desplazar el cursor hacia (*fila*, *columna*). Para desplazar el cursor a una ubicación específica, el software (como un editor) utilizaba una secuencia de escape genérica que después se convertía en la secuencia de escape actual para la terminal en la que se estaba escribiendo. De esta forma, el editor funcionaba en cualquier terminal que tuviera una entrada en la base de datos termcap. La mayoría del software de UNIX aún funciona de esta manera, incluso en las computadoras personales.

En cierto momento, la industria vio la necesidad de estandarizar la secuencia de escape, por lo que se desarrolló un estándar ANSI. En la figura 5-36 se muestran unos cuantos valores.

Considere cómo estas secuencias de escape podrían ser utilizadas por un editor de texto. Suponga que el usuario escribe un comando indicando al editor que elimine toda la línea 3 y después cierre el hueco entre las líneas 2 y 4. El editor podría enviar la siguiente secuencia de escape sobre la línea serial a la terminal:

```
ESC [ 3 ; 1 H ESC [ 0 K ESC [ 1 M
```

(donde los espacios se utilizan sólo para separar los símbolos; no se transmiten). Esta secuencia desplaza el cursor al principio de la línea 3, borra toda la línea y luego elimina la línea ahora vacía, con lo cual provoca que todas las líneas que empiezan en 5 se desplacen una línea hacia arriba. Entonces la que era la línea 4 se convierte en la línea 3; la línea 5 se convierte en la línea 4, y así en lo sucesivo. Es posible utilizar secuencias de escape análogas para agregar texto a la parte media de la pantalla. Las palabras se pueden agregar o eliminar de manera similar.

El sistema X Window

Casi todos los sistemas UNIX basan su interfaz de usuario en el **Sistema X Window** (que a menudo sólo se le llama **X**), desarrollado en el M.I.T. como parte del proyecto Athena en la década de 1980. Es muy portátil y se ejecuta por completo en espacio de usuario. En un principio tenía como

Secuencia de escape	Significado
ESC [*n* A	Se desplaza *n* líneas hacia arriba
ESC [*n* B	Se desplaza *n* líneas hacia abajo
ESC [*n* C	Se desplaza *n* espacios a la derecha
ESC [*n* D	Se desplaza *n* espacios a la izquierda
ESC [*m*; *n* H	Desplaza el cursor a (*m*, *n*)
ESC [*s* J	Borra la pantalla del cursor (0 al final, 1 del inicio, 2 todo)
ESC [*s* K	Borra la línea del cursor (0 al final, 1 al inicio, 2 todo)
ESC [*n* L	Inserta *n* líneas en el cursor
ESC [*n* M	Elimina *n* líneas en el cursor
ESC [*n* P	Elimina *n* caracteres en el cursor
ESC [*n* @	Inserta *n* caracteres en el cursor
ESC [*n* m	Habilita la reproducción *n* (0=normal, 4=negritas, 5=parpadeo, 7=inverso)
ESC M	Desplaza la pantalla hacia atrás si el cursor está en la línea superior

Figura 5-36. Las secuencias de escape ANSI aceptadas por el software controlador de terminal en la salida. ESC denota el carácter de escape ASCII (0x1B) y *n*, *m* y *s* son parámetros numéricos opcionales.

propósito principal conectar un gran número de terminales de usuario remotas con un servidor de cómputo central, por lo que está dividido lógicamente en software cliente y software servidor, que puede ejecutarse potencialmente en distintas computadoras. En la mayoría de las computadoras modernas, ambas partes se pueden ejecutar en el mismo equipo. En los sistemas Linux, los populares entornos de escritorio Gnome y KDE se ejecutan encima de X.

Cuando X se ejecuta en un equipo, el software que recolecta la entrada del teclado y el ratón, y que escribe la salida en la pantalla, se llama **servidor X**. Este software tiene que llevar el registro de cuál ventana está seleccionada en un momento dado (dónde se encuentra el ratón), para saber a qué cliente debe enviar cualquier entrada nueva del teclado. Se comunica con los programas en ejecución (posiblemente a través de una red), llamados **clientes X**. Les envía la entrada del ratón y del teclado, y acepta los comandos de pantalla de ellos.

Puede parecer extraño que el servidor X siempre esté dentro de la computadora del usuario, mientras que el cliente X puede estar en un servidor de cómputo remoto, pero sólo piense en el trabajo principal del servidor X: mostrar bits en la pantalla, por lo cual tiene sentido que esté cerca del usuario. Desde el punto de vista del programa, es un cliente que le dice al servidor que haga cosas, como mostrar texto y figuras geométricas. El servidor (en la PC local) hace justo lo que se le dice, al igual que todos los servidores.

El arreglo de cliente y servidor se muestra en la figura 5.37 para el caso en el que el cliente X y el servidor X se encuentran en distintos equipos. Pero al ejecutar Gnome o KDE en un solo equipo, el cliente es sólo un programa de aplicación que utiliza la biblioteca X y habla con el servidor X en el mismo equipo (pero usando una conexión TCP sobre sockets, igual que en el caso remoto).

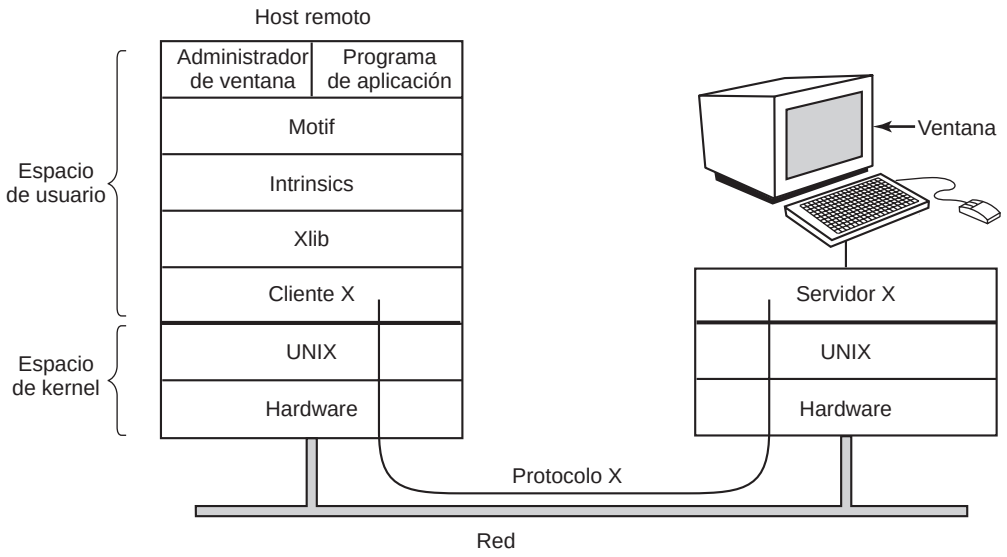

Figura 5-37. Clientes y servidores en el Sistema X Window del M.I.T.

La razón por la que es posible ejecutar el Sistema X Window encima de UNIX (o de cualquier otro sistema operativo) en una sola máquina o a través de una red, es que lo que X define en realidad es el protocolo X entre el cliente X y el servidor X, como se muestra en la figura 5-37. No importa si el cliente y el servidor están en el mismo equipo, separados por 100 metros a través de una red de área local o miles de kilómetros aparte y conectados por Internet. El protocolo y la operación del sistema es idéntico en todos los casos.

X es sólo un sistema de ventanas; no es una GUI completa. Para obtener una GUI completa, se ejecutan otras capas de software encima. Una de estas capas es **Xlib**, un conjunto de procedimientos de biblioteca para acceder a la funcionalidad de X. Estos procedimientos forman la base del Sistema X Window y son lo que examinaremos a continuación, pero son demasiado primitivos como para que la mayoría de los programas de usuario los utilicen de manera directa. Por ejemplo, cada clic del ratón se reporta por separado, así que el proceso para determinar que dos clics en realidad forman un doble clic se tiene que manejar arriba de Xlib.

Para facilitar la programación con X, se suministra un kit de herramientas como parte de X que consiste en **Intrinsics**. Este nivel maneja botones, barras de desplazamiento y otros elementos de la GUI, conocidos como **widgets**. Para crear una verdadera interfaz de GUI con una apariencia visual uniforme, se necesita otro nivel más (o varios de ellos). Un ejemplo es **Motif**, que se muestra en la figura 5-37 y forma la base del Entorno de Escritorio Común utilizado en Solaris y otros sistemas UNIX comerciales. La mayoría de las aplicaciones hacen uso de llamadas a Motif en vez de a Xlib. Gnome y KDE tienen una estructura similar a la de la figura 5-37, sólo que con distintas bibliotecas. Gnome utiliza la biblioteca GTK+ y KDE utiliza la biblioteca Qt. El caso de que tener dos GUIs sea mejor que una puede debatirse.

También vale la pena recalcar que la administración de ventanas no es parte de X. La decisión de omitir esta característica fue completamente intencional. En vez de ello, un proceso separado del cliente X, conocido como **administrador de ventana**, controla la creación, eliminación y movimiento de las ventanas en la pantalla. Para administrar las ventanas, envía comandos al servidor X para indicarle lo que debe hacer. A menudo se ejecuta en la misma máquina que el cliente X, pero en teoría se puede ejecutar en cualquier parte.

Este diseño modular, que consiste en varios niveles y programas, hace a X altamente portátil y flexible. Se ha llevado a la mayoría de las versiones de UNIX, incluyendo Solaris, todas las variantes de BSD, AIX, Linux, etcétera, de tal forma que los desarrolladores de aplicaciones pueden utilizar una interfaz de usuario estándar para varias plataformas. También se ha llevado a otros sistemas operativos. Por el contrario, en Windows los sistemas de ventanas y de GUI están mezclados en la GDI y se encuentran en el kernel, lo cual hace que sean más difíciles de mantener y, desde luego, no son portátiles.

Ahora vamos a analizar brevemente a X desde el nivel de Xlib. Cuando se inicia un programa de X, abre una conexión a uno o más servidores X, que vamos a llamar estaciones de trabajo, aun cuando se podrían colocar en el mismo equipo que el programa X en sí. X considera que esta conexión es confiable en cuanto a que los mensajes perdidos y duplicados se manejan mediante el software de red y no tiene que preocuparse por los errores de comunicación. Por lo general se utiliza TCP/IP entre el cliente y el servidor.

Pasan cuatro tipos de mensajes a través de la conexión:

1. Comandos de dibujo del programa a la estación de trabajo.
2. Respuestas de la estación de trabajo a las solicitudes del programa.
3. Mensajes del teclado, del ratón y de otros eventos.
4. Mensajes de error.

La mayoría de los comandos de dibujo se envían del programa a la estación de trabajo como mensajes de una sola vía. No se espera una respuesta. La razón de este diseño es que cuando los procesos cliente y servidor están en distintos equipos, se puede requerir un periodo considerable para que el comando llegue al servidor y se lleve a cabo. Si se bloquea el programa de aplicación durante este tiempo, se reduciría su velocidad sin necesidad. Por otra parte, cuando el programa necesita información de la estación de trabajo, simplemente tiene que esperar a que la respuesta regrese.

Al igual que Windows, X está en su mayor parte controlado por eventos. Los eventos fluyen de la estación de trabajo al programa, por lo general en respuesta a cierta acción humana, como las pulsaciones del teclado, los movimientos del ratón o la acción de descubrir una ventana. Cada mensaje de evento es de 32 bytes, en donde el primer byte indica el tipo de evento y los siguientes 31 bytes indican información adicional. Existen varias docenas de tipos de eventos, pero a un programa se le envían sólo los eventos que puede manejar. Por ejemplo, si un programa no está interesado en los eventos que ocurren cuando el usuario suelta una tecla, no se le envían eventos de liberación de tecla. Al igual que en Windows los eventos se ponen en cola, y los programas leen los eventos de la cola de entrada.

Sin embargo, a diferencia de Windows, el sistema operativo nunca llama a los procedimientos dentro del programa de aplicación por su cuenta. Ni siquiera sabe qué procedimiento maneja cuál evento.

Un concepto clave en X es el **recurso**. Un recurso es una estructura de datos que contiene cierta información. Los programas de aplicación crean recursos en las estaciones de trabajo. Los recursos se pueden compartir entre varios procesos en la estación de trabajo. Los recursos tienen un tiempo de vida corto y no sobreviven a los reinicios de la estación de trabajo. Algunos recursos típicos son las ventanas, los tipos de letra, los mapas de colores (paletas de colores), mapas de píxeles (mapas de bits), los cursores y los contextos gráficos. Estos últimos se utilizan para asociar las propiedades con las ventanas y son similares en concepto a los contextos de dispositivos en Windows.

En la figura 5-38 se muestra un esqueleto incompleto de un programa de X. Empieza por incluir ciertos encabezados requeridos y después declara algunas variables. Luego se conecta al servidor X especificado como el parámetro para *XOpenDisplay*. Después asigna un recurso de ventana y almacena un manejador para este recurso en *win*. En la práctica, aquí ocurriría cierta inicialización. Después de eso, le indica al administrador de ventanas que la nueva ventana existe, para que pueda administrarla.

La llamada a *XCreateGC* crea un contexto gráfico en el que se almacenan las propiedades de la ventana; en un programa más completo, podrían inicializarse en ese punto. La siguiente instrucción, que es la llamada a *XSelectInput*, indica al servidor X qué eventos está preparado el programa para manejar. En este caso está interesado en los clics de ratón, las pulsaciones de teclas y las ventanas que se descubren. En la práctica, un programa real estaría interesado también en otros eventos. Por último, la llamada a *XMapRaised* asigna la nueva ventana en la pantalla como la ventana de nivel más superior. En este punto, la ventana se vuelve visible en la pantalla.

El ciclo principal consiste en dos instrucciones y en sentido lógico es más simple que el ciclo correspondiente en Windows. La primera instrucción aquí obtiene un evento y la segunda envía el tipo de evento para su procesamiento. Cuando algún evento indica que el programa ha terminado, *en ejecución* se establece en 0 y el ciclo termina. Antes de salir, el programa libera el contexto gráfico, la ventana y la conexión.

Vale la pena mencionar que no a todos les gusta una GUI. Muchos programadores prefieren una interfaz de línea de comandos, del tipo que vimos en la sección 5.6.2 anterior. X se encarga de esto mediante un programa cliente llamado *xterm*. Este programa emula una venerable terminal inteligente VT102, completa con todas las secuencias de escape. Así, los editores como *vi* y *emacs* y demás software que utiliza termcap funcionan en estas ventanas sin modificación.

Interfaces gráficas de usuario

La mayoría de las computadoras personales ofrecen una **GUI** (*Graphic User Interface*, Interfaz gráfica de usuario).

La GUI fue inventada por Douglas Engelbart y su grupo de investigación en el Stanford Research Institute. Después fue copiada por los investigadores en Xerox PARC. Un buen día, Steve Jobs (co-fundador de Apple) estaba paseando por PARC y vio una GUI en una computadora Xerox

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
      Display pant;                                   /* identificador del servidor */
      Window vent;                                    /* identificador de ventana */
      GC gc;                                          /* identificador del contexto gráfico */
      XEvent evento;                                  /* almacenamiento para un evento */
      int enejecucion = 1;

      pant = XOpenDisplay("nombre_pantalla");         /* se conecta al servidor X */
      vent = XCreateSimpleWindow(pant, ...);          /* asigna memoria para la nueva ventana */
      XSetStandardProperties(pant, ...);              /* anuncia la ventana al admin. de ventanas */
      gc = XCreateGC(pant, vent, 0, 0);               /* crea el contexto gráfico */
      XSelectInput(pant,vent, ButtonPressMask | KeyPressMask | ExposureMask);
      XMapRaised(pant, vent);                         /* muestra la ventana; envía evento Expose */

      while (enejecucion) {
            XNextEvent(pant, &evento);                /* obtiene el siguiente evento */
            switch(evento.type) {
              case Expose:          …;  break;        /* vuelve a dibujar la ventana */
              case ButtonPress:     …;  break;        /* procesa click del ratón */
              case Keypress;        …;  break;        /* procesa entrada del teclado */
            }
      }

      XFreeGC(pant, gc);                              /* libera el contexto gráfico */
      XDestroyWindow(pant, vent);                     /* desasigna el espacio de memoria de la ventana */
      XCloseDisplay(pant);                            /* apaga la conexión de red */
}
```

Figura 5-38. Un esqueleto de un programa de aplicación de X Window.

y dijo algo así como: "¡Dios mío! Éste es el futuro de la computación". La GUI le dio la idea para una nueva computadora, que se convirtió en Lisa de Apple. Lisa era demasiado costosa y fue un fracaso comercial, pero su sucesora la Macintosh fue un enorme éxito.

Cuando Microsoft obtuvo un prototipo de la Macintosh para poder desarrollar Microsoft Office en ella, rogó a Apple para que otorgara la licencia a todos los que quisieran para convertirla en el nuevo estándar en la industria. (Microsoft hizo mucho más dinero con Office que con MS-DOS, por lo que estaba dispuesta a abandonar MS-DOS para tener una mejor plataforma para Office). El ejecutivo de Apple a cargo de la Macintosh, Jean-Louis Gassée, se rehusó y Steve Jobs ya no estaba ahí para contradecirlo. Con el tiempo Microsoft obtuvo una licencia para los elementos de la interfaz. Esto formó la base de Windows. Cuando Windows empezó a obtener popularidad, Apple demandó a Microsoft, arguyendo que Microsoft había excedido la licencia, pero el juez no estuvo

de acuerdo y Windows sobrepasó a la Macintosh. Si Gassée hubiera estado de acuerdo con todas las personas dentro de Apple que también querían otorgar licencia del software de Macintosh a todos, Apple tal vez se hubiera enriquecido inmensamente por las cuotas de la licencia y Windows no existiría en estos momentos.

Una GUI tiene cuatro elementos esenciales, denotados por los caracteres WIMP. Las letras representan ventanas (Windows), iconos (Icons), menús (Menus) y dispositivo señalador (Pointing device), respectivamente. Las ventanas son áreas rectangulares en la pantalla que se utilizan para ejecutar programas. Los iconos son pequeños símbolos en los que se puede hacer clic para que ocurra una acción. Los menús son listas de acciones, de las que se puede elegir una. Por último, un dispositivo señalador es un ratón, trackball u otro dispositivo de hardware utilizado para desplazar un cursor alrededor de la pantalla para seleccionar elementos.

El software de GUI se puede implementar en código a nivel de usuario, como en los sistemas UNIX, o en el mismo sistema operativo, como en el caso de Windows.

La entrada para los sistemas GUI sigue utilizando el teclado y el ratón, pero la salida casi siempre va a un hardware especial, conocido como **adaptador de gráficos**. Un adaptador de gráficos contiene una memoria especial conocida como **RAM de video** que contiene las imágenes que aparecen en la pantalla. Los adaptadores de gráficos de alto rendimiento a menudo tienen CPUs poderosas de 32 o 64 bits, y hasta 1 GB de su propia RAM, separada de la memoria principal de la computadora.

Cada adaptador de gráficos produce ciertos tamaños de pantalla. Los tamaños comunes son 1024×768, 1280×960, 1600×1200 y 1920×1200. Todos estos (excepto el de 1920×1200) se encuentran en proporción de 4:3, que se adapta a la proporción de aspecto de los televisores NTSC y PAL, y por ende proporciona pixeles cuadrados en los mismos monitores que se utilizan para los televisores. El tamaño de 1920×1200 está destinado para los monitores de pantalla amplia, cuya proporción de aspecto coincide con esta resolución. En la resolución más alta, una pantalla de colores con 24 bits por pixel requiere aproximadamente 6.5 MB de RAM sólo para contener la imagen, por lo que con 256 o más, el adaptador de gráficos puede contener muchas imágenes a la vez. Si la pantalla completa se actualiza 75 veces por segundo, la RAM de video debe ser capaz de transferir datos en forma continua a 489 MB/seg.

El software de salida para las GUIs es un tema masivo. Se han escrito muchos libros de 1500 páginas tan sólo acerca de la Windows GUI (por ejemplo, Petzold, 1999; Simon, 1997; y Rector y Newcomer, 1997). Es evidente que en esta sección sólo podemos rasguñar la superficie y presentar unos cuantos de los conceptos subyacentes. Para que el análisis sea concreto, describiremos la API Win32, que es proporcionada por todas las versiones de 32 bits de Windows. El software de salida para las otras GUIs apenas si es comparable en un sentido general, pero los detalles son muy distintos.

El elemento básico en la pantalla es un área rectangular llamada **ventana**. La posición y el tamaño de una ventana se determinan en forma única al proporcionar las coordenadas (en pixeles) de dos esquinas diagonalmente opuestas. Una ventana puede contener una barra de título, una barra de menús, una barra de desplazamiento vertical y una barra de desplazamiento horizontal. En la figura 5-39 se muestra una ventana ordinaria. Observe que el sistema de coordenadas de Windows posiciona el origen en la esquina superior izquierda, en donde *y* se incrementa hacia abajo, lo cual es distinto de las coordenadas cartesianas que se utilizan en matemáticas.

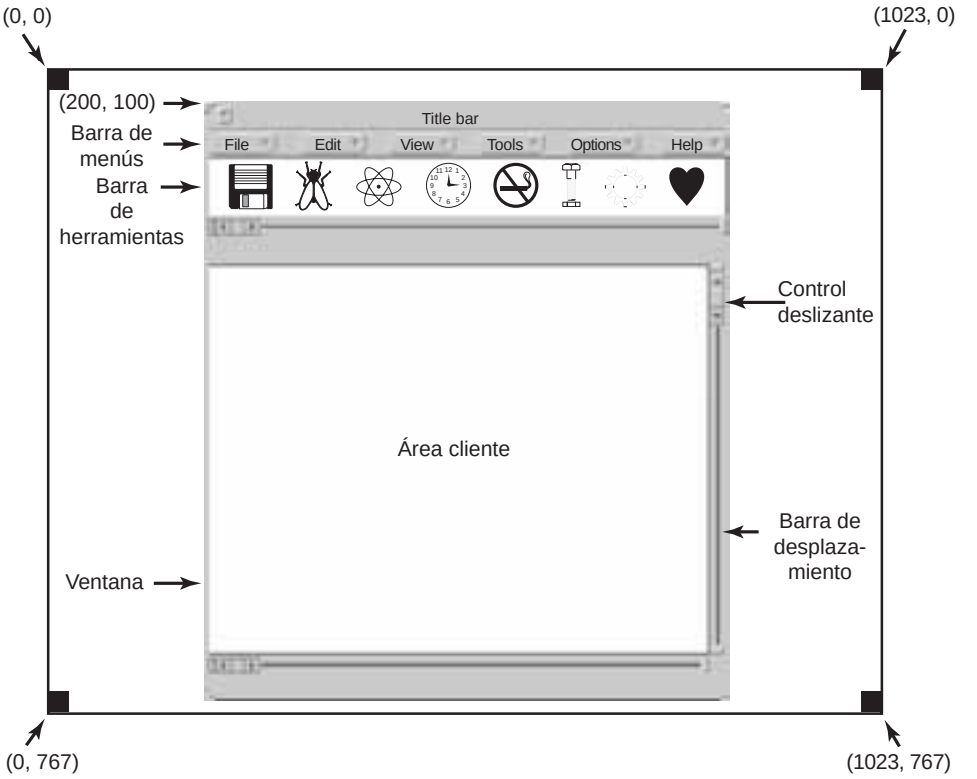

Figura 5-39. Una ventana de ejemplo ubicada en (200, 100), en una pantalla XGA.

Cuando se crea una ventana, los parámetros especifican si el usuario puede moverla, cambiar su tamaño o desplazarse en su interior (arrastrando el control deslizante en la barra de desplazamiento). La ventana principal que producen la mayoría de los programas se puede mover, cambiar su tamaño y desplazar por su interior, lo cual tiene enormes consecuencias en cuanto a la forma en que se escriben los programas de Windows. En especial, los programas deben estar informados acerca de los cambios en el tamaño de sus ventanas, y deben estar preparados para redibujar el contenido de sus ventanas en cualquier momento, incluso cuando menos lo esperan.

Como consecuencia, los programas de Windows están orientados a los mensajes. Las acciones de los usuarios que involucran al teclado o al ratón son capturadas por Windows y se convierten en mensajes para el programa propietario de la ventana que se está utilizando. Cada programa tiene una cola de mensajes a donde se envían los mensajes relacionados con todas sus ventanas. El ciclo principal del programa consiste en extraer el siguiente mensaje y procesarlo mediante la llamada a un procedimiento interno para ese tipo de mensaje. En algunos casos el mismo Windows puede llamar a estos procedimientos directamente, evadiendo la cola de mensajes. Este modelo es bastante distinto al modelo de UNIX de código por procedimientos que realiza las llamadas al sistema para interactuar con el sistema operativo. Sin embargo, X es orientado a eventos.

Para que este modelo de programación sea más claro, considere el ejemplo de la figura 5-40. Aquí podemos ver el esqueleto de un programa principal para Windows. No está completo y no realiza comprobación de errores, pero muestra el suficiente detalle para nuestros fines. Empieza por incluir un archivo de encabezado llamado *windows.h*, el cual contiene muchas macros, tipos de datos, constantes, prototipos de funciones y demás información necesaria para los programas de Windows.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE, hprev, char *szCmd int iCmdShow)
{
      WNDCLASS wndclass;                    /* objeto de clase para esta ventana */
      MSG msg;                              /* los mensajes entrantes se almacenan aquí */
      HWND hwnd;                            /* manejador (apuntador) para el objeto ventana */

      /* inicializa wndclass */
      wndclass.lpfnWndProc = WndProc;       /* indica a cuál procedimiento llamar */
      wndclass.lpszClassName = "Nombre del programa";   /* Texto para la barra de título */
      wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);  /* carga el icono del programa */
      wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);    /* carga el cursor del ratón */

      RegisterClass(&wndclass);             /* indica a Windows acerca de wndclass */
      hwnd = CreateWindow( … )              /* asigna espacio de almacenamiento para la ventana */
      ShowWindow(hwnd, iCmdShow);           /* muestra la ventana en la pantalla */
      UpdateWindow(hwnd);                   /* indica a la ventana que se pinte a sí misma */

      while (GetMessage(&msg, NULL, 0, 0)) {    /* obtiene mensaje de la cola */
            TranslateMessage(&msg);         /* traduce el mensaje */
            DispatchMessage(&msg);          /* envía el mensaje al procedimiento apropiado */
      }
      return(msg.wParam);
}

long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
  /* Aquí van las declaraciones */

      switch (mensaje) {
            case WM_CREATE:     …; return …;    /* crea la ventana */
            case WM_PAINT:      …; return …;    /* vuelve a pintar contenido de ventana */
            case WM_DESTROY:    …; return …;    /* destruye la ventana */
      }
      return(DefWindowProc(hwnd, message, wParam, lParam));       /* predeterminado */
}
```

Figura 5-40. Un esqueleto de un programa principal de Windows.

El programa principal empieza con una declaración en la que proporciona su nombre y parámetros. La macro *WINAPI* es una instrucción para que el compilador utilice cierta convención de paso de parámetros, y no es relevante para nosotros. El primer parámetro (*h*) es un manejador de instancia y se utiliza para identificar el programa para el resto del sistema. En cierto grado, Win32 está orientado a objetos, lo cual significa que el sistema contiene objetos (por ejemplo, programas, archivos y ventanas) que tienen cierto estado y un código asociado, conocidos como **métodos**, que operan sobre ese estado. Para hacer referencia a los objetos se utilizan manejadores, y en este caso, *h* identifica el programa. El segundo parámetro está presente sólo por razones de compatibilidad retroactiva. Ya no se utiliza. El tercer parámetro (*szCmd*) es una cadena con terminación cero que contiene la línea de comandos que inició el programa, incluso aunque no se haya iniciado desde una línea de comandos. El cuarto parámetro (*iCmdShow*) indica si la ventana inicial del programa debe ocupar toda la pantalla, parte de la misma o ninguna área de la pantalla (sólo en la barra de tareas).

Esta declaración ilustra una convención ampliamente utilizada por Microsoft, conocida como **notación húngara**. El nombre es un juego de palabras sobre la notación polaca, el sistema postfijo inventado por el lógico polaco J. Lukasiewicz para representa fórmulas algebraicas sin utilizar precedencia o paréntesis. La notación húngara fue inventada por un programador húngaro de Microsoft llamado Charles Simonyi, y utiliza los primeros caracteres de un identificador para especificar el tipo. Las letras y tipos permitidos incluyen c (carácter), w (palabra, que ahora significa un entero de 16 bits sin signo), i (entero con signo de 32 bits), l (long, también un entero con signo de 32 bits), s (cadena), sz (cadena terminada con un byte cero), p (apuntador), fn (función) y h (manejador). Así, *szCmd* es una cadena con terminación cero e *iCmdShow* es un entero, por ejemplo. Muchos programadores creen que codificar el tipo en nombres de variables de esta forma tiene poco valor, y hace que el código de Windows sea excepcionalmente difícil de leer. No hay nada análogo a esta convención presente en UNIX.

Cada ventana debe tener un objeto de clase asociado que defina sus propiedades. En la figura 5-40, ese objeto de clase es *wndclass*. Un objeto de tipo *WNDCLASS* tiene 10 campos, cuatro de los cuales se inicializan en la figura 5.40. En un programa real, los otros seis también se inicializarían. El campo más importante es *lpfnWndProc*, que es un apuntador largo (es decir, de 32 bits) a la función que maneja los mensajes dirigidos a esta ventana. Los demás campos que se inicializan aquí indican el nombre e icono a usar en la barra de título, y qué símbolo se debe usar para el cursor del ratón.

Una vez inicializado *wndclass*, se hace una llamada a *RegisterClass* para pasarlo a Windows. En especial, después de esta llamada Windows sabe a qué procedimiento debe llamar cuando ocurran varios eventos que no pasen a través de la cola de mensajes. La siguiente llamada, *CreateWindow*, asigna memoria para la estructura de datos de la ventana y devuelve un manejador para hacer referencia a ella posteriormente. Después el programa realiza dos llamadas más en fila, para colocar el contorno de la ventana en la pantalla, y por último se rellena por completo.

En este punto llegamos al ciclo principal del programa, que consiste en obtener un mensaje, realizarle ciertas traducciones y después pasarlo de vuelta a Windows, para que invoque a *WndProc* y lo procese. Para responder a la pregunta de si todo este mecanismo se hubiera podido simplificar, la respuesta es sí, pero se hizo de esta forma por razones históricas, (y ahora lo hacemos por costumbre).

Después del programa principal está el procedimiento **WndProc**, que maneja los diversos mensajes que se pueden enviar a la ventana. El uso de *CALLBACK* aquí, al igual que *WINAPI* anteriormente, especifica la secuencia de llamada a usar para los parámetros. El primer parámetro es el manejador de la ventana a utilizar. El segundo parámetro es el tipo de mensaje. Los parámetros tercero y cuarto se pueden utilizar para proveer información adicional cuando sea necesario.

Los tipos de mensajes *WM_CREATE y WM_DESTROY* se envían al inicio y al final del programa, respectivamente. Por ejemplo, dan al programa la oportunidad de asignar memoria para las estructuras de datos y después devolverla.

El tercer tipo de mensaje, *WM_PAINT*, es una instrucción para que el programa rellene la ventana. No sólo se llama cuando se dibuja la ventana por primera vez, sino que también se llama con frecuencia durante la ejecución del programa. En contraste a los sistemas basados en texto, en Windows un programa no puede asumir que lo que dibuje en la pantalla permanecerá ahí hasta que lo quite. Se pueden arrastrar otras ventanas encima de ésta, pueden desplegarse menús sobre ella, puede haber cuadros de diálogo y cuadros de información sobre herramientas cubriendo parte de ella, y así en lo sucesivo. Cuando se eliminan estos elementos, la ventana se tiene que volver a dibujar. La forma en que Windows indica a un programa que debe volver a dibujar una ventana es enviándole un mensaje *WM_PAINT*. Como gesto amigable, también proporciona información acerca de qué parte de la ventana se ha sobrescrito, en caso de que sea más fácil regenerar esa parte de la ventana en vez de volver a dibujarla toda.

Hay dos formas en que Windows puede hacer que un programa realice algo. Una de ellas es publicar un mensaje en su cola de mensajes. Este método se utiliza para la entrada del teclado, del ratón y los temporizadores que han expirado. La otra forma, enviar un mensaje a la ventana, implica que Windows llame directamente a *WndProc*. Este método se utiliza para todos los demás eventos. Como a Windows se le notifica cuando un mensaje se ha procesado por completo, puede abstenerse de realizar una nueva llamada hasta que termine la anterior. De esta manera se evitan las condiciones de competencia.

Hay muchos tipos de mensajes más. Para evitar un comportamiento errático en caso de que llegue un mensaje inesperado, el programa debe llamar a *DefWindowProc* al final de *WndProc* para dejar que el manejador predeterminado se encargue de los demás casos.

En resumen, un programa de Windows por lo general crea una o más ventanas con un objeto de clase para cada una. En última instancia, el comportamiento del programa está controlado por los eventos entrantes, que se procesan mediante los procedimientos del manejador. Éste es un modelo muy diferente del mundo que la visión más orientada a procedimientos de UNIX.

La acción de dibujar en la pantalla se maneja mediante un paquete que consiste en cientos de procedimientos, que en conjunto forman la **GDI** (*Graphics Device Interface*, Interfaz de dispositivo gráfico). Puede manejar texto y todo tipo de gráficos, y está diseñada para ser independiente de la plataforma y del dispositivo. Antes de que un programa pueda dibujar (es decir, pintar) en una ventana, necesita adquirir un **contexto de dispositivo**, que es una estructura de datos interna que contiene propiedades de la ventana, como el tipo de letra actual, color de texto, color de fondo, etcétera. La mayoría de las llamadas a la GDI utilizan el contexto de dispositivo, ya sea para dibujar o para obtener o establecer las propiedades.

Existen varias formas de adquirir el contexto de dispositivo. Un ejemplo simple de esta adquisición y uso es:

```
hdc = GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

La primera instrucción obtiene un manejador para un contexto de dispositivo, *hdc*. La segunda utiliza el contexto de dispositivo para escribir una línea de texto en la pantalla, especificar las coordenadas (*x, y*) de la posición en la que inicia la cadena, un apuntador a la misma cadena y su longitud. La tercera llamada libera el contexto del dispositivo para indicar que el programa ha terminado de dibujar por el momento. Observe que *hdc* se utiliza de una manera análoga a un descriptor de archivo de UNIX. Observe además que *ReleaseDC* contiene información redundante (el uso de *hdc* especifica una ventana en forma única). El uso de información redundante que no tiene valor actual es común en Windows.

Otra observación interesante es que cuando *hdc* se adquiere de esta forma, el programa sólo puede escribir en el área cliente de la ventana, no en la barra de título ni en otras partes de ella. Cualquier dibujo fuera de la región de recorte se ignora. Sin embargo, hay otra forma de adquirir un contexto de dispositivo, *GetWindowDC*, que establece la región de recorte como toda la ventana. Otras llamadas restringen la versión de recorte de otras formas. Tener varias llamadas que hacen casi lo mismo es algo característico de Windows.

Un análisis detallado de la GDI excede los alcances de este libro. Para el lector interesado, las referencias antes citadas proveen información adicional. Sin embargo, vale la pena mencionar unas cuantas palabras acerca de la GDI, dada su importancia. GDI tiene varias llamadas a procedimientos para obtener y liberar contextos de dispositivos, obtener información acerca de los contextos de dispositivos, obtener y establecer los atributos del contexto de dispositivo (por ejemplo, el color de fondo), manipular objetos de la GDI como pluma, brochas y tipos de letra, cada una de las cuales tiene sus propios atributos. Por último, desde luego que hay un gran número de llamadas de la GDI para dibujar en la pantalla.

Los procedimientos de dibujo se dividen en cuatro categorías: dibujo de líneas y curvas, dibujo de áreas rellenas, administración de mapas de bits y visualización de texto. Anteriormente vimos un ejemplo de cómo dibujar texto, por lo que ahora daremos un vistazo rápido a uno de los otros. La llamada

```
Rectangle(hdc, xizq, ysup, xder, yinf);
```

dibuja un rectángulo relleno cuyas esquinas son (*xizq*, *ysup*) y (*xder*, *yinf*). Por ejemplo,

```
Rectangle(hdc, 2, 1, 6, 4);
```

dibujará el rectángulo que se muestra en la figura 5-41. La anchura y color de línea, y el color de relleno se toman del contexto de dispositivo. Otras llamadas de la GDI son similares.

Mapas de bits

Los procedimientos de la GDI son ejemplos de gráficos vectoriales. Se utilizan para colocar figuras geométricas y texto en la pantalla. Se pueden escalar con facilidad a pantallas más grandes o pequeñas (siempre y cuando el número de pixeles en la pantalla sea el mismo). También son

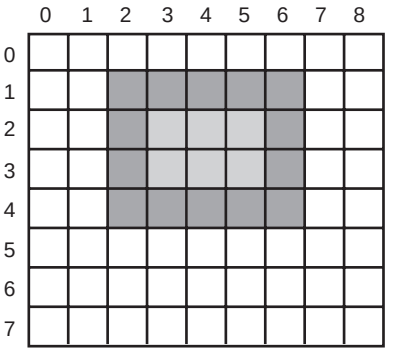

Figura 5-41. Un rectángulo de ejemplo, dibujado mediante el uso de *Rectangle*. Cada cuadro representa un pixel.

relativamente independientes del dispositivo. Una colección de llamadas a procedimientos de la GDI se puede ensamblar en un archivo que describa un dibujo completo. A dicho archivo se le conoce como **metarchivo** de Windows, y es ampliamente utilizado para transmitir dibujos de un programa de Windows a otro. Dichos archivos tienen la extensión *.wmf*.

Muchos programas de Windows permiten al usuario copiar (parte de) un dibujo y colocarlo en el portapapeles de Windows. Después, el usuario puede ir a otro programa y pegar el contenido del portapapeles en otro documento. Una manera de hacer esto es que el primer programa represente el dibujo como un metarchivo de Windows y lo coloque en el portapapeles en el formato *.wmf*. También existen otras formas.

No todas las imágenes que pueden manipular las computadoras se pueden generar mediante gráficos vectoriales. Por ejemplo, las fotografías y los videos no utilizan gráficos vectoriales. En vez de ello, estos elementos se exploran al sobreponer una rejilla en la imagen. Los valores rojo, verde y azul promedio de cada cuadro de la rejilla se muestrean y se guardan como el valor de un pixel. A dicho archivo se le conoce como **mapa de bits**. Hay muchas herramientas en Windows para manipular mapas de bits.

Otro uso para los mapas de bits es el texto. Una forma de representar un carácter específico en cierto tipo de letra es mediante un pequeño mapa de bits. Al agregar texto a la pantalla se convierte entonces en cuestión de mover mapas de bits.

Una forma general de utilizar mapas de bits es a través de un procedimiento llamado *bitblt*. Se llama de la siguiente manera:

```
BitBlt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

En su forma más simple, copia un mapa de bits de un rectángulo en una ventana a un rectángulo en otra ventana (o la misma). Los primeros tres parámetros especifican la ventana de destino y la posición. Después vienen la anchura y la altura. Luego la ventana de origen y la posición. Observe que

cada ventana tiene su propio sistema de coordenadas, con (0, 0) en la esquina superior izquierda de la ventana. Describiremos el último parámetro a continuación. El efecto de

```
BitBlt(hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);
```

se muestra en la figura 5-42. Observe con cuidado que se ha copiado toda el área de 5 x 7 de la letra A, incluyendo el color de fondo.

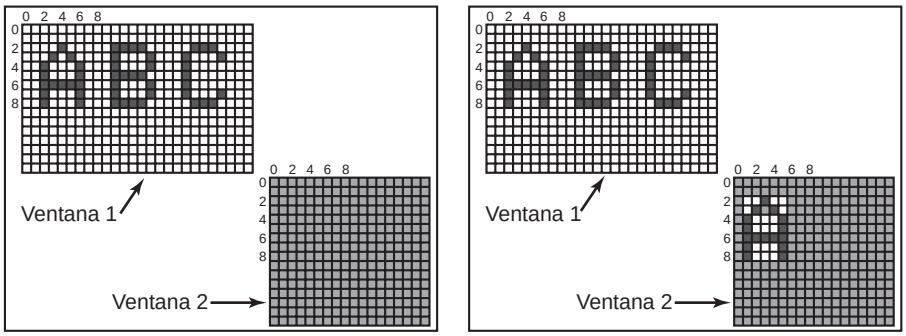

Figura 5-42. Copiado de mapas de bits usando *BitBlt*. (a) Antes. (b) Después.

BitBlt puede hacer más que sólo copiar mapas de bits. El último parámetro nos da la posibilidad de realizar operaciones booleanas para combinar el mapa de bits de origen y el mapa de bits de destino. Por ejemplo, se puede aplicar un OR al origen con el destino para fusionarlos. También se puede aplicar un OR EXCLUSIVO en ellos, con lo cual se mantienen las características tanto del origen como del destino.

Un problema con los mapas de bits es que no se escalan. Un carácter que está en un cuadro de 8 x 12 en una pantalla de 640 x 480 se verá razonable. No obstante, si este mapa de bits se copia a una página impresa a 1200 puntos/pulgada, que equivale a 10200 bits x 13200 bits, la anchura del carácter (8 pixeles) será de 8/1200 pulgadas, o 0.17 mm de ancho. Además, el copiado entre dispositivos con distintas propiedades de color o entre monocromo y color no funciona bien.

Por esta razón, Windows también tiene soporte para una estructura de datos llamada **DIB** (*Device Independent Bitmap*, Mapa de bits independiente del dispositivo). Los archivos que utilizan este formato usan la extensión *.bmp*. Estos archivos tienen encabezados de archivo y de información, y una tabla de color antes de los pixeles. Esta información facilita la acción de mover los mapas de bits entre dispositivos que no son similares.

Tipos de letras

En versiones de Windows anteriores a la 3.1, los caracteres se representaban como mapas de bits y se copiaban en la pantalla o en la impresora mediante *BitBlt*. El problema con eso, como acabamos de ver, es que un mapa de bits que se ve bien en la pantalla es demasiado pequeño para la impresora.

Además se necesita un mapa de bits diferente para cada carácter en cada tamaño. En otras palabras, dado el mapa de bits para A en un tipo de letra de 10 puntos, no hay manera de calcularlo para un tipo de letra de 12 puntos. Como cada carácter de cada tipo de letra podría ser necesario para tamaños que varíen entre 4 y 120 puntos, sería necesaria una cantidad enorme de mapas de bits. Todo el sistema era demasiado complejo para el texto.

La solución fue la introducción de los tipos de letra TrueType, que no son mapas de bits sino contornos de los caracteres. Cada carácter TrueType se define mediante una secuencia de puntos alrededor de su perímetro. Todos los puntos son relativos al origen (0, 0). Mediante este sistema es fácil escalar los caracteres hacia arriba o hacia abajo. Todo lo que se tiene que hacer es multiplicar cada coordenada por el mismo factor de escala. De esta forma, un carácter TrueType se puede escalar hacia arriba o hacia abajo a cualquier tamaño de punto, incluso hasta tamaños de punto fraccionados. Una vez que tengan el tamaño apropiado, los puntos se pueden conectar utilizando el reconocido algoritmo de "seguir los puntos" que se enseña en preescolar (las escuelas de preescolar modernas utilizan líneas tipo junquillo [splines] para obtener resultados más uniformes). Una vez que se ha completado el contorno, el carácter se puede rellenar. En la figura 5-43 se proporciona un ejemplo de algunos caracteres escalados a tres distintos tamaños de punto.

Figura 5-43. Algunos ejemplos de contornos de caracteres en distintos tamaños de punto.

Una vez que el carácter relleno está disponible en forma matemática, puede convertir en tramas o *rasterizarse*; es decir, convertirse en un mapa de bits a cualquier resolución deseada. Al escalar primero y convertir en tramas después, podemos estar seguros de que los caracteres mostrados en la pantalla y los que aparezcan en la impresora serán lo más parecidos posibles, con diferencias sólo en error de cuantización. Para mejorar aún más la calidad, es posible emplear la técnica de *hinting*, que ajusta

el resultado de convertir de contorno a trama. Por ejemplo, los patines de la parte superior de la letra T deben ser idénticos; dado un error de redondeo esto no es posible si no se utiliza el *hinting*. Esta técnica mejora la apariencia final.

5.7 CLIENTES DELGADOS

Con el paso de los años, el principal paradigma de la computación ha oscilado entre la computación centralizada y descentralizada. Las primeras computadoras (como ENIAC) eran de hecho computadoras personales, aunque muy extensas, ya que sólo una persona podía usarlas a la vez. Después llegaron los sistemas de tiempo compartido, en donde muchos usuarios remotos en terminales simples compartían una gran computadora central. Después llegó la era de la PC, en donde los usuarios tenían sus propias computadoras personales de nuevo.

Aunque el modelo de PC descentralizado tiene sus ventajas, también tiene algunas desventajas graves que apenas se están empezando a tomar con seriedad. Probablemente el mayor problema es que cada PC tiene un disco duro extenso y software complejo que se debe mantener. Por ejemplo, cuando sale al mercado una nueva versión del sistema operativo, hay que trabajar mucho para realizar la actualización en cada equipo por separado. En la mayoría de las empresas, los costos de mano de obra por realizar este tipo de mantenimiento de software empequeñecen a los costos actuales de hardware y software. Para los usuarios domésticos la labor es técnicamente gratuita, pero pocas personas son capaces de realizarla en forma correcta y menos personas aún disfrutan haciéndolo. Con un sistema centralizado, sólo una o unas cuantas máquinas tienen que actualizarse, y esas máquinas tienen un personal de expertos para realizar el trabajo.

Una cuestión relacionada es que los usuarios deben hacer respaldos con regularidad de sus sistemas de archivos con gigabytes de datos, pero pocos lo hacen. Cuando ocurre un desastre, casi siempre va acompañado de gemidos y estrujamiento de manos. Con un sistema centralizado se pueden realizar respaldos cada noche mediante robots de cintas automatizados.

Otra desventaja es que la compartición de recursos es más fácil con los sistemas centralizados. Un sistema con 256 usuarios remotos, cada uno con 256 MB de RAM tendrán la mayor parte de esa RAM inactiva durante la mayor parte del tiempo. Con un sistema centralizado con 64 GB de RAM, nunca pasa que algún usuario necesite temporalmente mucha RAM y no pueda obtenerla debido a que está en la PC de alguien más. Lo mismo se aplica para el espacio de disco y otros recursos.

Por último, estamos empezando a ver un cambio de la computación céntrica de PC a la computación céntrica de Web. Un área en donde este cambio está bien adelantado es el correo electrónico. La gente solía obtener su correo electrónico en su equipo en el hogar y lo leía ahí. Hoy en día, muchas personas entran en Gmail, Hotmail o Yahoo! y leen su correo ahí. El siguiente paso es que las personas inicien sesión en otros sitios Web para realizar procesamiento de palabras, crear hojas de cálculo y otras cosas que solían requerir software de PC. Incluso es posible que, en un momento dado, el único software que ejecute la gente en su PC sea un navegador Web, y tal vez ni siquiera eso.

Probablemente sea una conclusión justa decir que la mayoría de los usuarios desean una computación interactiva de alto rendimiento, pero en realidad no quieren administrar una computadora. Esto ha llevado a los investigadores a reexaminar la compartición de tiempo utilizando terminales

"tontas" (a las que ahora se conoce, en términos políticamente correctos, como **clientes delgados**) que cumplan con las expectativas de las terminales modernas. X fue un paso en esta dirección, y las terminales X dedicadas fueron populares durante un tiempo, pero perdieron popularidad porque costaban lo mismo que las PCs, podían hacer menos y de todas formas necesitaban cierto grado de mantenimiento del software. El descubrimiento del año sería un sistema de cómputo interactivo de alto rendimiento, en el que las máquinas de usuario no tuvieran software. Lo interesante es que esta meta se puede lograr. A continuación describiremos uno de estos sistemas de cliente delgado, conocido como **THINC** y desarrollado por los investigadores en la Universidad de Columbia (Baratto y colaboradores, 2005; Kim y colaboradores, 2006; Lai y Nieh, 2006).

La idea básica aquí es eliminar de la máquina cliente toda la inteligencia y el software, y utilizarla sólo como una pantalla, donde todo el cómputo (incluyendo el proceso de construir el mapa de bits a mostrar) se realice del lado servidor. El protocolo entre el cliente y el servidor sólo indica a la pantalla cómo debe actualizar la RAM de video, y nada más. Se utilizan cinco comandos en el protocolo entre los dos lados. Estos comandos se listan en la figura 5-44.

Comando	Descripción
Raw	Muestra los datos de pixeles crudos, en una ubicación dada
Copy	Copia el área del búfer de estructura a las coordenadas específicas
Sfill	Llena un área con un valor de color de pixel dado
Pfill	Llena un área con un patrón de pixeles dado
Bitmap	Llena una región utilizando una imagen de mapa de bits

Figura 5-44. Los comandos de visualización del protocolo THINC.

Ahora vamos a examinar los comandos. `Raw` se utiliza para transmitir los datos de los pixeles y hacer que se muestren directamente en la pantalla. En principio, éste es el único comando necesario. Los demás sólo son optimizaciones.

`Copy` instruye a la pantalla para que mueva datos de una parte de su RAM de video a otra parte. Es útil para desplazar la pantalla sin tener que volver a transmitir todos los datos.

`Sfill` llena una región de la pantalla con un solo valor de píxel. Muchas pantallas tienen un fondo uniforme en cierto color, y este comando se utiliza para generar primero el fondo, después del cual se pueden pintar texto, iconos y otros elementos.

`Pfill` replica un patrón sobre una región. También se utiliza para los fondos, pero algunos son un poco más complejos que un solo color, en cuyo caso este comando realiza el trabajo.

Por último, `Bitmap` también pinta una región, pero con un color de primer plano y un color de fondo. Con todo, estos son comandos muy simples, que requieren muy poco software del lado del cliente. Toda la complejidad de construir los mapas de bits que llenan la pantalla se llevan a cabo en el servidor. Para mejorar la eficiencia, se pueden agregar varios comandos en un solo paquete para transmitirlos a través de la red, del servidor al cliente.

Del lado servidor, los programas gráficos utilizan comandos de alto nivel para pintar la pantalla. Éstos son interceptados por el software THINC y se traducen en comandos que se pueden enviar al cliente. Los comandos se pueden reordenar para mejorar la eficiencia.

El artículo proporciona muchas mediciones de rendimiento que ejecutan numerosas aplicaciones comunes en servidores ubicados a distancias que varían entre 10 km y 10,000 km del cliente. En general, el rendimiento fue superior a otros sistemas de red de área amplia, incluso para el video en tiempo real. Para obtener más información, consulte los artículos.

5.8 ADMINISTRACIÓN DE ENERGÍA

La primera computadora electrónica de propósito general, ENIAC, tenía 18,000 tubos al vacío y consumía 140,000 watts de energía. Como resultado, generaba una factura de electricidad nada trivial. Después de la invención del transistor, el uso de la energía disminuyó en forma considerable y la industria de las computadoras perdió el interés en los requerimientos de energía. Sin embargo, hoy en día la administración de la energía vuelve a estar en la mira por varias razones, y el sistema operativo desempeña un papel aquí.

Vamos a empezar con las PCs de escritorio. A menudo, una PC de escritorio tiene una fuente de energía de 200 watts (que por lo general tiene una eficiencia de 85%; es decir, pierde el 15% de la energía entrante debido al calor). Si se encienden 100 millones de estas máquinas al mismo tiempo en todo el mundo, en conjunto utilizan 20,000 megawatts de electricidad. Ésta es la producción total de 20 plantas de energía nuclear de un tamaño promedio. Si se pudiera reducir a la mitad el requerimiento de energía, podríamos deshacernos de 10 plantas nucleares. Desde el punto de vista ambiental, deshacerse de 10 plantas de energía nuclear (o un número equivalente de plantas de combustible fósil) es un gran avance y bien vale la pena tratar de lograrlo.

El otro sitio donde la energía es una cuestión importante es en las computadoras operadas por batería, incluyendo las notebooks, de bolsillo y los Webpads, entre otras. El núcleo del problema es que las baterías no pueden contener suficiente carga para durar mucho tiempo, cuando mucho alcanzan unas horas. Además, a pesar de los esfuerzos de investigación masivos por parte de las empresas de baterías, computadoras y electrónica para el consumidor, el progreso está detenido. Para una industria acostumbrada a duplicar el rendimiento cada 18 meses (la ley de Moore), no tener ningún progreso parece como una violación de las leyes de la física, pero ésa es la situación actual. Como consecuencia, hacer que las computadoras utilicen menos energía de manera que las baterías existentes duren más tiempo es de alta prioridad para todos. El sistema operativo desempeña un importante papel aquí, como veremos a continuación.

En el nivel más bajo, los distribuidores de hardware están tratando de hacer que sus componentes electrónicos sean más eficientes en su uso de energía. Las técnicas utilizadas incluyen la reducción del tamaño de los transistores, el empleo de escalas de voltaje dinámicas, el uso de buses adiabáticos con poca desviación y técnicas similares. Esto está fuera del alcance de este libro, pero los lectores interesados pueden encontrar un buen estudio en un artículo por Venkatachalam y Franz (2005).

Hay dos métodos generales para reducir el consumo de energía. El primero es que el sistema operativo apague partes de la computadora (en su mayoría, dispositivos de E/S) que no estén en uso, debido a que un dispositivo que está apagado utiliza menos energía (o nada). El segundo método es que el programa de aplicación utilice menos energía, lo que posiblemente degradaría la calidad de la experiencia del usuario, para poder alargar el tiempo de la batería. Analizaremos cada uno de estos métodos en turno, pero primero hablaremos un poco acerca del diseño del hardware con respecto al uso de la energía.

5.8.1 Cuestiones de hardware

Las baterías son de dos tipos generales: desechables y recargables. Las baterías desechables (entre las más comunes se cuentan las AAA, AA y D) se pueden utilizar para operar dispositivos de bolsillo, pero no tienen suficiente energía como para operar computadoras notebook con grandes pantallas brillantes. Por el contrario, una batería recargable puede almacenar suficiente energía como para operar una notebook durante unas cuantas horas. Las baterías de níquel-cadmio solían dominar esta área, pero cedieron el paso a las baterías híbridas de níquel-metal, que duran más tiempo y no contaminan tanto el ambiente cuando se desechan. Las baterías de ion-litio son aún mejores, y se pueden recargar sin tener que drenarse por completo primero, pero su capacidad también es muy limitada.

El método general que la mayoría de los distribuidores de computadoras utilizan para conservar las baterías es diseñar la CPU, la memoria y los dispositivos de E/S para que tenga múltiples estados: encendido, inactivo, hibernando y apagado. Para utilizar el dispositivo, debe estar encendido. Cuando el dispositivo no se va a utilizar durante un tiempo corto se puede poner en inactividad, lo cual reduce el consumo de energía. Cuando no se va a utilizar durante un intervalo mayor, se puede poner en hibernación, lo cual reduce el consumo de energía aún más. La concesión aquí es que para sacar a un dispositivo de hibernación a menudo se requiere más tiempo y energía que para sacarlo del estado inactivo. Por último, cuando un dispositivo se apaga, no hace nada y no consume energía. No todos los dispositivos tienen estos estados, pero cuando los tienen es responsabilidad del sistema operativo administrar las transiciones de estado en los momentos apropiados.

Algunas computadoras tienen dos o incluso tres botones de energía. Uno de estos puede poner a toda la computadora en estado inactivo, del que se puede despertar rápidamente al teclear un carácter o mover el ratón; otro botón puede poner a la computadora en estado de hibernación, del cual para despertarse requiere más tiempo. En ambos casos, estos botones por lo general no hacen nada más que enviar una señal al sistema operativo, que se encarga del resto en el software. En algunos países los dispositivos eléctricos deben, por ley, tener un interruptor de energía mecánico que interrumpa un circuito y retire la energía del dispositivo, por razones de seguridad. Para cumplir con esta ley, tal vez sea necesario otro interruptor.

La administración de la energía hace que surjan varias preguntas con las que el sistema operativo debe lidiar. Muchas de ellas se relacionan con la hibernación de los recursos (apagar los dispositivos en forma selectiva y temporal, o al menos reducir su consumo de energía mientras están inactivos). Las preguntas que se deben responder incluyen las siguientes: ¿Qué dispositivos se pueden controlar? ¿Están encendidos/apagados, o tienen estados intermedios? ¿Cuánta energía se ahorra en los estados de bajo consumo de energía? ¿Se gasta energía al reiniciar el dispositivo? ¿Debe guardarse algún contexto cuando se pasa a un estado de bajo consumo de energía? ¿Cuánto se requiere para regresar al estado de energía máxima? Desde luego que las respuestas a estas preguntas varían de dispositivo en dispositivo, por lo que el sistema operativo debe ser capaz de lidiar con un rango de posibilidades.

Varios investigadores han examinado las computadoras notebook para ver a dónde va la energía. Li y colaboradores (1994) midieron varias cargas de trabajo y llegaron a la conclusión que se muestra en la figura 5-45. Lorch y Smith (1998) hicieron mediciones en otras máquinas y llegaron a las conclusiones que se muestran en la figura 5-45. Weiser y colaboradores (1994) también hicie-

ron mediciones, pero no publicaron los valores numéricos. Simplemente indicaron que los principales tres consumidores de energía eran la pantalla, el disco duro y la CPU, en ese orden. Aunque estos números no están muy cerca unos de otros, posiblemente debido a las distintas marcas de computadoras que se midieron sin duda tienen distintos requerimientos de energía, se ve claro que la pantalla, el disco duro y la CPU son objetivos obvios para ahorrar energía.

Dispositivo	Li y colaboradores (1994)	Lorch y Smith (1998)
Pantalla	68%	39%
CPU	12%	18%
Disco duro	20%	12%
Módem		6%
Sonido		2%
Memoria	0.5%	1%
Otros		22%

Figura 5-45. Consumo de energía de varias partes de una computadora notebook.

5.8.2 Cuestiones del sistema operativo

El sistema operativo desempeña un papel clave en la administración de la energía. Controla todos los dispositivos, por lo que debe decidir cuál apagar y cuándo hacerlo. Si apaga un dispositivo y éste se llega a necesitar rápidamente, puede haber un molesto retraso mientras se reinicia. Por otra parte, si espera demasiado para apagar un dispositivo, la energía se desperdicia.

El truco es buscar algoritmos y heurística que permitan al sistema operativo tomar buenas decisiones sobre lo que se va a apagar y cuándo se debe hacer. El problema es que "buenas" es muy subjetivo. Un usuario puede encontrar aceptable que después de 30 segundos de no utilizar la computadora, se requieran 2 segundos para que ésta responda a una pulsación de teclas. A otro usuario le puede parecer muy molesto bajo las mismas condiciones. En la ausencia de entrada de audio, la computadora no puede distinguir entre estos dos usuarios.

La pantalla

Ahora veamos los dispositivos que gastan la mayor parte de la energía, para ver qué se puede hacer al respecto. El principal elemento consumidor de energía es la pantalla. Para obtener una imagen nítida y brillante, la pantalla debe tener luz posterior, y eso requiere una energía considerable. Muchos sistemas operativos tratan de ahorrar energía aquí al apagar la pantalla cuando no hay actividad durante cierto número de minutos. A menudo, el usuario puede decidir cuál va a ser el intervalo de desconexión, con lo cual se deja al usuario la elección entre tener la pantalla en blanco frecuentemente y agotar la batería rápidamente (y tal vez el usuario no desee tener que hacer esta

elección). Apagar la pantalla es un estado de inactividad, debido a que se puede regenerar (a partir de la RAM de video) casi en forma instantánea, cuando se oprime una tecla o se mueve el dispositivo señalador.

Flinn y Satyanarayanan (2004) propusieron una posible mejora. Sugirieron hacer que la pantalla consista en cierto número de zonas que se pueden apagar o encender de manera independiente. En la figura 5-46 se ilustran 16 zonas, utilizando líneas punteadas para separarlas. Cuando el cursor se encuentra en la ventana 2, como se muestra en la figura 5-46(a), sólo se tienen que encender las cuatro zonas en la esquina inferior derecha. Las otras 12 pueden estar apagadas, con lo cual se ahorran 3/4 de la energía de la pantalla.

Cuando el usuario mueve el cursor a la ventana 1, se pueden apagar las zonas para la ventana 2 y las zonas detrás de la ventana 1 se pueden encender. Sin embargo, como la ventana 1 se extiende por 9 zonas, se necesita más energía. Si el administrador de ventanas puede detectar lo que está ocurriendo, puede mover de manera automática la ventana 1 para que se acomode en 4 zonas, con un tipo de acción de ajuste automático a la zona, como se muestra en la figura 5-46(b). Para lograr esta reducción de 9/16 de la energía máxima a 4/16, el administrador de ventanas tiene que comprender la administración de energía, o ser capaz de aceptar instrucciones de alguna otra pieza del sistema que lo haga. Algo más sofisticado sería la habilidad de iluminar en forma parcial una ventana que no esté completamente llena (por ejemplo, una ventana que contenga líneas cortas de texto se podría mantener apagada en la parte derecha).

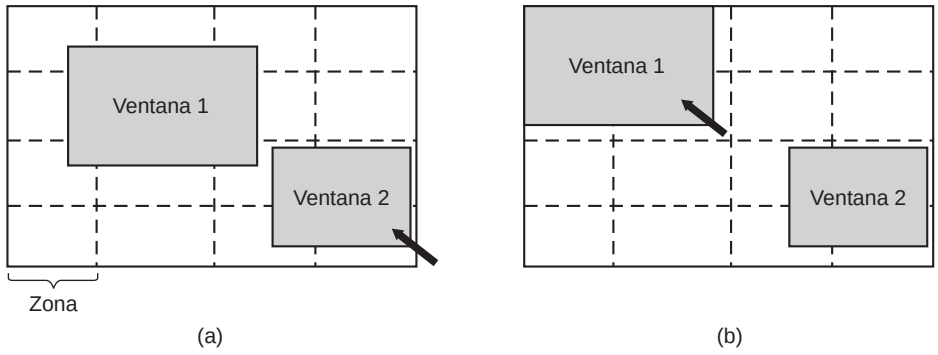

Figura 5-46. El uso de zonas para iluminar la pantalla por la parte posterior. (a) Cuando se selecciona la ventana 2 no se mueve. (b) Cuando se selecciona la ventana 1, se mueve para reducir el número de zonas iluminadas.

El disco duro

Otro de los principales villanos es el disco duro. Requiere una energía considerable para mantenerse girando a una alta velocidad, incluso aunque no haya accesos. Muchas computadoras, en especial las notebooks, hacen que el disco deje de girar después de cierto número de segundos o minutos de inactividad. Cuando se necesita otra vez, se vuelve a arrancar. Por desgracia, un disco detenido

está en hibernación en vez de inactividad, ya que se requieren unos cuantos segundos para hacer que vuelva a girar, lo cual produce retrasos considerables para el usuario.

Además, al reiniciar el disco se consume una energía adicional considerable. Como consecuencia, cada disco tiene un tiempo característico (T_d) que representa su punto muerto, a menudo en el rango de 5 a 15 segundos. Suponga que el siguiente acceso al disco se espera durante cierto tiempo t en el futuro. Si $t < T_d$, se requiere menos energía para mantener el disco girando que para apagarlo y después volver a encenderlo rápidamente. Si $t > T_d$, la energía que se ahorra hace que valga la pena apagar el disco y volver a encenderlo mucho después. Si se pudiera hacer una buena predicción (por ejemplo, con base en los patrones de acceso anteriores), el sistema operativo podría hacer buenas predicciones para apagar dispositivos y ahorrar energía. En la práctica, la mayoría de los sistemas son conservadores y sólo detienen el disco hasta después de unos cuantos minutos de inactividad.

Otra forma de ahorrar energía del disco es tener una caché de disco de un tamaño considerable en la RAM. Si se necesita un bloque que se encuentra en la caché, no hay que reiniciar un disco inactivo para satisfacer la lectura. De manera similar, si se puede colocar en el búfer de la caché una escritura en el disco, no hay que reiniciar a un disco detenido sólo para realizar la escritura. El disco puede permanecer apagado hasta que se llene la caché u ocurra un fallo en la lectura.

Otra forma de evitar arranques innecesarios del disco es que el sistema operativo mantenga informados a los programas acerca del estado del disco, enviándoles mensajes o señales. Algunos programas tienen escrituras discrecionales que se pueden omitir o retrasar. Por ejemplo, un procesador de palabras se puede configurar para que escriba el archivo que se está editando al disco cada cierto número de minutos. Si el procesador de palabras sabe que el disco está apagado en el momento en que normalmente escribiría el archivo, puede retrasar esta escritura hasta que el disco se vuelva a encender, o hasta que haya transcurrido cierto tiempo adicional.

La CPU

La CPU también se puede administrar para ahorrar energía. La CPU de una notebook se puede poner en estado inactivo mediante software, con lo cual se reduce el uso de la energía a casi cero. Lo único que puede hacer en este estado es despertarse cuando ocurra una interrupción. Por lo tanto, cada vez que la CPU está inactiva, ya sea en espera de E/S o debido a que no tiene nada que hacer, pasa al estado inactivo.

En muchas computadoras hay una relación entre el voltaje de la CPU, el ciclo de reloj y el uso de la energía. A menudo el voltaje de la CPU se puede reducir mediante software, lo cual ahorra energía pero también reduce el ciclo de reloj (aproximadamente en forma lineal). Como la energía consumida es proporcional al cuadrado del voltaje, al reducir el voltaje a la mitad la CPU se vuelve la mitad de rápida, pero a 1/4 de la energía.

Esta propiedad se puede explotar para los programas con tiempos de entrega bien definidos, como los visores de multimedia que tienen que descomprimir y mostrar un cuadro cada 40 mseg, pero se vuelven inactivos si lo hacen con más rapidez. Suponga que una CPU utiliza x joules mientras está en ejecución a toda velocidad durante 40 mseg, y $x/4$ joules cuando opera a la mitad de la velocidad. Si un visor de multimedia puede descomprimir y mostrar un cuadro en 20 mseg, el sistema operativo puede operar con la máxima energía durante 20 mseg y después apagarse por 20 mseg, para un uso total de energía de $x/2$ joules. De manera alternativa, puede operar a la mitad de la energía y sólo hacer la

entrega, pero usar sólo *x*/4 joules. En la figura 5-47 se muestra una comparación entre la operación a máxima velocidad y máxima energía durante cierto intervalo de tiempo, y la operación a la mitad de la velocidad y un cuarto de la energía por un intervalo del doble de tiempo. En ambos casos se realiza el mismo trabajo, pero en la figura 5-47(b) sólo se consume la mitad de la energía al realizarlo.

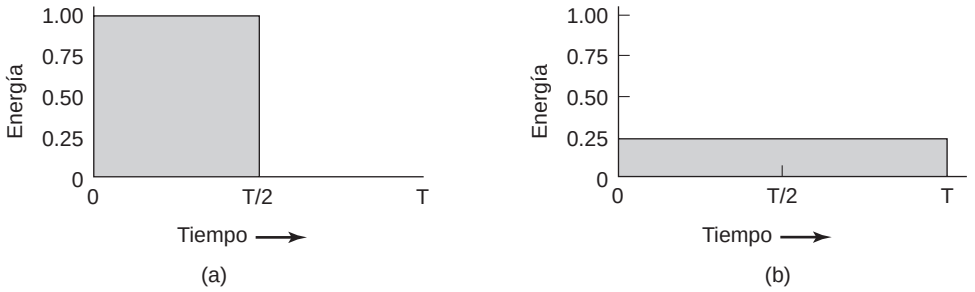

Figura 5-47. (a) Operación a máxima velocidad. (b) Recorte de voltaje por dos, de la velocidad por dos y del consumo de energía por cuatro.

De manera similar, si el usuario está escribiendo a 1 carácter/segundo, pero el trabajo necesario para procesar el carácter requiere 100 mseg, es mejor que el sistema operativo detecte los extensos periodos de inactividad y reduzca la velocidad de la CPU por un factor de 10. En resumen, es más eficiente operar a una menor velocidad que a una mayor velocidad.

La memoria

Existen dos opciones posibles para ahorrar energía con la memoria. En primer lugar, la caché se puede vaciar y después apagarse. Siempre se puede volver a cargar de la memoria principal sin pérdida de información. La recarga se puede realizar en forma dinámica y rápida, por lo que apagar la caché es por completo un estado de inactividad.

Una opción más drástica es escribir el contenido de la memoria principal en el disco y después apagar la memoria principal en sí. Este método es la hibernación, ya que se puede cortar casi toda la energía a la memoria a expensas de un tiempo de recarga considerable, en especial si el disco también está apagado. Cuando se apaga la memoria, la CPU tiene que apagarse también, o tiene que ejecutarse de la ROM. Si la CPU está apagada, la interrupción que la despierta tiene que hacer que salte al código en la ROM, de manera que la memoria se pueda recargar antes de utilizarla. A pesar de toda la sobrecarga, desconectar la memoria durante largos periodos (horas) puede valer la pena si es más conveniente reiniciar en unos cuantos segundos que tener que reiniciar el sistema operativo del disco, lo cual a menudo requiere de un minuto o más.

Comunicación inalámbrica

Cada vez hay más computadoras portátiles con una conexión inalámbrica al mundo exterior (por ejemplo, Internet). El transmisor y receptor de radio requeridos son a menudo grandes consumidores de energía. En especial, si el receptor de radio siempre está encendido para escuchar el correo

electrónico entrante, la batería se puede agotar con mucha rapidez. Por otra parte, si el radio se apaga después de, por ejemplo, 1 minuto de estar inactivo, se podrán perder los mensajes entrantes, lo cual sin duda es indeseable.

Kravets y Krishnan (1998) han propuesto una solución eficiente para este problema. El núcleo de su solución explota el hecho de que las computadoras móviles se comunican con estaciones de base fija que tienen grandes memorias y discos, sin restricciones de energía. Lo que proponen es hacer que la computadora móvil envíe un mensaje a la estación base cuando esté a punto de desconectar el radio. De ahí en adelante, la estación de radio coloca en un búfer los mensajes entrantes en su disco. Cuando la computadora móvil vuelve a encender el radio, se lo hace saber a la estación base. En ese momento se le puede enviar cualquier mensaje acumulado.

Los mensajes salientes que se generan cuando el radio está apagado se colocan en un búfer en la computadora móvil. Si el búfer amenaza con llenarse, el radio se enciende y la cola se transmite a la estación base.

¿Cuándo debe apagarse el radio? Una posibilidad es dejar que el usuario o el programa de aplicación lo decidan. Otra es apagarlo después de varios segundos de inactividad. ¿Cuándo debe volver a encenderse? De nuevo, el usuario o programa podría decidir, o se podría encender en forma periódica para comprobar el tráfico entrante y transmitir los mensajes en cola. Desde luego que también debería encenderse cuando el búfer de salida esté a punto de llenarse. También son posibles otras heurísticas.

Administración térmica

Una cuestión algo distinta, pero que también está relacionada con la energía, es la administración térmica. Las CPUs modernas se calientan en extremo debido a su alta velocidad. Los equipos de escritorio por lo general tienen un ventilador eléctrico interno para sacar el aire caliente del chasis. Como la reducción del consumo de energía comúnmente no es una cuestión preponderante con los equipos de escritorio, normalmente el ventilador está encendido todo el tiempo.

Con las notebooks, la situación es distinta. El sistema operativo tiene que monitorear la temperatura en forma continua. Cuando se acerca a la temperatura máxima permisible, el sistema operativo tiene que tomar una decisión. Puede encender el ventilador, que hace ruido y consume energía. De manera alternativa puede reducir el consumo de energía al reducir la luz posterior de la pantalla, reducir la velocidad de la CPU, ser más agresivo y desconectar el disco, o algo similar.

Cierta entrada por parte del usuario podría ser valiosa como guía. Por ejemplo, un usuario podría especificar por adelantado que el ruido del ventilador es objetable, por lo que el sistema operativo reduciría el consumo de energía en su defecto.

Administración de baterías

En los viejos tiempos, una batería sólo proporcionaba corriente hasta que se agotaba, momento en el cual se detenía. Esto ya no es así; actualmente, las laptops utilizan baterías inteligentes que se pueden comunicar con el sistema operativo. Si se les solicita, pueden informar su máximo voltaje, el voltaje actual, la máxima carga, la carga actual, la máxima proporción de agotamiento, la proporción de

agotamiento actual y mucho más. La mayoría de las computadoras notebook tienen programas que se pueden ejecutar para consultar y mostrar todos estos parámetros. A las baterías inteligentes también se les puede instruir para que cambien varios parámetros operacionales bajo el control del sistema operativo.

Algunas notebooks tienen varias baterías. Cuando el sistema operativo detecta que una batería está a punto de agotarse por completo, tiene que arreglar un cambio a la siguiente batería, sin que se corte la energía durante la transición. Cuando la última batería está a punto de agotarse, depende del sistema operativo advertir al usuario y después realizar un apagado ordenado; por ejemplo, asegurándose que el sistema de archivos no se corrompa.

Interfaz de drivers

El sistema Windows tiene un mecanismo elaborado para realizar la administración de energía, conocido como **ACPI** (*Advanced Configuration and Power Interface,* Interfaz avanzada de configuración y energía). El sistema operativo puede enviar a cualquier controlador que cumpla con este mecanismo por medio de comandos para pedir que reporte las capacidades de sus dispositivos y sus estados actuales. Esta característica es muy importante cuando se combina con los dispositivos "plug and play", ya que justo después de iniciarse, el sistema operativo ni siquiera sabe qué dispositivos hay presentes, y mucho menos sus propiedades con respecto al consumo de energía o a su capacidad de administración de la misma.

También puede enviar comandos a los controladores para pedirles que reduzcan sus niveles de energía (con base en las capacidades que detectó antes, desde luego). También hay cierto tráfico en el sentido contrario. En especial cuando un dispositivo como un teclado o un ratón detecta actividad después de un periodo de inactividad, ésta es una señal para el sistema de que debe regresar a la operación (casi) normal.

5.8.3 Cuestiones de los programas de aplicaciones

Hasta ahora hemos analizado las formas en que el sistema operativo puede reducir el uso de energía por parte de varios tipos de dispositivos. Pero también hay otro método: indicar a los programas que utilicen menos energía, aun si esto significa proporcionar una experiencia más pobre al usuario (es mejor una experiencia más pobre que ninguna experiencia, cuando la batería se agota y las luces se apagan). Por lo general, esta información se pasa cuando la carga de la batería está por debajo de cierto valor de umbral. Después es responsabilidad de los programas decidir entre degradar el rendimiento para extender la vida de la batería, o mantener el rendimiento y arriesgarse a quedarse sin energía.

Una de las preguntas que surge aquí es acerca de cómo puede un programa degradar su rendimiento para ahorrar energía. Esta pregunta ha sido estudiada por Flinn y Satyanarayanan (2004). Ellos proporcionaron cuatro ejemplos de cómo el rendimiento degradado puede ahorrar energía. Ahora los analizaremos.

En este estudio, la información se presenta al usuario en varias formas. Cuando no hay degradación se presenta la mejor información posible. Cuando hay degradación, la fidelidad (precisión)

de la información que se presenta al usuario es peor de lo que hubiera podido ser. En breve veremos ejemplos de esto.

Para poder medir el uso de la energía, Flinn y Satyanarayanan idearon una herramienta de software llamada PowerScope. Lo que hace es proporcionar un perfil de uso de energía de un programa. Para usarlo, una computadora debe estar conectada a una fuente de energía externa a través de un multímetro digital controlado por software. Mediante el uso del multímetro, el software puede leer el número de miliamperes que provienen de la fuente de energía y así determinar la energía instantánea que está consumiendo la computadora. Lo que hace PowerScope es muestrear en forma periódica el contador del programa y el uso de energía, y escribir estos datos en un archivo. Una vez que ha terminado el programa, se analiza el archivo para obtener el uso de energía de cada procedimiento. Estas mediciones formaron la base de sus observaciones. También se utilizaron mediciones en el ahorro de energía del hardware y formaron la línea de base con la que se midió el rendimiento degradado.

El primer programa que se midió fue un reproductor de video. En modo no degradado, reproduce 30 cuadros/segundo en resolución completa y a colores. Una forma de degradación es abandonar la información de color y mostrar el video en blanco y negro. Otra forma de degradación es reducir la velocidad de los cuadros, que produce parpadeos y proporciona a la película una calidad inferior. Otra forma más de degradación es reducir el número de pixeles en ambas direcciones, ya sea reduciendo la resolución espacial o la imagen a mostrar. Las mediciones de este tipo ahorraron aproximadamente 30% de la energía.

El segundo programa fue un reconocedor de voz. Realizaba un muestreo del micrófono para construir una forma de onda la cual podía analizarse en la computadora notebook o enviarse a través de un enlace de radio para analizarla en una computadora fija. Al hacer esto se ahorra energía de la CPU, pero se utiliza energía para el radio. La degradación se logró al utilizar un vocabulario más pequeño y un modelo acústico más simple. La ganancia aquí fue de aproximadamente un 35%.

El siguiente ejemplo fue un visor de mapas que obtenía el mapa a través del enlace de radio. La degradación consistía en recortar el mapa en medidas más pequeñas, o indicar al servidor remoto que omitiera los caminos más pequeños, requiriendo así menos bits para transmitir. De nuevo, aquí se obtuvo una ganancia de 35%.

El cuarto experimento fue con la transmisión de imágenes JPEG a un navegador Web. El estándar JPEG permite varios algoritmos para intercambiar la calidad de la imagen con el tamaño del archivo. Aquí, la ganancia promedio fue de sólo 9%. Inlcuso así, en general los experimentos mostraron que al aceptar cierta degradación de la calidad, el usuario puede trabajar más tiempo en una batería dada.

5.9 INVESTIGACIÓN ACERCA DE LA E/S

Hay una cantidad considerable de investigación acerca de la entrada/salida, pero la mayoría está enfocada en dispositivos específicos, en vez de la E/S en general. A menudo, el objetivo es mejorar el rendimiento de una manera u otra.

Los sistemas de disco son uno de los casos en cuestión. Los algoritmos de programación del brazo del disco son un área de investigación siempre popular (Bachmat y Braverman, 2006; y Zarandioon y Thomasian, 2006), al igual que los arreglos de discos (Arnan y colaboradores, 2007). La

optimización de la ruta completa de E/S también es de interés (Riska y colaboradores, 2007). También hay investigación sobre la caracterización de la carga de trabajo del disco (Riska y Riedel, 2006). Una nueva área de investigación relacionada con los discos son los discos flash de alto rendimiento (Birrell y colaboradores, 2007; y Chang, 2007). Los controladores de dispositivos también están obteniendo cierta atención necesaria (Ball y colaboradores, 2006; Ganapathy y colaboradores, 2007; Padioleau y colaboradores, 2006).

Otra nueva tecnología de almacenamiento es MEMS (*Micro-Electrical-Mechanical Systems*, Sistemas micro electromecánicos), que pueden reemplazar (o al menos complementar) a los discos (Rangaswami y colaboradores, 2007; y Yu y colaboradores, 2007). Otra área de investigación prometedora es cómo hacer el mejor uso de la CPU dentro del controlador de disco; por ejemplo, para mejorar el rendimiento (Gurumurthi, 2007) o para detectar virus (Paul y colaboradores, 2005).

Algo sorprendente es que el modesto reloj sigue siendo un tema de investigación. Para proveer una buena resolución, algunos sistemas operativos operan el reloj a 1000 Hz, lo cual produce una sobrecarga considerable. La investigación se enfoca en cómo deshacerse de esta sobrecarga (Etsion y colaboradores, 2003; Tsafir y colaboradores, 2005).

Los clientes delgados también son un tema de considerable interés (Kissler y Hoyt, 2005; Ritschard, 2006; Schwartz y Gerrazzi, 2005).

Dado el gran número de científicos computacionales con computadoras notebooks y el microscópico tiempo de batería en la mayoría de ellos, no debe sorprender que haya un enorme interés en cuanto al uso de técnicas de software para reducir el consumo de energía. Entre los temas especializados que se están analizando se encuentran: escribir código de aplicaciones para maximizar los tiempos de inactividad del disco (Son y colaboradores, 2006), hacer que los discos giren a menor velocidad cuando se utilicen poco (Gurumurthi y colaboradores, 2003), utilizar modelos de programa para predecir cuándo se pueden desconectar las tarjetas inalámbricas (Hom y Kremer, 2003), ahorro de energía para VoIP (Gleeson y colaboradores, 2006), examinar el costo de energía de la seguridad (Aaraj y colaboradores, 2007), realizar programación de multimedia haciendo uso eficiente de la energía (Yuan y Nahrstedt, 2006), e incluso hacer que una cámara integrada detecte si hay alguien viendo la pantalla y desconectarla cuando nadie la esté viendo (Dalton y Ellis, 2003). En el extremo de bajo rendimiento, otro tema popular es el uso de la energía en las redes de monitoreo (Min y colaboradores, 2007; Wang y Xiao, 2006). Al otro extremo del espectro, guardar energía en grandes granjas de servidores también es de interés (Fan y colaboradores, 2007; Tolentino y colaboradores, 2007).

5.10 RESUMEN

A menudo las operaciones de entrada/salida son un tema ignorado pero importante. Una fracción considerable de cualquier sistema operativo está relacionada con las operaciones de E/S. Estas operaciones se pueden llevar a cabo en una de tres formas. En primer lugar está la E/S programada, en la que la CPU principal recibe o envía cada byte o palabra y entra a un ciclo estrecho para esperar hasta que pueda obtener o enviar el siguiente byte. En segundo lugar está la E/S controlada por interrupciones, en la que la CPU inicia una transferencia de E/S para un carácter o palabra y se pone a hacer algo más hasta que llega una interrupción indicando que se completó la operación de E/S. En tercer lugar está el DMA, en el que un chip separado administra la transferencia completa de un bloque de datos, y recibe una interrupción sólo cuando se ha transferido todo el bloque completo.

La E/S se puede estructurar en cuatro niveles: los procedimientos de servicio de interrupciones, los controladores de dispositivos, el software de E/S independiente del dispositivo, y las bibliotecas de E/S y el uso de colas que se ejecutan en espacio de usuario. Los controladores de dispositivos se encargan de los detalles de operación de los dispositivos, y proporcionan interfaces uniformes para el resto del sistema operativo. El software de E/S independiente del dispositivo realiza cosas como el uso de búfer y el reporte de errores.

Los discos vienen en una variedad de tipos, incluyendo los discos magnéticos, RAIDs y varios tipos de discos ópticos. A menudo se pueden utilizar algoritmos de planificación del brazo del disco para mejorar su rendimiento, pero la presencia de la geometría virtual complica las cosas. Al colocar dos discos para formar un par, se puede construir un medio de almacenamiento estable con ciertas propiedades útiles.

Los relojes se utilizan para llevar la cuenta de la hora real y limitan el tiempo que se pueden ejecutar los procesos, manejan temporizadores guardianes y realizan la contabilidad.

Las terminales orientadas a caracteres tienen una variedad de cuestiones relacionadas con los caracteres especiales que se pueden introducir, y las secuencias de escape especiales que se pueden imprimir. La entrada puede estar en modo crudo o cocido, dependiendo de cuánto control desee el programa sobre la entrada. Las secuencias de escape en la salida controlan el movimiento del cursor y permiten insertar y eliminar texto en la pantalla.

La mayoría de los sistemas UNIX utilizan el Sistema X Window como la base de la interfaz de usuario. Consiste en programas enlazados con bibliotecas especiales que emiten comandos de dibujo, y un servidor X que escribe en la pantalla.

Muchas computadoras personales utilizan GUIs para mostrar la información al usuario. Éstas se basan en el paradigma WIMP: ventanas, iconos, menús y un dispositivo señalador. Los programas basados en GUI por lo general son controlados por eventos, en donde los eventos de teclado, ratón y otros dispositivos se envían al programa para procesarlos tan pronto como ocurren. En los sistemas UNIX, las GUIs casi siempre se ejecutan encima de X.

Los clientes delgados tienen algunas ventajas en comparación con las PCs estándar, siendo las más notables su simplicidad y menor necesidad de mantenimiento por parte de los usuarios. Los experimentos con el cliente delgado THINC han demostrado que con cinco primitivas simples es posible construir un cliente con un buen rendimiento, incluso para el video.

Por último, la administración de la energía es una cuestión importante para las computadoras notebook, ya que los tiempos de vida de las baterías son limitados. El sistema operativo puede emplear varias técnicas para reducir el consumo de energía. Los programas también pueden ayudar al sacrificar cierta calidad por tiempos de vida más largos para las baterías.

PROBLEMAS

1. Los avances en la tecnología de los chips han hecho posible colocar todo un controlador, incluyendo la lógica de acceso al bus, en un chip económico. ¿Cómo afecta eso al modelo de la figura 1-5?

2. Dadas las velocidades listadas en la figura 5-1, ¿es posible explorar documentos desde un escáner y transmitirlos sobre una red 802.11g a la máxima velocidad? Defienda su respuesta.

3. La figura 5-3(b) muestra una forma de tener E/S por asignación de memoria, incluso en presencia de buses separados para la memoria y los dispositivos de E/S, a saber, primero se prueba el bus de memoria y si falla, se prueba el bus de E/S. Un astuto estudiante de ciencias computacionales ha ideado una mejora sobre esta idea: probar ambos en paralelo, para agilizar el proceso de acceder a los dispositivos de E/S. ¿Qué piensa usted sobre esta idea?

4. Suponga que un sistema utiliza DMA para la transferencia de datos del controlador del disco a la memoria principal. Suponga además que se requieren t_1 nseg en promedio para adquirir el bus, y t_2 nseg par transferir una palabra a través del bus ($t_1 \gg t_2$). Una vez que la CPU ha programado el controlador de DMA, ¿cuánto tiempo se requiere para transferir 1000 palabras del controlador de disco a la memoria principal, si (a) se utiliza el modo de una palabra a la vez, (b) se utiliza el modo ráfaga? Suponga que para comandar el controlador de disco se requiere adquirir el bus para enviar una palabra, y para reconocer una transferencia también hay que adquirir el bus para enviar una palabra.

5. Suponga que una computadora puede leer o escribir una palabra de memoria en 10 nseg. Suponga además que cuando ocurre una interrupción, se meten en la pila los 32 registros más el contador del programa y el PSW. ¿Cuál es el número máximo de interrupciones por segundo que puede procesar esta máquina?

6. Los arquitectos de CPUs saben que los escritores de sistemas operativos odian las interrupciones imprecisas. Una manera de complacer a los escritores de SO es que la CPU deje de emitir nuevas instrucciones cuando se señale una interrupción, pero que permita que todas las instrucciones que se están ejecutando terminen y después obligue a que se produzca la interrupción. ¿Tiene este método alguna desventaja? Explique su respuesta.

7. En la figura 5-9(b), la interrupción no se reconoce sino hasta después de que se haya enviado el siguiente carácter a la impresora. ¿Podría haberse reconocido de igual forma justo al inicio del procedimiento de servicio de interrupciones? De ser así, mencione una razón de hacerlo al final, como en el texto. Si no es así, ¿por qué no?

8. Una computadora tiene una línea de tubería de tres etapas, como se muestra en la figura 1-6(a). En cada ciclo de reloj se obtiene una nueva instrucción de la memoria en la dirección a la que apunta el PC, se coloca la nueva instrucción en la tubería y se avanza el PC. Cada instrucción ocupa exactamente una palabra de memoria. Las instrucciones que ya están en la tubería se avanzan una etapa. Cuando ocurre una interrupción, el PC actual se mete en la pila y al PC se le asigna la dirección del manejador de interrupciones. Después la tubería se desplaza una etapa a la derecha, se obtiene la primera instrucción del manejador de interrupciones y se coloca en la tubería. ¿Tiene esta máquina interrupciones precisas? Defienda su respuesta.

9. Una página ordinaria de texto contiene 50 líneas de 80 caracteres cada una. Imagine que cierta impresora puede imprimir 6 páginas por minuto, y que el tiempo para escribir un carácter en el registro de salida de la impresora es tan corto que puede ignorarse. ¿Tiene sentido operar esta impresora mediante la E/S controlada por eventos, si cada carácter impreso requiere una interrupción que tarda 50 μseg en ser atendida?

10. Explique cómo un SO puede facilitar la instalación de un nuevo dispositivo sin necesidad de volver a compilar el SO.

11. ¿En cuál de los cuatro niveles de software de E/S se realiza cada una de las siguientes acciones?

(a) Calcular la pista, sector y cabeza para una lectura de disco.

(b) Escribir comandos en los registros de dispositivo.
(c) Comprobar si el usuario tiene permiso de utilizar el dispositivo.
(d) Convertir los enteros binarios a ASCII para imprimirlos.

12. Una red de área local se utiliza de la siguiente manera. El usuario emite una llamada al sistema para escribir paquetes de datos en la red. Después el sistema operativo copia los datos en un búfer del kernel. Luego copia los datos a la tarjeta controladora de red. Cuando todos los bytes están seguros dentro del controlador, se envían a través de la red a una velocidad de 10 megabits/seg. El controlador de red receptor almacena cada bit un microsegundo después de enviarlo. Cuando llega el último bit se interrumpe la CPU de destino, y el kernel copia el paquete recién llegado a un búfer del kernel para inspeccionarlo. Una vez que averigua para cuál usuario es el paquete, el kernel copia los datos en el espacio de usuario. Si suponemos que cada interrupción y su procesamiento asociado requiere 1 mseg, que los paquetes son de 1024 bytes (ignore los encabezados) y que para copiar un byte se requiere 1 µseg, ¿cuál es la velocidad máxima a la que un proceso puede enviar los datos a otro? Suponga que el emisor se bloquea hasta que se termine el trabajo en el lado receptor y que se devuelve una señal de reconocimiento. Para simplificar, suponga que el tiempo para obtener de vuelta el reconocimiento es tan pequeño que se puede ignorar.

13. ¿Por qué los archivos de salida para la impresora normalmente se ponen en cola en el disco antes de imprimirlos?

14. El nivel 3 de RAID puede corregir errores de un solo bit, utilizando sólo una unidad de paridad. ¿Cuál es el objetivo del nivel 2 de RAID? Después de todo, también puede corregir sólo un error y requiere más unidades para hacerlo.

15. Un RAID puede fallar si dos o más unidades fallan dentro de un intervalo de tiempo corto. Suponga que la probabilidad de que una unidad falle en una hora específica es p. ¿Cuál es la probabilidad de que falle un RAID de k unidades en una hora específica?

16. Compare los niveles 0 a 5 de RAID con respecto al rendimiento de lectura, de escritura, sobrecarga de espacio y confiabilidad.

17. ¿Por qué los dispositivos de almacenamiento ópticos son intrínsecamente capaces de obtener una densidad mayor de datos que los dispositivos de almacenamiento magnéticos? *Nota*: este problema requiere cierto conocimiento de física de secundaria y acerca de cómo se generan los campos magnéticos.

18. ¿Cuáles son las ventajas y desventajas de los discos ópticos, en comparación con los discos magnéticos?

19. Si un controlador de disco escribe los bytes que recibe del disco a la memoria, tan pronto como los recibe, sin uso interno de búfer, ¿puede el entrelazado ser útil? Explique.

20. Si un disco tiene doble entrelazado, ¿necesita también desajuste de cilindros para evitar pasar por alto información al realizar una búsqueda de pista a pista? Explique su respuesta.

21. Considere un disco magnético que consiste de 16 cabezas y 400 cilindros. Este disco está dividido en zonas de 100 cilindros, en donde los cilindros en distintas zonas contienen 160, 200, 240 y 280 sectores, respectivamente. Suponga que cada sector contiene 512 bytes, que el tiempo de búsqueda promedio entre cilindros adyacentes es de 1 mseg y que el disco gira a 7200 RPM. Calcule (a) la capacidad del disco; (b) el desajuste óptimo de pistas y (c) la velocidad máxima de transferencia de datos.

22. Un fabricante de discos tiene dos discos de 5.25 pulgadas, y cada uno de ellos tiene 10,000 cilindros. El más nuevo tiene el doble de la densidad de grabación lineal del más antiguo. ¿Qué propiedades de disco son mejores en la unidad más reciente y cuáles son iguales?

23. Un fabricante de computadoras decide rediseñar la tabla de particiones del disco duro de un Pentium para proporcionar más de cuatro particiones. ¿Cuáles son algunas consecuencias de este cambio?

24. Las peticiones de disco llegan al controlador de disco para los cilindros 10, 22, 20, 2, 40, 6 y 38, en ese orden. Una búsqueda requiere 6 mseg por cada cilindro desplazado. Determine cuánto tiempo de búsqueda se requiere para

(a) Primero en llegar, primero en ser atendido.
(b) El cilindro más cercano a continuación.
(c) Algoritmo del elevador (al principio se mueve hacia arriba).

En todos los casos, el brazo está al principio en el cilindro 20.

25. Una ligera modificación del algoritmo del elevador para planificar peticiones de disco es explorar siempre en la misma dirección. ¿En qué aspecto es mejor este algoritmo modificado que el algoritmo del elevador?

26. En el análisis del almacenamiento estable mediante el uso de RAM no volátil, se pasó por alto el siguiente punto. ¿Qué ocurre si se completa la escritura estable, pero ocurre una falla antes de que el sistema operativo pueda escribir un número de bloque inválido en la RAM no volátil? ¿Arruina esta condición de competencia la abstracción del almacenamiento estable? Explique su respuesta.

27. En el análisis sobre almacenamiento estable, se demostró que el disco se puede recuperar en un estado consistente (o la escritura se completa, o no se lleva a cabo) si ocurre una falla de la CPU durante una escritura. ¿Se mantiene esta propiedad si la CPU falla de nuevo durante un procedimiento de recuperación? Explique su respuesta.

28. El manejador de interrupciones de reloj en cierta computadora requiere 2 mseg (incluyendo la sobrecarga de la conmutación de procesos) por cada pulso de reloj. El reloj opera a 60 Hz. ¿Qué fracción de la CPU está dedicada al reloj?

29. Una computadora utiliza un reloj programable en modo de onda cuadrada. Si se utiliza un reloj de 500 MHz, ¿cuál debe ser el valor del registro contenedor para lograr una resolución de reloj de

(a) un milisegundo (un pulso de reloj cada milisegundo)?
(b) 100 microsegundos?

30. Un sistema simula varios relojes al encadenar todas las peticiones de reloj en conjunto, como se muestra en la figura 5-34. Suponga que el tiempo actual es 5000 y que hay peticiones de reloj pendientes para los tiempos 5008, 5012, 5015, 5029 y 5037. Muestre los valores del Encabezado del reloj, la Hora actual, y la Siguiente señal en los tiempos 5000, 5005 y 5013. Suponga que llega una nueva señal (pendiente) en el tiempo 5017 para 5033. Muestre los valores del Encabezado del reloj, la Hora actual y la Siguiente señal en el tiempo 5023.

31. Muchas versiones de LINUX utilizan un entero de 32 bits sin signo para llevar la cuenta del tiempo como el número de segundos transcurridos desde el origen del tiempo. ¿Cuándo terminarán estos sistemas (año y mes)? ¿Espera usted que esto realmente ocurra?

32. Una terminal de mapa de bits contiene 1280 por 960 pixeles. Para desplazarse por una ventana, la CPU (o el controlador) debe desplazar todas las líneas de texto hacia arriba, al copiar sus bits de una parte de la RAM de video a otra. Si una ventana específica es de 60 líneas de altura por 80 caracteres de anchura (5280 caracteres en total), y el cuadro de un carácter es de 8 pixeles de anchura por 16 pixeles de altura, ¿cuánto tiempo se requiere para desplazar toda la ventana, a una velocidad de copia de 50 nseg por byte? Si todas las líneas tienen 80 caracteres de largo, ¿cuál es la tasa de transferencia de baudios equivalente de la terminal? Para colocar un carácter en la pantalla se requieren 5 µseg. ¿Cuántas líneas por segundo se pueden mostrar?

33. Después de recibir un carácter SUPR (SIGINT), el controlador de la pantalla descarta toda la salida que haya en la cola para esa pantalla. ¿Por qué?

34. En la pantalla a color de la IBM PC original, al escribir en la RAM de video en cualquier momento que no fuera durante el retrazado vertical del haz del CRT, aparecían puntos desagradables por toda la pantalla. Una imagen de pantalla es de 25 por 80 caracteres, cada uno de los cuales se ajusta a un cuadro de 8 pixeles por 8 pixeles. Cada fila de 640 pixeles se dibuja en una sola exploración horizontal del haz, que requiere 63.6 µseg, incluyendo el retrazado horizontal. La pantalla se vuelve a dibujar 60 veces por segundo, y en cada una de esas veces se requiere un periodo de retrazado vertical para regresar el haz a la parte superior. ¿Durante qué fracción de tiempo está disponible la RAM de video para escribir?

35. Los diseñadores de un sistema de computadora esperaban que el ratón se pudiera mover a una velocidad máxima de 20 cm/seg. Si un mickey es de 0.1 mm y cada mensaje del ratón es de 3 bytes, ¿cuál es la máxima velocidad de transferencia de datos del ratón, suponiendo que cada mickey se reporte por separado?

36. Los colores aditivos primarios son rojo, verde y azul, lo cual significa que se puede fomar cualquier color a partir de una superposición lineal de estos colores. ¿Es posible que alguien pudiera tener una fotografía a color que no se pudiera representar mediante el uso de color completo de 24 bits?

37. Una forma de colocar un carácter en una pantalla con mapa de bits es utilizar bitblt desde una tabla de tipos de letras. Suponga que un tipo de letra específico utiliza caracteres de 16 x 24 pixeles en color RGB verdadero.

(a) ¿Cuánto espacio en la tabla de tipos de letras ocupa cada carácter?
(b) Si para copiar un byte se requieren 100 nseg, incluyendo la sobrecarga, ¿cuál es la velocidad de transferencia de salida para la pantalla en caracteres/seg?

38. Suponiendo que se requieren 10 nseg para copiar un byte, ¿cuánto tiempo se requiere para retrazar por completo una pantalla con asignación de memoria, en modo de texto de 80 caracteres $\times$ 25 líneas? ¿Y una pantalla gráfica de 1024 $\times$ 768 pixeles con colores de 24 bits?

39. En la figura 5-40 hay una clase para *RegisterClass*. En el código X Window correspondiente de la figura 5-38, no hay dicha llamada o algo parecido. ¿Por qué no?

40. En el texto vimos un ejemplo sobre cómo dibujar un rectángulo en la pantalla mediante la GDI de Windows:

```
Rectangle(hdc, xizq, ysup, xder, yinf);
```

¿Hay una verdadera necesidad por el primer parámetro (*hdc*), y de ser así, cuál es? Después de todo, las coordenadas del rectángulo se especifican de manera explícita como parámetros.

41. Una terminal THINC se utiliza para mostrar una página Web que contenga una caricatura animada de 400 pixeles x 160 pixeles a una velocidad de 10 cuadros/seg. ¿Qué fracción de una conexión Fast Ethernet de 100 Mbps se consume al mostrar la caricatura?

42. Se ha observado que el sistema THINC funciona bien con una red de 1 Mbps en una prueba. ¿Puede haber problemas en una situación multiusuario? *Sugerencia:* Considere que un gran número de usuarios ven un programa de TV y que el mismo número de usuarios navegan por World Wide Web.

43. Si el máximo voltaje de una CPU (*V*) se recorta a *V/n*, su consumo de energía disminuye a $1/n^2$ de su valor original, y su velocidad de reloj disminuye a *1/n* de su valor original. Suponga que un usuario está escribiendo a 1 carácter/seg, pero que el tiempo de CPU requerido para procesar cada carácter es de 100 mseg. ¿Cuál es el valor óptimo de *n* y cuál es el correspondiente ahorro de energía en porcentaje, comparado con la opción de no cortar el voltaje? Suponga que una CPU inactiva no consume energía.

44. Una computadora notebook se configura para sacar el máximo provecho de las características de ahorro de energía, incluyendo apagar la pantalla y el disco duro después de periodos de inactividad. Algunas veces un usuario ejecuta programas UNIX en modo de texto, y otras veces utiliza el Sistema X Window. Le sorprende descubrir que la vida de la batería es mucho mejor cuando utiliza programas de sólo texto. ¿Por qué?

45. Escriba un programa que simule un almacenamiento estable. Utilice dos archivos extensos de longitud fija en su disco para simular los dos discos.

46. Escriba un programa para implementar los tres algoritmos de planificación del brazo del disco. Escriba un programa controlador que genere una secuencia de números de cilindro (0 a 999) al azar, que ejecute los tres algoritmos para esta secuencia e imprima la distancia total (número de cilindros) que necesita el brazo para desplazarse en los tres algoritmos.

47. Escriba un programa para implementar varios temporizadores usando un solo reloj. La entrada para este programa consiste en una secuencia de cuatro tipos de comandos (S <int>, T, E <int>, P): S <int> establece el tiempo actual a <int>; T es un pulso de reloj; y E <int> programa una señal para que ocurra en el tiempo <int>; P imprime los valores de Tiempo actual, Siguiente señal y Encabezado de reloj. Su programa también debe imprimir una instrucción cada vez que sea tiempo de generar una señal.