5.2.3 Operaciones con directorios

Las **llamadas al sistema** permitidas para **administrar directorios** muestran variaciones más amplias de un sistema a otro que las **llamadas para archivos**. A fin de dar una idea de la naturaleza y el modo de operar de esas llamadas, presentamos una muestra (**UNIX**).

1. **`CREATE`**. Se crea un directorio, que está vacío con la excepción de **`punto`** y **`punto punto`**, que el sistema (o, en unos pocos casos, el programa **`mkdir`**) coloca ahí automáticamente.
2. **`DELETE`**. Se elimina un directorio. Sólo puede eliminarse un directorio vacío. Un directorio que sólo contiene **`punto`** y **`punto punto`** se considera vacío, ya que éstos normalmente no pueden eliminarse.
3. **`OPENDIR`**. Los directorios pueden leerse. Por ejemplo, para listar todos los archivos de un directorio, un programa para emitir listados abre el directorio y lee los nombres de los archivos que contiene. Antes de poder leer un directorio, es preciso abrirlo, de forma análoga a como se abren y leen los archivos.
4. **`CLOSEDIR`**. Una vez que se ha leído un directorio, debe cerrarse para liberar espacio de tablas internas.
5. **`READDIR`**. Esta llamada devuelve la siguiente entrada de un directorio abierto. Antes, era posible leer directorios empleando la llamada al sistema **`READ`** normal, pero ese enfoque tiene la desventaja de obligar al programador a conocer y manejar la estructura interna de los directorios. En contraste, **`READDIR`** siempre devuelve una entrada en un formato estándar, sin importar cuál de las posibles estructuras de directorio se esté usando.
6. **`RENAME`**. En muchos sentidos, los directorios son iguales que los archivos y podemos cambiar su nombre tal como hacemos con los archivos.
7. **`LINK`**. El enlace (**linking**) es una técnica que permite a un archivo aparecer en más de un directorio. Esta llamada al sistema especifica un archivo existente y un nombre de ruta, y crea un enlace del archivo existente al nombre especificado por la ruta. De este modo, el mismo archivo puede aparecer en múltiples directorios.
8. **`UNLINK`**. Se elimina una entrada de directorio. Si el archivo que está siendo desvinculado sólo está presente en un directorio (el caso normal), se eliminará del sistema de archivos. Si el archivo está presente en varios directorios, sólo se eliminará el nombre de ruta especificado; los demás seguirán existiendo. En **UNIX**, la llamada al sistema para eliminar archivos es, de hecho, **`UNLINK`**.

La lista anterior incluye las llamadas más importantes, pero hay otras, por ejemplo, para administrar la información de protección asociada con un directorio.

5.3 Implementación de sistemas de archivos

Ya es momento de pasar de la perspectiva del usuario del sistema de archivos a la perspectiva del implementador. A los usuarios les interesa la forma de nombrar los archivos, las operaciones que pueden efectuarse con ellos, el aspecto que tiene el árbol de directorios y cuestiones de interfaz por el estilo. A los implementadores les interesa cómo están almacenados los archivos y directorios, cómo se administra el espacio en disco y cómo puede hacerse que todo funcione de forma eficiente y confiable. En las secciones siguientes examinaremos varias de estas áreas para conocer los problemas y las concesiones.

5.3.1 Implementación de archivos

Tal vez el aspecto más importante de la implementación del almacenamiento en archivos sea poder relacionar **bloques de disco** con **archivos**. Se emplean diversos métodos en los diferentes sistemas operativos. En esta sección examinaremos algunos de ellos.

Asignación contigua

El esquema de **asignación** más sencillo es almacenar **cada archivo** como un **bloque contiguo** de **datos en el disco**. Así, en un disco con bloques de **1K**, a un archivo de **50K** se le asignarían **50 bloques** consecutivos. Este esquema tiene **dos ventajas** importantes. Primera, la **implementación es sencilla** porque para saber dónde están los bloques de un archivo basta con recordar un número, la dirección en disco del primer bloque. Segunda, el **rendimiento es excelente** porque es posible leer todo el archivo del disco en una sola operación.

Desafortunadamente, la **asignación contigua** tiene también **dos desventajas** igualmente importantes. Primera, **no es factible** si no se conoce el tamaño máximo del archivo en el momento en que se crea el archivo. Sin esta información, el sistema operativo no sabrá cuánto espacio en disco debe reservar. Sin embargo, en los sistemas en los que los archivos deben escribirse de un solo golpe, el método puede usarse con gran provecho. La segunda desventaja es la **fragmentación del disco** que resulta de esta política de asignación. Se desperdicia espacio que de otra forma podría haberse aprovechado. La compactación del disco suele tener un costo prohibitivo, aunque tal vez podría efectuarse de noche cuando el sistema estaría ocioso.

Asignación por lista enlazada

El segundo método para almacenar archivos es guardar cada uno como una **lista enlazada bloques de disco**, como se muestra en la **Figura 5.8**. La primera palabra de cada bloque se emplea como apuntador al siguiente. El resto del bloque se destina a datos.

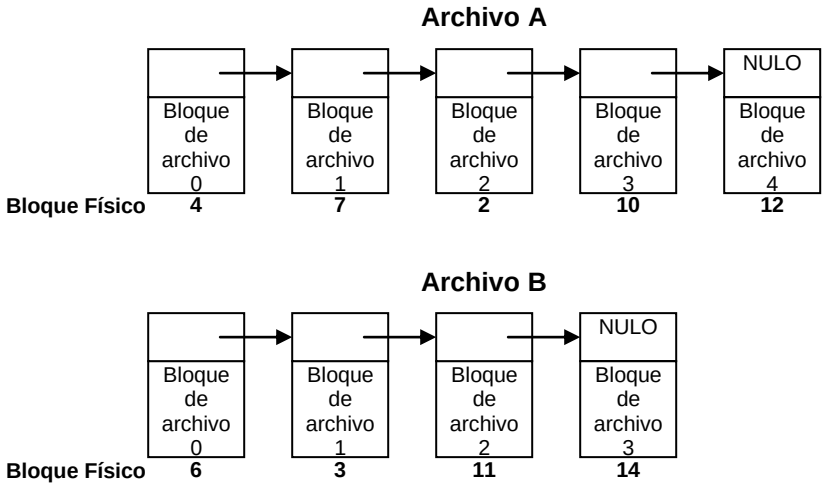

Figura 5.8: Almacenamiento de un archivo como lista enlazada de bloques de disco.

A diferencia de la asignación contigua, con este método es posible **utilizar todos los bloques**. No se pierde espacio por fragmentación del disco (excepto por fragmentación interna en el último bloque). Además, basta con que en la entrada de directorio se almacene la dirección en disco del primer bloque. El resto del archivo puede encontrarse siguiendo los enlaces.

Por otro lado, aunque la **lectura secuencial** de un archivo es sencilla, el **acceso aleatorio** es extremadamente lento. Además, la cantidad de almacenamiento de datos en un bloque ya no es una potencia de dos porque el apuntador ocupa unos cuantos bytes. Si bien tener un tamaño peculiar no es fatal, resulta menos eficiente porque muchos programas leen y escriben en bloques cuyo tamaño es una potencia de dos.

Asignación por lista enlazada empleando un índice

Las **dos desventajas** de la **asignación por lista enlazada** pueden eliminarse si se toma la palabra de apuntador de cada bloque y se le coloca en una **tabla** o **índice en la memoria**. La **Figura 5.9** muestra el aspecto que la tabla tendría para el ejemplo de la **Figura 5.8**. En ambas figuras, tenemos dos archivos. El **archivo A** usa los bloques de disco **4, 7, 2, 10** y **12**, en ese orden, y el **archivo** B usa los bloques de disco **6,3, 11** y **14**, en ese orden. Si usamos la tabla de la **Figura 5.9**, podemos comenzar en el bloque **4** y seguir la cadena hasta el final. Lo mismo puede hacerse comenzando en el bloque **6**.

Si se emplea esta organización, todo el bloque está disponible para **datos**. Además, el **acceso directo** es mucho más fácil. Aunque todavía hay que seguir la cadena para encontrar una distancia dada dentro de un archivo, la cadena está por completo en la memoria, y puede seguirse sin tener que consultar el disco. Al igual que con el método anterior, basta con guardar un solo entero (el número del bloque inicial) en la entrada de directorio para poder localizar todos los bloques, por más grande que sea el archivo. **MS-DOS** emplea este método para la asignación en disco.

Figura 5.9: Asignación por lista enlazada empleando una tabla en la memoria principal.

La desventaja primordial de este método es que **toda la tabla** debe estar en la **memoria** todo el tiempo para que funcione. En el caso de un disco grande con, digamos, 500,000 bloques de 1K (500M), la tabla tendrá 500,000 entradas, cada una de las cuales tendrá que tener un mínimo de 3 bytes. Si se desea acelerar las búsquedas, se necesitarán 4 bytes. Así, la tabla ocupará de 1.5 a 2 megabytes todo el tiempo, dependiendo de si el sistema se optimiza en cuanto al **espacio** o en cuanto al **tiempo**. Aunque **MS-DOS** emplea este mecanismo, evita manejar tablas muy grandes empleando bloques grandes (de hasta 32K) en los discos de gran tamaño.

Nodos-i

Nuestro último método para saber **cuáles bloques** pertenecen a **cuál archivo** consiste en asociar a cada archivo una pequeña tabla llamada **nodo-i** (**nodo-índice**), que lista los **atributos** y las **direcciones** en disco de los **bloques del archivo**, como se muestra en la **Figura 5.10**.

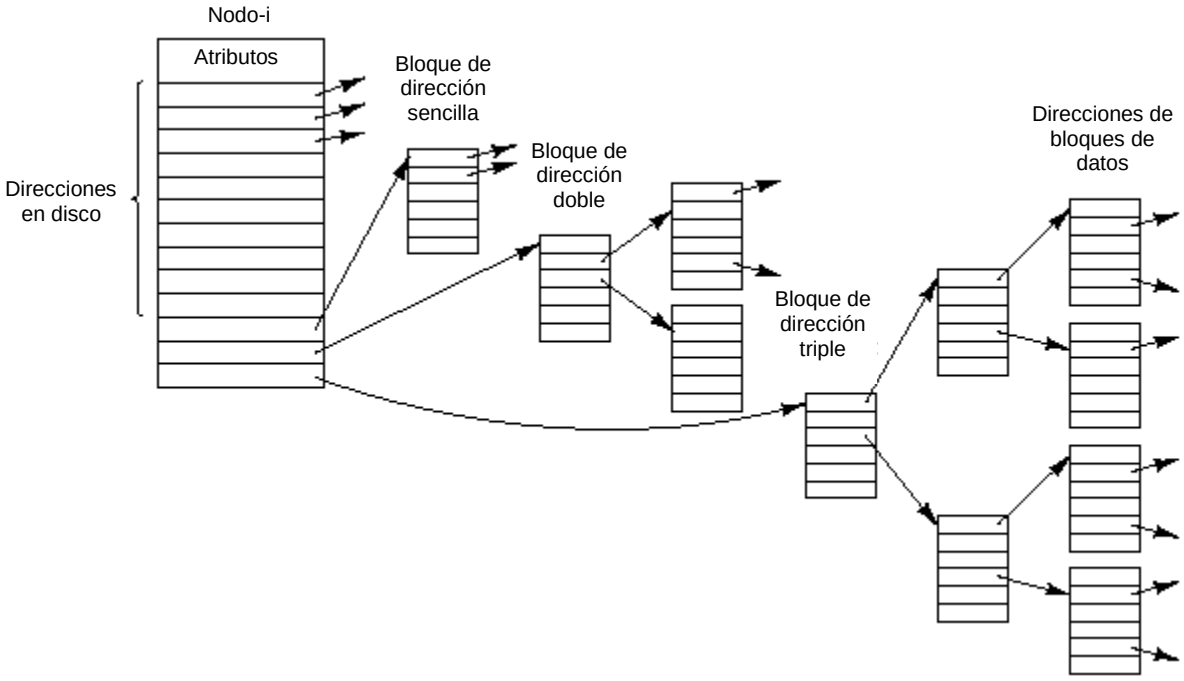

Figura 5.10: Un nodo-i.

Las primeras pocas direcciones de disco se almacenan en el **nodo-i** mismo, así que en el caso de archivos pequeños toda la información está contenida en el **nodo-i**, que se trae del disco a la memoria principal

cuando se abre el archivo. En el caso de archivos más grandes, una de las direcciones del **nodo-i** es la dirección de un bloque de disco llamado **bloque de indirección sencilla**. Este bloque contiene direcciones de disco adicionales. Si esto todavía no es suficiente otra dirección del **nodo-i**, llamada **bloque de indirección doble**, contiene la dirección de un bloque que contiene una lista de bloques de indirección sencilla. Cada uno de estos bloques de indirección sencilla apunta a unos cuantos cientos de bloques de datos. Si ni siquiera con esto basta, se puede usar también un **bloque de indirección triple**. **UNIX** emplea este esquema.

5.3.2 Implementación de directorios

Antes de poder leer un archivo, hay que abrirlo. Cuando se **abre un archivo**, el sistema operativo usa el nombre de ruta proporcionado por el usuario para localizar la **entrada de directorio**. Esta entrada proporciona la información necesaria para encontrar los bloques de disco. Dependiendo del sistema, esta información puede ser la dirección en disco de todo el archivo (asignación contigua), el número del primer bloque (ambos esquemas de lista enlazada) o el número del **nodo-i**. En todos los casos, la función principal del sistema de directorios es transformar el **nombre ASCII** del archivo en la información necesaria para localizar los **datos**.

Un problema muy relacionado con el anterior es dónde deben almacenarse los **atributos**. Una posibilidad obvia es almacenarlos directamente en la **entrada de directorio**. Muchos sistemas hacen precisamente esto. En el caso de sistemas que usan **nodos-i**, otra posibilidad es almacenar los atributos en el **nodo-i**, en lugar de en la **entrada de directorio**. Como veremos más adelante, este método tiene ciertas ventajas respecto a la colocación de los atributos en la entrada de directorio.

Directorios en MS-DOS

Es un ejemplo de sistema con **árboles de directorios jerárquicos**. La **Figura 5.11** muestra una entrada de directorio de **MS-DOS**, la cual tiene 32 bytes de longitud y contiene el **nombre de archivo**, los **atributos** y el **número del primer bloque de disco**. El **primer número de bloque** se emplea como **índice de una tabla** del tipo de la de la **Figura 5.9**. Siguiendo la cadena, se pueden encontrar todos los bloques.

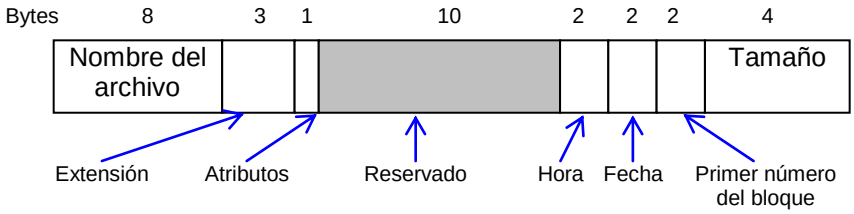

Figura 5.11: La entrada de directorio de MS-DOS.

En **MS-DOS**, los directorios pueden contener otros directorios, dando lugar a un **sistema de archivos jerárquico**. En este sistema operativo es común que los diferentes programas de aplicación comiencen por crear un directorio en el **directorio raíz** y pongan ahí todos sus archivos, con objeto de que no haya conflictos entre las aplicaciones.

Directorios en UNIX

La **estructura de directorios** que se usa tradicionalmente en **UNIX** es en extremo sencilla, como se aprecia en la **Figura 5.12**. Cada entrada contiene sólo un **nombre de archivo** y su **número de nodo-i**, Toda la información acerca del tipo, tamaño, tiempos, propietario y bloques de disco está contenida en el **nodo-i**. Algunos sistemas **UNIX** tienen una organización distinta, pero en todos los casos una entrada de directorio contiene en última instancia sólo una cadena ASCII y un número de **nodo-i**,

Cuando se **abre un archivo**, el sistema de archivos debe tomar el nombre que se le proporciona y localizar sus bloques de disco. Consideremos cómo se busca el nombre de ruta `/usr/ast/mbox`. Usaremos **UNIX** como ejemplo, pero el algoritmo es básicamente el mismo para todos los **sistemas de directorios jerárquicos**. Lo primero que hace el sistema de archivos es localizar el **directorio raíz**. En **UNIX** su **nodo-i** está situado en un lugar fijo del disco.

A continuación, el sistema de archivos busca el primer componente de la ruta, `usr`, en el **directorio raíz** para encontrar el **número de nodo-i** del archivo `/usr`. La localización de un **nodo.i**, una vez que se tiene su número es **directa**, ya que cada uno tiene una posición fija en el disco. Con la ayuda de este **nodo- i**, el sistema localiza el directorio correspondiente a `/usr` y busca el siguiente componente, `ast`, en él. Una vez que el sistema encuentra la entrada para **ast**, obtiene el **nodo-i** del directorio **/usr/ast**. Con este nodo, el sistema puede encontrar el directorio mismo y buscar `mbox`. A continuación se trae a la memoria el `nodo-i` de este archivo y se mantiene ahí hasta que se cierra el archivo. El proceso de búsqueda se ilustra en la **Figura 5.13**.

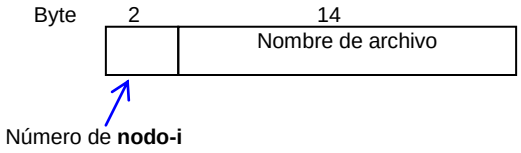

Figura 5.12: Una entrada de directorio UNIX.

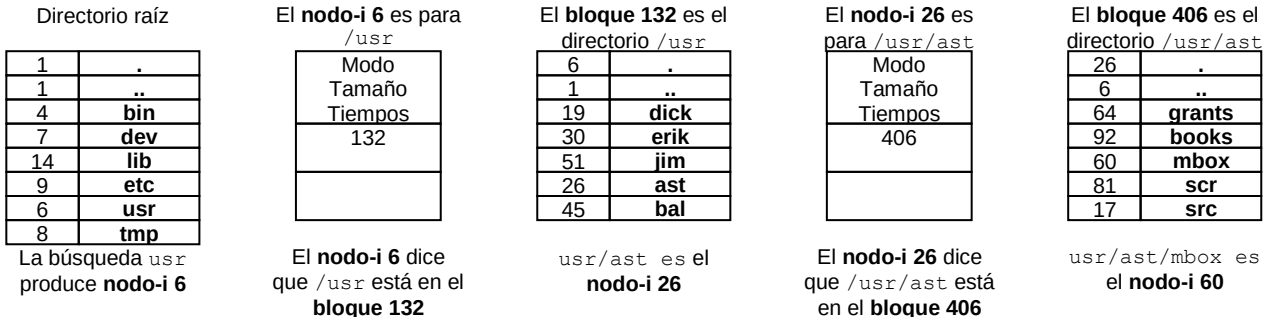

Figura 5.13: Pasos para buscar `/usr/ast/mbox`.

Los **nombres de ruta relativos** se buscan de la misma forma que los **absolutos**, sólo que el punto de partida es el **directorio de trabajo** en lugar del **directorio raíz**. Cada directorio tiene entradas para `.` y `..`, que se colocan ahí cuando se crea el directorio. La entrada `.` tiene el número de **nodo-i** del **directorio actual**, y la entrada `..` tiene el número de **nodo-i** del **directorio padre**. Así, un procedimiento que busque **`../dick/prog.c`** simplemente buscará `..` en el directorio de trabajo, encontrará el número de **nodo-i** de su **directorio padre** y buscará `dick` en ese directorio. No se requiere ningún mecanismo especial para manejar estos nombres. En lo que al sistema de directorios concierne, se trata de cadenas ASCII ordinarias, lo mismo que los demás nombres.

5.3.3 Administración del espacio en disco

Los archivos normalmente se almacenan en disco, así que la administración del espacio en disco es de interés primordial para los diseñadores de sistemas de archivos. Hay dos posibles estrategias para almacenar un archivo de **n bytes**: asignar **n bytes consecutivos** de espacio en disco, o dividir el archivo en varios bloques (no necesariamente) contiguos. Un trueque similar (entre segmentación pura y paginación) está presente en los sistemas de administración de memoria.

El almacenamiento de un archivo como **secuencia contigua** de bytes tiene el problema obvio de que si el archivo crece, probablemente tendrá que pasarse a otro lugar del disco. El mismo problema ocurre con los segmentos en la memoria, excepto que el traslado de un segmento en la memoria es una operación relativamente rápida comparada con el traslado de un archivo de una posición en el disco a otra. Por esta razón, casi todos los sistemas de archivos dividen los archivos en bloques de tamaño fijo que no necesitan estar adyacentes.

Tamaño de bloque

Una vez que se ha decidido almacenar archivos en bloques de tamaño fijo, surge la pregunta de qué **tamaño** deben tener los bloques. Dada la forma como están organizados los discos, el **sector**, la **pista** y el

cilindro son candidatos obvios para utilizarse como **unidad de asignación**. En un sistema con paginación, el tamaño de página también es un contendiente importante.

Tener una **unidad de asignación grande**, como un **cilindro**, implica que cada archivo, incluso un archivo de un byte, ocupará todo un cilindro. Estudios realizados han demostrado que la mediana del tamaño de los archivos en los entornos **UNIX** es de alrededor de **1K**, así que asignar un cilindro de **32K** a cada archivo implicaría un desperdicio de **31 / 32 = 97%** del espacio en disco total. Por otro lado, el empleo de una **unidad de asignación pequeña** implica que cada archivo consistirá en muchos bloques. La lectura de cada bloque normalmente requiere una búsqueda y un retardo rotacional, así que la lectura de un archivo que consta de muchos bloques pequeños será lenta.

Por ejemplo, consideremos un disco con `32,768` bytes por pista, tiempo de rotación de `16.67` ms y tiempo de búsqueda medio de `30` ms. El tiempo en milisegundos requerido para leer un bloque de **k bytes** es la suma de los tiempos de búsqueda, retardo rotacional y transferencia:

$$30 + 8.3 + (k / 32{,}768) \times 16.67$$

La curva continúa de la **Figura 5.14** muestra la **tasa de datos** para un disco con estas características en función del **tamaño de bloque**. Si utilizamos el supuesto burdo de que todos los bloques son de **1K** (la mediana de tamaño medida), la curva de guiones de la **Figura 5.14** indicará la **eficiencia** de espacio del disco. La mala noticia es que una buena utilización del espacio (tamaño de bloque **< 2K**) implica **tasas de datos bajas** y viceversa. La **eficiencia de tiempo** y la **eficiencia de espacio** están inherentemente en conflicto. El término medio usual es escoger un tamaño de bloque de **512**, **1K** ó **2K** bytes. Si se escoge un tamaño de bloque de **1K** en un disco cuyos sectores son de **512** bytes, el sistema de archivos siempre leerá o escribirá dos sectores consecutivos y los tratará como una sola unidad, indivisible. Sea cual sea la decisión que se tome, probablemente convendrá reevaluarla periódicamente, ya que, al igual que sucede con todos los aspectos de la tecnología de computadoras, los usuarios aprovechan los recursos más abundantes exigiendo todavía más.

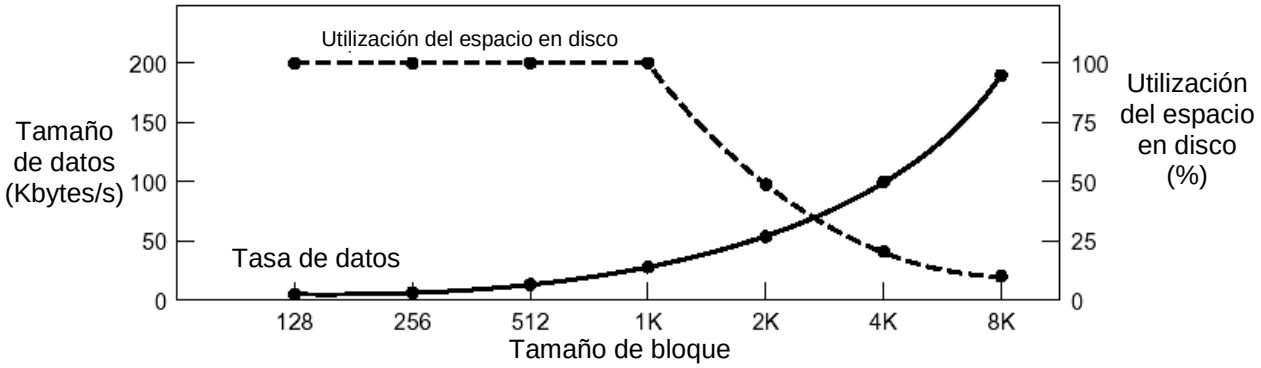

Figura 5.14: La curva continua (escala de la izquierda) da la tasa de datos de un disco. La curva de guiones (escala de la derecha) da la eficiencia de espacio de disco. Todos los archivos son de 1K.

Administración de bloques libres

Una vez que se ha escogido el tamaño de bloque, el siguiente problema es cómo seguir la pista a los **bloques libres**. Se utilizan ampliamente dos métodos, mismos que se ilustran en la **Figura 5.15**. El primero consiste en usar una **lista enlazada de bloques de disco**, en la que cada bloque contiene tantos números de bloques de disco libres como quepan en él. Con bloques de **1K** y números de bloque de **32** bits, cada bloque de la lista libre contiene los números de **255** bloques libres. (Se requiere una ranura para el apuntador al siguiente bloque.) Un disco de **200M** necesita una lista libre de, como máximo, **804** bloques para contener los **200K** números de bloque. Es común que se usen bloques libres para contener la lista libre.

La otra técnica de administración del espacio libre es el **mapa de bits**. Un disco con **n bloques** requiere un **mapa de bits con n bits**. Los bloques libres se representan con unos en el mapa, y los bloques asignados

con ceros (o viceversa). Un disco de **200M** requiere **200K** bits para el mapa, mismos que ocupan sólo **25** bloques. No es sorprendente que el mapa de bits requiera menos espacio, ya que sólo usa un bit por bloque, en vez de **32** como en el modelo de lista enlazada. Sólo si el disco está casi lleno el esquema de lista enlazada requerirá menos bloques que el mapa de bits.

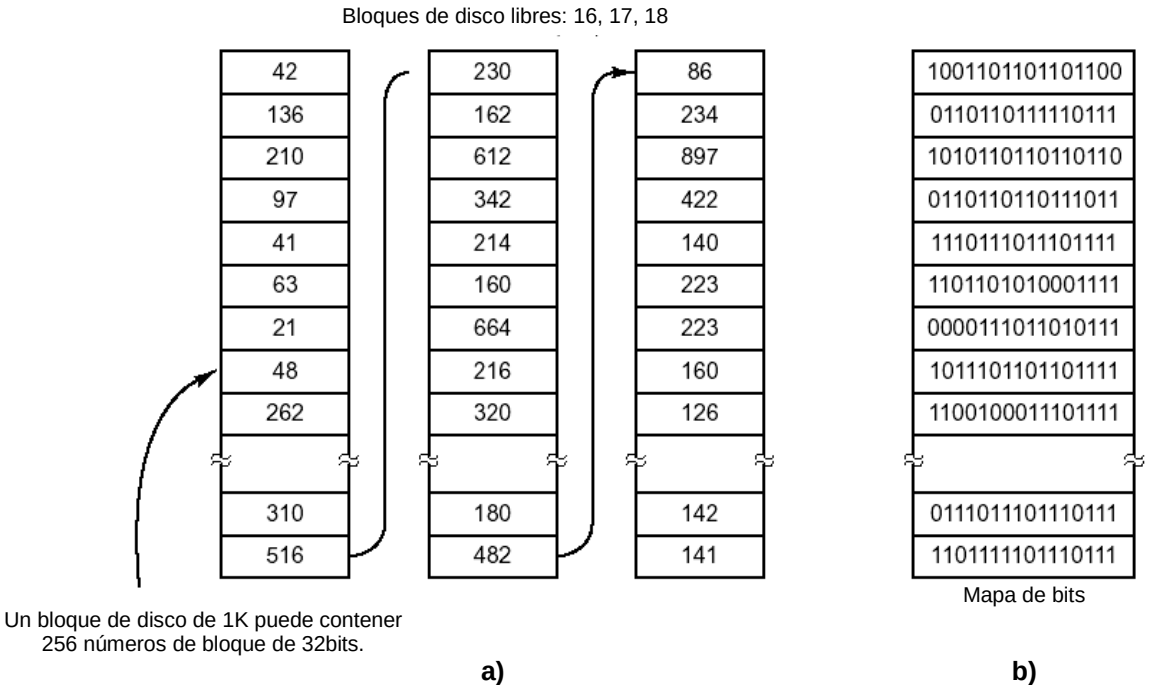

Figura 5.15: a) Almacenamiento de la lista libre en una lista enlazada. **b)** Mapa de bits.

Si hay suficiente memoria principal para contener el mapa de bits, este método generalmente es preferible. En cambio, si sólo se puede dedicar un bloque de memoria para seguir la pista a los bloques de disco libres, y el disco está casi lleno, la lista enlazada podría ser mejor. Con sólo un bloque del mapa de bits en la memoria, podría ser imposible encontrar bloques libres en él, causando accesos a disco para leer el resto del mapa de bits. Cuando un bloque nuevo de la lista enlazada se carga en la memoria, es posible asignar **255** bloques de disco antes de tener que traer del disco el siguiente bloque de la lista.

5.3.4 Confiabilidad del sistema de archivos

La destrucción de un sistema de archivos suele ser un desastre mucho peor que la destrucción de una computadora. Si una computadora se destruye por un incendio, picos de descarga eléctrica o una taza de café vertida sobre el teclado, esto implica una molestia y cuesta dinero, pero generalmente se puede adquirir un sustituto con un mínimo de problemas. Las computadoras personales económicas pueden reemplazarse en unas cuantas horas con sólo visitar una tienda.

Si el **sistema de archivos** de una computadora se pierde irremisiblemente, sea por causa del hardware, del software, o ratas que mordisquearon los discos flexibles, la restauración de toda la información es difícil, tardada y, en muchos casos, imposible. Para las personas cuyos programas, documentos, archivos de clientes, registros fiscales, bases de datos, planes de mercadeo u otros datos han dejado de existir, las consecuencias pueden ser catastróficas. Si bien el **sistema de archivos** no puede ofrecer protección contra la destrucción física del equipo y los medios, si puede ayudar a proteger la información.

Los discos pueden tener bloques defectuosos. Los discos flexibles normalmente son perfectos cuando salen de la fábrica, pero pueden adquirir defectos durante el uso. Los **discos winchester** con frecuencia tienen bloques defectuosos aun siendo nuevos; simplemente cuesta demasiado fabricarlos completamente libres de todo defecto. De hecho, antes los discos duros solían entregarse con una lista de los bloques defectuosos descubiertos por las pruebas del fabricante. En tales discos se reserva un sector para contener una lista de

bloques defectuosos. Cuando se inicializa originalmente el controlador en hardware, éste lee la lista de bloques defectuosos y escoge un bloque (o una pista) de repuesto para sustituir los defectuosos, registrando la correspondencia en la lista de bloques defectuosos. A partir de entonces, todas las solicitudes que pidan el bloque defectuoso usarán el de repuesto. Cada vez que se descubren nuevos errores se actualiza esta lista como parte de un formato de bajo nivel.

Ha habido una mejoría constante en las técnicas de fabricación, de modo que los bloques defectuosos son menos comunes que antaño; sin embargo, no han desaparecido. El controlador en hardware de una unidad de disco moderna es muy complejo. En estos discos, las pistas tienen por lo menos un sector más que el número de sectores necesarios con objeto de que al menos un punto defectuoso pueda pasarse por alto dejándolo en un hueco entre dos sectores consecutivos. También hay unos cuantos sectores de repuesto en cada cilindro con objeto de que el controlador pueda establecer una nueva correspondencia de sectores si se percata de que un sector requiere más de cierto número de reintentos para leerse o escribirse. Así, el usuario casi nunca se da cuenta de la existencia de bloques defectuosos ni de su administración. No obstante, cuando un disco **IDE** o **SCSI** moderno falla, casi siempre falla gravemente, porque se ha quedado sin sectores de repuesto. Los discos **SCSI** informan de un **error recuperado** cuando alteran la correspondencia de un bloque. Si el controlador en software se da cuenta de esto y exhibe un mensaje en la pantalla, el usuario sabrá que es momento de comprar un nuevo disco cuando estos mensajes comiencen a aparecer con frecuencia.

Existe una solución de software sencilla al problema de los bloques defectuosos, apropiado para usarse en los discos menos modernos. La estrategia requiere que el usuario del sistema de archivos construya cuidadosamente un archivo que contenga todos los bloques defectuosos. Esta técnica retira dichos bloques de la lista libre, así que nunca se usarán para archivos de datos. En tanto nunca se lea ni escriba el archivo de bloques defectuosos, no habrá problemas. Se debe tener cuidado durante los respaldos de disco para evitar la lectura de este archivo.

Respaldos

Incluso teniendo una estrategia ingeniosa para manejar los bloques defectuosos, es importante **respaldar los archivos con frecuencia**. Después de todo, cambiar automáticamente a una pista de repuesto después de que se ha arruinado un bloque que contenía datos cruciales es algo parecido a cerrar la puerta de la caballeriza después de que ha escapado un valioso caballo de carreras.

Los sistemas de archivos en disco flexible se pueden **respaldar** con sólo copiar todo el disco en uno en blanco. Los sistemas de archivos en **discos winchester** pequeños se pueden respaldar vaciando todo el disco en **cinta magnética**. Entre las tecnologías actuales están los cartuchos de cinta de **150M** y las cintas **Exabyte** o **DAT** de **8G**.

En el caso **de discos winchester grandes**, el respaldo de toda la unidad en cinta es tedioso y tardado. Una estrategia que es fácil de implementar pero desperdicia la mitad del espacio del almacenamiento es instalar en cada computadora dos unidades de disco en lugar de una. Ambas unidades se dividen en dos mitades: **datos** y **respaldo**. Cada noche la porción de datos de la **unidad 0** se copia en la porción de respaldo de la **unidad 1**, y viceversa, como se muestra en la **Figura 5.18**. De esta forma, si una unidad se arruina por completo no se perderá información.

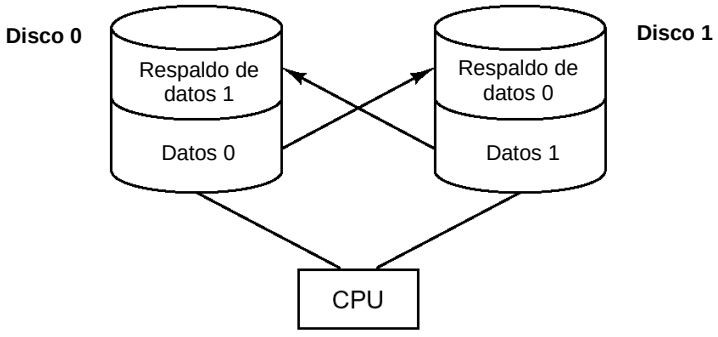

Figura 5.18: Respaldo de cada unidad en la otra desperdicia la mitad del espacio de almacenamiento.

Una alternativa al vaciado diario de todo el sistema de archivos es realizar **vaciados incrementales**. La forma más sencilla de **vaciado incremental** consiste en efectuar un **vaciado completo** periódicamente, digamos cada semana o cada mes, y realizar un vaciado diario de sólo aquellos archivos que han sido modificados después del último vaciado completo. Un esquema mejor sería vaciar sólo aquellos archivos que han cambiado desde que se vaciaron por última vez.

Para implementar este método, se debe mantener en disco una lista de los tiempos de vaciado para cada archivo. El programa de vaciado revisa cada archivo del disco. Si un archivo fue modificado después de la última vez que se vació, se vacía otra vez y su tiempo de último vaciado se cambia al tiempo actual. Si esto se hace en un ciclo mensual, el método requiere **31** cintas de vaciado diarias, una por día, más suficientes cintas para contener un vaciado completo, que se efectúa una vez al mes. También se usan otros esquemas más complejos que emplean menos cintas.

También están en uso **métodos automáticos** que emplean múltiples discos. Por ejemplo, la creación de espejos emplea dos discos. Las escrituras se efectúan en ambos discos, y las lecturas provienen de uno solo. La escritura en el disco espejo se retrasa un poco, efectuándose cuando el sistema está ocioso. Un sistema así puede seguir funcionando en **modo degradado** si un disco falla, y esto permite reemplazar el disco que falló y recuperar los datos sin tener que parar el sistema.

Consistencia del sistema de archivos

Otra área en la que la confiabilidad es importante es la **consistencia del sistema de archivos**. Muchos sistemas de archivos leen bloques, los modifican y los vuelven a escribir después. Si el sistema se cae antes de que todos los bloques modificados se hayan escrito en disco, el sistema de archivos puede quedar en un estado inconsistente. Este problema se vuelve crítico si algunos de los bloques que no se han escrito contienen **nodos-i**, **entradas de directorio** o la **lista libre**.

Para enfrentar el problema de un sistema de **archivos inconsistente**, la mayor parte de las computadoras cuentan con un programa de utilería que verifica la consistencia del sistema de archivos. Este programa puede ejecutarse cuando el sistema se arranca, sobre todo después de una caída. La descripción que sigue se refiere al funcionamiento de tal utilería en `UNIX` y otros sistemas tienen algo similar. Estos verificadores del sistema de archivos examinan cada sistema de archivos (disco) con independencia de los demás.

Se pueden realizar dos tipos de verificaciones de consistencia: de **bloques** y de **archivos**. Para comprobar la **consistencia de los bloques**, el programa construye dos tablas, cada una de las cual contiene un contador para cada bloque, que inicialmente vale `0`. Los contadores de la primera tabla llevan la cuenta de cuántas veces un bloque está presente en un archivo; los contadores de la segunda tabla registran cuántas veces está presente cada bloque en la lista libre (o en el mapa de bits de bloques libres).

A continuación, el programa lee todos los **nodos-i**. Partiendo de un **nodo-i**, es posible construir una lista de todos los números de bloque empleados en el archivo correspondiente. Al leerse cada número de bloque, su contador en la primera tabla se incrementa. Después, el programa examina la lista o mapa de bits de bloques libres, para encontrar todos los bloques que no están en uso. Cada ocurrencia de un bloque en la lista libre hace que su contador en la segunda tabla se incremente.

Si el **sistema de archivos es consistente**, cada bloque tendrá un `1` ya sea en la primera tabla o en la segunda, como se ilustra en la **Figura 5.17a**. Sin embargo, después de una caída las tablas podrían tener el aspecto de la **Figura 5.17b**, en la que el **bloque 2** no ocurre en ninguna de las dos tablas. Este bloque se informará como **bloque faltante**. Aunque los bloques faltantes no representan un daño real, desperdician espacio y, por tanto, reducen la capacidad del disco. La solución en el caso de haber bloques faltantes es directa: el verificador del sistema de archivos simplemente los agrega a la lista libre.

Otra situación que podría ocurrir es la de la **Figura 5.17c**. Aquí vemos un bloque, el número `4`, que ocurre dos veces en la lista libre. (Sólo puede haber duplicados si la lista libre es realmente una lista; si es un mapa de bits esto es imposible). La solución en este caso también es simple: reconstruir la lista libre.

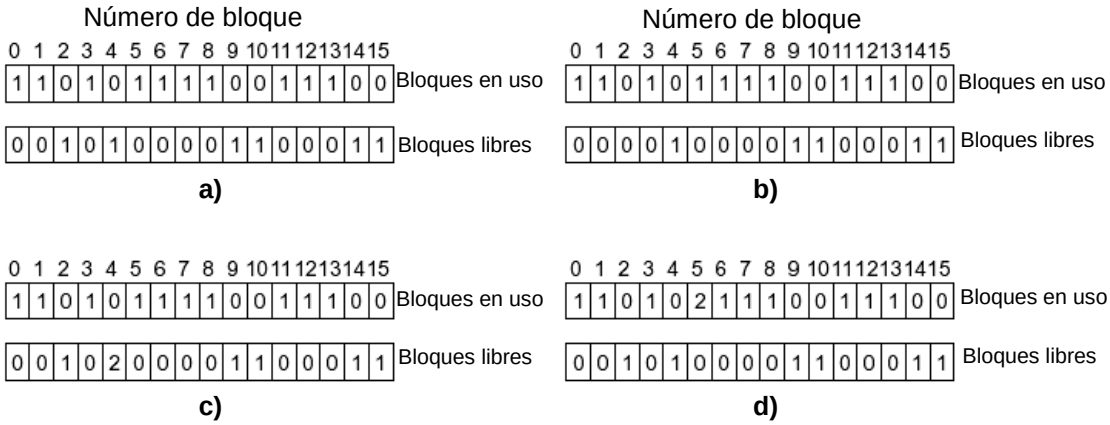

Figura 5.17: Estados del sistema de archivos. **a)** Consistente. **b)** Bloque faltante. **c)** Bloque duplicado en la lista libre. **d)** Bloque de datos duplicado.

Lo peor que puede suceder es que el mismo bloque de datos esté presente en dos o más archivos, como se ilustra en la **Figura 5.17b** con el **bloque 5**. Si cualquiera de estos archivos se elimina, el **bloque 5** se colocará en la lista libre, dando lugar a una situación en la que el mismo bloque está en uso y libre al mismo tiempo. Si se eliminan ambos archivos, el bloque se colocará en la lista libre dos veces.

La acción apropiada para el verificador del sistema de archivos es asignar un bloque libre, copiar en él el contenido del **bloque 5**, e insertar la copia en uno de los archivos. De este modo, el contenido de información de los archivos no cambiará (aunque casi con toda seguridad estará revuelto), y la estructura del sistema de archivos al menos será consistente. Se deberá informar del error, para que el usuario pueda inspeccionar los daños.

Además de verificar que cada bloque esté donde debe estar, el **verificador del sistema de archivos** también revisa el **sistema de directorios**. En este caso también se usa una tabla de contadores, pero ahora uno por cada archivo, no por cada bloque. El programa parte del directorio raíz y desciende recursivamente el árbol, inspeccionando cada directorio del sistema de archivos, Para cada archivo de cada directorio, el verificador incrementa el contador correspondiente al **nodo-i** de ese archivo.

Una vez que termina la revisión, el verificador tiene una lista, indizada por número de **nodo-i**, que indica cuántos directorios apuntan a ese **nodo-i**. Luego, el programa compara estos números con los conteos de enlace almacenados en los **nodos-i** mismos. En un sistema de archivos consistente, ambos conteos coinciden. Sin embargo, pueden ocurrir dos tipos de errores: el conteo de enlaces del **nodo-i** puede ser demasiado alto o demasiado bajo.

Si el conteo de enlaces es mayor que el número de entradas de directorio, entonces aunque se borraran todos los archivos de todos los directorios el conteo seguiría siendo mayor que cero y el **nodo-i** no se eliminaría. Este error no es grave, pero desperdicia espacio en el disco con archivos que no están en ningún directorio, así que debe corregirse asignando el valor correcto al conteo de enlaces del **nodo-i**.

El otro error puede ser catastrófico. Si dos entradas de directorio están enlazadas a un archivo, pero el **nodo-i** dice que sólo hay una, cuando se elimine cualquiera de las entradas de directorio, el conteo del **nodo-i** se convertirá en cero. Cuando esto suceda, el sistema de archivos lo marcará como desocupado y liberará todos sus bloques. El resultado de esta acción es que uno de los directorios ahora apunta a un **nodo-i** desocupado, cuyos bloques pronto pueden ser asignados a otros archivos. Una vez más, la solución es hacer que el conteo de enlaces del **nodo-i** sea igual al número real de entradas de directorio.

Estas dos operaciones, **verificar bloques** y **verificar directorios**, a menudo se integran por razones de eficiencia. También pueden realizarse otras verificaciones heurísticas. Por ejemplo, los directorios tienen un formato definido, con **números de nodo-i** y **nombres ASCII**. Si un número de **nodo-i** es mayor que el número de **nodos-i** que hay en el disco, es señal de que el directorio ha sido dañado.

Además, cada **nodo-i** tiene un **modo**, y algunos de estos modos son permitidos pero extraños, como el `0007`, que no permite que el propietario ni su grupo tengan acceso, pero sí permite a terceros leer, escribir y ejecutar el archivo. Podría ser útil al menos informar de la existencia de archivos que dan más derechos a terceros que al propietario. Los directorios que tienen más de **1000 entradas** también son sospechosos. Los archivos situados en directorios de usuarios, pero que son propiedad del superusuario y tienen encendido el bit `SETUID`, implican posibles **problemas de seguridad**. Con un poco de esfuerzo, es posible hacer una lista más o menos larga de situaciones permitidas pero peculiares que valdría la pena informar.

Los párrafos anteriores abordaron el problema de proteger al usuario contra las caídas del sistema. Algunos sistemas de archivos también se preocupan por proteger al usuario contra sí mismo. Si la intención del usuario era teclear: **`rm *.o`**, para eliminar todos los archivos que terminan con **`.o`** (archivos objeto generados por el compilador), pero accidentalmente teclea: **`rm * .0`** (espacio después del asterisco), **`rm`** eliminará todos los archivos del directorio actual y después se quejará de que no puede encontrar a **`.o`**. En **MS-DOS** y algunos otros sistemas, cuando un archivo se elimina lo único que sucede es que se enciende un bit en el directorio o **nodo-i** para marcar ese archivo como **eliminado**. No se devuelven bloques de disco a la **lista libre** en tanto no se necesitan realmente. Por tanto, si el usuario descubre el error de inmediato, es posible ejecutar un programa de utilería especial que **deselimina** (**restaura**) los archivos eliminados. En **WINDOWS**, los archivos eliminados se colocan en un **directorio de reciclado** especial, del cual pueden recuperarse posteriormente si es necesario. Desde luego, no se recupera espacio de almacenamiento en tanto los archivos no se borran de dicho directorio especial.

5.3.5 Rendimiento del sistema de archivos

El acceso a un disco es mucho más lento que el acceso a la memoria. La lectura de una palabra de memoria por lo regular toma **decenas de nanosegundos**. La lectura de un bloque de un disco duro puede tardar **50 microsegundos**, lo que implica que es cuatro veces más lenta por **palabra de 32** bits, pero a esto deben sumarse de **10** a **20 milisegundos** para mover la cabeza de lectura a la pista correcta y luego esperar a que el sector deseado se coloque debajo de la cabeza. Si sólo se necesita una palabra, el acceso a la memoria será del orden de `100,000` veces más rápido que al disco. En vista de esta diferencia en el tiempo de acceso, muchos sistemas de archivos se han diseñado pensando en reducir el número de accesos a disco requeridos.

La técnica más común empleada para reducir los accesos a disco es el **caché de bloques** o el **caché de buffer** (la palabra **caché** proviene del verbo francés **cacher**, que significa **esconder**). En este contexto, un caché es una colección de bloques que lógicamente pertenecen al disco pero que se están manteniendo en la memoria por razones de rendimiento.

Se pueden usar diversos algoritmos para administrar el caché, pero uno de los más comunes es inspeccionar todas las solicitudes de lectura para ver si el bloque requerido está en el caché. Si es así, la solicitud de lectura se puede satisfacer sin un acceso a disco. Si el bloque no está en el caché, primero se lee del disco y se coloca en el caché, y luego se copia al lugar donde se necesita. Las solicitudes subsecuentes del mismo bloque se pueden satisfacer desde el caché.

Si es necesario cargar un bloque en el caché y éste está lleno, es preciso deshacerse de un bloque y escribirlo en el disco si ha sido modificado desde que se trajo del disco. Esta situación es muy similar a la paginación, y todos los algoritmos de paginación, como FIFO, segunda oportunidad y LRU, son aplicables. Una diferencia agradable entre la paginación y el manejo de caché es que las referencias al caché son relativamente poco frecuentes, por lo que se hace factible mantener todos los bloques en orden LRU exacto empleando listas enlazadas.

Desafortunadamente, hay un pequeño detalle. Ahora que tenemos una situación en la que es posible usar LRU exacto, resulta que este algoritmo es indeseable. El problema tiene que ver con las **caídas** y la **consistencia** del sistema de archivos que vimos en la sección anterior. Si un bloque crítico, digamos un bloque de **nodo-i**, se coloca en el caché y se modifica, pero no se rescriben en el disco, una caída dejaría al sistema de archivos en un estado inconsistente. Si el bloque de **nodo-i** se coloca al final de la cadena de LRU, puede pasar un buen rato antes de que llegue al principio y se rescriba en el disco.

Además, a algunos bloques, como los de **doble indirección**, casi nunca se hace referencia dos veces

dentro de un tiempo corto. Estas consideraciones dan lugar a un esquema LRU modificado, teniendo en cuenta dos factores:

1. ¿Es probable que el bloque se necesite pronto otra vez?
2. ¿Es indispensable el bloque para la consistencia del sistema de archivos?

Para poder contestar ambas preguntas, los bloques pueden dividirse en categorías, como bloques de **nodo-i**, bloques de indirección, bloques de directorio, bloques de datos llenos y bloques de datos parcialmente llenos. Los bloques que probablemente no se necesitarán pronto otra ven colocan al frente, no al final, de la lista de LRU, con objeto de que sus buffers se reutilicen rápidamente. Los bloques que podrían necesitarse pronto otra vez, como un bloque parcialmente lleno que se está escribiendo, se colocan al final de la lista, para que estén en ella un buen tiempo,

La segunda pregunta es independiente de la primera. Si el bloque es indispensable para la consistencia del sistema de archivos (básicamente, todo excepto los bloques de datos) y ha sido modificado, se deberá escribir de inmediato en el disco, sin importar en qué extremo de la lista LRU se coloque. Al escribir los bloques críticos rápidamente, reducimos en buena medida la probabilidad de que una caída arruine el sistema de archivos.

Incluso con esta medida para mantener intacta la integridad del sistema de archivos, no es deseable mantener los bloques de datos demasiado tiempo en el caché antes de escribirlos en el disco. Consideremos la situación de una persona que está usando una computadora personal para escribir un disco. Incluso si nuestro escritor le dice periódicamente al editor que grabe en el disco el archivo que está editando, hay una buena probabilidad de que todo estará aún en el caché y nada en el disco. Si el sistema se cae, la estructura del sistema de archivos no estará corrompida, pero se habrá perdido todo un día de trabajo.

No hace falta que ocurra esta situación muchas veces para que tengamos un usuario muy descontento. Los sistemas adoptan dos enfoques para manejar el problema. Lo que hace **UNIX** es tener una llamada al sistema, **`SYNC`**, que obliga la escritura inmediata en disco de todos los bloques modificados. Cuando el sistema se inicia, se pone en marcha un programa de segundo plano, normalmente llamado **`update`**, que da vueltas en un ciclo infinito emitiendo llamadas **`SYNC`** y durmiendo durante 30 segundos entre una llamada y otra. Así, nunca se pierden más de 30 segundos de trabajo a causa de una caída.

Lo que hace **MS-DOS** es grabar todos los bloques modificados en el disco tan pronto como se modifican. Los caches en los que todos los bloques modificados se escriben en el disco de inmediato se denominan **caches de escritura inmediata,** y requieren mucha más actividad de E/S de disco que los de otro tipo. La diferencia entre estos dos enfoques puede verse cuando un programa escribe un bloque de **1K** carácter por carácter. **UNIX** acumula todos los caracteres en el caché y escribe el bloque en disco una vez cada 30 segundos, o cuando el bloque se quita del caché. **MS-DOS** efectúa un acceso a disco por cada carácter que se escribe. Desde luego, la mayor parte de los programas manejan buffers internos, así que normalmente no escriben un carácter, sino una línea o una unidad más grande en cada llamada al sistema **`WRITE`**.

Una consecuencia de esta diferencia en la estrategia de caché es que si retiramos un disco (flexible) de un sistema **UNIX** sin efectuar **`SYNC`** casi siempre perderemos datos, y en algunos casos corromperemos también el sistema de archivos. En **MS-DOS** no hay problema. Se escogieron estas diferentes estrategias porque **UNIX** se desarrolló en un entorno en el que todos los discos eran duros y no removibles, mientras que **MS-DOS** se inició en el mundo de los disquetes. Al generalizarse el uso de discos duros, incluso en las microcomputadoras pequeñas, el enfoque de **UNIX**, al ser más eficiente, definitivamente será el del futuro.

El uso de caché no es la única forma de aumentar el rendimiento de un sistema de archivos. Otra técnica importante es reducir la cantidad de movimiento del brazo del disco colocando cerca unos de otros, de preferencia en el mismo cilindro, bloques a los que probablemente se accederá en secuencia. Cuando se escribe un archivo de salida, el sistema de archivos tiene que asignar los bloques uno por uno, conforme se necesitan. Si los bloques libres están registrados en un mapa de bits, y el mapa de bits completo está en la memoria principal, no es difícil escoger un bloque libre lo más cercano posible al bloque anterior. En el caso de una lista libre, ubicada parcialmente en disco, es mucho más difícil asignar bloques cercanos unos a otros.

No obstante, incluso con una **lista libre**, es posible lograr cierto **agrupamiento de bloques**. El truco consiste en seguir la pista al espacio de almacenamiento en disco no por bloques, sino por **grupos de bloques** consecutivos. Si una pista consta de 64 sectores de 512 bytes, el sistema podría usar bloques de **1K** (dos sectores) pero asignar espacio de almacenamiento en disco en unidades de dos bloques (cuatro sectores). Esto no es lo mismo que tener bloques de disco de **2K**, ya que el caché seguiría usando bloques de **1K** y las transferencias de disco también serían de **1K**, pero la lectura secuencial de un archivo en un sistema que por lo demás está ocioso reduciría el número de búsquedas a la mitad, mejorando considerablemente el rendimiento.

Una variación del mismo tema es tener en cuenta el **posicionamiento rotacional**. Al asignar bloques, el sistema intenta colocar bloques consecutivos de un archivo en el mismo cilindro, pero intercalados a fin de obtener un rendimiento máximo. Por tanto, si un disco tiene un tiempo de rotación de **16.67 ms** y un proceso de usuario tarda unos **4 ms** para solicitar y obtener un bloque de disco, cada bloque deberá colocarse al menos un cuarto de giro más allá que su predecesor.

Otro cuello de botella de rendimiento en los sistemas que usan **nodos-i** o algo equivalente es que la lectura de incluso un archivo cortó requiere dos accesos a disco: uno para el **nodo-i** y el otro para el **bloque**. En la **Figura 5.18a** se muestra la colocación usual de los **nodos-i**. Aquí todos los **nodos-i** están cerca del principio del disco, así que la distancia media entre un **nodo-i** y sus bloques es cerca de la mitad del número de cilindros, lo que implica movimientos largos del brazo.

Una forma fácil de mejorar el rendimiento es colocar los **nodos-i** en la parte media del disco, no al principio, reduciendo así a la mitad el movimiento medio del brazo para desplazarse del **nodo-i** al primer bloque. Otra idea, que se muestra en la **Figura 5.18b**, es dividir el disco en grupos cilíndricos, cada uno con sus propios **nodos-i**, bloques y lista libre. Al crear un archivo nuevo se puede escoger cualquier **nodo-i**, pero se intenta encontrar un bloque que esté en el mismo grupo de cilindros que el **nodo-i**. Si no hay uno disponible, se usa un bloque de un cilindro cercano.

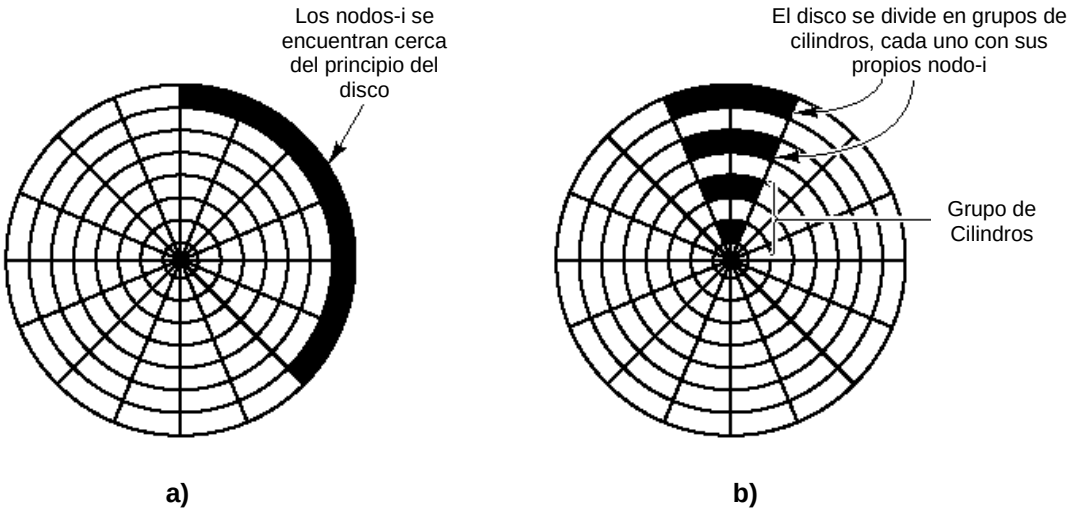

Figura 5.18: a) Nodos-i colocados al principio del disco. **b)** Disco dividido en grupos de cilindros, cada uno con sus propios bloques y nodos-i.

5.3.6 Sistemas de archivos estructurados por diario

Los cambios tecnológicos están ejerciendo presión sobre los sistemas de archivos actuales. En particular, las CPU cada vez son más rápidas, los discos cada vez son más grandes y económicos (pero no mucho más rápidos), y el tamaño de las memorias está creciendo exponencialmente. El único parámetro que no está mejorando a pasos agigantados es el **tiempo de búsqueda en disco**, La combinación de estos factores implica que en muchos sistemas de archivos está apareciendo un cuello de botella del rendimiento. Investigaciones efectuadas en Berkeley intentaron aliviar este problema diseñando un tipo totalmente nuevo de sistema de archivos, **LFS** (el **sistema de archivos estructurado por diario, log-structured file system**).

La idea en que se basa el diseño **LFS** es que conforme las CPU se hacen más rápidas y las memorias RAM se hacen más grandes, los caches de disco están aumentando aceleradamente. En consecuencia, ya es posible satisfacer una fracción sustancial de todas las solicitudes de lectura directamente del caché del sistema de archivos, sin tener que acceder al disco. De esta observación se desprende que, en el futuro, la mayor parte de los accesos a disco serán escrituras, y el mecanismo de lectura anticipada que se usa en algunos sistemas de archivos para obtener bloques antes de que se necesiten ya no representará una ganancia significativa en cuanto al rendimiento.

Para empeorar las cosas, en la mayor parte de los sistemas de archivos las escrituras se efectúan en fragmentos muy pequeños. Las **escrituras pequeñas** son muy ineficientes, ya que una escritura en disco de **50 microsegundos** típicamente va precedida por una **búsqued**a de **10 ms** y un **retardo rotacional** de **6 ms**. Con estos parámetros, la eficiencia del disco decae a una fracción de **1%**. A fin de ver de dónde provienen todas estas escrituras pequeñas, consideremos la creación de un archivo nuevo en un sistema **UNIX**. Para **escribir** este archivo, es preciso escribir el **nodo-i** del directorio, el bloque de directorio, el **nodo-i** del archivo y el archivo mismo. Si bien estas escrituras podrían diferirse, hacerlo expone el sistema de archivos a problemas de consistencia graves en caso de ocurrir una caída antes de llevarse a cabo las escrituras. Por esta razón, las escrituras de **nodos-i** generalmente se efectúan de inmediato.

Siguiendo este razonamiento, los diseñadores de **LFS** decidieron reimplementar el sistema de archivos de **UNIX** de forma tal que se aprovechara todo el **ancho de banda del disco**, incluso en casos en los que la carga de trabajo consiste en su mayor parte en escrituras aleatorias pequeñas. La idea fundamental consiste en estructurar todo el disco como un **diario**. Periódicamente, y cuando surge una necesidad especial, todas las escrituras pendientes que se han ido almacenando en la memoria se juntan y se escriben en el disco en forma de un solo segmento contiguo al final del diario. Así, un mismo segmento puede contener **nodos-i**, bloques de directorio y **bloques de datos**, todos revueltos. Al principio de cada segmento hay un resumen del segmento, que indica el contenido del mismo. Si puede hacerse que en promedio el segmento ocupe **1 MB**, se podrá aprovechar casi todo el ancho de banda del disco.

En este diseño, siguen existiendo **nodos-i** y tienen la misma estructura que en **UNIX**, pero ahora están dispersos por todo el diario, en lugar de estar en una posición fija en el disco. No obstante, cuando se localiza un **nodo-i**, la localización de los bloques se efectúa de la forma acostumbrada. Desde luego, ahora es mucho más difícil encontrar un **nodo-i**, ya que su dirección no se puede calcular simplemente a partir de su número, como en **UNIX**. Para poder encontrar los **nodos-i**, se mantiene un mapa de ellos, indizado por número de **nodo-i**. La entrada **i** de este mapa apunta al **nodo-i i** en el disco. El mapa se mantiene en disco, pero también en caché, así que las partes de uso más intenso están en la memoria casi todo el tiempo.

Resumiendo, todas las escrituras se colocan inicialmente en buffers en la memoria, y periódicamente se escriben en el disco en un solo segmento, al final del diario. Cuando se desea abrir un archivo, se utiliza el mapa para encontrar el **nodo-i** de ese archivo. Una vez localizado este nodo, se pueden obtener de él las direcciones de los bloques. Todos los bloques mismos están en segmentos en algún lugar del diario.

Si los discos fueran infinitamente grandes, la descripción anterior ya lo habría dicho todo. Sin embargo, los discos reales son finitos, y tarde o temprano el diario ocupará todo el disco, y ya no será posible escribir más segmentos en el diario. Por fortuna, es posible que muchos segmentos existentes tengan bloques que ya no se necesitan. Por ejemplo, si se sobrescribe un archivo, su **nodo-i** apuntará ahora a los nuevos bloques, pero los antiguos todavía estarán ocupando espacio en segmentos previamente escritos.

A fin de resolver estos dos problemas, **LFS** cuenta con un **hilo limpiador** que dedica su tiempo a **explorar circularmente el diario** y **compactarlo**. Lo primero que hace el hilo es leer el resumen del primer segmento del diario para ver qué **nodos-i** y archivos contiene. Luego examina el mapa de **nodos-i** actual para ver si los **nodos-i** todavía están vigentes y si los bloques de archivo todavía están en uso. Si no es así, la información correspondiente se desecha. Los **nodos-i** y bloques que todavía están en uso se pasan a la memoria para ser escritos en el siguiente segmento. A continuación, el segmento original se marca como **libre** a fin de que el diario pueda usarlo para datos nuevos. De esta forma, el **limpiador** avanza por el diario, eliminando segmentos antiguos de la parte de atrás y colocando cualesquier datos vigentes que encuentre en la memoria para ser rescritos en el siguiente segmento. Así, el disco es un gran buffer circular con el hilo escritor agregando nuevos segmentos al frente y el hilo limpiador quitando los viejos de la parte de atrás.