El sistema de ficheros

Enric Morancho Llena
Teodor Jové Lagunas

PID_00182626

UOC
Universitat Oberta de Catalunya

Índice

Introducción

En este módulo didáctico presentamos el sistema de ficheros desde la óptica de su implementación.

En primer lugar, analizamos las estructuras de datos que configuran un sistema de ficheros (SF), las cuales se encuentran almacenadas en un dispositivo físico, generalmente en un disco. Ahora bien, estas estructuras deberán adaptarse a los requerimientos del SF. En este módulo se describirán SF de propósito general, es decir, como mínimo deben permitir la creación/lectura/modificación y el borrado de ficheros y directorios.

En segundo lugar, veremos cómo el sistema operativo manipula estas estructuras para llevar a cabo las diferentes operaciones que se pueden realizar sobre el sistema de ficheros.

A continuación, se harán algunas consideraciones sobre la eficiencia y la fiabilidad de los SF.

Finalmente, se hará una descripción de los sistemas de ficheros de propósito general de uso más habitual.

Objetivos

En este módulo didáctico encontraréis los materiales necesarios para alcanzar los objetivos siguientes:

1. Saber cómo está estructurada la información en el disco.

2. Conocer diferentes alternativas para estructurar un sistema de ficheros.

3. Saber qué alternativas hay para organizar el espacio libre del disco.

4. Conocer las posibilidades para distribuir el espacio asignado a los ficheros.

5. Aprender a identificar las estructuras de datos necesarios de la memoria para llevar a cabo las operaciones relacionadas con el sistema de ficheros.

6. Saber qué pasos debe efectuar el sistema operativo para llevar a cabo una determinada operación y conocer su motivación.

7. Ser conscientes de las técnicas que permiten mejorar la eficiencia y la fiabilidad de los sistemas de ficheros.

8. Conocer las características de los sistemas de ficheros más habituales.

1. Visión estática del sistema de ficheros

Para poder gestionar el sistema de ficheros (SF[1]), debemos saber cómo está organizado, qué estructuras de datos tiene y dónde se encuentran dentro del disco que lo contiene. Para estudiar esta organización, veremos primero cómo están organizados los discos, qué tipo de estructuras de datos tienen y qué información contienen. Después, analizaremos cómo se estructura un SF dentro de un dispositivo de almacenamiento como un disco duro, cómo se representa el espacio libre y el espacio ocupado por los ficheros y veremos también la manera de representar los directorios y dónde se sitúa el directorio raíz.

(1)SF es la sigla de *sistema de ficheros*.

1.1. La estructura del espacio del disco duro

Ahora veremos las particularidades de los discos: cómo se organizan, qué geometría presentan y qué unidades de direccionamiento y de transferencia tienen. Este análisis nos permitirá entender mejor la complejidad de estos dispositivos y la gestión que hace de ellos el SO[2].

(2)SO es la sigla de *sistema operativo*.

El espacio físico de un disco se estructura de manera similar a un disco convencional de música, pero con algunas diferencias. Para poder dirigir un octeto[3] concreto del disco, hay que seguir los pasos siguientes:

(3)En inglés, *byte*.

1) Indicar en qué cara del disco se encuentra, ya que un disco puede tener más de una o dos caras, si está formado por más de un disco magnético.

2) Decir en qué pista está. Las pistas, a diferencia de los discos de música, son círculos concéntricos, no forman una espiral única.

3) Especificar en cuál de los muchos sectores en los que se divide una pista se encuentra el byte que buscamos.

4) Explicitar cuál de los bytes del sector señalado es el que buscamos.

En resumen, la dirección de un byte se compone de: pista, cara, sector y byte.

El sector no es únicamente la unidad de direccionamiento, también es la unidad de transferencia entre el disco y el SO.

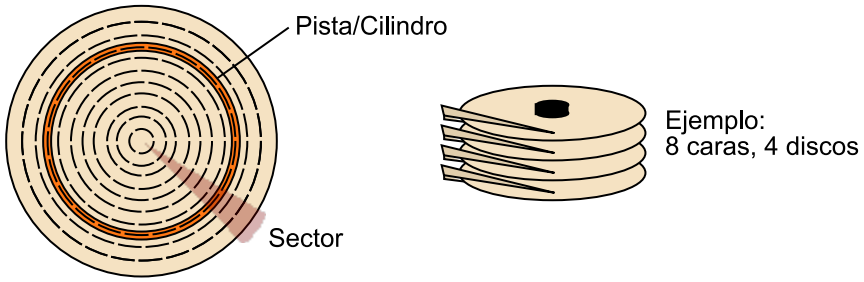
En algunos discos se puede definir, mediante el controlador del dispositivo, más de un disco lógico. Cada uno de estos discos se denominan particiones. La información sobre qué particiones hay en el disco y sobre qué características tienen se guarda en una estructura de datos llamada tabla de particiones, que suele estar en una memoria especial del controlador o en un espacio reservado del mismo disco. En caso de utilizar particiones, los sectores de la partición se dirigen correlativamente empezando por el cero, dejando de lado la pista y la cara donde se encuentran.

Características de las particiones

Algunas de las características de las particiones son el tamaño, el carácter de primaria o no, el modo de acceso (lectura/escritura), etc.

Por otra parte, el SO tiene su propia unidad de direccionamiento y transferencia: los bloques; su tamaño es múltiplo del del sector. Todos los bloques de un disco se suelen nombrar en un espacio contiguo, empezando desde el bloque cero hasta el número máximo de bloques del disco; este direccionamiento se conoce por la expresión *logical block addressing* (LBA). Así pues, el SO necesita definir una función de conversión entre bloques y sectores. Nosotros supondremos que existe esta función y, a partir de ahora, cuando hagamos alusión a un disco, nos referiremos a una partición y no hablaremos más de sectores, sino de bloques.

Figura 1. Estructura de un disco



1.2. La estructura del sistema de ficheros

El sistema de ficheros (SF[4]) se puede considerar como una estructura de datos almacenada en un disco. Las informaciones más importantes que contiene el SF son las siguientes: el tipo de SF, el espacio libre, el espacio ocupado, el directorio raíz, el modo de acceso (lectura/escritura), el propietario y la fecha de creación.

[4]SF es la sigla de *sistema de ficheros*.

La primera información, el tipo de SF, permite que el SO pueda entender más de un SF. Los tres datos siguientes, el espacio libre, los ficheros y los directorios, serán desarrollados más adelante. Finalmente, el resto de las informaciones hacen referencia a la protección o son estadísticas, y solo las mencionaremos.

La importancia de la primera información

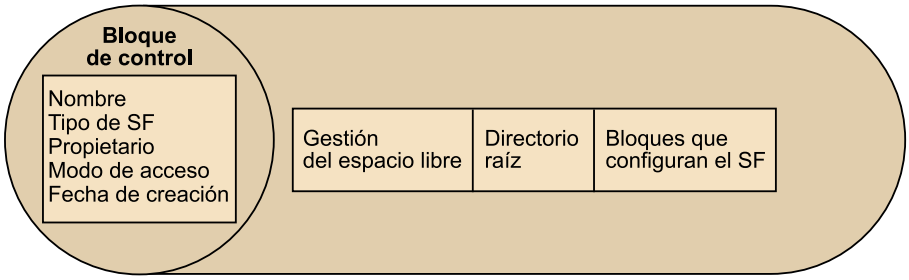
LINUX, por ejemplo, es un sistema operativo capaz de acceder a diferentes tipos de sistemas de ficheros: los propios de Linux, los del MS DOS / Windows, el de otros UNIX, como el XENIX, el SCO, etc. Además, gracias al *virtual file system* (VFS), esta diversidad

de SF es transparente al usuario. De manera análoga a los controladores de dispositivos Linux, VFS especifica una interfaz entre el núcleo del SO y cada sistema de ficheros.

Esta estructura de datos que caracteriza al SF debe estar contenida totalmente en el disco por los dos motivos siguientes:

- Para que no sea destruida cuando se desconecta el sistema.
- Para que pueda ser transportada con el disco desde un sistema a otro.

Figura 2. Sistema de ficheros



Además, esta estructura de datos debe estar situada en una posición concreta del disco, generalmente en el primer bloque, el llamado *bloque de control*. De esta manera, el SO la puede localizar con facilidad.

1.2.1. El espacio libre

La representación de los bloques libres del disco se puede hacer básicamente de dos maneras: con una lista de bloques o con un mapa de bits. A continuación, veremos con más detalle cada una de estas opciones.

1) La lista de bloques

Aquí presentamos tres maneras alternativas de representar las listas de bloques:

a) La lista de bloques libres es el modo más sencillo de representar el espacio libre. Con esta estructura para cada bloque libre se guarda en la lista el número que lo identifica (su índice). La lista deberá estar en un lugar fijo del sistema de ficheros y deberá tener el tamaño suficiente para contener todos los índices de los bloques del sistema de ficheros. Esta implementación tiene el inconveniente de que ocupa mucho espacio.

Espacio ocupado por el índice de un disco de 4 GB

Si tenemos un disco de 4 GB con bloques de 4 KB, necesitamos 20 bits para poder identificar cada uno de los bloques posibles. Si redondeamos el número de bits necesarios a un múltiplo de 8 (8 bits = 1 byte), necesitamos 3 bytes para contener al identificador de cada bloque. Por lo tanto, necesitaríamos 768 bloques de disco para contener la lista.

b) La lista encadenada de bloques libres es una alternativa que permite ahorrar espacio libre. Con esta estructura, el espacio de los índices forma parte del espacio libre y solo se debe tener de manera permanente la cabecera de la lista. El principal inconveniente de este sistema es el número de accesos al disco

que se necesitan para obtener espacio. En el caso de la lista de bloques libres, este coste se reducía, ya que se podía llevar a la memoria toda la lista o solo una parte; pero esto es imposible en este caso.

Espacio ocupado por la lista encadenada de un disco de 4 GB

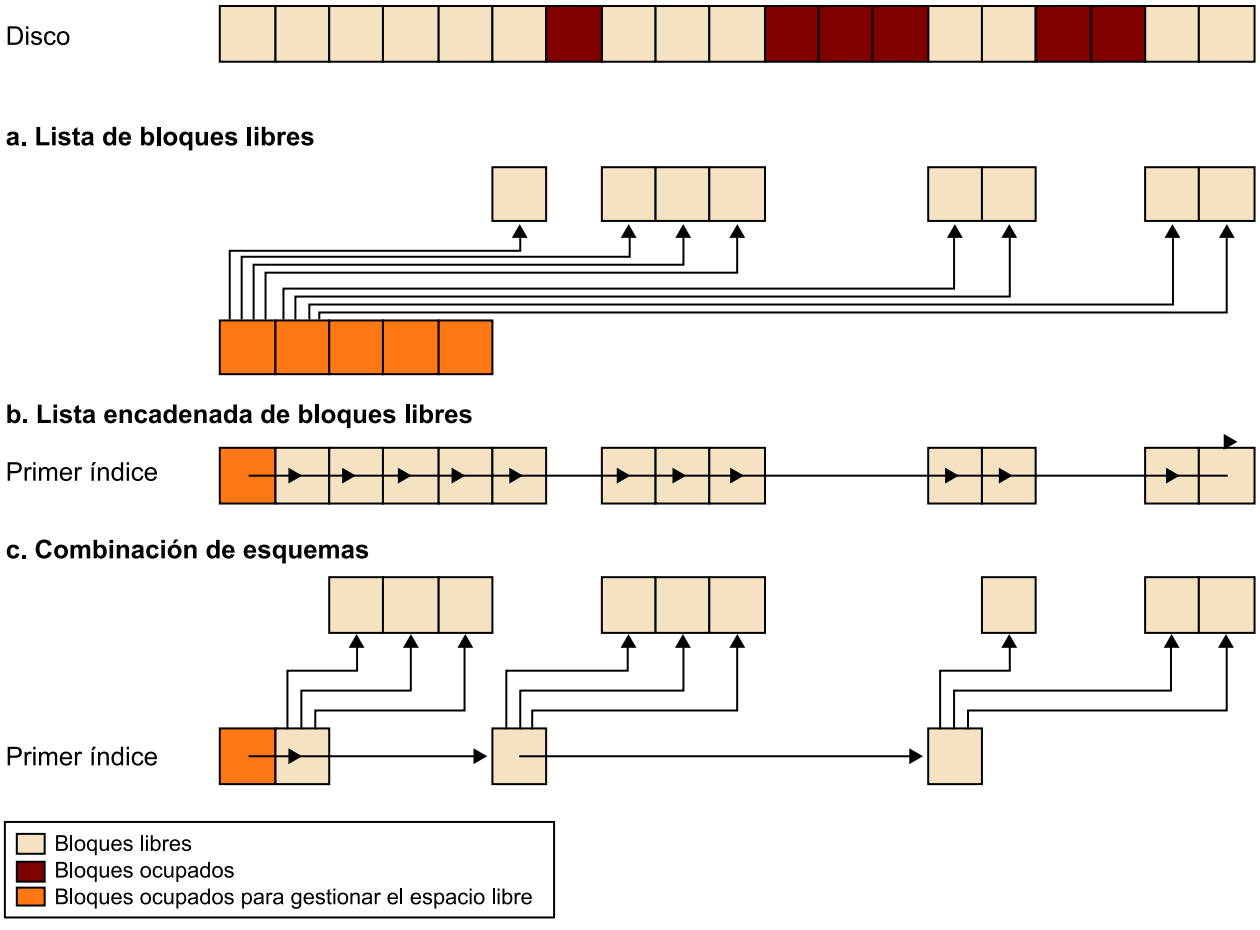
Si utilizáramos la lista encadenada como bloques libres para gestionar el espacio libre del disco, en el ejemplo anterior solo harían falta 6 bytes para guardar la información que hace referencia a los bloques libres: 3 para el puntero en el primer bloque y 3 para el contador de elementos de la lista.

c) Una tercera alternativa de implementación que es muy utilizada consiste en una combinación de los dos esquemas anteriores, aprovechando el coste reducido en accesos del primer esquema y el poco espacio necesario del segundo. Así pues, este esquema consiste en hacer una lista encadenada de bloques libres que contienen punteros en otros bloques libres. De esta manera, para obtener espacio libre solo se debe llevar a la memoria el primer bloque de la lista encadenada de bloques libres.

Espacio ocupado por la lista encadenada con punteros de un disco de 4 GB

En el caso anterior, por ejemplo, cada bloque libre de la lista encadenada contendría 1.364 punteros en bloques libres y un puntero en el bloque siguiente de la lista.

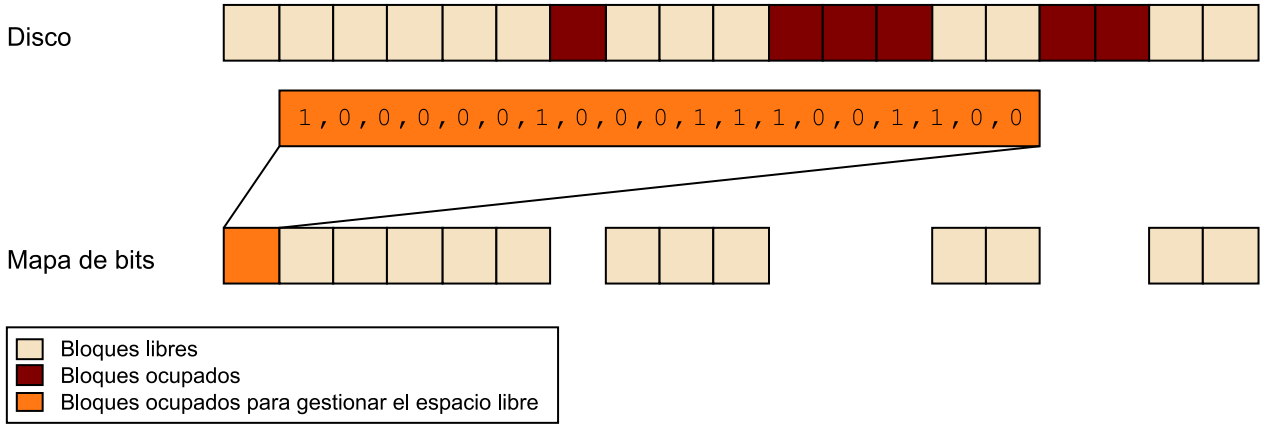
Figura 3. Representación del espacio libre con listas de bloques



2) El mapa de bits

También podemos representar el espacio libre del disco mediante un mapa de bits. El mapa contiene una lista de bits en la que cada bit corresponde a un bloque del disco. Si un bit vale 0, el bloque correspondiente está libre, y si vale 1, el bloque está ocupado.

Figura 4. Representación del espacio libre mediante un mapa de bits



Espacio de memoria ocupado por el mapa de bits de un disco de 4 GB

Siguiendo con el ejemplo de un disco de 4 GB con bloques de disco de 4 KB, tenemos un disco con 2^{20} bloques; por lo tanto, necesitaríamos 1 Mb para representarlos si utilizamos el mapa de bits, y eso ocupa 32 bloques de disco.

1.2.2. Los ficheros

Dentro del SF, los ficheros se representan mediante una estructura de datos que denominamos estructura del fichero, que tiene dos partes bien definidas:

- Una contiene información estadística y de protección, datos relacionados con el espacio de nombres, etc.
- La otra incluye información sobre los datos que almacena el fichero.

Nosotros nos centraremos en la segunda parte y desarrollaremos cuatro maneras de almacenar los datos en los ficheros.

1) El almacenamiento contiguo

El método del almacenamiento contiguo es el más sencillo para organizar la información contenida en un fichero. Un fichero con **estructura contigua** almacena la información en orden secuencial, sobre bloques del disco contiguos.

Almacenamiento contiguo

Si se utilizara almacenamiento contiguo, no sería necesario que el bloque de disco fuera la unidad de asignación de espacio; se podría utilizar el byte.

Así pues, para averiguar dónde se encuentra la información de un fichero con estas características, solo hay que saber cuántos bytes ocupa, cuántos bytes tiene un bloque y cuál es el primer bloque del fichero.

A la hora de crear este tipo de ficheros, es conveniente saber cuánto espacio de memoria ocuparán, ya que el SO deberá buscar un número de bloques libres consecutivos (que llamaremos hueco) lo suficientemente grande para contener el fichero y anotarse la longitud que ocupa en bytes. Análogamente, cuando un fichero quiere crecer y necesita más espacio del que tiene asignado, el SO lo deberá mover a un hueco que sea lo suficientemente grande para contenerlo. Esta organización facilita los accesos secuenciales y directos. En ambos casos, basta efectuar un acceso único para obtener los datos requeridos. Esta regularidad hace que sea una organización especialmente indicada para sistemas de tiempo real.

La asignación de bloques consecutivos de memoria implica los dos tipos de fragmentación de la memoria:

a) Fragmentación externa: un sistema de gestión de espacio (memoria, disco, etc.) tiene fragmentación externa si, al recibir una petición de espacio, no puede encontrar un hueco (espacio contiguo libre) lo bastante grande para satisfacerla, aunque haya trozos libres más pequeños y la suma de sus capacidades sea suficiente para atender la petición.

b) Fragmentación interna: un sistema de gestión de espacio (memoria, disco, etc.) tiene fragmentación interna si al recibir una petición de espacio no puede encontrar un hueco (espacio contiguo libre) lo bastante grande para satisfacerla, aunque la suma del espacio asignado no utilizado fuese suficiente para atender la petición.

Para encontrar un hueco de memoria suficientemente grande para ubicar un fichero, el SO puede recurrir a las tres políticas de asignación del espacio libre siguientes:

- *First-fit*: asigna el primer hueco que encuentra capaz de satisfacer la demanda recibida.
- *Best-fit*: asigna el hueco que se ajusta mejor a la petición que se atiende.
- *Worst-fit*: asigna el hueco más grande de todos los huecos libres.

Las tres opciones tienden a dejar huecos de memoria demasiado pequeños para ser aprovechados en ocasiones futuras (provocan fragmentación externa). Para aprovechar estos huecos, se debe hacer un proceso de compactación que reordene el espacio de la memoria y junte todos los bloques libres. Dada esta estructura y el modo como se debe buscar espacio, un posible método para gestionar el espacio libre es el mapa de bits. Una alternativa a la reordenación

Ved también

Las políticas de asignación de la memoria se tratan en el subapartado 2.1 del módulo didáctico "La gestión de la memoria" de la asignatura *Sistemas operativos*.

Hashing

La aleatorización, en inglés *hashing*, es una técnica orientada a hacer más ágil el acceso a la información mediante el contenido.

del espacio consiste en disponer de una estructura pensada especialmente para gestionar huecos, por ejemplo tener una cadena de huecos y una tabla de aleatorización indexada por el tamaño de los huecos.

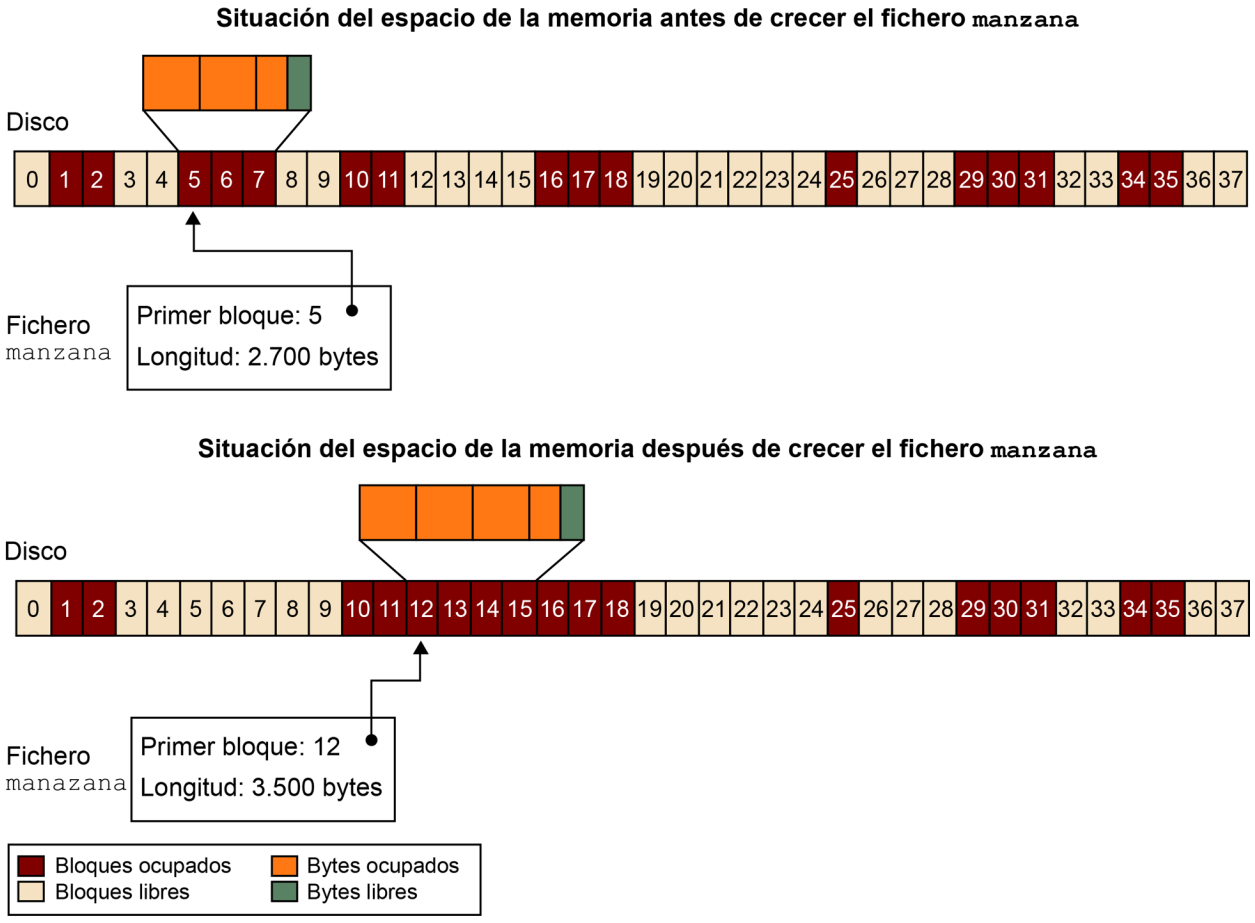
Acceso a un bloque de disco en un fichero estructurado de manera contigua

Supongamos que tenemos un fichero llamado *manzana* de 2.700 bytes de longitud guardado en un disco con bloques de 1 KB, lo que lo obliga a ocupar tres bloques de disco. Si queremos acceder de manera directa al tercer bloque del fichero *manzana*, el SO solo deberá ver cuál es el primer bloque que ocupa (en este caso el 5) y sumarle dos (5 + 2, ya que los bloques se numeran a partir de cero), y con esta dirección (bloque 7) podrá acceder al disco. Antes de acceder a él, sin embargo, deberá comprobar que no pedimos un bloque que esté más allá del final del fichero (el bloque 8 no pertenece al fichero), porque entonces el SO devolvería un error.

Para hacer un acceso secuencial, el SO solo deberá recordar cuál es el último bloque al que ha accedido (por ejemplo, el 6) y acceder al siguiente (el 7), habiendo hecho previamente las comprobaciones necesarias de final de fichero.

Si lo que se quiere es efectuar un acceso de escritura que haga crecer el fichero, el SO deberá buscar un hueco capaz de almacenar el fichero resultante, que ahora ocupa cuatro bloques de disco. Copiará el fichero en el hueco nuevo y liberará el espacio que ocupaba antes. Aquí, si aplicamos una política de *first-fit*, se pondrá en el hueco que empieza en el bloque 12. Observemos que si no estuvieran los huecos de los bloques 12 y del bloque 19, tendríamos fragmentación externa. Por otra parte, si observamos la longitud del fichero *manzana*, podemos ver que no ocupa la totalidad del último bloque. Como la unidad de asignación es el bloque, se le ha asignado un espacio que no utiliza en la totalidad y, por lo tanto, tenemos fragmentación interna.

Figura 5. Crecimiento de un fichero estructurado de manera contigua



Las **ventajas** de organizar los datos de los ficheros en bloques contiguos de la memoria son las siguientes:

- Facilita el acceso secuencial.
- Está especialmente indicado para ficheros estáticos o de longitud máxima conocida.
- Hace previsible el número de accesos al disco; por lo tanto, es apropiado para SO de tiempo real.

Los **inconvenientes** de este tipo de organización son los siguientes:

- Hay que conocer a priori el tamaño del fichero.
- Es complicado hacer crecer los ficheros.
- Fragmenta el espacio libre y requiere un proceso de compactación.

2) El encadenamiento de bloques

Una manera de solucionar los inconvenientes de la fragmentación externa consiste en configurar los ficheros como una **cadena** (lista ordenada) **de bloques**. Para averiguar dónde se encuentra la información de un fichero organizada con esta estructura, el SO solo debe saber cuáles son el primero y el último bloque del fichero (para facilitar el crecimiento de los ficheros hemos considerado que tenemos un puntero en el último bloque), cuántos bytes tiene un bloque y cuántos bytes almacena el fichero.

Con la estructura de bloques encadenados, cuando se crea un fichero no es necesario indicar qué tamaño tendrá, solo hay que asignarle un bloque libre y anotar su longitud en bytes. A medida que el fichero almacene más información y, por lo tanto, aumente su longitud, el SO actualizará la longitud e irá añadiendo, cuando convenga, bloques a la cadena sin tener que tener en cuenta su orden dentro del disco.

El acceso secuencial a los datos guardados en bloques encadenados se puede efectuar de manera sencilla, tal como se haría con una organización contigua. Pero el acceso directo requiere recorrer toda la cadena de bloques hasta llegar al dato deseado y, por lo tanto, hay que efectuar tantos accesos al disco como bloques deban recorrerse.

Desde el punto de vista de la gestión de los espacios libres y ocupados del disco, esta estructura es más eficiente que la asignación contigua, ya que permite eliminar la fragmentación externa que la organización contigua producía; por lo tanto, nos ahorra las operaciones de compactación. Desgraciadamente, ahora los bloques contienen, además de los datos que almacenan, los punteros en los bloques siguiente y anterior. Para facilitar la gestión del SO, sería

preferible separar estos dos tipos de información. De esta manera, se evitaría que acciones o errores en la gestión de uno de los dos afecten al conjunto. Por otra parte, el espacio libre se puede gestionar con la misma organización de lista encadenada, lo cual da más homogeneidad al sistema.

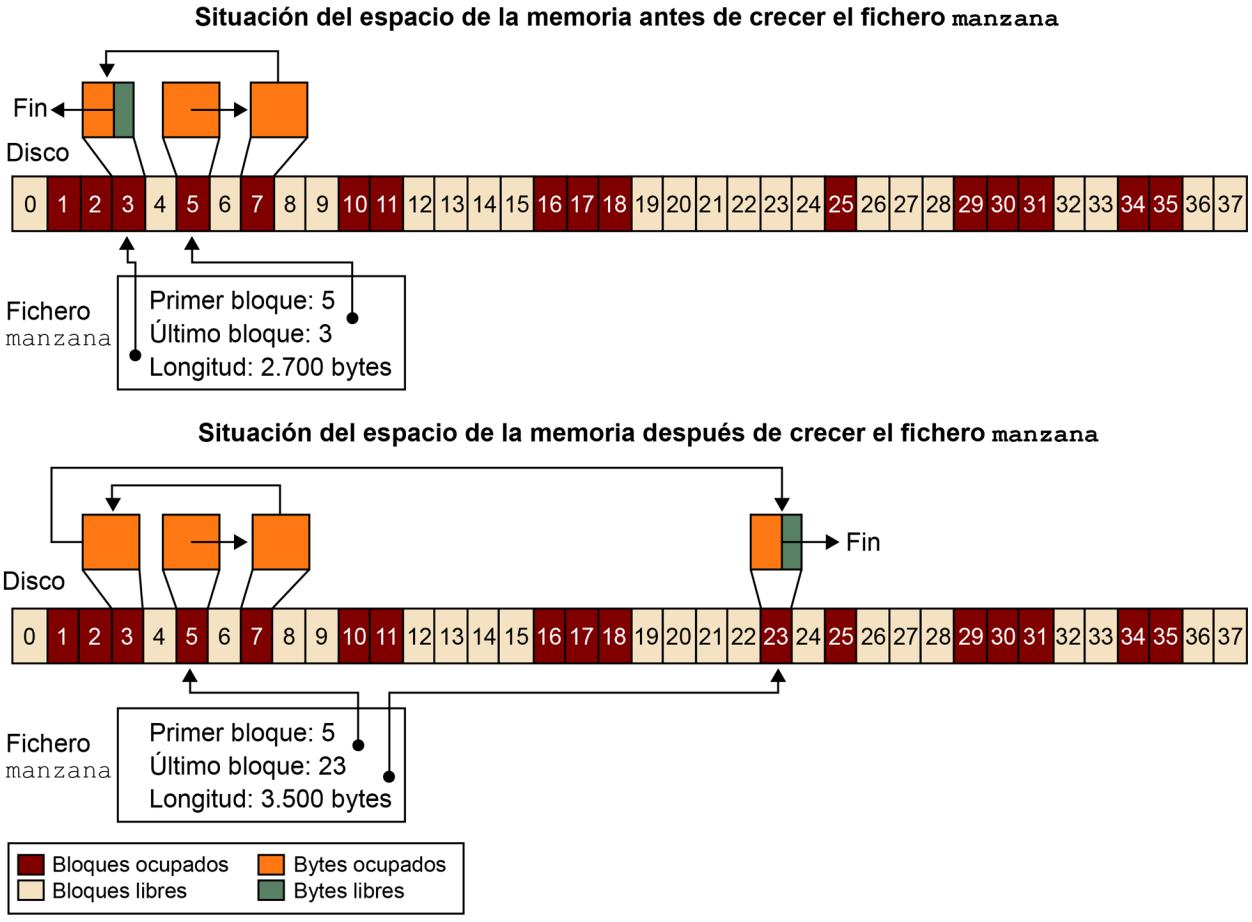
Acceso a un bloque de disco en un fichero estructurado con bloques encadenados

En este caso, el acceso directo al tercer bloque del fichero *manzana* implica recorrer toda la cadena de bloques; por lo tanto, el SO deberá leer 3 bloques: el 5, el 7 y el 3. Como siempre, deberá hacer las comprobaciones de longitud, pero, a diferencia de antes, no podremos acceder a datos de otros ficheros, ya que el último puntero ya es en sí una marca de final de fichero. No obstante, el SO debe comprobar que no se acceda a datos no inicializados del último bloque.

Para hacer un acceso secuencial, simplemente se debe seguir el puntero del último bloque de datos leído. En este caso, si el último bloque leído es el 7, el puntero nos lleva al bloque 3. El SO también deberá hacer las comprobaciones de final de fichero.

Para hacer crecer el fichero solo hay que localizar un bloque libre, por ejemplo el 23, y encadenarlo al final del fichero.

Figura 6. Crecimiento de un fichero estructurado con bloques encadenados



Organizar los datos de los ficheros con una estructura de bloques encadenados tiene las **ventajas** siguientes:

- Se elimina el problema de la fragmentación externa.
- No es costoso hacer crecer un fichero.
- Se facilita el acceso secuencial.

En cambio, organizar los datos de los ficheros con esta estructura implica los **inconvenientes** siguientes:

- Es muy costoso acceder directamente a una posición de un fichero.
- Se mezclan los contenidos con la estructura porque los bloques contienen datos y punteros.

3) La tabla de encadenamientos

La estructuración de los ficheros mediante tablas de encadenamientos es una alternativa de organización que intenta reunir ventajas de los dos sistemas de organización anteriores. Por una parte, elimina el problema de la fragmentación externa y las limitaciones al crecimiento de los ficheros, y, por otra, permite que las operaciones de acceso secuencial y directo se hagan con un número reducido de accesos al disco.

La **estructura con tabla de encadenamientos** mantiene una tabla con tantos punteros en bloques como bloques destinados a datos tiene el disco. Así pues, un fichero se representa como una cadena de entradas de esta tabla. El SO necesitará la tabla de encadenamientos y, como en la estructura de bloques encadenados, deberá saber cuáles son el primer y el último bloque del fichero, el número de bytes que tiene un bloque y el número de bytes que almacena el fichero.

Para crear un fichero o incrementar el contenido de los que ya existen, el SO debe llevar a cabo las mismas operaciones que en el caso del encadenamiento de bloques, con la única diferencia de que ahora los punteros no están en los bloques, sino en la tabla de encadenamientos. Para efectuar los accesos, tanto directos como secuenciales, el SO debe leer la tabla de encadenamientos del disco, y, una vez lo ha cargado en la memoria, puede encontrar el bloque que debe leer mediante el procedimiento adecuado (dependiendo de si el acceso es secuencial o directo). De esta manera, en el mejor de los casos, con dos accesos el sistema tiene suficiente para satisfacer la petición de acceso. Esta situación ideal puede degenerar hasta tener que hacer tantos accesos como en el sistema encadenado.

Si nos fijamos en el ejemplo "Espacio ocupado por el índice de un disco de 4 GB", vemos que necesitábamos 768 bloques para contener la tabla con todos los punteros. Como es evidente, no podemos llevar toda la tabla a la memoria para analizar con un solo acceso qué bloque es el que correspondería a la petición que se ha hecho al sistema operativo.

Ved también

Encontraréis el ejemplo "Espacio ocupado por el índice de un disco de 4 GB" en el subapartado 1.2.1 de este módulo didáctico.

Desde el punto de vista de la gestión del espacio del disco, este procedimiento tiene las mismas ventajas que el caso del encadenamiento de bloques. El espacio libre también se puede gestionar con la misma tabla de encadenamientos. Así se consigue que el sistema sea más homogéneo.

Acceso a un bloque de disco en un fichero estructurado con una tabla de encadenamientos

Para hacer un acceso directo al tercer bloque del fichero *manzana*, el SO deberá leer la tabla de encadenamientos y seguirla a la memoria. Por lo tanto, solo deberá hacer dos accesos al disco: uno para leer la tabla y otro para leer el bloque 3. Igual que en los ejemplos descritos, deberá comprobar que no se haga un acceso indebido.

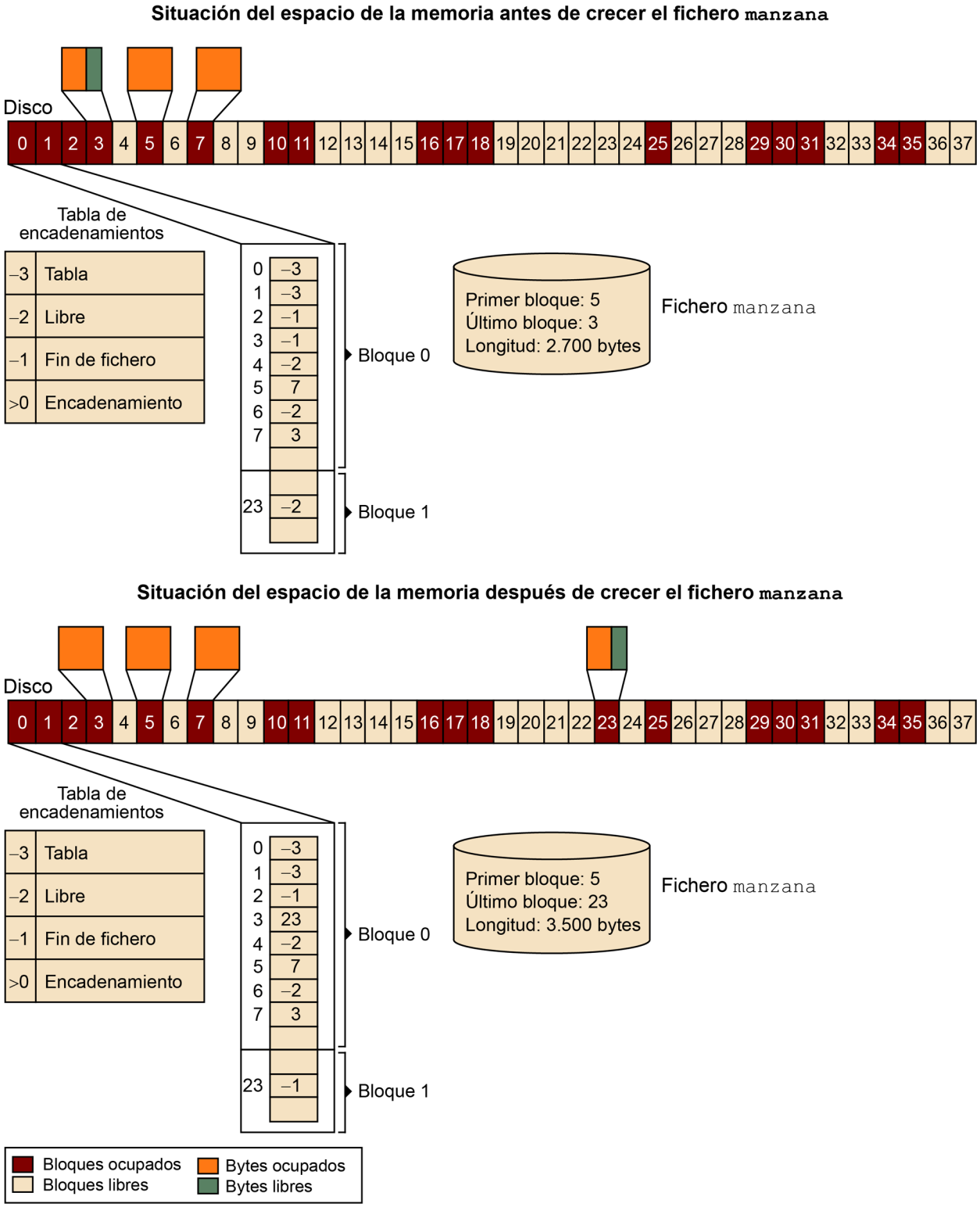
Con esta organización, los accesos secuenciales cuestan lo mismo que los directos. Si se ha accedido al segundo bloque del fichero y ahora hay que acceder al tercero, el SO lo deberá localizar utilizando el mismo procedimiento que en el caso anterior; por lo tanto, también necesitará dos accesos al disco.

La manera de incrementar la longitud de un fichero es muy similar a la del ejemplo precedente, con la única diferencia de que el encadenamiento se debe hacer sobre la tabla en lugar de hacerlo directamente en los bloques.

Observamos que la tabla de encadenamientos no cabe toda entera en un bloque, y que está dividida en dos. El bloque 23 se encuentra en el segundo bloque de la tabla.

Esta circunstancia en la realidad puede hacer incrementar el número de accesos necesarios para hacer un acceso secuencial o directo como los que hemos llevado a cabo antes.

Figura 7. Crecimiento de un fichero estructurado con una tabla de encadenamientos



Las **ventajas** de organizar los datos de los ficheros de la memoria con la tabla de encadenamientos son las siguientes:

- Se elimina el problema de la fragmentación externa.
- No es costoso hacer crecer un fichero.
- Se mantienen separadas la estructura del fichero y la de los datos.

El **inconveniente** de organizar los datos de los ficheros de la memoria con la tabla de encadenamientos es que los accesos secuencial y directo presentan variabilidad en el número de accesos al disco.

Ved también

En el subapartado 4.1 de este módulo podéis ver una descripción del sistema de ficheros FAT, que está implementado utilizando una tabla de encaminamiento.

4) La tabla de índices

Finalmente, presentamos una organización que, como la tabla de encadenamientos, elimina la fragmentación externa y al mismo tiempo intenta acotar el número de accesos a la memoria que se deben llevar a cabo para satisfacer cada petición de lectura/escritura.

La **organización con tabla de índices** consiste en tener, para cada fichero, una lista ordenada secuencialmente de los bloques que contienen los datos del fichero, con el fin de compactar en un mismo sitio toda la información referente a la localización del fichero. Así pues, el SO necesita saber cuál es la tabla de índices, cuántos bytes tiene un bloque y cuántos bytes almacena un fichero para poder acceder a él.

Para crear un fichero, el SO en primer lugar debe constituir la tabla de índices del fichero e incluir tantos bloques de datos como sea necesario y, en segundo lugar, debe anotar la longitud en bytes del fichero. Para hacer crecer el fichero, solo se requiere añadir los bloques de memoria necesarios a la tabla de índices y actualizar su longitud. El problema surge cuando se quiere crear un fichero que necesita más bloques de datos de los que puede referenciar la tabla de índices. Para solucionarlo, solo podemos hacer tres cosas:

a) Calcular cuál es el tamaño máximo que deben tener los ficheros y no permitir que lo superen.

b) Hacer que la tabla de índices permita crear ficheros tan grandes como la totalidad del espacio libre del dispositivo de almacenamiento.

c) Crear las tablas de índices con el número de punteros necesario para soportar la mayoría de los ficheros del sistema y, para aquellos que necesiten más espacio, hacer una cadena o lista de tablas de índices.

La segunda solución y la tercera permiten la creación de ficheros de cualquier longitud, pero la tercera es mejor, ya que permite dedicar a las tablas solo el espacio que necesitan realmente.

El número de accesos al disco necesarios para hacer un acceso secuencial o directo a un bloque de memoria queda reducido a dos: uno para leer la tabla de índices del fichero –ya que esta concentra los punteros en bloques de datos– y

otro para acceder a los datos propiamente dichos. El número de accesos solo se incrementará en caso de que sea necesario acceder a un fichero muy grande y se haya utilizado la tercera de las opciones que hemos descrito.

Desde el punto de vista de la gestión del espacio del disco, este sistema tiene las mismas ventajas que el encadenamiento de bloques y la tabla de encadenamientos. Como en estos dos procedimientos, el espacio libre se puede tratar con la misma organización de la tabla de índices y así el sistema es más homogéneo.

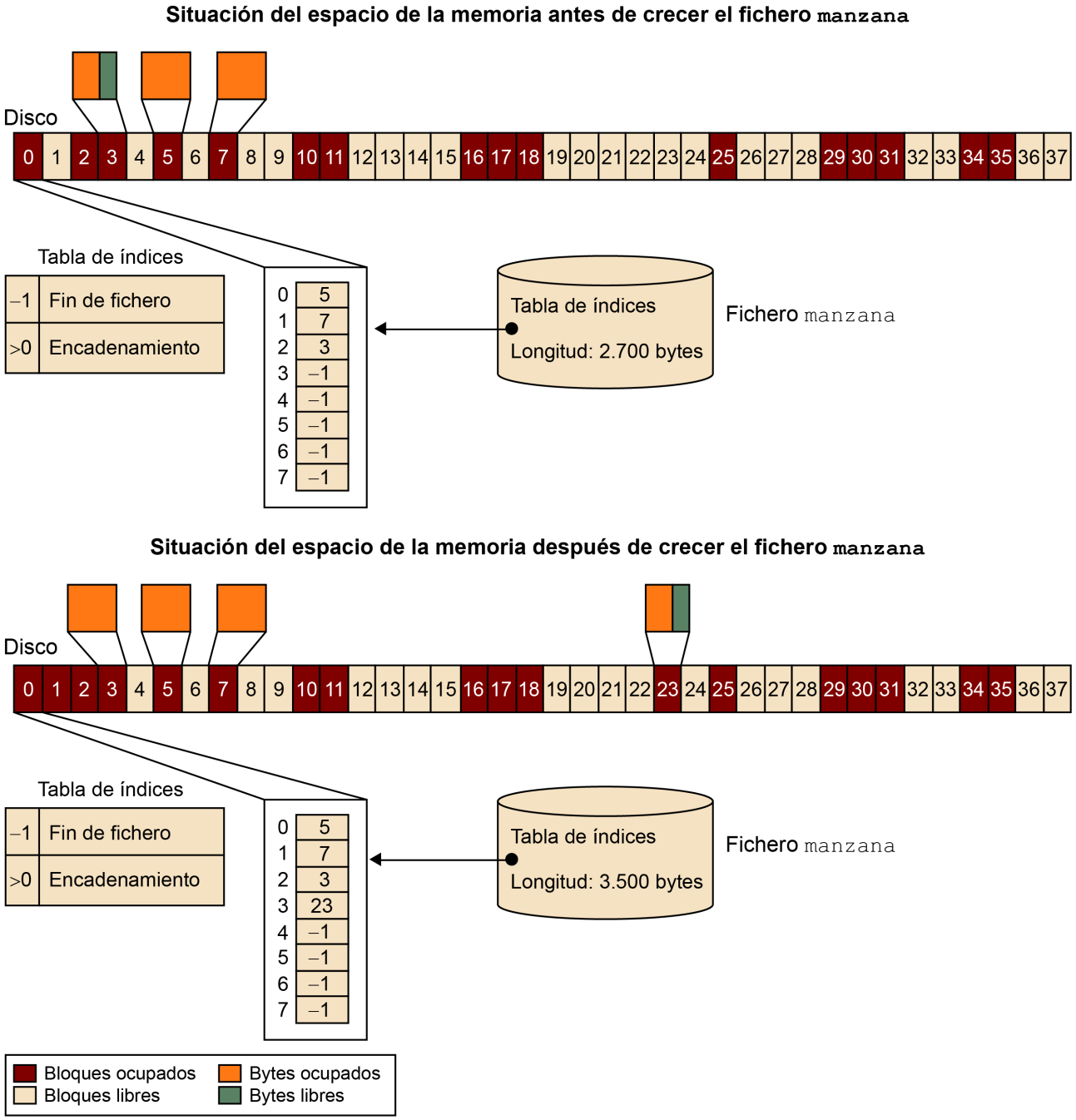
Acceso a un bloque de disco en un fichero estructurado con una tabla de índices

Para efectuar un acceso directo al tercer bloque del fichero *manzana*, el SO deberá leer la tabla de índices y seguirla a la memoria; por lo tanto, solo deberá hacer dos accesos al disco: uno para leer la tabla y otro para leer el bloque 3. La principal diferencia con la utilización de la tabla de encadenamientos es que esta solo lee la estructura del fichero *manzana* y, por lo tanto, tiene suficiente con un solo acceso.

Igual que en la estructura anterior, los accesos secuenciales tienen el mismo coste que los directos. Si se ha accedido al segundo bloque del fichero y ahora hay que acceder al tercero, el SO lo deberá localizar mediante el mismo procedimiento que en el caso anterior y, por lo tanto, también necesitará dos accesos al disco.

Para incrementar la longitud del fichero, se sigue un procedimiento muy similar al que se utiliza en la tabla de encadenamientos, pero ahora el bloque nuevo se anotará en la tabla de índices particular del fichero en lugar de anotarse en la tabla de encadenamientos. Con el fin de tener una representación clara en la figura 8, que ilustra esta implementación, hemos supuesto que hay una tabla de índices por bloque, pero en la realidad un mismo bloque puede contener más de una de estas tablas.

Figura 8. Crecimiento de un fichero estructurado con una tabla de índices



Organizar los datos de los ficheros de la memoria con la tabla de índices ofrece las **ventajas** siguientes:

- Se elimina el problema de la fragmentación externa.
- No es costoso hacer crecer un fichero.
- Se mantienen separadas la estructura del fichero y la de los datos.
- El número de accesos directos y secuenciales al disco es acotado.

El **inconveniente** de organizar los datos de los ficheros de la memoria con la tabla de índices es que los ficheros excesivamente largos hacen aumentar el espacio destinado a guardar la estructura y/o el número de accesos necesarios para acceder a los datos.

Ved también

En el subapartado 4.2 de este módulo podéis ver una descripción del sistema de ficheros EXT2, que está implementado utilizando tablas de índices.

1.2.3. Los directorios

Un **directorio** es el espacio de nombres que permite encontrar la estructura de datos que configura un fichero a partir de uno de sus nombres. El modo de encontrarlo es mediante unas tablas de conversión entre un nombre y la estructura del fichero.

Recordad que el mejor lugar para almacenar estas tablas está dentro de un fichero y que combinando los diferentes directorios podemos estructurarlos en diferentes formas, por ejemplo, en árbol o en grafo.

Para analizar los directorios desde la óptica de esta asignatura, nos centraremos en el estudio del directorio raíz y del contenido de los directorios. También recordaremos qué es un nombre simbólico y cómo se implementa dentro del SF.

1) El directorio raíz

De todos los directorios, el único que se trata de manera especial es el **directorio raíz**. Este directorio es la puerta de entrada al espacio de nombres y debe estar en un lugar conocido, normalmente en el primer bloque de datos del disco.

De esta manera, el SO sabe, dado el nombre de un fichero, por dónde debe iniciar su localización. Por este motivo, el directorio raíz se crea con el SF y no puede ser destruido aunque esté vacío.

2) El contenido de los directorios

Como hemos comentado, los directorios son tablas que, a partir de un nombre, nos dan acceso a la estructura del fichero. Para poder hacer esto, tenemos dos opciones:

- Guardar para cada nombre la estructura de datos que representa al fichero.
- Guardar para cada nombre un puntero en la estructura de datos que representa al fichero.

La primera alternativa tiene el inconveniente de que no permite la estructura de directorios en forma de grafo, ya que cada fichero solo puede tener un único nombre. Este inconveniente lo soluciona la segunda alternativa, dado que independiza las estructuras de datos que configuran el espacio de nombres (los directorios) de las estructuras que configuran los ficheros. En esta segunda alternativa hay que añadir a la estructura de datos que configura el fichero un campo que indique cuántos nombres (enlaces físicos[5]) o, lo que es lo mismo, cuántos punteros hay sobre este fichero. Cuando el valor de este campo es igual a cero, el fichero se puede eliminar, ya que este valor indica que no hay ningún nombre con el que se pueda acceder a él.

(5)En inglés, *hard links*.

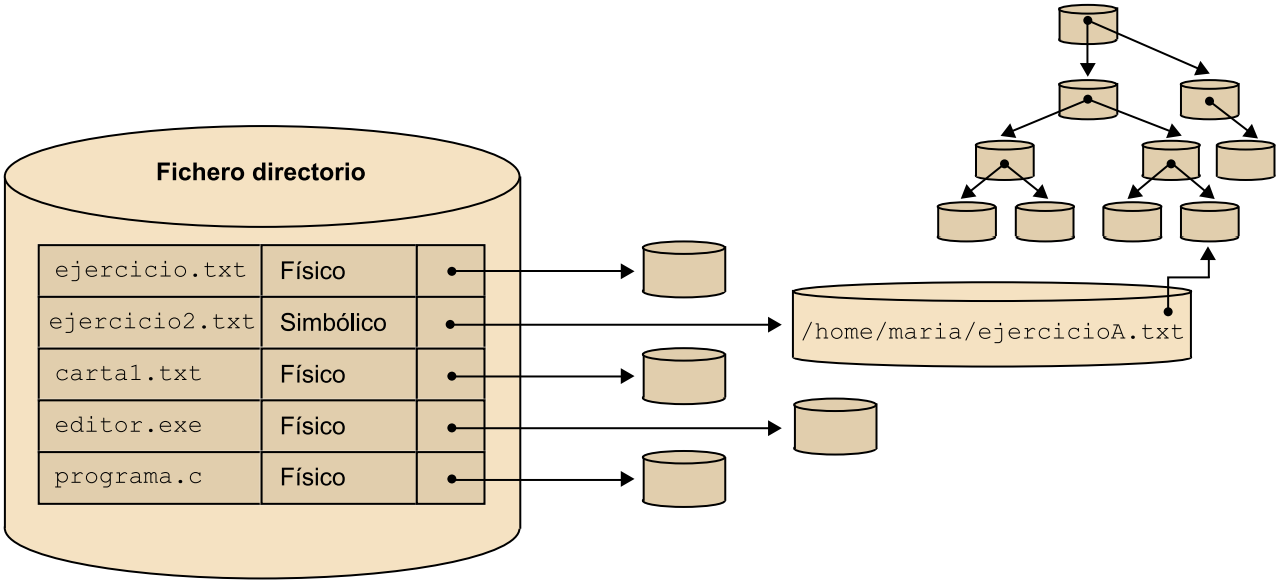
3) Los nombres simbólicos

Los **nombres simbólicos** (enlaces simbólicos[6]) relacionan un nombre con un fichero mediante el nombre de otro fichero, en lugar de hacerlo directamente con su estructura de datos.

(6)En inglés, *soft links*.

Para conseguirlo, se guarda el enlace simbólico dentro de un fichero intermedio y se marca, en la estructura de datos del fichero, que se trata de un fichero especial que contiene un nombre simbólico para poder tener en cuenta esta circunstancia a la hora de abrirlo.

Figura 9. Enlace simbólico



Del mismo modo que hemos marcado los ficheros simbólicos, para que el SO sepa lo que debe hacer con cada fichero, también los deberemos marcar según el tipo (directorios, ficheros normales, etc.).

2. Visión dinámica del sistema de ficheros

En este apartado analizaremos las estructuras de datos que el SO necesita para manipular el SF. Las operaciones necesarias para hacer esta manipulación se pueden clasificar en los tres grupos siguientes:

a) Operaciones sobre los ficheros, como leer o escribir.

b) Operaciones sobre el árbol de directorios, como cambiar de directorio de trabajo.

c) Operaciones propias de control del SF, como crear un SF nuevo.

2.1. Operaciones sobre los ficheros

Empezaremos por las operaciones de manipulación de los ficheros. En este caso, nos referiremos a las operaciones dependientes del dispositivo fichero y dejaremos de lado las operaciones independientes. En concreto, trataremos las operaciones siguientes:

- *Dispositivo = abrir(nombre,operación_modo_acceso)*
- *Estado = cerrar(dispositivo)*
- *Estado = posicionar(dispositivo,posición)*
- *Estado = leer(dispositivo,buffer,contador)*
- *Estado = escribir(dispositivo,buffer,contador)*

La información necesaria para llevar a cabo estas operaciones está relacionada con los procesos que pedirán las operaciones de acceso y con la misma estructura del SF.

Ved también

La información que podemos necesitar dentro del SO para caracterizar las entradas/salidas asociadas a un proceso desde un punto de vista genérico se ve en la asignatura *Sistemas operativos*.

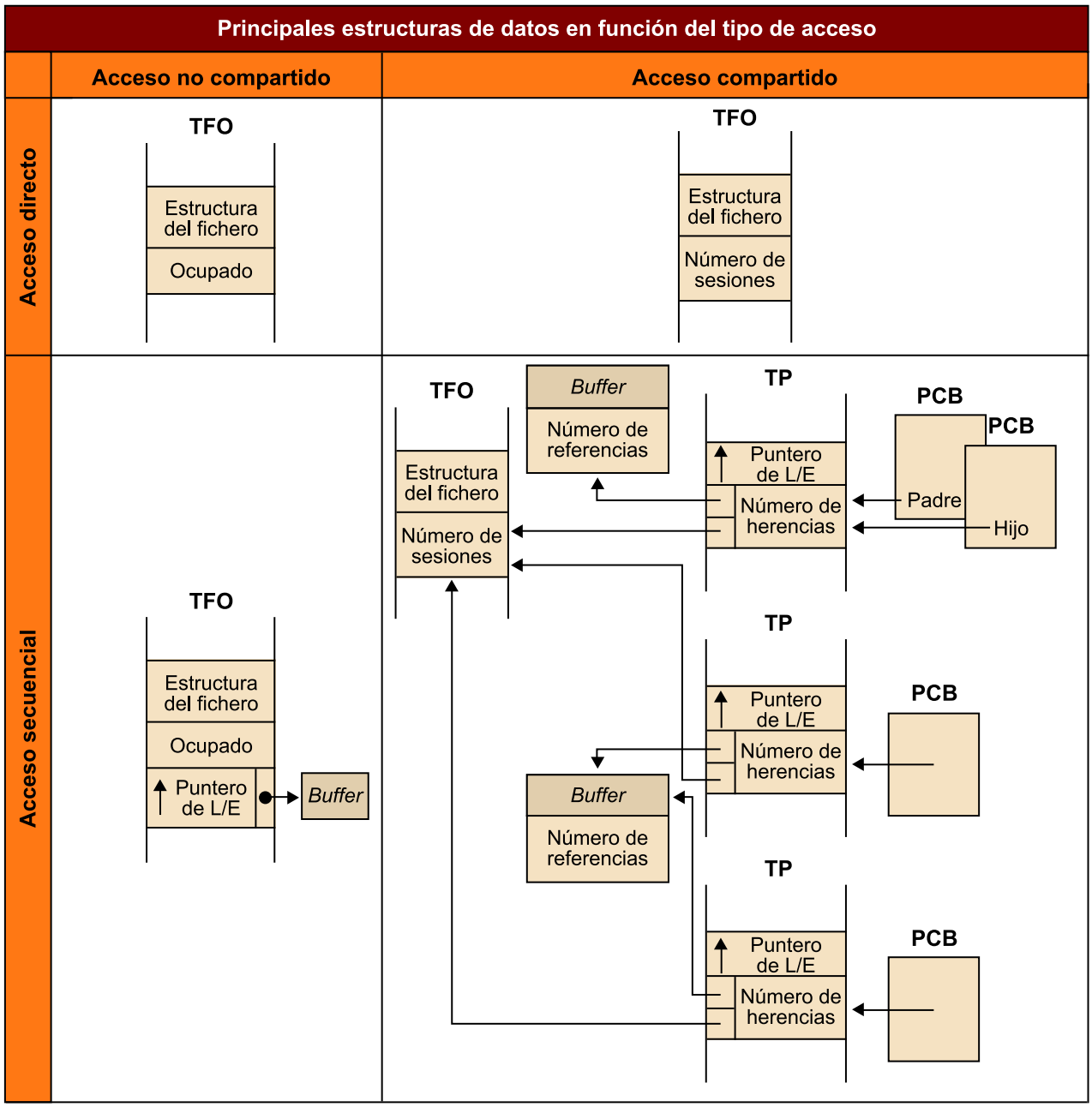
En este subapartado, ampliaremos primero esta información en función de las características de los accesos que permite hacer el SO. Después, y con el fin de reducir los tiempos de acceso a la información residente en el dispositivo de almacenamiento, veremos qué información referente a la estructura de los ficheros y qué datos del fichero se duplicarán en la memoria, siguiendo el principio de localidad en el que se basan las memorias caché. Las dos variantes de la localidad son las siguientes:

a) Localidad temporal: las posiciones de memoria, disco, datos, instrucciones o, en general, cualquier objeto se dice que tienen localidad temporal cuando es muy probable que en los instantes que siguen al acceso a un objeto se vuelva a acceder a él.

b) Localidad espacial: las posiciones de memoria, disco, datos, instrucciones o, en general, cualquier objeto se dice que tienen localidad espacial cuando es muy probable que en los instantes que siguen al acceso a un objeto se acceda a los objetos próximos a este objeto.

Así pues, clasificaremos los accesos a los ficheros en función de dos características: si son secuenciales o directos, y si son compartidos o no.

Figura 10. Principales estructuras de datos en función del tipo de acceso



1) El **acceso secuencial**. Como ya sabemos, es un acceso que sigue la secuencia de datos del fichero: al iniciar los accesos se accede a la primera posición del fichero y en los accesos siguientes, a las posiciones consecutivas a la última posición a la que se ha accedido.

2) El **acceso directo**. Permite acceder al contenido de un fichero en función de su posición y de manera independiente de los accesos anteriores; esto obliga a especificar la posición en cada acceso (las operaciones de lectura/escritura tienen un parámetro adicional para especificarlo).

3) El **acceso compartido**. Permite que haya más de una sesión de trabajo abierta simultáneamente, y, por lo tanto, se pueden efectuar varios accesos al mismo tiempo sobre un mismo fichero.

4) Un **acceso no compartido**. No permite que haya más de una sesión al mismo tiempo sobre un mismo fichero; por lo tanto, solo puede haber un acceso al fichero por unidad de tiempo.

Con estas cuatro características hemos dibujado el cuadro de posibilidades de la figura 10 y que pasaremos a detallar a continuación.

2.1.1. Acceso directo no compartido

Un SO que permita solo como método de acceso básico el acceso directo no compartido necesita conocer la estructura del fichero al que debe acceder. No se tiene que preocupar de mantener una exclusión mutua sobre los datos, de evitar los accesos simultáneos, ni tampoco ha de recordar a qué posición del fichero ha accedido en la última operación de acceso secuencial.

Si los accesos a los ficheros estuvieran aislados, no haría falta guardar permanentemente ninguna información dentro del SO. No obstante, sabemos que los accesos se organizan en sesiones de trabajo sobre un determinado fichero. Por lo tanto, el acceso a la estructura de datos que representa un mismo fichero se deberá repetir varias veces (localidad temporal). Por este motivo, es aconsejable guardar una copia de esta estructura la primera vez que se accede a ella. La tabla donde se guardan las copias de las estructuras de los ficheros abiertos se denomina tabla de ficheros abiertos (TFA[7]).

(7)TFA es la sigla de *tabla de ficheros abiertos*.

Como tenemos dos copias de la misma estructura de datos, una en el disco y otra en la memoria, se necesita un mecanismo para mantener la coherencia entre estas dos copias. Para hacerlo, tenemos dos técnicas procedentes de la teoría de memorias caché:

- ***Copy back***: técnica que consiste en actualizar la memoria principal con las modificaciones que se han hecho sobre una línea de la memoria caché únicamente cuando la memoria caché debe ser reemplazada.

- ***Write through***: técnica que consiste en actualizar la memoria principal con las modificaciones que se hacen sobre la memoria caché en el mismo instante en el que se introducen.

En el caso del SF, la primera opción consiste en actualizar la información del disco una vez que ya no se necesita más en la memoria. La segunda actualiza la información del disco cada vez que se modifica en la memoria. Evidentemente, la primera operación reduce el número de accesos al disco y la segunda minimiza el riesgo de dejar el disco en un estado incoherente si se produce un fallo. Las dos técnicas son aceptables, y será más adecuada una u otra en función de las características del entorno de cada SO.

Por lo tanto, las operaciones de manipulación del fichero quedarán como sigue:

1) Operación abrir: a partir del nombre del disco, localiza el lugar donde se encuentra la estructura de datos que representa al fichero. Para buscarlo, el SO utiliza la función *localizar_objeto*. Una vez ha localizado el lugar y ha comprobado que existe, la estructura reserva una entrada libre en la TFA[8] y copia la estructura del fichero del disco. Cuando ha finalizado la copia, el fichero ya está abierto. Si la operación de apertura implica la creación del fichero, antes de nada deberá crear la estructura del fichero en el disco y darle un nombre mediante la operación *crear_nombre* de manipulación de directorios. Entonces ya puede proceder como hemos explicado al principio del párrafo.

[8]TFA es la sigla de *tabla de ficheros abiertos*.

Ved también

Podéis ver la función *localizar_objeto* en el subapartado 2.2 de este módulo didáctico.

2) Operación cerrar: libera la entrada asociada al fichero que hay en la TFA. Antes, si la información que contiene ha sido actualizada en la memoria, deberá volcarla en el disco para mantener la coherencia de las dos copias.

3) Operación posicionar: esta operación no tiene sentido con un acceso directo.

4) Operaciones leer/escribir: el SO debe localizar el bloque del disco al que debe acceder. Para hacerlo, utiliza la estructura de datos que está asociada a la entrada de la TFA. Es evidente que esta descripción es muy simple y en un caso real es posible que se tenga que repetir la operación más de una vez. Además, las acciones que hay que efectuar son ligeramente diferentes en función de si la operación es de lectura o de escritura.

2.1.2. Acceso directo compartido

Si tenemos un SO que ofrece como operaciones básicas el acceso directo y la compartición de ficheros, la estructura que utiliza para hacer los accesos directos y no compartidos es insuficiente.

Incoherencia en la información de un fichero compartido con acceso directo

Supongamos que un proceso abre un determinado fichero para hacer una operación de escritura. Al hacerlo, el sistema localiza el fichero dentro del disco, busca una entrada libre en la TFA y copia la estructura de datos que representa

este fichero. A partir de este momento, ya puede iniciar las operaciones de lectura sobre el fichero. Al mismo tiempo, otro fichero también lo abre para escribir. El SO efectúa las mismas operaciones que antes, le busca una entrada libre en la TFA y copia la estructura de datos que representa el fichero. Entonces tenemos dos procesos que modifican el contenido del fichero sobre copias separadas de su estructura. Las dos estructuras evolucionarán de manera incoherente. Una vez que acaben la sesión de trabajo, si tenemos una política de escritura de *copy back*, las modificaciones que haya hecho el proceso que cierre primero se perderán, ya que serán sobrescritas cuando el segundo proceso cierre el fichero.

Es evidente que es necesario algún mecanismo que garantice la coherencia entre la información que hay en la memoria y la que utilizan diferentes sesiones de trabajo sobre un mismo fichero. La solución más sencilla consiste en permitir que haya solo una copia en la memoria de la estructura de datos de cada fichero y que sea compartida, mediante exclusión mutua, para todos los procesos que la necesitan. Esto implica modificar las entradas de la TFA y los procesos de abrir y cerrar.

En la TFA añadiremos un contador para cada entrada que indique cuántas sesiones de acceso al fichero utilizan aquella entrada. Si el contador está a cero, quiere decir que la entrada está libre.

Con respecto a las operaciones, tenemos lo siguiente:

1) Operación abrir: como en el caso del acceso directo no compartido, la operación, a partir del nombre del fichero, debe localizar el lugar del disco donde se encuentra la estructura de datos que este fichero representa. Una vez localizada y comprobada su existencia, a diferencia de antes, mira que esta estructura no esté en la TFA. En el caso de que no esté, procede como en la operación anterior e inicializa el contador de sesiones a uno. Si ya se encuentra en la TFA, incrementa el contador de sesiones que hay sobre este fichero. Recordemos que todos los accesos a una entrada de la TFA se deben hacer en exclusión mutua.

2) Operación cerrar: decrementa el contador de sesiones. Si este es mayor que cero, el SO dará la operación por finalizada. Por el contrario, si el contador llega a cero, se procederá como en el caso precedente.

3) Operación posicionar: no tiene sentido con un acceso directo.

4) Operaciones leer/escribir: se hacen de la misma manera que en el caso del acceso directo no compartido, con la condición de que cada acceso utilice una memoria intermedia propia y, tal como hemos dicho, cualquier acceso a la TFA se lleve a cabo en exclusión mutua.

2.1.3. Acceso secuencial no compartido

Como hemos visto, en los accesos directos no hay que guardar información histórica de lo que ha sucedido en el último acceso de lectura/escritura. Ahora, al hacer los accesos secuencialmente, aparece esta necesidad: hay que saber dónde se ha quedado el puntero de lectura/escritura en la última operación. Para solucionarlo, añadiremos este puntero a cada entrada de la TFA. El puntero se inicializará durante el proceso de apertura de una sesión de trabajo y se actualizará en cada acceso.

Por otra parte, en los casos de acceso directo, cada vez que se hacía una operación de lectura/escritura hemos supuesto que el SO iba al disco para acceder a los datos. Hemos hecho esta suposición porque en el acceso directo no es evidente que tenga un comportamiento con localidad. En cambio, en los accesos secuenciales se da claramente un comportamiento con localidad espacial, ya que después de acceder a un dato, se accederá a los datos próximos. Además, las demandas de información por parte de las operaciones de lectura/escritura suelen ser inferiores a un bloque.

Ved también

La localidad espacial se trata en el subapartado 2.2 de este módulo didáctico.

Si tenemos en cuenta las dos informaciones anteriores, podemos mejorar los procedimientos de lectura y escritura añadiendo al SO unas memorias intermedias que guarden los bloques que se han leído recientemente y haciendo que sean referenciadas desde la TFA. Así, cada vez que la información ya se encuentre en la memoria intermedia el SO se podrá ahorrar el acceso al disco.

Las operaciones quedarán modificadas de la manera siguiente:

1) Operación abrir: es idéntica a la que hemos descrito para el acceso directo no compartido, con la única diferencia de que una vez se ha abierto el fichero, antes de devolver el control al usuario, el SO debe inicializar el puntero de lectura/escritura. Típicamente, este puntero se inicializa a 0; una excepción se daría si abrimos el fichero en un modo destinado a añadir información al final de fichero (en el caso de Unix, sería el modo *O_APPEND*), en este caso lo inicializaría con el número de caracteres que contiene el fichero.

2) Operación cerrar: es idéntica a la operación cerrar del caso del acceso directo no compartido.

3) Operación posicionar: con estas características del SF la operación de posicionar sí que tiene sentido. Cuando se invoca esta operación, el SO deberá modificar el puntero de lectura/escritura actualizándolo con el valor que se indique. Lo único que debe hacer es velar para que no apunte más allá del final del fichero.

4) Operaciones leer/escribir: en el acceso secuencial, estas operaciones se diferencian, con respecto a las que efectuábamos cuando teníamos accesos directos, en el hecho de que no siempre requerirán un acceso al disco. Si la infor-

mación que se pide ya se encuentra en la memoria intermedia, el SO se ahorrará un acceso al disco. Para saberlo, el SO consulta el puntero de lectura/escritura y la parte de la memoria intermedia que indica qué contiene. Cuando ha acabado la operación de acceso, el SO actualiza el puntero de lectura/escritura.

Como ya hemos señalado en los casos de acceso directo, es evidente que esta descripción es muy simple. Un caso real, en cambio, puede suponer tener que repetir esta operación más de una vez. Las acciones que hay que llevar a cabo también son ligeramente diferentes en función de si la operación es una lectura o una escritura. Finalmente, y a causa de la utilización de memorias intermedias, también puede ser diferente según el tipo de política que se utilice para mantener la coherencia con la copia del disco.

2.1.4. Acceso secuencial compartido

Si a la situación planteada por los accesos secuenciales añadimos el hecho de tener accesos de más de una sesión de trabajo sobre el mismo fichero, surge el problema de la coherencia de la información. Podemos observar que esta situación da lugar a tres problemas que requieren alguna intervención para ser solucionados:

Ved también

La solución a los problemas que plantea la repetición de la estructura de ficheros en la TFA se trata en el subapartado 2.1.2 de este módulo didáctico.

a) El primer problema de coherencia surge si tenemos la estructura de los ficheros más de una vez dentro de la TFA. Este problema tiene la misma solución que en el caso del acceso directo.

b) El segundo problema está relacionado con el uso de memorias intermedias para almacenar los bloques de datos de los ficheros. Sobre estos datos también hay que mantener la coherencia entre las copias de la memoria y las del disco y, si hay más de una copia en la memoria, también deberíamos mantener la coherencia entre estas copias. La solución a los dos problemas consiste en permitir una única copia en la memoria de cada bloque y asegurar una política correcta de escritura en el disco. Del mismo modo que lo hemos hecho para solucionar los problemas derivados de la repetición de la estructura del SF en la TFA, incorporaremos en cada bloque un contador de referencias que nos permitirá saber cuántas referencias se hacen a una memoria intermedia y cuándo deja de ser utilizada.

c) El tercer problema aparece por el hecho de que este esquema de acceso al disco permite que haya herencia entre procesos, es decir, que haya padres e hijos. Esto se traduce en el hecho de que los punteros de lectura/escritura son compartidos por varios procesos. Por este motivo sacaremos los punteros de la TFA y crearemos una tabla específica para ellos, la tabla de punteros (TP). Pondremos un contador en cada puntero que nos indicará cuántos procesos lo utilizan y si puede ser eliminado o no.

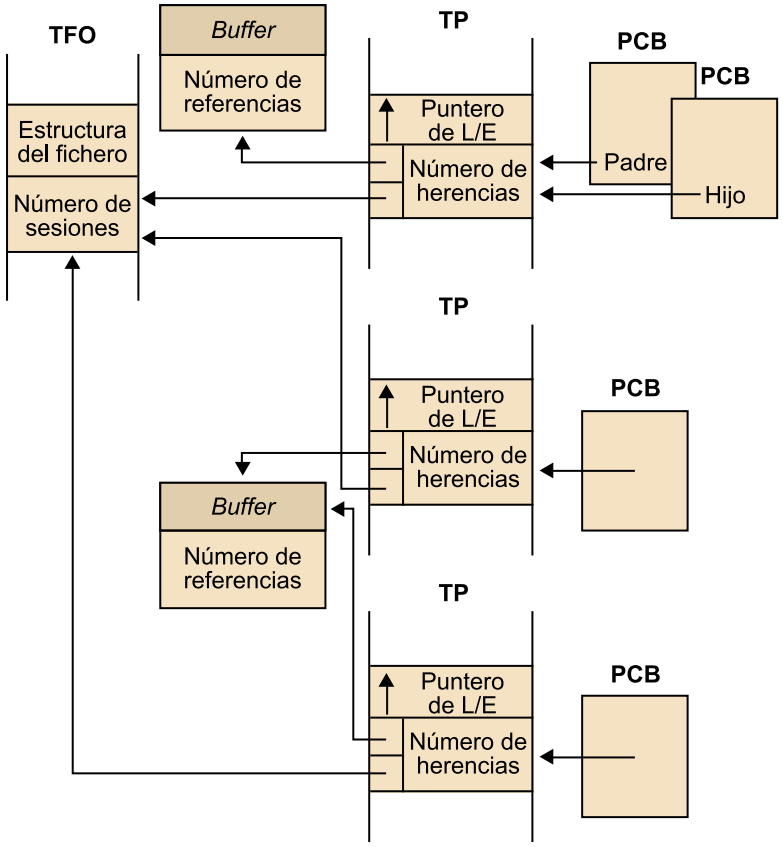
En la figura 11 podemos ver diferentes situaciones:

- En la parte superior de la figura hay dos procesos, padre e hijo, que comparten el mismo puntero de lectura/escritura sobre una sesión de trabajo.

- En la parte de abajo tenemos dos procesos que han abierto dos sesiones independientes sobre el mismo fichero, y que acceden de manera independiente al mismo bloque de datos del fichero.

- Observamos que el primer grupo de procesos (padre e hijo) no acceden al mismo bloque de datos que el segundo grupo de procesos.

Figura 11. Estructuras de datos necesarios en el caso de acceso secuencial compartido



Así pues, los procedimientos de acceso se modifican tal como se indica a continuación:

1) Operación abrir: en este caso, actuará de manera similar al caso de acceso directo compartido, con la diferencia de que cuando hayan finalizado todas los operaciones con la entrada correspondiente de la TFA, el SO deberá buscar una entrada libre en la tabla de punteros (TP), inicializar el puntero como lo haría si tuviera un acceso no compartido, inicializar el contador de herencias a uno y, finalmente, hacer que esta entrada apunte a la entrada de la TFA correspondiente. El SO no asociará ninguna memoria intermedia a la entrada de la TP[9], ya que lo hará cuando convenga durante las operaciones de acceso.

[9]TP es la sigla de tabla de punteros.

2) Operación cerrar: como en la operación abrir, el funcionamiento de la operación cerrar es muy similar al caso del acceso directo compartido. La diferencia radica en el tratamiento que hace del puntero y de la memoria intermedia asociada. Con respecto a la entrada de la TP, decrementa el contador de herencias, y si es cero, libera la entrada de la TP. Con respecto a la memoria intermedia, dependerá de si contiene información que no haya sido actualizada en el disco y de si es utilizada por algún otro proceso. En todo caso, aplicará una política para mantener coherente el SF.

3) Operación posicionar: es idéntica al caso secuencial no compartido.

4) Operaciones leer/escribir: las operaciones de acceso también son idénticas al caso anterior, excepto por el hecho de que el SO debe acceder a las estructuras de datos en exclusión mutua porque son compartidas y de que la asignación de la memoria intermedia se debe hacer de una manera global, teniendo en cuenta que, por motivos de coherencia, en la memoria solo puede haber una copia de cada bloque del disco.

2.2. Operaciones sobre los directorios

En este subapartado trataremos las operaciones relacionadas con los directorios. Podemos resumir estas operaciones de la manera siguiente:

- *Estado* = *modificar_nombre*(*nombre_nuevo, nombre_viejo*)
- *Estado* = *crear_nombre*(*directorio, nombre_nuevo, simbolo_fisico,objeto*)
- *Estado* = *destruir_nombre*(*nombre*)
- *Estado* = *cambio_de_directorio*(*nuevo_directorio*)
- *Valores* = *ver_nombres*(*directorio*)
- *Dir_actual* = *directorio_actual*( )
- *Localizar_objeto*, que forma parte del resto de las operaciones.

Con el fin de dar una descripción tan completa como sea posible de las operaciones anteriores, supongamos que tenemos un sistema con espacio de directorios estructurado en forma de grafo que admite los nombres simbólicos. Al describir estas operaciones separaremos las operaciones que crean, modifican o consultan el contenido de los directorios, de las operaciones que manipulan el concepto de directorio de trabajo y, finalmente, distinguiremos la operación que, dado un nombre, localiza un fichero.

2.2.1. Operaciones de creación, modificación, consulta y destrucción de un directorio

Antes de ver cómo se gestionan las operaciones de creación, modificación y destrucción de directorios, recordemos que un directorio es un fichero con una estructura especial que puede reconocer el SO. Por lo tanto, todas estas operaciones son la combinación de las operaciones posibles sobre los ficheros: abrir, leer y/o escribir y cerrar.

1) Operación de creación: esta operación consiste en crear un fichero e indicar que es de tipo directorio. Una vez que se tiene el fichero, se procederá a crear en su interior la estructura de datos propia de los directorios activando los punteros en el mismo fichero y en su directorio padre.

2) Operación de destrucción: esta operación consiste en destruir el fichero que contiene el directorio, habiendo tomado previamente la precaución de eliminar o comprobar que se hayan eliminado todos los nombres que contenía.

3) Operación de acceso: tanto para modificar como para consultar un directorio, el SO accede como lo haría a un fichero normal, con la diferencia de que los datos que contiene los manipula directamente. Por lo tanto, lo abrirá, lo leerá y/o lo escribirá y finalmente lo cerrará. Por ejemplo, para llevar a cabo las operaciones de acceso de crear o destruir un nombre, el SO solo debe llenar o vaciar una de las entradas que configuran el directorio. A continuación, se debe actualizar el campo que indica el número de nombres que tiene el fichero y que se encuentra en su estructura en el disco. Si este número llegara a cero, el SO destruiría el fichero. Por lo tanto, la creación o destrucción de un nombre también puede traer asociada la creación o destrucción del fichero al que hace referencia.

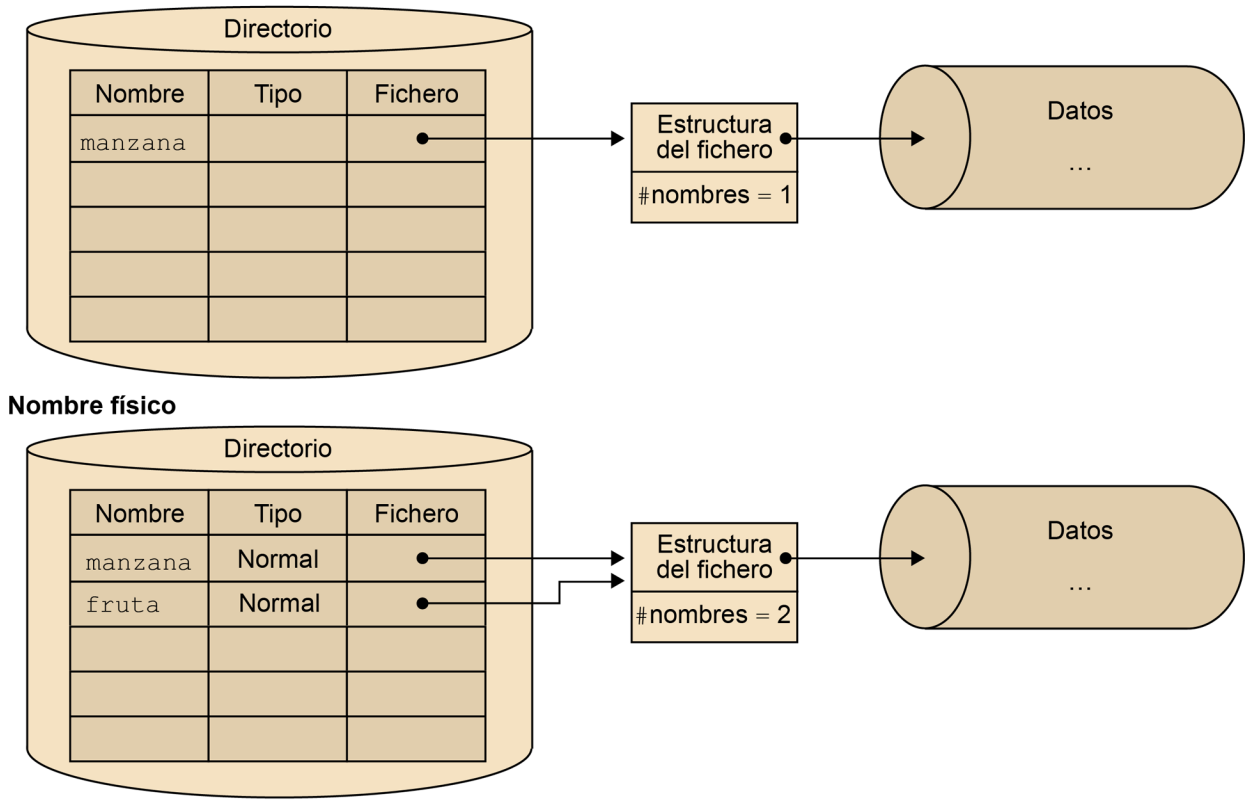
Creación de un nombre nuevo para un fichero

En la figura 12 representamos la creación de un nombre físico nuevo para el fichero *manzana*:

Ved también

La operación de localización de un fichero se trata en el subapartado 2.2.3 de este módulo didáctico.

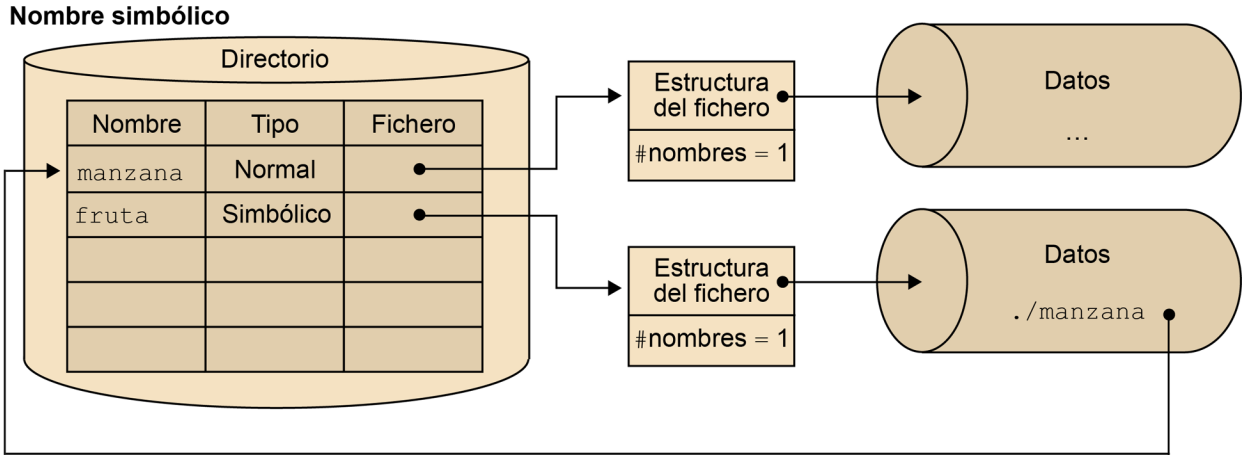
Figura 12. Creación de un nuevo nombre físico (enlace físico) para un fichero



Si se quiere crear un nombre físico nuevo del fichero *manzana* (por ejemplo, *fruta*), se debe localizar la estructura del fichero *manzana*, incrementar el campo de la estructura de este fichero donde se indica cuántos nombres tiene y, finalmente, actualizar el directorio con el nombre nuevo *fruta* y el puntero en la estructura del fichero.

Siguiendo con el ejemplo (figura 13), si se quiere añadir un nombre simbólico, se debe crear un fichero llamado *fruta* de tipo nombre simbólico que contenga como datos de fichero el nombre del fichero al que quiere apuntar, en este caso, *manzana*.

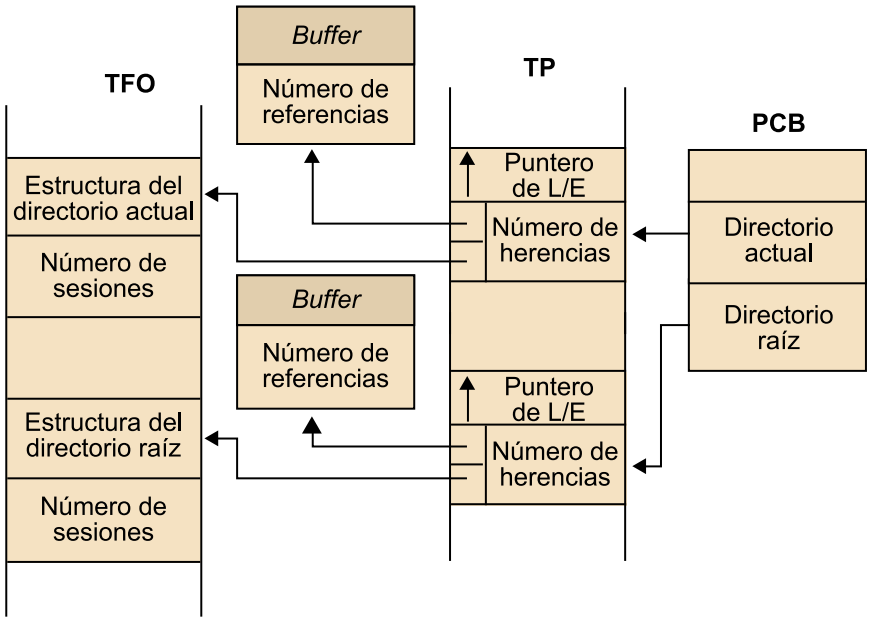
Figura 13. Creación de un nuevo nombre simbólico (enlace simbólico) para un fichero



2.2.2. Directorio de trabajo

Antes de tratar las operaciones relacionadas con el directorio de trabajo, veremos cómo se representan el directorio de trabajo y el directorio raíz dentro del SO (figura 14):

Figura 14. Localización del directorio de trabajo y del directorio raíz



Ambos son directorios sobre los cuales los procesos inician las operaciones de localización de ficheros, ya sea para crearlos, abrirlos, destruirlos, etc. Por lo tanto, el sistema los tiene abiertos para poder acceder rápidamente a sus datos. Además, el directorio de trabajo es uno de los elementos que configuran el entorno de los procesos; por lo tanto, en el bloque de control de procesos (PCB) deberemos incluir un campo que haga referencia a ellos. Si seguimos este razonamiento, para cada proceso tendremos un método para localizar el directorio raíz.

Referencia

Una referencia al directorio de trabajo puede ser, por ejemplo, un puntero en la TP que, al mismo tiempo, hace referencia a una posición de la TFA donde está la estructura de datos del fichero directorio actual.

La operación de cambio de directorio de trabajo debe cerrar el directorio actual que tenga el proceso y abrir uno nuevo siguiendo los mismos pasos que haría para abrir un fichero normal.

2.2.3. Localización de un fichero

La localización de un fichero es una operación interna que utiliza el SO en muchas otras funciones. Esta operación localiza la estructura de un fichero a partir de uno de sus nombres, los cuales pueden ser tanto absolutos como relativos, físicos como simbólicos. Para los nombres absolutos, la operación partirá del directorio raíz y, en el caso de los relativos, del directorio de trabajo. Recordemos que los dos directorios se encuentran abiertos y a punto para acceder a ellos.

Para hacer la operación de localización, el SO divide los nombres en componentes, cada uno de los cuales hace referencia a uno de los directorios que debe visitar durante la localización del fichero. En una iteración para cada componente del nombre, el SO leerá el directorio correspondiente, comparará sus nombres con el componente del nombre que debe buscar y, en caso de encon-

Error de acceso

Un error de acceso se produce, por ejemplo, cuando no existe uno de los componentes del nombre, o cuando un componente intermedio no es un directorio.

trarlo y no ser el último componente, lo abrirá y volverá a empezar la iteración. Saldrá de esta iteración cuando se produzca un error de acceso o cuando llegue al último componente y, por lo tanto, haya localizado el fichero.

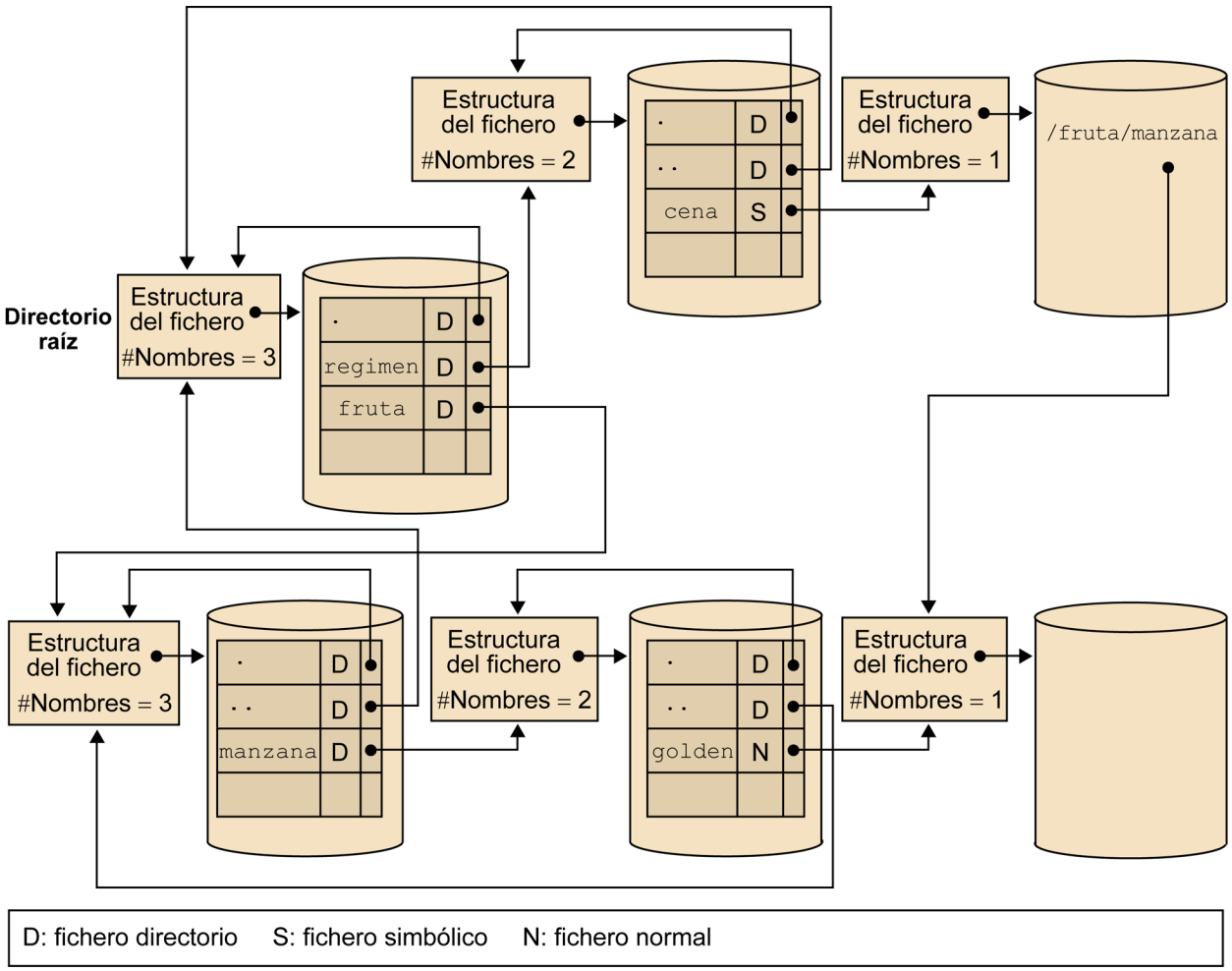
Para abrir el fichero y seguir el camino que marca el nombre del fichero que se debe buscar por el grafo de nombres, el algoritmo debe tener en cuenta el tipo de ficheros que va encontrando. En la descripción de la operación de localización hemos supuesto que todos los componentes del nombre eran ficheros de tipo directorio. Cuando algún componente no lo sea, se deberá aplicar un algoritmo apropiado. Por ejemplo, podría ser un fichero de tipo nombre simbólico. Entonces, se debería abrir el fichero, leer el camino que contiene y concatenarlo con los componentes del nombre que se busca y que todavía no se han resuelto para continuar la operación.

Localización de un fichero con un nombre simbólico

Supongamos que queremos localizar el fichero */regimen/cena/golden*, donde el fichero *cena* es un nombre simbólico de */fruta/manzana*.

Al llegar al componente *cena*, veremos que es un nombre simbólico, leeremos el fichero que contiene el camino del nombre simbólico y crearemos la ruta nueva concatenando */fruta/manzana* a *golden*. Esto da lugar a una localización nueva con el camino */fruta/manzana/golden*.

Figura 15. Localización de un fichero con un nombre simbólico



2.3. Operaciones sobre el sistema de ficheros

Las operaciones sobre el sistema de ficheros se centran en la creación y la activación de un SF:

- *Estado = crear_SF(nombre,tipo...)*
- *Estado = montar_SF(dispositivo,tipo,modoL/E...)*
- *Estado = desmontar_SF(dispositivo)*
- *Estado = verificar_SF(dispositivo, tipo)*

2.3.1. Creación de un sistema de ficheros

La operación de crear un SF es una operación que necesita acceder directamente al dispositivo, independientemente de la estructura que pueda tener antes de efectuar la operación. Durante la operación de creación de un SF se reescribe todo el disco: se crea el bloque de control del SF, donde se guarda la información estadística de quién ha creado el SF, cuándo, etc. Esta operación la lleva a cabo una utilidad externa al SO que se encuentra en el espacio de usuario y que debe tener privilegios para poder acceder al disco y reescribirlo de arriba abajo.

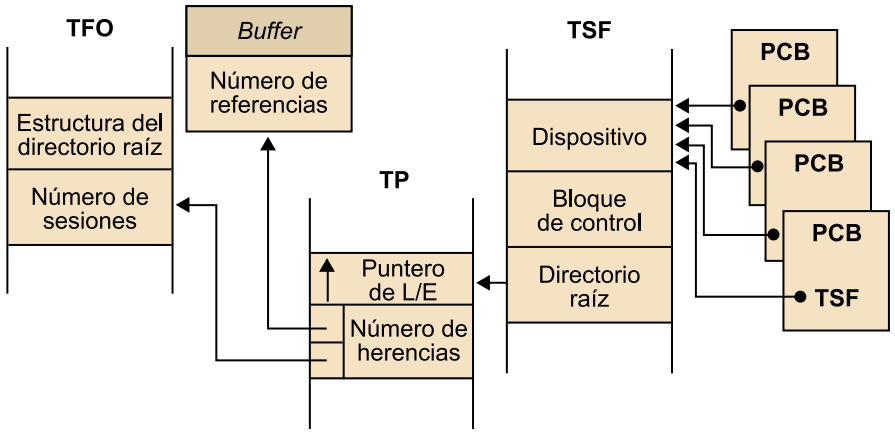
Ved también

La información que recoge el bloque de control del sistema de ficheros se trata en el subapartado 1.2 de este módulo didáctico.

2.3.2. Activación de un sistema de ficheros

Hacer que un SF sea reconocido por el SO y sea visible para los usuarios requiere una serie de operaciones que en algunos sistemas (por ejemplo, en Linux) son explícitas y en otros (por ejemplo, en los sistemas Windows), implícitas. En cualquier caso, el sistema, cuando es informado (o cuando detecta) que hay un SF nuevo, debe guardar una serie de informaciones del SF nuevo en la memoria y debe actualizar algunas de sus estructuras de datos con el fin de agilizar las operaciones futuras de manipulación de ficheros y directorios.

Figura 16. Estructura de datos necesaria para montar un SF



Las operaciones necesarias para activar y desactivar un SF son las dos que mencionamos a continuación:

1) Operación montar. Para hacer esta operación, el SO debe llenar una entrada de la tabla del SF (TSF), que es donde guarda, para cada SF montado, la información del dispositivo donde se encuentra y su bloque de control (qué espacio libre tiene, y otras informaciones de protección y estadísticas). Una vez cargada la tabla, debe abrir el directorio raíz y guardar en la TSF una referencia a su entrada en la TP. De esta manera, el sistema queda preparado para que los procesos puedan acceder al SF nuevo.

Nota

Debemos observar que el caso descrito corresponde a un sistema del estilo del MS DOS o Windows. En el sistema Linux, el procedimiento variaría ligeramente porque dispone de un único árbol de directorios para todos los SF montados y, por lo tanto, en tiempo de montaje se debería enlazar el SF nuevo a uno existente.

2) Operación desmontar. Desmontar un SF consiste en deshacer las operaciones que hemos descrito en la operación montar, pero ahora el SO debe garantizar la coherencia del bloque de control que ha guardado en la TSF y la coherencia del directorio raíz con respecto al disco. Cuando sea necesario, deberá actualizar estas informaciones en el disco.

Otra operación que ofrecen algunos SF es **verificar la coherencia** de sus estructuras de datos. Como esta operación está relacionada con la fiabilidad del SF, será explicada más adelante.

Ved también

La fiabilidad del sistema de ficheros se explica en el subapardado 3.2 de este módulo didáctico.

3. El rendimiento y la fiabilidad del sistema de ficheros

3.1. El rendimiento del sistema de ficheros

A continuación, describiremos algunas técnicas que se aplican para mejorar el rendimiento de los SF. En concreto, describiremos una que está relacionada con la geometría del disco, otra que tiene que ver con los procedimientos de acceso al disco y otra relacionada con la gestión de las peticiones de espacio libre para nuevos ficheros o para aumentar el tamaño de los ficheros.

3.1.1. El *interleaving* y el *skew*

A la hora de acceder al disco, el retraso que se produce desde el momento en el que se hace la petición al periférico hasta que este indica que la operación ha finalizado depende de la suma del tiempo que se tarda en transferir la información más el tiempo que pasa hasta que la información está en disposición de ser leída por el cabezal del disco. Este último tiempo se puede subdividir, a su vez, en el tiempo de rotación del disco y el tiempo de desplazamiento del cabezal hasta la pista a la que se debe acceder. De media, una vez movido el brazo del disco, el sistema se deberá esperar el tiempo que el disco tarda en dar media vuelta para tener la información bajo el cabezal.

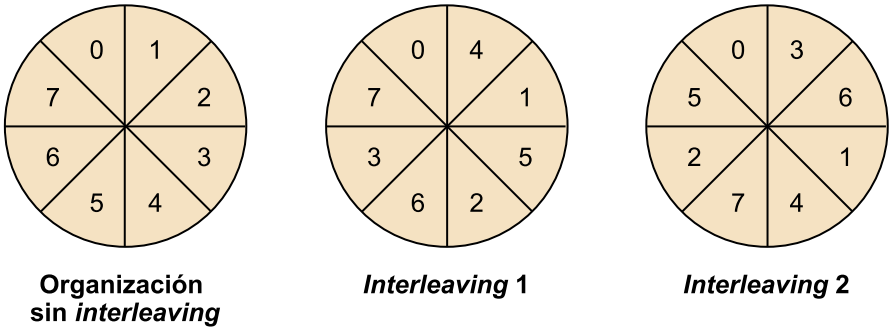
Teniendo en cuenta que a menudo un bloque agrupa más de un sector y que, por consiguiente, siempre se accederá a todos al mismo tiempo, podemos pensar que una ordenación de los sectores más eficiente que la secuencial puede reducir significativamente el tiempo medio de rotación que debe esperar el sistema una vez movido el brazo. Las dos técnicas que permiten hacer esta ordenación son el *interleaving* y el *skew*.

Antes de entrar en detalles, ilustraremos con un ejemplo las dos causas que producen los retrasos que hemos mencionado y los factores que intervienen. Supongamos que tenemos un sistema donde los bloques están formados por dos sectores consecutivos del disco situados en la misma pista. Por lo tanto, siempre leeremos uno inmediatamente después del otro. Una vez leído el primer sector, el controlador lo debe copiar en la memoria. Durante el tiempo que se tarda en hacer la copia, el disco sigue girando y cuando el sistema da la orden de leer el sector siguiente, este ya no se encuentra bajo el cabezal, y esto obliga al SO a esperar a que el disco complete la vuelta y se vuelva a posicionar en el sector deseado. El tiempo que dura la rotación necesaria para hacer el segundo acceso se perderá sistemáticamente, ya que la unidad de transferencia del SO es el bloque.

De la misma manera, si los dos sectores de un bloque pueden estar en pistas diferentes, tendremos que contar cuánto gira el disco durante el tiempo que se tarda en mover el cabezal de una pista a la siguiente.

La técnica del *interleaving* está pensada para solucionar los tiempos de espera derivados de las rotaciones excesivas del disco. Para eliminar el tiempo de espera, basta con numerar los sectores según el orden lógico en el que los verá el SO. Este orden se piensa de manera que el tiempo de transferencia de un sector a la memoria sea igual al tiempo que tarda el disco en girar y presentar el sector lógico siguiente del bloque bajo el cabezal de lectura/escritura. En la figura 17 podemos ver un disco sin *interleaving* y dos discos con *interleavings* diferentes.

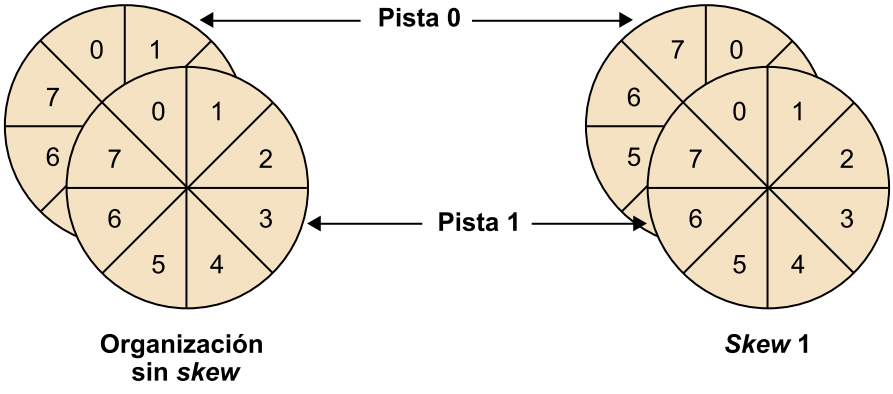
Figura 17. *Interleaving*



La técnica del *skew* está pensada para solucionar el mismo problema cuando los sectores de un bloque se encuentran en pistas diferentes. Así, el segundo sector de un bloque estará desplazado con respecto al primero en función del tiempo que el disco necesite para mover el brazo y de su velocidad de rotación. La figura 18 muestra dos inclinaciones[10] diferentes entre pistas.

(10)En inglés, *skews*.

Figura 18. *Skew*



3.1.2. La memoria caché de disco

La idea de utilizar memorias caché, es decir, memorias más rápidas, pero más pequeñas, como depósito temporal para reducir el tiempo de acceso en entornos con mucha localidad, se puede exportar a multitud de situaciones. Una de estas situaciones es la gestión del SF.

Ved también

La localidad se trata en el subapartado 2.1 de este módulo didáctico.

Los accesos a los ficheros tienen un comportamiento marcadamente local. Los accesos secuenciales se comportan con localidad espacial. Por otra parte, los accesos a las estructuras de control del fichero y las pautas de acceso a los datos que algunos algoritmos generan, principalmente de acceso directo, tienen un comportamiento con localidad temporal. Por lo tanto, el comportamiento de los accesos al SF permite aplicar la técnica de la memoria caché. Por motivos de eficiencia, el SO mantendrá en memoria una copia de las estructuras de datos del SF relativas a los ficheros que están siendo utilizados.

El principal inconveniente de la utilización de la memoria caché es que se debe mantener la coherencia entre la información que hay en el disco y la que hay en la memoria, pero ya hemos visto varias técnicas que nos permiten conseguirlo.

Ved también

Los métodos que permiten mantener la coherencia de la información se tratan en el subapartado 2.1.1 de este módulo didáctico.

También se debe resolver el problema de los discos extraíbles. En estos casos, el SO debe ser informado cuando se introduce un disco en el sistema o cuando se extrae con el fin de mantener la coherencia de la información de la memoria caché. Por ejemplo, un dispositivo de almacenamiento USB no debería ser extraído del puerto hasta que lo hubiéramos solicitado al SO y hubiéramos recibido la confirmación de que ya lo podemos extraer. Otro ejemplo lo tenemos con los CD/DVD en las instalaciones Linux; el reproductor puede expulsar el disco únicamente cuando el disco ha sido desmontado. En cambio, en el caso de los disquetes flexibles en MSDOS/Windows, el SO actualiza el contenido del disquete inmediatamente; por lo tanto, podemos extraer el disquete en cuanto el piloto muestre que la unidad de disco está inactiva.

3.1.3. Fragmentación interna

Todos los sistemas de almacenamiento que dividen el espacio en porciones de tamaño fijo y donde esta porción se convierte en la unidad de asignación de espacio presentan el problema de la fragmentación interna. Estos sistemas desperdician una cierta cantidad de almacenamiento en todos los objetos de tamaño no múltiple del tamaño de las porciones.

Un modo de reducir la fragmentación interna sería hacer que el tamaño de estas porciones fuera lo más pequeño posible, pero resulta inviable porque esta solución hace aumentar de manera sustancial el tamaño de las estructuras de datos de gestión. Además, por motivos de eficiencia, el tamaño de estas porciones cada vez es mayor.

Una posible solución consiste en permitir que, en algunos casos particulares, estas porciones no sean la unidad mínima de asignación. Esta técnica recibe el nombre de *sub-block allocation*.

3.1.4. Fragmentación de los datos en el SF

Además de la fragmentación interna y de la fragmentación externa, en un SF podemos considerar otro tipo de fragmentación, la fragmentación de los datos. Asumiendo un SF que divida el espacio de almacenamiento en bloques, la fragmentación de los datos aparece cuando los bloques asignados a un fichero están diseminados por todo el disco. Esto implica que para hacer una lectura completa del fichero haya que mover el cabezal del disco sucesivas veces. Como la operación de mover el cabezal del disco es costosa, este hecho puede incrementar de manera sustancial el tiempo necesario para leer el fichero.

Para resolver este problema existen varias técnicas:

- El SF puede utilizar una política de asignación de bloques libres que tenga en cuenta la proximidad entre los bloques. Para mejorar la efectividad de estas políticas, sería bueno conocer a priori cuál será el tamaño final del fichero.

- También es posible prerreservar bloques libres para los ficheros que sabemos que deben aumentar de tamaño. Estos bloques prerreservados deberían estar próximos a los bloques ya asignados al fichero.

- Existen herramientas que intentan desfragmentar el SF mediante la reubicación de los ficheros excesivamente fragmentados.

3.2. La fiabilidad del sistema de ficheros

A continuación, describiremos cuatro técnicas que se aplican para mejorar la fiabilidad de los SF:

a) Las copias de seguridad. Están pensadas para poder recuperar la información si el disco se convierte en ilegible o si el usuario borra accidentalmente un fichero.

b) La verificación del sistema de ficheros. Permite detectar incoherencias en las estructuras de datos del SF.

c) *Journaling*. Permite reducir la probabilidad de pérdida de información en caso de que se produzca una interrupción del suministro eléctrico.

d) La redundancia. Tiene como objetivo básico no tener que parar el sistema aunque se produzca un error de hardware en el disco.

3.2.1. Las copias de seguridad

Las copias de seguridad[11] almacenan la información del SF tal como estaba en un instante de tiempo. El objetivo de las copias de seguridad es poder recuperar la información del SF en caso de error del usuario (borrar accidentalmente un fichero, necesitar una versión previa del fichero, etc.) o que el disco se convierta en ilegible (sea por avería de hardware, inundación, incendio, etc.). En caso de tener que acceder a la información de la copia de seguridad, se recuperará el estado de los ficheros en el instante de tiempo en el que se realizó la copia de seguridad y se habrán perdido todas las modificaciones posteriores.

(11)En inglés, *back-ups*.

En todos los SF habría que plantearse la conveniencia de definir una política de copias de seguridad. En función del entorno concreto donde se utilizan los datos del SF, la política debe dar respuesta adecuada a preguntas como:

- ¿Ficheros involucrados? Hay que decidir si se hace copia de seguridad de todo el SF o únicamente de los ficheros de las zonas de usuario o únicamente de los que se encuentren en determinados directorios...

- ¿Cantidad de información? Total o incremental (solo guarda la información modificada desde que se hizo la última copia de seguridad). Hay que tener en cuenta que una copia de seguridad incremental puede aumentar el tiempo necesario para recuperar el fichero.

- ¿Estático o dinámico? Dinámicamente mientras el sistema está trabajando, o de manera estática, cuando no existe la posibilidad de que ningún usuario pueda modificar los ficheros.

- ¿Frecuencia? Diario/semanal/mensual...

- ¿Número de copias de seguridad que se quieren conservar? Una única copia de seguridad, las *N* últimas, los 5 últimas diarias y las 3 últimas mensuales...

- ¿Dispositivo de almacenamiento? Habría que considerar la fiabilidad de los tipos de dispositivo sobre los que se hará la copia de seguridad, el tiempo de vida que debería tener la copia de seguridad y si se continuará disponiendo de los aparatos necesarios para leer/escribir sobre estos dispositivos. Puede suponer la definición de alguna política de conversión de formatos para almacenar las copias de seguridad.

- ¿Ubicación geográfica? Para protegernos ante acontecimientos catastróficos, no tiene demasiado sentido conservar todas las copias de seguridad cerca del SF original. Habría que conservar alguna copia de seguridad razonablemente lejos para minimizar las opciones de que la catástrofe afecte tanto al SF original como a la copia de seguridad.

- ¿Encriptación? En función de la ubicación geográfica de las copias de seguridad, encriptarlas puede ser una buena precaución para evitar que alguien no autorizado tenga acceso a los datos.

- ¿Verificación? En función del dispositivo de almacenamiento elegido, periódicamente habría que verificar que las copias de seguridad sigan siendo legibles.

- ¿Identificación? Hay que disponer de algún mecanismo que permita saber, para cada fichero, cuántas copias de seguridad tengo, de qué fechas y dónde están.

3.2.2. Verificación del sistema de ficheros

Muchos SF disponen de una herramienta que permite verificar la coherencia de sus estructuras de datos. Típicamente se ejecuta de manera automática al montar el SF si se cumple una de estas tres condiciones:

fsck

En el caso de Linux, esta herramienta se denomina *fsck* (*file system check*).

- El SF no se desmontó correctamente la última vez que fue montado, por ejemplo debido a una interrupción del suministro eléctrico o a que el SO se "colgó" (*kernel panic* de Linux, "pantalla azul" de Windows).

- Desde la última vez que el SF fue verificado, el SF ha sido montado un cierto número de veces.

- Desde la última vez que el SF fue verificado, ha transcurrido un cierto tiempo.

Esta verificación puede detectar incoherencias como que un bloque esté marcado como libre pero que también esté asignado a un fichero, un bloque marcado como ocupado pero que no está asignado a ningún fichero... En algunos casos, la propia herramienta resolverá el problema. En otros, será el administrador del sistema quien tome la decisión.

3.2.3. Sistemas de ficheros con *journaling*

Esta técnica consiste en mantener una lista[12] de las operaciones que se harán sobre el sistema de fichero pero que aún no se han realizado físicamente. Esta lista se almacena en el sistema de ficheros y se actualiza a medida que las

(12)En inglés, *journal* o *log*.

operaciones se realizan físicamente. En caso de que haya un corte del suministro eléctrico mientras la máquina está en funcionamiento o de que el sistema operativo deje de responder (*kernel panic*), *journaling* permite reconstruir la coherencia del sistema de ficheros, lo que reduce la probabilidad de pérdida de información.

Nota

Recordad que, por motivos de eficiencia, el SO mantiene una copia en memoria de las estructuras de datos relativas a los ficheros que se están utilizando e, inicialmente, realiza las operaciones sobre estas copias ubicadas en memoria. Cuando el SO lo considera oportuno, estas copias ubicadas en memoria se escriben físicamente en el SF.

Hay varios niveles de *journaling* en función de los tipos de operaciones que se registran. Como es de esperar, registrar más tipos de operaciones en el *journal* reduce la probabilidad de pérdida de información, pero penaliza el rendimiento del sistema de ficheros. Es decisión del administrador del sistema determinar el nivel de *journaling* adecuado a cada sistema de ficheros.

3.2.4. La redundancia

Una manera de garantizar la accesibilidad y la fiabilidad del SF consiste en poner más de un disco de destino para cada disco que se necesita. Con este sistema de trabajo cada fichero se graba en más de un disco al mismo tiempo y se accede a él por lectura desde cualquiera de los discos en que se ha grabado. Así, en caso de fallar un disco no se pierde la información y se garantiza que el sistema pueda continuar trabajando.

Los discos duros tienen un tiempo de vida finito. Más tarde o más temprano, todos los discos acabarán experimentando un problema de hardware con el resultado de que una parte más o menos grande de la información almacenada no será legible. Gracias a las copias de seguridad, podemos recuperar el contenido del disco, pero habremos perdido las modificaciones realizadas desde el momento de hacer la copia de seguridad; además, el procesamiento de las copias de seguridad puede ser lento, con lo que el sistema de ficheros no podrá ser utilizado por un período de tiempo. En entornos de alta disponibilidad o 24x7 (servidores de entidades financieras, servidores de tiendas en línea, etc.), hay que encontrar una solución transparente al usuario al problema de los fallos de los discos.

Una posible solución pasa por utilizar la redundancia, es decir, escribir un mismo bloque de disco dos o más veces. Para estar protegidos de errores de hardware en un disco duro, hay que hacer estas escrituras redundantes sobre discos diferentes. Si más adelante uno de los discos falla, podremos leer el bloque de alguno de los otros discos donde ha sido escrito.

El sistema operativo observará un único disco lógico. Físicamente, el contenido de este disco lógico estará almacenando en dos o más discos duros. La controladora de disco se encargará de decidir sobre qué disco físico hay que hacer cada lectura solicitada por el sistema operativo y sobre qué discos físicos hay que hacer cada escritura. Para facilitar la operación en caso de fallo de un disco, las cabinas de discos suelen tener uno o más discos en reserva. En caso de que la controladora detecte que un disco duro está dando problemas, cargará el disco en reserva con el contenido del disco erróneo (para hacerlo,

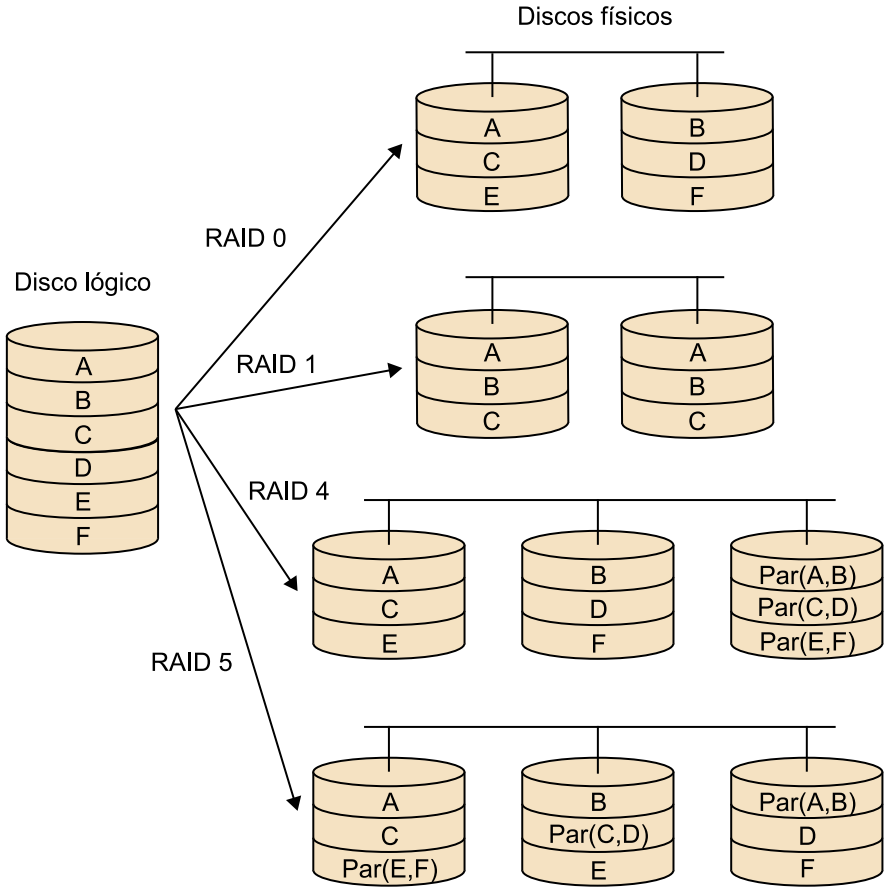
(13)En inglés, denominado *hot swap*.

probablemente habrá que leer las réplicas de los bloques distribuidos en los otros discos) y pasará a utilizar el disco en reserva en lugar del disco erróneo. Hay que notar que el proceso de recuperación[13] se puede realizar sin tener que detener ningún servicio.

Esta técnica recibe el nombre genérico de RAID. Hay varias arquitecturas RAID[14]; básicamente, difieren en el grado de fiabilidad que aportan y en el grado de redundancia que introducen. Con la ayuda de la figura 19, describimos algunas de las arquitecturas RAID más usuales (por simplicidad, asumiremos que todos los discos físicos que constituyen el RAID tienen las mismas características).

[14]RAID es la sigla de *redundant array of inexpensive disks*.

Figura 19. Esquema de varias arquitecturas RAID



El número de discos utilizados en cada esquema no es más que un ejemplo; en cualquier arquitectura RAID se pueden utilizar más discos físicos.

a) RAID 0 (*striping*): esta arquitectura no introduce redundancia, con lo que, propiamente, no es un RAID. Distribuye el contenido del disco lógico entre dos o más discos (en la figura 19, entre dos discos), pero cada bloque se escribe únicamente en un disco. Aunque esta arquitectura no aporta tolerancia a fallos, permite incrementar el ancho de banda de lectura/escritura en el sistema de ficheros si las operaciones se realizan sobre discos diferentes.

b) RAID 1 (*mirroring*): esta arquitectura replica el contenido del disco *N* veces (en la figura, *N* = 1). Por lo tanto, cada operación de lectura se puede servir desde cualquiera de los *N* + 1 discos y cada operación de escritura se debe hacer sobre todos los *N* + 1 discos simultáneamente. Esta arquitectura toleraría fallos en hasta *N* discos, pero puede llegar a desperdiciar mucho espacio de disco.

c) RAID 4 (*striping with dedicated parity*): esta arquitectura introduce el concepto de bloque de paridad. Añade un disco a un RAID 0 de *N* discos (en la figura, *N* = 2); en este nuevo disco almacenará bloques de paridad. Utilizando una función lógica sencilla (por ejemplo, la función *XOR*), construirá el bloque de paridad correspondiente a cada *N* bloques de datos. En caso de que, más adelante, un bloque de datos cualquiera de estos *N* no sea legible, el bloque de paridad correspondiente permite reconstruir el contenido del bloque ilegible. Hay que señalar que siempre que se modifique un bloque de datos, habrá que recalcular el bloque de paridad correspondiente y escribirlo en el disco que almacena los bloques de paridad.

Bloque de paridad

Asumido que el bloque A tenga el valor 010101 y el bloque B, el valor 100011, con la función *XOR* bit a bit calcularíamos el bloque de paridad correspondiente a A y B: 110110. En caso de que el bloque A se convierta en ilegible, lo podremos reconstruir aplicando la función *XOR* bit a bit en el bloque B y en el bloque de paridad.

d) RAID 5 (*striping with distibuted parity*): la diferencia entre el RAID 4 y el RAID 5 es que RAID 5 distribuye los bloques de paridad entre todos los discos físicos. Así, las operaciones de escritura se distribuyen de manera uniforme entre todos los discos físicos del RAID.

e) RAID 6 (*striping with double distributed parity*): es una extensión del RAID 5 pero incorporando dos bloques de paridad (calculados utilizando funciones diferentes). De esta manera, podemos tolerar hasta dos fallos en los discos. La contrapartida es que disminuye la eficiencia a la hora de utilizar el espacio en disco porque "desaprovechamos" el espacio equivalente a dos discos enteros del RAID.

Hay que notar que ninguna de estas arquitecturas RAID protegen del hecho de que un usuario borre accidentalmente un fichero. Este borrado se propagará inmediatamente en todos los discos que lo almacenaban. Por lo tanto, no hay que confundir un RAID con una copia de seguridad.

4. Anexo. Sistemas de ficheros de uso más habitual

Este apartado anexo describe algunos de los sistemas de ficheros de propósito general más habituales.

4.1. FAT

El FAT[15] es un sistema de ficheros creado por Microsoft a finales de la década de 1970 y ha sido utilizado amplísimamente porque es el sistema de ficheros propio de los sistemas operativos MSDOS y de las primeros versiones de Windows. A lo largo de los años ha ido evolucionando para adaptarse a la capacidad de los dispositivos de almacenamiento y para incorporar nuevas prestaciones. A pesar de esta evolución, todas las versiones de FAT son "monousuario"; por lo tanto, no es un sistema de fichero apropiado para los sistemas operativos multiusuario de propósito general. No obstante, actualmente FAT se continúa utilizando en los disquetes y, por su simplicidad e interoperabilidad, también se utiliza en dispositivos más modernos, como lápices de memoria USB[16] y tarjetas de memoria de cámaras digitales y dispositivos móviles.

(15)FAT proviene de *file allocation table*.

(16)En inglés, *pendrive*.

Monousuario

El sistema de ficheros FAT no almacena información relativa a qué usuario ha creado cada fichero. Hay que tener en cuenta que MSDOS es un sistema operativo monousuario.

Un sistema FAT divide el espacio de almacenamiento en clústeres, donde cada clúster está formado por un número fijo de sectores de disco contiguos. El clúster se convierte en la unidad de asignación de espacio al sistema de ficheros. Típicamente, los sectores son de 512 bytes y los clústeres pueden ser de hasta 32 KB.

La estructura de datos básica de un sistema FAT es una tabla de encadenamientos[17] similar a la descrita en el subapartado que trata de los ficheros. Esta tabla centraliza la información relativa a la secuencia de clústeres donde se almacena el contenido de cada fichero, así como la lista de clústeres libres. También identifica los clústeres defectuosos y que, por lo tanto, no pueden ser asignados a ningún fichero.

(17)En inglés, *file allocation table*.

Ved también

Podéis ver las tablas de encadenamientos en el subapartado 1.2.2 de este módulo didáctico.

La representación de los directorios utiliza un tipo especial de fichero donde se almacenan una serie de estructuras de datos de 32 bytes. Cada una de estas estructuras está asociada a un fichero y contiene:

- El nombre del fichero en formato 8 + 3: 8 caracteres de nombre y 3 caracteres de extensión.
- El tamaño del fichero (en bytes).
- El identificador del primer clúster con contenido del fichero.
- Las fechas de creación, último acceso y modificación.
- Otros atributos (oculto, solo lectura, directorio, etc.).

En un sistema de ficheros FAT podemos identificar las siguientes partes:

- Sector de boot: contiene el código máquina necesario para arrancar el sistema operativo y el bloque de control del sistema de ficheros. El bloque de control contiene datos como las características del dispositivo físico, tamaño de clúster, etc.

- FAT: la tabla FAT del sistema de ficheros. Por fiabilidad, puede estar replicada.

- Región de datos: clústeres donde se almacena el contenido de los ficheros y directorio.

Existen varias versiones del sistema de ficheros FAT: FAT12, FAT16, FAT32. La principal diferencia entre las diferentes versiones es el tamaño (en bits) del identificador de clúster. Este tamaño determina el número de entradas que puede tener la tabla FAT (por ejemplo, al caso de FAT16, la FAT puede tener un máximo de 2^{16} entradas) y también determina el tamaño máximo que puede llegar a tener el sistema de ficheros (siguiendo con el ejemplo, $2^{16}\times$ *tamaño_cluster*). Para captar el impacto de esta variación en el tamaño de los identificadores, hay que pensar que la primera versión de FAT estaba diseñada para disquetes de 160 KB y podía gestionar un sistema de ficheros de hasta 32 MB, y que la última versión puede gestionar un sistema de ficheros de hasta 2 TB.

Además del tamaño máximo del sistema de ficheros, hay otras diferencias:

- Las primeras versiones no permitían la creación de directorios, con lo que todos los ficheros se almacenaban en el directorio raíz.

- En FAT32, el directorio raíz se trata de la misma manera que cualquier otro directorio. En FAT12 y FAT16, se trata de modo especial (de hecho, se considera una parte más del sistema FAT); esto tiene el efecto lateral de limitar el número máximo de ficheros que se pueden crear en el directorio raíz.

- Los usuarios de Windows95 pueden utilizar nombres de fichero de hasta 256 caracteres.

Los sistemas FAT tienen una serie de limitaciones. Además de no incorporar el concepto de usuario propietario del fichero, podemos destacar las siguientes:

- El tamaño máximo de fichero es de 4 GB porque se utilizan 32 bits para codificar el tamaño del fichero. En algunos entornos (bases de datos, multimedia, etc.), esto es insuficiente.

- Las fechas de último acceso y modificación están almacenadas con una resolución de 2 segundos (la fecha de creación puede almacenarse con una precisión de 0,01 segundos). Algunas aplicaciones necesitan que el sistema de ficheros sea más preciso en este aspecto.

- El rango de fechas posibles es bastante limitado: del 1 de enero de 1980 al 31 de diciembre del 2107.

- Como las últimas versiones de FAT suelen utilizar un tamaño de clúster de 16 o de 32 KB, la fragmentación interna puede provocar una pérdida considerable de espacio.

- FAT no proporciona ningún mecanismo para intentar minimizar la fragmentación de los ficheros dentro del sistema de ficheros. Una excesiva fragmentación del sistema de ficheros puede incrementar significativamente el tiempo de acceso a los ficheros afectados.

- En algunos escenarios, el rendimiento de los sistemas FAT no es satisfactorio. Por ejemplo, accesos directos a ficheros de gran tamaño.

- En caso de caída del sistema o interrupción del suministro eléctrico, el sistema de ficheros puede quedar en un estado inconsistente. Hay que utilizar alguna herramienta externa para intentar resolver las inconsistencias.

4.2. EXT

La familia de sistemas de ficheros EXT[18] fue creada en 1992. El sistema de ficheros EXT fue primero específico para el sistema operativo Linux. Está inspirado en el sistema de ficheros clásico de los sistemas operativos Unix, pero a lo largo de los años ha ido evolucionado y ha incorporado nuevas funcionalidades y mejoras. A continuación, se describen las principales características de las versiones EXT2, EXT3 y EXT4.

[18]EXT denota *extended file system*.

1) EXT2 (*second extended file system*)

Fue introducido en 1993 para superar las limitaciones de la versión previa EXT. EXT2[19] ha sido el sistema de ficheros por excelencia de las distribuciones Linux a lo largo de casi 10 años, hasta su paulatina sustitución por EXT3/4. No obstante, EXT2 es todavía el sistema de ficheros Linux recomendado para dispositivos de almacenamiento basados en tecnología flash debido a que EXT2 realiza menos operaciones de escritura que EXT3/4.

[19]EXT2 denota *second extended file system*.

Tecnología flash

Los dispositivos flash tienen limitado el número de ciclos de escritura/borrado que pueden soportar.

Los sistemas de ficheros EXT2 dividen el espacio de almacenamiento de la partición en bloques de tamaño fijo (típicamente 4 KB). Los **bloques** son la unidad de asignación de espacio en EXT2 y cada bloque está identificado con

[20]En inglés, *block groups*.

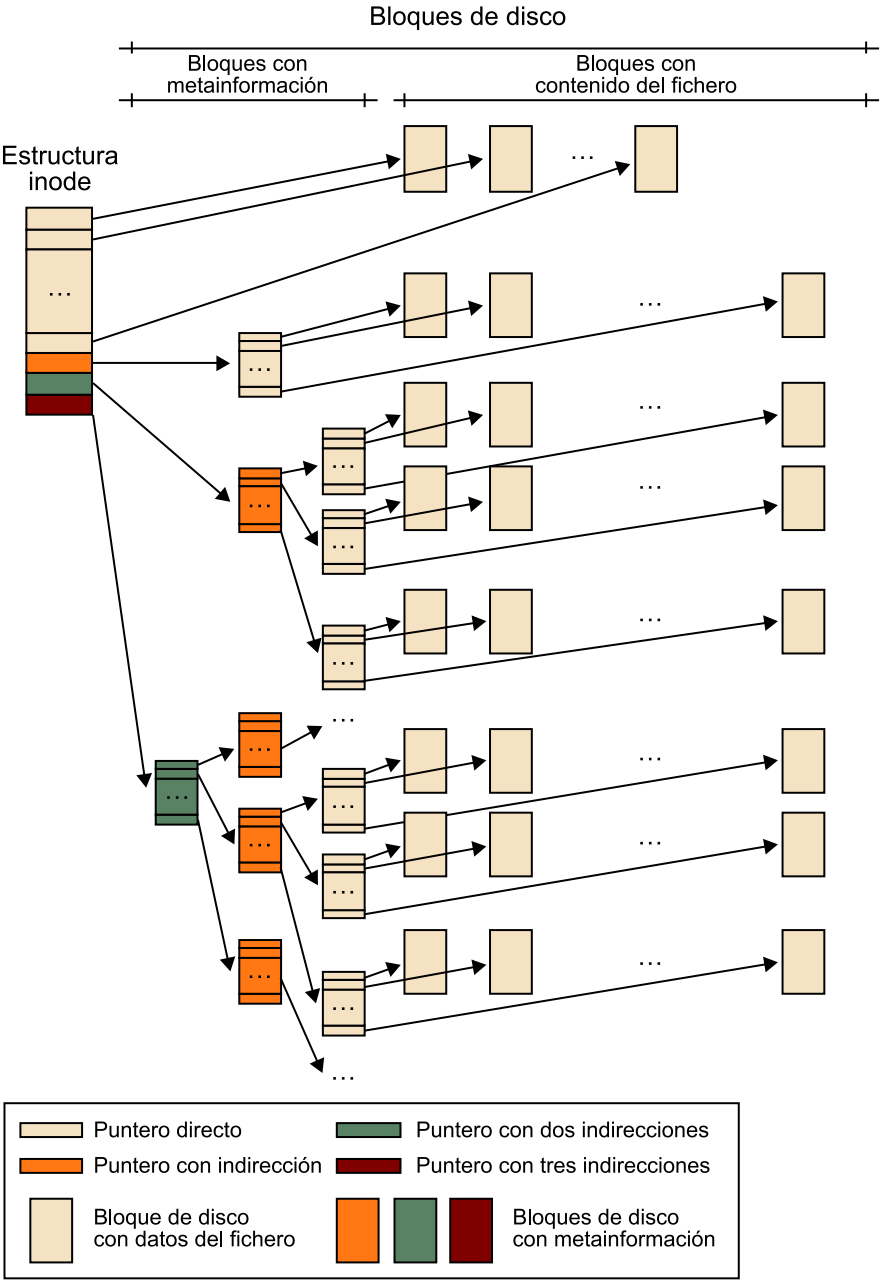
un número entero de 32 bits (lo que limita, por lo tanto, el número máximo de bloques que puede haber en un sistema de ficheros EXT2). Los bloques se agrupan en grupos de bloques[20]; se puede acceder a todos los bloques de un grupo de bloques sin desplazar (o desplazando mínimamente) el cabezal de lectura/escritura del disco.

Algunas de las características de los sistemas de ficheros EXT2 son:

a) Inodos. Una de las estructuras de datos básica de EXT 2 es el inodo. Cada fichero ordinario, directorio, fichero dispositivo... del sistema de ficheros se representa utilizando un inodo. Cada inodo ocupa 128 bytes y se identifica mediante un número entero; por convenio, el inodo 2 se asocia al directorio raíz del sistema de ficheros. El inodo contiene datos como el usuario propietario del fichero, el tamaño del fichero, las protecciones (en formato *rwxrwxrwx*), las fechas de creación/modificación/último acceso, la ubicación de los datos del fichero en disco, el tipo del fichero (ordinario, directorio, dispositivo, etc.) y el número de enlaces físicos al disco. Observemos que el nombre del fichero no se almacena en el inodo, sino que se almacena en las estructuras de datos que representan los directorios que integran el camino desde el directorio raíz hasta el directorio donde se encuentra el fichero.

Cada inodo dispone de una tabla de índices para representar la lista de bloques donde se almacena el contenido del fichero. Esta lista debe tener tantas posiciones como bloques ocupe el fichero, pero, en general, en un sistema de ficheros encontramos ficheros de tamaños muy diferentes. Por lo tanto, la implementación de esta lista debe poder representar tanto listas cortas como listas largas sin desperdiciar demasiado espacio ni penalizar el tiempo de acceso al fichero. La solución encontrada pasa por crear un sistema de almacenamiento multinivel (figura 20).

Figura 20. Estructura arbórea multinivel utilizada por EXT2 para representar la lista de bloques donde se almacena el contenido de un fichero



En cada inodo se almacenan 15 punteros. Cada puntero identifica un bloque de disco:

- Los 12 primeros punteros (punteros directos) identifican los 12 bloques de disco que almacenan el comienzo del contenido del fichero. Si el tamaño de bloque es de 4 KB, mediante estos punteros podremos acceder a los primeros 48 KB del fichero. Si el fichero tiene un tamaño superior, habrá que utilizar el siguiente puntero para extender la lista de bloques.

- El puntero número 13 es el puntero con una indirección. Este puntero apunta a un bloque de disco que contendrá la continuación de la lista de punteros directos. Asumiendo bloques de 4 KB y punteros de 4 bytes, en un bloque de disco es posible almacenar hasta 1.024 identificadores

de bloques de disco con contenido del fichero, los correspondientes a los identificadores de los bloques 13, 14, ..., 1.036 del fichero. Por lo tanto, con este puntero del inodo (y un bloque de disco con metainformación), podemos extender la lista de punteros directos en 1.024 posiciones, con lo que es posible representar ficheros de hasta 4 MB + 48 KB. Si el fichero tiene un tamaño superior, habrá que utilizar el siguiente puntero para extender la lista de bloques.

- El puntero número 14 es el puntero con dos indirecciones. La idea es análoga a la del puntero anterior, pero incorporando un segundo nivel de indirección. Este puntero apunta a un bloque de disco que apuntará a 1.024 bloques de disco. Estos 1.024 bloques contendrán la continuación de la lista de punteros directos. Por lo tanto, con este puntero del inodo (y hasta 1 + 1.024 bloques de disco con metainformación), podemos extender la lista de punteros directos en 1.024^2 posiciones, con lo que es posible representar ficheros de hasta 4 GB + 4 MB + 48 KB. Si el fichero tiene un tamaño superior, habrá que utilizar el siguiente puntero para extender la lista de bloques.

- El puntero número 15 es el puntero con tres indireccions. Análogamente al puntero anterior, incorpora un nivel más en la estructura arbórea que permite extender la lista de punteros directos. Por lo tanto, con este puntero del inodo (y hasta $1 + 1.024 + 1.024^2$ bloques de disco con metainformación) podemos extender la lista de punteros directos en 1.024^3 posiciones, con lo que es posible representar ficheros de hasta 4 TB + 4 GB + 4 MB + 48 KB. En estas condiciones, este es el tamaño máximo que puede alcanzar un fichero.

b) Representación de los directorios. Para representar los directorios, cada directorio tiene asociado un bloque de disco donde se almacena una tabla que relaciona los nombres de los ficheros almacenados en el directorio con el identificador de inodo correspondiente. La creación de un enlace físico a un fichero existente requiere asociar el nuevo nombre con el identificador de inodo correspondiente.

c) Estructura sistema EXT2. En un sistema de ficheros EXT2 encontramos el bloque de *boot* y una serie de grupos de bloques. El bloque de *boot*, propiamente no gestionado por EXT2, contiene el programa que se carga en memoria cuando arranca la máquina y que se encarga de iniciar el sistema operativo.

En cada grupo de bloques se almacena información de control del sistema de ficheros, inodos y bloques de datos. Concretamente:

- Superbloque[21] del sistema de ficheros. Es el bloque del control del sistema de ficheros. En EXT2 es una estructura de 1.024 bytes que contiene información como el tamaño de bloque, tamaño del sistema de ficheros,

(21)En inglés, *superblock*.

número de grupos de bloques, número total de inodos y de bloques de datos, fecha de última verificación, etc. Por fiabilidad, está replicado en cada grupo de bloques del sistema de ficheros.

- Descriptores de grupo[22]. Cada grupo de bloques está representado con una estructura de datos de 24 bytes. Por fiabilidad, en cada grupo de bloques tenemos una copia del descriptor de todos los grupos de bloques.

(22)En inglés, *group descriptors*.

- Tabla de inodos. En cada grupo de bloques se almacena un determinado número de estructuras inodo (cada una ocupa 128 bytes).

- Mapa de bits[23] de los inodos. Cada grupo de bloques dispone de un mapa de bits para gestionar qué inodos del grupo de bloques son libres. El tamaño de este mapa de bits coincide con el tamaño de bloque del sistema de ficheros. Por lo tanto, el tamaño de bloque determina el número máximo de bloques de datos que puede existir en cada grupo de bloques.

(23)En inglés, *bitmap*.

- Bloques de datos[24]. Bloques destinados a almacenar el contenido de los ficheros.

(24)En inglés, *data blocks*.

- Mapa de bits de los bloques de datos. Análogamente al mapa de bits de los inodos, cada grupo de bloques dispone de un mapa de bits para gestionar qué bloques de datos del grupo de bloques están libres. El tamaño de este mapa de bits coincide con el tamaño de bloque del sistema de ficheros. Por lo tanto, el tamaño de bloque determina el número máximo de bloques de datos que puede existir en cada grupo de bloques.

El objetivo de los grupos de bloques es facilitar que los bloques asignados a un fichero estén en el mismo grupo de bloques que su inodo, reduciendo la dispersión de los bloques de datos del fichero dentro del sistema de ficheros, con lo que se reduce el tiempo necesario para acceder al contenido del fichero.

Ejemplo

Asumiendo que el tamaño de la partición EXT2 es de 16 GB y que los bloques de disco son de 8 KB, contestad a las preguntas siguientes:

a) ¿Cuántas copias del superbloque hay en la partición? Como existe una copia del superbloque en cada grupo de bloques, necesitamos saber cuántos grupos de bloques hay en la partición. El tamaño de un grupo de bloques está determinado, básicamente, por el número de bloques de datos que puede contener.

El número de bloques de datos en cada grupo de bloques está determinado por el número de *blocks* que se pueden representar con un mapa de bits que tenga el tamaño de un bloque de disco. En nuestro caso, los bloques de disco son de 2^{13} bytes, por lo tanto, el mapa de bits contendrá $2^{13} \times 2^3 = 2^{16}$ bits. Por ende, cada grupo de bloques tendrá, como máximo, 2^{16} bloques de datos, es decir, 2^{16} bloques × 2^{13} bytes/bloque = 2^{29} bytes. Como el tamaño de la partición es de 16 GB = 2^{34} bytes, el número de grupos será 2^{34} bytes / 2^{29} bytes/grupo de bloques = 2^5 grupos de bloques = 32 grupos de bloques. Por lo tanto, el número de copias del superbloque será de 32.

b) ¿Cuántos bloques de la partición están destinados a contener mapas de bits? En cada grupo de bloques hay 2 bloques destinados a contener mapas de bits (uno para inodos y

Observación

Los grupos de bloques también contienen algunos bloques de control (superbloque, descriptores de grupo, tablas de inodos y mapas de bits). Esto hará que el último grupo de bloques sea más pequeño que el resto de los grupos de bloques.

otro para bloques de datos). Por lo tanto, 32 grupos de bloques × 2 mapas de bits / grupo de bloques × 1 *block*/mapa de bits = *64 blocks*.

c) Si la partición ha sido creada asumiendo un inodo para cada 16 KB, ¿cuántos bloques están destinados a tablas de inodo? 2^5 grupos de bloques × 2^{16} bloques de datos/grupo de bloques × 2^{-1} inodos/bloque de datos × 2^7 bytes/inodo × 2^{-13} bloques/byte = 2^{14} bloques.

d) Si la partición ha sido creada asumiendo un inodo para cada 8 KB, ¿cuántos bloques están destinados a tablas de inodos? El doble que en el caso anterior porque el único factor que cambia es que ahora tenemos 1 inodo/bloque de datos. Por lo tanto, 2^{15} bloques.

e) Si la partición ha sido creada asumiendo un inodo para cada 4 KB, ¿cuántos bloques están destinados a tablas de inodo? En principio, no tiene sentido porque la unidad mínima de asignación de espacio en disco es el bloque. Por lo tanto, como mucho, necesitaremos tantos inodos como bloques de datos. Además, esta proporción obligaría a que los mapas de bits correspondientes a inodos ocuparan dos bloques de disco en cada grupo de bloques.

Ahora bien, algunos ficheros especiales (*named pipes*, nombres simbólicos...), aunque tienen asociado un inodo, no tienen asignado ningún bloque de datos. Además, los ficheros de tamaño 0 bytes tampoco tienen asociado ningún bloque de datos.

f) En el caso d), ¿cuál es el tanto por ciento de espacio de la partición destinado a almacenar bloques de datos? Si nos fijamos en un grupo de bloques, tenemos:

- 1 bloque para el superbloque.
- 1 bloque para descriptores de grupo (2^5 grupos de bloques × 2^5 bytes/descriptor de grupo = 2^{10} bytes).
- 2 bloques de mapas de bits.
- 2^{10} bloques de tablas de inodo.
- 2^{16} bloques de datos.

Por lo tanto, $100 \times 2^{16} / (2^{16} + 2^{10} + 2 + 1 + 1) = 98{,}45$ %.

Si consideramos el resto de los grupos de bloques, la proporción sería la misma salvo el último grupo de bloques, donde la proporción será menor.

g) Asumiendo que los bytes de un fichero se numeran empezando por 0, ¿cuál es el primer byte que requiere utilizar el mecanismo de la triple indirección para obtener su *internal fragmentation* contenido? Aplicando la fórmula de la página 596 (y multiplicándola por el tamaño del bloque para obtener el número de byte), $2^{13} \times ((2^{11})^2 + 2^{11} + 12) = 2^{35} + 2^{24} + 12 \times 2^{13} = 32$ G + 16 M + 96 K = 34.376.613.888.

Dado que este valor es superior al tamaño de la partición (16 GB), esta indirección solo sería utilizable por un fichero "con huecos".

d) Otras características destacables:

- Permite utilizar nombres de ficheros de hasta 255 caracteres.

- El tamaño máximo que puede alcanzar un fichero es de 2 TB y el del sistema de ficheros es 32 TB.

- Soporte a ficheros dispersos: algunas aplicaciones almacenan estructuras de datos tipo *hash* en un fichero. Esto puede provocar que una cantidad significativa de bloques asignados al fichero estén vacíos. EXT2 marca de manera especial estos bloques con el fin de no desperdiciar espacio en disco.

- Porcentaje de reserva: es posible reservar un porcentaje de bloques que, cuando el resto de los bloques estén ocupados, serán utilizables únicamente por el administrador de sistemas. Esto permite que los procesos ejecutados por el administrador puedan continuar disponiendo de espacio en disco cuando los procesos de usuario ya no disponen de él.

Reserva

Esta reserva de espacio para el administrador del sistema aparece en otras estructuras de datos gestionadas por el sistema operativo como la tabla de procesos, la tabla de ficheros abiertos, etc.

- Al crear el sistema de ficheros, el administrador puede especificar una serie de parámetros que permiten adaptar el sistema de ficheros a sus necesidades, entre ellas el tamaño de bloque, el número total de inodos y el porcentaje de bloques en reserva.

e) Limitaciones de EXT2. Finalmente, indicamos algunas de las limitaciones de EXT2:

- No implementa ACL[25] para gestionar las protecciones de los ficheros.

[25]ACL es la sigla de la expresión inglesa *access control list*.

- La resolución de las fechas es de 1 segundo.
- El rango de fechas posibles es bastante limitado: del 14 de diciembre de 1901 al 18 de enero del 2038.
- No ofrece un mecanismo tipo *undelete* para recuperar ficheros que han sido borrados accidentalmente.
- No permite aumentar el número de inodos que se determinó al crear el sistema de ficheros.
- No implementa mecanismos de *journaling* para minimizar los problemas que puede causar una interrupción del suministro eléctrico o un *kernel panic*.
- No integra, de manera transparente al usuario, ni la compresión ni la encriptación de ficheros.
- No puede trabajar con diferentes tamaños de bloque simultáneamente con el fin de minimizar la fragmentación interna.

2) EXT3 (*third extended filesystem*)

EXT3[26] fue introducido en el 2001. La mejora más significativa con respecto a EXT2 es la técnica de *journaling*.

[26]EXT3 denota *third extended file system*.

Otras mejoras que aporta EXT3 con respecto a EXT2 son:

- Permite aumentar el número de inodos del sistema de fichero sin necesidad de reformatear el sistema.

- Incorpora estructuras de datos de tipo arbóreo para gestionar directorios con un número elevado de ficheros.

3) EXT4 (*fourth extended file system*)

EXT4[27] fue introducido en el 2006. El cambio más significativo con respecto a EXT3 es la aparición de los *extents*, que modifican el *block mapping* de EXT2/3, es decir, la manera de representar la lista ordenada de bloques con el contenido del fichero.

(27)EXT4 denota *fourth extended file system*.

Incorpora el concepto de *extents*, un grupo de bloques de disco físicamente contiguos que puede llegar a tener un tamaño de 128 MB. Esto permite reducir considerablemente el tamaño de la estructura de datos que representa la lista ordenada de identificadores de bloques de disco con el contenido del fichero porque ya no hay que hacer referencia a los bloques de manera individual.

Otras mejoras que aporta EXT4 con respecto a EXT3 son:

- Incrementa el tamaño máximo que puede llegar a alcanzar un fichero (16 TB, asumiendo bloques de 4 KB) y el sistema de ficheros (1 EB).

- Permite que un directorio pueda tener más de 32.000 subdirectorios.

- *Multiblock allocation*: permite reservar varios bloques de disco de una vez (EXT3 los tenía que reservar uno por uno). Esto permite implementar políticas para evitar la fragmentación del sistema de ficheros.

- *Persistent pre-allocation*: permite que el usuario pueda especificar que hay que prerreservar bloques para un fichero que se utilizarán cuando aumente el tamaño del fichero.

- *Delayed allocation*: retrasa el máximo posible la asignación de espacio de disco en los ficheros. De esta manera, en algunos casos será posible saber cuál será el tamaño del fichero, con lo que se podrá hacer una asignación de bloques más adecuada (*multiblock allocation*).

- Incrementa la resolución de las fechas de creación/último acceso/modificación almacenadas en las estructuras inodos hasta nanosegundos.

4.3. NFTS

NFTS[28] es el sistema de ficheros desarrollado por Microsoft para los sistemas operativos de nueva generación sucesores de MS-DOS como Windows NT, Windows XP, Windows Vista, Windows 7, etc. El objetivo de NTFS es ofrecer un sistema de ficheros para sistemas operativos multiusuario y que incorpore

(28)NTFS es la sigla de *new technology file system*.

importantes mejoras con respecto a FAT en términos de rendimiento, fiabilidad y prestaciones. La primera versión data de 1993, aunque es incompatible con versiones posteriores de NTFS.

La estructura de datos básica de NFTS es la *master file table* (MTF[29]), que contiene la metainformación de todos los ficheros almacenados en el sistema de ficheros. Si el fichero es lo suficientemente pequeño, el propio contenido del fichero se almacena en la MFT.

[29]MFT es la sigla de *master file table*.

Como características más destacadas de las últimas versiones de NTFS podemos indicar:

- Teóricamente, el tamaño máximo de fichero es de 16 EB, y el tamaño máximo del sistema de ficheros es de 2^{64} clústeres, pero las implementaciones de NTFS aún no alcanzan este valor. En Windows XP Profesional, el tamaño máximo de fichero era de 16 TB y el del sistema de ficheros, 2^{32} clústeres.
- El tamaño máximo de los nombres de los ficheros es de 255 caracteres unicode UTF-16.
- El rango de fechas válidas para los ficheros comprende del 1 de enero de 1601 al 28 de mayo del 60056.
- La resolución de las fechas es de 100 nanosegundos.
- Utiliza un sistema de protecciones basados en ACL.
- Soporta compresión y encriptación en el nivel de fichero de manera transparente al usuario.
- Posibilita definir cuotas de disco en el nivel de usuario.
- Soporta ficheros dispersos.
- Utiliza estructuras de datos arbóreos para gestionar la lista de ficheros que forman parte de los directorios.
- Incorpora mecanismo de *journaling* para minimizar el efecto de una interrupción del suministro eléctrico.
- Permite el crecimiento dinámico del sistema de ficheros.
- Posibilita mantener varias versiones de ficheros y directorios.

4.4. BTRFS

El sistema de ficheros BTRFS[30] está diseñado para sustituir los sistemas de ficheros EXT2/3/4 en máquinas Linux. Las primeras versiones no estables de este sistema estuvieron disponibles desde el 2009. El objetivo básico de Btrfs es ser un sistema de ficheros escalable en las nuevas capacidades de almacenamiento que estarán disponibles próximamente.

(30)BTRFS denota *better filesystem, b-tree filesystem, butter file system...*

Las principales mejoras que aporta BTRFS con respecto a EXT4 son:

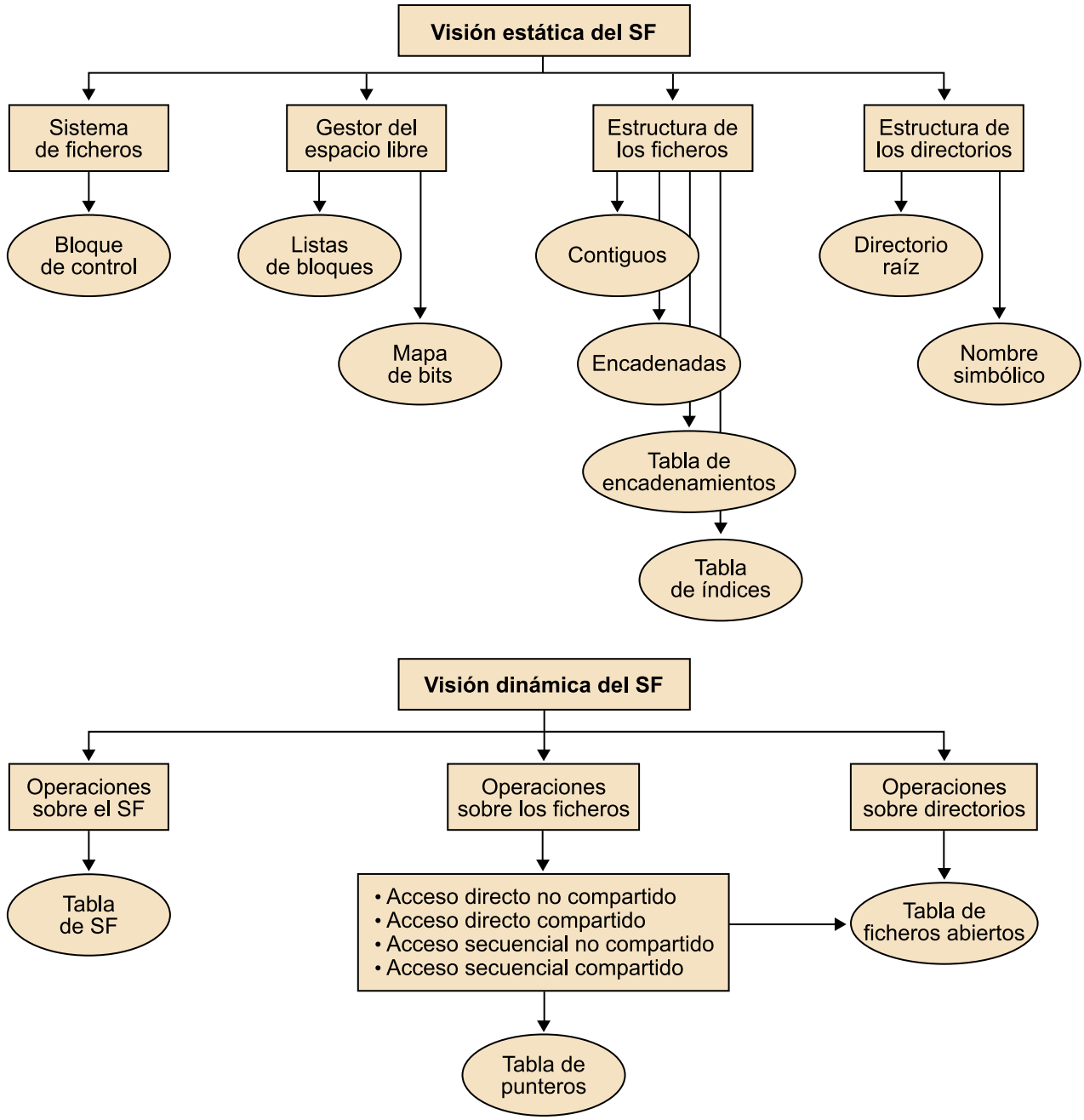
- Incrementa el tamaño máximo que puede llegar a alcanzar un fichero (16 EB) y el sistema de ficheros (16 EB).
- *Block suballocation*: para reducir la fragmentación interna, es capaz de asignar porciones disjuntas de un mismo bloque en dos ficheros.
- Compresión de ficheros transparente al usuario (y posiblemente, también la compresión).
- *Copy on write*: si hacemos una copia de un objeto, este no se replica físicamente en disco hasta que alguna de las copias sea modificada.
- *Allocatation* dinámica de inodos (permite crear nuevos inodos después de formatear el sistema de ficheros).
- Permite verificar el sistema de ficheros mientras este está montado.
- Permite verificar el sistema de ficheros de manera muy rápida mientras este no está montado.
- Está optimizado para discos SSD.
- *Checksums* a datos y metadatos.
- Capacidad de realizar "fotografías[31]" de los ficheros o de porciones de estos en un momento dado.
- Permite integrar varios dispositivos de almacenamiento dentro de un único sistema de ficheros.

(31)En inglés, *snapshots*.

Resumen

En este módulo didáctico hemos desarrollado dos visiones del SF, una visión estática y una dinámica. En la figura siguiente, presentamos un mapa conceptual en el que se recogen las características de las dos visiones:

Figura 21. Mapa conceptual de las visiones estática y dinámica del SF



En la visión estática hemos estudiado cómo están estructurados los SF y las diferentes modalidades de discos, tipos de SF, espacio libre, ficheros y directorios. Nos hemos podido dar cuenta de que muchas de las ideas que se han visto en otros módulos, de esta y de otras asignaturas, se podían aplicar a la gestión

de los espacios libres y ocupados. También hemos visto que las diferentes alternativas de gestión no son malas en sí mismas, sino que son más o menos adecuadas en función del entorno donde las aplicamos y de las necesidades de trabajo que tengamos. Una visión dinámica a partir de la cual hemos visto cómo el SO gestionaba el SF para crear y destruir los diferentes elementos (SF, ficheros y directorios) y para acceder a ellos.

Como en la visión estática, hemos aplicado técnicas estudiadas en otras asignaturas, especialmente algunas que están relacionadas con las memorias caché.

Al describir las estructuras y los procedimientos de los SF hemos adoptado el enfoque más genérico posible. Sin embargo, las particularidades de cada sistema harán que en la realidad haya pequeñas diferencias.

Actividades

1. Teniendo en cuenta los mecanismos de representación del espacio libre que hemos visto en el subapartado 1.2.1, encontrad los procedimientos adecuados para incluir y extraer un bloque de memoria en cada una de las tres listas de los esquemas anteriores.

2. Diseñad un sistema de ficheros para un dispositivo que permita únicamente lectura como un CD. ¿Qué simplificaciones podemos hacer sobre un sistema de ficheros de propósito general?

3. Buscad en el manual del sistema Linux, mediante la orden *ls*, qué campos tiene un inodo (estructura de datos que utilizan algunos sistemas de ficheros de Linux para representar los ficheros). Utilizando la operación *ls -al* tratad de encontrar nombres simbólicos y ficheros con más de un nombre.

4. Buscad en el manual del sistema Linux la instrucción *fsck*. Analizadla y comparad los resultados con lo que sabéis de la estructura de los ficheros de Unix.

5. Buscad en el manual del sistema Linux la instrucción *tune2f2*. Analizadla y comprobad los parámetros que podéis configurar en un sistema de ficheros de Linux.

6. Se quiere incrementar la fiabilidad de un SF. ¿Cuál es la técnica más adecuada para afrontar cada una de las siguientes situaciones?

a) Se estropea un disco duro pero queremos poder continuar trabajando como si nada hubiera pasado.

b) Un usuario borra accidentalmente un fichero y quiere recuperarlo.

c) Hay un corte de suministro eléctrico.

d) El SO se cuelga.

7. Haced una tabla comparativa de las diferentes arquitecturas RAID presentadas en este módulo. Asumiendo que el RAID tiene un total de M discos idénticos de tamaño K, para cada arquitectura indicad:

a) El número de discos erróneos que se puede llegar a soportar.

b) La cantidad de espacio destinado a almacenar información redundante.

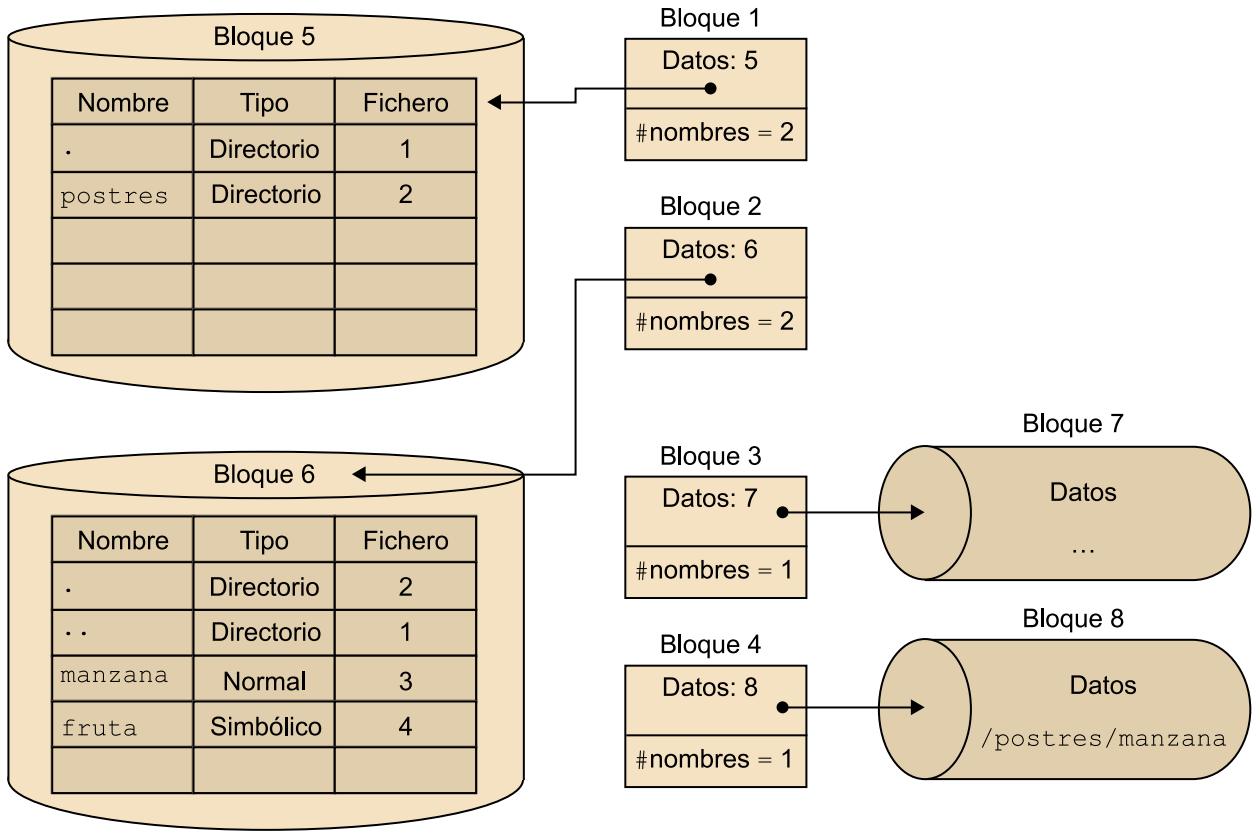
c) Asumiendo que se realizan W operaciones de escritura sobre los ficheros del SF, y que estas están uniformemente distribuidas, indicad cuántas escrituras se hacen sobre cada disco.

8. Buscad información sobre las arquitecturas RAID híbridas RAID 1 + 0 y RAID 0 + 1. ¿En qué se diferencian?

Ejercicios de autoevaluación

1. Suponed que tenemos un disco en el que los bloques contienen la información que se indica en la figura 22:

Figura 22



En función de la información de la figura 22, decid:

a) ¿Cuál es el árbol de directorios del SF que contiene este disco?

b) Si el directorio actual es *postres*, ¿qué información habrá en las estructuras de datos internos en el SO?

c) ¿Cuántos accesos al disco hay que hacer si desde el directorio de trabajo se quiere abrir el fichero *fruta*?

2. Tenemos un fichero de 10.100 bytes con bloques de 256 bytes. Una vez abierta y, por lo tanto, cargada la TFA, queremos leer el byte 830. En función de las estructuras siguientes, indicad cuántos bloques hay que leer:

a) Estructura contigua al disco.

b) Bloques encadenados (cada puntero ocupa 4 bytes).

c) Encadenado por tabla (consideramos que la tabla está en la memoria).

d) Tabla de índices (la tabla de índices forma parte de la estructura del fichero).

3. Tenemos un sistema con un procesador que va a una velocidad de 80 MIPS (millones de instrucciones por segundo) y un disco que gira a 3.600 rpm, con 16 sectores por pista y que invierte 2 ms para transferir un sector. En un SO que tiene bloques de 2 sectores y que necesita ejecutar 1.500 instrucciones después de la transferencia de cada sector, ¿qué *interleaving* es el más adecuado?

De selección

1. Los principales inconvenientes que plantea un sistema con ficheros contiguos son...

a) la dificultad que supone efectuar accesos secuenciales.
b) la fragmentación externa.
c) la dificultad para hacer crecer los ficheros.
d) b) y c).
e) Todas las anteriores.

2. Un mapa de bits es indicado...

a) para tener ficheros con estructura de tabla de índices.
b) para sistemas con pocas restricciones de espacio.
c) para facilitar el acceso secuencial a los ficheros.
d) para representar el espacio ocupado por los ficheros.
e) para representar el espacio libre.

3. La tabla de ficheros abiertos (TFA) contiene...

a) una copia de la estructura de los ficheros y de los directorios abiertos.
b) el mapa del espacio libre necesario para hacer crecer los ficheros.
c) el directorio raíz y el bloque de control del SF.
d) los datos del fichero que se ha leído recientemente.
e) los punteros de lectura/escritura de los ficheros con acceso compartido.

4. Con el fin de mantener la coherencia de la información, en la memoria solo puede haber una copia de...

a) los datos de los ficheros.
b) la estructura del fichero.
c) el mapa de bits.
d) el bloque de control del SF.
e) Todas las respuestas anteriores son correctas.

5. Indicad cuál de las afirmaciones siguientes es la que se ajusta más a las definiciones del *interleaving* y del *skew*.

a) Permiten minimizar el tiempo de ejecución del controlador de entrada/salida.
b) Permiten minimizar el tiempo de espera generado por la rotación del disco.
c) Permiten reducir el tiempo de transferencia del disco.
d) Permiten maximizar el rendimiento del disco.
e) Facilitan la localización del sector siguiente que se ha de transferir.

6. Indicad sobre cuál de las estructuras de ficheros siguientes es más sencillo implementar un acceso secuencial.

a) La tabla de índices.
b) La contigua.
c) El encadenado por tabla.
d) El encadenado.
e) Las estructuras a, b y c son adecuadas.

7. Dado un nombre relativo de un componente, ¿cuántos accesos al disco hay que efectuar para obtener la estructura del fichero al hacer una operación de apertura?

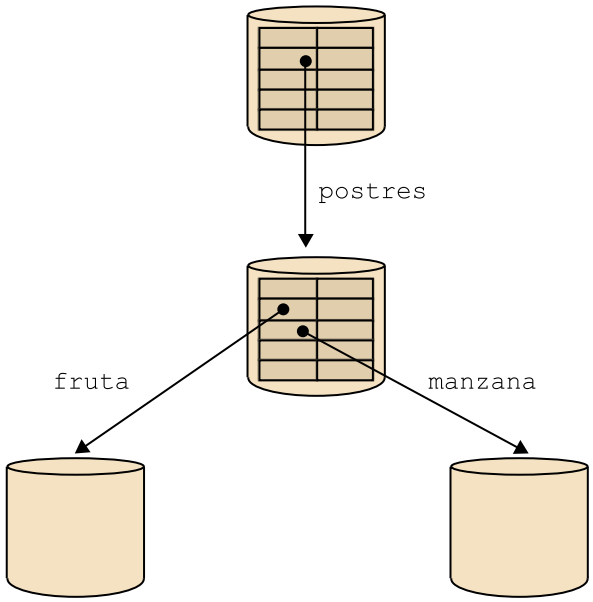
a) Uno para la estructura del fichero directorio de trabajo, uno para sus datos y uno para la estructura del fichero que hay que abrir.
b) Uno para los datos del directorio de trabajo y uno para la estructura del fichero que hay que abrir.
c) Uno para la estructura del fichero que se quiere abrir.
d) Uno para la estructura del directorio raíz, uno para sus datos, uno para la estructura del directorio de trabajo, uno para sus datos y uno para la estructura del fichero que hay que abrir.
e) Uno para la estructura del directorio raíz, uno para sus datos, uno para la estructura del directorio de trabajo y uno para la estructura del fichero que se quiere abrir.

Solucionario

Ejercicios de autoevaluación

1.a) El árbol de directorios que contiene el disco es el siguiente:

Figura 23



b) En la TFA habrá una entrada para la estructura del directorio raíz y una para la estructura del directorio de trabajo. El PCB del proceso con el que trabajamos apuntará a las entradas de la TP, desde donde se hace referencia al directorio de trabajo y al directorio raíz, ya que los dos se consideran abiertos por el proceso. Finalmente, en la TSF habrá una copia del bloque de control del SF y se apuntará a la entrada de la TP que hace referencia a la estructura del directorio raíz.

c) Si suponemos que no hay ningún dato en las memorias intermedias, hemos de hacer los accesos siguientes:

- Un acceso para leer los datos del directorio actual (su estructura ya está en la TFA).
- Un acceso para leer la estructura del fichero *fruta* (es un nombre simbólico).
- Un acceso para leer los datos del fichero *fruta* (obtenemos el nombre nuevo que se tiene que abrir).
- Un acceso para leer los datos del directorio raíz (su estructura ya está en la TFA).
- Un acceso para leer los datos del directorio *postres* (su estructura ya está en la TFA).
- Un acceso para leer la estructura del fichero *manzana* y colocarla en la TFA.

En total se necesitan 6 accesos al disco.

2.a) Conociendo el primer bloque y sabiendo que el byte que se debe leer está en el cuarto bloque del fichero, solo hace falta un acceso.

b) Teniendo en cuenta que con esta organización los bloques solo tienen 252 bytes útiles, resulta que el byte que hay que leer continúa en el cuarto bloque del fichero. Para localizarlo hay que seguir toda la cadena y, por lo tanto, hay que leer cuatro bloques.

c) El byte que se quiere leer, igual que en el primer caso, está en el cuarto bloque. Como la tabla de encadenamientos está toda en la memoria, solo hemos de analizarla para conocer el bloque en cuestión. Por lo tanto, un solo acceso al disco es suficiente.

d) Como en el caso anterior, una vez analizada la tabla de índices, que está en la TFA, se puede acceder a los datos con un único acceso.

3. El tiempo necesario para iniciar la transferencia del segundo sector de un bloque desde que ha finalizado la transferencia del primero es la suma de los tiempos de ejecución de las instrucciones

$$1.500 / 80.000 = 0{,}01875 \text{ ms}$$

y el de transferencia de sector:

0,01875 ms + 2 ms = 2,01875 ms.

Calculamos el tiempo que se tarda en hacer la rotación de un sector:

60 revoluciones/s = 1 / 60 s/revolución = 16,6666 ms/revolución

16,6666 ms/revolución / 16 sectores/revolución = 1,0416 ms/sector

Para acabar, calculamos el *interleaving* necesario en función de los dos datos precedentes:

2,01875 ms / 1,0416 ms / sector = 1,9381 sectores

Por lo tanto, hay que elegir un *interleaving* de 2 sectores.

4. d

5. e

6. a

7. e

8. b

9. b

10. b

Glosario

best-fit Política de asignación del espacio libre que asigna el hueco que mejor se ajusta a cada petición.

bloque *m* Unidad de direccionamiento y transferencia que utiliza internamente el SO.

bloque de control *m* Bloque del disco que contiene información del SF y que se encuentra en una posición concreta del disco a fin de que el SO lo pueda localizar fácilmente.

copia de seguridad (*back-up*) *f* Copia de un SF que se hace sobre otro dispositivo.

copy back *f* Técnica que actualiza, sobre la memoria principal, las modificaciones que se han hecho sobre una línea de la memoria caché en el momento en el que esta debe ser reemplazada.

directorio raíz *m* Directorio que es la puerta de entrada al espacio de nombres. Debe estar en un lugar conocido, normalmente en el primer bloque de datos del disco. Se crea con el SF y no puede ser destruido aunque esté vacío.

directorio de trabajo *m* Directorio sobre el que trabaja un usuario en un instante determinado. Es la base de donde salen los nombres relativos que utiliza este usuario.

enlace simbólico *m* Nombre que relaciona un nombre con un fichero mediante el nombre de otro fichero, en lugar de hacerlo directamente con su estructura de datos.

estructura con tabla de encadenamientos *f* Estructura de un fichero que consiste en tener una tabla con tantos punteros en bloques como bloques destinados a datos tiene el disco. Así pues, un fichero se representa como una cadena de entradas en la tabla.

estructura con tabla de índices *f* Estructura que consiste en tener para cada fichero una lista secuencial ordenada de los bloques que contienen los datos del fichero.

estructura contigua *f* Estructura de un fichero que almacena la información en orden secuencial, sobre bloques del disco contiguos.

estructura encadenada *f* Organización de los ficheros que consiste en hacer que se configuren físicamente como una cadena de bloques.

first-fit Política de asignación del espacio libre que asigna el primer hueco que se encuentra capaz de satisfacer la demanda recibida.

fragmentación externa *f* Situación en la que no se encuentra ningún hueco (espacio contiguo libre) lo suficientemente grande para satisfacer una petición de memoria, aunque haya trozos libres más pequeños, cuya suma de las capacidades sería suficiente para satisfacerla.

fragmentación interna *f* Situación en la que no se encuentra ningún hueco (espacio contiguo libre) lo suficientemente grande para satisfacer una petición de memoria, aunque la suma del espacio asignado que no se utiliza sería suficiente para satisfacerla.

hashing Técnica destinada a agilizar el acceso por medio del contenido de la información almacenada en una estructura de datos.

interleaving Técnica de minimización del tiempo de rotación del disco necesario para transferir más de un sector de la misma pista que consiste en numerar los sectores del disco según el orden lógico en el que los verá el SO.

localidad espacial *f* Un objeto tiene localidad espacial cuando es muy probable que en los instantes siguientes al acceso al objeto se acceda a los próximos a él.

localidad temporal *f* Un objeto tiene localidad temporal cuando es muy probable que en los instantes siguientes al acceso al objeto se vuelva a acceder a él.

memoria caché de disco *f* Técnica basada en la teoría de memorias caché que sirve para optimizar el tiempo de acceso al disco.

sector *m* Unidad de ordenación del disco.

skew Técnica destinada a minimizar el tiempo de rotación del disco necesario para transferir dos sectores que se encuentran en diferentes pistas de un disco que distribuye los sectores en

las pistas en función del tiempo que el disco necesita para mover el brazo y de su velocidad de rotación.

tabla de ficheros abiertos (TFA) *f* Estructura de datos interna del SO donde se guardan las copias de las estructuras de los ficheros abiertos.

tabla de los sistemas de ficheros (TSF) *f* Estructura de datos interna del SO donde se guarda el bloque de control de cada SF montado en el SO.

tabla de punteros (TP) *f* Estructura de datos interna del SO donde se guardan los punteros de lectura/escritura de los ficheros abiertos.

worst-fit Política de asignación del espacio libre de la memoria que otorga el hueco mayor de todos los disponibles.

write through Técnica que actualiza, sobre la memoria principal, las modificaciones que se hacen sobre la memoria caché en el mismo instante en el que se producen.

Bibliografía

Tanembaum, A. (2009). *Modern Operating Systems*. Prentice Hall.

Bovet, D.; Cesati, M. (2006). *Understanding the Linux Kernel*. O'Reilly.