# FICHEROS


1 CONCEPTO DE ARCHIVO ..... 1
2 METODOS DE ACCESO Y ASIGNACIÓN ..... 5
3 GESTIÓN DEL ESPACIO LIBRE ..... 12
4 DIRECTORIOS DE ARCHIVOS ..... 13
5 ARQUITECTURA DE UN SISTEMA DE ARCHIVOS ..... 20
6 SEGURIDAD Y PROTECCIÓN ..... 25
7 PLANIFICACIÓN DE LOS ACCESOS A DISCO ..... 33


Para la mayoría de los usuarios, el sistema de archivos es el aspecto más visible de un sistema operativo, pues proporciona el mecanismo para el almacenamiento y acceso en línea tanto de datos como de programas del S.O. y de todos los usuarios del sistema de computación

El sistema de archivos consiste en dos partes distintas: una colección de ***archivos***, cada uno de los cuales contienen datos relacionados, y una ***estructura de directorios***, que organiza todos los archivos del sistema y proporciona información de los mismos. Hoy los sistemas de archivos tienen una tercera característica, ***particiones***, que sirven para separar física o lógicamente grandes colecciones de directorios.

# 1 CONCEPTO DE ARCHIVO

La información se almacena en distintos tipos de soporte. El S.O. proporciona una visión lógica uniforme independiente de las propiedades físicas del dispositivo en el que está almacenado el archivo.

Pueden tener forma libre o rígida.

El concepto de archivo es muy general, casi siempre, los archivos representan programas (en formato fuente, objeto, ejecutables) y datos (formatos numérico, alfanumérico, alfabéticos o binarios, registros de nómina, imágenes gráficas, video, etc.).

Un archivo tiene una ***estructura*** definida que dependerá de su tipo.

La información que contienen es definida por el creador

## 1.1 Atributos de archivo

Se identifican por un nombre, dado por el usuario. Hay sistemas que son sensibles al tamaño. El nombre es independiente del proceso, del usuario y del sistema que lo creó.

Los atributos que generalmente tiene un fichero son:

- **Nombre**: simbólico, es la única información comprensible para el usuario.
- **Identificador**: etiqueta univoca, usualmente un número.
- **Tipo**: necesario para sistemas que reconocen distintos tipos de ficheros.
- **Ubicación**: puntero a un dispositivo para su localización en el mismo.
- **Tamaño**: dimensión del mismo.
- **Protección**: dice quien puede leer, escribir, ejecutar... etc.
- **Hora, fecha e identificación del usuario**: puede mantenerse para la creación, la modificación y el último uso.

Esta información se guarda en la estructura del directorio. Puede necesitarse entre 16 y más de 1000 bytes.

## *1.2 Operaciones con los archivos*

Un archivo es un **tipo de datos abstracto**. El S.O. va a poder:

- **Crear un archivo**. Obliga a dos operaciones, buscar espacio para el mismo e insertar una nueva entrada en la lista del directorio.
- **Escribir un archivo**. Para escribir se emite una llamada al sistema. Proporcionándole el nombre del fichero y la información que se escribirá en él. El sistema mantiene un ***puntero de escritura***.
- **Leer un archivo**. Para leer hacemos una llamada al sistema. Proporcionándole el nombre del fichero y el lugar de memoria en donde se colocará la información leída.
- **Reubicar dentro de un archivo**. Se busca en el directorio el fichero y se actualiza el puntero. Se conoce como ***búsqueda***.
- **Eliminar el archivo**. Se busca en el directorio el archivo y se libera el espacio y la entrada en el directorio.

Estas operaciones se pueden complementar con otras como ***añadir***, ***cambio de nombre*** y ***copia***.

Estas operaciones implican, casi siempre, la búsqueda en el directorio.

Para evitar esto algunos sistemas **abren** el archivo *open*() la primera vez y mantienen una **tabla de archivos abiertos.** Cuando se necesita una operación se consulta esta tabla, el **cierre** *close*() del fichero implica la eliminación de la entrada en la tabla, normalmente esta entrada también tiene información de la cantidad de procesos que han abierto el fichero. Las funciones *create*() y *delete*() son llamadas al sistema que funcionan con archivos cerrados.

Algunos sistemas abren implícitamente los archivos cuando se realiza la primera referencia a los mismos, se cierra implícitamente cuando termina el proceso que lo "abrió".

La función open() copia el nombre del archivo en la tabla de archivos abiertos, también puede aceptar información sobre el modo de acceso: creación, sólo lectura, lectura-escritura, sólo añadidos...etc. Este modo se compara con los permisos definidos para el fichero con objeto de permitir o no su uso.

La función open() devuelve un puntero que hace referencia a la entrada correspondiente dentro de la tabla de archivos abiertos. Este puntero y no el nombre del fichero es el que se utiliza para todas las operaciones de E/S.

La implementación de las operaciones open() y close() es más complicada en entornos donde se permite que varios procesos abran un mismo archivo simultáneamente. Normalmente el S.O. usa dos niveles de tablas internas: una para cada proceso y una tabla global del sistema.

En síntesis, cada fichero abierto tiene asociada la siguiente información:

- **Puntero al archivo.** En los sistemas que no incluyen el desplazamiento como parte de las llamadas al sistema leer y escribir, el sistema debe mantener una información de la posición de uso, como esa información es exclusiva de cada proceso usuario no puede mantenerse como atributo del archivo en disco.
- **Contador de aperturas del archivo.** Cada usuario actualiza esa información, cuando llega a cero, debe eliminarse esa entrada.
- **Ubicación del archivo en disco.** Con objeto de no tener que leer esta información del directorio cada vez que opera con el mismo.
- **Derechos de acceso.** Esta información se almacena en la tabla correspondiente a cada proceso.

Algunos S.O. ofrecen mecanismos para bloquear secciones de un fichero por parte de múltiples procesos, para compartir secciones de un archivo entre varios procesos, e incluso para establecer una correspondencia entre secciones de un archivo y la memoria en los sistemas de memoria virtual. Estos bloqueos se realizan con técnicas de concurrencia como las vistas para el problema de los lectores-escritores.

## *1.3 Tipos de archivos*

El contenido de un archivo está definido por su creador, pudiendo contener distintos tipos de información: programas fuentes, programas objetos, datos numéricos, etc.

Dependiendo del uso, los archivos tendrán una determinada estructura. Un archivo de texto será una serie de caracteres organizados en líneas y páginas, mientras que uno de un programa fuente será una serie de subrutinas y funciones con sentencias ejecutables y declaración de variables.

El sistema operativo puede tener conocimiento de estas distintas estructuras lógicas y considerar diferentes tipos de archivos. Esto permite al sistema operativo dar un mejor servicio. Por ejemplo, el sistema TOPS-20 distingue los archivos de los programas fuentes y objetos, de forma que cuando se quiere ejecutar un programa, comprueba si el archivo del programa fuente ha sido modificado desde la última compilación, si es así entonces vuelve a compilarlo antes de ejecutarlo. Esto garantiza que siempre se ejecuta la última versión, aunque al usuario se le olvidase compilar.

Las desventajas de tener conocimiento de los distintos tipos de archivos por parte del sistema operativo se pueden resumir en dos:

1. Un mayor tamaño del sistema operativo, ya que tendrá que tener el código para soportar cada uno de los tipos de archivos considerados.
2. Se tiene una gran rigidez, ya que sólo se pueden considerar los tipos de archivos definidos por el sistema. Si en algún momento se necesita crear un archivo diferente, se entraría en conflicto con el sistema operativo. Por ejemplo, si el sistema sólo soporta dos tipos de archivos, de texto (compuestos por caracteres ASCII) y ejecutables en binario, y se desea proteger el contenido de un archivo de texto encriptándolo, el

resultado es un archivo que no es una secuencia de caracteres ASCII (parece una secuencia de bits aleatorios) pero que tampoco es un archivo ejecutable, pudiendo provocar un mal funcionamiento de la gestión de archivos por parte del sistema operativo.

En sistemas operativos como Unix no se definen tipos de archivos, dejando plena libertad al usuario. En realidad en Unix se consideran los archivos como una secuencia de bytes cuya interpretación corre a cuenta de las aplicaciones, de esta forma se tiene una gran flexibilidad, pero con un soporte mínimo, de forma que los programas de aplicación tienen que incluir el código que interprete la estructura adecuada de los archivos. De hecho Unix sólo interpreta unos tipos especiales de archivos: los ***directorios*** (que se verán más adelante y que Unix los considera como archivos del sistema para el mantenimiento de la estructura del sistema de archivos) y los ***archivos especiales*** usados para modelar periféricos (impresoras, terminales, discos, redes, etc.). El resto son los denominados archivos regulares y corresponden a los que tienen información de los usuarios.

Es normal incluir el tipo como parte del nombre.

Un S.O que identifique todo este tipo de ficheros puede incluir rutinas que traten aspectos de los mismos.

# 1.4 Estructura de los archivos

Pudiera ser que los ficheros tuvieran una estructura reconocida por el S.O. pero esto es un problema porque impide el crear nuevas estructuras en los mismos y limita el uso del S.O. por ejemplo: fichero de texto y fichero binario ejecutable, esto trae problemas si queremos incluir un archivo cifrado para proteger nuestros datos, ¿Cómo lo tratamos?.

No obstante, ciertos archivos deben adaptarse a una estructura requerida, comprensible por el S.O., por ejemplo, los ejecutables necesitan ser identificados por el sistema para poder ubicarlos en la memoria y saber donde está la primera instrucción a ejecutar.

## 1.4.1. Estructura interna de los archivos

La estructura interna es otro aspecto a tener en cuenta por los S.O., un fichero de datos con un registro lógico muy dispar al registro físico puede dar lugar a un despilfarro de espacio o a un tratamiento adicional para conversión de una lectura física en otra lógica. ***Empaquetar*** es la operación de ubicar varios registros lógicos en uno físico.

Un problema que surge como consecuencia de esta no-coincidencia de tamaño físico y lógico es el de la ***fragmentación interna***.

El soporte fisico mas habitual es el disco, los discos se estructuran en cilindros, que a su vez, se componen de pistas y estas se dividen en sectores.

La unidad elemental de lectura/escritura se realiza sobre sectores. Un sector puede localizarse por sus tres coordenadas: cilindro, cara (o pista), y sector.

Para el acceso físico a los datos de un fichero, es necesario identificar las distintas posiciones físicas en las que se encuentra el mismo, para ello en el directorio de la unidad física de almacenamiento, disponemos de la información relativa a cada fichero, ahí leemos la posición inicial del fichero (primera unidad de asignación) y a partir de ahí seguimos leyendo guiándonos por la información que nos proporciona la tabla de asignación de ficheros (FAT, tiene distintos nombres y formas de acceso según el S.O.)

El S.O. gestiona esta información como una tabla unidimensional (por ejemplo una tabla de 5 millones de unidades de asignación o bloques) y para acceder a la información física convierten la tabla tridimensional (de posición física) en una tabla de bloques de disco unidimensional (de gestión lógica).

Supongamos un disco dividido en 100 cilindros, cada cilindro lo componen 10 pistas y cada pista tiene 20 sectores. Esto implica 100*10*20 = 20000 sectores en el disco. Si asumimos que cada unidad de asignación implica un sector, el S.O. tiene una tabla unidimensional de 20000 unidades de asignación para gestionar.

Para transformar una medida unidimensional en una tridimensional operamos de la siguiente forma:

Sean: s_p: sectores por pista p_c: pistas por cilindro

c: cilindro p: pista s: sector

Si quisiéramos grabar la unidad de asignación o bloque 16432 (b= 16432) tendríamos que obtener el cilindro de la siguiente forma:

$c = \text{fix}\left(\frac{b}{p_c * s_p}\right)$ Luego haríamos: $b = b \bmod (p_c * s_p)$

Y posteriormente obtendríamos la pista y el sector de la siguiente forma:

$p = \text{fix}\left(\frac{\text{b}}{\text{s_p}}\right)$ $s = b \bmod \text{ s_p}$

Por el contrario, si disponemos del sector, pista y cilindro, podemos obtener la unidad de asignación o bloque de la siguiente forma:

$b = s + s_p * (p + c * p_c)$

De esta gestión se encarga el sistema básico de archivos, ver Ilustración 4 Sistema de archivos en niveles

# 2 METODOS DE ACCESO Y ASIGNACIÓN

Para leer un fichero hay que acceder a él, algunos sistemas ofrecen un único método de acceder a los mismos, el secuencial.

## 2.1 Acceso secuencial

Se basa en el modelo de ficheros en cinta: los bloques son accedidos en orden. Para acceder al i_esimo bloque hay que haber accedido a todos los anteriores. Es el más comúnmente usado.

## 2.2 Acceso directo

Se basa en el modelo de fichero en disco. También se conoce como ***acceso relativo***. El fichero se ve como una secuencia numerada de bloques de registro. Esto permite explorar un fichero muy grande con muy poca E/S.

Las operaciones sobre ficheros son ahora leer ***n***, escribir **n**, posicionarse en ***n***, leer el siguiente, escribir el siguiente... etc. Siempre **n** es posición relativa al comienzo del fichero.

No todos los sistemas soportan los dos métodos de acceso.

Algunos obligan a que un fichero quede definido como secuencial o directo en el momento de crearse.

## 2.3 Otros métodos de acceso

Otros métodos requieren el empleo de índices. Son métodos basados en el acceso directo, hacen uso de índices para localizar la posición específica dentro del fichero. Tienen el problema de que necesitan un fichero adicional para los índices, llevar a cabo la gestión de los mismos y, por consiguiente, un tiempo de acceso mas largo que el directo.

Son ejemplos de este método los desarrollados por IBM (ISAM, index sequential access method), Arboles B+... etc.

El modo de acceso y el de organización (se verá mas adelante) se complementa.

## 2.4 METODOS DE ASIGNACIÓN

El acceso directo inherente a la naturaleza de los discos nos proporciona una gran flexibilidad a la hora de implementar los archivos. En casi todos los casos, habrá múltiples archivos almacenados en el mismo disco. El principal problema es cómo asignar el espacio a esos archivos de modo que el espacio de disco se utilice eficazmente y que se pueda acceder a los archivos de forma rápida. Hay tres métodos principales de asignación del espacio de disco que se utilizan de manera común: asignación contigua, enlazada e indexada. Cada método tiene sus ventajas y desventajas

### 2.4.1. Asignación contigua

La **asignación contigua** requiere que cada archivo ocupe un conjunto de bloques contiguos en el disco. Las direcciones de disco definen una ordenación lineal del disco. Con esta ordenación, suponiendo que sólo haya una tarea accediendo al disco, acceder al bloque b + 1 después del bloque *b* normalmente no requiere ningún movimiento de cabezal. Cuando hace falta un movimiento de cabezal (para pasar del último sector de un cilindro al primer sector del siguiente cilindro), el cabezal sólo necesitará moverse de una pista a la siguiente, por tanto, el número de reposicionamientos del cabezal del disco requeridos para acceder a los archivos que están asignados de modo contiguo es mínimo, al igual que lo es el tiempo de búsqueda cada vez que hace falta reposicionar los cabezales. El sistema operativo VM/CMS de IBM utiliza una asignación contigua, debido precisamente a las altas prestaciones que proporciona.

La asignación contigua de un archivo está definida por la asignación de disco del primer bloque y por la longitud del archivo (en unidades de bloques). Si el archivo tienen *n* bloques de longitud y comienza en la ubicación *b,* entonces ocupará los bloques $b, b + 1, b + 2, ..., b+ n-1$ . La entrada de directorio de cada archivo indicará la dirección del bloque de inicio y la longitud del área asignada al archivo.

Acceder a un archivo que haya sido asignado de manera contigua resulta sencillo. Para el acceso secuencial, el sistema de archivos recuerda la dirección de disco del último bloque al que se haya hecho referencia y leerá el siguiente bloque cuando sea necesario. Para el acceso directo al bloque i de un archivo que comience en el bloque b, podemos acceder inmediatamente al bloque $b + i$. Así, la asignación contigua permite soportar tanto el acceso directo como el secuencial.

Sin embargo, la asignación contigua también tiene sus problemas. Una de las dificultades estriba en encontrar espacio para un nuevo archivo. El sistema elegido para gestionar el espacio libre determinará como se lleve a cabo esta tarea.

El problema de la asignación contigua puede verse como un caso concreto del problema general de **asignación dinámica del almacenamiento** que hemos analizado en la gestión de la memoria y que se refiere a cómo satisfacer una solicitud de tamaño *n* a partir de una lista de huecos libres. Las estrategias más comunes para seleccionar un hueco libre a partir del conjunto de huecos disponibles son las de:

- Primer ajuste
- Mejor ajuste
- Peor ajuste

Las simulaciones muestran que tanto la técnica de primer ajuste como la de mejor ajuste son más eficientes que la de peor ajuste tanto en términos de tiempo como en términos de utilización del almacenamiento. Las técnicas de primer ajuste y de mejor ajuste son muy similares en lo que se refiere a la utilización del espacio de almacenamiento, pero la técnica de primer ajuste es generalmente más rápida.

Todos estos algoritmos sufren el problema de la **fragmentación externa.** A medida que se asignan y borran archivos, el espacio libre del disco se descompone en pequeños fragmentos. Para solucionarlo hay que desfragmentar el disco. Hoy esta función la permiten los S.O. independientemente del tipo de asignación que utilicen; la razón es de facilitar el acceso al disco.

Otro problema de los mecanismos de asignación contigua es determinar cuánto espacio necesita para un archivo. En el momento de crear el archivo, será necesario determinar la cantidad total de espacio que vamos a necesitar y asignar esa cantidad. ¿Cómo sabe el creador (programa o persona) del archivo qué tamaño va a tener éste? En algunos casos, esta determinación puede ser bastante simple (por ejemplo, cuando se copia un archivo existente); sin embargo, en general, el tamaño de un archivo de salida puede ser difícil de estimar. Si asignamos demasiado poco espacio a un archivo, podemos encontrarnos con que el archivo no puede ampliarse. Especialmente con las estrategias de asignación de mejor ajuste, puede que el espacio situado a ambos lados del archivo ya esté siendo utilizado, por tanto, no podemos ampliar el tamaño del archivo sin desplazarlo a otro lugar. Entonces, existen dos posibilidades. La primera es terminar el programa de usuario, emitiendo el apropiado mensaje de error. El usuario deberá entonces asignar más espacio y volver a ejecutar el programa. Estas ejecuciones repetidas pueden ser costosas; para prevenirlas, el usuario sobreestimará normalmente la cantidad de espacio necesaria, lo que dará como resultado un desperdicio de espacio considerable. La otra posibilidad consiste en localizar un hueco de mayor tamaño, copiar el contenido del archivo al nuevo espacio y liberar el espacio anterior. Esta serie de acciones puede repetirse siempre y cuando exista suficiente espacio, aunque pueden consumir bastante tiempo. Sin embargo, con este método nunca necesario informar explícitamente al usuario acerca de lo que está sucediendo; el sistema continuará operando a pesar del problema, aunque lo hará de forma cada vez más lenta.

Incluso si la cantidad total de espacio necesario para un archivo se conoce de antemano, el mecanismo de preasignación puede ser poco eficiente. A un archivo que crezca lentamente a lo largo de un período muy dilatado (meses o años) será necesario asignarle suficiente espacio para su tamaño final, aun cuando buena parte de ese espacio vaya a estar sin utilizar durante un largo tiempo. Por tanto, el archivo tendrá un alto grado de fragmentación interna.

Para minimizar estos problemas, algunos sistemas operativos utilizan un esquema modificado de asignación contigua, mediante el que se asigna inicialmente un bloque contiguo de espacio y,

posteriormente, si dicho espacio resulta no ser lo suficientemente grande, se añade otra área de espacio contiguo, denominado **extensión.** La ubicación de los bloques de un archivo se registra entonces mediante una dirección y un número de bloques, junto con un enlace al primer bloque de la siguiente extensión. En algunos sistemas, el propietario del archivo puede establecer el tamaño de la extensión, pero este modo de configuración puede hacer que disminuya la eficiencia si el propietario realiza una estimación incorrecta. La fragmentación interna puede continuar siendo un problema si las extensiones son demasiado grandes y la fragmentación externa puede llegar a ser un problema a medida que se asignen y desasignen extensiones de tamaño variable. El sistema de archivos comercial Veritas utiliza las extensiones para optimizar el funcionamiento; se trata de un sustituto de altas prestaciones para el sistema UFS estándar de UNIX.

### 2.4.2. Asignación enlazada

El mecanismo de **asignación enlazada** resuelve todos los problemas de la asignación contigua. Con la asignación enlazada, cada archivo es una lista enlazada de bloques de disco, pudiendo estar dichos bloques dispersos por todo el disco. El directorio contiene un puntero al primer y al último bloque de cada archivo. Por ejemplo, un archivo de cinco bloques podría comenzar en el bloque 9 y continuar en el bloque 16, luego en el 1, después en el bloque 10 y, finalmente, en el bloque 25. Cada bloque contiene un puntero al bloque siguiente.

Para crear un nuevo archivo, simplemente creamos una nueva entrada en el directorio. Con el mecanismo de asignación enlazada, cada entrada de directorio tiene un puntero al primer bloque de disco del archivo. Este puntero se inicializa con el valor nulo (el valor de puntero que indica el fin de la lista) para representar un archivo vacío. El campo de tamaño también se configura con el valor 0. Una escritura en el archivo hará que el sistema de gestión de espacio libre localice un bloque libre y la información se escribirá en este nuevo bloque y será enlazada al final del archivo. Para leer un archivo, simplemente leemos los bloques siguiendo los punteros de un bloque a otro. No hay ninguna fragmentación externa con asignación enlazada y puede utilizarse cualquiera de los bloques de la lista de bloques libres para satisfacer una solicitud. No es necesario declarar el tamaño del archivo en el momento de crearlo y el archivo puede continuar creciendo mientras existan bloques libres. En consecuencia, nunca es necesario compactar el espacio de disco.

Sin embargo, el mecanismo de asignación enlazada también tiene sus desventajas. El principal problema es que sólo se lo puede utilizar de manera efectiva para los archivos de acceso secuencial. Para encontrar el bloque i-ésimo de un archivo, podemos comenzar al principio de dicho archivo y seguir los punteros hasta llegar al bloque i-ésimo. Cada acceso a un puntero requiere una lectura de disco y algunos de ellos requerirán un reposicionamiento de cabezales. En consecuencia, resulta muy poco eficiente tratar de soportar una funcionalidad de acceso directo para los archivos de asignación enlazada.

Otra desventaja es el espacio requerido para los punteros. Si un puntero ocupa 4 bytes de un bloque de 512 bytes, el 0,758 por ciento del espacio del disco se estará utilizando para los punteros, en lugar de para almacenar información útil. Cada archivo requerirá un poco más de espacio de lo que requeriría con otros mecanismos.

La solución usual para este problema consiste en consolidar los bloques en grupos, denominados clusters, y asignar clusters. Por ejemplo, el sistema de archivos puede definir un cluster como cuatro bloques y operar con el disco sólo en unidades de cluster. Los punteros utilizarán entonces un porcentaje mucho más pequeño del espacio de disco del archivo. Este método permite que el mapeo entre bloques lógicos y físicos siga siendo simple, pero mejora la tasa de transferencia del disco (porque hacen falta menos reposicionamientos de los cabezales) y reduce el espacio necesario para la asignación de bloques y para la gestión de la lista de bloques libres. El coste asociado con esta

técnica es un incremento en el grado de fragmentación interna, porque ahora se desperdiciará más espacio cuando un *cluster* esté parcialmente lleno.

Otro problema de la asignación enlazada es la fiabilidad. Un error en el software del sistema operativo o un fallo del hardware del disco podrían hacer que se interpretara incorrectamente un puntero, este error, a su vez, podría hacer que un enlace apuntara a la lista de bloques libres o a otro archivo. Una solución parcial a este problema consiste en utilizar listas de elementos enlazadas, mientras que otra consiste en almacenar el nombre del archivo y el número de bloque relativo en cada bloque; sin embargo, estos esquemas requieren desperdiciar todavía más espacio en cada archivo

Una variante importante del mecanismo de asignación enlazada es la que se basa en el uso una **tabla de asignación de archivos** (FAT, file-allocation table). Este método simple, pero eficiente, de asignación del espacio del disco se utiliza en los sistemas operativos MS-DOS y OS/2. Una sección del disco al principio de cada volumen se reserva para almacenar esa tabla, que tiene una entrada por cada bloque del disco y está indexada según el número de bloque. La FAT se utiliza de forma bastante similar (conceptualmente hablando) a una lista enlazada. Cada entrada de directorio contiene el número de bloque del primer bloque del archivo. La entrada de la tabla indexada según ese número de bloque contiene el número de bloque del siguiente bloque del archivo. Esta cadena continua hasta el último bloque, que tiene un valor especial de fin de archivo como entrada de la tabla. Los bloques no utilizados se indican mediante un valor de tabla igual a 0. Para asignar un nuevo bloque a un archivo, basta con encontrar la primera entrada de la tabla con valor 0 y sustituir el anterior de fin de archivo por la dirección del nuevo bloque. Después, el 0 se sustituye por el valor de fin de archivo.

El esquema de asignación FAT puede provocar un número significativo de reposicionamiento de los cabezales del disco, a menos que la tabla FAT se almacene en caché. El cabezal del disco debe moverse al principio del volumen para leer la FAT y encontrar la ubicación del bloque en cuestión, después de lo cual tendrá que moverse a la ubicación del propio bloque. En el caso peor, ambos movimientos tendrá lugar para cada uno de los bloques. Una de las ventajas de este esquema es que se mejora el tiempo de acceso aleatorio, porque el cabezal del disco puede encontrara la ubicación de cualquier bloque leyendo la información de la FAT.

## 2.4.3. Asignación indexada

El método de la asignación enlazada resuelve los problemas de fragmentación externa y de declaración del tamaño que presentaba el método de la asignación contigua. Sin embargo, si no se utiliza una FAT, la asignación enlazada no puede soportar un acceso directo eficiente, ya que los punteros a los bloques están dispersos junto con los propios bloques por todo el disco y deben extraerse secuencialmente. El mecanismo de **asignación indexada** resuelve este problema agrupando todos los punteros en una única ubicación: el **bloque de índices.**

Cada archivo tiene su propio bloque de índices, que es una matriz de direcciones de bloques de disco. La entrada i-ésima del bloque de índices apunta al bloque í-ésimo del archivo. El directorio contiene la dirección del bloque de índices. Para localizar y leer el bloque i-ésimo, utilizamos el puntero contenido en la entrada i-ésima del bloque de índices.

Cuando se crea el archivo, se asigna el valor *nulo* a todos los punteros del bloque de índice. Cuando se escribe por primera vez en el bloque i-ésimo, se solicita un bloque al gestor de espacio libre y su dirección se almacena en la entrada i-ésima del bloque de índice.

El mecanismo de asignación indexada soporta el acceso directo, sin sufrir el problema de la fragmentación externa, ya que puede utilizarse cualquier bloque del disco para satisfacer una

solicitud de más espacio. Sin embargo, el mecanismo de asignación indexada sí que sufre el problema del desperdicio de espacio. El espacio adicional requerido para almacenar los punteros del bloque de índice es, generalmente, mayor que el que se requiere en el caso de la asignación enlazada. Considere un caso común en el que tenemos un archivo de sólo uno o dos bloques. Con el mecanismo de asignación enlazada, sólo perderemos el espacio de un puntero por cada bloque, mientras que con el mecanismo de asignación indexada, es preciso asignar un bloque de índice completo, incluso si sólo uno o dos punteros de este bloque van a tener un valor distinto del valor *nulo.*

Este punto nos plantea la cuestión de cuál debe ser el tamaño del bloque de índices. Cada archivo debe tener un bloque de índices, así que ese bloque deberá ser lo más pequeño posible. Sin embargo, si el bloque de índices es demasiado pequeño, no podrá almacenar suficientes punteros para un archivo de gran tamaño y será necesario implementar un mecanismo para resolver este problema. Entre los mecanismos que pueden utilizarse para ello se encuentran los siguientes:

- **Esquema enlazado.** Cada bloque de índices ocupa normalmente un bloque de disco. Por tanto, puede leerse y escribirse directamente. Para que puedan existir archivos de gran tamaño, podemos enlazar varios bloques de índices. Por ejemplo, un bloque de índices puede contener una pequeña cabecera que indique un nombre del archivo y un conjunto formado por las primeras 100 direcciones de bloque del disco. La siguiente dirección (la última palabra del bloque de índices) será el valor *nulo* (para un archivo de pequeño tamaño) o un puntero a otro bloque de índice (para un archivo que necesite mas bloques de los apuntados en un bloque).
- **Índice multinivel.** Una variante de la representación enlazada consiste en utilizar un bloque de índices de primer nivel para apuntar a un conjunto de bloques de índice de segundo nivel, que a su vez apuntarán a los bloques del archivo. Para acceder a un bloque, el sistema operativo utiliza el índice de primer nivel para encontrar un bloque de índices de segundo nivel y luego emplea dicho bloque para hallar el bloque de datos deseado. Esta técnica podría ampliarse hasta un tercer o cuarto nivel, dependiendo del tamaño máximo de archivo que se desee. Con bloques de 4096 bytes, podríamos almacenar 1024 índices de 4 bytes en un bloque de índices; dos niveles de índice nos permitirían direccionar 1.048.576 bloques de datos, lo que equivale a un tamaño de archivo de hasta 4 GB.
- **Esquema combinado.** Otra alternativa, utilizada con el sistema **UFS**, consiste en mantener, por ejemplo, los primeros 13 punteros del bloque de índice en el nodo del archivo. Los primeros 10 de estos punteros hacen referencia a **bloques directos,** es decir, contienen la dirección de una serie de bloques que almacenan los datos del archivo. De este modo, los datos para los archivos de pequeño tamaño (de no más de 10 bloques) no necesitan un bloque de índices separado. Si el tamaño de bloque es de 4 KB, podrá accederse directamente a 40 KB de datos. Los siguientes tres punteros hacen referencia a **bloques indirectos.** El primero apunta a un **bloque indirecto de un nivel,** que es un bloque de índice que no contiene datos sino las direcciones de otros bloques que almacenan los datos. El segundo puntero hace referencia a un **bloque doblemente indirecto,** que contiene la dirección de un bloque que almacena las direcciones de otras series de bloques que contienen punteros a los propios bloques de datos. El último puntero contiene la dirección de un **bloque triplemente indirecto.** Con este método, el número de bloques que pueden asignarse a un archivo excede de la cantidad de espacio direccionable por los punteros de archivo de 4 bytes que se utiliza en muchos sistemas operativos. Un puntero de archivo de 32 bits sólo permite direcciona $2^{32}$ bytes, es decir, 4 GB. Muchas implementaciones de **UNIX**, incluyendo Solaris y **AIX** de IBM, soportan ahora punteros de archivo de hasta 64 bits. Los punteros de este tamaño per-

miten que los archivos y los sistemas de archivos tengan un tamaño del orden de los terabytes.

Los esquemas de asignación indexados sufren de los mismos problemas de rendimiento que el mecanismo de asignación enlazada. Específicamente, los bloques de índice pueden almacenarse en memoria caché, pero los bloques de datos pueden estar distribuidos por todo un volumen,

### 2.4.4. Prestaciones

Los métodos de asignación que hemos expuesto varían en lo que respecta a su eficiencia de almacenamiento y a los tiempos de acceso a los bloques de datos. Ambos criterios son importantes a la hora de seleccionar el método o métodos apropiados para implementarlos en un sistema operativo. Antes de seleccionar un método de asignación, necesitamos determinar cómo se utilizará el sistema: un sistema en el que el acceso sea preferentemente secuencial no debe utilizar el mismo método que otro donde el acceso sea fundamentalmente aleatorio.

Para cualquier tipo de acceso, el mecanismo de asignación contigua sólo requiere un acceso para extraer un bloque de disco. Puesto que podemos mantener fácilmente la dirección del archivo en memoria, podemos calcular inmediatamente la dirección de disco del bloque i-ésimo (o siguiente bloque) y leerlo directamente.

En el mecanismo de asignación enlazada, también podemos mantener en memoria la dirección del siguiente bloque y leerlo directamente. Este método resulta adecuado para el acceso secuencial, pero para el acceso directo, un acceso al bloque i-ésimo puede requerir i lecturas de disco.

Este problema muestra por qué no debe utilizarse el mecanismo de asignación enlazada para las aplicaciones que requieran acceso directo.

Como resultado, algunos sistemas soportan los archivos de acceso directo utilizando un mecanismo de asignación contigua y emplean el mecanismo de asignación enlazada para el acceso secuencial. Para estos sistemas, ***debe declararse el tipo de acceso*** que va a realizarse en el momento de crear el archivo. Los archivos creados para acceso secuencial estarán enlazados y no podrán utilizarse con acceso directo. Los archivos creados para acceso directo serán contiguos y podrán soportar tanto el acceso directo como el acceso secuencial, pero será necesario declarar su longitud máxima en el momento de crear el archivo. En este caso, el sistema operativo deberá disponer de las estructuras de datos y los algoritmos apropiados para soportar ***ambos*** métodos de asignación. Los archivos pueden convertirse de un tipo a otro creando un nuevo archivo del tipo deseado, en el que se copiará el contenido del archivo antiguo. El archivo antiguo puede entonces borrarse, después de lo cual se renombrará el nuevo archivo.

El esquema de asignación indexada es más complejo. Si el bloque de índice ya está en memoria, puede realizarse el acceso directamente. Sin embargo, mantener en memoria el bloque de índice requiere un espacio considerable. Si este espacio de memoria no está disponible, podemos tener que leer el primer el bloque de índice y luego el bloque de datos deseado. Para un índice en dos niveles, puede que sean necesarias dos lecturas de bloques de índice. Para un archivo extremadamente grande, acceder a un bloque situado cerca del final del archivo podría requerir leer todos los bloques de índice antes de leer el bloque de datos necesario. Así, las prestaciones del mecanismo de asignación indexada dependerán de la estructura del índice, del tamaño del archivo y de la posición del bloque deseado.

Algunos sistemas combinan la asignación contigua con la asignación indexada, utilizando una asignación contigua para los archivos de pequeño tamaño (de hasta tres o cuatro bloques) y conmutando automáticamente a un mecanismo de asignación indexada si el tamaño del

archivo crece. Puesto que la mayoría de los archivos son pequeños y la asignación contigua resulta eficiente para los pequeños archivos, las prestaciones medias pueden ser bastante buenas.

# 3 GESTIÓN DEL ESPACIO LIBRE

Puesto que el espacio del disco está limitado, necesitamos reutilizar el espacio de los archivos borrados para los nuevos archivos, siempre que sea posible (los discos ópticos de una sola escritura sólo permiten una escritura en cada sector, por lo que dicha reutilización no es físicamente posible). Para controlar el espacio libre del disco, el sistema mantiene una **lista de espacio libre**La lista de espacio libre indica todos los bloques de disco **libres**, aquellos que no están asigna a ningún archivo o directorio. Para crear un archivo, exploramos la lista de espacio libre hasta obtener la cantidad de espacio requerida y asignamos ese espacio al nuevo archivo. A continuación, este espacio se elimina de la lista de espacio libre. Cuando se borra un archivo, su espacio de disco se añade a la lista de espacio libre.

## 3.1 Vector de bits

Recientemente, la lista de espacio libre se implementa como un **mapa de bits** o **vector de bits** Cada bloque está representado por 1 bit. Si el bloque está libre, el bit será igual a 1; si el bloque está asignado, el bit será 0.

Por ejemplo, considere un disco en el que los bloques 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17,18, 25, 26 y 27 están libres y el resto de los bloques están asignados. El mapa de bits de espacio libre sería

001111001111110001100000011100000...

La principal ventaja de este enfoque es su relativa simplicidad y la eficiencia que permite a la hora de localizar el primer bloque libre o ***n*** bloques libres consecutivos en el disco. De hecho, muchas computadoras suministran instrucciones de manipulación de bits que pueden utilizarse de manera efectiva para dicho propósito. Por ejemplo, la familia Intel a partir del 80386 y la familia Motorola a partir del 68020 (procesadores incorporados en los sistemas PC y Macintosh, respectivamente) tienen instrucciones que devuelven el desplazamiento dentro de una palabra del primer bit con el valor 1. Una técnica para localizar el primer bloque libre en un sistema que utilice un lector de bits para asignar el espacio de disco consiste en comprobar secuencialmente cada palabra del mapa de bits para ver si dicho valor es distinto de 0, ya que una palabra con valor 0 tendrá todos los bits iguales a 0 y representará un conjunto de bloques asignado. Cuando encuentra la primera palabra distinta de 0 se la explora en busca del primer bit 1, que se corresponderá con la ubicación del primer bloque libre. El cálculo del número de bloque será:

(número de bits por palabra) x (número de palabras de valor 0)
+ desplazamiento de primer bit 1

Desafortunadamente, los vectores de bit son ineficientes a menos que se mantenga el vector completo en la memoria principal (y se lo escriba en disco ocasionalmente para propósitos de recuperación). Mantener ese vector en la memoria principal es posible para los discos de pequeño tamaño, pero no necesariamente para los discos de tamaño más grande. Un disco de 1,3 GB con bloques de 512 bytes necesitaría un mapa de bits de más de 332 KB para controlar todos los bloques libres, aunque si agrupamos todos los bloques en *clusters* de cuatro se reduce este número a unos 83 KB por disco. Un disco de 40 GB con bloques de 1 KB requeriría más de 5 MB para almacenar el mapa de bits.

## 3.2 Lista enlazada

Otra técnica para la gestión del espacio libre consiste en enlazar todos los bloques de disco libres, manteniendo un puntero al primer bloque libre en ubicación especial del disco y almacenándolo en la memoria caché. Este primer bloque contendrá un puntero al siguiente bloque libre del disco, etc. Sin embargo, este esquema es poco eficiente; para recorrer la lista, debemos leer cada bloque, lo que requiere un tiempo sustancial de E/S. Afortunadamente, no resulta frecuente tener que recorrer la lista de espacio libre. Usualmente, el sistema operativo simplemente necesita un bloque libre para poder asignar dicho bloque a un archivo, por lo que se utiliza el primer bloque de la lista de espacio libre. El método FAT incorpora el control de los bloques libres dentro de la estructura de datos utilizada para el algoritmo de asignación; en este caso, no hace falta utilizar ningún método separado.

## 3.3 Agrupamiento

Una modificación de la técnica basada en una lista de espacio libre consiste en almacenar las direcciones de *n* bloques libres en el primer bloque libre. Los primeros *n - 1* de estos bloques estarán realmente libres. El último bloque contendrá las direcciones de otros *n* bloques libres, etc. De este modo, podrán encontrarse rápidamente las direcciones de un gran número de bloques libres, a diferencia del caso en que se utilice la técnica estándar de lista enlazada.

# 4 DIRECTORIOS DE ARCHIVOS

Los directorios son, básicamente, tablas simbólicas de archivos. En la Tabla 1 se muestra un ejemplo con algunas de las posibles entradas en la tabla. Una entrada típica del directorio puede contener la siguiente información:

1. Nombre, tipo y número de versión del archivo.
2. Puntero de acceso al archivo, dirección de comienzo en el disco.
3. Lista de atributos: tamaño, estructura, dueño, modos de protección, fecha de creación, fecha de la última copia de seguridad, fecha de la última modificación o referencia, etc.

| Nombre | Tipo | Versión | Puntero | Tamaño | Permisos | Fechas |
|---|---|---|---|---|---|---|
| libro | txt | 3 | 30125 | 50 | -rw-r--r-- | 12/12/94 10/01/05 |
| ordena | obj | 1 | 25432 | 120 | -rw-rw-rw- | 05/11/94 22/01/05 |
| ordena | exc | 1 | 10125 | 30 | -rwxrw-rw- | 01/01/95 01/10/05 |
| lista | txt | 2 | 65390 | 40 | -rw-r--r-- | 10/10/94 05/12/04 |
| libmat | lib | 4 | 00912 | 90 | -rw-rw-rw- | 01/03/94 05/08/04 |

**Tabla 1 Tabla de directorio**

En muchos sistemas, la tabla del directorio está dividida en dos (ver Tabla 2). En una sólo se mantienen los nombres de los archivos con un número de identificación, el cual da acceso a la otra que es donde se tiene el puntero de acceso al archivo y la lista de atributos.

La separación de la tabla de nombres de los archivos de la de descripción agiliza la gestión de los enlaces.

El número de directorios varía de un sistema a otro. Uno de los diseños más sencillos es el ***directorio de nivel único***: un único directorio contiene todos los archivos del sistema o volumen. Este tipo de organización presenta muchos inconvenientes, sobre todo en sistemas multiusuarios. Por ejemplo, hay que tener un especial cuidado para no duplicar los nombres de los archivos, que suele ser bastante problemático cuando hay varios usuarios que pueden usar archivos para propósitos análogos. Además se presentan más problemas para asignar protección a los archivos. Éste fue un sistema usado en los primeros microcomputadores.

| Nombre | ID |
|---|---|
| libro | 1 |
| ordena | 2 |
| ordena | 3 |
| lista | 4 |
| Libmat | 5 |

| ID | Tipo | Versión | Puntero | Tamaño | Permisos | Fechas |
|---|---|---|---|---|---|---|
| 1 | txt | 3 | 30125 | 50 | -rw-r--r-- | 12/12/04 10/01/07 |
| 2 | obj | 1 | 25432 | 120 | -rw-rw-rw- | 05111/04 22/01/07 |
| 3 | exe | 1 | 10125 | 30 | -rwxrw-rw- | 01/01/05 01/10/07 |
| 4 | txt | 2 | 65390 | 40 | -rw-r—r-- | 10/10/04 05/12/07 |
| 5 | lib | 4 | 00912 | 90 | -rw-rw-rw- | 01/03/04 05/08/07 |

**Tabla 2 Tabla de directorio dividida en dos**

Una mejora se obtiene al considerar un directorio por usuario. De esta forma se elimina el conflicto entre ellos. Pero los usuarios que tienen muchos archivos siguen teniendo un gran problema de organización de los mismos.

La solución es permitir una jerarquización total, es decir, crear un ***árbol de directorios***. De esta forma cada usuario dispone de su propia estructura jerárquica con tantos subdirectorios como necesite. Se puede tener una estructura como la del ejemplo de la Ilustración 1 , donde las entradas al directorio correspondiente tienen un atributo más que indica si esa entrada corresponde a un archivo o a un subdirectorio. Esta distinción se ha hecho en la Ilustración 1 marcando los directorios con rectángulos y los archivos con círculos.

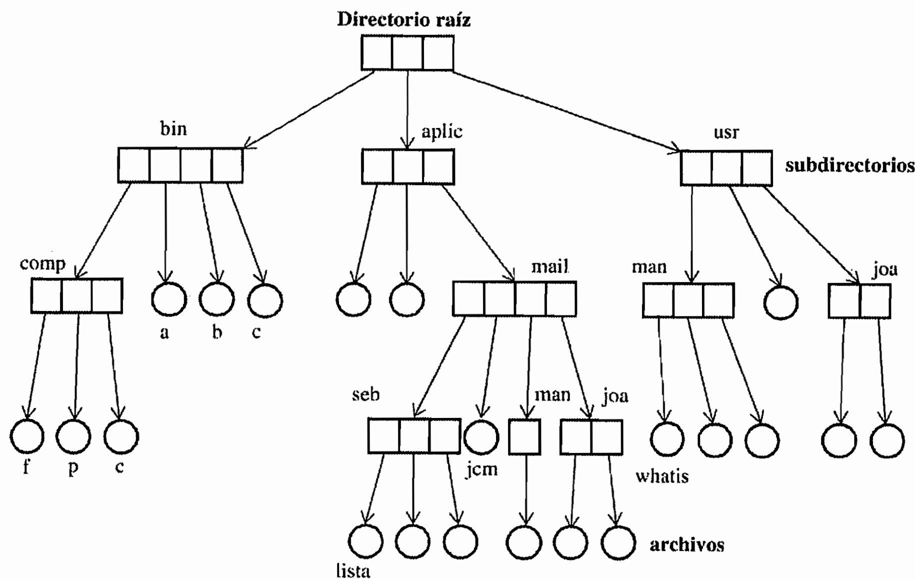



**Ilustración 1 Estructura jerárquica de directorios**

Normalmente, cada usuario tiene su "directorio inicial". Cuando el proceso de conexión de un usuario arranca (proceso ***login***), el sistema operativo busca en un archivo de "cuentas" (con los nombres de usuarios y sus claves de acceso) una entrada para este usuario. También hay almacenado en este archivo un puntero o el nombre del directorio inicial del usuario, que se registra en una variable especial de este usuario como el directorio actual. Cuando se hace una referencia a un archivo o directorio, éste se busca en el directorio actual. Si no se encuentra en él, el usuario debe cambiar el directorio actual o especificar el camino completo. Para cambiarlo se hace una llamada al sistema operativo indicando qué directorio se quiere que sea el actual.

Los nombres de los caminos pueden ser de dos tipos: completos o relativos. Un nombre de camino completo comienza en el directorio raíz y sigue hacia abajo en el árbol de directorios hasta el archivo indicado. Por ejemplo, en la Ilustración 1, un camino completo es

*/aplic/mail/seb/lista*

que indica que partiendo del directorio raíz, hay un directorio *aplic*, el cual tiene un subdirectorio *mail* y éste a su vez un subdirectorio *seb* que es en el que se encuentra el archivo *lista.* Suponiendo que el directo actual es */aplic/mail*, un camino relativo es *seb/lista* que indica que en el directorio actual hay un subdirectorio *seb* en el que se encuentra el archivo *lista.* Los nombres absolutos siempre comienzan en el directorio raíz y son únicos.

La mayoría de los sistemas operativos con estructura de directorios jerárquicos tienen dos entras especiales para cada directorio:

1. "." (punto): es una entrada para el propio directorio (con un puntero a si mismo).
2. ".." (punto-punto): es una entrada para el directorio padre (el que está por encima en la jerarquía

Esto está reflejado en la Ilustración 2, que corresponde a una estructura de directorios típica de un sistema Unix. Se puede usar "." y".." en los caminos relativos.

Por ejemplo, si el directorio actual es */usr/man*, copiar el archivo notas de este directorio en el directorio */usr/joa* se puede hacer de las siguientes formas:

cp *notas ../joa/notas*

cp *notas /usr/joa/notas*

cp */usr/man/notas /usr/joa/notas*

En el primer caso se han indicado los archivos origen y destino con caminos relativos al directorio actual, en el segundo caso se ha usado un camino relativo para el directorio actual y absoluto para el destino y en el tercer caso se han usado caminos absolutos para los dos.

Otro problema es donde y como ubicar los directorios. En un principio se puede pensar hacerlo en la memoria principal, sobre todo por cuestiones de rapidez, pero los directorios suelen ser muy grandes, por lo que se adoptó la solución de tenerlos en el propio disco. Esto facilita el montar y desmontar dispositivo, ya que el acceso a los archivos está indicado en él mismo. Además, para uniformizar las operaciones en disco, es habitual considerar los directorios como archivos. De esta forma los directorios son archivos que tiene una lista de todos los archivos. El problema planteado con esta situación es la localización del directorio raiz al arrancar el sistema. La solución es situar el directorio raíz en una dirección conocida por el volumen desde el que se arranca el sistema.

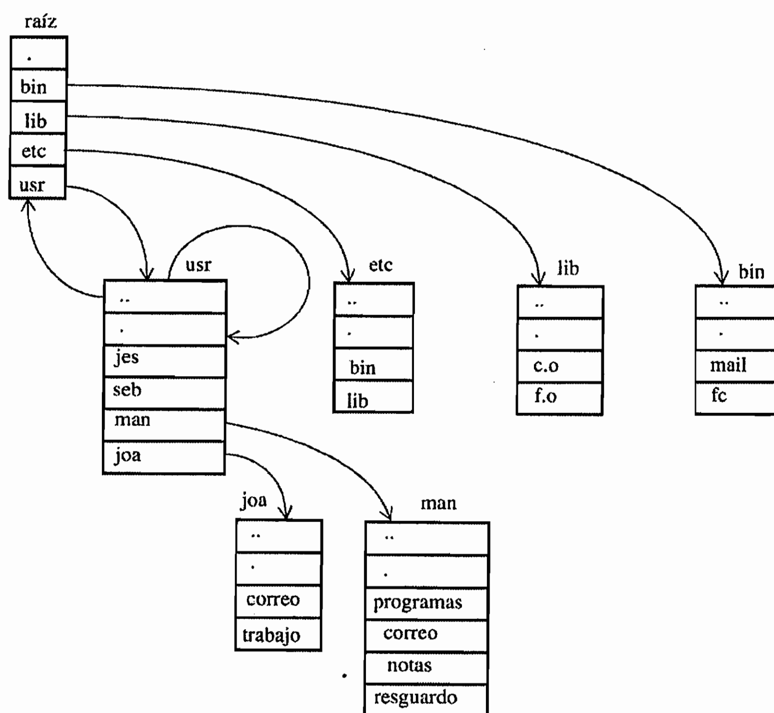



**Ilustración 2 Estructura jerárquica de directorios de un sistema Unix**

## *4.1 Directorios con forma de grafos acíclicos*

Cuando varios usuarios están trabajando en un mismo proyecto y quieren tener todos ellos acceso a ficheros compartidos se puede crear un subdirectorio en el que están accesibles dichos ficheros. Ambos usuarios pueden desear tener accesible ese subdirectorio. Se debe tener en cuenta que sólo existe un directorio real y que no es lo mismo que existan copias de un mismo subdirectorio.

El grafo acíclico permite que un subdirectorio se encuentre presente en diferentes directorios de diferentes usuarios.

Los ficheros y subdirectorios compartidos pueden implementarse de diversas maneras. Una forma corriente es a través de ***enlaces***. Un enlace es un puntero a otro fichero o directorio. Un enlace puede implementarse como un ***enlace simbólico*** o como un ***enlace duro***. En el primer caso se trata de un fichero que tiene el path al otro directorio, en el segundo caso se trata de un enlace normal al otro directorio.

Otra solución consiste en duplicar la información.

Una estructura en grafo acíclico es más flexible que una estructura en árbol pero hay que considerar varios problemas.

Un fichero ahora puede tener varios caminos completos. Eso implica que el mismo fichero puede tener diferentes nombres. Si tratamos de recorrer todo el sistema de ficheros (para obtener estadísticas...) llegamos al mismo sitio por diferentes caminos y esto distorsiona los dados de gestión del sistema.

Otro problema lo tenemos con el borrado de ficheros. ¿Cuando puede utilizarse de nuevo el espacio asignado a un fichero compartido?.

Si se borra un fichero apuntado por otros, puede suceder que queden punteros a objetos inexistentes y si este espacio se utiliza por otros pueden apuntar a ficheros no deseados.

Si la compartición se hace por enlaces simbólicos el problema se limita a eliminar el enlace.

Si se borra el fichero original, surge el problema con los enlaces simbólicos. Se puede hacer de forma que estos tengan referencias del origen y borrar estos enlaces en el momento en que se borra el fichero. Si no existen esas referencias, se pueden buscar los enlaces pero esto resulta costoso en tiempo. Podemos dejar los enlaces hasta que se haga una tentativa de utilizarlos. Así en el momento de utilizarlos indicamos que dicho fichero no existe. Aquí hay que tratar el problema de un fichero que se borra físicamente y se crea uno nuevo con el mismo nombre y continúan existiendo los enlaces del antiguo.

Otra alternativa consiste en preservar el fichero hasta que se borran todos los enlaces al mismo. Hay que implementar un mecanismo que cuente las referencias al mismo y sólo se borra cuando está nula esta lista de referencias. El problema es el tamaño variable de la misma. Se suele hacer con enlaces normales (duros). Ej.: UNIX.

## 4.2 Directorio con forma de grafo general

Es un problema el asegurarse de que no hay ciclos cuando se utiliza una estructura de grafo acíclico.

La ventaja de un grafo acíclico es la relativa simplicidad de los algoritmos necesarios para recorrerlo y para determinar que no quedan referencias a un fichero.

Si accedemos a un directorio en una búsqueda general de un fichero y no está en él, es conveniente no volver sobre el mismo cuando accedamos desde un enlace a hacer la misma operación.

Si se permite la existencia de ciclos, esta operación se puede hacer indefinida por la búsqueda sobre los enlaces que no acaban de explorar los directorios al volver sobre sí mismos. Esto puede suceder si el algoritmo no está bien definido y no prevé estas situaciones.

Un problema similar se produce al borrar un fichero. En este caso puede suceder que el contador de referencias no esté a cero porque hay una referencia de un directorio que está dentro de sí mismo,

esto implica que no puede borrarse porque puede ser utilizado por otro y en realidad este otro ya no existe.

En este caso es necesario realizar una ***recogida de residuos*** para determinar cuando se ha borrado la última referencia y puede volver a utilizarse el espacio disponible. Este sistema trabaja identificando primero todo lo que puede ser accedido y después la lista de espacios libres, lo que no figura en uno u otro sitio puede ser borrado. La función fsck() de UNIX se encarga de ello.

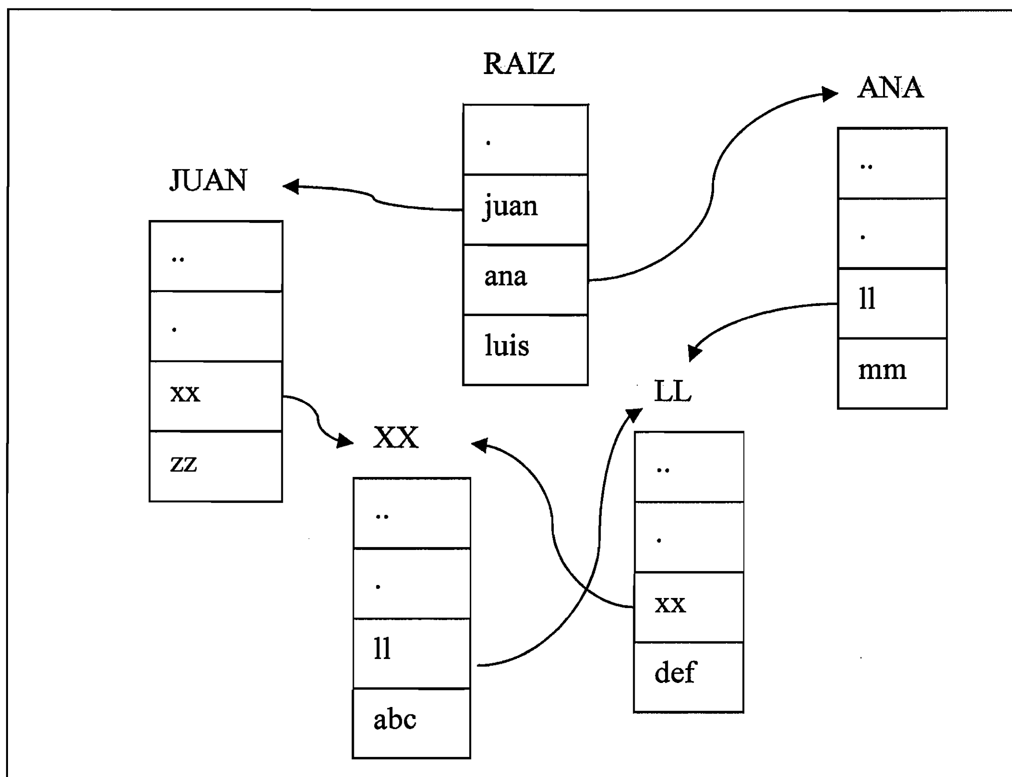



**Ilustración 3 grafo general**

## *4.3 Implementación de directorios*

La selección de los algoritmos de asignación de directorios y gestión de directorios afecta significativamente a la eficiencia, las prestaciones y la fiabilidad del sistema de archivos. Las opciones para implementar directorios son las que se ven a continuación.

### 4.3.1. Lista lineal

El método más simple para implementar un directorio consiste en utilizar una lista lineal de nombres de archivos, con punteros a los bloques de datos. Este método es simple de programar, pero requiere mucho tiempo de ejecución. Para crear un nuevo archivo, debemos primero explorar el directorio para asegurarnos de que no haya ningún archivo existente con el mismo nombre, después, añadiremos una nueva entrada al final del directorio. Para borrar un archivo, exploraremos el directorio en busca del archivo especificado y liberaremos el espacio asignado al mismo. Para reutilizar la entrada del directorio, podemos hacer varias cosas: podemos marcar la entrada como no utilizada (asignándole un nombre especial, como por ejemplo un nombre

completamente en blanco o utilizando un bit de usado-libre para cada entrada), o podemos insertarla en una lista de entradas libres de directorio. Una tercera alternativa consiste en copiar la última entrada del directorio en la ubicación que ha quedado libre y reducir la longitud del directorio. También puede utilizarse una lista enlazada para reducir el tiempo requerido para borrar un archivo.

La principal desventaja de una lista lineal de entradas del directorio es que, para localizar un archivo, se requiere realizar una búsqueda lineal. La información de directorio se utiliza frecuentemente y los usuarios notarán inmediatamente que el acceso a esa información es muy lento. De hecho, muchos sistemas operativos implementan una caché software para almacenar la información de directorio más recientemente utilizada. Cada acierto de caché evita la necesidad de volver a ver constantemente la información del disco. Una lista ordenada permite efectuar una búsqueda ordinaria y reduce el tiempo medio de búsqueda; sin embargo, el requisito de mantener la lista ordenada puede complicar los procesos de creación y borrado de archivos, ya que puede que tengamos que mover cantidades sustanciales de información de directorio para poder mantener el directorio ordenado. En este caso, podría sernos de ayuda utilizar una estructura de datos en árbol más sofisticada, como por ejemplo un árbol-B. Una ventaja de la lista ordenada es que puede generarse un listado ordenado del directorio sin ejecutar un paso separado de ordenación.

### 4.3.2. Tabla hash

Otro tipo de estructura de datos utilizada para los directorios de archivos son las **tablas *hash*.** Con este método, se almacenan las entradas de directorio en una lista lineal, pero también se utiliza una estructura de datos ***hash***. La tabla ***hash*** toma un valor calculado a partir del nombre del archivo y devuelve un puntero a la ubicación de dicho nombre de archivo dentro de la lista lineal. Por tanto, puede reducir enormemente el tiempo de búsqueda en el directorio. La inserción y el borrado son también bastante sencillas, aunque es necesario tener en cuenta la posible aparición de colisiones, que son aquellas situaciones en las que dos nombres de archivo proporcionan, al aplicar la función ***hash***, la misma ubicación dentro de la lista.

Las principales dificultades asociadas con las tablas ***hash*** son que su tamaño es, por regla general, fijo y que la función ***hash*** depende de dicho tamaño. Por ejemplo, supongamos que implementamos una tabla ***hash*** de prueba lineal que tenga 64 entradas. La función ***hash*** convertirá los nombres de archivo a enteros comprendidos en el rango de 0 a 63, utilizando por ejemplo el resto de una división por 64. Si después tratamos de crear otro archivo más, el número 65, deberemos agrandar la tabla ***hash*** del directorio, por ejemplo a 128 entradas. Como resultado, necesitaremos una nueva función ***hash*** que mapee los nombres de archivo al rango comprendido entre 0 y 127, y deberemos reorganizar las entradas de directorio existentes para reflejar los nuevos valores proporcionados por la función ***hash***.

Alternativamente, podemos usar una tabla ***hash*** con desbordamiento encadenada. Cada entrada ***hash*** puede ser una lista enlazada en lugar de un valor individual y podemos resolver las colisiones añadiendo la nueva entrada a esa lista enlazada. Este mecanismo puede ralentizar algo las búsquedas, ya que la búsqueda de un nombre requerirá recorrer una lista enlazada de todas las entradas de la tabla que se mapeen sobre el mismo valor ***hash***. De todos modos, este método será normalmente mucho más rápido que una búsqueda lineal a través de todo el directorio.

# 5 ARQUITECTURA DE UN SISTEMA DE ARCHIVOS

Se utilizan varias estructuras en disco y en memoria para implementar un sistema de archivos. Estas estructuras varían dependiendo del sistema operativo y del sistema de archivos, aunque hay algunos principios básicos que son de aplicación general.

En el disco, el sistema de archivos puede contener información acerca de cómo iniciar un sistema operativo que esté almacenado allí, acerca del número total de bloques, del número y la ubicación de los bloques libres, de la estructura de directorios y de los archivos individuales. De forma general podemos tener:

- Un **bloque de control de arranque** (por cada volumen) puede contener la información que el sistema necesita para iniciar un sistema operativo a partir de dicho volumen. Si el disco no contiene un sistema operativo, este bloque puede estar vacío. Normalmente, es el primer bloque del volumen. En UFS, se denomina **bloque de inicio.** En NTFS, se denomina **sector de arranque de la partición.**
- Un **bloque de control de volumen** (por cada volumen) contiene detalles acerca del volumen (o partición), tales como el número de bloques que hay en la partición, el tamaño de los bloques, el número de bloques libres y los punteros de bloques libres, así como un contador de bloques de información `FCB` libres y punteros `FCB`. En **`UFS`**, esto se denomina superbloque, en **`NFTS`** esta información se almacena en la **tabla maestra de archivos.**
- Se utiliza una estructura de directorios por cada sistema de archivos para realizar los archivos. En **`UFS`**, esto incluye los nombres de los archivos y los nombres de **inodo** asociados. En **`NFTS`**, esta información se almacena en la **tabla maestra de archivos.**
- Un bloque **`FTB`** por cada archivo contiene numerosos detalles acerca del archivo, incluyendo los permisos correspondientes, el propietario, el tamaño y la ubicación de los bloques de datos. En **`UFS`**, esta estructura se denomina inodo. En **`NFTS`**, esta información se almacena dentro de la tabla maestra de archivos, que utiliza una estructura de base de datos relacional, con una fila por cada archivo.

La información almacenada en memoria se utiliza tanto para la gestión de un sistema de archivos como para la mejora del rendimiento mediante mecanismos de caché. Los datos se cargan en el momento del montaje y se descartan cuando el dispositivo se desmonta. Las estructuras existentes (tabla de montaje) pueden incluir las que a continuación se describen por cada volumen:

- Una caché de la estructura de directorios en memoria almacena la información relativa a los directorios a los que se ha accedido recientemente (para los directorios en los que hay montar los volúmenes, puede contener un puntero a la tabla del volumen).
- La **tabla global de archivos abiertos** contiene una copia del `FCB` de cada archivo abierto además de otras informaciones.
- La **tabla de archivos abiertos de cada proceso** contiene un puntero a la entrada apropiada de la entrada global de archivos abiertos, así como otras informaciones adicionales.

Para crear un nuevo archivo, un programa de aplicación llama al sistema lógico de archivos. El sistema lógico de archivos conoce el formato de las estructuras de directorio y, para crear un

nuevo archivo, asigna un nuevo **FCB** (alternativamente, si la implementación del sistema de archivos crea todos los bloques FCB en el momento de creación del sistema de archivos, se asignara un **FCB** a partir del conjunto de bloques **FCB** libres). El sistema lee entonces el directorio apropiado en la memoria, lo actualiza con el nuevo nombre de archivo y el nuevo **FCB** y lo vuelve a escribir en el disco. En la Tabla 3, se muestra un **FCB** típico.

Algunos sistemas operativos, incluyendo UNIX, tratan a los directorios exactamente igual que a los archivos, salvo porque se tratará de un archivo cuyo campo de tipo indicará que es un directorio. Otros sistemas operativos, incluyendo Windows NT, poseen llamadas al sistema diferentes para archivos y directorios

| Permisos del archivo |
|---|
| Fechas del archivo |
| Propietario del archivo, grupo, ACL |
| Tamaño del archivo |
| Bloque de datos del archivo o puntero a los bloques de datos |

**Tabla 3 Bloque de control de archivo típico**

Una vez que hemos creado un archivo, podemos utilizarlo para **E/S**. Pero primero, sin embargo, es necesario *abrirlo.* La llamada open () pasa un nombre de archivo al sistema de archivos. La llamada al sistema open () primero busca en la tabla global de archivos abiertos para ver si el archivo está siendo ya utilizado por otro proceso. En caso afirmativo, se crea una entrada en la tabla de archivos abiertos del proceso que apunte a la tabla global de archivos abiertos existente. Este algoritmo puede ahorrarnos una gran cantidad de trabajo. Cuando un archivo está abierto, se busca en la estructura de directorios para encontrar el nombre de archivo indicado. Usualmente, se almacena parte de la estructura del directorio en una caché de memoria para acelerar las operaciones de directorio. Una vez encontrado el archivo, el **FCB** se copia en la tabla global de archivos abiertos existente en la memoria. Esta tabla no sólo almacena el **FCB,** sino que también controla el número de procesos que han abierto el archivo.

A continuación, se crea una entrada en la tabla de archivos abiertos del proceso, con un puntero a la entrada de la tabla global de archivos abiertos y algunos otros campos de información. Estos otros campos de información pueden incluir un puntero a la ubicación actual dentro del archivo (para la siguiente operación read() o write()) y el modo de acceso en que se ha abierto el archivo. La llamada open () devuelve un puntero a la entrada apropiada de la tabla de archivos abiertos del proceso; todas las operaciones de archivo se realizan a partir de ahí mediante este puntero. El nombre del archivo puede no formar parte de la tabla de archivos abiertos, ya que el sistema no lo utiliza para nada una vez que se ha localizado el FCB apropiado en el disco. Sin embargo, lo puede almacenar en caché para ahorrar tiempo en las subsiguientes aperturas del mismo archivo. El nombre que se proporciona a esas entradas varía de unos sistemas a otros. En los sistemas UNIX se utiliza el término inglés **file descriptor,** mientras que Windows prefiere el término **file handle**; en castellano, utilizamos el término **descriptor de archivo.** En consecuencia, hasta que se cierre el archivo, todas las operaciones con el archivo se llevan a cabo mediante el descriptor de archivo contenido en la tabla de archivos abierto.

Cuando un proceso cierra el archivo, se elimina la entrada de la tabla del proceso y se decrementa el contador de aperturas existente en la tabla global. Cuando todos los usuarios que hayan abierto el

archivo lo cierren, los metadatos actualizados se copian en la estructura de directorio residente en el disco y se elimina la entrada de la tabla global de archivos abiertos.

Algunos sistemas complican este esquema todavía más, utilizando el sistema de archivos como interfaz para otros aspectos del sistema, como por ejemplo la comunicación por red. Como ilustración, en **UFS**, la tabla global de archivos abiertos almacena los inodos y otras informaciones para los archivos y directorios, pero también alberga información similar para las conexiones de red y para los dispositivos. De esta forma, puede usarse un mismo mecanismo para múltiples propósitos.

No deben infravalorarse los aspectos de almacenamiento en caché de las estructuras de los sistemas de archivos. La mayoría de los sistemas mantienen en la memoria toda la información acerca de un archivo abierto, salvo los propios bloques de datos. El sistema BSD UNIX resulta bastante típico en su uso de cachés para tratar de ahorrar operaciones de E/S de disco. Su tasa media de aciertos de caché, que es del 85 por ciento, muestra que merece la pena implementar estas técnicas.

# 5.1 Estructura de un sistema de archivos

Como hemos visto, el sistema de archivos proporciona el mecanismo para el almacenamiento en línea y para el acceso a los contenidos de los archivos, incluyendo datos y programas. El sistema de archivos reside permanentemente en ***almacenamiento secundario***, que está diseñado para albergar de manera permanente una gran cantidad de datos. Este apartado se ocupa principalmente de los temas relativos al almacenamiento de archivos y al acceso a archivos en el medio más común de almacenamiento secundario, que es el disco.

Los discos constituyen el principal tipo de almacenamiento secundario para mantener sistemas de archivos. Tienen dos características que los convierten en un medio conveniente para el almacenamiento de múltiples archivos:

1. El disco puede ser reescrito de manera directa; es posible leer un bloque del disco, modificar el bloque y volverlo a escribir en el mismo lugar.
2. Con un disco, se puede acceder directamente a cualquier bloque de información que contenga. Por tanto, resulta simple acceder a cualquier archivo de forma secuencial aleatoria y el conmutar de un archivo a otro sólo requiere mover los cabezales de lectura-escritura y esperar a que el disco termine de rotar.

En lugar de transferir un byte cada vez, las transferencias de E/S entre la memoria y el disco se realizan en unidades de ***bloques***, para mejorar la eficiencia de E/S. Cada bloque tiene uno o más sectores. Dependiendo de la unidad de disco, los sectores varían entre 32 bytes y 4096 bytes; usualmente, su tamaño es de 512 bytes.

Para proporcionar un acceso eficiente y cómodo al disco, el sistema operativo impone uno o más **sistemas de archivos,** con los que los datos pueden almacenarse, localizarse y extraerse fácilmente. Un sistema de archivos acarrea dos problemas de diseño bastante diferentes. El primer problema es definir qué aspecto debe tener el sistema de archivos para el usuario. Esta tarea implica definir un archivo y sus atributos, las operaciones permitidas sobre los archivos y la estructura de directorio utilizada para organizar los archivos. El segundo problema es crear algoritmos y estructuras de datos que permitan mapear el sistema lógico de archivos sobre los dispositivos físicos de almacenamiento secundario.

El propio sistema de archivos está compuesto, generalmente, de muchos niveles diferentes. La estructura que se muestra en la Ilustración 4 Sistema de archivos en niveles, es un ejemplo de

diseño en niveles; cada nivel del diseño utiliza las funciones de los niveles inferiores para crear nuevas funciones que serán, a su vez utilizadas por los niveles superiores a ese.

El nivel más bajo, el ***control de E/S,*** está compuesto por **controladores de dispositivo** y rutinas de tratamiento de interrupción, que se encargan de transferir la información entre la memoria principal y el sistema de disco. Un controlador de dispositivo puede considerarse una especie de traductor: su entrada está compuesta por comandos de alto nivel tales como "extraer bloque 123"; su salida consta de instrucciones de bajo nivel específicas del hardware que son utilizadas por la controladora hardware, que es quien establece la interfaz del dispositivo de E/S con el resto del sistema. El controlador de dispositivo escribe usualmente una serie de patrones de bits específicos en ubicaciones especiales de la memoria de la controladora de E/S, para decir a la controladora en qué ubicación del dispositivo debe operar y que acción debe llevar a cabo.

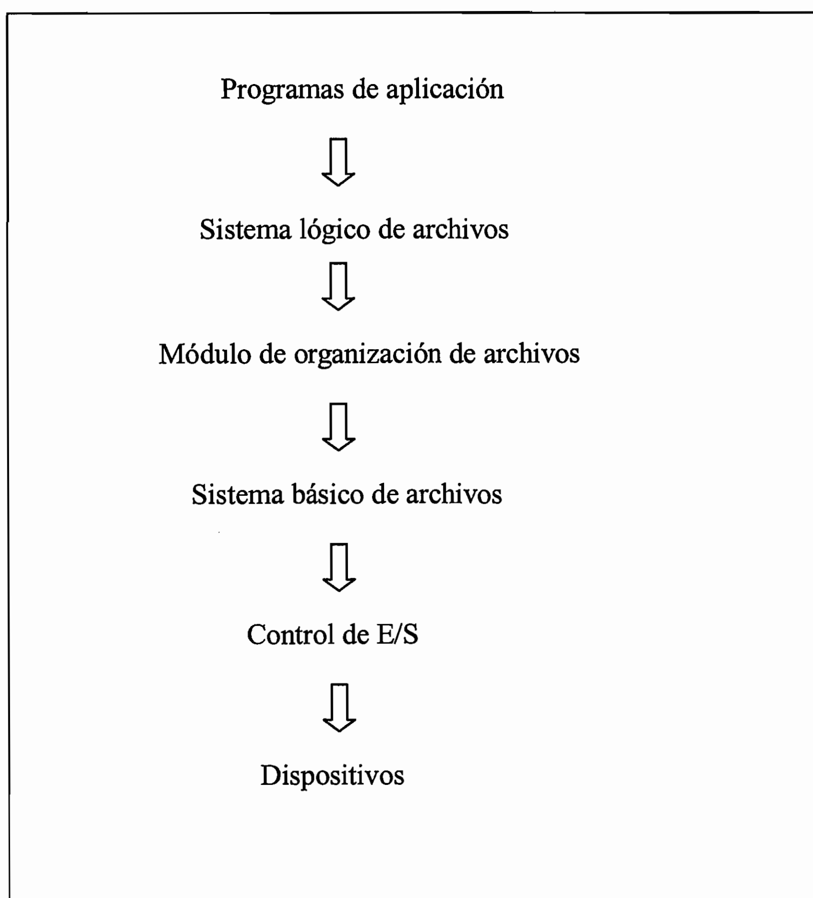



**Ilustración 4 Sistema de archivos en niveles**

El **sistema básico de archivos** sólo necesita enviar comandos genéricos al controlador de dispositivo apropiado, con el fin de leer y escribir bloques físicos en el disco. Cada bloque físico se identifica mediante su dirección de disco numérica (por ejemplo: unidad 1, cilindro 73, pista2, sector 10).

El **módulo de organización de archivos** tiene conocimiento acerca de los archivos y de sus bloques lógicos, así como de sus bloques físicos. Conociendo el tipo de asignación de archivos utilizado y la ubicación del archivo, el módulo de organización de archivos puede traducir direcciones lógicas de bloque a direcciones físicas de bloque, que serán las que envíe al sistema básico de archivos para que realice las necesarias transferencias. Los bloques lógicos de archivo están numerados desde 0 (o 1) a N. Puesto que los bloques físicos que contienen los datos no suelen corresponderse con los números lógicos de bloque, es necesaria una tarea de traducción para localizar cada bloque. El módulo de organización de archivos incluye también el gestor de espacio libre, que controla los bloques no asignados y proporciona dichos bloques al módulo de organización de archivos cuando así se solicita.

Finalmente, el **sistema lógico de archivos** gestiona la información de metadatos. Los metadatos incluyen toda la estructura del sistema de archivos, excepto los propios ***datos*** (es decir, el propio contenido de los archivos). El sistema lógico de archivos gestiona la estructura de directorio para proporcionar al módulo de organización de archivos la información que éste necesita, a partir de un nombre de archivo simbólico. Este nivel mantiene la estructura de los archivos mediante bloques de control de archivo. Un **bloque de control de archivo** (FCB, file-control block) contiene información acerca del archivo, incluyendo su propietario, los permisos y la ubicación del contenido del archivo. El sistema lógico de archivos también es responsable de las tareas de protección y seguridad.

Cuando se utiliza una estructura de niveles para la implementación de un sistema de archivos, se minimiza la duplicación de código. El código correspondiente al control de E/S y, en ocasiones, al sistema básico de archivos puede ser utilizado por múltiples sistemas de archivos. Cada sistema de archivos puede tener entonces su propio sistema lógico de archivos y su propio módulo de organización de archivos.

Hoy en día se utilizan muchos sistemas de archivos distintos. La mayoría de los sistemas operativos soportan más de un sistema. Por ejemplo, la mayoría de los CD-ROM están escritos en el formato ISO 9660, que es un formato estándar adoptado por consenso de los fabricantes de CD-ROM. Además de los sistemas de archivos para soportes extraíbles, cada sistema operativo tiene un (o más de un) sistema de archivos basado en disco. UNIX utiliza el **sistema de archivos** UNIX (UFS, UNIX file system), que está basado en el sistema FFS (Fast File System) de Berkeley. Windows NT, 2000 y XP soportan los formatos de sistema de archivos de disco FAT, FAT32 y NTFS (Windows NT File System), además de formatos de sistemas de archivos para CD-ROM, DVD y disquete. Aunque Linux soporta más de cuarenta sistemas de archivos distintos, el sistema de archivos estándar en Linux se denomina **sistema de archivos extendido,** siendo las versiones más comunes ext2 y ext3. También existen sistemas de archivos distribuidos, en los que un sistema de archivos que reside en un servidor puede montarse en uno o más clientes.

# 6 SEGURIDAD Y PROTECCIÓN

Uno de los mayores problemas que se presenta al almacenar información en un computador es la seguridad de la misma, teniéndose que idear mecanismos que protejan esta información tanto de daños físicos como de accesos inadecuados o mal intencionados.

A menudo, los conceptos de seguridad y protección se utilizan de forma indistinta y abarcan diferentes problemas. El término seguridad se suele referir al problema general y el término mecanismo de protección a los procedimientos específicos utilizados por el sistema operativo para asegurar la información del computador.

La fiabilidad se logra haciendo copias de seguridad. Muchos sistemas operativos hacen copias de seguridad, de forma automática, a cintas de sus ficheros en disco. Los daños pueden producirse por problemas hardware o software.

La protección es diferente si se trata de sistemas monousuario o multiusuario. En el primer caso el problema es sencillo. En multiusuario son necesarios otros sistemas.

## 6.1 Tipos de acceso

Podemos hacer un control total impidiendo el acceso o no hacer ningún control. Entre los dos extremos tenemos **el acceso controlado**. Hay varios tipos de operaciones que pueden controlarse:

***Lectura, Escritura, Ejecución., Adición, Borrado, Listado***

Otras operaciones, como el renombrado, el copiado y la edición de archivos también pueden controlarse.

## 6.2 Control de acceso

La técnica más común para resolver el problema de la protección consiste en hacer que el acceso dependa de la identidad del usuario. Los diferentes usuarios pueden necesitar diferentes tipos acceso a un archivo o directorio. El esquema más general para implementar un acceso dependiente de la identidad consiste en asociar con cada archivo y directorio una **lista de control** de acceso (ACL, access-control list) que especifique los nombres de usuario y los tipos de acceso que se permiten para cada uno. Cuando un usuario solicita acceder a un archivo concreto, el sistema operativo comprueba la lista de acceso asociada con dicho archivo; si dicho usuario está incluido en la lista para el tipo de acceso solicitado, se permite el acceso. En caso contrario, se producirá una violación de la protección y se denegará al trabajo de usuario el acceso al archivo.

Esta técnica tiene la ventaja de permitir la implementación de complejas metodologías de acceso. El problema principal con las listas de acceso es su longitud. Si queremos permitir que todo el mundo lea un archivo, deberemos enumerar todos los usuarios que disponen de ese acceso de lectura. Esta técnica tiene dos consecuencias poco deseables:

- Construir dicha lista puede ser una tarea tediosa y poco gratifícame, especialmente si no sabemos de antemano la lista de usuarios del sistema.
- La entrada de directorio, que anteriormente tenía un tamaño fijo, ahora tendrá que ser de tamaño variable, lo que requiere mecanismos más complejos de gestión del espacio.

Estos problemas pueden resolverse utilizando una versión condensada de la lista de acceso.

Para condensar la longitud de la lista de control de acceso, muchos sistemas clasifican a los usuarios en tres grupos, en lo que respecta con cada archivo:

- **Propietario.** El usuario que creó el archivo será su propietario.
- **Grupo.** Un conjunto de usuarios que están compartiendo el archivo y necesitan un acceso similar al mismo es un grupo, o grupo de trabajo.
- **Universo.** Todos los demás usuarios del sistema constituyen el universo.

La técnica reciente más común consiste en combinar las listas de control de acceso con el esquema más general (y más fácil de implementar) de control de acceso que acabamos de describir, compuesto por los conceptos de propietario, grupo y universo.

Una dificultad es la relativa a la asignación de precedencias cuando los permisos y las listas **ACL** entran en conflicto. Por ejemplo, si "X" pertenece al grupo de un archivo, que tiene permiso de lectura, pero el archivo tiene una lista ACL que concede a "X" permiso de lectura y escritura, ¿debemos conceder o denegar una solicitud de escritura realizada por "X"? Solaris proporciona a las listas ACL prioridad sobre los permisos (ya que son de granularidad más fina y no se asignan de manera predeterminada). Esto sigue la regla general de que lo más específico debe tener prioridad sobre lo más genérico.

## *6.3 Integridad del sistema de archivos*

Uno de los mayores desastres que se pueden presentar en un computador es la destrucción del sistema de archivos. Ello puede representar la pérdida de meses de trabajo y de datos imprescindibles.

Los problemas en el sistema de archivos pueden venir por el uso de bloques del disco que están en mal estado. Este problema se puede solventar conociendo cuales son los bloques defectuosos. Los fabricantes de discos duros dan una lista de los bloques defectuosos de cada unidad. Conociéndolos se puede dedicar un sector del disco para contener la lista de los bloques defectuosos. El controlador del disco lee esta lista y al inicializarse elige una pista de reserva para sustituir a los que están inservibles, de forma que cuando se solicita un bloque de los defectuosos el sistema utilizará uno de los de repuesto. Otra solución a este problema consiste en construir un archivo con todos los bloques defectuosos, así quedan eliminados de la lista de bloques libres y mientras no se lea o escriba este archivo de bloques defectuosos no habrá problemas.

Las inconsistencias en el sistema de archivos pueden ocurrir por otras muchas causas, por ejemplo, si el sistema falla en la mitad de una operación de lectura, modificación y escritura de un bloque (al cortarse el suministro de energía eléctrica). Si además el bloque que se ha modificado, y que no ha dado tiempo a escribirlo, es un bloque con i-nodos, directorios o la lista de bloques libres el problema puede ser crítico. Problemas usuales de inconsistencia que se pueden presentar son:

a) Que un bloque aparezca en la lista de bloques usados y bloques libres. En este caso la solución es eliminarlo de la lista de bloques libres.
b) Que un bloque no aparezca en ninguna lista. La solución es añadirlo ala lista de bloques libres.
c) Que un bloque esté repetido en la lista de bloques libres. La solución también es sencilla, vale con reconstruir la lista de bloques libres, con una entrada para cada bloque libre.
d) La peor situación es que un bloque esté asignado a dos o más archivos. En este caso, si se elimina uno de los archivos se presenta la situación a), o si se eliminan los dos archivos la situación c). La solución es que se asigne un bloque libre a un archivo y se copie el contenido del bloque que estaba asignado a los dos; de esta forma no se tendrán bloques repetidos en las listas de cada archivo, aunque seguramente la información de los archivos no será consistente.

Para eliminar este tipo de inconsistencia suele haber utilidades del sistema operativo que las detectan y, si no son extremadamente graves, las pueden corregir. Este es el caso de Unix en el que hay una utilidad (fsck) que se puede ejecutar cada vez que el administrador del sistema lo crea conveniente, pero, además, si cuando se arranca el sistema se comprueba que no fue apagado de forma correcta, se ejecuta automáticamente para detectar las inconsistencias que pudieran ocurrir por un mal apagado y las corrige. Cuando el deterioro del sistema de archivos es irreparable es necesario disponer de copias de seguridad a partir de las cuales poder restaurarlo.

La forma más fácil de realizar copias de seguridad es haciendo volcados periódicos de todo el sistema de archivos. Antes estos volcados se hacían en cintas magnéticas, pero también se pueden hacer en discos de diferentes capacidades. Por ejemplo, hay sistemas que al disponer de dos discos iguales, tienen repetidos los datos, aunque esto supone aprovechar la mitad de la capacidad de almacenamiento del sistema. También se pueden usar unidades de discos extraibles.

Otra forma de hacer copias de seguridad es mediante los volcados incrementales. Con esta técnica sólo se transfieren aquellos archivos que han sido modificados desde la última vez que se hizo un volcado de seguridad. Para ello el sistema mantiene información de cuando se hacen los volcados de seguridad.

La principal desventaja de los volcados incrementares es la cantidad de datos que se generan y la complejidad del procedimiento de restauración. Por ejemplo, si se hacen volcados totales cada mes e incrementales de forma diaria, se necesitarán 31 cintas para los volcados incrementales, mas una para los totales. En caso de fallo del sistema, la restauración supone restaurar primero la cinta con el último volcado total y luego ir actualizando con las cintas de volcados incrementales.

## 6.4 Ataques a la integridad y seguridad del sistema de archivos

Los fallos y deterioros del sistema de archivos, causados por actos fortuitos o errores de la máquina o humano (se puede haber borrado un archivo sin querer), se solucionan con las adecuadas copias de seguridad, tal como se ha visto.

Pero un problema más grave es el de los intrusos que intentan acceder, de forma no autorizada al sistema de archivos.

El objetivo de la seguridad es prevenir y eliminar estas amenazas. En particular, un sistema seguro debe mantener la integridad (los datos deben ser correctos), la disponibilidad y la privacidad de la información Esto supone la protección frente a modificaciones no autorizadas y a la modificación no detectada de datos, así como la resistencia a la penetración.

La penetración en un sistema informático se puede hacer de diferentes formas y por diversos medios. Entre los más conocidos están:

a) La utilización por parte del intruso de la cuenta de un usuario legítimo. Para conseguirlo puede usar un terminal con una sesión abierta, situación que se da cuando el usuario legítimo deja una sesión abierta en un terminal, con lo que el agresor puede acceder a toda la información disponible en esa cuenta. O también, obteniendo la contraseña de un usuario, para ello puede utilizar distintas técnicas, que van desde la adivinación y los ensayos de prueba y error, al robo o intimidación, pasando por el engaño.

b) La ejecución de programas denominados ***"caballos de Troya"***, los cuales ocultan parte de su funcionalidad, frecuentemente destinada a obtener datos o derechos de acceso del usuario. Esta es la situación que se puede dar cuando el intruso crea un programa falso de "login:", idéntico en presentación al del sistema, de forma que el usuario escribe su "login" y

contraseña, la cual será utilizada posteriormente por el intruso para acceder a sus archivos y programas.

c) La propagación de gusanos y virus informáticos. La diferencia entre un gusano y un virus está en que el virus es parte del código de un programa, mientras que el gusano es un programa en a mismo. El gusano causará graves problemas al sistema debido a que carga en exceso al computador, usando para su propagación recursos desproporcionados de procesamiento comunicación, con lo que el sistema puede denegar servicios a usuarios legítimos. Por el contrario el virus es un trozo de código de un programa, que infectará a otros programas copiándose. Por lo general también realiza actividades dañinas, como eliminar archivos o corromper los bloques de arranque del disco.

d) La inspección del sistema de archivos. Sistemas como Unix permiten, por defecto, a todos los usuarios leer los archivos. Un usuario mal intencionado podría acceder al archivo de contraseñas, copiarlo y utilizar técnicas de análisis criptográfico y métodos de prueba y error para descifrar las contraseñas, además en estos archivos suele haber información relativa a los usuarios, como su nombre, dirección o teléfono de contacto, su ID (número de identificación en el sistema), programa de arranque (generalmente el interprete de comandos que utiliza), directorios, y otras informaciones que no están codificadas.

La forma típica de funcionar de un virus es la siguiente. Normalmente el virus vendrá en el código de un programa, es frecuente que para los computadores personales este programa sea algún juego. El programa se suele distribuir de forma gratuita o a bajo coste. Cuando se arranca el programa, este examina todos los programas del disco, si encuentra uno sin infectar, lo infecta añadiendo el código del virus al final del archivo del programa y sustituyendo la primera instrucción por un salto a la primera instrucción del código del virus. Además de infectar otros programas, el virus suele realizar alguna actividad dañina o, al menos, molesta. Son típicos los virus que corrompen el sistema de archivos o los eliminan, o que provocan algún tipo de mal función en los programas infectados.

## 6.5 Principios de diseño de sistemas seguros

Salltzer y Schroeder (1975) identificaron varios principios generales que se pueden utilizar como guía para el diseño de sistemas seguros. Un resumen de sus ideas es la siguiente:

1) El diseño del sistema debe ser público. Los diseñadores se engañan si confían la seguridad del sistema en la ignorancia de los atacantes. Los algoritmos pueden ser conocidos pero las claves deben ser secretas.
2) El estado predefinido es el de no acceso. Los derechos de acceso deben ser adquiridos sólo con permiso explícito.
3) Verificar la autorización actual. Cada petición de acceso a un objeto debe conllevar la comprobación de la autorización. Esta verificación de autorización no debe ser abandonada por el sistema, ya que se pueden producir situaciones como la de que un usuario abra un archivo y lo mantenga abierto durante días, aunque cambien las autorizaciones de uso de este archivo.
4) Mínimos privilegios. Cada proceso debe utilizar el mínimo grupo de privilegios para completar su tarea. Esto delimita los posibles daños causados por caballos de Troya. Por ejemplo, si un editor solo puede tener acceso al archivo que edita, un editor con un caballo de Troya solo puede ocasionar un daño limitado.
5) Mecanismos simples e integrados. Mantener el diseño tan sencillo como sea posible facilita la verificación y corrección de las implementaciones. Además, para que el sistema sea verdaderamente seguro, el mecanismo debe estar integrado hasta las capas más bajas del sistema.
6) Psicológicamente aceptable. El mecanismo debe ser fácil de usar de forma que sea aplicado correctamente y no sea rechazado por los usuarios.

A continuación y en la sección siguiente se estudiarán algunos de los principales mecanismos de seguridad de los sistemas informáticos, como la identificación de usuarios o el control de acceso, siguiendo los principios expuestos anteriormente.

# 6.6 Identificación de usuarios

Muchos esquemas de seguridad se basan en la suposición de que el sistema conoce al usuario. El problema en este caso es la identificación del mismo. Este problema se suele llamar de validación y se basa en tres puntos (o en una combinación de ellos):

1) Posesión de un secreto, algo conocido por el usuario. Una contraseña que le da acceso al sistema.
2) Posesión de un artefacto, algo que al poseerlo el usuario le permite acceder al sistema; por ejemplo tarjetas magnéticas o llaves físicas.
3) Uso de alguna característica fisiológica o de comportamiento del usuario.

## Contraseñas

La contraseña es uno de los mecanismos de validación más comunes en los computadores. En este caso el computador le pide al usuario una contraseña que se comprueba en una tabla (generalmente almacenada en un archivo). Si la contraseña es correcta el computador permite el acceso. Por lo general las contraseñas están cifradas.

El uso de contraseñas tiene muchos puntos débiles. Generalmente, los usuarios eligen contraseñas fáciles de recordar, por ejemplo el nombre del hijo o de algún pariente o amigo, o de su ciudad natal.

Los computadores modernos afrontan este problema de diferentes formas. Una de ellas es cifrar la contraseña junto con un número aleatorio de n bits. El número aleatorio se modifica al cambiar la contraseña. Este número aleatorio se guarda en el archivo de contraseñas en forma no cifrada. En el archivo de contraseñas no se almacena la contraseña cifrada, sino que primero se concatena la contraseña al número aleatorio y después se encripta, guardando el resultado en el archivo de contraseñas.

Otros computadores piden a los usuarios que cambien periódicamente las contraseñas o limitan el número de intentos de acceso, de forma que si después de una serie de intentos no se introduce la contraseña correcta, la cuenta del usuario queda bloqueada o se corta la línea.

## Identificación mediante artefactos

los artefactos suelen ser bandas magnéticas o tarjetas electrónicas. En este caso, junto a los terminales suele haber un lector de tarjetas o de distintivos. Este debe disponer del distintivo o tarjeta para poder hacer uso del terminal. Es normal que este tipo de identificación esté asociado a una contraseña, como es el caso de los terminales bancarios: el usuario debe introducir la tarjeta y además suministrar una contraseña.

Este tipo de identificación funciona bien en sitios en donde el distintivo de identificación se usa para otros propósitos. Por ejemplo, en muchas empresas los empleados tienen tarjetas de identificación con las que acceden a las dependencias. El uso de estas tarjetas como artilugio de identificación para el computador reduce la probabilidad de pérdidas no detectadas del artefacto de identificación.

Otras variantes son las tarjetas inteligentes, que mantienen la contraseña del usuario secreta para el sistema, ya que está almacenada en la propia tarjeta. Esto hace mas difícil descubrir las contraseñas.

## Identificación física

Otro método consiste en utilizar características propias del usuario para la identificación. Las características se pueden catalogar en dos grupos:

1) Fisiológicas. En este grupo se utilizan características físicas difíciles de reproducir, como huellas dactilares o vocales, características faciales o geometría de la mano (por ejemplo dispositivos para medir la longitud de los dedos).
2) De comportamiento. En este grupo entran técnicas como el análisis de firmas o patrones de voz.

Sea cual sea la técnica de identificación que se utilice su objetivo es permitir el acceso a los usuarios legítimos y denegarlo a los intrusos. La efectividad de las técnicas de identificación se hace en función de la tasa de falsas aceptaciones y de falsos rechazos, es decir, el porcentaje de intrusos admitidos erróneamente y el porcentaje de usuarios legítimos a los que se deniega acceso debido a un fallo en el mecanismo de identificación.

## 6.7 Mecanismos de protección y control de acceso

Los mecanismos de protección surgieron con la multiprogramación, con la intención de que los programas de cada uno de los usuarios estuvieran en la partición de memoria asignada e impedir así que los programas traspasaran sus límites a otras particiones y las dañaran. Pero la necesidad de compartir objetos tanto en memoria principal como en la memoria secundaria motivó que los mecanismos de control de acceso hicieran se más complejos.

Es importante distinguir entre política de protección y mecanismo de protección. La diferencia está en que los mecanismos dicen como se hará algo, mientras que la política dice qué se hará. Esta separación permite una mayor flexibilidad, ya que puede cambiar la política a lo largo del tiempo, pero manteniendo un mecanismo general. Este cambio requerirá modificar sólo algunos parámetros o tablas del sistema.

### 6.7.1. Dominios de protección

Un sistema de cálculo es un sistema complejo que se puede idealizar como un conjunto de procesos y objetos Por objetos se entienden tanto las distintas unidades del computador (CPU, segmentos de memoria, discos,..) como las diferentes informaciones que almacenan (archivos, programas,...). Cada objeto es un tipo abstracto de dato con un único nombre que lo distingue de los demás objetos, y sólo es posible acceder a él por medio de operaciones útiles y bien definidas. Las operaciones pueden depender del objeto en cuestión, así *read y write* son operaciones adecuadas para un archivo, mientras que *up* y *down* lo son para un semáforo.

Lógicamente, un proceso sólo debe poder acceder a aquellos recursos para los cuales está autorizado y que necesita en ese momento para completar su tarea. Este requisito se denomina *principio de la necesidad de saber,* y es útil para limitar los posibles daños que puede causar en el sistema un proceso con errores.

Para analizar los mecanismos de protección es conveniente incorporar el concepto de *dominio de protección.* Así cada proceso trabaja dentro de un dominio, el cual especifica los recursos a los cuales puede tener acceso. Cada dominio define un conjunto de objetos y las operaciones que se les pueden aplicar. La capacidad para ejecutar una operación sobre un objeto es un derecho de acceso. Un dominio es un conjunto de derechos de acceso, cada uno de los cuales esta formado por un par de la forma:

***<nombre del objeto, conjunto de sus derechos>***

En la Ilustración 5 se muestran los derechos de acceso para cada objeto en los tres dominios ($D_1$, $D_2$, $D_3$). Para poder leer y escribir el objeto programa1 es necesario que un proceso se esté ejecutando en el dominio $D_1$, pero para poderlo ejecutar es necesario que esté en el dominio $D_3$. Por otra parte los dominios $D_2$ y $D_3$ comparten el objeto impresora con derecho de sólo escritura. En cada momento se ejecuta un proceso en algún dominio de protección

Estas ideas se pueden aclarar si se estudia el caso de Unix. En Unix el dominio de un proceso está definido por el identificador de usuario *(uid) y* del grupo *(gid).* Para un *uid* y un *gid* dados se puede elaborar una lista completa de todos los objetos a los que puede tener acceso y el tipo de acceso, es decir, una lista de archivos, dispositivos, etc., con sus correspondientes permisos de acceso.

## 6.7.2. Matriz de acceso

Las relaciones entre dominios y objetos se pueden representar de forma abstracta mediante una matriz denominada *matriz de acceso.* Las filas de la matriz de acceso representan dominios y las columnas objetos. La Tabla 4 es un ejemplo de una matriz de acceso para el mismo caso de la Ilustración 5

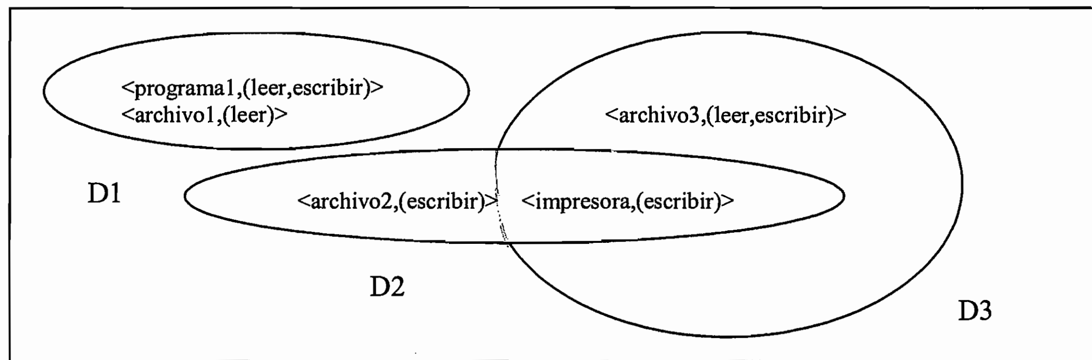



**Ilustración 5 Sistema de tres dominios**

Cada elemento de la matriz consiste en un conjunto de derechos de acceso.

En la Tabla 4 hay tres dominios y cinco objetos (un programa, tres archivos y una impresora). Cuando se ejecuta un proceso en el dominio $D_1$, puede leer y escribir programa1 y sólo leer archivo1. En el dominio $D_3$ se puede ejecutar programa1 pero no se puede leer ni escribir en él, sin embargo se puede leer y escribir en archivo3. En el dominio $D_2$, solo se puede escribir en el archivo2 y en la impresora.

Con esta matriz y el número del dominio, el sistema puede determinar que tipo de acceso se permite a un objeto específico. El problema es como realizar de una forma eficaz esta matriz.

| **Dominios** | **programa1** | **archivo1** | **archivo2** | **archivo3** | **impresora** |
|---|---|---|---|---|---|
| $D_1$ | leer/escribir | leer | | | |
| $D_2$ | | | escribir | | escribir |
| $D_3$ | ejecutar | | | leer/escribir | escribir |

**Tabla 4 Matriz de acceso**

Generalmente, la matriz de acceso es una matriz dispersa, con muchos elementos vacíos. Por ello una forma sencilla de realizarla es mediante una tabla global, consistente en tripletas ordenadas (dominio, objeto, derechos). Cuando un proceso ejecuta una operación P para un objeto O dentro de un dominio D, se busca en la tabla global la tripleta (D, O, derechos acceso), donde $P \in$ derechos acceso. Si se encuentra esta tripleta se permite la operación y el proceso puede continuar, sino se produce una condición de error. Este diseño presenta varios inconvenientes. La tabla suele ser grande y no se puede conservar en memoria principal, por lo que requiere operaciones adicionales de entrada/salida. Además, no se pueden aprovechar las agrupaciones de objetos o dominio: por ejemplo, si todos pueden leer un objeto éste deberá tener una entrada en cada dominio.

Dos métodos prácticos que se suelen utilizar están basados en almacenar la matriz por columnas *(lista de accesos)* o por filas *(lista de capacidades).*

## 6.7.3. Lista de accesos

En la primera técnica, a cada objeto se le asocia una lista ordenada con todos los dominios que pueden tener acceso al objeto y la forma de dicho acceso. Esta lista se denomina *lista de accesos* para el objeto especificado.

Un ejemplo de la lista de acceso para los dominios y objetos de la Ilustración 5 es la siguiente:

program1 ($D_1$, leer-escribir). ($D_3$, ejecutar)

archivo1: ($D_1$, leer)

archivo2: ($D_2$, escribir)

archivo3: ($D_3$, leer-escribir)

impresora: ($D_2$, escribir), ($D_3$, escribir)

El principal inconveniente de las listas de accesos es el retardo que se provoca con la búsqueda para verificar la autoridad de un sujeto para acceder al objeto solicitado.

Para evitar búsquedas de listas potencialmente muy largas de usuarios autorizados y ahorrar espacio de almacenamiento, algunos sistemas dividen a los usuarios en grupos y sólo almacenan los derechos de acceso de los grupos. Este esquema ahorra almacenamiento y agiliza el procesamiento reduciendo la flexibilidad y limitando el número de dominios. Unix usa este tipo de esquema, las listas están reducidas a tres entradas por archivo, una para el propietario, otra para el grupo y otra para todos los usuarios.

## 6.7.4. Lista de capacidades

La otra técnica comentada es almacenar la matriz de acceso por filas. En este caso a cada dominio (o sujeto) se le asocia una lista de objetos a los cuales puede tener acceso, junto con una indicación de las operaciones permitidas sobre cada objeto. Esta lista se denomina lista de capacidades. Un ejemplo de una lista de capacidades para el dominio $D_1$, de la Tabla 4 se encuentra en la Tabla 5 Listado de capacidades del dominio $D_1$.

Originalmente las capacidades se propusieron como un tipo de apuntador seguro para satisfacer la necesidad de protección de recursos en los sistemas multiprogramados. La lista de capacidades está asociada a un dominio, pero un proceso que se ejecuta en ese dominio no puede acceder a ella directamente.

| | Tipo | Derechos | Objeto |
|---|---|---|---|
| 0 | programa | Leer/escribir | Puntero a programa1 |
| 1 | archivo | leer | Puntero a archivo1 |

**Tabla 5 Listado de capacidades del dominio $D_1$**

# 7 PLANIFICACIÓN DE LOS ACCESOS A DISCO

Los Sistemas actuales suelen utilizar discos como dispositivos de almacenamiento secundario. La velocidad a la que se realicen los accesos al disco condicionará el rendimiento general del sistema (paginación, swap, carga de programas, etc.) y el de las aplicaciones de usuario (acceso a los datos en ficheros, etc.).

Ante la petición de acceso a un bloque determinado, el hardware del disco debe realizar, fundamentalmente, tres operaciones que quedan gráficamente representadas en la figura 7.2:

1. Mover el brazo buscando el cilindro correspondiente **(búsqueda)**

2. Esperar a que el bloque se sitúe frente a la cabeza de lectura y escritura **(latencia)**

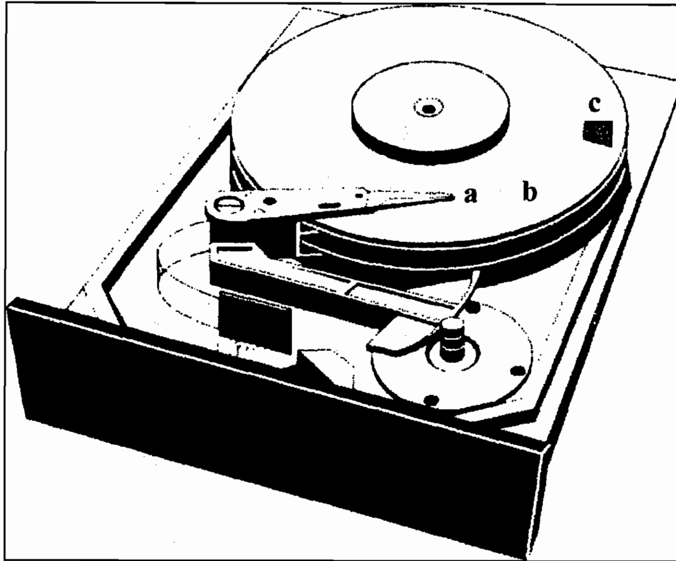
3. Transmitir el bloque deseado **(transmisión)**

El tiempo de transmisión será igual en todos los accesos puesto que todos los bloques son del mismo tamaño, pero el valor de los otros dos tiempos dependerá de la posición desde la que se inicie la búsqueda.

a = tiempo de búsqueda

b = tiempo de latencia

c = tiempo de transferencia




En los Sistemas multiprogramados puede haber varios procesos en ejecución en un momento dado, y pueden hacer peticiones simultáneas de acceso a un disco. Si mientras se está accediendo al disco llegan más peticiones, el sistema deberá encolarlas en una lista de espera (una por dispositivo). Cuando el disco quede libre, el sistema podrá decidir el orden en el que atenderá las peticiones pendientes, buscando minimizar el desplazamiento del brazo del disco y por tanto el tiempo de servicio de la petición. Vamos a analizar alguno de los algoritmos que pueden utilizarse para planificar dichas peticiones.

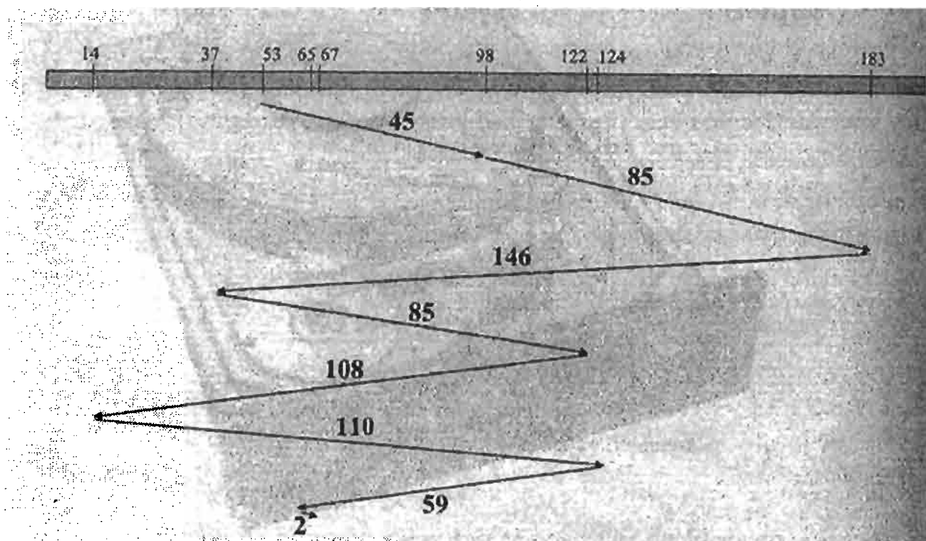
## 7.1 Primero en llegar, primero en acceder (FCFS)

El criterio más simple es aquél que sirve las peticiones según su orden de llegada. Su programación es sencilla y no produce una sobrecarga significativa, pero su eficacia es relativa.

Consideremos la siguiente lista de peticiones en espera de un disco:

98, 183, 37, 122, 14, 124, 65, 67

peticiones indicadas por el número de pista al que se desea acceder. Suponiendo que el brazo del disco parte de la pista 53 cuando comienza a atender esas peticiones, el gráfico de la figura 7.3 muestra los desplazamientos que realizará dicho brazo cuando se utilice el algoritmo FCFS.

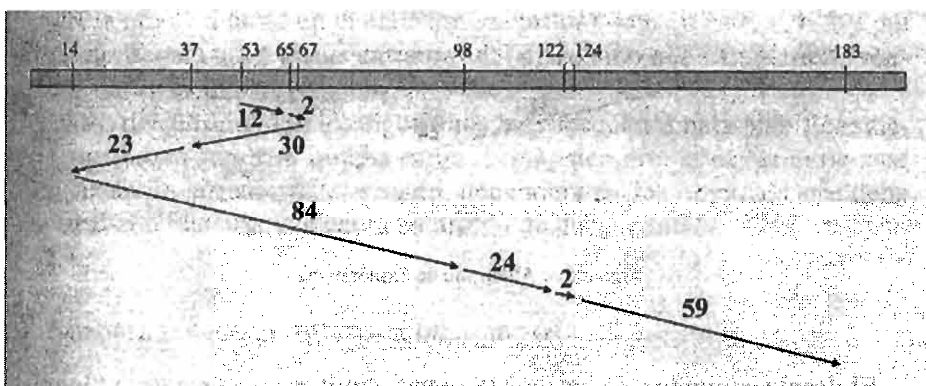



Para atender todas las peticiones, el brazo se ha desplazado en total 640 pistas. Sin embargo, se podría reducir notablemente ese recorrido, si las peticiones 37 y 14 se atendieran juntas, antes o después de servir las peticiones 122 y 124.

Este algoritmo puede ser útil para planificar discos con poca carga (pocos accesos).

## 7.2 Primero el de menor tiempo de búsqueda (SSTF)

Atenderá primero la petición más cercana a la última servida, o lo que es lo mismo, aquella que requiera un desplazamiento menor del brazo. Si aplicamos este criterio a la lista de peticiones del apartado anterior, obtendremos el gráfico de la figura 7.4




El desplazamiento total es ahora de 236 pistas, notablemente menor que el obtenido con el algoritmo anterior; pero su forma de atender las peticiones puede postergar indefinidamente algunas de ellas. Supongamos que existen dos peticiones pendientes, pistas 25 y 110. Si mientras se atiende

la 25 llegan más peticiones cercanas a ella (o menos alejadas que la 110), este algoritmo atenderá las nuevas aplazando el servicio de la 110.

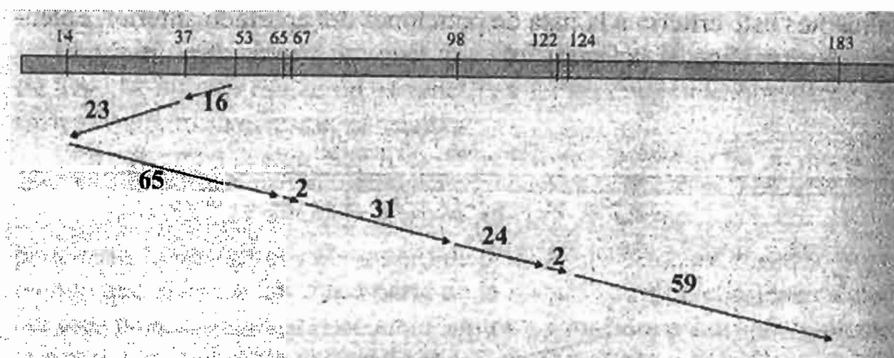
La velocidad de servicio de una petición variará según que las peticiones posteriores provoquen su postergación o no. La imposibilidad de poder predecir los tiempos de acceso al disco, hace que el algoritmo SSTF sea poco adecuado para los sistemas Interactivos.

Esas variaciones tienen menor importancia cuando se trata de procesar trabajos por lotes y, por ello, este algoritmo es más útil para Sistemas Batch.

# 7.3 Exploración (Scan)

Ahora se fija un recorrido para el brazo del disco que va desde la primera a la última pista y vuelta hacia la primera, tratando todas las peticiones que se encuentre en su camino. En su camino irá atendiendo las peticiones que existan según el sentido del desplazamiento, independientemente del momento en que se han producido.

Aplicando este algoritmo a la lista de peticiones de los apartados anteriores, y suponiendo que el sentido de desplazamiento de la cabeza es hacia la primera pista (pista 0). Obtenemos el gráfico de la figura 7.5.

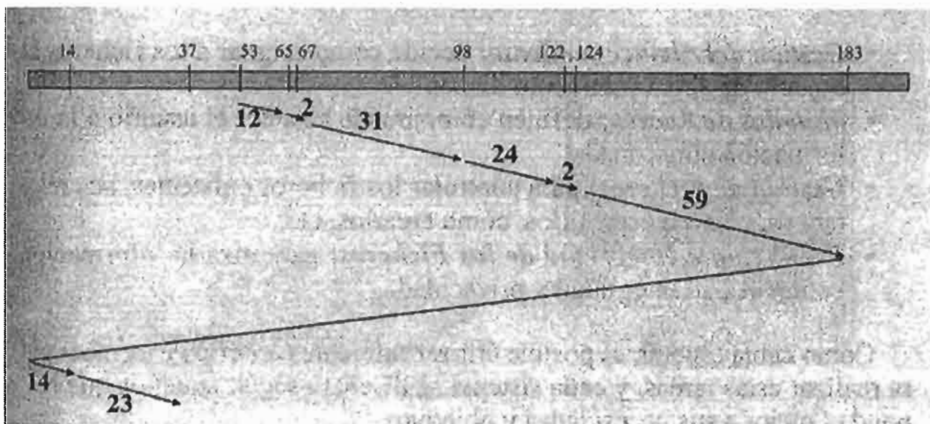



El desplazamiento total es de 238 pistas, similar, en este caso, al logrado con el SSTF pero sin sus inconvenientes. Este ha sido el algoritmo base a partir del cual se han desarrollado los algoritmos de planificación de discos más utilizados actualmente.

# 7.4 Exploración circular (C-Scan)

La ***Exploración Circular*** es una variación del algoritmo anterior para conseguir unos tiempos de espera más equilibrados, con independencia de que las peticiones correspondan a pistas de los extremos o del interior del disco.

Con ese fin, el movimiento del brazo sigue siendo el mismo, pero sólo atiende peticiones en uno de los sentidos de marcha. Al llegar a la última pista regresará al principio rápidamente sin atender petición alguna (figura 7.6).




En la práctica tanto en el algoritmo anterior como en éste, el brazo del disco no llegará a las pistas extremas del disco sino que cambiará de sentido al servir la última petición.

Estos dos últimos son los algoritmos más adecuados para planificar discos que deban soportar mucha carga. Como siempre, la decisión de cuál utilizar en un sistema determinado, dependerá de los objetivos que deba cubrir dicho Sistema y la carga en disco que deba manejar.

# INDICE COMPLETO DE NIVELES

1 CONCEPTO DE ARCHIVO .......... 1
1.1 Atributos de archivo .......... 1
1.2 Operaciones con los archivos .......... 2
1.3 Tipos de archivos .......... 3
1.4 Estructura de los archivos .......... 4
1.4.1. Estructura interna de los archivos .......... 4
2 METODOS DE ACCESO Y ASIGNACIÓN .......... 5
2.1 Acceso secuencial .......... 5
2.2 Acceso directo .......... 5
2.3 Otros métodos de acceso .......... 6
2.4 METODOS DE ASIGNACIÓN .......... 6
2.4.1. Asignación contigua .......... 6
2.4.2. Asignación enlazada .......... 8
2.4.3. Asignación indexada .......... 9
2.4.4. Prestaciones .......... 11
3 GESTIÓN DEL ESPACIO LIBRE .......... 12

3.1 Vector de bits ........ 12
3.2 Lista enlazada ........ 13
3.3 Agrupamiento ........ 13
4 DIRECTORIOS DE ARCHIVOS ........ 13
4.1 Directorios con forma de grafos acíclicos ........ 16
4.2 Directorio con forma de grafo general ........ 17
4.3 Implementación de directorios ........ 18
4.3.1. Lista lineal ........ 18
4.3.2. Tabla hash ........ 19
5 ARQUITECTURA DE UN SISTEMA DE ARCHIVOS ........ 20
5.1 Estructura de un sistema de archivos ........ 22
6 SEGURIDAD Y PROTECCIÓN ........ 25
6.1 Tipos de acceso ........ 25
6.2 Control de acceso ........ 25
6.3 Integridad del sistema de archivos ........ 26
6.4 Ataques a la integridad y seguridad del sistema de archivos ........ 27
6.5 Principios de diseño de sistemas seguros ........ 28
6.6 Identificación de usuarios ........ 29
6.7 Mecanismos de protección y control de acceso ........ 30
6.7.1. Dominios de protección ........ 30
6.7.2. Matriz de acceso ........ 31
6.7.3. Lista de accesos ........ 32
6.7.4. Lista de capacidades ........ 32
7 PLANIFICACIÓN DE LOS ACCESOS A DISCO ........ 33
7.1 Primero en llegar, primero en acceder (FCFS) ........ 34
7.2 Primero el de menor tiempo de búsqueda (SSTF) ........ 35
7.3 Exploración (Scan) ........ 36
7.4 Exploración circular (C-Scan) ........ 36

# EVITACION DEL INTERBLOQUEO

## Algoritmo del banquero

n procesos
m recursos

DISPONIBLE[m]
MAXIMO[n][m]
ASIGNADO[n][m]          /*Datos del sistema*/
NECESIDAD[n][m]

SOLICITUD[n][m]

1. Si SOLICITUD$_i$ <= NECESIDAD$_i$, paso 2
   si no error

2. Si SOLICITUD$_i$ <= DISPONIBLE, paso 3
   si no
      recursos no disponibles,
      P$_i$ espera

3. DISPONIBLE := DISPONIBLE - SOLICITUD$_i$
   ASIGNADO$_i$ := ASIGNADO$_i$ + SOLICITUD$_i$
   NECESIDAD$_i$ := NECESIDAD$_i$ - SOLICITUD$_i$

## Algoritmo de seguridad

DISPONIBLE[m]
ASIGNADO[n][m]          /* datos del sistema */

Sean TRABAJO[m], ACABAR[n];   /* datos del algoritmo */

1) TRABAJO := DISPONIBLE
   ACABAR[i].=FALSO para i = 1,2 … n.

2) Hallar un i tal que:
   a) ACABAR[i]:= FALSO, y
   b) NECESIDAD$_i$ <= TRABAJO
   Si no existe tal i, ir al paso 4.

3) TRABAJO := TRABAJO + ASIGNADO$_i$;
   ACABAR[i] := CIERTO;
   Ir al paso 2.

4) Si ACABAR[i] := CIERTO para todo i,
   Entonces, el sistema está en estado seguro

# DETECCIÓN

| Algoritmo de detección del interbloqueo |
|---|

DISPONIBLE[m]
ASIGNADO[n][m]  /* datos del sistema */
SOLICITUD[n][m]

Sean TRABAJO[m], ACABAR[n];  /* datos del algoritmo */

1) TRABAJO := DISPONIBLE
   Para i=1,2…n, Si ASIGNADO$_i$<>0, entonces
   ACABAR[i].=FALSO;
   si no, ACABAR[i]:= CIERTO

2) Hallar un i tal que:
   i. ACABAR[i]:= FALSO, y
   ii. SOLICITUD$_i$ <= TRABAJO
   Si no existe tal i, ir al paso 4.

3) TRABAJO := TRABAJO + ASIGNADO$_i$;
   ACABAR[i] := CIERTO;
   Ir al paso 2.

4) Si ACABAR[i] := CIERTO para todo i,
   Entonces, el sistema está en estado seguro
   Sino, hay interbloqueo y lo forman todos los procesos para los cuales
   ACABAR[i] := FALSO.

- La unidad de trabajo (interrumpible) de la CPU es la instrucción en código máquina (o ensamblador), no la instrucción de lenguaje en alto nivel.

**P1**

C

U1=U1+18;

U1--;

ENS

#P1.1/ LD A,#2000

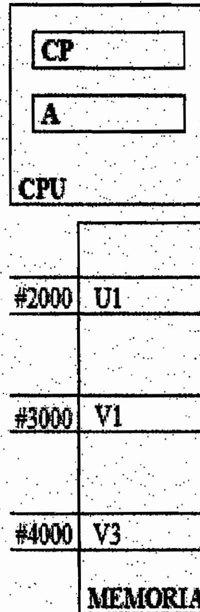
#P1.2/ ADD A,18

#P1.3/ SV A,#2000

#P1.4/ LD A,#2000

#P1.5/ DEC A

#P1.6/ SV A,#2000




**P2**

C

V3=V1-25;

V1=V3+2;

ENS

#P2.1/ LD A,#3000

#P2.2/ SUS A,25

#P2.3/ SV A,#4000

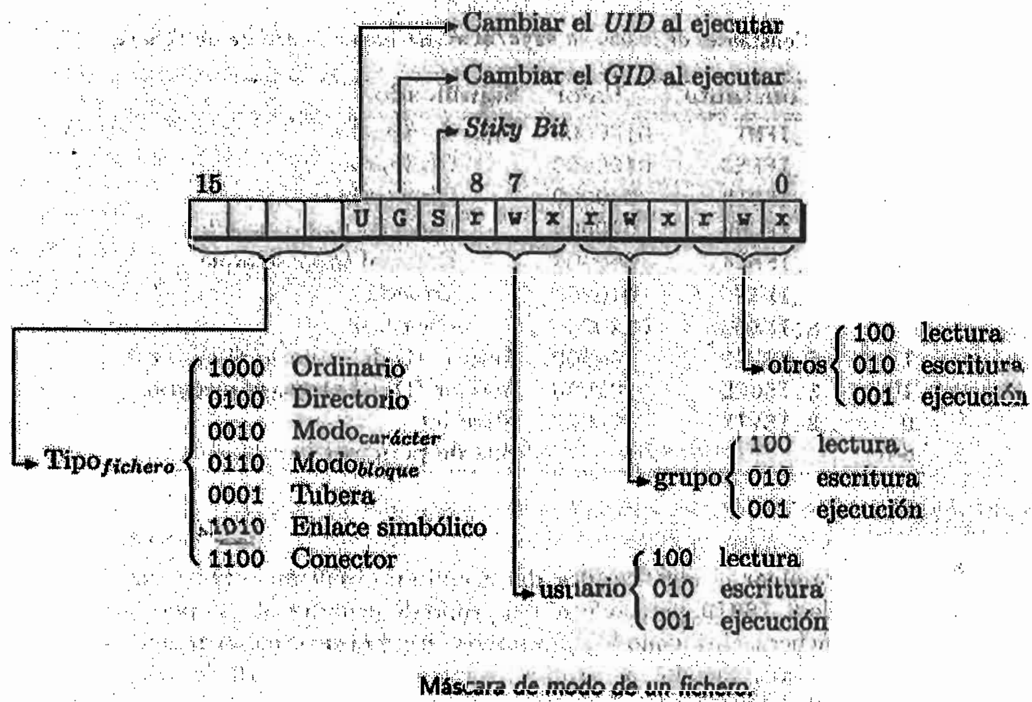
#P2.4/ LD A,#4000

#P2.5/ ADD A,2

#P2.6/ SV A,#3000



| ejecutando | | CP | A | U1 | V1 | V3 |
|---|---|---|---|---|---|---|
| | SO | P1.1 | ? | 3 | 5 | 2 |
| | P1.1 | P1.2 | 3 | 3 | 5 | 2 |
| Int | P1.2 | P1.3 | 21 | 3 | 5 | 2 |
| | SO | P2.1 | ? | 3 | 5 | 2 |
| | P2.1 | P2.2 | 5 | 3 | 5 | 2 |
| | P2.2 | P2.3 | -20 | 3 | 5 | 2 |
| Int | P2.3 | P2.4 | -20 | 3 | 5 | -20 |
| | SO | P1.3 | 21 | 3 | 5 | -20 |
| | P1.3 | P1.4 | 21 | 21 | 5 | -20 |
| | P1.4 | P1.5 | 21 | 21 | 5 | -20 |
| Int | P1.5 | P1.6 | 20 | 21 | 5 | -20 |
| | SO | P2.4 | -20 | 21 | 5 | -20 |
| | P2.4 | P2.5 | -20 | 21 | 5 | -20 |
| Int | P2.5 | P2.6 | -18 | 21 | 5 | -20 |
| | SO | P1.6 | 20 | 21 | 5 | -20 |
| | P1.6 | int | 20 | 20 | 5 | -20 |
| | SO | P2.6 | -18 | 20 | 5 | -20 |
| | P2.6 | Int | -18 | 20 | -18 | -20 |




Máscara de modo de un fichero.

Constantes definidas en <sys/stat.h> para el modo de un fichero.

| Bits | Constante | Valor | Significado |
|---|---|---|---|
| 15-12 | S_IFMT | 0170000 | Tipo de fichero: |
| | S_IFREG | 0100000 | Ordinario |
| | S_IFDIR | 040000 | Directorio |
| | S_IFCHR | 020000 | Especial modo carácter |
| | S_IFBLK | 060000 | Especial modo bloque |
| | S_IFIFO | 010000 | Tubería |
| | S_IFSOCK | 0140000 | Conector |
| 11 | S_ISUID | 04000 | Activar ID del usuario al ejecutar |
| 10 | S_ISGID | 02000 | Activar ID del grupo al ejecutar |
| 9 | S_ISVTX | 01000 | *Stiky* bit |
| 8-0 | | | Bits de permisos |