

APUNTES DE  
Introducción a los Sistemas Operativos  
2º DE GRADO EN INGENIERÍA INFORMÁTICA

**TEMA 4. SISTEMAS DE FICHEROS**

CURSO 2021/2022

© Reservados todos los derechos. Estos apuntes se proporcionan como material de apoyo de la asignatura Introducción a los Sistemas Operativos impartida en la Facultad de Informática de la Universidad de Murcia, y su uso está circunscrito exclusivamente a dicho fin. Por tanto, no se permite la reproducción total o parcial de los mismos, ni su incorporación a un sistema informático, ni su transmisión en cualquier forma o por cualquier medio (electrónico, mecánico, fotocopia, grabación u otros). La infracción de dichos derechos puede constituir un delito contra la propiedad intelectual de los autores.

## ÍNDICE GENERAL

<b>4. Sistemas de ficheros</b>	<b>1</b>
4.1. Introducción . . . . .	1
4.2. Ficheros . . . . .	2
4.2.1. Tipos de ficheros . . . . .	2
4.2.2. Acceso a un fichero . . . . .	4
4.2.3. Atributos de fichero . . . . .	4
4.2.4. Operaciones sobre ficheros . . . . .	5
4.3. Directorios . . . . .	6
4.3.1. Sistemas jerárquicos de directorios . . . . .	6
4.3.2. Nombre de la ruta de acceso . . . . .	7
4.3.3. Operaciones con directorios . . . . .	7
4.4. Implementación del sistema de ficheros . . . . .	8
4.4.1. Implementación de ficheros . . . . .	9
4.4.2. Implementación de directorios . . . . .	15
4.4.3. Resolución de rutas . . . . .	18
4.4.4. Ficheros compartidos . . . . .	20
4.4.5. Administración del espacio en disco . . . . .	21
4.5. Caché de disco . . . . .	23
4.6. Discos y sistemas de ficheros . . . . .	24
4.6.1. Particiones . . . . .	24
4.6.2. Estructura de un sistema de ficheros . . . . .	25



## CAPÍTULO 4

### SISTEMAS DE FICHEROS

#### 4.1 INTRODUCCIÓN

Los sistemas de computación actuales necesitan algún tipo de **almacenamiento secundario** para guardar información que complementa al **almacenamiento primario** proporcionado por la memoria RAM. Al menos, hay tres razones importantes para el uso de este tipo de almacenamiento:

- **Almacenar gran cantidad de información**, más de la que cabría en memoria principal.
- **Preservar la información para que esta no desaparezca** aunque termine el proceso que la utiliza, o aunque el sistema se apague o reinicie.
- **Permitir que cierta información** como programas, documentos, bases de datos, etc. **sea fácilmente compartida** y accedita de forma concurrente por varios procesos.

Para satisfacer estas necesidades, se hace uso de discos y otros dispositivos de almacenamiento secundario. Ahora bien, dado que **todos estos dispositivos almacenan la información en un simple conjunto de bloques, sin ninguna estructura de más alto nivel**, el sistema operativo tendrá que crear abstracciones que faciliten a los usuarios y a los programadores el uso de este tipo de almacenamiento. La solución suele ser almacenar la información en unidades llamadas **ficheros**<sup>1</sup>, los cuales se organizan y agrupan mediante **directorios**. La estructura de datos a través de la cual los **ficheros y directorios** se almacenan en el almacenamiento secundario se llama **sistema de ficheros**<sup>2</sup>.

Ya que el almacenamiento y uso de información es una de las tareas más comunes en cualquier sistema operativo, su sistema de ficheros será su parte más visible para los usuarios. De ahí que sea importante diseñarlo correctamente. Desde el punto de vista del usuario, son aspectos importantes de un sistema de ficheros: la forma de nombrar a los ficheros, las operaciones permitidas sobre ellos, el tipo de protección que podemos aplicar, etc. Desde el punto de vista del sistema operativo, en cambio, hay otros aspectos

<sup>1</sup>En este tema se llama **fichero** a lo que en Unix se denomina **fichero regular**. En Unix, el término «fichero» representa a un fichero de cualquier tipo: fichero regular, directorio, enlace simbólico, fichero especial de bloques o caracteres, etc.

<sup>2</sup>También se llama muchas veces **sistema de ficheros** a la parte del sistema operativo que gestiona la estructura de datos en disco. Dada la estrecha relación que existe entre los dos conceptos, aquí usaremos la expresión **sistema de ficheros** para referirnos indistintamente a cualquiera de ellas.

## CAPÍTULO 4. SISTEMAS DE FICHEROS

a tratar, especialmente la forma en la que se implementan los ficheros, los directorios y el sistema de ficheros en su conjunto.

En este capítulo vamos a describir los ficheros y directorios primero desde el punto de vista de los usuarios, para después pasar a estudiar las posibles implementaciones de los mismos.

### 4.2 FICHEROS Diapo 3

Un **fichero** es la **unidad lógica de almacenamiento**, es decir, para poder guardar información en un dispositivo de almacenamiento secundario debemos hacer uso de un fichero, y para leer información de dicho dispositivo también tenemos que hacer uso de un fichero.

En los sistemas operativos modernos (Windows, Linux, Mac OS X, etc.), un **fichero** es una secuencia de bytes cuyo significado está definido por el **programa** o programas que acceden al mismo. Con esta estructura de fichero se consigue máxima flexibilidad, ya que los **programas** pueden colocar cualquier información dentro de un **fichero** y organizar dicha información como deseen según una determinada estructura de datos.

Secuencia de bytes arbitraria.

Cada fichero tiene un nombre que lo identifica. Dependiendo del diseño realizado en el sistema operativo y en el propio sistema de ficheros, en el nombre se distinguirá o no entre mayúsculas y minúsculas, se permitirán o no caracteres especiales, el nombre tendrá cierta estructura (por ejemplo, «nombre.extensión») o no, podrá tener más caracteres o menos de longitud, etc.

#### 4.2.1 Tipos de ficheros Semana siguiente.

Es habitual que en un sistema operativo existan diferentes tipos de ficheros. Algunos tipos comunes son:

- **Ficheros regulares:** contienen información del usuario. Son de dos tipos: de texto o binarios. Los ficheros de texto constan de líneas de texto (codificado en UTF-8, Latin1, ASCII, etc) terminadas en CR, LF o LF/CR, que se pueden ver e imprimir tal cual son y pueden modificarse con un editor de texto común. Los ficheros binarios (lo de binario indica, simplemente, que no son de texto) suelen aparecer como una lista incomprendible de símbolos al ojo humano, si bien tienen una estructura interna, como comentaremos a continuación.
- **Directorios:** son ficheros gestionados por el propio sistema operativo para poder organizar y registrar los ficheros existentes en el sistema de ficheros (ver sección 4.3).
- **Ficheros especiales de caracteres:** están relacionados con la E/S y se utilizan para referenciar y acceder a dispositivos serie de E/S como terminales, impresoras y redes. Son comunes en los sistemas tipo Unix.
- **Ficheros especiales de bloques:** son iguales que los anteriores, pero para referenciar y acceder a discos y otros dispositivos de almacenamiento secundario que permiten un acceso aleatorio (ver sección 4.2.2).

Como hemos dicho, cada fichero regular binario tiene una estructura o formato que indica cómo se guarda la información que contiene. Generalmente, esta estructura la determina el programa que accede al fichero. Un caso particular son los ficheros ejecutables, cuya estructura la especifica el propio sistema operativo, que es el programa que

## 4.2. FICHEROS

accede a ellos. Esto es así porque es el sistema operativo el que debe cargar el contenido de estos ficheros en memoria para su ejecución. La figura 4.1 muestra dos posibles ejemplos de ficheros binarios: un ejecutable y un archivo de biblioteca. El primero contiene una cabecera y varias secciones. La cabecera básicamente contiene un *número mágico* (un número concreto que identifica el formato del ejecutable), el tamaño de las secciones de código y datos almacenadas después en el fichero y, finalmente, el punto de entrada o dirección del código donde debe empezar la ejecución del programa. El archivo de biblioteca, por su parte, almacena varios módulos objeto que contienen el código de diversas funciones.

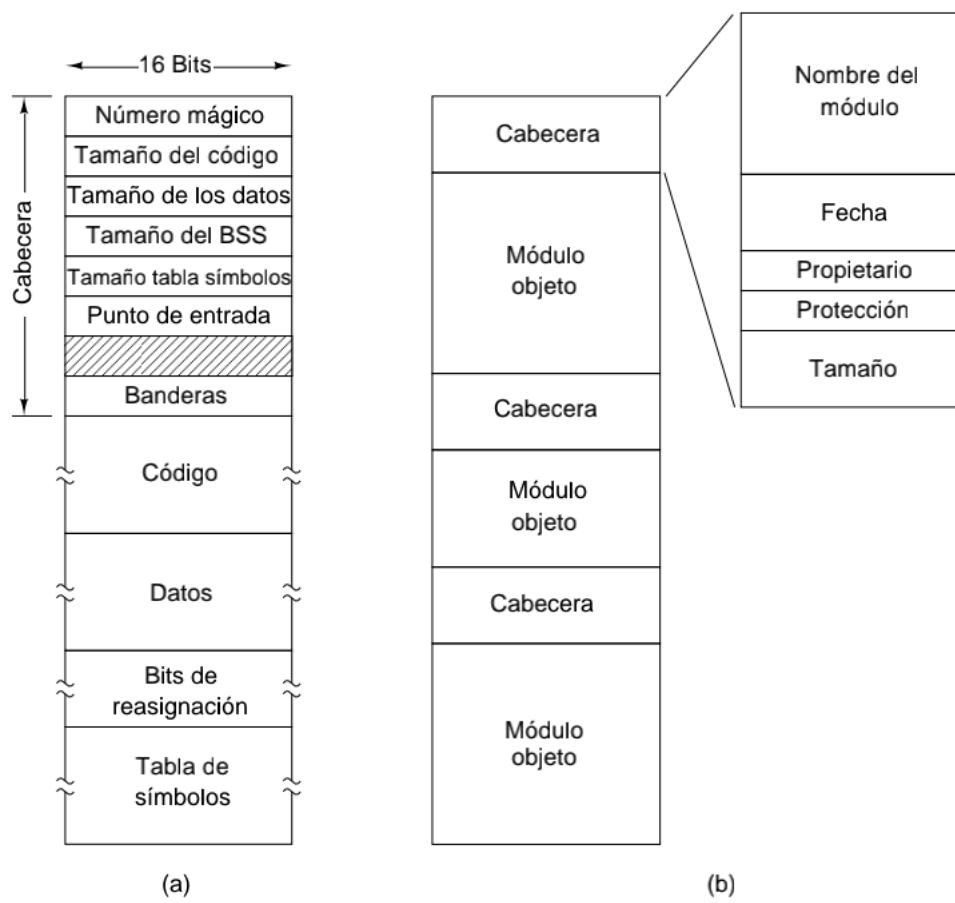


Figura 4.1. Dos ejemplos de un fichero binario. (a) Un fichero ejecutable con un cierto formato. (b) Un archivo de biblioteca.

Muchas veces el nombre de un fichero contiene una *extensión* que indica de qué tipo de fichero se trata: de texto o binario y, dentro de estos últimos, el tipo concreto. La tabla 4.1 muestra algunos ejemplos de extensiones. Obsérvese que para los sistemas operativos modernos la extensión no es significativa. Así, en Linux, podremos ejecutar un programa tanto si no tiene extensión, como si esta es .exe, .doc o cualquier otra cosa. Ahora bien, hay programas, como un explorador de ficheros, para los que la extensión puede ser importante de cara a determinar qué acción realizar con un fichero al hacer doble clic con el ratón sobre él. Las extensiones también ayudan a los usuarios a saber, con un simple vistazo, qué tipo de información contiene cada fichero.

## CAPÍTULO 4. SISTEMAS DE FICHEROS

Extensión	Significado
fichero.bak	Copia de seguridad
fichero.c	Código fuente en C
fichero.gif	Imagen en formato GIF
fichero.hlp	Fichero de ayuda
fichero.html	Código fuente en HTML
fichero.jpg	Imagen fija en formato JPEG
fichero.mp3	Sonido codificado en MP3
fichero.mpg	Vídeo codificado en MPEG
fichero.o	Fichero objeto (salida del compilador, todavía no enlazada)
fichero.pdf	Fichero PDF
fichero.ps	Fichero PostScript
fichero.tex	Fichero fuente de TeX
fichero.txt	Fichero de texto genérico
fichero.zip	Archivo comprimido con formato ZIP

Tabla 4.1. Algunas de las posibles extensiones en los nombres de fichero.

### 4.2.2 Acceso a un fichero

Hay dos tipos principales de acceso a un fichero: *acceso secuencial* y *acceso aleatorio*. En el acceso secuencial todos los bytes de un fichero deben leerse en orden, uno detrás de otro, sin posibilidad de saltarse algunos o leerlos en otro orden. En los ficheros de acceso aleatorio, es posible leer los bytes en un orden cualquiera, existiendo llamadas al sistema que permiten el posicionamiento (como `lseek`).

Si un fichero admite un acceso aleatorio entonces también admite un acceso secuencial. Sin embargo, hay ficheros donde el único tipo de acceso posible es el secuencial. Esto ocurre, por ejemplo, en los ficheros especiales de caracteres de Unix que representan a teclados o cintas. Aquí, por la propia naturaleza de los dispositivos, solo es posible leer byte a byte, uno detrás de otro. Otro ejemplo son las tuberías que permiten transferir información de un proceso a otro. En este caso, aunque no hay un dispositivo físico que condicione el tipo de acceso, este es impuesto por el propio sistema operativo.

el orden en que se introducen, se leen  
f[0] ← (→) f[1] → pipe

### 4.2.3 Atributos de fichero

Un fichero tiene un nombre y contiene ciertos datos. Además de eso, todos los sistemas operativos asocian información adicional a cada fichero, como fecha y hora de creación, tamaño actual, tamaño máximo, etc. A estos elementos adicionales se les llama *atributos*.

El número y tipo de atributos varía considerablemente de un sistema operativo a otro pues dependen del tipo de acceso permitido, del tipo de protección (si existe), etc. La tabla 4.2 muestra posibles atributos para un fichero que, como hemos dicho, aparecerán o no según el sistema operativo:

- Los 4 primeros atributos están relacionados con la protección del fichero e indican quién puede acceder al fichero y quién no.
- Las banderas son 1 o varios bits que permiten o no cierta propiedad, como las banderas de ocultación, de biblioteca (útil para hacer copias de respaldo o de seguridad), temporal, de lectura/escritura, etc.

## 4.2. FICHEROS

Campo	Significado
Protección	Quién debe tener acceso y de qué forma
Contraseña	Contraseña necesaria para tener acceso al fichero
Creador	Identificador de la persona que creó el fichero
Propietario	Propietario actual
Bandera de solo lectura	0 Lectura/escritura, 1 para lectura exclusivamente
Bandera de ocultación	0 normal, 1 para no exhibirse en listas
Bandera del sistema	0 fichero normal, 1 fichero de sistema
Bandera de biblioteca	0 ya se ha respaldado, 1 necesita <u>respaldo</u>
Bandera texto/binario	0 fichero de texto, 1 fichero binario
Bandera de acceso aleatorio	0 solo acceso secuencial, 1 acceso aleatorio
Bandera temporal	0 normal, 1 eliminar al salir del proceso
Banderas de cerradura	0 no bloqueado, ≠ 0 bloqueado
Tiempo de creación	Fecha y hora de creación del fichero
Tiempo del último acceso	Fecha y hora del último acceso al fichero
Tiempo de la última modificación	Fecha y hora de la última modificación del fichero
Tamaño actual	Número de bytes en el fichero
Tamaño máximo	Tamaño máximo al que puede crecer el fichero

Tabla 4.2. Algunos de los posibles atributos de un fichero.

- Los distintos atributos de **tiempo** son útiles por varias razones. Por ejemplo, si un **fichero fuente** ha sido modificado tras crear el **fichero objeto**, este deberá **compilarse de nuevo**.
- Finalmente, para todo fichero se suele guardar su **tamaño actual** y, en algunos casos, **el tamaño máximo** que puede llegar a tener.

### 4.2.4 Operaciones sobre ficheros

Cualquier sistema operativo debe proporcionar a los programadores las siguientes funciones para el manejo de ficheros:

- **Create** (crear): crea un fichero vacío. Al crear, se pueden indicar algunos atributos.
- **Delete** (eliminar): borra un fichero, liberando así el espacio que ocupa en disco.
- **Open** (abrir): abre un fichero para su uso.
- **Close** (cerrar): cierra un fichero abierto previamente. Será necesario abrir de nuevo el fichero para poder acceder a él.
- **Read** (leer): lee de un fichero la cantidad de bytes indicada como parámetro.
- **Write** (escribir): escribe en un fichero la información indicada como parámetro.
- **Append** (añadir): es una forma restringida de *write*. Solo se puede añadir datos al final del fichero.

## CAPÍTULO 4. SISTEMAS DE FICHEROS

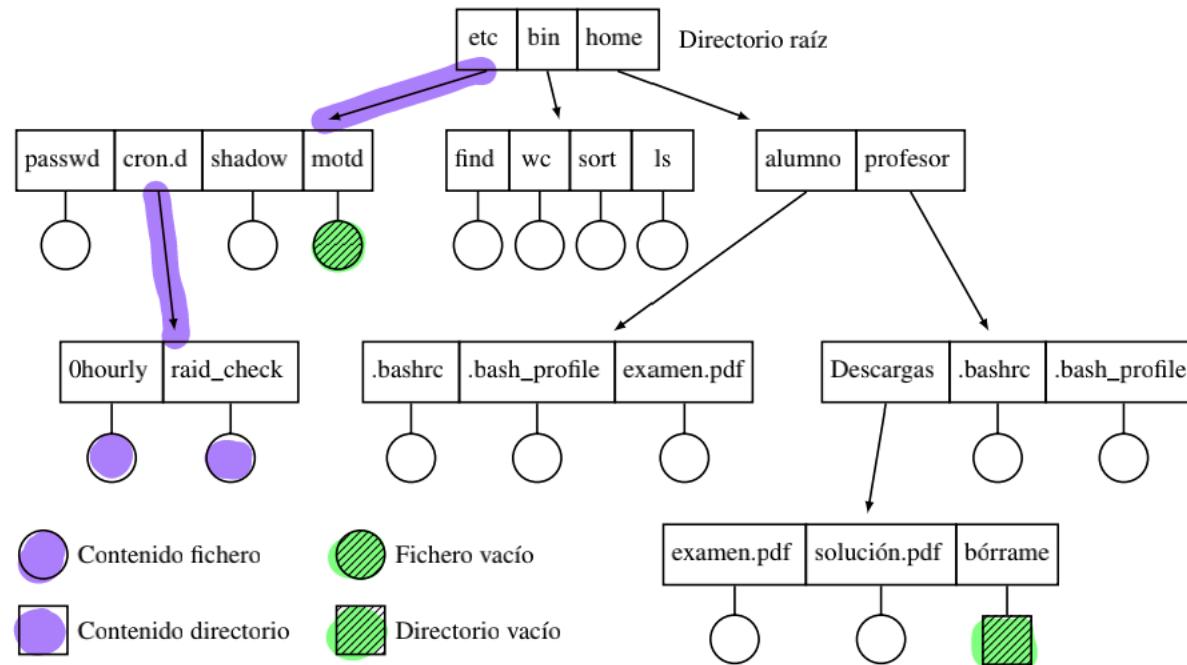


Figura 4.2. Directorios con estructura de árbol.

- **Seek** (buscar): para los ficheros de acceso aleatorio, nos permite indicar una posición dentro del fichero a partir de la cual leer o escribir.
- **Get attributes** (obtener atributos): obtiene los atributos asociados a un fichero.
- **Set attributes** (establecer atributos): nos permite cambiar un atributo (o varios) con el valor indicado como parámetro.
- **Rename** (cambiar de nombre): nos permite cambiar el nombre de un fichero y/o moverlo a otro directorio.
- **Truncate** (truncar): elimina el contenido de un fichero a partir de una posición dada.

### 4.3 DIRECTORIOS

Para organizar y llevar un registro de los ficheros, los sistemas operativos utilizan, por lo general, *directorios*. Como veremos después (ver apartado 4.4.2), los directorios son *ficheros* que contienen información sobre otros *ficheros*; principalmente, contienen el nombre de esos otros ficheros, aunque también pueden incluir información sobre sus atributos y otros datos dependiendo de la implementación realizada. Estos ficheros (los directorios) son gestionados por el propio sistema operativo y, por tanto, no pueden ser ni accedidos ni manipulados directamente por los programas de usuario.

Como son ficheros.  
Tienen atributos.

#### 4.3.1 Sistemas jerárquicos de directorios

Los sistemas operativos actuales utilizan un *árbol de directorios* para organizar los ficheros (ver figura 4.2). En esta estructura, cada directorio, además de ficheros, puede contener, a su vez, un número cualquiera de otros directorios (que serán *subdirectorios* de él).

## 4.3. DIRECTORIOS

Con un árbol de directorios, un usuario puede tener tantos directorios como desee y así agrupar sus ficheros según sus propias necesidades.

### 4.3.2 Nombre de la ruta de acceso

Cuando un sistema de ficheros está organizado como un árbol de directorios, se necesita una forma de determinar los nombres de los ficheros. Se suelen utilizar dos métodos.

En el primero, cada fichero tiene una *ruta de acceso absoluta* (o, simplemente, *ruta absoluta*), la cual consta de la ruta de acceso desde el directorio raíz hasta el fichero. Estos nombres absolutos son únicos y siempre comienzan por el carácter separador, que es «\» en sistemas Windows y «/» en sistemas Unix<sup>3</sup>.

En el segundo método, cada fichero tiene una *ruta de acceso relativa* (o, simplemente, *ruta relativa*), que se utiliza junto con el concepto de *directorio de trabajo* o *directorio actual* de un proceso. La ruta relativa nunca comienza por el carácter separador, para diferenciarla de una ruta absoluta, y se construye indicando los directorios que hay que recorrer para llegar desde el directorio actual hasta el fichero o directorio deseado.

Dentro de cada directorio suelen existir dos directorios que tienen un nombre especial: el directorio «..» y el directorio «...». El primero es otra forma de referirse al directorio actual, mientras que el segundo es una forma de representar al *directorio padre*, es decir, al directorio justo por encima del directorio actual en el árbol de directorios. Estas entradas especiales se suelen utilizar a la hora de construir rutas relativas. Así, si para llegar a un fichero desde el directorio actual tenemos que ascender en la jerarquía, cada ascenso a un directorio padre se indicará con «...». Por ejemplo, si existe la ruta absoluta /usr/bin/ls y el directorio actual es /usr/sbin, entonces, una posible ruta relativa correspondiente a la ruta absoluta sería .. /bin/ls.

Nótese que las expresiones «..» y «...» permiten construir innumerables rutas relativas para un mismo fichero o directorio. Por ejemplo, para la ruta absoluta anterior, además de la ruta relativa .. /bin/ls que hemos indicado, desde el mismo directorio /usr/sbin también existen las siguientes rutas relativas: ... / /usr/bin/ls, ... /bin/./ls, ... / /etc/... /usr/bin/ls, etc.

### 4.3.3 Operaciones con directorios

Como hemos dicho, aunque los directorios son realmente ficheros, no pueden ser manejados como tales por los programas de usuario. En su lugar, el sistema operativo proporciona diversas llamadas al sistema para acceder y cambiar estos ficheros de forma indirecta.

Las llamadas al sistema para el manejo de directorios tienen más variación de sistema a sistema que las relacionadas con ficheros. Veamos algunas operaciones:

- **Create** (crear): crea un directorio «vacío» (en realidad, un directorio recién creado siempre tendrá las entradas «..» y «...»).
- **Delete** (eliminar): elimina un directorio, siempre que este esté «vacío» (es decir, solo contenga «..» y «...»).

<sup>3</sup>La ruta absoluta de un fichero es única siempre que sea «mínima», es decir, siempre que el carácter separador no aparezca repetido varias veces seguidas y siempre que no se utilicen las expresiones «..» y «...», típicas de las rutas relativas. Por ejemplo, la ruta // /usr/... /etc// /passwd sería una ruta absoluta válida en Linux, pero no mínima; la ruta absoluta mínima equivalente sería /etc/passwd.

## CAPÍTULO 4. SISTEMAS DE FICHEROS

- **Opendir** (abrir directorio): abre un directorio para ser recorrido, es decir, obtener una lista de los nombres de los ficheros y subdirectorios que contiene.
- **Closedir** (cerrar directorio): cierra un directorio previamente abierto. Será necesario abrir de nuevo el directorio para poder recorrerlo.
- **Readdir** (leer directorio): devuelve la siguiente *entrada* (es decir, nombre de fichero o directorio) de un directorio abierto.
- **Rename** (cambiar nombre): cambia de nombre y/o directorio un directorio.
- **Link** (ligar o enlazar): permite que un mismo fichero aparezca a la vez con nombres diferentes en un mismo directorio, o en varios directorios con el mismo nombre o nombres distintos. Hablaremos más de esta operación en la sección 4.4.4.
- **Unlink** (desligar): elimina una entrada del directorio. Si esta es la única entrada que existe para el fichero (es decir, el fichero no tiene más de un nombre) entonces equivale a borrar el fichero, liberando el espacio que ocupa en el almacenamiento secundario. También hablaremos más de esta operación en la sección 4.4.4.

Obsérvese que, aunque el directorio sea un fichero, no se utiliza *read* para acceder a él, sino *readdir*. La razón es que, como hemos dicho, un directorio es un fichero manejado por el propio sistema operativo. Por lo tanto, es este el que determina su formato. Este formato puede cambiar de un sistema operativo a otro, de un sistema de ficheros a otro (como veremos más adelante) e, incluso, de una versión a otra de un mismo sistema operativo. Si permitiéramos que los programas de usuario leyeron los directorios con *read*, como si de ficheros se tratase, entonces dichos programas deberían conocer los detalles de la estructura interna. Es más, se tendrían que modificar y compilar de nuevo si la estructura cambiara.

La modificación con *write* de un directorio tendría el mismo problema. Observa, sin embargo, que no existe una operación *writedir* para cambiar el contenido de un directorio. Esto es así porque las modificaciones se hacen indirectamente a través de otras llamadas al sistema, como las que permiten crear, borrar o renombrar un fichero o directorio. Por ejemplo, si creamos un fichero, aparecerá una entrada con su nombre en el directorio en el que se crea, por lo que se estará modificando dicho directorio.

Nótese que el hecho de que los directorios sean ficheros tiene algunas implicaciones adicionales. Así, si los ficheros tienen atributos, los directorios también los tendrán, aunque, para algunos atributos, el significado podría cambiar ligeramente.

} Por trasparencia  
y portabilidad.

### 4.4 IMPLEMENTACIÓN DEL SISTEMA DE FICHEROS

Existe una gran variedad de dispositivos de almacenamiento secundario (discos duros, discos SSD, memorias USB, ...), cada uno construido sobre su propia tecnología y, por tanto, con funcionamientos internos bastante distintos. Para facilitar el uso de estos dispositivos tanto a los usuarios como a los propios sistemas de ficheros, muchos de ellos exportan una interfaz que permite ver a cada dispositivo como un simple *array lineal de bloques*. Hay, sin embargo, dispositivos que no proporcionan esta interfaz. En estos casos, es el propio sistema operativo el encargado de crearla, por lo que, al final, todos los dispositivos de almacenamiento secundario se ven como arrays lineales de bloques. Esto hace que estos dispositivos también se conozcan como *dispositivos de bloques*.

#### 4.4. IMPLEMENTACIÓN DEL SISTEMA DE FICHEROS

Lecturas y escrituras  
se hacen en unidades  
de bloques.

Se pueden leer 20  
bytes? NO  
Mínimo 1 bloque.

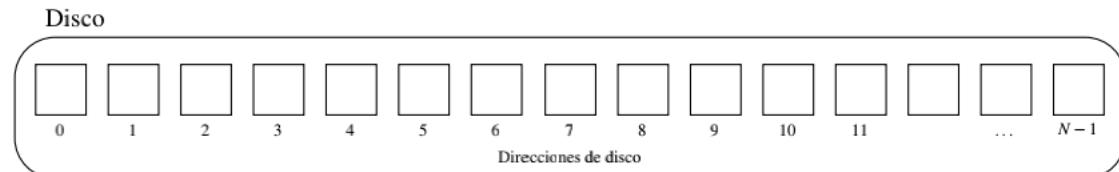


Figura 4.3. Un disco visto como un array de bloques.

Como podemos ver en la figura 4.3, un disco visto como un array lineal de bloques tiene  $N$  bloques, desde 0 hasta  $N - 1$ , todos del mismo tamaño. Tanto el tamaño como el número total de bloques dependen del dispositivo y su capacidad (ver sección 4.4.5 para más detalles). Cada bloque tiene también una posición dentro del array que lo identifica. Puesto que el array no deja de ser una representación de un disco, a esta posición la llamaremos *dirección de disco* del bloque o, simplemente, *dirección*.

Una vez que todo dispositivo de almacenamiento secundario se ve como un array lineal de bloques, el objetivo ahora de un sistema de ficheros es usar dicha estructura de datos para almacenar en ella información usando ficheros, y organizar dichos ficheros usando directorios. Como los directorios son en el fondo ficheros, un aspecto importante será decidir cómo implementar los ficheros, es decir, dado un fichero, saber en qué bloques de disco (o sea, en qué bloques del array) se almacena su información.

Veamos a continuación distintas formas de implementar ficheros y directorios, así como otros aspectos relacionados con los sistemas de ficheros.

##### 4.4.1 Implementación de ficheros

Todo fichero tiene asociado un conjunto de bloques de disco donde guarda sus datos. Cuando hablamos de implementación de ficheros estamos hablando de una forma de llevar un registro de esos bloques, es decir, de las direcciones de los bloques de disco que pertenecen al fichero. En ese registro hay que tener en cuenta el orden de los bloques dentro del fichero. De esta forma, por ejemplo, si un fichero tiene 10 bloques, habrá que saber dónde están almacenados en disco su bloque 0, su bloque 1, y así hasta llegar a su bloque 9. Puesto que hay varias formas de llevar ese registro, existen varias implementaciones.

no se comparten  
bloques entre  
ficheros.

###### Asignación adyacente o contigua

Es el esquema más sencillo. En esta implementación, cada fichero se almacena como un conjunto de bloques adyacentes en disco, donde el primer bloque del fichero aparece en primer lugar, el segundo bloque del fichero se encuentra en disco justo tras el primero, el tercer bloque justo tras el segundo y así sucesivamente (ver figura 4.4).

Esta implementación posee dos ventajas. Una es que es de fácil implementación, ya que el registro de la localización de los bloques de un fichero se reduce a recordar un solo número: la dirección en disco del primer bloque del fichero. Sabiendo esto se sabrá que el segundo bloque estará en la siguiente dirección de disco, el tercer bloque a continuación y así sucesivamente. En general, si el primer bloque del fichero (el 0) se encuentra en la dirección  $D$  del disco, entonces su bloque  $X$  (contando desde 0), se encontrará en la dirección  $D + X$  del disco.

La otra ventaja es que esta implementación ofrece un rendimiento excelente, especialmente en discos duros, ya que, al estar todos los bloques de un fichero juntos en disco, el recorrido de ese fichero supondrá muy pocos movimientos del brazo del disco o ninguno (ya hablaremos de este tipo de movimientos en el tema dedicado a E/S).

## CAPÍTULO 4. SISTEMAS DE FICHEROS

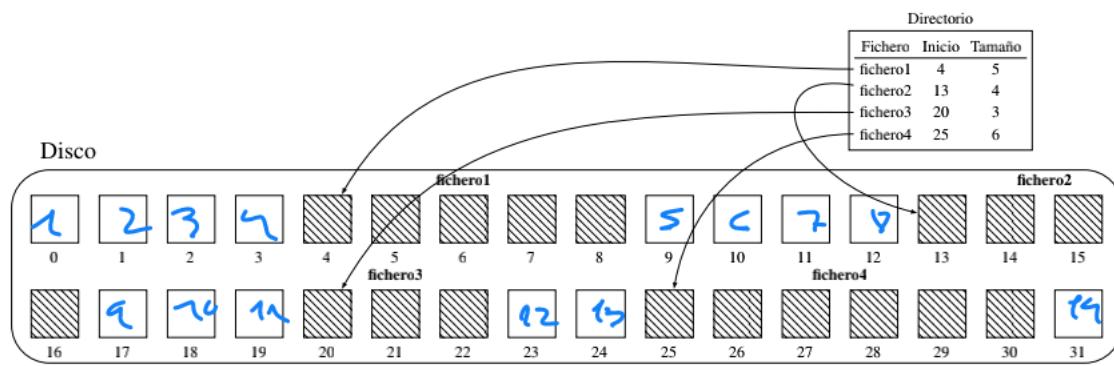


Figura 4.4. Asignación contigua del espacio en disco.

Frag. externa  
14 libres, no puedes meter uno de 5, no estén juntos.

Además, es posible leer o escribir varios bloques consecutivos del fichero en una única operación de disco, puesto que esos bloques también se encuentran consecutivos en disco. Este tipo de operaciones (lecturas o escrituras de varios bloques consecutivos) es el que ofrece un mejor rendimiento en los discos duros.

Sin embargo, esta implementación también presenta algunos inconvenientes. El primero es que *no es realizable* de manera eficiente en un sistema de ficheros de propósito general, a no ser que se conozca el tamaño máximo de cada fichero en el momento de su creación (lo cual rara vez ocurre). El problema es que, si no se sabe qué tamaño máximo va a tener un fichero, no se sabe qué espacio reservarle. Si se reserva un espacio demasiado pequeño puede ocurrir que, si un fichero quiere crecer, no pueda por estar el siguiente bloque a su último bloque ocupado por otro fichero. Para que el fichero pudiera crecer, habría que moverlo a un hueco mayor, lo cual, además de costoso, no siempre es posible.

Otro problema de esta implementación es el de la *fragmentación externa*<sup>4</sup>: puede ocurrir que el espacio libre esté repartido en huecos pequeños, de tal manera que, a pesar de haber espacio libre suficiente, no se pueda crear un nuevo fichero. Por ejemplo, en la figura 4.4 podemos ver que hay hasta 14 bloques libres. Sin embargo, no podremos crear un fichero de 5 o más bloques por no haber un hueco de ese tamaño.

A pesar de los inconvenientes, esta implementación de ficheros es útil y realizable en algunos casos, por ejemplo, en la creación de DVDs o CD-ROMs, donde se conoce de antemano el tamaño de cada fichero y donde los ficheros no se van a modificar una vez escritos (y, por tanto, no van a cambiar de tamaño).

interna → no ocupa totalmente el bloque, espacio en desuso.

externa → espacio libre, no pertenece a nadie, y no se pierde apoderar.

### Asignación mediante lista ligada

En esta implementación cada fichero se mantiene como una lista ligada de bloques en disco. Como se ve en la figura 4.5, al principio de cada bloque (podría ser también al final) se guarda la dirección de disco del siguiente bloque del fichero, es decir, se guarda un puntero al siguiente bloque.

Las ventajas de esta implementación también son dos. Por un lado, no hay fragmentación externa, ya que se pueden utilizar todos los bloques de disco. Esto es así porque los bloques de un fichero pueden estar dispersos por el disco (no es necesario que estén

<sup>4</sup>Hay dos tipos de fragmentación: interna y externa. La interna surge cuando un fichero no ocupa completamente un bloque que le pertenece. Esto suele ocurrir con el último bloque, ya que es raro que el tamaño de un fichero sea múltiplo del tamaño de bloque. La externa, en cambio surge cuando hay bloques libres que no se pueden ocupar por no estar en huecos suficientemente grandes. Como veremos en el siguiente tema, estas fragmentaciones también pueden aparecer en diversos esquemas de administración de memoria.

#### 4.4. IMPLEMENTACIÓN DEL SISTEMA DE FICHEROS

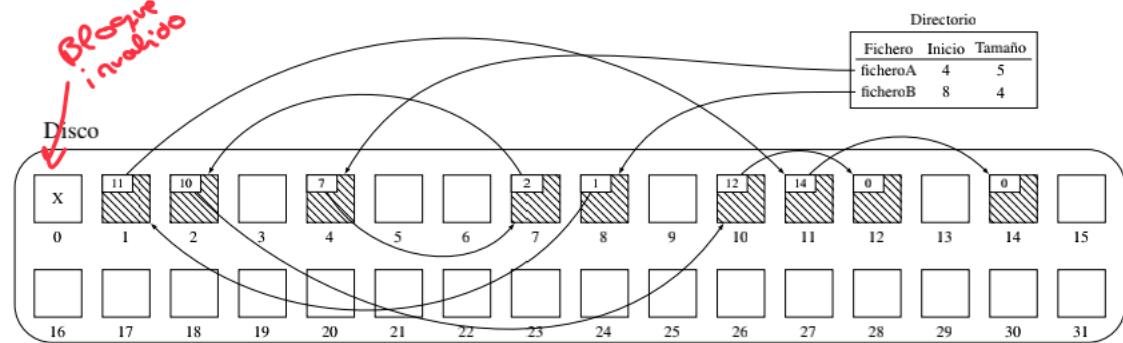


Figura 4.5. Almacenamiento de un fichero como una lista ligada de bloques en disco.

todos juntos, como antes). Por otro lado, es suficiente que la entrada de directorio correspondiente al fichero guarde solo la dirección en disco del primer bloque. A partir de él se puede acceder a todos los demás.

Sin embargo, hay dos inconvenientes importantes. El primero es que, aunque la lectura secuencial es directa (basta con ir leyendo bloques siguiendo la lista dada por los punteros), el acceso aleatorio a un fichero es lento. Por ejemplo, si quisieramos añadir algo al final de un fichero, tendríamos que leer todos sus bloques, pues la dirección de disco del último bloque se guarda en el penúltimo, la dirección de este se guarda en el antepenúltimo y así sucesivamente, hasta llegar al primero, cuya dirección de disco es la única que se guarda en la entrada de directorio.

El segundo inconveniente es que el tamaño del espacio para almacenamiento de datos en un bloque ya no es potencia de dos, puesto que el apuntador ocupa algunos bytes. El problema de esto es que muchos programas leen y escriben en bloques cuyo tamaño es potencia de 2. Así, si los bloques fueran de 512 bytes e hicieran falta 4 bytes para guardar la dirección de disco del siguiente bloque, entonces solo habría disponibles 508 bytes para datos en cada bloque. Un programa que leyera de 1024 en 1024 bytes necesitaría leer siempre 3 bloques al menos, y no dos bloques como pasaría con la asignación adyacente, donde los 512 bytes de cada bloque estarían disponibles para datos.

acceso lento

espacio no es potencia de dos

##### Asignación mediante lista ligada e índice

Una forma de eliminar los dos inconvenientes del esquema anterior es almacenar los punteros en una estructura aparte, por ejemplo, en una tabla o índice en memoria, como se muestra en la figura 4.6. La idea es que en esa tabla o índice haya una entrada por cada bloque de disco, de tal manera que la entrada 0 corresponda al bloque 0, la entrada 1 al bloque 1 y así sucesivamente. Ahora, si la dirección de disco del bloque  $X$  de un fichero es  $D_X$ , la dirección de disco  $D_{X+1}$  de su bloque  $X + 1$  no se guarda al principio de su bloque  $X$ , como antes, sino que se guarda en la entrada  $D_X$  de la tabla. Por ejemplo, el fichero *ficheroA* de la figura 4.6 ocupa los bloques de disco 4, 7, 2, 10 y 12, en ese orden, por lo que la entrada 4 de la tabla almacena un 7, la entrada 7 un 2, y así sucesivamente. La entrada 12, que corresponde al último bloque del fichero, almacena un 0, que no se considera un número de bloque válido en este ejemplo, por lo que se utiliza para indicar el final de la lista.

Con esta técnica se eliminan las dos desventajas del esquema anterior. Ahora todo el bloque está disponible para datos y el acceso aleatorio es mucho más rápido, puesto que, aunque hay que seguir la cadena de punteros para encontrar la dirección de disco de un bloque concreto de un fichero, este recorrido se hace en memoria.

## CAPÍTULO 4. SISTEMAS DE FICHEROS

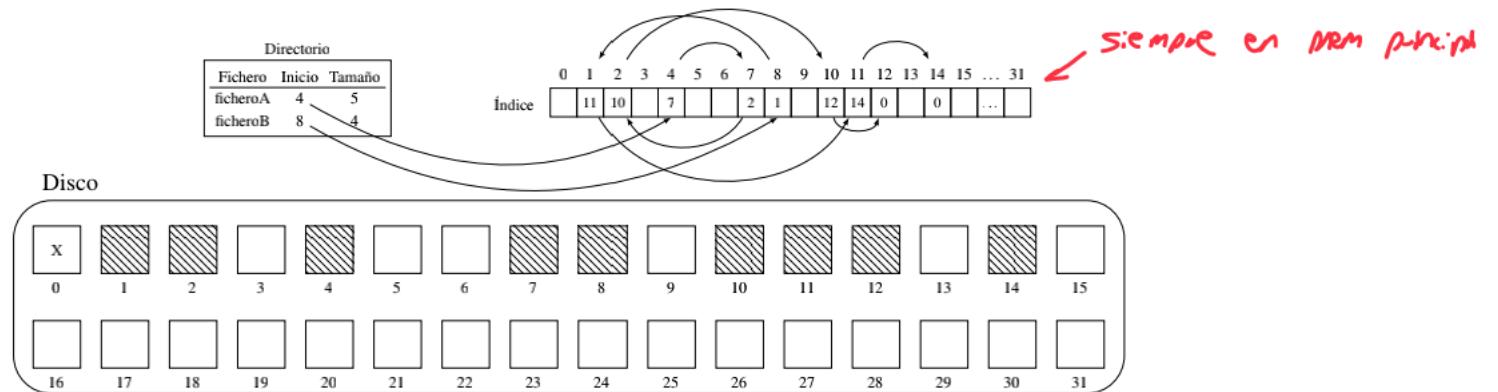


Figura 4.6. Asignación mediante una lista ligada y una tabla en la memoria principal.

La principal desventaja de esta técnica es que toda la tabla debe estar en memoria, lo que puede ser un problema si la tabla es grande debido a un disco de gran capacidad. Una posible solución es utilizar bloques lógicos grandes, lo que hará que el disco tenga menos bloques, pero de mayor tamaño. Si hay menos bloques, la tabla tendrá menos entradas y será más pequeña. El problema de utilizar bloques lógicos grandes es que se desperdicia espacio por fragmentación interna. Hablaremos de los bloques lógicos y su tamaño un poco más tarde.

Otro aspecto a tener en cuenta es que, aunque la tabla esté en memoria, se tiene que guardar en disco para que esté disponible cada vez que se arranca el sistema. Esto puede dar lugar a inconsistencias, ya que, si la tabla se modifica porque, por ejemplo, se crea un fichero, y las modificaciones no se escriben en disco, una caída del sistema podría hacer que ciertos ficheros desaparecieran o quedaran incompletos.

El sistema de ficheros FAT, en sus diferentes versiones, utiliza este método. Recordemos que FAT es el sistema de ficheros que usaba MS-DOS y que es uno de los implementados en Windows.

Observa que si hacemos que ciertos números de bloque sean inválidos (por ejemplo, 0xFFFFFFFF, suponiendo entradas de la tabla de 4 bytes), entonces podríamos usar la propia tabla y esos valores para indicar qué bloques están libres, es decir, no son usados por ningún fichero o directorio. Esta técnica es usada por FAT.

### Nodos-i

En este caso, a cada fichero se le asocia una pequeña tabla llamada *nodo-i* (nodo índice), la cual contiene las direcciones en disco de los bloques del fichero, en el orden de esos bloques dentro del fichero. Este esquema se muestra en la figura 4.7.

Como se ve también en la figura, el nodo-i de cada fichero se almacena en un bloque de disco. Aunque esta opción es posible, no se suele utilizar, pues la inmensa mayoría de los ficheros son pequeños (incluso hay ficheros vacíos) y el nodo-i guardaría unas pocas direcciones de disco, estando el resto del bloque vacío. Por eso, lo habitual es utilizar nodos-i pequeños, capaces de almacenar unas pocas direcciones de disco, y guardar varios nodos-i juntos en un mismo bloque de disco.

El problema entonces, que también surge cuando se usa todo un bloque como nodo-i, es qué pasa con los ficheros grandes, que tienen más bloques que direcciones caben en el nodo-i. La solución es utilizar bloques indirectos, como se muestra en la figura 4.8. La idea es utilizar el nodo-i para almacenar las direcciones de disco de los primeros bloques del fichero correspondiente. Para ficheros pequeños, las direcciones de todos sus bloques se almacenarán en el nodo-i. Para ficheros un poco más grandes, una de

#### 4.4. IMPLEMENTACIÓN DEL SISTEMA DE FICHEROS

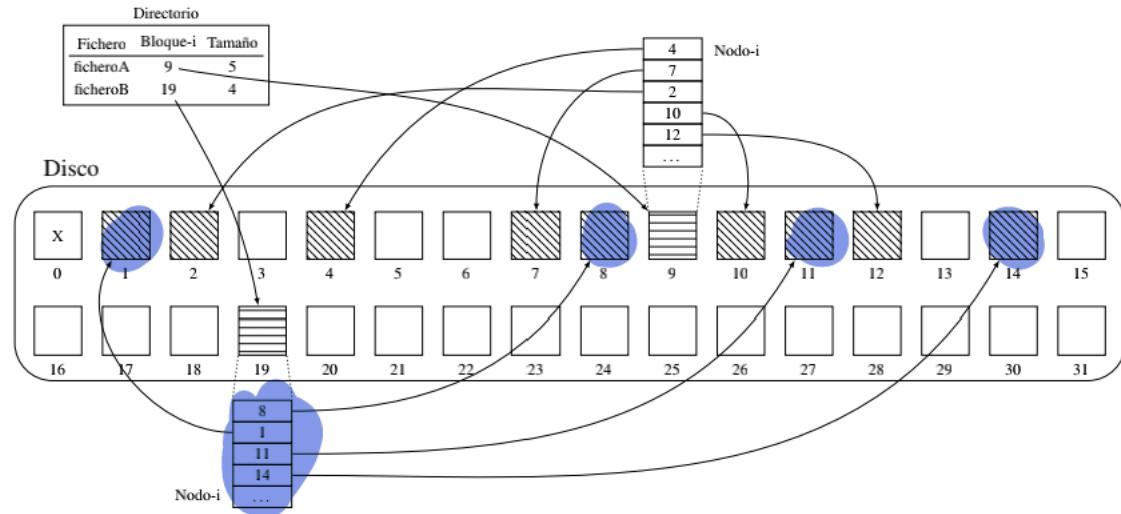


Figura 4.7. Asignación mediante nodos-i del espacio en disco.

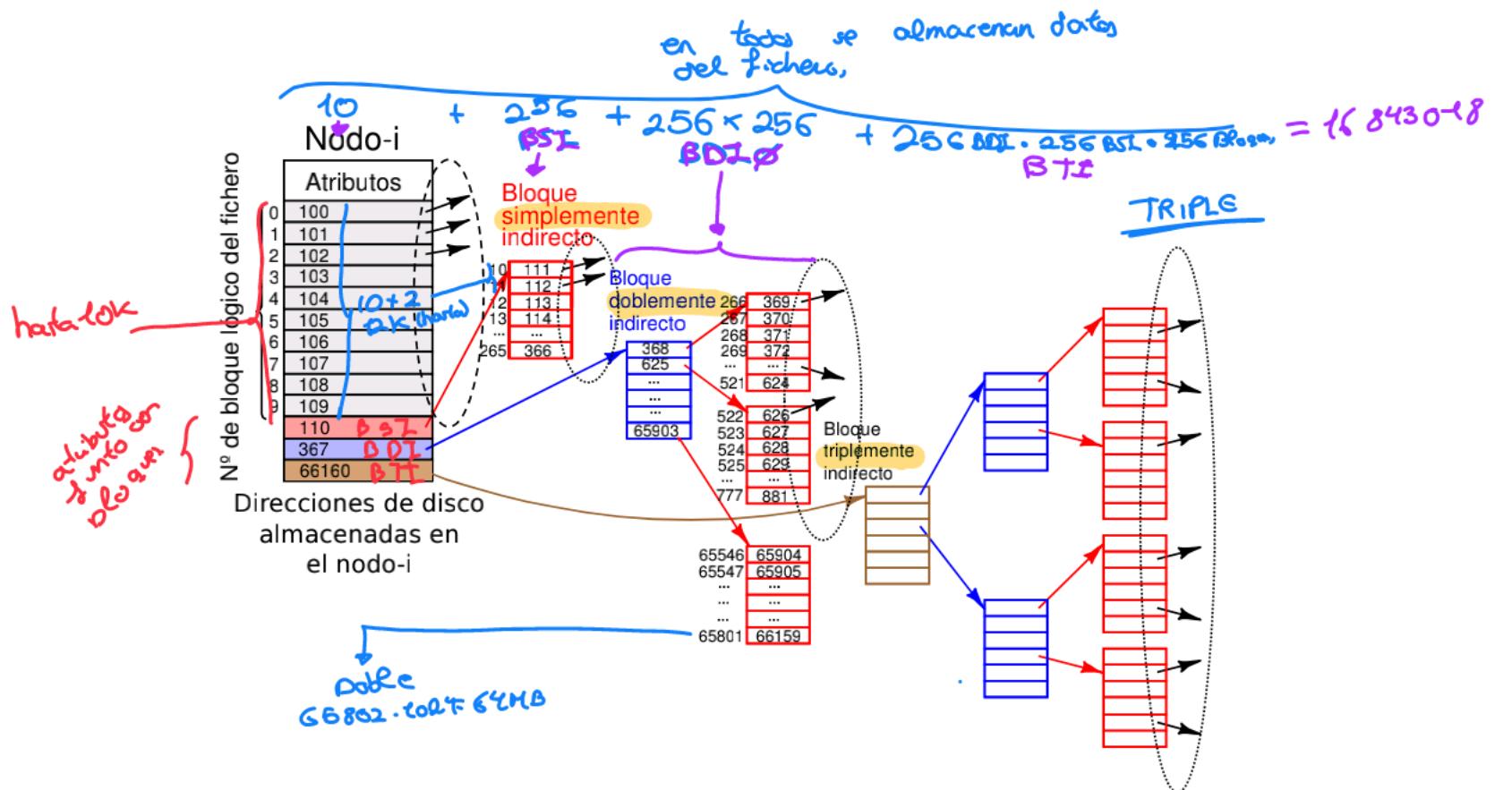


Figura 4.8. Nodo-i según se usa en Unix. Los bloques que contienen la información del fichero no aparecen en la figura (serían los apuntados por las flechas que quedan dentro de las elipses).

## CAPÍTULO 4. SISTEMAS DE FICHEROS

las direcciones en el nodo-i será la dirección de un bloque en el disco, llamado **bloque simplemente indirecto** (BSI), que contendrá las direcciones en disco de bloques de datos adicionales del fichero. Si lo anterior no es suficiente, otra dirección en el nodo-i será la dirección de un **bloque doblemente indirecto** (BDI), cuyo contenido serán las direcciones de más bloques simplemente indirectos. Si aún no es suficiente, se puede utilizar un **bloque triplemente indirecto** (BTI), que contendrá direcciones de más bloques doblemente indirectos. Unix utiliza este esquema.

Bloque 100 libre  
10 primeros 100 - 109

En el caso de la figura 4.8 suponemos un nodo-i capaz de guardar 13 direcciones de disco (10 de bloques de datos y 3 de bloques indirectos), direcciones de disco de 4 bytes y bloques de 1 KiB (lo que hace que en un bloque quepan  $\frac{1024}{4} = 256$  direcciones). Como podemos ver, los 10 primeros bloques del fichero (del 0 al 9) se almacenan en los bloques de disco del 100 al 109. Estas direcciones se guardan en el propio nodo-i. Los siguientes 256 bloques del fichero (del 10 al 265) se almacenan en los bloques de disco del 111 al 366. Sus direcciones de disco se guardan en el primer bloque simplemente indirecto, el cual, a su vez, se almacena en el bloque de disco 110. La dirección de este primer BSI se guarda en el nodo-i. Los siguientes 256 bloques del fichero (del 266 al 521) se guardan en los bloques de disco del 369 al 624. Sus direcciones de disco se almacenan en el segundo BSI el cual, a su vez, se guarda en el bloque 368 de disco. Esta dirección se almacena en el primer bloque doblemente indirecto, que usa el bloque 367 de disco. La dirección de este bloque se guarda en el nodo-i. El resto de la figura muestra más direcciones de disco de bloques del fichero que se almacenan en bloques BSI, cuyas direcciones de disco se guardan en el BDI. También se ve que, según el nodo-i, hay un bloque triplemente indirecto almacenado en el bloque de disco 66160. Este bloque almacenaría las direcciones de disco de más bloques BDI (estas direcciones ya no se muestran) que, a su vez, almacenarían más direcciones de disco de más bloques BSI que, a su vez, guardarían más direcciones de disco de bloques de datos del fichero.

Las ventajas de este método con respecto a la lista ligada con índice son diversas. Por ejemplo, utilizando los datos de la figura 4.8 (bloques de 1 KiB, direcciones de disco de 4 bytes y nodo-i con 10+3 direcciones en total), se puede calcular que con este método un fichero puede tener hasta  $16\ 843\ 018^5$  bloques de datos de 1 KiB cada uno, lo que representa un tamaño de poco más de 16 Gbytes. Aún así, acceder a cualquier posición del fichero supone, como mucho, leer 4 bloques de disco: el triplemente indirecto, uno doblemente indirecto, uno simplemente indirecto y el bloque que contiene los datos a los que queremos acceder. Con el método de la lista ligada con índice necesitaríamos, solo para el fichero, un índice con un tamaño superior a  $64\ \text{MiB}^6$  para poder guardar todas las direcciones de bloques de datos. Este índice tendría que estar completamente en memoria para poder acceder al fichero. Además, leer o escribir las últimas posiciones del fichero supondría recorrer una lista de  $16\ 843\ 018$  posiciones.

En los sistemas de ficheros que usan nodos-i de este tipo, todos los nodos-i se guardan juntos en bloques consecutivos de disco a los que se llama *tabla de nodos-i*<sup>7</sup>. Cada nodo-

<sup>5</sup>Este dato se calcula de la siguiente manera. El nodo-i es capaz de almacenar las direcciones de 10 bloques de datos. El BSI cuya dirección se almacena en el nodo-i guarda las direcciones de disco de otros 256 bloques de datos. El BDI cuya dirección está en el nodo-i almacena las direcciones de 256 bloques BDI, cada uno de los cuales es capaz de almacenar 256 direcciones de bloques de datos. Por lo tanto, a través del BDI con dirección en el nodo-i podemos acceder a  $256 \cdot 256 = 65\ 536$  bloques de datos. Finalmente, el BTI cuya dirección está en el nodo-i almacena las direcciones de 256 bloques BDI, cada uno de los cuales nos da acceso a 65 536 bloques de datos, como acabamos de calcular. En total, podemos almacenar las direcciones de  $10 + 256 + 256 \cdot 256 + 256 \cdot 256 \cdot 256 = 16\ 843\ 018$  bloques de datos.

<sup>6</sup>Si cada dirección de disco ocupa 4 bytes, 16 843 018 direcciones ocuparán 67 372 072 bytes o, lo que es lo mismo, unos 64.25 MiB.

<sup>7</sup>Por razones de eficiencia, esta tabla muchas veces está dividida en varios trozos repartidos por todo

#### 4.4. IMPLEMENTACIÓN DEL SISTEMA DE FICHEROS

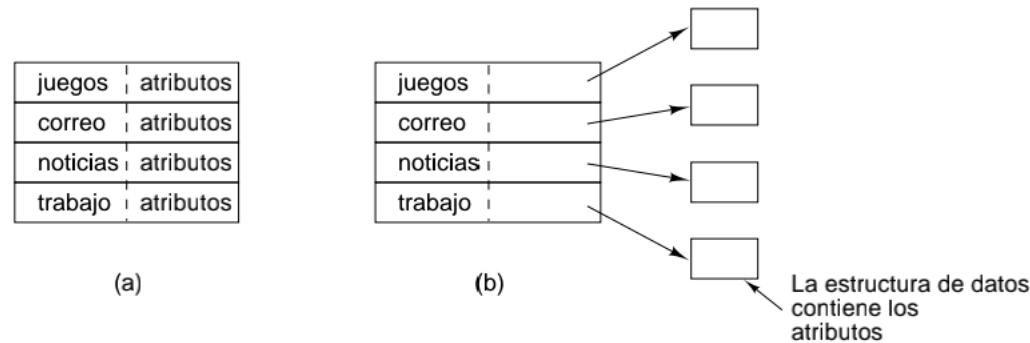


Figura 4.9. Directorios. (a) Atributos en la entrada del directorio. (b) Atributos en otro lugar.

i recibe un número que es el que se guarda en la entrada de directorio correspondiente a su fichero. Así, sabiendo el número de nodo-i al que queremos acceder y cuántos nodos-i caben en un bloque, podemos saber qué bloque de la tabla de nodos-i tenemos que leer, y la posición de nuestro nodo-i dentro de ese bloque.

Un nodo-i se lee cuando se abre su fichero correspondiente y solo tiene que estar en memoria mientras el fichero esté abierto. Por lo tanto, la memoria necesaria para almacenar nodos-i es independiente del tamaño del sistema de ficheros y solo depende del número de ficheros abiertos en un momento dado.

La figura 4.8 nos muestra que, además de direcciones de disco, un nodo-i en Unix también guarda los atributos del fichero correspondiente. Como veremos en las siguientes secciones, esto determina la forma en la que se implementan los directorios y los enlaces de fichero.

##### 4.4.2 Implementación de directorios *(Acceso a los atributos del fichero)*

La principal función de un directorio es la de asociar el nombre de un fichero con la información necesaria para localizar los datos de dicho fichero. De esta manera, al abrir un fichero para acceder a él, el sistema operativo utilizará su ruta para localizar la entrada del fichero en su directorio y obtener, a partir de ella, todo lo que necesita: los atributos del fichero, las direcciones de sus bloques en disco, etc. Toda esta información se almacena en una tabla, dentro de la memoria principal, hasta que el fichero se cierra. Cualquier operación posterior sobre el fichero utilizará la información que hay en la tabla.

Un aspecto estrechamente relacionado con lo anterior es dónde se guardan los atributos de un fichero. Básicamente, tenemos dos posibilidades, que se muestran en la figura 4.9: (a) almacenarlos junto con el nombre, en la propia entrada de directorio (como hace MS-DOS), o (b) almacenarlos en una estructura aparte, como puede ser un nodo-i si se utiliza esta implementación de ficheros (como hace Unix).

##### Directorios en MS-DOS

En MS-DOS, los directorios son ficheros que almacenan una lista desordenada de entradas (o registros) de 32 bytes, una por fichero. Cada una de estas entradas se divide en varios campos, como se muestra en la figura 4.10. Estos campos son los siguientes:

- Nombre y extensión del fichero: los primeros 11 bytes.

el disco.

## CAPÍTULO 4. SISTEMAS DE FICHEROS

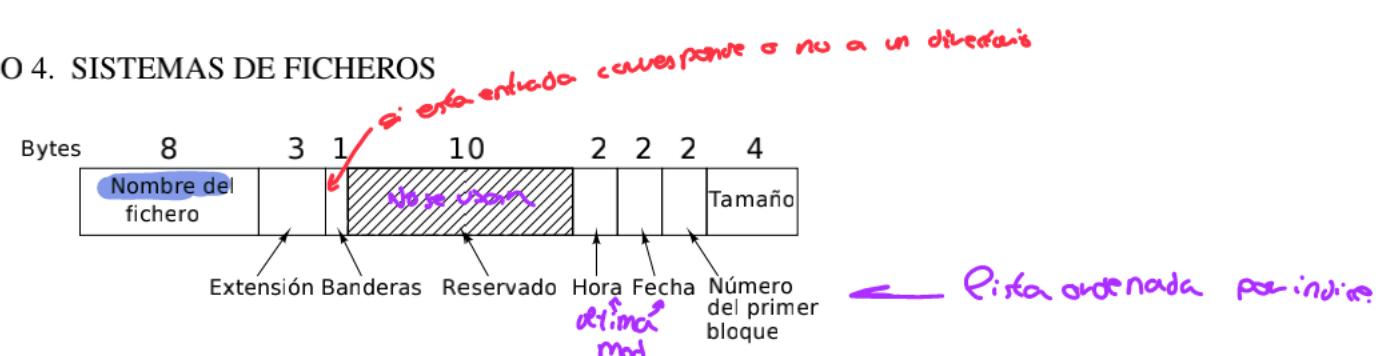


Figura 4.10. Una entrada de directorio en MS-DOS.

- Atributos:** un byte donde varios bits se utilizan como banderas. En concreto, uno de ellos distingue a un directorio de un fichero normal.

- Un campo de 10 bytes reservado para usos futuros.**

- Hora de la última modificación del fichero.** Observa que este campo tiene 16 bits. Sin embargo, para poder guardar la hora, necesitaríamos 5 bits para la hora (de 0 a 23), 6 bits para los minutos (de 0 a 59) y otros 6 bits para los segundos (de 0 a 59), lo que en total suma 17 bits. La solución pasa por guardar solo segundos pares, por lo que bastan 5 bits que, junto con los bits de la hora y los minutos, hacen un total de 16 bits, que es el tamaño del campo.

← 0-23 → 5 bits  
 ← 0-59 → 6 bits  
 ← 0-59 → 6 bits →  
 OJO  
 SOLO SEGUNDOS  
 PARES

- Fecha** de la última modificación del fichero. Este campo tiene un tamaño de 16 bits. Los primeros 7 bits almacenan el año, correspondiendo el valor 0 al año 1980. Los siguientes 4 bits almacenan el mes (de 1 a 12) y los últimos 5 bits almacenan el día del mes (desde 1 hasta 31).

dia 0-31 → 5 bits  
 mes 0-11 → 4 bits  
 año → 7 bits. ← año a partir  
 de 1980 → mes 1980 + 128  
 2008

- Número del primer bloque del fichero**, es decir, dirección de disco donde comienza el fichero. El resto de direcciones se obtiene recorriendo la FAT que, como hemos dicho, es una lista ligada con índice.

Tam max 4GB

- Tamaño** del fichero, en bytes. Al ser este campo de 4 bytes, solo tenemos 32 bits para el tamaño, por lo que ningún fichero podrá ser más grande de 4 GiB (este tamaño solo se alcanzará si otros límites, como el tamaño de la partición que aloja al sistema de ficheros, lo permiten).

El directorio raíz es una excepción, ya que ocupa unos bloques fijos en disco en lugar de implementarse como un fichero, lo que hace que tenga un tamaño máximo preestablecido.

MS-DOS permite crear un árbol de directorios de tamaño arbitrario. Observa que un directorio puede tener ficheros que en realidad sean otros directorios (es decir, subdirectorios de él). Como ya hemos dicho, un bit del campo de atributos permite distinguir a los ficheros normales de los ficheros que son directorios.

Esta estructura de directorios de MS-DOS se ha mantenido hasta nuestros días y es utilizada, con diferentes extensiones y mejoras, por los sistemas de ficheros FAT32 y vFAT.

### Directarios en Unix

En Unix, al igual que en MS-DOS, los directorios son ficheros gestionados por el propio sistema de ficheros. También es posible crear un árbol de directorios utilizando un método similar, ya que uno de los bits de los atributos almacenados en los nodos i permite distinguir a un fichero normal de un directorio. En Unix, sin embargo, el

#### 4.4. IMPLEMENTACIÓN DEL SISTEMA DE FICHEROS

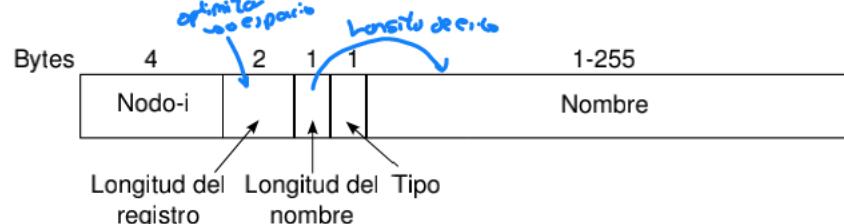


Figura 4.11. Una entrada de directorio en Unix.

directorio raíz no tiene ningún tratamiento especial y es un fichero como cualquier otro directorio.

En las primeras versiones de UNIX, cada entrada de directorio tenía un tamaño fijo de 16 bytes, dos de los cuales se utilizaban para guardar el número de nodo-i asociado al fichero y los otros 14 bytes almacenaban el nombre, por lo que los ficheros podían tener un nombre de hasta 14 caracteres. Posteriormente, se han creado sistemas de ficheros para Unix que soportan nombres de hasta 255 caracteres. En este caso, las entradas ya no tienen un tamaño fijo y se estructuran en varios campos. La figura 4.11 muestra una de estas posibles entradas de directorio con los siguientes campos:

- Un número de nodo-i, que ocupa los primeros 4 bytes.
- La longitud del registro, que indica el tamaño de toda la entrada en bytes. Es habitual que esta longitud sea múltiplo de 4.
- La longitud del nombre. Como se utiliza un único byte para este campo, el nombre no puede exceder los 255 caracteres (o menos, si se utilizan caracteres multibyte como los de UTF-8).
- El tipo de la entrada: si se trata de un directorio, fichero regular o cualquier otra cosa. Este campo es una copia del que hay almacenado en el nodo-i y se utiliza para acelerar los listados de directorio.
- El nombre del fichero.

Como se ve, una entrada fundamentalmente contiene el nombre de un fichero y su número de nodo-i. Toda la información relativa al tipo, tamaño, tiempos, propiedad y bloques en disco del fichero está contenida en su nodo-i.

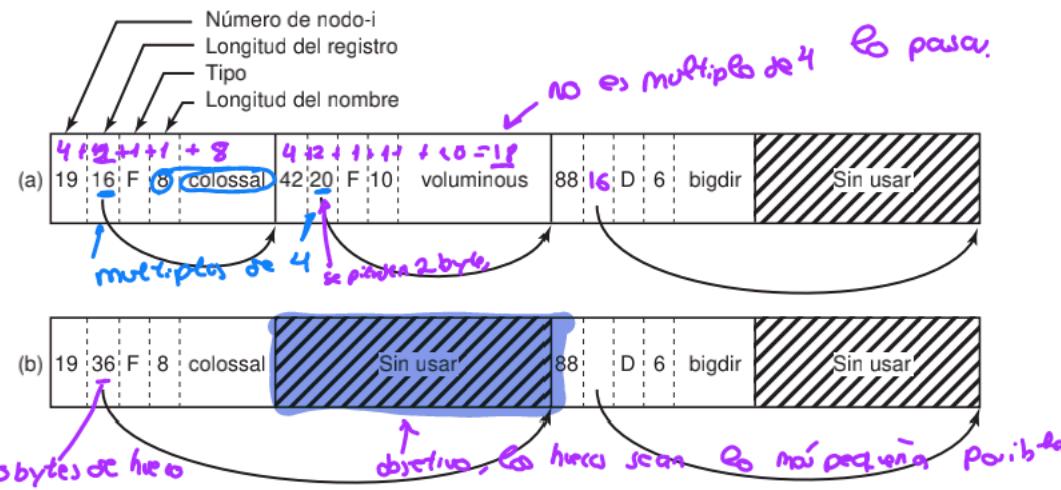
Una pregunta que nos puede surgir es para qué necesitamos el campo con la longitud del registro. En teoría, sumando el tamaño del resto de campos, podemos obtener el tamaño total del registro. Sin embargo, este campo nos permite tener registros más grandes de lo estrictamente necesario. Hay varias razones para querer poder hacer esto.

Una es que, si renombramos una entrada y usamos un nombre más corto, los bytes que sobren en el campo del nombre seguirán formando parte del registro y no se generará un hueco que, posiblemente, no se podrá aprovechar para otro registro por ser excesivamente pequeño. Además, si el tamaño del registro es múltiplo de 4 como hemos dicho, será frecuente que en el campo del nombre siempre haya algún byte libre. Esto nos permitiría renombrar una entrada a un nombre con uno, dos y hasta tres caracteres más (dependiendo de los bytes libres) sin tener que mover el registro a un hueco mayor. Es decir, al tener un campo que guarda la longitud del registro, suele ser posible renombrar entradas sin tener que cambiar el tamaño del registro y, por tanto, sin tener que borrar, crear y copiar registros.

Otra razón es la gestión del espacio libre dentro del fichero que contiene al directorio. Así, como ya hemos comentado, cuando se crea un directorio, este no está totalmente

## CAPÍTULO 4. SISTEMAS DE FICHEROS

**Todos registros  
multiplos de 4**



si borraras bigdir  
+16 de hueco.

Figura 4.12. Eliminación de una entrada de directorio en Unix. (a) Tres entradas de directorio donde el campo «Longitud del registro» nos permite ir saltando fácilmente de una entrada a la siguiente. (b) Eliminación de la entrada «voluminous». Se observa que el espacio ocupado por esta entrada se añade al tamaño del registro de la entrada anterior.

vacio sino que tiene dos entradas especiales que son «..» y «.». Cada una de ellas ocupa unos pocos bytes y cada una se guarda en un registro. Ahora bien, si el único bloque que tiene el directorio donde se guardan estas entradas es de, por ejemplo, 4096 bytes, ¿dónde se registra el espacio que queda libre? La respuesta es que el registro asociado a la entrada «..» ocupa todo el espacio que queda tras crear la entrada «..». Si ahora se crea un fichero, el sistema de ficheros buscará un registro con suficiente espacio libre para crear un nuevo registro, dividiendo el registro existente en dos: uno para la entrada que ya existe y otro para la nueva entrada que, además, se quedará con el resto del espacio disponible. Cualquier otra entrada nueva se crea siguiendo el mismo procedimiento. Si no hay ningún registro con espacio libre suficiente, entonces se añade un nuevo bloque de datos al directorio.

Un aspecto a tener en cuenta es que un registro solo puede estar en un bloque, es decir, un registro no puede atravesar la frontera entre bloques.

El campo con la longitud del registro también facilita el borrado de entradas. Basta con añadir el espacio que queda libre al registro que hay justo antes, es decir, basta con hacer una fusión de registros (ver figura 4.12). Si no hay ningún registro previo, entonces el registro se marca como libre utilizando un número de nodo-i inválido (habitualmente, 0).

### 4.4.3 Resolución de rutas

Aprovechando la implementación de ficheros y directorios de Unix, vamos a ver cómo se resuelve una ruta, es decir, dada la ruta de un fichero, cómo llegamos finalmente al número de nodo-i asociado a dicho fichero. Aunque lo veamos para Unix, el algoritmo es esencialmente el mismo para todos los sistemas jerárquicos de directorios.

La figura 4.13 muestra la resolución de la ruta /usr/ast/mbox, donde suponemos que cualquier directorio tiene solo un bloque de datos, y que tanto el nodo-i del directorio raíz como su bloque de datos se encuentran ya en memoria. Los pasos que se dan son los siguientes:

1. Buscamos en el bloque de datos del directorio raíz la entrada correspondiente a

#### 4.4. IMPLEMENTACIÓN DEL SISTEMA DE FICHEROS

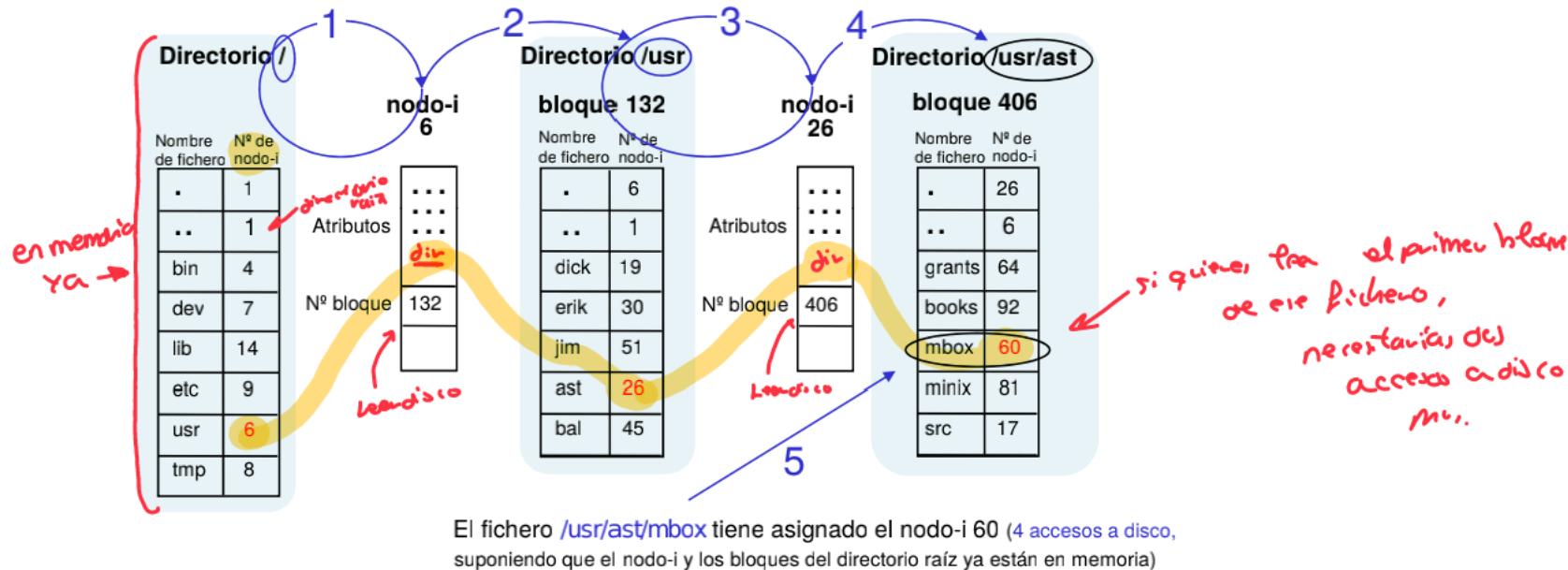


Figura 4.13. Pasos para obtener el número de nodo-i asociado al fichero `/usr/ast/mbox`.

/usr. Una vez localizada la entrada, de ella obtenemos su número de nodo-i: el 6. Leemos dicho nodo-i de disco.

2. Del nodo-i 6 obtenemos la dirección del bloque de datos del directorio /usr. *Sin trámite no es válido*
3. Ahora, en el bloque de datos de /usr, buscamos la entrada ast, correspondiente al directorio /usr/ast. Esta entrada nos dice que su nodo-i es el 26. Leemos el nodo-i de disco.
4. Del nodo-i 26 obtenemos la dirección del bloque de datos del directorio /usr/ast. Este bloque es el 406, el cual también se lee de disco.
5. Finalmente, en el bloque de datos de /usr/ast, buscamos la entrada mbox, correspondiente al fichero /usr/ast/mbox, la cual nos dice que el nodo-i del fichero es el 60.

Por lo tanto, hemos necesitado 4 accesos a disco para, dada la ruta `/usr/ast/mbox`, obtener el número de su nodo-i (recordemos que estamos suponiendo que el nodo-i del directorio raíz y su bloque de datos se encuentran ya en memoria). Si quisieramos leer el nodo-i en sí, necesitaríamos un acceso adicional a disco.

Puesto que las resoluciones de rutas son algo muy frecuentes, e incluso una misma ruta puede ser resuelta muchas veces en un corto espacio de tiempo, para acelerar su procesamiento el sistema operativo suele mantener en memoria información de las últimas resoluciones realizadas (en Linux, esta memoria se llama *dentry cache* o *caché de entradas de directorio*).

Los nombres relativos se buscan de la misma manera, pero comenzando en el directorio de trabajo, no en el directorio raíz. Los directorios `..` y `...` no tienen tratamiento especial, se buscan como cualquier otro nombre, ya que existen entradas reales para ellos (como ya hemos comentado y también se puede apreciar en la figura 4.13).

Examen

cuantos accesos ruta + 1

## CAPÍTULO 4. SISTEMAS DE FICHEROS

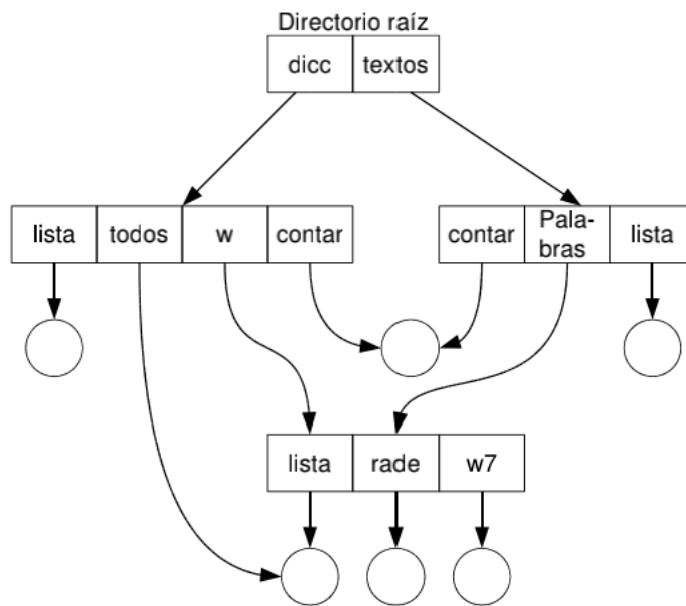


Figura 4.14. Sistema de ficheros con ficheros y directorios compartidos.

### 4.4.4 Ficheros compartidos

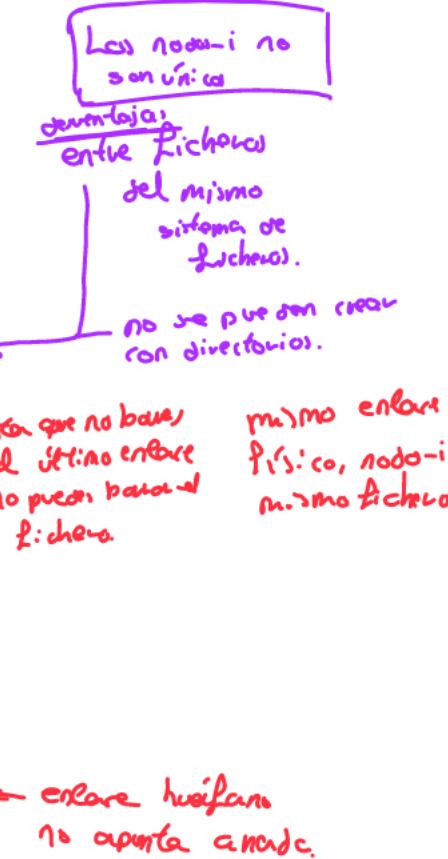
Cuando un **mismo fichero** puede aparecer en varios directorios, con el mismo o distinto nombre, o en un mismo directorio con nombres distintos, se dice que es un **fichero compartido**. La conexión entre un directorio y un fichero compartido se llama **enlace**. El sistema de ficheros deja de tener una estructura de árbol para pasar a ser una gráfica dirigida acíclica, como se puede ver en la figura 4.14.

Los ficheros compartidos se pueden implementar básicamente de dos formas. La primera es que los **datos relativos a un fichero** (atributos, hora, fecha, tamaño, bloques en disco) **se guarden en una estructura de datos** y las entradas de los directorios **contengan apunadores a esa estructura**. Este es el caso de Unix con los nodos-i en donde es posible que varias entradas de un mismo directorio o de directorios distintos guarden un mismo número de nodo-i. A este tipo de enlace se le conoce como **enlace físico** (o *hard link*).

Es evidente que para que esta primera implementación funcione es necesario que **exista un contador** de enlaces en el nodo-i (o estructura equivalente). Así, **si se elimina un enlace** borrando un fichero y hay más enlaces (es decir, más nombres para el mismo fichero, todos ellos con el mismo número de nodo-i), el nodo-i no se debe liberar, pues, de lo contrario, ese nodo-i podría ser utilizado por otro fichero que no tendría nada que ver con el fichero original.

La segunda forma de implementar ficheros compartidos es creando un nuevo tipo de fichero cuyo contenido sea la ruta de acceso del fichero al que se enlaza. A estos nuevos ficheros se les llama **enlaces simbólicos** (o *soft links*). Al igual que los directorios, estos ficheros especiales se distinguen del resto de ficheros mediante un bit de bandera presente en el campo de atributos. En este esquema, cuando el fichero original se borra, el uso posterior del fichero a través de un enlace simbólico simplemente fallará, ya que el enlace apuntará a un fichero inexistente. Obsérvese, sin embargo, que si se crea un nuevo fichero con el mismo nombre, el enlace simbólico volverá a estar operativo, aunque posiblemente apuntará a un fichero con un contenido totalmente diferente.

El problema de los enlaces simbólicos es su elevado coste, tanto temporal (se tiene que resolver la ruta del enlace simbólico y, después, la ruta del fichero real), como de espacio (hace falta un nodo-i para cada enlace y un bloque en disco para guardar la ruta de acceso aunque, por optimización, dicho dato podría estar en el propio nodo-i).



Mismo contenido.  
En

Diferentes contenidos  
En -> 23

cat 3 → sobrehaciendo  
cat 2.

## 4.4. IMPLEMENTACIÓN DEL SISTEMA DE FICHEROS

Los enlaces simbólicos, sin embargo, son más flexibles que los enlaces físicos y no presentan algunos problemas de estos. Por ejemplo, Unix no permite crear enlaces físicos de directorios, pero sí simbólicos, y no permite crear un enlace físico a un fichero en un sistema de ficheros distinto al suyo<sup>8</sup>, pero sí permite crear un enlace simbólico.

Un problema de todos los enlaces es el riesgo de duplicar datos. Por ejemplo, el programa de copia de seguridad puede, salvo que se tomen las medidas oportunas (como detectar la presencia de enlaces), copiar un fichero compartido varias veces, una vez por enlace.

### 4.4.5 Administración del espacio en disco

En este apartado vamos a ver dos aspectos relacionados con la administración del espacio en disco: el tamaño del bloque lógico y el registro de los bloques libres.

#### Tamaño de bloque lógico

No se  
pueden  
combinar

Los discos son dispositivos de bloques en los que la unidad mínima de lectura y escritura es el bloque. Estos **bloques físicos** (que en la terminología de los discos se llaman **sectores**), suelen tener tamaños de 512, 1024, 2048 e, incluso, 4096 bytes. Como ya hemos explicado, para facilitar el uso de los discos, bien los propios dispositivos, bien el sistema operativo, hacen que estos se vean como un array lineal de bloques físicos.

Ahora bien, es común que el sistema operativo agrupe (o, incluso, divida) los sectores para formar **bloques lógicos** o unidades de asignación más grandes (o más pequeñas). Esta agrupación o división no cambia la visión de un disco como un array lineal de bloques, simplemente cambia el número y el tamaño de los bloques del array.

Un aspecto importante es el tamaño del bloque lógico. Si el bloque lógico es grande, un fichero constará de pocos bloques, pero puede aparecer una fragmentación interna muy grande en el último bloque lógico de cada fichero, lo que puede dar lugar a un gran desperdicio de espacio. En cambio, si el bloque lógico es pequeño, se producirá poca fragmentación interna, pero cada fichero ocupará muchos bloques, lo que puede dar lugar a listas ligadas muy grandes o a nodos-i con muchos bloques indirectos, dependiendo de la implementación de ficheros.

El tamaño de bloque lógico también afecta al rendimiento. Así, bloques lógicos grandes proporcionarán tasas de transferencia altas en las lecturas/escrituras de disco, mientras que bloques lógicos pequeños pueden dar lugar a tasas de transferencia más pequeñas. Como veremos en el tema de E/S, esto se debe a que el tiempo que se tarda en leer/escribir un bloque lógico de disco no es completamente proporcional a su tamaño. Por ejemplo, leer o escribir en disco un bloque de 128 KiB suele llevar solo un poco más de tiempo que leer o escribir un bloque de 64 KiB, y no el doble.

Por lo tanto, a la hora de elegir un tamaño de bloque lógico, hay que buscar un equilibrio razonable entre la eficiencia en el uso del espacio en disco y la tasa de transferencia en las operaciones de disco.

Grandes	Pequeños
buena rendimient	poca frag. interna
tasa alta	baja rend. y poca frags.
mucha fragmentación intern	taus bajas

#### Registro de bloques libres



Para llevar un **registro de qué bloques lógicos de un disco están libres y qué bloques ocupados**, se suelen utilizar dos métodos: **la lista ligada de bloques y el mapa de bits**.

<sup>8</sup>Observa que, por ejemplo, el nodo-i 500 en un sistema de ficheros es distinto del nodo-i 500 en otro sistema de ficheros, por lo que no tiene sentido crear dos ficheros con el mismo número de nodo-i en dos sistemas de ficheros distintos, ya que nunca serían el mismo fichero.

## CAPÍTULO 4. SISTEMAS DE FICHEROS

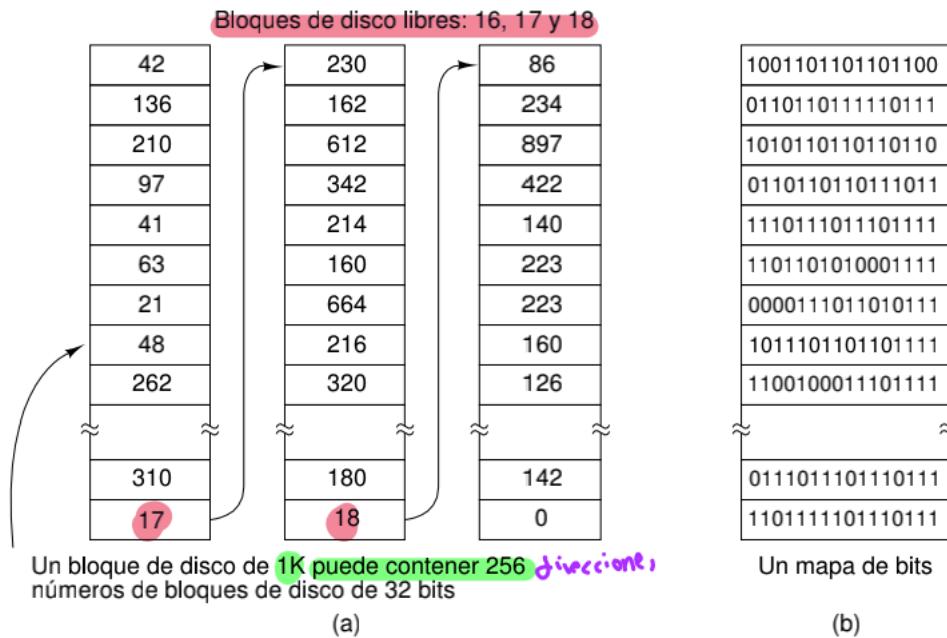


Figura 4.15. Registro de bloques libres. (a) Mediante una lista ligada de bloques. (b) Mediante un mapa de bits.

En la **lista ligada de bloques**, cada bloque de la lista contiene tantos números de bloques libres como pueda, además de un puntero al siguiente bloque. Por ejemplo, con bloques lógicos de 1 KiB y 32 bits por número de bloque, un bloque será capaz de almacenar 256 direcciones: 255 direcciones de bloques libres más la dirección del siguiente bloque de la lista. La figura 4.15(a) muestra este caso. Observa que los propios bloques de la lista son bloques libres en el sentido de que no pertenecen a ningún fichero o directorio, por lo que podrán ser usados en el momento en el que dejen de contener información útil.

En el **mapa de bits**, un disco con  $N$  bloques necesita un mapa de bits con  $N$  bits. Los bloques libres se representan con el valor 0 en el mapa y los bloques asignados con el valor 1 (o viceversa). Esta técnica se muestra en la figura 4.15(b). Evidentemente, el mapa de bits debe guardarse también en bloques de disco. A diferencia de antes, estos bloques nunca se convertirán en bloques libres (es decir, no se quedarán vacíos) pues el mapa de bits siempre tiene el mismo tamaño al depender del tamaño del disco y no del número de bloques libres.

De las dos opciones, la más utilizada en la práctica es el mapa de bits. Sistemas de ficheros como Ext2, Ext3 y Ext4 en Linux, o NTFS en Windows, usan esta técnica. Un motivo es que el mapa de bits suele ocupar bastante menos espacio que la lista ligada. Solo si el disco está casi lleno, el mapa de bits ocupará más bloques que la lista ligada. Otro motivo es que el mapa de bits permite buscar de forma sencilla grupos de bloques libres consecutivos en disco. Basta con buscar una secuencia de bits a 0 (o a 1, dependiendo de cómo indiquemos los bloques libres) lo suficientemente larga. Esto es importante de cara al rendimiento, ya que es más eficiente leer o escribir un fichero que tiene todos sus bloques consecutivos en disco que hacer lo mismo con otro cuyos bloques están dispersos. Para conseguir lo mismo con la lista ligada, los bloques tendrían que estar ordenados por dirección de disco, lo cual puede ser bastante costoso.

Como hemos dicho en la sección 4.4.1 al describir la asignación mediante lista ligada e índice, la FAT se utiliza no solo para registrar los bloques que pertenecen a cada fichero, sino también para registrar los bloques libres. En este sentido, la FAT actúa

## 4.5. CACHÉ DE DISCO

como un mapa de bits, ya que cada bloque de disco tiene una entrada en el índice que dice si el bloque está ocupado o está libre.

### 4.5 CACHÉ DE DISCO

El acceso a un disco (incluyendo los modernos discos SSD y NVMe) es bastante más lento que el acceso a memoria principal. Por ello, existen diversos mecanismos que tratan de mejorar su rendimiento. Uno de estos mecanismos nos lo proporciona la *caché de disco* (también llamada *caché de bloques* o *caché de buffers*) que trata de reducir los accesos a disco necesarios. Esta caché es implementada por el sistema operativo utilizando una parte de la memoria principal y contiene bloques que, desde un punto de vista lógico, pertenecen al disco, pero que se mantienen temporalmente en la memoria principal por razones de rendimiento.

El funcionamiento es similar al de cualquier otra caché. Cuando se lee un bloque, se comprueba si está o no ya en memoria principal. En caso afirmativo, se satisface la solicitud sin acceder al disco; si no, se lee el bloque del disco, se coloca en caché y después se procesa la solicitud de lectura. En este segundo caso la caché puede estar totalmente ocupada, por lo que habrá que eliminar algún bloque (escribiéndolo en disco si ha sido modificado) para hacer hueco.

A la hora de decidir qué bloque expulsar de una caché llena, podemos usar un algoritmo LRU con listas ligadas<sup>9</sup>, ya que las referencias a caché son mucho menos frecuentes que a memoria. Sin embargo, aplicar un LRU puro no es recomendable si queremos mantener la *consistencia del sistema de ficheros*<sup>10</sup> ante los posibles fallos del sistema.

Por ejemplo, si se borra un fichero, y el bloque del directorio donde está la entrada se encuentra en caché y se modifica para eliminar dicha entrada, puede ocurrir que, si se produce un fallo del sistema antes de que el bloque se vuelva a escribir en disco, el sistema de ficheros quede inconsistente (habría un nodo-i ocupado, pero sin ningún fichero que apuntara al mismo). Obsérvese que, si el bloque acaba de ser modificado, muy probablemente se encontrará en la cabeza de la lista LRU, por lo que podrá transcurrir mucho tiempo antes de que llegue al final y sea expulsado y escrito en disco. El problema se agrava si el bloque se utiliza con mucha frecuencia, ya que casi siempre se encontrará en la cabeza de la lista. Cuanto más tiempo pase sin que un bloque modificado se escriba en disco, mayor será la probabilidad de que se produzcan inconsistencias por caídas del sistema.

Además, hay bloques, como los bloques doblemente indirectos, que rara vez tienen dos referencias en un intervalo corto de tiempo, por lo que puede interesar colocarlos en la cola de la lista para que sean los primeros en salir de caché.

Las dos consideraciones anteriores dan lugar a un *LRU modificado* que tiene en cuenta dos factores independientes:

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

<sup>9</sup>Recordemos que el algoritmo LRU selecciona el bloque que hace más tiempo que no se usa. Una posible implementación de este algoritmo es mediante una lista ligada donde, cada vez que se referencia un elemento de la lista, se mueve a la cabeza de la misma. Por tanto, en la cola siempre estarán los elementos que hace más tiempo que no se usan. En el siguiente tema hablaremos más de este algoritmo y de sus posibles problemas de implementación.

<sup>10</sup>Un sistema de ficheros se considera consistente cuando sus estructuras de datos no contienen información contradictoria. Por ejemplo, en un sistema de ficheros consistente, un fichero nunca usará un nodo-i (o un bloque) que el sistema de ficheros tenga registrado como libre.

coger luto de  
Mem Ram → y lo  
usa como cache.

asociativa, algoritmo  
Complejo.



dices un bloque  
y escubes un  
bloque que ahi que  
se lo quites cambie  
un bit.

## CAPÍTULO 4. SISTEMAS DE FICHEROS

Según la primera pregunta, los bloques recién usados que probablemente no se vuelvan a utilizar pronto (como los bloques indirectos) pasarán directamente al final de la lista LRU (y no al frente) para que sus *buffers* se reutilicen con rapidez.

Según la segunda pregunta, si un bloque es esencial para la consistencia del sistema de ficheros (como un bloque de nodos-i o un bloque del mapa de bits de bloques libres) y ha sido modificado, debe escribirse en disco lo antes posible, sin importar en qué lugar de la lista LRU se encuentre (esto no significa que se lleve al final de la lista, simplemente, que se escribe en disco sin esperar a que sea expulsado).

Tampoco es recomendable tener mucho tiempo los bloques de datos (que no son importantes para la consistencia del sistema de ficheros, pero sí para los usuarios) en caché, por el riesgo de perder información.

En Unix, cualquier *bloque de datos* modificado se escribe en disco a los 30 segundos o antes. En cambio, los *bloques de metadatos* (aquellos que no contienen datos de los usuarios y que, generalmente, son importantes para la consistencia del sistema de ficheros, como bloques de nodos-i, bloques de directorio, mapas de bits, etc.) se escriben en disco a los 5 segundos o antes, aunque el intervalo exacto puede depender del sistema de ficheros usado.

Los usuarios de Unix también disponen de la orden `sync`. Al ejecutarse, esta orden fuerza la escritura en disco de todos los bloques, tanto de datos como de metadatos, que se encuentren modificados en memoria principal.

### 4.6 DISCOS Y SISTEMAS DE FICHEROS

Aunque hasta ahora hemos supuesto que un sistema de ficheros ocupa todo un disco, en realidad, no tiene por qué ser así y, de hecho, no suele ser así. En su lugar, lo habitual es que un disco contenga varios sistemas de ficheros. Veamos esto y también, para finalizar, una posible estructura de sistema de ficheros.

#### 4.6.1 Particiones

Para facilitar el uso de los discos, los sistemas operativos permiten crear *particiones*. Una partición es una porción de bloques consecutivos de un disco. Las particiones son manejadas por el sistema operativo (es decir, no existen para los propios discos), que las representa también como un array lineal de bloques. Este array tendrá un bloque 0 y un bloque  $N - 1$ , donde ahora  $N$  dependerá del tamaño de la partición en bloques lógicos. Las particiones, por tanto, se ven como pequeños discos dentro de un disco mayor.

El número y tamaño de las particiones existentes en un disco, junto con los bloques de inicio y fin de cada una, se ha guardado tradicionalmente en la *tabla de particiones*, la cual, a su vez, se almacena en el bloque 0 del disco. Este bloque se conoce como MBR (*Master Boot Record*, es decir, *registro de arranque maestro*). La figura 4.16 muestra un disco dividido en varias particiones y su MBR<sup>11</sup>.

Junto con la tabla de particiones, el MBR también suele contener una pequeña porción de código de arranque que la BIOS carga en memoria y ejecuta para, tras una serie de pasos, terminar arrancando un sistema operativo instalado en el computador.

Existen varias razones para usar particiones. Una es poder tener en cada una un sistema de ficheros distinto e incluso un sistema operativo distinto. Por ejemplo, en Linux, podríamos *formatear* (ver apartado 4.6.2) la partición del sistema de ficheros principal

<sup>11</sup>En el esquema de particiones GPT, que cada vez se usa con más frecuencia y que veremos en prácticas, el funcionamiento de las particiones es idéntico al descrito aquí, si bien la tabla de particiones se almacena en sus propios bloques de disco y no en el sector 0.

Disco pequeño dentro de grande



Respaldo GPT Power point  
28/30.

## 4.6. DISCOS Y SISTEMAS DE FICHEROS

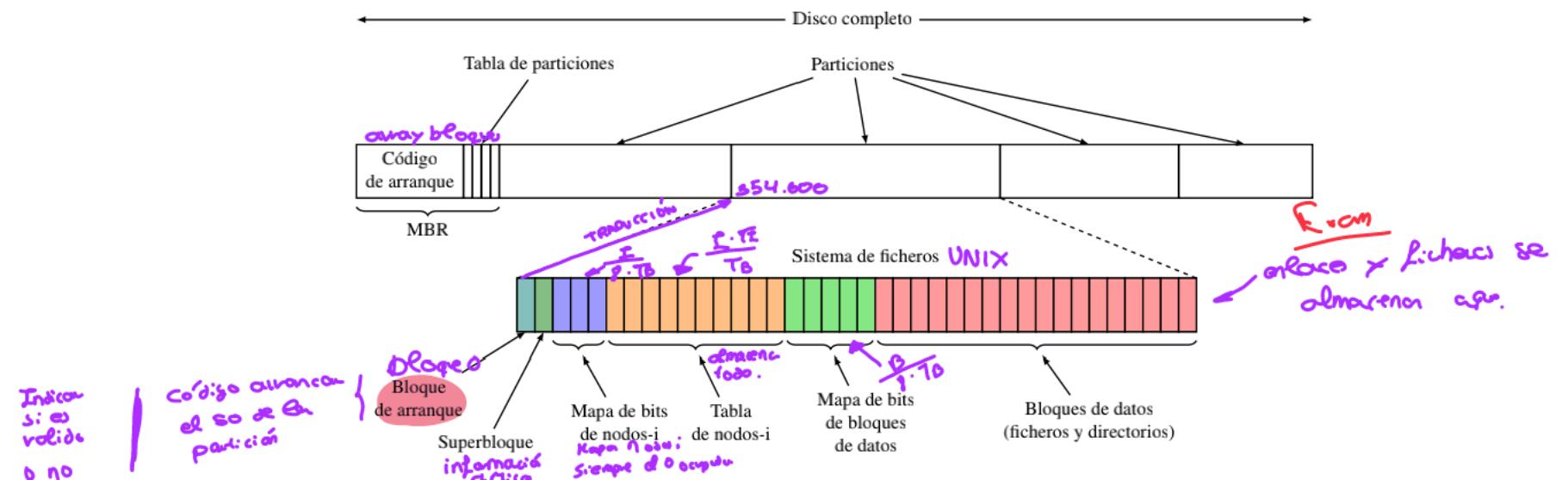


Figura 4.16. División de un disco en particiones y estructura de un posible sistema de ficheros en una de las particiones.

del sistema operativo (o *sistema de ficheros raíz*) con Ext4 y tener también otra *partición de intercambio* con un formato especial para guardar aquellas páginas que no caben en memoria principal (como veremos en el siguiente tema). Otra razón para usar particiones es poder dar a una un uso distinto, aunque el tipo de sistema de ficheros sea el mismo. Por ejemplo, en Linux podríamos tener una partición formateada con Ext4 para el sistema de ficheros raíz y otra formateada también con Ext4 para almacenar los datos de los usuarios. De esta forma, si hiciera falta reinstalar el sistema operativo, bastaría con borrar la información de la partición del sistema de ficheros raíz y dejar intacta la partición con los datos de los usuarios.

### 4.6.2 Estructura de un sistema de ficheros

En las secciones anteriores ya hemos visto cómo un sistema de ficheros puede implementar sus ficheros y directorios. También hemos visto cómo puede llevar un registro del espacio libre y ocupado. Además de todo esto, el sistema de ficheros tendrá que llevar un control de más elementos. Por ejemplo, si usa nodos-i, tendrá que saber cuáles están libres y cuáles ocupados. También tendrá que conocer en todo momento cuántos bloques hay en total libres y ocupados, y lo mismo para los nodos-i. Toda esta información se tiene que guardar en disco de alguna forma ordenada, construyendo así el sistema de ficheros propiamente dicho.

La figura 4.16 muestra, además de la división de un disco en particiones, la posible estructura de un sistema de ficheros en una de las particiones. Como podemos ver, el sistema de ficheros usa *zonas* o grupos de bloques consecutivos de disco para almacenar distintas estructuras de datos: superbloque, mapas de bits, etc. Estas estructuras de datos, su organización en disco y su contenido inicial se crean y escriben en disco cuando se *formatea* la partición que va a contener el sistema de ficheros. Veamos a continuación en detalle cada uno de estos elementos.

El *bloque de arranque* ocupa el bloque 0 de la partición. Este bloque generalmente no se usa, pero puede contener código de arranque del sistema operativo que se encuentra en la partición. En este caso, el código de arranque en el MBR identifica la partición como activa y carga su bloque de arranque para continuar con la carga del sistema operativo. Ya que generalmente este bloque no es usado por el sistema de ficheros, se

## CAPÍTULO 4. SISTEMAS DE FICHEROS

puede usar su dirección (0) como valor nulo o no válido.

El bloque 1 es el *superbloque*. Contiene información crítica relativa a la organización del sistema de ficheros: número total de bloques lógicos, número total de nodos-i, tamaño de los mapas de bits de bloques y nodos-i, etc. La destrucción del superbloque provocaría que el sistema de ficheros quedara ilegible. Por eso algunos sistemas de ficheros tienen varias copias del mismo en distintas zonas del disco. El superbloque suele existir en sistemas de ficheros de Unix. En otros sistemas de ficheros, en cambio, el superbloque forma parte del bloque de arranque, es decir, el bloque de arranque contiene tanto código para cargar el sistema operativo de la partición como información relativa a la estructura del sistema de ficheros.

Después del superbloque viene el *mapa de bits de bloques*. En este mapa, cada bit indica si el bloque correspondiente está libre u ocupado. Su tamaño, en bloques, depende del tamaño de la zona dedicada a ficheros y directorios<sup>12</sup>. Así, si esta zona tiene  $B$  bloques en total, de tamaño  $T_B$  bytes cada uno, entonces el mapa de bits ocupa  $\lceil \frac{B}{8 \cdot T_B} \rceil$  bloques.

A continuación tenemos el *mapa de bits de nodos-i*. Su función es similar a la del mapa de bits de bloques, aunque en este caso se utiliza para saber qué nodos-i hay libres y cuáles están ocupados. Su tamaño, en bloques, depende del total de nodos-i. En este caso, si tenemos  $I$  nodos-i y un tamaño de bloque de  $T_B$  bytes, entonces este mapa necesita  $\lceil \frac{I}{8 \cdot T_B} \rceil$  bloques.

Lo siguiente es la *tabla de nodos-i*, que contiene los nodos-i usados para ficheros. Para saber si un nodo-i está libre u ocupado, tenemos que usar el bit correspondiente del mapa de bits de nodos-i, como ya hemos visto. El tamaño de la tabla de nodos-i, en bloques, depende del tamaño de nodo-i y del total de nodos-i en el sistema. En concreto, si tenemos  $I$  nodos-i de tamaño  $T_I$  bytes cada uno, y bloques de tamaño  $T_B$  bytes, esta tabla ocupa  $\lceil \frac{I \cdot T_I}{T_B} \rceil$  bloques.

Finalmente, el resto de bloques se usa para almacenar en ellos los bloques de datos de los ficheros regulares, los bloques de los directorios y los bloques indirectos (para aquellos ficheros que los necesitan).

Una vez que hemos visto las distintas zonas que hay dentro de un sistema de ficheros y sus posiciones en una partición, vamos a terminar este tema viendo en detalle qué pasos se tienen que dar para crear un fichero regular y qué se hace con esas zonas.

La figura 4.17 muestra el proceso de creación del fichero regular `mbox` dentro del directorio `/usr/ast`. Supondremos que los bloques lógicos son de 4 KiB y que el fichero va a alcanzar un tamaño de 45 KiB, por lo que ocupará 12 bloques de datos (11 de ellos completamente y el último solo parcialmente). Vamos a suponer también que las direcciones de disco son de 4 bytes, que los nodos-i tiene un tamaño de 64 bytes y que en cada nodo-i caben 13 direcciones de disco: 10 de bloques de datos, un BSI, un BDI y un BTI. Teniendo en cuenta todo esto, y observando que en la figura se parte del bloque de datos del directorio `/usr/ast`, que se encuentra almacenado en la dirección de disco 406, los pasos que se dan para crear el fichero, y que aparecen indicados en dicha figura, son los siguientes:

1. Se comprueba si en el bloque de datos del directorio `/usr/ast` existe ya otro fichero con el nombre `mbox`. Si es así, el proceso terminará devolviendo un código de error adecuado.

<sup>12</sup>En algunos sistemas de ficheros, el mapa de bits de bloques representa a todos los bloques de la partición, por lo que su tamaño depende del tamaño de esta. En este caso, los bloques destinados al bloque de arranque, superbloque, los mapas de bits de bloques y nodos-i, y la tabla de nodos-i directamente se marcan como ocupados en el mapa de bits de bloques.

## 4.6. DISCOS Y SISTEMAS DE FICHEROS

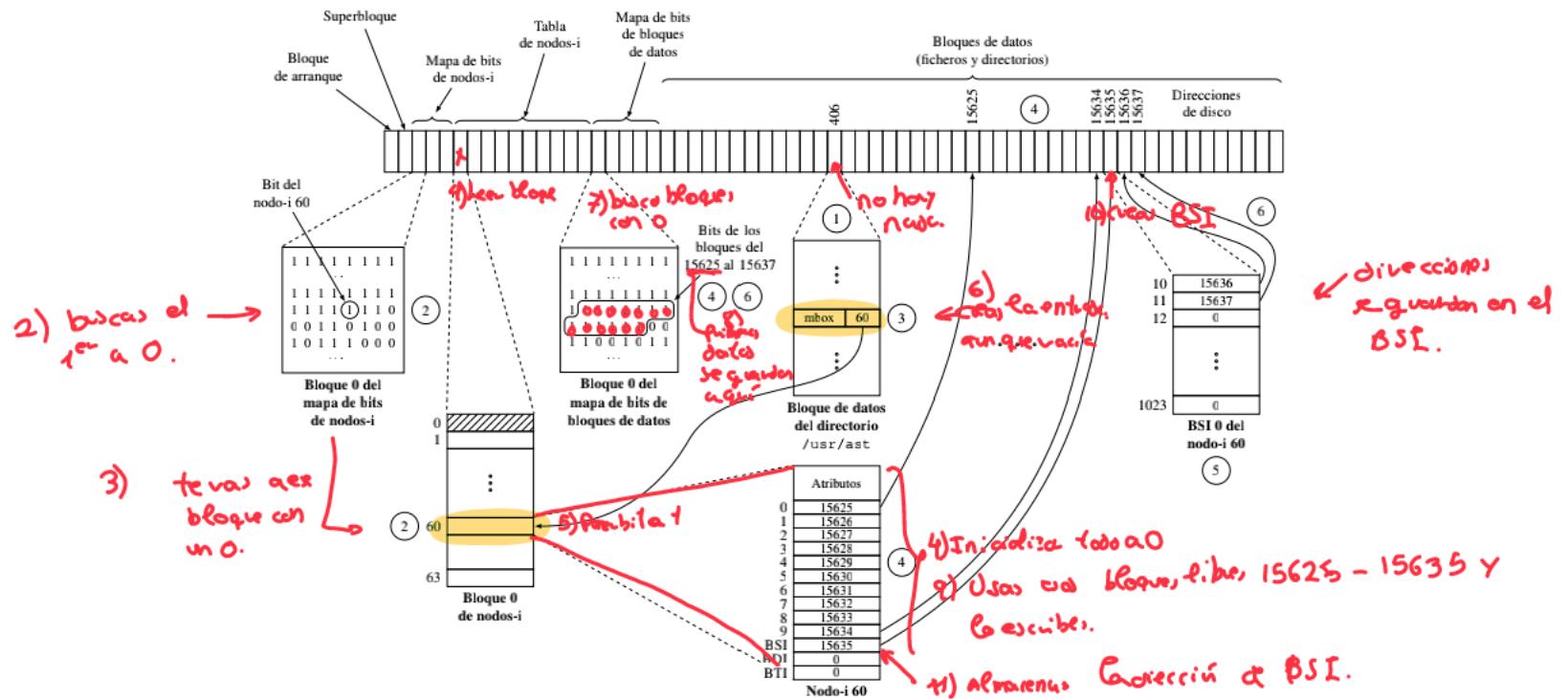


Figura 4.17. Pasos para crear el fichero regular `mbox` en el directorio `/usr/ast` y bloques del sistema de ficheros involucrados.

2. Una vez que hemos comprobado que **no hay otro fichero con el mismo nombre**, buscamos en el **mapa de bits de nodos-i** un bit a 0. Supongamos que encontramos que el **bit 60** está a 0. Eso significa que el **nodo-i 60**, que se encuentra en el **bloque 0** de la **tabla de nodos-i** (ya que en cada bloque caben  $\frac{2^{12} \text{ bytes/bloque}}{2^6 \text{ bytes/nodo-i}} = 2^6 = 64$  nodos-i/bloque) está libre. Ponemos el bit 60 a 1 (como podemos ver en la figura) indicando así que el **nodo-i 60** ahora está ocupado.
3. Añadimos una nueva entrada al directorio `/usr/ast` modificando su primer bloque de datos que, como ya hemos dicho, se encuentra en la dirección de disco 406. Básicamente, la nueva entrada contiene el nombre del fichero, `mbox`, y el **nodo-i** asignado, 60.
4. A continuación, según se va escribiendo en el fichero, le vamos asignando bloques de datos libres y guardando las direcciones de disco de los mismos en el **nodo-i**. El primer bloque de datos asignado es el 15625, ya que estamos suponiendo que el bit de dicha posición en el **mapa de bits de bloques de datos** estaba a 0. Dicha dirección (15625) se guarda en el **nodo-i**, poniendo el bit correspondiente a 1 para indicar así que ahora está ocupado. Cuando este primer bloque del fichero se llene, se le asignará al fichero otro bloque de datos libre que, en este caso, supondremos que es el 15626, guardando también la dirección en el **nodo-i**. Y así sucesivamente. Observa que, en la figura 4.17, los bits de los bloques de datos ocupados ya se muestran a 1 (originalmente, estaban a cero).
5. Cuando el fichero alcanza un tamaño de 40 KiB o, lo que es lo mismo, ocupa ya 10 bloques, no caben más direcciones de bloques de datos en el **nodo-i**, por lo que hay que asignarle un primer bloque simplemente indirecto. Según se ve en la figura, el bloque asignado a este bloque indirecto es el 15635 de disco, guardando

## CAPÍTULO 4. SISTEMAS DE FICHEROS

esta dirección en el nodo-i.

6. Finalmente, una vez añadido ese primer BSI, se continúa escribiendo en el fichero hasta ocupar dos bloques de datos más, cuyas direcciones, 15636 y 15637, se guardan en el BSI.

Como podemos apreciar, tanto en el nodo-i como en el bloque simplemente indirecto, las posiciones no ocupadas por direcciones de disco contienen un 0, que no es una dirección de disco válida, indicando así que dichas posiciones están vacías.

Observa también que los bloques indirectos (como el BSI que hemos añadido al nodo-i 60) se almacenan en la zona de bloques de datos del sistema de ficheros, a pesar de no contener datos del fichero. Recuerda que estos bloques contienen direcciones de otros bloques en la misma zona: pueden ser bloques indirectos o bloques de datos, dependiendo del tipo concreto de bloque. Por ejemplo, un bloque simplemente indirecto contiene direcciones de bloques de datos, mientras que un doblemente o triplemente indirecto contiene direcciones de otros bloques simplemente o doblemente indirectos, respectivamente.