Sistemas Operativos

Universidad Complutense de Madrid

2020-2021

Módulo 2: Sistemas de Ficheros

Juan Carlos Sáez

Contenido

Contenido

¿Qué es un fichero?

Definición

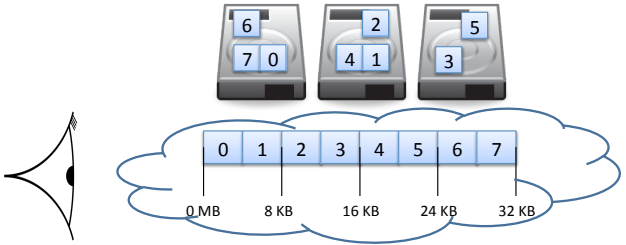
- Un fichero es una unidad de almacenamiento lógico no volátil que agrupa un conjunto de información relacionada entre sí bajo un mismo nombre
- Los ficheros suelen servir...
  - para almacenar código máquina o código fuente
  - como fuente de entrada a los programas
  - para guardar a largo plazo las salidas de los programas

El sistema de Gestión de Ficheros se encarga de:

1. Ofrecer servicios de manipulación de ficheros a los programas
2. Gestionar los permisos de acceso
3. Garantizar la integridad de atributos y contenidos
4. Soportar los ficheros sobre medios de almacenamiento

Visión lógica vs. visión física

- Visión lógica:
  - Ficheros
  - Directorios
  - Sistemas de ficheros y particiones
- Visión física:
  - Bloques o bytes ubicados en dispositivos

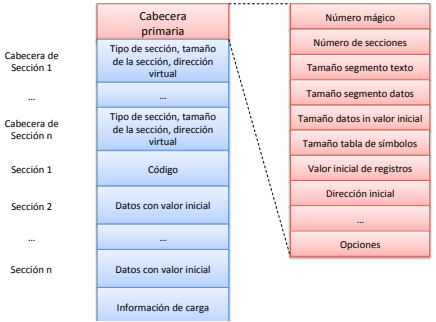


Características para el usuario

- Abstraen los dispositivos de almacenamiento físico
- Almacenamiento permanente de información. Los ficheros no desaparecen aunque se apague el computador
  - Conjunto de información estructurada de forma lógica según criterios de aplicación
- Nombres lógicos y estructurados
- No están ligados al ciclo de vida de una aplicación particular
- Se accede a ellos a través de llamadas al sistema operativo o de bibliotecas de utilidades

Contenido de un fichero para el SO

- Desde el punto de vista del SO, la mayor parte de los ficheros son una colección de bytes desperdigada por uno o varios dispositivos de almacenamiento
- El SO reconoce solamente unos pocos formatos de fichero:
  - Ficheros ejecutables
  - Ficheros de texto (p.ej., scripts)



Ficheros: visión del usuario

Clasificaciones de ficheros

- Por codificación de los datos almacenados:
  - Texto o Caracteres
  - Binarios
- Programas:
  - Código fuente
  - Ficheros objeto (imagen de carga)
- Documentos (específico de aplicación)

Tipos de ficheros en UNIX

- Los SSOO tipo UNIX (Linux, Mac OS X, Solaris, ...) modelan como ficheros distintas abstracciones del SO
  - No todos los ficheros que pueden manipular los programas de usuario son ficheros "normales"

Tipos de ficheros en UNIX

- Regulares (normales)
- Directorios
- Fichero especial de caracteres
- Fichero especial de bloques
- Enlaces simbólicos
- Tuberías con nombre (FIFOs)
- ...

Atributos del fichero

- **Fichero:** Datos + Atributos (metadatos)

Atributos comunes

- **Nombre**: la única información en formato legible por una persona
- **Identificadores**: ID del descriptor interno del fichero, ID propietario y grupo del fichero
- **Tipo de fichero (para el SO)**: necesario en sistemas que proporcionan distintos formatos de ficheros: normales y especiales
  - Para más información `man 2 stat`
- **Tamaño del fichero**: número de bytes en el fichero, máximo tamaño posible, etc.
- **Protección**: control de acceso (`rwx` para usuario, grupo y todos)
- **Información temporal**: de creación, de acceso, de modificación, etc.
- ...

Descriptores de un fichero

- Cada objeto/entidad gestionada por el SO tiene un nombre o descriptor único
- Cada fichero posee dos descriptores únicos:
  1. **Descriptor lógico**: nombre del fichero
  2. **Descriptor físico**: estructura de datos interna que el SO manipula para realizar operaciones sobre el fichero
     - Se almacena en disco (*persistente*)
     - Se carga en memoria cuando el SO accede al fichero en nombre de algún programa de usuario
- Los directorios permiten al SO asociar el descriptor lógico de un fichero con su descriptor físico

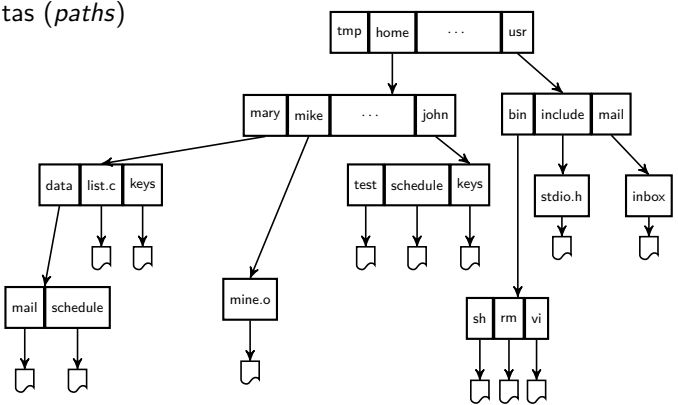
Más sobre descriptores lógicos (nombres)

Los convenios de nombrado son específicos de cada SO

- **Longitud máxima**: 8 en MS-DOS, 4096 en UNIX
- **Sensibles a tipografía**: `MiFichero` y `mifichero` son el mismo fichero en Windows pero distintos en LINUX
- **Extensiones**
  - Obligatoria o no
  - Más de una o no
  - Fija para cada tipo de documentos
  - En general, las extensiones sólo son significativas para las aplicaciones (pdf, doc, html, c, cpp, ...)

Directorios y Jerarquías de directorios

- Objetivo directorio: asociación *nombre* $\leftrightarrow$ *descriptor físico*
- Directorios suelen ser parte de una jerarquía
  - *Directorio raíz* (GNU Linux y UNIX: "/" ; Windows: "\")
  - Uso de rutas (*paths*)



Nombres jerárquicos (I)

Directorios especiales

- **Directorio de trabajo '.'**
  - Ejemplo: `cp /users/mike/keys .`
  - En BASH se cambia con `cd`, y la ruta se obtiene con `pwd`
- **Directorio padre '..'**
  - Ejemplo: `ls ..`
- **Directorio HOME**: El directorio *base* del usuario
  - `cd`
  - `cd $HOME`

SO	Ubicación por defecto del HOME
Windows 7 y posteriores	`C:\Users\<username>`
GNU/Linux	`/home/<username>`
Solaris	`/export/home/<username>`
Mac OS X	`/Users/<username>`

Nombres jerárquicos (II)

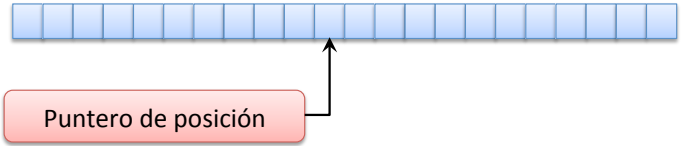
Dos tipos de ruta (*path*)

1. **Ruta absoluta**: especificación del nombre con respecto al directorio raíz ("/" en LINUX, "\" en Windows)
   - Ejemplo: `/users/mike/keys`
2. **Ruta relativa**: especificación del nombre con respecto al directorio de trabajo del proceso
   - Relativos al dir. de trabajo o actual: aquel en el se está al indicar el nombre relativo.
   - Ejemplo: (Dir. actual es `/users`)
     - `mike/keys` ‘
     - `../users/mike/keys`
     - `./mike/keys`

Contenido

Visión lógica

- Un espacio lógico de direcciones contiguas usado para almacenar datos + puntero de posición



- Abrir/crear fichero para acceder a su representación lógica
  - Descriptor del fichero (abstracción programador)
    - Objeto, manejador (`FILE*`) o número
- Operaciones (en base a modo de apertura)
  - lectura, escritura
  - movimiento explícito puntero posición
  - cierre

Servicios POSIX para ficheros (I)

- Descriptores de ficheros: enteros de 0 a 64K
  - Valor entero que maneja el programador para realizar operaciones sobre el fichero abierto
  - Puede referirse a un fichero regular o a fichero especial
  - Advertencia: Descriptor de un fichero abierto (abstracción del API) $\neq$ descriptor lógico (nombre del fichero) $\neq$ descriptor físico o interno (p. ej., nodo-i)
- Descriptores númericos predefinidos:
  - 0 $\rightarrow$ entrada estándar
  - 1 $\rightarrow$ salida estándar
  - 2 $\rightarrow$ salida de error

Servicios POSIX para ficheros (II)

- Se proporcionan servicios para consultar y modificar el valor de los atributos de un fichero
  - ID usuario y grupo propietario, tamaño del fichero, fechas de modificación o último acceso, protección, etc.
- Cada fichero tiene información de protección asociada

```
propietario grupo   todos
rwx         rwx     rwx
```

  - Los permisos de lectura (r), escritura (w) y ejecución (x) para cada *agente* del sistema se codifican con 3 bits
  - El API de ficheros representa esta información mediante 3 dígitos octales
    - Ejemplo: 755 significa `rwxr-xr-x`

Operaciones básicas sobre ficheros

- **creat**: crea un fichero con un nombre y protección y devuelve un descriptor
- **open**: abre un fichero con nombre para realizar operaciones sobre el y devuelve un descriptor
- **read**: lee datos de un fichero abierto, usando su descriptor, copiándolos a una región de memoria del proceso
- **write**: escribe datos a un fichero abierto, usando su descriptor, desde una región de memoria del proceso
- **lseek**: mueve el puntero de posición del fichero (movimiento relativo a un punto dado)
- **close**: cierra un fichero abierto
- **unlink**: borra el fichero con un nombre
- **stat**: devuelve los atributos de un fichero

Crear un fichero

creat()

```
int creat(char *name, mode_t mode);
```

- Argumentos:
  - `name`: Nombre de fichero
  - `mode`: Bits de permiso para el fichero
- Valor de retorno:
  - Devuelve un descriptor de fichero ó -1 si error.
- Ejemplos:

```
fd=creat("data.txt", 0751);
fd=open("data.txt", O_WRONLY|O_CREAT|O_TRUNC,0751);
```

Abrir un fichero

open()

```
int open(char *name, int flag, ...);
```

- Argumentos:
  - `name`: ruta del fichero
  - `flags`: opciones de apertura
    - `O_RDONLY` Sólo lectura
    - `O_WRONLY` Sólo escritura
    - `O_RDWR` Lectura y escritura
    - `O_APPEND` El puntero de posición se desplaza al final del fichero
    - `O_CREAT` Si el fichero existe no tiene efecto. Si no existe lo crea
    - `O_TRUNC` Trunca el fichero si se abre para escritura
- Valor de retorno:
  - Un descriptor de fichero ó -1 si hay error

Leer de un fichero

read()

```
ssize_t read(int fd, void *buf, size_t n_bytes);
```

- Argumentos:
  - `fd`: descriptor de fichero
  - `buf`: dirección de memoria de la región donde se copiarán los datos
  - `n_bytes`: número de bytes a leer
- Valor de retorno:
  - Número de bytes realmente leídos ó -1 si error
- Descripción:
  - Transfiere `n_bytes` (como mucho) al espacio de direcciones del proceso
    - Podrían transferirse menos de `n_bytes` (p.ej., se llegó a fin del fichero)
  - `read()` avanza de forma implícita el puntero de posición del fichero tantas posiciones cómo bytes se hayan transferido

Escribir en un fichero

write()

```
ssize_t write(int fd, void *buf, size_t n_bytes);
```

- Argumentos:
  - `fd`: descriptor de fichero
  - `buf`: zona de datos a escribir
  - `n_bytes`: número de bytes a escribir
- Valor de retorno:
  - Número de bytes realmente escritos ó -1 si error
- Descripción:
  - Transfiere `n_bytes` (como mucho) al fichero
    - Podrían transferirse menos de `n_bytes` (p.ej., se rebasa el tamaño máximo de un fichero o se interrumpe por una señal)
  - `write()` avanza de forma implícita el puntero de posición del fichero tantas posiciones cómo bytes se hayan transferido

Cerrar un fichero

close()

```
int close(int fd);
```

- Argumentos:
  - `fd`: descriptor de fichero
- Valor de retorno:
  - Cero ó -1 si error
- Descripción:
  - El proceso pierde la asociación a un fichero

Ejemplo: copia de un fichero (1/2)

Modo de uso

```
copy_file <input-file> <output-file>
```

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#define BUFSIZE     512

void main(int argc, char **argv) {
    int fd_in, fd_out, n_read;
    char buffer[BUFSIZE];

    fd_in = open(argv[1], O_RDONLY); /* open the input file */
    if (fd_in < 0){
        perror("open");    exit(1);
    }

    fd_out = creat(argv[2], 0644);      /* create the output file */
    if (fd_out < 0){
        close(fd_in);
        perror("creat");    exit(1);
     }
```

Ejemplo: copia de un fichero (2/2)

```
/* main loop to transfer data between files */
    while ((n_read = read(fd_in, buffer, BUFSIZE)) > 0)  {
        /* Transfer data from the buffer onto the output file */
        if (write(fd_out, buffer, n_read) < n_read)  {
            perror("write");
            close(fd_in); close(fd_out);
            exit(1);
        }
    }
    if (n_read < 0)  {
        perror("read");
        close(fd_in); close(fd_out);
        exit(1);
    }
    close(fd_in);  close(fd_out);
    exit(0);
}
```

Eliminar una entrada de directorio

unlink()

```
int unlink(const char* path);
```

- Argumentos:
  - `path`: nombre del archivo
- Valor de retorno:
  - Devuelve 0 ó -1 si error.
- Descripción:
  - Elimina la entrada de directorio y decrementa el número de enlaces del archivo correspondiente.
  - Cuando el número de enlaces es igual a cero y ningún proceso lo mantiene abierto, se libera el espacio ocupado por el fichero y el fichero deja de ser accesible

Modificar el puntero de posición

lseek()

```
off_t lseek(int fd, off_t offset, int whence);
```

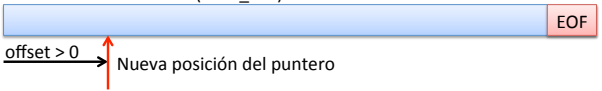
- Argumentos:
  - `fd`: Descriptor de fichero
  - `offset`: desplazamiento relativo or absoluto
  - `whence`: base del desplazamiento
- Valor de retorno:
  - La nueva posición del puntero ó -1 si error
- Descripción:
  - Establece una nueva ubicación del puntero de posición del fichero:
    - `whence == SEEK_SET` → posición = offset
    - `whence == SEEK_CUR` → posición = posición actual + offset
    - `whence == SEEK_END` → posición = tamaño fichero + offset

Modificar el puntero de posición

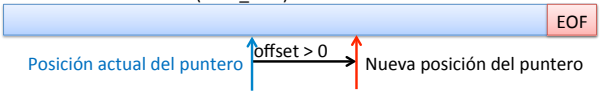
```
off_t lseek(int fd,
            off_t offset,
            int whence)
```

En un sistema de ficheros tipo Unix, es sencillo realizar un lseek debido a la estructura lineal de los ficheros. De hecho, esta operación se realiza habitualmente con una única búsqueda en disco.

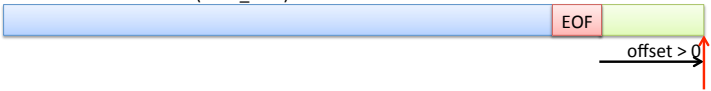
- Caso 1: whence == 0 (SEEK_SET)



- Caso 2: whence == 1 (SEEK_CUR)



- Caso 3: whence == 2 (SEEK_END)



Consultar atributos de un fichero

stat() y fstat()

```
int stat(char *name, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

- Argumentos:
  - `name`: nombre del fichero
  - `fd`: descriptor de fichero
  - `buf`: puntero a un objeto de tipo `struct stat` donde se almacenará el valor de los atributos del fichero
    - "`man 2 stat`" para consultar campos de `struct stat`
- Valor de retorno:
  - Cero ó -1 si error

Contenido

Servicios POSIX para directorios

- **Directorio:** tabla de entradas
  - Implementación de cada entrada es específica del sistema de ficheros
  - No es posible el acceso/alteración de la información con `read()` y `write()`
- *Gestión manual complicada:*
  - Nombres de fichero tienen longitud variable
  - En UNIX, varios sistemas de ficheros "conviven" en un único árbol/grafo de directorios
    - Distintas representaciones de directorios en un mismo sistema
- Servicios POSIX: operaciones para crear, modificar y recorrer directorios
  - Directorios se exponen como una tabla de entradas genéricas, cada una con un nombre

Operaciones básicas sobre directorios

- **mkdir**: crea un directorio con un nombre y protección
- **rmdir**: borra el directorio vacío con un nombre
- **opendir**: abre un directorio como una secuencia de entradas y se sitúa en la primera
- **readdir**: lee la siguiente entrada del directorio
- **rewinddir**: sitúa el puntero de posición en la primera entrada
- **closedir**: cierra un directorio abierto con un descriptor
- **link/symlink**: crea una nueva entrada en el directorio para un enlace físico o simbólico
- **unlink**: elimina una entrada del directorio
- **rename**: cambiar el nombre de una entrada del directorio
- **chdir**: cambia el directorio de trabajo del proceso
- **getcwd**: obtener el directorio de trabajo

Crear un directorio

mkdir()

```
int mkdir(const char *name, mode_t mode);
```

- Argumentos:
  - `name`: nombre del directorio
  - `mode`: bits de protección
- Valor de retorno:
  - Cero ó -1 si error
- Descripción:
  - Crea un directorio de nombre name
  - `UID_propietario = UID_efectivo`
  - `GID_propietario = GID_efectivo`

Borrar un directorio

rmdir()

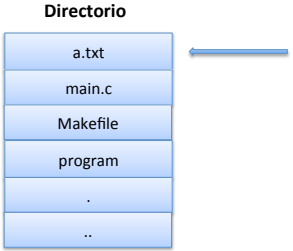
```
int rmdir(const char *name);
```

- Argumentos:
  - `name`: nombre del directorio
- Valor de retorno:
  - Cero ó -1 si error
- Descripción:
  - Borra el directorio si está vacío
  - Si el directorio no está vacío, `rmdir()` devuelve -1 (error)

Servicios POSIX para directorios (II)

- El directorio se expone al usuario como una tabla (array de struct dirent) con un puntero de posición asociado

```
struct dirent {
        /* file name */
        char *d_name;
        ...
}
```



- Al abrir el directorio (`opendir()`) el puntero apunta a la primera entrada
- `readdir()` devuelve entrada actual y avanza puntero
  - Al llegar a la última entrada, `readdir()` devuelve NULL
- `rewinddir()` permite reubicar el puntero en la primera entrada

Abrir un directorio

opendir()

```
DIR *opendir(char *dirname);
```

- Argumentos:
  - `dirname`: puntero al nombre del directorio
- Valor de retorno:
  - Un puntero para utilizarse en readdir(), rewindir() o closedir(). NULL si hubo error.
- Descripción:
  - Abre un directorio como una secuencia de entradas. Se coloca en el primer elemento.

Leer entradas de directorio

readdir()

```
struct dirent *readdir(DIR *dirp);
```

- Argumentos:
  - `dirp`: puntero retornado por `opendir()`
- Valor de retorno:
  - Un puntero a un objeto del tipo `struct dirent` que representa una entrada de directorio o NULL si hubo error
- Descripción:
  - Devuelve la siguiente entrada del directorio asociado a `dirp` y avanza el puntero a la siguiente entrada
  - La representación de la entrada de directorio (`struct dirent`) es dependiente de la implementación. Debería asumirse que tan solo se obtiene un miembro: `char *d_name`.

Resetear el puntero de posición

rewinddir()

```
void rewinddir(DIR *dirp);
```

- Argumentos:
  - `dirp`: puntero devuelto por `opendir()`
- Descripción:
  - Sitúa el puntero de posición dentro del directorio en la primera entrada.

Cerrar un directorio

closedir()

```
int closedir(DIR *dirp);
```

- Argumentos:
  - `dirp`: puntero devuelto por `opendir()`
- Valor de retorno:
  - Cero ó -1 si error
- Descripción:
  - Destruye la asociación entre `dirp` y la secuencia de entradas de directorio

Programa que lista un directorio (1/2)

```c
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>

#define MAX_BUF   256

void main(int argc, char **argv){
   DIR *dirp;
   struct dirent *dp;
   char buf[MAX_BUF];

   /* Prints the path of the current working directory */
   getcwd(buf, MAX_BUF);

   printf("Current working directory: %s\n", buf);

   ...
```

Modo de uso

```
list_dir <directory>
```

Programa que lista un directorio (2/2)

```
    /* Open the directory passed as the first argument */
    dirp = opendir(argv[1]);

    if (dirp == NULL)  {
        fprintf(stderr,"Couldn't open %s\n", argv[1]);
    } else {
        /* Iterate the set of directory entries */
        while ( (dp = readdir(dirp)) != NULL)
            printf("%s\n", dp->d_name);
      closedir(dirp);
   }
   exit(0);
}
```

Renombrar una entrada de directorio

rename()

```
int rename(char *old, char *new);
```

- Argumentos:
  - `old`: nombre de un fichero existente
  - `new`: nuevo nombre del fichero
- Valor de retorno:
  - Cero ó -1 si error
- Descripción:
  - Cambia el nombre del fichero `old`. El nuevo nombre es `new`

Cambiar el directorio de trabajo

chdir()

```
int chdir(char *name);
```

- Argumentos:
  - `name`: nombre de un directorio
- Valor de retorno:
  - Cero ó -1 si error
- Descripción:
  - Modifica el directorio de trabajo actual, aquel a partir del cual se forman los nombre relativos

Obtener ruta del directorio de trabajo

getcwd()

```
char *getcwd(char *buf, size_t size);
```

- Argumentos:
  - `buf`: puntero al espacio donde almacenar el nombre del directorio actual
  - `size`: longitud en bytes de dicho espacio
- Valor de retorno:
  - Puntero a `buf` o `NULL` si error
- Descripción:
  - Obtiene el nombre del directorio actual

Contenido

Sistemas de ficheros

El acceso a bajo nivel a los dispositivos es ...

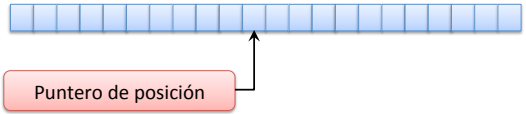
- Incómodo
  - Detalles físicos de los dispositivos
  - Dependiente de las direcciones físicas
- No seguro
  - Si el usuario accede al nivel físico no tiene restricciones

El sistema de gestión de ficheros

- Suministra una *visión lógica* de los dispositivos
- Ofrece primitivas de acceso cómodas e independientes de los detalles físicos (crear, borrar, leer, escribir, modificar, etc.)
- Implementa mecanismos de protección y recuperación frente a fallos
- Garantiza velocidad y eficiencia de uso

Ficheros: visión lógica y física

- Usuario: representación lógica (array de bytes)



- SO: visión física ligada a dispositivos (conjunto de bloques)

Fichero A	
Bloques:	13
	20
	1
	8
	3
	16
	19

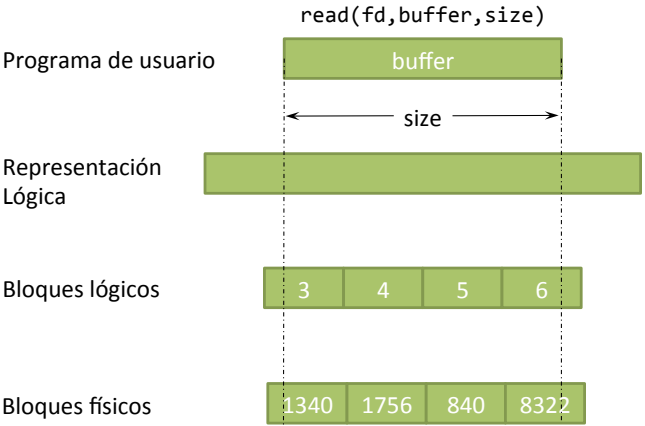
1	2	**3**	4	5	6
7	**8**	9	10	11	12
13	14	15	**16**	17	18
19	**20**	21	22	23	24
25	26	27	28	29	30

Bloque lógico vs. bloque físico

- El array de bytes de la representación lógica se divide en *bloques lógicos*
  - Si el tamaño de bloque es *k* bytes, un bloque lógico es un conjunto de *k* bytes contiguos en el fichero

- Un bloque físico representa una dirección de disco donde pueden almacenarse los datos de un bloque lógico

Operaciones sobre ficheros



Traducción dir. lógica ⇒ dir. física

- Los programas de usuario acceden a los datos de los ficheros empleando una dirección lógica (en bytes)
  - Para acceder a los datos, el SO debe obtener la *dirección física* correspondiente a la dirección lógica (traducción)
- Mecanismo de traducción
  1. Dir. lógica → Dir. bloque lógico = (Núm. bloque lógico, Offset)
     - Num. bloque lógico = $\left\lfloor \frac{\text{DirLógica(bytes)}}{\text{TamBloque(bytes)}} \right\rfloor$
     - Offset = DirLógica(bytes) **mod** TamBloque(bytes)
  2. Dir. bloque lógico → Dirección física = (Num. bloque físico, Offset)
     - Para establecer la correspondencia entre un bloque lógico y su bloque físico asociado se emplean estructuras de datos específicas de cada sistema de ficheros

Organización interna del sistema de ficheros

?	?	?	?	?	?
?	?	?	?	?	?
?	?	?	?	?	?
?	?	?	?	?	?
?	?	?	?	?	?

¿Cómo hacemos corresponder esos bloques con la imagen de fichero que tiene la aplicación?

¿Cómo asignamos los bloques de disco a un fichero cuando éste necesita más espacio?

Estructuras de datos

- El SO necesita un conjunto de estructuras datos para poder manipular ficheros y sistemas de ficheros

Estructuras de datos en disco

- El sistema de ficheros ha de ser autocontenido, no volátil e independiente del SO
- Además de los datos es preciso almacenar metadatos en el disco
  - Estructuras para llevar la cuenta de bloques libres y ocupados
  - Estructuras para realizar la correspondencia entre direcciones lógicas (en bytes) y direcciones físicas (bloque físico,desplazamiento)
  - ...

Estructuras de datos (II)

- Los datos han de estar memoria cuando sea necesario manipularlos

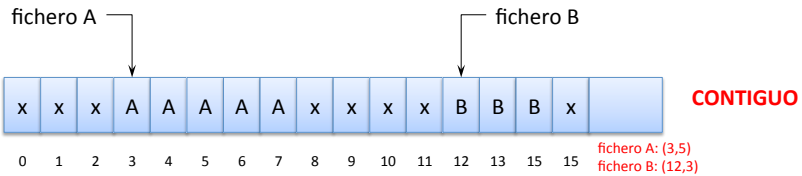
Estructuras de datos en memoria

- El SO mantiene estructuras de datos en memoria para poder realizar lecturas/escrituras sobre datos y metadatos más eficientemente
- No todos los datos de disco están en memoria

Estrategias de organización de ficheros

- Organización contigua
  - Usada en CD-ROM y cintas
- Organización enlazada
  - Usada en sistemas FAT (File Allocation Table)
- Organización indexada
  - Típica de UNIX-SV, FFF (Fast File System), ext2, etc.
- Organizaciones basadas en árboles equilibrados
  - NTFS, JFS, Reiser, XFS, etc.

Organización contigua



Ventajas

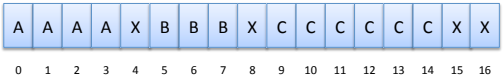
- Acceso secuencial óptimo
- Permite lecturas anticipadas
- Implementación sencilla de accesos aleatorios

Desventajas

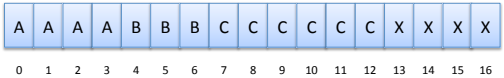
- Redimensionar un fichero puede ser costoso
- Fragmentación externa, predeclaración de tamaño
- Necesidad de compactación

Fragmentación externa (ejemplo)

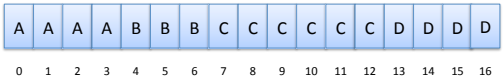
- Se desea crear un fichero de 4 bloques (D) en un SF con organización contigua



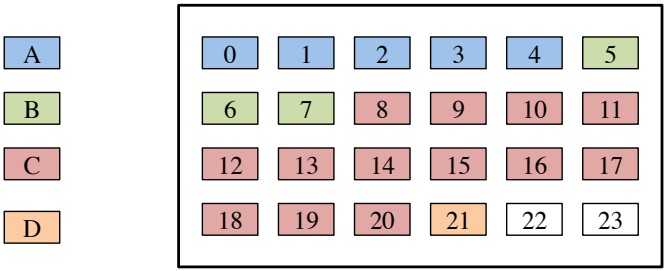
- 3 ficheros usan 13 de los 17 bloques disponibles
  - No es posible crear el fichero D en este escenario
- Aplicar compactación



- Crear el fichero

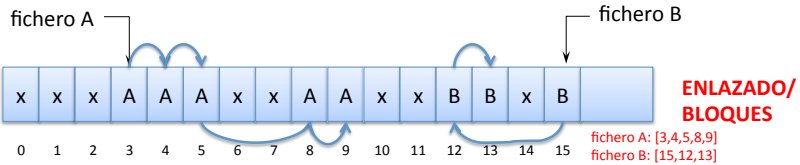


Organización contigua: CD-ROM (ISO-9660)



Organización enlazada

- Se emplea una lista enlazada de bloques para representar un fichero
- La lista puede ser de dos tipos:
  - Lista de bloques
  - Lista de grupos de bloques



Organización enlazada (II)

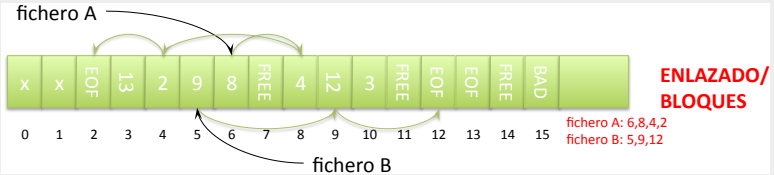
¿Dónde se almacenan los punteros de la lista?

- *Opción 1*: Dejar espacio al final de cada bloque de disco
  - Opción mala para acceso aleatorio
    - Acceso al bloque lógico $n \rightarrow$ Exige lectura de n bloques
  - Espacio para datos en un bloque $\neq 2^k$
- *Opción 2*: Dedicar una sección del disco aparte para almacenar los punteros
  - Estrategia utilizada en sistema de ficheros FAT

Organización enlazada (III)

File allocation table (FAT)

- Tabla que se almacena en una región especial de disco
  - Tiene tantas entradas como bloques direccionables tiene el disco
- Cada posición *i* de la tabla puede almacenar distintos valores:
  - Número: Puntero al siguiente bloque de lista enlazada
  - EOF: Indicador de fin de lista de bloques
  - FREE: Bloque i-ésimo está libre
  - BAD: Bloque i-ésimo está defectuoso



Organización enlazada (IV)

Ventajas

- No produce fragmentación externa
- Asignación dinámica simple
  - Cualquier bloque/grupo libre puede ser añadido al final de la lista de bloques
- Acceso secuencial fácil

Desventajas

- Accesos aleatorios requieren realizar un recorrido de la lista de bloques
- No toma en cuenta el principio de localidad
  - Los bloques de un fichero pueden estar desperdigados por el disco
- Es conveniente realizar compactaciones periódicas

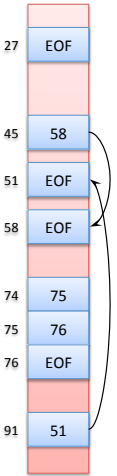
Organización enlazada: MS-DOS (FAT)

Directorio raíz

Nombre	Atrib.	KB	Agrup.
pep_dir	dir	5	27
fiche1.txt		12	45

Directorio pep_dir

Nombre	Atrib.	KB	Agrup.
carta1.wp	R	24	74
prue.zip		16	91



Las distintas variantes de FAT usan agrupaciones (clusters) en lugar de bloques.
Una agrupación es un conjunto de bloques contiguos.
Cada fichero ocupa al menos una agrupación en disco.

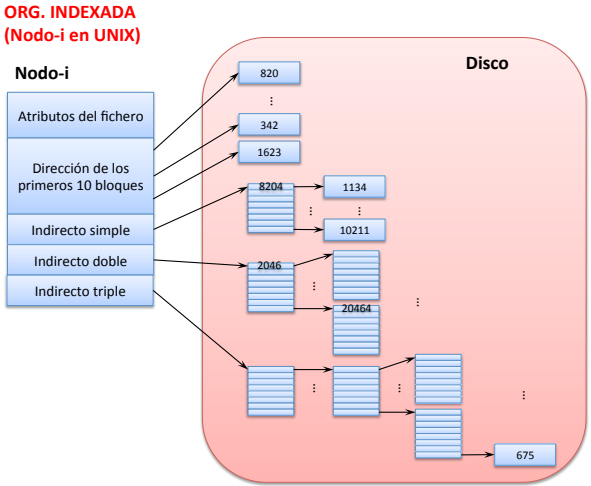
Organización indexada

- En la organización indexada, el descriptor físico del fichero contiene un conjunto de índices (punteros) para acceder a los datos
- Dos tipos de índices:
  1. **Directo**: apunta a un bloque de datos
  2. **Indirecto**: apunta a un bloque de índices
     - Cada bloque de índices contiene k punteros a bloques de datos o a otros bloques de índices

       $$k = \left\lfloor \frac{\text{TamBloque(bytes)}}{\text{TamÍndice(bytes)}} \right\rfloor$$

     - Distintos tipos de bloques de índices según su nivel de indirección: simples, dobles, triples, ...

Organización indexada: Nodos-I (UNIX)



Organización indexada: Nodos-I (UNIX)

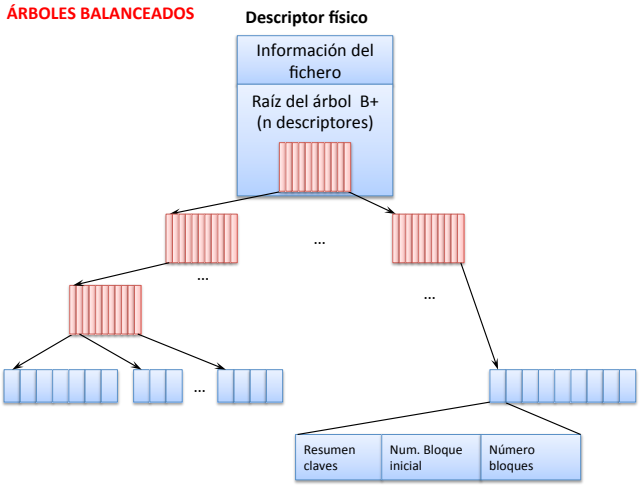
Ventajas

- No es necesario usar bloques de índices para ficheros pequeños (nodo-i almacena toda la información)
- Acceso secuencial y aleatorio sencillo
  - No es preciso recorrer listas enlazadas

Desventajas

- Nodo-i debe estar en memoria para garantizar accesos eficientes
- Accesos aleatorios tienen tiempo de acceso variable
  - El coste depende del nivel de índice asociado al bloque lógico al que accedemos
- Tamaño máximo de fichero limitado por índices multinivel:
  - T_b: tamaño bloque; d: número de índices directos; n: tam. índice
  - TamFicheroMax$=\left(d+(T_b/n)+(T_b/n)^2+(T_b/n)^3\right)\cdot T_b$

Organización basada en árboles equilibrados



Gestión del espacio libre

- El SO debe saber qué bloques y descriptores físicos de ficheros (p.ej., nodos-i) están libres
- Alternativas:
  1. Mapa de bits (ej: 11000000100101010101001...)
     - Tamaño mapa(bytes) = tam_disco(bytes) / (8 * tam_bloque(bytes))
     - Ejemplo: 16 GB / (8 * 1KB) = 2 MB
  2. Bloques libres encadenados
     - Lista de bloques libres implementada como pila o cola con parte en memoria
     - Variante: Lista de grupos de bloques libres (bloque-inicial, num-bloques)
  3. Indexación de bloques libres
     - El espacio libre modelado como un fichero con organización indexada

Contenido

Concepto de directorio

- Objeto que relaciona de forma unívoca un nombre de fichero (descriptor lógico) con su descriptor interno o físico
  - Se representa típicamente como una tabla (conjunto de entradas de directorio)
  - En algunos sistemas, como las distintas variantes de FAT, la entrada de directorio es en sí el descriptor físico del fichero
- Los directorios también permiten al usuario organizar su información de forma estructurada
  - Un directorio es un nodo en sistemas de ficheros jerárquicos

Propiedades deseables de los directorios

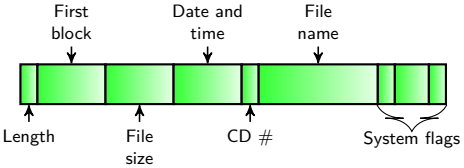
Propiedades

- **Eficiencia**: localizar un fichero rápidamente
- **Sencillez**: la entrada de directorio debe ser lo más sencilla posible
- **Nombrado**: conveniente y sencillo para los usuarios
  - Dos usuarios pueden tener el mismo nombre para ficheros distintos
  - Un mismo fichero puede tener nombres distintos
  - Nombres de longitud variable
- **Agrupación**: agrupación lógica de los ficheros según sus propiedades (por ejemplo: programas C, juegos, etc.)
- **Visión estructurada**: operaciones claramente definidas y ocultación

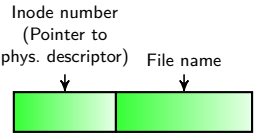
Estructura de los directorios

- Tanto la estructura del directorio como los ficheros residen en discos
- Los directorios se suelen implementar como ficheros con formato conocido para el SO
- Se han explorado principalmente dos implementaciones de directorio:
  1. Cada entrada en un directorio es el descriptor físico del fichero
     - Los atributos del fichero se almacenan en su entrada del directorio
     - Ejemplo: CD-ROM, FAT
  2. Directorio es simplemente una tabla de pares *(nombre_fichero, ID)*
     - ID es un identificador único ("puntero") para referirse al descriptor físico del fichero
     - Ejemplo: UNIX

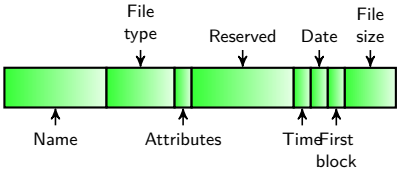
Contenido de la entrada de directorio



Entrada de directorio en ISO-9660 (CD-ROM)



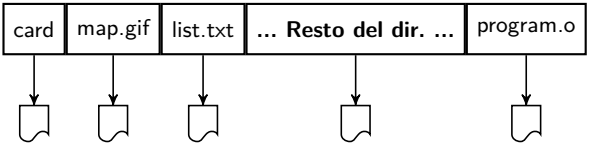
Entrada de directorio en Unix SV



Entrada de directorio en FAT (MS-DOS)

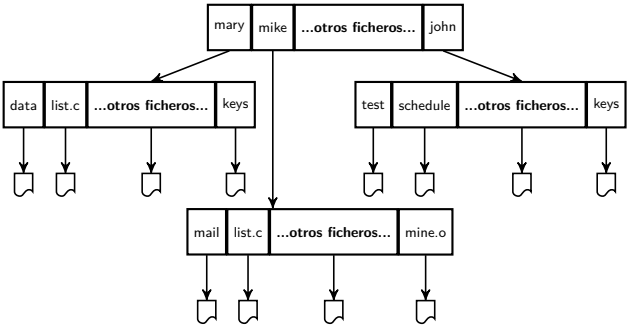
Directorio de un nivel

- Un único directorio en el sistema
  - Compartido por todos los usuarios
- Problemas de nombrado y agrupación



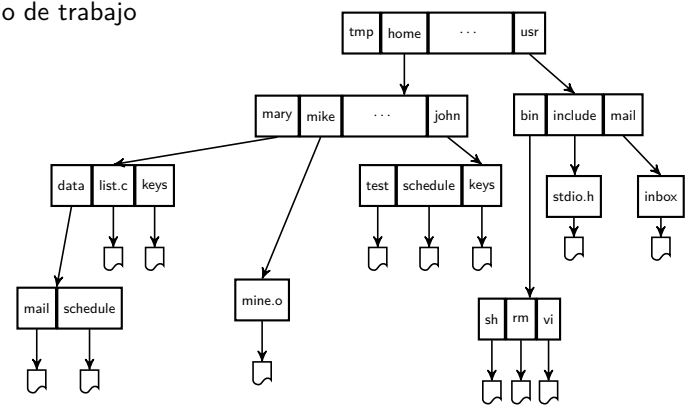
Directorio de dos niveles

- Un directorio por cada usuario
- Caminos de acceso pueden ser explícitos (*absolutos*) o implícitos (relativos al directorio del usuario)
- Varios usuarios pueden poseer ficheros distintos con el mismo nombre
- Búsqueda eficiente, pero problemas de agrupación



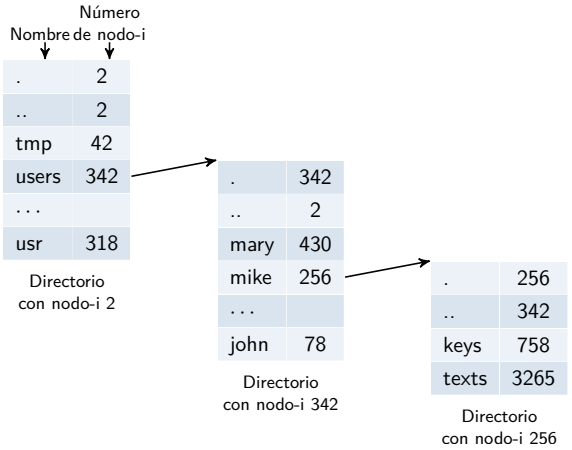
Directorio con estructura de árbol

- Búsqueda eficiente y agrupación
- Necesario mecanismos más sofisticados para referirse a los ficheros en la jerarquía
  - Nombres relativos y absolutos
  - Directorio de trabajo



Interpretación de nombres en Linux (I)

- Obtener el descriptor físico del fichero a partir de una ruta
- Ejemplo para `/users/mike/keys`



Interpretación de nombres en Linux (II)

Pasos para procesar `/users/mike/keys`

- Traer a memoria entradas fichero con nodo-i 2
- Se busca dentro `users` y se obtiene el nodo-i 342
- Traer a memoria entradas fichero con nodo-i 342
- Se busca dentro `mike` y se obtiene el nodo-i 256
- Traer a memoria entradas fichero con nodo-i 256
- Se busca dentro `keys` y se obtiene el nodo-i 758
- Se lee el nodo-i 758 y ya se tienen los datos del fichero

Interpretación de nombres en Linux (III)

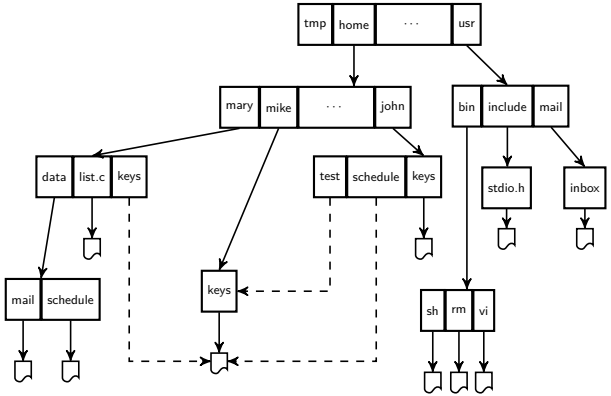
¿Cuándo para el algoritmo iterativo?

1. Se ha encontrado el nodo-i del fichero
2. No se ha encontrado y no hay más subdirectorios
3. Estamos en un directorio y no contiene la siguiente componente del nombre (por ejemplo, `mike`).

Contenido

Directorio de grafo acíclico

- Dos rutas no equivalentes pueden llevarnos al mismo fichero
- El sistema de ficheros debe soportar la creación de *alias*

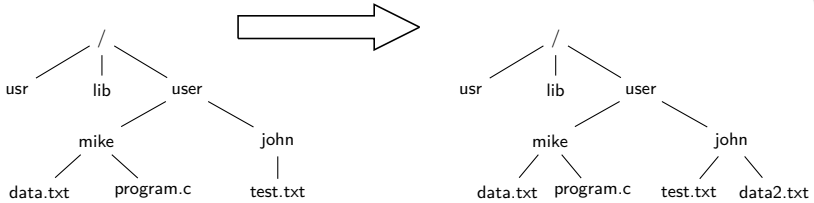


Jerarquías de grafo acíclico: enlaces (I)

- **Enlace**: una de los posibles nombres para referirse a un archivo
  - **Físico o rígido**: Un único fichero con contador enlaces en descriptor
    - Número de alias del fichero en el sistema
  - **Simbólico**: Un fichero especial que almacena la ruta de otro fichero
    - Se dice que un enlace simbólico esta *roto* si la ruta que almacena no existe
- Posibles estrategias de borrado de enlaces:
  1. (enlaces rígidos): Decrementar contador de enlaces; si 0 → borrar fichero
  2. Recorrer los enlaces y borrar todos
  3. (enlaces simbólicos): Borrar únicamente el enlace y dejar los demás

Enlace físico: ejemplo

```
$ ln /user/mike/data.txt /user/john/data2.txt
```



mike

.	23
..	100
data.txt	28
program.c	400

john

.	80
..	100
test.txt	60

mike

.	23
..	100
data.txt	28
program.c	400

john

.	80
..	100
test.txt	60
data2.txt	28

nodo-i 28

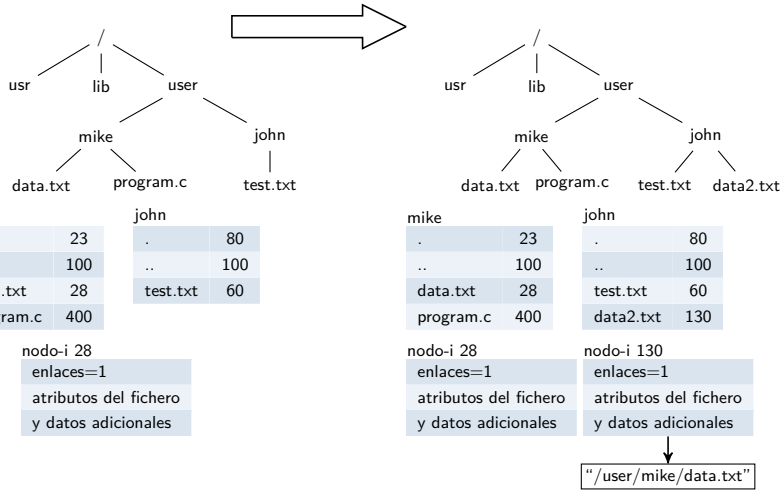
enlaces=1
atributos del fichero
y datos adicionales

nodo-i 28

enlaces=2
atributos del fichero
y datos adicionales

Enlace simbólico: ejemplo

```
$ ln -s /user/mike/data.txt /user/john/data2.txt
```



Jerarquías de grafo acíclico: enlaces (II)

Problemas y limitaciones

- La creación de enlaces puede conllevar la creación de ciclos en el grafo. Dos soluciones:
  1. Permitir sólo enlaces a ficheros, no a directorios
  2. Algoritmo de búsqueda de ciclos cuando se solicita la creación de un enlace
- En UNIX, solo se permite crear enlaces físicos dentro del mismo sistema de ficheros
  - Enlaces simbólicos sí permitidos entre sistemas de ficheros

Enlaces físicos vs. simbólicos

Terminal

```
jcsaez@debian:~/OS/links$ ls
info.pdf
jcsaez@debian:~/OS/links$ ln info.pdf alias1
jcsaez@debian:~/OS/links$ ln -s info.pdf alias2
jcsaez@debian:~/OS/links$ stat info.pdf
  File: ‘info.pdf’
  Size: 68883       Blocks: 136        IO Block: 1048576 regular file
Device: 24h/36d Inode: 26802399    Links: 2
Access: (0644/-rw-r--r--)  Uid: ( 1058/  jcsaez)   Gid: ( 1060/  jcsaez)
Access: 2015-08-31 14:36:43.564966921 +0200
Modify: 2015-08-31 14:36:43.585657807 +0200
Change: 2015-08-31 14:37:00.532906762 +0200
 Birth: -
```

- Se crean dos enlaces al fichero `info.pdf`
  - `alias1` (enlace físico)
  - `alias2` (enlace simbólico)
- El comando `stat` imprime por pantalla los atributos de un ficheros, así como el número de nodo-i y el valor actual del contador de enlaces

Enlaces físicos vs. simbólicos

Terminal

```
jcsaez@debian:~/OS/links$ stat alias1
  File: ‘alias1’
  Size: 68883       Blocks: 136        IO Block: 1048576 regular file
Device: 24h/36d Inode: 26802399    Links: 2
Access: (0644/-rw-r--r--)  Uid: ( 1058/  jcsaez)   Gid: ( 1060/  jcsaez)
Access: 2015-08-31 14:36:43.564966921 +0200
Modify: 2015-08-31 14:36:43.585657807 +0200
Change: 2015-08-31 14:37:00.532906762 +0200
 Birth: -
jcsaez@debian:~/OS/links$ stat alias2
  File: ‘alias2’ -> ‘info.pdf’
  Size: 8           Blocks: 1          IO Block: 1048576 symbolic link
Device: 24h/36d Inode: 26802400    Links: 1
Access: (0777/lrwxrwxrwx)  Uid: ( 1058/  jcsaez)   Gid: ( 1060/  jcsaez)
Access: 2015-08-31 14:37:06.702552995 +0200
Modify: 2015-08-31 14:37:06.702552995 +0200
Change: 2015-08-31 14:37:06.702552995 +0200
 Birth: -
```

- `info.pdf` y `alias1` “comparten” el mismo nodo-i
- `alias2` no es un fichero regular y usa un nodo-i diferente

Crear una entrada de directorio

link() y symlink()

```
int link(const char *existing, const char *new);
int symlink(const char *existing, const char *new);
```

- Argumentos:
  - `existing`: nombre del fichero existente
  - `new`: nombre de la nueva entrada de directorio
- Valor de retorno:
  - Cero ó -1 si error
- Descripción:
  - Crea un nuevo enlace, físico o simbólico, para un fichero existente
  - El sistema no registra cuál es el enlace original
  - `existing` no debe ser el nombre de un directorio

Eliminar una entrada de directorio

unlink()

```
int unlink(const char* path);
```

- Argumentos:
  - `path`: nombre del archivo
- Valor de retorno:
  - Devuelve 0 ó -1 si error.
- Descripción:
  - Elimina la entrada de directorio y decrementa el número de enlaces del archivo correspondiente.
  - Cuando el número de enlaces es igual a cero y ningún proceso lo mantiene abierto, se libera el espacio ocupado por el fichero y el fichero deja de ser accesible

Contenido

Sistemas de ficheros

- El sistema de ficheros permite organizar la información dentro de los dispositivos de almacenamiento secundario en un formato inteligible para el SO
- Previamente a la instalación del SF, es necesario dividir físicamente, o lógicamente, los discos en particiones o volúmenes
  - Una partición es una porción de un disco a la que se la dota de una identidad propia y que puede ser manipulada por el SO como una entidad lógica independiente
  - Estrategias de particionado típicas:
    - Master Boot Record (MBR)
    - GUID/GPT - Mac OS X (x86)
    - Apple Partition Map (APM) - Mac OS (PowerPC)
- Una vez creadas las particiones, el SO debe crear las estructuras de los SF dentro de esas particiones
  - Para ello se proporcionan al usuario comandos como `format` (Windows) o `mkfs` (UNIX)

Metadatos de una partición

- El sistema de ficheros instalado en una partición ha de ser autocontenido
  - Tanto datos como metadatos del SF se almacenan en disco

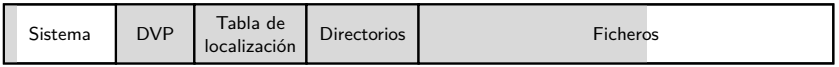
Tipos de metadatos

1. **Parámetros globales del SF**
   - Tamaño de la partición
   - Tamaño de bloque
   - Número de bloques de datos y de descriptores físicos
2. **Metadatos críticos**
   - Directorio raíz
   - Tablas de índices (en FAT) o array de descriptores físicos (nodos-i en UNIX)
3. **Estructuras para gestión del espacio libre**
   - Control de bloques del disco libres/ocupados
   - Control de descriptores físicos libres/ocupados (p.ej., nodos-i)

Sistemas de ficheros y particiones

- Volumen o partición: conjunto coherente de metainformación y datos.
- Ejemplos:

CD-ROM

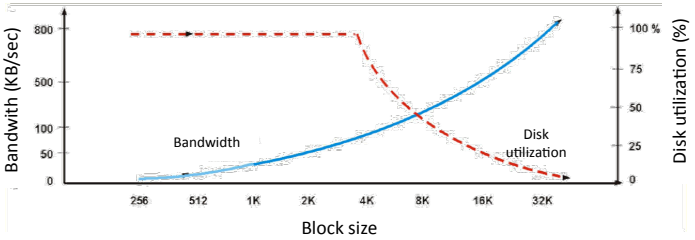


FAT (MS-DOS)

UNIX

Tamaño de bloque

- **Bloque**: agrupación lógica de sectores de disco y es la unidad de transferencia mínima que usa el sistema de ficheros.
  - Todo fichero ocupa al menos un bloque en disco
  - La elección del tamaño de bloque tiene un gran impacto en el rendimiento de la E/S
  - Todos los sistemas operativos definen un tamaño de bloque por defecto
  - Los usuarios pueden definir el tamaño de bloque a usar en el sistema de ficheros al instalarlo en una partición



Fragmentación interna

Ejemplo

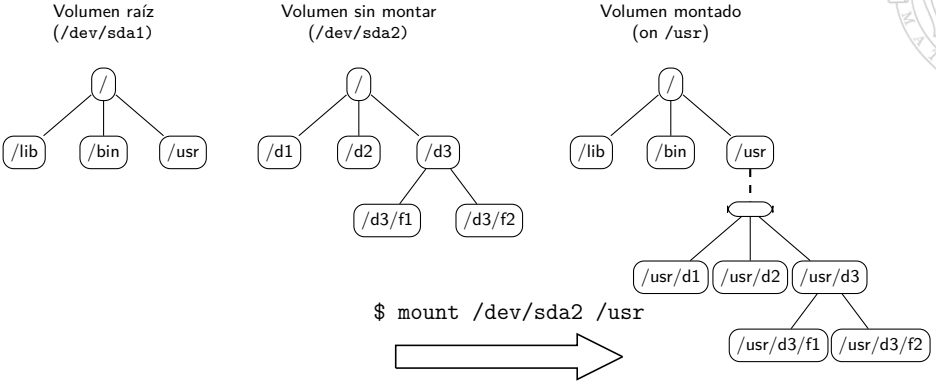
- Sistema de ficheros donde el volumen está completamente lleno de ficheros de 7KB cada uno
  - TBloque = 2KB
- Calcular el grado de fragmentación interna (fracción de disco desaprovechada) suponiendo despreciable el espacio ocupado por los metadatos del SF
  - ¿Cuánto espacio se necesita para almacenar un fichero de 7KB?
    - $\text{NBloques} = \left\lceil \frac{\text{TFichero(bytes)}}{\text{TBloque(bytes)}} \right\rceil = \left\lceil \frac{7KB}{2KB} \right\rceil = 4 bloques\ (8KB)$

    - Porcentaje de espacio desaprovechado $= \frac{1KB}{8KB} \cdot 100 = 12{,}5\,\%$
- Para un disco de 1TB $\rightarrow$ se desperdiciarían 125GB!!

Gestión de múltiples sistemas de ficheros

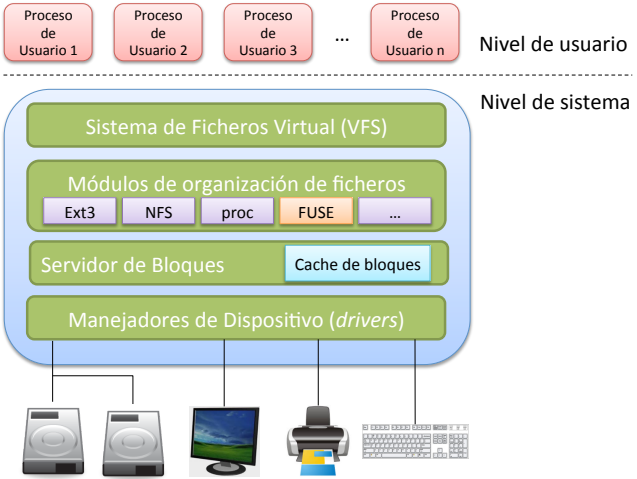
- *¿Cómo exponer múltiples sistemas de ficheros al usuario?*
  - En Windows se emplea un árbol de directorios distinto por cada sistema de ficheros (`c:\users\mike\keys`, `j:\joe\tmp`)
  - En UNIX/Linux se mantiene un árbol único para todo el sistema (`/users/mike/keys`, `/mnt/joe/tmp`, …)
- Tener un árbol único proporciona una imagen unificada del sistema y oculta el tipo de dispositivo al usuario
- En UNIX se proporcionan comandos de administración para mantener el árbol único: `mount`, `umount`
  - `mount /dev/sda3 /users`
  - `umount /users`
- Lamentablemente, tener un único árbol complica la traducción de nombres, y origina problemas relacionados con los enlaces

Montaje de sistemas de ficheros o particiones



Contenido

Arquitectura del Sist. de Gestión de Ficheros



Componentes del Sist. de Gestión de Ficheros

Sistema de ficheros virtual (VFS)

- Capa encargada de proporcionar la interfaz de llamadas al sistemas
  - Manejo de directorios e interpretación de rutas
  - Servicios genéricos de ficheros (API SO)
  - Abstracción de los distintos sistemas de archivos. Ficheros abiertos representados por mediante un envoltorio (`wrapper`) del descriptor físico en cuestión: nodo-v (nodo virtual)

Componentes del Sist. de Gestión de Ficheros

Módulos de organización de ficheros:

- Implementan los servicios para cada tipo de sistema de ficheros soportado por el SO
  - Un módulo por cada tipo de sistemas de fichero soportado (UNIX, EXT3, NTFS, FAT etc.) o pseudo-ficheros como /proc
  - Proporcionan algoritmos para traducir direcciones lógicas de bloques a sus correspondientes direcciones físicas
  - Gestionan el espacio de los sistemas de ficheros, la asignación de bloques a ficheros y el manejo de los descriptores internos de fichero (i-nodos de UNIX o entradas de directorio FAT).

Componentes del Sist. de Gestión de Ficheros

Servidor de bloques:

- Emite los mandatos genéricos para leer y escribir bloques a los manejadores de dispositivo
  - Ej: Leer bloque 320 del sistema de ficheros #1 → `read_block 320 /dev/sda2`
- Implementa mecanismos de optimización de la E/S como la cache de bloques

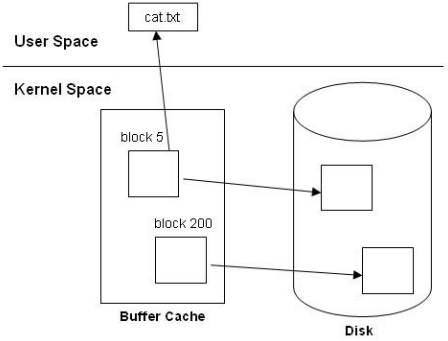
Componentes del Sist. de Gestión de Ficheros

Manejadores de dispositivo (Drivers):

- Recibe ordenes de E/S de alto nivel y las traduce al formato que entiende el controlador HW del dispositivo en cuestión
- Existe un *driver* por cada clase de dispositivo específico que hay en el sistema
  - Para cada dispositivo de almacenamiento hay una cola de peticiones que el *driver* debe planificar adecuadamente para obtener buen rendimiento de la E/S
  - Explotar la localidad de los accesos garantiza un mayor ancho de banda

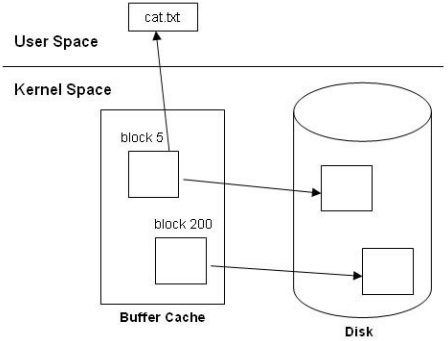
Cache de bloques

- El acceso a disco es órdenes de magnitud más lento que el acceso a RAM
- **Cache de bloques**: Región de memoria que se usa como cache controlada por software para almacenar bloques de los dispositivos de almacenamiento
  - Explota principios de localidad espacial y localidad temporal



Cache de bloques

- Principal problema: consistencia de la información
  - Los datos pueden estar actualizados en la cache de bloques pero no en disco
- El SO debe implementar una **política de reemplazo** y una **política de escritura** para la cache de bloques



Políticas de reemplazo

Escenario

- Cuando el servidor de bloques (SB) recibe una petición de L/E de un bloque específico, debe buscarlo en la *cache de bloques*
  - En caso de que no esté, se lee del dispositivo y se copia a la cache
  - Si la cache está llena, es necesario hacer hueco para el nuevo bloque reemplazando uno de los existentes

Políticas de reemplazo

- **FIFO** (First In First Out)
- **MRU** (Most Recently Used)
- **LRU** (Least Recently Used)
  - Los bloques más usados se mantienen en la cache (RAM)
  - Política más común

Políticas de escritura (I)

- **Escritura inmediata** (*write-through*): se escribe cada vez que se modifica el bloque
  - No hay problema de fiabilidad, pero se reduce el rendimiento del sistema.
- **Escritura diferida** (*write-back*): sólo se escriben los datos a disco cuando se eligen para su reemplazo por falta de espacio en la cache.
  - Optimiza el rendimiento, pero genera los problemas de fiabilidad anteriormente descritos.

Políticas de escritura (II)

- **Escritura retrasada** (*delayed-write*), que consiste en escribir a disco los bloques de datos modificados en la cache de forma periódica cada cierto tiempo (30 segundos en UNIX).
  - Compromiso entre rendimiento y fiabilidad.
  - Reduce la extensión de los posibles daños por pérdida de datos.
  - Los bloques especiales se escriben inmediatamente al disco.
  - No se puede quitar un disco del sistema sin antes volcar los datos de la cache.
- **Escritura al cierre** (*write-on-close*): cuando se cierra un fichero, se vuelcan al disco los bloques del mismo que tienen datos actualizados.