



# Sistemas Operativos

Curso  
2016-2017

Módulo 2: Gestión de archivos y directorios



# Contenido

- Ficheros
- Directorios
- API del Sistema Operativo
  - Ficheros
  - Directorios
- Aumento de Prestaciones



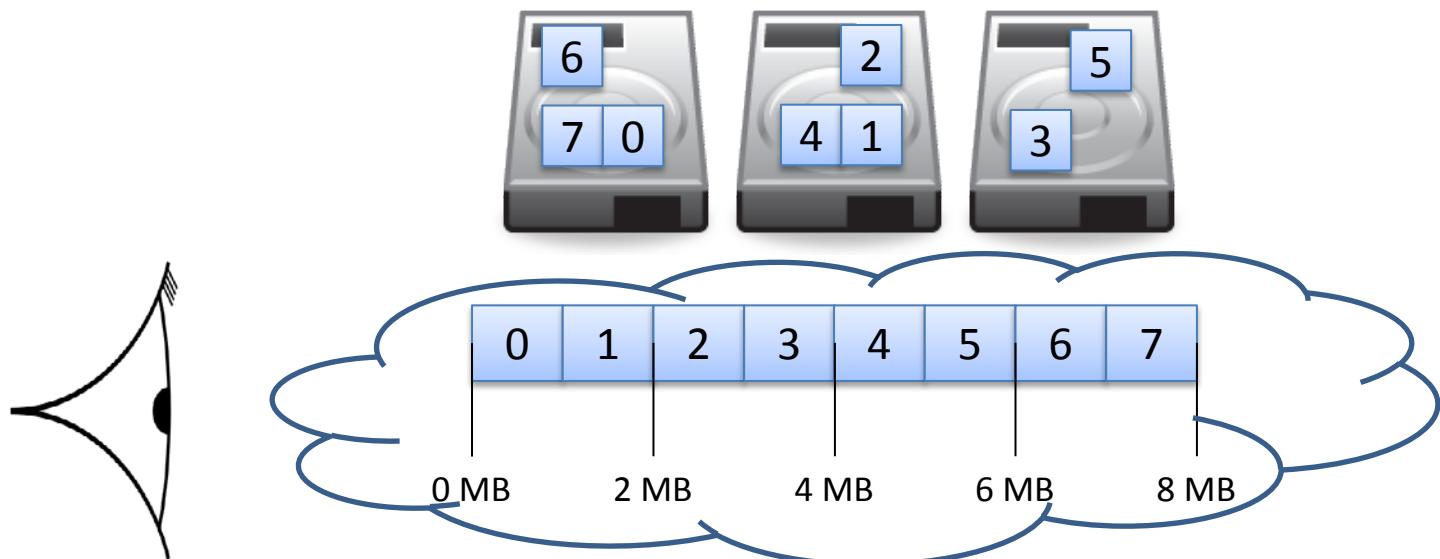
# ¿Qué es un fichero?

- 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 como fuente de entradas a los programas y para guardar a largo plazo las salidas de los programas
- El sistema de Gestión de Ficheros se encarga de:
  - Gestionar los permisos de accesos
  - Garantizar la integridad de atributos y contenidos
  - Soportar los ficheros sobre medios de almacenamiento
- El Sistema de Gestión de Ficheros suele estar diseñado como una aplicación especial integrada en parte o totalmente en el S.O.



# Visión del usuario

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



Visión física (arriba) y lógica (abajo) de un fichero



# Características para el usuario

- 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
- Abstraen los dispositivos de almacenamiento físico
- Se accede a ellos a través de llamadas al sistema operativo o de bibliotecas de utilidades



# Concepto de archivo

- Un espacio lógico de direcciones contiguas usado para almacenar datos
- Tipos de archivos:
  - Datos:
    - Carácter
    - Binarios
  - Programas:
    - Código fuente
    - Archivos objetos (imagen de carga)
  - Documentos



# Atributos del archivo

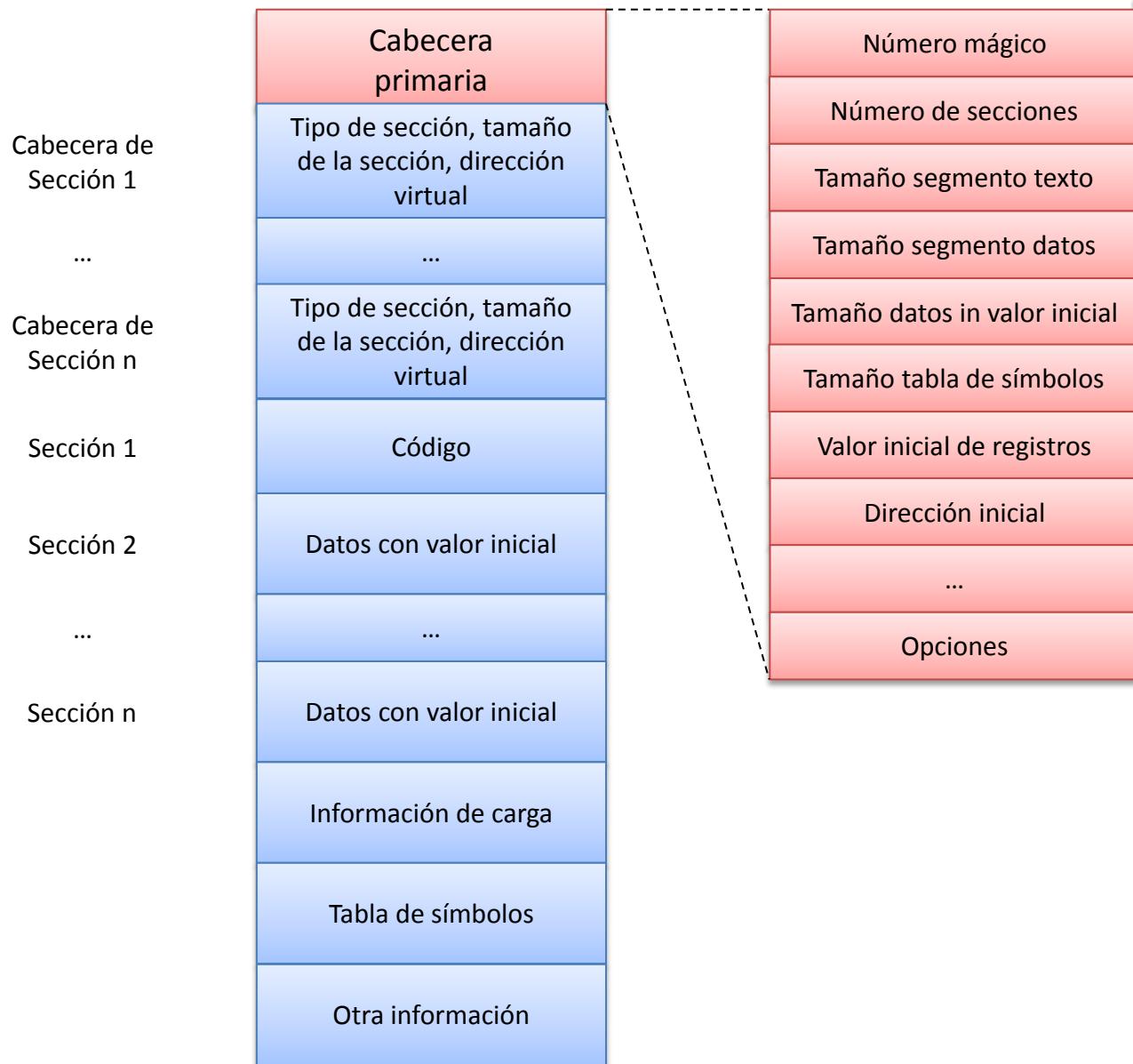
- **Nombre:** la única información en formato legible por una persona.
- **Identificación única del archivo y del usuario:** descriptor interno del archivo, dueño y grupo del archivo
- **Tipo de archivo:** necesario en sistemas que proporcionan distintos formatos de archivos: normales y especiales
- **Tamaño del archivo:** número de bytes en el archivo, máximo tamaño posible, etc.
- **Protección:** control de acceso
- **Información temporal:** de creación, de acceso, de modificación, etc.
- **Información de control:** archivo oculto, de sistema, normal o directorio, etc.



# Nombres de fichero y extensiones

- Todo objeto del SO debe tener un nombre o descriptor único. Los ficheros tienen descriptor físico (interno) y lógico (pues son manipulados desde el exterior)
- **Nombre lógico:** el tipo y longitud cambia de un sistema a otro:
  - Longitud máxima: 8 en MS-DOS, 4096 en UNIX
  - Sensibles a tipografía: CATALINA y catalina son el mismo archivo en Windows pero distintos en LINUX
  - Extensión: obligatoria o no, más de una o no, fija para cada tipo de archivos, etc. Las extensiones son significativas para las aplicaciones (html, c, cpp, etc.)
- **Nombre interno:** cuando se crea un fichero, el SO le asigna un identificador interno único que es el que realmente se utiliza en todas las operaciones del SO (UNIX, LINUX y WINDOWS usan números enteros).
  - Los directorios relacionan nombres lógicos y descriptores internos de ficheros
- El SO sólo distingue algunos formatos (ejecutables, texto, ...).
  - Ejemplo: número mágico UNIX

# Estructura de archivo ejecutable LINUX





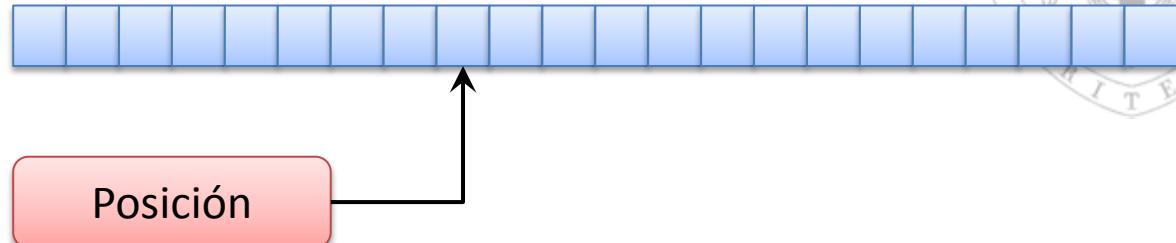
# Sistema de archivos

- El acceso 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 archivos es la capa de software entre dispositivos y usuarios
- Objetivos:
  - Suministrar una visión lógica de los dispositivos
  - Ofrecer primitivas de acceso cómodas e independientes de los detalles físicos (crear, borrar, leer, escribir, modificar, etc.)
  - Mecanismos de protección y recuperación frente a fallos
  - Velocidad y eficiencia de uso



# Archivos: visión lógica y física

- Usuario: Visión lógica
  - Secuencia de bytes



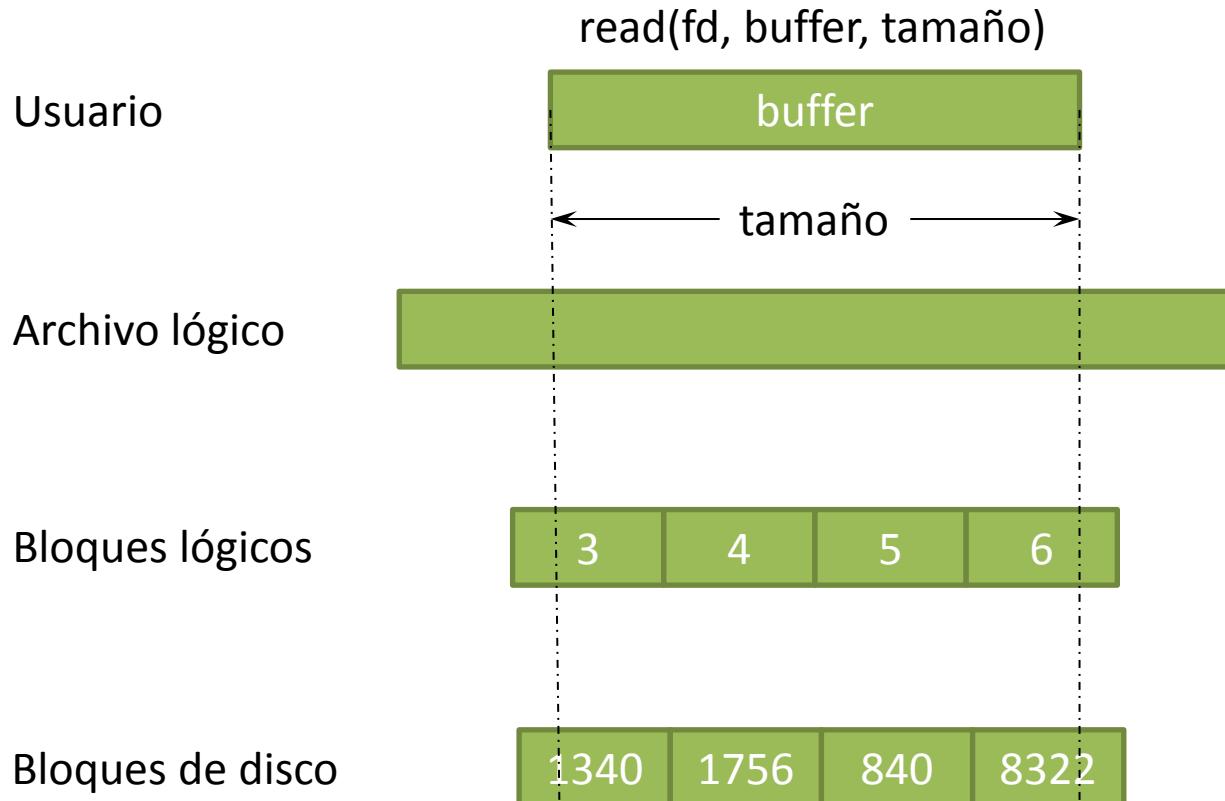
- Sistema operativo: visión física ligada a dispositivos
  - Conjunto de bloques

Archivo A	
Bloques:	13
	20
	1
	8
	3
	16
	19





# Operaciones en ficheros





# Estructura y almacenamiento del fichero



¿Cómo asignamos los bloques de disco a un fichero?

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

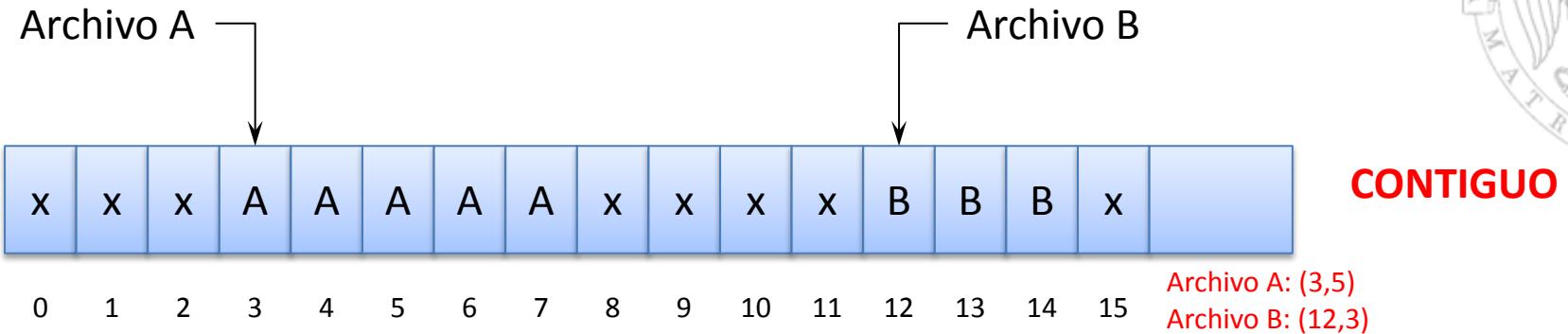


# Estructura y almacenamiento del fichero

- Ficheros contiguos
  - Usado en CD-ROM y cintas
- Enlazados
  - Usado en sistemas FAT (File Allocation Table)
- Indexados
  - Típico en UNIX-SV, FFF (Fast File System), ext2, etc.
- Árboles balanceados
  - NTFS, JFS, Reiser, XFS, etc.



# Ficheros Contiguos



- Ventajas:
  - Acceso secuencial óptimo, permite lecturas anticipadas, fácil acceso aleatorio
- Desventajas:
  - Fragmentación externa, pre-declaración de tamaño, necesidad de compactación
- Tamaño máximo de fichero:
  - Num. Bloques dispositivo x Tam. bloque



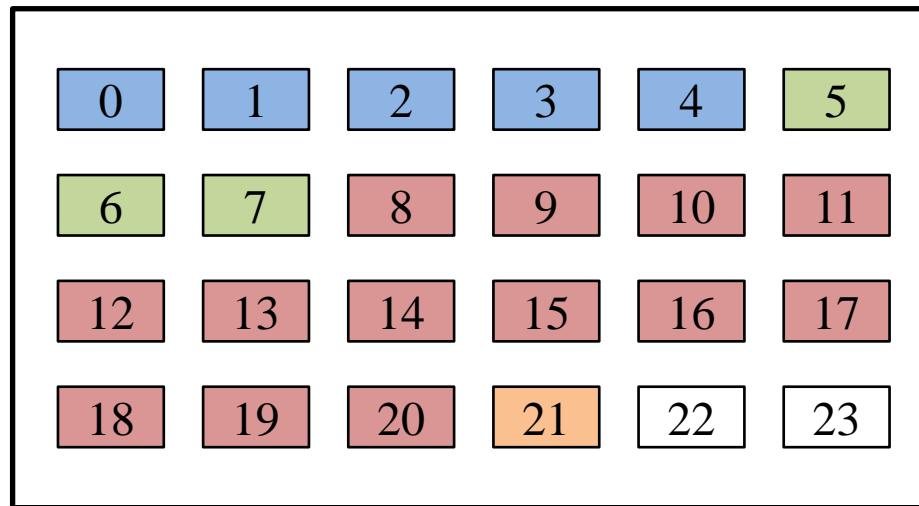
# Ficheros contiguos: CD-ROM (ISO-9660)

A

B

C

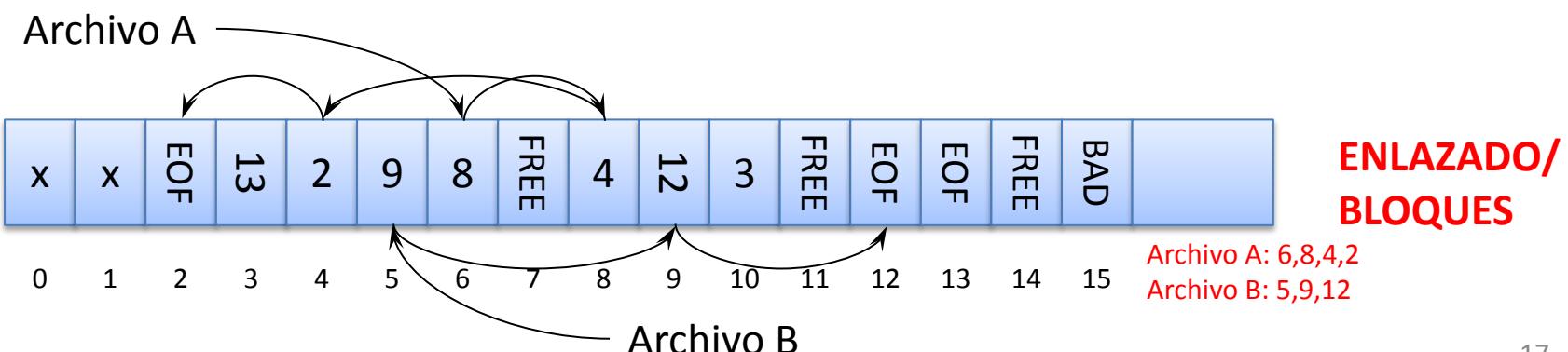
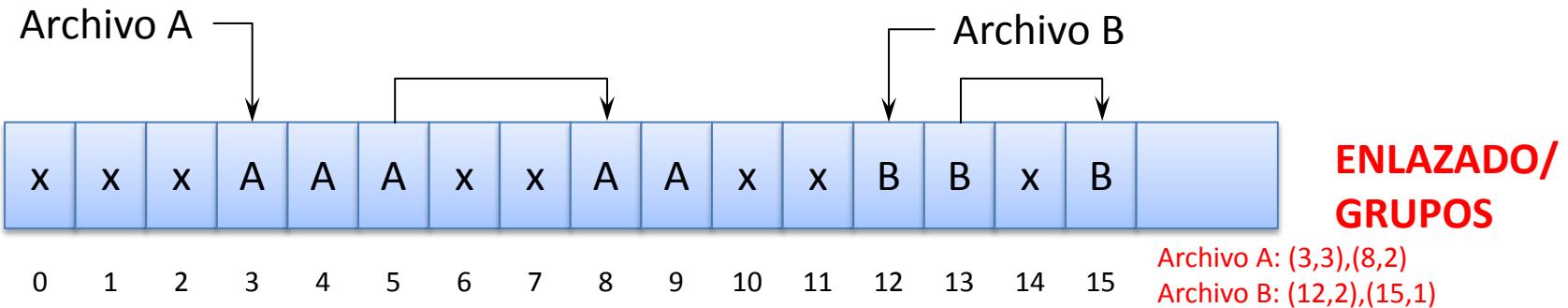
D





# Ficheros enlazados

- Se basan en el uso de una lista de bloques enlazados para representar un fichero. La lista puede ser de dos tipos:
  - Lista de grupos de bloques
  - Lista de bloques





# Ficheros enlazados

- Ventajas:
  - No produce fragmentación externa
  - Asignación dinámica simple: Cualquier clúster libre puede ser añadido a la cadena
  - Acceso secuencial fácil
- Desventajas:
  - No toma en cuenta el principio de localidad, falta de contigüidad
  - Acceso aleatorio complicado
  - Es conveniente realizar compactaciones periódicas



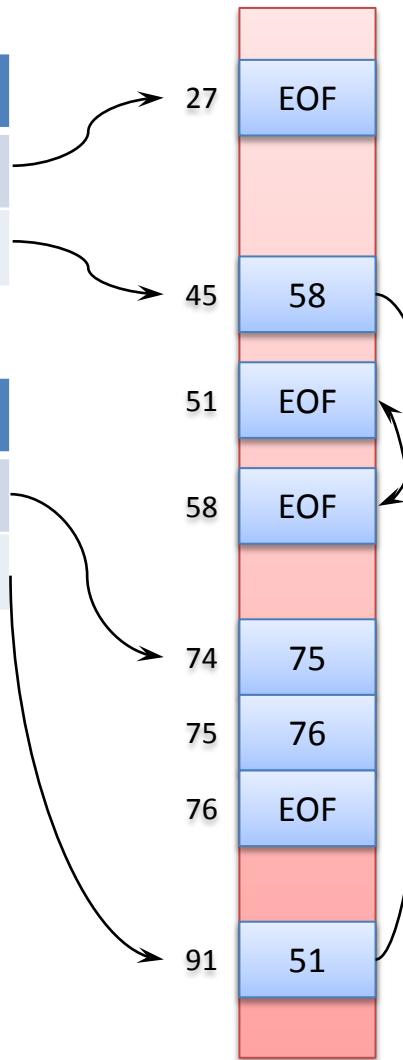
# Ficheros enlazados: MS-DOS (FAT)

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



Recuerda:  
Lista enlazada/grupos  
Como son bloques consecutivos,  
no son necesarios los punteros



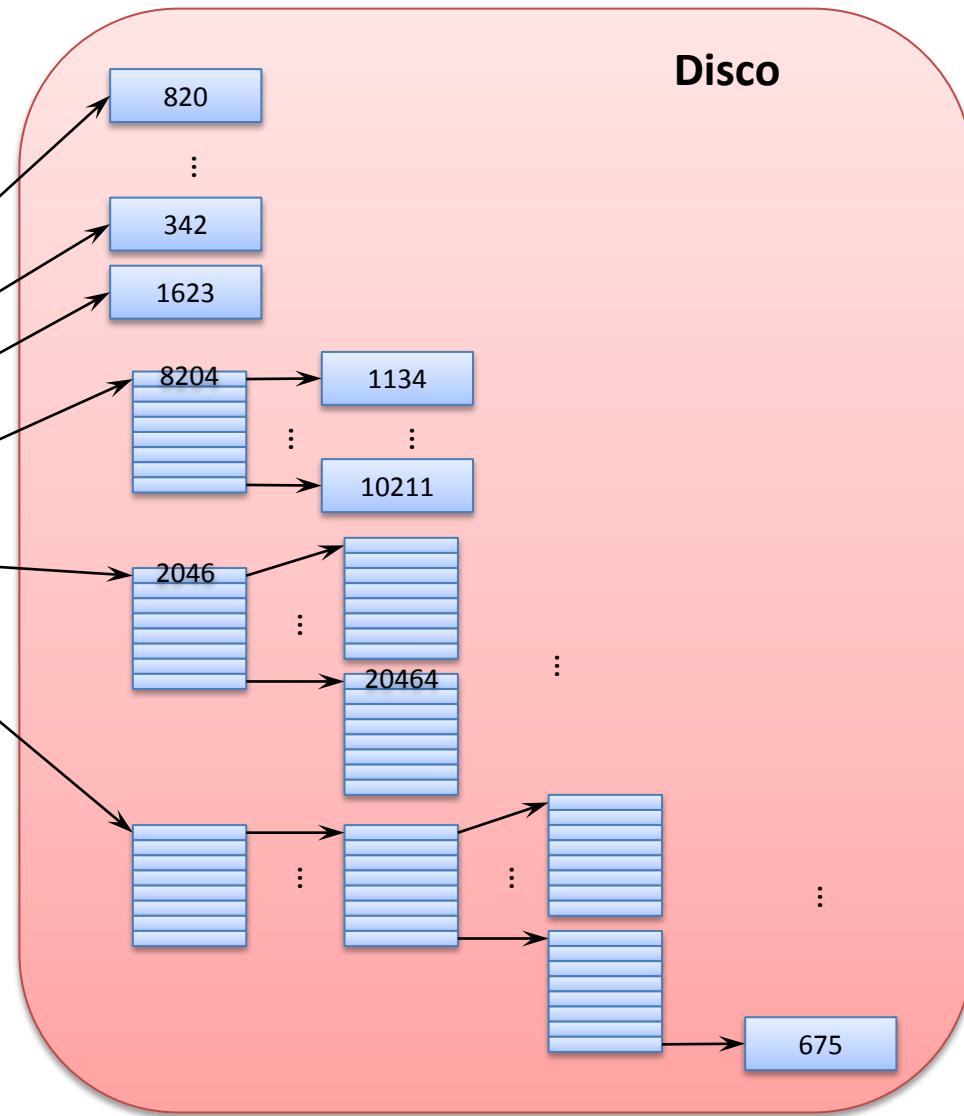
# Ficheros indexados

**INDEXADO**  
**(Nodo-i en UNIX)**

**Nodo-i**

Información del archivo
Dirección de los primeros 10 bloques
Indirecto simple
Indirecto doble
Indirecto triple

**Disco**





# Ficheros indexados: Nodos-i en UNIX

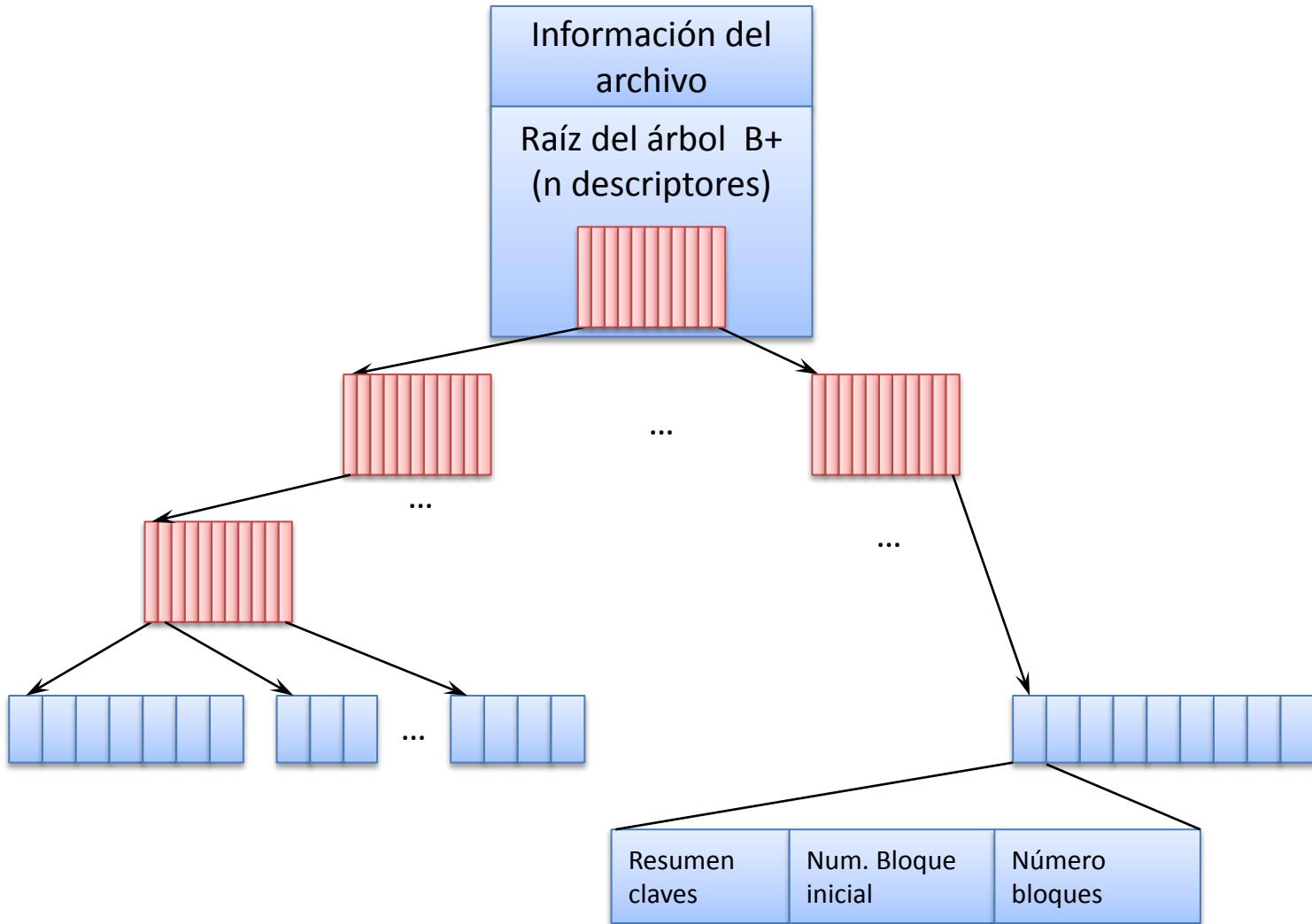
- Tamaño máximo de archivo
  - Sb: tamaño del bloque
  - Direcciones de bloques de 4 bytes
  - $10Sb + (Sb/4)Sb + (Sb/4)^2Sb + (Sb/4)^3Sb$



# Árboles balanceados

## ÁRBOLES BALANCEADOS

Nodo-i





# Gestión de Espacio Libre

- Mapa de bits
  - Tamaño mapa = Tamaño\_de\_disco / (8 \* Tamaño\_de\_bloque)
  - Ej.: 16 GB / (8 \* 1KB) = 2 MB
- Bloques libres encadenados
- Indexación de bloques libres
  - El espacio libre como fichero con indexación sobre bloques de tamaño variable
  - Variante: Indexación con zonas de tamaño variable
- Lista de bloques libres implementada como pila o cola con parte en memoria



# Semántica de cutilización

- Cualquier forma de acceso tiene problemas cuando varios usuarios trabajan con el archivo simultáneamente
- Semántica de cutilización: especifica el efecto de varios procesos accediendo de forma simultánea al mismo archivo y cuando se hacen efectivas las modificaciones
- Tipos de semánticas:
  - Semántica UNIX (POSIX)
    - Las escrituras son inmediatamente visibles para todos los procesos con el archivo abierto
    - Los procesos pueden compartir archivos. Si existe relación de parentesco pueden compartir el puntero. La cutilización afecta también a los metadatos



# Semántica de coutilización (II)

## – Semántica de sesión

- Las escrituras que hace un proceso no son inmediatamente visibles para los demás procesos con el archivo abierto
- Cuando se cierra el archivo los cambios se hacen visibles para las futuras sesiones
- Un archivo puede asociarse temporalmente a varias imágenes

## – Semántica de versiones

- Las actualizaciones se hacen sobre copias con nº versión
- Sólo son visibles cuando se consolidan versiones
- Sincronización explícita si se requiere actualización inmediata

## – Semántica de archivos inmutables

- Una vez creado el archivo sólo puede ser compartido para lectura y no cambia nunca



# Tablas del Servidor de Ficheros

- Tabla de descriptores de ficheros abiertos (**TDDA**)
  - Una por proceso
- Tabla intermedia de posiciones (**TFA**). Global (única)
- Tabla intermedia de *nodos-i* (**TIN**). Global (única)

FD	IDFF
0	23
1	4563
2	56
3	4
4	678

FD	IDFF
0	230
1	563
2	98
3	3

FD	IDFF
0	2300
1	53
2	3
3	465
4	326

Nodo-i	Contador
...	
info de nodo-i 98	2
...	

Pos L/E	num nodo-i	Perm.	Cont. Refs.
...			
456	98	rw	2
3248	98	r	1



# Contenido

- Ficheros
- Directorios
- API del Sistema Operativo
  - Ficheros
  - Directorios
- Aumento de Prestaciones



# Concepto de directorio

- Objeto que relaciona de forma única un nombre de usuario de archivo con su descriptor interno
- Organizan y proporcionan información sobre la estructuración de los sistemas de archivos
- Una colección de nodos que contienen información acerca de los archivos



# Organización del directorio

- Eficiencia: localizar un archivo rápidamente
- Nombrado: conveniente y sencillo para los usuarios
  - Dos usuarios pueden tener el mismo nombre para archivos distintos
  - Los mismos archivos pueden tener nombres distintos
  - Nombres de longitud variable
- Agrupación: agrupación lógica de los archivos según sus propiedades (por ejemplo: programas Pascal, juegos, etc.)
- Estructurado: operaciones claramente definidas y ocultación
- Sencillez: la entrada de directorio debe ser lo más sencilla posible

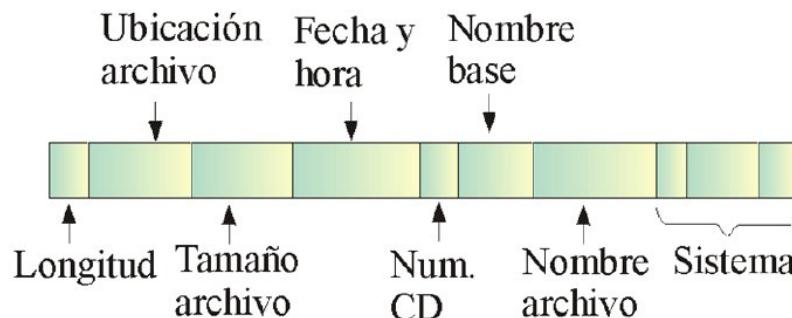


# Estructura de los directorios

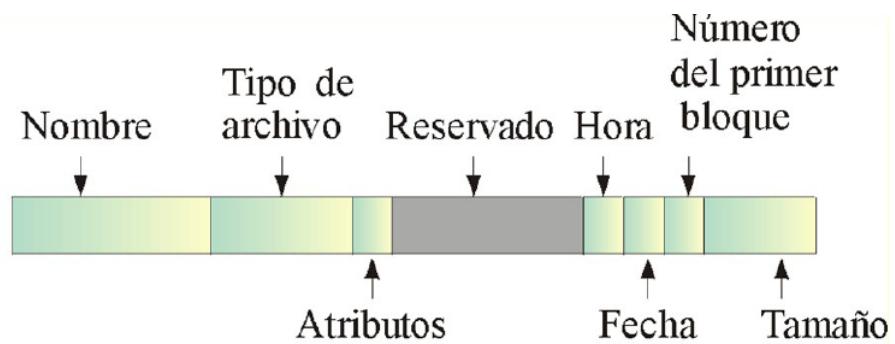
- Tanto la estructura del directorio como los archivos residen en discos
- Los directorios se suelen implementar como archivos con formato conocido para el SO
- Hay estructuras de directorio muy distintas. La información contenida en el directorio depende de esa estructura. Dos alternativas principales:
  - Almacenar atributos de archivo en entrada directorio (CD-ROM, FAT)
  - Almacenar <nombre, identificador>, con datos archivo en una estructura distinta (UNIX)



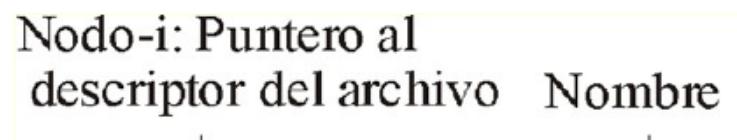
# Contenido de los directorios



Directorio de ISO-9660 (CD-ROM)



Directorio de MS-DOS

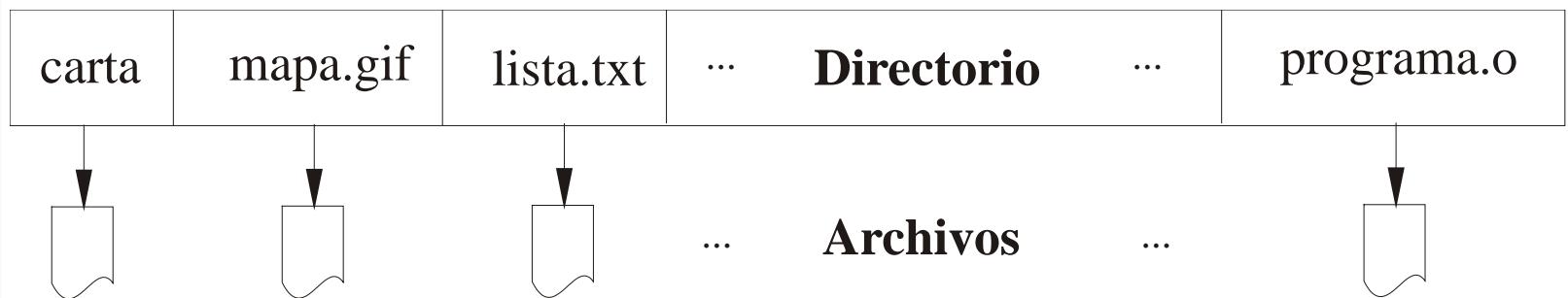


Directorio de UNIX SV



# Directorio de un nivel

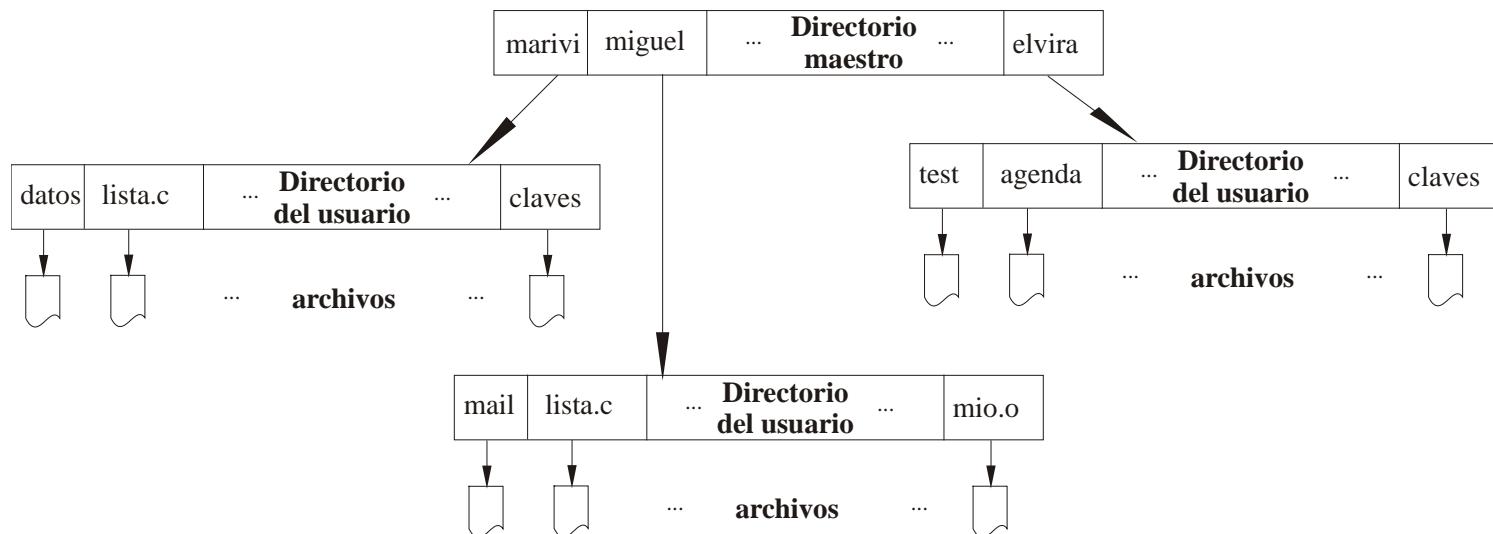
- Un único directorio para todos los usuarios
- Problemas de nombrado y agrupación

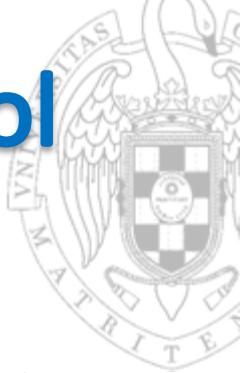




# Directorio de dos niveles

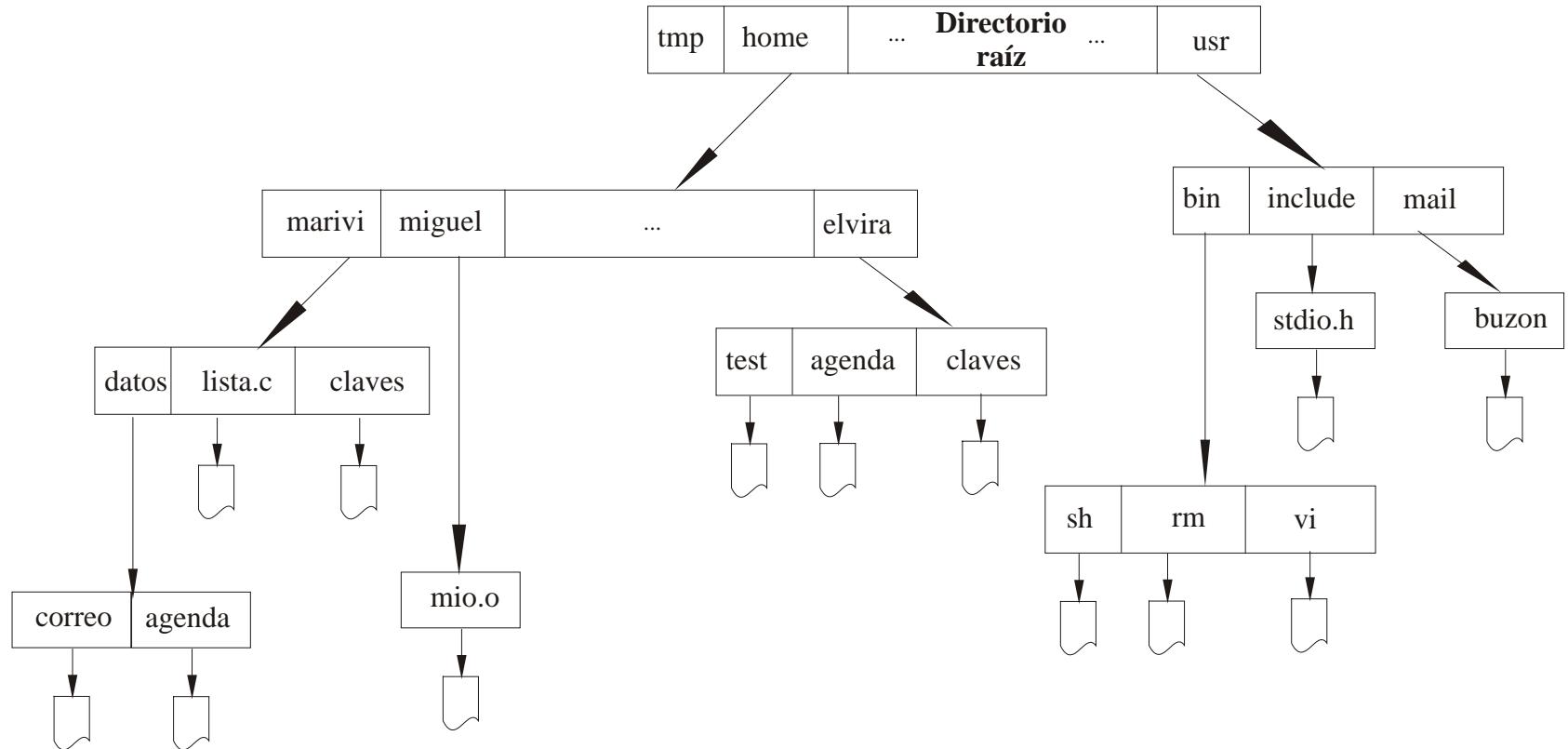
- Un directorio por cada usuario
- Camino de acceso automático o manual
- El mismo nombre de archivo para varios usuarios
- Búsqueda eficiente, pero problemas de agrupación





# Directorio con estructura de árbol

- Búsqueda eficiente y agrupación
- Nombres relativos y absolutos -> directorio de trabajo



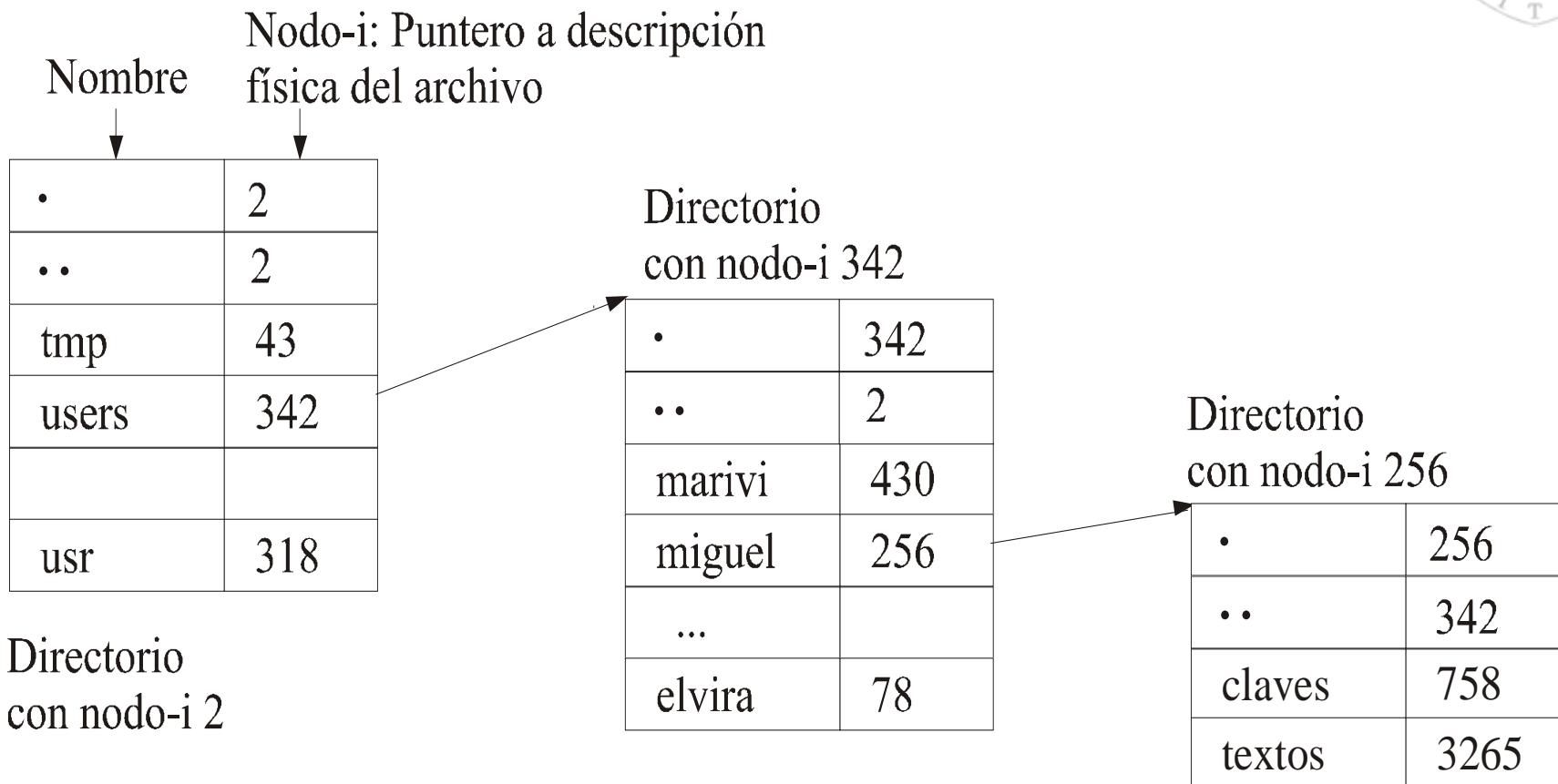


# Nombres jerárquicos

- Nombre absoluto: especificación del nombre respecto a la raíz (/ en LINUX, \ en Windows).
- Nombre relativo: especificación del nombre respecto a un directorio distinto del raíz
  - Ejemplo: (Estamos en /users/) miguel/claves
  - Relativos al dir. de trabajo o actual: aquel en el se está al indicar el nombre relativo. En Linux se obtiene con pwd
- Directorios especiales:
  - Directorio de trabajo ‘.’
    - Ejemplo: cp /users/miguel/claves .
  - Directorio *padre* ‘..’
    - Ejemplo: ls ..
  - Directorio HOME: el directorio base del usuario



# Interpretación de nombres en LINUX. I





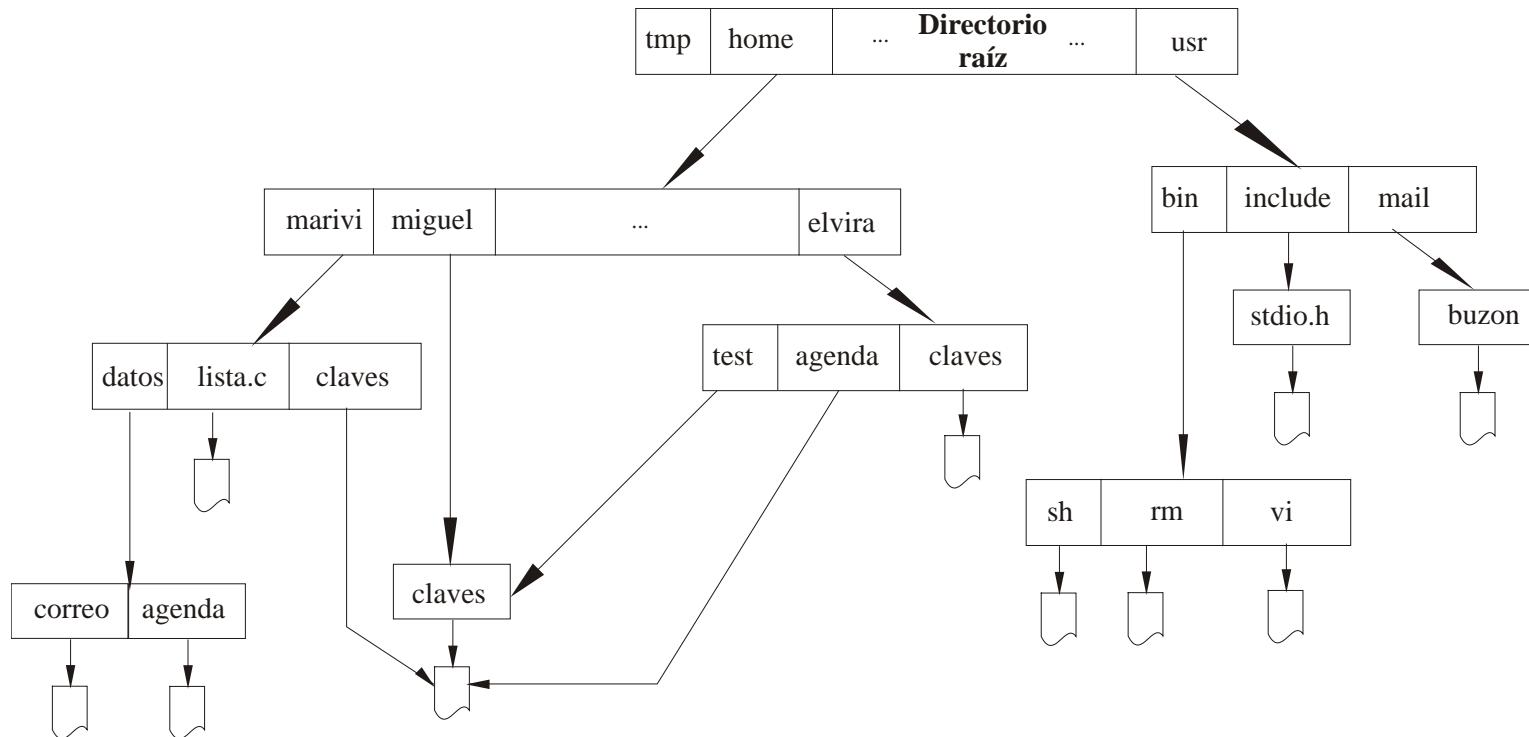
# Interpretación de nombres en LINUX. II

- Interpretar /users/miguel/claves
  - Traer a memoria entradas archivo con nodo-i 2
  - Se busca dentro users y se obtiene el nodo-i 342
  - Traer a memoria entradas archivo con nodo-i 342
  - Se busca dentro miguel y se obtiene el nodo-i 256
  - Traer a memoria entradas archivo con nodo-i 256
  - Se busca dentro claves y se obtiene el nodo-i 758
  - Se lee el nodo-i 758 y ya se tienen los datos del archivo
- ¿Cuándo parar?
  - Se ha encontrado el nodo-i del archivo
  - No se ha encontrado y no hay más subdirectorios
  - Estamos en un directorio y no contiene la siguiente componente del nombre (por ejemplo, miguel).



# Directorio de grafo acíclico I

- Tienen archivos y subdirectorios compartidos
- Este concepto no existe en FAT



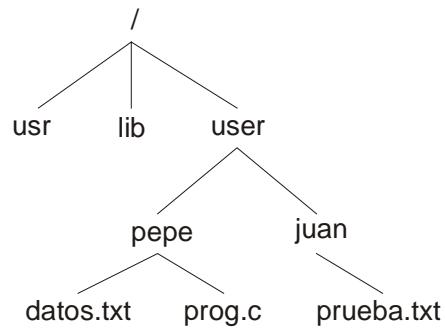


# Directorio de grafo acíclico II

- **link:** Un archivo con varios nombres -> control de enlaces
  - Un único archivo con contador enlaces en descriptor (e. físicos)
  - Archivos nuevos con el nombre destino dentro (e. simbólicos)
- Borrado de enlaces:
  - a) Decrementar contador; si 0 borrar archivo
  - b) Recorrer los enlaces y borrar todos
  - c) Borrar únicamente el enlace y dejar los demás
- Problema grave: existencia de bucles en el árbol. Soluciones:
  - Permitir sólo enlaces a archivos, no subdirectorios
  - Algoritmo de búsqueda de bucle cuando se hace un enlace
- Limitación de implementación en UNIX: sólo enlaces físicos dentro del mismo sistema de archivos.

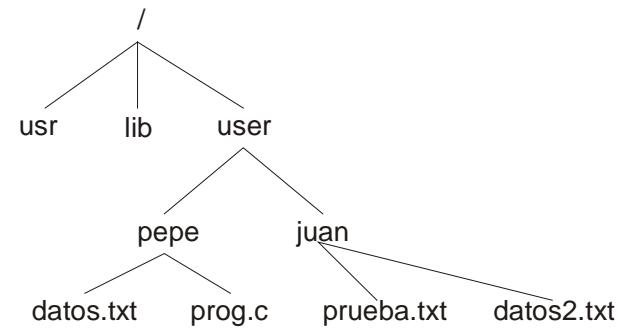


# Enlace simbólico



pepe		
.	23	
..	100	
datos.txt	28	
prog.c	400	

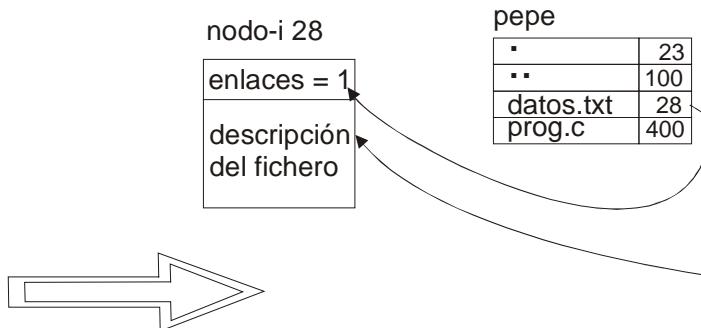
juan		
.	80	
..	100	
prueba.txt	60	



pepe		
.	23	
..	100	
datos.txt	28	
prog.c	400	

juan		
.	80	
..	100	
prueba.txt	60	
datos2.txt	130	

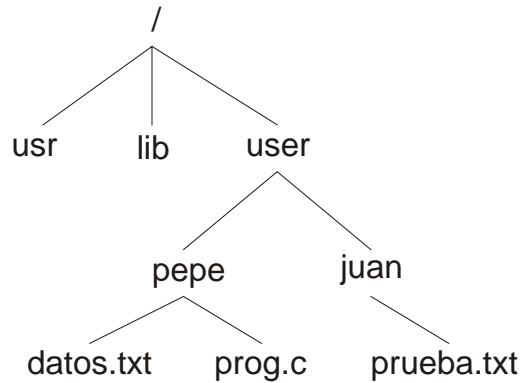
nodo-i 130		
enlaces = 1		



In -s /user/pepe/datos.txt /user/juan/datos2.txt

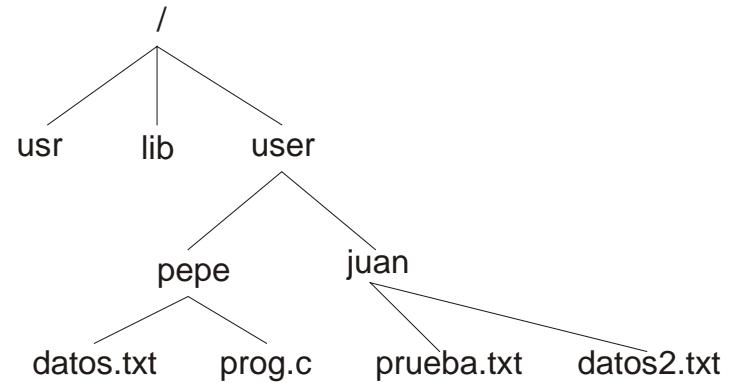


# Enlace físico



pepe		
.	23	
..	100	
datos.txt	28	
prog.c	400	

juan		
.	80	
..	100	
prueba.txt	60	



pepe		
.	23	
..	100	
datos.txt	28	
prog.c	400	

juan		
.	80	
..	100	
prueba.txt	60	
datos2.txt	28	

nodo-i 28  
enlaces = 2  
descripción del fichero

In /user/pepe/datos.txt /user/juan/datos2.txt



# Ejemplo

## ■ Enlace físico

```
$ echo "Hola" > archivo
```

```
$ ln archivo archivo_fis
```

```
$ stat archivo # ver "Links" (es 2)
```

```
$ stat archivo_fis #ver "Links" (es 2)
```

- Comparar campos "Inode" (son iguales)

## ■ Enlace simbólico

```
$ ln -s archivo archivo_sim
```

```
$ stat archivo # ver "Links" (no cambia)
```

```
$ stat archivo_sim # ver "Links" (es 1)
```

- Comparar campos "Inode" (son diferentes)



# Sistemas de archivos y particiones

- Volumen: conjunto coherente de metainformación y datos.
- Ejemplos de Sistemas de archivos:

## CD-ROM

Sistema	DVP	Tabla de localización	Directorios	Archivos
---------	-----	-----------------------	-------------	----------

## MS-DOS

Boot	Dos copias de la FAT	Directorio Raíz		Datos y Directorios	
------	----------------------	-----------------	--	---------------------	--

## UNIX

Boot	Super Bloque	Mapas de bits	nodos-i		Datos y Directorios		
------	--------------	---------------	---------	--	---------------------	--	--



# Sistemas de Ficheros

- El sistema de ficheros permite organizar la información dentro de los dispositivos de almacenamiento secundario en un formato intelible 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.
- Una vez creadas las particiones, el SO debe crear las estructuras de los SF dentro de esas particiones. Para ello se proporcionan mandatos como `format` o `mkfs` al usuario.

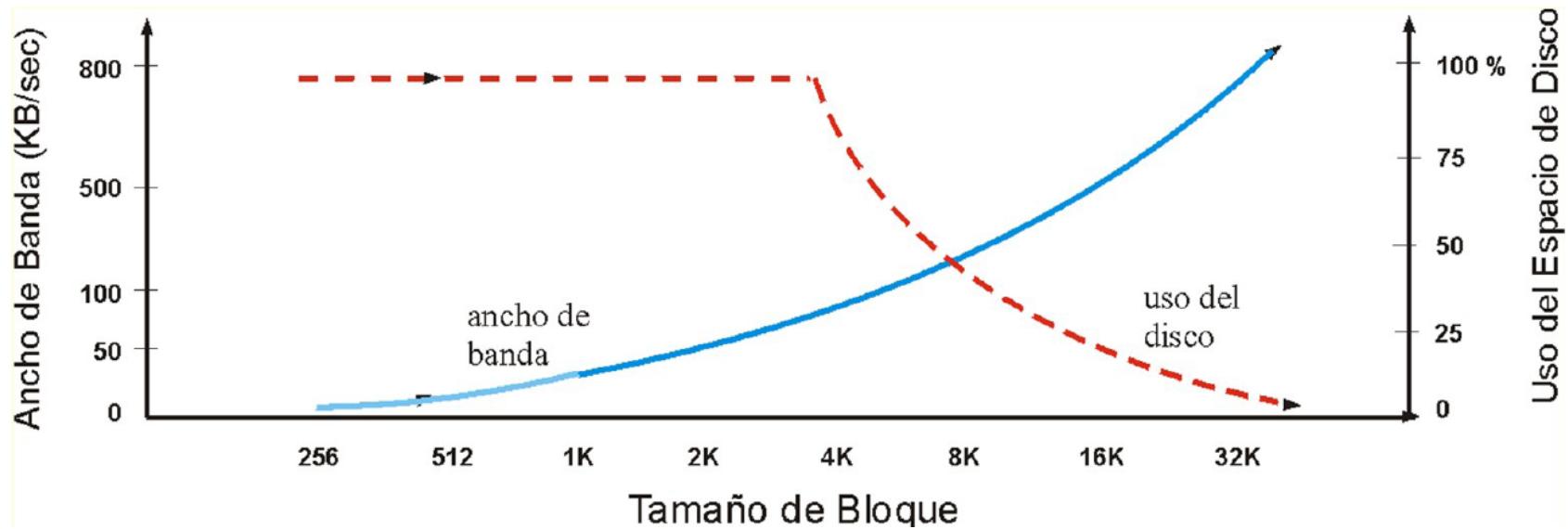


# Bloques y agrupaciones

- **Bloque:** *agrupación lógica de sectores de disco y es la unidad de transferencia mínima que usa el sistema de ficheros.*
  - Optimizar la eficiencia de la entrada/salida de los dispositivos secundarios de almacenamiento.
  - Todos los sistemas operativos proporcionan un tamaño de bloque por defecto.
  - Los usuarios pueden definir el tamaño de bloque a usar dentro de un sistema de archivos mediante el mandato `mkfs`.
- **Agrupación (o Cluster):** *conjunto de bloques que se gestionan como una unidad lógica de gestión del almacenamiento.*
  - El problema que introducen las agrupaciones, y los bloques grandes, es la existencia de fragmentación interna.



# Tamaño de bloque vs BW y DU





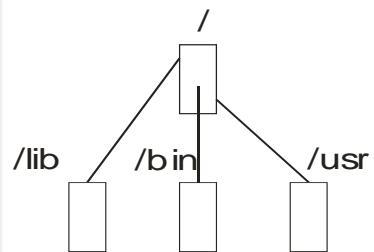
# Jerarquía de directorios

- ¿Árbol único de directorios?
  - Por dispositivo lógico en Windows (c:\users\miguel\claves, j:\pepe\tmp, ...)
  - Para todo el sistema en UNIX (/users/miguel/claves, /pepe/tmp, ...).
- Hacen falta servicios para construir la jerarquía: mount y umount.
  - mount /dev/hda /users
  - umount /users
- Ventajas: imagen única del sistema y ocultan el tipo de dispositivo
- Desventajas: complican la traducción de nombres, problemas para enlaces físicos entre archivos

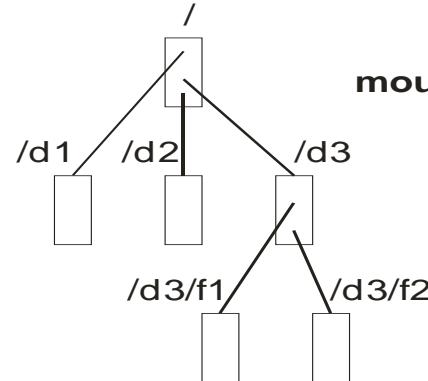
# Montado de Sistemas de archivos o particiones



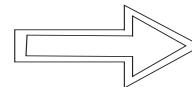
Volumen raíz  
(/dev/hd0)



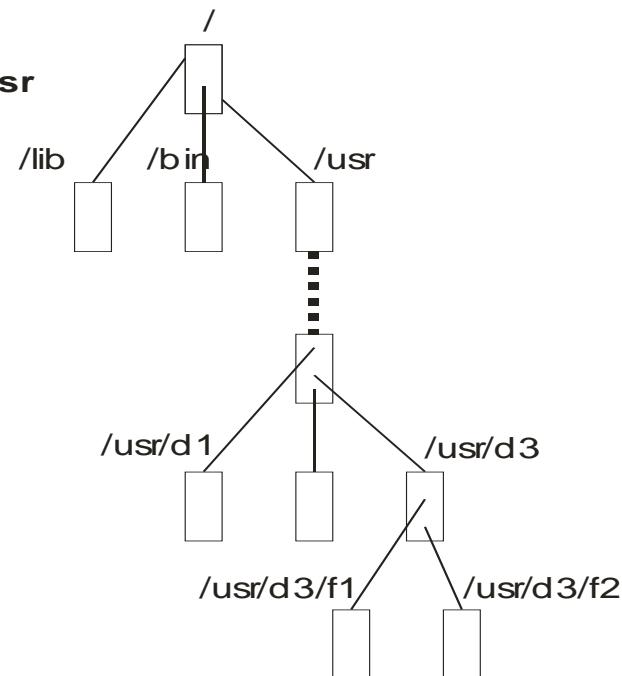
Volumen sin montar  
(/dev/hd1)



`mount /dev/hd1 /usr`



Volumen montado





# Ejemplo

- Creamos el archivo vacío (1.2e5 bloques)  
# dd if=/dev/zero of=/tmp/disk.img count=120000
- Creamos el sistema de ficheros sobre el archivo  
# mkfs -t ext3 -b 1024 /tmp/disk.img
- Montamos el sistema de ficheros  
# mkdir disk  
# mount -t ext3 -o loop /tmp/disk.img disk  
# mount
- Copiamos un archivo existente  
# cp archivo.txt disk/  
# ls disk
- Desmontamos el sistema de ficheros  
# umount disk  
# ls disk  
# rm /tmp/disk.img



# Contenido

- Ficheros
- Directorios
- API del Sistema Operativo
  - Ficheros
  - Directorios
- Aumento de Prestaciones



# Operaciones genéricas sobre archivos

- **creat:** crea un archivo con un nombre y protección y devuelve un descriptor
- **unlink:** borra el archivo con un nombre
- **open:** abre un archivo con nombre para realizar operaciones sobre el y devuelve un descriptor
- **close:** cierra un archivo abierto con un descriptor
- **read:** lee datos de un archivo abierto, usando su descriptor, a un almacén en memoria
- **write:** escribe datos a un archivo abierto, usando su descriptor, desde un almacén en memoria
- **Iseek:** mueve el apuntador a relativo al desplazamiento
- **stat:** devuelve los atributos de un fichero



# Servicios POSIX para archivos

- Visión lógica: tira secuencial de bytes
- Apuntador de posición a partir del cual se efectúan las operaciones
- Descriptores de archivos: enteros de 0 a 64K
- Predefinidos (describir programas independientes de dispositivos):
  - 0: entrada estándar
  - 1: salida estándar
  - 2: salida de error
- Fork: duplicación de BCP, pero compartición de tabla de archivos.
- Servicios consulta y modificación atributos.
- Protección:

dueño	grupo	mundo
rwx	rwx	rwx
- Ejemplos: 755 indica rwxr-xr-x



# Crear un archivo

- Servicio:

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

- Argumentos:

- name Nombre de archivo
- mode Bits de permiso para el archivo

- Devuelve:

- Devuelve un descriptor de archivo ó -1 si error.

- Ejemplos:

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



# Borrar un archivo

- Servicio:

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

- Argumentos:

- path nombre del archivo

- Devuelve:

- 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 archivo y el archivo deja de ser accesible.



# Abrir un archivo

- Servicio:

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

- Argumentos:

- name puntero al nombre del archivo
- flags opciones de apertura:
  - O\_RDONLY Sólo lectura
  - O\_WRONLY Sólo escritura
  - O\_RDWR Lectura y escritura
  - O\_APPEND El puntero de acceso se desplaza al final del archivo
  - O\_CREAT Si existe no tiene efecto. Si no existe lo crea
  - O\_TRUNC Trunca si se abre para escritura

- Devuelve:

- Un descriptor de archivo ó -1 si hay error.



# Cerrar un descriptor de archivo

- Servicio:

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

- Argumentos:

- fd descriptor de archivo

- Devuelve:

- Cero ó -1 si error.

- Descripción:

- El proceso pierde la asociación a un archivo.



# Leer de un archivo

## ■ Servicio:

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

## ■ Argumentos:

- fd descriptor de archivo
- buf zona donde almacenar los datos
- n\_bytes número de bytes a leer

## ■ Devuelve:

- Número de bytes realmente leídos ó -1 si error

## ■ Descripción:

- Transfiere n\_bytes.
- Puede leer menos datos de los solicitados si se rebasa el fin de archivo o se interrumpe por una señal.
- Después de la lectura se incrementa el puntero del archivo con el número de bytes realmente transferidos.



# Escribir en un archivo

## ■ Servicio:

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

## ■ Argumentos:

- fd descriptor de archivo
- buf zona de datos a escribir
- n\_bytes número de bytes a escribir

## ■ Devuelve:

- Número de bytes realmente escritos ó -1 si error

## ■ Descripción:

- Transfiere n\_bytes.
- Puede escribir menos datos de los solicitados si se rebasa el tamaño máximo de un archivo o se interrumpe por una señal.
- Después de la escritura se incrementa el puntero del archivo con el número de bytes realmente transferidos.
- Si se rebasa el fin de archivo el archivo aumenta de tamaño.



# Modificar el puntero de posición

- Servicio:

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

- Argumentos:

- fd Descriptor de archivo
- offset desplazamiento
- whence base del desplazamiento

- Devuelve:

- La nueva posición del puntero ó -1 si error.

- Descripción:

- Coloca el puntero de acceso asociado a fd
- La nueva posición se calcula:
  - SEEK\_SET posición = offset
  - SEEK\_CUR posición = posición actual + offset
  - SEEK\_END posición = tamaño del archivo + offset

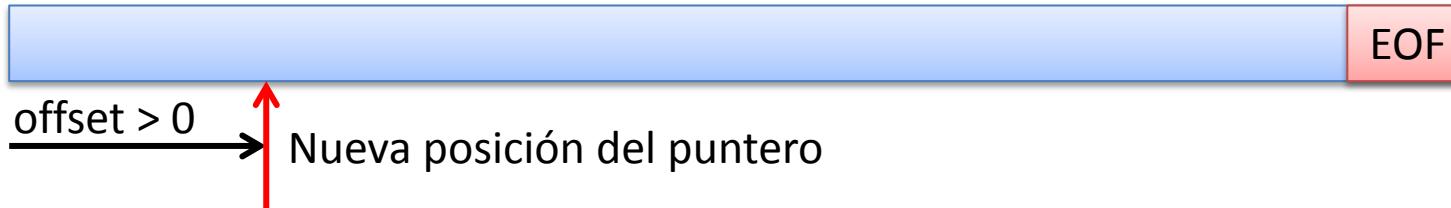


# Modificar el puntero de posición

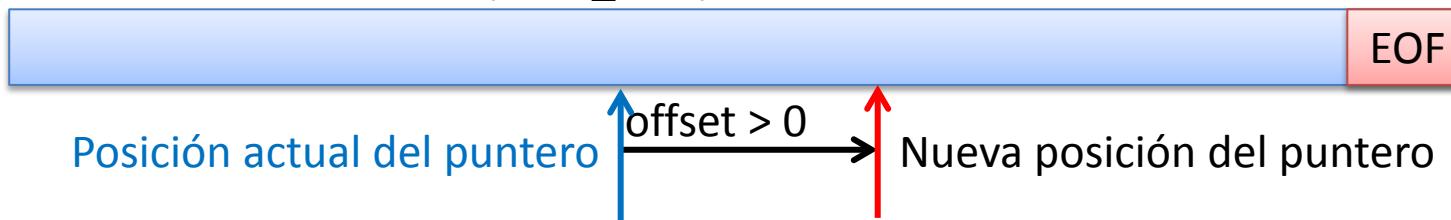
```
off_t lseek(int fd,  
           off_t offset,  
           int whence)
```

*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.*

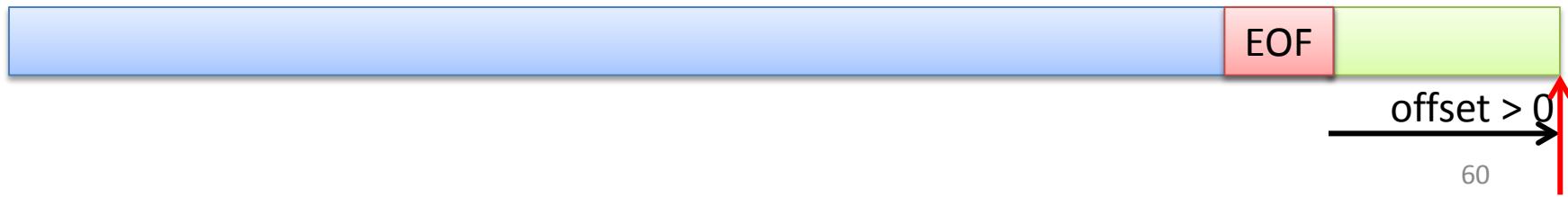
- Caso 1: whence == 0 (SEEK\_SET)



- Caso 2: whence == 1 (SEEK\_CUR)



- Caso 3: whence == 2 (SEEK\_END)





# Obtiene información sobre un archivo

- Servicio:

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

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

- Argumentos:

- name nombre del archivo

- fd descriptor de archivo

- buf puntero a un objeto de tipo struct stat donde se almacenará la información del archivo.

- Devuelve:

- Cero ó -1 si error



# Ejemplo. Copia un archivo en otro

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define BUFSIZE          512

main(int argc, char **argv) {
    int fd_ent, fd_sal, n_read;
    char buffer[BUFSIZE];

    fd_ent = open(argv[1], O_RDONLY); /* abre el archivo de entrada */
    if (fd_ent < 0){
        perror("open");
        exit(-1);
    }

    fd_sal = creat(argv[2], 0644);      /* crea el archivo de salida */
    if (fd_sal < 0){
        close(fd_ent);
        perror("creat");
        exit(-1);
    }

    while ((n_read = read(fd_ent, buffer, BUFSIZE)) > 0)
        write(fd_sal, buffer, n_read);
}
```



# Ejemplo. Copia (II)

```
/* bucle de lectura del archivo de entrada */
while ((n_read = read(fd_ent, buffer, BUFSIZE)) > 0)  {
    /* escribir el buffer al archivo de salida */
    if (write(fd_sal, buffer, n_read) < n_read)  {
        perror("write");
        close(fd_ent); close(fd_sal);
        exit(-1);
    }
    if (n_read < 0)  {
        perror("read");
        close(fd_ent); close(fd_sal);
        exit(-1);
    }
    close(fd_ent);  close(fd_sal);
    exit(0);
}
```



# Servicios POSIX para directorios

- Visión lógica: tabla de entradas nombre lógico, nodo-i
- Cada entrada tiene la siguiente estructura:

```
Struct dirent {  
    char *d_name;          /* nombre archivo */  
    ...  
}
```

- Gestión complicada porque los nombres de archivo tienen longitud variable.
- Servicios POSIX: gestión de directorios y de la tabla



# Operaciones genéricas 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
- **closedir:** cierra un directorio abierto con un descriptor
- **readdir:** lee la siguiente entrada del directorio
- **rewinddir:** sitúa el puntero de posición en la primera entrada
- **link/symlink:** crea una nueva entrada en el directorio para un enlace físico o lógico
- **unlink:** elimina una entrada del directorio
- **chdir:** cambia el directorio actual
- **getcwd:** obtener el directorio actual
- **rename:** cambiar el nombre de una entrada del directorio



# Crear un directorio

- Servicio:

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

- Argumentos:

- name nombre del directorio
  - mode bits de protección

- Devuelve:

- Cero ó -1 si error

- Descripción:

- Crea un directorio de nombre name.
  - UID\_dueño = UID\_efectivo
  - GID\_dueño = GID\_efectivo



# Borrar un directorio

- Servicio:

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

- Argumentos:

- name nombre del directorio

- Devuelve:

- Cero ó -1 si error

- Descripción:

- Borra el directorio si está vacío.
  - Si el directorio no está vacío no se borra.



# Abrir un directorio

- Servicio:

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

- Argumentos:

- dirname puntero al nombre del directorio

- Devuelve:

- Un puntero para utilizarse en readdir( ) o closedir( ).
  - NULL si hubo error.

- Descripción:

- Abre un directorio como una secuencia de entradas. Se coloca en el primer elemento.



# Cerrar un directorio

- Servicio:

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

- Argumentos:

- dirp puntero devuelto por `opendir()`.

- Devuelve:

- Cero ó -1 si error.

- Descripción:

- Cierra la asociación entre dirp y la secuencia de entradas de directorio.



# Leer entradas de directorio

- Servicio:

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

- Argumentos:

- dirp puntero returned por `opendir()`.

- Devuelve:

- 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.
  - Avanza el puntero a la siguiente entrada.
  - La estructura es dependiente de la implementación. Debería asumirse que tan solo se obtiene un miembro: `char *d_name`.



# Situar el puntero de directorio

- Servicio:

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

- Argumentos:

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



# Crea una entrada de directorio

## ■ Servicio:

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

## ■ Argumentos:

- existing nombre del archivo existente.
- new nombre de la nueva entrada que será un

## ■ Devuelve:

- Cero ó -1 si error.

## ■ Descripción:

- Crea un nuevo enlace, físico o simbólico, para un archivo existente
- El sistema no registra cuál es el enlace original
- existing no debe ser el nombre de un directorio



# Elimina una entrada de directorio

- Servicio:

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

- Argumentos:

- name nombre de archivo

- Devuelve:

- Cero 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 archivo y el archivo deja de ser accesible.



# Cambiar el directorio actual

- Servicio:

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

- Argumentos:

- name nombre de un directorio

- Devuelve:

- Cero ó -1 si error

- Descripción:

- Modifica el directorio actual, aquel a partir del cual se forman los nombre relativos.



# Obtener el directorio actual

- Servicio:

```
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

- Devuelve:

- Puntero a buf o NULL si error.

- Descripción:

- Obtiene el nombre del directorio actual



# Cambia el nombre de un archivo

- Servicio:

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

- Argumentos:

- old nombre de un archivo existente
- new nuevo nombre del archivo

- Devuelve:

- Cero ó -1 si error

- Descripción:

- Cambia el nombre del archivo old. El nuevo nombre es new.



# Programa que lista un directorio

```
#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];

    /* imprime el directorio actual */
    getcwd(buf, MAX_BUF);
    printf("Directorio actual: %s\n", buf);
```



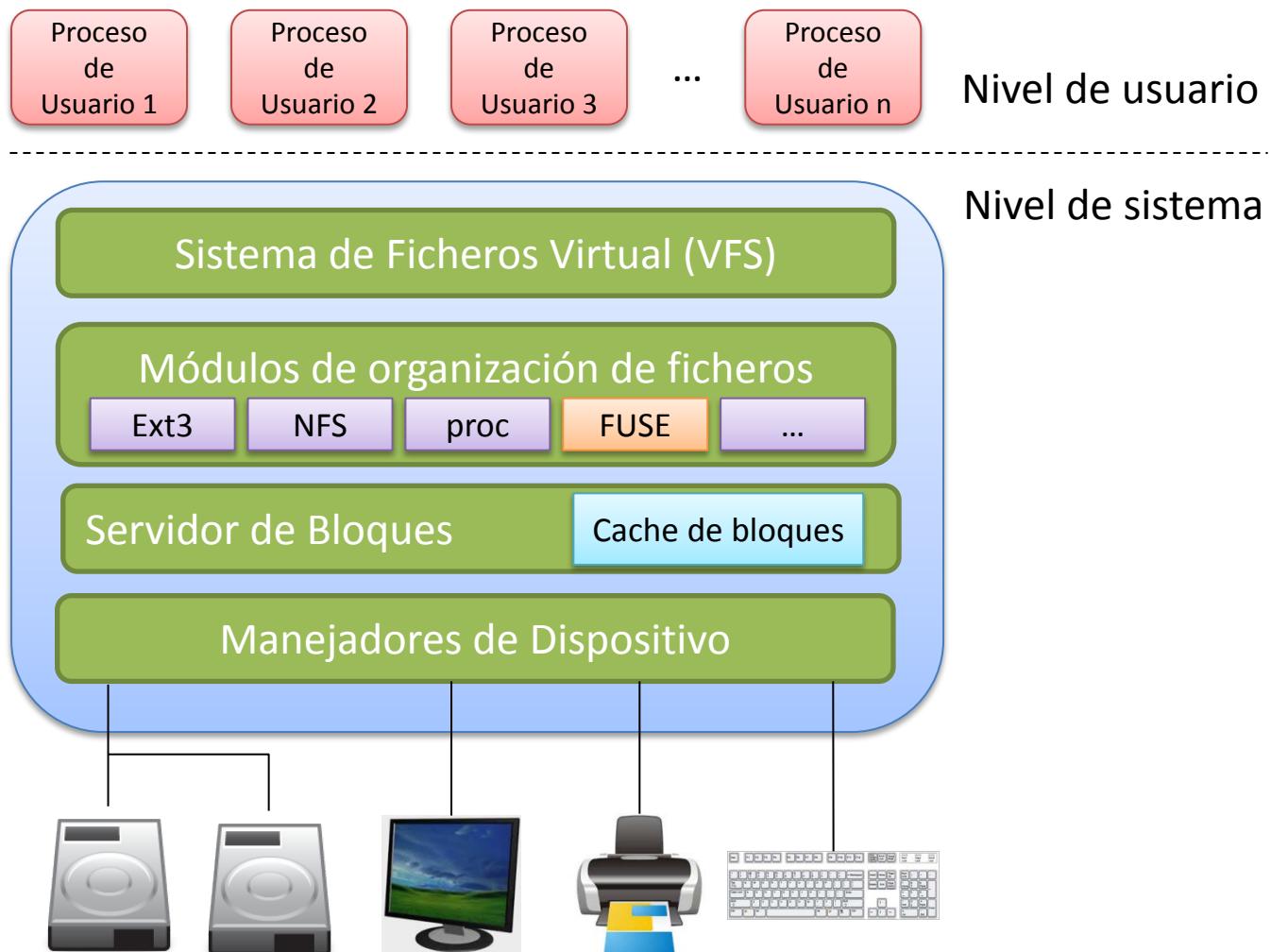
# Programa que lista un directorio (II)

```
/* abre el directorio pasado como argumento */
dirp = opendir(argv[1]);

if (dirp == NULL)  {
    fprintf(stderr,"No puedo abrir %s\n", argv[1]);
} else {
    /* lee entrada a entrada */
    while ( (dp = readdir(dirp)) != NULL)
        printf("%s\n", dp->d_name);
    closedir(dirp);
}
exit(0);
}
```



# Estructura del servidor de ficheros





# Módulo de organización de ficheros

- Proporciona el modelo del fichero del sistema operativo y los servicios de ficheros.
- Relaciona la imagen lógica del fichero con su imagen física, proporcionando algoritmos para trasladar direcciones lógicas de bloques a sus correspondientes direcciones físicas.
- Gestiona 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 registros de Windows NT).
- Un módulo de este estilo por cada tipo de fichero soportado (UNIX, AFS, Windows NT, MS-DOS, EFS, MINIX, etc.).
- Dentro de este nivel también se proporcionan servicios para *pseudo-ficheros*, tales como los del sistema de ficheros *proc*.
- Las llamadas de gestión de ficheros y de directorios particulares de cada sistema de ficheros se resuelven en el módulo de organización de ficheros. Para ello, se usa la información existente en el *i-nodo* del fichero afectado por las operaciones.



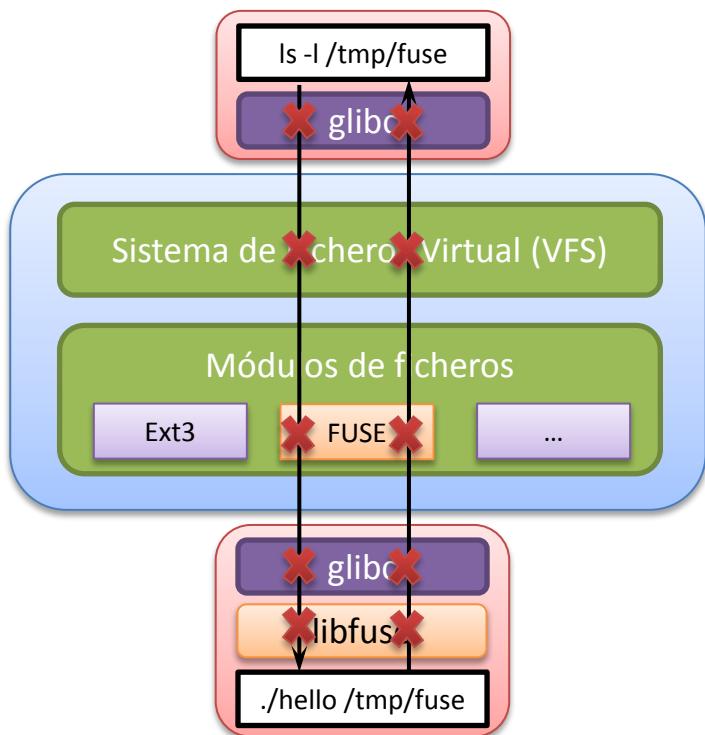
# Ejemplo - FUSE

- RECUERDA: # mount -t ext3 /dev/hdb2 /mnt/unidad



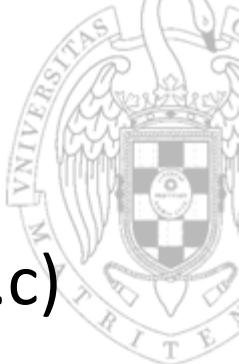
A partir de ahora, cada acceso que se realiza sobre /mnt/unidad, VFS lo redirige al módulo controlador del sistema ext3

- FUSE significa Filesystem in USErspace.



FUSE funciona exactamente de la misma manera.

- 1) Módulo para el kernel que se registra como un manejador de un sistema de archivos cualquiera.
  - 2) Cuando queremos acceder a una unidad FUSE montada, el VFS le redirige la llamada al módulo de FUSE, y FUSE nos redirige la llamada a nuestro programa controlador
- ➔ Es una capa entre el Kernel y nuestro programa en el espacio de usuario.



# Ejemplo - FUSE

- ¿Cómo se usa? → Creamos controlador (archivo .c)
  1. Hay que incluir `fuse.h` y enlazar con `libfuse`
  2. Declarar estructura llamada **`fuse_operations`**
    - Contiene punteros a funciones que serán llamados por cada operación
  3. Se termina el programa con la llamada a `fuse_main`
- Normas generales
  - Las funciones deben devolver 0 en caso de éxito y un número negativo indicando el error en caso de fallo.
    - Excepción: Las llamadas a write/read deben devolver un número positivo indicando los bytes leídos, 0 en caso de EOF o número negativo en caso de error.



# Ejemplo - FUSE

## ■ Miembros básicos de la estructura **fuse\_operations**:

```
$ more /usr/include/fuse/fuse.h
```

```
int (*getattr) (const char *, struct stat *);
```

- Función llamada cuando se quieren obtener los atributos de un archivo, p.e. cuando se le hace un “stat” a un archivo se llama a esta función. El primer parámetro indica la ruta del archivo. El segundo parámetro es la estructura stat a llenar.

```
int (*open) (const char *, struct fuse_file_info *);
```

- Se llama al abrir un archivo. El primer parámetro es la ruta del archivo, el segundo parámetro contiene información acerca de los flags de apertura. Permite devolver un handler, pero no es un parámetro necesario para un sistema de archivos sencillo y se puede ignorar. Lo único que hay que hacer es comprobar si se puede abrir el archivo, en caso afirmativo se devuelve cero.

```
int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);
```

- Se llama al leer un archivo. El primer parámetro es la ruta del archivo, el segundo es el buffer donde almacenar los datos, el tercer parámetro es la cantidad de bytes a leer, el cuarto el desplazamiento y el quinto es el mismo que el de open. Se devuelve la cantidad de bytes leídos.

```
int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t, struct fuse_file_info *);
```

- Se utiliza para leer un directorio. El primer parámetro es la ruta del directorio, el segundo una estructura que hay que llenar, el tercero es una función usada para llenar la estructura del segundo parámetro y los otros dos se pueden ignorar para ejemplos sencillos.



# Ejemplo - FUSE

- Más miembros de la estructura **fuse\_operations**:

```
int (*mknod) (const char *, mode_t, dev_t);  
int (*unlink) (const char *);  
int (*rename) (const char *, const char *);  
int (*truncate) (const char *, off_t);  
int (*write) (const char *, const char *, size_t, off_t, struct  
    fuse_file_info *);
```



# Ejemplo - FUSE

## ■ Un ejemplo sencillo (1/4):

```
#include <fuse.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

static const char *hello_str = "Hello World!\n";
static const char *hello_path = "/hello";

static int hello_getattr(const char *path, struct stat *stbuf) {
    int res = 0;
    memset(stbuf, 0, sizeof(struct stat));
    if (strcmp(path, "/") == 0) {
        stbuf->st_mode = S_IFDIR | 0755;
        stbuf->st_nlink = 2;
    } else if (strcmp(path, hello_path) == 0) {
        stbuf->st_mode = S_IFREG | 0444;
        stbuf->st_nlink = 1;
        stbuf->st_size = strlen(hello_str);
    } else
        res = -ENOENT;
    return res;
}
```

El path que nos pasa FUSE siempre es un path "absoluto" pero referenciado al sistema de archivos.



# Ejemplo - FUSE

- Un ejemplo sencillo (2/4):

```
static int hello_open(const char *path, struct fuse_file_info *fi) {
    if (strcmp(path, hello_path) != 0)
        return -ENOENT;
    if ((fi->flags & 3) != O_RDONLY)
        return -EACCES;
    return 0;
}

static int hello_read(const char *path, char *buf, size_t size, off_t offset,
                     struct fuse_file_info *fi) {
    size_t len;
    (void) fi;
    if(strcmp(path, hello_path) != 0)
        return -ENOENT;
    len = strlen(hello_str);
    if (offset < len) {
        if (offset + size > len)
            size = len - offset;
        memcpy(buf, hello_str + offset, size);
    } else
        size = 0;
    return size;
}
```



# Ejemplo - FUSE

## ■ Un ejemplo sencillo (3/4):

```
static int hello_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                         off_t offset, struct fuse_file_info *fi) {
    (void) offset;
    (void) fi;
    if (strcmp(path, "/") != 0)
        return -ENOENT;
    filler(buf, ".", NULL, 0);
    filler(buf, "..", NULL, 0);
    filler(buf, hello_path + 1, NULL, 0);
    return 0;
}
static struct fuse_operations hello_oper = {
    .getattr = hello_getattr,
    .readdir = hello_readdir,
    .open     = hello_open,
    .read     = hello_read,
};

int main(int argc, char *argv[]) {
    return fuse_main(argc, argv, &hello_oper);
}
```



# Ejemplo - FUSE

- Un ejemplo sencillo (4/4):

```
$ gcc hello.c -o hello -lfuse -D_FILE_OFFSET_BITS=64 -DFUSE_USE_VERSION=22
$ mkdir /tmp/fuse
$ ./hello /tmp/fuse
$ ls -l /tmp/fuse
    total 0
    -r--r--r--  1 root root 13 Jan  1 1970 hello
$ cat /tmp/fuse/hello
Hello World!
$ fusermount -u /tmp/fuse
```



# Contenido

- Ficheros
- Directorios
- API del Sistema Operativo
  - Ficheros
  - Directorios
- Aumento de Prestaciones



# Memoria Caché de bloques

- El acceso a disco es órdenes de magnitud más lento que el acceso a RAM
- Se usa parte de la RAM como caché de bloques y nombres basándonos en:
  - Localidad espacial
  - Localidad temporal
- Problemática: consistencia de datos
- Algoritmos de búsqueda y reemplazamiento



# Políticas de reemplazo

- Comprobar si el bloque a leer está en la cache.
  - 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**.
  - Si el bloque ha sido escrito (sucio): política de escritura.
- Políticas:
  - **FIFO** (First In First Out)
  - **MRU** (Most Recently Used)
  - **LRU** (Least Recently Used)
    - Política más común, se reemplaza el bloque que lleva más tiempo sin ser referenciado.
    - Los bloques más usados se encuentran en RAM



# Políticas de escritura

- 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.
- 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 archivo, se vuelcan al disco los bloques del mismo que tienen datos actualizados.