



AMPLIACIÓN DE SISTEMAS OPERATIVOS Y REDES

Grado en Ingeniería Informática / Doble Grado

Universidad Complutense de Madrid

TEMA 2.2. Sistema de Ficheros

PROFESORES:

Rubén Santiago Montero

Eduardo Huedo Cuesta

Rafael Rodríguez Sánchez

Características de los Sistemas de Ficheros

Desde el punto de vista del usuario

- Colección de ficheros y directorios usados para guardar y organizar la información

Desde el punto de vista del sistema operativo

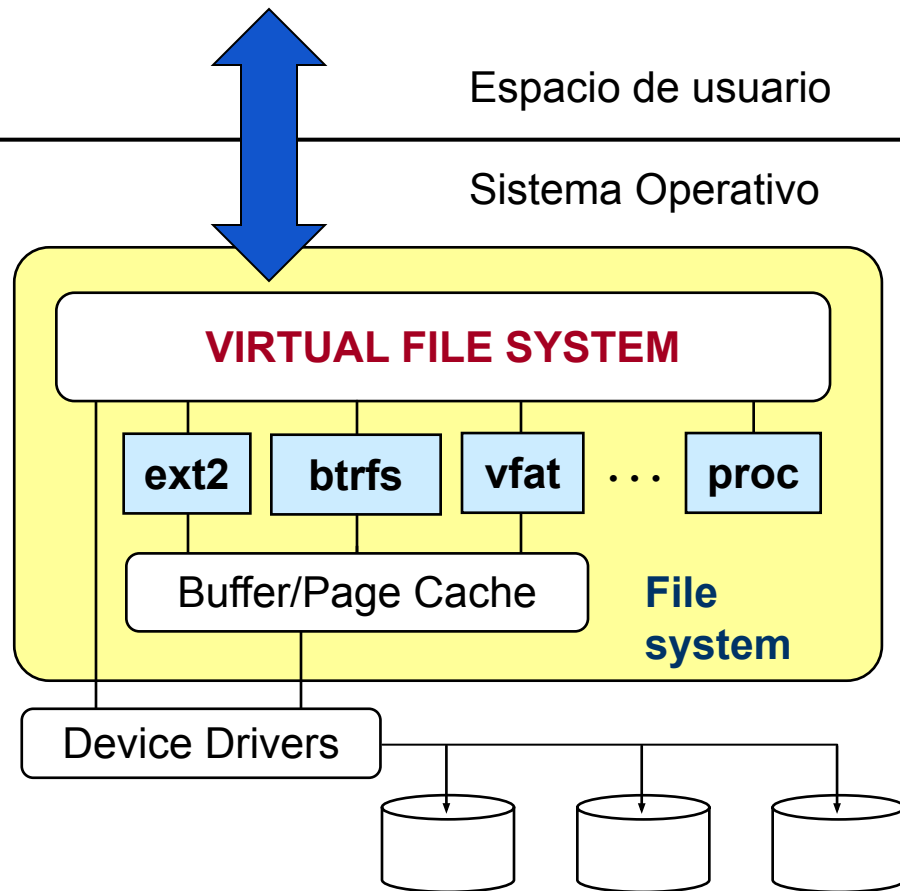
- Conjunto de tablas y estructuras que permiten gestionar los ficheros y directorios

Tipos de Sistemas de Ficheros:

- **Basados en disco:** Residen en soportes de almacenamiento físicos como discos duros magnéticos (HDD), discos ópticos o unidades de estado sólido (SSD)
 - Ejemplos: ext2-3-4, FAT, NTFS, ISO9660, UDF, UFS, HPFS, XFS, Btrfs, ZFS...
- **Basados en red (o distribuidos):** Se utilizan para acceder a sistemas de ficheros remotos independientemente del tipo
 - Ejemplos: NFS (Network File System) y SMB
- **Basados en memoria (o pseudo):** Residen en memoria principal mientras el sistema operativo se está ejecutando
 - Ejemplos: proc, tmpfs...

filesystems (5)

Estructura



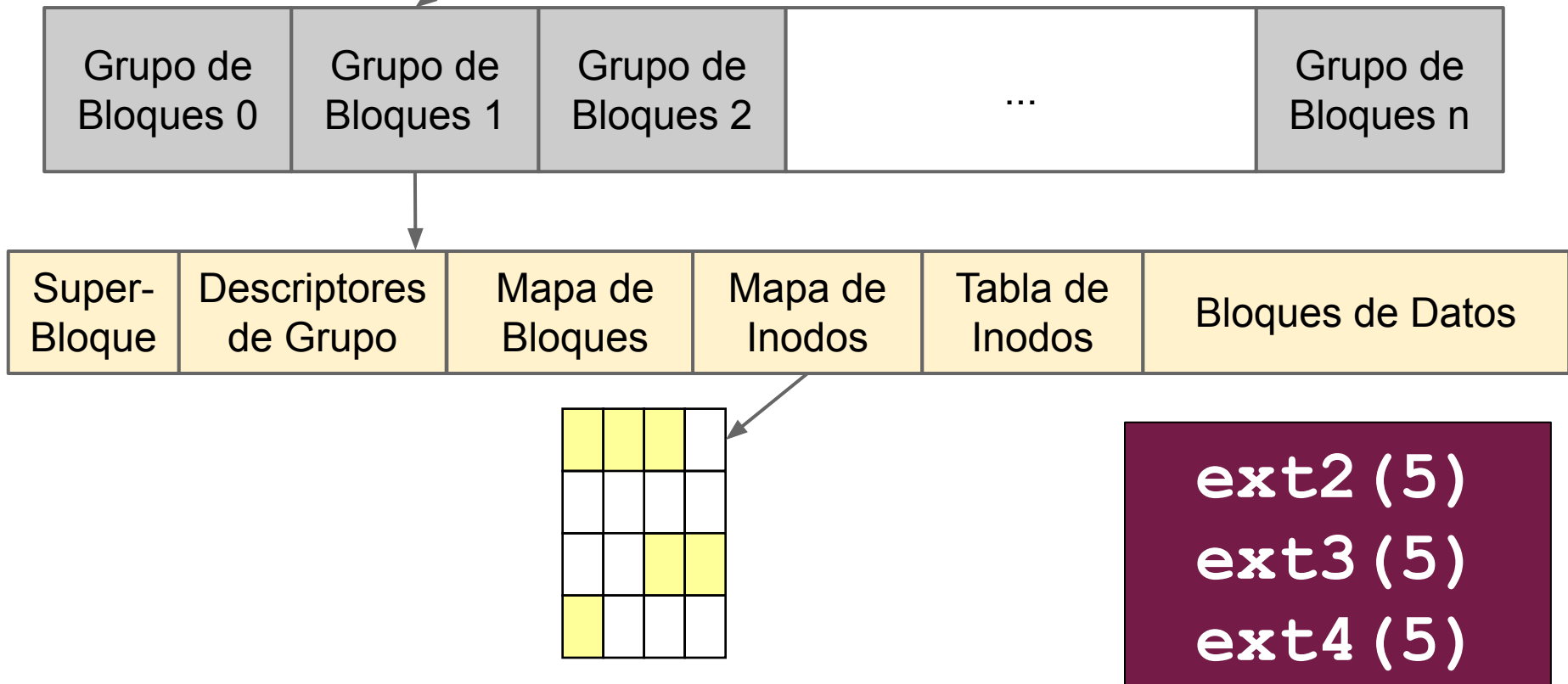
- La capa VFS establece un **enlace** bien definido **entre el kernel del SO** y los diferentes **sistemas de ficheros**
 - Proporciona las diferentes **llamadas** para la gestión de ficheros, **independientes del sistema de ficheros**
 - Permite acceder a múltiples sistemas de ficheros distintos
- Se **optimiza la entrada/salida** por medio de:
 - La *cache* de inodos y la *cache* de entradas de directorio (*dentry*) de VFS
 - La *cache* de *buffers*/páginas (*sync*)

Estructura: Grupos de bloques

Evolución del sistema de ficheros: Minix → ext → ext2 → ext3 → ...
Inspirado en el FFS (Fast File System) de BSD

Estructura de ext2:

- Bloques de datos cerca de sus inodos
- Los inodos cerca de su directorio

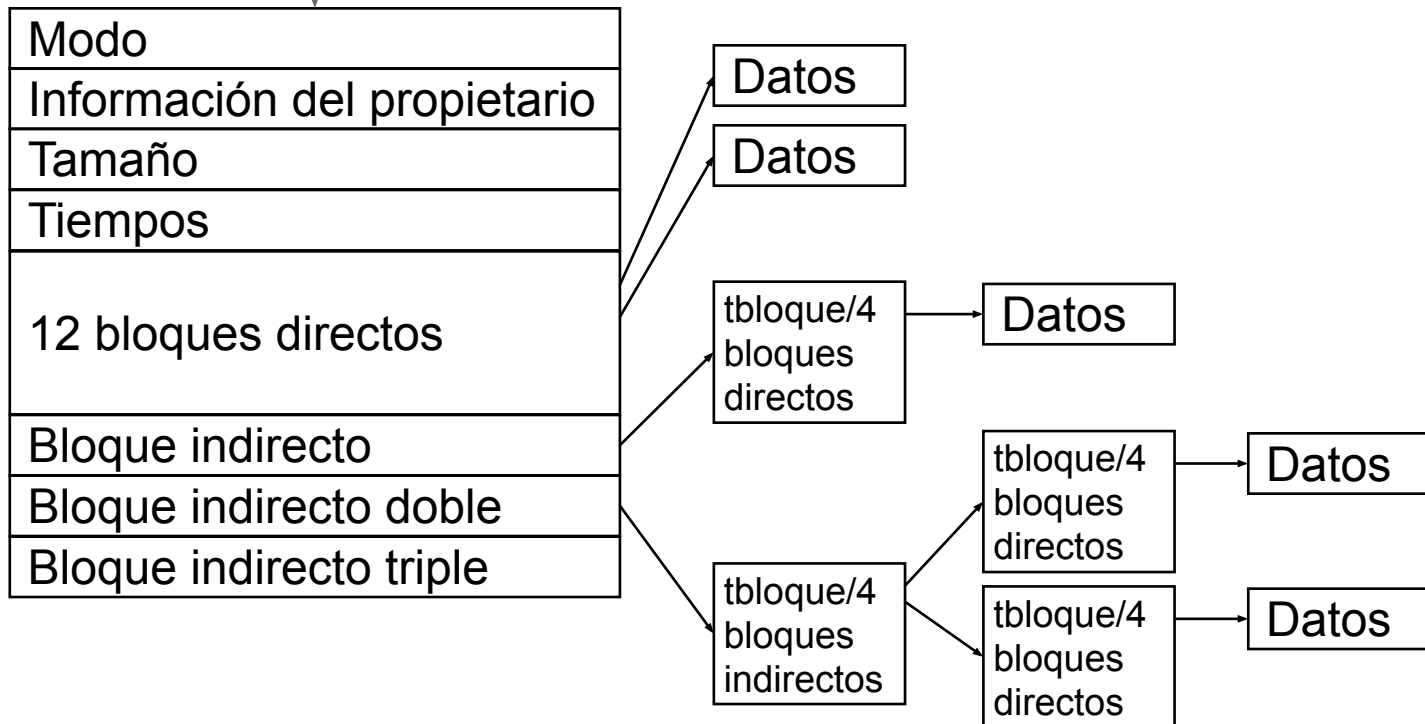


Estructura: Inodos

Directorio:

.	#555	..	#2	home	#345	vmlinux	#654
---	------	----	----	------	------	---------	------

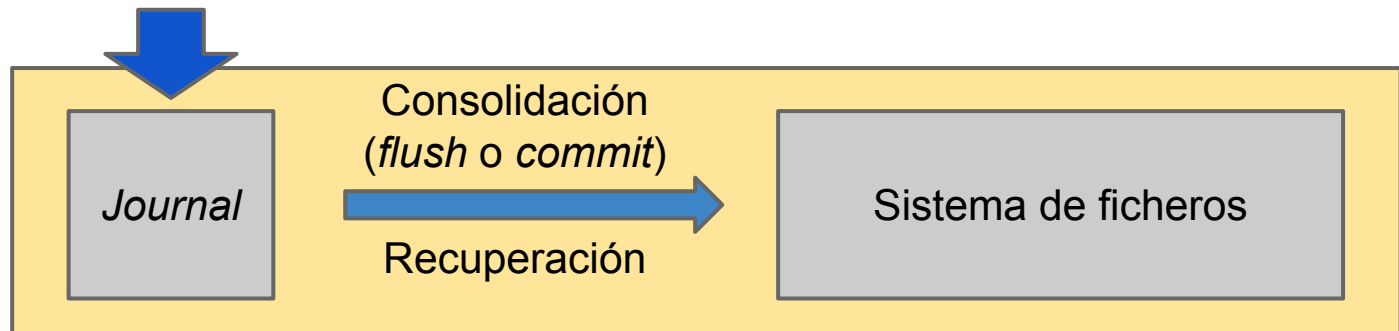
Inodo:



Journaling

- Cuando un sistema de ficheros tradicional no se apaga correctamente, el SO debe comprobar su integridad y consistencia en el siguiente arranque (utilidad `fsck`)
 - Implica recorrer toda la estructura en búsqueda de inodos huérfanos e inconsistencias, lo que puede llevar mucho tiempo en sistemas grandes
 - En ocasiones no es posible reparar automáticamente la estructura, y se debe hacer de manera manual
- Los sistemas de ficheros modernos (ReiserFS, XFS, ext3...) incorporan un fichero, región o dispositivo especial, denominado *log* o *journal*, para evitar la corrupción
 - Los metadatos (inodos, mapas...) se escriben primero en el *journal*
 - En caso de fallo o apagado brusco del sistema, se usa el *journal* para devolver el sistema de ficheros a un estado consistente
 - La consolidación se hace periódicamente o si el *journal* supera un tamaño

Escritura en el sistema de ficheros



Atributos de ficheros

- Obtener el estado de un fichero:

```
int stat(const char *file_name,  
         struct stat *buf);  
int lstat(const char *file_name,  
          struct stat *buf);  
int fstat(int fd, struct stat *buf);
```

<sys/types.h>
<sys/stat.h>

SV+BSD+POSIX

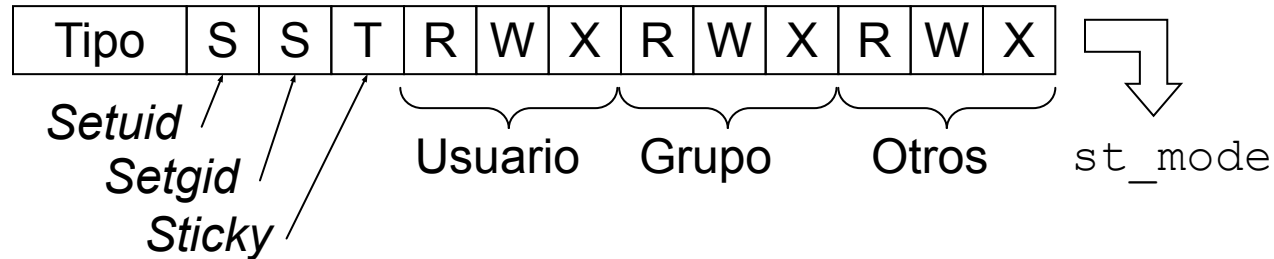
- `stat` sigue enlaces simbólicos, mientras que `lstat` no
- `fstat` recibe un descriptor de fichero abierto
- No se necesitan permisos sobre el fichero, pero sí para buscar en la ruta
- Errores:
 - **EBADF**: Descriptor no válido
 - **ENOENT**: Ruta incorrecta o nula
 - **ENOTDIR**: Componente de la ruta no es un directorio
 - **ELOOP**: Demasiados *links* en la búsqueda
 - **EFAULT**: Dirección no válida
 - **EACCES**: Permiso denegado
 - **ENAMETOOLONG**: Nombre de fichero muy largo
- El comando `stat` proporciona acceso a esta funcionalidad

Atributos de ficheros

```
struct stat {  
    dev_t st_dev;           /* Dispositivo que lo contiene */  
    ino_t st_ino;           /* Inodo */  
    mode_t st_mode;        /* Protección */  
    nlink_t st_nlink;       /* Número de enlaces rígidos */  
    uid_t st_uid;          /* UID del propietario */  
    gid_t st_gid;          /* GID del propietario */  
    dev_t st_rdev;         /* Dispositivo, si fich. especial */  
    off_t st_size;         /* Tamaño en bytes */  
    unsigned long st_blksize; /* Tamaño de bloque E/S */  
    unsigned long st_blocks;  /* Bloques asignados */  
    time_t st_atime;        /* Último acceso */  
    time_t st_mtime;        /* Última modificación */  
    time_t st_ctime;        /* Último cambio de estado */  
}
```

- `st_blksize`: Tamaño de bloque para E/S de sistema de ficheros eficiente
- `st_blocks`: Número de bloques de 512 bytes asignados al fichero
- `st_atime`: Último acceso a los datos (`read`, `execve`...)
- `st_mtime`: Última modificación de los datos (`write`, `mknod`...), no del inodo
- `st_ctime`: Último cambio en el inodo (propietario, permisos, tamaño...)

Atributos de ficheros



- Macros y *flags* (`sys/stat.h`) para comprobar el tipo de fichero y los permisos:

- Macros para **comprobar el tipo de fichero**:

`S_ISLNK(m)` : Comprueba si es un enlace simbólico

`S_ISREG(m)` : Comprueba si es un fichero normal

`S_ISDIR(m)` : Comprueba si es un directorio

`S_ISCHR(m)` : Comprueba si es un dispositivo por caracteres

`S_ISBLK(m)` : Comprueba si es un dispositivo por bloques

`S_ISFIFO(m)` : Comprueba si es un FIFO o pipe

`S_ISSOCK(m)` : Comprueba si es un socket

- Flags para comprobar permisos** (con operadores de bit `|` `&` `~` `^`):

`S_IRWXU`: Permisos para el usuario (00700)

`S_IRWXG`: Permisos para el grupo (00070)

`S_IRWXO`: Permisos para otros (00007)

$$S_I \begin{Bmatrix} R \\ W \\ X \end{Bmatrix} \begin{Bmatrix} \text{USR} \\ \text{GRP} \\ \text{OTH} \end{Bmatrix} : \begin{Bmatrix} \text{Lectura} \\ \text{Escritura} \\ \text{Ejecución} \end{Bmatrix} \text{ para } \begin{Bmatrix} \text{Usuario} \\ \text{Grupo} \\ \text{Otros} \end{Bmatrix}$$

Atributos de ficheros: Permisos

- Cambiar los permisos (no se puede cambiar el tipo):

```
int chmod(const char *path, mode_t mode);  
int fchmod(int fd, mode_t mode);
```

<sys/types.h>

<sys/stat.h>

SV+BSD+POSIX

- La modificación suele hacerse leyendo los permisos actuales y realizando operaciones lógicas (bit a bit) con los flags anteriores
- El EUID del proceso debe coincidir con el del propietario del fichero, o el proceso debe ser privilegiado
- Algunos errores:
 - **EIO**: Error de E/S
 - **ENOTDIR**: Elemento del PATH no existe
 - **ELOOP**: Demasiados *links* simbólicos
- El comando `chmod` proporciona acceso a esta funcionalidad

Atributos de ficheros: Permisos

- Comprobar el tipo de permisos sobre un fichero:

```
int access(const char *path, int mode);
```

<unistd.h>

SV+BSD+POSIX

- El modo es una combinación de los siguientes *flags*:
 - R_OK: El fichero existe y tenemos permisos de lectura
 - W_OK: El fichero existe y tenemos permisos de escritura
 - X_OK: El fichero existe y tenemos permisos de ejecución
 - F_OK: El fichero existe
- En la comprobación de los permisos se tiene en cuenta la ruta completa
- La comprobación se realiza con los identificadores de usuario y grupo reales, a diferencia de la escritura o lectura
- La llamada fallará si alguno de los permisos no se satisface

Creación y apertura de ficheros

- Abrir y/o crear un fichero o dispositivo:

```
int open(const char *path, int flags);  
int open(const char *path, int flags,  
         mode_t mode);
```

<code><sys/types.h></code> <code><sys/stat.h></code> <code><fcntl.h></code>

SV+BSD+POSIX

- `path` es la ruta del fichero o dispositivo
- `flag` debe indicar el modo de acceso y puede incluir otras opciones:
 - `O_RDONLY`: Acceso de solo lectura
 - `O_WRONLY`: Acceso de solo escritura
 - `O_RDWR`: Acceso de lectura y escritura
- `mode` indica los permisos a aplicar en caso de que se cree un nuevo fichero (con `O_CREAT`)
 - En octal (precedidos por 0 en C/C++) o como OR bit a bit de *flags*
 - Estos permisos se ven modificados por el `umask` del proceso
- Devuelve un descriptor de fichero con el puntero de acceso posicionado al principio del fichero, o -1 si ocurre un error (y establece `errno`)
 - El descriptor del fichero es el menor disponible en el sistema

Creación y apertura de ficheros

- Opciones adicionales en `flags`:
 - `O_CREAT`: Si el fichero no existe, lo crea
 - Con los permisos proporcionados en `mode` y, si no se proporciona, con un **valor arbitrario de la pila**
 - El UID y GID del nuevo fichero serán el EUID y EGID del proceso, pero si el directorio tiene el bit `setgid` activo, el GID será el GID del directorio
 - `O_EXCL`: Usado en combinación con `O_CREAT` provoca un error si el fichero existe (*Exclusively Create*)
 - `O_TRUNC`: El fichero es truncado a tamaño 0
 - `O_APPEND`: Antes de realizar cualquier escritura se posiciona el puntero de fichero a la última posición del fichero (puede corromper ficheros en NFS)
 - `O_NONBLOCK`: Abre en modo no bloqueante, en el que ni `open(2)` ni ninguna operación de E/S posterior sobre el fichero harán que el proceso espere
 - `O_SYNC`: Abre en modo síncrono, en el que las operaciones de escritura se bloquean hasta que los datos son físicamente escritos, evitando la pérdida de información en caso de fallo del sistema

Creación y apertura de ficheros: Permisos

- Establecer la máscara de permisos para creación de ficheros:

```
mode_t umask(mode_t mask);
```

- Establece el valor de la máscara (`umask`) del proceso a `mask & 0777`
- `umask` es usado por `open(2)`, `mkdir(2)` y otras llamadas que crean ficheros para modificar los permisos asignados a los ficheros o directorios creados
 - En concreto, los permisos indicados en `umask` son borrados del argumento `mode` con `mode & ~umask`, por ejemplo:

$$0666 \& \sim 0022 = 0644$$

- El valor por defecto de `umask` es `S_IWGRP | S_IWOTH (0022)`
 - Esta llamada siempre se ejecuta correctamente y devuelve la máscara anterior
- El comando interno de la *shell* `umask` proporciona acceso a esta funcionalidad

```
<sys/types.h>
```

```
<sys/stat.h>
```

```
SV+BSD+POSIX
```

Duplicación de descriptores

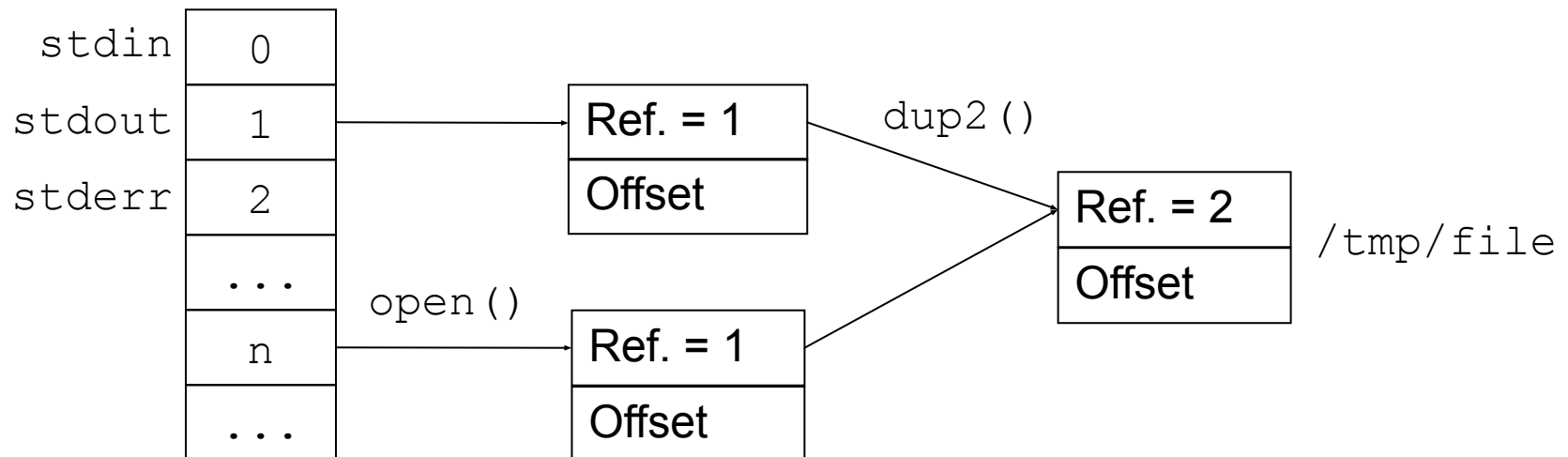
- Duplicar un descriptor:

```
int dup(int oldfd);  
int dup2(int oldfd, int newfd);
```

<unistd.h>

SV+BSD+POSIX

- Los dos descriptores se refieren al mismo fichero abierto, por lo que comparten cerrojos, punteros y opciones, de forma que puede intercambiarse su uso
- El descriptor devuelto por `dup()` es el menor disponible en el sistema
- Con `dup2()`, `newfd` referirá a `oldfd` y, si `newfd` está abierto, se cerrará
- Errores:
 - **EBADF**: `oldfd` no está abierto o `newfd` está fuera de rango
 - **EMFILE**: Número máximo de ficheros abiertos alcanzado



Lectura y escritura de ficheros

- Leer, escribir, posicionar y cerrar ficheros:

```
ssize_t write(int fd, void *buffer,
              size_t count);
ssize_t read(int fd, void *buffer, size_t count);
off_t lseek(int fd, off_t offset, int where);
int close(int fd);
```

<unistd.h>

SV+BSD+POSIX

- No mezclar estas llamadas al sistema con funciones de librería (ej. `fopen`, `fread`, `fwrite`... de `stdio.h`, o clases `fstream` en C++)
- La escritura de ficheros se realiza a través de la *Buffer/Page cache*, proporcionando un acceso eficiente
- Sincronizar un fichero:

```
int fsync(int fd);
```

 - La llamada se bloquea hasta que el dispositivo informa de que la transferencia se ha completado

Enlaces rígidos y simbólicos

- Crear un enlace rígido (*hard link*):

```
int link(const char *old, const char *new);
```

<unistd.h>

SV+BSD+POSIX

- Únicamente sobre ficheros en el mismo sistema de ficheros
- Si el nuevo fichero existe no será sobrescrito

- Crear un enlace simbólico (*soft link* o *symlink*):

```
int symlink(const char *old, const char *new);
```

- Entre ficheros o directorios en distintos sistemas de ficheros
- El fichero original puede no existir
- Si el nuevo fichero existe no será sobrescrito

- Leer el contenido de la ruta de un enlace simbólico:

```
int readlink(const char *path, char *b, size_t tb);
```

- El tamaño del enlace puede determinarse con `lstat(2)`
- La cadena `b` no contiene el carácter de fin de cadena

- Los comandos `ln` y `readlink` proporcionan acceso a estas funcionalidades

Borrado de ficheros

- Eliminar un nombre de fichero y posiblemente el fichero al que se refiere:

<unistd.h>

SV+BSD+POSIX

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

- Borra la entrada del directorio y decrementa el número de enlaces en el inodo
 - Cuando el número de enlaces llega a 0 y no hay ningún proceso que mantenga abierto el fichero, este se elimina, devolviendo el espacio al sistema
 - El fichero (fifo, socket, dispositivo) permanecerá en el sistema mientras que exista un proceso que lo mantenga abierto
 - Un fichero en un directorio con el bit *sticky* activo solo puede ser borrado por el propietario del fichero o del directorio
- El comando `rm` proporciona acceso a esta funcionalidad

Cerrojos de ficheros

- Crear, comprobar y eliminar un cerrojo POSIX:

<unistd.h>

SV+POSIX

```
int lockf(int fd, int cmd, off_t len);
```

- `fd` es un descriptor de fichero abierto para escritura
- `cmd` es una de las siguientes operaciones:
 - `F_LOCK`: Bloquea la región especificada del fichero, esperando hasta que un cerrojo incompatible previo se libere
 - `F_TLOCK`: Como `F_LOCK`, pero la llamada no espera, sino que devuelve -1 con `errno=EAGAIN` o `EACCESS` si la región está bloqueada
 - `F_ULOCK`: Desbloquea la región indicada del fichero
 - `F_TEST`: Devuelve 0 si la región está desbloqueada o bloqueada por el proceso, o -1 con `errno=EAGAIN` o `EACCESS` si la bloquea otro proceso
- `len` especifica la región (relativa a `pos`, que es la posición actual):
 - `len = 0, pos..infinito` (fin de fichero actual y sucesivos)
 - `len > 0, pos..pos+len-1`
 - `len < 0, pos-len..pos-1`
- Los cerrojos son consultivos (`read(2)` y `write(2)` no comprueban su existencia), por lo que solo son útiles entre procesos que cooperan

Acceso a directorios

- Abrir un directorio indicado por `name`:

```
DIR *opendir(const char *name);
```

- Devuelve un puntero al flujo de directorio, posicionado en la primera entrada del directorio
- El tipo de datos `DIR` se usa de forma similar al tipo `FILE` especificado por la librería de entrada salida estándar

- Leer entradas de un directorio:

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

- La función retorna una estructura `dirent` que apunta a la siguiente entrada en el directorio, y `NULL` cuando llega al final u ocurre un error
- El único campo contemplado por el estándar `POSIX` es `d_name`, de longitud variable (menor que `NAME_MAX`)

- Cerrar el directorio definido por el descriptor, haciéndolo inaccesible a subsecuentes llamadas:

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

<code><sys/types.h></code> <code><dirent.h></code>
SV+BSD+POSIX

Creación y borrado de directorios

- Crear un directorio:

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

- `mode` especifica los permisos para el nuevo directorio (modificados por `umask`)
- El UID y GID del nuevo directorio serán el EUID y EGID del proceso, pero si el directorio padre tiene el bit *setgid* activo, el GID será el GID del directorio

<code><sys/stat.h></code> <code><sys/types.h></code>
SV+BSD+POSIX

- Eliminar un directorio:

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

- Cambiar el nombre o ubicación de un fichero o directorio:

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

- Si `new` existe se elimina y si es un directorio ha de estar vacío
- `old` y `new` han de ser del mismo tipo y estar en el mismo sistema de ficheros
- Si `old` es un enlace simbólico será renombrado, si lo es `new` será sobrescrito

<code><unistd.h></code>

SV+BSD+POSIX

<code><stdio.h></code>

BSD+POSIX

- Un fichero en un directorio con el bit *sticky* activo solo puede ser borrado o renombrado por el propietario del fichero o del directorio
- Los comandos `mkdir`, `rmdir` y `mv` proporcionan acceso a estas funcionalidades

Ejemplos de Preguntas Teóricas

¿Qué máscara (umask) debería fijar el proceso para que los ficheros se creen con los permisos simbólicos `rw- -w- r--`, considerando que `open(2)` especifica los permisos 0666?

- ☐ 0024.
- ☐ 0042.
- ☐ 0624.

¿Cuál de las siguientes afirmaciones sobre `dup(2)` y `dup2(2)` es cierta?

- ☐ Solo se pueden duplicar descriptores asociados a ficheros regulares.
- ☐ Los descriptores 0 (*stdin*), 1 (*stdout*) y 2 (*stderr*) no se pueden duplicar.
- ☐ Después de duplicar un descriptor, se puede cerrar.

¿Qué diferencias hay entre un enlace rígido y uno simbólico a un fichero?

- ☐ El enlace rígido tiene el mismo inodo que el fichero original y el simbólico, uno distinto.
- ☐ El enlace simbólico incrementa el número de enlaces del inodo y el simbólico no.
- ☐ El enlace rígido se puede hacer a cualquier fichero y el simbólico solo dentro del mismo sistema de ficheros.



AMPLIACIÓN DE SISTEMAS OPERATIVOS Y REDES

Grado en Ingeniería Informática / Doble Grado

Universidad Complutense de Madrid

Material adicional

Control de ficheros

- Manipular un descriptor de fichero:

```
int fcntl(int fd, int cmd);  
int fcntl(int fd, int cmd, long arg);
```

<unistd.h>
<fcntl.h>

SV+BSD+POSIX

- cmd determina la operación que se realizará sobre el fichero:
 - F_DUPFD: Duplica el descriptor como `dup(2)`
 - F_GETFD: Obtiene los flags del descriptor (`FD_CLOEXEC`)
 - F_SETFD: Fija los flags del descriptor al valor especificado en `arg`
 - F_GETFL: Obtiene los flags del fichero que se fijaron con `open(2)`
 - F_SETFL: Fija algunos flags del fichero (por ejemplo `O_APPEND`, `O_NONBLOCK` o `O_ASYNC`) al valor especificado en `arg`

Control de ficheros: Cerrojos

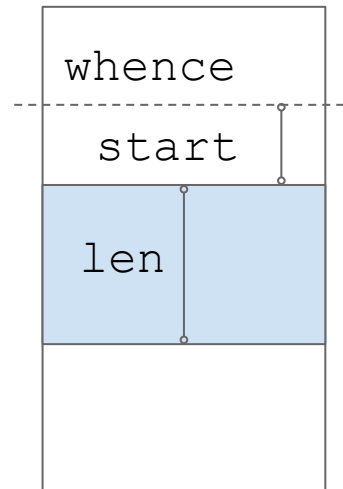
- Bloquear regiones de un fichero:

```
int fcntl(int fd, int cmd, struct flock *lock);
```

```
struct flock {  
    short int l_type;      /* F_RDLCK, F_WRLCK o F_UNLCK */  
    short int l_whence;    /* SEEK_SET, SEEK_CUR o SEEK_END */  
    off_t l_start;         /* Inicio de la región bloqueada */  
    off_t l_len;           /* Tamaño de la región (0=hasta EOF) */  
    pid_t l_pid;           /* Proceso que bloquea (F_GETLK) */  
}
```

- Tipos de cerrojos:

- **De lectura o compartido (F_RDLCK):** El proceso está leyendo el área bloqueada por lo que no puede ser modificada
 - Pueden establecerse varios sobre una misma región
- **De escritura o exclusivo (F_WRLCK):** El proceso está escribiendo, por lo que ningún otro debe leer o escribir del área bloqueada
 - Solo puede haber uno



Control de ficheros: Cerrojos

- `cmd` determina la operación que se realizará sobre el cerrojo:
 - `F_GETLK`: Comprueba si se puede activar el cerrojo descrito en `lock`
 - Si se puede activar, establece el campo `l_type` de `lock` a `F_UNLCK`
 - Si no, devuelve en `lock` los detalles de uno de los cerrojos que lo impiden, incluyendo el PID del proceso que lo mantiene
 - `F_SETLK`: Activa (si `l_type` es `F_WRLCK` o `F_RDLCK`) o libera (si `l_type` es `F_UNLCK`) el cerrojo descrito por `lock`
 - Si hay un cerrojo incompatible, devuelve -1 con `errno=EAGAIN`
 - `F_SETLKW`: Igual que `F_SETLK`, pero
 - Si hay un cerrojo incompatible, espera a que sea liberado
- Los cerrojos activos pueden consultarse en `/proc/locks`
- `flock(2)` y `flock(1)` permiten gestionar cerrojos BSD (no POSIX)