TEMA 2.2. Sistema de Ficheros

PROFESORES:

Rubén Santiago Montero
Eduardo Huedo Cuesta
Luis M. Costero

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

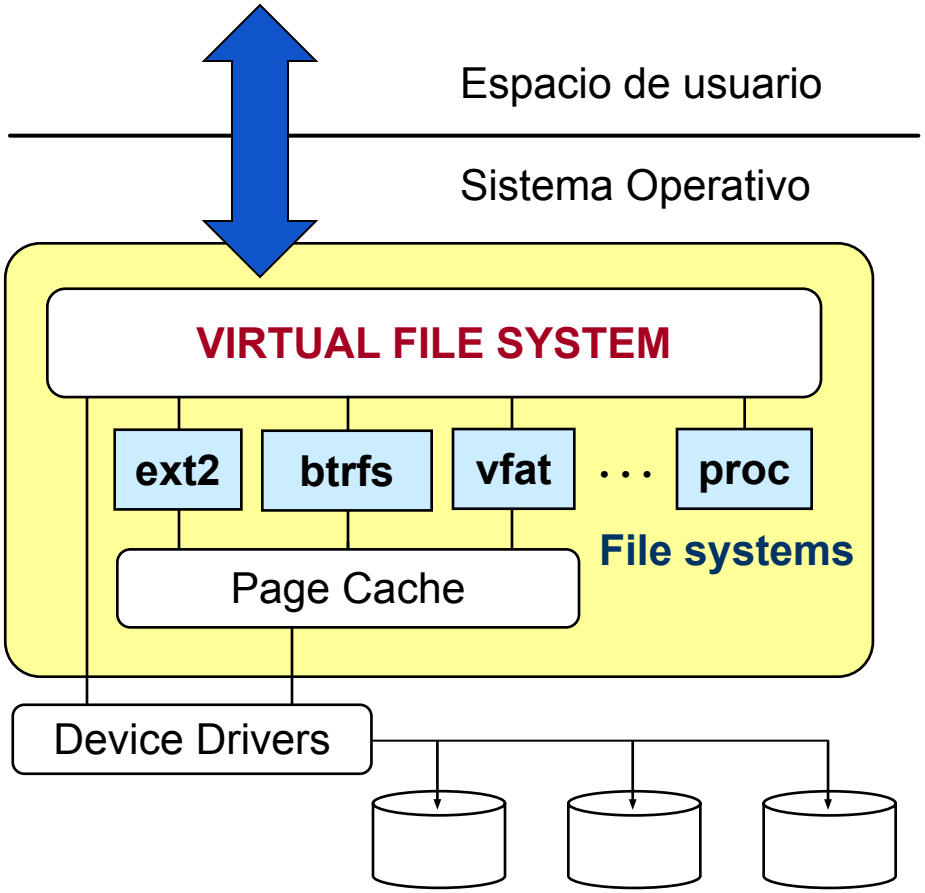
- Conjunto de tablas y estructuras que permiten gestionar los ficheros y directorios

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: procfs, tmpfs...

```
filesystems(5)
```

Estructura



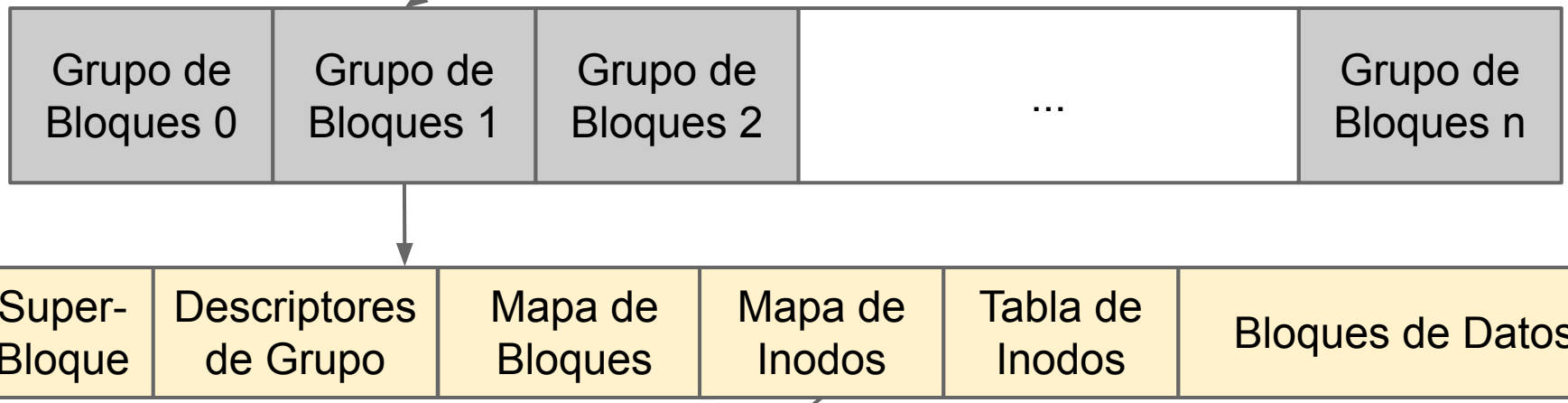
- La capa VFS establece un **enlace** bien definido **entre el kernel del SO** y los diferentes **sistemas de ficheros**
  - Proporciona las diferentes **llamadas** para la gestión de ficheros, **independientes del sistema de ficheros**
  - Permite acceder a múltiples sistemas de ficheros distintos
- 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 páginas (*sync*)

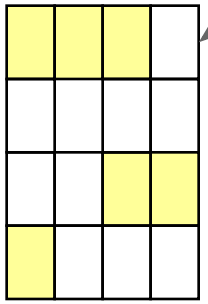
Estructura: Grupos de bloques

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



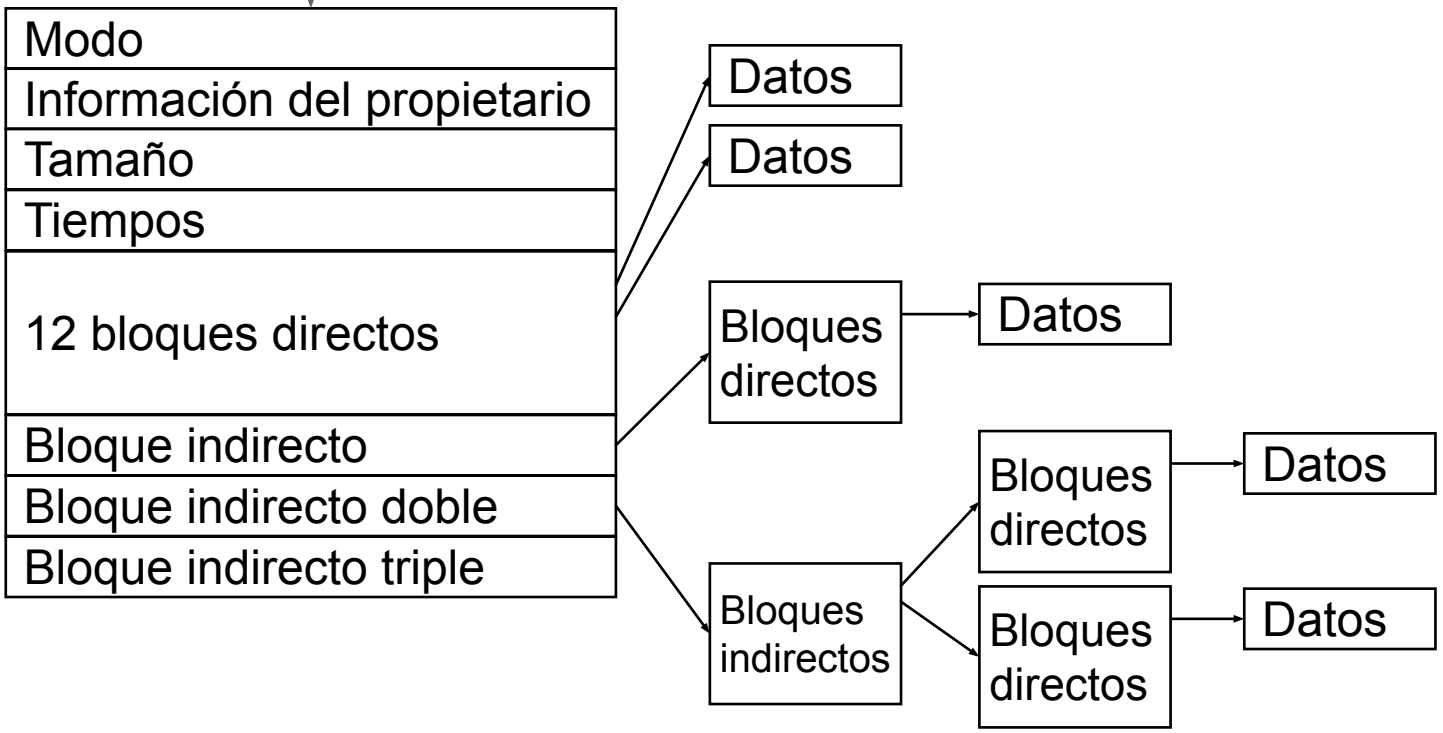


`ext2(5)`
`ext3(5)`
`ext4(5)`

Estructura: Inodos

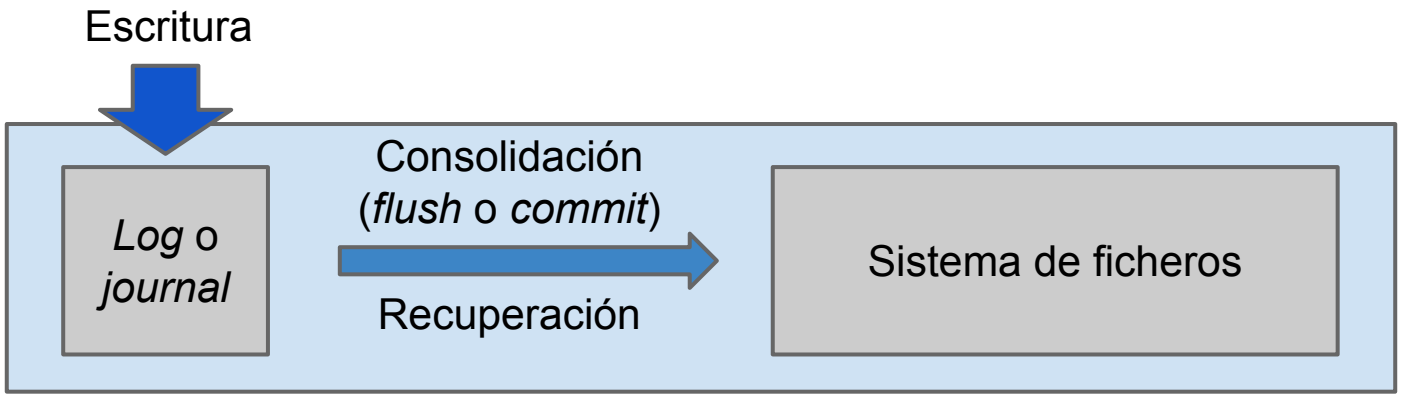
Directorio:

Inodo:



Journaling

- Cuando un sistema de ficheros tradicional no se apaga correctamente, se 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, por lo que se debe hacer de manera manual
- Los sistemas de ficheros modernos (XFS, ext3...) incorporan un fichero, región o dispositivo especial, denominado *log* o *journal*
  - Los metadatos (inodos, mapas…) se escriben primero en el *journal*
  - En caso de fallo se usa el *journal* para devolver el FS a un estado consistente
  - La consolidación se hace periódicamente o cuando tamaño del *journal* supera un umbral



Atributos de ficheros

- Obtener el estado de un fichero:

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

SV+BSD+POSIX

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

  - `lstat` no sigue enlaces simbólicos
  - 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 enlaces en la búsqueda
    - **EFAULT**: Dirección no válida
    - **EACCES**: Permiso denegado
    - **ENAMETOOLONG**: Nombre de fichero muy largo

- El comando `stat` proporciona acceso a esta funcionalidad

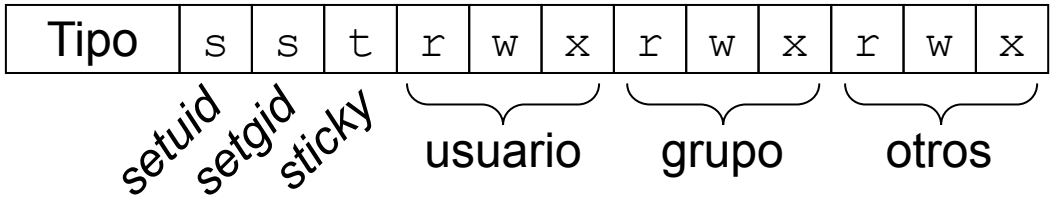
Atributos de ficheros

```
struct stat {
    dev_t     st_dev;     /* Dispositivo que lo contiene */
    ino_t     st_ino;     /* Inodo */
    mode_t    st_mode;    /* Tipo de fichero y permisos */
    nlink_t   st_link;    /* Número de enlaces rígidos */
    uid_t     st_uid;     /* UID del propietario */
    gid_t     st_gid;     /* GID del propietario */
    dev_t     st_rdev;    /* Disp. si fichero especial */
    off_t     st_size;    /* Tamaño en bytes */
    blksize_t st_blksize;/* Tamaño bloque E/S ficheros */
    blkcnt_t  st_blocks; /* Bloques físicos 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 (sectores) 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

- Campo `st_mode`:



- Permisos especiales en ficheros ejecutables (tema 2.1):
  - *setuid*: El EUID del proceso creado se establece al UID del propietario del fichero (ej. `/usr/bin/passwd`, `/usr/bin/su` o `/usr/bin/sudo`)
  - *setgid*: El EGID del proceso creado se establece al GID del propietario del fichero
- Permisos especiales en directorios:
  - *setgid*: Los ficheros se crean con el GID del directorio, en lugar del EGID del proceso
  - *sticky*: Los ficheros sólo pueden ser borrados o renombrados por el propietario del fichero, por el propietario del directorio o por un proceso privilegiado (ej. `/tmp`)

`inode(7)`

Atributos de ficheros

- Macros para **comprobar el tipo:**

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

- Máscaras y bits para **comprobar los permisos** (operadores bit a bit: `|` `&` `~` `^`):

  `S_IRWXU`: Permisos para el usuario (`00700`)
  `S_IRWXG`: Permisos para el grupo (`00070`)
  `S_IRWXO`: Permisos para otros (`00007`)

  `S_I` { `R` / `W` / `X` } { `USR` / `GRP` / `OTH` } : { Lectura / Escritura / Ejecución } para { Usuario / Grupo / Otros }

  `S_ISUID`: Bit *setuid* (`04000`)
  `S_ISGID`: Bit *setgid* (`02000`)
  `S_ISVTX`: Bit *sticky* (`01000`)

`inode(7)`

Atributos de ficheros: Permisos

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

SV+BSD+POSIX

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

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

  - La modificación suele hacerse leyendo los permisos actuales y realizando operaciones lógicas bit a bit
  - El EUID del proceso debe coincidir con el del propietario del fichero, o el proceso debe ser privilegiado
  - Algunos errores:
    - **ENOENT**: El fichero no existe
    - **EIO**: Error de E/S
    - **ELOOP**: Demasiados enlaces simbólicos

- El comando `chmod` proporciona acceso a esta funcionalidad

Atributos de ficheros: Permisos

<unistd.h>
SV+BSD+POSIX

- Comprobar los permisos de un fichero:

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

  - `mode` puede ser:
    - `F_OK`: Comprueba que el fichero existe
    - `R_OK`, `W_OK` y `X_OK` combinados con OR bit a bit: Comprueba que el fichero existe y permite lectura, escritura o ejecución, respectivamente
  - Se tiene en cuenta la ruta completa
  - Se usan los identificadores de usuario y grupo reales
  - La llamada devuelve -1 si el fichero no existe o se deniega alguno de los permisos, o si ocurre un error (y establece `errno`)

Creación y apertura de ficheros

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

SV+BSD+POSIX

- Abrir y posiblemente crear un fichero:

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

  - `flags` debe indicar el modo de acceso y puede incluir otras opciones:
    - `O_RDONLY`: Acceso de sólo lectura
    - `O_WRONLY`: Acceso de sólo escritura
    - `O_RDWR`: Acceso de lectura y escritura
  - `mode` indica los permisos a aplicar en caso de que se cree un nuevo fichero (con la opción `O_CREAT`)
    - En octal (precedidos por cero 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, se crea con los permisos proporcionados en `mode` (si se omite, se usa un valor arbitrario de la pila)
  - `O_EXCL`: Se usa con `O_CREAT` para provocar 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`: 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`: 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 pero aumentando la latencia

Creación y apertura de ficheros: Permisos

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

SV+BSD+POSIX

- 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`
  - `open(2)`, `mkdir(2)` y otras llamadas que crean ficheros borran los permisos indicados en `umask` del argumento `mode` con `mode & ~umask`, por ejemplo:

```
0666 & ~0022 = 0644
```

  - El valor por defecto de `umask` suele ser `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

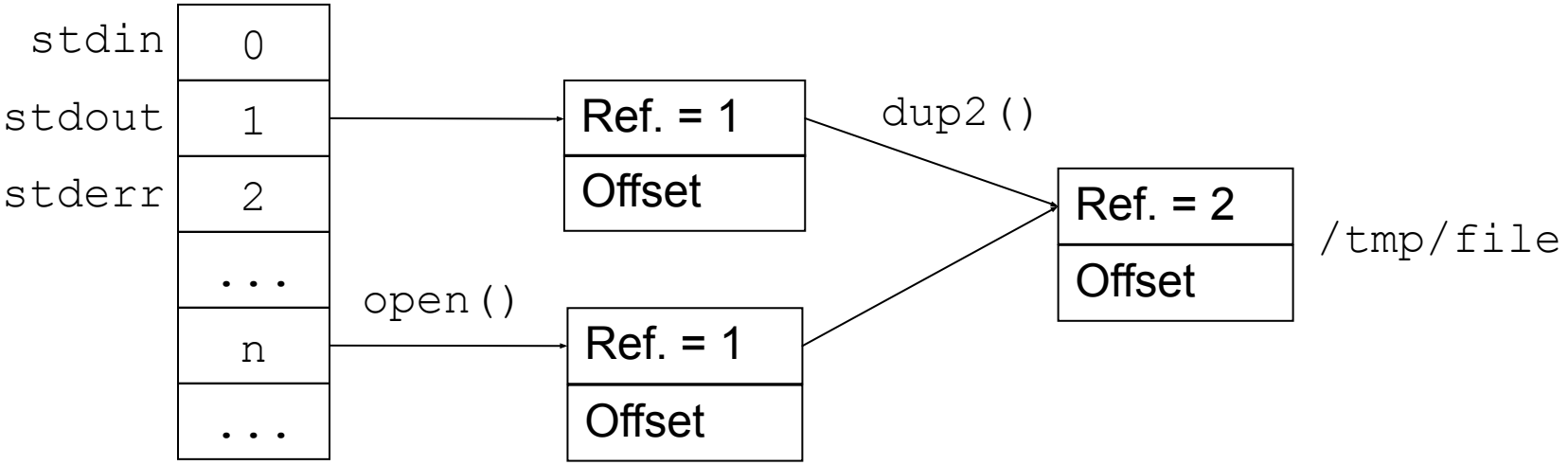
Duplicación de descriptores

<unistd.h>
SV+BSD+POSIX

- Duplicar un descriptor:

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

  - 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` estuviera 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

<unistd.h>
SV+BSD+POSIX

- 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 whence);
int close(int fd);
```

  - `whence` puede ser `SEEK_SET`, `SEEK_CUR` o `SEEK_END`
  - No deben mezclarse estas llamadas al sistema con funciones de librería (ej. `fopen`, `fread`, `fwrite`… de `stdio.h`, clases `fstream` en C++...)

- La escritura de ficheros se realiza a través de la *cache* de páginas, proporcionando un acceso eficiente (puede evitarse con la opción `O_DIRECT`)

- 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

<unistd.h>
SV+BSD+POSIX

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

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

  - No puede hacerse a otro sistema de ficheros ni con directorios
  - Si el nuevo fichero existe no será sobrescrito

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

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

  - Puede hacerse a otro sistemas de ficheros y con directorios
  - 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 *buf, size_t bufsize);
```

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

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

Borrado de ficheros

<unistd.h>
SV+BSD+POSIX

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

```
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 permanecerá en el sistema mientras que exista un proceso que lo mantenga abierto

- El comando `rm` proporciona acceso a esta funcionalidad

Cerrojos de ficheros

<unistd.h>
SV+POSIX

- Crear, comprobar y eliminar un cerrojo 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`: Establece un cerrojo en la región especificada y se bloquea si hay un cerrojo incompatible hasta que se libere
    - `F_TLOCK`: Como `F_LOCK` pero no se bloquea si hay un cerrojo incompatible, sino que devuelve -1 con `errno=EAGAIN`
    - `F_ULOCK`: Elimina un cerrojo de la región especificada
    - `F_TEST`: Devuelve 0 si la región no tiene cerrojo o pertenece al proceso, o -1 con `errno=EAGAIN` si tiene un cerrojo de 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 sólo son útiles entre procesos que cooperan

Acceso a directorios

<sys/types.h>
<dirent.h>

SV+BSD+POSIX

- Abrir un directorio:

```
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 E/S 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 un directorio:

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

Creación y borrado de directorios

- Crear un directorio:

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

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

  - `mode` especifica los permisos para el nuevo directorio (modificados por `umask`)
  - El nuevo directorio se crea con el EUID y EGID del proceso (salvo si el directorio padre tiene el bit *setgid* activo)

- Eliminar un directorio:

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

<unistd.h>
SV+BSD+POSIX

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

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

<stdio.h>
BSD+POSIX

  - 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

- Los comandos `mkdir`, `rmdir` y `mv` proporcionan acceso a estas funcionalidades

Ejemplos de Preguntas Teóricas

¿Qué valor de umask se debería fijar para que los ficheros se creen con los permisos 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 rígido no.
- ☐ El enlace rígido se puede hacer a cualquier fichero y el simbólico sólo dentro del mismo sistema de ficheros.

Material adicional

Control de ficheros

<unistd.h>
<fnctl.h>
SV+BSD+POSIX

- Manipular un descriptor de fichero:

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

  - `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`

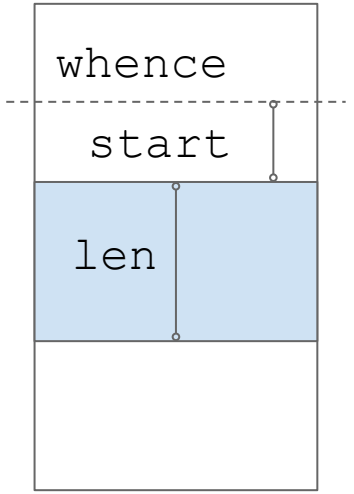
Control de ficheros: Cerrojos

- 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)