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

OTROS AUTORES:

Ignacio Martín Llorente

Juan Carlos Fabero Jiménez

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

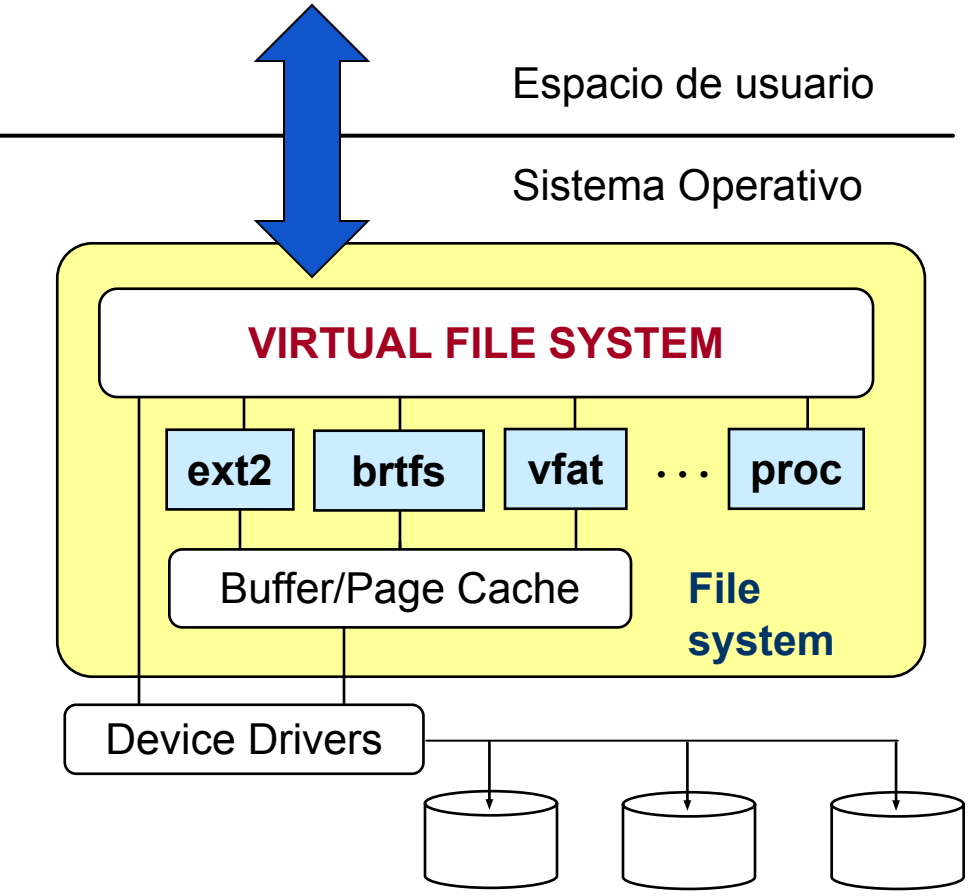
Desde el punto de vista del sistema operativo

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

Tipos de Sistema de Ficheros:

- **Basados en disco:** Están implementados en discos duros. Ejemplos: Minix, ext2-3-4, FAT, NTFS, ISO 9660, ufs, hpfs, BTRFS, ZFS…
- **Basados en red (o distribuidos):** Se utilizan para acceder a sistemas de ficheros remotos. Independientemente del tipo normalmente se acceden como NFS (Network File System)
- **Basados en memoria (o pseudo):** Residen en memoria principal mientras el sistema operativo se está ejecutando. Ejemplos: procfs, tmpfs...

Estructura

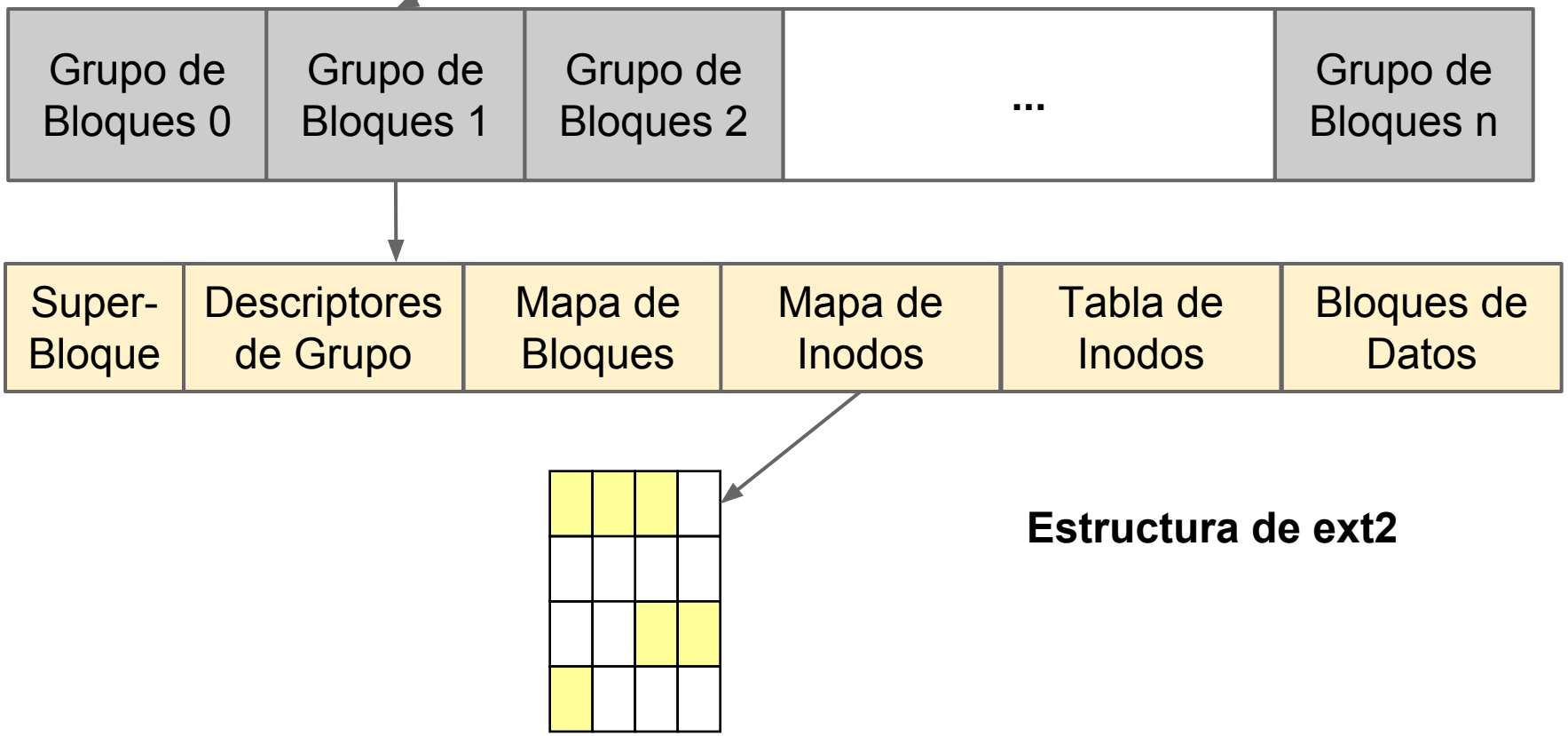


- Establece un **enlace** bien definido **entre el kernel del SO** y los diferentes **sistemas de ficheros**
- Proporciona las diferentes **llamadas** para la gestión de archivos, **independientes del sistema de fichero**
- Permite acceder a múltiples sistemas de ficheros distintos
- **Optimiza la entrada/salida** por medio de:
  - La *cache* de i-nodos y entradas de directorio (*dentry*) del 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

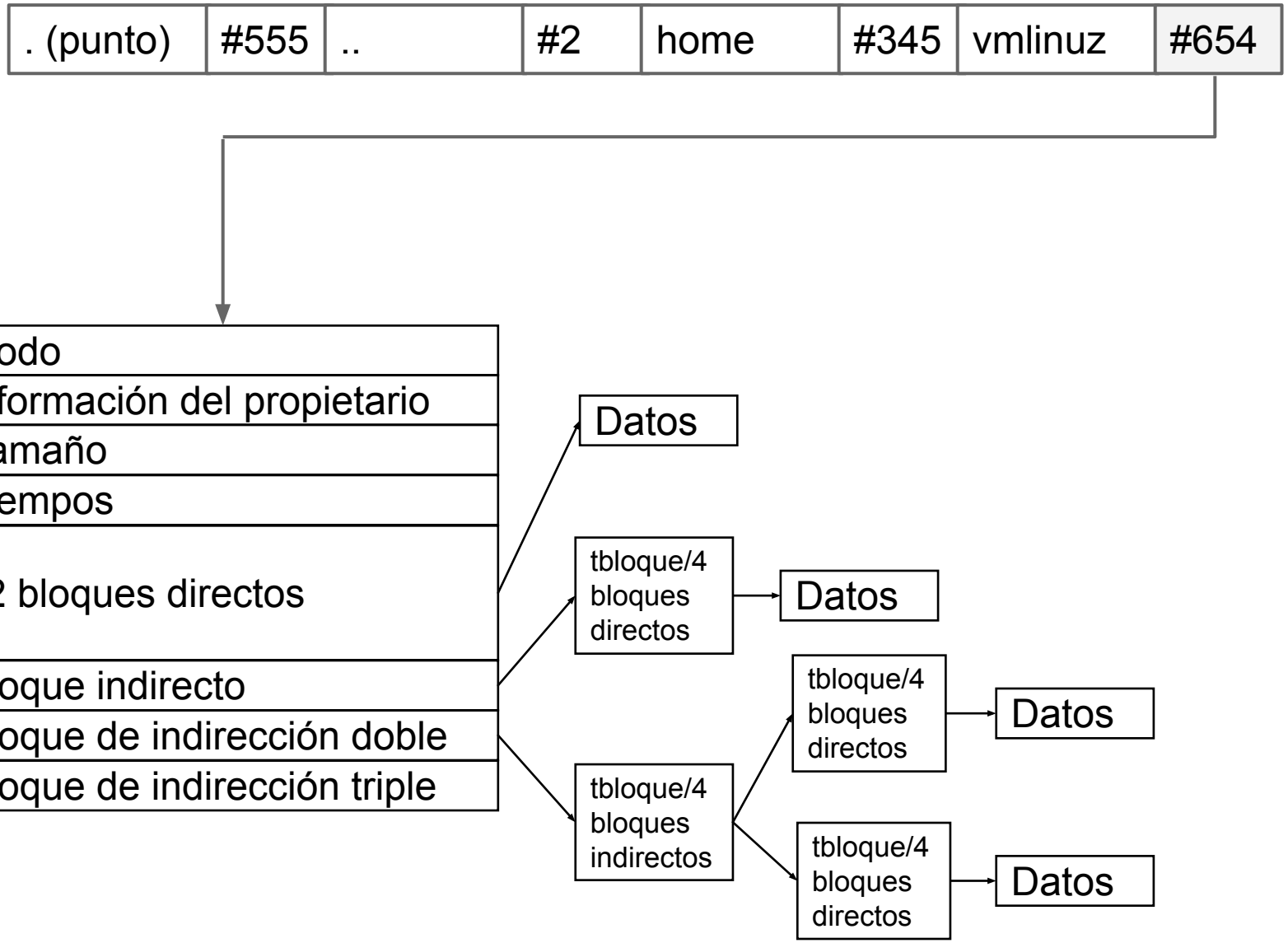
- Bloques de datos cerca de sus inodos
- Los inodos cerca del directorio de inodos



Estructura de ext2

Estructura: i-nodos

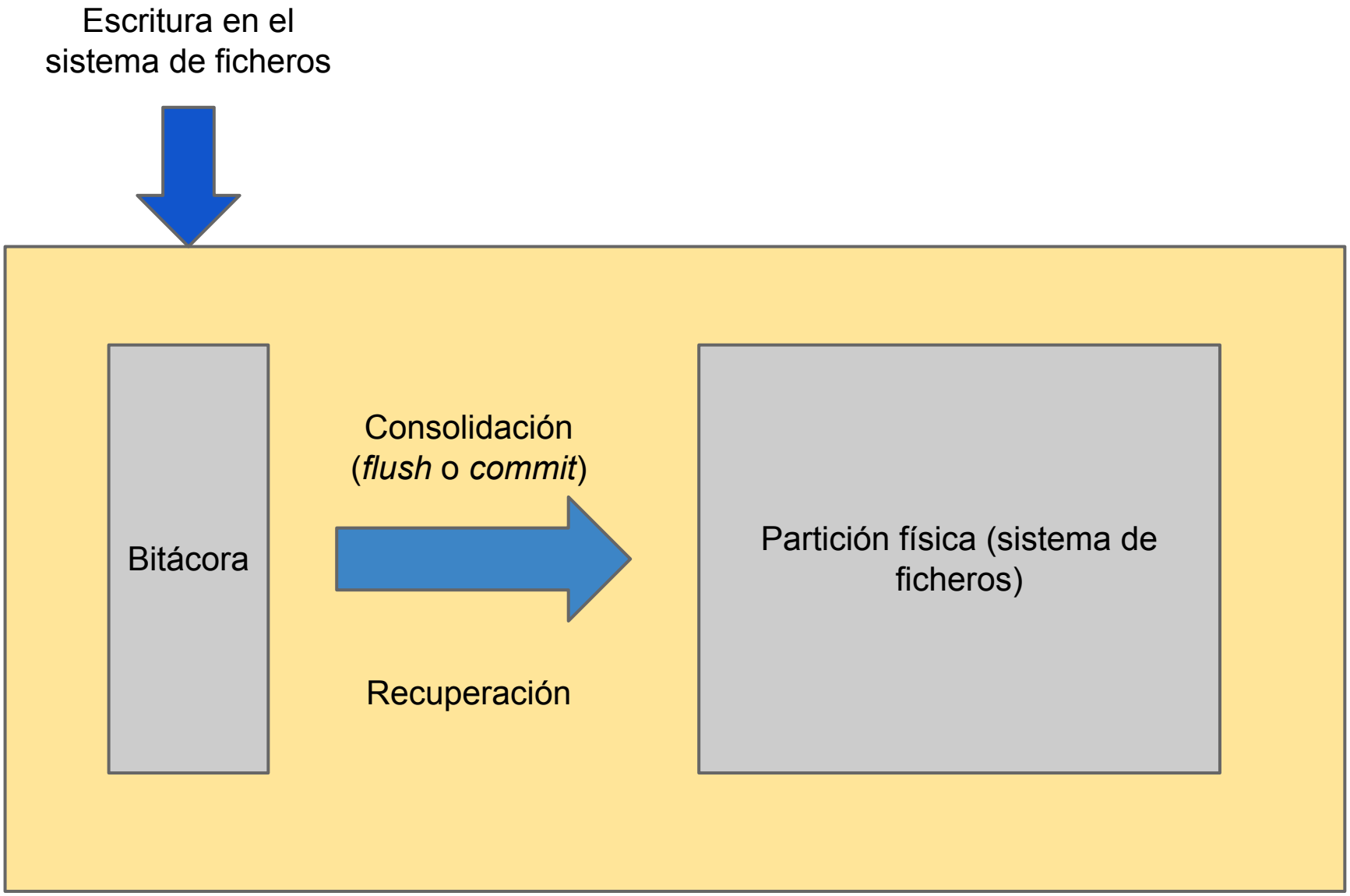
Directorio:



Journaling

- Cuando un sistema de ficheros tradicional no se apaga correctamente, el SO debe comprobar la integridad y consistencia del sistema de ficheros en el siguiente arranque (utilidad `fsck`)
  - Implica recorrer toda la estructura del disco en búsqueda de i-nodos huérfanos, 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 incorporan un archivo especial, denominado **bitácora** (*journal* o *log*), que evita la corrupción de los datos
- Los cambios en el disco se escriben primero en el archivo de bitácora
- En caso de apagado brusco del sistema, puede utilizarse el archivo de bitácora para devolver el sistema de ficheros a un estado coherente
- Los sistemas de ficheros ReiserFS, ext3 y ext4 incorporan la funcionalidad de *journaling*

Journaling


Escritura en el sistema de ficheros
Bitácora
Consolidación (*flush* o *commit*)
Recuperación
Partición física (sistema de ficheros)

Journaling. Variantes

- Los metadatos se escriben siempre en el archivo de bitácora de forma inmediata
- Dependiendo de cómo se escriben los datos, se tienen tres variantes principales:
  - ***Writeback*:** No mantiene el orden de actualización entre bloques y la bitácora. Es el más rápido pero no garantiza la integridad en caso de fallo (inconsistencia metadatos - bloques).
  - ***Ordered*:** Las actualizaciones de bloques y metadatos se realizan al mismo tiempo en una transacción.
  - ***Journal mode*:** Los bloques de datos también se escriben en el archivo de bitácora. Ofrece mayor protección frente a fallos, pero se degrada el rendimiento.
- La consolidación puede hacerse periódicamente o cuando el archivo de bitácora se llena hasta cierto punto

Atributos de Ficheros

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

SV+BSD+POSIX

- Funciones para la obtención de información de ficheros
- No se necesitan permisos sobre el archivo pero sí para buscar en PATH
- Obtención del estado de un fichero:

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

- Obtención de estado de ficheros y enlaces:

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

- Obtención de estado de ficheros mediante un descriptor (`open()`):

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

- **EBADF:** Descriptor no válido.
- **ENOENT:** PATH incorrecto o nulo.
- **ENOTDIR:** Componente del PATH no es un directorio
- **ELOOP:** Demasiados *links* en la búsqueda
- **EFAULT:** Dirección no válida
- **EACCES:** Permiso denegado
- **ENAMETOOLONG:** Nombre de archivo muy largo

Atributos de Ficheros

```
struct stat {
  dev_t st_dev;     /* Dispositivo que contiene el i-nodo */
  ino_t st_ino;     /* I-nodo */
  mode_t st_mode;   /* Permisos */
  nlink_t st_link; /* Número de enlaces duros (hard) */
  uid_t st_uid;     /* UID del propietario */
  gid_t st_gid;     /* GID del propietario */
  dev_t st_rdev;    /* Dispositivo, si fichero especial */
  off_t st_size;    /* Tamaño (bytes) */
  unsigned long st_blksize; /* Tamaño de bloque E/S */
  unsigned long st_blocks;  /* Bloques reservados */
  time_t st_atime; /* Último acceso */
  time_t st_mtime; /* Última modificación */
  time_t st_ctime; /* Último cambio (I-nodo) */
}
```

Atributos de Ficheros

- `unsigned long st_blksize:`

  Representa el tamaño de bloque *preferido* en las operaciones de E/S sobre el fichero/dispositivo para un rendimiento óptimo

- `unsigned long st_blocks:`

  Representa el tamaño del fichero en bloques de 512 bytes T(Kb) = `st_blocks` * 0.5

- `time_t st_atime:`

  Modificado por llamadas `read, write, mknod, utime, truncate`

- `time_t st_mtime:`

  Modificado por llamadas `write, mknod, utime`. No en cambio de usuario, grupo, permisos...

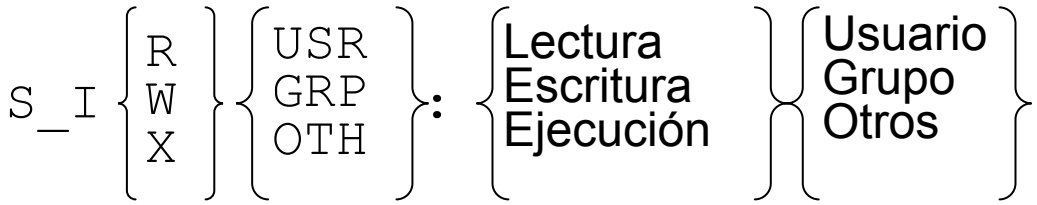
- `time_t st_ctime:`

  Modificado únicamente cuando se altera la información del i-nodo

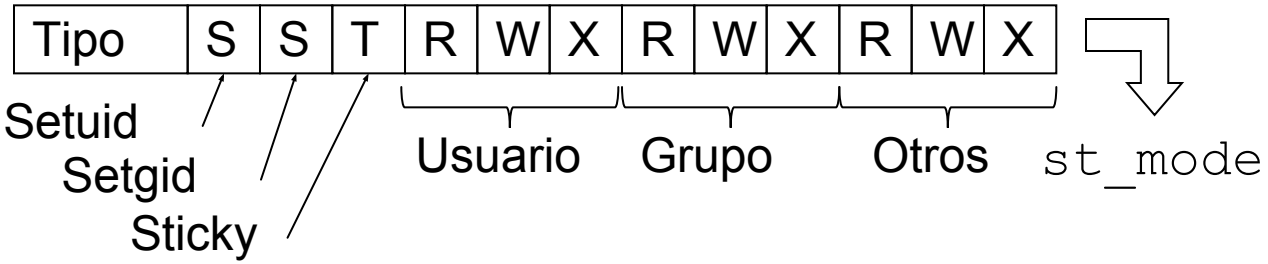
Atributos de Ficheros

El estándar POSIX ofrece una serie de macros y *flags* para comprobar el tipo de archivo y permisos (`st_mode`), consultar `<sys/stat.h>`

- **Tipo de archivo:**
  - `S_ISLNK(mode)`: es un enlace simbólico
  - `S_ISREG(mode)`: es un fichero normal
  - `S_ISDIR(mode)`: es un directorio
  - `S_ISCHR(mode)`: es un dispositivo por caracteres
  - `S_ISBLK(mode)`: es un dispositivo por bloques
  - `S_ISFIFO(mode)`: es un FIFO o pipe
  - `S_ISSOCK(mode)`: es un socket
- Flags útiles para **comprobar permisos** (usar con operadores *bitwise* | & ~ ^*)*:
  - `S_IRWXU`: Permisos de usuario (`0x00700`)
  - `S_IRWXG`: Permisos de grupo (`0x00070`)
  - `S_IRWXO`: Permisos para el mundo (`0x00007`)



Atributos de Ficheros: Permisos



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

SV+BSD+POSIX

- Cambio del tipo de permisos (no se puede cambiar el tipo de fichero), puede realizarse la modificación leyendo los permisos actuales:

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

int fchmod(int filedes, mode_t mode);
```

- Al igual que antes la modificación de los permisos ha de hacerse mediante operaciones lógicas (*bitwise*)
- El UID efectivo del proceso debe ser 0 (root) ó coincidir con el del propietario del fichero

- **EPERM:** Error de permisos.
- **EROFS:** Sistema de sólo lectura.
- **ENOENT:** No existe el fichero.
- **EIO:** Error de E/S.
- **ENOTDIR:** Elemento del PATH no existe
- **ELOOP:** Demasiados *links* simbólicos.

Atributos de Ficheros: Permisos

<unistd.h>
SV+BSD+POSIX

- Comprobación del tipo de permisos sobre un fichero:

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

- El modo es una combinación de los siguientes flags:
  - R_OK: El archivo existe y tenemos permisos de lectura.
  - W_OK: El archivo existe y tenemos permisos de escritura.
  - X_OK: El archivo existe y tenemos permisos de ejecución.
  - F_OK: El archivo 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 a la función fallará si alguno de los permisos no se satisface

Creación y Apertura de Ficheros

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

SV+BSD+POSIX

- Apertura y/o creación de un archivo o dispositivo:

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

  - `path`: Ruta del fichero o dispositivo
  - `flag`: Se debe indicar uno de los siguientes *flags*:
    - `O_RDONLY`: Acceso de solo lectura
    - `O_WRONLY`: Acceso de solo escritura
    - `O_RDWR`: Acceso de lectura y escritura
  - `mode`: Determina los permisos con que se creará el archivo (necesario con `O_CREATE` o `O_TMPFILE`). Estos permisos se ven modificados por el `umask` del proceso.
    - En octal (precedidos por 0 en C/C++)
    - Como OR de bits (`S_IRWXU = 00700`, `S_IRUSR = 00400`)
    - (ver atributos fichero)
  - Devuelve un descriptor de fichero (compartido mediante `fork()`, `exec()`), fijando el puntero de acceso al principio del archivo
    - El descriptor del archivo es el menor disponible en el sistema

Creación y Apertura de Ficheros

Algunos *flags*:

- `O_CREAT`: Si el archivo no existe, lo crea (con los permisos de `mode`)
- `O_EXCL`: Usado en combinación con `O_CREAT` provoca un error si el archivo existe (*Exclusively Create*)
- `O_TRUNC`: Una vez abierto el archivo puede ser truncado (a pesar de `O_RDONLY`)
- `O_APPEND`: Antes de realizar cualquier escritura se posiciona el puntero de archivo a la última posición del fichero (actualizaciones simultáneas en NFS)
- `O_NONBLOCK`: Abre el archivo en modo no bloqueante
- `O_SYNC`: Abre el archivo en modo síncrono, bloqueando las llamadas `write` hasta que los datos sean físicamente escritos

Creación y Apertura de Ficheros: Permisos

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

SV+BSD+POSIX

- El argumento de permisos que se usa en la llamada `open()`, no es necesariamente los que recibe el archivo creado
- Establecer la máscara de permisos para la apertura de ficheros (`mask & 0777`)

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

- La función `open()` establece los permisos del nuevo archivo de la forma:

  Permisos = `mode & (~umask)`

  Ejemplo:

  `0666 & (~022) = 110 110 110 & 111 101 101 = 110 100 100 = 0644`

  `- rw- rw- rw- & (~--- -w- -w-) ⇒ - rw- r-- r--`

- Es útil para especificar los permisos que nunca se concederán a los archivos creados en el programa
- La función siempre se ejecuta correctamente, devolviendo la máscara anterior

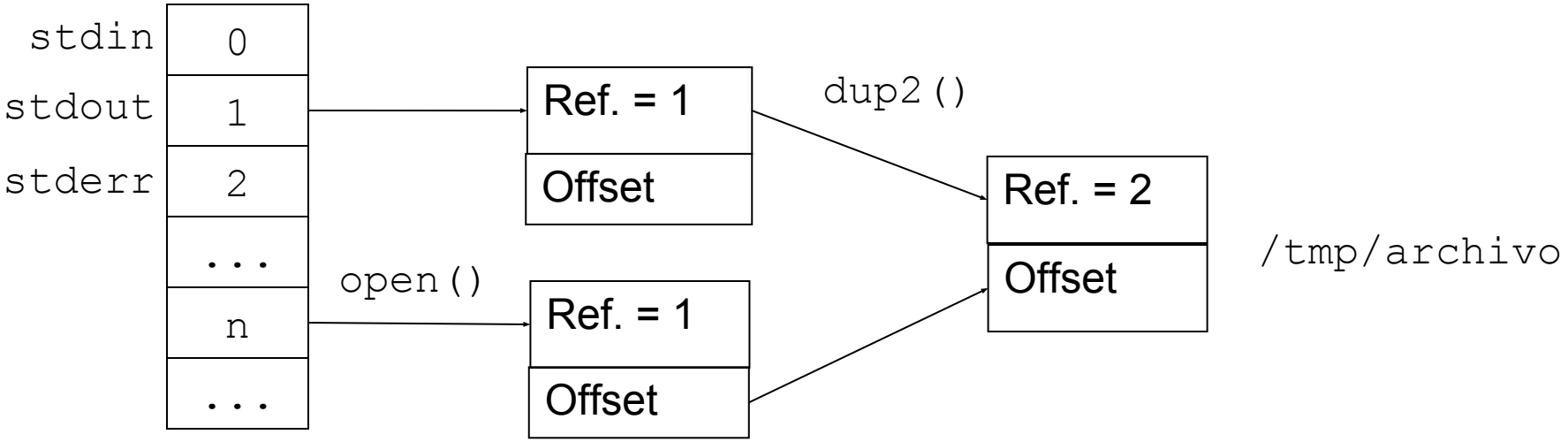
Duplicación de Descriptores

<unistd.h>
SV+BSD+POSIX

- Creación de un descriptor que se refiere a otro archivo previamente abierto:

```
int dup(int old_fd);
int dup2(int old_fd, int new_fd);
```

- Los dos descriptores comparten los cerrojos, punteros en el archivo y opciones del fichero, de forma que puede intercambiarse su uso
- El descriptor devuelto por `dup()` es el menor disponible en el sistema
- Con `dup2()`, `new_fd` referirá a `old_fd` y, si `new_fd` está abierto se cerrará
  - **EBADF:** El descriptor `old_fd` no corresponde a un fichero abierto o `new_fd` está fuera de rango
  - **EMFILE:** Número máximo de archivos abiertos alcanzado



Lectura y Escritura en fichero

<unistd.h>

SV+BSD+POSIX

- Lectura, escritura, posicionamiento y cierre de 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);
```

- No mezclar las funciones de librería con las llamadas al sistema (`fopen` y `open`, `fread` y `read`, `fwrite` y `write`… o clases `fstream` en C++)
- La escritura de los ficheros se realiza a través de la *Buffer/Page cache*, proporcionando un acceso eficiente. Para evitar la pérdida de información:
  - Flag `O_SYNC` en `open()`, sincroniza fichero y disco en cada escritura
  - Realizar la sincronización explícita mediante una llamada al sistema:

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

    La llamada no retorna hasta que termina la sincronización

Enlaces Simbólicos y Duros

<unistd.h>
SV+BSD+POSIX

- Enlaces duros (*hard links*)*:*

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

Únicamente sobre archivos en el mismo sistema de ficheros. Si el nuevo archivo existe no será sobrescrito

- Enlaces simbólicos (*soft links* o *symlinks*):

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

Los enlaces simbólicos pueden realizarse entre archivos en distintos sistemas de ficheros, o a pesar de que el archivo original no exista. Si el nuevo archivo existe no será sobrescrito

- Lectura del 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`

La cadena `b` no contiene el carácter de fin de cadena

Borrado de Ficheros

<unistd.h>
SV+BSD+POSIX

- Elimina 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 referencias en el i-nodo
- Cuando el número de referencias se reduce a 0 y no hay ningún proceso que mantenga abierto el archivo, el archivo se elimina, devolviendo el espacio al sistema
- El archivo (fifo, socket, dispositivo) permanecerá en el sistema mientras que exista un proceso que lo mantenga abierto o en uso

Control de Ficheros

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

- Manipula un descriptor de fichero:

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

- `cmd` determina la operación que se realizará sobre el archivo:
  - `F_DUPFD`: Duplica el descriptor en la forma de `dup()`. Los dos descriptores no comparten el flag *close-on-exec*
  - `F_GETFD`: Obtiene los flags del descriptor (solo *close-on-exec*)
  - `F_SETFD`: Fija los flags del descriptor (solo *close-on-exec*)
  - `F_GETFL`: Obtención de los flags del archivo como se fijaron con `open()`
  - `F_SETFL`: Fija los flags (`argv`) del descriptor: `O_APPEND`, `O_NONBLOCK` y `O_ASYNC`, únicamente

Control de Ficheros: Cerrojos

- Bloquear regiones de un fichero:

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

- El argumento `lock` es una estructura de la forma:

```
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;      /* Offset de la región bloqueada */
  off_t l_len; /* Longitud de la región, 0=hasta el final */
  pid_t l_pid; /* Proceso que mantiene el cerrojo (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 cerrojos 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 un cerrojo

Control de Ficheros: Cerrojos

- `cmd` determina la operación que se realizará sobre el cerrojo:
  - `F_GETLK`: Si se puede activar el cerrojo descrito en `lock` en el archivo, establece el campo `l_type` de `lock` a `F_UNLCK`. Si uno o más cerrojos incompatibles impiden que se pueda activar, devuelve los detalles de uno de ellos en los campos `l_type`, `l_whence`, `l_start` y `l_len` y establece el campo `l_pid` al 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 otro proceso mantiene un cerrojo incompatible, devuelve -1 y pone `errno` a **`EACCES`** o **`EAGAIN`**
  - `F_SETLKW`: Igual que `F_SETLK`, pero si hay un cerrojo incompatible, espera a que sea liberado
- Los cerrojos son consultivos, es decir, `read` y `write` no comprueban su existencia (Linux también ofrece cerrojos obligatorios, no POSIX)
- Los cerrojos se asocian al proceso, por lo que no se heredan con `fork` (Linux también ofrece cerrojos asociados al descriptor de fichero, que sí se heredan)
- Los cerrojos activos pueden consultarse en `/proc/locks`
- `flock(2)` proporciona los antiguos cerrojos de ficheros de UNIX (no POSIX)
- `lockf(3)` proporciona un interfaz para cerrojos de regiones basado en `fcntl`

Acceso a directorios

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

- Abre 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

- Lee un directorio:

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

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

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

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

Creación y borrado de directorios

- Creación y eliminación de directorios:

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

  - `mode`: Permisos con los que se crea el directorio, modificados en la forma habitual (`mode & ~umask`)
  - En la creación de los directorios el usuario y grupo del nuevo directorio serán los efectivos del proceso

- Para cambiar el nombre de un fichero o directorio:

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

  - Si `new` existe, se elimina antes de asociarlo `old`. Si `new` es un directorio, ha de estar vacío
  - Tanto `old` como `new` han de ser del mismo tipo (ficheros o directorios) y pertenecer al mismo sistema de ficheros
  - Si `old` es un enlace simbólico, será renombrado. Si `new` es un enlace simbólico, será sobrescrito