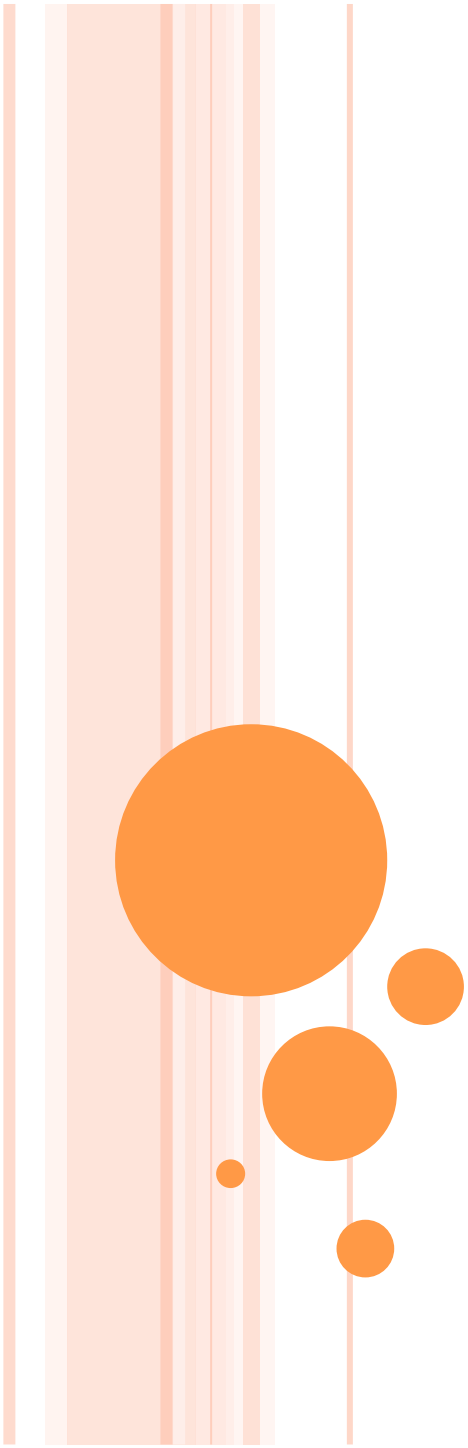


ULPGC Universidad de Las Palmas de Gran Canaria | Escuela de Ingeniería Informática

Práctica 3 : archivos

Fundamentos de los Sistemas Operativos

Grado en Ingeniería Informática

Rubén García Rodríguez
Alexis Quesada Arencibia
Eduardo Rodríguez Barrera
Francisco J. Santana Pérez
José Miguel Santos Espino

Curso 2019/2020

v1.1

PASO DE PARÁMETROS POR LÍNEA DE ÓRDENES: LA FUNCIÓN MAIN

- Función de entrada al programa C

```
int main(int argc, char *argv[]);
```

shell

```
$ ./ejecutable cadena1 cadena2
...
$
```

```
argc=3

argv[0] => ./ejecutable
argv[1] => cadena1
argv[2] => cadena2
```

PASO DE PARÁMETROS POR LÍNEA DE ÓRDENES: LA FUNCIÓN MAIN

```c
#include <stdio.h>
#include <stdlib.h>
// Procedimiento que comprueba los argumentos
void ComprobarArgumentos(int argc, char **argv);
// Procedimiento que imprime los argumentos pasados por línea de órdenes
void ImprimeArgumentos(int argc, char **argv);

void ComprobarArgumentos(int argc, char **argv){
   if (argc!=3) {
      printf("Invocación incorrecta: ejecutable cadena1 cadena2\n");
      exit(-1);
   }
}
void ImprimeArgumentos(int argc, char **argv) {
int cont;
for(cont=0;cont<argc;cont++)
  printf("%s\n", argv[cont]);
}
int main(int argc, char **argv)
{
  ComprobarArgumentos(argc,argv);
  ImprimeArgumentos(argc,argv);
  exit(0);
}
```

Manejo de errores

- En general la forma en la que una función de una librería o una llamada al sistema informa de errores es mediante la devolución de algún valor especial de retorno, que en general suele ser **-1** o **NULL** cuando se ha producido algún error y **0** o un valor distinto a los anteriores cuando la ejecución ha sido exitosa.
- Sin embargo, los valores exactos que informan de tales estados dependen de la función/llamada al sistema y por ello es necesario averiguarlo en cada caso, por ejemplo a través de la ayuda integrada en el *shell* (man).

```
int *asientos;
...
asientos=(int*)malloc(-100*sizeof(int));
if (asientos==NULL) {
  // Se ha producido un error en malloc
…
```

MANEJO DE ERRORES

- El valor de retorno en caso de error indica que efectivamente se ha producido un error pero no da ninguna pista sobre el error exacto que se ha generado. Para ello se utiliza la variable global **errno**.
- La variable **errno** se encuentra declarada en **<errno.h>** de la siguiente forma:

```
extern int errno;
```

- La variable **errno** almacena un valor entero que identifica el error concreto. Cada valor numérico se corresponde con un error.

errores.c

```
int *asientos;
...
asientos=(int*)malloc(-100*sizeof(int));
if (asientos==NULL) {
  // Se ha producido un error en malloc
  // errno identifica el error concreto
  printf("%d",errno);
}
```

shell

```
$ gcc -o errores errores.c

$ ./errores
12

$
```

MANEJO DE ERRORES

- La variable **errno** almacena un valor entero que identifica el error concreto. Cada valor numérico se corresponde con un error y un texto descriptivo de dicho error.
- La librería estándar de C proporciona una función que permite "traducir" cada valor entero de **errno** en su correspondiente texto descriptivo.

```
#include <stdio.h>
void perror(const char *str)
```

errores.c

```
int *asientos;
...
asientos=(int*)malloc(-100*sizeof(int));
if (asientos==NULL) {
  // Se ha producido un error en malloc
  // errno identifica el error concreto
  perror("Error en el malloc");
}
```

shell

```
$ gcc -o errores errores.c

$ ./errores
Error en el malloc: Cannot
allocate memory

$
```

Manejo de errores

errores.c

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main() {
int cont;

for(cont=0;cont<=53;cont++) {
  printf("%d",cont);
  fflush(NULL);
  errno=cont;
  perror(" ");
}

}
```

shell

```
$ gcc -o errores errores.c

$ ./errores
0 : Success
1 : Operation not permitted
2 : No such file or directory
3 : No such process
4 : Interrupted system call
5 : Input/output error
6 : No such device or address
7 : Argument list too long
8 : Exec format error
9 : Bad file descriptor
10 : No child processes
...
$
```

Manejo de errores

- Otras alternativas a **perror()** son:

```
#include <string.h>
char * strerror(int errnum);

#include <string.h>
int strerror_r(int errnum, char *buf, size_t len)
```

Manejo de errores

- Si queremos conocer el error que se ha podido producir tras la llamada a una función de librería o después de una llamada al sistema es necesario consultar o almacenar el valor de la variable **errno** justo tras la llamada a la función de librería o tras la llamada al sistema.
- La consulta de la variable **errno** debería realizarse justo a continuación ya que nuevas llamadas a funciones de librería o llamadas al sistema podrían cambiar el valor de **errno** al ser una variable global y por panto se podría perder el rastro del primer error.

```
int *asientos;
int local_error;
...
asientos=(int*)malloc(-100*sizeof(int));
if (asientos==NULL) {
  local_error=errno;
  // otras instrucciones, llamadas a funciones...
  printf("%s\n", strerror(local_error));
  ...
}
```

Manejo de errores

- Para algunas **pocas** funciones no hay un código específico de retorno en caso de error. Estas funciones aseguran que **errno** tendrá un valor distinto de 0 en caso de que ocurra algún error.

- En estos casos el error se puede detectar inicializando **errno** a 0 antes de la llamada a la función y consultando a continuación el valor de **errno** .

```
...
errno=0;
//llamada a la función
if (errno) {
  perror("Error ...");
...
}
```

Introducción

- El manejo de la E/S de un sistema operativo es un punto clave
- Los programadores en C en Unix tienen disponibles dos conjuntos de funciones para acceso a archivos:
  - Funciones de la librería estándar: *printf, fopen...*
  - Llamadas al sistema
- Las funciones de librería se implementan haciendo uso de llamadas al sistema
- Aunque las funciones de librería hacen más cómodo el acceso, las llamadas al sistema proporcionan el tipo de acceso más directo que puede tener un programa

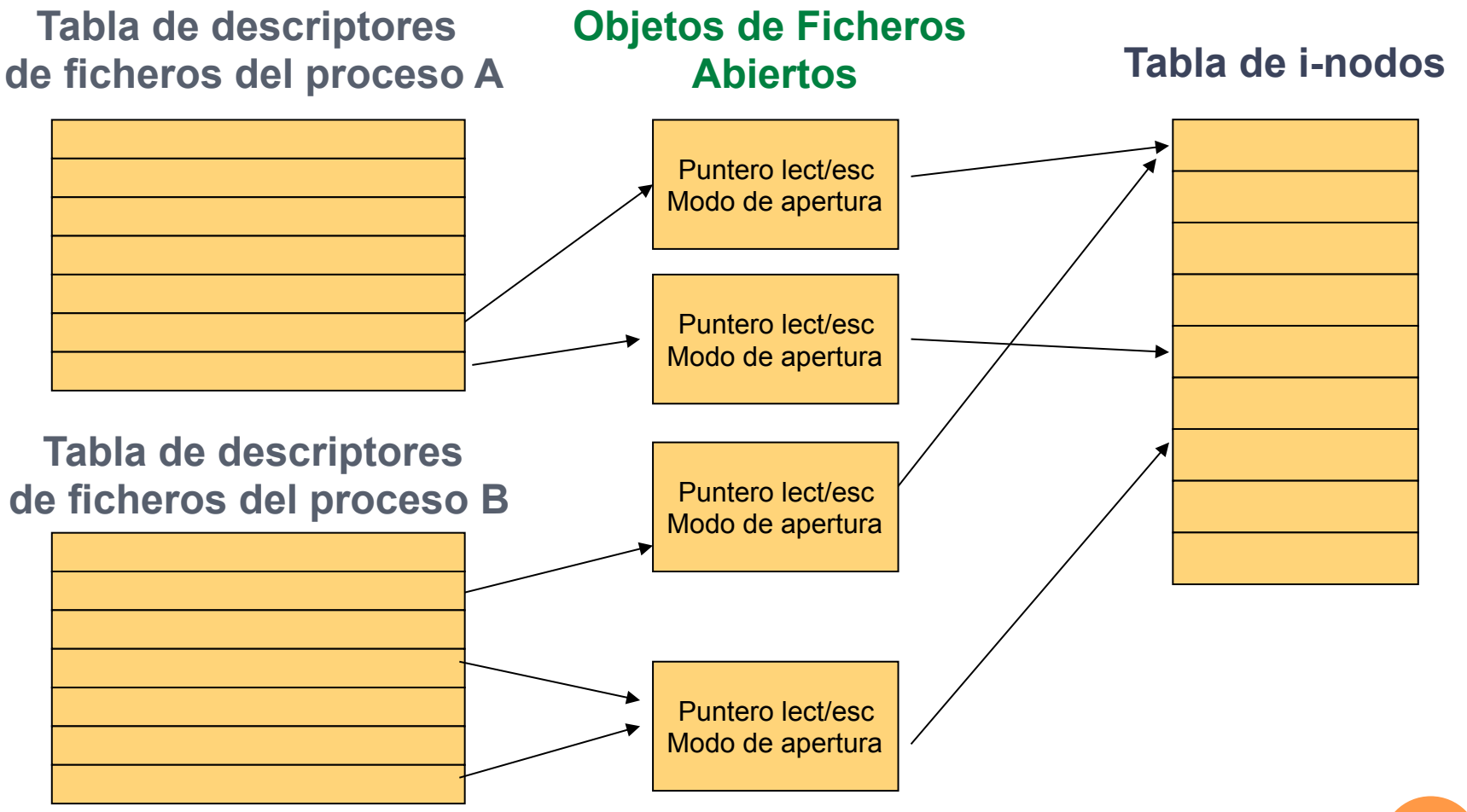
Introducción

- Para el kernel, todo fichero abierto se identifica con un descriptor de fichero
- Los accesos al fichero se realizan usando dicho descriptor
- Los descriptores sirven en realidad para acceder a cualquier otro componente del sistema que sea capaz de enviar y recibir datos: dispositivos, sockets,...
- Por convención se definen los siguientes descriptores:
  - 0 = entrada estándar
  - 1 = salida estándar
  - 2 = error estándar

INTRODUCCIÓN

- El kernel mantiene varias estructuras de datos muy importantes relacionadas con ficheros:
  - **Tabla de descriptores de fichero**. Es una estructura local a cada proceso que indica todos los ficheros abiertos por un proceso. Cada entrada apunta a su vez a una entrada en la tabla de ficheros del sistema.
  - **Objetos de ficheros abiertos**. Es una estructura manejada por el núcleo asociada a cada fichero abierto en el sistema (puntero de acceso - desplazamiento de lectura/escritura, etc…)
  - **Tabla de i-nodos**. Guarda información asociada a cada fichero (propietario, permisos, situación del fichero en disco, …)

INTRODUCCIÓN



LLAMADAS DE E/S

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags).
int open(const char *pathname, int flags, mode_t mode);
```

- *mode* indica los permisos de acceso (necesario si el fichero se va a crear)
- *flags* puede ser una combinación OR de:
  - O_RDONLY = Solo lectura
  - O_WRONLY = Solo escritura
  - O_RDWR = Lectura/escritura

  } Una de estas es obligatoria
  - O_CREAT = Crea el fichero si no existe
  - O_EXCL = Falla si el fichero ya existe (junto con O_CREAT)
  - O_TRUNC = Trunca el fichero a longitud 0 si ya existe
  - O_APPEND = Se añade por el final
  - O_NONBLOCK = Si las operaciones no se pueden realizar sin esperar, volver antes de completarlas
  - O_SYNC = Las operaciones no retornarán antes de que los datos sean físicamente escritos al disco o dispositivo

LLAMADAS DE E/S

```
#include <unistd.h>
int close(int fd);

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, int mode);
```

- *creat* es equivalente a:

  open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)

- Cuando un proceso termina todos sus ficheros abiertos son automáticamente cerrados por el kernel

LLAMADAS DE E/S

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

- Todo fichero tiene asociado un puntero que marca la posición del siguiente acceso, respecto al principio
- El puntero se guarda en la tabla de ficheros
- La interpretación de *offset (que puede ser negativo)* depende de *whence*:
  - SEEK_SET => posicionamiento respecto al inicio del fichero
  - SEEK_CUR => posicionamiento respecto a la posición actual
  - SEEK_END => posicionamiento respecto al fin del archivo
- *lseek* devuelve el puntero actual
- Si se lleva el puntero más allá del fin del fichero y luego se escribe, el fichero se hará más grande, rellenando con ceros

LLAMADAS DE E/S

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

- *count* es el nº de bytes a leer
- *read* devuelve el nº de bytes leídos o 0 si estamos al final del fichero
- Si p.ej. el fichero tiene 30 bytes e intentamos leer 100, *read* devuelve 30. La próxima vez que se llame *read* devuelve 0

Llamadas de E/S

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

- *nbytes* es el nº de bytes a escribir
- Si el fichero se abrió con O_APPEND el puntero del fichero se pone al final del mismo antes de cada operación de escritura
- devuelve el nº de bytes escritos

Llamadas de E/S

```
#include <unistd.h>
int fsync(int fd);
```

- El sistema puede mantener los datos en memoria por varios segundos antes de que sean escritos a disco, con el fin de manejar la E/S más eficientemente
- *fsync* hace que se escriban todos los datos pendientes en el disco o dispositivo

```
#include <unistd.h>
int ftruncate(int fd, size_t length);
```

- *ftruncate* trunca el fichero fd al tamaño length

LLAMADAS DE E/S

```
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
```

- *stat* devuelve una estructura de información de un fichero
- *fstat* hace lo mismo pero a partir de un fichero abierto
- *lstat* es para enlaces simbólicos (si se llama a *stat* para un enlance simbólico se devuelve información del fichero al que apunta el enlace, pero no del enlace mismo)
- La información de *stat* es similar a la que devolvería el comando

  ls -l

LLAMADAS DE E/S

```
struct stat
{
dev_t st_dev;              /* device (major and minor numbers) */
ino_t st_ino;              /* inode */
mode_t st_mode;                  /* protection and type of file */
nlink_t st_nlink;                /* number of hard links */
uid_t st_uid;              /* user ID of owner */
gid_t st_gid;              /* group ID of owner */
dev_t st_rdev;             /* device type (if inode device) */
off_t st_size;             /* total size, in bytes */
unsigned long st_blksize;        /* blocksize for filesystem I/O */
unsigned long st_blocks;         /* number of blocks allocated */
time_t st_atime;                 /* time of last access */
time_t st_mtime;                 /* time of last modification */
time_t st_ctime;           /* time of last change */
};
```

LLAMADAS DE E/S

- Tipos de ficheros en UNIX:
  - Fichero regular. El kernel no distingue si es de texto o con datos binarios
  - Directorio. Un fichero que contiene los nombres de otros ficheros y punteros a la información en los mismos. Cualquier proceso con permiso de lectura para el directorio puede leer los contenidos del directorio, pero solo el kernel puede escribirlo
  - Ficheros especiales: para dispositivos
    - de carácter
    - de bloque (típicamente discos)
  - FIFO
  - Enlace simbólico. Un tipo de fichero que apunta a otro fichero
  - Socket

LLAMADAS DE E/S

- El tipo de fichero va codificado en el campo *st_mode* de la estructura *stat*
- Podemos determinar el tipo de fichero pasando *st_mode* a las macros:
  - S_ISREG() => fichero regular
  - S_ISDIR() => directorio
  - S_ISCHR() => fichero especial de carácter
  - S_ISBLK() => fichero especial de bloque
  - S_ISFIFO() => FIFO
  - S_ISLNK() => enlace simbólico
  - S_ISSOCK() => socket

LLAMADAS DE E/S

```
#include <unistd.h>
int access(char *path, int amode);
```

- *access* permite comprobar que podemos acceder a un fichero
- *amode* es el modo de acceso deseado, combinación de uno o más de R_OK, W_OK, X_OK
- Un valor de F_OK en *amode* sirve para comprobar si el fichero existe

LLAMADAS DE E/S

```
#include <sys/types.h>
#include <sys/stat.h>
int fchmod(int fildes, mode_t mode);
Int chmod(char *path, mode_t mode);
```

- *fchmod* y *chmod* cambia permisos de acceso a un fichero

```
#include <sys/types.h>
#include <unistd.h>
int fchown(int fd, uid_t owner, gid_t group);
int chown(char *path, uid_t owner, gid_t group);
```

- *fchown* y chown cambia el usuario y grupo asociado a un fichero

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

- *umask* sirve para establecer los permisos de creación de nuevos ficheros

LLAMADAS DE E/S

```
#include <sys/types.h>
#include <utime.h>
int utime(char *path, struct utimebuf *times);
```

- *utime* permite modificar las fecha de último acceso y última modificación de un fichero

```
struct utimebuf
{
    time_t actime;          /* fecha de acceso */
    time_t modtime;         /* fecha de modificación */
}
```

LLAMADAS DE E/S

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod(char *path, mode_t mode, int dev);
```

- Permite crear directorios, ficheros especiales y tuberías con nombre
- Con mknod podemos crear un directorio, si *mode* es S_IFDIR (en ese caso se ignora *dev*)
- Sin embargo, *mknod* solo puede ser invocado por un superusuario, por lo que no es muy útil para crear directorios

Llamadas de E/S

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(char *path, mode_t mode);
```

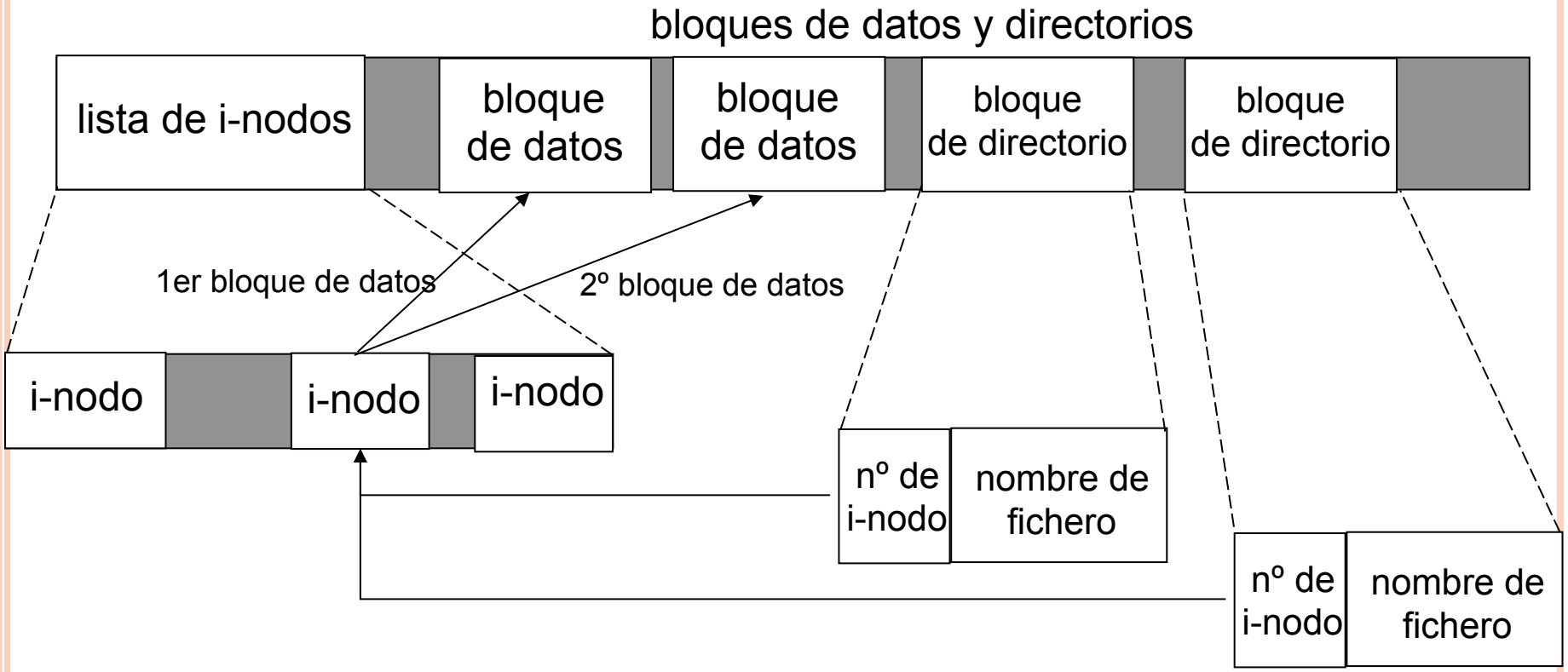
- Con *mkdir* podemos crear directorios

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

- *rmdir* borra un directorio, siempre que:
  - esté vacío
  - no sea el directorio de trabajo de ningún proceso

LLAMADAS DE E/S

- enlaces e i-nodos:



LLAMADAS DE E/S

- La mayor parte de la información de un fichero está en su i-nodo (tipo de fichero, permisos, tamaño, punteros a los bloques de datos,etc.)
- Dos entradas de directorio pueden apuntar al mismo i-nodo (hard links)
- Cada i-nodo tiene una cuenta de enlaces que cuenta el nº de entradas de directorio que apuntan al i-nodo
- Solo cuando la cuenta llega a cero se borra el fichero (los bloques de datos)
- Los enlaces simbólicos (symbolic links) son ficheros en los que los bloques de datos contienen el nombre del fichero al que apunta el enlace

LLAMADAS DE E/S

```
#include <unistd.h>
int link(const char *existingpath, const char *newpath);
int unlink(const char *pathname);
```

- Estas funciones permiten crear y eliminar enlaces (hard links)
- Solo el superusuario puede crear hard links que apunten a un directorio (se podrían provocar bucles en el sistema de ficheros difíciles de tratar)
- *unlink* no necesariamente borra el fichero. Solo cuando la cuenta de enlaces llegue a cero se borra el fichero. Pero si algún proceso tiene el fichero abierto, se retrasa el borrado. Cuando el fichero se cierre se borrará automáticamente

LLAMADAS DE E/S

- Esta propiedad del *unlink* se usa a menudo para asegurarnos de que un fichero temporal no se deje en disco si el programa termina (o aborta)

Llamadas de E/S

- El enlace simbólico se crea con:

```
#include <unistd.h>
int symlink(const char *realpath, const char
   *sympath);
```

- Se crea una nueva entrada de directorio *sympath* que apunta a *realpath*
- No es necesario que *realpath* exista al momento de usar *symlink*

LLAMADAS DE E/S

- La función *open* sigue un enlace simbólico, así que es necesaria una función para abrir el enlace y leer el nombre en él:

```
#include <unistd.h>
int readlink(const char *path, char *buf, int
   bufsize);
```

- *readlink* abre el enlace y lee el contenido, devolviéndolo en *buf*

LLAMADAS DE E/S

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dp);
int closedir(DIR *dp);
```

- Para leer el contenido de un directorio se debe seguir los siguientes pasos:
  - llamar a *opendir* pasándole el path deseado
  - llamar repetidamente a *readdir* pasándole el DIR* obtenido. Cuando se han leído todas las entradas de directorio *readdir* devuelve NULL
  - llamar a *closedir* pasándole el DIR*

- *dirent* es una estructura que contiene el nº de i-nodo y el nombre de la entrada entre otros datos