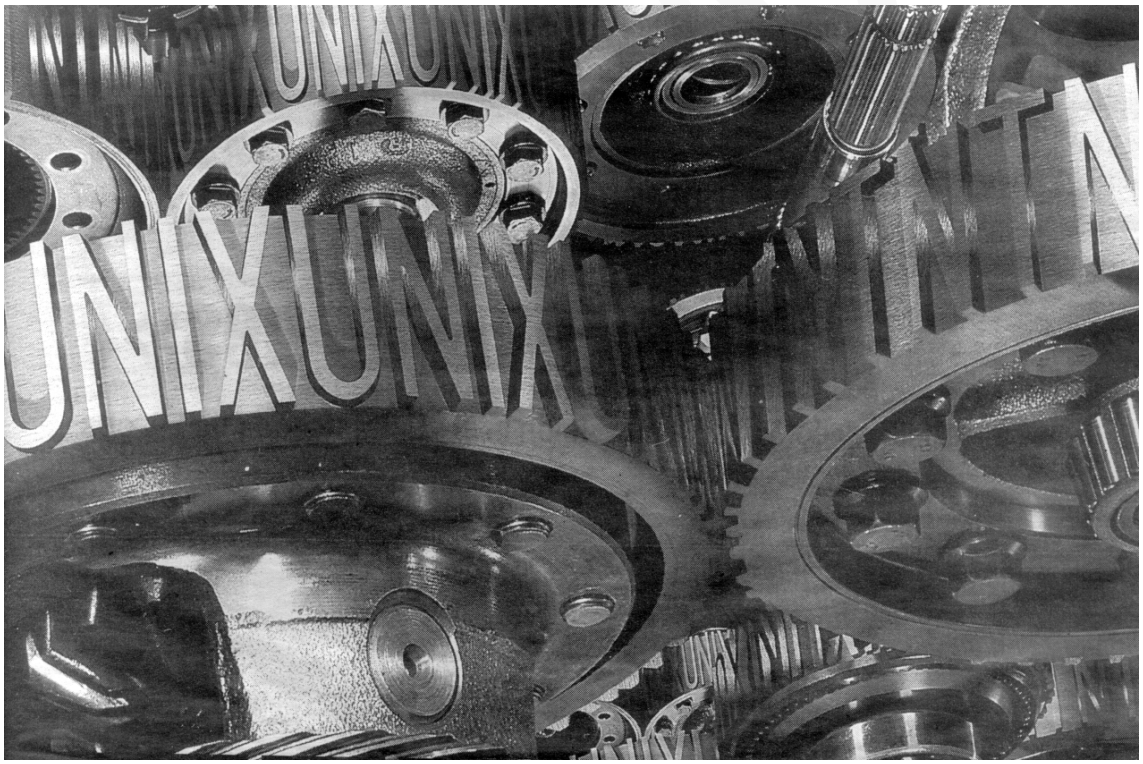
Universidad de Cantabria
Grado de Ingeniería Informática

Sistemas Operativos

Práctica 1 - Parte 4



Rafael Menéndez de Llano Rozas

Guion

1

Llamadas al sistema usando ficheros

En esta práctica adquirirás la visión del sistema como programador haciendo uso de las llamadas al sistema POSIX para la creación, sincronización y comunicación de procesos.

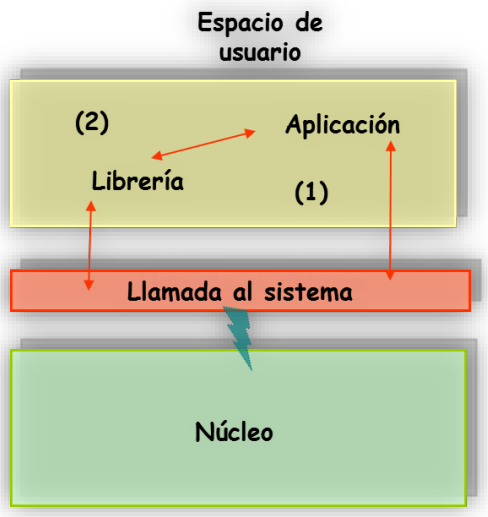
Llamadas al sistema

Para realizar las llamadas al sistema seguiremos el estándar POSIX (o IEEE 1003). Este estándar cuenta con 17 secciones:

- POSIX.1 define el núcleo de llamadas e incorpora el estándar del lenguaje ANSI C (110 funciones).
- POSIX.1b es la parte de tiempo real.
- POSIX.1c la encargada de definir las llamadas para manejar *threads*.
- Etc.

Las llamadas al sistema (*system call*) tienen formato de función de C y provocan una interrupción *software* o excepción que será atendida por el núcleo (*kernel*) y al que habrá que dejar ciertos parámetros en alguna parte de la memoria o en los registros de la CPU que serán los argumentos de la función que hace la llamada y lo que devuelve.

Una llamada al sistema se puede hacer de dos maneras: o bien de forma directa (1) siguiendo el prototipo de una función del estándar POSIX, o bien a través de una función de librería (2) que hace una llamada al sistema después de dar una funcionalidad extra. Por ejemplo, la función `printf` hace uso de la llamada al sistema `write`.



Siempre que se tengan dudas sobre una determinada llamada al sistema o a una librería debemos saber que la sección dos del manual (haz `man` de `man`) hace referencia a las llamadas al sistema servidas por el núcleo y la sección tres hace referencia a las funciones de librería contenidas en el sistema (ya sabes que la sección uno se encarga de los comandos ejecutables). Ten en cuenta que una llamada a librería no

necesariamente implica una llamada al sistema, por ejemplo, una llamada `strcpy` o `atoi` no hacen una llamada al sistema. En cualquier caso, desde el punto de vista del usuario, son simplemente unas llamadas a funciones C, obviamente, desde el punto de vista de la implementación (provocación de una excepción o no), la distinción entre una llamada al sistema o a una librería es fundamental.

En GNU/Linux la implementación de POSIX1 (no threads o tiempo real) se hace a través de la librería C (*libc* o en GNU *glibc*) que se incluye por defecto en todos los programas. Por eso, para hacer llamadas al sistema no es necesario incluir ninguna librería, al contrario de lo que ocurría por ejemplo con las llamadas a funciones matemáticas. Como hemos visto en el guion 1.3, normalmente cuando se utiliza una función de librería había que seguir dos pasos: primero, incluir un fichero de cabecera que nos da acceso a determinadas constantes y prototipos para utilizarla; y segundo, incluir el cuerpo de la librería utilizando el *linker*. Así, si queremos utilizar la librería matemática, teníamos que incluir la cabecera *math.h* y después enlazar (`gcc` llama a `ld`) con la propia librería (`-lm`).

```
/* este programa pro.c se compila con: gcc pro.c -lm */
#include <stdio.h>
#include <math.h> /*este es el fichero de cabecera NO la libreria*/
int main()
{
        float prueba;
        prueba = sin(M_PI/2);
        printf("%f \n", prueba);
}
```

Por defecto, cualquier programa tiene incluidas dos librerías: la `crt0.o` para la E/S y la `libc.a` o `libc.so` para las llamadas al sistema, por eso, en el caso de hacer una llamada al sistema, el acceso es directo y no se necesita incluir ninguna librería, el uso de un fichero de cabecera no es estrictamente necesario (pero sí muy conveniente) dependiendo de que necesitemos la declaración del prototipo de la llamada o alguna constante definida en la misma que facilite el trabajo de programación y además lo aclare, por ejemplo, para abrir un fichero de “sólo escritura” el parámetro es el 1, pero queda más claro poner O_WRONLY definido en `unistd.h` (unix estándar).

```
descriptor = open(“nombre_de_fichero”, O_WRONLY);
```

Las funciones que hacen de llamadas al sistema normalmente son de tipo entero (en algunas ocasiones de tipo puntero) y si devuelven un valor negativo, usualmente -1, nos indican que ha habido un error en la llamada o su ejecución. Para saber el tipo de error, existen tres variables globales externas definidas en `errno.h`, estas variables son:

1. `sys_errlist`: es un array de *strings*, donde cada uno indica un mensaje de error.
2. `sys_nerr`: un entero que nos da el número de errores del sistema.
3. `errno`: un entero que indica el código del error producido en alguna llamada al sistema.

Si quisiéramos ver todos los errores del sistema podríamos usar este programa:

```
#include <stdio.h>   /* esta forma de errores esta obsoleta */
// ahora mejor char *strerror(int errnum);
//extern char *sys_errlist[];
//extern int sys_nerr;
int main()    /* de hecho al compilar os saldrán warnings */
{
        int i;
        for (i=0; i<sys_nerr;i++)
            printf("%d: %s \n", i, sys_errlist[i]);  // obsoleto
}
```

Observa los comentarios del programa (de hecho, cuando compiles te lo advertirá). Actualmente, es mejor usar la llamada `strerror` que da las variables de error directamente. E incluso más cómodo

usar la función `perror` que toma automáticamente la variable externa `errno`, y la utiliza como índice para obtener dentro de `sys_errlist` el mensaje de error adecuado al que se le puede acompañar nuestro propio mensaje de error aclaratorio. El siguiente ejemplo escribirá por el canal de errores 2 nuestro mensaje seguido del error oficial cometido (cero indica que no ha habido error):

```
printf("%d: %s\n", i, strerror(i));
perror("Esto es un error en una llamada realizada");
```

Hay una macro definida en la cabecera `assert.h` a la que se le pasa una expresión escalar, si vale cero se pintará un mensaje de error por el canal de errores y se abortará el programa llamando a `abort`, que producirá una terminación anormal del programa, pero cerrando los ficheros ordenadamente. Esta macro se podría utilizar al realizar una llamada al sistema (negándola), para terminar el programa si la llamada no ha funcionado, pero es preferible usar la combinación if + perror + exit.

```
assert(!llamada_al_sistema); /* Falla */
```

1) Bájate el programa "errores.c" y pruébalo. Verás que salen avisos. Haz *man* de *man* y observa las secciones que puedes ver en el manual. Observa sobre todo las secciones dos y tres. Comprueba con `ldd` con qué librerías dinámicas (compartidas) has enlazado. Incluye la librería matemática y averigua si cambia.

2) Prueba el programa de los errores para saber el número y mensajes de error de tu sistema. Cámbialo para usar `strerror`. Introduce una línea con `perror` y prueba en qué canal sale el mensaje.

Utilización de llamadas para ficheros desde POSIX

Con el único fin de ver cómo se realiza una llamada al sistema desde un lenguaje de alto nivel, vamos a ver cómo trabajar con ficheros a través de las facilidades que éste nos proporciona. Serán parecidas a las que existen desde el estándar de C (siguiente apartado).

Las principales llamadas son:

Llamada al sistema	Significado
open	Abre o crea un fichero para lectura o escritura
creat	Crea un fichero vacío
close	Cierra un fichero abierto
read	Extrae información del fichero
write	Coloca información en el fichero
lseek	Coloca el puntero en un determinado byte

Un programa típico, realizará una llamada `open` (o `creat`) para inicializar un fichero, lo usará mediante `read`, `write` y `lseek` para manipular datos dentro del fichero y, por último, indicará que ha acabado mediante la llamada a `close`.

La llamada al sistema: open

Antes de que se pueda leer o escribir en un fichero es necesario previamente realizar una llamada a `open` para abrirlo (en las versiones antiguas el fichero debía existir previamente, en las nuevas no es necesario). Su uso básico es el siguiente (junto con `creat`):

```
#include <fcntl.h> /* file control ficheros de cabecera */
#include <sys/types.h>
#include <sys/stat.h>

int fd, flags;    /* descriptor y como abrirlo */
char *pathname;    /* nombre */
 ...   /* se puede hacer un man para saber el fichero de cabecera */
int open(const char *pathname, int flags);
```

donde *pathname* es el camino en forma de puntero a carácter (*string*), por ejemplo "`/home/pedro/mifichero`". El segundo parámetro, `flags`, es un entero que debe ser alguna de las constantes definidas en la librería `fcntl.h`, las más importantes son las siguientes:

O_RDONLY	abre un fichero sólo para leer.
O_WRONLY	" " " escribir.
O_RDWR	" " para leer y escribir.

si la llamada tiene éxito (puede abrirse el fichero) devuelve un valor entero positivo que es recogido en la variable `fd`, que será el descriptor del fichero y es el que se utiliza a partir de ese momento para identificar al fichero hasta su finalización. Si la llamada no tiene éxito el valor devuelto será negativo (típicamente -1), por ejemplo, si el fichero es para lectura y no existe. Ejemplo:

```
#include <fcntl.h>
/* el nombre del fichero tambien en formato relativo */
static char fich_trabajo[] = "un_nombre";

int main ()
{
      int filedes;
      /* abrir solo para leer, ojo con los parentesis */
      if((filedes = open(fich_trabajo, O_RDONLY)) == -1) /* o <0 */
      {
            perror("No puedo abrirle");
            exit(1); /* sale al existir error y devuelve un 1 */
      }
        /* aqui sigue el resto del programa */
      exit(0);    /* salida normal devolviendo un 0*/
}
```

La llamada al sistema: creat

Esta llamada se usa para crear un fichero nuevo o reescribir uno ya existente. Como en el caso de `open`, retorna un descriptor de fichero entero positivo si no hay error y usa los mismos ficheros de cabecera (si hay dudas sobre cual usar para una llamada se usa `man`).

Su utilización es la siguiente (se puede usar también open con 3 argumentos):

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

En este caso se añade el argumento `modo` que es un entero (es de tipo `mode_t` definido en `sys/types.h`) que da los permisos de acceso al fichero. Es decir, qué usuarios del sistema pueden leer, escribir o ejecutar el contenido del fichero (de forma análoga al comando `chmod`).

```
#define PERM 0644  /* permisos rw-r--r—en octal */
char nombre_fich[] = "nuevo";
int main ()
{
      int  fd;

      if ((fd = creat(nombre_fich, PERM)) == -1)
      {
            perror("no puedo crearlo");
            exit (1);    /* error*/
      }

      /*el resto del programa*/
      exit (0);
}
```

Si se hace un `man 2 creat` se verá que hay constantes definidas para estos casos, es equivalente a usar la llamada a open con el *flag* `O_CREAT`. Los *flag* se puede combinar con OR (|).

```
S_IRWXU  00700 user (file owner)
S_IRUSR  00400 user has read permission
S_IWUSR  00200 user has write permission
S_IXUSR  00100 user has execute permission
S_IRWXG  00070 group has read, write and execute permission
…
```

La llamada al sistema: close

Esta llamada es la inversa de `open`. Le dice al sistema que el proceso ha acabado de realizar operaciones con el fichero. Es necesario, entre otras razones porque siempre hay un límite máximo de ficheros abiertos por proceso (256 – 4).

Su uso es sencillo:

```
#include <fcntl.h> /* file control */

int fd, control;
…
control = close(fd);
```

Hay que tener en cuenta que cuando un proceso termina normalmente, de forma automática y por seguridad, se cierran todos sus ficheros.

Las llamadas al sistema: read y write

Estas llamadas se usan para copiar un número arbitrario de caracteres o bytes (recordar que en C ambos términos son intercambiables) desde (hacia) un fichero a (desde) memoria.

El modo de uso es:

```
#include <unistd.h>
ssize_t nleido     /*tamaño con signo parecido a long int */
int fd;
size_t n;          /*tamaño sin signo parecido unsigned long int */
void   *bufptr;
…
nleido = read(fd, bufptr, n);
```

donde, `fd` es el descriptor de fichero (obtenido de `open` o `creat`), `bufptr` es un puntero al buffer donde se copiarán los datos leídos, y `n` es el número de bytes a ser leídos. Hay que tener en cuenta que las lecturas y escrituras siempre se efectúan en una posición indicada por el puntero de lectura/escritura, y que éste se cambia automáticamente en cada operación (en la apertura está en la posición 0). Debemos asegurar que la dirección de memoria es válida y que hay sitio suficiente para colocar los datos leídos, se puede hacer con memoria dinámica, un *array* o el operador &.

La llamada será negativa si hay error, o en caso contrario, el número de bytes leídos, colocados en `nleido`, que normalmente coincidirá con `n` pero que puede ser menor si se ha alcanzado el final del fichero. Si la función devuelve 0 significará que el final de fichero ya ha sido alcanzado.

La llamada a `write` es la misma que la `read`, salvo que la operación es la contraria, tendremos que tener algo en la variable `buffer` que se volcará en el fichero. Si el valor devuelto por la llamada es menor que `count` significará que algo extraño ha ocurrido como por ejemplo que se ha llenado el sistema de ficheros.

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

La llamada al sistema: lseek.

La llamada `lseek` permite al usuario cambiar la posición del puntero de lectura/escritura, es decir, el número de orden del siguiente byte que se va a leer o escribir. Por tanto, posibilita un acceso aleatorio (*random*) no secuencial al fichero.

La utilización es:

```
#include <sys/types.h>
#include <unistd.h>

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

donde `fd` es el descriptor de fichero, el entero largo (`off_t` en `types.h`) `offset` es el número de bytes a ser sumados desde una posición de partida que viene determinada por partida:

```
SEEK_SET     (0) Comienzo del fichero
SEEK_CUR     (1) Posición actual
SEEK_END     (2) Final del fichero (el offset debiera ser negativo)
```

Se observa de nuevo que el uso de constantes hace más claro el programa. Hay que tener en cuenta que si ponemos como punto de partida 2, el offset tendrá que ser negativo; que, si el offset supera al final del fichero, éste simplemente crecerá y que si vamos antes del principio de fichero, la función nos devolverá un error.

Entrada / salida con ficheros en ANSI C. Comparación (repaso)

Podemos considerar un fichero como una zona de almacenamiento permanente, normalmente en disco, a la que nos referimos a través de un nombre (se supone que el lector está familiarizado con el manejo de ficheros). Para el lenguaje C internamente los ficheros son una estructura (`struct`) que se declara en la librería `stdio.h`, en la cual, se hace un *#define* de la estructura, al identificador `FILE` (también hay otros sistemas que definen el `FILE` como un `typedef`). Este identificador será usado para definir una variable de tipo puntero a `FILE`, que es la que usaremos en el programa para referirnos al fichero (hace las veces de descriptor).

Las operaciones que se pueden hacer con los ficheros son las habituales (como el caso de usar las llamadas al sistema). Así, tendremos operaciones para abrir y cerrar ficheros, para leer y escribir en ellos distintos tipos de datos y para poder movernos (acceso aleatorio) en su información.

La función para abrir ficheros es la `fopen`, que será de tipo puntero a `FILE` (puntero a `struct`), pero que no hay que declarar donde se utilice porque ya está hecho en `stdio.h`. Esta función admite dos argumentos: el nombre del fichero y el modo de apertura, ambos como tira de caracteres (los dos pueden ser punteros a `char`). El modo puede ser: "r" para sólo lectura, "w" para escritura (borrando el fichero si existe) y "a" para escritura, pero añadiendo datos, no borrando el fichero existente. Existen otros modos menos usuales como "r+", "w+", "a+" para hacer ambas cosas. También existen los modos equivalentes para ficheros binarios: "rb", "wb", "ab" y, "r+b", "w+b", "a+b".

En todos los modos, si la operación falla se devolverá el valor `NULL` (definido en `stdio.h`). La utilización es:

```
FILE *fichero;
fichero = fopen("nombre", "r");
```

La función para cerrar ficheros es la `fclose`, que tomará como argumento un puntero a fichero. Es una función entera, por lo tanto, si la operación es correcta, devolverá un 0 y si falla un -1.

```
int error;
error = fclose(fichero);
```

Estas funciones trabajan con ficheros con *buffer* o *stream* (almacenamiento intermedio en memoria), de tal manera que al cerrarlo, se vacía el buffer al fichero[1]. Cuando un programa termina normalmente (final de `main` o llamada a `exit`) se cierran todos sus ficheros automáticamente, pero esto no ocurre así cuando el programa termina anormalmente, por lo que la información del buffer asociado al fichero sin cerrar se pierde, por eso siempre es conveniente cerrarlos cuando se dejen de usar en el programa.

Al igual que existen funciones para la entrada/salida por los canales estándar, debido a la equivalencia entre dispositivos y ficheros dada por el sistema operativo, también existen funciones para la entrada/salida por ficheros, de hecho, son las mismas con distintos nombres (la entrada estándar es `stdin`, la salida es `stdout`, y la de error `stderr`).

Para la salida (escritura) de distintos tipos de datos está la equivalente a `printf`, que se llama `fprintf`. Esta función tiene tres argumentos, el primero es el puntero a fichero, el segundo la tira de control y el tercero las variables a escribir.

```
fichero = fopen("nombre","w");
int edad = 7;
fprintf(fichero, "Pedro tiene %d meses \n", edad);
```

[1] Se hace así por eficiencia, si tratamos de leer o escribir datos de longitud pequeña, el manejo asociado a esas operaciones puede ser costoso en tiempo y es más eficaz escribir en memoria estas variables y después pasar de un solo golpe toda esa memoria al fichero.

Para la lectura de ficheros, como se intuye, está la función `fscanf`, con la cual hay que tener las mismas precauciones que con `scanf` en lo que se refiere a la lectura de strings y caracteres. Debes saber que las funciones `scanf` filtran la información que quiere obtener y dejan en el buffer de entrada los caracteres separadores. Por ejemplo, si leemos don enteros separados por espacios (tabuladores o retornos de carros), éstos separadores se quedarán en el buffer; si siempre leemos enteros no habrá problemas, pero si leemos caracteres lo que conseguiremos serán estos separadores. También, lo que toman son las direcciones donde dejar la información, no las propias variables, por lo que se deberá utilizar el operador de dirección "&".

```
fichero = fopen("nombre","r");
int edad, *punt_entero;
fscanf(fichero, "%d", &edad);
fscanf(fichero, "%d", punt_entero);
```

Por la misma razón que existen las funciones `getchar` y `putchar`, también existen sus equivalentes `getc` y `putc` para ficheros, a las cuales habrá que darlas como último argumento el puntero a fichero.

```
ch = getc(fichero);
putc(ch, fichero);
```

Como ejemplo de utilización de ficheros, vamos a ver cómo con las dos funciones de caracteres podemos hacer un “vuelca el contenido” de un fichero (el fichero si es de lectura debe existir):

```
#include <stdio.h>
int main ()          /* este programa podria ser el commando cat */
{
       FILE *entrada;    /* fichero de entrada */
       int ch;
       /* abre fichero para lectura que se llama nombre */
       if ((entrada = fopen("nombre", "r")) != NULL)
       {
              /* lee hasta final de fichero, EOF es una constante definida
               en stdio.h que se utiliza como final de fichero */
              while ((ch = getc(entrada)) != EOF)
                     /* stdout es la pantalla */
                     putc(ch, stdout);
              fclose(entrada);
       }
       else
              perror("No se puede abrir");
}
```

Por último, al igual que la entrada/salida de *strings*, también existe la equivalencia para ficheros; estas funciones son `fgets` y `fputs`. La primera tiene como primer argumento el puntero a carácter (tira de caracteres) donde se quiere realizar la lectura, después el número de caracteres que se quieren leer y por último el puntero a fichero. La segunda tiene como primer argumento la tira (puntero) de caracteres que se quiere escribir y como último argumento el puntero a fichero.

Además, `fgets` es de tipo puntero a carácter y devuelve un `NULL` si encuentra el `EOF` y a diferencia de `gets` no convierte el carácter '\n', y `fputs` devuelve un `EOF` si hay un error, quita el nulo del final, pero no añade el carácter '\n'.

Su uso sería:

```
fichero = fopen("nombre","r");
(fgets(tira, 81, fichero) != NULL);
fichero = fopen("nombre","a");
control = fputs("Ya era hora", fichero);
```

Hay otras funciones menos importantes para trabajar con ficheros (ver apéndice):

- `rewind()` Coloca el índice del fichero al comienzo del mismo y toma como argumento un puntero a fichero.
- `remove()` Borra un fichero y también toma como argumento el fichero.
- `rename()` Toma como argumentos dos tiras de caracteres con el nombre viejo y el nuevo y cambia el nombre.
- `feof()` Nos dice si para el fichero indicado se ha llegado a su final.
- `ferror()` Nos dice si para el fichero indicado ha habido algún error.
- `fflush()` Para el fichero indicado fuerza que el buffer interno se vuelque al fichero.
- `fread()` Lecura de un fichero en modo bloque.
- `fwrite()` Escritura de un fichero en modo bloque.
- `fseek()` Sirve para moverse por el fichero. Tiene tres argumentos: un puntero a fichero, un offset de tipo *long* (distancia donde se busca, positiva o negativa) y un modo de búsqueda igual a su equivalente `lseek`. La función devuelve un 0 si todo va bien.

```
fseek(fichero, 32L, 1);
```

3) Realiza un programa para crear un fichero. El tiempo de creación del fichero debe ser de aproximadamente dos segundos. Esto lo puedes conseguir con el comando `$time ejecutable` (se suele escoger la suma del tiempo de usuario más el tiempo de sistema). El nombre se lo puedes dar por cabecera (argc / argv) y el tamaño se lo puedes parametrizar con la opción del compilador –D.

4) Desarrolla un programa que permita copiar el contenido del fichero creado en otro fichero. El programa tomará los nombres de los dos ficheros: origen y destino por línea de comando (cabecera): `$ programa origen destino`. En la primera versión lo harás utilizando las llamadas a la librería de C usando bytes (char).

5) En la segunda versión lo harás usando las llamadas al sistema puras. Se recomienda usar open con tres argumentos. Varia el tamaño de la copia (tu variable) con tamaños de 2, 8, 16, 256, 1024 y 8192 bytes (lo puedes hacer también con la opción -D). Mide lo que tarda en hacer la copia y compáralo con la versión anterior.

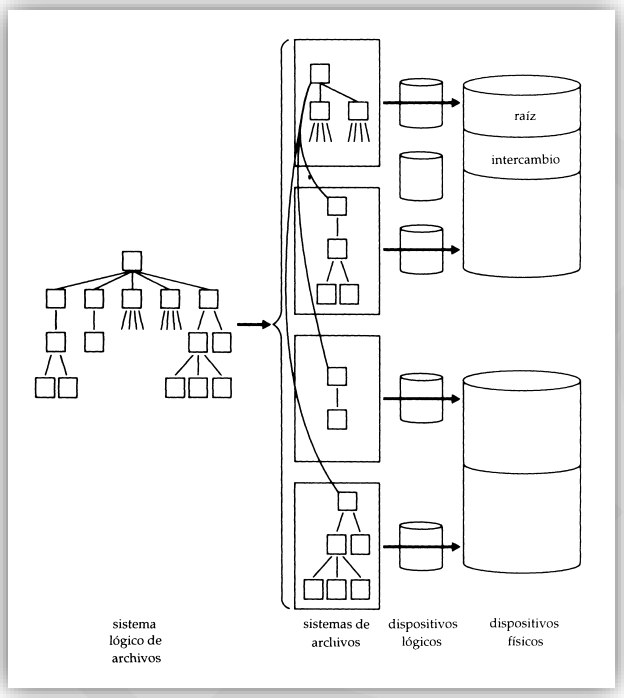
Estructura interna (solo información)

Ya se vio como era la estructura de ficheros desde el punto de vista del usuario y sus principales características. Esa estructura de ficheros es lo que se llama la estructura *lógica* de ficheros.

Pero esa estructura lógica de ficheros puede estar constituida por varias estructuras físicas de ficheros que pueden incluso estar soportadas en distintos dispositivos de almacenamiento masivo, cada uno con sus propias características. Es lo que llamábamos el proceso de montaje de un sistema de ficheros.

Puesto que las características de los dispositivos difieren entre sí, cada dispositivo hardware define su propio sistema de ficheros físico. De hecho, es deseable partir los grandes dispositivos físicos en múltiples dispositivos lógicos por razones de eficiencia y seguridad.

En la figura se muestra un ejemplo de partición de una estructura lógica de ficheros en diferentes sistemas de ficheros. En el mismo existen dos dispositivos físicos de almacenamiento diferentes, uno principal y otro auxiliar, los cuales a su vez están partidos en cinco dispositivos lógicos, cada uno con unas características diferentes (por razones de eficiencia debido al uso a que se destinan).
Cada uno de estos dispositivos lógicos tiene un sistema de ficheros asociado. El primero de ellos contendrá el directorio raíz, los restantes sistemas estarán montados a partir de él. El sistema operativo almacena está estructura en una tabla que se llama tabla de montaje (*mount table*) para saber a qué dispositivo se tiene que acceder cuando se recorre el sistema de ficheros lógicos.



La partición de un dispositivo en varios sistemas de ficheros tiene varias ventajas, entre las que pueden destacarse las siguientes:

- Distintos sistemas de ficheros pueden soportar distintos usos. En el ejemplo una de ellas será utilizada como área de intercambio (swap) para la memoria virtual.
- La fiabilidad queda reforzada puesto que un error software solo puede provocar daños en un sistema de ficheros y no en todo el dispositivo.
- Puede mejorarse la eficiencia del sistema variando, para cada sistema de ficheros y según su uso, parámetros tales como tamaño del bloque, o el número de ficheros.
- Además, sistemas de ficheros independientes impiden que un programa utilice todo el espacio disponible para un fichero muy grande, puesto que su tamaño no puede extenderse fuera del sistema de ficheros.
- Es simple añadir un nuevo dispositivo, por ejemplo, otro disco.

Cada sistema de ficheros está constituido por unas tablas de control y por un conjunto de bloques lógicos de datos con un tamaño fijo (múltiplo de 512 bytes) en todo el sistema, donde se almacenan todos los ficheros de esa partición. Como hemos dicho este tamaño puede variar en cada sistema de ficheros, dependiendo del uso que se haga del mismo[1]. En los antiguos sistemas operativos Unix era de 512 bytes, en la versión system V es de 1024 bytes (1K) y en la BSD de 4K en adelante.

[1] Aunque no aparezca en la Figura , en la mayoría de sistemas hay una capa intermedia entre el dispositivo lógico (sistema de ficheros) y el dispositivo físico y es el *buffer cache* común a todos los sistemas de ficheros. Cada vez que se accede a uno de los bloques de la partición, éste se guarda en memoria (buffer cache), de tal manera que si se necesita otra vez (principio de localidad) se acceda a él en memoria y no en disco.

La estructura de cada sistema de ficheros está compuesta de cuatro grandes partes:

- Bloque de arranque (boot). Normalmente está en el primer sector del sistema de ficheros y contiene un pequeño programa que se encarga de buscar el sistema operativo y cargarlo en memoria.
- El superbloque. Es el bloque que describe el sistema de ficheros dando información sobre su tamaño, lista de bloques disponibles, primer bloque libre, número de iodos, lista de inodos libres, índice del primer inodo libre, flag de modificación, etc.
- La lista de inodos. Es una lista donde se guarda una entrada (inodo) por cada fichero, donde se almacena información respecto del mismo, como situación del mismo, propietario, permisos, fecha y hora de modificación, etc.
- Bloque de datos. Es la zona donde se almacenan los ficheros del sistema y a la cual hacen referencia los inodos. Como hemos dicho tienen un tamaño fijo en cada sistema de ficheros y sólo pueden estar ocupados por un sólo fichero, aunque éste no ocupe toda su longitud.

Aunque los ficheros estén contenidos en bloques de datos, a diferencia de otros sistemas operativos, estos no están organizados en registros o bloques, por lo que el usuario, para su manejo sólo necesita su dirección de origen y la longitud del mismo. Pero si éste fuera el funcionamiento interno del sistema, supondría tener todos los bloques de un fichero contiguos, lo cual provocaría dos problemas debido a la expansión dinámica que pueden sufrir: primero habría que mover los bloques de los ficheros que crecen y no tienen sitio donde están ubicados; y segundo, haría que la zona de bloques se llenara de huecos donde sólo cabrían ficheros de ese tamaño.

Para evitar estos problemas lo que se hace es guardar en cada entrada de inodos la localización de los bloques del mismo, además de toda una serie de información para el control y la gestión del fichero, esta información es la siguiente:

- La identificación del usuario propietario.
- La identificación del grupo al que pertenece el usuario.
- El tipo de fichero que puede ser regular o directorio, dispositivo (especial de bloque o carácter) o tubería (FIFO).
- Los permisos de acceso de lectura, escritura y ejecución.
- Los tiempos de último acceso, última modificación y cambio en el inodo.
- El número de enlaces al fichero (número de nombres). El nombre de los ficheros no está almacenado en el inodo del fichero sino en el del directorio.
- La dimensión del fichero en bytes.
- Un indicador de si el directorio tiene montado un sistema de ficheros.
- En la versión System V, 13 punteros, donde los 10 primeros corresponden a los primeros 10 bloques donde están almacenados los datos y los tres restantes apuntan a otros bloques donde están las direcciones de otros bloques para el acceso indirecto.

La necesidad de los punteros a los bloques es por la variedad de tamaños que puede tener un fichero. Así, si tenemos bloques de 1K podemos almacenar ficheros directamente de 10K con 10 punteros, pero necesitaríamos 100 punteros para almacenar un fichero de 100K, con lo que el tamaño del inodo sería muy grande.

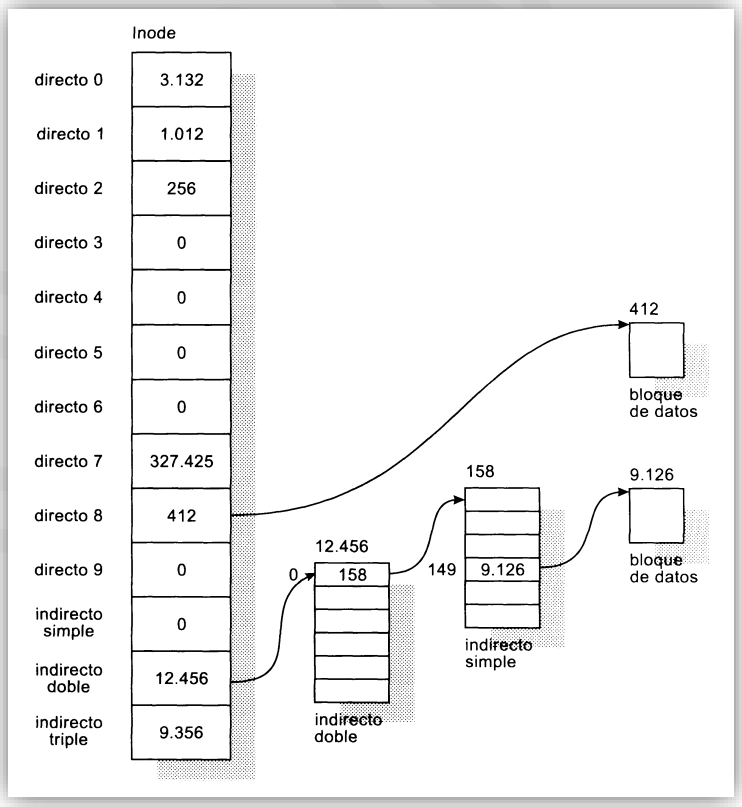
Por eso surge la necesidad de tener otra forma de acceder a los bloques de los ficheros, la primera forma es la directa, donde se almacenan en el inodo las diez primeras direcciones de los primeros diez bloques, la segunda forma es la indirecta, para la cual se guardan otros tres punteros, el primer puntero señala a una tabla de direcciones de bloques, el segundo señala a una tabla de tablas de direcciones de bloques (acceso doble) y el tercero a una tabla de tablas de acceso doble (acceso triple).

La capacidad suponiendo bloques de 1 K y por tanto tablas de 256 direcciones, está en la tabla siguiente. Hay que tener en cuenta que, si el espacio del tamaño del fichero es de 32 bits, el fichero no podrá tener más de 4 Gbytes, lo cual resulta más que suficiente. Imaginad para 64 bits.

Entrada	Bloques	Bytes
10 entradas directas	10	10 K
1 entrada simple	256	256 K
1 entrada doble	256 x 256	64 M
1 entrada triple	256 x 256 x 256	16 G

De esta manera, con los punteros directos podemos acceder a los primeros 10 bloques de un fichero (0 al 9), con el puntero de direcciones simple podemos acceder a los 256 bloques siguientes, es decir desde el bloque 10 al bloque 265 (inclusive), con el puntero doble desde los bloques 266 al 65.801 (inclusive), y con el triple desde el 65.802 hasta el final.

Como ejemplo de utilización supongamos que queremos acceder al byte 9.125. Lo primero que tenemos que hacer es dividir este número entre el tamaño del bloque, para saber en qué bloque accederemos a ese byte, en nuestro ejemplo al 8, después sólo tenemos que utilizar el offset en ese bloque (el resto de la división) para acceder al byte deseado, que en este caso es el 933.



Si quisiéramos acceder al byte 425.000, haríamos lo mismo, con lo cual tendríamos que acceder al bloque 415, que pertenece al direccionamiento doble. Iríamos a la dirección de la tabla de dobles y miraríamos la primera dirección de la tabla de direcciones simple ya que en ella está la 415. Dentro de

esa tabla calculamos el offset relativo de la 415 (415-266) que da 149, con lo que nos iremos a la posición 149 de esa tabla simple que nos dará la dirección del bloque 415, dentro del cual nos moveremos para encontrar el byte pedido con el offset de la primera división.

En el inodo pueden aparecer punteros a cero, esto significa que hay dentro del fichero bloques sin asignar y por tanto están vacíos. Este sería el caso en que se crea un fichero y se escribe en la posición un millón, dejando el resto libre. No se tendrán que reservar mil bloques de 1 K, sino sólo un bloque que se direccionará con el puntero de direccionamiento triple.

Según este esquema de funcionamiento, los ficheros que tengan menos de 10 bloques de tamaño serán muy rápidos de localizar, penalizándose la utilización de ficheros muy grandes. Por eso es muy importante, según el tamaño de ficheros que se use, elegir un buen tamaño de bloque de datos, si es pequeño se gastará mucho tiempo en direccionamientos (dobles y triples) y si es muy grande se gastará mucho espacio de disco en ficheros pequeños.

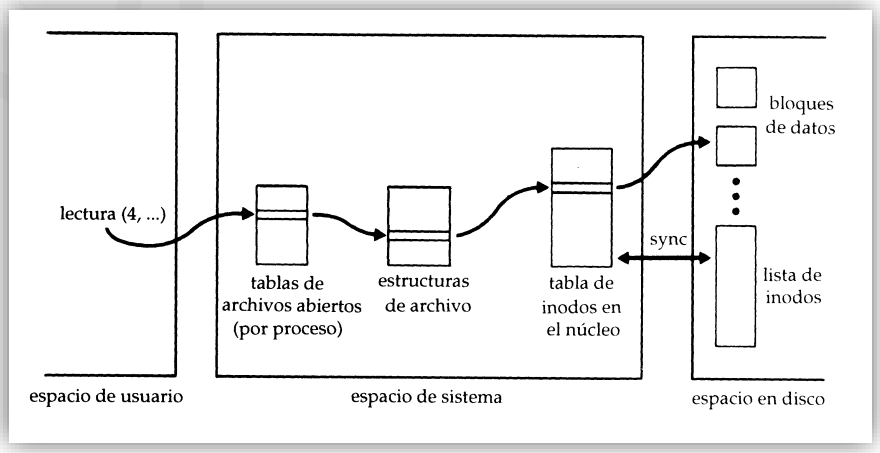
Por eso en la versión BSD se utilizan bloques de 8K, y para evitar en parte el desperdicio de disco se utiliza otro parámetro, también único en el sistema lógico de ficheros, que es el fragmento, que suele tener un octavo del tamaño del bloque. Así, un fichero puede tener un número variable de bloques y un sólo fragmento que se encargará de recoger lo que no quepa en los bloques.

Otra mejora del sistema de Berkeley es la distribución de los inodos en el sistema de ficheros, que ya no se hace al principio sino en un mismo grupo de cilindros del disco físico. Esto se hace así para que los inodos estén lo más cerca posible de sus bloques de datos y con una pasada de las cabezas lectoras del dispositivo se puedan leer inodos y bloques a la vez.

Debido a que las operaciones sobre disco son bastante pesadas en tiempo de ejecución, lo que se hace normalmente es tener todas las tablas necesarias para el acceso de un fichero en memoria, es decir, el superbloque y los inodos. Esto puede plantear problemas de coherencia de datos al haber dos versiones de estas tablas, una en memoria y otra en disco (ese es el gran problema de apagar el sistema de repente, que se pierde el superbloque y los inodos). Por eso, para evitar problemas de coherencia (no todos) existe un proceso del sistema operativo que se despierta periódicamente y que hace un volcado en disco de la versión de memoria (operación sync). A estos procesos periódicos del sistema operativo se les llama demonios (daemons), en este caso su nombre es syncer[1].

En un sistema multiusuario / multiproceso puede ocurrir que varios procesos estén accediendo en distintas operaciones a un mismo fichero a la vez, y tengan distintos punteros de lectura/escritura por donde están recorriendo el mismo.

Esto hace necesario mantener otras dos tablas de control, una pertenecerá al proceso (PCB) y será de nivel usuario, en ella se almacenarán los ficheros que tiene abiertos ese proceso, y recibe el nombre



[1] A esta tabla en memoria se le suele denominar cache de inodos.

de tabla de descriptores (un descriptor es un número que sirve de identificación del fichero por el proceso). En cada entrada de esta tabla habrá un puntero hacia una entrada de la siguiente tabla.
La otra pertenecerá al núcleo, se llama tabla de ficheros y tiene una entrada por cada fichero abierto del sistema. En cada entrada se guarda información sobre las operaciones que se pueden realizar sobre el fichero, el inodo que lo describe, los punteros de lectura/escritura, permisos del fichero, y un contador de cuantos descriptores tiene asociada esa entrada.

De esta manera, cuando un proceso quiere realizar una operación sobre un determinado fichero, sólo le tiene que dar al kernel el descriptor del fichero (índice en la tabla). El buscará en la tabla de descriptores del proceso (que tiene un tamaño limitado definido por el administrador para cada usuario) el puntero a la tabla de ficheros, y en ella el inodo que describe ese fichero.

Existen por defecto asociados a cada proceso tres descriptores que ya hemos visto en el apartado de redirección de la shell. El descriptor 0 indicará la entrada estándar (normalmente teclado), el 1 la salida estándar (pantalla) y el 2 la salida de errores. En una redirección lo que se hace es cambiar estos descriptores por los del fichero señalado.

Utilización de ficheros en el ambiente multiusuario

Los ficheros no están completamente especificados simplemente por los datos que ellos contienen. Cada fichero Linux-Unix posee un número de propiedades necesarias para la administración en un entorno multiusuario.

Cada fichero Linux-Unix es propiedad de uno de los usuarios del sistema, que normalmente, es el que crea el fichero. La identidad del propietario está representada por un entero no negativo denominado *user-id* que es almacenado por el sistema cuando se crea el fichero.

Cada proceso Linux-Unix está, normalmente, asociado con el user-id del usuario que lo arrancó. Y cuando un proceso crea un fichero, el sistema crea una relación de propiedad entre el fichero y el proceso a través del user-id. La propiedad de un fichero puede ser cambiada posteriormente, bien por el superusuario (este siempre tiene como nombre de usuario root y user-id 0) o por el propietario del fichero.

Junto al user-id, un fichero cuando es creado también toma como group-id, el del proceso que lo creó que también es tomado del usuario que creó el proceso.

Existen otros tres tipos de permisos con especial relevancia cuando el fichero contiene un programa ejecutable:

- El permiso set-user-id (s): cuando un proceso arranca un programa contenido en un fichero con este permiso activo, el sistema le da, al proceso resultante un user-id efectivo tomado del propietario del fichero en lugar del del usuario que arrancó el proceso. Esto puede entrañar algún riesgo de seguridad y por eso muchos sistemas ignoran este bit en shell-scripts.
- El permiso set-group-id, posee características similares al anterior, pero para el group-id.
- El permiso save-text-image, más conocido como el sticky bit, está diseñado para ahorrar tiempo de acceso a programas muy utilizados. Así, si este permiso está activo, el fichero permanecerá en el área de intercambio. Actualmente, los discos y sistemas de ficheros son más eficientes, y se ha cambiado su significado. Solo se usa en directorios para indicar que solo el propietario del directorio puede renombrar o borrar ficheros, aunque el directorio tenga permisos de “w”.

Desde un programa se pueden cambiar los permisos y el propietario de un fichero a través de dos llamadas al sistema, de la misma manera que existen los dos comandos de la shell. En cualquier caso, sólo pueden ser utilizados por el propietario del fichero o por el superusuario.

La utilización de estas llamadas sería:

```
int retval1, retval2, nuevomodo, id_prop, id_grupo;
char  *camino;
 .
 .
retval1 = chmod(camino, nuevomodo);
 .
retval2 = chown(camino, id_prop, id_grupo);
```

Utilización de directorios

Hemos visto que en Linux-Unix los directorios y los dispositivos son tratados como si fueran ficheros, aunque de una forma especial. Desgraciadamente, la utilización de directorios y dispositivos depende en gran medida de la implementación del sistema operativo utilizado.

Los directorios son un conjunto de ristras de 16 bytes, donde cada una contiene en los primeros dos bytes el número del inodo del fichero que está dentro del directorio, y en los 14 siguientes el nombre del mismo en formato ascii. De esta manera cuando hacemos un link de un fichero lo que estamos haciendo en repetir el número de inodo de un fichero existente y ponerle otro nombre.

122	f	i	c	h	e	r	o	1	\0				
243	f	i	c	h	e	r	o	2	\0				
654	o	t	r	o	f	i	c	h	e	r	o	\0	
122	f	i	c	h	e	r	o	4	\0				

Y cuando borramos un fichero dentro de un directorio, lo que se hace como primer paso es poner a cero los dos primeros bytes que contienen el inodo, con lo cual esa entrada en el directorio se puede volver a utilizar, y como segundo paso liberar los bloques que contienen el fichero (si es que no es un fichero con link). Por esta razón en Linux-Unix los ficheros no se pueden recuperar (*undelete*).

122	f	i	c	h	e	r	o	1	\0				
0	f	i	c	h	e	r	o	2	\0				
654	o	t	r	o	f	i	c	h	e	r	o	\0	
122	f	i	c	h	e	r	o	4	\0				

A su vez un directorio puede contener otros subdirectorios que se tratan de nuevo de la misma manera que ficheros, en particular existen dos subdirectorios especiales que están en todos los directorios, que son el "." y el "..", que lo único que hacen es enlazar el directorio actual y el directorio padre con sus nombres.

132	.	\0											
123	.	.	\0										
243	d	i	r	e	c	t	o	r	i	o	\0		
122	f	i	c	h	e	r	o	1	\0				

Como un fichero más, los directorios también tienen permisos de acceso, el de lectura significa que se puede acceder a los nombres de ficheros que contiene ese directorio, el de escritura permite al usuario crear nuevos ficheros en ese directorio, y el de ejecución permite a un usuario moverse por los directorios con el comando `cd`. Podremos ver esos permisos utilizando el comando `ls -l`:

```
$ ls -l
-rw-r-----   1   pepe    other     43244   Oct 12 21:12  trabajo
drwxr-x---  2    pepe    other        32   Oct 12 22:00  direc
```

El que sea un directorio lo podemos comprobar porque la primera letra de la línea es una "d". Para los ficheros normales será una "-", para las FIFOS, que se verán después, será una "p", y para los dispositivos una "c" o un "b".

Para utilizar los directorios desde un programa C se utiliza una estructura definida en el encabezamiento <sys/dir.h> el cual utiliza tipos de datos definidos en <sys/types.h>, su declaración es:

```
#define  DIRSIZ    14
struct direct {
   ino_t      d_ino;
   char d_name[DIRSIZ];
};
```

Donde están definidas las dos partes de la estructura interna del directorio, el inodo y el nombre del fichero.

Con los directorios se pueden utilizar las llamadas vistas open y read para abrir el directorio y poder leer el nombre de los ficheros que contiene. Esto se puede ver en el siguiente ejemplo donde se escribe una función C para leer un directorio:

```
#include   <sys/types.h>
#include   <sys/dir.h>
#include   <fcntl.h>

directorio(nombre)
char  *nombre;
{
          struct direct dir;
          int descriptor;

          /* se abre el directorio */
          if((descriptor = open(nombre, O_RDONLY)) < 0)
               return (-1);
          /* se leen los ficheros que contienen */
          while(read(descriptor, (char *)&dir,
                                        sizeof(dir))==sizeof(dir))
               if(dir.d_ino != 0)
                    printf("%.14s \n", dir.d_name);
          close(descriptor);
          return(0);
}
```

como se ve, se ha abierto el directorio de la forma usual, devolviéndonos la función un descriptor del mismo. Después se han empezado a leer las ristras de 16 bytes que contienen la información de los ficheros, para lo cual se ha utilizado la función `read` (no se para de leer hasta que devuelva algo distinto del tamaño de la ristra) y por último se representa en pantalla sólo el nombre de los ficheros que tengan su descriptor distinto de 0. Como simplificación, se puede decir que los directorios sólo se pueden abrir para lectura, por lo que no se podrán utilizar con la función `write`.

A parte de las funciones normales vistas, existen otras especiales para la creación y encaminamiento de directorios. La primera función es `chdir` que toma como argumento un camino hacia un directorio y coloca a éste como directorio por defecto:

```
char *camino;
int  retorno;
...
retorno = chdir(camino);
```

Para crear los directorios se utiliza la función `mknod`, que es más general y también se utiliza para crear FIFOS (se verán posteriormente) y dispositivos. La función toma dos argumentos, un nombre de directorio (si se quiere con su camino completo) y un modo de apertura, similar al de la función `creat`. La diferencia con éste, es que el permiso de creación se tiene que completar con otros bits que le indiquen al sistema que se está creando un directorio, estos bits en octal son: 040000 (está definido en el encabezamiento `<sys/stat.h>` como `S_IFDIR`). Así, si queremos crear un directorio con permisos de lectura, escritura y ejecución para todos los usuarios tendríamos que escribir 040777. Hay que tener en cuenta que esta función no crea los directorios "." y "..", por lo que se tendrán que crear con link si queremos que el directorio creado sea válido. La declaración es:

```
int retorno, modo;
char *camino;
...
retorno = mknod(camino, modo);
```

Debido a las limitaciones mencionadas anteriormente, y a que esta función sólo la puede utilizar el superusuario, en muchos sistemas existe la función `mkdir` que crea el directorio sin más problemas. También toma como argumentos un camino y un modo de creación, pero en este caso sólo incluyendo los permisos, no el 040000.

Otras funciones para la utilización de directorios que no se detallarán aquí son `rmdir` que elimina un directorio si éste sólo contiene "." y "..", `chdir` que cambia el directorio de trabajo, y `chroot` que cambia el directorio raíz.

Por último, igual que ocurre con los bloques de datos y con los inodos, el sistema mantiene una tercer cache llamada cache de directorios donde se guardan en memoria la información de los directorios recientemente usados, esto se puede comprobar al hacer un `ls` de un directorio no usado anteriormente, la primera vez tardará en darnos la información las posteriores veces no.

Utilización de dispositivos

En Unix, los dispositivos (discos, terminales, impresoras, cintas, etc.) también se representan mediante ficheros en el sistema de ficheros. A estos ficheros se les denomina ficheros especiales.

Las lecturas y escrituras a estos ficheros causan que los datos sean transferidos hacia (desde) el dispositivo periférico apropiado.

Típicamente, estos ficheros especiales se almacenan en el directorio denominado `/dev`:

```
/dev/tty00
/dev/tty01
/dev/lp
....
```

Estos ficheros especiales están divididos en dos grandes categorías: dispositivos de bloque y de carácter.

Los ficheros especiales de bloque incluyen dispositivos tales como discos y cintas magnéticas. La transferencia de datos entre estos dispositivos y el kernel ocurre en bloques de dimensión estandarizada y todos los dispositivos de bloque aceptan acceso aleatorio. Es importante señalar que los sistemas de ficheros sólo pueden existir sobre dispositivos de este tipo.

Los ficheros especiales de carácter incluyen dispositivos tales como un terminal, impresoras, etc., que no comparten el mecanismo de transferencia tan estructurado como los de bloque. El acceso *random* puede o no estar soportado por el dispositivo.

El Linux-Unix utiliza dos tablas de configuración del sistema operativo denominadas las tablas de dispositivos de bloque y la de dispositivos de carácter para asociar un dispositivo periférico con el código específico necesario para su manejo.

Estas tablas son mantenidas dentro del propio kernel e indexadas usando un entero denominado el número de dispositivo mayor (*major device number*) que se almacena en el inodo del fichero especial.

Además del número mayor, también se almacena en el inodo un segundo valor entero denominado el número de dispositivo menor (*minor device number*). Este número se le pasa al driver para indicarle exactamente a qué puerta se desea acceder para el caso de aquellos dispositivos que soportan más de una.

Ambos números se pueden ver al hacer un ls -l en el directorio /dev, estos aparecerán en lugar de la longitud del fichero.

Llamadas ANSI C con ficheros

En la tabla que aparece a continuación se describen muchas de las llamadas utilizadas para usar ficheros con buffer (*stream*), todas pertenecen a ANSI C.

Tipo	Función	Argumentos
FILE *f	fopen (char string, char* string)	Nombre del fichero, Tipo de apertura: “r”, “w”, “a”, “r+”
int	fclose (FILE *f f)	Puntero a FILE definido en stdio.h
int	fprintf (FILE *f f, string de control , variables)	Fichero f, string con mensaje y conversores, lista de variables a escribir
int	fcanf (FILE *f, string de control , variables)	Fichero f, string con conversores, lista de variables a leer
int	getc (FILE *f)	Fichero f donde leer
int	putc (int entero, FILE *f)	Entero (char) a escribir en fichero f
char *	fgets (char *string, int entero, FILE *f)	String a leer de *entero* caracteres del fichero f
int	fputs (char *string, FILE *f)	String a escribir en el fichero f
size_t	fread (char * dir, size_t, size_t, FILE *f)	Dirección dir donde colocar los datos compuestos de size_t bloques de tamaño size_t del fichero f
size_t	fwrite (char *dir, size_t, size_t, FILE *f)	Dirección dir donde encontrar los datos a escribir (ver anterior)
int	fflush (FILE *f)	Fichero f de donde hacer el volcado del buffer interno (stream)
long_off	fseek (FILE *f, long_off offset, int entero)	Del fichero f, irse a la posición offset desde entero (0, 1, 2)
int	ferror (FILE *f)	Fichero f donde puede haber un error
int	feof (FILE *f)	Fichero f donde podemos haber alcanzado la marca de final de fichero (eof)