3. Manejo de Archivos

3.1 Introducción

En el tema anterior hemos delineado la estructura interna del sistema de archivos de Unix. En este tema vamos a estudiar la interfaz que ofrece el sistema para comunicarnos con el kernel y poder acceder a los recursos del sistema de archivos. Más concretamente, la comunicación se va a realizar con una parte del kernel, el subsistema de archivos o subsistema de entrada/salida. Trabajaremos con los archivos aprovechando la estructura de alto nivel que ofrece el sistema de archivos y que permite realizar una abstracción de lo que es el soporte físico de la información (discos).

Para empezar, vamos a ver un ejemplo sencillo que ilustra cómo se comunica un programa con el subsistema de archivos. El programa siguiente es una versión simplificada de la orden `cp`, que permite copiar archivos. Lo llamaremos `copiar`.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>

char buffer[BUFSIZ];

int main(int argc, char *argv[]) {
	int fd_origen, fd_destino;
	int nbytes;

	if (argc != 3) {
		fprintf(stderr, "Forma de uso: %s archivo_origen archivo_destino\n",
argv[0]);
		return -1;
	}

	if ((fd_origen = open(argv[1], O_RDONLY)) < 0) {
		fprintf(stderr, "Error con el archivo: %s", argv[1]);
		return -1;
	}

	if ((fd_destino = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT, 0666)) < 0) {
		fprintf(stderr, "Error con el archivo: %s", argv[2]);
		return -1;
	}

	while ((nbytes = read(fd_origen, buffer, sizeof(buffer))) > 0) {
		write(fd_destino, buffer, nbytes);
	}

	close(fd_origen);
	close(fd_destino);
	return 0;
}
```

Este programa tan sencillo muestra el esquema general que se debe seguir para trabajar con archivos. La secuencia recomendada es:

- Abrir los archivos (llamada `open`) y comprobar si se produce algún error en la apertura. En caso de error, open devuelve el valor de −1.
- Manipular los archivos de acuerdo a nuestras necesidades. En este ejemplo leemos del archivo origen y escribimos en el archivo destino. Hay que hacer notar que la lectura/escritura se realiza en bloques de tamaño `BUFSIZ` (constante definida en `<stdio.h>`). La forma de detectar que hemos leído todo el archivo origen es analizando el valor devuelto por `read`. Si este valor es igual a cero, significa que ya no quedan más datos que leer del archivo origen.
- Cerrar los archivos una vez que hemos terminado de trabajar con ellos.

Para el kernel, todos los archivos abiertos son accedidos mediante descriptores. Un descriptor es un número entero no negativo. Cuando nosotros abrimos un archivo existente o creamos uno nuevo, el kernel nos regresa un descriptor para el programa (o proceso). Cuando queremos leer o escribir un archivo, nosotros usamos el archivo con el descriptor (que nos ha devuelto `open` o `creat`) como un parámetro de las funciones de `read` o `write`.

Por convención, los shells de Unix asocia el descriptor `0` a la entrada estándar de un programa, el descriptor `1` con la salida de datos estándar y el descriptor 2 con la salida de errores estándar. Esta convención es utilizada por shells y muchas aplicaciones; esto no es una característica del kernel de Unix. Sin embargo, muchas aplicaciones pueden tener problemas al ejecutarse si no se siguen estas asociaciones. Los números mágicos `0`, `1` y `2` pueden ser reemplazados con las constantes `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`. Estas constantes están definidas en el archivo `<unistd.h>`. El rango para los descriptores va desde `0` a `OPEN_MAX`. Históricamente el máximo de archivos permitidos por un proceso era de 20, pero muchos sistemas han incrementado este número hasta 63.

También suele ser convenio bastante extendido que un programa, cuando termina satisfactoriamente, devuelva el valor 0 al sistema operativo. Este valor es almacenado en la variable de entorno `$?`, que puede ser analizada por otro proceso para ver el código de error devuelto por el último proceso que la modificó. Algunas utilidades, como `make`, emplean estos códigos de error para determinar si deben proseguir su ejecución o deben detenerse. Desde la línea de instrucciones podemos ver el contenido de `$?` escribiendo:

```
$ echo $?
```

Para los códigos devueltos en caso de error no existe un criterio concreto, salvo el de que sean distintos de 0.

3.2 Entrada y Salida sobre Archivos Ordinarios

En los siguientes párrafos vamos a estudiar las llamadas al sistema necesarias para realizar entrada/salida sobre archivos ordinarios.

Función open

`open` es la función que utilizaremos para indicarle al kernel que habilite las instrucciones necesarias para trabajar con un archivo que especificaremos mediante una ruta. El kernel devolverá un descriptor de archivos con el que podremos referenciar al archivo en funciones posteriores. La declaración de open es:

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

Mostramos el tercer argumento como …, es el formato que utiliza ISO C para especificar que el número y tipos de los argumentos restante puede variar. Para esta función, el tercer argumento es usado solo cuando un nuevo archivo está siendo creado, como veremos más tarde.

`pathname` es la ruta del archivo que queremos abrir. Puede ser una ruta absoluta o relativa y su longitud no debe exceder de `PATH_MAX` bytes.

`oflags` es una máscara de bits (varias opciones pueden estar presentes usando el operador OR a nivel de bits) que le indica al kernel el modo en que queremos que se abra el archivo. Uno de los bits, O_RDONLY, O_WRONLY u O_RDWR, y sólo uno, debe estar presente al componer la máscara; de lo contrario, el modo de apertura quedaría indefinido. Los `oflags` más significativos que hay disponibles son:

Bandera	Descripción
O_RDONLY	Abrir en modo sólo lectura.
O_WRONLY	Abrir en modo sólo escritura.
O_RDWR	Abrir para leer y escribir.
O_APPEND	El apuntador de la lectura/escritura del archivo se sitúa al final del mismo antes de empezar la escritura. Así garantizamos que lo escrito se añade al final del archivo.
O_CREAT	Si el archivo que queremos abrir ya existe, esta bandera no tiene efecto, excepto en lo que se indicará para la bandera O_EXCL. El archivo es creado en caso de que no exista y se creará con los permisos indicados en el parámetro mode.
O_EXCL	Genera un error si O_CREAT también está especificado y el archivo ya existe. Esta bandera es usada para determinar si el archivo no existe y crearlo en caso de que así sea, en una operación atómica.
O_TRUNC	Si el archivo ya existe, trunca su longitud a cero bytes, incluso si el archivo se abre para leer.
O_NDELAY	Esta bandera afectará las futuras operaciones de lectura/escritura. En relación con O_NDELAY, cuando abrimos una tubería con nombre y activamos el modo O_RDONLY u O_WRONLY: • Si O_NDELAY está activo, un open en modo sólo lectura regresa inmediatamente. Un open en modo sólo escritura devuelve error si en el instante de la lectura no hay otro proceso que tenga abierto la tubería en modo sólo lectura. • Si O_NDELAY no está activo, un open en modo sólo lectura no devuelve

	el control hasta que un proceso no abre la tubería para escribir en ella. Un open en modo sólo escritura no devuelve el control hasta que un proceso no abre la tubería para leer de ella. Si el archivo que queremos abrir está asociado con un socket: • Si O_NDELAY está activo, open regresa sin esperar por la portadora (llamada no bloqueante). • Si O_NDELAY está inactivo, open no regresa hasta que detecta la portadora (llamada bloqueante).
O_NONBLOCK	Si la ruta se refiere a un FIFO, archivo de acceso por bloques especial, o a un archivo de acceso carácter especial, esta opción establece un modo de no bloqueo tanto para la apertura del archivo como para cualquier operación de entrada/salida.

Estas son las banderas más comunes para open.

`mode` es el tercer parámetro de open y sólo tiene significado cuando está activa la bandera `O_CREAT`. Le indica el kernel qué permisos queremos que tenga el archivo que va a crear. `mode` es también una máscara de bits y se suele expresar en octal, mediante un número de 3 dígitos. El primero de los dígitos hace referencia a los permisos de lectura, escritura y ejecución para el propietario del archivo; el segundo se refiere a los mismos permisos para el grupo de usuarios al que pertenece el propietario, y el tercero se refiere a los permisos del resto de usuarios. Así por ejemplo, `0644` (`110 100 100`) indica los permisos de lectura y escritura para el propietario, y permiso de lectura para el grupo y para el resto de los usuarios.

Si el kernel realiza satisfactoriamente la apertura del archivo, `open` devolverá un descriptor de archivo. En caso contrario, devolverá `-1` y en la variable `errno` pondrá el valor del tipo de error producido.

3.2.1 Función creat

La función `creat` permite crear un archivo ordinario o reescribir sobre uno existente. Su declaración es:

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

`pathname` es la ruta del archivo que queremos crear.

`mode` es una máscara de bits con el mismo significado que vimos en la llamada `open`. En esta máscara se especifican los permisos de lectura, escritura y ejecución para el propietario, grupo al que pertenece y el resto de los usuarios.

Si `creat` se ejecuta correctamente, devuelve un descriptor y el archivo es abierto en modo sólo escritura, incluso si `mode` no permite este tipo de acceso. Si el archivo ya existe, su tamaño es truncado a 0 bytes y el puntero de escritura se sitúa al principio. Si la llamada a `creat` falla; por ejemplo, si no tenemos los permisos para crear un archivo en el directorio en el que intentamos hacerlo, la función devolverá `-1` y en `errno` estará el código del tipo de error producido.

La función `creat` tiene la misma funcionalidad que una llamada a `open` en la que activemos las banderas `O_WRONLY | O_CREAT | O_TRUNC`. Así, las siguientes llamadas tienen la misma funcionalidad.

```
fd = creat("file.dat", 0666);
fd = open("file.dat", O_WRONLY | O_CREAT | O_TRUNC, 0666);
```

3.2.2 Función close

Utilizaremos la función `close` para indicarle al kernel que dejamos de trabajar con un archivo previamente abierto. El kernel se encargará de liberar las estructuras que había establecido para poder trabajar con el archivo. La declaración `close` es:

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

Si `filedes` es un descriptor de archivo correcto devuelto por una llamada a `creat`, `open`, `dup`, `fcntl` o `pipe`; `close` cierra su archivo asociado y devuelve el valor de 0; en caso contrario devuelve el valor de -1 y `errno` contendrá el tipo de error producido. El único error que se puede producir en una llamada a `close` es que `filedes` no sea un descriptor válido.

Al cerrar un archivo, la entrada que ocupaba en la tabla de descriptores de archivos del proceso queda libre para que la pueda utilizar una llamada `open`. Por otro lado, el kernel analizada la entrada correspondiente en la tabla de archivos de sistema y si el contador que tiene asociado este archivo es 1 (esto quiere decir que no hay más procesos que estén unidos a esta entrada), esa entrada también se libera.

Si un proceso no cierra los archivos que tiene abiertos, al terminar su ejecución el kernel analiza la tabla de descriptores y se encarga de cerrar los archivos que aún estén abiertos. Muchos programas toman ventaja de este hecho y no cierran explícitamente cualquier archivo abierto.

3.2.3 Función lseek

Con la función `lseek` vamos a modificar el apuntador de lectura/escritura de un archivo. Su declaración es la siguiente:

```
#include <unistd.h>
off_t lseek (int filedes, off_t offset, int whence);
```

`lseek` modifica el apuntador de lectura/escritura del archivo asociado a `filedes` de la siguiente forma:

- Si `whence` vale `SEEK_SET`, el apuntador avanza `offset` bytes con respecto al inicio del archivo.
- Si `whence` vale `SEEK_CUR`, el apuntador avanza `offset` bytes con respecto a su posición actual.
- Si `whence` vale `SEEK_END`, el apuntador avanza `offset` bytes con respecto al final del archivo.

Si `offset` es un número positivo, los avances deben entenderse en su sentido natural; es decir, desde el inicio del archivo hacia el final de este. Sin embargo, también se puede conseguir que el apuntador retroceda pasándole a `lseek` un desplazamiento negativo.

Cuando `lseek` se ejecuta satisfactoriamente, devuelve un número entero no negativo, que es la nueva posición del apuntador de lectura/escritura medida con respecto al principio del archivo. Si `lseek` falla, devuelve `−1` y en `errno` estará el código del error producido.

Debido a que una llamada exitosa regresa el nuevo desplazamiento del archivo, es posible invocar la función con un desplazamiento de 0 con el fin de determinar el desplazamiento actual del archivo:

```
off_t curr_pos;
curr_pos = lseek(fd, 0, SEEK_CUR);
```

Esta técnica también puede ser usada para determinar si es un archivo es capaz soportar desplazamiento. Hay que tener presente que en algunos archivos no está permitido el acceso aleatorio y por lo tanto la llamada a `lseek` no tiene sentido. Ejemplos de estos archivos son las tuberías con nombres y los archivos de dispositivo en los que la lectura se realice siempre a través de un mismo registro a posición de memoria.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
	if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1) {
		printf("cannot seek\n");
	} else {
		printf("seek OK\n");
	}
	return EXIT_SUCCESS;
}
```

```
$ ./a.out < /etc/motd
Seek OK
$ cat < /etc/motd | ./a.out
cannot seek
```

`lseek` solo registra de la posición actual del apuntador, no impide que ninguna operación de entrada/salida se realice.

El desplazamiento del archivo puede ser mayor al tamaño real del archivo, en cuyo caso el siguiente `write` sobre el archivo hará que se extienda. Lo anterior hace que exista un “agujero” en el archivo, lo cual es permitido. Cualquier byte que no haya sido escrito en un archivo es leído como 0.

Un “agujero” es un archivo que no requiere espacio para ser almacenado. Dependiendo del sistema operativo, cando se escribe más allá del final del archivo, nuevos bloques deben ser

agregados para almacenar los datos, pero no hay necesidad de agregar bloques para los datos que, se supone, están entre el viejo fin del archivo y la posición donde se hizo la escritura.

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

char buff1[] = "abcdefghij";
char buff2[] = "ABCDEFGHIJ";

int main() {
	int fd;

	if ((fd = open("file.nohole", O_WRONLY | O_CREAT | O_TRUNC)) < 0) {
		printf("creat error\n");
	}
	if (write(fd, buff1, 10) != 10) {
		printf("buff1 write error\n");
	}

	if (lseek(fd, 16384, SEEK_SET) == -1) {
		printf("lseek error\n");
	}

	if (write(fd, buff2, 10) != 10) {
		printf("buff2 write error\n");
	}
	return EXIT_SUCCESS;
}

$ ls -la file.hole
-rw-rw-rw- 1 manchas manchas 16394 2009-01-27 12:23 file.hole
$ od -c file.hole
0000000   a   b   c   d   e   f   g   h   i   j  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0040000   A   B   C   D   E   F   G   H   I   J
0040012
```

Con el comando `od` podemos ver el contenido del archivo. La bandera de `-c` nos permite imprimir el contenido como caracteres. Aquí podemos ver que los bytes que no han sido escritos están en la mitad del archivo y son leídos como 0. El número de 7 dígitos que aparece el inicio de cada línea es desplazamiento (bytes) en octal.

3.2.4 Función read

`read` es la función que vamos a emplear para leer datos de un archivo. Su declaración es la siguiente:

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

`read` lee `nbyte` bytes del archivo asociado del descriptor `filedes` y los coloca en la memoria intermedia referenciada por `buf`. Si la lectura se lleva a cabo correctamente, `read` devuelve el

número de bytes leídos y copiados en la memoria intermedia. Este número puede ser menor que `nbyte` en el caso de que el archivo esté asociado a un socket, o de que quedasen menos de `nbyte` bytes por leer.

Cuando se intenta leer más allá del final de archivo, `read` devuelve el valor 0. Sólo en el caso de que `read` falle, devuelve el valor de `−1` y `errno` contendrá el tipo de error que se ha producido.

En los archivos con capacidad de acceso aleatorio, la lectura empieza en la posición indicado por el apuntador de lectura/escritura. Este apuntador queda actualizado después de efectuar la lectura. En los archivos asociados a dispositivos sin capacidad de acceso aleatorio (por ejemplo, líneas serie), `read` siempre lee de la misma posición y el valor del apuntador no tiene significado.

La lectura no tenemos por qué hacerla siempre sobre un arreglo de caracteres, también se puede hacer sobre una estructura. Supongamos que queremos leer 40 registros con un formato concreto de un archivo de datos. Si la composición de cada registro la tenemos definida en una estructura de nombre RECORD, una secuencia de código para efectuar esta lectura puede ser:

```
struct RECORD buffer[40];
int nbytes, fd;
.
.
nbytes = read (fd, buffer, 40 * sizeof(RECORD));
```

3.2.5 Función write

Utilizaremos la función `write` para escribir datos en un archivo. Su declaración es muy parecida a la de `read`:

```
#include <unistd.h>
ssize_t write (int filedes, void *buf, size_t nbytes);
```

`write` escribe `nbyte` bytes de la memoria referencia por buf en el archivo asociado al descriptor `filedes`. Si la escritura se lleva a cabo correctamente, `write` devuelve el número de bytes realmente escritos; en caso contrario, devuelve `−1` y `errno` contendrá el tipo del error producido.

En los archivos con capacidad de acceso aleatorio, la escritura se realiza en la posición indicada por el apuntador de lectura/escritura del archivo. Después de la escritura, el apuntador queda actualizado. En los archivos sin capacidad de acceso aleatorio, la escritura siempre tiene efecto sobre la misma posición.

Si el indicador `O_APPEND` estaba presente al abrir el archivo, el apuntador se situará al final del mismo para que las llamadas de escritura añadan información al archivo.

En los archivos ordinarios, la escritura se realiza a través del buffer caché, por lo que una llamada a `write` no implica una actualización inmediata del disco. Este mecanismo acelera la gestión del disco, pero presenta problemas de cara a la consistencia de datos. Si no ocurre algo imprevisto, no hay nada que temer, pero en el caso de fallo no previsto (un corte de la alimentación del equipo, por ejemplo) es posible que se pierdan los datos del buffer caché que no habían sido actualizados. Si al abrir el archivo estaba presente el indicador `O_SYNC`, forzamos que las llamadas a `write` no

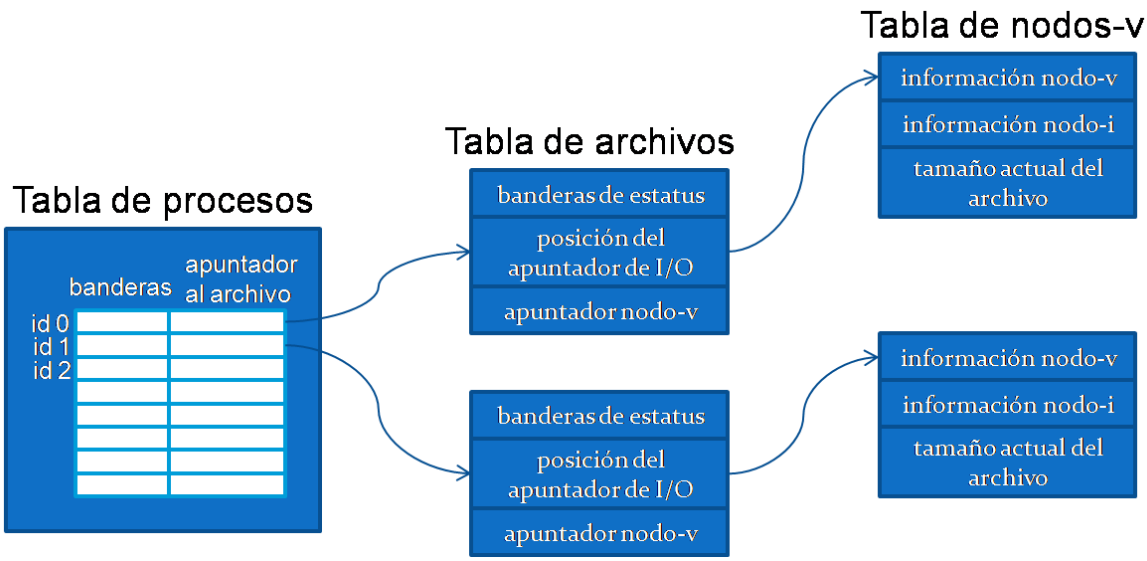
devuelvan el control hasta que se escriban los datos en el disco, asegurando así la consistencia. Naturalmente, este modo de trabajo está penalizado con un mayor tiempo de ejecución de nuestro proceso.

3.2.6 Compartición de Archivos

Unix/Linux soporta la compartición de archivos abiertos entre diferentes procesos. Antes de describir la función `dup`, es necesario explicar el mecanismo para lograr lo anterior. Para hacer esto, examinaremos las estructuras de datos que usa el kernel para todas las operaciones entrada/salida.

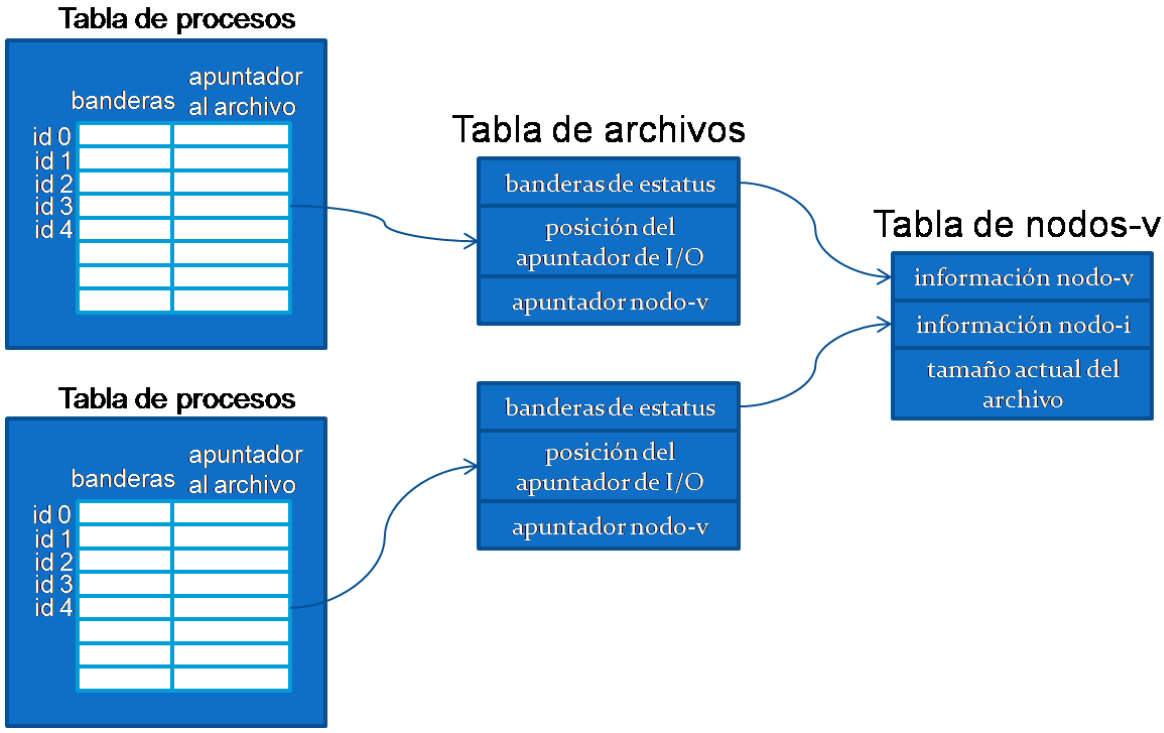
El kernel usa tres estructuras de datos para representar un archivo abierto, y las relaciones entre ellas determinan el efecto que un proceso tiene sobre otro que intenta compartir el archivo.

1. Cada proceso tiene una entrada en la tabla de procesos. En cada una de ella existe una tabla de descriptores de archivos abiertos, podemos verlos como un vector, con una localidad para cada descriptor. Asociado a cada descriptor encontramos:
    a. Las banderas del descriptor.
    b. Una referencia a una posición en la tabla de archivos.
2. El kernel mantiene una tabla de archivos con todos los archivos abiertos. Cada localidad contiene:
    a. Las banderas de estatus para cada archivo, tales como `read`, `write`, `append`, `sync` y `nonblocking`.
    b. La posición actual del apuntador de lectura/escritura.
    c. Un apuntador a una posición de la tabla de nodos-v para ese archivo.
3. Cada archivo abierto (o dispositivo) tiene una estructura llamada nodo-v que contiene información acerca del tipo de archivo y apuntadores a las funciones que operan sobre ese archivo. Para la mayoría de los archivos, el nodo-v también contiene el nodo-i del archivo. Esta información es leída desde el disco cuando el archivo es abierto, de tal forma que toda la información necesaria del archivo esté rápidamente disponible. En el caso particular de Linux, éste no maneja nodo-v. En su lugar manejar una estructura nodo-i genérico. Aunque la implementación difiere, el nodo-v es conceptualmente lo mismo que un nodo-i genérico. Ambos hacen referencia hacia una estructura nodo-i específica del sistema de archivos.



La figura superior muestra el arreglo de estas tablas para un solo proceso que tiene dos diferentes archivos abiertos: un archivo está abierto sobre la entrada estándar (descriptor 0), y el otro está abierto sobre la salida 0 (descriptor 1). Esta configuración de tablas ha existido desde las primeras versiones de Unix, y es crítico para la forma en que se comparten archivos entre los procesos.

Si dos procesos independientes tienen el mismo archivo abierto, el arreglo quedaría como lo muestra la siguiente figura.



Podemos ver que el primer proceso abierto el archivo a través del descriptor 3 y que el segundo proceso tiene el mismo archivo abierto usando el descriptor 4. Cada proceso que abrió el archivo

tiene su propia tabla de archivos, pero maneja una sola tabla de nodos-v para ese archivo. La razón por la que cada proceso tiene su propia tabla de archivos es que cada proceso tiene su propio apuntador de lectura/escritura para ese archivo.

Dada estas estructuras, ahora necesitamos ser un poco más específicos acerca de lo que sucede cuando ciertas operaciones son realizadas:

- Después de que un `write` es completado, el apuntador de lectura/escritura en el archivo es incrementado en el número de bytes que se escribieron. Si esto causa que la posición actual de apuntador exceda el tamaño actual del archivo, entonces la información sobre el tamaño es modificada en el nodo-v (para este ejemplo, el archivo es extendido).
- Si el archivo es abierto con la bandera de `O_APPEND`, la correspondiente bandera es actualizada en las banderas de estatus de la tabla de archivos. Cada que un `write` es realizado en un archivo que tiene activada esta bandera, primero se mueve el apuntador de lectura/escritura al final del archivo, en la posición indicada por el nodo-i. Esto permite que cada operación `write` siempre se haga al final del archivo.
- Si un archivo es posicionado en su posición final usando `lseek`, sucede lo siguiente: la posición del apuntador es actualizado en la tabla de archivos al tamaño del archivo que está indicado en el nodo-i.
- La función `lseek` sólo modifica la posición actual del apuntador en la tabla de archivos. No se realiza ninguna operación de entrada/salida.

Es posible que más de un descriptor se encuentre apuntando a la misma tabla de archivos, como veremos que sucede después de ejecutar la función `dup`. Esto también sucede después de invocar a la función `fork` cuando el proceso padre e hijo apuntan a una determinada tabla de archivos. Cuando hablemos de la función `fcntl` veremos cómo obtener y modificar las banderas y el estatus del descriptor de archivos.

Todo lo que hemos mencionado funciona para cualquier cantidad de archivos que estén leyendo el mismo archivo. Cada proceso tiene su propia tabla de archivos con su propio apuntador de lectura/escritura. Sin embargo, resultados inesperados pueden ocurrir cuando múltiples procesos escriben sobre un mismo archivo. Para evitar estas sorpresas veremos el concepto de operaciones atómicas.

3.2.7 Operaciones Atómicas

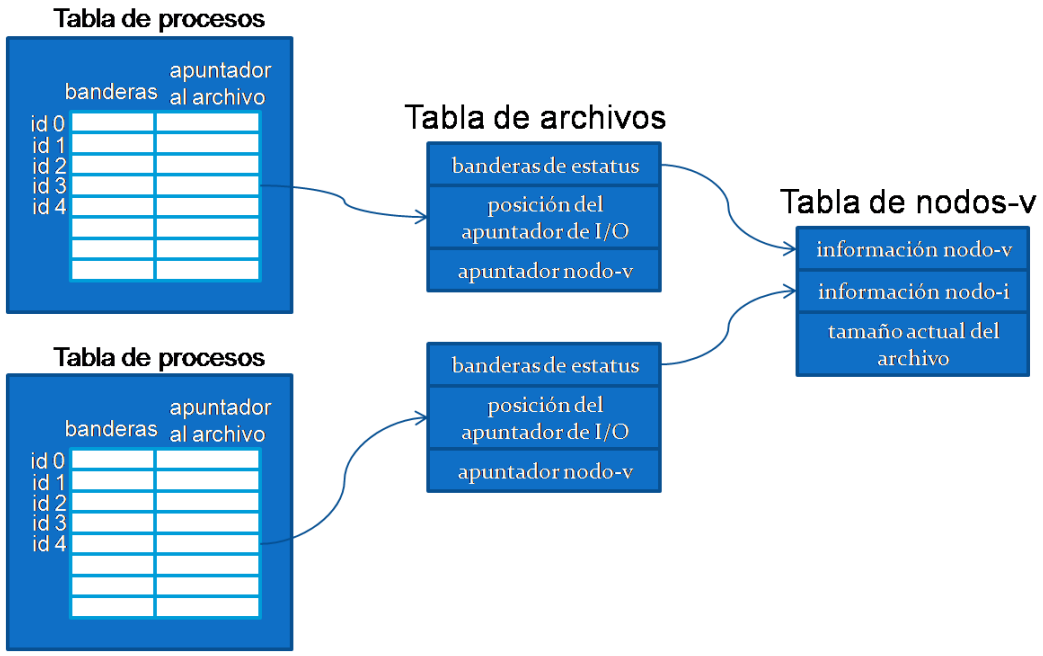
3.2.7.1 Agregando a un archivo

Considera un solo proceso que quiere agregar datos al final de un archivo. Las versiones anteriores de Unix no soportaban la opción de `O_APPEND` en `open`, de tal forma que el código del programa sería el siguiente:

```
if (lseek(fd, OL, 2) < 0)
       printf("lseek error");
if (write(fd, buf, 100) != 100)
       printf("write error");
```

Esto funciona muy bien para un solo proceso, pero surgen problemas si múltiples procesos usaran esta técnica para agregar al mismo archivo. Un posible escenario podría ser: múltiples instancias de un mismo programa están agregando mensajes a un archivo log.

Vamos a asumir que dos procesos independientes, A y B, están agregando al mismo archivo. Cada uno lo ha abierto, pero sin la bandera de `O_APPEND`. Esto nos daría el siguiente escenario:



Cada proceso tiene su propia tabla de archivos, pero comparten el mismo nodo-v. Imaginémonos que el proceso A hacer un `lseek` y coloca su apuntador de lectura/escritura en la posición 1,500 (que resulta ser el fin del archivo). Entonces el kernel cambia de procesos, y B continúa corriendo. El proceso B realiza un `lseek` y coloca su apuntador en la misma posición de 1,500 (fin de archivo). Entonces B ejecuta la instrucción `write`, lo que hace que su apuntador de lectura/escritura se incremente en 100 (1,600). Como el archivo se ha extendido, el kernel actualiza el tamaño actual del archivo en el nodo-v a 1,600. Entonces el kernel cambia de procesos y A se reanuda. Cuando A ejecuta la función `write`, los datos son escritos en la posición del apuntador que tiene señalada A, que es el byte 1,500. Esto hace que se sobrescriba la información escrita por B.

El problema es que nuestra operación lógica de “posiciónate al final del archivo y escribe” requiere de dos funciones separadas. La solución está en que posicionar y escribir se haga como una operación atómica. Cualquier operación que requiere de más una función no puede ser considerada atómica y siempre existe la posibilidad de que el kernel, temporalmente, suspenda el proceso entre alguna de las llamadas.

Unix provee una manera atómica de hacer esta operación cuando nosotros utilizamos la bandera `O_APPEND` al momento de abrir un archivo.

3.2.7.2 Funciones pread y pwrite

Unix incluye algunas extensiones que permiten a las aplicaciones posicionarse y realizar operaciones de entrada/salida de manera atómica. Estas extensiones son: `pread` y `pwrite`.

```
#include <unistd.h>
ssize_t pread (int filedes, void *buf, size_t nbytes, off_t offset);
ssize_t pwrite (int filedes, void *buf, size_t nbytes, off_t offset);
```

Ambas funciones regresan -1 en caso de que exista algún error.

Invocar `pread` es equivalente a llamar `lseek` seguido por una llamada a `read`, con las siguientes excepciones:

- No hay manera de interrumpir las dos operaciones usando `pread`.
- El apuntador del archivo no es actualizado.

Ejecutar la función `pwrite` es equivalente a realizar un `lseek` seguido de un `write`, con excepciones similares a `pread`.

3.2.7.3 Creando un Archivo

Otro ejemplo de una operación atómica se da cuando utilizamos las banderas de `O_CREAT` y `O_EXCL` en la función `open`. Cuando ambas opciones son especificadas, `open` fallará si es que el archivo existe. Al mismo tiempo de verificar la existencia también nos permite crear el archivo, todo en una operación atómica. Si no tuviéramos esta opción, deberíamos de intentar lo siguiente:

```
if ((fd = open(pathname, O_WRONLY)) < 0) {
       if (errno == ENOENT) {
              if ((fd = creat(pathname, made)) < 0) {
                     printf("creat error");
              }
       } else {
              printf("write error");
       }
}
```

El problema ocurre si el archivo ya ha sido creado por otro proceso entre el `open` y el `creat`. Si el archivo es creado por otro proceso entre estas dos llamadas, y si el otro proceso escribe algo en el archivo, los datos serán borrados cuando `creat` se ejecute. Combinando la verificación de existencia y el proceso de creación en una solo operación atómica se evita este problema.

3.2.8 Funciones dup y dup2

La llamada `dup` duplica el descriptor de archivo que ya ha sido asignado y que está ocupando una entrada en la tabla de descriptores de archivo. Su declaración es:

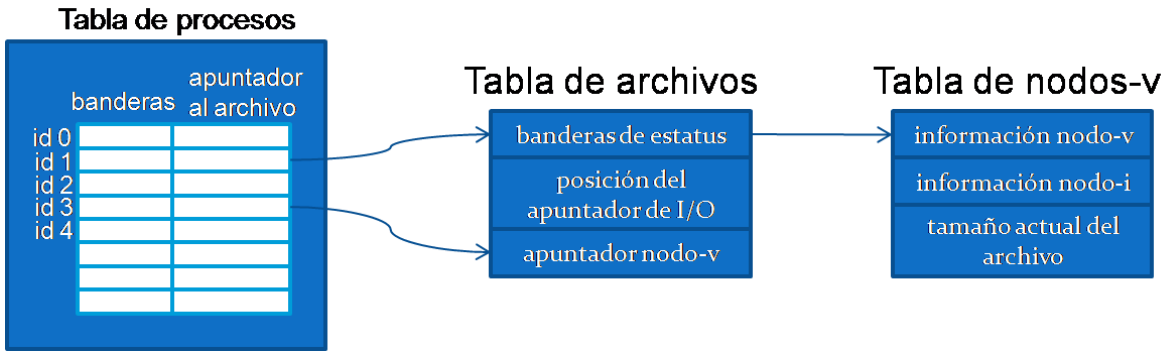
```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

`filedes` es un descriptor obtenido a través de una llamada previa a `creat`, `open`, `dup`, `fcntl` o `pipe`. El nuevo descriptor regresado por `dup` es el menor número entero posible que se pueda asignar. Con `dup2`, es posible especificar el valor del nuevo descriptor con el argumente

`filedes2`. Si `filedes2` ya se encuentra abierto, primero es cerrado. Si `filedes` es igual a `filedes2`, entonces `dup2` regresa `filedes2` sin que haya sido cerrado.

La llamada a `dup` va a recorrer la tabla de descriptores y va a marcar como ocupado la primera entrada que encuentre libre, devolviéndonos el descriptor asociado a esa entrada. Si falla en su ejecución, devolverá el valor `-1`, indicando a través de `errno` el error producido.

Los dos descriptores (original y duplicado) tienen en común que comparten el mismo archivo, por lo que a la hora de leer o escribir podemos usarlos indistintamente.



Otra forma de duplicar un descriptor es usar la función `fcntl`, la cual describiremos más adelante. De hecho, la llamada:

```
dup(filedes);
```

es equivalente a:

```
fcntl(filedes, F_DUPFD, 0);
```

Similarmente, la llamada:

```
dup2(filedes, filedes2);
```

es equivalente a:

```
close(filedes2);
fcntl(filedes, F_DUPFD, filedes2);
```

En este último caso, `dup2` no es exactamente lo mismo que un `close` seguido de un `fcntl`. Ya que `dup2` es una función atómica, mientras que la forma alterna involucra dos funciones, lo que puede hacer que surjan problemas cuando hablamos de procesos concurrentes.

3.2.9 Funciones sync, fsync y fdatasync

Las tradicionales implementaciones de Unix tienen un buffer caché a través del cual pasan todas las operaciones de entrada/salida. Cuando se escriben en un archivo, el kernel coloca los datos en uno de estos buffers y agrega la instrucción de escritura en una fila con el fin de realizar las operaciones más tarde. Esto es llamado escritura retrasada.

El kernel, eventualmente, escribe los bloques a discos cuando se necesita reusar el buffer para almacenar nuevos bloques de datos. Para asegurar la consistencia del sistema de archivos del disco con el contenido del buffer caché, tenemos las funciones de `sync`, `fsync` y `fdatasync`.

```
#include <unistd.h>
void sync(void);
int fsync(int filedes);
int fdatasync(int filesdes);
```

La función `sync` simplemente enfila todos los buffers de bloques modificados para escritura y termina; es decir, no espera que las operaciones de escritura se realicen.

La función `sync` es normalmente llamada de forma periódica (usualmente cada 30 segundos) por un proceso del sistema llamada `update`. Esto garantiza un vaciado regular de los buffers de bloque. El comando `sync(1)` también llamada a la función `sync`.

La función `fsync` se refiere al archivo especificado por `filedes`; y espera a que la escritura se haya realizado, antes de terminar. Un uso adecuado de `fsync` sería en una aplicación de base de datos que necesita asegurarse de que los bloques modificados sean escritos a disco.

La función `fdatasyc` es muy similar a `fsync`, pero solo afecta a la parte de información de un archivo.

3.2.10 Función fcntl

Con la llamada `fcntl` vamos a tener control sobre un archivo abierto mediante una llamada previa a `open`, `creat`, `dup`, `fcntl` o `pipe`. Este control va a consistir en las posibilidades de cambiar los modos permitidos de acceso al archivo, y de bloquear el acceso a una parte del mismo o su totalidad. El bloqueo tiene especial importancia cuando varios procesos trabajan simultáneamente con un archivo, y es imprescindible que los accesos a determinados registros de este sean atómicos. Imaginemos el caso de dos procesos que acceden a una base de datos común, uno para actualizar los registros y otro para leer esos mismos registros. Si no implementamos ningún mecanismo de sincronización, puede darse el caso de que el proceso lector lea una información parcialmente actualizada. Esto ocurrirá cuando el proceso que actualiza interrumpa al proceso lector en mitad de una operación de consulta de la base de datos.

La declaración de `fcntl` es la siguiente:

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int filedes, int cmd, … /* int arg */);
```

`filedes` es el descriptor de un archivo previamente abierto. `arg` es un entero o un apuntador, dependiendo del valor que tome `cmd`. Los siguientes son valores permitidos de `cmd`:

F_DUPFD	La llamada devuelve un descriptor de un archivo que se encuentra libre en este instante y que reúne las siguientes características: • Es el menor descriptor de valor mayor o igual a `arg`.

	• Tiene asociado el mismo archivo que el descriptor `filedes`. • Tiene asociado el mismo apuntador al archivo que `filedes`. • El modo del archivo referenciado por el nuevo descriptor es el mismo que el de `filedes`. • Los indicadores de estado del archivo de ambos descriptores serán los mismos. • El descriptor se heredará de padre a hijos en las llamadas a `exec`.
F_GETFD	La función devuelve el valor del indicador `close-on-exec` asociado al descriptor `filedes`. Si este indicador está activado, el archivo no se cerrará después de ejecutar una llamada a `exec`. Del valor devuelto por `fcntl`, sólo tendrá validez el bit menos significativo, que tendrá 0 si el indicador no está activo y 1 en caso contrario.
F_SETFD	Fija el indicador `close-on-exec` asociado a `filedes` de acuerdo con el bit menos significativo de `arg`. Si el bit está a 1, el indicador está activo.
F_GETFL	Devuelve los indicadores de estado y modo de acceso del archivo referenciado por `filedes`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_NDELAY`, `O_APPEND`, etc.
F_SETFL	Fija los indicadores de estado de `filedes` de acuerdo con el valor de `arg`.
F_GETOWN	Devuelve el ID del proceso o el ID del grupo que actualmente está recibiendo las señales de `SIGIO` y `SIGURG`. Estas señales se verán en temas posteriores.
F_SETOWN	Establece el ID del proceso o el ID del grupo que recibirán las señales de `SIGIO` y `SIGURG`. Un `arg` positivo especifica el ID del proceso. Un `arg` negativo implicar el ID de un grupo igual al valor absoluto de `arg`.
F_GETLK	Devuelve el primer candado que se encuentra bloqueando la región del archivo referenciado por `filedes` y descrito en la estructura de tipo `struct flock` recibida como `arg`. La información devuelta sobreescribe la información pasada a `fcntl` en `arg`. Si no se encuentra ningún candado sobre esa región, la estructura es devuelta sin cambios, excepto en el campo `l_type`, donde se activa el bit `F_UNLCK`.
F_SETLK	Activa o desactiva un candado sobre la región del archivo referenciado por `filedes` y descrita por la estructura de tipo `struct flock` recibida como `arg`. La orden `F_SETLK` se utiliza para establecer un candado de lectura (`F_RDLCK`), de escritura (`F_WRLCK`) o para eliminar uno existente (`F_UNLCK`). Si no se puede establecer alguno de estos candados, la función termina inmediatamente y regresa el valor de -1.
F_SETLKW	Esta orden es la misma que `F_SETLK`, con la diferencia de que si no se puede establecer algún candado, porque lo impiden otros ya establecidos, el proceso se pondrá a dormir hasta que se den las condiciones que lo permitan.

Un candado de lectura indica que el proceso actual está leyendo el archivo, por lo que ningún otro proceso debe escribir en el área bloqueada. Puede haber varios candados de lectura simultáneos sobre una misma región de un archivo.

Un candado de escritura indica que el proceso actual está escribiendo en el archivo, por lo que ningún proceso debe leer o escribir del área bloqueada. Sólo puede haber un candado de escritura sobre una misma área del archivo.

La estructura `struct flock` se define como sigue:

```
struct flock {
       short l_type; /* Tipo de candado:
                        F_RDLCK – lectura,
                        F_WRLCK – escritura,
                        F_UNLCK – eliminar candado */
       int l_whence; /* Punto al que se refiere la posición de la región
                        a bloquear:
                        SEEK_SET – origen del archivo
                        SEEK_CUR – posición actual
                        SEEK_END – final del archivo*/
       off_t l_start; /* Posición relativa de inicio a punto indicado por
                        l_whence */
       off_t l_len; /* Longitud de la región a bloquear. Si vale 0, se
                        bloquea desde el punto indicado en l_start hasta
                        el final del archivo*/
       pid_t l_pid; /* Identificador del proceso (PID) que tiene
                        establecido el candado. Devuelto con la orden
                        F_GETLK*/
       long l_sysid; /* Identificador del sistema que tiene establecido el
                        candado. Devuelto con la orden F_GETLK*/
};
```

Los candados fijados por un proceso sobre un archivo se borran cuando el proceso termina. Además, los candados no son heredados por los procesos hijos tras la llamada `fork`.

Si `fcntl` no se ejecuta satisfactoriamente, regresa el valor de -1 y en `errno` estará codificado el tipo de error producido.

3.2.10.1 Ejemplo 1 – Banderas de estatus

El siguiente programa toma una línea de comando que especifica un descriptor de archivo e imprime una descripción de las banderas de estatus para ese descriptor.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

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

       if (argc != 2) {
              fprintf(stderr, "uso: %s <#descriptor>\n", argv[0]);
              return -1;
       }

       if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0) {
              fprintf(stderr, "error en fcntl  con el fd %d\n", atoi(argv[1]));
       }

       switch (val & O_ACCMODE) {
              case O_RDONLY:
                     fprintf(stdout, "read only");
                     break;
              case O_WRONLY:
```

```
                fprintf(stdout, "write only");
                break;
        case O_RDWR:
                fprintf(stdout, "read & write");
                break;
        default:
                fprintf(stderr, "acceso desconocido\n");
                return -1;
    }

    if (val & O_APPEND) {
        fprintf(stdout, ", append");
    }
    if (val & O_NONBLOCK) {
        fprintf(stdout, ", nonblocking");
    }

    #if defined(O_SYNC)
        if (val & O_SYNC) {
            fprintf(stdout, ", sincrono");
        }
    #endif

    #if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC)
        if (val & O_FSYNC) {
            fprintf(stdout, ", sincrono");
        }
    #endif

    fprintf(stdout, "\n");
    return 0;
}
```

Notemos el uso de la macro `_POSIX_C_SOURCE` que nos permite compilar, condicionalmente, las banderas de acceso que no son parte de POSIX.1. Las siguientes líneas muestran la operación del programa cuando es ejecutado desde la línea de comandos:

```
$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > temp.foo
$ cat temp.foo
write only
$ ./a.out 2 2>>temp.foo
write only, append
$ ./a.out 5 5<>temp.foo
read & write
$
```

La línea de comando `5 <> temp.foo` abre el archivo `temp.foo` para ser leído y escrito usando el descriptor 5.

3.2.10.2 Ejemplo 2 – Cambiando Banderas de Estatus

Cuando modificamos ya sea las banderas del descriptor de archivos o la bandera de estatus del archivo, debemos ser muy cuidadosos al obtener el valor existente de las banderas, modificarlo como lo deseamos, y establecer ese nuevo valor. No podemos simplemente hacer un `F_SETFD` o un `F_SETFL`, como si esto pudiera apagar los bits los valores previos de cualquier bit.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFSIZE 4096

void set_fl(int fd, int flags) {
	int val;
	if ((val = fcntl(fd, F_GETFL, 0)) < 0) {
		fprintf(stderr, "error fcntl F_GETFL\n");
		return;
	}

	val |= flags;

	if (fcntl(fd, F_SETFL, val) < 0) {
		fprintf(stderr, "error fcntl F_SETFL\n");
		return -1;
	}
}

int main() {
	int n;
	char buf[BUFFSIZE];

	set_fl(STDOUT_FILENO, O_SYNC);

	while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0) {
		if (write(STDOUT_FILENO, buf, n) != n) {
			fprintf(stderr, "write error\n");
			return -1;
		}
	}
	if (n < 0) {
		fprintf(stderr, "read error\n");
		return -1;
	}
	return 0;
}
```

Las siguientes líneas muestran la operación del programa cuando es ejecutado desde la línea de comandos:

```
$ ./a.out < entrada.txt > salida.txt
```

3.2.10.3 Ejemplo 3 – Estableciendo candados

Como ejemplo, vamos a ver el uso de `fcntl` para realizar bloqueos sobre un archivo y sincronizar el acceso al mismo por parte de dos procesos. El siguiente programa lee un número que se encuentra en un archivo y tras incrementarlo lo presenta en pantalla y lo vuelve a grabar en el disco. Esta operación se repite varias veces. Vamos a ver qué ocurre cuando son dos los procesos que acceden a un archivo para realizar la misma operación. Dependiendo del tipo de acceso, con bloqueo o sin él, los resultados serán distintos. El código del programa es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

#define EQ(str1, str2) (strcmp(str1, str2) == 0)

#define ARCHIVO "tmp"

void sin_bloqueo() {}

void con_bloqueo(int fd, int orden) {
	struct flock cerrojo;

	cerrojo.l_type = orden;
	cerrojo.l_whence = SEEK_SET;
	cerrojo.l_start = 0;
	cerrojo.l_len = 0;

	if (fcntl(fd, F_SETLKW, &cerrojo) == -1) {
		perror("fcntl");
		exit(-1);
	}
}

int main(int argc, char *argv[]) {
	int fd;
	int numero, i;
	void (*bloquear) ();

	if (argc == 1) {
		bloquear = sin_bloqueo;
	} else if (argc == 2 && EQ(argv[1], "-b")) {
		bloquear = con_bloqueo;
	} else {
		fprintf(stderr, "Forma de uso: %s [-b]\n", argv[0]);
		return -1;
	}

	if ((fd = open(ARCHIVO, O_RDWR | O_CREAT, 0644)) < 0) {
		perror(ARCHIVO);
		return -1;
	}

	if (read(fd, &numero, sizeof(numero)) != sizeof(numero)) {
		numero = 0;
		write(fd, &numero, sizeof(numero));
	}

	for (i = 0; i < 10; i++) {
		lseek(fd, 0L, SEEK_SET);
		(*bloquear) (fd, F_WRLCK);
		read(fd, &numero, sizeof(numero));
		numero++;
		lseek(fd, 0L, SEEK_SET);
		write(fd, &numero, sizeof(numero));
		fprintf(stdout, "PID = %d, nro = %d\n", getpid(), numero);
		(*bloquear) (fd, F_UNLCK);
```

```
            sleep(1);
        }
        close(fd);
        return 0;
}
```

Si ejecutamos las siguientes líneas de comando, veremos cómo se comportan los objetos en cada caso:

```
$ ./a.out & ./a.out &
$ ./a.out & ./a.out –b &
```

3.3 Administración de archivos

Ahora vamos a estudiar una serie de llamadas al sistema que nos van a permitir acceder y cambiar la información de tipo administrativo y estadístico de un archivo.

3.3.1 Funciones stat, Fstat y Lstat

Todas son funciones que devuelven la información que se almacena en la tabla de nodos-i sobre el estado de un archivo concreto. La declaración de estas funciones es:

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

La diferencia entre `stat` y `fstat` es que la primera recibe como primer parámetro un apuntador al nombre del archivo (`path`), mientras que la segunda trabaja con un archivo ya abierto y le debemos pasar su descriptor (`filedes`).

Ambas funciones devuelven, a través de la estructura apuntada por `buf`, la información estadística del archivo.

`lstat` trabaja de forma similar a `stat` excepto cuando el nombre del archivo corresponde a un enlace simbólico. En este caso, `lstat` devuelve la información correspondiente al archivo que sirve de enlace mientras que `stat` devuelve información correspondiente al archivo al cual apunta el enlace.

Si la llamada se ejecuta correctamente, devuelve el valor 0; en caso contrario, devuelve -1 y en `errno` podremos consultar el tipo de error producido.

La información administrativa del archivo se almacena en una estructura de tipo `struct stat`. Este tipo está definido en el archivo cabecera `<sys/stat.h>`. Según la versión de Unix/Linux con la que trabajemos, la estructura `stat` tendrá algunas campos u otros (consultar `stat(5)`). A continuación, listamos algunas de los campos estándar de esta estructura:

```
struct stat {
       mode_t st_mode; /* 16 bits que codifican el modo del archivo.*/
       inot_t st_ino; /* Número del nodo-i.*/
       dev_t st_dev; /* Número del dispositivo que contiene el nodo-i.*/
       dev_t st_rdev; /* Identificador de dispositivo. Tiene significado
```

```
                        únicamente para los archivos especiales en modo
                        carácter y en modo bloque.*/
        nlink_t st_nlink; /* Número de enlaces al archivo.*/
        uid_t st_uid; /* Identificador de usuario (UID) del propietario del
                        archivo.*/
        gid_t set_gid; /* Identificador del grupo (GID) al que pertenece el
                        propietario del archivo.*/
        off_t st_size; /* Tamaño, en bytes, del archivo.*/
        time_t st_atime; /* Fecha del último acceso al archivo (lectura).*/
        time_t st_mtime; /* Fecha de la última modificación del archivo.*/
        time_t st_ctime; /* Fecha del último cambio de la información
                        administrativa del archivo (cambio de propietario,
                        permisos, etc.). Todas las fechas se miden en
                        segundos con respecto a las 00:00:00 GMT del día 1
                        de enero de 1970.*/
        blksize_t st_blksize; /* Bloque de E/S óptimo.*/
        blkcnt_t st_blocks; /* Número de bloques de discos utilizados.*/
};
```

Los tipos `dev_t`, `ino_t`, `mode_t`, `nlink_t`, `uid_t`, `gid_t`, `blksize_t`, `blkcnt_t`, `off_t` y `time_t` están definidos en el archivo cabecera `<sys/types.h>` y suelen ser alias de los tipos de datos `short`, `int` y `long`.

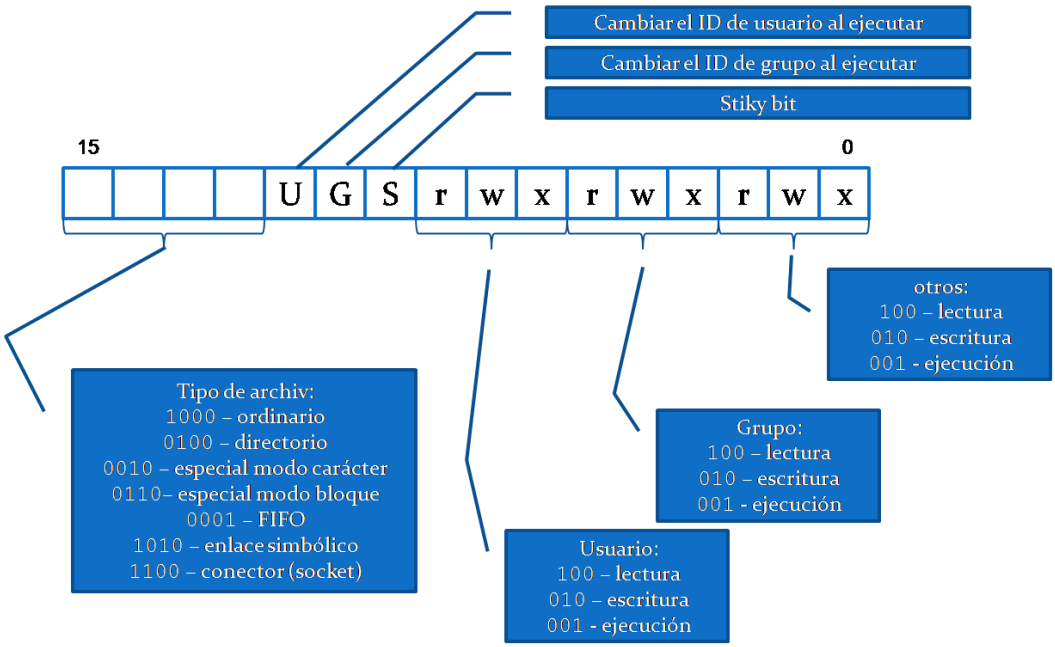
3.3.3 Tipos de Archivo

La mayoría de los archivos en Unix/Linux son archivos ordinarios (regulares) o directorios, pero existen tipos adicionales. Los tipos son:

- Archivo ordinario: El tipo más común de archivo, el cual contiene datos en alguna forma. Para Unix/Linux no existe distinción entre información guardada en formato texto o binario. Cualquier interpretación del contenido del archivo es dejada al programa que procese el archivo. Un caso especial son los archivos binarios ejecutables. Para ejecutar un programa, el kernel debe entender su formato.
- Archivo directorio. Este archivo contiene los nombres de otros archivos y apuntadores a la información de los mismos. Cualquier proceso que tenga derecho de lectura sobre un archivo directorio puede leer su contenido, pero el único que puede escribir es el kernel. Un proceso puede usar las funciones descriptas en este tema para realizar cambios en un directorio.
- Archivo especial de bloque: Un tipo de archivo que provee acceso de entrada/salida con buffer en unidades de tamaño fijo a dispositivos tales como discos duros.
- Archivo especial de carácter: Un tipo de archivo que provee acceso de entrada/salida sin buffer en unidades de longitud variable a dispositivos. Todos los dispositivos en un sistema pueden ser de bloque o carácter.
- FIFO: Un tipo de archivo usado para comunicación entre procesos. Algunas veces llamados pipes.
- Socket: Un tipo de archivo utilizado para comunicaciones de red entre procesos. Un socket también puede ser usado para comunicaciones entre procesos en un solo host.
- Enlace simbólico: Un tipo de archivo que apunta hacia otro archivo.

3.3.4 Modos de un Archivo

Al estudiar la función `open` hemos visto que cada archivo tiene asociada una máscara de 9 bits que indica los permisos de lectura, escritura y ejecución que tiene el propietario, grupo y demás usuarios sobre el archivo. En realidad, estos 9 bits forman parte de una máscara más amplia, conocida como modo del archivo. La máscara de modo se compone de 16 bits cuyo significado es el que se muestra en la siguiente figura:



En el archivo cabecera `<sys/stat.h>` hay definidas unas constantes para acceder a los bits de modo y extraer su información. Las constantes definidas son:

Bits	Constante	Valor	Significado
15-12	S_IFMT S_IFREG S_IFDIR S_IFCHR S_IFBLK S_IFIFO S_IFSOCK	0170000 0100000 040000 020000 060000 010000 0140000	Tipo de archivo: Ordinario Directorio Especial modo carácter Especial modo bloque FIFO Conector (Socket)
11	S_ISUID	04000	Activar ID del usuario al ejecutar
10	S_ISGID	02000	Activar ID del grupo al ejecutar
9	S_ISVTX	01000	Stiky Bit
8	S_IRUSR	0400	Permiso de lectura para el usuario
7	S_IWUSR	0200	Permiso de escritura para el usuario
6	S_IXUSR	0100	Permiso de ejecución

			para el usuario
5	S_IRGRP	040	Permiso de lectura para el grupo
4	S_IWGRP	020	Permiso de escritura para el grupo
3	S_IXGRP	010	Permiso de ejecución para el grupo
2	S_IROTH	04	Permiso de lectura para otros
1	S_IWOTH	02	Permiso de escritura para otros
0	S_IXOTH	01	Permiso de ejecución para otros

Podemos usar estas constantes como filtros sobre el bit que nos interese. Hay que advertir que para determinar el tipo de archivo se debe usar la máscara `S_IFMT`. Por ejemplo, si queremos saber si un archivo es un directorio o no, se deber usar una expresión cómo:

```
if ((mode & S_IFMT) == S_IFDIR)
```

Porque si utilizamos,

```
if ((mode & S_IFDIR) == S_IFDIR)
```

Nos dará también el valor lógico de verdad cuando ese archivo sea de tipo especial de bloque.

Hay tres bits cuyo significado no se ha definido de momento, son: `S_ISUID` (no. 11), `S_ISGID`(no. 10) y `S_ISVTX` (no. 9). Vamos a ver qué significan:

- `S_ISUID` (cambiar el identificador del usuario en ejecución) le indica al kernel que cuando un proceso accede a este archivo, cambie el identificador de usuario del procesos y le ponga el del archivo. Esto tiene una aplicación cuando intentamos acceder a archivos que son de otro usuario y no tenemos permiso para escribir en ellos. Como ejemplo vamos a mencionar la orden `passwd` (utilizada para cambiar la clave de acceso). Al cambiar de clave, la nueva se debe guardar en el archivo `/etc/passwd`; sin embargo, este archivo sólo puede ser modificado por un usuario con privilegios de superusuario. Ante este impedimento, a un usuario normal le resultaría imposible cambiar de clave. No obstante, la realidad es distinta; como el programa `passwd` tiene activo el bits de `S_ISUID`, al ejecutarlo nos convertimos momentáneamente en superusuarios, porque nuestro UID cambia y toma el valor del usuario `root` (propietario de `passwd`), pudiendo así escribir sobre el archivo `/etc/passwd`, cuyo propietario es también `root`.
- `S_ISGID` (cambiar el identificador del grupo en ejecución) tiene un significado parecido al de `S_ISUID`, pero referido al grupo de usuarios al que pertenece el propietario del archivo. Así, cuando ejecutamos un programa que tiene activo ese bit, nuestro GID (identificador de grupo) toma el valor del GID del propietario del programa.
- `S_ISVTX` (stiky bit) le indica al kernel que este archivo es un programa con capacidad para que varios procesos compartan su segmento de código y que este segmento se debe mantener en memoria, aun cuando alguno de los procesos que lo utiliza deje de

ejecutarse o pase al área de swap. La técnica de compartir el mismo código entre varios procesos permite gran ahorro de memoria en el caso de programa muy utilizados, como editores de texto, compiladores, etc.

Otra manera de determinar el tipo de archivo es usar las macros definidas en `<sys/stat.h>`. El argumento de cada una de estas macros es el campo `st_mode` que se obtiene de la estructura `stat`.

Macro	Tipo de archivo
S_ISREG()	Archivo ordinario
S_ISDIR()	Archivo directorio
S_ISCHR()	Archivo especial de carácter
S_ISBLK()	Archivo especial de bloque
S_ISFIFO()	FIFO o pipe
IS_ISLNK()	Enlace simbólico
S_ISSOCK()	Socket
S_TYPEISMQ()	Fila de mensajes
S_TYPEISSEM()	Semáforo
S_TYPEISSSHM()	Objeto de memoria compartida

Un ejemplo del uso de estas macros sería:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
	int i;
	struct stat buf;
	char *ptr;

	for (i = 1; i < argc; i++) {
		fprintf(stdout, "%s: ", argv[i]);
		if (lstat(argv[i], &buf) < 0) {
			fprintf(stdout, "error de lstat\n");
		} else {
			if (S_ISREG(buf.st_mode)) {
				ptr = "ordinario";
			} else if (S_ISDIR(buf.st_mode)) {
				ptr = "directorio";
			} else if (S_ISCHR(buf.st_mode)) {
				ptr = "especial caracter";
			} else if (S_ISBLK(buf.st_mode)) {
				ptr = "especial bloque";
			} else if (S_ISFIFO(buf.st_mode)) {
				ptr = "fifo";
			} else if (S_ISLNK(buf.st_mode)) {
				ptr = "enlace simbólico";
			} else if (S_ISSOCK(buf.st_mode)) {
				ptr = "socket";
			} else {
				ptr = "*** tipo desconocido ***";
			}
			fprintf(stdout, "%s\n", ptr);
```

```
            }
        }
        return 0;
}
```

El programa anterior imprime el tipo de archivo para cada argumento que recibe de la línea de comando. Este sería un ejemplo de su salida:

```
$ ./a.out /etc/passwd /etc /dev/initctl /dev/log /dev/tty /dev/sda1  /dev/cdrom
/etc/passwd: ordinario
/etc: directorio
/dev/initctl: fifo
/dev/log: socket
/dev/tty: especial caracter
/dev/sda1: especial bloque
/dev/cdrom: enlace simbólico
$
```

3.3.4.1 Permisos de Acceso a Archivo

Las tres categorías de permiso (lectura, escritura y ejecución) son usadas de varias maneras por diferentes funciones.

- La primera regla es que siempre que queramos abrir cualquier tipo de archivo por su nombre, debemos tener el permiso de ejecución en cada directorio mencionado en el nombre, incluyendo el actual, si está implicado. Esto es porque el bit de permiso de ejecución para un directorio también es llamado bit de búsqueda.

  Por ejemplo, para abrir el archivo `/usr/include/stdio.h`, necesitamos tener permiso de ejecución en el directorio `/`, permiso de ejecución `/usr`, y permiso de ejecución en el directorio `/usr/include`. Luego, necesitamos tener los permisos necesarios para el archivo, dependiendo de como lo queramos abrir: solo lectura, solo escritura o lectura-escritura.

  Si el directorio actual es `/usr/include`, entonces tenemos que tener el permiso de ejecución en el directorio actual para poder abrir el archivo `stdio.h`. Este es un ejemplo de cuando el directorio está siendo implicado, aunque no sea específicamente mencionado.

  Otro ejemplo de un archivo directorio implícitamente referenciado es cuando la variable de ambiente `PATH` especifica un directorio que no tiene el permiso de ejecución activado. En este caso, el shell nunca será capaz de encontrar archivos ejecutables en ese directorio.
- El permiso de lectura para un archivo determina si podemos abrir un archivo existente para lectura.
- El permiso de escritura para un archivo determina si podemos abrir un archivo existente para escritura.
- Debemos tener permiso de escritura sobre un archivo para poder usar la bandera `O_TRUNC` en la función de open.

- No podemos crear un nuevo archivo en un directorio, a menos que tengamos permiso de escritura y ejecución sobre ese archivo.
- Para borrar un archivo existente, necesitamos tener permiso de escritura y ejecución sobre el directorio que contiene el archivo. No necesitamos ningún otro permiso para el archivo mismo.
- El permiso de ejecución para un archivo debe estar activado si queremos ejecutar este archivo usando cualquiera de las 6 funciones `exec`.

3.3.4.2 chmod y fchmod

Las llamadas `chmod` y `fchmod` se utilizan para cambiar el modo de un archivo. Sus declaraciones son:

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

Las dos llamadas cambian el modo de un archivo haciendo que tome el valor `mode`. En `chmod` especificamos el archivo por su ruta, `path`, y con `fchmod` actuamos sobre un archivo ya abierto y que tiene asociado el descriptor `filedes`. Si se ejecutan correctamente, estas funciones devuelven 0; en caso contrario, devuelven -1 y hacen que `errno` tome el código asociado al tipo de error producido.

3.3.4.3 access

Determina la accesibilidad de un archivo por parte de un proceso. Su declaración es:

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

`path` es un apuntador a la ruta del archivo al que queremos acceder. `amode` es una máscara que codifica el tipo de acceso por el que preguntamos. En `<unistd.h>` están definidos los siguientes valores para `amode`:

`R_OK` – Permiso para leer.
`W_OK` – Permiso para escribir.
`X_OK` – Permiso para ejecutar (buscar en caso de directorios)

Si la petición de acceso se satisface, la función devuelve 0, en caso contrario, devuelve -1 y `errno` contendrá el tipo de error producido.

En la llamada a `access` se emplean siempre el UID y el GID reales y no los efectivos.

Un ejemplo del uso de esta llamada sería:

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[]) {
	if (argc != 2) {
		fprintf(stderr, "uso: %s <pathname>", argv[0]);
		return -1;
	}
	if (access(argv[1], R_OK) < 0) {
		fprintf(stderr, "access error for: %s <pathname>\n", argv[0]);
		return -1;
	} else {
		fprintf(stdout, "acceso de lectura - OK\n");
	}
	if (open(argv[1], O_RDONLY) < 0) {
		fprintf(stderr, "access error for: %s <pathname>\n", argv[0]);
		return -1;
	} else {
		fprintf(stdout, "abierto para lectura - OK\n");
	}
	return 0;
}
```

Y aquí una ejecución de este programa:

```
$ ls -la a.out
-rwxr-xr-x 1 manchas manchas 9220 2009-02-04 12:09 a.out
$ ./a.out a.out
acceso de lectura - OK
abierto para lectura - OK
$ ls -la /etc/shadow
-rw-r----- 1 root shadow 905 2009-01-23 00:02 /etc/shadow
$ ./a.out /etc/shadow
access error for: ./a.out <pathname>
$ sudo ./a.out /etc/shadow
[sudo] password for manchas:
acceso de lectura - OK
abierto para lectura - OK
$
```

3.3.4.4 umask

La usamos para definir la máscara de permisos que va a tener asociados un proceso a la hora de crear archivos. Su declaración es:

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

`cmask` es el valor que queremos que tome la máscara por defecto. `umask` devuelve el valor que tenía la máscara anterior. Hay que tener presente que los bits que estén activados en `cmask` se interpretarán como desactivados cuando creemos un archivo, y si las llamadas a `open` o `creat` intentan activar estos bits, no van a poder hacerlo. Así, por ejemplo, en la siguiente secuencia de código:

```
int filedes;
.
.
```

```
umask(0066);
filedes = creat("miarchivo", 0666);
```

creat intenta crear `miarchivo` con permisos de lectura y escritura para el propietario, grupo y otros usuarios (`rw-rw-rw-`), pero `umask` prohíbe expresamente la creación de archivos con esos permisos, por lo que `miarchivo` sea crea con permisos 0600 (`rw-------`).

El siguiente programa crea dos archivos (`foo` y `bar`). El primer archivo es creado usando la máscara 0, mientras que el segundo es creando con la máscara estándar de Unix.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define RWRWRW (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)

int main(int argc, char *argv[]) {
	umask(0);
	if (creat("foo", RWRWRW) < 0) {
		fprintf(stderr, "error de creat para foo");
		return -1;
	}
	umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
	if (creat("bar", RWRWRW) < 0) {
		fprintf(stderr, "error de creat para bar");
		return -1;
	}
	return EXIT_SUCCESS;
}
```

Veamos cómo se comporta en línea de comandos:

```
$ umask
0022
$ ./a.out
$ ls -l foo bar
-rw------- 1 manchas manchas 0 2009-02-04 12:59 bar
-rw-rw-rw- 1 manchas manchas 0 2009-02-04 12:59 foo
$ umask
0022
$
```

3.3.5 Cambio de la Información Estadística de un Archivo

Hemos visto cómo cambiar el modo asociado a un archivo a través de la llamada `chmod`. Ahora veremos otras llamadas para cambiar parámetros tales como nombre del archivo, propietario, fechas y tamaño.

3.3.5.1 Cambio del Nombre de un Archivo – rename

`rename` se declara como sigue:

```
#include <stdio.h>
int rename(const char *source, const char *target);
```

`source` indica el nombre original del archivo, y `target`, el nuevo nombre que queremos que adopte. En el caso de que `target` exista, se borra previamente.

Como siempre, si la llamada falla, devuelve -1 y en caso de funcionar correctamente devuelve 0.

3.3.5.2 Cambio del Propietario y del Grupo de un Archivo – chown y fchown

Las funciones `chown` y `fchown` sirven tanto para cambiar el identificador del propietario de un archivo como el identificador del grupo. Sus declaraciones son:

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

La diferencia entre `chown` y `fchown` es que mientras la primera trabaja con la ruta (`path`) de un archivo, la segunda lo hace con el descriptor (`filedes`) de un archivo ya abierto.

`owner` debe ser el identificador de un usuario declarado en el sistema y `group` el identificador de un grupo. Si queremos dejar alguno de estos parámetros inalterados mientras cambiamos el otro, podemos usar las constantes `UID_NO_CHANGE` y `GID_NO_CHANGE` como parámetros para `owner` y `group`, respectivamente.

Si la llamada funciona correctamente, devuelve 0; en caso contrario, devuelve -1 y en `errno` encontraremos el código del error producido.

3.3.5.3 Cambio de la Fecha de un Archivo – utime

Para cambiar las fechas de último acceso y última modificación de un archivo, debemos usar la función `utime`. Su declaración es:

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

`path` es el apuntador al nombre del archivo cuyas fechas queremos cambiar. `times` es un apuntador a una estructura cuyo tipo está definido en el archivo cabecera `<utime.h>`

Si times vale `NULL`, las fechas de acceso y modificación tomar el valor de la fecha y hora locales de la computadora (la hora actual). Si un proceso tiene el mismo identificador de usuario que el archivo, o tiene permiso de escritura sobre el archivo, podrá modificar las fechas según este procedimiento.

En el caso de que times no sea un apuntador a `NULL`, las fechas se asignan de acuerdo con los campos de la estructura `struct utimebuf`.

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

Tanto `actime` como `modtime` se expresan en segundos con respecto a las 00:00:00 GMT del día 1 de enero de 1970. Este procedimiento de cambio de fechas sólo lo puede emplear el propietario del archivo y el superusuario.

Si la llamada a `utime` se ejecuta correctamente, devuelve 0; en caso contrario devuelve -1 y `errno` tendré el código del tipo de error producido.

3.3.5.4 Cambiar la Longitud de un Archivo – truncate y ftruncate

En el Unix System V no es posible modificar el tamaño de un archivo salvo para reducirlo a 0 byte o para incrementarlo añadiéndole bytes al final. En el sistema 4.3 BSD, sí se puede truncar la longitud de un archivo para que tome cualquier valor comprendido entre la longitud nula y la longitud actual del archivo. Con las funciones `truncate` y `ftruncate` se puede realizar esta operación, sus declaraciones son:

```
int truncate(char *path, unsigned long length);
int ftruncate(int filedes, unsigned long length);
```

`length` es la nueva longitud, en bytes, que va a poseer el archivo. `truncate` trabaja con un archivo especificado por su nombre (`path`) y `ftruncate` lo hace con un archivo ya abierto en modo escritura y que tiene asociado `filedes` como descriptor.

Si las funciones se ejecutan correctamente, devuelven 0. En caso contrario devuelven -1, indicando el tipo de error a través de `errno`.

3.3.5.5 Ejemplo de utilización de la Información Estadística de un Archivo

A continuación, presentaremos un programa que se puede utilizar para mostrar en pantalla toda la información de un archivo a la que te tenemos acceso con la función `stat`. Este programa se llamará `estado` y la forma es invocarlo es:

```
$ estado archivo
```

Algunos ejemplos de utilización son:

```
$ ./estado /dev/tty0
Archivo: /dev/tty0
Se encuentra en el dispositivo: 0, 14
Numero de nodo-i: 721
Tipo: especial modo caracter
Permisos: 0660 rw-rw----
Enlaces: 1
User ID: 0; Nombre: root
Group ID: 0; Nombre: root
Numero de dispositivos: 4, 0
Tamanio: 0 bytes.
Ultimo acceso: Wed Feb  4 09:50:20 2009
Ultima modificacion: Wed Feb  4 09:50:20 2009
Ultimo cambio de estado: Wed Feb  4 15:51:03 2009

$ ./estado /bin/pwd
Archivo: /bin/pwd
Se encuentra en el dispositivo: 8, 1
Numero de nodo-i: 135549
```

```
Tipo: ordinario
Permisos: 0755 rwxr-xr-x
Enlaces: 1
User ID: 0; Nombre: root
Group ID: 0; Nombre: root
Tamanio: 30200 bytes.
Ultimo acceso: Thu Jan 22 23:47:58 2009
Ultima modificacion: Thu Jun 26 19:31:57 2008
Ultimo cambio de estado: Thu Jan 22 23:23:17 2009
```

Programa estado.c:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <pwd.h>
#include <grp.h>
#include <stdio.h>
#include <stdlib.h>

char permisos[] = {'x', 'w', 'r'};

void estado(char *archivo) {
	struct stat buf;
	struct passwd *pw;
	struct group *gr;
	int i;

	if (stat(archivo, &buf) == -1) {
		perror(archivo);
		exit(-1);
	}

	fprintf(stdout, "Archivo: %s\n", archivo);
	fprintf(stdout,	"Se encuentra en el dispositivo: %d, %d\n",
			(int) ((buf.st_dev & 0xff00) >> 8),
			(int) (buf.st_dev & 0xff));
	fprintf(stdout, "Numero de nodo-i: %d\n", (int) buf.st_ino);

	fprintf(stdout, "Tipo: ");
	switch(buf.st_mode & S_IFMT) {
		case S_IFREG:
			fprintf(stdout,"ordinario\n");
			break;
		case S_IFDIR:
			fprintf(stdout,"directorio\n");
			break;
		case S_IFCHR:
			fprintf(stdout,"especial modo caracter\n");
			break;
		case S_IFBLK:
			fprintf(stdout,"especial modo bloque\n");
			break;
		case S_IFIFO:
			fprintf(stdout,"FIFO\n");
			break;
	}
```

```
	if (buf.st_mode & S_ISUID) {
		fprintf(stdout,"Cambiar el ID del propietario en ejecucion.\n");
	}
	if (buf.st_mode & S_ISGID) {
		fprintf(stdout, "Cambiar el ID del grupo en ejecucion.\n");
	}
	if (buf.st_mode & S_ISVTX) {
		fprintf(stdout,"Stiky bit activo.\n");
	}

	fprintf(stdout,"Permisos: 0%o ", buf.st_mode & 0777);
	for (i = 0; i < 9; i++) {
		if (buf.st_mode & (0400 >> i)) {
			fprintf(stdout,"%c", permisos[(8-i) % 3]);
		} else  {
			fprintf(stdout, "-");
		}
	}
	fprintf(stdout,"\n");

	fprintf(stdout,"Enlaces: %d\n", buf.st_nlink);

	fprintf(stdout,"User ID: %d; Nombre: ", buf.st_uid);
	if ((pw = getpwuid(buf.st_uid)) == NULL) {
		fprintf(stdout,"???\n");
	} else {
		fprintf(stdout,"%s\n", pw->pw_name);
	}

	fprintf(stdout,"Group ID: %d; Nombre: ", buf.st_gid);
	if ((gr = getgrgid(buf.st_gid)) == NULL) {
		fprintf(stdout,"???\n");
	} else {
		fprintf(stdout,"%s\n", gr->gr_name);
	}

	switch (buf.st_mode & S_IFMT) {
		case S_IFCHR:
		case S_IFBLK:
			fprintf(stdout,"Numero de dispositivos: %d, %d\n",
				(int) ((buf.st_rdev & 0xff00) >> 8),
				(int) (buf.st_rdev & 0x00ff));
	}

	fprintf(stdout,"Tamanio: %d bytes.\n", (int) buf.st_size);

	fprintf(stdout,"Ultimo acceso: %s",
			(char *) asctime(localtime(&buf.st_atime)));
	fprintf(stdout,"Ultima modificacion: %s",
			(char *) asctime(localtime(&buf.st_mtime)));
	fprintf(stdout,"Ultimo cambio de estado: %s",
			(char *) asctime(localtime(&buf.st_ctime)));
}

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

	if (argc != 2) {
```

```
            fprintf(stderr, "forma de uso: %s nombre_archivo.\n", argv[0]);
            return -1;
      }
      for (i = 1; i < argc; i++) {
            estado(argv[i]);
      }
      return 0;
}
```

En este programa hemos usados dos funciones de la librería estándar de C que merecen ser comentadas con detalle. Se trata `getpwuid` y `getgrid`.

`getpwuid` se emplea para leer la información relativa al usuario propietario del archivo que se encuentra en el archivo `/etc/passwd`. La declaración es la siguiente:

```
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
```

Si el usuario cuyo identificador indica `uid` existe, la función devuelve un apuntador a una estructura de tipo `struct passwd` que contiene la información más detallada sobre el usuario. De cada al programa, interesa el campo `pw_name` de la estructura `passwd`, ya que contiene el nombre del usuario. Para más información sobre esta función, podemos consultar la entrada `getpwent(3C)` del manual de Unix.

La función `getgrgid` la empleamos para buscar en el archivo `/etc/group` más información sobre el grupo al que pertenece el propietario del archivo. La declaración de esta función es:

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
```

Si al grupo indicado en `gid` existe, la función devuelve un apuntador a una estructura de tipo `struct group` que contiene, entre otros, el campo `gr_name` que da el nombre del grupo. Para más información sobre esta función, podemos consultar la entrada `getgrent(3C)` del manual.

3.3.6 Compartición y Bloqueo de Archivos

En puntos anteriores hemos visto cómo la llamada `fcntl` puede servirnos para bloquear el acceso a la totalidad o parte de un archivo. Este bloqueo se hace especialmente útil cuando hay varios procesos que simultáneamente hacer uso de un mismo archivo.

A la hora de bloquear un recurso (ya sea este recurso un archivo o de otra naturaleza) podemos distinguir dos implementaciones o formas de trabajo distintas. Por un lado, tenemos el bloqueo consultivo, y por otro, el bloqueo obligatorio.

Mediante el bloqueo consultivo, el sistema operativo conoce en cada momento qué recursos se encuentran bloqueados y por qué procesos, pero no prohíbe a ningún otro proceso que haga uso de esos recursos. Las medidas que se deben tomar son consultar el estado del recurso antes de usarlo, y si se encuentra libre, trabajar con él. En caso contrario, habrá que esperar a que quede libre. La decisión de si se usa o no, depende del usuario, ya que el sistema operativo se limita informar del estado del recurso. Este tipo de bloque es adecuado para procesos cooperativos.

Estos son procesos diseñados para consultar el estado del recurso que comparten y esperar a que quede libre. La inclusión de un proceso que no respete estas reglas puede tener resultados desagradables para todos los procesos que trabajan con un recurso compartido.

Con el bloqueo obligatorio, el sistema operativo comprueba cada uno de los accesos al recurso compartido con objeto de negarle el acceso a aquellos procesos no autorizados en ese instante. Con este tipo de bloqueo no es necesario consultar el estado del recurso, ya que aunque intentemos acceder indebidamente, el sistema lo va a impedir.

La llamada `lockf` está diseñada para bloquear la totalidad o parte de un archivo, tanto en bloqueo consultivo como obligatorio. La declaración de esta llamada es:

```
#include <unistd.h>
int lockf(int filedes, int function, long size);
```

`lockf` bloquea la región deseada del archivo cuyo descriptor es `filedes`, impidiendo que otros procesos pueden acceder a esa región. Puede bloquearse más de una región de un archivo. Cuando un proceso terminar su ejecución o cierra alguno de los archivos que tiene bloqueados, se borran todos los candados que tenía definidos sobre ese archivo.

Para poder definir candados sobre un archivo mediante `lockf`, es necesario tenerlo abierto en modo sólo escritura o lectura/escritura.

El parámetro `function` especifica el tipo de acción que se va a realizar para definir el candado. Sus posibles valores son:

Valores	Significado
F_ULOCK	Desbloquear una región previamente bloqueada.
F_LOCK	Bloquear una región para uso exclusivo del proceso que invoca `lockf`. Si la región no está disponible, porque ha sido bloqueada por otro proceso, el proceso actual se pondrá a dormir hasta que la región esté disponible.
F_TLOCK	Comprobar si la región a bloquear está disponible, si lo está, bloquear la región para uso exclusivo del proceso que llama a `lockf`. Si la región no está disponible, `lockf` devuelve -1 y en `errno` estará el código del error producido.
F_TEST	Comprobar si la región especificada está bloqueada por otro proceso o no. Si la región está accesible, `lockf` devolverá el valor 0; en caso contrario, devolverá -1 y en `errno` el código `EAGAIN`.

`size` es el total de bytes contiguos que se van a bloquear o desbloquear. El segmento por bloquear empieza con la posición actual del apuntador de lectura/escritura y ocupa tantos bytes como indique `size`. `size` puede ser de valor negativo, con lo que el segmento a bloquear es el situado antes del apuntador de lectura/escritura, sin que éste quede incluido. Si `size` vale 0, el segmento bloqueado se extiende hasta el final del archivo, incluso aunque el archivo crezca de tamaño. Esto quiere decir que los bytes que añadan en el futuro también van a estar bloqueados.

Conviene destacar que la diferencia entre los modos de bloqueo `F_LOCK` y `F_TLOCK` es la actuación que se sigue en el caso de que el recurso a bloquear ya esté ocupado. Con `F_LOCK` el proceso dormirá hasta que el recurso quede libre, con `F_TLOCK` la llamada a `lockf` falla y devuelve -1. El modo `F_TLOCK` es equivalente en su funcionamiento al de las instrucciones conocidas como test-and-set. Lo importante es que las operaciones de comprobar y bloquear se realizan de forma atómica. Imaginemos que no disponemos de esta opción; la forma de comprobar, y bloquear en caso de que el archivo esté libre, sería como muestra la siguiente secuencia de código:

```
if ((lockf(fd, F_TEST, size) == 0) {
      lockf(fd, F_LOCK, size);
}
```

Pero la operación anterior no es indivisible, porque después de la comprobación, el planificador puede cambiar de contexto y pasarle el control a otro proceso que bloquee el archivo, ocasionando que falle la segunda llamada a `flock`. Este tipo de errores se soluciona empleado el modo `F_TLOCK`. A los candados fijados con `F_TLOCK` se les conoce como candados no bloqueantes, puesto que, si el proceso que lo utiliza no puede seguir adelante con el bloqueo, no se queda durmiendo en espera de poder establecer el candado. Ejemplo de programas que hacen uso de este tipo de candados son aquellos que impiden que haya más de una copia de sí mismo en memoria. Estos programas declaran un candado no bloqueante sobre un archivo y si la declaración falla es porque ya hay una copia cargar en memoria; en caso contrario, se carga la primera copia y el archivo queda bloqueado impidiendo la ejecución de futuras copias. Los demonios que dan servicios de red, impresora, sincronismos con el disco, etc., son programas de este tipo.

Existe peligro de que el uso de bloqueos nos lleve a la situación indeseada de un deadlock. Pensemos por un momento que dos procesos que tiene bloqueados sendos archivos, y que cada uno de ellos está esperando a que el otro desbloquee su archivo para continuar. En el escenario descrito, ambos procesos dormirán eternamente. Para prevenir esta situación, el sistema revisa las llamadas a `fcntl` y `lockf`, y en caso de que se dé un deadlock, las llamadas fallarán, devolviendo el valor -1 y haciendo que `errno` valga `EDEADLK`.

En el siguiente ejemplo se muestran los dos modos de bloqueo que permite `lockf`. El programa está implementado para bloquear la totalidad de un archivo durante 30 segundos. La primera vez que se ejecuta, el bloque se efectúa sin problemas. Si mientras el archivo está bloqueado intentamos bloquearlo con el mismo programa, veremos que el bloqueo no se puede efectuar, y dependiendo del tipo de bloqueo que utilicemos, sin comprobación previa o con ella, el programa se pondrá a dormir o nos devolverá el control. En el caso de que duerma, el programa despertará cuando el archivo quede desbloqueado y pasará inmediatamente a bloquearlo.

```
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define EQ(str1, str2) (strcmp(str1, str2) == 0)
```

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

	if (argc != 3) {
		fprintf(stderr, "forma de uso: %s modo archivo\n", argv[0]);
		fprintf(stderr, "     modo: -a [consultivo]\n");
		fprintf(stderr, "           -m [obligatorio]\n");
		return -1;
	}

	if ((fd = open(argv[2], O_RDWR | O_CREAT, 0744)) < 0) {
		perror(argv[2]);
		return -1;
	}

	if (EQ(argv[1], "-a")) {
		if (lockf(fd, F_TLOCK, 0L) < 0) {
			fprintf(stderr, "el archivo [%s] está bloqueado por otro
proceso.\n", argv[2]);
			return -1;
		}
	} else if (EQ(argv[1], "-m")) {
		if (lockf(fd, F_LOCK, 0L) < 0) {
			perror(argv[2]);
			return -1;
		}
	}

	fprintf(stdout, "PID %d: el archivo [%s] estará bloqueado durante 30
segundos.\n", getpid(), argv[2]);
	sleep(30);

	if (lockf(fd, F_ULOCK, 0L) < 0) {
		perror("lockf");
		return -1;
	} else {
		fprintf(stdout, "\007PID %d: el archivo [%s] ha sido
desbloqueado.\n", getpid(), argv[2]);
	}

	close(fd);
	return 0;
}
```

Para ver qué ocurre cuando ejecutamos el programa en sus diferentes modos, tenemos que ejecutar en segundo plano varias copias de este. Por ejemplo:

```
$ ./a.out -a tmp &
PID 5160: el archivo [tmp] estará bloqueado durante 30 segundos.
[1] 5160
$ ./a.out -a tmp
el archivo [tmp] está bloqueado por otro proceso.
$ ./a.out -m tmp
PID 5162: el archivo [tmp] estará bloqueado durante 30 segundos.
PID 5160: el archivo [tmp] ha sido desbloqueado.
PID 5162: el archivo [tmp] ha sido desbloqueado.
[1]+  Done                    ./a.out -a tmp
```

```
$
```

La segunda vez que ejecutamos el programa, éste devuelve el control, ya que se ha empleado el modo `F_TLOCK` y el archivo tmp estaba bloqueado. La tercera vez, el proceso se pone a dormir hasta que el archivo quede desbloqueado. En esta ocasión hemos usado el modo `F_LOCK`.