postdata9
www.wuolah.com/student/postdata9

39693

sesion2.pdf

Módulo II

2° Sistemas Operativos

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

KEEP CALM AND ESTUDIA UN POQUITO

WUOLAH + #QuédateEnCasa

Sesión 2:

Llamadas al sistema para el SA (II)

Índice:

1. Llamadas al sistema relacionadas con los permisos de los archivos

1.1 umask

Los permisos de un fichero tienen la siguiente estructura:

rwx	rwx	rwx
111	111	111
U	G	O

Los 3 primeros bits son los permisos de usuario (read, write y execution); los 3 siguientes para grupo y los 3 últimos para otros. Si el bit está a 0, no está activado dicho permiso; si está a 1, está activado el permiso.

Como ejemplo, si tenemos 110 100 100 (640 en octal), vemos:

- con los 3 primeros bits, que el usuario tiene permiso de lectura y escritura;
- con los siguientes 3 bits, que el grupo tiene permiso de lectura;
- con los últimos bits, que otro tiene permiso de lectura.

umask es la llamada al sistema que fija la **máscara de creación de permisos** para un proceso. Puede formarse mediante una combinación OR de las nueve constantes de permisos (rwx para ugo).

```
mode_t umask(mode_t máscara);
```

- **máscara**: establece los permisos que NO se van a poder establecer a la hora de crear el fichero;
- esta llamada siempre tiene éxito y devuelve el valor anterior de la máscara.

Para entender esta llamada, si hacemos umask(640):

- 640 en binario → 110 100 100;
- con umask estamos indicando que los bits a 1 (que se corresponden con los permisos de lectura por user, group y other, y el permiso de escritura para user), NO puedan establecerse a la hora de crear un fichero. Es decir, que el fichero NO tendrá estos permisos.
- Por tanto, si establezco dicha máscara, y a la hora de crear el fichero utilizo las banderas (S_IRUSR | S_IRGRP | S_IWGRP | S_IXGRP | S_IWOTH), el fichero se va a crear con permisos de escritura para grupo y otro, y permiso de ejecución para grupo, pero no va tener permisos de lectura para usuario y para grupo (porque con umask hemos establecido que no se pudieran activar dichos permisos).

Si hacemos umask(0):

- en binario 000 000 000; estamos indicando que podemos asignar todos los permisos que queramos.

Sólo activaremos los permisos (pondremos el bit a 1) a aquellos permisos que NO queremos dar.

Se puede configurar el valor umask en **/etc/bashrc** o **/etc/profile** (este último para todos los usuarios). Por defecto, la mayoría de las distribuciones Linux establecen el valor en 0022 (022) o 0002 (002).

- El umask por defecto **0002** se utiliza para los **usuarios regulares**. Con esta máscara, los permisos predeterminados de los directorios son *775*, y los permisos predeterminados de los archivos son 664.
- El umask por defecto para el usuario **root** es **0022** , y como resultado, los permisos predeterminados de los directorios son *755*, y los permisos predeterminados de los archivos son 644.
- Para los directorios, los permisos de base son **0777** (**rwxrwxrwx**) y para los archivos son **0666** (**rw-rw-rw**).

1.2 stat, fstat y lstat

Estas funciones proporcionan información de estado de un fichero.

```
int stat(const char *nombre, struct stat *buffer);
int fstat(int fd, struct stat *buffer);
int lstat(const char *nombre, struct stat *buffer);
```

- **nombre:** nombre/path del archivo;
- **fd**: descriptor de archivo;
- **buffer**: es donde la función almacena toda la información del fichero, por eso es de tipo struct stat.
- devuelve 0 si no ha habido error; -1 si ha habido error y actualiza *errno* según el error.

Las diferencias entre ellas son:

- **stat():** recupera toda la información del archivo pasado como argumento;
- **lstat():** es igual que stat(), pero si el archivo pasado es un enlace simbólico, recupera la información del enlace, no del archivo en sí;
- **fstat():** es igual que stat(), pero en vez de especificar el archivo por el nombre, es especificado por el descriptor de archivo.

Los errores más comunes son:

- **EBADF**: el descriptor de fichero no es válido.
- **ENOENT**: el fichero no existe.
- **ENOTDIR**: un componente del prefijo de ruta no es un directorio.
- **ELOOP**: Se encontraron demasiados enlaces simbólicos al resolver la ruta.
- **ENAMETOOLONG:** path es muy largo.
- **EFAULT**: path apunta fuera del espacio de direcciones accesible.
- **EACCES**: se deniega el permiso de búsqueda en un componente del prefijo de ruta.
- **ENOMEM**: memoria libre de kernel insuficiente.

1.3 chmod y fchmod

Ambas llamadas cambian los permisos de un fichero existente, pero difieren en cómo se especifica el fichero: **chmod** con el **path** y **fchmod** con el **descriptor de fichero**, con un fichero previamente abierto por open.

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

- **path**: es el pathname del fichero a cambiar los permisos;
- **fd**: es el descriptor de fichero del archivo a cambiar los permisos;
- **permisos:** es una máscara de bits que especifica los permisos nuevo del fichero. El valor que puede tomar está en la tabla siguiente:

S_ISUID	0400	activar la asignación del UID del propietario al UID efectivo del proceso que ejecute el archivo.
S_ISGID	02000	activar la asignación del GID del propietario al GID efectivo del proceso que ejecute el archivo.
S_ISVTX	01000	activar sticky bit. En directorios significa un borrado restringido → un proceso no privilegiado no puede borrar o renombrar archivos del directorio salvo que tenga permiso de escritura y sea propietario.
S_IRWXU	00700	user (propietario del archivo) tiene permisos de lectura, escritura y ejecución
S_IRUSR	00400	lectura para el propietario (= S_IREAD no POSIX)
S_IWUSR	00200	escritura para el propietario (= S_IWRITE no POSIX)
S_IXUSR	00100	ejecución/búsqueda para el propietario (=S_IEXEC no POSIX)
S_IRWXG	00070	group tiene permisos de lectura, escritura y ejecución
S_IRGRP	00040	lectura para el grupo
S_IWGRP	00020	escritura para el grupo
S_IXGRP	00010	ejecución/búsqueda para el grupo
S_IRWXO	00007	other tienen permisos de lectura, escritura y ejecución
S_IROTH	00004	lectura para otros
S_IWOTH	00002	escritura para otros
S_IXOTH	00001	ejecución/búsqueda para otros

- devuelve: **0** en caso de éxito; **-1** en caso de error y se le asigna a errno el valor adecuado.

Algunos conceptos a tener en cuenta:

- Para cambiar los **bits de permisos de un archivo**, el **UID efectivo del proceso debe ser igual al del propietario del archivo**, o el proceso debe tener permisos de **root** o superusuario (UID efectivo del proceso debe ser 0).
- Si el UID efectivo del proceso no es cero y el grupo del fichero no coincide con el ID de grupo efectivo del proceso o con uno de sus ID's de grupo suplementarios, el **bit S_ISGID se desactivará**, aunque esto no provocará que se devuelva un error.
- Dependiendo del sistema de archivos, los bits S_ISUID y S_ISGID podrían desactivarse si el archivo es escrito.
- En algunos sistemas de archivos, **solo el root puede asignar el 'sticky bit'**, lo cual puede tener un significado especial (por ejemplo, para directorios, un archivo sólo puede ser borrado por el propietario o el root).

Los errores más comunes de **chmod()** son:

- **EACCES**: se deniega el permiso de búsqueda en un componente del prefijo de ruta.
- **EFAULT**: path apunta fuera del espacio de direcciones accesible.
- **EIO**: ocurre un error de E/S.
- **ELOOP**: Se encontraron demasiados enlaces simbólicos al resolver la ruta. ENAMETOOLONG: path es muy largo.
- **ENOENT**: el fichero no existe.
- **ENOMEM**: memoria libre de kernel insuficiente.
- **ENOTDIR**: un componente del prefijo de ruta no es un directorio.

- **EPERM**: el UID efectivo no coincide con el propietario del archivo y el proceso no tiene privilegios (Linux: no tiene la capacidad CAP_FOWNER).
- **EROFS**: el archivo nombrado reside en un sistema de archivos de solo lectura.

Los errores más comunes de **fchmod()** son:

- **EBADF**: el descriptor de fichero no es válido.
- **EIO**: ocurre un error de E/S.
- **EPERM**: el UID efectivo no coincide con el propietario del archivo y el proceso no tiene privilegios (Linux: no tiene la capacidad CAP_FOWNER).
- **EROFS**: el archivo nombrado reside en un sistema de archivos de solo lectura.

1.4 Ejercicio 1

Ejercicio 1. ¿Qué hace el siguiente programa?

```
/*
tarea3.c
Trabajo con llamadas al sistema del Sistema de Archivos "POSIX 2.10 compliant".
Este programa fuente está pensado para que se cree primero un programa con la parte de CREACION DE ARCHIVOS y se haga un ls -l para fijarnos en los permisos y entender la llamada umask. En segundo lugar (una vez creados los archivos) hay que crear un segundo programa con la parte de CAMBIO DE PERMISOS para comprender el cambio de permisos relativo a los permisos que actualmente tiene un archivo frente a un establecimiento de permisos absoluto.
*/
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<errno.h>
int main(int argc, char *argv[]) {
        int fd1,fd2;
        struct stat atributos;

        //CREACIÓN DE ARCHIVOS
        if( (fd1=open("archivo1",O_CREAT|O_TRUNC|O_WRONLY,S_IRGRP|S_IWGRP|S_IXGRP)) < 0){
                printf("\nError %d en open(archivo1,...)",errno);
                perror("\nError en open");
                exit(-1);
        }
        umask(0);

        if( (fd2=open("archivo2",O_CREAT|O_TRUNC|O_WRONLY,S_IRGRP|S_IWGRP|S_IXGRP)) < 0){
                printf("\nError %d en open(archivo2,...)",errno);
                perror("\nError en open");
                exit(-1);
        }
        //CAMBIO DE PERMISOS
        if(stat("archivo1",&atributos) < 0) {
                printf("\nError al intentar acceder a los atributos de archivo1");
                perror("\nError en lstat");
                exit(-1);
```

```
        }

        if(chmod("archivo1", (atributos.st_mode & ~S_IXGRP) | S_ISGID) < 0) {
                perror("\nError en chmod para archivo1");
                exit(-1);
        }

        if(chmod("archivo2", S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH) < 0) {
                perror("\nError en chmod para archivo2");
                exit(-1);
        }

        close(fd1);
        close(fd2);

        return 0;
}
```

Este ejercicio trabaja con las llamadas umask y chmod para cambiar permisos:

- En el primer if, crea y abre el archivo archivo1 de sólo escritura, permisos de lectura para el usuario, de escritura y de ejecución para grupo.

  Si el descriptor de fichero que devuelve es negativo, ha habido error en la apertura del fichero, con lo que sale del programa.
- Con la llamada umask(0), permito que se pueda crear los siguientes ficheros con cualquier permiso. Por tanto, a la hora de crear el fichero archivo2, se crea con los permisos que se le pasa como argumento a la llamada open.
- En el segundo if, crea y abre el archivo archivo2 de sólo escritura, con permisos de lectura, escritura y ejecución para grupo.

  Si el descriptor de fichero que devuelve es negativo, ha habido error en la apertura del fichero, con lo que sale del programa.
- En el tercer if, obtiene los atributos de archivo1. Si devuelve un valor negativo, ha habido error, con lo que sale del programa.
- En el cuarto if, le cambia los permisos al archivo1 con la llamada chmod. A los permisos que ya tenía el archivo, le quita el permiso de ejecución para grupo (~S_IXGRP) y le establece el ID del grupo (S_ISGID).
- En el quinto if, al archivo2 con la llamada chmod, le da permisos de lectura, escritura y ejecución para el usuario (S_IRWXU), permiso de lectura y escritura para grupo (S_IRGRP, S_IWGRP) y permiso de lectura para otros (S_IROTH).

Si hubiéramos hecho umask(030):

- 030 en binario es: 000 011 000;
- estaríamos negando que se le asignen a los ficheros siguientes los permisos de escritura y ejecución para grupo.
- Esto supone que cuando creamos el archivo2, los permisos S_IWGRP y S_IXGRP no se asignan (lo hemos prohibido con umask).

2. Funciones de manejo de directorios

Aunque los directorios se pueden leer utilizando las mismas llamadas al sistema que para los archivos normales, como la estructura de los directorios puede cambiar de un sistema a otro, los programas en este caso no serían transportables, por ello, se usa una biblioteca estándar de funciones de manejo de directorios, con funciones como: **opendir**, **readdir**, **closedir**, **seekdir**, **telldir** y **rewinddir**.

Para entender estas funciones, necesitamos conocer el **tipo DIR** y la **estructura dirent**. También serán necesarios el contenido de los archivos **<sys/type.h>** y **<dirent.h>**.

Tipo DIR

```
typedefstruct __dirdesc {      #if 0
        int     dd_fd;         /* descriptor de fichero*/
        long    dd_loc;        /* buf desplazamiento de entrada desde el último readdir() */
        long    dd_size;       /* cantidad de datos válidos en el buffer */
        long    dd_bsize;      /* cantidad de entradas leídas en el momento */
        long    dd_off;        /* current offset en el directorio (para telldir) */
        char    *dd_buf;       /* buffer de datos del directorio */
} DIR;
```

Estructura dirent

```
struct dirent {
        ino_t           d_ino;      /* número de inodo */
        off_t           d_off;      /* posición actual en el stream del directorio */
        unsigned short  d_reclen;   /* tamaño en bytes del registro devuelto */
        unsigned char   d_type;     /* tipo de fichero */
        char            d_name[256]; /* nombre del fichero */
};
```

Sin embargo, los únicos campos que están en el estándar POSIX.1 son **d_name** y **d_ino**.

2.1 opendir

Abre un directorio.

```
DIR *opendir(char *nombre)

DIR *fdopendir(int fd)
```

- **nombre:** es el nombre del directorio a abrir;
- **fd**: es el descriptor de fichero
- ambas devuelven la estructura DIR del directorio abierto; en caso de **error**, devuelven **NULL** y **errno** toma el valor correspondiente.

La diferencia entre **opendir** y **fdopendir** es que a opendir se le pasa el nombre del fichero y a fdopendir se le pasa el descriptor de archivo.

Los errores que puede dar:

- **EACCES**: permiso denegado.
- **EBADF**: fd no es un descriptor de fichero válido abierto para lectura.
- **EMFILE**: se ha llegado al límite por procesos en el número de descriptores de fichero abiertos.
- **ENFILE**: se ha llegado al número total del sistema de ficheros abiertos.
- **ENOENT**: el directorio no existe o el nombre es una cadena vacía.
- **ENOMEM**: insuficiente memoria para completar la operación.
- **ENOTDIR**: el nombre no es un directorio.

2.2 readdir

Lee la entrada de un directorio abierto; después de leer, adelanta el puntero una posición.

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

- **dirp**: puntero del stream del directorio a leer.
- devuelve **struct dirent**, o **NULL** si ha habido un error o se ha llegado al **final del stream** del directorio.

El error que puede dar es:

- **EBADF**: puntero al stream de directorio inválido.

2.3 closedir

Cierra el directorio pasado por argumento.

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

- **dirp**: puntero del stream del directorio a cerrar.
- devuelve un entero:
  - 0 se ha cerrado;
  - -1 en caso contrario.

El error que puede dar es:

- **EBADF**: puntero al stream de directorio inválido.

2.4 seekdir

Establece la posición de la siguiente llamada readdir() en el stream del directorio.

```
void seekdir(DIR *dirp, long loc);
```

- **dirp**: puntero del stream del directorio.
- **loc**: debe ser un valor devuelto por la llamada previa a telldir().
- No devuelve ningún valor.

2.5 telldir

```
long telldir(DIR *dirp);
```

- **dirp**: puntero del stream del directorio.
- devuelve la localización actual en el stream del directorio pasado como argumento. Si hay un error, devuelve -1 y errno se establece apropiadamente.

El error que puede dar es:

- **EBADF**: puntero al stream de directorio inválido.

2.6 rewinddir

Posiciona el puntero de lectura al principio del directorio.

```
void rewinddir(DIR *dirp);
```

- **dirp:** puntero del stream del directorio.
- No devuelve nada.

2.7 Ejercicio 2

Ejercicio 2. Realiza un programa en C utilizando las llamadas al sistema necesarias que acepte como entrada:

- **Un argumento que representa el 'pathname' de un directorio.**
- **Otro argumento que es un número octal de 4 dígitos (similar al que se puede utilizar para cambiar los permisos en la llamada al sistema chmod). Para convertir este argumento tipo cadena a un tipo numérico puedes utilizar la función strtol. Consulta el manual en línea para conocer sus argumentos.**

El programa tiene que usar el número octal indicado en el segundo argumento para cambiar los permisos de todos los archivos que se encuentren en el directorio indicado en el primer argumento.

El programa debe proporcionar en la salida estándar una línea para cada archivo del directorio que esté formada por:

<nombre_de_archivo> : <permisos_antiguos> <permisos_nuevos>

Si no se pueden cambiar los permisos de un determinado archivo se debe especificar la siguiente información en la línea de salida:

<nombre_de_archivo> : <errno> <permisos_antiguos>

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>      //Needed for open
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
```

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

	char *nom_dir;		//argumento 1, nombre del directorio
	int permisos;		//argumento 2, permisos
	DIR *directorio;	//directorio de nombre nom_dir
	struct dirent *arch;	//cada entrada del directorio nom_dir
	char pathname[1000];	//nombre de cada entrada del directorio
	struct stat atributos;	//atributos de pathname
	mode_t perm_arch;	//permisos antiguos de cada entrada del directorio

	//comprobamos que hay 3 argumentos de entrada: ejecutable + nombre directorio + permisos
	if(argc != 3){
		printf("ERROR en argumentos. \n El formato es ./ejercicio2 nombre_directorio permisos");
		exit(-1);
	}
	//si están los 3 argumentos
	else{
		//guardamos el nombre del fichero
		 nom_dir = argv[1];

		//guardamos los permisos de la cadena de char en un long int en octal
		/*la función strtol(cadena a convertir, NULL, base)*/
		permisos = strtol(argv[2], NULL, 8);

		//abrimos el directorio
		directorio = opendir(nom_dir);

		//comprobamos si ha habido error en la apertura
		if(directorio == NULL){
			printf("ERROR en la apertura del directorio %s", nom_dir);
			exit(-2);
		}
	}

	//leemos mientras no haya error o no llegue al final del stream del directorio
	while((arch = readdir(directorio)) != NULL){

		//si es el directorio padre o actual, ignoramos
		if(strcmp(arch->d_name, ".") != 0 && strcmp(arch->d_name, "..") != 0){

			//almacenamos la ruta con el nombre de la carpeta y el archivo en pathname
			//de forma que quedaría carpeta/archivo
			sprintf(pathname, "%s/%s", nom_dir, arch->d_name);

			//obtenemos los atributos del archivo: si hay error, lo mostramos por pantalla
			if(stat(pathname, &atributos) < 0){
				printf("ERROR en la obtención de los atributos de %s - %s\n", pathname,
				strerror(errno));
			}
			//si no ha habido problema al obtener los atributos
			else{

				//guardamos los permisos
				perm_arch = atributos.st_mode;
```


```
                        //modificamos los permisos de pathname a permisos (argumento 2)
                        if(chmod(pathname, permisos) < 0){

                                //si hay error, mostramos el nombre + error + permisos antiguos
                                printf("%s : %o  %s \n", arch->d_name, perm_arch, strerror(errno));
                        }
                        else{
                                //obtenemos los atributos actualizaodos
                                stat(pathname, &atributos);

                                //si no ha habido error, mostramos:
                                //nombre + permisos antiguos + permisos nuevos
                                printf("%s : %o  %o\n", arch→d_name, perm_arch,
                                atributos.st_mode);
                        }
                }
        }
        }

        return 0;
}
```

2.8 Ejercicio 3

Ejercicio 3. Programa una nueva orden que recorra la jerarquía de subdirectorios existentes a partir de uno dado como argumento y devuelva la cuenta de todos aquellos archivos regulares que tengan permiso de ejecución para el grupo y para otros. Además del nombre de los archivos encontrados, deberá devolver sus números de inodo y la suma total de espacio ocupado por dichos archivos. El formato de la nueva orden será:

$> ./buscar <pathname>

donde <pathname> especifica la ruta del directorio a partir del cual queremos que empiece a analizar la estructura del árbol de subdirectorios. En caso de que no se le de argumento, tomará como punto de partida el directorio actual. Ejemplo de la salida después de ejecutar el programa:

Los i-nodos son:

./a.out *55*

./bin/ej 123

./bin/ej2 87

...

Existen 24 archivos regulares con permiso x para grupo y otros

El tamaño total ocupado por dichos archivos es 2345674 bytes

```
#include <sys/types.h>                  #include <errno.h>
#include <unistd.h>                     #include <stdlib.h>
#include <sys/stat.h>                   #include <dirent.h>
#include <fcntl.h>      //Needed for open   #include <string.h>
#include <stdio.h>
```

```
/*definimos una macro para comprobar si un fichero tiene:
   - permiso x para grupos       -> 000010
   - permiso x para otros        -> 000001
   1. Al modo le quitamos el flag de tipo de fichero (0170000)

   2. Le hacemos & 011 (para ver si tiene activado los permisos x de GRP (010) y OTH(011))
   (GRP & OTH = 011)

   3. Si el resultado es 011, tiene los permisos x activados para GRP y OTH

   Ejemplo: st_mode = 100675 (en octal)
      st_mode: 100675 -> 001 000 000 110 111 101 en binario
      S_IFMT:  170000  -> 001 111 000 000 000 000 en binario
      ~S_IFMT:         -> 110 000 111 111 111 111

      Por tanto, st_mode & ~ S_IFMT:
                  001 000 000 110 111 101
                & 110 000 111 111 111 111
                ---------------------------------------
                  000 000 000 110 111 101
                              u   g   o
      A esta altura, nos hemos quedado con los permisos para user, group y other
         El permiso x de other es (0000001):          El permiso x de group es (0000010):
            000 000 000 000 000 001                      000 000 000 000 000 010

      La suma de estos dos es 000011(octal), por tanto, volviendo al resultado de st_mode & ~S_IFMT, le
      hacemos & 011:
                  000 000 000 110 111 101
                & 000 000 000 000 001 001
                -----------------------------------
                  000 000 000 000 001 001 -> en octal es 000011

      Al ser el resultado 011 en octal, indica que el fichero tiene permisos x para gr y oth; en caso que no
      saliera 011, supondría que dichos bits no están activados, por tanto, no tendría permisos x para gr y oth.
*/
#define permisos(mode) (((mode & ~S_IFMT) & 011) == 011)

/*estructura para almacenar:
   - el número de bytes totales
   - el número de archivos regulares
*/
struct salida{
        int bytes;
        int n;
};

/*función para recorrer el subárbol
        - recibe como argumento la ruta/nombre del directorio a recorrer
        - devuelve la estructura salida con:
                el número de archivos regulares con permiso x en gr y oth
                el número de bytes totales de los archivos regulares con permiso x en gr y oth*/
```

```
struct salida buscar(char nom_dir[1000]){
        DIR *directorio;
        struct dirent *arch;
        char pathname[1000];
        struct stat atributos;
        struct salida s;   s.bytes = 0;      s.n = 0;

        //abrimos el directorio
        directorio = opendir(nom_dir);

        //comprobamos si ha habido error
        if(directorio == NULL){
                printf("ERROR en la apertura del directorio %s", nom_dir);
                exit(-2);
        }

        //leemos el directorio hasta llegar al final
        while((arch = readdir(directorio)) != NULL){

                //si es el directorio padre o actual, ignoramos
                if(strcmp(arch->d_name, ".") != 0 && strcmp(arch->d_name, "..") != 0){
                        //almacenamos la ruta con el nombre de la carpeta y el archivo en pathname
                        //de forma que quedaría carpeta/archivo
                        sprintf(pathname, "%s/%s", nom_dir, arch->d_name);

                        //obtenemos los atributos del archivo: si hay error, lo mostramos por pantalla
                        if(stat(pathname, &atributos) < 0){
                                printf("ERROR en la obtención de los atributos de %s - %s\n", pathname,
                                strerror(errno));
                        }
                        //si no ha habido problema al obtener los atributos
                        else{
                                //si es un directorio
                                if(S_ISDIR(atributos.st_mode)){
                                        //volvemos a llamar a buscar
                                        s = buscar(pathname);
                                }
                                //si es un archivo regular y tiene permisos de ejecución para grupo y otro
                                else if(S_ISREG(atributos.st_mode) && permisos(atributos.st_mode)){
                                        //Imprimimos el nombre y el número de inodo
                                        printf("\t%s %li\n", pathname, atributos.st_ino);

                                        //sumamos el tamaño del archivo
                                        s.bytes += atributos.st_size;

                                        //aumentamos el número de archivos regulares
                                        s.n += 1;
                                }
                        }
                }
        }
        return s;
}
```

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

        char nom_dir[1000];
        struct salida s;

        //si se especifica la ruta ./buscar <ruta>
        if(argc == 2){
                //copiamos el nombre del directorio en nom_dir
                strcpy(nom_dir, argv[1]);
        }
        //si no se especifica la ruta
        else if(argc == 1){
                //copiamos la ruta actual en nom_dir
                strcpy(nom_dir, ".");
        }
        //en cualquier otro caso, error
        else{
                printf("ERROR en argumentos.\n El formato es:\ ./buscar <ruta> ó ./buscar");
                exit(-1);
        }

        //imprimimos el mensaje
        printf("Los i-nodos son:\n");

        //recorremos el directorio con la función buscar; la función devuelve el número de bytes totales
        s = buscar(nom_dir);

        //imprimimos el número de archivos regulares y el tamaño total en bytes
        printf("\nExisten %i archivos regulares con permiso x para grupos y otros.\n", s.n);
        printf("El tamaño total ocupado por dichos archivos es %i bytes.\n", s.bytes);

        return 0;
}
```

2.9 nftw

La función nftw() permite **recorrer recursivamente un sub-árbol** y realizar alguna operación sobre los archivos del mismo.

```
#include <ftw.h>

int nftw (const char *dirpath, int *(función), int nopenfd, int flags);
```

- **dirpath:** nombre del directorio a recorrer;
- **nopenfd:** la función nftw abre un descriptor de archivo por nivel de árbol; este parámetro indica el máximo número de descriptores que puede usar, es decir, el nivel de profundidad al que puede llegar abriendo directorios;

- **flags:** modifica la operación de la función, es creado mediante OR con cero o constantes. Puede tomar los siguientes valores:

FTW_DIR	Realiza un chdir (cambia de directorio) en cada directorio antes de procesar su contenido. Se utiliza cuando func debe realizar algún trabajo en el directorio en el que el archivo especificado por su argumento pathname reside.
FTW_DEPTH	Realiza un recorrido postorden del árbol. Esto significa que nftw llama a func sobre todos los archivos (y subdirectorios) dentro del directorio antes de ejecutar func sobre el propio directorio.
FTW_MOUNT	No cruza un punto de montaje.
FTW_PHYS	Indica a nftw que nos desreferencie los enlaces simbólicos. En su lugar, un enlace simbólico se pasa a func como un valor typedflag de FTW_SL.

- devuelve 0 si recorre completamente el árbol; -1 si error o el primer valor no cero devuelto por función.

nftw() para cada archivo del directorio que abre, llama a función, y le pasa 4 argumentos concretos.

La función que se pasa por parámetros es la que va a realizar las operaciones sobre los ficheros de un directorio; es la que tenemos que programar nosotros. Necesariamente, su cabecera será la siguiente:

```
int *(función) (const char *pathname, const struct stat *atributos, int typeflag, struct FTW *ftwbuf);
```

- **pathname**: ruta del archivo;
- **atributos:** estructura stat que contiene información del archivo;
- **typeflag:** añade información adicional sobre el archivo y tiene uno de los siguientes nombres simbólicos:

FTW_D	Es un directorio
FTW_DNR	Es un directorio que no puede leerse (no se lee sus descendientes).
FTW_DP	Estamos haciendo un recorrido post-orden de un directorio, y el ítem actual es un directorio cuyos archivos y subdirectorios han sido recorridos.
FTW_F	Es un archivo de cualquier tipo diferente de un directorio o enlace simbólico.
FTW_NS	stat ha fallado sobre este archivo, probablemente debido a restricciones de permisos. El valor statbuf es indefinido.
FTW_SL	Es un enlace simbólico. Este valor se retorno solo si nftw se invoca con FTW_PHYS
FTW_SLN	El ítem es un enlace simbólico perdido. Este se da cuando no se especifica FTW_PHYS.

- **ftwbuf:** un puntero a una estructura que se define de la forma:

```
struct FTW {
    int base;        /* Desplazamiento de la parte base del pathname */
    int level;       /*Profundidad del archivo dentro recorrido del árbol */
};
```

- devuelve 0 si no ha habido problema, con lo que nftw() continúa con el resto del árbol; en caso de error, devuelve un valor distinto de cero, con lo que nftw() para inmediatamente y retorna el mismo valor que esta función.

Por defecto, ntfw realiza un recorrido no ordenado en preorden del árbol, procesando primero cada directorio antes de procesar los archivos y subdirectorios dentro del directorio.

Ejercicio 4. Implementa de nuevo el programa buscar del ejercicio 3 utilizando la llamada al sistema nftw.

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
#include <ftw.h>

#define permisos(mode) (((mode & ~S_IFMT) & 011) == 011)

int BYTES = 0;
int N = 0;

//función para buscar archivos regulares con permisos de ejecución de grupos y otros
int buscar(const char *path, const struct stat* stat, int flags, struct FTW* ftw){

	//ignoramos si es el fichero actual o el padre
	if(strcmp(path, ".") != 0 && strcmp(path, "..") != 0){
		//si es un archivo regular con permisos de ejecución para grupos y otros
		if(S_ISREG(stat->st_mode) && permisos(stat->st_mode)){

			//Imprimimos el nombre y el número de inodo
			printf("\t%s %li\n", path, stat->st_ino);

			//sumamos el tamaño del archivo
			BYTES += stat->st_size;

			//aumentamos el número de archivos regulares
			N += 1;
		}
	}
	return 0;
}

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

	char nom_dir[1000];

	//si se especifica la ruta ./buscar <ruta>
	if(argc == 2){
		//copiamos el nombre del directorio en nom_dir
		strcpy(nom_dir, argv[1]);
	}
	//si no se especifica la ruta
	else if(argc == 1){
		//copiamos la ruta actual en nom_dir
		strcpy(nom_dir, ".");
	}
	//en cualquier otro caso, error
	else{
		printf("ERROR en argumentos.\n El formato es: ./buscar <ruta> ó ./buscar");
```

```
            exit(-1);
        }

        //imprimimos el mensaje
        printf("Los i-nodos son:\n");

        //función para recorrer el árbol
        if(nftw(nom_dir, buscar, 5, 0) != 0){
            printf("ERROR en nftw");
            exit(-2);
        }

        //imprimimos el número de archivos regulares y el tamaño total en bytes
        printf("\nExisten %i archivos regulares con permiso x para grupos y otros.\n", N);
        printf("El tamaño total ocupado por dichos archivos es %i bytes.\n", BYTES);

        return 0;
}
```

3. Preguntas de repaso

1. **¿Cómo se acceden a los atributos de un archivo?**

   struct stat atributos;

   stat("archivo.c", &atributos)

   → si esto devuelve un número negativo, ha habido error al obtener los atributos.

2. **¿Dónde podemos ver los campos del struct stat?**

   En man 2 stat.

3. **¿Qué contiene el campo st.mode del struct stat?**

   Contiene los permisos del fichero y el tipo de fichero (si es regular, directorio, enlace simbólico), se corresponde con la salida del ls -l, los permisos drwxrwxrwx.

4. **¿Cómo podemos ver el contenido de st.mode?**

   Para verlo en decimal: printf(" %d", atributos.st_mode).

   En octal: printf(" %o", atributos.st_mode);

5. **¿Cómo podemos implementar en C la condición de que si un fichero es regular haga algo?**

   if(S_ISREG(atributos.st_mode))

6. **¿Cómo podemos ver todas las macros que hay para st_mode?**

   man 7 inode

7. **¿Cómo podemos leer todos los elementos que hay en un directorio?**

   ```
   while( (elemento = readdir(directorio) != NULL );
   ```

8. **¿Cómo podemos leer todos los elementos que hay en un directorio, evitando que lea el directorio actual y el padre?**

   ```
   //leemos una entrada del directorio
   while( (elemento = readdir(directorio)) != NULL){
       //con el if ignoramos el directorio actual y padre
       if( strcmp(elemento→d_name, ".") != 0 && strcmp(elemento→d_name, "..") != 0 ){
           …
       }
   }
   ```