sesion1.pdf

Módulo II

2º 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 1:

Llamadas al sistema para el SA (I)

Índice:

1. Objetivos del Módulo II

Las llamadas al sistema utilizadas siguen el estándar POSIX 1003.1 para la interfaz del sistema operativo. Este estándar define los servicios que debe proporcionar un sistema operativo.

La siguiente tabla muestra los números de sección del manual y los tipos de páginas que contienen. Se puede acceder a una determinada sección utilizando la siguiente orden:

```
$> man <numero_seccion> <orden>
```

1	Programas ejecutables y guiones del intérprete de órdenes
2	**Llamadas del sistema (funciones servidas por el núcleo)**
3	**Llamadas de la biblioteca (funciones contenidas en la bibliotecas del sistema)**
4	Ficheros especiales (se encuentran generalmente en /dev)
5	Formato de ficheros y convenios p.ej. /etc/passwd
6	Paquetes de macros y convenios p.ej man(7), groff(7)
7	Órdenes de administración del sistema (generalmente sólo son para usuario **root**)

Cuando se produce un error, todas las funciones devuelven el código del error producido en la variable errno, que se puede imprimir con la función perror. Esta función devuelve un literal descriptivo de cuál ha sido el error concreto. En el archivo <errno.h> hay una lista con todas las circunstancias de error que hay para las llamadas al sistema.

strace: proporciona la información de las llamadas al sistema que han sido invocadas en la ejecución de un programa y de las señales recibidas.

2. Entrada/Salida de archivos regulares

Vamos a trabajar con el sistema de archivos mediante las **llamadas al sistema**. La mayor parte de las E/S se pueden realizar con las llamadas: **creat, open, read, write, lseek** y **close**. Estas funciones son E/S sin buffer, es decir, que cuando se realiza un read o un write no se almacena en un buffer de la biblioteca, sino que se invoca a una llamada al sistema en el núcleo.

Cuando creamos un archivo, el núcleo del sistema le asocia un **descriptor de archivo**, que es un número entero NO negativo. El shell de Linux por convenio le suele asociar el descriptor de archivo:

- **0** a la entrada estándar de un proceso, conocido como la constante simbólica **STDIN_FILENO;**
- **1** a la salida estándar, conocido como la constante simbólica **STDOUT_FILENO;**
- **2** a la salida de error estándar, conocido como la constante simbólica **STDERR_FILENO;**

  estas constantes se encuentran definidas en **<unistd.h>** (está en /usr/include).

Cada archivo abierto tiene una posición de lectura/escritura actual, **current file offset**:

- es un **entero no negativo;**
- mide el **número de bytes** desde el comienzo del archivo para indicar en qué posición nos encontramos leyendo o escribiendo;
- cuando se abre un archivo, esta posición **está inicializada a 0**, a menos que se especifique en la opción O_APPEND;
- una vez abierto el archivo, podemos cambiar esta posición con la llamada al sistema **lseek**.

2.1 creat

Llamada al sistema para crear un fichero.

```
int creat(char* nombre, int modo);
```

- **nombre**: es el nombre del fichero a crear;
- **modo:** indica el modo en el que se va a abrir el archivo una vez creado; los posibles modos son:
  - O_RDONLY: solo lectura;
  - O_WRONLY: solo escritura;
  - O_RDWR: lectura y escritura;
- devuelve un descriptor si el fichero se ha creado y abierto, y -1 si no;
- equivale a open(nombre, O_RDWR | O_CREAT, permisos).

2.2 open

Esta llamada al sistema es para abrir archivos que ya existen o crear uno nuevo.

```
int open(char* nombre, int modo, int permisos);
```

- **nombre**: es el nombre del fichero que se quiere abrir;
- **modo**: es la forma en la que se desea trabajar con el fichero, si queremos leer, escribir, leer y escribir, crear el fichero; algunas constantes que definen los modos básicos son:

O_RDONLY	abre en modo solo lectura
O_WRONLY	abre en modo solo escritura
O_RDWR	abre en modo lectura-escritura
O_CREAT	crea el fichero y lo abre (si existía, lo machaca)
O_EXCL	se usa con el modo anterior, O_CREAT; si el fichero existe, devuelve un error
O_TRUNC	si el fichero existe y es un fichero regular, y tiene permisos de escritura, trunca su longitud a 0; si es un FIFO o de dispositivo de terminal, esta opción se ignora. En otro caso, el efecto de esta flag es indefinido.

  - Para poder usar varios modos se realiza un "or" lógico, de la siguiente forma:
    O_CREAT | O_WRONLY.

- **permisos**: este parámetro se debe de utilizar cuando creemos un archivo, cuando incluyamos O_CREAT en el modo. Algunos posibles permisos son los siguientes:

S_IRWXU	00700 el propietario tiene el permiso para leer, escribir y ejecutar
S_IRUSR	00400 el usuario tiene permiso de lectura
S_IWUSR	00200 el usuario tiene permito de escritura
S_IXUSR	00100 el usuario tiene permiso de lectura-escritura

  para darle permisos a los grupos, cambiamos la última letra U por G y USR por GRP;
  para darle permisos a otros, cambiamos la última letra U por O y USR por OTH.

- devuelve un **descriptor válido** de fichero; si no se ha podido abrir devuelve el valor **-1**.

2.3 read y write

Son llamadas al sistema para leer y escribir en un fichero.

```
int read(int fichero, void* buffer, unsigned bytes);
int write(int fichero, const void* buffer, unsigned bytes);
```

- **fichero:** es el descriptor de fichero del que se quiere leer/escribir;
- **buffer:** donde se almacena lo que se lee / lo que queremos escribir;
- **bytes:** es la cantidad de bytes que se quiere leer del archivo/escribir en el archivo;
- devuelve el número de bytes que realmente se han transferido; es particularmente útil (en read) ya que sirve para saber si hemos llegado al final del fichero o no. En caso de error, retornan un -1.

2.4 lseek

Esta llamada modifica el **current file offset,** el puntero de lectura/escritura de un archivo abierto, es decir, puede modificar la posición actual.

```
long lseek(int fichero, long desplazamiento, int origen);
```

- **fichero:** es el descriptor del archivo abierto;
- **desplazamiento:** es un entero largo que indica cuántos bytes hay que moverse y puede tomar valores negativos para retroceder siempre que se pueda;
- **origen:** indica a partir de dónde quiero moverme, desplazamiento bytes;
- tanto desplazamiento como origen pueden tomar estos valores:

0	SEEK_SET	inicio de fichero
1	SEEK_CUR	la posición actual del puntero, del current offset
2	SEEK_END	final del fichero

- lseek devuelve un entero largo que es la posición absoluta donde se ha posicionado el puntero o -1 si hubo error.

Esta llamada puede usarse para leer la posición actual del puntero.

2.5 close

Llamada para cerrar un fichero abierto.

```
int close(int fichero);
```

- **fichero**: es el descriptor del archivo que se quiere cerrar;
- devuelve 0 si se ha cerrado correctamente, -1 si ha habido problemas.

Ejercicio 1. ¿Qué hace el siguiente programa? Probad tras la ejecución del programa las siguientes órdenes del shell: $> cat archivo y $> od -c archivo.

```
/*
tarea1.c
Trabajo con llamadas al sistema del Sistema de Archivos "POSIX 2.10 compliant". Probar tras la ejecución del programa: $>cat archivo y $> od -c archivo
*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>          /* Primitive system data types for abstraction of implementation-dependent
                                   data types. POSIX Standard: 2.6 Primitive System Data Types
                                   <sys/types.h>*/
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

char buf1[]="abcdefghij";
char buf2[]="ABCDEFGHIJ";

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

        int fd;

        //primer if
        if( (fd=open("archivo", O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR | S_IWUSR))<0){
                printf("\nError %d en open",errno);
                perror("\nError en open");
                exit(EXIT_FAILURE);
        }

        //segundo if
        if(write(fd,buf1,10) != 10){
                perror("\nError en primer write");
                exit(EXIT_FAILURE);
        }

        //tercer if
        if(lseek(fd,40,SEEK_SET) < 0){
                perror("\nError en lseek");
                exit(EXIT_FAILURE);
        }

        //cuarto if
        if(write(fd,buf2,10) != 10){
                perror("\nError en segundo write");
                exit(EXIT_FAILURE);
        }

        return EXIT_SUCCESS;
}
```

Vamos a entender qué hace el programa:

- **Primer if:** crea "archivo" (O_CREAT) desde cero (O_TRUNC) para escribir (O_WRONLY) con permisos de lectura (S_IRUSR) y escritura (S_IWUSR) para el usuario. Si el descriptor de archivo es negativo, ha habido error, con lo que da un mensaje de error por pantalla y termina el programa; si no ha habido error, devuelve el descriptor de archivo que es un entero positivo.
- **Segundo if:** si hemos conseguido crear/abrir el fichero, en esta función se intentará escribir en él. Para escribir, a write le pasamos 3 argumentos:
  - fd: que es el descriptor de "archivo" devuelto en la función anterior;
  - buf1: que es "abcdefghij", lo que queremos escribir en el archivo;
  - 10: queremos escribir 10 bytes en el archivo.

  La función write devuelve el número de bytes que se han escrito; si no se ha escrito 10 bytes, ha habido un error, lo muestra por pantalla y termina el programa.
- **Tercer if:** Si hemos podido crear y escribir en el archivo, ahora vamos a cambiar el *current file offset,* es decir, la posición en la que se encuentra para escribir. Lo modificamos para que en "archivo" (fd) se coloque a 40 bytes (40) del inicio (SEEK_SET). Esta función devuelve la posición absoluta en la que se ha posicionado el puntero, por lo que si es un valor negativo es porque ha habido un error y termina el programa.
- **Cuarto if:** Escribimos en la posición en la que se encuentra el puntero, el buf2 que es "ABCDEFGHIJ".

Por tanto, el programa escribe dos cadenas de caracteres, "abcdefghij" al principio del archivo y "ABCDEFGHIJ" a 40 bytes del inicio. Tras ejecutar el programa, se nos crea un fichero *archivo* en el directorio actual.

```
@postdata9:~$ gcc -o tarea1 tarea1.c
@postdata9:~$ ./tarea1
```

Si ejecutamos cat y od, nos muestra:

```
@postdata9:~$ cat archivo
    abcdefghijABCDEFGHIJ
@postdata9:~$ od archivo
    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
    0000040 \0 \0 \0 \0 \0 \0 \0 \0  A  B  C  D  E  F  G  H
    0000060  I  J
    0000062
```

Con cat vemos el contenido del archivo, que es abcdefghijABCDEFGHIJ, tal y como se especifica en el programa; y con od vemos el contenido del archivo byte a byte.

2.7 Ejercicio 2

Ejercicio 2. Implementa un programa que realice la siguiente funcionalidad. El programa acepta como argumento el nombre de un archivo (pathname), lo abre y lo lee en bloques de tamaño 80 Bytes, y crea un nuevo archivo de salida, salida.txt, en el que debe aparecer la siguiente información:

Bloque 1

// Aquí van los primeros 80 Bytes del archivo pasado como argumento.

Bloque 2

// Aquí van los siguientes 80 Bytes del archivo pasado como argumento.

…

Bloque n

// Aquí van los siguientes 80 Bytes del archivo pasado como argumento.

Si no se pasa un argumento al programa se debe utilizar la entrada estándar como archivo de entrada.

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

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

    //descriptores de archivos de los ficheros de
    //entrada y de salida
    int entrada, salida;

    //bytes leídos/escritos
    int bytes = 80;

    //buffer del que vamos a leer y escribir
    char buffer[81];

    //vamos a guardar el Bloque n para
    //escribirlo en el fichero
    char bloque[10];

    //escribimos el salto de línea
    char salto_linea[3];

    //si le hemos pasado un argumento al programa
    if(argc == 2){

        //abrimos el fichero para solo lectura
        entrada = open(argv[1], O_RDONLY);
    }
    //si no le hemos pasado un archivo
    else{

        //cogemos la entrada estándar
        entrada = STDIN_FILENO;
    }

    //vemos si hemos tenido un problema al abrir el archivo
    if(entrada < 0){
        printf("Error en la apertura del archivo de entrada.");
        exit(-1);
    }

    //creamos el archivo de salida
    salida = open("salida.txt", O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR | S_IWUSR);
```

```
        //vemos si hemos tenido un problema al crear el archivo
        if(salida < 0){
                printf("Error en la apertura del archivo de salida.");
                exit(-2);
        }

        //vamos leyendo hasta que lleguemos a fin de fichero, la condición de parada del for es hasta que
        //read devuelva 0 que será cuando llegue a final de fichero
        for(int i = 0; read(entrada, buffer, bytes); i++){

                //para escribir en el archivo Bloque numero cuando vamos a escribir un bloque
                sprintf(bloque, "Bloque %d", i);
                write(salida, bloque, sizeof(bloque));

                //para escribir salto de línea
                write(salida, "\n", sizeof("\n"));

                //escribimos lo que hemos leído de entrada
                write(salida, buffer, bytes);

                //para escribir salto de línea
                write(salida, "\n", sizeof("\n"));

                bl = i;
        }
}
```

Modificación adicional. ¿Cómo tendrías que modificar el programa para que una vez finalizada la escritura en el archivo de salida y antes de cerrarlo, pudiésemos indicar en su primera línea el número de etiquetas "Bloque i" escritas de forma que tuviese la siguiente apariencia?:

El número de bloques es <no_bloques>
Bloque 1
// Aquí van los primeros 80 Bytes del archivo pasado como argumento.
…

Es el mismo código de antes, sólo cambia en la última parte:

```
        //hacemos espacio para escribir el número de etiquetas al principio del fichero, con lo que
        //avanzamos el current file offset
        lseek(salida, sizeof("El número de bloques es 100"), SEEK_SET);

        //for de escritura en salida
        for(int i = 0; read(entrada, buffer, bytes); i++){
                sprintf(bloque, "Bloque %d", i);
                write(salida, bloque, sizeof(bloque));
                write(salida, "\n", sizeof("\n"));
                write(salida, buffer, bytes);
                write(salida, "\n", sizeof("\n"));
                bl = i;
        }

        //al terminar de escribir, nos posicionamos al principio del fichero
        lseek(salida, 0, SEEK_SET);

        //escribimos la línea con el número de bloques
        sprintf(bloque, "El número de bloques es %d", bl);

        //escribimos la frase al principio del fichero
        write(salida, bloque, sizeof(bloque));
```

3. Metadatos de un Archivo

En este apartado nos centramos en las características adicionales del sistema de archivos y en las propiedades de un archivo (metadatos y atributos).

3.1 Tipo de archivos

Linux soporta los siguientes tipos de archivos:

- **Archivo regular**: contiene cualquier tipo de dato, binario o de texto.
- **Archivo de directorio**: es un archivo que contiene el nombre de otros archivos (incluidos directorios) y punteros a la información de esos archivos.
- **Archivo especial de dispositivo de caracteres**: se utiliza para representar ciertos tipos de dispositivos en un sistema.
- **Archivo especial de dispositivo de bloques**: se utiliza para representar discos duros, CDROM, …
- **FIFO**: este archivo se utiliza para la comunicación entre procesos (IPC).
- **Enlace simbólico**: es un archivo que se utiliza para apuntar a otro.
- **Socket**: se usa para la comunicación en red entre procesos y para comunicar procesos en un único nodo.

3.2 Estructura stat

La estructura stat contiene los metadatos de un archivo en una representación concreta. Dicha estructura se puede obtener mediante la llamada al sistema **stat**.

La representación que tiene la estructura stat es la siguiente:

```
struct stat{
    //dispositivo en el que se encuentra el archivo
    dev_t st_dev;

    //tipo de dispositivo en el que se encuentra el archivo
    dev_t st_rdev;

    //inodo asociado al archivo
    ino_t st_ino;

    //permisos de archivo e información del archivo
    mode_t st_mode;  (man 7 inode)

    //número de enlaces duros con el archivo
    nlink_t st_nlink;

    //identidad de usuario del propietario del archivo (UID)
    uid_t st_uid;

    //identidad de grupo del propietario del archivo (GID)
    gid_t st_gid;

    //tamaño del archivo en caracteres o bytes
    off_t st_size;

    //tamaño de bloque preferido para operaciones de E/S para el SA; este tamaño coincide con el tamaño de bloque de formateo del SA
    unsigned long st_blksize;

    //número de bloques asignados, da el tamaño en bloques de 512 bytes
    unsigned long st_blocks;

    //último acceso, modificado por mknod(2), utime(2), read(2), write(2), truncate(2)
    time_t   st_atime;

    //última modificación, modificado por mknod(2), utime(2), write(2)
    time_t st_mtime;

    //último cambio en los permisos, se modifica al escribir o actualizar el inodo
    time_t st_ctime;

};
```

Se definen las siguientes macros POSIX para comprobar el tipo de fichero:

- S_ISLNK(st_mode) Verdadero si es un enlace simbólico (soft);
- S_ISREG(st_mode) Verdadero si es un archivo regular;
- S_ISDIR(st_mode) Verdadero si es un directorio;
- S_ISCHR(st_mode) Verdadero si es un dispositivo de caracteres;
- S_ISBLK(st_mode) Verdadero si es un dispositivo de bloques;
- S_ISFIFO(st_mode) Verdadero si es una cauce con nombre (FIFO);
- S_ISSOCK(st_mode) Verdadero si es un socket;

3.3 Permisos de acceso a archivos

El valor **st_mode** codifica además del tipo de archivo, los permisos de acceso. Disponemos de 3 categorías: **usr(owner), group** y **other** para establecer los permisos de lectura, escritura y ejecución y se utilizan de forma diferente según la llamada al sistema.

- Cuando queremos abrir cualquier tipo de archivo, debemos tener permiso de ejecución en el directorio en el que se encuentra, de aquí que el bit de permiso de ejecución para directorios se llame **bit de búsqueda**.
- El permiso de lectura de un directorio nos permite leer el directorio, es decir, nos proporciona una lista con todos los nombres de archivo que se encuentran en él; el permiso de ejecución nos permite pasar a través del directorio.
- El permiso de lectura para un archivo determina si podemos abrir para lectura un archivo existente: los flags O_RDONLY y O_RDWR para la llamada open.
- El permiso de escritura para un archivo determina si podemos abrir para escritura un archivo existente: los flags O_WRONLY y O_RDWR para la llamada open.
- Debemos tener permiso de escritura en un archivo para poder especificar el flag O_TRUNC en la llamada open.
- **No podemos crear un nuevo archivo en un directorio a menos que tengamos permisos de escritura y ejecución en dicho directorio.**
- **Para borrar un archivo existente necesitamos permisos de escritura y ejecución en el directorio que contiene el archivo. No necesitamos permisos de lectura o escritura en el archivo.**
- El permiso de ejecución para un archivo debe estar activado si queremos ejecutar el archivo usando cualquier función de la familia exec o si es un script de un shell. Además el archivo debe ser regular.

Las banderas para trabajar con st_mode, referente a **tipo de archivo:**

S_IFMT	0170000	máscara de bits para los campos de bit del tipo de archivo. Esta máscara & st_mode nos da las siguientes banderas, para identificar el tipo de archivo.
S_IFSOCK	0140000	socket.
S_IFLNK	0120000	enlace simbólico.
S_IFREG	0100000	archivo regular.
S_IFBLK	0060000	Dispositivo de bloques
S_IFDIR	0040000	Directorio
S_IFCHR	0020000	Dispositivo de caracteres
S_IFIFO	0010000	Cauce con nombre (FIFO)

Las banderas para trabajar con st_mode, referente a **los permisos**:

- **Los 3 bits especiales:**

S_ISUID	0004000	bit SUID
S_ISGID	0002000	bit SGID. Para un directorio, si está activado este bit, indica que cualquier fichero que se cree en ese directorio heredará el grupo del directorio padre, no del ID del grupo efectivo del proceso que lo crea.
S_ISVTX	0001000	sticky bit, si está activo, indica que un fichero puede ser renombrado o eliminado solo por el propietario del archivo, por el propietario del directorio y por un proceso privilegiado.

- **Para el usuario (propietario del archivo):**

S_IRWXU	0000700	user tiene permisos de lectura, escritura y ejecución.
S_IRUSR	0000400	user tiene permiso de lectura
S_IWUSR	0000200	user tiene permiso de escritura
S_IXUSR	0000100	user tiene permiso de ejecución

- **Para el grupo:**

S_IRWXG	0000070	group tiene permisos de lectura, escritura y ejecución.
S_IRGRP	0000040	group tiene permiso de lectura
S_IWGRP	0000020	group tiene permiso de escritura
S_IXGRP	0000010	group tiene permiso de ejecución

- **Para otros:**

S_IRWXO	0000007	other tiene permisos de lectura, escritura y ejecución.
S_IROTH	0000004	other tiene permiso de lectura
S_IWOTH	0000002	other tiene permiso de escritura
S_IXOTH	0000001	other tiene permiso de ejecución

Ejercicio 3. ¿Qué hace el siguiente programa?

```
/*tarea3.c
Trabajo con llamadas al sistema del Sistema de Archivos "POSIX 2.10 compliant"
*/
#include <unistd.h>  /* POSIX Standard: 2.10 Symbolic Constants <unistd.h> */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h> /* Primitive system data types for abstraction of implementation-dependent data
                          types. POSIX Standard: 2.6 Primitive System Data Types <sys/types.h>
                          */
#include <sys/stat.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

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

        int i;
        struct stat atributos;
        char tipoArchivo[30];

        if(argc<2)
                printf("\nSintaxis de ejecución: tarea3 [<nombre_archivo>]+\n\n");
                exit(EXIT_FAILURE);

        for(i = 1; i < argc; i++)
                printf("%s: ", argv[i]);

                if(lstat(argv[i], &atributos) < 0){
                        printf("\nError al intentar acceder a los atributos de %s",argv[i]);
                        perror("\nError en lstat");

                }else{
                        if(S_ISREG(atributos.st_mode)) strcpy(tipoArchivo,"Regular");

                        else if(S_ISDIR(atributos.st_mode)) strcpy(tipoArchivo,"Directorio");

                        else if(S_ISCHR(atributos.st_mode)) strcpy(tipoArchivo,"Especial de caracteres");

                        else if(S_ISBLK(atributos.st_mode)) strcpy(tipoArchivo,"Especial de bloques");

                        else if(S_ISFIFO(atributos.st_mode)) strcpy(tipoArchivo,"Tubería con nombre (FIFO)");

                        else if(S_ISLNK(atributos.st_mode)) strcpy(tipoArchivo,"Enlace relativo (soft)");

                        else if(S_ISSOCK(atributos.st_mode)) strcpy(tipoArchivo,"Socket");

                        else strcpy(tipoArchivo,"Tipo de archivo desconocido");

                        printf("%s\n",tipoArchivo);
                }
        }

        return EXIT_SUCCESS;
}
```

El programa determina el tipo de fichero de los archivos pasados como argumento. El programa hace lo siguiente:

- Comprueba que hay argumentos de entrada; si no los hay, sale del programa (primer if(argc < 2);
- Si hay argumentos, recorre todos los argumentos, que son nombres de ficheros, y para cada uno de ellos:
  - imprime el nombre;
  - comprueba si puede acceder a sus atributos;
  - comprueba el tipo de archivo con el flag que tiene activado (S_ISREG, S_ISDIR);
  - imprime el tipo de fichero.

3.5 Ejercicio 4

Ejercicio 4. Define una macro en lenguaje C que tenga la misma funcionalidad que la macro S_ISREG(mode) usando para ello los flags definidos en <sys/stat.h> para el campo st_mode de la struct stat, y comprueba que funciona en un programa simple. Consulta en un libro de C o en Internet cómo se especifica una macro con argumento en C.

#define S_ISREG2(mode) …

#define S_ISREG2(mode) (mode & S_IFMT == S_IFREG)

Para ver si un fichero es de tipo regular, tomamos S_IFMT, que es la máscara para el tipo de fichero, y le hacemos un AND con el modo recibido. Si el resultado es igual a S_IFREG, es que es regular; en caso contrario, es de otro tipo.

4. Preguntas de repaso

1. **¿Para qué sirve el flag O_WRONLY? ¿Cómo podemos descubrirlo?**

   Abre el fichero en modo sólo escritura. Podemos descubrirlo mirando el manual de la llamada open, **man 2 open**.

2. **¿Qué hace la siguiente orden del ejercicio 1? lseek(fd, 40, SEEK_SET);**

   Posiciona el puntero en la posición 40 (adelanta 40 bytes la posición del puntero), desde el comienzo del fichero (SEEK_SET).

   Si el fichero tuviera solo 10 bytes, ¿dónde posicionaría el current file offset?

   Si vemos en el ejercicio 1, escribe primero 10 bytes, desplaza el puntero 40 bytes para después escribir el buffer2. Los bytes desde la posición 10 a la 39, los rellena con 0 (podemos verlo viendo el fichero resultante en hexadecimal con *od -c archivo*).

3. **¿Para qué sirven todas estas opciones: O_CREAT | O_TRUNC | O_WRONLY | S_IRUSR | S_IWUSR?**

   Para crear un fichero (O_CREAT), si existe el archivo los sobreescribe para dejarlo con tamaño 0 (O_TRUNC), con permisos de sólo escritura (O_WRONLY) y permisos de lectura y escritura para el usuario (S_IRUSR, S_IWUSR).

   Estos flags podemos verlo en el manual man 2 open.

4. **¿Qué pasa si ponemos la siguiente línea en el código del ejercicio 1 y luego escribimos algo en el fichero?**

```
lseek (fd, 11, SEEK_SET);
//y luego …
write(fd, buf3, 1);
```

   Mueve el puntero al byte 11 y sobreescribe lo que hay en dicha posición 11 con el buf3 que tiene tamaño 1.

   El ejercicio 1 quedaría:

   - Desde la posición/byte 0 a la 9 escribe el buf1, con tamaño 10;
   - Desde la posición/byte 10 a la 39 escribe /0;
   - Desde la posición/byte 40 al 50 escribe buf2, con tamaño 10.

   Al añadir el buf3 en el byte 11, nos quedaría:

   - Desde la posición/byte 0 a la 9 escribe el buf1, con tamaño 10;
   - El byte 10 tiene valor /0;
   - El byte 11 contiene buf3 que es de tamaño 1;
   - Desde la posición/byte 12 a la 39 escribe /0;
   - Desde la posición/byte 40 al 50 escribe buf2, con tamaño 10.