Actividad 3.1 Trabajo con llamadas de gestión y procesamiento sobre archivos regulares

Consulta la llamada al sistema open en el manual en línea. Fíjate en el hecho de que puede usarse para abrir un archivo ya existente o para crear un nuevo archivo. En el caso de la creación de un nuevo archivo tienes que entender correctamente la relación entre la máscara umask y el campo mode, que permite establecer los permisos del archivo. El argumento mode especifica los permisos a emplear si se crea un nuevo archivo. Es modificado por la máscara umask del proceso de la forma habitual: los permisos del fichero creado son (modo & ~umask). Mira la llamada al sistema close en el manual en línea.

Mira la llamada al sistema lseek fijándote en las posibilidades de especificación del nuevo current_offset. Mira la llamada al sistema read fijándote en el número de bytes que devuelve a la hora de leer desde un archivo y los posibles casos límite. Mira la llamada al sistema write fijándote en que devuelve los bytes que ha escrito en el archivo.

Llamada open

Para consultar la llamada al sistema open hacemos **man 2 open**

```
OPEN(2)                    Linux Programmer's Manual                    OPEN(2)

NAME
       open, openat, creat - open and possibly create a file

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

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

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

       int openat(int dirfd, const char *pathname, int flags);
       int openat(int dirfd, const char *pathname, int flags, mode_t mode);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       openat():
           Since glibc 2.10:
               _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _ATFILE_SOURCE

DESCRIPTION
       The  open()  system  call opens the file specified by pathname.  If the
       specified file does not exist, it may optionally (if O_CREAT is  speci-
       fied in flags) be created by open().

       The  return  value of open() is a file descriptor, a small, nonnegative
       integer that is used in subsequent  system  calls  (read(2),  write(2),
       lseek(2), fcntl(2), etc.) to refer to the open file.  The file descrip-
       tor returned by a successful call will be the lowest-numbered file  de-
       scriptor not currently open for the process.

       By default, the new file descriptor is set to remain open across an ex-
       ecve(2) (i.e., the FD_CLOEXEC file descriptor  flag  described  in  fc-
       ntl(2) is initially disabled); the O_CLOEXEC flag, described below, can
       be used to change this default.  The file offset is set to  the  begin-
       ning of the file (see lseek(2)).

       A  call  to open() creates a new open file description, an entry in the
       system-wide table of open files.  The open file description records the
       file  offset  and the file status flags (see below).  A file descriptor
       is a reference to an open file description;  this  reference  is  unaf-
       fected  if  pathname  is subsequently removed or modified to refer to a
       different file.  For further details on  open  file  descriptions,  see
       NOTES.

       The  argument  flags  must  include  one of the following access modes:
       O_RDONLY, O_WRONLY, or O_RDWR.  These request opening  the  file  read-
       only, write-only, or read/write, respectively.

 Manual page open(2) line 1 (press h for help or q to quit)
```

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

  int open(char* nombre, int modo, int permisos);
  Como sale en la terminal → int open(const char *pathname, int flags, mode_t mode);

- nombre: es el nombre del archivo 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 sólo lectura
O_WRONLY	abre en modo sólo 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

- Este parámetro se debe utilizar cuando creemos un archivo, cuando incluyamos O_CREAT en el modo. Algunos permisos posibles son:

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 permiso 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.

  Esta llamada devuelve un descriptor válido de fichero; si no se ha podido abrir devuelve el valor -1.
- La llamada para crear un fichero es:

int creat(char* nombre, int modo);
Como sale en la terminal → int creat(const char *pathname, mode_t mode);

- 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	sólo lectura
O_WRONLY	sólo escritura
O_RDWR	lectura y escritura

Esta llamada devuelve un descriptor si el fichero se ha creado y abierto, y -1 si no ha sido así.

Esta llamada equivale a → open(nombre, O_RDWR | O_CREAT, permisos).

- umask es la máscara para aplicar los permisos a los archivos

Llamada close

Para ver la información de close hacemos man 2 close

int close(int fd);

- fd es el descriptor de archivo (file descriptor).

Esta llamada devuelve 0 si tiene éxito. SI hay un error, devuelve -1

Llamada al sistema 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);
como sale en la terminal → off_t lseek(int fd, off_t offset, int whence);

- 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.

El desplazamiento y el origen pueden tomar los siguientes 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 algún error.

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

Llamada read

Para esta llamada hacemos man 2 read

Como sale en la terminal → ssize_t read(int fd, void *buf, size_t count);

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

- fichero: es el descriptor de fichero del que se quiere leer.
- buffer: donde se almacena lo que se lee.
- bytes: es la cantidad de bytes que se quiere leer del archivo.

Esta llamada devuelve el número de bytes que realmente se han transferido. Esta llamada sirve para saber si hemos llegado al final del fichero o no. En caso de error, devuelve -1.

Llamada write

Como sale en la terminal → ssize_t write(int fd, const void *buf, size_t count);
int write(int fichero, const void* buffer, unsigned bytes);

- fichero: es el descriptor de fichero del que se quiere escribir.
- buffer: donde se almacena lo que se escribe.
- bytes: es la cantidad de bytes que se quiere escribir del archivo.

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''
Probad tras la ejecución del programa: $>cat archivo y $> od -c archivo
*/
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<errno.h>

char buf1[]="abcdefghij";
char buf2[]="ABCDEFGHIJ";
int main(int argc, char *argv[])
int fd;
if( (fd=open("archivo",O_CREAT|O_WRONLY,S_IRUSR|S_IWUSR))<0) {
        printf("\nError %d en open",errno);
        perror("\nError en open");
        exit(-1);
}
if(write(fd,buf1,10) != 10) {
        perror("\nError en primer write");
        exit(-1);
}
if(lseek(fd,40,SEEK_SET) < 0) {
        perror("\nError en lseek");
        exit(-1);
}
if(write(fd,buf2,10) != 10) {
        perror("\nError en segundo write");
        exit(-1);
}
close(fd);
return 0;
}
```

- Primer if:
  Crea un archivo de nombre “archivo”. En el campo de modo, que es la forma en la que se desea trabajar con el fichero, tenemos O_CREAT|O_WRONLY, que significa que crea el archivo y lo abre pero sólo en modo escritura. En el campo de permisos tenemos S_IRUSR|S_IWUSR, que significa que el usuario tiene permiso de lectura (S_IRUSR) y que el usuario tiene permiso de escritura (S_IWUSR).

  Si todas estas condiciones son menor que 0, es decir, si hay un fallo, el if imprime que hay un error.

- Segundo if: si hemos conseguido crear/abrir el fichero, con esta función se intentará escribir en él. Para escribir, a wirte 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. En este código, si no se han 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). Se coloca a 40 bytes ya que es lo que ha ocupado el primer bufer. 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: scribimos en la posición en la que se encuentra el puntero, el buf2 que es “ABCDEFGHIJ”. Si hay un error, se imprime por pantalla que ha habido un error.

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.

Vamos a ejecutar el archivo:

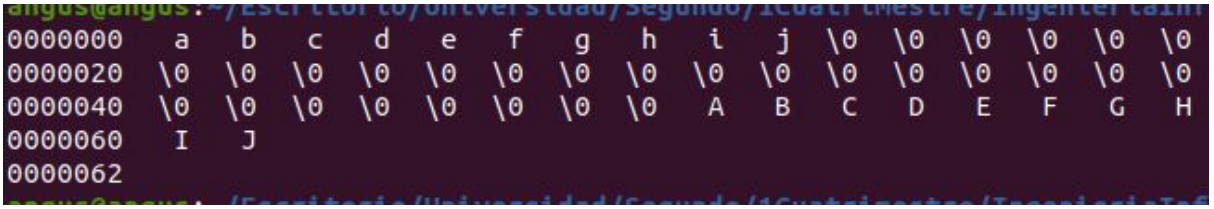
Para crear el archivo .o hacemos gcc tarea1.c -o tarea1. Posteriormente hacemos ./tarea1 y se supone que nos ha creado el archivo “archivo”.

Vamos a probar hacer cat archivo:

Al hacer cat archivo nos ha impreso abcdefghijABCDEFGHIJ en la terminal.

Vamos a hacer od -c archivo:

El comando Od (Octal Dump) de Linux, nos permite visualizar el contenido byte a byte de los archivos en código ASCII, octal, decimal y hexadecimal.
Nos imprime lo siguiente:



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>          /* 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>


int main(int argc, char *argv[])
{
int entrada,salida;
int bytes = 80;
char bloque[10];
char salto_linea[3];


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

if(argc==2){

//Abrimos el fichero para solo lectura
entrada = open(argv[1],O_RDONLY)

}

else{
entrada = STDIN_FILENO; //STDIN_FILENO es la entrada estándar, como cin
}

if(entrada<0){

printf("Error en la apertura del archivo de entrada");
exit(-1);

}

//Creamos el archivo salida

salida = open("salida.txt",O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR | S_IWUSR); //creamos el archivo con O_CREAT, si el fichero existe y es un fichero regular, y tiene permisos de escritura, trunca su longitud a 0. Con O_WRONLY le damos permisos de escritura. Con S_IRUSR y S_IWUSR, el usuario tiene permiso de lectura y escritura
```

```
35 else{
36 entrada = STDIN_FILENO; //STDIN_FILENO es la entrada estándar, como cin
37 }
38
39 if(entrada<0){
40
41 printf("Error en la apertura del archivo de entrada");
42 exit(-1);
43
44 }
45
46 //Creamos el archivo salida
47
48 salida = open("salida.txt",O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR | S_IWUSR); //creamos el archivo con O_CREAT, si el
   fichero existe y es un fichero regular, y tiene permisos de escritura, trunca su longitud a 0. Con O_WRONLY le damos
   permisos de escritura. Con S_IRUSR y S_IWUSR, el usuario tiene permiso de lectura y escritura
49
50 if(salida<0){
51
52         printf("Error en la apertura del archivo de salida");
53         exit(-2);
54 }
55
56 //Vamos leyendo hasta que lleguemos al final del fichero. La condición de parada del for es hasta que read devuelva 0, que
   será cuando llegue al final del fichero.
57 for(int i=0; read(entrada,buffer,bytes);i++){
58
59         //para escribir en el archivo Bloque numero cuando vamos a escribir un bloque
60         sprintf(bloque, "Bloque %d", i);
61
62         write(salida, bloque, sizeof(bloque));
63
64         //escribimos lo que hemos leído de entrada
65         write(salida, buffer, bytes);
66
67         //para escribir salto de línea
68         write(salida, "\n", sizeof("\n"));
69
```

```
gcc ejercicio3.1.2.c -o ejercicio3.1.2
./ejercicio3.1.2 archivo
```

Al hacer esto nos ha creado el archivo salida.txt

¿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.
Bloque 2
// Aquí van los siguientes 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"));
}

//al terminar de escribir, nos posicionamos al principio del fichero.
//lseek sirve como puntero para posicionarnos en los lugares que queramos
lseek(salida, 0, SEEK_SET);

//escribimos la línea con el número de bloques. Utilizamos un vector de char llamado
//bloque para escribir la frase para posteriormente poder pasarlo a la función write
sprintf(bloque, "El número de bloques es %d", bl);

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

Actividad 3.2. Trabajo con llamadas al sistema de la familia stat.

Consulta las llamadas al sistema stat y lstat para entender sus diferencias

- Llamada stat.

  Hacemos man 2 stat.

  int stat(const char *pathname, struct stat *statbuf);

  - pathname. Contiene la ruta del archivo que contiene los permisos.
  - statbuf. Buffer que apunta al archivo.

  Esta llamada recupera información sobre el archivo al que apunta la ruta.

- Llamada lstat.

  int lstat(const char *pathname, struct stat *statbuf);

  - pathname. Contiene la ruta del archivo que contiene los permisos.
  - statbuf. Buffer que apunta al archivo.

  lstat () es idéntico a stat (), excepto que si el nombre de ruta es un enlace simbólico, devuelve información sobre el enlace en sí, no el archivo al que se refiere.

3. ¿Qué hace el siguiente programa?

```
tarea2.c
Trabajo con llamadas al sistema del Sistema de Archivos ''POSIX 2.10 compliant''
*/
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.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 ejecucion: tarea2 [<nombre_archivo>]+\n\n");
exit(-1);
}
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,"Cauce 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 0;
}
```

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);

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 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.

S_ISREG devuelve verdadero si el archivo es regular.

Un macro es un documento que, a través de la elaboración de comandos predeterminados, facilitado a un usuario la realización de ciertos trabajos en un determinado programa.