Sistemas Operativos

Modulo II: Uso de los Servicios del SO mediante la API

Sesión 1. Llamadas al sistema para el Sistema de Archivos (Parte I)

Actividad 1

¿Qué hace el siguiente programa? Probad tras la ejecución del programa las siguientes órdenes del shell: **`cat archivo`** y $> **`od –c archivo`**

El programa tiene la funcionalidad de crear dos arrays: buf1 y buf2 que contienen 10 caracteres cada uno. Crea un entero fd al cual asigna el valor de la llamada al sistema open del archivo "archivo", el resultado del cual da error si es menor que cero.

→ **`O_CREAT`** se crea si no existe.
→ **`O_TRUNC`** si existe el fichero y tiene habilitada la escritura, lo sobreescribe a tamaño 0
→ **`O_WRONLY`** decimos que solo se permite escritura,
→ **`S_IRUSR`** comprobamos que el usuario tiene permiso de lectura,
→ **`S_IWUSR`** comprobamos que el usuario tiene permiso de escritura.

Después comprueba que si, después de hacer la orden **`write`** del primer búfer, el resultado asociado a esta operación es distinto de 10 se muestra error, dado que hemos escrito los 10 caracteres del primer búfer en el archivo la operación es correcta.

Después con lseek ponemos el puntero del archivo en la posición 40(en bytes) desde **`SEEK_SET`** (inicio del fichero) y da error si el resultado de la operación es menor que 0.

Con esto conseguimos que el puntero se sitúe justo después de los 40 bytes que hemos escrito previamente, al final de los 10 caracteres de buf1.

Por último llamamos a **`write`** para el segundo búfer igual que hemos hecho antes con el primero. Esto imprime los 10 caracteres de buf2.

`cat archivo`
`muestra el contenido del archivo`

`od –c archivo`
`muestra el contenido del archivo caracter a carácter incluyendo backslash espacios entre caracteres saltos de linea.`

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.

```
/*
actividad2.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>
#include<string.h>
//-----------------------------------------
char buf1[80];
char buf2[80];
char titulo[40];
char salto[2]="\n";
//-----------------------------------------
int main(int argc, char *argv[])
{
      int f_IN, f_OUT, leidos;
      char caracter[1];
      int contador = 0;
      int cont_caracteres = 0;
      if (argc == 2){
            //abre el fichero
            f_IN  =  open(argv[1],  O_RDONLY);//O_RDONLY  compueba  si  tiene
permiso de lectura

            if( f_IN <0) {
                  printf("\nError %d en open",errno);
                  perror("\nError en open archivo de entrada");
                  exit(EXIT_FAILURE);
            }

      }else{
                  //si no se facilita el fichero de entrada
                  f_IN =  STDIN_FILENO;
            }


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

```
	if( f_OUT<0) {
		printf("\nError %d en open",errno);
		perror("\nError en open archivo de salida");
		exit(EXIT_FAILURE);
	}

	//dejamos sitio para luego escribir el titulo (modificación)
	if(lseek(f_OUT,40,SEEK_SET) < 0) {
		perror("\nError en lseek");
		exit(EXIT_FAILURE);
	}

	//mientras se pueda leer
	while ( (leidos = read(f_IN,caracter,1)) != 0 ) {

		cont_caracteres++;

		if (cont_caracteres % 80 == 0 || cont_caracteres == 1)
		{
			contador++ ;
			write( f_OUT,salto,strlen(salto));
			sprintf(buf2,"Bloque %d\n",contador );
			write( f_OUT,buf2,strlen(buf2));
		}
		write( f_OUT,caracter,1); //escribe
	}

	// volvemos al inicio para escribir el titulo
	if(lseek(f_OUT,0,SEEK_SET) < 0) {
		perror("\nError en lseek");
		exit(EXIT_FAILURE);
	}
	//escribimos el titulo
	sprintf(titulo,"\nNumero de Bloques %d\n",contador);
	write( f_OUT,titulo,strlen(titulo) );
	//cerramos los ficheros
		close(f_IN);
		close(f_OUT);

	return EXIT_SUCCESS;
}
```

Ejercicio 3

→ Una vez que hemos compilado el ejercicio y lo hemos ejecutado pasándole como argumento <nombre_archivo> , lo que hace es decirnos que tipo de archivo es, ya sea un archivo regular, un directorio, un dispositivo de bloques, etc.

→ Internamente, el programa comprueba cada flag correspondiente a un archivo determinado con el archivo que hemos introducido como parámetro, y si se activa nos mostrará el tipo de archivo en el cual se ha activado.

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) ...

→ Tendría que definir la macro en el código como está definida S_ISREG(mode)
#define S_ISREG2(mode) (((mode) & S_IFMT) == S_IFREG)

→ También se podría dar los valores directamente.

#define S_ISREG2(mode) (((mode) & 00170000) == 0100000)

Luego podría llamarla igual que llamo a S_ISREG

Lo que hará la macro es que el compilador reemplazará S_ISREG2 por lo definido a la derecha.

```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<errno.h>
#include<string.h>
//-------------------------------------------------------
#define S_ISREG2(mode) (((mode) & S_IFMT) == S_IFREG)

#define S_ISREG3(mode) (mode & S_IFMT)
#define S_ISREG4(mode) S_ISREG(mode)

#define S_ISREG5(mode) (((mode) & 00170000) == 0100000)
//-------------------------------------------------------
int main(int argc, char *argv[])
{
int i;
struct stat atributos;
char tipoArchivo[30];

if(argc<2) {
      printf("\nSintaxis de ejecucion: actividad4 [<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");
            printf("\n\tS_ISREG %s\n",tipoArchivo);

            if(S_ISREG2(atributos.st_mode)) strcpy(tipoArchivo,"Regular");
            printf("\tS_ISREG2 %s\n",tipoArchivo);

            if(S_ISREG3(atributos.st_mode)) strcpy(tipoArchivo,"Regular");
            printf("\tS_ISREG3 %s\n",tipoArchivo);
```

```
            if(S_ISREG4(atributos.st_mode)) strcpy(tipoArchivo,"Regular");
            printf("\tS_ISREG4 %s\n",tipoArchivo);

            if(S_ISREG5(atributos.st_mode)) strcpy(tipoArchivo,"Regular");
                  printf("\tS_ISREG5 %s\n",tipoArchivo);
      }
}

return 0;

}
```

Sesión 2. Llamadas al sistema para el Sistema de Archivos (Parte II)

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>   //Primitive system data types for abstraction of
implementation-dependent data types.
                                    //POSIX Standard: 2.6 Primitive System Data
Types <sys/types.h>
#include<unistd.h>            //POSIX   Standard:   2.10   Symbolic   Constants
<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>       //Needed for open
#include<stdio.h>
#include<errno.h>


int main(int argc, char *argv[])
{
int fd1,fd2;
struct stat atributos;

//CREACION 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);
}

return 0;
}
```

Lo que hace el programa es:
→ crear un archivo llamado archivo1 con permisos , lectura y ejecución para ugo (0555)
→ Pone la máscara a 0 con la orden **umask(0)**
→ Después crea otro archivo llamado archivo2, con los mismos permisos que el archivo anterior.

(0555) **000 101 101 101** & ~(000 000 000 000)
000 101 101 101 & 111 111 111 111 = **000 101 101 101 (0555)**

→ Comprueba que se puede acceder a los atributos del primer archivo con la orden **stat**.
→ Cambia los permisos de archivo1 con **chmod** haciendo un AND lógico del estado del archivo(accediendo al struct atributos) con el negado del permiso de ejecución para el grupo. También se activa la asignación del GID del propietario al GID efectivo del proceso que ejecute el archivo.
→ Cambia los permisos del segundo archivo con **chmod** para que tenga todos los permisos para el propio usuario, permiso de lectura y escritura para el grupo, y lectura para el resto de usuarios.

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>

```
/*     ejercicio2.c   */
#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> // manejo de directorios
#include <string.h>

int main(int argc, char *argv[])
{
       //compueba el numero  de argumentos
       if (argc != 3){
              printf("Uso: %s  <directorio> <permiso> \n",argv[0]);
              exit(-1);
       }
       DIR* dir;
       struct dirent *ent;
       struct stat atributos;

       char cadena[100];
       char * pathname;
       pathname = argv[1];
       int nuevo_permiso;
       nuevo_permiso = strtol(argv[2],0,8);//cambia a octal

       dir = opendir(pathname);//opendir devuelve un puntero a la estructura de tipo DIR
       if (dir==NULL){
              perror("\nError en opendir");
              exit(EXIT_FAILURE);
       }
       //imprime la cabecera
       printf("\narchivo: \t\t\tpermisos_antiguos  \tpermisos_nuevos/errno\n\n");

       //lectura del directorio hasta que devuelva NULL(final o error)
       while( (ent=readdir(dir) ) != NULL)
       {
              sprintf(cadena,"%s/%s",pathname,ent->d_name);//concatena el pathname
(chmod)
              if(stat(cadena,&atributos)<0){//obtiene los permisos
                     perror("Error: stat");
                     exit(EXIT_FAILURE);
              }
              if(S_ISREG(atributos.st_mode)){ //si es un archivo regular
                     printf("%-35s %-25o ",cadena, atributos.st_mode);
                     if( (chmod(cadena, nuevo_permiso)) < 0){//cambia los permisos
                            printf("Error: %s \n",strerror(errno) );
                     }else{
                            lstat(cadena,&atributos);
                            printf(" %o \n", atributos.st_mode);//muestra los nuevos
permisos
                     }
              }
       }
       closedir(dir);
       return 0;
}
```

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
```

```c
/*  ejercicio3.c
* Recorre la jerarquía de directorios existentes a partir de uno dado como argumento.
* devuelve la cuenta de todos aquellos archivos regulares que tengan permiso de
ejecución
* para el grupo y para otros. Ademas del numero de archivos encontrados devuelve sus
números * de la suma total del espacio ocupado.
* uso: ./buscar <pathname>
*/
#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> // manejo de directorios
#include <string.h>

// macro condicion
// usuario   grupo  otros
// r w x    r w x   r w x
// 0 0 0    0 0 1   0 0 1
#define S_IXGRPUSR 011
#define condicion(mode) (((mode) & ~S_IFMT) & 011) == S_IXGRPUSR
//procedimiento para la búsqueda
void buscar(DIR *dir, char pathname[] , int *cont, int *tam);

int main(int argc, char *argv[]){
      char pathname[500];
      //compueba el numero  de argumentos
      if (argc == 2)
            strcp(pathname,argv[1]);
      else
            strcpy(pathname,"."); //empieza en el directorio actual

      DIR* dir;
      struct dirent *ent;
      struct stat atributos;
```

```
       int contador = 0,tamanio = 0;

       dir = opendir(pathname);//opendir devuelve un puntero a la estructura de tipo DIR
       if (dir==NULL){
              perror("\nError en opendir");
              exit(EXIT_FAILURE);
       }
       //imprime la cabecera
     printf("\nLos i-nodos son:\n\n");
     //realiza la busqueda
     buscar(dir, pathname,&contador,&tamanio);

     printf("\nExisten %d archivos regulares con permiso x para grupo y otros\n",
contador);
     printf("El tamaño total ocupado por dichos archivos es %d bytes\n", tamanio);
     return 0;
}/*fin main*/

void buscar(DIR *dir, char pathname[] , int *cont, int *tam){
       DIR* dir_ac;
       struct dirent *ent;
       struct stat atributos;
       char cadena[500];

       while( (ent=readdir(dir) ) != NULL){
              //ignorar . ,..
              if(strcmp(ent->d_name, ".")!=0 && strcmp(ent->d_name,"..")!=0){
              sprintf(cadena,"%s/%s",pathname,ent->d_name);//concatena el pathname
(chmod)

                     if(stat(cadena,&atributos)< 0){//obtiene los atributos
                            perror("\nError en lstat");
                            exit(EXIT_FAILURE);
                     }
                     if(S_ISDIR(atributos.st_mode)){//si es un directorio
                            if( (dir_ac=opendir(cadena))== NULL ){//control de error
                                   perror("\nError en opendir");
                                   exit(EXIT_FAILURE);
                            }else  buscar(dir_ac, cadena, cont, tam);//búsqueda recursiva

                     }else{
                            //-------------------------------------------------
                            printf("%-40s %ld\n",cadena, atributos.st_ino);
                            //-------------------------------------------------
                            if(S_ISREG(atributos.st_mode)){ //si es un archivo regular

                                   if( condicion(atributos.st_mode) ){//permiso x go
                                     (*cont)++;//incrementa el contador
                                     (*tam)+=(int)atributos.st_size;//acumula el tamaño
                                     }
                                 }
                     }
              }
       }
       closedir(dir);
}/*fin buscar*/
```

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

```
/* ejercicio4.c
 Recorre la jerarquia de directorios existentes a partir de uno dado como argumento.
 devuelve la cuenta de todos aquellos archivos regulares que tengan permiso
 de ejecucion para el grupo y para otros. Ademas del numero de archivos encontrados
 devuelve sus numeros de inodos y la suma total del espacio ocupado (nftw)
*/
```

```
#define _XOPEN_SOURCE 500
#include <ftw.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

// macro condicion
// usuario  grupo   otros
// r w x    r w x   r w x
// 0 0 0    0 0 1   0 0 1
#define S_IXGRPUSR 011
#define condicion(mode) (((mode) & ~S_IFMT) & 011) == S_IXGRPUSR

int tamanio=0;
int contador=0;
static int
buscar(const char *fpath, const struct stat *atributos,
             int tflag, struct FTW *ftwbuf)
{
        if(S_ISREG(atributos->st_mode))
            if( condicion(atributos->st_mode) ){
                contador++;
                tamanio+= (int)atributos->st_size;
            }

        printf("%-40s %7jd \n",fpath,(intmax_t) atributos->st_ino);

    return 0;
}

int
main(int argc, char *argv[])
{
    int flags = 0;
    char * pathname;
    pathname = argv[1];
    printf("\nLos i-nodos son:\n\n");
   if (nftw((argc < 2) ? "." : pathname, buscar, 20, flags) == -1) {
        perror("nftw");
        exit(EXIT_FAILURE);
    }

    printf("\nExisten %d archivos regulares con permiso x para grupo y otros\n",contador
);
    printf("El tamaño total ocupado por dichos archivos es %d bytes\n", tamanio);

    exit(EXIT_SUCCESS);
}
```

Sesión 3. Llamadas al sistema para el Control de Procesos

Actividad 1 Trabajo con la llamada al sistema fork

Ejercicio 1. Implementa un programa en C que tenga como argumento un número entero. Este programa debe crear un proceso hijo que se encargará de comprobar si dicho número es

un número par o impar e informará al usuario con un mensaje que se enviará por la salida estándar.

A su vez, el proceso padre comprobará si dicho número es divisible por 4, e informará si lo es o no usando igualmente la salida estándar.

```
/* ejercicio1.c
 Este programa debe crea un proceso hijo que se encarga de comprobar   si el numero pasado como argumento es un número par o impar e informa al usuario con un mensaje por la salida estándar. A su vez, el proceso padre comprueba si dicho número es divisible por 4, e informa si lo es o no usando la salida estándar.
*/
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
	pid_t pid;
	int estado, numero;
	if(argc!=2){
		printf("Uso: %s <numero> \n",argv[0] );
		exit(0);
	}

	numero =atoi(argv[1]);

	if( (pid=fork())<0) {
		perror("\nError en el fork");
		exit(-1);
	}
	else
		if(pid==0) {  //proceso hijo
			 //comprueba si es par
			if(numero%2 == 0) printf("Hijo: %d es par.\n",numero);
			else printf("Hijo: %d es impar.\n",numero);

		}else{// proceso padre
			wait(0);//espera al hijo
			//comprueba si es divisible por 4
			double res = numero%4;
			if( res == 0) printf("Padre: %d es divisible por 4.\n",numero);
			else printf("Padre: %d no es divisible por 4.\n",numero);
		}
	exit(0);
}
```

Ejercicio 2. ¿Qué hace el siguiente programa? Intenta entender lo que ocurre con las variables y sobre todo con los mensajes por pantalla cuando el núcleo tiene activado/desactivado el mecanismo de buffering.

Ejercicio 3. Indica qué tipo de jerarquías de procesos se generan mediante la ejecución de cada uno de los siguientes fragmentos de código. Comprueba tu solución implementando un código para generar 20 procesos en cada caso, en donde cada proceso imprima su PID y el del padre, PPID.

```
/* ejercicio3.c
 Indica qué tipo de jerarquías de procesos se generan mediante la ejecución de
cada
 uno de los siguientes fragmentos de código. Comprueba tu solución implementando
un código
 para generar 20 procesos en cada caso, en donde cada proceso imprima su PID y el
del padre,
 PPID.
*/

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

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

	pid_t id_proceso;
	pid_t id_padre;

	int childpid;
	int count = 2;//20

	/* Jerarquia de proceso tipo 1 */

	for (int i = 0; i < count; ++i){
		if ((childpid = fork()) ==-1){
		//fprintf(stderr, "Could not create child %d: %s\n",i,
strerror(errno ));
			exit(-1);
		}
		if (childpid)/* Jerarquia de proceso tipo 1 */
			break;
		else{
			id_proceso = getpid();	//pid del proceso que invoca
			id_padre = getppid();	//pid del proceso padre del proceso
que invoca

			printf("\n\ttipo1: PID: %-5d PPID: %-5d \ti:%-
5d\n",id_proceso,id_padre,i);
		}
	}
	wait(0);
	/* Jerarquia de proceso tipo 2 */
	for (int i = 0; i < count; ++i)  {
		if ((childpid = fork()) ==-1)		{
			//fprintf(stderr, "Could not create child %d:
%s\n",i,strerror(errno));
			exit(-1);
		}
		if (!childpid)
			break;
		else{
			id_proceso = getpid();	//pid del proceso que invoca
			id_padre = getppid();	//pid del proceso padre del proceso
que invoca

			printf("\ntipo2: PID: %-5d PPID: %-5d \ti:%-
5d\n",id_proceso,id_padre,i);
		}
	}

}
//ejecución con dos procesos para visualizar mejor
```

Actividad 2: Trabajo con las llamadas al sistema wait, waitpid y exit

Ejercicio 4. Implementa un programa que lance cinco procesos hijo. Cada uno de ellos se identificará en la salida estándar, mostrando un mensaje del tipo Soy el hijo PID. El proceso padre simplemente tendrá que esperar la finalización de todos sus hijos y cada vez que detecte la finalización de uno de sus hijos escribirá en la salida estándar un mensaje del tipo:

```
Acaba de finalizar mi hijo con <PID>

Sólo me quedan <NUM_HIJOS> hijos vivos
```

```
/* ejercicio4.c
 Programa que lanza cinco procesos hijo.
 Cada uno de ellos se dentifica en la salida estándar,  mostrando un mensaje del
tipo
              Soy el hijo PID.
 El proceso padre espera la finalización de todos sus hijos y cada vez que
detecta  la finalización de uno de sus hijos escribe en la salida estándar
 un mensaje del tipo:
             Acaba de finalizar mi hijo con <PID>
             Sólo me quedan <NUM_HIJOS> hijos vivos
*/
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

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

       pid_t  id_proceso,d_wait_;

       int childpid[5], estado, n_proc= 5, count = 5;

       for (int i = 0; i < n_proc; ++i) {
              if ((childpid[i] = fork()) ==-1){//creación
                     perror("\nError en el fork");
                     exit(-1);
              }
              if (!childpid[i]){/* hijo*/
                     id_proceso = getpid();     //pid del proceso que invoca

                     printf("Soy el hijo: PID: %-5d \n",id_proceso);
                     sleep(1);
                     exit(EXIT_SUCCESS);
              }
       }
       for (int i = 0; i < n_proc; ++i)
       {
              if(childpid[i]){ /*padre*/
                     do{
                            id_wait_ = waitpid(childpid[i],&estado,0);

                            if (id_wait_ == -1) {
                                  perror("waitpid");
                                  exit(EXIT_FAILURE);
                             } else  count--;

                     } while (!WIFEXITED(estado));//mientras el estado != exit

                     printf("\nAcaba de finalizar mi hijo con PID: %-
5d\n",childpid[i]);
                     printf("Sólo me quedan %d hijos vivos\n", count);
              }
       }
       return 0;
}
```

Ejercicio 5. Implementa una modificación sobre el anterior programa en la que el proceso padre espera primero a los hijos creados en orden impar (1º ,3º , 5º) y después a los hijos pares (2º y 4º)

```
/* ejercicio4.c
 Implementa un programa que lance cinco procesos hijo.
 Cada uno de ellos se     identificará en la salida estándar,
 mostrando un mensaje del tipo:
              Soy el hijo PID.
 El proceso padre simplemente tendrá que esperar
 la finalización de todos sus hijos y cada vez que  detecte
 la finalización de uno de sus hijos escribirá en la salida estándar
 un mensaje del tipo:
             Acaba de finalizar mi hijo con <PID>
             Sólo me quedan <NUM_HIJOS> hijos vivos
MODIFICACION:
 el proceso padre espera primero a los hijos creados en orden impar (1o,3o,5o) y
 después a los hijos pares (2o y 4o)

*/

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

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

       pid_t  id_proceso,
                     id_wait_;

       int childpid[5],
              estado,
              n_proc= 5,
              count = 5;

       for (int i = 0; i < n_proc; ++i)
       {
              if ((childpid[i] = fork()) ==-1){//creacion
                     perror("\nError en el fork");
                     exit(-1);
              }
              if (!childpid[i])// hijo
              {
                     id_proceso = getpid();     //pid del proceso que invoca
                     if((i+1)%2==0)      printf("Soy el hijo par \tPID: %-5d
\n",id_proceso);
                     else printf("Soy el hijo impar \tPID: %-5d \n",id_proceso);
                     sleep(1);
                     exit(EXIT_SUCCESS);
              }
       }

       for (int i = 0; i < n_proc; ++i)
       {
              if((i+1)%2!=0)

              if(childpid[i]){ //padre

                     do{
                            id_wait_ = waitpid(childpid[i],&estado,0);

                            if (id_wait_ == -1) {
                        perror("waitpid");
                        exit(EXIT_FAILURE);
                      } else count--;

             } while (!WIFEXITED(estado));//mientras el estado != exit

              printf("\nAcaba de finalizar mi hijo impar con PID: %-
```

```
5d\n",childpid[i]);
              printf("Sólo me quedan %d hijos vivos\n", count);
              }
       }
       for (int i = 0; i < n_proc; ++i)
       {
              if((i+1)%2==0)

              if(childpid[i]){ //padre

              do{
                     id_wait_ = waitpid(childpid[i],&estado,0);

                     if (id_wait_ == -1) {
                        perror("waitpid");
                        exit(EXIT_FAILURE);
                      } else        count--;

             } while (!WIFEXITED(estado));//mientras el estado != exit

              printf("\nAcaba de finalizar mi hijo par con PID:%-
5d\n",childpid[i]);
              printf("Sólo me quedan %d hijos vivos\n", count);

              }
       }
       return 0;
}
```

Actividad 3 Trabajo con la familia de llamadas al sistema exec

Ejercicio 6. ¿Qué hace el siguiente programa?

```
```

Ejercicio 7. Escribe un programa que acepte como argumentos el nombre de un programa, sus argumentos si los tiene, y opcionalmente la cadena “bg”. Nuesto programa deberá ejecutar el programa pasado como primer argumento en foreground si no se especifica la cadena “bg” y en background en caso contrario. Si el programa tiene argumentos hay que ejecutarlo con éstos.

```
/* ejercicio 7.
Escribe un programa que acepte como argumentos el nombre de un programa,
sus argumentos si los tiene, y opcionalmente la cadena “bg”.
Nuesto programa deberá    ejecutar el programa pasado como primer argumento en
foreground
si no se especifica la    cadena “bg” y en background en caso contrario.
Si el programa tiene argumentos hay que ejecutarlo con éstos.
*/

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

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

       int arg;
       bool backg;
       char paramts[100];
       char *prog;
       if(argc < 2){
```

```
		printf("Modo de uso: %s <programa> [opciones] [bg]\n\n", argv[0]);
		exit(EXIT_SUCCESS);
	}

	arg = argc;// control de numero de argumentos
	prog = argv[1];// obtenemos la prog a ejecutar
	backg = strcmp(argv[argc-1],"bg") ? false : true , arg-- ; //compara y
decrementa

	int i;
	for (i = 2; i < arg; i++)
	{// obtenemos los parametros
		strcat(paramts, argv[i]);
		strcat(paramts, " ");
	}
	sleep(1);
	int childpid;

		if (backg)//en background
		{	/*creacion  un hijo*/
			if ((childpid = fork()) ==-1){
				perror("\nError en el fork");
				exit(EXIT_FAILURE);
			}

			if(!childpid){ // el hijo ejecuta
				if( (execl(prog,paramts,paramts, NULL)<0) ) {
					perror("\nError en el excl");
					exit(EXIT_FAILURE);
				}
			}else{//el padre realiza exit

				exit(EXIT_SUCCESS);
			}
		}else{//no background
				if( (execl(prog,paramts,paramts, NULL)<0) ) {
					perror("\nError en el excl");
					exit(EXIT_FAILURE);
				}
			}
	return 0;
}
```

Sesión 4. Comunicación entre procesos utilizando cauces

Actividad 1 Trabajo con cauces con nombre

Ejercicio 1. Consulte en el manual las llamadas al sistema para la creación de archivos especiales en general (mknod) y la específica para archivos FIFO (mkfifo). Pruebe a ejecutar el siguiente código correspondiente a dos programas que modelan el problema del productor/consumidor, los cuales utilizan como mecanismo de comunicación un cauce FIFO.
Determine en qué orden y manera se han de ejecutar los dos programas para su correcto funcionamiento y cómo queda reflejado en el sistema que estamos utilizando un cauce FIFO.

Justifique la respuesta

```
Para que el funcionamiento sea correcto primero tenemos que ejecutar el
programa consumidor. Una vez iniciado el proceso consumidor ya podemos ejecutar
el productor pasándole como parámetro la cadena de caracteres a escribir. Como
el consumidor
```

está en un bucle infinito esperando a la entrada estándar, imprimirá el mensaje recibido inmediatamente, hasta recibir la cadena “fin”.

```
mar@marlen:-/Escritorio/Sesion4/src$ gcc productorFIFO.c -o productorFIFO
mar@marlen:-/Escritorio/Sesion4/src$ gcc consumidorFIFO.c -o consumidorFIFO
mar@marlen:-/Escritorio/Sesion4/src$ ./bin/consumidorFIFO

Mensaje recibido: primer mensaje

Mensaje recibido: 2do mensaje

Mensaje recibido: texto: dfjsssssssssddddddddddddd
mar@marlen:-/Escritorio/Sesion4/src$
```

Magentix_Init

```
mar@marlen:-/Escritorio/Sesion4/src$ ./productorFIFO "primer mensaje"
mar@marlen:-/Escritorio/Sesion4/src$ ./productorFIFO "2do mensaje"
mar@marlen:-/Escritorio/Sesion4/src$ ./productorFIFO "texto: dfjsssssssssddddddddddddd"
mar@marlen:-/Escritorio/Sesion4/src$ ./productorFIFO "fin"
mar@marlen:-/Escritorio/Sesion4/src$
```

Actividad 4.2 Trabajo con cauces sin nombre

Ejercicio 2. Consulte en el manual en línea la llamada al sistema pipe para la creación de cauces sin nombre. Pruebe a ejecutar el siguiente programa que utiliza un cauce sin nombre y describa la función que realiza (tarea6.c). Justifique la respuesta.

Ejercicio 3. Redirigiendo las entradas y salidas estándares de los procesos a los cauces podemos escribir un programa en lenguaje C que permita comunicar órdenes existentes sin necesidad de reprogramarlas, tal como hace el shell (por ejemplo ls | sort). En particular, ejecute el siguiente programa que ilustra la comunicación entre proceso padre e hijo a través de un cauce sin nombre redirigiendo la entrada estándar y la salida estándar del padre y el hijo respectivamente.

Ejercicio 4. Compare el siguiente programa con el anterior y ejecútelo. Describa la principal diferencia, si existe, tanto en su código como en el resultado de la ejecución.

Ejercicio 5

```
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
//---------------------------------------------------------------------
// Función que crea un proceso con fork() y este proceso creado realiza un execlp
// cuyo resultado es el calulo de los nº primos del intervalo pasado como parámetros.
//---------------------------------------------------------------------
void encargar_calculo(int descriptor[], char ini_inter_1[], char fin_inter_1[], pid_t
ESCLAVO);
int main(int argc,char *argv[]){
```

```
	if(argc<3){
		printf("Uso: %s [inicio intervalo] [fin intervalo]\n",argv[0]);
		return(0);
	}
	//cálculo del intervalo que cada uno de los hijos va ha calcular
	int inicio, fin, mitad;
	inicio = atoi(argv[1]);
	fin    = atoi(argv[2]);
	mitad  = ( fin - inicio )/2;
	char ini_inter_1[10], fin_inter_1[10],
	       ini_inter_2[10], fin_inter_2[10];

	//1er intervalo
	sprintf(ini_inter_1, "%d", inicio);
	sprintf(fin_inter_1, "%d", inicio+mitad);

	//2do intervalo
	sprintf(ini_inter_2, "%d", inicio+(mitad+1));
	sprintf(fin_inter_2, "%d", fin);

	//descriptor
	int descriptor[2];

	//vrble para identificar a los hijos;
	pid_t  ESCLAVO_1, ESCLAVO_2;
	//llamada al sistema para crear un pipe
	pipe(descriptor);
	//llamada a la funcion que crea un hijo y le encarga el calculo
	encargar_calculo(descriptor, ini_inter_1, fin_inter_1,ESCLAVO_1);

	//llamada al sistema para crear pipe
	pipe(descriptor);

	//llamada a la funcion que crea un hijo y le encarga el calculo
	encargar_calculo(descriptor, ini_inter_2, fin_inter_2,ESCLAVO_2);
      return(0);
}
//=========================================================================
void encargar_calculo(int descriptor[], char ini_inter_1[], char fin_inter_1[], pid_t
ESCLAVO){
	//vrble para guardar los valores leidos del cauce
	int valor,
	    i=0,j=0;//vrbles extras (printf)
	//crea el hijo (esclavo) para que calcule los números primos
	//del intervalo pasado como parámetros
	if((ESCLAVO = fork()) <0){
		perror("Error en la creación del 1er esclavo (fork)");
		exit(EXIT_FAILURE);
	}
	//encargar el cálculo de los primos del intervalo
	if(ESCLAVO == 0){
		printf("\nSoy el esclavo con PID <%d> y mi padre con PID <%d>\n",getpid(),
getppid() );
		printf("Intervalo [%s-%s]\n\n",ini_inter_1,fin_inter_1 );

		close(descriptor[0]);//cierra el descriptor usado para lectura
		dup2(descriptor[1], STDOUT_FILENO); // duplica

		if((execlp("./bin/esclavo","esclavo", ini_inter_1, fin_inter_1, NULL)) < 0)
{
			perror("Error en execpl ");
			exit(EXIT_FAILURE);
		}
	}else {// el padre lee los valores depositados
		close(descriptor[1]);//cierra el descriptor de escritura

		while(read(descriptor[0],&valor,sizeof(int))>0){ //usa el descriptor de
lectura
			printf("%-9d",valor );
			i++;if(i==10){printf("\n"); i=0;}//salto de linea cada 10 valores
leidos

		}//fin while
		close(descriptor[0]);//cierra el descriptor usado para lectura
		printf("\n");
	}
```

```
}
```

```
marlen@mar:/mnt/hgfs/C_VMWARE/SO/mod_II/Sesion4/src/actividades$ gcc esclavo.c -o esclavo -lm
marlen@mar:/mnt/hgfs/C_VMWARE/SO/mod_II/Sesion4/src/actividades$ gcc maestro.c -o maestro
marlen@mar:/mnt/hgfs/C_VMWARE/SO/mod_II/Sesion4/src/actividades$ ./maestro 1000 1500

Soy el esclavo con PID <8014> y mi padre con PID <8013>
Intervalo [1000-1250]

1009    1013    1019    1021    1031    1033    1039    1049    1051    1061
1063    1069    1087    1091    1093    1097    1103    1109    1117    1123
1129    1151    1153    1163    1171    1181    1187    1193    1201    1213
1217    1223    1229    1231    1237    1249

Soy el esclavo con PID <8015> y mi padre con PID <8013>
Intervalo [1251-1500]

1259    1277    1279    1283    1289    1291    1297    1301    1303    1307
1319    1321    1327    1361    1367    1373    1381    1399    1409    1423
1427    1429    1433    1439    1447    1451    1453    1459    1471    1481
1483    1487    1489    1493    1499
```

Sesión 5. Llamadas al sistema para gestión y control de señales

Actividad 1. Trabajo con las llamadas al sistema sigaction y kill.

A continuación se muestra el código fuente de dos programas. El programa envioSignal permite el envío de una señal a un proceso identificado por medio de su PID. El programa reciboSignal se ejecuta en background y permite la recepción de señales.

Ejercicio 1. Compila y ejecuta los siguientes programas y trata de entender su funcionamiento.

```
```

Ejercicio 2. Escribe un programa en C llamado contador, tal que cada vez que reciba una señal que se pueda manejar, muestre por pantalla la señal y el número de veces que se ha recibido ese tipo de señal, y un mensaje inicial indicando las señales que no puede manejar. En el cuadro siguiente se muestra un ejemplo de ejecución del programa.

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

const static int TAM = 31;
static int contador[31];

static void contador_handler(int nSignal){
   contador[nSignal]++;
   printf("\nLa señal %d se ha realizado %d veces.\n", nSignal, contador[nSignal]);
}
```

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

    struct sigaction sig_USR;

    if(setvbuf(stdout,NULL,_IONBF,0))
        perror("\nError en setvbuf");

    //Inicializar la estructura sig_USR
    sig_USR.sa_handler = contador_handler;

   //inicializamos el conjunto de señales
   sigemptyset(&sig_USR.sa_mask);

    sig_USR.sa_flags = 0;

    int i;
    for(i=1 ;i < TAM ;i++)
        contador[i]=0;
    for(i=1; i<TAM; i++){
        if(sigaction(i, &sig_USR, NULL) < 0){
            printf("\nNo se puede manejar la señal  %d\n",i);
        }
    }
    printf("\nEsperando el envio de señales...\n");
    while(1);//no termina el programa
}
```

```
marlen@mar:~/AA_SO/SO/mod_II/Sesion5/src/actividades$ gcc contador.c -o bin/contador
marlen@mar:~/AA_SO/SO/mod_II/Sesion5/src/actividades$ ./bin/contador &
[1] 5562
marlen@mar:~/AA_SO/SO/mod_II/Sesion5/src/actividades$
No se puede manejar la señal  9

No se puede manejar la señal  19

Esperando el envio de señales...
kill -7 5562
marlen@mar:~/AA_SO/SO/mod_II/Sesion5/src/actividades$
La señal 7 se ha realizado 1 veces.
kill -7 5562
marlen@mar:~/AA_SO/SO/mod_II/Sesion5/src/actividades$
La señal 7 se ha realizado 2 veces.
```

Actividad 5.2. Trabajo con las llamadas al sistema sigsuspend y sigprocmask

Ejercicio 3. Escribe un programa que suspenda la ejecución del proceso actual hasta que se reciba la señal SIGUSR1. Consulta en el manual en línea sigemptyset para conocer las distintas operaciones que permiten configurar el conjunto de señales de un proceso.

```
// ejercicio3.c
#include <stdio.h>
#include <signal.h>
int main(){
       sigset_t new_mask;
       /* inicializar la nueva mascara de señales */
       if(sigemptyset(&new_mask) <0  )  {
              perror(" ERROR: sigemptyset");
              return 0;
       }
       if(sigfillset(&new_mask) < 0)//inicializa un conjunto con todas las señales
       {
              perror("ERROR: sigfillset");
              return 0;
       }
```

```
        if (sigdelset(&new_mask , SIGUSR1) < 0)//elimina la señal SIGUSR1
         {
               perror("ERROR: sigdelset");
               return 0;
         }
        // esperar a cualquier señal menos a SIGUSR1
        if (sigsuspend(&new_mask) < 0)   {
               perror("ERROR: sigsuspend");
               return 0;
        }
}
```

```
marlen@mar:~/AA_SO/SO/mod_II/Sesion5/src/actividades$ ./bin/ejercicio3 &
[1] 6456
marlen@mar:~/AA_SO/SO/mod_II/Sesion5/src/actividades$ kill -2 6456
marlen@mar:~/AA_SO/SO/mod_II/Sesion5/src/actividades$ kill -7 6456
marlen@mar:~/AA_SO/SO/mod_II/Sesion5/src/actividades$ kill -10 6456
[1]+  Señal definida por el usuario 1        ./bin/ejercicio3
marlen@mar:~/AA_SO/SO/mod_II/Sesion5/src/actividades$ kill -7 6456
bash: kill: (6456) - No existe el proceso
```

Ejercicio 4. Compila y ejecuta el siguiente programa y trata de entender su funcionamiento.

```
Crea un manejador de señales, inicializa todas las señales del manejador a 0, luego crea un conjunto de mascaras vacía a la que agrega SIGTERM, luego bloqueamos la señal, hacemos una pausa (sleep) de 10, restauramos la señal y esperamos recibir las señales que estaban en espera en el manejador cambiando el valor de signal_recibida a 1.
```

Sesión 6. Control de archivos y archivos proyectados en memoria

Actividad 6.1 Trabajo con la llamada al sistema fcntl

Ejercicio 1. Implementa un programa que admita t argumentos. El primer argumento será una orden de Linux; el segundo, uno de los siguientes caracteres “<” o “>”, y el tercero el nombre de un archivo (que puede existir o no). El programa ejecutará la orden que se especifica como argumento primero e implementará la redirección especificada por el segundo argumento hacia el archivo indicado en el tercer argumento. Por ejemplo, si deseamos redireccionar la entrada estándar de sort desde un archivo temporal, ejecutaríamos:

```
$> ./mi_programa sort “<” temporal
```

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

```
#include <string.h>//strcmp
#include <errno.h>
int main(int argc, char *argv[]) {
	//Comprobamos si se le ha pasado un pathname y unos permisos como parámetros
	if(argc != 4)
	{
		//Si no se le han pasado los parámetros correctos, mensaje de ayuda
		printf("\nModo de uso: %s <comando> <simbolo> <archivo>\n", argv[0]);
		printf("Simbolos: \"<\" o \">\" debe pasarse entre comillas.\n\n");
		exit(EXIT_FAILURE);
	}
		//Declaracion de variables
		char *str_command;
		char *str_file;
		int fd, rdir;

		//Extraemos el comando
		str_command = argv[1];
		//Extraemos el fichero
		str_file = argv[3];
		//Comprobamos el segundo parametro, tiene que ser < o >
		char *str_simbolo = argv[2];

		if (strcmp(str_simbolo, "<") == 0) {
			//abre el fichero para lectura
			fd = open (str_file, O_RDONLY);
			if(fd <0){
				perror("open falló");
				exit(EXIT_FAILURE);
			}
			rdir = STDIN_FILENO;
			//cierra el descriptor de entrada
			close(STDIN_FILENO);

		} else if (strcmp(str_simbolo, ">") == 0) {
			//abre el fichero para escritura, si no existe lo crea
			fd = open (str_file, O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR);
			if(fd <0){
				perror("open falló");
				exit(EXIT_FAILURE);
			}
			// guardamos el tipo de redirecionamiento
			rdir = STDOUT_FILENO;
			//cierra  el descriptor de salida
			close (STDOUT_FILENO);

		} else  {
		//Si no se le han pasado el parametro correcto, mensaje de ayuda
		    printf("Modo de uso: %s [comando] [opciones: \"<\" o \">\" ]
[archivo]\n\n", argv[0]);
		  exit(EXIT_FAILURE);
		}

		//duplicamos fd, el descriptor duplicado depende del valor de rdir
		if (fcntl(fd,F_DUPFD, rdir)==-1) {
			perror ("fcntl falló");
			exit(EXIT_FAILURE);
		}
		//Ejecutamos el comando
	    if( (execlp(str_command, "", NULL) < 0)) {
			perror("Error en el execlp\n");
			exit(EXIT_FAILURE);
		}
		//Cerramos el fichero
		close(fd);
		return 0;
}
```

```
marlen@mar:~/AA_SO/SO/mod_II/Sesion6/src/actividades$ gcc -o ejercicio1 ejercicio1.c
marlen@mar:~/AA_SO/SO/mod_II/Sesion6/src/actividades$ ./ejercicio1 sort ">" sort.txt
JUAN
LUIS
MARIA
LAURA
marlen@mar:~/AA_SO/SO/mod_II/Sesion6/src/actividades$ ./ejercicio1 sort "<" sort.txt
JUAN
LAURA
LUIS
MARIA
```

Nota. El carácter redirección (<) aparece entre comillas dobles para que no los interprete el shell sino que sea aceptado como un argumento del programa mi_programa.

Ejercicio 2. Reescribir el programa que implemente un encauzamiento de dos órdenes pero utilizando fcntl. Este programa admitirá tres argumentos. El primer argumento y el tercero serán dos órdenes de Linux. El segundo argumento será el carácter "|". El programa deberá ahora hacer la redirección de la salida de la orden indicada por el primer argumento hacia el cauce, y redireccionar la entrada estándar de la segunda orden desde el cauce. Por ejemplo, para simular el encauzamiento ls|sort, ejecutaríamos nuestro programa como:

```
$> ./mi_programa2 ls "|" sort
```

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>//strcmp
#include <errno.h>
int main(int argc, char *argv[]) {
	//Comprobamos si se le ha pasado un pathname y unos permisos como parámetros
	if(argc != 4)
	{
		//Si no se le han pasado los parámetros correctos, mensaje de ayuda
		printf("\nModo de uso: %s <comando> <simbolo> <archivo>\n", argv[0]);
		printf("Simbolos: \"<\" o \">\" debe pasarse entre comillas.\n\n");
		exit(EXIT_FAILURE);
	}



		//Declaracion de variables
		char *str_command;
		char *str_file;
		int fd, rdir;

		//Extraemos el comando
		str_command = argv[1];
		//Extraemos el fichero
		str_file = argv[3];
		//Comprobamos el segundo parametro, tiene que ser < o >
		char *str_simbolo = argv[2];

		if (strcmp(str_simbolo, "<") == 0) {
			//abre el fichero para lectura
			fd = open (str_file, O_RDONLY);
			if(fd <0){
				perror("open falló");
				exit(EXIT_FAILURE);
			}
			rdir = STDIN_FILENO;
			//cierra el descriptor de entrada
			close(STDIN_FILENO);

		} else if (strcmp(str_simbolo, ">") == 0) {
```

```
                    //abre el fichero para escritura, si no existe lo crea
                    fd = open (str_file, O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR);
                    if(fd <0){
                           perror("open falló");
                           exit(EXIT_FAILURE);
                    }
                    // guardamos el tipo de redirecionamiento
                    rdir = STDOUT_FILENO;
                    //cierra  el descriptor de salida
                    close (STDOUT_FILENO);

             } else  {
             //Si no se le han pasado el parametro correcto, mensaje de ayuda
                 printf("Modo  de  uso:  %s  [comando]  [opciones:  \"<\"  o  \">\"  ]
[archivo]\n\n", argv[0]);
               exit(EXIT_FAILURE);
             }

             //duplicamos fd, el descriptor duplicado depende del valor de rdir
             if (fcntl(fd,F_DUPFD, rdir)==-1) {
                    perror ("fcntl falló");
                    exit(EXIT_FAILURE);
             }
             //Ejecutamos el comando
          if( (execlp(str_command, "", NULL) < 0)) {
                    perror("Error en el execlp\n");
                    exit(EXIT_FAILURE);
             }
             //Cerramos el fichero
             close(fd);
             return 0;
}
```

```
marlen@mar:/mnt/hgfs/C_VMWARE/SO/mod_II/Sesion6/src/actividades$ gcc -o ejercicio2 ejercicio2.c
marlen@mar:/mnt/hgfs/C_VMWARE/SO/mod_II/Sesion6/src/actividades$ ./ejercicio2 ls "|" sort
ayuda.txt
ejercicio1
ejercicio1.c
ejercicio1.sublime-workspace
ejercicio2
ejercicio2.c
ejercicio3.c
sort.txt
```

Actividad 6.2: Bloqueo de archivos con la llamada al sistema fcntl

Ejercicio 3. Construir un programa que verifique que, efectivamente, el kernel comprueba que puede darse una situación de interbloqueo en el bloqueo de archivos.

```
#include <unistd.h>//sleep
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
int main(int argc, char *argv[]) {
       //Declaracion de variables
       struct flock cerrojo;
       int fd, i;

       if(argc != 2) {
              //Si no se le han pasado los parámetros correctos, mensaje de ayuda
              printf("Modo de uso: %s <archivo>\n\n", argv[0]);
              exit(1);
       } else {
              if(setvbuf(stdout,NULL,_IONBF,0)) {
                     perror("\nError en setvbuf");
              }
              //Extraemos el nombre del archivo a usar (por comodidad)
              char *str_file = argv[1];
              //Abrimos el archivo
              if ((fd = open(str_file, O_RDWR)) == -1 ){
                     perror("Fallo al abrir el archivo");
```

```
                        return 0;
                }
                //creación de la estructura del cerrojo
                cerrojo.l_type      = F_WRLCK;//cerrojo para escritura
                cerrojo.l_whence= SEEK_SET;//desde el origen del archivo
                //Bloquearemos el archivo entero
                cerrojo.l_start     = 0;//dituandose en le byte 0
                cerrojo.l_len= 0; //bloquea hasta el final del archivo

                //Intentamos un bloqueo de escritura del archivo
                printf ("Intentando bloquear %s\n", str_file);

                //Si el cerrojo falla, mostramos un mensaje
                if (fcntl (fd, F_SETLKW, &cerrojo) == -1){
                        printf(" error en el bloqueo ");
                        if(errno == EDEADLKW) printf(" ha dado un EDEADLK\n");

                } //Mientras el bloqueo no tenga exito

                        //Ahora el bloqueo tiene exito y podemos procesar el archivo
                printf ("Procesando el archivo %s\n", str_file);
                //Hacemos un bucle con sleep para que de tiempo de lanzar otra vez el
programa
                sleep(20);
                //Una vez finalizado el trabajo, desbloqueamos el archivo
                cerrojo.l_type      = F_UNLCK;
                cerrojo.l_whence= SEEK_SET;
                cerrojo.l_start     = 0;
                cerrojo.l_len= 0;

                if (fcntl (fd, F_SETLK, &cerrojo) == -1) {
                        perror ("Error al desbloquear el archivo");
                }
                return 0;
        }
}
```

Ejercicio 4. Construir un programa que se asegure que solo hay una instancia de él en ejecución en un momento dado. El programa, una vez que ha establecido el mecanismo para asegurar que solo una instancia se ejecuta, entrará en un bucle infinito que nos permitirá comprobar que no podemos lanzar más ejecuciones del mismo. En la construcción del mismo, deberemos asegurarnos de que el archivo a bloquear no contiene inicialmente nada escrito en una ejecución anterior que pudo quedar por una caída del sistema.

```
```

Actividad 6.3 Trabajo con archivos proyectados

Ejercicio 5: Escribir un programa, similar a la orden cp, que utilice para su implementación la llamada al sistema mmap() y una función de C que nos permite copiar memoria, como por ejemplo memcpy(). Para conocer el tamaño del archivo origen podemos utilizar stat() y para establecer el tamaño del archivo destino se puede usar ftruncate().

```
```