Sesión 1. Llamadas al sistema para el SA (Parte I)

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

```
/*
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;

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

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

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

```
}

return EXIT_SUCCESS;
}
```

Lo que hace el siguiente programa es 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. Con O_CREAT se crea si no existe, con O_TRUNC si existe el fichero y tiene habilitada la escritura, lo sobreescribe a tamaño 0, con O_WRONLY decimos que solo se permite escritura, con S_IRUSR comprobamos que el usuario tiene permiso de lectura, por último con 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.
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.

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.
Guía Práctica de Sistemas Operativos-78Bloque 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.
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.
Bloque 2
// Aquí van los siguientes 80 Bytes del archivo pasado como argumento.

Ejercicio1-2.c

```
#include<sys/types.h>
```

```
#include<sys/stat.h>
#include<fcntl.h>
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<unistd.h>
int main(int argc, char *argv[]){
   int cont=1,leidos;
   int filein,fileout;
   char cadena[30];
   char cad_bloque[40];
   char salto_linea[2]="\n";
   char caracter[1];
   int num_char = 1;
   if (argc==2){
      filein=open(argv[1], O_RDONLY);
   }
   else{
      filein=STDIN_FILENO;
   }
   fileout=open("archivo_salida", O_CREAT|O_TRUNC|O_WRONLY,S_IRUSR|S_IWUSR);

   if (fileout < 0){
      printf("El fichero de salida no se pudo abrir correctamente\n");
      exit(-1);
   }
   while((leidos=read(filein,caracter,1))!=0){
      if (num_char == 1 || num_char%80 == 0){
         if (num_char != 1)
            write(fileout, salto_linea, strlen(salto_linea));
         else{
            sprintf(cad_bloque, "El numero de bloques es <%d>\n",cont);
            write(fileout, cad_bloque, strlen(cad_bloque));
         }

         sprintf(cad_bloque, "%s%d\n", "Bloque", cont);
         write(fileout, cad_bloque, strlen(cad_bloque));
         cont++;
      }
      write(fileout, caracter, 1);
      num_char++;
   }
   sprintf(cad_bloque, "El numero de bloques es <%d>\n", cont);
   lseek(fileout,0,SEEK_SET);
   write(fileout, cad_bloque, strlen(cad_bloque));
   close(filein);
   close(fileout);
   return 0;
```

```
}
```

Ejercicio 3.
¿Qué hace el siguiente programa?
Tarea2.c

```
/*
tarea2.c
Trabajo con llamadas al sistema del Sistema de Archivos "POSIX 2.10 compliant"
*/

#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<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,"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 0;
}
```

- 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) (mode & S_IFMT == S_IFREG)
```

Sesión 2. Llamadas al sistema para el SA (Parte II)

Ejercicio 1.
¿Qué hace el siguiente programa?
Tarea3.c

```
/*
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 de lectura, escritura y ejecución para el grupo, seguidamente pone la máscara a 0 con la orden umask(0) y después crea otro archivo llamado archivo2, con los mismos permisos que el archivo anterior.
Luego comprueba que se puede acceder a los atributos del primer archivo con la orden stat.
Después con chmod cambiamos los permisos del primer archivo haciendo un AND lógico del estado del archivo(accediendo al struct atributos) con el negado del permiso de ejecución para el grupo, con lo que le quitamos el permiso de ejecución para el grupo. También activamos la asignación del GID del propietario al GID efectivo del proceso que ejecute el archivo.
Por último con chmod cambiamos los permisos del segundo archivo 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>

Ejercicio2-2.c

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

int main(int argc, char *argv[]){
   DIR *direct;
   unsigned int permisos;
   char *pathname;
   struct stat atributos;
   struct dirent *ed;
   char cadena[100];
   char cadena2[100];
   extern int errno;

   if (argc==3){
      pathname=argv[1];
      direct=opendir(pathname);
      permisos=strtol(argv[2],NULL,8);
   }
   else{
```

```
        printf("Uso: ejercicio2.c <pathname> <permisos>\n");
        exit(-1);
    }
    readdir(direct);
    while((ed=readdir(direct))!=NULL){
        sprintf(cadena,"%s/%s",pathname,ed->d_name);
        if(stat(cadena,&atributos) < 0) {
            printf("\nError al intentar acceder a los atributos de archivo");
            perror("\nError en lstat");
            exit(-1);
        }
        if(S_ISREG(atributos.st_mode)){
            sprintf(cadena2,"%s",ed->d_name);
            printf("%s: %o ",cadena2,atributos.st_mode);
            chmod(cadena,permisos);
            if(chmod(cadena,permisos) < 0) {
                printf("Error: %s\n",strerror(errno));
            }
            else{
                stat(cadena,&atributos);
                printf("%o \n",atributos.st_mode);
            }
        }
    }
    closedir(direct);
    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

ejercicio2-3.c

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<unistd.h>
#include<dirent.h>
#define mymask(mode) ((mode) & ~S_IFMT)
// Permisos de ejecución para grupo y otros.
#define S_IFXGRPOTH 011
// Se define la macro con la regla para comprobar si tiene permiso x en grupo y otros.
#define regla1(mode) ((((mode) & ~S_IFMT) & 011) == S_IFXGRPOTH)
void buscar_dir(DIR *direct, char pathname[], int *reg, int *tamanio){
   struct stat atributos;
   struct dirent *ed;
   DIR *direct_act;
   char cadena[500];
   while((ed=readdir(direct)) != NULL){
      // Ignorar el directorio actual y el supoerior
      if (strcmp(ed->d_name, ".") != 0 && strcmp(ed->d_name, "..") != 0){
         sprintf(cadena,"%s/%s",pathname,ed->d_name);
         if(stat(cadena,&atributos) < 0) {
            printf("\nError al intentar acceder a los atributos de archivo");
            perror("\nError en lstat");
            exit(-1);
         }
         if (S_ISDIR(atributos.st_mode)){
            if ((direct_act = opendir(cadena)) == NULL)
               printf("\nError al abrir el directorio: [%s]\n", cadena);
            else
               buscar_dir(direct_act, cadena, reg, tamanio);
         }
         else{
            printf("%s %ld \n", cadena, atributos.st_ino);
            if (S_ISREG(atributos.st_mode)){
               if (regla1(atributos.st_mode)){
                  (*reg)++;
                  (*tamanio) += (int) atributos.st_size;
               }
            }
         }
      }
   }
   closedir(direct);
}
int main(int argc, char *argv[]){
   DIR *direct;
```

```
    char pathname[500];
    int reg=0,tamanio=0;
    if (argc==2){
        strcpy(pathname,argv[1]);
    }
    else{
        strcpy(pathname,".");
    }
    if((direct=opendir(pathname)) == NULL){
        printf("\nError al abrir directorio\n");
        exit(-1);
    }
    printf("Los inodos son: \n\n");
    buscar_dir(direct,pathname,&reg,&tamanio);
    printf("Hay %d archivos regulares con permiso x para grupo y otros\n",reg);
    printf("El tamaño total ocupado por dichos archivos es %d bytes\n",tamanio);
    return 0;
}
```

Ejercicio 4.

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

Ejercicio2-4.c

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

int n_files = 0;
int size = 0;

int visitar(const char* path, const struct stat* stat, int flags, struct FTW* ftw) {

        if ((S_ISREG(stat->st_mode)) && (stat->st_mode & S_IXGRP) && (stat->st_mode &
S_IXOTH)) {

            printf("%s %llu\n",path, stat->st_ino);

            size += stat->st_size;
            n_files++;

        }

        return 0;
```

```
}

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

        printf("Los i-nodos son:\n");

   if (nftw(argc >= 2 ? argv[1] : ".", visitar, 10, 0) != 0) {
      perror("nftw");
   }


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

        return 0;

}
```

Sesión 3. Llamadas al sistema para el Control de Procesos

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.

Ejercicio3-1.c

```
#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 <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <stdbool.h> //Cabeceras para poder usar el tipo booleano

bool es_par(int numero) {
        return numero % 2 == 0;
}
bool es_divisible_entre_cuatro(int numero) {
        return numero % 4 == 0;
}
int main(int argc, char *argv[]) {
        int numero;
```

```
	pid_t pid;
	//Comprobamos si se le ha pasado un pathname y unos permisos como parámetros
	if(argc!=2) {
		//Si no se le han pasado los parámetros correctos muestra un mensaje de ayuda
		printf("Modo de uso: %s <entero>\n\n", argv[0]);
		exit(1);
	}

	//Convertimos el argumento a int
	numero=atoi(argv[1]);

	if( (pid=fork())<0) {
		perror("\nError en el fork");
		exit(-1);
	} else if(pid==0) {  //proceso hijo ejecutando el programa
		if (es_par(numero)) {
			printf("El numero %d es par\n",numero);
		}else
			printf("El numero %d no es par\n",numero);
	} else { //proceso padre ejecutando el programa
		if (es_divisible_entre_cuatro(numero)) {
			printf("El numero %d es divisible por 4\n",numero);
		}else
			printf("El numero %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.

Tarea4.c

```
//tarea4.c
//Trabajo con llamadas al sistema del Subsistema de Procesos "POSIX 2.10 compliant"
//Prueba el programa tal y como está. Después, elimina los comentarios (1) y pruebalo de nuevo.

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

```
int global=6;
char buf[]="cualquier mensaje de salida\n";

int main(int argc, char *argv[])
{
int var;
pid_t pid;

var=88;
if(write(STDOUT_FILENO,buf,sizeof(buf)+1) != sizeof(buf)+1) {
        perror("\nError en write");
        exit(-1);
}
//(1)if(setvbuf(stdout,NULL,_IONBF,0)) {
//        perror("\nError en setvbuf");
//}
printf("\nMensaje previo a la ejecución de fork");

if( (pid=fork())<0) {
        perror("\nError en el fork");
        exit(-1);
}
else if(pid==0) {  //proceso hijo ejecutando el programa
                global++;
                var++;
        } else  //proceso padre ejecutando el programa
                sleep(1);

printf("\npid= %d, global= %d, var= %d\n", getpid(),global,var);
exit(0);

}
```

Lo que hace el programa es declaras una variable global con valor 6 y un string cualquiera llamado buf. En la función principal se declara una variable entera y una de tipo pid_t. Da el valor 88 a la variable entera y procede a imprimir por pantalla el string de antes mediante la función write en la salida estándar (STDOUT_FILENO).

Luego procede a crear el proceso hijo con la función fork y lo asigna a la variable tipo pid_t pid. Si pid vale 0 se ejecuta el código del proceso hijo, que lo único que hace es incrementar tanto la variable global como la entera var.
Si no vale 0 (es decir, su valor es el PID del proceso hijo) ejecuta el código del proceso padre, que solamente hace un sleep de 1 segundo.

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.

Ejercicio3A.c y Ejercicio3B.c

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

int main(){

  int nprocs = 21 , i = 0 , estado;
  pid_t childpid;

/*
Jerarquía de procesos tipo 1
*/
  printf( "Vamos a crear 20 hijos recursivos:\n" );
  for (i=0; i < nprocs; i++) {
    if ((childpid= fork()) == -1) {
      fprintf(stderr, "Could not create child %d: %s\n",i,strerror(errno));
      exit(-1);
    }
    if (childpid){
      break;
    }
    printf( "%d.- Tengo el PID: %d y mi padre tiene el PID: %d\n" , i+1 , getpid() , getppid() );
  }
  wait(&estado);
  return 0;
}

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

int main(){

  int nprocs = 21 , i = 0 , estado;
  pid_t childpid;

  /*
  Jerarquía de procesos tipo 2
  */
  printf( "SOY EL PROCESO PADRE CON PID: %d\n" , getpid() );
```

```
  for (i=1; i < nprocs; i++) {
    if ((childpid= fork()) == -1) {
      fprintf(stderr, "Could not create child %d: %s\n",i,strerror(errno));
      exit(-1);
    }
    if (!childpid){
      break;
    }
    wait(&estado);
  }
  if(!childpid)
    printf( "Tengo el PID: %d y mi padre tiene el PID: %d\n" , getpid() , getppid() );

  return 0;
}
```

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 Guía Práctica de Sistemas Operativos-97padre 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

ejercicio3-4.c

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

int main(){
  int i , estado , pidpadre;
  const int nprocs = 5;
  pid_t childpid[nprocs];

 printf( "PID PADRE: %d\n" , pidpadre = getpid() );
  for (i=0; i < nprocs; i++) {
    if ((childpid[i]= fork()) == -1) {
      fprintf(stderr, "Could not create child %d: %s\n",i,strerror(errno));
      exit(-1);
    }
    if (!childpid[i]){
      break;
    }
  }
```

```
  if( getpid() != pidpadre)
    printf( "Soy el hijo con PID: %d\n" , getpid() );

  for( i = 0; i < nprocs; i++){
    if( waitpid( childpid[i] , &estado , 0 ) > 0 ){
      printf("Acaba de finalizar mi hijo con PID:%d\n", childpid[i] );
            printf("Solo me quedan %d hijos vivos\n", nprocs - (i+1) );
    }
  }

  exit(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)

ejercicio3-5.c

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

int main(){
  int i , estado , pidpadre , nhijos = 5;
  const int nprocs = 5;
  pid_t childpid[nprocs];

 printf( "PID PADRE: %d\n" , pidpadre = getpid() );
  for (i=0; i < nprocs; i++) {
    if ((childpid[i]= fork()) == -1) {
      fprintf(stderr, "Could not create child %d: %s\n",i,strerror(errno));
      exit(-1);
    }
    if (!childpid[i]){
      break;
    }
  }
  if( getpid() != pidpadre)
    printf( "Soy el hijo con PID: %d\n" , getpid() );

  for( i = 0; i < nprocs; i = i+2){
    if( waitpid( childpid[i] , &estado , 0 ) > 0 ){
      printf("Acaba de finalizar mi hijo con PID:%d\n", childpid[i] );
            printf("Solo me quedan %d hijos vivos\n", nhijos-1 );
      nhijos--;
```

```
    }
  }
  for( i = 1; i < nprocs; i = i+2){
    if( waitpid( childpid[i] , &estado , 0 ) > 0 ){
      printf("Acaba de finalizar mi hijo con PID:%d\n", childpid[i] );
              printf("Solo me quedan %d hijos vivos\n", nhijos-1 );
      nhijos--;
    }
  }

  exit(0);
}
```

Ejercicio 6. ¿Qué hace el siguiente programa?

Tarea5.c

```
/*
tarea5.c
Trabajo con llamadas al sistema del Subsistema de Procesos conforme a POSIX 2.10
*/
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>
#include<errno.h>
int main(int argc, char *argv[]){
pid_t pid;
int estado;
if( (pid=fork())<0) {
perror("\nError en el fork");
exit(-1);
}
else if(pid==0) { //proceso hijo ejecutando el programa
if( (execl("/usr/bin/ldd","ldd","./tarea5",NULL)<0)) {
perror("\nError en el execl");
Guía Práctica de Sistemas Operativos-99}
exit(-1);
}
wait(&estado);
/*
<estado> mantiene información codificada a nivel de bit sobre el motivo de
finalización del proceso hijo que puede ser el número de señal o 0 si alcanzó su
finalización normalmente.
Mediante la variable estado de wait(), el proceso padre recupera el valor
especificado por el proceso hijo como argumento de la llamada exit(), pero
desplazado 1 byte porque el sistema incluye en el byte menos significativo el
código de la señal que puede estar asociada a la terminación del hijo. Por eso se
utiliza estado>>8 de forma que obtenemos el valor del argumento del exit() del
hijo.
```

```
*/
printf("\nMi hijo %d ha finalizado con el estado %d\n",pid,estado>>8);
exit(0);
}
```

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.

Ejercicio3-7.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <stdbool.h>    //Cabeceras para poder usar el tipo booleano
#include <string.h>     //Cabeceras para poder usar cadenas

void execute_command(char *str_command, char *str_params) {


   if( (execl(str_command, str_params, str_params,NULL)<0)) {

      //if( (execlp(str_command,str_params)<0)) {
            perror("Error en el execl\n");
            exit(-1);
      }

}

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

      pid_t pid;
      bool en_background = false;
      int n_params = argc;
      const char *str_bg = "bg";
      char *str_command;
      char str_params[200];


      //Comprobamos si se le ha pasado un pathname y unos permisos como parámetros
      if(argc<2) {
            //Si no se le han pasado los parámetros correctos muestra un mensaje de ayuda
            printf("Modo de uso: %s <programa> [opciones] [bg]\n\n", argv[0]);
            exit(1);
      } else {
```

```
            //Comprobamos si el ultimo parametro pasado es bg
            if (strcmp(argv[argc-1], str_bg) == 0) {
                  en_background = true;
                  //Habrá un parametro menos
                  n_params--;
            }

            //Extraemos el comando
            str_command = argv[1];

            //Recorremos los parametros pasados a partir del 2
            for (int i = 2; i < n_params; i++) {
                  strcat (str_params, argv[i]);
                  strcat (str_params, " ");
            }

            if (en_background) {
                  printf("%s se ejecutara en background\n", str_command);
               if  ((pid =fork()) < 0)
                  return (-1) ;
               else if (pid != 0)
                  exit(0) ; //El proceso padre sale

                  execute_command(str_command,str_params);

            } else {
                  printf("%s se ejecutara en foreground\n", str_command);;
                  execute_command(str_command,str_params);
            }

      }

      exit(0);

}
```

Sesión 4. Comunicación entre procesos utilizando cauces

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.

consumidorFIFO.c

```
//Consumidor que usa mecanismo de comunicacion FIFO.

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#define ARCHIVO_FIFO "ComunicacionFIFO"

int main(void)
{
int fd;
char buffer[80];// Almacenamiento del mensaje del cliente.
int leidos;

//Crear el cauce con nombre (FIFO) si no existe
umask(0);
mknod(ARCHIVO_FIFO,S_IFIFO|0666,0);
//también vale: mkfifo(ARCHIVO_FIFO,0666);

//Abrir el cauce para lectura-escritura
if ( (fd=open(ARCHIVO_FIFO,O_RDWR)) <0) {
perror("open");
exit(-1);
}

//Aceptar datos a consumir hasta que se envíe la cadena fin
while(1) {
leidos=read(fd,buffer,80);
if(strcmp(buffer,"fin")==0) {
close(fd);
return 0;
}
printf("\nMensaje recibido: %s\n", buffer);
}

return 0;
}
```

productorFIFO.c

```
//Productor que usa mecanismo de comunicacion FIFO

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

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#define ARCHIVO_FIFO "ComunicacionFIFO"

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

//Comprobar el uso correcto del programa
if(argc != 2) {
printf("\nproductorFIFO: faltan argumentos (mensaje)");
printf("\nPruebe: productorFIFO <mensaje>, donde <mensaje> es una cadena de caracteres.\n");
exit(-1);
}

//Intentar abrir para escritura el cauce FIFO
if( (fd=open(ARCHIVO_FIFO,O_WRONLY)) <0) {
perror("\nError en open");
exit(-1);
}

//Escribir en el cauce FIFO el mensaje introducido como argumento
if( (write(fd,argv[1],strlen(argv[1])+1)) != strlen(argv[1])+1) {
perror("\nError al escribir en el FIFO");
exit(-1);
}

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

Para el correcto funcionamiento del productor/consumidor primero tenemos que ejecutar el programa consumidor, ya sea en un terminal en segundo plano o en primer plano.Una vez iniciado el proceso consumidor ya podemos ejecutar el productor (en el mismo terminal si el consumidor está en segundo plano o en otro terminal si éste está en primer plano) pasándole como parámetro entre comillas el string 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”.
Queda reflejado que estamos usando un cauce FIFO porque permanece el archivo FIFO que hemos usado para la comunicación.

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. Justifique la respuesta.

```
/*
tarea6.c
Trabajo con llamadas al sistema del Subsistema de Procesos y Cauces conforme a POSIX 2.10
*/

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

int main(int argc, char *argv[])
{
int fd[2], numBytes;
pid_t PID;
char mensaje[]= "\nEl primer mensaje transmitido por un cauce!!\n";
char buffer[80];

pipe(fd); // Llamada al sistema para crear un cauce sin nombre

if ( (PID= fork())<0) {
perror("fork");
exit(1);
}
if (PID == 0) {
//Cierre del descriptor de lectura en el proceso hijo
close(fd[0]);
// Enviar el mensaje a través del cauce usando el descriptor de escritura
write(fd[1],mensaje,strlen(mensaje)+1);
exit(0);
}
else { // Estoy en el proceso padre porque PID != 0
//Cerrar el descriptor de escritura en el proceso padre
close(fd[1]);
//Leer datos desde el cauce.
numBytes= read(fd[0],buffer,sizeof(buffer));
printf("\nEl número de bytes recibidos es: %d",numBytes);
printf("\nLa cadena enviada a través del cauce es: %s", buffer);
}

return(0);
}
```

Lo primero que hace es crear un cauce con la orden pipe, pasándole como parámetro el vector de enteros fd, lo cual asigna por defecto el modo lectura a fd[0] y el de escritura a fd[1].
Después crea el hijo con la orden fork. Seguidamente entramos en la zona de código del hijo comprobando que el PID sea 0, y lo primero que hace es cerrar el descriptor de lectura del mismo. Después envía el mensaje a través del cauce usando el descriptor de escritura fd[1], mediante la orden write.
En la zona de código del proceso padre, en cambio, lo primero que hacemos es cerrar el descriptor de escritura fd[1] y seguidamente lee los datos del cauce usando el descriptor de lectura fd[0], mediante la orden read. Por último imprime el número de bytes de la cadena recibida y la misma cadena.
Hecho esto hemos conseguido la comunicación por cauces deseada en la dirección deseada, la cual también se podría hacer alrevés.

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.

```
/*
tarea7.c
Programa ilustrativo del uso de pipes y la redirección de entrada y
salida estándar: "ls | sort"
*/

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

int main(int argc, char *argv[])
{
int fd[2];
pid_t PID;

pipe(fd); // Llamada al sistema para crear un pipe

if ( (PID= fork())<0) {
perror("fork");
exit(1);
}
if(PID == 0) { // ls
//Establecer la dirección del flujo de datos en el cauce cerrando
```

```
// el descriptor de lectura de cauce en el proceso hijo
close(fd[0]);

//Redirigir la salida estándar para enviar datos al cauce
//-------------------------------------------------------
//Cerrar la salida estándar del proceso hijo
close(STDOUT_FILENO);

//Duplicar el descriptor de escritura en cauce en el descriptor
//correspondiente a la salida estándar (stdout)
dup(fd[1]);
execlp("ls","ls",NULL);
}
else { // sort. Estoy en el proceso padre porque PID != 0

//Establecer la dirección del flujo de datos en el cauce cerrando
// el descriptor de escritura en el cauce del proceso padre.
close(fd[1]);

//Redirigir la entrada estándar para tomar los datos del cauce.
//Cerrar la entrada estándar del proceso padre
close(STDIN_FILENO);

//Duplicar el descriptor de lectura de cauce en el descriptor
//correspondiente a la entrada estándar (stdin)
dup(fd[0]);
execlp("sort","sort",NULL);
}

return(0);
}
```

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.

```
/*
tarea8.c
Programa ilustrativo del uso de pipes y la redirección de entrada y
salida estándar: "ls | sort", utilizando la llamada dup2.
*/

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

```
int main(int argc, char *argv[])
{
int fd[2];
pid_t PID;

pipe(fd); // Llamada al sistema para crear un pipe

if ( (PID= fork())<0) {
perror("\Error en fork");
exit(-1);
}
if (PID == 0) { // ls
//Cerrar el descriptor de lectura de cauce en el proceso hijo
close(fd[0]);

//Duplicar el descriptor de escritura en cauce en el descriptor
//correspondiente a la salida estda r (stdout), cerrado previamente en
//la misma operación
dup2(fd[1],STDOUT_FILENO);
execlp("ls","ls",NULL);
}
else { // sort. Proceso padre porque PID != 0.
//Cerrar el descriptor de escritura en cauce situado en el proceso padre
close(fd[1]);

//Duplicar el descriptor de lectura de cauce en el descriptor
//correspondiente a la entrada estándar (stdin), cerrado previamente en
//la misma operación
dup2(fd[0],STDIN_FILENO);
execlp("sort","sort",NULL);
}

return(0);
}
```

El resultado de la ejecución es el mismo en ambos.
Las diferencias a nivel de código son que el primer programa usa la orden close para cerrar la salida estándar y así dejar la entrada del descriptor de lectura del hijo libre y luego dup para duplicar el descriptor de escritura en el cauce. Lo mismo hace con el proceso padre pero con la entrada estándar.
- El segundo usa solamente la orden dup2, que hace lo mismo que las otras dos juntas en una sola orden, cerrando el descriptor antiguo y duplicando el descriptor después. Se garantiza que la llamada es atómica, por lo que si por ejemplo, si llega una señal al proceso, toda la operación transcurrirá antes de devolverle el control al núcleo para gestionar la señal.

Así pues hemos creado un cauce en el que el proceso hijo ejecuta la orden ls y lo redirecciona al descriptor de escritura de salida deseado, no a la salida estándar, con lo cual el proceso padre, en el que también hemos redirigido la entrada para que no sea la

estándar, recibe el la información de la orden que ejecutó el hijo y seguidamente ejecuta la orden sort. Con la salida que da el programa podemos comprobar que el cauce se ha efectuado correctamente.

Ejercicio 5.

Este ejercicio se basa en la idea de utilizar varios procesos para realizar partes de una computación en paralelo. Para ello, deberá construir un programa que siga el esquema de computación maestro-esclavo, en el cual existen varios procesos trabajadores (esclavos) idénticos y un único proceso que reparte trabajo y reúne resultados (maestro). Cada esclavo es capaz de realizar una computación que le asigne el maestro y enviar a este último los resultados para que sean mostrados en pantalla por el maestro.

Guía Práctica de Sistemas Operativos-116El ejercicio concreto a programar consistirá en el cálculo de los números primos que hay en un

intervalo. Será necesario construir dos programas, maestro y esclavo. Ten en cuenta la siguiente especificación:

1. El intervalo de números naturales donde calcular los número primos se pasará como argumento al programa maestro. El maestro creará dos procesos esclavos y dividirá el intervalo en dos subintervalos de igual tamaño pasando cada subintervalo como argumento a cada programa esclavo. Por ejemplo, si al maestro le proporcionamos el intervalo entre 1000 y 2000, entonces un esclavo debe calcular y devolver los números primos comprendidos en el subintervalo entre 1000 y 1500, y el otro esclavo entre 1501 y 2000. El maestro creará dos cauces sin nombre y se encargará de su redirección para comunicarse con los procesos esclavos. El maestro irá recibiendo y mostrando en pantalla (también uno a uno) los números primos calculados por los esclavos en orden creciente.
2. El programa esclavo tiene como argumentos el extremo inferior y superior del intervalo sobre el que buscará números primos. Para identificar un número primo utiliza el siguiente método concreto: un número n es primo si no es divisible por ningún k tal que $2 < k <= sqrt(n)$, donde sqrt corresponde a la función de cálculo de la raíz cuadrada (consulte dicha función en el manual).

El esclavo envía al maestro cada primo

encontrado como un dato entero (4 bytes) que escribe en la salida estándar, la cuál se tiene que encontrar redireccionada a un cauce sin nombre. Los dos cauces sin nombre necesarios, cada uno para comunicar cada esclavo con el maestro, los creará el maestro inicialmente. Una vez que un esclavo haya calculado y enviado (uno a uno) al maestro todos los primos en su correspondiente intervalo terminará.

Maestro.c

```
#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 <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <stdbool.h> //Cabeceras para poder usar el tipo booleano
```

```
#define ARCHIVO_FIFO "comunicacion"

int main(int argc, char *argv[]) {
	int inicio;
	int fin;
	int mitad;
	int numBytes;
	char buffer[80];
	char params[2];
	//Comprobamos si se le ha pasado dos enteros como parámetro
	if(argc!=3) {
		//Si no se le han pasado los parámetros correctos muestra un mensaje de ayuda
		printf("Modo de uso: %s <inicio> <fin>\n\n", argv[0]);
		exit(1);

	}
	//Convertimos el argumento a int
	inicio=atoi(argv[1]);
	fin=atoi(argv[2]);
	mitad = inicio + ((fin - inicio) / 2);
	int fd1[2];
	int fd2[2];
	pid_t PID;

	pipe(fd1); // Llamada al sistema para crear un pipe
	pipe(fd2); // Llamada al sistema para crear un pipe

	if ( (PID= fork())<0) {
		perror("Error al hacer fork");
		exit(1);
	}
	if (PID == 0) {
		//Cierre del descriptor de lectura en el proceso hijo
		close(fd1[0]);
		params[0] = inicio;
		params[1] = mitad;
		// Enviar el mensaje a través del cauce usando el descriptor de escritura
		printf("paso 1\n");
		dup2(fd1[1],STDOUT_FILENO);
		execlp("esclavo","esclavo", params, NULL);
	} else { // Estoy en el proceso padre porque PID != 0
		if ( (PID= fork())<0) {
			perror("Error al hacer fork");
			exit(1);
		}
		if (PID == 0) {
			close(fd2[0]);
			params[0] = mitad + 1;
			params[1] = fin;
```

```
                printf("paso 2\n");
                dup2(fd2[1],STDOUT_FILENO);
                execlp("esclavo","esclavo", params,NULL);
                exit(0);
        } else { // Estoy en el proceso padre porque PID != 0

                //Cerrar el descriptor de escritura en el proceso padre
                close(fd1[1]);
                //Leer datos desde el cauce.
                numBytes= read(fd1[0],buffer,sizeof(buffer));
                dup2(fd1[0],STDIN_FILENO);
                printf("paso 3\n");
                //Recorremos el archivo de lectura leyendo en tramos de 80 bytes
                while ((numBytes = read(fd1[0], &buffer, 4)) > 0) {
                        printf("Valor: %d",1);
                }
         }
    }
    exit(0);
}
```

esclavo.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <math.h>
int main(int argc, char *argv[]) {

    int inicio;
    int fin;

    //Comprobamos si se le ha pasado dos enteros como parámetro
    if(argc!=3) {

        //Si no se le han pasado los parámetros correctos muestra un mensaje de ayuda
        printf("Modo de uso: %s <inicio> <fin>\n\n", argv[0]);
        exit(1);
    }
    //Convertimos el argumento a int
    inicio=atoi(argv[1]);
    fin=atoi(argv[2]);

    for (int n = inicio; n < fin; n++) {
    int c;
    //NOTA: Calculo si es primo a la vieja usanza por que no acabo de hacer funcionar
    //el método rápido con la raiz cuadrada
```

```
        for (c = 2;c <= n - 1; c++) {
                if ( n % c == 0 )
                        break; //No es primo, paramos la iteracion
        }
        if (c == n)
                printf("El numero %d es primo.\n", n);
        }
        exit(0);
}
```

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

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

```
#include <sys/types.h> //POSIX Standard: 2.6 Primitive System Data Types
// <sys/types.h>
#include<limits.h> //Incluye <bits/posix1_lim.h> POSIX Standard: 2.9.2 //Minimum    //Values Added
to <limits.h> y <bits/posix2_lim.h>
#include <unistd.h> //POSIX Standard: 2.10 Symbolic Constants         <unistd.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>
int main(int argc, char *argv[])
 {
         long int pid;
         int signal;
         if(argc<3) {
         printf("\nSintaxis de ejecución: envioSignal [012] <PID>\n\n");
         exit(-1);
          }
        pid= strtol(argv[2],NULL,10);
        if(pid == LONG_MIN || pid == LONG_MAX)
          {
        if(pid == LONG_MIN)
        printf("\nError por desbordamiento inferior LONG_MIN %d",pid);
        else
           printf("\nError por desbordamiento superior LONG_MAX %d",pid);
           perror("\nError en strtol");
           exit(-1);
          }
       signal=atoi(argv[1]);
       switch(signal) {
           case 0: //SIGTERM
           kill(pid,SIGTERM); break;
           case 1: //SIGUSR1
```

```
          kill(pid,SIGUSR1); break;
          case 2: //SIGUSR2
          kill(pid,SIGUSR2); break;
          default : // not in [012]
          printf("\n No puedo enviar ese tipo de señal");
            }
}
```

reciboSignal.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <errno.h>
static void sig_USR_hdlr(int sigNum)
  {

    if(sigNum == SIGUSR1)
    printf("\nRecibida la señal SIGUSR1\n\n");
    else if(sigNum == SIGUSR2)
    printf("\nRecibida la señal SIGUSR2\n\n");
   }

int main(int argc, char *argv[])
   {
    struct sigaction sig_USR_nact;
    if(setvbuf(stdout,NULL,_IONBF,0))
       {
    perror("\nError en setvbuf");
       }

//Inicializar la estructura sig_USR_na para especificar la nueva acción para la señal.

sig_USR_nact.sa_handler= sig_USR_hdlr;


//'sigemptyset' inicia el conjunto de señales dado al conjunto vacío.

sigemptyset (&sig_USR_nact.sa_mask);
sig_USR_nact.sa_flags = 0;

//Establecer mi manejador particular de señal para SIGUSR1
if( sigaction(SIGUSR1,&sig_USR_nact,NULL) <0)
    {
perror("\nError al intentar establecer el manejador de señal para SIGUSR1");
exit(-1);
    }
```

```
//Establecer mi manejador particular de señal para SIGUSR2
if( sigaction(SIGUSR2,&sig_USR_nact,NULL) <0)
 {
perror("\nError al intentar establecer el manejador de señal para SIGUSR2");
exit(-1);
    }
for(;;)
   {
   }
}
```

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.

```
kawtar@kawtar-VirtualBox:~$ ./contador &
[2] 1899
kawtar@kawtar-VirtualBox:~$
No puedo manejar la señal 9
No puedo manejar la señal 19
Esperando el envío de señales...
kill -SIGINT 1899
kawtar@kawtar-VirtualBox:~$ La señal 2 se ha recibido 1 veces
kill -SIGINT 1899
La señal 2 se ha recibido 2 veces
kill -15 1899
kawtar@kawtar-VirtualBox:~$ La señal 15 se ha recibido 1 veces
kill -111 1899
bash: kill: 111: especificación de señal inválida
kawtar@kawtar-VirtualBox:~$ kill -15 1899 // el programa no puede capturar la
señal 15
[2]+ Detenido
./contador
kawtar@kawtar-VirtualBox:~$ kill -cont 1899
La señal 18 se ha recibido 1 veces
kawtar@kawtar-VirtualBox:~$ kill -KILL 1899
[2]+
Terminado (killed)
./contador
```

contador.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdbool.h> //Cabeceras para poder usar el tipo booleano
```

```
//Vector donde almacenar el contador de señales
const static int SIGNAL_SIZE = 10;
static int received_signals[SIGNAL_SIZE];


static void sig_handler (int signum) {

	if (signum >= 1 && signum <= SIGNAL_SIZE) {

		//Incrementamos el elemento del vector de señales recibidas
		received_signals[signum]++;

		printf("La señal %d se ha recibido %d veces\n", signum, received_signals[signum]);

	} else {

		printf("No puedo manejar la señal %d\n", signum);

	}
}

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

	//Declaración de variables
	struct sigaction sig_action;

	//Asociamos la funcion definida como manejador
	sig_action.sa_handler = sig_handler;

	//inicializar un conjunto con todas las señales
	sigfillset(&sig_action.sa_mask);


	sig_action.sa_flags = 0;

	for (int i = 1; i < SIGNAL_SIZE - 1; i++) {

		if (sigaction(i, &sig_action, NULL) < 0) {
			perror("Error al intentar establecer el manejador de señal\n");
			exit(-1);
		}
	}

	//Dejamos en ejecución un bucle infinito, asi el programa no termina
	while(true);

}
```

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.

Ejercicio5-3.c

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

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

	//definimos una nueva mascara
	sigset_t new_mask;

	//inicializar un conjunto con todas las señales
	sigfillset(&new_mask);

	//eliminamos la señal SIGUSR1
	sigdelset(&new_mask, SIGUSR1);

	//esperar solamente a la señal SIGUSR1
	sigsuspend(&new_mask);

	return 0;


}
```

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

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

static int signal_recibida = 0;

static void manejador (int sig)
{
	signal_recibida = 1;
}

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

```
    sigset_t conj_mascaras_original;
    struct sigaction act;
//Iniciamos a 0 todos los elementos de la estructura act
    memset (&act, 0, sizeof(act));

    act.sa_handler = manejador;

    if (sigaction(SIGTERM, &act, 0)) {
        perror ("sigaction");
        return 1;
    }

    //Iniciamos un nuevo conjunto de mascaras
    sigemptyset (&conjunto_mascaras);
    //Añadimos SIGTERM al conjunto de mascaras
    sigaddset (&conjunto_mascaras, SIGTERM);

    //Bloqueamos SIGTERM
    if (sigprocmask(SIG_BLOCK, &conjunto_mascaras, &conj_mascaras_original) < 0) {
       perror ("primer sigprocmask");
       return 1;
    }

    sleep (10);

    //Restauramos la señal ơ desbloqueamos SIGTERM
    if (sigprocmask(SIG_SETMASK, &conj_mascaras_original, NULL) < 0) {
       perror ("segundo sigprocmask");
       return 1;
       }

    sleep (1);

   if (signal_recibida)
       printf ("\nSenal recibida\n");
   return 0;
}
```

El programa crea una máscara, en la que solo añade la señal SIGTERM. Aplica la máscara y con ello bloquea la señal SIGTERM. Realiza un sleep de 10s, y si durante este tiempo le mandamos dicha señal no reaccionará porque está bloqueada. (la variable signal_recibida se activará) Una vez termina de “dormir”, desbloquea la señal SIGTERM reanudando la máscara antigua y comprueba si la hemos introducido comprobando si la variable signal_recibida está activada , si es así nos muestra que la señal ya ha sido recibida , si no acabará el programa sin mostrar nada.

Sesión 6. Control de archivos y archivos proyectados a memoria

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

ejercicio6-1.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <stdbool.h> //Cabeceras para poder usar el tipo booleano

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 muestra un mensaje de ayuda
		printf("Modo de uso: %s <programa> <simbolo> <archivo>\n\n", argv[0]);
		exit(1);
	} else {

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

		//Extraemos el comando
		str_command = argv[1];
		str_file = argv[3];

		//Comprobamos el segundo parametro, tiene que ser < o >
		if (strcmp(argv[2], "<") == 0) {

			//Redireccion de entrada
			fd = open (str_file, O_RDONLY);
			close(STDIN_FILENO);
			if (fcntl(fd, F_DUPFD, STDIN_FILENO) == -1 ) perror ("fcntl falló");

		} else if (strcmp(argv[2], ">") == 0) {
```

```
                        //Redireccion de salida
                        fd = open (str_file, O_CREAT|O_WRONLY);
                        close (STDOUT_FILENO);
                        if (fcntl(fd, F_DUPFD, STDOUT_FILENO) == -1 ) perror ("fcntl falló");


                } else {

                        printf("Debe pasarse \"<\" o \">\" con las comillas %s\n\n", argv[2]);

                        //Si no se le han pasado los parámetros correctos muestra un mensaje de ayuda
                        printf("Modo de uso: %s <programa> [opciones] [bg]\n\n", argv[0]);
                        exit(1);

                }

                //Ejecutamos el comando
            if( (execlp(str_command, "", NULL) < 0)) {

                        perror("Error en el execlp\n");
                        exit(-1);
                }

                //Cerramos el fichero
                close(fd);

        }

}
```

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

ejercicio6-2.c

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

int main( int argc , char *argv[]){
  if( argc < 2 ){
```

```
    printf("Sintaxis: %s [programa] '|' [programa]",argv[0]);
    exit(-1);
  }
  int fd[2];

  pid_t pid;
  pipe(fd);

  if( (pid = fork()) < 0 ){
    perror("Error en fork");
    exit(-1);
  }
  if( pid == 0 ){
    close(fd[0]);
    close(1);
    if( (fcntl( fd[1], F_DUPFD, STDOUT_FILENO)) < 0 ){
      perror("Error en primer fcntl");
      exit(-1);
    }
    execlp(argv[1],argv[1],NULL);
  }
  else{
    close(fd[1]);
    close(0);
    if( (fcntl( fd[0], F_DUPFD, STDIN_FILENO)) < 0 ){
      perror("Error en segundo fnctl");
      exit(-1);
    }
    execlp(argv[3],argv[3],NULL);
  }
  return 0;
}
```

Ejercicio 3.

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

Ejercicio6-3.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.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 muestra un mensaje de ayuda
	printf("Modo de uso: %s <programa> <archivo>\n\n", argv[0]);
	exit(1);

} else {

	//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;
	}

	cerrojo.l_type=F_WRLCK;
	cerrojo.l_whence=SEEK_SET;
	cerrojo.l_start=0;
	cerrojo.l_len=0; //Bloquearemos el archivo entero

	//Intentamos un bloqueo de escritura del archivo
	printf ("Intentando bloquear %s\n", str_file);
	if (fcntl (fd, F_SETLKW, &cerrojo) == EDEADLK) {

		//Si el cerrojo falla, pintamos un mensaje
		printf ("%s ha dado un EDEADLK\n", str_file);

	} //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 a lanzar otra vez el programa
	for (i = 0; i < 10; i++) {
		sleep(1);
	}
	//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_SETLKW, &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.

…

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().

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <stdbool.h> //Cabeceras para poder usar el tipo booleano
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

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

        //Comprobamos si se le ha pasado un pathname y unos permisos como parámetros
        if(argc != 3) {
                //Si no se le han pasado los parámetros correctos muestra un mensaje de ayuda
                printf("Modo de uso: %s <programa> <origen> <destino>\n\n", argv[0]);
                exit(1);

        } else {

                //Declaracion de variables
      struct stat sb;
                char *str_orig = argv[1];
                char *str_dest = argv[2];
```

```
	int fd_orig, fd_dest;
	char *mem_orig, *mem_dest;
	int filesize;


	//Abrimos el fichero de origen
	fd_orig = open(str_orig, O_RDONLY);
	if (fd_orig == -1) {
		perror("Fallo al abrir el archivo de origen\n");
		exit(2);
	}

	//Obtenemos su stat, para comprobar si es regular y obtener su tamaño
if (fstat (fd_orig, &sb) == -1) {
	printf("Error al hacer stat en el fichero de origen\n");
	return 1;
}

if (!S_ISREG (sb.st_mode)) {
	printf ("El fichero de origen no es un archivo regular\n");
	return 1;
}

//Guardamos el tamaño en una variable (por comodidad)
filesize = sb.st_size;


//Creamos el archivo de destino
umask(0);
	fd_dest = open(str_dest, O_RDWR|O_CREAT|O_EXCL, S_IRWXU);
	if (fd_dest == -1) {
		perror("Fallo al crear el archivo de salida");
		exit(2);
	}

	//Asignamos el espacion en el fichero de destino
	ftruncate(fd_dest, filesize);

	//Creamos el mapa de memoria del fichero de origen
	mem_orig = (char *) mmap(0, filesize, PROT_READ, MAP_SHARED, fd_orig, 0);
	if(mem_orig == MAP_FAILED) {

		perror("Fallo mapeando el archivo de entrada");
		exit(2);

	}

	//Creamos el mapa de memoria del fichero de destino
	mem_dest = (char *) mmap(0, filesize, PROT_WRITE, MAP_SHARED, fd_dest, 0);
```

```
        if(mem_dest == MAP_FAILED) {

                perror("Fallo mapeando el archivo de salida");
                exit(2);

        }

        //Copiamos un mapa de memoria en otro
        memcpy(mem_dest, mem_orig, filesize);


        //Liberamos los mapas de memoria
        munmap(mem_orig, filesize);
        munmap(mem_dest, filesize);

        //Cerramos los descriptores de fichero
        close(fd_orig);
        close(fd_dest);

        //Terminamos la ejecución del programa
        return 0;


    }

}
```

Sesión 7. Construcción de un spool de impresión