

Sumario

Aspectos importantes para programar en C.....	2
Compile y ejecutar:.....	2
Includes que nos harán falta:.....	2
Aspectos importantes:.....	2
Sesión 1. Las llamadas al sistema para el SA (Parte I).....	4
Actividades Sesión 1:.....	6
Repaso Sesión 1:.....	7
Sesión 2. Las llamadas al sistema para el SA (Parte II).....	8
Actividades Sesión 2:.....	9
Repaso Sesión 2:.....	11
Sesión 3. Las llamadas al sistema para el Control de Procesos.....	12
Actividades Sesión 3:.....	14
Repaso Sesión 3:.....	14
Sesión 4. Comunicación entre procesos utilizando cauces.....	15
Actividades Sesión 4:.....	17
Repaso Sesión 4:.....	23
Sesión 5. Llamadas al sistema para gestión y control de señales.....	23
Actividades Sesión 5:.....	26
Repaso Sesión 5:.....	26
Sesión 6. Control de archivos y archivos proyectados a memoria.....	28
Repaso Sesión 6:.....	28
Sesión 7. Construcción de un Spool de impresión.....	29

Aspectos importantes para programar en C

Compilar y ejecutar:

Si nuestro programa se llama ejercicio.c ...

Dar permisos: `chmod u+x ejercicio.c`

Compilar: `gcc ejercicio.c -o ej`

Ejecutar: `./ej [Parámetros]`

Includes que nos harán falta:

Para saber que includes usar escribimos en la terminal: `$ man 2 open , $ man 2 read ...`

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

La siguiente tabla muestra los números de sección del manual y los tipos de páginas que contienen:

`$> man <numero_sección> <orden>`

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

Aspectos importantes:

1) Tipos de print:

A la hora de hacer un programa (utilizando lenguaje C y gcc para compilar), cuando se quiera escribir por pantalla, usaremos la función printf.

printf (cadena de formato, parámetros)

Si en la cadena de formatos nos encontramos con el símbolo “%”, se refiere a los parámetros que hay después de la coma. Tras el símbolo “%” encontraremos una letra, la cual hace referencia al tipo de dato que vamos a leer:

- **c** : character
- **d** : entero
- **f** : float
- **x** : hexadecimal
- **o** : octal
- **s** : string (char *)

- **fprintf(stderr, "Error: numero de argumentos incorrectos");** → print de error

- **sprintf(ruta, "%s/%s", argv[1], entrada → d_name);** → en vez de mostrarlo por pantalla, lo guarda en una cadena de caracteres pasada como primer argumento.

2) sizeof de un argumento: `strlen(argv[1])*sizeof(char)`

3) Generar números aleatorios

```
srand(time(NULL)); //Semilla para que se generen números distintos al ejecutar
int aleatorio = (rand()%MAX) + MIN;
```

4) Comprobar si el número de argumentos de main es correcto:

```
int main(int argc, char*argv[]){
    if(argc != 3){
        //perror NO porque no ha habido una llamada del sistema
        //Opcion 1: printf("Numero de argumentos incorrectos");

        fprintf(stderr, "Error: numero de argumentos incorrectos");
        exit(2);
    }
}
```

4) Leer entero por teclado: `scanf("%d", &valor);`

5) `char * buffer = malloc(sizeof(char*));` → Asigna la memoria solicitada y devuelve un puntero.

4) `buffer = memset(buffer, 0, sizeof(char*))` → Rellena de 0s el buffer

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

Archivos

Principales llamadas al sistema:

- `open("ruta",flags)`

Esta operación *devuelve un descriptor de archivo*, el cual es un entero no negativo que representa la apertura de un fichero.

NOTA: Si abrimos dos veces el mismo archivo, obtenemos DOS descriptors de archivo DIFERENTES.

Por convenio, los shell de Linux asocian el descriptor de archivo **0** con la entrada estándar de un proceso, el descriptor de archivo **1** con la salida estándar y el descriptor **2** con la salida de error *STDIN_FILENO*, *STDOUT_FILENO*, *STDERR_FILENO*, definidos en *<unistd.h>*.

ruta es el nombre del archivo que queremos abrir

flags es el modo de apertura:

* **O_RDONLY** -- Solo lectura

* **O_WRONLY** -- Solo escritura

* **O_RDWR** -- Lectura y escritura

* **O_APPEND** -- Escritura y añade información al final del fichero

* **O_TRUNC** -- Borra todos los datos del fichero cuando se abre

* **O_CREAT** -- Si no existe el fichero, lo crea

* **O_EXCL** -- Va en combinación con **O_CREAT**. Si el fichero ya existe, falla (evita manipular ficheros por error)

Los flags se *concatenan* usando "|".

Cuando usamos **O_CREAT**, debemos añadir con qué permisos creamos el archivo, por lo cual la orden `open` pasa a ser: **`open("ruta",flags,permisos);`**

TODOS los permisos empiezan por las letras **S_I** y pueden ser:

* **S_IRWX** -> U (usuario), G (grupo) ó O (otro)

ó * **S_I** -> **R** | **USR**

-> **W** | **GRP**

-> **X** | **OTH**

EJEMPLO: Permisos de lectura y escritura a usuario y lectura a grupo:
S_IRUSR | S_IWUSR | S_IRGRP

EJEMPLO:

```
int fd = open("mifichero", O_RDONLY | O_TRUNC);

if(fd > 0){    //Open si falla devuelve -1
    perror("error en el open");
    exit(1); //Se usara para indicar que hay un fallo en una LLAMADA AL SISTEMA
}
```

- **read(fd, dirección de memoria del dato a leer, tamaño en bytes del dato a leer)**
(¡LA FUNCIÓN **WRITE** ES IGUAL!!)

Devuelve la cantidad de Bytes leídos (o -1 en caso de error).

Las operaciones de lectura y escritura comienzan normalmente en la posición actual y provocan un incremento en dicha posición, igual al número de Bytes leídos o escritos. Por defecto, esta posición está inicializada a 0 cuando se abre un archivo, a menos que se especifique la opción **O_APPEND**. La posición actual (*current_offset*) de un archivo abierto puede cambiarse explícitamente utilizando la llamada al sistema **lseek**.

NOTA: ¿Qué pasa si hay un archivo con datos y read devuelve 0? No queda nada que leer.

¿Y si write devuelve 0? Significa que no hay espacio en disco.

Para obtener el tamaño en Bytes de un dato, usaremos la función **sizeof**.

EJEMPLO:

```
int valor;

if (read(fd, &valor, sizeof(int)) < 0 ){
    perror("error en el read");
    exit(1);
}
```

- **close(fd)**: Cierra el descriptor de archivo.

Estructura STAT

Con ella podemos consultar los metadatos (atributos) de un archivo.

Para crear una estructura en C: **struct stat * st;**

Algunos de los atributos más importantes son:

- **st_ino** – inodo (número que identifica ese fichero).

- **st_size** – tamaño en Bytes.

- **st_mode** – permisos.

```
struct stat {
    dev_t st_dev; /* n° de dispositivo (filesystem) */
    dev_t st_rdev; /* n° de dispositivo para archivos especiales */
    ino_t st_ino; /* n° de inodo */
    mode_t st_mode; /* tipo de archivo y mode (permisos) */
    nlink_t st_nlink; /* número de enlaces duros (hard) */
    uid_t st_uid; /* UID del usuario propietario (owner) */
    gid_t st_gid; /* GID del usuario propietario (owner) */
    off_t st_size; /* tamaño total en bytes para archivos regulares */
    unsigned long st_blksize; /* tamaño bloque E/S para el sistema de archivos */
    unsigned long st_blocks; /* número de bloques asignados */
    time_t st_atime; /* hora último acceso */
    time_t st_mtime; /* hora última modificación */
    time_t st_ctime; /* hora último cambio */
};
```

st_mode también proporciona información sobre qué tipo de fichero es con el que está tratando:

S_IFMT	0170000	máscara de bits para los campos de bit del tipo de archivo (no POSIX)
S_IFSOCK	0140000	socket (no POSIX)
S_IFLNK	0120000	enlace simbólico (no POSIX)
S_IFREG	0100000	archivo regular (no POSIX)
S_IFBLK	0060000	dispositivo de bloques (no POSIX)

S_IFDIR	0040000	directorio (no POSIX)
S_IFCHR	0020000	dispositivo de caracteres (no POSIX)
S_IFIFO	0010000	cauce con nombre (FIFO) (no POSIX)
S_ISUID	0004000	bit SUID
S_ISGID	0002000	bit SGID
S_ISVTX	0001000	sticky bit (no POSIX)
S_IRWXU	0000700	user (propietario del archivo) tiene permisos de lectura, escritura y ejecución
S_IRUSR	0000400	user tiene permiso de lectura (igual que S_IREAD, no POSIX)
S_IWUSR	0000200	user tiene permiso de escritura (igual que S_IWRITE, no POSIX)
S_IXUSR	0000100	user tiene permiso de ejecución (igual que S_IEXEC, no POSIX)
S_IRWXG	0000070	group tiene permisos de lectura, escritura y ejecución
S_IRGRP	0000040	group tiene permiso de lectura
S_IWGRP	0000020	group tiene permiso de escritura
S_IXGRP	0000010	group tiene permiso de ejecución
S_IRWXO	0000007	other tienen permisos de lectura, escritura y ejecución
S_IROTH	0000004	other tienen permiso de lectura
S_IWOTH	0000002	other tienen permiso de escritura
S_IXOTH	0000001	other tienen permiso de ejecución

EJEMPLO: Mostrar por pantalla el inodo de un fichero.

```
struct stat st;

stat("archivo.txt", &st);

printf("i-nodo: %d", st.st_ino);
```

Actividades Sesión 1:

Ejercicio 1: Programa que copie lo que hay en el fichero datos.txt en copia.txt

```
1 int main(int argc, char * argv[]){
2
3     if(argc != 3){
4         /*
5          *error("Numero de parametros incorrecto");
6          *NO escribimos error porque no ha habido una llamada del sistema
7          */
8
9         fprintf(stderr, "Error: Numero de parametros incorrecto\n");
10        exit(2);
11    }
12
13    int fd, fd_2;
14    char valor; // Variable que se va a leer
15
16    // Abrimos "datos.txt" para lectura
17    if((fd = open(argv[1], O_RDONLY)) < 0){
18        perror("error al abrir datos.txt");
19        exit(1);
20    }
21
22    // Abrimos "copia.txt" con permisos de escritura. Si no existe se
23    // crea. O_TRUNC sera por si volvemos a ejecutar habiendo modificado
24    // datos.txt, borramos lo que hay y volvemos a copiar
25    if((fd_2 = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IWUSR |
26        S_IRUSR)) < 0){
27        perror("error al abrir copia.txt");
28        exit(1);
29    }
30
31    // Mientras podamos leer de datos.txt, escribiremos en copia.txt
32    while((read(fd, &valor, sizeof(char))) > 0){
33        if(write(fd_2, &valor, sizeof(char)) < 0){
34            perror("Error al escribir");
35            exit(1);
36        }
37    }
38    close(fd);
39    close(fd_2);
40}
```

Ejercicio 2: Programa que muestre el i-nodo y el tamaño de un archivo pasado por argumento

```
1 int main(int argc, char * argv[]){
2
3     if(argc != 2){
4         fprintf(stderr, "Error: Numero de parametros incorrecto\n");
5         exit(1);
6     }
7
8     struct stat st;
9     stat(argv[1], &st);
10
11     printf("i nodo: %d y tamaño: %d\n", st.st_ino, st.st_size);
12
13 }
```

Ejercicio 3: Programa que reciba dos argumentos: primero el nombre de un fichero en el cual vamos a escribir el nombre del segundo fichero, solamente si este es regular y tiene mas de 100 Bytes.

```
1 int main(int argc, char * argv[]){
2     if(argc != 3){
3         fprintf(stderr, "Error: Numero de parametros incorrecto\n");
4         exit(2);
5     }
6
7     int fd;
8
9     if((fd = open(argv[1], O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR |
10         S_IWUSR)) < 0){
11         perror("Error al abrir el descriptor de fichero");
12         exit(1);
13     }
14
15     struct stat st;
16     stat(argv[2], &st);
17
18     // Si el archivo pasado como SEGUNDO argumento es regular
19     if(S_ISREG(st.st_mode)){
20         // Si tiene mas de 100 Bytes
21         if(st.st_size > 100){
22             // Escribimos en fd el valor de la variable argv[2] (nombre del
23             // 2ndo parametro)
24             if(write(fd, argv[2], strlen(argv[2])*sizeof(char)) < 0){
25                 perror("Error en el write");
26                 exit(1);
27             }
28         }
29     }
30     close(fd);
31 }
```

Repaso Sesión 1:

Pregunta 1: ¿Para qué sirve el flag O_WRONLY? ¿Cómo podemos descubrirlo?

Solamente escritura. Podemos descubrirlo con man 2 open.

Pregunta 2: ¿Qué hace la orden lseek(fd,40,SEEK_SET)?

Posiciona el puntero de lectura en el archivo fd en la posición 40.

Si el fichero tuviese 10 bytes y posicionamos el puntero de lectura en la posición 40 no da error, sino que rellena los bytes de por medio con valor 0. Esto podríamos comprobarlo con la orden **od -c archivo** (ver un archivo en octal).

```
abcdefghijABCDEFGHIJ[ luis: ~/Desktop/SO-P-Todos_MaterialModc archivo n1
0000000  a  b  c  d  e  f  g  h  i  j  \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000040  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000060  I  J
0000062
```

Pregunta 3: ¿Para qué sirven las opciones O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR | S_IWUSR?

Para indicar el tipo de acceso y flags de un archivo. Podemos consultar estos datos en man 2 open.

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

Llamada al sistema Umask

La llamada al sistema Umask *fija la máscara de creación de permisos* para el proceso y devuelve el valor previamente establecido.

El argumento de la llamada puede formarse mediante una combinación OR de las nueve constantes de permisos vistas.

SINOPSIS

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

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

La máscara de usuario es usada por open(2) para establecer los permisos iniciales del archivo que se va a crear.

En resumen, con **umask(0) -> podemos dar cualquier permiso (0 habilita y 1 deshabilita).**

Los permisos si el fichero YA EXISTE NO SE CAMBIAN.

Llamada al sistema chmod

Cambian los permisos de acceso para un archivo del sistema de archivos.

chmod sobre un archivo especificado por su *pathname* y fchmod sobre un *archivo* que ha sido *previamente abierto* con open.

```
chmod(“fichero”permisos);
```

Devuelve **0** (bien) o **-1** (mal).

Los permisos se pueden especificar mediante un OR lógico:

- Operaciones a nivel de bit:

* | OR

* & AND

* ~ NOT

Directorios

-**opendir**: se le pasa el *pathname* del directorio al abrir y devuelve un puntero a la estructura de tipo DIR, llamada stream de directorio. El tipo DIR está definido en *<dirent.h>*.

- **readdir**: lee la entrada donde esté situado el puntero de lectura de un directorio ya abierto cuyo stream se pasa a la función. Después de la lectura adelanta el puntero una posición. Devuelve la entrada leída a través de un puntero a una estructura (***struct dirent***), o devuelve ***NULL*** si llega al final del directorio o se produce un error.

- **closedir**: cierra un directorio, devuelve 0 si tiene éxito, en caso contrario devuelve -1.

- **seekdir**: permite situar el puntero de lectura de un directorio (se tiene que usar en combinación con telldir).
- **telldir**: devuelve la posición del puntero de lectura de un directorio.
- **rewinddir**: posiciona el puntero de lectura al principio del directorio.

Tipos de datos para recorrer un directorio:

```
DIR * directorio;
struct dirent * entrada; (cada una de las cosas que tiene el directorio)
    → Entradas especiales: . - directorio actual y .. - directorio padre.
```

Recorrer todas las entradas de un directorio:

```
while((entrada = readdir(directorio)) != NULL){
    printf("%s\n", entrada->d_name);
}
```

Acceder al nombre de la entrada del directorio:

```
(*entrada).d_name
entrada->d_name
```

Actividades Sesión 2:

Ejercicio 4: Programa que reciba un argumento: el nombre de un directorio. Añadir permisos de escritura para otros a todos los ficheros regulares contenidos en dicho directorio.

```
1 int main(int argc, char*argv[]) {
2     if(argc != 2){
3         fprintf(stderr, "Error: Numero de parametros incorrecto\n");
4         exit(2);
5     }
6
7     DIR * directorio;
8     struct dirent * entrada; //Cada una de las cosas que tiene el
9         directorio
10
11     // Primero abrimos el directorio
12     if((directorio = opendir(argv[1])) == NULL){
13         perror("Error al abrir el directorio");
14         exit(1);
15     }
16     /*
17     El uso de chmod es chmod("fichero", permisos)...
18     Sin embargo, si dentro del directorio que estamos recorriendo hay
19     otro directorio, tenemos que acceder a l y el chmod fallaria.
20     La solcuion? -> Guardar LA RUTA
21     */
22
23     struct stat st;
24     char ruta[512]; //Tamaño suficiente para almacenar la ruta
25
26     while((entrada = readdir(directorio)) != NULL){
27
28         //AQUI CREAMOS LA RUTA!!!! Concatenamos nombre_dir/nombre.fichero
29         sprintf(ruta, "%s/%s", argv[1], entrada->d_name);
30
31         // Para cada fichero, vemos si es regular
32         // Por eso debemos de hacer el stat de LA RUTA
33         if(stat(ruta, &st) < 0){
34             perror("Error en el stat");
35             exit(1);
36         }
37
38         if(S_ISREG(st.st_mode)){
39             // CONCATENAMOS permisos para que no se machaquen
40             if(chmod(ruta, st.st_mode | S_IWOTH) < 0){
41                 perror("Error al dar los permisos");
42                 exit(1);
43             }
44         }
45     }
46     closedir(directorio);
47 }
48 }
```

Ejercicio 5: Programa que reciba un argumento: el nombre de un directorio. Añadir permisos de escritura para otros, y si ya los tiene: los quita.

```
1 int main(int argc, char*argv[]){
2     if(argc != 2){
3         fprintf(stderr, "Error: Numero de parametros incorrecto\n");
4         exit(2);
5     }
6
7     DIR * directorio;
8     struct dirent * entrada;
9     struct stat st;
10    char ruta[512];
11    mode_t permisos;
12
13    if((directorio = opendir(argv[1])) == NULL){
14        perror("Error al abrir el directorio");
15        exit(1);
16    }
17
18    while((entrada = readdir(directorio)) != NULL){
19
20        sprintf(ruta, "%s/%s", argv[1], entrada->d.name);
21
22        if(stat(ruta,&st) < 0){
23            perror("Error en el stat");
24            exit(1);
25        }
26
27        if((st.st_mode & S_IWOTH) == S_IWOTH){ // Tiene permisos
28            // Los quita
29            permisos = st.st_mode & (~S_IWOTH);
30        }else{
31            // Los aniaade
32            permisos = st.st_mode | S_IWOTH;
33        }
34
35        if(chmod(ruta,permisos) < 0){
36            perror("Error al dar los permisos");
37            exit(1);
38        }
39    }
40    closedir(directorio);
41 }
```

Ejercicio 6: Programa que muestre por pantalla el nombre de todos los ficheros regulares contenidos en el directorio pasado por argumento. *NOTA: Para ejecutar con el directorio padre: ./ej6 .*

```
1 int main(int argc, char *argv[]){
2     if( argc != 2){
3         fprintf (stderr, "Error. Numero de parametros incorrecto");
4         exit(2);
5     }
6
7     DIR * directorio;
8     struct dirent * entrada;
9     char ruta[512];
10    struct stat st;
11
12    if((directorio = opendir(argv[1])) == NULL){
13        perror("Error al abrir el directorio");
14        exit(1);
15    }
16
17    while((entrada = readdir(directorio)) != NULL){
18        //sprintf : en vez de mostrarlo por pantalla, lo guarda en una
19        //cadena de caracteres pasada como primer argumentos
20        sprintf(ruta,"%s/%s", argv[1], entrada->d.name);
21
22        if(stat(ruta, &st) < 0){
23            perror("Error en el stat");
24            exit(1);
25        }
26
27        if(S_ISREG(st.st.mode)){
28            printf("\n Nombre: %s\n", entrada->d.name);
29        }
30    }
31    closedir(directorio);
32 }
```

Repaso Sesión 2:

Pregunta 1: ¿Cómo se acceden a los atributos de un archivo?

```
struct stat atributos;
```

```
stat("archivo", &atributos);
```

Para ver los atributos de la estructura de forma decimal: ***printf("%d\n",file.st_mode);***

Pregunta 2: ¿Cómo podríamos implementar en C la condición de que si un fichero es regular haga algo?

```
if(S_ISREG(atributos.st_mode){...}
```

Para ver todas las macros, hay que buscar en **man inode** o **man lstat**.

Pregunta 3: ¿Cómo podríamos leer todos los elementos que hay en un directorio?

```
DIR *direct;
```

```
struct dirent *elemento_d;
```

```
while((elemento_d=readdir(direct) != NULL){  
    print("%s",elemento_d->d_name);}
```

Pregunta 4: Igual que la pregunta anterior pero, ¿y si queremos evitar que lea el directorio actual y el padre?

```
DIR *direct;
```

```
struct dirent *elemento_d;
```

```
while((elemento_d=readdir(direct) != NULL){  
    if(strcmp(elemento_d->d_name, ".") != 0 && strcmp(elemento_d->d_name, "..") != 0){  
        print("%s",elemento_d->d_name);}}
```

Usamos **strcmp** para comparar cadenas porque en C no existe el == !!

Sesión 3. Las llamadas al sistema para el Control de Procesos

Identificadores de proceso

Cada proceso tiene un *único identificador de proceso (PID)* que es un número entero no negativo.

Importante saber:

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

```
pid_t getpid(void); // devuelve el PID del proceso que la invoca.
```

```
pid_t getppid(void); // devuelve el PID del proceso padre del proceso que la invoca
```

Ejemplo con otras llamadas:

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

void main(void){
    printf("Identificador de usuario: %d\n", getuid());
    printf("Identificador de usuario efectivo: %d\n", geteuid());
    printf("Identificador de grupo: %d\n", getgid());
    printf("Identificador de grupo efectivo: %d\n", getegid());
}
```

Llamada al sistema fork

```
int main(){
    fork();
    printf("Hola :)\n"); // Aparece DOS VECES el mensaje al ejecutar este programa
}
```

fork() **CREA** otro proceso (copia casi idéntica del proceso que se está ejecutando).

Devuelve un PID.

Ejemplo:

```
int main(){
    pid_t pid;
    fork();
    printf("pid=%d , getpid()=%d , getppid()=%d\n", pid, getpid(), getppid());
}
```

El nuevo proceso que se crea tras la ejecución de la llamada fork se denomina proceso hijo. Esta llamada al sistema se ejecuta una sola vez, pero devuelve un valor distinto en cada uno de los procesos (padre e hijo).

La única diferencia entre los valores devueltos es que *en el proceso hijo el valor es 0* y en el *proceso padre* (el que ejecutó la llamada) *el valor es el PID del hijo*. La razón por la que el identificador del nuevo proceso hijo se devuelve al padre es porque un proceso puede tener más de un hijo y, de esta forma, podemos identificar los distintos hijos. De igual manera, la razón por la que fork devuelve un 0 al proceso hijo se debe a que un proceso solamente puede tener un único padre, con lo que el hijo siempre puede ejecutar getppid para obtener el PID del padre.

Un proceso finaliza y hasta que el padre no libere los recursos, el proceso hijo se queda en estado zombie.

Si el padre muere antes que el hijo, se le asigna como proceso padre (*getppid()*) el proceso 1.

Estructura básica:

```
pid_t pid;

if((pid = fork()) < 0){
    perror("error en el fork");
    exit(1);
}else if(pid == 0){
    //Soy el hijo
}else{
    //Soy el padre
}
```

Wait

wait(NULL) – Espera a que termine un hijo.

Llamada al sistema exec

Se trata de una familia de funciones.

exec – ejecuta un programa.

- **execl** -> lista |
| p -> path (para cuando buscamos cosas como ls, cat....)
- **execv** -> vector |

exec("nombre del programa", "argumentos del main", NULL);

Ejemplo: execlp("ls", "ls", "-l", NULL);

Después de un **exec** **NO SE VA A EJECUTAR NADA**, a no ser que falle. Así que debajo de escribiremos un **perror** y un **exit(1)** por si fallase.

Llamada al sistema clone

fork() se implementa a través de la llamada al sistema **clone()**, que permite crear procesos e hilos con un grado mayor en el control de sus propiedades. La sintaxis es:

```
#include <sched.h>
int clone(int (*func) (void *), void *child_stack, int flags, void *func_arg,
...
/* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
→ Retorna: si éxito, el PID del hijo; -1, si error.
```

Actividades Sesión 3:

Ejercicio 7: Programa que cree 5 hijos. Cada hijo tiene que mostrar su PID y el de su padre.

```
1 int main(){
2     pid_t pid;
3
4     for(int i = 0; i < 5; i++){
5         if((pid = fork()) < 0){
6             perror("Error en el fork");
7             exit(1);
8         } else if(pid == 0){ // HIJO
9             printf("Mi PID es: %d, y el PID de mi padre es: %d\n", getpid(),
10                  getppid());
11             // PARA QUE LOS HIJOS DEL PROCESO NO CREEN HIJOS Y EVITAR BOMBA DE
12             // FORK!!
13             exit(0);
14         }
15     }
16
17     for(int i = 0; i < 5; i++){
18         wait(NULL);
19     }
20     printf("SOY EL PAPA %d\n", getpid());
21 }
```

```
alba@albauwu:~/MisCosas/Universidad/Curso_2020_2021/Pruebas
Ejercicios$ gcc Ejercicio7.c -o ej7
alba@albauwu:~/MisCosas/Universidad/Curso_2020_2021/Pruebas$ ./ej7
Mi PID es: 48517, y el PID de mi padre es: 48515
Mi PID es: 48516, y el PID de mi padre es: 48515
Mi PID es: 48519, y el PID de mi padre es: 48515
Mi PID es: 48518, y el PID de mi padre es: 48515
Mi PID es: 48520, y el PID de mi padre es: 48515
SOY EL PAPA 48515
```

Repaso Sesión 3:

Pregunta 1: ¿Cómo podríamos comprobar que el UID que dice el programa en C es efectivamente el del usuario cuando hacemos getpid?

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

    // Obtenemos el ID del usuario.
    id_usuario = getuid();

    printf("ID del usuario: %d\n", id_usuario);
}
```

Podemos comprobarlo con el comando `id [usuario]` o consultando el fichero `/etc/passwd`, mediante:

`cat /etc/passwd | grep [usuario]`

Pregunta 2: ¿Qué pasaría si en el siguiente trozo de código el fork lo ejecutamos antes?

```
int main(int argc, char *argv[]){
    pid_t id_proceso;
    pid_t res_fork;

    id_proceso = getpid();

    res_fork = fork();

    // Error.
    if(res_fork < 0){
        perror("ERROR - %d.\n", id_proceso);
    }
    else if(res_fork != 0){
        // Padre.
        printf("Soy el padre (%d).\n", id_proceso);
        sleep(1);
    }
    else{
        // El hijo.
        printf("Soy el hijo (%d).\n", id_proceso);
    }
}
```

```
Soy el padre (2321).
Soy el hijo (2322).
```

El padre da su propio identificador, y el hijo da su propio identificador.

Si el fork NO se pone antes, ambos mostrarían lo mismo porque el id se asigna una vez, y al hacer el fork se hace una copia idéntica de lo que había antes.

```
int main(int argc, char *argv[]){
    pid_t id_proceso;
    int estado;
    pid_t res_fork;

    res_fork = fork();

    id_proceso = getpid();

    // res_fork = fork();

    // Error.
    if(res_fork < 0){
        perror("ERROR.\n");
        printf("ID: %i\n", id_proceso);
    }
    else if(res_fork != 0){
        // Padre.
        printf("Soy el padre (%d) res_fork=%i.\n",
              id_proceso, res_fork);
        sleep(1);
    }
    else{
        // El hijo.
        printf("Soy el hijo (%d) res_fork=%i.\n",
              id_proceso, res_fork);
    }
}
```

La variable `res_fork` si la imprimimos en el if del padre devolverá el pid del hijo y si la imprimimos en el if del hijo devolverá 0:

```
[ Luis: ~/Desktop/SO-P-Todos_MaterialModulo2/Sesion3 ]$ ./fork
Soy el padre (2358) res_fork=2359.
Soy el hijo (2359) res_fork=0.
```

Pregunta 3: ¿Cómo se podría implementar que un proceso padre espere su ejecución a la terminación de un proceso aleatorio?

Con `res_wait = wait(NULL)`. (`res_wait` guarda el pid del proceso hijo).

Sesión 4. Comunicación entre procesos utilizando cauces

Un **cauce** es un mecanismo para la comunicación de información y sincronización entre procesos. Los datos pueden ser *enviados (escritos)* por varios procesos al cauce, y a su vez, *recibidos (leídos)* por otros procesos desde dicho cauce.

La comunicación a través de un cauce sigue el paradigma de interacción productor/consumidor, donde típicamente existen dos tipos de procesos que se comunican mediante un **búfer**: aquellos que generan datos (productores) y los que los toman (consumidores). Estos datos se tratan en **orden FIFO (First In First Out)**.

Cauce sin nombre

Es una comunicación *HALF-DUPPLEX* (en un solo sentido).

```
int fd[2];
pipe(fd);
```

DONDE: `fd[0]` LECTURA
`fd[1]` ESCRITURA

El Pipe se tiene que ejecutar antes del Fork, ya que el hijo va a heredar los fd del padre para comunicarse.

Se tiene que cerrar el canal que no va a utilizarse:

```
int fd[2];

if(pipe(fd) < 0){
    perror("error en el pipe");
    exit(1)
}else if (pipe == 0){
    close(fd[1]); //Si el hijo va a leer CIERRA el canal de ESCRITURA.
    ...
}else{
    close(fd[0]); //Si el hijo va a escribir CIERRA el canal de LECTURA.
    ...
}
```

Cauce con nombre

Una vez creado el cauce con nombre cualquier proceso puede abrirlo para lectura y/o escritura, de la misma forma que un archivo regular. Sin embargo, el cauce debe estar abierto en ambos extremos simultáneamente antes de que podamos realizar operaciones de lectura o escritura sobre él.

Abrir un archivo FIFO para sólo lectura produce un bloqueo hasta que algún otro proceso abra el mismo cauce para escritura.

```
int mknod (const char *FILENAME, mode_t MODE, dev_t DEV)
```

La llamada al sistema **mknod** crea un archivo especial de nombre *FILENAME*. El parámetro *MODE* especifica los valores que serán almacenados en el campo *st_mode* del i-nodo correspondiente al archivo especial:

- **S_IFCHR**: representa el valor del código de tipo de archivo para un archivo de dispositivo orientado a caracteres.

- **S_IFBLK**: representa el valor del código de tipo de archivo para un archivo de dispositivo orientado a bloques.

- **S_IFSOCK**: representa el valor del código de tipo de archivo para un socket.

- **S_IFIFO**: representa el valor del código de tipo de archivo para un FIFO.

Ejemplo: mkfifo("MIFIFO", S_IRWXU);

El argumento DEV especifica a qué dispositivo se refiere el archivo especial.

Ejemplo: mknod("/tmp/FIFO", S_IFIFO|0666,0);

Los archivos FIFO se eliminan con la llamada **unlink**.

read → esta llamada es **bloqueante** para los procesos consumidores cuando no hay datos que leer en el cauce. Desbloquea devolviendo 0 (ningún byte leído) cuando todos los procesos que tenían abierto el cauce en modo escritura (los procesos que actuaban como productores), lo han cerrado o han terminado.

Llamada al sistema DUP2

dup2(descriptor del fichero a duplicar, descriptor de fichero donde lo queremos duplicar)

Ejemplo:

```
int main(){
    int fd;

    if((fd = open("Ejemplo", O_WRONLY | O_TRUNC | O_CREAT, S_IRWXU)) < 0){
        perror("error en el open");
        exit(1);
    }

    dup2(fd, STDOUT_FILENO);
    printf("duplicado\n");
}
```

"duplicado" se guarda en "Ejemplo"

Ejemplo:

```
int main(){
    int fd;

    if((fd = open("Ejemplo", O_WRONLY | O_TRUNC | O_CREAT, S_IRWXU)) < 0){
        perror("error en el open");
        exit(1);
    }

    dup2(fd, STDOUT_FILENO);
    execlp("ls", "ls", "-l", NULL);
    printf("duplicado\n");
}
```

El ls se ejecuta pero no escribe el resultado por pantalla, sino en "Ejemplo"

Actividades Sesión 4:

Ejercicio 8: Programa que cree un hijo, el cual va a leer un entero a través de teclado y se le va a enviar al padre mediante un cauce. El padre va a leer del cauce y lo va a mostrar por pantalla.

```
1 int main(){
2     pid_t pid;
3     int fd[2];
4     int valor;
5
6     if(pipe(fd) < 0){
7         perror("Error en la creacion de pipe");
8         exit(1);
9     }
10
11     if((pid = fork()) < 0){
12         perror("Error al hacer el fork");
13         exit(1);
14     }else if(pid == 0){ // HIJO
15         // Como el hijo va a escribir por el cauce, cerramos el de lectura
16         close(fd[0]);
17
18         // leer entero por teclado
19         scanf("%d", &valor);
20
21         if (write(fd[1], &valor, sizeof(int)) < 0){
22             perror("Error al enviar el valor");
23             exit(1);
24         }
25     }else{ // PADRE
26         // Como el padre va a escribir, cerramos el cauce de escritura
27         close(fd[1]);
28
29         if (read(fd[0], &valor, sizeof(int)) < 0){
30             perror("Error al recibir el valor");
31             exit(1);
32         }
33
34         printf("Leido: %d\n", valor);
35     }
36 }
37 }
```

Ejercicio 9: Proceso que cree un hijo el cual va a buscar la primera entrada del directorio actual que sea un fichero regular y va a enviar el inodo de ese fichero a través del cauce. El padre cuando lea el inodo va a mostrar por pantalla el número de inodo y el tamaño en Bytes correspondiente.

```
1 int main(){
2     int fd[2];
3     pid_t pid;
4     DIR * directorio;
5     struct dirent * entrada;
6     struct stat st;
7     ino_t inodo;
8     off_t size;
9
10    if(pipe(fd) < 0){
11        perror("Error al crear el pipe");
12        exit(1);
13    }
14
15    if((pid = fork()) < 0){
16        perror("Error al crear el fork");
17        exit(1);
18    }else if(pid == 0){
19        // Cerramos cauce de lectura
20        close(fd[0]);
21
22        if((directorio = opendir(".")) == NULL){
23            perror("error al abrir el directorio");
24            exit(1);
25        }
26
27        while((entrada = readdir(directorio)) != NULL){
28            if(stat(entrada->d_name,&st) < 0){
29                perror("Error al hacer el stat");
30                exit(1);
31            }
32
33            if(S_ISREG(st.st_mode)){
34                if(write(fd[1], &st.st_ino, sizeof(st.st_ino)) < 0){
35                    perror("Error al escribir en el pipe");
36                    exit(1);
37                }
38
39                break; // Nos quedamos con el primero que encuentre
40            }
41        }
42        closedir(directorio);
43        exit(0);
44    }else{
45        // Cerramos cauce de escritura
46        close(fd[1]);
47
48        if(read(fd[0], &inodo, sizeof(inodo)) < 0){
49            perror("Error al leer del pipe");
50            exit(1);
51        }
52
53
54        if((directorio = opendir(".")) == NULL){
55            perror("error al abrir el directorio");
56            exit(1);
57        }
58
59        while((entrada = readdir(directorio)) != NULL){
60            if(entrada->d_ino == inodo){
61                if(stat(entrada->d_name,&st) < 0){
62                    perror("Error al hacer el stat");
63                    exit(1);
64                }
65
66                size = st.st_size;
67                break;
68            }
69        }
70
71        printf("inodo: %u - tamaño: %d\n", inodo, size);
72        closedir(directorio);
73    }
74 }
```

Ejercicio 10: Proceso que obtenga el menor y mayor tamaño en bytes de las entradas del directorio actual. Creamos dos hijos: el primero que reciba desde min hasta mitad, y el segundo desde mitad + 1 hasta max. Escriben en el pipe el inodo y tamaño de las entradas comprendidas en ese rango. El padre leerá el inodo y tamaño escrito por los hijos y comprobara si dicho inodo tiene el tamaño indicado

¡¡Como varios procesos que escriben en el mismo cauce usar un struct (para mantener concurrencia)!!

Debido a la complejidad del ejercicio, lo dividiré en varias partes. Primero: la declaración de variables. Hay una estructura Info ya que a través del cauce le pasamos al padre más de un elemento y, de esta forma, lo podemos pasar todo junto. Es de gran importancia saber que el orden es importante: los datos deben leerse/escribirse en el mismo orden que son declarados.

```
1 struct Info{
2     ino_t inodo;
3     off_t size;
4 };
5
6 int main(){
7     int fd[2];
8     pid_t pid;
9     DIR * directorio;
10    struct dirent * entrada;
11    struct stat st;
12    off_t min, max, mitad, max_aux, min_aux;
13    struct Info informacion;
14    ino_t inodo;
15    off_t size;
16
17    if((directorio = opendir(".")) == NULL){
18        perror("Error al abrir el directorio");
19        exit(1);
20    }
```

Después, le damos valores a las variables max y min recorriendo las entradas del directorio:

```
1 // Para no dar valores arbitrarios iniciales a min, max
2 // le asignaremos el tamaño en Bytes de la primera entrada
3 if((entrada = readdir(directorio)) == NULL){
4     perror("Error al leer la primera entrada");
5     exit(1);
6 }
7
8 if(stat(entrada->d_name,&st) < 0){
9     perror("Error en el primer stat");
10    exit(1);
11 }
12
13 min = max = st.st_size;
14
15 while((entrada = readdir(directorio)) != NULL){
16     if(stat(entrada->d_name,&st) < 0){
17         perror("Error al hacer el stat");
18         exit(1);
19     }
20
21     if(st.st_size > max){
22         max = st.st_size;
23     }
24
25     if(st.st_size < min){
26         min = st.st_size;
27     }
28 }
29
30 mitad = (min + max)/2;
31 printf("tamaño min: %d y tamaño max: %d\n", min, max);
32 closedir(directorio);
```

Tras esto, creamos los dos hijos. El código estará escrito de manera secuencial ya que, antes de acabar el hijo, indicaremos con `exit(0)` que su ejecución finalizará si se ha hecho todo correctamente.

```

1 // PRIMER HIJO
2 if(pipe(fd) < 0){
3     perror("Error al abrir el pipe");
4     exit(1);
5 }
6
7 if((pid = fork()) < 0){
8     perror("Error al crear el hijo");
9     exit(1);
10 }else if(pid == 0){ // HIJO 1
11     close(fd[0]); // Cerramos CAUCE LECTURA
12     max_aux = mitad;
13
14     if((directorio = opendir(".")) == NULL){
15         perror("Error al abrir directorio - 1er hijo");
16         exit(1);
17     }
18
19     while((entrada = readdir(directorio)) != NULL){
20         if(stat(entrada->d_name,&st) < 0){
21             perror("Error al hacer el stat - 1er hijo");
22             exit(1);
23         }
24
25         if(st.st_size >= min && st.st_size <= max_aux){
26             informacion.inodo = st.st_ino;
27             informacion.size = st.st_size;
28
29             if(write(fd[1], &informacion, sizeof(informacion)) < 0){
30                 perror("Error al escribir - 1er hijo");
31                 exit(1);
32             }
33         }
34     }
35
36     closedir(directorio);
37     exit(0); //PARA QUE NO SIGA
38 }

```

```

1 // SEGUNDO HIJO
2 if((pid = fork()) < 0){
3     perror("Error al crear el hijo");
4     exit(1);
5 }else if(pid == 0){ // HIJO 2
6     close(fd[0]); // Cerramos CAUCE LECTURA
7     min_aux = mitad + 1;
8
9     if((directorio = opendir(".")) == NULL){
10         perror("Error al abrir directorio - 2do hijo");
11         exit(1);
12     }
13
14     while((entrada = readdir(directorio)) != NULL){
15         if(stat(entrada->d_name,&st) < 0){
16             perror("Error al hacer el stat - 2do hijo");
17             exit(1);
18         }
19
20         if(st.st_size >= min_aux && st.st_size <= max){
21             informacion.inodo = st.st_ino;
22             informacion.size = st.st_size;
23
24             if(write(fd[1], &informacion, sizeof(informacion)) < 0){
25                 perror("Error al escribir - 2er hijo");
26                 exit(1);
27             }
28         }
29     }
30
31     closedir(directorio);
32     exit(0); //PARA QUE NO SIGA
33 }

```

El padre leerá todos los inodos que le hayan mandado los dos hijos (while) y para cada uno de ellos, comprobará que dicho inodo tiene el tamaño indicado.

```

1 // CODIGO PADRE
2 close(fd[1]); // Cierra CAUCE ESCRITURA
3 //printf("inodo: %u\n", inodo);
4
5 // El padre lee cada uno de los inodos enviados por el hijo
6 // Tras esto, lee el tamaño del inodo
7 while(read(fd[0], &inodo, sizeof(inodo)) > 0){
8     if(read(fd[0], &size, sizeof(size)) < 0){
9         perror("Error al leer size");
10        exit(1);
11    }
12
13    // Realizamos comprobaciones
14    if((directorio = opendir(".")) == NULL){
15        perror("Error al abrir en el directorio - Padre");
16        exit(1);
17    }
18
19    int encontrado = 0;
20
21    while((entrada = readdir(directorio)) != NULL && (encontrado == 0)){
22        // Encuentra el inodo que le ha pasado un hijo
23        if(entrada->d_ino == inodo){
24            encontrado = 1;
25            if(stat(entrada->d_name,&st) < 0){
26                perror("Error al hacer el stat - PADRE");
27                exit(1);
28            }
29
30            if(st.st_size == size){
31                printf("CORRECTO :)\n");
32            }else{
33                // No tiene el tamaño enviado :(
34                printf("ERRADO!!! :(\n");
35            }
36        }
37    }
38
39    if(encontrado == 0){
40        printf("NO SE ENCONTRO EL INODO");
41    }
42    closedir(directorio);
43 }
44 }

```

Ejercicio 11: Proceso que cree dos hijos: el primero ejecuta `grep root /etc/passwd` y el segundo `cut -d: -f6` de forma que la salida quede: `grep root /etc/passwd | cut -d: -f6`. El padre TIENE QUE esperar a que terminen sus dos hijos.

```
1 int main(){
2     int fd[2];
3     pid_t pid;
4
5     if(pipe(fd) < 0){
6         perror("Error al abrir el cauce");
7         exit(1);
8     }
9
10    // LA IDEA ES QUE AMBOS HIJOS SE COMUNIQUEN POR EL CAUCE
11
12    if((pid = fork()) < 0){
13        perror("Error al crear el hijo");
14        exit(1);
15    } else if(pid == 0){
16        close(fd[0]); // Cerramos CAUCE LECTURA
17
18        /*
19         grep escribe por pantalla, así que con dup2
20         hacemos que lo escriba en el cauce
21        */
22        if(dup2(fd[1], STDOUT_FILENO) < 0){
23            perror("Error en el dup2");
24            exit(1);
25        }
26
27        execlp("grep", "grep", "root", "/etc/passwd", NULL);
28        perror("Error en el exec");
29        exit(1);
30    }
31
32    if((pid = fork()) < 0){
33        perror("Error al crear el hijo");
34        exit(1);
35    } else if(pid == 0){
36        // El segundo hijo LEE la primera parte de la orden
37        close(fd[1]); // Cerramos CAUCE ESCRITURA
38
39        if(dup2(fd[0], STDIN_FILENO) < 0){
40            perror("Error en el dup2");
41            exit(1);
42        }
43
44        execlp("cut", "cut", "-d:", "-f6", NULL);
45        perror("Error en el exec");
46        exit(1);
47    }
48 }
```

Código del padre:

```
1 // EL padre tiene abierto los descriptores de fichero
2 // Si no cierras, el cut se queda esperando (no finaliza)
3 // a una posible escritura.
4 // TODOS LOS PROCESOS CIERRAN EL PIPE AUNQUE NO LO USEN
5 close(fd[0]);
6 close(fd[1]);
7
8 // Espera a los DOS HIJOS
9 wait(NULL);
10 wait(NULL);
11 }
```

También podemos resolverlo con un cauce FIFO:

```
1 int main(){
2     int fd;
3     pid_t pid;
4
5     if(mkfifo("MLFIFO", S_IRWXU) < 0){
6         perror("Error al abrir el fifo");
7         exit(1);
8     }
9
10    // LA IDEA ES QUE AMBOS HIJOS SE COMUNIQUEN POR EL CAUCE
11
12    if((pid = fork()) < 0){
13        perror("Error al crear el hijo");
14        exit(1);
15    } else if (pid == 0){
16        if((fd = open("MLFIFO", O_WRONLY)) < 0){
17            perror("Error al abrir el fifo");
18            exit(1);
19        }
20
21        if(dup2(fd, STDOUT_FILENO) < 0){
22            perror("Error en el dup2");
23            exit(1);
24        }
25
26        execlp("grep", "grep", "root", "/etc/passwd", NULL);
27        perror("Error en el exec");
28        exit(1);
29    }
30
31
32    if((pid = fork()) < 0){
33        perror("Error al crear el hijo");
34        exit(1);
35    } else if (pid == 0){
36        // El segundo hijo LEE la primera parte de la orden
37        if((fd = open("MLFIFO", O_RDONLY)) < 0){
38            perror("Error al abrir el fifo");
39            exit(1);
40        }
41
42
43        if(dup2(fd, STDIN_FILENO) < 0){
44            perror("Error en el dup2");
45            exit(1);
46        }
47
48        execlp("cut", "cut", "-d:", "-f6", NULL);
49        perror("Error en el exec");
50        exit(1);
51    }
```

Código del padre:

```
1 // Espera a los DOS HIJOS
2 wait(NULL);
3 wait(NULL);
4 }
```

Repaso Sesión 4:

Pregunta 1: ¿Cuál es la diferencia entre cauce sin nombre y con nombre?

Que los cauces sin nombre son temporales (**temporalidad**) y los cauces con nombre son permanentes.

Pregunta 2: ¿Qué funciones se utilizan para crear cada uno de los dos cauces?

Cauces con nombre: `mknod`

Cauces sin nombre: `pipe`

Pregunta 3: ¿Cómo podríamos programar la eliminación de un cauce con nombre?

Con `unlink(archivo_fifo);`

Pregunta 4: ¿Cómo podríamos redireccionar la salida y la entrada estándar de un proceso?

`close(fd[0]);`

`dup2(fd[1],STDOUT_FILENO);`

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

Señales

Es el mecanismo básico de **sincronización**. Los procesos pueden enviarse señales para la notificación de cierto evento (la señal es generada cuando ocurre este evento) y, pueden determinar qué acción realizarán como respuesta a la recepción de una señal determinada.

Un **manejador de señal** es una función definida en el programa que se invoca cuando se entrega una señal al proceso. ¡ **Puede interrumpir el flujo de control** !

Se dice que una señal es depositada cuando el proceso inicia una acción en base a ella, y se dice que una señal está pendiente si ha sido generada pero todavía no ha sido depositada. Además, un proceso puede bloquear la recepción de una o varias señales a la vez.

Las señales **bloqueadas** de un proceso se almacenan en un conjunto de señales llamado máscara de bloqueo de señales. No se debe confundir una señal bloqueada con una señal ignorada, ya que una señal ignorada es desecha por el proceso, mientras que una señal bloqueada permanece pendiente y será depositada cuando el proceso la desenmascare (la desbloquee). Si una señal es recibida varias veces mientras está bloqueada, se maneja como si se hubiese recibido una sola vez.

Señales más importantes (*man 7 signal*):

Símbolo	Acción	Significado
SIGHUP	Term	Desconexión del terminal (referencia a la función <code>termio(7)</code> del <i>man</i>). También se utiliza para reanudar los demonios <code>init</code> , <code>httpd</code> e <code>inetd</code> .
		Esta señal la envía un proceso padre a un proceso hijo cuando el padre finaliza.
SIGINT	Term	Interrupción procedente del teclado (<Ctrl+C>)
SIGQUIT	Core	Terminación procedente del teclado
SIGILL	Core	Excepción producida por la ejecución de una instrucción ilegal
SIGABRT	Core	Señal de aborto procedente de la llamada al sistema <code>abort(3)</code>
SIGFPE	Core	Excepción de coma flotante
SIGKILL	Term	Señal para terminar un proceso (no se puede ignorar ni manejar).
SIGSEGV	Core	Referencia inválida a memoria
SIGPIPE	Term	Tubería rota: escritura sin lectores
SIGALRM	Term	Señal de alarma procedente de la llamada al sistema <code>alarm(2)</code>
SIGTERM	Term	Señal de terminación
SIGUSR1	Term	Señal definida por el usuario (1)
SIGUSR2	Term	Señal definida por el usuario (2)
SIGCHLD	Ign	Proceso hijo terminado o parado
SIGCONT	Cont	Reanudar el proceso si estaba parado
SIGSTOP	Stop	Parar proceso (no se puede ignorar ni manejar).
SIGTSTP	Stop	Parar la escritura en la tty
SIGTTIN	Stop	Entrada de la tty para un proceso de fondo
SIGTTOU	Stop	Salida a la tty para un proceso de fondo

Las llamadas al sistema que podemos utilizar en Linux para trabajar con *señales* son principalmente:

- **kill** – se utiliza para enviar una señal a un proceso o conjunto de procesos.

int kill (pid_t pid, int sig);

- **sigaction** – permite establecer la acción que realizará un proceso como respuesta a la recepción de una señal. Las únicas señales que no pueden cambiar su acción por defecto son *SIGKILL* y *SIGSTOP*.

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

- **sigprocmask** – se emplea para cambiar la lista de señales bloqueadas actualmente.

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

- **sigpending** – permite el examen de señales pendientes (las que se han producido mientras estaban bloqueadas).

int sigpending(sigset_t *set);

- **sigsuspend** – reemplaza temporalmente la máscara de señal para el proceso con la dada por el argumento mask y luego suspende el proceso hasta que se recibe una señal.

int sigsuspend(const sigset_t *mask);

Sigaction

Estructura básica:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

donde: **void(*sa_handler)(int);** es un **PUNTERO A FUNCIÓN**

NOTA: Manejador/Handler: función que se va a ejecutar cuando llega una señal

Ejemplo:

LAS SEÑALES SE PONEN AL PRINCIPIO DEL PROGRAMA

```
int main(){
    struct sigaction sa;
    sa.sa_handler = manejador ; // NO ponemos el (int) porque NO queremos llamarla, solo
hacer la asignación
    sa.sa_flags = 0;

    //Cada vez que llega un, por ejemplo, SIG_INT, ejecutar la función
    if(sigaction(SIG_INT, &sa, NULL) < 0){
        perror("error en el sigaction");
        exit(1);
    }

    while(1){
        sleep(1);
    }
}
```

Con *sigaction* puedo:

-**Modificar (SIG_INT, &sa, NULL);**

-**Consultar (SIG_INT, NULL, &sa);** //Rellena el struct con la información actual del SIG_INT.

Actividades Sesión 5:

Ejercicio 12: Cuando llegue la señal SIGCHLD, mostramos el pésame (RIP) por la muerte del hijo.

```
1 void manejador(int s)
2 {
3     printf("RIP");
4     wait(NULL);
5 }
6
7 int main()
8 {
9     struct sigaction nuevo_valor;
10    nuevo_valor.sa_handler = manejador;
11    nuevo_valor.sa_flags = 0;
12
13    setbuf(stdout, NULL);
14
15    if (sigaction(SIGCHLD, &nuevo_valor, NULL) < 0)
16    {
17        perror("sigaction");
18        exit(1);
19    }
20
21    while (1)
22    {
23        sleep(5);
24        if (fork() == 0)
25        {
26            printf("Soy un hijo\n");
27            sleep(1);
28            printf("Finalizo\n");
29            exit(0);
30        }
31    }
32 }
```

Repaso Sesión 5:

Pregunta 1: ¿Cómo podemos enviar a un proceso expresamente la señal que hemos bloqueado en la tarea 11 por terminal?

```
// tareall.c
#include <stdio.h>
#include <signal.h>

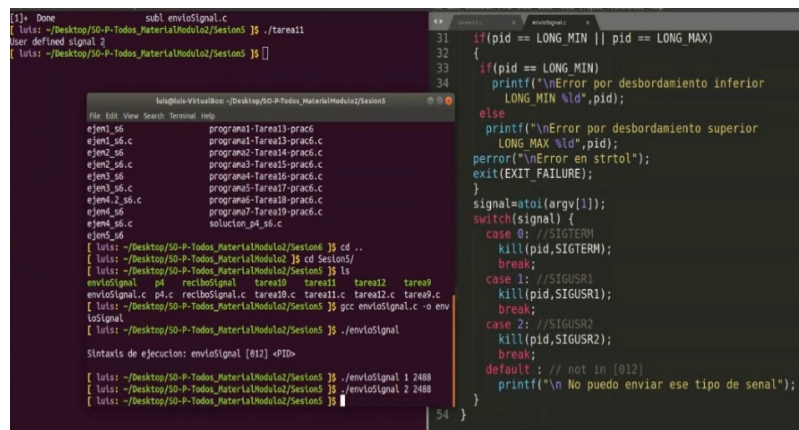
int main()
{
    sigset_t new_mask;

    /* inicializar la nueva mascara de señales */
    sigemptyset(&new_mask);
    sigaddset(&new_mask, SIGUSR1);

    /*esperar a cualquier señal excepto SIGUSR1 */
    sigsuspend(&new_mask);
}
```

kill -señal PID_proceso

En C: kill(pid,SIGUSR1);



Pregunta 2: ¿Cómo podemos implementar que al recibir una señal, simplemente muestre por pantalla que ha recibido dicha señal?

struct sigaction sig_USR_nact; //Define una acción cuando llega una señal.

sig_USR_nact.sa_handler = sig_USR_hdlr; //Le asignamos el manejador

if(sigaction(SIGUSR1,&sig_USR_nact,NULL)<0){error(...)} //Cuando se recibe la señal, se le asigna el manejador.

donde sig_USR_hdlr:

```
static void sig_USR_hdlr(int sigNum){
    if(sigNum == SIGUSR1)
        printf("\nRecibida la señal\n");
}
```

The image shows a code editor with a C program named `reciboSignal.c` and a terminal window. The code in the editor is as follows:

```
31 if(pid == LONG_MIN || pid == LONG_MAX)
32 {
33     if(pid == LONG_MIN)
34         printf("\nError por desbordamiento inferior
35             LONG_MIN %ld",pid);
36     else
37         printf("\nError por desbordamiento superior
38             LONG_MAX %ld",pid);
39     perror("\nError en strtol");
40     exit(EXIT_FAILURE);
41 }
42 signal=atoi(argv[1]);
43 switch(signal) {
44     case 0: //SIGTERM
45         kill(pid,SIGTERM);
46         break;
47     case 1: //SIGUSR1
48         kill(pid,SIGUSR1);
49         break;
50     case 2: //SIGUSR2
51         kill(pid,SIGUSR2);
52         break;
53     default: // not in [012]
54         printf("\n No puedo enviar ese tipo de senal");
55 }
```

The terminal window shows the following commands and output:

```
luis@luis-VirtualBox: ~/Desktop/50-P-Todos_MaterialModulo2/Sesiones
luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesiones $ gcc reciboSignal.c -o reciboSignal
luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesiones $ ./reciboSignal
recibida la senal SIGUSR1
recibida la senal SIGUSR1

luis@luis-VirtualBox: ~/Desktop/50-P-Todos_MaterialModulo2/Sesiones
luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesiones $ ps -e | grep reciboSigna
2596 pts/0  00:00:08 ./reciboSignal
luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesiones $ ./envioSignal 1 2596
luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesiones $ ./envioSignal 1 2596
luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesiones $
```

¡¡MUY IMPORTANTE!!

Para guardar en una variable el pid de un proceso pasado por argumento:

```
int main(){
// Código del padre:
long int pid;
int signal;
if(argc<3) {
    printf("\nSintaxis de ejecucion: envioSignal [012] <
        PID>\n\n");
    exit(EXIT_FAILURE);
}
pid= strtol(argv[2],NULL,10);
if(pid == LONG_MIN || pid == LONG_MAX)
{
    if(pid == LONG_MIN)
        printf("\nError por desbordamiento inferior
            LONG_MIN %ld",pid);
    else
        printf("\nError por desbordamiento superior
            LONG_MAX %ld",pid);
    perror("\nError en strtol");
    exit(EXIT_FAILURE);
}
signal=atoi(argv[1]);
```

Pregunta 3: ¿Cómo podríamos implementar que un proceso le mande una señal a otro proceso sin pasarle por argumento el PID del proceso al que se le va a mandar la señal?

Con un `fork` podríamos resolver esta pregunta, ya que `fork()` crea otro proceso y se puede conocer el identificador de dos procesos: del suyo con `getpid()` y del hijo, porque lo da el `fork()`. Nos encargamos de que el padre envíe la señal y que el hijo la reciba.

El problema de esto es que el padre puede acabar de ejecutar (enviar la señal) antes de que el hijo siquiera haya configurado sus cosas, entonces se queda esperando. Es esencial hacer la espera de unos segundos (`sleep(3)`) antes de mandar la señal.

Sesión 6. Control de archivos y archivos proyectados a memoria

struct flock f;

Nos interesa: - **f.l_type = F_RDLCKM o F_WRLCK o F_UNLCK;**
- **f.l_whence = SEEK_SET; (o SEEK_CUR, SEEK_END...)**
- **f.l_start = 0;** (Bloquear el fichero completo)
Indica en que Byte del fichero va a empezar el bloqueo
- **f.l_len = 0;** ("Hasta el final")
Indica cuantos bytes quiero Bloquear

Para añadir el bloqueo, usaremos la función fcntl:

int fcntl(int fd, int orden, /* argumento_orden */); // orden = operaciones a realizar

Esta función permite:

1. Consultar o ajustar las banderas de control de acceso de un descriptor.
2. Duplicar
3. Bloquear

```
if(fcntl(fd, F_GETLEK (saber si hay bloqueo) o F_SETLK)) (poner bloqueo), &f) < 0){  
    perror("error en el fcntl");  
    exit(1);  
}
```

Para saber si hay o no bloqueo:

if(f.l_type == F_UNLCK){ ... } //En este caso NO hay bloqueo

**SI HAGO UN BLOQUEO DE LECTURA, IMPIDO LA ESCRITURA EN EL FICHERO
SI HAGO UN BLOQUEO DE ESCRITURA, IMPIDO LA ESCRITURA Y LA LECTURA**

Por eso, si no queremos que nadie más vea el fichero == *bloqueo de lectura (F_RDLCK)*

La estructura básica entonces será: bloquear – código – desbloquear

```
struct flock f;  
f. .... ;  
if(fcntl...){....}  
// codigo  
f.l_type = F_UNLCK; //Desbloquear  
f. ....  
if(fcntl....){...}
```

Repaso Sesión 6:

Pregunta 1: ¿Cómo podríamos modificar las banderas del estado de un archivo? ¿Cómo podríamos hacer que se eliminara el estado que estaba puesto y quedase solamente el nuevo? (Sin borrar el resto de banderas) Por ejemplo: que la bandera O_SYNC está activada. Queremos quitarla y poner O_APPEND nada más.

a) **fcntl(fd, F_SETFL,banderas);** // setfl = set bandera

b) **banderas = fcntl(fd, F_GETFL);**

```
// 0 0 1 1 0 0 0 1 0 1 0 BANDERAS  
// 1 1 0 0 0 0 0 0 0 0 0 |  
// .....  
// 1 1 1 1 0 0 0 1 0 1 0 (Ya añadida la bandera de O_APPEND)  
// 1 1 0 0 1 1 1 1 1 1 1 ~ O_SYNC  
// ..... &  
// 1 1 0 0 0 0 0 1 0 1 0
```

```
if(banderas & O_SYNC)  
    printf("Las escrituras son  
    sincronizadas.\n");  
  
if(banderas & O_APPEND)  
    printf("Modo O_APPEND activado.\n");  
  
printf("-----\n");  
// Cambiamos una de las banderas de  
// estado.  
banderas |= O_APPEND; // ¿A qué sería  
// esto igual?  
banderas = banderas & ~O_SYNC;
```

Pregunta 2: ¿A qué equivale la línea `fcntl(fd, F_DUPFD, 1)`?

Es equivalente a duplicar el descriptor del archivo `fd` y lo devuelve con `F_DUPFD`.

Con `man stdin` encontramos que el 1 hace referencia a `STDOUT_FILENO`, redireccionando la salida estándar.

```
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ cat archivo.txt
Hola.
[1] Done      sub1 ejen5_s6.c
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ gcc ejen5_s6.c -o ejen5_s6
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ ./ejen5_s6
Hola.
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ rm archivo.txt
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ touch archivo2.txt
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ ./ejen5_s6
Fallo en fcntl: Bad file descriptor
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ ls
ejen3_s6      ejen6_s6      programa4-Tarea16-prac6.c
ej3.c        ejen1_s6.c    ejen6_s6.c    programa5-Tarea17-prac6.c
archivo2.txt ejen4_2_s6.c  Ejemplos_luis programa6-Tarea18-prac6.c
ejen1_s6     ejen4_s6     programa1-Tarea13-prac6 programa7-Tarea19-prac6.c
ejen1_s6.c   ejen4_s6.c   programa1-Tarea13-prac6.c solucion_p4_s6.c
ejen2_s6     ejen5_s6     programa2-Tarea14-prac6.c
ejen2_s6.c   ejen5_s6.c   programa2-Tarea14-prac6.c
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ touch archivo.txt
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ ./ejen5_s6
Hola.
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ cat archivo.txt
Hola de prueba.
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ gcc ejen5_s6.c -o ejen5_s6
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ ./ejen5_s6
Hola de prueba.
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $ cat archivo.txt
Hola de prueba.
[luis: ~/Desktop/50-P-Todos_MaterialModulo2/Sesion6] $
```

```
8 int main (int argc, char * argv[]) {
9     const char ARCHIVO[] = "archivo.txt";
10    int fd;
11    char buffer[] = "Hola de prueba.\n";
12    int cont;
13
14    // Abrimos el archivo.
15    fd = open(ARCHIVO, O_WRONLY);
16
17    // Cerramos la salida estandar.
18    close(1); //STDIN...
19    // Redireccionamos la salida estandar al archivo.
20    if (fcntl(fd, F_DUPFD, 1) == -1 )
21        perror ("Fallo en fcntl");
22
23    cont = write(1, buffer, strlen(buffer));
24
25    close(fd);
26 }
```

Pregunta 3: ¿Cómo podríamos asegurar que se está bloqueando un archivo para escritura? Si el archivo estuviera bloqueado, ¿cómo podríamos obtener el PID del proceso que lo bloquea?

- a) Con el `struct flock` → `l_type = F_WRLCK` // Variable de cerrojos.
- b) Con el `struct flock` → `l_pid`

Sesión 7. Construcción de un Spool de impresión

• Esquema general:

