

# Tema 1: Programación multiproceso en C

1. INTRODUCCIÓN .....	1
2. PROCESOS Y SISTEMA OPERATIVO .....	1
2.1. Estados de un proceso .....	3
2.2. Control de procesos en Linux .....	5
2.2.1. Creación y ejecución de procesos .....	7
2.2.2. Comunicación entre procesos .....	13
2.2.3. Sincronización entre procesos .....	26

## 1. INTRODUCCIÓN

Todos los ordenadores actuales realizan varias tareas a la vez, por ejemplo, ejecutar un programa de procesador de textos, leer información de un disco duro, imprimir un documento por la impresora, visualizar información en pantalla, etc.

Cuando un programa se carga en la memoria para su ejecución se convierte en un proceso. En un sistema operativo **multiproceso** o **multitarea** se puede ejecutar más de un proceso (programa) a la vez, dando la sensación al usuario de que cada proceso es el único que se está ejecutando. La única forma de ejecutar varios procesos simultáneamente es tener varias CPUs (ya sea en una máquina o en varias). En los sistemas operativos con una única CPU se va alternando la ejecución de los procesos, es decir, se quita un proceso de la CPU, se ejecuta otro y se vuelve a colocar el primero sin que se entere de nada; esta operación se realiza tan rápido que parece que cada proceso tiene dedicación exclusiva.

La programación multiproceso tiene en cuenta la posibilidad de que múltiples procesos puedan estar ejecutándose simultáneamente sobre el mismo código de programa. Es decir, desde una misma aplicación podemos realizar varias tareas de forma simultánea, o lo que es lo mismo, podemos dividir un proceso en varios subprocesos.

## 2. PROCESOS Y SISTEMA OPERATIVO

Se puede definir un proceso como un programa en ejecución. Consiste básicamente en el código ejecutable del programa, los datos y la pila del programa, el contador de programa, el puntero de pila y otros registros, y toda la información necesaria para ejecutar el programa.

Todos los programas que se ejecutan en el ordenador se organizan como un conjunto de procesos. El sistema operativo decide parar la ejecución de un proceso, por ejemplo, porque ha consumido su tiempo de CPU, y arrancar la de otro. Cuando se suspende temporalmente la ejecución de un proceso debe

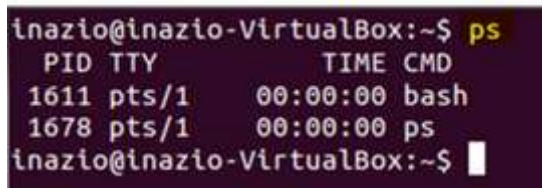
## Tema 1 PSP

rearrancarse posteriormente en el mismo estado en que se encontraba cuando se paró, esto implica que toda la información referente al proceso debe almacenarse en alguna parte.

El **BCP** es una estructura de datos llamada Bloque de Control de Proceso donde se almacena información acerca de un proceso:

- Identificación del proceso. Cada proceso que se inicia es referenciado por un identificador único. Estado del proceso.
- Contador de programa.
- Registros de CPU.
- Información de planificación de CPU como la prioridad del proceso.
- Información de gestión de memoria.
- Información contable como la cantidad de tiempo de CPU y tiempo real consumido.
- Información de estado de E/S como la lista de dispositivos asignados, archivos abiertos, etc.

Mediante el comando **ps** (*process status*) de Linux podemos ver parte de la información asociada a cada proceso. El siguiente ejemplo muestra los procesos actualmente vivos en la máquina, se muestran 2 procesos ejecutándose, uno es el shell y el otro es la ejecución de la orden ps:



```
inazio@inazio-VirtualBox:~$ ps
  PID TTY          TIME CMD
 1611 pts/1        00:00:00 bash
 1678 pts/1        00:00:00 ps
inazio@inazio-VirtualBox:~$
```

- **PID**: identificador del proceso
- **TTY**: terminal asociado del que lee y al que escribe. Si no hay aparece interrogación
- **TIME**: tiempo de ejecución asociado, es la cantidad total de tiempo de CPU que el proceso ha utilizado desde que nació.
- **CMD**: nombre del proceso.

La orden **ps -af** muestra más información:

```
lnazio@lnazio-VirtualBox:~$ ps -af
```

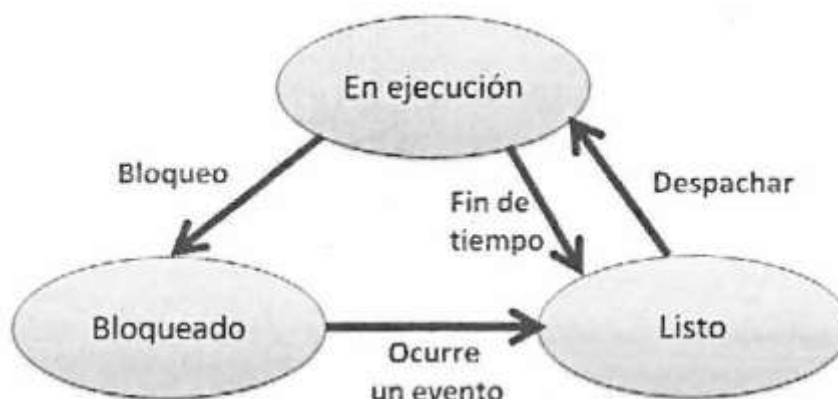
UID	PID	PPID	C	SZ	RSS	PSR	STIME	TTY	TIME	CMD
root	1	0	0	885	1980	0	00:30	?	00:00:00	/sbin/init
root	2	0	0	0	0	0	00:30	?	00:00:00	[kthreadd]
root	3	2	0	0	0	0	00:30	?	00:00:00	[ksoftirqd/0]
root	5	2	0	0	0	0	00:30	?	00:00:00	[kworker/0:0H]
root	7	2	0	0	0	0	00:30	?	00:00:00	[migration/0]
root	8	2	0	0	0	0	00:30	?	00:00:00	[rcu_bh]
root	9	2	0	0	0	0	00:30	?	00:00:00	[rcu_sched]
root	10	2	0	0	0	0	00:30	?	00:00:00	[watchdog/0]
root	11	2	0	0	0	0	00:30	?	00:00:00	[khelper]
root	12	2	0	0	0	0	00:30	?	00:00:00	[kdevtmpfs]
root	13	2	0	0	0	0	00:30	?	00:00:00	[netns]
root	14	2	0	0	0	0	00:30	?	00:00:00	[writeback]
root	15	2	0	0	0	0	00:30	?	00:00:00	[kintegrityd]
root	16	2	0	0	0	0	00:30	?	00:00:00	[bioset]
root	17	2	0	0	0	0	00:30	?	00:00:00	[crypto]
root	18	2	0	0	0	0	00:30	?	00:00:00	[kworker/u3:0]
root	19	2	0	0	0	0	00:30	?	00:00:00	[kblockd]
root	20	2	0	0	0	0	00:30	?	00:00:00	[ata_sff]
root	21	2	0	0	0	0	00:30	?	00:00:00	[khubd]
root	22	2	0	0	0	0	00:30	?	00:00:00	[nd]
root	23	2	0	0	0	0	00:30	?	00:00:00	[devfreq_wq]
root	24	2	0	0	0	0	00:30	?	00:00:00	[kworker/0:1]
root	26	2	0	0	0	0	00:30	?	00:00:00	[khungtaskd]

## 2.1. Estados de un proceso

Un proceso, aunque es una entidad independiente puede generar una salida que se use como entrada para otro proceso. Entonces este segundo proceso tendrá que esperar a que el primero termine para obtener los datos a procesar, en este caso debe bloquearse hasta que sus datos de entrada estén disponibles.

Un proceso también se puede parar porque el sistema operativo decida asignar el procesador a otro proceso. En definitiva, los estados en los que se pueden encontrar un proceso son los siguientes:

- **En ejecución:** el proceso está actualmente ejecutándose, es decir, usando el procesador.
- **Bloqueado:** el proceso no puede hacer nada hasta que no ocurra un evento externo, como por ejemplo la finalización de una operación de E/S.
- **Listo:** el proceso está parado temporalmente y listo para ejecutarse cuando se le dé oportunidad.



## Tema 1 PSP

Las transiciones entre los estados son las siguientes:

- **En ejecución - Bloqueado:** un proceso pasa de ejecución a bloqueado cuando espera la ocurrencia de un evento externo.
- **Bloqueado - Listo:** un proceso pasa de bloqueado a listo cuando ocurre el evento externo que se esperaba.
- **Listo - En ejecución:** un proceso pasa de listo a ejecución cuando el sistema le otorga un tiempo de CPU.
- **En ejecución - Listo:** un proceso pasa de ejecución a listo cuando se le acaba el tiempo asignado por el sistema operativo.

## 2.2. Control de procesos en Linux

Seguro que más de una vez hemos necesitado dentro de un programa ejecutar otro programa que realice alguna tarea concreta. Linux ofrece varias funciones para realizar esto: **system()**, **fork()** y **exec()**

La función **system()** se encuentra en la librería estándar *stdlib.h* por lo que funciona en cualquier sistema operativo que tenga un compilador de C/C++ como por ejemplo Linux, Windows, etc. El formato es el siguiente:

```
int system(const char *cadena)
```

La función recibe como parámetro una cadena de caracteres que indica el comando que se desea procesar. Dicha instrucción es pasada al intérprete de comandos del ambiente en el que se esté trabajando y se ejecuta. Devuelve el valor -1 si ocurre un error y el estado devuelto por el comando en caso contrario. La ejecución del siguiente ejemplo en C lista el contenido del directorio actual y lo envía a un fichero, abre el editor *gedit* con el fichero generado y ejecuta un comando que no existe en el intérprete de comandos de Linux:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    printf("Ejemplo de uso de system():");
    printf("\n\tListado del directorio actual y envio a un fichero:");
    printf("%d",system("ls > ficsalida"));
    printf("\n\tAbrimos con el gedit el fichero...");
    printf("%d",system("gedit ficsalida"));
    printf("\n\tEste comando es erróneo: %d",system("ged"));
    printf("\nFin de programa....\n");
}
```

Lo compilamos y lo ejecutamos desde Linux:

```
mj@ubuntu-mj:~$ gcc ejemploSystem.c -o ejemploSystem
mj@ubuntu-mj:~$ ./ejemploSystem
Ejemplo de uso de system():
    Listado del directorio actual y envio a un fichero:0
sh: ged: not found
    Abrimos con el gedit el fichero...0
    Este comando es erróneo: 32512
Fin de programa....
mj@ubuntu-mj:~$
```

Esta función no se debe usar desde un programa con privilegios de administrador porque pudiera ser que se emplearan valores extraños para algunas variables de entorno y podrían comprometer la integridad del sistema. En este caso se utiliza **execl()**.

## Tema 1 PSP

La función **execl()** tiene otras 5 funciones relacionadas (que no se tratarán en el tema). Realiza la ejecución y terminación del comando. Su formato es:

```
#include <unistd.h>

int execl(const char *fichero, const char *arg0, ...,
          char *argn, (char *)NULL);
```

La función recibe el nombre del fichero que se va a ejecutar y luego los argumentos terminando con un puntero nulo. Devuelve -1 si ocurre algún error y en la variable global *errno* se pondrá el código de error adecuado. Por ejemplo, para ejecutar el comando `/bin/ls -l` escribimos:

```
#include <unistd.h>
#include <stdio.h>
void main()

{
    printf("Ejemplo de uso de exec():");
    printf("Los archivos en el directorio son:\n");
    execl("/bin/ls", "ls", "-l", (char *)NULL);
    printf("!!! Esto no se ejecuta !!!\n");
}
```

Compilamos y ejecutamos:

```
mj@ubuntu-mj:~$ gcc ejemploExec.c -o ejemploExec
mj@ubuntu-mj:~$ ./ejemploExec
Ejemplo de uso de exec():Los archivos en el directorio son:
total 176
-rwxr-xr-x  1 mj    mj      7259 2012-11-15 17:17 a.out
-rw-r--r--  1 root root  59105 2011-10-11 16:07 core.img
drwxr-xr-x  2 mj    mj      4096 2012-11-14 22:24 Descargas
drwxr-xr-x  2 mj    mj      4096 2012-09-19 20:29 Documentos
drwxrwsr-x  9 mj    mj      4096 2012-11-15 00:36 eclipse
-rw-r--r--  1 mj    mj       919 2012-11-15 01:39 ejemplo.c
-rwxr-xr-x  1 mj    mj      7256 2012-11-15 18:44 ejemploExec
-rwxr-xr-x  1 mj    mj      1110 2012-11-15 18:43 ejemploExec.c
-rwxr-xr-x  1 mj    mj      7259 2012-11-15 17:45 ejemploSystem
. . . . .
```

### 2.2.1. Creación y ejecución de procesos

Hasta ahora hemos visto funciones que ejecutaban comandos ya sea del intérprete de comandos o de ficheros en disco, a continuación, veremos una función cuya misión es crear un proceso. Se trata de la función **fork()**. Su sintaxis es:

```
#include <unistd.h>

pid_t fork(void);
```

Al llamar a esta función se crea un nuevo proceso (proceso hijo) que es una copia exacta en código y datos del proceso que ha realizado la llamada (el proceso padre), salvo el PID y la memoria que ocupa. Las variables del proceso hijo son una copia de las del padre, por lo que modificar una variable en uno de los procesos no se refleja en el otro (ya que tienen distintas memorias).

El valor devuelto por **fork()** es un valor numérico:

- Devuelve -1 si se produce algún error en la ejecución.
- Devuelve 0 si no se produce ningún error y nos encontramos en el proceso hijo.
- Devuelve el PID asignado al proceso hijo si no se produce ningún error y nos encontramos en el proceso padre.

Antes de hacer un ejemplo con la función **fork()** vamos a ver cómo obtener el identificador de un proceso o PID. Para ello usamos 2 funciones que devuelven un tipo pid\_t. Las funciones son las siguientes:

```
pid_t getpid(void);
```

Devuelve el identificador del proceso que realiza la llamada, es decir, del proceso actual.

```
pid_t getppid(void);
```

Devuelve el identificador del proceso padre del proceso actual.

## Tema 1 PSP

Ejemplo:

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

void main(void)
{
    pid_t id_pactual, id_padre;

    id_pactual = getpid();
    id_padre = getppid();

    printf("PID de este proceso: %d\n", id_pactual);
    printf("PID del proceso padre: %d\n", id_padre);
}
```

Lo compilamos y ejecutamos. Se visualiza una salida similar a:

```
mj@ubuntu-mj:~$ gcc ejemploPadres.c -o ejemploPadres
mj@ubuntu-mj:~$ ./ejemploPadres
PID de este proceso: 2833
PID del proceso padre: 1923
```

Si ejecutamos el comando `ps` para ver los procesos que se están ejecutando, podemos ver que el PID del shell de Ubuntu (1923) coincide con el padre del proceso ejecutado anteriormente:

```
mj@ubuntu-mj:~$ ps
  PID TTY          TIME CMD
 1923 pts/0        00:00 bash
 2834 pts/0        00:00 ps
```



A continuación, vamos a ver un ejemplo donde el proceso actual (proceso padre) crea un proceso (proceso hijo) con la función **fork()**:

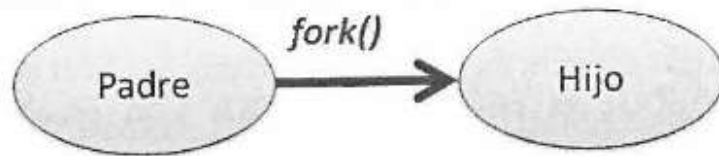


Figura 1.4. Proceso padre crea un hijo.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
void main() {
    pid_t pid, Hijo_pid;
    pid = fork();

    if (pid == -1 ) //Ha ocurrido un error
    {
        printf("No se ha podido crear el proceso hijo...");

        exit(-1);
    }
    if (pid == 0 ) //Nos encontramos en Proceso hijo
    {
        printf("Soy el proceso hijo \n\t
            Mi PID es %d, El PID de mi padre es: %d.\n",
            getpid(), getppid() );
    }
    else //Nos encontramos en Proceso padre
    {
        Hijo_pid = wait(NULL); //espera la finalización del proceso hijo
        printf("Soy el proceso padre:\n\t
            Mi PID es %d, El PID de mi padre es: %d.\n\t
            Mi hijo: %d terminó.\n",
            getpid(), getppid(), pid);
    }
    exit(0);
}
```

*Handwritten notes:*

- Next to the child's printf: *4 Hijo*
- Next to the parent's printf: *1 Padre*
- At the bottom right, a small tree diagram showing a parent process (P) with a child process (C) and a grandchild process (G).

En el código anterior se utiliza la función **wait()** para que el proceso padre espere la finalización del proceso hijo, el proceso padre quedará bloqueado hasta que termine el hijo. La sintaxis de la orden es la siguiente:

```
pid_t wait(int *status);
```

Devuelve el identificador del proceso hijo cuya ejecución ha finalizado. La sentencia **wait(NULL)** es la forma más básica de esperar a que un hijo termine.

Partiendo del ejemplo anterior, creamos un nuevo proceso en el proceso hijo; así tendremos el proceso padre (ABUELO), el proceso hijo (HIJO) y el proceso hijo del hijo (NIETO):



## Tema 1 PSP

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
//ABUELO-HIJO-NIETO
void main() {
    pid_t pid, Hijo_pid, pid2, Hijo2_pid;

    pid = fork(); //Soy el Abuelo, creo a Hijo

    if (pid == -1) //Ha ocurrido un error
    {
        printf("No se ha podido crear el proceso hijo...");
        exit(-1);
    }

    if (pid == 0) //Nos encontramos en Proceso hijo
    {
        pid2 = fork(); //Soy el Hijo, creo a Nieto
        switch(pid2)
        {
            case -1: // error
                printf("No se ha podido crear el proceso hijo
                        en el HIJO...");
                exit(-1);
                break;
            case 0: // proceso hijo
                printf("\t\tSoy el proceso NIETO %d; Mi padre es = %d \n",
                    getpid(), getppid());
                break;
            default: // proceso padre
                Hijo2_pid=wait(NULL);
                printf("\tSoy el proceso HIJO %d, Mi padre es: %d.\n",
                    getpid(), getppid());
                printf("\tMi hijo: %d terminó.\n", Hijo2_pid);
        }
    }

    else //Nos encontramos en Proceso padre
    {
        Hijo_pid = wait(NULL); //espera la finalización del proceso hijo
        printf("Soy el proceso ABUELO: %d, Mi HIJO: %d terminó.\n",
            getpid(), pid);
    }
    exit(0);
}
```

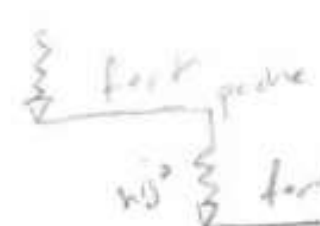


Diagram illustrating process creation:

- Root process (lightning bolt) forks to create a 'padre' process.
- The 'padre' process forks to create a 'hijo' process.
- The 'hijo' process forks to create another 'hijo' process.

## Tema 1 PSP

### Compilación y salida:

```
mj@ubuntu-mj:~$ gcc ejemplo1_2Fork.c -o ejemplo1_2Fork
mj@ubuntu-mj:~$ ./ejemplo1_2Fork
    Soy el proceso NIETO 4486; Mi padre es = 4485
    Soy el proceso HIJO 4485, Mi padre es: 4484.
    Mi hijo: 4486 terminó.
    Soy el proceso ABUELO: 4484, Mi HIJO: 4485 terminó.
mj@ubuntu-mj:~$
```

---

#### ACTIVIDAD 1.1

Realiza un programa en C que cree un proceso (tendremos 2 procesos uno padre y otro hijo). El programa definirá una variable entera y le dará el valor 6. El proceso padre incrementará dicho valor en 5 y el hijo restará 5. Se deben mostrar los valores en pantalla. A continuación se muestra un ejemplo de la ejecución:

```
mj@ubuntu-mj:~$ gcc actividad1_1.c -o actividad1_1
mj@ubuntu-mj:~$ ./actividad1_1
Valor inicial de la variable: 6
Variable en Proceso Hijo: 1
Variable en Proceso Padre: 11
mj@ubuntu-mj:~$
```

---

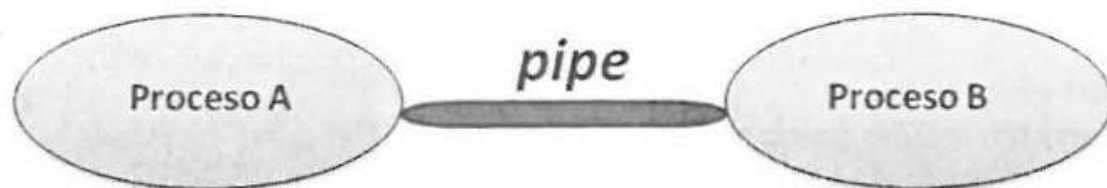
### 2.2.2. Comunicación entre procesos

Existen varias formas de **comunicación entre procesos** (Inter-Process Communication o IPC) de Linux: pipes, colas de mensajes, semáforos y segmentos de memoria compartida.

En este tema trataremos los mecanismos más sencillos, los pipes (tuberías en castellano).

#### PIPES SIN NOMBRE

Un pipe es una especie de falso fichero que sirve para conectar dos procesos. Si el proceso A quiere enviar datos al proceso B, los escribe en el pipe como si este fuera un fichero de salida. El proceso B puede leer los datos sin más que leer el pipe como si se tratara de un fichero de entrada. Así la comunicación entre procesos es parecida a la lectura y escritura en ficheros normales.



Cuando un proceso quiere leer del *pipe* y este está vacío, tendrá que esperar (es decir, se bloqueará) hasta que algún otro proceso ponga datos en él.

Igualmente cuando un proceso intenta escribir en el *pipe* y está lleno se bloqueará hasta que se vacíe.

El pipe es bidireccional pero, cada proceso lo utiliza en una única dirección, es este caso, el kernel gestiona la sincronización.

Para crear un *pipe* se realiza una llamada a la función **pipe()**:

```
#include <unistd.h>

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

Esta función recibe un solo argumento, que es un array de dos enteros: `fd[0]` contiene el descriptor para lectura y `fd[1]` el de escritura.

Si la función tiene éxito devuelve 0 y el array contendrá dos nuevos descriptors de archivos para ser usados por la tubería.

Si ocurre algún error devuelve -1.

## Tema 1 PSP

Para enviar datos al pipe, se usa la función **write()**, y para recuperar datos del *pipe*, se usa la función **read()**. La sintaxis es la siguiente:

```
int read( int fd, void *buf, int count );  
int write( int fd, void *buf, int count );
```

**read()** intenta leer **count** bytes del descriptor de fichero definido en **fd**, para guardarlos en el buffer **buf**. Devuelve el número de bytes leídos; si comparamos este valor con la variable **count** podemos saber si ha conseguido leer tantos bytes como se pedían.

La llamada al sistema **write** hace que los primeros **bytes** del **buffer** sean escritos en el archivo asociado con el descriptor de archivos **fichero**. Envía el número de **bytes** escritos realmente. Puede ser menor que **bytes** si ha habido un error en el descriptor de archivos o si el controlador del dispositivo subyacente es sensible al tamaño del bloque. Si la función retorna **0**, significa que no se han escrito datos. Si envía **-1**, ha habido un error en la llamada **write**, y el error se especificará en la variable global **errno**.

Para cerrar el fichero usamos **close()** indicando entre paréntesis el descriptor de fichero a cerrar. Se parte de la existencia de un fichero vacío de nombre **texto.txt**, el programa abre el fichero para escritura, escribe un saludo y después cierra el fichero. Posteriormente vuelve a abrir el fichero en modo lectura y hace un recorrido leyendo los bytes de uno en uno. Al finalizar la lectura se cierra el fichero.

## Tema 1 PSP

El programa es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
    char saludo[] = "Un saludo!!!\n";
    char buffer[10];
    int fd, bytesleidos;

    fd=open("texto.txt",1);//fichero se abre solo para escritura

    if( fd == -1 )
    {
        printf("ERROR AL ABRIR EL FICHERO...\n");
        exit(-1);
    }

    printf("Escribo el saludo...\n");
    write(fd,saludo, strlen(saludo));
    close(fd); //cierro el fichero

    fd=open("texto.txt",0);//el fichero se abre solo para lectura
    printf("Contenido del Fichero: \n");

    //leo bytes de uno en uno y lo guardo en buffer
    bytesleidos= read(fd, buffer, 1);
    while (bytesleidos!=0){
        printf("%s", buffer); //pinto el byte leido
        bytesleidos= read(fd, buffer, 1);//leo otro byte
    }
    close(fd);
}
```

*¡ Se guarda en buffer el !*

La compilación y ejecución muestra la siguiente salida:

```
administrador@ubuntu1:~$ gcc ejemWriteRead.c -o ejemWriteRead
administrador@ubuntu1:~$ ./ejemWriteRead
Escribo el saludo...
Contenido del Fichero:
Un saludo!!!
administrador@ubuntu1:~$
```

## Tema 1 PSP

Envío de un mensaje del hijo al padre haciendo uso de un pipe:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    int fd[2];
    char buffer[30];
    pid_t pid;

    pipe(fd); //se crea el pipe

    pid=fork(); //se crea el proceso hijo

    switch(pid) {
        case -1 : //ERROR
            printf("NO SE HA PODIDO CREAR HIJO...");
            exit(-1);
            break;
        case 0 : //HIJO
            printf("El HIJO escribe en el pipe...\n");
            write(fd[1], "Hola papi", 10);
            break;
        default : //PADRE
            wait(NULL); //espera que finalice proceso hijo
            printf("El PADRE lee del pipe...\n");
            read(fd[0], buffer, 10);
            printf("\tMensaje leído: %s\n",buffer);
            break;
    }
}
```

*Handwritten notes:*  
fd[0] -> descriptor de lectura  
fd[1] -> descriptor de escritura  
return

La compilación y ejecución muestra la siguiente salida:

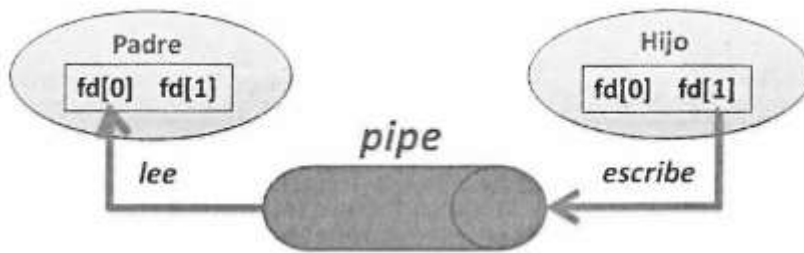
```
administrador@ubuntu1:~$ gcc ejemploPipe1.c -o ejemploPipe1
administrador@ubuntu1:~$ ./ejemploPipe1
El HIJO escribe en el pipe...
El PADRE lee del pipe...
    Mensaje leído: Hola papi
administrador@ubuntu1:~$
```

Primero se crea la tubería con `pipe()` y a continuación el proceso hijo. Recordemos que cuando se crea un proceso hijo con `fork()`, recibe una copia de todos los descriptores de ficheros del proceso padre, incluyendo copia de los descriptores de ficheros del pipe (`fd[0]` y `fd[1]`).



## Tema 1 PSP

Esto permite que el proceso hijo mande datos al extremo de escritura del pipe fd[1], y el padre los reciba del extremo de lectura fd[0],



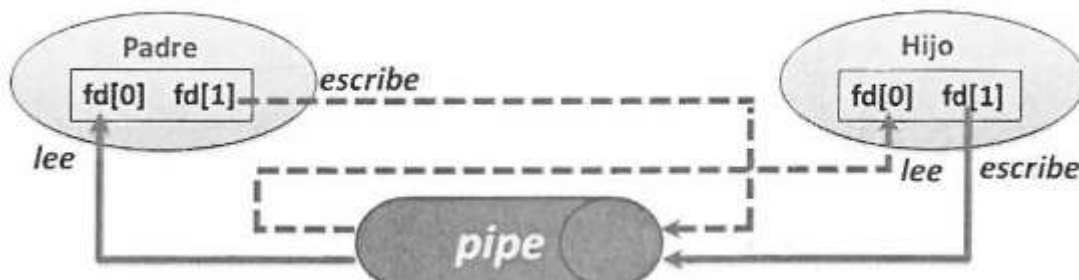
Los procesos padre e hijo están unidos por el pipe, pero la comunicación es en una única dirección, por tanto se debe decidir en qué dirección se envía la información, del padre al hijo o del hijo al padre; y dado que los descriptors se comparten siempre, debemos estar seguros de cerrar el extremo que no nos interesa.

Cuando el flujo de información va del padre hacia el hijo:

- El padre debe cerrar el descriptor de lectura fd[0].
- El hijo debe cerrar el descriptor de escritura fd[1].

Cuando el flujo de información va del hijo hacia padre ocurre lo contrario:

- El padre debe cerrar el descriptor de escritura fd[1].
- El hijo debe cerrar el descriptor de lectura fd[0].



## Tema 1 PSP

El siguiente ejemplo crea un pipe en el que el padre envía un mensaje al hijo, el flujo de la información va del padre al hijo, el padre debe cerrar el descriptor fd[0] y el hijo fd[1]; el padre escribe en fd[1] y el hijo lee de fd[0]:

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

int main(void)
{
    int fd[2];
    pid_t pid;
    char saludoPadre[]="Buenos dias hijo.\0";
    char buffer[80];

    pipe(fd); //creo pipe
    pid=fork(); //creo proceso

    switch(pid) {
        case -1 : //ERROR
            printf("NO SE HA PODIDO CREAR HIJO...");
            exit(-1);

        case 0 : //HIJO RECIBE
            close(fd[1]); //cierra el descriptor de entrada
            read(fd[0], buffer, sizeof(buffer)); //leo el pipe
            printf("\tEl HIJO recibe algo del pipe: %s\n",buffer);
            break;

        default : //PADRE ENVIA
            close(fd[0]);
            write(fd[1],saludoPadre,strlen(saludoPadre)); //escribo en pipe
            printf("El PADRE ENVIA MENSAJE AL HIJO...\n");
            wait(NULL); //espero al proceso hijo
            break;
    }
    return 0;
}
```

La compilación y ejecución muestra la siguiente salida:

```
administrador@ubuntu1:~$ gcc ejemploPipe3.c -o ejemploPipe3
administrador@ubuntu1:~$ ./ejemploPipe3
El PADRE ENVIA MENSAJE AL HIJO...
    El HIJO recibe algo del pipe: Buenos días hijo.
```

---

### ACTIVIDAD 1.2

Siguiendo el ejemplo anterior, realiza un programa en C que cree un pipe en el que el hijo envíe un mensaje al padre, es decir, la información fluya del hijo al padre. La ejecución debe mostrar la siguiente salida:

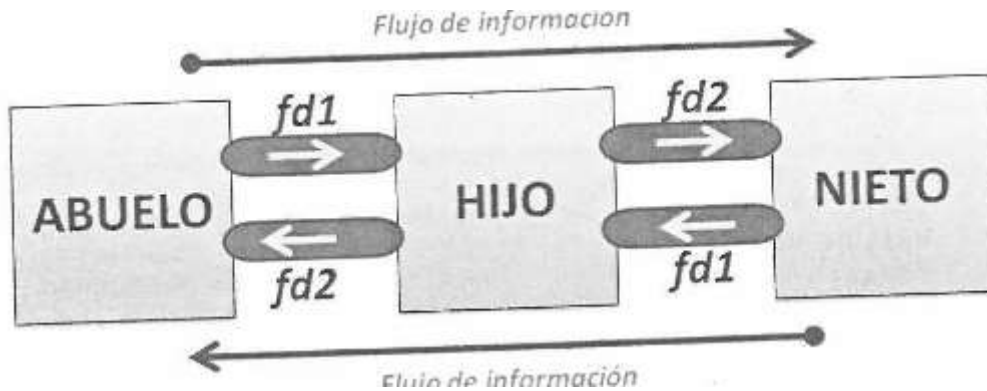
```
administrador@ubuntu1:~$ gcc actividad1_2.c -o actividad1_2
administrador@ubuntu1:~$ ./actividad1_2
    El HIJO envia algo al pipe.
El PADRE recibe algo del pipe: Buenos dias padre.
administrador@ubuntu1:~$
```

---

## Tema 1 PSP

En el siguiente ejemplo vamos a hacer que padres e hijos puedan enviar y recibir información, como la comunicación es en un único sentido crearemos dos pipes `fd1` y `fd2`.

Cada proceso usará un pipe para enviar la información y otro para recibirla. Partimos de los procesos ABUELO, HIJO y NIETO, la comunicación entre ellos se muestra en la Figura:



- El ABUELO usará el `fd1` para enviar información al HIJO y recibirá la información de este a través del `fd2`.
- El HIJO usará el `fd2` para enviar información al NIETO y recibirá la información de este a través del `fd1`.
- El NIETO usará el `fd1` para enviar información al HIJO (su padre) y recibirá la información de este a través del `fd2`.

## Tema 1 PSP

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
//ABUELO-HIJO-NIETO
void main() {
    pid_t pid, Hijo_pid, pid2, Hijo2_pid;

    int fd1[2];
    int fd2[2];

    char saludoAbuelo[]="Saludos del Abuelo.\0";
    char saludoPadre[]="Saludos del Padre.\0";
    char saludoHijo[]="Saludos del Hijo...\0";
    char saludoNieto[]="Saludos del Nieto...\0";

    char buffer[80]="";

    pipe(fd1);
    pipe(fd2);
    pid = fork(); //Soy el Abuelo, creo a Hijo

    if (pid == -1) //Ha ocurrido un error
    {
        printf("No se ha podido crear el proceso hijo...");
        exit(-1);
    }

    if (pid == 0) //Nos encontramos en Proceso hijo
    {
        pid2 = fork(); //Soy el Hijo, creo a Nieto
        switch(pid2)
        {
            case -1: // error
                printf("No se ha podido crear el proceso hijo en el HIJO.");
                exit(-1);
                break;
            case 0: // proceso hijo (nieto)
                //NIETO RECIBE
                close(fd2[1]); //cierra el descriptor de entrada
                read(fd2[0], buffer, sizeof(buffer)); //leo el pipe
                printf("\t\tNIETO RECIBE mensaje de su padre:
                    %s\n", buffer);

                //NIETO ENVIA
                printf("\t\tNIETO ENVIA MENSAJE a su padre...\n");
                close(fd1[0]);
                write(fd1[1], saludoNieto, strlen(saludoNieto));
                break;
            default: // proceso padre (hijo)
                //HIJO RECIBE
                close(fd1[1]); //cierra el descriptor de entrada
                read(fd1[0], buffer, sizeof(buffer)); //leo el pipe
                printf("\tHIJO recibe mensaje de ABUELO: %s\n", buffer);

                //HIJO ENVIA a su hijo
                close(fd2[0]);
                write(fd2[1], saludoPadre, strlen(saludoPadre));
                Hijo2_pid = wait(NULL); //espero al hijo

                //RECIBE de su hijo
                close(fd1[1]); //cierra el descriptor de entrada
                read(fd1[0], buffer, sizeof(buffer)); //leo el pipe
                printf("\tHIJO RECIBE mensaje de su hijo: %s\n", buffer);

                //HIJO ENVIA a su PADRE
                printf("\tHIJO ENVIA MENSAJE a su padre...\n", buffer);
                close(fd2[0]);
                write(fd2[1], saludoHijo, strlen(saludoHijo));
        }
    }
    else //Nos encontramos en Proceso padre (abuelo)
    {
        //PADRE ENVIA
        printf("ABUELO ENVIA MENSAJE AL HIJO...\n");
        close(fd1[0]);
        write(fd1[1], saludoAbuelo, strlen(saludoAbuelo)); //escribo
        Hijo_pid = wait(NULL); //espera la finalización del hijo

        //PADRE RECIBE
        close(fd2[1]); //cierra el descriptor de entrada
        read(fd2[0], buffer, sizeof(buffer)); //leo el pipe
        printf("El ABUELO RECIBE MENSAJE del HIJO: %s\n", buffer);
    }
    exit(0);
}
```

La compilación y ejecución muestra la siguiente salida:

```
administrador@ubuntul:~$ gcc ejemploForkPipe.c -o ejemploForkPipe
administrador@ubuntul:~$ ./ejemploForkPipe
ABUELO ENVIA MENSAJE AL HIJO...
    HIJO recibe mensaje de ABUELO: Saludos del Abuelo.
        NIETO RECIBE mensaje de su padre: Saludos del Padre..
            NIETO ENVIA MENSAJE a su padre...
                HIJO RECIBE mensaje de su hijo: Saludos del Nieto..
                    HIJO ENVIA MENSAJE a su padre...
El ABUELO RECIBE MENSAJE del HIJO: Saludos del Hijo...
administrador@ubuntul:~$
```

### PIPES CON NOMBRE O FIFOS (First In First Out)

Los pipes vistos anteriormente establecían un canal de comunicación entre procesos emparentados (padre-hijo).

Los FIFOS permiten comunicar procesos que no tienen que estar emparentados. Un FIFO es como un fichero con nombre que existe en el sistema de ficheros y que pueden abrir, leer y escribir múltiples procesos.

Los datos escritos se leen como en una cola, primero en entrar (FIRST IN), primero en salir (FIRST OUT); y una vez leídos, no pueden ser leídos de nuevo.

Los FIFOS tienen algunas diferencias con los ficheros:

- Una operación de escritura en un FIFO queda en espera hasta que el proceso pertinente abra el FIFO para iniciar la lectura.
- Solo se permite la escritura de información cuando un proceso vaya a recoger dicha información.

Hay varias formas de crear un FIFO: ejecutando el comando **mknod** desde la línea de comandos de Linux o desde un programa C usando la función `mknod()`.

## Tema 1 PSP

Para usar mknod desde la línea de comandos de Linux seguimos el siguiente formato:

```
mknod [opciones] nombreFichero p
```

Dónde: nombreFichero es el nombre del FIFO.

Las opciones pueden ser:

- m modo, --mode =modo: establece los permisos de los ficheros creados según el valor de modo (su comportamiento es similar al del comando chmod).

- help: muestra en la salida estándar ayuda sobre el modo de empleo del comando, y luego finaliza.

- version: muestra en la salida estándar información sobre la versión, y luego finaliza.

El siguiente ejemplo crea un FIFO llamado FIFO 1 desde la línea de comandos y luego se muestra la información del fichero creado. Se puede observar el indicador "p" que aparece en la lista del directorio y el símbolo de pipe | detrás del nombre:

```
administrador@ubuntu1:~$ mknod FIFO1 p
administrador@ubuntu1:~$ ls -l FIFO1
prw-r--r-- 1 administrador administrador 0 2012-11-16 17:09 FIFO1|
```

## Tema 1 PSP

A continuación veamos cómo funciona el FIFO. Ejecuto desde la línea de comandos la orden *cat* con el nombre FIFO1:

```
administrador@ubuntu1:~$ cat FIFO1
```

Observamos que se queda a la espera. Abro una nueva terminal y ejecuto desde la línea de comandos la orden *l* para enviar la información del directorio al FIFO1:

```
administrador@ubuntu1:~$ l > FIFO1
```

Veremos que el *cat* que anteriormente estaba a la espera se ejecuta ya que ha recibido la información.

Para crear un FIFO en C, utilizamos la función *mknod()*. Su formato es el siguiente:

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

int mknod(const char *pathname, mode_t modo, dev_t dev);
```

Donde:

*pathname*: es el nombre del dispositivo creado.

*modo*: especifica tanto los permisos de uso y el tipo de nodo que se creará. Debe ser una combinación (utilizando OR bit a bit) de uno de los tipos de fichero que se enumeran a continuación y los permisos para el nuevo nodo.

El tipo de nodo debe ser uno de los siguientes:

- S\_IFREG o 0: para especificar un fichero normal (que será creado vacío).
- S\_IFCHR: para especificar un fichero especial de caracteres.
- S\_IFBLK: un fichero especial de bloques.
- S\_IFIFO: para crear un FIFO.

Si el tipo de fichero es S\_IFCHR o S\_IFBLK entonces *dev* debe especificar los números mayor y menor del fichero especial de dispositivo creado; en caso contrario, es ignorado. Si *pathname* ya existe, o es un enlace simbólico, esta llamada fallará devolviendo el error EEXIST.

La función **mknod()** devuelve 0 si ha funcionado correctamente, -1 si ha ocurrido un error.

A continuación, se muestra un ejemplo de uso de FIFOS. El programa *fifoCrea* crea un FIFO de nombre FIFO2 y lee la información del FIFO, mientras no hay información quedará en espera. El programa *fifoEscribe* escribe información en el FIFO. La Figura muestra la ejecución, primero se ejecuta *fifoCrea* desde un terminal y después ejecutamos varias veces *FIFO escribe* desde otro terminal.

## Tema 1 PSP



The image shows two overlapping terminal windows from an Ubuntu system. The top window displays the compilation and execution of a program named 'fifocrea'. The bottom window shows the compilation and execution of a program named 'fifoescribe'.

```
administrador@ubuntu1: ~  
Archivo Editar Ver Buscar Terminal Ayuda  
administrador@ubuntu1:~$ gcc fifocrea.c -o fifocrea  
administrador@ubuntu1:~$ ./fifocrea  
OBTENIENDO Información...Un saludo!!!  
OBTENIENDO Información...Un saludo!!!  
OBTENIENDO Información...Un saludo!!!  
[ ]
```

```
administrador@ubuntu1: ~  
Archivo Editar Ver Buscar Terminal Ayuda  
administrador@ubuntu1:~$ gcc fifoescribe.c -o fifoescribe  
administrador@ubuntu1:~$ ./fifoescribe  
Mandando información al FIFO...  
administrador@ubuntu1:~$ ./fifoescribe  
Mandando información al FIFO...  
administrador@ubuntu1:~$ ./fifoescribe  
Mandando información al FIFO...  
administrador@ubuntu1:~$
```



## Tema 1 PSP

### Código:

```
//fifocrea.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(void)
{
    int fp;
    int p, bytesleidos;
    char saludo[] = "Un saludo!!!\n", buffer[10];

    p=mknod("FIFO2", S_IFIFO|0666, 0);//permiso de lectura y escritura

    if (p==-1) {
        printf("HA OCURRIDO UN ERROR...\n");
        exit(0);
    }

    while(1) {
        fp = open("FIFO2", 0);
        bytesleidos= read(fp, buffer, 1);
        printf("OBTENIENDO Información...");
        while (bytesleidos!=0){
            printf("%s", buffer);
            bytesleidos= read(fp, buffer, 1);//leo otro byte
        }
        close(fp);
    }
    return(0);
}
```

```
//fifoescribe.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    int fp;
    char saludo[] = "Un saludo!!!\n";
    fp = open("FIFO2", 1);

    if(fp == -1) {
        printf("ERROR AL ABRIR EL FICHERO...");
        exit(1);
    }
    printf("Mandando información al FIFO...\n");
    write(fp,saludo, strlen(saludo));
    close(fp);
    return 0;
}
```

Con `mknod("FIFO2", S_IFIFO|0666, 0)` se crea un FIFO de nombre *FIFO2* con permisos de lectura y escritura.

### 2.2.3. Sincronización entre procesos

En el epígrafe anterior se han tratado los mecanismos más sencillos de comunicación entre procesos. Pero para que los procesos interactúen unos con otros necesitan cierto nivel de sincronización, es decir, es necesario que haya un funcionamiento coordinado entre los procesos a la hora de ejecutar alguna tarea.

Podemos utilizar señales para llevar a cabo la sincronización entre dos procesos.

A continuación, se muestran una serie de funciones útiles que utilizaremos para que un proceso padre y otro hijo se comuniquen de forma síncrona usando señales.

Una señal es como un aviso que un proceso manda a otro proceso. La función **signal()** es el gestor de señales por excelencia que especifica la acción que debe realizarse cuando un proceso recibe una señal.

```
#include <signal.h>

void (*signal(int Señal, void (*Func)(int)))(int);
```

Recibe dos parámetros:

- Señal: contiene el número de señal que queremos capturar. En nuestro ejemplo pondremos SIGUSR1 que es una señal definida por el usuario para ser usada en programas de aplicación. Otra señal interesante es SIGKILL que se usa para terminar con un proceso.
- Func: contiene la función a la que queremos que se llame. Esta función es conocida como el manejador de la señal (signal handler). En el ejemplo que se verá a continuación se definen dos manejadores de señal, uno para el proceso padre (void gestion\_padre( int signal )) y otro para el hijo (void gestion\_hijo( int signal )).

La función devuelve un puntero al manejador previamente instalado para esa señal. Un ejemplo de uso de la función: signal(SIGUSR1, gestion\_padre); significa que cuando el proceso (en este caso el proceso padre) recibe una señal SIGUSR1 se realizará una llamada a la función gestion \_padr().

## Tema 1 PSP

Para enviar una señal usaremos la función **kill()**:

```
#include <signal.h>
int kill(int Pid, int Señal);
```

Recibe dos parámetros: el PID del proceso que recibirá la señal y la señal. Por ejemplo y suponiendo que `pid_padre` es el PID de un proceso padre: `kill(pid_padre, SIGUSR1)`; envía una señal SIGUSR1 al proceso padre.

Cuando queremos que un proceso espere a que le llegue una señal, usamos la función **pause()**. Para capturar esa señal, el proceso debe haber establecido un tratamiento de la misma con la función `signal()`. Este es su formato:

```
int pause(void);
```

Por último la función **sleep()** suspende al proceso que realiza la llamada la cantidad de segundos indicada o hasta que se reciba una señal.

```
#include <unistd.h>
unsigned int sleep (unsigned int seconds);
```

En el siguiente ejemplo se crea un proceso hijo y el proceso padre le va a enviar dos señales SIGUSR1. Se define la función `manejador()` para gestionar la señal, visualizará un mensaje cuando el proceso hijo la reciba.

En el proceso hijo se realiza la llamada a `signal()` donde se decide lo que se hará en el caso de recibir una señal, en este caso pinta un mensaje.

Después hacemos un bucle infinito que no hace nada.

En el proceso padre se hacen las llamadas a `kill()` para enviar las señales.

Con la función `sleep()` hacemos que los procesos esperen un segundo antes de continuar.

## Tema 1 PSP

El código es el siguiente:

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <fcntl.h>
/*-----*/
/* gestión de señales en proceso HIJO */
void manejador( int signal )
{
    printf("Hijo recibe señal..%d\n", signal);
}
/*-----*/
int main()
{
    int pid_hijo;
    pid_hijo = fork(); //creamos hijo

    switch(pid_hijo)
    {

        case -1:
            printf( "Error al crear el proceso hijo...\n");
            exit( -1 );
        case 0: //HIJO
            signal( SIGUSR1, manejador); //MANEJADOR DE SEÑAL EN HIJO
            while(1) {
            };
            break;
        default: //PADRE envia 2 señales
            sleep(1);
            kill(pid_hijo, SIGUSR1); //ENVIA SEÑAL AL HIJO
            sleep(1);
            kill(pid_hijo, SIGUSR1); //ENVIA SEÑAL AL HIJO
            sleep(1);
            break;
    }
    return 0;
}
```

La compilación y ejecución muestra la siguiente salida:

```
administrador@ubuntu1:~$ gcc sincronizar-1.c -o sincronizar-1
administrador@ubuntu1:~$ ./sincronizar-1
Hijo recibe señal..10
Hijo recibe señal..10
administrador@ubuntu1:~$
```

En el ejemplo que se muestra a continuación un proceso padre y otro hijo se ejecutan de forma síncrona. Se han definido dos funciones para gestionar la señal uno para el padre y otro para el hijo, con las acciones que se realizarán cuando los procesos reciban una señal; en este caso se visualizará un mensaje.

En primer lugar el proceso padre crea el proceso hijo. Dentro de cada proceso se realiza una llamada a `signal()` donde se decide lo que se hará en el caso de recibir una señal:

## Tema 1 PSP

En el proceso padre tenemos las siguientes instrucciones donde se observa que entra en bucle infinito esperando a recibir una señal. Cuando recibe la señal se ejecutaría la función `gestionpadre()`. Con `kill()` envía la señal de respuesta al proceso hijo mediante su PID, y el proceso se vuelve a repetir

```
signal( SIGUSR1, gestion_padre );
while(1) {
    pause();//padre espera hasta recibir una señal del hijo
    sleep(1);
    kill(pid_hijo, SIGUSR1);//ENVIA SEÑAL AL HIJO
}
```

En el proceso hijo también tenemos un trozo de código parecido, por la colocación del `pause()` se puede deducir que es el proceso hijo el que inicia la comunicación con el padre mediante la llamada `kill()`. Primero envía la señal al padre y después espera a que le llegue una señal de respuesta, cuando recibe la señal ejecutaría la función `gestion_hijo()`:

```
signal( SIGUSR1, gestion_hijo );
while(1) { //bucle infinito

    sleep(1);
    kill(pid_padre, SIGUSR1);//ENVIA SEÑAL AL PADRE
    pause();//hijo espera hasta que llegue una señal de respuesta
}
```

## Tema 1 PSP

El código completo es el siguiente:

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <fcntl.h>
/*-----*/
/* gestión de señales en proceso padre */
void gestion_padre( int signal )
{
    printf("Padre recibe señal..%d\n", signal);
}

/* gestión de señales en proceso hijo */
void gestion_hijo( int signal )
{
    printf("Hijo recibe señal..%d\n", signal);
}
/*-----*/

int main()
{
    int pid_padre, pid_hijo;

    pid_padre = getpid();
    pid_hijo = fork(); //se crea el hijo

    switch(pid_hijo)
    {
        case -1:
            printf( "Error al crear el proceso hijo...\n");
            exit( -1 );

        case 0: //HIJO

            //tratamiento de la señal en proceso hijo
            signal( SIGUSR1, gestion_hijo );
            while(1) { //bucle infinito
                sleep(1);
                kill(pid_padre, SIGUSR1); //ENVIA SEÑAL AL PADRE
                pause(); //hijo espera hasta que llegue una señal de respuesta
            }
            break;

        default: //PADRE
            //tratamiento de la señal en proceso padre
            signal( SIGUSR1, gestion_padre );
            while(1) {
                pause(); //padre espera hasta recibir una señal del hijo
                sleep(1);

                kill(pid_hijo, SIGUSR1); //ENVIA SEÑAL AL HIJO
            }
            break;
    }
    return 0;
}
```

## Tema 1 PSP

Momento de la ejecución:

```
administrador@ubuntu1: ~  
Archivo Editar Ver Buscar Terminal Ayuda  
administrador@ubuntu1:~$ gcc sincronizar.c -o sincronizar  
administrador@ubuntu1:~$ ./sincronizar  
Padre recibe señal..10  
Hijo recibe señal..10  
Padre recibe señal..10  
Hijo recibe señal..10  
Padre recibe señal..10  
Hijo recibe señal..10  
Padre recibe señal..10  
Hijo recibe señal..10  
Padre recibe señal..10  
Hijo recibe señal..10  
^C  
administrador@ubuntu1:~$
```

Para detener el proceso podemos pulsar las teclas [CTRL+C] o bien mediante el comando `ps` podemos ver el PID de los procesos padre e hijo que se están ejecutando:

```
administrador@ubuntu1:~$ ps -fe | grep sincronizar  
1000      1678  1549  0 22:20 pts/0    00:00:00 ./sincronizar  
1000      1679  1678  0 22:20 pts/0    00:00:00 ./sincronizar  
1000      1687  1572  0 22:21 pts/1    00:00:00 grep --color=auto  
                                     sincronizar  
administrador@ubuntu1:~$ kill 1679  
administrador@ubuntu1:~$ kill 1678  
administrador@ubuntu1:~$
```

Primero eliminaremos el proceso hijo (PID 1679) y después el padre (PID 1678). Al eliminar el hijo el padre quedará esperando.

---

### ACTIVIDAD 1.3

Realiza un programa C en donde un hijo envíe 3 señales `SIGUSR1` a su padre y después envíe una señal `SIGKILL` para que el proceso padre termine.

---