

## 1.- Procesos y Sistema Operativo

### Funcionamiento básico del Sistema Operativo.

**Kernel** o núcleo del S.O. se encarga de la funcionalidad básica del sistema, el responsable de la gestión de los recursos del ordenador, se accede al núcleo a través de las **llamadas al sistema**, es la parte más pequeña del sistema en comparación con la interfaz. El resto del sistema operativo se le denomina como **programas del sistema**.

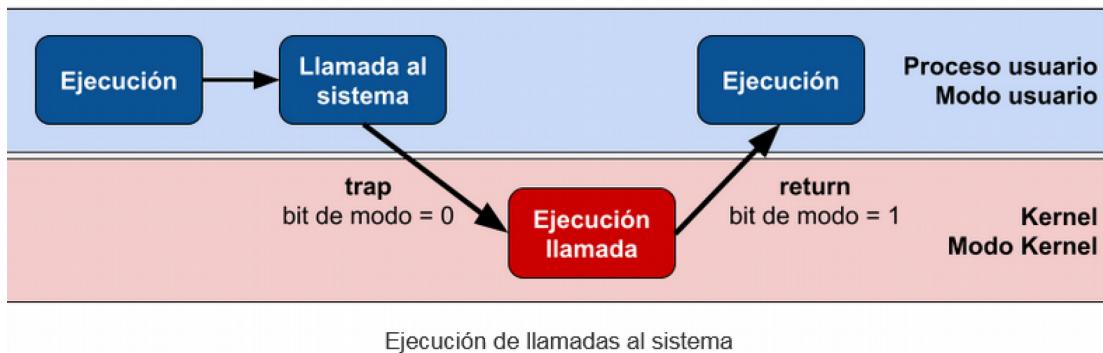
El kernel funciona normalmente a base de **interrupciones**, que no es sino una suspensión temporal de la ejecución de un proceso, para ejecutar una rutina (independiente del S.O.) que trate dicha interrupción. Mientras una rutina trata la interrupción, se deshabilitan la llegada de nuevas interrupciones.

Las **llamadas al sistema** son la **interfaz** que proporciona el kernel para que los programas de usuario puedan hacer uso de manera segura de determinadas partes del sistema.

El **modo dual**, es una característica del hardware que permite al S.O. protegerse:

- Modo **usuario** (1): utilizado para la ejecución de programas de usuario.
- Modo **kernel** (0): también denominado “**modo supervisor**” o “**modo privilegiado**”, donde las instrucciones del procesador más delicadas solo se pueden ejecutar si el procesador está en modo kernel.

Las llamadas al sistema por parte de un programa de usuario generan una interrupción que se denomina **trap**.



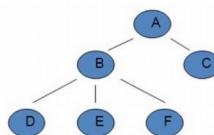


Las llamadas al sistema suelen estar escritas en lenguajes de bajo nivel como C o C++. No obstante, los programadores no suelen hacer llamadas al sistema directamente sino que suelen utilizar las API's que proporcionan los sistemas operativos como Win32, Win64, API POSIX para Unix, Linux o Mac OS.

**Proceso** = programa en ejecución (código ejecutable + datos + pila del programa + contador de programa + puntero de pila + registros).

## Procesos

- ✖ Programa en ejecución:
  - + Código ejecutable del programa
  - + Datos
  - + Pila del programa
  - + Contador de programa
  - + Puntero a la pila y otros registros
  - + Toda la información necesaria para ejecutar el programa
- ✖ Tabla de procesos
- ✖ Estructura en árbol de los procesos
- ✖ Señales
- ✖ uid

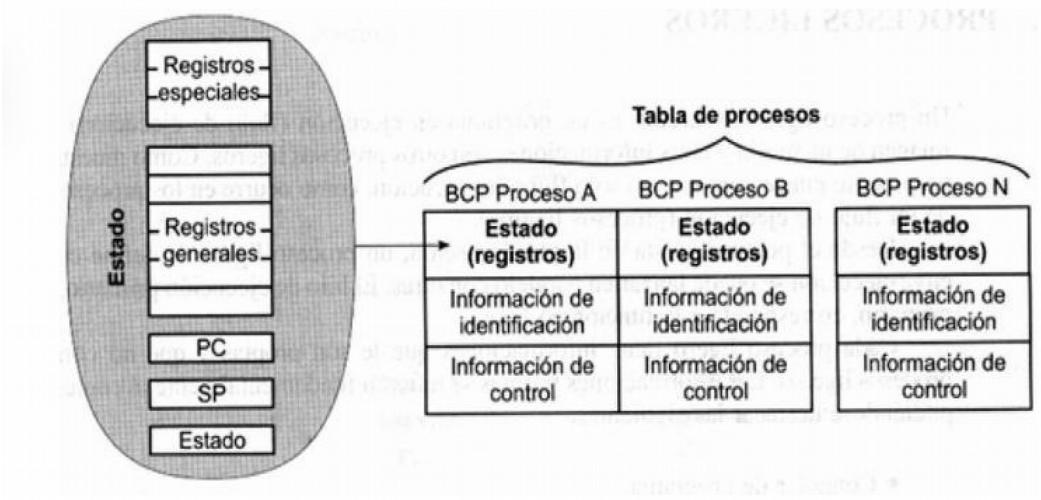


5

Código ejecutable del programa. Datos. Pila del programa. Contador de programa. Puntero a la pila y otros registros. Toda la información necesaria para ejecutar el programa. Tabla de procesos. Estructura en árbol de los procesos. Señales. uid. A. B. C. D. E. F.

TODOS los programas se ejecutan y organizan como un conjunto de procesos y es el sistema operativo el que decide los estados a los que pasan cada proceso. Cuando se suspende la ejecución de un proceso, luego deberá rearrancarse en el mismo estado en el que se encontraba antes de ser suspendido. Esto implica que debemos almacenar en algún sitio la información referente a ese proceso para poder luego restaurarla tal como estaba antes.

La **BCP** (Bloque de Control de Proceso) es donde se almacenará esa información: *identificador, estado, contador de programa (PC), registros de la CPU, puntero de pila (SP), prioridad del proceso, gestión de memoria, ....*



En Unix / Linux mediante el comando **ps** (process status) podemos ver información de los procesos.

Opciones:

- s : simple
- l: lista
- a: salida
- h: hilos
- v: varios
- t: todos

```

juan@juan-Lenovo-ideapad-310-15IKB: ~
Archivo Editar Ver Buscar Terminal Ayuda
juan@juan-Lenovo-ideapad-310-15IKB:~$ ps
  PID TTY          TIME CMD
 5059 pts/1    00:00:00 bash
 5077 pts/1    00:00:00 ps
juan@juan-Lenovo-ideapad-310-15IKB:~$ █

```

```
root@juan-Lenovo-ideapad-310-15IKB:/home/juan# ps l
F  UID  PID  PPID PRI  NI   VSZ   RSS WCHAN  STAT TTY          TIME COMMAND
4  0  1242  1217  20  0 413104 80768 poll_s Ssl+  tty7      2:05 /usr/lib/xorg/Xorg -co
4  0  1532     1  20  0 18448  1796 poll_s Ss+  tty1      0:00 /sbin/agetty --noclear
4  0  5135  5059  20  0 57616  4228 poll_s S     pts/1      0:00 sudo su
4  0  5136  5135  20  0 56888  3472 wait   S     pts/1      0:00 su
4  0  5137  5136  20  0 23892  3892 wait   S     pts/1      0:00 bash
0  0  5813  5137  20  0 31480  1448 -       R+    pts/1      0:00 ps l
root@juan-Lenovo-ideapad-310-15IKB:/home/juan# ps h
1242  tty7      Ssl+  2:05 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm/r
1532  tty1      Ss+   0:00 /sbin/agetty --noclear tty1 linux
5135  pts/1      S     0:00 sudo su
5136  pts/1      S     0:00 su
5137  pts/1      S     0:00 bash
5814  pts/1      R+   0:00 ps h
root@juan-Lenovo-ideapad-310-15IKB:/home/juan# 
```

Si ejecutamos el programa ***sleep*** podremos ver como quedan registrados dichos procesos durante los segundos que les hayamos indicado, en nuestro ejemplo 45 segundos cada uno a partir de cuando se lanazan.

### Sleep 45 &

```
root@juan-Lenovo-ideapad-310-15IKB:/home/juan# ps
  PID TTY          TIME CMD
 5135 pts/1      00:00:00 sudo
 5136 pts/1      00:00:00 su
 5137 pts/1      00:00:00 bash
 5182 pts/1      00:00:00 ps
root@juan-Lenovo-ideapad-310-15IKB:/home/juan# sleep 45 &
[1] 5183
root@juan-Lenovo-ideapad-310-15IKB:/home/juan# sleep 45 &
[2] 5184
root@juan-Lenovo-ideapad-310-15IKB:/home/juan# ps
  PID TTY          TIME CMD
 5135 pts/1      00:00:00 sudo
 5136 pts/1      00:00:00 su
 5137 pts/1      00:00:00 bash
 5183 pts/1      00:00:00 sleep
 5184 pts/1      00:00:00 sleep
 5185 pts/1      00:00:00 ps
root@juan-Lenovo-ideapad-310-15IKB:/home/juan# 
```

Para saber el **pid** de un proceso utilizamos la instrucción ***pidof***

```
root@juan:/home/juan# ps
  PID TTY          TIME CMD
 6344 pts/2      00:00:00 sudo
 6345 pts/2      00:00:00 su
 6346 pts/2      00:00:00 bash
 6635 pts/2      00:00:00 ps
root@juan:/home/juan# pidof su
6345
root@juan:/home/juan# pidof bash
6346 6333 5688
root@juan:/home/juan# 
```

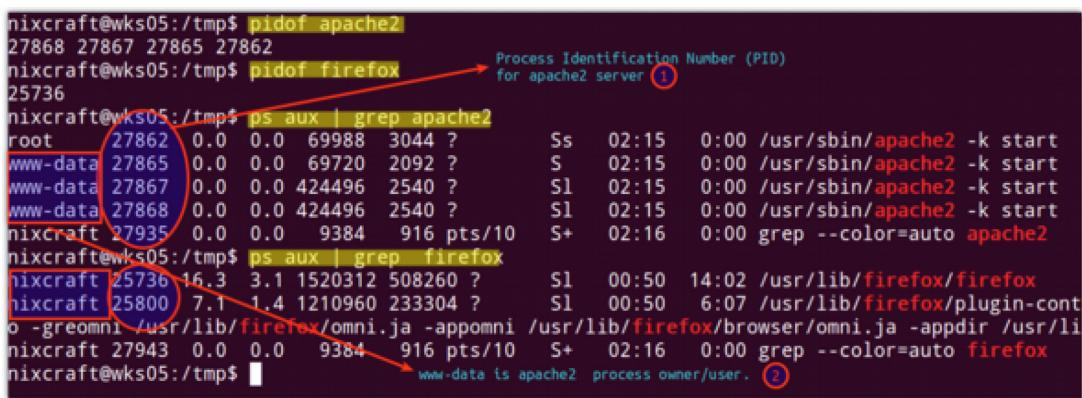
procesos del sistema y no solo los nuestros.

Si queremos saber todos los

user id efectivo (EUID), utilizamos **ps aux**

- a: eliminar la restricción BSD “only yourself” para agregar procesos de otros usuarios
- u: utilizar el formato orientado al usuario
- x: eliminar la restricción BSD “must have a tty” para agregar procesos que no tengan una tty(tty= tipo de terminal asociada)

```
root@juan-Lenovo-ideapad-310-15IKB:/home/juan# ps aux | grep apache2
root      5553  0.0  0.0 16756 1028 pts/1    S+   11:20   0:00 grep --color=auto apache2
root@juan-Lenovo-ideapad-310-15IKB:/home/juan# ps aux | grep firefox
root      5780  0.0  0.0 16756 1016 pts/1    S+   11:21   0:00 grep --color=auto firefox
```



nixcraft@wks05:/tmp\$ pidof apache2  
 27868 27867 27865 27862  
 nixcraft@wks05:/tmp\$ pidof firefox  
 25736  
 nixcraft@wks05:/tmp\$ ps aux | grep apache2  
 root 27862 0.0 0.0 69988 3044 ? Ss 02:15 0:00 /usr/sbin/apache2 -k start  
 www-data 27865 0.0 0.0 69720 2092 ? S 02:15 0:00 /usr/sbin/apache2 -k start  
 www-data 27867 0.0 0.0 424496 2540 ? S1 02:15 0:00 /usr/sbin/apache2 -k start  
 www-data 27868 0.0 0.0 424496 2540 ? S1 02:15 0:00 /usr/sbin/apache2 -k start  
 nixcraft 27935 0.0 0.0 9384 916 pts/10 S+ 02:16 0:00 grep --color=auto apache2  
 nixcraft@wks05:/tmp\$ ps aux | grep firefox  
 nixcraft 25736 16.3 3.1 1520312 508260 ? S1 00:50 14:02 /usr/lib/firefox/firefox  
 nixcraft 25800 7.1 1.4 1210960 233304 ? S1 00:50 6:07 /usr/lib/firefox/plugin-cont  
 o-greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appdir /usr/li  
 nixcraft 27943 0.0 0.0 9384 916 pts/10 S+ 02:16 0:00 grep --color=auto firefox  
 nixcraft@wks05:/tmp\$

The terminal window shows several processes for apache2 and firefox. Red circles highlight the process IDs (27868, 27867, 27865, 27862 for apache2; 25736, 25800 for firefox) and the user 'www-data' associated with them. A red arrow points from the text 'Process Identification Number (PID)' to the first highlighted PID. Another red arrow points from the text 'www-data is apache2 process owner/user.' to the second highlighted PID.

Instrucción **ps -AF** que muestra todos los procesos activos con todos los detalles

- SZ: tamaño virtual
- RSS: tamaño de la parte residente en memoria
- PSR: procesador asignado



root@juan: /home/juan  
 Archivo Editar Ver Buscar Terminal Ayuda  
 juan@juan:~\$ sudo su  
 [sudo] password for juan:  
 root@juan:/home/juan# clear  
 root@juan:/home/juan# ps -AF  
 UID PID PPID C SZ RSS PSR STIME TTY TIME CMD  
 root 1 0 0 29967 5896 2 16:42 ? 00:00:00 /sbin/init splash  
 root 2 0 0 0 0 2 16:42 ? 00:00:00 [kthreadd]  
 root 3 2 0 0 0 0 16:42 ? 00:00:00 [ksoftirqd/0]  
 root 5 2 0 0 0 0 16:42 ? 00:00:00 [kworker/0:0H]  
 root 6 2 0 0 0 1 16:42 ? 00:00:00 [kworker/u8:0]  
 root 7 2 0 0 0 2 16:42 ? 00:00:00 [rcu\_sched]

En Windows podemos ver los procesos ejecutando la instrucción **tasklist**

Nombre de imagen	PID	Nombre de sesión	Núm. de ses.	Uso de memoria
System Idle Process	0	Services	0	8 KB
System	4	Services	0	2.328 KB
Registry	96	Services	0	39.180 KB
smss.exe	368	Services	0	540 KB
csrss.exe	576	Services	0	2.044 KB
wininit.exe	660	Services	0	3.124 KB
services.exe	780	Services	0	6.284 KB
lsass.exe	820	Services	0	17.612 KB
svchost.exe	924	Services	0	892 KB
svchost.exe	948	Services	0	29.736 KB
fontdrvhost.exe	968	Services	0	904 KB
svchost.exe	416	Services	0	13.672 KB
svchost.exe	428	Services	0	5.052 KB
svchost.exe	1112	Services	0	5.516 KB
svchost.exe	1108	Services	0	4.608 KB
svchost.exe	1184	Services	0	5.548 KB
svchost.exe	1192	Services	0	5.944 KB
svchost.exe	1280	Services	0	2.496 KB
svchost.exe	1304	Services	0	4.552 KB
svchost.exe	1508	Services	0	12.032 KB
svchost.exe	1556	Services	0	5.016 KB
svchost.exe	1568	Services	0	4.580 KB
igfxCUIService.exe	1628	Services	0	3.164 KB
svchost.exe	1656	Services	0	5.884 KB
svchost.exe	1664	Services	0	2.260 KB
svchost.exe	1760	Services	0	6.144 KB
svchost.exe	1768	Services	0	5.712 KB
svchost.exe	1800	Services	0	1.980 KB
svchost.exe	1812	Services	0	4.344 KB
svchost.exe	1916	Services	0	3.820 KB
svchost.exe	1952	Services	0	3.292 KB
Memory Compression	1960	Services	0	206.580 KB
svchost.exe	2004	Services	0	6.488 KB
svchost.exe	1360	Services	0	6.012 KB
svchost.exe	2084	Services	0	11.420 KB
svchost.exe	2144	Services	0	7.432 KB
svchost.exe	2192	Services	0	8.212 KB
svchost.exe	2280	Services	0	14.572 KB
svchost.exe	2348	Services	0	3.148 KB

Con esta instrucción sabremos que servicios se están ejecutando bajo el proceso **svchost.exe**, es el nombre de proceso de host genérico para servicios que se ejecutan desde bibliotecas de vínculos dinámicos (DLL), hay tantos para evitar riesgos ya que si estuviera todo en uno unívoco fallo podría colapsar el sistema.

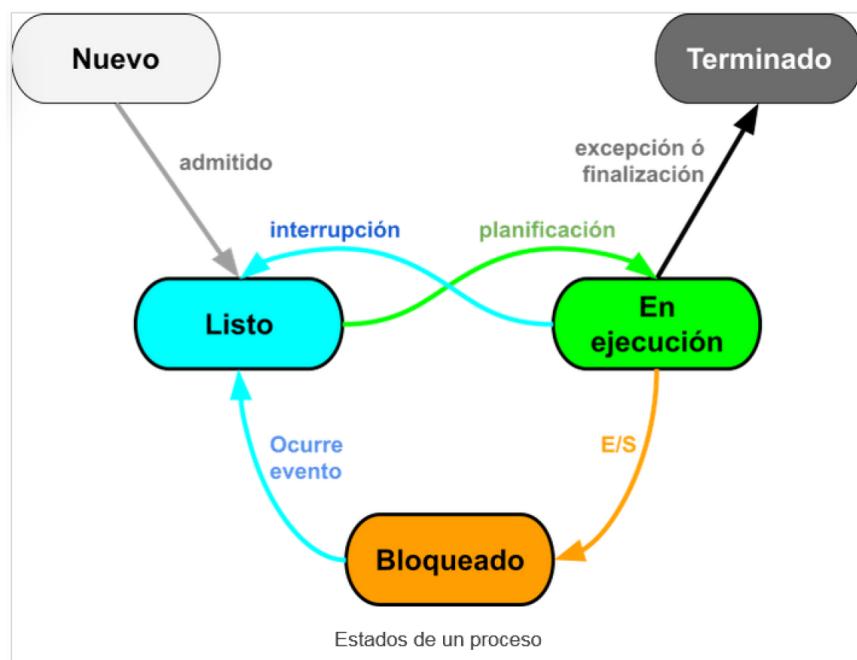
**tasklist /svc /fi “imagename eq svchost.exe”**

```
Símbolo del sistema
C:\Users\JUANJO>tasklist /svc /fi "imagename eq svchost.exe"
Nombre de imagen          PID Servicios
=====
svchost.exe                924 PlugPlay
svchost.exe                948 BrokerInfrastructure, DcomLaunch, Power,
                           SystemEventsBroker
svchost.exe                416 RpcEptMapper, RpcSs
svchost.exe                428 LSM
svchost.exe                1112 bthserv
svchost.exe                1108 BthAvctpSvc
svchost.exe                1184 TimeBrokerSvc
svchost.exe                1192 NcbService
svchost.exe                1280 DisplayEnhancementService
svchost.exe                1304 EventSystem
svchost.exe                1508 EventLog
svchost.exe                1556 SENS
svchost.exe                1568 SEMgrSvc
svchost.exe                1656 ProfSvc
svchost.exe                1664 BTAGService
svchost.exe                1760 SysMain
svchost.exe                1768 nsi
```

## Estados de un proceso.

- **En ejecución.** Está usando el procesador. El sistema operativo utiliza el mecanismo de interrupciones para controlar su ejecución. Si el proceso necesita un recurso, incluyendo la realización de operaciones de E/S, invocará a la llamada al sistema correspondiente. Si un proceso se ejecuta durante el máximo tiempo permitido por la política del sistema, salta un temporizador que lanza una interrupción. Si el sistema es de tiempo compartido, lo para y lo pasa a estado de *listo*.
- **Bloqueado.** El proceso se encuentra bloqueado *esperando* a que ocurra algún suceso. Por ejemplo puede estar esperando a que termine alguna operación de E/S, o bien a sincronizarse con otro proceso. Cuando ocurre el **evento** que lo desbloquea, el proceso queda pendiente de ser planificado por el S.O. **no pasa directamente a ejecución**.

- **Listo.** Está *parado* temporalmente y listo para ejecutarse cuando se le dé la oportunidad. El sistema operativo todavía no le asignó un procesador para ejecutarse. El *planificador* del S.O. será el responsable de seleccionar el proceso para que pase a estado de ejecución.
- **Nuevo.** El fichero es creado a partir de un ejecutable.
- **Terminado.** El proceso termina y libera su imagen de memoria. Es el propio proceso el que debe llamar al sistema para indicar que ha terminado, aunque el sistema puede finalizarlo con una excepción (que es una interrupción especial).



### Transiciones entre estados:

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

### Colas de procesos.

Uno de los objetivos de los sistemas operativos es la multiprogramación, es decir, admitir varios procesos en memoria para maximizar el uso del procesador. Esto funciona ya que los procesos se irán intercambiando el uso del procesador para su ejecución de forma concurrente. Para ello, el sistema operativo organiza los procesos en varias colas migrándolos de unas colas a otras:

- **Cola de procesos:** contiene todos los procesos del sistema
- **Cola de procesos preparados:** todos los procesos listos esperando para ejecutarse.
- **Varias colas de dispositivos:** procesos que están esperando alguna operación de E/S.

## Planificación de procesos

El **planificador** es el encargado de seleccionar los movimientos de los procesos entre las distintas colas. Existe una planificación a corto plazo y otra a largo plazo, veamos cada una:

- **Corto plazo:** selecciona los procesos de la cola de preparados para pasarlo a ejecución, se invoca con mucha frecuencia, del orden de milisegundos, por lo que el algoritmo debe ser muy sencillo.
  - **Planificación sin desalojo:** un proceso en ejecución sólo se saca si termina o bien se queda bloqueado.
  - **Planificación apropiativa:** solo se saca un proceso de ejecución si termina, se bloquea o bien por último aparece un proceso con mayor prioridad.
  - **Tiempo compartido:** cada cierto tiempo (**cuanto**), se desaloja un proceso y se mete otro, Se considera que todos los procesos tienen la misma prioridad.
- **Largo plazo:** selecciona que procesos nuevos pasan a la cola de preparados. Hace un control del grado de multiprogramación del proceso para tomar sus decisiones.

## Cambios de contexto

El cambio de contexto que se hace al cambiar un proceso, es tiempo perdido, no se considera trabajo útil. Cambiar el estado del proceso, conlleva un cambio de estado del procesador (cambio valores de registro) e información de la gestión de memoria, por muy rápido que se haga si se hace con mucha frecuencia puede provocar una ralentización del sistema, **por eso tener muchos programas abiertos provoca una disminución importante en el rendimiento del sistema.**

## 2.- Control de procesos en Linux

Para que dentro de un programa podamos lanzar otro programa y realizar una tarea concreta, Linux ofrece varias funciones:

- **system()**: esta función está en la librería de C/C++ **stdlib.h** su interfaz recibe como parámetro una cadena que indica el comando que se desea procesar, esta pasará al interprete de comando en el que esté ejecutándose. devolverá -1 si hay error y el estado en caso contrario.

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

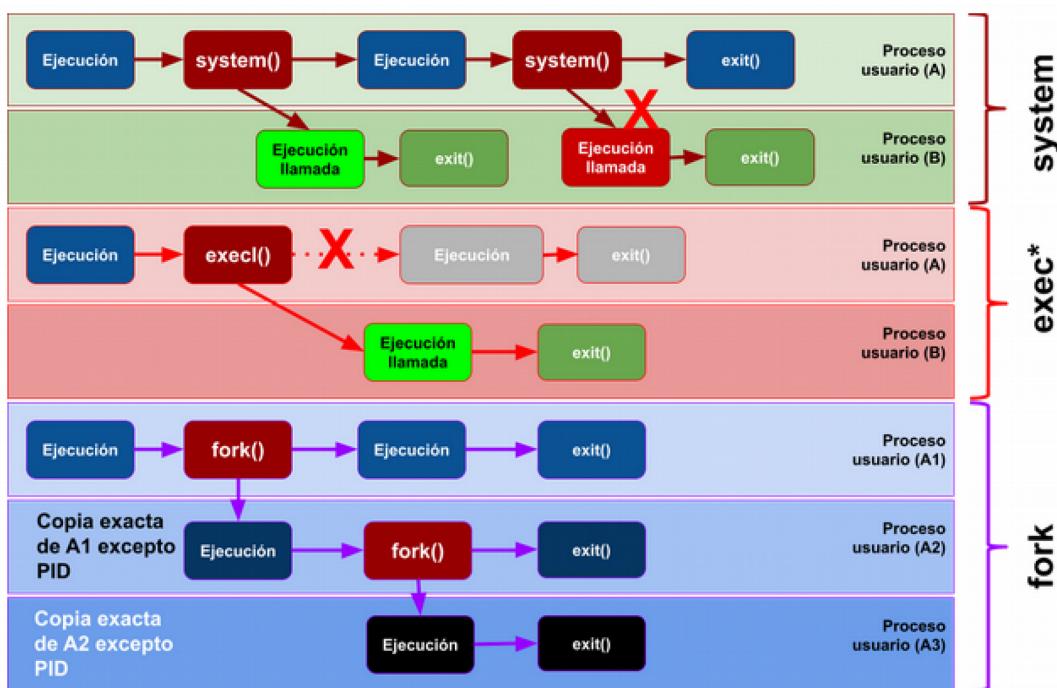
**NOTA:** Esta función no debe usarse des un programa con privilegios de administrador ya que puede usar valores extraños con algunas variables de entorno y se podría comprometer la seguridad del sistema.

- **execl()**: esta función realiza la ejecución y terminación del comando. Hay que usarla en lugar de system(), y se encuentra en la librería *unistd.h*.

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

- **fork()**: esta función crea nuevos procesos sin ningún tipo de parámetro, se encuentra también en la librería *unistd.h*.

```
pid_t fork(void);
```



Preparando nuestro sistema linux. Compilador y utilidades de desarrollo

- sudo apt-get update
- sudo apt-get install build-essential

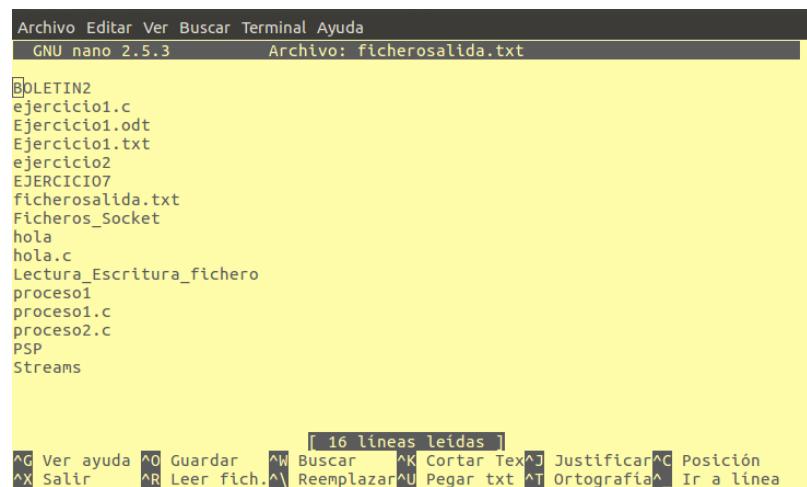
### Ejercicio 1

#### Creando el primer programa de control de procesos “proceso1.c” (Ejemplo de system)

Crea un programa que liste el contenido del directorio actual y lo guarde en un fichero . A continuación abre el fichero con nano y lo muestra por pantalla.

## SOLUCIÓN

```
#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 > ficherosalida.txt"));
    printf("\n\tAbrimos con nano el fichero ...");
    printf("%d",system("nano ficherosalida.txt"));
    printf("\n\tEste comando es erroneo: %d",system("msword"));
    printf("\nFin del programa");
}
```



```
Archivo Editar Ver Buscar Terminal Ayuda
GNU nano 2.5.3 Archivo: ficherosalida.txt

BOLETIN2
ejercicio1.c
Ejercicio1.odt
Ejercicio1.txt
ejercicio2
EJERCICIO7
ficherosalida.txt
Ficheros_Socket
hola
hola.c
Lectura_Escritura_fichero
procesol
proceso1.c
proceso2.c
PSP
Streams

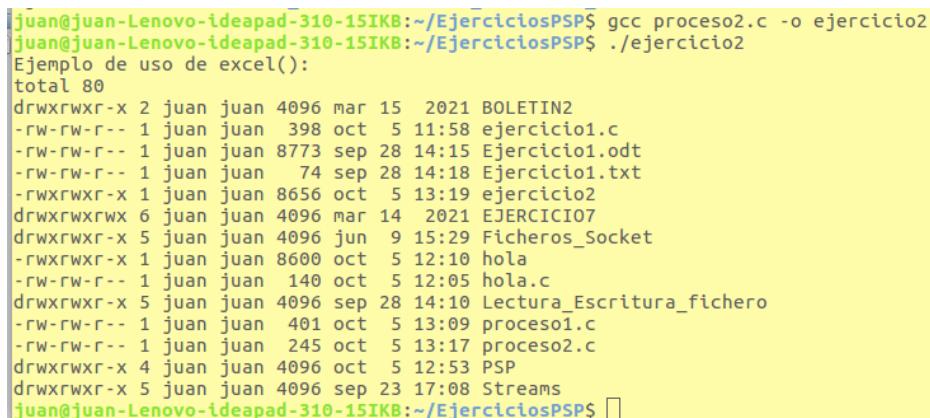
[ 16 líneas leidas ]
^G Ver ayuda ^O Guardar ^W Buscar ^K Cortar Tex^J Justificar^C Posición
^X Salir ^R Leer fich.^V Reemplazar^U Pegar txt ^T Ortografía^I Ir a linea
```

## Ejercicio 2

### Segundo programa de control de proceso uso de excel

Crea un programa que muestre el contenido del directorio actual y cuando lo muestre termine el proceso. Poner la siguiente línea para comprobar que el proceso termina.

```
printf("\n Esta instrucción no se llega a ejecutar");
```



```
juan@juan-Lenovo-ideapad-310-15IKB:~/EjerciciosPSP$ gcc proceso2.c -o ejercicio2
juan@juan-Lenovo-ideapad-310-15IKB:~/EjerciciosPSP$ ./ejercicio2
Ejemplo de uso de excel():
total 80
drwxrwxr-x 2 juan juan 4096 mar 15 2021 BOLETIN2
-rw-rw-r-- 1 juan juan 398 oct 5 11:58 ejercicio1.c
-rw-rw-r-- 1 juan juan 8773 sep 28 14:15 Ejercicio1.odt
-rw-rw-r-- 1 juan juan 74 sep 28 14:18 Ejercicio1.txt
-rwxrwxr-x 1 juan juan 8656 oct 5 13:19 ejercicio2
drwxrwxrwx 6 juan juan 4096 mar 14 2021 EJERCICIO7
drwxrwxr-x 5 juan juan 4096 jun 9 15:29 Ficheros_Socket
-rwxrwxr-x 1 juan juan 8600 oct 5 12:10 hola
-rw-rw-r-- 1 juan juan 140 oct 5 12:05 hola.c
drwxrwxr-x 5 juan juan 4096 sep 28 14:10 Lectura_Escritura_fichero
-rw-rw-r-- 1 juan juan 401 oct 5 13:09 procesol.c
-rw-rw-r-- 1 juan juan 245 oct 5 13:17 proceso2.c
drwxrwxr-x 4 juan juan 4096 oct 5 12:53 PSP
drwxrwxr-x 5 juan juan 4096 sep 23 17:08 Streams
juan@juan-Lenovo-ideapad-310-15IKB:~/EjerciciosPSP$
```

## SOLUCIÓN

```
#include <stdio.h>
#include <unistd.h>
void main() {
    printf("Ejemplo de uso de exec():");
    printf("\n\tListado del directorio actual:");
    execl("/bin/ls","ls","-l", (char *)NULL);
    printf("\n Esta instrucción no se llega a ejecutar");
}
```

### **3.- Procesos con getpid(), getppid(), fork() , wait()**

En C la instrucción para crear procesos es fork(). Las funciones system() y exec\*() lo que hacen es llamar a programas ya existentes para crear un proceso, en cambio fork no llama a ningún otro programa existente sino que se replica el proceso donde ese ejecuta dicha instrucción.

**getpid() : Devuelve el identificador del proceso actual.**

**getppid(): Devuelve el identificador del proceso padre del proceso actual.**

#### **Ejercicio3: Ejemplo de uso de getpid() y getppid()**

Escribe un programa que imprima el PID del proceso actual y el de su padre

```
root@juan:/home/juan/EjerciciosProcesos# ./Ejercicio3b
Mi PID es: 8001
El PID de mi padre es: 6346
root@juan:/home/juan/EjerciciosProcesos# 
root@juan:/home/juan/EjerciciosProcesos# ps
  PID TTY      TIME CMD
  6344 pts/2    00:00:00 sudo
  6345 pts/2    00:00:00 su
  6346 pts/2    00:00:00 bash
  8027 pts/2    00:00:00 ps
root@juan:/home/juan/EjerciciosProcesos# 
```

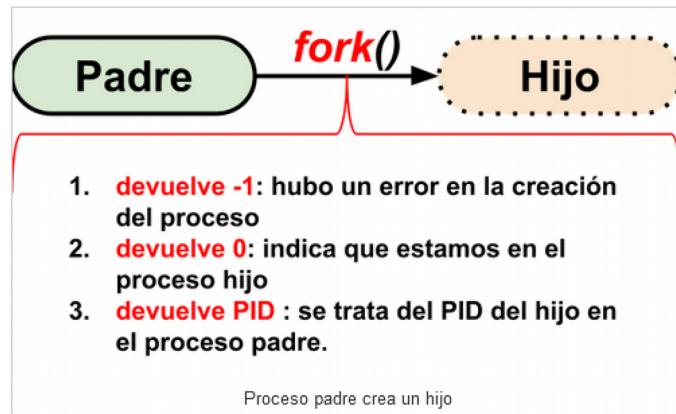
## SOLUCIÓN

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

void main(void) {
    pid_t , id_actual, id_padre;
    id_actual = getpid();
    id_padre = getppid();

    printf("Mi PID es: %d\n", id_actual);
    printf("El PID de mi papa es: %d\n", id_padre);

}
```



## Ejercicio 4

Crea un programa que muestre en pantalla la siguiente información

```
root@juan:/home/juan/EjerciciosProcesos# ./Ejercicio4
Yo soy el padre de la criatura:
    Mi PID es 17248, el de mi padre (abuelo de la criatura) es 17169.
    Mi hijo si es de verdad hijo mio deberia tener el PID 17249.
Soy el proceso hijo
    Mi PID es 17249, y el de mi papa 2833
root@juan:/home/juan/EjerciciosProcesos# 
```

## SOLUCIÓN

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

void main(void) {
    pid_t id_actual, id_padre, pid;

    pid = fork();

    if (pid == -1) { // Hubo error
        printf("Hubo un problema de impotencia al crear el hijo");
        exit(-1);
    }
    // Si todo va bien y se crea el hijo tenemos que hacer
    // que el programa ejecute un código con distinto para cada
    // proceso
    if (pid == 0) { // Nos encontramos en el hijo
        printf ("Soy el proceso hijo\n\t");
        printf(" Mi PID es %d, y el mi papa %d\n",getpid(),getppid());
    } else { // Nos encontramos en el padre
        printf("Yo soy el padre de la criatura:\n\t");
        printf("Mi PID es %d, el de mi padre (abuelo de la criatura) es %d.\n\t",getpid(),getppid());
        printf("Mi hijo si es de verdad hijo mio deberia tener el PID %d.\n",pid);
    }
    exit(0);
}
```

## Procesos huérfanos

Si estando en el proceso hijo dice que su padre es **PID=1**. Esto ocurre porque cuando el hijo ejecuta la instrucción el proceso padre ya ha terminado su ejecución de manera que el proceso del S.O. adopta todos los procesos hijos del padre que acaba de terminar dejando procesos huérfanos. Para evitar crear procesos huérfanos tenemos la función `wait()` que nos permite parar un proceso hasta que un proceso hijo no termine.

### Ejercicio 5

Crea un programa que muestre la siguiente información

```
root@juan:/home/juan/EjerciciosProcesos# ./Ejercicio4
Soy el proceso hijo
    Mi PID es 17300, y el mi papa 17299
Yo soy el padre de la criatura:
    Mi PID es 17299, el de mi padre (abuelo de la criatura) es 17169
    Mi hijo si es de verdad hijo mio deberia tener el PID 17300.
root@juan:/home/juan/EjerciciosProcesos#
```

El abuelo será el interprete de comando del sistema:

```
root@juan:/home/juan/EjerciciosProcesos# ps
  PID TTY      TIME CMD
17167 pts/0    00:00:00 sudo
17168 pts/0    00:00:00 su
17169 pts/0    00:00:00 bash
17361 pts/0    00:00:00 ps
```

## SOLUCIÓN

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

void main(void) {
    pid_t id_actual, id_padre, pid;

    pid = fork();

    if (pid == -1) { // Hubo error
        printf("Hubo un problema de impotencia al crear el hijo");
        exit(-1);
    }
    // Si todo va bien y se crea el hijo tenemos que hacer
    // que el programa ejecute un código con distinto para cada
    // proceso
    if (pid == 0) { // Nos encontramos en el hijo
        printf ("Soy el proceso hijo\n\t");
        printf(" Mi PID es %d, y el mi papa %d\n",getpid(),getppid());
    } else { // Nos encontramos en el padre
        id_actual = wait(NULL);
        printf("Yo soy el padre de la criatura:\n\t");
        printf("Mi PID es %d, el de mi padre (abuelo de la criatura) es %d.\n\t",
               getpid(),getppid());
        printf("Mi hijo si es de verdad hijo mio deberia tener el PID %d.\n",pid);
    }
    exit(0);
}
```

**Ejercicio 6 : Creación de un proceso que cree un hijo y éste a su vez cree otro proceso:**



```
root@juan:/home/juan/EjerciciosProcesos# gcc Ejercicio5.c -o Ejercicio5
root@juan:/home/juan/EjerciciosProcesos# ./Ejercicio5
      Soy el proceso NIETO 18126, Mi padre es = 18125
      Soy el proceso HIJO 18125, Mi padre es = 18124
      Mi hijo: 18126 terminó.
Yo soy el abuelo de las dos criaturas anteriores:
      Mi PID es 18124, el de mi padre (Sistema Operativo) es 17169.
      Mi hijo si es de verdad hijo mio deberia tener el PID 18125.
A mi nieto no lo puedo conocer, solo reconozco a mi generación inmediata
Para conocer a mi NIETO deberia implementar algún sistema de comunicación entre procesos
root@juan:/home/juan/EjerciciosProcesos# █
```

### SOLUCIÓN

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
void main(void) {
    pid_t pid, Hijo_pid, pid2, Hijo2_pid, id_actual;
    pid = fork(); // Soy el abuelo e intento crear a mi hijo
    if (pid == -1) { // Hubo error
        printf("Hubo un problema de impotencia al crear el hijo");
        exit(-1);
    }
    // Si todo va bien y se crea el hijo tenemos que hacer
    // que el programa ejecute un código con distinto para cada
    // proceso
    if (pid == 0) {
        // Nos encontramos en el hijo
        pid2 = fork(); //soy el hijo y creo al nieto
        switch(pid2) {
            case -1: //error
                printf("No se ha podido crear el proceso nieto en el hijo");
                exit(-1);
            break;
            case 0: // Estoy en el nieto
```

```
printf("\t\tSoy el proceso NIETO %d, Mi padre es = %d \n",getpid(),getppid());
break;
default: //proceso padre
Hijo2_pid=wait(NULL); //espero a que termine nieto, que es mi hijo.
printf("\t\tSoy el proceso HIJO %d, Mi padre es = %d \n",getpid(),getppid());
printf("\t\tMi hijo: %d terminó.\n", Hijo2_pid);
}

} else {
// Nos encontramos en el abuelo
id_actual = wait(NULL); // Espero a que termine mi hijo, que a su vez espera que termine el
nieto
printf("Yo soy el abuelo de las dos criaturas anteriores:\n\t");
printf("Mi PID es %d, el de mi padre (Sistema Operativo) es %d.\n\t", getpid(),getppid());
printf("Mi hijo si es de verdad hijo mio deberia tener el PID %d.\n",pid);
printf("A mi nieto no lo puedo conocer, solo reconozco a mi generación inmediata\n");
printf("Para conocer a mi NIETO debería implementar algún sistema de comunicación entre
procesos\n");
}
exit(0);
}
```

## Ejercicio 7

Realiza un programa en C que cree un proceso (tendremos 2 procesos el padre y el hijo)- El programa definirá una variable entera y le dará valor de 6. El proceso padre incrementará el valor en 5 y el hijo le restará 5. Se deberán mostrar los valores en pantalla. Piensa un poco en el resultado que debería dar y luego comprueba si coincide con lo que estabas pensando.

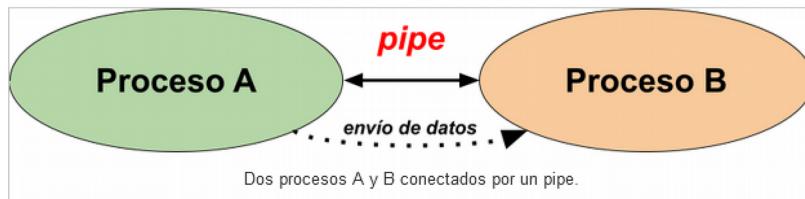
```
int a;
a=6;
```

```
Valor inicial de la variable = 6
Variable proceso hijo = ?
Variable proceso padre = ?
```

#### 4.- Comunicación entre procesos pipe(), open(), close()

Tenemos varias formas de comunicación entre procesos (Inter-Process Communication o IPC) en los sistemas Unix. *pipes, colas de mensajes, semáforos y segmentos de memoria.*

##### Los pipes (tuberías en castellano)



Cuando un proceso quiere leer del *pipe* y este está vacío, se queda esperando (**bloqueado**) hasta que algún proceso ponga datos en el *pipe*. De la misma forma si algún proceso intenta escribir en el *pipe* y este está lleno este se bloquea hasta que se vacía. El *pipe* es bidireccional pero cada proceso lo utiliza en una única dirección.

*Podemos entender el pipe como un falso fichero que conecta dos procesos, si por ejemplo A quiere enviar datos a B los escribe en el pipe. A partir de ahí el proceso B puede leer datos del pipe.*

La función **pipe()** crea una pipe.

```
#include <unistd.h>  
  
int pipe(int fd[2]);
```

*Array de 2 enteros, el primero es el descriptor para la lectura mientras que el segundo lo es para la escritura. Si la función tiene éxito devuelve 0, y los dos descriptores, si hay problemas devolverá -1.*

Para enviar datos al pipe se utiliza la función **write()** y para recuperar **read()**.

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

- **read()** intenta leer *count* bytes del descriptor *fd*, y los guarda en *buf*.
- **write()** intenta escribir *count* bytes en el descriptor *fd*, y los lee de *buf*.

##### Ejemplo

**Antes de utilizar el pipe, veamos un ejemplo de funcionamiento con archivos que utilizan las funciones *open()* y *close()*.**

Compila y prueba.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
void main(void) {
    char saludo[] = "Saludos a los alumnos!!!\n";
    char buffer[10];
    int fd, bytesleidos;

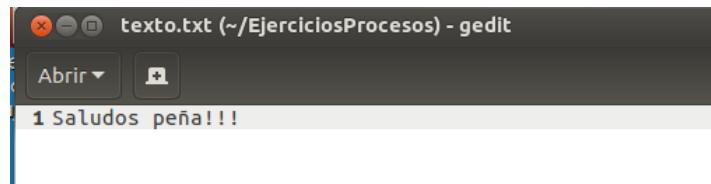
    fd = open("texto.txt",1); // abrimos para escritura
    //este fichero debe de estar creado en el directorio correspondiente
    if (fd== -1) {
        printf("Algo salió mal\n");
        exit (-1);
    }

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

    fd=open("texto.txt",0); // abrimos para lectura
    printf("Contenido del Fichero: \n");

    bytesleidos = read(fd,buffer,1);
    while (bytesleidos!=0) {
        printf("%s", buffer);
        bytesleidos = read(fd,buffer,1);
    }
    close (fd);
}
```

```
root@juan:/home/juan/EjerciciosProcesos# ./LeeEscribe
Escribo el saludo en el fichero...
Contenido del Fichero:
Saludos peña!!!
root@juan:/home/juan/EjerciciosProcesos#
```



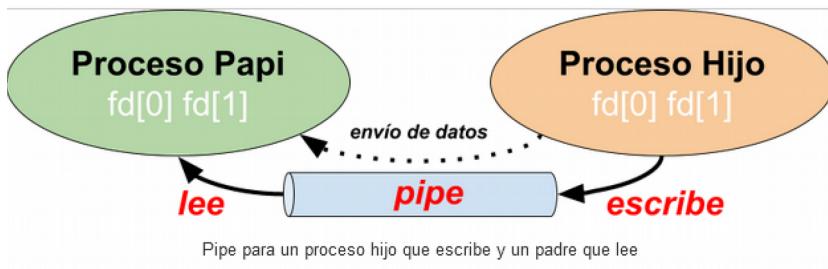
## Ejercicio 8 Realiza lo mismo que en el ejercicio 7 usando pipe

### SOLUCIÓN

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
void main(void) {
    int fd[2]; // 0 lectura y 1 escritura
    char buffer[30];
    pid_t pid;

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

    switch(pid) {
        case -1: // error
            printf("Algo falló ...");
            exit(-1);
            break;
        case 0: // soy el hijo
            printf("El hijo escribe en el pipe ...\\n");
            write(fd[1], "Hola papa",10);
            break;
        default: // soy el padre
            wait(NULL); //espero a fin del hijo
            printf("El padre lee del pipe ...\\n");
            read(fd[0], buffer, 10);
            printf("\\tMensaje recibido: %s\\n", buffer);
            break;
    }
}
```



- Los procesos padre e hijo están unidos por el *pipe* pero la comunicación sólo es en una dirección.
- Dado que los descriptores se comparten debemos estar seguros de cerrar el extremo que no nos interesa.
  - Si la información va de padre a hijo, el padre cierra ***fd[0]*** de lectura y el hijo debe cerrar el descriptor de ***escritura fd[1]***;
  - Si es al revés como nuestro ejemplo, el padre cierra la escritura ***fd[1]*** y el hijo cierra la lectura ***fd[0]***
  -

### Ejemplo, donde el padre le envía un mensaje al hijo de forma segura:

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

void main(void) {
    char saludoPadre[]="Buenos días hijo.\0";
    char buffer[30];

    int fd[2]; // 0 lectura y 1 escritura
    pipe (fd); // creamos el pipe

    pid_t pid;
    pid = fork(); // se crea el proceso hijo
    switch(pid) {
        case -1: // error printf("Algo falló ...");
                    exit(-1);
                    break;
        case 0: // soy el hijo
                    close(fd[1]); // cierra el descriptor de escritura = entrada
                    read(fd[0], buffer, sizeof(buffer));
                    printf("\tEl hijo recibe algo del pipe: %s\n",buffer);
    }
}
```

```
        break;
default: // padre envía
close(fd[0]);//cierra el descriptor de lectura = salida
write(fd[1],saludoPadre, strlen(saludoPadre)); //escribo en pipe
printf("El padre envía el mensaje al hijo ...\\n");
wait(NULL);
break;
}

}
```

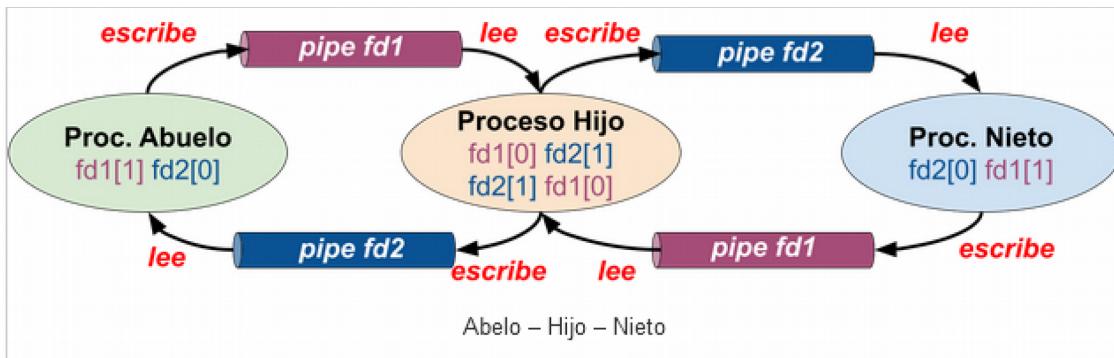
```
root@juan:/home/juan/EjerciciosProcesos# ./pipeSeguro
El padre envia el mensaje al hijo ...
    El hijo recibe algo del pipe: Buenos días hijo.@
root@juan:/home/juan/EjerciciosProcesos# 
```

### Ejercicio 9

Siguiendo el ejemplo anterior crea un programa que cree un *pipe* en el que el HIJO será el que envíe el mensaje al padre por ejemplo “*Buenos días padre*”, y el padre muestre dicho mensaje en pantalla.

## Ejemplo de comunicación entre tres procesos ABUELO, HIJO y NIETO.

Completa el código para realizar la comunicación entre los procesos.



### Ejercicio : Completa el siguiente código

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

//Abuelo-hijo-nieto

void main(void) {
    pid_t pid, Hijo_pid, pid2, Hijo2_pid;
    int fd1[2];
    int fd[2];

    // saludos de pantalla
    char saludoAbuelo[]="Saludos del Abuelo al Hijo.\0";
    char saludoPadre[]="Saludos del Hijo al Nieto.\0";
    char saludoHijo[]="Saludos del Nieto al Hijo.\0";
    char saludoNieto[]="Saludos del Hijo al Abuelo.\0";

    char buffer[80]="";

    pipe (fd1);
    pipe (fd2);

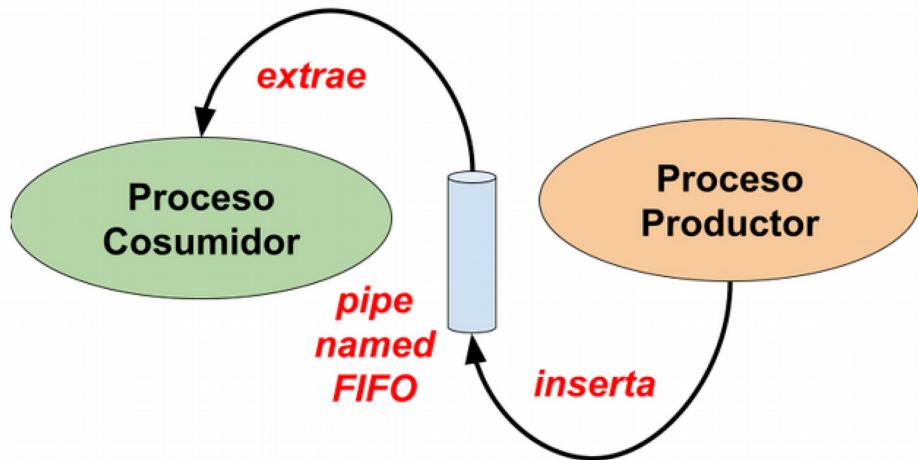
    pid = fork(); // Abuelo creando al hijo
    if (pid == -1) {
        ...
    }
    if (pid == 0) { //Soy el hijo
        // creamos al nieto
    }
}
```

```
pid2 = ...;
switch (pid2) {
    case -1:
        ...
        break;
    case 0: // Soy el nieto
        // cierra fd2 como escritura; y leo de fd2
        ...
        // Imprimo el mensaje que me mandó Hijo (mi padre)
        ...
        // Envío un mensaje a Hijo (mi padre) a través de fd1
        // con lo que tengo que cerrar fd1 como lectura
        ...
        break;
    default:
        // Soy el hijo el que tiene más faena
        // Leo lo que me manda el abuelo por fd1 y lo imprimo por pantalla
        ...
        // envio a nieto (mi hijo) por fd2 el mensaje de la variable "saludoPadre"
        ...
        // me quedo esperando a que Nieto termine (mi hijo)
        Hijo2_pid=wait(NULL);
        // Recibo el mensaje de Nieto por fd1, y lo imprimo por pantalla
        ...
        // Envío mensaje al abuelo a través de fd2
        ...
    }
} else { // Soy el abuelo
    ...
    printf("Abuelo envia mensaje al hijo ...\\n");
    // cierra fd1[0] para la lectura el abuelo lee por fd2[0]
    ...
    // escribe en fd1 saludoAbuelo, acuérdate de calcular la longitud del string.
    ...
    // EL abuelo se queda esperando a que termine el hijo
    ...
    // El abuelo recibe el mensaje por fd2, luego cierra fd2 en modo escritura
    ...
    // Lee de la pipe fd2, entendemos que su hijo le dejó un mensaje antes de terminar.
    read(fd2[0],buffer, sizeof(buffer));
    printf("El abuelo recibe el siguiente mensaje del hijo: %s\\n",buffer);

}
exit(0);
}
```

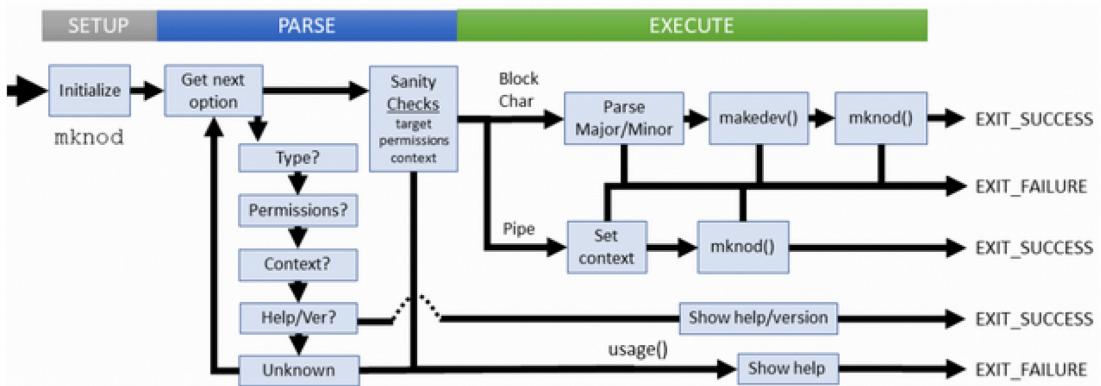
## 5.- Comunicación pipes FIFO (named pipes)

Los pipes hasta lo visto ahora, sólo crean canales entre procesos emparentados (padre-hijo). Con los **FIFO** (First In First Out) podemos comunicar procesos sin necesidad de que estén emparentados. Un vez leído un dato, éste ya no puede ser leído de nuevo.



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

**Para crear un FIFO se utiliza la función de C `mknod()`.**



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

```
#include <unistd.h>

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

#include <fcntl.h>      /* Definition of AT_* constants */
#include <sys/stat.h>

int mknodat(int dirfd, const char *pathname, mode_t mode, dev_t dev);
```

El FIFO lo podemos crear:

### 1. directamente desde la línea de comandos

mknod [opciones] nombreFichero p

### Ejemplo

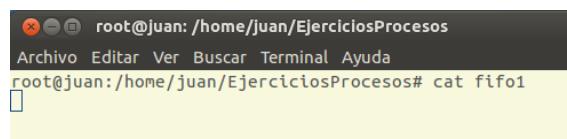
```
root@juan:/home/juan/EjerciciosProcesos# mknod fifo1 p
root@juan:/home/juan/EjerciciosProcesos# ls -l
total 152
-rwxr-xr-x 1 root root 8664 oct  6 18:33 Ejercicio1
-rw-rw-r-- 1 juan juan   88 oct  6 18:34 Ejercicio1.c
-rwxr-xr-x 1 root root 8712 oct  6 18:38 Ejercicio3
-rwxr-xr-x 1 root root 8720 oct  7 10:09 Ejercicio3b
-rw-rw-r-- 1 juan juan  231 oct  7 10:09 Ejercicio3b.c
-rw-rw-r-- 1 juan juan  817 oct  6 19:23 Ejercicio3.c
-rwxr-xr-x 1 root root 8864 oct  6 18:59 Ejercicio4
-rw-rw-r-- 1 juan juan  903 oct  6 18:59 Ejercicio4.c
-rwxr-xr-x 1 root root 8912 oct  6 19:20 Ejercicio5
-rw-rw-r-- 1 juan juan 1742 oct  6 19:20 Ejercicio5.c
-rw-r--r-- 1 root root   42 oct  6 18:33 ficherosalida.txt
-rw-r--r-- 1 root root   42 oct  6 18:46 ficherosalida.txt.save
prw-r--r-- 1 root root     0 oct  7 11:26 fifo1
```

opciones: -m ó –mode: similar chmod

### Fucionamiento:

Abrir 2 terminales, en el **primer** terminal ejecutamos ...

cat fifo1



El terminal se quedará como bloqueado, ... cat está esperando a que le llegue algo por fifo1

En un **segundo** terminal ejecutamos ...

ls > fifo1

The screenshot shows a terminal window with the following content:

```
Archivo Editar Ver Buscar Terminal Ayuda
root@juan:/home/juan/EjerciciosProcesos# cat fifo1
Ejercicio1
Ejercicio1.c
Ejercicio3
Ejercicio3b
Ejercicio3b.c
Ejercicio3.c
Ejercicio4
Ejercicio4.c
Ejercicio5
Ejercicio5.c
ficherosalida.txt
ficherosalida.txt.save
fifo1
LeeEscribe
LeeEscribe.c
PadreHijoNieto.c
phn.c
pipe1
pipe1.c
pipeSeguro
pipeSeguro.c
root@juan:/home/juan/EjerciciosProcesos#
Archivo Editar Ver Buscar Terminal Ayuda
root@juan:/home/juan/EjerciciosProcesos# ls > fifo1
root@juan:/home/juan/EjerciciosProcesos#
```

Simplemente en este segundo terminal estamos enviando un listado de ficheros a el pipe fifo1, automáticamente cuando termina el proceso en el terminal 2, la instrucción *cat* se desbloquea en el terminal 1 y mostrará el contenido de fifo1. Una vez mostrado el *pipe* se vacía.

## 2. utilizando la función C *mknod()*

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

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

**pathname:** nombre del *pipe*

**modo:**

- **S\_IFREG o 0** (un fichero normal que se crea vacío).
- **S\_IFCHR** (fichero especial de caractéres).
- **S\_IFBLK** (fichero especial de bloques). S
- **S\_IFIFO** que crea un **FIFO**.

**dev:** especifica los números mayor y menor del fichero especial, sólo se tiene en cuenta con el modo **S\_IFCHR** y **S\_IFBLK**, para los otros casos el parámetro es ignorado.

La función devolverá **0** si todo va bien y **-1** en caso de error.

**fifocrea.c**

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

int main(void) {

    int fp;
    int p, bytesleidos;
    char saludo[] = "Un saludo, alumnos!!!\n", buffer[10];
    p=mknod("FIFO2", S_IFIFO|0666,0); // permisos de lectura y escritura

    if (p== -1) {
        printf ("Ocurrió un error...\n");
        exit(0);
    }

    while(1) {
        fp = open("FIFO2",0);
        bytesleidos = read(fp, buffer, 1);
        printf("Obtenido información ... ");
        while (bytesleidos!=0){
            printf("%s",buffer);
            bytesleidos= read(fp, buffer,1);
        }
        close(fp);
    }
    return(0);
}
```

**fifoescribe.c**

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

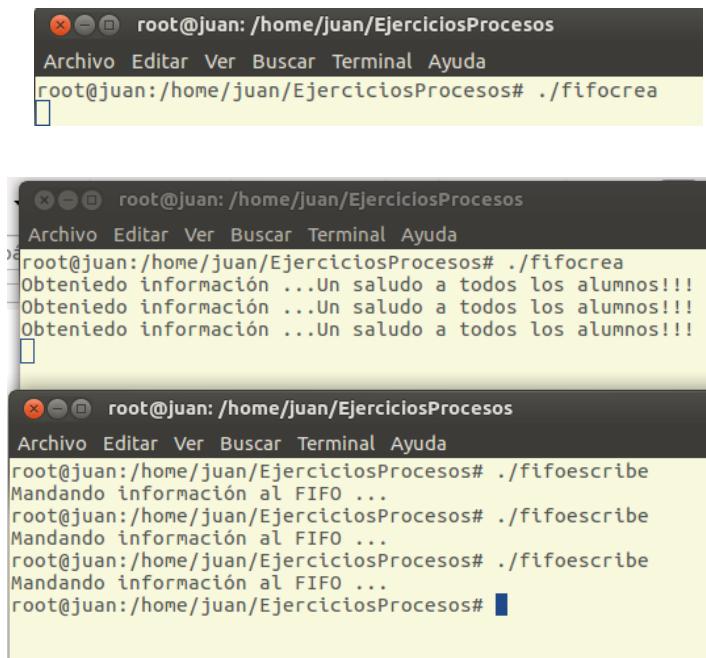
int main() {
    int fp;
    char saludo[] = "Un saludo!!!\n";
    fp = open("FIFO2",1);
```

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

1. Compilar ambos ficheros

```
root@juan:/home/juan/EjerciciosProcesos# gcc fifocrea.c -o fifocrea  
root@juan:/home/juan/EjerciciosProcesos# gcc fifoscribe.c -o fifoscribe  
root@juan:/home/juan/EjerciciosProcesos# ls  
Ejercicio1  Ejercicio4.c      fifocrea.c      pipe1  
Ejercicio1.c Ejercicio5      fifoscribe      pipe1.c  
Ejercicio3  Ejercicio5.c      fifoscribe.c    pipeSeguro  
Ejercicio3b ficherosalida.txt LeeEscribe      pipeSeguro.c
```

2. Abre dos terminal . En una de ella ejecuta primero fifocrea y a continuación fiescribe. Mira en el directorio de trabajo que no tenga creado el fifo.



## Ejercicio

Modifica **fifocrea.c** para que cuando le lleguen 5 mensajes el proceso termine, además deberás indicar el número de mensaje leído también por pantalla.

## 6.- Sincronización entre procesos **signal()**, **kill ()**, **sleep()**.

Para que los procesos interactúen se necesitan otras funciones de coordinación más interesantes, así pues vamos a hablar de las **señales**.

Una **señal** es como un aviso que un proceso manda a otro proceso, antes las señales las enviaba el Sistema Operativo para indicarnos que el proceso hijo había terminado o que alguien había escrito en el **pipe**. Los procesos no se comunicaban directamente. Ahora si es posible con la función **signal()**. (Para comunicar procesos)

Cada señal tiene un nombre que comienza por SIG...

El archivo <[signum.h](#)> contiene la definición de todas las señales

Numerosas condiciones pueden generar señales. Entre ellas:

- Presionando la tecla DELETE (también control-C)
- Excepción de hardware: División por cero, referencia de memoria invalida (SIGSEGV)
- kill : existe como función y como comando. El procesador que envía debe pertenecer al mismo usuario del que recibe o ser superusuario.
- Condiciones de software. Por ejemplo: SIGALRM generada por la alarma del reloj cuando expira. SIGPIPE: (cuando se escribe en una "pipe" que ha terminado por el lado lector).

Hay **tres cosas que un programa puede solicitar al kernel como acción** cuando llegue una señal:

1. Ignorar la señal. Sin embargo hay dos señales que no pueden ser ignoradas: SIGKILL y SIGSTOP.
2. Capturar la señal. El kernel llama una función dentro del programa cuando la señal ocurre.
3. Permitir que se aplique la acción por defecto. Esta puede ser: terminal el proceso, terminar con un core dump, parar el proceso, o ignorar la señal.

**Algunas señales:**

- SIGALRM: generada cuando el timer asociado a la función *alarm* expira. También cuando el timer de intervalo es configurado (*setitimer*)
- SIGCHLD: Cuando un proceso termina o para, el proceso envía esta señal a su padre. Por defecto esta señal es ignorada. Normalmente el proceso padre invoca la función wait para obtener el estatus de término del proceso hijo. Se evita así la creación de procesos "zombies".
- SIGCONT: es enviada para reanudar un proceso que ha sido parado (suspendido) con SIGSTOP.
- SIGINT: generada con DELETE o Control-C
- SIGKILL: Permite terminar un proceso.
- SIGTSTP: generada cuando presionamos Control-Z. Puede ser ignorada.
- SIGSTOP: similar a SIGTSTP pero no puede ser ignorada o capturada.

- SIGUSR1: Es una señal definida por el usuario para ser usada en programas de aplicación.
- SIGUSR2: Otra como la anterior.

Para probar las señales se recomienda usar:

% kill -SIG??? <pid>

## Función signal

```
#include <signal.h>  
  
void (*signal (int señal, void (*Func) (int)))(int);
```

*Recibe 2 parámetros:*

**señal:** Es el nombre de la señal. Contiene el número de señal que queremos capturar, en nuestro caso **SIGUSR1**. Si utilizáramos **SIGKILL** estaríamos utilizando una señal para terminar procesos.

**Func:** Puede ser la constante **SIG\_IGN** para ignorar la señal, **SIG\_DFL** para usar una acción por defecto, o puede ser una función definida por el programa. A esta función se le conoce como **el manejador de la señal (signal handler)**. El valor devuelto es un puntero al manejador previamente instalado para esa señal.

Un ejemplo de uso de la función: `signal(SIGUSR1, gestion_padre);` significa que cuando un proceso (el padre en este caso) recibe la señal **SIGUSR1** debe realizar una llamada a la función `gestion_padre()`.

En el ejemplo definiremos un manejador para el proceso padre y otro para el hijo.

## **Funciones alarm y pause**

La función **alarm** permite especificar un timer que expire en un tiempo dado.

La función **pause** suspende al proceso llamador hasta que llegue una señal.

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

```
unsigned int alarm(unsigned int seconds);
```

En nuevas versiones de sistema operativo también se puede usar

```
unsigned int ualarm(unsigned int microseconds);  
  
int pause(void); /* retorna -1 con errno en EINTR */
```

El control del tiempo no es exacto por la incertidumbre del kernel, carga del sistema, etc.

Hay sólo una alarma por proceso!

La alarma se puede cancelar con seconds=0 en el argumento. El valor returned es lo que falta para que el reloj expire.

## Función sleep

*sleep suspende al proceso llamador por la una cantidad de segundos indicada o hasta que se reciba una señal.*

### ¿Cómo se envía la señal?

Para enviar una señal utilizamos la función **kill()** , **raise()**

**kill** envía us señal a un proceso o a un grupo de procesos. **raise** permite enviar señales al mismo proceso (hacia uno mismo).

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int signo);
```

Pid es el PID del proceso que recibe la señal y la propia señal.

```
kill(pid_hijo, SIGUSR1);
```

```
int raise(int signo);
```

Este último es como `kill( getpid(), signo);`

### ¿Cómo hace un proceso para esperar una señal?

Los procesos pueden esperar con la función **pause()**.

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

## Ejercicio: Ejemplo de signal()

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

void manejador(int);
```

```
int main() {
    int pid_hijo;
    pid_hijo = fork();

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

void manejador(int sig) {
// haz algo

    printf("He recibido la señal perfecta: %d\n",sig);
}
```

```
root@juan:/home/juan/EjerciciosProcesos# ./sincroniza1
He recibido la señal perfecta: 10
He recibido la señal perfecta: 10
root@juan:/home/juan/EjerciciosProcesos#
```

### Ejercicio : Programa que trata la señal SIGINT generada al pulsar las teclas Control+C

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

void manejador_sigint(int);
int main(void) {
    if (signal (SIGINT, manejador_sigint) == SIG_ERR)
    {
        perror("Error en signal");
        exit (-1);
    }
    while (1)
```

```
{  
    printf ("Esperando un Ctrl-C\n");  
    sleep(999);  
}  
}  
  
void manejador_sigint(int sig){  
    printf ("Señal numero %d recibida.\n", sig);  
}
```

```
root@juan:/home/juan/EjerciciosProcesos# ./sincroniza2  
Esperando un Ctrl-C  
^CSeñal numero 2 recibida.  
Esperando un Ctrl-C  
^CSeñal numero 2 recibida.  
Esperando un Ctrl-C
```

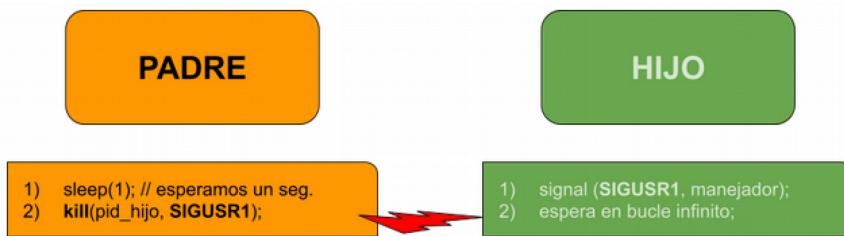
## Ejemplo de utilización de pause

pause hace que el proceso se quede esperando la llegada de alguna señal. Cuando esto ocurre, y tras ejecutar la rutina de tratamiento de la señal, devuelve -1 y en perror coloca el valor EINTR, para indicar que se ha producido una interrupción de la llamada. El programa continúa con la sentencia siguiente a pause.

```
#include <stdio.h>  
#include <stdlib.h>  
#include <signal.h>  
#include <errno.h>  
void manejador_sigusr1(int);  
  
void manejador_sigterm(int);  
int main(void){  
if      (signal(SIGTERM,      manejador_sigusr1)==      SIG_ERR      ||signal(SIGUSR1,  
manejador_sigterm)== SIG_ERR)  
{  
    perror("Manejador no asignado");  
    exit(1);  
}  
while (1){  
    pause();  
}  
}  
  
void manejador_sigterm(int sig){  
    printf("El usuario pide terminar %ld\n", (long)getpid());  
    exit (-1);
```

}

```
void manejador_sigusr1(int sig){  
    signal(sig, SIG_IGN);  
    printf ("%d\n", rand());signal (sig, manejador_sigusr1);  
}
```



## Ejercicio

Un proceso padre se ejecutan de forma sincrona. Se definen dos funciones para gestionar la señal una para el padre y otro para el hijo, serán simplemente mensajes por pantalla.

### PADRE

- El padre crea al hijo
- Cada proceso realizará una señal signal decir lo que tiene que hacer cuando reciba la señal
- El padre se queda en un bucle esperando recibir la señal.
- Cuando el padre recibe la señal ejecutará: *gestion\_padre()*
- Con *kill()* envía la señal de respuesta al proceso hijo mediante su PID.
- El proceso se repite indefinidamente.

### HIJO

- El proceso hijo inicia la comunicación con el padre por lo que utilizará *pause()* al final del código y enviará anteriormente la señal *kill()*.
- Cuando reciba la señal del padre deberá ejecutar la función *gestion\_hijo()*;

## SOLUCIÓN

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

void gestion_padre (int senyal) {
    printf ("Padre recibe señal ...%d\n", senyal); }

void gestion_hijo (int senyal) {
    printf ("Hijo recibe señal ... %d\n", senyal);
}

int main () {
    int pid_padre, pid_hijo;
    pid_padre =getpid();
    pid_hijo = fork();

    switch(pid_hijo) {
        case -1:
            printf("Error al crear el proceso hijo...\n");
            exit(-1);
        case 0: // fill
            // tratamiento de la señal
            signal(SIGUSR1, gestion_hijo);
            // bucle infinito
            while(1) {
                // dentro del bucle cada segundo enviaremos una señal al padre
                sleep(1);
                kill(pid_padre, SIGUSR1);
                // y dentro también esperaremos "pause()" a que llegue una señal del padre
                pause();
            }
            break;
        default: // papi
            //tratamiento de la señal
            signal(SIGUSR1, gestion_padre);
            //bucle infinito
            while(1) {
                //aquí lo primero que hacemos es esperar la señal del hijo
                pause();
                //una vez recibida esperamos 1 seg. y enviamos señal al hijo
                sleep(1);
            }
    }
}
```

```
    }  
    break;  
}  
return 0;  
  
}
```