

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

Unidad Didáctica 01: Programación multiproceso

Demostración: Programación multiproceso en C

Resumen de planificación de actividad		
Objetivos	Contenidos que se trabajan	
<ul style="list-style-type: none">• Poner en práctica la creación y sincronización de procesos en sistemas Linux mediante el lenguaje de programación C	<ul style="list-style-type: none">• Ejecutables. Procesos. Servicios.• Programación concurrente.• Creación de procesos.• Comunicación entre procesos.• Gestión de procesos.• Comandos para la gestión de procesos en sistemas libres y propietarios.• Sincronización entre procesos.• Programación de aplicaciones multiproceso.• Documentación.• Depuración.	
Objetivos didácticos	Criterios de evaluación	
RA1: Desarrolla aplicaciones compuestas por varios procesos reconociendo y aplicando principios de programación paralela.	<p>c) Se han analizado las características de los procesos y de su ejecución por el sistema operativo.</p> <p>e) Se han utilizado clases para programar aplicaciones que crean subprocesos.</p> <p>f) Se han utilizado mecanismos para sincronizar y obtener el valor devuelto por los subprocesos iniciados.</p> <p>g) Se han desarrollado aplicaciones que gestionen y utilicen procesos para la ejecución de varias tareas en paralelo.</p> <p>h) Se han depurado y documentado las aplicaciones desarrolladas.</p>	
Recursos necesarios	Agrupamiento	Duración estimada
<ul style="list-style-type: none">• Ordenador Ubuntu con acceso a Internet	Grupo clase	1 sesión
Secuencia de desarrollo de la actividad		
<ol style="list-style-type: none">1. Planteamiento y objetivos.2. Explicación guiada de cada ejemplo3. Resolución de dudas		

Demostración: Programación multiproceso en C

INSTRUCCIONES

Esta demostración se hará de forma individual.

TALLER 1: CREACIÓN DE PROCESOS EN LINUX

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

system

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 intérprete de comando en el que esté ejecutándose. devolverá -1 si hay error y el estado en caso contrario.

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

Esta función no debe usarse desde 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.

Abre el fichero `ejemploSystem.c` con un editor de texto plano y revisa el código

Compila `ejemploSystem.c` con el nombre de ejecutable `proceso1`:

```
gcc ejemploSystem.c -o ejemploSystem
```

Ejecuta `ejemploSystem`.

execl

execl() realiza la ejecución y terminación del comando. Con `exec` se reemplazará la imagen de proceso actual con una nueva imagen de proceso. La nueva imagen se construirá a partir de un archivo ejecutable indicado como parámetro.

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);
```

Hay otras variantes de `exec` como `execle`, `execlp`, `execv`, `execve`, `execvp`

Abre el `ejemploExec.c` con un editor de texto plano y revisa el código

Compila `ejemploExec.c` con el nombre de ejecutable `ejemploExec`:

```
gcc ejemploExec.c -o ejemploExec
```

Ejecuta `ejemploExec`.

Ejercicios

Ejercicio 1: Modifica el código de proceso 2 para que se muestren también los ficheros ocultos (argumento `-a`)

Ejercicio 2: Crea un programa en C que muestre todos los procesos en ejecución.

fork()

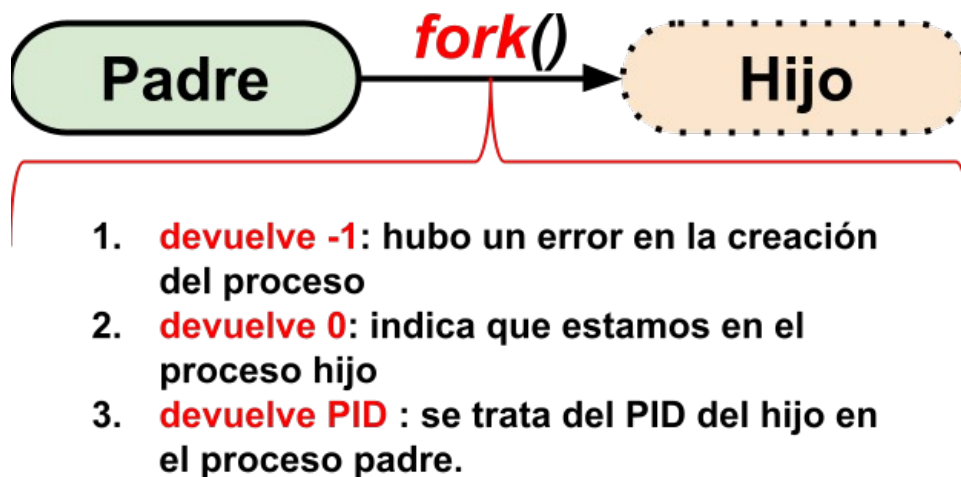
Hasta ahora hemos ejecutado procesos partiendo de ejecutables en el disco duro. ¿Qué pasa si queremos crear un proceso sin que parta de un ejecutable en disco?

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.

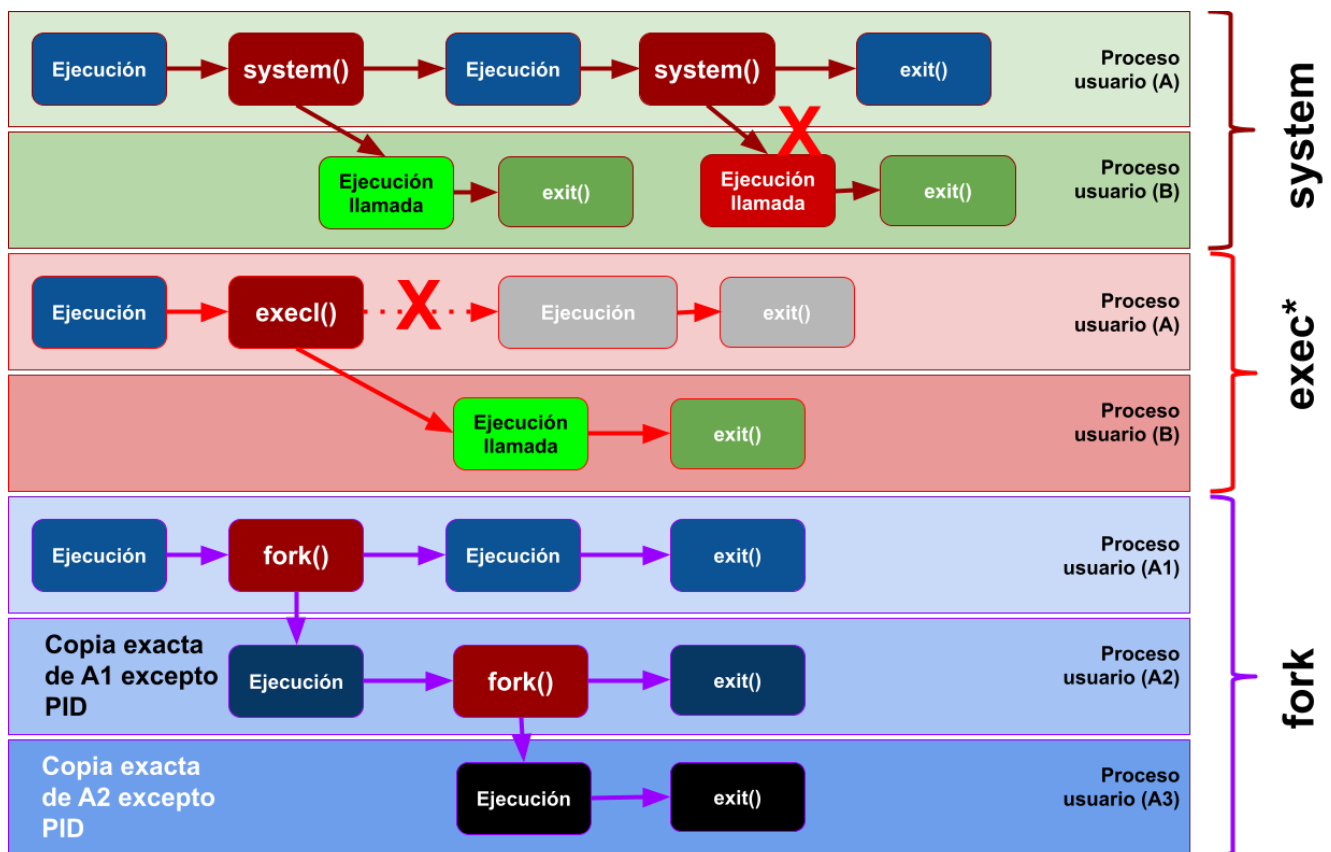
`fork()` crea nuevos procesos sin ningún tipo de parámetro, se encuentra también en la librería `unistd.h`.
`pid_t fork(void);`

Devuelve:

- -1 : si se produce algún error.
- 0: si no se produce ningún error y estamos en el proceso hijo.
- PID asignado al proceso hijo, si todo ha ido bien y estamos en el proceso padre.



Aquí tienes un resumen de los diferentes comportamientos de cada función:



Procesos con getpid(), getppid(), fork(), wait()

Otras funciones de C son útiles para trabajar con procesos:

```
getpid(): devuelve el pid (identificador del proceso) del proceso actual (el que ejecuta la llamada)
getppid(): devuelve el identificador del proceso PADRE del actual
```

Vamos a probarlas:

```
Abre el ejemploPadres.c con un editor de texto plano y revisa el código
Compila ejemploPadres.c con el nombre de ejecutable ejemploPadres:
gcc ejemploPadres.c -o ejemploPadres
Ejecuta ejemploPadres.
```

Como vemos si ejecutamos un ps el ppid de nuestro proceso es el propio bash

Ahora usaremos **fork** para crear procesos:

```
Abre el ejemploFork.c con un editor de texto plano y revisa el código
Compila ejemploFork.c con el nombre de ejecutable ejemploFork:
gcc ejemploFork.c -o ejemploFork
Ejecuta ejemploFork.
```

Con fork el proceso padre creará un proceso hijo y no se detendrá, es decir tanto el proceso padre como el hijo seguirán ejecutándose concurrentemente en la línea de código siguiente al fork.

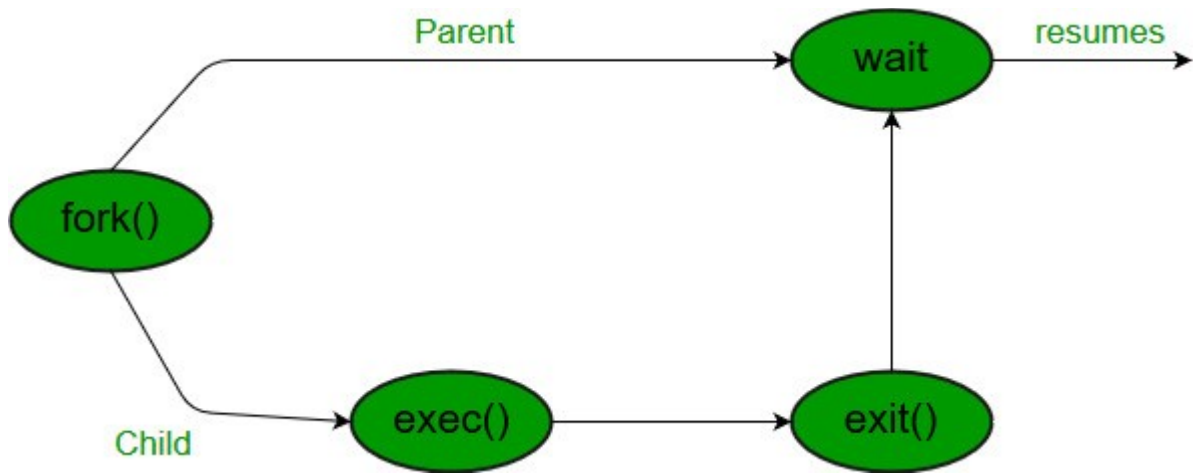
En ocasiones el proceso padre puede finalizar su ejecución antes que el proceso hijo con lo que el proceso hijo se queda **huérfano**. Cuando esto ocurre, el proceso del S.O. adopta todos los procesos hijos del padre que acaba de terminar. Cuando un proceso ha sido adoptado por el S.O., su ppid es 1.

Los hijos también pueden tener sus propios hijos.

```
Abre el ejemploMultipleFork.c con un editor de texto plano y revisa el código
Revisa el código, ¿Cuántos hola mundo crees que saldrán?
Vamos a comprobarlo:
Compila ejemploMultipleFork.c con el nombre de ejecutable ejemploMultipleFork:
gcc ejemploMultipleFork.c -o ejemploMultipleFork
Ejecuta ejemploMultipleFork.
```

Si queremos que el proceso padre se detenga esperando a que todos sus hijos finalicen su ejecución, hemos de emplear la función **wait**.

```
wait (int* status)
```



Wait en caso de éxito, devuelve el ID del proceso hijo que ha finalizado, en caso de error, se devuelve -1.

En el argumento si se le pasa un puntero a entero, se almacenará en él un valor que indica es código de estado de salida del proceso hijo. Si fuesen varios procesos hijos, se devuelve el estado de uno de ellos (es arbitrario). Si no se necesita esta información, se ha de usar wait(NULL).

Un proceso hijo puede terminar debido a cualquiera de estos motivos:

- Llama a exit ();
- Devuelve (un int) de la función main
- Recibe una señal (del sistema operativo u otro proceso) cuya acción predeterminada es terminar.

Ahora usaremos **fork** en combinación con **wait** para crear procesos:

Abre el ejemploForkWait.c con un editor de texto plano y revisa el código
 Compila ejemploForkWait.c con el nombre de ejecutable ejemploForkWait:
`gcc ejemploForkWait.c -o ejemploForkWait`
 Ejecuta ejemploForkWait.

Como puedes deducir, en la creación de procesos se define una jerarquía.



Veamos ahora a un ejemplo de creación de un proceso que cree un hijo y éste a su vez cree otro proceso (podemos decir que nieto del primero):

Abre el ejemploPadreAbuelo.c con un editor de texto plano y revisa el código
 Compila ejemploPadreAbuelo.c con el nombre de ejecutable ejemploPadreAbuelo:
`gcc ejemploPadreAbuelo.c -o ejemploPadreAbuelo`
 Ejecuta ejemploPadreAbuelo.

Ejercicios

Ejercicio 3. 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 el 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.

Valor inicial de la variable = 6

Variable proceso hijo = ?

Variable proceso padre = ?

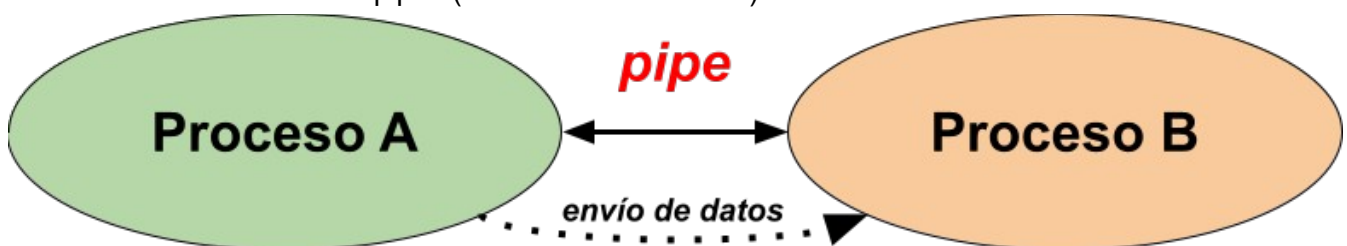
TALLER 2: COMUNICACIÓN ENTRE PROCESOS

Hasta ahora hemos conseguido que nuestro programa cree varios procesos aunque cada uno de estos procesos “va a su bola”. Para conseguir funcionalidades más sofisticadas se hacen necesarios **mecanismos de comunicación** entre los procesos.

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.

Comunicación entre procesos con pipes

A continuación veremos los pipes (tuberías en castellano)



Un pipe es como un falso fichero que sirve para comunicar dos procesos, de forma que lo que uno de los procesos escribe el otro es capaz de leerlo. Si por ejemplo el proceso A quiere enviar datos al proceso B los escribe en el pipe. A partir de ahí el proceso B puede leer datos del pipe.

Los pipes se usan comúnmente en sistemas Unix en las herramientas en línea de comandos con el símbolo |, por ejemplo:

```
ls -l | grep mifichero
```

Se han de tener en cuenta dos aspectos del modo de funcionamiento de un pipe:

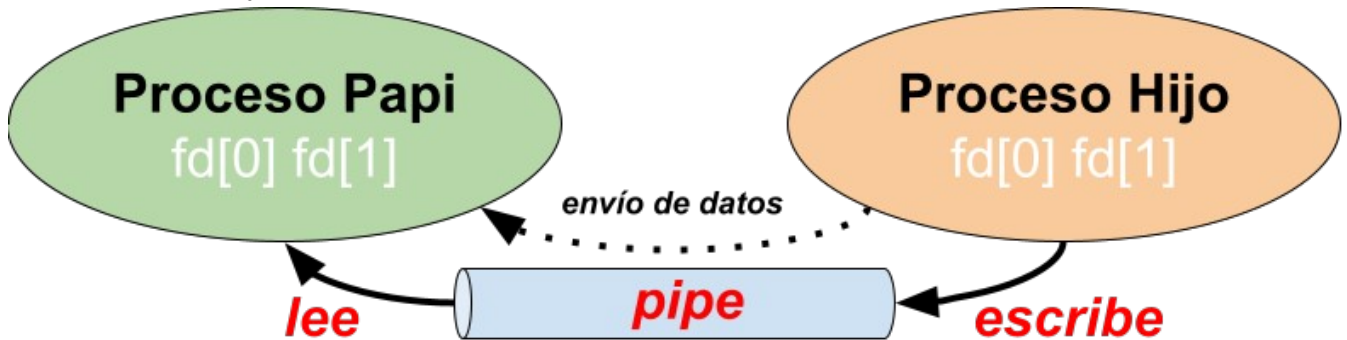
- Cuando un proceso quiere leer del pipe y este está vacío, se queda esperando (bloqueado) hasta que otro proceso ponga datos en el pipe.
- De la misma forma si algún proceso intenta escribir en el pipe y este está lleno, el proceso se bloquea hasta que el pipe vacía.
- El pipe es bidireccional pero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

La función **pipe()** crea una pipe.

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

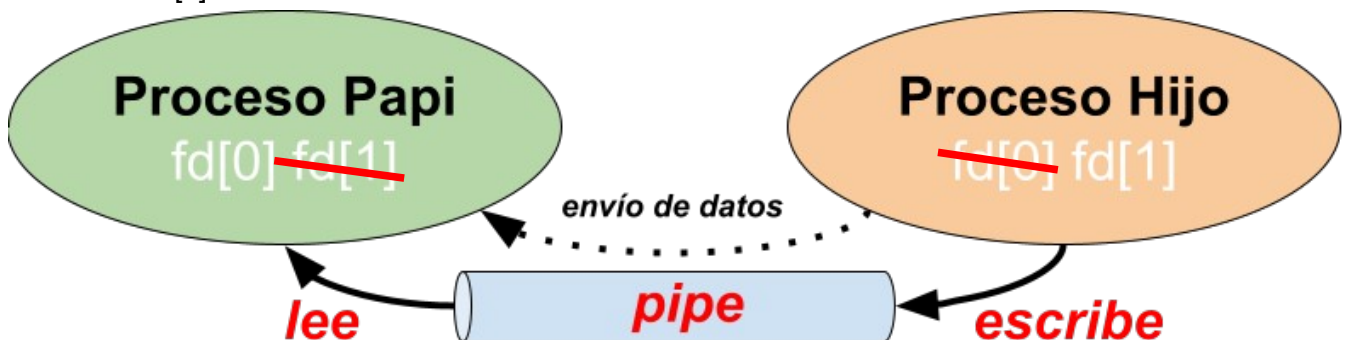
```
int pipe(int fd[2]);
```

El parámetro `fd` es un array de 2 enteros, el primero (`fd[0]`) es el descriptor para la lectura mientras que el segundo (`fd[1]`) lo es para la escritura. Si la función tiene éxito devuelve 0, y los dos descriptors, si hay problemas devolverá -1.



Recordemos que los procesos padre e hijo están unidos por el pipe pero la **comunicación sólo es en una dirección**. Dado que los descriptors se comparten entre ambos procesos debemos estar seguros de cerrar el extremo que no nos interesa. Por ejemplo:

- 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 anterior, el padre cierra la escritura `fd[1]` y el hijo cierra la lectura `fd[0]`.



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`. Si hay menos bytes no pasa nada, leerá lo que haya. Recordemos que la lectura produce que el proceso se bloquee hasta que llegue la información.
- **write()** intenta escribir el contenido de `buf` en el descriptor `fd`, y los lee de `buf`. En `count` se le ha de indicar el número de bytes que tiene `buf`.

Abre el `ejemploPipe.c` con un editor de texto plano y revisa el código
Compila `ejemploPipe.c` con el nombre de ejecutable `ejemploPipe`:
`gcc ejemploPipe.c -o ejemploPipe`
Ejecuta `ejemploPipe`.

Ejercicios

Ejercicio 4. 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.

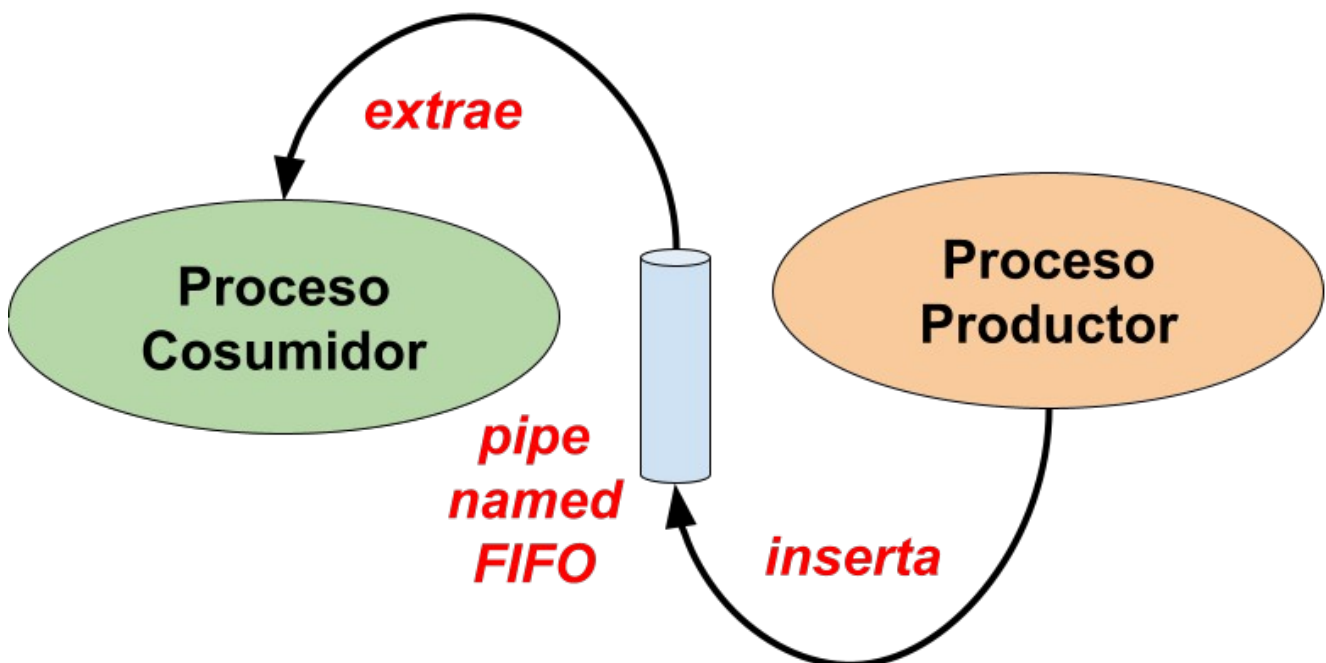
Ejercicio 5. Crea un programa en el que se comuniquen de forma similar tres procesos ABUELO, HIJO y NIETO. Recuerda que cada proceso cierre los pipes que no va a usar.

- El abuelo mandará un mensaje a su hijo: "Saludos del Abuelo al Hijo"
- El hijo mandará un mensaje a su hijo: "Saludos del Hijo al Nieto"
- El nieto mandará un mensaje a su padre (el hijo) "Saludos del Nieto al Hijo"
- El hijo mandará un mensaje a su padre (el abuelo) "Saludos del Hijo al Abuelo"

Comunicación entre procesos con pipes FIFO (named pipes)

Hasta ahora hemos comunicado procesos emparentados (padres e hijos), ¿pero que ocurre si queremos comunicar procesos que no lo estén?

Con los **pipes con nombre o FIFO** (First In First Out) podemos comunicar procesos sin necesidad de que estén emparentados. Los datos se escriben como en una cola (primero en entrar – primero en salir) y una vez leído un dato, éste ya no puede ser leído de nuevo.



Los named pipes se pueden usar desde el shell. Para crearlos se usa el comando **mkfifo**. Vamos a demostrarlo con dos terminales (cada uno de ellos es un proceso diferente).

- En el terminal de la izquierda creamos un named pipe llamado **fifo1**
- En el terminal de la derecha nos quedamos leyendo, esperando que cuando llegue algo se imprima por la pantalla con el comando **cat**.


```
ruben@Ubuntu-R530: ~/psp/taller02
ruben@Ubuntu-R530:~/psp/taller02$ mkfifo fifo1
ruben@Ubuntu-R530:~/psp/taller02$ ls -l fifo1
prw-rw-r-- 1 ruben ruben 0 sep 23 12:24 fifo1
ruben@Ubuntu-R530:~/psp/taller02$

ruben@Ubuntu-R530: ~/psp/taller02
ruben@Ubuntu-R530:~/psp/taller02$ cat fifo1
```

Ahora escribimos algo en el named pipe que recibirá el proceso que está leyendo de ahí, o sea el terminal de la derecha:

```
ruben@Ubuntu-R530: ~/psp/taller02
ruben@Ubuntu-R530:~/psp/taller02$ mkfifo fifo1
ruben@Ubuntu-R530:~/psp/taller02$ ls -l fifo1
prw-rw-r-- 1 ruben ruben 0 sep 23 12:24 fifo1
ruben@Ubuntu-R530:~/psp/taller02$ echo "hola, que tal por ahi?" > fifo1
ruben@Ubuntu-R530:~/psp/taller02$



ruben@Ubuntu-R530: ~/psp/taller02
ruben@Ubuntu-R530:~/psp/taller02$ cat fifo1
hola, que tal por ahi?
ruben@Ubuntu-R530:~/psp/taller02$
```

Desde C podemos usar también **mkfifo**:

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

- **path**: ruta del fichero especial named pipe, si quieres usar el directorio actual simplemente pon el nombre que le quieres dar al named pipe
- **mode**: modificadores, para crear un named fifo en el que se pueda leer y escribir tendrás que indicar S_IFIFO junto con alguno o varios de las siguientes constantes (que puedes encontrar en sys_stat.h). Por ejemplo S_IFIFO | S_IRWXU crea un named fifo con permisos de lectura, escritura y ejecución para el usuario propietario.

Name	Numeric Value	Description
S_IRWXU	0700	Read, write, execute/search by owner.
S_IRUSR	0400	Read permission, owner.
S_IWUSR	0200	Write permission, owner.
S_IXUSR	0100	Execute/search permission, owner.
S_IRWXG	070	Read, write, execute/search by group.
S_IRGRP	040	Read permission, group.
S_IWGRP	020	Write permission, group.

S_IXGRP	010	Execute/search permission, group.
S_IRWXO	07	Read, write, execute/search by others.
S_IROTH	04	Read permission, others.
S_IWOTH	02	Write permission, others.
S_IXOTH	01	Execute/search permission, others.
S_ISUID	04000	Set-user-ID on execution.
S_ISGID	02000	Set-group-ID on execution.
[XSI]  S_ISVTX	01000	On directories, restricted deletion flag. 

- **Devolución:** 0 si funcionó ok o un código de error en otro caso.

Vamos a probar los named pipes:

Abre un terminal y compila fifoCrea.c con el nombre de ejecutable fifoCrea y fifoEscribe.c con el nombre de ejecutable fifoEscribe
 Ejecuta fifoCrea en un terminal
 Abre otro terminal y ejecuta fifoEscribe

Ejercicios

Ejercicio 6: Modifica fifoCrea.c para que cuando le lleguen 5 mensajes el proceso termine, además deberás indicar el número de mensajes leído también por pantalla.

TALLER 3: SINCRONIZACIÓN ENTRE PROCESOS

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, con lo que hemos visto hasta ahora, 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().

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

Recibe 2 parámetros:

- **señal:** contiene el número de señal que queremos capturar. Podemos usar el valor predefinido SIGUSR1 que está definida para ser usada en aplicaciones de usuario. Un ejemplo de señal del sistema es SIGKILL cuya finalidad es terminar procesos.
- **func:** contiene la función que queremos que se llame, es decir el manejador de la señal (signal handler). Ha de devolver void y recibir un int como argumento.

- La función `signal` devolverá un puntero al manejador previamente creado para esa señal.

Un ejemplo de uso de la función:

```
signal(SIGUSR1, gestion_senyal);
```

Significa que cuando el proceso reciba la señal `SIGUSR1` la gestionará mediante una llamada a su función `gestion_senyal()`.

¿Cómo se envía la señal?

Para enviar una señal utilizamos la función `kill()`.

```
#include <signal.h>
int kill (int pid, int Senyal);
```

- **pid** es el PID del proceso al que se envía la señal
- **señal** es el código de la propia señal.

¿Cómo hace un proceso para esperar una señal?

Los procesos pueden esperar con la función `pause()` a que llegue una señal enviada por otro proceso.

```
#include <unistd.h>
int pause (void);
```

Ejemplos

Primero veremos un ejemplo simple de sincronización mediante señales.

```
Abre el ejemploSincronizar.c con un editor de texto plano y revisa el código
Compila ejemploSincronizar.c con el nombre de ejecutable ejemploSincronizar:
gcc ejemploSincronizar.c -o ejemploSincronizar
Ejecuta ejemploSincronizar
Demuestra que el proceso hijo se ha quedado en ejecución
```

En este ejemplo hemos usado la función `sleep` para forzar a que un proceso esté detenido durante un tiempo determinado.

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

La sincronización con las señales puede ir en ambos sentidos con lo que podemos conseguir que un proceso padre y otro hijo se envíen señales entre ellos y consigamos un comportamiento sincronizado entre ambos. En el siguiente ejemplo se definen dos funciones para gestionar la señal una para el padre y otra para el hijo, serán simplemente mensajes por pantalla.

```
Abre el ejemploSincronizar2.c con un editor de texto plano y revisa el código
Compila ejemploSincronizar2.c con el nombre de ejecutable ejemploSincronizar2:
gcc ejemploSincronizar2.c -o ejemploSincronizar2
```

Ejercicios

Ejercicio 7. Modifica ejemploSincronizar.c para que el proceso hijo finalice su ejecución cuando reciba dos señales.

Ejercicio 8. Crea un programa en el que se establezca un diálogo entre los procesos padre e hijo dando un significado a cada señal:

- SIGUSR1 del hijo al padre será el mensaje: "Me compras un coche?"
- SIGUSR1 del padre al hijo será la respuesta: "No"
- SIGUSR2 del padre al hijo será la respuesta: "Venga, si lo apruebas todo.."

El padre responderá en comprarle el coche si lo aprueba todo hijo tras 10 preguntas del hijo. Tras responder esto el proceso padre terminará.

El proceso hijo terminará cuando el padre le prometa comprarle el coche si lo aprueba todo.