

# Práctica 4

## Comunicación entre procesos.

### Comunicación entre procesos.

Existen varias formas de comunicación entre procesos de Linux: pipes, colas de mensajes, semáforos y segmentos de memoria compartida.

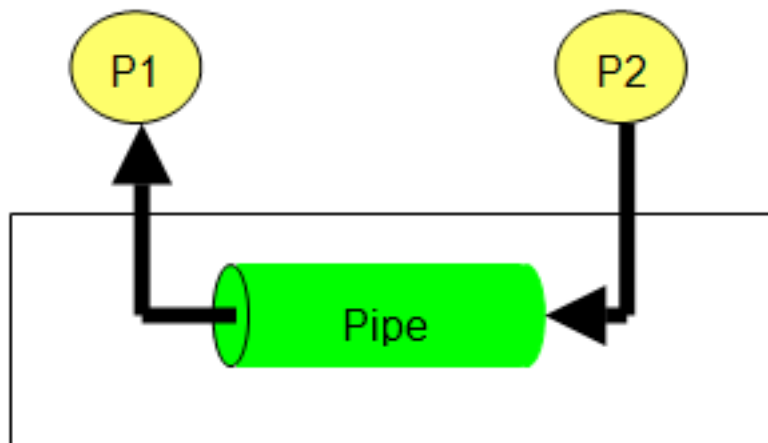
#### Cuestión 1

Averigua qué son y cómo funcionan:

- Semáforos
- Monitores.

Nos centramos en los **pipes** (tuberías en castellano).

¿Qué es un **Pipe**? Es una especie de falso fichero que sirve para conectar dos procesos.



Cuando P2 quiere enviar datos a P1, los escribe en el pipe como si fuera un fichero de salida, el proceso P1 puede leer los datos sin más que leer el pipe como si se tratará de un fichero de entrada. (comunicación similar a lectura y escritura en ficheros normales).

## ¿Cómo creamos un pipe?

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

Observa que la función sólo recibe fd[2], un array de dos enteros.

fd[0] Contiene descriptor lectura.

fd[1] Contiene descriptor escritura.

Si la función tiene éxito devuelve 0, si ocurre algún error devuelve -1

## ¿Cómo escribir y leer del pipe?

```
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. Devolverá el número de bytes leídos.
- **write()** A buf le damos el valor que queramos escribir, definimos su tamaño en count y especificamos el fichero en el que vamos a escribir en fd.

Vamos a ver un ejemplo que nos quedará más claro del **uso de ficheros en C** (recuerda en C usamos open() y close() para abrir y cerrar archivos)

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

void main(void)
{
    char saludo[]="Un saludo....";
    char buffer[10];
    int fd, bytesleidos;

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

    //controlo error en abertura fichero
    if (fd==-1)
    {
        printf ("ERROR al abrir fichero");
        exit(-1);
    }

    // escribo saludo en fichero y cierro
    printf ("escribo el saludo..");
```

```
write (fd,saludo,strlen(saludo));
close(fd);

fd=open("texto.txt",0); // fichero abierto 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); //escribo byte leído
    bytesleidos =read(fd,buffer,1); //leo otro byte
}

close(fd);
}
```

Una vez hemos recordado como escribir fichero veamos ejemplos con pipes.

### Ejemplo

Creamos proceso con fork(). El proceso hijo envía al proceso padre mediante el uso de pipes el mensaje “hola papi” en el descriptor para escritura fd[1], el proceso padre mediante el descriptor fd[0] lee los datos enviados por el hijo.

```
#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(); //creo proceso hijo

    switch(pid)
    {
        case -1: //error
            printf("no se ha podido crear el hijo\n");
            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); //padre espera hijo termine escribir
    printf("el padre lee del pipe...\n");
    read (fd[0], buffer ,10);
    printf ("\tMensaje leído:%s\n",buffer);
    break;
} //fin switch

```

La compilación y ejecución:

```

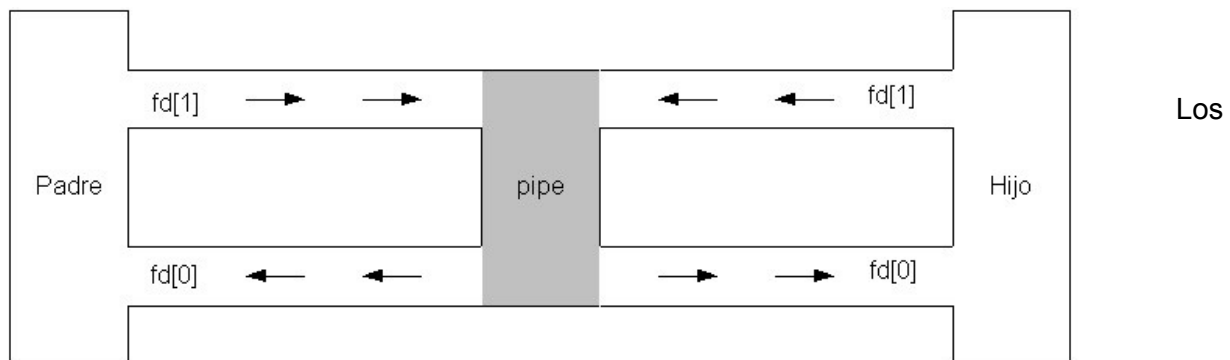
david@david-OEM ~/pss $ ./ejemplopipe
el hijo escribe en el pipe...
el padre lee del pipe...
    Mensaje leído:Hola papi

```

El proceso que sigue el ejemplo es:

1. Crea tubería con pipe()
2. Crea proceso hijo.

Cuando creamos proceso con fork(), el hijo recibe copia de todos los descriptores de ficheros del proceso padre (incluyendo fd[0] y fd[1]). Esto nos permite que el proceso hijo escriba en el extremo de escritura fd[1] y el padre los reciba en el extremo de lectura fd[0]



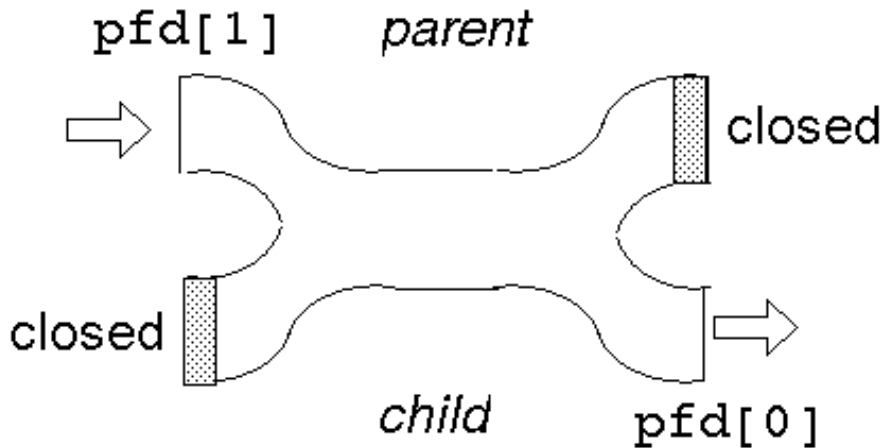
procesos padre e hijo están unidos por el pipe (ver figura anterior), pero la comunicación es en una única dirección, por tanto se debe decidir en qué dirección se envía la información, del padre al hijo o del hijo al padre. Al compartirse los descriptores debemos estar seguros de cerrar el extremo que no nos interesa.

Flujo información de padre a hijo

- El padre debe cerrar el descriptor de lectura fd[0] (no va a usar)
- El hijo debe cerrar el descriptor de escritura fd[1] (no va a usar)

Flujo información de hijo a padre

- El padre debe cerrar el descriptor de lectura `fd[1]` (no va a usar)
- El hijo debe cerrar el descriptor de escritura `fd[0]` (no va a usar)



Vamos a implementar la comunicación de la imagen anterior donde creamos 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 el `fd[1]`. El padre escribe en `pdf[1]` y el hijo lee de `pdf[0]`.

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

```
int main()
{
    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 en el pipe
    printf ("\El hijo recibe algo del pipe.%s\n",buffer);
    break;
default: //padre envia
    close (fd[0]); //cierra el descriptor de entrada
    write (fd[1],saludoPadre, strlen(saludoPadre)); //escribo en el pipe
    printf ("\El hijo envia mensaje al hijo\n",buffer);
    wait(NULL); //espero al proceso hijo
    break;
}
return 0;
}
```

La compilación y ejecución muestra los siguientes resultados:

```
david@david-OEM ~/pss $ ./ejemplopipe3
hijo envia mensaje al hijo
hijo recibe algo del pipe..Buenos dias hijo.
```

### Entregable1.

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:

```
david@david-OEM ~/pss $ gcc entregable3_1.c -o entregable3_1
david@david-OEM ~/pss $ ./entregable3_1
hijo envia mensaje al padre
El padre recibe algo del pipe:Buenos dias Padre
```

### Comunicación en ambos sentidos entre proceso abuelo,padre e hijo

Vamos a hacer que padre e hijos puedan enviar y recibir información, como la comunicación es un único sentido necesitaremos dos pipes fd1 y fd2. Cada proceso usará un pipe para enviar la información y otro para recibirla.

Realizamos un esquema gráfico para tener claro lo que queremos implementar

ABUELO	fd1 ---->	HIJO	fd2 ---->	NIETO
	fd2 ←----		fd1 ←--	

- EI ABUELO
  - Envía información a través de fd1
  - Recibe a través de fd2
- EI HIJO
  - Envía información a través de fd2
  - Recibe la información a través de fd1
- EI NIETO
  - Usará fd1 para enviar información
  - Recibirá la información a través de fd2

Intenta desarrollar el programa por tu cuenta, si necesitas ayuda el código del programa es

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

```
void main(){
```

```
pid_t pid,Hijo_pid,pid2,Hijo2_pid;
```

```
int fd1[2];
```

```
int fd2[2];
```

```
char saludoAbuelo[]="Saludos del Abuelo..\n";
```

```
char saludoPadre[]="Saludos del Padre..\n";
```

```
char saludoHijo[]="Saludos del Hijo..\n";
```

```
char saludoNieto[]="Saludos del Nieto..\n";

char buffer[80]="";

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

if (pid==-1) //nos encontramos en proceso hijo
{
printf("No se ha podido crear el proceso hijo...\n");
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(nieto) en el HIJO");
exit(-1);
break;
case 0: //proceso hijo de HIJO (nieto)

//nieto recibe
close (fd2[1]); //cierro descriptor entrada
read (fd2[0],buffer,sizeof(buffer)); //leo el pipe
printf ("NIETO recibe mensaje de su padre %s\n",buffer);

//nieto envia
printf ("NIETO envia mensaje a su padre..");
close(fd1[0]);
write (fd1[1],saludoNieto, strlen(saludoNieto));
break;

default: //proceso padre (hijo)

//hijo recibe de abuelo
close (fd1[1]); //cierro descriptor entrada
read (fd1[0],buffer,sizeof(buffer)); //leo el pipe
printf ("HIJO recibe mensaje de ABUELO %s\n",buffer);

//hijo envia a su hijo(nieto)
```



```
printf ("HIJO envia mensaje a su hijo(nieto)..\\n");
close(fd2[0]);
write (fd2[1],saludoPadre, strlen(saludoPadre));
Hijo2_pid=wait(NULL); //Espero al hijo

//hijo recibe de hijo
close (fd1[1]); //cierro descriptor entrada
read (fd1[0],buffer,sizeof(buffer)); //leo el pipe
printf ("HIJO recibe mensaje de hijo %s\\n",buffer);

//hijo envia a su padre
printf ("HIJO envia mensaje a su padre...\\n",buffer);
close(fd2[0]);
write(fd2[1],saludoHijo,strlen(saludoHijo));

} //fin casos hijo (padre de nieto)

} //fin proceso hijo

else //nos encontramos en Proceso padre (abuelo)
{
//padre envia
printf ("Abuelo envia mensaje al HIJO..\\n");
close(fd1[0]);
write (fd1[1],saludoAbuelo, strlen(saludoPadre)); //escribo
Hijo_pid=wait(NULL); //Espero a la finalización del hijo

//padre recibe
close (fd1[1]); //cierro descriptor entrada
read (fd2[0],buffer,sizeof(buffer)); //leo el pipe
printf ("El abuelo recibe mensaje del hijo %s\\n",buffer);

} //fin proceso padre (abuelo)

exit(0);
}
```

**LA compilación y ejecución:**

```
david@david-OEM ~/pss $ gcc ejemplo_fork_pipe.c -o ejemplo_fork_pipe
david@david-OEM ~/pss $ ./ejemplo_fork_pipe
Abuelo envia mensaje al HIJO..
HIJO recibe mensaje de ABUELO Saludos del Abuelo..
HIJO envia mensaje a su hijo(nieto)..
NIETO recibe mensaje de su padre Saludos del Padre..

NIETO envia mensaje a su padre..HIJO recibe mensaje de hijo Saludos del Nieto..

HIJO envia mensaje a su padre...
El abuelo recibe mensaje del hijo Saludos del Hijo..
```

## PIPES con nombre o FIFOs

Los PIPES vistos hasta ahora establecen comunicación entre procesos (padre-hijo), pero ¿sólo podremos comunicarnos si somos familia? Los FIFOs permiten comunicar procesos que no tienen porque 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 en el se leen como en una cola, y una vez leídos no pueden ser leídos de nuevo.

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.

Formas usar FIFO:

- Ejecutando mknod desde el terminal
- Desde programa en C usando mknod()

Uso mknod desde linea de comandos

**mknod** [opciones] nombre **p**

El argumento que sigue a nombre especifica el tipo de fichero a construir:

p para un FIFO

- b para un fichero especial de bloques (con búfer)
- c para un fichero especial de caracteres (sin búferes)

En cuanto a las opciones: -m modo, --mode=modo Establece los permisos de los ficheros creados a modo, que es simbólico como en chmod.

Veremos mejor su funcionamiento mediante un ejemplo

### Ejemplo

Creo fichero FIFO1

```
>mknod FIFO1 P
```

A continuación, ejecuto orden de lectura del FIFO1

```
>cat FIFO1
```

No muestra nada porque no hay nada en la tubería y queda en espera, pero y si envío algo.. abro otro terminal y ejecuto

```
>ls > FIFO1
```

Veremos que el cat se ejecuta y muestra la información...

```
david@david-OEM ~/pss $ mknod FIFO1 p
david@david-OEM ~/pss $ cat FIFO1
actividad2
actividad2.c
doshijos
doshijos.c
ejemplo_fork_pipe
ejemplo_fork_pipe.c
ejemplopipe
ejemplopipe1.c
ejemplopipe3
ejemplopipe3.c
ejemwriteread
ejemwriteread.c
entregable3_1
entregable3_1.c
execv
execv.c
FIFO1
fork
fork2.c
```

Vamos a ver su funcionamiento en C

Para crear un FIFO en C, se puede hacer uso de la llamada del sistema `mknod()` con sus respectivos argumentos:

→ `mknod("/tmp/MIFIFO", S_IFIFO|0644 , 0);`

En este caso el archivo `"/tmp/MIFIFO"` se crea como archivo FIFO. El segundo argumento es el modo de creación, que sirve para indicarle a `mknod()` que crea un FIFO (con `S_IFIFO` ) y pone los permisos de acceso a ese archivo (644 octal, `rw-r--r--`) al igual como se puede hacer con el comando `chmod` . Finalmente, el tercer argumento de `mknod()` se ignora (al crear un FIFO) salvo que se este creando un archivo de dispositivo. En ese caso, se debería especificar los números mayor y menor del archivo de dispositivo

Las operaciones E/S sobre un FIFO son esencialmente las mismas que para las tuberías normales, con una gran excepción. Se debe usar una llamada del sistema `open()` o una función de librería para abrir físicamente un canal para la tubería.

## Ejemplos

Vamos a programar un programa `fifocrea.c` que crea un FIFO y lee la información del FIFO. (mientras no hay información permanece a la espera) y otro `fifoescribe.c` que escribe en el FIFO.

### **fifocrea.c**

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

#define TAM_BUF 80
#define TRUE 1

int main(void)
{
    int fp;
    char buffer[TAM_BUF];
    int nbytes;

    mknod("FIFO1",S_IFIFO|0660,0);

    while(TRUE)
    {
        fp=open("FIFO1",O_RDONLY);
        nbytes=read(fp,buffer,TAM_BUF-1);
        buffer[nbytes]='\0';
        printf("Cadena recibida: %s \n",buffer);
        close(fp);
    }
}
```

```
    return 0;
}
```

**fifoescribe.c**

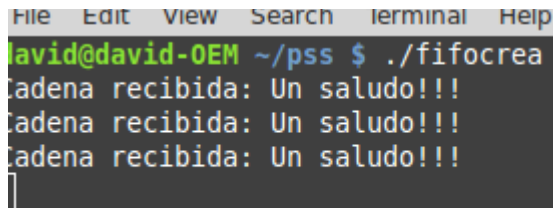
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main()
{
    int fp;
    char saludo[]="Un saludo!!!";
    fp=open("FIFO1",1);

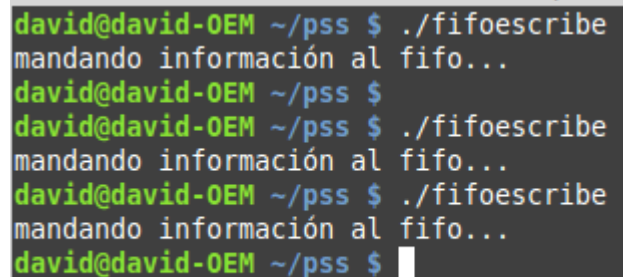
    if (fp==-1)
    {
        printf ("ERROR AL ABRIR ARCHIVO\n");
        exit(1);
    }

    printf ("mandando información al fifo...\n");
    write (fp,saludo,strlen(saludo));
    close(fp);
    return 0;
}
```

Vemos la ejecución como el fifocrea se crea y espera a recibir desde el otro proceso



```
File Edit View Search Terminal Help
david@david-OEM ~/pss $ ./fifocrea
cadena recibida: Un saludo!!!
cadena recibida: Un saludo!!!
cadena recibida: Un saludo!!!
█
```



```
david@david-OEM ~/pss $ ./fifoescribe
mandando información al fifo...
david@david-OEM ~/pss $
david@david-OEM ~/pss $ ./fifoescribe
mandando información al fifo...
david@david-OEM ~/pss $ ./fifoescribe
mandando información al fifo...
david@david-OEM ~/pss $ █
```

**Entregable**

Realizar el problema del productor/consumidor mediante FIFOS. Realiza dos programas consumidor.c y productor.c. El consumidor es el que se debe encargar de crear y borrar la tubería. El productor debe escribir en la tubería con nombre caracteres proporcionados por el usuario desde teclado. Cuando el usuario proporcione un carácter punto (o cualquier que determines) desde teclado se supone que debe terminar la comunicación. El consumidor por su parte, debe leer desde la tubería con nombre hasta que no quede nada en la tubería.