LLamadas al sistema para comunicación de procesos

1 FIFOS

Una FIFO es un fichero ‘especial’ que se utiliza para poder comunicar procesos.

Todos los procesos que acceden a la FIFO pueden leer o escribir información que es dejada o será recogida por otro proceso que tenga acceso a la FIFO.

Para tener acceso a una FIFO primero tiene que existir. Para crearla se usa la función ***mkfifo***.

La FIFO sólo debe ser creada por uno de los procesos que van a usarla, a partir de que uno de ellos la creen el resto sólo tendrá que abrirla para poder usarla

mkfifo()

Sintaxis: int mkfifo (const char *camino, modo_t modo);

Archivo de cabecera: #include <sys/types.h>

#include <sys/stat.h>

mkfifo construye un fichero especial FIFO con el nombre camino. modo especifica los permisos del FIFO.

Un fichero especial FIFO es similar a una interconexión o tubería, excepto en que se crea de una forma distinta. En vez de ser un canal de comunicaciones anónimo, un fichero especial FIFO se mete en el sistema de ficheros mediante una llamada a mkfifo.

Una vez que se ha creado un fichero especial FIFO de esta forma, cualquier proceso puede abrirlo para lectura o escritura, de la misma manera que con un fichero normal. Sin embargo, tiene que ser abierto en los dos extremos antes de que se pueda proceder a cualquier operación de entrada o salida.

El valor de retorno normal, si todo va bien, es 0. En caso de error, se devuelve -1

Los modos habituales de apertura serán:

S_IRUSR	Lectura para el propietario
S_IWUSR	Escritura para el propietario
S_IRUSR | S_IWUSR	Lectura y escritura para el propietario
S_IRWXU	Lectura, escritura y ejecución para el propietario
S_IRWXG	Lectura, escritura y ejecución para el grupo
S_IRWXO	Lectura, escritura y ejecución para el resto de usuarios

unlink()

Sintaxis int unlink(const char *pathname);

Archivos de cabecera #include <unistd.h>

Una vez que se ha dejado de trabajar con la FIFO deberemos eliminarla del sistema. Para ello usamos la llamada a unlink(), que borra un nombre del sistema de ficheros. Si dicho nombre era el último enlace a un fichero, y ningún proceso tiene el fichero abierto, el fichero es borrado y el espacio que ocupaba vuelve a estar disponible.

Si el nombre era el último enlace a un fichero, pero algún proceso sigue teniendo el fichero abierto, el fichero seguirá existiendo hasta que el último descriptor de fichero referente a él sea cerrado.

Si el nombre hace referencia a una FIFO, el nombre es eliminado, pero los procesos que tengan el objeto abierto pueden continuar usándolo.

En caso de éxito, se devuelve cero. En caso de error, se devuelve –1.

1.1 APERTURA, LECTURA Y ESCRITURA DE FIFOS

Una vez creada la FIFO, cualquier proceso puede abrir la FIFO para leer o escribir como lo haría con cualquier fichero (teniendo en cuenta las características especiales de la FIFO).

Abrir un FIFO para lectura normalmente produce un bloqueo hasta que algún otro proceso abre el mismo FIFO para escritura, y viceversa.

Las operaciones de lectura y/o escritura tendrán la misma sintaxis que las mismas operaciones sobre cualquier otro fichero.

Para abrir una FIFO (previamente creada) para lectura haremos:

```
int fd;
fd = open("nombre",O_RDONLY);
```

Para abrir una FIFO (previamente creada) para escritura haremos:

```
int fd;
fd = open("nombre",O_WRONLY);
```

Para escribir en una FIFO usaremos la orden *write* con la siguiente sintaxis:

```
int write(fd, &variable, nº bytes):
```

Para leer de una FIFO usaremos la orden read con la siguiente sintaxis

```
int read(fd, &variable, nº bytes):
```

Si se intenta leer de una FIFO, que no tiene información, y ningún proceso la tiene abierta para escritura, la orden read devuelve 0 y continúa la ejecución del proceso

Si se intenta escribir en una FIFO que no está abierta por ningún otro proceso para lectura, el proceso que intenta realizar la escritura recibe la señal SIGPIPE. (Si el proceso no está preparado para recibirla muere).

EJEMPLO: El proceso padre crea una fifo que es usada para que el hijo 2 le mande un dato de tipo float al hijo 1.

Proceso: **Padre**

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

void main()
{

int vpidb,vpidc,retorno;

mkfifo("fifo1",0777);

vpidb=fork();
if(vpidb==0) {
  execl("hijo1","hijo1",NULL);
  perror("no se puede ejecutar el hijo1");
  exit(-1);
}
else if(vpidb==-1) printf("Imposible hacer el fork\n");

vpidc=fork();
if(vpidc==0) {
  execl("hijo2","hijo2",NULL);
  perror("no se puede ejecutar el hijo2");
  exit(-1);
}
else if(vpidc==-1) printf("Imposible hacer el fork\n");

wait(&retorno);
wait(&retorno);

printf("Fin de Padre\n");

unlink("fifo1");
}
```

Proceso: **hijo1**

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


void main()
{

 int f1;
 float dato;

 f1=open("fifo1",O_RDONLY);

 printf("Hola soy el hijo 1\n");
 read(f1,&dato,sizeof(dato));
 printf("Fin de Hijo 1, y leo %f\n",dato);

 // AQUÍ PUEDE HABER MAS READS, ANTES DEL CLOSE

 close(f1);
}
```

Proceso: **hijo2**

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


void main()
{

 int f1;
 float dato=3.1415;

 f1=open("fifo1",O_WRONLY);

 printf("Hola soy Hijo 2\n");
 sleep(2);
 write(f1,&dato,sizeof(dato));
 printf("Fin de Hijo 2\n");

 // AQUÍ PUEDE HABER MAS WRITES, ANTES DEL CLOSE

 close(f1);
}
```

2 TUBERÍAS

Una *tubería* es un fichero ‘especial’ que se utiliza para poder comunicar procesos, que están emparentados de alguna forma, sin necesidad de ponerle nombre.

pipe()

Sintaxis: pipe (int nombre de tubería[2])

Archivo de cabecera: #include <unistd.h>

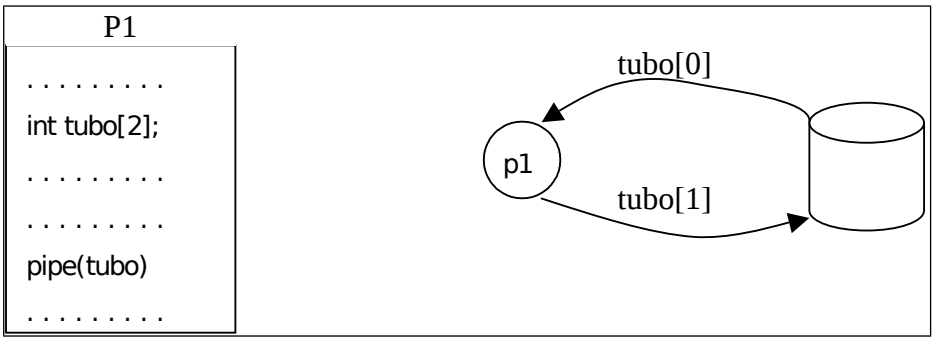
Es la llamada encargada de crear una tubería. La tubería es la forma con la que dos procesos intercambian información. La tubería es un almacenamiento temporal donde los procesos pueden escribir o leer información.

Como parámetro tomará un array de dos posiciones de tipo entero.

La posición 0 del array será la posición de lectura de la tubería.

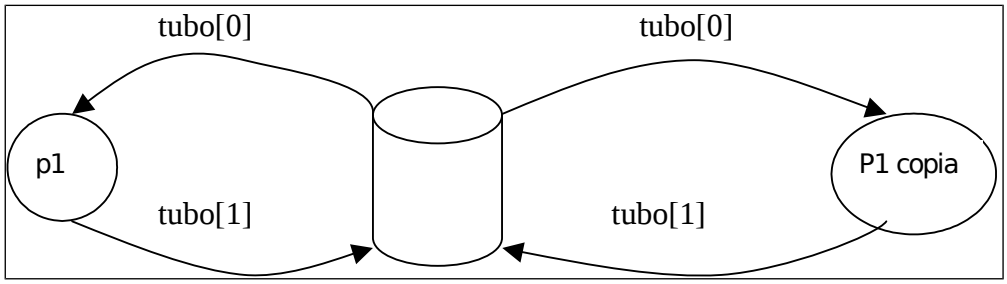
La posición 1 del array será la posición de escritura de la tubería.

Al ejecutar la llamada a pipe se buscan las dos primeras posiciones libres de la tabla de canales y se asignan a la posición 0 del array la primera posición libre encontrada de la tabla de canales y a la posición 1 del array la segunda posición libre de la tabla de canales. Dichas posiciones no tienen por que ser consecutivas.



¿Cómo nos comunicamos con otro proceso?

Habrá que crear el nuevo proceso con el que queremos comunicarnos mediante un fork(), con lo cual hereda la tabla de canales.



Para escribir en una tubería usaremos la orden *write* con la siguiente sintaxis:

int write(entrada T.C, &variable, nº bytes)

al finalizar devolverá el número de bytes que ha escrito. La escritura en la tubería es atómica, es decir, o se escribe todo o no se escribe nada. Si en el momento de escribir la tubería estuviese llena se esperará a que haya espacio para escribir. El parámetro *entrada T.C* será el contenido que tengamos en la posición 0 del array de esa tubería.

Para leer de una tubería usaremos la orden read con la siguiente sintaxis

int read(entrada T.C, &variable, nº bytes)

al finalizar se devolverá el nº de bytes que se han leido. Si en el momento de la lectura la *pipe* estuviese vacía se esperará a que haya información. . El parámetro *entrada T.C* será el contenido que tengamos en la posición 1 del array de esa tubería.

¿ Que pasaría con la siguiente situación?

P1 cierra tubo[1]

P1copia cierra tubo[0] y tubo[1]

P1 lee de tubo[0]

Se detecta que no hay puntero de escritura y el proceso P1 leerá un fin de fichero. El read devolverá un 0.

¿ Y con esta otra ?

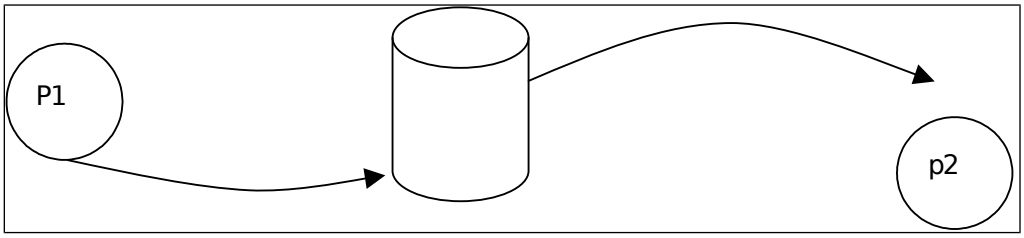
P1copia cierra tubo[0] y tubo[1]

P1 cierra tubo[0]

P1 escribe en tubo[1]

Se detecta que no hay puntero de lectura y el sistema manda la señal SIGPIPE al proceso P1 que intenta escribir. (Si el proceso no está preparado para recibirla muero).

¿ Cómo hacemos lo siguiente ?



Para conseguirlo necesitamos hacer un *execl*() tras el *fork*(), pero entonces nos encontramos con que sólo se heredarán las 3 primeras posiciones de la *tabla de canales*. Para conseguir la comunicación necesitamos que el puntero de lectura esté situado en las tres primeras posiciones de la *tabla de canales*, para que se herede al hacer el *execl*().

Para hacer esto necesitaremos llevar el puntero de lectura del proceso que va a hacer el *execl*() a una de las tres primeras posiciones de la *tabla de canales*. Para poder hacer esto necesitamos la orden que veremos a continuación.

dup()

Sintaxis: int dup (int posición a duplicar)

Archivo de cabecera: #include <unistd.h>

Esta orden duplica la posición de la tabla de canales que indiquemos como parámetro en la primera que encuentre libre. El valor que devuelve será el de la posición donde ha duplicado la entrada que toma como parámetro.

EJEMPLO: El proceso Padre crea un proceso hijo y una tubería entre ellos, que es usada para que el hijo envíe información al padre.

Padre

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

void main()
{
 int tubo[2],pidb,numero,retorno;

 printf("hola soy El Padre\n");
 pipe(tubo);


pidb=fork();
if(pidb==0) {
  close(2);
  dup(tubo[1]);
  execl("hijo","hijo",NULL);
  perror("no se puede ejecutar el hijo");
  exit(-1);
}
else if(pidb==-1) printf("Imposible hacer el fork\n");;

close(tubo[1]);
printf("Ya soy padre \n");
read(tubo[0],&numero,sizeof(numero));
printf("Mi hijo me manda %d\n",numero);
wait(&retorno);
}
```

Hijo

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

void main()
{

 int valor=48;

 printf("Hola soy el Hijo\n");
 sleep(3);
 write(2,&valor,sizeof(valor));
 printf("Adios\n");
 exit(0);
}
```