Pipes (Tuberías)

Los **Pipes** son herramientas que permiten conectar la salida estándar de un proceso con la entrada estándar de otro. La idea es similar a tener un programa que, en lugar de mostrar por pantalla el texto escrito por teclado, lo guarda en un fichero directamente. Normalmente los **Pipes** son de un solo sentido: sirven para leer o para escribir, pero no ambas cosas al mismo tiempo (half-duplex). Si deseas que dos **Procesos** sean capaces de comunicarse en ambos sentidos, se requerirán dos Pipes (full-dúplex).

Los **Pipes** internamente son, por tanto, flujos unidireccionales de bytes que conectan la salida estándar de un **Proceso** con la entrada estándar de otro **Proceso**.

Cuando dos procesos están enlazados mediante un **Pipe**, ninguno de ellos es inicialmente consciente de la redirección de las E/S. Así, cuando el **Proceso Escritor** desea escribir en la tubería, utiliza las funciones normales para escribir por pantalla. Lo único especial que sucede es que ya no se escriben cosas por la pantalla, sino que el texto va a un 'fichero especial'. El **Proceso Lector** se comporta de forma similar, usando las funciones de lectura del teclado.

Los procesos están autorizados a realizar **lecturas no bloqueantes** de la tubería, es decir, así que, si no hay datos para ser leídos o si la tubería está bloqueada, se devolverá un error.

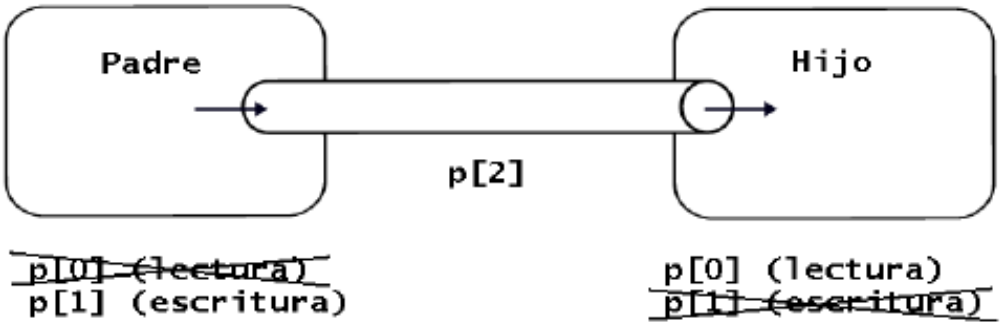
Programar **Pipes** en C es bastante sencillo. Para abrir un fichero, leerlo, escribirlo y cerrarlo se emplea su **Descriptor de Fichero**. Por su parte, los **Pipes** tienen **dos descriptores** de fichero: uno para el extremo de Escritura y otro para el extremo de Lectura. Dado que en Linux los descriptores de fichero son simplemente enteros, se usa un array para definir un Pipe. Es decir: **int pipe [2];**

Para crear un **Pipe** se emplea la función **pipe ()**, que abre dos descriptores de fichero y almacena su valor en el array. El primer descriptor de fichero es abierto como sólo lectura. El segundo se abre como sólo escritura. De esta manera se asegura que el pipe sea de un solo sentido.

Una vez creado un **Pipe**, se podrán hacer lecturas y escrituras de manera normal, como si se tratase de cualquier fichero. Se suelen utilizar para intercambiar datos entre **Procesos**. Como ya sabemos, un Proceso Hijo hereda todo de su padre (salvo el PID), por lo que la comunicación entre el **Pipe** y el Proceso Hijo es bastante cómoda mediante **Pipes**.

Para asegurar la unidireccionalidad de la tubería, es necesario que tanto el Padre como el Hijo cierren (**close ()**) los descriptores de ficheros que no van a usar del **Pipe**. Por ejemplo, si un Proceso Padre puede enviarle datos a su hijo a través de un **Pipe**:

- El Proceso Padre cierra el extremo de Lectura del Pipe.
- El Proceso Hijo cierra el extremo de Escritura del Pipe.



Resumen de comandos

Los comandos para trabajar con **Pipes** son:

Pipes	
int pipe (int fd[2]);	Crea un Pipe. [0] descriptor del fichero de lectura, [1] de escritura. Devuelve un 0 si todo va bien, -1 si hay error.
int close (int fd);	Cierra el Pipe
int read (int fd, void *buf, int cont);	Intenta leer del Pipe. Retorna el número de bytes leídos
int write (int fd, void *buf, int cont);	Intenta escribir. Retorna el número de bytes escritos

Ejemplo 1 – Un Pipe sencillo

En este ejemplo, vamos a crear un **Pipe** simple. Se crean dos Procesos, y el **Padre** quiere enviarle un texto al **Hijo** mediante un **Pipe**.

- El Padre cierra el extremo de Lectura del Pipe, le envía un texto, cierra el extremo de Escritura del Pipe y espera a que el Hijo finalice.
- El Hijo cierra el extremo de Escritura del Pipe, recoge el texto, lo muestra por la pantalla, cierra el extremo de Lectura del Pipe y finaliza.

En la terminal al ejecutar veremos el texto enviado. El código es algo confuso, pero recuerda que al generarse el Hijo éste tiene su propia variable miPipe [2], por tanto, el Padre no está interfiriendo con el Hijo ni el Hijo con el Padre.

pipes1.cpp

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <csignal>
#include <cstring>

#define size 512
int main () {
    pid_t pid;
    int miPipe[2], readbytes, status;
    char buffer [size];

    // Creamos el pipe
    pipe (miPipe);

    if ((pid = fork()) == 0) {
        // HIJO - Lee del Pipe
        close (miPipe[1]); // Cerramos Escritura
        while ((readbytes = read (miPipe [0], buffer, size)) > 0)
            write (1, buffer, readbytes);
        close (miPipe[0]);
    } else {
        // Padre - Escribe en el Pipe
        close (miPipe[0]); // Cerramos Lectura
        strcpy (buffer, "Texto que mandamos al Pipe\n");
        write (miPipe[1], buffer, strlen (buffer));
        close (miPipe[1]);
        wait (&status); // Esperamos a que el Hijo acabe
    }
    exit (0);
}
```

Ejemplo 2 – Comunicación bidireccional

Una comunicación bidireccional entre Padre e Hijo requerirá de **dos Pipes**. En cada proceso habrá que cerrar descriptores correctos. Por ejemplo, vamos a usar el **Pipe** padreHijo [2] para la comunicación desde el Padre al Hijo; y el **Pipe** hijoPadre [2] para la inversa. Por tanto, habría que cerrar:

- En el Padre: el lado de lectura de padreHijo [2] y el lado de escritura de hijoPadre [2].
- En el Hijo: el lado de escritura de padreHijo [2] y el lado de lectura de hijoPadre [2].

pipes2.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <csignal>
#include <cstring>

#define size 512
int main () {
    pid_t pid;
    int pipeA[2], pipeB[2], readbytes, status;
    char buffer [size];

    // Creamos los pipes
    pipe (pipeA);
    pipe (pipeB);

    if ((pid = fork()) == 0) {
        // HIJO - Leemos del pipeA
        close (pipeA[1]);
        close (pipeB[0]);
        while ((readbytes = read (pipeA [0], buffer, size)) > 0)
            write (1, buffer, readbytes); // Mostramos por pantalla
        close (pipeA[0]);

        // HIJO - Reenviamos por el pipeB
        strcpy (buffer, "HIJO - Mensaje que llega de pipeA \n");
        write (pipeB[1], buffer, strlen (buffer));
        close (pipeB[1]);
    } else {
        // Padre - Escribe en el pipeA
        close (pipeA[0]);
        close (pipeB[1]);
        strcpy (buffer, "Soy el padre \n");
        write (pipeA[1], buffer, strlen (buffer));
        close (pipeA[1]);

        // Padre - Leemos del pipeB la respuesta del Hijo
        while ((readbytes = read (pipeB [0], buffer, size)) > 0)
            write (1, buffer, readbytes); // Mostramos por pantalla
        close (pipeB[0]);
        wait (&status);
    }
    exit (0);
}
```

Ejercicio 3

Crea un programa en C que:

- Cree un Proceso Hijo y un Proceso Nieto (Hijo del Hijo)
- Cada Proceso debe de escribir un texto que diga “Soy el …”

Ejercicio 4

Crea un programa en C que:

- Cree un Proceso Padre y un Proceso Hijo
- Defina una variable entera con valor de al principio del programa.
- El Padre incrementará el valor en 5. El Hijo lo decrementará en 5.
- Muestra los valores por pantalla.
- ¿Por qué no se confunde el programa con las sumas y restas?

Ejercicio 5

Copia el siguiente programa y ejecútalo. ¿Qué se muestra por pantalla?

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(){
      int estado, hijo;
      hijo = fork();
      if (hijo != 0) {
            wait(&estado);
            printf("PADRE: pid=%d ppid=%d user-id=%d \n", getpid(),
            getppid(), getuid());
            printf("Codigo de retorno del hijo: %d\n", estado);
      }else{
            printf("HIJO: pid=%d \n", getpid());
      }
      exit(0);

}
```

Responde:

- ¿Que se ejecuta antes? ¿El padre o el hijo? ¿Por qué?
- En la instrucción wait(&estado), ¿qué es lo que se recibe en la variable estado?

Ejercicio 6

Realiza un programa en C que cree una carpeta llamada “trabajo”

Ejercicio 7

Copia el siguiente programa, ejecútalo y apunta el PID.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void trapper(int);
int main(){
      int i;
      for(i=1;i<=64;i++)
            signal(i, trapper);
            printf("Identificador del proceso: %d\n", getpid() );
            pause();
            printf("Continuando...\n");
            printf("Termino.\n");
            return 0;
      }
void trapper(int sig){
      printf("\nSeñal que he recogido: %d\n", sig);
}
```

Responde:

- Este Proceso no termina nunca, porque está en pausa. Desde otra terminal, envíale la señal SIGUSR1. ¿Cuál es el comando que has utilizado?
- ¿Qué es lo que pasa si un proceso recibe una Señal que no trata?
- Cambia el programa para que sólo recoja las señales del 1 al 9.
- Ejecútalo y mándale la señal SIGUSR1. ¿Cuál es el comando que has utilizado?
- ¿Qué ha pasado al enviarle la señal? ¿Por qué?

Ejercicio 8

Crea un programa en C que:

- Cree un proceso Hijo y un proceso Nieto (hijo del hijo)
- Tenga dos PIPE
- El padre envíe un mensaje al Hijo; y el Hijo se lo pasará al Nieto.