Introducción a la programación de PIPES

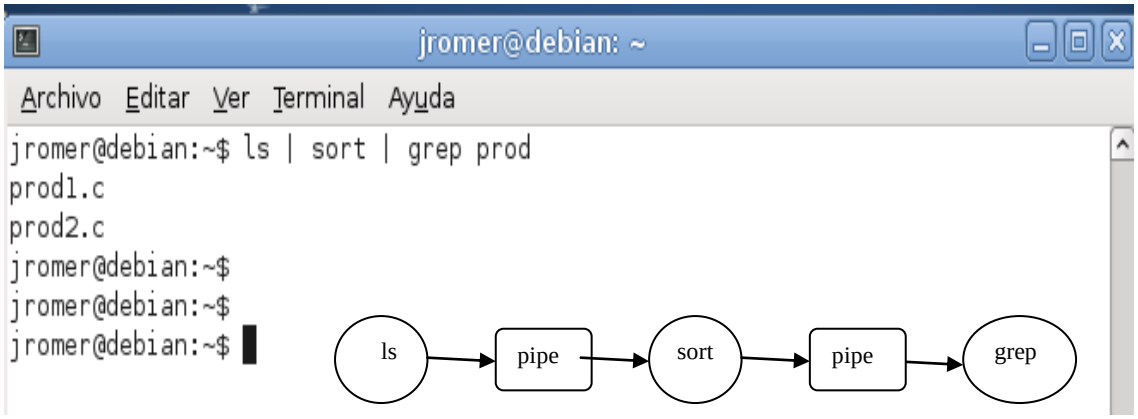
Una de las formas mas simples de comunicación entre procesos emparentados, son las tuberías sin nombre también llamadas pipes sin nombre cuya llamada al sistema para crear el recurso es **int pipe(int [ ])**, utilizadas entre procesos emparentados que son procesos creados con la llamada al sistema **pid_t fork(void)**, la cual crea una relación entre un proceso padre y un proceso hijo, esta llamada al sistema, hace que el proceso hijo herede los descriptores del recurso de comunicación pipe creado por el proceso padre. El pipe es un área de memoria donde los procesos pueden escribir y leer en un orden fifo permitiendo de esta manera realizar una comunicación.

En todos los códigos ejemplo suponemos que las funciones que se ejecutan nunca fallan, esto lo hacemos para darle menor complejidad al código y concentrarnos en el manejo de pipes.

pipe1.c

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

int main(void)
{
  pid_t hijo;
  int p[2];
  if (pipe(p) == -1) exit(-1);
  hijo = fork();
  if (hijo == -1) exit(-2);
  if ( hijo == 0 ) // soy el hijo
   {
      printf("hijo informa extremo lector %d extremo escritor %d\n",p[0],p[1]);
   }
  else
   {
      printf("padre informa extremo lector %d extremo escritor %d\n",p[0],p[1]);
   }
   exit(0);
}
```

Los pipes son mas conocidos por el uso que se hace de ellos desde la línea de comandos del shell, donde los procesos se conectan a través de una tubería y el flujo de datos se realiza en una dirección, como lo muestra la siguiente línea de comandos, donde el pipe es la | (barra vertical).



Para conectar dos o más procesos a través de un pipe, tenemos que crear un pipe antes de que un proceso cree un proceso hijo, ya que el hijo al heredar la tabla de descriptores de archivos abiertos por el padre también heredara la copia de los descriptores asociados al pipe.

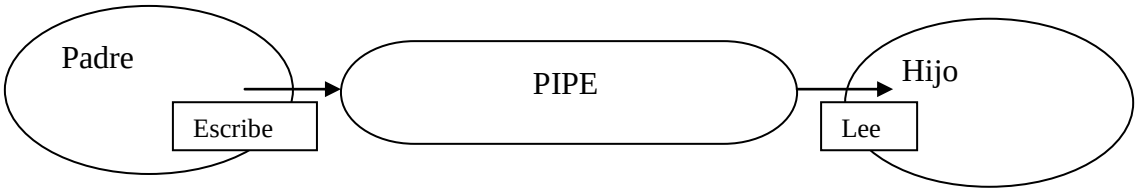
Establecer una comunicación entre procesos usando pipes es bastante sencillo, ya que los pipes proporcionan una capa de sincronización de bloqueos en la escritura y en la lectura haciendo que el programador no tenga que incluir código para la sincronización en la comunicación entre dos procesos. Un proceso escritor en el pipe puede llenar el pipe y otro proceso lector del pipe puede vaciar el pipe ya que la lectura del pipe es destructiva, en estos casos cuando el pipe esta lleno el escritor se bloquea en la escritura mágicamente y cuando el pipe esta vacío el lector se bloquea en la lectura también mágicamente, por supuesto que para que esto ocurra se tienen que dar ciertas condiciones que veremos mas adelante, si ha esto agregamos que la lectura del pipe es destructiva permitiendo usar el pipe de forma que nunca agote su capacidad de almacenamiento y que la lectura se realiza en el mismo orden en el cual se escribió en el pipe, todo esto hace que la comunicación sea muy eficiente, con muy poca programación.

Por supuesto que debemos tener un gran dominio del pipe en la programación para que no ocurran interbloqueos.

Primero hay que tener en cuenta que, si bien un pipe permite que muchos procesos escriban en él, y muchos procesos lean de él en forma concurrente, lo más adecuado sería tener un solo escritor y un solo lector sobre un pipe en un instante dado, así evitaremos interbloqueos no deseados.

Los pipes solo permiten comunicar procesos emparentados, podemos decir que los procesos emparentados son aquellos que se crean utilizando un fork y que tienen una relación padre hijo.

Muestra un proceso lector y un proceso escritor usando un pipe



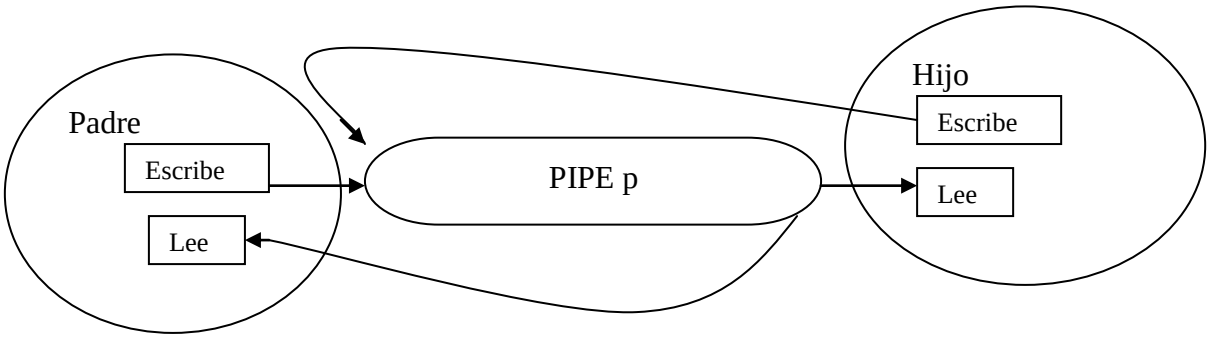
El siguiente código ejemplifica esta situación.

*pipe2.c pipe1.c

```
#include <sys/types.h>
#include <stdlib.h>

int main(void)
{
  pid_t hijo;
  int p[2];
  if (pipe(p) == -1) exit(-1);
  hijo = fork();
  if (hijo == -1) exit(-2);
  if ( hijo == 0 ) // soy el hijo
   {
      close(p[1]);
      printf("hijo informa extremo lector %d extremo escritor %d\n",p[0],p[1]);
      // comienza la comunicacion el hijo lee del pipe
   }
  else // soy el padre
   {
      close(p[0]);
      printf("padre informa extremo lector %d extremo escritor %d\n",p[0],p[1]);
      // comienza la comunicacion el padre escribe en el pipe
   }
   exit(0);
}
```

Es importante destacar la instrucción close(p[1]) en el proceso hijo y close(p[0]) en el proceso padre, ya que de esta manera se especifica que solo hay un proceso lector y un proceso escritor.

Dos procesos lectores y dos procesos escritores, aunque solo se ve un proceso padre y un proceso hijo hay dos escritores y dos lectores esto quiere decir que cada proceso cumple la función de escritor y lector.



El siguiente código ejemplifica esta situación.

pipe3.c | pipe1.c | pipe2.c

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

int main(void)
{
  pid_t hijo;
  int p[2];
  if (pipe(p) == -1) exit(-1);
  hijo = fork();
  if (hijo == -1) exit(-2);
  if ( hijo == 0 ) // soy el hijo
   {
      printf("hijo informa extremo lector %d extremo escritor %d\n",p[0],p[1]);
      // comienza la comunicacion el hijo puede leer del pipe y escribir en el pipe
   }
  else // soy el padre
   {
      printf("padre informa extremo lector %d extremo escritor %d\n",p[0],p[1]);
      // comienza la comunicacion el padre puede leer del pipe y escribir en el pipe
   }
   exit(0);
}
```

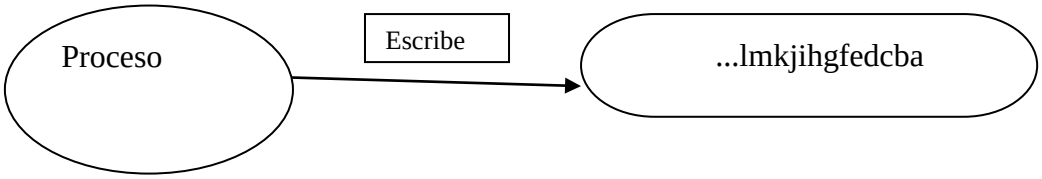
Es importante destacar la ausencia de la instrucción close(p[1]) en el proceso hijo y close(p[0]) en el proceso padre, ya que de esta manera se especifica que cada proceso cumple la función de lector y escritor.

La comunicación en los pipes es unidireccional o semiduplex, esto quiere decir que se escribe solo por un extremo del pipe y se lee solo por el otro extremo del pipe.

¿Qué sucede con el escritor cuando intenta escribir en un pipe lleno?

Un proceso escritor intenta escribir en el pipe el abecedario muchas veces y se bloquea cuando el pipe esta lleno.

En este caso, el único proceso es escritor y lector. Por defecto, cuando se crea un pipe, el o los procesos que obtienen los descriptores de lectura y escritura, son lectores y escritores, si no se especifica lo contrario. Se puede utilizar la función close para cerrar el descriptor que no va usar, convirtiendo así un proceso en solo escritor o en solo lector.



El siguiente código ejemplifica esta situación.

pipe4.c

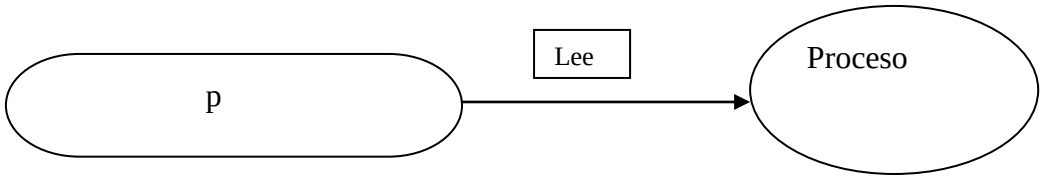
```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>

int main(void)
{
  char letra = 'a';
  int cuentaabecedarios = 0;
  int p[2];
  if (pipe(p) == -1) exit(-1);
  while(1)
     {
       if (letra == 'z')
          {
            cuentaabecedarios++;
            printf("Cantidad de abecedarios escritos en el pipe = %d
\n",cuentaabecedarios);
          }
      write(p[1],&letra,1);
      letra++;
     }
   exit(0);
}
```

Este proceso es lector y escritor porque no hizo close(p[0]) indicando que no realizaría lectura del pipe, entonces aquí el proceso escribe hasta que el pipe se llena, cuando el pipe se llena ya que el pipe tiene una capacidad limitada de bytes para almacenar, el proceso se bloquea en el write, esperando que se lea del pipe para que se genere lugar libre para la escritura, ya que la lectura es destructiva.

El proceso escritor se bloquea en la escritura cuando el pipe esta lleno y si hay un proceso lector.

¿Qué sucede con el lector cuando intenta leer de un pipe vacío?

Un proceso lector lee de un pipe y se bloquea cuando el pipe se vacía.



En este caso, el único proceso es escritor y lector. Por defecto, cuando se crea un pipe, el o los procesos que obtienen los descriptores de lectura y escritura, son lectores y escritores, si no se especifica lo contrario. Se puede utilizar la función close para cerrar el descriptor que no va usar, convirtiendo así un proceso en solo escritor o en solo lector.

El siguiente código ejemplifica esta situación.

pipe4.c pipe5.c

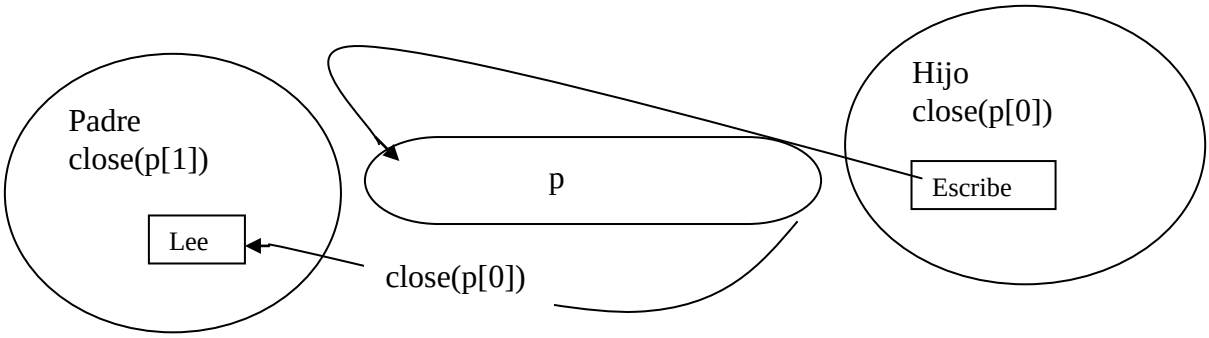
```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <stdlib.h>
4
5 int main(void)
6 {
7   char letra ;
8   int p[2];
9   if (pipe(p) == -1) exit(-1);
10  while(1)
11     {
12      read(p[0],&letra,1);
13      printf("%c\n",letra);
14     }
15   exit(0);
16 }
17
```

Este proceso es lector y escritor porque no hizo close(p[1]) indicando que no realizaría escritura en el pipe, entonces aquí el proceso lee hasta que el pipe se vacía, cuando el pipe se vacía (de hecho el pipe está vacío ya que ningún proceso escribió en él) el proceso se bloquea en el read, esperando que se escriba en el pipe para poder desbloquearse de la lectura.

El proceso lector se bloquea en la lectura, cuando el pipe esta vació y si hay un proceso escritor.

¿Qué sucede con el proceso escritor cuando terminan todos los procesos lectores?

Un proceso Hijo escribe el abecedario en el pipe y el proceso Padre lee el abecedario del pipe, pero cuando el lector lee la letra 'R' cierra el canal de lectura close(p[0]), entonces el proceso Hijo es terminado por el Sistema Operativo.



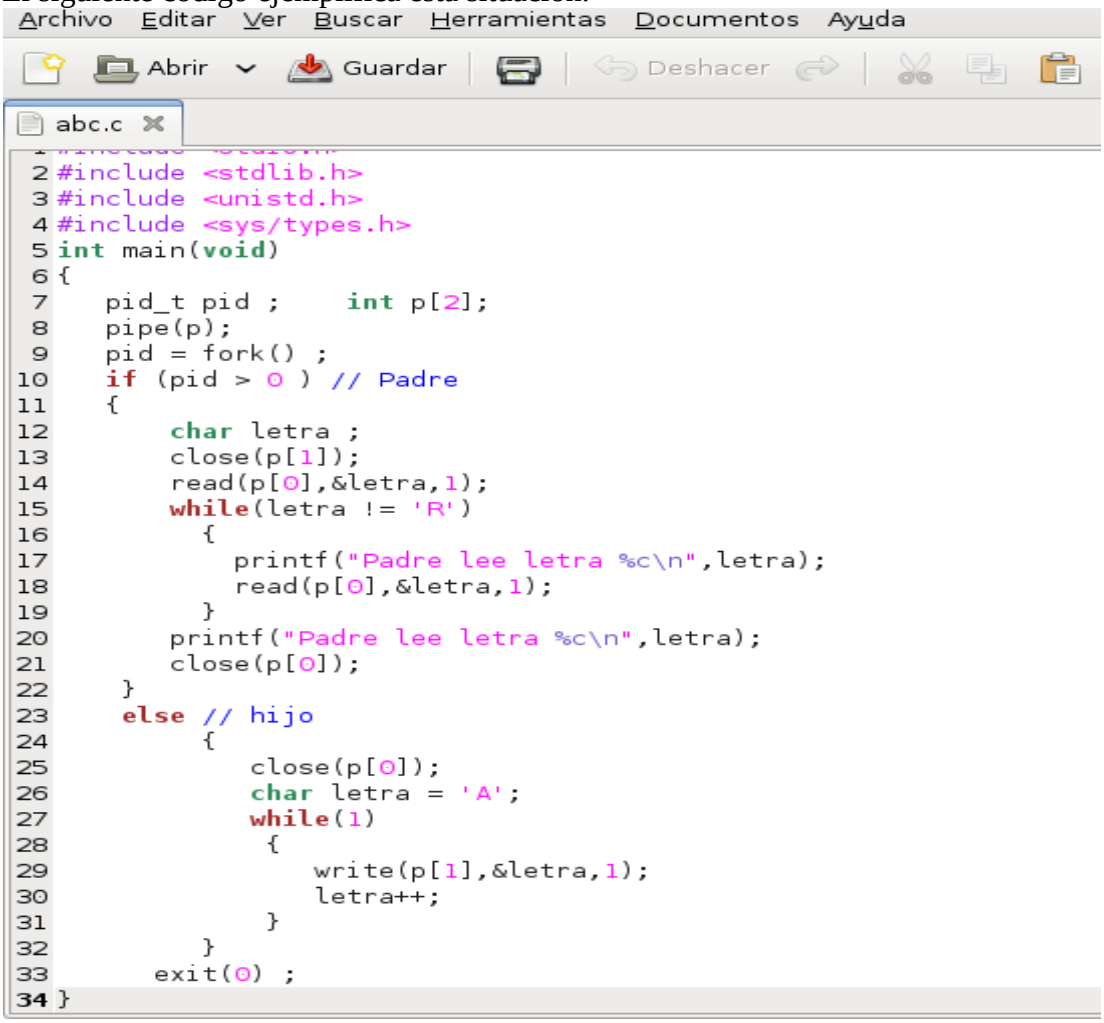
El proceso escritor es terminado en la escritura cuando el proceso lector cierra su file descriptor de lectura ejecutando close(p[0]) o simplemente termina.

Por defecto, cuando se crea un pipe, cada proceso que tenga acceso al identificador de lectura y escritura del pipe, podrá leer y escribir del mismo, entonces decimos que ese proceso es lector y escritor del pipe si no se especifica lo contrario, que quiere decir especificar lo contrario, quiere decir que debemos hacer explícito en el código del proceso, que función va a cumplir el mismo en la comunicación usando pipes, el proceso será solo lector, será solo escritor o será lector y escritor.

```
close(p[1]);  // El proceso especifica que no va a escribir en el pipe
close(p[0]);  // El proceso especifica que no va a leer del pipe
```

Mientras no se explicite alguna de estas dos operaciones en el código, el proceso será escritor y lector del pipe.

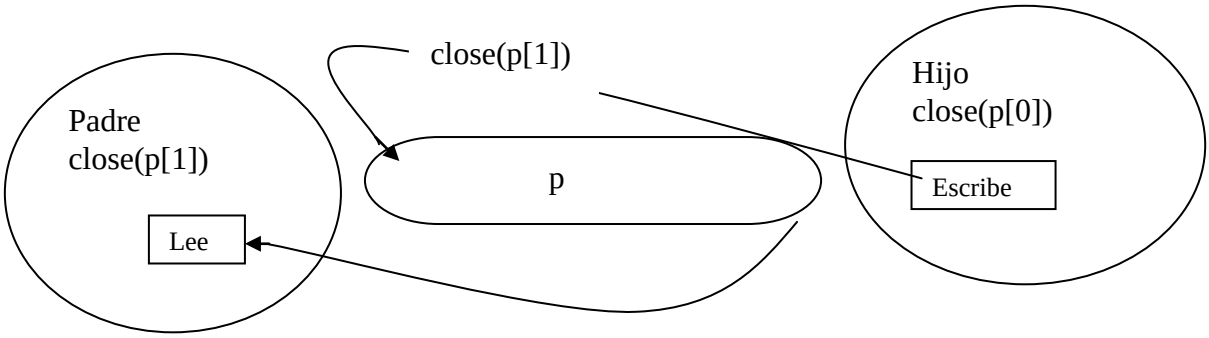
El siguiente código ejemplifica esta situación.




```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{
    pid_t pid ;     int p[2];
    pipe(p);
    pid = fork() ;
    if (pid > 0 ) // Padre
    {
        char letra ;
        close(p[1]);
        read(p[0],&letra,1);
        while(letra != 'R')
          {
            printf("Padre lee letra %c\n",letra);
            read(p[0],&letra,1);
          }
        printf("Padre lee letra %c\n",letra);
        close(p[0]);
     }
     else // hijo
          {
             close(p[0]);
             char letra = 'A';
             while(1)
              {
                 write(p[1],&letra,1);
                 letra++;
              }
          }
       exit(0) ;
}
```

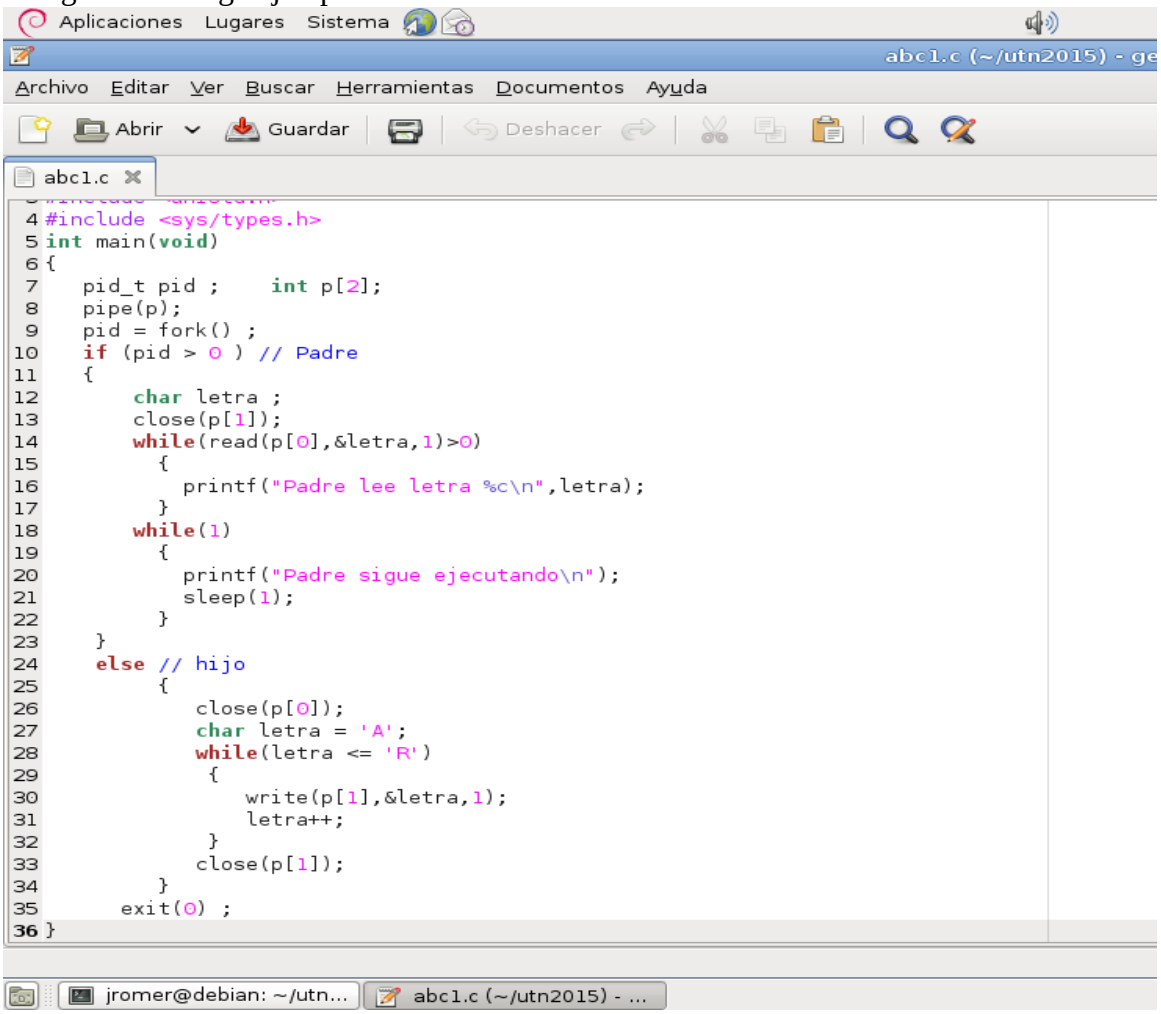
¿Qué sucede con el proceso lector cuando terminan todos los procesos escritores?

Un proceso Hijo escribe el abecedario en el pipe y el proceso Padre lee el abecedario del pipe, pero cuando el escritor escribe la letra ‘R’ cierra el canal de escritura, entonces el proceso Hijo lee del pipe hasta que este se vacía y no queda bloqueado, continúa su ejecución.



El proceso lector no se bloquea en la lectura cuando el pipe se vacía y no hay escritores porque el o los escritores cerraron su file descriptor de escritura ejecutando close(p[1]) o simplemente terminan.

El siguiente código ejemplifica esta situación.



Características más importantes de los pipes

1) Los pipes solo permiten comunicar procesos emparentados, podemos decir que los procesos emparentados son aquellos que se crean utilizando un fork y que tienen una relación padre hijo.

2) La comunicación es unidireccional, esto quiere decir que se escribe solo por un extremo del pipe y se lee por el otro extremo del pipe. Representado en el código por el write(p[1],…,…) y por read(p[0],…,…)

3) El proceso escritor se bloquea en la escritura cuando el pipe esta lleno y si hay un proceso lector por lo menos.

4) El proceso lector se bloquea en la lectura, cuando el pipe esta vació y si hay un proceso escritor por lo menos.

5) La lectura de un pipe es destructiva.

6) El orden de lectura es FIFO.

7) El proceso escritor es terminado en la escritura por el sistema operativo cuando el proceso lector cierra su canal de lectura con close(p[0]) o termina, lo que es lo mismo decir, cuando ya no hay procesos lectores el proceso escritor es terminado en el intento de escritura sobre el pipe.

8) Un proceso lector si esta bloqueado en el read en un pipe vacío esperando que un proceso escritor escriba, pero el escritor cierra su canal de escritura con close(p[1]) o por algún motivo termina, el proceso lector se desbloquea y sigue con la ejecución lógica de su código, si así fue programado.