Universidad de Cantabria
Grado de Ingeniería Informática

Sistemas Operativos

Práctica 2 - Parte 3

Rafael Menéndez de Llano Rozas

Guion

2

Comunicación entre procesos

En esta parte adquirirás la visión del sistema como programador haciendo uso de las llamadas al sistema POSIX para la comunicación entre procesos usando el mecanismo de las pipe sin nombre o con nombre (fifo).

pipe: Comunicación de procesos

Como se ha dicho, las señales son apropiadas para situaciones de error o para sincronizar y transmitir eventos entre procesos, pero no para transmitir información. Para ello se han creado las pipeS (conducto, tubería o canalización), que son una versión de los ficheros y se utilizan con llamadas al sistema `read` y `write` (las mismas que utilizan las llamadas de ANSI C `scanf` y `printf`). Se pueden entender como canales unidireccionales que conectan varios procesos.

Como en el caso de las señales, desde la *shell* hemos utilizado esta característica para concatenar la información de salida de un comando con la entrada de otro a través del símbolo "|" :

```
$ cat doc | more
```

esto ocasiona que la *shell* arranque los comandos `cat` y `more` simultáneamente y que se transmitan la información sincronizadamente. El efecto final es como si se hubiera ejecutado la siguiente secuencia de comandos:

```
$ cat doc > fich_temp
$ more  <  fich_temp
$ rm fich_temp
```

El flujo de control a través de la pipe se maneja automáticamente y de forma transparente. Así, si cat produce información demasiado rápido, se suspenderá su ejecución hasta que more haya retirado de la pipe una cantidad aceptable de información.

Utilización de las pipe desde un programa

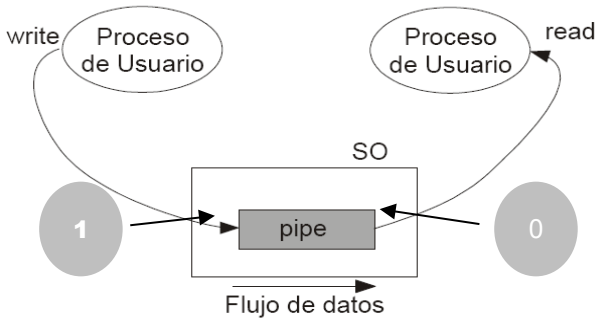
Al ser una clase de ficheros, para utilizar las pipe desde un programa se tendrán que hacer las llamadas `write` y `read`, pero como son un tipo especial de ficheros (soportado en memoria), para crearlas y abrirlas no se empleará las llamadas `creat` y `open` sino la llamada `pipe`.

Esta llamada tiene un argumento que es una pareja de descriptores de ficheros, en definitiva, dos enteros que pueden ser dados a la función en forma de un *array*. El primero se utilizará para realizar las lecturas sobre la pipe y el segundo las escrituras. La llamada, como es habitual, es de tipo entero, si devuelve algo negativo es que se habrá producido un error en su creación.

Su declaración y uso es el siguiente:

```
#include <unistd.h>
int  pareja[2], devuelto;
devuelto = pipe(pareja);
```

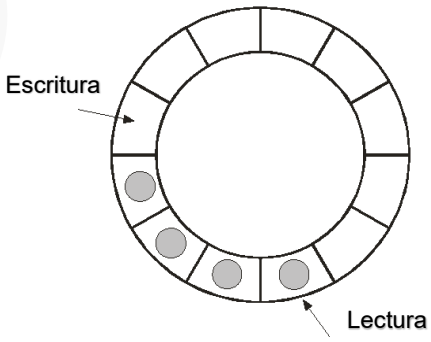
donde `pareja` es un *array* de dos enteros que contienen los descriptores de ficheros que identifican la pipe. Si la llamada tiene éxito `pareja[0]` se abre para leer de la pipe y `pareja[1]` se abre para escribir.



Las llamadas al sistema para lectura y escritura tienen tres argumentos: el primero el descriptor de fichero, el segundo una zona de memoria donde se realizará el intercambio de información (nuestra variable) y el tercero la longitud de datos a intercambiar en bytes.

Las diferencias fundamentales con respecto a los ficheros son:

- Las llamadas son sincronizadas y se bloquean si se lee/escribe de una pipe vacía/llena.
- Las escrituras sobre las pipe son atómicas. En la mayoría de las implementaciones la lectura también.
- Las pipe actúan como una cola circular fifo con dos punteros, uno de lectura y otro de escritura (lectura = borrar, escritura = añadir) como se ve en la figura.

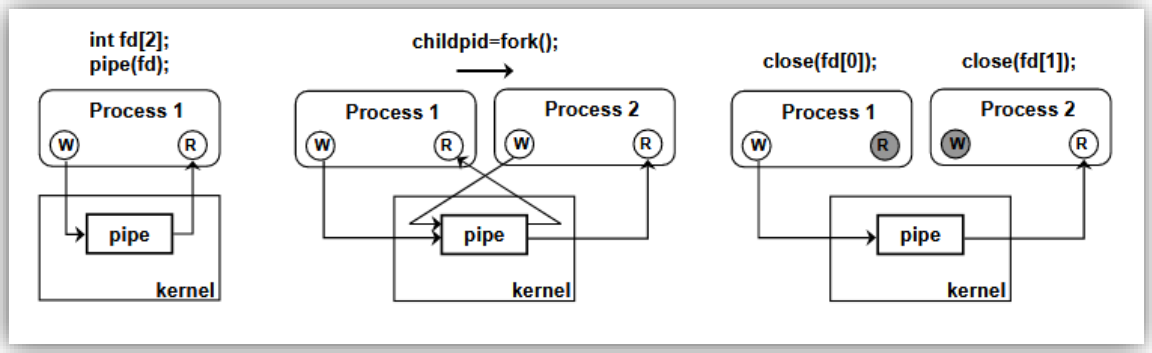


Un ejemplo muy sencillo de uso de las pipe se muestra a continuación, donde se conecta un proceso consigo mismo:

```
#include <stdio.h>
#include <stdlib.h>
#define   tamano   40


int main()
{
       char *msg1 = "El primer mensaje";
       char *msg2 = "El segundo mensaje";
       char *msg3 = "El tercer mensaje";

       char buffer[tamano];    /* de lectura */
       int descriptor[2];
       int i;


       /* se abre la pipe */
       if (pipe(descriptor) < 0)
       {
              perror("Error en la llamada");
              exit(1);
       }

       /* se escribe en la pipe 40 caracteres x 3*/
       write(descriptor[1], msg1, tamano);
       write(descriptor[1], msg2, tamano);
       write(descriptor[1], msg3, tamano);
       /* se lee de la pipe 40 caracteres x 3*/
       for(i=0; i<3; i++)
       {
              read(descriptor[0],buffer, tamano);
              /* al ser puts filtra lo posterior a \n */
              puts(buffer);
       }
      /* se podrian cerrar */
}
```

1) Descárgate el programa `uno.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. Imagínate que no conoces el formato de lo que se ha escrito en la pipe. Cambia la lectura de este programa para que sea genérica (lectura de 120 caracteres). Indica lo que sale por pantalla y búscale una explicación relacionada con las variables inicializadas y su ubicación. Cambia este programa suponiendo que no conocemos la cantidad de caracteres escritos. ¿qué ocurre?

En el ejemplo anterior, el proceso se comunicaba consigo mismo, lo cual no tiene mucha utilidad, por eso las pipe se utilizan en combinación con la llamada `fork`, aprovechando que los procesos creados tienen los mismos descriptores de ficheros abiertos antes de la llamada (lo hemos visto en un apartado



anterior). Date cuenta que, si después del `fork` hacemos una llamada a `exec`, la memoria del proceso se pierde, incluido el *array* de descriptores, y no podríamos usar la pipe a no ser que hagamos algo para transmitir esos enteros.

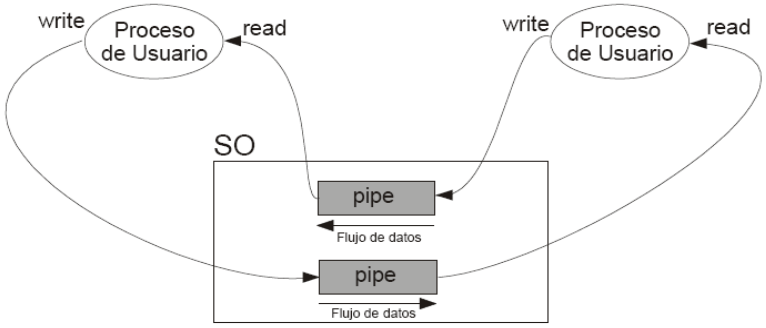
La pipe puede ser cerrada en cada proceso independientemente. Además, como la pipe está compuesta de dos descriptores, uno de lectura y otro de escritura, se puede cerrar cualquiera de ellos. De hecho, se hace por seguridad y en el proceso que lee se cierra el extremo de escritura y viceversa.

Esto se puede ver en el siguiente ejemplo, se crean tres procesos, el abuelo lee datos de la pipe, y cierra el canal de escritura, el padre y el nieto colaboran en la escritura y cierra el canal de lectura:

```
#include <stdio.h>
#include <stdlib.h>
#define    tamano   40

int main()
{
       char *msg1 = "El primer mensaje";
       char *msg2 = "El segundo mensaje";
       char *msg3 = "El tercer mensaje";
       char buffer[tamano];   /* de lectura */
       int descriptor[2];
       int i, creado;

       /* se abre la pipe */
       if (pipe(descriptor) < 0)
       {
              perror("Error en la llamada pipe");
              exit(1);
       }

       if((creado=fork()) < 0)
       {
              perror("Error en el fork 1");
              exit(2);
       }

       if (creado > 0)  /* abuelo lector */
       {
              close(descriptor[1]);
              for(i=0;i<3;i++)
              {
                     read(descriptor[0],buffer,tamano);
                     puts(buffer);
              }
       }

       if (creado == 0)   /* descendientes escritores */
       {
              close(descriptor[0]);
              if((creado=fork()) < 0)
              {
                     perror("Error en el fork 2");
                     exit(3);
              }
              else if (creado > 0)  /* padre */
                     write(descriptor[1],msg1, tamano);
              else (creado == 0)   /* nieto */
              {
                     write(descriptor[1],msg2, tamano);
                     write(descriptor[1],msg3, tamano);
              }
       }
}
```

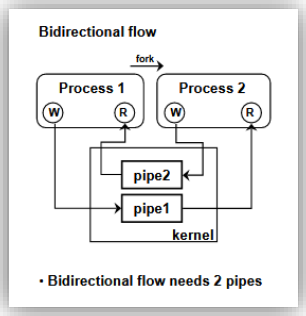
2) Descárgate el programa `dos.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. Crea una copia y haz que la lectura sea realizada por dos procesos, es decir, cambia los papeles de los procesos.

Si se quieren comunicar dos procesos en ambas direcciones se pueden crear dos pipe para realizar este propósito, una en una dirección y otra en otra:



Esto se puede ver en el siguiente ejemplo, donde se crean las dos pipe, una para un sentido y otra para otro:

```
#include <stdio.h>
#include <stdlib.h>
#define   tamano   40
int main()
{
        char *msg1 = "El primer mensaje";
        char *msg2 = "El segundo mensaje";
        char *msg3 = "El tercer mensaje";
        char buffer[tamano];   /* de lectura */
        int descrip1[2];
        int descrip2[2];
        int j, creado;

        /* se abre la pipe primera */
        if (pipe(descrip1) < 0)
        {
                perror("Error al crear pipe 1");
                exit(1);
        }
        /* se abre la segunda pipe */
        if (pipe(descrip2) < 0)
        {
                perror("Error al crear pipe 2");
                exit(2);
        }

        if((creado=fork()) < 0)
        {
                perror("Error en el fork");
                exit(3);
        }

        if (creado > 0)
        {
            close(descrip1[0]);
            write(descrip1[1], msg1, tamano);
            write(descrip1[1], msg2, tamano);
            write(descrip1[1], msg3, tamano);
            /* el read se bloquea si esta vacia */
```



```
            close(descrip2[1]);
            for(j=0;j<3;j++)
            {
                read(descrip2[0], buffer, tamano);
                printf("padre %s \n", buffer);
            }
        }
        if (creado == 0)
        {
                /* el read se bloquea si esta vacia */
                close(descrip1[1]);
                for(j=0;j<3;j++)
                {
                        read(descrip1[0], buffer, tamano);
                        printf("Hijo %s \n", buffer);
                }
                close(descrip2[0]);
                write(descrip2[1], msg1, tamano);
                write(descrip2[1], msg2, tamano);
                write(descrip2[1], msg3, tamano);
        }
}
```

3) Descárgate el programa `doble.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. El orden en la ejecución de los dos procesos dependerá de la planificación de los mismos. Cambia este programa para que los dos procesos empiecen escribiendo y después leyendo y comprueba el resultado.

Consideraciones sobre las pipe

A la hora de trabajar con pipe hay que tener en cuenta varias cosas:

- Como hemos dicho, las operaciones sobre las pipe son sincronizadas y las escrituras son atómicas (en la mayoría de las implementaciones la lectura también, pero no es obligatorio según el estándar). Hay dos situaciones en que esto no se da:
  1. Sólo cuando se van a escribir más bytes que el límite **pipe_BUF** (en `limits.h`*)* no se realiza de forma atómica. Básicamente la información no cabe en la pipe.
  2. Si O_NONBLOCK está puesto con la llamada `fcntl`.
- El tamaño de las pipe es limitado (pipe_BUF). La escritura se realiza hasta llenarla de forma atómica, después el proceso se suspende.
- Si el proceso lee de una pipe vacía se suspende.
- Si la pipe está desconectada y sólo queda(n) un proceso lector `read`, la lectura devuelve ceros y no se suspende (nadie va a escribir más).
- Si sólo queda un proceso escritor, éste recibe la señal SIGpipe (si no está preparado muere) ya que nadie va a leer lo que escriba.

Un ejemplo para descubrir el tamaño de las pipe de tu sistema (además de comprobar la constante pipe_BUF) aparece a continuación, donde se mezclan las señales de alarma y las pipe:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <limits.h>
#define  TAMANO   1

int cuenta = 0;      /* global para acceder desde la manejadora */

int main()
{
```

```
        char msg = 'A';
        int descriptor[2];
        int creado;
        void alarma();

        /* recojo la senal de alarma */
        signal(SIGALRM, alarma);

        /* se abre la pipe */
        if (pipe(descriptor) < 0)
        {
               perror("Error en la llamada");
               exit(1);
        }

        for(;;) /* infinito, se saldra por senal no manejada */
        {
               alarm(1);  /* se activa la alarma */
               write(descriptor[1], &msg, tamano);
               /* si pasa la escritura se desactiva */
               alarm(0);
               /* se aumenta el tamano */
               cuenta++;
               if((cuenta%1000) == 0)
                      printf("Cuenta %d \n", cuenta);
        }
}

void alarma()
{
        printf("Se paro en %d \n",cuenta);
        printf("No se deberia escribir mas de %d \n",pipe_BUF);
        exit(0);
}
```

4) Descárgate el programa `tama.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. Indica cuál es el tamaño de tu sistema y si coincide con la constante pipe_BUF.

Las pipe con nombre: fifo - Opcional.

Una característica que define a las pipe es que se basan en un mecanismo de herencia entre procesos y para que se puedan utilizar, los procesos deben ser padres e hijos, pero no pueden utilizar las llamadas a `exec`, ya que esto haría que desapareciera el *array* de dos descriptores (salvo que se pase convertida a *string* como argumento).

Para evitar este inconveniente existen unas pipe modificadas que se llaman pipe con nombre o fifo. Comparten las mismas propiedades, pero se crean y abren de diferente manera, ya que al crearlas se las da un nombre como si fueran ficheros de verdad.

Para ello se utiliza la llamada `mknod`, que sirve para crear dispositivos, ficheros especiales y ficheros ordinarios:

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

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

El primer argumento será el nombre del fichero (de la fifo), en el segundo argumento, a través de constantes definidas es `stat.h`, se dice el tipo de elemento a crear, en nuestro caso fifo con `S_IFIFO` (vale 010 en octal) y el tercero se deja a cero (sólo sirve en caso de crear dispositivos). Otras constantes son S_IFREG (ficheros regulares), S_IFCHR (dispositivos modo carácter), S_IFBLK (dispositivos modo bloque), S_IFIFO o S_IFSOCK (para sockets de redes). Estás constantes se combinan con los permisos de acceso (vistos con el comando `chmod`) para crear el elemento con los permisos correspondientes.

Dos maneras diferentes de creación de una fifo aparecen a continuación:

```
if (mknod("fifo", 010600, 0) < 0)  será S_IFIFO | S_IRUSR | S_IWUSR
      perror("llamada mknod");

if (mkfifo("fifo", 0600) < 0)       será  S_IRUSR | S_IWUSR
      perror("llamada mkfifo");
```

En el primer caso es una combinación de la constante `S_IFIFO` que es 010 con los permisos del fichero (ver guion 1 apartado `chmod`). En el segundo caso sólo aparecen los permisos ya que es una llamada especializada (`0600` es rw para mí, nada al resto). Siempre se recomienda usar constantes literales en vez de códigos octales.

De igual modo que se pueden crear con llamadas al sistema, también se pueden crear con el comando `mknod` o `mkfifo` antes de realizar la ejecución del programa. Una manera habitual de que falle la llamada es intentar crear una fifo ya existente, antes se debe borrar.

```
$ mknod –m=S_IRUSR fifo p
$ mkfifo –m=S_IWUSR fifo
$ rm fifo
```

Una vez creada, hay que abrirla. Dado que las pipe/fifo son sincronizadas, la llamada de apertura bloqueará al proceso hasta que la fifo sea abierta desde al menos dos procesos.

La llamada de apertura es la misma que la de ficheros:

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

int open(const char *pathname, int flags);
```

Los *flags* están definidos en `stat.h`, en ellos se indicará si la fifo se ha abierto para lectura (`O_RDONLY`) o para escritura (`O_WRONLY`). Recordemos que la fifo ya está creada.

Un ejemplo de uso equivalente a los vistos en las pipe aparece a continuación:

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

#define   tamano   40
#define   FIFO  010000    /* mejor S_IFIFO ya definida */
#define   PERMISO 0600    /* mejor S_IRUSR|S_IWUSR ya definidas */

int main()
{
       char *msg1 = "El primer mensaje";
       char *msg2 = "El segundo mensaje";
       char *msg3 = "El tercer mensaje";

       char buffer[tamano];
```

```
 int desc;
 int j, creado;

 /* se crea la fifo, se podria hacer desde comandos en la Shell */
if(mknod("nombre", FIFO | PERMISO, 0) < 0)
        perror("error en mknod");

 /* Seria mejor utilizar esta llamada especifica, una u otra */
 if (mkfifo("nombre", PERMISO) < 0)
        perror("error en mkfifo");

 if ((creado=fork())>0)  /* el padre */
 {
        /* se abre para lectura sincronizada */
        if((desc=open("nombre",O_RDONLY)) < 0)
        {
               perror("error en open");
               exit(1);
        }
        /* se lee de la fifo */
        for(j=0; j<3; j++)
        {
               read(desc, buffer, tamano);
               puts(buffer);
       }
       close(desc);
 }

 if(creado==0)   /* el hijo */
{
        /* se abre para escritura sincronizada */
        if((desc=open("nombre", O_WRONLY)) < 0)
        {
               perror("error en open");
             exit(1);
       }
       /* se escribe en la fifo */
       write(desc, msg1, tamano);
       write(desc, msg2, tamano);
       write(desc, msg3, tamano);
       /* se cierra la fifo */
       close(desc);
 }
}
```

Si después de la ejecución utilizas el comando `ls`, verás que tienes en el directorio un fichero especial que es la fifo (está clasificado con la letra p de pipe). No intentes ejecutar el programa de nuevo, ya que al existir la fifo, te dará error. Antes bórrala.