ÍNDICE

1. PROGRAMACIÓN DISTRIBUÍDA UTILIZANDO PIPES

1.1. El concepto de pipe

Ya hemos visto como se crea un proceso y como sincronizarlos de una manera muy tosca mediante la llamada al sistema **wait()**, vamos a ver a ahora el mecanismo de sincronización con más tradición en UNIX, los pipes.

Los canales de comunicación más sencillos para un proceso son su entrada estándar (descriptor de fichero 0), y su salida estándar (descriptor de fichero 1). Estos dos ficheros son heredados del proceso padre (el shell p.e.) y por defecto conectados al teclado (el descriptor 0) y la pantalla (el descriptor 1)

La mayoría de los usuarios de UNIX saben como redireccionar los streams de entrada y salida estándar.

```
who > temp_file
sort < temp_file
```

También la mayoría de los usuarios de UNIX están habituados a emplear el concepto de pipe, y saben como solicitar al shell la creación de uno, con comandos como los siguientes:

```
who | sort
```

Este comando ejecuta who y sort como dos procesos concurrentes, con la salida estándar de who conectada la entrada estándar de sort.

El uso de pipes en esta forma constituye el núcleo de la filosofía UNIX que da solución a algoritmos distribuidos, donde la solución se plantea mediante la combinación de dos o más herramientas de propósito general (como puedan ser aquí who y sort).

1.2. Creación de pipes

El pipe se trata de un mecanismo unidireccional constituido por un buffer en la zona de kernel, que presenta un punto de entrada donde se puede escribir y un punto de salida del que se puede leer. Los pipes se crean mediante la llamada al sistema **pipe(*dfs)**, el argumento dfs es un puntero a un array de dos enteros. **pipe()** devuelve en dfs los descriptores de dos ficheros, en el elemento 0 de dfs devuelve el descriptor del fichero de salida de un pipe, y en el elemento 1 de dfs devuelve el descriptor del ficheros de entrada de un pipe (Figura 1). Al finalizar el proceso que ha creado el pipe, éste se destruye, no siendo necesaria su eliminación.

Los pipes son empleados comúnmente para establecer una comunicación unidireccional entre dos procesos hijos de un mismo padre. Sea por ejemplo el proceso A, con hijos B y C que se comunican a través de un pipe. Los estados por los que se pasará hasta que se tiene un pipe operativo suelen ser los siguientes (Figura 2):

- **Estado1**: El proceso A crea un pipe y recibe los descriptores de entrada y salida del en el array p.

- **Estado2**: El proceso A invoca la llamada del sistema **fork()** dos veces para crear los procesos hijo B y C. Ambos hijos heredan los descriptores del pipe, de modo que en este momento los tres procesos tienen los descriptores abiertos para ambos extremos del pipe.
- **Estado3**: Cada proceso cierra los descriptores del pipe que no necesita. El proceso B cierra el descriptor de salida, C cierra el descriptor de entrada y A cierra ambos descriptores.
- **Estado4**: Los procesos B y C, habitualmente ejecutarán otros programas. Los descriptores de ficheros de los extremos del pipe, permanecerán intactos durante la ejecución de las llamadas a **exec()**. B escribirá en el pipe con la llamada al sistema **write()**, C leerá del pipe empleando la llamada al sistema **read()**.

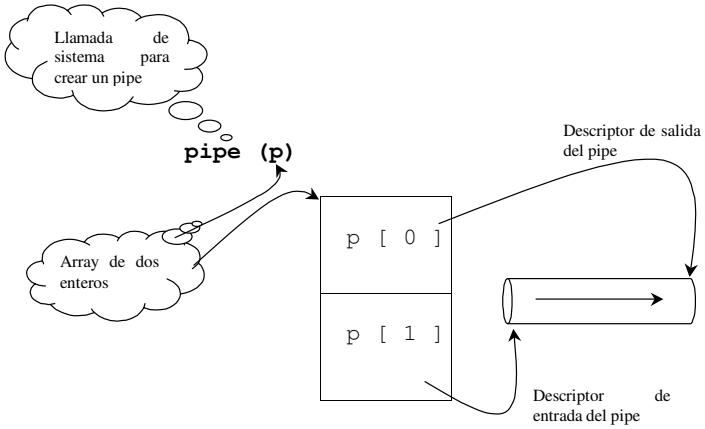

Figura 1. Creación de un pipe

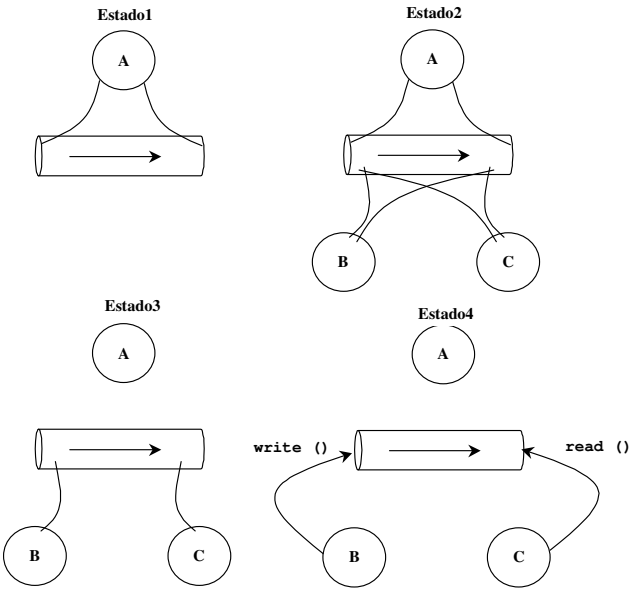

Figura 2. Estados para el establecimiento de un pipe entre dos hijos

1.3. Sincronización empleando pipes

Los pipes tienen un tamaño finito por lo que esto puede suponer una pérdida de sincronismo entre la entrada y la salida. Si el proceso de escritura en la entrada trata de escribir cuando éste está lleno (se ha llegado al máximo de su capacidad), el proceso será bloqueado hasta que el proceso de salida (lectura) consuma algunos datos del pipe.

Es particularmente interesante comprender el comportamiento del proceso de salida del pipe. Consideremos un programa que procesa el flujo de lectura del pipe, que lee hasta 100 bytes a la vez, y que continua hasta que no haya más datos disponibles, es decir hasta que la llamada al sistema **read()** devuelva 0. Este proceso tendría la siguiente forma:

```
while ( (count = read(fd, bugffer, 100)) > 0) {
/*process data in buffer */
}
```

Supongamos que el descriptor fd referencia la salida de un pipe, y el proceso de entrada del pipe ha escrito ya 230 bytes de datos. En el proceso de salida del pipe, las dos primeras iteraciones leerán 100 bytes y la tercera 30 bytes. En la cuarta iteración, la función **read()** quedará bloqueada, porque el pipe está vacío. Si el proceso de entrada escribiese más datos, el proceso de salida recibiría inmediatamente esos datos. Sin embargo, si el proceso de entrada al pipe hubiera finalizado, y cerrado el descriptor del entrada del pipe (o simplemente sale), la llamada **read()** del proceso del salida del pipe devolvería un fin de fichero (EOF). En resumen:

Siempre que haya algún descriptor abierto sobre el flujo de entrada del pipe la lectura del flujo salida del pipe no devolverá EOF, cuando todos los descriptores asociados al flujo de entrada del pipe estén cerrados entonces la lectura del flujo de salida del pipe obtendrá un EOF.

1.4. Un ejemplo de pipeline

Anteriormente vimos los cuatro estados empleados para establecer un pipe entre dos procesos hijo. El ejemplo clásico visto hace referencia a la invocación desde el shell de dos procesos como los siguientes:

```
who | sort
```

Según la Figura 2 el proceso A es el shell, el proceso B ejecutará el comando who y el proceso C ejecutará sort. En realidad si es el shell el que crea el pipe la cosa no es tan simple como los cuatro estados descritos anteriormente. Todo lo que hace who es escribir a la salida estándar (descriptor 1), y sort simplemente lee de la entrada estándar. De alguna manera, el shell tiene que coger la salida de who y conectarla con la entrada del pipe, y la entrada del sort a la salida del pipe.

La llamada al sistema **dup2()** puede emplearse para rematar estas últimas conexiones. La llamada **dup(old,new)** toma como parámetro el descriptor old y lo duplica sobre el descriptor new. Si new ya es un descriptor abierto, se cerrará primero. En otras palabras, después de la llamada, new referenciará al mismo flujo que old.

Por ejemplo tras la ejecución del fragmento:

```
int p[2];
pipe(p);
dup(p[1], 1);
```

La salida estándar del proceso se conectará a la entrada del pipe.

Con **dup2()** podremos canalizar la salida del nuestro comando who a la entrada del comando sort mediante un pipe empleando el siguiente programa:

```
/* File whosort.c */

main()
{
  int fds[2];
  pipe(fds);                                        /*Crea un pipe */
  /* El primer hijo reconecta su entrada estándar (stdin) al flujo de salida del pipe y
cierra el descriptor de la entrada del pipe
  */
  if (fork()  == 0) {
          dup2(fds[0], 0);
          close(fds[1]);
          execpl("sort", "sort", 0);
  }
  /* El segundo hijo reconecta su salida estándar (stdout) a la entrada del pipe y cierra el
descriptor de la salida del pipe
  */
  else if ( fork() == 0) {
          dup2(fds[1], 1);
          close(fds[0]);
          execlp("who", "who", 0);
  }
  /*El padre cierra ambos descriptores del pipe
          y espera la finalización de sus hijos
  */
  else {
  close(fds[0]);
  close(fds[1]);
  wait(0);
  wait(0);
  }
}
```

La llamada a **close()** ejecutada por el primer hijo para cerrar la entrada es esencial, ya que sin esa llamada ese hijo nunca vería el EOF al leer del pipe. Aún cuando el proceso no tuviera la intención de escribir en el pipe, la realidad es que se mantiene el descriptor abierto sobre la entrada del pipe previniendo de la lectura de un EOF.

¿Por qué los pares de llamadas a close y wait al final del bloque del padre? Las llamadas a wait está claro que es para que el proceso padre no finalice hasta que sus hijos lo hagan. En cuanto a llamadas a **close()**, es por el motivo ya explicado, sin esas llamadas el proceso sort, nunca leería un EOF del pipe.

1.5. Limitaciones de los pipe

Aún cuando los pipes puedan parecer elegantes, tienen ciertas limitaciones:

- Los pipes son unidireccionales. Aunque siempre se pueden crear dos pipes uno para cada dirección.
- No ofrecen mecanismos de autentificación del proceso del otro extremo. Por ejemplo el proceso de lectura del pipe no tiene idea de quien es el proceso que le está enviando los datos.
- Los pipes deben ser creados antes que los procesos que los usan. Dos procesos no relacionados no pueden emplear un pipe para conectarse. Los procesos deben tener un ascendiente común que cree el pipe y les trasmita por herencia los descriptores del mismo.
- Los pipes no funcionan sobre la red. Ambos procesos deben de ejecutarse en la misma máquina.

1.6. Pipes con nombre

Los pipes con nombre funcionan muy parecido a los pipes vistos hasta ahora, con la diferencia de que salvan una de sus restricciones: permiten ser empleados por procesos no relacionados. Un pipe con nombre tiene una entrada en el sistema de ficheros, y por lo tanto un propietario y unos permisos. Los pipes con nombre se crean con la llamada al sistema **mknod()** o con el comando mknod y aparecerá en la columna de directorio con una ‘p’.

Los pipes con nombre son tan fáciles de usar como los pipes ordinarios, un fichero lo abre para escritura otro lo hace la lectura. El resto de comportamientos es igual al de los pipes ordinarios.

ANEXO A

```
/* File whosort.c */

#include <unistd.h>
main()
{
  int fds[2];
  pipe(fds);                                        /*Crea un pipe */
/* El primer hijo reconecta su entrada estándar (stdin) al flujo de salida del pipe y
cierra su descriptor de la entrada del pipe
*/
if (fork()  == 0) {
 dup2(fds[0], 0);
 close(fds[1]);
execlp("sort", "sort", 0);
}
/* El segundo hijo reconecta su salida estándar (stdout) a la entrada del pipe y cierra el
descriptor de la salida del pipe
*/
else if ( fork() == 0) {
dup2(fds[1], 1);
close(fds[0]);
execlp("who", "who", 0);
}
/*El padre cierra ambos descriptores del pipe
y espera la finalización de sus hijos
*/
else {
  close(fds[0]);
  close(fds[1]);
  wait(0);
  wait(0);
  }
}
```

```
# Escuela Técnica Superior de Ingeniería Informática (Gijón)
# Sistemas de Computación (Curso 5)
# Aplicaciones Distribuidas en Unix con pipes

#************************************************
# Variables
#************************************************


CC=cc
CFLAGS=
CLIENTE=whosort

todo : $(CLIENTE)


#************************************************
# Generación del cliente/servidor
#************************************************

$(CLIENTE) : whosort.c
        $(CC) $(CFLAGS) -o $(CLIENTE) whosort.c

#************************************************
# Utilidades
#************************************************

limpiaobjs :
        rm -f *.o
```

BIBLIOGRAFÍA

Unix. Distributed Programming

Autor: Chris Brown
Editorial: Prentice-Hall
ISBN: 0-13-075896-5

Advanced Programming in the Unix Environment

Autor: W. Richard Stevens
Editorial: Addison-Wesley
ISBN: 0-201-56317-1

Unix Network Programming

Autor: W. Richard Stevens
Editorial: Prentice-Hall
ISBN: 0-13-949876-1

Unix. Manual Pages