Práctica 2.4: Tuberías

Objetivos

Las tuberías ofrecen un mecanismo sencillo y efectivo para la comunicación entre procesos en un mismo sistema. En esta práctica veremos los comandos e interfaz para la gestión de tuberías, y los patrones de comunicación típicos.

Contenidos

Preparación del entorno para la práctica

Esta práctica únicamente requiere las herramientas y entorno de desarrollo de usuario.

Tuberías sin nombre

Las tuberías sin nombre son entidades gestionadas directamente por el núcleo del sistema y son un mecanismo de comunicación unidireccional eficiente para procesos relacionados (padre-hijo). La forma de compartir los identificadores de la tubería es por herencia (en la llamada `fork(2)`).

Ejercicio 1. Escribir un programa que emule el comportamiento de la shell en la ejecución de una sentencia en la forma: `comando1 argumento1 | comando2 argumento2`. El programa creará una tubería sin nombre y creará un hijo:

- El proceso padre redireccionará la salida estándar al extremo de escritura de la tubería y ejecutará `comando1 argumento1`.
- El proceso hijo redireccionará la entrada estándar al extremo de lectura de la tubería y ejecutará `comando2 argumento2`.

Probar el funcionamiento con una sentencia similar a: `./ejercicio1 echo 12345 wc -c`

***Nota*:** Antes de ejecutar el comando correspondiente, deben cerrarse todos los descriptores no necesarios.

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

#define PIPE_READ 0
#define PIPE_WRITE 1

int main(int argc, char** argv){
   if (argc < 2) {
      printf("Error: Tienes que pasar los comandos\n");
   }

   int fd[2];
   pipe(fd);
```

```
   int pid = fork();
   switch(pid) {
      case -1:
         perror("ERROR al crear el hijo");
         return -1;
         break;

      case 0: //Hijo
         dup2(fd[PIPE_READ], 0);  //Redireccionamos entrada estándar al extremo de lectura de la
tubería
         close(fd[PIPE_WRITE]);
         close(fd[PIPE_READ]);

         if(execlp(argv[3], argv[3], argv[4], 0) == -1){ //Ejecutamos comando2 argumento2
            printf("Error al ejecutar comando 2");
            return -1;
         }

         return 0;
         break;

      default: //Padre
         dup2(fd[PIPE_WRITE], 1); //Redireccionamos salida estándar al extremo de escritura de la
tubería
         close(fd[PIPE_WRITE]);
         close(fd[PIPE_READ]);

         if(execlp(argv[1], argv[1], argv[2], 0) == -1){ //Ejecutamos comando1 argumento1
            printf("Error al ejecutar comando 1");
            return -1;
         }

         return 0;
         break;
   }

   return 0;
}
```

Lo que hacemo es con echo 12345 escribir en la entrada de escritura de la tubería eso, pues hemos cambiado la salida estandar del padre (que ejecuta el comando 1) por esta y el hijo (que ejecuta el comando 2) al ejecutar wc -c (lo que hace es algo de contar y tal) lee de la entrada de lectura de la tubería (pues su entrada ahora no es la estándar pues la hemos cambiado) y coge el 12345 y hace lo que tenga que hacer

Ejercicio 2. Para la comunicación bi-direccional, es necesario crear dos tuberías, una para cada sentido: `p_h` y `h_p`. Escribir un programa que implemente el mecanismo de sincronización de parada y espera:

- El padre leerá de la entrada estándar (terminal) y enviará el mensaje al proceso hijo, escribiéndolo en la tubería `p_h`. Entonces permanecerá bloqueado esperando la confirmación por parte del hijo en la otra tubería, `h_p`.
- El hijo leerá de la tubería `p_h`, escribirá el mensaje por la salida estándar y esperará 1 segundo. Entonces, enviará el carácter `'1'` al proceso padre, escribiéndolo en la tubería `h_p`, para indicar que está listo. Después de 10 mensajes enviará el carácter `'q'` para indicar al padre que finalice.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

#define PIPE_READ 0
#define PIPE_WRITE 1

int main(){
   //Tuberías Padre-Hijo Hijo-Padre
   int tuberia_ph[2];
   int tuberia_hp[2];
   pipe(tuberia_ph);
   pipe(tuberia_hp);

   char buffer[256];
   bool fin = false;
   int cont  = 0;

   int pid = fork();
   switch(pid) {
      case -1:
         perror("ERROR al crear el hijo");
         return -1;
         break;

      case 0: //Hijo
         close(tuberia_ph[PIPE_WRITE]); //Cerramos los extremos que no vamos a utilizar. El hijo lee
de p-h y escribe en h-p
         close(tuberia_hp[PIPE_READ]);

         while(!fin){
            int rc = read(tuberia_ph[PIPE_READ], buffer, 255);
            buffer[rc] = '\0';
            printf("[Hijo] Mensaje %d recibido: %s\n", cont+1, buffer);

            sleep(1);

            if(++cont == 10){
               write(tuberia_hp[1], "q", 1); //Escribimos en la tibería Hijo-Padre el caracter que toque
según el nº de mensajes recibidos
               fin = true;
            }
```

```
            else{
               write(tuberia_hp[1], "l", 1);
            }
         }

         close(tuberia_ph[PIPE_READ]); //Cerramos los extremos que no vamos a utilizar. El hijo lee
de p-h y escribe en h-p
         close(tuberia_hp[PIPE_WRITE]);

         return 0;
         break;

      default: //Padre
         close(tuberia_ph[PIPE_READ]);
         close(tuberia_hp[PIPE_WRITE]);

         while(!fin){
            printf("[Padre] Mensaje a enviar: ");
            scanf("%s", buffer);
            write(tuberia_ph[PIPE_WRITE], buffer, strlen(buffer) + 1); //Escribimos el mensaje en la
tuberia Padre-Hijo
            read(tuberia_hp[PIPE_READ], buffer, 1); //Leemos el carcater que nos envía el hijo en la
tubería Hijo-Padre

            if(buffer[0] == 'q'){
               fin = true;
            }
         }

         //Cerramos el resto de extremos
         close(tuberia_ph[PIPE_WRITE]);
         close(tuberia_hp[PIPE_READ]);

         return 0;
         break;
   }

   return 0;
}
```

Tuberías con nombre

Las tuberías con nombre son un mecanismo de comunicación unidireccional, con acceso de tipo FIFO, útil para procesos sin relación de parentesco. La gestión de las tuberías con nombre es igual a la de un archivo ordinario (`open`, `write`, `read`…). Revisar la información en `fifo(7)`.

Ejercicio 3. Usar la orden `mkfifo` para crear una tubería con nombre. Usar las herramientas del sistema de ficheros (`stat`, `ls`…) para determinar sus propiedades. Comprobar su funcionamiento usando utilidades para escribir y leer de ficheros (ej. `echo`, `cat`, `tee`...).

mkfifo [OPTION]... NAME… : Crea tuberías con nombre con el nombre NAME
-m, --mode=MODE
set file permission bits to MODE, not a=rw - umask

-Z set the SELinux security context to default type

--context[=CTX]
like -Z, or if CTX is specified then set the SELinux or SMACK security context to CTX

$ mkfifo TuberiaPrueba
$ ls -l
prw-rw-r-- 1 usuarioso usuarioso 0 nov 29 10:00 TuberiaPrueba (p indica que es de tipo pipe)
$ stat TuberiaPrueba
Fichero: TuberiaPrueba
Tamaño: 0 Bloques: 0 Bloque E/S: 4096 `fifo'
Dispositivo: 801h/2049d Nodo-i: 543718 Enlaces: 1
Acceso: (0664/prw-rw-r--) Uid: (1000/usuarioso) Gid: (1001/usuarioso)
Acceso: 2021-11-29 10:00:47.710817304 +0100
Modificación: 2021-11-29 10:00:47.710817304 +0100
Cambio: 2021-11-29 10:00:47.710817304 +0100
Creación: -

TERMINAL 1:
$ echo "prueba" > TuberiaPrueba
Se queda en espera

TERMINAL 2:
$ cat TuberiaPrueba
prueba
Acaba la espera en Terminal 1 y en el 2

Ejercicio 4. Escribir un programa que abra la tubería con el nombre anterior en modo sólo escritura, y escriba en ella el primer argumento del programa. En otro terminal, leer de la tubería usando un comando adecuado.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>


int main(int argc, char **argv){
  if (argc < 2) {
    printf("ERROR: Tienes que pasar un argumento\n");
    return -1;
  }

  int fd = open("TuberiaPrueba", O_WRONLY);
  write(fd, argv[1], strlen(argv[1]));
  close(fd);

  return 0;
}
```

TERMINAL 1:

```
$ ./ej4 Hola
    *Se queda en espera
```

TERMINAL 2:

```
$ cat ./TuberiaPrueba
Hola
    *Termina la espera del otro terminal
```

Multiplexación síncrona de entrada/salida

Es habitual que un proceso lea o escriba de diferentes flujos. La llamada `select(2)` permite multiplexar las diferentes operaciones de E/S sobre múltiples flujos.

Ejercicio 5. Crear otra tubería con nombre. Escribir un programa que espere hasta que haya datos listos para leer en alguna de ellas. El programa debe mostrar la tubería desde la que leyó y los datos leídos. Consideraciones:

- Para optimizar las operaciones de lectura usar un *buffer* (ej. de 256 bytes).
- Usar `read(2)` para leer de la tubería y gestionar adecuadamente la longitud de los datos leídos.
- Normalmente, la apertura de la tubería para lectura se bloqueará hasta que se abra para escritura (ej. con `echo 1 > tuberia`). Para evitarlo, usar la opción `O_NONBLOCK` en `open(2)`.
- Cuando el escritor termina y cierra la tubería, `read(2)` devolverá `0`, indicando el fin de fichero, por lo que hay que cerrar la tubería y volver a abrirla. Si no, `select(2)` considerará el descriptor siempre listo para lectura y no se bloqueará.

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

int main(){
   char buffer[256];

   //Creamos las dos tuberias
   mkfifo("./Tuberia1", 0644);
   mkfifo("./Tuberia2", 0644);
   int fd1 = open("./Tuberia1", O_NONBLOCK|O_RDONLY);
   int fd2 = open("./Tuberia2", O_NONBLOCK|O_RDONLY);

   int max = fd1;
   if(fd2 > fd1){
      max = fd2;
   }

   //Creo conjunto, lo inicializao y añado las tuberias
   fd_set set;
   FD_ZERO(&set);
   FD_SET(fd1, &set);
   FD_SET(fd2, &set);

   int ready = select(max + 1, &set, NULL, NULL, NULL);
   while(ready == 0){
      ready = select(max + 1, &set, NULL, NULL, NULL);
      if(ready == -1){
         printf("ERROR al hacer el select");
         return -1;
      }
```

```
    } //Cuando salga es porque se ha producido un cambio

    if(FD_ISSET(fd1, &set)){ //tuberia 1 lista
        int bytes = read(fd1, &buffer, 256);
        buffer[bytes] = '\0';
        printf("[Tubería 1] %s", buffer);
        close(fd1);
        fd1 = open("./Tuberia1", O_NONBLOCK|O_RDONLY);
    }
    if(FD_ISSET(fd2, &set)){ //tuberia 2 lista
        int bytes = read(fd2, &buffer, 256);
        buffer[bytes] = '\0';
        printf("[Tubería 2] %s", buffer);
        close(fd2);
        fd2 = open("./Tuberia2", O_NONBLOCK|O_RDONLY);
    }

    return 0;
}
```

TERMINAL 1

```
$ ./ej5
*Se queda esperando hasta * en terminal 2
[Tubería 2] Artu artu artu artu artu artuuuuuuuuuuuuuuuuuu
$ ./ej5
*Se queda esperando hasta ** en terminal 2
[Tubería 1] Artu artu artu artu artu artuuuuuuuuuuuuuuuuuu123
```

TERMINAL 2

```
$ echo "Artu artu artu artu artu artuuuuuuuuuuuuuuuuuu" > Tuberia2
*
$ echo "Artu artu artu artu artu artuuuuuuuuuuuuuuuuuu123" > Tuberia1
**
```