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 ...

int main(int argc, char **argv){

      if (argc != 5) {
         fprintf(stderr, "Usage: %s <string>\n", argv[0]);
         exit(EXIT_FAILURE);
      }

      int pipefd[2];
      pipe(pipefd);
      pid_t pid = fork();
```

```
      if(pid == -1){
         printf("fork\n");
         exit(EXIT_FAILURE);
      }
      else if(pid == 0){ //Hijo
         close(pipefd[1]);
         dup2(pipefd[0], 0);
         execlp(argv[3], argv[3], argv[4], NULL);
      }
      else{ // Padre
         close(pipefd[0]);
         dup2(pipefd[1],1);
         execlp(argv[1], argv[1], argv[2], NULL);
      }
      return 0;
}
```

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 ...

int main(int argc, char **argv){

      if (argc != 1) {
         fprintf(stderr, "Usage: %s\n", argv[0]);
         exit(EXIT_FAILURE);
      }

      /* Creación de dos tuberias */
      int p_h[2];
      int h_p[2];
      pipe(p_h);
      pipe(h_p);

      pid_t pid = fork();

      if(pid == -1){
         printf("fork\n");
         exit(EXIT_FAILURE);
      }
```

```
	//Hijo
	else if(pid == 0){
	close(p_h[1]);
	close(h_p[0]);

	char recibe_padre[256];
	char *senial_lq = "l";

	for(int i = 0; i < 10; i++){
		sleep(1);
		int size_r = read(p_h[0], recibe_padre, sizeof(recibe_padre));
		printf("[HIJO] Mensaje recibido: %s\n", recibe_padre);
		sleep(1);
		write(h_p[1], senial_lq, 1);
	}
	senial_lq = "q";
	write(h_p[1], senial_lq, 1);

	close(p_h[0]);
	close(h_p[1]);
	}

	// Padre
	else{
	close(p_h[0]);
	close(h_p[1]);

	char envia_padre[256];
	char *msg_hijo = "l";

	while(msg_hijo[0] != 'q'){
		printf("[PADRE] Escribe el mensaje: \n");
		int size_e = read(0, envia_padre, sizeof(envia_padre));
		write(p_h[1], envia_padre, size_e);

		while(msg_hijo[0] != 'l' && msg_hijo[0] != 'q'){
		   read(h_p[0], msg_hijo, sizeof(msg_hijo));
		}
		sleep(1);
	}

	close(p_h[1]);
	close(h_p[0]);
	}

	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`...).

TERMINAL 1:

```
$ man mkfifo
$ mkfifo tuberia

$ ls -l
total 48
-rwxr-xr-x 1 luis luis 17064 dic 26 16:15 ej1
-rw-r--r-- 1 luis luis   700 dic 26 16:20 ej1.c
-rwxr-xr-x 1 luis luis 17144 dic 27 13:24 ej2
-rw-r--r-- 1 luis luis  1593 dic 27 13:24 ej2.c
prw-r--r-- 1 luis luis     0 dic 27 13:27 tuberia

[luis@luis p4_ASOR]$ stat tuberia
  File: tuberia
  Size: 0           Blocks: 0          IO Block: 4096   fifo
Device: 802h/2050d    Inode: 2888590   Links: 1
Access: (0644/prw-r--r--)  Uid: ( 1000/ luis)   Gid: ( 1000/      luis)
Access: 2020-12-27 13:27:57.218713733 +0100
Modify: 2020-12-27 13:27:57.218713733 +0100
Change: 2020-12-27 13:27:57.218713733 +0100
 Birth: 2020-12-27 13:27:57.218713733 +0100

$ echo "prueba" > tuberia
```

TERMINAL 2:

```
$ cat tuberia
prueba
```

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 ...

//Ejemplo suponiendo creada la tuberia llamada "tuberia"
int main(int argc, char **argv){

      if (argc < 2) {
         printf("ERROR: Falta el mensaje para enviar por argumento.\n");
         return -1;
      }

      int fd = open("tuberia", O_WRONLY);

      argv[1] = strcat(argv[1], "\n");  //Para poner el salto de linea
```

```
        write(fd, argv[1], strlen(argv[1]));

        close(fd);

        return 0;
}
```

```
//Ejemplo sin haber creado previamente la tuberia
int main(int argc, char **argv){

        if (argc < 2) {
           printf("ERROR: Falta el mensaje para enviar por argumento.\n");
           return -1;
        }

        char *tuberia = "tuberia";
        mkfifo(tuberia, 0777);

        int fd = open(tuberia, O_WRONLY);

        write(fd, argv[1], strlen(argv[1]));

        close(fd);

        return 0;
}
```

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

```c
#include <string.h>
#include <stdlib.h>
#include <sys/select.h>

int main(int argc, char **argv){

	char *tuberia1 = "tuberia1";
	if(mkfifo(tuberia1, 0777) == -1){
	   printf("Error mkfifo primera tuberia.\n");
	   exit(EXIT_FAILURE);
	}

	char *tuberia2 = "tuberia2";
	if(mkfifo(tuberia2, 0777) == -1){
	   printf("Error mkfifo segunda tuberia.\n");
	   exit(EXIT_FAILURE);
	}

	int fd1 = open(tuberia1, O_RDONLY | O_NONBLOCK);
	int fd2 = open(tuberia2, O_RDONLY | O_NONBLOCK);

	if(fd1 == -1 || fd2 == -1){
	   printf("Error en la apertura de las tuberias.\n");
	   close(fd1);
	   close(fd2);
	   exit(EXIT_FAILURE);
	}

	char buffer[256]; //Buffer para los datos enviados
	int cambios = 0;
	fd_set set;
	int size_read = 0;

	while(cambios != -1){
	   FD_ZERO(&set);
	   FD_SET(fd1, &set);
	   FD_SET(fd2, &set);

	   cambios = select((fd1 > fd2)? fd1+1 : fd2+1, &set, NULL, NULL, NULL);

	   if(cambios){
		if(FD_ISSET(fd1, &set)){
		   size_read = read(fd1, buffer, sizeof(buffer));

		   //Si el buffer anterior tiene mas tamaño relleno hay que despreciarlo.
		   buffer[size_read] = '\0';

		   if(size_read == 0){
		       close(fd1);
		       fd1 = open(tuberia1, O_RDONLY | O_NONBLOCK);
		   }
		   else printf("Tuberia 1: %s", buffer);
		}
		if(FD_ISSET(fd2, &set)){
		   size_read = read(fd2, buffer, sizeof(buffer));
		   buffer[size_read] = '\0';
```

```
                    if(size_read == 0){
                        close(fd2);
                        fd2 = open(tuberia2, O_RDONLY | O_NONBLOCK);
                    }
                    else printf("Tuberia 2: %s", buffer);
                }
            }
            else{
                printf("Ningun dato nuevo.\n");
            }

        }

        close(fd1);
        close(fd2);
        return 0;
}
```

DESDE TERMINAL 1:

```
$ gcc e5.c -o e5
$ ./e5
Tuberia 2: hola
Tuberia 2: pruebaT2
Tuberia 1: que tal
Tuberia 1: pruebaT1
```

DESDE TERMINAL 2:

```
$ echo "hola" > tuberia2
$ echo "pruebaT2" > tuberia2
$ echo "que tal" > tuberia1
$ echo "pruebaT1" > tuberia1
```