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 con nombre

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

Ejercicio 1. Usar la orden `mkfifo` para crear una tubería (ej. `$HOME/tuberia`). 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`, `less`, `tail`).
En un terminal: mkfifo pipe cat pipe
En otro terminal: echo > pipe mensaje

Ejercicio 2. Escribir un programa que abra la tubería con el nombre anterior (`$HOME/tuberia`) 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 <sys/types.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
        if(argc != 2) {
                printf("Vuelve a ejecutar el programa especificando algo para
escribir\n");
                exit(EXIT_FAILURE);
        }
```

```
        char *home = getenv("HOME");
        char *path = "/tuberia";
        char *buf = malloc(sizeof(argv[1]));
        strcat(home, path); // home: root/tuberia
        buf = strcat(argv[1], "\n");

        // Creamos la tuberia con mkfifo(<nombre>, <modo>)
        // Modo solo escritura (permisos): 0222 p-w--w--w-
        if(mkfifo(home, 0222) == -1) perror("Error al crear la tuberia");
        int fd = open(home, O_WRONLY);
        if(fd == -1) perror("Error al abrir la tuberia");

        // Tener en cuenta los caracteres '\0' y "\n" en la escritura, NO USAR
sizeo$
        ssize_t escrito = write(fd, argv[1], strlen(buf));
        if(escrito == 0) printf("No se ha escrito nada\n");
        else if (escrito == -1) perror("Error al escribir en la tuberia");
        close(fd);
        return 0;
}
```

Multiplexación de canales de entrada/salida

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

Ejercicio 1. Crear otra tubería con nombre (ej. `tuberia2`). Escribir un programa que espere hasta que haya datos listos para leer en alguna de ellas. El programa debe escribir en la tubería desde la que se leyó y los datos leídos. Consideraciones:

- Para optimizar las operaciones de lectura usar un *buffer* (ej. de 256 bytes).
- Usar `read()` para leer de la tubería y gestionar adecuadamente la longitud de los caracteres 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()`.
- Cuando el escritor termina y cierra la tubería, `select()` considerará el descriptor siempre listo para lectura (para detectar el fin de fichero) y no se bloqueará. En este caso, hay que cerrar la tubería y volver a abrirla.

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/time.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
        char buf[256];
        //char bufw[256];
```

```
        int fd0, fd1;
        int rc;
        struct timeval timeout;
        timeout.tv_sec = 10;
        timeout.tv_usec = 0;
        fd_set fdsread, fdswrite;

        // Creamos la tuberia con mkfifo(<nombre>, <modo>)
        // Modo solo escritura (permisos): 0666 prw-rw-rw-
        // Modo solo escritura (permisos): 0666 prw-rw-rw-
        /*if(mkfifo(tmp, 0666) == -1) perror("Error al crear la tuberia");
        if(mkfifo(home, 0666) == -1) perror("Error al crear la tuberia");*/

        fd0 = open("tuberia", O_RDWR | O_NONBLOCK);
        fd1 = open("tuberia2", O_RDWR | O_NONBLOCK);
        if(fd0 == -1) perror("Error en el open fd0");
        if(fd1 == -1) perror("Error en el open fd1");

        //while(1) {
                FD_ZERO(&fdsread);
                FD_ZERO(&fdswrite);
                FD_SET(fd0, &fdsread);
                FD_SET(fd1, &fdsread);
                FD_SET(fd0, &fdswrite);
                FD_SET(fd1, &fdswrite);

                rc = select(fd1+1, &fdsread, &fdswrite, NULL, &timeout);
                if(rc == 0) {
                        //printf("Han pasado 10 segundos sin recibir nada\n");
                } else if(rc == -1) {
                        perror("Fallo en el select()");
                } else {
                        if(FD_ISSET(fd0, &fdsread)) {
                                ssize_t size = read(fd0, buf, sizeof(buf));
                                if(size == -1) perror("Error en read()");
                                else if (size == 0) { // Cerramos y volvemos a
abrir el descriptor
                                        close(fd0);
                                        fd0 = open("tuberia", O_RDWR |
O_NONBLOCK);
                                } else {
                                        buf[size] = '\0';
                                        printf("Se ha recibido algo por la
tuberia 0: %s\n", buf);
                                        ssize_t size2 = write(fd0, buf,
sizeof(buf));
                                        buf[size2] = '\0';
                                        printf("Se ha escrito %s\n", buf);
```

```
                                }
                        }
                        if(FD_ISSET(fd1, &fdsread)) {
                                ssize_t size = read(fd1, buf, sizeof(buf));
                                if(size == -1) perror("Error en read()");
                                else if (size == 0) { // Cerramos y volvemos a
abrir el descriptor
                                        close(fd1);
                                        fd1 = open("tuberia2", O_RDWR |
O_NONBLOCK);
                                } else {
                                        buf[size] = '\0';
                                        printf("Se ha recibido algo por la
tuberia 1: %s\n", buf);
                                        ssize_t size2 = write(fd1, buf,
sizeof(buf));
                                        buf[size2] = '\0';
                                        printf("Se ha escrito %s\n", buf);
                                }
                        }
                }
        //}
        close(fd0);
        close(fd1);
        return 0;
}
```

Tuberías sin nombre

Las tuberías sin nombre son entidades gestionadas directamente por el núcleo del sistema y son un mecanismo eficiente para procesos relacionados (padre-hijo). La forma de comunicar los identificadores de la tubería es por herencia (en la llamada `fork`). En este caso no hay una ruta bien conocida en el sistema de ficheros.

Ejercicio 1. Comunicación por tuberías. 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 abrirá una tubería sin nombre y creará un hijo:

- El programa padre ejecutará `comando1 argumento1` y redireccionará la salida estándar al extremo de escritura de la tubería.
- El hijo, ejecutará `comando2 argumento2`, en este caso la entrada estándar deberá duplicarse con el extremo de lectura de la tubería.
- 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 <sys/types.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
```

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

// Probar el programa con ./shell echo holaa | wc -c
int main(int argc, char *argv[]) {
        if(argc < 3) {
                printf("Uso: comando1 argumento1 | comando2 argumento2\n");
                exit(EXIT_FAILURE);
        }
        int descf[2]; //descf[0] -> lectura, descf[1] -> escritura
        int res1, res2;
        char *cmd1[] = {argv[1], argv[2], NULL};
        char *cmd2[] = {argv[4], argv[5], NULL};

        // Creamos la tuberia sin nombre
        pipe(descf);

        // Si no ha habido errores (-1) fork() retorna dos veces, una con un
pid = 0 (ejecución proceso hijo) y otra con un pid > 0 (ejecución proceso
padre)
        pid_t pid = fork();
        if(pid == -1) {
                perror("Error en el fork()");
       } else if(pid > 0) { // Ejecución del padre:
                close(descf[1]); // Cerramos el extremo de escritura del pipe
                dup2(descf[1], 1); // Redirecciona salida estandar al extremo
de escritura del pipe
                close(descf[0]);
                res1 = execvp(cmd1[0], cmd1); //Ejecuta comando1 argumento 1
                if(res1 == -1) perror("Fallo en execvp1()");
        } else if(pid == 0) { // Ejecución del hijo
                close(descf[0]); // Cerramos el extremo de lectura del pipe
                dup2(descf[0], 0); // Duplica la entrada estandar con el
extremo de lectura del pipe
                close(descf[1]);
                res2 = execvp(cmd2[0], cmd2); // Ejecuta comando2 argumento 2
                if(res2 == -1) perror("Fallo en execvp2()");
        }
        close(descf[1]);
        close(descf[0]);
        return 0;
}
```

IMPORTANTE: Funciona casi completamente bien, da un fallo en execvp2(), no sé por qué.

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 escribiendo 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`, cuando haya leído y procesado el mensaje (escribiéndolo por la salida estándar y esperando 1 segundo) enviará el carácter `'l'` al proceso padre para indicar que está listo escribiendo en la tubería `h_p`. Después de 10 mensajes enviará `'q'` para indicar al padre que finalice.

```
#include <sys/types.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/select.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
        int p_h[2]; //p_h[0] -> lectura, p_h[1] -> escritura
        int h_p[2];
        // Creamos las tuberias sin nombre
        if(pipe(p_h) == -1) perror("Fallo en pipe()");
        if(pipe(h_p) == -1) perror("Fallo en pipe()");
        // Si no ha habido errores (-1) fork() retorna dos veces, una con un
pid = 0 (ejecución proceso hijo) y otra con un pid > 0 (ejecución proceso
padre)
        pid_t pid = fork();
        if(pid == -1) {
                perror("Error en el fork()");
       } else if(pid > 0) { // Ejecución del padre:
                // Cerramos los extremos opuestos a los que vamos a utilizar
                close(p_h[0]);
                close(h_p[1]);
                char parentbuf[256];
                char childmsg[1] = {'l'};
                while(childmsg[0] != 'q') {
                        printf("Padre, Mensaje a enviar:\n");
                        // Lee de la entrada estandar
                        // Al ser la primera vez que leemos el size es 256 y
hay que poner el '\0'
                        ssize_t size = read(0, parentbuf, 256);
                        if(size == -1) perror("Padre, error leyendo");
                        parentbuf[size] = '\0';
                        // Envía el mensaje al hijo escribiendo en p_h
                        size = write(p_h[1], parentbuf, size+1);
                        if(size == -1) perror("Padre, error escribiendo");
```

```
                        // Permanece bloqueado esperando la confirmación por
parte del hijo en la tuberia h_p
                        while(childmsg[0] != 'l' && childmsg[0] != 'q') {
                                ssize_t size = read(h_p[0], childmsg, 1);
                                if(size == -1) perror("Padre, error leyendo");
                        }
                }
                // Cerramos los otros extremos de los pipes
                close(p_h[1]);
                close(h_p[0]);
        } else if(pid == 0) { // Ejecución del hijo
                // Cerramos los extremos opuestos a los que vamos a utilizar
                close(p_h[1]);
                close(h_p[0]);
                char bufmsgs[257];
                char parentmsg[1] = {'l'};
                int i;
                for(i = 0; i < 10; i++) {
                        ssize_t size = read(p_h[0], bufmsgs, 256); // Lee de
la tuberia p_h
                        if(size == -1) perror("Hijo: error leyendo");
                        bufmsgs[size] = '\0';
                        // Escribe por la salida estandar y espera un segundo
                        printf("Hijo, Mensaje recibido: %s:", bufmsgs);
                        sleep(1);
                        // Tras 10 mensajes envía 'q' para indicar que el
padre finalice
                        if(i == 9) parentmsg[0] = 'q';
                        // Si i < 9 escribimos 'l' en la tuberia para
notificar que el hijo esta listo
                        size = write(h_p[1], parentmsg, 1);
                        if(size == -1) perror("Hijo: error escribiendo");
                }
                // Cerramos los otros extremos de los pipes
                close(p_h[0]);
                close(h_p[1]);
        }
        return 0;
}
```