

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
Tuberías sin nombre
Tuberías con nombre
Multiplexación síncrona de entrada/salida

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 <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

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

    if (argc < 4){
        printf("\n\n\tError! Usage [cmd1 arg1 | cm2 arg2].\n\n");
        return 1;
    }
}
```

```

    }

    int tuberia_ph[2];
    pipe(tuberia_ph);
    char buffer[256];

    int pid = fork();

    switch(pid){
    case -1: {
        perror("\n\tERROR: Couldn't create child process.\n");
        close(tuberia_ph[0]); close(tuberia_ph[1]);
        return 1;
        break;
    }
    case 0: { //HIJO
        close(tuberia_ph[1]);
        dup2(tuberia_ph[0], 0);
        strcpy(buffer, argv[3]);
        strcat(buffer, " ");
        strcat(buffer, argv[4]);
        system(buffer);
        close(tuberia_ph[0]);
        return 0;
        break;
    }
    default:{
        close(tuberia_ph[0]);
        dup2(tuberia_ph[1], 1);
        strcpy(buffer, argv[1]);
        strcat(buffer, " ");
        strcat(buffer, argv[2]);
        system(buffer);
        close(tuberia_ph[1]);
        return 0;
        break;
    }
    }
    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 <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

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

    bool fin = 0;
    int m = 0;
    int tuberia_ph[2];
    int tuberia_hp[2];
    pipe(tuberia_ph);
    pipe(tuberia_hp);
    char buffer[256];

    int pid = fork();

    switch(pid){
        case -1: {
            perror("\n\tERROR: Couldn't create child process.\n");
            close(tuberia_hp[0]); close(tuberia_hp[1]);
            close(tuberia_ph[0]); close(tuberia_ph[1]);
            return 1;
            break;
        }
        case 0: { //HIJO
            close(tuberia_ph[1]);
            close(tuberia_hp[0]);
            while(!fin){
                int rc = read(tuberia_ph[0], buffer, 256);
                buffer[rc] = '\0';
                printf("\n\tMsg recieved: %s.\n", buffer);
                if(++m == 10){
                    write(tuberia_hp[1], "Q", 1);
                    fin = 1;
                }
                else {
                    write(tuberia_hp[1], "L", 1);
                }
            }
            close(tuberia_ph[0]);
            close(tuberia_hp[1]);
            return 0;
            break;
        }
        default:{
            close(tuberia_ph[0]);
            close(tuberia_hp[1]);
            while(!fin){
                printf("\n\t? ");
                scanf("%s", buffer);
                write(tuberia_ph[1], buffer, strlen(buffer)+1);
            }
        }
    }
}

```

```

        read(tuberia_hp[0], buffer, 1);
        if(buffer[0] == 'Q') fin = 1;
    }
    close(tuberia_ph[1]);
    close(tuberia_hp[0]);
    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 pipe cat pipe Hola	echo "Hola" > pipe
---	------------------------------

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 <sys/stat.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char** argv){
    if(argc < 2){
        printf("\n\tERROR: Usage [\"Mensaje\"]\n");
        return 1;
    }

    int fd = open("pipe", O_WRONLY);
    if(fd == -1){
        perror("\n\tERROR: Couldn't open [pipe]\n");
        perror("\tDid you forget to [mkfifo pipe]?\n");
        return 1;
    }
    printf("\n\tWriting [%s] in [pipe]\n\n", argv[1]);
}

```

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

```
mkfifo pipe
mkfifo pipe2
```

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

int main(int argc, char** argv){
    int fd1 = open("pipe", O_RDONLY | O_NONBLOCK);
    int fd2 = open("pipe2", O_RDONLY | O_NONBLOCK);

    if(fd1 == -1 || fd2 == -1){
        perror("\n\tERROR: Couldn't open [pipe || pipe2].\n");
        printf("\tDid you forget to [mkfifo pipe2]?\n");
        return 1;
    }

    int fin = 0, m = 0, sel;

    fd_set rfd;

    FD_ZERO(&rfd);
    FD_SET(fd1, &rfd);
```

```

    FD_SET(fd2, &rfd);

    struct timeval timeout;
    timeout.tv_sec = 20;
    timeout.tv_usec = 0;

    char buffer[256];

    int mayor = (fd1 > fd2)? fd1 + 1: fd2 + 1;
    int fdact;

    while(!fin){
        sel = select(mayor, &rfd, NULL, NULL, &timeout);
        switch(sel){
            case -1:
                perror("\n\tERROR: Couldn't select [pipe || pipe2].\n");
                return 1;
                break;
            default:
                if(FD_ISSET(fd1,&rfd)){
                    int b = read(fd1, buffer, 256);
                    fin = (b)? 1:0;
                    buffer[b] = '\0';
                    printf("\tLectura de pipe [%s].\n", buffer);
                    if(fin) {
                        close(fd1);
                        int fd1 = open("pipe", O_RDONLY | O_NONBLOCK);
                        int fd2 = open("pipe2", O_RDONLY | O_NONBLOCK);
                    }
                }
                if(FD_ISSET(fd2,&rfd)){
                    int b = read(fd2, buffer, 256);
                    fin = (b)? 1:0;
                    buffer[b] = '\0';
                    printf("\tLectura de pipe2 [%s].\n", buffer);
                    if(fin){
                        close(fd2);
                        int fd2 = open("pipe2", O_RDONLY | O_NONBLOCK);
                    }
                }
                break;
        }

        }
        printf("\n\tFin\n");

        close(fd1);
        close(fd2);

        return 0;
    }

```