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.

```c
#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

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

    if (argc < 5) {

        printf("Numero insuficiente de argumentos \n");

        exit(1);
```

```
}
//Creamos tubería sin nombre
int fd[2];
if (pipe(fd) == -1) {
    perror("pipe");
    exit(1);
}
//Creamos un hijo
int pid = fork();
if (pid == -1) {
    perror("fork");
    exit(1);
}
//el proceso hijo redirecciona la salida estandar al extremo
//de lectura de la tuberia y ejecuta com2 arg2
else if (pid == 0) {


     if (close(fd[1]) == -1) {
        perror("close fd1 hijo");
        exit(1);
    }
    //redireccionamos entrada estandar (0) a extremo de lectura (fd[0])
    if (dup2(fd[0], 0) == -1) {
        perror("dup2 hijo");
        exit(1);
    }


    if (close(fd[0]) == -1) {
        perror("close fd0 hijo");
```

```
        exit(1);
    }



    //ejecutamos com2 arg2
    execlp(argv[3], argv[3], argv[4], (char* ) 0);
}
//el proceso padre redirecciona la entrada estandar al extremo
//de escritura de la tuberia y ejectua com1 arg1
else {


     //cerramos descriptores
    if (close(fd[0]) == -1) {
        perror("close fd0 padre");
        exit(1);
    }


    //redireccionamos salida estandar(1) a extremo de escritura(fd[1])
    if (dup2(fd[1], 1) == -1) {
        perror("dup2 padre");
        exit(1);
    }


    if (close(fd[1]) == -1) {
        perror("close fd1 padre");
        exit(1);
    }
    //ejecutamos com1 arg1
    execlp(argv[1], argv[1], argv[2], (char* ) 0);
```

```
    }


    return 0;

}
```

Comando: *./a.out echo 12345 wc -c*

Salida: *6*

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

int main(int argc, char** argv) {
    //creamos las dos tuberias
    int p_h[2];
    int h_p[2];
    if (pipe(p_h) == -1) {
        perror("pipe p_h");
        exit(1);
    }
    if (pipe(h_p) == -1) {
        perror("pipe h_p");
        exit(1);
    }

    int MAX = 256;
    char msg[MAX];

    int pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(1);
    }
```

```
//el hijo
else if (pid == 0) {
    //no se escribe en la tuberia p_h
    if(close(p_h[1]) == -1) {
        perror("close p_h[1]");
        exit(1);
    }
    //no se lee en la tuberia h_p
    if (close(h_p[0]) == -1) {
        perror("close h_p[0]");
        exit(1);
    }
    int received = 0;
    while (received < 10) {
        //lee la tuberia p_h
        if (read(p_h[0], msg, MAX) == -1) {
            perror("read p_h[0]");
            exit(1);
        }
        //escribe el mensaje por salida estandar
        printf("%i: %s \n", received, msg);
        received = received + 1;
        //espera un segundo
        sleep(1);
        //indica que esta listo al proceso padre via pipe h_p
        if (received < 10) {
            strcpy(msg,"1");
        }
        //tras 10 mensajes, indica al padre que finalice
        else {
            strcpy(msg,"q");
        }

        //via pipe h_p
        if (write(h_p[1], msg, strlen(msg) + 1) == -1) {
            perror("wirte h_p[1]");
        }

    }
    //cerramos lectura en la tuberia p_h
        if(close(p_h[0]) == -1) {
            perror("close p_h[0]");
            exit(1);
        }
        //cerramos escritura en la tuberia h_p
        if (close(h_p[1]) == -1) {
            perror("close h_p[1]");
            exit(1);
        }
        exit(0);
}
```

```
    //padre
    else {
        //no se lee en la tuberia p_h
        if(close(p_h[0]) == -1) {
            perror("close p_h[0]");
            exit(1);
        }
        //no se escribe en la tuberia h_p
        if (close(h_p[1]) == -1) {
            perror("close h_p[1]");
            exit(1);
        }

        int fin = 1;
        while (fin) {
            //lee la entrada estandar
            printf("Mensaje: ");
            if (fgets(msg, sizeof(msg), stdin) == NULL) {
                perror("fgets stdin");
                exit(1);
            }
            //envia al hijo el mensaje via pipe p_h
            if (write(p_h[1], msg, strlen(msg) + 1) == -1) {
                perror("write p_h[1]");
                exit(1);
            }
            //bloqueado esperando la confirmacion del hijo via pipe h_p
            int ready_stop = read(h_p[0], msg, MAX);
            if (ready_stop == -1) {
                perror("read h_p[0]");
                exit(1);
            }
            while (strcmp(msg,"l")!= 0 && strcmp(msg,"l")!= 0) {
                ready_stop = read(h_p[0], msg, MAX);
                if (ready_stop == -1) {
                    perror("read h_p[0]");
                    exit(1);
                }
            }
            //quit
            if (msg == "q"){
                fin = 0;
            }
        }
        //cerramos en la tuberia p_h
        if(close(p_h[1]) == -1) {
            perror("close p_h[1]");
            exit(1);
        }
        //cerramos en la tuberia h_p
        if (close(h_p[0]) == -1) {
```

```
            perror("close h_p[0]");
            exit(1);
        }

    }
    return 0;
}
```

Salida:

Mensaje: a
0: a

Mensaje: b
1: b

Mensaje: c
2: c

Mensaje: pepito grillo
3: pepito grillo

Mensaje: ejemplo
4: ejemplo

Mensaje: hola que tal
5: hola que tal

Mensaje: como estas
6: como estas

Mensaje: yo muy bien
7: yo muy bien

Mensaje: pepe
8: pepe

Mensaje: adios
9: adios

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`...).

Comando: *mkfifo tuberia*

Comando: *stat tuberia*

Salida:

Fichero: tuberia
Tamaño: 0 Bloques: 0 Bloque E/S: 4096 `fifo'
Dispositivo: 801h/2049d Nodo-i: 555755 Enlaces: 1
Acceso: (0664/prw-rw-r--) Uid: (1000/usuarioso) Gid: (1001/usuarioso)
Acceso: 2022-12-05 16:41:05.755925930 +0100
Modificación: 2022-12-05 16:41:05.755925930 +0100
Cambio: 2022-12-05 16:41:05.755925930 +0100
Creación: -

[Terminal 1]

Comando: *echo hola mundo > tuberia*

[Terminal 2]

Comando: *cat tuberia*
Salida: *hola mundo*

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.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char** argv) {
    if (argc < 2) {
        printf("Numero de argumentos insuficientes \n");
        exit(1);
    }
    int fd = open("./tuberia", O_WRONLY);
    if (fd == -1) {
        perror("open tuberia");
        exit(1);
    }
    if (write(fd, argv[1], strlen(argv[1])) == -1) {
        perror("write");
        exit(1);
    }
    close(fd);
```

```
    return 0;
}
```

[Terminal 1]
Comando: *./a.out hola mundo*
[Terminal 2]
Comando: *cat tuberia*
Salida: *hola*

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 <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/select.h>
#include <fcntl.h>
int main(int argc, char** argv) {
    //creamos otra tuberia
    if (mkfifo("./tuberia2", S_IFIFO)== -1) {
        perror("mkfifo");
        exit(1);
    }

    char buff[256];

    //abrimos los ficheros
    int fd1 = open("./tuberia", O_RDONLY | O_NONBLOCK);
    if (fd1 == -1) {
        perror("open tuberia");
        exit(1);
    }
    int fd2 = open("./tuberia2", O_RDONLY | O_NONBLOCK);
    if (fd2 == -1) {
```

```
        perror("open tuberia2");
        exit(1);
    }
    while (1) {
        fd_set fds; //conjunto de descriptores de fichero
        FD_ZERO(&fds); //inicializa a conjunto vacio
        FD_SET(fd1, &fds); //añade descriptor de pipe 1
        FD_SET(fd2, &fds); //añade descriptor pipe 2
        int max = (fd1 > fd2) ? fd1 : fd2;
        //seleccionamos descriptores de fichero preparados
        int ready = select(max + 1, &fds, NULL, NULL, NULL);
        //fd1 esta listo
        if (FD_ISSET(fd1, &fds)) {

            printf("Se ha escrito en tuberia 1 \n");
            if (read(fd1, buff, 256) == -1) {
                perror("read fd1");
                exit(1);
            }
            if (close(fd1) == -1) {
                perror("close fd1");
                exit(1);
            }
            fd1 = open("./tuberia", O_RDONLY | O_NONBLOCK);
            if (fd1 == -1) {
                perror("open fd1");
                exit(1);
            }
        }
        else { printf("%s \n", buff); }
        //fd2 esta listo
        if (FD_ISSET(fd2, &fds)) {
            printf("Se ha escrito en tuberia 2 \n");
            if (read(fd2, buff, 256) == -1) {
                perror("read fd2");
                exit(1);
            }
            if (close(fd2) == -1) {
                perror("close fd2");
                exit(1);
            }
            fd2 = open("./tuberia2", O_RDONLY | O_NONBLOCK);
            if (fd2 == -1) {
                perror("open fd2");
                exit(1);
            }
        }
        else { printf("%s \n", buff); }
    }
    return 0;
}
```

[Terminal 1]

Comando: *./a.out*

[Terminal 2]

Comando: *echo hola > tuberia*

[Terminal 1]

Salida: *Se ha escrito en tuberia 1*
hola