

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 <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[]) {
    if(argc < 3) {
        printf("Uso: comando1 argumento1 comando2 argumento2\n");
```

```

        exit(EXIT_FAILURE);
    }
    int desc[2];
    int res1, res2;
    char *cmd1[] = {argv[1], argv[2], NULL};
    char *cmd2[] = {argv[3], argv[4], NULL};
    pipe(desc);
    pid_t pid = fork();
    if(pid == -1) {
        perror("Error en el fork()");
    } else if(pid > 0) { // Ejecución del padre:
        close(desc[1]); // Cerramos el extremo de escritura del pipe
        dup2(desc[1], 1);
        close(desc[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(desc[0]); // Cerramos el extremo de lectura del pipe
        dup2(desc[0], 0);
        close(desc[1]);
        res2 = execvp(cmd2[0], cmd2); // Ejecuta comando2 argumento 2
        if(res2 == -1) perror("Fallo en execvp2()");
    }
    close(desc[1]);
    close(desc[0]);
    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 <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 tuberías sin nombre
if(pipe(p_h) == -1) perror("Fallo en pipe()");
if(pipe(h_p) == -1) perror("Fallo en pipe()");
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");
        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");
        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);
        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);
        if(i == 9) parentmsg[0] = 'q';
        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;}

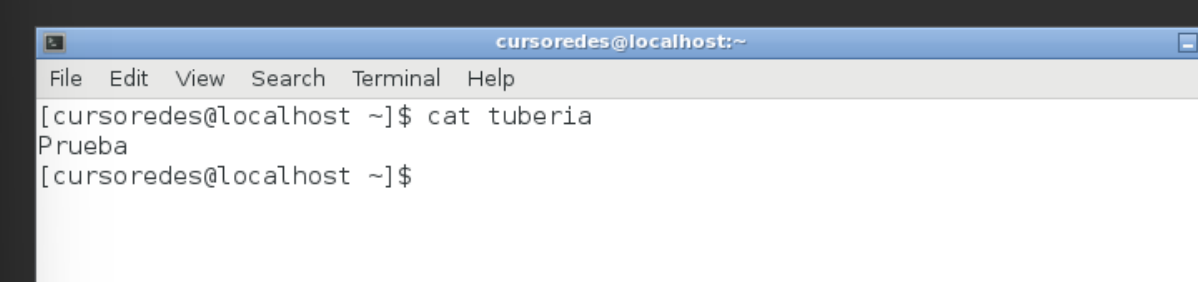
```

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`...).

```
[cursoredes@localhost ~]$ stat tuberia
File: 'tuberia'
Size: 0          Blocks: 0          IO Block: 4096   fifo
Device: fd00h/64768d Inode: 53012648   Links: 1
Access: (0664/prw-rw-r--)  Uid: ( 1000/cursoredes)   Gid: ( 1000/cursoredes)
Access: 2022-12-05 09:51:23.132881326 +0100
Modify: 2022-12-05 09:51:23.132881326 +0100
Change: 2022-12-05 09:51:23.132881326 +0100
Birth: -
[cursoredes@localhost ~]$ echo "Prueba" > tuberia
[cursoredes@localhost ~]$
```



En una terminal:

```
$man mkfifo
$mkfifo tuberia
$ls -l
$stat tuberia
$echo "Test" > tuberia
```

En otra terminal:

```
$cat tuberia
```

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 <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[]) {
    if(argc != 2) {
        printf("Vuelve a ejecutar el programa\n");
        exit(EXIT_FAILURE);
    }
    char *casa = getenv("HOME");
    char *path = "/tuberia";
    char *buf = malloc(sizeof(argv[1]));
    strcat(casa, path); // home: root/tuberia
    buf = strcat(argv[1], "\n");
    if(mkfifo(casa, 0222) == -1) perror("Error al crear la tuberia");
    int fd = open(casa, O_WRONLY);
    if(fd == -1) perror("Error al abrir la tuberia");
    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 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 <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];
    fd_set tub_read, tub_write;
    int tuberia_1, tuberia_2;

```

```

int rc;
struct timeval timeout;

timeout.tv_sec = 10;
timeout.tv_usec = 0;

tuberia_1 = open("tuberia", O_RDWR | O_NONBLOCK);
tuberia_2 = open("tuberia2", O_RDWR | O_NONBLOCK);
if(tuberia_1 == -1) perror("Error en el open");
if(tuberia_2 == -1) perror("Error en el open");
FD_ZERO(&tub_read);
FD_ZERO(&tub_write);
FD_SET(tuberia_1, &tub_read);
FD_SET(tuberia_2, &tub_read);
FD_SET(tuberia_1, &tub_write);
FD_SET(tuberia_2, &tub_write);
rc = select(tuberia_2+1, &tub_read, &tub_write, NULL, &timeout);
if(rc == 0) {} else if(rc == -1) {
    perror("Fallo en el select()");
} else {
    if(FD_ISSET(tuberia_1, &tub_read)) {
        ssize_t size = read(tuberia_1, buf, sizeof(buf));
        if(size == -1) perror("Error en read()");
        else if (size == 0) {
            close(tuberia_1);
            tuberia_1 = 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(tuberia_1, buf, sizeof(buf));
            buf[size2] = '\0';
            printf("Se ha escrito %s\n", buf);
        }
    }

    if(FD_ISSET(tuberia_2, &tub_read)) {
        ssize_t size = read(tuberia_2, buf, sizeof(buf));
        if(size == -1) perror("Error en read()");
        else if (size == 0) {
            close(tuberia_2);
            tuberia_2 = 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(tuberia_2, buf, sizeof(buf));
            buf[size2] = '\0';
            printf("Se ha escrito %s\n", buf);
        }
    }
}

close(tuberia_1);
close(tuberia_2);
return 0;
}

```