

## 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>

int main(int argc, char** argv){
    if(argc < 5){
        printf("Faltan argumentos.\n");
        exit(1);
    }

    int pipefd[2];
    if(pipe(pipefd)==-1){
```

```

    perror("Error en pipe.");
    exit(1);
}
int pid = fork();
if(pid == -1){
    perror("Error en fork.");
    exit(1);
}
else if(pid == 0){
    if(dup2(pipefd[0],0)==-1){
        perror("Error en dup2.");
        exit(1);
    }
    if(close(pipefd[0])==-1){
        perror("Error en close.");
        exit(1);
    }
    if(close(pipefd[1])==-1){
        perror("Error en close.");
        exit(1);
    }
    execlp(argv[3], argv[3], argv[4], (char*) 0);
} else {
    if(dup2(pipefd[1],1)==-1){
        perror("Error en dup2.");
        exit(1);
    }
    if(close(pipefd[0])==-1){
        perror("Error en close.");
        exit(1);
    }
    if(close(pipefd[1])==-1){
        perror("Error en close.");
        exit(1);
    }
    execlp(argv[1], argv[1], argv[2], (char*) 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 <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){
    int p_h[2], h_p[2];
    int BUF_MAX = 256;
    char buf[BUF_MAX];
    if(pipe(p_h)==-1){
        perror("Error en pipe.");
        exit(1);
    }
    if(pipe(h_p)==-1){
        perror("Error en pipe.");
        exit(1);
    }
    int pid = fork();
    if(pid == -1){
        perror("Error en fork.");
        exit(1);
    }else if(pid == 0){
        if(close(p_h[1])==-1){
            perror("Error en close.");
            exit(1);
        }
        if(close(h_p[0])==-1){
            perror("Error en close.");
            exit(1);
        }
        int i = 0;
        while(i<10){
            if(read(p_h[0],buf, BUF_MAX)==-1){
                perror("Error en read p_h.");
                exit(1);
            }
            printf("Mensaje recibido: %s", buf);
            i++;
            if(i<10){
                strcpy(buf,"l");
                sleep(1);
            }
            else{
                strcpy(buf, "q");
            }
            if(write(h_p[1],buf,strlen(buf)+1)==-1){
                perror("Error en write.");
                exit(1);
            }
        }

        if(close(p_h[0])==-1){
            perror("Error en close.");
        }
    }
}

```

```

        exit(1);
    }
    if(close(h_p[1])==-1){
        perror("Error en close.");
        exit(1);
    }
}
else{
    if(close(p_h[0])==-1){
        perror("Error en close.");
        exit(1);
    }
    if(close(h_p[1])==-1){
        perror("Error en close.");
        exit(1);
    }

    int q=0;
    while(q==0){
        if(fgets(buf, sizeof(buf), stdin)==NULL){
            perror("Error en read 0.");
            exit(1);
        }

        if(write(p_h[1], buf, strlen(buf)+1)==-1){
            perror("Error en write.");
            exit(1);
        }

        if(read(h_p[0], buf, BUF_MAX)==-1){
            perror("Error en read h_p.");
            exit(1);
        }
        if(strcmp(buf,"q")==0){
            q = 1;
        }

    }
    if(close(p_h[1])==-1){
        perror("Error en close.");
        exit(1);
    }
    if(close(h_p[0])==-1){
        perror("Error en close.");
        exit(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...`).

```
mkfifo tuberia
ls -l
total 56
-rwxrwxr-x 1 cursoredes cursoredes 8792 Dec  5 11:25 ej1
-rw-r--r-- 1 root    root    1175 Dec  5 11:25 ej1.c
-rwxrwxr-x 1 cursoredes cursoredes 13144 Dec  5 19:37 ej2
-rwxrwxr-x 1 cursoredes cursoredes 13144 Dec  5 19:49 ej2b
-rw-r--r-- 1 cursoredes cursoredes 2387 Dec  5 19:49 ej2b.c
-rw-r--r-- 1 root    root    2483 Dec  5 19:37 ej2.c
prw-rw-r-- 1 cursoredes cursoredes    0 Dec  5 20:06 tuberia

stat tuberia
  File: 'tuberia'
  Size: 0          Blocks: 0      IO Block: 4096   fifo
Device: fd00h/64768d Inode: 51269476 Links: 1
Access: (0664/prw-rw-r--)  Uid: ( 1000/cursoredes)  Gid: ( 1000/cursoredes)
Access: 2022-12-05 20:06:09.492350117 +0100
Modify: 2022-12-05 20:06:09.492350117 +0100
Change: 2022-12-05 20:06:09.492350117 +0100
Birth: -

Terminal 1
echo jaja > tuberia

Terminal 2
cat tuberia
Salida: jaja
```

**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 <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("Faltan argumentos.\n");
        exit(1);
    }
}
```

```

}

int fd = open("./tuberia", O_WRONLY);
if(fd==-1){
    perror("Error en open.");
    exit(1);
}
if(write(fd, argv[1], strlen(argv[1]))==-1){
    perror("Error en open.");
    exit(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 <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/select.h>
#include <sys/time.h>

int main(int argc, char** argv){
    if(mkfifo("./tuberia2",0666)==-1){
        perror("Error en mkfifo.");
        exit(1);
    }
    int fd1 = open("./tuberia", O_NONBLOCK|O_RDONLY);
    int fd2 = open("./tuberia2", O_NONBLOCK|O_RDONLY);
}

```

```

if(fd1==-1 || fd2==-1){
    perror("Error en open.");
    exit(1);
}
fd_set set;
while(1){
    FD_ZERO(&set);
    FD_SET(fd1, &set);
    FD_SET(fd2, &set);
    int max = fd1;
    if(fd2>fd1){
        max = fd2;
    }
    max++;
    int sl = select(max, &set,NULL,NULL,NULL);

    if(FD_ISSET(fd1, &set)){
        char buffer[256];
        ssize_t sizeread = read(fd1, buffer, 256);
        if(sizeread==0){
            close(fd1);
            fd1 = open("./tuberia", O_NONBLOCK|O_RDONLY);
        }
        else{
            printf("Escrito en tuberia 1:\n");
            write(1,buffer,sizeread);
        }
    }
    if(FD_ISSET(fd2, &set)){
        char buffer[256];
        ssize_t sizeread = read(fd2, buffer, 256);
        if(sizeread==0){
            close(fd2);
            fd2 = open("./tuberia2", O_NONBLOCK|O_RDONLY);
        }
        else{
            printf("Escrito en tuberia 2:\n");
            write(1,buffer,sizeread);
        }
    }
}
return 0;
}

```