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 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.

```
fork - create a child process
pipe, pipe2 - create pipe
dup, dup2, dup3 - duplicate a file descriptor
close - close a file descriptor
execl, execlp, execle, execv, execvp, execvpe - execute a file

#include <unistd.h>

pid_t fork(void);
int pipe(int pipefd[2]);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int close(int fd);
```

```
extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,  ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);


#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[]) {
   if (argc != 5) {
                        printf("Incorrect number of arguments\n");
                        return 1;
           }

   int pipefd[2];
   pipe(pipefd);

   pid_t pid = fork();
   if (pid == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
   }
   else if (pid == 0) {
      close(pipefd[1]);
      dup2(pipefd[0], 0);
      close(pipefd[0]);

      execlp(argv[3], argv[3], argv[4], NULL);
   }
   else {
      close(pipefd[0]);
      dup2(pipefd[1], 1);
      close(pipefd[1]);

      execlp(argv[1], argv[1], argv[2], NULL);
   }
   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.

```
read - read from a file descriptor
write - write to a file descriptor

#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);


#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(int argc, char *argv[]) {
   int p_h[2];
   pipe(p_h);

   int h_p[2];
   pipe(h_p);

   pid_t pid = fork();
   if (pid == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
   }
   else if (pid == 0) {
      close(p_h[1]);
      close(h_p[0]);

      char buf[512];
      char response[1] = {'l'};

      int i;
      for (i = 0; i < 10; i++) {
         ssize_t size = read(p_h[0], buf, 511);
         printf("Received '%s' from my parent\n", buf);
         memset(buf, 0, sizeof(buf));


         sleep(1);

         if (i == 9)
            response[0] = 'q';

         write(h_p[1], response, 1);
      }

      close(p_h[0]);
      close(h_p[1]);
   }
   else {
      close(p_h[0]);
      close(h_p[1]);

      char buf[512];
      char response[1];
```

```
    while(response[0] != 'q') {
      printf("Write a message for my child:\n");
      ssize_t size = read(0, buf, 512);

      write(p_h[1], buf, size-1);
      memset(buf, 0, sizeof(buf));

      read(h_p[0], response, 1);
    }

    wait(NULL);

    close(p_h[1]);
    close(h_p[0]);
  }
  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 - make FIFOs (named pipes)

mkfifo [OPTION]... NAME…
-m, --mode=MODE: set file permission bits to MODE, not a=rw - umask

[cursoredes@localhost ~]$ stat tuberia1
  File: ‘tuberia1’
  Size: 0               Blocks: 0          IO Block: 4096   fifo
Device: fd00h/64768d    Inode: 53012990    Links: 1
Access: (0664/prw-rw-r--)  Uid: ( 1000/cursoredes)   Gid: ( 1000/cursoredes)
Access: 2021-11-29 11:11:39.639418789 +0100
Modify: 2021-11-29 11:11:39.639418789 +0100
Change: 2021-11-29 11:11:39.639418789 +0100
 Birth: -
[cursoredes@localhost ~]$ ls -l
total 0
drwxr-xr-x 2 cursoredes cursoredes 116 Sep  9  2018 Desktop
drwxr-xr-x 2 cursoredes cursoredes   6 Sep  9  2018 Documents
drwxr-xr-x 2 cursoredes cursoredes   6 Sep  9  2018 Downloads
drwxr-xr-x 2 cursoredes cursoredes   6 Sep  9  2018 Music
drwxr-xr-x 2 cursoredes cursoredes 147 Sep 22  2018 Pictures
drwxr-xr-x 2 cursoredes cursoredes   6 Sep  9  2018 Public
drwxr-xr-x 2 cursoredes cursoredes   6 Sep  9  2018 Templates
prw-rw-r-- 1 cursoredes cursoredes   0 Nov 29 11:11 tuberia1
drwxr-xr-x 2 cursoredes cursoredes   6 Sep  9  2018 Videos
[cursoredes@localhost ~]$ echo Hola > tuberia1
```

```
[cursoredes@localhost ~]$ cat tuberia1
Hola
```

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.

```
mkfifo - make a FIFO special file (a named pipe)

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

int mkfifo(const char *pathname, mode_t mode);


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

int main(int argc, char *argv[]) {
   if (argc != 2) {
      printf("Incorrect number of arguments\n");
            return 1;
   }
   char *path = "/home/cursoredes/tuberia1";
   int v = mkfifo(path, 0777);
   if (v == 0) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
   }

   int fd = open(path, O_WRONLY);
   v = write(fd, argv[1], strlen(argv[1]));
   if (v == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 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.

```
select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - synchronous I/O multiplexing

#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);


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

int main(int argc, char *argv[]) {
   char *path1 = "/home/cursoredes/tuberia1";
   char *path2 = "/home/cursoredes/tuberia2";
   char buf[256];
   int v = mkfifo(path1, 0777);
   if (v == -1) {
      printf("Error: %d %s\n", errno, strerror(errno));
      return 1;
   }

   v = mkfifo(path2, 0777);
   if (v == -1) {
      printf("Error: %d %s\n", errno, strerror(errno));
      return 1;
   }

   int fd1 = open(path1, O_RDONLY | O_NONBLOCK);
   int fd2 = open(path2, O_RDONLY | O_NONBLOCK);

   fd_set readfds;
   int fdin, n;

   while (1) {
      FD_ZERO(&readfds);
      FD_SET(fd1, &readfds);
      FD_SET(fd2, &readfds);
```

```
    if (fd1 < fd2)
      fdin = fd2+1;
    else
      fdin = fd1+1;
    int fd = select(fdin, &readfds, NULL, NULL, NULL);

    if (FD_ISSET(fd1, &readfds)){
      n = read(fd1, buf, 255);
      buf[n] = '\0';
      printf("Pipe 1: %s", buf);
      if (n < 255) {
        close(fd1);
        fd1 = open(path1, O_RDONLY | O_NONBLOCK);
      }
    }
    else {
      n = read(fd2, buf, 255);
      buf[n] = '\0';
      printf("Pipe 2: %s", buf);
      if (n < 255) {
        close(fd2);
        fd2 = open(path2, O_RDONLY | O_NONBLOCK);
      }
    }
  }

  close(fd1);
  close(fd2);

  return 0;
}
```