Práctica 8: Tuberías

Esta práctica se ha realizado en una máquina virtual corriendo el sistema operativo Ubuntu 15.04 LTS.

Apartado 1: Tuberías con nombre

***Ejercicio 1.*** Usar la orden mkfifo para crear una tubería (ej. $HOME/tuberia). 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, less, tail).

mkfifo tuberia

ls -l ./tuberia

stat ./tuberia

cat ./tuberia &

echo "Holaaaa" > tuberia

tail ./tuberia &

echo "Adiooos" > tuberia

***Ejercicio 2.*** Escribir un programa en C que abra la tubería con el nombre anterior ($HOME/tuberia) en modo sólo escritura, y escriba en ella el primer argumento del programa. En otro terminal, leer de la tubería usando un comando adecuado.

cat tuberia

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

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

**#include <fcntl.h>**

**#include <stdio.h>**

**#include <errno.h>**

**#include <unistd.h>**

**#include <string.h>**

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

**if (argc < 2)**

**return -1;**

**int fd = open("./tuberia", O\_WRONLY);**

**if (fd == -1) {**

**perror("Unable to open the pipe");**

**close(fd);**

**return -1;**

**}**

**size\_t arglen = strlen(argv[1]);**

**ssize\_t wr = write(fd, argv[1], arglen);**

**if (wr == -1) {**

**perror("Unable to write on the pipe");**

**close(fd);**

**return -1;**

**}**

**close(fd);**

**return 0;**

**}**

Apartado 2: Multiplexación de canales de entrada/salida

***Ejercicio 1.*** Crear otra tubería con nombre (ej. tuberia2). Escribir un programa que espere hasta que haya datos listos para leer en alguna de ellas. El programa debe escribir la tubería desde que se leyó y los datos leídos. Consideraciones:

● Para optimizar las operaciones usar un búfer para la lectura (ej. de 256 bytes).

● Usar read(2) para leer de la tubería y gestionar adecuadamente la longitud de los caracteres leídos.

● Las operaciones open(2) serán bloqueantes (hasta que se abra el otro extremo para escritura).

Para evitarlo, puede usarse la opción O\_NONBLOCK.

● Cuando el escritor termina y cierra la tubería, select considerará siempre listo el descriptor (para leer EOF) y no bloqueará. En este caso hay que reabrir la tubería o se puede abrir la tubería en el selector como lectura/escritura. Comprobar este punto abriendo las tuberías con diferentes opciones.

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

**int main() {  
 if (-1 == mkfifo("tuberia1", 0644)) {  
 perror("Unable to create the first pipe");  
 return -1;  
 }**

**if (-1 == mkfifo("tuberia2", 0644)) {  
 perror("Unable to create the second pipe");  
 return -1;  
 }**

**char buffer[257];  
 int nbuff = 0;**

**int pipe1 = open("tuberia1", O\_RDONLY | O\_NONBLOCK);  
 if (pipe1 == -1) {  
 perror("Unable to open the first pipe");  
 close(pipe1);  
 return -1;  
 }**

**int pipe2 = open("tuberia2", O\_RDONLY | O\_NONBLOCK);  
 if (pipe2 == -1) {  
 perror("Unable to open the second pipe");  
 close(pipe1);  
 close(pipe2);  
 return -1;  
 }**

**int selection, curpipe;  
 do {  
 fd\_set pipes;**

**FD\_ZERO(&pipes);  
 FD\_SET(pipe1, &pipes);  
 FD\_SET(pipe2, &pipes);**

**selection = select((pipe1 < pipe2) ? pipe2 + 1 : pipe1 + 1, &pipes, NULL, NULL, NULL);**

**if (selection > 0) {  
 int ntub = 0;  
 if (FD\_ISSET(pipe1, &pipes)) {  
 ntub = 1;  
 curpipe = pipe1;  
 } else if (FD\_ISSET(pipe2, &pipes)) {  
 ntub = 2;  
 curpipe = pipe2;  
 }**

**ssize\_t readsize = 256;  
 while (readsize == 256) {  
 readsize = read(curpipe, buffer, 256);  
 if (readsize == -1) {  
 perror("Unable to read");  
 close(pipe1);  
 close(pipe2);  
 return -1;  
 }**

**buffer[readsize] = '\0';  
 printf("Tuberia %i: %s", ntub, buffer);  
 }**

**if (readsize != 256 && ntub == 1) {  
 close(pipe1);  
 pipe1 = open("tuberia1", O\_RDONLY | O\_NONBLOCK);**

**if (pipe1 == -1) {  
 perror("Unable to open the second pipe");  
 close(pipe1);  
 close(pipe2);  
 return -1;  
 }  
 } else if (readsize != 256 && ntub == 2) {  
 close(pipe2);  
 pipe2 = open("tuberia2", O\_RDONLY | O\_NONBLOCK);**

**if (pipe2 == -1) {  
 perror("Unable to open the second pipe");  
 close(pipe1);  
 close(pipe2);  
 return -1;  
 }  
 }  
 }  
 } while (selection != -1);**

**perror("Error in the selection");  
 close(pipe1);  
 close(pipe2);  
 return -1;**

**}**

Apartado 3: Tuberías sin nombre

***Ejercicio 1.*** Comunicación por tuberías. Escribir un programa que emule el comportamiento de la shell en la ejecución de una sentencia en la forma: c omando1 argumento 1 | comando2 argumento2. El programa abrirá una tubería sin nombre y creará un hijo:

● El programa padre ejecutará “comando1 argumento1” y redireccionará la salida estándar al extremo de escritura.

● El hijo, ejecutará “comando2 argumento2”, en este caso la entrada estándar deberá duplicarse con el extremo de lectura de la tubería.

● 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 <stdio.h>**

**#include <errno.h>**

**#include <unistd.h>**

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

**#include <stdlib.h>**

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

**if (argc != 5) {**

**perror("Invalid arguments");**

**return -1;**

**}**

**int tub[2];**

**if (pipe(tub) == -1) {**

**perror("Unable to create the tub");**

**return -1;**

**}**

**pid\_t pid = fork();**

**int nfd;**

**switch (pid) {**

**case -1:**

**perror("Unable to fork");**

**exit(-1);**

**break;**

**case 0:**

**close(tub[1]);**

**nfd = dup2(tub[0], 0);**

**close(tub[0]);**

**execlp(argv[3], argv[3], argv[4], NULL);**

**break;**

**default:**

**close(tub[0]);**

**nfd = dup2(tub[1], 1);**

**close(tub[1]);**

**execlp(argv[1], argv[1], argv[2], NULL);**

**}**

**return 0;**

**}**

***Ejercicio 2.*** Comunicación *full-duplex* . Para la comunicación bi-direccional o full-duplex 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

escribiendo 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, cuando haya leído y procesado el mensaje (escribirlo por la salida estándar, terminal y esperar 1 segundo) enviará el carácter ‘l’ al proceso padre para indicar que está listo. Después de 10 mensajes enviará ‘q’ para indicar al padre que finalice.

**#include <stdio.h>**

**#include <errno.h>**

**#include <unistd.h>**

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

**#include <string.h>**

**#include <stdlib.h>**

**int main() {**

**int p\_h[2];**

**int h\_p[2];**

**if (-1 == pipe(p\_h) | -1 == pipe(h\_p)) {**

**perror("Unable to create the pipe");**

**return -1;**

**}**

**pid\_t pid = fork();**

**switch (pid) {**

**case -1:**

**perror("Unable to fork");**

**exit(-1);**

**break;**

**case 0:**

**close(p\_h[1]);**

**close(h\_p[0]);**

**char recv\_msg[257];**

**char parent\_msg[1] = {'l'};**

**int i;**

**for (i = 0; i < 10; i++) {**

**ssize\_t c\_size = read(p\_h[0], recv\_msg, 256);**

**if (c\_size == -1) {**

**perror("[Child] Error while reading from the pipe");**

**exit(-1);**

**}**

**recv\_msg[c\_size] = '\0';**

**printf("[Child] Message: %s", recv\_msg);**

**sleep(1);**

**if (i == 9)**

**parent\_msg[0] = 'q';**

**c\_size = write(h\_p[1], parent\_msg, 1);**

**if (c\_size == -1) {**

**perror("[Child] Error while wrinting into the pipe");**

**exit(-1);**

**}**

**}**

**close(p\_h[0]);**

**close(h\_p[1]);**

**exit(0);**

**break;**

**default:**

**close(p\_h[0]);**

**close(h\_p[1]);**

**char user\_msg[257];**

**char child\_msg[1] = {'l'};**

**while (child\_msg[0] != 'q') {**

**printf("[Parent] Enter a new message:\n");**

**ssize\_t p\_size = read(0, user\_msg, 256);**

**if (p\_size == -1) {**

**perror("[Parent] Error while reading from stdin");**

**exit(-1);**

**}**

**user\_msg[p\_size] = '\0';**

**p\_size = write(p\_h[1], user\_msg, p\_size + 1);**

**if (p\_size == -1) {**

**perror("[Parent] Error while wrinting into the pipe");**

**exit(-1);**

**}**

**while (child\_msg[0] != 'l' && child\_msg[0] != 'q') {**

**p\_size = read(h\_p[0], child\_msg, 1);**

**if (p\_size == -1) {**

**perror("[Parent] Error while reading from the pipe");**

**exit(-1);**

**}**

**}**

**}**

**close(p\_h[1]);**

**close(h\_p[0]);**

**exit(0);**

**}**

**return 0;**

**}**