Operating Systems [2020-2021]

Assignment 07 – Signals and Pipes

Introduction

The ability to communicate between processes that cooperate in solving a problem is essential. One of the simplest mechanisms used for this purpose are UNIX pipes. A pipe is a file or a special buffer, to which are assigned a descriptor for writing and one for reading. The writing and reading of data in a pipe follow a policy first-in first-out (FIFO). There are two types of pipes: one based on a buffer memory that can only be used among hierarchically related processes, and the other, named pipe, based on a special file (created in the *filesystem*) that can be used by any process, hierarchically related or not.

A signal can be viewed as an interrupt to the software. Operating systems use signals to report events to running processes. Signals may also be used between processes to enable communication between them. There are different types of signals according to the type of event that is reported. Some signals can be blocked or treated.

Objectives

Students concluding this work successfully should be able to:

- Use the signalling primitive between asynchronous processes (signals)
- Use pipes and named pipes
- Use input and output multiplexing on Unix (select)

Support Material

- K. A. Robbins, S. Robbins, “Unix Systems Programming: Communication, Concurrency, and Threads”, Prentice Hall:
  - Chapter 4 – UNIX I/O
  - Chapter 6 – Unix Special Files
  - Chapter 8 – Signals
- “Programming in C - UNIX System Calls and Subroutines using C” (http://www.cs.cf.ac.uk/Dave/C/CE.html):
  - Input and Output (I/O): stdio.h
  - Interprocess Communication (IPC): Pipes
  - Interrupts and Signals <signal.h>
  - signal() man page.

Exercises

Note: Only some of the exercises provided in this assignment will be done during the practical classes. The extra exercises should be done by the student as homework and any questions about them should be clarified with the teacher.

1. Key-press challenge

In this exercise you will implement a small game that measures how well you mentally count seconds. First it writes in the screen the number of seconds that are going to be counted. Two seconds later it starts the counting. The user must mentally count the seconds and press CTRL-C when that time is reached. On pressing the ENTER key another game is started. The user can press CTRL-Z at any time to terminate the program. Between each game CTRL-C is ignored, i.e, the CTRL-C is only active when the program waits for the user to stop the time count.

Complete the code given in file `press.c` in order to obtain an output similar to the one in the next text box. An executable solution is also given in the assignment files.

```
user@ubuntu:~ /SO/Ficha6$ ./press

Press CTRL-C in 9 seconds!
Countdown starting in 2 seconds... Get Ready!!
Start counting the seconds!!
^C

=> 10 seconds elapsed! You were too slow...

Press ENTER to continue!

Press CTRL-C in 8 seconds!
Countdown starting in 4 seconds... Get Ready!!
Start counting the seconds!!
^C

=> 2 second elapsed! You were too fast...

Press ENTER to continue!

Press CTRL-C in 3 seconds!
Countdown starting in 2 seconds... Get Ready!!
Start counting the seconds!!
^C

=> 3 seconds elapsed! Great shot!

Press ENTER to continue!^Z

^Z pressed. Do you want to abort? (y=yes)y
Ok, bye bye!
```

2. Measures

In this exercise you will implement a small measurement server. The server receives the measurements to display from 2 other processes (to which it connects by unnamed pipes) and the commands by a named pipe. Next, the overall architecture is displayed.

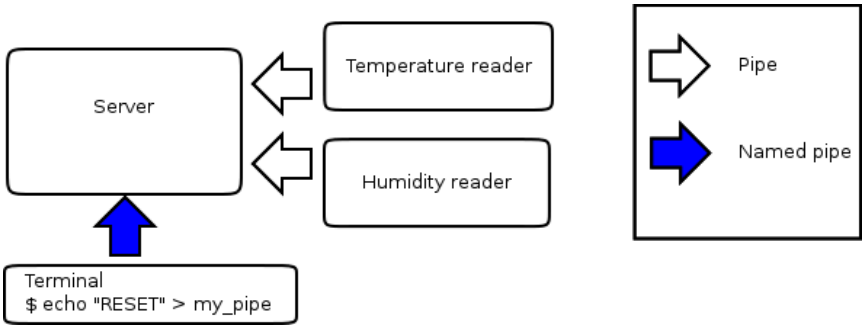

Figure 1 - Architecture

After starting, the Server creates two child processes, the Temperature Reader and the Humidity Reader, and one named pipe "my_pipe". Each child process generates respectively a random value of temperature (10 to 40ºC), and a random value for humidity (50-100%). Each value is generated each 3 seconds and sent to the Server process using an unnamed pipe. The Server process reads each value from the corresponding pipe, adds the value to a variable that stores the sum of all values of that type collected before, increments 1 to the variable that stores the number of samples of that type, and displays the value. The user uses a Terminal and a named pipe ("my_pipe") to send commands to the Server. There are 4 different commands available:

- "AVG TEMP" – writes in the screen the average temperature considering all samples saved;
- "AVG HUM" - writes in the screen the average humidity considering all samples saved;
- "RESET" – resets the values of the samples and the total sum of each measurement received;
- "SHUTDOWN" – shuts down the server cleaning all used resources.
- When a command is not recognized, the Server prints "[Server Received unknown command]"

Each command is sent by a user through a command line terminal.

Ex: `$ echo "RESET" > my_pipe`

The Server also exits on receiving a CTRL-C. Before exiting, all resources used must be cleaned.

After a process is created, all file descriptors related to pipes that are not used should be closed.

The Server must block until a message is received from any of the pipes.

Complete the code given in file `measures.c` in order to obtain an output similar to the one in the next text box.

```
user@ubuntu:~/SO/Ficha6$ ./measures

Listening to all pipes!

[SERVER received new humidity]: 76 %
[SERVER received new temperature]: 11°C
[SERVER received new humidity]: 70 %
[SERVER received new temperature]: 39°C
[SERVER Received "AVG HUM" command]
Average Humidity= 73.00 %
[SERVER received new humidity]: 56 %
[SERVER received new temperature]: 36°C
[SERVER received new humidity]: 78 %
[SERVER received new temperature]: 23°C
[SERVER Received "AVG TEMP" command]
Average Temperature= 27.25 °C
[SERVER received new humidity]: 76 %
[SERVER received new temperature]: 10°C
[SERVER Received unknown command]: RESxxx
[SERVER received "RESET" command]
Counters reset!
[SERVER received new humidity]: 93 %
[SERVER received new temperature]: 12°C
[SERVER Received "AVG TEMP" command]
Average Temperature= 12.00 °C
[SERVER received "SHUTDOWN" command]
Server shutdown initiated!
Bye bye
```

3. Signal tester

In this exercise you will implement a program that blocks, ignores and handles different signals by using `sigaction` and `sigprocmask`. The program will ignore SIGINT (CTRL-C) and SIGTSTP (CTRL-Z). Each 10 seconds it alternates from blocking to unblocking both SIGUSR1 and SIGUSR2.

Open a separate terminal window to test the program. From the new shell you can send signals to the running program by using the command `kill -{signal} {PID}`.

Ex: `kill -SIGUSR1 23500`

To exit the program, you can use the default `Kill` signal, which is SIGTERM, by using the command `kill {PID}`.

Complete the code given in file `signal_tester.c` . An executable solution is also given in the assignment files.

Note: Try to send the signal SIGUSR1 several times, while that signal is blocked. How many times will it be handled when the signal is unblocked?

4. SMS Mobile operator

Implement a simulator for sending and receiving SMS messages between mobile phones. This simulator goal is to implement the functionality of the antenna and the communication between the mobile phone and the mobile operator infrastructure. The mobile operator (represented by a process) is responsible for creating two pipes for each mobile phone it allows to connect (one to write and one to read), representing the mobile operator cells. Each mobile phone, represented by a process, will use the available cell (composed by two pipes) to register, unregister and send SMS.

The registration requires a two messages process, the first, from the mobile phone to the mobile operator with the number of the phone and the respective pin, and the second, from the mobile operator to the mobile phone with the result of the registration and the signal strength.

After the registration the mobile phone can start sending SMS, by sending a message to the mobile operator with the SMS number, the SMS text and the SMS destination phone number. The mobile operator replies to the mobile phone with a message containing the number of the SMS sent and the result of the operation requested (SMS sent successfully, or unsuccessfully).

Before a mobile phone is turned off, the mobile phone should send an unregister message to the mobile operator. At this point, the mobile operator can use the released resources to another mobile phone.

Use the following information to implement this exercise:

- Create new mobile phones every 5 seconds
- Send SMS messages every 2 to 3 seconds
- The mobile operator has 5 cells
- Each SMS takes 1 second to be processed by the mobile operator
- Each mobile phone sends 10 to 20 sms before turning off
- Share the state of the cells by shared memory
- Exit when a SIGINT is received

5. SMS Mobile operator (named pipe)

Re-implement exercise 1, but split the simulator in two different applications, the mobile operator, and the mobile phone. Use named pipes to communicate between the two applications. The mobile phone application should start by creating all available connections named pipes and one additional named pipe that should be used by the mobile phone to request for one available named pipe.

6. SMS Mobile operator (signals)

Update the two previous exercises to include the handling of signals. Both simulators should close gracefully when CTRL+C is pressed. CTRL+Z, when executed in the Mobile operator, should list in the screen the total of messages received by each mobile phone.

7. OutRun

In this exercise you will implement a version of the 1986 game "Sega OutRun". The program will create a road that will randomly turn left or right. The car represented by a "V" will move left or right according to signals sent by the user.

```
>    V   <
 >   V    <
  >   V    <
   >    V   <
     >    V   <
       >    V   <
        >   V    <
         >  V     <
          > V      <
```

Complete the code given in file `outrun.c` taking into account the following rules:

- The game has a maximum width of MAX characters;
- The road has a width of SIZE characters;
- The road starts with a shift to the right and then randomly shifts direction;
- The road is managed by a child process;
- The original process receives shifting commands from the user and saves the new position of the car in shared memory, where it can be accessed by the child process that is responsible to print the road;
- The game ends when a maximum of LINES is reached or when the car crashes;
- CTRL-Z: moves the car 1 character to the left;
- CTRL-C: moves the car 1 character to the right;
- The process that manages the road should ignore CTRL-Z and CTRL-C.

An executable solution is also given in the assignment files.