Programação Concorrente

Week 6 – Lab 6

Communication between threads

pipe(2) - Linux manual page - man7.org

read(2) - Linux manual page - man7.org

write(2) - Linux manual page - man7.org

The Linux Programming Interface – Sections 44.1 44.2

In this laboratory students will use pipes to transmit information from one thread to the other.

Pipes, are special objects that allow programs to write data onto it and later read the same data. These objects are created using the **pipe** function and afterward regular **read** and **write** functions can be used on it.

The function that creates and initializes a pipe receives as arguments one array of two integers. After its correct execution, the first integer of the array corresponds to the place where the program can read data and the the second integer to the place to write data.

<table>
<tr><td colspan="2">main()</td></tr>
<tr><td colspan="2"><pre>int pipefd[2];
pipe(pipefd);</pre></td></tr>
<tr><td>Thread 1</td><td>Thread 2</td></tr>
<tr><td><pre>int a = 12;
write(pipefd[1],&a, sizeof(a));</pre></td><td><pre>int b;
read(pipefd[0],&b, sizeof(b));
printf(“%d”, b);</pre></td></tr>
</table>

For pipes to work correctly with multiple threads (one thread writing and multiple threads reading) it is necessary that the amount of data written (**count** argument) is always the same and equal to the data that is going to be read.

1 Exercise 1

Modify the **lab5-pipe-example.c** code so that the **main** writes on the **pipe_fd[1]**, but 4 threads read concurrently from **pipe_fd[0]**. The output should be similar to the following:

```
going to write 0 into pipe_fd[1]
going to write 1 into pipe_fd[1]
        Thread 1 just read 1 from pipe_fd[0]
going to write 2 into pipe_fd[1]
        Thread 3 just read 2 from pipe_fd[0]
going to write 3 into pipe_fd[1]
        Thread 2just read 3 from pipe_fd[0]
...
```

(The order of the prints by threads may be different.)

It is not necessary to perform a **pthread_join** because the loop in the **main** should be infinite. Between each **write** a **sleep(1)** should be executed.

2 Exercise 2

The **lab6-serial-prime.c** program generates a certain amount of random numbers and verifies if they are prime sequentially in the **main**.

Modify the program so that the numbers generated in the **main** are written into a pipe and 4 other threads reading them from the pipe and verify if they are prime.

This version of the program should only print every prime number found, it does not need to terminate: after all numbers are processes the threads should continue to run and remain blocked in the read of the pipe.

This program already contains the skeleton for the creation and execution of the threads. Besides the creation of threads, students should follow the next steps to implement this exercise:

- STEP 1 - Declaration of a **int pipe_fd[2]**
- STEP 2 - Creation of the pipe in **main**
- STEP 3 - After the generation of the random number, that number should be written into the pipe.
- STEP 4 - Each thread should read a random number from the pipe before verifying if t is prime.

3 Exercise 3

Modify the previous program so that, after the verification of all the random numbers, the 4 threads print how many prime number found and terminate (with **pthread_exit**), and subsequently the **main** returns from the **pthread_join** and terminates.

Implement this exercise using these two different approaches:

- **Exercise 3-a** - the **main** may send a message trough the pipes for all threads to indicate the end of the data;
- **Exercise 3-b** - declare a counter, initialize it with the total number of random numbers, decremented it whenever a number is read, and terminate each thread when such counter reaches 0 .
    - This solution may not work on certain cases , which will be solved in Laboratory 6 .

4 Exercise 4

Modify the resolution of the previous exercise (**Exercise 3-a**) so that the **main** prints the total number of found primes by summing the number of primes found by each thread.

5 Exercise 5

Modify the resolution of **Exercise 1** so that, instead of writing integers on the pipe, the **main** writes strings in the pipe. Each string will only be read by a single thread, after which its length is printed.

The **main** should continuously read strings from the keyboard using the **fgets** function and write them to the pipe.

To guarantee that each thread will read completely a single line, the **count** argument for all writes and reads should be always the same: define a maximum length for the strings and always use such value when writing and reading from the pipe.