Programação Concorrente

Week 6 – Lab 6

Shared data

pthread_mutex_init(3p) - Linux manual page - man7.org

pthread_mutex_lock(3p) - Linux manual page - man7.org

pthread_mutex_destroy(3p) - Linux manual page - man7.org

The Linux Programming Interface – Sections 30.1

In this laboratory students will start to use locks/mutex to guarantee that the concurrent access to the same variable by multiple threads renders always a correct value.

The supplied code has two threads that loop over an array with random numbers and count how many of them are prime. In this simplistic version, each thread iterates over the whole set of values and just increments a local variable. The main waits for the termination of every thread before exiting.

In the following exercises students will implement different versions of this solution where the threads need to access and update shared data.

1 Sequential retrieving of values

Modify the provided program so that each thread verifies his own set of random numbers, and each values is not verified by two threads (i.e. each value is only verified by one thread).

This work division is not defined by an expression of specific division algorithm, but by the order of access to the global array: each thread will access and process one of the unprocessed numbers on the array (the one in the lowest index).

The program should have a new variable that stores the index of the next number to be processed (**next_random_index**) and all the threads: access the value of **next_random_index** , and process the corresponding random number from the array, and increment **next_random_index**.

After each thread increments this index (**next_random_index)** it is guaranteed that other threads will not process the same number, but the next one in the array.

When the **next_random_index** reaches **LENGTH_ARRAY** every thread should exit.

2 Race condition

Experiment running the code with different data configurations, array lengths and number of thread:

- delete the comment on line **//rand_num_array[i] = i;**
- increase the array length to **100000**
- increase number of threads to 4 or 5

Try to find an execution where the sum of the numbers processed by all threads is not correct, as in the next example with an array of of integers from 0 to 100000:

```
Thread 3 found 2300 primes on 24518 numbers
Thread 0 found 2671 primes on 26882 numbers
Thread 1 found 2399 primes on 24878 numbers
Thread 2 found 2300 primes on 24329 numbers
```

3 Critical region / Mutual exclusion

Correct the previous code by defining a critical region for the **next_random_index** global variable.

To guarantee that the access to the critical region does not affect the final result of the program execution, implement mutual exclusion using **pthread_mutex**:

- **pthread_mutex_t** – data type corresponding a mutex
- **pthread_mutex_init** – creation and initialization of a mutex
- **pthread_mutex_lock** – function to call before accessing the critical region
- **pthread_mutex_unlock** – function after leaving the critical region

- **pthread_mutex_destroy** – to destroy the mutex before exiting the program

4 Storage of results

Modify the previous program so that each thread stores the prime numbers in a shared array:

- declare a global variable (called **prime_array**) of the same length of **rand_num_array**
- every time a prime number is found, store it in the new variable and increment the global counter of prime numbers.

After the **pthread_join()**, the main should print the number of prime numbers stored in the **prime_array**.

Use all the necessary mutexes to guarantee that all prime number are correctly stored.