

Lab 4: Threading

COP4600 – Operating Systems

Overview

The aim of this exercise is to familiarize you with multithreading in C++. In the right situation, threads can significantly speed up your programs by allowing parts of it to run concurrently rather than sequentially. Here is a simple example:

```
1  #include <thread>
2  #include <iostream>
3
4  void foo(int id) {
5      std::cout << "My id is " << id << std::endl;
6  }
7
8  int main() {
9
10     std::cout << "Spawning a thread." << std::endl;
11
12     // Create a thread. The first argument is the function that the thread should execute.
13     // The second argument will be passed to the function itself.
14     std::thread th1(foo, 4);
15
16     // Wait for the threads to finish executing before continuing.
17     th1.join();
18
19     std::cout << "Thread has finished executing." << std::endl;
20
21     return 0;
22 }
```

Note the use of the `join()` function above. Without it, it would be possible for the main function to finish executing and exit before the thread has completed execution.

Structure

To complete this exercise, you will be writing a program that spawns 10 threads, each of which will attempt to do the same job. Due to the nature of threads, you'll notice that they finish in a different order every time.

1. Write a C++ program that takes a number as a command-line argument (i.e. `./thread 1414`)

- Write a function inside this program with two parameters: one is an ID number, and the other will be the number passed in as a command-line argument.
 - This function will generate random numbers between 0 and 9999 until it generates one that matches the number given as a command-line argument, then print “Thread <id> completed.”
- In your program’s main function, spawn 10 threads, each of which will call your new function with a unique ID (0 through 9) and the number given as a command-line argument.

Note: *Your threads must be spawned inside of a loop. You should NOT have 10 individual calls to thread() in your program.*

- Finally, once all of your threads have finished generating numbers, print “All threads have finished finding numbers.”

Note: *In order to compile your program with threads, you will need to use the flags `-std=c++11 -pthread` with the `g++` command.*

Enabling Race Conditions

In order to ensure that the execution of threads finish in a randomized order, you can use the **nice** utility. **nice** is a program that allows setting or altering the priority of a process. You should give your process the lowest priority to ensure that the CPU will execute it less often. In addition, you should lower your virtual machine’s memory and maximize the number of cores. This will slow the process down and cause the threads to execute out of order due to the interruptions and race conditions.

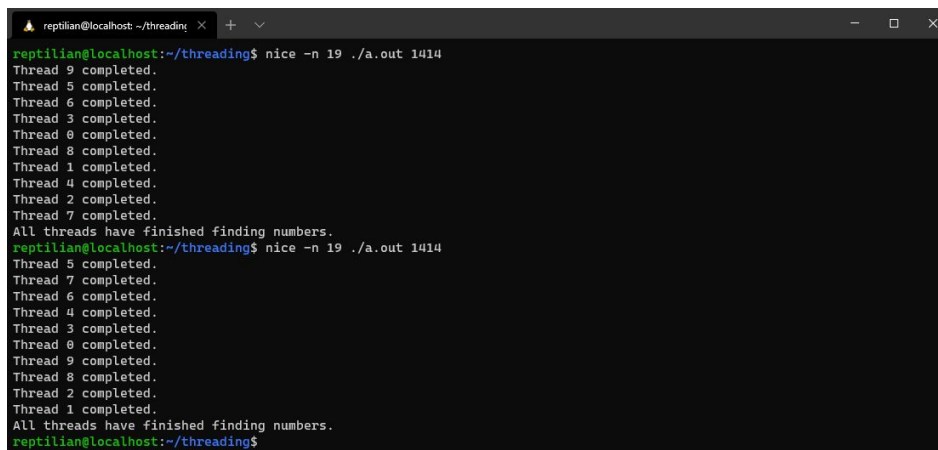
Read about **nice** here: <https://man7.org/linux/man-pages/man1/nice.1.html>

Note: *RAM and CPU settings differ depending on your machine. You should give your virtual machine at least 512 MBs.*

Submissions

You will submit the following at the end of this exercise on Canvas:

- C++ source file for your program
- Screenshot of the output from running your program *twice*.



```
reptilian@localhost: ~/threading
reptilian@localhost:~/threading$ nice -n 19 ./a.out 1414
Thread 9 completed.
Thread 5 completed.
Thread 6 completed.
Thread 3 completed.
Thread 0 completed.
Thread 8 completed.
Thread 1 completed.
Thread 4 completed.
Thread 2 completed.
Thread 7 completed.
All threads have finished finding numbers.
reptilian@localhost:~/threading$ nice -n 19 ./a.out 1414
Thread 5 completed.
Thread 7 completed.
Thread 6 completed.
Thread 4 completed.
Thread 3 completed.
Thread 0 completed.
Thread 9 completed.
Thread 8 completed.
Thread 2 completed.
Thread 1 completed.
All threads have finished finding numbers.
reptilian@localhost:~/threading$
```

Figure 1: *An example screenshot with output from running the program twice.*