MULTI-THREADED PRIME NUMBER GENERATOR

OVERVIEW

This assignment will explore the use of threads in a computational setting and attempt to quantify their usefulness. It will also introduce a common problem encountered in multi-threaded and multi-process environments: a race condition. Although you will not be required to solve the concurrency problems, after this assignment, you should be aware of their existence and the difficulties they present.

THE PROGRAM

When completed, your program (named `mt-prime.c`) should produce a list of prime numbers between 1 and N. The program will have two command-line arguments. The first argument, N, is the maximum number to test for primality. The second argument, T, is the number of threads to create to compute the results. For example, the following execution would use 8 threads to compute the prime numbers from 1 to 10000:

```
./mt-prime 10000 8
```

The list of prime numbers should be written, one per line, to standard output (`stdout`). The time required to produce the list should be written to the standard error stream (`stderr`). The format of the timing output should be a comma-separated record containing the values N, T, and the time required to complete the program in seconds and nanoseconds. The following example would be a possible result of the above execution. (The timing values are fabricated and just an example of format).

```
10000,8,3.852953000
```

IMPLEMENTATION SUGGESTIONS

I suggest keeping a global variable that represents the highest number checked for primality thus far, COUNTER. As each thread becomes ready for work, it repeatedly increments the value of COUNTER and determines if the new value is prime or not until all values have been tested. Values that are prime numbers should be printed to standard output. Because multiple threads may read the value of COUNTER simultaneously, expect some duplication of answers. Once threads reach the highest number to test, they terminate. All worker threads must join the main thread before the program computes the elapsed time, prints it, and terminates.

When using the BASH shell, you have the option of redirecting standard input, standard output, and/or standard error to a file. The < and > symbols are used to redirect the first two. To redirect `stderr`, you can use `2>`. For example, this command would display `stdout` on the screen but send `stderr` to the file *results.csv*.

```
./mt-prime 10000 8 2> results.csv
```

You can also append to an output file by using the `>>` and `2>>` symbols. To redirect both `stdout` and `stderr` to the same file, you can reference back to a previous variable in the argument list. First, redirect `stdout`, and then send `stderr` to the same place as `stdout` using `2>&1`:

```
./mt-prime 10000 8 > results.csv 2>&1
```

Experiment with this functionality before trying to save the output of your program executions. It is easy to delete your files if not used correctly. Also, experiment with the appropriate system call for time measurement (see below) to compute the elapsed time from the start of the program to the full completion. The elapsed time should approximate the runtime of your program as assessed by considering the system-wide real-time clock.

SUGGESTED SYSTEM CALLS

Most implementation details are left for you to decide. Below is a list of system calls that you need to use in the implementation of your code. You may use other system calls as necessary.

- `pthread_create(3)` – create a new thread
- `pthread_exit(3)` – terminate a thread
- `pthread_join(3)` – wait for a child thread to terminate
- `clock_gettime(3)` – get the time of a specified clock

DELIVERABLES

Your project submission should follow the instructions below. Any submissions that do not follow the stated requirements will not be graded.

1. Follow the submission requirements of your instructor as published on *eLearning* under the Content area.
2. You should have at a minimum the following files for this assignment:
    a. `mt-prime.c`
    b. analysis.doc (the results from an experiment including a table with the data, a graph, & conclusions)
    c. README (defines the functionality you want to expose)
    d. algorithm.doc (a short write-up of the algorithm used for primality test)
    e. Makefile

Your program should perform an experiment involving thread counts from 1 to 30 and a large *N* for the list of prime numbers. Select a value of *N* to force your program in taking measurable time to execute. Create a table and graph to show the time required for the experiment as the number of threads increases for a fixed *N*. Analyze the results and discuss performance improvements or degradation that you may observe in your experiment. As time measurements are influenced by other programs running on your machine, you must consider running your time measurement tests multiple times for a fixed T and N and taking the average of each run.

Submit a *README* file that describes purpose and usage of your program. Keep in mind that documentation of source code is an essential part of programming. If you do not include comments in your source code, points will be deducted. If you do not refactor code appropriately, points will be deducted. In addition to the source code and the *README* file, submit a short write-up describing your algorithm for testing primality, the results from your experiment, including the table, the graph and your assessment about the performance improvements through parallel execution.

GRADING

This project is worth 100 points total. The points will be given based on the following criteria:

- 55 points for correct implementation of code,
- 20 points for the write-up of your experiment,
- 10 points for the discussion of algorithm,
- 5 points for README and Makefile,
- 10 points for appropriate documentation of source code and code refactoring.

DUE DATE

The project is due Wednesday, Oct. 9th by 3:00 pm to the Dropbox for project 3 in *eLearning*. I will not accept submissions emailed to me or the grader. Upload ahead of time, as last minute uploads may fail. Please review the policy in the syllabus regarding late work.

COMMENTS

Start with this project right away to leave yourself enough time for handling unexpected problems. Insert lots of output statements to help debug your program. You should consider using debugging techniques from the first programming practice. Compile with the `-DDEBUG=1` option until you are confident your program works as expected. Then recompile without using the `-D` option to "turn off" the debugging output statements.