MULTI-THREADED COLLATZ STOPPING TIME 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. You will need to solve the concurrency problem as part of this assignment. However, you should experiment with the application of thread synchronization to prevent race conditions and the bottlenecks it introduces.

THE PROGRAM

When completed, your program (named `mt-collatz.c`) should produce a list of Collatz sequences for numbers between 1 and N. The program will have two command-line arguments. The first argument, N, defines the range of numbers for which a Collatz sequence must be computed. The second argument, T, is the number of threads to create to compute the results in parallel. For example, the following execution would use 8 threads to compute Collatz sequences for numbers between 2 and 10,000:

```
./mt-collatz 10000 8
```

Each thread computes a Collatz sequence according to the following formula:

$$a_i = \begin{cases} n, & i = 0 \\ f(a_{i-1}), & i > 0 \end{cases}$$

$$f(n) = \begin{cases} \frac{n}{2}, & n \text{ is even} \\ 3n+1, & n \text{ is odd} \end{cases}$$

The smallest value of *i* for which $a_i = 1$ is called the stopping time. More details on the problem can be found here. The thread counts the stopping time for each Collatz sequence in a global array so that all threads together essentially compute a histogram of Collatz stopping times across a range of numbers. A visualization of a histogram of Collatz stopping times can be found here. Stopping times are shown on the X-axis, frequency values on the y-axis. Your program must initialize the array with zeros before it creates the threads and computes the Collatz stopping times for numbers 2 through N. Each thread selects a number in the specified range, computes the stopping time and records the result in the global histogram array before it chooses the next number not yet selected by another thread. This process continues until all threads together have computed stopping times up to the value of *N*. The final histogram should be printed to standard out (`stdout`) in the following format:

<k=1,...,N>, <frequency of Collatz stopping times where *i = k*>

Each line contains a value *k* in the range of 1 to 1000 and the corresponding frequency of Collatz stopping times for *i = k*, i.e. Collatz numbers that stop at the value k (a_k=1). The time required to produce the entire histogram must 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).

```
500,8,3.852953000
```

IMPLEMENTATION SUGGESTIONS

In addition to the global histogram array, I suggest keeping a global variable called *COUNTER* that represents the next number for which a Collatz stopping times must be computed. *COUNTER* should start at 2 and end at *N*. As each thread becomes ready for work, it repeatedly increments the value of *COUNTER* and then performs its job of computing a Collatz stopping time and recording the stopping time in the histogram array. Because multiple threads may read the value of COUNTER simultaneously, there is the possibility of a race condition. Therefore, you must introduce thread synchronization using a mutex variable (also known as a lock) to avoid duplication of values in the histogram array. 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, prints out the histogram array and finally 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-collatz 10000 8 [-nolock] 2> results.csv
```

You can also append to an output file by using the `>>` and `2>>` symbols. The parameter `–nolock` is optional indicating that when it is set, you should not use a lock in the program to prevent race conditions on the COUNTER variable. The option will allow you to experiment with race conditions in the code.

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
- `pthread_mutex_init(3)` – initializes a mutex variable
- `pthread_mutex_lock(3)` – locks the mutex variable
- `pthread_mutex_unlock(3)` – unlocks the mutex variable

Hint: Use the real-time setting of `clock_gettime()`. Also, this function returns elapsed time measured in seconds and nanoseconds. The nanoseconds count the fraction of a second of the system clock measured at any given time. In order to compute the correct elapsed time you may have to "borrow" a second from the difference of seconds to compute the correct time difference measured in nanoseconds.

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-collatz.c`
   b. analysis.doc (the results from an experiment including a table with the data, a graph, & conclusions)
   c. histogram of your Collatz stopping times for a chosen value of N
   d. README (describes any significant problems you had)
   e. `Makefile`

Your program should perform an experiment involving thread counts from 1 to 30 and a large *N* for the range of Collatz stopping times to be computed. Select a reasonably large 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 any significant problems you had. 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 (if needed), submit a visualized histogram of your Collatz stopping time, the results from your experiment, including the table, the graph and your assessment about the performance improvements through parallel execution.

DUE DATE

The project is due as indicated by the Dropbox for project 2 in *eLearning*. Upload your complete solution to the dropbox and the shared drive. I will not accept submissions emailed to the grader or me. Upload ahead of time, as last minute uploads may fail.

COMMENTS

I strongly recommend you starting to work on this project right away to leave enough time for handling unexpected problems. Insert many 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.

GRADING

This project is worth 100 points total. The rubric used for grading is included below. Keep in mind that there will be deductions if your code does not compile, has memory leaks, or is otherwise, poorly documented or organized. The points will be given based on the following criteria:

Correct Submission Format	Perfect	Deficient		
eLearning	5 points individual files have been uploaded	0 points files are missing		
shared drive	5 points individual files have been uploaded	0 points files are missing		
Compilation	**Perfect**	**Good**	**Attempted**	**Deficient**
Makefile	5 points make file works; includes clean rule	3 points missing clean rule	2 points missing rules; doesn't compile project	0 points make file is missing
compilation	10 points no errors	7 points some warnings	3 points many warnings	0 points errors
Documentation & Style	**Perfect**	**Good**	**Attempted**	**Deficient**
documentation & program structure	5 points follows documentation and code structure guidelines	3 points follows mostly documentation and code structure guidelines; minor deviations	2 points some documentation and/or code structure lacks consistency	0 points missing or insufficient documentation and/or code structure is poor; review sample code and guidelines
Collatz & Threads	**Perfect**	**Good**	**Attempted**	**Deficient**
Creates threads with pthread_create	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing
Computes Collatz sequence for a number	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing
Computes stopping times concurrently	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing
pthread_join	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing
Computes elapsed time using clock_gettime	5 points correct, completed	3 points minor errors	2 points incomplete	0 points missing
Addresses race-condition problems	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing
Chart, analysis, and conclusions	**Perfect**	**Good**	**Attempted**	**Deficient**
Chart	5 points correct, completed	3 points minor errors	2 points incomplete	0 points missing
Analysis	5 points correct, completed	3 points minor errors	2 points incomplete	0 points missing
Conclusions	5 points correct, completed	3 points minor errors	2 points incomplete	0 points missing