CSCI 3120 Assignment 3

Due date: 11:59pm, Sunday, July 3, 2022, submitted via Git

Objectives

This assignment has several objectives: (i) to provide an opportunity to write a multithreaded program; (ii) to reinforce your understanding of locks, critical sections, and process synchronization; and (iii) to provide additional practice in C programming and understanding specifications.

Preparation:

1. Complete Assignment 0 or ensure that the tools you would need to complete it are installed.
2. Review the Assignment 1 and 2 specifications. Assignment 3 builds on Assignment 2.
3. Complete Assignment 2 and/or review the provided solution to Assignment 2.
4. Clone your assignment repository:
   `https://git.cs.dal.ca/courses/2022-summer/csci-3120/assignment-3/????.git`
   where ???? is your CSID.
5. Copy either your own solution or the provided solution as a starting point for Assignment 3 because Assignment 3 builds on Assignment 2. A solution will be accessible on June 17 from Brightspace.
6. **Important**: If you are using CLion with CMake you will need to modify your CMakeLists.txt file by adding the following bolded statements:

```
cmake_minimum_required(VERSION 3.11)
project(miner C)

set(CMAKE_C_STANDARD 99)
set(THREADS_PREFER_PTHREAD_FLAG ON)

add_executable(miner main.c)
find_package(Threads REQUIRED)
target_link_libraries(miner PRIVATE Threads::Threads)
```

The repository has the same structure and caveats as Assignment 1 and Assignment 2.

Background[1]

Searching for the *nonce* when mining blocks is considered the computation intensive portion of mining and it is the discovery of this *nonce* that allows a miner to claim that they have mined the block. Naturally, if you have an expensive mining rig with many CPU cores, you would like to use all of them to search for the *nonce* as quickly as possible and claim the prize.

Problem Statement

Extend your program from Assignment 2 (or the provided solution) to make your miner multithreaded so that it spawns multiple threads when searching for the *nonce*. Your program must be compiled to an executable called **`miner`**. The program will read in five events. The events are the same events as in Assignment 2, except that the **mine** event will now take one parameter, specifying the number of threads that should be created to search for the *nonce.*

[1] This background description assumes that you have read and understood the background for Assignment 1 and 2.

Please see the Problem Statement in Assignment 1 and Assignment 2 for a description of the events.

On a **mine** event, your program must do the following:

1. Construct the block in the same manner as in Assignment 2.
2. When searching for the *nonce*, your program must spawn the specified number of threads and each thread will search a portion of the search space (see Processing Section for details).
3. Once the *nonce* is found, the threads should be destroyed and the rest of the computation proceeds as in Assignment 2.

Note: the changes to the simulator in Assignment 3 deal with how the *nonce* is selected. All other functionality remains the same.

Input

Your program will read its input from **stdin**. The input will consist of events in the format described below and in Assignment 1.

End, Epoch. Transaction, and Block Events

Please see the Input section in Assignment 1 and Assignment 2 for the format description.

Mine Event

The format of the **mine** event is now the word `MINE` followed by an integer, e.g.,

```
MINE threads
```

where **threads** is greater than 0 and denotes the number of threads to spawn.

Processing

The **end**, **epoch**, **transaction**, and **block** events are processed in the same way as in Assignments 1 and 2.

The **mine** event is processed in the same way as in Assignment 1, except that when the *nonce* is searched for the program must spawn the number of threads. The threads are assigned identifiers 0 … ***threads*** *-1.* Thread **t** is assigned to check all *nonces* whose values are congruent to (**t mod N) where N is the number of threads**, e.g., ***t, N + t, 2N + t, 3N + t, …***, where $0 \leq t < N = threads$.

Once the *nonce* is discovered, all threads stop their search. Your program can then continue with the found *nonce* as it does in Assignment 2.

Output

The output for this assignment is the same as that of Assignment 2, except that for each *nonce* that the threads check, they should print out the following message:

```
Thread T checking nonce N
```

where ***T*** is the assigned identifier of the thread and **N** is the current *nonce* being checked. The *nonce* should be in outputted hexadecimal format. Each message should be on a single line and terminated by new line.

Note: that the order of the messages may be different each time because the scheduler may schedule the threads differently each run. However, as long the messages from each thread are in order, and each message is on its own line and is uncorrupted, that is fine.

Example: Same Input, Different Outputs

Input	Output
TRX 12358 Cat Dov 150000 20 TRX 35813 Eve Fin 150000 25 TRX 11235 Al Beth 150000 40 MINE 3 END	Adding transaction: 12358 Cat Dov 150000 20 Adding transaction: 35813 Eve Fin 150000 25 Adding transaction: 11235 Al Beth 150000 40 Block mined: 1 0 0x00000000 3 11235 Al Beth 150000 40 12358 Cat Dov 150000 20 35813 Eve Fin 150000 25 Thread 1 checking nonce 0x00000001 Thread 2 checking nonce 0x00000002 Thread 0 checking nonce 0x00000000 Thread 0 checking nonce 0x00000003 Thread 1 checking nonce 0x00000004 Thread 2 checking nonce 0x00000005 0x00000003 0x00996b1f

Input	Output
TRX 12358 Cat Dov 150000 20 TRX 35813 Eve Fin 150000 25 TRX 11235 Al Beth 150000 40 MINE 3 END	Adding transaction: 12358 Cat Dov 150000 20 Adding transaction: 35813 Eve Fin 150000 25 Adding transaction: 11235 Al Beth 150000 40 Block mined: 1 0 0x00000000 3 11235 Al Beth 150000 40 12358 Cat Dov 150000 20 35813 Eve Fin 150000 25 Thread 1 checking nonce 0x00000001 Thread 1 checking nonce 0x00000004 Thread 1 checking nonce 0x00000007 Thread 1 checking nonce 0x0000000a Thread 1 checking nonce 0x0000000d Thread 0 checking nonce 0x00000000 Thread 0 checking nonce 0x00000003 Thread 1 checking nonce 0x00000010 Thread 2 checking nonce 0x00000002 Thread 2 checking nonce 0x00000005 0x00000003 0x00996b1f

Note: The order of the threads will change from execution to execution.

Assignment Submission

Submission and testing are done using Git, Gitlab, and Gitlab CI/CD. You can submit as many times as you wish, up to the deadline. Every time a submission occurs, functional tests are executed and you can view the results of the tests. To submit use the same procedure as Assignments 1 and 2.

Hints and Suggestions

- You will need to use locks to protect critical sections in your program. The sample solution has two critical sections. One is easy to identify, the other less so.
- The number of threads you will need to create will differ from MINE event to MINE event, so do not assume the same number all the time.
- To make things simpler, the solution places the nonce search code (and all the multithreading) into a separate file (`nonce.c`). This way, there is less chance of introducing bugs in the rest of the code.

Grading

The assignment will be graded based on three criteria:

Functionality: "Does it work according to specifications?". This is determined in an automated fashion by running your program on a number of inputs and ensuring that the outputs match the expected outputs. The score is determined based on the number of tests that your program passes. So, if your program passes t/T tests, you will receive that proportion of the marks.

Quality of Solution: "Is it a good solution?" This considers whether the approach and algorithm in your solution is correct. This is determined by visual inspection of the code. It is possible to get a good grade on this part even if you have bugs that cause your code to fail some of the tests.

Code Clarity: "Is it well written?" This considers whether the solution is properly formatted, well documented, and follows coding style guidelines. A single overall mark will be assigned for clarity. Please see the Style Guide in the Assignment section of the course in Brightspace.

If your program does not compile, it is considered non-functional and of extremely poor quality, meaning you will receive 0 for the solution.

The following grading scheme will be used:

Task	100%	80%	60%	40%	20%	0%
Functionality (20 marks)	Proportionate to the fraction of tests passed.					
Solution Quality (20 marks)	Solution uses multithreading in an efficient and correct manner. Appropriate locks are used.	Solution uses multithreading correctly. Appropriate locks are used.	Solution uses multithreading correctly. Locks are not entirely correct.	Solution does not use multithreading correctly or does not use locks.	An attempt has been made.	No code submitted or code does not compile
Code Clarity (10 marks) Indentation, formatting, naming, comments	Code looks professional and follows all style guidelines	Code looks good and mostly follows style guidelines.	Code is mostly readable and mostly follows some of the style guidelines.	Code is hard to read and follows few of the style guidelines.	Code is illegible	