CSCI 3120 Assignment 4

Due date: 11:59pm, Sunday, July 24, 2022, submitted via Git

Objectives

This assignment has several objectives: (i) provide an opportunity to implement thread synchronization in a multithreaded program (ii) reinforce the concept of process synchronization and the producer consumer problem; and (iii) provide additional practice in C programming and understanding specifications.

Preparation:

1. Complete Assignment 3 and/or review the provided solution to Assignment 3.
2. Clone your assignment repository:
   `https://git.cs.dal.ca/courses/2022-summer/csci-3120/assignment-4/????.git`
   where ???? is your CSID.
3. Copy either your own solution or the provided solution as a starting point for Assignment 4 because Assignment 4 builds on Assignment 3. A solution will be accessible on July 8 from Brightspace.
4. **Important**: If you are using CLion with CMake you will need to ensure your CMakeLists.txt file has the additional statements as provided in Assignment 3.

The repository has the same structure and caveats as Assignment 1, Assignment 2, and Assignment 3.

Background[1]

Adding and removing transactions and searching for the *nonce* when mining blocks are all computationally intensive activities that prevent the miner from receiving new transactions and blocks. I.e., if the miner is very busy, it will become unresponsive causing other hosts in the network to wait until it can receive their broadcasts. This is not optimal. Ideally, the miner should be able to receive new transactions and blocks at any time and process them later, when it becomes free.

Problem Statement

Extend your program from Assignment 3 (or the provided solution) to make the **main** thread of your miner spawn a **reader** thread that is responsible for reading all the input. The **reader** thread should then insert the input into a queue (buffer). The **main** thread will be responsible for dequeuing the input and processing it, just like in Assignment 3. This is essentially the Unbounded Buffer Producer-Consumer problem where the **reader** thread is the *producer*, and the **main** thread is the *consumer.* Your program must be compiled to an executable called **`miner`**. The program will read in five events, which are the same as in Assignment 3. Please see the Problem Statements in Assignment 1, 2 and 3 for a description of the events.

The **reader** thread should consist of a loop that reads events from **stdin** and enqueues them onto an unbounded queue. The queue can never fill up, so a linked-list implementation is recommended.

The **main** thread will use the same main loop as in Assignment 3, except that instead of reading from **stdin** it should dequeue events from the queue fed by the **reader** thread and process the events. Please note that if the queue is empty, the **main** thread should block until more events are added by the **reader** thread.

[1] This background description assumes that you have read and understood the background of previous Assignments.

Input

Your program reads its input from **stdin**. The input consists of events described in previous assignments.

Processing

All events are processed in the same way as in Assignment 3. The only difference in processing is how the events are read in. A separate **reader** thread, which is spawned at the start of execution, should be the only thread reading from **stdin.** The **reader** should read in the events and insert them into a shared queue. The **reader** thread must never block. I.e., the queue is unbounded. If the **reader** thread receives the **end** event, it should leave its loop and end after it has inserted the event into the queue.

The **main** thread should dequeue events from the shared queue. If the queue is empty, the **main** thread should be blocked (suspended) until the queue is not empty. Once the **main** thread is resumed, it processes the event in the same manner as in Assignment 3.

Output

The output for this assignment is the same as that of Assignment 3, except that when the **reader** thread inserts an event into the queue, it should print out one of two messages.

- If the event is a **block** or **transaction**, it should print out

  `Received event` ***E*** `with ID` ***X***

  where ***E*** is either `BLK` or `TRX` and ***X*** is the identifier of the block or transaction.
- If the event **end**, **epoch**, or **mine**, it should print out

  `Received event` ***E***

  where ***E*** is either `END`, `EPOCH`, or `MINE`.

Example:

Input	Output
TRX 12358 Cat Dov 150000 20 TRX 35813 Eve Fin 150000 25 TRX 11235 Al Beth 150000 40 MINE 3 END	Received event TRX with ID 12358 Received event TRX with ID 35813 Received event TRX with ID 11235 Adding transaction: 12358 Cat Dov 150000 20 Adding transaction: 35813 Eve Fin 150000 25 Adding transaction: 11235 Al Beth 150000 40 Received event MINE Received event END 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

Note that the **Received** statements may be interleaved with the output in a different order for each run because they are being produced by a different thread.

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, 2, and 3.

Hints and Suggestions

- The queue is shared and is a critical section.
- The queue is unbounded and should be implemented with a linked list.
- The **main** thread should block (be suspended) if the queue is empty. It should **not** spin! You will need to use locks to protect critical sections in your program.
- Using pthread's condition variables is an easy way to block the consumer if the queue is empty.
- To make things simpler, the solution places the reader code into a separate file (`reader.c`) and uses a separate file (`event_q.c`) for the multithreaded queue implementation.

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 solves the producer-consumer problem in an efficient and correct manner.	Solution solves the producer-consumer problem correctly.	Solution mostly solves the producer-consumer problem. Some synchronization cases are not handled	Solution has major issues but does follow the template for a producer-consumer solution.	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	