CS3103: Operating Systems (Spring 2019)

Programming Assignment 3
P3: A Simulator for FIFO Queue with Progressive Flow

1 Goals

This assignment is designed to help you:

1. get familiar with multi-threaded programming with Pthread.
2. get familiar with critical section with Pthread mutex.
3. get familiar with synchronization with Pthread semaphore. Note that **any semaphore should be given an initial value, or otherwise it is logically wrong!**

You are required to implement your solution in C++ (other languages are not allowed). Your work will be tested on the Linux server cs3103-01.cs.cityu.edu.hk.

A good tutorial on pthread could be found at https://computing.llnl.gov/tutorials/pthreads/.

2 Requirements

2.1 Background

This assignment is built above your solution to Assignment 2. To make your life easy, you should re-use your code in Assignment 2.

In this assignment, we add a new type of flow, called progressive flow (pflow for short), to the system described in Assignment 2.

The behavior of pflow is as follows: **whenever the queue is emptied by the server**[1], it generates a random number of tokens, uniformly distributed in $[1, 5]$, and inserts the tokens to the queue. Otherwise, it is blocked if the queue is not empty.

The behavior of flow, queue, and server are the same as in Assignment 2. Note that the sequence number of each token should be unique and in an increasing order in the system. Same as in Assignment 2, the simulation stops after MaxToken number of tokens have been served, where MaxToken is an input parameter.

[1]For simplicity, we ignore the situation that pflow runs before flow has injected some tokens in the queue.

2.2 Implementation Requirement

Your program, named by PFIFO, should consist of four threads: the main thread, the flow thread, the pflow thread, and the server thread. The main thread is responsible for creating the other three threads, and then waits for them to return. The flow thread simulates the behavior of the flow. The pflow thread simulates the behavior of the pflow. The server thread simulates the behavior of the server.

Same as in Assignment 2, the queue is implemented as a data structure, rather than a thread. You need to implement **your own queue data structure and its related functions**, which means that **you cannot use the queue data structure provided by the C/C++ library**.

Since flow, pflow, and server all need to access the queue data structure, the code segment for accessing the queue is a critical section and must be protected with the pthread mutex lock.

Since the sequence number of each token should be unique and in an increasing order in the system, you need to introduce a global variable currentSeq to tell the current sequence number to flow and pflow whenever they are active. There will be a race condition on accessing currentSeq between flow and pflow, and hence the code for accessing currentSeq should be protected with a pthread mutex lock.

Note that there is a synchronization issue between the pflow and the server: When the server empties the queue, it must notify the pflow to insert tokens into the queue. However, if the server empties the queue **again** before the pflow responds to the last notification, the server **must not** repeat the notification. You need to use semaphore to fulfill such synchronization.

Same as in Assignment 2, your code **must** print out useful information, and the output **must** be formatted as the following table:

```
Flow                                      Queue           Server
Token added   Latest sequence number   Current Length   Token fetched   Total Token fetched


aaa(flow)     bbb                      ccc
                                       zzz              xxx             yyy
```

where the header of the table is printed by the main thread before creating other threads. The numbers aaa, bbb, and ccc are printed by the flow (or pflow) when it wakes up. You must label after aaa with either (flow) or (pflow) to show the creator of the tokens. The numbers xxx, yyy, and zzz are printed by the server when it wakes up.

In addition, the number aaa is **not** an accumulative number but the number of tokens at the current round of flow/pflow wake-up; the number bbb is the highest sequence number at the current round of flow/pflow wake-up; the number ccc is the queue length after tokens generated at the current round of flow/pflow wake-up have been added into the queue. The number xxx is **not** an accumulative number but the number of tokens that will be fetched by the server at the current round of its wake-up; the number yyy is the total number of tokens that have been fetched by the server; the number zzz is the current queue length after tokens have been fetched by the server at the current round of server wake-up.

Note that at the end of the program, the total number of tokens that have been served is equal to the total number of tokens that have been fetched by the server and the total number of tokens that have been dropped by the queue. As such, **at the end of the program**, you must print out the total number of tokens that have been fetched by the server thread, the total number of tokens that have been generated by the flow thread, the total number of tokens that have been generated by the pflow thread, and the total number of tokens that have been dropped by the queue. The output should be:

```
The total number of tokens that have been fetched by the server is eee.
The total number of tokens that have been generated by the flow is fff.
The total number of tokens that have been generated by the pflow is ggg.
The total number of tokens that have been dropped by the queue is hhh.
```

where `eee`, `fff`, `ggg`, and `hhh` indicate the corresponding numbers.

Remark 1: After the pflow is woken up by the server, there might be a competition between the flow and the pflow for accessing the queue. If the flow wins the competition and inputs the tokens before the pflow, the pflow will inject tokens into a non-empty queue. Nevertheless, this special case is acceptable.

Remark 2: Since the number of last-batch tokens fetched by the server before it stops is random, the sum of `eee` and `hhh` might be equal to or slightly larger than `MaxToken`. Both cases are acceptable. This relaxation is to ease your coding.

Remark 3: Refer to the flowchart for a baseline implementation.

2.3 Prompt for user input

Your `PFIFO` needs to accept two integer inputs, `MaxToken` as the first and `flowInterval` as the second, e.g.,

```
$ ./PFIFO 500 2
```

2.4 Code Quality

We cannot specify completely the coding style that we would like to see but it includes the following:

1. Proper decomposition of a program into subroutines (and multiple source code files when necessary)—A 500 line program as a single routine won't suffice.
2. Comment—judiciously, but not profusely. Comments also serve to help a marker, in addition to yourself. To further elaborate:
   (a) Your favorite quote from Star Wars or Douglas Adams' Hitch-hiker's Guide to the Galaxy does not count as comments. In fact, they simply count as anti-comments, and will result in a loss of marks.
   (b) Comment your code in English. It is the official language of this university.
3. Proper variable names—`leia` is not a good variable name, it never was and never will be.
4. Small number of global variables, if any. Most programs need a very small number of global variables, if any. (If you have a global variable named `temp`, think again.)
5. **The return values from all system calls and function calls listed in the assignment specification should be checked and all values should be dealt with appropriately.**

3 Marking

Your program will be tested on our CSLab Linux servers (cs3103-01, cs3103-02, cs3103-03).

You should tell TA how to compile and run your code in your `readme` text file.

TAs are not supposed to fix the bugs, either in your source code or in your `Makefile`.

If an executable file cannot be generated and running successfully on our Linux servers, it will be considered as unsuccessful.

Any program does not print the output as required in Section 2.2 will be considered as unsuccessful.

If the program can be run successfully and output is in the correct form, your program will be graded with the following marking scheme.

Table 1: Marking scheme.

Components	**Weight**
`Initialization of semaphore`	5%
`Correct thread create/join/exit`	15%
`Correctness of programming logic`	10%
`Correct use of semaphore`	20%
`Synchronization with correct mutex`	15%
`Correctness of printed summary`	10%
`Queue structure & related functions`	10%
`Error handling`	10%
`Programming style and in-program comments`	5%

4 Submission

1. This assignment is to be done individually or by a group of two students. You are encouraged to discuss the high-level design of your solution with your classmates but you must implement the program on your own. Academic dishonesty such as copying another student's work or allowing another student to copy your work, is regarded as a serious academic offence.

2. Each submission consists of three files: a source program file (.cpp file), a readme (.txt) file telling us how to compile your code, a text file containing the possible outputs produced by your program (.txt file), and a `Makefile` if applicable.

3. Use your student ID(s) to name your submitted files, such as 5xxxxxxx.cpp, 5xxxxxxx-readme.txt, 5xxxxxxx.txt for individual submission, or 5xxxxxxx_5yyyyyyy.cpp, 5xxxxxxx_5yyyyyyy-readme.txt, 5xxxxxxx_5yyyyyyyy.txt for group submission. Only ONE submission is required for each group.

4. Submit the files to Canvas.

5. The deadline is 10:00 am, 11-APR. No late submission will be accepted.

Question?
Contact Mr LIU, Dawei at `daweiliu2-c@my.cityu.edu.hk` or your course lecturer.

The End
