Multi-Threaded Synchronized Queue Assignment

Due: March 14, 2025 at 23:59

1. Infrastructure Description

Welcome to the next OS assignment where we will build Multi-Threaded Synchronized Queue!

This is the only programming assignment that is independent of the project. Thus, we have provided you a new starter code. You will use the **C programming language** in your implementation, since most of the practical operating systems kernels are written in C or a mix of C and C++ (e.g., including Windows, Linux, MacOS, and many others).

The assignment is presented from a Linux point of view using a server like Mimi. Our grading infrastructure will pull your code from GitLab automatically and will run the unit tests for this assignment every day on the Mimi server. The autograder may not run for the first few days as we take all the administrative steps to set it up. (And because it will still be running for A2!)

To get started, once again you should synchronize your fork with ours. If you are reading this description, you've probably already done so.

1.1 Starter files description:

We provided you with a starter code for a multi-threaded queue interface, which you will complete in this assignment. Once you sync your fork with ours, the starter code will be in `multithread/starter-code`. Whether or not you want to use the starter code, run `mkdir -p multithread/src` to create the src folder where you will write your code. **You must place your work in the multithread/src folder, or our autograder will not be able to find it.** If you want to use the starter code, now run

```
cp -a multithread/starter-code/* multithread/src/
```

This will copy the starter code, Makefile, and style configuration into your new src folder. **Do not forget the -a flag** or else the style configuration might not be copied. The mimi shell will ignore it otherwise since the name starts with a dot. **Warning:** we have slightly modified the .indent.pro file from the project for this assignment.

Starter Files Overview

Navigate to the src folder with cd and take a moment to get familiar with the code.

- `main.c:` This program demonstrates a **producer-consumer model** using the queue. It initializes a queue, creates producer and consumer threads, and ensures that items are added and removed correctly. It does not use synchronization primitives – that will be the job of your queue.
- `queue.c`: Includes function headers of a **thread-safe queue interface**. This file is your most important working place.
- `test_queue.c`: A testing program that evaluates the correctness of the queue implementation. The provided tests check:

  - The queue maintains correct ordering.
  - The behavior of enqueue and dequeue operations under different conditions.
  - The queue handles multiple producer-consumer threads correctly.
  - The queue performs as expected under **high load**.
- `Makefile:` Automates the compilation process. Running `make` will generate two executables:
  - `main` → Runs the producer-consumer demonstration.
  - `test_queue` → Executes the test suite.
- The Makefile also has a `make style` target as in the project. **You must run this make target before submitting**.

Compiling the Code

You can compile this code using simple Linux commands.

- Use the following command to compile: `make`
- Re-compiling your shell after making modifications: `make clean; make`
- Running the automatic code formatter: `make style`
  - This command will leave behind lots of files with names like "`queue.c~`". These are backups of your code before it was formatted. We have configured the git repository to ignore these backup files, but you can also safely delete them with `rm *~`. Do not run this command without the ~ at the end -- it will delete all of your code!
- *Note: The starter code compiles and runs on Mimi and on our server. If you'd like to run the code in your own Linux virtual machine, you may need to install build essentials to be able to compile C code:* `sudo apt-get install build-essential`

Your code will be able to compile in any linux environment with a C compiler so long as you do not hardcode assumptions about the machine into your code. For example, write `sizeof(int)` rather than assuming that this value is 4.

1.2 Your tasks:

Your task is to implement a **multi-threaded queue** in C using **POSIX threads (`pthread`) and semaphores (`sem_t`)**. The assignment consists of the following tasks:

1.2.1 Implement a Synchronized Queue (`queue.c`)

You must implement a queue that supports **concurrent enqueue and dequeue operations** while ensuring **thread safety**. The queue should:

- Use **mutex lock(s) (`pthread_mutex_t`)** to prevent race conditions.
- Use **semaphore(s) (`sem_t`)** to manage available items in the queue and synchronize access to it.
- Provide the following functions:
  - `make_queue()`: Initializes and returns a new queue.
  - `enqueue(queue*, void*)`: Adds an item to the queue.

- `dequeue(queue*)`: Removes and returns an item (blocks if the queue is empty).
- `destroy_queue(queue*)`: Cleans up allocated memory and releases resources.

You should ensure that your implementation:

- Maintains **FIFO order** when enqueuing and dequeuing elements.
- Properly handles an **empty queue** (blocking if necessary).
- Works correctly under **high-load conditions**.
- Correctly synchronizes when multiple threads are used.

2. TESTCASES

We have provided you with **10 test cases** in `test_queue.c` to validate your implementation. Before submitting your assignment, ensure that your implementation passes all test cases.

Running the Tests

To run the test suite, use: `./test_queue`

This will execute multiple automated tests on your queue implementation. The program will print **[PASS]** or **[FAIL]** for each test, helping you identify issues.

If a test case segfaults, the entire tester is aborted. Therefore, **memory errors are highly penalized** by this assignment's grader. You should carefully test your code with Valgrind to be sure that there are no memory errors. This is a much lighter assignment than A2 but you have a similar amount of time to complete it, so you have plenty of time to ensure that your code is free of memory errors.

3. WHAT TO HAND IN

The assignment is **due on March 14, 2025 at 23:59, no extensions.**
Your final grade will be determined by running the code in the GitLab repository that is crawled by our grading infrastructure. We will use the most recent commit that happened before the deadline, on the main branch of your fork.
In addition to the code, please include a README mentioning the author name(s) and McGill ID(s), any comments the author(s) would like the TA to see, and mention whether the code uses the starter code provided by the OS team or not. (This README is not very important if you have no comments to make.)
The project must compile on our server by running `make clean; make`. You should confirm this when the autograder runs **before** the deadline.

Note: You must submit your own work. You can speak to each other for help but copied code will be handled as to McGill regulations. Submissions are automatically checked via plagiarism detection tools.

4. HOW IT WILL BE GRADED

Your program must compile and run on our server to be graded. If the code does not compile/run in our grading infrastructure, you will receive **0 points** for the entire assignment. If you think your code is correct and there is an issue with the grading infrastructure, contact Mohamad Danesh at mo.danesh@mcgill.ca.

Your assignment is graded out of 10 points. You were provided 10 testcases in test_queue.c, with expected outputs. If your code matches the expected output, you will receive 1 point for each testcase. You will receive 0 points for each testcase where your output does not match the expected output.

If your code segfaults, you will receive a 0 on the assignment. Use the extra time to ensure your code is completely free of memory errors.

Your code needs to follow a consistent programming style. To ensure this, a configuration file for the `indent` tool has been provided, and it is installed on all the mimi machines. A Makefile target (`make style`) is also included to format your code automatically using `indent`. Before submitting, you must run `make style` to format your code. Failure to do so may result in a 1-point deduction (out of 10) if running `indent` reveals any discrepancies between the submitted code and the formatted output.

If you do not use the starter code, or if you add additional source files, you will have to modify the Makefile to correctly compile your source files. You should also fix the style target so that it correctly styles your code. Failure to do so will result in the autograder being unable to run your code, or possibly in losing the style point.

We will use **your Makefile** and **our .indent.pro**. You may change your Makefile however you wish to fit your project, but your programming style **must** match the indent configuration that we gave you.

The TA will look at your source code only if the program runs (correctly or not). The TA looks at your code to verify that you implemented the requirement as requested. Specifically:

- **Hardcoded solutions will receive 0 points for the hardcoded testcase**, even if the output is correct.
- **You must write this assignment in the C Programming language**, otherwise the assignment will receive 0 points.