1 Introduction

In this project, we will solve the producers-consumers problem using monitors. As part of the assignment, you will get to create a thread-safe version of your linked-list code!

You will need to have the generic shared library version of your linked list code for this project.

2 Specification

2.1 Producers and Consumers using Monitors (100 points)

We will simulate a size-bounded queue (implemented as a doubly-linked list with a limit on maximum size) being shared by multiple producers and multiple consumers running simultaneously. Suppose we have $m$ producers and $n$ consumers (where $m$ and $n$ are supplied as command line arguments). The initial main process creates a queue data structure. Make this queue global so all threads can access it. Then the main process will create $m$ producer threads and $n$ consumer threads.

We will fix the number of items each producer produces (as a command line argument). Then we will keep track of how many items each consumer actually consumed. At the end of the program, we print out the total number of items produced versus the total number of items consumed. These two numbers should be the same unless the program has race conditions.

The items are a structure that contain an unique id (basically number the items 1, 2, 3, ... as they are entered in the queue) and also contains the index of the producer that created it.

2.1.1 Approach

- A template program has been provided for you to use as a starting point. The program is in the examples repository at `lab/projects/producers-consumers` and the main file is called `pc.c`. The directory also contains a relevant `Makefile` that you should use.

- First, you will need to create a `monitor` version of your doubly linked list to use in this project. An additional feature we will add to it is the ability to set the maximum size of the list in the constructor. You will be using the PThread mutexes and PThread condition variables to implement the monitor version of your doubly linked list. Relevant man pages are `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_cond_init`, `pthread_cond_wait`, `pthread_cond_signal`. 
• Please create a separate version that is a monitor and check into your subversion tree.

• Then the producer/consumer threads test program will use the monitor version of the doubly linked list to run the simulation. To test your monitor version, we will have the producers randomly insert new items at the front or back of the queue. Similarly the consumers randomly remove items from the front or back of the queue.

2.1.2 Notes on testing

• You should not modify the test program `pc.c`. Adjust your list implementation if needed. Your list implementation will need to be fully generic. You will need to add an additional function to your list class that allows the simulation to be stopped after the producers are done. This function is called `finishUp`.

• When testing your program limit the number of threads to less than ten and number of items produced per producer to less than ten thousand to avoid overloading `onyx`. At home you can do whatever you want :-)

• Run the program several times for the same input arguments. Verify that the results do not vary.

• Comment out the output statements. See if that changes the results for the number of items consumed versus produced.

• Use the testing script provided in the assignment folder.

2.2 Multiple Queues (10 points)

Required for graduate students. Optional for undergraduates.

Create another version of the testing program `pc.c` and call it `pc-mq.c` and adjust the Makefile accordingly.

Add another command line argument that allows the user to specify the number of queues (with the same size limit on each). The number of queues should be the first command line argument.

Usage: `pc-mq <#queues> <poolsize> <#items/producer> <#producers> <#consumers> <sleep interval(microsecs)>

Now modify your the testing code to create multiple queues and have the producers/consumers access the queues in round-robin fashion, which is explained below.

Suppose we have $k$ queues. Then if the $i$th queue is full, the producer moves on to the $(i+1) \mod k$th queue. Similarly if the $i$th queue is empty, the consumer moves on to the $(i + 1) \mod k$th queue.

Testing would be the same as before. You could also informally test if multiple queues allows the producers and consumers to get their work done faster although that would depend on the mix of the work being simulated.
3 Required Files for Submission

Include a README file in the usual format. Please submit the files below. Your final executable should be in your top level folder.

- README, Item.c Item.h Makefile pc.c test.sh
- Your linked list files

4 Submitting the Project

You are using Git for this project thus you will submit code just like you would in industry.

- git add [Your files]
- git commit -m "Project p3 complete"
- git push origin master