CSCE 313

QUIZ 4

Fall 2021

Total Points 100

Q1 [10 pts]: How would you change the Producer for an unbounded buffer, given below into an unbounded buffer, where there is no limit to its capacity.

```
Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}
```

However, make sure that the Consumer(s) must still wait for an item to be present in the buffer before they can consume. In other words, the producers never stall, only the consumers stall in the absence of items.

```
Producer(item) {
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}
```

CSCE 313

QUIZ 4

Fall 2021

Q2 [20 pts]: Using only condition variables and mutexes, write a BoundedBuffer that does not let a consumer consume when the buffer is below 10% of its capacity. Hint: Use a condition variable that is waited on by the pop() function and signaled by the producer. This condition variable will eliminate a semaphore.

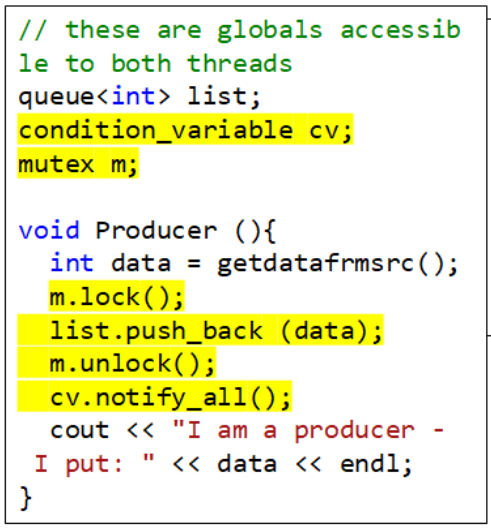
```
mutex m;
int cap;                    // maximum allowed items in buffer (capacity)

void push(auto data ){
        m.lock();
        list.push_back(data);
        m.unlock();
        cv.notify_all();
}

auto pop(){
        unique_lock<mutex> ul(m);
        cv.wait(ul, [] { return list.size() > 0.1*cap });
        auto data = list.front();
        list.pop();
        ul.unlock();
        return item;
}
```

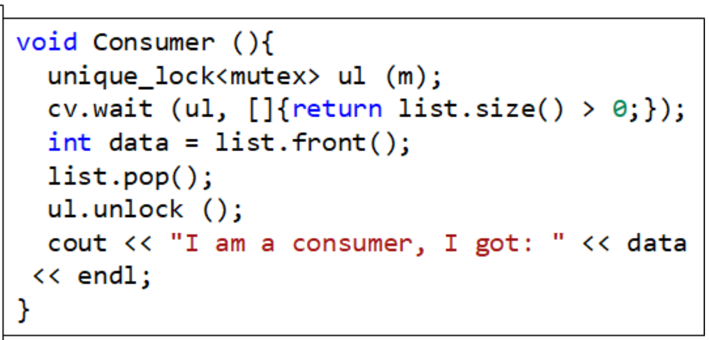
CSCE 313 **QUIZ 4** Fall 2021

Q3 [10 pts]: In the following producer-consumer solution, why must we use the same mutex variable for both the producer and consumer? Why cannot they use separate mutex variables? Please explain your answer.



```
// these are globals accessib
le to both threads
queue<int> list;
condition_variable cv;
mutex m;

void Producer (){
  int data = getdatafrmsrc();
  m.lock();
  list.push_back (data);
  m.unlock();
  cv.notify_all();
  cout << "I am a producer -
 I put: " << data << endl;
}
```



```
void Consumer (){
  unique_lock<mutex> ul (m);
  cv.wait (ul, []{return list.size() > 0;});
  int data = list.front();
  list.pop();
  ul.unlock ();
  cout << "I am a consumer, I got: " << data
 << endl;
}
```

Because if they share the same mutex, this will allow for only 1 thread being allowed to lock the mutex. If threads had their own mutex it would defeat the purpose of having a mutex in the first place because they could interrupt each other because any thread could lock their own mutex whenever they want to.

In this specific case, if the consumer had its own mutex, then the consumer and producer could interrupt each other and corrupt the integrity of the program/instructions.

CSCE 313

QUIZ 4

Fall 2021

Q4 [10 pts]: The following is a thread function that is being run from 5 threads.

```
int  x=0;
void ThreadFunc(int &x){
    x += 2;
}
```

Can you have x=4 after all the threads finish running? If yes, show the schedule of 5 threads using assembly language that would lead to x=4.

You must write functional assembly code as shown in class lectures - pseudo code is not OK

Never written real assembly before but I imagine it is similar to hack assembly from 312 and this is what it looks like from what I gathered from the lecture.

```
load r,x
add r,r,2
load r,x
add r, r, 2
store x,r                    // here it is incremented by 2
store x,r
load r,x
add r,r,2
load r,x
add r,r,2
load r,x
add r,r,2
store x,r                    // here it is incremented by 2
store x,r
store x,r
```

Q5 [10 pts]: If we run 5 instances of ThreadA() and 1 instance of ThreadB(), what can be the maximum number of threads active simultaneously in the Critical Section? The mutex is initially unlocked. Note that ThreadB() is buggy and mistakenly unlocks the mutex first instead of locking first. Explain your answer.

```
ThreadA(){
      mutex.P()
      /* Start Critical Section */
      …….
      /* End Critical Section */
      mutex.V();

}
```

```
ThreadB(){
   mutex.V()
   /* Start Critical Section */
      …….
   /* End Critical Section */
   mutex.P();

}
```

Initially it starts unlocked, but one of ThreadA's lock it and start their critical section. None of the other ThreadA's can start their critical section until it is unlocked.
Then ThreadB, unlocks, and starts its critical section.
Then now that it is unlocked, it allows one more ThreadA to start it's critical section and lock it again.

Only a maximum of 3 threads can be in the critical section at the same time. This is because ThreadB starts its critical section by unlocking rather than locking, which allows itself to enter the critical section, and allows another thread to enter the critical section.

Q6 [20 pts]: Consider a multithreaded web crawling and indexing program, which needs to first download a web page and then parse the HTML of that page to extract links and other useful information from it. The problem is both downloading a page and parsing it can be very slow depending on the content. Your goal is to make both these components as fast as possible. First, to speed up downloading, you delegate the task to **m** downloader threads, each with only a portion of the page to download. (Note that this is quite common in real life and a typical web browser does this all the time as long as the server supports this feature. Usually it is done through opening multiple TCP connections with the server and downloading equal sized chunks through each connection). The M chunks are downloaded to a single page buffer. Once all the chunks are downloaded into the buffer, you can then start parsing it. However, since you want to speed up parsing as well, you now use **n** parsing threads who again can parse the page independently, and together they take much less time.

By now, you probably see that **M** download threads are acting as Producers and **N** parser threads as Consumers. Additionally, note that the both downloader and parser threads come from a pool of M Producer threads and N Consumer threads where M>m and N>n. Out of many of these, you have to let exactly **m** Producer threads carry out the download and then exactly **n** consumer threads parse, and then the whole cycle will repeat. IOW, in each cycle, you are employing **m** out of **M** Producer worker threads (who are all eagerly waiting) to download the page simultaneously, and then **n** out of **N** Consumers are concurrently parsing the downloaded page. You cannot assume m=M or n=N. Assume that you can do a function call `download(URL)` to download the page and parse (chunk) to parse a chunk of the page. No need to be any more specific/concrete than that. The main thing of interest is the Producer-Consumer relation.

Look at the given program **1PNC.cpp** that works for 1 Producer and n Consumer threads. Be sure to run the program first to see how it behaves. You need to extend the program such that it works for m-producers instead of just 1. Add necessary semaphores to the program. However, you will lose points if you add unnecessary Semaphores or Mutexes. Use **Semaphore.h** as the Semaphore class definition. To keep things simple, declare the mutexes as instances of the Semaphore classes. Test your program to make sure that it is correct. In your submission directory, include a file called **Q6.cpp** that contains the correct program.

CSCE 313

QUIZ 4

Fall 2021

Q7 [20 pts]: There are 3 sets of threads A, B, C. First 1 instance of A has to run, then 2 instances of B and then 1 instance of C, then the cycle repeats. This emulates a chain of producer-consumer relationship that we learned in class, but between multiple pairs of threads. Write code to run these sets of threads.

Assumptions and Instructions: There are 100s of A, B, C threads trying to run. Write only the thread functions with proper wait and signal operation in terms of semaphores. You can use the necessary number of semaphores as long as you declare them in global and initialize them properly with correct values. The actual operations done by A, B and C does not really matter. Submit a separate C++ file called **Q7.cpp** that includes the solution.