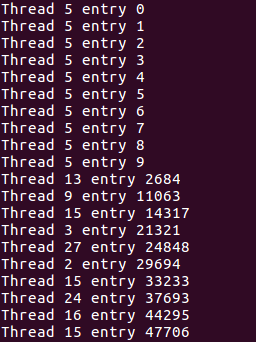
**CS2106 Operating Systems**

**Lab 5 – Semaphores**

|  |  |
| --- | --- |
| **Name 1:**  **Joshua Lee Kai Sheng** | **Student Number 1:**  **A0161868R** |
| **Name 2:**  **Ryan Tan Wei Keat** | **Student Number 2:**  **A0161551M** |
| **Name 3:**  **Ng Jun Wei** | **Student Number 3:**  **A0155156E** |

**Question 1 (1 mark)**

**This program does not run correctly.**



**The thread numbers are considered correct because all thread numbers are printed (eventually) despite being out of order. The entry numbers become wrong after the first few output. For example, the first entry number thread 13 output is 2684, which is wrong (should be 0).**

**Question 2. (3 marks)**

This program may not operate correctly in a multithreaded because

**i) Race conditions might happen**

**An example is when thread A is executing deq() and thread B is executing enq(). Assume that front and back are the same.**

**When thread A is executing deq(), before it could execute memcpy(), it is preempted. The scheduler picks thread B and runs it. Thread B, when executing enq(), successfully executes memcpy(). When thread A resumes, the content in buffer->data[buffer->front] has already been overwritten by thread B, hence resulting in wrong results getting printed.**

**ii) The circular buffer slots are limited**

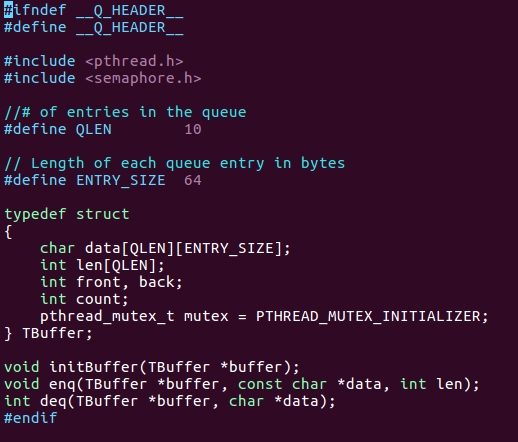
**Even if there are no race conditions, data might be dropped due to a full buffer. This could happen in the following example:**

**If too many senderThreads get executed consecutively, the buffer will become full. Hence, some senderThreads carrying data, when executing enq(), will satisfy the (buffer->count >= QLEN) condition and hence return without actually writing to the buffer. As a result, the data carried by these senderThreads will be lost.**

**Question 3. (4 marks)**

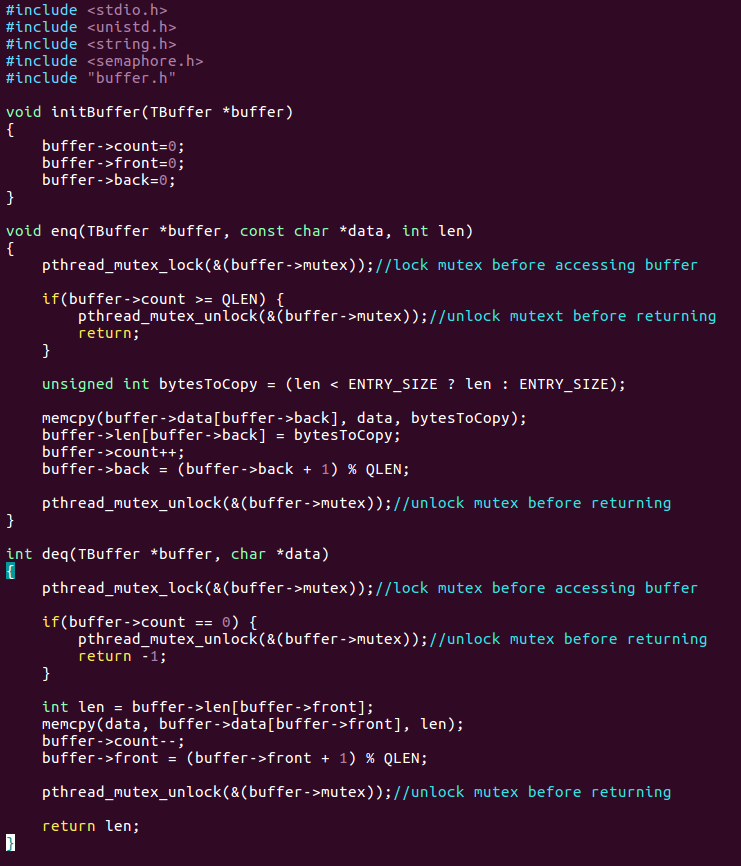
My code is shown below:

**For buffer.h,**



**Initialise a mutex variable inside the TBuffer structure to ensure data encapsulation**

**For buffer.cpp,**



**Inside enq() and deq(), lock the mutex before accessing the buffer and unlock the mutex before exiting. This is to prevent race conditions on the buffer among the threads through mutual exclusion.**

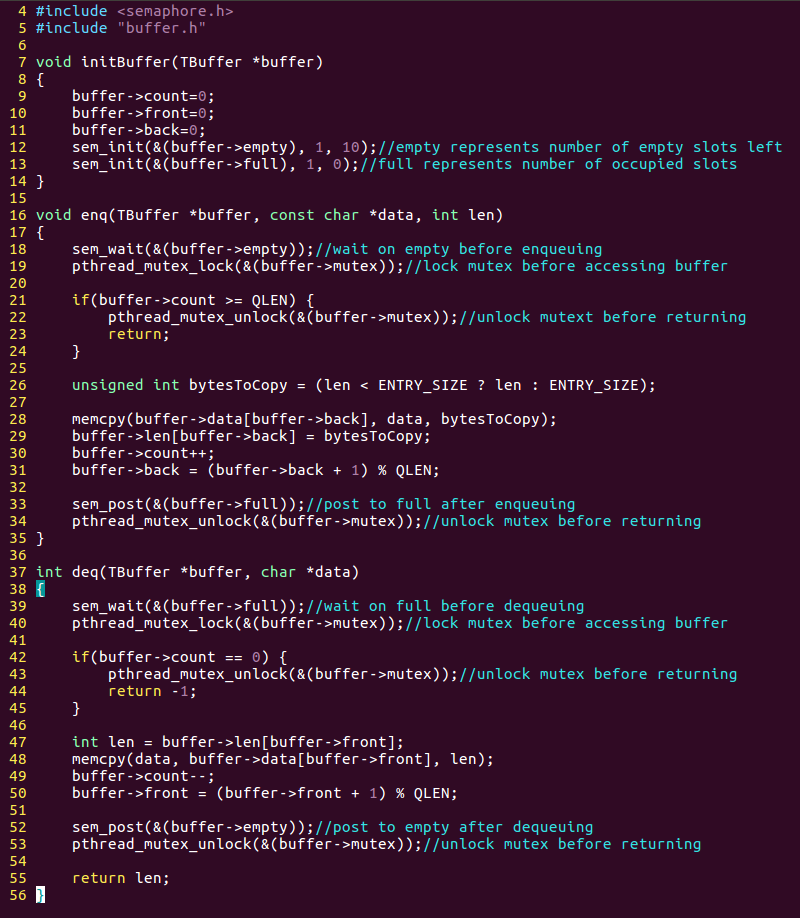
**Question 4. (1 mark)**

**The answer is not correct.**

**Question 5. (3 marks)**

**The problem in Q4 is that the circular buffer slots are limited and the senderThreads are not regulated and coordinated. When enqueuing, if a senderThread finds the buffer count to be equal to or larger than QLEN, it will simply return causing its data to be lost (not written to buffer). By right, the senderThread should wait until a slot is available in the buffer for it to enqueue/ copy over its data. However, this is not done by the original code.**

**The sem\_wait and sem\_post calls fixed the problem because it imposes coordination on the senderThreads.**

****

**In our modified code for buffer.cpp, there are 2 semaphores created, i.e. empty and full. Semaphore “empty” represents the count of the empty slots left in the buffer, whereas, semaphore “full” represents the count of occupied slots in the buffer.**

**With such modifications, when a senderThread executes enq(), it will first wait for the “empty” semaphore. If there is no more empty slots left in the buffer, the senderThread will “wait” until there is some empty slots after the serialSendThread executes deq() and frees up space in the buffer. As a result, the senderThread will not simply return from enq() when there is no empty buffer slots left. Hence, this makes sure that the data carried by senderThread is written to the buffer all the time.**

**Question 6. (4 marks)**

Changes to my web server are shown below with explanation:

**Inside buffer.h**

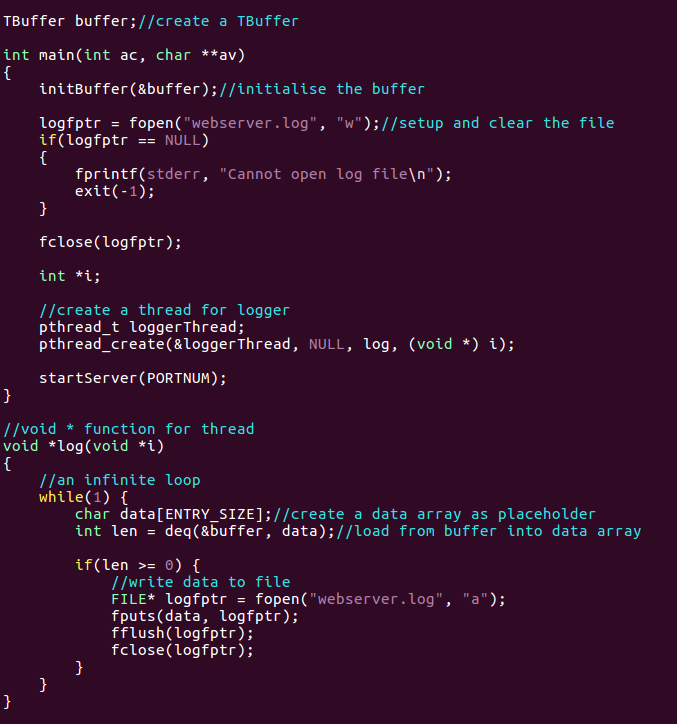


**Change the ENTRY\_SIZE to 1024 to fit LOG\_BUFFER\_LEN**

**Inside web server code (lab3p3.cpp):**

****

**Include buffer.h**

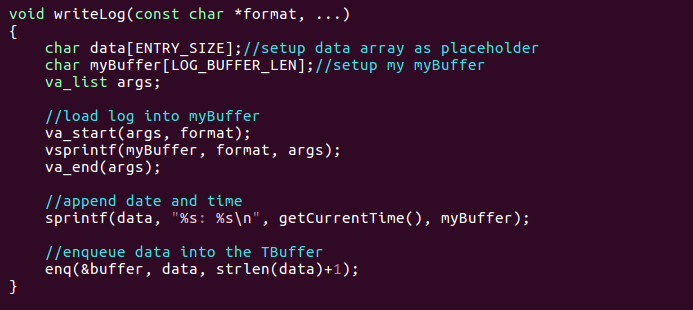


**Create a global variable buffer of type TBuffer.**

**In the main(), initialise the TBuffer.**

**Inside \*log(), create an infinite loop that constantly tries to dequeue from the TBuffer.**

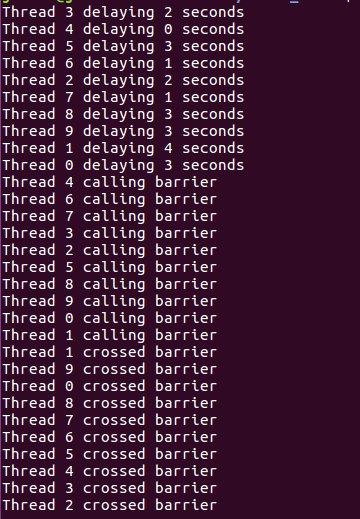
**If the data dequeued has non-zero length, which means there is log entry to be written to the log file, the data will be appended to webserver.log.**



**Inside writeLog(), create a data array to serve as a placeholder. Prepare the log entry and copy it into data array. Enqueue the data array into the TBuffer.**

**Question 7.** **(1 mark)**

This is what’s happening in testbarrier.cpp:



**In testbarrier.cpp, 10 threads are created and each of them are delayed with a random number of seconds as shown by the different number of seconds delayed in the output. Hence, each thread will reach the barrier at different times. This is shown by the threads printing “calling barrier” at different times.**

**Before all threads have arrived at the barrier, threads that reach the barrier earlier are blocked at the barrier. Once the last thread has reached the barrier, all the threads cross the barrier together, as shown by all the threads printing “crossed barrier” at the same time.**

**Question 8. (3 marks)**

This is how reachBarrier works:

**Whenever a thread reaches the barrier and calls reachBarrier(), it will increment the numReach of the barrier by 1. numReach represents the number of threads that have already reached the barrier.**

**If numReach is smaller than numProcesses (number of threads that have reached is fewer than total number of processes expected), the thread will wait on the semaphore allocated to it in the semArray. This thread will block as the semaphores are all initialised to 0. This gives rise to the phenomenon where threads are blocked at the barrier.**

**When the last thread arrives at the barrier and calls reachBarrier(), it will increment numReach, making numReach equal to numProcesses. This signifies all processes have reached the barrier. This last thread will then post on the semaphore of the process with the biggest process number (10). This process will then unblock from the sem\_wait in the else case and proceed to posting to the semaphore of the process with the second biggest process number (9). The thread with the second biggest process number will similarly unblock from the sem\_wait in the else case and proceed to posting to the semaphore of the process with the third biggest process number (8).**

**This process will repeat until it eventually reaches the thread with process number 1. This chain of posting will unblock all the processes at the barrier. This gives rise to the phenomenon where all processes cross the barrier together.**

TOTAL: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / 20