EECS 678 Lab 09 Report

Author: Jace Kline 2881618

Questions:

1. Briefly describe the problem you are trying to solve and describe the design of your solution.

The problem to be solved is the producer-consumer problem. This problem entails managing an arbitrary number of producer and consumer threads, each of which accesses/updates a shared buffer. A correct solution requires that no value is repeated (produced twice) by the producers and no value is consumed twice by the consumers. My solution uses a semaphore approach to achieve the required functionality. There are 3 "locking" components in my solution. There is a binary mutex that protects the shared queue buffer. Any thread must acquire this before entering the critical section to update the queue contents, and this serves to ensure mutual exclusion and data consistency between all threads. The next component is the 'slotsToPut' semaphore. This semaphore is what the producer threads wait on, and it is initialized to value QUEUESIZE as defined in the .c file. Essentially, each producer will invoke a 'sem_wait()' call, which puts the thread to sleep and place the thread in the 'slotsToPut' semaphore queue. When a consumer thread calls the 'sem_post()' call on this semaphore, the next producer thread in the queue will be woken up, and will continue on to obtain the queue mutex to enter and complete its critical section. When the critical section is complete, the producer thread will unlock the queue mutex and then call a 'sem_post()' on the 'slotsToGet' semaphore, which in turn will wake up a consumer thread. The process outlined above is almost identical to that which occurs in the consumer threads, except for that the signaling semaphore used is called 'slotsToGet', and it is initialized to value 0.

2. Discuss why the busy-wait solution is inefficient for threads running on the same CPU, even though it produces the "desired behavior".

The busy-wait solution is inefficient because it requires the CPU to waste many valuable CPU cycles in threads where no useful computation is occurring. This means that a thread (or many consecutive) could busy-wait for its entire scheduled time slice on the CPU. This is very inefficient because it prevents other threads with work to do from running, wasting time and energy. This problem is exacerbated with more threads that all perform busy-waiting.

3. Why are you confident your solution is correct? You will need to argue from your narrated output as to why your solution is correct. Note, your output will likely not match the output listed here exactly. Two successive runs of your application will probably not match even vaguely, due to random variations in how threads are scheduled. However, you should be able to discuss each of the following points and discuss how your output supports your discussion of each:

    - Are each producer and consumer operating on a unique item? Is each item both produced and consumed?

    Yes. In all of the tests provided, all produced items are unique from the others. The same holds for all consumed items.

    - Do the producers produce the correct number of items and the consumers consume the correct number of items?

    Yes. The total number of produced and consumed items is equal over all test cases run. For example, if the producers produce values 0-29, then the consumers will, combined, consume values 0-29. An interesting observation to note is that, when there n number of producer threads, the produced values from each of the threads will occur n productions apart. This is because of the queue property of the semaphore. The implication is that the producers essentially "take turns", each producing value one at a time, until all items have been produced. The same observation can be made about the consumer threads in regards to consumption. This means that each thread (both consumers and producers) performs roughly the same amount of work, which is a desireable property for a multi-threaded application.

    - Does each thread exit appropriately?

    After tinkering with the 'pthread_join' loops at the end, I finally got all threads to exit properly. What was occurring was that in the case of a low number of producers and a high number of consumers, the remaining consumer threads (after all values were consumed) were getting stalled (indefinitely waiting) at the 'sem_wait()' statement because all the producers exited and therefore no signal was being sent to these remaining threads. To fix this issue, I added a 'sem_post()' call at the beginning of each iteration of the loop that called 'pthread_join()' on each consumer thread. In addition, in the consumer threads I had to add an additional conditional check to verify that all values weren't consumed before entering the critical section and trying to read a value. If this condition does not hold, the loop is broken and the thread exits.

    - Does your solution pass the above tests with the different numbers of producer and consumer threads in the Makefile tests?

Yes. After running all the tests from the Makefile, the above conditions hold for each test. This indicates that the solution I implemented is likely to hold in all other valid combinations of producers and consumers. In conclusion, I believe my solution is correct.