Mina Vu

Assignment 1 Report

CS515 Parallel Programming

Spring 2022

**Producer & Consumer Programs in C++ and Java Using Threads**

For assignment 1, all five of my programs are complete. The producers and consumers run currently as can be seen in the execution outputs. The items in the queue never exceed the buffer size, and the queue size is never negative. Through many executions these behaviors held true. All five programs run to completion indicating that the main thread started and created the producer and consumer threads, and these threads complete their tasks and join back with the main thread so that the programs conclude successfully.

The programs use global queues and other global variables that are shared between threads. This is apparent in the interleaving output where it can be observed that a producer may add a few items to the queue, increasing its size, and a consumer may remove a few items from the queue, decreasing the queue size. It can happen that the producers fill the queue quickly and must wait for more queue space. This is checked every time a producer thread wants to add to the queue. On the other hand, the consumers may have to wait for more items in the queue before consuming. This is also checked every time a consumer thread wants to remove from the queue. The producers and consumers signal to waiting threads after each addition or removal respectively.

When there are multiple producers and multiple consumers in one program, threads share the work load. Among producer threads, the items to be produced are divided up as evenly as possible between all producer threads with only the last producer picking up the remainder. Among consumer threads, it is a race condition to get the lock and condition variable to the critical section of execution. It is observed that the consumer threads distribute the workload relatively evenly with this race condition. This trait is seen when critical sections are kept short and without display statements.

The first issue I encountered was the sharing of the queue among threads. I implemented the C++ programs first and habit made me think of creating a local queue in main to pass back and forth between threads to execute their producer or consumer routines. This also applied with the number of consumers, the number of producers, and the consumer work distribution array. The resolution here was to make all these variables global because they are shared among threads. Global variables are not something I am used to when coding in C++, so I hesitated at first. My second issue was that the interleaving outputs were printing on top of each other. First, I decided to put print statements inside critical sections. Upon hearing again that we should not do this, I found out that the stream insertion operator in C++ does not guarantee atomic behavior. The fix for this was to create the string before outputting. This also solved another issue that I observed. With the print statements inside critical sections, I noticed from the outputs that the producers and consumers were not running as concurrently as should be. Multiple consumers were also not evenly grabbing items off the queue. A producer or consumer thread would put or get data from the queue in chucks.

I wrote the C++ programs before I wrote the Java programs. This was simply because writing C++ in the Linux environment was more intuitive to me. I really did not want to write Java code in Linux so I put that off for later. I used an IDE to write my Java programs and tested them before transferring them to Linux to test again. The verbose nature of Java discourages me from ever wanting to write Java programs in a programmer’s text editor. As for thread support from the two languages, I see pros and cons in both. In C++, it is easy to create threads and write different routines for them, but the programmer has to include three different libraries to implement parallel threads correctly. In Java, threads can be created without special imports; however, programmers must remember to start them otherwise those threads are dormant. Another important distinction I saw was that consumer threads in Java distributed workload a lot more evenly than in C++.