CS 131 Homework 3 Report

Ryan Riahi

February 2021

1 Introduction

In this assignment I tested various Java synchronization methods. I used a simple prototype that manages a data structure that represents an array of longs. The array is initalized with all values being set to zero. The main method of the program, when executed, starts up a specified number of threads and has them perform state transitions called swaps on the array. A swap simply adds one to a certain index of the array and subtracts one from another index. Thus, at the end of the program, if nothing goes wrong, all the values of the array should remain at zero. These swaps are performed based on the various state implementations: Null, SynchronizedState, UnsynchronizedState, and AcmeSafeState.

2 Testing Platform

These tests were run on lnxsrv06.seas.ucla.edu and lnxsrv11.seas.ucla.edu. The former uses "Intel(R) Xeon(R) CPU E5620 @ 2.40GHz" and has 16 processors while the latter uses "Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz" and has 4 processors. This suggests that our programs will run faster on lnxsrv06 because of its higher clock speed and larger number of processors. The more processors, the more threads we can run concurrently without the operating system having to schedule other waiting programs to run while our multi threaded program is running. On both servers, we are running java 15.0.2. Lnxsrv11's operating system is "Red Hat Enterprise Linux 8.2", whereas lnxsrv06's operating system is "Red Hat Enterprise Linux 7". The difference in operating systems could potentially change our measurements because of variances in the OS's schedulers; however, this shouldn't make too much of a difference. At the time my tests were run, lnxsrv06 had a lot more free memory available and a lot less actively cached memory. This allows for better performance of threads because more free memory and more cache availability means more cached memory for our threads to use for faster memory access. Overall, the two servers are not too different.

3 AcmeSafeState Impl.

My AcmeSafeState class was implemented using the AtomcLongArray class, which belongs to the java.util.concurrent.atomic package. The AtomicLongArray is a long array data structure that is updated atomically. This means that the data structure ensures that updates are either performed entirely without interruption or are counted as not having been executed. Using AtomicLongArray ensures that this class is Data-Race Free (DRF) because no two threads are allowed to read/write to the same location in the array at the same time. If one thread is in the middle of performing an atomic operation, then the other must wait until that operation is completed in its entirety. This method of preventing race conditions varies from the synchronised keyword in Java in that we do not simply place locks around our entire swap method. Java synchronization is based on the Java Memory Model (JMM) which defines how applications can avoid race conditions when dealing with threads. The JMM is unique to Java in that it gives Java built in support for multi threaded programs. This built in support comes in the form of synchronized methods and the volatile keyword.

Figure 1: threads vs total time for array size 5

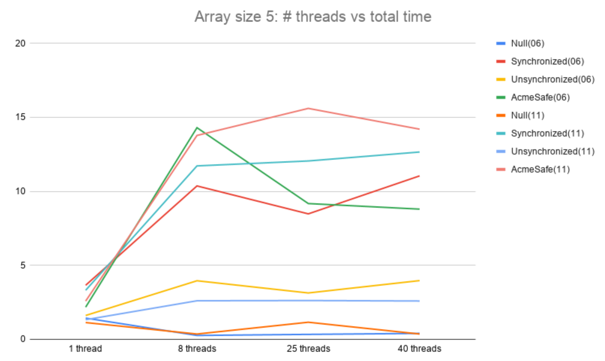
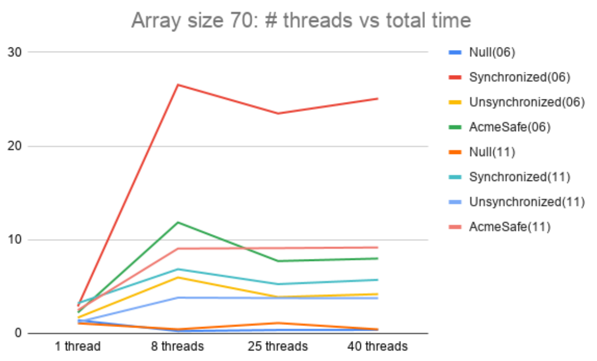


Figure 2: threads vs total time for array size 70



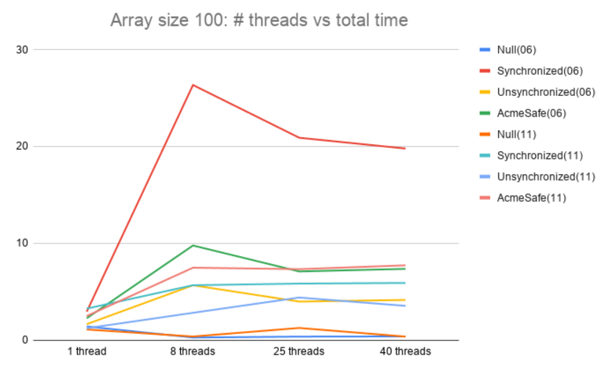
4 Problems

The first problem I faced was figuring out how to implement the AtomicLongArray so that it "implements" the State interface. The main difficulty came with the current() method describe by the State interface. In order to properly implement this function, I had to convert the AtomicLongArray into a long array. To do this I had to loop through the AtomicLongArray, insert every element into a long[], and then return the long[]. The other problem I faced was actually running all the tests. I didn't want to have to manually type the commands for each test, so I wrote a simple bash script to run the tests for the various thread and array sizes. The most challenging/annoying part was due to the fact that I would often run into errors when trying to run my program with 40 threads. This was due to the lnxsrv most likely being overused by other students who were trying to run the same tests, and thus the server was not able to run so many threads at once. I had to re run my bash script until I got the results that I wanted.

5 Measurements and Analysis

Using the UnsafeMemory, I tested my various implementations of State by specifying various numbers of threads, array sizes, and implementations to use. For every test I had the program do a total of 100 million swaps. This number was kept constant because I did not want different implementations having to do different amounts of work. Additionally, I tested the implementations on array sizes of 5, 70, and 100 with either 1, 8, 25, or 40 threads. The three figures are for the different array sizes and measure how well the different implementations do on that array size given a certain number of threads. "How well" is determine by the total time of the program. I chose to use total time as opposed to cpu time and real time because I figured that the users of this software would mainly care about how long the program took as a whole, as opposed to say, time spent on the CPU.

Figure 3: threads vs total time for array size 100



The first and most obvious thing about these figures is that, as the array size increases, the total time for all implementations increases. The difference between 5 and 70 is prominent, whereas the difference between 70 and 100 is not too noticeable. One interesting result of the graphs is that the lines are not simply decreasing linearly as you increase the number of threads. One would assume that the more threads, the faster the computation process takes. However, we can see that this is not always the case and adding more thread can increase the program's time. This is most likely due to the overhead of adding more threads. For example, using SynchronisedState with 8 threads takes longer than with a single thread, which is most likely due to the all the waiting that the thread have to do when attempting to access the critical sections of code. Another interesting thing to note is that the AcmeSafe implementation runs a lot faster than the synchronised implementation. This makes sense because the whole point of the AcmeSafe implementation is to have better handling of race conditions instead of using the slow method of placing locks around the entire swap method. Lastly, the Unsynchronised implementation is clearly the fastest, but at the cost of errors in our array. The more threads that I used, the more error prone the array was due to an increase in race conditions. Overall, it appears that the AcmeSafe implementation was the best due to it being faster than SynchronisedState while retaining thread safety!

6 References

https://web.cs.ucla.edu/classes/winter21/cs131/hw/hw3.html
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/atomic/AtomicLongArray.html
http://gee.cs.oswego.edu/dl/html/j9mm.html