CS131 Homework 3 Report

Abstract

Exploring three different ways in which one can use multithreading in Java through the use of three similarly implemented classes. Each class was tested on the same program and the results were analyzed for efficiency.

1 Introduction

This homework assignment was focused around multithreading within the programming language of Java. As such, basic knowledge about the Java Memory Model (JMM) is necessary for any tests and results to be analyzed in a meaningful manner. The JMM is designed for optimization and as such it freely reorders some actions so long as this reordering would not cause a change in result for a single thread. Because the JMM makes no guarantees about the influence on other threads, Java has ways for this problem to manually be avoided, such as the "synchronized" keyword. However, as documented in Doug Lea's "Using JDK 9 Memory Order Modes", this keyword alone is insufficient for good multithreading practices with multicore processors, as overuse significantly slows the program and underuse may too often allow for simultaneous memory accesses that yields an incorrect result. The purpose of this assignment was to explore just that: the other ways in which one can prevent data races and still keep sufficient efficiency.

2 Implementation

Students were provided with a .jar file containing a program that creates an array of longs of a given size with each value initialized to 0. It contains a swap function that randomly takes two array indices i and j and subtracts one from i and adds one to j. This is all performed on a specified number of threads. Additionally, the final argument taken specifies which implementation of the class State, which contains the array of longs, the swap function, and more, the program should use. Two of these implementations were provided: NullState and SynchronizedState. NullState's implementation of swap has it do nothing, and all of the implementations of SynchronizedState were, rather straightforwardly, synchronized. Students were then tasked to create two more of these implementations.

2.1 UnsynchronizedState

The goal of UnsynchronizedState's implementation is to be nearly identical to SynchronizedState's implementation with one key difference: UnsynchronizedState is of course unsynchronized. This task was rather straightforward, and I accomplished this by copying SynchronizedState and subsequently simply removing all instances of the "synchronized" keyword.

2.2 AcmeSafeState

The goal of AcmeSafeState's implementation is to perform faster than SynchronizedState while still keeping the implementation safe. In my implementation of AcmeSafeState, I turned to the java package java.util.concurrent.atomic and its class AtomicLongArray for assistance. This is effective for ensuring a safe implementation because the class includes atomic methods. Atomic methods, of course, create a precedence that must be followed by all operations, namely that 'set' operations must be performed before 'get' operations of the same variable. Thus, for AcmeSafeState specifically, other threads are prevented from accessing the array simultaneously and race conditions are prevented, and is therefore DRF.

To show that AcmeSafeState accomplishes its goal of improved efficiency over SynchronizedState, we must first characterize SynchronizedState. The "synchronized" keyword that serves as the foundation for SynchronizedState acts as a lock, meaning that any thread that accesses a variable creates a lock and prevents all other threads from doing the same. Once finished with the variable, it is unlocked and other threads once again are able to access it. As such, according to Lea's paper, SynchronizedState would be characterized as an example of Locked mode, a classification that requires total ordering of locked regions.

Compare this now to AcmeSafeState, which, as aforementioned, uses atomic accesses as its safety insurance. Our implementation is harder to classify in accordance with Lea's paper because VarHandle is built-in to our package. However, the lack of a lock clearly communicates that AcmeSafeState is not an example of Locked mode and furthermore serves as a chief reason that AcmeSafeState has better efficiency.

3 Problems With Measurements

A problem that I faced while obtaining these measurements was the fact that I was using a VPN to connect to seasnet. This, of course, is to be expected during a global pandemic, but the internet from my house is rather unstable, and at times I would disconnect at random.

As such, this may have led to a different problem with my measurements that I was unable to work around: some tests had to be taken at different times. The inconsistency of the times all the tests were taken could be problematic because the seasnet servers are public and used by many people. However, the activity on the servers throughout the day surely changes and as such the results could have been affected by the amount of activity on the servers at any given time.

4 Measurements and Analysis

In order to test my implementations of these classes and measure their effectiveness against each other, I ran these tests on lnxsrv06 and lnxsrv10. The purpose for this was to measure the performance on two different processors: lnxsrv06 has an "Intel(R) Xeon(R) CPU E5620 @2.40 GHz" cpu model and lnxsrv10 has an "Intel(R) Xeon(R) Silver 4116 CPU @2.10 GHz" cpu model. To ensure consistency, I used "java -version" and found that both servers were running java 13.0.2.

On each server, I tested and recorded the results for each of the four classes: NullState, SynchronizedState, UnsynchronizedState, and AcmeSafeState. For each class, I tested for array sizes of 5, 100, and 824, and within each array size I tested for 1, 8, 24, and 40 threads. Every test was run with 100,000,000 iterations of swap as to ensure that the results reflected the time of the swaps themselves and not of other factors such as the startup time. The results can be found in the following figures.

	Thread	1	8	24	40
lnxsrv	Class				
06	Null	1.42	0.26	0.44	0.29
	Sync	2.31	23.91	17.72	22.85
	Unsync	1.60	4.43	2.98	2.75
	Acme	2.28	14.45	9.45	6.00
10	Null	1.14	0.35	0.53	1.43
	Sync	1.66	5.62	5.12	4.92
	Unsync	1.21	1.91	2.06	2.07
	Acme	2.54	13.73	10.58	5.74

Figure 1: Real time runtime output for each number of threads for each class for each server for an array of size 5. Red boxes indicate that the program failed by returning an incorrect result.

lnxsrv	Thread / Class	1	8	24	40
06	Null	1.37	0.28	0.26	0.26
	Sync	2.82	25.49	20.88	22.16
	Unsync	1.89	5.79	3.87	4.21
	Acme	2.42	7.09	5.37	5.36
10	Null	1.07	0.37	0.52	0.36
	Sync	2.16	4.95	4.94	4.87
	Unsync	1.23	3.66	3.75	3.66
	Acme	2.49	7.56	8.03	5.16

Figure 2: Real time runtime output for each number of threads for each class for each server for an array of size 100. Red boxes indicate that the program failed by returning an incorrect result.

lnxsrv	Thread / Class	1	8	24	40
06	Null	1.38	0.27	0.31	0.34
	Sync	2.35	27.68	22.03	22.77
	Unsync	1.73	6.49	5.60	5.66
	Acme	2.30	3.13	2.83	3.31
10	Null	1.08	0.37	0.52	0.66
	Sync	1.77	3.90	4.39	4.26
	Unsync	1.22	2.35	2.39	2.41
	Acme	2.46	3.90	3.86	3.68

Figure 3: Real time runtime output for each number of threads for each class for each server for an array of size 824. Red boxes indicate that the program failed by returning an incorrect result.

The natural place to begin our data inspection is with the differences between the times for SynchronizedState and AcmeSafeState. It is also here where we see a rather interesting phenomena occur. If you draw your attention to Figure 1, it's clear that SynchronizedState outperformed AcmeSafeState on lnxsrv10 regardless of the number of threads. However, the more threads that were used, the closer AcmeSafeState was to reaching the efficiency of SynchronizedState. Looking now at Figure 2, we can see almost this exact same occurrence again for lnxsrv10, but something interesting happens in Figure 3. Though, yes, it starts the same as the other two with SynchronizedState outperforming AcmeSafeState, AcmeSafeState outperforms SynchronizedState for 24 and 40 threads. The reasonable conclusion to draw from these observations is that SynchronizedState performs better with small arrays and a minimal number of threads, but as each of those increase AcmeSafeState's efficiency is the one to triumph. Finding the point at which AcmeSafeState's efficiency overtakes that of SynchronizedState is an intriguing idea that I may perhaps explore on my own.

Interestingly, however, we do not see this same phenomena for lnxsrv06. The results from lnxsrv06 show that AcmeSafeState is always more efficient than SynchronizedState, regardless of the array size or number of threads. This is the type of improvement we expected by switching from locks to an atomic implementation, and it's curious as to why the different servers produced these differing results.

Though the natural place to begin was with the comparison of SynchronizedState and AcmeSafeState, the most noticeable feature of the tables is the red painted cells. In both servers, regardless of the size of the array, UnsynchronizedState failed for every number of threads besides just one. This, of course, was what was expected of a multithreaded program using a class with no protections, and is unsurprising to see. Because UnsynchronizedState is not DRF, it was inevitably going to run into data races through the 100,000,000 iterations of swap.

The discrepancy between SynchronizedState and every other class used on lnxsrv06 is enormous and should not be understated. If, for example, we look at Figure 3, lnxsrv06 Synchronized for 8 threads, we see that it took 27.68 real time seconds. The next slowest for that category was UnsynchronizedState's 6.49 seconds, which is over four times faster than its opponent. And in comparison to AcmeSafeState, which was tasked specifically with having greater

efficiency than SynchronizedState, SynchronizedState is close to nine times slower.

Unsurprisingly, NullState was the fastest across the board regardless of server, threads, or array size. Additionally, UnsynchronizedState was faster than SynchronizedState in every instance of course, as its implementation was identical outside of the synchronization. Something that is interesting, however, is that AcmeSafeState was able to outperform UnsynchronizedState for array size 824 on lnxsrv06.

Overall, lnxsrv10 outperformed lnxsrv06 in nearly every cell. Though my aforementioned internet issues may have played a factor as I tested lnxsrv06 before lnxsrv10, which I tested late into the night when fewer people were likely using the servers, the greater memory of lnxsrv10 likely played a bigger role.

5 Conclusion

Though there are some exceptions, such as running the program on smaller arrays on specific servers, AcmeSafeState is generally the best choice. SynchronizedState performs much slower on larger arrays and more threads, and UnsynchronizedState produces incorrect results for more than one thread. AcmeSafeState is DRF and efficient, and is thus the best choice for most applications.

References

Eggert, Paul. "Homework 3. Java shared memory performance races". https://web.cs.ucla.edu/classes/fall20/cs131/hw/hw3.html, 2020.

Lea, Doug. "Using JDK 9 Memory Order Modes". http://gee.cs.oswego.edu/dl/html/j9mm.html, 2018.