Homework 3 Report

1 Introduction

The purpose of this assignment was to experiment with the Java Memory Model, or JMM. The JMM defines how Java programs are able to access shared memory while avoiding data races. In particular, this assignment was targeted towards the testing of various synchronization methods used on large arrays of data. These methods were then measured in terms of real time and CPU time on the lnxsrv06 and lnxsrv11 servers. Both servers ran Java 15.0.2, but provided different conditions for testing. lnxsrv06 has a Intel(R) Xeon(R) CPU E5620 @ 2.40GHz, while lnxsrv11 has a Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz. In addition, lnxsrv06 runs on Red Hat Version 7.8, while lnxsrv11 runs on Red Hat Version 8.2. Testing was done through the provided `UnsafeMemory` test harness. In the end, the goal was to achieve data-race free behavior while also analyzing potential performance gains compared to the use of the `synchronized` keyword.

2 AcmeSafeState Implementation

Performance-wise, the problem with `SynchronizedState` was that of locking. As described in Lea's paper, use of `synchronized` blocks in the program results in mutual exclusion of execution. This adds extra overhead to our runtime, as threads are requried to process Acquire mode reads and Release mode writes in order to maintain a DRF state. As seen in all tests run on `SynchronizedState`, this extra overhead resulted in a worse runtime when compared to single-thread execution:

```
Single-thread:
Total time 2.99502 s real, 2.99319 s CPU
Multi-threading:
Total time 28.4691 s real, 91.4220 s CPU
```

While these results are pulled from a specific test case, the same jump in runtime was found for all test cases. When the `synchronized` keyword is removed from the program, the runtime improves, but the program becomes extremely vulnerable to race conditions, which is why we need to implement `AcmeSafeState` as an alternative method to maintaining DRF. Instead of using `synchronized` blocks, `AcmeSafeState` uses the `java.util.concurrent.atomic` package. More specifically, it makes use of the `AtomicLongArray` class implemented by the package. According to Lea, this class defines methods based around the VarHandle constructions in JDK 9. These methods, including `getAndDecrement()` and `getAndIncrement()`, which were used to implement the `swap()` function of `AcmeSafeState`, are applied to individual elements of the `AtomicLongArray`. By using these methods instead of traditional increments and decrements, each operation becomes atomic. This means that it is impossible for them to be interrupted like in the race conditions generated by `UnsynchronizedState`. At the same time, we avoid the locking behavior that dominates the runtime of `SynchronizedState`, deferring to VarHandles-based structures instead in order to maintain DRF.

3 Obstacles

The largest obstacle I ran into when collecting data on my classes was the inconsistency of the Linux servers. Since these servers are available to many students, the load on each server varies greatly over time. As a result, they do not necessarily offer a stable testing environment. To account for this, I did my best to record data for all my classes in as little time as possible to minimize these potential variances. However, this methodology means that I only ran a single test for each test case. Admittedly, this makes my method vulnerable to possible outliers that would have been caught with repeated trials. However, I decided that the benefits of running my tests in a stable environment outweighed the possible downsides of outlier cases, especially since many test cases were used in the first place. Other than that, my data collection process went smoothly, although manually testing and recording data did become tedious, so, if I were to repeat these tests, I would attempt to automate them in some way.

SynchronizedState lnxsrv06 Tests (s)

	5 Elements	50 Elements	100 Elements
1 Thread	2.99502	2.93878	3.11584
8 Threads	28.4691	26.8422	29.1476
20 Threads	21.6274	21.2001	22.9366
40 Threads	23.8207	22.2985	23.4051

Figure 1: Table of total time results from testing SynchronizedState on lnxsrv06, measured in seconds.

SynchronizedState lnxsrv11 Tests (s)

	5 Elements	50 Elements	100 Elements
1 Thread	3.28289	3.32105	3.22266
8 Threads	11.7353	9.12654	7.42817
20 Threads	15.7293	9.54230	7.31392
40 Threads	15.4813	9.07697	7.46487

Figure 2: Table of total time results from testing SynchronizedState on lnxsrv11, measured in seconds.

AcmeSafeState lnxsrv06 Tests (s)

	5 Elements	50 Elements	100 Elements
1 Thread	2.16252	2.17529	2.19701
8 Threads	15.9321	8.96345	6.50802
20 Threads	9.89013	5.69045	4.49573
40 Threads	9.95894	5.84947	4.38166

Figure 3: Table of total time results from testing AcmeSafeState on lnxsrv06, measured in seconds.

AcmeSafeState lnxsrv11 Tests (s)

	5 Elements	50 Elements	100 Elements
1 Thread	2.58048	2.48382	2.50308
8 Threads	10.9011	9.31282	7.23213
20 Threads	11.6536	9.27939	7.76318
40 Threads	11.6423	8.92297	7.34787

Figure 4: Table of total time results from testing AcmeSafeState on lnxsrv11, measured in seconds.

4 Measurements and Analysis

The first measurements that were taken over the course of this assignment were the test cases for the `SynchronizedState` class. To keep consistency, 100,000,000 swaps were performed for all tests, but the number of threads used for multi-threading varied between 1, 8, 20, and 40, while the array size varied between 5, 50, and 100. The provided `UnsafeMemory` test harness outputs multiple time measurements, including real time, system CPU time, and user CPU time. However, these times tended to follow the same trends as the test cases were varied, so this analysis will focus on the total time to simplify things. After running tests on `SynchronizedState`, I proceeded to run the same test cases on the `AcmeSafeState` class. Each set of test cases were performed across the lnxsrv06 and lnxsrv11 servers in order to detect changes across different hardware implementations. The results of these test cases are pictured above.

The first thing that stands out is that, regardless of the synchronization technique used, the single-threaded execution is always the most efficient in terms of runtime. This shows that `SwapTest` is not well suited for parallelism. It's likely too simple of an operation to gain much benefit from multi-threading, resulting in the added overhead of multi-threading hurting the runtime more than it helps. As threads are added, a trend appears where the runtime spikes at 8 threads, and proceeds to improve as 20 and 40 are tested. This is likely because the additional threads begin to negate the overhead generated from multi-threading, resulting in worse runtimes being found at low thread counts.

Another simple conclusion to make focuses on the single-threaded tests. For both the `SynchronizedState` and `AcmeSafeState` classes, the single-threaded tests ran faster on lnxsrv06 than on lnxsrv11. This seems to make sense, as lnxsrv06 has more cache memory, which would result in more caching and faster memory accesses over the course of the testing. Of course, any of the other differences in the hardware of the servers may also contribute to this difference, but they would be out of the scope of my knowledge.

It is also possible to see that the increase in array size helps performance. This is because, for both classes, when an array entry is being operated on, it cannot be accessed by other threads. For this reason, in small arrays, most of the entries will be locked at any given time, preventing threads from being useful. As the array size is increased the rate at which these collisions happen will go down, resulting in more efficient processing. This is true of both synchronization techniques used in this assignment, so both classes operate more efficiently on large arrays.

When looking at the results of `SynchronizedState`, it's clear that the use of `synchronized` blocks was much more effective on lnxsrv11 than on lnxsrv06. This is despite the single-threaded tests running slower on lnxsrv11 than lnxsrv06. Together, these results seem to point to the conclusion that lnxsrv11 is more suited for taking advantage of multi-threading. However, when looking at the `AcmeSafeState` results, the opposite seems to be true. While the differences are not as drastic as the ones present in the `SynchronizedState` tests, running the test harness on lnxsrv06 does seem to produce better runtimes across the board. The only exception to this is the 5 element, 8 thread case, however, it is likely that this result is simply a result of the inconsistency of the Linux servers. With these results in mind, it is perhaps more accurate to say that lnxsrv11 is better suited for parallelism by way of locking, while lnxsrv06 is better suited for the VarHandle style of synchronization used by `java.util.concurrent.atomic`.

Continuing on with that conclusion, it is also possible to see this trend when examining the benefits obtained from using `AcmeSafeState` over `SynchronizedState`. On lnxsrv06, the benefits gained from `AcmeSafeState` were very large, with runtimes going down dramatically. However, on lnxsrv11, this is not the case. The benefits gained on lnxsrv11 are negligible at best. With the inconsistency of the servers, it is impossible to say whether any benefits actually exist at all. Once again, this points to the conclusion that lnxsrv06 is much better suited for the parallel behavior exploited by `AcmeSafeState`.

In summary, the `SynchronizedState` class seems to work best on lnxsrv11, while the `AcmeSafeState` class seems to work best on lnxsrv06. Both classes benefit greatly from larger array sizes, and, while single-thread execution is most efficient, multi-threaded execution is improved as threads are added. Overall, the `AcmeSafeState` class has superior performance to the `SynchronizedState` class in the vast majority of the tests used here, while maintaining the same degree of reliability.

5 References

"Package java.util.concurrent.atomic." Available:
https://docs.oracle.com/en/java/javase/15/docs/api/index.html

D. Lea "Using JDK 9 Memory Order Modes." Updated November 16, 2018. Available:
http://gee.cs.oswego.edu/dl/html/j9mm.html

P. Eggert "Homework 3. Java shared memory performance races." Updated January 28, 2021. Available:
https://web.cs.ucla.edu/classes/winter21/cs131/hw/hw3.html