CS 131: Homework 3 Report

Java shared memory performance races

1 Introduction

This report will discuss the results gathered by running a modified Java program under a given test harness using two different servers - lnxsrv06 and lnxsrv11. In addition, I will explore topics mentioned in Lea's paper, such as data race conditions and how they relate to VarHandle methods and operations.

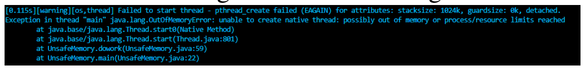
2 AcmeSafeState implementation

By using the java.util.concurrent.atomic.AtomicLongArray object, I was able to surpass the performance of SynchronizedState (exact figures will be shown later on in the report). Since part of the challenge was to avoid using locks / mutexes, I used a form of atomic increments and decrements to achieve the desired behavior. Namely, the void swap(int i, int j) method within my implementation of AcmeSafeState utilized two functions from the java.until.concurrent.atomic package getAndIncrement(i) and getAndDecrement(i). The memory effects of these functions,as specified by VarHandle.getAndAdd(java.lang.Object...) VarHandle references are rather dynamic - as Lea states, "they can be associated with any field, array element, or static, allowing control over access modes."

3 Data Race Free?

In the implementation of UnsynchronizedState, which intentionally disallows the use of locks and the "synchronized" keyword, threads are accessing, reading, and writing to memory concurrently. Data races are bountiful here, as threads are allowed to break sequential consistency. This causes particularly nasty behavior - namely a "sum mismatch" within this particular program. My implementation for AcmeSafeState is free of data races, as it performs atomic operations which eschew issues related to schedulers and thread interference / conflicts.

Figure 1: Error Message



```
[0.115s][warning][os,thread] Failed to start thread - pthread_create failed (EAGAIN) for attributes: stacksize: 1024k, guardsize: 0k, detached.
Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limits reached
        at java.base/java.lang.Thread.start0(Native Method)
        at java.base/java.lang.Thread.start(Thread.java:801)
        at UnsafeMemory.dowork(UnsafeMemory.java:59)
        at UnsafeMemory.main(UnsafeMemory.java:22)
```

4 Problems Faced

Although I didn't run into too many problems when designing my program, I ran into a few issues when testing on the various servers. It was a bit tedious gathering results for every benchmark, so I used an Excel spreadsheet to help me keep track of my data. Instead of dumping the data directly into this report, I created a few bar graphs to maximize user readability.

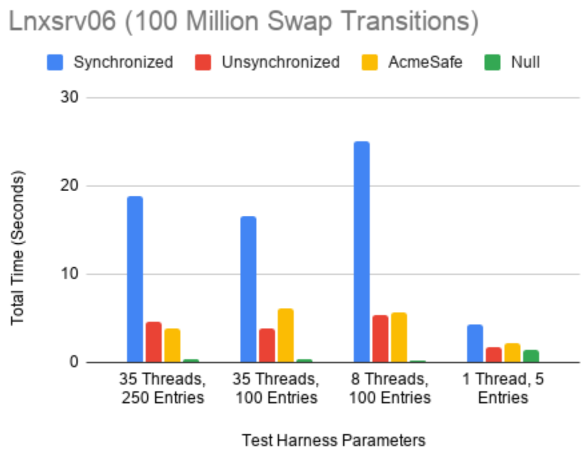
I received the error message depicted in Figure 1 when attempting to run the test harness with a thread count of 40 on lnxsrv06. Any thread count above 35-36 proved to be impossible to run, as the memory/resource capacities were at their limit on this particular virtual machine. As I do not have a local machine to test these on, I simply omitted the results for these specific test cases per the advice of my TA. I've included the results from lnxsrv11, however. To overcome this problem, I simply capped the number of threads at 35 when testing - this also served as the thread count of "my choice" as specified in the project spec.

5 Measurements and Results

5.1 lnxsrv06

As shown in Figure 2, the total time to complete the required operations with 0 sum mismatches was **significantly** higher when running the SynchronizedState class. The program took upwards of 25 seconds to complete when running 8 threads and only 100 entries - curious, since the test that was ran with 35 threads and 100 entries performed nearly 10 seconds faster. Of course, the tests ran with the baseline Null program took less than a second, as expected. It appears as if the AcmeSafeState class performed quite well, especially when compared even to the Unsynchronized speeds. AcmeSafe ran

Figure 2:



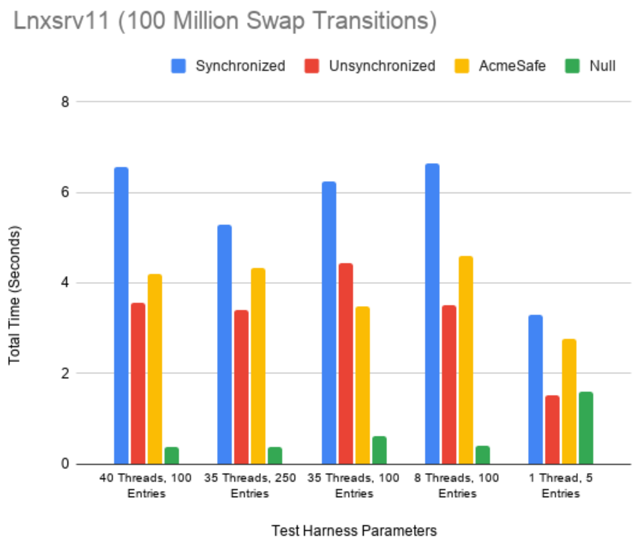
slightly slower than Unsynchronized in all but one category, where it slightly outperformed when ran with 35 threads and using a state array of 250 entries. As mentioned earlier, no tests with 40 threads were ran for this server, due to server capacity limits.

A general trend that can be observed is that more threads leads to increased time: likely due to increased overhead required for the swap transitions (with the exception of the test that required 8 threads). In general, AcmeSafe performed faster than Synchronized consistently - this is good news, as it is preventing data races while still minimizing overhead required with locking functions in Synchronized. While Unsynchronized is faster overall, it is a practically broken program that allows sequential consistency violations. GDI will not want anything to do with such an implementation.

5.2 lnxsrv11

lnxsrv11 fared much better in terms of overall performance: no test ran for more than 8 seconds. This is likely do to it being severely less overwhelmed, resource-wise, when compared to lnxsrv06. As expected, Null state runs in less than a second for all but one of the tests done - our baseline program appears to be working. Just as in the tests ran on lnxsrv06, Synchronized performed much worse than both Unsynchronized and AcmeSafe (Figure 3). AcmeSafe performed only marginally worse than Unsynchronized for each test, except where it outperformed Unsynchronized when ran with 35 threads and 100 entries. Since I was able to run the test harness with more 35 threads on this server, I have included those results in this chart as well. AcmeSafe fared relatively well, ans it performed only slower than Unsynchronized which was just a few ns shy of 4 seconds. As noted in the analysis of the

Figure 3:



results from lnxsrv06, the less threads ran, typically the faster the output(again, curiously with the exception of 8 threads). Again, AcmeSafe still appears to be more efficient than Synchronized, as its atomic operations are less costly while still preventing thread contention.

6 Final Thoughts

Overall, it seems as if AcmeSafe is the winner of these brief experiments. Again, it consistently outperforms Synchronized across servers by a large gap, and is only slightly slower than Unsynchronized. Thanks to Lea's documentation, it was easy to get a grasp on why this is the case and why locks can be so expensive to a program. It also became clear on how certain flaws in the Java Memory Model can lead to severe security-breaking consequences through means of data race side effects such as pointer forgery. Regardless, GDI should see through the implementation of AcmeSafeState due to its consistently strong performance and DRF compliance.

Acknowledgments

Much appreciation to Chi Zhang and the other TAs for their assistance with this assignment, as well as Professor Eggert.

References

gee.cs.oswego.edu/dl/html/j9mm.html
https://web.cs.ucla.edu/classes/winter21/cs131/hw/hw3.html