Java Shared Memory Performance Races

CS 131 Homework 3

Stephanie Doan

I. Abstract

In this study, we inspect the costs and tradeoffs of multithreading in Java, which powerful and necessary but requires careful attention to data race conditions to maintain sequential consistency. We will deviate from the default usage of Java's *synchronized* keyword, which places locks on critical code sections to prevent data races between threads but incurs a significant bottleneck on performance. By testing a simple program across various thread, data, and machine conditions, we seek to evaluate accuracy and performance of programs with and without an alternate approach to thread safety.

II. Background

A. AcmeSafeState Implementation

Writing the AcmeSafeState class involved adapting the functionality of the SynchronizedState class using *java.util.concurrent.atomic.AtomicLongArray* instead of Java's *synchronized* keyword used to safely update shared memory. The use of *synchronized* is known to cause a bottleneck, allowing only one thread to enter a synchronized section of code at a time. It is expensive when multiple threads attempt to run this section at once, since all non-running threads are blocked while waiting for access to the lock. Overhead also incurred from a thread having to check if it is allowed to run. Therefore, we see if our new implementation will achieve better performance and preserve accuracy in the *swap* operation.

AtomicLongArray is an array of longs that allows atomic updates of its elements. Atomic classes in Java have data structures that are nonblocking, to write lock and wait-free algorithms at the system and JVM levels. In this class, it is clear that the increment and decrement operations in the *swap* function must be atomic to prevent race conditions because simple operations actually compile into several machine-level instructions that may be executed out of order between the threads. For example, two threads may try to increment a counter but the second thread accesses, increments, and writes an incremented value between the read and write of the first thread, so these two threads wrongfully perform the increment operation on the same original value, only incrementing the value by one instead of two. To prevent such a situation, SynchronizedState uses the *synchronized* keyword for the entire *swap* function, but our AcmeSafeState instead uses the atomic *getAndDecrement(int i)* and the *getAndIncrement(int j)* instead of value[i]-- and value[j]++ in the swap function itself. These methods of AtomicLongArray essentially perform the same operation but atomically, allowing one operation to execute as if it were a single instruction, unable to be interrupted by other threads. Moving this thread-safety measure to these individual operations instead of the entire swap function will minimize the critical section that causes bottleneck, allowing for improvement in throughput.

B. Data-Race Free Classification

The AcmeSafeState class can be classified as data-race free (DRF) because of the usage of AtomicLongArray and its atomic methods. The getAndIncrement and getAndDecrement operations used are executed at the machine-level as if they are single instructions. These single instructions that add or subtract from the AtomicLongArray can be performed in any order, the 100 million swap transitions performed as tests will yield no difference in result. The compiler's freedom to reorder these atomic operations proves our thread safety measures sufficient and DRF.

III. Methods

I tested the Synchronized, Unsynchronized, and AcmeSafe implementations by calling 100 million swaps with varied numbers of threads (1,8,20,40) and state array sizes (5,20,100). We timed and compared performance of these programs run on Java 13.0.2, on different machines: lnxsrv09 with eight cores and lnxsrv10 with four cores.

Minor obstacles were encountered when performing the tests, including gaining familiarity with running Java as well as dealing with the repetitiveness of running the tests manually. Therefore, I wrote a bash script to iterate through the combinations of thread count, state array size, and synchronization method, and save results as a csv file. The tests in total also took longer than expected.

IV. Results

Threads	**Size of State Array**		
	5	20	100
1	21.1055	22.2718	23.4928
8	1651.65	1933.35	2045.19
20	4084.83	4672.88	5337.57
40	9284.31	9178.63	10475.9

Figure 1: Average swap time real (ns) for synchronized on lnxsrv09 (ns)

Threads	**Size of State Array**		
	5	20	100
1	14.9588	15.3969	16.7073
8	192.327*	392.927*	370.096*
20	838.247*	897.538*	688.831*
40	1062.32*	1538.71*	1157.65*

Figure 2: Average swap time real (ns) for unsynchronized on lnxsrv09. Values marked with * means incorrect nonzero array entries due to data race.

Threads	**Size of State Array**		
	5	20	100
1	26.9739	26.4344	27.3442
8	663.560	1063.31	467.352
20	2348.78	2233.06	916.681
40	2837.77	3442.21	1875.47

Figure 3: Average swap time real (ns) for AcmeSafe on lnxsrv09

Threads	**Size of State Array**		
	5	20	100
1	17.6514	16.4129	17.0767
8	398.413	381.793	374.786
20	1015.93	1005.82	952.882
40	2019.19	2006.63	1887.10

Figure 4: Average swap time real (ns) for synchronized on lnxsrv10

Threads	**Size of State Array**		
	5	20	100
1	12.1564	12.1707	15.4851
8	285.589*	328.511*	273.594*
20	419.747*	806.001*	713.643*
40	796.423*	1578.06*	1387.84*

Figure 5: Average swap time real (ns) for unsynchronized on lnxsrv10. Values marked with * means incorrect nonzero array entries due to data race.

Threads	**Size of State Array**		
	5	20	100
1	25.3899	24.7634	25.3016
8	1097.60	1054.04	664.666
20	2429.14	2679.82	1380.16
40	6324.83	3927.08	3725.97

Figure 6: Average swap time real (ns) for AcmeSafe on lnxsrv10

V. Analysis

A. Unsynchronized vs. Synchronization

Analyzing the data, I immediately noticed that the Unsynchronized implementation is much faster than Synchronized and AcmeSafe. It does not implement any thread-safety measures such as Synchronized's locks and AcmeSafe's non-blocking atomic operations. Especially on lnxsrv09 with eight cores, the performance advantage of Unsynchronized was large, with the state array size 100 and 40 thread trial performing at almost 10 times faster than the Synchronized version and 40% faster than the AcmeSafe version. The advantage was less prominent but still apparent on the four-core machine.

However, since unsynchronized threads execute with no regard for race conditions, there is significant drawback in accuracy. Every trial with more than one thread yields mismatched output sums. We find that simply removing *synchronized* safety measures quite consistently breaks the result, but the amount of breakage can be evaluated in terms of the tradeoff between inadequacy and speed.

B. AcmeSafe vs. Synchronized

On lnxsrv09, it is apparent that for test cases with multiple threads, AcmeSafeState outperforms Synchronized while maintaining accuracy in no mismatched sums. The 8-thread trial is two to four times faster with AcmeSafeState than Synchronized, and the 40-thread trial yielded approximately 5 times speedup. However, lnxsrv10 with large thread count (20 and 40) yielded better performance for Synchronized than for AcmeSafe, which is unexpected. AcmeSafe performed better with 8 threads.

This improvement in multi-threaded performance on the 8-core machine can be attributed to the different thread safety measures employed. AcmeSafe uses the nonblocking atomic increment and decrement operations in the swap method. It treats the multiple machine instructions as one, guaranteeing atomicity for each increment and decrement instruction. This is a more performant solution than Synchronized which places a lock on the entire *swap* function, making other threads wait one to finish all machine instructions inside the function before executing. This essentially eliminates all speed up due to concurrency in this code section. The slowdown from Synchronized is more apparent with large thread count.

C. Performance Across Machines

However, the most unexpected result came from the difference between performance on lnxsrv09 and lnxsrv10. No significant difference was found between the two in the Unsynchronized implementation. However, the Synchronized trials were more performant on the 8-core machine than on the 4-core machine for the largest number of threads (40) but the opposite was true in the AcmeSafe trials. We expect better performance from greater processor count on lnxsrv09, but AcmeSafe yielded an unexpected result. This suggests a profound difference between the two thread-safety measures, atomic operations and locks, and how they are affected by processor count. More experimentation is needed to draw conclusions about the differences between the machines because performance can also be affected by traffic on those servers at the time of testing.

VI. Conclusion

In conclusion, we found significant differences between the two approaches to improve upon Synchronized's approach to thread safety. Placing a lock on the entire *swap* operation incurs significant overhead that is immediately eliminated when deleting the *synchronized* keyword all together. The Unsynchronized approach was clearly faster but yielded very wrong results due to data races when using more than one thread, so multithreading would create significant error. The AcmeState implementation used atomic operations, treating the increment and decrement methods of *AtomicLongArray* as if it were a single instruction even though it is broken down into three machine instructions. This DRF approach is finer grained and applies to the individual increments and decrements instead of the entire swap operation. Despite AcmeState's worse performance than Synchronized in the 4-core machine with large thread count, AcmeState yielded much more significant improvement in the 8-core machine, from 2 to 5 times the efficiency, while maintaining the accuracy from keeping the code data race free.