Homework 3: Java Shared Memory Performance Races

1 Introduction

For this assignment, we look at the Java Memory Model and its language implementation optimization that assumes sequential consistency for a gain in performance in concurrent programs. However, Java's concurrency suffers from race conditions due to these assumptions, so without proper synchronization, a program may behave incorrectly. Here, we look at some of the different methods of synchronization that Java and its packages provide in order to remedy data races and compare the performance and reliability of each. More specifically, the 'synchronized' keyword, volatile assignments and accesses, and locks are the basic synchronization methods that will be implemented and tested, and our goal is to see how badly a program could break when trying to speed it up by less honest means.

2 Machine Specifications

All implementations and testing were done using Java versions 9 and 11.0.2 with OpenJDK Runtime Environment builds 1.8.0_201-b09 and 11.0.2+9. The testing machine is SEASnet Linux Server 10, which is a quad CPU machine running four Intel(R) Xeon(R) Silver 4116 CPUs, each with a clock speed of 2.10GHz and 4 cores, for a total of 16 cores. Some further research using the 'lscpu' bash command shows that each CPU runs one thread per core. The operating system of the machine is RHEL 7 and its memory total is approximately 65.8 GB.

3 Packages and Classes

When considering different synchronization methods in designing the BetterSafe class to perform better than the Synchronized class, while still maintaining 100% reliability, four packages were examined.

3.1 java.util.concurrent

The *java.util.concurrent* package is useful in that it contains five concurrency tools: Semaphore, CountDownLatch, CyclicBarrier, Phaser, and Exchanger, alongside thread-safe data structures like queues, making it a very versatile package.[2] However, each individual concurrency tool offers its own advantages and their overall APIs are much harder to grasp than locks, making use of the package overwhelming. Furthermore, tools such as Semaphore have special constructors that allow for additional constraints on the order of thread accesses.[2] However, while this may make a tool more reliable, the question of "how reliable" and the complexity of the API still drove the implementation of BetterSafe in a different direction for me.

3.2 java.util.concurrent.atomic

The *java.util.concurrent.atomic* package contains atomic implementations of some primitive types, including integer and long, as well as extensions of arrays of these types. [2] Furthermore, the package provides both low-level and high-level atomic transformations through the use of the 'compareAndSet' and 'getAndUpdate methods', which is an overall pro. However, while atomic operations may result in better performance, because atomic accesses fall under the opaque memory mode and have weaker guarantees, these class were rejected when considering implementations for the BetterSafe class.

3.3 java.util.concurrent.locks

The *java.util.concurrent.locks* package provides a framework with the 'Lock' interface that allows for greater flexibility and customizability, but at the cost of more awkward syntax.[2] The package contains two basic locking mechanisms: ReentrantLock and StampedLock; however, when deciding between the two mechanisms, it is understood that StampedLocks are more likely to incur deadlocked threads, so ReentrantLocks are more favored. Ultimately, ReentrantLock is the chosen method of synchronization for the BetterSafe class, since it falls under the same memory order mode as the 'synchronized' keyword, ensures mutual exclusion with more extended capabilities, and has a relatively simple API.

3.4 java.lang.invoke.VarHandle

This package implements the VarHandle class that is a dynamically strongly type reference to a variable; for concurrency, this class implements methods that allow for volatile reads and writes, and

compare-and-sets amongst other things through access modes.[2] While this package does offer a variety of access modes to control the atomicity and consistency of a variable, the API of the package is complicated and for someone who is limited by time, learning and implementing these methods is difficult.

4 Testing Conditions

The testing harness uses a multithreaded program that simply updates two elements in an array of integers; one element is decremented, while the other is incremented, so the sum before and after the operations should remain the same. My testing methodology uses a bash script with several loops to expedite the process of trying different testing constraints. Such testing constraints include using Java versions 9 and 11.0.2, using 2, 4, and 8 threads, trying 10, 1000, 10000, and 100000 iterations, and using maxvals ranging from 6, 64, to 127. All possible combinations of these constraints were used for every class and the results were stored in their own individual text files.

5 Analysis

To greater elucidate the tradeoff between reliability and performance, we look at two metrics by which to measure both aspects of my implementations: average time per transition in nanoseconds and the failure rate of a given class, which implies how often a class causes data races.

5.1 Benchmarks

For my performance benchmarks, the test cases that use 8 threads and a maxval of 64 with varying numbers of iterations were used to represent all data sets; for the most part, all other combinations of constraints produced results similar to the ones shown in the tables below. These tables compare the average transition time in nanoseconds for each class when using Java versions 9 and 11.0.2.

	Time Per Transition (ns)				
Iterations	Null	Synchronized	Unsynchronized	GetNSet	BetterSafe
10	1.63E+06	2.37E+06	4.37E+06	2.36E+06	2.77E+06
1,000	27913	63095.1	109990	81944.6	64345.6
10,000	9998.29	12498.3	5566.29	15714.9	10759
100,000	1099.79	3405.04	1837.24	3726.99	2268.44

Table 5.1 Performance Benchmark using Java 9

	Time Per Transition (ns)				
Iterations	Null	Synchronized	Unsynchronized	GetNSet	BetterSafe
10	6.18E+06	4.35E+06	4.88E+06	2.74E+06	3.77E+06
1,000	44417.6	55471.7	52494.7	71752.3	86667.2
10,000	160087	10275.7	14617.7	29716	12249.1
100,000	3149.99	4440.42	3465.67	4856.25	2988.03

Table 5.2 Performance Benchmark using Java 11.0.2

From the data shown, it can be seen that the Unsynchronized class does speed the program up in comparison to the Synchronized class, although this becomes much less apparent in the Java 11.0.2 tests. However, this may be due to the CPU load of the server affecting measurements, amongst other factors. The BetterSafe class also performs better than the Synchronized class for larger numbers of iterations.

Other interesting observations are that for larger numbers of iterations, the time per operation decreases; this is likely due to the overhead of thread creation and resource acquisition being amortized by the greater number of operations. For the most part, Java 9 and Java 11.0.2 perform about same for these tests.

5.2 Reliability

In testing, we see that the Synchronized and BetterSafe classes perform as desired, providing 100% reliability in preventing data races, as they operate under locked memory order modes, wherein they use ‘synchronized’ blocks and ReentrantLocks respectively. The Null class provides no actual functionality, so its reliability is certain.

On the other hand, the Unsynchronized and GetNSet classes produce varying results due to data races, caused by their lack of synchronization methods. To exemplify this, the Unsynchronized and GetNSet classes were tested using different combinations of

thread counts and iteration counts with a maxval of 64. Each test was repeated 100 times, counting all tests which failed; a test is considered a failure if it produces an mismatch (exit code 127) or times out after 3 seconds (exit code 124), indicative of a hanging execution. The tables below represent the failure rates of the aforementioned classes for each test; the failure rate is simply calculated by dividing the number of failures by 100 per test.

Threads	Iterations	Failure Rate
2	1,000	69%
	10,000	96%
	100,000	99%
4	1,000	80%
	10,000	97%
	100,000	99%
8	1,000	80%
	10,000	98%
	100,000	99%

Table 5.3 Reliability test for Unsynchronized

Threads	Iterations	Failure Rate
2	1,000	85%
	10,000	92%
	100,000	96%
4	1,000	97%
	10,000	98%
	100,000	95%
8	1,000	96%
	10,000	99%
	100,000	98%

Table 5.4 Reliability test for GetNSet

From the tables above, it's apparent that both classes suffer from a high rate of data races and infinite loops, where the failure rate for the Unsynchronized class may be more related to the number of iterations than the GetNSet class, but still unreliable nonetheless. It can be seen that despite GetNSet using volatile accesses over the Unsynchronized class, it is still as unreliable as a class with no synchronization at all.

8 Conclusion

While unsynchronized programs may be faster, they are far and away less reliable than synchronized programs as evident by their failure rates. For 100% reliability, the 'synchronized' keyword trades performance for a simple API and data-race-free programs. As Doug Lea explains, this is a necessary tradeoff as stricter memory order modes like the locked mode place greater constraints on the optimizations the Java Memory Model can perform.[2] However, to regain some of this performance, I used a ReentrantLock, which seems to perform better than the 'synchronized' keyword due to its unstructured nature and finer granularity. The 'synchronized' keyword works by locking block structures of code, which in this case is an entire method. However, ReentrantLocks allow for only the critical section of the code to be locked, so threads are not restricted from doing useful work when they need to access certain methods, while still having 100% reliability from locking the critical section, and for operations as small as the ones performed by the testing harness, this can result in some performance gain.

The only problems I faced when taking my measurements were issues with GetNSet executions hanging for large numbers of iterations and small maxvals; this was probably due to the small maxval constraining the swaps. Some tests had to be omitted due to this, but I wrote a 'timeout' loop that retries certain tests in my testing script for these special cases.

References

1. Java API Reference [Internet]. Overview (Java SE 11 & JDK 11). Oracle; 2018 [cited 2019May9]. Available from: https://docs.oracle.com/en/java/javase/11/docs/api/index.html
2. Lea D. Using JDK 9 Memory Order Modes [Internet]. 2018 [cited 2019May8]. Available from: http://gee.cs.oswego.edu/dl/html/j9mm.html