CS 131 HW 3 Report

1 Introduction

In this assignment, we were tasked with developing an understanding of the Java Memory Model by experimenting with different methods to synchronize a basic multi-threaded task. Below we will discuss the reliability and performance of different synchronization methods for this task and analyze their interactions with the Java Memory Model.

2 Test Design

The test task we used to measure performance and reliability of synchronization methods was to begin with a byte array where all values were less than or equal to some beginning value *maxval*. Our program is then charged with going through a certain number of *transitions*, or *swaps*, where it randomly selects two indices in the array and checks if it's possible to decrement the first and increment the second while keeping values within 0 and *maxval*. If it's possible it does so, otherwise it returns false.

2.1 Test Platform

The program was tested on two systems. The first, my personal computer, contains two Intel Core i5-3320M physical cores, each with two logical cores for four logical cores total. In addition there is about 11.5GB of available memory for non-system processes. This system runs OpenJDK 9, build "9-internal".

The second, the SEASnet's *lnxsrv10*, has four Intel Xeon Silver 4116 physical cores visible, each with one logical core, for four logical cores total. In addition there was about 14.0GB of available memory for new processes when measured. This system contains OpenJDK 9 and 11, builds "9+181" and "11.0.2+9", respectively. Although my personal computer isn't as powerful as the SEASnet's server, it will be useful for taking measurements isolated from the effects of other users running programs on the server.

2.2 Designs Tested

In our experiment we tested four classes, *Synchronized*, *Unsynchronized*, *GetNSet*, and *BetterSafe*. The Synchronized class uses the synchronized keyword to ensure that calls to the swap method are synchronous and is data race-free (DRF). See below for more information on internal implementation. The Unsynchronized class is the same class but without the synchronized keyword and so has no protections against data races caused by multiple threads accessing the same variables. The GetNSet class uses an AtomicIntegerArray to store the data instead of a byte array. The implementation of AtomicIntegerArray means that all variables are *volatile*, ie they are not cached thread-locally. However, there is still a data race present, namely when checking that two indices can be swapped and actually doing the increment/decrement. Finally, BetterSafe is an implementation based on ReentrantLock, see below for a discussion why I decided to use this.

We will measure the reliability of each of the four classes; however, note that since Unsynchronized and GetNSet are not actually DRF, I decided to only measure the performance characteristics of Synchronized and BetterSafe. We also use a Null class that does nothing as a benchmark for time taken up by the test harness.

3 Synchronization Methods

synchronized keyword

The *synchronized* keyword when attached to a method signature tells the Java compiler that calls to the method should be guaranteed to be synchronous. This is implemented by acquiring that object's *intrinsic lock* before method invocation and releasing that object's lock after exiting the method (either through a return statement or exception). This enforces exclusive access to the object during the method's runtime and establishes a happens-before relationship with all later code interacting with the object. See Ref. 1 for more information.

java.lang.invoke.VarHandle package

This package contains VarHandles a feature introduced in Java 9 to allow for accesses to a variable to be conducted in specific *modes*. A VarHandle wraps a variable and allows users to access that variable through in specific access modes, eg *setOpaque* allows users to set the variable in Opaque mode. There are four modes in Java 9, Plain, Opaque, Release/Acquire(RA), and Volatile, allowing for granularity when selecting how a variable access should interact with other variable access. See Ref. 2 for more details.

This package provided the most fine-grained ability to control variable accesses, this was unnecessary for the scope of this assignment.

java.util.concurrent package

This package contains various low-level constructs for implementing concurrency. These include classes to support Executors/Thread Pools, Semaphores, Exchangers, Countdown Latches, and more. While all of these are very powerful ways of handling concurrency, they are quite complicated to deal with so I decided not to use them.

java.util.concurrent.atomic package

This package contains classes that allow for volatile accesses to their underlying data structures, such as AtomicIntegerArray. While we can use classes in this package to ensure synchronicity, eg by building our own semaphore, there were easier ways to do this using the next package, so I didn't use this for BetterSafe.

java.util.concurrent.locks package

This package contains different types of locks, including ReentrantLock, a basic mutual-exclusion lock I decided to use. I chose to use this because the class had a simple interface and it was sufficient for the simple task we wanted to synchronize. We only have one critical section in swap, so all we have to do is acquire the lock before then and release afterwards.

4 Reliability

A test was considered *successful* if the sum of all elements in the array is the same before and after running the swaps. A test was considered *unsuccessful* if the sum did not match and a test was considered *failed* if the program failed to terminate. Because some of the test classes were not synchronized, they were prone to entering infinite loops caused by issues like the byte array having no candidates entries left for swapping. This made it difficult to record data for this section since it required manual intervention to realize when an infinite loop was occuring and terminate the program.

To test the reliability of the four classes, I ran them with 8 threads, maximum value 6, and initial array [5,6,3,0,3]. I tested all of them with 1,000, 100,000, and 10,000,000 iterations 10 times each on the two Java versions on the linux server and the one Java version on my laptop. Synchronized and BetterSafe were successful 100% of the time so I will be omitting the tables for those. Unsynchronized and GetNSet had reliability varying inversely with the number of iterations run for. Because the data from the three platforms all illustrated the same trend, I will just show the data from running on Java 11 on lnxsrv.

Iterations	Successful	Unsuccessful	Failure
1,000	8	2	0
100,000	3	6	1
10,000,000	0	0	10

Figure 1: Reliability data of Unsynchronized on lnxsrv Java 11

Iterations	Successful	Unsuccessful	Failure
1,000	6	4	0
100,000	1	2	7
10,000,000	0	0	10

Figure 2: Reliability data of GetNSet on lnxsrv Java 11

5 Performance

I tested my classes along three parameters, number of iterations, thread count, and array size. For every class, I ran it 10 times on a given set of parameters and worked with the arithmetic mean of the runs. All the data shown is from lnxsrv Java 11 because all the trends were the same on the three platorms. Readers can repeat this portion by using the following Bash script and gnuplot script.

5.1 Iterations

In this section I chose to test with the parameters 1 thread, maximum value 6, and initial array [5,6,3,0,3]. Number of iterations varied from 100 to 100,000,000. I chose to only use one thread so I could get meaningful data all the classes, even the unsynchronized ones, because otherwise, the unsynchronized classes could run unsuccessfully or fail on higher number of iterations.

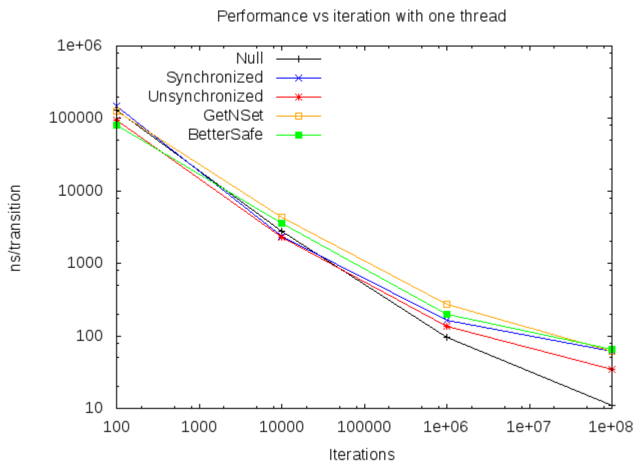

Figure 3: Performance across different number of iterations on lnxsrv Java 11

5.2 Threads

In this section I chose to test with the parameters 1,000,000 iterations, maximum value 6,and initial array [5,6,3,0,3]. Number of threads varied from 1 to 32. Note Java 11 is required for more than 20 threads. In these tests, data was only obtained from Null, Synchronized, and BetterSafe since these were the only ones that were guaranteed to succeed with these parameters and multiple threads.

5.3 Array size

In this section I chose to test with the parameters 6 threads, 1,000,000 iterations, and maximum value 6. Array sizes varied from 5 to 26.

6 Analysis

Reliability

As expected, the reliability of our synchronized classes, Synchronized and BetterSafe, was 100%. Both of the unsynchronized classes, had reliability that varied inversely with the number of iterations. This made sense as the more iterations

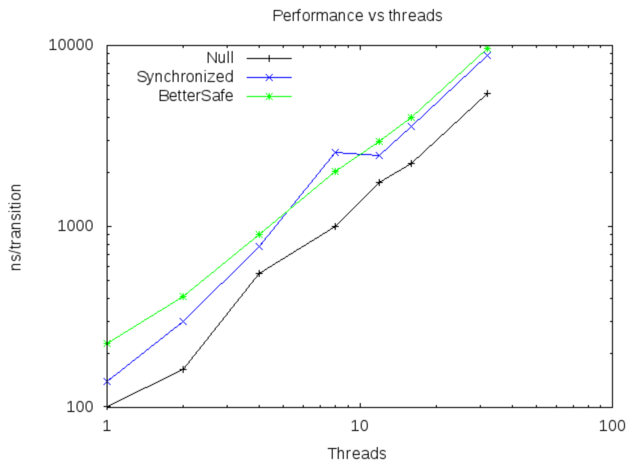

Figure 4: Performance across different number of threads on lnxsrv Java 11

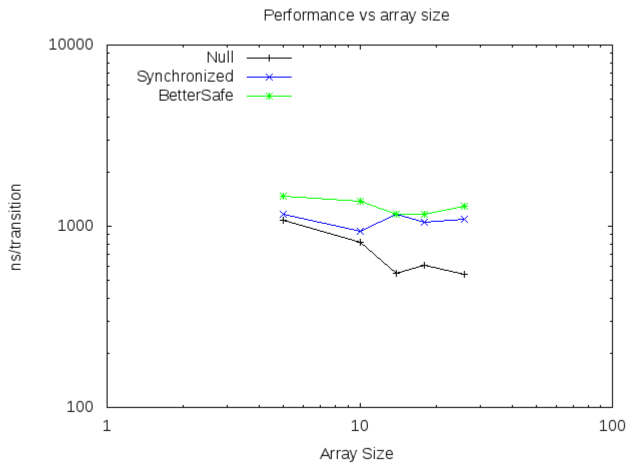

Figure 5: Performance across different array sizes on lnxsrv Java 11

that occurred, the more likely one thread was modifying a variable while another one was accessing it. Also, aross multiple runs on different platforms, GetNSet seemed to be generally less reliable than Unsynchronized. This made sense since the overhead of volatile accesses introduced in GetNSet means that the swap method is in the critical section longer than in Unsynchronized, increasing the chance of simultaneous access by two threads.

Performance

In the tests where we varied the number of iterations, we could see that although Unsynchronized was the least reliable, it was the fastest when run on a given thread. All the other classes performed roughly the same, although they all trended downwards as number of iterations increased. This is likely because as the number of iterations increases, costs like construction and destruction get amortized over more iterations.

In the tests where we varied the number of threads, we could see that at first Synchronized outperformed the BetterSafe, but this was reversed for higher thread counts. One explanation of this is because the critical section in BetterSafe is bound more tightly by its lock than the intrinsic lock used in Synchronized. So at first, Synchronized is faster because the instrinsic lock used to support the synchronized keyword is more optimized because it's a core component of Java. But, the advantage of having a more specific lock in BetterSafe appears as there are more threads since there are more opportunities for a thread to be stuck waiting.

In the tests where we varied the size of the array, the data appears inconclusive. It seems like the variance in the results are dominated more by other factors like current load of running programs than the size of the array itself. Further investigation is needed, perhaps with larger arrays.

Synchronized vs BetterSafe

Overall, we can see that BetterSafe is 100% reliable while also outperforming Synchronized. It is 100% reliable like Synchronized because the critical section, the part of swap where we check the values at indices *i* and *j* then increment/decrement, is fully fenced off by having to acquire and release a lock, ie it is DRF. This design also leads to it outperforming Synchronized because Synchronized effectively fences off the entire function *swap* while BetterSafe only does the critical section. This means that threads can do more work in BetterSafe before being stopped by having to wait to acquire a lock, thereby allowing for better performance.

Experimental Challenges

There were a couple of challenges obtaining data for these experiments. Firstly, measuring reliability required having a person intervene and terminate infinite loops since one can't guarantee whether a program is just still running or if it really is infinitely looping. This meant it took a person with judgement about how long the test framework should run for a given set of iterations.

Secondly, it seemed like measurements taken on the lnxsrv varied a lot depending on what time tests were ran. This is likely because of load from other students running their code. So, I compiled my tests into a data generation script that I could run during the day when other students were likely in class.

7 References

1. Oracle and/or its affiliates, The Java Tutorials specifically the section on synchronization

2. Doug Lea, Using JDK 9 Memory Modes, link