Java: Performance and Reliability of Different Multithreading Methods

Abstract

Given a synchronized Java class Synchronized that is called, accessed, and updated by multiple threads, we are to find alternative implementations of the class that allow the Java program to run faster while making sure that it remains "data-race free" (DRF) when accessing/modifying shared memory following the Java Memory Model.

1. Experiment Results

The test cases were done on with the an array of 64 integers all initialized to 64 and a maximum value of 127, on the server lnxsrv10.seas.ucla.edu. The server comprises of 65GB of RAM and 4 Intel(R) Xeon(R) Silver 4116 CPUs with clock speeds of 2.10GHz, each with 4 cores. Both Java 9 and 11.0.2 were used, and the units of the measurements is ns/transition. The results are shown below.

Synchronized (version 9)		Number of Swaps		
		10^4	10^5	10^6
Number of Threads	5	6153.73	2053.23	677.955
	10	9352.54	5338.51	1203.09
	20	124169.9	9513.55	2250.92
	40	54854.8	17465.8	7168.90

Synchronized (version 11.0.2)		Number of Swaps		
		10^4	10^5	10^6
Number of Threads	5	4827.48	1720.58	547.620
	10	14400.7	4015.53	991.043
	20	23808.1	8403.65	2916.87
	40	65514.8	18568.8	5901.45

Unsynchronized (version 9)		Number of Swaps		
		10^4	10^5	10^6
Number of Threads	5	5622.11 (mismatch)	1958.19 (mismatch)	733.174 (mismatch)
	10	10785.2 (mismatch)	2516.08 (mismatch)	2189.64 (mismatch)
	20	23135.6 (mismatch)	4198.39 (mismatch)	3518.96 (mismatch)
	40	44636.0 (mismatch)	13026.8 (mismatch)	5222.82 (mismatch)

Unsynchronized (version 11.0.2)		Number of Swaps		
		10^4	10^5	10^6
Number of Threads	5	3820.84 (mismatch)	1247.47 (mismatch)	671.826 (mismatch)
	10	14287.3 (mismatch)	3049.02 (mismatch)	2183.05 (mismatch)
	20	30564.7 (mismatch)	6638.64 (mismatch)	6538.03 (mismatch)
	40	65380.4 (mismatch)	13361.1 (mismatch)	5798.37 (mismatch)

GetNSet (version 9)		Number of Swaps		
		10^4	10^5	10^6
Number of Threads	5	6056.07 (mismatch)	2904.08 (mismatch)	869.300 (mismatch)
	10	17431.0 (mismatch)	4162.30 (mismatch)	2054.99 (mismatch)
	20	28835.5 (mismatch)	10508.9 (mismatch)	5139.44 (mismatch)
	40	75648.4 (mismatch)	19905.5 (mismatch)	7352.78 (mismatch)

GetNSet (version 11.0.2)		Number of Swaps		
		10^4	10^5	10^6
Number of Threads	5	7782.79 (mismatch)	1885.22 (mismatch)	783.919 (mismatch)
	10	28609.4 (mismatch)	7101.52 (mismatch)	2746.86 (mismatch)
	20	56485.5 (mismatch)	10323.0 (mismatch)	9966.53 (mismatch)
	40	106558 (mismatch)	17723.0 (mismatch)	20652.5 (mismatch)

BetterSafe (version 9)		Number of Swaps		
		10^4	10^5	10^6
Number of Threads	5	8038.74	1273.17	431.633
	10	11737.2	2911.49	909.977
	20	29017.7	6017.50	1757.12
	40	89607.7	15787.1	4078.95

BetterSafe (version 11.0.2)		Number of Swaps		
		10^4	10^5	10^6
Number of Threads	5	8684.97	1446.73	437.565
	10	19088.7	3138.80	895.416
	20	53082.3	7593.70	1893.86
	40	117682	20203.6	5000.20

1.1. Analysis of Results

As far as the performance goes, Synchronized is on the most part slower than Unsynchronized and BetterSafe. Because the synchronized keyword is implemented with a spin lock, it would be expected that each thread would still consume processing power even though they are on hold. Unsynchronized is definitely the fastest, as there is no waiting or locking of any form. BetterSafe, implemented with waiting and queuing, is of course expected to be faster as well. However, GetNSet is slower probably because of failing writes to the shared array.

For reliability, Synchronized and BetterSafe are both 100% DRF, as they both make sure that there is no interrupting reading/writing taking place between multiple threads, which can interfere with the correctness of the operations taking place in each thread. Unsynchronized and GetNSet's final checksums ranged from just a little off too a few hundred units off.

For the performance benchmarks for each Java versions, for Synchronized, the program definitely ran faster on Java 9 than Java 11.0.2, but for the rest, there is much variation, and on average there doesn't seem to be too much of a difference.

1.2. Best Choice

All in all, as far as reliability goes, only Synchronized and BetterSafe should be considered. The other two classes can potentially have huge errors (large difference from checksum). As far as performance goes, BetterSafe offers a small amount of additional performance to utilize. Unsynchronized offers the most performance gain (of course, because no waits or locks), but the potential error may not be worth it. GetNSet cannot be implemented in a way that the conditional checking and atomic swapping can all be done atomically, so permanent swap failures may occur, which causes a hit in performance.

2. Bettersafe Implementation Considerations

As given by the assignment specification, the 4 packages and classes given to consider implementing BetterSafe with in this assignment are "java.util.concurrent", "java.util.concurrent.atomic", "java.util.concurrent.locks", and "java.lang.invoke.VarHandle". The specific details of the considerations are given below.

2.1. Details to Consider

In the package "java.util.concurrent", two options given are classes for locks and atomic methods. As mentioned before, atomic methods cannot do much good, as we need to make the "range-check" and update methods atomic together. The error case with this is when the conditional checking is done first, then another thread interrupts and changes the values to swap to invalid integers outside of the range 0-maxval. In order to make the two atomic together, we might as well just use a lock to make the entire function atomic.

As for "java.lang.invoke.VarHandle", the VarHandle class stores reference to variables and provides atomic methods along with access modes for read/write control. However, atomic methods can fail because both reading and writing are involved (compare-and-set methods wouldn't be effective if another method were to change the value to compare beforehand). Access control wouldn't do too much (volatile access doesn't make the entire read-and-write operation DRF, and release/acquire access is basically the same as a lock). So, it would be much simpler to just a lock on the sequential read-and-write method swap.

2.2. How BetterSafe Compares with Synchronized

By simply locking the sequential read-and-write operation swap without using a spin lock like using synchronize, BetterSafe would do the same thing as Synchronize without having the waiting threads eat up resources unnecessarily.

3. Problems Encountered

While working on the assignment and testing the code, I ran into two main problems: invalid experimental results and some troubles with the code.

3.1. Faulty Results under Lagging Server

I had tested my code a few times on the school's Linux server, and when I was recording the data to present, I didn't realize that there was much traffic with the server, meaning that the code ran differently under limited resources. The faulty results I had obtained while the server was busy was much different than the normal. The times per transition were on average 3-6 time longer/slower, and there

was little differentiation between each of their times (each time/transition results were somewhat close given the same number of threads and swap operations). Most of the time, Java 9 actually ran slightly faster than Java 11.0.2, and the classes Unsynchronized and GetNSet only failed the checksum about 2/9 of the time under Java 9 and about 4/9 of the time under Java 11.0.2 on average (compared to the 100% checksum fail rate under normal conditions).

3.2. BetterSafe Locking Delays

One implementation that I tried was having multiple locks to lock each individual integer of the array, and another lock to make the locking atomic. However, I found that the code ran slightly slower than with synchronized. This was probably likely due to delay from the frequent reading from / writing to the locks, as opposed to simply reading from / writing to a single lock each time for each iteration.