CS131 Java Shared Memory Performance Races

I. Introduction

A. In this assignment, I analyzed and studied the Java Memory model which aids in applications safely avoiding data races when dealing with shared memory. I looked at different methods and attempts to optimize access to variables and data such as *synchronized*. *locks*, and *volatile variables.*

II. Classes Overview

B. Synchronized

1. This class uses the synchronized keyword which makes sure that only one thread will be allowed to use the method. In short, synchronized can be summarized by mutual exclusion: only one thread can enter the block at once, meaning that once one thread enters a synchronized block, no other thread can enter a block until the first thread exits the synchronized block.
2. This is safe, but can be slow since threads have to wait on other threads to use the same method.

C. Unsynchronized

1. The Unsynchronized class does not use the synchronized keyword meaning that there is a possibility that different threads will have access to data at similar times and are not mutually exclusive. Although there were cases where everything worked fine, many cases caused a data race since the sum was mix-matched and threads were changing the execution results.

D. GetNSet

1. The GetNSet way consisted of using volatile variables where each read of a volatile will see the last write to that volatile by any thread. However, this did not necessarily mean it was safe. The implementation enforces updates to its value between threads.

E. BetterSafe

1. For BetterSafe, I decided to use the ReentrantLock mode. I chose this because it was reliable and simple to lock the operations to make them atomic and not cause a data race. The cons that one has to keep in mind when using locks is the case of deadlock, but this can be avoided if implemented correctly.

III. Testing Methods

To test the different classes, I created a list of test cases ranging in number of threads, swaps, and max int values. The tests were also run in different versions of Java. Below is an example of some test case results using Java 9. The number of threads range from 1, 2, 4, 8, 16. The number of swaps range from 10^4, 10^5, and 10^6. In this cluster of test data, the max int value remained consistent at 127. The tests were also run on the Null class for comparison.

Test Results

Array input: 2 4 6 8 10

MaxVal: 127

∗ indicates sum mismatch (Data Race)

		# Swaps		
Class	**#Thread**	**10000**	**100000**	**1000000**
Synch	1	615.591	230.779	70.4559
	2	1845.33	807.909	265.293
	4	4208.24	2402.89	1096.28
	8	8422.92	4351.06	1997.75
	16	17259.7	8344.60	4338.10
Unsych	1	633.708	216.782	54.2092
	2	**1262.38***	**477.368***	**334.271***
	4	**3632.25***	**876.182***	**468.117***
	8	**7801.10***	**1830.84***	**FAIL**
	16	**12596.7***	**3397.67***	**FAIL**
GetNSet	1	781.935	236.162	71.2148
	2	1733.22	582.618	263.611
	4	4823.57	1980.02	695.261
	8	9944.64	2899.41	1434.34
	16	19459.2	5033.99	3661.99
BetterSafe	1	973.486	263.825	96.1979
	2	3935.35	1378.94	333.663
	4	6227.62	2683.86	1444.08
	8	14018.2	5473.97	3139.70
	16	28930.7	10515.7	5388.23

		# Swaps		
Class	**#Thread**	**10000**	**100000**	**1000000**
NULL	1	587.333	203.701	42.1517
	2	1224.34	455.706	134.089
	4	3351.92	781.076	499.017
	8	7747.97	1497.08	2474.29
	16	13572.1	3183.52	4787.31

IV. Test Results and Analysis

F. As seen in the tests, the fastest implementation was Unsynchronized followed by: Synchronized, GetNSet, Synchronized, and finally BetterSafe.

G. My BetterSafe implementation was not faster than Synchronized. However, it was not slower by much especially when using 1 thread for 1000000 swaps, and as the name suggests it would be better to use in the long run.

V. Problems

H. While testing, the biggest problem was related to the Unsynchronized class, where it would fall into an infinite loop and hang. I also had to run the tests many times to make sure that the results were not affected by the servers being heavily used.

1. An example of a test where a class failed was: is java UnsafeMemory Unsynchronized 16 1000000 127 2 4 6 8 10

VI. Conclusion

Different multithreading implementations were tested and explored in this assignment. The results were not exactly as I expected with BetterSafe as the slowest, followed by Synchronized, GetNSet, and Unsynchronized as the fastest. SynchronizedState and BetterSafe were completely reliable and Data Race free. However, Unsynchronized had many cases of data races and failed many trials. GetNSet produced what seems to be reliable results, but it may have possibility of a data race.

In conclusion, the best choice out of these classes is BetterSafe. The results of the tests were very similar to GetNSet, but is much more reliable and data race free.