CS 131 Homework 3 Report

Jonathan Chu
004832220

1. Introduction

The Java Memory Model is a set of rules describing how a program can access shared memory while avoiding data races. The "synchronized" keyword allows users to make any method thread-safe in all of its reads and writes to shared memory. However, there exist a number of ways to avoid data races in Java, some of which exhibit better performance than others and some not 100% data race free (DRF). In this homework we explore the pros and cons of different options for making shared data thread-safe in Java. We will compare four different implementations: first using the synchronized keyword, second using no synchronization at all, third using volatile variables and get() and set() library functions, and finally using a lock.

2. Implementation

There were different classes/packages available to make BetterSafe faster than SynchronizedState but still completely safe.

2.1 java.util.concurrent

To make BetterSafe faster than Synchronized but still 100% DRF, I used java.util.concurrent.locks because the only feature we need to make the class thread-safe is a single lock in order to protect reads and writes to the array. The lock provided by the locks class provided a fine-grained, straightforward way to make the program thread-safe. The atomic class was not necessary because there were no operations we needed to be atomic, other than the locking and unlocking operations which are made atomic by the locks class itself.

2.2 VarHandle

The VarHandle class supports plain reads/writes, volatile reads/writes, and compare-and-set. This class would not have been useful because plain reads/writes are not protected at all, volatile reads/writes are partially protected but still not DRF, and compare-and-set does not ensure single-thread access to shared data. Thus the VarHandle class could not be used to directly create a safe implementation, at least not in as straightforward an implementation as using a simple lock.

3. Results

The classes were tested systematically with a shell script using [1, 2, 6, 12, 20] threads, [10000, 100000, 1000000] swaps, and a maximum of 127. The elements were initialized to 1, 2, 3, 4, and 5.

Class	#Threads	#Swaps		
		10^4	10^5	10^6
Synchronized	1	591.486	195.322	63.6498
	2	2252.14	1156.77	240.952
	6	8008.78	3481.39	2306.44
	12	13633.1	5648.56	3501.10
	20	25366.5	9420.52	5729.95
Unsynchronized	1	574.285	196.879	51.8124
	2	**2031.78**	**934.745**	FAIL
	6	**5194.00**	**1566.10**	FAIL
	12	**15006.9**	**2761.31**	**2705.66**
	20	**22286.9**	**5348.31**	**5491.36**
GetNSet	1	1063.08	244.475	73.5783
	2	**2149.50**	**1505.26**	FAIL
	6	**15115.5**	**2446.22**	FAIL
	12	**38495.9**	**5947.58**	FAIL
	20	**54567.3**	**9121.31**	FAIL
BetterSafe	1	924.986	261.018	83.6808
	2	3436.37	1530.23	771.993
	6	9941.16	3966.76	824.746
	12	19839.7	5250.30	1860.85
	20	41163.8	8949.67	5706.03

Values in the table are in nanoseconds. Bolded values are results that were inconsistent (i.e. there was a data

race), and FAIL is written for trials that caused the program to loop infinitely.

3.1 Performance Analysis

The unsynchronized was clearly the fastest because it wasted no time with extra volatile checks/updates or with locks. GetNSet was next in speed because volatile variables made it slower than Unsynchronized but still faster than locking. Synchronized performed better than BetterSafe in some tests, particularly when there were smaller numbers of threads and swaps, but BetterSafe is clearly more scalable, beating Synchronized in performance at higher numbers of competing threads and more swaps. Thus, BetterSafe does outperform synchronized in the long run.

3.2 Reliability Analysis

Synchronized and BetterSafe were 100% DRF in every trial, as expected. However, Unsynchronized and GetNSet were visibly not DRF. Both classes were only reliable under tests with just one thread, since there was no possibility of shared access in these trials. These non-DRF classes, however, should be expected to fail when so many operations occur (10,000+), so there was not much of a surprise in these results. GetNSet was implemented using volatile variables, but declaring a variable volatile does not make it thread-safe – it merely enforces updates to its value between threads. There were a few trials, all with 10^6 swaps, that looped infinitely and caused the terminal to hang, but these were in the non-DRF classes, where shared access was not protected, and swaps could have occurred simultaneously.

4. Difficulties

The code-writing portion of this homework assignment was relatively straightforward and involved few changes to the existing Synchronized State class. BetterState required some reading, but given the simple problem statement and functionality of the class, it was fairly easy to determine which protection mechanism to use. What took a little longer was testing each class systematically because some test cases would cause the terminal to hang. At first, I thought my implementations for Unsynchronized and GetNSet were incorrect because I was only running tests with 1 million swaps, so I spent a good deal of time looking for problems there, but Piazza showed me that the lack of mutual exclusion was what was leading to the infinite looping. It wasn't too difficult to figure out a way around the infinite looping in my shell script, once I understood the problem.

5. Conclusion

Through this assignment, we explored different methods of implementing safe multithreading and also the performance implications that came with these methods. After running the test cases, I mostly obtained the results I was expecting. SynchronizedState was the slowest, followed by BetterSafe , GetNSet, and finally Unsynchronized as the fastest. SynchronizedState and BetterSafe were 100% DRF, as they were supposed to be, while Unsynchronized was full of data races and failed every trial. GetNSet similarly produced incorrect results, eventually, but we expect that on each individual run, GetNSet would last longer producing correct results than Unsynchronized because it is at least somewhat mutually exclusive.

From these results, the best class for GDI is probably BetterSafe. Its performance was close to that of GetNSet, but GetNSet was infinitely looping for many of the more work-intensive trials. Depending on how intensive GDI's applications will be, GetNSet might be a good choice, but from these results and the workload tested, BetterSafe is more suitable. Overall, this assignment was a great opportunity to apply material learned in class and see practical results in real time.