CS 131 Homework 3 Report

Introduction

This assignment has us working for Ginormous Data Inc (GDI). Our "boss" is asking us to speed their company's code which uses the Java keyword "synchronized" to protect their program from race conditions.

As programmers, it is our job to form an understanding of the Java Memory Model, creating new classes which, while more unreliable, can run the code much faster. We can then use our results to come to a conclusive answer for which type of thread protection GDI should use. We must implement Unsynchronized (unreliable, but quick), GetNSet (uses atomic volatile changes), and BetterSafe (implements locks).

Performance Analysis

Below are tables containing test data using each of the implemented classes. These results were found using a shell script which ran each class various numbers of times. For this table, the amount of swaps was fixed at 10,000 and only the number of threads were changed. While some classes took a few attempts (hanging issue addressed in 'Issues' section), most ran smoothly. Each of the recorded values are in nanoseconds.

Class	Thread Count (Swaps Fixed at 10,000) (Results in nanoseconds)						
	1	2	4	8	12	16	24
Null	578.426	1524.32	3846.92	6658.81	13274.1	13314.2	28740.8
Synchronized	683.040	2173.62	5307.13	10503.2	15825.7	41173.2	73636.5
Unsynchronized	580.787	1826.58	3714.66	7922.45	11219.8	14501.5	18813.6
GetNSet	1052.78	2731.03	9313.39	13371.7	20668.8	25148.3	44458.2
BetterSafe	1032.45	3720.46	6760.96	13530.9	20732.4	26636.9	48473.6

For the following table, thread count was kept at a fixed amount of 8 while the number of swaps were changed. Just as before, a few of the classes had to be fun multiple times due to hanging the terminal, but after a few attempts, the data was able to be collected and recorded. Again, each of the values within the table are recorded in nanoseconds.

Class	Swap Count (Threads Fixed at 8) (Results in nanoseconds)		
	1,000	10,000	100,000
Null	29861.4	6658.81	1657.23
Synchronized	52399.9	20743.2	8731.92
Unsynchronized	27925.6	6560.86	1644.91
GetNSet	44348.6	15496.2	3182.62
BetterSafe	85820.8	14273.3	4267.39

From the data in the tables, we can tell that the fastest class is Unsynchronized. This should be the case because it is taking no precautions to keep the threads safe from race conditions. The next fastest GetNSet, followed by Synchronized (at lower thread counts, addressed in ('BetterSafe Vs. Synchronized').

When the tests were run using Java 9, the numbers were nearly the same. The correlations detailed in this section were identical. The differences in performance and reliability seem to be negligible for these classes.

Reliability Analysis

For this case, reliability comes in the form of having a zero sum and not hanging the terminal.

Unsynchronized tops the other classes in unreliability. It provides no protection from race conditions and has failed every sum test with more than 1 thread. The unsynchronized class was also difficult to obtain data for. This class very often hung the terminal which made me have to run multiple tests to fill out the tables above. This class resides on the side of the spectrum with high performance and low reliability.

GetNSet, surprisingly, offers a similarly low reliability rate. Despite using volatile accesses to the array, GetNSet has been shown to have sum inconsistencies with more than 1 thread as well. It should be noted however, that the difference in the original sum and the end sum are much closer with this class than with the unsynchronized class. This class, to a smaller degree, also hung the terminal.

This made testing the class difficult as multiple tests had to be run to obtain data at all.

Lastly, the better safe was reliable 100% of the time. For every test, BetterSafe showed no sum errors and never hung the terminal. The reliability of this class was the same as the Synchronized class which will be examined in more detail in the 'BetterSafe Vs. Synchronized' section.

Concurrent Vs. Atomic Vs. Locks Vs. VarHandle

The first presented package is java.util.concurrent. The pro to using this would be a large library with many different implementations. Unfortunately, that is its con as well. Its size provides multiple frameworks which are not straightforward to implement. In its documentation, it even states, "This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement."

The second package would be a subpackage of the first (making it more straightforward to implement). java.util.concurrent.atomic provides the same interface which was used for GetNSet. Its class is incredibly easy to implement which is its main advantage. While it does provide some reliability, it is still not 100% reliable due to concurrent swaps. For this reason, I chose not to use this package.

The third package java.util.concurrent.locks is the one I ended up utilizing for BetterSafe. It provides a clean interface which locks critical sections of code and blocks other threads from changing shared data. This provides 100% reliability which is a major advantage. Another advantage is its use for Project 2 of CS111. This gave me experience with using locks. A con to this package would be the possibility of deadlock if locks are not released correctly.

The last package provided to us was java.lang.invoke.VarHandle. This package allows different types of access to variables including volatile data and compare-and-set methods. This package also does not provide 100% reliability. It doesn't guarantee that critical sections remain untouched by other threads. Just as with the second package, this means this is not the optimal package for BetterSafe,

BetterSafe Vs. Synchronized

As you can see in the tables above, BetterSafe does run slower than the Synchronized class with low thread counts. However, at higher thread counts, BetterSafe becomes much faster. This means that while Synchronized has a higher performance initially, BetterSafe scales better with larger amounts of threads. This is useful for GDI, our employer, as the specs state they often use high amounts of threads for their operations.

My implementation for BetterSafe was heavily inspired by Lea's paper. The 'Locks' section directly mentions Reentrant Locks and how to use them. By locking the critical code sections in the swap method of BetterSafe, other threads which want to swap on the array are blocked. This guarantees that BetterSafe is a DRF method.

Issues

The main issue with measuring performance was due to many of the classes hanging with usage by more than 1 thread. Huge offenders of this were Unsynchronized and GetNSet. To produce the data for the figures. I ran the tests multiple times until 1 trial ended up going through.

The only classes that are data-race free are Synchronized and BetterSafe. Synchronized is DRF due to using the "synchronized" Java keyword. This prevents other threads from calling that method while it is already being called by a different thread. This guarantees all other threads will see the changes made preventing race-conditions.

BetterSafe on the other hand, uses Reentrant locks to protect critical code sections. I run into a slight issue when implementing this. In the case that swap were to return false, I originally had forgotten to unlock the lock. This would cause deadlock in my program and took a bit (more than I would like to admit) of time to figure out. After finding and fixing the bug, BetterSafe became DRF due to other threads not being able to interrupt changes to shared variables.

Conclusion

In conclusion, GDI should implement the BetterSafe class due to its high reliability and better scalability than Synchronized. Due to going through so much data, Unsynchronized and GetNSet would be bad choices as they offer low reliability. While they are much faster, both have been shown to have sum errors when running with more than 1 thread. This would definitely have a huge negative impact on the quality of GDI's work and would not be worth the performance increase.