Homework 3 Report

1 Introduction

This homework explores the Java memory model (JMM), which is a Java synchronization tool that specifies how threads interact through memory. The synchronized keyword in Java allows the states of memory to be updated safely without race conditions. However, guaranteeing race-free code often sacrifices performance; in fact, this is usually the overhead bottleneck in multithreaded Java programs. There are different options for multithreaded programs to interact with the same memory, some with high performance but not completely DRF (data race free) and some that guarantee DRF behavior but have slower performance. This homework compares programs that utilize the synchronized keyword, those that don't and are unsynchronized, those that utilize volatile variables, and those that use explicit locking mechanisms.

2 Packages

Many packages were considered and used to implement the homework assignment. I chose java.util.concurrent.locks to implement BetterSafe. To explain my decision, each subsection explores the pros and cons of using that package for implementing the BetterSafe class.

2.1 java.util.concurrent

There are many sub-packages under the java.util.concurrent package, including queues, hashing, collections, as well as critical section protection that utilizes semaphores and countdown latches. Although these packages can be utilized to create a safe solution, they are extremely advanced and may be overkill for simply protecting small modifications to an array of values. An advantage is a rich set of solutions to providing DRF programs but the disadvantage is the complexity of these solutions compared to simply using locks.

2.2 java.util.concurrent.atomic

The java.util.concurrent.atomic package offers atomic versions of existing Java types. This package is advantageous because classes for atomic values and arrays are provided that support incrementing these atomic values and allowing volatile access semantics for elements of arrays. However, a major disadvantage is that these classes do not guarantee a race-free outcome in a multithreaded environment. The package only guarantees that the elements within the array are updated between threads. This claim is validated by observing *Table 1* and *Table 2*, which show that race conditions happen with the GetSetN class, even though its underlying data structure is an AtomicIntegerArray. Furthermore, in order to convert an array of bytes to an AtomicIntegerArray, an O(N) operation of copying each individual element of the byte array must be performed. Likewise, converting the atomic array back into an array of bytes also takes O(N). The failure to uphold safety within a multithreaded context and the performance overhead of converting the underlying type contradict with the reliability and performance guarantees that the BetterSafe class should promise; thus, java.util.concurrent.atomic was not chosen to implement BetterSafe.

2.3 java.util.concurrent.locks

In order to exceed the performance of the Synchronized class but maintain the guarantee of DRF, I implemented the class BetterSafe with java.util.concurrent.locks. I simply needed to wrap read and write operations to the byte array in memory with lock() and unlock() methods in order to guarantee a correct multithreaded behavior of the program. The advantage of using locks is simplicity, as there is no need to create a new array type to represent atomic integer values. This also reduces the overhead of the O(N) operation of converting a byte array to an AtomicIntegerArray. The disadvantage of using locks is cluttering the code with lock() and unlock() function calls instead of allowing a sophisticated library to handle the underlying implementation of guaranteeing DRF code. However, this clutter does not severely affect a small program with few operations. Because we are only reading and writing to a single byte array, this package was chosen to build an efficient but safe solution. According to Doug Lea's paper Using JDK 9 Memory Order Nodes, safety is guaranteed, as "correct use of mutual exclusion locks such as… ReentrantLock maintains total ordering of locked regions." Lea further explains that locking provides "roach motel" rules because reads and writes inside a region can't move out. See the Analysis section to see the exact performance advantages that BetterSafe with locks has over the provided Synchronized solution.

2.4 java.lang.invoke.VarHandle

The java.lang.invoke.VarHandle package is a relatively complete solution to keeping multithreaded programs race-free. Due to the large number of features, this solution has a steep learning curve and provides a heavyweight solution to guaranteeing DRF code. According to Lea's paper, "VarHandle memory order modes…can be described in terms of constraints layered over… access rules: Opaque mode… Release/Acquire mode… [and] Volatile mode." The biggest advantage of VarHandle is the provided framework for supporting multiple modes of access. However, the disadvantage is the complexity and learning curve of the solution, especially in the context of modifying a couple of values in a single array, as well as the overhead of using this solution compared to lightweight, performant locks.

3 Analysis

Each class was tested with a Bash shell script. The script tested with [1, 2, 4, 8, 16] threads, [10000, 100000, 1000000] swaps, a maximum byte value of 127, and an array with values initialized to [6, 5, 6, 3, 0, 3]. Here is a table summarizing the results using openjdk 11.0.2 2019-01-15, OpenJDK Runtime Environment 18.9 (build 11.0.2+9), and OpenJDK 64-Bit Server VM 18.9 (build 11.0.2+9, mixed mode):

Java Version: 11.0.2				
Class	Threads	Swaps		
		10000	100000	1000000
Null	1	594.754	197.809	43.6579
	2	1792.56	655.992	153.332
	4	3540.68	1076.32	528.352
	8	7294.49	1464.35	1994.97
	16	15120.9	3390.87	4480.77
Synchronized	1	824.269	224.911	70.2987
	2	2740.33	1182.78	691.825
	4	5376.48	2661.17	1220.57
	8	12003.5	6301.63	2586.92
	16	24777.2	7490.6	5143.49
Unsynchronized	1	669.653	221.977	54.9557
	2	***2194.45***	***641.416***	INF
	4	***4486.43***	INF	INF
	8	INF	INF	INF
	16	17378.8	INF	INF
GetNSet	1	1047.6	292.451	76.1699
	2	***2903.34***	926.686	***279.006***
	4	INF	INF	INF
	8	INF	INF	INF
	16	***24899.6***	***5587.95***	INF
BetterSafe	1	1143.56	296.415	92.477
	2	4307.11	2044.98	626.08
	4	8147.78	2473.87	701.869
	8	20316	5328.45	1348.19
	16	40874.7	10922.1	3352.48

Table 1: The average ns/transition results from running the test harness on Java 11.0.2 for various class, thread count, and swap count combinations. The ***bolded and italicized*** elements represent the cases where a sum mismatch (race condition) happened. The INF cells represent instances where the code was stuck in an infinite loop.

Here is another table summarizing the results using openjdk 9, OpenJDK Runtime Environment (build 9+181), and OpenJDK 64-Bit Server VM (build 9+181, mixed mode):

Java Version: 9				
Class	Threads	Swaps		
		10000	100000	1000000
Null	1	474.218	174.058	44.8117
	2	1572.14	487.815	142.95
	4	3376.69	735.053	498.728
	8	6966.15	1306.55	2165.32
	16	12579.2	2955.06	3595.36
Synchronized	1	664.586	220.628	71.0242
	2	2001.32	1129.94	561.098
	4	4709.86	3140.16	1374.15
	8	9522.88	5184.87	2986.45
	16	18735.4	10870.2	5747.1
Unsynchronized	1	600.745	206.914	55.2996
	2	***1682.79***	***634.669***	INF
	4	INF	INF	INF
	8	***7880.89***	***2420.68***	INF
	16	***14594***	INF	INF
GetNSet	1	843.603	255.437	71.9278
	2	***2413.65***	***769.358***	***254.77***
	4	INF	INF	INF
	8	***10565.2***	INF	INF
	16	***24244.6***	INF	INF
BetterSafe	1	1072.28	260.509	85.559
	2	2591.25	1289.7	610.892
	4	6694.24	2306.01	621.025
	8	15277.6	5162.46	1279.99
	16	30651.7	9310.55	2543.85

Table 2: The average ns/transition results from running the test harness on Java 9 for various class, thread count, and swap count combinations. The ***bolded and italicized*** elements represent the cases where a sum mismatch (race condition) happened. The INF cells represent instances where the code was stuck in an infinite loop.

Each test was performed with these specifications:

Specifications	
Server	lnxsrv07
Vendor ID	GenuineIntel
CPU	Intel(R) Xeon(R) CPU E5-2640 v2
Frequency	2.00 GHz
Cores	8
Main Memory	65755884 kB

Table 3: The CPU and memory information of the testing platform used so that others can reproduce the results.

3.1 Performance

The results from Java 11.0.2 and Java 9 are similar, so the performance comparison analysis applies to both versions of Java. Null and Unsynchronized were the most performant. This is expected because no overhead from guaranteeing DRF code is implemented, as Null does not read or write memory and Unsynchronized does no work to maintain multithreaded correctness. Synchronized is faster than both GetNSet and BetterSafe with low thread and swap parameters, but is slower when these parameters are increased. It was difficult to compare GetNSet with the other classes due to how often infinite loops happened. BetterSafe is more performant than Synchronized when the thread and swap counts are high. This means that BetterSafe is more efficient with larger amounts of data and is thus a more scalable and performant DRF solution over Synchronized.

3.2 Reliability

The results from Java 11.0.2 and Java 9 are similar, so the reliability comparison analysis applies to both versions of Java. Unsynchronized and GetNSet were not DRF. Specifically, here are reliability tests that these classes are extremely likely to fail on:

Unsynchronized:
java UnsafeMemory Unsynchronized 16 1000000 6 5 6 3 0 3
GetNSet:
java UnsafeMemory GetNSet 16 1000000 6 5 6 3 0 3

These commands were chosen because race conditions are more likely when the thread count and the number of swaps is high. The more threads that fight over a single data structure and the more swaps that each thread performs, the higher the likelihood of two threads reading and writing to the same value. Although GetNSet utilized a volatile Integer array, this did not guarantee safety within a multithreaded context but rather only ensured that updates to the values happen between the threads. Some instances of GetNSet were stuck in an infinite loop; this was due to its non-DRF nature. In the case where shared access to the array was possible, swaps happening simultaneously caused the array to consist only of zeros and maxvals. Thus, the swap always fails and the program never makes progress. On the other hand, Null, Synchronized, and BetterSafe were completely DRF. Null is DRF because no reads or writes to memory were done, so there is no opportunity for a race condition to occur. Both Synchronized and BetterSafe are DRF because of correct locking mechanisms that protect the access and modification to memory. Specifically, "roach motel" rules were applied, as reads and writes to the byte array were always performed by one thread at a time by locking the resource before the access and freeing the resource once finished.

4 Problems

Some problems arose when it came to measuring each class' performance. Specifically, infinite loops would occur and leave the program hanging on certain (seemingly random) test cases. At first, this was confusing and difficult to debug; I initially thought there was a problem with my code. However, after examining the run() function in the SwapTest.java file, I realized that race conditions could cause an infinite loop when the swap function continues to return false. This happens when the thread that increments a position modifies its value before the threads that decrement. Once I understood this race condition, it became easier to understand the results of the measurements and to modify the parameters accordingly. Given the unpredictability of how an unsafe, multithreaded program would operate, I simply noted when the unsafe programs got caught in an infinite loop instead of treating the issue as a problem with the code.

5 Conclusion

This experiment compared the performance and reliability of different methods that implement both safe and unsafe multithreaded programs. The results show that only Null, Synchronized, and BetterSafe were DRF, while Unsynchronized and GetNSet produced unsafe behavior. Furthermore, the test harness demonstrated that of the DRF solutions, BetterSafe was faster than Synchronized with higher threads and higher amounts of swaps, and Synchronized was faster with smaller threads and smaller amounts of swaps. Null was the fastest but did not perform the actual task, so this is ignored when considering which class to use for GDI. Of the non-DRF solutions, Unsynchronized was more performant than GetNSet, but both were plagued with many race conditions. From the experimental results, the best class for GDI is BetterSafe, because GDI prioritizes performance as a design goal over large amounts of data. Although GetNSet is slightly more performant, the error rate due to too many race conditions is too high and GDI is only willing to put up with a few errors. The time saved is most likely not worth the large number of incorrect results.