Java Synchronization Report

Kevin Zhang
104939334

Abstract

The Java Memory Model defines how an application processes and allocates memory. A large task designated to the Java Memory Model is its responsibility of safely avoiding data races during a programs run time when the program accesses a shared memory space. The model allows Java to optimize accesses to memory using threads in a schedule that assumes sequential consistency. There exists a number of ways to avoid data races in Java and in this homework, we write a number of different programs with different implementations of a the same class, with identical functionality in an attempt to first test out sequential consistency and compare the pros and cons of different options for making a program DRF(Data Race Free).

1 Implementation

1.1 BetterSafe

Java.util.concurrent.locks

The goal of our implementation was to be DRF without the high overheads of synchronizing a method. We only required the class to be thread-safe since we didn't have any other operations other than reading and writing and our main goal was to keep a region mutually exclusive which is exactly the functionality of locking(making a region serializable). This package also allows lock coarsening which combines two adjacent locked regions which is relevant to our implementation since our thread-safe regions are adjacent and similar. This prevents interleaving of instructions that is the main cause of error and does away with the overhead of synchronizing an entire method but instead just locks the portions that we need.

1.2 GetNSet

java.util.concurrent.atomic

GetNSet used the atomic concurrent library that provided an object implementation for AtomicIntegerArray. This effectively made any operations on the internal private array atomic, meaning a set or get operation would either go to completion or fail regardless of what happened during the call. This stops interleaving of specific updates to the array. that produced overhead in exchange for synchronization. GetNSet proved to be around the same time per transition but became slower than unsynchronized as the number of swaps However, making the operations atomic would not change the behavior when threads interleave after the bounds of the values are checked. It has higher performance as a result because it doesn't have the synchronizing overhead but it also doesn't have full DRF. Using atomic integer arrays only solve a portion of the problems making it, as the spec points out "halfway between unsynchronized and synchronized." In summary, using this package removes a high performance overhead but does not solve the problem of synchronization fully.

1.3 Unsynchronized

The implementation of Unsynchronized merely attempted to break the sequential consistency by removing all checks for synchronized behavior including the synchronized keyword. Removing the synchronized keywords removes all attempts to keep the original State class DRF and just allows it to run by itself.

2 Alternative Packages and Classes

2.1 java.util.concurrent

The java.util.concurrent package has many implementations for different types of mechanisms that can free a program of data races completely. However, most of these mechanisms were more than what this particular State Class required. All we needed to remove our class's data races is synchronizing each thread's interactions with the given resources. Because we wanted to make the class as fast as possible, most of the tools in the class were irrelevant.

2.2 java.lang.invoke.VarHandle

The java.lang.invoke.VarHandle class provided a variety of modes including plain mode, opaque mode, and release/acquire mode. Most of the functionality provided was relevant to DRF discussions but added unnecessary complexity. For example, RA mode attempts to keep the program causally consistent so that there are no cycles in the orderings. This is more relevant to message-passing designs that is of higher complexity than what we are dealing with here. Furthermore, the Opaque mode which is of a similar complexity as locking does not ensure specific ordering. It tries to guarantee coherence by making visible overwrites ordered but it uses a similar implementation to Locking using Thread.onSpinWait and Thread.yield to wait for a thread to finish. I chose to use the simpler locking since it avoided functionality that we didn't need to use. In summary, plain mode guaranteed bitwise atomicity which we already had from GetNSet. Opaque operations guaranteed a coherent ordering with respect to accesses to the same variable which I was not faster than locking when run on the test cases I used which caused me to settle with ReentrantLock as the best option to implement.

3. Results

These classes were each tested with this system call
java UnsafeMemory [Class] [Threads] [Swaps] 127 3 7 35 9 4

Class	#Threads	#Swaps		
		10^4	10^5	10^6
Sync	2	6682.55	2132.13	581.142
	4	33444.3	9839.43	918.931
	8	152056	12880.0	2539.72
	16	472124	52973.2	7514.62
Unsync	2	1101.96*	1769.10*	DNE
	4	3360.35*	3610.46*	DNE
	8	6305.99*	2059.84*	DNE
	16	12500.6*	DNE	DNE
GetNSafe	2	2594.19*	2943.84*	DNE
	4	8306.18*	1346.08*	DNE
	8	15938.1*	3395.78*	DNE
	16	26490.9*	5659.15*	DNE
BetterSafe	2	11400.0	2932.15	438.623
	4	31995.5	4148.00	1752.61
	8	13068.8	5067.54	2394.55
	16	24161.7	8501.92	3301.97

Tabled values are measured in ns/transition. I added DNE as the keyword for an infinite loop while any tabled value with a * resulted in a data race, more specifically a sum mismatch error.

3.1 Overall Performance Analysis

Overall performance favored the unsynchronized class because this class's implementation did not contain any checks for data race conditions, volatility, or other operations and the number of threads increased which is normal since as the number of threads and operations increase, more actions are required to ensure atomicity. Finally Synchronized and BetterSafe performed the worst in terms of efficiency(ns/transition). The Synchronized class was faster than BetterSafe in when the number of swaps and threads were relatively small. However as the number of threads grew, BetterSafe began outperforming the synchronized class by more and more. This shows the scalability of the BetterSafe implementation which wins out in the long run.

3.2 Reliability Analysis

Synchronized and BetterSafe both resulted in 100% DRF as shown by many tests across all iterations and combinations of parameters. No errors or infinite loops presented themselves. On the other hand, Unsynchronized and GetNSet were not DRF. GetNSet, due to its atomic constraints managed to perform well without error for small number of iterations and/or threads but resulted in sum mismatches for most of the data tests I tried. The behavior of Unsynchronized is due to its implementation lacking any type of synchronization. This gives the Java compiler free reign to proceed as it sees fit which optimizes the code to run the fastest but doesn't protect the data's integrity. GetNSet on the other hand, is implemented with atomic variables that keep the variables atomic but not thread-safe. Finally, for each of the classes, Unsynchronized and GetNSet, the iterations on the order of 10^6 generated infinite loops causing the terminal to hang.

4. Difficulties

The implementation of BetterState was difficult as it required reading through most of the documentation of on the java.util.concurrent package, not specifically constrained to the lock package and the VarHandle package. It was difficult to ascertain which class exactly was the best so I wrote many different implementations using each of the ones I thought would be possible. The version using ReentrantLock turned out to be the best one but this required some digging and also deep analysis on why concurrent.lock was the most relevant package.

The other difficulty had to do with overcrowding of the Linux server. At any given time, there were at least 10 other students on same server as me, which caused my runtimes to vary wildly. I decided to run BetterSafe and Unsynchronized classes consecutively so that their runtimes would be the most consistent because the original results saw extremely wild results since the times between running BetterSafe and Synchronized was long so the environmental factors changed.

5. Conclusion

Throughout this assignment, we explored multiple different techniques for both implementing synchronized multithreadings and breaking sequential consistency. Overall, my results aligned with what was expected. Unsynchronized and GetNSet performed the fastest, with Unsynchronized outperforming GetNSet as the number of threads and computations grew. Synchronized and BetterSafe were slowest with BetterSafe outscaling Synchronized in the long run because of the optimizations it does to the specific function calls that would cause DRFs. Both Synchronized and BetterSafe were DRF while the other two classes suffered from a myriad of issues including sum mismatches after computation and terminal hanging.

Overall, we can conclude that the best class for GDI is the BetterSafe class. It performed similarly to Synchronized when the number of threads was small but this is irrelevant since the goal is the multithread as much as possible. BetterSafe scaled much better performing much better than synchronized as the number of threads increased which means BetterSafe is much more suitable for the application problem of multithreading overall.