CS 131 Homework 3 Report

1. Introduction

The goal of this homework was to show off different implementations of Java's memory model and how each one performs in terms of speed and reliability. There are four different implementations that we chose to explore in this homework, including the synchronized keyword, no synchronization at all, volatile variables using AtomicIntegerArray, and using a lock. Each implementation is judged based off the time it took and whether or not it was able to finish without any race conditions. We test each implementation on a multithreaded program which maintains an array of bytes and increments one number in the array while decrementing the other. All tests were performed on SEASNet Linux Server 10 on OpenJDK version 9.

2. Implementation

These are the different classes/packages that were allowed for this homework. We ended up using ReentrantLock from java.util.concurrent.locks.

2.1 java.util.concurrent

While this package provides a lot of low-level implementations that could achieve synchronization faster than our current BetterSafe, it is difficult to use in comparison to the ReentrantLock that we decided to use. If we wanted to achieve the better performance, we would use this package.

2.2 java.util.concurrent.atomic

This package allows us to update values atomically, ensuring that a value in the array is either updated or not updated at all. This would have been faster than the ReentrantLock because of less overhead than locking and unlocking, but the problem is that one thread could check that it is lower than the max value, and then another thread takes over and pushes it to the max value. When it switches back to the original thread, it atomically increments it past the max value and would be a reliability error. Because of this, it was chosen to use the ReentrantLock.

2.4 java.util.concurrent.locks

This package provides us with many different types of locks. We used the ReentrantLock because it was simple, and only requires us to lock before a critical section and then unlock after we are finished. This guarantees reliability at least more than the synchronized keyword. While other implementations are probably faster, this package is the easiest to implement.

2.4 java.lang.invoke.VarHandle

This package provides a reference to a variable and allows this variable to be changed atomically to a new value. It would be faster than the lock-based approach we used because there is less overhead with atomic updates. But execution switches can cause this to fail and be less reliable than our ReentrantLock.

3. Results

Class	#Threads	#Swaps	
		10^4	10^5
Synchronized	1	4416.47	867.625
	2	11146.2	1536.48
	4	46284.6	3587.71
	8	135783	27342.5
	16	463436	47875.1
	32	1.97981e+06	259700
Unsynchronized	1	407.655	126.124
	2	***1730.66***	***513.721***
	4	***5338.51***	***924.050***
	8	***12162.1***	***3661.45***
	16	***23123.6***	***4540.50***
	32	***31610.8***	***13658.9***
GetNSet	1	1582.25	236.855
	2	***4087.63***	***575.631***
	4	***7383.47***	***1205.76***
	8	***25312.8***	***3633.48***
	16	***37892.4***	***6477.95***
	32	***84113.5***	***16289.5***

BetterSafe	1	3376.00	447.290
	2	11146.9	1435.15
	4	57678.0	3028.78
	8	194170	10140.5
	16	471592	42989.7
	32	1.69128e+06	172338

All values in the table are in nanoseconds and all ***bolded italicized*** values are results that had a race condition and is unreliable.

4. Analysis

4.1 Synchronized

Because Synchronized is protected through a synchronized method, it is impossible for two threads to call the same method and update the array at the same time. But, even though Synchronized is completely reliable, it is consistently the slowest in comparison to the other implementations. Therefore while this isn't the worst implementation, it isn't the best.

4.2 Unsynchronized

We see that while the Unsynchronized class is the fastest in terms of performance, it is incredibly unreliable. Because there is no attempt to solve the race conditions and two simultaneous updates to the array, it fails in terms of reliability. A command extremely likely to fail would be:

java UnsafeMemory Unsynchronized 32 10000 20 10 10 10 10 10

This command will result in a sum mismatch even though it is incredibly fast. Its unreliability means that it is not a useful implementation.

4.3 GetNSet

GetNSet is not data-race free even though it tries to achieve synchronization using the AtomicIntegerArray class. Though it makes each read and write to the array atomic, execution switches to other threads pose a problem to its reliability. If a thread reads a value right before another thread takes over and writes to the array, the value that the thread has when it regains control will be outdated and can lead to a failure. An example command that would lead to a failure is

java UnsafeMemory Unsynchronized 32 10000 20 10 10 10 10 10

However, it is at least better than BetterSafe in terms of performance, because atomic accesses take less overhead than locking. Unfortunately since reliability is more important than speed, GetNSet is not a good implementation.

4.4 BetterSafe

BetterSafe uses the ReentrantLock class to ensure that the array cannot be accessed by two different threads by locking when reading the array and unlocking once it is done writing to the array. It is able to run faster than Synchronized because it only locks the section from the method that reads and writes to the array rather than the entire swap method. The combination of it being 100% reliable and faster than Synchronized means that it is our best implementation.

5. Conclusion

With this homework, we explored four different implementations that attempted to solve our original problem with both performance and reliability in mind. In terms of performance, Unsynchronized was the fastest, followed by GetNSet, BetterSafe, and then Synchronized. This is to be expected because Unsynchronized had no overhead and no attempt to fix the data reliability problem, GetNSet has less overhead with atomic instructions, and that BetterSafe would be better than Synchronized due to a finer granularity on its lock. But our program needed 100% reliability, which was only achievable with Synchronized and BetterSafe, leaving those two as our only viable option. Since BetterSafe was faster of the two, we propose through our testing that BetterSafe be used instead of Synchronized.

6. References

1. Java Standard Edition Version 11 API Specification. Oracle and/or its affiliates. Available: https://docs.oracle.com/en/java/javase/11/docs/api/

2. Lea, Doug. Using JDK 9 Memory Order Modes. Available: http://gee.cs.oswego.edu/dl/html/j9mm.html