CS131 HW3 Report

Abstract

In this homework, we tested various ways of synchronizing (and not synchronizing) the method "swap," to gain a better picture on which method is best for short, quick operations. We tested them on different test cases, with varying numbers of threads and swaps. After the various testing, it seems that using a Reentrant Lock serves us the best for this sort of multithreading.

Description of the Java Files

jmm.jar

These include the java files: NullState.java, State.java, SwapTest.java, and SynchronizedState.java (as well as UnsafeMemory.java, but more on that later). This serves as the backbone of our testing, and gives us a platform to test the effectiveness of different synchronization methods.

UnsafeMemory.java

The .java file that contains the main method of this program. This gives us the ability to change different arguments and specify which method we would like to use. This was edited a small bit to allow the testing of the synchronization methods said below.

UnsynchronizedState.java

Allows for the "Unsynchronized" option when calling the main function within UnsafeMemory. This is similar to that of Synchronized, but removes the synchronized keyword, making it not DRF.

GetNSetState.java

Allows for the "GetNSet" option when calling the main function within UnsafeMemory. This operates using the AtomicIntegerArray. Unfortunately, due to the input array being made of bytes, this means we have to convert it to ints, and convert it back when UnsafeMemory is looking to test the array. It also does not make use of the AtomicIntegerArray in the way it should, because instead of using the atomic commands GetAndDecrement / GetAndIncrement, the spec instructed us to use Get and Set, which is not DRF.

BetterSafe.java

Allows for the "BetterSafe" option when calling the main function within UnsafeMemory. This operates using the ReentrantLock. When a thread enters the critical part of the method swap, it will lock the method from other threads, who will be forced to wait. When it leave the critical portion, it will unlock the ReentrantLock, allowing others to go through the code. Due to this, it is DRF.

BetterSafe.java Options

For this assignment, we had four possible packages to use in our implementation of BetterSafe.java.

java.util.concurrent

+ extremely versatile
+ much more broad, and allows for different kinds of locks and different kind of conditions
+ allows for concurrency that timeshares equally.
- … all we need is a simple lock, and this added complexity just makes things more difficult.
- Because all threads are running the same small code, it does not matter / happens rarely that one thread is starved out.

java.util.concurrent.atomic

+ might be one of the most efficicient, as it does not lock threads out of a given operation.
+ instead, it uses specialized hardware operations (that may be optimized better) to decrement and increment
- it only works with types that are specified by the library, i.e. bool, int, long and references
- requires an initialization of a completely new type.

java.util.concurrent.locks

+ simple
+ can be made fair
- might be somewhat inefficient compared to atomic commands, as we are still locking threads out.
- requires the initialization of the lock.

java.lang.invoke.VarHandle

+ able to set variables as volatile, which may be more efficient.
- really complicated
- requires a dynamically typed refrence

My Choice

I ended up choosing to use the ReentrantLock from java.util.concurrent.locks that works in the way I described above (in the description of Java Files). This is because it was very simple, and we really didn't need a very complicated locking procedure for this assignment, because the swap function is very simple.

Despite the simplicity, it was still quite a bit more efficient than using the synchronize keyword.

Test Results

These tests were done on lnxsrv07 on RedHat 7.

Base Case (8 threads, 1000000 swaps)

Null	2355.07
Synchronized	2469.18
Unsynchronized	∞
GetNSet	∞
BetterSafe	1550.41

(* Results are in ns/thread *)

Varying Threads (n Threads, 1000000 swaps)

	4	**12**	**16**
Null	503.64	3207.34	4067.31
Synchronized	1535.07	4238.33	5934.11
Unsynchronized	∞	∞	∞
GetNSet	∞	∞	∞
BetterSafe	782.14	2082.7	2599.96

(* Results are in ns/thread *)
(* 16 threads were the maximum possible due to the limitations of the lnxsrv -- out of memory error *)

Varying Swaps (8 Threads, n swaps)

	100	**10000**	**1000000**
Null	164345	6066.59	119.45
Synchronized	164213	9847.13	1883.70
Unsynchronized	163077*	7477.96*	∞
GetNSet	309481*	15609.6*	∞
BetterSafe	533145	15219.1	838.17

(* Results are in ns/thread *)
*sum mismatch

Comparisons

Speed

In the general case, the results speak for themselves. Due to the if-statement that returns false under the conditions that the byte to be decremented is smaller than or equal to 0 or the byte to be incremented is greater than or equal to 127, and the non-DRF nature of Unsynchronized and GetNSet, these states cause an infinite loop (more on that later). As such, under these conditions, BetterSafe is faster than Synchronized, which is faster than both Unsynchronized and GetNSet.

When we decrease the number of threads however, we see a general decrease in per-thread time(though obviously an increase in total time, which was not tracked by this simulation). Nonetheless, the relative results are the same as the general case, where BetterSafe is faster than Synchronized, which is faster than both Unsynchronized and GetNSet.

We have slightly more interesting results when we change the number of swaps required. We see that a decrease in the amount of swaps increases per-thread time (but decreases total time). However, the increase in time is not symmetric across all methods. We can see that in the 10000 and 100 case, BetterSafe takes much longer than Synchronized and Unsynchronized. However, is the 10000000 and 1000000 (default) case BetterSafe is faster than Synchronized. This can most likely be attributed to BetterSafe's (and GetNSet's) overhead time, for they both have to do things other than swap when they initialize the thread. However, this is a fixed cost (at least for BetterSafe, as GetNSet has to convert the ints to bytes when returning the array), so it would be amortized when the number of swaps increases. From this test we can determine that for a small amount of swaps, Unsynchronized > Synchronized > BetterSafe > GetNSet, while for a large amount of swaps, BetterSafe > Synchronized > GetNSet = Unsynchronized (where > is faster).

BetterSafe is more efficient at a high number of swaps as compared to Synchronized due to each thread not having to wait as long. The synchronized keyword locks the entire function, while manually using the ReentrantLock allows us to just lock the critical portion of the code (the if-statement and decrement/increment). As such, even though each thread has a lower wait time, and is able to run more quickly, we are still able to maintain 100% reliability.

Reliability

We can see some issues with the non-DRF functions. In the general case, both GetNSet and Unsynchronized options create an infinite loop. The reasons that this would occur is due to the possibility of all elements

within the array being at zero or all elements being at 127.

Additionally, we see that even when an infinite loop does not occur, it is not 100% valid. This can be seen in the 100 and 10000 swap cases. This invalidity when piled up is what causes the all-zero or all-127 elements case that creates the infinite loop.

This invalidity could be caused when Thread B interrupts Thread A while it has just gotten the number to be decremented, then runs to completion. When Thread A continues, that decremented number will be one number too high when it is finally setted. A similar thing occurs in Unsynchronized because the operation value[i]-- is actually three (not atomic) operations: getting value[i], decremented, and making value[i] = to that decremented number. Like before, if the thread is interrupted when it has gotten but before it had set, the decremented number will be one too high.

The following are commands that can be used to see these bugs occur:

- Infnite Loops

  java UnsafeMemory Unsynchronized 8 10000000 6 5 6 3 0 3

  java UnsafeMemory GetNSet 8 10000000 6 5 6 3 0 3

- Invalid Results (sum mismatch)

  java UnsafeMemory Unsynchronized 8 10000 6 5 6 3 0 3

  java UnsafeMemory GetNSet 8 10000 6 5 6 3 0 3

Conclusion

For the purposes of GDI, I believe that BetterSafe.java and its implementation of the ReentrantLock is the best choice out of the four. Despite GDI not necessarily needing 100% reliability, BetterSafe still runs the most consistently and the fastest out of the other options. There is room for improvement however, as we are not making use of ability to have some error.