CS131 Homework 3 Report

Devyan Biswas

1. Overview

Java's Memory model allows for the use of synchronization techniques by which multithreaded programming applications can be optimized and, as a result, protect agains data race conditions (DRC's). In this lab we explore 4 different methods to prevent DRC's: the in-built synchronization keyword, no synchronization, an atomic array with get and set methods provided by the class in java.util.concurrent.atomic, and lastly the Reentrant lock class from the java.util.concurrent.locks.

2. Synchronization Methods for BetterSafe

There were 4 main libraries with which we could have worked for BetterSafe. They are java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks, and java.lang.invoke.VarHandle. For my implementation, I chose to go with locks, specifically the ReentrantLock within util.concurrent.locks. The pros and cons for each are as follows.

2.1 java.util.concurrent

When looking through this package, I noticed that besides some low level data structures like Queues and semaphores, there weren't many other higher-level ways to synchronize. While this would allow for greater customization, it increases complexity, and I as usual did this lab 2 days before due. Though I chose ReentrantLock for its ease of implementation and relatively good speed, using this method would have allowed for a more refined and specified way of optimizing multithreading, ideally through use of a semaphore implemented with a queue. This would make overall operations a lot faster while still maintaining the race-free condition.

2.2 java.util.concurrent.atomic

This is the package in which the AtomicIntegerArray class used for GetNSet can be found. The package has classes that update their values atomically, which prevents the possibility for race conditions in terms of updating values. However, this doesn't prevent mis-ordering of operations between threads (for example if after a check there is a thread switch before an increment/decrement, which then increments/decrements then returns to the original thread, this may cause the value to go over or under the maximum or minimum value). Though this method would be much faster, the risk of data race conditions makes this way non-optimal (although, there may be a middle ground implementing this with some form of locks that minimizes the flaws of both methods).

2.3 java.util.concurrent.locks

This is the method chose, and as such I will go into a bit of the implementation. The overall methodology of a lock is simple: define a section/block of code called the critical section. Simply put, once a lock is claimed by some thread, no other thread can operate/execute the block of code until the thread currently running the code releases the lock. In terms of reliability, this method works well and does not allow for race conditions. However, because the lock forces mutual exclusion, no other threads can operate on the value while the current thread is. For this particular case, the array cannot be updated in parallel. This acts as a bottleneck for performance.

2.3.1 Implementation

The main locked part of the code is in the swap section of BetterSafe. It is as follows:

```
{
    lock.lock();
    if(value[i] <= 0 || value[j] >= max){
      lock.unlock();
      return false;
    }
    value[i]--;
    value[j]++;
    lock.unlock();
    return true;
}
```

What this basically says that, once a thread enters the function and acquires the lock, the only ways it will unlock is either if the value it is looking at is invalid for increment or decrement, or if it finishes the increment/decrement.

2.4 java.lang.invoke.VarHandle

This class essentially performs similarly to the atomic package. The VarHandle class allows you to define a reference to a variable and then provides methods to atomically interact with the variable. The same problem of race conditions arises with this class as it does with the AtomicIntegerArray class, making this method unviable.

3. Testing and Results

For clarity, I will only include the tests from Java version 9, as I tested extensively with version 11 and 9 and found little to no difference in behavior. The following is the table representing the different ns/transition for different number of threads with a constant value of transitions/ops [detailed testing script were used, but this is the essence of the data].

#Thds. [Itr = 100,000]	1	2	8	16
Better	236.320 ns/ transition	1914.17 ns/ transition	3165.46 ns/ transition	6648.84 ns/ transition
Sync.	190.081 ns/ transition	1059.70 ns/ transition	3088.82 ns/ transition	5811.22 ns/ transition
Unsync.	176.349 ns/ transition	766.891 ns/ transition [mismatch sums]	2031.50 ns/ transition [mismatch sums]	[Timeout]
GetNSet	240.722 ns/ transition	[Timeout]	[Timeout]	[Timeout]
Null	178.423 ns/ transition	625.350 ns/ transition	1317.15 ns/ transition	2292.78 ns/ transition

*** There is one more test with 32 threads in which it is clearly shown that the BetterSafe method works over Synchronized for larger # of threads (not included in above table due to formatting problems):**

$ java UnsafeMemory BetterSafe 32 100000 4 2 2

Threads average 13553.8 ns/transition

$ java UnsafeMemory Synchronized 32 100000 4 2 2

Threads average 15261.9 ns/transition

Additionally, here is some data for an increasing number of transitions/ops for a constant number of threads.

# Trans. [Thds = 8]	10,000	100,000	1,000,000
Better	11184.8 ns/ transition	2932.29 ns/ transition	1045.16 ns/ transition
Sync.	7571.45 ns/ transition	3255.70 ns/ transition	2377.02 ns/ transition
Unsync.	6779.76 ns/ transition [mismatch sum]	1820.14 ns/ transition [mismatch sum]	3026.07 ns/ transition [mismatch sum]
GetNSet	20159.5 ns/ transition [mismatch sums]	[Timeout]	[Timeout]
Null	7039.08 ns/ transition	1471.96 ns/ transition	2166.14 ns/ transition

3.1 Analysis

According to the above, the clearest winner in terms of speed would appear to be Unsynchronized. However, since there is a high amount of mismatch when the number of threads increase beyond a single thread, it doesn't outperform the others overall. The GetNSet method doesn't perform well in any regard, as it takes far too long when scaling up the number of threads and/or transitions from 1 thread or <10,000 transitions. Between Synchronized and BetterSafe, there are a few points in which Synchronized comes out on top: mainly with fewer transitions and threads, it performs quicker. However, on average, when threads or transitions are increasing, BetterSafe scales more reliably and eventually outperforms Synchronized.

3.2 Reliability

As expected, Unsynchronized and GetNSet were not completely data race free. There were many instances of mismatching sums and timeouts (which represent calculations being incorrect due to the fact that the SwapTest will run infinitely when values in the array are <=0 or >= maxVal, which also implies mismatched sums). However, Synchronized and BetterSafe were both very reliable, having a 100% correctness rate (based on my testing). This lines up with the previous descriptions of the various classes and packages. GetNSet utilized atomic loads and stores but did not prevent unordered accesses, and BetterSet took a hit in performance early on but proved itself in terms of scalability over the Synchronized block, as a lock will be much more predictable.

4. Problems

This lab wasn't technically too difficult. The main issue was in going through documentation to figure out which package to utilize when building BetterSafe. Once I chose the ReentrantLocks class, the documentation was very straightforward and useful when figuring how to use the lock and where to put it based on the nature of the swap function. The main trouble I had was figuring out testing methodologies. Thankfully Java11 and Java9 performed similarly to each other, facing space in the report. But I had to build 2 different testing scripts: one to be run through the Makefile and another to be run manually due to it testing GetNSet and Unsynchronized, both of which have a tendency to hang and infinitely run. In retrospect I definitely should have had a higher granularity of testing when it comes to comparing the different classes on different numbers of threads and transitions, but through the above data the important trends are still interpretable.

5. GDI and Conclusion

GDI's product(s) rely on efficiency while being alright with sacrificing some reliability. While GetNSet offers this, there is a massive tradeoff with the frequency of unpredictable race conditions, which throw the program into an infinite loop. Unsynchronized is a relatively viable option, able to scale with threads with minimal difference in computed sum (<=2 in my testing), but with larger transitions it quickly becomes evident that this method times out far too often. This is improbable for a company that requires a massive amount of transitions/operations on a particular data set. Between Synchronized and BetterSafe, BetterSafe is superior simply given its scalability. Without that, Synchronized would have been better. But because BetterSafe so reliably scales and operates quickly with no race conditions, it is the method GDI should implement.

6. Sources

-https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/package-summary.html

-https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

-https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/package-summary.html

-https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/invoke/VarHandle.html

-https://www.cs.umd.edu/~pugh/java/memoryModel/