Homework 3: Java Shared Memory Performance Races

Shaan Mathur, *UCLA Computer Science Undergraduate*

Abstract

We examine possible implementations for synchronized access to an array of range-bounded counters using Java SE 9. The primary purpose of this is to see if we can avoid using the `synchronized` keyword so that the JVM does not have to handle synchronization. The route this paper decides to take is to use the *adder* abstraction that is provided in the `atomic` concurrency package. The abstraction allows us to have higher throughput and consequently reduce overall latency.

1. Platform

Java 9.0.1

Java(™) SE Runtime Environment (build 9.0.1+11)

CPU: 4-Core Intel(R) Xeon(R) CPU E5-2640 v2; 2.00 GHz, 64 GB RAM.

2. Choosing Between the Various Packages

2.1 `java.util.concurrent`

`java.util.concurrent` has various subpackages that provide various concurrency alternatives. Each of these packages are specific in application and are useful in different scenarios. We go into depth into each of these, but our main comment here is that this package has many subpackages that provide deeper abstractions that go beyond standard synchronization idioms (e.g. mutex, condition variables, semaphores). For instance CyclicBarriers are one possible abstraction, allowing us to wait on several threads before letting them all through (an extension of our common notion of semaphores). Though this can be useful when we are trying to effectively "join" several threads, we have no reason to use it here (we want threads to never stop ideally). The same reasoning is used to disregard Phasers and CountdownLatches.

2.2 `java.util.concurrent.locks`

The subpackage `locks` has several classes that implement a Lock interface, essentially allowing basic mutex operations such as locking, unlocking, and trying locks. This is a fundamental synchronization abstraction, and so using this makes things very **readable** and **clear**, since this is a common abstraction that many programmers are familiar with. However, we must also recognize that implementing a coarse-grained locking mechanism is essentially no different than just using Java's object monitors via the `synchronized` keyword.

However, one may imagine a more finely-grained implementation where every entry in the array has a corresponding lock that has to be acquired in order to access it. To prevent deadlock, a total ordering of lock acquisition of "lower index first" could be used, and we are off to the races. Unfortunately, this only pans out if the number of elements in our array are large. Given the fact that this application requires the user to specify each and every element in the array, it is clear that this will not be practical.

One interesting idea also is the notion of a ReadWrite lock, which essentially allows concurrent readers but only one writer at a time. However, this is only advantageous if we spend most time reading values versus writing them. In reality, this is not the case, so this would not be appropriate for this application.

2.3 `java.lang.invoke.VarHandle`

This is also a very useful concurrency package, its particular strength being the various protocols that can be used in atomic operations such as compare and swap/exchange. One has the option to have lazy setting of variables, strict volatile reads and writes, or somewhere in between within that spectrum of possibilities.

However, the main issue with an implementation using this package is that our use case requires the reading of two different values, not just one. If we wished to test to see if one of the values has a certain value prior to setting its value, then compare and exchange would be a perfect idiom to apply. However, we have two reads and two writes that we wish to make atomic.

Another issue is that this package does not really provide a straightforward and generic conditional swaps (e.g. if x > y, then we swap x and z atomically). This application requires that our values be locked within a range, so this cannot help us.

2.3 `java.util.concurrent.atomic`

At first it seems like the atomic package is not going to do us much good for the aforementioned reasons. We need to perform operations on two different bytes of memory that involve reads and conditional writes. Although we could force a read from memory and force an atomic write, mayhem can still happen in between these reads and writes, causing race conditions.

However, it turns out this package has an abstraction that is very useful for us. The package goes beyond the atomic variable abstraction and gives us the notion of an *adder*. The adder provides an illusion of atomicity and so is thread safe. The real power in the adder is that if two threads try adding to the same adder, the internal implementation will create two separate variables for each thread and add to those thread copies. When we ask for the sum, the implementation will then take the net effect of each thread's copy and then produce the correct result.

A huge advantage of the *adder* is that no thread has to wait, so we achieve much higher throughput and consequently lower latencies per thread. We thus approach this problem by using this abstraction to optimize throughput.

2. Implementation with LongAdder

The exact adder class we use is the LongAdder, which allows us to use the adder abstraction for integral types (particularly long in this case). So instead of having a `byte` array, we will have a `LongAdder` array. When trying to access a value, we invoke the sum function to resolve all the thread's counters for the specific adders of interest. We then, if appropriate, invoke either increment or decrement methods on those adders and return. Another huge plus with this implementation, besides the increased throughput, we also win fine grained concurrency for free, since we no longer have to lock the entire array (as we would with the `synchronized` keyword) just to access two elements.

This implementation is fairly easy and requires only a couple extra lines of code and some invocations. The use of this package is very lightweight, comparable to the ease of using `synchronized`.

2. Statistics

Threads	Synchronized (ns)	BetterSafe (ns)	Null (ns)
1	25.98142	23.69414	5.366332
2	145.7093	85.78093	6.823662
4	324.4663	139.1063	12.85706
8	335.1692	279.4729	29.1384
16	691.1026	568.4936	68.97352
32	1348.869	1121.235	158.9184

Figure 1: ***UnsafeMemory Test Runs****. Our implementations were tested against the other possible synchronization techniques. BetterSafe performs better than synchronized, but not near the lower bound. Other classes could not be tested because they repeatedly failed and provided no meaningful data.*

To measure the effectiveness of our BetterSafe implementation, a shell script was created that ran the following command:

```
java  UnsafeMemory  <module>  <threads>
100000000 127 50 20 58 25 75
```

This command was run with each module and varying number of threads. See Figure 1 for the statistics, and Figure 2 for a graphical depiction.

We can see from the data that BetterSafe does indeed do better than just using the `synchronized` keyword. Clearly, we are getting increased throughput and, as consequence, a reduced overall latency. This becomes more evident as we have more threads contending.

Future tests may include varying array size so we can see how finer grained approaches may improve overall performance.

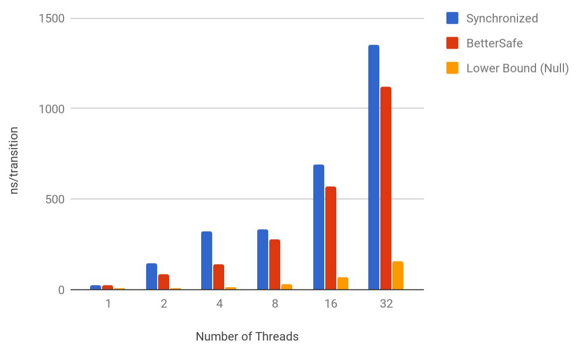

Figure 2: ***UnsafeMemory Performane Graph.*** *Our implementations are much higher than the theoretical lower bound, though this may also be an unattainable since it assumes no contention. BetterSafe increasingly performs better as the number of contending threads increases.*

3. Reliability and DRF

The statistics only show data for two of our classes: Synchronized and BetterSafe. The other classes were unreliable and we could not collect any data for them. Obviously, there are clear data races in the other implementations, so they are not data race free (DRF). The synchronized implementation is DRF since it is a coarse-grained synchronization approach that enforces mutual exclusion anytime the byte array is tampered with. Now this only leaves BetterSafe.

BetterSafe was seemingly able to be free of any data races since there are no errors after the fact once our test suite completes. Of course, this is no guarantee. It could always be that we had data races but they canceled out leading to false positives. However this test suite was run repeatedly with a required 10,000,000 swaps, without issues with a high number of threads.

One huge guarantee we get from our adder is that even if multiple threads try to increment or decrement it, all of them will be served without blocking (high throughput in consequence). This is done by creating a separate adder for each thread and then merging the results once a request to find the current count is made. Therefore concurrent increments/decrements are data race free.

One thought is that there may be a data race if two threads ask for the count, find out it is just by the barrier but allowed to mutate, and then both perform an operation that pushes it over the edge. There is no guard against this case, but in practice it does not occur. Trying to create a mutex to guard against that one rare case destroys our performance in the long run since we fall right back into doing something similar to `synchronized`. Thus we do not use such a mutex, and have had no issues within our test suite.