CS 131 Homework 3 Report

Tian Ye

Abstract

The objective of Homework 3: Java Shared Memory Performance Races is to quantifiably test and record differences in speed and accuracy of synchronized Java classes in comparison to those that are unsynchronized. We observed overall that synchronized Java classes tend to have higher accuracy due to elimination of data races; however, that same feature caused the synchronized classes overall to run slower than those that are unsynchronized.

1 Testing

To test the various implementations of the classes, I ran 10,000,000 swaps on an array of 600 bytes using 8, 16, and 32 seperate threads in the range of 0 to 127 (maxval). The results are listed in the table below:

Model	Time Taken for Transitions			DRF
	8 Threads	16 Threads	32 Threads	
Null	148.147 ns	249.381 ns	785.381 ns	Yes
Synchronized	2220.45 ns	4141.20 ns	6886.52 ns	Yes
Unsynchronized	354.793 ns	722.730 ns	1413.33 ns	No
GetNSet	1059.19 ns	2428.79 ns	5333.19 ns	No
BetterSafe	393.397 ns	841.468 ns	1733.38 ns	Yes

2 Comparisons

Viewing the test data, we can quickly see that unsurprisingly, Synchronized performs the worst. This is not surprising as Synchronized uses the synchronized keyword which prevents preemption by other threads. This in turn causes the Synchronized to be data race free (DRF) and always reliable.

In contrast, since Unsynchronized has no synchronization overhead, it runs faster than Synchronized; however, due to the lack of overhead it does not have blocks against competing threads and permits whichever thread that comes first to perform the swap, resulting in race conditions and causing it to return an incorrect sum.

Null runs the fastest out of the four as it always returns True whenever the swap function is called - it doesn't actually perform any tasks. It has no accesses to memory and therefore no protection needed on memory.

Similarly to Unsynchronized, GetNSet does not use the synchronized keyword. As a result, there is again no synchronization overhead. However, it still performs worse than Unsynchronized as there is some synchronization overhead occuring when it is updating the AtomicIntegerArray data structure. Instead of blocking the entire swap method, GetNSet only makes access to individual array elements atomic. Therefore it performs worse than Unsynchronized as it has stronger memory protection, but is still nonetheless not DRF as it trades reliability by not synchronizing the entirety of the critical section for some speed performance instead.

Viewing BetterSafe, we see that it performs nearly as well as Unsynchronized and is still relatively close in speed to Null while being DRF. BetterSafe achieves 100% reliability by only synchronizing the critical section using finer grained locks as opposed to locking the entire function. While BetterSafe does have overall a relatively strong memory order model, it manages to outperform Synchronized by a significant margin, especially at higher thread counts, by using a finer grained lock rather than simply synchronizing an entire function.

Rather than synchronizing the entire swap function, BetterSafe synchronizes only the critical section within the swap function, specifically the portion that is performing reads and writes from shared memory space, which is protected by a reentrant lock. This provides 100% reliability as this is the only section of the code that is susceptible to data races, as without the lock, a write can happen before, during, or after a read, depending on the thread scheduling. Locking only this small section provides a lightweight solution to this issue as opposed to synchronizing the entire swap function.

Hence, we can say that BetterSafe is the best class for GDI for the reasons mentioned above.

3 Package Analysis

`java.util.concurrent`

While upon first glance it would seem that the concurrent package would be well suited to an implementation of BetterSafe, upon closer analysis we see that the class definitions within this package are catered towards low-level synchronization programming paradigms and data structures, such as Thread Factories and Deques.

Such a package enables a programmer to have much control over ordering of threads and gives the ability to enforce a strong memory order by ranking threads and sorting them onto a Thread Deque such that the highest order thread runs first.

However, this adds a lot of complexity and as a result it will be extremely difficult to coordinate many threads in an object method that the threads will be running in. Consequently, this package is not suited to being used in an implementation of BetterSafe. However, it can be used with effectiveness in a class that is spawning the threads, such as an extended implementation UnsafeMemory.

`java.util.concurrent.atomic`

The concurrent.atomic package provides classes that do not use locks to provide atomicity of executing critical section or updating data. This provides the benefit of not needing to lock an object ourselves before accessing it - the class automatically handles it for us. Furthermore, if multiple data structures within a single object are being modified, each individual structure will lock itself to allow accesses without affecting the other data structures within the same object. Therefore it is a more fine-grained mutual exclusion mechanism.

However, since we are performing both reads and writes, rather than just one or the other, this package is not enough. The atomic package provides data structures that can perform atomic reads and writes and protect them individually; however, an interrupt in the middle of the semi-protected critical section after the read but before the write will affect the validity of the result - therefore the package is not viable. The data structures included in this particular package are just not enough.

`java.util.concurrent.locks`

This package provides a slew of lock classes that allow programmers to lock a critical section with customizability. Locks do have the drawback of locking an entire object. This is unfortunate in the case such as when another thread is prevented from modifying a different shared memory space that is not the one currently being modified by the lockholding thread. However, this method is simple to use and fits our implemenation goals of BetterSafe.

By locking only the critical section, we reduce the overhead that was associated with the synchronization keyword. We use the reentrant lock as it is the main implementation of mututal exclusion. We are therefore able to achieve complete reliability by locking only the critical section, while simultaneously achieving better performance than Synchronized by decreasing the synchronized area of code.

`java.lang.invoke.VarHandle`

The VarHandle class acts similarly to the AtomicIntegerArray class with the exception that is is generalized to multiple types besides only integers. While this does permit synchronization of an array of objects other than just integers, it is not useful in the case of the implementation of BetterSafe, as we are handling only integers and it still comes with the problems associated with AtomicIntegerArray that were discussed earlier.

4 Challenges

The primary challenge that I came across was populating large arrays for my tests - it was not feasible to manually populate more than 20 or so random elements. As a result, I used a random number generator to populate a large array of a set size (in my case 600) to complete my tests. The randomization of inputs creates a more random distribution of timing results, which makes the data more inclusive of the testing sample space.

5 DRF Class Analysis

The reliability test that I used was to have UnsafeMemory output a sum mismatch in the event that the sums do not match, and if so, it will display the mismatch onto the output of the terminal.

Null is DRF as it does not modify any shared memory. Hence, a mismatch in sum will never occur.

Syncrhonized is DRF as when it modifies shared memory, it locks the entire object and only unlocks after the function returns the synchronized keyword. Therefore it too will never have a mismatch.

Unsynchronized is not DRF as it modifies shared memory without any memory protection. Hence a call with a large number of swaps and threads, such as

```
java UnsafeMemory Unsynchronized 32 10000000
```

will almost certainly result in a sum mismatch.

GetNSet is also not DRF as it only protects a portion of the critical section, rather than the entirety. The reads and writes to shared memory are independently protected accesses; therefore, when a thread is interrupted between a read and a write, there will be a data race. Therefore, similarly to above, a case with a large number of swaps and threads greatly increases the chance of an interrupt occuring during the semi-protected critical section. The following command will likely cause a sum mismatch on the SEASnet server:

```
java UnsafeMemory GetNSet 32 10000000
```

BetterSafe is DRF as the entire critical section in each thread is locked. Thus, interruptions such as thread preemptions will not affect the validity of the execution of the critical section. Hence, this guarantees the reentrancy of the execution of each thread's critical section, and therefore guarantees 100% reliability in regards to data races for each thread reading or updating.