CS 131 Homework 3 Report

Abstract

This lab tested different methods to implement concurrent programing using Java. The program swaps elements in a byte array, but maintains the sum, so that it is the same each time a swap occurs.

1. Comparing Packages and Classes

I compare the four different packages and classes, briefly describing what each can do. I also explain why I chose a certain package or class over the other to implement BetterSafe.

1.1 java.util.concurrent:

The java.util.concurrent package is used to create concurrent applications. It has various classes that can be used in concurrent programming. This package has a class called Semaphore that can be used to block thread level access; whenever a thread tries to enter, it is checked for an available permit by the semaphore. The package also has a class called Locks that is used to block other threads from accessing certain parts of a code that a current thread in the lock is not currently executing.

1.2. java.util.concurrent.atomic :

The java.util.concurrent.atomic package supports lock-free thread-safe concurrent programming on single variables. It includes classes and methods that watch for access control, incorporating VarHandle operations. This method was used in GetNSet, therefore I did not use it again for BetterSafe.

1.3 java.invoke.VarHandles:

The java.invoke.VarHandle package includes VarHandle, a typed reference to a variable that can be used in concurrent programming using methods that gives or denies access to variable under certain access modes.

1.4 java.util.concurrent.locks :

The java.util.concurrent.locks package includes Lock, which is an interface and the main implementation is ReentrantLock, which has the same behavior and semantics as a monitor using the Synchronized method. It provides methods that work to acquire locks for the threads. The lock is owned by the thread that was the most recent locked, but has not unlocked yet. Once a thread has obtained a lock that means it was successful and does not have to wait. When a thread has a lock it will follow with the code in try{…}. At the end of the try{..} is a finally{..} that unlocks the lock.

I chose to use the ReentrantLock class from the java.util.concurrent package for the BetterSafe implementation.

My BetterSafe implementation is faster than Synchronized, because in Synchronized when a thread is executed and in a synchronized block and another thread tries to enter the block, it cannot and has to wait until the current thread has finished and exits the block. Once that thread exits the block, any running thread can acquire the lock, which can lead to unfairness in the amount of locks a thread has; it has no waiting system that ReentrantLock has. Also, because of this, synchronized blocks have to be available in the same method to be used by a thread. With the ReentrantLock, you can choose where or when to have locks and to unlock those locks, rather than having the entire function be "locked" when using Synchronized.

1.5 Summary of Packages and Classes

Overall, all of the packages have similar concepts and methods to implement a synchronize-like class. The java.util.concurrent.atomic and java.util.concurrent.locks both fall under the main package, java.util.concurrent, so they are all similar. Also, java.invoke.VarHandles has the same idea as locks, but uses read and write access modes to do so.
In the end, I chose ReentrantLock over the other methods, because it has more flexibility in the sense that in ReentrantLock, threads do not need to be blocked for an indefinite period of time like it does when using Synchronized.

2. Comparing Different Classes/Methods

This section includes data from testing Synchronized, Unsynchronized, GetNSet, and BetterSafe with different number of threads and transitions. The data is then analyzed to see whether or not each method is Data Race Free and why or why not they may be or not be Data Race Free.

2.1 Data

The data below is tested on the SEASnet between servers 6, 7, and 9 using at most 20 threads due to memory allocation error if too many threads are created.

Number of Threads and ns/transition using 1 ,000,000 swaps:

java UnsafeMemory method_here #threads 1000000000 127 0 127 90 2 127

	4	8	16	20
Synchronized	1067.56	2011.59	3392.35	4826.02
Unsynchronized	--	--	--	--
GetNSet	--	--	--	--
BetterSafe	494.931	964.850	1645.80	2159.48

In this test, Unsynchronized and GetNSet are proven to not be data race free. The maxval is set to 127 with the first five elements of the array containing values that would go out of bound if incremented or decremented by one. When testing these two methods with this setting, they both hang and never end. Synchronized and BetterSafe are data race free, therefore when given this setting, they both run fine and output the threads average in ns/transition each time, with no mismatch sums.

Number of Threads and ns/transition using 100,000 swaps:

java UnsafeMemory method_here #threads 100000 127 5 6 3 0 3

	4	8	16	20
Synchronized	2849.69	4129.87	8037.45	9357.78
Unsynchronized	1239.25 mismatch	1625.60 mismatch	4154.62 mismatch	9464.69 mismatch
GetNSet	1862.66 mismatch	3612.28 mismatch	9077.08 mismatch	17694.1 mismatch
BetterSafe	1857.18	3291.49	6644.11	8828.40

In this test, I changed the number of transitions to accommodate for Unsynchonize and GetNSet, so that I can test them without having the hanging problem. This data shows that Unsynchronized and GetNSet are not data race free, because each run they result in mismatch sums.

2.2 Comparison

Looking at the data from the previous two charts, Unsynchronized and GetNSet are proven to be unreliable, because they can go into infinite loops and result in various sum mismatches.

Comparing Synchronized and BetterSafe, BetterSafe has a better performance than Synchronized. This is because BetterSafe uses locks, or specifically the ReentrantLock from the java.util.concurrent.locks package. Synchronized and BetterSafe both have the same general method of using “locks”, however Syncrhonized sues the synchronize method that locks the object, which is costlier than using ReentrantLock; ReentrantLock has more flexibility and therefore, better performance.

2.3 Reliability

The following chart tests the reliability between Synchronized and BetterSafe using 20 threads, 10,000,000 swaps, maxval of 127, and elements that will go out of bound if incremented or decremented by one, over the span of 30 runs. Reported are the number of times each method went into an infinite loop or had sum mismatch, if any.

java UnsafeMemory Synchronization 20 10000000 127 127 0 127 122 1

	Infinite Loop	Mismatch
BetterSafe	0	0

The result was that BetterSafe averaged around 1400-1600 ns/transition for every run and zero infinite loops and zero sum mismatch occruances.

Because Synchronized was given to us, we know that it is data race free, so we are testing whether or not our implementation of BetterSafe is also data race free, while having a better performance than Synchronized. After testing BetterSafe many times, the results have shown that it does not go into any infinite loops or have any sum mismatches, therefore my implementation of BetterSafe is 100% just as reliable as Synchronized.

3. CPU and Memory Information

Here I gathered reports and statics on the CPU and memory information of Java –version using SEASNet.

3.1 CPU

Number of Cores: 8
Number of Processors: 32
Vendor_id: GenuineIntel

Model: 62

Model Name: Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz

3.2 Memory

MemTotal: 65757652 kB

MemFree: 55281368 kB

MemAvailable: 64473524 kB

Buffers: 249116 kB

Cached: 8943820 kB

SwapTotal: 20479996 kB

SwapFree: 20365952 kB

Conclusion

My BetterSafe implementation using ReentrantLock is faster than Synchronized and still 100% reliable. As seen in section 2, the data shows that BetterSafe outperforms Synchronized in using 4 8, 16, and 20 threads with 1,000,000, 100,000 and 10,000,000 swaps or transitions all while maintaining reliability, because it does not run on forever and have any occurrences of sum mismatches, under the amount of times I have tested it.

Using ReentrantLock helps maintain ordering of locked regions, whether a thread can have a lock or when one is unlocked so another can access a lock.