CS 131 Homework 3 Report

Summary
The purpose of this assignment was to show the effects of concurrency when implemented in several ways. Several different versions were tried out to observe how synchronized and unsynchronized threads, as well as a few intermediate levels, behaved. The results were then recorded through testing on seasnet server 7.

Experiment
I conducted the tests on seasnet server with the following specifications:

Java version: 1.8.0_112
CPU: Intel Xeon ES-2640 v2
Processors: 32, 8 cores each
Cache: 20Mb
RAM: 64Gb

I ran 1,000,000 swaps on each implementation, running 8, 16, and 32 threads. The test data was an array of 100 byte values ranging from 0 to 127. Here is an overview of what each class is:

Null: Swapping does nothing, and is used to measure the total overhead of running the program.
Synchronized: Uses synchronized java keyword so threads run in turn, not parallel.
Unsynchronized: Threads all run concurrently, without safeguards.
GetNSet: Unsynchronized, but has added safety that retrieval and modification are an atomic unit.
BetterSafe: Uses a reentrant lock so that the shared resource can only be modified by one thread at a time.
Better Sorry: Still uses a byte array, but temporarily converts to atomic integer for swapping.

One thing to note here is the concept of Data Race Free, or DRF. Some may seem faster than others, but this is at the cost of it not always giving the correct results. This is a result of multiple threads modifying the same value, which will deliver inconsistent data every data. Next to each column, I note whether or not it is DRF.

State	DRF	Transition time per thread count (ns)		
		8	16	32
Null	Y	1884.94	4781.57	7945.35
Synchronized	Y	2477.51	5024.01	11367.7
Unsynchronized	N	1496.80	4723.83	6640.89
GetNSet	N	1718.78	3155.60	7056.13
BetterSafe	Y	1317.34	2726.84	4784.66
BetterSorry	N	1678.57	3561.45	7381.75

Figure 1: Chart comparing the different transition times at different thread counts.

Observations & Analysis
At a glance, we can see that the average transition times for the Synchronized state takes the longest. This is to be expected, since there is no concurrency involved. The Unsynchronized state is much faster because there is no blocking whatsoever and every thread can run through to completion without waiting.
Next are the intermediary states. GetNSet is a little slower than Unsynchronized for 8 and 32 threads, much still faster than Synchronized. This is because GetNSet uses an AtomicIntegerArray to conduct volatile operations. It will be run in parallel, but the array may have to wait to maintain atomicity. However, it is not DRF because even though individual instructions are atomic, multiple instructions must be run. Any combination of interlacing two threads through these instructions would create a data race condition.

BetterSafe is much better than Synchronized, almost twice as fast, while still retaining 100% reliability. It achieves this by placing a lock only in the portion of the code that modifies the shared resource. This way, the rest of the code can still be run without being blocked unnecessarily.

BetterSorry takes a similar approach to GetNSet, using AtomicInteger. However, the resource itself is stored as a normal byte array, and only during swapping does it use AtomicInteger. Because there is no locking, this should run faster than BetterSafe. On the chart above, this is not the case however. I think it is the way the hardware is configured on the seasnet server that optimizes for parallel computing. On my own computer, it is clearly shown that BetterSorry is faster than BetterSafe by a couple hundred nanoseconds. Although it is not DRF for the same reason as GetNSet, it is more reliable than Unsynchronized because of the atomic read and writes.

Importance of DRF

Data Race Free is an important concept because only DRF states can provide the level of reliability that can guarantee the exact same result every time. Synchronized state does this in the most brute force way possible by eliminating concurrency and synchronizing the threads, which is why it is very slow. BetterSafe does the job in a much more efficient way, by only making threads wait for the parts that actually need to be done in turn.

Unsynchronized, GetNSet, and BetterSorry are not DRF, as explained above. Unlike DRF, there are different levels reliability. Although non-DRF states run much faster with multiple threads, they can introduce extremely difficult to detect bugs, so one must be very cautious when going this route. For example, Unsynchronized reported a mismatch every single time, suggesting an extremely low rate of reliability. GetNSet and BetterSorry, while a bit slower, very rarely reported a mismatch, suggesting a high rate of reliability, while still not being completely DRF.

Conclusion

Through the testing of various models of concurrency, we can observe the tradeoffs between speed and reliability. For the most part, reliability is the most important, but speed is always a plus. For 100% reliability, definitely choose BetterSafe. If some errors are allowed, depending on what machine you are running, most likely you should choose BetterSorry. Both in theory and on my personal machine it is able to run faster with a fairly low and acceptable rate of error.