CS 131 Homework 3 Report

Testing Environment Information

Java Version

openjdk version "1.8.0_111"
OpenJDK Runtime Environment (build 1.8.0_111-b15)
OpenJDK 64-Bit Server VM (build 25.111-b15, mixed mode)

CPU & Memory

SEASNet lnxsrv09: 8-core Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz with 64GB RAM

Testing Methodology

To facilitate quick testing, I automated the interface with the `UnsafeMemory` test runner using a Bash script, `test.sh`, which is in the top-level directory of my submission. For each class that doesn't have the potential to deadlock, the script runs 100 simulations for 1, 2, 4, 8, 16, and 32 threads and records the average ops/second to a file from which I calculated median and mean running times across the test instances.

Class Implementations

Synchronized

Synchronized is DRF because the entire method is "synchronized," which means only one thread can execute it at a time. This prevents any potential conflicts but also means that the swap doesn't really take advantage of multiple threads. As a result, it runs the slowest of the five implementations.

Unsynchronized

The code of this implementation is identical to Synchronized except the swap method is not synchronized. Unsynchronized is not DRF because, for example, multiple threads can read the same value at the same array index, then write back a value incremented by one. Because multiple operations took place, the value after all threads finished executing should have instead been incremented by the number of threads that ran the code.

GetNSet

GetNSet is not DRF because while it uses `AtomicIntegerArray`, we are instructed in the spec to "implement it with [`AtomicIntegerArray`'s] `get` and `set` methods" rather than with its atomic operation methods `getAndIncrement` and `getAndDecrement`. By using `get` and `set` individually, the scheduler still has the opportunity to interleave instructions between the read and the write, introducing a race condition.

BetterSafe

I implemented BetterSafe using `ReentrantLock` from `java.util.concurrent.locks`. The implementation is DRF because it locks the thread for the entire procedure's reads and writes. Even though same block of code is under lock as in Synchronized, it performs better than Synchronized because according to IBM, `ReentrantLock` "offers far better performance [than `synchronized`] under heavy contention. In other words, when many threads are attempting to access a shared resource, the JVM will spend less time scheduling threads and more time executing them."[1]

BetterSorry

I implemented BetterSorry using Synchronized blocks after discovering that the primary place data races occur in practice is in the incrementing and decrementing of the individual elements of the values array. My implementation is more reliable than Unsynchronized because it ensures that these increment and decrement operations are not executed by multiple threads at once, and it is faster than BetterSafe because it does not block multiple threads from running the conditional statement simultaneously.

Because there is no locking around the `if` statement, the implementation is not DRF. It is most likely to fail when the number of threads is high and the `maxval` is low, because this configuration introduces the most opportunities in which the early return in the conditional should be run (i.e. incrementing or decrementing would cause the integer to be out of range) which introduces more opportunities for failure. E.g., `java UnsafeMemory BetterSorry 32 100000 2 1 1 1 1 1`.

Measurements

All measurements in ns, rounded to the nearest ns. Averages are across running 100 simulations each using 1, 2, 4, 8, 16, and 32 threads. All results are from running on the lnxsrv09 SEASNet server. Unsynchronized and GetNSet are not included because they are not DRF and can deadlock.

	Null	**Sync**	**BetterSorry**	**BetterSafe**
Mean	2979	7042	4880	5132

Recommendations

As we are told in the spec that GDI is willing to sacrifice some accuracy to obtain results quicker, BetterSorry seems like a good tradeoff between accuracy and speed because it performs about 25% faster than Synchronized while still making few errors.

[1] Goetz, Brian. "Java Theory and Practice: More Flexible, Scalable Locking in JDK 5.0." IBM DeveloperWorks. IBM, 26 Oct. 2004. Web. 21 Dec. 2016.