Homework #3 Report
Patrick Killian Jackson
2/9/16

Testing Environment:
For this assignment, I remotely SSH'ed into the SEASnet GNU/Linux server via terminal on OSX. All programs were compiled using Java standard edition 8, specifically version 1.8.0_51.

Purpose:
The purpose of this homework is to explore Java synchronization by comparing various implementations of synchronization that differ in speed and reliability. To do so, I modeled an array of integers and performed a number of "swaps" on them. Using the test harness "UnsafeMemory" I tested each implementation with shell commands of the form:

```
java UnsafeMemory Synchronized 8 1000000 6 5 6 3 0 3
```

In this shell command I specify which state implementation to be tested, the number of threads, number of swaps, the maximum value allowed in the array, and the initial values in the array.

Results:
To begin, I ran the test harness on the Synchronized and Null states to understand how changes to number of threads, number of swaps, max value, and contents of the array effected the time per transition. Observing the results of many test cases, I concluded that increasing the number of threads results in increased time per transition and increasing the number of swaps results in decreased time per transition. Varying the max value and contents of the array had no noticeable effect on the time per transition.

Synchronized:
I ran the following shell command 10 times.

```
java UnsafeMemory Synchronized 8 1000000 10 0 1 2 3 4 5 6 7 8 9
```

Average time/transition = 2912.151 ns/transition

The Synchronized implementation is 100% reliable. Because this implementation uses the `synchronized` keyword for the swap function this implementation is DRF. The `synchronized` keyword ensures that no thread can read/write to the same memory location at the same time. In this case it ensures that only one thread can execute the swap function at a time. This ensures reliability and that the code is DRF but sacrifices speed. This is the slowest implementation.

Null:
I ran the following shell command 10 times.

```
java UnsafeMemory Null 8 1000000 10 0 1 2 3 4 5 6 7 8 9
```

Average time/transition = 2048.985 ns/transition

The Null implementation is 100% reliable and DRF because it does not do anything. Swapping has no effect.

Unsynchronized:
I ran the following shell command 10 times.

```
java UnsafeMemory Unsynchronized 8 1000000 10 0 1 2 3 4 5 6 7 8 9
```

Average time/transition = 1596.649 ns/transition

This implementation is identical to that of Synchronized except it does not use the `synchronized` keyword. This implementation is completely unreliable as it always results in a sum mismatch such as `sum mismatch (45 != 34)`. It is not DRF because there is no synchronization being done. Threads are free to read and write from the byte array at any point. This implementation additionally can experience deadlocks. I experienced deadlocks occasionally when running the command:

```
java UnsafeMemory Unsynchronized 8 10000 6 5 6 3 0 3
```

Overall, this implementation is extremely fast, with time per transition nearly half that of the Synchronized implementation, because it does not have the bottleneck caused by the `synchronized` keyword that is present in the Synchronized implementation. One

interesting result I observed in comparing Unsynchronized to Synchronized was when the thread count was increased to 16, Unsynchronized did not have much of a speed increase.

GetNSet:
I originally tried to run the shell command with the same parameters as the above cases but I experienced too many deadlocks to get meaningful results. Thus, to test GetNSet, I changed the thread count to 8 and reduced the number of swaps to 100000. I ran the following shell command 10 times.

```
java UnsafeMemory GetNSet 8 100000  10 0 1 2 4 5 6 7 8 9
```

Average time/transition = 3282.904 ns/transition

To compare these results to Synchronized and Unsynchronized I tested Synchronized and Unsynchronized with the parameters used to test GetNSet. The average time/transition for Synchronized and Unsynchronized were 4965.19 ns/transition and 1712.64 ns/transition, respectively. As expected, the speed of GetNSet is in between that of Synchronized and Unsynchronized. GetNSet uses an Atomic Integer Array and makes use of the `get` and `set` methods of `java.util.concurrent.atomic.AtomicIntegerArray`. While this ensures atomic access to the array, data races are still possible because multiple threads will interleave their accesses to the array with the `get` and `set` methods resulting in elements in the array being incorrectly written and read. Thus GetNSet is not DRF. Deadlocks are also possible.

BetterSafe:
I ran the following command 10 times.
```
java UnsafeMemory BetterSafe 8 100 10 0 1 2 3 4 5 6 7 8 9
```

Average time/transition = 294940.2 ns/transition

To implement BetterSafe, I used a ReentrantLock to lock to achieve synchronization. The ReentrantLock provides better performance than using the `synchronized` keyword because when using `synchronized`, threads cannot be interrupted while waiting to acquire a lock. This causes a bottleneck that the lock interruptibility of ReentrantLock avoids. When a thread is waiting for a lock, it can be interrupted and pass use of the CPU to another thread so that work can be done while a thread waits for the lock to become available. This implementation is 100% reliable and DRF because no thread can execute swap until it acquires the lock.

BetterSorry:
I ran the following command 10 times.

```
java UnsafeMemory BetterSorry 8 100 10 0 1 2 3 4 5 6 7 8 9
```

Average time/transition = 211711.6 ns/transition

To implement BetterSorry, I used a similar implementation to that of GetNSet using an Atomic Integer Array. However, instead of using the `get` and `set` methods to update the values in the array when executing a swap I used the methods `decrementAndGet` and `incrementAndGet`. These methods allowed me to atomically decrement and increment, respectively, by one the element at a specific index in the atomic integer array. Because the increment and decrement are each done in a single method call this implementation is faster than GetNSet and is additionally more reliable. However, it is not DRF because it has the same issue of interleaving threads that is present in GetNSet. Data races are less likely in BetterSorry than in GetNSet because using the `decrementAndGet` and `incrementAndGet` methods reduces the number of methods calls and thus opportunities for interleaving. I found that reducing the number of swaps increases the likelihood of a data race. I was able to cause data races using the following shell command.

```
java UnsafeMemory BetterSorry 8 10 10 0 1 2 3 4 5 6 7 8 9
```

producing the following output.

```
Threads average 1.78385e+06 ns/transition
output too large (12 != 10)
```

Conclusion:
In conclusion, in terms of speed the different implementations rank as follows starting with the fastest implementation: Unsynchronized, BetterSorry, BetterSafe, GetNSet, Synchronized. In terms of reliability the different implementations rank as follows starting with the most reliable: Synchronized = BetterSafe, BetterSorry, GetNSet, Unsynchronized. In the case of selecting an implementation for GDI's application, I think the best suited implementation is BetterSorry because it is the fastest while still providing a decent level of reliability.