CS 131 HW3

Ziheng Zhou
604171655

Running Environment:

I test my code on the SEASNET machines from UCLA, with the following Java info:

- Java version "1.8.0_45"
- Java(TM) SE Runtime Environment (build 1.8.0_45-b14)
- Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02, mixed mode)

And the machine info:

- 16 Xeon E5620 4-core CPUs
- Each CPU 2.40GHz
- 32 GB memory

Analysis:

Unsynchronized: Here I just simply removed the Synchronized keyword from the code from SynchronizedState. It's not DRF. From the tests, it's doesn't behave reliable: it would stuck when the number of transitions is getting large, and frequently has race conditions. But the performance gets almost 3X faster.

```
$ java UnsafeMemory Unsynchronized 8 100000 100 90 40 60 40 30 50
 Threads average 1703.17 ns/transition
 sum mismatch (310 != 589)
$ java UnsafeMemory Synchronized 8 100000 100 90 40 60 40 30 50 Threads average
4541.22 ns/transition
```

GetNSet: Here I used "AtomicIntegerArray" to implement it with the methods get(i) and set(I, new_value). This approach takes almost twice the time as that of Unsynchronized, but still faster than Synchronized approach. However, this is not DRF because we only ensure atomic access on get and set, separately. So other threads can interfere in the middle. And it exhibits frequent mismatch too.

```
$java UnsafeMemory Synchronized 8 100000 100 90 40 60 40 30 50 Threads average
4541.22 ns/transition
$java UnsafeMemory GetNSet 8 100000 100 90 40 60 40 30 50
 Threads average 2707.98 ns/transition
```

BetterSafe: Here I used getAndIncrement and getAndDecrement from AtomicIntegerArray to ensure atomic access. This approach is DRF because both the read and write are glued together as one single atomic action. This approach is almost twice fast than Synchronized because it has much smaller critical section – only the writing memory parts are enclosed.

```
$ java UnsafeMemory Synchronized 32 10000000 100 90 40 60 40 30 50
 Threads average 6555.22 ns/transition
$ java UnsafeMemory BetterSafe 32 10000000 100 90 40 60 40 30 50
 Threads average 4562.18 ns/transition
```

BetterSorry: Here I used Reentrantlock to achieve better reliability. And actually in this way we can guarantee reliability because it locks the writing memory parts. And it's faster than BetterSafe because it has smaller critical section, I guess. But AtomicIntegerArray should be able to leverage CPU atomic instructions and should be faster – but the data shows Reentrantlock is faster:

```
$java UnsafeMemory BetterSafe 32 1000000 100 90 40 60 40 30 50
 Threads average 7660.25 ns/transition
$ java UnsafeMemory BetterSorry 32 1000000 100 90 40 60 40 30 50
 Threads average 6699.42 ns/transition
```

Problem:

The main problem is that when the transition numbers gets very big, the states that are not DRF would get stuck frequently – should be because all the values become 0 or maxvalue and therefore stuck in the loop. So I have to use smaller transition numbers. But this way the data may get inaccurate because there's always some overhead when creating a new thread. This cost would be less relevant if we have a large amount of calculations to do. But in not so large calculations those overhead would matter (but why having this many threads when the calculations are not that many?).