HW3 Report

Platform Stats:

- Java version 1.8.0_45
- CPU:
    - 16 processors
    - Intel Xeon CPU @ 2.4 GHz (model No. 44)
    - 4 cores
    - Cache size/processor: 12288 KB
    - Address sizes: 40 bits physical, 48 bits virtual
- Memory:
    - Total Memory: ~ 33 GB, Free Memory: ~ 17 GB

Test Result:

	Synchronized	Null	Unsynchronized	GetNSet	BetterSafe	BetterSorry
Avg time/ transition	5407 ns	2934 ns	2845 ns	3312 ns	3031 ns	2698 ns
Reliability	100%	100%	~0%	~0%	100%	~100%

- Test parameters: # swaps = 1 million, # threads = 16, array size = 100, maxval = 20

Synchronized:

This model uses the keyword “synchronized” when declaring the swap function. This ensure that only one thread can execute swap, which prevents race conditions where two threads interfere on the same object. This “Synchronized” model is DRF.

However, Synchronized model is not preferred in terms of performance. If a first thread executes the swap code, then a second thread also tries to execute the same method, it will block until the first thread exits. Therefore, a lot of time is spent waiting for other threads to finish.

Unsynchronized:

The “Unsynchronized” model uses the same implementation as “Synchronized”, except that it omit the use of keyword “synchronized” when declaring the swap method. And this would cause serious problems. Since nothing stops race conditions, it is possible for two threads to read or write the same element. The increment or decrement would not be atomic. As a result, the sum of the array would change after the swap. “Unsynchronized” model is not DRF.

The result of the swap test on this Unsynchronized model verifies its unreliability. This model fails to maintain the same array sum in all 10 runs. Since the number of swaps was set to 1 million, it is natural for this model to have race condition in some swaps.

GetNSet:

The “GetNSet” model declares the array as AtomicIntegerArray, so that it can use the atomic functions of this class get() and set() to access and change the array elements. However, the atomic methods does not prevent race condition between different threads. This is because get() and set()

are two separated methods. When the swap method changes an element in the array, it first invokes get() to obtain an old value of an element. Then it invokes set() to increment or decrement from that value. Therefore, swapping is really a two-step procedure. There is no guarantee about what happens between these two methods are invoked.

For example, after a first thread finished calling get(), a second thread can invoke get() on the same element. Later when these two threads changes value of the element, they will use the same old value, and as a result the final value will be incremented or decremented by 1 after two swaps.

When testing the "GetNSet" model, I had to decrease the number of swaps to 10 or 100 instead of 1 million, and increase maxval to 100. Otherwise, the error rate is so high that all values in the array would be either 0 or maxval. In this case, the swap method fails to find an element to increment or decrement, and the swap test will loop forever. I also find that this model is even more unreliable than "Unsynchronized." If number of swaps is set to 10, "GetNSet" has sum mismatch in ~50% of runs. Therefore, "GetNSet" is not DRF.

BetterSafe:

Implementation of this model uses a lock to prevent thread interference. It maintains a lock object in its interface. Since the whole block of code in swap method is wrapped inside lock() and unlock(), this model ensures single-threaded access in a way similar to the "Synchronized" model. Therefore, "BetterSafe" model is definitely DRF.

When running the swap test on this model, I used all parameters stated in the beginning of this report. I observed a ~40% performance improvement over the "Synchronized" model. I believe this is because that maintaining a lock in the class reduces the overhead for creating and keeping track of the lock.

BetterSorry

I implemented "BetterSorry" with an array of AtomicIntegers. That is, the access to the array is not atomic, but the access to each element is. With this change, multiple threads can now access the array at the same time, as long as they access different elements. Therefore, a performance improved can be expected over the "GetNSet" model.

With "Unsynchronized" model, two threads can read or write the same element in the array at the same time. However, "BetterSorry" prevents this problem because the access to each element is guaranteed to be atomic.

With the regular tests, this model seems like a DRF. However, in an extreme case, if the array of size n contains one "0" and n-1 maxval, this model would fail the sum matching test. This is because a second thread can read or write element i or j after the first thread finishes getAndDecrement() but have not started getAndIncrement(). Despite this, this model still have a reliability close to 100%, much better than either Unsynchronized or GetNSet.

Conclusion: The BetterSorry model seems to be the preferred model for GDI, since the program is expected to run as fast as possible and can tolerate a small number of race conditions.