Report

BetterSafe implementation:

ReentrantLock confines exclusive assess and execution a block of code among muti-threads. So it guarantee 100% reliability.
The performance of BetterSafe is better than synchronized version which is accordant with some articles suggest that with increasing number of threads, using lock generally offers a better performance than using synchronize keyword. One possible reason is that in addition to have the same concurrency and memory semantics as synchronized, ReentrantLock also provides additional features that offer better performance under heavy contention. For example, it provides tryLock() method which acquires lock only when it is not held by other thread. In this way, it reduced the chances of blocking a thread for waiting on a lock.

BetterSorry implementation:

The reason why BetterSorry should be faster than BetterSafe is because BetterSafe simply lock the whole block of function which literally eliminate parallelism among threads, while BetterSafe using atomic operations does not halt parallelism as much as the function level lock does(however, the test result suggest the opposite). Besides, it is more reliable than Unsychronized, because in BetterSorry I use an atomic instruction getAndIncrement(), getAndDecrement() to avoid the race condition that could happen in interleave of incrementing a variable and of decrementing a variable.

However, this implementation is not DRF. For example, when two threads both pass the value condition check. and value[i] = 1. Then one thread calls getAndDecrement(i), so value i decreased to 0. Subsequently, the other thread calls getAndDecrement(i) which realist value i to -1, however, both thread still return true even if value i violate the requirement of being positive.
Even though the race condition do exist, however, it is very unlikely to catch. Because the only chance is when more than two threads operation on same integer in the array and the value has to be a boundary value, either 1 or max_value - 1. Then the two threads passed the range check, and do the increment or decrement operation on the boundary value to make it below 0 or exceed max value.
All these listed requirement make it very unlikely to occur that particular race condition.

Test Result:

All the testing were running on SEASnet GNU/Linux servers, with Java version 1.8.0_51-b16.
Each model were running on 3 test cases and 10 iterations:
test1:java UnsafeMemory GetNSet 10 10000000 120 1 10 20 30 40 50 60 70 80 80 100 110 120 (this help with speed test)

test2:java UnsafeMemory Unsynchronized 20 1000000 3 1 3 1 3 1 3 1 3 1
(this help with check if any race condition happens that break integer range assumption)

NullSate:
It is DFR since it simply return true.
Test1: Threads average 220.169 ns/transition
Test2: Threads average 5736.73 ns/transition

Synchronized:
It is DFR, because swap method is declared as a synchronized method which will be synchronized by JVM to gaurantee DFR. However, it suffers in relatively slow performance because the overhead of synchronization.
Test1: Threads average 2807.48 ns/transition
Test2: Threads average 11054.6 ns/transition

Unsynchronized:
It is obviously not a DFR implementation, race condition can happens among different threads' reads and writes. However, it runs as the fastest one.
Test1:Threads average 2234.25 ns/transition
Test2: hang forever(because all number out of range(due to race condition) before accomplishing the target successful swap)

GetNSet:
It has some atomic operations to help reduce race condition, but it doesn't avoid race condition entirely because race condition can happen between get() and set(). However, it slightly faster than Synchronized one.
Test1: Threads average 2254.15 ns/transition; sum mismatch (771 != 38)
Test2: hang forever(because all number out of range(due to race condition) before accomplishing the target successful swap)

BetterSafe:
Test1: Threads average 845.926 ns/transition
Test2: Threads average 4045.28 ns/transition

BetterSorry:
Test1: Threads average 2666.68 ns/transition
Test2: Threads average 4789.21 ns/transition

Noteworthy, it is abnormal that BetterSafe even runs faster than NullSate, however, I can not figure out how is happens!

Theoretically, I would choose BetterSorry model the GDI application, since it should have a good balance between time efficiency and reliability. However, from the test result, the BetterSafe have the best performance in speed and it guarantee the reliability as well.