1. Base States

1.1 Null

This state does not swap at all. It is used for figuring out the baseline times.

```
public boolean swap(int a, int b) {
    return true;
}
```

1.2 Unsynchronized

This state implements the swapping without any sort of synchronization.

```
 public boolean swap(int a, int b) {
       return super.swap(a,b);
}
```

1.3 Synchronized

This state implements the swapping using the synchronized keyword.

```
 public synchronized boolean swap(int a, int b) {
       return super.swap(a,b);
   }
```

1.4 GetNSet

This state implements the swapping using the java library for atomic arrays. The atomic array will always change values atomically. It guarantees that array values are incremented and decremented correctly. However, it does not guarantee that array values are the same as when they were last checked by the if statement.

2. BetterSafe

2.1 Multiple Locks

I had a lot of difficulty implementing BetterSafe. Initially, I tried to implement BetterSafe with an array of reentrant locks with length equal to the length of the values array. Whenever a swap occurred, the thread would attempt to gain two locks: the lock to the increment index and the lock to the decrement index of the array. However, I kept on running into deadlocks since I was asking the threads to obtain two locks at once. For example, thread A might hold lock 1 and wait for lock 2, while thread B might hold lock 2 and wait for lock 1. To solve the deadlock issue, I tried asking threads to always hold the lock of the higher index before attempting the lock of the lower index. Although this successfully prevented deadlocks, it caused a lot of overhead. The many lock approach was only faster than synchronized when the values array was length fifty or longer.

```
  public boolean swap(int a, int b) {
       int high=a>b? a :b;
       int low=a>b? b :a;
      locks[high].lock();
      locks[low].lock();
       boolean res=super.swap(a,b);
      locks[low].unlock();
      locks[high].unlock();
       return res;
   }
```

2.2 Single Lock

Since I was looking for a solution which was almost always better than the synchronized, I switched to a single reentrant lock approach. The single lock locked the same critical section which the synchronized approach locked. However, because it used lock api rather than the keyword synchronized, it was faster. This solution is not as creative as the many lock solution, but it is effective.

```
 public boolean swap(int a, int b) {
       lock.lock();
        boolean res=super.swap(a,b);
       lock.unlock();
        return res;
}
```

3. Reliability

3.1 GetNSet

GetNSet works the vast majority of the time when max value is significantly less than 127. However, it is not DRF safe because it doesn't prevent the values in the array from changing between the time which a thread verifies that a value is valid for incrementing and decrementing and the time which the thread increments and decrements the value. Thus, sometimes a value goes slightly over the max value or slightly under zero. This is vulnerable to byte overflow failures. The following code will almost always fail with GetNSet because of byte overflow.

java UnsafeMemory GetNSet 28 10000000 127 126 126 126 126 126

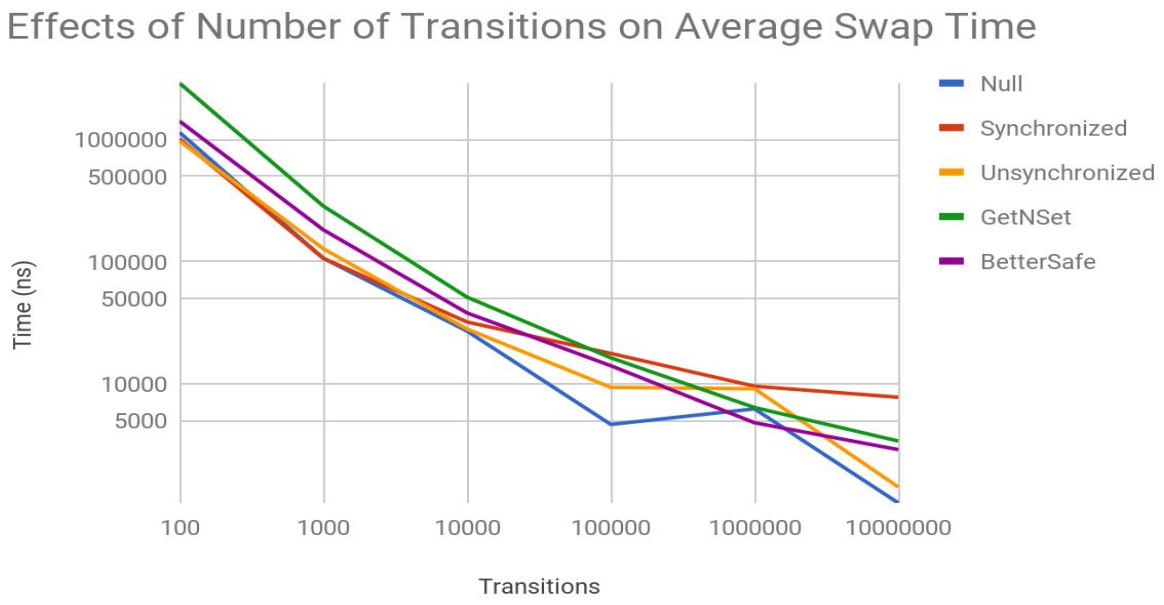
3.2 Synchronized and BetterSafe

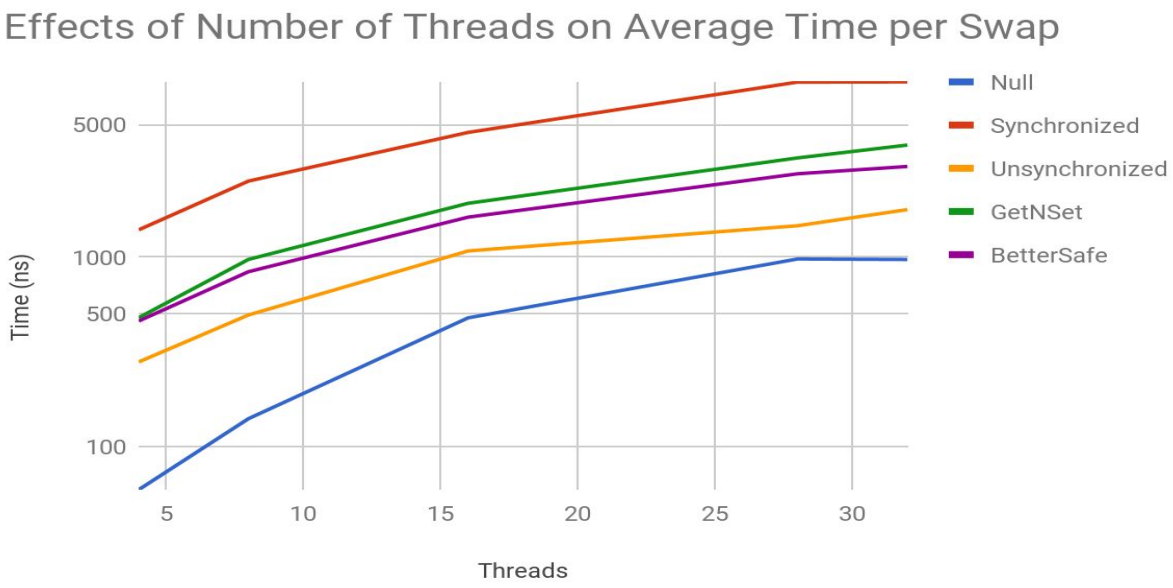
Out of the three states with some sort of synchronization, Synchronized and BetterSafe are DRF safe because they both fully lock up super.swap(a,b). BetterSafe is faster than synchronized, however, since BetterSafe uses the locks API which is more efficient than the synchronized keyword.

4. Transitions

In order to study the effects of different numbers of transitions on the average swap time, I tested a set of transition numbers on an array of length 5 and max value of 10 while using 28 threads. The results are graphed below.

Note that at very few transitions, BetterSafe and GetNSet, the two states which import libraries, take significantly longer than the other three states which don't import any libraries. This data hints that a large amount of time at the beginning of each run is spent loading and caching code and data. The time of the Null state is a good indicator of loading time. At lower numbers of transitions, this loading time is far exceeds swap time. Only at very high numbers of transitions does the swap time start amortizing the loading time. This means that in order to study the swap speed of each state, the number of transitions should be at least 10,000,000 in order to amortize the loading time.





5. Threads

In order to study the effects of different numbers of threads on the average swap time, I tested a set of thread numbers on 10,000,000 swaps using an array of length 5 and max value 10. The results are graphed below.

According to this data, increasing the number of threads slows down all states, including the null state. Even the null state time increases because at higher numbers of threads, chances are higher that any particular thread is waiting to run or already done, rather than processing swaps. Therefore at higher thread numbers, more time per thread is inevitably wasted, even if collisions don't occur. Even though all states run into this same issue, there's a clear hierarchy in terms of speed. The Unsynchronized state is the fastest and the Synchronized state is the slowest. Between them, Bettersafe and GetNSet are about the same speed.

Surprisingly, GetNSet isn't faster than BetterSafe even though it uses much finer grained locking. This is because the atomic functions decrementAndGet and incrementAndGet from GetNSet have significant overhead. Furthermore, it's also because the arrays I tested on were very small and GetNSet is more effective on large arrays.

6. Infinite Loops

In order for a thread to finish the its work, it must complete a certain number of successful swaps. Furthermore, the thread cannot possibly complete more successful swaps if all elements of the array are at least max or if all elements of the array are at most 0. Assuming the initial array is valid, DRF states will never allow the array to reach this invalid state. However, the two non-DRF states might. Since the states aren't synchronized, they may decrement one number more than they increment another, force a 127 number to overflow into a -128, or otherwise change the balance of the array. In order to end the infinite loops, I added the following logic to my unsynchronized state.

If swapping failed, check whether all numbers are at least max value or at most zero. If so, reset all values in the array to half of max value.