# Michael Skirpan - Homework 3

## Problem 1

Informally the read method could look like so:

```
public int read(){
    boolean[] reedit1 = new boolean[N];
    boolean[] reddit2 = new boolean[N];
    boolean[] reddit3 = new boolean[N];

        for (int i = 3N; i >2N; i--){
            reddit1[i%N] = b[i];
        }
         for (int i = 2N; i >N; i--){
            reddit2[i%N] = b[i];
        }
        for (int i = N; i >=0; i--){
            reddit3[i%N] = b[i];
        }

        if(reedit1 == reddit2){
            booleanArrayToInt(reddit1);
            return reddit1;
        } else if (reddit2 == reddit3){
            booleanArrayToInt(reddit3);
            return reddit3;
        } else {
            booleanArrayToInt(reddit1);
            return reddit1;
        }
```

The above code should cover all possible overlap cases. There are three possible cases that occur during a read()/write() overlap:

1. the writer is in the first for loop; meaning segments 2 and 3 of the boolean array are equal and segment 1 is different.
2. the writer is in the second for loop, in which case all three segments of the boolean array are

different.
3. the writer is in the third for loop; meaning segments 1 and 2 of the boolean array are equal and 3segment is different.

The only other cases would be that the writer hasn't started writing yet or that the writer has finished its write, which, either way, means all segments are equal.

In case (1) we want to return segment 3 (or 2) since we don't know whether or not the writer has completed writing to segment 1, and thus we'll return the original value of the array (likely 0 since it's a write-once method).

In case (2), we are safe to return the value found in segment 1 because the fact that segments are all unequal means that a writer has come in and finished segment 1 and is in the process of writing segment 2.

In case (3), we can return the value of segment 1 since we know the writer has completed segments 1 and 2.

## Problem 2

Code is attached as 4 class files:
BoolArray.java
SharedCounter.java
FooBar.java
TTASLock.java

Running on the ecen5033 server 14 times, throwing out the high and low outliers, average runtime was:

*Boolean Array Method*: .198s
*Shared Counter Method*: .167s

The Shared Counter performs better. We could have expected this to be the case because the boundary section of the Boolean Array has each thread spinning until the thread with the prior index finishes; imposing a strong order on the threads. This means if any thread takes over another there will be extra wait time.

There is also a possible slow down in the Boolean Array method due to the fact that each time the shared array is updated it invalidates each threads cached local copy of the array. This means there are lots of cache misses which then require more updates to propagate through the bus.

Using a shared counter avoids both of these problems by a) allowing threads to finish in any order, minimizing dead time and b) having threads spin on their own cached copy of the lock meaning the only time they have to clog up the bus is when a cache miss hap pence once the lock opens up.