Homework 3

Mathew Kurian (mk26473)
Kapil Gowru (krg766)

Question 1

True. Replacing a safe Boolean SRSW register array with an array of regular M-valued SRSW registers yields a regular M-valued MRSW register. Regular registers satisfy the following conditions: (1) No read call returns a value from the future. (2) No read call returns a value from the distant past, that is, one that precedes the most recently written non-overlapping value. (3) If a read overlaps a write, it returns either the old or the new value. If `read()` and `write()` calls to do not overlap, `read()` returns the most recent value. Under concurrent `read()` and `write()`, the `read()` operation must return the value of the register before the `write()` operation or the value currently being written to the register.

1. If new value has not been written yet, the old value is returned.
2. If new value has already been written, the new value is returned.
3. If new value is being written as it is being read, it can return either the old value or the new value.

All conditions are satisfied, thus the construction does a yield a regular M-valued MRSW register.

```
// Example
Read()
 for j = 0 to M
 if table[j].read() = 1 then return(j)
```

```
Write(v)
 table[v]write(1);
 for j=v-1 downto 0
    table[j].write(0);
```

Question 2

False, replacing a safe Boolean MRSW register with a safe M-valued MRSW register does not yield a regular M-valued MRSW register. A regular M-valued MRSW register needs to return either the old or the new value in case of a `read()` overlapping a `write()`. In the original case, the `new != old` check is binary so it must equal the old or the new. This will not be the same case with M-valued registers, since an overlapping `write()` can return any element. Therefore the construction is not yield a regular M-valued MRSW register.

Question 3

The principle difference between atomic and regular registers is that atomic registers cannot switch between the values returned when reading during a `write()`. In Peterson's algorithm, if threads A and B both try to acquire the lock, overlaps may occur in line `8` or `9`.

If the overlap occurs with B in line `8` while A is spinning in 10, the follow cases may occur:

Case	Occurence
1	A reads old value of `flag[i]` –> A enters critical section
2	A reads new value of `flag[i]` –> A continues spinning

This leads to two further cases with B in line `9`:

Case	Occurence
1	A reads old value of victim –> A spins until B finishes writing
2	A reads new value of victim –> A enters critical section

In all cases, B will spin at line 10 because the value of flag[i] is never changed and the B has completed the write to victim before reaching line 10. The algorithm is therefore correct even if regular registers are used.

Question 4

To accomplish that, we need two arrays:

```
ConsensusObject binaryConsensusObjects [N]; // Each index contains 0 or 1 to make a binary number of length N
AtomicIntegers activeThread [N]; // Marks the active object
```

Each thread will first `compareAndSet` the corresponding activeThread before proposing a value for the Nth consensus object. The decided value by each binary object will be set if and only if the corresponding activeThread is set to false.

```
boolean value = false;
Integer decide(boolean b){
   if(activeThread[index].compareAndSet(-1, currentThreadId)){
      value = b;
   }
   return activeThread[index].get(); // return the id of last thread that has written
}
```

The threads will continue the process for each binary consesnus object *iff* the value returned by `decide()` is equal to the its thread id.

The final value can be calculated by reading the decided bits in order.

Question 5

In `FifoConsensus` with a focus on `peek()`, the `decide()` method will produce a wait-free solution because there are no loops as well as no `dequeue()` which means that it is not possible for the value of the multiple threads can read different values when calling the `peek()` function. As a result, the `peek()` method has an infinite consensus number. To put it simpler, infinite threads can agree upon the same return value for `peek()` since there is no `dequeue()`

Question 6

Algorithms and performance time for 8 threads counting to 120000.

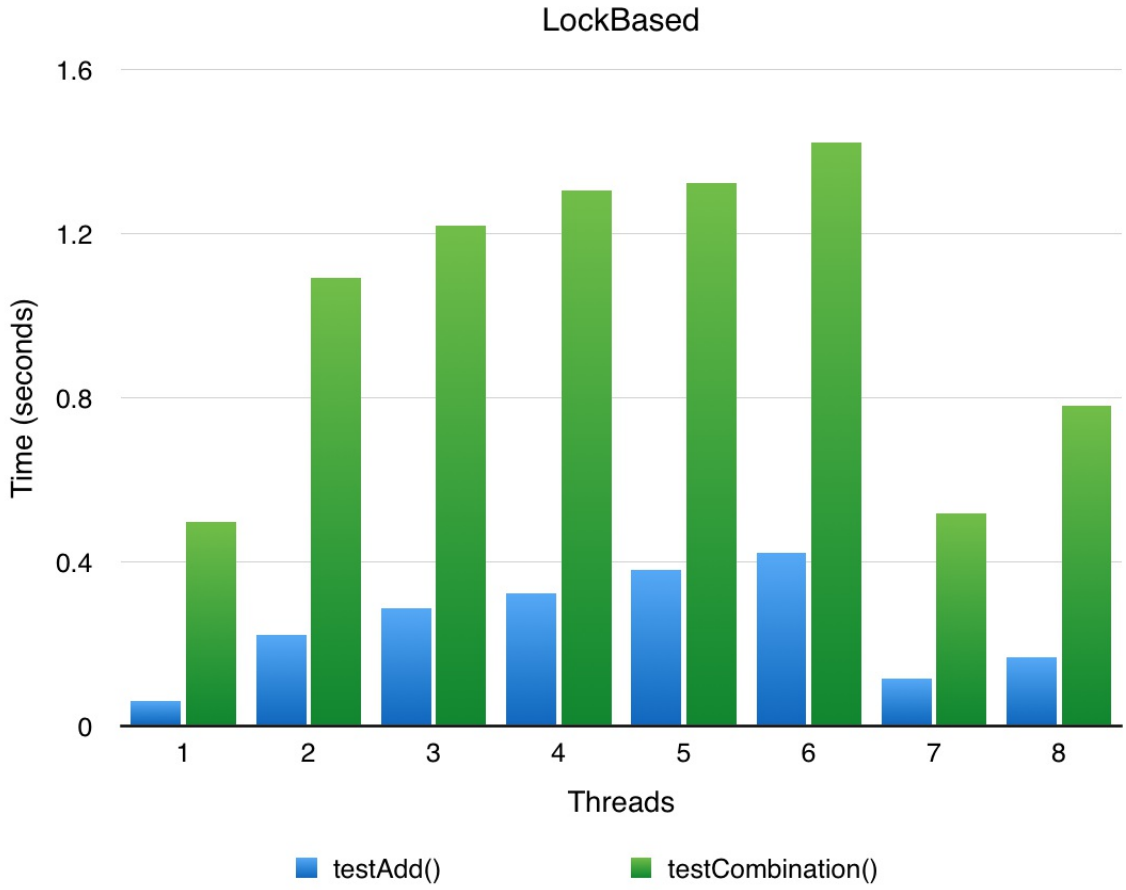
Algorithm	**Time**
CLH Lock	0.21883575 ms
MCS Lock	0.19976075 ms
Reentrant Lock	0.0511645 ms

Question 7 Graphs

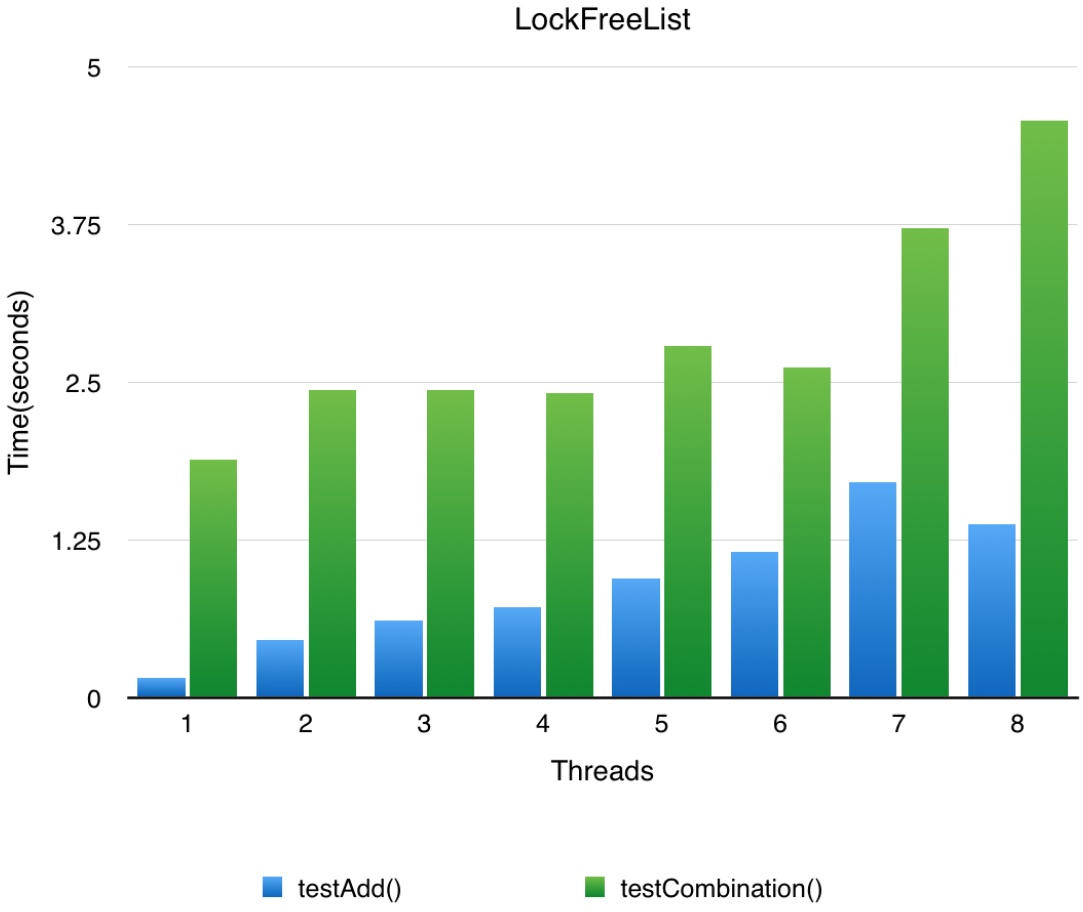
The Lock Based link list and the Lock-Free Synchronization link list are two

different concurrent implementations of a linked list. The following graphs show their performance utilizing two different test methods. The `testAdd()` method just adds numbers to the linked list. The `testCombination()` method does a combination of adding, removing, and contains.

Graph 1 - Lock Based Linked List Performance



Graph 2 - Lock-Free Synchronization Linked List


LockFreeList
5
3.75
2.5
1.25
0
Time(seconds)
1
2
3
4
5
6
7
8
Threads
testAdd()
testCombination()