ECSE 420 Assignment 2
Group 16

Student1 : Erdong Luo
ID: 260778475

Student2 : Sameen Mahtab
ID: 260737048

Q1

1.1 See Filter.java
1.2 Yes, Filter allows some threads to overtake others an arbitrary number of times. There is concept called bounded waiting. r-bounded waiting implies that a thread cannot overtake another thread more than r times. But for filter algorithm, there is no value for “r”. This means a thread can be arbitrarily overtaken.

Answer: there is no value of “r”

1.3 See BakeryLock.java
1.4 The bakery algorithm does not allow for a thread to overtake another thread an arbitrary number of times. Bakery algorithm provides first-come-first-served for n threads. A first-come-first-served has r = 0. This means no thread can overtake another.
1.5 We can create a pool of threads. (the number of threads should be largher than 5). Every time, a thread enters the critical section, other threads should not be able to lock. Or mutual exclusion is failed.
1.6
Thread 10 acquire Lock
Thread 11 acquire Lock
Thread 12 acquire Lock
Thread 14 acquire Lock
Thread 13 acquire Lock
Thread 11 Hold Lock
Thread 11 Release lock
Thread 12 Hold Lock
Thread 12 Release lock
Thread 13 Hold Lock
Thread 13 Release lock
Thread 10 Hold Lock
Thread 10 Release lock
Thread 14 Hold Lock
Thread 14 Release lock

It provide mutual exclusion.

Q2:

Yes, LockOne and LockTwo satisfy two-thread mutual exclusion when “flag” and “victim” are replaced my regular registers.
In an atomic register, read() will always return the last written value. In a regular register however, if a read overlaps a write then it may return that written value, or possibly the previous one.

```
class LockOne implements Lock {
  private boolean[] flag = new boolean[2];
  // thread-local index, 0 or 1
  public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
    flag[i] = true;
    while (flag[j]) {}        // wait
  }
  public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;
  }
}
```

Figure 2.4 The LockOne algorithm.

For example, let's say we have two threads (A and B). For the 'flickering' to occur, then flag[A] must been read and written in an overlapping fashion, or the same must occur for flag[B]. This means that flag[A] must be written to true by thread A and simultaneously, Thread[B] must read the value. If the old value is read, then B will enter its critical section. However, when the write is completed, then B will eventually enter its critical section. Also, since flag[B] is already set to true, thread A will not be able to enter its critical section and mutual exclusion will not be violated. If the new value of flag[A] is read, then B will not enter its critical section and for the same reason as stated above, A will not be able to e

```
class LockTwo implements Lock {
  private int victim;
  public void lock() {
    int i = ThreadID.get();
    victim = i;              // let the other go first
    while (victim == i) {}   // wait
  }
  public void unlock() {}
}
```

Figure 2.5 The LockTwo algorithm.

If A marks itself as the victim and B reads victim, then B will not enter its critical section as it believes that it is the victim (reading old value). Once A finishes writing, in a subsequent read, B will be able to enter its critical section. A will not be able to enter its critical section as it has set itself as the victim. Therefor mutual exclusion is satisfied. If B reads the new value, then the algorithm works as expected so mutual exclusion is satisfied.

Q3:

3.1

Suppose there are two threads, thread A and thread B. Both of them want to enter the critical section at the same time. Without the loss of generality, we assume that A arrive first. When A arrived, "busy" must have been set to true. When B arrives, turn is set to B, and since busy is true, B will be stuck in its while loop. So, if B want to enter the critical section, the busy flag must have been set to false. One of the thread A or B must have called unlock (). This is contradicting to our assumption.

3.2

Concurrent executions will not experience deadlock because as soon as a new thread enters, it will set *turn* to its *ThreadID*, allowing another thread to break out of the *while* loop.
However sequential executions can experience deadlock. This is because *turn* will never be changed to another *ThreadID*, hence the initial thread will stay in the while loop while the other thread is waiting for it to exit.

3.3

It is not starvation-free. Assume that a thread A is in the critical section and all the other threads are in the while loop (**while** (turn = me || busy);). When A exits the critical section and unlock, there should be another thread entering the critical section. But there isn't a way to select the next thread, thus causing starvation.

Q4

4.1. History (a) is sequentially consistent in the following order of execution:
A: r.write(0)
B r.write(1)
A: r.read(1)
A: r.write(2)
B: r.read(2)
C: r.read(2)
C: r.write(3)

History (a) is not linearizable. C: r.write(3) happens before B: r.read(2) and C: r.write(3) might happen before A: r.write(2) so B: r.read(2) cannot execute properly. Flattening the 3 lines will not produce a sequential execution.

History (b) is not sequentially consistent. B: r.write(1) has to occur before B: r.read(2) and C: r.write(2) has to occur after B: r.write(1) and before B: r.read(2) and C: r.read(1).

For B: r.read(2) we need the sequence:
B: r.write(1)
C: r.write(2)
B: r.read(2)

For C: r.read(1) we need the sequence:
C: r.write(2)
B: r.write(1)
C: r.read(1)

But the above two sequences are contradictory. B: r.write(1) will be overwritten by C:r.write(2) so C: r.read(1) cannot happen.

Since sequence (b) is not sequentially consistent it is also not linearizable.

Q5:

5.1. In this class, v is a volatile boolean. This means that the value of v is written to and read from main memory. If thread A changes both v and x, thread B will only be able to see the change to v. If thread A calls writer() it will set x to42 and v to true, the change to x this is not visible to thread B.

After that, if thread B calls reader() method without calling writer(), it will have v as true but x will remain 0. So the reader method will divide by 0.

5.2. Division by 0 will not occur if both v and x are volatile. If thread A calls writer() and then thread B calls reader(), both of the updated values of v and x will be visible to thread B.

If neither are volatile, division by 0 might occur. JVM does not guarantee that the variables will be updated in sequence, as a result, v might be updated first by thread A and then thread A might be interrupted by thread B before setting x to 42. And then if thread B calls reader() without calling writer(), there will be a division by 0.

Q6:

6.1. Changing the loop at line 11 will cause the read() method to work incorrectly as the write() method will not update the lower bits. The values after x will be false. For instance, if we have two subsequent write() operations, since read() returns values of the lowest index, every subsequent read() will only return the value of the first write and will not return the second write. The construction will <u>not</u> remain a *regular* M-valued MRSW register.

6.2. Similar to 6.1, changing the loop at line 11 means that the write() method will not update the lower bits. The values after x will be set to false. As r_bit[0] is initialized as true, the read() method will keep on returning 0. The construction will <u>not</u> yield a *safe* M-valued MRSW register.

7: Supposing there exists a protocol for binary consensus using atomic registers for n threads. A two-thread binary consensus protocol can be made by having n=2; 2 of the threads to progress, the remaining threads to hold. This brings forth the contradiction that: two-thread binary consensus is impossible. Hence, binary consensus is also impossible for n-threads.

8: Supposing that there exists a protocol for consensus over *k* values. One value could be mapped to 1 and the other to 0, the consensus protocol will be reduced to binary values and will serve as a binary consensus protocol. This brings forth the contradiction that: binary consensus is impossible. Hence, consensus over k-values is impossible.