Spiro Mavroidakos - 260689391
Jastaj Virdee - 260689027

ECSE 420- Assignment 2

Q1

1.1

Refer to appendix section A.1

1.2

Yes filter lock does allow other threads to arbitrarily overtake other threads. This can be seen by using r-bounded waiting. r-bounded waiting implies that a thread (i.e. thread A) cannot overtake another thread (i.e. thread b) more than r times. For example, first-come-first-served, has a bounded waiting of r = 0 as it is impossible for a thread to overtake another. However, for the filter algorithm, we saw in class that there is no value of r which means that a thread can be arbitrarily overtaken.

1.3

Refer to appendix A.2

1.4

The bakery algorithm does not allow for a thread to overtake another thread an arbitrary number of times. The bakery algorithm is a first-come-first-served algorithm and therefore has an r = 0. With r-bounded waiting, an r = 0 means that no thread can overtake another which is by definition in a first-come-first-served approach. The bakery algorithms works by a client (thread), taking a number and then waiting until all lower numbers have been served. This is why a thread cannot overtake another thread an arbitrary number of times.

1.5

A test to verify that mutual exclusion is satisfied is to create an arbitrary amount of threads (for testing, a number >= 5 will suffice) and have them all try to access the same shared resource. However, this resource will be protected using the bakery algorithm. Once the lock is obtained, it might be nice to have a print statement declaring that the lock has been obtained/released and a sleep statement to make sure that the other threads are not accessing it at the same times.

1.6

Refer to appendix A.3

Output of test with 5 threads

```
Thread with id = 13 acquiring Lock 1
Thread with id = 14 acquiring Lock 1
Thread with id = 16 acquiring Lock 1
Thread with id = 15 acquiring Lock 1
Thread with id = 12 acquiring Lock 1
Thread with id = 13 Holding Lock 1
Thread with id = 13 Releasing lock
Thread with id = 14 Holding Lock 1
Thread with id = 14 Releasing lock
Thread with id = 16 Holding Lock 1
```

```
Thread with id = 16 Releasing lock
Thread with id = 15 Holding Lock 1
Thread with id = 15 Releasing lock
Thread with id = 12 Holding Lock 1
Thread with id = 12 Releasing lock
```

As expected, there is no overtaking and mutual exclusion is satisfied.

Q2

Regular registers can cause a value to flicker between the old and new value when a read and write are overlapping. For a 2 thread mutual exclusion example, LockOne will satisfy mutual exclusion. 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 enter its critical section either, resulting in deadlock. However, this does not violate mutual exclusion.

LockTwo will also satisfy mutual exclusion for similar reasons to LockOne. 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

Assume CS_A^j overlaps CS_B^k

From the code:

- $Write_A(turn=A) \rightarrow write_A(busy=true)$
- $write_A(busy=true) \rightarrow read_A(turn==B) \rightarrow read_A(busy==true)$

From the assumption:

- $read_A(turn==B) \rightarrow write_B(turn=B)$
- $read_B(turn==A) \rightarrow write_A(turn=A)$

Combining all this:

- $Write_A(turn=A) \rightarrow write_A(busy=true) \rightarrow read_A(turn==B) \rightarrow write_B(turn=B) \rightarrow write_B(busy=true) \rightarrow read_B(turn==A) \rightarrow write_A(turn=A)$

We get a cycle, meaning that CS_A^j cannot overlap CS_B^k, hence there is mutual exclusion.

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

Thread A gets blocked if it repeatedly enters the while loop so that *turn==A* and *busy==true.* However when B enters it will set *turn==B*, allowing A to break out of the while loop. Therefore the A will not starve. However the last thread to execute will starve as it will always be its turn, which will keep it in the while loop.

Q4

Fig. 2 History (a)

History (a) is not linearizable because thread C needs to execute r.write(3) before B does r.read(2). Therefore in between these two events something must do r.write(2), in this case thread A is the only thread capable of this. However if thread A executes r.write(2) after C does r.write(3), then there is no way for C to execute r.read(2), because there is no thread that does r.write(2).

History (a) is sequentially consistent, here is a valid order of execution:
A: r.write(0) → B: r.write(1) → A: r.read(1), r.write(2) → B: r.read(2) → C: r.read(2), r.write(3)

Fig. 3 History (b)

History (b) is not linearizable because C must execute r.read(1) before B does r.read(2). However in between these two events there is no thread that does r.write(2).

History (b) is not sequentially consistent. To show this we can look at reading/writing (2). We can allow B to execute r.read(2) right after C does r.write(2). However it will then be impossible for C to execute r.read(1) as B's r.write(1) will be overwritten by C's r.write(2).

Q5

5.1

No, the reader method will never divide by zero. Volatile variables provide a visibility guarantee. This means that if thread A writes to a volatile variable, then thread B reads the same volatile variable, then every variable written by thread A will be visible to thread B meaning that they will be reread with the updated value.

5.2

Dividing by zero is not possible when both are volatile since volatile variables are always read from main memory. In this case, Boolean V is written after variable x so x will be updated before v. Since the value of v needs to be read prior to x, and x is updated prior to v in the writer method, this means that x will never be zero when the reader method is invoked.

If x and v are not volatile, then it is possible to divide by zero. This can happen if both x and v are cached with their initial value of zero and false respectively. If v gets its value updated in cache but x does not then when the reader method is invoked, it will divide by zero.

Q6

6.1

False, implementing the change at line 11 will cause the values after *x* to be *false*, leaving the values before *x* as is. This is problematic as the *read()* method will no longer function correctly. The *read()* method starts at the first index and continuously increments until it gets to a *true* value. If the values before *x* do not get set to *false*, then *read()* can potentially not return the index that was most recently written to. Instead it may return an old index, which is not what we want.

6.2

False, again after implementing the change, index *x* would be set to true, all values after *x* would be set to *false*, and all values before will remain unchanged. However this is not what we want for a safe Boolean MRSW. Instead we would want to write the same value to all of the indices, one at a time.

Q7

Suppose a protocol exists such that binary consensus using atomic registers is possible for *n* threads. This protocol would still work if only two out of the *n* threads progressed, essentially serving as a two thread binary consensus protocol. However this is a contradiction that "two-thread binary consensus is impossible", therefore binary consensus for *n* threads must also be impossible.

Q8

Suppose a protocol existed for k-value (k > 2), n-thread consensus. This protocol would still work if we restricted the values to binary values. This would essentially serve as a binary consensus protocol. However this is a contradiction that “binary consensus is impossible”, hence consensus over k-values (k > 2) must also be impossible.