Alexander Harris - 260688155
Abbas Yadollahi - 260680343

ECSE 420
November 9, 2018

Assignment 2 Report

Q1

1.1

See FilterLock class in Fig. 1 for the code implementation.

1.2

No because it is impossible for one thread to overtake another within a loop iteration, as we would have to change victim and this would kick the current thread out of the while loop. However, it is possible for one thread to overtake another between loop iterations due to external factors such as CPU scheduling, which may prioritize a certain thread over another.

1.3

See BakeryLock class in Fig. 2 for the code implementation.

1.4

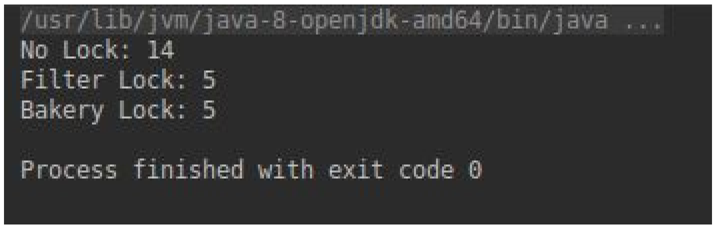
No, since Lamport's Bakery is a FCFS protocol and threads should be executed in sequential order which would not allow for overtaking.

1.5

We can have a counter that only counts up to a certain value before stopping, say 5. For each thread's run() method, we acquire the lock, check if the current counter is less than 5 and increment counter if true, then release the lock. We can then create a large number of threads, say 50, in order to guarantee the counter will become larger than 5 if there is no mutual exclusion.

1.6

See TestLock class in Fig. 3 for the code implementation of above. We can see from the output below that both the Filter and Bakery locks appear to provide mutual exclusion, as neither counters were incremented above 5, while the Runnable without locks is above 5.



```
/usr/lib/jvm/java-8-openjdk-amd64/bin/java ...
No Lock: 14
Filter Lock: 5
Bakery Lock: 5

Process finished with exit code 0
```

It is still possible that this test will show the appropriate value for the counter in the no lock case, but the majority of the time we should obtain a negative result. Therefore, we could run the same test over multiple iterations to ensure we obtain the expected behaviour.

Q2

For LockOne, changing the shared atomic registers to regular registers should not affect it's mutual exclusion property. Since the flag variables don't require finite read times, they are agnostic to value it takes, whether it be before or after the overlapping write. Although, it can cause the deadlock situation to happen more randomly during overlapping read/writes as the flag may take on either of the old or new value.

For LockTwo, changing the shared atomic registers to regular registers should not affect it's mutual exclusion property either. The victim variable is shared by both threads, where they set it to their own id to allow the other thread to enter its critical section. Changing the register type would only introduce a delay in very specific situations: in the case where the value of victim is being read in one thread's while loop at the same time as the other thread is setting victim to its own id. This overlap will either cause the first thread to execute its critical section (as intended) or loop once more before continuing to its critical section.

Q3

3.1
While it does provide mutual exclusion for cases where we have two threads, for instances where we have 3 or more threads mutual exclusion is not guaranteed, as the algorithm assumes that thread 1 will have finished its critical section before a third thread arrives.

Namely, it is possible than in the case where we have 3 threads, thread 1 is in its critical section and thread 2 is waiting, thread 3 could arrive and kick thread 2 into its critical section, which would result in thread 1 and thread 2 being in their critical sections simultaneously, which violates mutual exclusion.

3.2
This will deadlock in the case where we only have one thread, as it will wait forever in the loop for another thread to arrive and release it from the loop. Therefore, the protocol is not deadlock-free.

3.3
This protocol can not be starvation free since it is not deadlock free, as deadlock implies starvation. We can again look at the situation proposed in (3.2), where we have one thread waiting forever for another one to arrive and release it from its loop. This is an example of starvation.

Q4

a)
This history is sequentially consistent because the sequence is: [A] r.write(0), [B] r.write(1), [A] r.read(1), [C] r.read(2), [A] r.write(2), [C] r.write(3), [B] r.read(2). However, it is not linearizable due to the fact that [A] r.read(1) and [B] r.write(1) overlap, as well as [A] r.write(2) and [C] r.read(2), which would necessitate some kind of interrupt.

b)
This history is not sequentially consistent because of the sequence: [B] r.write(1), [A] r.read(1), [C] r.write(2), [C] r.read(1), [B] r.read(2). This is because of the order of execution between B and C. Additionally, it is not linearizable as [A] r.read(1) overlaps with [B] r.write(1) and [C] r.read(1).

Q5

5.1
For a division by zero in the reader to occur, we would need to call writer in one thread before calling reader in another. The volatile variable v would be set to true for all threads, however x would be 42 in the writer thread, but still equal to 0 in the reader thread. Then when we call reader in the reader thread this will result in a division by 0.

5.2
If both are volatile, then division by 0 would be impossible as both variables would be the same across all threads. Therefore in the situation discussed previously, x would be set to 42 in the reader thread and so we would not have a division by 0. If none are volatile, then variables are only thread-local, so v would still be false in the reader thread and we would not enter the block where the division by 0 could happen.

Q6

6.1
True, seeing as in the new for loop, the array r_bit will be setting the value at index x to true, but everything beyond it to false, while leaving everything before it as is. As a result, when we attempt to read, we return the index of the first true element, which, thanks to the new for loop, can be either the old or new value. This falls in line with the functionality of a regular M-valued MRSW register.

6.2
False, seeing as in the new for loop, the array r_bit will be setting the value at index x to true, but everything beyond it to false, while leaving everything before it as is. During overlaps with reading and writing, unlike a regular register, a safe Boolean register needs to return the most recent value of a variable, never the old. As a result, the new for loop breaks this functionality of the safe Boolean MRSW register since it can return either the new or old value when searching for the index of the first true element.

Q7

If we supposed that binary consensus was possible for n > 2, then for any set of 2 threads in the set n > 2 it is also possible, therefore binary consensus is possible for 2 threads, which contradicts our initial statement. Therefore binary consensus must also be impossible for n threads.

Q8

If we assume that consensus over k > 2 values is possible, and binary consensus is impossible, then we could design a binary consensus protocol by mapping [0, k/2] to 0 and [k/2, k] to 1. However, this contradicts our initial statement that binary consensus is impossible. Therefore, consensus over k > 2 values is not possible if binary consensus is not possible.

APPENDIX

Fig. 1 - Filter Lock

```
package ca.mcgill.ecse420.a2;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;

public class FilterLock implements Lock {

   private AtomicInteger[] level;
   private AtomicInteger[] victim;
   private int n;

   public FilterLock(int n) {
       level = new AtomicInteger[n];
       victim = new AtomicInteger[n];
       this.n = n;
       for (int i = 0; i < n; i++) {
           level[i] = new AtomicInteger();
           victim[i] = new AtomicInteger();
       }
   }

   @Override
   public void lock() {
       int id = (int) Thread.currentThread().getId() % n;
       for (int i = 1; i < n; i++) {
           level[id].set(i);
           victim[i].set(id);
           for (int j = 0; j < n; j++) {
               while ((j != id) && (level[j].get() >= i && victim[i].get() ==
id)) {
                   // wait here
               }
           }
       }
   }


   @Override
```

```
    public void unlock() {
        int id = (int) Thread.currentThread().getId() % n;
        level[id].set(0);
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {

    }

    @Override
    public boolean tryLock() {
        return false;
    }

    @Override
    public boolean tryLock(long l, TimeUnit timeUnit) throws InterruptedException
{
        return false;
    }

    @Override
    public Condition newCondition() {
        return null;
    }
}
```

Fig. 2 - Bakery Lock

```
package ca.mcgill.ecse420.a2;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;

public class BakeryLock implements Lock {

   private AtomicBoolean[] flag;
   private AtomicInteger[] label;

   private int n;

   public BakeryLock(int n) {
       flag = new AtomicBoolean[n];
       label = new AtomicInteger[n];
       this.n = n;
       for (int i = 0; i < n; i++) {
           flag[i] = new AtomicBoolean();
           label[i] = new AtomicInteger();
       }
   }

   @Override
   public void lock() {
       int id = (int) Thread.currentThread().getId() % n;
       flag[id].set(true);
       label[id].set(findLargestValue(label) + 1);
       for (int i = 0; i < n; i++) {
           while ((i != id) && flag[i].get() && ((label[i].get() <
label[id].get()) || ((label[i].get() == label[id].get()) && i < id))) {
               // wait here
           }
       }
   }

   @Override
   public void unlock() {
       int id =  (int) Thread.currentThread().getId() % n;
       flag[id].set(false);
```

```java
        label[id].set(0);
    }

    private int findLargestValue(AtomicInteger[] array) {
        int max = Integer.MIN_VALUE;
        for (AtomicInteger a : array) {
            int val = a.get();
            if (val > max) {
                max = val;
            }
        }
        return max;
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {

    }

    @Override
    public boolean tryLock() {
        return false;
    }

    @Override
    public boolean tryLock(long l, TimeUnit timeUnit) throws InterruptedException
{
        return false;
    }

    @Override
    public Condition newCondition() {
        return null;
    }
}
```

Fig. 3 - TestLock

```java
package ca.mcgill.ecse420.a2;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class TestLock {

   private static final int NUMBER_THREADS = 50;
   private static int cnt = 0;

   private static FilterLock filterLock = new FilterLock(NUMBER_THREADS);
   private static BakeryLock bakeryLock = new BakeryLock(NUMBER_THREADS);

   public static void main(String[] args) {

       // No Lock Test
       ExecutorService executor = Executors.newFixedThreadPool(NUMBER_THREADS);

       for (int i = 0; i < NUMBER_THREADS; i++) {
           executor.execute(new NoLockRunnable());
       }

       executor.shutdown();
       try {
           executor.awaitTermination(  10, TimeUnit.SECONDS);
       } catch (InterruptedException e) {
           e.printStackTrace();
       }

       System.out.println("No Lock: " + cnt);

       // Filter Lock Test
       cnt = 0;
       executor = Executors.newFixedThreadPool(NUMBER_THREADS);

       for (int i = 0; i < NUMBER_THREADS; i++) {
           executor.execute(new FilterLockRunnable());
       }

       executor.shutdown();
       try {
            executor.awaitTermination(  10, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Filter Lock: " + cnt);

        // Bakery Lock Test
        cnt = 0;
        executor = Executors.newFixedThreadPool(NUMBER_THREADS);
        for (int i = 0; i < NUMBER_THREADS; i++) {
            executor.execute(new BakeryLockRunnable());
        }

        executor.shutdown();
        try {
            executor.awaitTermination(  10, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Bakery Lock: " + cnt);
    }

    private static class NoLockRunnable implements Runnable {

        private NoLockRunnable() {

        }

        @Override
        public void run() {
            try {
                if (cnt < 5) {
                    Thread.sleep(10);
                    cnt++;
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private static class FilterLockRunnable implements Runnable {
        private FilterLockRunnable() {

        }

        @Override
        public void run() {
            try {
                filterLock.lock();
                if (cnt < 5) {
                    Thread.sleep(10);
                    cnt++;
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                filterLock.unlock();
            }
        }
    }

    private static class BakeryLockRunnable implements Runnable {

        private BakeryLockRunnable() {

        }

        @Override
        public void run() {
            try {
                bakeryLock.lock();
                if (cnt < 5) {
                    Thread.sleep(10);
                    cnt++;
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                bakeryLock.unlock();
            }
        }
    }
}
```