Assignment 2

Spiros Mavroidakos 26068931 Jastaj Virdee 260689027

November 9, 2018

1 Q.1

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 section 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 section A.3

A Question 1 Code

A.1 Filter algorithm

```java
import java.util.concurrent.locks.Lock;
import java.lang.*;

class Filter implements Lock{
    private int level[];
    private int victim[];

    /**
     * Constructor for the filter class
     * @param n the amount of threads
     */
    public Filter (int n){
        level = new int[n];
        victim = new int[n];

        // Initialize the levels
        for (int i = 0; i < n; i++){
            level[i] = 0;
        }
    }

    /**
     * Method to lock the lock
     */
    public void lock(){
        int threadId = (int) Thread.currentThread().getId() % level.length;
        for (int L = 1; L < level.length; L++){
            level[threadId] = L;
            victim[L] = threadId;

            boolean spinwait = true;
            while (spinwait) {
                spinwait = false;

                for (int k = 1; k < level.length; k++) {
                    if (k != threadId && level[k] >= L && victim[L] == threadId) {
                        spinwait = true;
                        break;
                    }
                }
            }
        }
    }

    /**
     * Method to release the lock
     */
    public void unlock(){
        int threadId = (int) Thread.currentThread().getId() % level.length;
        level[threadId] = 0;
    }

    /*
    * Abstract methods that need to be defined. They do nothing basically since they
    * is no the focus of the filter lock
    */
    public java.util.concurrent.locks.Condition newCondition() {
        return null;
    }

    public boolean tryLock(long timeout, java.util.concurrent.TimeUnit unit) throws
    java.lang.InterruptedException {
        return false;
    }

    public boolean tryLock() {
        return false;
    }

    public void lockInterruptibly() throws java.lang.InterruptedException {
    }
}
```

A.2 Bakery Algorithm

```
import java.util.concurrent.locks.Lock;

class Bakery implements Lock{
    boolean[] flag;
    int[] label;

    /**
     * Number of threads. Used for interation in the lock
     * method.
     */
    private int n;

    public Bakery(int n){
        flag = new boolean[n];
        label = new int[n];
        this.n = n;
        for(int i = 0; i < n; i++){
            flag[i] = false;
            label[i] = 0;
        }
    }

    public void lock(){
        int me = (int) Thread.currentThread().getId() % n;
        flag[me] = true;
        label[me] = findMax(label) + 1;

        boolean spinwait = true;
        while (spinwait) {
            spinwait = false;

            for (int k = 0; k < n; k++) {
                if ((k != me) && flag[k]
               && ((label[k] < label[me]) || ((label[k] == label[me]) && k < me))){
                    spinwait = true;
                    break;
                }
            }
        }
    }

    public void unlock(){
        int me = (int) Thread.currentThread().getId() % n;
        flag[me] = false;
    }

    private int findMax(int[] array) {
        int max = Integer.MIN_VALUE;
        for (int i = 0; i < array.length; i++)
        {
            if(array[i] > max){
                max = array[i];
            }
        }
        return max;
    }

```

```java
    /*
    * Abstract methods that need to be defined. They do nothing basically since they
    * is no the focus of the filter lock
    */
    public java.util.concurrent.locks.Condition newCondition() {
        return null;
    }

    public boolean tryLock(long timeout, java.util.concurrent.TimeUnit unit) throws
    java.lang.InterruptedException {
        return false;
    }

    public boolean tryLock() {
        return false;
    }

    public void lockInterruptibly() throws java.lang.InterruptedException {
    }
}
```

A.3 Test Code for Bakery Algorithm

```java
import java.util.concurrent.ExecutionException;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class BakeryTest {

    public static Bakery lock1 = new Bakery(5);

    public static void main(String arg[]) {
        BakeryClient[] clients = new BakeryClient[5];

        for(int i = 0; i < clients.length; i++)
        {
            clients[i] = new BakeryClient();
            clients[i].start();
        }
    }

    private static class BakeryClient extends Thread {
        public void run() {
            System.out.println("Thread with id = " + this.getId() + " acquiring Lock 1"
    );
            lock1.lock();
            System.out.println("Thread with id = " + this.getId() + " Holding Lock 1");

            try
            {
                sleep(5000);
            }
            catch(Exception e)
            {
                System.out.println("Error: " + e.getMessage());
                lock1.unlock();
                return;
            }

            System.out.println("Thread with id = " + this.getId() + " Releasing lock");
            lock1.unlock();
        }
    }
}
```