ECSE 420 Assignment 2 Report
Group 19

Stuart Mashaal
260639962

Oliver Tse Sakkwun
260604362

Due: November 7, 2018

Question 1

1.2

Yes, the Filter Lock allows threads to overtake others an arbitrary number of times. Under the assumption that the thread scheduler is non-deterministic, consider the following scenario:

Thread t_1 enters the lock and stops just before the `while` loop. Thread t_2 enters and gets stuck in the `while` loop because of t_1. t_3 then enters `lock()` and gets stuck at the while loop too, but in so doing allows t_1 or t_2 to pass – whoever is scheduled first! Say t_2 is scheduled first and goes on to pass all levels and acquire the Filter Lock. t_4 can now come and let t_3 acquire the lock, then t_5 can let t_4, and so on. And when t_n is the one waiting, t_2 can come back in and free it and the cycle can continue. In this way, an arbitrary number of threads can overtake t_1 in the Filter Lock.

1.4

No, the Bakery Lock does not allow other threads to overtake others an arbitrary number of times. The Bakery Lock is a *'First-Come, First-Serve'* (a.k.a. FIFO) lock, so by definition it does not.

The Bakery Lock achieves FIFO behaviour because if, for any two threads A and B, A completes the *doorway* and acquires a `label` before B, then A's label is less than B's. Thereofre, B must wait for A to set its `flag` to false.

So we see that the Bakery Lock is fundamentally more fair than the Filter Lock. Despite the fact that the Filter Lock **is** starvation-free, a thread waiting at its `while` loop can be overtaken an arbitrary number of times before it is scheduled to run again. Conversely, if a thread completes the *doorway* in the Bakery Lock, no threads can overtake it no matter how much time passes before it is scheduled to run again.

1.5

If a lock provides mutual exclusion, then only one thread at a time can acquire the lock. Other threads that attempt to acquire the lock while the first thread has already done so will wait at the call to `lock()` until the first thread calls `unlock()` .

To test that a lock behaves in this way, we must allow one thread to acquire a lock and then have another thread attempt to acquire it before the first has unlocked it. Both threads will increment (therefore read and then write) the value of a shared atomic register while they have the lock. After both threads have released the lock, the final value of the shared register must be 2 greater than its initial value, otherwise the lock does not provide mutual exclusion.

To guarantee that the second thread attempts to acquire the lock while the first thread still has it, we can have the first thread sleep for a relatively long time after acquiring the lock and reading the register but before writing the register. This will *almost certainly* force the thread scheduler to run the second thread. If the lock does not provide mutual exclusion, then while the first thread is sleeping the second thread will both read and write to the shared register. Finally the first thread will awaken and write to the register, makings the second's write a *lost update.*

Question 2

2.1 - LockOne

No. If the shared `flag` atomic registers are replaced by regular registers, `LockOne` does not still provide mutual exclusion. **Proof:**

Thread 1

```java
public void lock() {
    int i = ThreadID.get();
    int j = i - 1;
    flag[i] = true // only local
//
//
//
//
//
//
    while (flag[j]) {} // false
} // enter critical section
```

Thread 2

```java
//
//
//
//
public void lock() {
    int i = ThreadID.get();
    int j = i - 1;
    flag[i] = true // only local
    while (flag[j]) {} // false
} // enter critical section
//
//
```

The problem is that if we don't use atomic registers, then the writes to `flag` are not necessarily shared right away. This means that both threads can write to `flag` without the other being able to detect it. Then, both can enter the critical section.

2.2 - LockTwo

No, if shared variable `victim` uses a regular register instead of an atomic one `LockTwo` does not still provide mutual exclusion, for similar reasons to those of 2.1. **Proof:**

Thread 1

```java
public void lock() {
    int i = ThreadID.get();
    victim = i //only local
//
//
//
//
//
    while (victim == i){} //see t2 write
} // enter critical section
```

Thread 2

```java
//
//
//
public void lock() {
    int i = ThreadID.get();
    victim = i //only local
    while (victim == i){} //see t1 write
} //enter critical section
//
//
```

Since the `victim` register is no longer atomic, writes to it by either thread are not only *not shared immediately* but also are *not sequentially consistent*. The Java Concurrency Model does not guarantee sequential consistency without the use of synchronization primitives (such as the `volatile` keyword, which makes a register atomic).

Question 3

3.1 - Mutual Exclusion

Yes, the `LockThree` protocol provides mutual exclusion.

Suppose both threads A and B are in the critical section at the same time. This means that

$$write_A(\text{turn} = A) \to write_A(\text{busy} = \text{true}) \to read_A(\text{turn} = B) \to \text{CS}_A$$

and that

$$write_B(\text{turn} = B) \to write_B(\text{busy} = \text{true}) \to read_B(\text{turn} = A) \to \text{CS}_B$$

However, in order for the value of `turn` to be read as A or B, it must be written to as such first. This means

$$write_B(\text{turn} = B) \to read_A(\text{turn} = B)$$

and

$$write_A(\text{turn} = A) \to read_B(\text{turn} = A)$$

Together, this implies that, without loss of generality

$$write_A(\text{turn} = A) \to write_B(\text{turn} = B) \to read_B(\text{turn} = A) \to read_A(\text{turn} = B)$$

or

$$write_A(\text{turn} = A) \to write_B(\text{turn} = B) \to read_A(\text{turn} = B) \to read_B(\text{turn} = A)$$

which is a contradiction in both cases.

3.2 - Deadlock Freedom

If thread t is the only thread that will ever acquire this lock, or if it's the last thread to try and acquire the lock and no other thread is currently holding the lock, then `LockThree` deadlocks. Our lone thread t will set `turn = me` , then `busy = true` , and then will wait forever at `while (turn == me && busy);` .

3.3 - Starvation Freedom

A lock cannot suffer from the deadlock problem (which `LockThree` does) and *also* be starvation-free, so **no,** the protocol is not starvation-free.

Question 4

Some stuff.