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 and release it. t_4 can now do the same 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 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 satisfies the requirements of 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. Therefore, 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 the 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*. The *lost update* can be detected by the final value of the register **not** being 2 greater thant its initial value.

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

$$\textit{write}_A(\text{turn} = A) \rightarrow \textit{write}_A(\text{busy} = \text{true}) \rightarrow \textit{read}_A(\text{turn} = B) \rightarrow \text{CS}_A$$

and that

$$\textit{write}_B(\text{turn} = B) \rightarrow \textit{write}_B(\text{busy} = \text{true}) \rightarrow \textit{read}_B(\text{turn} = A) \rightarrow \text{CS}_B$$

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

$$\textit{write}_B(\text{turn} = B) \rightarrow \textit{read}_A(\text{turn} = B)$$

and

$$\textit{write}_A(\text{turn} = A) \rightarrow \textit{read}_B(\text{turn} = A)$$

Together, this implies that, without loss of generality

$$\textit{write}_A(\text{turn} = A) \rightarrow \textit{write}_B(\text{turn} = B) \rightarrow \textit{read}_B(\text{turn} = A) \rightarrow \textit{read}_A(\text{turn} = B)$$

or

$$\textit{write}_A(\text{turn} = A) \rightarrow \textit{write}_B(\text{turn} = B) \rightarrow \textit{read}_A(\text{turn} = B) \rightarrow \textit{read}_B(\text{turn} = A)$$

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

First, note the following definitions from *Herlihy and Shavit*:

1. **Sequential Consistency:** a concurrent execution is sequentially consistent if there exists a *global ordering of method calls* such that the method calls
   (a) are consistent with the intra-thread program order
   (b) meet the objects' sequential specifictaion
2. **Linearizability:** Sequential consistency, but with the added requirement that "each method call invocation should appear to take effect instantaneously, at a 'linearization point', at some moment between its invocation and response"

4.1 - History A

4.1.1 - Sequential Consistency

Yes, History A is sequentially consistent. The following global ordering of method calls would be

1. consistent with intra-thread program order, and
2. meet the objects' sequential specification

```
Thread A                 Thread B                 Thread C

r.write(0)
                         r.write(1)
r.read(1)
r.write(2)
                                                  r.read(2)
                         r.read(2)
                                                  r.write(3)
```

4.1.2 - Linearizability

No, History A is not linearizable. To prove it, let's name some of the method calls in History A as follows:

Method Call	Name
thread A: `r.write(2)`	W
thread C: `r.read(2)`	X
thread B: `r.read(2)`	Y
thread C: `r.write(3)`	Z

Let $W \to X$ denote 'W must happen before X'. Then History A gives us that according to

Sequential Consistency, $W \to X,\ W \to Y,\ X \to Z,\ Y \to Z$

Linearizability, $Z \to Y$

If we make the variables W, X, Y, Z nodes in a graph and make the '$\to$' relationships into edges, the graph will **not be acyclic**. Therefore, History A is **not linearizable**.

4.2 - History B

4.2.1 - Sequential Consistency

No, History B is not sequentially consistent. To prove it, let's name some method calls from History B as follows:

Method Call	Name
thread B: `r.write(1)`	W
thread B: `r.read(2)`	X
thread C: `r.write(2)`	Y
thread C: `r.read(1)`	Z

Then History B gives us that according to

Sequential Consistency, $W \rightarrow X,\ W \rightarrow Z,\ Y \rightarrow X,\ Y \rightarrow Z$. However, recall that sequential consistency also demands that the 'sequential specifications of the objects are met'. Therefore we also have that either $Y \rightarrow W,\ X \rightarrow W$ or that $W \rightarrow Y,\ Z \rightarrow Y$.

If we create a graph from these 'happens-before' relationships, the graph will **not be acyclic**, and therefore History B is **not sequentially consistent**.

4.2.2 - Linearizability

No, History B is not linearizable. All linearizable executions must be sequentially consistent, and History B is not sequentially consistent so it is not linearizable.

Question 5

5.1

Yes, the `reader()` method may divide by zero.

Thread 1

```java
int x = 0;                    // shared
volatile boolean v = false; // shared
public void writer() {
    x = 42;   // only local
    v = true; // shared
}




```

Thread 2

```java
int x = 0;                    // shared
volatile boolean v = false; // shared




public void reader() {
    if (v == true) {
        int y = 100/x; // division by zero!
    }
}
```

We see that since variable `x` is not `volatile`, the assignment by thread 1 of `x = 42` is not seen by thread 2 despite that thread 1's assignment `v = true` is.

5.2

If both `v` and `x` are volatile, then division by zero is not possible. Any time that `v = true`, it's required that `x = 42` is done first, so inside the `if (v == true)`, dividing by `x` is dividing by 42 and not dividing by zero.

If both variables are **not** volatile, then all bets are off. As we know, sequential consistency is not guaranteed by the JVM when the variables in question are not volatile. Therefore, the exact same trace as in 5.1 is possible and thus so is division by zero.

Question 6

6.1

The answer is true. In the first case, if reads dont overlap with writes, then the most recent value will be returned because all registers have been written to already.

In the case where read calls overlap with write calls, by the definition of a safe MRSW register, the value read must be any value within the domain of the values. Because the register array is composed of safe SRSW registers, this condition is satisfied.

6.2

The answer is true. As for the previous question, for nonoverlapping calls, read calls return the most recently written value.

If read calls overlap with write calls, then, by definition of a regular MRSW register, the value returned must either be the old value or the new value. Because the array contains regular registers, that condition is satisfied.

Question 7

Suppose we had a protocol for binary consensus for n threads. We could reduce it to a protocol for two threads by simply having two threads take steps and the remaining n threads hold. We then have a protocol for two threads. Because that is impossible, we have a contradiction. Therefore, binary consensus for n threads using atomic registers is impossible.

Question 8

Suppose consensus over k values is possible following some protocol. We could then reduce the consensus protocol to a binary consensus protocol by mapping one value to 0 and all other values to 1. This implies that we have binary consensus. But, since binary consensus is impossible, we have a contradiction, therefore consensus over k values is impossible.