ECSE 420 Parallel Computing **Dennis Liu #260581270**
Assignment #2 **Yijie Zhou #260622641**

Question 1.1

Filter lock is implemented as $filterLock.java$

Question 1.2

Between levels it is possible for other threads to overtake threads. E.g. thread A entered level 1 and about to enter the next iteration of for loop with i = 2, and A sleeps for some time maybe because of scheduling etc. every other threads which completed level 1 may overtake thread A.

Question 1.3

Bakery lock is implemented as $Bakery.java$

Question 1.4

No. it is a FIFO algorithm lock.

Question 1.5

6 threads try to transfer random amount of money between different accounts with total balance of 60000.

Question 1.6

$Bakery_test.java$ and $filter_test.java$ provide implementations for proposed tests. After running, total balance stay the same and no deadlock. Both locks provide mutual exclusion.

Question 2

Peterson's two-thread mutual exclusion algorithm does not work if the shared atomic flag registers are replaced by regular registers. This is because atomic registers ensure that when a register is being write to and read at the same time, after a new value is read, the old value will no longer be read by subsequent processes.

Using a regular register which does not have this feature can introduce synchronization issues. For instance, if both processes are writing to the turn flag at the same time, the condition for busy waiting will read either read the turn flag as 0 or 1, hence potentially causing both processes to enter their critical sections at the same time.

Question 3.1

From the Code:

$$\begin{aligned}Write_A(turn = A)\\ \rightarrow Read_A(busy = false)\\ \rightarrow Write_A(busy = true)\\ \rightarrow Read_A(turn! = A)\\ \rightarrow CS_A\end{aligned} \tag{1}$$

$$\begin{aligned}Write_B(turn = B)\\ \rightarrow Read_B(busy = false)\\ \rightarrow Write_B(busy = true)\\ \rightarrow Read_B(turn! = B)\\ \rightarrow CS_B\end{aligned} \tag{2}$$

Assume both threads enter the critical section at the same time:

$$\begin{aligned}Write_A(turn = A)\\ \rightarrow Write_B(busy = false)\\ \rightarrow Write_B(turn = B)\end{aligned} \tag{3}$$

Combining:

$$\begin{aligned}
&Write_A(turn = A) \\
&\rightarrow Write_B(busy = false) \\
&\rightarrow Write_B(turn = B) \\
\\
&AND \\
\\
&Write_B(turn = B) \\
&\rightarrow Write)_B(busy = false)
\end{aligned} \tag{4}$$

Writing turn = B and busy = false cannot precede each other at the same time, thus there is a contradiction when both threads are in their critical sections at the same time. Therefore the protocol does satisfy mutual exclusion.

Question 3.2

Threads are blocked at $while(busy = true)$ and $while(turn! = ThreadID)$. $turn$ can only be equal to the ThreadID of A or B, thus both threads cannot be blocked by this loop at the same time. However, both threads share the same register *busy*, thus if busy is initialized to be true, both threads will be blocked and a deadlock would occur. Therefore, this protocol is not deadlock-free.

Question 3.3

Thread A is blocked only if thread B repeatedly re-enters so that either $busy = true$, or $turn! = A$. Each time B enters, it sets $turn = B$.

Thus if:

$$Write_A(turn = A) \rightarrow Write_B(turn = B) \rightarrow Read_A(turn! = A) \tag{5}$$

then thread A would be starved and never return. Therefore the protocol is not starvation-free

Question 4

History (a) is linearizable in the following order:

B r.write(1)
B r:void
A r.read()
A r:1
C r.write(2)
C r:void
B r.read()
B r:2

Since the history is linearizable, it is also sequentially consistent.

Thread A's read of 1 indicates that thread B's write of 1 precedes thread A's read. Also, thread C's write of 2 must also happen after thread A's read. Lastly, since thread C's write of 2 precedes thread B's read of 1, the value of register r at the time of thread B's read should be 2 and not 1. Therefore, history (b) is not linearizable.

Although not linearizable, history (b) is sequentially consistent and can be rearranged in the following order:

C r.write(2)
C r:void
B r:write(1)
B r:void
B r:read()
B r:1
A r:read()
A r:1

Question 5

The *reader*() method will divide by zero in the case that the non-volatile variable x is updated after the volatile variable v is updated. Since writing to volatile variable v has similar effects to releasing a lock, when the *writer*() updates v, the *reader* will stop waiting and perform the division. Thus if the variable x is updated to 0 after v is updated to true, the *reader*() will divide by 0.

Question 6.1

True

MRSW register implementation is safe if: (1) read() (that doesn't overlap write() call) returns the value written by the most recent write() call. (2) when overlap, read() call may return any value within the register's allowed range of values (e.g. 0 to M-1 if M-value register). If read() does not overlap write() call, then all registers in the array have been written and will return the most recent value since these are safe SRSW registers.

Then we consider when read() overlaps write(), if any value within the domain is returned. Since that is already guaranteed by the properties of the safe SRSW registers in the array, requirement is satisfied. Thus the construction yields a safe M-valued MRSW register.

Question 6.2

True

Very similar to previous question. For non-overlapping method calls, each table does hold the most recent value and will be returned by read() call. For overlapping method calls, the reader may return either the new or old value since the registers are regular. Thus the construction yields a regular Boolean MRSW register.

Question 7

If some wait free, n thread consensus protocol work well, then it must work well when two threads take steps, so that two threads can simply run the n-thread protocol.

Question 8

If we had a k value, n threads consensus protocol, we could solve binary 2 thread consensus simply by having only two threads participate, and restricting them to binary values. Proof by contradiction.