Assignment 2

March 22nd, 2021

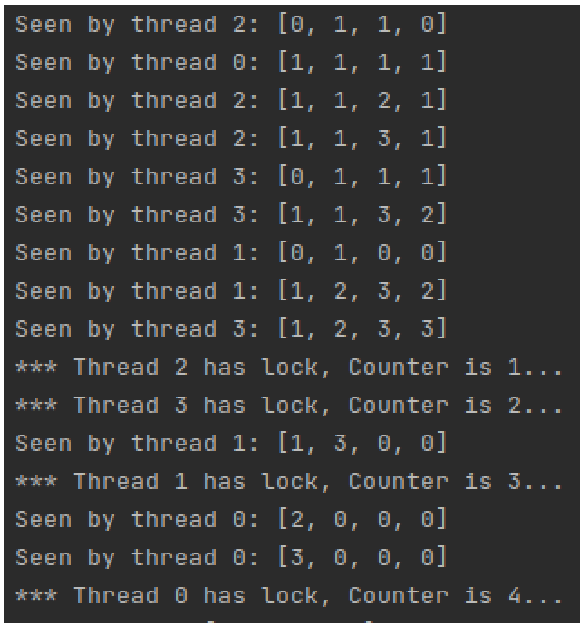
ECSE 420
Parallel Computing
Winter 2021
Group 26

Rintaro Nomura - 260781007
Edward Latulipe-Kang - 260746475

<u>Question 1</u>

1.1)
Please look at code for the implementation.

1.2)
The Filter Lock algorithm does, in fact, allow some arbitrary number of threads to overtake other threads. We can see this to be the case as per Figure 1.1.



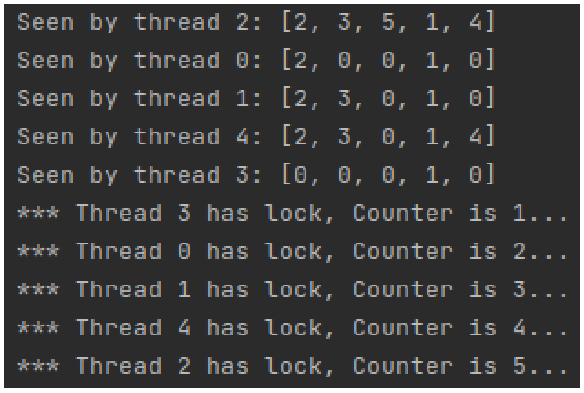
```
Seen by thread 2: [0, 1, 1, 0]
Seen by thread 0: [1, 1, 1, 1]
Seen by thread 2: [1, 1, 2, 1]
Seen by thread 2: [1, 1, 3, 1]
Seen by thread 3: [0, 1, 1, 1]
Seen by thread 3: [1, 1, 3, 2]
Seen by thread 1: [0, 1, 0, 0]
Seen by thread 1: [1, 2, 3, 2]
Seen by thread 3: [1, 2, 3, 3]
*** Thread 2 has lock, Counter is 1...
*** Thread 3 has lock, Counter is 2...
Seen by thread 1: [1, 3, 0, 0]
*** Thread 1 has lock, Counter is 3...
Seen by thread 0: [2, 0, 0, 0]
Seen by thread 0: [3, 0, 0, 0]
*** Thread 0 has lock, Counter is 4...
```

Figure 1.1. Filter Lock Process

At one point in time, all threads are interested in level 1. However, Thread 0 is the last thread to get the lock. This indicates that Thread 0 has allowed the other 3 threads to proceed ahead of it. This is normal, because the filter algorithm, despite being starvation free, does not guarantee the fair treatment of all threads.

1.3)
Please look at code for the implementation.

1.4)
The Bakery Lock algorithm does not allow some arbitrary number of threads to overtake other threads. We can see this to be the case as per Figure 1.2.



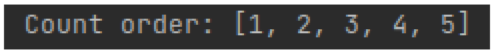
```
Seen by thread 2: [2, 3, 5, 1, 4]
Seen by thread 0: [2, 0, 0, 1, 0]
Seen by thread 1: [2, 3, 0, 1, 0]
Seen by thread 4: [2, 3, 0, 1, 4]
Seen by thread 3: [0, 0, 0, 1, 0]
*** Thread 3 has lock, Counter is 1...
*** Thread 0 has lock, Counter is 2...
*** Thread 1 has lock, Counter is 3...
*** Thread 4 has lock, Counter is 4...
*** Thread 2 has lock, Counter is 5...
```

Figure 1.2. Bakery Lock Process

This is because the bakery algorithm provides a first-come-first-served (FIFO) implementation for the n-threads. This is the programming equivalent of going to the hospital, taking a number, and waiting your turn. Each thread will get a turn in the order in which they have arrived. Ergo, The Bakery algorithm treats all threads fairly.

1.5)
A possible solution is to create a counter array that keeps track of all the threads. In essence, each thread will increment the count. Therefore, we should be able to see the count for each thread increment as a thread accesses a lock. The array will store the count in order to show that our counter is, in fact, useful. Should the values be in any other arrangement, then we know that our program is not mutually exclusive and that the test fails.

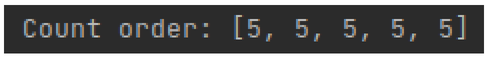
1.6)
This is the count order for a working counter. Of course, this will establish a control result that we can compare our tests to. For the Bakery Lock algorithm, as per Figure 1.3, we do in fact count in order when the lock is used.



```
Count order: [1, 2, 3, 4, 5]
```

Figure 1.3. Successful Test

However, if we chose not to use the lock, we will get the same count value applied to all threads. Observe Figure 1.4.



```
Count order: [5, 5, 5, 5, 5]
```

Figure 1.4. Failed Test

As we can see, with the lock, our counter does not count up as intended. Instead, we only get the final count since all threads are operating concurrently. However, we only wanted the count to increment with every thread, and for that count to be relayed back to us. The same can be said for the Filter Lock implementation, as it uses the same test. This is proof that our implementation of the algorithm works.

Question 2

2.1)
By definition, an atomic register functions so as to ensure that reads and writes occur at a single point, and any reads or writes beyond that point will act upon the new value. However, regular registers are allowed to "flicker" back and forth throughout the duration of the instruction. This means that any reads that occur during this interval may register an earlier result or the new result. This ambiguity lends itself poorly to LockOne and LockTwo since the values of the flag and the victim may be inaccurate. Consider the case where LockOne reads flag = false for both threads. This can occur if the thread reads overlap slightly, and the both happen to use the previous value, before flag = true is set. This would cause both threads to access the critical section, which is dangerous. We must, then, conclude that mutual exclusion is not always guaranteed with regular registers.

Question 3

3.1)
We will start by assuming that both thread A and thread B are in the critical section at the same time. Notice that for this to be true, both threads must leave a while loop with the following parameters:

while (turn == me || busy)

In other words, the following sequence of events must have occurred for our thread to exit the while loop:

writeC(busy = !busy) → unlock()

writeA(turn = me) → readA
readA(turn != me && !busy) → CSAi

writeB(turn = me) → readB
readB(turn != me && !busy) → CSBj

We must assume that a thread has reached the unlock method. This is necessary, because otherwise we would not be able to set the busy state to be free. Furthermore, we can disregard the busy component of the while loop for analysis. We will use an arbitrary thread C for this purpose. When observing the write states of both thread A and B, we notice that 'turn' is set to be equivalent to the thread ID, or 'me'. However, upon the read state 'turn' and 'me' must not be equivalent for a thread to access the critical section. In fact, 'turn' and 'me' will always be equivalent, and the conditional of the while loop will always be true. This in turn, contradictions the assumption that both threads can access the critical section. Therefore, this protocol does satisfy mutual exclusion.

3.2)
This protocol is not deadlock-free unfortunately. Earlier we assumed that some thread C made use

of the unlock method. This assumption was necessary because, as stated earlier, the while loop is never broken. The fact that ‘turn’ is equivalent to ‘me’ will always be true, and so the lock method can never be exited.

3.2)
This protocol is not starvation-free. Similar to the argument made for the deadlock property of this protocol, the while loop can never be broken. The do-while loop forces the algorithm to set the lock to busy, without any way of setting it back to free except for invoking the unlock method. Unfortunately, this method, along with the critical section, will never be reached.

Question 4

4.1)
History A is not linearizable, however it is sequentially consistent. It is not linearizable because of the last two method invocations. We can linearize all other method intervals as shown in figure 4.1.

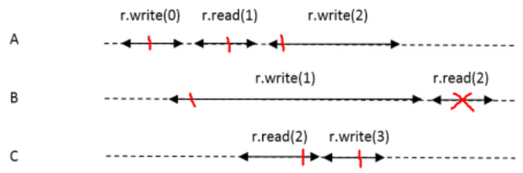


Figure 4.1. History 1 Linearization

However, once we arrive at r.write(3), there is no possible way to read 2 thereafter as these operations do not overlap. History A, however, can be made sequentially consistent, by shifting thread B such that it happens earlier. This way, r.read(2) of thread B can overlap with r.read(2) of thread C and r.write(2) of thread A. In this way, despite the real-time ordering being violated, r.write(2) will still be possible as r.write(3) will occur ‘’later”.

4.1)
History B, similar to A, is not linearizable. However, It is also not sequentially consistent. History B is not linearizable due to r.read(1) occurring in thread C, as per figure 4.2.

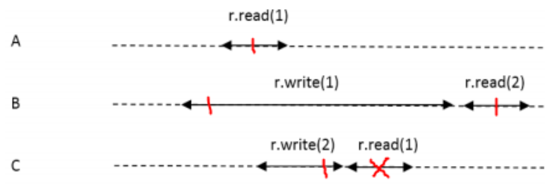


Figure 4.2. History 2 Linearization

This is because this operation is immediately preceded by another operation that performs a write for another value. If threads must be sequentially consistent with respect to themselves, then this is violated. There is no way to linearize r.write(1) without causing some type of sequential conflict. Furthermore, History B is not sequentially consistent. There is no way to shift or extend any thread in time, such that it is sequentially consistent. This is because thread B and thread C perform identical operations with opposite values. Ergo, the register value will switch before the next read/write operation.

Question 5

5.1)
The divide-by-zero could not happen because v has been declared volatile, and (v == true) is verified in the reader() in prior to reading the x. Since v has been declared volatile, writing to v is like releasing a lock, reading from v is like releasing a lock. In terms of concurrency, v could only become true after the write() method has completed, thus the reader() method only divides after the writer() method has executed at least once. Therefore, divide-by-zero would not happen.

5.2)

1. If both variables are defined as volatile then the divide-by-zero would not happen because if the variables are volatile they always read from the main memory, and act like there is a lock. In addition, they will be atomic, and therefore divide-by-zero would not happen.

2. If only v is volatile, divide-by-zero would not happen as stated in question 5.1.

3. If v is not volatile, then the divide-by-zero could be possible because the reader() and writer() method could be running at the same interval of method call.

Question 6

6.1)
True. For example, we may have the situation that reading and writing happen simultaneously. Since we have a safe MRSW register, the register array is composed of the safe SRSW registers, such that the value we obtain from the register would be always within the domain. For the regular read/write, since we write all registers, the read value would be always the most recent value.

6.2)
True. By definition, the value returned by the register would be either the original value or the new value, which does not matter because they are always written into the registers. It returns the most recent value if reading and writing are not overlapping. Thus, it always satisfies the condition.

Question 7

We may suppose that we have a protocol for binary consensus for n-threads. Then, we should be able to reduce it to a protocol for 2-thread by having 2 threads to proceed with its steps and the remaining n threads on hold, such that we have a protocol for 2 threads. However, this is not possible, therefore there is a contradiction. Consequently, it is impossible for two or more threads.

Question 8

Similarly, we may suppose that the consensus over k values is possible, while binary consensus is impossible. If so, we should be able to reduce the consensus protocol to a binary consensus protocol by mapping one value to zero, and other values to 1, such that we have binary consensus. Now, we have a contradiction because that implies that the binary consensus is possible. Therefore the binary consensus is impossible. Consequently, consensus over k values is also impossible.