CS5510: Multiprocessor Programming : Homework IV

Sergio Sainz

October 2019

1 Part I : Theory [25 points]

1.1 [10 points] A proposed algorithm for mutual-exclusion works as follows:

Let there be a multi-writer register roll, whose range is 1, 0, . . . , N 1 and the initial value is 1. To enter the critical section, a process i spins on roll, until the value is 1. And when it finds the value to be 1, it sets the value of roll to i, within one time unit of the read. After that, the process waits for more than one time unit, and then reads roll. If it still has the value i, then it enters the critical section. Otherwise, it returns to spinning on roll until the value is 1. When a process exits its critical section, its sets the value of roll to 1.

1. Does the algorithm guarantee mutual exclusion?
2. Is it livelock-free?
3. What are the drawbacks of this algorithm (in terms of performance and requirement)?

Answer:
I am going to assume code is something like this:

```
class RollLock{
    AtomicInteger roll = new AtomicInteger(-1);
    public void lock (){
        int i = Thread.getId();
        while(true){
            while(roll.get() != -1){
                true; // spinning
            }
            roll.set(i);
            TimeUnit.SECONDS.sleep(1);
            if(roll.get() == i){
                break ;
```

```
13              }
14          }
15      }
16      public void unlock(){
17          roll.set(-1);
18      }
19  }
```

1. Does the algorithm guarantee mutual exclusion?
   Yes, as long as the invariant that the read of roll equals to -1 and the write of roll with i is true, mutual exclusion should be guaranteed. But in real world this invariant is hard to achieve.
   If the invariant is true then we prove by contradiction. Assume both threads A, B are in critical section, then the following steps must have happened:

   $$read_A(roll = -1) \rightarrow write_A(roll = A) \rightarrow sleep_A(60sec) \rightarrow read_A(roll = A) \rightarrow CS_A$$
   $$read_B(roll = -1) \rightarrow write_B(roll = B) \rightarrow sleep_B(60sec) \rightarrow read_B(roll = B) \rightarrow CS_B$$

   Then, if we put them into a line:

   $$read_A(roll = -1) \rightarrow read_B(roll = -1) \rightarrow write_A(roll = A) \rightarrow write_B(roll = B) \rightarrow sleep_A(60sec) \rightarrow sleep_B(60sec) \rightarrow read_A(roll = A) \rightarrow read_B(roll = B) \rightarrow CS_A \rightarrow CS_B$$

   But we know is not possible for roll to store both A and B at the same time. Thus a contradiction. We know the reads (-1) must happen before the writes (i) because otherwise if write happens before read then the last thread to read (say B) will keep spinning in lines 6-8 (because it finds the roll being set to something other than -1. Then we also know the writes do not happen after the waiting because of the invariant that read and write happens within the period of time (same period of time than the sleep time). Thus writes happen before sleeping period finishes.

2. Is it livelock-free?
   Yes. Actually there is a lot of traffic in one line: when setting roll to -1. In these two cases the phenomenon called *cache-coherence* traffic will occur, which means that when all the spinning threads notice the thread is set to -1 (unlock method sets to -1) all of them will flood the memory bus with requests to invalidate working threads caches and replace with its value , i. This will occur for as many threads as there are spinning threads. After the last thread finishes setting the roll to its i, then no

more cache coherence flood will happen. When this last thread finishes critical section and sets roll to -1 the same process repeats itself. But in no moment threads will keep replying to each other's actions without progress (coming closer to entering critical section).

3. What are the drawbacks of this algorithm (in terms of performance and requirement)?
   The requirement of having read/write to happen within a period of time (say one second) seems hard/impossible to achieve on high number of threads. That invariant seems to support an undefined number of concurrent threads. But as we know current typical multiprocessor architectures rely on bus technique as means to communicate with ,*shared main memory*. This bus is shared and if all processors are using bus at the same time causes collisions which causes traffic when reading / writing data to memory. More traffic causes more delay and then the more threads the longer it takes to read/write. But invariant is abstracted from number of threads. Thus **invariant will not be supported once number of threads causes enough traffic to delay read/write for period of time longer than the period of time specified**. Having many threads spinning on a lock is called *high contention*.
   Another disadvantage is **under-utilization** of critical section. Every thread needs to wait two time units, one to read and write and a second unit to wait for the *winner* thread to write its value. All this time could be instead spent on doing valuable work in critical section.

1.2 [5 points] Figure 1 shows a modified implementation of ALock, with the only change being the two statements (Line 24 and Line 25) in the unlock procedure are switched. Prove/disprove that this implementation is correct. (Consider size = 3)

Figure 1:

```
1  public class ALock implements Lock {
2
3      ThreadLocal <Integer> mySlotIndex = new ThreadLocal<Integer> (){
4          protected Integer initialValue(){
5              return 0;
6          }
7      };
8      AtomicInteger tail;
9      boolean[] flag;
10     int size;
11
12     public ALock(int capacity){
13         size = capacity;
```

```
14          tail = new AtomicInteger(0);
15          flag = new boolean[capacity];
16          flag[0] = true;
17      }
18      public void lock(){
19          int slot = tail.getAndIncrement() % size;
20          mySlotIndex.set(slot);
21          while(!flag[mySlotIndex.get()]{};
22      }
23      public void unlock(){
24          flag[(myslotIndex.get() + 1) % size] = true;
25          flag[mySlotIndex.get()] = false;
26      }
27  }
```

Answer

It is not correct. Please consider the following flow of actions leading to two threads being in critical section at the same time.

Thread0 enters lock and enters its critical section, slot = 0 →
Thread1 enters lock and waits in line 21, slot = 1 →
Thread2 enters lock and waits in line 21, slot = 2 →
Thread0 enters unlock(), line 24: sets flag[1] = true →
Thread1 enters its critical section →
Thread1 enters unlock(), line 24: sets flag[2] = true →
Thread2 enters its critical section →
Thread1 exists unlock, line 25: sets flag[1] = false →
Thread1 enters lock and reads flag[0] = true, enters critical section →

At the end of the sequence, both thread 1 and thread 2 are in critical section which is invalid. Problem is that thread 0 just did line 24 (sets flag[1] to true), but did not perform line 35 (set flag[0] to false) allowing in this small period of time for other two threads to enter their critical sections and one of them to exit and then come back again.

1.3 [10 points] MCS Lock (Figure 2)

Figure 2:

```
1   public class MCSLock implements Lock{
2       AtomicReference<QNode> queue;
3       ThreadLocal<QNode> myNode;
4       public MCLock(){
5           queue = new AtomicReference<QNode>(null);
```

```
        myNode = new ThreadLocal<QNode>(){
            protected QNode initialValue(){
                return new QNode();
            }
        };
    }
    public void lock(){
        QNode qnode = myNode.get()
        QNode qnode = queue.getAndSet(qnode);
        if(pred != null){
            qnode.locked = true;
            pred.next = qnode;
            while(qnode.locked){}
        }
    }
    public void unlock(){
        QNode qnode = myNode.get();
        if(qnode.next == null){
            if(queue.compareAndSet(qnode, null))
                return;
            while(qnode.next == null){}
        }
        qnode.next.locked = false;
        qnode.next = null;
    }

    static class QNode{
        boolean locked = false;
        QNode next = null;
    }
}
```

---

1. Consider a modified implementation of MCS Lock when getAndSet is not atomic, i.e., get() and set() are separate. Prove/disprove that this implementation is correct.
   Answer
   It is not correct, please consider below sequence of steps between three threads:

Thread0 enters lock() and sets its node0 as the tail of the queue. Enters CS →
Thread1 enters lock() and gets node0 referred by queue (line 14) →
Thread2 enters lock() and gets node0 referred by queue (line 14) →
Thread2 sets queue as its node2 (line 14, set part) →
Thread1 sets queue as its node1 (line 14, set part) →
Thread1 sets Thread0's node0.next = node1 →
Thread2 sets Thread0's node0.next = node2 →

At this point, once Thread0 finishes its critical section it will alert Thread2 node to start entering critical section. And once Thread2 finishes its critical section it will hang in line 26 because queue property is actually referring to Thread1 node1. Thus no thread will add another node after Thread2's node2. On the other hand , Thread1 will always hang in line 18 because no other node refers to node1 (node0.next is set to node2) , then no other thread can set node1.locked to false. Thus both thread1 and thread2 deadlocks as well as any thread trying to obtain lock afterwards.

2. Consider a modified implementation of MCSLock when compareAndSet is not atomic, i.e., compare() and set() are separate. Prove/disprove that this implementation is correct.
It is not correct, consider following sequence of events:

Thread0 enters lock() and sets its node0 as the tail of the queue. Enters CS →
Thread0 exits critical section (CS), in line 24
, performs the compare queue with qnode and finds it is true
, thus, it prepares to set queue to null next →
Thread1 enters lock() and gets node0 referred by queue (line 14)
, sets node0.next to its node1 and
starts waiting in line 18 for its property locked to become false. →
Thread0 continues with its execution and sets queue to null. Then exits unlock() →

At this point thread 1 will spin in line 18 forever because thread 0 did not notice that while it was setting the queue to null another thread scheduled itself after the node referred by queue. Thus, it is not correct.

3. Consider a modified implementation of MCS Lock where Line 16 and Line 17 are swapped. Prove/disprove that this implementation is correct.

Answer It is not correct. Consider following execution:

Thread0 calls lock() and enters CS →
Thread1 calls lock(), reaches line pred.next = qnode →
Thread0 enters unlock(), executes line queue.next.locked = false →
Thread1 executes line qnode.locked = true →
Thread1 loops in line while(qnode.locked) →

At this point Thread1 is going to hang forever in line 18 because the previous thread0 already existed unlock(). Future threads accessing queue will also be starved because thread1 is deadlocked.