CS5510: Multiprocessor Programming : Homework VI

Sergio Sainz

November 2019

1 Part I : Problems (65 points)

1.1 Stack (1) [14 points]

So far in class, we dealt with an atomic stack that allows n threads to push () and pop() from the stack concurrently preserving the LIFO semantics. Consider a new operation called get() on the stack: the operation should return the least recently pushed item present in the stack (i.e. the item at the bottom of the stack) for the first x calls. After the first x calls, the get() operation should return null. If the stack is empty, the operation should return null as well. Let's call this stack object NewStack(x), where x is used for get().

- Can a wait-free **Queue** (accessed by at most two threads) be constructed from an arbitrary number of NewStack(1) objects and atomic read-write registers? Prove your claim.
- Can a wait-free n-thread NewStack(2) object be constructed from an arbitrary number of atomic Stack objects and atomic read-write registers? Prove your claim.

Answer

A clarification here is that a read-write register means only read and only write separately. Thus task is to build concurrent wait-free Queue using only NewStack and without CompareAndSet or GetAndSet.

Here we use hint from Balaji where we can consider the Queue as implemented by an Array of limited capacity. Then we can assign each array node a stack. This stack will indicate who is truly the owner. Thus, even though there is a collision between two threads, they can still figure out the thread owner of slot by checking the NewStack get() method.

We implement only the enqueue method

```java
public class Node<T> {
    public volatile NewStack owner = new NewStack(1);
    public T item ;
}
public class QueueMadeOfStackOne<T>{
    static int QUEUE_SIZE = 64;
    int[] buckets = new int[QUEUE_SIZE];
    volatile int writer = 0;

    public QueueMadeOfStackOne(){
        for(int i = 0 ; i < QUEUE_SIZE ; i++){
            buckets[i] = new Node();
        }
    }

    public void enqueue(T item){
        int id = ThreadID.get();
        while(true){
            int ticket = writer;
            writer = writer + 1;
            int position = (ticket % QUEUE_SIZE);
            Node n = buckets[position];
            n.owner.push(id);
            if(owner.pop() == id){ // we get to go!
                n.item = item;
                return;
            }
        }
    }
}
```

As we can see from listing above there is a while(true) in the enqueue method telling us that the method is not wait-free. **This queue is lock-free but not wait-free.**

Why a wait-free enqueue method cannot be constructed? Because whenever the collision happens (because if using just reads and writes, both threads could end up with same ticket), the thread that looses the competition in inserting to the stack would need to retry. And once retrying the other thread could also participate and also win, and so on.

About the second question, ***Can a wait-free n-thread NewStack(2) object be constructed from an arbitrary number of atomic Stack objects and atomic read-write registers? Prove your claim.***

So we can assume below try:

```
public class NewStack2<T>{
    volatile int peekCount = 0;
    volatile int len = 0;
    volatile T first = null;
    volatile Stack stack = new Stack();

    public void push(T item){
        if(len == 0){
            first = item;
            len = len + 1;
            return;
        }
        stack.push(item);
        len = len + 1;
        return;
    }

    public T pop() throws EmptyException{
        if(len == 0){
            throws EmptyException ;
        }
        if(len == 1){
            T item = first;
            first = null;
            len --;
            return item;
        }
        T item = stack.pop();
        len--;
        return item;
    }

    public T get(){
        if(peakCount < 1){
            peakCount++;
            return first;
        }
        return null;
    }
}
```

Assume by contradiction that indeed push is supported. Then the following sequence of operations should not lose any item:

$$push_A(1) \rightarrow push_B(2) \rightarrow push_C(3)$$

But, if all three threads (A,B,C) call the push method at the same time, the three of them will read len as 0. Then they will enter line 9 and only element belonging to the thread that executes this line last will remain in the first variable.

Thus, the other two items that do set first earlier will not be in the stack. But this breaks the stack legal sequence specification. **Thus, at least this version of implementation, using solely Stacks and Read-Write registry does not allow for concurrent stack implementation with 2 peek support.**

1.2 Stack (2) [9 points]

A standard stack data structure has the following operations: push() and pop(). Are each of the executions presented in Figure 1 (for two threads), linearizable and/or sequentially consistent? Provide an explanation.

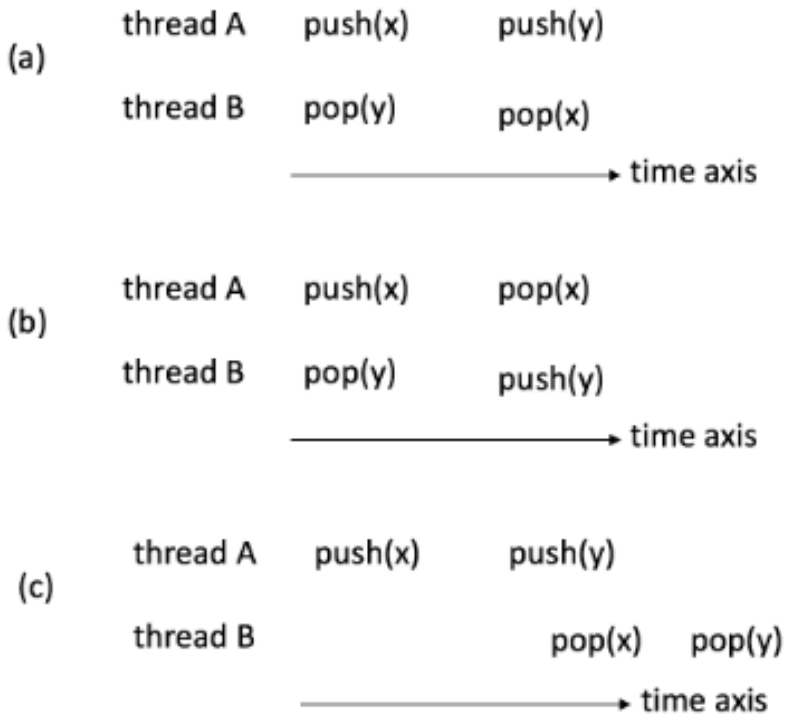

Figure 1:

Answer

From figure 1 we know that example :

- (a) - Not linearizable: Thread B pops y before any thread pushes y. Thus, breaking with Stack legal sequence. (an item must be pushed before it can be poped).
- (b) - Not linearizable: Thread B pops y before any thread pushes y. Thus, breaking with Stack legal sequence. (an item must be pushed before it can be poped).
- (c) - Linearizable - Thread A pushes x, then two operations happen simultaneously. These two operations that happen simultaneously could be applied in any order among each other. After these two operations are applied a pop of y is executed. Then we can arrange this history, H, as sequence S: $push(x) \rightarrow pop(x) \rightarrow push(y) \rightarrow pop(y)$ and because is a legal sequence for stack original (c) history is linearizable.

1.3 Stack (3) [10 points]

The implementation of lock-free unbounded stack is in Figure 2 and Figure 3. The ABA problem could arise in pop() method with improper garbage collection. How can this happen? Give a scenario. Also, explain how the code can be fixed to avoid the ABA problem.
Figure 2:

```
1  public class LockFreeStack<T>{
2      AtomicReference<Node> top = new AtomicReference<Node>(null);
3      static final int MIN_DELAY = 10;
4      static final int MAX_DELAY = 100;
5      Backoff backoff = new Backoff(MIN_DELAY,MAX_DELAY);
6
7      protected boolean tryPush(Node node){
8          Node oldTop = top.get();
9          node.next = oldTop;
10         return (top.compareAndSet(oldTop, node));
11     }
12
13     public void push(T value){
14         Node node = new Node(value);
15         while(true){
16             if(tryPush(node)){
17                 return;
18             }else{
19                 backoff.backoff();
20             }
21         }
22     }
```

```
}
```

Figure 3:

```
protected Node tryPop() throws EmptyException{
    Node oldTop = top.get();
    if(oldTop == null){
        throw EmptyException();
    }
    Node newTop = oldTop.next;
    if(top.compareAndSet(oldTop, newTop)){
        return oldTop;
    }else{
        return null;
    }
}

public T pop() throws EmptyException{
    while(true){
        Node rerturnNode = tryPop();
        if(returnedNode != null){
            return returnNode.value;
        }else{
            backoff.backoff();
        }
    }
}

public class Node{
    public T value;
    public Node next;
    public Node (T value){
        this.value = value;
        this.next = null;
    }
}
```

Answer

ABA problem could happen in the following sequence of operations:

Assume the stack has these elements: $X \rightarrow Y \rightarrow Z$.

$$inv_A : pop() : line6 \rightarrow inv_B : pop() \rightarrow ret_B : pop() : X \rightarrow$$
$$inv_B : pop() \rightarrow ret_B : pop() : Y \rightarrow inv_B : push(T) \rightarrow$$
$$ret_B : push(T) : true \rightarrow inv_B : push(X) \rightarrow ret_B : push(X) : true \rightarrow$$
$$inv_A : pop() : line7 : CAS \rightarrow ret_A : pop() : true$$

At the end of the sequence, the stack looks like this: $Y \rightarrow Z$. Even though T was pushed it does not appear in the stack at the end. Reason is that in between line 6 and line 7 of the tryPop method, there was a pop of element X, pop of element Y and push of element T then push of element X again. At this point stack has values $X \rightarrow T \rightarrow Z$. But then, the line 7 checks whether top is still X (which it still is) and if so replaces with the oldTop.next , Y, back in the first step when thread A executed line 6. Then, it replaces top with an element that does not even exists in the stack anymore, Y. And also element T is removed from stack.

The solution is to make the top variable an AtomicStampedReference. Then, everytime it is modified instead of just checking whether value is the same, we check whether value and stamp are same and if so increment stamp by one, $top.compareAndSet(oldTop, oldStamp, newTop, oldStamp{+}1)$. This would need to be changed in both line 7 of tryPop and in line 10 in tryPush methods.

If the top variable is stamped, in the sequence above, when thread A tries to execute line 7, it would fail because the stamp it obtained back when it obtained the oldTop.next variable is different compare with current stamp. Because everytime the top variable got modified (in the pop():X, pop():Y, push(T), push(X)) the stamp gets incremented by one, thus the stamp is 4 when the original stamp read by thread A is 0. Thus, the comparison will fail.

1.4 Queue (1) [10 points]

An implementation of an unbounded queue has been provided in Figure 4. The deq() is blocking in the sense that it spins until there is an item to dequeue. Assume that the bucket array is too large to be filled and last is the index of the next unused position in bucket.

- Explain whether enq() and deq() methods are wait-free or lock-free.
- What are the linearization points for enq() and deq()? Are they execution dependent or independent?

Figure 4:

```
1  public class YetAnotherQueue<T> {
2      AtomicReference<T>[] bucket;
3      AtomicInteger last;
4
5      public void enq(T item){
6          int idx = last.getAndIncrement();
7          bucket[idx].set(item);
8      }
9
10     public T deq(){
11         while(true){
12             int end = last.get();
13             for(int idx = 0 ; idx < end ; idx++){
14                 T value = bucket[idx].getAndSet(null);
15                 if(value != null){
16                     return value;
17                 }
18             }
19         }
20     }
21 }
```

Answer

- Explain whether enq() and deq() methods are wait-free or lock-free.

  Wait-free means that the method returns in a finite number of steps. Lock-free means that a delay from any given thread will not delay execution from other threads. Compared against lock-based algorithms and basically it means that a thread in a lock-based algorithm critical section could generate a cache miss and hence delay to fetch data from main memory and this delay affects every thread waiting for lock. On the other hand lock-free algorithms do not have critical section and therefore a cache miss in a thread does not causes other threads to wait for such thread.

  Based on it, then, method enq does not have any while(true) statement and thus it will just take two instructions always. Therefore method **enq is wait-free, and therefore also lock-free**.

  Method deq does have a while(true) statement and thus it could keep looping through it as long as all the statements bucket[idx].getAndSet(null) return null. Thus the number of instructions deq takes is not finite: as long as queue is empty the threads calling deq are going to spin in deq indefinitely. **deq is not wait-free, but it is lock-free as it does not wait on any particular thread (it waits on the queue being**

not-empty which is different).

- What are the linearization points for enq() and deq()? Are they execution dependent or independent?

  Linearization point of enq is execution dependent , because there are cases where the idx variable is assigned but the assignment of item to the bucket idx position gets delay. During this delay other enqueue threads could also get their idx counter and assign to their assigned location in bucket. If during such concurrent circumstance a dequeuer arrives and dequeues the item of the bucket related to the thread with higher index then , from perspective of the enqueuer, although has higher index it got dequeued before the item of the enqueuer that has lower index.

  To account for such situation, we can think that the **enqueue** linearization point of the enqueuer is either one of two points, the one that happens first is the linearization point:

  - Line 7: bucket[idx].set(item)
  - Or a bit after the maximum time among all points in time for which an element in bucket, whose index is higher than idx, gets dequeued from the bucket (line 14 in dequeuer T value = bucket[idx].getAndSet(null)).

  Thus, we can also express this in following way: enqueue linearization point of time is $= min(bucket[idx] = item,$ a bit after $max(\forall idx_b > idx; deq : bucket[idx_b] = null))$

  For **dequeue** is a bit simpler and execution independent: the linearization point is line 14, when the element is removed from bucket: $Tvalue = bucket[idex].getAndSet(null)$. Once a thread dequeues the item, no other thread can dequeue item afterwards.

1.5 Queue [12 points]

Figure 5 shows the implementation of BoundedLockFreeQueue's enqueue method. Provide a corresponding implementation of the dequeue method.
Reason about the correctness of your implementation. Note that the queue must support multiple concurrent enqueuers and dequeuers. Specify the points of linearization for both the enqueue and dequeue methods.
Figure 5:

```
1 public class BLFQueue{
2     private final static int QUEUE_SIZE = 64;
3     private final Item[] queue = new Item[QUEUE_SIZE];
```

```
4
5       private final AtomicInteger writer = new AtomicInteger();
6
7       BLFQueue(){
8           for(int i = 0 ; i < QUEUE_SIZE ; i++){
9               queue[i] = new Item();
10          }
11      }
12
13      void enqueue(int value){
14          int ticket = writter.getAndIncrement();
15          int turn = (ticket / QUEUE_SIZE) * 2;
16          int position = (ticket % QUEUE_SIZE);
17          Item it = queue[position];
18          while(it.lastID != turn);
19          it.value = value;
20          it.lastID = turn + 1;
21      }
22  }
```

Answer

The dequeue method would look like this:

```
1   public class BLFQueue{
2       ...
3       private final AtomicInteger reader = new AtomicInteger();
4       ...
5       int dequeue(){
6           int ticket = reader.getAndIncrement();
7           int turn = ((ticket / QUEUE_SIZE) * 2) + 1;
8           int position = (ticket % QUEUE_SIZE);
9           Item it = queue[position];
10          while(it.lastID != turn);
11          int val = it.value ;
12          it.lastID = turn + 1;
13          return val;
14      }
15  }
```

A new atomic integer is introduced, reader. reader keeps track of the dequeueing order. Then, every dequeuer concurrently gets a distinct read ticket to be allocated a space in the item array. Then dequeueing threads will wait until their allocated item has been assigned a value. After value has been assigned (controlled by flag "lastID"), dequeue threads will take the value, increment lastID by one and then return.

Notice both enq and deq are lock-free but not wait-free. They both need to wait for queue to be either not-full or not-empty respectively. We know this because both methods have a while(it.lastID != turn) statement that will be true when queue is full (for enq) or when queue is empty (for deq). Thus threads could wait indefinitely.

The linearization point for enqueue is similar as previous exercise: there could be cases when the ticket gets granted to many threads but the threads that have the higher ticket set its item's lastID to the next value and hence giving chance to spinning threads to dequeue this item. In these cases the linearization point is not defined by the write ticket number, but instead by the order when the items get dequeued. Thus, the **enqueue linearization** point is as follows:

$$
\begin{aligned}
&min(\\
&\quad enq : it.lastID = turn + 1 : line20, \\
&\quad max(\forall ticket_B > ticket; deq : queue[ticket_B \% QUEUE\_SIZE].lastID = turn + 1 : line13) \\
&\quad)
\end{aligned}
$$

Linearization point for dequeue is the line 13: $it.lastID = turn + 1;$. After this point the enqueuers can keep adding items to the queue.

1.6 StackQueue [10 points]

Is the Queue implementation in Figure 6 correct (i.e. does it support FIFO semantics)? Is it linearizable?

Figure 6

```
1  class StackQueue<T>{
2      private Stack<T> s1 = new LockFreeStack<>();
3      private Stack<T> s2 = new LockFreeStack<>();
4
5      public void enQueue(T x){
6          s1.push(x);
7      }
8
9      public T deQueue() throws EmptyException {
10         T x ;
11         try{
12             x = s2.pop();
```

```
13              return x;
14          } catch(EmptyException ignored){
15              try{
16                  while(true){ s2.push(s1.pop());}
17              } catch(EmptyException ignoredAlso){
18              }
19          }
20          x = s2.pop();
21          return x;
22      }
23  }
```

Answer

The following sequence of actions breaks the legal sequence of queue:

1. $enQueue_A(1)$
2. $enQueue_A(2)$
3. $enQueue_A(3)$
4. $deDeque_A() : line16 : s2.push(s1.pop() : 3)$
5. $enQueue_B(4)$
6. $deDeque_A() : line16 : s2.push(s1.pop() : 4)$
7. $deDeque_A() : line16 : s2.push(s1.pop() : 2)$
8. $deDeque_A() : line16 : s2.push(s1.pop() : 1)$
9. $deDeque_A() : returns : 1$
10. $deDeque_A() : returns : 2$
11. $deDeque_A() : returns : 4$
12. $deDeque_A() : returns : 3$

Notice how enq(3) has a precendence relationship with enq(4) (enq(3) happened before enq(4)). Nevertheless, deq():4 happens before deq():3, breaking the FIFO legal sequence specification. **Thus is not linearizable**.