CS5510: Multiprocessor Programming : Homework V

Sergio Sainz

November 2019

1 Part I : Theory [56 points]

1.1 Set (1) [7 points]

In listing 3, two entries are locked before key presence is determined. If no entries were locked and it instead returned true/false based on key existence, would this alternative still be linearizable? If so, explain; if not, give a counterexample.
Listing 3: contains:

```
1  public boolean contains(T item){
2      int key = item.hashCode();
3      while(true){
4          Node pred = this.head;
5          Node curr = pred.next;
6          while(curr.key < key){
7              pred = curr ; curr = curr.next;
8          }
9          pred.lock(); curr.lock();
10         try{
11             if(validate(pred, curr)){
12                 return curr.key == key;
13             }
14         } finally{
15             pred.unlock(); curr.unlock();
16         }
17     }
18 }
```

Answer:
Assuming lines 9, 15 are removed (pred.lock();curr.lock()/respective unlocks). And also we remove lines 11, 13: validate(pred, curr), then we just check if curr.key == key. If so, **then the method would still be linearizable**.
The explanation why follows first reminder of what is linearizability, then demonstrate that if we do not lock or validate then linearizability still is valid.
Linerizability is applicable to histories produced by systems. A history , H, is

linearizable if it has an extension, H', and there is a legal sequential history , S, such that :

- $complete(H')$ is equivalent to S.
- And if method call m_0 precedes method call m_1 in H, then the same is true in S.

In layman's terms, linearization is: in a history H if two method invocations have a precedence relationship among them (one of them finishes before the other one starts), then the effect of their actions should follow a legal sequential specification. And in other hand, if the methods do not have precedence relationship (both methods are being executed at the same time at some point) then the order of their actions is **ambiguous** and we are free to order them in whatever way is convenient for us.
We now look at potential problems that could arise in removing the locks from contains:

$$inv : contains_A(a) \rightarrow contains_A(line\ 6:\ curr.key == key). \rightarrow$$
$$inv : remove_B(a) \rightarrow return : remove_B(a) : true \rightarrow$$
$$contains_A(line\ 12:\ curr.key == key) \rightarrow returns : contains_A(a) : true$$

Similar problem could arise when searching for a non existing key and then midway some other thread adds the key.
Although above invocation of *contains* by thread A does indeed return true as if the element a exists in list even after thread B removed a from list, it is still linearizable because both method invocations happen at the same time and thus there is no precedence relationship among them. Thus, we can set the linearization point of contains at some point before thread B removed the element a. Similar argument could be made for the case when contains return false even though at mid point another thread C adds the key contains was looking for. The linearization point for contains is some point before thread C adds the element.
An important point is that the problems happen whenever several threads run contains and add/remove at the same time. If add/remove happen before or after contains, then everything is fine. Thus, linearization is respected in both cases (when methods run at the same time and when they run one after another).

1.2 Set (1) [7 points]

Will listing 1, by itself, function correctly if we switch the locking order on line 9? What about in context with listings 2 and 3? If it has issues in context, how can that be fixed?
Listing 1: add item

```
1 public boolean add(T item){
2     int key = item.hashCode();
```

```
    while(true){
        Node pred = head;
        Node curr = pred.next;
        while(curr.key < key){
            pred = curr; curr = curr.next;
        }
        pred.lock(); curr.lock();
        try {
            if(validate(pred, curr)){
                if ( curr.key == key){
                    return false;
                }else{
                    Node node = new Node (item);
                    node.next = curr;
                    pred.next = node;
                    return true;
                }
            }
        } finally{
            pred.unlock(); curr.unlock();
        }
    }
}
```

listing 2: remove item

```
public boolean remove(T item){
    int key = item.hashCode();
    while(true){
        Node pred = head;
        Node curr = pred.next;
        while(curr.key < key){
            pred = curr ; curr = curr.next;
        }
        pred.lock() ; curr.lock();
        try{
            if(validate(pred,curr)){
                if(curr.key == key){
                    pred.next = curr.next;
                    return true;
                }else{
                    return false;
                }
            }
        } finally {
            pred.unlock(); curr.unlock();
        }
    }
```

```
23 }
```

Answer
By itself it should work fine. And by itself I mean that there are many threads calling add() concurrently. Reason is that even if threads lock current and then come back to lock pred. If pred is already locked, it means is other thread's current. Thus, this thread would be locked waiting for its own pred lock or adding the item. This logic could continue until we reach head. Head with lock on head will finish at some point and then vacant its current which would allow other threads to continue.

If we consider other methods such as remove or contains a dead-lock could occur like so:
If list is like $a \rightarrow c$.
And thread A wants to remove c, and thread B wants to add b.

$$inv : remove_A(c) \rightarrow inv : add_B(b)$$
$$inv : remove_A(a.lock()) \rightarrow inv : add_B(c.lock())$$
$$inv : remove_A(wait : c.lock()) \rightarrow inv : add_B(wait : a.lock())$$

Similar behavior happens for contains.

1.3 Set (3) [7 points]

Show that listing 1 only needs to lock pred.
Answer
There are two cases when lock is needed: (1) when during the execution of add, another thread wants to add another node right after pred and before current. (2) when during the execution of add, another thread wants to remove either pred or current.
Let us explore the first case (1). In this case we have another thread that wants to add another node right after pred. In such case the other thread's add method will also lock pred and thus there will be no problem. Once the pred becomes unlocked, other thread will validate and find that pred no longer refers to that thread's current and thus restart from the beginning of the list.
Let us now consider the second case (2), when another thread wants to remove current. In such case the other thread will first lock pred, thus there will be no problem because the thread that is calling add will also lock pred. The thread that remains waiting for the lock will need to restart from the beginning of the list after obtaining the lock because the pred next node will no longer be current (current will either be pushed further into the list or removed) and thus validation will fail.

If another thread wants to remove pred itself, it would still first lock pred's pred, and then try to lock pred, thus having to wait until pred lock becomes available. Thus, in all cases where pred or current are removed or a node is added between them it is safe. Main reason is that although add does not lock on current, remove does lock on current and thus avoid having problems of adding nodes on nodes that are already removed (And thus adding no node). And avoid problems of removing node when really it is not really removed.

1.4 Linked List [7 points]

In linked-list algorithms, it is possible to reach a later node from one that is not reachable from the head of the list. Is this behavior useful and/or desirable? Explain and, if possible, give an example.

Answer

Yes, it is possible to reach a later node from one that is not reachable from the head of the list. When several threads are reading and modifying same list. The behavior is desirable because it allows for reading threads to keep on reading even though the node where its current iteration exists has already been removed. This is useful for lock-free contains methods such as Lazy List and Lock-free list. For example consider the lazy list contains method, in it, the calling thread iterates over the list until it finds he key. Consider the sequence of steps:

List is $a \rightarrow b \rightarrow c$

$$\begin{aligned} inv : contains_A(c) \rightarrow inv : remove_B(b) \rightarrow \\ contains_A(c : curr = b) \rightarrow ret : remove_B(b) : true \rightarrow \\ contains_A(c : curr = b.next) \rightarrow contains_A(c : curr == c) \rightarrow \\ ret : contains_A(c) : true \end{aligned}$$

Notice that even though node b is removed thread A still used it to move to c.

1.5 Compare-And-Swap (1) [7 points]

Many lock-free data structures are implemented using compare-and-swap. Could test-and-set operations instead be used to implement lock-freedom? Explain your answer.

Answer

Not, the reason why we cannot use test-and-set instead of compare-and-swap is because getting the values as one operation and then compare them as two different steps leaves room (in between getting the values and then comparing them) for other threads to modify the values (reference and mark) and then invalidate any subsequent modification original thread may plan to do. Meanwhile

using compare-and-swap allows both compare and swap as an atomic operation without leaving room for other threads to modify the state of the shared memory between the comparison and the swap.
For example: if thread A is trying to compare that pred.next is curr and also that pred.next is not marked. Then thread A first gets pred.next reference and marked value. Suppose that indeed pred.next points to curr and marked value is false (that is not deleted). Secondly, another thread , B, adds a node between pred and curr. Thirdly, thread A gets scheduled again and compares that the value obtained from pred.next previously is indeed curr and thus the comparison returns true. Subsequentially thread A adds another node between pred and curr and thus removing the node that thread B added.

1.6 Compare-And-Swap (2) [7 points]

Some naive approaches to lock-free deletion in linked lists use a single CAS, swapping the element to be deleted with its next node as shown below.

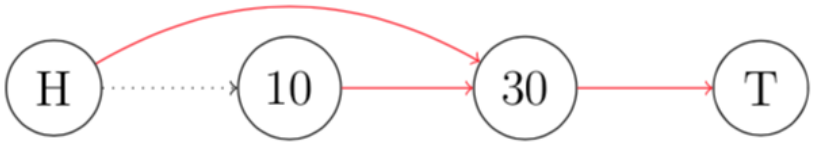

Figure 1:

Unfortunately, this can lead to the following situation:

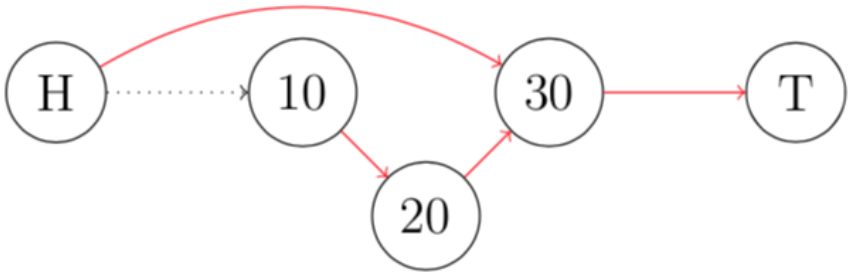

Figure 2:

Why can this happen? Why is it a problem?

Answer
This can happen when if the following sequence of operations happen:

$$\begin{aligned} inv : remove_A(10) &\rightarrow inv : add_B(20) \\ remove_A(10 : \text{pred=H,curr=10,curr.next=30}) &\rightarrow \\ add_B(20 : \text{pred=10,curr=30}) &\rightarrow \\ ret : remove_A(10 : \text{pred.next = curr.next}) : true &\rightarrow ret : add_B(20 : \text{pred.next = 20}) : true \end{aligned}$$

That is, two threads concurrently remove 10 and add 20. But because there is a gap of time between the time the node that will be removed is found and the time the reference of its predecessor next node is updated to the removed node next node (in example move H.next to 30). Then during this gap another thread , B, could come and add a node after node 10. Then once thread A completes moving H.next to 30 , it removed the node 20 added by thread B from the list.

1.7 Compare-And-Swap (3) [7 points]

To resolve the issue in section 1.6, some modern list implementations use two CAS's per delete. What is each one responsible for and how does that resolve the problem?
Answer
The new CAS operation is useful to mark a node to be deleted before the actual deletion of the node occurs (the original CAS). After node is marked for deletion (1st CAS) the thread that marks the node removes it physically from list (2nd CAS). Then, this solves the problem from 1.6 because once the node 10 is marked to be deleted by thread A, the concurrent add operation by thread B will not use a marked node 10 as predecessor of a new node 20.
Thus, in such case once thread B finds out node 10 is marked for deletion, add operation by thread B will restart its search for the best position to add node 20 from the beginning of the list.

1.8 Hash Collision [7 points]

How would you modify the LockFreeList and OptimisticList algorithms if the object hash codes are not guaranteed to be unique?
Answer
There are two options:

1. we could overwrite the **hashCode()** method in the Node class. Within hashCode, we can use more properties of item (type **T**) to generate an actual distinct hashCode. This solution of course requires domain knowledge about T. Thus, there is a second generic solution. Modify the algorithms themselves.

2. Modify the algorithms. **Lock-free list**, based on lines from figures 9.24 - 9.27:
   find $\rightarrow$ line 22, add parameter item to find and change condition from $curr.key >= key$ to $curr.key > key||(curr.key == key\&\&curr.item == item)$.
   add $\rightarrow$ change line 6 from $curr.key == key$ to $curr.key == key\&\&curr.item = item$.
   remove $\rightarrow$ change line 23 from $curr.key! = key$ to $curr.key! = key\&\&curr.item! = item$
   contains $\rightarrow$ change line 43 from $curr.key == key$ to $curr.key == key\&\&curr.item == item$.
   Then for **optimistic list**, based on lines from figures 9.11 - 9.14 from textbook:
   add $\rightarrow$ line 6 from $curr.key < key$ to $curr.key < key||(curr.key == key\&\&curr.item! = item)$. Then line 12 from $curr.key == key$ to $curr.key == key\&\&curr.item == item$.
   remove $\rightarrow$ line 31 from $curr.key < key$ to $curr.key < key||(curr.key == key\&\&curr.item! = item)$. Line 37 from $curr.key == key$ to $curr.key == key\&\&curr.item == item$.
   contains $\rightarrow$ line 54 from $curr.key < key$ to $curr.key < key||(curr.key == key\&\&curr.item! = item)$. Line 60 from $curr.key == key$ to $curr.key == key\&\&curr.item == item$.