The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 9

July 14, 2009

Exercise 100. Describe how to modify each of the linked list algorithms if object hash codes are not guaranteed to be unique.

Solution The simplest solution is to keep entries with the same hash codes next to one another in the list. Replace every line of the form

```
while (curr.key < key) {
```

with

```
  while (curr.key < key || (curr.key == key && !curr.item.equals(item))) {
```

(Note that if items are naturally ordered, say by implementing the Comparable interface, then we wouldn't need to sort them by hash code.)

Exercise 101. Explain why the fine-grained locking algorithm is not subject to deadlock.

Solution We can describe the synchronization state of a lock-based data structure by a directed *waits-for* graph: each node reprsents a thread, and there is an edge from B to A if B is waiting for a lock held by B. There is a deadlock if and only if the waits-for graph has a cycle.

The fine-grained locking algoirthm is not subject to deadlock because locks can be ordered by their position in the list. When a thread tries to acquire one lock, it holds only locks earlier in the list. If there were a cycle, then some thread would have to be holding a later lock and requesting an earlier one, which is impossible.

Exercise 102. Explain why the fine-grained list's add() method is linearizable.

Solution As usual, there are two cases to consider: successful and unsuccessful calls.

If the call is successful (returns *true*), then the linearization point occurs at Line **??**, when it sets the pred node's next field to point to its newly-allocated node (while pred is locked).

If the call is unsuccessful, then the linearization point occurs at Line **??** or Line **??** when it locks a curr node in the list with the sought-after key.

Exercise 103. Explain why the optimistic and lazy locking algorithms are not subject to deadlock.

Solution In the both algorithm, method calls acquire locks in the order they appear in the list: earlier locks before later locks. (See discussion of waits-for graph above.)

```
1
2   public boolean contains(T item) {
3     Node pred = null, curr = null;
4     int key = item.hashCode();
5     head.lock();
6     try {
7       pred = head;
8       curr = pred.next;
9       curr.lock();
10      try {
11        while (curr.key < key) {
12          pred.unlock();
13          pred = curr;
14          curr = curr.next;
15          curr.lock();
16        }
17        return (curr.key == key);
18      } finally {
19        curr.unlock();
20      }
21    } finally {
22      pred.unlock();
23    }
24  }
```

Figure 1: The FineList class: contains() method

Exercise 104. Show a scenario in the optimistic algorithm where a thread is forever attempting to delete a node.

Hint: since we assume that all the individual node locks are starvation-free, the livelock is not on any individual lock, and a bad execution must repeatedly add and remove nodes from the list.

Solution The remove() method could starve if each call to validate() returns *false* because some other thread had deleted pred or curr, or inserted a node between them.

Exercise 105. Provide the code for the contains() method missing from the fine-grained algorithm. Explain why your implementation is correct.

Solution See Figure 1. This method follows the same hand-over-hand locking strategy as the other FineList methds, but it returns without modifying the list.

Exercise 106. Is the optimistic list implementation still correct if we switch the order in which add() locks the pred and curr entries?

Solution No, because it can deadlock with a remove() call that could lock the same nodes in the opposite order.

Exercise 107. Show that in the optimistic list algorithm, if pred_A is not *null*, then tail is reachable from pred_A, even if pred_A itself is not reachable.

Solution It is easier to show that once tail is reachable from any node, it remains reachable. Initially, tail is reachable from every node in the list. There are only two places the list is modified. Both add() and remove() update their pred nodes. If tail was reachable from a node without passing through pred, then it remains reachable after modifying pred. It remains to check that if tail was reachable from a node through pred, then it remains reachable after each modification.

- The add() method (Figure 2) prepares a new node by setting its next field to a node in the list Line 16, so tail is reachable from the new node. The new node is then spliced into the list by updating pred's next field (Line 17). If tail was reachable from a node through pred, then it is still reachable.
- The remove() method (Figure 3) modifies the list at Line 38, it sets pred.next to pred.next, where by assumption tail is reachable through pred.next. It follows that if tail was reachable from a node through pred, then it is still reachable.

Each time the method assigns to pred_A, it assigns it a node reachable from

Exercise 108. Show that in the optimistic algorithm, the add() method needs to lock only pred.

Solution We examine the three possible ways a method call can overlap an add() call.

- Two concurrent add() calls: For two concurrent adds to cause any problems, they must be adding an entry at the same position. Since both threads must lock the predecessor, one thread will have to wait for the other. If one thread is trying to insert an element between entries a and b and another is trying to insert one between b and c, then there is no conflict since b is only modified by the thread inserting between b and c. If a thread is trying to insert after a value that is currently being added by another thread, then it must have validated it successfully in order to succeed, and so the insertion must have completed.

```
1  public boolean add(T item) {
2    int key = item.hashCode();
3    while (true) {
4      Node pred = 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         if (curr.key == key) {
13           return false;
14         } else {
15           Node node = new Node(item);
16           node.next = curr;
17           pred.next = node;
18           return true;
19         }
20       }
21     } finally {
22       pred.unlock(); curr.unlock();
23     }
24   }
25 }
```

Figure 2: The OptimisticList class: the add().

```
26 public boolean remove(T item) {
27   int key = item.hashCode();
28   while (true) {
29     Node pred = head;
30     Node curr = pred.next;
31     while (curr.key < key) {
32       pred = curr; curr = curr.next;
33     }
34     pred.lock(); curr.lock();
35     try {
36       if (validate(pred, curr)) {
37         if (curr.key == key) {
38           pred.next = curr.next;
39           return true;
40         } else {
41           return false;
42         }
43       }
44     } finally {
45       pred.unlock(); curr.unlock();
46     }
47   }
48 }
```

Figure 3: The OptimisticList class: the remove() method.

- remove() call overlapping with mAnadd call: First, let's see what happens when the node marked as curr in the add() method is being deleted. Then one method would lock the other out since both try to acquire the same pred lock. If the node marked as pred is the one being deleted, then we are saved again by the locks since the node is being locked as the pred in add() and as the currEntry in remove().

- contains() overlaps add(): The contains() method does not cause any problems since it simply goes through the list and checks if it finds an element in the list. By not locking curr in add(), contains() is left completely as it was.

Exercise 109. In the optimistic algorithm, the contains() method locks two entries before deciding whether a key is present. Suppose, instead, it locks no entries, returning *true* if it observes the value, and *false* otherwise.

Either explain why this alternative is linearizable, or give a counterexample showing it is not.

Solution It is linearizable. Suppose the contains(x) call returns *true*. Then there must have been some instant during that call when x's node was reachable from head. Linearize the call at any such moment.

Suppose the contains(x) call returns *false*. Then there must have been some instant during that call when x's node was not reachable from head. Linearize the call at any such moment.

Exercise 110. Would the lazy algorithm still work if we marked a node as removed simply by setting its next field to *null*? Why or why not? What about the lock-free algorithm?

Solution The lazy algorithm would not work. The problem is that by removing the node by simply setting its next field to null we lose information - specifically, the rest of the list! This could be modified to work if we make a couple of changes. First, we must save a pointer to the rest of the list after setting the removed node's next field to null. We also need to have a special sentinel node to mark the end of the list instead of using the null pointer. Note that this "fix" is pointless since we aren't really setting the next field to null, we are just copying it to another location.

Not only would the LockFreeList algorithm not work if we remove a node by setting its next field to null, our "solution" above would also not work because of the compareAndSet()() operation. We would need to lock the tail of the list to remedy this problem, but then the list would no longer be lock-free.

Exercise 111. In the lazy algorithm, can pred_A ever be unreachable? Justify your answer.

Solution Yes, there is nothing to prevent one thread from unlinking pred_A while another thread is scannng though the list.

Exercise 112. Your new employee claims that the lazy list's validation method (Fig. **??**) can be simplified by dropping the check that pred.next is equal to curr. After all, the code always sets pred to the old value of curr, and before pred.next can be changed, the new value of curr must be marked, causing the validation to fail. Explain the error in this reasoning.

Solution The problem is that another thread could insert a new node between pred and curr. An add() or remove() call using the simplified validation would remove all the nodes between pred and curr.

Exercise 113. Can you modify the lazy algorithm's remove() so it locks only one node?

Solution It is not necessary to lock curr.

Both add() and remove() lock pred before accessing curr. Once pred is locked, by either method, then no other add() or remove() can access that node. The contains() method is lock-free, and works whether or not curr is locked.

Exercise 114. In the lock-free algorithm, argue the benefits and drawbacks of having the contains() method help in the cleanup of logically removed entries.

Solution The principal disadvantage is that the contains() method is usually the most frequently called, and calling compareAndSet() would slow it down considerably.

The principal advantage is that doing so might speed up the other methods, which would have to do fewer phyical deletions.

Exercise 115. In the lock-free algorithm, if an add() method call fails because pred does not point to curr, but pred is not marked, do we need to traverse the list again from head in order to attempt to complete the call?

Solution Since any unmarked reachable node is reachable, it would be correct to resume the traversal from pred.

Exercise 116. Would the contains() method of the lazy and lock-free algorithms still be correct if logically removed entries were not guaranteed to be sorted?

Solution No, because the contains() method might stop searching prematurely if it found a logically deleted node with a higher key.

Exercise 117. The add() method of the lock-free algorithm never finds a marked node with the same key. Can one modify the algorithm so that it will simply insert its new added object into the existing marked node with same key if such a node exists in the list, thus saving the need to insert a new node?

Solution No, because the node might be physically removed while the new value is being installed.

Exercise 118. Explain why this cannot happen in the LockFreeList algorithm. A node with item x is logically but not yet physically removed by some thread, then the same item x is added into the list by another thread, and finally a contains() call by a third thread traverses the list, finding the logically removed node, and returning *false*, even though the linearization order of the remove() and add() implies that x is in the set.

Solution The thread adding x the second time would physically remove the marked node containing x.