Lucas Rosa
04/06/16
2847275

Homework Assignment 2

1. **Chapter 9**
   - **Exercise 102**

     To show that every call to add() is linearizable, it must be shown that there exist linearization points such that:

     i. Whenever add returns true the given item was not in the list before this point and it is inserted to the list after this point.

     ii. Whenever add() returns false, the item must already have been in the list before this point.

     In both case (1) and (2), choose the point when control exists the loop on line 14 as the linearization point. After the while loop terminates, the thread is holding the locks on curr and pred. By the way in which the locks are acquired, the loop ensures that pred must be reachable from head and that pred.next == curr must still hold after the loop. Moreover, the loop ensures pred.key < key. From the loop conditions we additionally know that curr.key >= key holds after the loop.

     Now, if add returned true, then the condition on line 15 was false, i.e., we must have curr.key > key. From the established properties and the sortedness of the list, it follows that item is not yet in the list. Hence, the insertion of the item is correct.

     If on the other hand add() returned false, then we have curr.key = key. Since curr is reachable from head and by the uniqueness of keys, it follows that item is already in the list. Again, add() returns the correct result.

   - **Exercise 104**

     If validate keeps failing thread A will loop forever and never delete the node. It is said to be rare but if it does happen then it will loop forever. Thread A will traverse the list and look for the node to remove. After finding the node to remove it then proceeds to acquire the node level locks. If in between the recognition of the node and the locking of the node another thread B manages to insert a node, then thread A will fail the validation and start the method all over again. If this repeats every time, then thread A will attempt to delete a node forever.

Lucas Rosa
04/06/16
2847275

- **Exercise 106**

  No, because it creates a conflict with the lock order in the remove method. Thread A attempts an add(5) and find the location to add the new node by traversing the list. Thread A locks curr and will then proceed to lock pred. Thread B attempts a remove(4) and happens to be looking for thread A's pred. If thread B successfully manages to acquire the lock on it's curr, which is threads A's pred, before thread A can lock pred then thread B will remove that node and pred will be referencing something no longer in the list, causing the validate method to return false and thread A restarting it's method call to add. This creates the potential for add() to attempt to add a node forever, although it will probably be rare given a large enough list.

- **Exercise 108**

  Thread A attempts an add(5). It traverses the list, finds curr which is greater than 5, lets say 6 for this example. Let us also assume pred is 4 in this case. Thread A locks pred and may proceed as usual without locking curr, 6, before inserting 5. Let's imagine that a thread B attempts remove(6) or contains(6) while thread A has the lock for pred which is 4 in our example. Thread B will have to wait for thread A to finish with 4 before locking it and thus cannot acquire the lock for 6 which would be the curr for both threads. A would insert 5 without interference and release the lock on pred allowing B to proceed. The first validate would fail because thread B's current pred and curr now has a new node in between it. On it's next attempt it should succeed if there is no more interference.

- **Exercise 109**

  The alternative implementation is not linearizable. Consider an interleaved execution of two threads A and B. Suppose thread A first executes add(1) and then executes remove(1) up to the beginning of line 34. Now, let B execute contains(1) right up to before it evaluates curr.key == key and returns the result. At this point, since 1 is in the set, we must have curr.key == key. Now, let A continue its execution of remove(1). Since B is not holding any locks, A can complete its execution without blocking, removing 1 from the set and returning true indicating successful removal. Now, B continues. Since curr.key == key still holds it returns true, even though 1 is no longer in the set.

- **Exercise 110**

  No, because you would lose the rest of the list in the process. Another thread would attempt to traverse the list and would stop at this logically removed node with no access to the rest of the list. Pretty much, doing this without setting pred.next to curr.next breaks the list. This logic also applies to the lock-free algorithm or any linked list in general. In a linked list you cannot break a connection without creating a new one to "stitch" the list back together.

- **Exercise 112**

  The new employee forgot about the add() method, which may modify pred.next without making curr. The modified implementation would no longer be linearizable. Specifically, suppose thread A executes add(1) right up to the beginning of line 9. At this point, pred and curr point to the two sentinel nodes of the list. In particular, curr.key == key does not hold. Now, another thread B executes add(1) until completion, returning true to indicate the successful insertion. Finally, A continues its execution. The call to validate returns true since neither of the sentinel nodes has been marked. Moreover, curr.key == key does not still hold. Hence, A reinserts 1 into the list and also returns true, even though 1 was already present in the list.

- **Exercise 115**

  Yes, because as long as pred does not point to curr it will continue to fail. It will need to loop around again and get new pred and curr pointer. Otherwise the only way it will proceed is if some how the list was modified in between execution and pred now points to curr. Usually if pred does not point to curr that is because there is now something in between the two nodes, which could have been placed there for a number of reasons.

- **Exercise 116**

  No, because if the logically removed entries were unsorted then the contains could possibly terminate early if a value that is greater than the key shows up before the key. The method returns false when the value to look for is actually in the list. All of the implementations of linked lists in this chapter has built on top of one another and from the beginning the sorted nodes has been detrimental to the correctness of these linked lists.