Quiz 2

Oct. 2023

1. Hand-over-hand algorithm: (a) Check if contains requires locking; (b) What if traverse (in remove, insert) checks the value in curr before locking it (only holds lock on pred when traverse terminates)? (c) Can we just use one lock at a time? (d) Prove starvation-freedom (assuming starvation- free locks)

   (a) `contains` does require locking because `remove` can be happened concurrently with `contains`. If `update` of `remove` happens before `contains` (which means `remove` the link between the nodes first and then `return curr.key == item`) under the condition of non-locking, then `contains` will return false.
   (b) If traverse checks the values in `curr` before locking it, it will make some unsuccessful operations.
   (c) It is impossible to use one lock without leading any errors, because all the operations are concerning about two nodes, which are `pred` and `curr`.

2. Optimistic algorithm: (a) Show that validation is necessary for updates (Hint: consider an algorithm without validation and show that an update can get lost because of a series of concurrent removes); (b) Is validation necessary for contains? (c) Show that the algorithm is not starvation-free (even if all locks are)

   (a) `Validation` aims to determine that the node can be reached from the head. If there is a thread attempts to `remove` the node while the node before the current node might be removed by another thread, which causes the concurrency between `remove`. Without `validation`, the current node would be removed successfully even if it is no longer in the list.
   (b) `validation` is not necessary for the `contain`, because if a node has been removed before the `validation`, the result is also correct.
   (c) A thread might be delayed forever if new nodes are repeatedly added and removed.

3. Lazy algorithm: (a) Is the check !curr.marked necessary in contains? (b) Show that both conditions (checking !pred.marked and pred.next=curr) in validation are necessary (Hint: consider concurrent removes on two consecutive nodes, or a remove concurrent to an insert of a preceding node);(c) Determine linearization points for all operations: insert(successful or not), remove (successful or not), contains (successful or not) (Hint: for an unsuccessful contains(x), linearization point may vary depending on the presence of a concurrent insert(x))

   (a) It is necessary in `contains`, because `!curr.marked` means the node is in the list logically. If there is a thread is attempting to `remove` the node before the `contains` return, `curr.marked` could be modified directly by `remove` so that the result of contains can be correct.
   (b) `!pred.marked` means that `pred` is in the list and `pred.next = curr` means that `pred` connects to `curr`. If `!pred.marked` ignored, then all of the modifications to `pred` would be not updated. If `pred.next = curr` ignored, then when we insert between `pred` and `curr`, there is a concurrency between `insert` and another operation to `curr` and all the modifications will not be updated.
   (c) `insert` (successful): `pred.next = node`, `insert` (unsuccessful): `curr.key == item`, `remove` (successful): `curr.marked = true`, `remove` (unsuccessful): `curr.key != item`, `contains` (successful): the moment unmarked node has been found, `contains` (unsuccessful): the moment that the node found has been marked or the moment that found a node but not the correct node (`remove` the node and then `insert` the same node).