1 Linked-List

Harris Linked-List uses an Atomic-Markable-Reference object, in which the next field of a Node, in addition to a reference to the next node in the list, is also marked or unmarked. The two fields can be update atomically, either together or individually. This can be done by using the most-significant-bit of next for the marking. For simplicity, we assume reading next returns the reference only, while get_data() function is used to get (atomically) both the reference and mark bit. Moreover, whenever performing CAS on node.next, both reference and mark state should be mention. For ease of presentation, we assume a List is initialized with head and tail, containing keys $-\infty, \infty$ respectively. We allow no insert or delete of these keys.

A brief description of the original implementation and its linearization is as follows. The Lookup procedure is used by Insert and Delete in order to find the node with the lowest key greater or equal to the input key, and its predecessor on the list, while physically removing any marked node on its way. To insert a key α, a process first finds the right location for α using the Lookup procedure, and then tries to set pred.next to point to a new node containing α by performing CAS. To delete a key α, a process search for it using the Lookup procedure, and then tries to logically delete it by marking the next field using CAS. In case the marking was successful, the process also tries to physically remove the node. To find a key α, a process simply looks for a node in the list with key α which is unmarked.

The linearization point for the original implementation are as follows:

Insert: At the point of a successful CAS

Delete: At the point of a successful CAS for marking the node (logical delete)

Find: At the point where the procedure return, that is, at the read of either curr.key or curr.next.

Following the given linearization points (omitting proof...), insert and delete operation are linearized at the point where they affect the system. That is, if an insert operation performed a successful CAS, then all process will see the new node starting from this point, and if a node was logically delete, then all processes treat it as if it was removed. Therefore, once a process p recovers following a crash, the list data structure is consistent - if p has a pending operation, either the operation already had a linearization point which affected all other processes, or it did not affect the data structure at all, nor will in any future run.

However, even though the list data structure is consistent, the response of the pending operation is lost. Consider for example a scenario in which process p performs $Delete(\alpha)$ and crash right after applying a successful CAS to mark a node. Upon recovery, p may be able to decide α was removed, as the node is marked. Nevertheless, even if no other process takes steps, p is not able to determine whether it is the process to successfully delete α, or that it was done by some other process, and therefore it does not able to determine the right response. Moreover, in case the node was physically removed, p is not able to determine whether α has been deleted at all, as it is no longer part of the list.

1.0.1 Linked-List Recoverable Version

To solve the problems mention above, we present a modification for the algorithm such that in case of a process crash, upon recovery it is able to complete its last pending operation if needed, and also return the response value in such case. The algorithm presented in figure 1. Blue lines represents changes comparing to the original algorithm.

Each node is equipped with a new field named deleter. This field is used to determine which process is the one to delete the node. After the node was successfully marked (logical delete), process p tries to announce itself as the one to delete the node by writing its id to deleter using CAS. This way, if a process crash during a delete, it can use deleter in order to determine the response value. We assume deleter is initialised to null when creating a new node.

Each process p has a designated location in the memory, Backup[p]. Before trying to apply an operation, p writes to Backup[p] the entire data needed to complete the operation. Upon recovery, p can read Backup[p], and based on it to complete its pending operation, in case there is such. Formally, Backup[p] contains a pointer to a structure containing all the relevant data. For simplicity, process p creates new such structure for each of its operations, although a more efficient way will be to use two such structures in an alternating way.

Algorithm 1: ⟨Node, Node⟩ SEARCH (T *key*)

```
1  Data: Node $*pred, *curr, *succ$
         boolean $mbit$
2  retry: while true do
3      $pred := Head$
4      $curr := pred.next$
5      while true do
6          $\langle succ, mbit \rangle := curr.next$.get_data()
7          if $mbit$ then                                          // succ was logically deleted
8              if $pred.next$.CAS($unmarked\ curr, unmarked\ succ$) = false then   // help physical delete
9                  go to retry                                     // help failed
10             $curr := succ$                                      // help succeed
11         else
12             if $curr.key \geq key$ then          // $curr$ is the first unmarked node with key $\geq key$
13                 return $\langle pred, curr \rangle$
14             $pred := curr$                                      // advance $pred$ and $curr$
15             $curr := succ$
16     end
17 end
```

Algorithm 5: boolean RECOVER ()

```
Data:    Node $*nd := Backup[pid].nd$
if $Backup[pid].result \neq \perp$ then                          // operation was completed
    return $Backup[pid].result$
if $Backup[pid].optype =$ INSERT then
    $\langle pred, curr \rangle :=$ SEARCH($nd.key$)               // search for $nd$ in the list
    if $curr = nd \mid\mid nd.next$ is marked then                // $nd$ is in the list or marked
        $Backup[pid] :=$ true
        return true
    return FAIL
if $Backup[pid].optype =$ DELETE then
    if $nd \neq \perp$ && $nd.next$ is marked then                // $nd$ was logically deleted
        $nd.deleter.$CAS$(\perp, pid)$                            // try to complete the deletation
        if $nd.deleter = pid$ then                                // you are the deleter
            $Backup[pid].result :=$ true
            return true
    return FAIL
```

Shared variables: Node $*Head$

```
Type Info {
    {INSERT, DELETE} optype
    Node *nd
    boolean result
}
```

Code for process p:

Algorithm 2: boolean INSERT (T key)

```
Data:  Node *pred, *curr
       Node newnd := new Node (key)
Backup[pid] := new Info (INSERT, newnd, ⊥)
while true do
    ⟨pred, curr⟩ := SEARCH(key)                    // search for the right location for insertion
    if curr.key = key then                                       // key is already in the list
        Backup[pid].result := false
        return false
    else
        newnd.next := unmarked curr
        if pred.next.CAS (unmarked curr, unmarked newnd) then                  // try to add newnd
            Backup[pid].result := true
            return true
end
```

Algorithm 3: boolean DELETE (T key)

```
Data:  Node *pred, *curr, *succ
Backup[pid] := new Info (DELETE, ⊥, ⊥)
⟨pred, curr⟩ := SEARCH(key)                                       // search for key in the list
if curr.key ≠ key then                                               // key is not in the list
    Backup[pid].result := false
    return false
else
    Backup[pid].nd := curr
    while curr.next is unmarked do                          // repeatedly attempt logical delete
        succ := curr.next
        curr.next.CAS (unmarked succ, marked succ)
    end
    succ := curr.next
    pred.next.CAS (unmarked curr, unmarked succ)                          // physical delete attempt
    res := curr.deleter.CAS(⊥, pid)                               // try to announce yourself as deleter
    Backup[pid].result := res
    return res
```

Algorithm 4: boolean FIND (T key)

```
Data:  Node *curr := Head
while curr.key < key do             // search for the first node with key greater or equal to key
    curr = curr.next
end
return (curr.key = key && curr.next is unmarked)
```

Figure 1: Recoverable Non-Blocking Linked-List

Correctness Argument

In the following, we give an high-level proof for the correctness of the algorithm.

First, notice that quitting the Lookup procedure at any point, or repeating it, can not violet the list consistency. The Lookup procedure simply traverse the list, while trying to physically delete marked nodes. Once curr.next is marked, a single process can perform the physical delete. This follows from the fact that at any point there is a single node in the list which points to curr. Once curr is physically delete, no node in the list points to curr, and thus any CAS operation with curr as the first parameter will fail. This observation relays on the fact that any new allocated node has a different address then curr. As a result, repeating the attempt to physically delete a node does not affect the list.

Assume a process p performs an $insert(key)$ operation. First, p writes to Backup[p], updating it is about to perform an Insert. If a process p does not crash, then, as in the original algorithm, it repeatedly tries to find the right location for the new node, and insert it by performing a CAS changing $pred.next$ to point to newnd. In addition, it is clear from the code that a crash after updating $Backup[p].result$ is after the operation had its linearization point, and the Recover procedure will return the right response. Therefore, we need to consider a crash before an update to $Backup[p].result$. There are two scenarios to consider.

Assume p crash without performing a successful CAS in line 27. p is the only process to have a reference to newnd, and it is yet to update any node with this reference, and thus no node points to newnd. As a result, the operation did not affects any other process, nor it will be in the future. Hence, considering the operation as not having a linearization point does not violate the list consistency. Indeed, since no node points to newnd, upon recovery p will see that newnd is not in the list and also not marked, and thus will return FAIL. Notice this argument holds whether key is already in the tree, or not, as the operation in both cases did not affect the system.

Assume now p crash after performing a successful CAS in line 27. In such case, newnd is part of the list, as pred.next points to it. Also notice we did not delete any other node, since pred.next pointed to curr, and after the CAS it points to newnd which points to curr. As a result, when p executes the Recover procedure, either it will see newnd in the list, or that it is no longer part of the list, and it must be some other process deleted it, and hence newnd.next is marked. In any case, p will return true as required. The above argument relies on the fact a marked node can not be unmarked, and that an Insert and Delete can not mistakenly remove nodes from the list. We have claimed it for Insert, and we will prove the same holds for Delete. Therefore, if a node is no longer in the list, it must be marked.

Assume a process p performs a $delete(key)$ operation. First, p writes to Backup[p], updating it is about to perform a Delete. As before, a crash after writing to Backup[p].result will return the right response. Also, a crush before updating any of Backup[p].result or Backup[p].nd implies p is yet to try and mark any node, and thus the operation did not affect the system so far, nor it will be in the future. Therefore, we can consider the operation as not having a linearization point (even in case key is not the list), and indeed, the Recover procedure returns FAIL in such case.

Assume thus p writes to Backup[p].nd. It follows that p completed the lookup procedure and finds a node curr storing key. The lookup procedure guarantees there is a point in time (of the procedure execution) where curr is in the list and curr.next is not marked. If p crash and recovers, and observe that curr is unmarked, then in returns FAIL. Since a marked node can not be unmarked, as there is no CAS changing a marked node, it follows that p did not marked curr. Therefore, the operation did not affects any process, nor it will be, and we consider it as having no linearization point. Otherwise, the Recover function observe curr as marked, and we can conclude the marking point of curr is along the delete operation. We now prove we can linearize the operation, according to its response.

Let q be the process to mark curr. Since once curr.next is marked it will never be changed, the reference of curr.next is fixed to succ (of q at the point of the marking). This also implies q is unique and well defined, and any future CAS on curr.next will fail. As a result, any process leaving the while loop in line 39 reads the same value in line 43, which is this succ. The attempt to physically delete curr in line 44 will succeed only if pred.next points to curr, and as we said, curr point to succ, and any other attempt will fail. Thus, if this attempt succeed, it deletes only curr, and can not delete additional nodes.

In line 45 process p tries to writes its id to curr.deleter. As it is initialised to null, only the first process to perform this CAS will succeed. Also, any p must go through line 45 in order to complete its operation, as the Recover procedure redirect the process to this line. Therefore, if there is a process to complete its delete operation while observing curr.next is marked, there must be a CAS to curr.deleter. Let q' be the first process to perform this CAS. As proved above, q' tries to delete curr, and the point in time where curr is marked must be contained in its operation interval. Moreover, q' is the only process to write to curr.deleter, and the first one to do so, thus q' is the only process to obtain true when testing (curr.deleter $= q'$) in line 46 (and thus to also return true), while any other process will obtain false. We linearize the operation of q' at the point of the marking, and any other attempt to delete curr is linearized after it (in an arbitrary order).

A corollary of the analysis is that processes trying to delete the same node curr "helps" each other, in the sense that they all keep trying to mark curr. However, the marking process is not necessarily the one to return true. Also, in the original algorithm, if a process fails to mark a node, it starts the delete operation from the beginning. In our implementation, process can keep trying to mark the node without the need to perform a lookup again after each failed CAS. We guarantee that once curr is marked, exactly one process will return true, while the rest can consider curr as being deleted (in the course of their delete execution), and thus there is a point along their execution is which key is not in the tree, and they can return false.