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.

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 looks for it using the Lookup procedure, and then tries to logically remove it by marking curr.next 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 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 removed, then all processes treat it as a removed node. 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 a process fails, upon recovery it is able to complete its last pending operation and also return the response value. The algorithm presented in figure [.....]. Blue lines represents changes comparing to the original algorithm (lines that has been added, except for one that was change. See the notes).

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 a process p successfully mark a node (logical delete), it tries to write its id to deleter using CAS. This way, if a process fails during a delete, it can use deleter in order to determine the response value. We assume deleter is initialized 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, a process creates a new operation structure each time it writes to Backup, although, if used in an alternating manner, two such structures are enough.

Procedure Recover (void)

```
54 if Backup[p].result ≠ null then
55 |  return Backup[p].result
56 if Backup[p].optype = Insert then
57 |  if Backup[p].nd is in the list || Backup[p].nd.next is marked then
58 |  |  return true
59 |  else
60 |  |  return FAIL
61 if Backup[p].optype = Delete then
62 |  if Backup[p].nd ≠ null && Backup[p].nd.next is marked then
63 |  |  go to 45 with curr := nd                          // try to complete the deletation
64 |  else
65 |  |  return FAIL
```

Procedure < Node, Node> Lookup (T key)

```
1  Data: Node *pred, *curr, *succ
          boolean mbit
2  retry: while true do
3  |  pred := head
4  |  curr := head.next
5  |  while true do
6  |  |  ⟨succ, mbit⟩ := curr.next.get_data()
7  |  |  if mbit then                                     // is succ was logically deleted?
8  |  |  |  if pred.next.CAS (unmarked curr,unmarked succ) = false then
9  |  |  |  |  go to retry                                                  // help failed
10 |  |  |  curr := succ                                                   // help succeed
11 |  |  else
12 |  |  |  if curr.key ≥ key then
13 |  |  |  |  return ⟨pred,curr⟩
14 |  |  |  pred := curr
15 |  |  |  curr := succ
16 |  end
17 end
```

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.

A key argument in the proof is that if an insert operation is yet to perform a successful CAS (in line 27), then considering the operation as not having a linearization point does not violate the list consistency. This follows from the fact the operation did not affected any process. In particular, repeating the operation in such case is also safe. As a result, if a process performing insert is about to return false, due to the fact key is already in the tree, and crash, then considering the operation as FAIL upon recovery does not violate consistency. A similar argument holds for delete operation and CAS for logic delete.

fails because key is already in the list, then aborting it (i.e., considering it as not having a linearization point) does not violate the consistency of list. This follows from the fact that such an operation does not affect any other process. Moreover, even repeating the operation in such case (assuming it did not return yet) can not violate consistency, from the same reason. A similar argument holds for delete operation which fails because key is not in the list.

Assume a process p performs $insert(key)$ operation. Then, p creates a new node ndnew with

For the Insert operation, p tries to add the new node by performing a CAS. If it succeeds it will return true if it suffers no failure. In case of a failure after writing to Backup[p], upon recovery p tries to complete its operation. If it already performed a successful CAS, that is, the node was added to the list, then either it is still in the list or that it was deleted. Therefore, if p can find the new node in the list (using a procedure similar to find), or that it is marked, it must be that the node was added, and p can return true. Otherwise, p either crashed before performing the CAS, or that the CAS was unsuccessful. In both cases, the new node was not added, and p can restart the Insert procedure.

For the delete operation, once p logically delete a node v, it also tries to announce itself as the "removal" of the node by writing its id to deleter using CAS. Assume a process p crash while trying to delete the node. Upon recovery, if p sees the node is not marked, then obliviously its deletion did not took affect, and it can restart the delete operation. However, if the node is marked, it might be that p marked it before the crash, or it might be some other process trying to delete the same node did so. As p can not distinguish between the two, and since we desire for a lock-free implementation, we let p to try and complete the deletion, even if it is not to process to logically delete the node. To avoid a scenario in which more then a single process "delete" the same node, they all compete for deleter using CAS. The first one to perform it will win, and it is the only process to return true. It is easy to verify once a process writes to deleter, then eventually, if given enough time with no crash, it returns true, while any other process trying to delete the same node will have to retry the delete operation.

Shared variables: Node *head

```
Type Info {
    {Insert, Delete} optype
    Node *nd
    boolean result
}
```

Code for process p:

Procedure boolean Insert (T key)

```
18  Data:  Node *pred, *curr
           Node new := new Node (key)
19  Backup[p] := new Info (Insert, new, null)
20  while true do
21      ⟨pred, curr⟩ := lookup(key)
22      if curr.key = key then
23          Backup[p].result := false
24          return false                                   // key is already in the list
25      else
26          node.next := unmarked curr
27          if pred.next.CAS (unmarked curr, unmarked new) then
28              Backup[p].result := true
29              return true                                // new node has been inserted
30  end
```

Procedure boolean Delete (T key)

```
31  Data:  Node *pred, *curr, *succ
32  Backup[p] := new Info (Delete, null, null)
33  while true do
34      ⟨pred, curr⟩ := lookup(key)
35      if curr.key ≠ key then
36          Backup[p].result := false
37          return false                                   // key is not in the list
38      else
39          Backup[p].nd := curr
40          while curr.next is unmarked do                 // repeatedly attempt logical delete
41              succ := curr.next
42              curr.next.CAS (unmarked succ, marked succ)
43          end
44          pred.next.CAS (unmarked curr, unmarked succ)   // physical delete
45          curr.deleter.CAS (null, p)                     // announce yourself as deleter
46          Backup[p].result := (curr.deleter = p)
47          return (curr.deleter = p)
48  end
```

Procedure boolean Find (T key)

```
49  Data:  Node *curr := head
50  while curr.key < key do
51      curr = curr.next
52  end
53  return (curr.key = key && curr.next is unmarked)
```

Figure 1: Recoverable Non-Blocking Linked-List