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 node.next returns the reference, while a query can be used to identify if it is marked. Therefore, whenever writing to node.next, or performing CAS, both the reference and marking 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
46 if Backup[p].optype = Insert then
47     if Backup[p].new is in the list || Backup[p].curr.next is marked then
48         return true
49     else
50         return FAIL
51 if Backup[p].optype = Delete then
52     if Backup[p].curr.next is marked then
53         go to 38                                // try to complete the deletation
54     else
55         return FAIL
```

```
Procedure < Node, Node> Lookup(T key)
1  Data:  Node *pred, *curr, *succ
2  retry: while true do
3      pred := head
4      curr := head.next
5      while true do
6          succ := curr.next
7          if curr.next is marked 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. Moreover, in order to mark a node curr as logically deleted, a process needs to mark the field next in its predecessor in the list, pred, which points to curr. Thus, once pred.next is marked, only the first process to perform a CAS which tries and physically delete curr will succeed. This follows from the fact that the first CAS will swing pred.next to point to a different node then curr, and thus succeed. Any future change to pred.next, if it is done by an insert operation, then a new located node is used, or by a later delete operation.

which marks pred.next after the physical delete of curr, and thus no node in the list points to curr, and in particular it is not te successor of any node.

once a node curr is marked only the first process to perform a CAS which tries and physically delete curr will succeed.

This follows from the fact that no node can point to curr from the moment of the marking, except for pred, its predecessor at this point. Insert operation uses only new nodes, while any CAS which tries to change reference part of any next fields uses an and pred is pointing to curr. Then, only the first process to perform a CAS which tries and physically delete curr will succeed. This follows from the fact that once curr is marked, it will never be unmarked, and we always use new node when performing an insert operation. Therefore, once curr is physically deleted once, pred will never point to it again, and thus any subsequent CAS on pred.next with curr as first argument will return false.

Assume a process p performs

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 *pred, *curr, *new
}

Code for process p:

```
Procedure boolean Insert(T key)
Data:   Node *pred, *curr
        Node new := new Node (key)
while true do
    ⟨pred, curr⟩ := lookup(key)
    if curr.key = key then
        return false                                  // key is already in the list
    else
        node.next := unmarked curr
        Backup[p] := new Info (Insert, pred, curr, new)
        if pred.next.CAS (unmarked curr, unmarked new) then
            return true                               // new node has been inserted
end
```

```
Procedure boolean Delete(T key)
Data:   Node *pred, *curr, *succ
while true do
    ⟨pred, curr⟩ := lookup(key)
    if curr.key $\neq$ key then
        return false                                  // key is not in the list
    else
        succ := curr.next
        Backup[p] := new Info (Delete, pred, curr, null)
        if curr.next.CAS (unmarked succ, marked succ) then       // logical delete
            curr.deleter.CAS (null, p)                // announce yourself as deleter
            pred.next.CAS (unmarked curr, unmarked succ)         // physical delete
            return (curr.deleter = p)                 // originally return true
end
```

```
Procedure boolean Find(T key)
Data: Node* curr = head
while curr.key < key do
    curr = curr.next
end
return (curr.key == key && curr.next is unmarked)
```

Figure 1: Recoverable Non-Blocking Linked-List