1 Linked-List

The original algorithm by Harris is presented in Figure 1. Harris approach 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 initialised 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.

Procedure Lookup(int key)

```
   Data: Node* pred, curr, succ
 1 retry: while true do
 2 |   pred = head
 3 |   curr = head.next
 4 |   while true do
 5 |   |   succ = curr.next
 6 |   |   if curr.next is marked then
 7 |   |   |   if pred.next.CAS (unmarked curr,unmarked succ) == false then
 8 |   |   |   |   go to retry
 9 |   |   |   end
10 |   |   |   curr = succ
11 |   |   else
12 |   |   |   if curr.key ≥ key then
13 |   |   |   |   return <pred,curr>
14 |   |   |   end
15 |   |   |   pred = curr
16 |   |   |   curr = succ
17 |   |   end
18 |   end
19 end
```

Shared variables: Node* head

Procedure Insert(int key)

```
   Data: Node* pred, curr
         Node node = new Node (key)
20 while true do
21 |  <pred, curr> = lookup(key)
22 |  if curr.key == key then
23 |  |  return false
24 |  else
25 |  |  node.next = unmarked curr
26 |  |  if pred.next.CAS (unmarked curr, unmarked node) then
27 |  |  |  return true
28 |  |  end
29 |  end
30 end
```

Procedure Delete(int key)

```
   Data: Node* pred, curr, succ
31 while true do
32 |  <pred, curr> = lookup(key)
33 |  if curr.key != key then
34 |  |  return false
35 |  else
36 |  |  succ = curr.next
37 |  |  if curr.next.CAS (unmarked succ, marked succ) then
38 |  |  |  pred.next.CAS (unmarked curr, unmarked succ)
39 |  |  |  return true
40 |  |  end
41 |  end
42 end
```

Procedure Find(int key)

```
   Data: Node* curr = head
43 while curr.key < key do
44 |  curr = curr.next
45 end
46 return (curr.key == key && curr.next is unmarked)
```

Figure 1: Harris Non-Blocking Algorithm

1.0.1 Crash-Recovery

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 of the procedure return, that is, either when curr.key != key, or at the second condition test.

Following these linearization points (committing proof...), insert and delete operation are linearized at the point where they affect the system. That is, if there is a linearization point for insert operation, 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 and affect all other processes, or it did not affect the data structure at all.

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 at line 37 after performing a successful CAS. 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, as it is no longer part of the list.

1.0.2 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.

Each node is equipped with a new field named owner. 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 owner using CAS. This way, if a process fails during a delete, it can use owner in order to determine the response value. We assume owner is initialized to null when creating a new node.

Each process p has a designated location in the memory, Announce[p]. Before trying to apply an operation, p writes to Announce[p] the entire data needed to complete the operation. Upon recovery, p can read Announce[p], and based on it to complete its pending operation, in case there is such. Formally, Announce[p] contains a pointer to a structure containing all the relevant data.

We present the modified algorithm. Only the procedures which require changes are presented.

Shared variables: Node* head

Define operation: struct { *type*: OperationType, *pred*,*curr*, *new*: Node* }

Code for process p:

```
Procedure Insert(int key)
    Data: Node* pred, curr
          Node node = new Node (key)
47  while true do
48  |   <pred, curr> = lookup(key)
49  |   if curr.key == key then
50  |   |   return false
51  |   else
52  |   |   node.next = unmarked curr
53  |   |   Announce[p] = new operation (Insert, pred, curr, new)
54  |   |   if pred.next.CAS (unmarked curr, unmarked node) then
55  |   |   |   return true
56  |   |   end
57  |   end
58  end
```

```
Procedure Delete(int key)
    Data: Node* pred, curr, succ
59  while true do
60  |   <pred, curr> = lookup(key)
61  |   if curr.key != key then
62  |   |   return false
63  |   else
64  |   |   succ = curr.next
65  |   |   Announe[p] = new operation (Delete, pred, curr, null)
66  |   |   if curr.next.CAS (unmarked succ, marked succ) then
67  |   |   |   curr.owner.CAS (null, p)
68  |   |   |   pred.next.CAS (unmarked curr, unmarked succ)
69  |   |   |   if curr.owner == p then
70  |   |   |   |   return true
71  |   |   |   end
72  |   |   end
73  |   end
74  end
```

Figure 2: Recoverable Non-Blocking Linked-List

Procedure Recover

```
75 if Announce[p].type = Insert then
76 |  if Announce[p].new is in the list || Announce[p].curr.next is marked then
77 |  |  return true
78 |  else
79 |  |  go to 47                                                    // restart Insert
80 |  |
81 |  end
82 end
83 if Announce[p].type == Delete then
84 |  if Announce[p].curr.next is marked then
85 |  |  go to 67                                  // try to complete the deletation
86 |  |
87 |  else
88 |  |  go to 59                                                    // restart Delete
89 |  |
90 |  end
91 end
```

Correctness Argument

In the following, we give an intuition for the correctness of the algorithm.

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 Announce[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 owner 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 owner 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 owner, 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.