1 Linked-List

The original algorithm by Harris is presented in Figure 2. 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
retry: while true do
    pred = head
    curr = head.next
    while true do
        succ = curr.next
        if curr.next is marked then
            if pred.next.CAS (unmarked curr,unmarked succ) == false then
                go to retry
            end
            curr = succ
        else
            if curr.key ≥ key then
                return <pred,curr>
            end
            pred = curr
            curr = succ
        end
    end
end
```

Shared variables: Node* head

Procedure Insert(int key)

```
Data: Node* pred, curr
      Node node = new Node (key)
while true do
    <pred, curr> = lookup(key)
    if curr.key == key then
        return false
    else
        node.next = unmarked curr
        if pred.next.CAS (unmarked curr, unmarked node) then
            return true
        end
    end
end
```

Procedure Delete(int key)

```
Data: Node* pred, curr, succ
while true do
    <pred, curr> = lookup(key)
    if curr.key != key then
        return false
    else
        succ = curr.next
        if curr.next.CAS (unmarked succ, marked succ) then
            pred.next.CAS (unmarked curr, unmarked succ)
            return true
        end
    end
end
```

Procedure Find(int key)

```
Data: Node* curr = head
while curr.key < key do
    curr = curr.next
end
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 67 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 Harris Recoverable Version

To solve the problems mention above, we present a mechanise such that in case a process fails, upon recovery it is able to determine whether its last operation already took affect, and in such case to complete it and also return the right response. In case the operation did not took affect nor it will in any future run, p is allowed to ignore the operation.

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

Each node is equipped with a new field, owner, which keeps the identity of the deleting process. When deleting a node, process p first tries to set owner to p using CAS, and only then tries to mark the node. Since the deletion process is split, an helping mechanism is needed. If a process does not win the race for owner, it still tries to mark the node, helping the owner to complete its operation.

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.

Shared variables: Node* head

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

```
Procedure Insert(int key)
  Data: Node* pred, curr
        Node node = new Node (key)
  while true do
      <pred, curr> = lookup(key)
      if curr.key == key then
          return false
      else
          node.next = unmarked curr
          Announce[p] = new operation (Insert, pred, curr, new, false)
          if pred.next.CAS (unmarked curr, unmarked node) then
              Announce[p].done = true
              return true
          end
      end
  end
```

```
Procedure Delete(int key)
  Data: Node* pred, curr, succ
  while true do
      <pred, curr> = lookup(key)
      if curr.key != key then
          return false
      else
          succ = curr.next
          Announe[p] = new operation (Delete, pred, curr, null, false)
          if curr.next.CAS (unmarked succ, marked succ) then
              pred.next.CAS (unmarked curr, unmarked succ)
              return true
          end
      end
  end
```

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

Figure 2: Harris Non-Blocking Algorithm