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
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 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 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.

We present the modified algorithm. Only the procedures which require changes are presented. For simplicity, a process creates a new operation structure each time it writes to Backup, although a process can use two such structures alternately.

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 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.

Procedure Recover

```
if Backup[p].type = Insert then
    if Backup[p].new is in the list || Backup[p].curr.next is marked then
        return true
    else
        go to 47                                   // restart Insert
    end
end
if Backup[p].type == Delete then
    if Backup[p].curr.next is marked then
        go to 67                   // try to complete the deletation
    else
        go to 59                                   // restart Delete
    end
end
```

2 Elimination Stack

Procedure Exchange(⟨T,flag⟩ slot, T myitem, long timeout)

```
Data: long timeBound = getNanos() + timeout
while true do
    if getNanos() > timeBound then
        return TIMEOUT
    end
    ⟨youritem,state⟩ = slot
    switch state do
        case EMPTY do
            if slot.CAS (⟨youritem,EMPTY⟩, ⟨myitem, WAITING⟩) then
                while getNanos() < timeBound do
                    ⟨youritem,state⟩ = slot
                    if state == BUSY then
                        slot = ⟨null, EMPTY⟩
                        return youritem
                    end
                end
                if slot.CAS (⟨myitem,WAITING⟩, ⟨null,EMPTY⟩) then
                    return TIMEOUT
                else
                    ⟨youritem,state⟩ = slot
                    slot = ⟨null, EMPTY⟩
                    return youritem
                end
            end
            break
        end
        case WAITING do
            if slot.CAS (⟨youritem,state⟩, ⟨myitem,BUSY⟩) then
                return youritem
            end
            break
        end
        case BUSY do
            break
        end
    end
end
```

Procedure TryPush(Node newNode)

```
Node* oldTop = Top
newNode.next = oldTop
if Top.CAS (oldTop, newNode) then
    return true
else
    return false
end
```

Procedure TryPop(void)

```
Node *oldTop = Top
Node *newTop
if oldTop == NULL then
    return EMPTY
end
newTop = oldTop.next
if Top.CAS (oldTop, newTop) then
    return oldTop
end
return NULL
```

3 BST

Procedure push(T myitem)

```
Node *nd = new Node (myitem)
while true do
    if TryPush(nd) then
        return true
    end
    othervalue = visit(nd)
    if othervalue == NULL then
        Record Success ()
        return true
    else
        Record Failure ()
    end
end
```

Shared variables: Node* head

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

Code for process p:

```
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
          Backup[p] = new Info (Insert, pred, curr, new)
          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
          Announe[p] = new Info (Delete, pred, curr, null)
          if curr.next.CAS (unmarked succ, marked succ) then
              curr.deleter.CAS (null, p)
              pred.next.CAS (unmarked curr, unmarked succ)
              return (curr.deleter == p)
          end
      end
  end
```

Figure 2: Recoverable Non-Blocking Linked-List

type Update { // stored in one CAS word
    {CLEAN ,DFLAG ,IFLAG ,MARK } state
    Info **info*
}
type Internal { // subtype of Node
    $Key \cup \{\infty_1, \infty_2\}$ *key*
    Update *update*
    Node *$left$, *$right$
}
type Leaf { // subtype of Node
    $Key \cup \{\infty_1, \infty_2\}$ *key*
}
type IInfo { // subtype of Info
    Internal *p, *$newInternal$
    Leaf *l
}
type DInfo{ // subtype of Info
    Internal *gp, *p
    Leaf *l
    Update *pupdate*
}
$\triangleright$ Initialization:
shared Internal **Root* := pointer to new Internal node with *key* field ∞_2, *update* field $\langle CLEAN, \perp \rangle$, and pointers to new Leaf nodes with keys ∞_1 and ∞_2, respectively, as $left$ and $right$ fields.

Figure 3: Type definitions and initialization.