1 Elimination Stack

For simplicity, we assume a value $\perp$, which is different from NULL and any other value the stack can store. Since NULL is used as a legit return value, representing the value of POP operation (when exchanging values using the elimination array), NULL can not be used to represent an initialization value, different then any stack value. The same holds for a Node, since a NULL node represent an empty stack, the value $\perp$ is used to distinguish between initialization value and empty stack.

For simplicity, we split the RECOVER routine into sub-routines, based on which operation (PUSH, POP, EXCHANGE) is pending, or needs to be recover. This can be concluded easily by the type of record stored in *Announce*[*pid*] (ExInfo or OpInfo), thus there is no need to explicitly know where exactly in the code the crash took place. Also, the RECOVER routine returns FAIL in case the last pending operation did not took affect (no linearization point), nor it will take in any future run. In such case, the user has the option to either re-invoke the operation, or to skip it, depends on the needs and circumstances of the specific use of the data structure.

The given implementation ignores the log of failures and successes of the exchange routine when recovering. That is, in case of a crash during an EXCHANGE, a process is able to recover the EXCHANGE routine, however, the log of successes and failures is not update, since it might be the process already updated it. In addition, in case of a FAIL response, we do not know whether the time limit (timeout) was reached, or that the process simply crashed earlier in the routine without completing it. The given implementation can be expanded to also consider the log. Nonetheless, for ease of presentation we do not handle the log in case of a crash. Assuming crash events are rare, the log still gives a roughly good approximation to the number of failures and successes, thus our approach might be useful in practice.

1.1 A Lock-Free Exchanger

An exchanger object supports the EXCHANGE procedure, which allows exactly two processes to exchange values. If process A calls the EXCHANGE with argument a, and B calls the EXCHANGE of the same object with argument b, then A's call will return value b and vice versa.

On the original algorithm [cite the book?!], processes race to win the exchanger using a **CAS** primitive. A process accessing the exchanger first reads its content, and act according to the state of it. The first process observe an EMPTY state, and tries to atomically writes its value and change the state to WAITING. In such case, it spins and wait for the second process to arrive. The second, observing the state is now WAITING, tries to write its value and change the state to BUSY. This way, it informs the first one a successful collision took place. Once the first process notice the collision, it reads the other process value and release the exchanger by setting it back to EMPTY. In order to avoid an unbounded waiting, if a second process does not show up, the call eventually timeout, and the process release the exchanger and return.

Assume a process p successfuly capture the exchanger by setting its status to WAITING, followed by a crash. Now, some other process q complete the exchange by setting the exchanger to BUSY. Upon recovery, p can conclude some exchange was completed, but it can not tell whether its value is part of the exchange, and thus it can not complete the operation. Moreover, p and q must agree, otherwise q will return p's value, and thus the operation of p must be linearized together with q operation.

In order to avoid the above problem, we take an approach resembling the BST implementation. Instead of writing a value to the exchanger, processes will use an info record, containing the relevant

information for the exchange. This way, processes use the exchanger in order to exchange info records (more precisely, pointers to such records), and not values. To overcome the problematic scenario described earlier, if a process q observe the exchanger state is WAITING with some record *yourop*, it first update its own record *myop* it is about to try and collide with *yourop*, and only then performs the **CAS**. This way, if the collision is successful, the record *myop* which now stored in the exchanger implies which two records collide. Also, the fact that different processes uses different records guarantee that at most one record can collide with *yourop*.

Using records instead of values results some more positive implications. First, there is no need to store the exchanger's state in it (by using 2 bits of it to mark the state), but we can rather have this info in the record. Second, if there is a BUSY record in the exchanger, it contains the info of the two colliding records. Therefore, a third process, trying to also use the exchanger, can help the processes to complete the collision, and then can try and set the exchanger back to EMPTY, so it can use it again. In the original implementation, a process observaing a BUSY exchanger, have to wait for the first process to read the value and release the exchanger. Therefore, if the first process crash after the collision, the exchanger will be hold by it forever. The helping mechanism avoids this scenario, making the exchange routine non-blocking.

Notice that no exchange record with EMPTY state is ever created, except for the $default$ record. Therefore, reading EMPTY state is equivalent to the exchanger storing a pointer to $default$. Every process creates new record when accessing the exchanger, with a unique address. As long as a process fails to perform a successful **CAS**, and thus fails to store its pointer in *slot*, it is allowed to try again. However, once a process performs a successful **CAS** and stores *myop* in *slot*, the only other **CAS** it is allowed to do are in order to try and store $defualt$ in *slot*. Thus, if no crash occurs, *myop* can be written exactly once to *slot*. It follows that a collision can occur between two processes exactly - once a WAITING record stored in *slot*, only a single **CAS** can replace it with a BUSY record. As the two records can not be written again to *slot*, no other process can collide with any of the records.

Assume process p crash along the EXCHANGE routine. Upon recovery, in the EXCHANGE-RECOVER it first read $myop = Announce[pid]$. There are two scenarios to consider. If process p crash while $myop.state =$ BUSY, then p crashed while or after an attempt to collide with $myop.partner$. Notice that there was a point in time where *partner* was stored in *slot* with state WAITING, as these are the only values that can be written to $myop.partner$. if p did not succeed to collide with *partner*, then *myop* is never written to *slot*. In particular, no other process can read *myop*, or write to any of its fields. Thus, the if in lines 70 and 74 do not hold, and p will return FAIL. Otherwise, p performed a successful **CAS**, replacing *partner* with *myop*. The only way to write a new value to *slot* is by a **CAS** with *myop* as a parameter. This in turns, happens only after a process reads *myop*, and since its status is BUSY, a SWITCHPAIR routine is invoked with *myop* and *partner*. Thus, either p will read *myop* from *slot*, or that *myop* was replaced, but a SWITCHPAIR routine completed the collision, and $myop.result$ is updated. In both cases, p will return $myop.result$ as required.

Otherwise, p crash while $myop.state =$ WAITING. Thus, p may be after trying to perform a **CAS** to replace $defalut$ with *myop*.

```
Type Node {
    T value
    int popby
    Node *next
}

Type PushInfo {                    ▷ subtype of Info
    Node *pushnd
}

Type PopInfo {                     ▷ subtype of Info
    Node *popnd
}

Type ExInfo {                      ▷ subtype of Info
    {EMPTY, WAITING, BUSY} state
    T value, result
    ExInfo *partner, *slot
}
```

Figure 1: Type definition

ExInfo $default$ - global static ExInfo object with state = EMPTY

Algorithm 1: T EXCHANGE (ExInfo $*slot$, T $myitem$, long $timeout$)

```
1  long timeBound := getNanos() + timeout
2  ExInfo myop := new ExInfo(WAITING, myitem, ⊥, ⊥, slot)
3  Announce[pid] := myop
4  while true do
5      if getNanos() > timeBound then
6          myop.result := TIMEOUT                                  // time limit reached
7          return TIMEOUT
8      yourop := slot
9      switch yourop.state do
10         case EMPTY
11             myop.state := WAITING                               // attempt to replace default
12             myop.partner := ⊥
13             if slot.CAS(yourop, myop) then                      // try to collide
14                 while getNanos() < timeBound do
15                     yourop := slot
16                     if yourop ≠ myop then                       // a collision was done
17                         if youop.parnter = myop then            // yourop collide with myop
18                             SWITCHPAIR(myop, yourop)
19                             slot.CAS(yourop, default)           // release slot
20                         return myop.result
21                 end
                   // time limit reached and no process collide with me
22                 if slot.CAS(myop, default) then                 // try to release slot
23                     myop.result := TIMEOUT
24                     return TIMEOUT
25                 else                                            // some process show up
26                     yourop := slot
27                     if yourop.partner = myop then
28                         SWITCHPAIR(myop, yourop)                // complete the collision
29                         slot.CAS(yourop, default)               // release slot
30                     return myop.result
31             end
32             break
33         case WAITING                                            // some process is waiting in slot
34             myop.partner := yourop                              // attempt to replace yourop
35             myop.state := BUSY
36             if slot.CAS(yourop, myop) then                      // try to collide
37                 SWITCHPAIR(myop, yourop)                        // complete the collision
38                 slot.CAS(myop, default)                         // release slot
39                 return myop.result
40             break
41         case BUSY                                               // a collision in progress
42             SWITCHPAIR(yourop, yourop.parnter)                  // help to complete the collision
43             slot.CAS(yourop, default)                           // release slot
44             break
45     endsw
46 end
```

Algorithm 2: void SwitchPair(ExInfo *first*, ExInfo *second*)

```
   /* exchange the valus of the two operations                                    */
47 first.result := second.value
48 second.result := first.value
```

Algorithm 3: T Visit (T *value*, int *range*, long *duration*)

```
   /* invoke Exchange on a random entery in the collision array                   */
49 int cell := randomNumber(range)
50 return Exchange(exchanger[cell], value, duration)
```

Algorithm 4: T Exchange-Recover ()

```
51 ExInfo *myop := Announce[pid]                          // read your last operation record
52 ExInfo *slot := myop.slot                               // and the slot on which it acts
53 if myop.state = Waiting then
   |  /* crash while trying to exchange defualt, or waiting for a collision       */
54 |  yourop := slot
55 |  if yourop = myop then                                // still waiting for a collision
56 |  |  if slot.CAS(myop, default) then                          // try to release slot
57 |  |  |  return Fail
58 |  |  else                                                  // some process show up
59 |  |  |  yourop := slot
60 |  |  |  if yourop.partner = myop then
61 |  |  |  |  SwitchPair(myop, yourop)                    // complete the collision
62 |  |  |  |  slot.CAS(yourop, default)                              // release slot
63 |  |  |  return myop.result
64 |  else if yourop.partner = myop then                  // yourop collide with myop
65 |  |  SwitchPair(myop, yourop)                          // complete the collision
66 |  |  slot.CAS(yourop, default)                                  // release slot
67 |  |  return myop.result
68 if myop.state = Busy then
   |  /* crash while trying to collide with myop.partner                          */
69 |  yourop := slot
70 |  if yourop = myop  then                  // collide was successful and in progress
71 |  |  SwitchPair(myop, myop.partner)                    // complete the collision
72 |  |  slot.CAS(myop, default)                                    // release slot
73 |  |  return myop.result
74 if myop.result ≠ ⊥  then
75 |  return myop.result                              // collide was successfuly completed
76 else
77 |  return Fail
```

Figure 2: Elimination Array routines

Algorithm 5: boolean TryPush (Node $*nd$)

```
   /* attempt to perform Push to the central stack                    */
78 Node *oldtop := Top
79 nd.next := oldtop
80 if Top.CAS(oldtop, nd) then          // try to declare nd as the new Head
81 |  nd.popby.CAS(⊥, Null)                    // announce nd is in the stack
82 |  return true
83 return false
```

Algorithm 6: boolean Push (T *myitem*)

```
84 Node *nd = new Node (myitem)
85 nd.popby := ⊥
86 PushInfo data := new PushInfo (nd)
87 while true do
88 |  Announce[pid] := data                  // declare - trying to push node nd
89 |  if TryPush(nd) then                  // if central stack Push is successful
90 |  |  return true
91 |  range := CalculateRange()             // get parameters for collision array
92 |  duration := CalculateDuration()
93 |  othervalue := Visit(myitem, range, duration)                  // try to collide
94 |  if othervalue = Null then          // successfuly collide with Pop operation
95 |  |  RecordSuccess ()
96 |  |  return true
97 |  else if othervalue = Timeout then                          // failed to collide
98 |  |  RecordFailure ()
99 end
```

Algorithm 7: boolean Push-Roceover ()

```
100 Node *nd := Announce[pid].pushnd
101 if nd.popby ≠ ⊥ then                  // nd was announced to be in the stack
102 |  return true
103 if Search(nd) || nd.popby ≠ ⊥ then   // nd in the stack, or was announced as such
104 |  nd.popby.CAS(⊥, Null)                    // announce nd is in the stack
105 |  return true
106 return Fail
```

Algorithm 8: boolean Search (Node $*nd$)

```
   /* search for node nd in the stack                                 */
107 Node *iter := Top
108 while iter ≠ ⊥ do
109 |  if iter = nd then
110 |  |  return true
111 |  iter := iter.next
112 end
113 return false
```

Figure 3: Push routine

Algorithm 9: T TryPop()

```
114 Node *oldtop := Top
115 Node *newtop
116 Announce[pid].popnd := oldnop                  // declare - trying to pop node oldtop
117 if oldtop = ⊥ then                                              // stack is empty
118 |   return Empty
119 newtop := oldtop.next
120 oldtop.popby.CAS(⊥, Null)                        // announce oldtop is in the stack
121 if Top.CAS(oldtop, newtop) then    // try to pop oldtop by changing Top to newtop
122 |   if newtop.popby.CAS(Null, pid) then     // try to announce yourself as winner
123 |   |   return oldtop.value
124 else
125 |   return ⊥
```

Algorithm 10: T Pop ()

```
126 Node *result
127 PopInfo data := new PopInfo (Top)
128 while true do
129 |   Announce[pid] := data                      // declare - trying to perform Pop
130 |   result := TryPop()                        // attempt to pop from central stack
131 |   if result ≠ ⊥ then                      // if central stach Pop is successful
132 |   |   return result
133 |   range := CalculateRange()                // get parameters for collision array
134 |   duration := CalculateDuration()
135 |   othervalue := Visit(Null, range, duration)                    // try to collide
136 |   if othervalue = Timeout then                              // failed to collide
137 |   |   RecordFailure ()
138 |   else if othervalue ≠ Null then     // successfuly collide with Push operation
139 |   |   RecordSuccess ()
140 |   |   return othervalue
141 end
```

Algorithm 11: T Pop-Recover()

```
142 Node *nd := Announce[pid].popnd              // crash while trying to pop node nd
143 if nd = ⊥ then                                          // pop from an empty stack
144 |   return Empty
145 if Search(nd) then                          // nd was not removed from the stack
146 |   return Fail
147 nd.popby.CAS(Null, pid)        // nd was removed.  Try to complete the operation
148 if nd.popby = pid then                 // you are the process to win the pop of nd
149 |   return nd.value
150 return Fail
```

Figure 4: Pop routine