

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 As call will return value b and vice versa. On a high level, the exchanger works by having the first thread arrive to write its value, and spin until a second arrives. The second then detects that the first is waiting, reads its value, and signals the exchange. They each have now read the others value, and can return. The first threads call may timeout if the second does not show up, allowing it to proceed and leave the exchanger, if it is unable to exchange a value within a reasonable duration.

```

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,  $\perp$ ,  $\perp$ , 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 :=  $\perp$ 
13      | if slot.CAS(yourop, myop) then // try to collide
14        | while getNanos() < timeBound do
15          | | yourop := slot
16          | | if yourop  $\neq$  myop then // a collision was done
17            | | | if yourop.parnter = myop then // yourop collide with myop
18              | | | SWITCHPAIR(myop, yourop)
19              | | | slot.CAS(yourop, default) // release slot
20            | | return myop.result
21          | end
22          | // time limit reached and no process collide with me
23          | if slot.CAS(myop, default) then // try to release slot
24            | | myop.result := TIMEOUT
25            | | return TIMEOUT
26          | else // some process show up
27            | | yourop := slot
28            | | if yourop.partner = myop then
29              | | | SWITCHPAIR(myop, yourop) // complete the collision
30              | | | slot.CAS(yourop, default) // release slot
31            | | return myop.result
32          | end
33        | break
34    case WAITING // some process is waiting in slot
35      | myop.partner := yourop // attempt to replace yourop
36      | myop.state := BUSY
37      | if slot.CAS(yourop, myop) then // try to collide
38        | | SWITCHPAIR(myop, yourop) // complete the collision
39        | | slot.CAS(myop, default) // release slot
40        | | return myop.result
41      | break
42    case BUSY // a collision in progress
43      | SWITCHPAIR(yourop, yourop.parnter) // help to complete the collision
44      | slot.CAS(yourop, default) // release slot
45      | break
46  endsw
47 end

```

Algorithm 2: void SWITCHPAIR(ExInfo <i>first</i> , ExInfo <i>second</i>)	
/* exchange the value of the two operations	*/
47 <i>first.result</i> := <i>second.value</i>	
48 <i>second.result</i> := <i>first.value</i>	
<hr/>	
Algorithm 3: T VISIT (T <i>value</i> , int <i>range</i> , long <i>duration</i>)	
/* invoke EXCHANGE on a random entry in the collision array	*/
49 int <i>cell</i> := randomNumber(<i>range</i>)	
50 return EXCHANGE(<i>exchanger</i> [<i>cell</i>], <i>value</i> , <i>duration</i>)	
<hr/>	
Algorithm 4: T EXCHANGE-RECOVER ()	
51 ExInfo * <i>myop</i> := Announce[<i>pid</i>]	// read your last operation record
52 ExInfo * <i>slot</i> := <i>myop.slot</i>	// and the slot on which it acts
53 if <i>myop.state</i> = WAITING then	
/* crash while trying to exchange <i>default</i> , or waiting for a collision	*/
54 <i>yourop</i> := <i>slot</i>	
55 if <i>yourop</i> = <i>myop</i> then	// still waiting for a collision
56 if <i>slot.CAS</i> (<i>myop</i> , <i>default</i>) then	// try to release slot
57 return FAIL	
58 else	// some process show up
59 <i>yourop</i> := <i>slot</i>	
60 if <i>yourop.partner</i> = <i>myop</i> then	
61 SWITCHPAIR(<i>myop</i> , <i>yourop</i>)	// complete the collision
62 <i>slot.CAS</i> (<i>yourop</i> , <i>default</i>)	// release slot
63 return <i>myop.result</i>	
64 else if <i>yourop.partner</i> = <i>myop</i> then	// <i>yourop</i> collide with <i>myop</i>
65 SWITCHPAIR(<i>myop</i> , <i>yourop</i>)	// complete the collision
66 <i>slot.CAS</i> (<i>yourop</i> , <i>default</i>)	// release slot
67 return <i>myop.result</i>	
68 if <i>myop.state</i> = BUSY then	
/* crash while trying to collide with <i>myop.partner</i>	*/
69 <i>yourop</i> := <i>slot</i>	
70 if <i>yourop</i> = <i>myop</i> then	// collide was successful and in progress
71 SWITCHPAIR(<i>myop</i> , <i>myop.partner</i>)	// complete the collision
72 <i>slot.CAS</i> (<i>myop</i> , <i>default</i>)	// release slot
73 return <i>myop.result</i>	
74 if <i>myop.result</i> ≠ ⊥ then	
75 return <i>myop.result</i>	// collide was successfully completed
76 else	
77 return FAIL	

Figure 2: Elimination Array routines

Algorithm 5: boolean TRY PUSH (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( $\perp$ , 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 :=  $\perp$ 
86 PushInfo data := new PushInfo (nd)
87 while true do
88   | Announce[pid] := data           // declare - trying to push node nd
89   | if TRY PUSH(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       // successfully 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  $\neq \perp$  then           // nd was announced to be in the stack
102   | return true
103 if SEARCH(nd) || nd.popby  $\neq \perp$  then // nd in the stack, or was announced as such
104   | nd.popby.CAS( $\perp$ , 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  $\neq \perp$  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 =  $\perp$  then                     // stack is empty
118 |   return EMPTY
119 newtop := oldtop.next
120 oldtop.popby.CAS( $\perp$ , 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  $\perp$ 
```

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  $\neq \perp$  then               // if central stack 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  $\neq$  NULL then      // successfully 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 =  $\perp$  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