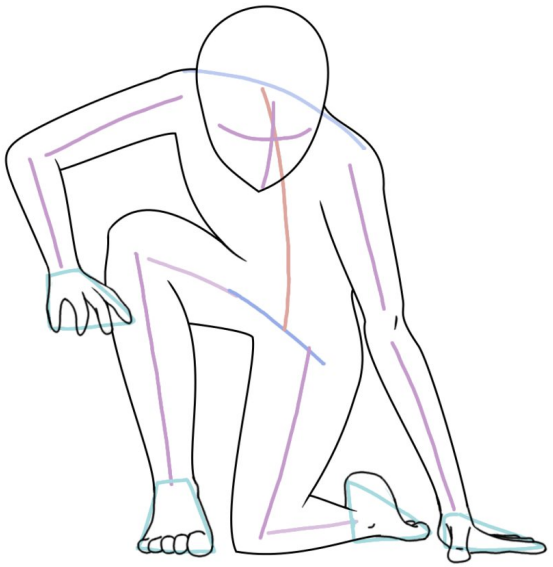
Ben-Gurion University of the Negev
אוניברסיטת בן-גוריון בנגב

Recoverable Algorithms for Non-Volatile Main Memory: Exercises and Solutions

Danny Hendler
Ben-Gurion University



Exercise #1

Shared C initially <null, init>, R[N][N] initially *null*

```
1.    proc CAS(old,new)
2.      <j,val>:=C.read()
3.      if val ≠ old
4.        return false
5.      if j ≠ null
6.        R[j][p] := val
7.      ret:=C.cas(<j,val>,<p,new>
8.      return ret

9.    proc READ()
10.     <j,val>:=C
11.     return val
```

```
12.   proc CAS.RECOVER(old,new)
13.     if C=<p,new> ∨
          new ∈ {R[p][1],…,R[p][N]}
14.       return true
15.     else
          proceed from line 2

16.   proc READ.RECOVER()
17.     <j,val>:=C
18.     return val
```

Does the order in which the two parts of the conjunction in the condition of line 13 matter for linearizability?

Exercise #1: solution

Yes, the order matters, the left side of the condition must be evaluated before the right side. Assume otherwise. Consider the following scenario, where we use distinct values v1,v2 (both ≠ init).

1. Process p1 performs a CAS(init, v1) operation, its cas operation in line 7 succeeds but then p1 fails.
2. Process p1 starts to recover and first evaluates the second part of the expression in line 12, which returns false.
3. Process p2 performs a CAS(v1,v2) operation to completion, so now C=<p2,v2> holds.
4. Process p1 next evaluates the first part of the expression in line 12 which returns false as well, so it attempts to re-execute the CAS and returns false in line 8.

From an initial state of init, we had a CAS(init,v1) that fails and a CAS(v1,v2) which succeeds. Since init, v1, v2 are distinct values, this is a non-linearizable execution.

Exercise #2

Exit for $p_{i \in \{0,\ldots,n-1\}}$

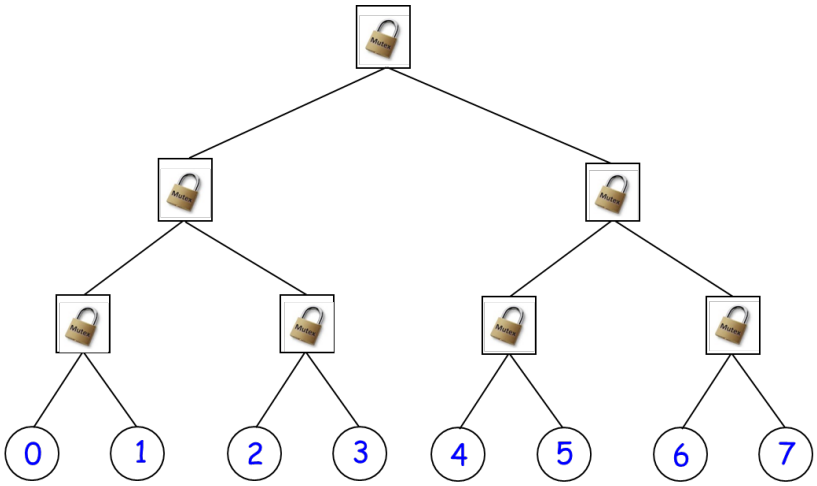
```
14.  node := root
15.  repeat {
16.      if i'th leaf on node's left subtree
17.          side := left
18.      else
19.          side := right
20.      node.exit(side)
21.      node := node's child on
                 path to the leaf }
22.  until node is a leaf
```

Recover for $p_{i \in \{0,\ldots,n-1\}}$

(empty)

Shared variables:

A complete binary tree, where each node is an instance of the 2-process recoverable lock



Is Golab and Ramaraju's n-process algorithm still correct if locks are released bottom-up instead of top-down in the exit section?

Exercise #2: solution

No, it is not. Consider the following scenario in a tree with 8 leaves (of processes p0…p7) as shown in the previous slide.

1. Process p0 enters the critical section
2. Process p3 climbs up and waits for the lock of the root's left child in line 11 (slide 44 of my presentation)
3. Process p0 exits, starts by releasing the locks bottom up, so it performs line 14-15 (slide 44) on the root's left child and now p3 is able to climb to the root lock and enters the CS
4. Process p0 releases the root lock by performing lines 14-15 (slide 44) thus finishing its exit code
5. Process p7 now executes its entry code and is able to enter the CS together with p3, a violation of mutual exclusion.

Exercise #3

Extend the recoverable compare-and-swap algorithm to support write operations, in addition to read and CAS.

You may use an *mswap* operation that atomically swaps the values of two shared-memory variables, otherwise this exercise may require a lot of work…

We still assume that all values written by a process (either using write or CAS) are unique. We also assume that evaluation of disjunctive expressions is done in order (see exercise #1).

Exercise #3: solution, code for process p

Shared C initially <null, init>, R[N][N] initially *null,* W[N] initially <null,null>

```
1.    proc CAS(old,new)
2.      <j,val>:=C.read()
3.      if val ≠ old
4.        return false
5.      if j ≠ null
6.        R[j][p] := val
7.      ret:=C.cas(<j,val>,<p,new>)
8.      return ret

9.    proc READ()
10.     <j,val>:=C
11.     return val

12.   proc WRITE(new)
13.     W[p]:=<p,new>
14.     mswap(C,W[p])
15.     <j,val>:=W[p]
16.     if (j ≠ null)
17.       R[j,p]:=val
18.     return
```

```
19.   proc CAS.RECOVER(old,new)
20.     if C=<p,new> ∨
          <p,new>∈ {W[1],...,W[N]} ∨
          new ∈ {R[p][1],...,R[p][N]}
21.       return true
22.     else
          proceed from line 2

23.   proc READ.RECOVER()
24.     <j,val>:=C
25.     return val

26.   proc WRITE.RECOVER(new)
27.     if C=<p,new> ∨ W[p]=<p,new> ∨
          new ∈ {R[p][1],...,R[p][N]}
28.       return
29.     else
30.       proceed from line 13
```