Recoverable FCFS Mutual Exclusion with Wait-Free Recovery

Distributed Computing Seminar

Mickey Vänskä

2018.04.17

Mutual Exclusion

Mutual Exclusion

Goal: Share a resource in a system while preventing **race conditions**.

System

- n processes $(1, 2, \ldots, n)$
- Atomic shared variables (*write*, *read*, *fetch&update*)

Mutual Exclusion

Process Sections

- Remainder (REM)

Mutual Exclusion

Process Sections

- Remainder (REM)
- Critical Section (CS)

Mutual Exclusion

Process Sections

- Remainder (REM)
- Try (TRY)
- Critical Section (CS)
- Exit (EXIT)

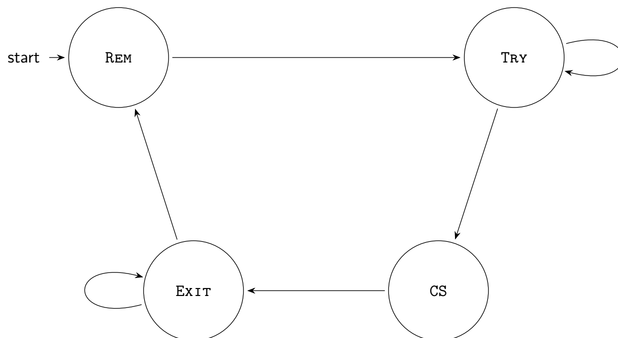
Mutual Exclusion

Try & Exit Properties

- MUTUAL EXCLUSION: At most one process is in CS at any time
- BOUNDED EXIT: Every process completes EXIT in a bounded number of its steps
- STARVATION FREEDOM: If a process is in TRY, it eventually enters CS (under the assumptions that every process that enters CS eventually leaves it, and no process permanently stops taking steps while in TRY or EXIT).

Process Step

Assumption: Performs single operation on single shared variable and state changed according to return value of shared variable.



Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

1. $wait[p] \leftarrow true$
2. `inc` $(Counter, 1)$
3. $v \leftarrow$ `read`$(Counter)$
4. `insert`$(ProcQueue, [p,v])$
5. `promote()`
6. **while** $wait[p] = true$ **do** wait
7. `delete`$(ProcQueue)$
8. `CriticalSection()`
9. $csbusy \leftarrow false$
10. `promote()`

Procedure promote

1. **if** `LL`$(csbusy) = true$ **then return**
2. $e \leftarrow$ `findmin`$(ProcQueue)$
3. **if** $e \neq \infty$ **then**
4.   **if** `SC`$(csbusy, true)$ **then**
5. $wait[e_{pid}] \leftarrow false$

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1  if LL(csbusy) = true then return
2  e ← findmin(ProcQueue)
3  if e ≠ ∞ then
4      if SC(csbusy, true) then
5          wait[e_pid] ← false
```

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1  if LL(csbusy) = true then return
2  e ← findmin(ProcQueue)
3  if e ≠ ∞ then
4      if SC(csbusy, true) then
5          wait[e_pid] ← false
```

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1  if LL(csbusy) = true then return
2  e ← findmin(ProcQueue)
3  if e ≠ ∞ then
4  |   if SC(csbusy, true) then
5  |   |   wait[e_pid] ← false
```

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1  if LL(csbusy) = true then return
2  e ← findmin(ProcQueue)
3  if e ≠ ∞ then
4     if SC(csbusy, true) then
5        wait[e_pid] ← false
```

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1 if LL(csbusy) = true then return
2 e ← findmin(ProcQueue)
3 if e ≠ ∞ then
4 |   if SC(csbusy, true) then
5 |   |   wait[e_pid] ← false
```

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

1. $wait[p] \leftarrow true$
2. `inc` $(Counter, 1)$
3. $v \leftarrow$ `read`$(Counter)$
4. `insert`$(ProcQueue, [p,v])$
5. `promote()`
6. **while** $wait[p] = true$ **do** wait
7. `delete`$(ProcQueue)$
8. `CriticalSection()`
9. $csbusy \leftarrow false$
10. `promote()`

Procedure promote

1. **if** `LL`$(csbusy) = true$ **then return**
2. $e \leftarrow$ `findmin`$(ProcQueue)$
3. **if** $e \neq \infty$ **then**
4.   **if** `SC`$(csbusy, true)$ **then**
5. $wait[e_{pid}] \leftarrow false$

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1  if LL(csbusy) = true then return
2  e ← findmin(ProcQueue)
3  if e ≠ ∞ then
4      if SC(csbusy, true) then
5          wait[e_pid] ← false
```

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1  if LL(csbusy) = true then return
2  e ← findmin(ProcQueue)
3  if e ≠ ∞ then
4     if SC(csbusy, true) then
5        wait[e_pid] ← false
```

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1  if LL(csbusy) = true then return
2  e ← findmin(ProcQueue)
3  if e ≠ ∞ then
4      if SC(csbusy, true) then
5          wait[e_pid] ← false
```

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1  if LL(csbusy) = true then return
2  e ← findmin(ProcQueue)
3  if e ≠ ∞ then
4     if SC(csbusy, true) then
5        wait[e_pid] ← false
```

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1  if LL(csbusy) = true then return
2  e ← findmin(ProcQueue)
3  if e ≠ ∞ then
4  |   if SC(csbusy, true) then
5  |   |   wait[e_pid] ← false
```

Example Algorithm

Algorithm 1: Adaptive FCFS Mutual Exclusion

1. $wait[p] \leftarrow true$
2. `inc` ($Counter$, 1)
3. $v \leftarrow$ `read`($Counter$)
4. `insert`($ProcQueue$, $[p,v]$)
5. `promote()`
6. **while** $wait[p] = true$ **do** wait
7. `delete`($ProcQueue$)
8. `CriticalSection()`
9. $csbusy \leftarrow false$
10. `promote()`

Procedure promote

1. **if** `LL`($csbusy$) $= true$ **then return**
2. $e \leftarrow$ `findmin`($ProcQueue$)
3. **if** $e \neq \infty$ **then**
4.   **if** `SC`($csbusy$, $true$) **then**
5. $wait[e_{pid}] \leftarrow false$

Wait-Free

Definition

An implementation is wait-free if every process completes its operation on the implemented object in a bounded number of its steps, regardless of whether other processes are slow, fast or have crashed.

Example Algorithm

Algorithm 2: Adaptive FCFS Mutual Exclusion

```
1  wait[p] ← true
2  inc (Counter, 1)
3  v ← read(Counter)
4  insert(ProcQueue, [p,v])
5  promote()
6  while wait[p] = true do wait
7  delete(ProcQueue)
8  CriticalSection()
9  csbusy ← false
10 promote()
```

Procedure promote

```
1  if LL(csbusy) = true then return
2  e ← findmin(ProcQueue)
3  if e ≠ ∞ then
4  |   if SC(csbusy, true) then
5  |   |   wait[e_pid] ← false
```

Recoverability

Concepts

Non-trivial extension to a mutex

Process crash $\rightarrow$ loss of all state

A problem has been detected and windows has been shut down to prevent damage
to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBFE7617,0x00000000)

*** SPCMDCON.SYS - Address FBFE7617 base at FBFE5000, DateStamp 3d6dd67c

Concepts

Global Reset

Idea: Set all processes back to REM, set all variables to defaults

2 Problems:

- Manipulation of data structures problematic
- Local failure causes global reset $\rightarrow$ Not wait-free

Concepts

Using NVRAM

Idea: Each process stores complete state in NVRAM and recovers state after crash

- Sounds like a good idea ...
- ...except that this fails. Why?

Concepts

Using NVRAM

Idea: Each process stores complete state in NVRAM and recovers state after crash

- Sounds like a good idea ...
- ...except that this fails. Why?
- When storing and recovering the PC, cannot tell if error happened before increment of PC or after increment of PC.

Concepts

Using NVRAM

Idea: Each process stores complete state in NVRAM and recovers state after crash

- Sounds like a good idea ...
- ...except that this fails. Why?
- When storing and recovering the PC, cannot tell if error happened before increment of PC or after increment of PC.

We need a smarter algorithm! Maybe only back up some and not all.

Recoverable Mutex Algorithms

Design

Adapting existing (non-recoverable) mutual exclusion algorithms.

Recoverable Mutex Algorithms

Properties

- Mutual Exclusion
- Bounded Exit
- Starvation Freedom
- **Well-Formedness**
- **Critical Section Reentry**

Recoverable Mutex Algorithms

System

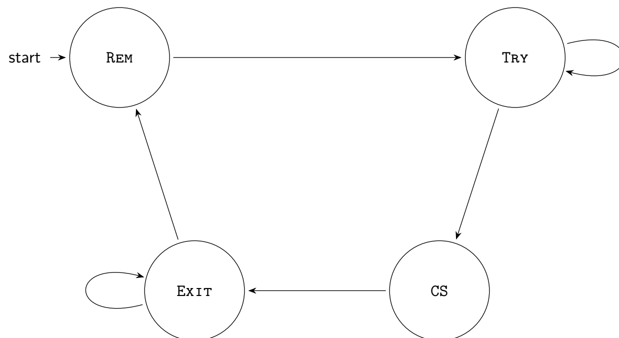
- n processes $(1, 2, \ldots, n)$
- Atomic shared variables (*write*, *read*, *fetch&update*)

Process p

- Remainder (REM)
- **Recover** (REC)
- Try (TRY)
- Critical Section (CS)
- Exit (EXIT)

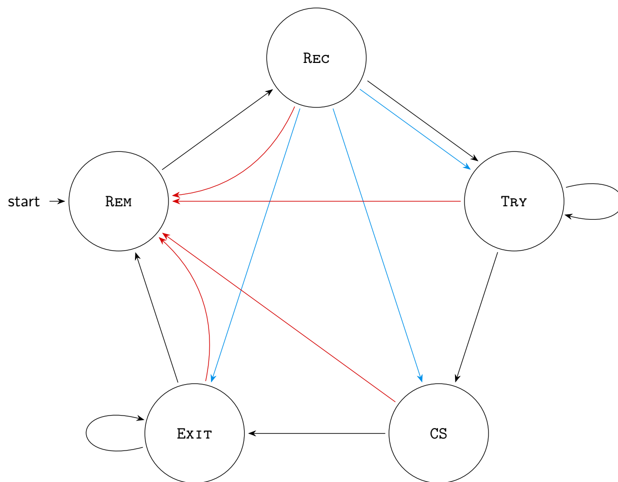
Process Step

Assumption: Performs single operation on single shared variable and state changed according to return value of shared variable.



Process Step

Assumption: Performs single operation on single shared variable and state changed according to return value of shared variable.



Recoverable FCFS Mutual Exclusion with Wait-Free Recovery

Novelties

- Intelligent recovery from `Rec`.
- Bounded reentry of `CS`.
- Based in `CAS`.
- Constant RMR complexity per crash (ξ crashes $\rightarrow O(\xi + log(n))$).

Novelties

- Intelligent recovery from REC.
- Bounded reentry of CS.
- Based in CAS.
- Constant RMR complexity per crash (ξ crashes $\rightarrow O(\xi + log(n))$).

Individual Sections

Procedure Try

```
1  WAIT[p] ← true
2  ATTEMPT[p] ← true
3  t_p ← TOKEN
4  CAS(TOKEN, t_p, t_p + 1)
5  MYTOKEN[p] ← t_p
6  RECQUEUE.write(p, (t_p, p))
7  if (i_p, j_p) ← CSOWNER ∧ (i_p = ⊥) then
8      if RECQUEUE.findmin() = (t_p, p) then
9          if CAS(CSOWNER, (⊥, j_p), (p, p)) then
10             WAIT[p] ← false
11 while WAIT[p] = true do WAIT[]
```

Individual Sections

Procedure Try

```
1  WAIT[p] ← true
2  ATTEMPT[p] ← true
3  t_p ← TOKEN
4  CAS(TOKEN, t_p, t_p + 1)
5  MYTOKEN[p] ← t_p
6  RECQUEUE.write(p, (t_p, p))
7  if (i_p, j_p) ← CSOWNER ∧ (i_p = ⊥) then
8      if RECQUEUE.findmin() = (t_p, p) then
9          if CAS(CSOWNER, (⊥, j_p), (p, p)) then
10             WAIT[p] ← false
11 while WAIT[p] = true do WAIT[]
```

Individual Sections

Procedure Try

```
1  WAIT[p] ← true
2  ATTEMPT[p] ← true
3  t_p ← TOKEN
4  CAS(TOKEN, t_p, t_p + 1)
5  MYTOKEN[p] ← t_p
6  RECQUEUE.write(p, (t_p, p))
7  if (i_p, j_p) ← CSOWNER ∧ (i_p = ⊥) then
8  |  if RECQUEUE.findmin() = (t_p, p) then
9  |  |  if CAS(CSOWNER, (⊥, j_p), (p, p)) then
10 |  |  |  WAIT[p] ← false
11 while WAIT[p] = true do WAIT[]
```

Individual Sections

Procedure Try

```
1  WAIT[p] ← true
2  ATTEMPT[p] ← true
3  t_p ← TOKEN
4  CAS(TOKEN, t_p, t_p + 1)
5  MYTOKEN[p] ← t_p
6  RECQUEUE.write(p, (t_p, p))
7  if (i_p, j_p) ← CSOWNER ∧ (i_p = ⊥) then
8  |   if RECQUEUE.findmin() = (t_p, p) then
9  |   |   if CAS(CSOWNER, (⊥, j_p), (p, p)) then
10 |   |   |   WAIT[p] ← false
11 while WAIT[p] = true do WAIT[]
```

Individual Sections

Procedure Try

```
1  WAIT[p] ← true
2  ATTEMPT[p] ← true
3  t_p ← TOKEN
4  CAS(TOKEN, t_p, t_p + 1)
5  MYTOKEN[p] ← t_p
6  RECQUEUE.write(p, (t_p, p))
7  if (i_p, j_p) ← CSOWNER ∧ (i_p = ⊥) then
8      if RECQUEUE.findmin() = (t_p, p) then
9          if CAS(CSOWNER, (⊥, j_p), (p, p)) then
10             WAIT[p] ← false
11 while WAIT[p] = true do WAIT[]
```

Individual Sections

Procedure Try

```
1  WAIT[p] ← true
2  ATTEMPT[p] ← true
3  t_p ← TOKEN
4  CAS(TOKEN, t_p, t_p + 1)
5  MYTOKEN[p] ← t_p
6  RECQUEUE.write(p, (t_p, p))
7  if (i_p, j_p) ← CSOWNER ∧ (i_p = ⊥) then
8  |   if RECQUEUE.findmin() = (t_p, p) then
9  |   |   if CAS(CSOWNER, (⊥, j_p), (p, p)) then
10 |   |   |   WAIT[p] ← false
11 while WAIT[p] = true do WAIT[]
```

Individual Sections

Procedure Try

```
1  WAIT[p] ← true
2  ATTEMPT[p] ← true
3  t_p ← TOKEN
4  CAS(TOKEN, t_p, t_p + 1)
5  MYTOKEN[p] ← t_p
6  RECQUEUE.write(p, (t_p, p))
7  if (i_p, j_p) ← CSOWNER ∧ (i_p = ⊥) then
8      if RECQUEUE.findmin() = (t_p, p) then
9          if CAS(CSOWNER, (⊥, j_p), (p, p)) then
10             WAIT[p] ← false
11 while WAIT[p] = true do WAIT[]
```

Individual Sections

Procedure Exit

```
1 MyToken[p] ← ∞
2 RecQueue.write(p, (∞, ∞))
3 CSOwner ← (⊥, p)
4 if (–, α_p) ← RecQueue.findmin() ∧ (α_p ≠ ∞) then
5 │  if CAS(CSOwner, (⊥, p), (α_p, p)) then
6 │  │  Wait[α_p] ← false
7 Attempt[p] ← false
```

Individual Sections

Procedure Exit

```
1 MyToken[p] ← ∞
2 RecQueue.write(p, (∞, ∞))
3 CSOwner ← (⊥, p)
4 if (−, α_p) ← RecQueue.findmin() ∧ (α_p ≠ ∞) then
5 |  if CAS(CSOwner, (⊥, p), (α_p, p)) then
6 |  |  Wait[α_p] ← false
7 Attempt[p] ← false
```

Individual Sections

Procedure Exit

1. $\textsc{MyToken}[p] \leftarrow \infty$
2. $\textsc{RecQueue}.\texttt{write}(p, (\infty, \infty))$
3. $\textsc{CSOwner} \leftarrow (\bot, p)$
4. **if** $(-, \alpha_p) \leftarrow \textsc{RecQueue}.\texttt{findmin}() \wedge (\alpha_p \neq \infty)$ **then**
5.   **if** $\texttt{CAS}(\textsc{CSOwner}, (\bot, p), (\alpha_p, p))$ **then**
6. $\textsc{Wait}[\alpha_p] \leftarrow \mathit{false}$
7. $\textsc{Attempt}[p] \leftarrow \mathit{false}$

Individual Sections

Procedure Exit

```
1 MyToken[p] ← ∞
2 RecQueue.write(p, (∞, ∞))
3 CSOwner ← (⊥, p)
4 if (–, α_p) ← RecQueue.findmin() ∧ (α_p ≠ ∞) then
5 |   if CAS(CSOwner, (⊥, p), (α_p, p)) then
6 |   |   Wait[α_p] ← false
7 Attempt[p] ← false
```

Individual Sections

Procedure Exit

1 $\textsc{MyToken}[p] \leftarrow \infty$
2 $\textsc{RecQueue}.\texttt{write}(p, (\infty, \infty))$
3 $\textsc{CSOwner} \leftarrow (\bot, p)$
4 **if** $(-, \alpha_p) \leftarrow \textsc{RecQueue}.\texttt{findmin}() \wedge (\alpha_p \neq \infty)$ **then**
5   **if** $\texttt{CAS}(\textsc{CSOwner}, (\bot, p), (\alpha_p, p))$ **then**
6         $\textsc{Wait}[\alpha_p] \leftarrow \mathit{false}$
7 $\textsc{Attempt}[p] \leftarrow \mathit{false}$

Individual Sections

Procedure Exit

```
1 MyToken[p] ← ∞
2 RecQueue.write(p, (∞, ∞))
3 CSOwner ← (⊥, p)
4 if (–, α_p) ← RecQueue.findmin() ∧ (α_p ≠ ∞) then
5 |   if CAS(CSOwner, (⊥, p), (α_p, p)) then
6 |   |   Wait[α_p] ← false
7 Attempt[p] ← false
```

Individual Sections

Procedure Exit

```
1 MyToken[p] ← ∞
2 RecQueue.write(p, (∞, ∞))
3 CSOwner ← (⊥, p)
4 if (–, α_p) ← RecQueue.findmin() ∧ (α_p ≠ ∞) then
5 |   if CAS(CSOwner, (⊥, p), (α_p, p)) then
6 |   |   Wait[α_p] ← false
7 Attempt[p] ← false
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4  |   if WAIT[s_p] = true then
5  |   |   if (s_p, p) = CSOWNER then
6  |   |   |   if s_p = p then Try(L10)
7  |   |   |   else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10 |   if t_p = ∞ then Try(L3)
11 |   else Try(L6)
12 else if s_p = p then
13 |   if t_p = ∞ then Exit(L2)
14 |   else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER  then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ←  MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4  |   if WAIT[s_p] = true then
5  |   |   if (s_p, p) = CSOWNER then
6  |   |   |   if s_p = p then Try(L10)
7  |   |   |   else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10 |   if t_p = ∞ then Try(L3)
11 |   else Try(L6)
12 else if s_p = p then
13 |   if t_p = ∞ then Exit(L2)
14 |   else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4  |   if WAIT[s_p] = true then
5  |   |   if (s_p, p) = CSOWNER then
6  |   |   |   if s_p = p then Try(L10)
7  |   |   |   else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10 |   if t_p = ∞ then Try(L3)
11 |   else Try(L6)
12 else if s_p = p then
13 |   if t_p = ∞ then Exit(L2)
14 |   else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4  |   if WAIT[s_p] = true then
5  |   |   if (s_p, p) = CSOWNER then
6  |   |   |   if s_p = p then Try(L10)
7  |   |   |   else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10 |   if t_p = ∞ then Try(L3)
11 |   else Try(L6)
12 else if s_p = p then
13 |   if t_p = ∞ then Exit(L2)
14 |   else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, –) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4  |   if WAIT[s_p] = true then
5  |   |   if (s_p, p) = CSOWNER  then
6  |   |   |   if s_p = p then Try(L10)
7  |   |   |   else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10 |   if t_p = ∞ then Try(L3)
11 |   else Try(L6)
12 else if s_p = p then
13 |   if t_p = ∞ then Exit(L2)
14 |   else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Individual Sections

Procedure Recover

```
1  if ATTEMPT[p] = false then Try(L1)
2  w_p ← WAIT[p]
3  if (s_p, −) ← CSOWNER ∧ (s_p ≠ ⊥) then
4      if WAIT[s_p] = true then
5          if (s_p, p) = CSOWNER then
6              if s_p = p then Try(L10)
7              else α_p ← s_p; Exit(L6)
8  t_p ← MYTOKEN[p]
9  if w_p = true then
10     if t_p = ∞ then Try(L3)
11     else Try(L6)
12 else if s_p = p then
13     if t_p = ∞ then Exit(L2)
14     else CriticalSection(L1)
15 else Exit(L4)
```

Thank you for your attention!

Questions?

In a Nutshell: min-array

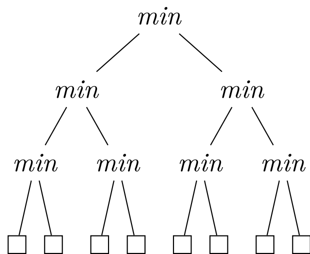


Figure 1 – *Recoverable min-array which emulates a priority process queue.*

In a Nutshell: min-array

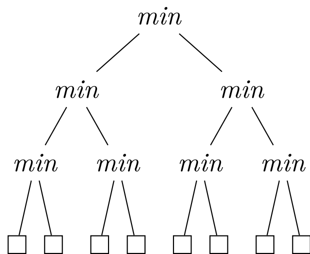


Figure 1 – *Recoverable min-array which emulates a priority process queue.*