Adaptive and Efficient Abortable Mutual Exclusion

Paper by Prasad Jayanti, 2003

Presented by Zhuoya Xiang, Zoe

- **Mutual Exclusion**

At most one process in the critical section at any time

- **Abortable**

A waiting process may abort its attempt to enter the critical section

- **Adaptive and Efficient**

As few remote references as possible
Bounded space and time complexities

Previous Attempts

- **M. L. Scott and W.N. Scherer III.**

  Scalable queue-based spin locks with timeout. (*June 2001*)

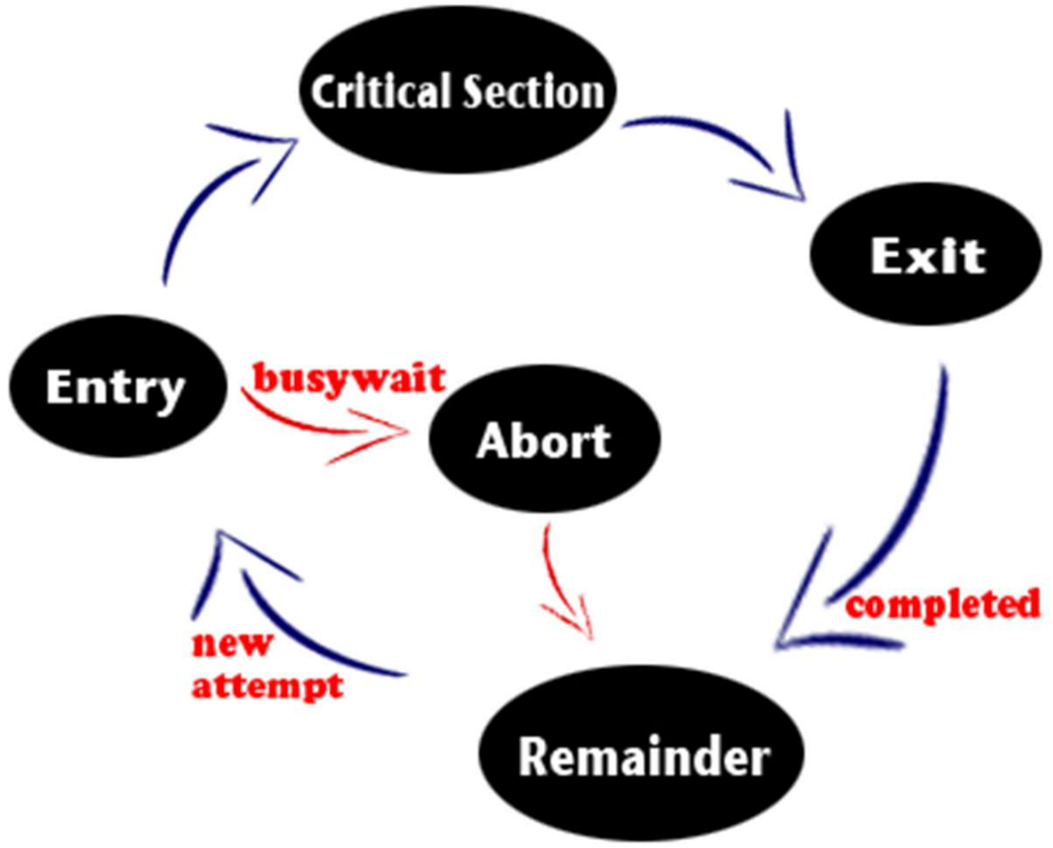
An aborting process may be blocked.

- **M. L. Scott.**

  Non-blocking timeout in scalable queue-based spin locks. (*July 2002*)

Unbounded worst-case time and space complexity

Basic Flow


Critical Section
Exit
Entry
busywait
Abort
completed
new attempt
Remainder

The behavior of LL and SC

- **From Wiki,** **load-link** (LL) and **store-conditional** (SC) are a pair of instructions that together implement a lock-free atomic read-modify-write operation.

- **From this paper,** it says

The operation **LL(O)** returns O's value.

The operation **SC(O, v)** by a process p "succeeds" if and only if no process performed a successful SC on O since p's latest LL.

If SC succeeds, it changes O's value to v and returns true. Otherwise, O's value remains unchanged and SC returns false.

```
procedure Entry(p)
1. Wait(p) = true
2. inc(C, 1)
3. t = read(C)
4. insert(Q, (p, t))
5. promote()
6. promote()
7. wait till Wait(p) = false
```

```
procedure Abort(p)
11. delete(Q, (p, t))
12. promote()
13. if CSowner = p then
14.       CSowner = ⊥
15.       promote()
```

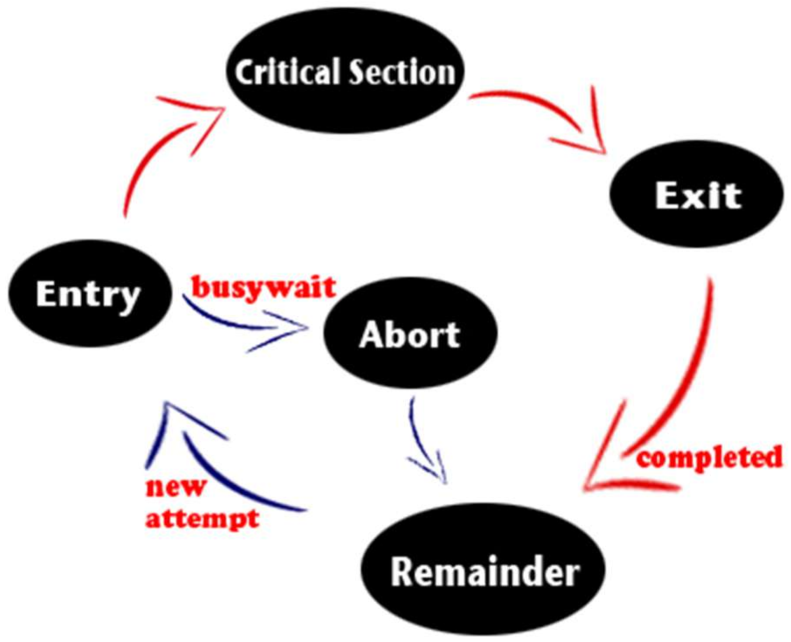
```
procedure Exit(p)
8.  delete(Q, (p, t))
9.  CSowner = ⊥
10. promote()
```

```
procedure promote()
16. if LL( CSowner)   ⊥ then return
17. (q,t') = findmin(Q)
18. if q   ⊥ then LL(Wait(q))
19. if SC( CSowner, q) then
20.       if q   ⊥ then SC(Wait(q),false)
```

Note: Code shown here is for process p.

Scenario 1

Assume two "normal processes" **P1** and **P2** would go through the Entry Section, Critical Section, Exit Section and then Remainder Section, i.e. they will not abort their attempts at the moment.



procedure Entry(p)

```
1. Wait(p) = true
2. inc(C, 1)
3. t = read(C)
4. insert(Q, (p, t))
5. promote()
6. promote()
7. wait till Wait(p) = false
```

procedure promote()

```
16. if LL( CSowner)   ⊥ then return
17. (q,t') = findmin(Q)
18. if q   ⊥ then LL(Wait(q))
19. if SC( CSowner, q) then
20.       if q   ⊥ then SC(Wait(q),false)
```

procedure Exit(p)

```
8.  delete(Q, (p, t))
9.  CSowner = ⊥
10. promote()
```

Initialization:

CSowner = ⊥

C = 1

Q = {(P1,1)}

P2 -> 1

```
    Wait(p1) = true
    C = 2,   t = 2
    Q = {(p1,1),(p2,2)}
    q = p1, t' = 1
    CSowner = p1
    Wait(p1) = false
```

P1 -> 7

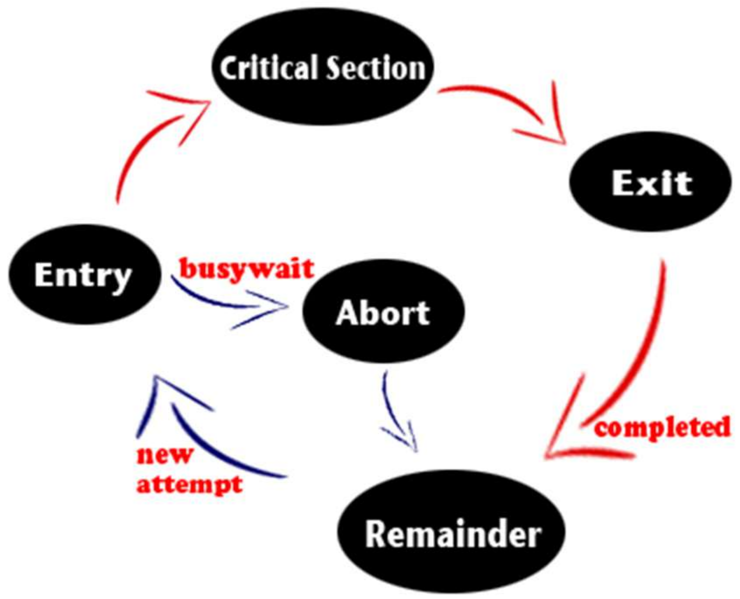
Enter the Critical Section and exit

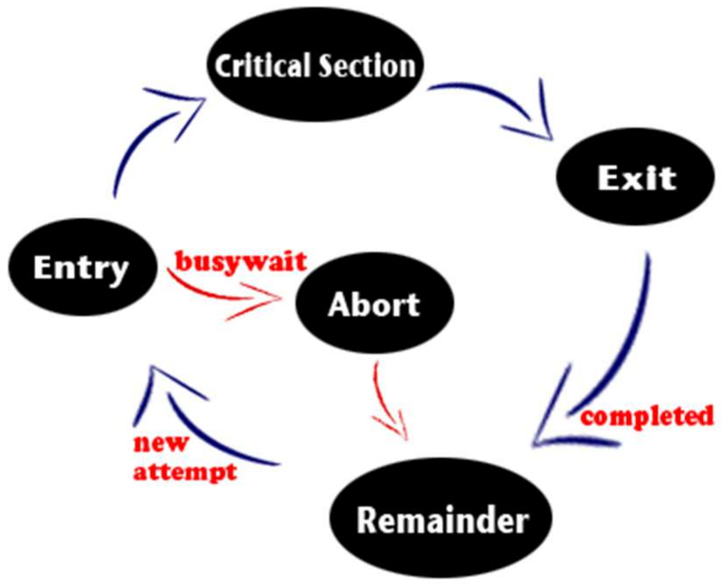
P1 -> 8

```
    Q = {(p2,2)}
    CSowner = ⊥
    q = p2, t' = 2
    CSowner = p2
    Wait(p2) = false
```

Scenario 2

Assume a "normal process" **P1** would go through the Entry Section, Critical Section, Exit Section and then Remainder Section, while **P2** would **abort** its attempt when it is busywaiting.





```
procedure Entry(p)
1. Wait(p) = true
2. inc(C, 1)
3. t = read(C)
4. insert(Q, (p, t))
5. promote()
6. promote()
7. wait till Wait(p) = false
```

```
procedure promote()
16. if LL(CSowner)   ⊥ then return
17. (q,t') = findmin(Q)
18. if q   ⊥ then LL(Wait(q))
19. if SC(CSowner, q) then
20.       if q   ⊥ then SC(Wait(q),false)
```

P1 -> 7

  Wait(p1) = true

P2 -> 17

  q = p1, t′ = 1

P2 -> 18 : LL(Wait(p1))

P2 -> 19

  CSowner = p1

P1 -> 7

  Abort(), Remainder reinitiate a new attempt

P1 -> 1

  Wait(p1) = true

  t = 11

  Q = {……,(p1,11)}

P1 -> 7 : busywait loop

P2 -> 20

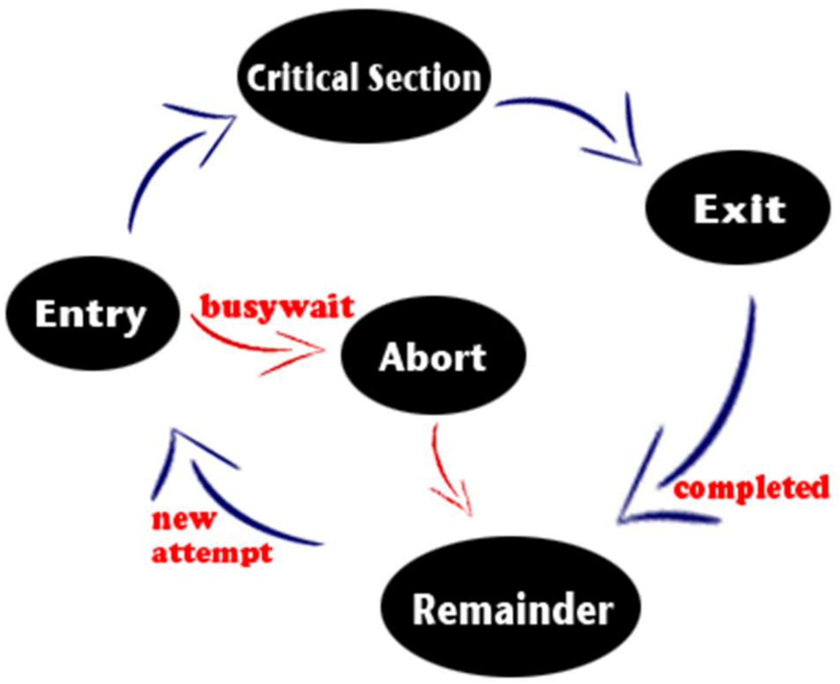
If it is a write,

  Wait(p1) = false

If it is a SC, it fails

Scenario 3

Assume three processes **P1**, **P2**, **P3** and **P1** is going to **abort** its attempt when it is busywaiting.



procedure Abort(p)

```
11. delete(Q, (p, t))
12. promote()
13. if CSowner = p then
14.     CSowner = ⊥
15.     promote()
```

procedure promote()

```
16. if LL(CSowner)   ⊥ then return
17. (q,t') = findmin(Q)
18. if q   ⊥ then LL(Wait(q))
19. if SC(CSowner, q) then
20.     if q   ⊥ then SC(Wait(q),false)
```

P3 -> 17

q = p1, t1 = 1

P1 -> 11 : delete(Q,(p1,1))

If Abort() finishes here → deadlock

P1 -> 12

q = p2, t' = 2

Advance p2 into the Critical Section

(but this could also fail)

P1 - > 13 double check

We are confident that if CSowner doesn't contain p1 by this moment, CSowner will never be assigned to be p1 later, when p1 is in the Remainder Section.

Scenario 4

- Assume two processes **P1**, **P2** and **P1** is in the Critical Section while **P2** is in the Remainder Section.
- Suppose Line 6 is removed.

```
procedure Entry(p)
1. Wait(p) = true
2. inc(C, 1)
3. t = read(C)
4. insert(Q, (p, t))
5. promote()
~~6. promote()~~
7. wait till Wait(p) = false

procedure Exit(p)
8.  delete(Q, (p, t))
9.  CSowner = ⊥
10. promote()

procedure promote()
16. if LL( CSowner)   ⊥ then return
17. (q,t') = findmin(Q)
18. if q   ⊥ then LL(Wait(q))
19. if SC( CSowner, q) then
20.       if q   ⊥ then SC(Wait(q),false)
```

P1 -> 8…18

$q = \perp, t' = \perp$

P2 -> 1,2,3,4,16,17,18

$q = p2$

P1 -> 19

CSowner = ⊥

P2 -> 19

SC fails because p1's successful SC occurred between p2's LL and SC on CSowner.

~~**P2 -> 6**~~

~~**p2 would be written in CSowner successfully and SC(Wait(p2), false) on Line 20 would also be successful.**~~

P2 -> 7 : busywait loop

The loop will never terminate.

Very Basic and Informal Proofs

- (P1) Mutual Exclusion
- (P2) Lockout-freedom
- (P3) Bounded Abort
- (P4) Bounded Exit
- (P5) First-Come-First-Served (FCFS)
- (P6) Local-spin
- (P7) Adaptivity

procedure Entry(p)

```
1. Wait(p) = true
2. inc(C, 1)
3. t = read(C)
4. insert(Q, (p, t))
5. promote()
6. promote()
7. wait till Wait(p) = false
```

procedure Abort(p)

```
11. delete(Q, (p, t))
12. promote()
13. if CSowner = p then
14.       CSowner = ⊥
15.       promote()
```

procedure Exit(p)

```
8.   delete(Q, (p, t))
9.   CSowner = ⊥
10. promote()
```

procedure promote()

```
16. if LL( CSowner)   ⊥ then return
17. (q,t') = findmin(Q)
18. if q   ⊥ then LL(Wait(q))
19. if SC( CSowner, q) then
20.       if q   ⊥ then SC(Wait(q),false)
```

Note: Code shown here is for process p.

Very Basic and Informal Proofs

- **(P1) Mutual Exclusion**
- **(P2) Lockout-freedom**
- **(P3) Bounded Abort**
- **(P4) Bounded Exit**
- **(P5) First-Come-First-Served (FCFS)**
- **(P6) Local-spin**
- **(P7) Adaptivity**

(Scenario 4)

Deadlock-freedom + Starvation-freedom

The time complexity depends only on point contention k and not on the number of processes n for which the algorithm is designed. In practice, k << n.

Conclusion and open problems

- **Conclusion:** The first local-spin abortable mutual exclusion algorithm with bounded complexities.
- **P1:** The algorithm uses token numbers that grow without bound.
- **P2:** Either design an abortable algorithm of O(1) remote reference complexity or prove its impossibility.

This algorithm has O(min(k, log n)) remote reference complexity.

Influences – 29 Citations

- **Adaptive randomized mutual exclusion in sub-logarithmic expected time** by Danny Hendler & Philipp Woelfel in 2010

"We present a randomized adaptive mutual exclusion algorithms with O(log k/loglog k) expected amortized RMR complexity…This establishes that sub-logarithmic adaptive mutual exclusion, using reads and writes only, is possible."

- **Group mutual exclusion in O(log n) RMR** by Vibhor Bhatt & Chien-Chung Huang in 2010

"We show that in the CC model, using registers and LL/SC variables, our algorithm achieves O(min(log n,k)) RMR, which is so far the best. Moreover, given a recent result of Attiya, Hendler and Woelfel showing that exclusion problems have a Ω(log n) RME lower bound using registers, comparison primitives and LL/SC variables."

Discussion