



NOVA SCHOOL OF  
SCIENCE & TECHNOLOGY

COMPUTER SCIENCE DEPARTMENT

# Solving Mutual Exclusion (2)

lecture 14 (2021-05-03)

**Master in Computer Science and Engineering**

— Concurrency and Parallelism / 2020-21 —

João Lourenço <[joao.lourenco@fct.unl.pt](mailto:joao.lourenco@fct.unl.pt)>

# Summary

- **Solving Mutual Exclusion**

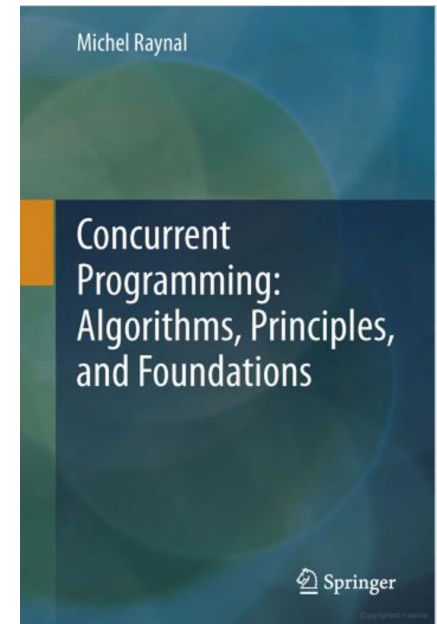
- Mutex based on Specialized Hardware Primitives

- **Reading list:**

- **Chapter 2** of the book  
Raynal M.;

- Concurrent Programming: Algorithms, Principles, and Foundations;**

- Springer-Verlag Berlin Heidelberg (2013);  
ISBN: 978-3-642-32026-2



# Mutex Based on Specialized Hardware Primitives

- In the previous lecture we studied mutual exclusion algorithms based on atomic read/ write registers
- These algorithms are important because
  - Understanding their design and their properties provides us with precise knowledge of the difficulty and subtleties that have to be addressed when one has to solve synchronization problems
  - They capture the essence of synchronization in a read/write shared memory model
- Nearly all shared memory multiprocessors propose built-in primitives (i.e., atomic operations implemented in hardware) specially designed to address synchronization issues

# The *test&set()*/*reset()* primitives

- This pair of primitives, denoted **test&set()** and **reset()**, is defined as follows:
- Let  $X$  be a shared register initialized to 1
- **$X.test\&set()$**  sets  $X$  to 0 and returns its previous value
- **$X.reset()$**  writes 1 into  $X$  (i.e., resets  $X$  to its initial value)
- Both **test&set()** and **reset()** are atomic

# Mutual exclusion with *test&set()* / *reset()*

Invariant?

$$X + \sum_{i=1}^n r_i = 1$$

**operation** `acquire_mutex()` **is**

```
repeat  $r \leftarrow X.\text{test\&set}()$  until ( $r = 1$ ) end repeat;  
return()
```

**end operation.**

**operation** `release_mutex()` **is**

```
 $X.\text{reset}();$  return()
```

**end operation.**

- ✓ mutual exclusion
- ✓ progress **Really??**

# The *swap()* primitive

---

- Let  $X$  be a shared register
- The primitive  **$X.swap(v)$**  atomically assigns  $v$  to  $X$  and returns the previous value of  $X$

# Mutual exclusion with swap()

Invariant?

$$X + \sum_{i=1}^n r_i = 1$$

**operation** acquire\_mutex() **is**

$r \leftarrow 0;$

**repeat**  $r \leftarrow X.swap(r)$  **until**  $(r = 1)$  **end repeat;**

return()

**end operation.**

**operation** release\_mutex() **is**

$X.swap(r);$  return()

**end operation.**

Assumes 'r' was  
not changed  
(i.e.,  $r=1$ )

- ✓ mutual exclusion
- ✓ progress **Really??**

# The *compare&swap()* primitive

- Let 'X' be a shared register and 'old' and 'new' be two values
- The primitive **X.compare&swap(old, new)**
  - returns a Boolean value
  - is defined by the following code that is assumed to be executed atomically:

```
X.compare&swap(old, new) is  
    if (X = old) then X ← new; return(true)  
    else return(false)  
end if.
```



# Mutual exclusion with *compare&swap()*

$X$  is an atomic *compare&swap* register initialized to 1

```
operation acquire_mutex() is old new  
    repeat  $r \leftarrow X.\text{compare\&swap}$  1, 0 until ( $r$ ) end repeat;  
    return()  
end operation.
```

```
operation release_mutex() is  
     $X \leftarrow 1$ ; return()  
end operation.
```

# Mutual exclusion with *compare&swap()*

$X$  is an atomic *compare&swap* register initialized to 1

**operation** *acquire\_mutex()* **is**

**repeat**  $r \leftarrow X.\text{compare\&swap}(1, 0)$  **until**  $(r)$  **end repeat;**  
return()

**end operation.**

**operation** *release\_mutex()* **is**

$X \leftarrow 1$ ; return()

**end operation.**

Invariant?

$$X + \sum_{i=0}^n (r_i = TRUE) = 1$$

# Starvation freedom

- All the previous algorithms, implementing mutexes with

test&set()  
swap()  
compare&swap()

✓ mutual exclusion  
✗ progress but not for all the processes

**are not starvation free!**

*This means that in presence of contention a process  $p_i$  may always “lose the race” and never get the lock*

# Mutual exclusion: deadlock and starvation-free algorithm

**operation** `acquire_mutex(i)` **is**

- (1)  $FLAG[i] \leftarrow up;$
- (2) **wait**  $(TURN = i) \vee (FLAG[TURN] = down)$  ;
- (3) `LOCK.acquire_lock(i);`
- (4) `return();`

**end operation.**

**operation** `release_mutex(i)` **is**

- (5)  $FLAG[i] \leftarrow down;$
- (6) **if**  $(FLAG[TURN] = down)$  **then**  $TURN \leftarrow (TURN \bmod n) + 1$  **end if;**
- (7) `LOCK.release_lock(i);`
- (8) `return();`

**end operation.**

- ✓ mutual exclusion
- ✓ progress
- ✓ no starvation
- ✗ fairness

# The *fetch&add()* primitive

- Let  $X$  be a shared register
- The primitive  **$X.\text{fetch\&add}()$**  atomically adds 1 to  $X$  and returns the new value
  - In some variants the value that is returned is the previous value of  $X$
  - In other variants, a value  $c$  is passed as a parameter and, instead of being increased by 1,  $X$  becomes  $X + c$

# Mutual exclusion with *fetch&add()*

**operation** `acquire_mutex()` **is**

```
my_turn ← TICKET.fetch&add();
```

```
repeat no-op until (my_turn = NEXT) end repeat;
```

```
return()
```

**end operation.**

**operation** `release_mutex()` **is**

```
NEXT ← NEXT + 1; return()
```

**end operation.**

Not atomic!  
Why does it work?

- ✓ mutual exclusion
- ✓ progress
- ✓ no starvation
- ✓ fairness

# The END

---