SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

# CONCURRENT SYSTEMS
# LECTURE 3

Prof. Daniele Gorla

# MUTEX with specialized HW primitives

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Atomic R/W registers provide quite a basic computational model.

We can strenghten the model by adding specialized HW primitives, that essentially perform in an atomic way the combination of some atomic instructions.

Usually, every operating system provides at least one specilized HW primitive.

The most common ones are:

- **Test&set**: atomic read+write of a boolean register
- **Swap**: atomic read+write of a general register
- **Fetch&add**: atomic read+increase of an integer register
- **Compare&swap**: atomic comparison+write of a general register; returns a boolean (the result of the comparison)
- …

# MUTEX with Test&set

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Let X be a boolean register; the **Test&set** primitive is implemented as follows:

```
X.test&set() :=
        tmp ← X
        X ← 1
        return tmp
```

*atomic (by hardware means)*

By using this primitive, MUTEX can be ensured by this simple protocol:

```
Initialize X at 0

lock() :=
     wait X.test&set() = 0
     return
```

```
unlock() :=
     X ← 0
     return
```

# MUTEX with Swap

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Let X be a general register; the **Swap** primitive is implemented as follows:

```
X.swap(v) :=
        tmp ← X
        X ← v
        return tmp
```

*atomic (by hardware means)*

By using this primitive, the previous protocol for MUTEX can be adapted to the swap primitive by noting that

```
X.test&set() = X.swap(1)
```

# MUTEX with Compare&swap

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Let X be a boolean register; the **Compare&swap** primitive is implemented as follows:

```
X.compare&swap(old, new) :=
        if X = old then X ← new
                                  return true
        return false
```

*atomic*

By using this primitive, MUTEX can be obtained as follows:

```
Initialize X at 0

lock() :=
     wait X.compare&swap(0,1)=true
     return

unlock() :=
     X ← 0
     return
```

# MUTEX with Fetch&add

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Up to now, all solutions enjoy deadlock freedom, but allow for starvation

→ use Round Robin to promote the liveness property

Let X be an integer register; the **Fetch&add** primitive is implemented as follows:

```
X.fetch&add(v) :=
        tmp ← X
        X ← X+v          } atomic
        return tmp
```

By using this primitive, MUTEX can be obtained as follows:

*Bounded bypass with bound n-1*

```
Initialize TICKET and NEXT at 0

lock() :=
     my_tick ← TICKET.fetch&add(1)
     wait my_tick = NEXT
     return

unlock() :=
     NEXT ← NEXT+1
     return
```

# Safe Registers

Atomic R/W and specialized HW primitives provide some form of atomicity

→ is it possible to enforce MUTEX without atomicity?

A **MRSW Safe register** is a register that provides READ and WRITE such that:

1. Every READ that does not overlap with a WRITE returns the value stored in the register
2. A READ that overlaps with a WRITE returns any value (of the register domain)

A **MRMW Safe register** behaves like a MRSW safe register, when WRITE operations do not overlap; otherwise, in case of overlapping WRITEs, the register can contain any value (of the register domain)

This is the weakest type of register that is useful in concurrency

# Bakery algorithm (Lamport 1974)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Idea:

- Every process gets a ticket
- Because we don't have atomicity, tickets may be not unique
- Tickets can be made unique by pairing them with the process ID
- The smallest ticket (seen as a pair) grants the access to the CS

```
Initialize FLAG[i] to down and MY_TURN[i] to 0, for all i

lock(i) :=
     FLAG[i] ← up
     MY_TURN[i] ← max{MY_TURN[1],…,MY_TURN[n]}+1      } doorway
     FLAG[i] ← down
     forall j ≠ i
         wait FLAG[j] = down                          } bakery
         wait (MY_TURN[j] = 0 OR                      } (including CS)
               ⟨MY_TURN[i],i⟩ < ⟨MY_TURN[j],j⟩)

unlock(i) :=
     MY_TURN[i] ← 0
```

**Lemma 1:** Let pi enter the bakery before pj enters the doorway; then,

MY_TURN[i] < MY_TURN[j].

*Proof:*

- Let t be the value of MY_TURN[i] after that pi exits the doorway
- When pj computes its ticket, it reads t from MY_TURN[i] (there is no write overlapping with this read)
- Hence, MY_TURN[j] is at least t+1

**Lemma 2:** Let pi be in the CS and pj is in the doorway or in the bakery; then,

$$\langle MY\_TURN[i] , i\rangle < \langle MY\_TURN[j] , j\rangle.$$

*Proof:*

If pi is in the CS, it has terminated its first wait for j

→ let's consider the read of FLAG[j] done by pi that terminates such wait

W.r.t. the execution of pj, it can be that

- This read overlaps with FLAG[j] ← up: by Lemma1, MY_TURN[i] < MY_TURN[j] and √
- This read is contained within the computation of MY_TURN[j]

  → this is not possible, since MY_TURN is computed with the FLAG up
- This read overlaps with FLAG[j] ← down or this read happens when pj is in the bakery:
  - MY_TURN[j] has been decided and no write will change it until pj is in the bakery
  - MY_TURN[j] > 0 (it has been obtained by summing 1 to some natural number)
  - When pi has evaluated the second wait for j, it found ⟨MY_TURN[i] , i⟩ < ⟨MY_TURN[j] , j⟩ and √

**MUTEX:** pi and pj cannot simultaneously be in the C.S.

*Proof:* By contradiction, by Lemma2 applied twice, we would have

⟨MY_TURN[i] , i⟩ < ⟨MY_TURN[j] , j⟩ and ⟨MY_TURN[j] , j⟩ < ⟨MY_TURN[i] , i⟩

**Deadlock freedom:** by contradiction, assume that there is a lock but nobody enters its CS

- All processes in the bakery (call this set Q) are blocked in their wait
- The first wait cannot block for ever
  - All pi ∈ Q have their FLAG down
  - All pi ∉ Q have their FLAG down (if they're not in the doorway) or will eventually put their FLAG down (they cannot remain in the doorway for ever)
- The second wait cannot block all of them for ever
  - Tickets can be totally ordered (lexicographically)
  - Let ⟨MY_TURN[i] , i⟩ be the minimum
  - The second wait evaluated by pi eventually succeeds for all j
    - If pj is before the doorway → MY_TURN[j] = 0
    - If pj is in the doorway → MY_TURN[i] < MY_TURN[j] (bec.of Lemma1)
    - If pj is in the bakery, by assumption ⟨MY_TURN[i] , i⟩ < ⟨MY_TURN[j] , j⟩ since it is the minimum

**Bounded bypass (with bound n-1):**

Let pi and pj competing for the CS and pj wins

Then, pj enters its CS, completes it, unlocks and then invokes lock again

- If pi has entered the CS → √
- Otherwise, by Lemma1, MY_TURN[i] < MY_TURN[j]
  → pj cannot bypass pi again!
- At worse, pi has to wait all other proceeses before entering its CS
  (indeed, since there is no deadlock, when pi is waiting somebody enters the CS)

# Aravind's algorithm (2011)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Problem with Lamport's alg.: registers must be unbounded (every invocation of lock potentially increases the counter by 1 → domain of the registers is all naturals!)

For all processes, we have a FLAG and a STAGE (both binary MRSW), and a DATE (a MRMW register that ranges from 1 to 2n)

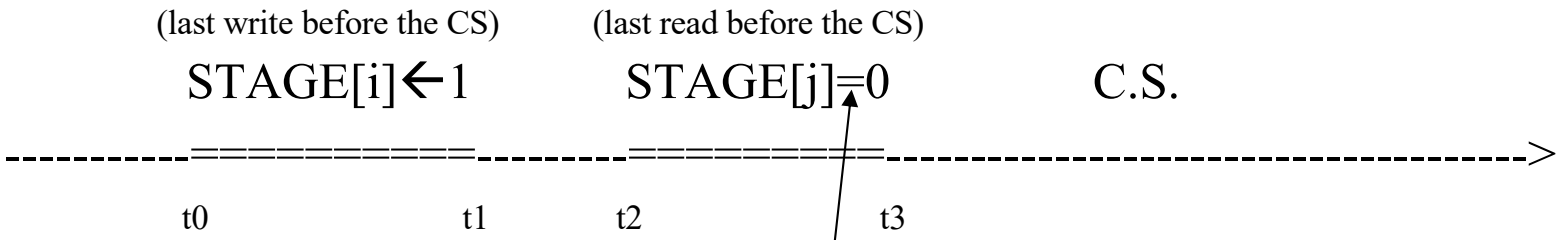
For all i, initialize
- FLAG[i] to down
- STAGE[i] to 0
- DATE[i] to i

```
lock(i) :=
    FLAG[i] ← up
    repeat
        STAGE[i] ← 0
        wait (∀j≠i. FLAG[j] = down OR
                    DATE[i] < DATE[j])
        STAGE[i] ← 1
    until ∀j≠i. STAGE[j] = 0
```

```
unlock(i) :=
    tmp ← max_j{DATE[j]}+1
    if tmp ≥ 2n
       then ∀j.DATE[j] ← j
       else DATE[i] ← tmp
    STAGE[i] ← 0
    FLAG[i] ← down
```

**Thm.:** if pi is in the CS, then pj cannot simultaneously be in its CS.

*Proof:* By contradiction. Let us consider the execution of pi leading to its CS:




Let's consider the last write of pj before its CS (i.e., STAGE[j]←1)

- It cannot complete before t2, because of
- So, it must overlap with [t2,t3] or happen after t3
  - → but then the last read of pj from STAGE[i] happens after t1
  - → pj finds STAGE[1]=1 and cannot enter its CS

**Cor.:** DATE is never written concurrently.

**Lemma 1:** exactly every n CSs there is a reset of DATE.

*Proof:* Because of the previous corollary

- The first CS leads $\max_j\{DATE[j]\}$ to n+1
- The second CS leads $\max_j\{DATE[j]\}$ to n+2
- …
- The n-th CS leads $\max_j\{DATE[j]\}$ to n+n = 2n → RESET

**Lemma 2:** there can be at most one reset of DATE during an invocation of lock.

Proof:

Let pi invoke lock. If no reset occurs → √

Otherwise, let us consider the moment in which a reset occurs.

If pi is the next process that enters the CS → √

Otherwise, let pj be the process that enters; its next date is n+1 > DATE[i]

→ pj cannot surpass pi again (before a RESET)

The worst case is when all proc's perform lock together and i=n

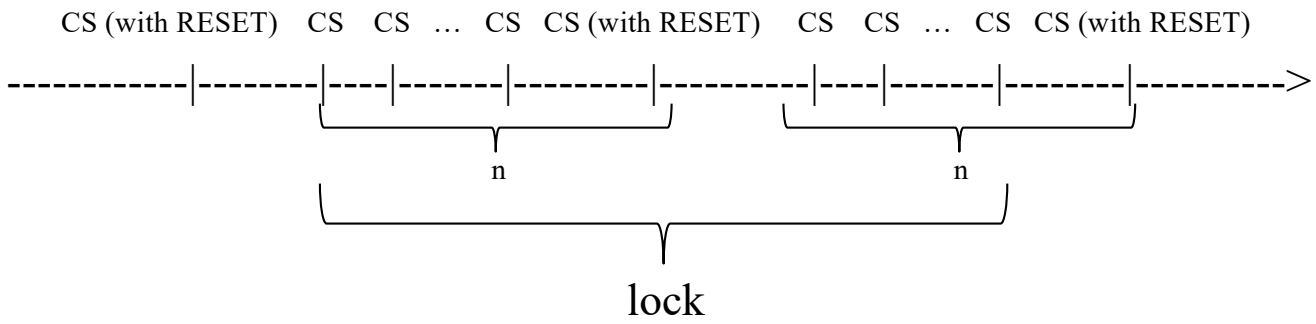
→ all $p_1,\ldots,p_{n-1}$ surpass $p_n$

→ then $p_n$ enters and it resets the DATE in its unlock

**Thm.:** The algorithm satisfies bounded bypass with bound 2n-2.

*Proof:*

By Lemma1:




By Lemma2:

i.e., there is an upper bound of 2n-1

→ this bound is not reachable, whereas the bound reachable is 2n-2:

- pn invokes lock alone, completes its CS (the first after the reset) and its new DATE is n+1
- Then all processes invoke lock simultaneously
- pn has to wait all other processes to complete their CSs (after that pi completes its CS it has DATE[i] ← n+i+1)
- When $p_{n-1}$ completes its CS, its new DATE will be n+(n-1)+1 = 2n → RESET
- Now all p1,…,$p_{n-1}$ invoke lock again and complete their CSs (after that pi completes its CS, now it has DATE[i] ← n+i)
- So, pn has to wait n-1 CSs for the reset, and another n-1 CSs before entering again

# Improvement of Aravind's algorithm

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Consider the following revision of Aravind's UNLOCK:

```
unlock(i) :=
    ∀j≠i.if DATE[j] > DATE[i] then DATE[j] ← DATE[j]-1
    DATE[i] ← n
    STAGE[i] ← 0
    FLAG[i] ← down
```

Since the LOCK is like before, the revised protocol satisfies MUTEX.

Furthermore, you can prove that it satisfies bounded bypass with bound $n$-1

→ EXERCISE!