SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

CONCURRENT SYSTEMS
LECTURE 2

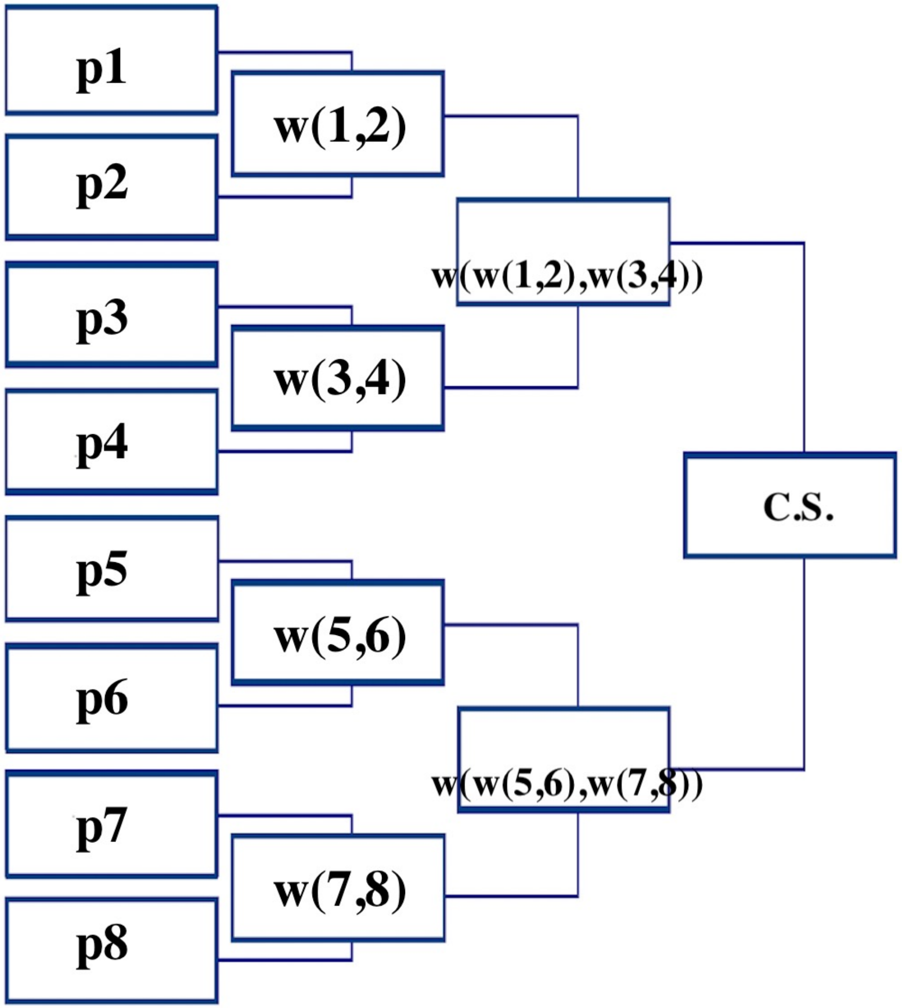
Prof. Daniele Gorla

Tournament-based algorithm

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Even without contention, Peterson's algorithm costs $O(n^2)$

A first way to reduce this cost is by using a tournament of MUTEX between pairs of processes:



By using Peterson's algorithm for 2 proc, a process wins after $\lceil \log_2 n \rceil$ competitions, each of constant cost.

→ $O(\log n)$

A constant-time algorithm (for *n* processes)

The cost can be further reduced to O(1).

To begin, consider the following idea:

```
Initialize Y at ⊥, X at any value (e.g., 0)
lock(i) :=                                          unlock(i) :=
     X ← i                                               Y ← ⊥
     if Y ≠ ⊥ then FAIL                                  return
               else Y ← i
                    if X = i then return
                              else FAIL
```

Without contention, this requires 4 accesses to the registers for entering the CS

Problem:

- we don't want the FAIL (that forces the process to invoke lock again and again), but an implementation of lock that keeps the process inside this primitive until it wins
- It is possible to have an execution where nobody accesses its CS

  → if repeated for ever, enatils a deadlock

Fast MUTEX algorithm (by Lamport)

```
Initialize Y at ⊥, X at any value (e.g., 0)

lock(i) :=
*   FLAG[i] ← up
    X ← i
    if Y ≠ ⊥ then FLAG[i] ← down
                  wait Y = ⊥
                  goto *
             else Y ← i
                  if X = i then return
                           else FLAG[i] ← down
                                ∀j.wait FLAG[j] = down
                                if Y = i then return
unlock(i) :=                             else wait Y = ⊥
    Y ← ⊥                                     goto *
    FLAG[i] ← down
    return
```

MUTEX: if pi is in CS, then no other pj can simultaneaously be in CS

Proof:

How can pi enter its CS?

```
a)   F[i]←up  X←i    Y=⊥    Y←i     X=i             CS
      ------|---------|--------|---------|----------|------------------------>
```

For pj to enter its CS, it must find Y at ⊥, so it must read Y here

Where did pj write X?

not here, otherwise pi would not have read i in X

So, it must have written X here.

Hence, when pj reads X, it finds it different from j

→ it must wait for pi's unlock before starting again

→ pj cannot be in CS while pi is

b) F[i]←up X←i Y=⊥ Y←i X≠i F[i]←down ∀k.F[k]=down Y=i CS

```
------|-----|------|------|------|-------|-------------|---------|----------->
```

pj must read Y
here

Where did pj write X?

If here,
like (a)

So, let pj write X here

- if pj reads X at j → it enters its CS, that however must be finished before
  → pj's CS doesn't overlap with pi's one
- otherwise, pj must have written Y before pi (since pi finds Y at i)
  → pj is blocked until pi unlocks

<u>Deadlock freedom:</u> let pi invoke lock

- If it eventually wins → √
- If it is blocked for ever, where can it be blocked?

1. In the second wait Y = ⊥

   → in this case, it read a value in Y different from i

   → there is a ph that wrote Y after pi

   → let us consider the last of such ph's → it will eventually win → √

2. In the ∀j.wait FLAG[j]=down

   → this wait cannot block a process for ever

   - if pj doesn't lock, it flag is down
   - if pj doesn't find Y at ⊥, it puts its flag down
   - if pj doesn't find X at j, it puts its flag down

     otherwise pj enters its CS and eventually unlocks (flag down)

3. In the first wait Y = ⊥

→ since pj read a value different from ⊥, there is at least one pk that wrote Y before (but has not yet unlocked)

→ if pk eventually enters its CS → √
otherwise, it must be blocked for ever as well. Where?
- In the second wait Y = ⊥: but then there exists a ph that eventually enters its CS (see point 1 above) → √
- In the ∀j.wait FLAG[j]=down: this wait cannot block a process for ever (see point 2 above)

Fast MUTEX algorithm (by Lamport)

Without contention, this algorithm requires 5 accesses to the shared registers

It can be proved to satisfy MUTEX and deadlock freedom (you can easily built a scenario where a process is starved)

→ we will see that every deadlock-free algorithm can be turned into a bounded bypass one (but with a quadratic bound…)

To sum up: with atomic R/W registers, we have

- With 2 processes, a O(1) algorithm that satisfies bounded bypass (with bound 1)
- With *n* processes:
  - a $O(n^2)$ algorithm that satisfies starvation freedom
  - a $O(\log n)$ algorithm that satisfies bounded bypass (with bound $\lceil \log_2 n \rceil$)
  - a O(1) algorithm that satisfies deadlock freedom

From deadlock freedom to bounded bypass

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Let DLF be a deadlock free protocol for MUTEX.
We now want to turn it into a bounded bypass protocol for MUTEX

Round Robin algorithm

→ the name comes from a middle age habit for signing petitions, called *Ruban Rond* (that means «round ribbon»)

→ a circular way of signing, to hide the identity of the initiator

```
Initialize FLAG[i] to down (∀i) and TURN to any proc.id.

lock(i) :=
   FLAG[i] ← up
   wait (TURN = i OR
         FLAG[TURN] = down)
   DLF.lock(i)
   return

unlock(i) :=
   FLAG[i] ← down
   if FLAG[TURN] = down then
        TURN ← (TURN+1) mod n
   DLF.unlock(i)
   return
```

MUTEX for RR algorithm follows from the assumed MUTEX of DLF

<u>Deadlock freedom of RR:</u> if at least one process invokes RR.lock, then at least one process enters the CS.

Proof:
Since DLF enjoys deadlock freedom, it suffices to prove that at least one process invokes DLF.lock (i.e., at least one proc exists from its wait)

If TURN=k and p_k invoked lock, then it finds TURN = k and exits its wait
Otherwise, any other process finds FLAG[TURN]=down and exits from its wait

Lemma 1: If TURN = i and FLAG[i] = up, then pi enters the CS in at most (*n*-1) iterations

Proof:

OBS1: TURN changes only when FLAG[i] is down (i.e., after pi has completed its CS)

OBS2: FLAG[i]=up → either pi is in its CS → √
or pi is competing for its CS → it eventually invokes (or has already invoked) DLF.lock

OBS3: if pj invokes lock after that FLAG[i] is set, pj blocks in its wait

Let Y be the set of processes competing for the CS (i.e., suspended on DLF.lock)

- Because of OBS2, $i \in Y$
- Because of OBS3, once FLAG[i] is set, Y cannot grow anymore

- Because DLF is deadlock free, eventually one py ∈ Y wins

  If y=i → √

  otherwise, Y shrinks by one (the py that entered the CS). Indeed:

  because of OBS1, TURN (and FLAG[TURN]) don't change

  → py cannot enter Y again

  We can iterate this reasoning and eventually pi will win

  → the worst case is when Y contains all proc's and pi is the last winner

Lemma 2: If FLAG[i] = up, then TURN is set to i in at most $(n\text{-}1)^2$ iterations

Proof:

If TURN=i when FLAG[i] is set → √

By Deadlock freedom of RR, at least one proc eventually unlocks

- If FLAG[TURN]=down, then TURN is increased; othw., by Lemma1 p_{TURN} wins in at most (n-1) iterations (and increases TURN)
- If now TURN=i then √; otherwise, we repeat the reasoning

The worst case is when TURN=(i+1) mod n when FLAG[i] is set

Bounded bypass of RR: if a process invokes RR.lock, then it enters the CS in at most $n(n\text{-}1)$ iterations

Proof:

- pi invokes lock → FLAG[i] is set to up
- By lemma 2, in $(n\text{-}1)^2$ itrerations TURN is set to i
- By lemma 1, in $(n\text{-}1)$ iterations pi enters the CS
- $(n\text{-}1)^2 + (n\text{-}1) = n(n\text{-}1)$