SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

CONCURRENT SYSTEMS
LECTURE 1

Prof. Daniele Gorla

Concurrency

- **Sequential algorithm**: formal description of the behaviour of an *abstract* sequential state machine

  → IDEA
- **Program**: a sequential algorithm written in a programming language

  → TEXT
- **Process**: a program executed on a *concrete* machine, characterized by its *state* (the values of the PC and of other registers)

  → ACTION

- **Sequential process** (or **thread**): is a process that follows one single control flow (i.e., one program counter)

- **Concurrency**: a *set* of sequential state machines, that run simultaneously and interact through a *shared medium*

  → **Multiprocess program** or **Concurrent system**

- Advantages:
  - Combine the work of different processes, that in parallel solve different tasks
  - Simplify the programming of a complex task by dividing it into simpler ones

Features of a Concurrent System

Many features can be assumed, e.g.

- Reliable vs Unreliable
- Synchronous vs Asynchronous
- Shared memory vs Channel-based communication
- …

We shall focus on reliable, asynchronous and shared memory systems

- **Reliable** = every process correctly executes its program
- **Asynchronous** = no timing assumption (i.e., every process has its own clock, and clocks are independent one from the other)
- **Shared memory** = every process has a local memory (accessible only by itself) but there are a few registers that can be accessed by every process

How many processors?

- Usually, **one for every process** (we assume this, to simplify the presentation)
- But we can also have fewer (actually, also just one!)

Synchronization: Cooperation vs Competition

Synchronization = the behaviour of one process depends on the behaviour of the others.

This requires two fundamental interactions:

- Cooperation
- Competition

COOPERATION

Different processes work to let all of them succeed in their task.

Examples:

1. *Rendezvous*: every involved process has a control point that can be passed only when all processes are at their control points

   → The set of all control points is called ***Barrier***

2. *Producer-consumer*: 2 kinds of processes, one that produces data and one that consumes them, under the following constraints:
   - Only produced data can be consumed
   - Every datum can be consumed at most once

Synchronization: Cooperation vs Competition

COMPETITION

Different processes aim at executing some action, but only one (or few) of them succeeds.

Usually, this is related to the access of the same shared resource.

Example: two processes want to withdraw from a bank account (e.g., 1 M€)

Basic (sequential) program:

```
function withdraw() {
        x := account.read();
        if x ≥ 1M€ then account.write(x – 1M€)
}
```

The problem is that, while `read` and `write` are usually considered as atomic, their sequential composition is not. Assume to have an account with exactly 1M€:

```
                     p1      a.read; a.write
Correct execution:   -----------------|--------|------------------|---------|------------ time
                     p2                                    a.read;   return

                     p1      a.read                        a.write
Wrong execution:     ------------------|------------|-------------|-------------|------- time
                     p2                         a.read;                   a.write
```

Mutual Exclusion (MUTEX)

Ensure that some parts of the code are executed as *atomic* (i.e., without intermission of any other process)

This is needed both in competition, but also in cooperation (when accessing a shared resource) → EXAMPLE: if both previous processes want to increase the account balance of 1M€

Remark: not all code parts require MUTEX (only those that affect shared data)

Critical section: a set of code parts that must be run without interferences, i.e., when a process is in a C.S. (on a certain shared object), then no other process is in a C.S. (on that shared object).

MUTEX problem: design an entry protocol (*lock*) and an exit protocol (*unlock*) such that, when used to encapsulate a C.S. (for a given shared object), ensure that at most one process at a time is in a C.S. (for that shared object).

Assumptions:

1. All C.S.s terminate
2. The code is well-formed (*lock* ; *<critical_section>* ; *unlock*)

MUTEX: Safety and Liveness properties

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Every solution to a problem should satisfy (at least) 2 properties:

1. **Safety**: «nothing bad ever happens»
2. **Liveness**: «something good eventually happens»

Both of them are needed to avoid trivial solutions:

- Liveness without safety: allow anything → this also allows wrong solutions
- Safety without liuveness: forbid anything → no activity in the system

So, safety is necessary for correctness, liveness for meaningfulness.

For MUTEX:

- Safety: there is at most one process at a time that is in a C.S.
- Liveness: various options
  - **Deadlock freedom**: if there is at least one invocation of lock, eventually after at least one process enters a C.S.
  - **Starvation freedom**: every invocation of lock eventually grants access to the associated C.S.
  - **Bounded bypass**: let n be the number of processes; then, there exists $f : \mathbf{N} \longrightarrow \mathbf{N}$ s.t. every lock enters the CS after at most $f(n)$ other CSs.

A hierarchy of liveness properties

Bounded bypass ➔ Starvation freedom ➔ deadlock freedom (by def.)

Both inclusions are strict:

- Deadlock freedom $\not\Rightarrow$ Starvation freedom:

  Let p1, p2, p3 run the same code: `while TRUE do {lock; unlock}`

  and consider the following sequence of actions (underlined actions succeed):

p1	*lock*		*lock*		*lock*		*lock*		…
p2	$\underline{\textit{lock}}$	*unlock*			$\underline{\textit{lock}}$	*unlock*			…
p3			$\underline{\textit{lock}}$	*unlock*			$\underline{\textit{lock}}$	*unlock*	…

  ---|------|--------|--------|--------|------|--------|------------- *time*

- Starvation freedom $\not\Rightarrow$ Bounded bypass:

  Assume a f and consider the scheduling above, where p2 wins $f(3)$ times and so does p3

  → p1 looses (at least) $2f(3)$ times before winning

Atomic R/W registers

We will consider different computational models according to the available level of atomicity of the operations provided.

Atomic Read/Write registers: these are storage units that can be accessed through two operations (READ and WRITE) such that

1. Each invocation of an operation
   - looks instantaneous, i.e. it can be depicted as a single point on the timeline (there exists a function t : $\mathbf{OpInv} \longrightarrow \mathbf{R}^{+}$)
   - may be located in any point between its starting and ending time (we have that $t(\text{opInv}) \in [t_{\text{start}}(\text{opInv}) , t_{\text{end}}(\text{opInv})]$)
   - does not happen together with any other operation (function t is injective: $t(\text{opInv}) \neq t(\text{opInv'})$ whenever $\text{opInv} \neq \text{opInv'}$)
2. Every READ returns the closest preceeding value written in the register, or the initial value (if no WRITE has occurred).

According to whether a register can be read/written by just one process or by many different ones, we have: *single-read/single-write* (**SRSW**), *single-read/multiple-write* (**SRMW**), *multiple-read/single-write* (**MRSW**), or *multiple-read/multiple-write* (**MRMW**).

Peterson algorithm (for 2 processes)

Let's try to enforce MUTEX with just 2 processes.

1st attempt:

```
lock(i) :=                                  unlock(i) :=
    AFTER_YOU ← i                               return
    wait AFTER_YOU ≠ i
    return
```

This protocol satisfies MUTEX, but suffers from deadlock (if one process never locks)

2nd attempt:

```
Initialize FLAG[0] and FLAG[1] to down
lock(i) :=                                  unlock(i) :=
    FLAG[i] ← up                                FLAG[i] ← down
    wait FLAG[1-i] = down                       return
    return
```

Still suffers from deadlock if both processes simultaneously raise their flag.

Peterson algorithm (for 2 processes)

Correct solution:

```
Initialize FLAG[0] and FLAG[1] to down

lock(i) :=
    FLAG[i] ← up
    AFTER_YOU ← i
    wait (FLAG[1-i] = down
          OR AFTER_YOU ≠ i)
    return

unlock(i) :=
    FLAG[i] ← down
    return
```

Features:

- It satisfies MUTEX (if p is in CS then q cannot)
- It satisfies bounded bypass, with bound = 1
- It requires 2 one-bit SRSW registers (the flags) and 1 one-bit MRMW register (AFTER_YOU)
- Each lock-unlock requires 5 accesses to the registers

<u>MUTEX:</u> by contr., assume that p0 and p1 are simultaneously in CS.
How has p0 entered its CS?

a) FLAG[1] = down → This is possible only with the following interleaving:

```
p0    … F[0]←up    A_Y←0    F[1]=down                          C.S.
      --------|---------------|--------------|---------|----------|--------------->
p1                                             F[1]←up  A_Y←1    p1 cannot be in CS
```

(this is needed for p0 to find F[1] at down)

b) AFTER_YOU = 1 → This is possible only with the following interleaving:

```
p0    … F[0]←up    A_Y←0              A_Y=1                    C.S.
      --------|---------------|---------|-----------|------------------------------->
p1                               A_Y←1                        p1 cannot be in CS
```

(this is needed for p0 to find A_Y at 1)

<u>Bounded Bypass (with bound 1):</u> let p0 invoke lock.

If the wait condition is true → it wins (and waits 0)

Otherwise, it must be that FLAG[1]=up AND AFTER_YOU=0

- FLAG[1]=up → p1 has invoked lock
  → p1 will eventually pass its wait, enter in CS and then unlock

- If p1 never locks anymore → p0 will eventually read F[1] and win (waiting 1)

- If p1 locks again
  - If p0 reads F[1] before p1 locks → p0 wins (waiting 1)
  - Otherwise, p1 sets A_Y at 1 and suspends in its wait (F[0]=up ∧ A_Y=1)
    → p0 will eventually read F[1] and win (waiting 1)

Peterson algorithm (*n* processes)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

- FLAG now has *n* levels (from 0 to *n*-1)
- Every level has its own AFTER_YOU

```
Initialize FLAG[i] to 0, for all i

lock(i) :=
  for lev = 1 to n-1 do
    FLAG[i] ← lev
    AFTER_YOU[lev] ← i
    wait (∀k≠i. FLAG[k] < lev
                 OR AFTER_YOU[lev] ≠ i)
  return

unlock(i) :=
    FLAG[i] ← 0
    return
```

We say that pi is at level h when it exists from the h-th wait

→ a process at level h is at any level ≤ h

MUTEX

<u>Lemma:</u> for every $\ell \in \{0,\ldots,n-1\}$, at most n- ℓ processes are at level ℓ.

→ this implies MUTEX by taking $\ell = n-1$

Proof (by induction on ℓ)

<u>Base</u> ($\ell = 0$): trivial

<u>Induction</u> (true for ℓ, to be proved for $\ell+1$):

- p at level ℓ can increase its level by writing its FLAG at $\ell+1$ and its index in A_Y[$\ell+1$]
- Let p_x be the last one that writes A_Y[$\ell+1$] (so, A_Y[$\ell+1$]=x)
- For p_x to pass at level $\ell+1$, it must be that $\forall k \neq x.\ F[k] < \ell+1$

  → p_x is the only proc at level $\ell+1$ and the thesis holds, since $1 \leq n-\ell-1$
- Otherwise, p_x is blocked in the wait and so we have at most $n-\ell-1$ processes at level $\ell+1$ (i.e., those at level ℓ, that by induction are at most $n-\ell$, except for p_x that is blocked in its $(\ell+1)$-th wait)

Starvation Freedom

Lemma: every process at level ℓ ($\leq$ n-1) eventually wins

→ starvation freedom holds by taking $\ell = 0$

Proof (by reverse induction on ℓ)

Base (ℓ = n-1): trivial

Induction (true for ℓ+1, to be proved for ℓ):

- Assume a p_x blocked at level ℓ (i.e., blocked in its (ℓ+1)-th wait)

  → $\exists k \neq x.\ F[k] \geq \ell+1 \ \wedge\ A\_Y[\ell+1] = x$
- If some p_y will eventually set A_Y[ℓ+1] to y

  → p_x will eventually exit from its wait and pass to level ℓ+1
- Otherwise, let $G = \{p_i : F[i] \geq \ell+1\}$ and $L = \{p_i : F[i] < \ell+1\}$
  - If p ∈ L, it will never enter its (ℓ+1)-th loop (otherwise would write A_Y[ℓ+1])
  - All p ∈ G will eventually win (by induction) and move to L

    → eventually, p_x will be the only one in its (ℓ+1)-th loop, will all other processes at level < ℓ+1

    → p_x will eventually pass to level ℓ+1 and win (by induction)

Peterson algorithm (*n* processes)

Costs:

- n MRSW registers of $\lceil \log_2 n \rceil$ bits (FLAG)
- n-1 MRMW registers of $\lceil \log_2 n \rceil$ bits (AFTER_YOU)
- $(n-1) \times (n+2)$ accesses for locking and 1 access for unlocking

It satisfies MUTEX and starvation freedom.

It doesn't satisfy bounded bypass:

- Consider 3 processes, one «sleeping» in its first wait, the others alternating in the CS
- When the first process wakes up, it can pass to level 2 and eventually win
- But the sleep can be arbitrary long and in the meanwhile the other two processes may have entered an unbounded number of CSs

Easy to generalize to *k*-MUTEX (at most *k* processes simultaneously in the CS)

→ it suffices to have `for lev = 1 to n-k`