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

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 liveness: 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**: for every 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) 2*f(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) :=
    LAST ← i                                return
    wait LAST ≠ 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) :=                              unlock(i) :=
     FLAG[i] ← up                            FLAG[i] ← down
     LAST ← i                                return
     wait (FLAG[1-i] = down
            OR LAST ≠ i)
     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 MRSW registers (the flags) and 1 one-bit MRMW resister (LAST)
- Each lock-unlock requires 5 accesses to the registers

MUTEX: 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   LAST←0   F[1]=down                    C.S.
     --------|---------------|--------------|---------|----------|--------------->
p1                                F[1]←up  LAST←1    p1 cannot be in CS
                    (this is needed for p0 to find F[1] at down)
```

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

```
p0   … F[0]←up   LAST←0             LAST=1                 C.S.
     --------|---------------|---------|-----------|------------------------------>
p1                             LAST←1                    p1 cannot be in CS
                    (this is needed for p0 to find LAST at 1)
```

Bounded Bypass (with bound 1): let p0 invoke lock.

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

Otherwise, it must be that FLAG[1]=up AND LAST=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 LAST at 1 and suspends in its wait (F[0]=up ∧ LAST=1)
    → p0 will eventually read F[1] and win (waiting 1)

Peterson algorithm (*n* processes)

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

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

lock(i) :=                                        unlock(i) :=
  for lev = 1 to n-1 do                               FLAG[i] ← 0
    FLAG[i] ← lev                                     return
    LAST[lev] ← i
    wait (∀k≠i. FLAG[k] < lev
                OR LAST[lev] ≠ i)
  return
```

Peterson algorithm (*n* processes)

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

Costs:

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

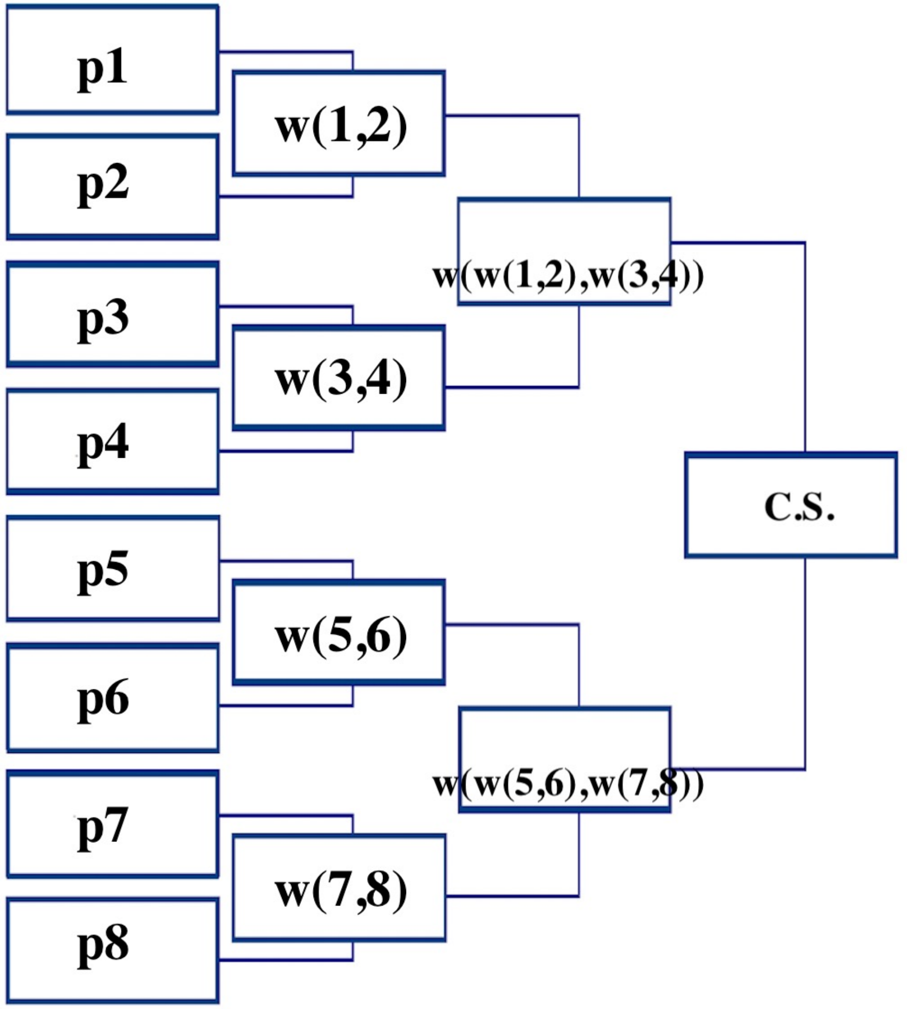
  → this quadratic cost has to be paid also when there is no contention!

Tournament-based algorithm

A first way to reduce the quadratic cost of the previous algorithm 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

Problems:

- 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
- There can be 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
```

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)

$\rightarrow$ 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 logarithmic bound)
    - 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:

```
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

For liveness, we can prove that (*bounded bypass of RR*): if a process invokes RR.lock, then it enters its CS after at most $n(n-1)$ CSs.

MUTEX with specialized HW primitives

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 basic instructions (R/W/test/sum…).

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

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

```
X.test&set() :=
        tmp ← X         ⎫
        X ← 1           ⎬ atomic (by hardware means)
        return tmp      ⎭
```

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

```
Initialize X at 0

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

MUTEX with Swap

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

```
X.swap(v) :=
        tmp ← X        ⎫
        X ← v          ⎬ atomic (by hardware means)
        return tmp     ⎭
```

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

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      } atomic
        return false
```

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

```
Initialize X at 0

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

MUTEX with Fetch&add

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

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 TICK[i] to 0, for all i

lock(i) :=                                              unlock(i) :=
    FLAG[i] ← up                                             TICK[i] ← 0
    TICK[i] ← max{TICK[1],…,TICK[n]}+1
    FLAG[i] ← down
    forall j ≠ i
        wait FLAG[j] = down
        wait (TICK[j] = 0 OR ⟨TICK[i],i⟩ < ⟨TICK[j],j⟩)
```

The algorithm satisfies MUTEX and bounded bypass with $f(n) = n$-1.

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

Aravind's algorithm (2011)

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
```

This algorithm can be proved to satisfy MUTEX and bounded bypass with $f(n) = 2(n\text{-}1)$ (actually, reducible to $f(n) = n\text{-}1$ with a small modification in the `unlock`)