

# Real-time systems: modelling languages

**Claire Pagetti**

[claire.pagetti@enseeiht.fr](mailto:claire.pagetti@enseeiht.fr)

ENSEEIHT - Département Télécommunication et Réseaux  
2, rue Camichel, 31000 Toulouse

# Course Objectives

## 1. Introduction to real-time

- What is a real-time system?
- Definition of classical notions and terms for real-time
- WCET computation

## 2. Formal real-time languages : zoom on synchronous / asynchronous modelling languages

- Lustre
- SCADE (commercial version of Lustre)
- SDL (System Description Language)

## 3. Uniprocessor and multiprocessor scheduling

- Scheduling policies
- Scheduling analyses

# Course organisation

- **Lecture 1 : introduction**
  - **Lecture 2 & practical sessions 3, 4: Lustre**
  - **Lectures 5, 8 & practical sessions 6, 9: SCADE**
  - **Lectures 7, 12 & 13: scheduling**
  - **Lecture 10 & practical session 11: SDL**
  - **Practical session 14: scheduling analyses**
- } François-Xavier  
Dormoy (Esterel Tech)

## Evaluation:

- Commented code of practical sessions 3 & 4
- Code and report (max 2 pages) of practical session 11

# Outline

1. **Part I - What is a real-time system?**
2. **Part II – High level formal programming languages**
3. **Part III – Uniprocessor and multiprocessor scheduling**

# Outline - Part I What is a real-time system?

## **1. First definitions**

1. Definitions
2. A real example

## **2. General architecture**

## **3. WCET computation**

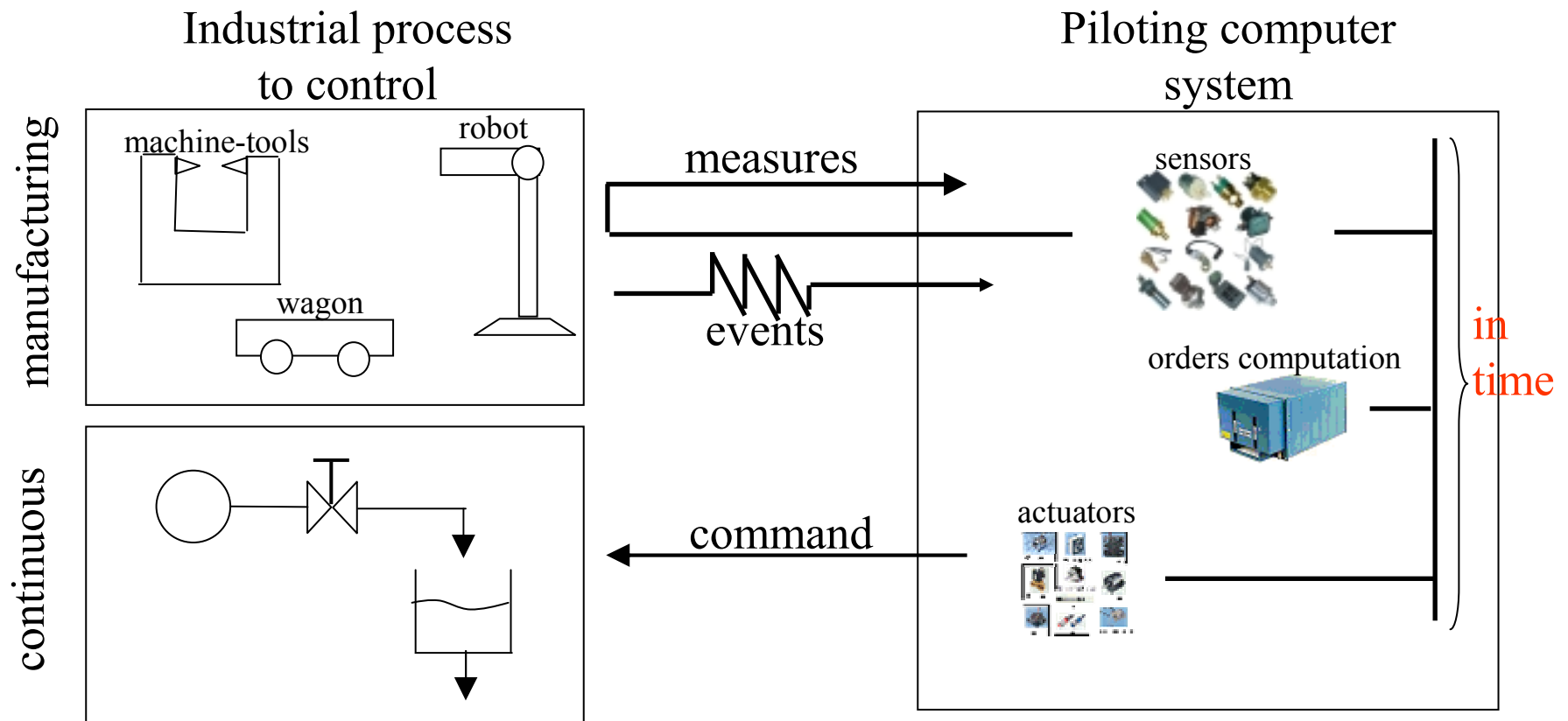
## **4. Real-time problems**

## 1.1 Introduction

- **Most of human made machines require a control or regulation system to work correctly (and safely)**
- **These kinds of systems existed long before the invention of computers**
- **Example:**
  - To maintain a locomotive at constant speed, the system regulates the amount of steam. Downhill, it should inject less steam and uphill, it should inject more.
- **Automation: reduce human intervention**

## 1.1 First definitions

Is called *real-time* the behavior of a computer system subjugated to the dynamic evolution of a process connected to it. This process is controlled, piloted or supervised by the system which *reacts* to the state changes of the process.



## History – early 60s

- The first notable embedded real-time system was the *Apollo Guidance Computer*, onboard computation for guidance, navigation, and control of the spacecraft of the Apollo program. It has been developed by Charles Stark Draper of Massachusetts Institute of Technology.
- First computer to use integrated circuits
- Used in real-time by the astronauts to collect and provide flight data, and to automatically control the navigation functions of the spacecraft.
- 16-bit wordlength memory : composed of 64 ko (32 000 words) ROM containing all the programs and of 4 ko (2 042 words) RAM for the computations. The processor weights around 35 kg.



Source: [http://fr.wikipedia.org/wiki/Apollo\\_Guidance\\_Computer](http://fr.wikipedia.org/wiki/Apollo_Guidance_Computer)



## **Embedded system market**

**“Over 4 billion embedded processors were sold last year and the global market is worth €60 billion with annual growth rates of 14%. Forecasts predict more than 16 billion embedded devices by 2010 and over 40 billion by 2020.**

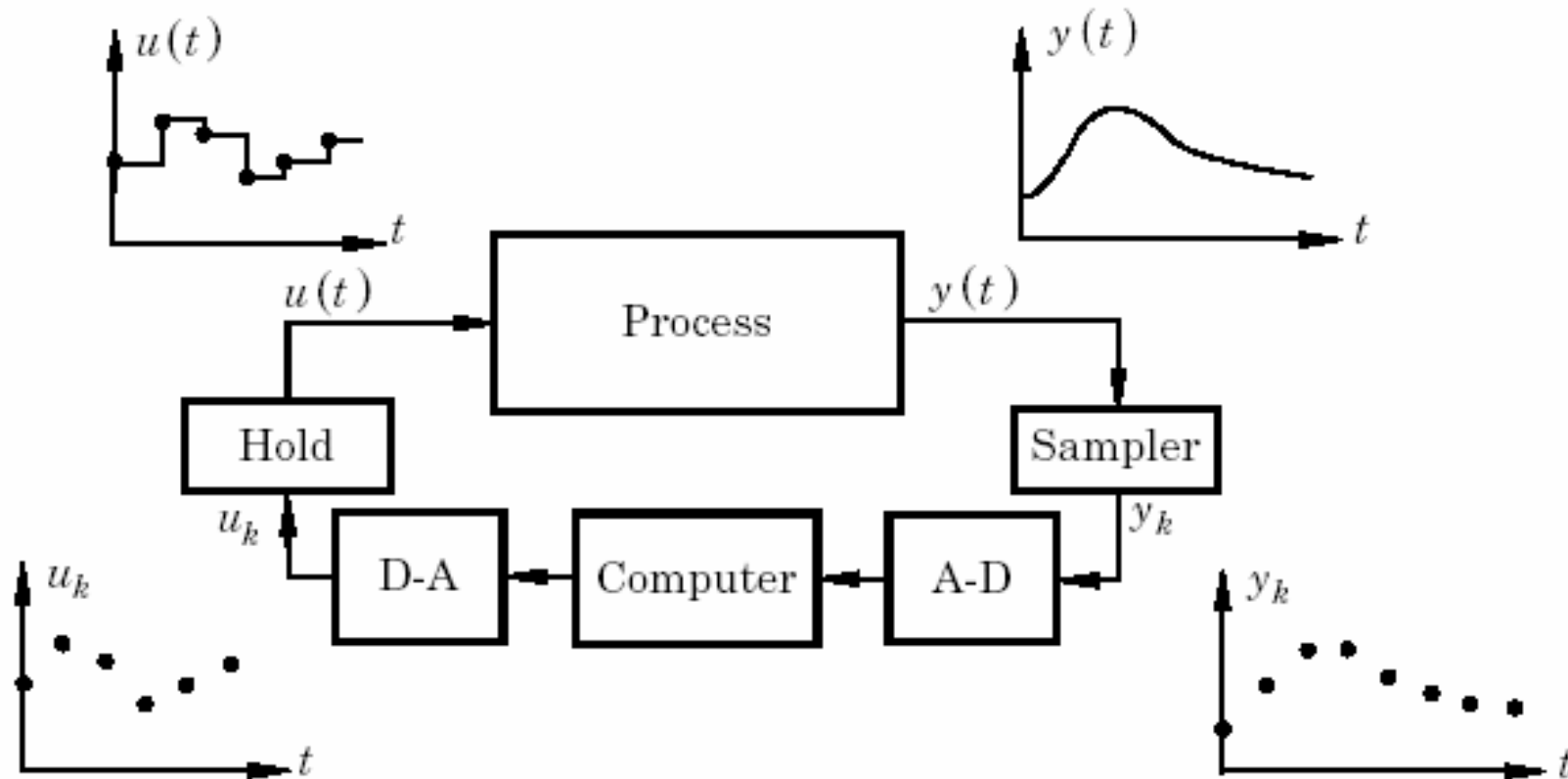
**Embedded computing and electronics add substantial value to products. Within the next five years, the share of embedded systems are expected to increase substantially in markets such as automotive (36%), industrial automation (22%), telecommunications (37%), consumer electronics (41%) and health/medical equipment (33%). The value added to the final product by embedded software is much higher than the cost of the embedded device itself. For example, in the case of a modern car, by 2010 over 35% of its value will be due to embedded electronics. ”**

**[ARTEMIS JU Organisation - 2009]**

## Some examples

- command and control in production line
- guiding of mobile systems (robotics...)
- embedded systems (railways, aeronautics, automotive...)
- supervising of physical reactions or phenomena (nuclear, chemistry,...)
- computer-assisted instrumentations and operations (medical...)
- communication systems (multimedia...)
- dedicated systems (scientist experience, signal processing...)

## Example: sample control loop



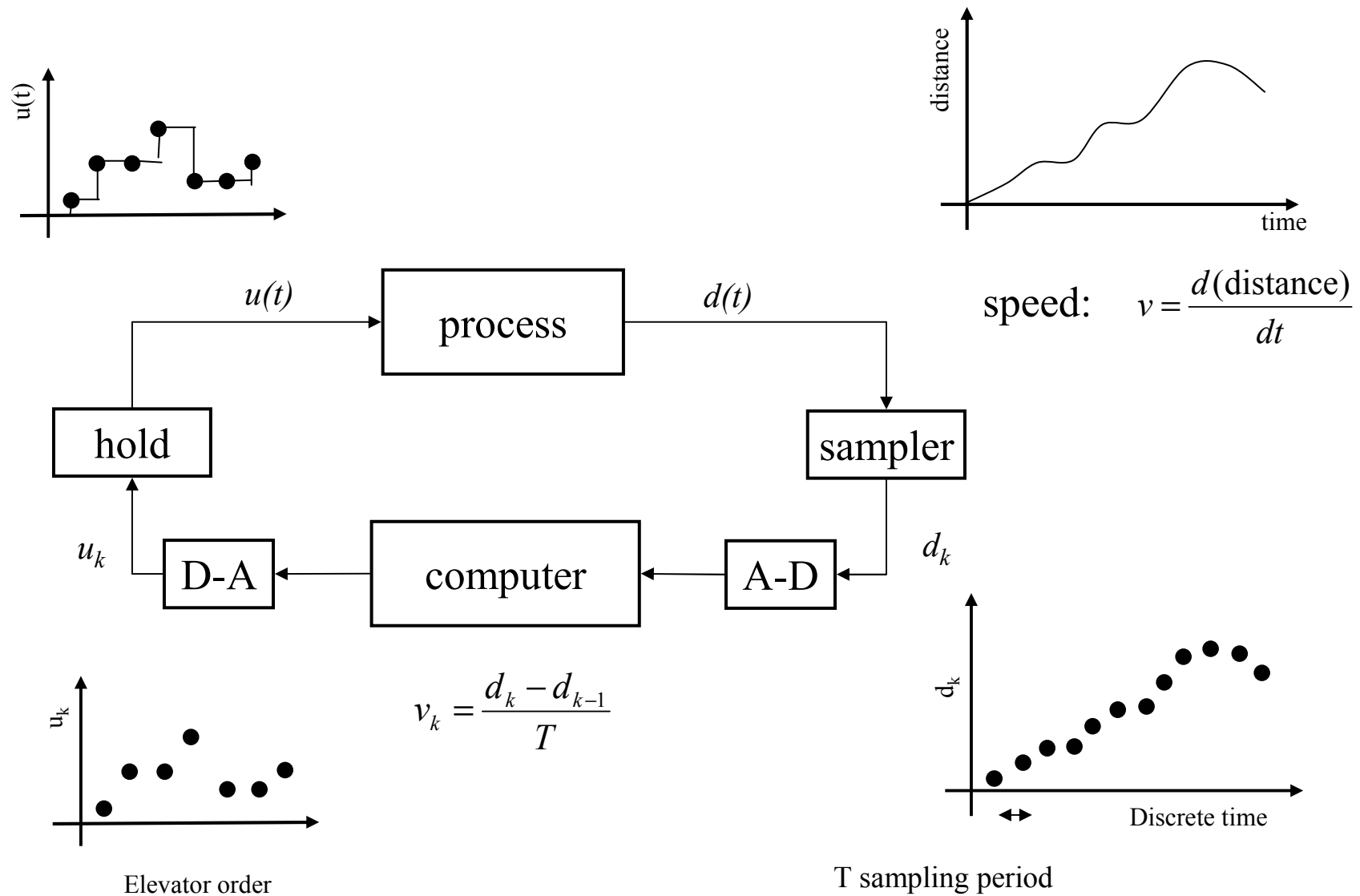
A-D : input data collection

D-A: output signal transmission

$y$  : process output

$u$  : new control signal

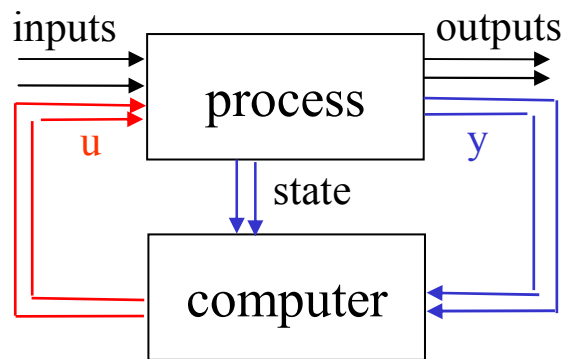
# Example: sample control loop



## Example : control-command system

**Command:** Laws which govern the dynamical evolution of a system

- Command a system = make the system evolve in order to reach a particular configuration or to follow a given trajectory



Equations of state

$$\begin{cases} dx = f(x, u) \\ y = h(x) \end{cases}$$

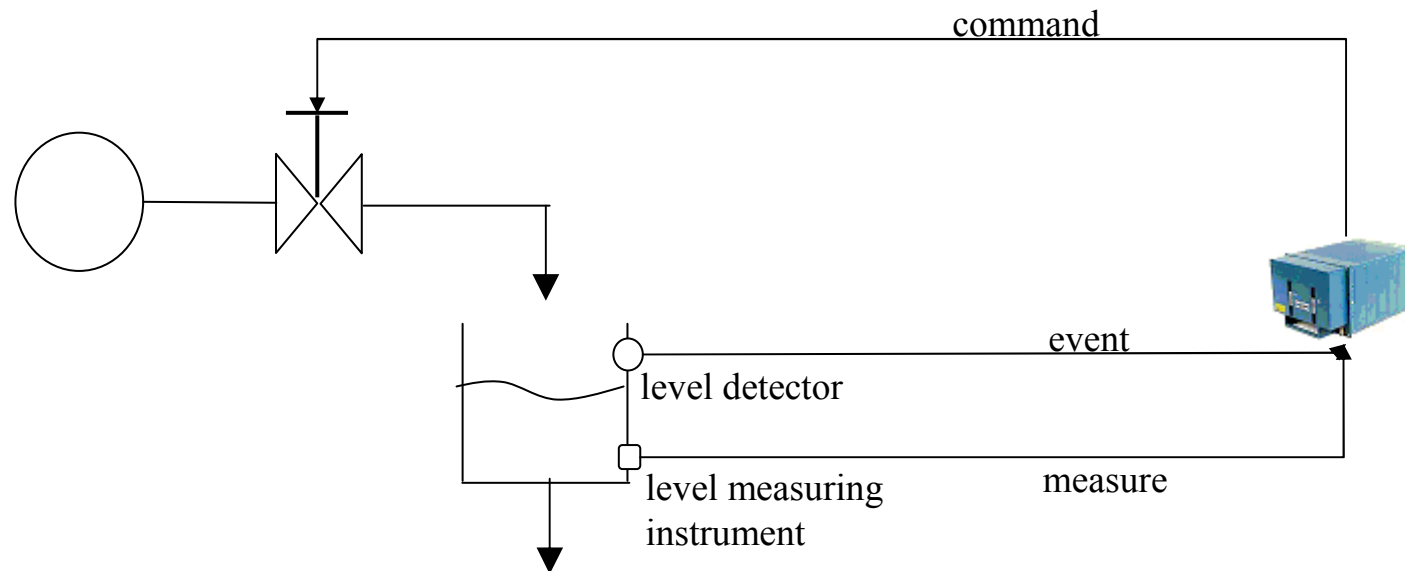
**Control:** Laws that govern the behaviour of a system

- choice between several commands depending on the environment and the system
- series of command
- realisation of a mission and monitoring of its correct progress
- discrete time

## Example: command of a continuous process

Regulation of a liquid level:

- control the level in order to respect an order of the operator
- respect the thresholds



## Real-time System

The terminology *real-time* hides the notion of time of reaction relative to the dynamics of the industrial process to control. Indeed, the computer system must react in constraint time to the evolution and the delay of the process reactions is conditioned by the internal dynamics and the cadence of production. (reactive system)

In other words: a system works in real-time if it is able to absorb all the inputs not too old and reacts in time such that the reaction is adapted to the current state.

An absence of reaction can lead to a catastrophic situation (critical system).

# What time is it?

Two entities: the plant (or physical process) to control and the real-time computer system

⇒ Two times: the time of the environment and the time of the real-time system

- **Environment time** = **chronometric** time (the real time)
- **Computer system time** = **chronological** time, composed by the sequence of events or instructions of the system (steps of the real time seen by the system)

⇒ Requirements of real-time = concordance between the chronometric time of the environment and the chronological time of the computer system.

⇒ The computer system must put its actions in phase with the chronometric time of the process

⇒ the actions of the system will be tasks and messages. Use of techniques of scheduling of tasks and communications



## Real-time $\neq$ go quick

- Need of a short reaction (1 ms) for the control of an military aircraft
- Need of a slower reaction (10 ms) for the control of a civil aircraft
- Need of a slower reaction (1s) for an HCI (human-computer interaction)
- Need of a slower reaction (1mn) for the control of a production line
- Need of a slower reaction (1h) for the control of chemical reaction
- ...
- Need of a reaction of several hours to make a meteorological prediction
- Need of a reaction of several days for computation of the pay of the employees...

=> Respect the deadline is essential

=> A correct result out of deadline may be unusable and may generate a fault.

# Hard and soft real-time

## Two notions of criticality when a deadline is not respected

- **Strict** real-time constraints : the exceed of a deadline is catastrophic
- **Relatives** real-time constraints: the exceed of some deadlines may be tolerated

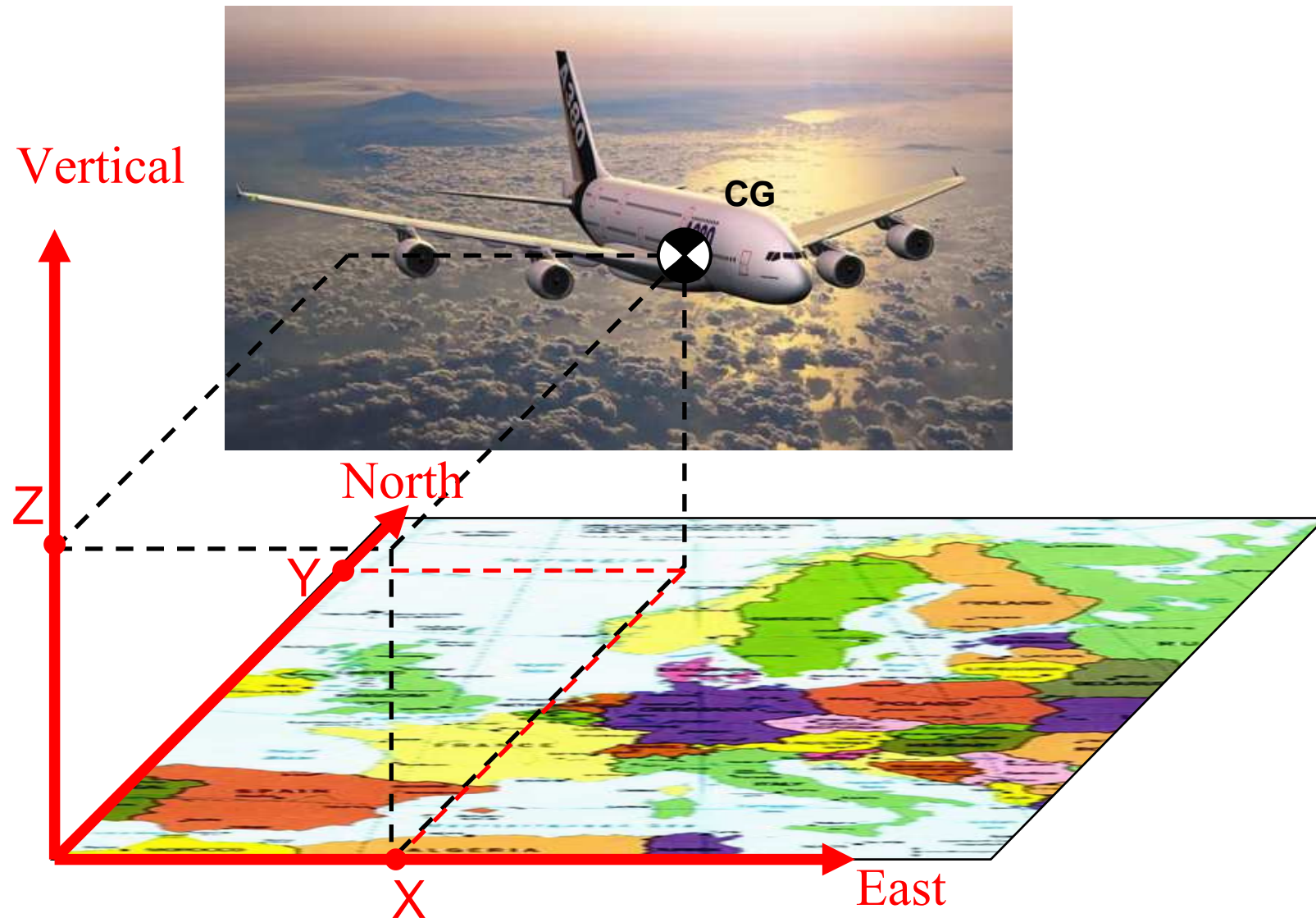
=> **Hard real-time** / **soft real-time**

## Consequences...

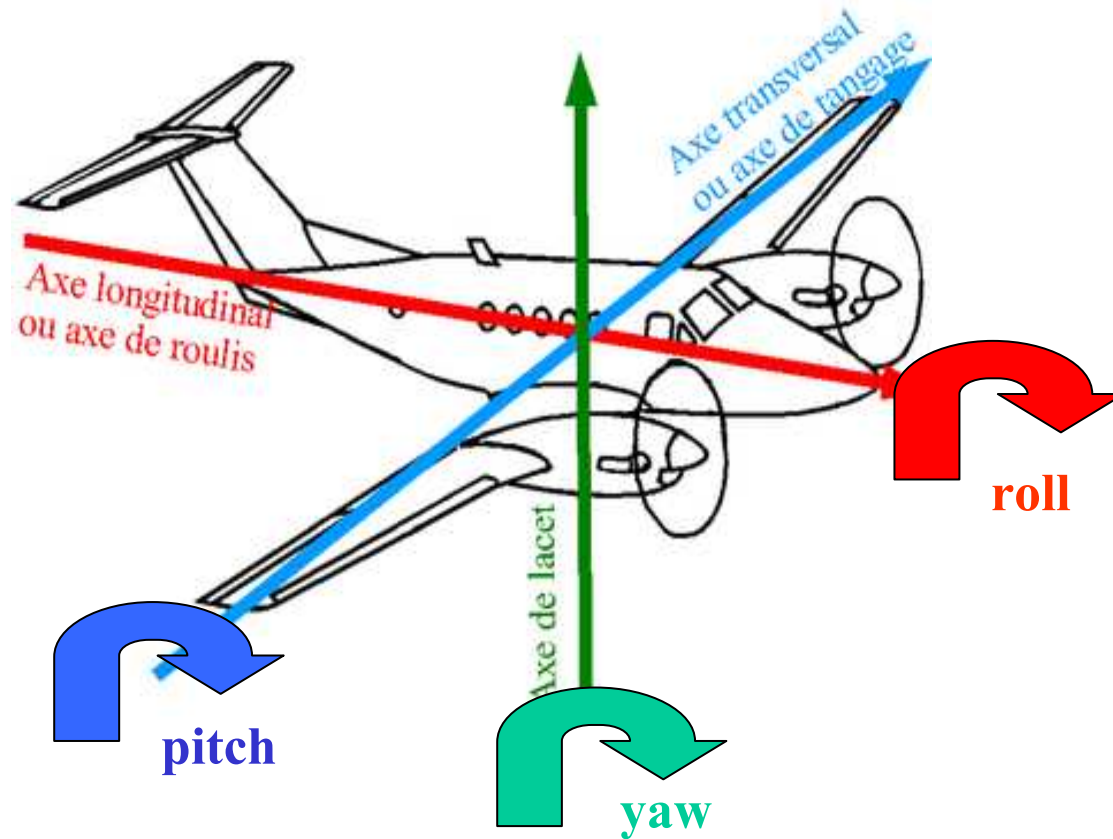
- => In the case of hard real-time systems, conceivers want to be **predictable**, **deterministic** and **reliable**
  - => Use of mathematical techniques (scheduling, worst case evaluation...)
- => In the case of soft real-time systems, conceivers want to minimise the probability to miss a deadline several times

## 1.2 Example of hard real-time: aircraft flight control

From physics to a computer-aided navigation ...



# Aircraft attitude



pitch



roll



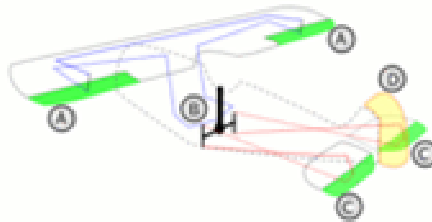
yaw

# Flight control system

The **aircraft primary flight control system** is the set of elements between the stick and the surfaces which aim at controlling the attitude, the trajectory and the speed of the aircraft.

The system is composed of:

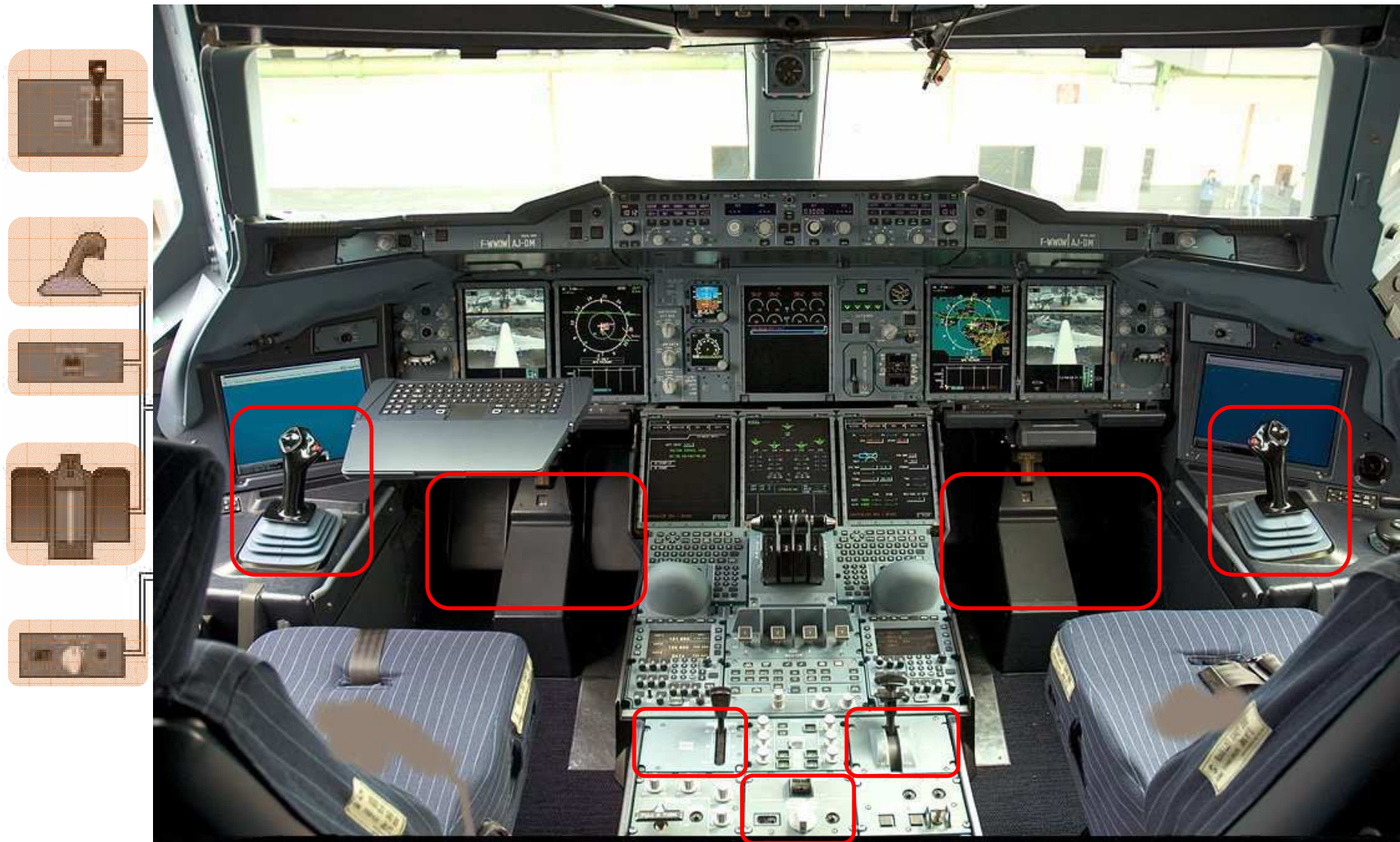
- piloting elements: stick (or yoke or control column), pedals, throttle controls...



- transmitting and computing organs
  - cables and calculators (for fly-by-wire)
- sensors, actuators and servocommand to command the surfaces



# Fly-by-wire flight control system



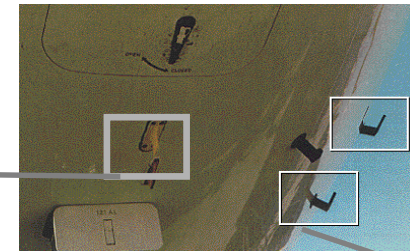
## (Some) Aircraft sensors

- GPS
- altimeter: measures the altitude of an object above a fixed level
- inertial measurement unit is an electronic device that measures and reports on a craft's velocity, orientation, and gravitational forces, using a combination of accelerometers and gyroscopes.



An attitude indicator, also known as gyro horizon or artificial horizon, is an instrument used in an aircraft to inform the pilot of the orientation of the aircraft relative to earth. It indicates pitch and roll.

A weather vane, also known as a peach patrolwind vane or weathercock, is an instrument for showing the direction of the wind.



A pitot tube is a pressure measurement instrument used to measure fluid flow velocity



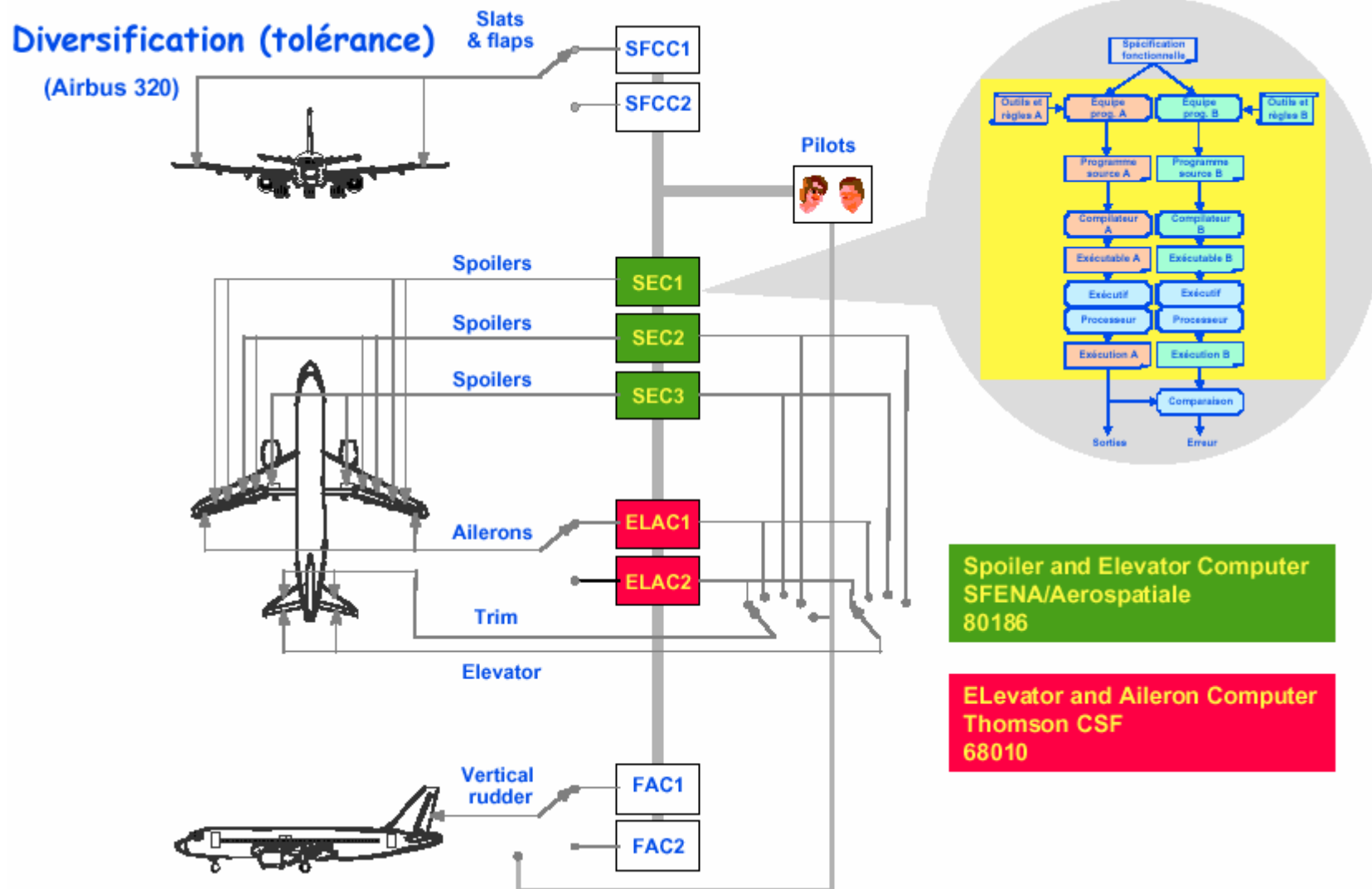
# A320 flight controls systems

## Fly-by-wire system

- 9 calculators
  - Functions allocation :
    - 2 redundant calculators for the slats and the flaps (SFCC1-2)
    - 2 redundant calculators for the rudder (FAC1-2)
    - 3 redundant calculators the spoilers, the elevators and the trim (SEC1-2-3)
    - 2 redundant calculators for the ailerons, the elevators and the trim (ELAC1-2), replaced in case of failure by the SEC1-2-3
  - Safety requirements
    - each calculator must be "fail-silent"
    - each calculator must have a failure rate less than  $10^{-3}$  per flight hour
    - ...

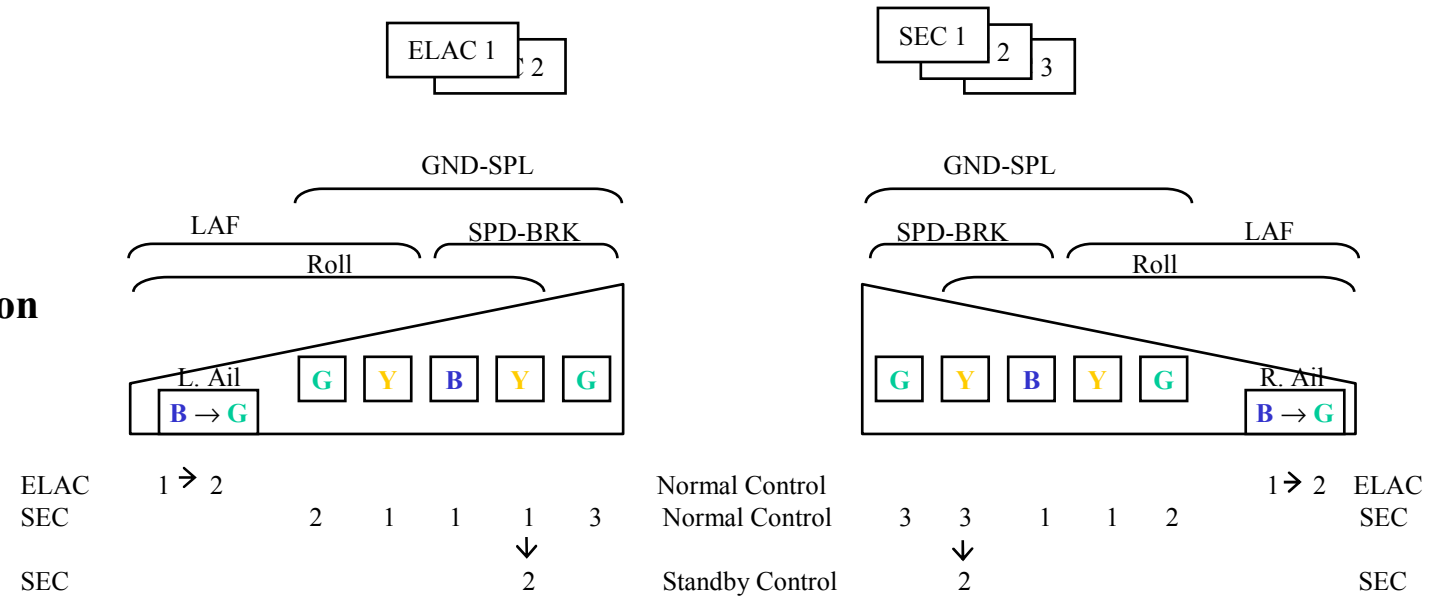


# A320 flight controls systems architecture



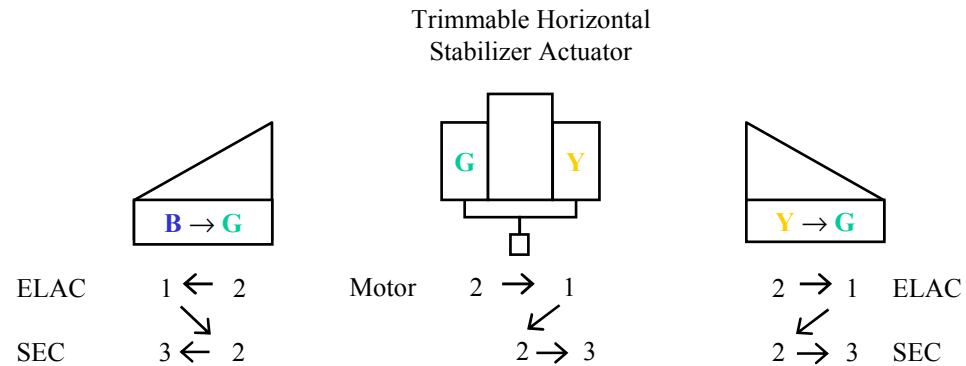
# Reconfiguration policies

## Redundancies allocation



**B** Hydraulic blue system  
**G** Hydraulic green system  
**Y** Hydraulic yellow system

GND-SPL: Ground Spoiler  
 SPD-BRK: Speed Brake  
 LAF: Load Alleviation Function



# Outline - Part I What is a real-time system?

## 1. First definitions

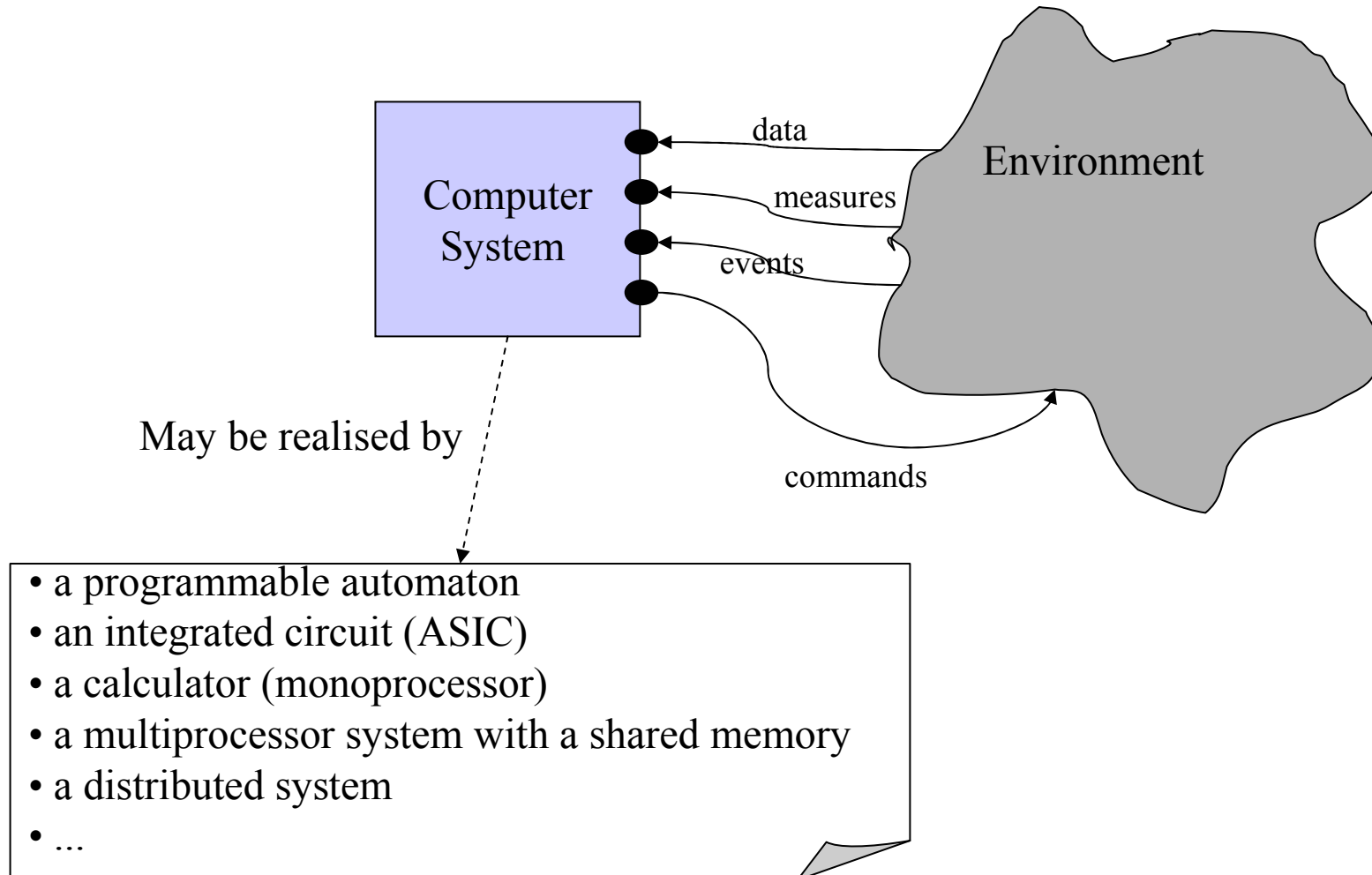
## 2. General architecture

1. Material architecture
2. Functional behaviour
3. Executive support

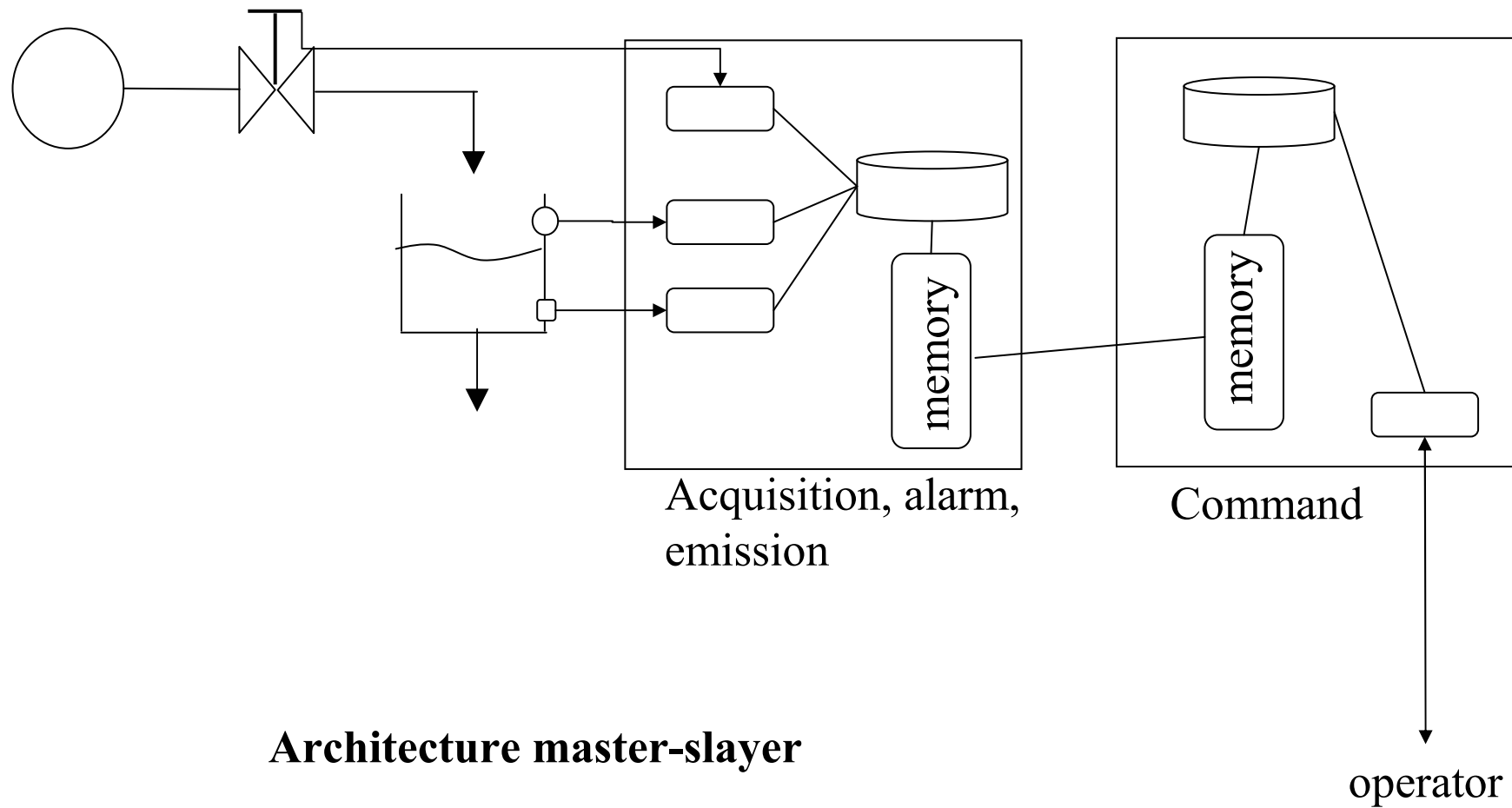
## 3. WCET computation

## 4. Real-time problems

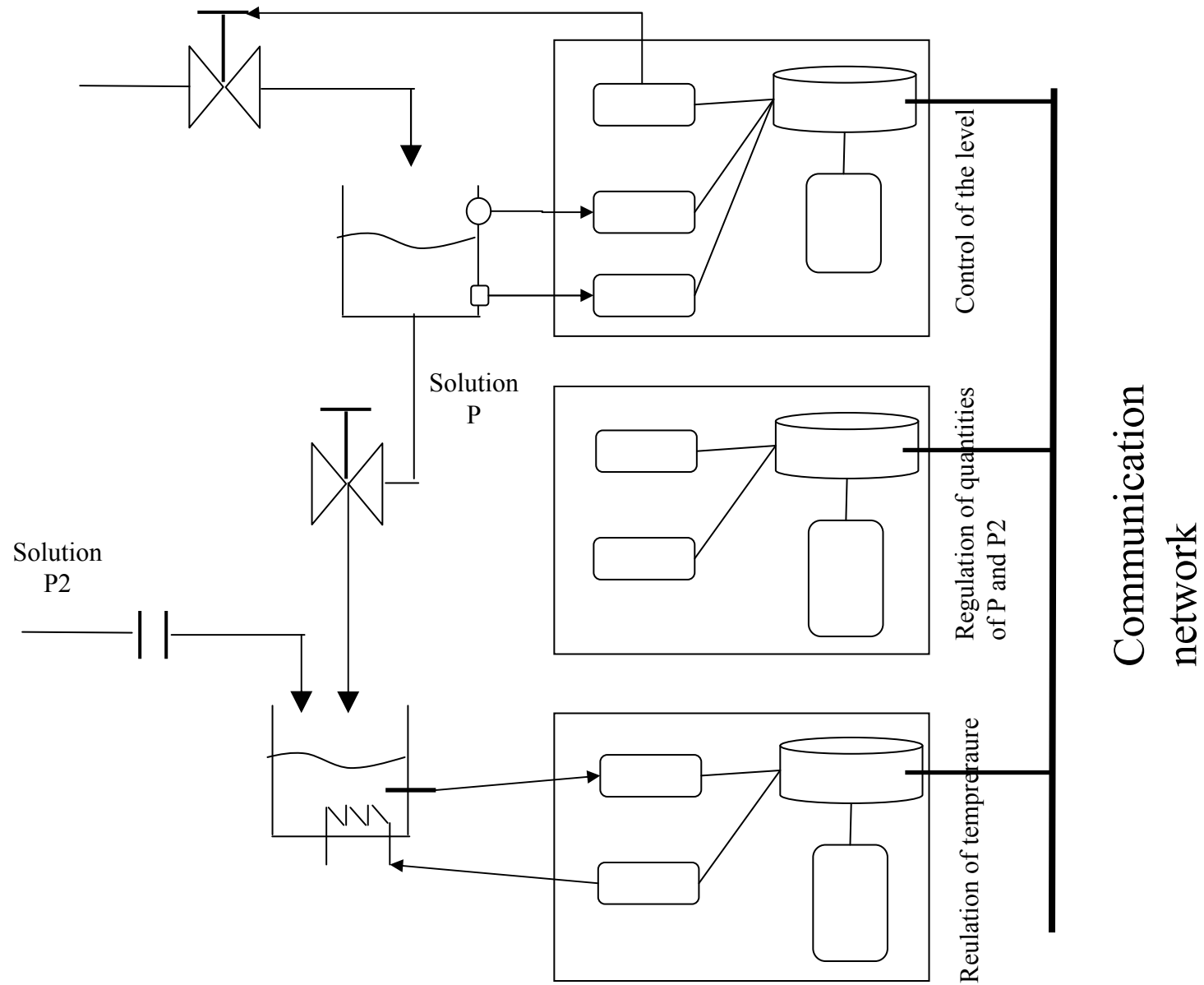
## 2.1 Embedded real-time architectures



# Multiprocessor architecture



# Distributed architecture



## 2.2 Functioning

### General functioning : infinite loop

```
While TRUE do
  Inputs acquisition(measures...)
  Computation of orders to send to the process
  Orders emission
End while
```

### But, two kinds of functioning :

- **cyclical** running (time driven or synchronous system)
- **event** running (event driven)

=> mixed functioning : based on cyclic and aperiodic treatments

# Cyclic run

## Cyclic scanning of an input memory (polling)

- => Sampling of inputs on the system clock
- => Activation at each tick of the clock

```
At each clock tick do
    Read of the input memory
    Orders computation for the process
    Emission of orders
End
```

But :

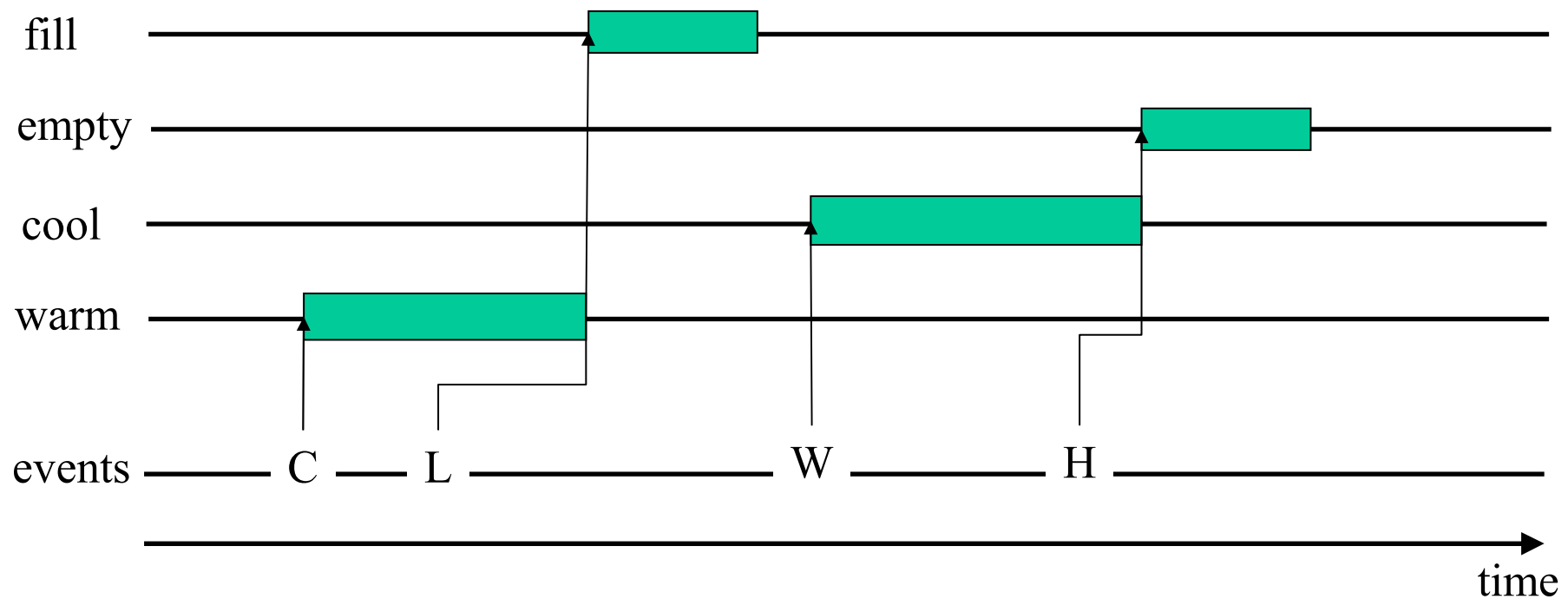
- system not much "reactive" if the environment produces information at different frequencies
- => need to foresee all possible reactions of the system in the loop
  - => bad performance
- => or interleaving of loops at different frequencies
  - => difficulties for realisation, code readability, evolution



# Chronogram of cyclical implementation

## Regulation of the level and the temperature of liquid

- Events: H (for high), L (for low), C (for cold) and W (warm)



# Event run

**Activation of the system at each event occurrence (=> notion of interruption)**

```
At each interruption do
  Read of the new information
  Activation of the correspondent treatment
  Emission of orders
Fin
```

**But : what happens when an interruption appears when the system is treating the precedent interruption?**

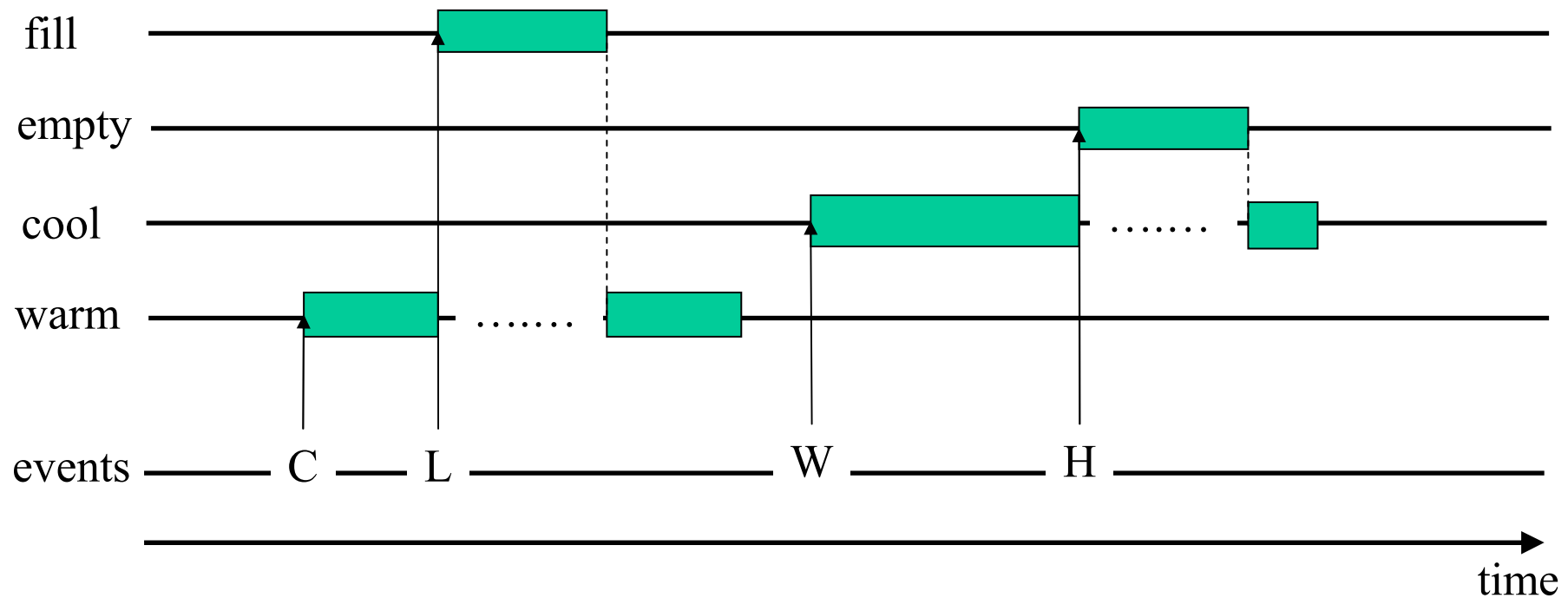
- => Priority among the interruptions
- => notion of tasks associated to interruptions
- => mechanism of preemption and resumption of tasks
- => management of concurrent execution of tasks (scheduling)

**=> A real-time system is often a multitasks system including a scheduler**

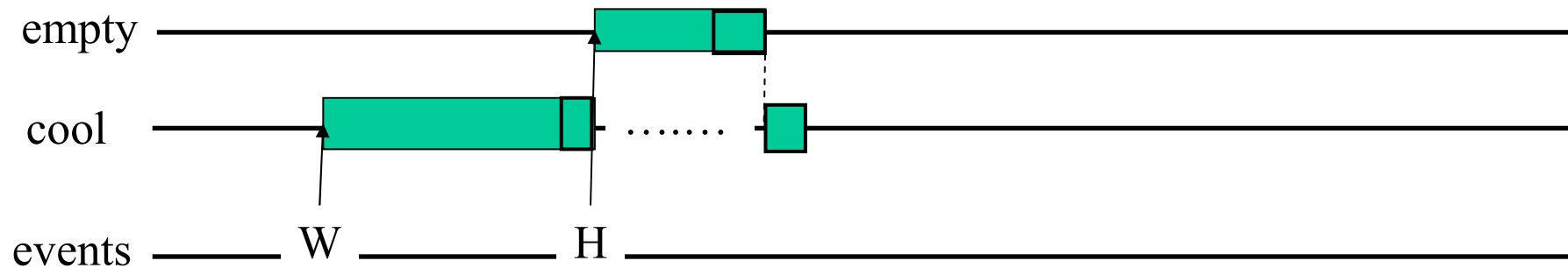
# Chronogram of event driven implementation

## Regulation of the level and the temperature of liquid

- Events: H (for high), L (for low), C (for cold) and W (warm)
- Priorities: 2, 2, 1, 1

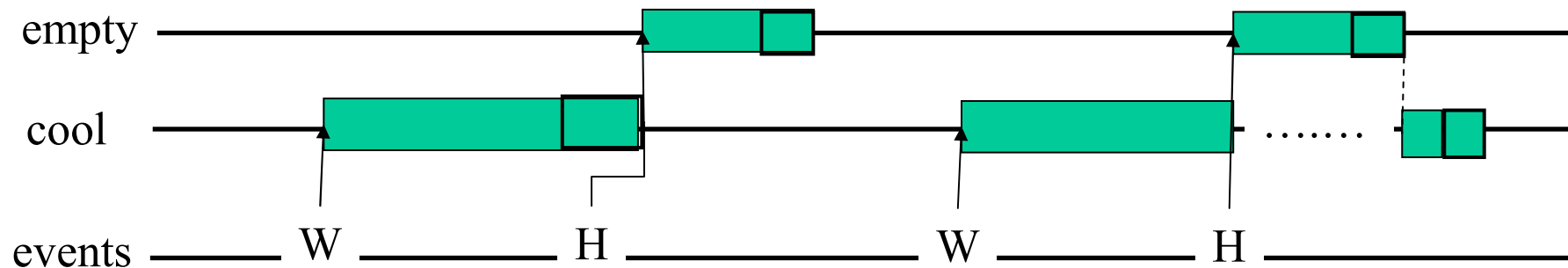


## Shared resources



when interrupted, the task cool was acceding to a shared resource (for instance memory). Conflict problem.

=> A solution: interdiction to preempt a task when using a shared memory.



## Example: Sojourner

The Sojourner rover was the second space exploration rover to successfully reach another planet, and the first to actually be deployed on another planet. Sojourner landed on Mars as part of the Mars Pathfinder mission on July 4, 1997.



### Priority inversion:

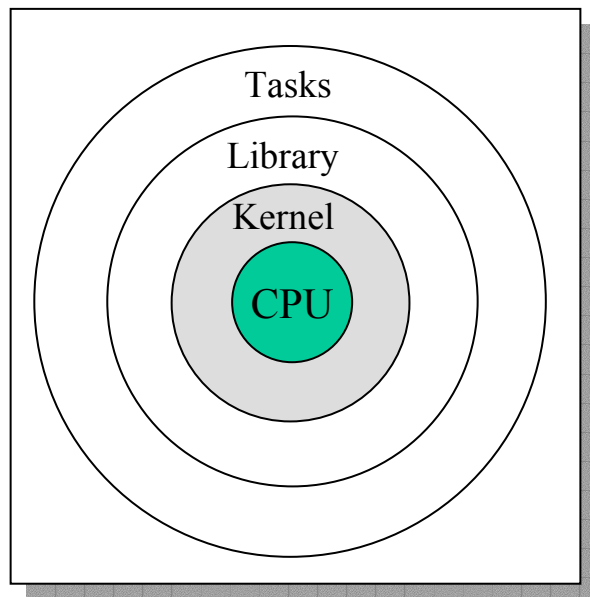
**“Even though NASA knew the problem, because it already occurred on all the tests that had been performed. But NASA thought it won’t be a problem because on earth the situation didn’t occur very often, NASA only underestimated the number of situations the problem appeared.”**

[\[http://en.wikipedia.org/wiki/Sojourner\\_%28rover%29\]](http://en.wikipedia.org/wiki/Sojourner_%28rover%29)

**Robert Franz. Advanced Course Seminar Analysis of computer bugs, Priority Inversion Mars Sojourner. 2008.**

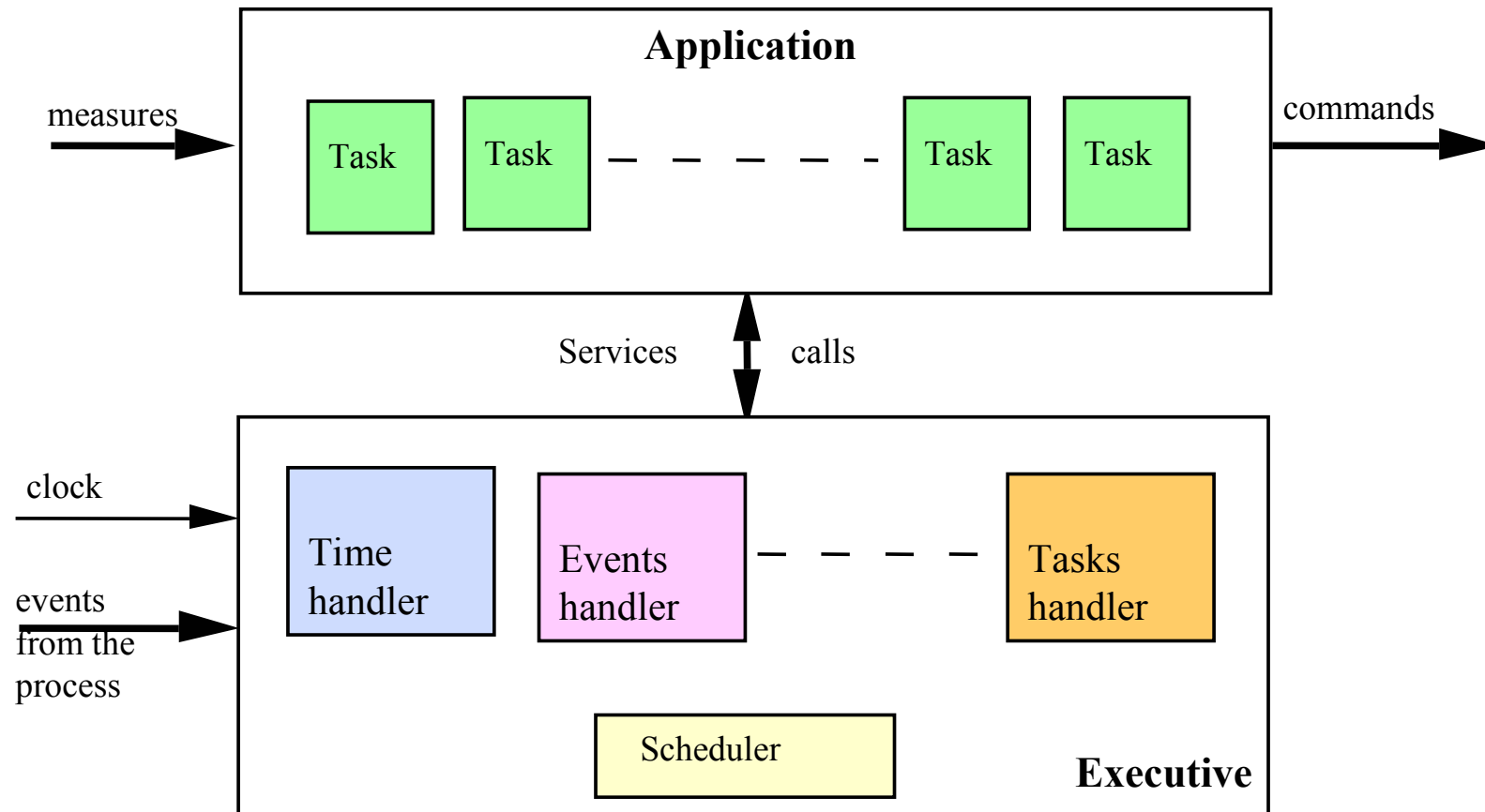
## 2.3 Real-time system support

**Simpler solution: nude application on the calculator**



Polling of the data  
Library with deterministic functions  
Sequencing by hand of tasks

## Solution with an executive component

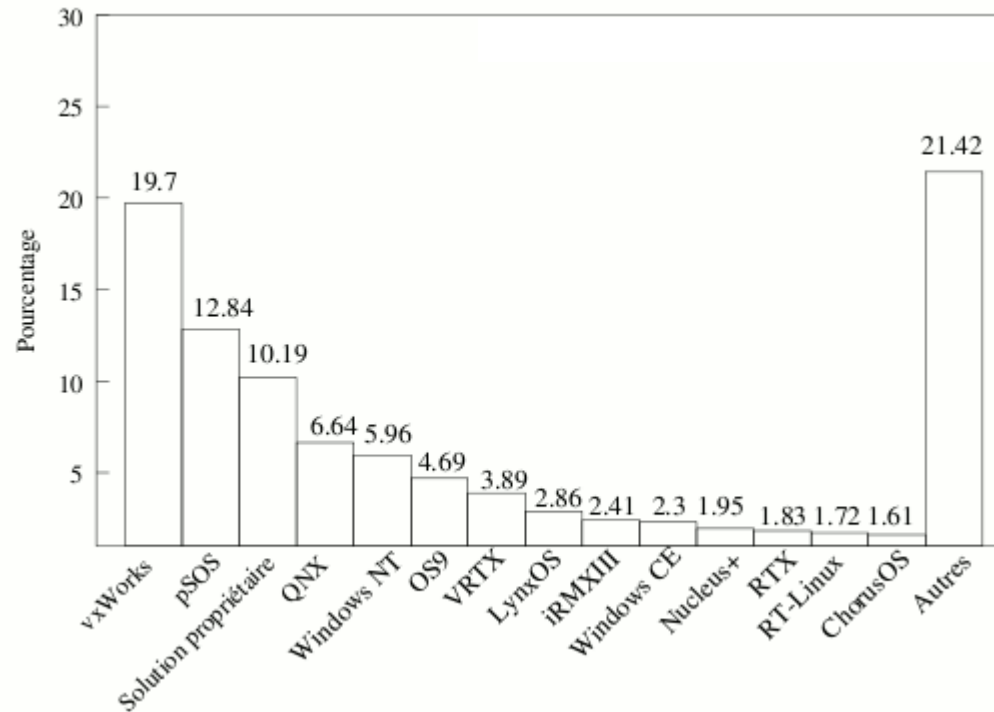


# Real-time OS

- **OS for hard real-time**
  - Precise management of priorities
  - Rapid system primitives, in bounded time (interruptions, semaphores...)
  - No virtual memory
  - Minimisation of the « overhead » (time for the system to execute and monitor its own behaviour)
- **OS for soft real-time**
  - Extension of classical OS (such as UNIX..)
  - Extension of real-time scheduling
  - Light processes
  - Preemption ...



## Market for the real-time OS

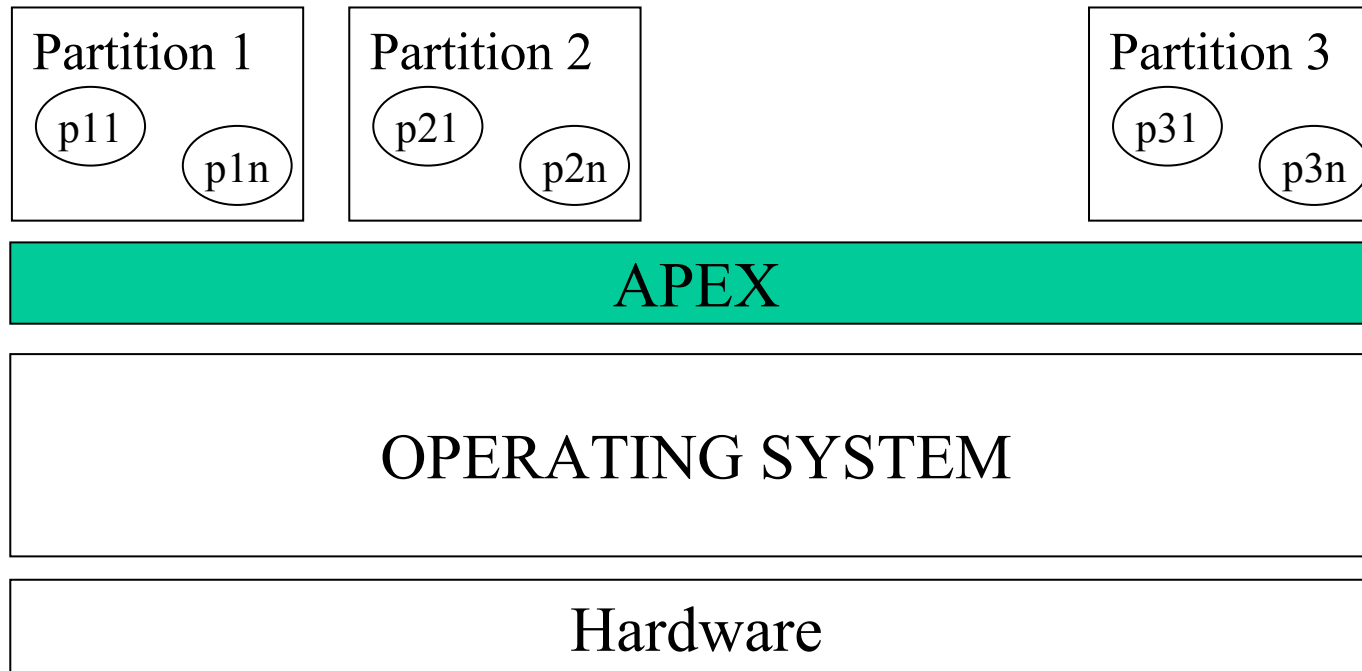


[TIM00] M. Timmerman. « RTOS Market survey : preliminary result ». Dedicated System Magazine, (1):6–8, January 2000.

## **Example: APEX Arinc 653**

- **Arinc (Aeronautical Radio, Incorporated) established in 1929, is the leading provider of transport communications and system engineering solutions**
- **Publication of the first standard ARINC 653 : summer 1996**
- **Start of the works : 1991**
- **Arinc 653: Application programming interface (API) between an OS of an avionics resource and an application. Airbus implementation interface APEX (APplication / EXecutive)**

# A653 functioning



## Operating system :

- Schedule the module partitions
- Schedule the processes of each partition
- Ensure segregation (partitioning) spatially and temporally

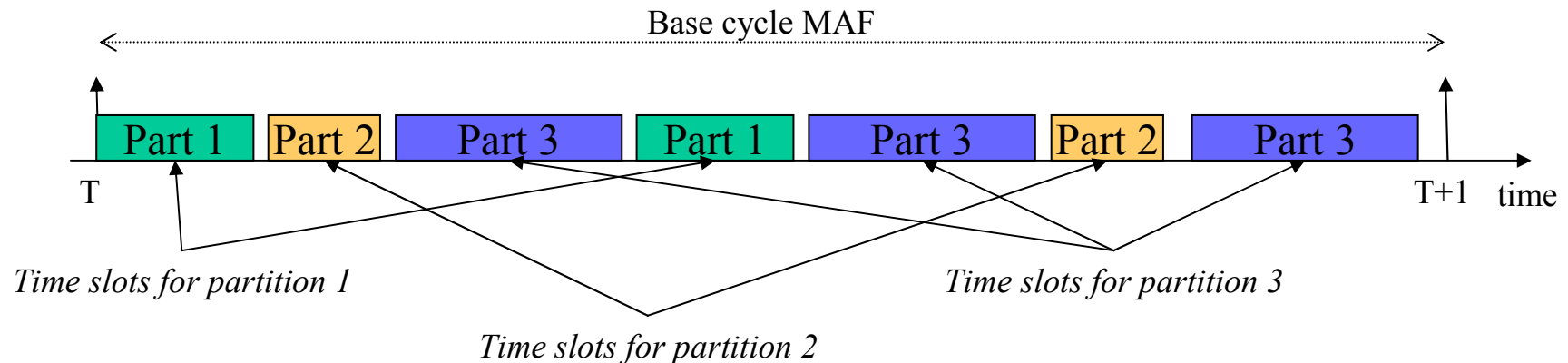
# Sharing of resources

## Spatial segregation:

Memory zone predetermined for each partition

## Temporal segregation (CPU) :

- A partition does not have any priority
- Deterministic and cyclical allocation of the CPU to the partitions :
  - => OS repeats a base cycle (MAJor time Frame: MAF) of fix duration
  - => Allocation of time slots in the MAF to each partition



**=> The resource allocation is statically defined off-line and cannot be modified**

# Partitions handling

- **OS starts the partitions scheduler**
- **During a time slot, a partition is in one of the following states:**
  - Idle : no processus is executed
  - Cold\_start ou Warm\_start : initialisation of the partition
    - Cold start : initial values
    - Warm start : recovery of the context of the partition
  - Normal : execution of the processes
- **Out of its slices, a partition is suspended**
- **All the objects (ports, processes ...) are created during the initialisation phase**

# Processes handling

## Standard

- **2 types of processes: periodic and aperiodic**
- **No segregation between processes of a same partition**
- **The processes of a partition are not visible by other partitions**
- **A unique process is running at a time**
- **Several scheduling are proposed for the processes**

## **Example: Paparazzi project**

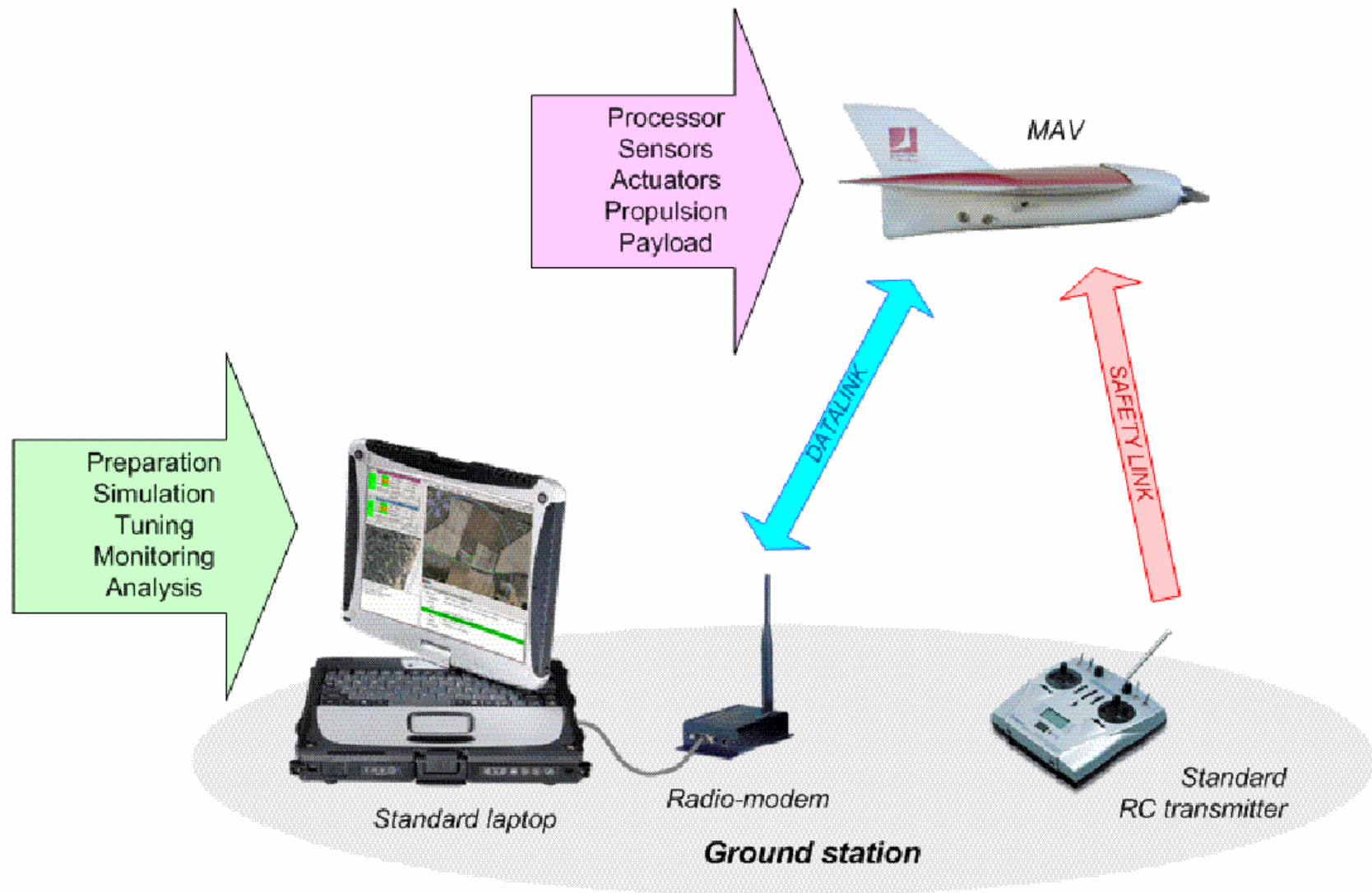
**Paparazzi is a free and open source project for programming UAV (Unmanned Air Vehicle System). Developed at ENAC.**

[http://paparazzi.enac.fr/wiki/Main\\_Page](http://paparazzi.enac.fr/wiki/Main_Page)

**The environment provides the code for:**

- airframe**
  - airframes ranging from 30cm to 1.4m, and 180g to 1.4 kg
- flight simulator**

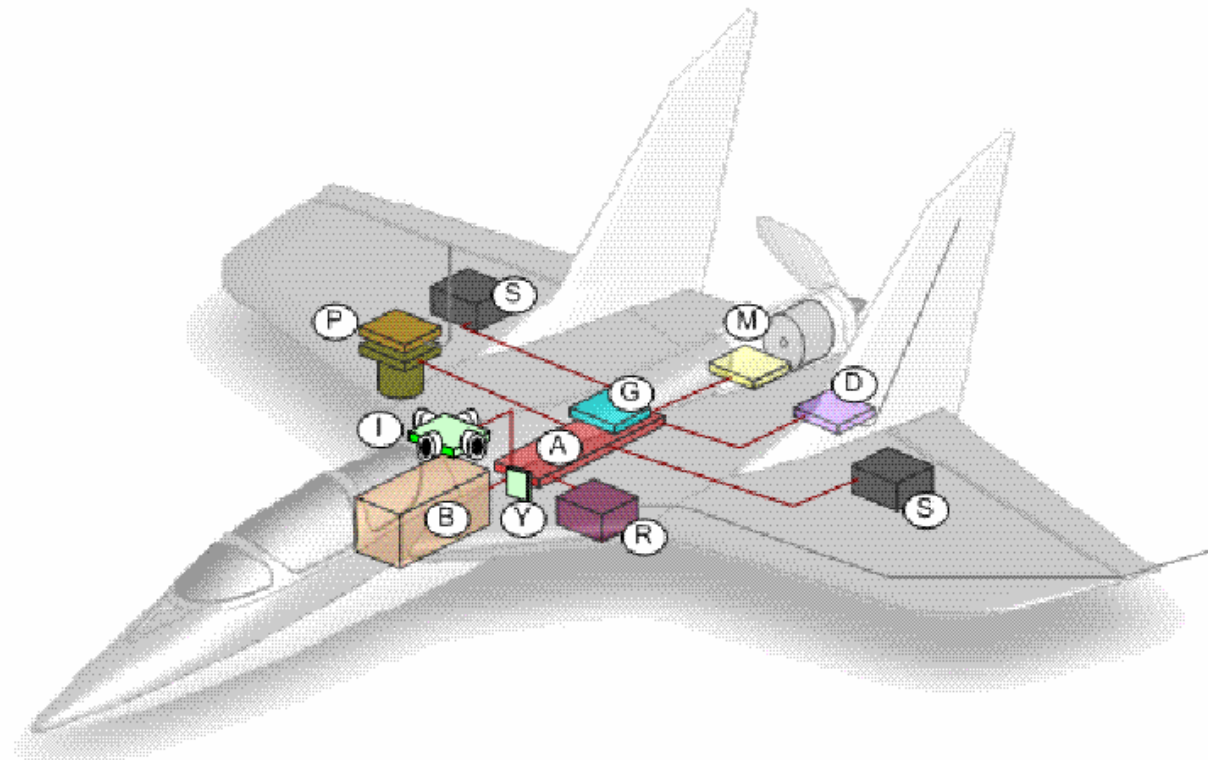
# Paparazzi project



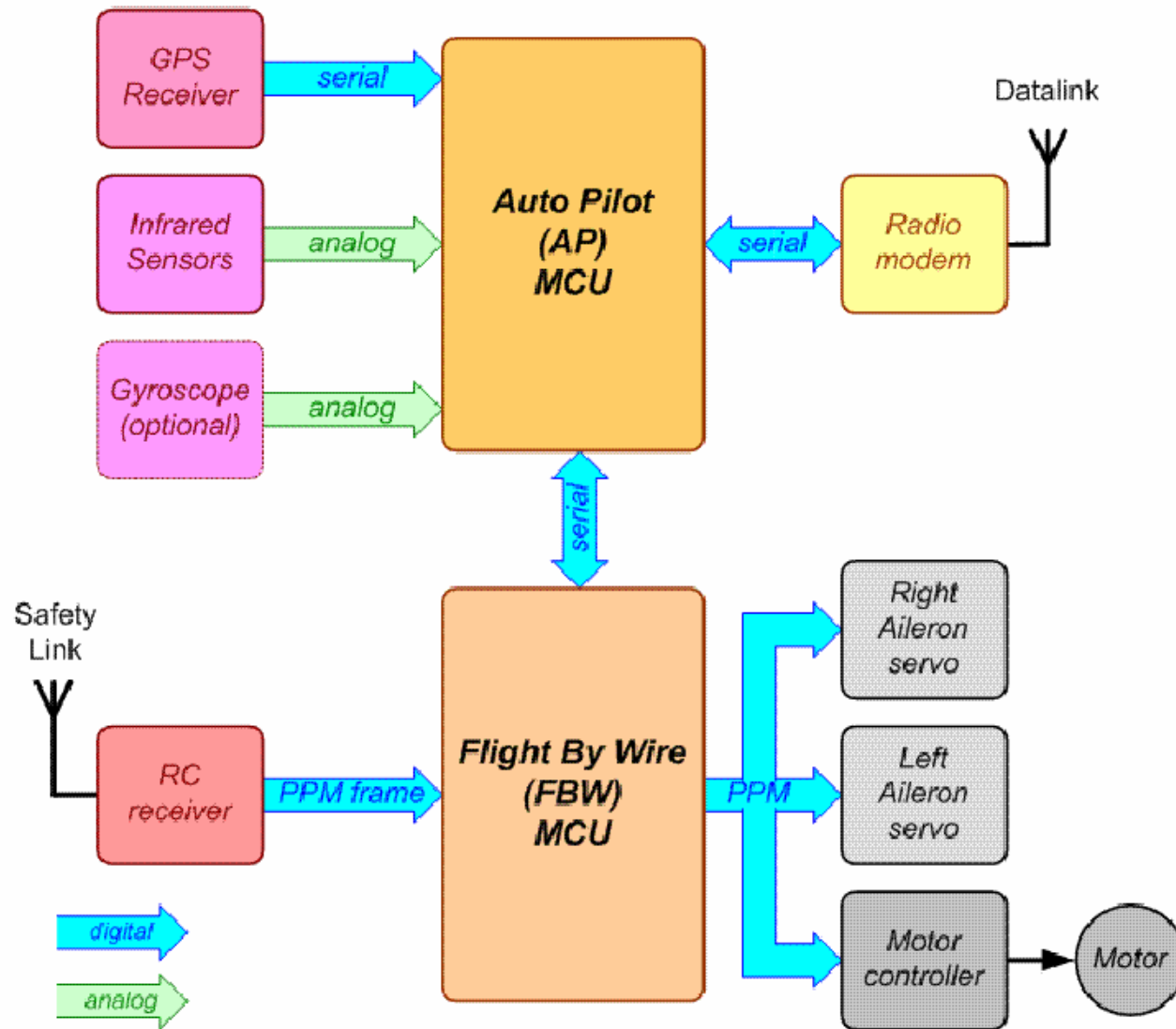


# Hardware architecture

A autopilot  
B attery  
D atalink  
G PS  
I nfrared  
M otor  
P ayload  
R eceiver  
S ervo  
GY ro



# Functional architecture



## Code

### No Operation System

#### Main Loop

```
int main( void ) {
    Fbw(init);
    Ap(init);
    InitSysTimePeriodic()
    while (1) {
        if (sys_time_periodic()) {
            Fbw(periodic_task);
            Ap(periodic_task);
        }
        Fbw(event_task);
        Ap(event_task);
    }
    return 0;
}
```

# Outline - Part I What is a real-time system?

1. First definitions
2. General architecture
3. **WCET computation**
4. Real-time problems

# The WCET Problem

## Given

- ☐ the code for a software task
- ☐ the platform (OS + hardware) that it will run on

## Determine the WCET of the task.

## Can the WCET always be found?

- ☐ In general, no, because the problem is undecidable.

## Usual restrictions in embedded systems (to ease the estimation)

- ☐ loops with finite bounds
- ☐ no recursion
- ☐ no dynamic memory allocation
- ☐ no goto statements
- ☐ single-threaded if possible

# Methods

## Measuring:

- ☐ Compile, link and download onto target CPU
- ☐ Hook up logic analyzer or oscilloscope or use built in registers
- ☐ Run the code with test inputs, and record execution times
- ☐ Take the maximum as WCET
- ☐ No guarantee to hit the worst case!

## Simulation:

- ☐ Various levels of precision possible (cycle accurate, instruction accurate); difficulty to simulate behavior of environment.

## Analysis:

- ☐ Compute estimate of run time, based on program analysis and model of target hardware

# Overview of the methods results

**WCET: The longest time taken by a software task to execute**

=> Function of input data and environment conditions

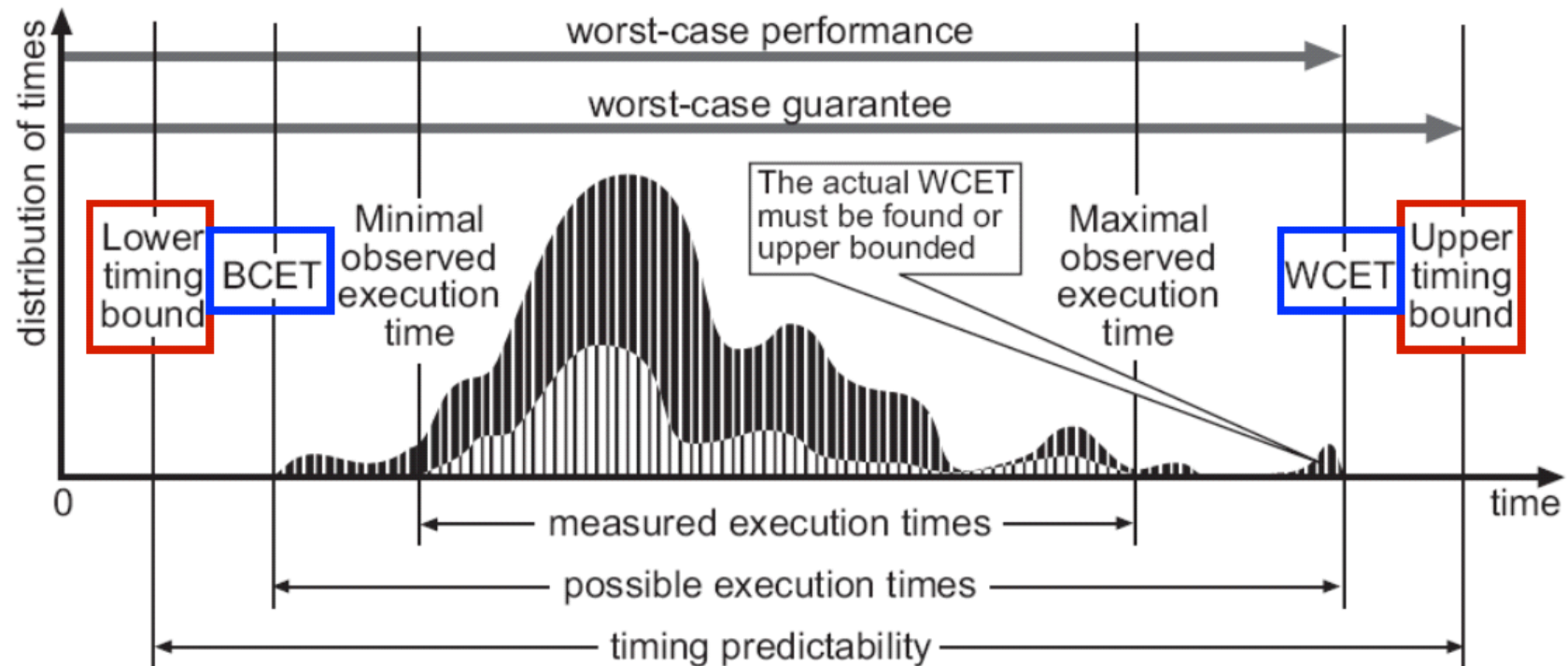


Figure from R.Wilhelm et al., ACM Trans. Embed. Comput. Sys, 2007.

## Influence of input size

- ❑ **Instruction execution time can be different**
  - ❑ MUL (multiply) can take a variable # clock cycles, depending on input size
- ❑ **Control flow (while, for loops) can run for variable number of iterations**

**Parameters:**

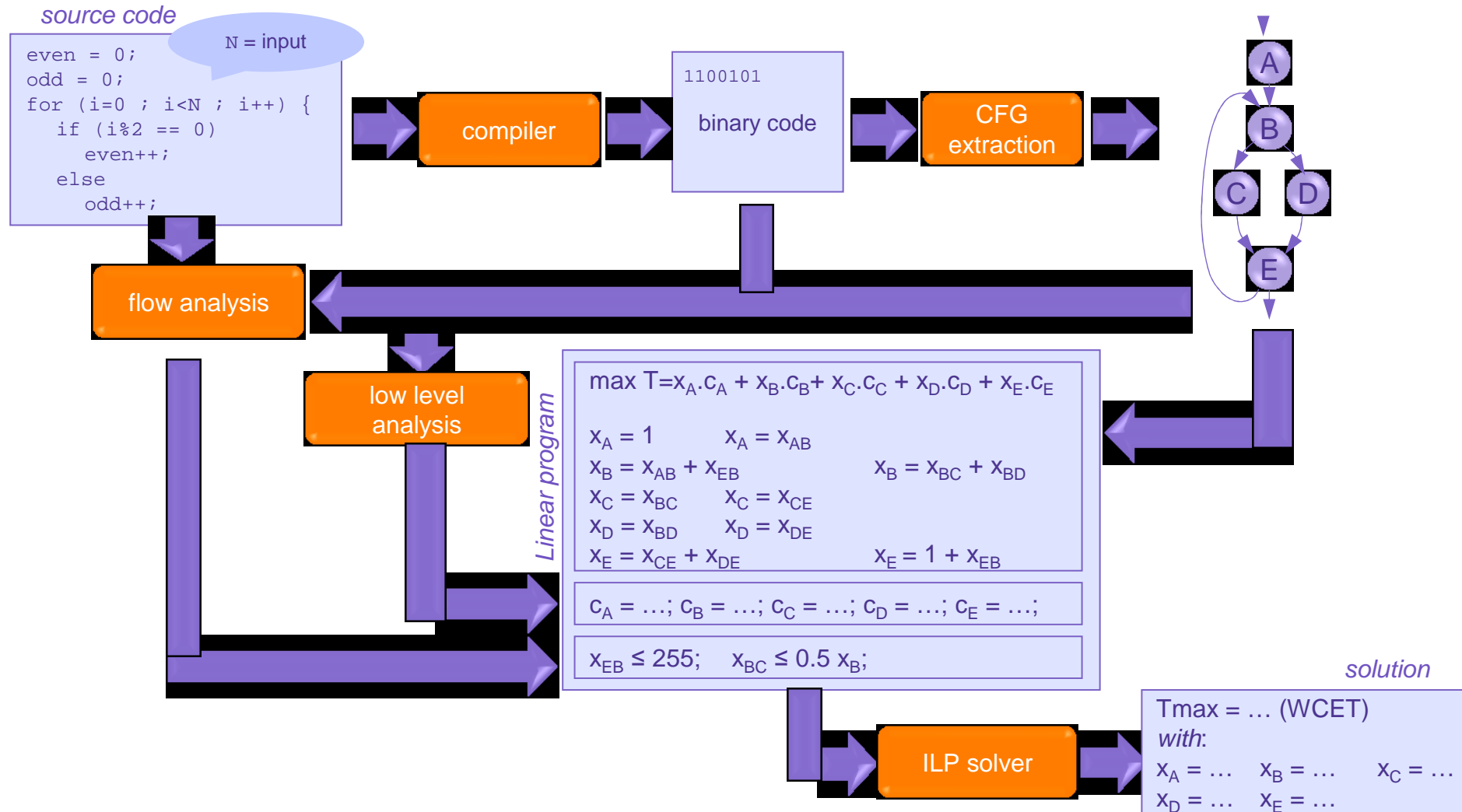
- A positive integer, n.

**Returns:** The sum of the integers from 1 to n.

```
{ sum := 0;  
  for i := 1 upto n  
  { sum := sum + i;}  
return sum;}
```



# WCET computation by static analysis



[Rochange2012]

# Control flow graph (CFG)

- **Nodes** represent basic blocks. A Basic Block (BB) is piece of code with a single entry point and a single exit point, with no branching in-between.
- **Edges** represent flow of control (jumps, branches, calls,...)

## Example

```
test.c
int ex(int n) {
    int i, s=0;
    for(i=0; i<n; i++){
        s+=i;
    }
    return s;
}
```

```
terminal
>gcc -c -g -lrn test.c
>dissy test.o
```

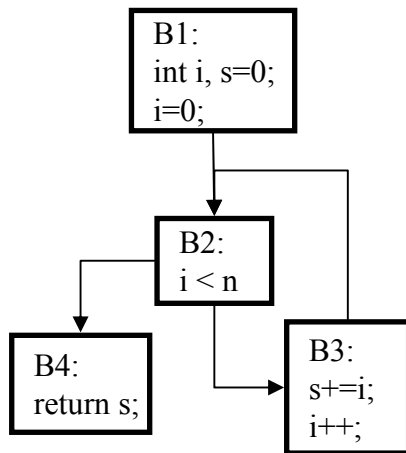
Deassemble the code

0x0000005e	push	%ebp	
0x0000005f	mov	%esp,%ebp	
0x00000061	sub	\$0x10,%esp	
0x00000064	movl	\$0x0,-0x8(%ebp)	BB1
0x0000006b	movl	\$0x0,-0x4(%ebp)	
0x00000072	movl	\$0x1,-0x4(%ebp)	
0x00000079	jmp	85	
<hr/>			
0x0000007b	mov	-0x4(%ebp),%eax	BB3
0x0000007e	add	%eax,-0x8(%ebp)	
0x00000081	addl	\$0x1,-0x4(%ebp)	
<hr/>			
0x00000085	mov	-0x4(%ebp),%eax	BB2
0x00000088	cmp	0x8(%ebp),%eax	
0x0000008b	jnl	7b	
<hr/>			
0x0000008d	mov	-0x8(%ebp),%eax	
0x00000090	leave		BB4
0x00000091	ret		

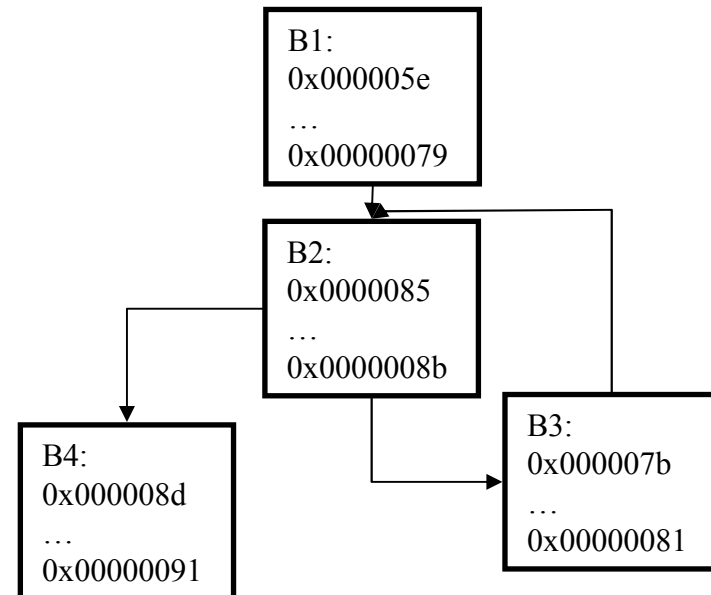
# CFG - example

```
test.c  
int ex(int n) {  
    int i, s=0;  
    for(i=0; i<n; i++){  
        s+=i;  
    }  
    return s;  
}
```

## CFG idea



## Generated CFG



# Identification of the longest path in a CFG

## CFG can have loops, how to infer loop bounds?

- unroll loops, resulting in directed acyclic graph (DAG)
- construct system of equations

### Example

$x_i \rightarrow$  # times  $B_i$  is executed

$d_{ij} \rightarrow$  # times edge is executed

$C_i \rightarrow$  WCET of  $B_i$

**maximize**  $\sum_i C_i x_i$

subject to flow constraints

$$d_{01} = 1$$

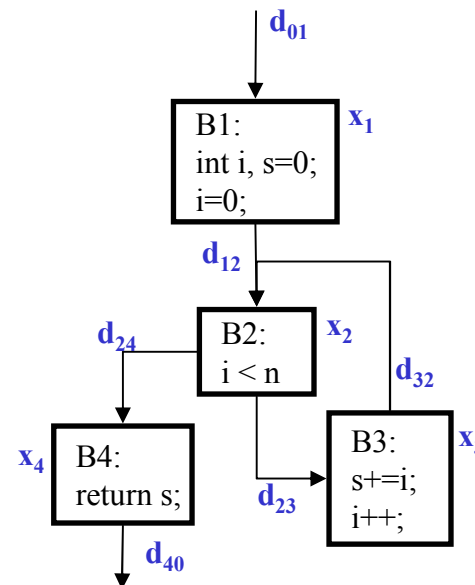
$$x_1 = d_{01} = d_{12}$$

$$x_2 = d_{12} + d_{32} = d_{24} + d_{23}$$

$$x_3 = d_{23} = d_{32} = n$$

$$x_4 = d_{24} = d_{40} = 1$$

designer must give an upper bound of  $n$



# Computation of basic blocks WCET

## Require to make a micro-architecture modeling

- perform a cycle-wise evolution of the pipeline, determining all possible successor pipeline states

## Model of the behaviour of the architecture

### –Pipeline

- Determine each instruction's worst case effective execution time by looking at its surrounding instructions within the same basic block.

### – branch prediction

- Predict which branch to fetch (ex static, always then for an if)

### – Data dependent instruction execution times

### – caches

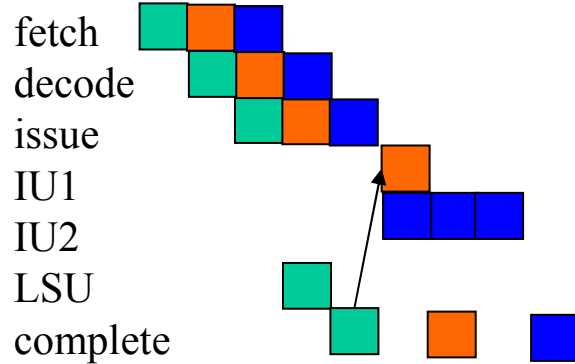
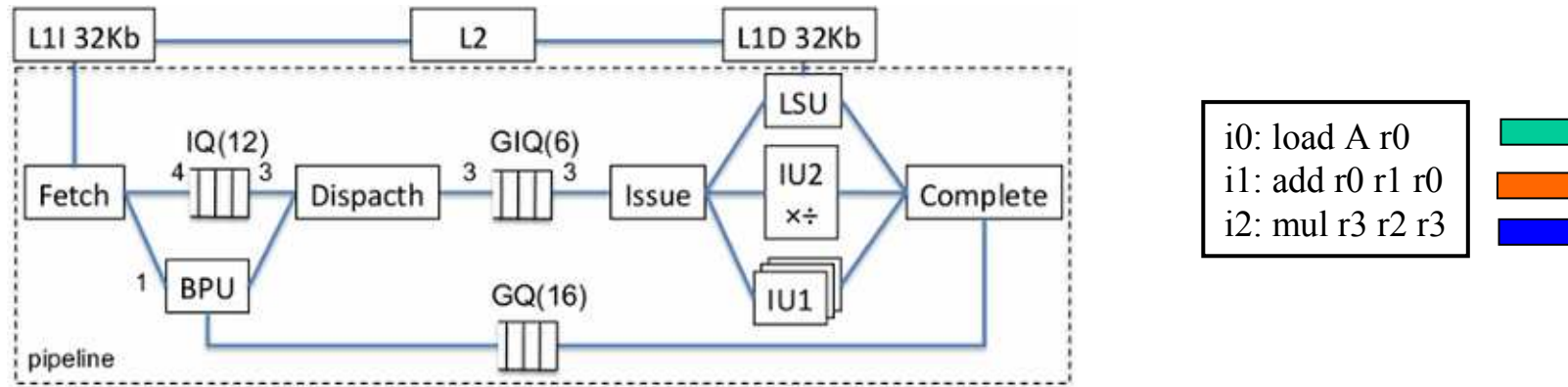
### –in case of multicore, common bus

### –...

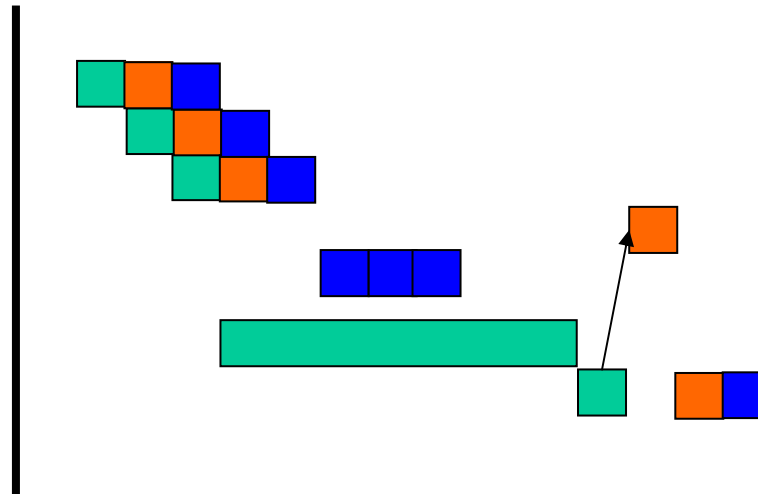
## Read the processor handbook

# Example - micro architectural analysis

Let us consider the following processor architecture and basic block



scenario 1  
A in the L1D



scenario 1  
A not in L1 and not in L2

# Outline - Part I What is a real-time system?

1. First definitions
2. General architecture
3. WCET computation
4. Real-time problems

## Choices for the conceiver

1. **The material architecture:** at system and component level. The system architecture must take into account several constraints (such as the distance between components). The component must consider the input/output and interconnections means.
2. **Communication model:** internal communication of process on a component and distant communication. Different kinds: message passing, rendez-vous, ... Physical support: time triggered busses and networks (1553, Arinc429, TTP...), real-time (CAN, AFDX...), classical network (Ethernet, Wifi...)
3. **Calculator technology:** a PC, workstation, processor, real-time calculator, SOC, ...
4. **Operating system and scheduling:** real-time operating system (VxWorks, PikeOS, Arinc 653,...), micro kernel, no OS ...
5. **Languages:** programming by hand (C, Ada,...), high level semi formal languages (AADL, UML...), high level formal languages (SCADE, Lustre, SDL...)



# Characteristics

## **“Classical” program:**

1. (1) ends; (2) returns a result and (3) handles complex data structure with quite simple control structure.
2. For such programs, properties to be fulfilled are often “when the function is called and the pre-condition is satisfied, then the function ends and the post-condition is satisfied”.

**Typical examples of classical program : compiler, sort algorithm.**

## **Real-time reactive programs.**

1. (1) do not necessarily end; (2) do not compute a result but instead maintain an interaction ; (3) data structures are often simple but the control graph is complex (concurrent execution of components). Moreover, they interact with their environment via actuators (information acquisition) and actuators (action).

**[Bar08]**

# Temporal properties

The properties are very different from those of standard programs. Typically, we are interested of event interleaving all along infinite executions. For instance:

- If a process requests infinitely often to execute, then the OS will execute it at some point;
- it is always possible during the execution to return to the initial state;
- whenever a failure is detected, an alarm is raised;
- whenever an alarm is raised, a failure has been detected.

[Bar08]

# Bibliography

1. **Frédéric Boniol: course on real-time systems**
2. **Pierre-Emmanuel Hladik: course on real-time systems**
3. **Jean-Pierre Elloy: course on real-time systems**
4. **Wikipedia**
5. **Gilles Geeraerts: Introduction aux systèmes embarqués**
6. **Wikipedia**
7. **Pascal Brisset. Paparazzi.**  
[http://www.recherche.enac.fr/~brisset/2010-02\\_ONERA\\_paparazzi.pdf](http://www.recherche.enac.fr/~brisset/2010-02_ONERA_paparazzi.pdf)
8. **Stéphane Bardin. Introduction au Model Checking, 2008.**  
<http://sebastien.bardin.free.fr/MC-ENSTA.pdf>.
9. **Zonghua Gu. Worst-Case Execution Time Analysis.**
10. **Edward A. Lee & Sanjit Seshia, UC Berkeley: Introduction to Embedded Systems - Lecture 19: Execution Time Analysis. 2008**

## Future courses

1. **The material architecture:** dependability and safety course, AADL
2. **Communication model:** courses on network and busses
3. **Calculator technology:** ?
4. **Operating system:** real-time operating system course
5. **Scheduling**
6. **Languages**

# Outline

1. **Part I - What is a real-time system?**
2. **Part II – High level formal programming languages**
3. **Part III – Uniprocessor and multiprocessor scheduling**

# Outline - Part II – High level programming language

**1. Lustre**

**2. SDL**

## **Outline – II.1 - Presentation of Lustre**

- 1. Programming real-time system**
- 2. Lustre overview**
- 3. Semantics, clocks and activations**

# Difficulties

- **Classical difficulties of programming**
- **... and difficulties associated to the management of time**
  - Signal dating
  - Order of the action execution
  - Wait for signal during some delays ...

**And:** the world is full of delays, deadlines, drifts ...

## Typical example:

In a world where actions like communication or computation take a variable amount of time, how to interpret the absence of an information?

- The absence may be due to:
  - A delay
  - A real absence
- A misunderstanding of the situation
- ...



# Asynchronous vs. Synchronous programming

## Asynchronous programming:

- Processes are independent with each other and with the environment
- No global time
- Actions have a non-deterministic duration
- Needs for synchronisation mechanisms (rendez-vous...)

## Advantage:

- Close to the real world

## Problem:

- Concurrency is not deterministic
- Example:

```
product x(0); [product x(1);product x(2);] || product Y(x+1);
```

The result can be either Y(1), Y(2) or Y(3)

=> `product x(1);product x(2);` **not equiv to** `product x(2)`

No code compaction!

# Asynchronous vs. Synchronous programming

## Synchronous programming idea:

- Simplify the world in order to simplify the programming
- Omission of some details
- Synchronous hypothesis:
  - Assumption of the existence of a global time
  - Two actions executed at the same time are supposed to *take no time* (or at least to not exceed the basic period evolution)

## Advantage:

- Determinism
- Simpler programs

## But:

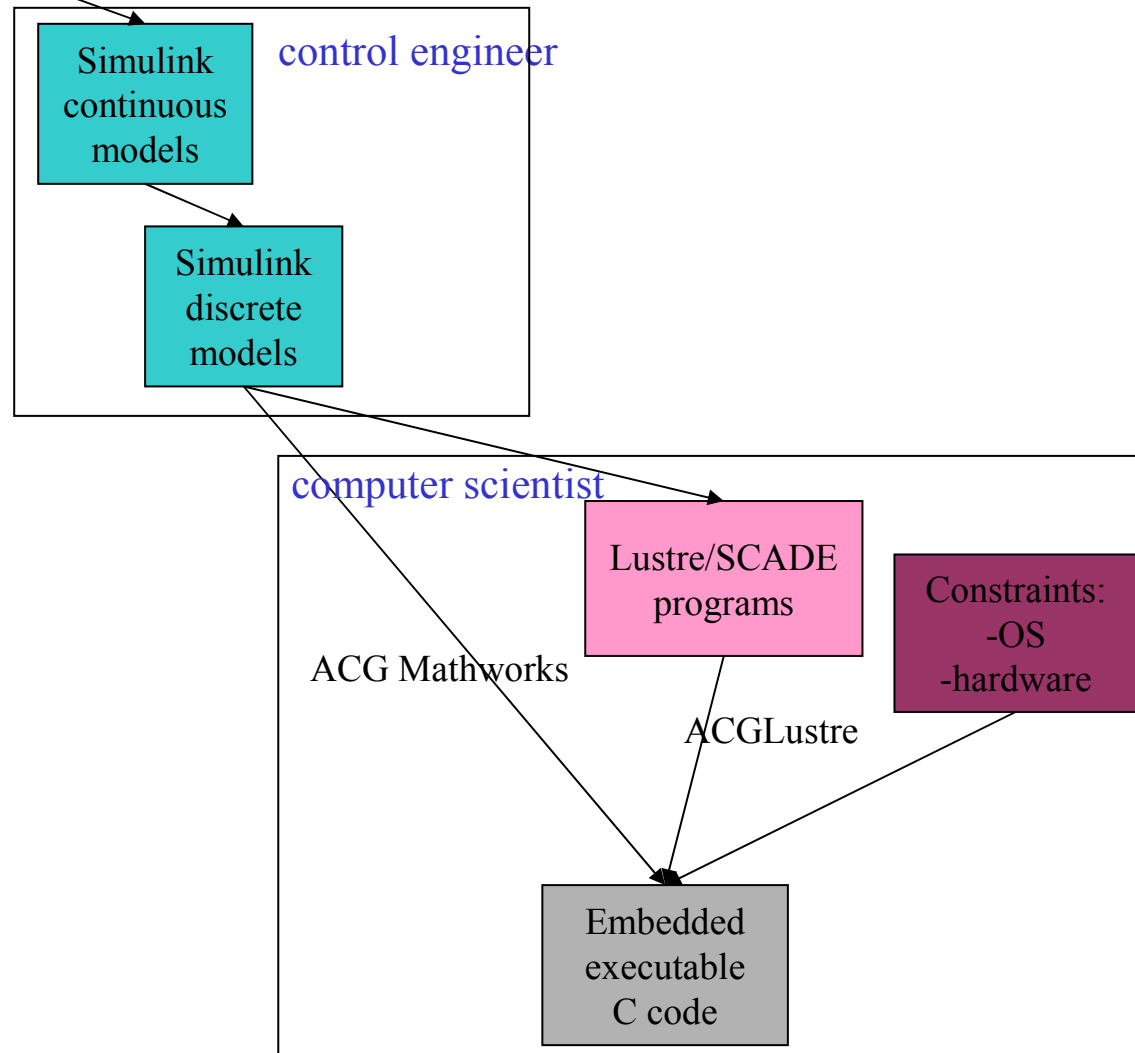
- Need to verify that the implementation respects the hypothesis

## Similar to physicians or chemists approaches:

- Simplification of the world
- Here simplification of the time dimension!

# Development process of control command systems

physicist



# Panorama of synchronous languages

## Main applicative domains

- control-command
- Circuits

<b>Name</b>	<b>Location</b>	<b>People</b>	<b>Type</b>	<b>Commercial</b>
<b>Esterel</b>	<b>Paris, Sophia-Antipolis</b>	<b>G. Berry (1983)</b>	<b>Control flow</b>	<b>Esterel</b>
<b>Lustre</b>	<b>Grenoble</b>	<b>Paul Caspi and Nicolas Halbwachs (1984)</b>	<b>Data flow</b>	<b>SCADE</b>
<b>Signal</b>	<b>Rennes</b>	<b>Albert Benveniste and Paul LeGuernic (1984)</b>	<b>Data flow</b>	
<b>StateCharts</b>		<b>David Harel</b>	<b>Finite state machine extension</b>	<b>Integrated in several languages such as UML</b>
<b>And others</b>				

# Outline– II.1 - Presentation of Lustre

## 1. Programming real-time system

## 2. Lustre overview

1. Introduction
2. Syntax
3. Examples
4. Over sampling and sub sampling

## 3. Semantics, clocks and activations

# Presentation

## Motivation:

**Allow a natural programming**

- of control command systems
- of circuits ...

for the safe programming (of critical and reactive systems)

## Mean:

**Classical techniques of programming close to traditional methods used in industries by engineers**

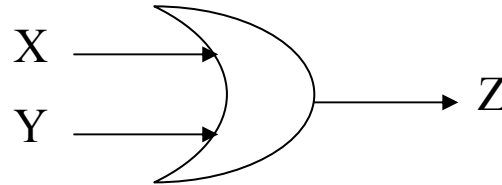
- block diagrams and data-flow
- sampled systems

## LUSTRE:

- Formal language defined in 1985 by P. Caspi and N. Halbwachs in Grenoble Verimag.
- Commercial distribution SCADE - Esterel Technology
- Industrial use: Airbus, Schneider electric

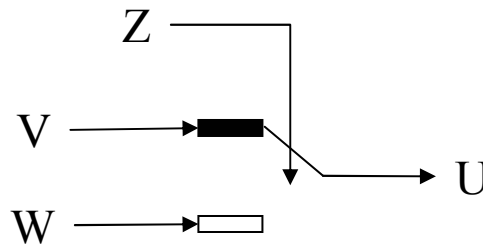
# Examples

## Logic gate



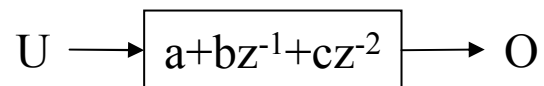
For all  $n$ ,  $Z_n = X_n \text{ or } Y_n$

## Relay



For all  $n$ ,  $U_n = \text{if } Z_n \text{ then } V_n \text{ else } W_n$

## Filter



$$O_0 = aU_0$$

$$O_1 = aU_1 + bU_0$$

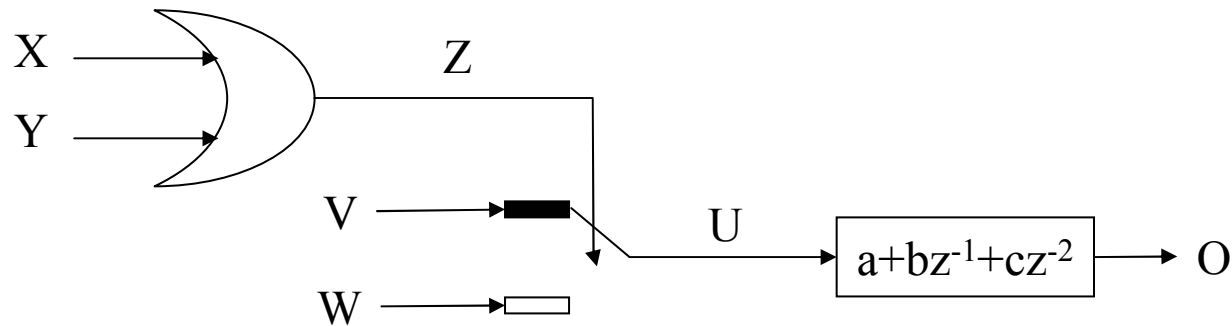
$$\forall n \geq 2, O_n = aU_n + bU_{n-1} + cU_{n-2}$$

# Examples

## Generalisation:

- Description of a system by a sequence of sampled values: the data-flow
- A system = a set of operators applied on the data-flow

## Example



$Z = X \text{ or } Y;$

$U = \text{if } Z \text{ then } V \text{ else } W;$

$O = a.U + b.\text{pre}(U) + c.\text{pre}(\text{pre}(U));$



# Approach

## Data-flow:

- $X$  is a sequence of values  $X_n$  with  $n \geq 0$
- $X_n$  is the value of  $X$  at the instant  $n$  ( $n^{\text{th}}$  top)

**Basic clock:** all the flows are assumed to be cadenced at the same clock

- $X_n$  and  $Y_n$  are the values of  $X$  and  $Y$  at the same instant

## Definition of flows:

- A flow is defined by an equation  $O=F(X,Y,..)$  which computes  $O_n$  depending on  $X_n$  and  $Y_n$  (at the same instant)

## A program Lustre:

- set of equations
- at each top, the variables are evaluated depending on the values of the inputs

# Approach

## Working hypothesis

- We program *as if* the communications and the reactions take no time.
- We focus on the relation between inputs and outputs

## Validation of working hypothesis

- The method guarantees that the reactions are realised in a **bounded time**, computable for a given architecture
- The hypothesis is valid if this bound is less than the dynamic of the environment

## General syntax

```
[declaration of types and external functions]
node name (declaration of input flows)
returns (declaration of output flows)
[var declaration of local flows]
let
  [assertions]
  system of equations defining once each local flow
  and output depending on them and the inputs
tel.

[other nodes]
```

### Types :

- basic types: int, bool, real
- tabular :  $\text{int}^3, \text{real}^{5^2} \dots$

# Equations

- an equation is defined on a internal flow or an output depending on internal flows, inputs and outputs

$$X = Y + Z$$

$$Z = U$$

means

$$\text{for all } n \geq 0, X_n = Y_n + Z_n \text{ et } Z_n = U_n$$

- => an equation defines a mathematical equality, and not an computer assignment :  
a flow may be replaced by its definition in all the equations of the node

$$X = Y + Z$$

is equivalent to

$$X = Y + U$$

$$Z = U$$

$$Z = U$$

- => equations are not ordered

$$X = Y + Z$$

is equivalent to

$$Z = U$$

$$Z = U$$

$$X = Y + Z$$

# Operators

## Classical operators:

Arithmetical :

Binary : +, -, \*, div, mod, /, \*\*

Unary : -

Logical :

Binary : or, xor, and, =>

Unary : not

Comparison :

=, <>, <, >, <=, >=

Control :

if . then . else

## Temporal operators :

**pre** (precedent) : operator which allows to work on the past of a flow

**->** (followed by) : operator which allows to initiate a flow

# Operator pre

**Memorization of the precedent value of a flow or a set of flows.**

Let

$X$  be the flow  $(X_0, X_1, \dots, X_n, \dots)$

then

$\text{pre}(X)$  is the flow  $(\text{nil}, X_0, X_1, \dots, X_n, \dots)$

By extension, the equation

$$(Y, Y') = \text{pre}(X, X')$$

means

$$Y_0 = \text{nil}, \quad Y'_0 = \text{nil}$$

$$\text{and for all } n \geq 1, Y_n = X_{n-1} \text{ and } Y'_n = X'_{n-1}$$

**Example** : detection of the overtaking

Distance = if  $(X > \text{pre}(X))$  then  $X - \text{pre}(X)$  else  $\text{pre}(X) - X$  ;

Over =  $(\text{Distance} > \text{Threshold})$  ;

# Operator ->

## Initialisation of a flow.

Let

X be the flow  $(X_0, X_1, \dots, X_n, \dots)$  and Y the flow  $(Y_0, Y_1, \dots, Y_n, \dots)$

then

$Y \rightarrow X$  is the flow  $(Y_0, X_1, \dots, X_n, \dots)$

By extension, the equation

$$(Z, Z') = (Y, Y') \rightarrow (X, X')$$

means

$$Z_0 = Y_0, \quad Z'_0 = Y'_0$$

$$\text{and for all } n \geq 1, Z_n = X_n \text{ and } Z'_n = X'_n$$

**Example** : monitoring of a temperature

$A = (T > 100) \rightarrow \text{if } (T > 100) \text{ then true else pre}(A) ;$

equivalent to :

$$A_0 = (T_0 > 100)$$

$$A_n = \text{true si } (T_n > 100)$$

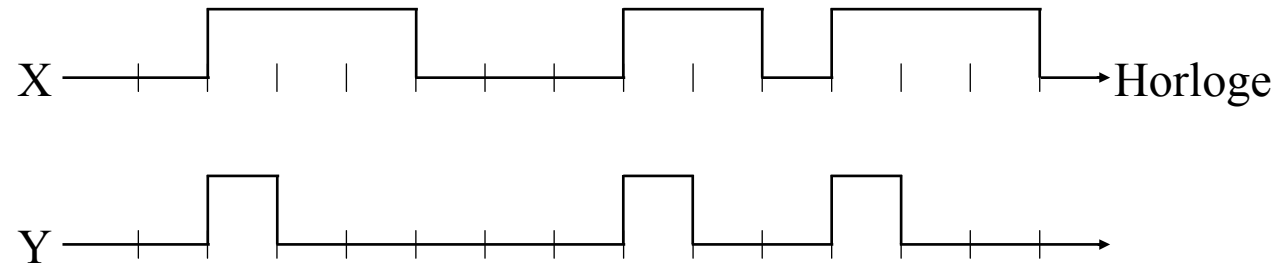
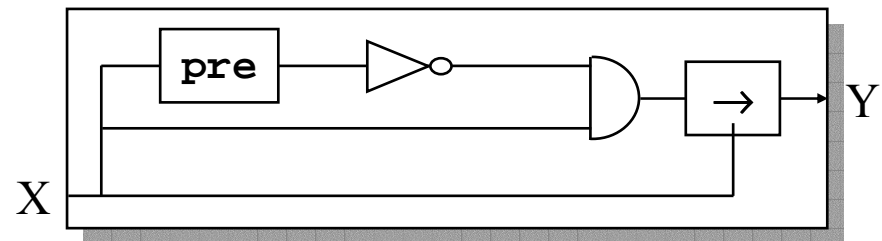
$$A_{n-1} \text{ sinon}$$

# Example

## Detection of rising edges

Let X be an input Boolean flow, let Y be an output Boolean flow

```
node EDGE (X : bool) returns (Y : bool)
let
  Y = X → (X and not pre(X));
tel ;
```



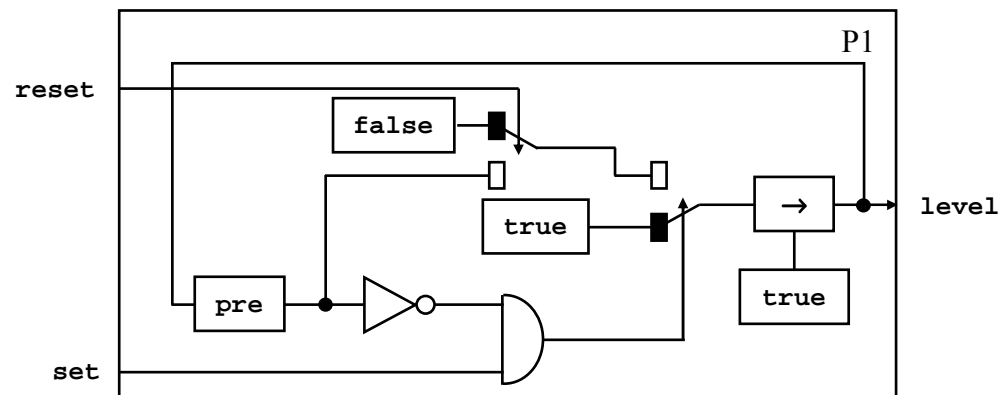


# Example

## Flip-flop

Let set and reset be two input Boolean flows, let level be an output Boolean flow

```
node Q (reset, set : bool) returns (level : bool)
let
  level = true → if set and not pre(level)
                then true else if reset then false
                else pre(level) ;
tel ;
```



## Exercises

### Compute the minimum and the maximum of a sequence

```
node MinMax (X:int)
returns (min, max : int)
let

tel
```

### Compute the minimum and the maximum of 2 flows

```
node MinMaxPaire (X,Y:int)
returns (min, max : int)
let

tel
```

## Exercise

### A resetable counter

- Input: reset            reset the counter (Boolean flow)
- Output: counter        value of the counter (Integer flow)

### Write the program

```
node a_counter  
let
```

```
tel
```

## Exercise

### A timer:

- Input: set                      activation of the timer (Boolean flow)
- Output: level                  state of the timer (Boolean flow)
- Constante: delay              duration of the timer in number of tops

### Write the program

```
const delay : int;  
node timer  
let  
  
tel
```

# Assertion

**Allows the designer to make its hypothesis, on the environment and the program, explicit**

- Optimisation at compilation
- Verification of properties

## Example

```
assert (not (X and Y))
```

imposes that the Boolean flows X and Y are never true simultaneously

```
assert (true -> not (X and pre(X)))
```

imposes that the Boolean flow X can never transport two values equal to true consecutively

## **Example: Lulu, the wolf, the goat and the cabbage**

**Lulu needs to bring a wolf, a goat, and a cabbage across the river.**

- The boat is tiny and can only carry one passenger at a time.
- If he leaves the wolf and the goat alone together, the wolf will eat the goat.
- If he leaves the goat and the cabbage alone together, the goat will eat the cabbage.
- How can he bring all three safely across the river?

## **Example: Lulu, the wolf, the goat and the cabbage**

### **Inputs of the program: Lulu's actions**

- m: Lulu crosses the river alone
- mw: Lulu crosses the river with the wolf
- mg: Lulu crosses the river with the goat
- mc: Lulu crosses the river with the cabbage

### **Outputs of the program: positions of each one**

- L: position of Lulu
- W: position of the wolf
- G: position of the goat
- C: position of the cabbage
- A position X is in  $\{0,1,2\}$ : 0 stands for X is on the river 0, 1 for X is on the river 1 and 2 for X has been eaten

## Example: Lulu, the wolf, the goat and the cabbage

### Program

```
node river(m, mw, mg, mc : bool) returns (L, W, G, C : int)
  assert (m or mw or mg or mc);
  assert( not (m and mw));
  assert( not (m and mg));
  assert( not (m and mc));
  assert( not (mw and mg));
  assert( not (mw and mc));
  assert( not (mg and mc));
  assert( true -> not (mw and not (pre(L)=pre(W))));
  assert( true -> not (mg and not (pre(L)=pre(G))));
  assert( true -> not (mc and not (pre(L)=pre(C))));

  let
    L = 0 -> 1 - pre(L);
    W = 0 -> if mw then 1 - pre(W) else pre(W);
    G = 0 -> if pre(G) = 2 then pre(G)
              else if mg then 1 - pre(G)
                    else if (pre(G)=pre(W) and not mw) then 2
                          else pre(G);
    C = 0 -> if pre(C) = 2 then pre(C)
              else if mc then 1 - pre(C)
                    else if (pre(C)=pre(G) and not mg) then 2
                          else pre(C);
  tel.
```



## **Example: Lulu, the wolf, the goat and the cabbage**

**Winning strategy: mg,m,mc,mg,mw,m,mg**

**What happens for the sequence: m,m,mw,m...**

## First conclusion

**Lustre allows to describe naturally cyclical programs.**

- deterministic: the order of the equation has no impact
- bounded execution time (no dynamical process, no variable loop ...)
- bounded memory (number of pre)
- modular (reuse of nodes by nodes)

## Outline– II.1 - Presentation of Lustre

1. Programming real-time system
2. Lustre overview
3. Semantics, clocks and activations

# Clock-based semantics

## Clock:

- a clock is a Boolean flow

## Basic clock:

- is the flow true

## Semantics of a flow:

- is the sequence of pairs  $(v_i, c_i)$  where  $v_i$  is the value and  $c_i$  is the clock associated to the flow

## Equation:

- must be homogenous in term of clock
  - $X + Y$  has a sense iff  $X$  and  $Y$  have the same clock

# Operator when

## Sub-samples a flow on a lower clock

Let  $X$  be a flow and  $B$  a Boolean flow (assimilated to a clock) of same clock.

The equation

$$Y = X \text{ when } B$$

defines a flow  $Y$ , of same type than  $X$ , and of clock  $B$

- $Y$  is present when  $B$  is true
- $Y$  is absent when  $B$  is false or when  $B$  and  $X$  are absent

# Operator current

## Over-samples a flow on a quicker clock

Let  $X$  be a flow and  $B$  a Boolean flow (assimilated to a clock) of same clock.

The equation

$$Y = X \text{ current } B$$

defines a flow  $Y$ , of same type than  $X$ , and of clock the clock of  $B$

- $Y$  is present iff  $B$  is present
- when  $Y$  is present,  $Y$  is equal to  $X$  if  $X$  is present and to the previous value of  $X$  otherwise

# Sampling

**The operator *when* defines a *slower* flow than the input**

X	4	1	-3	0	2	7	8
C	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>
X when C	4			0	2		8

**Question:** is the flow  $X + X \text{ when } C$  well defined?

**The operator *current* constructs a *faster* flow than the input**

X	4	1	-3	0	2	7	8
C	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>
Y=X when C	4			0	2		8
current (Y)	4	4	4	0	2	2	8

**Warning:**  $\text{current}(X \text{ when } C) \neq X$

## Problem of initialisation of current

X	4	1	-3	0	2	7	8
C	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>
current(X when C)	nil	nil	nil	0	2	2	8

### Idea 1 : sample with clocks which are initially true

C1 = true -> C	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>
current(X when C1)	4	4	4	0	2	2	8

### Idea 2: force a default value

E = if C then current(X when C) else (dft -> pre E);

### Idea 3: (to avoid additionnal memory)

X1 = (if C then X else dft) -> X;

E = current(X1 when C1);



## Exercise

**For the following counters:**

```
node counter (reset : bool) returns (n : int)
let n = 0 -> if reset then 0 else pre n + 1; tel
```

```
node decounter (reset : bool) returns (n : int)
let n = 0 -> if reset then 0 else pre n - 1; tel
```

```
node counter2 (reset,c : bool) returns (n : int)
let n = if c then counter(reset)
        else decounter(reset); tel
```

**Precise the behaviour of the node**

```
node counter3 (reset,c : bool) returns (n : int)
var c1,c2 : int;
let c1 = current (counter(reset when c));
    c2 = current (counter2 (reset when not c, false when not
c));
    n = if c then c1 else c2; tel
```

# Recapitulative

B	false	true	false	true	false	false	true	true
X	x0	x1	x2	x3	x4	x5	x6	x7
Y	y0	y1	y2	y3	y4	y5	y6	y7
pre(X)	nil	x0	x1	x2	x3	x4	x5	x6
Y->pre(X)	y0	x0	x1	x2	x3	x4	x5	x6
Z=X when B		x1		x3			x6	x7
T=current Z	nil	x1	x1	x3	x3	x3	x6	x7
pre(Z)		nil		x1			x3	x6
0->pre(Z)		0		x1			x3	x6

## Exercise

**Define the following flow with some LUSTRE equations:**

1. 0,1,2,3,.. .
2. 0,1,0,1,0,1,.. .
3. 0,0,1,0,1,0,1,0,1,.. .
4. 1,1,2,3,5,8,13,.. . (suite de Fibonacci)
5. a flow S which value increments when E is true, for instance:
  1. E: 0 1 0 0 1 0 0 0 1 .. .
  2. S: 1 2 3 .. .
6. A flow S which takes the value of E each N tops, for instance:
  1. E: 1 3 0 4 0 2 3 1 ...
  2. S: 1 4 3 ...

## Exercise: 3 bits adder

Booleans are interpreted as binary:

- `false = 0, true = 1`

Write a node

- `node Add3b(cin, x, y : bool) returns (cout, s : bool);`
- Which computes cin, x and y  
i.e. for all  $t$   $\text{cin}_t + x_t + y_t = 2 * \text{cout}_t + s_t$

## Exercise: serial adder

The Boolean flows are interpreted (when possible) as binary numbers :

$$X = X_0 + X_1 * 2 + X_2 * 2^2 + \dots + X_t * 2^t + \dots$$

1. What is the integer for the flow *false* ?
2. What is the integer for the flow *true -> false* ?
3. Write a serial additionneur:
  1. node `AddSerie(X, Y : bool)` returns `(S : bool)`;
  2. Such that S represents the sum of the binary numbers X and Y.
4. What is the flow *AddSerie(true->false, true->false)* ?
5. How is interpreted the flow *true* ?

## References

1. **Frédéric Boniol: course on Lustre**
2. **Alain Girault: course on Lustre**
3. **Pascal Raymond: course on Lustre**
4. **Julien Forget: course on real-time systems**
5. **Nicolas Halbwachs, Paul Caspi, Pascal Raymond, Daniel Pilaud: The synchronous dataflow programming language Lustre (1991)**
6. **Nicolas Halbwachs. Synchronous programming of reactive systems. Kluwer Academic Pub., 1993**
7. **Jérôme Ermont. TP on the Lego robot.**

# Outline - Part II – High level programming language

1. Lustre

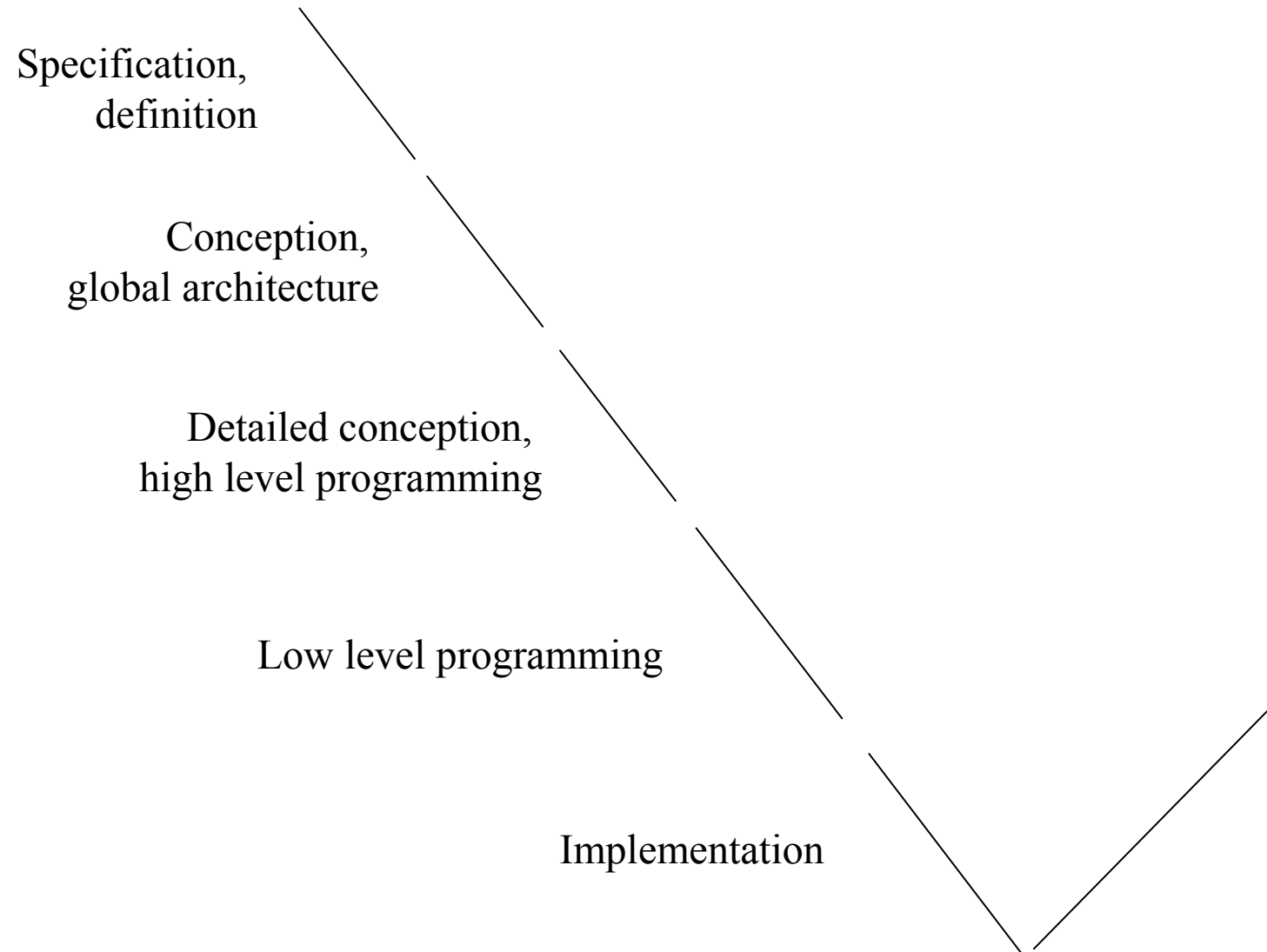
2. SDL

## Outline – II.2 SDL

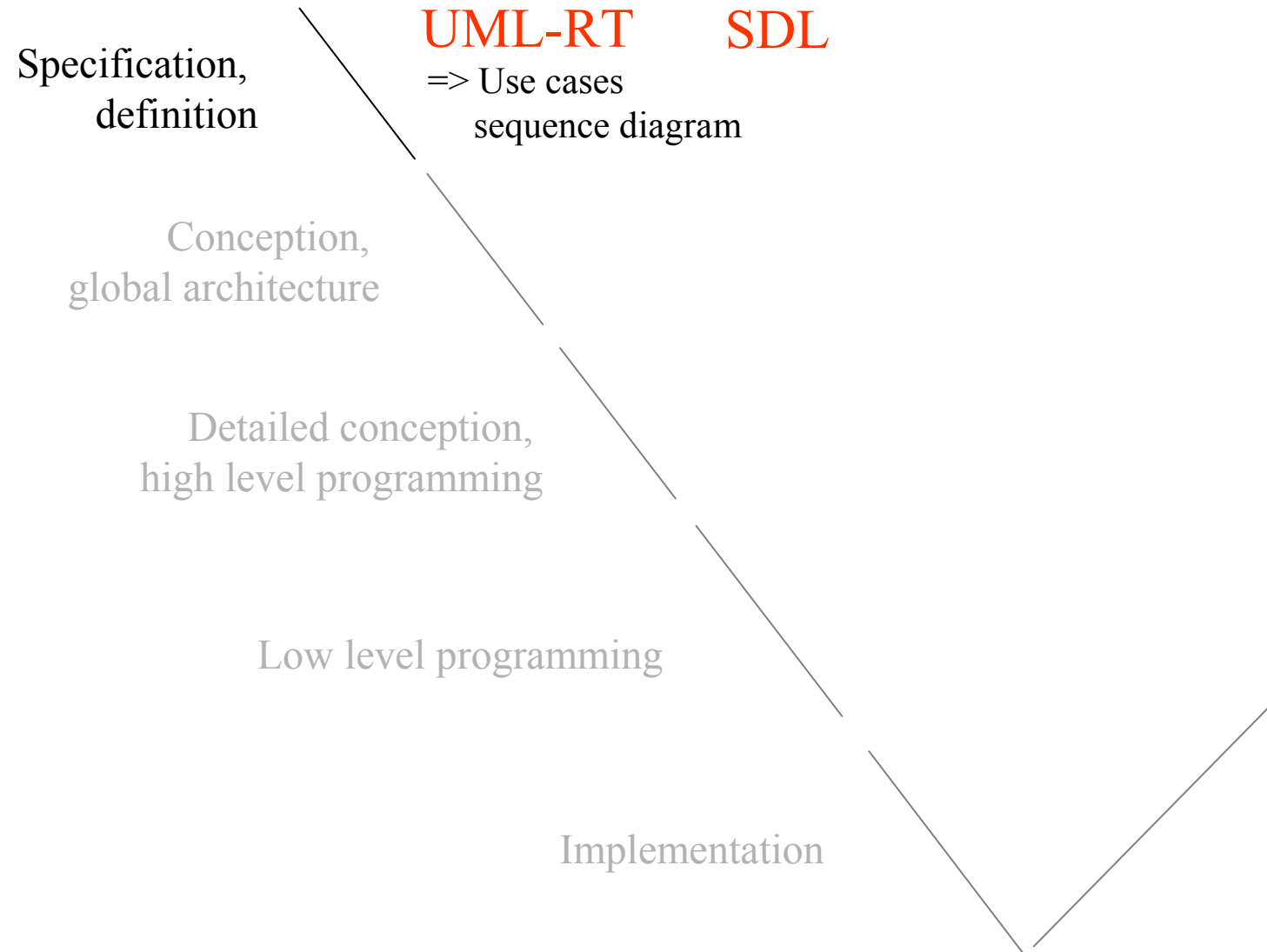
1. Where is it possible to apply a language?
2. Introduction of SDL
3. Syntax of SDL
4. Simulation and MSC



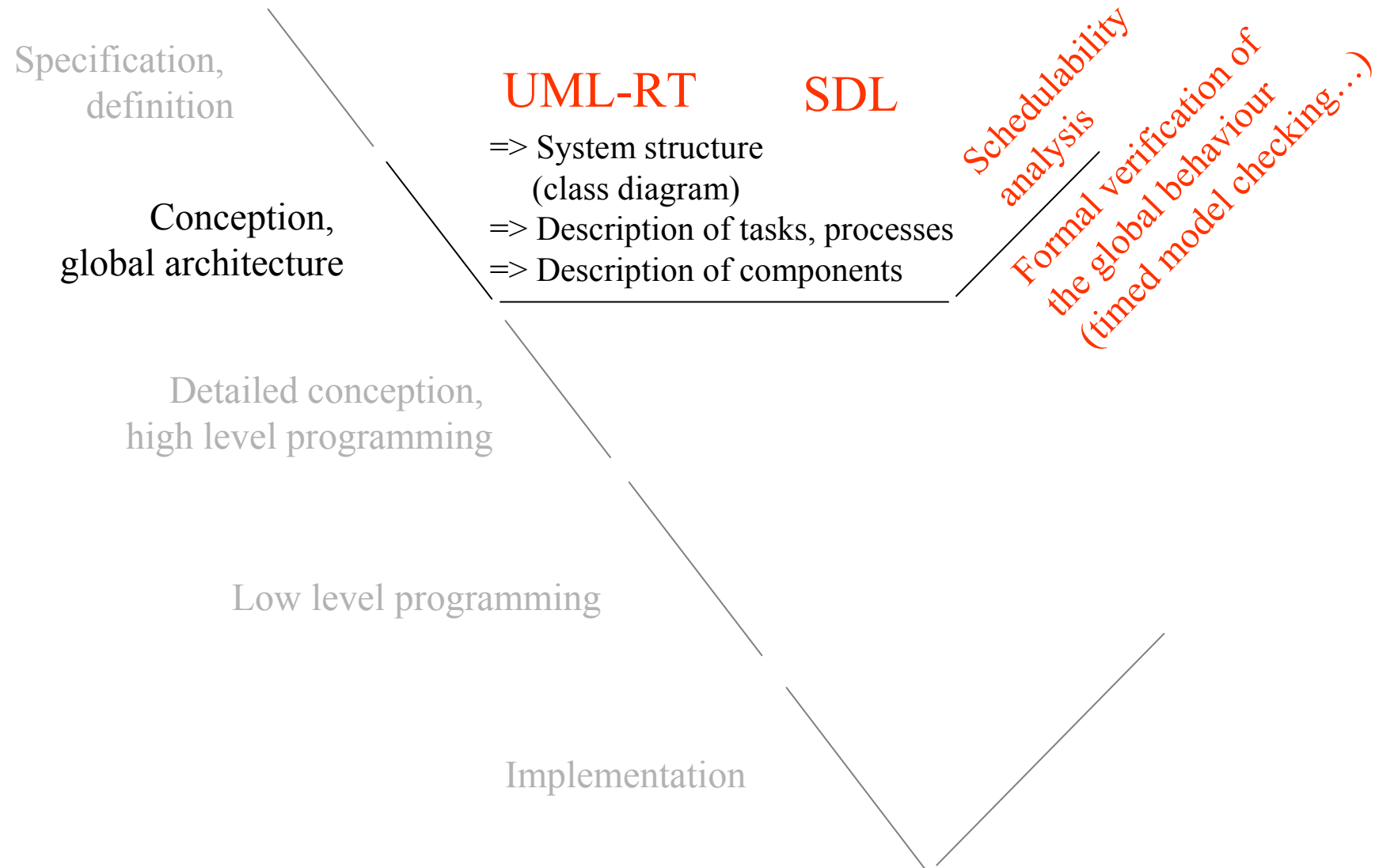
## Development process



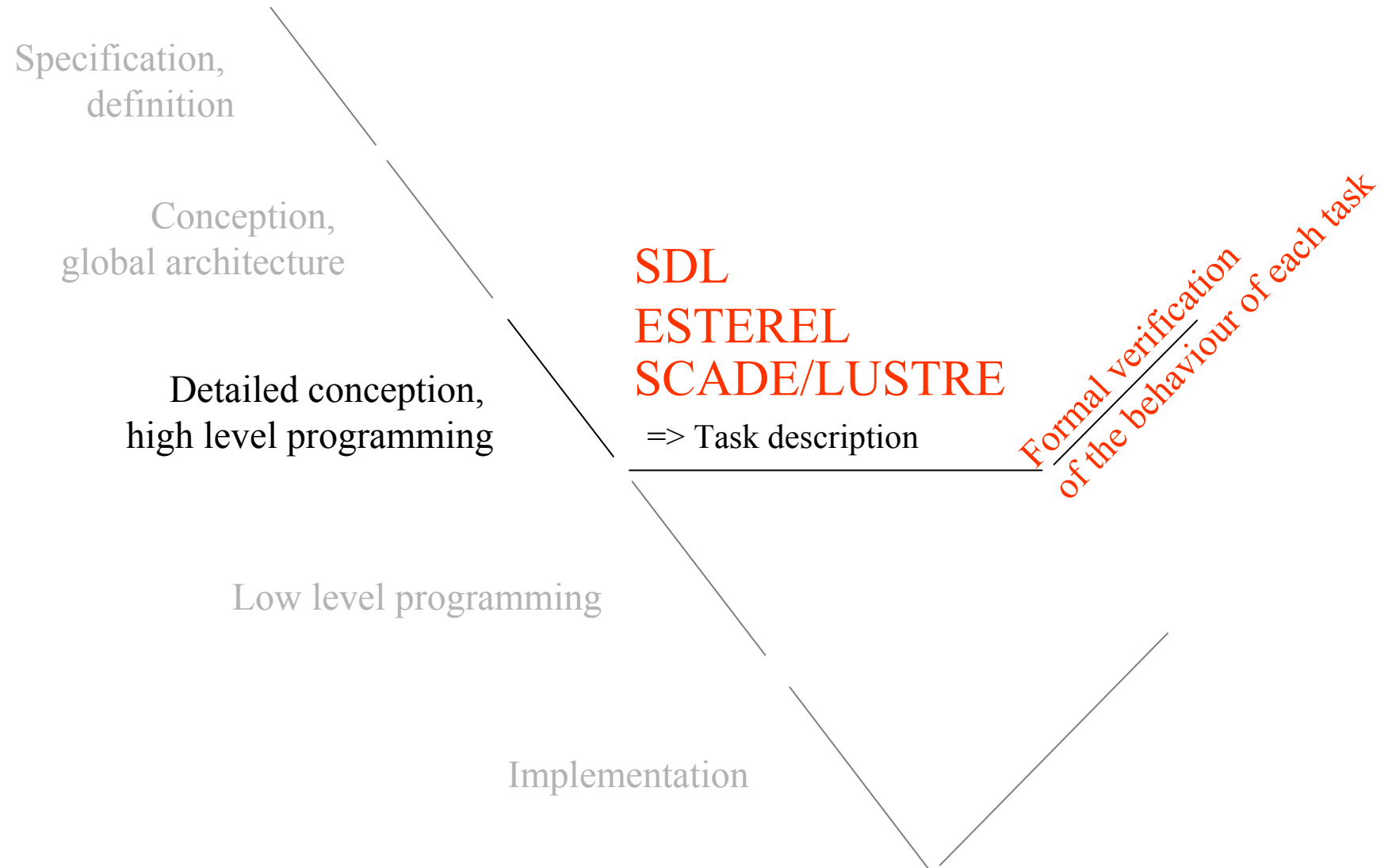
## Development process



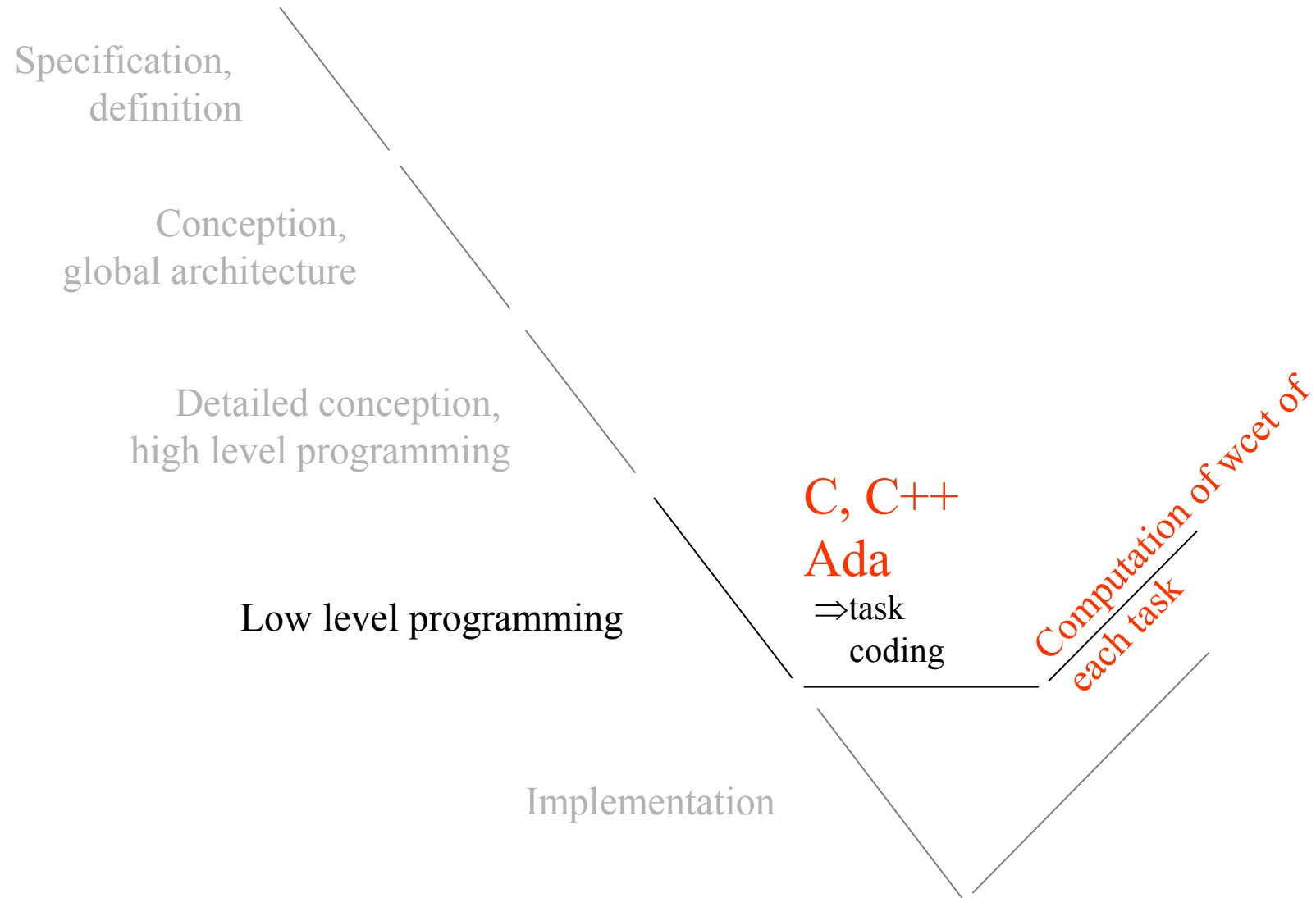
# Development process



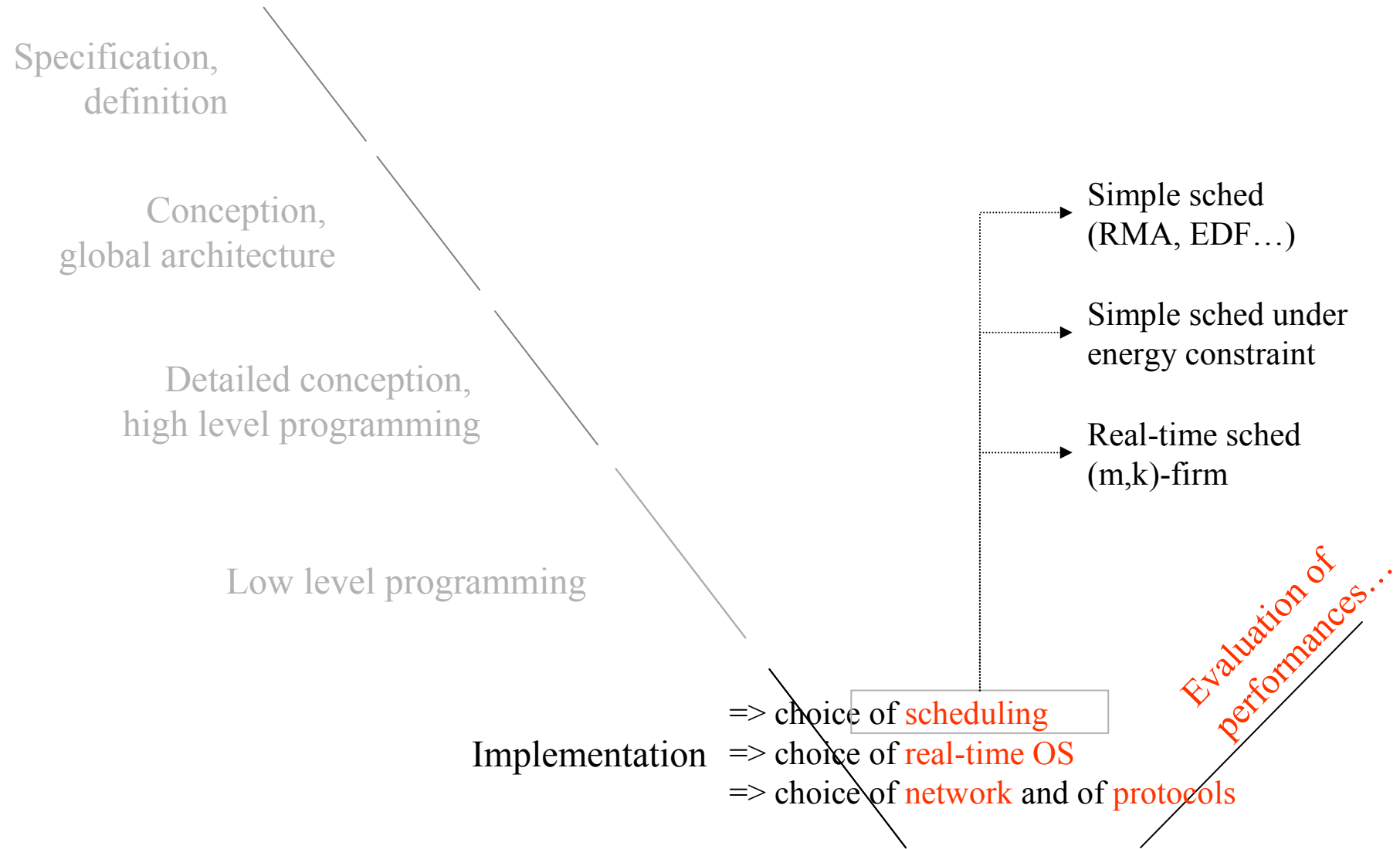
# Development process



# Development process



# Development process



## Outline – II.2 SDL

1. Where is it possible to apply a language?
2. Introduction of SDL
3. Syntax of SDL
4. Simulation and MSC

# Generalities

## SDL (Specification and Description Language):

- Defined in 1976 by the CCITT, Comité consultatif internationale télégraphique et téléphonique – now ITU-T International Telecommunication Union – recommendation Z.100 – Z.109
- Initially for the specification and the development of protocols
- Well adapted for the formal description of reactive (embedded real-time distributed ...) systems
- Living language (regular updates)

## General features:

- the *specification* of a system is the description of the expected *behaviour*
- the *description* of a system is the description of the real *behaviour*
- the specifications and the descriptions made in SDL must be *formal* in the sense that it could be possible to analyse and interpret them without ambiguity
- SDL is easy to learn and use (several formal methods should be applied)
  - => Graphical language



## Brief historical

**1968** first studies

**1976** Orange book SDL – elementary graphical language

**1980** Yellow book SDL – process semantics definition

**1984** Red book SDL – data types, first tools

**1988** Blue book SDL – concurrence, composition, tools

=> SDL-88 simple language with a formal semantics

**1992** White book SDL – object oriented approach

=> SDL-92 more complex language

**1996** SDL-96 minor modifications

**2000** Integration with UML

# Advantages in the development process

**Specification:** description of the expected behaviour

**Description:** description of the real behaviour

- Possibility to simulate/execute system partially described
- Compiler verifies the coherence of the architecture
- Formal semantics: no ambiguity for the supplier and designer
- Behavioural semantics vs sequence diagram: diagram are incomplete and do not describe prohibited behaviour
- Interface with UML
- Tests: automatic generation and validation

**Implementation:** automatic code generation

- Independence with the API
- Automata coding error prone

**Unique language all along the development**

## **Industrial example: ATC (Air Traffic Control)**

**ATC is a service provided by ground-based controllers who direct aircraft on the ground and in the air. France is divided into 5 zones controlled by an ATC centre. Their purpose is**

- to separate aircraft to prevent on ground and on air collisions,
- to organize and expedite the flow of traffic,
- to provide information and other support for pilots: weather and traffic information, route clearances;
- Information for rescue services

**[wikipedia]**

# Embedded ATC

**ON board data (position, altitude, speed, weather) are collected, transmitted regularly to on ground equipments. Electronic dialogue between pilot and controllers.**

- **ADS (Automatic Dependant Surveillance)** : surveillance technology for tracking aircraft. If an aircraft deviates from its flight profile, it reaches a specific monitored mode to quickly correct its position. Future generation: ADS-B (Broadcast), delays are improved
- **CPDLC (Controller-Pilot Data Link Communications)** : method by which air traffic controllers can communicate with pilots over a datalink system. Formatted electronic dialogue (no ambiguity): set of clearance/information/request message elements which correspond to voice phraseology employed by Air Traffic Control procedures (eg clearance to climb or descend to a given flight altitude)
- **CAP (Controller Access Parameters)**. On board systems compute and automatically send surveillance data (such as magnetic heading, indicated airspeed, vertical rate...). This allows a tighter surveillance of the traffic for the ATC and reduce the overload on human

[\[wikipedia\]](#)

# Embedded ATC

**On board application: sub tasks are executed within an IMA partition.  
These sub functions are coded with SL processes.**

**456 000 C code lines from SDL**

**156 000 manual C code lines**

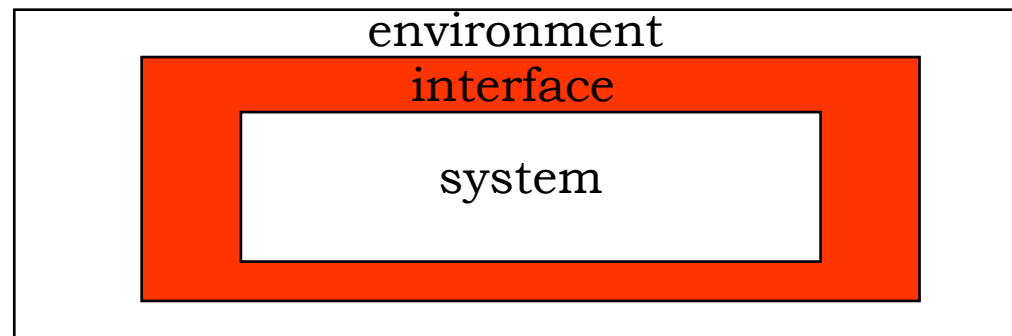
## Outline – II.2 SDL

1. Where is it possible to apply a language?
2. Introduction of SDL
3. Syntax and semantics of SDL
4. Simulation and MSC

# Structure of an SDL system

**An SDL system is a structured set of processes which execute in parallel and communicate by exchanging messages**

**An SDL system is a real-time system immersed in its environment through an interface**

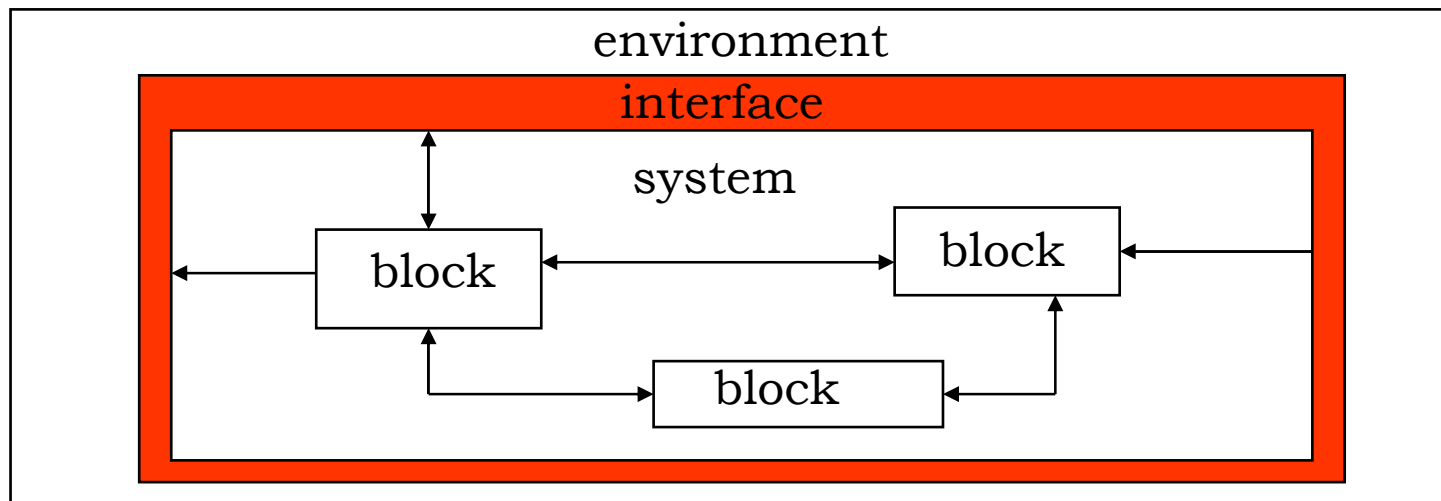


# Structure of an SDL system

**SDL provides structuring principles which consist in decomposing a system in components that can be developed independently and in no specific order.**

## **3 levels of structuring:**

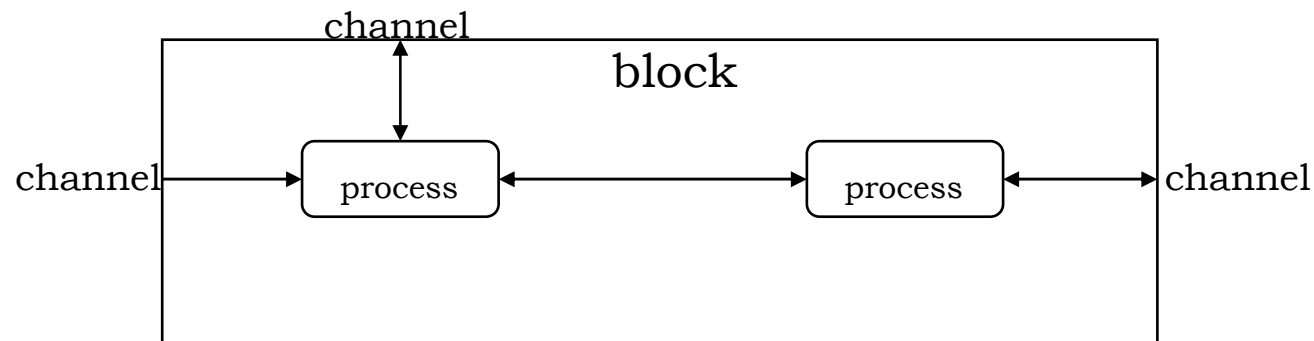
1. System = block + communication channel
2. Block = processes + signal route
3. Process = an automaton



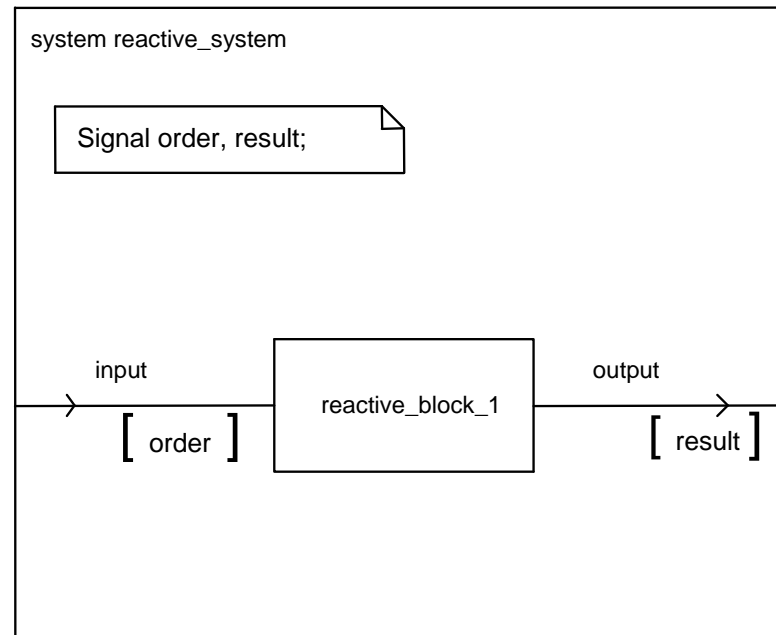


# Structure of an SDL system

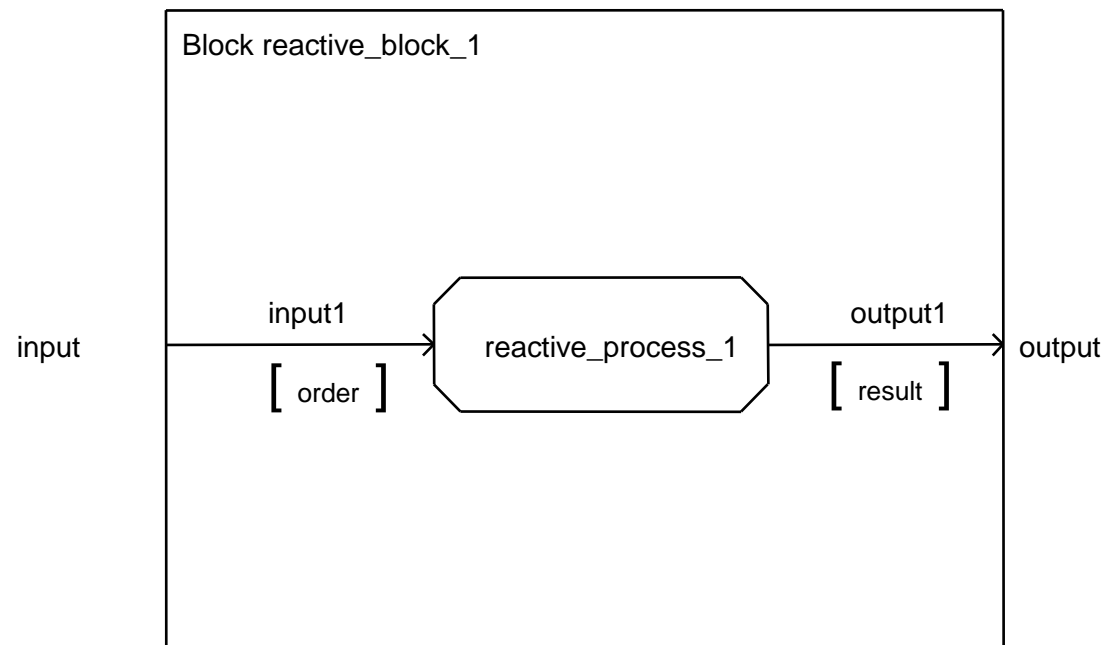
A block can be decomposed into blocks, but a block leaf is necessary a process or a set of processes



## Example: simple reactive system

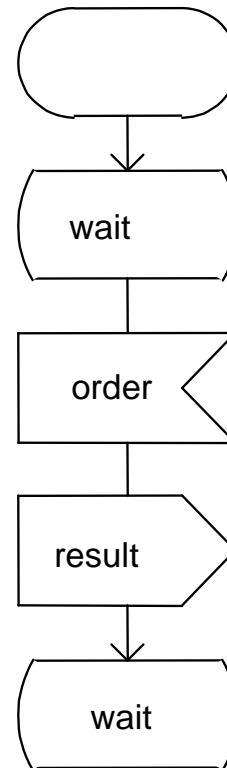


## Example: simple reactive system



## Example: simple reactive system

process reactive\_process\_1



# Lexical rules

## Comments:

- between /\* ...\*/
- after keywords COMMENT
- special graphical symbol for the graphical version

a comment

**Identifier:** [a-z\_+#@]\*

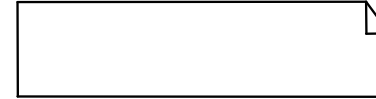
**Channels and signal route are typed**

**No shared variable**

**Components are visible by the parents**

# Declarations

**Declarations are made in block text:**



**At system level, declaration of signal**

**SIGNAL <ident> (, <ident>)\***

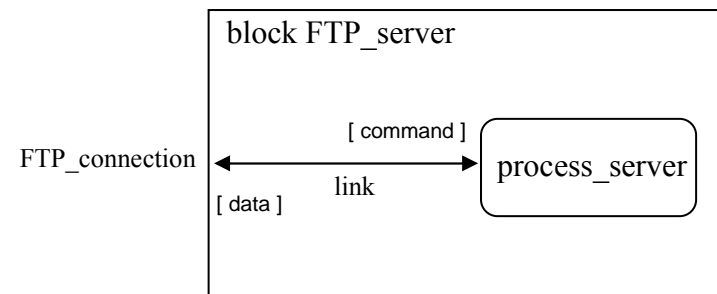
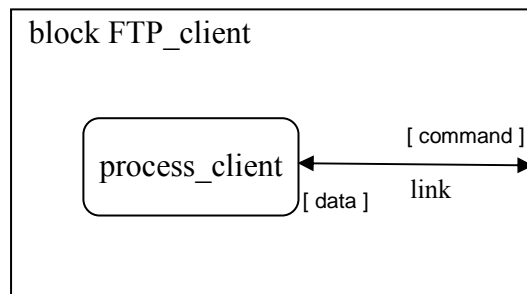
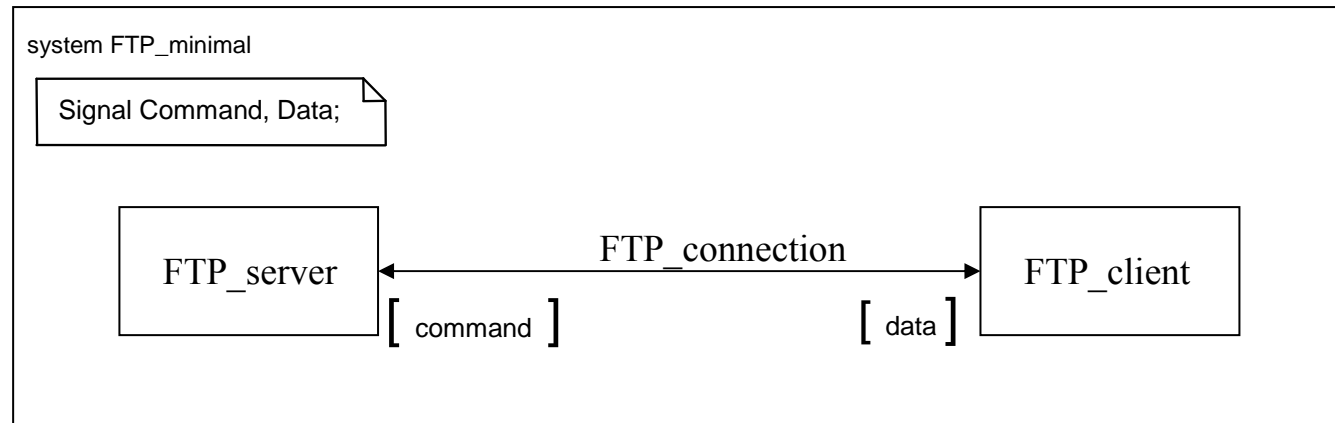
Signal Command, data;

**A signal is a type, a signal occurrence is a typed message**

**Elementary types:**

- Boolean: true, false
- Character: 'A', '1', ...
- Integer:  $\mathbb{Z}$
- Natural:  $\mathbb{N}$
- Real
- Charstring
- Pid: identifier of process (2 operators =, /=)
- Time

## Example 2

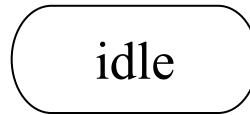


Verification of the coherence of the exchanges

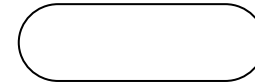
# Definition of the behaviour in the process

A process is a finite machine state where the transition depend on the reception of some signal.

Symbol of a state:



(initial state



)

Symbol of reception:

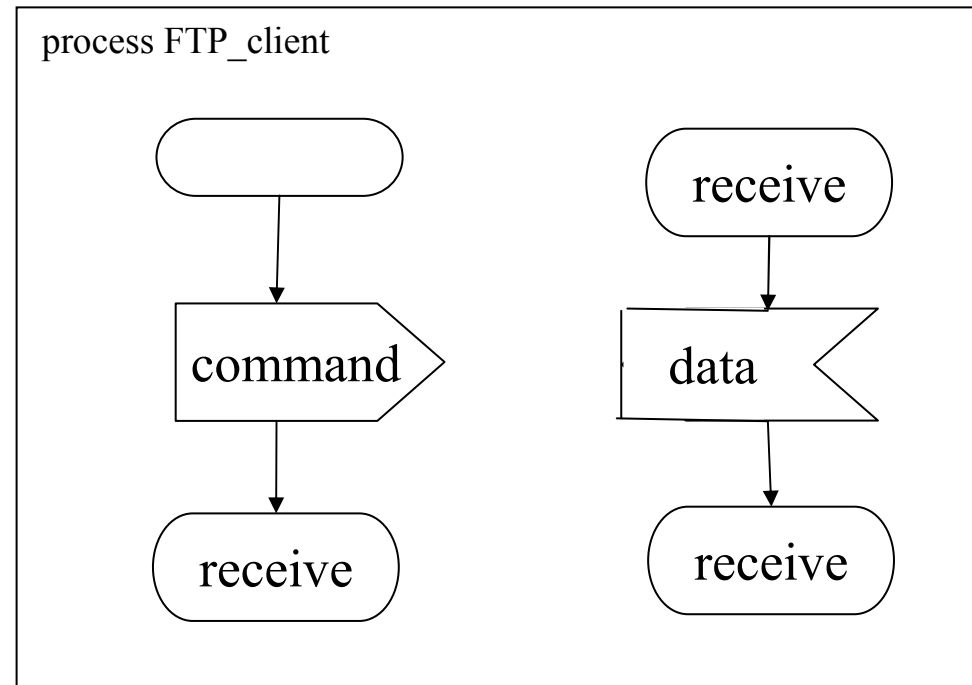


Symbol of emission:

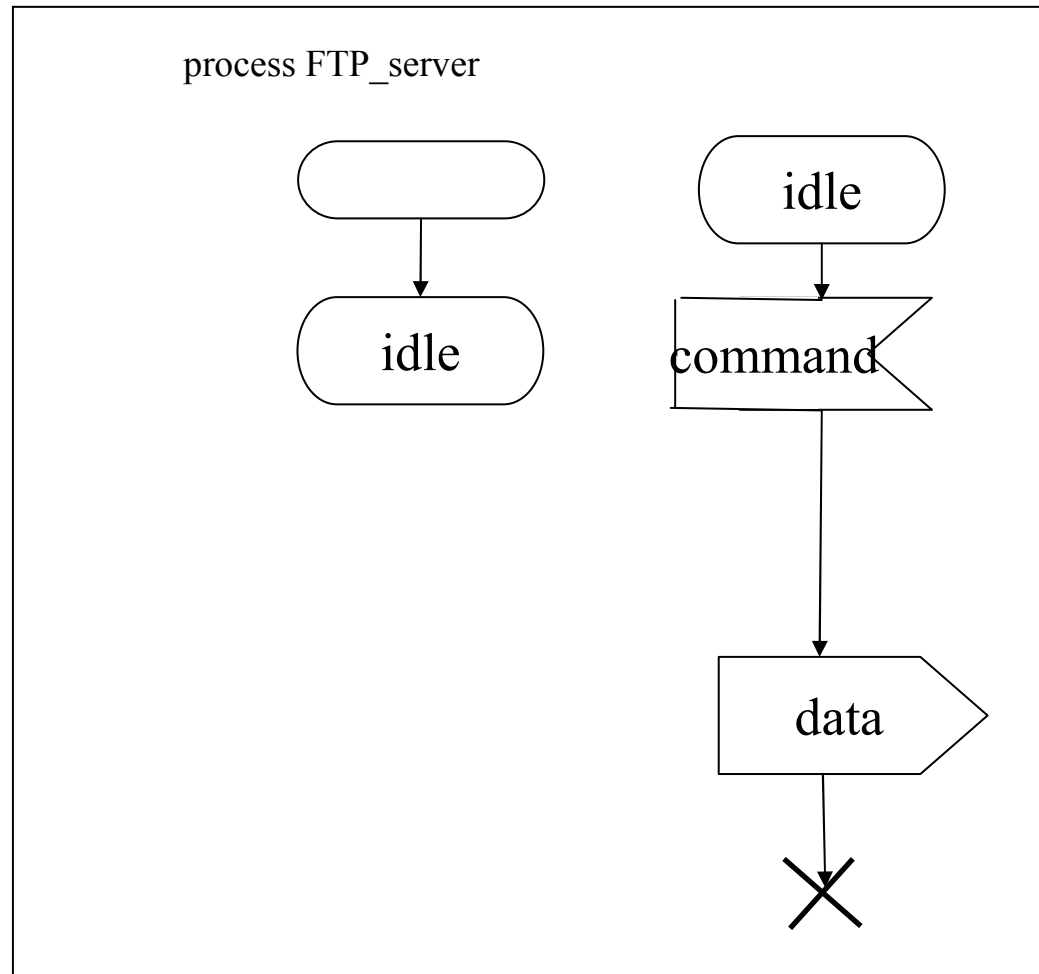




## Example: process of the client



## Example: process of the server

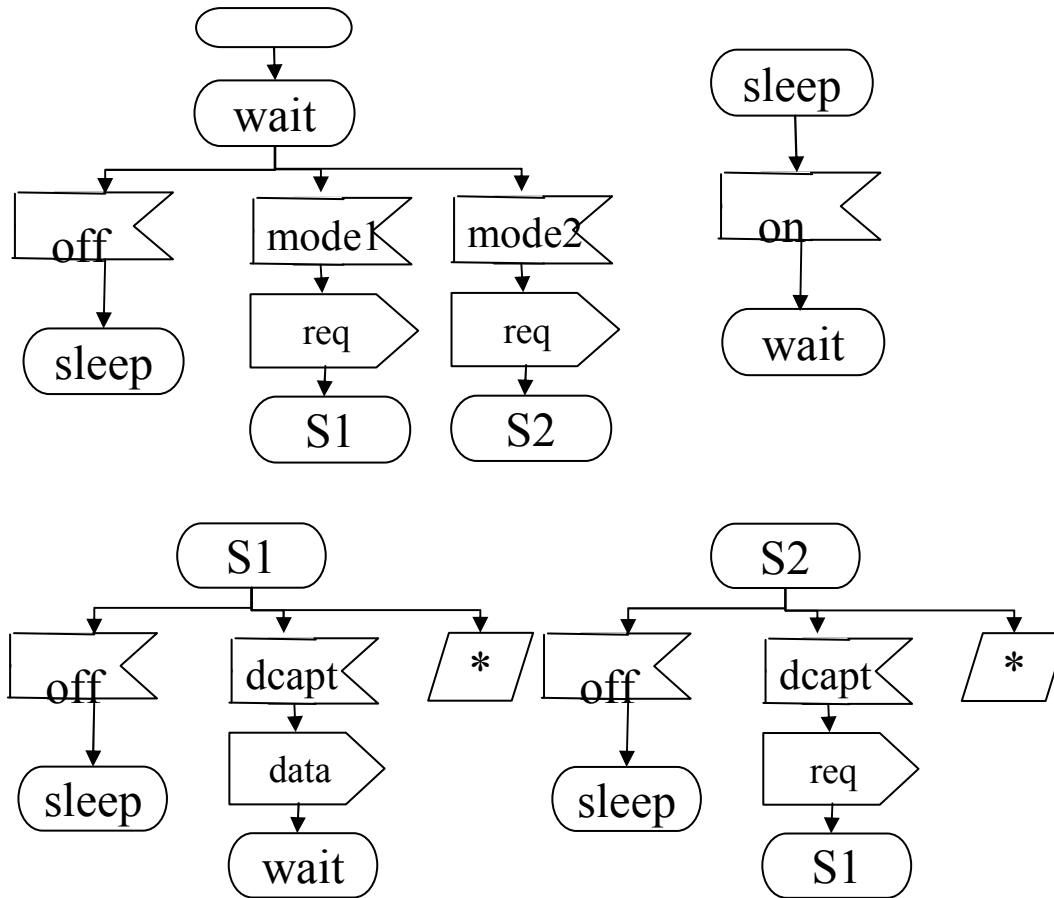


## Semantics of a process

An SDL process is transition system  $S = \langle \Sigma, s_0, T, \text{Sig}_{in}, \text{Sig}_{out}, Sv \rangle$ :

- $\Sigma$  is a finite set of states
- $s_0 \in \Sigma$  is the initial state
- $\text{Sig}_{in}$  and  $\text{Sig}_{out}$  are input and output data of S
- $T \subseteq \Sigma \times \text{Sig}_{in} \times \text{Sig}_{out}^* \times \Sigma$  is the set of transitions. A transition is  $(s_1, a, \sigma, s_2)$  where  $s_1$  is the source,  $a$  is the triggering event,  $\sigma$  is the sequence of output events produced by the process,  $s_2$  is the reached state
- $Sv : \Sigma \rightarrow 2^{\text{Sig}_{in}}$  is the function that associates to each state the saved input events.

# Example



$\Sigma = \{\text{init, wait, sleep, S1, S2}\}$

$\text{Sig}_{\text{in}} = \{\text{off, on, mode1, mode2, dcapt}\}$

$\text{Sig}_{\text{out}} = \{\text{req, data}\}$

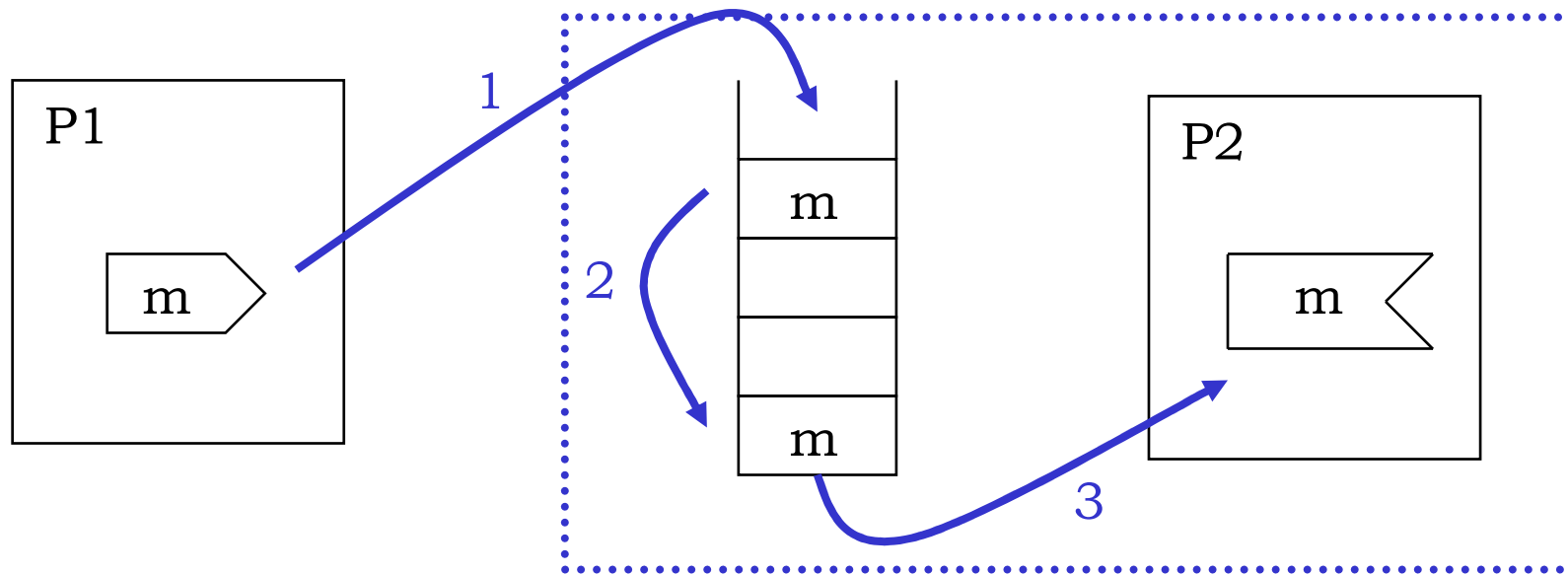
$T = \{ (\text{wait, off, ()}, \text{sleep});$   
 $(\text{wait, mode1, (req)}, \text{S1});$   
 $(\text{wait, mode2, (req)}, \text{S2});$   
 $(\text{sleep, on, ()}, \text{wait});$   
 $(\text{S1, off, ()}, \text{sleep});$   
 $(\text{S1, dcapt, (data)}, \text{wait});$   
 $(\text{S2, off, ()}, \text{sleep});$   
 $(\text{S2, dcapt, (req)}, \text{sleep});$

$\text{Sv}(\text{wait}) = \text{Sv}(\text{sleep}) = ();$

$\text{Sv}(\text{S1}) = \text{Sv}(\text{S2}) = \{\text{on, mode1, mode2}\}.$

# Communication semantics

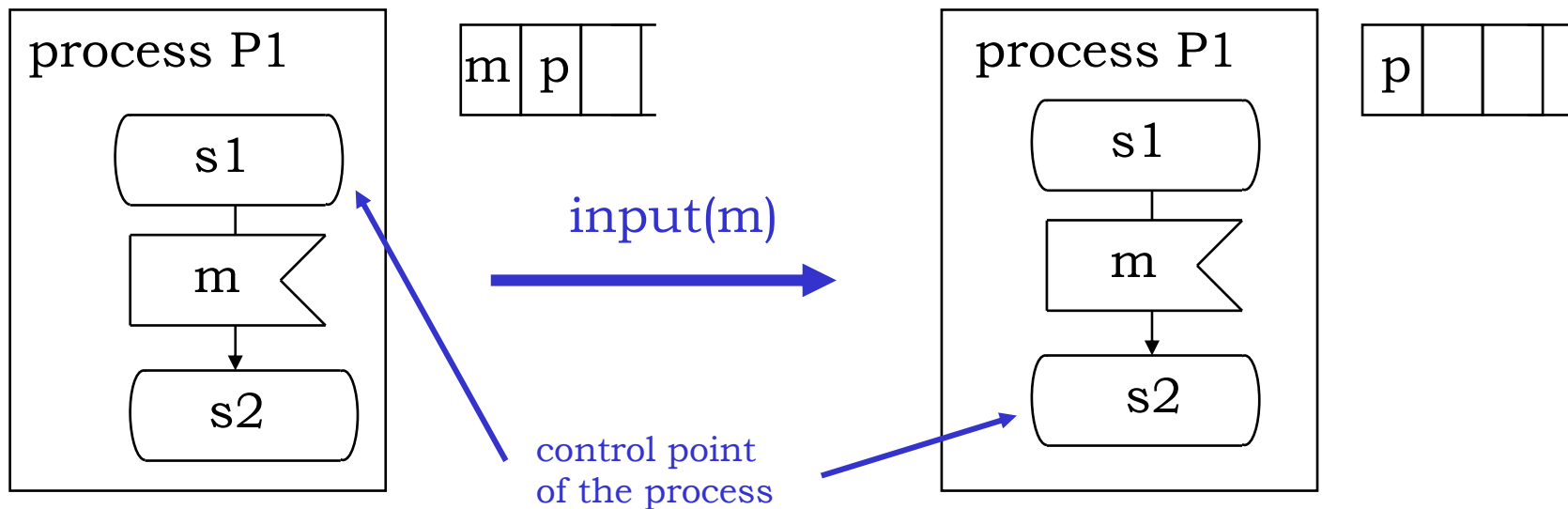
- Communication between processes is asynchronous
- To each process, is associated a unique unbounded FIFO file



# Communication semantics

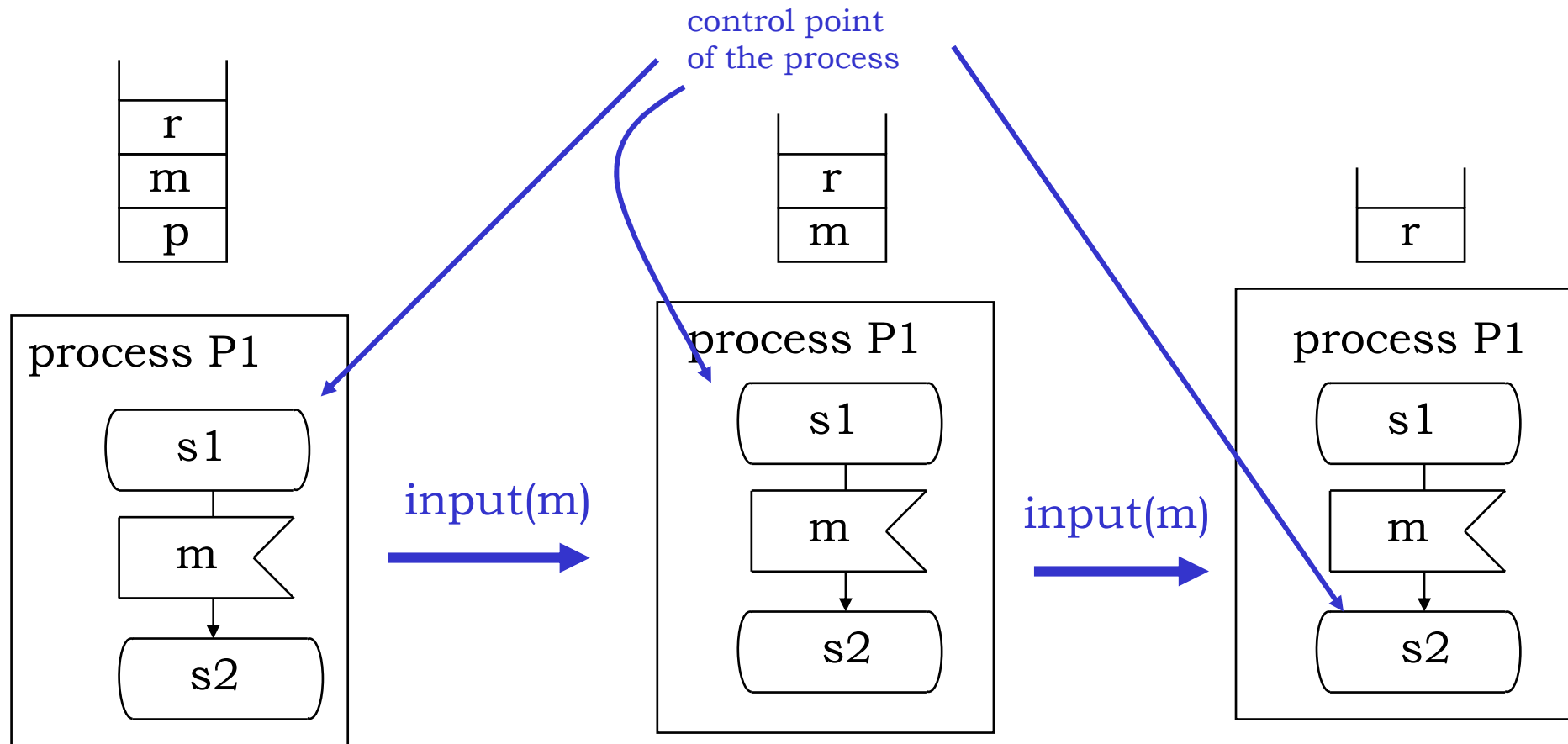
## At the reception of a message:

- The message is removed from the file
- Progress of the process



# Communication semantics

Unexpected messages are deleted.



## Signal handling

The signal received by a process are stored in a FIFO.

In a state, only a subset of signals are awaited. By default, any signal in the head of the FIFO which is not awaited is deleted.

A back-up is possible by the use of the symbol





# Variables declaration

In a block text.

**DCL** <ident> (,<ident>)\* <type> (, <ident> (<ident>)\* <type> )\*

```
DCL a Character;  
DCL b Character, i integer;  
DCL c,d Character, n,m Natural ;
```

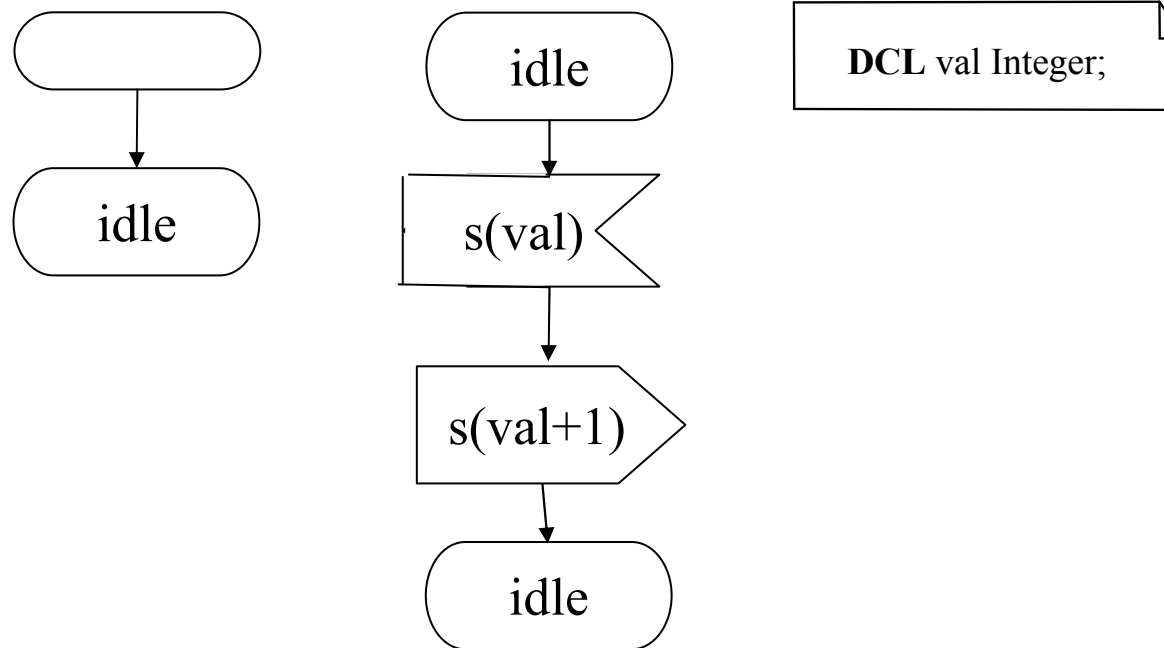
# Declarations

## Signals can transport some values:

Signal Data(Integer, Boolean);

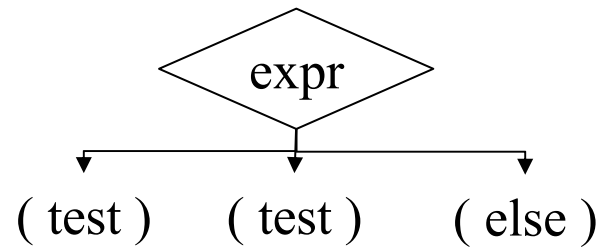
Signal Ack(Boolean);

## Example: SIGNAL s(Integer);

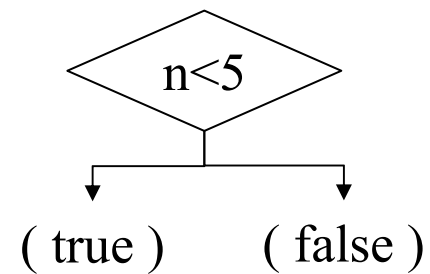
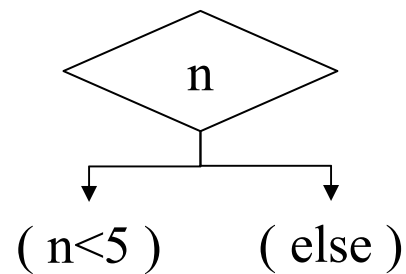


# Decision

During a transition (reception of a signal), it is possible to make a decision depending on some expression.



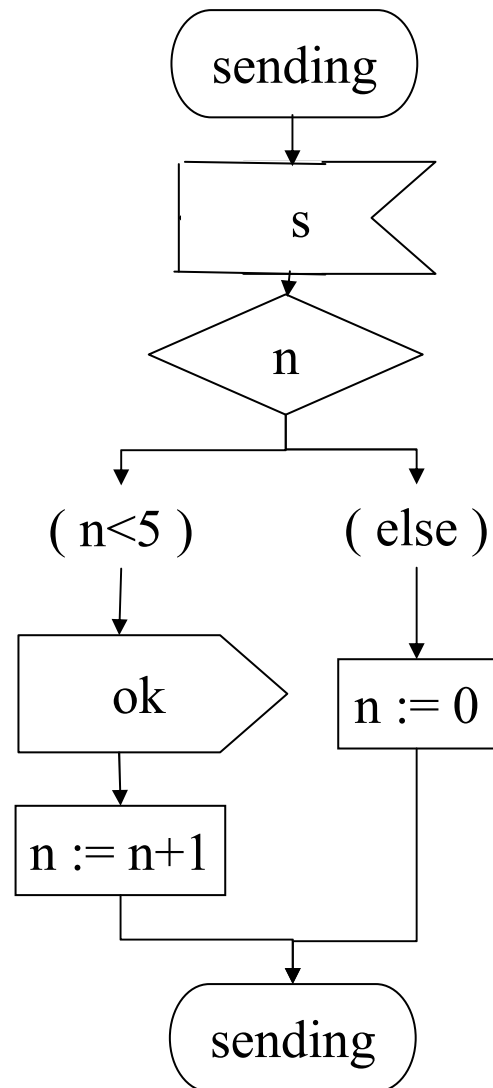
## Examples



# Tasks

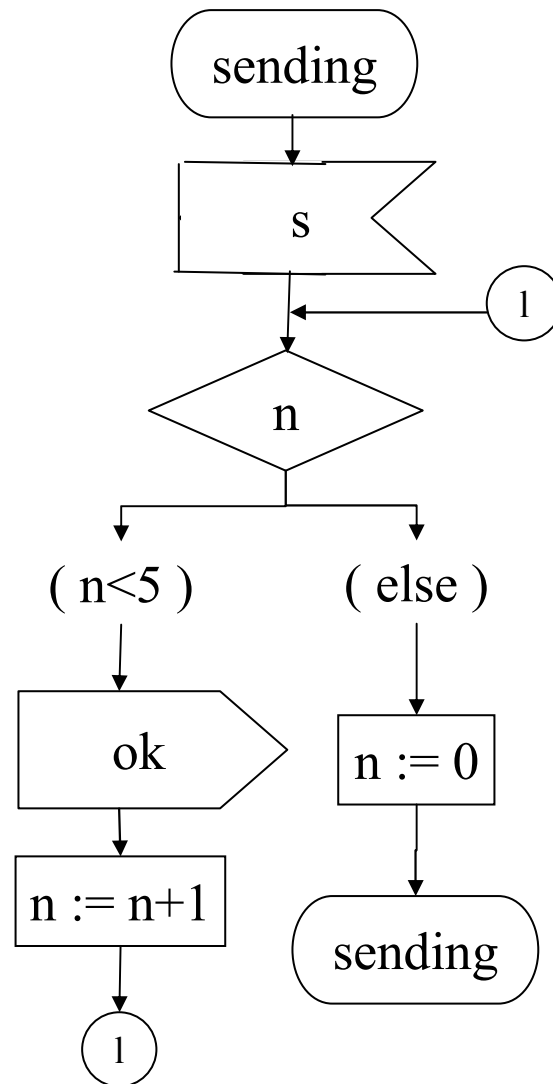
After a transition, it is possible to make some computation step

task



# Loops

At any time after entering in a transition, it possible to put a loop

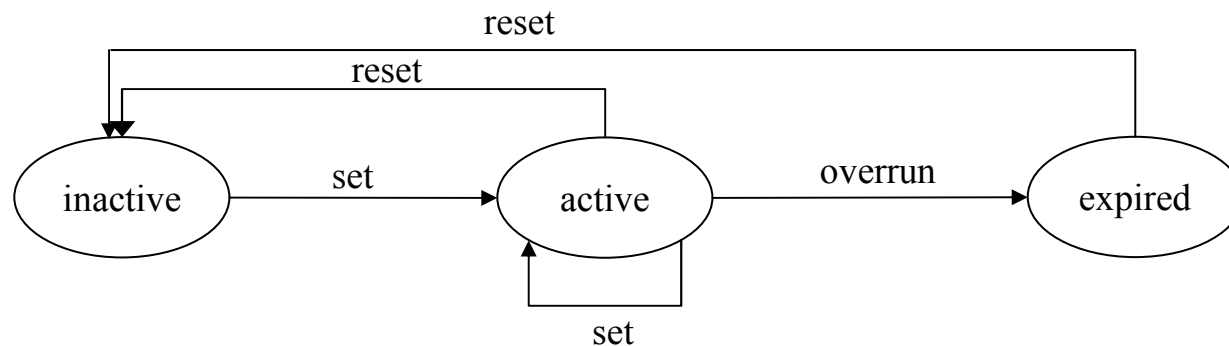
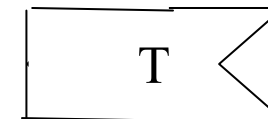


# Timers

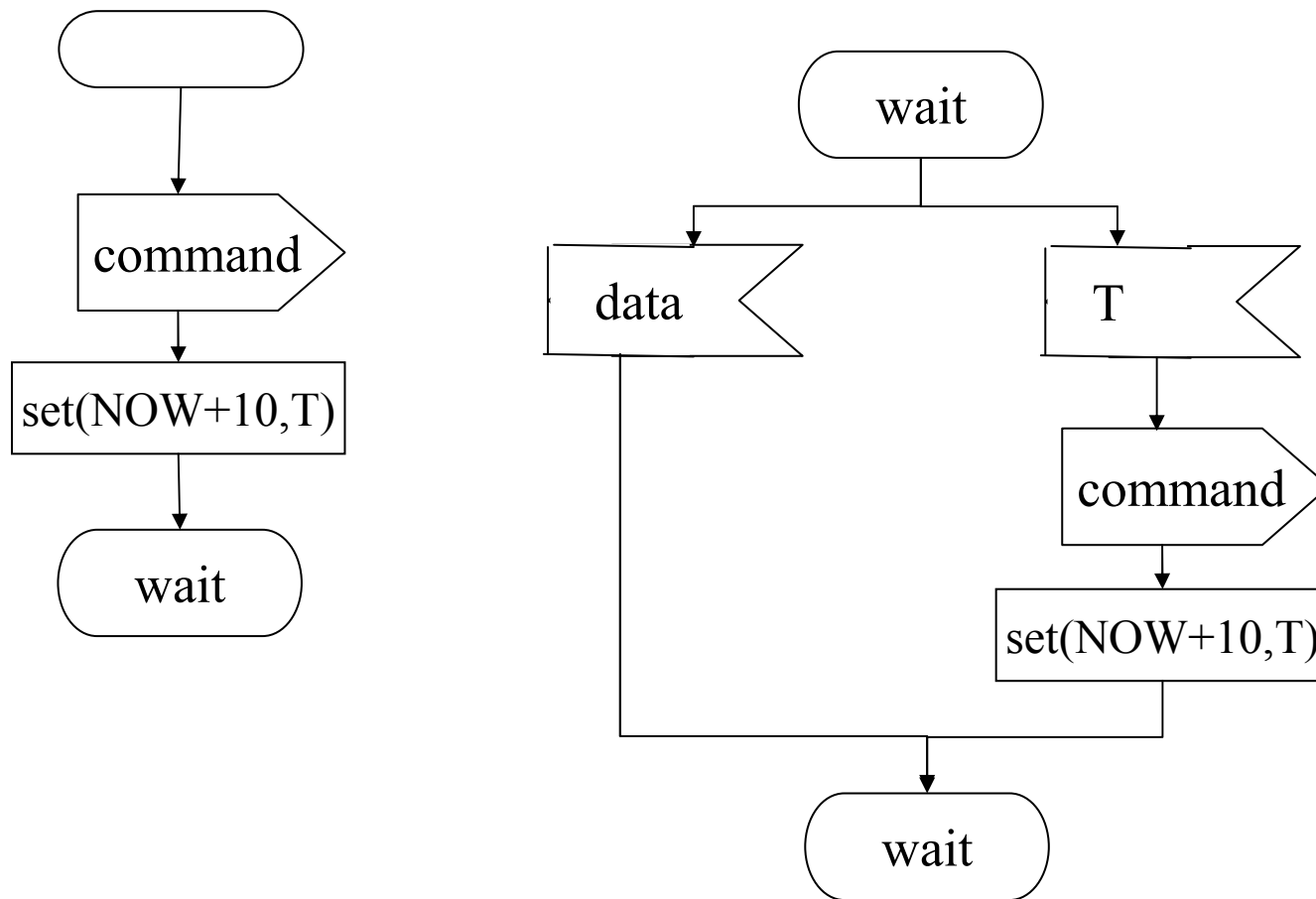
Variables of type **TIMER**. Such a variable has 3 states: *inactive*, *active* and *expired*.

When declared, a timer is inactive. We set a timer by giving it a delay:  
**SET (<delay>,<timer>)**

When the delay is overrun, the timer is expired.



## Example: process of the client



## **Outline – II.2 SDL**

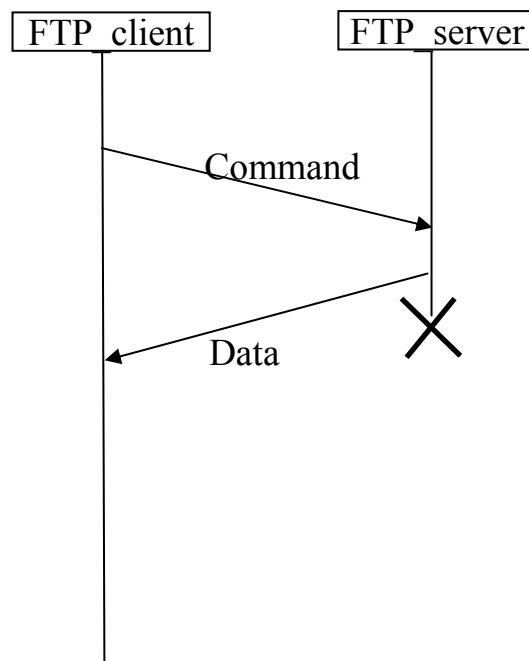
- 1. Where is it possible to apply a language?**
- 2. Introduction of SDL**
- 3. Syntax and semantics of SDL**
- 4. Simulation and MSC**



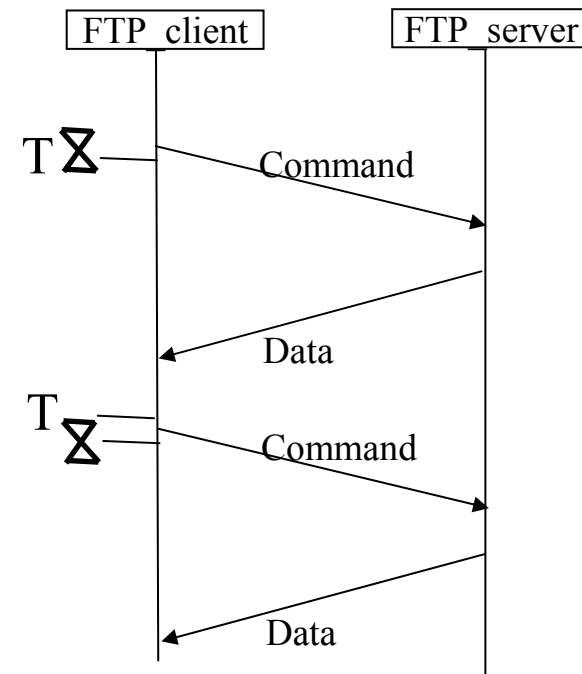
# Tools

At ENSEEIHT: RTDS par Pragma-dev

Using the simulator, generation of MSC (Message Sequence Chart)



Version 1



Version 2

## Exercise

**Define a system composed of an emitter, a receiver and a medium. The emitter sends a data and the receiver replies a ack for each reception. The medium is not reliable and loses a data each 3 emissions and duplicates an ack each 5 emissions.**

**Model this system in SDL. This will be useful for the next practical session.**

## References

1. **Marc Boyer: course on SDL.**
2. **Ahmed Bouadballah: course on SDL.**
3. **Jan Ellsberger, Dieter Hogrefe and Amardeo Sarma: SDL formal object-oriented language for communicating systems, Prentice hall, 1997.**
4. **Laurent Doldi. SDL illustrated – visually design executable models. Number ISBN 2-9516600-0-6. 2001.**
5. **Specification and description language (SDL). Recommendation Z.100, International Telecommunication union, 1999.**
6. **Eric Bonnafeous, Frédéric Boniol, Philippe Dhaussy et Xavier Dumas. Experience of an efficient and actual MDE process : design and verification of ATC onboard system, 2008, Conference on UML&FORMAL METHODS, Kitakyushu-city, Japan**
7. **Xavier Dumas. Application des méthodes par ordres partiels à la vérification formelle de systèmes asynchrones clos par un contexte: application à SDL. Thèse 2011.**

# Outline

1. Part I - What is a real-time system?
2. Part II – High level formal programming languages
3. Part III – Uniprocessor and multiprocessor scheduling

# Outline - Part III - Scheduling

1. **First definitions**
2. **Uniprocessor real-time scheduling**
3. **Multiprocessors real-time scheduling**

# Definitions

**Functional architecture:** set of communicating functions with a data-flow

**Software architecture:** set of applicative programs (or tasks) to be executed (some sequentially and others concurrently). Tasks and constraints are derived from the functional architecture.

**Material architecture:** limited set of heterogeneous interconnected resources (calculators, bus ...).

**Scheduling:** spatial and temporal assignment of resources to the tasks with respect to the constraints (performance, location ...). (e.g. for a monoprocesor, management of the sequence of tasks execution while optimising the CPU occupation).

# How to model a task?

## A model should be :

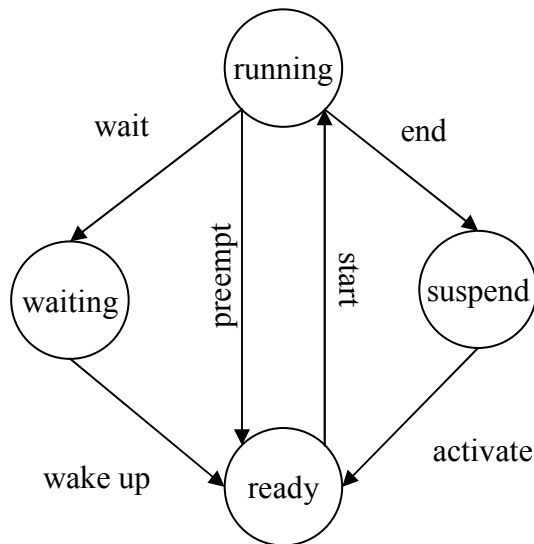
- accurate and precise enough to express the features of the application;
- exploitable for validation.

## Several models have been proposed:

- Liu et Layland (1973)
- Mok (1983)
- Mok et Chen (1996) : multiframe model
- Baruah (1998) : recurring branching task model
- Baruah et al. (1999) : GMF = generalized multiframe ...

**A task will be denoted by  $t_i$  and a software architecture will be represented by a set of tasks,  $T = \{t_i \mid i = 1..n\}$**

# Model of a task



- **Running:** the task is allocated to a processor and is executed. Only one task can be running at a time on a processor
- **Suspend:** the task is passive and can be activated
- **Ready:** the task is ready to execute and waits for the resource to start. The scheduler chooses among the ready tasks which one will start next.
- **Waiting:** the task is blocked and is waiting for an event

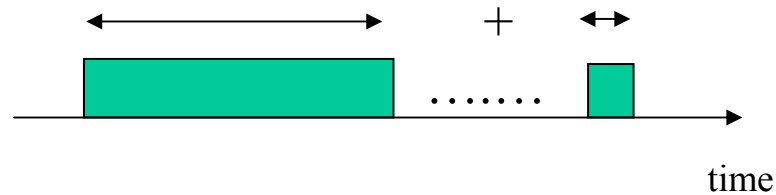


# Execution time

**Definition:** duration necessary for a processor to execute the code of task integrally without preemption

- Worst case execution time (wcet): maximal duration
- Best case execution time (bcet): minimal duration

**For any execution,  $bcet \leq d \leq wcet$**



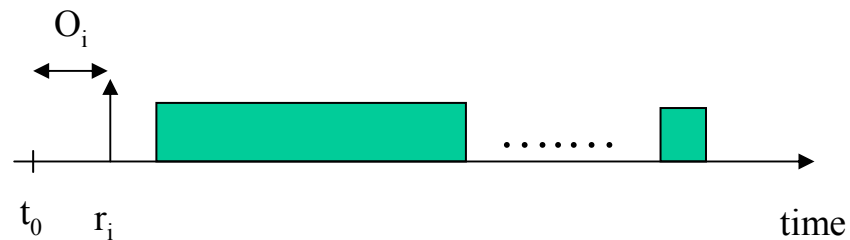
# Release time

## Release time (or arrival time):

- Instant when a task becomes ready, denoted by  $r_i$

## Offset (or initial date):

- Delay after which a task becomes ready after the start of an application, denoted by  $O_i$



## Synchronous tasks

- If  $r_i = t_0$

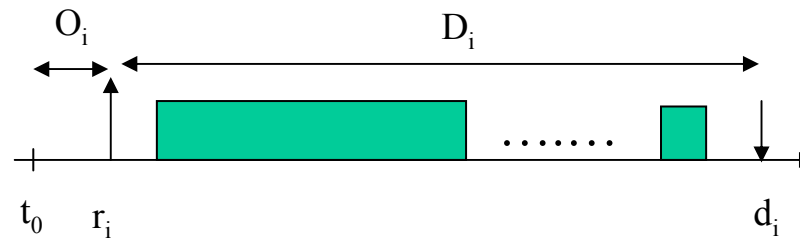
# Deadline

## Relative deadline:

- Maximal delay for the execution of a task, denoted by  $D_i$ .

## Deadline (or absolute deadline):

- Date before which the execution of a task must be terminated, denoted by  $d_i$  :  $d_i = r_i + D_i$



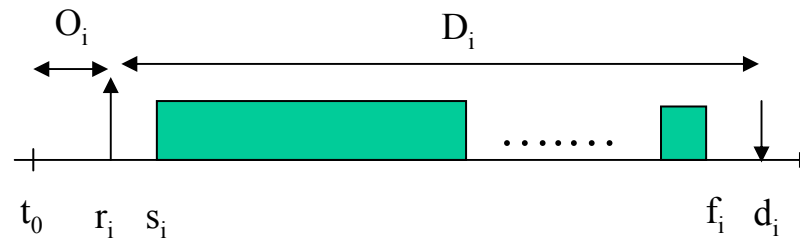
# Start and end of an execution

## Start time:

- Instant when the execution starts, denoted by  $s_i$

## Finishing time (or completion time)

- Instant when the execution terminates, denoted by  $f_i$ .

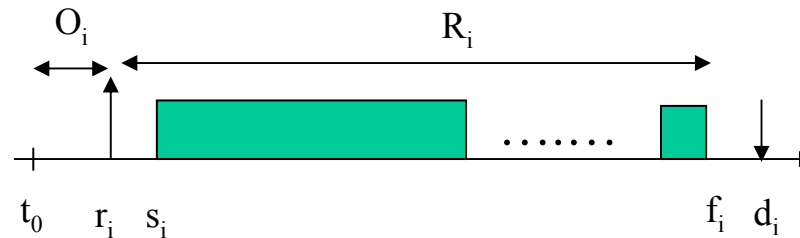


# Response time

## Response time:

- Delay between the activation and the end of an execution, denoted by  $R_i$

$$R_i = f_i - r_i$$



# Repetitive task

## Repetitive task:

- Task which executes several times

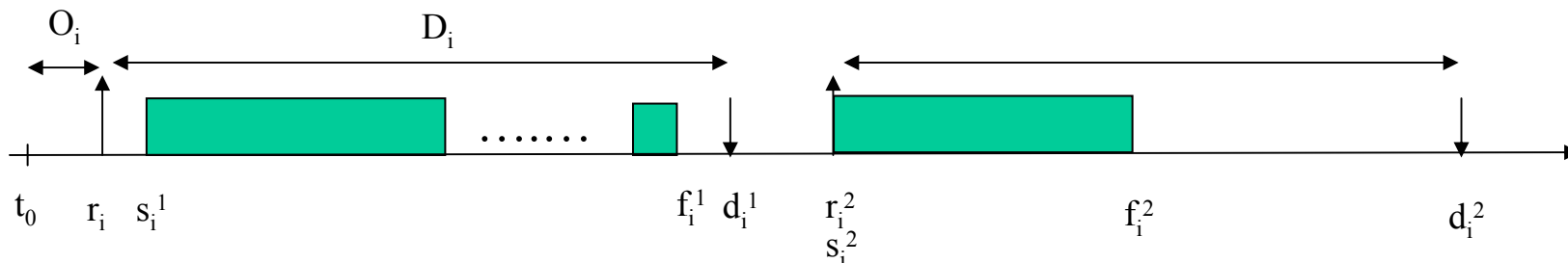
## Task instance (or job):

- An execution of a task. The  $k$ th instance of a task  $t_i$  is denoted  $t_i^k$

## The execution of a task is given by a set of:

- Release times  $\{r_i^1 = r_i, r_i^2, r_i^3 \dots\}$
- Deadlines:  $\{d_i^1, d_i^2, \dots, d_i^k = r_i^k + D_i \dots\}$
- Start times:  $\{s_i^1, s_i^2, s_i^3 \dots\}$
- Finishing times:  $\{f_i^1, f_i^2, f_i^3 \dots\}$

$$\begin{aligned} r_i^k &> r_i^{k-1} \\ f_i^{k-1} &< s_i^k \end{aligned}$$



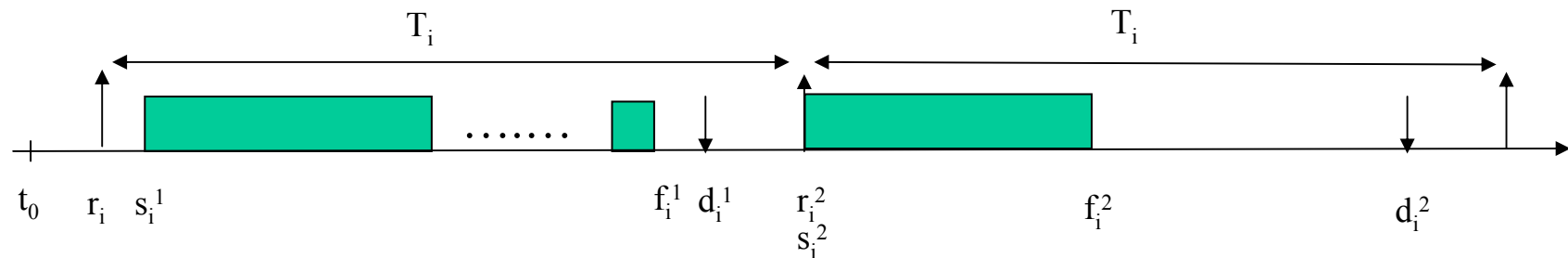
# Periodic task

## Periodic task:

- Time interval between 2 activations is constant, of period  $T_i$

$$r_i^1 = r_i$$

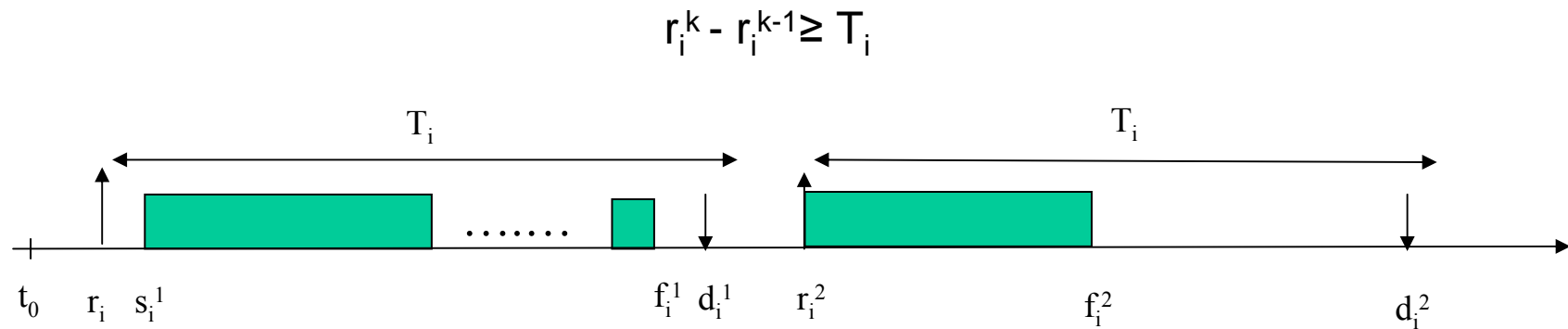
$$r_i^k = r_i^1 + (k-1) T_i$$



# Non-periodic task

## Sporadic task:

- Time interval between 2 activations is more than a value, named inter arrival time denoted by  $T_i$



## Aperiodic task:

- No constraint on the activation times



# Real-time constraints

## Hard real-time task:

- A finishing time that exceeds a deadline ( $f_i^k > d_i^k$ ) has catastrophic consequences

## Soft real-time task:

- A finishing time that exceeds a deadline ( $f_i^k > d_i^k$ ) reduces the performance of the system

## (m,k)-firm task:

- At least  $m$  instances among  $k$  consecutive instances ( $m < k$ ) must respect their deadlines

## Examples :

- ABS control system: hard real-time
- Emission of multimedia stream on internet: contrainte  $(m, k)$ -firm
- Air conditioning control system: soft real-time

# Outline - Part II - Scheduling

1. First definitions
2. Uniprocessor real-time scheduling
3. Multiprocessors real-time scheduling

## Outline – III.2 – Uniprocessor scheduling

1. **Recalls**
2. **Real-time Scheduling**
3. **Priority-based Scheduling**
4. **Scheduling with shared resources**

# Definitions

## Predictability:

- The application performances must be defined in the worst case for any possible behaviour in order to ensure the respect of timed constraints

## Determinism:

- There is non uncertainty on the behaviour: this behaviour is always the same in a given context

## Reliability:

- ability of a system to perform and maintain its functions in normal circumstances. For real-time, reliability refers to the timed constraints respect. We may also want the system to remain reliable even when some failures occur: we then speak of fault tolerant system.

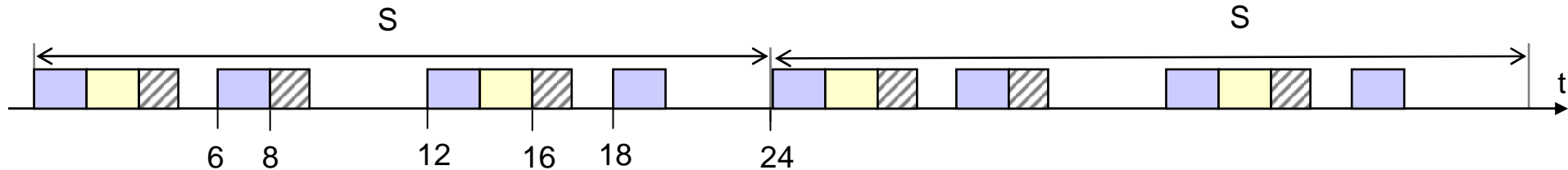
# Periodic sequencing

For a set of periodic tasks  $T = \{\tau_1, \tau_2, \dots, \tau_n\}$  which are sequenced off line. The sequence is made over a feasibility interval (or a meta period)  $H$ .

**Case:** for all  $i$   $r_i = 0$

$$H = \text{lcm}(T_i)$$

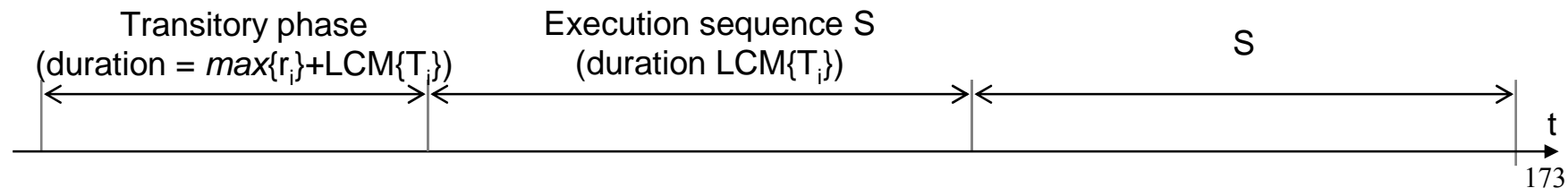
Task	$r_0$	C	D	T
1	0	2	6	6
2	0	1	8	8
3	0	2	10	12



**General case:**

$$H = \max(r_i) + 2 * \text{lcm}(T_i)$$

Task	$r_0$	C	D	T
1	0	2	6	6
2	2	1	8	8
3	0	2	10	12



# Processor utilisation

## Processor utilisation

- For a task  $U_i = C_i / T_i$
- In general: fraction of processor time spent in the execution of the task set

$$U = \sum_{i=1}^n C_i / T_i$$

## Example

Task	$r_o$	C	D	T
1	0	2	6	6
2	2	1	8	8
3	0	2	10	12

- $U_i =$

**Schedulability: if a set of tasks is schedulable then  $U \leq 1$**

# Dynamic parameters

## Remaining execution time

- remaining time of execution  $C(t) = C_{\max} - C_{\text{executed}}$

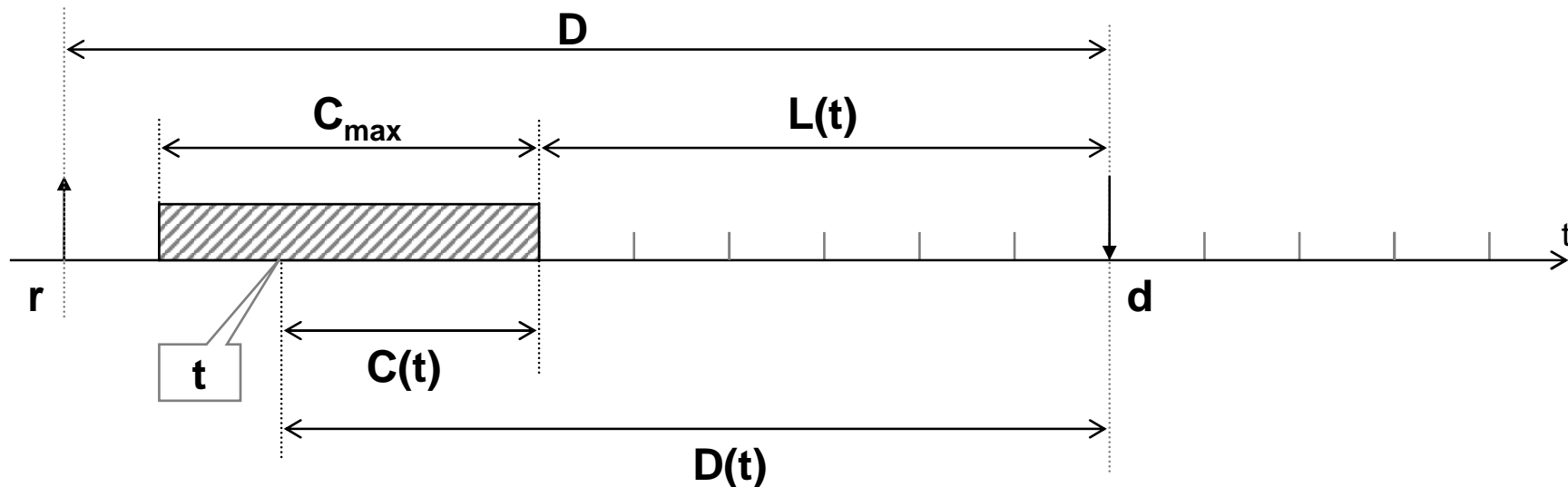
## Critical dynamic delay

- remaining time before the deadline  $D(t) = d - t$

## Dynamic laxity

- spare time leaving by the task  $L(t) = D(t) - C(t)$

*Temporal execution diagram*



# Scheduling overview

## Scheduling policy:

- Algorithm by which tasks are given access to system resources (e.g. processor time, communications bandwidth).

## Valid scheduling:

- The scheduling of a set of tasks is said to be valid if and only if no task instance misses its absolute deadline.

## Schedulability:

- A set of tasks is said to be feasible with respect to a given class of schedulers (we consider 4 classes preemptive/non-preemptive, fixed/dynamic priority) if and only if there exists at least one valid schedule for this class.

## Activities:

- For a given system, there are 2 main activities: choice of the scheduling policy and validation of this policy on the set of tasks.



## **Outline - III.2 – Uniprocessor scheduling**

- 1. Recalls**
- 2. Real-time scheduling**
- 3. Priority-based Scheduling**
- 4. Scheduling with shared resources**

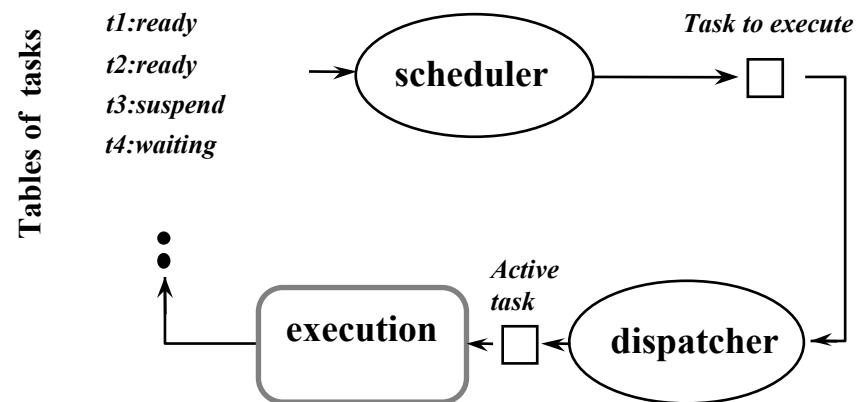
## Scheduler internal structure

## Scheduler:

- the kernel module which applies the scheduling algorithm and handles the tables

## Dispatcher:

- the kernel module which realises the effective activation and the context change



# Criteria associated to a scheduling

## Efficiency:

- the processor must spend the maximum time in executing the application (minimisation of the overhead)

## Response time (adaptability):

- time between submission of requests and first response to the request (reaction to an external interruption, taking into account of changes ...)

## Predictability:

- guaranty of the delays, capacity of predicting the behaviour

## Flexibility:

- ability to dynamically reallocate units of resource, fault tolerance ...

...

# 4 classes of scheduling

## Off-line scheduling (or static, or pre-run-time)

- Scheduling decision are taken before the application execution. The sequence is pre-computed.
- ➡ – Sequencing (simplest case)
- Not much flexible

## On-line (or dynamic) scheduling:

- Scheduling decision are taken during the execution
- ➡ – Additional cost of the scheduler
- Very flexible

## Non preemptive policy:

- Tasks cannot be interrupted (except on their demand)
- ➡ – Important response time
- No mechanism for the shared critical resources
- Easy programming

## Preemptive policy:

- Tasks can be interrupted at any time and the processor allocated to an other task
- ➡ – Better response time
- Mechanism for shared resources
- Context handling

## Exercises

**Are the previous classes incompatible?**

**Some sets of tasks can be scheduled for a preemptive policy but not for a non preemptive one. Give an example?**

# General purpose vs real-time scheduling

## General purpose scheduling:

- Avoid situation of starvation
- Improve the average performance
- Examples: round robin, multi-level feedback (MLF)

## Real-time scheduling:

- Guaranty the respect of deadlines
- Even in the worst case

# Classical scheduling policies

## First Come First Served (FIFO)

- Non preemptive

## Shortest Job First

- Non preemptive

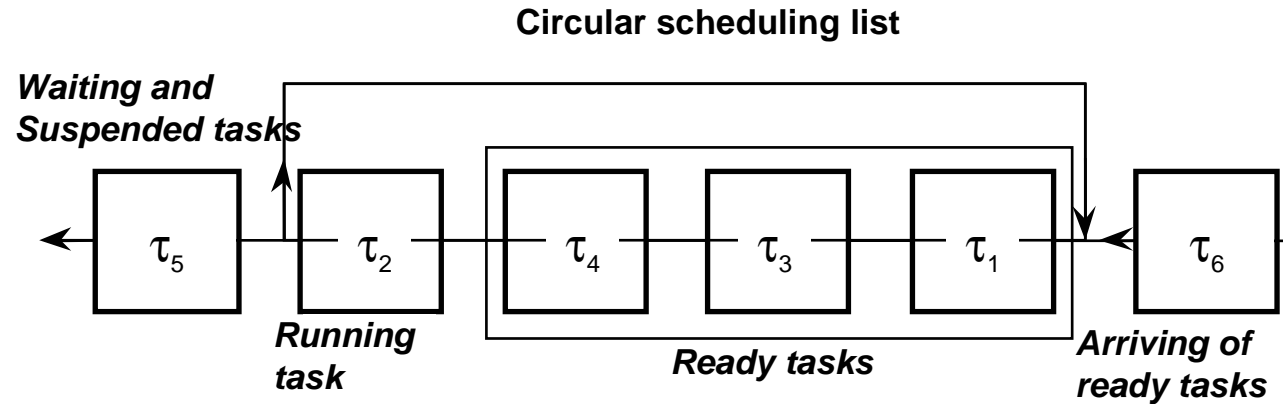
## Best effort

- Preemptive





## Round robin policy

- Preemptive

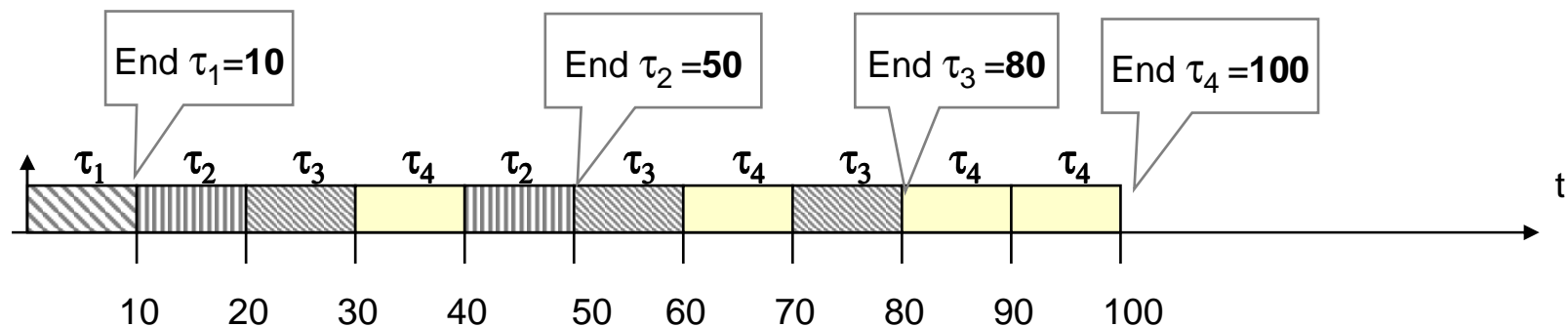
# Round Robin strategy



Configuration

Tasks	$\tau_1$	$\tau_2$	$\tau_3$	$\tau_4$
Execution time	10 ms	20 ms	30 ms	40 ms
				

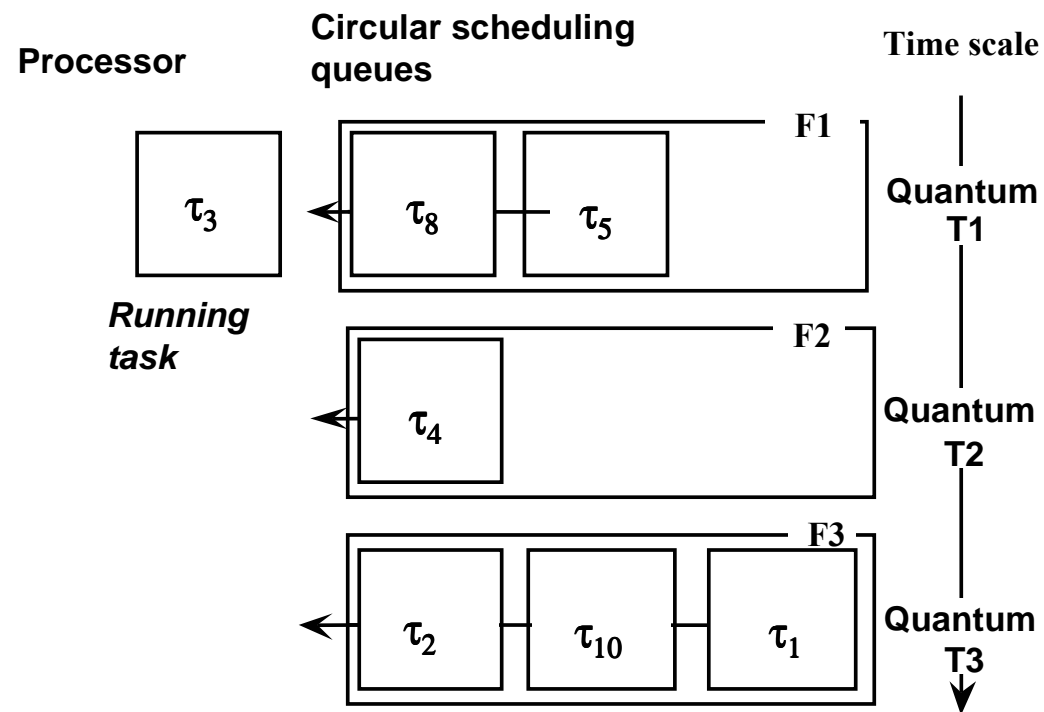
Scheduling: time quantum 10 ms










## Round Robin with several queues

M queues will execute in a circular manner with each other. Usually the quantum are  $4 \times T_0$ ,  $2 \times T_0$ ,  $T_0$ . The sequence is then **F1 F2 F3 F1 F2 F3...**

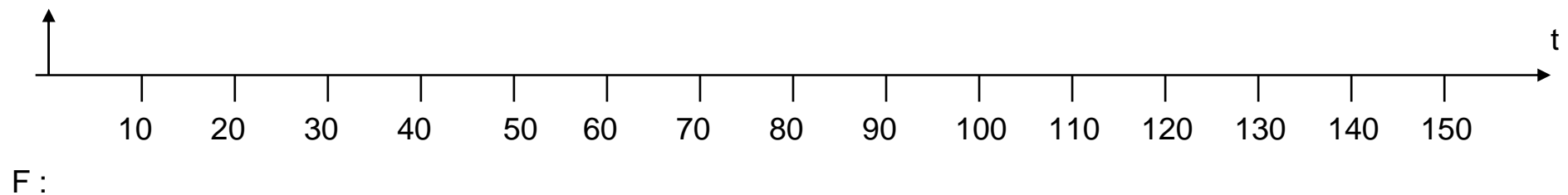


## Example of circular scheduling

### Configuration

Tasks	$\tau_1$	$\tau_2$	$\tau_3$	$\tau_4$	$\tau_5$
Execution time	10 ms	20 ms	30 ms	40 ms	50 ms
					
	File F1 (quantum <b>20 ms</b> )		File F2 (quantum <b>10 ms</b> )		

### Scheduling



## Outline - III.2 – Uniprocessor scheduling

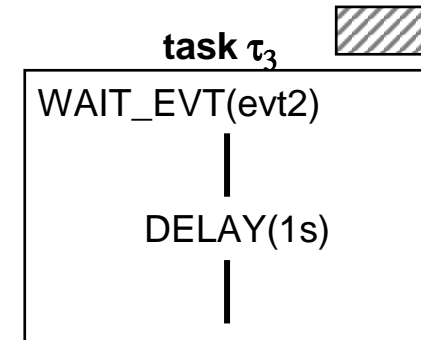
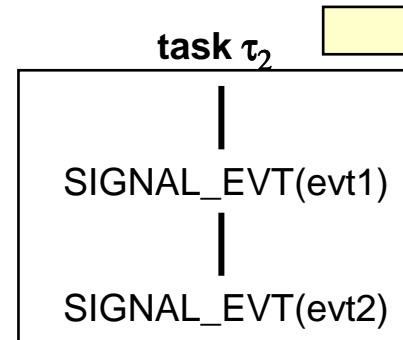
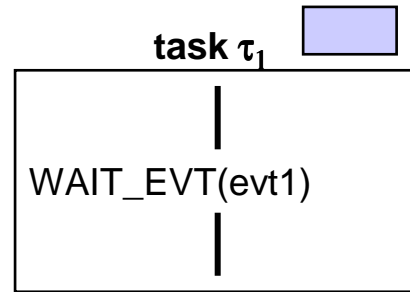
1. Recalls
2. Real-time scheduling
3. **Priority-based Scheduling**
4. Scheduling with shared resources

## Priority-based scheduling

**Scheduler is often provided with several level of priorities. It means that it allocates the processor to the task with the highest priority. The designer must realise the priority allocation of the task during the execution. Note that 2 tasks can have the same level of priority. The decision is often empirical.**

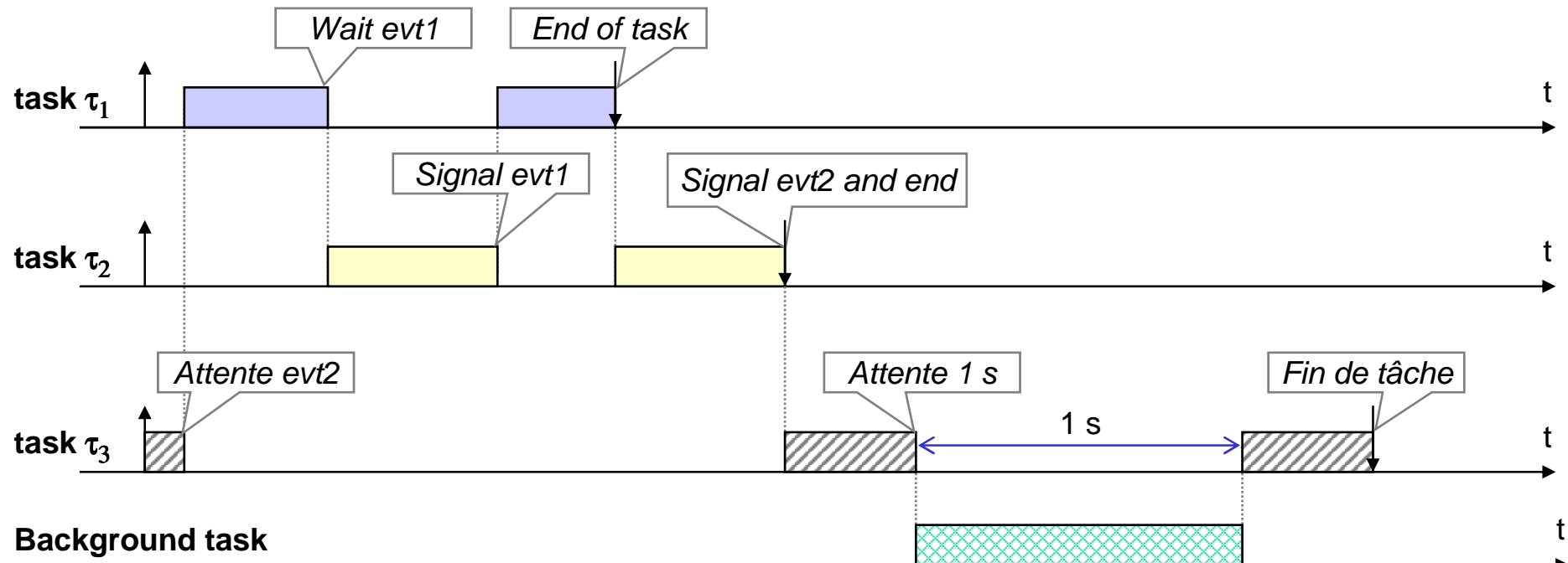
# Priority-based scheduling: example

**Configuration :**  
(3 tasks)



**Scheduling**

$\text{Prio}(\tau_1)=2$  -  $\text{Prio}(\tau_2)=1$  -  $\text{Prio}(\tau_3)=3$



# Classical algorithms for choosing the priority

## Constant priority:

1. Rate monotonic (RM)
2. Deadline monotonic (or inverse deadline, DM)
3. Audsley algorithm
4. ...

## Dynamic priority:

1. Earliest deadline first (EDF)
2. Least laxity (LL)
3. ...

# Optimality

A scheduling policy  $P$  of a family of scheduling  $C$  is optimal for the applications of a class  $T$  if for all application  $A$  in  $T$ ,  $A$  is not schedulable by  $P$  implies that  $A$  is not schedulable for any algorithm in  $C$ .

**Ex:** The scheduling policy **RM** is optimal of the family of fixed priority preemptive scheduling for independent non blocking tasks with  $D=T$  and  $r=0$ .

## Corollary :

- Any application  $A$  in  $T$  schedulable by an algorithm in  $C$ , is also schedulable by  $P$ .

# Rate monotonic (Liu & Layland 1973)

**Hypothesis:**

**Periodic independent tasks with simultaneous start (for all  $i$   $r_i = 0$ )**

**Priority:**

**given to the smallest period**

**Example:**

**( $r=0$ ,  $C=3$ ,  $T=20$ )**

**( $r=0$ ,  $C=2$ ,  $T=5$ )**

**( $r=0$ ,  $C=2$ ,  $T=10$ )**

**$U=?$**

**Scheduling?**

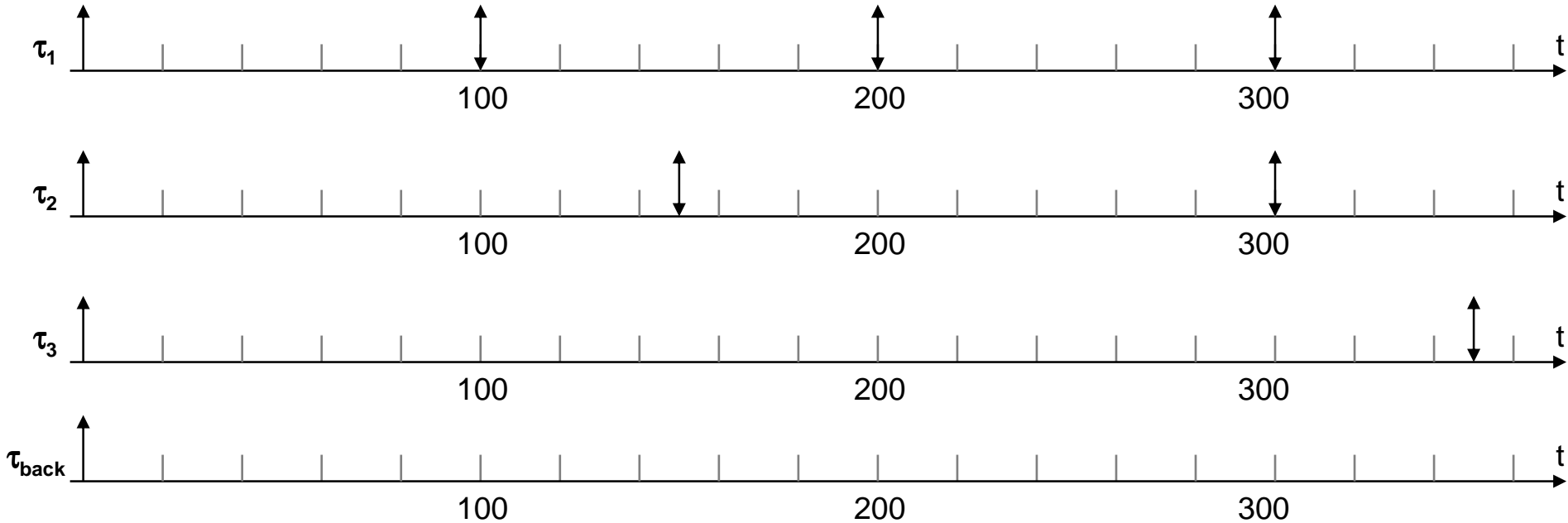


# Exercise 1 on RM

$T$	$r_i$	$C_i$	$D_i$	$T_i$	$u_i$	$Prio_i$
$\tau_1$	0	20	100	100		
$\tau_2$	0	40	150	150		
$\tau_3$	0	100	350	350		

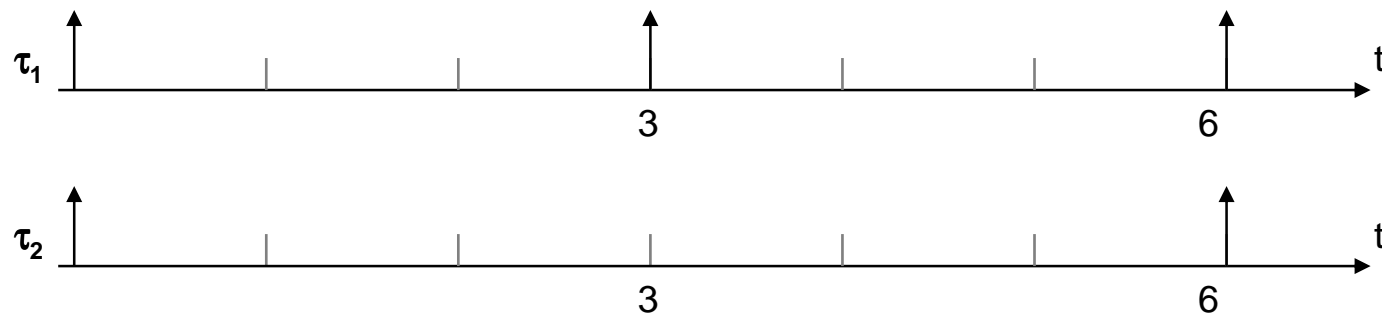


- Processor utilisation  $U =$
- Period of study  $H =$
- Response time for each task



## Exercise 2 on RM

$T$	$r_i$	$C_i$	$D_i$	$T_i$	$Prio_i$
$\tau_1$	0	1	2	3	
$\tau_2$	0	3	5	6	



## Results for RM

### Optimality:

RM is optimal for the family of independent periodic tasks with  $r=0$ ,  $P=D$  and static priority:

if any *static priority preemptive* scheduling algorithm can meet all the deadlines, then the *rate monotonic* algorithm can too

### Critical instant theorem

the critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher-priority tasks. Then, it is sufficient to study RM schedulability the case where all tasks start at 0.

## Feasibility tests for RM

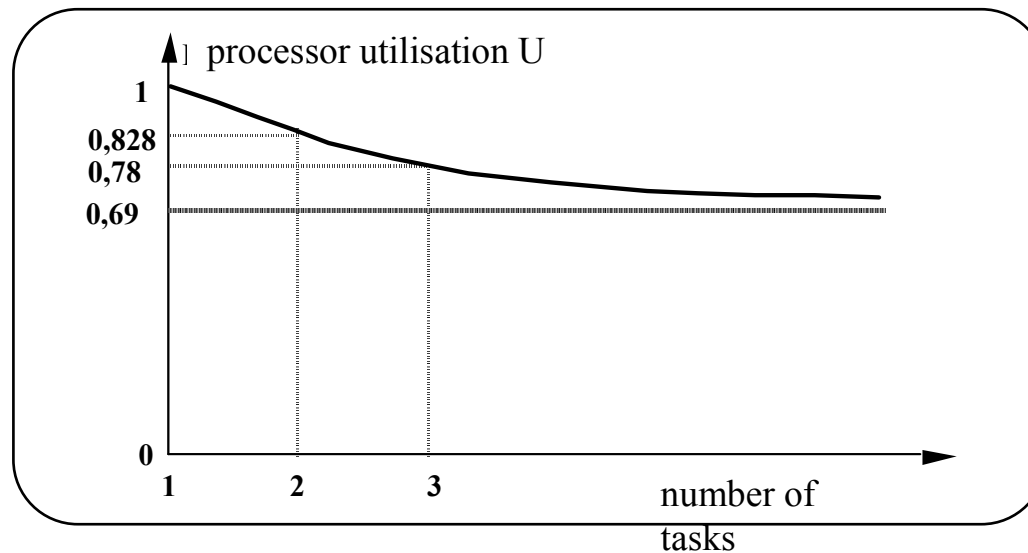
1. **Simulation (NSC on the feasibility interval)**
2. **Processor utilisation (SC)**
3. **Computation of the worst case response time (NSC)**

# Schedulability for RM: processor utilisation

Sufficient condition of schedulability:

$$U \leq n (2^{1/n} - 1)$$

$$n (2^{1/n} - 1) \longrightarrow \ln 2 \sim 0.69$$



Reference: Liu&Layland 73

## Critical zone theorem for RM

The theorem of the critical zone is less restrictive than the processor utilisation condition.  
*If all the tasks respect their first deadline then they will respect all their deadlines.*

If for all  $i$   $r_i = 0$ , this condition is necessary and sufficient

Otherwise, it is sufficient.

Let  $\tau_1, \dots, \tau_n$  be  $n$  tasks  $(T_i, 0, C_i, D_i)$  such that  $T_i \leq T_{i+k}$ , the tasks are schedulable if and only if

$$\forall i, 1 \leq i \leq n \min_{0 \leq t \leq D_i} \sum_{j=1}^i \frac{C_j}{t} \lceil t/T_j \rceil \leq 1$$

Example: Is the task set  $\{(10,0,3,10), (10,0,4,10), (20,0,4,20)\}$  schedulable?

### Reference:

-J P Lehoczky, L. Sha and Y. Ding, The rate monotonic scheduling algorithm: exact characterization and average case behavior (1989), In Proc. of the 10th IEEE Real-Time Systems Symposium

# Computation of the response time for RM

- Let  $hp(i)$  the set of indices of tasks with a higher (or equal) priority than  $\tau_i$ .
- The response time is given by the interference with tasks with higher (or equal) priority. The worst response time is equal to the least fixed point of the sequence:

$$R_i^0 = C_i$$
$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j$$

The sequence converges if and only if  $U \leq 1$

The system is schedulable iff the sequence converges and  $\forall i \in [1, n], R_i^* \leq D_i$

## References:

- M. Joseph and P. Pandya, Finding response times in a real-time system, The Computer Journal 29 (5) (1986), pp. 390–395.
- J P Lehoczky, Fixed priority scheduling of periodic task sets with arbitrary deadlines (1990), In Proc. of the 11th IEEE Real-Time Systems Symposium

## Computation of the response time for RM

Compute the response times for this set of tasks

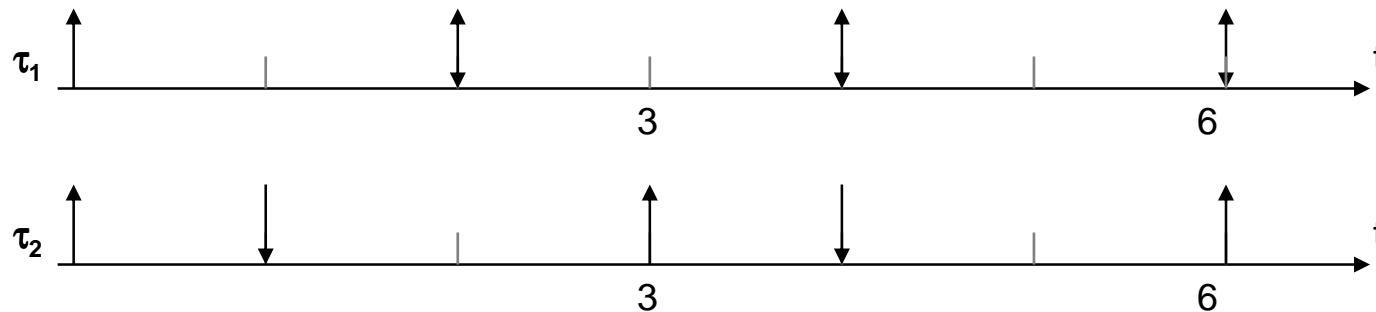
$T$	$r_i$	$C_i$	$T_i$
$\tau_1$	0	2	7
$\tau_2$	0	1	4
$\tau_3$	0	6	14



# Deadline monotonic (Leung & Whitehead, 1980)

The priority depends on the relative deadline: the smaller is the deadline, the higher is the priority.

$T$	$r_i$	$C_i$	$D_i$	$T_i$	$Prio_i$
$\tau_1$	0	1	2	2	
$\tau_2$	0	1	1	3	



## Results for DM

### Optimality:

**RM is optimal for the family of independent periodic tasks with  $r=0$ ,  $T \geq D$  and static priority**

### Feasibility tests for DM:

- **Simulation (NSC)**
- **Computation of the worst response time (idem that for RM, NSC)**
- **Processor utilisation (SC)**

$$\sum_{i=1}^n C_i / D_i \leq n(2^{1/n} - 1)$$

## Critical zone for DM

Let  $\tau_1, \dots, \tau_n$  be  $n$  tasks  $(T_i, 0, C_i, D_i)$  such that  $T_i \leq T_{i+k}$ , the tasks are schedulable if and only if

$$\forall i, 1 \leq i \leq n \quad C_i + \sum_{j=1}^{i-1} C_j \left\lceil \frac{D_i}{T_j} \right\rceil \leq D_i$$

# Optimality of DM

**DM and RM are equivalent for the tasks where  $T=D$**

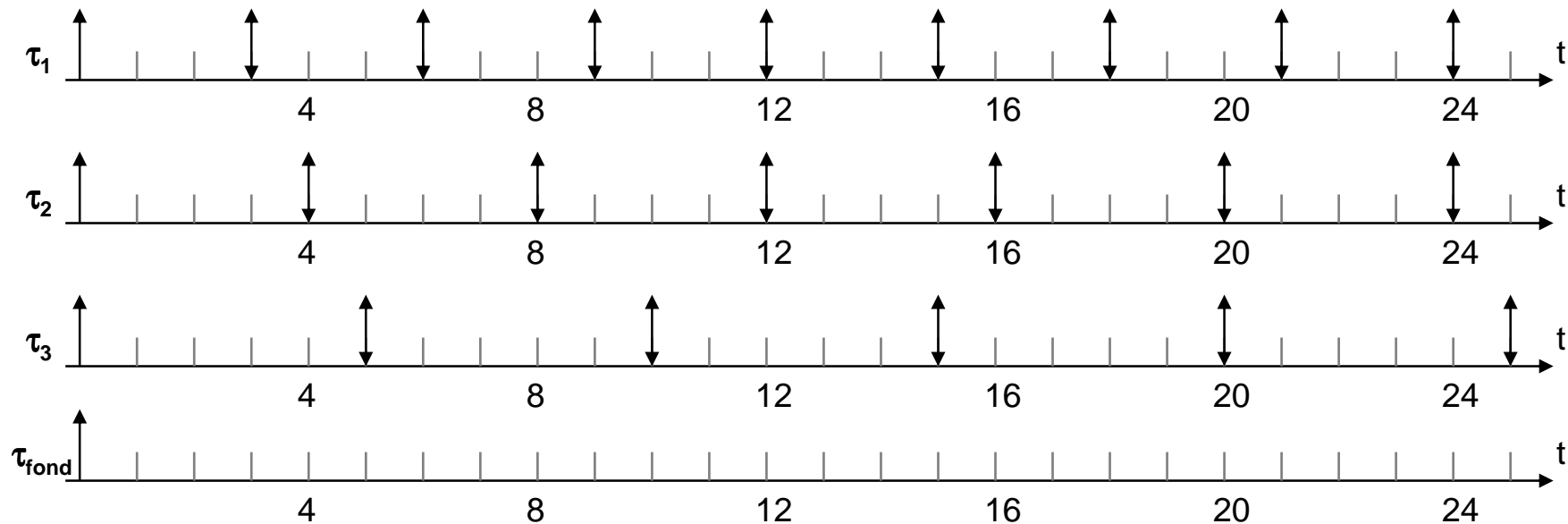
## **Optimality of DM:**

- In the class of static priority preemptive scheduling, for independent periodic tasks with  $T \leq D$ , the policy DM is optimal.
- In the class of static priority non-preemptive scheduling, for independent periodic tasks with  $T \leq D$ , the policy DM is not optimal except if

$$\forall i, D_i \leq P_i \wedge \forall (i, j), C_i \leq C_j \Rightarrow D_i \leq D_j$$

# RM and DM example

$T$	$r_i$	$C_i$	$D_i$	$T_i$	$Prio_{RM}$	$Prio_{DM}$
$\tau_1$	0	1	3	3		
$\tau_2$	0	1	4	4		
$\tau_3$	0	2	5	5		



# Audsley priority assignment (Audsley 1991)

## Context:

- For independent periodic tasks with  $T \leq D$  and  $r$  non necessarily null.

## Algorithm:

- The response time only depends (for static priority) on the higher priority tasks and is independent from the combination of lower priorities. The idea is then to search among the set of tasks a candidate to have the lowest priority. The task accepts if the feasibility test is fine. Re do the same for the rest of tasks.

## Feasibility test:

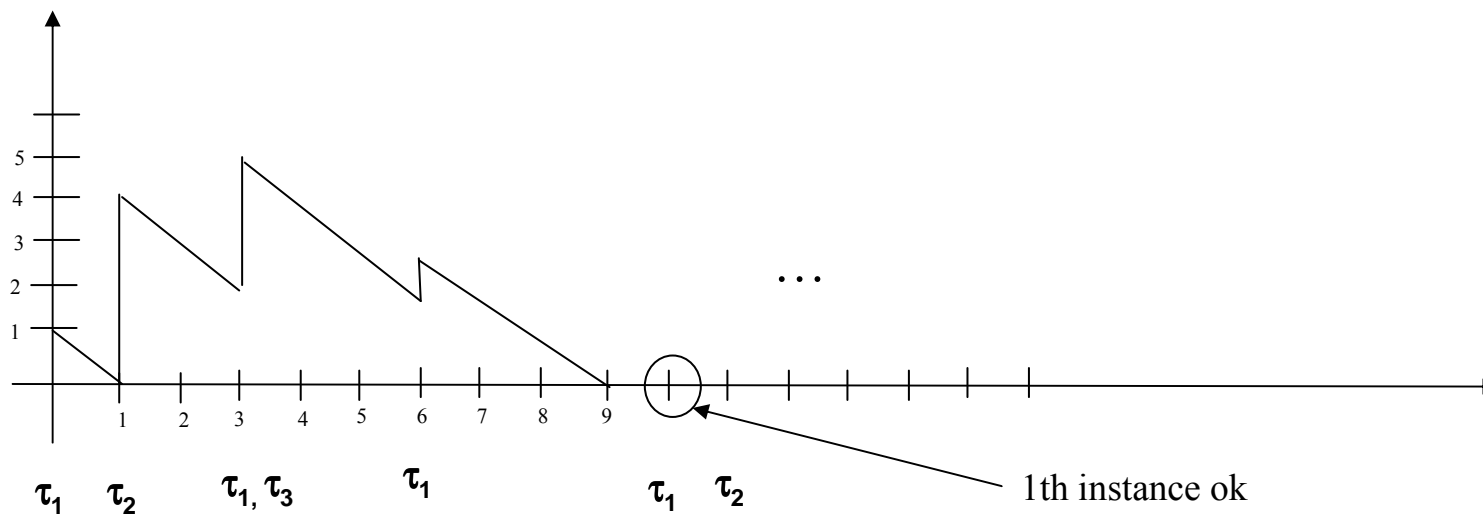
- The schedulability is ensured by construction

## Optimality

# Audsley priority assignment

$T$	$r_i$	$C_i$	$D_i$	$T_i$	Prio
$\tau_1$	0	1	3	3	
$\tau_2$	1	4	9	9	
$\tau_3$	3	2	5	7	

$\tau_2$  with the lowest priority?



# Earliest Deadline First (EDF)

The priority evolves during the execution (dynamic priority). For an instance  $k$  of a task  $\tau_i$ , the priority depends on the next absolute deadline  $d_i^k$ . At an instant  $t$ , the priority can be computed using the critical dynamic delay

$$D(t) = d_i^k - t = r_i^k + D_i - t$$

The priority increases when the critical dynamic delay decreases.

For a set of  $n$  tasks with  $T=D$ , a necessary and sufficient condition of schedulability is:  $U \leq 1$

In general, the condition is sufficient.

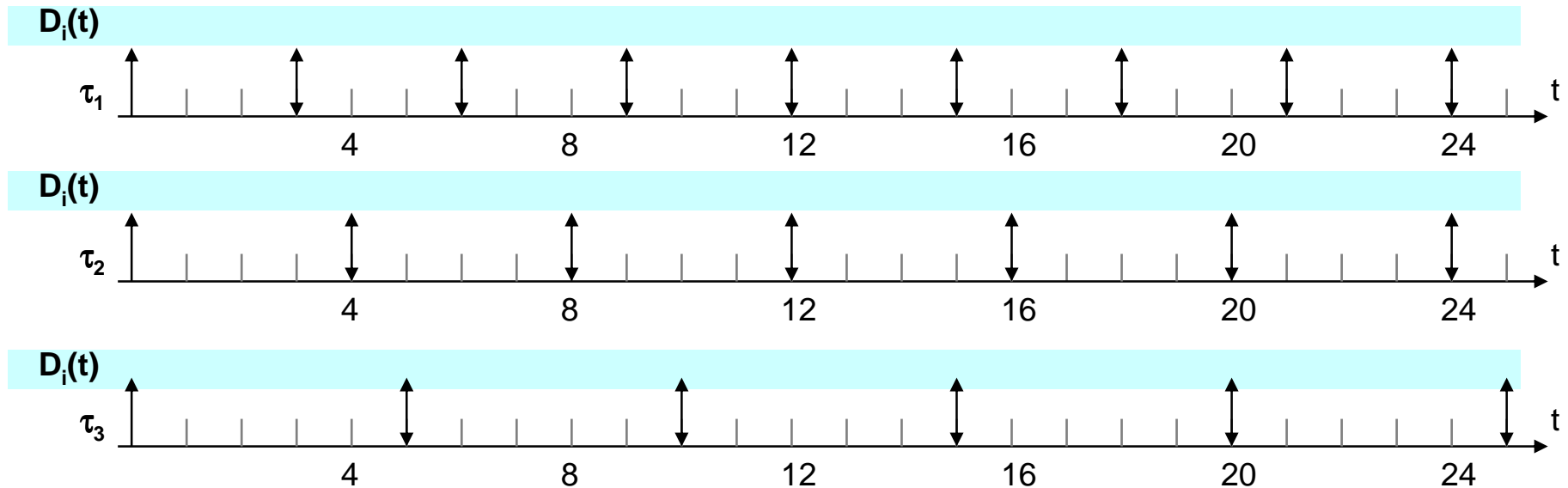
## Optimality:

- In the class of dynamic priority preemptive scheduling, for independent periodic tasks, the policy EDF is optimal.
- For non preemptive scheduling, EDF is not anymore optimal



# EDF: example

$T$	$r_i$	$C_i$	$D_i$	$T_i$
$\tau_1$	0	1	3	3
$\tau_2$	0	1	4	4
$\tau_3$	0	2	5	5



## Least laxity first (LLF)

The priority evolves during the execution (dynamic priority) and depends on the dynamic laxity

$$L(t) = d_i^k - t - C_i(t) = D_i(t) - C_i(t)$$

The priority increases when the dynamic laxity decreases.

For a set of  $n$  tasks with  $T=D$ , a necessary and sufficient condition of schedulability is:  $U \leq 1$

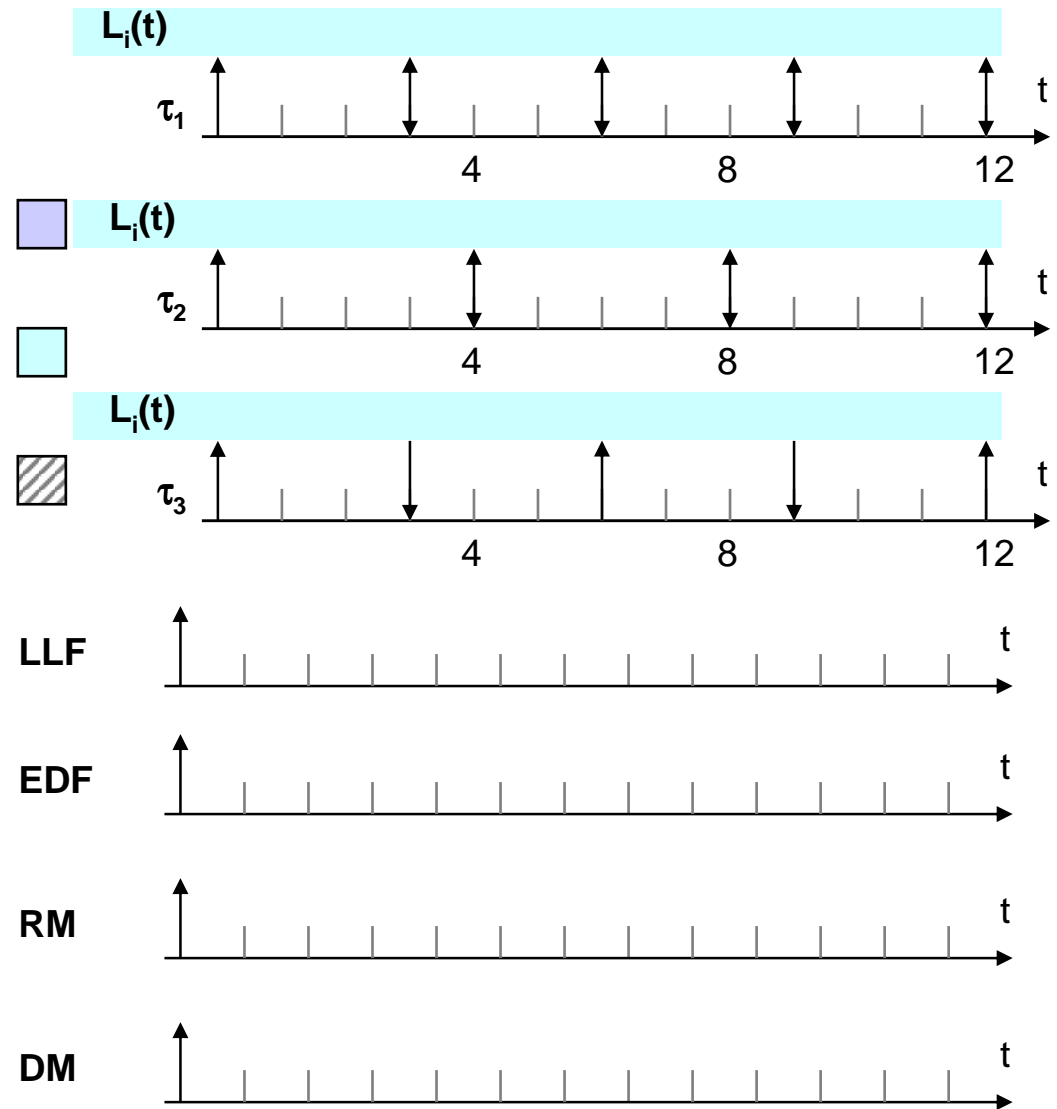
In general, the condition is sufficient.

**Optimality:**

- In the class of dynamic priority preemptive scheduling, for independent periodic tasks, the policy LLF is optimal.

# LLF: example

$T$	$r_i$	$C_i$	$D_i$	$T_i$
$\tau_1$	0	1	3	3
$\tau_2$	0	1	4	4
$\tau_3$	0	2	3	6



## Outline - III.2 – Uniprocessor scheduling

1. Recalls
2. Real-time scheduling
3. Priority-based Scheduling
4. **Scheduling with shared resources**

# Scheduling with shared resources

## Tasks with resource constraints

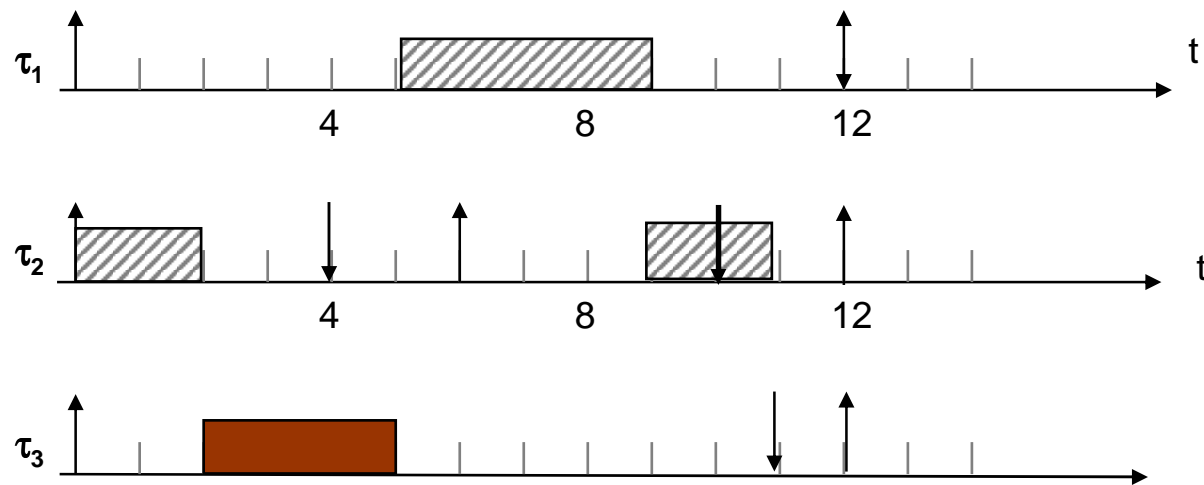
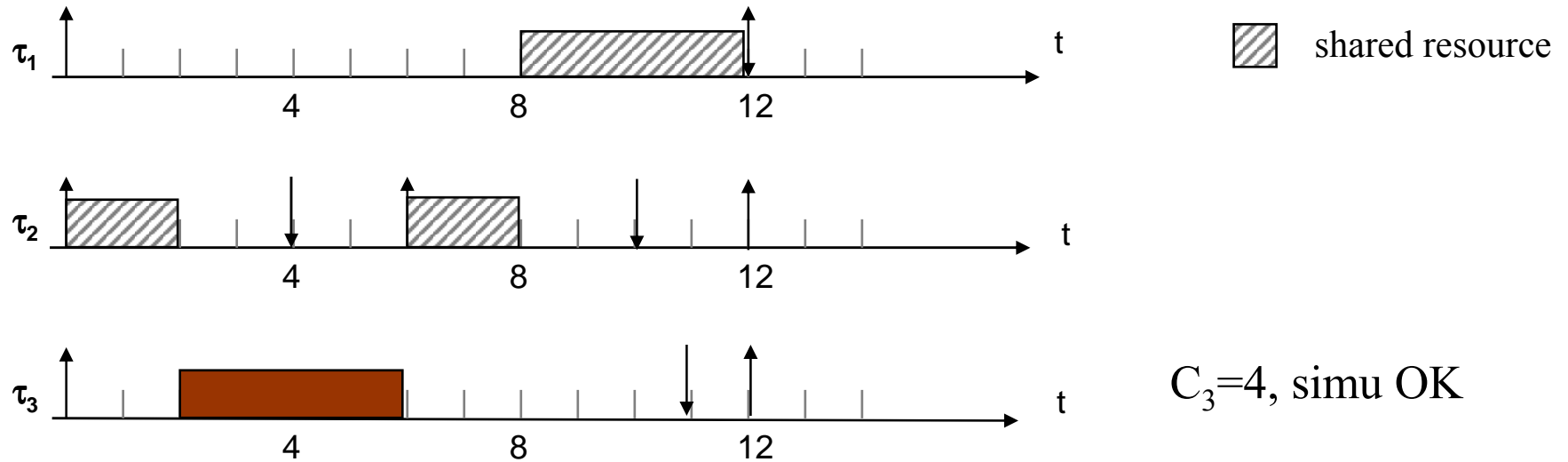
- Critical sections due to exclusive use of shared resources.
- Semaphores.

## Problems encountered:

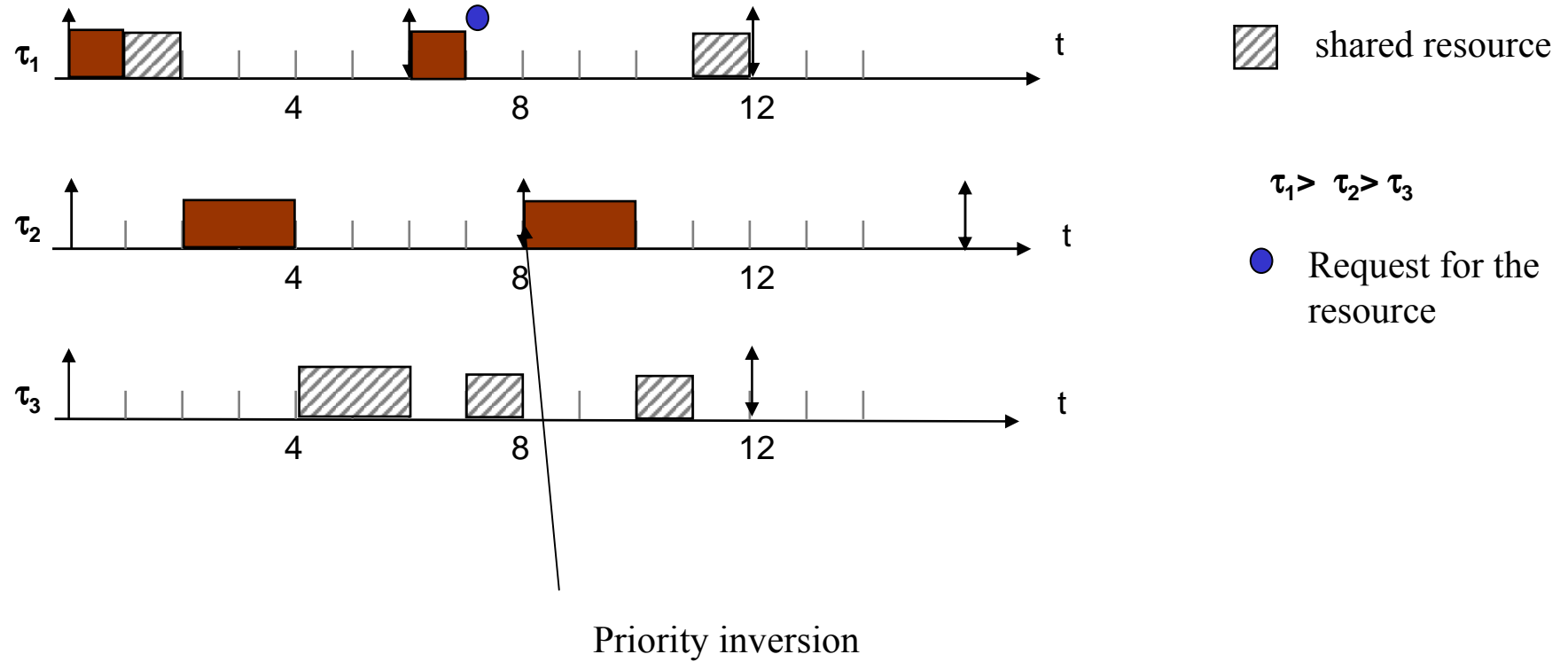
- Blocking
- Priority inversion phenomenon
- Scheduling anomaly

**Consequence: simulation of the worst case is not anymore a NSC**

# Anomaly of scheduling



# Priority inversion



# A solution: PIP Priority inheritance protocol

## Context:

- Static priority algorithms

## Principle:

- When a task is blocked when accessing a resource, the task which is locking the resource inherits during the critical section of the priority of the requesting task.

## Result:

- Suppression of priority inversion
- Other tasks may be blocked

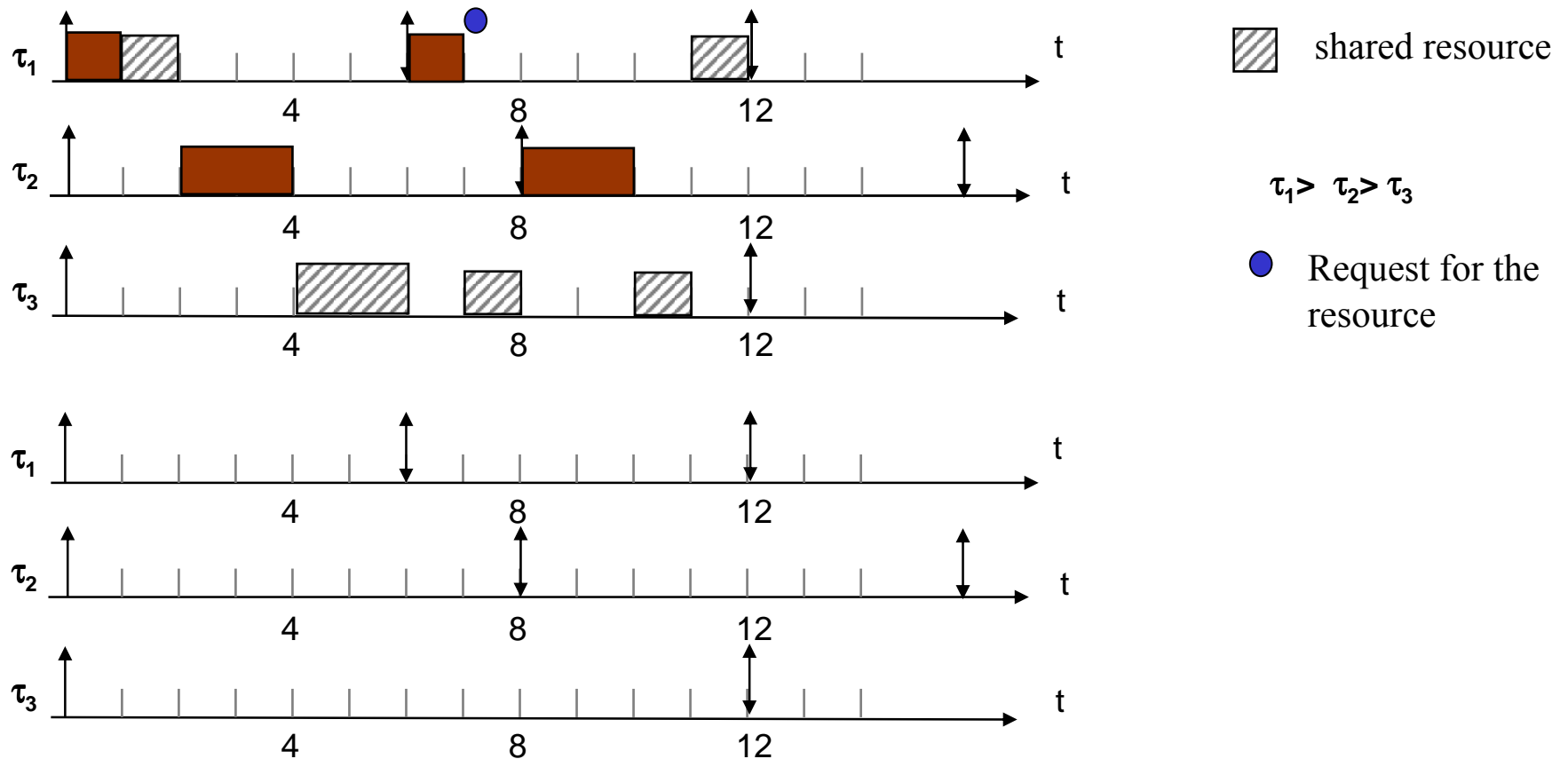
## Reference:

- **L. Sha, R. Rajkumar and J. P. Lehoczky, Priority inheritance protocols : an approach to real-time synchronisation, IEEE Transactions on Computer, vol. 39 (1990), pp. 1175–1185.**



# PIP example

Include the PIP protocol in this example



## References

1. **Francis Cottet: course on real-time systems**
2. **Pierre-Emmanuel Hladik: course on real-time systems**
3. **Emmanuel Grolleau: course on real-time systems**
4. **Book Francis Cottet, Joëlle Delacroix, Claude Kaiser and Zoubir Mammeri: Ordonnancement temps réel, Hermes (Scheduling in real-time systems, John Wiley & Sons)**
5. **Book Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza and Michael Gonzalez Harbour: A practitioner's handbook for real-time analysis, Kluwer Academic Publishers.**

# Outline - Part III - Scheduling

1. First definitions
2. Uniprocessor real-time scheduling
3. **Multiprocessors real-time scheduling**

## Outline – III.3 – multiprocessor scheduling

1. Generalities
2. Partitioned scheduling
3. Global scheduling

# General principles

## We consider:

- a set of  $n$  tasks  $S = \{\tau_1, \tau_2, \dots, \tau_n\}$
- a parallel architecture composed of  $m$  processors  $P = \{p_1, p_2, \dots, p_m\}$   
(applies for multicore)

## Objective:

- schedule the tasks on the platform.
  - Resolution of two problems: allocation (on which processor) and priority

## Constraints:

- A processor executes at most a task at a time
- A task executes at most on one processor at an instant

# Taxonomy of the multiprocessor platform

- **Identical:** all the processors are assumed to be identical, with the same computing capacity.
- **Uniform:** each processor is characterised by its computing capacity and the model assumed that: for a processor of capacity  $s$ , a time duration  $t$ , the processor executes  $s*t$  units of work.
- **Specialised:** we define an execution rate for every couple  $r_{i,j} = (J_i, p_j)$  meaning that:  $J_i$  executes  $r_{i,j} * t$  units of work when it is hosted by  $p_j$  during  $t$  units of time.

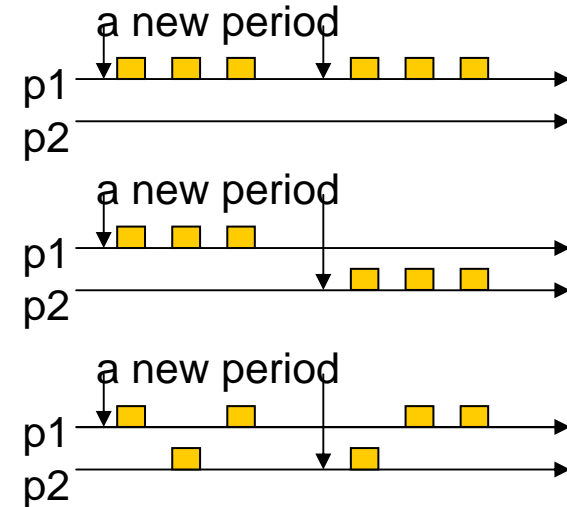
**Identical  $\subset$  Uniform  $\subset$  Specialised**

**In the following, we will only consider identical platforms.**

# Taxonomy of the multiprocessor scheduling (I)

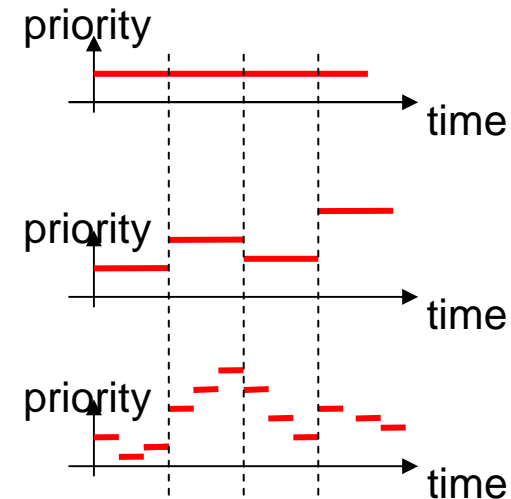
## Task execution on the processors:

- **No migration:** each task is allocated on a unique processor and never changes
- **Task-level migration:** the jobs may execute on different processors, but a job is allocated on a unique processor and never changes
- **Job-level migration (or full migration):** a job can migrate



## Priorities:

- **Fixed task priority:** each task has a fixed priority forever
- **Fixed job priority:** the jobs may have several priorities, but each job has a fixed priority (e.g. EDF)
- **Dynamic priority:** the priority of a job may evolve (e.g. LLF)



## Taxonomy of the multiprocessor scheduling (II)

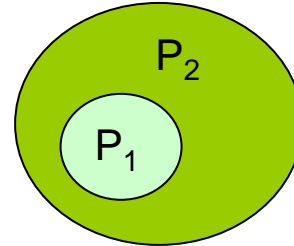
<b>3: full migration</b>	<b>(1,3)-scheduling</b>	<b>(2,3)</b>	<b>(3,3)</b>
<b>2: task level migration</b>	<b>(1,2)</b>	<b>(2,2)</b>	<b>(3,2)</b>
<b>1: no migration</b>	<b>(1,1)</b>	<b>(2,1)</b>	<b>(3,1)</b>
	<b>1: fixed task priority</b>	<b>2: fixed job priority</b>	<b>3: fully dynamic</b>



# Comparison of scheduling policies

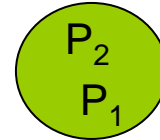
- Dominance

$$P_1 \subset P_2$$



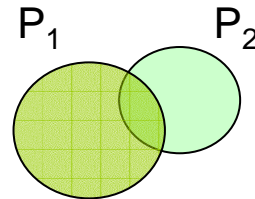
- Equivalence

$$P_1 = P_2$$



- Incomparable

$$P_1 \otimes P_2$$



[Park 2007]

# Current state of the art

## Solid theory of single processor systems

- Optimal schedulers, tight schedulability tests, shared resource protocols, bandwidth reservation schemes, hierarchical schedulers, OS, etc.

## Much less results for multiprocessors

- “Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.”
  - C.L. Liu, “Scheduling algorithms for multiprocessors in a hard real-time environment”. JPL Space Programs Summary, vol. 37-60, pp. 28-31, 1969.
- Many NP-hard problems, few optimal results, heuristic approaches, simplified task models, only sufficient schedulability tests, etc.
- On going research

## Optimality (non) results

“In 1988, Hong and Leung proved that there is no optimal online scheduling algorithm for the case of an arbitrary collection of jobs that have more than one distinct deadline, and are scheduled on more than one processor. Hong and Leung showed that such an algorithm would require knowledge of future arrivals and execution times to avoid making decisions that lead to deadline misses; hence optimality in this case is impossible without clairvoyance. In 1989, this result was extended by Dertouzos and Mok who showed that knowledge of arrival times is necessary for optimality, even if execution times are known.”

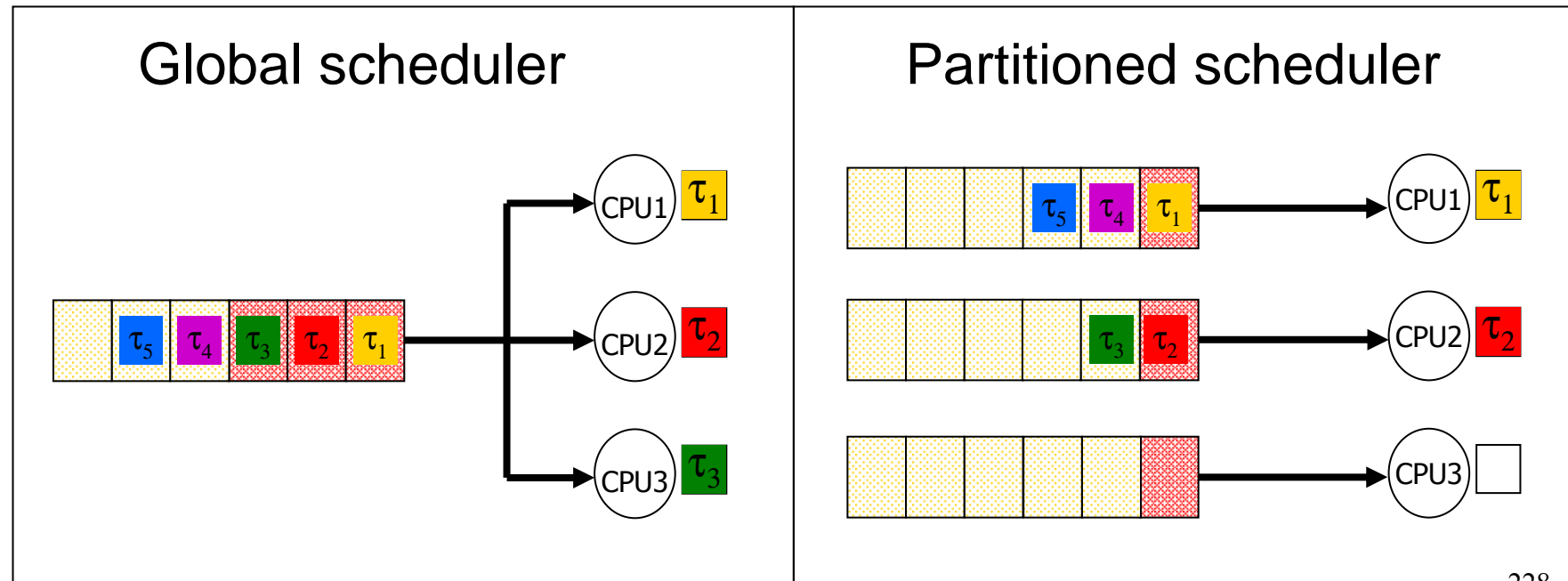
### [Davis&Burns09]

**NB: A scheduling algorithm is said to be clairvoyant if it makes use of information about future events, such as the precise arrival times of sporadic tasks, or actual execution times, which are not generally known until they happen.**

# Families of algorithms

## Two main approaches:

1. **Partitioning:** partition the set of tasks into  $m$  subsets, each set is assigned to a unique processor. Tasks are not allowed to migrate, thus the multiprocessor scheduling is transformed into many uniprocessor scheduling problems.
2. **Global strategy:** store the tasks ready in one queue which is shared among all the processors. At every moment, the  $m$  highest priority tasks of the queue are selected for the  $m$  processors.



## **Outline – III.3 – multiprocessor scheduling**

- 1. Generalities**
- 2. Partitioned scheduling**
- 3. Global scheduling**

# Partitioning

- **Partitioning problem is a *bin packing problem***
  - Inputs:
    - $N$  objects of size  $S(i)$  for  $i=1, \dots, N$
    - infinite number of bins of size  $C$
  - Question: find an optimal packing of the objects in the bins
  - NP-hard problem
- **Correspondence**
  - objects = tasks
  - The size of the boxes depends on the policy

## Heuristic for partitioning

1. Tasks are ordered according to a given parameter

## Ordering policies

1. **Increasing in Execution time (IE):** sorts the tasks in ascending order of their execution times
  2. **Decreasing in Execution time (DE):** sorts the tasks in descending order of their execution times
  3. **Increasing in Period (IP):** sorts the tasks in ascending order of their periods
  4. **Decreasing in Period (DP):** sorts the tasks in descending order of their periods
  5. **Increasing in Utilization factor (IU):** sorts the tasks in ascending order of their utilization factors
  6. **Decreasing in Utilization factor (DU):** sorts the tasks in descending order of their utilization factors
- Example:**  $\{\tau_1=(T_1=4, C_1=1, D_1=4, r_1=0), \tau_2=(8,2,8,0), \tau_3=(20,10,20,0)\}$ . Sort the task set for all the policies.



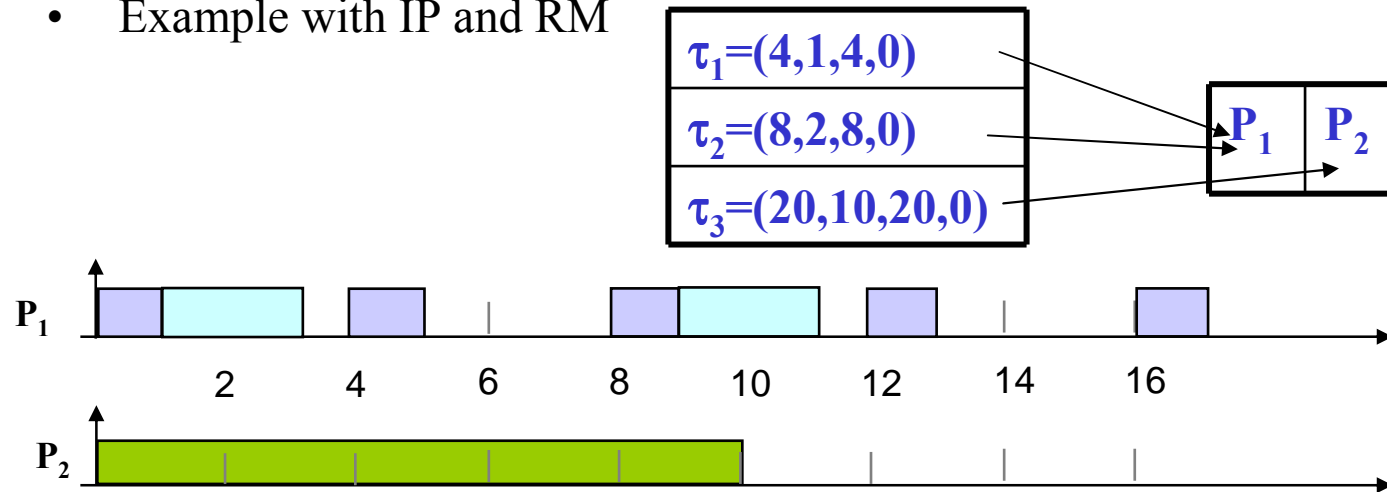
## Heuristic for partitioning

1. Tasks are ordered according to a given parameter
2. Sequential assignment
3. The current task is assigned to the *first* processor according to a policy (e.g. : Best Fit, First Fit, Next Fit, ...)

# Allocation algorithm

1. **First fit:** place the item in the first bin that can accommodate it

- Example with IP and RM



2. **Best fit:** place the item in a bin that can accommodate it and with the smallest available size
3. **Next fit:** place the item in the next bin that can accommodate it (it starts from the previous bin which have been used)
4. **Worst fit:** place the item in a bin that can accommodate it and with the largest available size

**Exercise:** apply for all the algorithms on the example

## Heuristic for partitioning

1. Tasks are ordered according to a given parameter
2. Sequential assignment
3. The current task is assigned to the *first* processor according to a policy (e.g. : Best Fit, First Fit, Next Fit, ...)
4. An assignment is valid if the maximal size is not exceeded
5. If no processor is available, add a new processor

## Exercises

$T$	$r_i$	$C_i$	$\tau_i$
$\tau_1$	0	3	7
$\tau_2$	0	2	4
$\tau_3$	0	6	14
$\tau_4$	0	7	20

Number of processors = 2

Apply the following heuristics:

1. FFDU-EDF
2. WFDE-LLF

# Outline – III.3 – multiprocessor scheduling

## 1. Generalities

## 2. Partitioned scheduling

## 3. Global scheduling

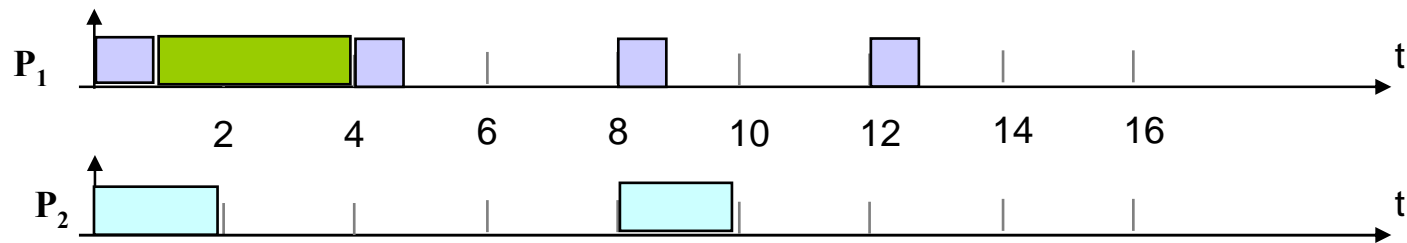
1. Policies
2. Schedulability tests
3. PFair
4. LLREF

# Multiprocessor scheduling policies

- **Any monoprocessor scheduling policy can be extended on a multiprocessor platform.**
  - Generally addition of the letter g in the name for global. (gRM, gEDF, gLLF...)
  - Optimality is not preserved. gLLF and gEDF are incomparable. None of them is optimal.
- **New scheduling policies:**
  - PFair
  - LLREF

## Example of global DM scheduling

$T$	$r_i$	$C_i$	$D_i$	$T_i$
$\tau_1$	0	1	4	4
$\tau_2$	0	2	8	8
$\tau_3$	0	3	16	16



## gEDF and gLLF

$T$	$C_i$	$T_i$
$\tau_1$	2	3
$\tau_2$	2	3
$\tau_3$	2	3

This task set is gLLF schedulable but not gEDF  
Prove it.

$T$	$r_i$	$C_i$	$D_i$	$T_i$
$\tau_1$	0	1	5	10
$\tau_2$	0	1	5	10
$\tau_3$	0	3	6	10
$\tau_4$	1	3	9	10
$\tau_5$	4	5	9	10
$\tau_6$	4	5	9	10

This task set is gEDF schedulable but not gLLF  
Prove it.

[Kal00]



# Comparison between 9 classes

3: full migration	(1,3)-scheduling	(2,3)	(3,3)
2: task level migration	(1,2)	(2,2)	(3,2)
1: no migration	(1,1)	(2,1)	(3,1)
	1: fixed task priority	2: fixed job priority	3: fully dynamic

- (3,3) dominates all other classes
- (1,\*) are incomparable (cf below, [Leung82])
- (\*,1) are incomparable with (\*,2)

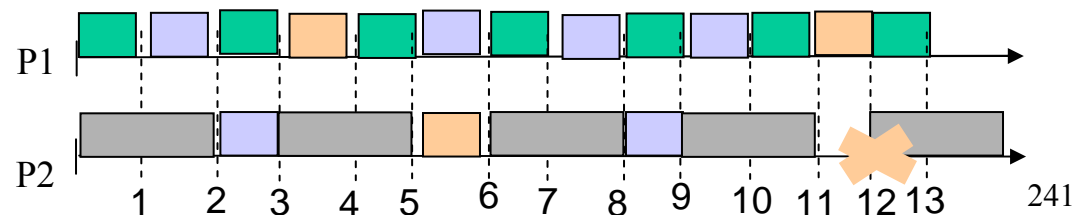
[Carpenter et al. 2004]

$T$	$r_i$	$C_i$	$D_i$	$T_i$
$\tau_1$	0	2	3	3
$\tau_2$	0	4	6	6
$\tau_3$	0	6	12	12

This system is schedulable for a global scheduling on 2 processors with priorities  $\tau_1 > \tau_2 > \tau_3$  and there exists no way to partition this set of tasks.

$T$	$r_i$	$C_i$	$D_i$	$T_i$	Prio
$\tau_1$	0	1	2	2	1
$\tau_2$	0	2	4	4	3
$\tau_3$	0	2	3	3	2
$\tau_4$	0	2	6	6	4

This system is schedulable for the partition  $\tau_1$  and  $\tau_2$  on a processor, and  $\tau_3$  and  $\tau_4$  on an other; and the policy RM. But there is no solution for a global static priority policy.



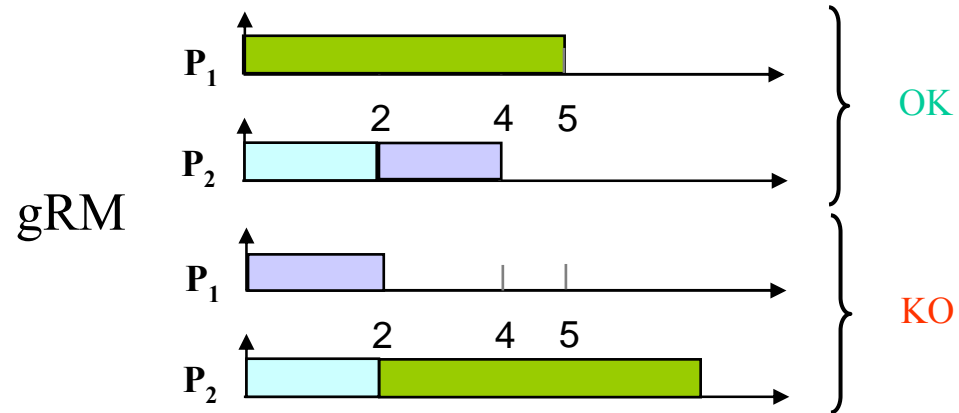
# Outline – III.3 – multiprocessor scheduling

1. Generalities
2. Partitioned scheduling
3. **Global scheduling**
  1. Policies
  2. Schedulability tests
  3. PFair
  4. LLREF

# Not applicable

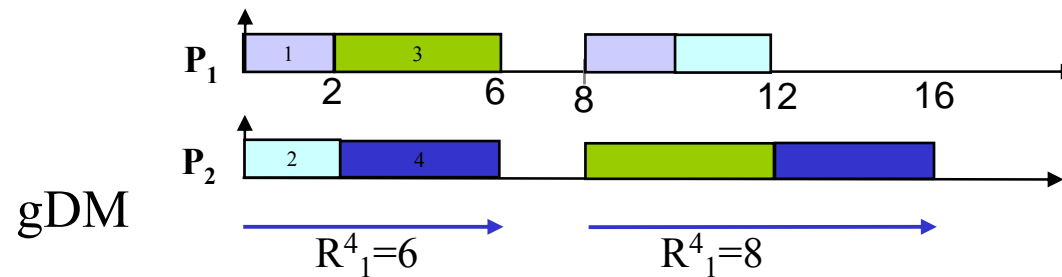
- Simulation is generally not sufficient

$T$	$r_i$	$C_i$	$T_i$
$\tau_1$	0	5	5
$\tau_2$	0	2	5
$\tau_3$	0	2	5



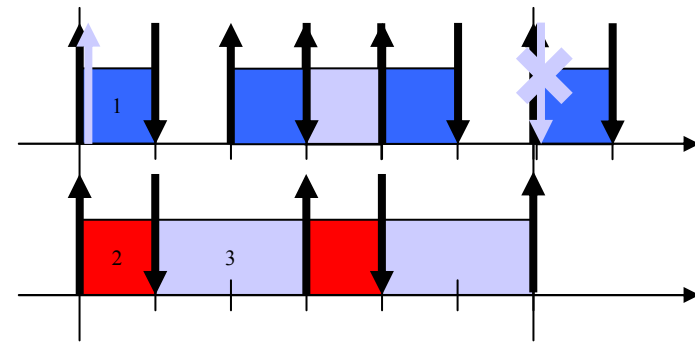
- Critical instant: synchronous release is the worst-case scenario for periodic tasksets on uniprocessor platform. Not true for multiprocessor

$T$	$D_i$	$C_i$	$T_i$
$\tau_1$	2	2	8
$\tau_2$	2	2	10
$\tau_3$	6	4	8
$\tau_4$	7	4	8



## Contre-intuitive observations (1)

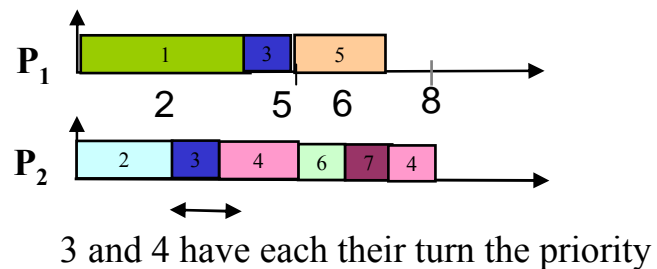
$T$	$r_i$	$C_i$	$D_i$	$T_i$
$\tau_1$	0	1	1	<b>3</b>
$\tau_2$	0	1	1	3
$\tau_3$	0	5	6	6



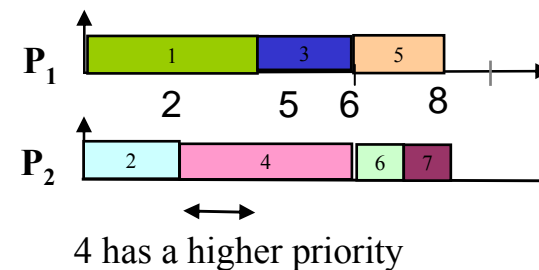
**The processor demand decreases but the task set become unschedulable**

## Contre-intuitive observations (2)

$T$	$r_i$	$C_i$	$D_i$	$T_i$
$\tau_1$	0	4	4	100
$\tau_2$	0	2	2	100
$\tau_3$	0	2	6	100
$\tau_4$	0	4	8	100
$\tau_5$	5	2	2	100
$\tau_6$	5	1	1	100
$\tau_7$	6	1	2	100



$T$	$r_i$	$C_i$	$D_i$	$T_i$
$\tau_1$	0	4	4	100
$\tau_2$	0	2	2	100
$\tau_3$	0	2	6	100
$\tau_4$	0	4	8	100
$\tau_5$	6	2	2	100
$\tau_6$	6	1	1	100
$\tau_7$	6	1	2	100



The policy must be clairvoyant

## Schedulability test

- **The feasibility interval is not known**
  - Unique result:  $[0, \text{lcm}(T)]$  for fixed priority policy and synchronous task set
- **Existing necessary and sufficient schedulability tests all have exponential time complexity**
- **Existing sufficient tests are pessimistic**

# Utilisation bounds

## Theorem

1. For (3,3), synchronous task set with implicit deadline,  $U_{\max} = m$
2. For (x,y) with  $x, y \neq 3$ ,  $U_{\max} \leq (m+1)/2$

## Proof:

1. Cf PFair or LLREF
2. Consider a task set

$$t_1 = (1+\epsilon, 2)$$

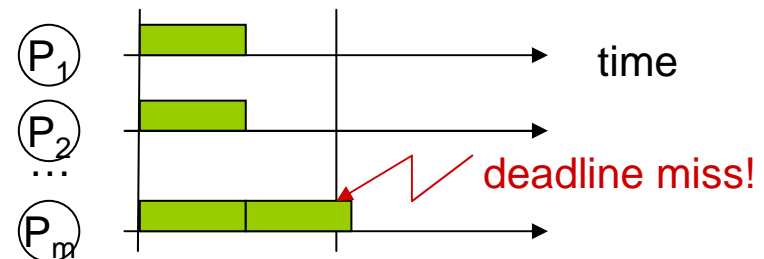
$$t_2 = (1+\epsilon, 2)$$

...

$$t_{m+1} = (1+\epsilon, 2)$$

$$U = \sum_{i=1}^m (1+\epsilon)/2 = (m+1)(1+\epsilon)/2 \rightarrow_{\epsilon \rightarrow 0} (m+1)/2$$

1. If  $y < 3$ , at least two tasks are assigned to the same processor
2. If  $x < 3$ , the job with the lowest priority is executed after  $(1+\epsilon)$



[Park 2007]

# Outline – III.3 – multiprocessor scheduling

1. Generalities
2. Partitioned scheduling
3. **Global scheduling**
  1. Policies
  2. Schedulability tests
  3. PFair
  4. LLREF

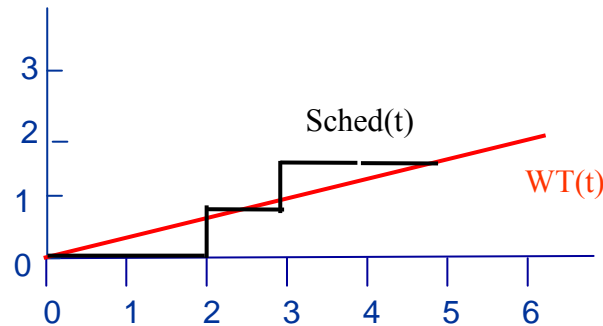


# PFair (Proportionate Fair) [Baruah et al 1996]

## Basic Ideas:

- PFair tries to track the ideal scheduling
- For task  $(C,T)$ , the ideal scheduling is  $WT(t)=t \cdot C/T$

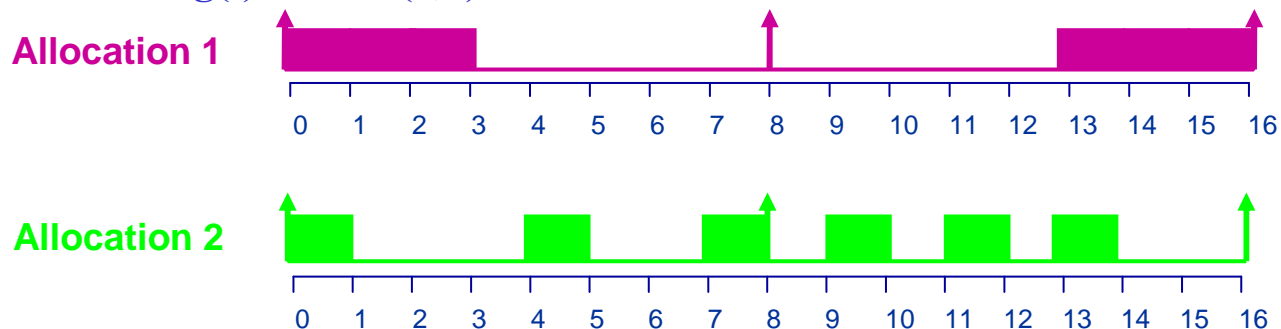
Example:  $\tau = (2,5)$



The tracking is formalised using the function

- $lag(t) = WT(t) - sched(t)$
- A schedule is PFair if  $lag(t) \in [-1,1]$

Exercise: draw  $lag(t)$  for  $\tau = (3,8)$



[Park 2007]

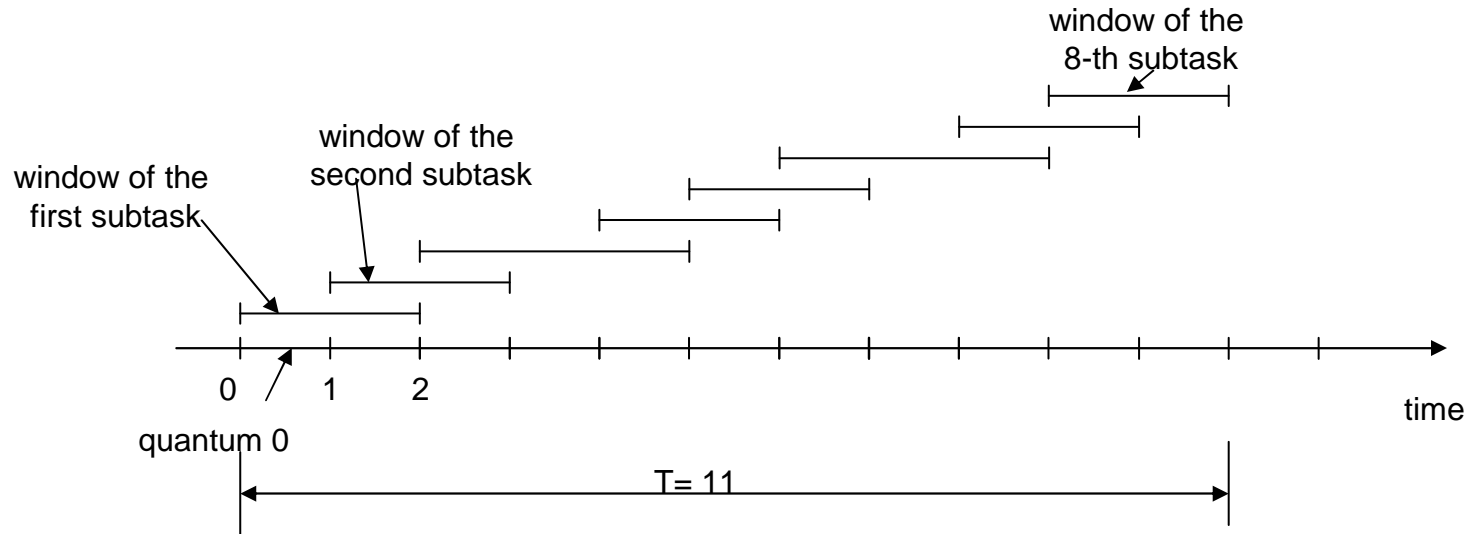
# PFair scheduling

## Ideas:

- Processor time is allocated in multiples of some basic *quantum*.
- Break tasks into *subtasks of length 1*.
- Assign deadlines and release times to subtasks.
  - deadline and release time of subtask  $i$  are

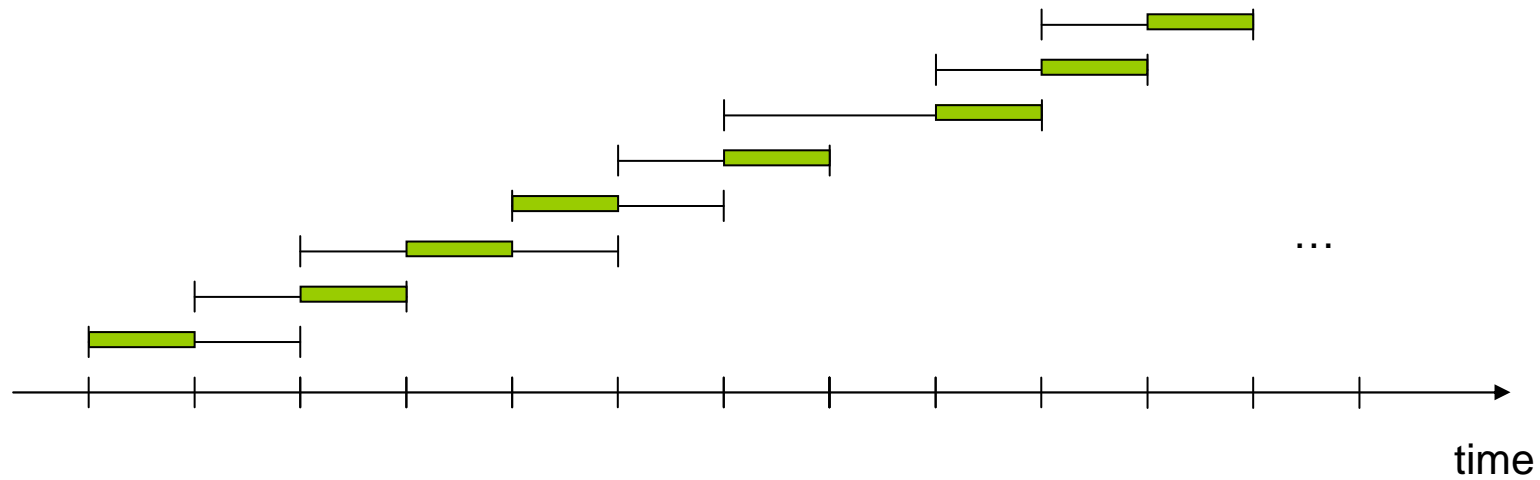
$$d_i = \left\lceil \frac{i}{U} \right\rceil \quad r_i = \left\lfloor \frac{i-1}{U} \right\rfloor$$

**Example:  $\tau = (8,11)$ . We have  $d_1 = \lceil 1/(8/11) \rceil = 2$  and  $r_1 = 0$ .**



# PFair scheduling

Example of PFair scheduling for  $\tau = (8,11)$ .



Exercise: Find a PFair scheduling for (4,9)

# PFair scheduling algorithm

## A PFair scheduling algorithm

- is an algorithm that generates a PFair schedule for given tasks
- role: determining priorities of subtasks
- existing algorithms: PF, PD, PD<sup>2</sup>

## PD<sup>2</sup> is the most efficient (O(m log n))

Subtask  $\tau_1^k$  gets higher priority than subtask  $\tau_2^{k'}$  if one of the following rules is satisfied:

1-  $d(\tau_1^k) < d(\tau_2^{k'})$

2-  $d(\tau_1^k) = d(\tau_2^{k'})$  and  $b(\tau_1^k) > b(\tau_2^{k'})$

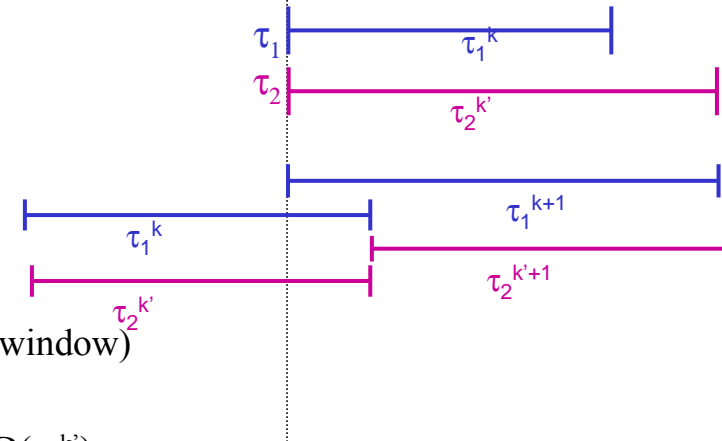
with  $b(\tau_1^k) = d(\tau_1^k) - r(\tau_1^{k+1})$

(number of slots by which  $\tau_i$ 's window overlaps  $\tau_{i+1}$ 's window)

3-  $d(\tau_1^k) = d(\tau_2^{k'})$  and  $b(\tau_1^k) = b(\tau_2^{k'}) = 1$  and  $D(\tau_1^k) \geq D(\tau_2^{k'})$

with

$$D(\tau^k) = \min \left\{ t \mid \left( t = d(\tau^{k+p}) \wedge b(\tau^{k+p}) = 0 \right) \vee \left( t = d(\tau^{k+p}) - 1 \wedge W(\tau^{k+p}) = 3 \right) \right\}$$



## Exercise

$\tau_i$	C	T
$\tau_1$	2	3
$\tau_2$	4	6
$\tau_3$	6	12

Apply PD<sup>2</sup> Pfair for 2 processors

## PFair results

- **Optimal for synchronous periodic task set with implicit deadline**
- **Schedulable if  $U \leq m$  where  $m$  is the number of processors**

<b><math>T</math></b>	<b><math>r_i</math></b>	<b><math>C_i</math></b>	<b><math>T_i</math></b>
$\tau_1$	0	2	3
$\tau_2$	0	2	3
$\tau_3$	0	2	3

**Platform is composed of 2 processors**

**$U=??$**

**Is the task set schedulable with:**

- **A partitionned strategy?**
- **gEDF?**
- **gLLF**
- **PFair?**

# Outline – III.3 – multiprocessor scheduling

1. Generalities
2. Partitioned scheduling
3. **Global scheduling**
  1. Policies
  2. Schedulability tests
  3. PFair
  4. LLREF

# LLREF (Least Local Remaining Execution First)[Cho et al 2006]

Tasks are splitted not at time quantum, but at scheduling events (at release instants)

## Two parameters

1. Local remaining execution time 
$$l_{\tau}(t) = C_{\tau}(t) \times \frac{ND(t) - t}{T_{\tau} - t}$$

2. Local laxity 
$$L_{\tau}(t) = ND(t) - l_{\tau}(t)$$

where

$$ND(t) = \min\{t' \geq t \mid t' = kT_{\tau}\}$$

## Priority

1. Highest priorities for the tasks with  $L_{\tau}=0$
2.  $\tau_1$  has a higher priority than  $\tau_2$  if  $l_{\tau_1} > l_{\tau_2}$

## Quantum

1. If  $\tau_1, \dots, \tau_M$  have the highest priority at time  $t$ , they keep the CPUs until  $t' = \min\{l_{\tau}\}$

**Example: (3,8),(10,20),(2,6) on 2 processors. Compute the lag**



## References

1. **Joël Gossens: course on multiprocessor scheduling**
2. **Robert I. Davis and Alan Burns: A Survey of Hard Real-Time Scheduling Algorithms and Schedulability Analysis Techniques for Multiprocessor Systems, University of York, Department of Computer Science, 2009, Technical report YCS-2009-443.**
3. **Jiyong Park, Real-time Multiprocessor Scheduling Problems and Algorithms, 2007**
4. **Marko Bertogna: Real-Time Scheduling Analysis for Multiprocessor Platforms, PhD Defense, 2008**
5. **Theodore P. Baker and Sanjoy K. Baruah: Schedulability analysis of multiprocessor sporadic task systems, 2007.**
6. **Ted Baker: Introductory Seminar on Research, CIS5935 Fall 2008**
7. **Bala Kalyanasundaram, Kirk R. Pruhs, Eric Torng: Errata - a New algorithm for scheduling periodic real-time tasks, 2000.**