

Real-Time systems

Paolo Burgio

paolo.burgio@unimore.it



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

High Performance
Real Time **Lab**



Breve storia
informatica triste

**"Sul mio
PC
funziona."**



@vitadainformatici

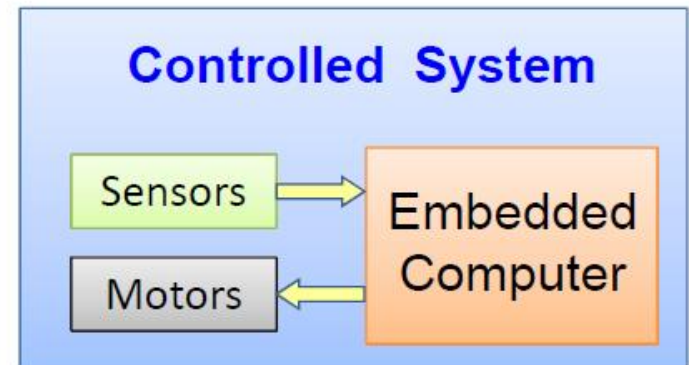


Definition

Real-Time Systems are computing systems that must perform computation within given timing constraints.



They are typically embedded in a larger system to control its functions:



Real-Time Embedded Systems



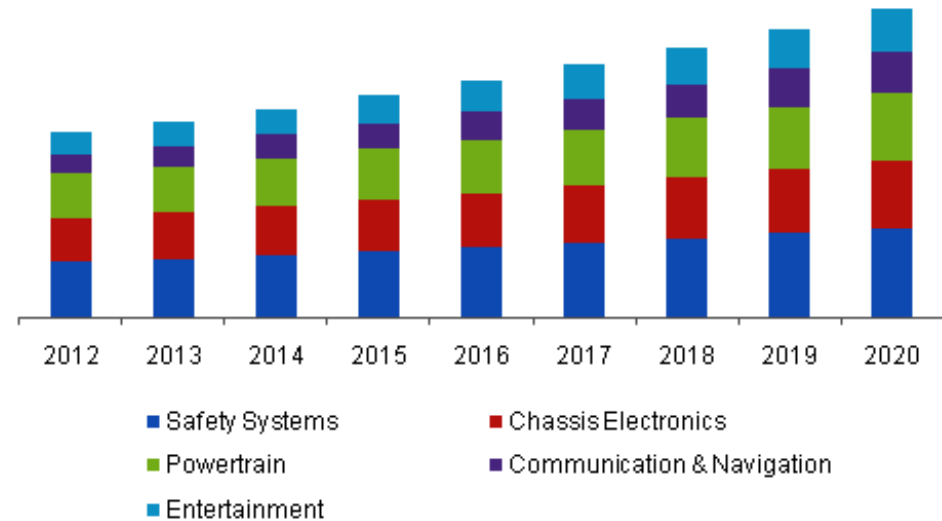
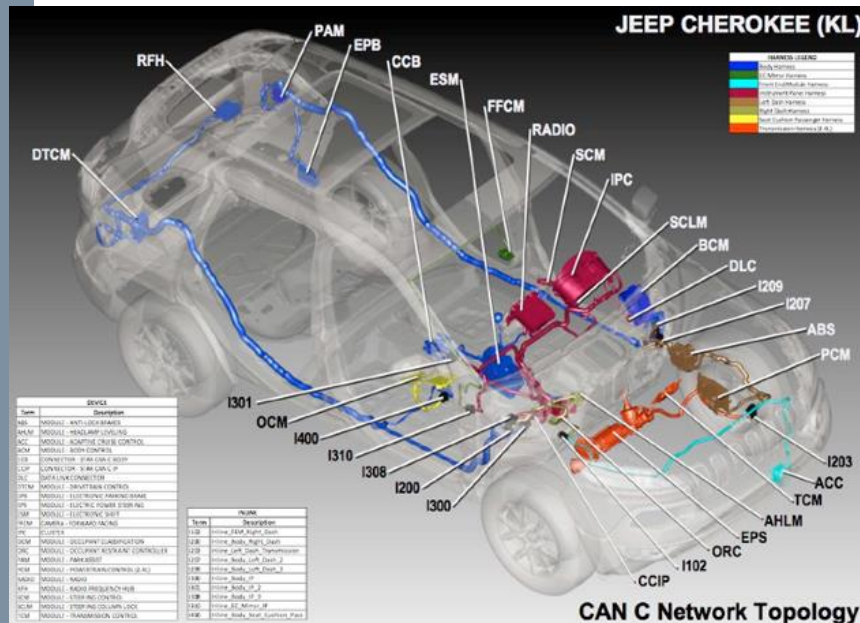
Computers Everywhere

Today, **98%** of all processors are **embedded** in other objects



Example: ECU Adoption in Automotive

* Electronic Control Unit



Example:

- 2010 [Range Rover](#) contained 41 ECUs
- 2014 Range Rover contains 98 ECUs



Increasing Complexity

The price to be paid is a higher software **complexity**

Related problems

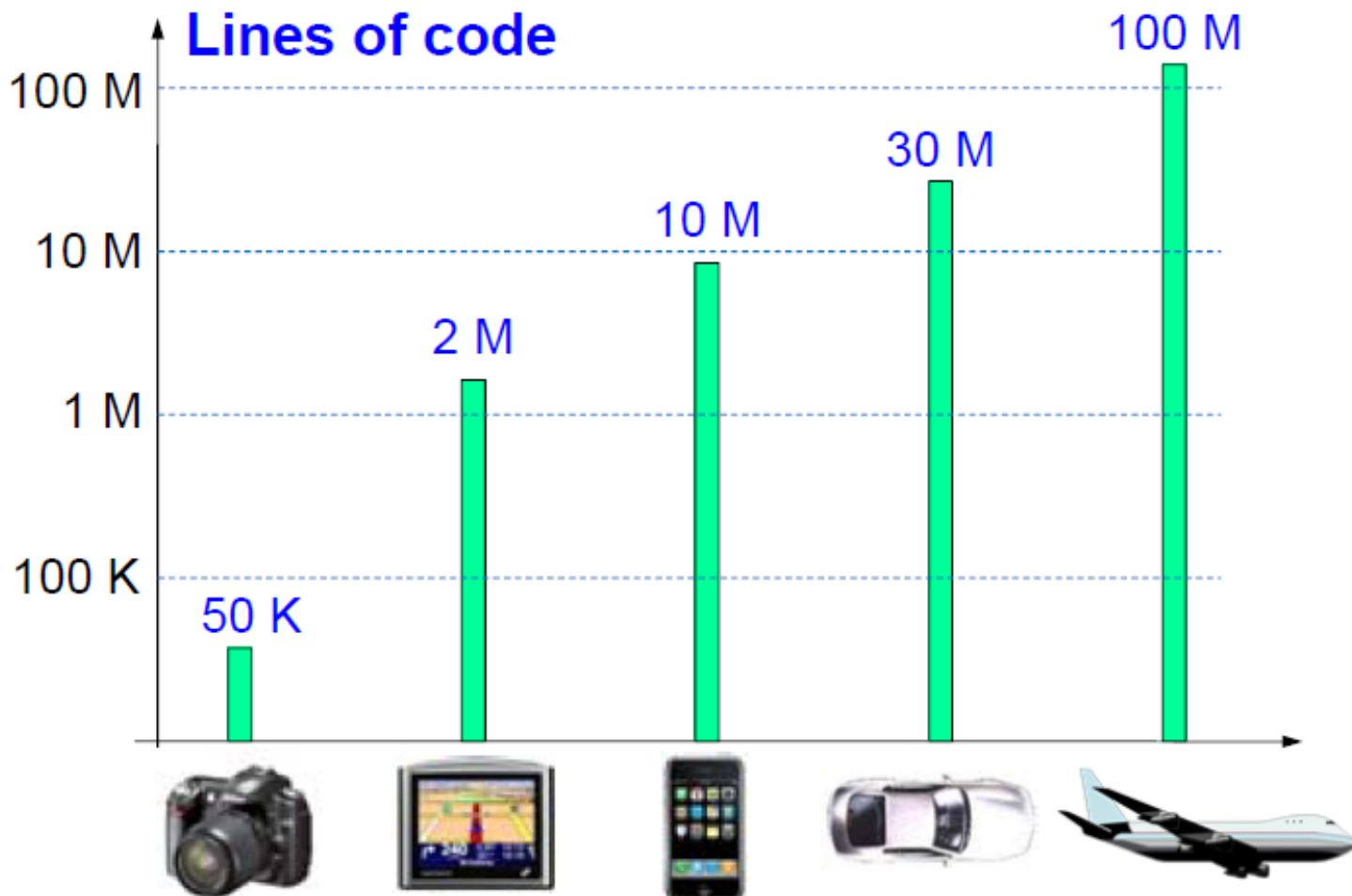
- › Difficult design
- › Less predictability
- › Less reliability

Novel solutions for

- › Component-based software design
- › Analysis for guaranteeing predictability and safety
- › Testing



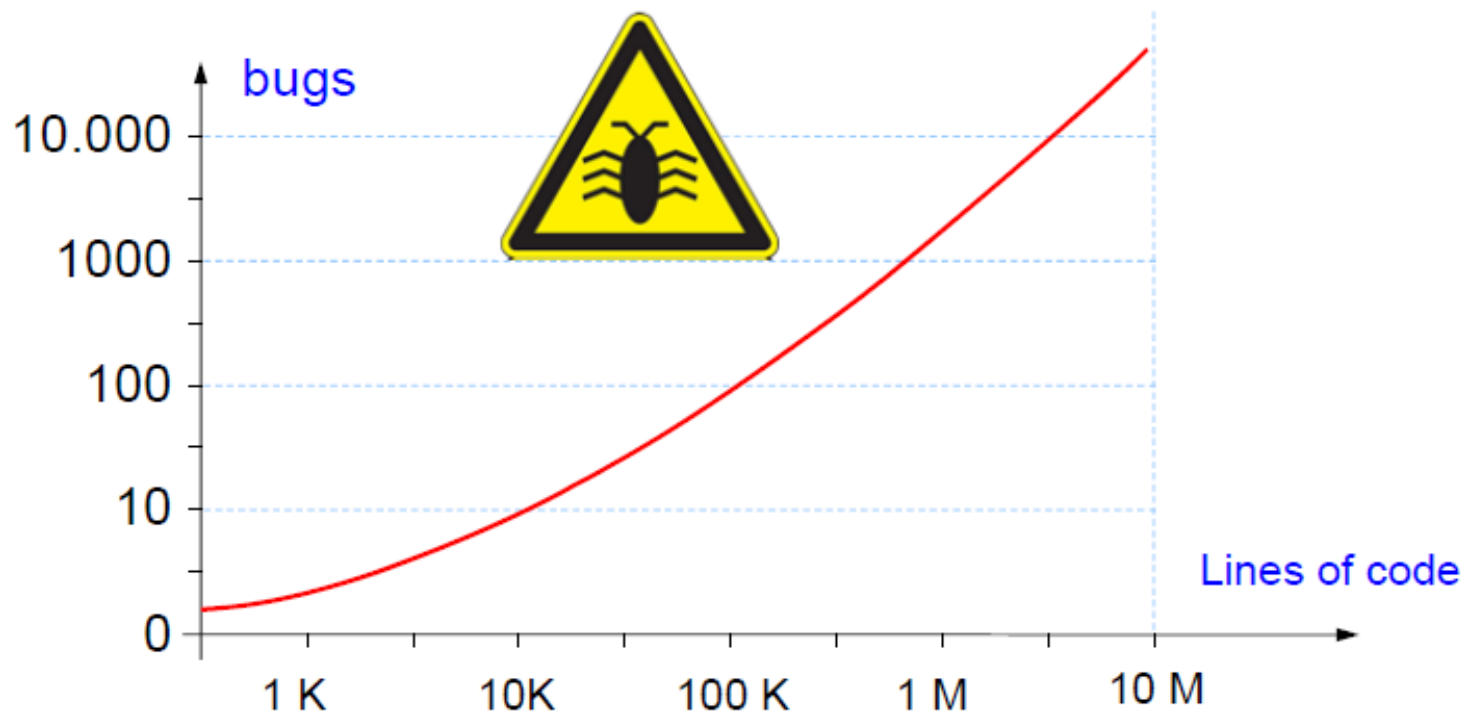
Comparing Software Complexity





Complexity and Bugs

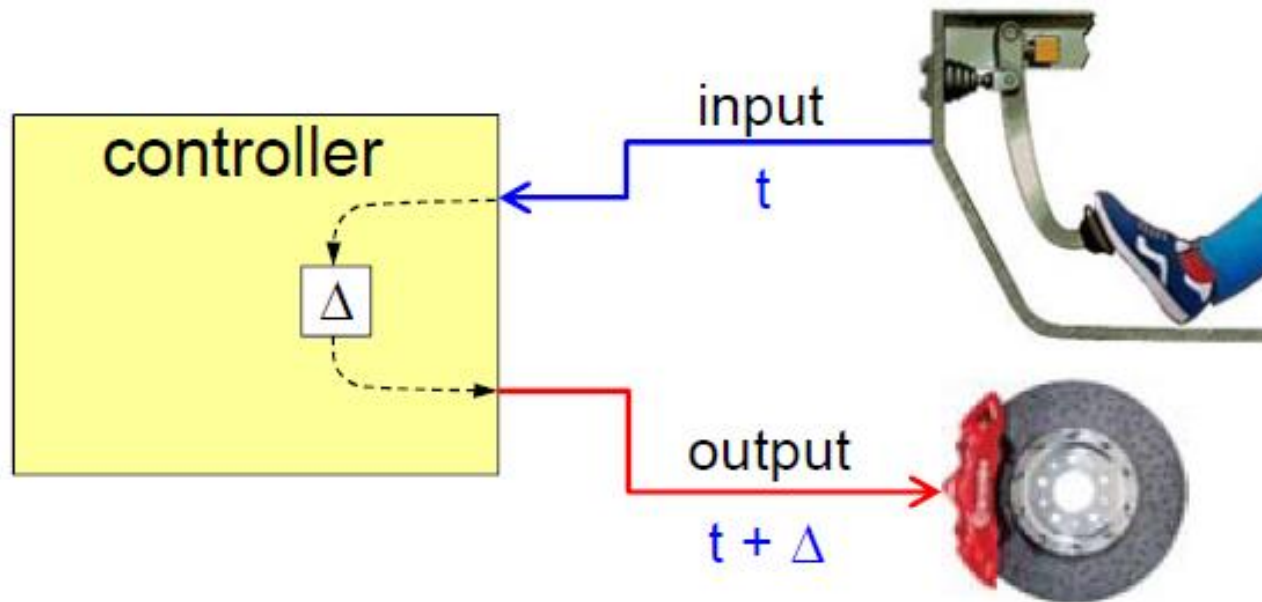
- › Bugs increase with complexity





...and the situation is ever worse....

- › Reliability does not only depend on the correctness of computation (bug-free)
- ›but also on having it **timely**



- › A correct action executed **too late** is useless or even dangerous!



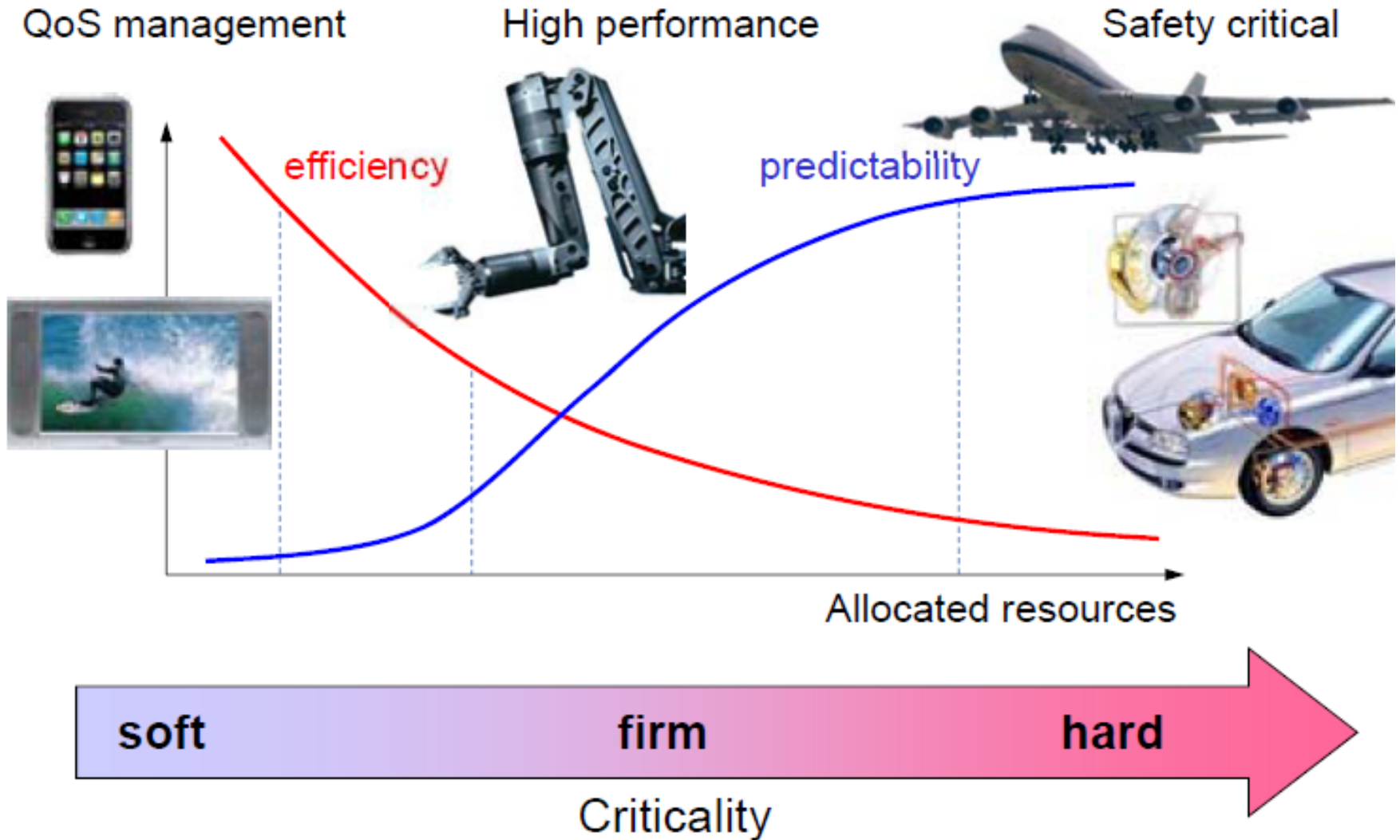
Real-Time Systems

“**Real-Time Systems** are computing systems that must guarantee bounded and predictable response times”

Predictability of response times must be **guaranteed**

- › for each critical activity
- › for all possible combination of events

Predictability vs Efficiency



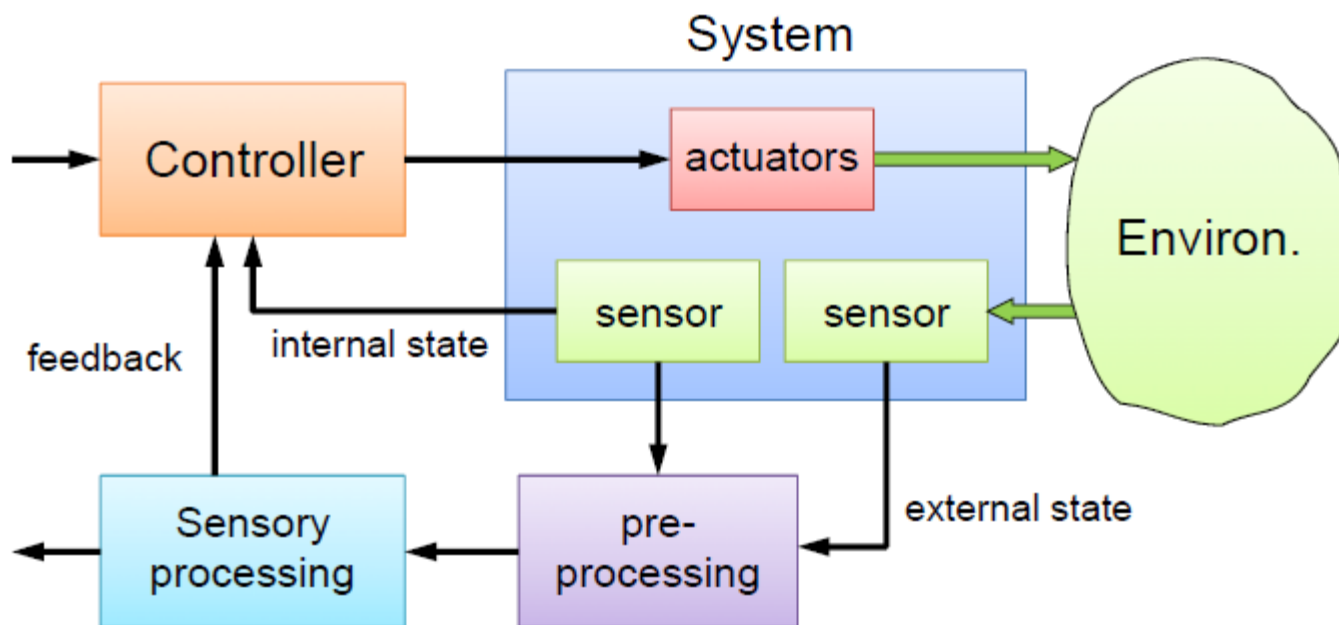


Embedded System Characteristics

FEATURES		REQUIREMENTS
Scarce resources (space, weight, time, memory, energy)	➔	High efficiency in resource management
High concurrency and resource sharing (high task interference)	➔	Temporal isolation to limit the interference
Interaction with the environment (causing timing constraints)	➔	High predictability in the response time
High variability on workload and resource demand	➔	Adaptivity to handle overload situations



Recap: a typical control system





Control and Implementation

Often, control and implementation are done by different people that do not talk to each other:

$$\dot{x} = Ax + Bu$$



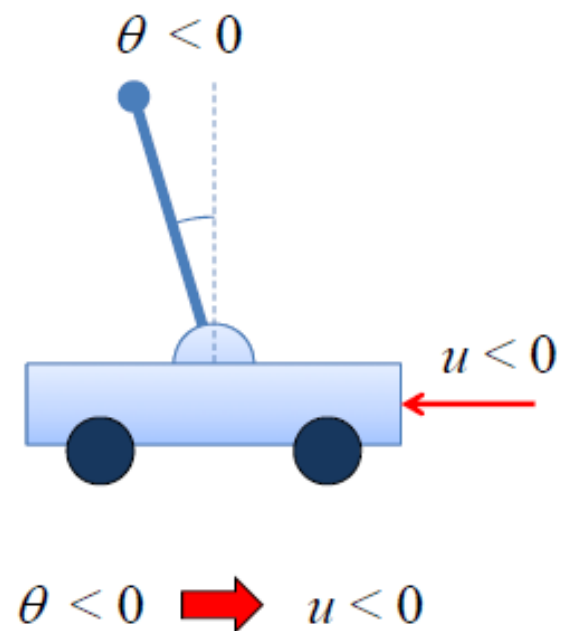
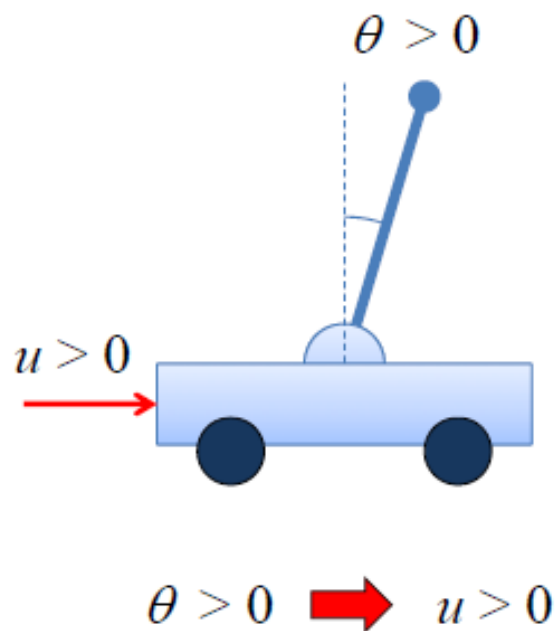
```
if (b != 0) y = a/b;  
else printf("error\n");
```



Control guys typically assume a computer with infinite resources and computational power

Example: Inverted Pendulum

A positive angle θ requires a positive control action u



A Control Task

```

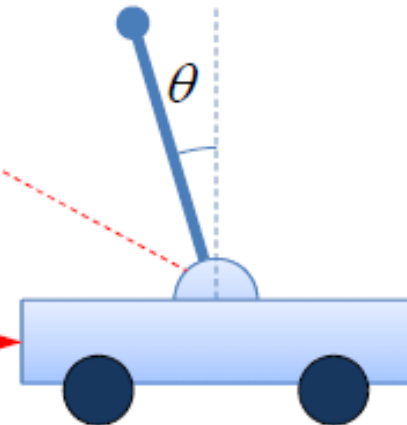
task control(float theta0, float k)
{
    float error;
    float u;
    float theta;

    while (1) {
        theta = read_sensor();
        error = theta - theta0;
        u = k * error;
        output(u);
        wait_for_next_period();
    }
}

```

control gain
reference angle

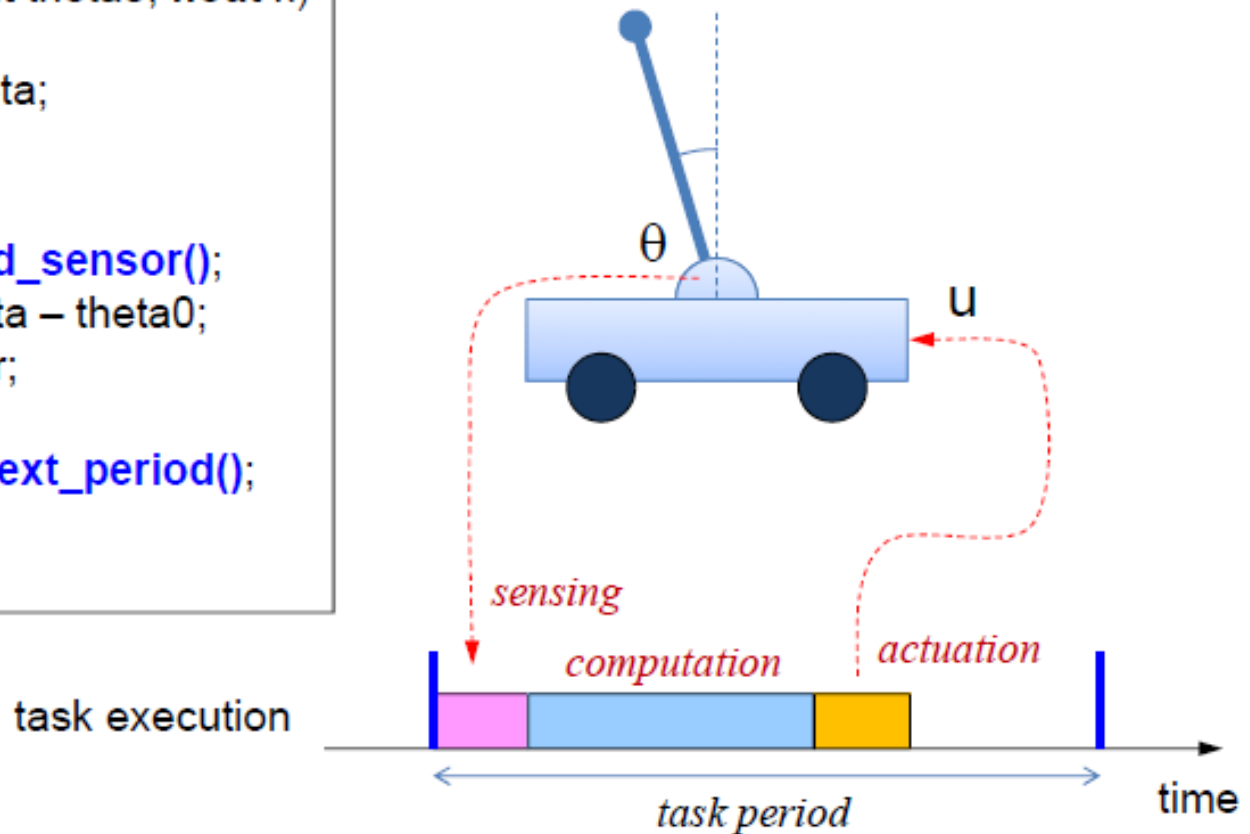
sensing
computation
actuation
synchronization



A Control Task

```
task control(float theta0, float k)
{
  float error, u, theta;

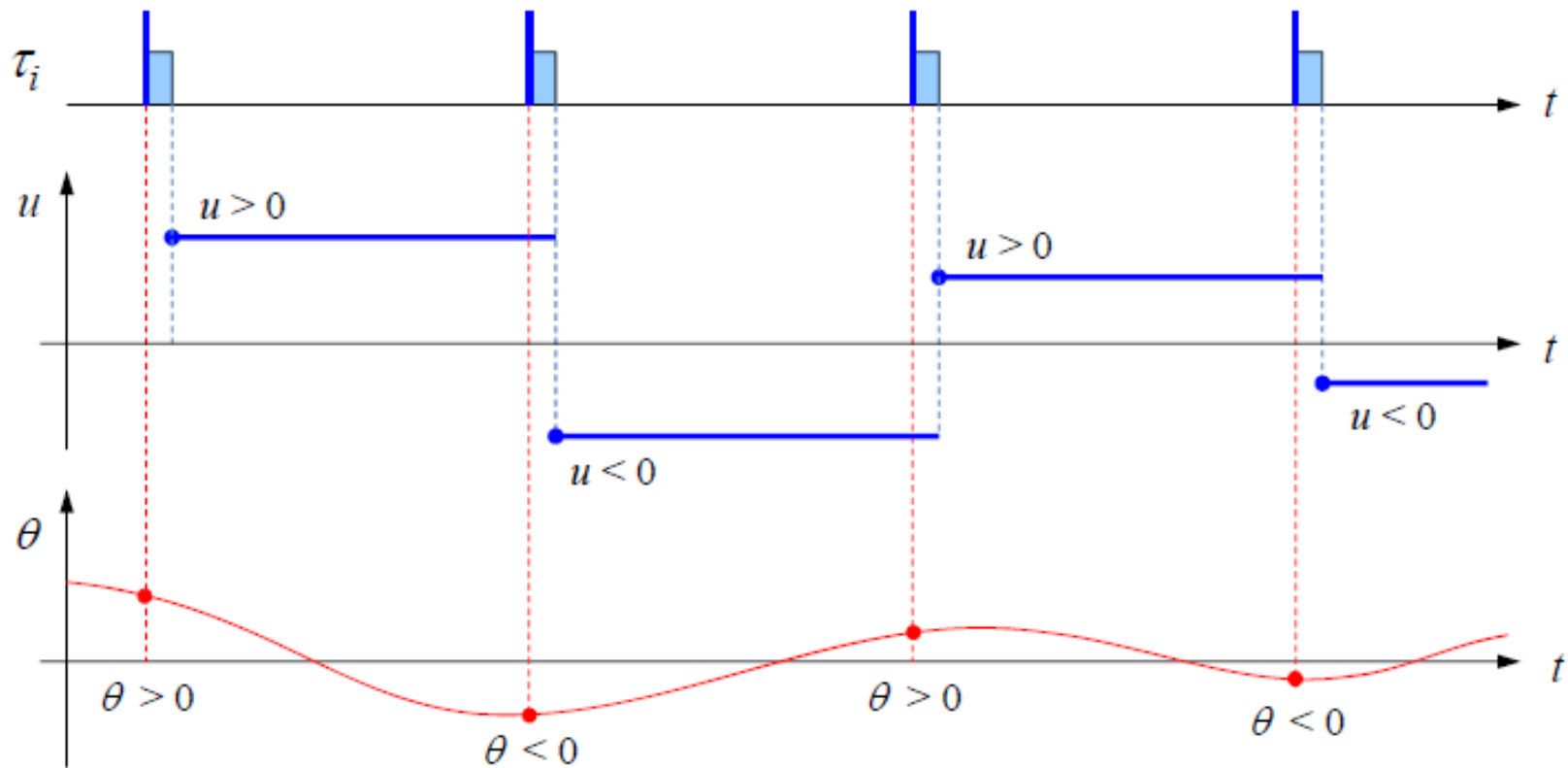
  while (1) {
    theta = read_sensor();
    error = theta - theta0;
    u = k * error;
    output(u);
    wait_for_next_period();
  }
}
```





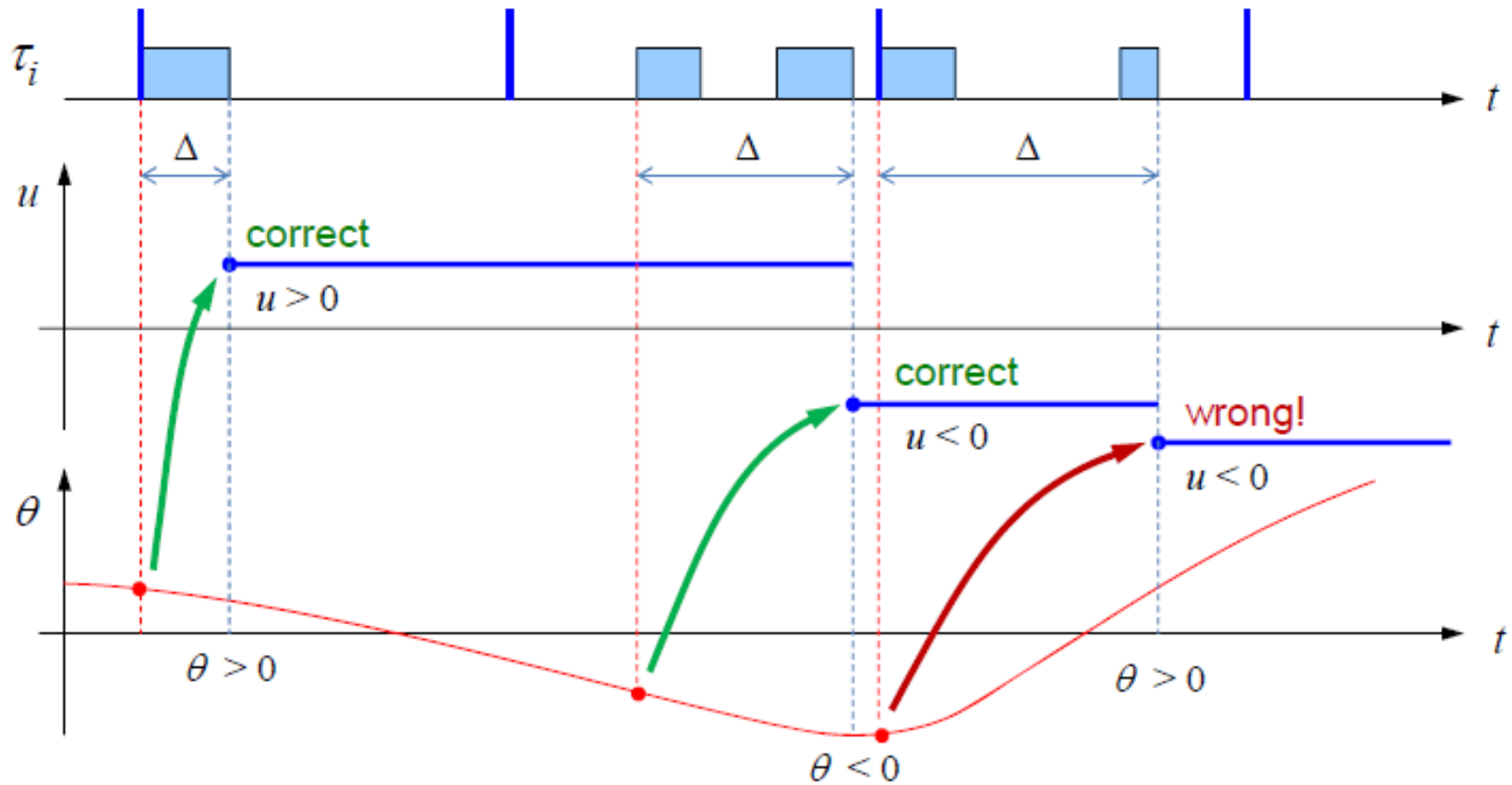
Traditional Control View

Negligible delay and jitter



Real Situation

Variable delay and jitter





Implications

- › The tight interaction with the environment requires the system to react to events within **precise timing constraints**
- › Timing constraints are imposed by the **performance** requirements and the **dynamics** of the system to be controlled

The operating system must be able to execute tasks within timing constraints



Design Requirements

Modularity

- › A subsystem must be developed without knowing the details of other subsystems (*system engineering and team work are essential*)

Configurability

- › Software must be adapted to different situations (through the use of suitable parameters) without changing the source code

Portability

- › Minimize code changes when porting the system to different hardware platforms

Predictability

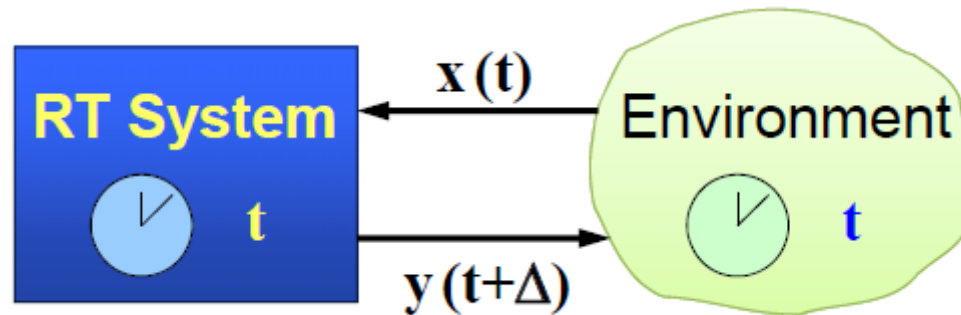
- › Allow the estimation of maximum delays

Efficiency

- › Optimize the use of available resources (computation time, memory, energy).

Cyber-Physical Real-Time Systems

It is a system in which the correctness depends not only on the output values, but also on the **time** at which results are produced



When we put the environment into the picture...

- › REAL means that system time must be synchronized with the time flowing in the environment



Real-Time \neq Fast

“To guarantee timing constraints it is sufficient to use faster and more powerful ECUs”



Real-Time \neq Fast



“To guarantee timing constraints, it is not sufficient to use faster and more powerful ECUs”



A real-time system is not a fast system



The objective of a fast system is to minimize the average response time

But ...

Real-time systems need to guarantee the **WORST CASE RESPONSE TIME**



Real-Time Requirements

Don't trust the average when you have to guarantee worst-case performance

“A guy once drowned crossing a river which was 10 inches deep on average”

A real-time system needs to guarantee that **multiple** critical tasks are **always** computed within well defined deadlines

- › Testing is often NOT sufficient
- › Timing behavior depends on actual situation **at runtime**

Worst case behavior **might never happen** in a lifetime!!!

- › This doesn't mean you can't identify and bound it **analytically**



Sources of non determinism

Platform architecture

- › Cache, pipelining, interrupts, DMA

Operating system

- › Scheduling, synchronization, communication

Programming Language

- › Lack of explicit support for time predictability

Design methodologies

- › Lack of analysis and verification techniques



Traditional (wrong) approach

Traditional RT applications are typically designed using empirical techniques:

- › Assembly programming
- › Timing through dedicated timers
- › Control through driver programming
- › Priority manipulation

Disadvantages

- › Tedious programming which heavily depends on programmer's ability
- › Difficult code understanding
- › Difficult maintainability
 - Millions LoC → understanding takes more than rewriting
- › Difficult to verify timing constraints
- › High risk of undetected failures
 - Low reliability

A new approach

Tests, although necessary, allow only a **partial verification** of system's behavior

- › Analytical design
- › Component by component
- › Interaction between component is ~~also~~ modeled **first**

Predictability at the level of the **controller, operating system** and **ECUs**

- › They are our “actuators”

Critical systems must be designed under **pessimistic assumptions**

- › Think of **Worst case**





Real-Time Operating System (RTOS)

A real-time operating system is responsible for:

- › Managing **concurrency**
- › Activating periodic tasks at the beginning of each period (**time management**)
- › Deciding the execution order of tasks (**scheduling**)
- › Solving possible timing conflicts during the access of shared resources (**mutual exclusion**)
- › Manage the timely execution of asynchronous events (**interrupt handling**)



Multi-process and multi-task management



What are processes?

A process is an executing program

(an OS can execute many processes at the same time → Concurrency)

A (sequential) process goes through **states**, which change over time

- › what are the actual data values?
 - set of processor registers + vars
- › what is processor doing?
 - running, waiting,



Example for (data) state: GCD

› Greatest common divisor

```
int gcd(int a, int b)
{
    while (a!=b)
    {
        if (a < b) b = b - a;
        else a = a - b;
    }
    return a;
}
```

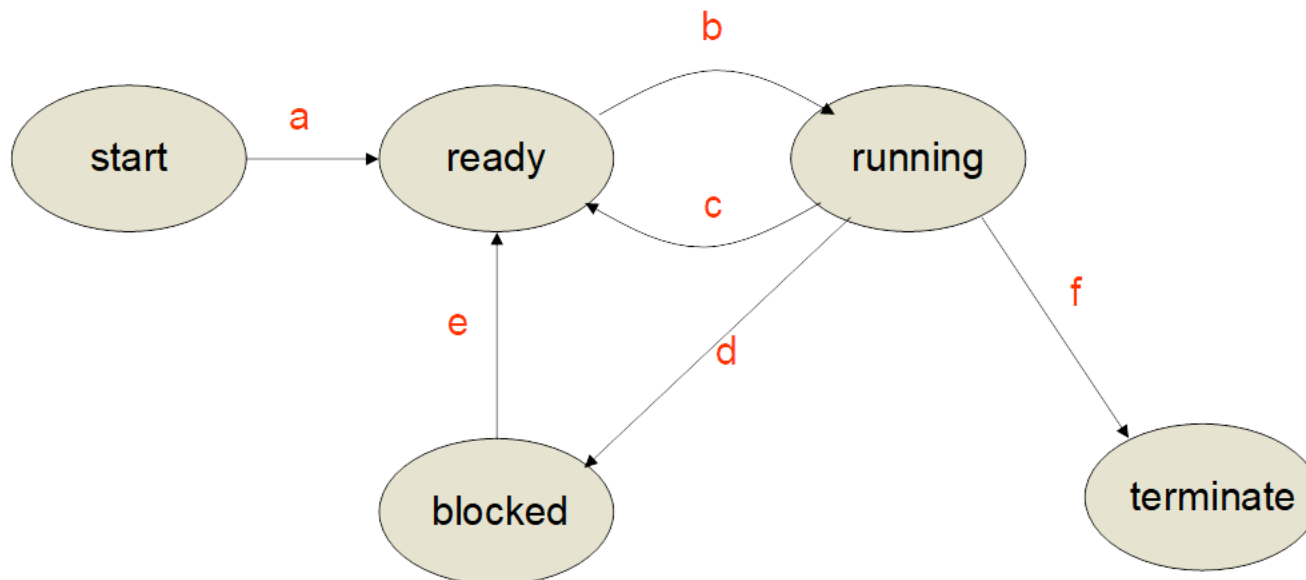
Step	a	B
1	21	15
2	6	15
3	6	9
4	6	3
5	3	3



Process states

The OS executes many processes at the same time, each of them is either:

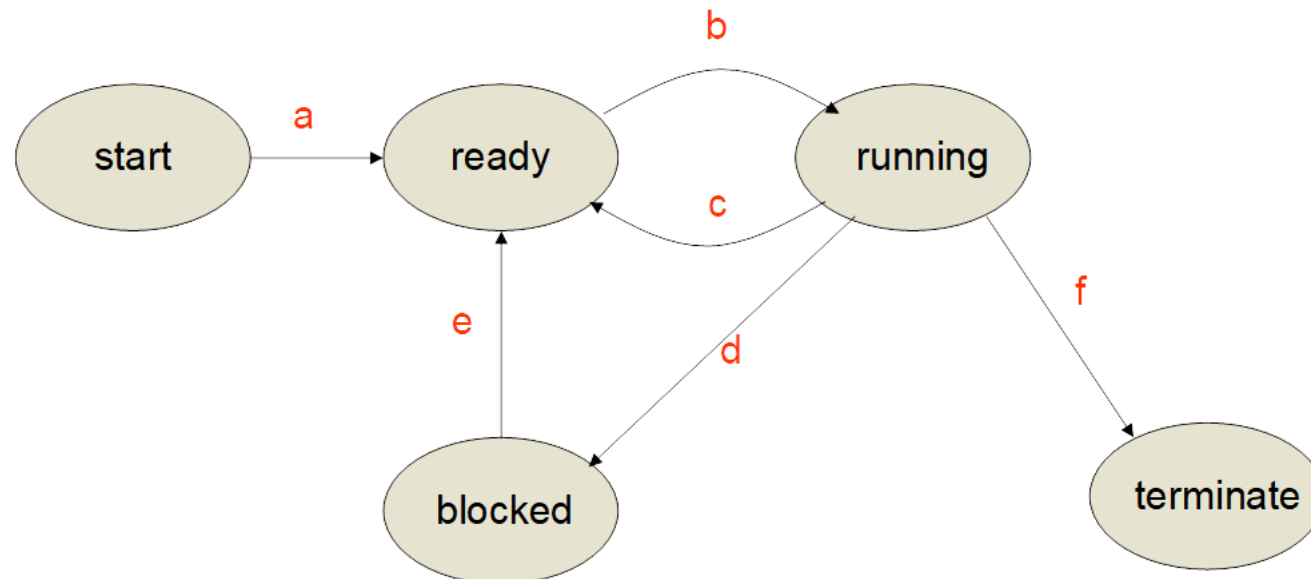
- › starting (the process is being created)
- › ready (the process is ready to be executed)
- › executing (the process is executing)
- › blocked (the process is waiting on a condition)
- › terminating (the process is about to terminate)





Process state events

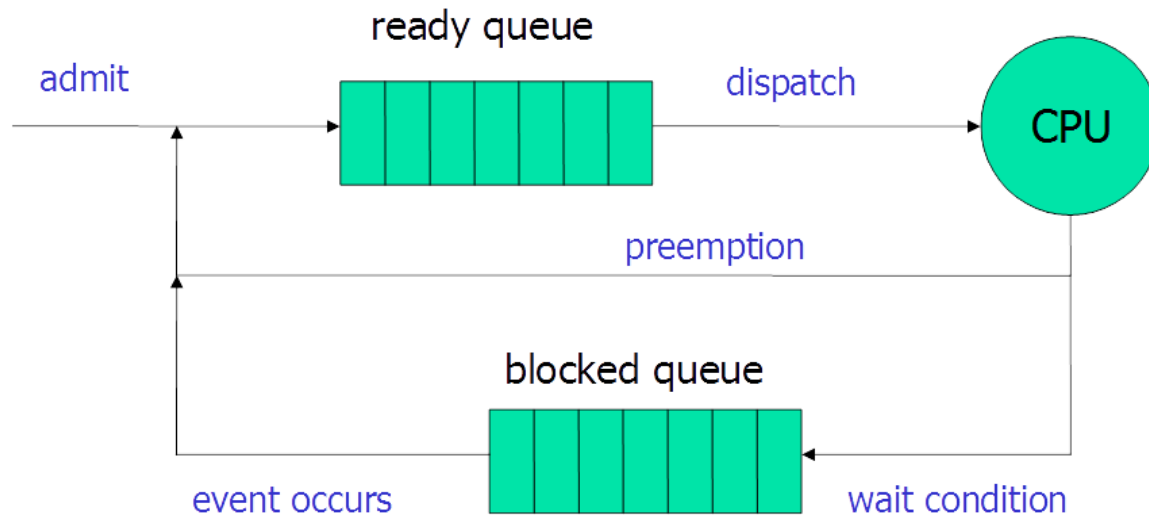
- a) Creation the process is created
- b) Dispatch the process is selected to execute
- c) Preemption the process leaves the processor
- d) Wait on condition the process is blocked on a condition
- e) Condition true the process is unblocked
- f) Exit the process terminates





Scheduling— single processor

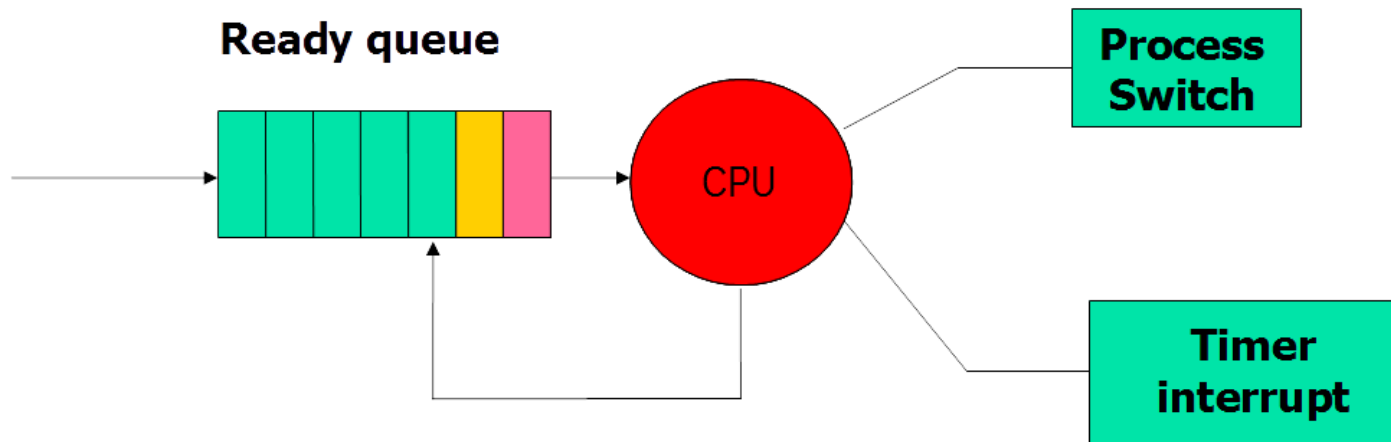
- › The scheduling problem: choose which process goes first





Time sharing - fairness

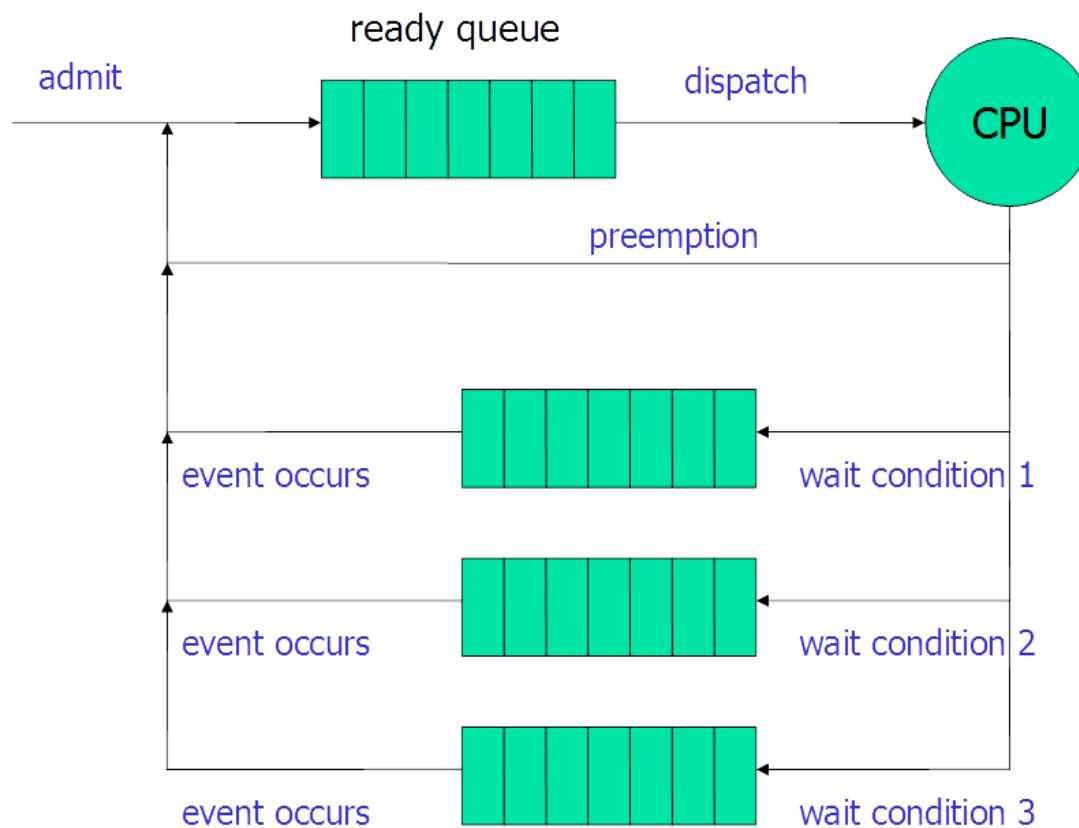
- › Given a time T (e.g., 1 sec)
- › be sure we allocate the CPU at least T/N , where $N = \text{\#processes}$





Scheduling – multi processor

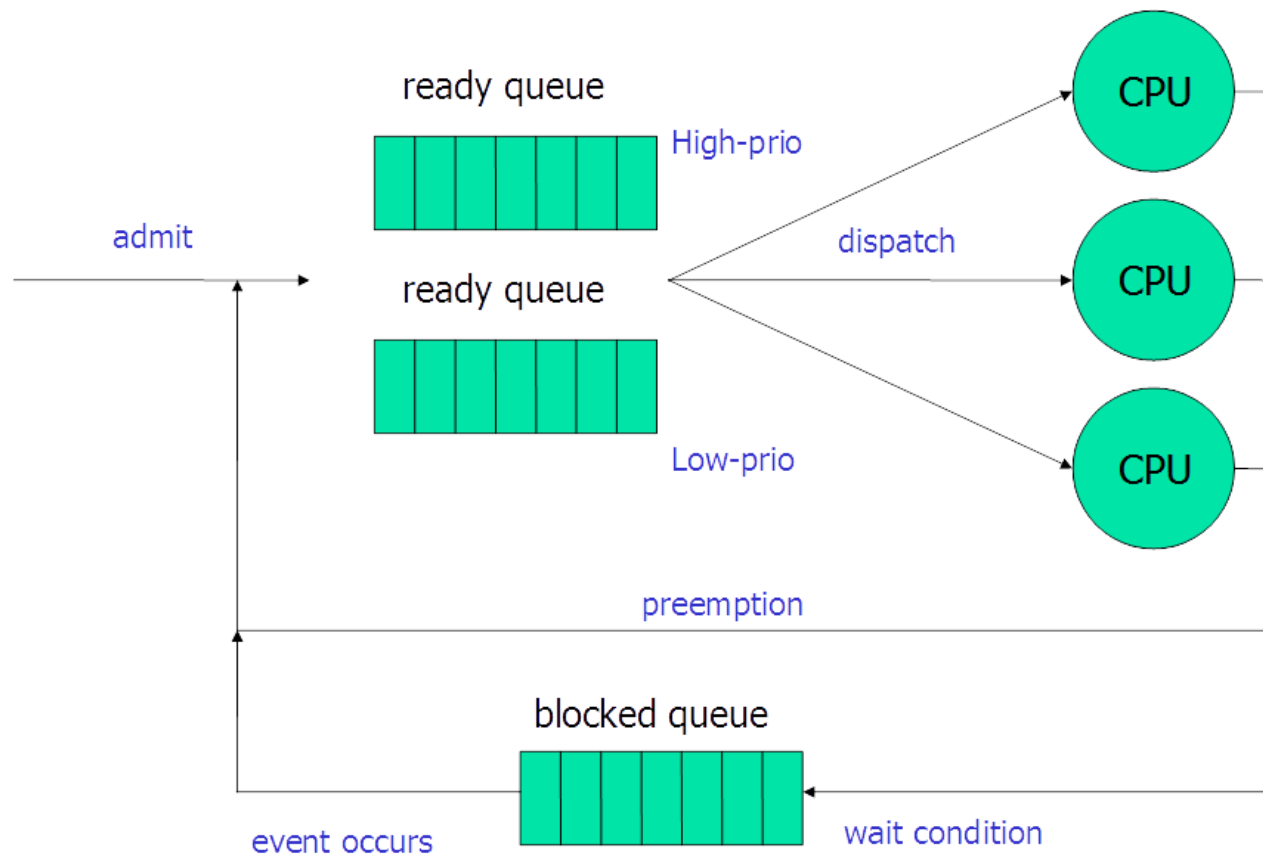
- › Multiple wait queues, a single ready queue





Priority scheduling

- › Multiple ready queues associated with PRIORITY





Process switch

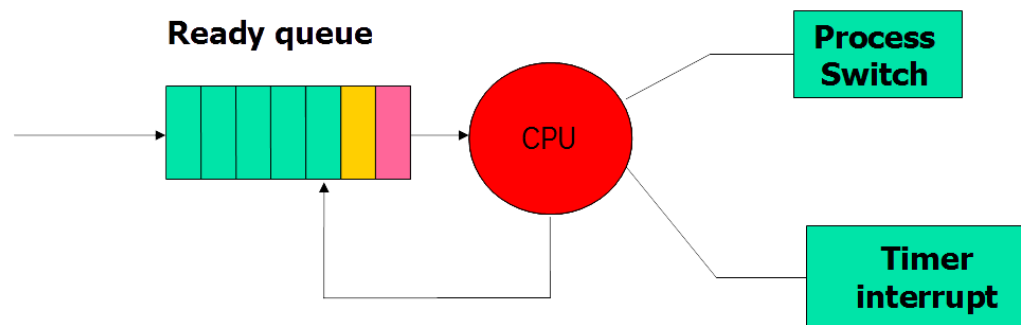
A process goes to the wait queue when it gives control to the OS

› Typically: system calls

...but we don't want a "bad" process might reserve 100% of processor!!

A switch can happen if:

- › The process has been "**preempted**" by another higher priority process
- › The process **blocks** on some condition, or syscall
- › In **time-sharing** systems, the process has completed its "round" and it is the turn of some other process





Scheduling and resources

Scheduling/execution

- › The execution of a process follows an execution path, and generates a trace (sequence of internal states)
- › It has a state (ready, running, etc.) and scheduling parameters (priority, time left in the round, etc.)
- › Already seen

Resource ownership

- › A process includes a virtual address space, a process image (code + data)
- › It is allocated a set of resources, like file descriptors, I/O channels, etc.
- › We won't see this (for the moment..)



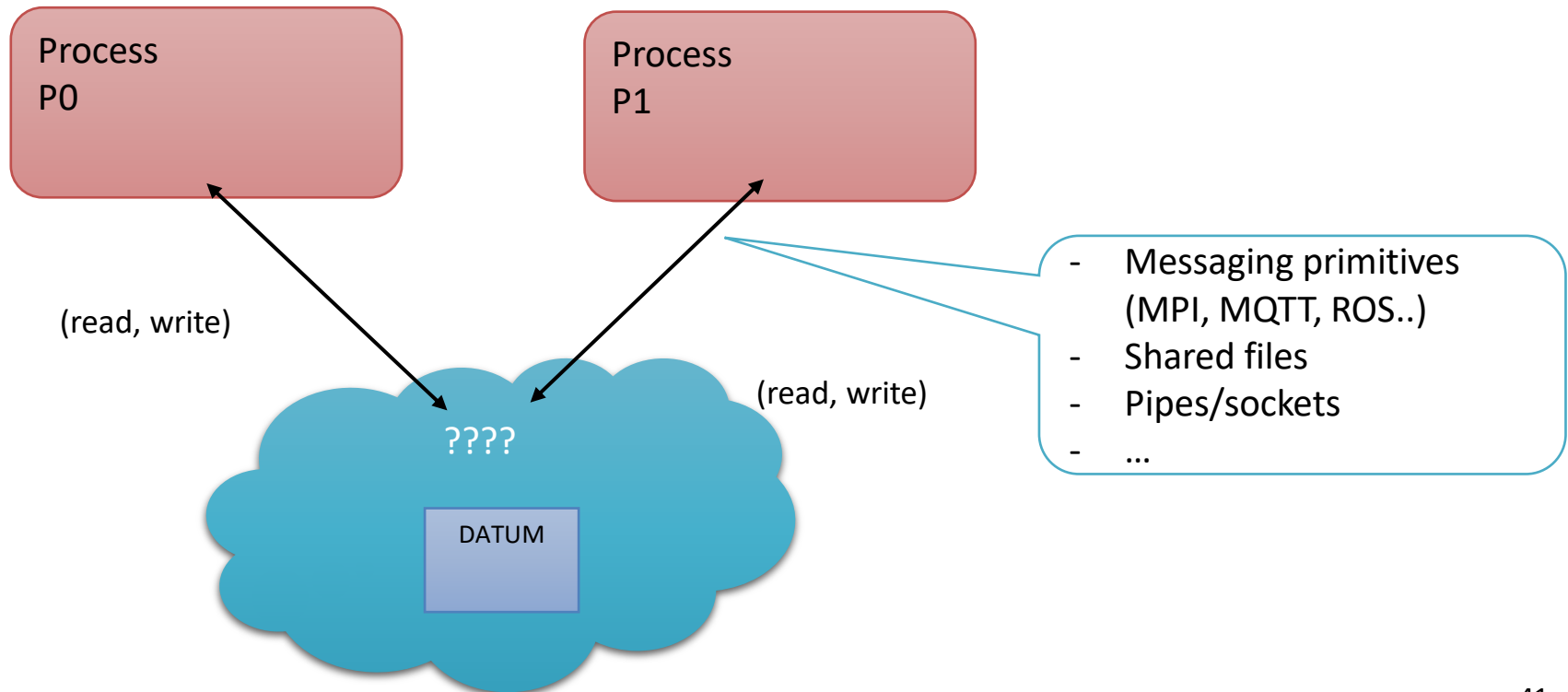
Multi-threading



Multi-processing: limitations

Typically, processes do not share memory

- › To communicate between process, it is necessary to use OS primitives: heavy and cumbersome
- › Process switch is more complex because we have to change address space





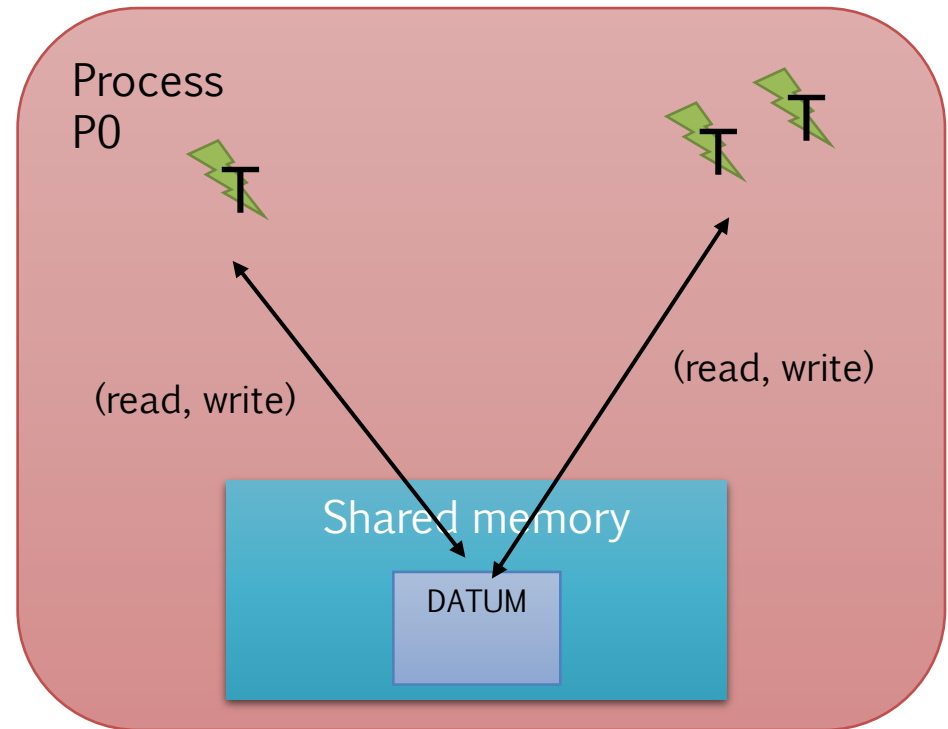
Multi-threading

Threads in the same process share the same address space

- › They can access the same variables in memory
- › Communication between threads is simpler
- › Thread switch has less overhead

If possible, preferred for implementing concurrent applications

Let's see
this in
action





Processes vs. threads

Speed of creation


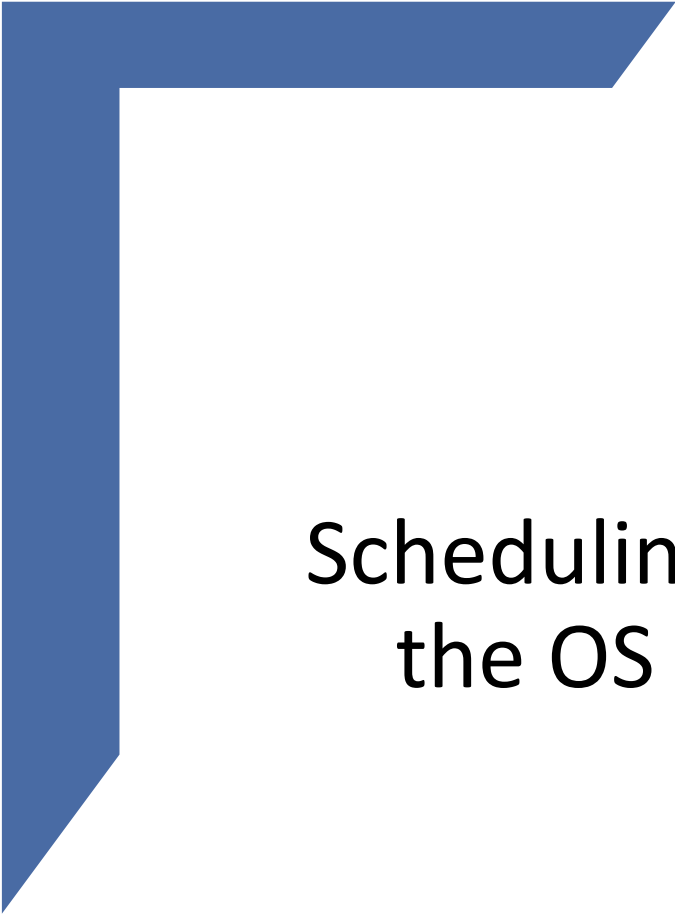
- › Creating a thread takes far less time than a process

Speed of switching

- › Thread switch is faster than process switch

Shared memory

- › Threads of the same process run in same memory space
- › You don't need to use heavyweight primitives such as sockets, and message-passing

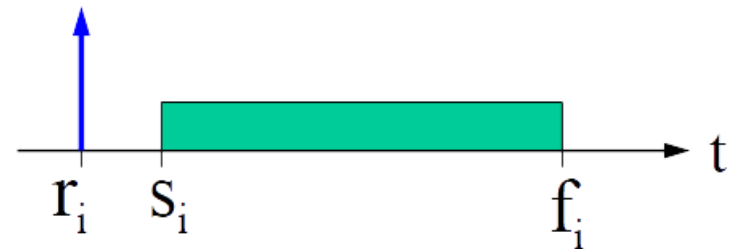
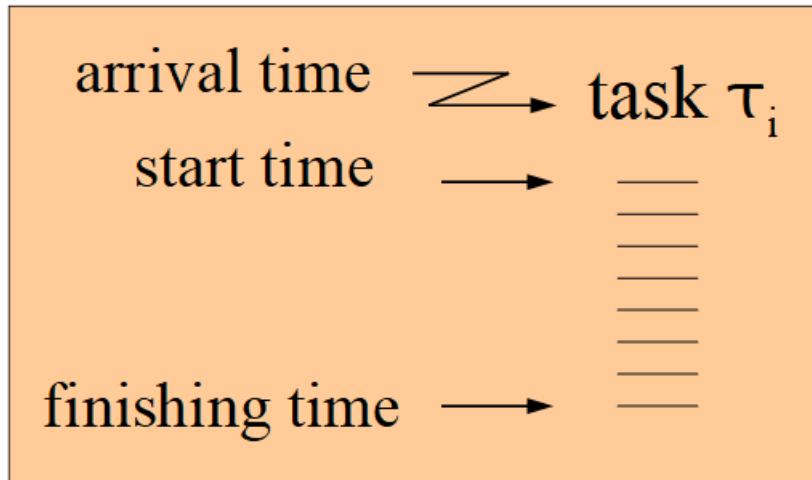


Scheduling theory from the OS perspective

Definitions: tasks

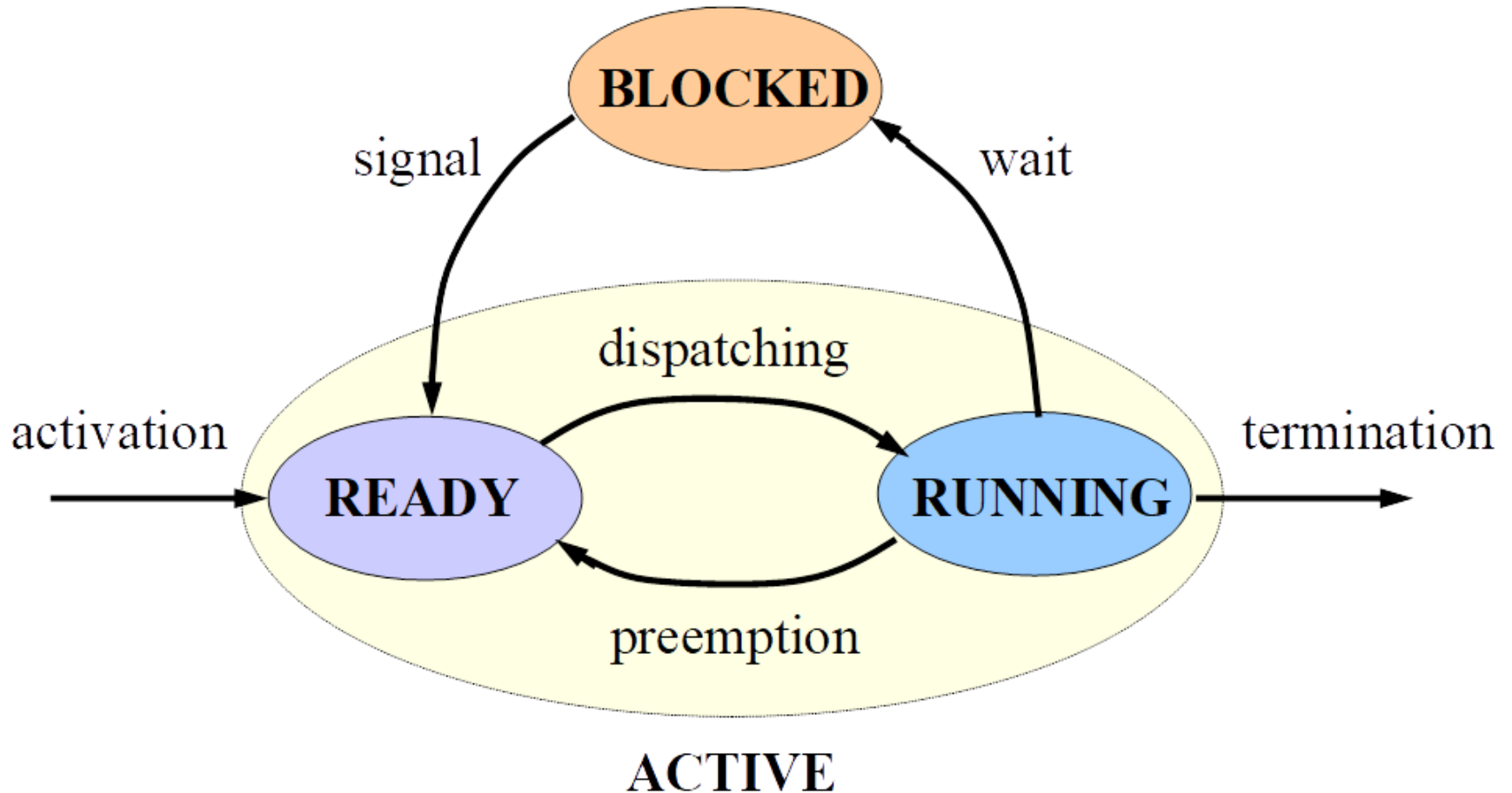
“A **task** is a sequence of instructions that in absence of other activities is continuously executed by the processor until completion”

- › It can be a process or a thread depending on the operating system





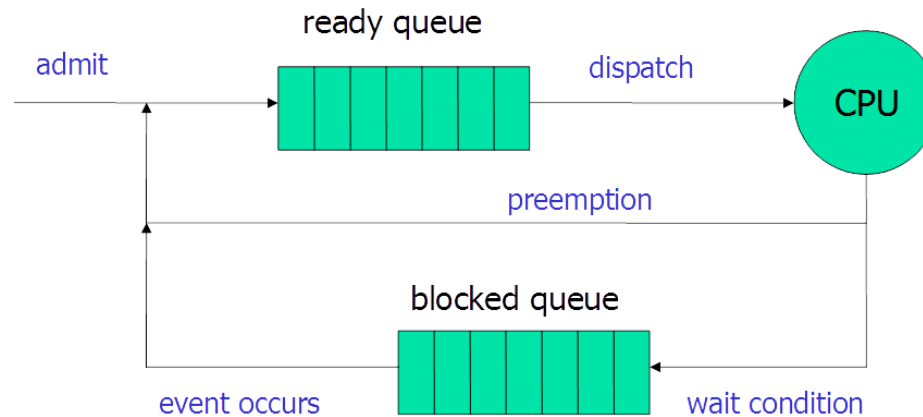
Task state transitions





Task scheduling

- › The ready tasks are kept in a waiting queue, called the ready queue;
- › The strategy for choosing the ready task to be executed on the CPU is the **scheduling algorithm**



Can be

- › **Preemptive** : if the running task can be temporarily suspended to execute a more important task.
- › **Non-preemptive** : if the running task cannot be suspended until completion.



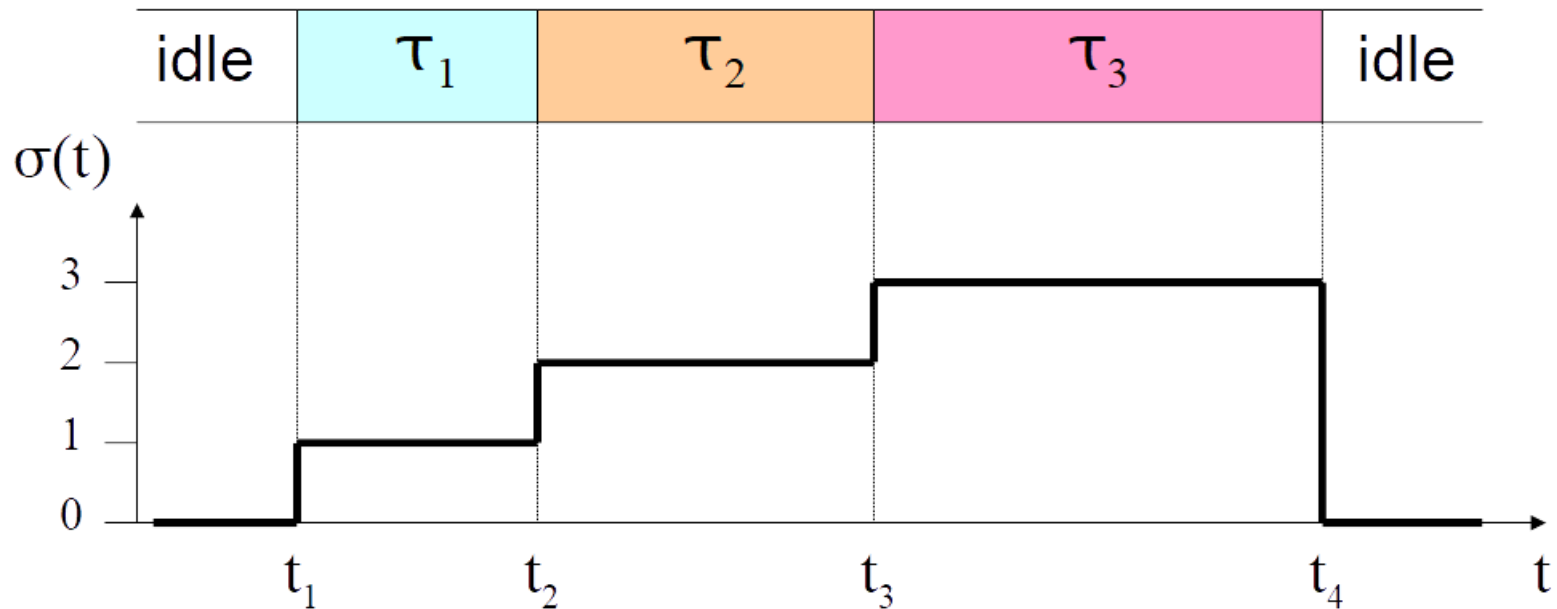
Schedule

A particular assignment to task(s) to processor(s)

- › Given a task set $\Gamma = \{\tau_1, \dots, \tau_n\}$, a schedule is a mapping $\sigma : \mathbb{R}^+ \rightarrow \mathbb{N}$ such that $\forall t \in \mathbb{R}^+$:

$$\sigma(t) = \begin{cases} k > 0 & \text{if } \tau_k \text{ is running} \\ 0 & \text{if the processor is idle} \end{cases}$$

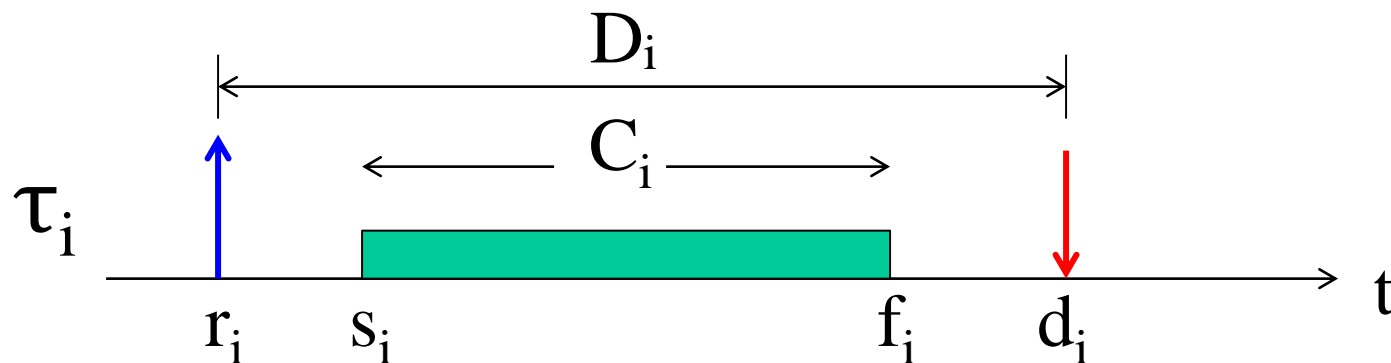
Schedule: example



- › At time t_1 , t_2 , t_3 , and t_4 a **context switch** is performed
- › Each interval $[t_i, t_{i+1})$ is called a **time slice**

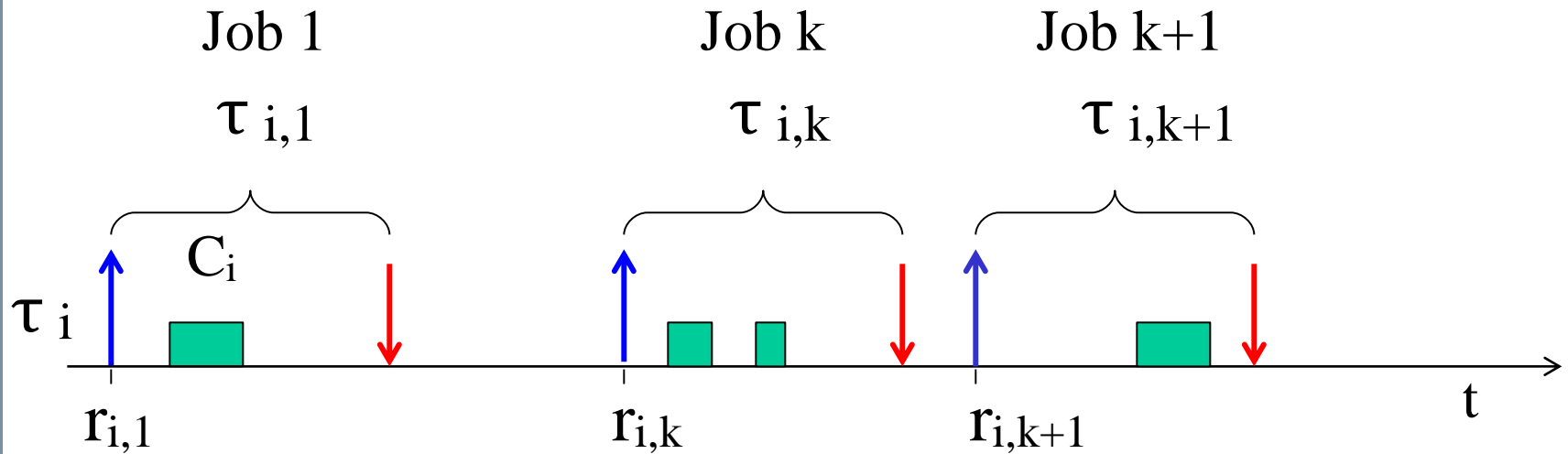
Real-time tasks

- › r_i request time (arrival time a_i)
- › s_i start time
- › C_i worst-case execution time (WCET)
- › d_i absolute deadline
- › D_i relative deadline
- › f_i finishing time



Tasks and Jobs

- › a task is an infinite sequence of instances (jobs):





Task criticality

HARD tasks

- › all jobs must meet their deadlines: missing a deadline may have serious consequences
 - sensory acquisition
 - low-level control
 - sensory-motor planning

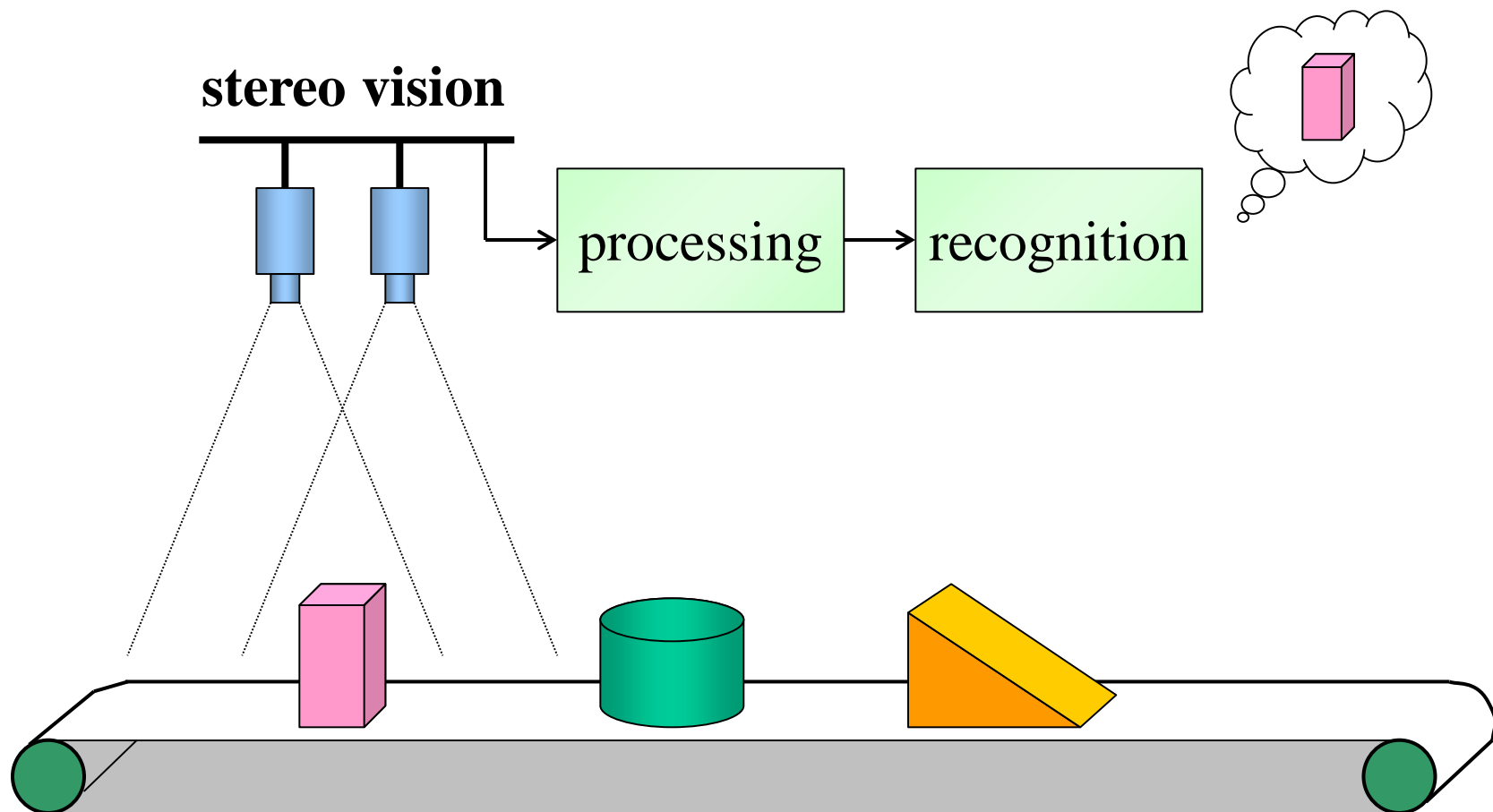
FIRM tasks

- › only some jobs can miss their deadline

SOFT tasks

- › jobs may miss deadlines: the goal is to minimize responsiveness
 - reading data from the keyboard
 - user command interpretation
 - message displaying
 - graphical activities

Sample application





Activation modes

time driven

periodic tasks

- › the task is automatically activated by the kernel at regular intervals.

event driven

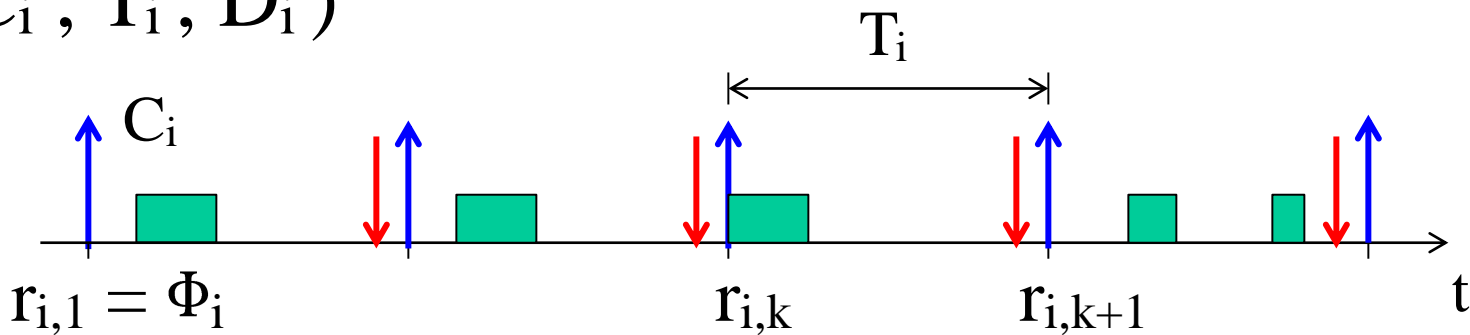
aperiodic tasks

- the task is activated upon the arrival of an event or through an explicit invocation of the activation primitive.

Periodic task model

$$\left\{ \begin{array}{l} > r_{i1} = \Phi_i \\ > r_{i,k+1} = r_{i,k} + T_i \end{array} \right.$$

$\tau_i (C_i, T_i, D_i)$



$$r_{i,k} = \Phi_i + (k-1) T_i$$

$$d_{i,k} = r_{i,k} + D_i$$

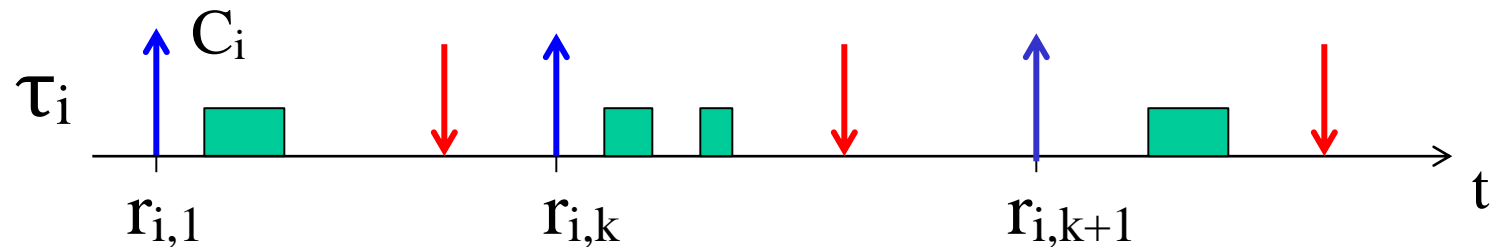
$$\left[\begin{array}{l} \text{often} \\ D_i = T_i \end{array} \right]$$



Aperiodic task model

Aperiodic: $r_{i,k+1} > r_{i,k}$

Sporadic: $r_{i,k+1} \geq r_{i,k} + T_i$





Task constraints

Timing constraints

- › deadline, activation, completion, jitter

Precedence constraints

- › they impose an ordering in the execution

Resource constraints

- › they enforce a synchronization in the access of mutually exclusive resources.



Timing constraints

Explicit

- Included in the specification of the system activities.

Examples

- › open the valve in 10 seconds
- › send the position within 40 ms
- › read the altimeter every 200 ms

Implicit

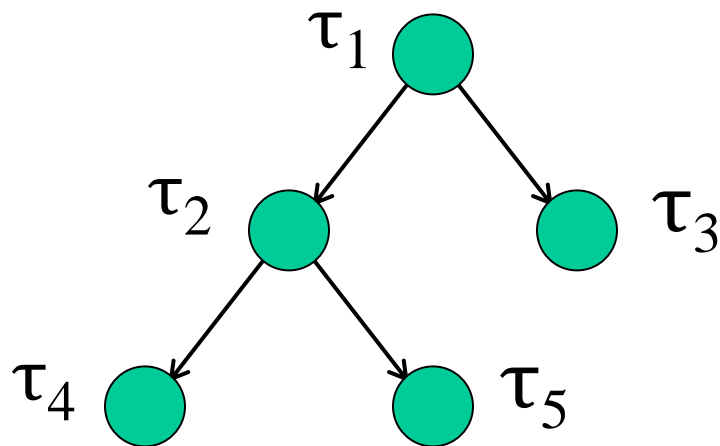
- Do not appear in the system specification but must be respected to meet the requirements.

Examples

- › avoid obstacles while running at speed v .
- › control an inverted pendulum of height h and weight w .

Precedence constraints

Sometimes tasks must be executed with specific precedence relations, specified by a **Directed Acyclic Graph - DAG**



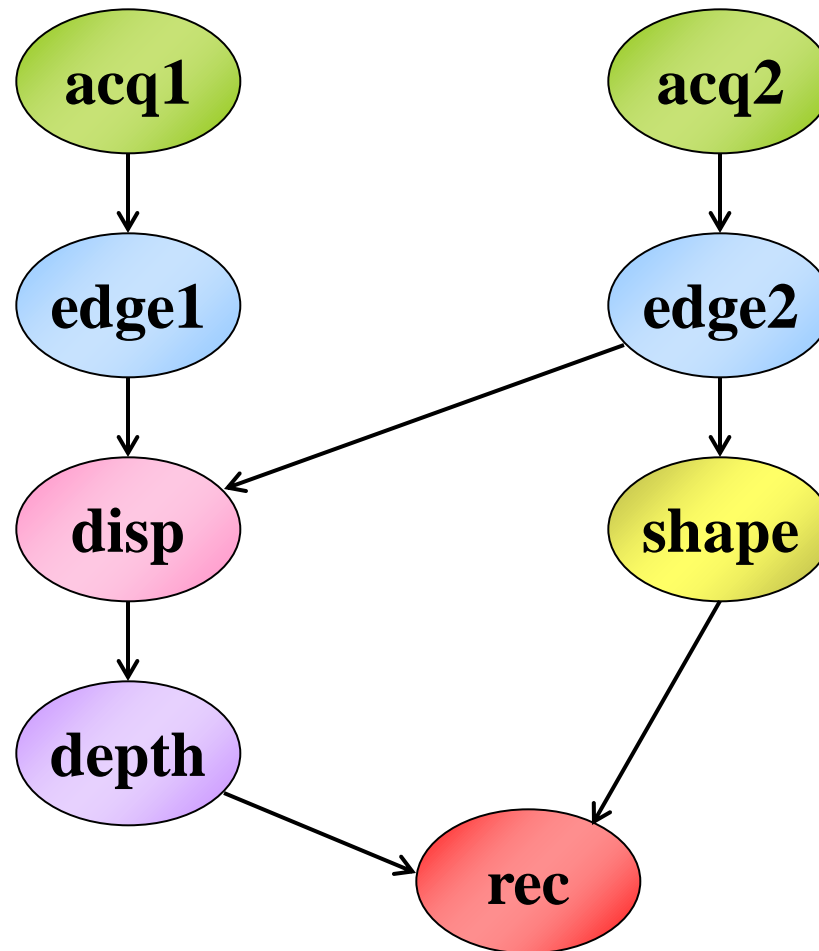
predecessor

$$\tau_1 \prec \tau_4$$

immediate predecessor

$$\tau_1 \rightarrow \tau_2$$

Precedence graph



References



Course website

- › http://hipert.unimore.it/people/paolob/pub/Industrial_Informatics/index.html

My contacts

- › paolo.burgio@unimore.it
- › <http://hipert.mat.unimore.it/people/paolob/>

Resources

- › Giorgio Buttazzo, "Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications". 3rd Edition. 2011. Springer
- › "Real-Time Embedded Systems" course by Prof. Bertogna @UNIMORE
- › A "small blog"
 - <http://www.google.com>