Introduction to Distributed Real-Time Systems

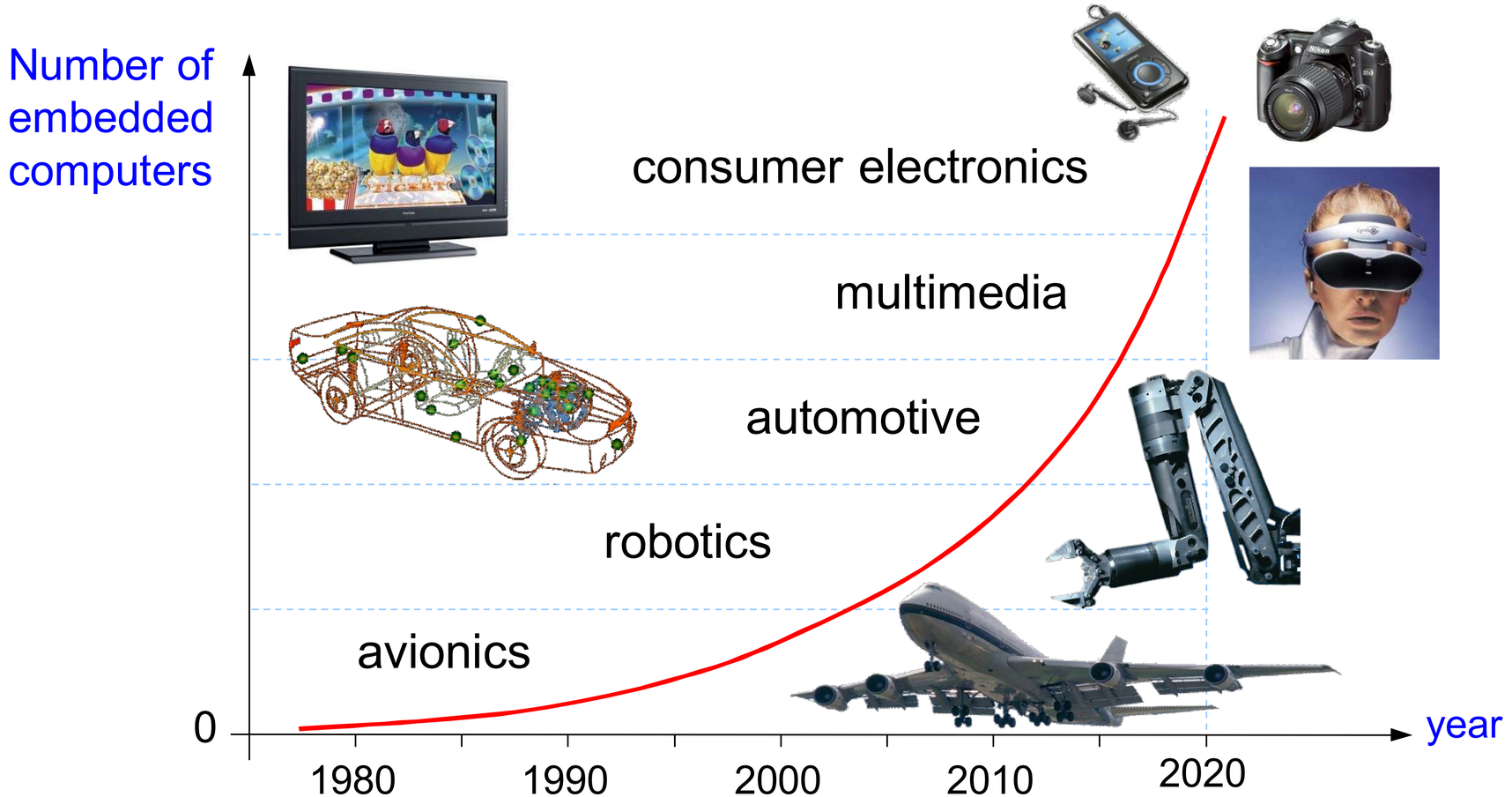
Slides source: Prof. Giorgio Buttazzo

Computers everywhere

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

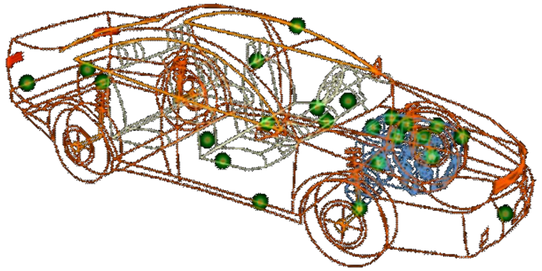
Evolution of CPS

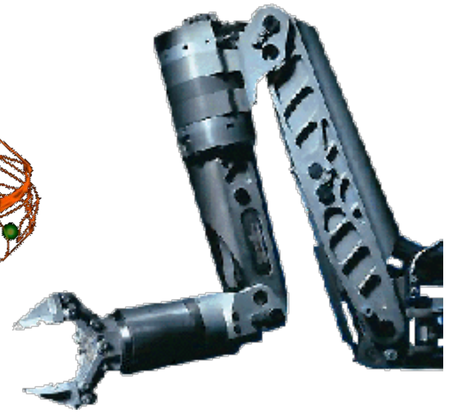
Cyber-physical systems have grown exponentially in several application domains:



Typical applications

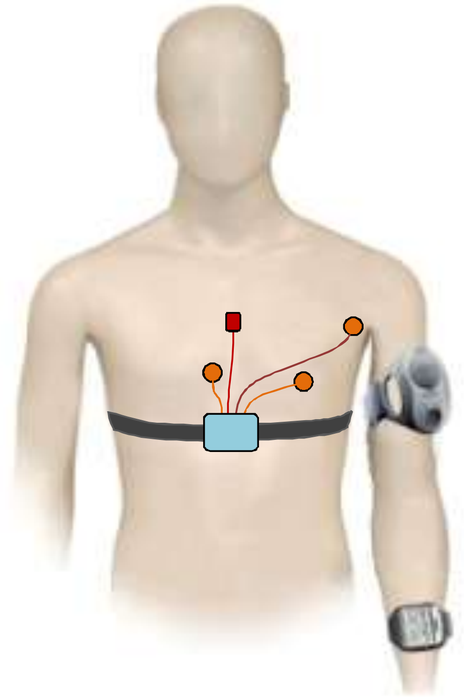
- avionics
- automotive
- robotics
- industrial automation
- telecommunications
- multimedia systems
- consumer electronics

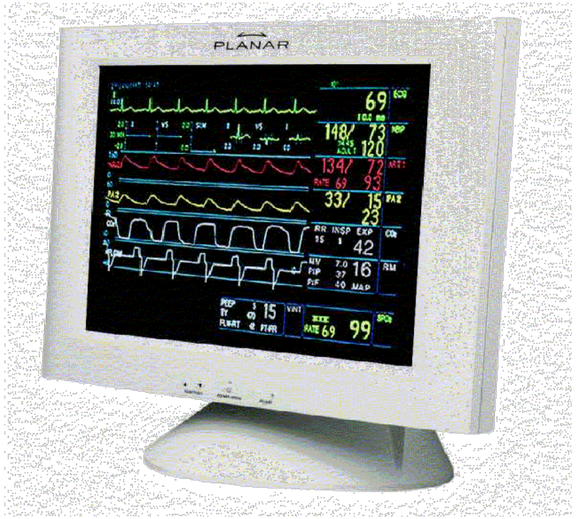




Health Care

- Tele-monitoring
- Tele-rehabilitation
- Assisted Living
- Sport





Smart objects

Smart Tire

Smart Shoe

Smart glasses

Smart cork

Inside body

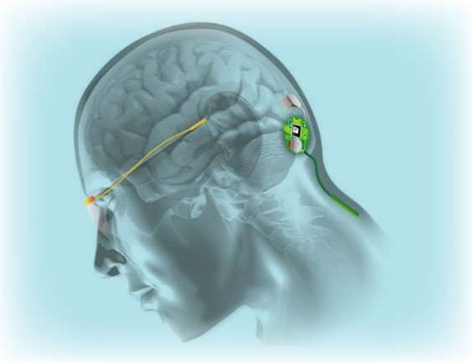
Computers will be embedded even in our body:

heart

ear

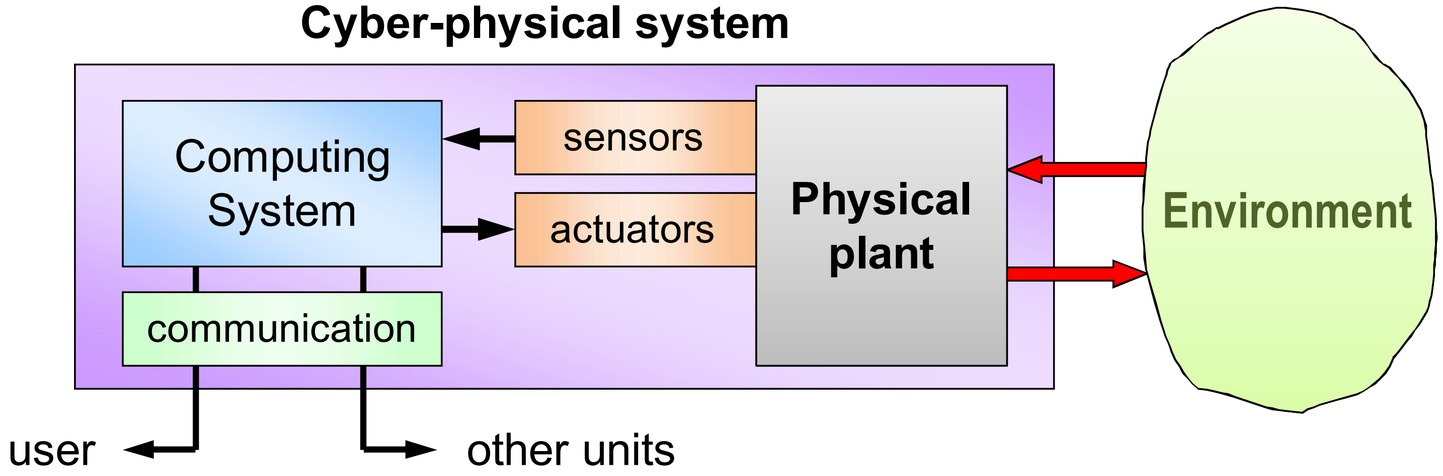
eye

brain



Cyber-physical systems

Cyber-physical systems (**CPS**) are computing systems that are tightly connected to a physical plant:

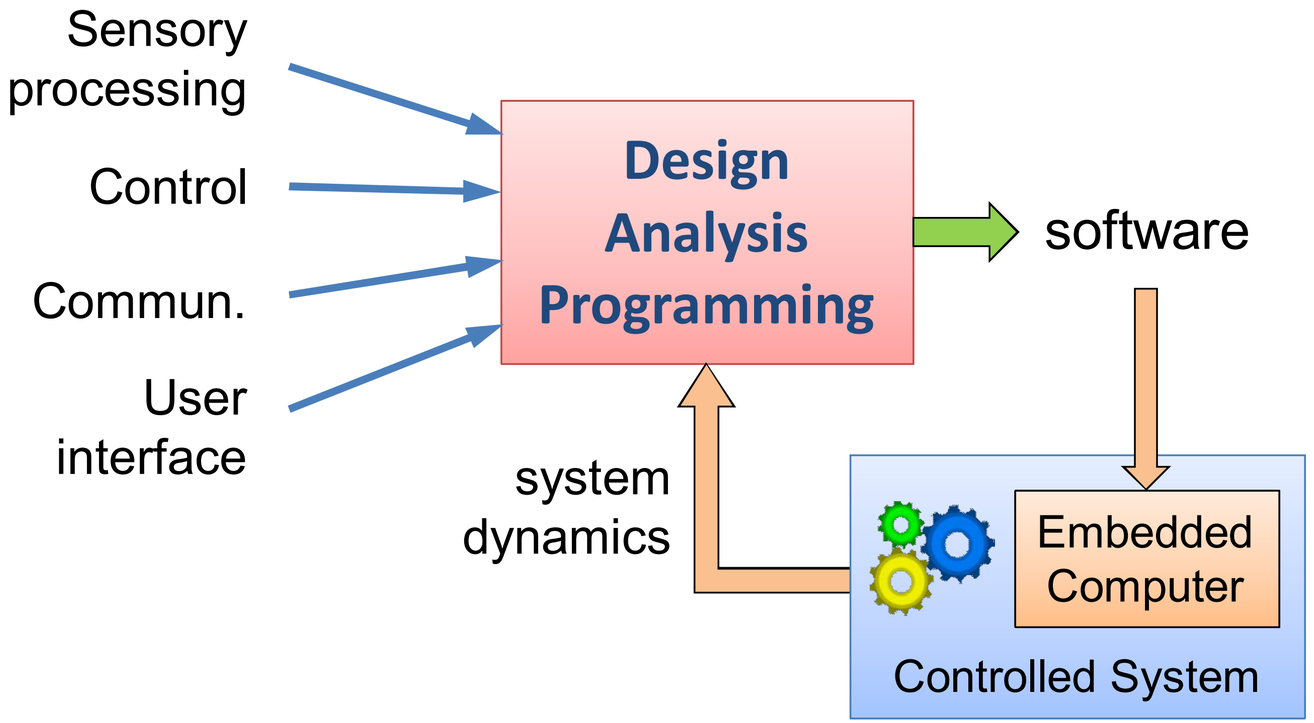


Typical CPS functions:

- Resource management
- Control
- Interaction with the user
- Interaction with the physical plant and the environment

This is a very crucial aspect affecting all other functions!

Cyber-physical systems



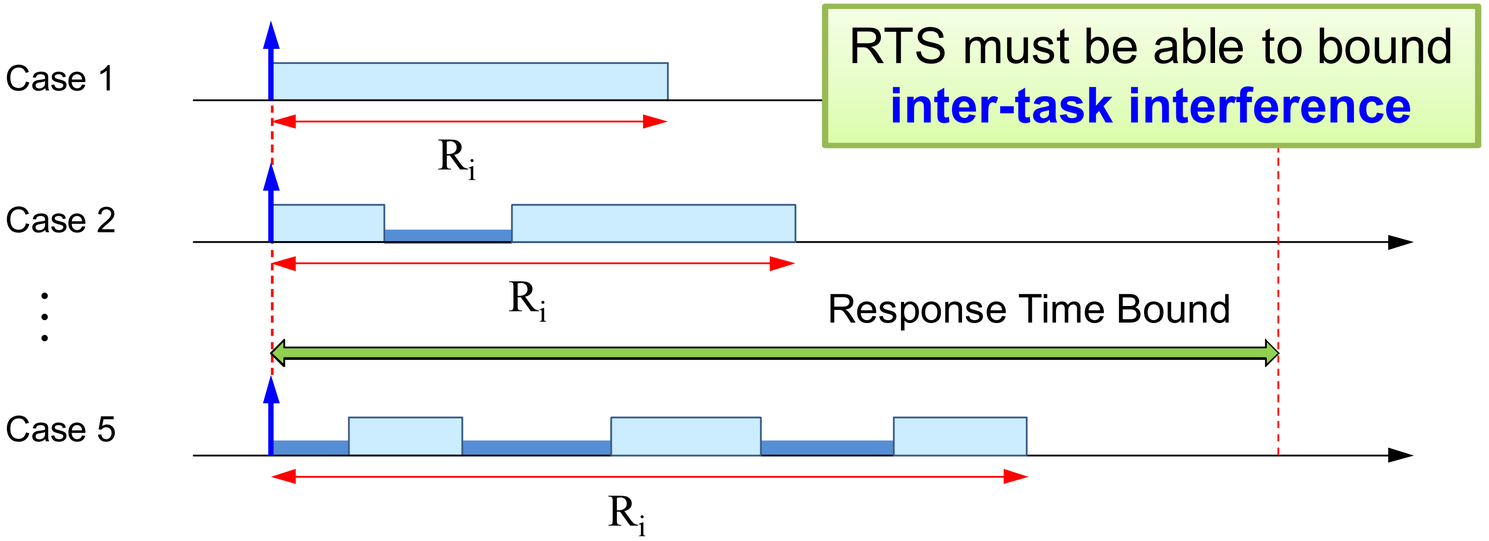
The interaction with the physical part imposes timing constraints on the computation, therefore

most cyber-physical systems are real-time systems.

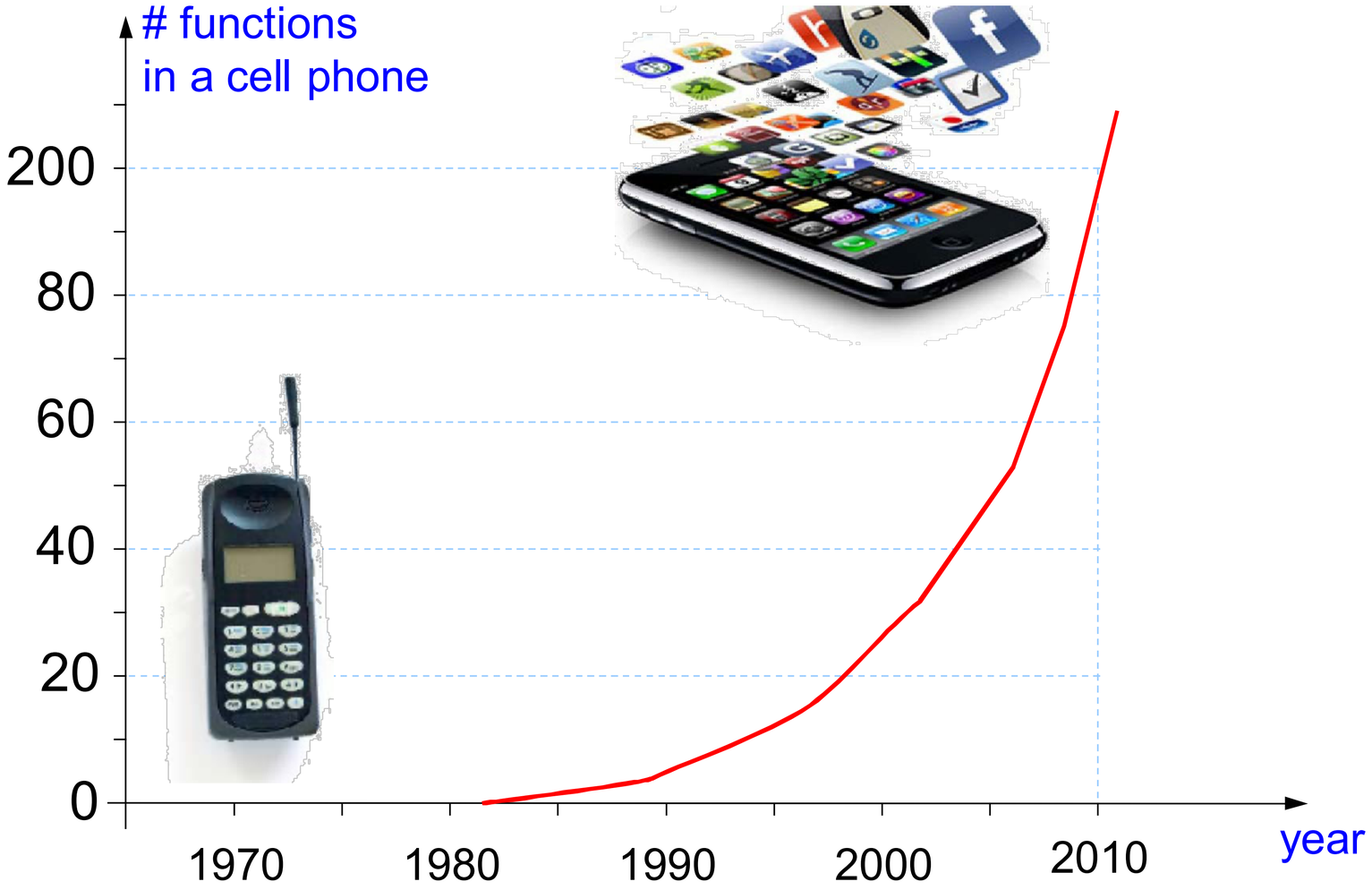
Real-Time Systems

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

They must provide bounded response times in all possible scenarios. → ∀ event combination

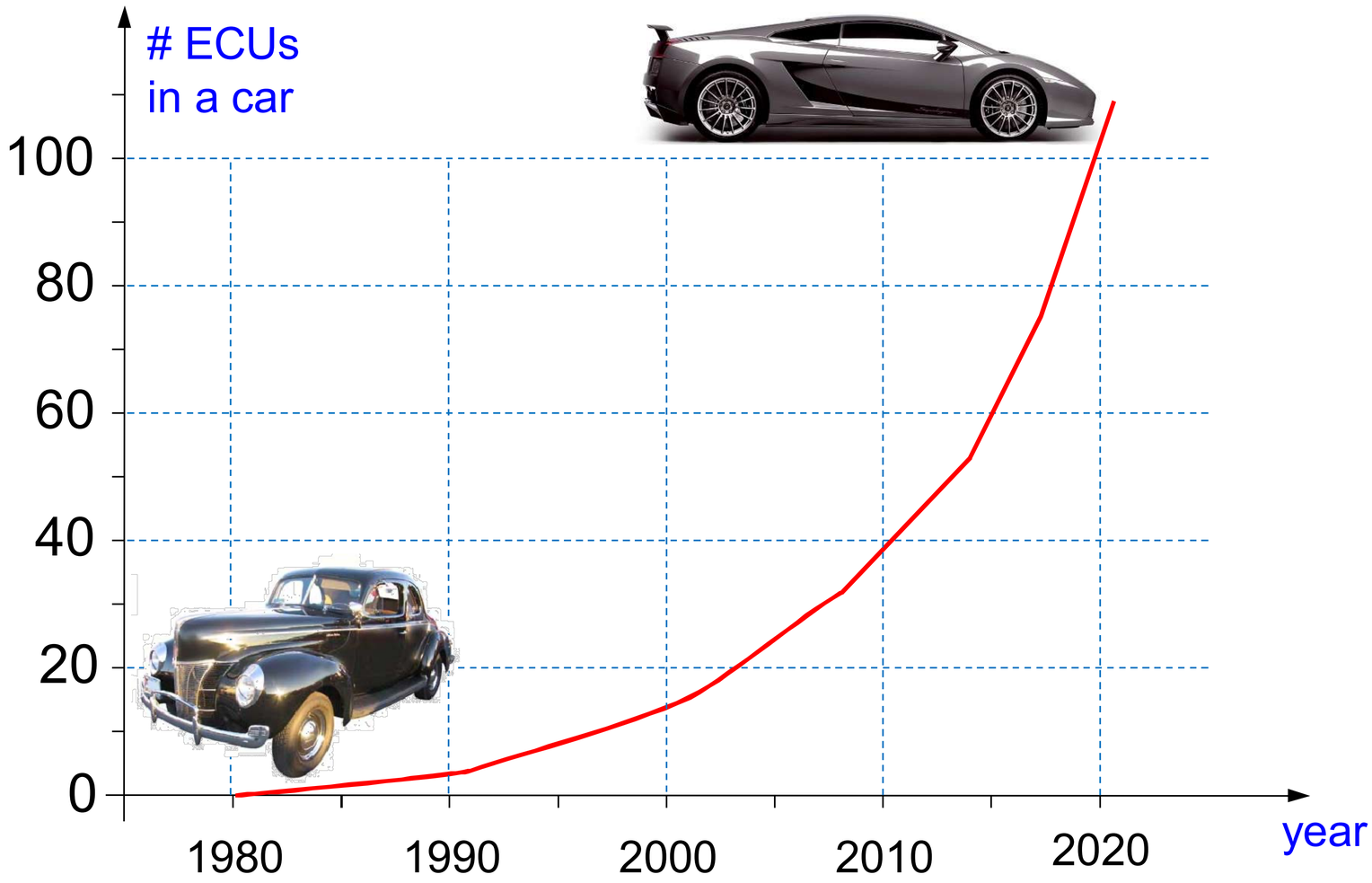


Increasing complexity

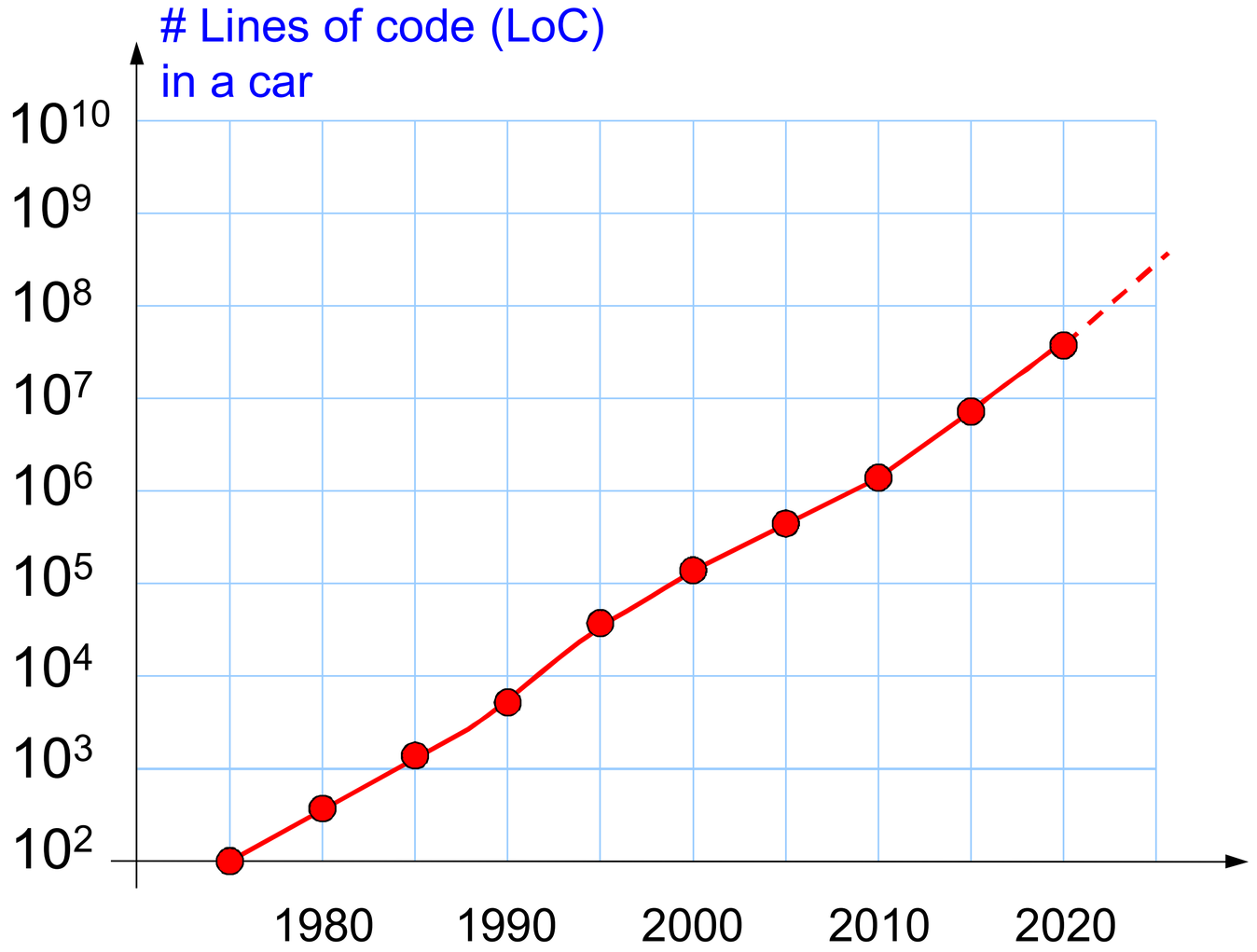


ECU growth in a car

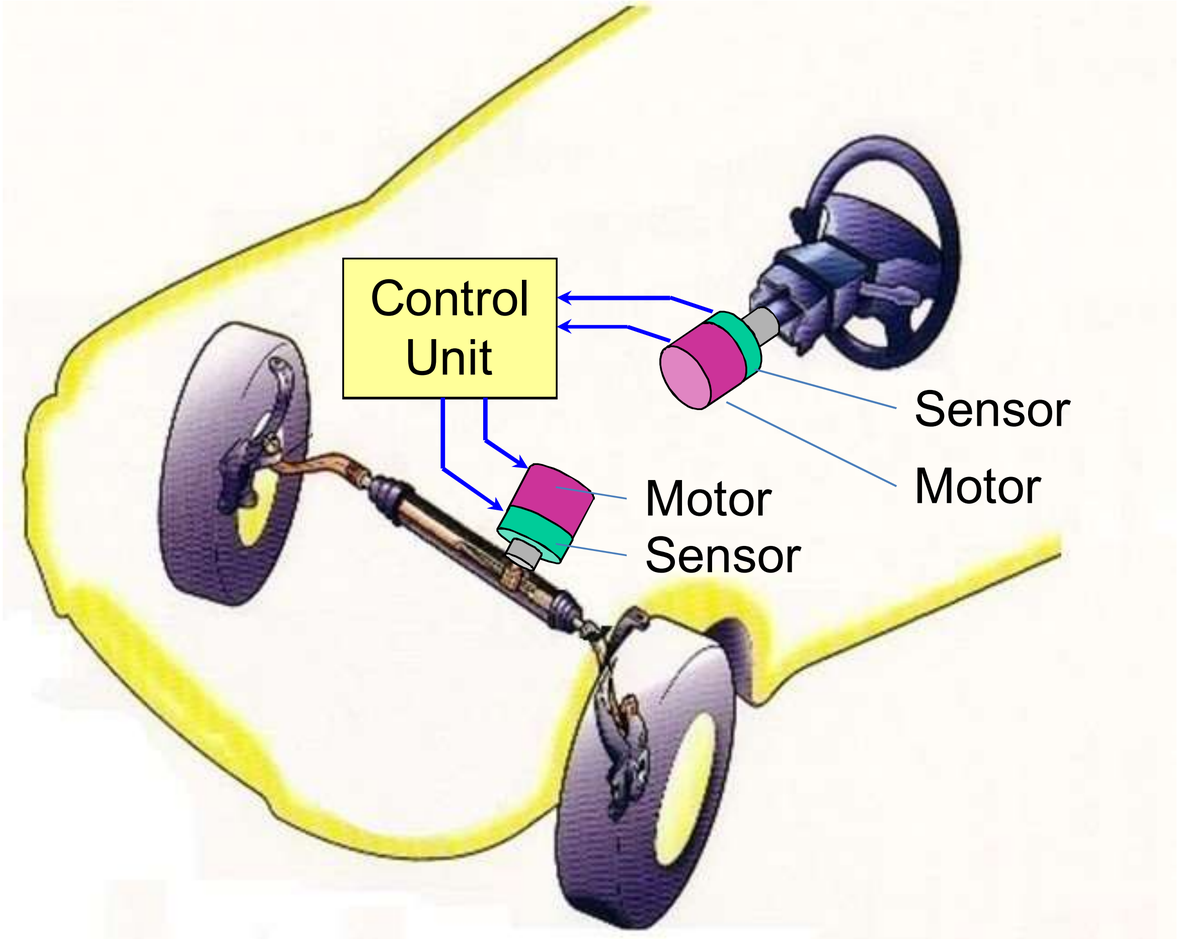
ECU: Electronic Control Unit



Software evolution in a car



Steer by Wire



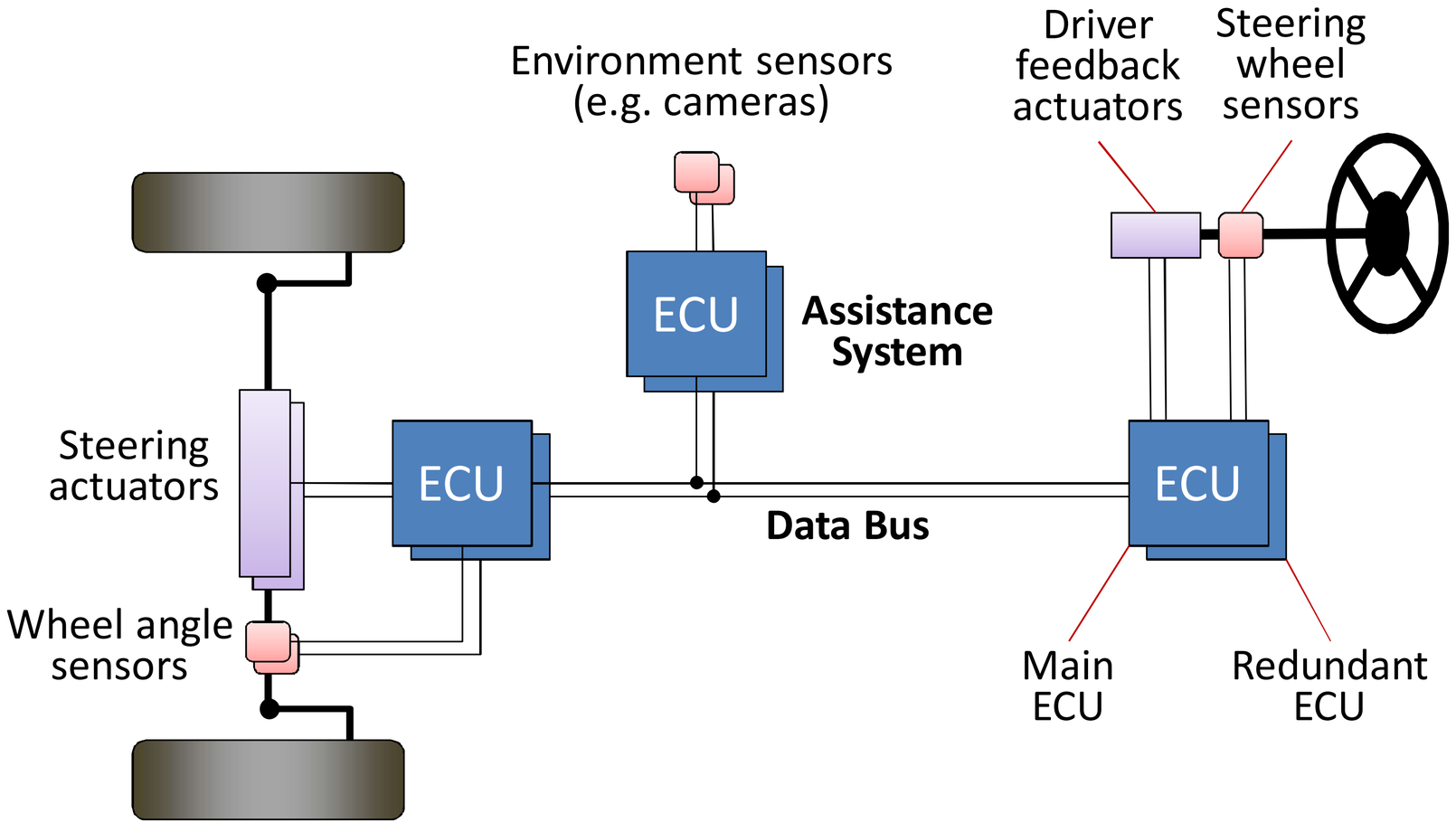
Steer by Wire

Advantages

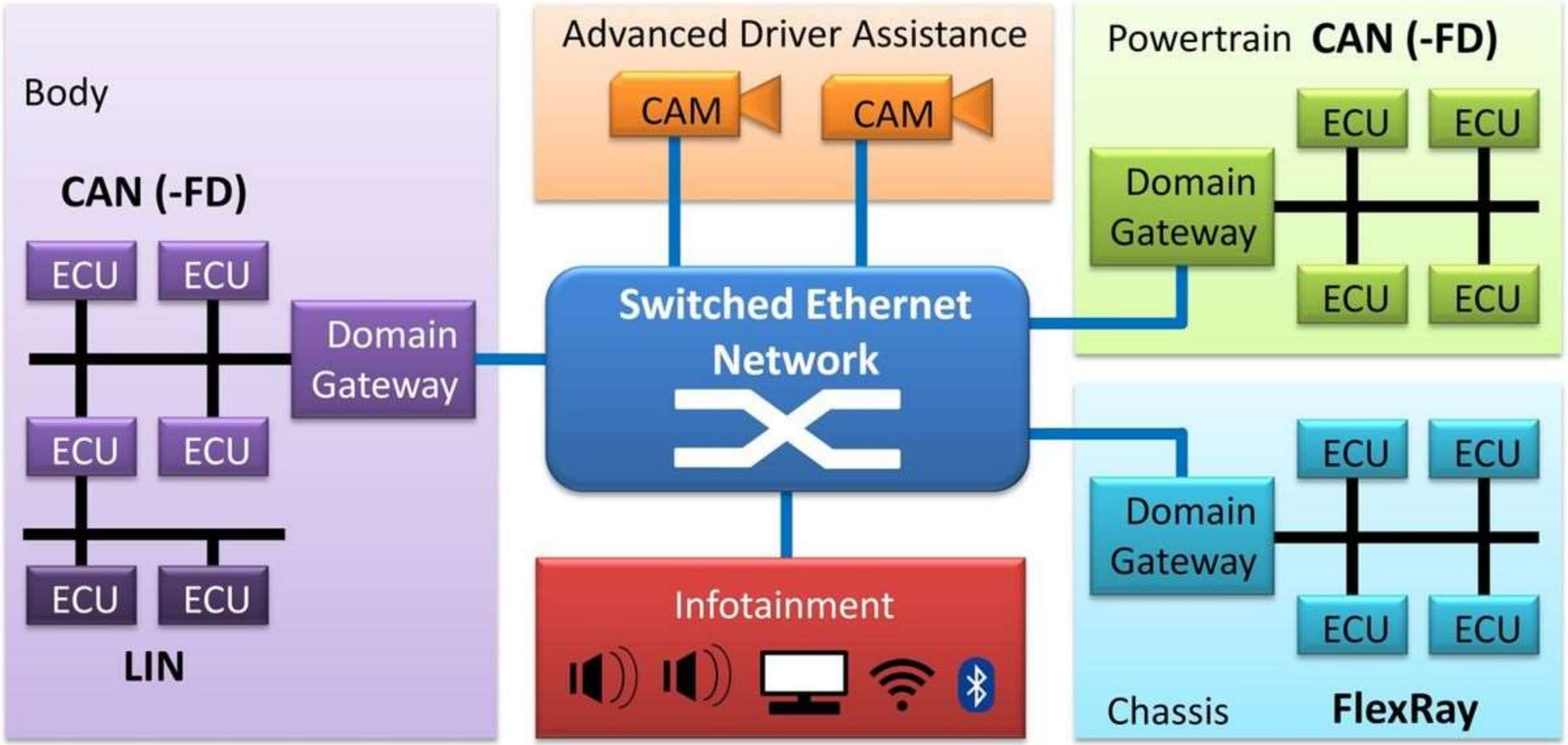
1. Placing the steering wheel in any position
2. Record the driving style of the driver
3. Remote control
4. Develop self-parking functions
5. Autonomous driving

Software in a car

Distributed architecture



Heterogeneous networks

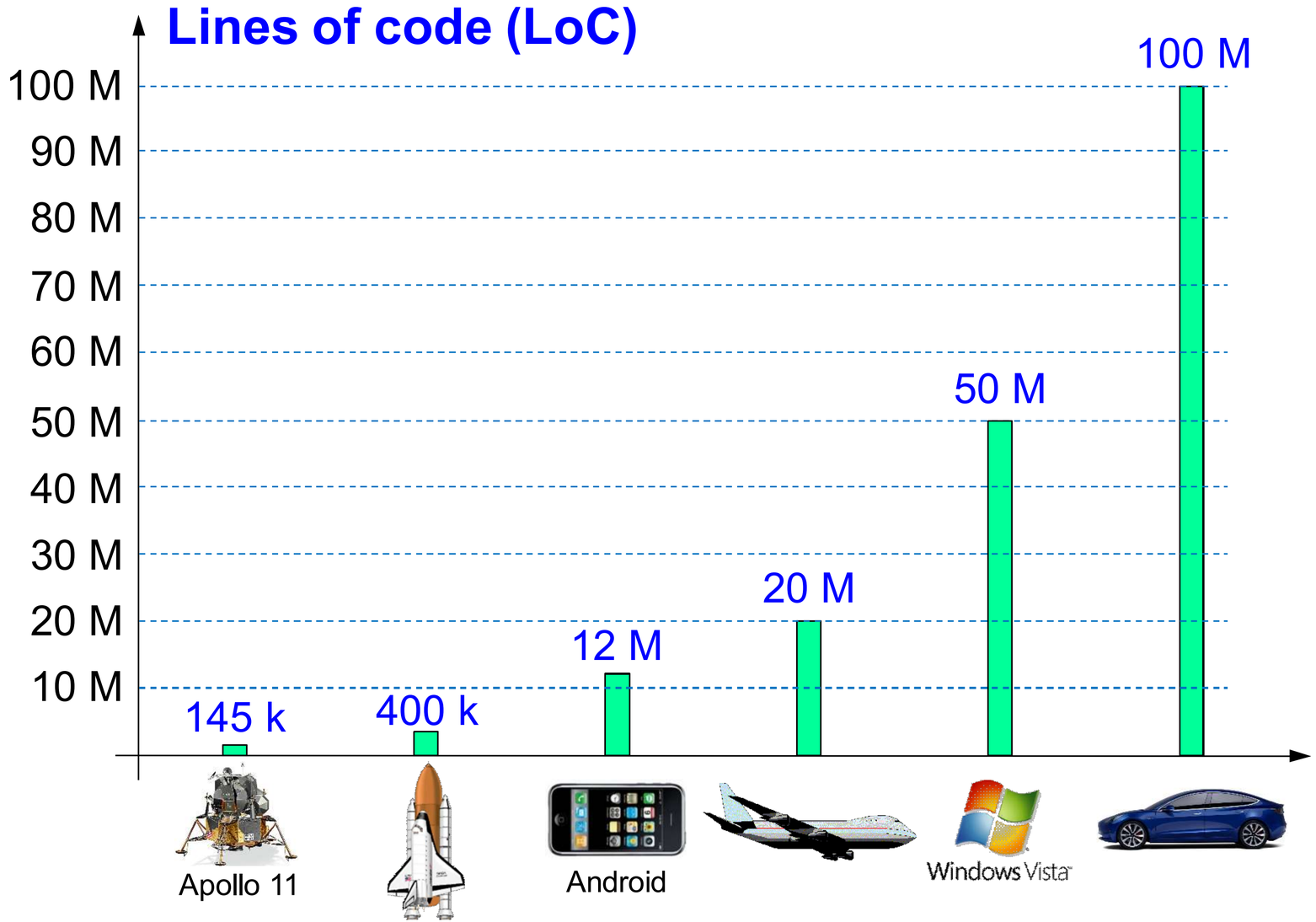


Software in a car

Car software controls almost everything:

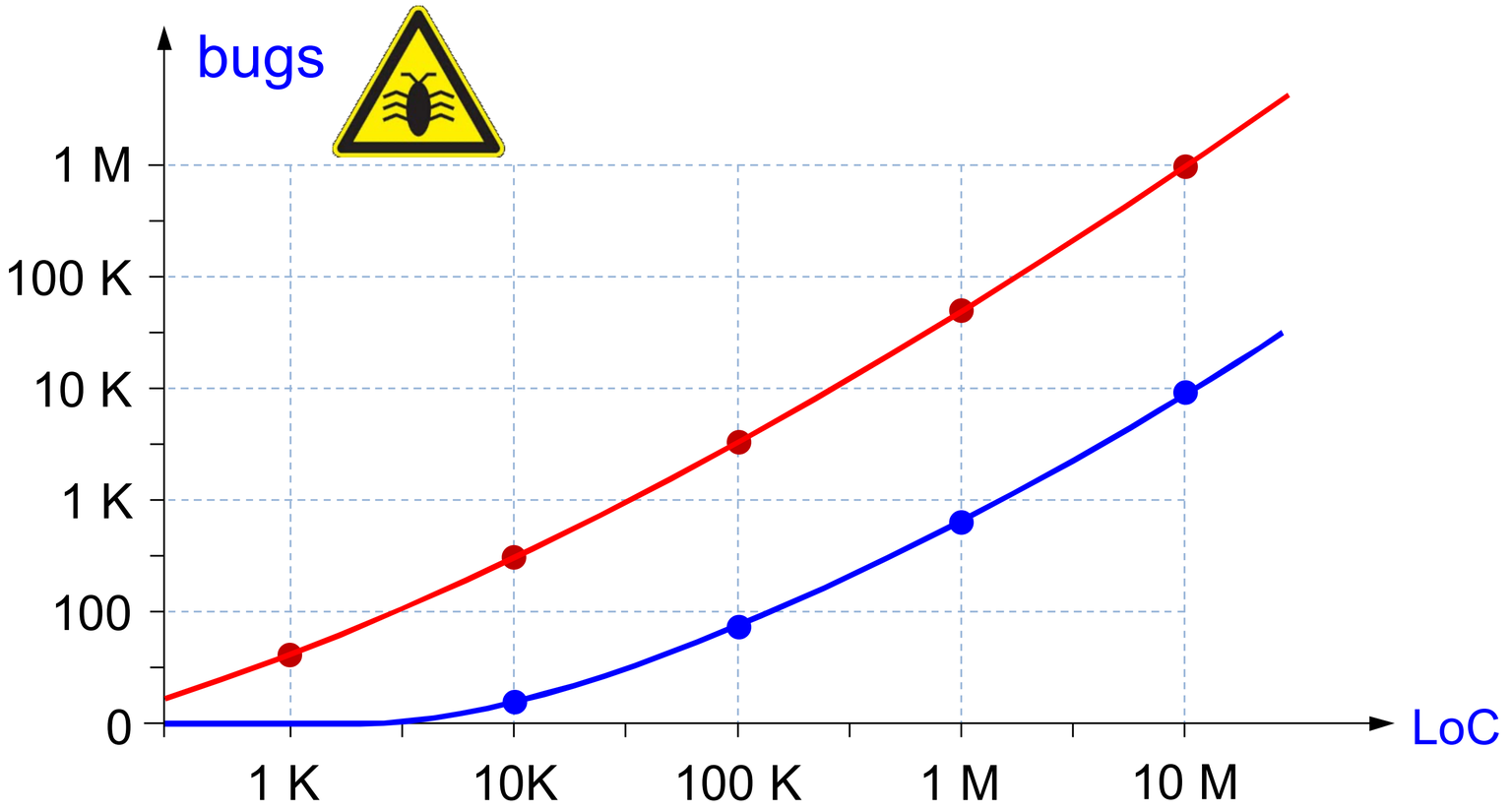
- Engine: ignition, fuel pressure, water temperature, valve control, gear control,
- Dashboard: engine status, message display, alarms
- Diagnostic: failure signaling and prediction
- Safety: ABS, ESC, EAL, CBC, TCS
- Assistance: power steering, navigation, sleep sensors, parking, night vision, collision detection
- Comfort: fan control, heating, air conditioning, music, active light control, noise control & cancellation, regulations: steer/lights/sits/mirrors…

Comparing Software Complexity

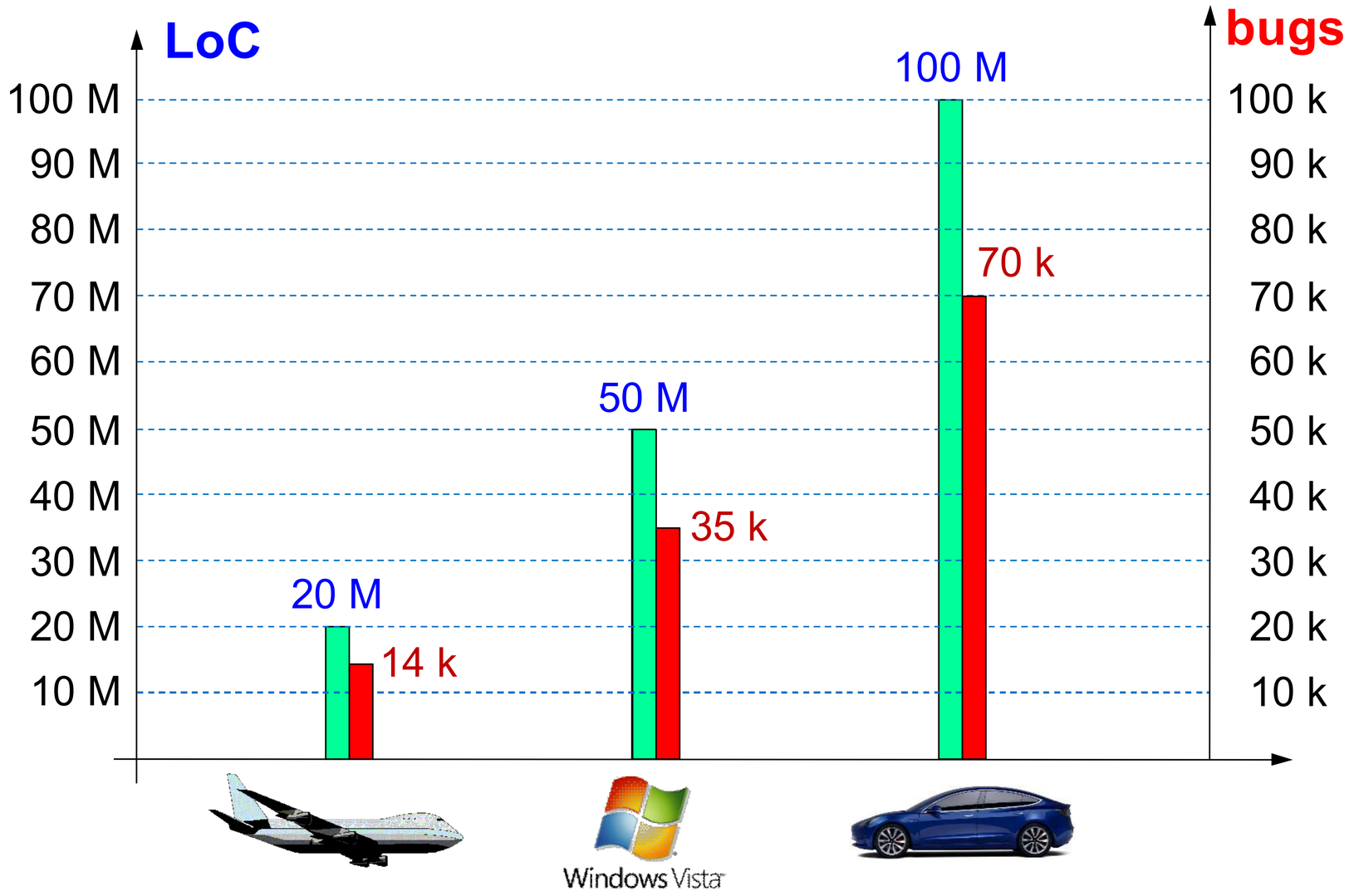


Software complexity and bugs

~70 bugs per 1000 LoC → (after testing, ÷100) → 0.5÷1 bugs per 1000 LoC



Software complexity and bugs



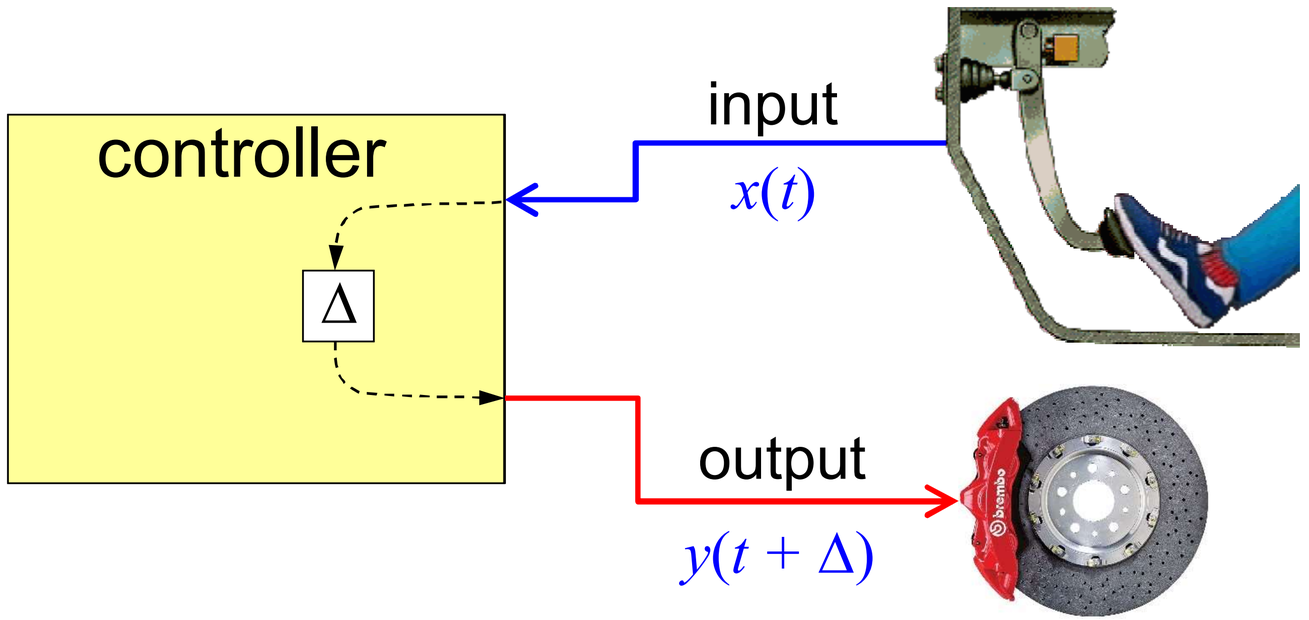
Software reliability

When aircrafts depend on a program with 20 million instructions, reliability is a primary objective.

20 M instructions
(**14,000 bugs**)

Software reliability

In RTS, reliability not only depends on the correctness of the output values, but also on when they are delivered:



A correct action executed too late can be useless or even dangerous.

Typical misconception

~~A real-time system is a computing system able to perform computation as fast as possible.~~

Real-time **does not mean** fast.

- Speed is always relative to a specific environment.
- Running fast is good, but does not necessarily guarantee a correct behavior.

Given an arbitrary computer speed, we must guarantee that timing constraints will be met in all possible scenarios.

Testing is **NOT** sufficient.

Fast vs. Predictable

Real Time does not mean fast, but predictable.

Fast and Predictable systems have different design objectives

Fast ↔ Predictable

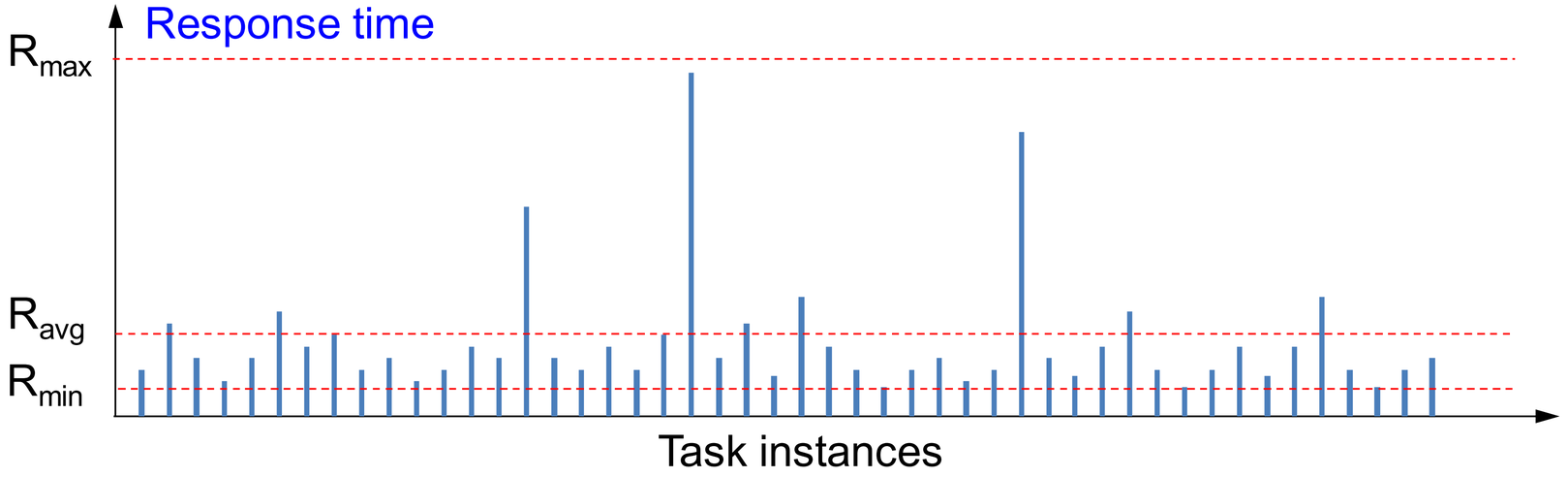
minimize the
average response time
of a task set

guarantee that the
worst-case response time
of each individual task does
not exceed a given bound

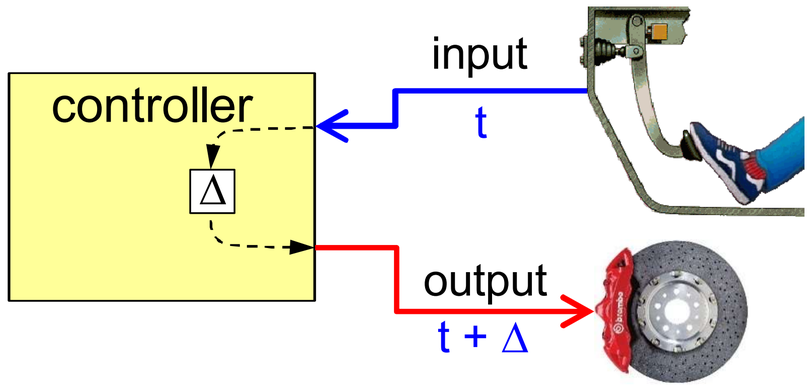
Don't trust the average
when you have to guarantee
individual performance

Modern architectures

Modern architectures are built to be fast in the average, but response times can have large fluctuations:

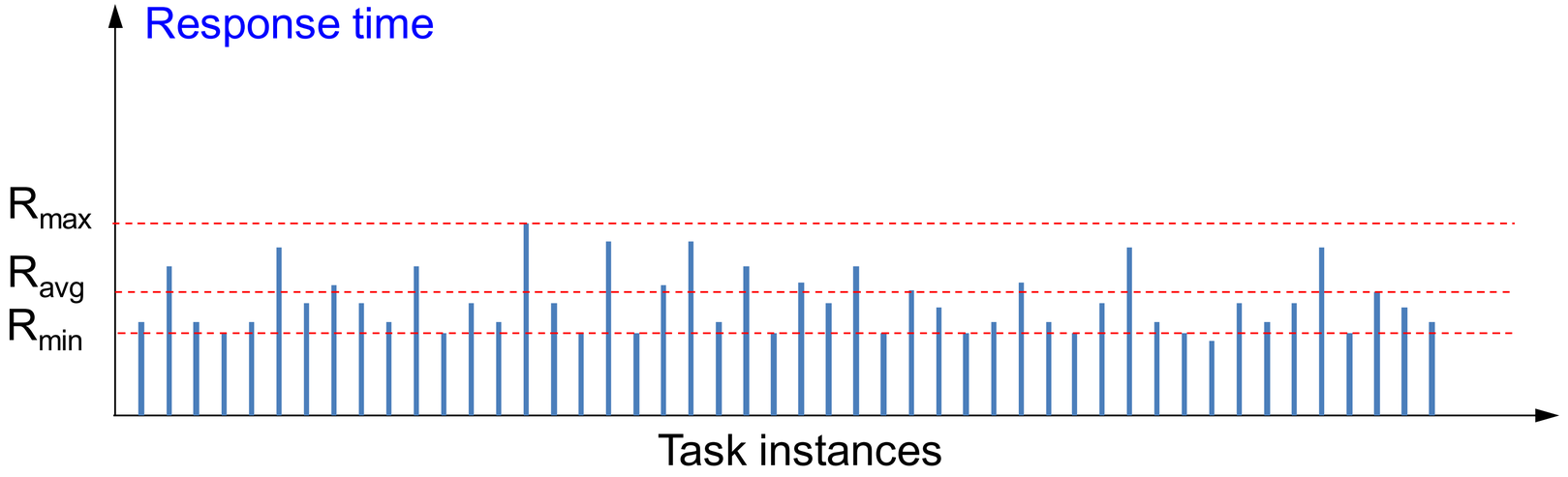


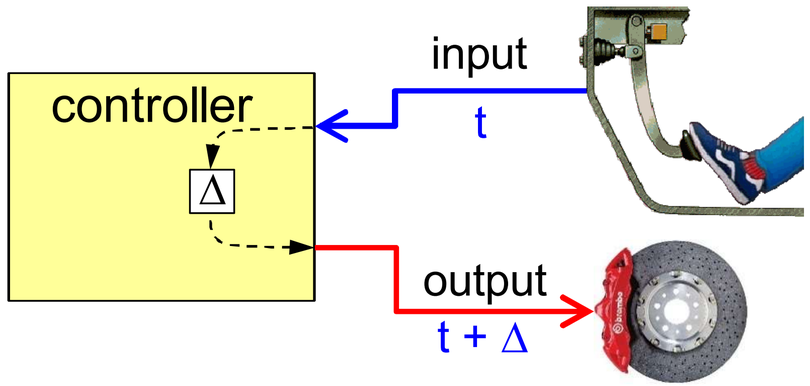
Can you imagine what happens if this system is used for the brakes?



Modern architectures

A slower system having higher average response time but smaller fluctuations would be much more preferable:

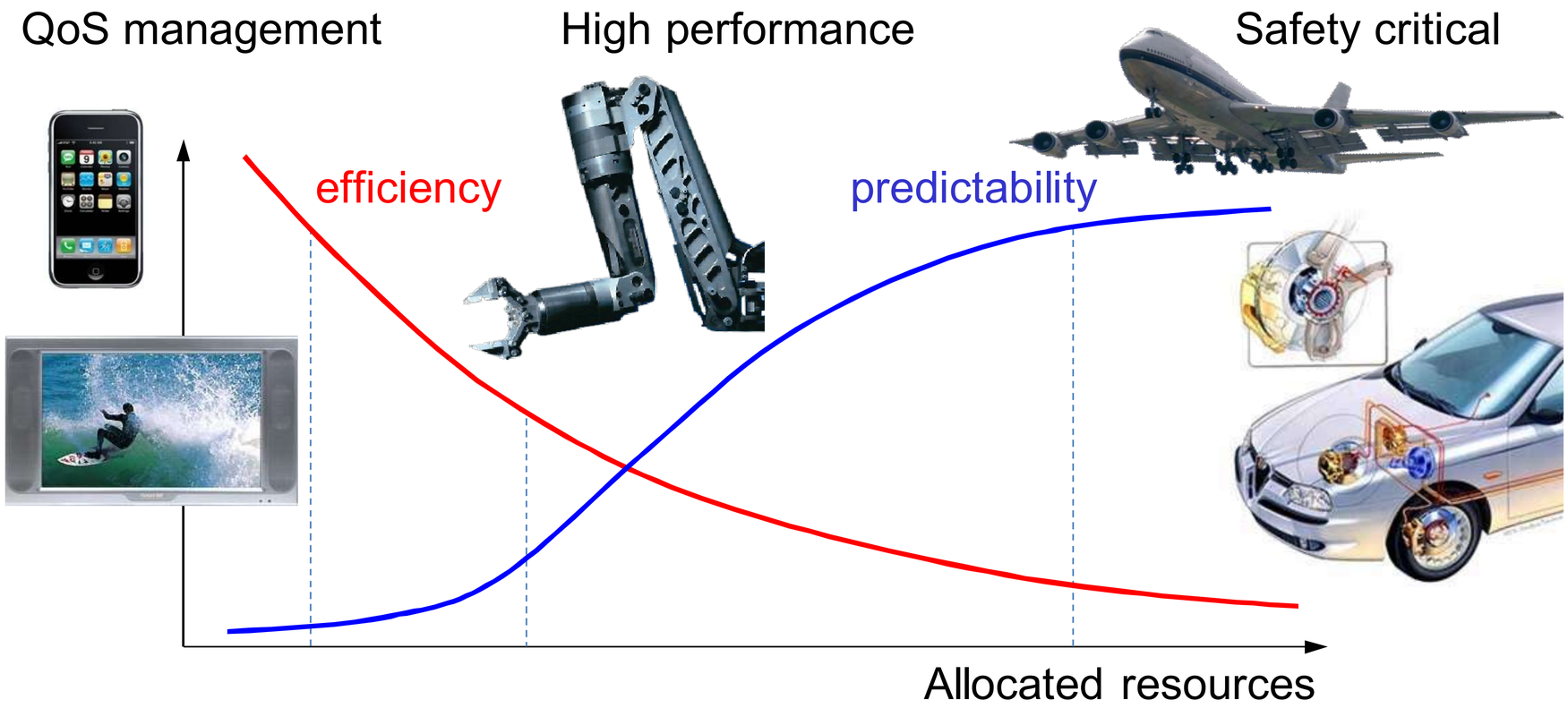




Sources of non determinism

- **Architecture**
  - cache, pipelining, interrupts, DMA
- **Operating system**
  - scheduling, synchronization, communication
- **Language**
  - lack of explicit support for time
- **Design methodologies**
  - lack of analysis and verification techniques

Predictability vs. Efficiency

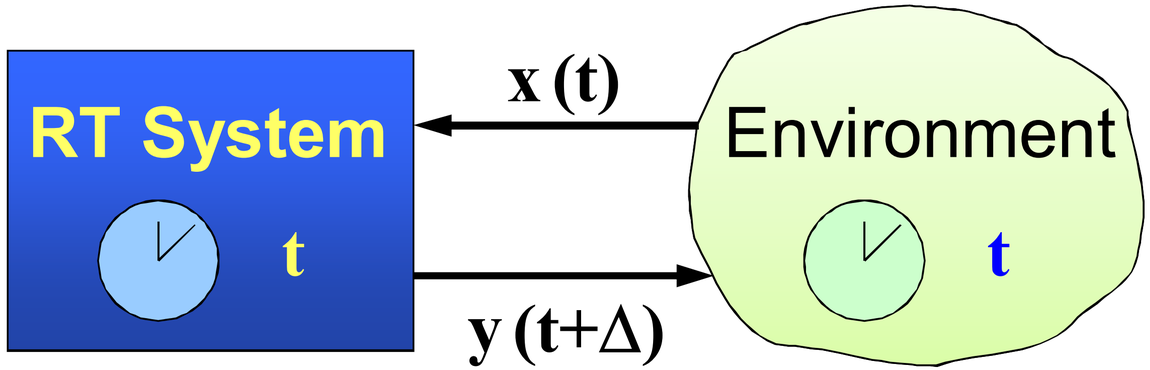


What's special in CPS?

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

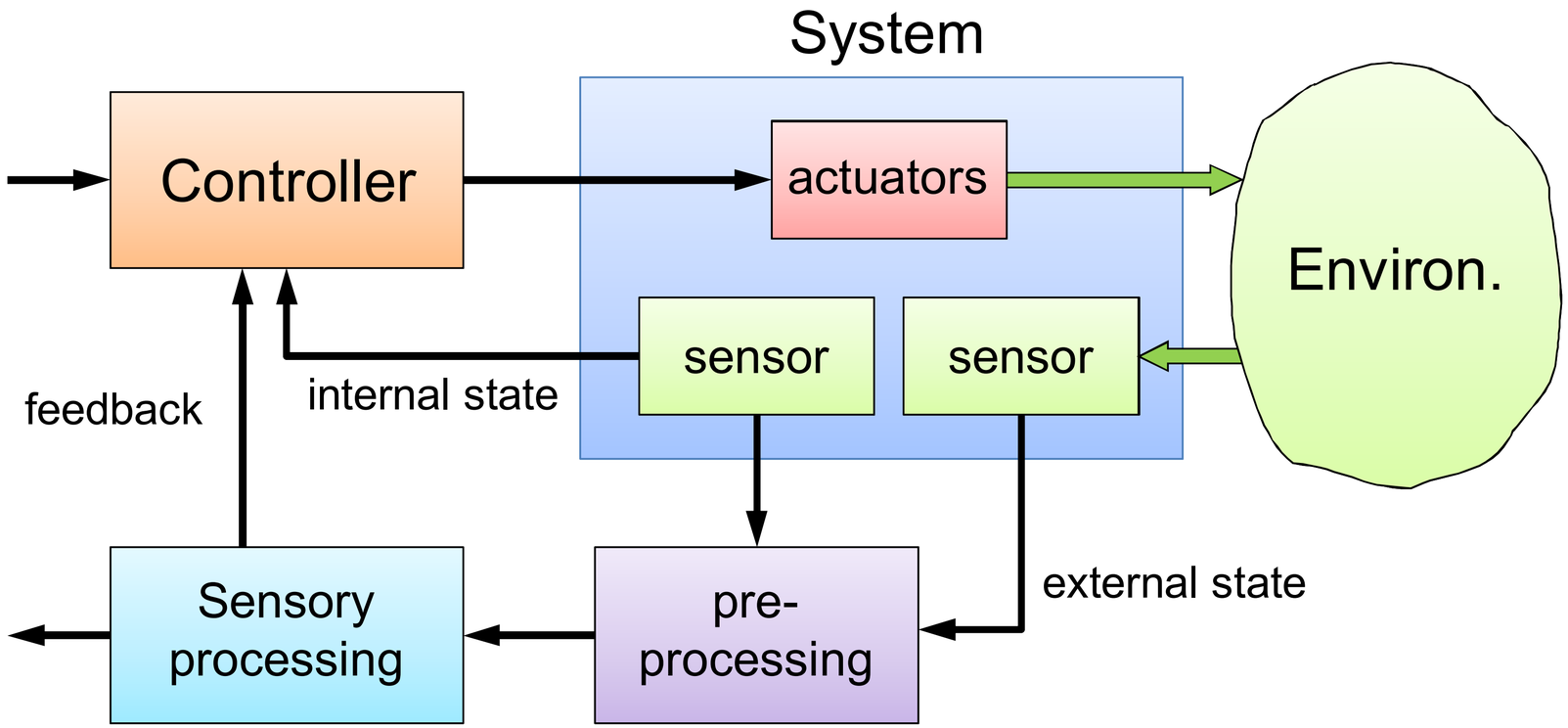
Definition: Real-Time System

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.



REAL means that the system time must proceed at the same speed as the time in the environment.

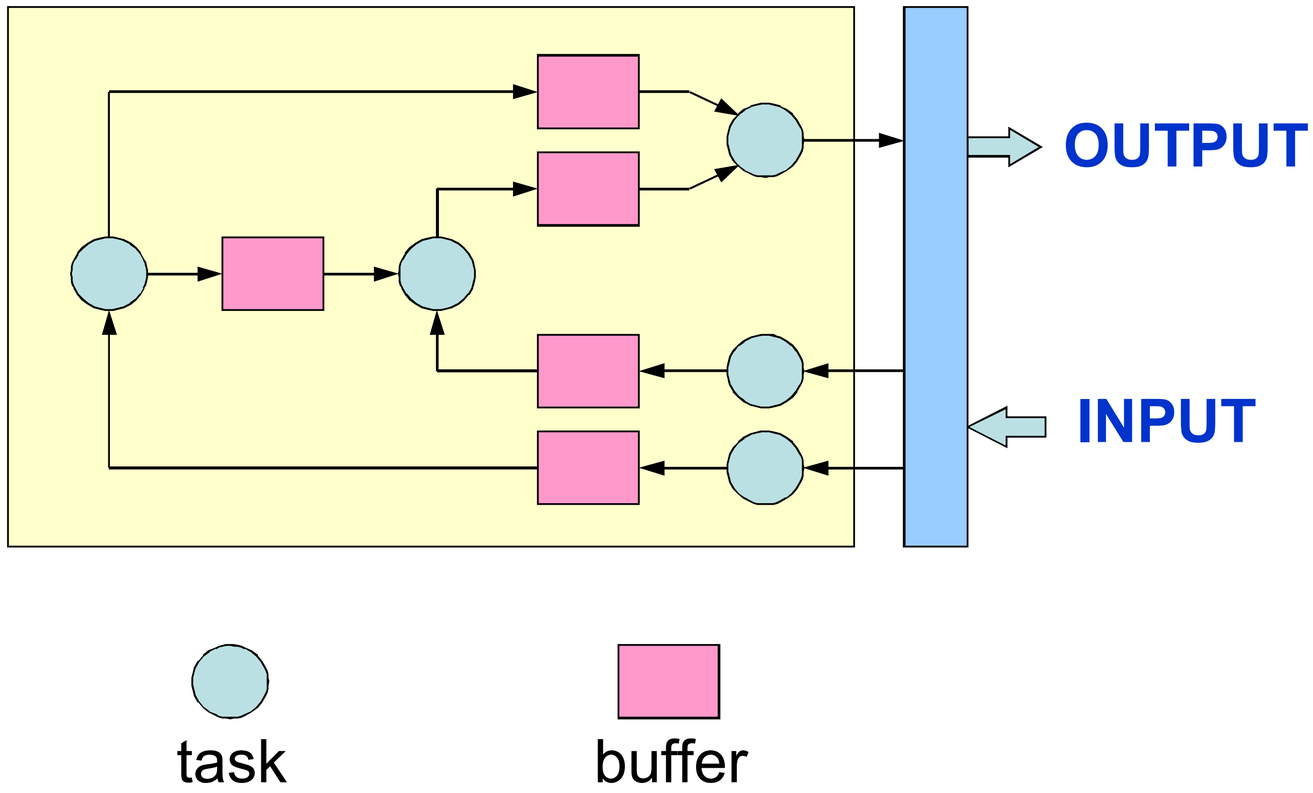
A typical control system



Other activities

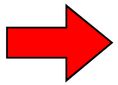
filtering, classification, data fusion, recognition, planning

Software view



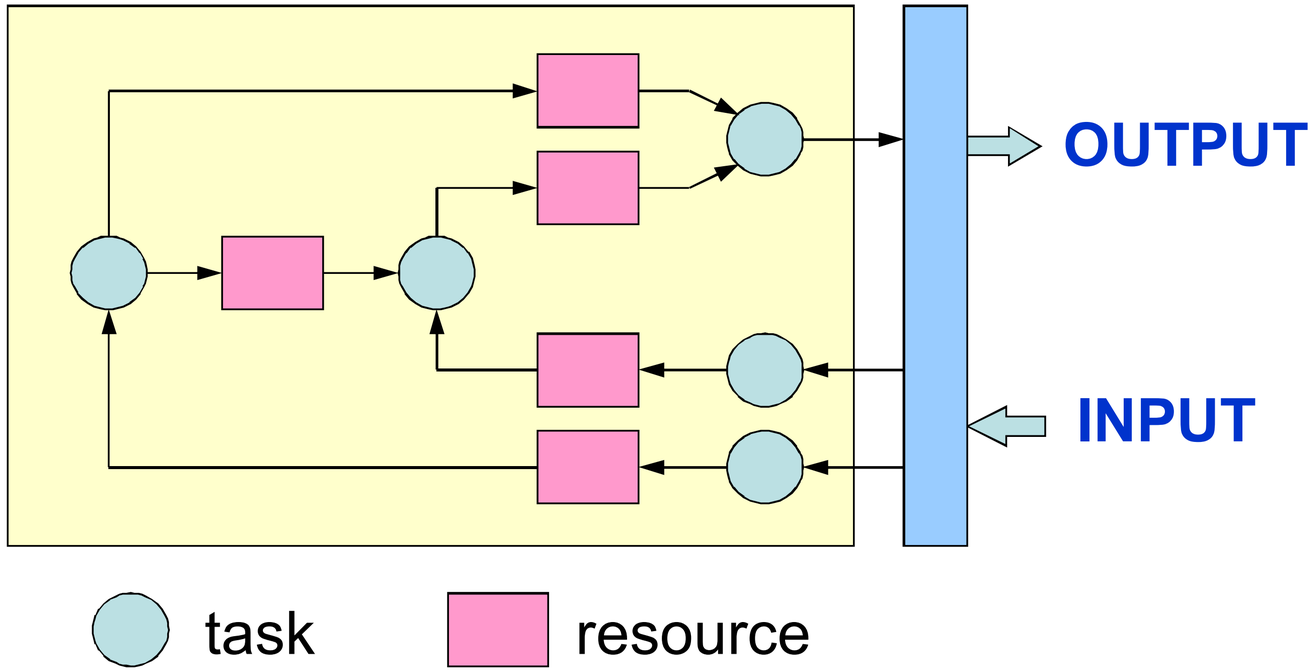
Implications

- 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.

Software structure



- The RTOS is responsible for providing proper mechanisms for a predictable interaction between tasks and resources.
- We are responsible for analyzing the application and guaranteeing deadlines in worst-case scenarios.

RTOS responsibilities

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).

Lessons learned

- Tests, although necessary, allow only a partial verification of system's behavior.
- Predictability must be improved at the level of the operating system.
- The system must be designed to be fault-tolerant and handle overload conditions.
- Critical systems must be designed under pessimistic assumptions.

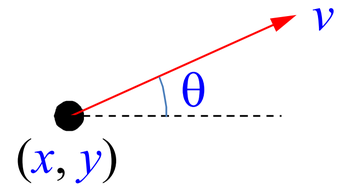
Modeling
Real-Time Activities

What is a model?

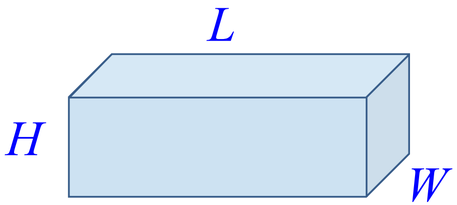
A model is a simplified representation of something. It captures not all attributes of the represented thing, but rather only those that are relevant for a specific purpose.

Example

trajectory → model 1



encumbrance → model 2



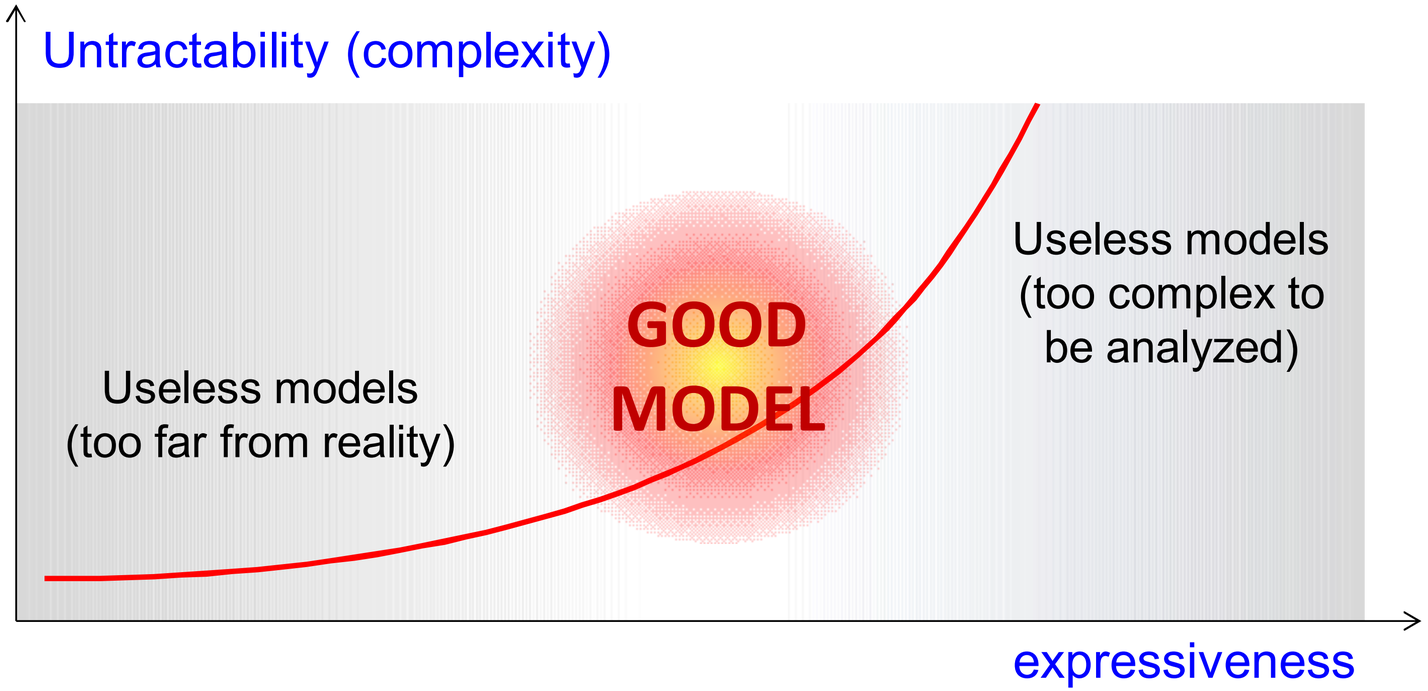
performance → model 3

- Max speed
- Max acceleration
- Weight
- Fuel consumption
- CX

What is a good model?

- It should be expressive (an accurate representation of reality)
- It should be tractable (provide results in a bounded time)

Unfortunately, expressiveness and tractability do not get along very well



Important aspects

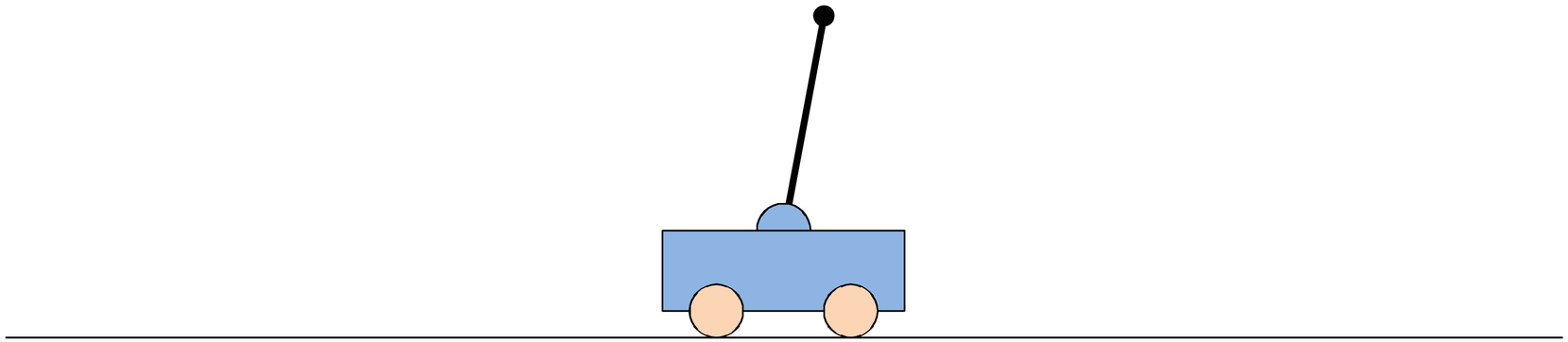
Building a model implies:

- clearly identifying the assumptions you need to simplify reality (but don't simplify too much);
- defining the variables that characterize the model.
- defining the system interface (variables are exposed to the user);
- defining the metrics for evaluating the outputs of your system and its performance.

Types of variables

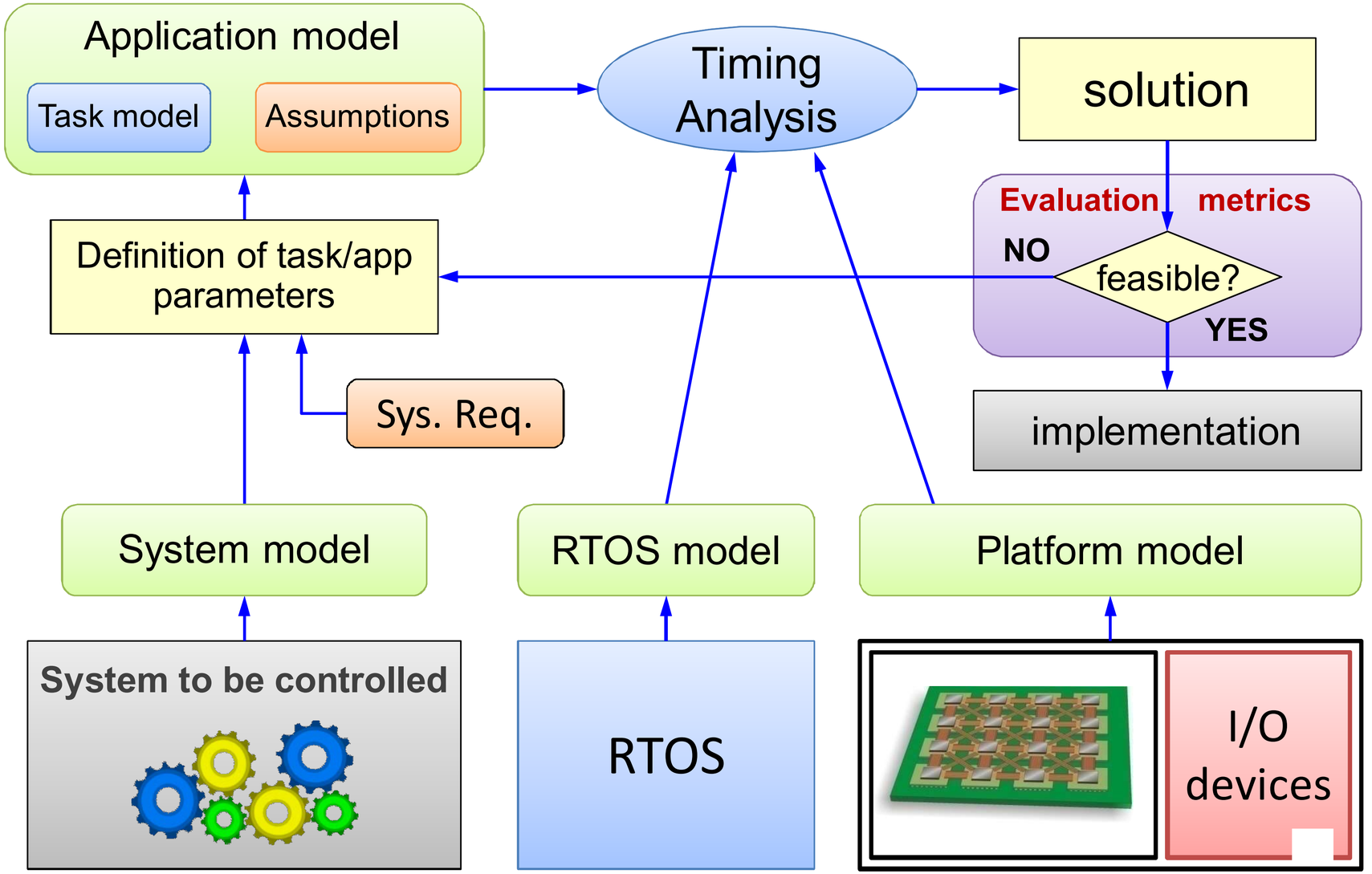
- Parameters (variables you don’t want to change);
- Input variables (commands given by the user/controller)
- Design variables (variables you want to identify to apply your control actions);
- State variables (variables describing the system state and behavior);
- Output variables (variables you want to measure to evaluate the performance of your method).

Example



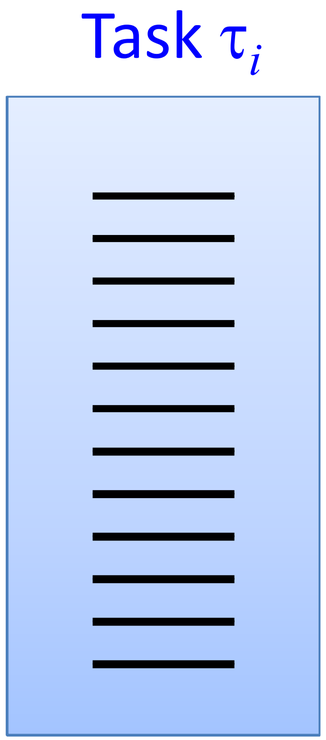
- Parameters: Pole length/mass, cart mass
- Input variables: Force applied to the cart
- Design variables: Control parameters (K_P, K_I, K_D)
- State variables: Position/speed of the cart and pole
- Output variables: Pole angle

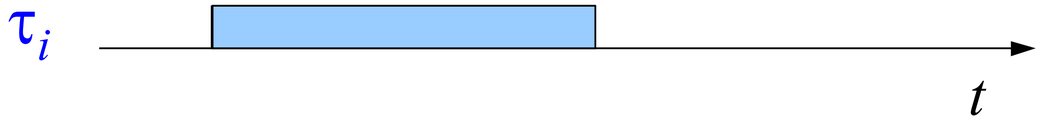
Modeling computation



Definition: Task

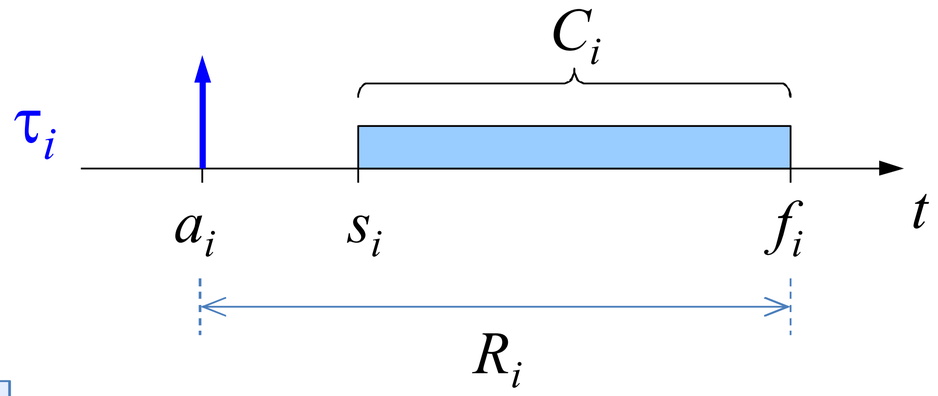
A **task** (or **thread**) is a sequence of instructions that in the absence of other activities is continuously executed by the processor until completion.

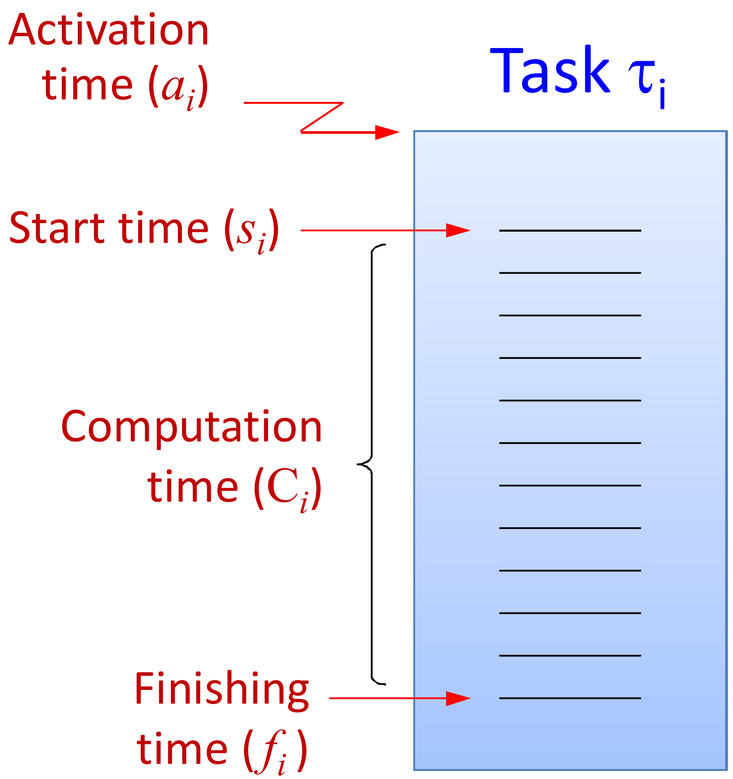




What are the important variables that characterize a computation if we want to perform a timing analysis?

Task parameters





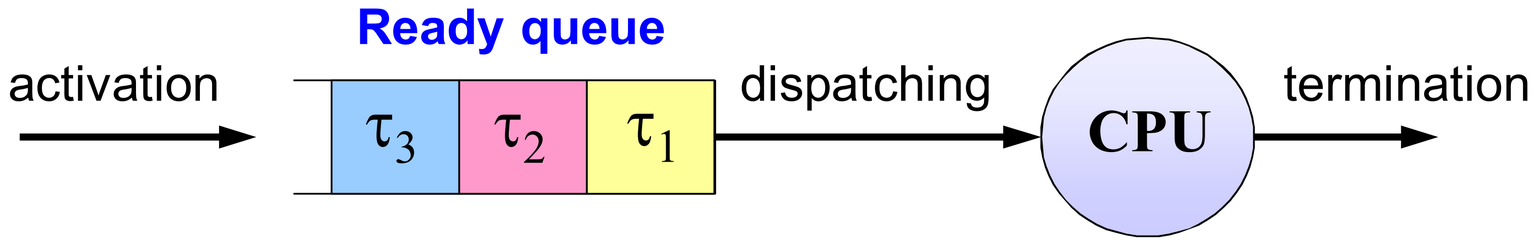
The interval $f_i - a_i$ is referred to as the task response time R_i

A task τ_i is also assigned a priority P_i reflecting its importance relative to the other tasks.

Ready queue

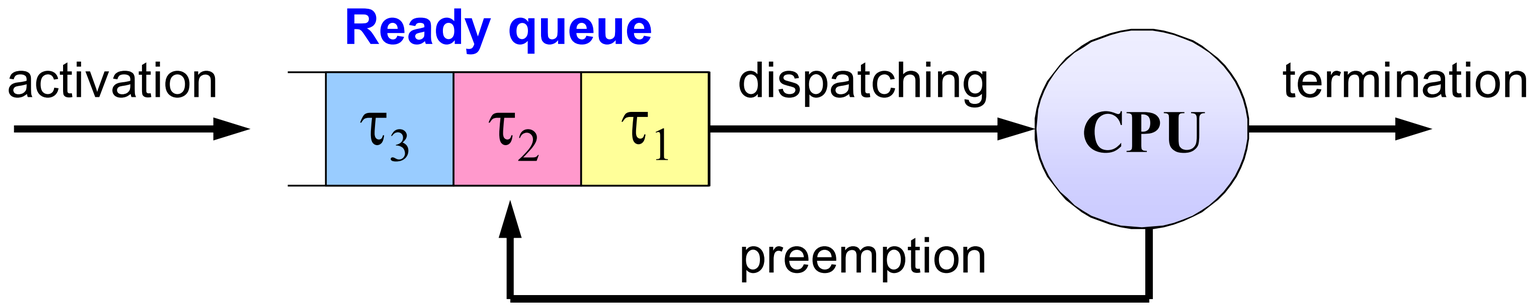
In a concurrent system, more tasks can be simultaneously active, but only one can be in execution (running).

- An active task that is not in execution is said to be ready.
- Ready tasks are kept in a ready queue, managed by a scheduling policy.
- The processor is assigned to the first task in the queue through a dispatching operation.



Preemption

It is a kernel mechanism that allows to suspend the execution of the running task in favor of a more important task. The suspended task goes back in the ready queue.



- Preemption enhances concurrency and allows reducing the response times of high priority tasks.
- It can be disabled (completely or temporarily) to ensure the consistency of certain critical operations.

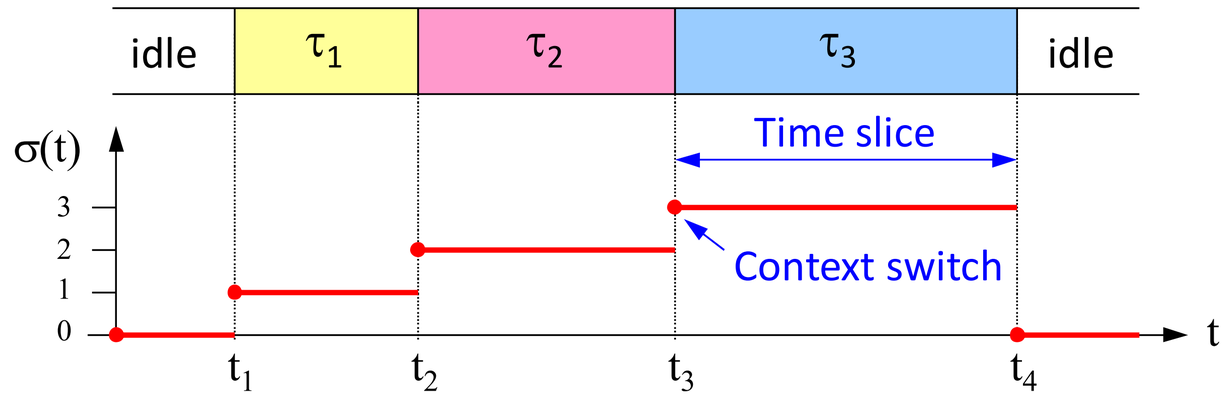
Schedule

A schedule is a specific allocation of tasks to the processor, which determines the corresponding execution sequence.

Formally, given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a schedule is a function $\sigma: R^+ \rightarrow N$ that associates an integer k to each interval of time $[t, t+1)$ with the following meaning:

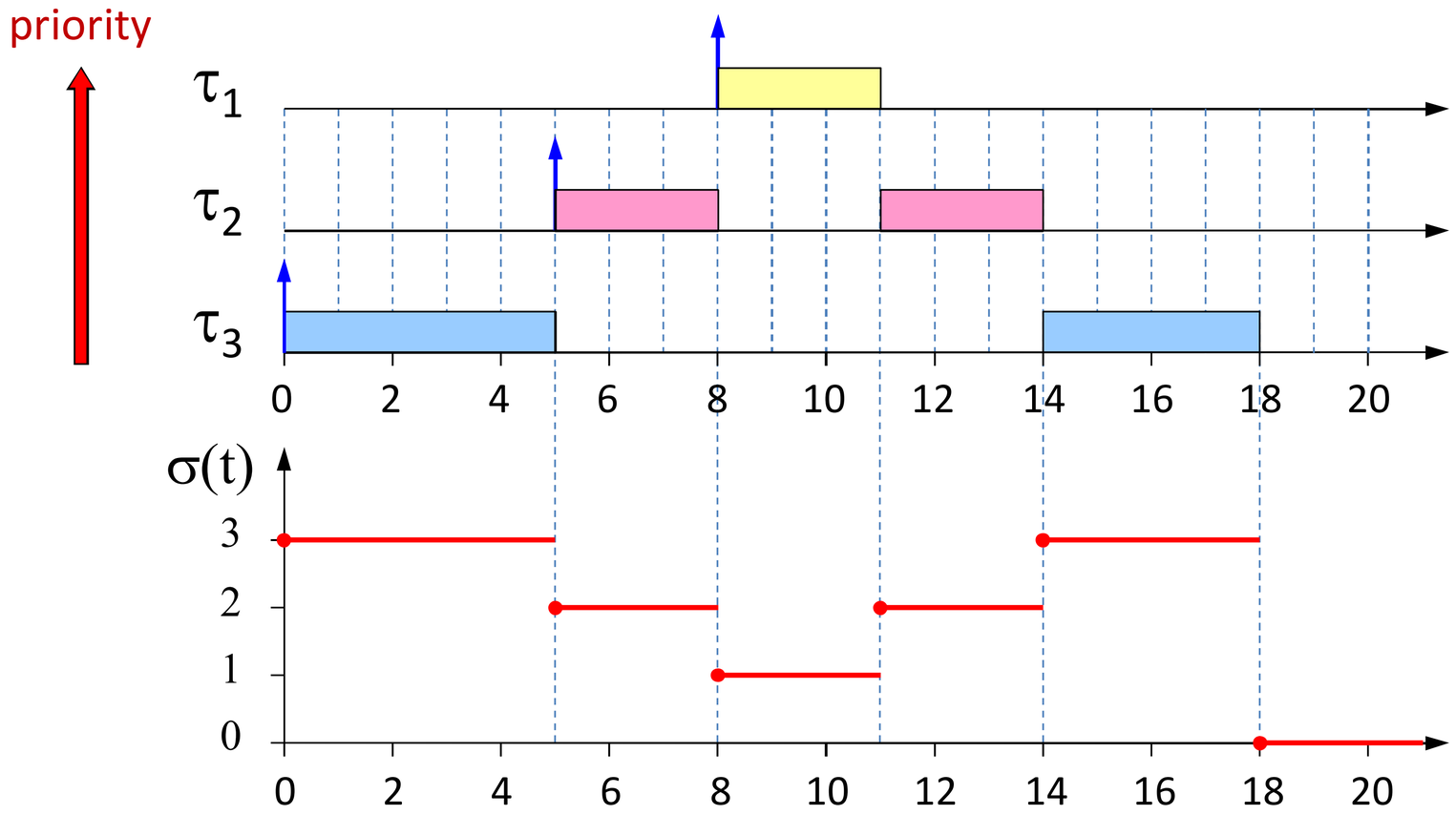
$k = 0$ ⟹ in $[t, t+1)$ the processor is IDLE

$k > 0$ ⟹ in $[t, t+1)$ the processor executes τ_k

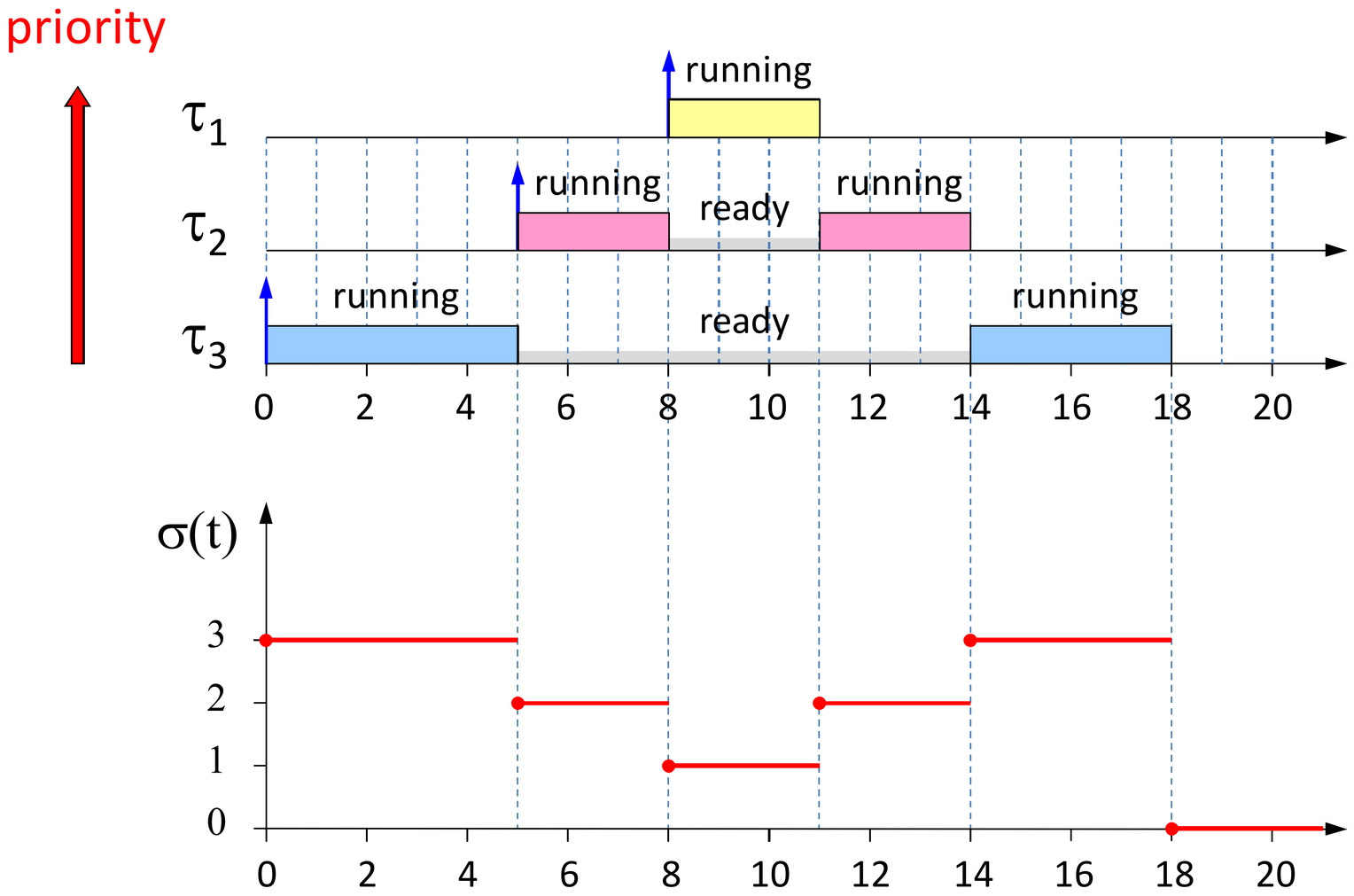


Preemptive schedule

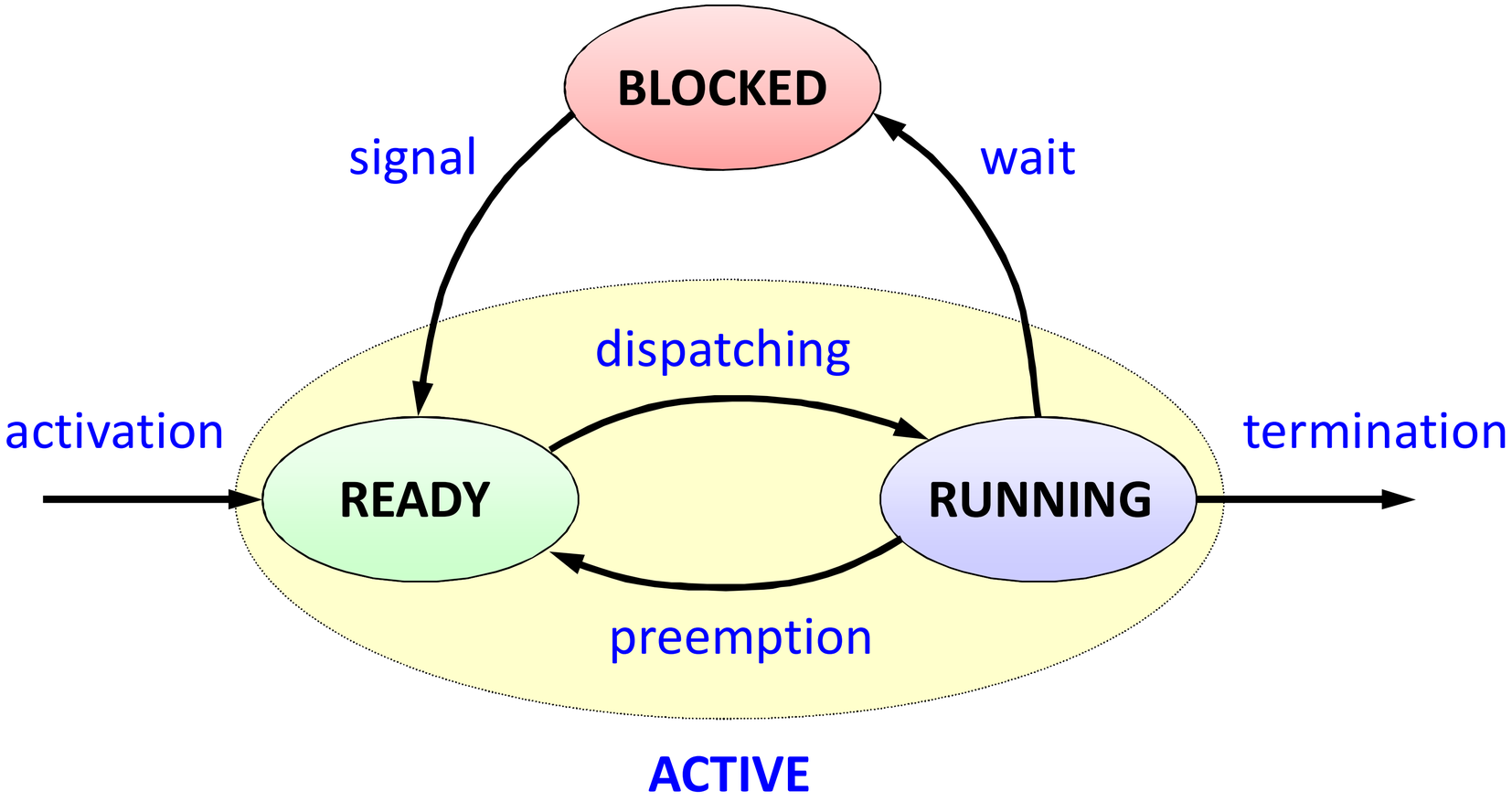
A schedule is said to be preemptive if a task can be interrupted at any time in favor of another task and then resumed later:



Task states

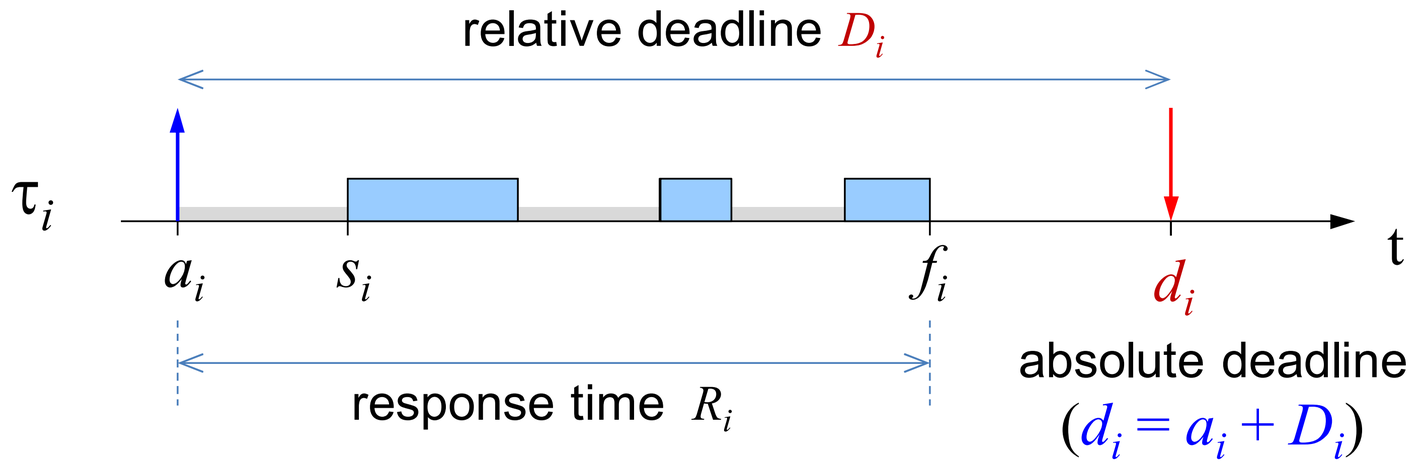

priority
τ1
τ2
τ3
running
running
ready
running
running
ready
running
0
2
4
6
8
10
12
14
16
18
20
σ(t)
3
2
1
0
0
2
4
6
8
10
12
14
16
18
20

Task states


BLOCKED
signal
wait
dispatching
activation
termination
READY
RUNNING
preemption
ACTIVE

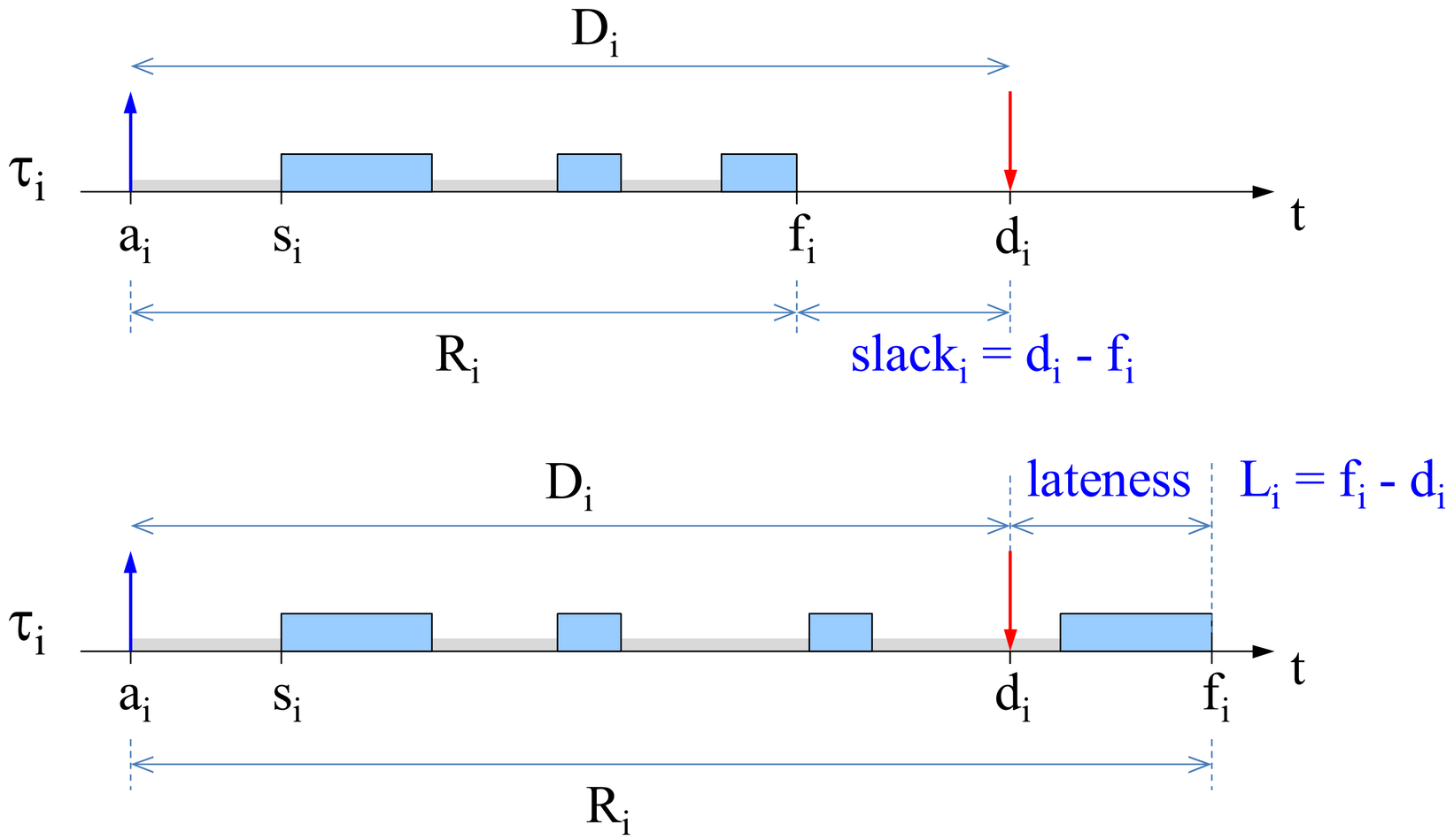
Definition: Real-Time Task

It is a task with a timing constraint on its response time, called deadline:



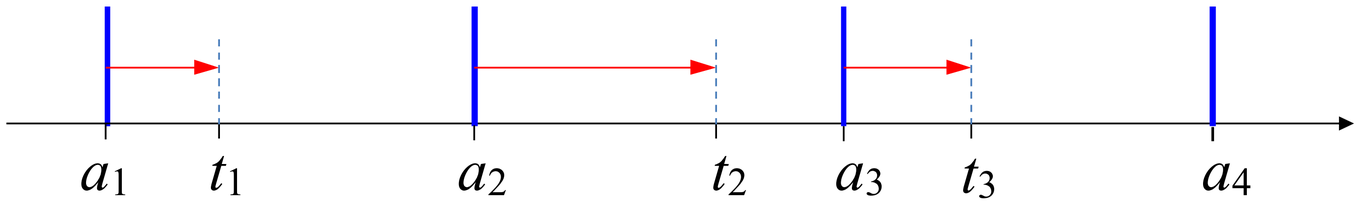
A real-time task τ_i is said to be feasible if it is guaranteed to complete within its deadline, that is, if $f_i \leq d_i$ (or $R_i \leq D_i$).

Slack and Lateness



Jitter

It is a measure of the time variation of a periodic event:

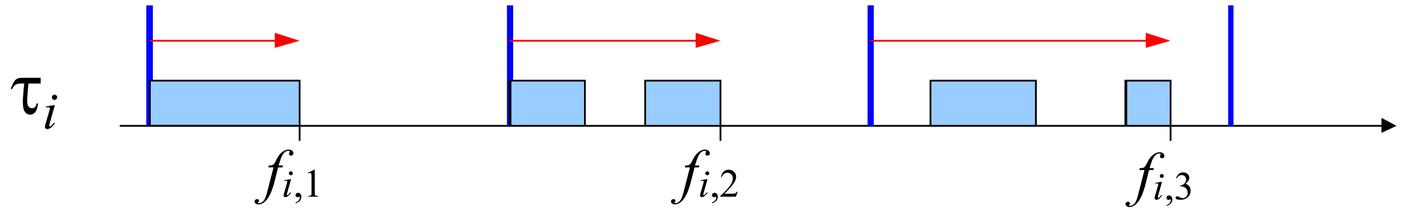


Absolute: $\max_k (t_k - a_k) - \min_k (t_k - a_k)$

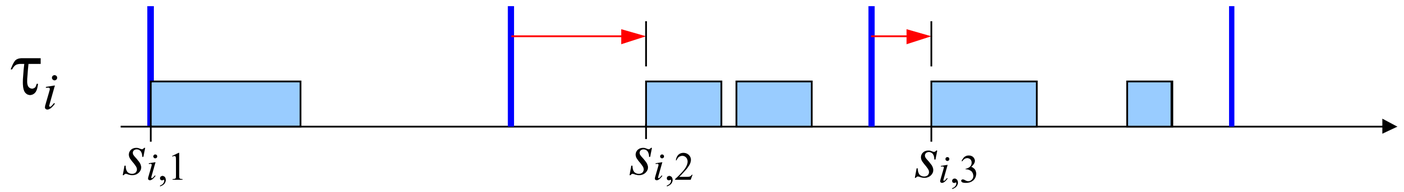
Relative: $\max_k \left| (t_k - a_k) - (t_{k\text{-}1} - a_{k\text{-}1}) \right|$

Types of Jitter

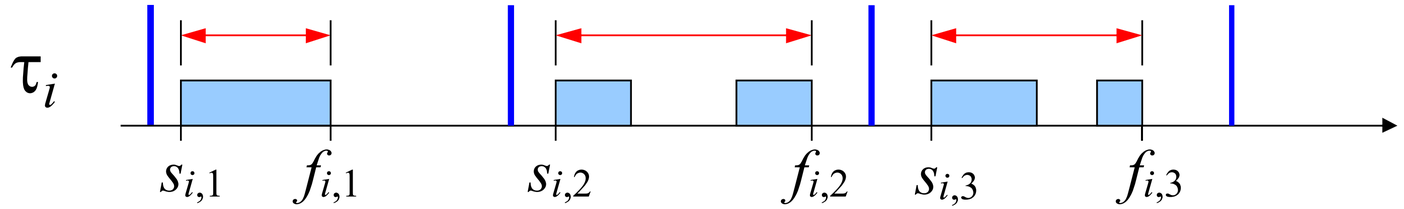
Finishing-time Jitter



Start-time Jitter

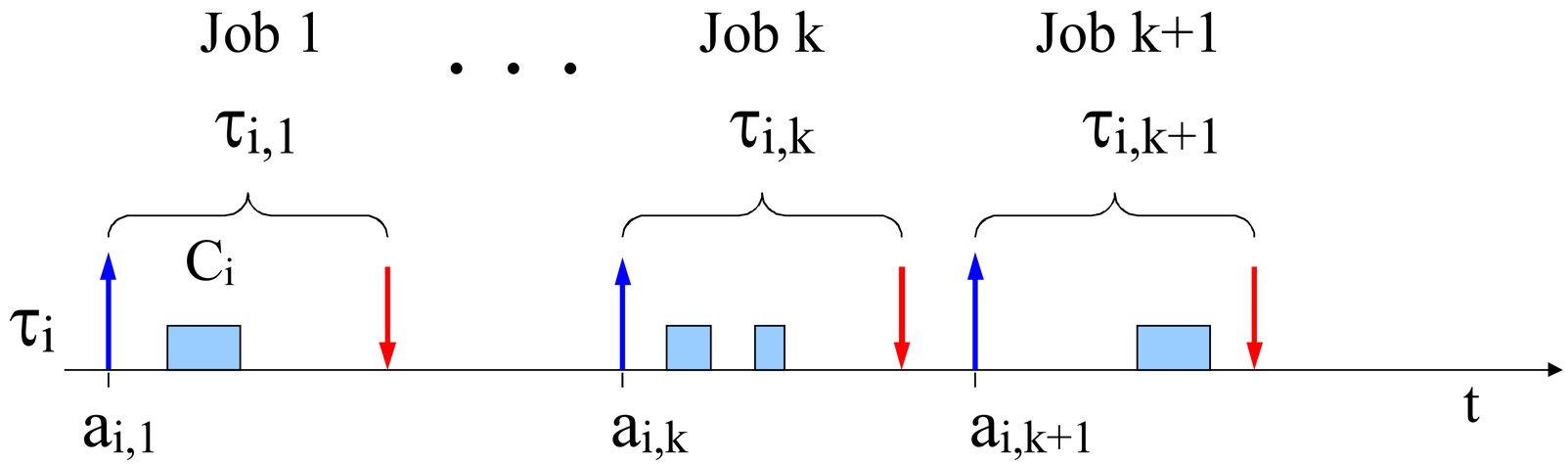


Completion-time Jitter (I/O Jitter)



Tasks and jobs

A task running several times on different input data generates a sequence of instances (or jobs):



Activation mode

- **Periodic** (time-driven activation)

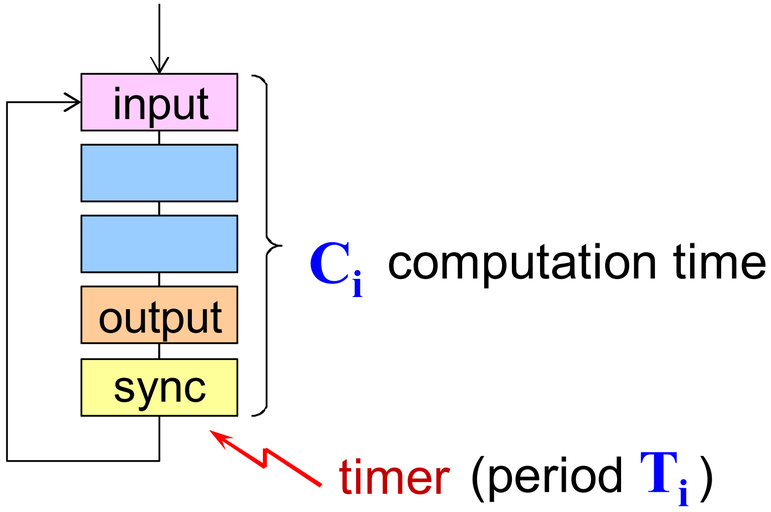
  A task is said to be periodic if its jobs are automatically activated by the operating system at predefined time instants.

- **Aperiodic** (event-driven activation)

  A task is said to be aperiodic if its jobs are activated at the arrival of an event (by interrupt or by another task through an explicit system call).

  If the activation interval between consecutive jobs cannot be smaller than a given quantity, the task is said to be **sporadic**.

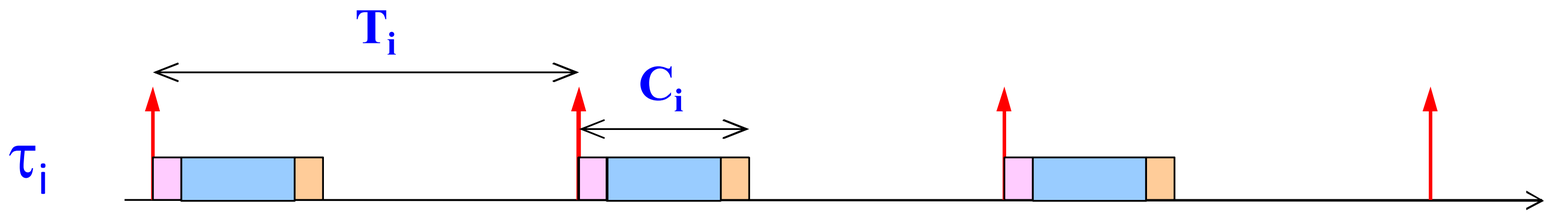
Periodic task



$$U_i = \frac{C_i}{T_i}$$

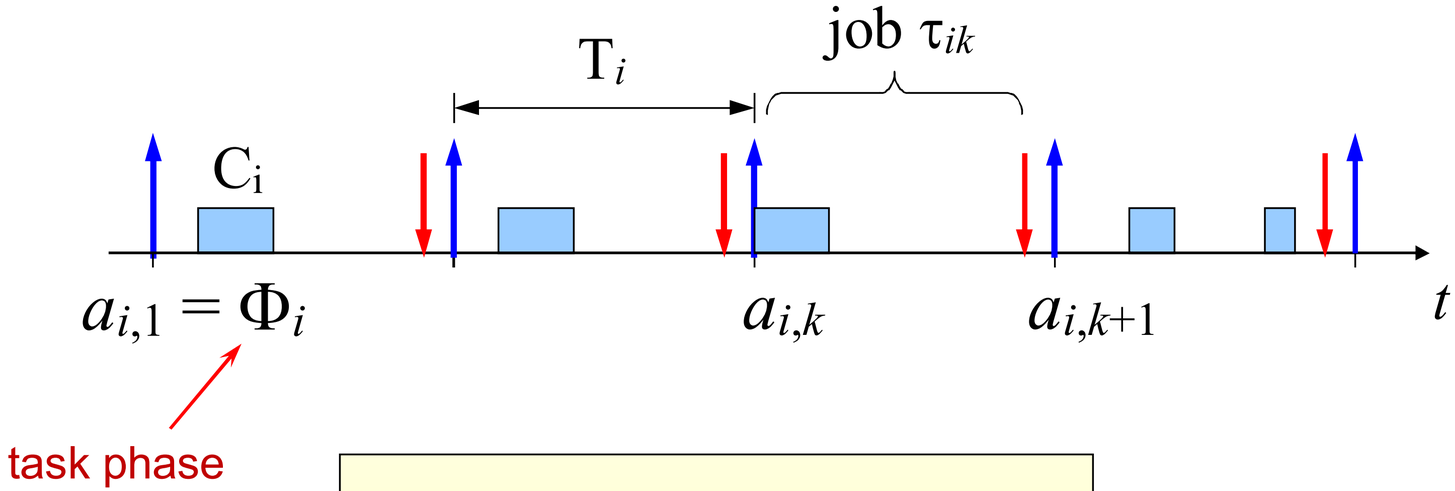
utilization factor

- A periodic task τ_i generates an infinite sequence of jobs: τ_{i1}, τ_{i2}, …, τ_{ik} (same code on different data):



Periodic task

A periodic task can be fully described by four parameters only: phase (Φ_i), worst-case computation time (C_i), period (T_i), and relative deadline (D_i).



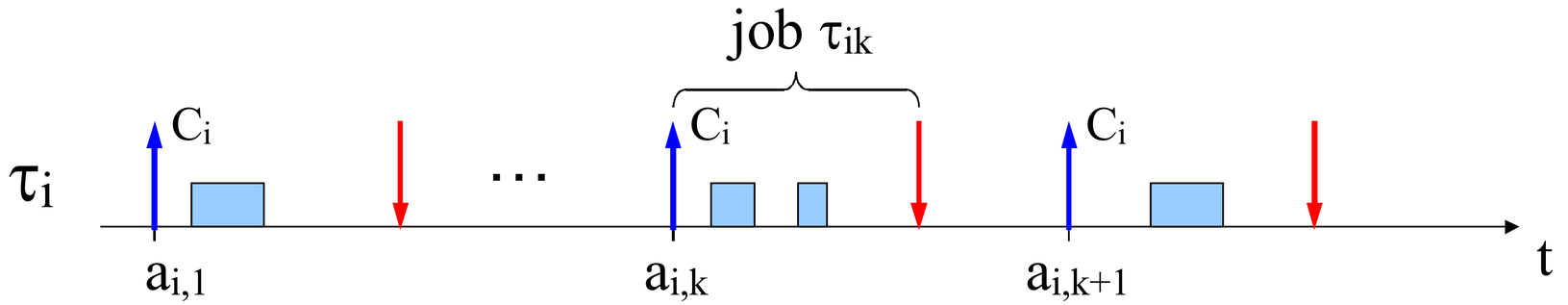
$$a_{i,k} = \Phi_i + (k-1)\,T_i$$

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

Aperiodic task

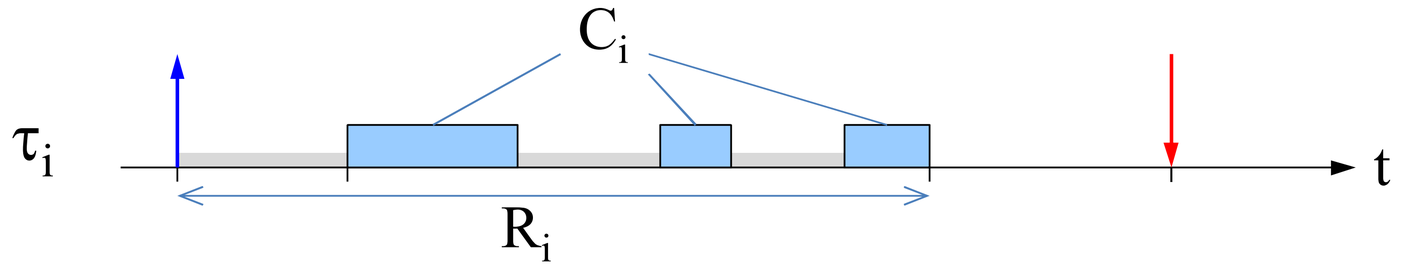
- **Aperiodic:** $a_{i,k+1} > a_{i,k}$
- **Sporadic:** $a_{i,k+1} \geq a_{i,k} + T_i$

minimum interarrival time



Estimating C_i is not easy

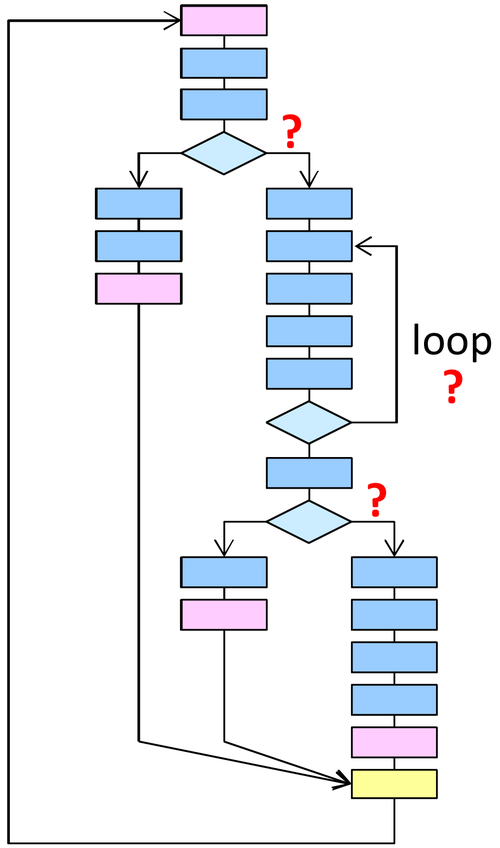
C_i (computation time or execution time) is the time taken by a processor to execute task τ_i, without considering suspension times.



If we want to guarantee that a task meets its deadline for all possible cases (i.e., event combinations), we have to estimate its worst-case execution time (WCET).

It can be done in two ways:
- by analysis
- by measurements

By Analysis



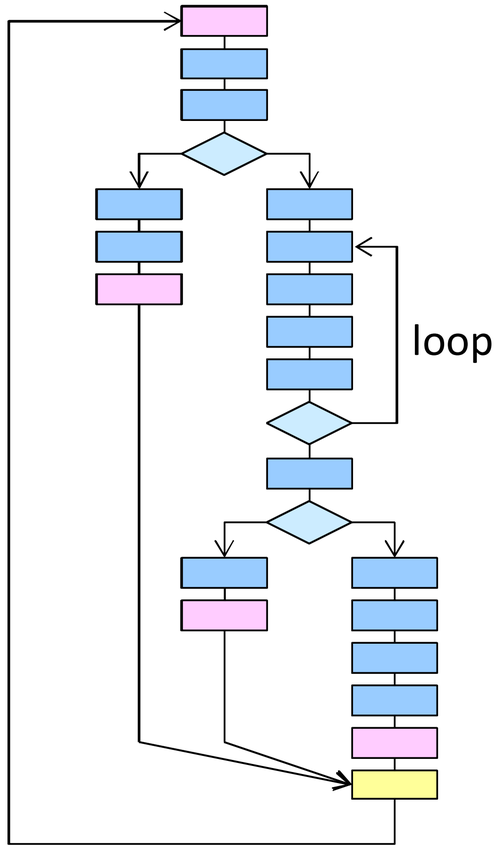
To make a precise estimate, we need to know the task code and the CPU speed.

Each job operates on different data and can take different paths.

To estimate C_i you have to

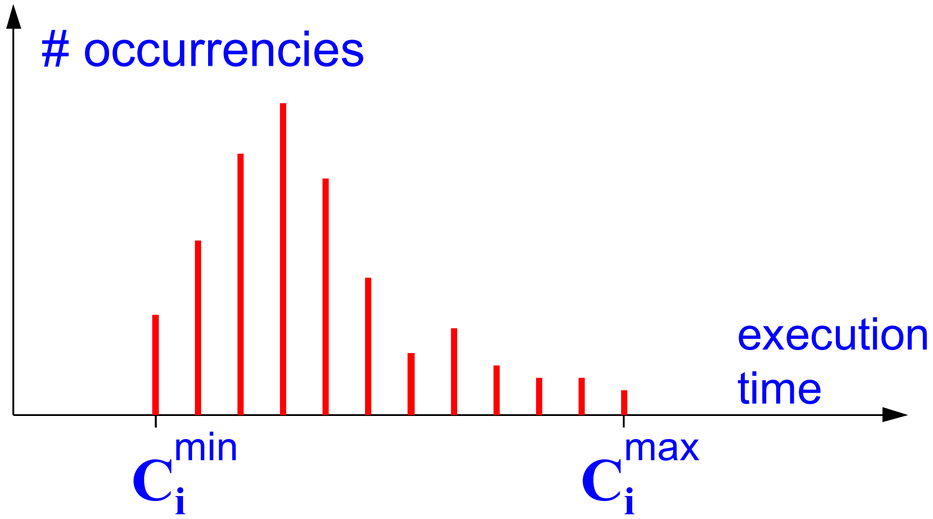
- Bound all loop cycles;
- Compute the longest path;
- Bound the number of cache misses;
- Compute the execution time for each instruction in the longest path.

By measurements

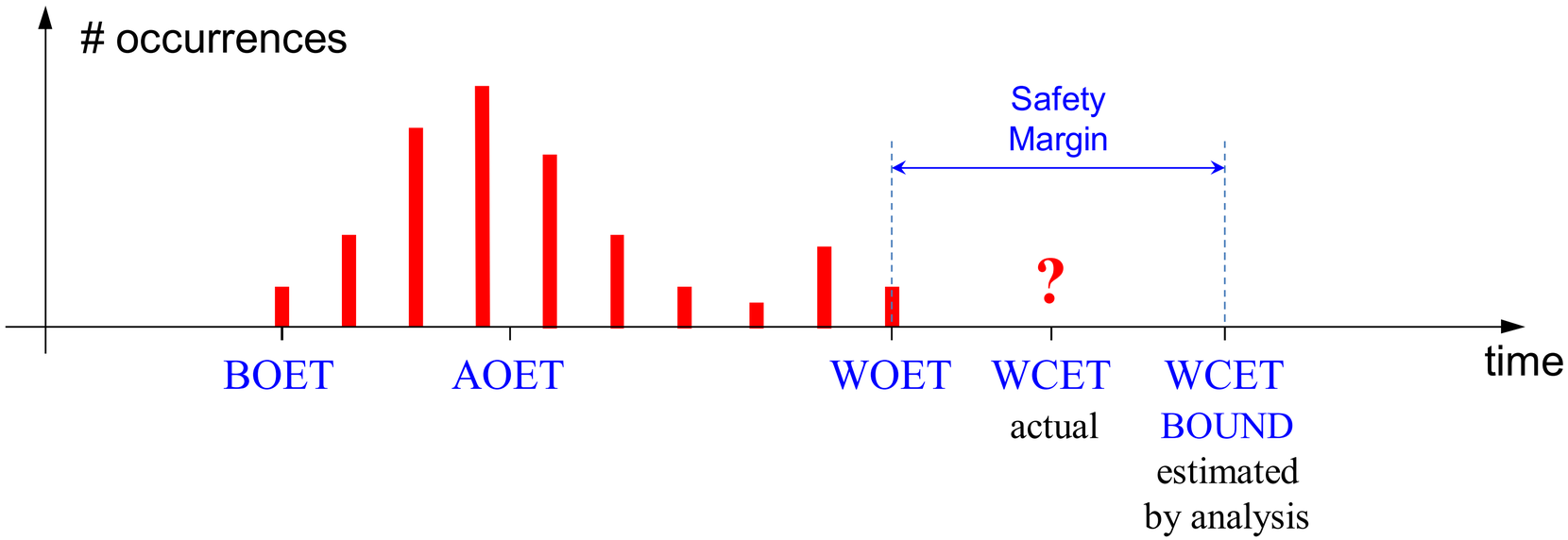


To estimate C_i you have to

- Execute the task in isolation (i.e., alone) several times under different workloads and input data.
- Collect statistics on the execution times:



By measurements



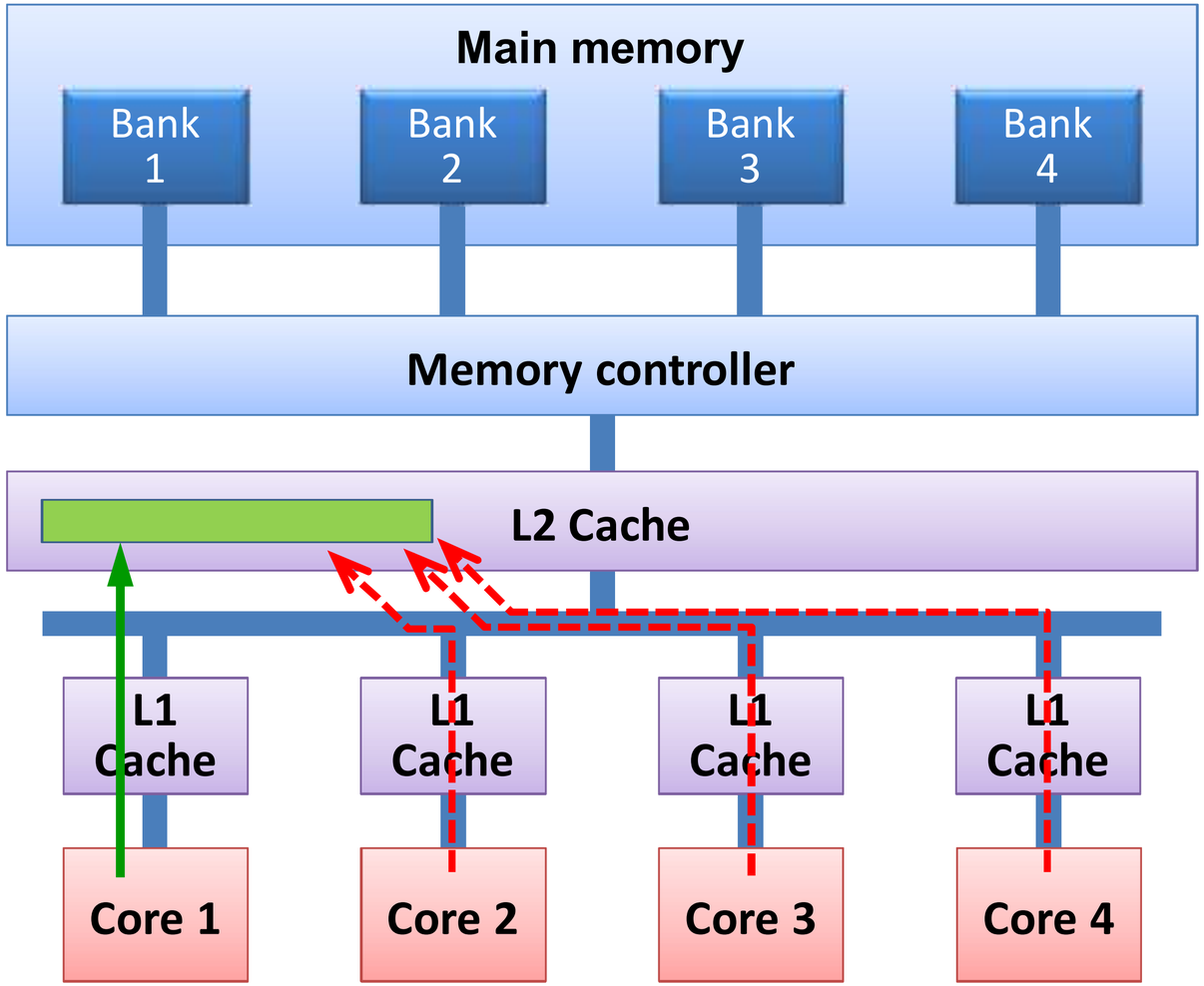
BOET: Best Observed Execution Time

AOET: Average Observed Execution Time

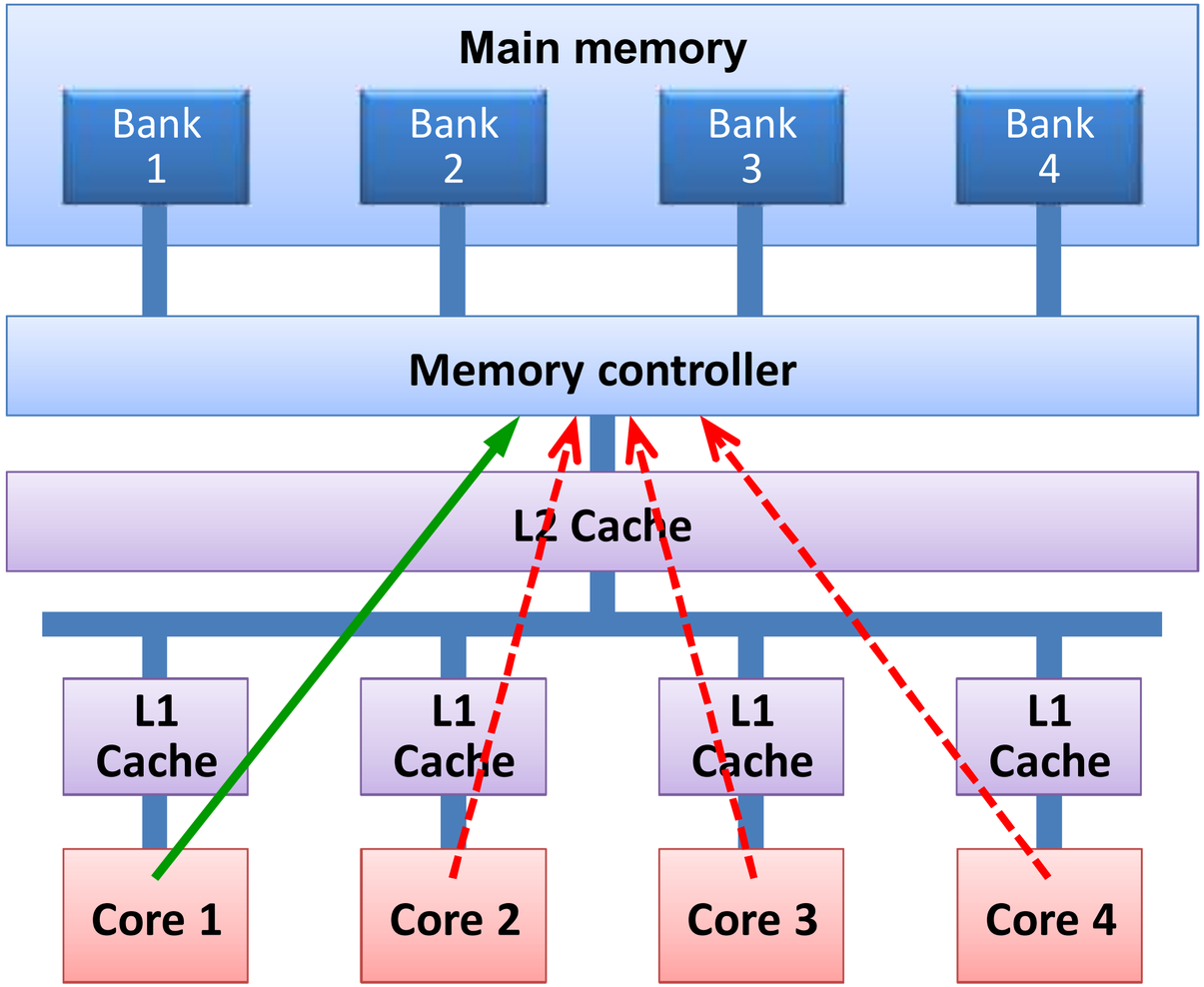
WOET: Worst Observed Execution Time

WCET: Worst-Case Execution Time

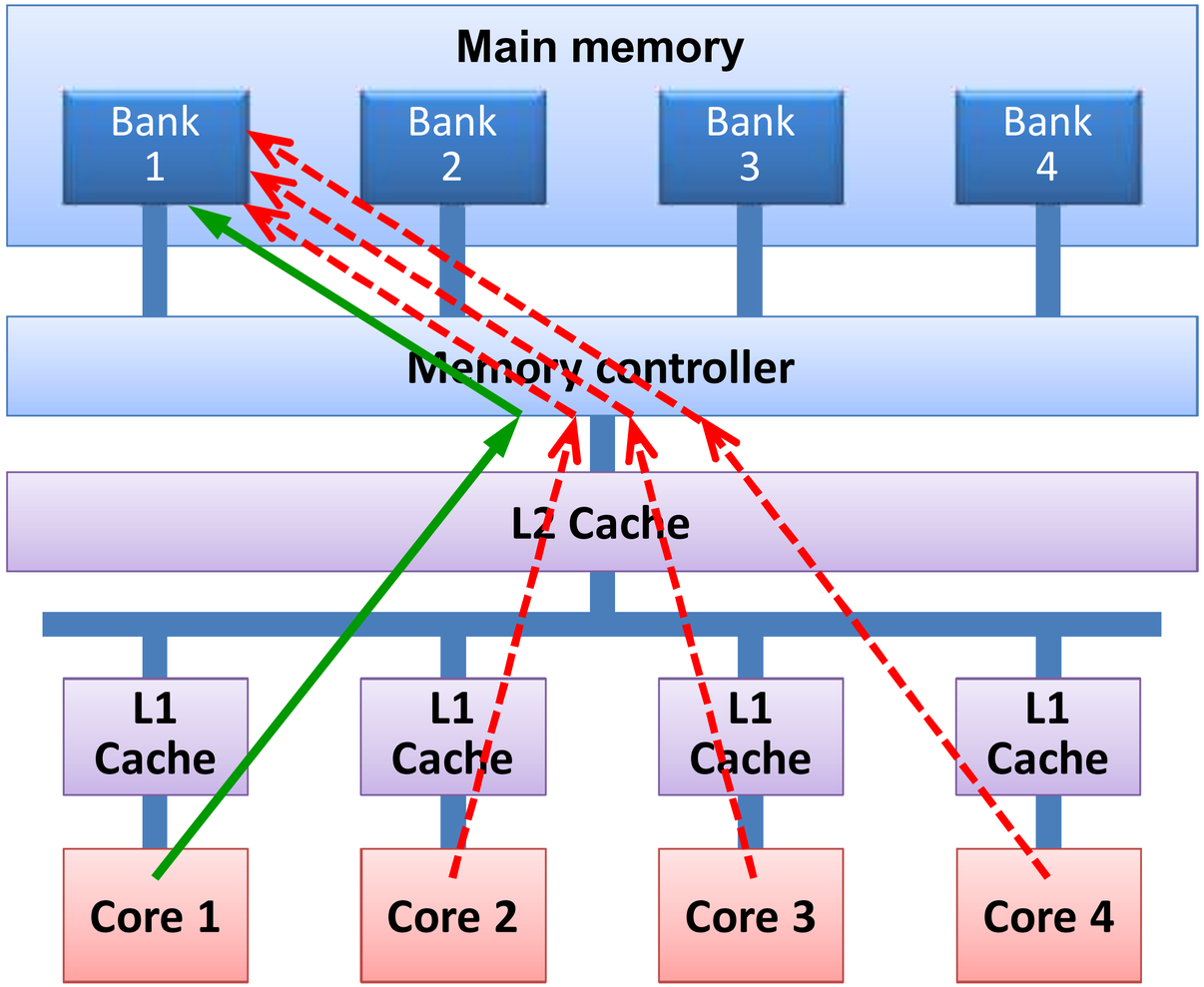
Types of interference



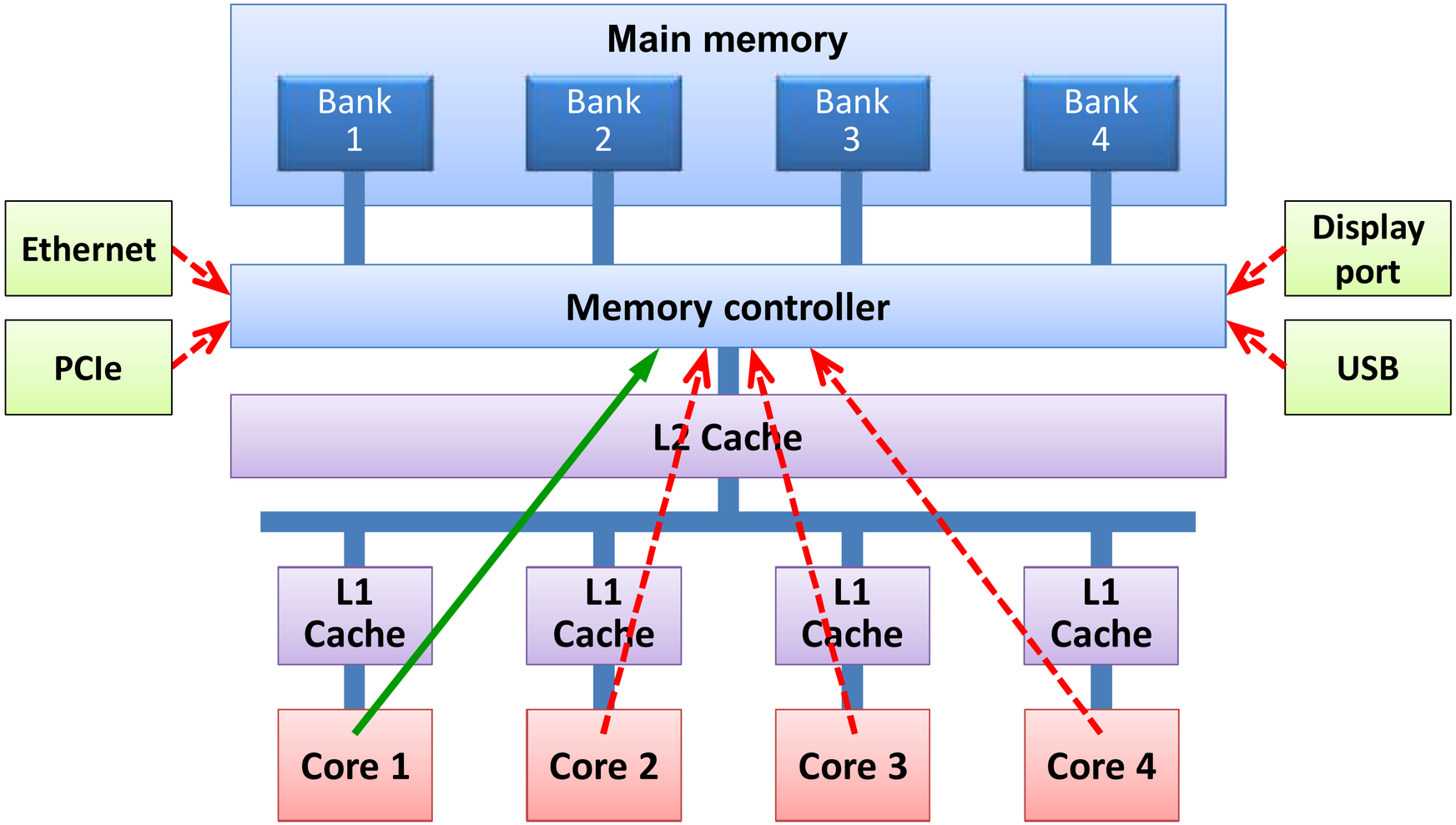
Types of interference



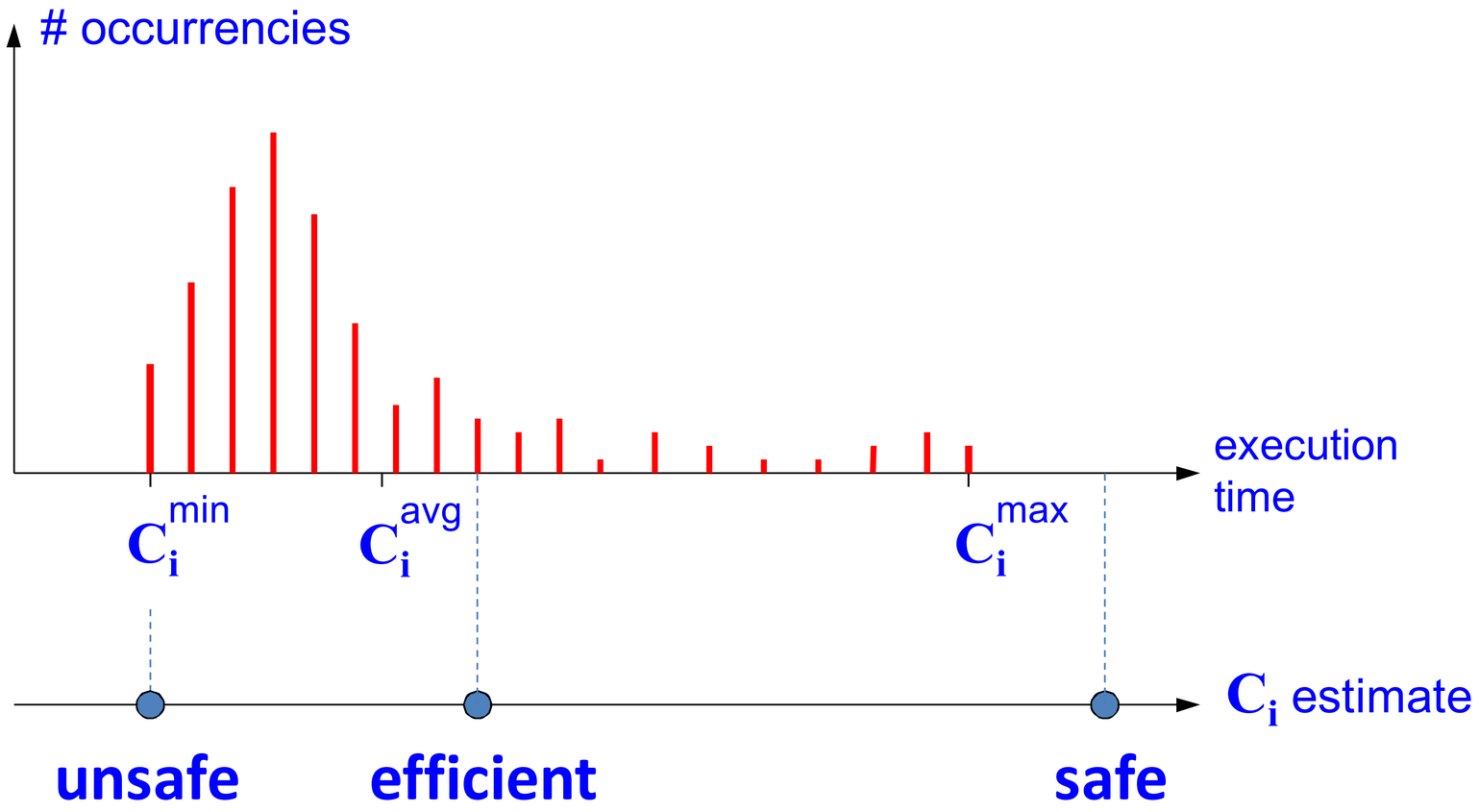
Types of interference



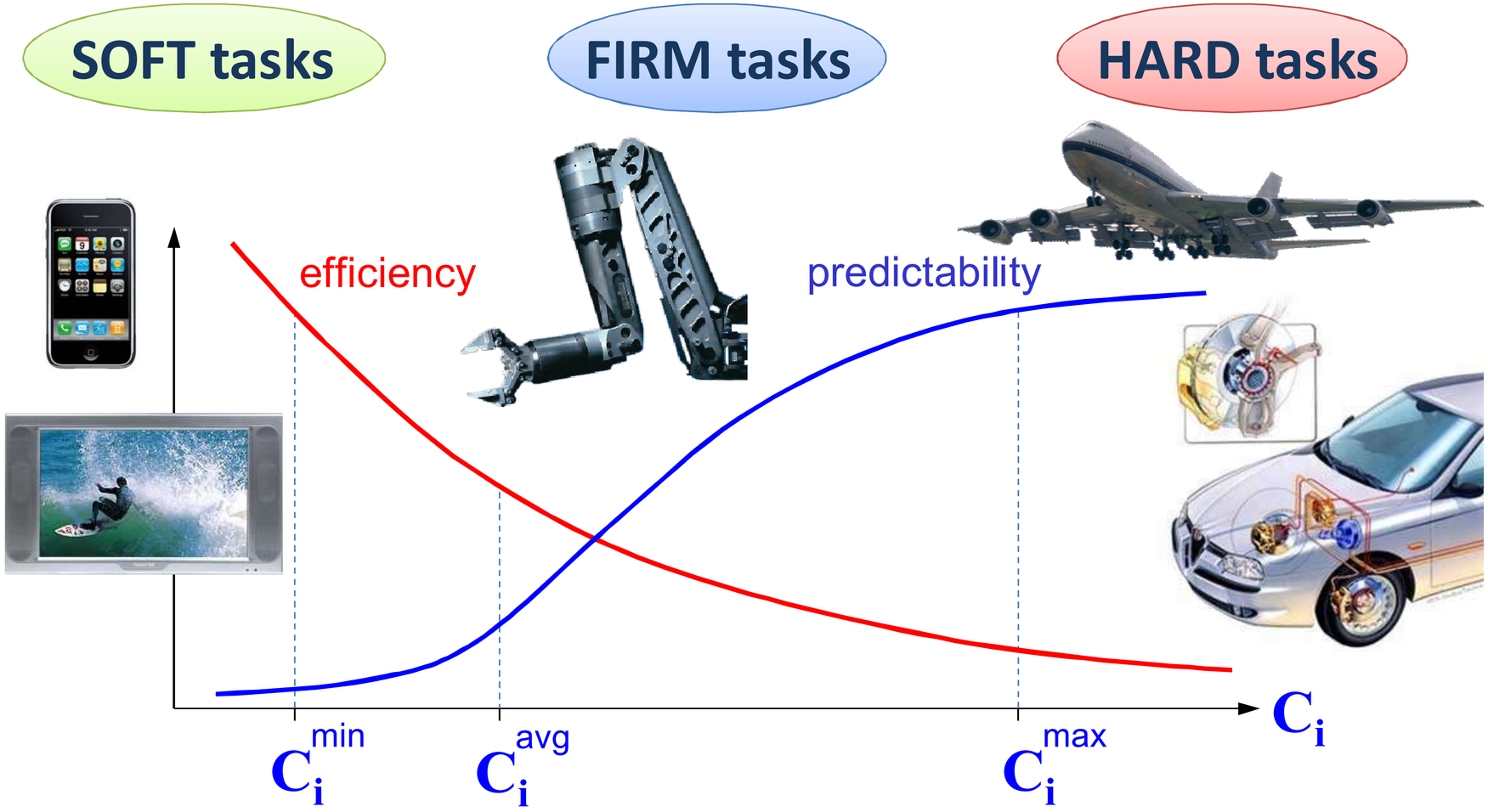
Types of interference



Predictability vs. Efficiency



Predictability vs. Efficiency



Criticality

HARD task

All jobs must meet their deadlines. Missing a single deadline may cause catastrophic effects on the whole system.

FIRM task

Missing a job deadline has not catastrophic effects on the system, but invalidates the execution of that particular job.

SOFT task

Missing a deadline is not critical. A job finishing after its deadline has still some value but causes a performance degradation.

An operating system able to handle hard real-time tasks is called a **hard real-time** system.

Criticality

Typical HARD tasks

- sensory acquisition
- low-level control
- sensory-motor planning

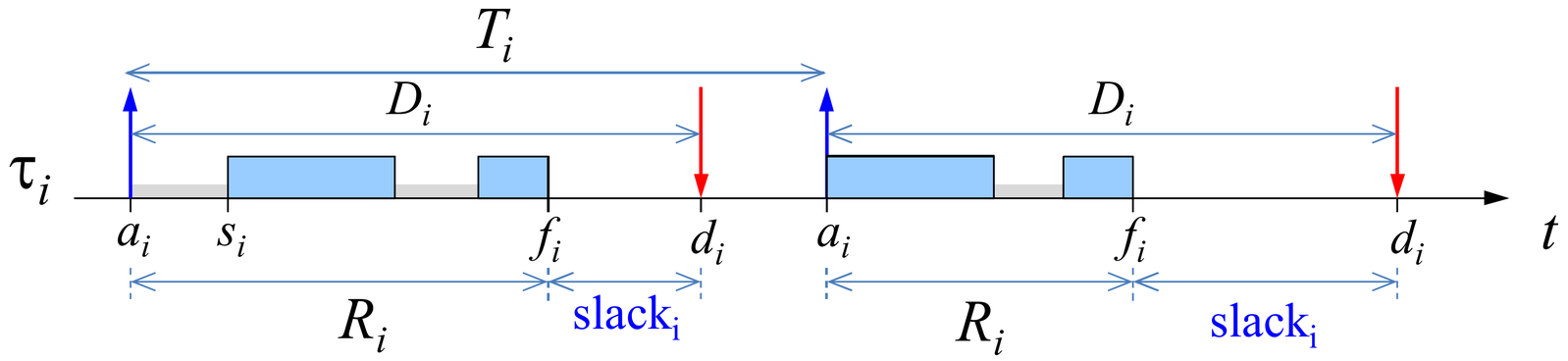
Typical FIRM tasks

- RT audio processing
- RT video decoding

Typical SOFT tasks

- reading data from the keyboard
- user command interpretation
- message displaying
- graphical activities

Parameters summary



- Computation time (C_i)
- Period (T_i)
- Relative deadline (D_i)

These parameters are specified by the programmer and are known off-line

- Arrival time (a_i)
- Start time (s_i)
- Finishing time (f_i)
- Response time (R_i)
- Slack and Lateness
- Jitter

These parameters depend on the scheduler and on the actual execution, and are known at run time.

Task Constraints

Types of constraints

- **Timing constraints**
  - 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

They can be explicit or implicit.

Explicit timing constraints

They are directly included in the system specifications.

Examples

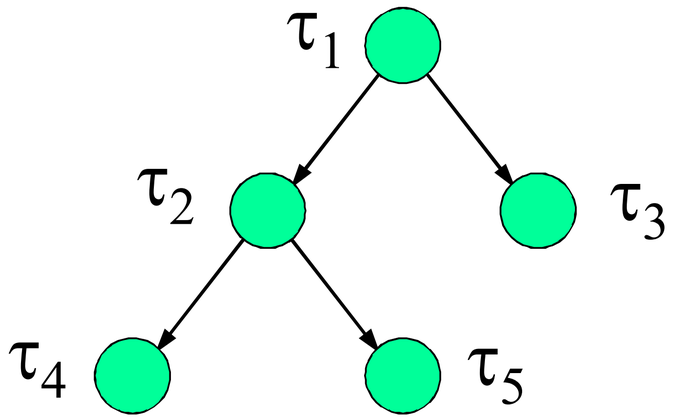
- open the valve **in** 10 seconds
- send the position **within** 40 ms
- read the altimeter **every** 200 ms
- acquire the camera **every** 20 ms

Implicit timing constraints

They are not explicitly included in the system specifications, but need to be satisfied to meet performance requirements.

Precedence constraints

Sometimes tasks must be executed with specific precedence relations, specified through a **Directed Acyclic Graph** (**DAG**):



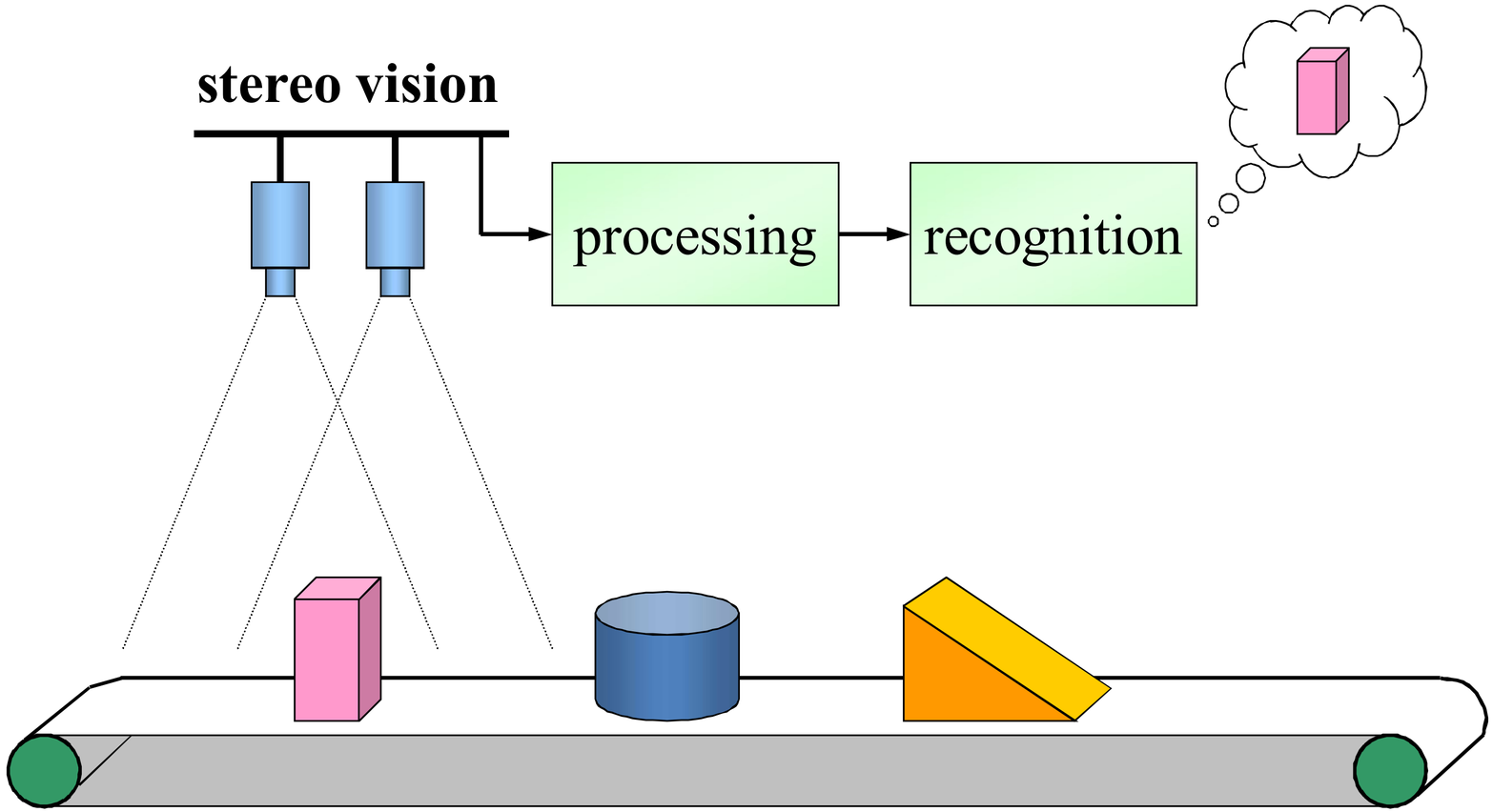
predecessor

$$\tau_1 \prec \tau_4$$

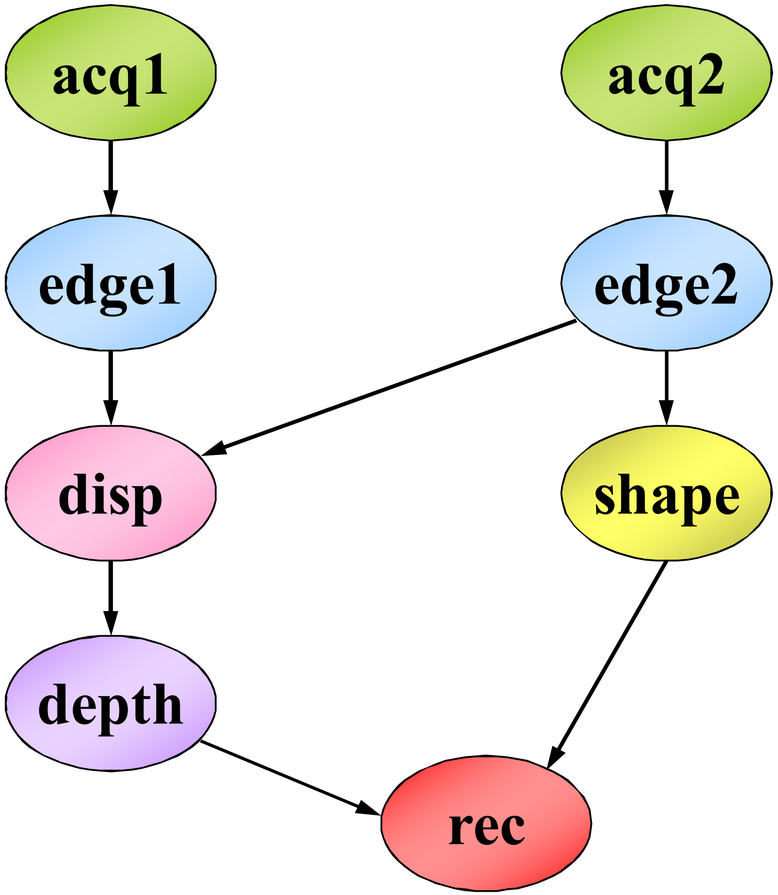
immediate predecessor

$$\tau_1 \rightarrow \tau_2$$

Sample application

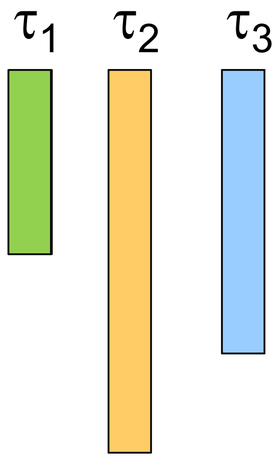


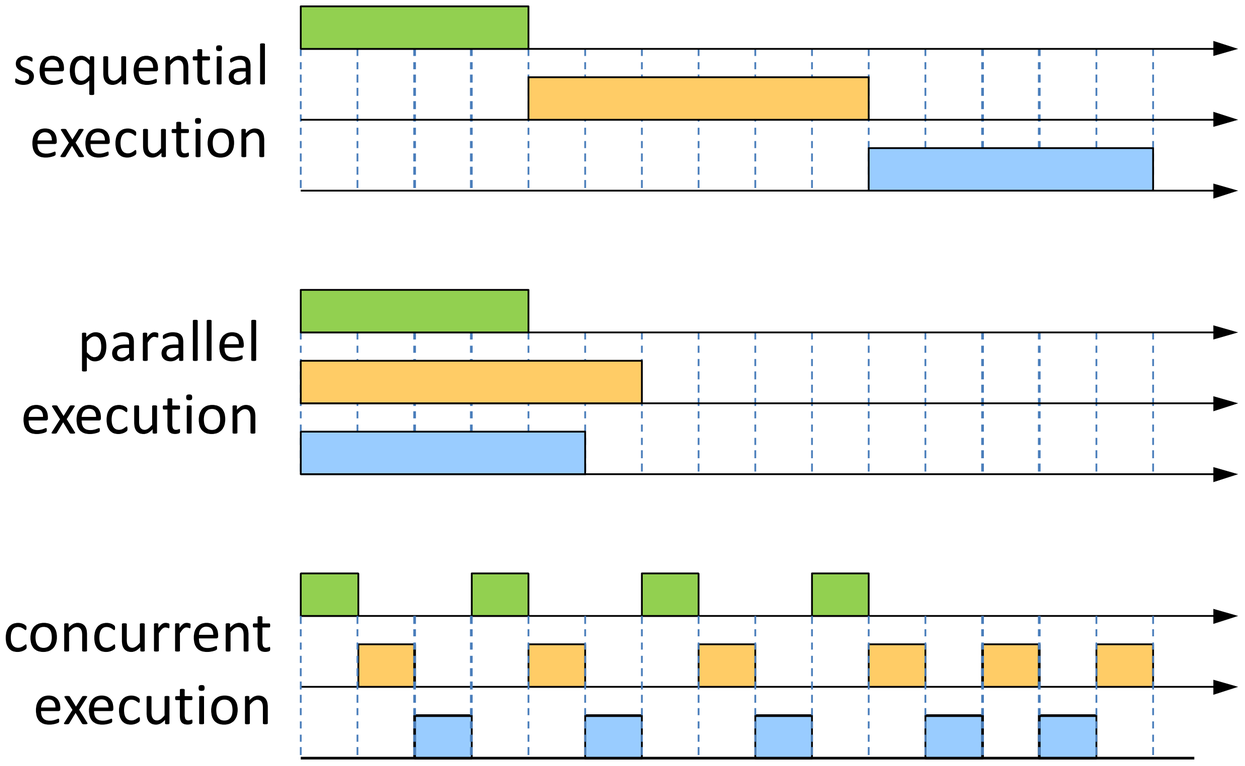
Precedence graph



Concurrency

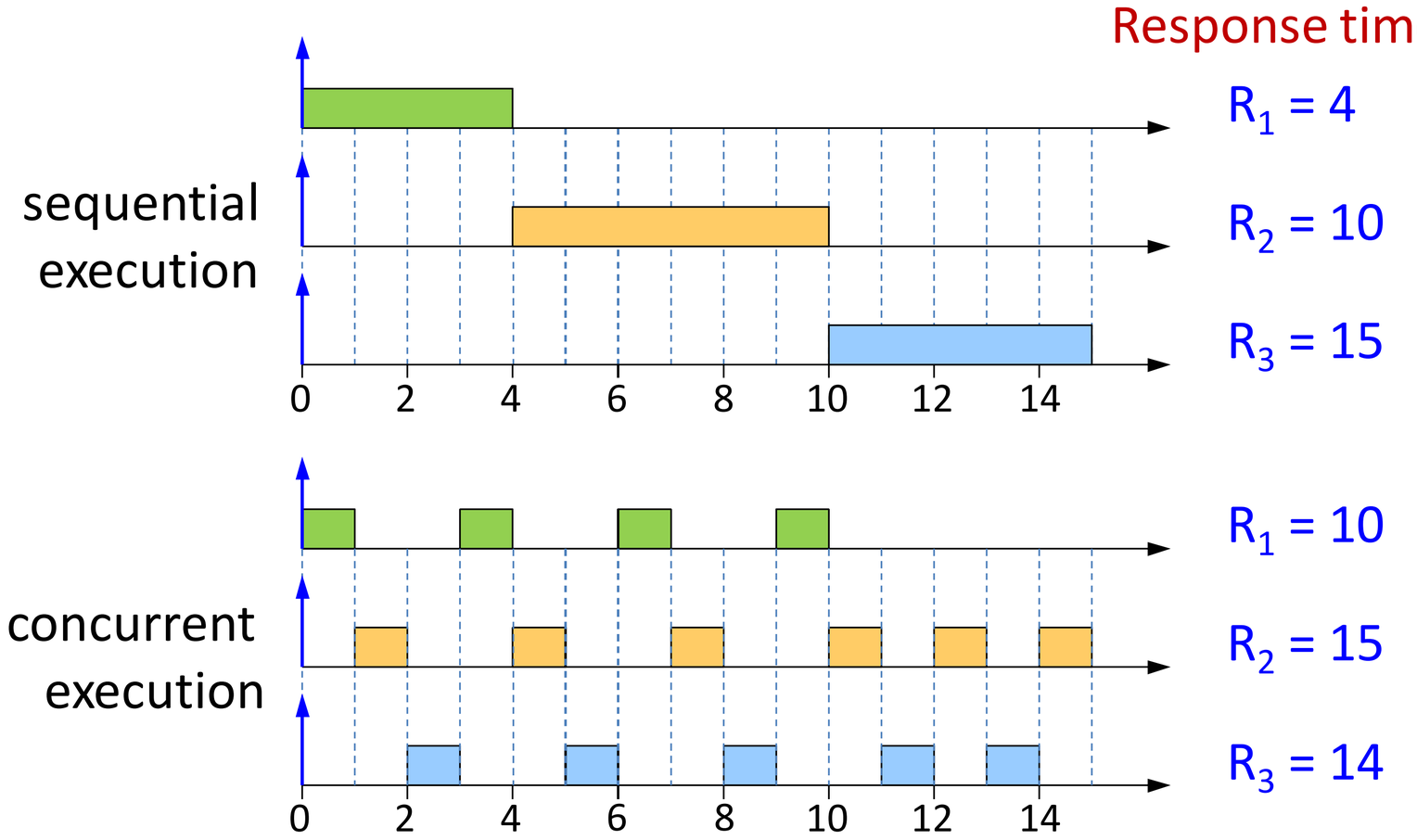
Resource conflicts are caused by concurrency, that is the ability of the processor to execute more tasks at a time, by alternating their executions:





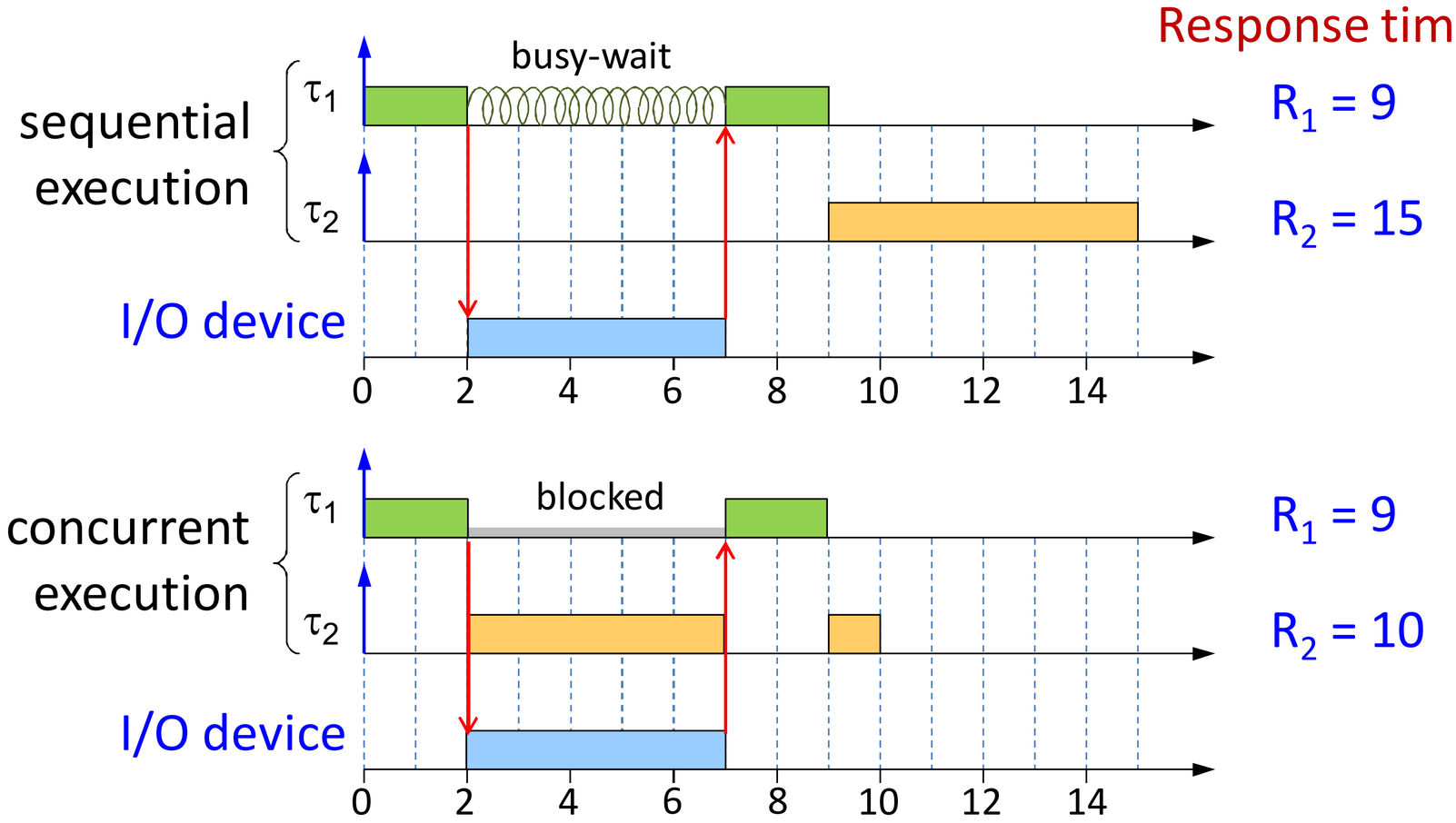
Concurrency

Comparing sequential with concurrent executions, it seems that concurrency has no advantages:



Concurrency and VO

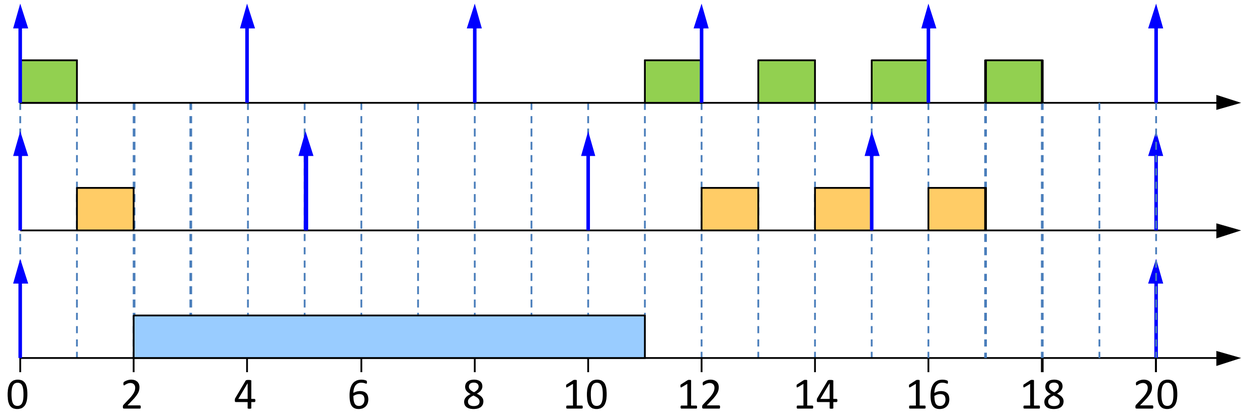
If a task must wait for I/O data, concurrency allows another task to run during that interval:



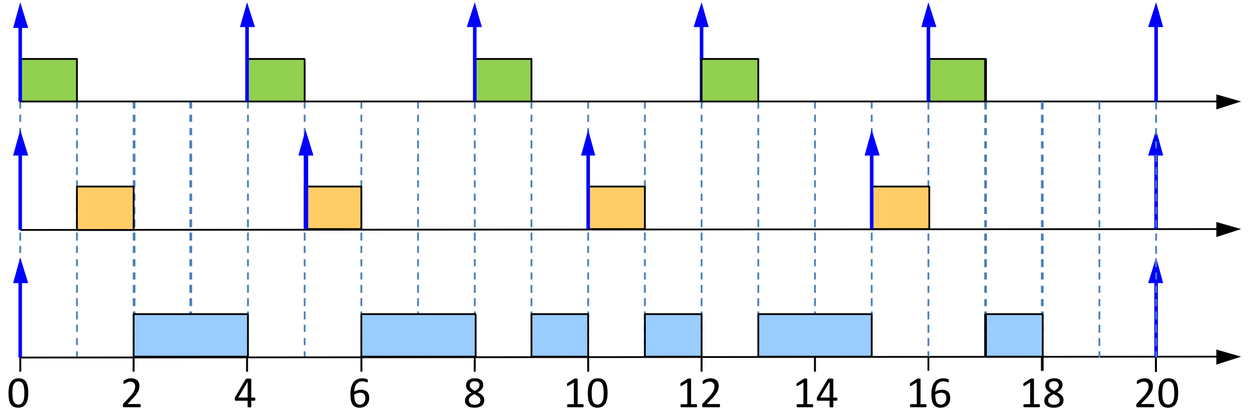
Periodic tasks

Concurrency becomes superior when managing periodic tasks at different rates (waiting times are used to execute other tasks):

sequential execution (FIFO)



concurrent execution (Rate Monotonic)

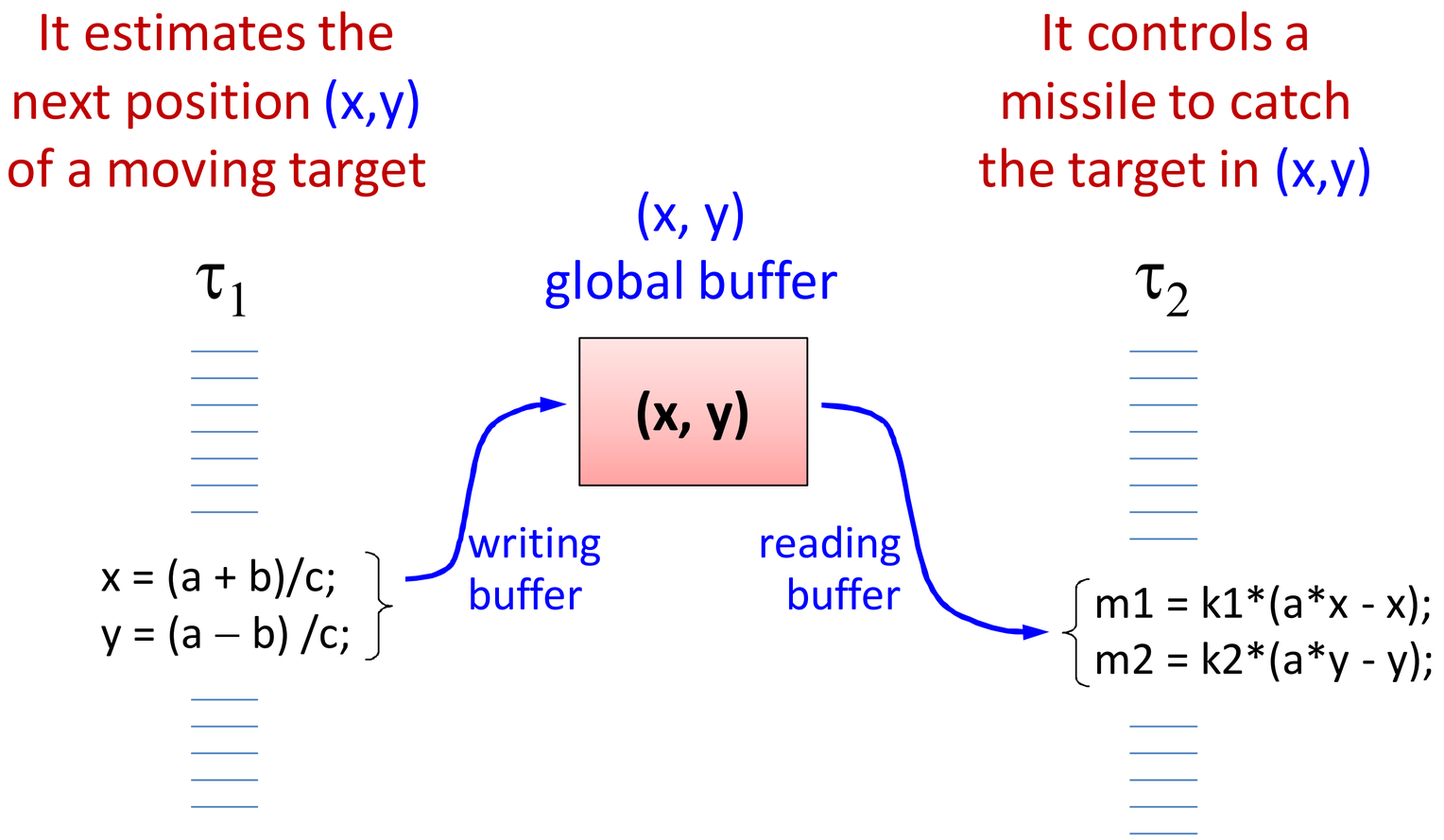


Concurrency: pro and cons

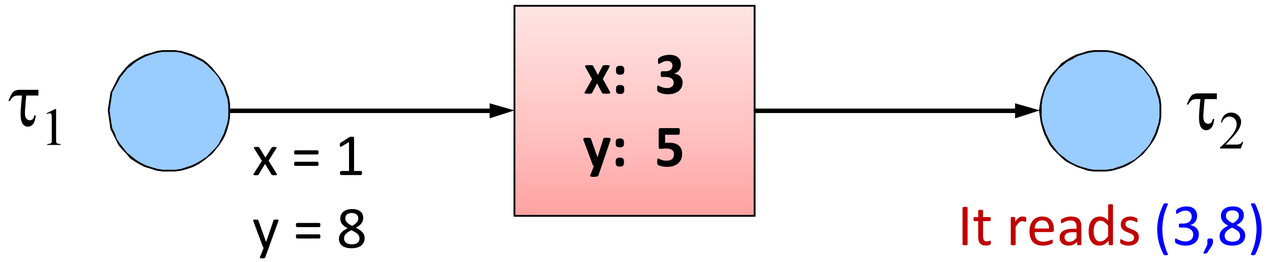
Concurrency allows exploiting tasks inactive intervals (e.g., waiting times for input data or periodic task activation) to execute other tasks.

However, concurrency can generate conflicts when using shared resources (for example, when more tasks operate on global data).

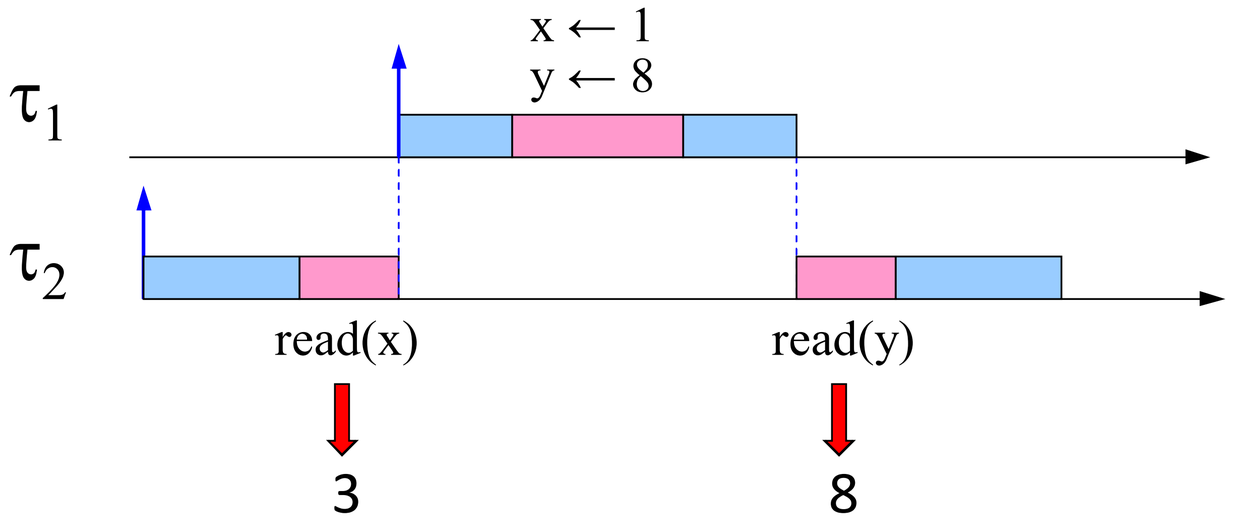
Example of conflict



Example, cont.

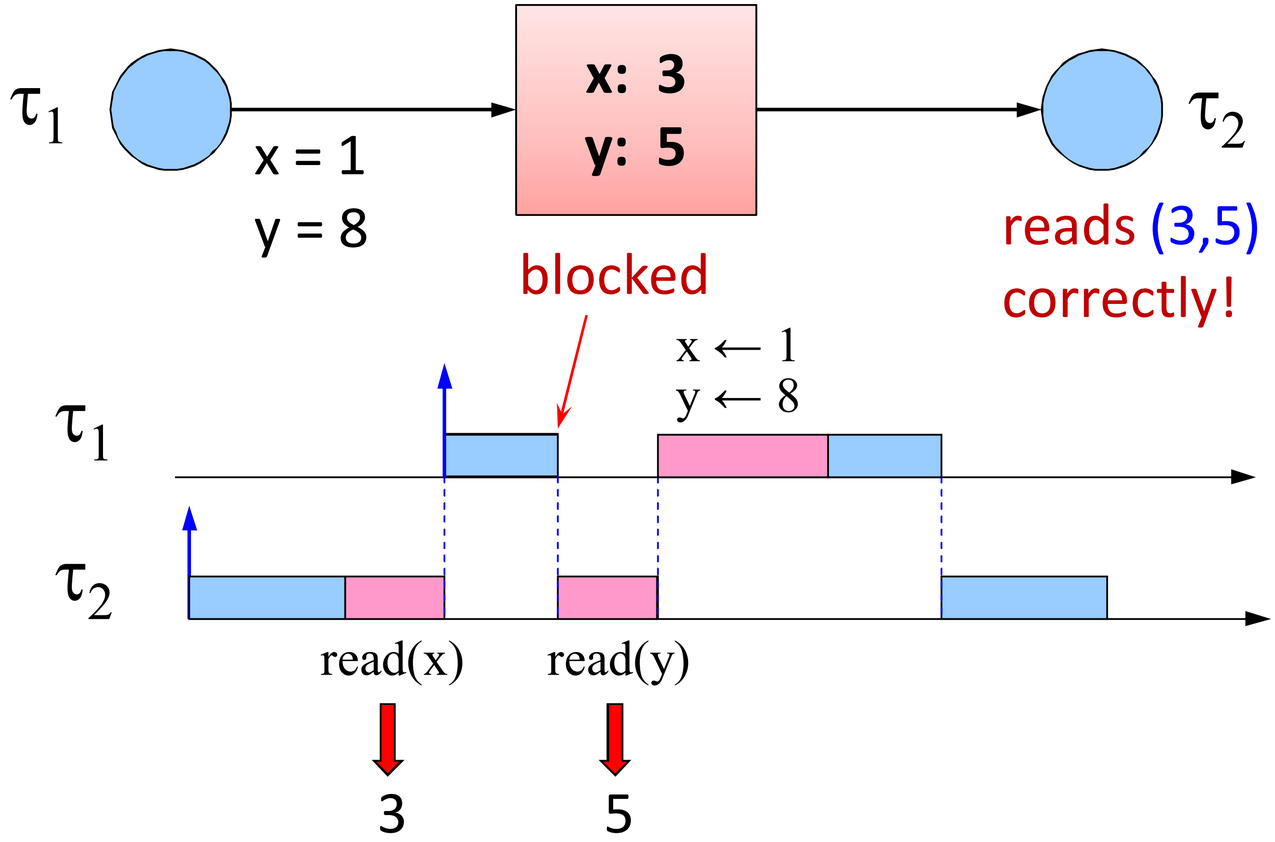


It reads (3,8)
which does not belong
to the trajectory!



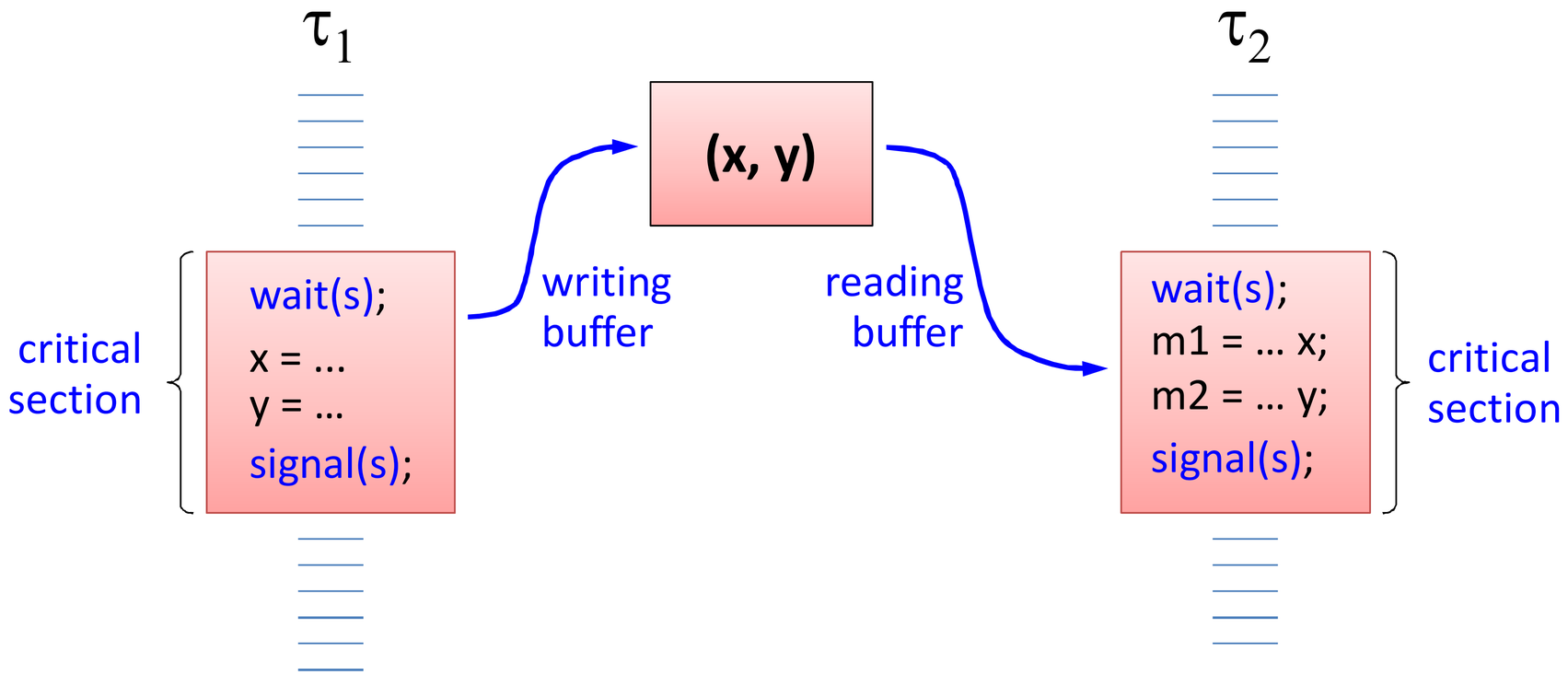
Solution

Regulate the use of shared resources so that tasks can only access them one at the time (i.e., in **mutual exclusion**):



Semaphores

Mutual exclusion is implemented by two primitives, wait(s) and signal(s), that use a system variable s, called semaphore:

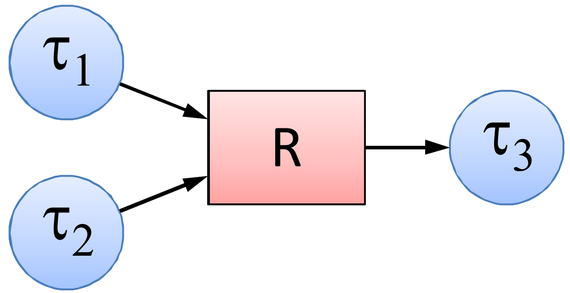


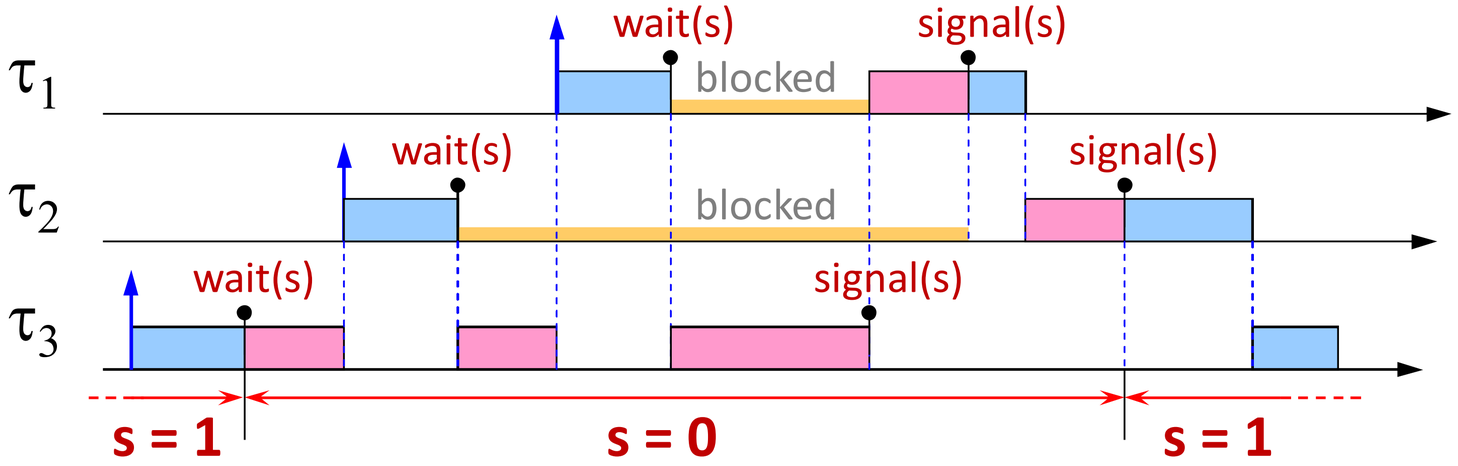
Semaphores

- Each shared resourse is protected by a different semaphore.
- s = 1 $\Rightarrow$ free resource, s = 0 $\Rightarrow$ busy (locked) resource.
- wait(s):
  - if s == 0, the task must be blocked on a queue of the semaphore. The queue management policy depends on the OS (usually it is FIFO or priority-based).
  - else set s = 0.
- signal(s):
  - if there are blocked tasks, the first in the queue is awaken (s remains 0), else set s = 1.

Semaphores

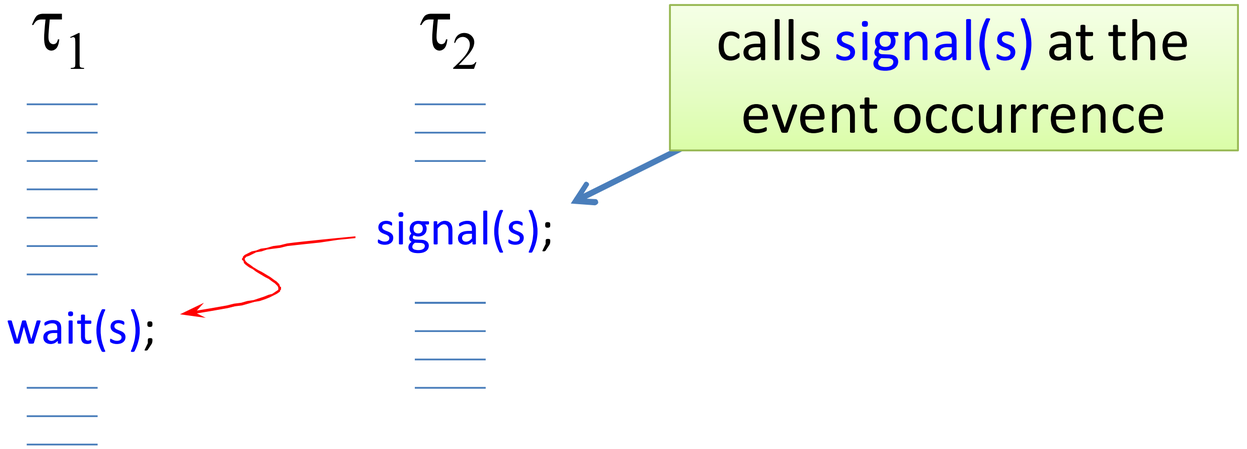
- If the semaphore s is initialized to 1, the pair wait(s) and signal(s) can be used for enforcing mutual exclusion:

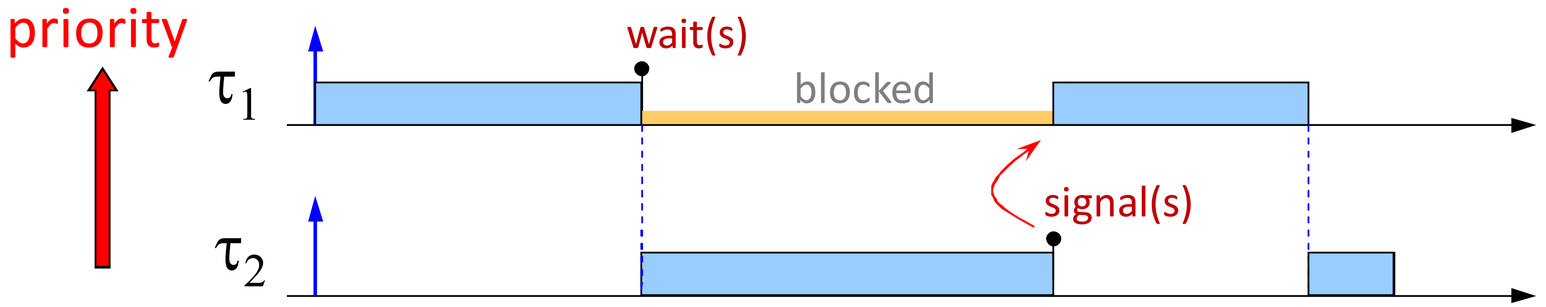




Synchronization semaphores

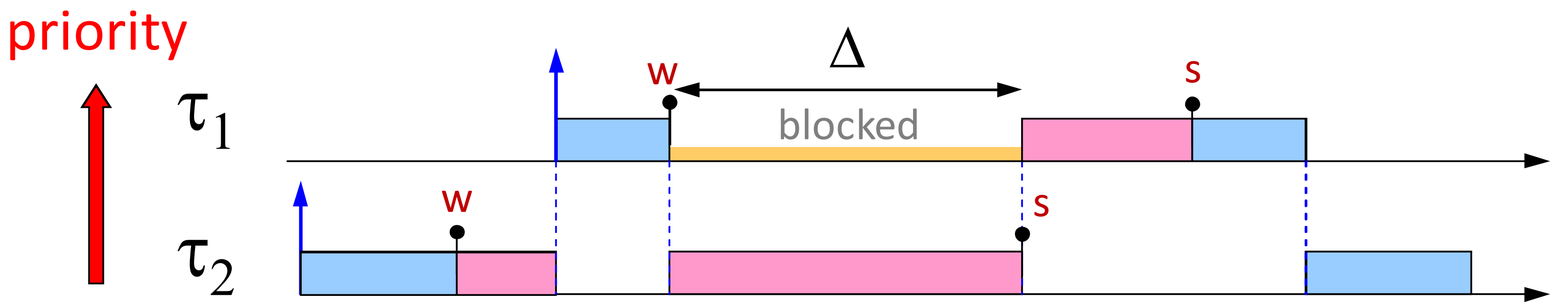
- A semaphore initialized to 0 can be used to wait for an event generated by another task:



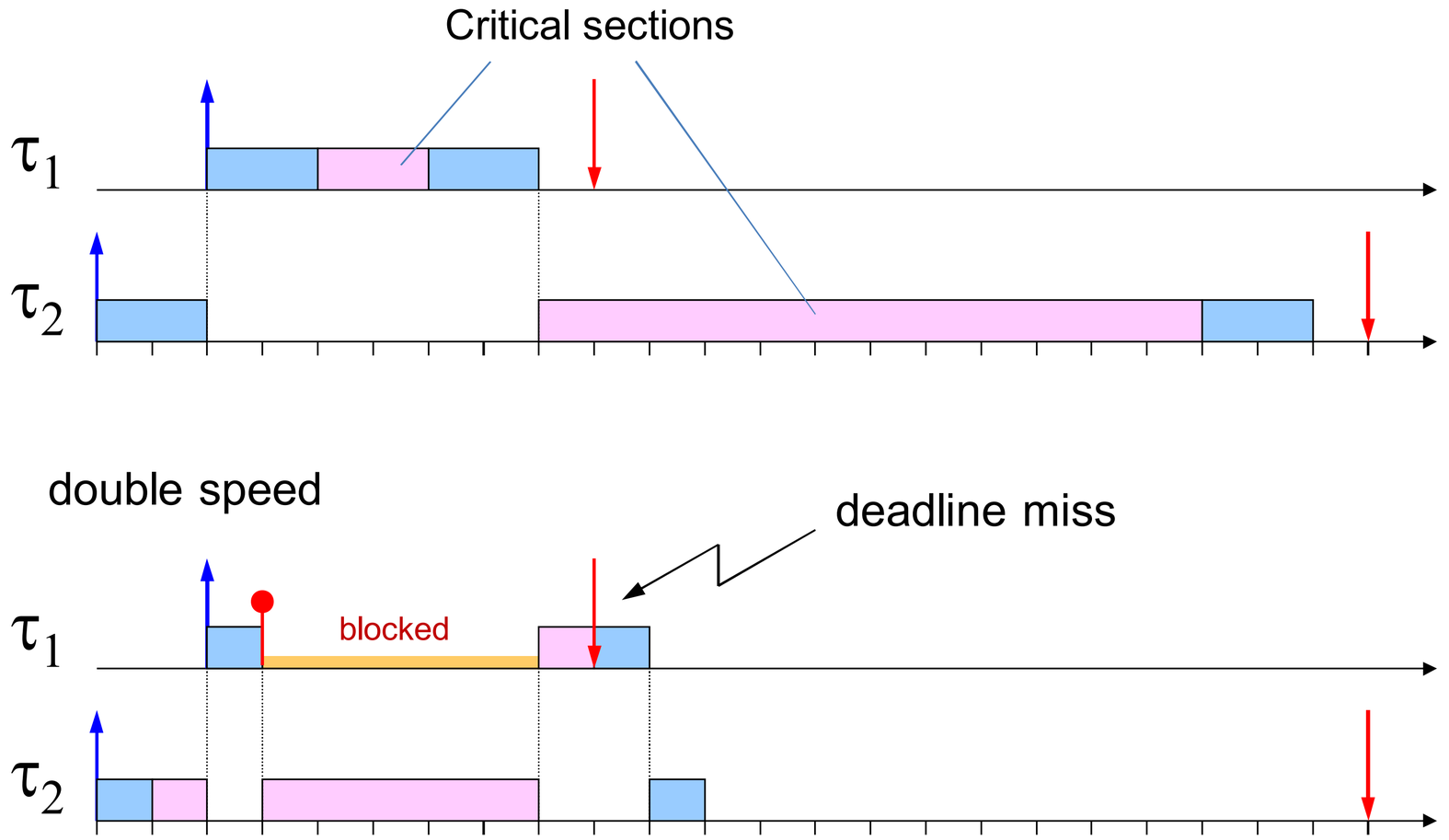


Problem with semaphores

- Semaphores (when properly used) guarantee the consistency of shared global data, but introduce extra blocking delays in high priority tasks.



Faster processor

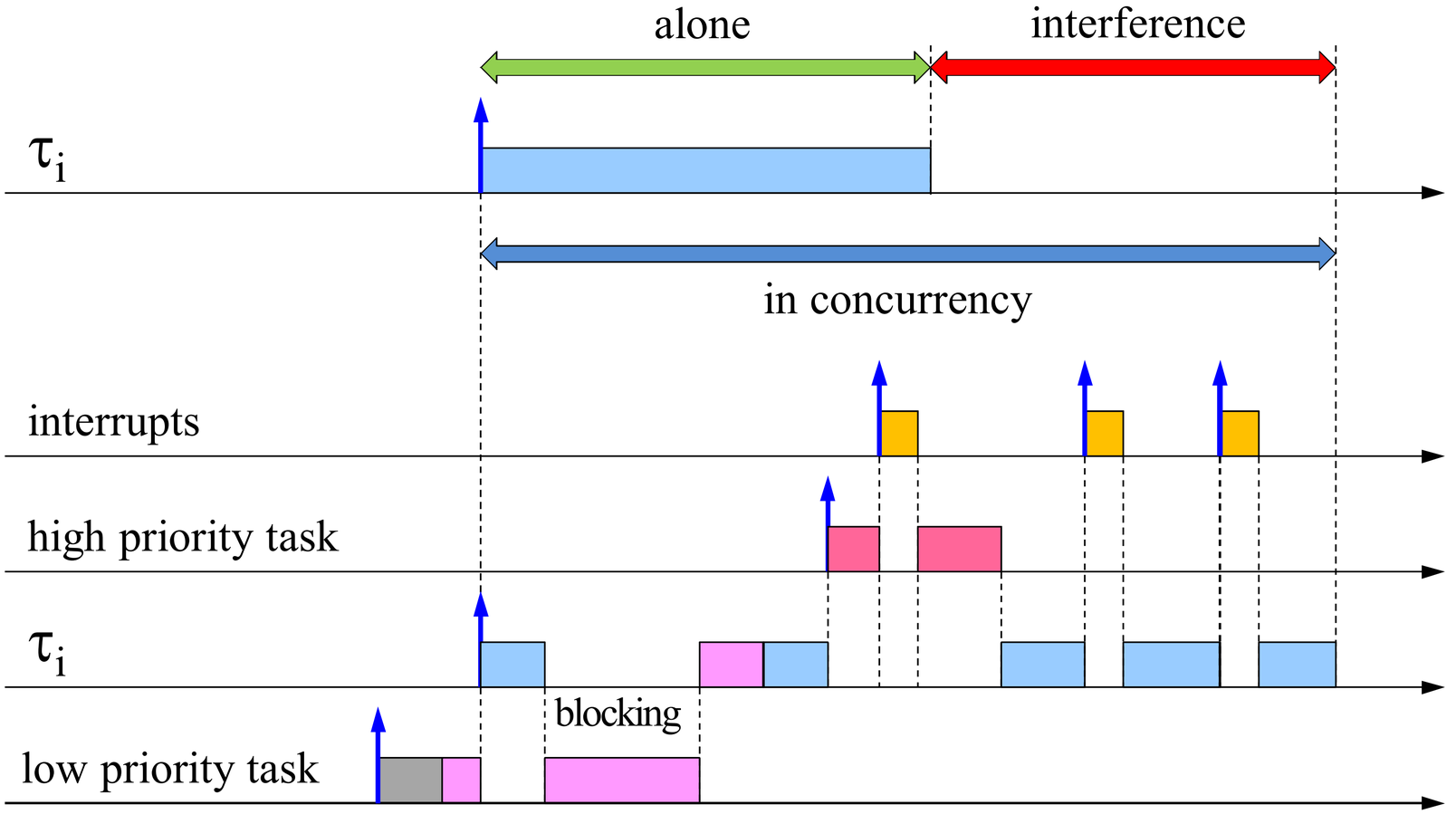


Main sources of interference

There are three major sources of interference in concurrent systems:

1. **High-priority tasks** → Proper priority assignment
2. **Resource sharing** → Resource access protocols
3. **Aperiodic executions** (asynchronous interrupts, event-driven activations) → Aperiodic Servers

Main sources of interference



Achieving predictability

- The operating system is the most important component responsible for achieving a predictable execution.

- Concurrency control must be enforced by:
  - appropriate scheduling algorithms
  - appropriate synchronization protocols
  - efficient communication mechanisms
  - predictable interrupt handling