ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Real-Time Operating Systems M

9. Real-Time: Basic Concepts

Notice

The course material includes slides downloaded from:

http://codex.cs.yale.edu/avi/os-book/

(slides by Silberschatz, Galvin, and Gagne, associated with Operating System Concepts, 9th Edition, Wiley, 2013)

and

http://retis.sssup.it/~giorgio/rts-MECS.html

(slides by Buttazzo, associated with Hard Real-Time Computing Systems, 3rd Edition, Springer, 2011)

which has been edited to suit the needs of this course.

The slides are authorized for personal use only.

Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.

Objectives

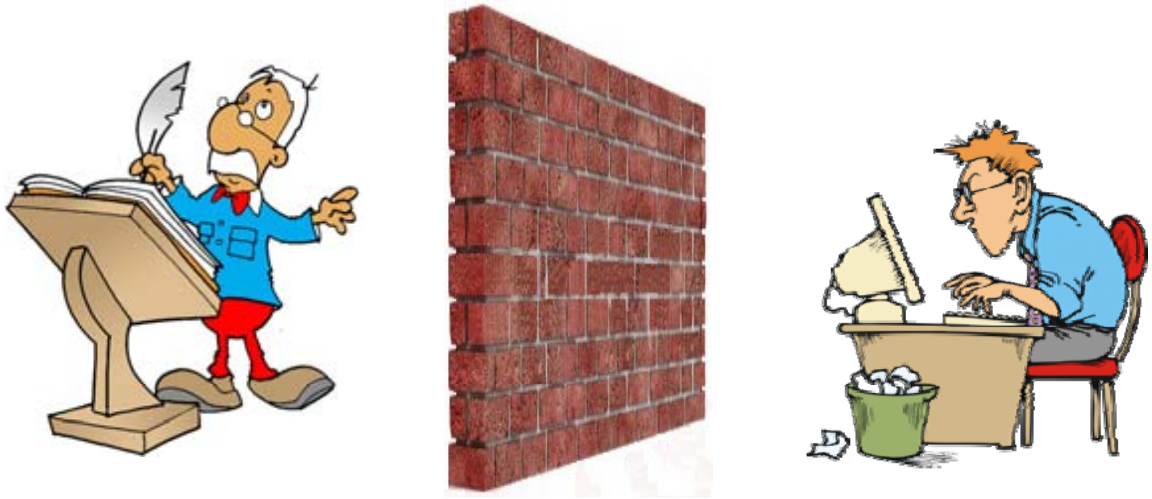
- Study software methodologies to support **time critical** systems:
  - Study software methodologies and algorithms to increase **predictability** in (embedded) computing system…

    …consisting of several **concurrent activities**…

    …subject to **timing constraints**
  - Learn how to **model** and **analyze** a real-time application to predict worst-case response times and **verify its feasibility** under a set of constraints

Control and Implementation

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



- Control guys typically assume a computer with infinite resources and computational power. In some case, computation is modeled by a fixed delay Δ.

Control and Implementation

- In reality, a computer:
  - has **limited** resources
  - **finite** computational power (**non null execution times**)
  - executes several **concurrent** activities
  - introduces **variable** delays (often **unpredictable**)

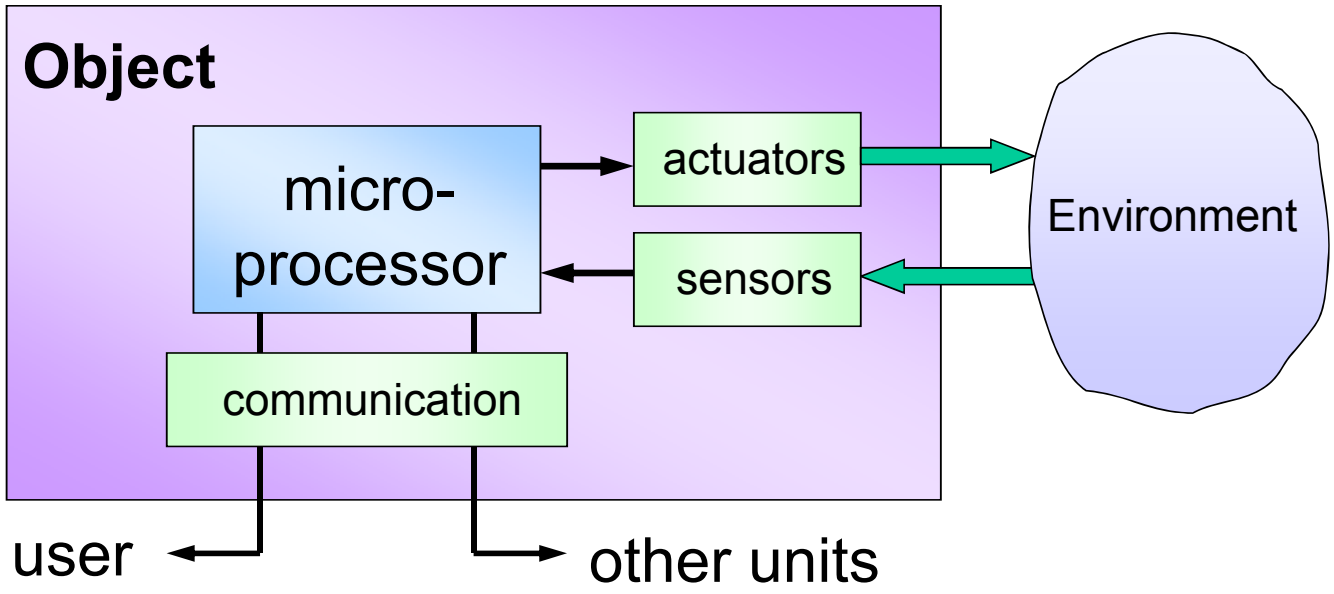
- Modeling such factors and taking them into account in the design phase allows a significant improvement in performance and reliability

Definitions
and
Sample Applications

Retis
Real-Time Systems Laboratory

Embedded System

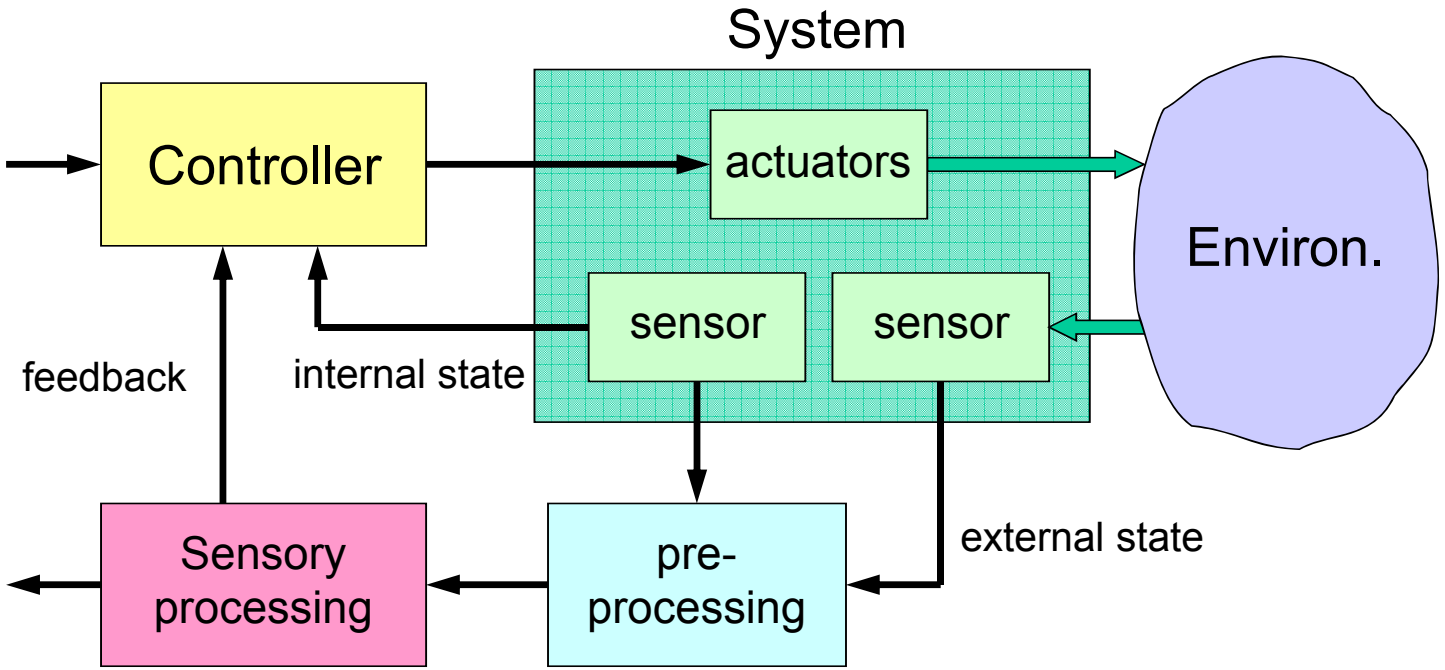
- A computing system hidden in an object to control its functions, enhance its performance, manage the available resources and simplify the interaction with the user.



Control System Components

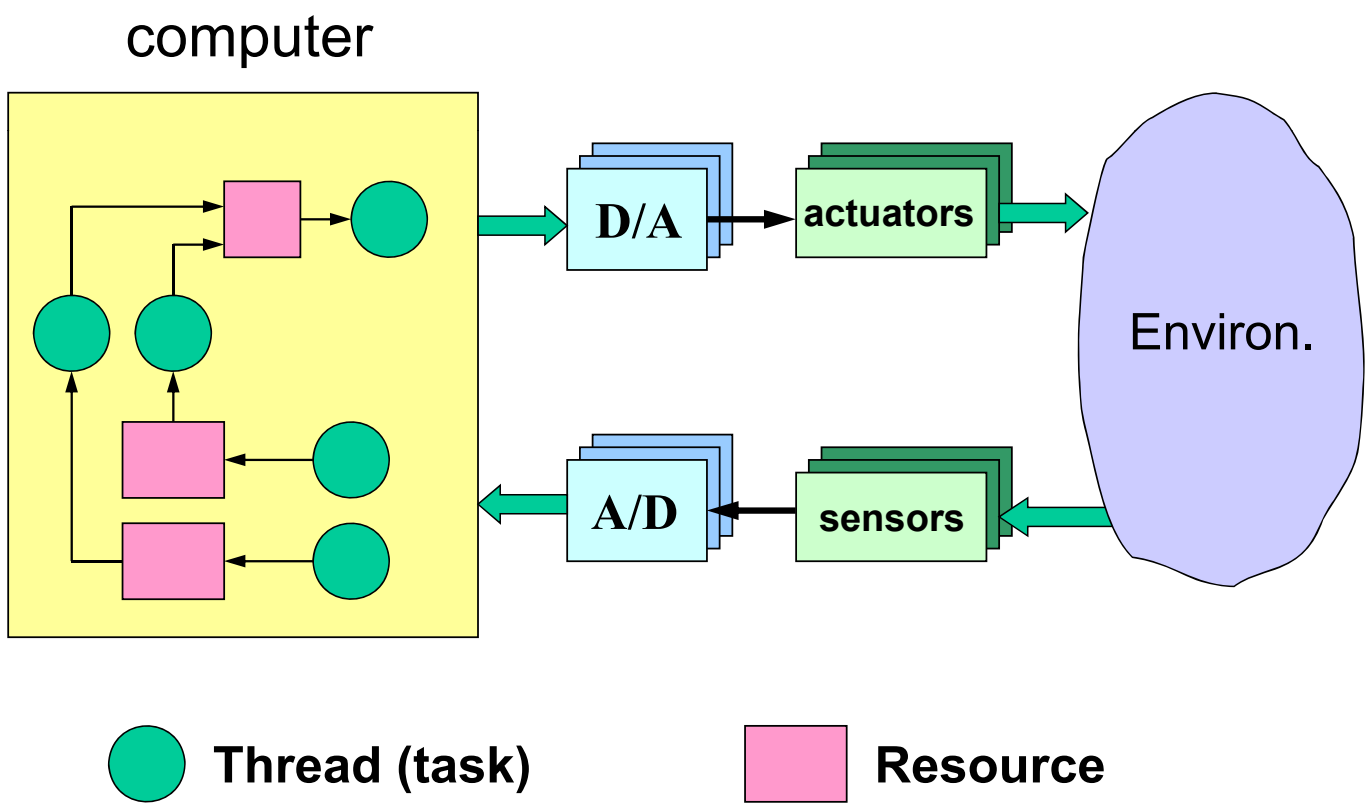
- In every control application, we can distinguish 3 basic components:
  - The **system** to be controlled
    - may include sensors and actuators
  - The **controller**
    - sends signals to the system according to a predetermined control objective
  - The **environment** in which the system operates

A Typical Control System



- Other activities
  - filtering, classification, data fusion, recognition, planning

Software Vision

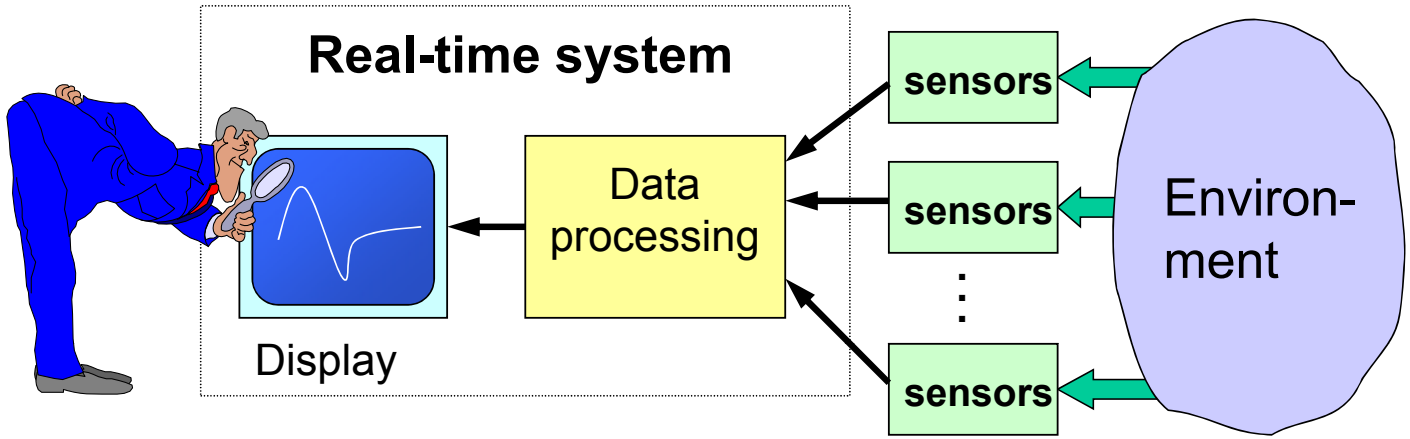


Types of Control Systems

- Depending on the system-environment interactions, we can distinguish among 3 types of control systems:
  - **Monitoring systems**
    - do not modify the environment
  - **Open-loop control systems**
    - loosely modify the environment
  - **Closed-loop control systems**
    - tight interaction between perception and action

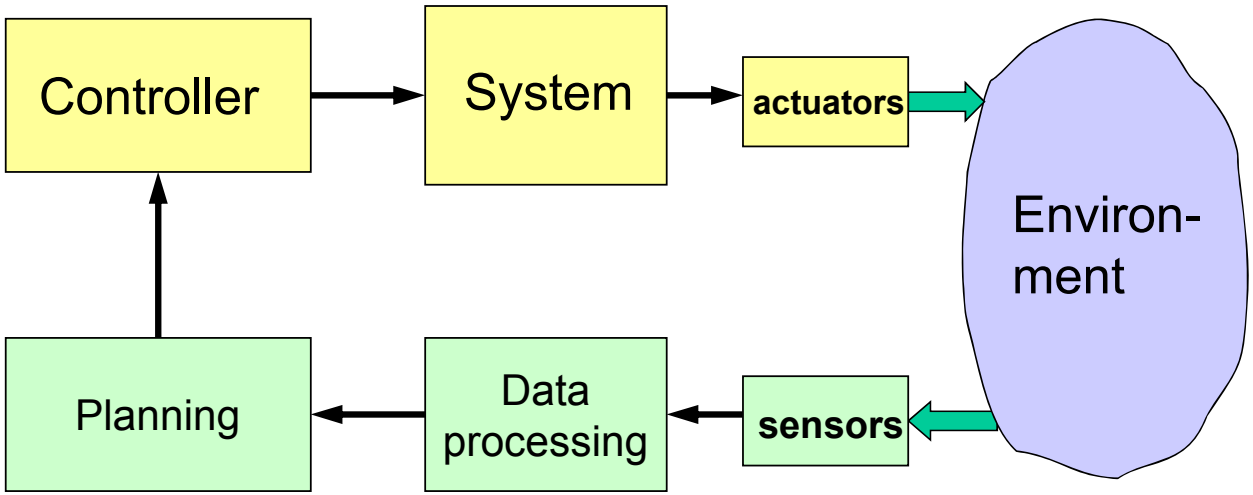
Monitoring Systems

- Do not modify the environment
  - surveillance systems, air traffic control



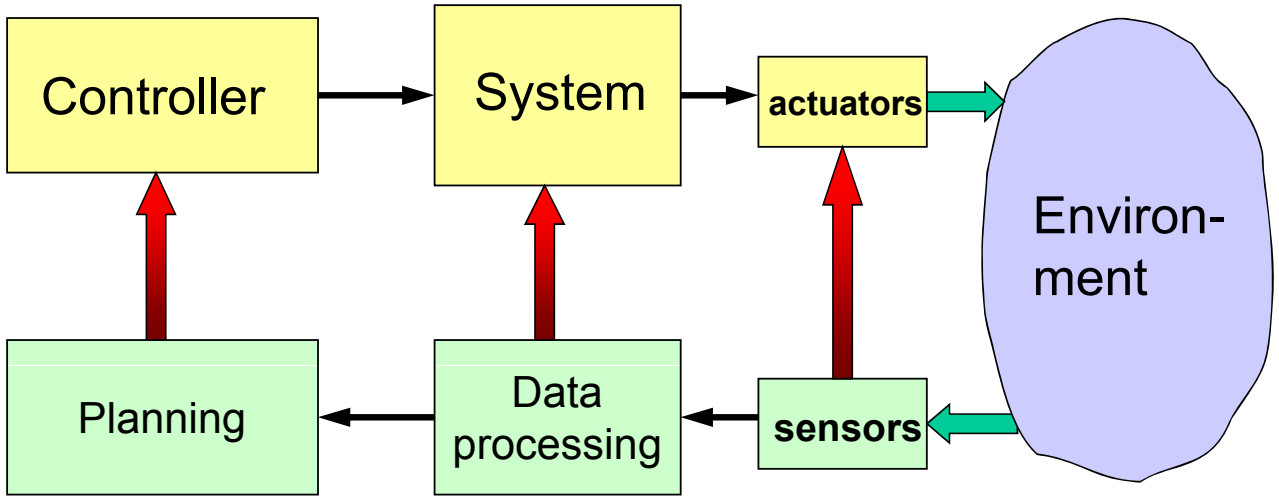
Open-Loop Control Systems

- Sensing and control are loosely coupled
  - Assembly robots, sorting robots



Closed-Loop Control Systems

- Sensing and control are tightly coupled
  - Flight control systems, military systems, living beings

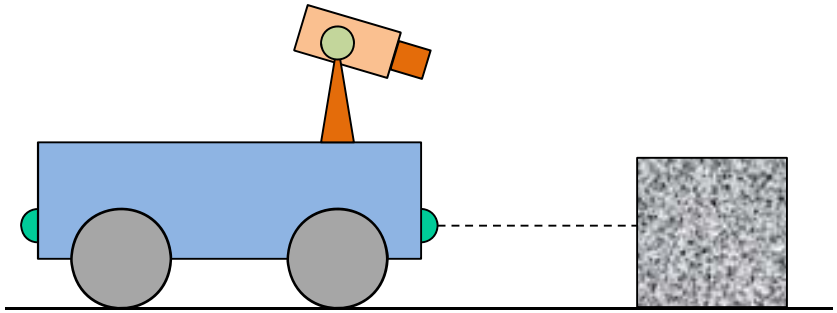


Implications

- The tight interaction with the **environment** requires the system to react to events within precise timing constraints
- Timing constraints are **imposed** by the dynamics of the environment
- The **operating system** must be able to execute tasks within timing constraints

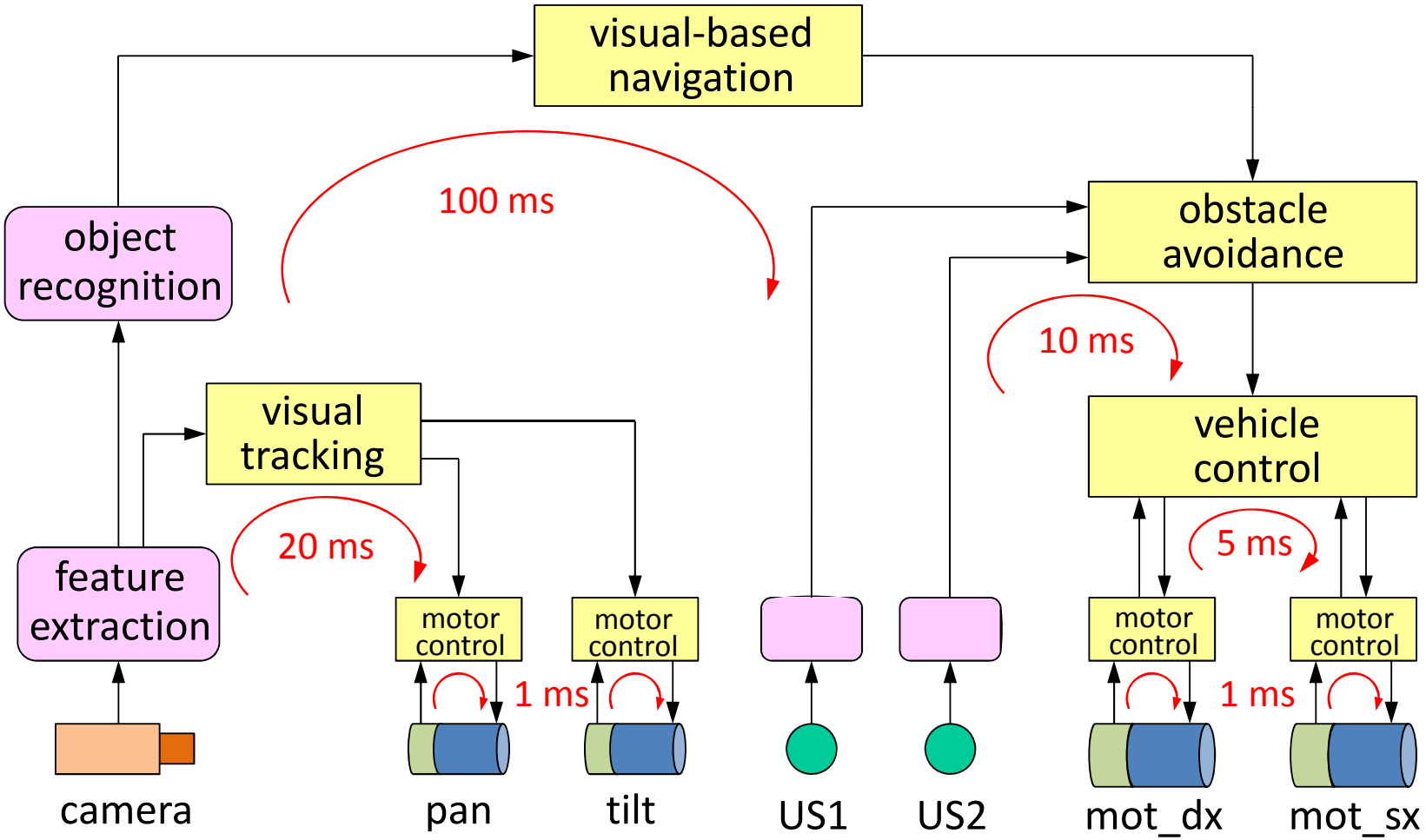
A Robot Control Example

- Consider a robot equipped with:
  - two actuated wheels
  - two proximity (US) sensors
  - a mobile (pan/tilt) camera
  - a wireless transceiver



- Goal:
  - follow a path based on visual feedback
  - avoid obstacles
  - send complete robot status every 20 ms

Hierarchical Control



Design Requirements

- **Modularity**: a subsystem must be developed without knowing the details of other subsystems (team work)
- **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)

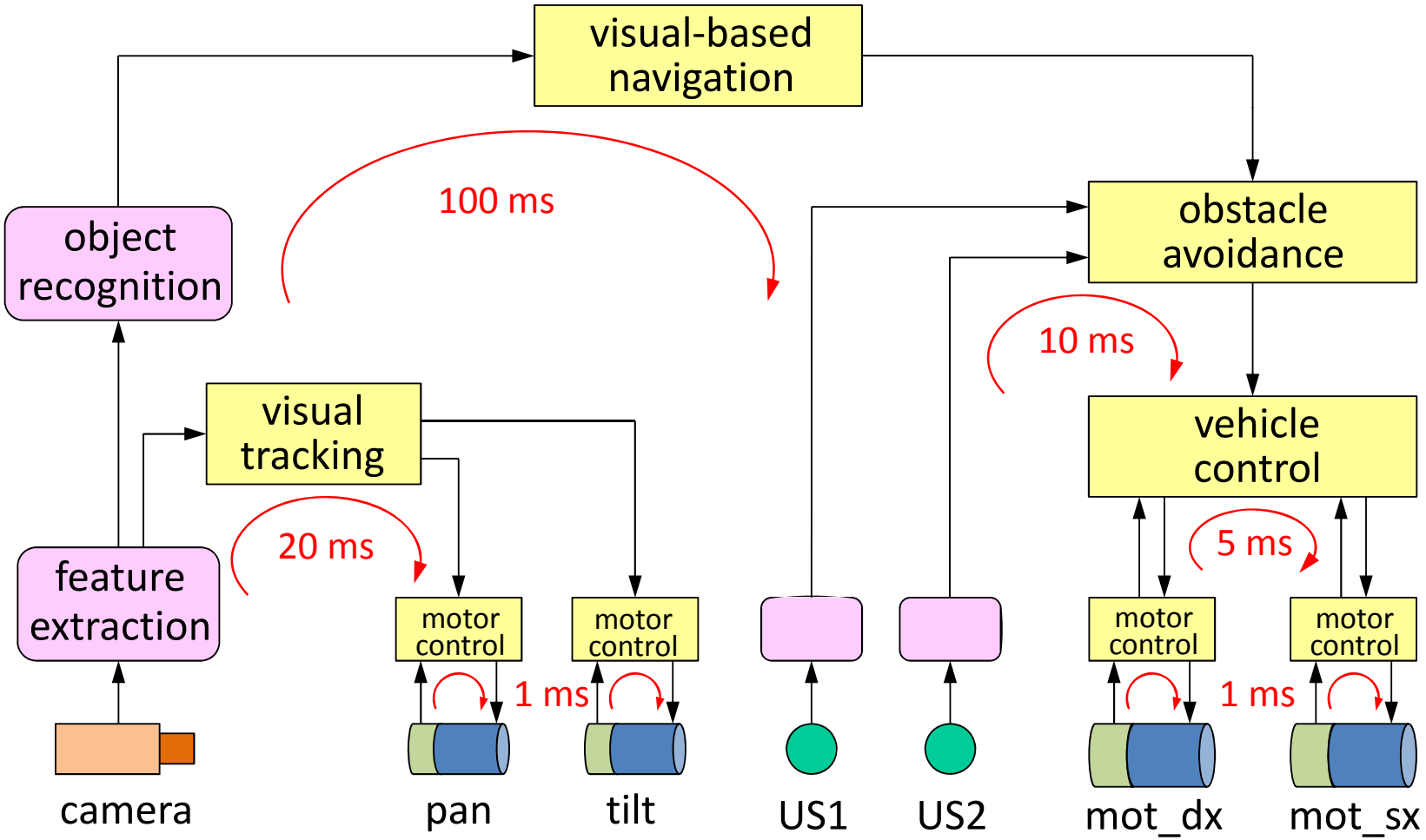
Operating System Requirements

- **Timeliness**: results must be correct not only in their value but also in the time domain
  - provide kernel mechanism for time management and for handling tasks with explicit timing constraints and different criticality
- **Predictability**: system must be analyzable to predict the consequences of any scheduling decision
  - if some task cannot be guaranteed within time constraints, system must notify this in advance, to handle the exception (plan alternative actions)
- **Efficiency**: operating system should optimize the use of available resources (computation time, memory, energy)
- **Robustness**: must be resilient to peak-load conditions
- **Fault tolerance**: single software/hardware failures should not cause the system to crash
- **Maintainability**: modular architecture to ensure that modifications are easy to perform

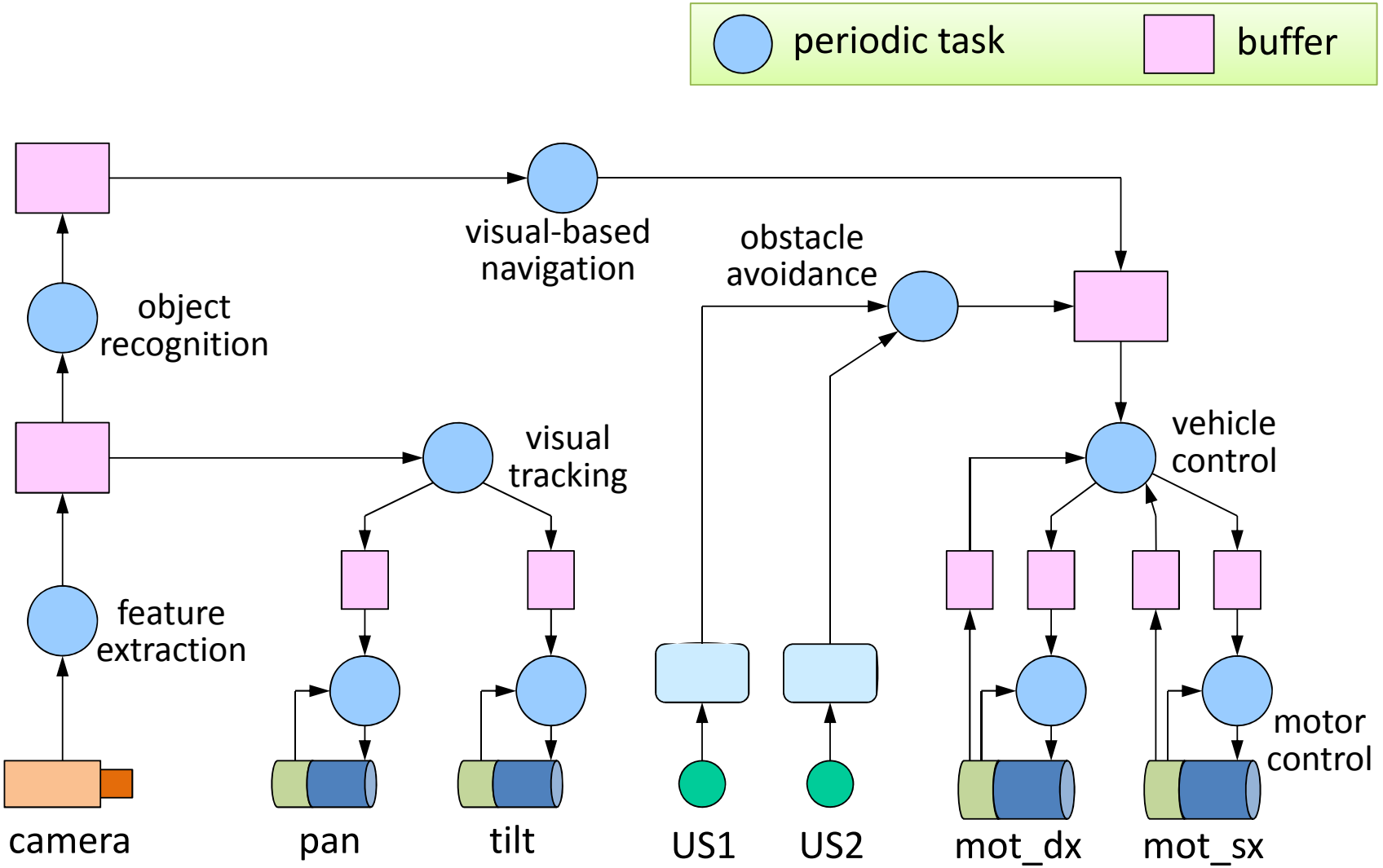
Modularity

- Modularity can be achieved by:
  - partitioning the system into a set of subsystems, each managed by one or more **computational tasks**
  - the definition of precise **interfaces** between tasks, each specifying:
    - data exchanged with the other tasks (input and output)
    - functionality of the task (what it has to do)
    - validity assumptions (e.g., admissible ranges)
    - performance requirements (priority, period, deadline, jitter)
- **Asynchronous communication mechanisms**

Control View



Software View

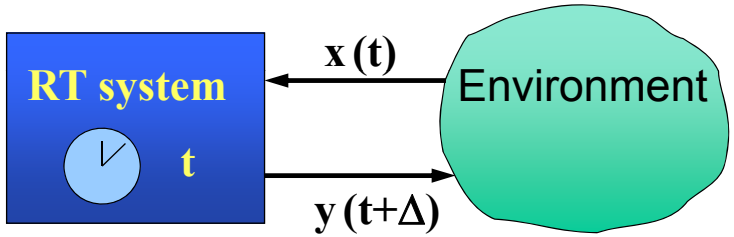


RTOS Responsibilities

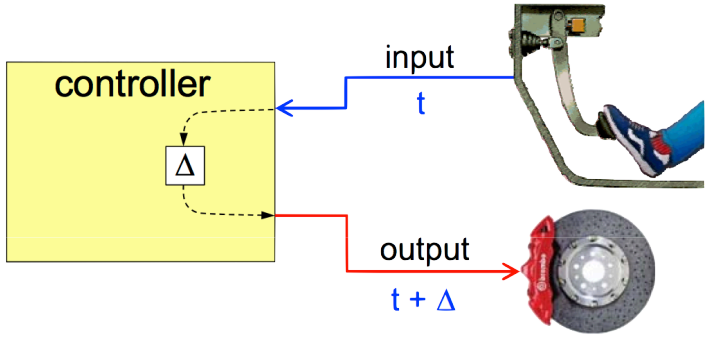
- The Real-Time Operating System (RTOS) is responsible for:
  - managing the concurrent execution of the various activities
    - **concurrent tasks**
  - decide the order of execution of the tasks, satisfying the specified requirements
    - **scheduling**
  - solving possible timing conflicts during the access of shared resources
    - **critical sections**
  - manage the timely execution of asynchronous events
    - **interrupts**

What is a Real-Time System?

- A computer system able to **respond to events within precise timing constraints**



- A system where the **correctness** depends not only on the **output** values, but also on the **time** at which results are produced



Typical Objection

- *"It is not worth to invest in Real Time theory, because computer speed is increasing exponentially, and all timing constraints can eventually be handled."*

  Answer

- Given an arbitrary computer speed, we must always **guarantee** that timing constraints can be met. Testing is **NOT** sufficient

Real-Time ≠ Fast

- A real-time system is **not** a fast system
- Speed is always relative to a specific environment
- Running faster is good, but does not guarantee a correct behaviour.

- The objective of a real-time system is to guarantee the timing behaviour of each individual task
- The objective of a fast system is to minimize the average response time of a task set. But…
  - Don't trust **average** when you have to guarantee **individual** performance

Sources of Nondeterminism

- **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

Design Methodologies

- Traditional approach: **empirical techniques**
  - assembly programming
  - timing through dedicated timers
  - control through driver programming
  - priority manipulations

- Many **disadvantages**!
  - tedious programming, heavily relies on programmer's ability
  - difficult code understanding (readability × efficiency = *k*)
  - difficult software maintainability
    - MLOC, understanding takes more than rewriting => bug prone
  - difficult to verify timing constraints without OS & language support

Implications

- Dangerous way of programming real-time applications
- May work in most situations, but high risk of failure
- When the system fails, it is very difficult to understand why
- **Low reliability**
- Many famous failures
  - First flight of the Space Shuttle, 1979 (transient overload at initialization)
    - probability of failure ~1.5%
  - Scud missile on Dhahran, 1993 (delay due to interrupt handling)
    - program flow depends on sensory data, cannot be fully replicated
    - testing is not enough
  - Ariane 5, 1996 (integer overflow in inertial reference system routine)
    - Environment
  - Mars Pathfinder, 1997 (priority inversion, see Silberschatz)

Take-Home Message

- Tests, although necessary, allow only a partial verificaiton of system's behaviour
- Predictability must be improved at the kernel level
- Overload handling and fault-tolerance
- Critical systems must be designed by making **pessimistic assumptions**…

- …Murphy's laws
  - If something can go wrong, it will go wrong
  - If a system stops working, it will do it at the worst possible time
  - Sooner or later, the worst possible combination of circumstances will happen…

Achieving predictability

1. **DMA**

- Cycle stealing

- Possible solution: **time-slice method**
  - each memory cycle split into two adjacent time slots
    - one reserved for the CPU, the other for the DMA device
  - more expensive than cycle stealing
    - but more predictable

Achieving predictability

2. **Cache**

- Hit ratio
  - 80% of times: hits
  - 20% of times: performance degrades
  - Preemptive systems destroy locality
  - Cache-related preemption delay difficult to precisely estimate

Achieving predictability

3. **Interrupts**

- Source: peripheral devices
- Can introduce unbounded delays
- Handling routines with static priorities
  - generic OS: I/O have real-time constraints
  - RTOS: a control process could be more urgent than interrupt handling

- 3 different approaches

Achieving predictability

3. Interrupts

- Can introduce unbounded delays

A. Disable all interrupts, except the one from the timer; devices handled by application tasks using polling
   - + predictability, kernel-independent; - efficiency

B. Disable all interrupts except the one from the timer; manage devices via periodic kernel routines
   - + encapsulation; - overhead

C. Leave all interrupts enabled; minimize drivers' size (only activates device management task)
   - + no busy waiting; - (small) unbounded overhead due to drivers

Achieving predictability

4. **System Calls**

- Could be difficult to evaluate worst-case execution time of each task

- All system calls should be characterized by bounded execution time
- Desirable that system calls be preemptable

Achieving predictability

5. **Semaphores**

- Priority inversion

- Must be avoided!
- Several methods:
  - Basic priority inheritance
  - Priority ceiling

Achieving predictability

6. **Memory management**

- Demand paging

- Solution: static partitioning
  - memory segmentation rule with fixed memory management scheme
- \+ predictability, - flexibility in dynamic environments

Achieving predictability

7. **Programming language**

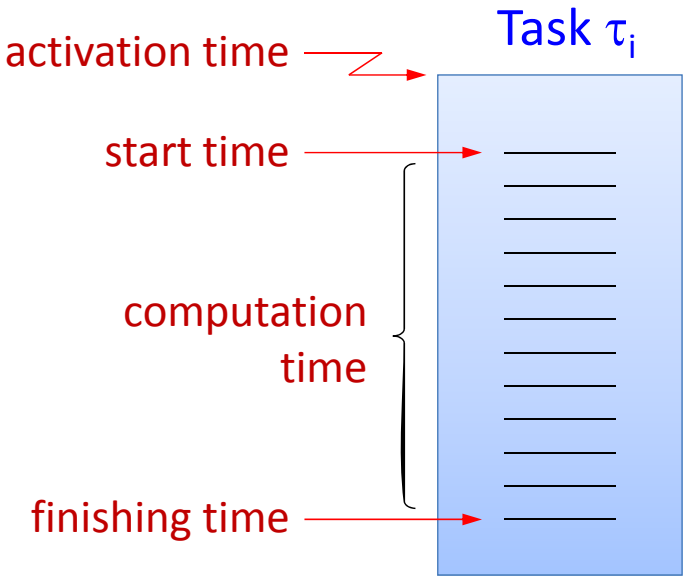
- Dynamic data structures
- Recursion
- Cycles

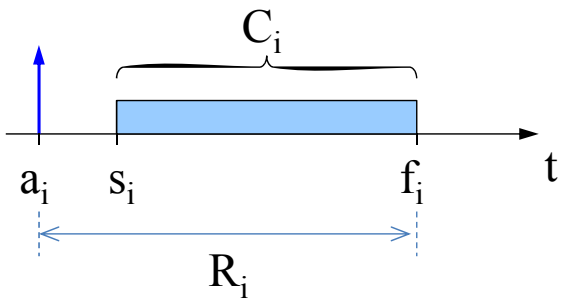
- High-level languages for programming hard real-time applications
  - Real-Time Euclid
  - Real-Time Concurrent C

Modeling
Real-Time Activities

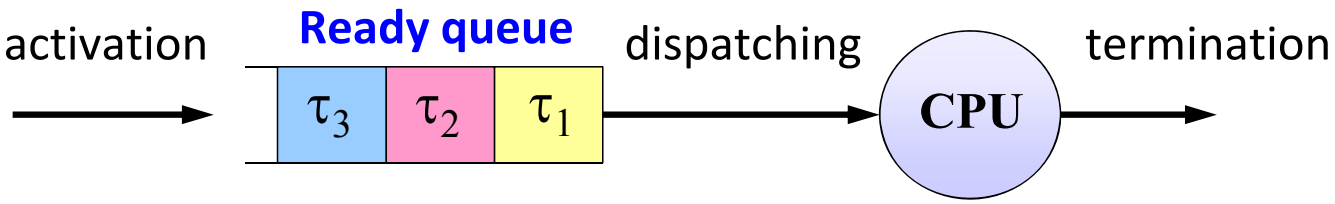
Task

- Sequence of instructions that in the absence of other activities is continuously executed by the processor until completion.





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



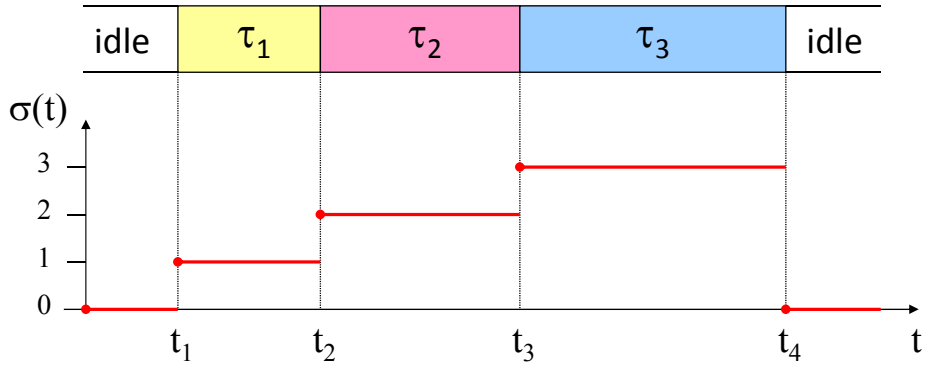
- *Note: "activation" = "arrival" = "request" = "release" time*

Schedule

- A particular assignment of tasks to the processor that determines the task execution sequence. Formally:

  Given a task set $\Gamma=\{ \tau_1, \ldots, \tau_n \}$, a **schedule** is a function σ: $\mathbf{R^+ \rightarrow N}$ that associates an integer k to each **time slice** $[t_i, t_{i+1})$ with the meaning:

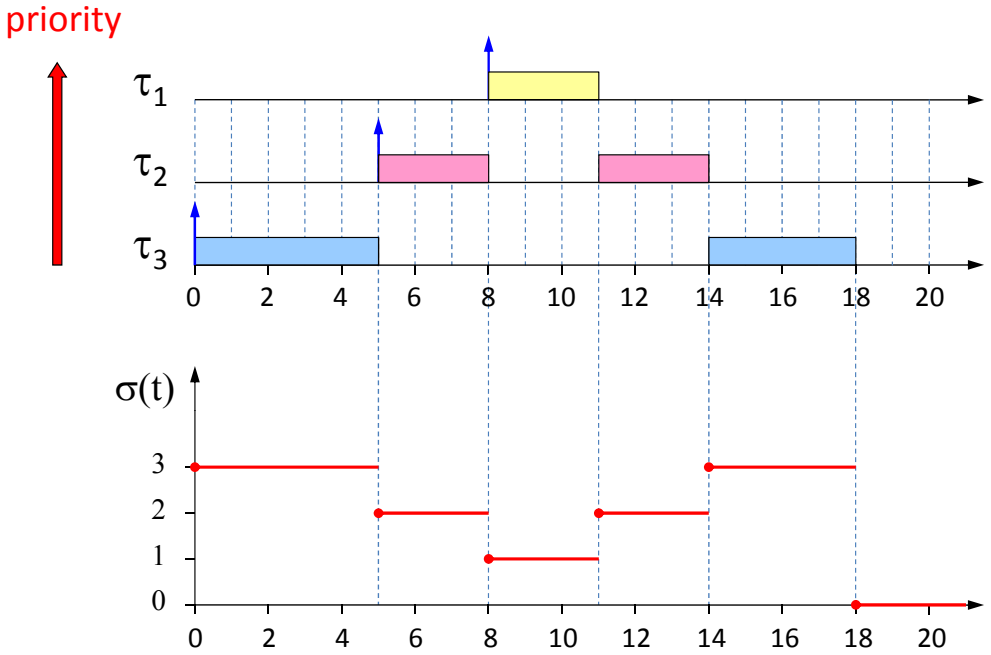
  - $k=0$: in $[t_i, t_{i+1})$ the processor is **idle**
  - $k>0$: in $[t_i, t_{i+1})$ the processor **executes** τ_k



- At times t_1, t_2,…: **context switch**

Preemptive Scheduling

- A running tasks may be suspended and placed in the ready queue
  - + Exception handling: timely response to issues
  - + Different levels of criticality: preemption executes most critical tasks
  - + Higher efficiency (CPU utilization)
  - - Destroys program locality
  - - Introduces runtime overhead



Real-Time Systems Laboratory

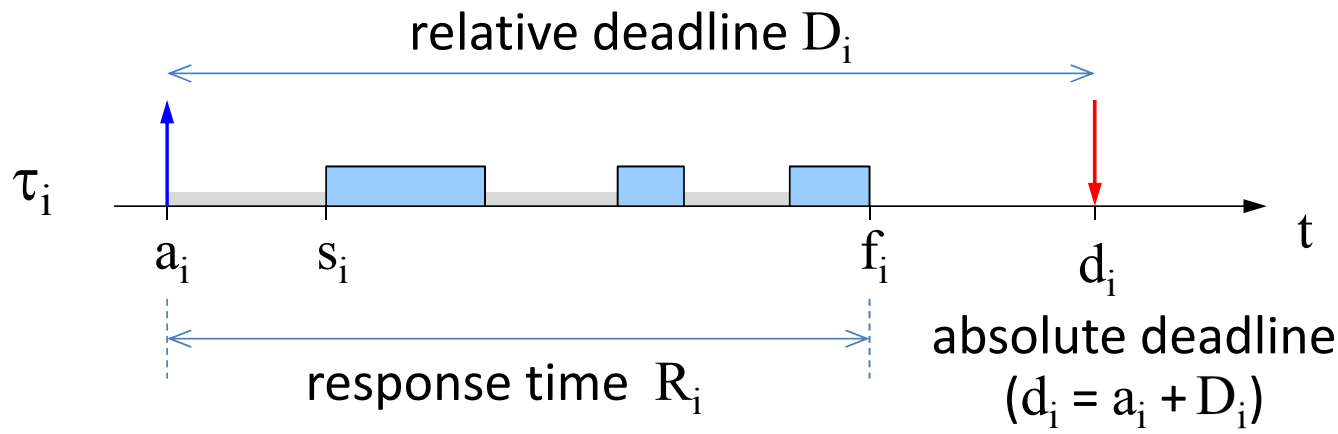
Definitions

- Definition: **feasible schedule**
  - A schedule σ is said to be feasible if ***all*** the tasks can complete according to a set of specified constraints.

- Definition: **schedulable set of tasks**
  - A set of tasks Γ is said to be schedulable if there exists ***at least one*** algorithm that can produce a feasible schedule for it.

Real-Time Task

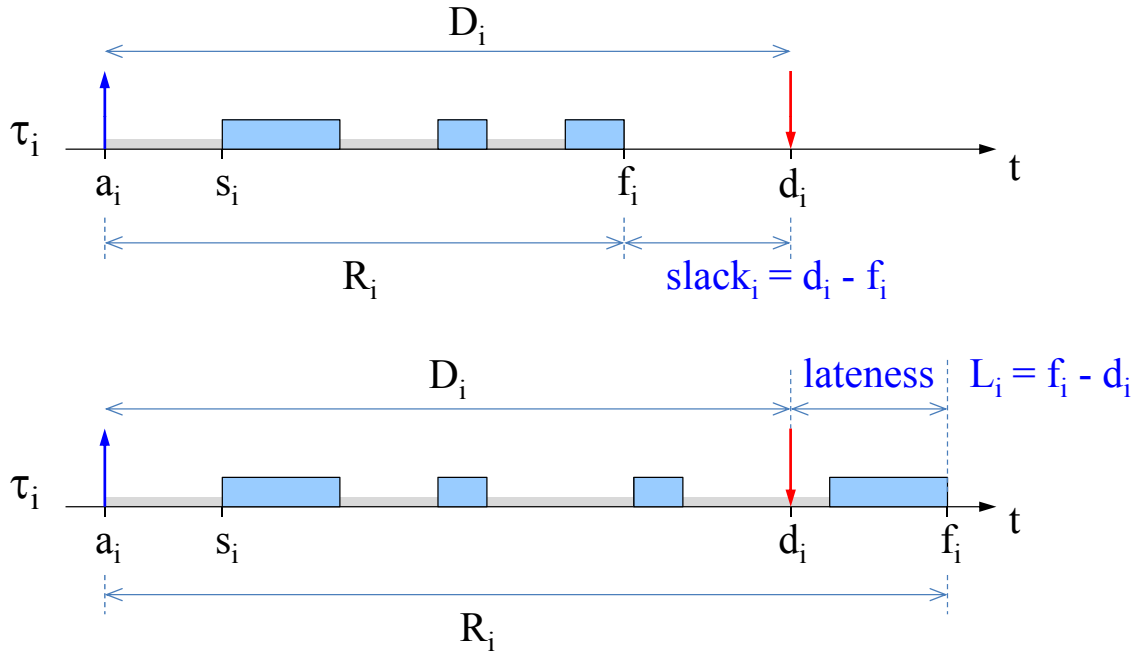
- It is a task characterized by a timing constraint on its response time, called **deadline**:



- "Completion time" = $f_i - s_i = R_i - (s_i - a_i)$

- Definition: **feasible task**
  - A real-time task τ_i is said to be feasible if it completes within its absolute deadline, that is, if $f_i \leq d_i$, or, equivalently, if $R_i \leq D_i$.

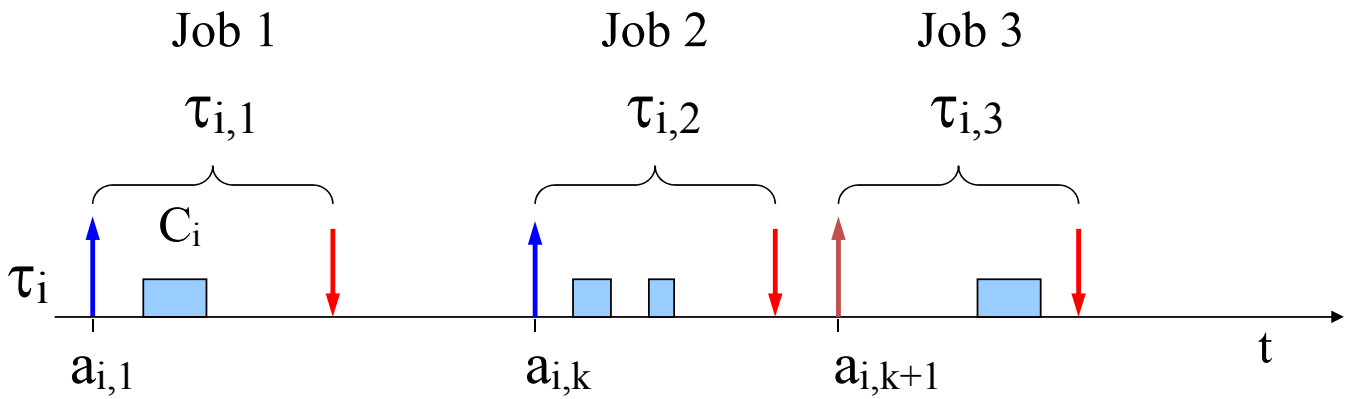
Slack and Lateness



- "Slack" if lateness is negative (task completes before deadline)
- "Laxity" or "slack time" $X_i = d_i - a_i - C_i$
- "Tardiness" or "exceeding time" $E_i = max(0, L_i)$

Tasks and Jobs

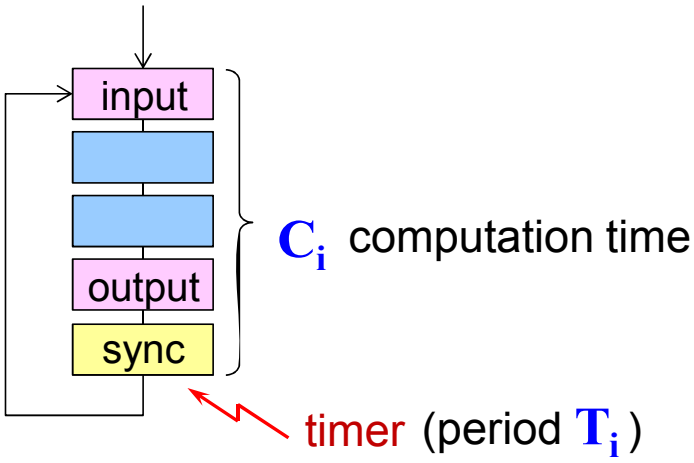
- A task running several times on different input data generates a sequence of instances (**jobs**):



Activation Mode

- **Time-driven**
  - **Periodic tasks** (τ_i)
  - The task is automatically activated by the operating system at predefined time instants.
- **Event-driven**
  - **Aperiodic tasks: "jobs"** (J_i)
  - The task is activated at an event arrival or by explicitly invocating a system call.

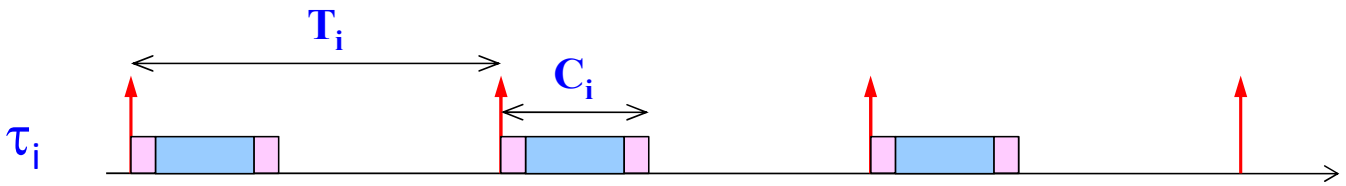
Periodic Task



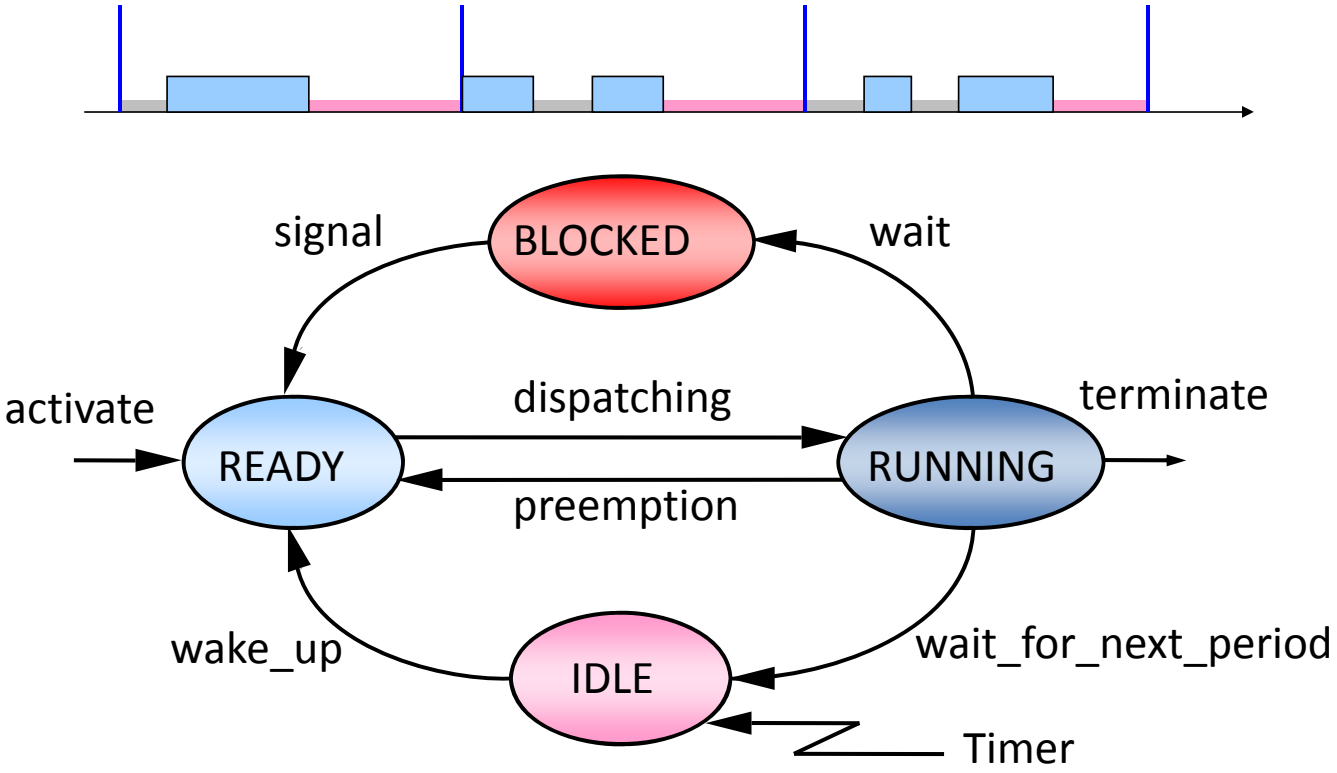
$$U_i = \frac{C_i}{T_i}$$

utilization factor

- A **periodic task** τ_i generates an infinite sequence of **instances** or **jobs** (same code on different data): $\tau_{i,1}$, $\tau_{i,2}$, …, $\tau_{i,k}$, …

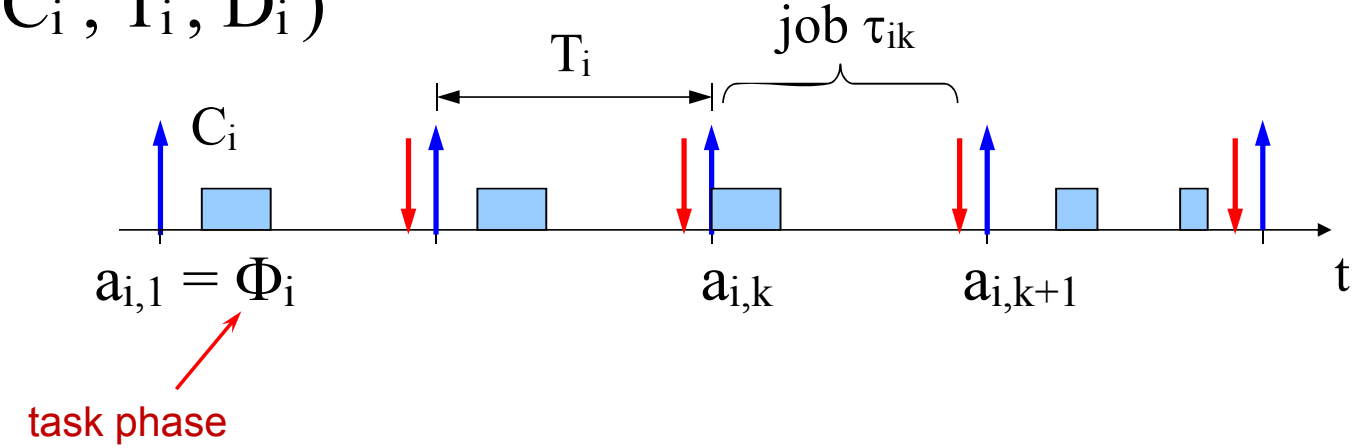


The IDLE State



Periodic Task

$\tau_i (C_i , T_i , D_i)$



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

often
$D_i = T_i$

Exercise

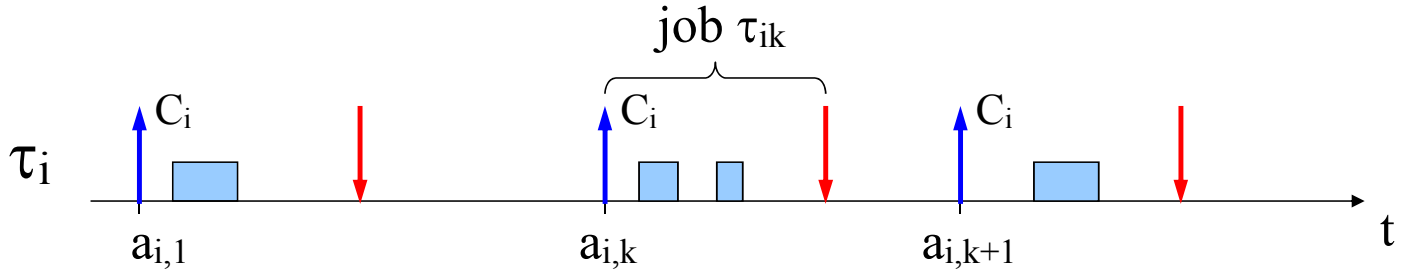
- Consider a periodic task $\tau_1(C_1,T_1,D_1)$ with phase Φ_1, where:
  - C_1 = 10 ms, T_1 = 50 ms, D_1 = 25 ms, and Φ_1 = 100 ms

- What is τ_1's utilization factor?
- Is τ_1 feasible?
- What is $\tau_{1,1}$'s absolute deadline?
- What is $\tau_{1,1}$'s laxity?
- What is $\tau_{1,2}$'s release time?
- Can $\tau_{1,1}$ and $\tau_{1,2}$ have different laxity?
- Can $\tau_{1,1}$ and $\tau_{1,2}$ have different slack?
- If $\tau_{1,2}$'s slack is 10ms, what is $\tau_{1,s}$'s finishing time?
- What is $\tau_{1,2}$'s response time?
- With a 2-CPU machine, can $\tau_{1,2}$ and $\tau_{1,3}$ have the same release time?
- Can $\tau_{1,2}$ and $\tau_{1,3}$ have the same finishing time?

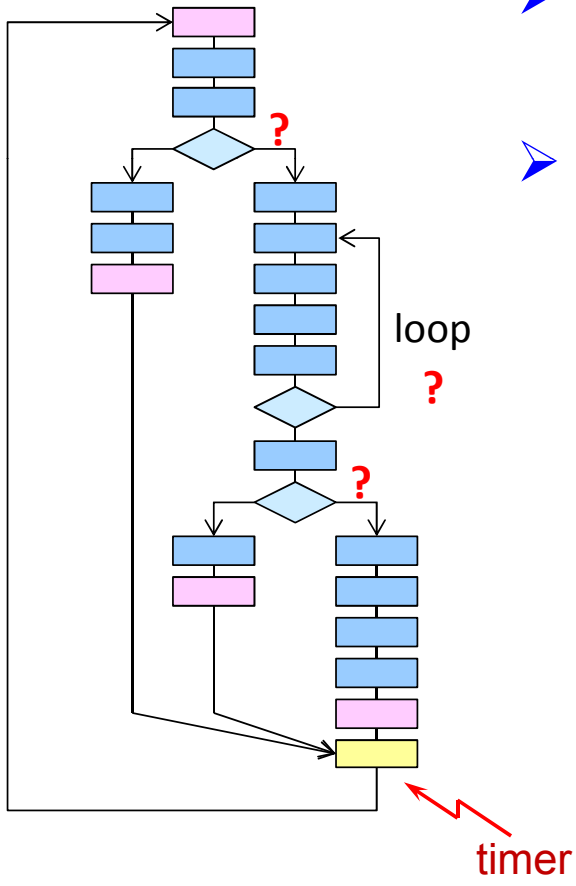
Aperiodic Task

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

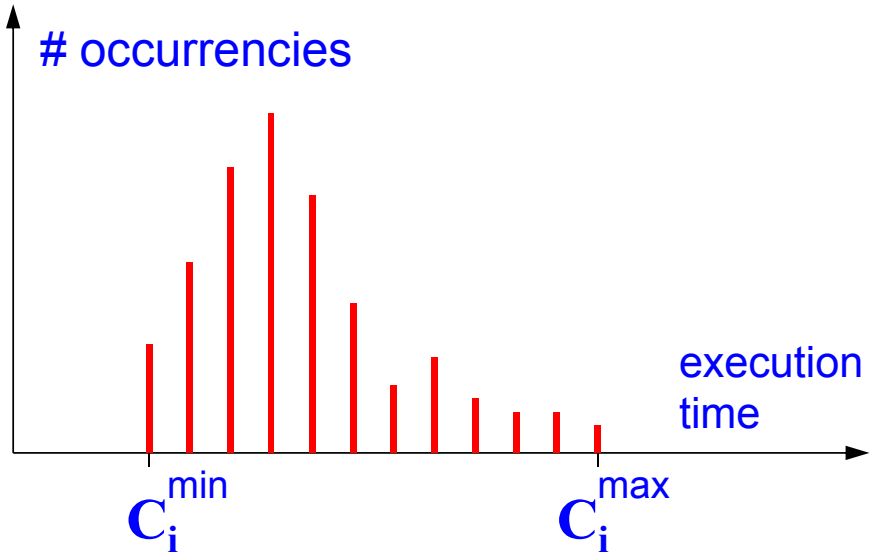
minimum interarrival time



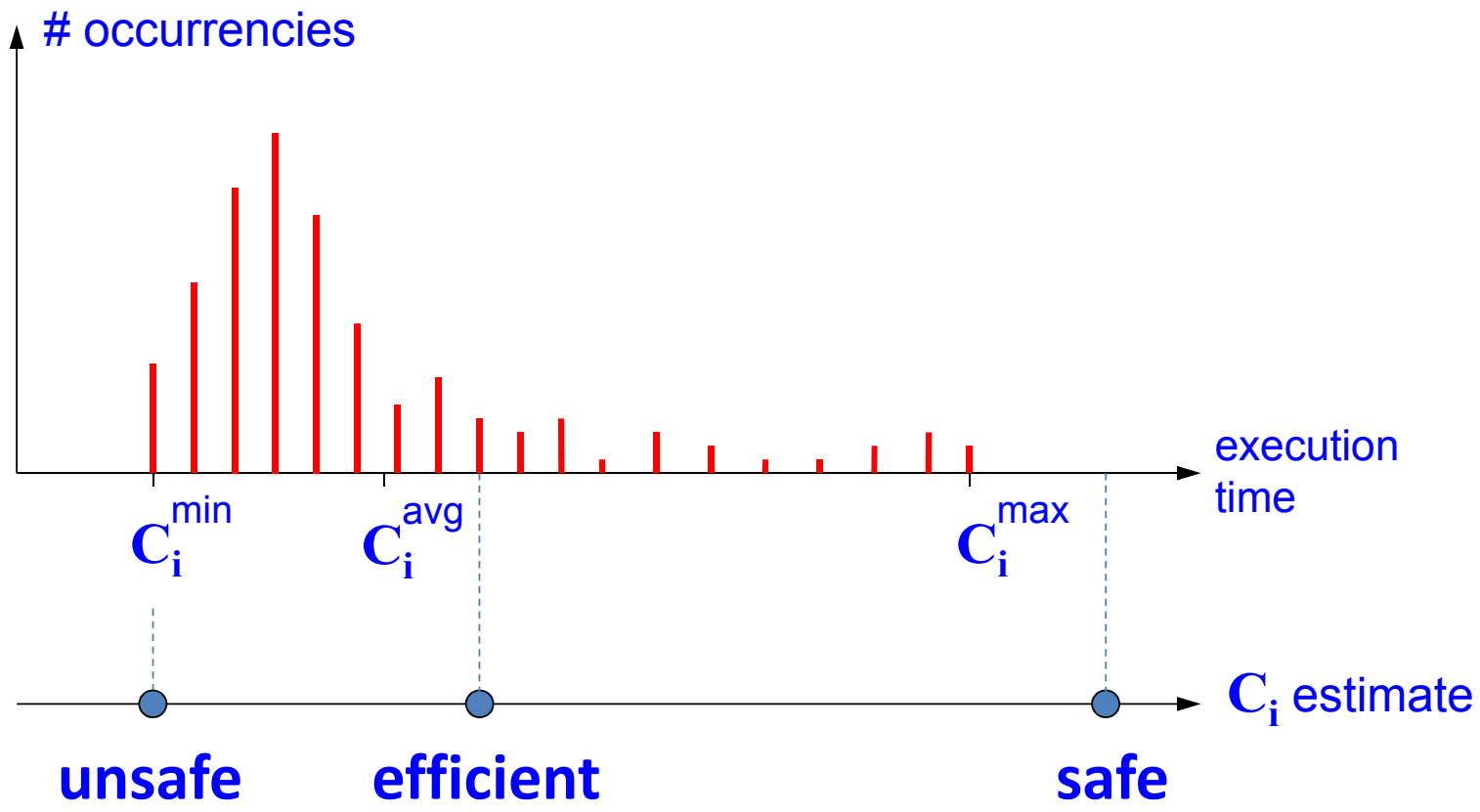
Estimating C_i is Not Easy



- Each job operates on different data and can take different paths.
- Even for the same data, computation time depends on the processor state (cache, prefetch queue, number of preemptions).



Predictability vs. Efficiency

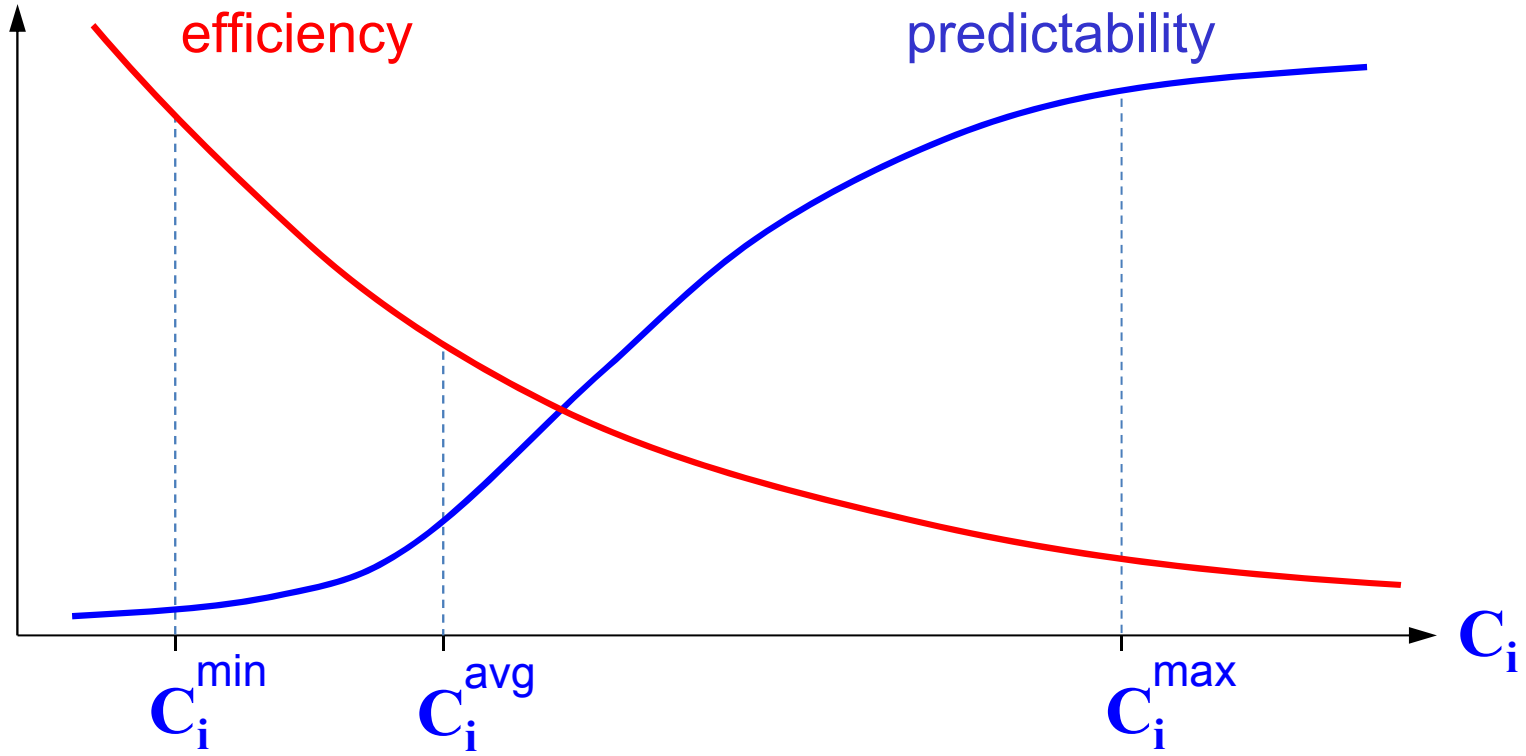


Predictability vs. Efficiency

non-RT task

SOFT task

HARD task



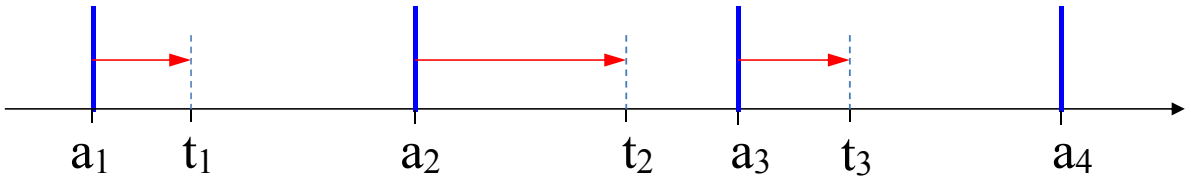
Criticality and Value

- It is a parameter related to the consequences of missing the deadline
  - **Hard**: missing deadline may have **catastrophic consequences**
  - *Hard Real-Time System* if it can handle hard tasks
    - sensory acquisition
    - low-level control
    - sensory-motor planning
  - **Soft**: missing a deadline causes **performance degradation**
    - reading data from the keyboard—user command interpretation
    - message displaying
    - graphical activities

- **Value**, v_i = the relative importance of a task wrt other tasks

Jitter

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

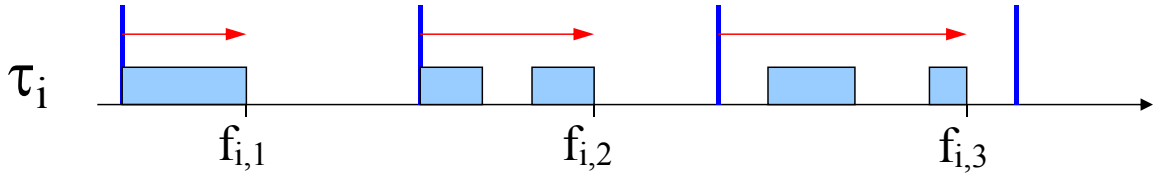


Absolute: $\max_k (t_k - a_k) - \min_k (t_k - a_k)$

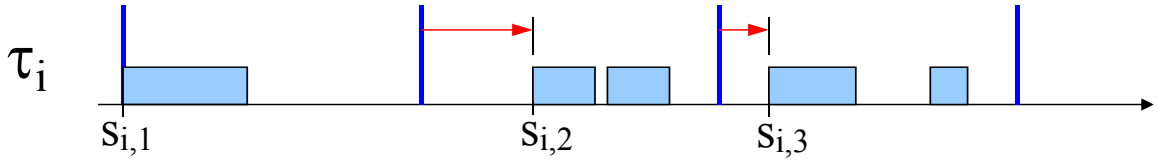
Relative: $\max_k \left| (t_k - a_k) - (f_{k-1} - a_{k-1}) \right|$

Types of Jitter

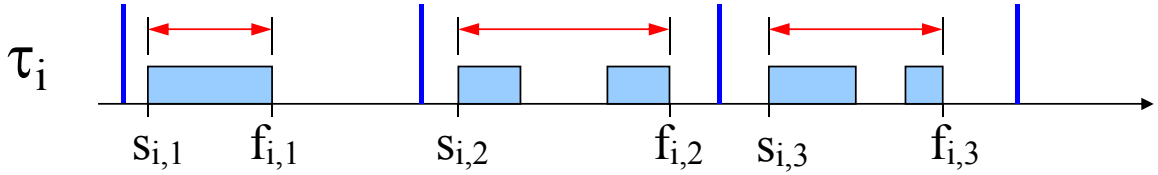
Finishing-time Jitter



Start-time Jitter



Completion-time Jitter (I/O Jitter)



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.

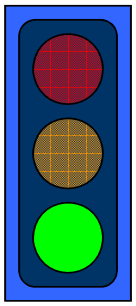
Explicit Timing Constraints

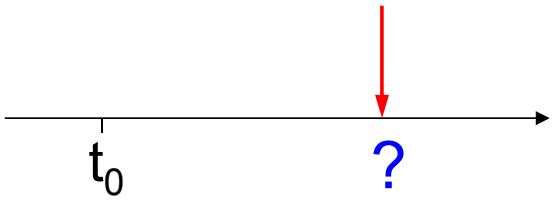
- Timing constraints can be explicit or implicit.
- **Explicit timing constraints**
  - Directly included in the system specifications.
- Example:
  - open the valve **in** 10 seconds
  - send the position **within** 40 ms
  - read the altimeter **every** 200 ms
  - acquire the camera **every** 20 ms

Implicit Timing Constraints

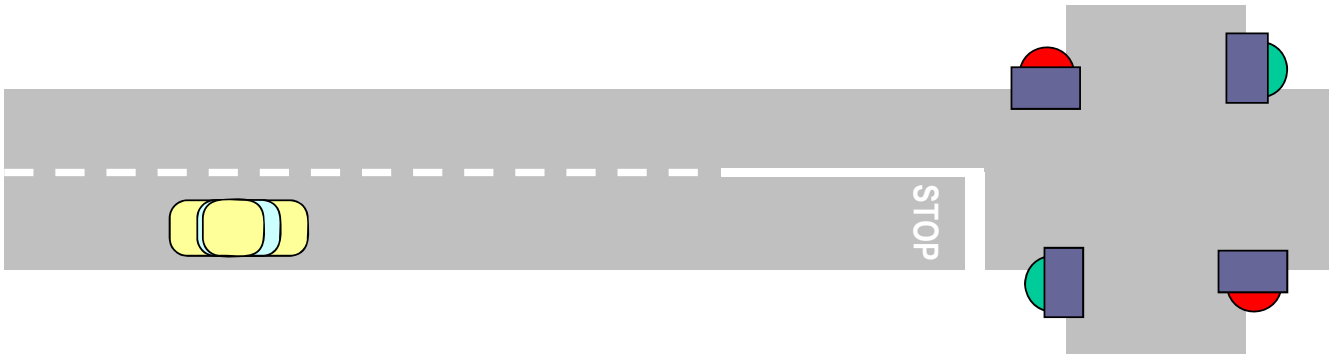
- They do not appear in the system specifications…
  - but need to be met in order to satisfy the performance requirements

- **Example**
  - What is the validity of a sensory data?





Computing the Yellow Duration

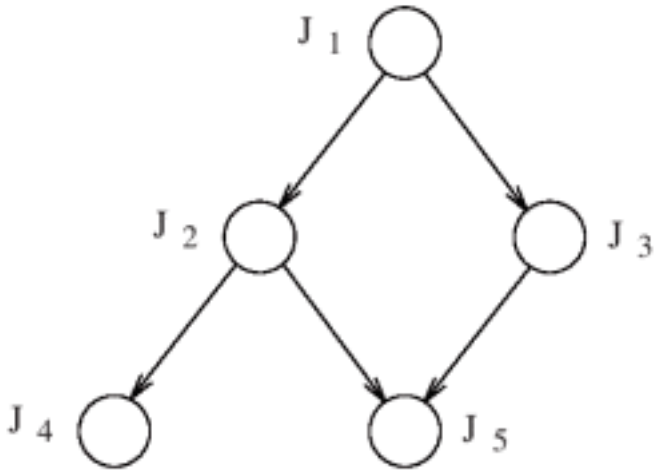


- $D \geq T_d + T_r + T_b$
  - T_d = Detection time
  - T_r = Reaction time
  - T_b = Braking time ~ v / .5 g

- T_d = .8s, T_r = .8s, v = 50 km/h (14 m/s) → D ≥ ???
- v_{max}???

Precedence Constraints

- Sometimes tasks must be executed with specific precedence relations, specified by a Directed Acyclic Graph (**Precedence Graph**):



$J_1 \prec J_2$

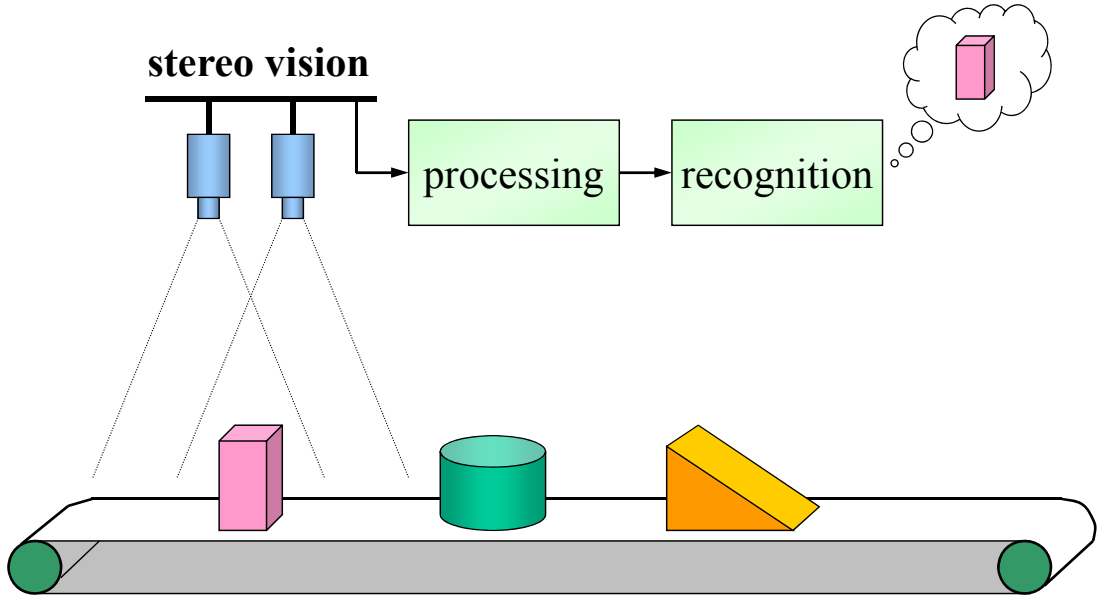
$J_1 \rightarrow J_2$

$J_1 \prec J_4$

$J_1 \not\rightarrow J_4$

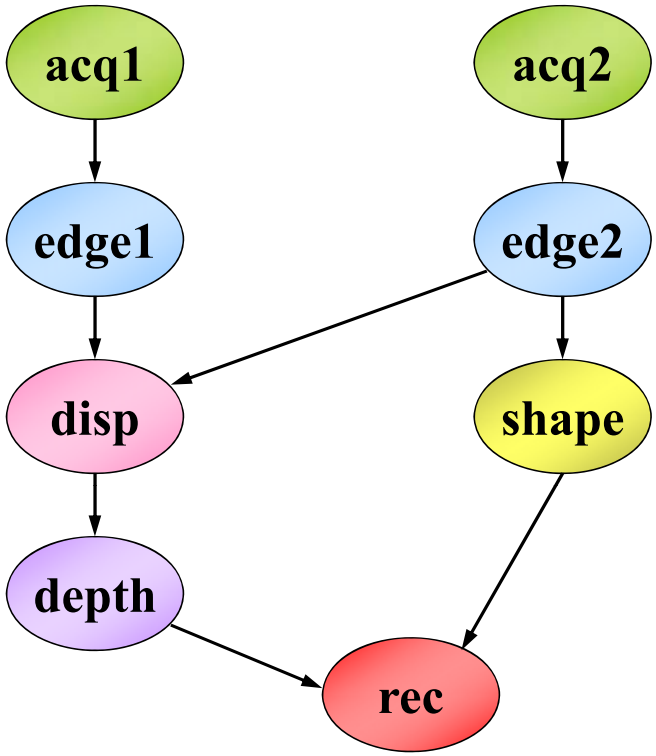
  - Immediate predecessor
  - Predecessor

Sample Application



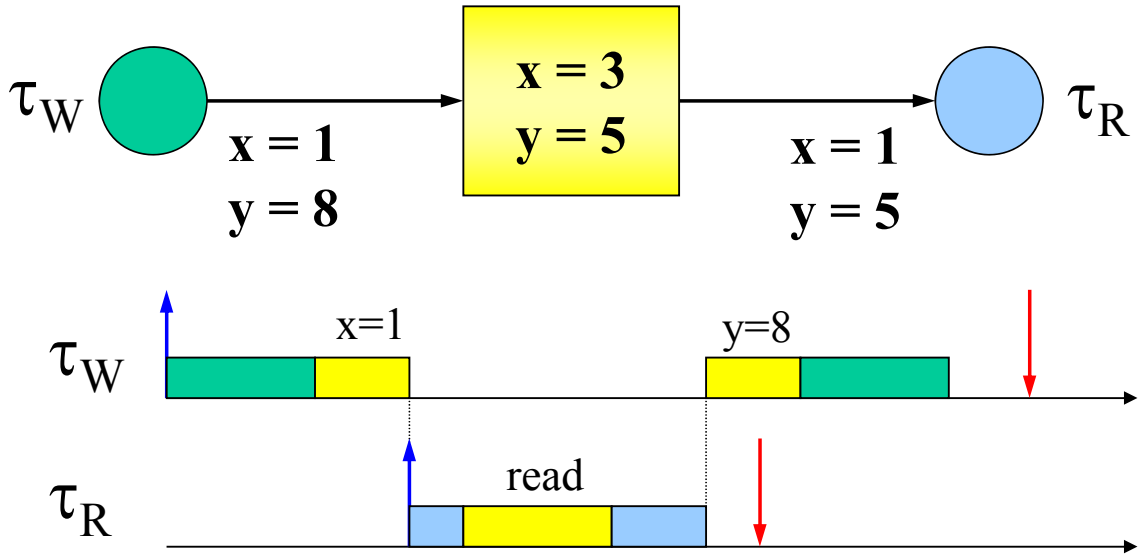
- Tasks:
  - Acquisition (*acq1*, *acq2*)
  - Edge detection (*edge1*, *edge2*)
  - Shape detection (*shape*), pixel disparities (*disp*)
  - Height determination (*height*), recognition (*rec*)
- Precedence graph?

Precedence Graph



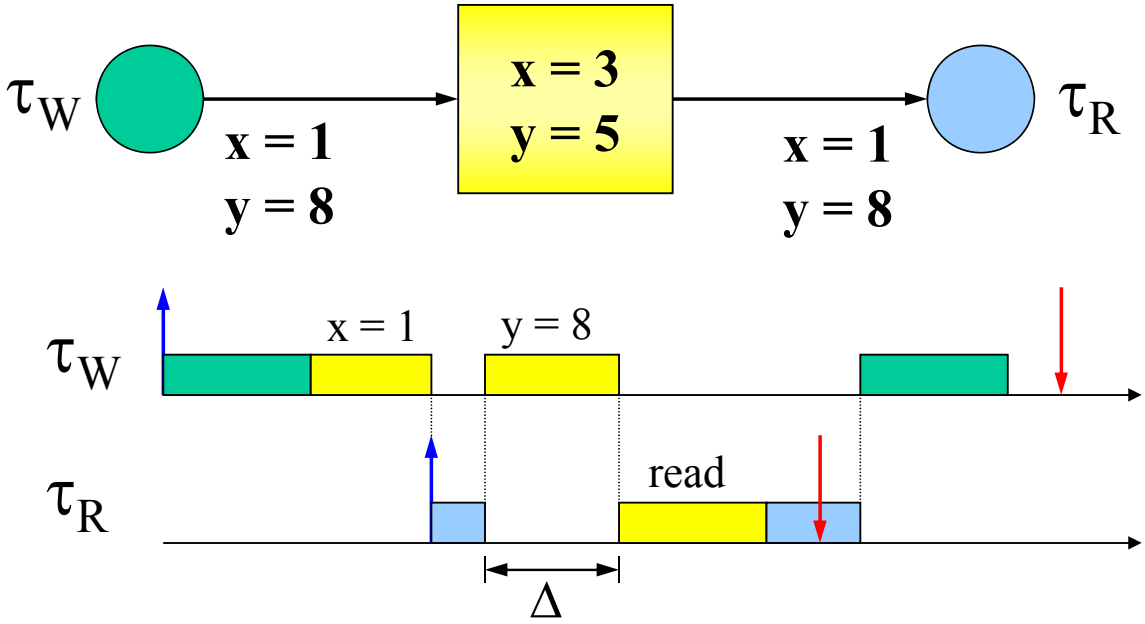
Resource Constraints

- To preserve data consistency, shared resources must be accessed in **mutual exclusion**:



Mutual Exclusion

- However, mutual exclusion introduces **extra delays**:



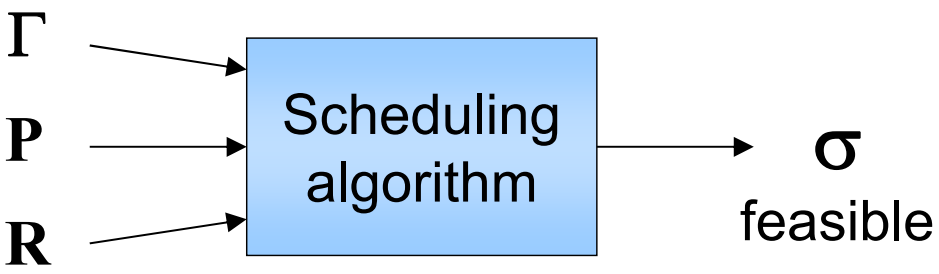
Definition of Scheduling Problems

Real-Time Systems Laboratory

General Scheduling Problem

- Given:
  - a set of *n* **tasks**, $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$
    - a **precedence** graph
    - a set of **timing constraints** associated with each task
  - a set of *m* **processors**, $P = \{P_1, P_2, \ldots, P_m\}$
  - a set of *s* types of **resources**, $R = \{R_1, R_2, \ldots, R_s\}$

find an assignment of P and R to Γ which produces a feasible schedule.



Scheduling Complexity

- In 1975, Garey and Johnson showed that the general scheduling problem is NP hard.
  - There is no known polinomial time algorithm
  - Meaning:
    - Consider n = 30 tasks; elementary step = 1µs
    - Alg. 1: O(n)
    - Alg. 2: $O(n^6)$
    - Alg. 3: $O(6^n)$
    - Computation time?
- However, polynomial time algorithms can be found **under particular conditions**

Simplifying Assumptions

- Simplify architecture
  - Single processor
- Homogeneous task sets
  - Only periodic / only aperiodic
- Fully preemptive tasks
- Simultaneous activations
- No precedence constraints
- No resource constraints
- …

- Different classes of algorithms

Algorithm Tassonomy

- Preemptive vs. Non Preemptive
- Static vs. Dynamic
- On-line vs. Off-line
- Optimal vs. Heuristic
- Guaranteed vs. Best-effort

- **Clairvoyant** algorithm

Static vs Dynamic

- **Static scheduling algorithms**
  - scheduling decisions are taken based on fixed parameters, statically assigned to tasks before activation.

- **Dynamic scheduling algorithms**
  - scheduling decisions are taken based on parameters that can change with time.

Off-line vs. On-line

- **Off-line scheduling algorithms**
  - all scheduling decisions are taken before task activation: the schedule is stored in a table and later executed by a dispatcher
    - **"table-driven scheduling"**
- **On-line scheduling algorithms**
  - scheduling decisions are taken at run-time on the set of active tasks
    - When?

Optimal vs. Heuristic

- **Optimal scheduling algorithms**
  - They generate a schedule that minimizes a cost function, defined based on an optimality criterion.

- **Heuristic scheduling algorithms**
  - They generate a schedule according to a heuristic function that tries to satisfy an optimality criterion, but there is no guarantee of success.

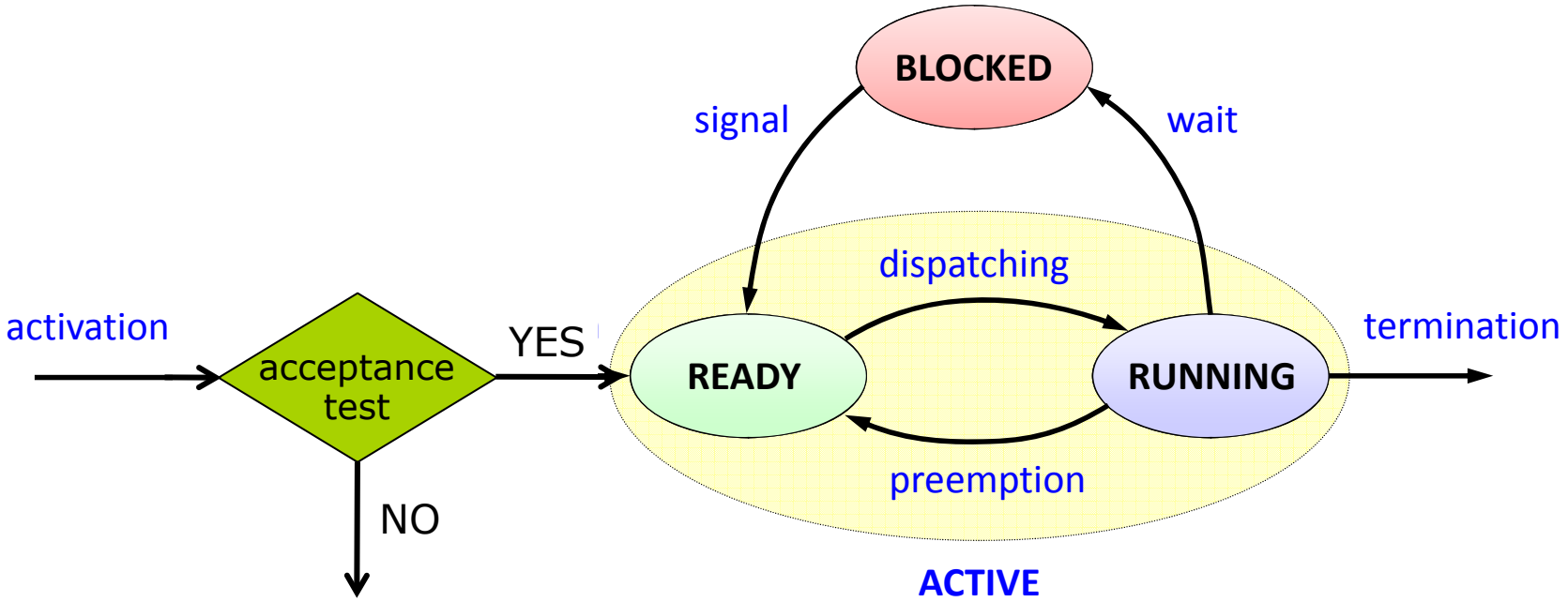
Guaranteed vs. Best Effort

- **Guaranteed scheduling algorithms**
  - They generate a feasible schedule if there exists one
  - Needed is hard real-time
  - Pessimistic assumptions
- **Best effort scheduling algorithms**
  - No guarantee of a feasible schedule.
  - Useful if soft real-time
  - Optimize average performance

Guarantee-Based Algorithms

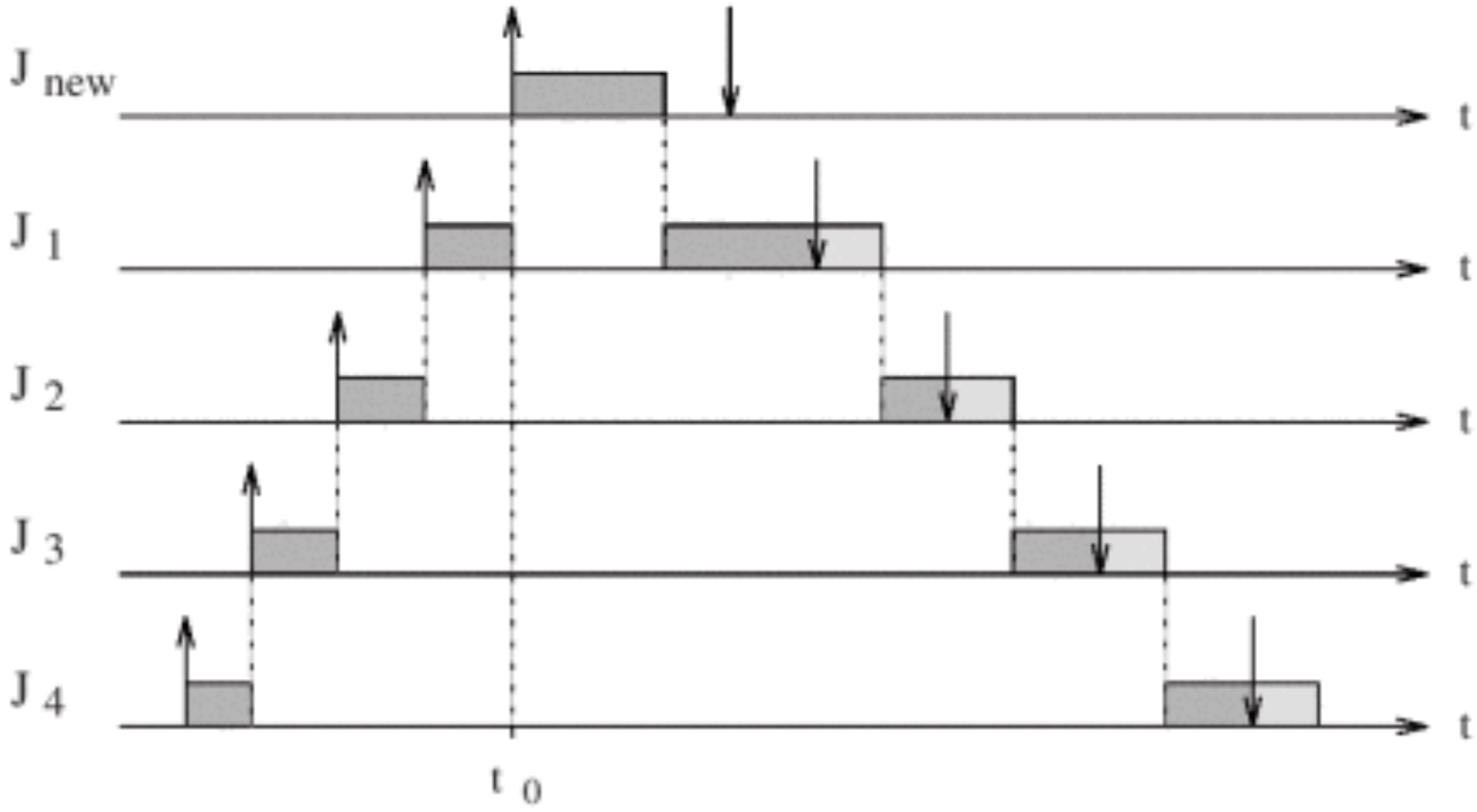
- In hard real-time applications, the feasibility of the schedule must be guaranteed before task execution
  - Give the system time to try and avoid catastrophic consequences
  - Look-ahead and worst-case reasoning
- **Static** real-time systems: guarantee off-line; table-based scheduling
  - + Run-time overhead does not depend on complexity of scheduling algorihtm
  - - Flexibility
- **Dynamic** real-time systems: task can be created at run-time
  - Guarantee online every time a new task is created

Guarantee Mechanism



- Worst-case assumption: a task could unnecessarily be rejected
  - - Efficiency
- Early detection of potential overload situation
  - + Avoid negative effects (possible catastrophe, domino effect)

Domino Effect



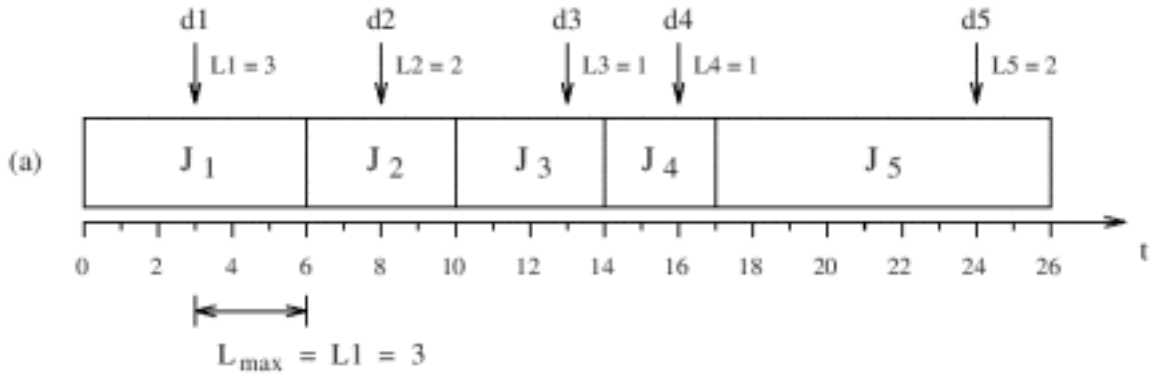
Metrics for Performance Evaluation

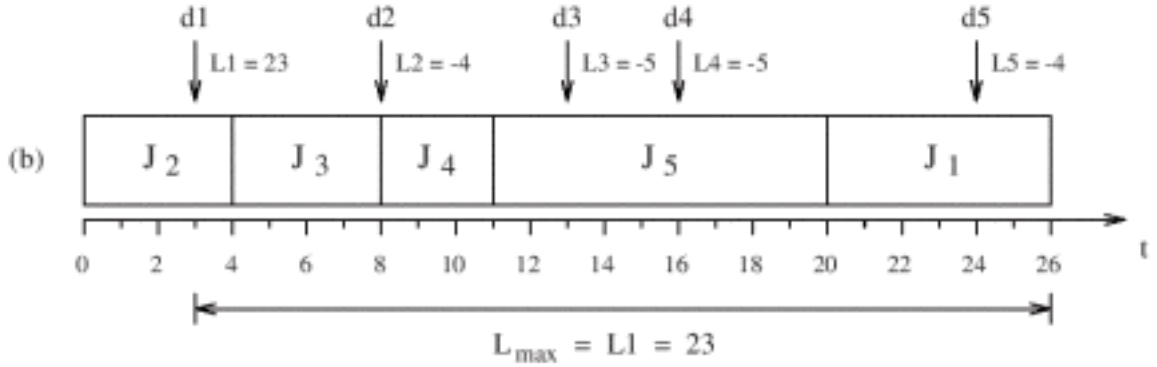
- Classical operating systems
- Optimality: min cost function
  - Average response time
  - Total completion time
  - Weighted sum of completion times
  - Maximum lateness
  - Maximum number of late tasks
  - …
- Real-Time Operating Systems: these cost functions are not necessarily of interest
  - No individual assessment of timing properties (periods, deadlines)
  - Maximum lateness has no direct relation with number of tasks that miss their deadline

Maximum Lateness

a) Min maximum lateness

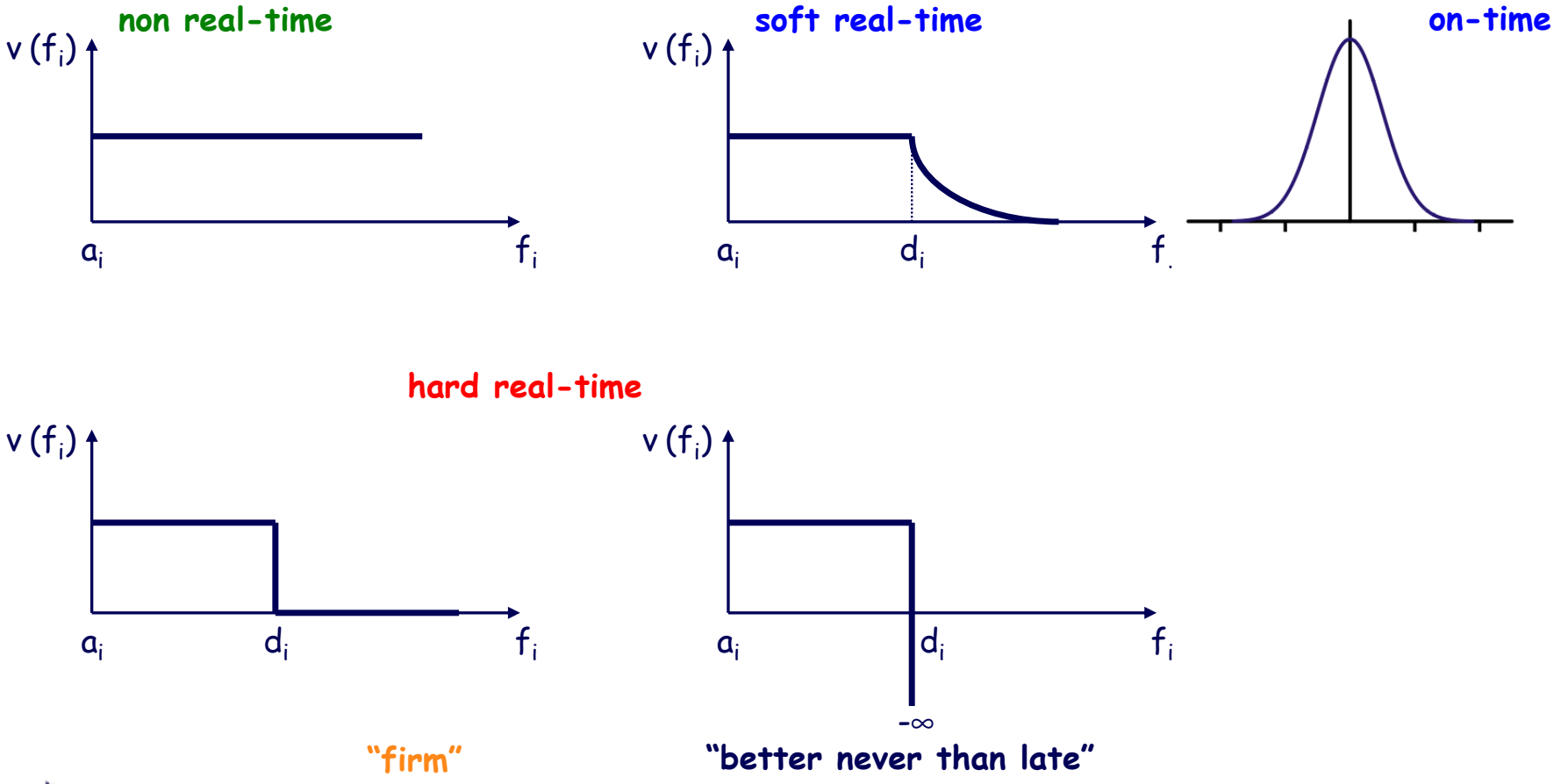
b) Min number of tasks that miss their deadline





Cumulative Value

- Sum of the utility functions computed at each completion time



Scheduling Anomalies

Retis
Real-Time Systems Laboratory

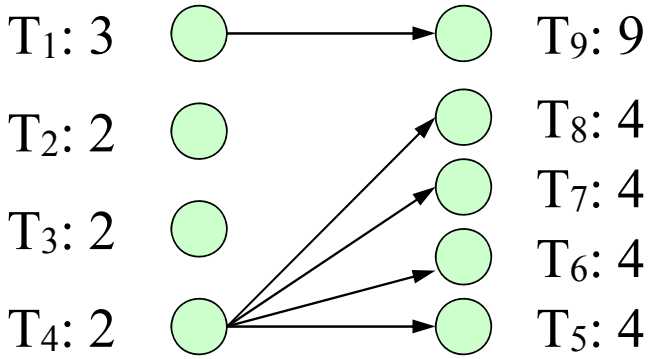
A Surprising Result…

- **Theorem** (Graham, 1976)

If a task set is optimally scheduled on a multiprocessor with some priority assignment, a fixed number of processors, fixed execution times, and precedence constraints, then increasing the number of processors, reducing execution times, or weakening the precedence constraints can increase the schedule length

- **Brittleness of scheduling algorithms**: small changes can have big, unexpected consequences

Richard's Anomalies (Graham, 1976)



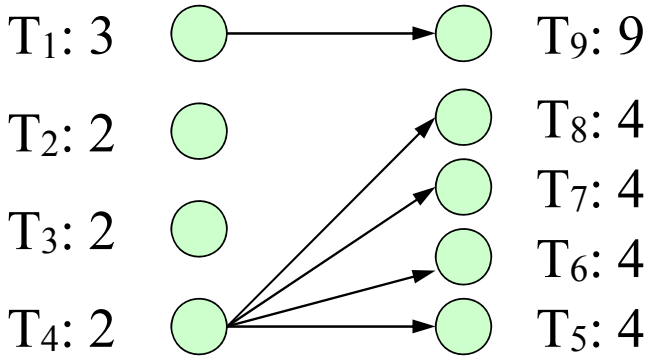
priority

$P_i > P_j \quad \forall\, i < j$

- Assume a parallel machine with 3 CPU. All tasks arrive at the same time.

1. What priority-based schedule?
2. How to improve schedule length?

1) Increase number of CPUs

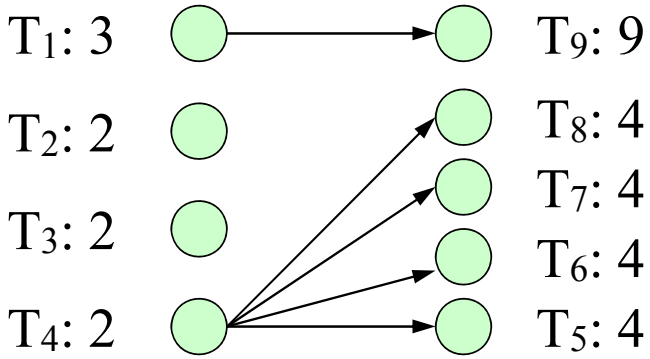


priority

$P_i > P_j \quad \forall\, i < j$

- Assume 4 CPUs

2) Reduce computation time

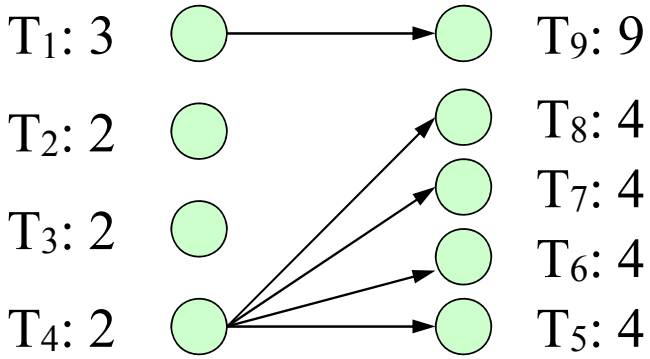


priority

$P_i > P_j \quad \forall\, i < j$

- Reduce computation time of each task by 1 unit

3) Weaken precedence constraints



priority

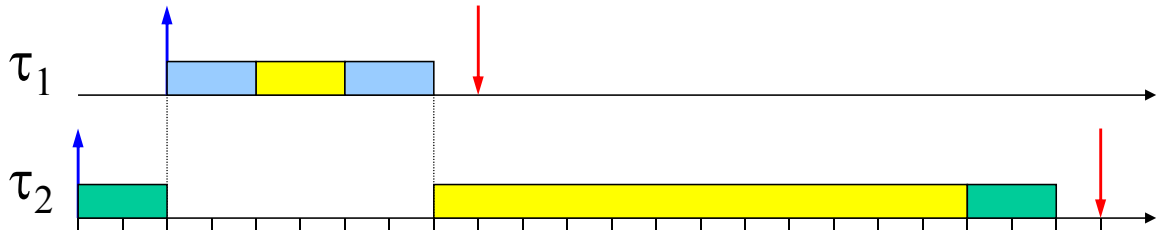
$P_i > P_j \quad \forall\, i < j$

- Remove constraints on T_7 and T_8

Another Surprising Result…

- If tasks share **mutually exclusive** resources, or are non-preemptive, scheduling anomalies may also occur in uniprocessor systems
- **Theorem** (Buttazzo, 2006)

*A real time application that is feasible on a given processor can become infeasible when running on a **faster processor***

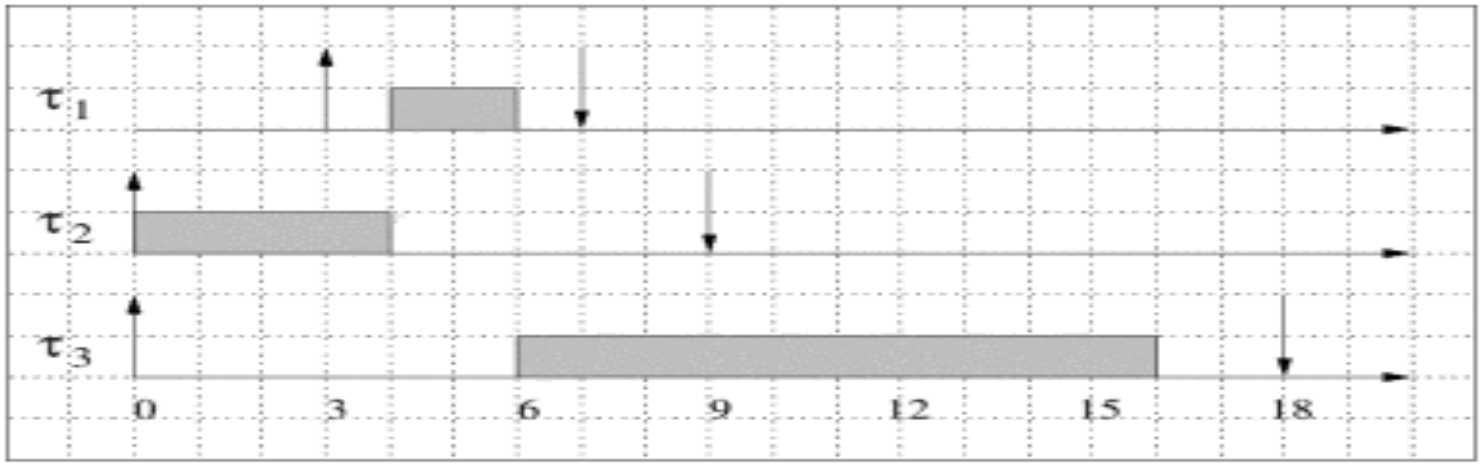


What if double the processor's speed?

Another Surprising Result…

- If tasks share mutually exclusive resources, or are **non-preemptible**, scheduling anomalies may also occur in uniprocessor systems
- **Theorem** (Buttazzo, 2006)

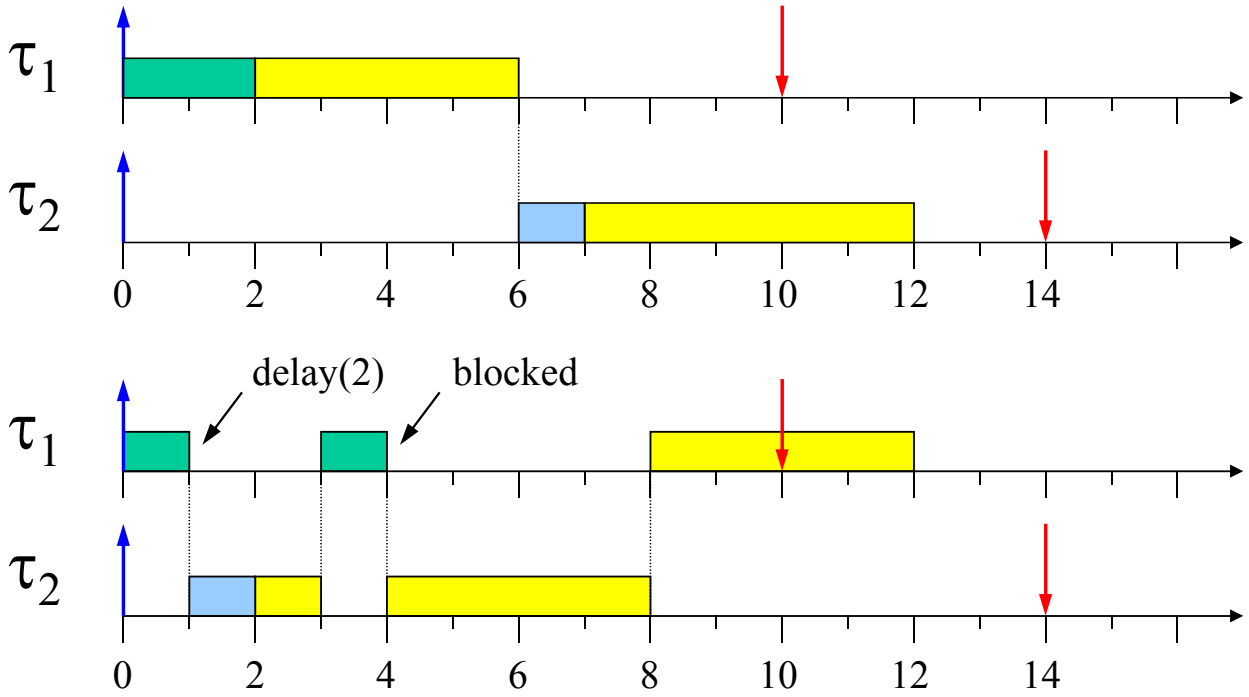
*A real time application that is feasible on a given processor can become infeasible when running on a **faster processor***



What if double the processor's speed?

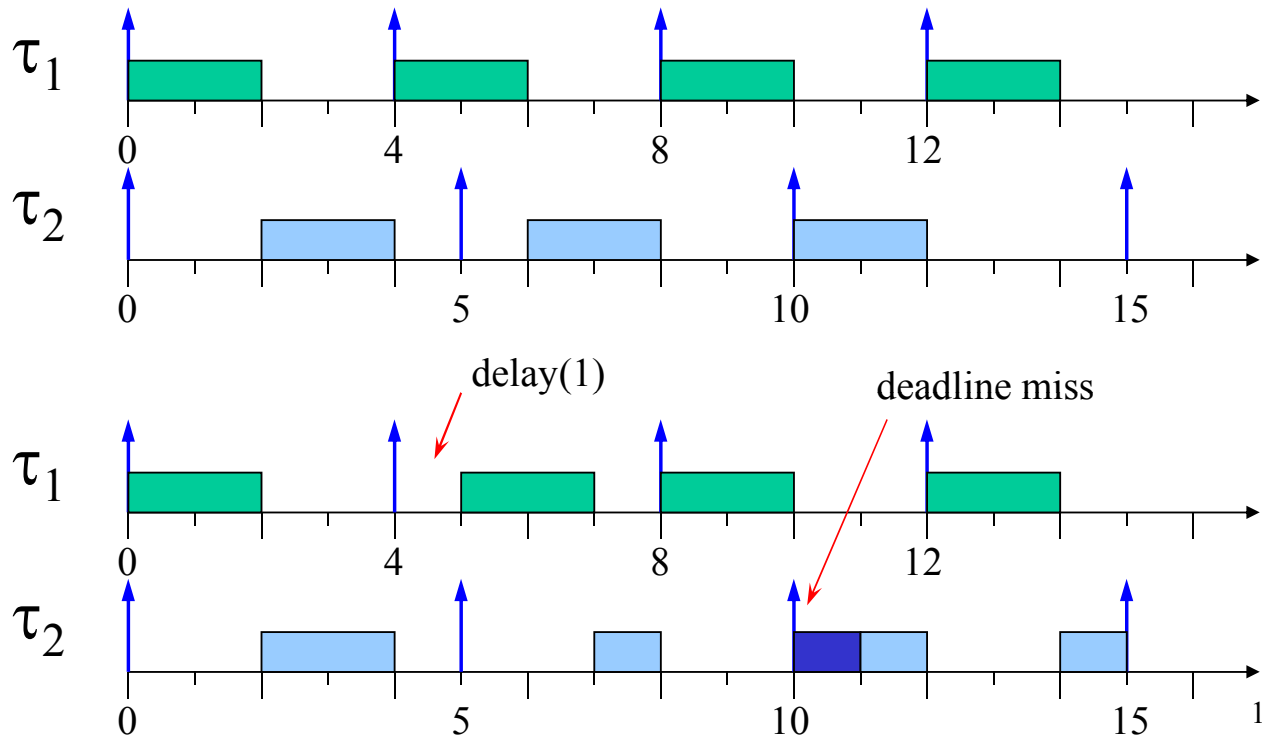
A dangerous operation: DELAY

- A delay(Δ) may introduce a delay greater than Δ



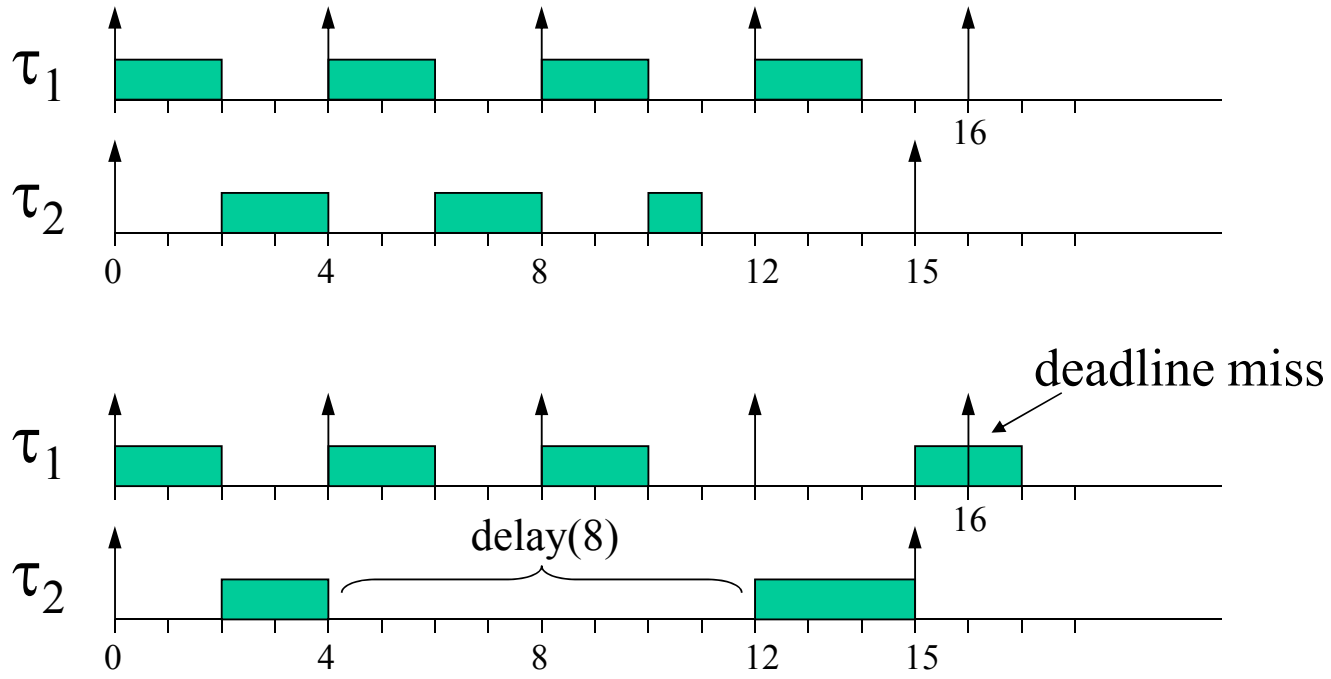
A dangerous operation: DELAY

- A delay(Δ) may also increase the response times of other tasks
  - Example for fixed priorities



A dangerous operation: DELAY

- A delay(Δ) may also increase the response times of other tasks
  - Example for deadline scheduling



Take-Home Message

- **Tests** are not enough for real-time systems
- **Intuitive solutions** do not always work
- **Delay** should not be used in real-time tasks
- The safest approach:
  - Use **predictable kernel** mechanisms
  - **Analyze** the system to predict the behaviour

- The operating system is the part most responsible for a predictable behavior. Concurrency control must be enforced by:
  - appropriate **scheduling** algorithms
  - appropriate **syncronization** protocols
  - efficient **communication** mechanisms
  - predictable **interrupt** handling