

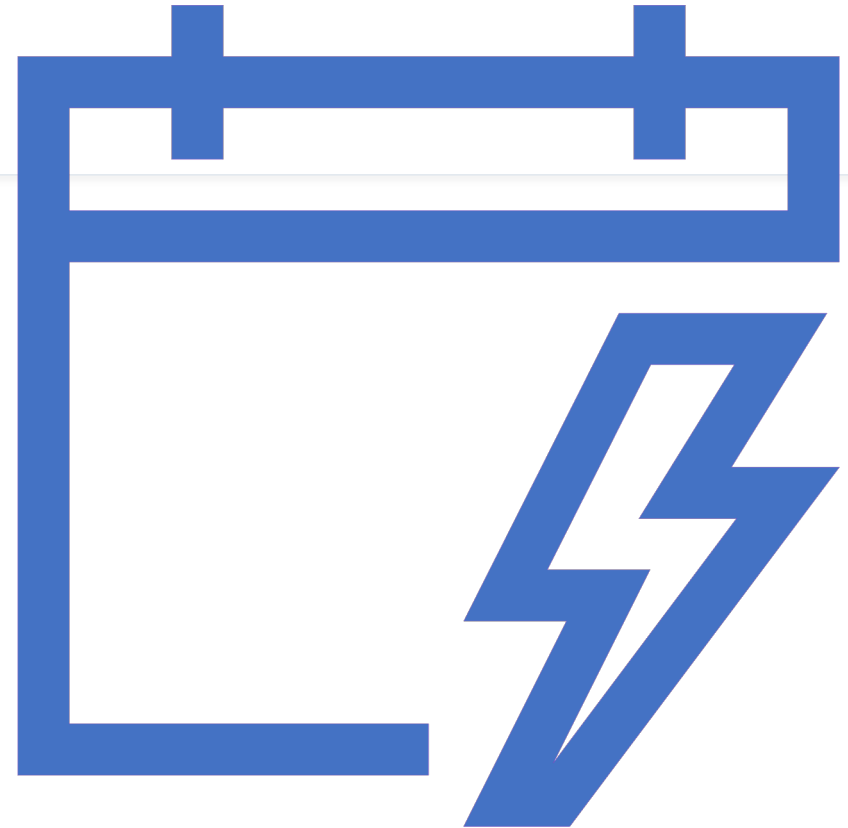


# **Embedded Operating System RTOS scheduling - introduction**

Graziano Pravadelli



# Introduction



# Terminology

## Task

- A processes in the RT community

## Active task

- A task that can potentially execute on the CPU

## Ready task

- A task waiting for the CPU in the ready queue

## Running task

- A task in execution

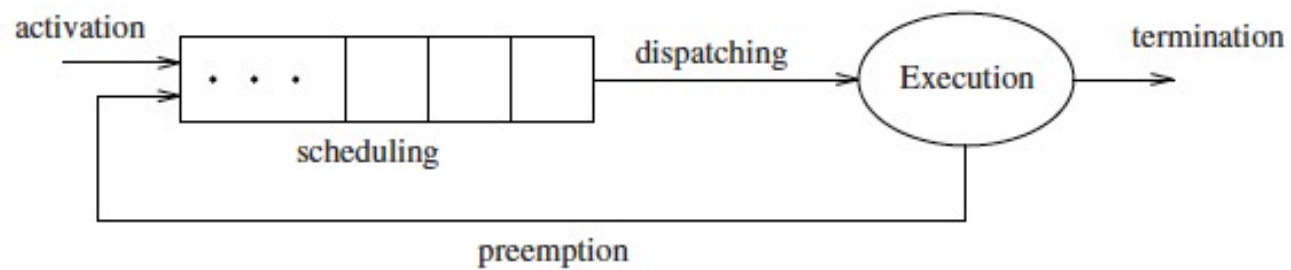
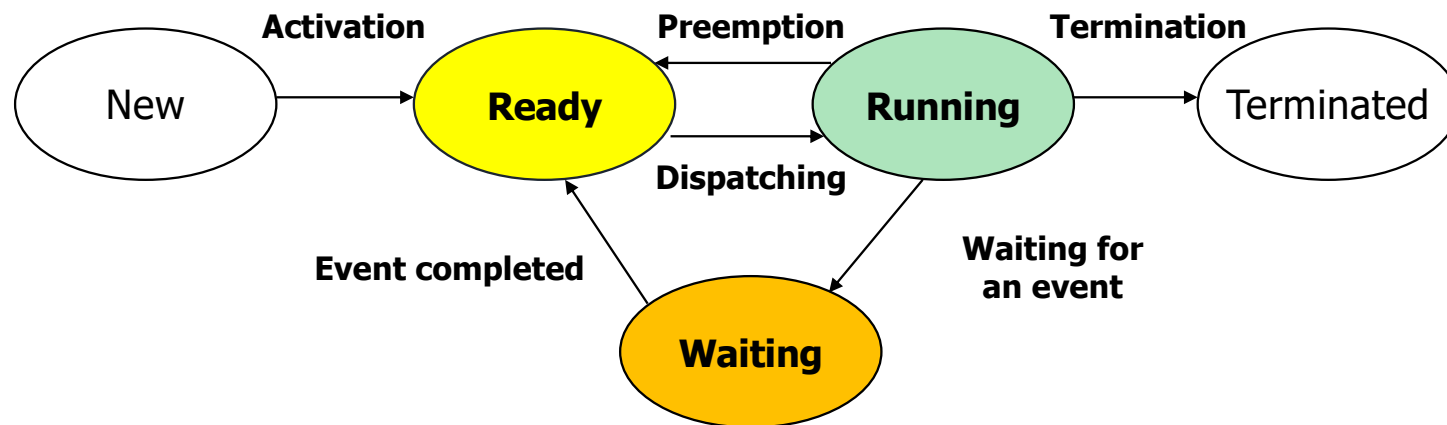
## Scheduling algorithm

- Rules to decide the task execution order

## Dispatcing

- Allocation of the CPU to the task selected by the scheduler


# Task states



# Key facts

---

Any RT scheduling  
policy must be  
preemptive  
because



Tasks performing exception handling may need to preempt running tasks to ensure timely reaction  
Tasks may have different levels of criticalness  
More efficient schedules can be produced with preemption

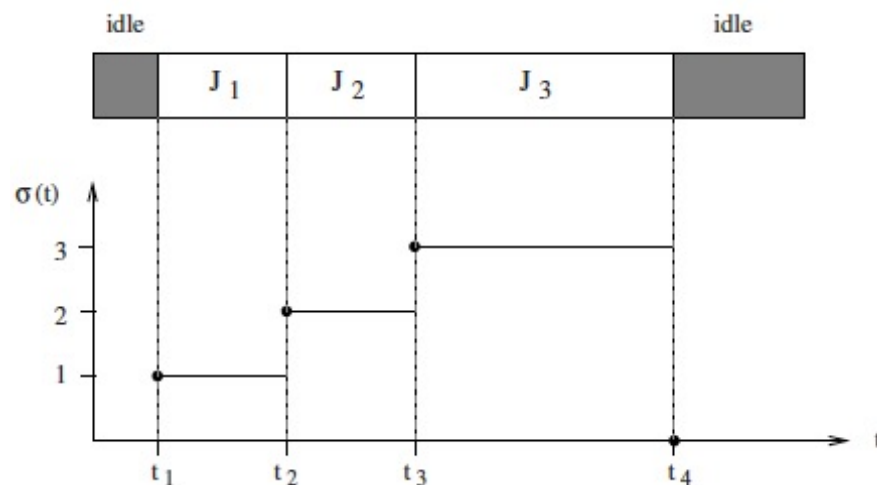
But preemption

Destroys program locality  
Introduces overhead  
Must be used accurately

# Definition of schedule

- Given a set of tasks  $J = \{J_1, \dots, J_n\}$  a schedule is an assignment of tasks to the processor so that each task is executed until completion
- Formally, **a schedule is a function**  $s : \mathbb{R}^+ \rightarrow \mathbb{N}$  such that
  - $\forall t \in \mathbb{R}^+, \exists t_1, t_2 \in \mathbb{R}^+ \mid \forall t' \in [t_1, t_2) \ s(t) = s(t')$

# Scheduling – Example



- In practice,  $s$  is an integer step function
  - $s(t) = k$  means task  $J_k$  is executing at time  $t$
  - $s(t) = 0$  means CPU is idle
- Each interval  $[t_i, t_{i+1})$  with  $s(t)$  constant for  $t \in [t_i, t_{i+1})$  is called a time slice

# Scheduling – Properties

---

A schedule is called **feasible** if all tasks can be completed according to a set of specified constraints

A set of tasks is called **schedulable** if there exist at least one algorithm that can produce a feasible schedule



# Scheduling constraints

Timing  
constraints

Meet your  
deadline

Precedence  
constraints

Respect  
prerequisites

Resource  
constraints

Access only  
available  
resources

# Timing constraints

---

Real-time systems are characterized mostly by timing constraints

- Typical timing constraint: deadline

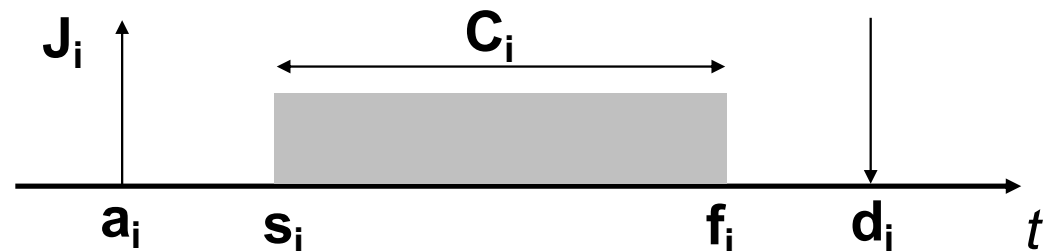
Deadline missing separates three classes of RT systems

- Hard
  - missing of deadline can cause catastrophic consequences
- Soft
  - missing of deadline decreases performance of system
- Firm
  - Missing of deadline does not cause damage, but the result is useless

# Task characterization (1)

- **Arrival time  $a_i$** 
  - the time  $J_i$  becomes ready for execution
  - Also called release/request time  $r_i$
- **Computation time  $C_i$** 
  - time necessary for execution without interruption
- **Deadline  $d_i$** 
  - time before which task has to complete its execution

- **Start time  $s_i$** 
  - time at which  $J_i$  start its execution
- **Finish time  $f_i$** 
  - time at which  $J_i$  finishes its execution



## Task characterization (2)

- **Response time  $R_i$**

- $R_i = f_i - a_i$

- **Lateness  $L_i$**

- $L_i = f_i - d_i$

- $L_i < 0$  if task meets deadline

- **Laxity or slack time  $X_i$**

- $X_i = d_i - a_i - C_i$

- Maximum time a task can be delayed to complete withing  $d_i$





## Task characterization (3)

---

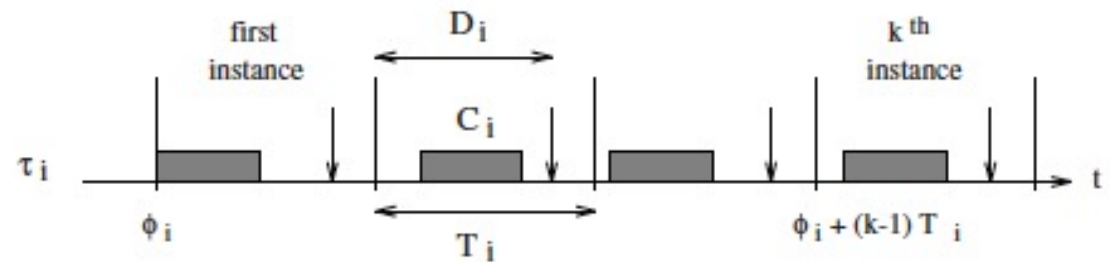
### Time-driven activation

- Periodic tasks

### Event-driven activation

- Aperiodic tasks
- Sporadic tasks

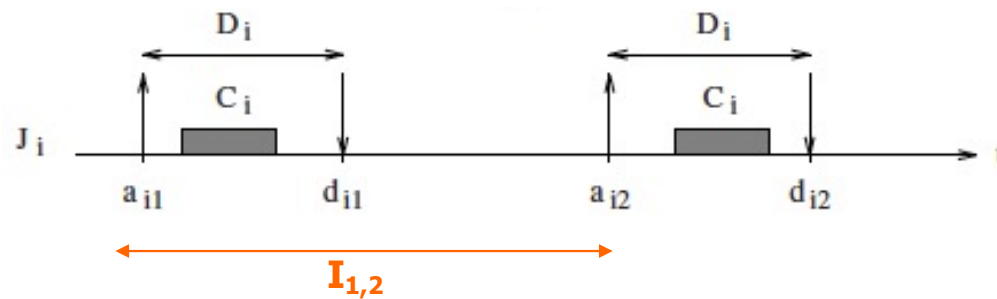
# Periodic tasks



- A periodic task  $\tau_i$  is an infinite sequence of identical activities, called *instances*
  - Regularly activated at a constant rate
  - Activation time of first instance of  $\tau$  is called phase ( $\phi_i$ )
  - Characterized by  $C_i$ ,  $T_i$ ,  $D_i$ 
    - $T_i$  = period of the task
    - $C_i$ ,  $T_i$ ,  $D_i$  constant for each instance
    - In most cases  $T_i = D_i$

# Aperiodic tasks

- An aperiodic task  $J_i$  is an infinite sequence of identical activities, called *instances*
  - Activations are not regular
  - Usually, there is a small number of instances
- Sporadic similar to aperiodic, but inter-arrival time is bounded



Aperiodic:  $I_{1,2}$  unknown  
Sporadic:  $I_{1,2} > IT_{\min}$

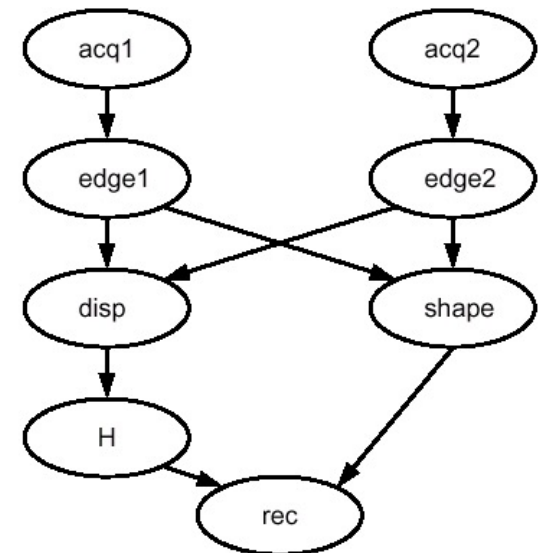
# Precedence constraints

- Tasks have often to respect some precedence relations
  - Described by a DAG
    - Nodes = tasks
    - Edges = precedence relations
  - The DAG induces a partial order on the task set
- Notation
  - $J_a < J_b$  means  $J_a$  is a predecessor of  $J_b$ 
    - There exists a path from task (node)  $J_a$  to task  $J_b$  in the DAG
  - $J_a \rightarrow J_b$  means  $J_a$  is an immediate predecessor of  $J_b$ 
    - There exist an edge  $(J_a, J_b)$  in the DAG



# Precedence constraints – Example

- System for recognizing objects on a conveyor belt through two cameras
- Tasks
  - For each camera
    - image acquisition (acq1 and acq2)
    - low level image processing (edge1 and edge2)
  - Extraction of two-dimensional features from object contours (shape)
  - Computation of pixel disparities from the two images (disp)
  - Computing of object height (H)
  - Final recognition of the object (rec)



# Resource constraints

---

## Resources

- Any SW structure that can be used by process to advance execution
  - Data structure, set of variables, memory area, files, registers of a peripheral, ...
- Distinction between private resources, shared resources and exclusive resources

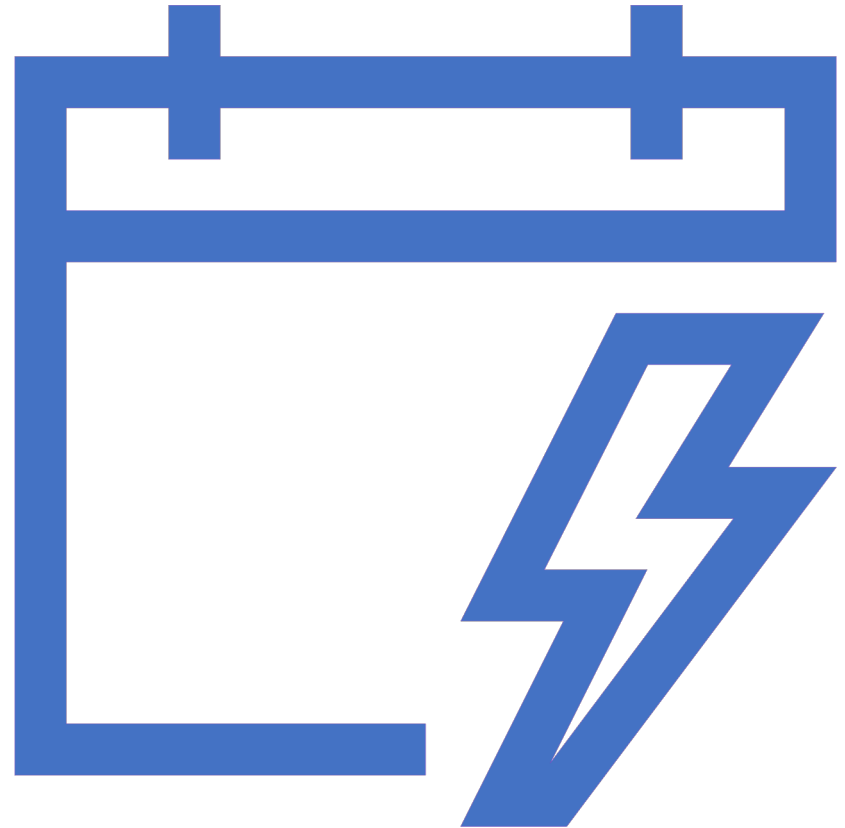
## Critical section as for conventional systems

- Conventional semaphore-like structure suffer from priority inversion problem



# The scheduling problem

---



# Problem formulation

- Given
  - a set of tasks  $J = \{J_1, \dots, J_n\}$
  - a set of processor  $P = \{P_1, \dots, P_m\}$
  - a set of resources  $R = \{R_1, \dots, R_s\}$
  - precedencies specified by using a precedence graph
  - timing constraints associated to each task
- Scheduling means to assign processors from  $P$  and resources from  $R$  to tasks from  $J$  in order to complete all tasks under the imposed constraints



NP  
COMPLETE

# Scheduling – Classification (1)

---

## Preemptive vs. Non-preemptive

- Preemptive: task can be interrupted
- Non-preemptive: tasks leaves CPU when completed

## Static vs. Dynamic

- Static: decisions based on fixed parameters assigned before activation
- Dynamic: decisions based on parameters that change during system evolution

# Scheduling – Classification (2)

## Off-line vs. On-line

- Off-line: preformed on the entire task set before start of system
- On-line: decisions taken at run-time when a task enters or leaves the system

## Optimal vs. Heuristic

- Optimal: if it minimizes a cost function
- Heuristic: if it tends to the optimal schedule

# Scheduling – Guarantee-based algorithms

Hard RT systems require feasibility of schedule guaranteed in advance



Solutions

Static RT systems

Dynamic RT systems

# Static RT systems

---

## Pros

- All task activations can be pre-calculated off-line
- Sophisticated algorithms can be used to find optimal scheduling
- Entire schedule can be stored in a table
- Overhead for dispatching does not depend on the scheduling algorithm

## Cons

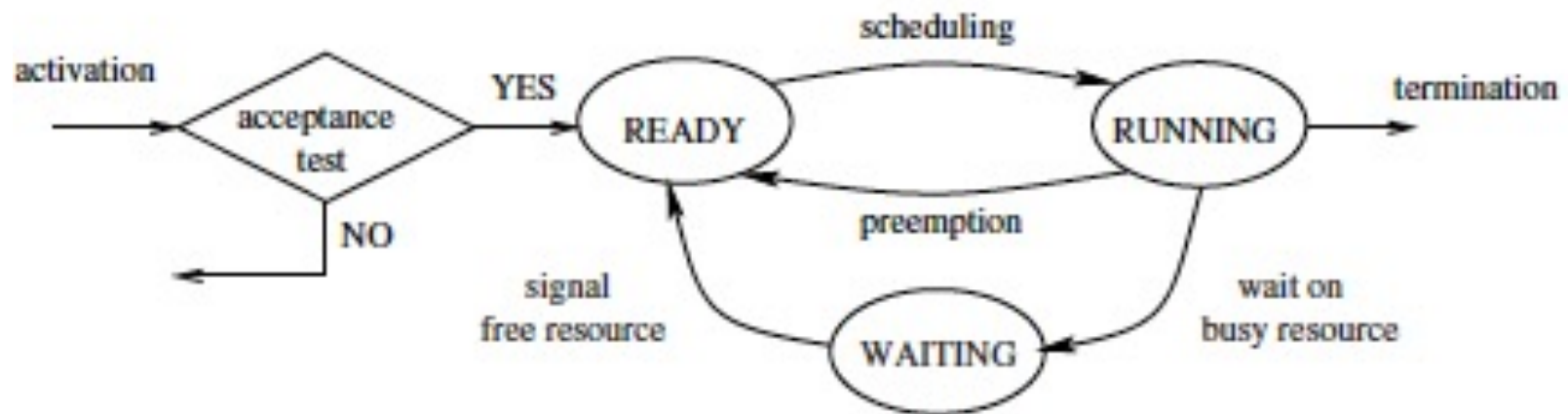
- Not flexible
- Predictability depends on the observance of hypotheses made



# Dynamic RT systems

- Activation of new (sporadic) tasks subject to **acceptance test**
  - $J$  = current task set, previously guaranteed
  - $J_{\text{new}}$  = newly arriving task
  - $J_{\text{new}}$  is accepted iff task set  $J' = J \cup \{J_{\text{new}}\}$  is schedulable
- Guarantee mechanism based on worst case assumptions
- Pessimistic
  - task could be unnecessarily rejected, but
  - potential overload are known in advance

# Acceptance test



# Scheduling metrics

---

**Average response time:**

$$\overline{t_r} = \frac{1}{n} \sum_{i=1}^n (f_i - a_i)$$

**Total completion time:**

$$t_c = \max_i(f_i) - \min_i(a_i)$$

**Weighted sum of completion times:**

$$t_w = \sum_{i=1}^n w_i f_i$$

**Maximum lateness:**

$$L_{max} = \max_i(f_i - d_i)$$

**Maximum number of late tasks:**

$$N_{late} = \sum_{i=1}^n miss(f_i)$$

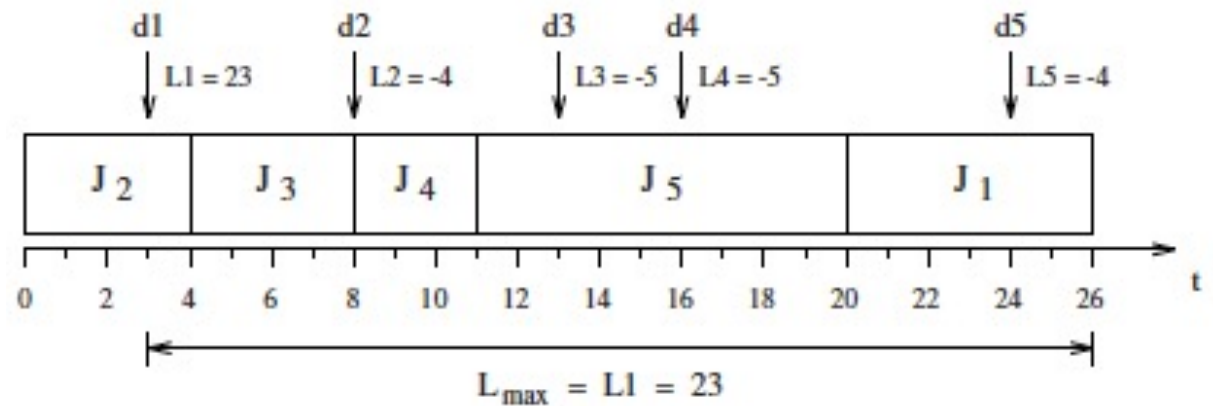
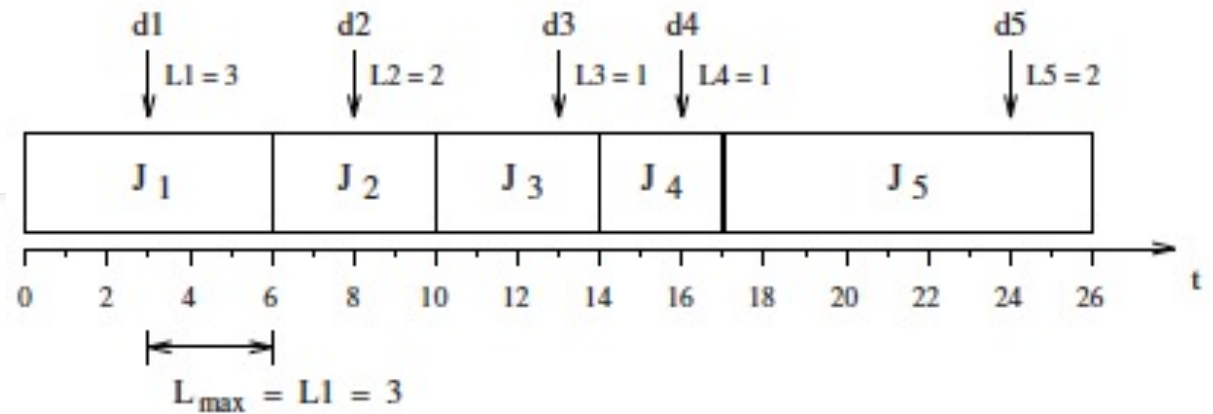
where

$$miss(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$$

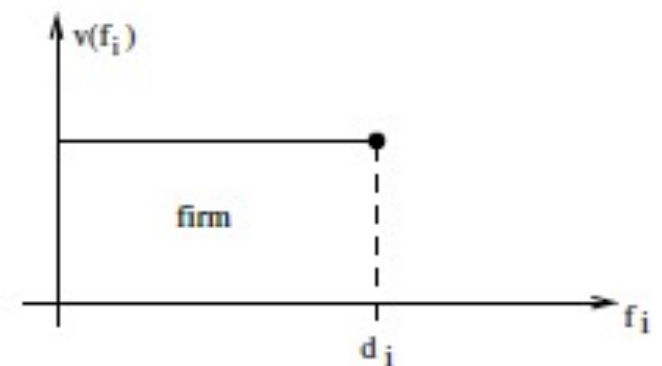
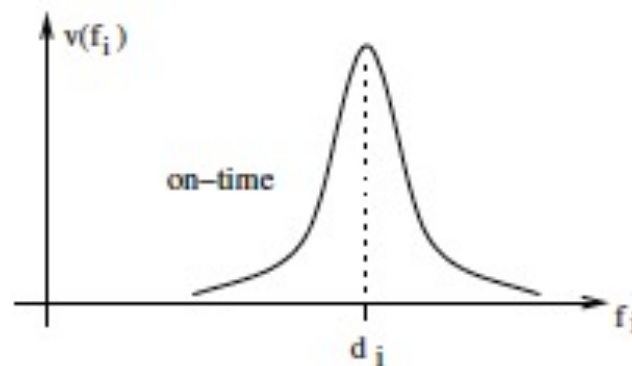
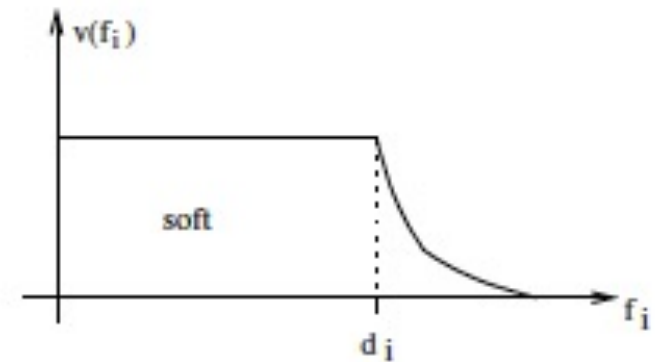
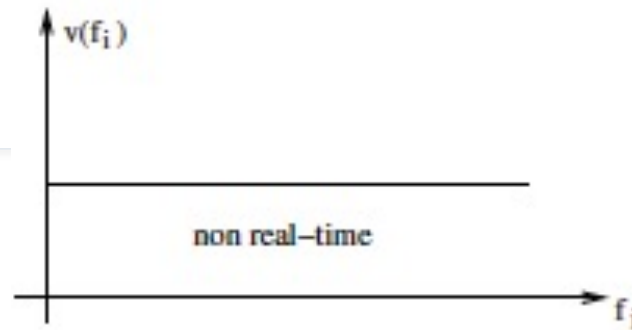
# Scheduling metrics - Considerations

- **Average response time/total completion time**
  - Not appropriate for hard real time tasks → loses information about deadline satisfaction
- **Maximum lateness**
  - Useful for “initial exploration”, but
  - Minimizing maximum lateness does not minimize number of tasks that miss deadlines
- **Max # of late task**
  - Interesting in case of soft deadlines
- In general, a mix of “**utility functions**” is used

# Scheduling metrics - Examples



# Utility functions - Examples



$$Cumulative\_value = \sum_{i=1}^n v(f_i)$$

# Scheduling anomalies (Graham)

*RT-computing is not equivalent to fast computing!*

If a task set is optimally scheduled on a multiprocessor with

- some priority assignment, a fixed number of processors, fixed times, and precedence constraints

execution

Then optimizations like

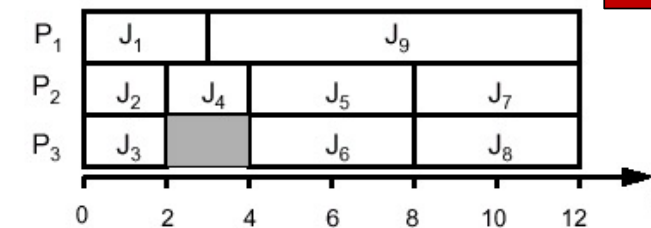
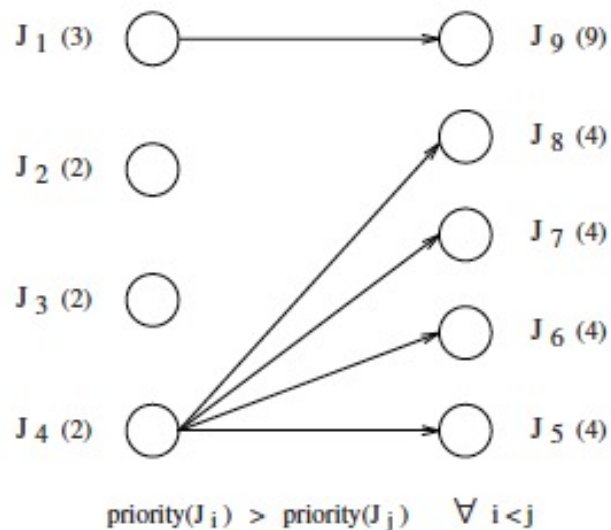
- increasing the number of processors,
- reducing execution times,
- weakening the precedence constraints

Can increase the schedule length

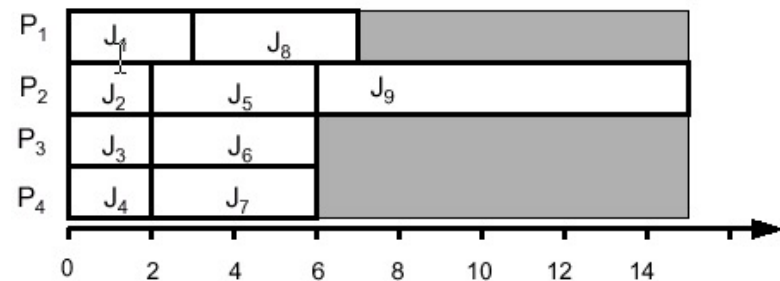
# Increasing the number of processors

Global completion time = 12

Precedence constraints



Optimal schedule of task set  $J$  on a three-processor machine



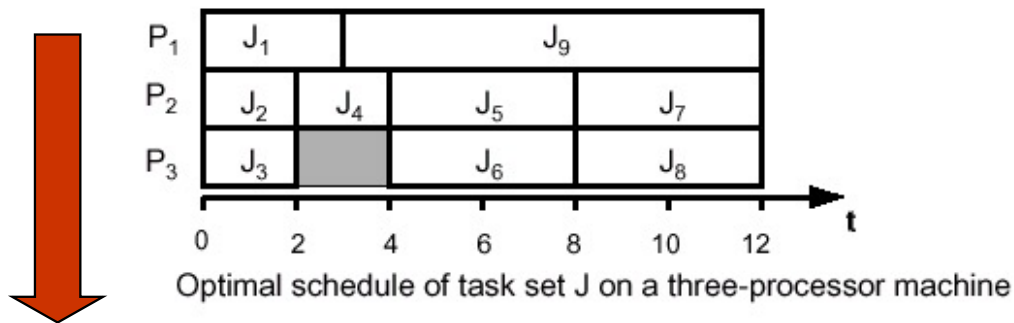
Schedule of task set  $J$  on a four-processor machine

Global completion time = 15



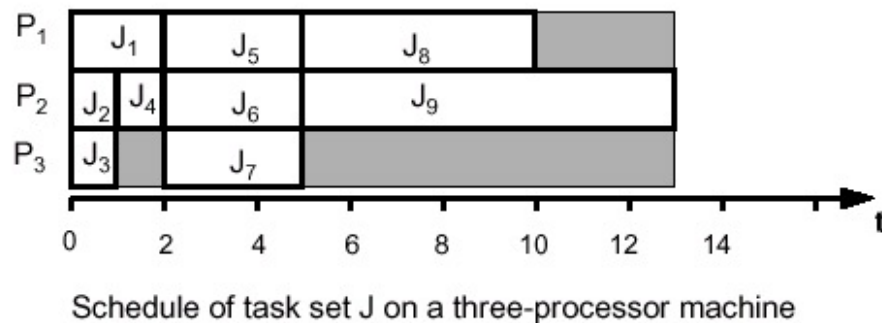
# Decreasing the computation time

Schedule with original computation times



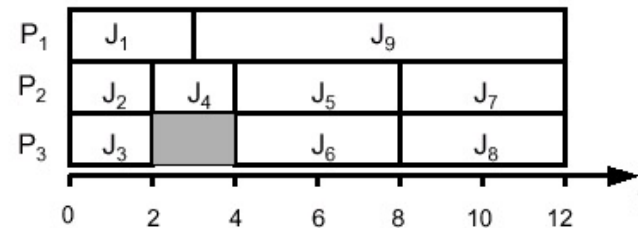
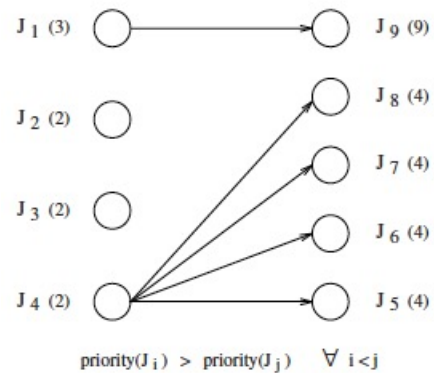
Global completion time = 12

Schedule with all computation times reduced by one

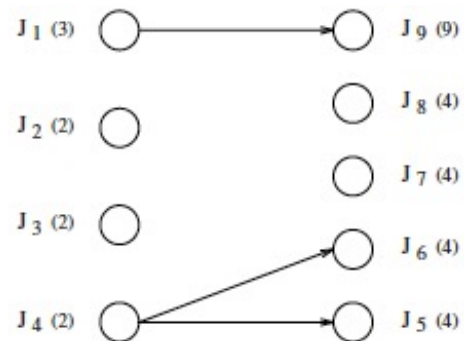
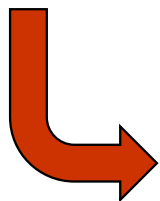


Global completion time = 13

# Weakening precedences



Global completion time = 12



Global completion time = 16