Real-Time Operating Systems and Task Scheduling

Giorgio Buttazzo

E-mail: buttazzo@sssup.it

Scuola Superiore Sant'Anna

Aim of the course

- Studing software technologies e methodologies for supporting **time critical** computing systems.
- We will not consider how to control a system, but only how to provide a proper operating system support.

Course Outline

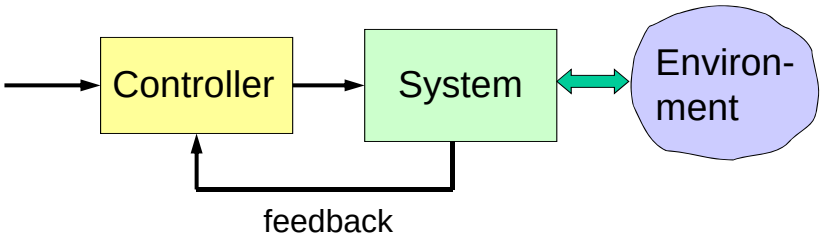
- Basic concepts
- Where timing constraints come from?
- Concurrency control
- Task scheduling
- Feasibility Analysis
- Handling shared resources

General Definitions

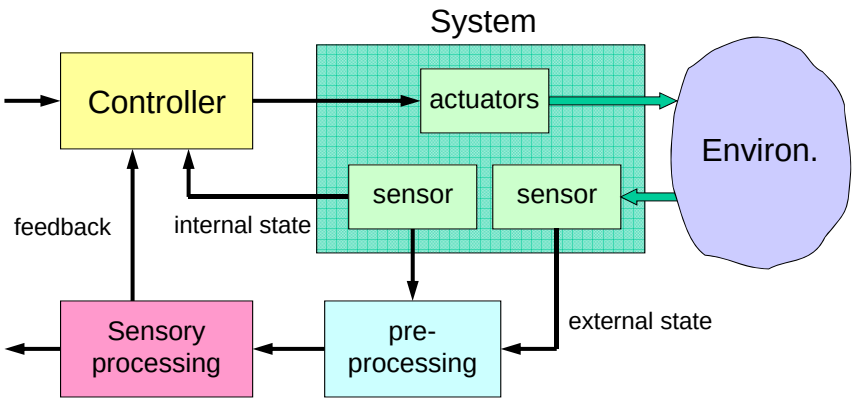
In every control application, we can distinguish 3 basic components:

- the **system** to be controlled
  - it may include sensors and actuators
- the **controller**
  - it sends signals to the system according to a predetermined control objective
- the **environment** in which the system operates

A typical control system



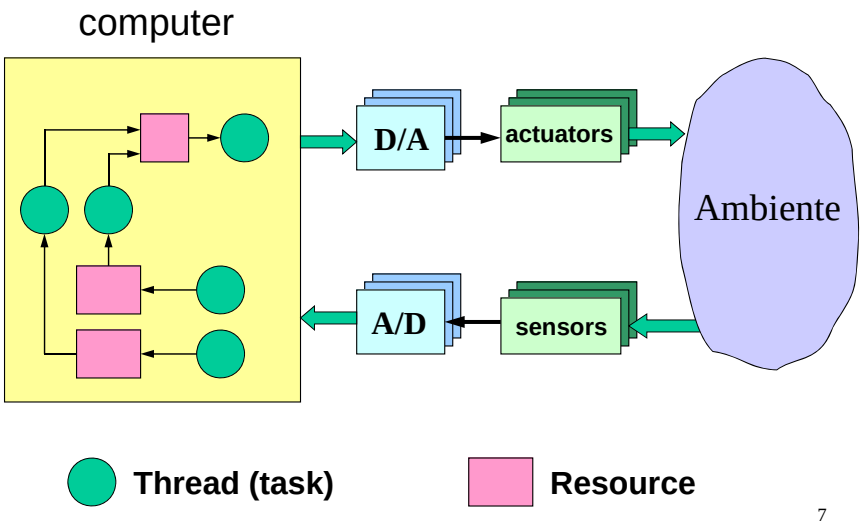
Detailed block diagram



Other activities

filtering, classification, data fusion, recognition, planning

Software Vision



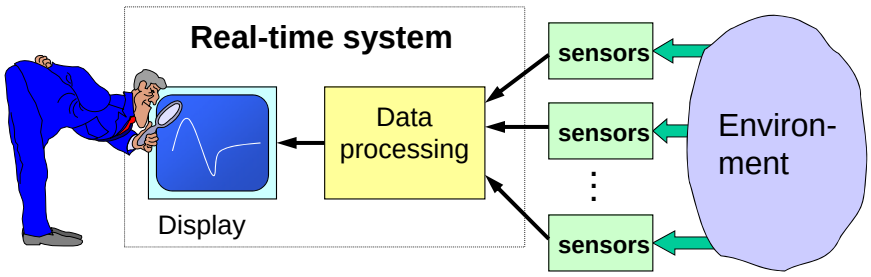
Types of control systems

Depending of the system-environment interactions, we can distinguish 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

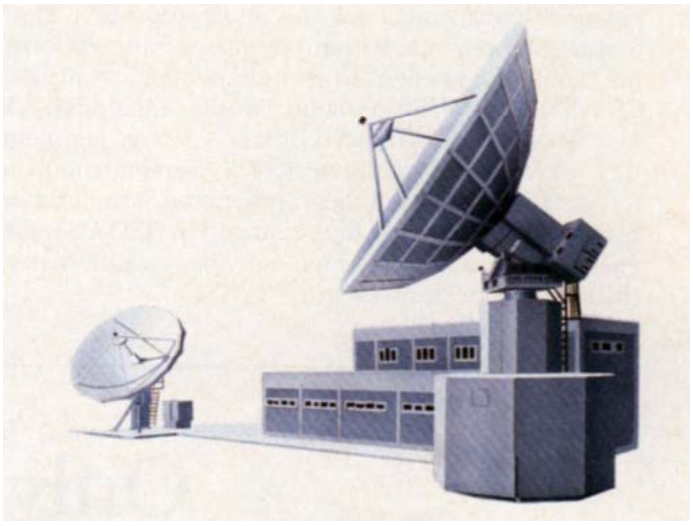
Monitoring Systems

Do not modify the environment



Examples: surveillance systems, air traffic control

Radar tracking systems

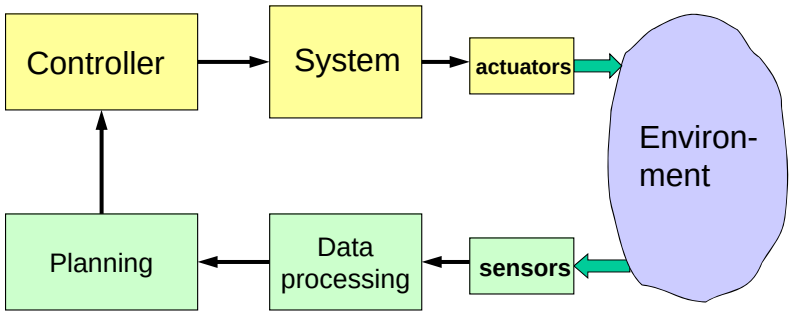


Air traffic control

Biomedical Systems

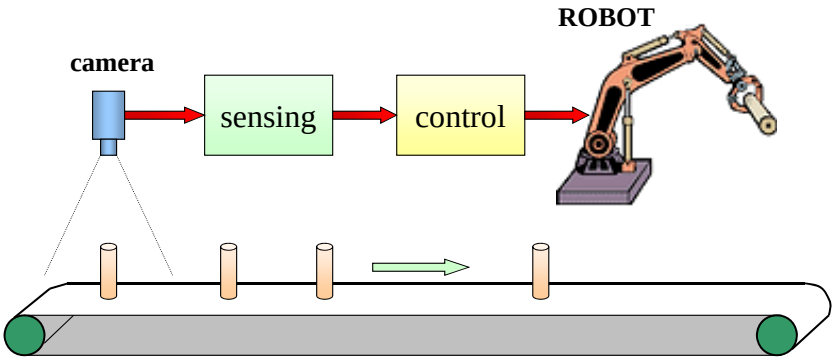
Open-loop control systems

Sensing and control are loosely coupled

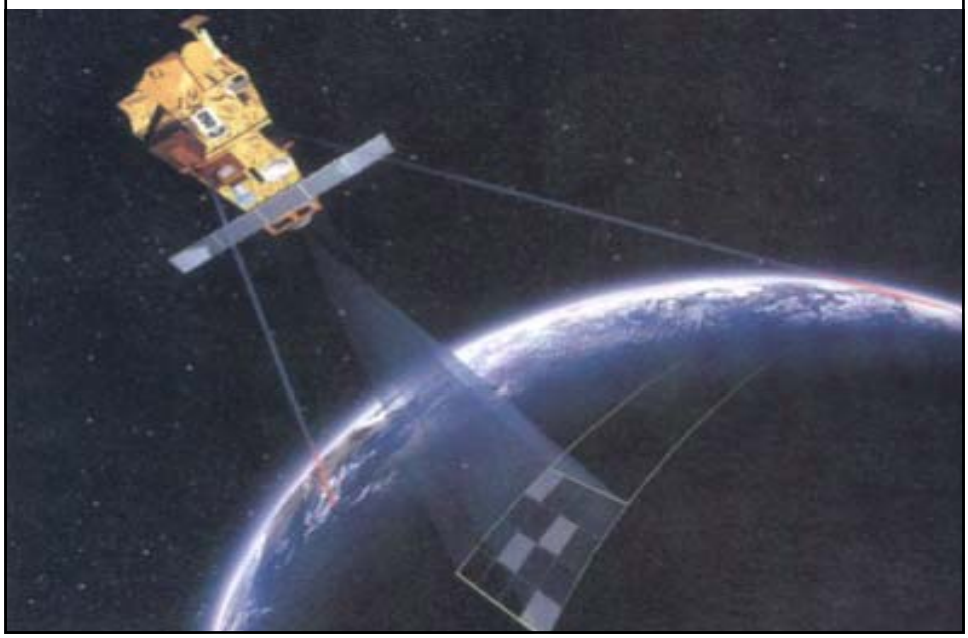


Examples: assembly robots, sorting robots

Robot Assembly

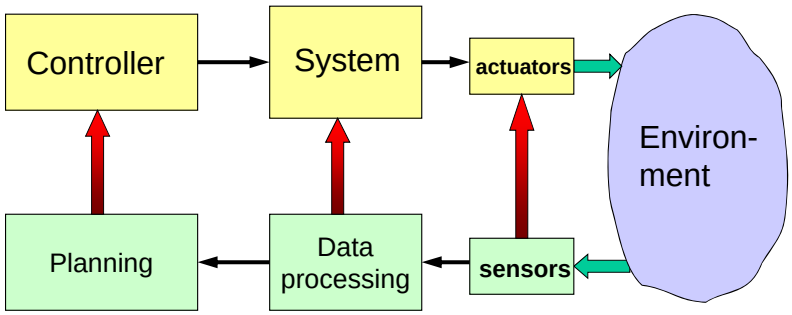


Telecommunication systems



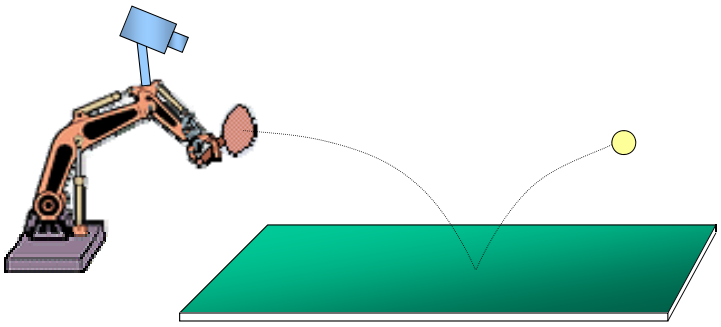
Closed-loop control systems

Sensing and control are tightly coupled



Examples: flight control systems, military systems, living beings

Ping-pong robot



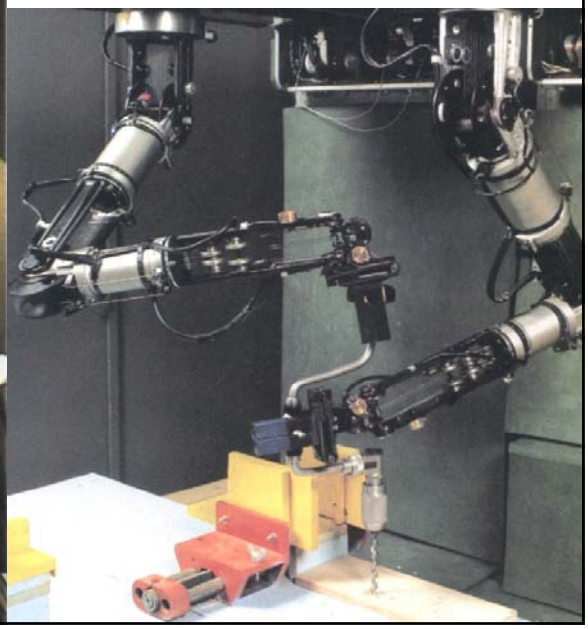
Other RT applications

Defense military systems

Flight control systems

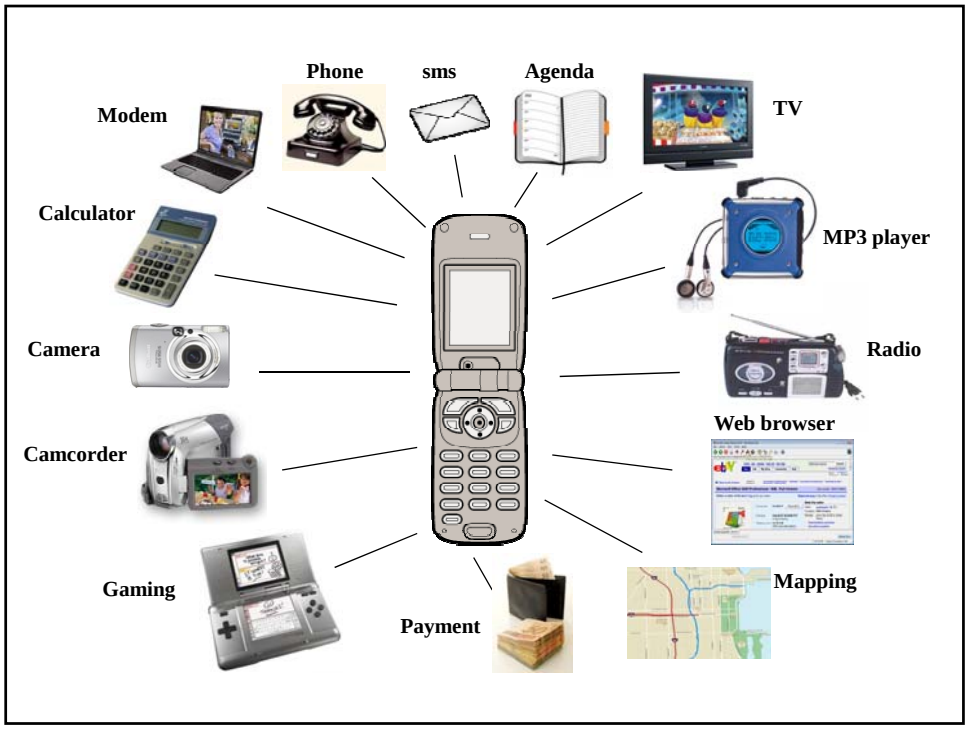
Flight simulators

Robotics

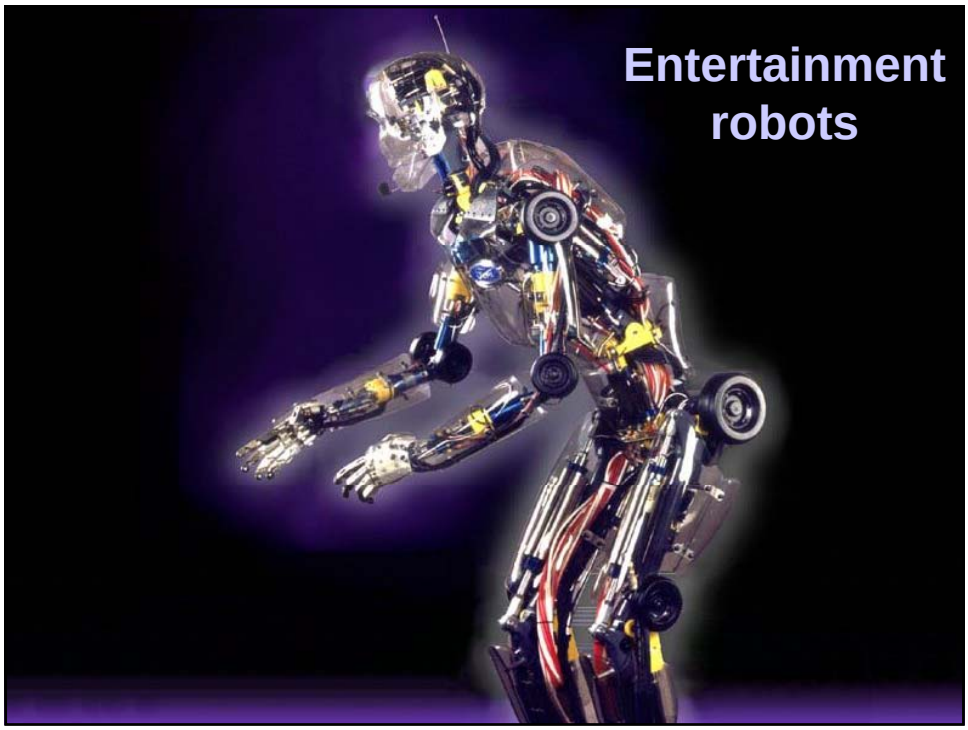


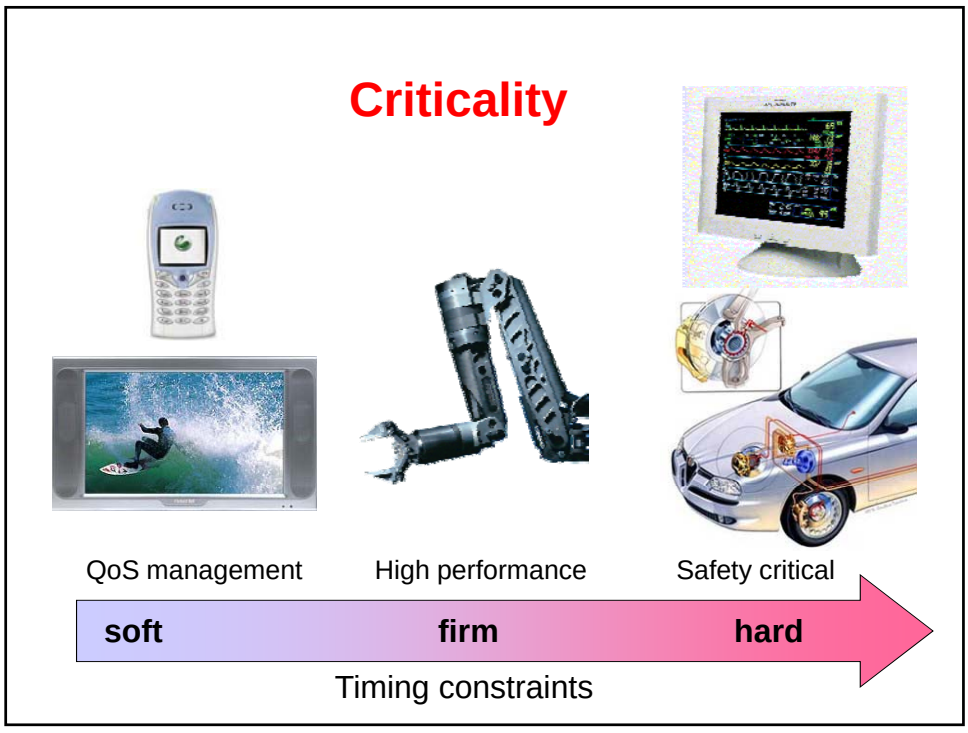
… and many others

- control of chemical/nuclear power plants
- automotive applications
- multimedia systems
- small embedded devices
  - ⇒ cell phones
  - ⇒ digital TV
  - ⇒ videogames
  - ⇒ intelligent toys

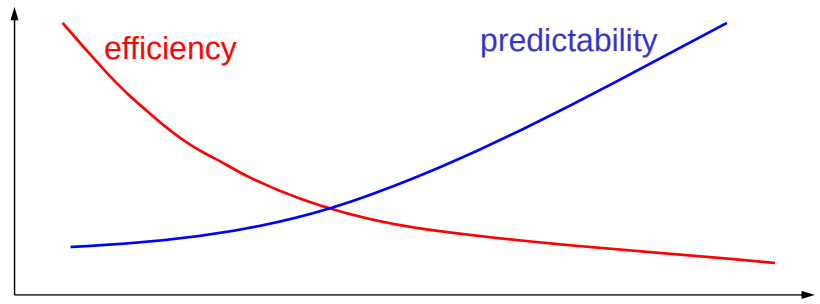

Phone
sms
Agenda
Modem
TV
Calculator
MP3 player
Camera
Radio
Web browser
Camcorder
Gaming
Mapping
Payment

AIBO - Sony


Entertainment
robots


Criticality
QoS management
High performance
Safety critical
soft
firm
hard
Timing constraints

Requirements

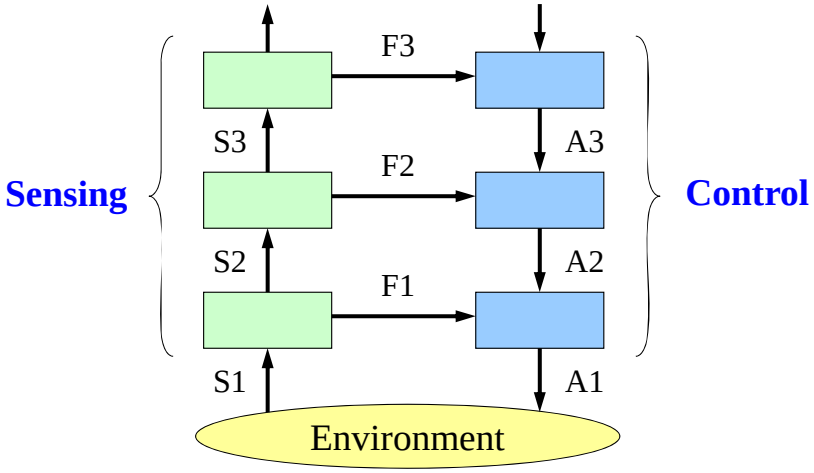


QoS management | High performance | Safety critical

soft | **firm** | **hard**

Timing constraints

Multi-level feedback control

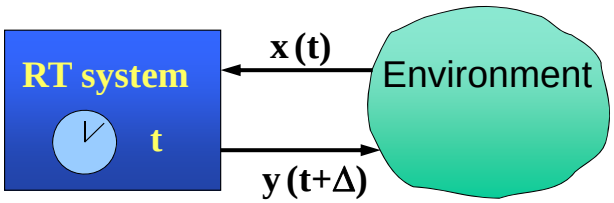


Implications

- 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.

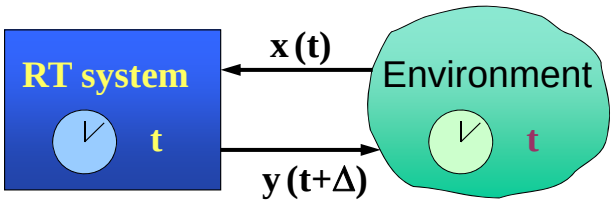
The operating system must be able to execute tasks within timing constraints.

What's a 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.

What's a real-time system?



REAL TIME means that system time must be synchronized with the time in the environment.

Typical objection to RT systems

It is not worth to invest in RT theory, because computer speed is increasing exponentially, and all timing constraints can eventually be handled.

Answer

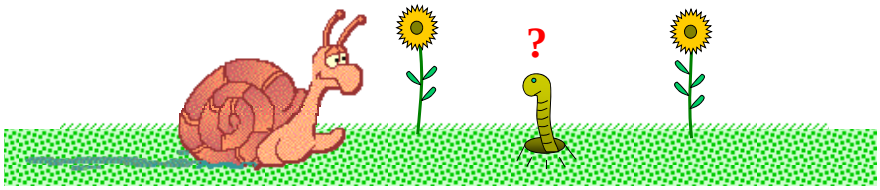
Given an arbitrary computer speed, we must always guarantee that timing constraints can be met. Testing is **NOT** sufficient.

Real-Time ≠ Fast

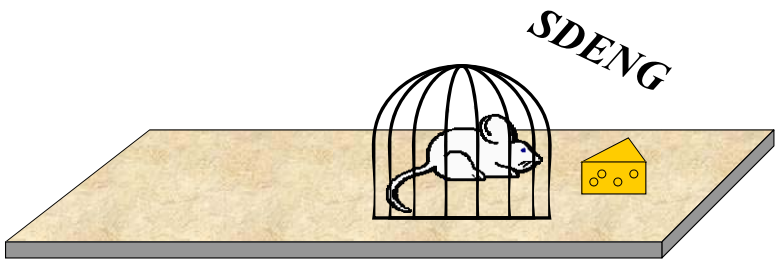
- A real-time system is **not** a fast system.
- Speed is always relative to a specific environment.
- Running faster is good, but does not guarantee a correct behavior.

Biological examples

- Living beings react in real-time to events in their **natural habitat** independently of their speed:

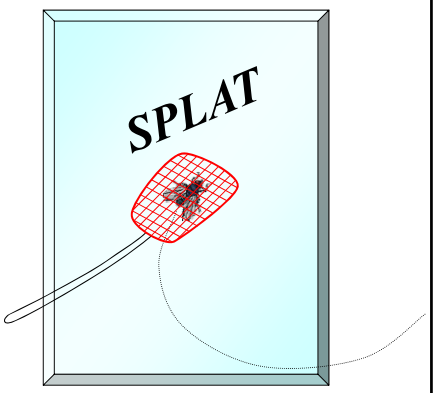


- If we insert “anomalous” events in the environment, even a fast mouse can fail:



- … or a faster fly

  can be catched!



What do we learn?

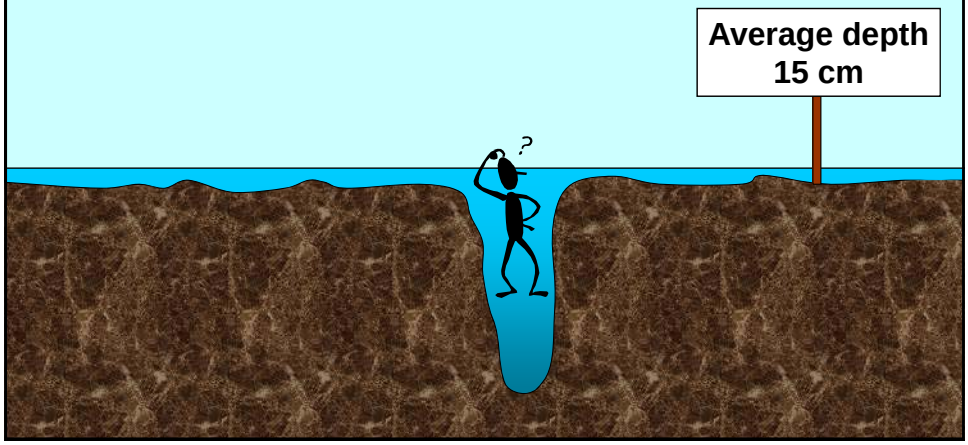
A RT system must be designed together with the environment in which it operates

Speed vs. Predictability

- The objective of a real-time system is to guarantee the timing behavior 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

There was a man who drowned crossing a stream with an average depth of 15 cm.



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

Traditional Approach

- In spite of this large application domain, most of RT applications are designed using empirical techniques:
  - assembly programming
  - timing through dedicated timers
  - control through driver programming
  - priority manipulations

Disadvantages

1. Tedious programming which heavily depends on programmer's ability
2. Difficult code understanding

$$\text{Readability} \propto \frac{1}{\text{efficiency}}$$

An efficient C program

```
int a[1817];
main(z,p,q,r)
{
 for(p=80;q+p-80;p-=2*a[p])
  for(z=9;z--;)
   q=3&(r=time(0)+r*57)/7,q=q?q-1?
     q-2?1-p%79?-1:0:p%79-77?
   1:0:p<1659?79:0:p>158?-79:0,
     q?!a[p+q*2]?a[p+=a[p+=q]=q]=q :0:0;
 for(;q++-1817;)
  printf(q%79?"%c":"%c\n"," Û"[!a[q-1]]);
}
```

Disadvantages

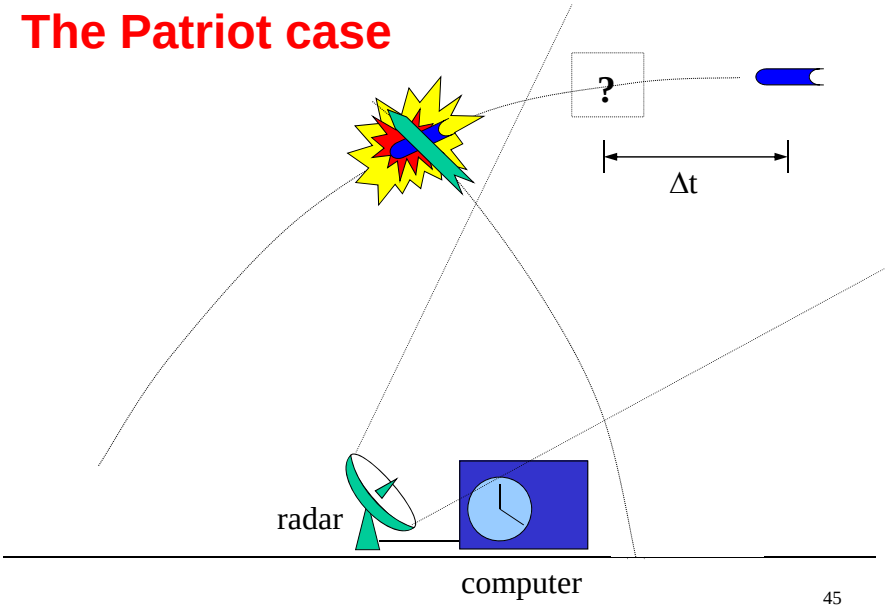
3. Difficult software maintainability
   - Complex appl.s consists of millions lines of code
   - Code understanding takes more that re-writing
   - But re-writing is VERY expensive and bug prone
4. Difficult to verify timing constraints without explicit support from the OS and the language

Implications

- Such a way of programming RT applications is very dangerous.
- It may work in most situations, but the risk of a failure is high.
- When the system fails is very difficult to understand why.

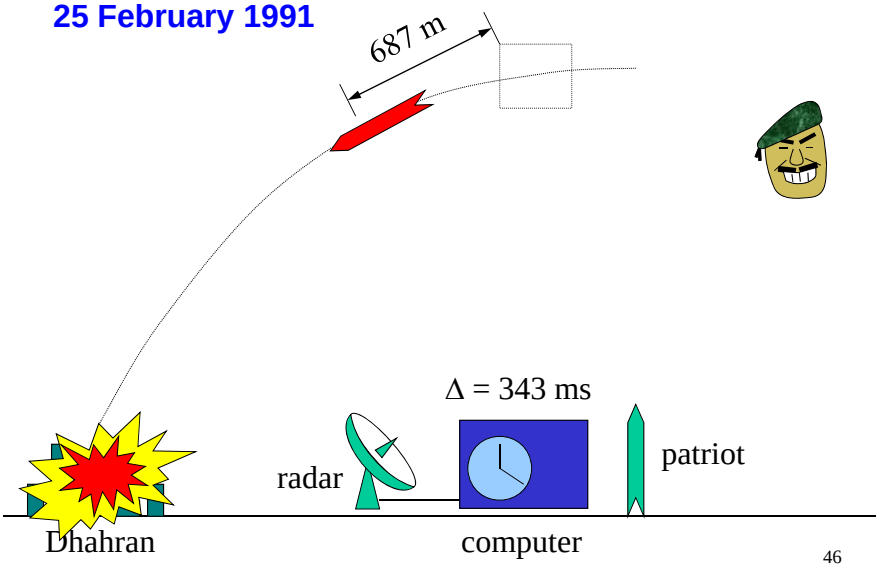
low reliability

The Patriot case



Gulf War

25 February 1991



Other accidents due to SW

- Task overrun during LEM lunar landing
- First flight of the Space Shuttle (synch)
- Ariane 5 (overflow)
- Airbus 320 (cart task)
- Airbus 320 (holding task)
- Pathfinder (reset for timeout)

Lessons learned

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

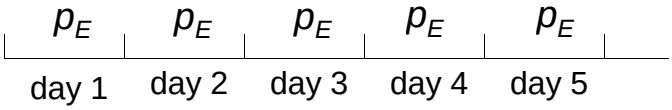
If something can go wrong, it will go wrong

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

Why Murphy's Laws are true

What's the probability for E to occur in n days?



prob. for E not to occur in a day: $q_E = 1 - p_E$

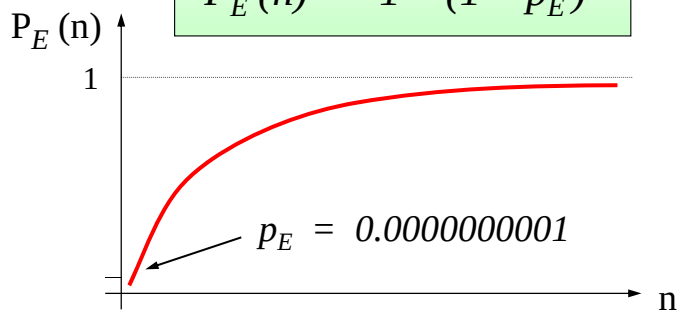
prob. for E not to occur in n days: $Q_E(n) = (1 - p_E)^n$

prob. for E to occur in n days: $P_E(n) = 1 - Q_E(n)$

Understanding Murphy's Laws

If something can go wrong (no matter how small p_E is), it will go wrong (that is, the probability for E to occur in long time intervals tends to 1).

$$P_E(n) = 1 - (1 - p_E)^n$$



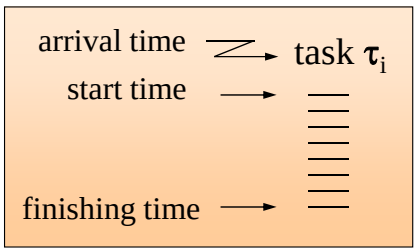
Let's studying real-time scheduling

and fight against Murphy's Laws

Definitions

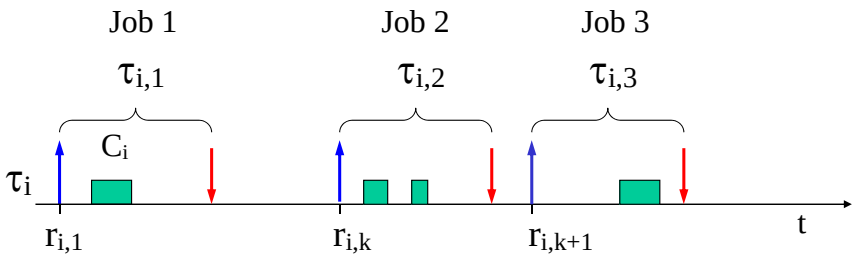
- **Task** (or thread)

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

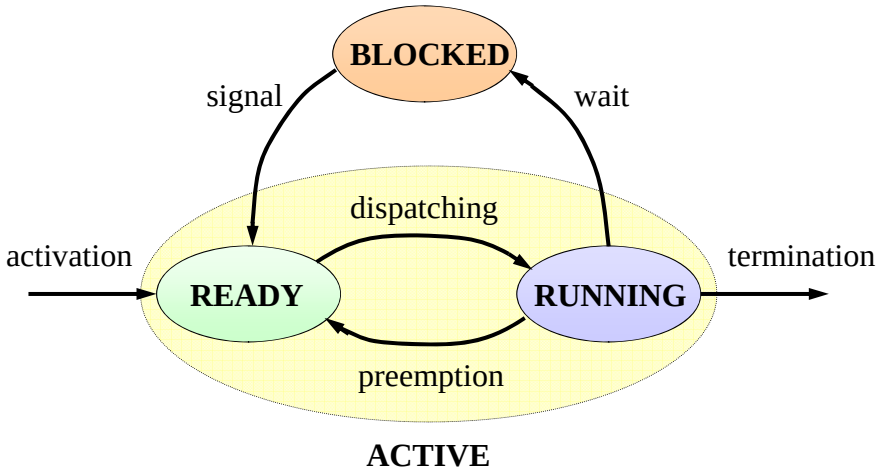


Tasks and jobs

A task is an infinite sequence of instances (jobs):

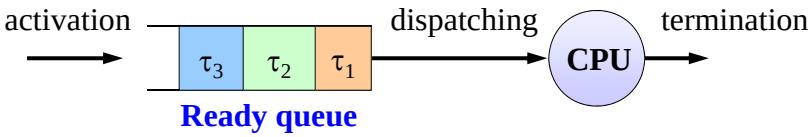


Task State Transitions



Ready Queue

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



Scheduling

- A scheduling algorithm is said to be:
  - **preemptive**: if the running task can be temporarely suspended in the ready queue to execute a more important task.
  - **non preemptive**: if the running task cannot be suspended until completion.

Schedule

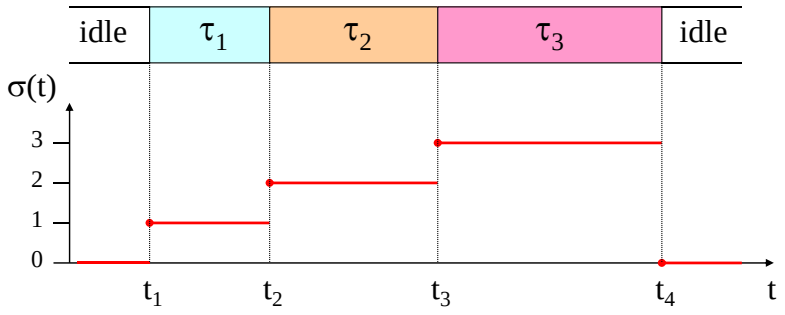
A **schedule** is a particular assignment of tasks to the processor.

Given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a schedule is a mapping $\sigma : \mathbf{R}^+ \rightarrow \mathbf{N}$ such that $\forall t \in \mathbf{R}^+, \exists t_1, t_2$:

$$t \in [t_1, t_2) \quad e \quad \forall t' \in [t_1, t_2) : \sigma(t) = \sigma(t')$$

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

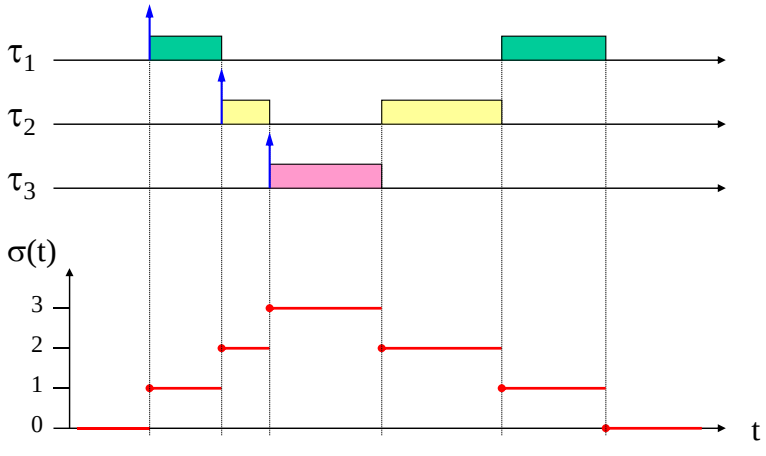
A sample schedule



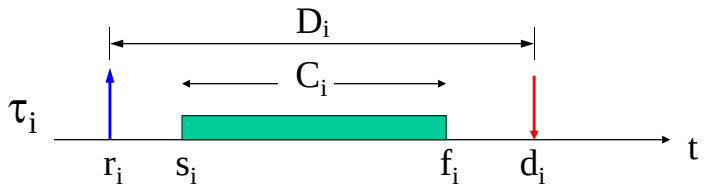
At time t_1, t_2, t_3, e t_4 a **context switch** is performed.

Each interval $[t_i, t_{i+1})$ is called a **time slice**.

A preemptive schedule

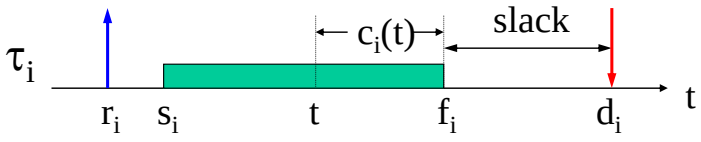


Real-Time tasks



$\mathbf{r_i}$	request time (arrival time $\mathbf{a_i}$)
$\mathbf{s_i}$	start time
$\mathbf{C_i}$	worst-case execution time (wcet)
$\mathbf{d_i}$	absolute deadline
$\mathbf{D_i}$	relative deadline
$\mathbf{f_i}$	finishing time

Other parameters



Lateness: $L_i = f_i - d_i$

Tardiness: $\max(0, L_i)$

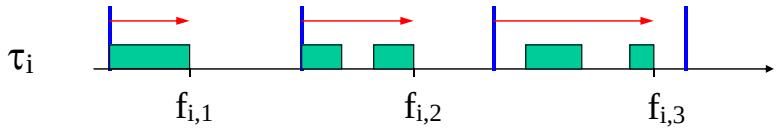
Residual wcet: $c_i(t)$ $c_i(r_i) = C_i$

Laxity (o slack): $d_i - t - c_i(t)$

Jitter

It is the time variation of a periodic event:

Finishing-time Jitter

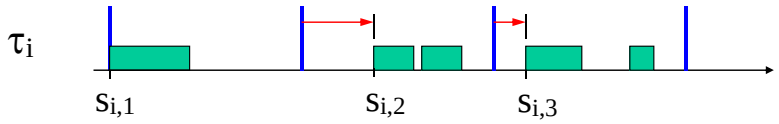


Absolute: $\max_k (f_{i,k} - r_{i,k}) - \min_k (f_{i,k} - r_{i,k})$

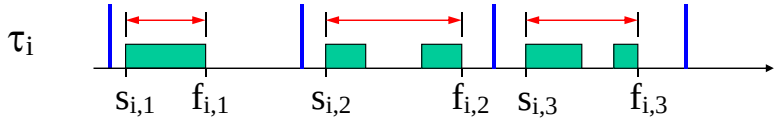
Relative: $\max_k \left| (f_{i,k} - r_{i,k}) - (f_{i,k-1} - r_{i,k-1}) \right|$

Other types of Jitter

Start-time Jitter



Completion-time Jitter (I/O jitter)



Task Criticality

HARD tasks

All jobs must meet their deadlines. Missing a deadline may cause catastrophical effects.

SOFT tasks

Missing deadlines is not desired but causes only a performace degradation.

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

Typical HARD tasks

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

Typical SOFT tasks

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

Activation modes

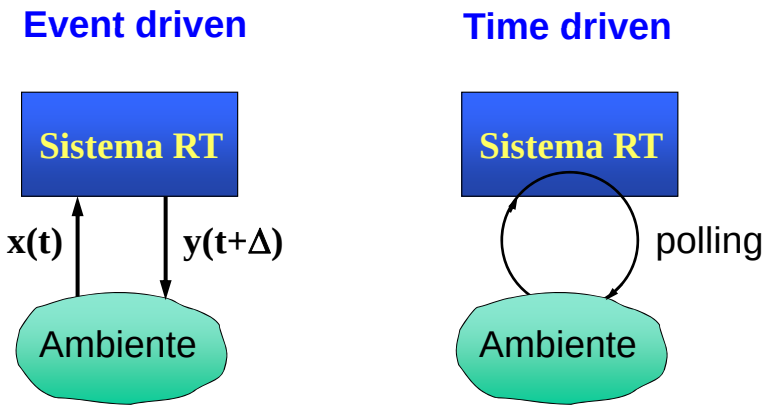
- **Time driven**: **periodic** tasks

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

- **Event driven**: **aperiodic** tasks

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

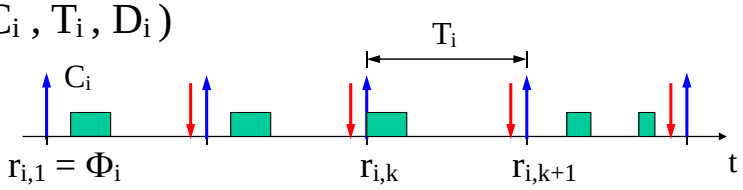
Two different approaches



Periodic task model

$$\begin{cases} r_{i1} = \Phi_i \\ r_{i,k+1} = r_{i,k} + T_i \end{cases}$$

$\tau_i (C_i , T_i , D_i)$

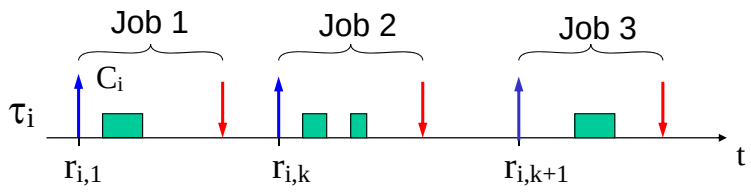


$$r_{i,k} = \Phi_i + (k-1)\, T_i$$
$$d_{i,k} = r_{i,k} + D_i$$

often
$D_i = T_i$

Aperiodic task model

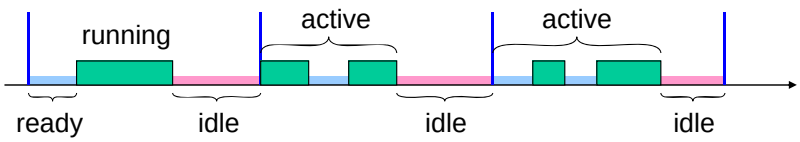
- **Aperiodic:** $r_{i,k+1} > r_{i,k}$
- **Sporadic:** $r_{i,k+1} \geq r_{i,k} + T_i$



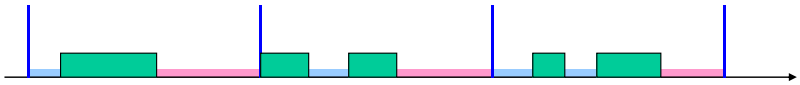
OS support for periodic tasks

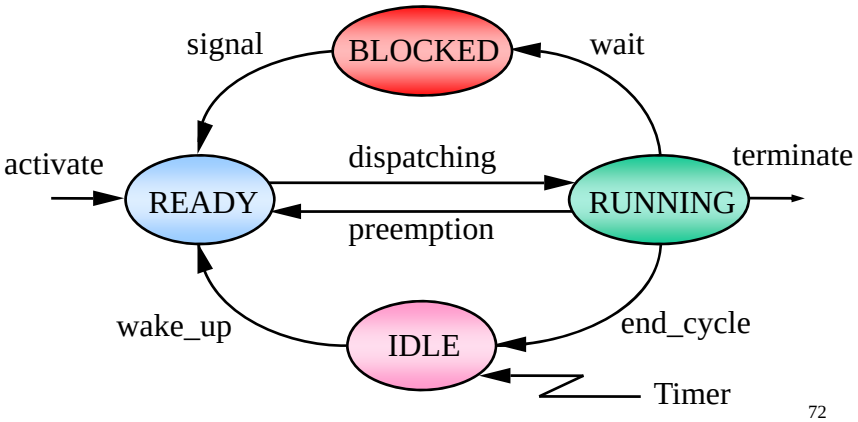
task τ_i

```
while (condition) {
    ——
    ——
    ——
    ——
    ——
    wait_for_next_period();
}
```



The IDLE state





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

Can be explicit or implicit.

- **Explicit constraints**
  - Are included in the specification of the system activities.

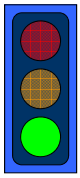
Examples

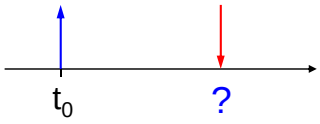
- open the valve **in** 10 seconds
- send the position **within** 40 ms
- read the altimeter **every** 200 ms

- **Implicit constraints**
  - do not appear in the system specification, but must be respected to meet the requirements.

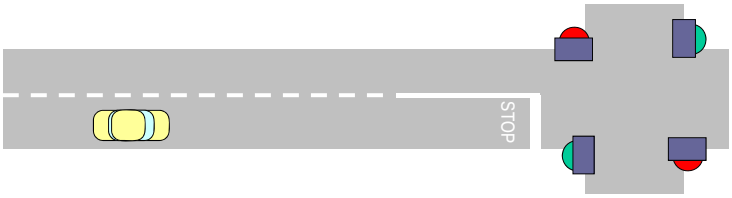
Example

What's the time validity of a sensory data?





Duration of the yellow light



$$D \geq T_d + T_r + T_b$$

T_d = detection time
T_r = reaction time
T_b = braking time

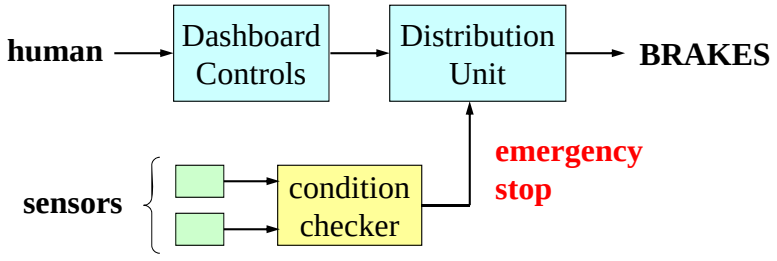
Duration of the yellow light

Detection time: $T_d = 1$ s
Reaction time: $T_r = 1$ s
Braking time: $T_b = v/(\mu g)$

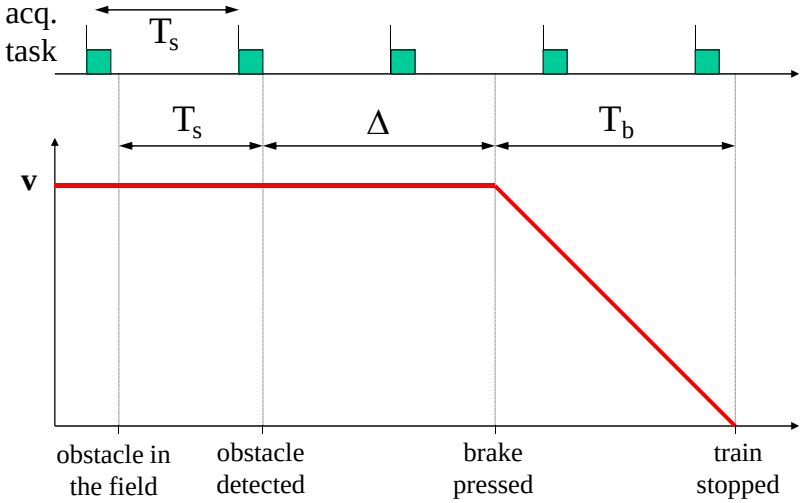
For $\begin{cases} v = 50 \text{ Km/h} = 14 \text{ m/s} \\ \mu = 0.5 \end{cases}$ we have: $T_f = 2.8$ s

Time to stop the car from the yellow: **D ≅ 5 s**

Example: automatic braking



Worst-case reasoning



D = sensor visibility

$$v(T_s + \Delta) + X_b < D$$

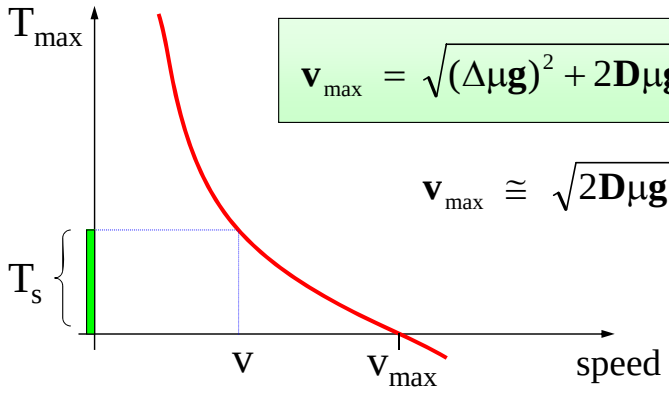
$$\begin{cases} X_b = vt - \frac{1}{2}at^2 \\ v = a\,t \end{cases}$$

$$a = \mu g$$

$$X_b = \frac{v^2}{2\mu g}$$

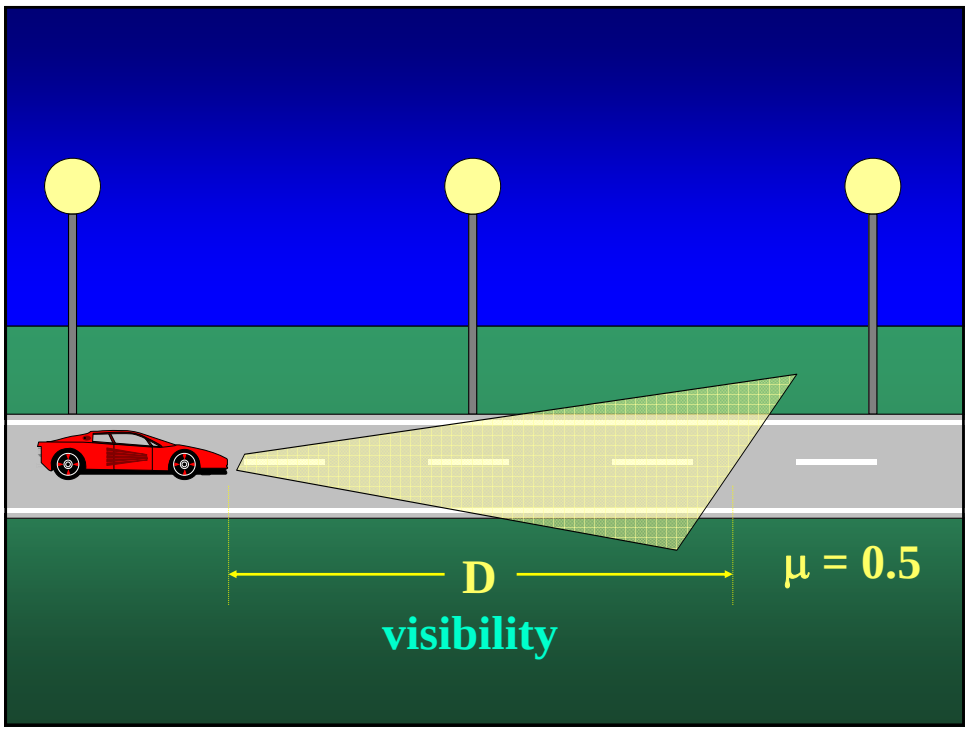
$$v(T_s + \Delta) + \frac{v^2}{2\mu g} < D$$

$$T_s < \frac{D}{v} - \frac{v}{2\mu g} - \Delta$$

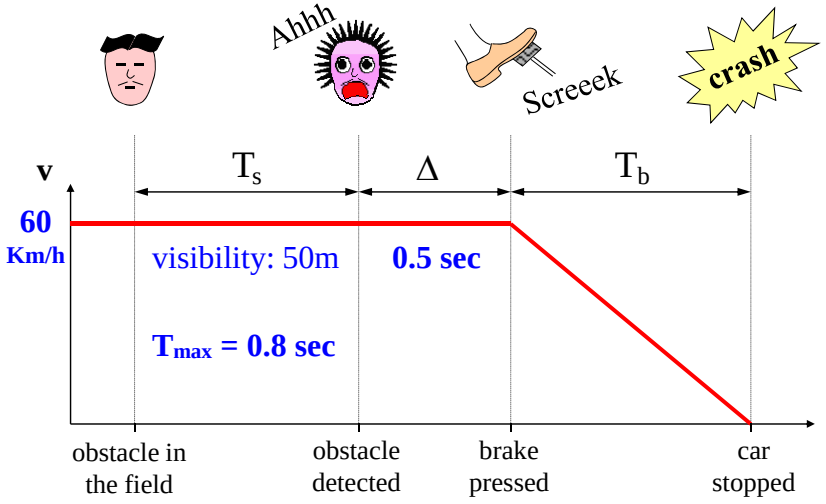


$$\mathbf{v}_{max} = \sqrt{(\Delta\mu\mathbf{g})^2 + 2\mathbf{D}\mu\mathbf{g}} - \Delta\mu\mathbf{g}$$

$$\mathbf{v}_{max} \cong \sqrt{2\mathbf{D}\mu\mathbf{g}}$$



Car driving



Lessons learned

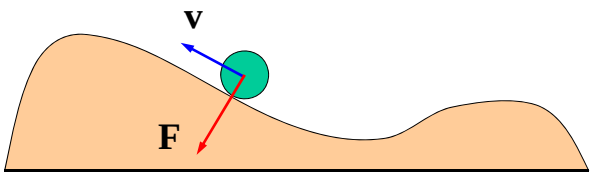
The farther we look, the faster we can run

To go fast safely, look ahead!!!

If $v \geq v_{max}$ no feasible solution exists, no matter how fast you react!!!

Don't look away from the road for too long!!!

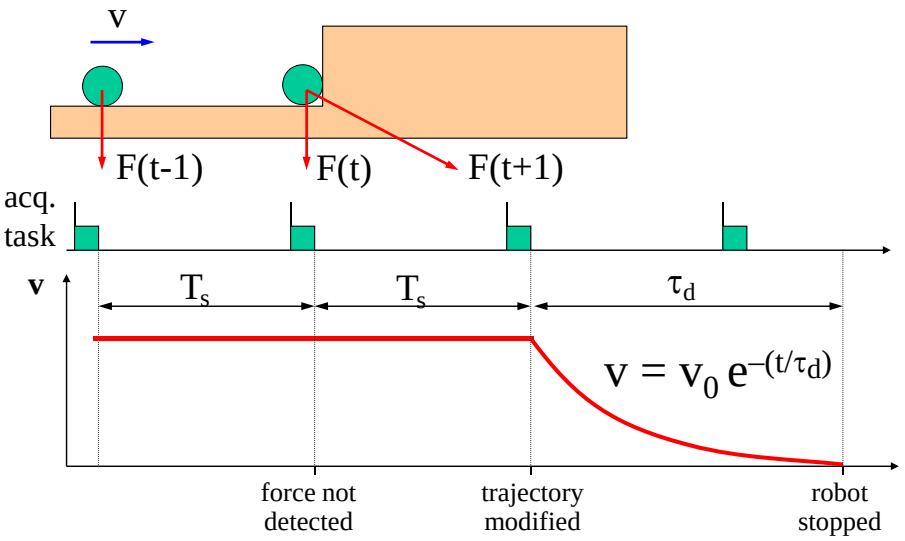
Esempio2: contour following



Goal

Move at velocity **v** along the surface tangent, exerting a force $F < F_{max}$ along its normal direction.

Worst-case reasoning



Lenght covered by the robot after the contact:

$$L = vT_s + x_f$$

$$x_f = \int_0^\infty v(t)dt = \int_0^\infty v_0 e^{-t/\tau_d} dt = -v_0\tau_d(e^{-\infty} - e^0) = v_0\tau_d$$

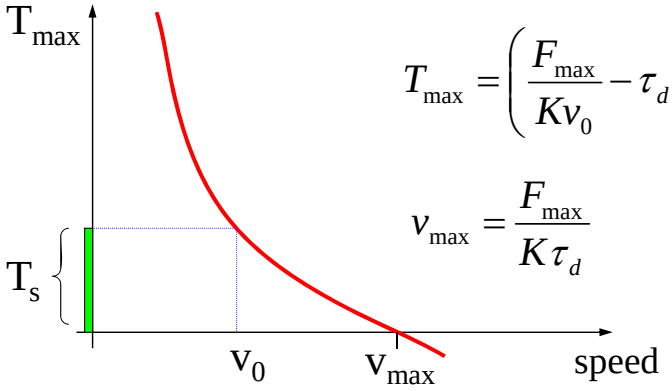
$$L = v(T_s + \tau_d)$$

Force on the robot tool: (K = elastic coefficient)

$$F = KL = Kv(T_s + \tau_d) < F_{max}$$

Condition on the sampling period:

$$T_s < \frac{F_{max}}{Kv_0} - \tau_d$$



$$T_{max} = \left(\frac{F_{max}}{Kv_0} - \tau_d\right)$$

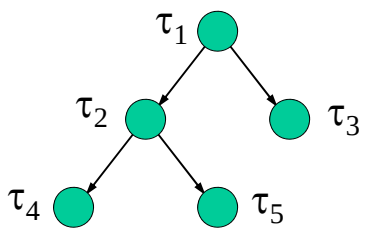
$$v_{max} = \frac{F_{max}}{K\tau_d}$$

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.

Precedence constraints

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



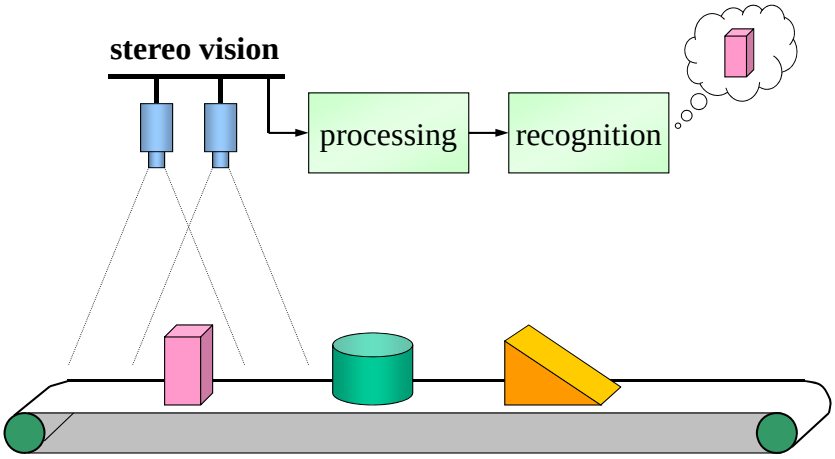
predecessor

$\tau_1 \prec \tau_4$

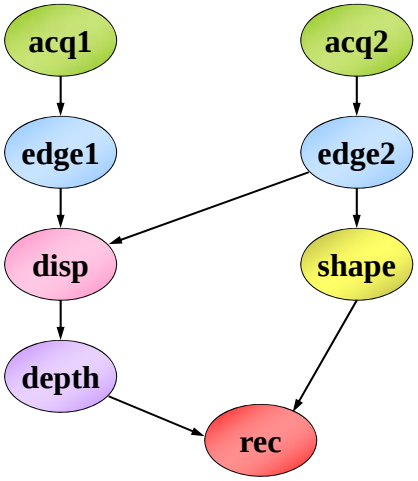
immediate predecessor

$\tau_1 \rightarrow \tau_2$

Sample application



Precedence graph

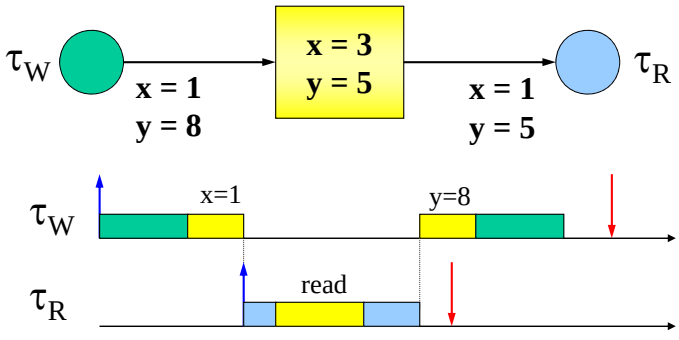


Types of constraints

- **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.

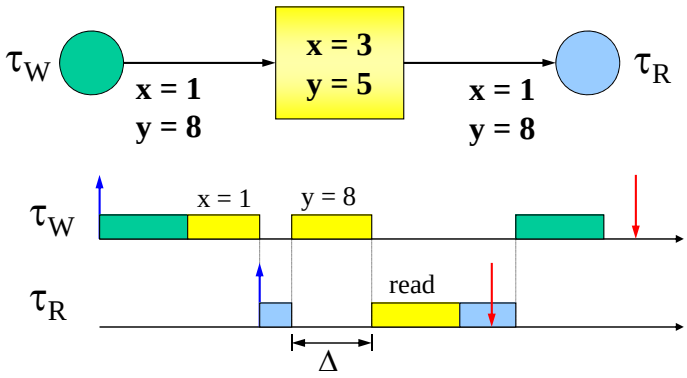
Resource constraints

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

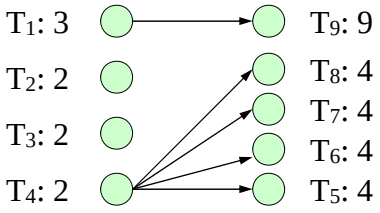


Mutual exclusion

However, mutual exclusion introduces extra delays:

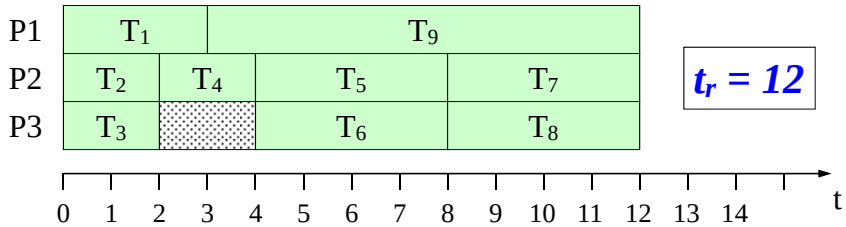


Scheduling anomalies



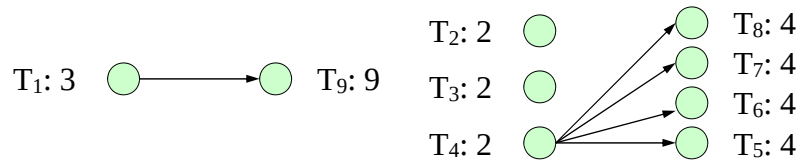
priority

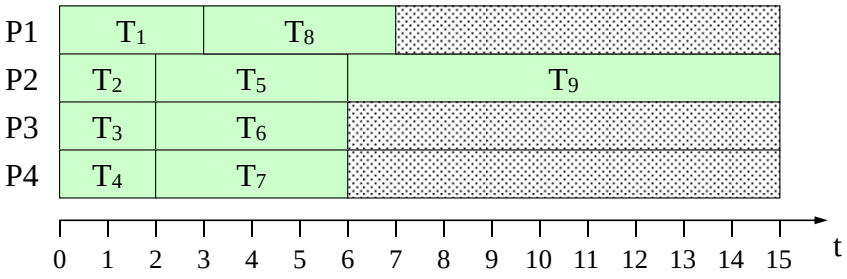
$P_i > P_j \quad \forall\, i < j$



$t_r = 12$

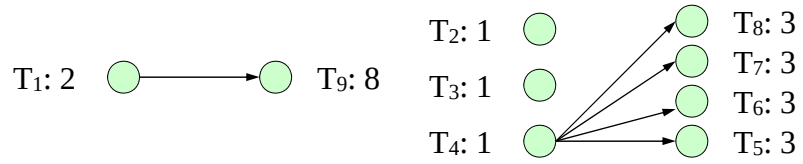
Increased processors

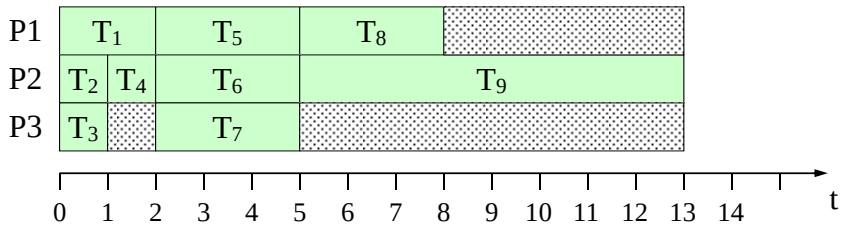




$t_r = 15$

Shorter tasks





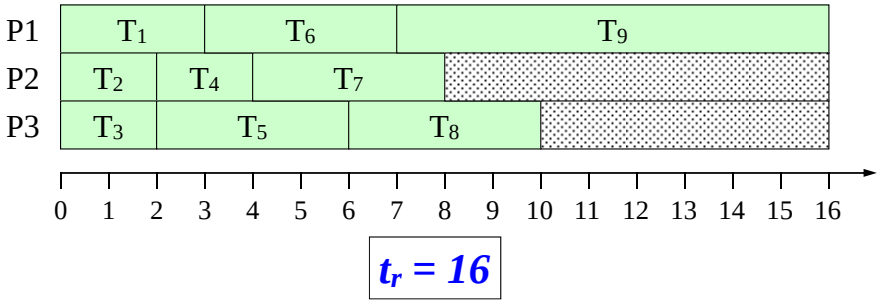
$t_r = 13$

Released constraints

T_1: 3 T_4: 2 T_7: 4

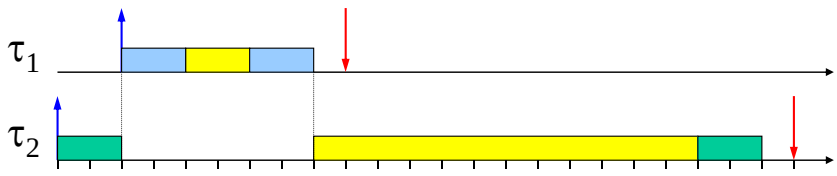
T_2: 2 T_5: 4 T_8: 4

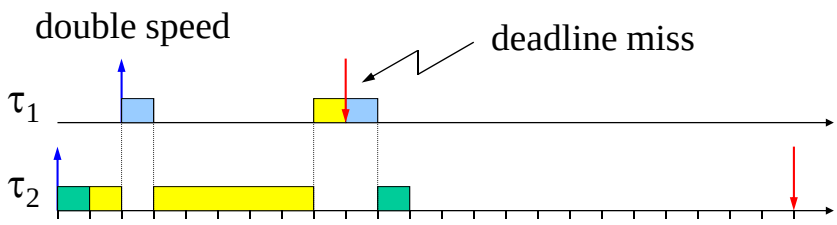
T_3: 2 T_6: 4 T_9: 9



$t_r = 16$

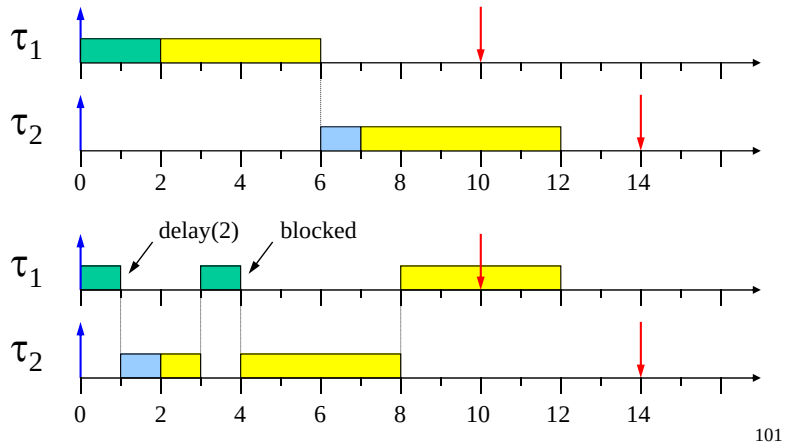
Faster processor





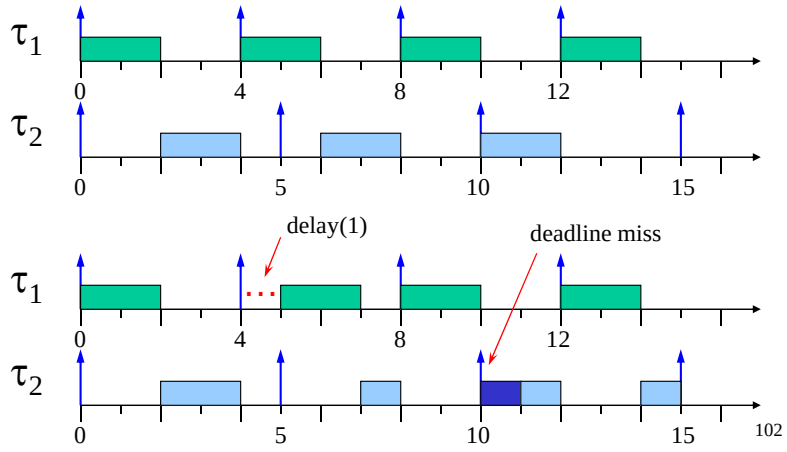
A dangerous operation: DELAY

A **delay(Δ)** may cause a delay longer than Δ.



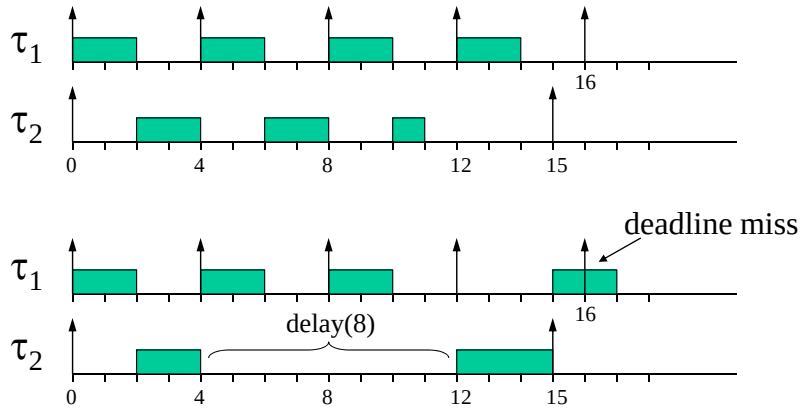
A dangerous operation: DELAY

A **delay** in a task may also increase the response time of other tasks (example for fixed priorities):



A dangerous operation: DELAY

A **delay** in a task may also increase the response time of other tasks (example for deadline scheduling):



Lessons learned

- 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:

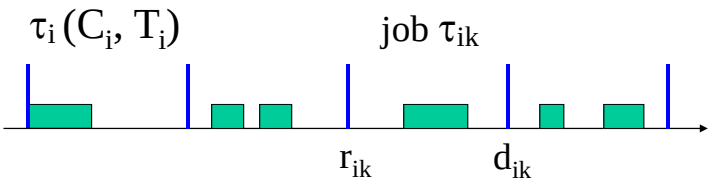
- **use predictable kernel mechanisms**
- **analyze the system to predict its behavior**

Achieving predictability

- 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

Periodic Task Scheduling

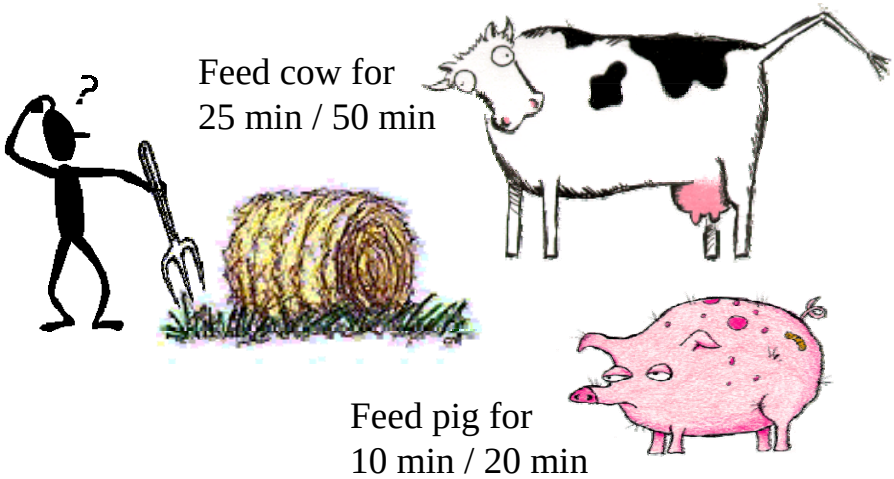
Problem formulation



For each periodic task, guarantee that:

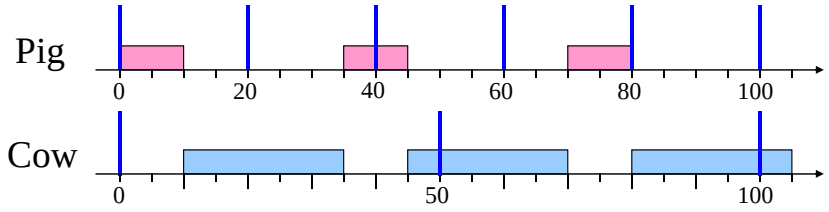
- each job τ_{ik} is activated at $r_{ik} = (k-1)T_i$
- each job τ_{ik} completes within $d_{ik} = r_{ik} + D_i$

A farm scheduling problem



First algorithm

Alternate pig with cow



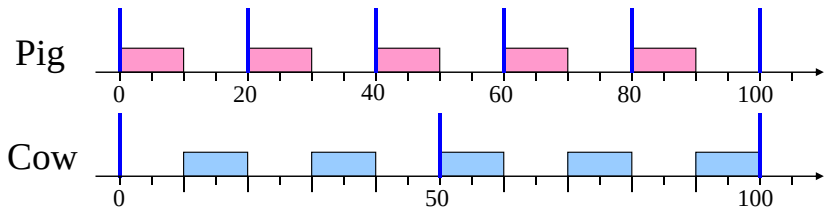
Evaluation:

Pig gets hungry

Cow gets fat

Second algorithm

Feed pig and cow 10 min each



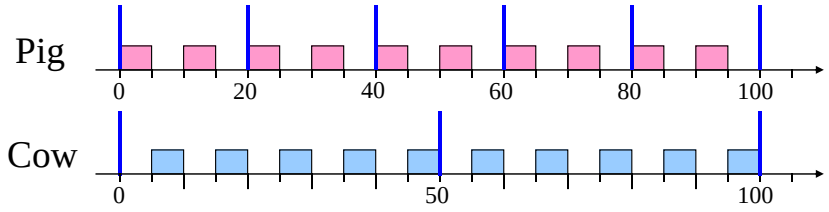
Evaluation:

Pig is OK

Cow is not happy

Third algorithm

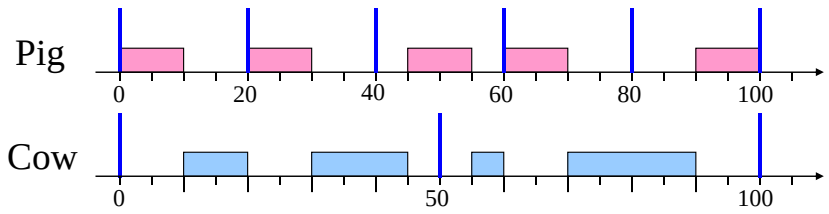
Feed pig and cow 5 min each



Evaluation: Pig is OK, Cow is OK but the farmer is tired

Optimal algorithm

Feed the most starving animal



Evaluation: Everybody is happy

Timeline Scheduling (cyclic scheduling)

It has been used for 30 years in military systems, navigation, and monitoring systems.

Examples

- Air traffic control
- Space Shuttle
- Boeing 777

Timeline Scheduling

Method

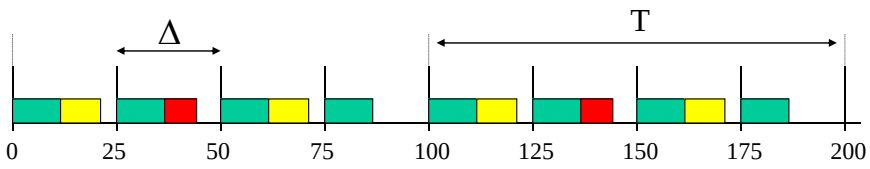
- The time axis is divided in intervals of equal length (***time slots***).
- Each task is statically allocated in a slot in order to meet the desired request rate.
- The execution in each slot is activated by a timer.

Example

task	f	T
A	40 Hz	25 ms
B	20 Hz	50 ms
C	10 Hz	100 ms

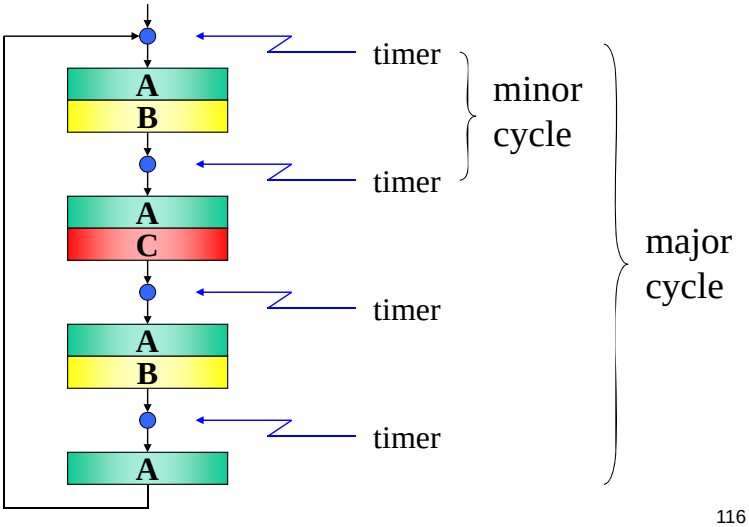
Δ = GCD (minor cycle)

T = lcm (major cycle)



Guarantee: $C_A + C_B \leq \Delta$, $C_A + C_C \leq \Delta$

Implementation



Timeline scheduling

Advantages

- Simple implementation (no real-time operating system is required).
- Low run-time overhead.
- It allows jitter control.

Timeline scheduling

Disadvantages

- It is not robust during overloads.
- It is difficult to expand the schedule.
- It is not easy to handle aperiodic activities.

Problems during overloads

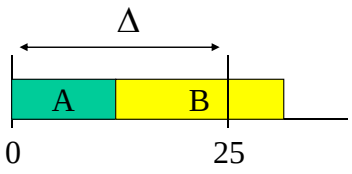
What do we do during task overruns?

- Let the task continue
  - we can have a ***domino effect*** on all the other tasks (timeline break)
- Abort the task
  - the system can remain in inconsistent states.

Expandibility

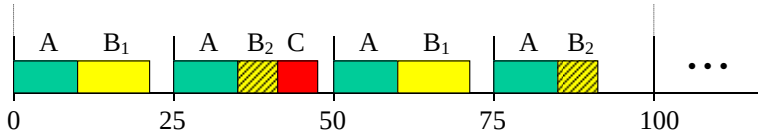
If one or more tasks need to be upgraded, we may have to re-design the whole schedule again.

Example: B is updated but $C_A + C_B > \Delta$



Expandibility

- We have to split task B in two subtasks (B_1, B_2) and re-build the schedule:



Guarantee: $\begin{cases} C_A + C_{B1} \leq \Delta \\ C_A + C_{B2} + C_C \leq \Delta \end{cases}$

Expandibility

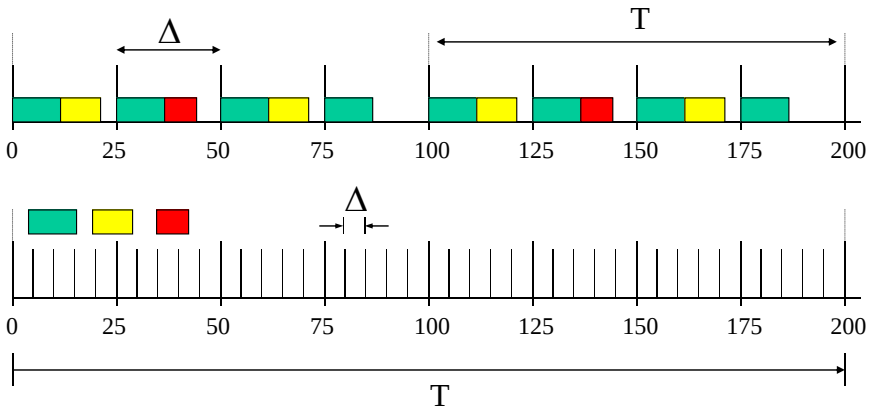
If the frequency of some task is changed, the impact can be even more significant:

task	T	T
A	25 ms	25 ms
B	50 ms	**40 ms**
C	100 ms	100 ms
	before	**after**

minor cycle:	$\Delta = 25$	$\Delta = 5$
major cycle:	$T = 100$	$T = 200$

(40 sync. per cycle!)

Example



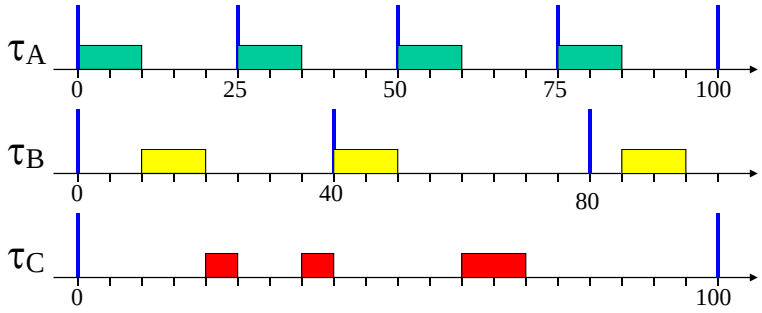
Priority Scheduling

Method

- Each task is assigned a priority based on its timing constraints.
- We verify the feasibility of the schedule using analytical techniques.
- Tasks are executed on a priority-based kernel.

Rate Monotonic (RM)

- Each task is assigned a fixed priority proportional to its rate [Liu & Layland '73].

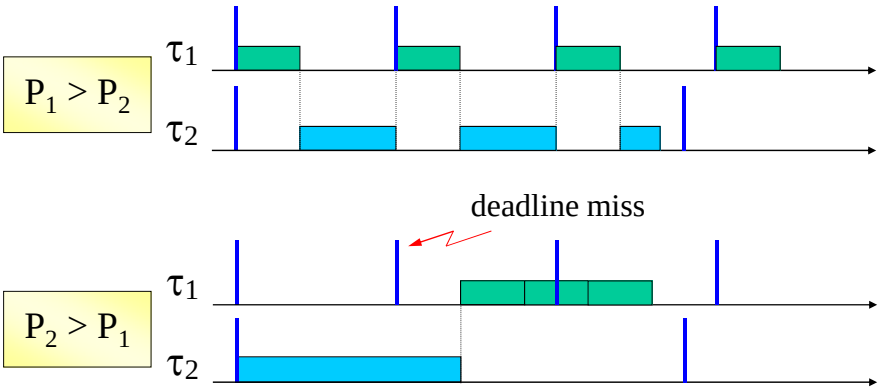


How to assign priorities?

- Typically, task priorities are assigned based on the their relative importance.
- However, different priority assignments can lead to different utilization bounds.

Priority vs. importance

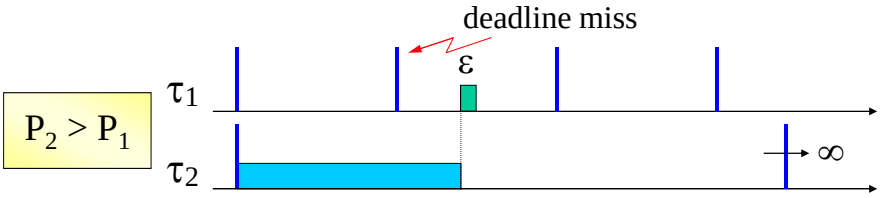
If τ_2 is more important than τ_1 and is assigned higher priority, the schedule may not be feasible:



Priority vs. importance

But the utilization upper bound can be arbitrarily small:

An application can be unfeasible even when the processor is almost empty!



$$U = \frac{\varepsilon}{T_1} + \frac{C_2}{\infty} \rightarrow 0$$

Rate Monotonic is optimal

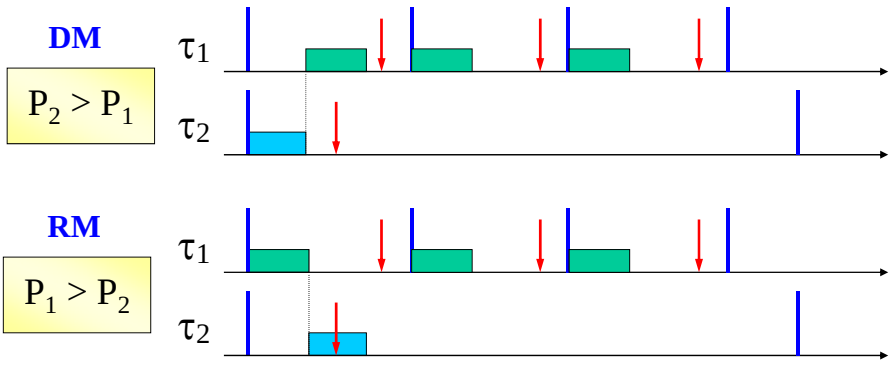
RM is **optimal** among all fixed priority algorithms (if $D_i = T_i$):

If there exists a fixed priority assignment which leads to a feasible schedule for Γ, then the RM assignment is feasible for Γ.

If Γ is not schedulable by RM, then it cannot be scheduled by any fixed priority assignment.

Deadline Monotonic is optimal

If $D_i \leq T_i$ then the **optimal** priority assignment is given by **Deadline Monotonic** (**DM**):



EDF Optimality

EDF is **optimal** among all algorithms:

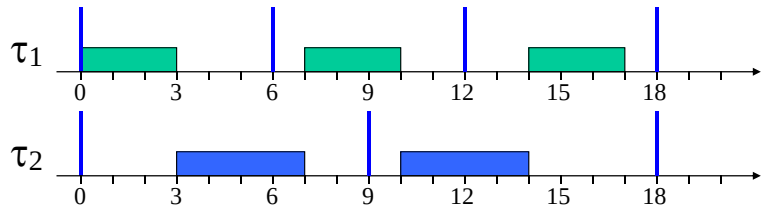
If there exists a feasible schedule for Γ, then EDF will generate a feasible schedule.

If Γ is not schedulable by EDF, then it cannot be scheduled by any algorithm.

EDF Example

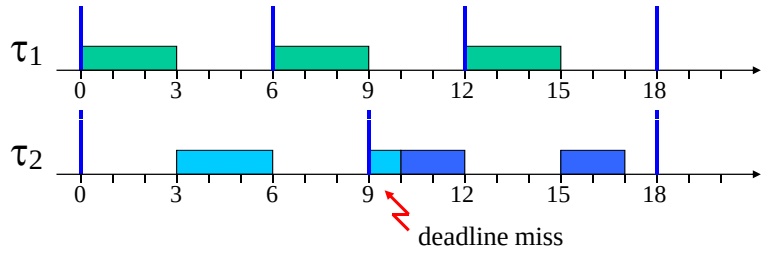
$D_i = T_i$

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.94$$



The RM unfesible schedule

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



Priority Assignments

- **Rate Monotonic (RM):**

  $P_i \propto 1/T_i$ (static)

- **Deadline Monotonic (DM):**

  $P_i \propto 1/D_i$ (static)

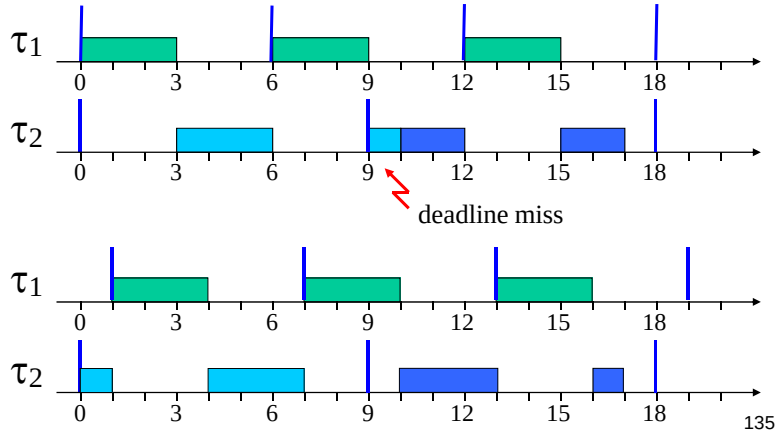
- **Earliest Deadline First (EDF):**

  $P_i \propto 1/d_{ik}$ (dynamic) $d_{i,k} = r_{i,k} + D_i$

Identifying the worst case

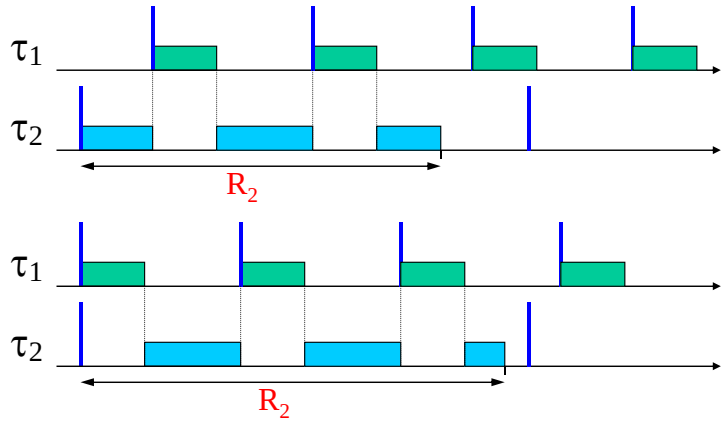
Feasibility may depend on the initial activations (phases):

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



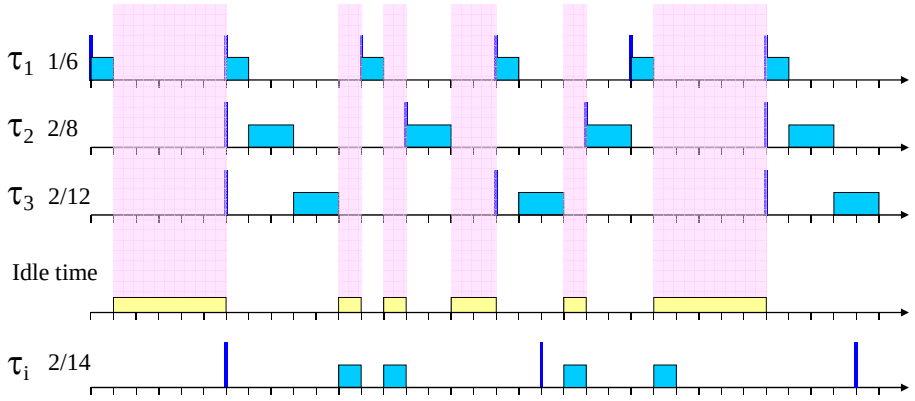
Critical Instant

For any task τ_i, the longest response time occurs when it arrives together with all higher priority tasks.



Critical Instant

For independent preemptive tasks under fixed priorities, the critical instant of τ_i, occurs when it arrives together with all higher priority tasks.



How can we verify feasibility?

- Each task uses the processor for a fraction of time:

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

- Hence the total **processor utilization** is:

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

- U_p is a misure of the **processor load**

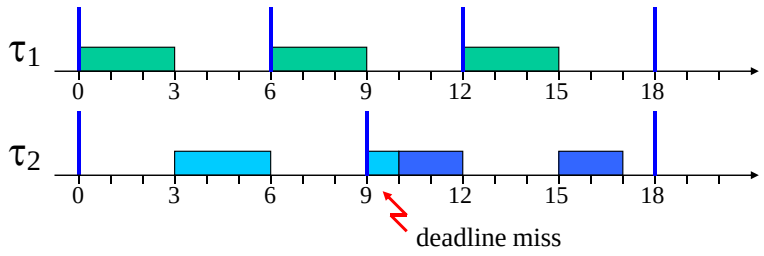
A necessary condition

If $\mathbf{U_p} > \mathbf{1}$ the processor is overloaded hence the task set cannot be schedulable.

However, there are cases in which $\mathbf{U_p} < \mathbf{1}$ but the task is not schedulable by RM.

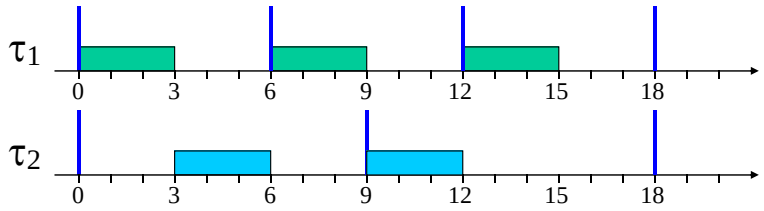
An unfeasible RM schedule

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



Utilization upper bound

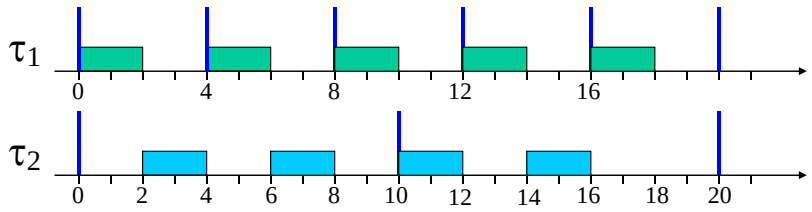
$$U_p = \frac{3}{6} + \frac{3}{9} = 0.833$$



NOTE: If C_1 or C_2 is increased, τ_2 will miss its deadline!

A different upper bound

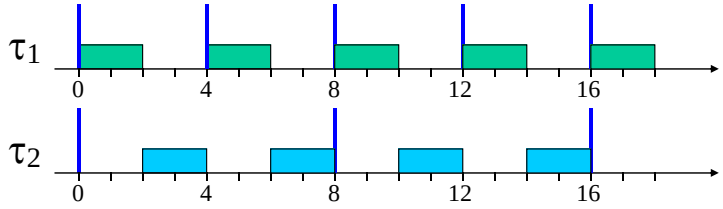
$$U_{ub} = \frac{2}{4} + \frac{4}{10} = 0.9$$



The upper bound U_{ub} depends on the specific task set.

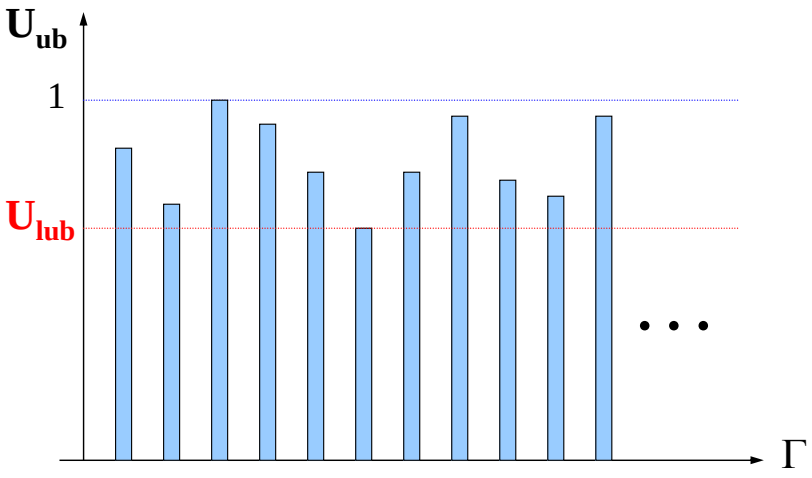
A different upper bound

$$U_p = \frac{2}{4} + \frac{4}{8} = 1$$



The upper bound U_{ub} depends on the specific task set.

The least upper bound



A sufficient condition

If $\mathbf{U_p \leq U_{lub}}$ the task set is certainly schedulable with the RM algorithm.

NOTE

If $\mathbf{U_{lub} < U_p \leq 1}$ we cannot say anything about the feasibility of that task set.

Basic results

In 1973, Liu & Layland proved that a set of ***n*** periodic tasks can be feasibly scheduled

under RM **if** $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$

under EDF **if and only if** $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$

Assumptions:
- Independent tasks
- $\Phi_i = 0$
- $D_i = T_i$

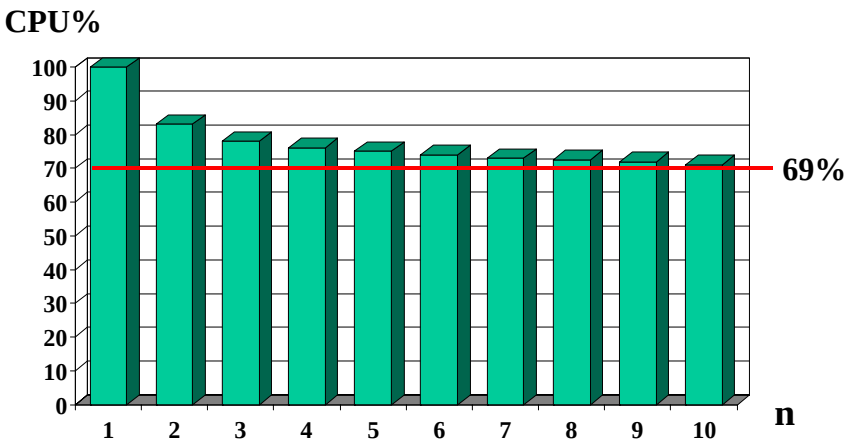
U_{lub} for RM

- In 1973, **Liu and Layland** proved that for a set of n periodic tasks:

$$U_{\text{lub}}^{RM} = n\left(2^{1/n} - 1\right)$$

for $n \to \infty$ $U_{lub} \to ln\ 2$

RM Schedulability



RM Guarantee Test

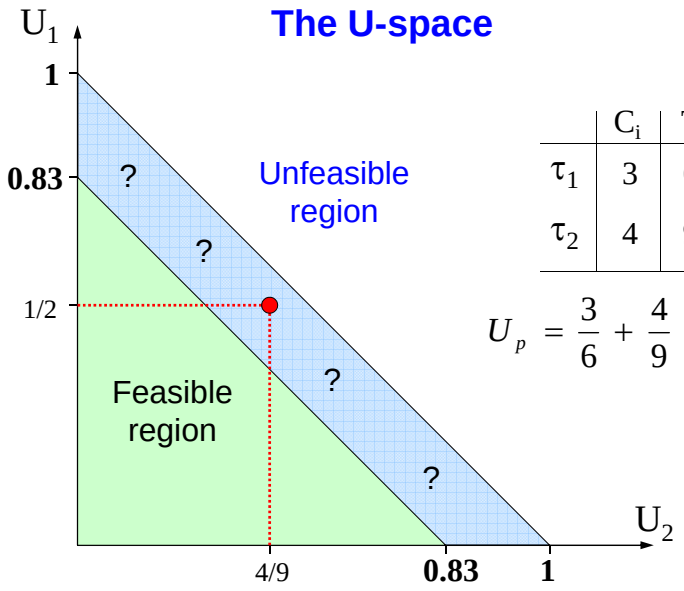
- We compute the processor utilization as:

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

- Guarantee Test (only sufficient):

$$U_p \leq n\left(2^{1/n} - 1\right)$$

Schedulability region for RM



	C_i	T_i
τ_1	3	6
τ_2	4	9

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.94$$

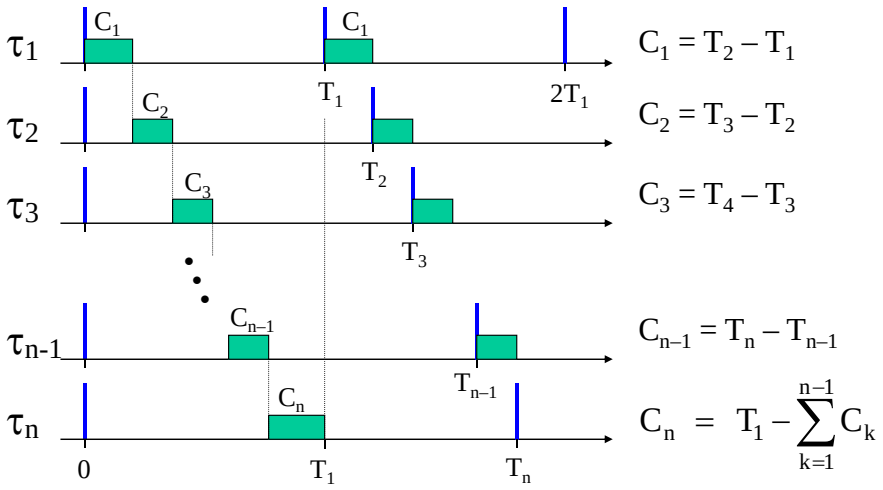
The Hyperbolic Bound

- In 2000, **Bini et al.** proved that a set of *n* periodic tasks is schedulable with RM if:

$$\prod_{i=1}^{n} (U_i + 1) \leq 2$$

Proof sketch

$$T_1 < T_i < 2T_1$$



Proof sketch

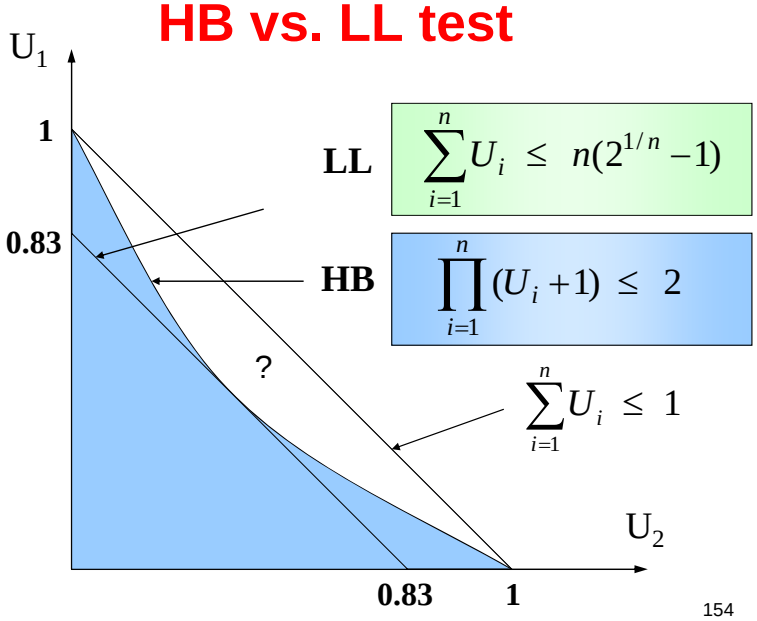
$C_i = T_{i+1} - T_i \Rightarrow U_i = R_i - 1 \Rightarrow R_i = U_i + 1$

$\sum_{k=1}^{n} C_k \le T_1 \Rightarrow C_n + \sum_{k=1}^{n-1} C_k \le T_1 \Rightarrow C_n + T_n \le 2T_1$

$$\frac{T_n}{T_1}(U_n + 1) \le 2$$

$$\frac{T_n}{T_1} = \prod_{i=1}^{n-1} R_i = \prod_{i=1}^{n-1} (U_i + 1) \Rightarrow \boxed{\prod_{i=1}^{n} (U_i + 1) \le 2}$$

HB vs. LL test



Response Time Analysis
[Audsley '90]

- For each task τ_i compute the interference due to higher priority tasks:

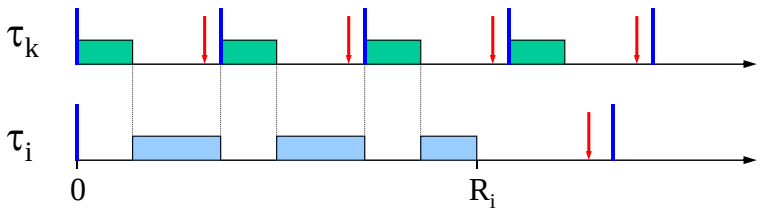
$$I_i = \sum_{D_k < D_i} C_k$$

- compute its response time as

$$\mathbf{R_i = C_i + I_i}$$

- verify if $\mathbf{R_i \leq D_i}$

Computing Interference



Interference of τ_k on τ_i in the interval $[0, R_i]$:

$$I_{ik} = \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Interference of high priority tasks on τ_i:

$$I_i = \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Computing Response Time

$$R_i = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Iterative solution:

$$\begin{cases} R_i^0 = C_i \\ R_i^s = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^{(s-1)}}{T_k} \right\rceil C_k \end{cases}$$

iterate while $R_i^s > R_i^{(s-1)}$

Dynamic Priority Scheduling

Earliest Deadline First (EDF)

- Each job receives an absolute deadline:

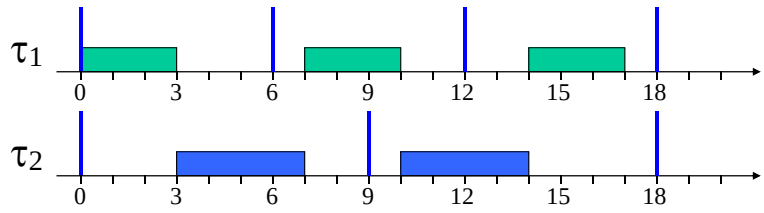
$$d_{i,k} = r_{i,k} + D_i$$

- At any time, the processor is assigned to the job with the earliest absolute deadline.
- Under EDF, any task set can utilize the processor up to 100%.

EDF Example

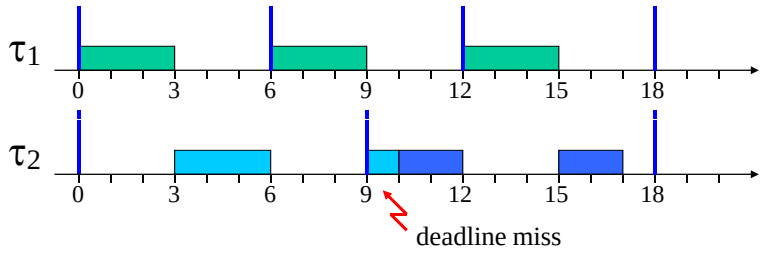
$D_i = T_i$

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.94$$



The RM unfesible schedule

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



EDF schedulability

- In 1973, **Liu and Layland** proved that for a set of n periodic tasks:

$$U_{\text{lub}}^{EDF} = 1$$

- This means that a task set Γ is schedulable by EDF **if and only if**

$$\mathbf{U_p \leq 1}$$

Dynamic Priority

EDF with D ≤ T

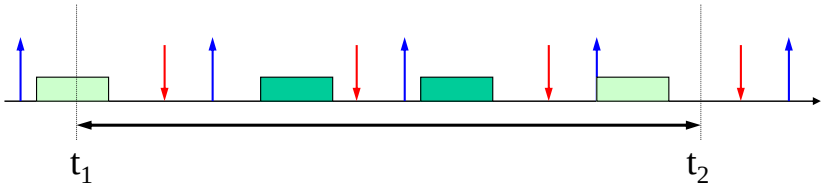
Schedule based on absolute deadlines

Schedulability Analysis

Processor Demand Criterion [Baruah '90]

In any interval, the computation demanded by the task set must be no greater than the available time.

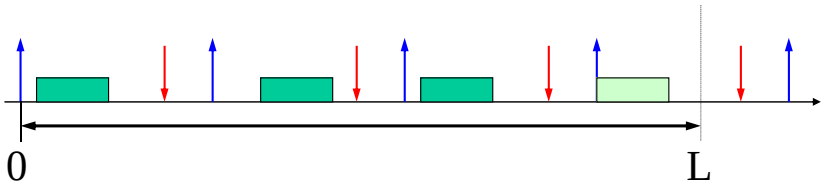
Processor Demand



The demand in $[t_1, t_2]$ is the computation time of those tasks started at or after t_1 with deadline less than or equal to t_2:

$$g(t_1, t_2) = \sum_{\substack{d_i \le t_2 \\ r_i \ge t_1}} C_i$$

Processor Demand



Processor Demand in [0, L]

$$g(0,L) = \sum_{i=1}^{n} \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i$$

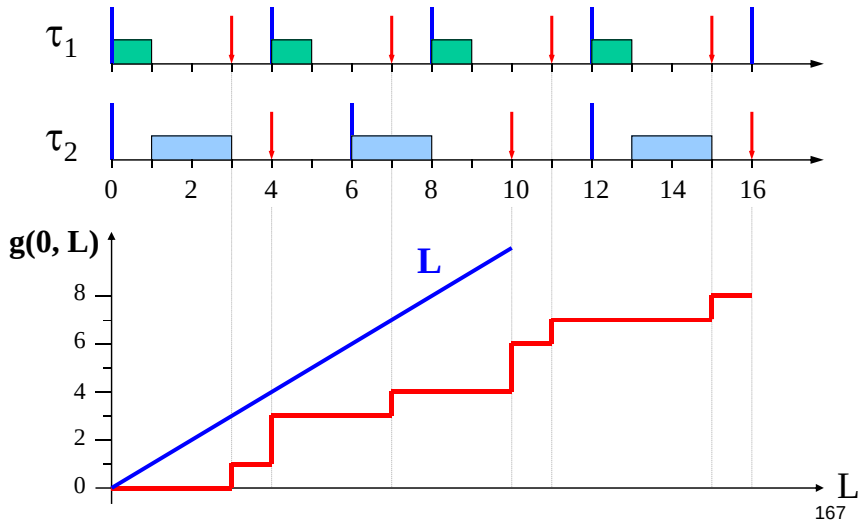
Processor Demand Test

$$\forall L > 0, \quad g(0,L) \leq L$$

Question

How can we bound the number of intervals in which the test has to be performed?

Example



Bounding complexity

- Since g(0,L) is a step function, we can check feasibility only at deadline points.
- If tasks are synchronous and $U_p < 1$, we can check feasiblity up to the hyperperiod H:

$$H = \text{lcm}(T_1, \ldots, T_n)$$

Bounding complexity

- Moreover we note that: $g(0, L) \leq G(0, L)$

$$G(0,L) = \sum_{i=1}^{n} \left(\frac{L + T_i - D_i}{T_i} \right) C_i$$

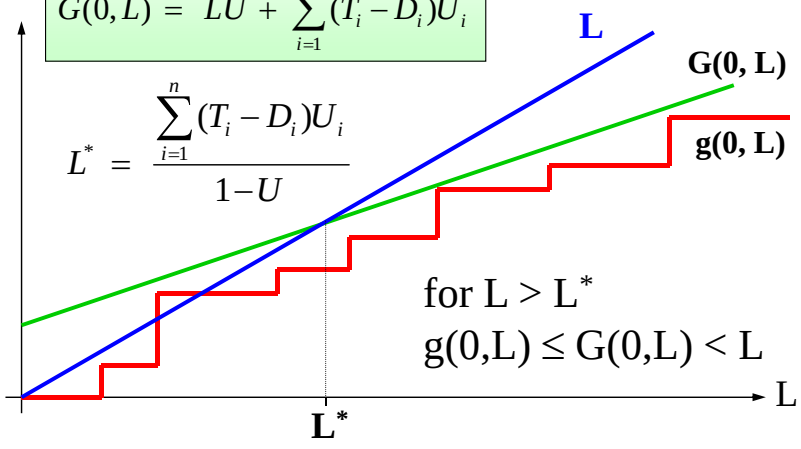
$$= \sum_{i=1}^{n} L \frac{C_i}{T_i} + \sum_{i=1}^{n} (T_i - D_i) \frac{C_i}{T_i}$$

$$= LU + \sum_{i=1}^{n} (T_i - D_i) U_i$$

Limiting L

$$G(0,L) = LU + \sum_{i=1}^{n} (T_i - D_i) U_i$$

$$L^* = \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}$$



Processor Demand Test

$$\forall L \in D, \quad g(0,L) \leq L$$

$$D = \{d_k \mid d_k \leq \min(H, L^*)\}$$

$$\begin{cases} H = lcm(T_1, \ldots, T_n) \\ L^* = \dfrac{\sum_{i=1}^{n} (T_i - D_i)U_i}{1-U} \end{cases}$$

Summary

- **Three scheduling approaches:**
  - Off-line construction (Timeline)
  - Fixed priority (RM, DM)
  - Dynamic priority (EDF)
- **Three analysis techniques:**
  - Processor Utilization Bound $U \leq U_{lub}$
  - Response Time Analysis $\forall i \; R_i \leq D_i$
  - Processor Demand Criterion $\forall L \; g(0,L) \leq L$

Complexity Issues

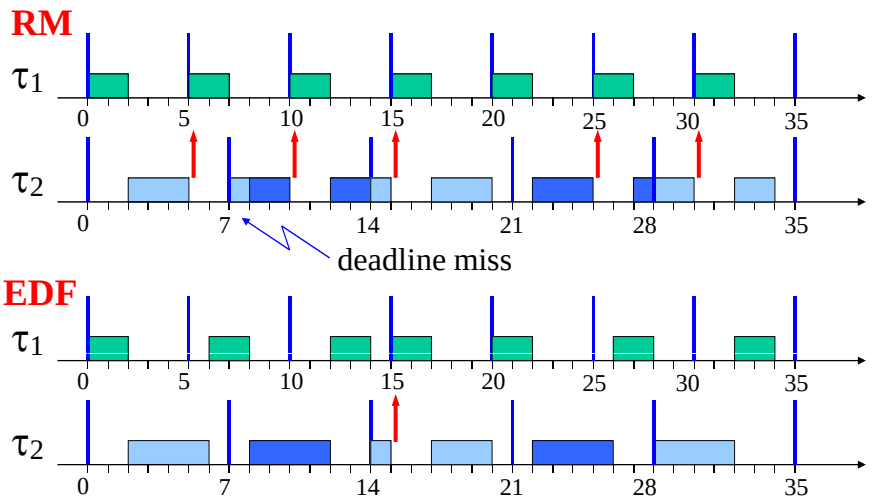
- **Utilization based analysis** $(U \leq U_{lub})$
  - O(*n*) complexity
- **Response time analysis** $(\forall i \ R_i \leq D_i)$
  - Pseudo-polynomial complexity
- **Processor demand analysis** $(\forall L \ g(0,L) \leq L)$
  - Pseudo-polynomial complexity

RM vs. EDF

Metrics

- Implementation complexity
- Efficiency
- Schedulability analysis
- Runtime overhead
- Overload conditions
- Jitter
- Aperiodic task handling

Context switches



Schedulability Analysis

	$D_i = T_i$	$D_i \leq T_i$
RM	***Suff.:*** ***polynomial*** $O(n)$ LL: $\Sigma U_i \leq n(2^{1/n}-1)$ HB: $\Pi(U_i+1) \leq 2$ ***Exact*** ***pseudo-polynomial*** RTA	***pseudo-polynomial*** Response Time Analysis $\forall i \quad R_i \leq D_i$ $R_i = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$
EDF	***polynomial:*** $O(n)$ $\Sigma U_i \leq 1$	***pseudo-polynomial*** Processor Demand Analysis $\forall L > 0, \quad g(0,L) \leq L$

Questions

- If EDF is more efficient than RM, why commercial RT systems are still based on RM?

Main reason

- RM is simpler to implement on top of commercial (fixed priority) kernels.
- EDF requires explicit kernel support for deadline scheduling, but gives other advantages.

RM: harmonic periods

Harmonic task sets are schedulable by RM

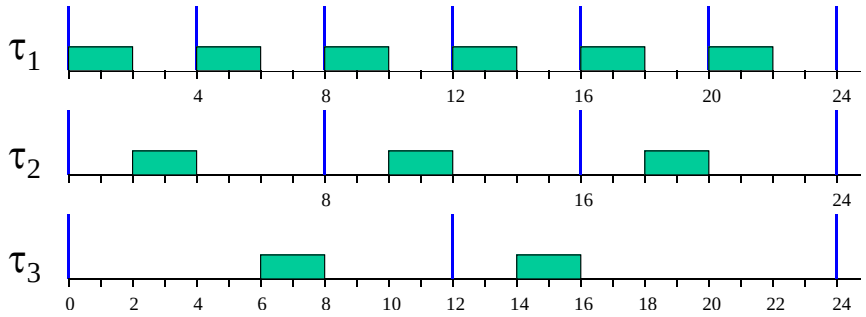
if and only if $U \leq 1$.

A set of tasks is harmonic if **every** pair of periods are in harmonic relation.

A common misconception

The RM schedulability bound is 1 if every period is multiple of the shortest period.

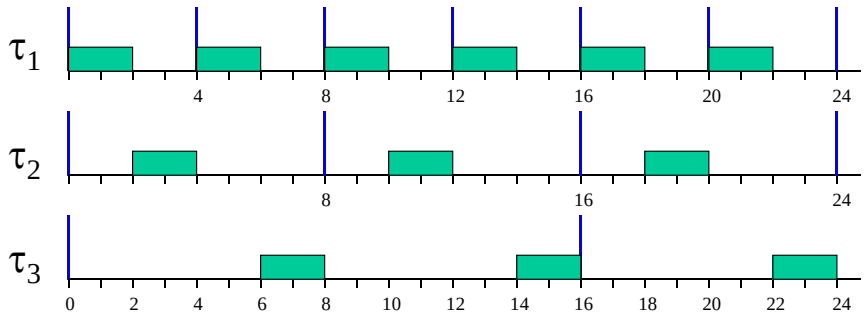
Non harmonic periods



$$U = \frac{2}{4} + \frac{2}{8} + \frac{2}{12} \cong 0.917$$

Any increase in the C_i's makes the system unschedulable

Harmonic task set



$$U = \frac{2}{4} + \frac{2}{8} + \frac{4}{16} = 1$$

Robustness under overloads

Two situations are considered:

1. Permanent overload

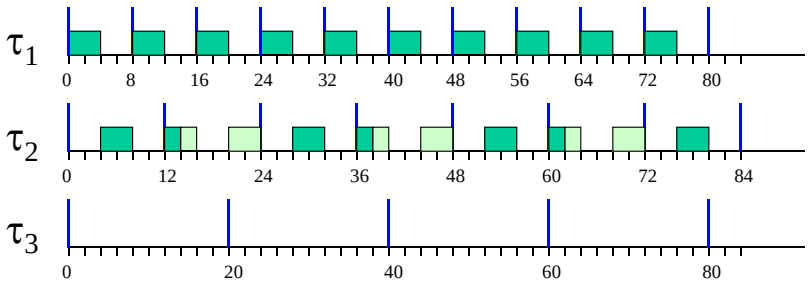
⇒ This occurs when U > 1

2. Transient overload

⇒ This occurs when some job executes more than expected

RM under permanent overload

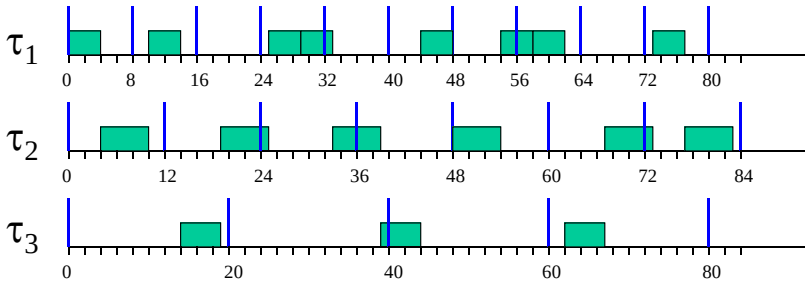
$$U = \frac{4}{8} + \frac{6}{12} + \frac{5}{20} = 1.25$$



- High priority tasks execute at the proper rate
- Low priority tasks are completely blocked

EDF under permanent overload

$$U = \frac{4}{8} + \frac{6}{12} + \frac{5}{20} = 1.25$$

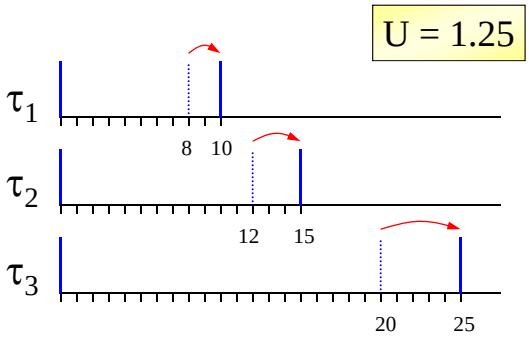


- All tasks execute at a slower rate
- No task is blocked

EDF is predictable in overloads

Theorem (Cervin '03)

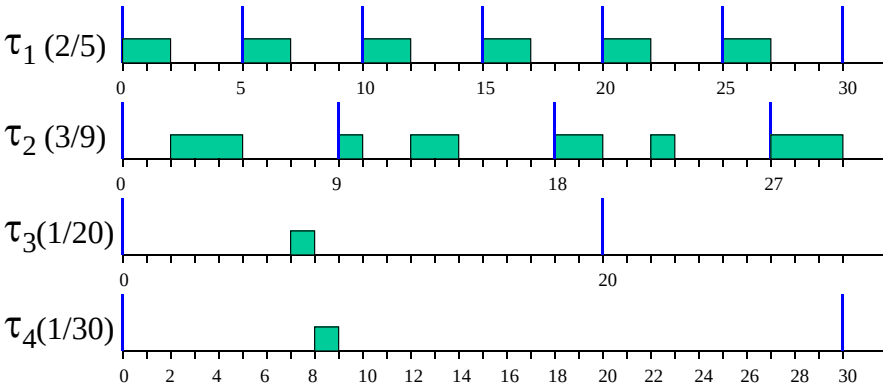
If U > 1, EDF executes tasks with an average period $T'_i = T_i U$.



	T_i	T'_i
τ_1	8	10
τ_2	12	15
τ_3	20	25

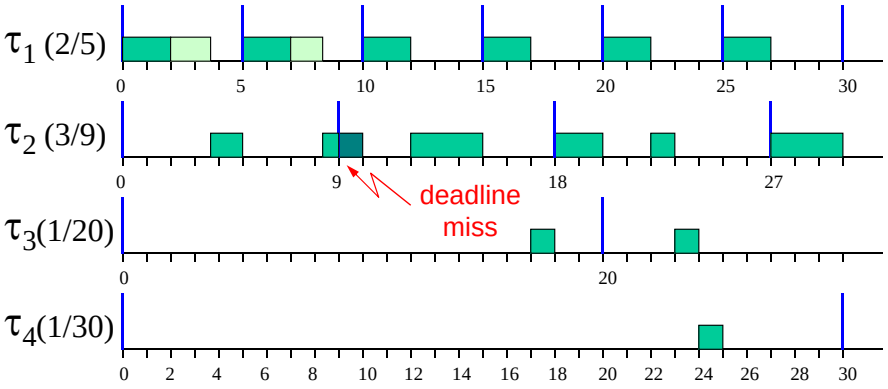
RM during transient overruns

$U_{avg} = 0.817$ $C_{1avg} = 2$, $C_{1max} = 4$



RM during transient overruns

$U_{avg} = 0.817$ $C_{1avg} = 2$, $C_{1max} = 4$



Who is missing its deadline is not the lowest priority task

Advantages of EDF

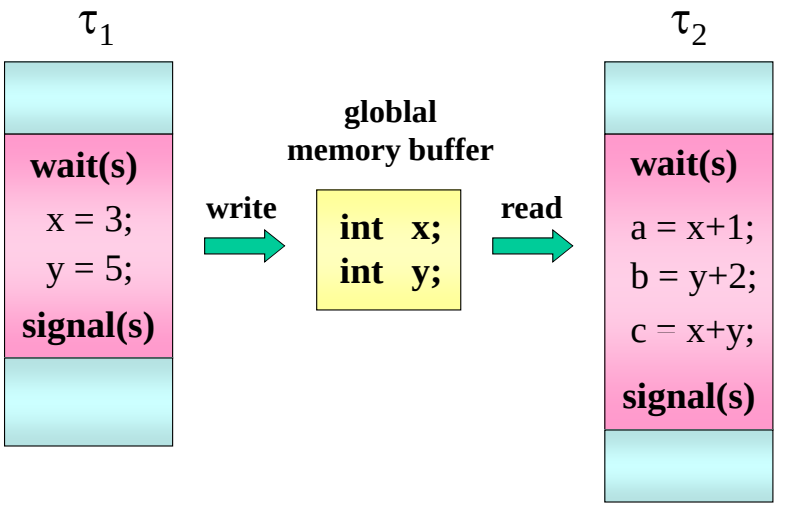
However, EDF offers the following advantages with respect to RM:

- Better processor utilization (100%)
- Less overhead due to preemptions;
- More flexible behavior in overload situations;
- More uniform jitter control;
- Better aperiodic responsiveness.

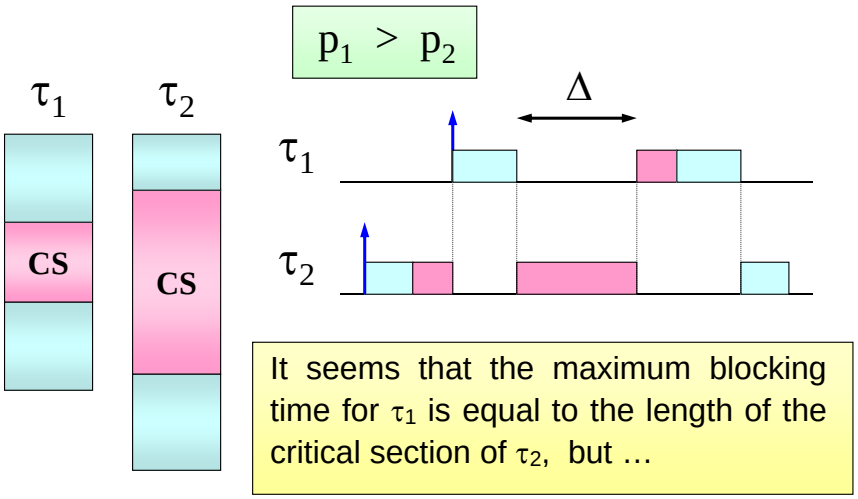
Handling shared resources

Problems caused by mutual exclusion

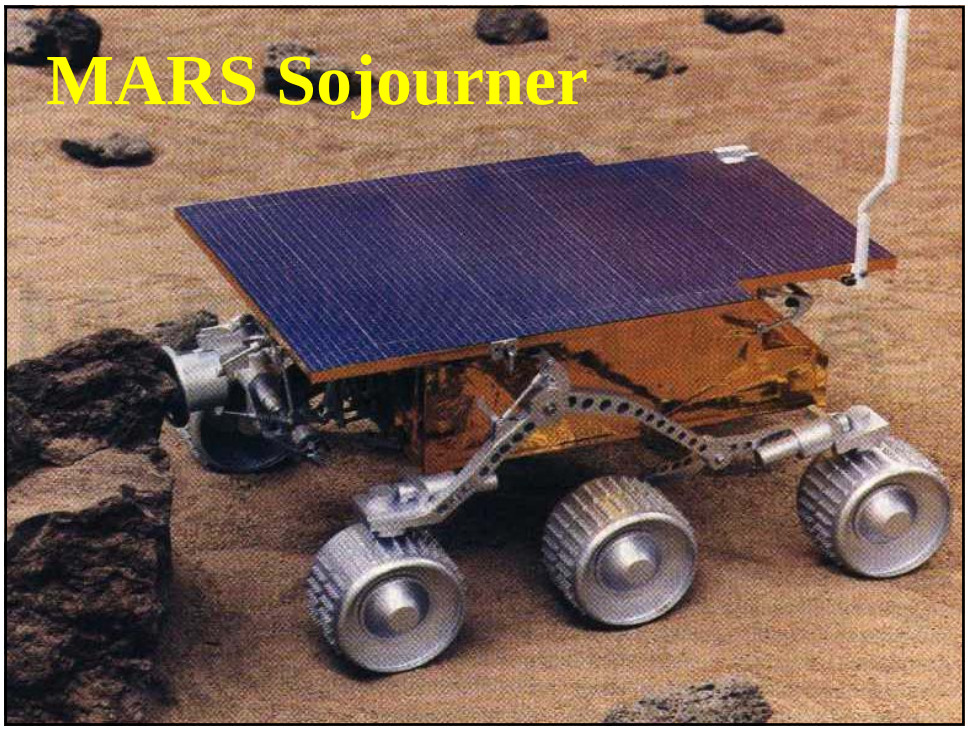
Critical sections



Blocking on a semaphore



It seems that the maximum blocking time for τ_1 is equal to the length of the critical section of τ_2, but …

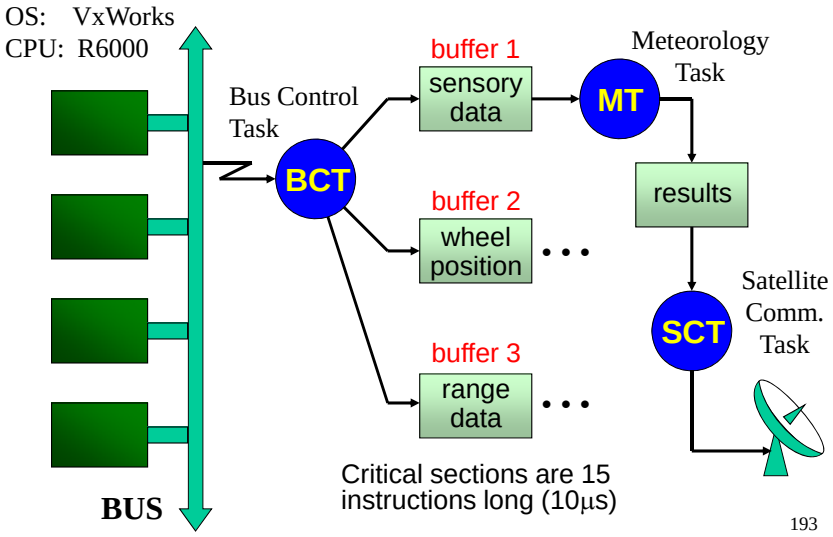


Mars Mission

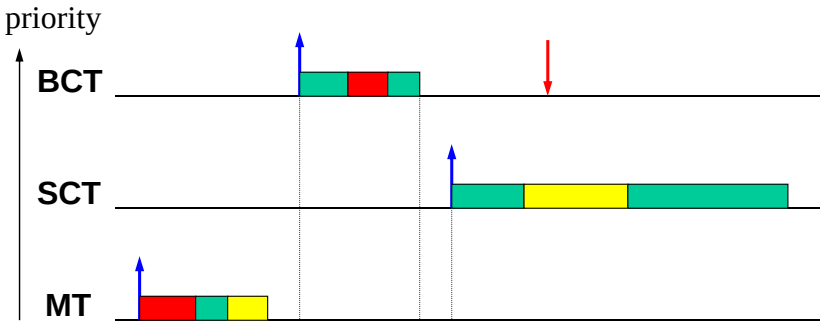
Pathfinder + Sojourner

July 4th, '97	landing on Mars (after 6 months)
July 15th, '97	the system self-resets for timeout: the robot stops working
July 16th, '97	News spread all over the world: engineers start working
July 17th, '97	the error is identified: probability = 10^{-6}

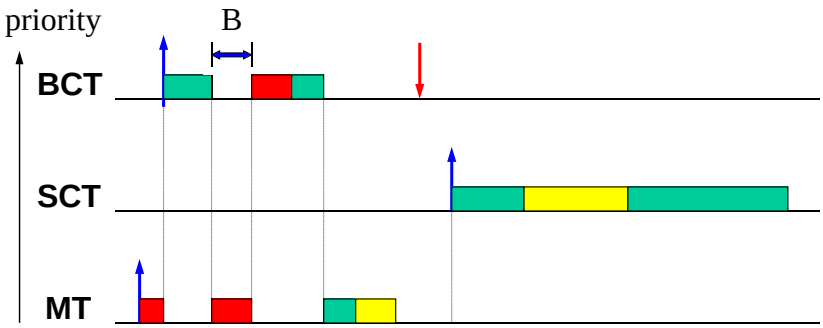
Application structure



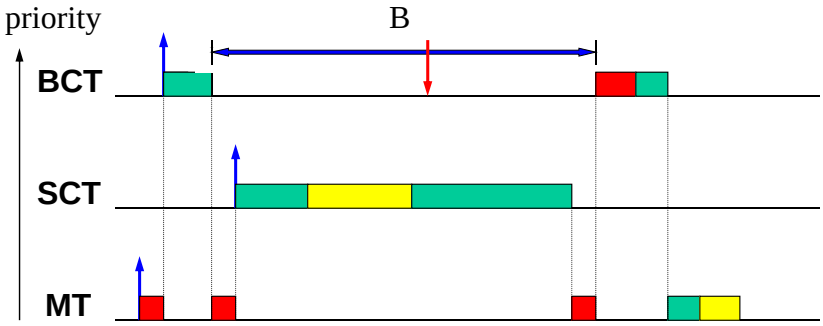
Schedule with no conflicts



Conflict on a critical section



Conflict on a critical section



Priority Inversion

A high priority task is blocked by a lower- priority task a for an unbounded interval of time.

Solution

Introduce a concurrency control protocol for accessing critical sections.

Resource Access Protocols

- Non Preemptive Protocol (NPP)
- Highest Locker Priority (HLP)
- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Stack Resource Policy (SRP)

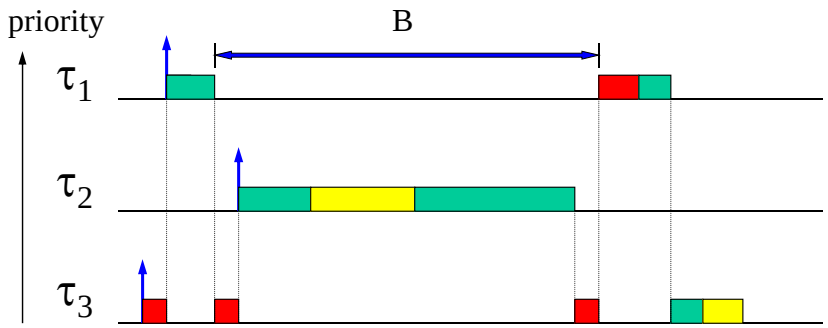
Non Preemptive Protocol

- Preemption is forbidden in critical sections.
- Implementation: when a task enters a CS, its priority is increased at the maximum value.

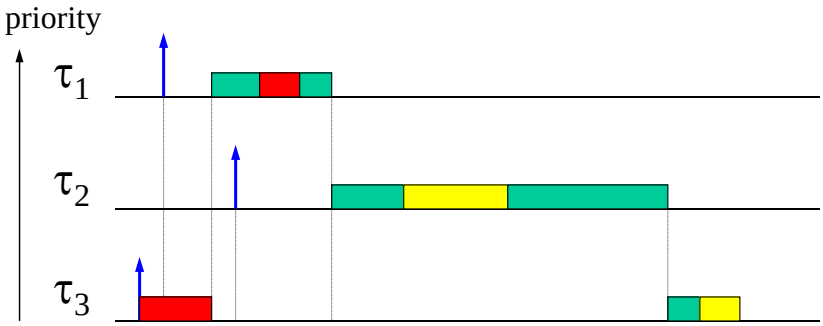
ADVANTAGES: simplicity

PROBLEMS: high priority tasks that do not use the CS may also block

Conflict on critical section

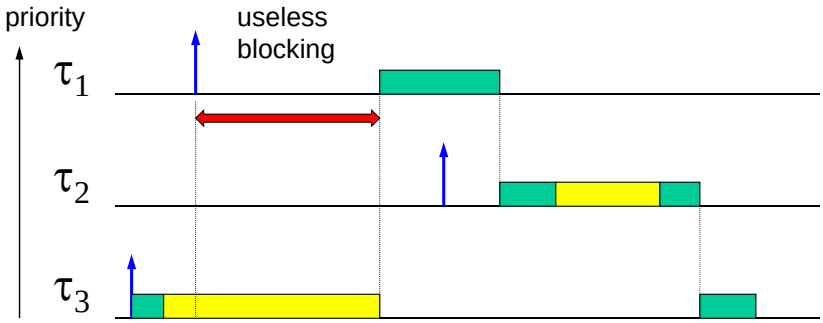


Schedule with NPP



$P_i(R) = \max\{P_1, \ldots P_n\}$

Problem with NPP



τ_1 cannot preemt, although it could

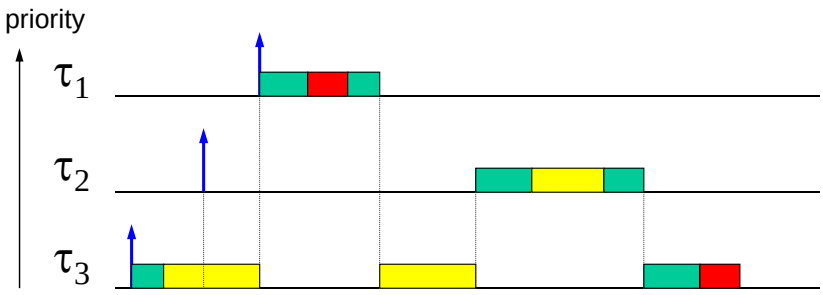
Highest Locker Priority

A task in a CS gets the highest priority among the tasks that use it.

FEATURES:

- Simple implementation.
- A task is blocked when attempting to preempt, not when entering the CS.

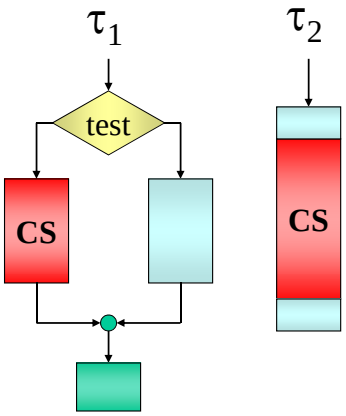
Schedule with HLP



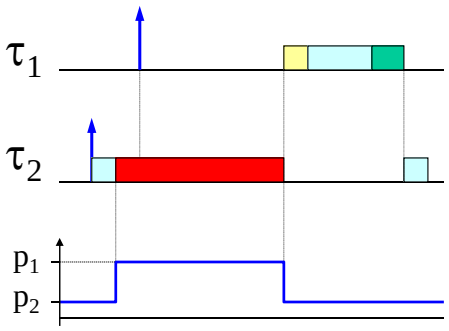
$P_i(R) = \max \{P_k \mid \tau_k \text{ uses } R\}$

τ_2 is blocked, but τ_1 can preempt within a CS

Problem with HLP



τ_1 blocks just in case ...



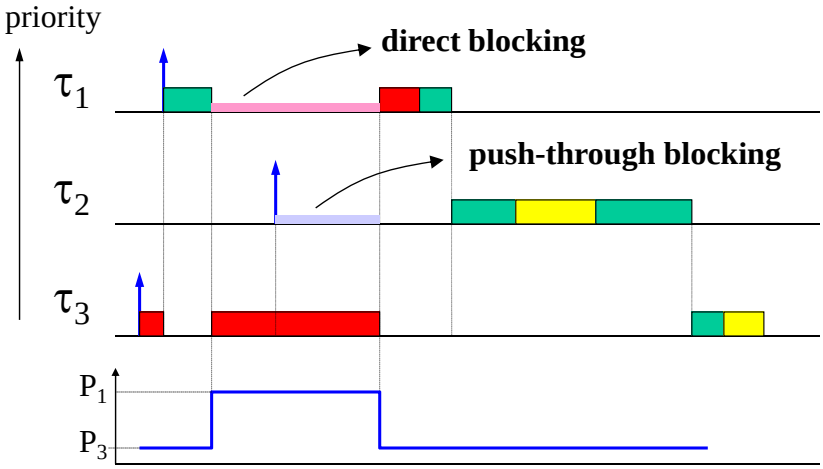
Priority Inheritance Protocol

[Sha, Rajkumar, Lehoczky, 90]

- A task in a CS increases its priority only if it blocks other tasks.
- A task in a CS inherits the highest priority among those tasks it blocks.

$$P_i(R) = \max \{P_k \mid \tau_k \text{ blocked on } R\}$$

Schedule with PIP



Types of blocking

- **Direct blocking**

  A task blocks on a locked semaphore

- **Push-through blocking**

  A task blocks because a lower priority task inherited a higher priority.

BLOCKING:
a delay caused by a lower priority task

Identifying blocking resources

A task τ_i can be blocked by a semaphore used by lower priority tasks only if it is

- directly shared with τ_i (direct blocking) or
- shared with tasks having priority higher than τ_i (push-through blocking).

Identifying blocking resources

Lemma 1: A task τ_i can be blocked at most once by a lower priority task.

If there are n_i tasks with priority lower than τ_i, then τ_i can be blocked at most at most n_i times, independently of the number of critical sections it uses.

Identifying blocking resources

Lemma 2: A task τ_i can be blocked at most once on a semaphore S_k.

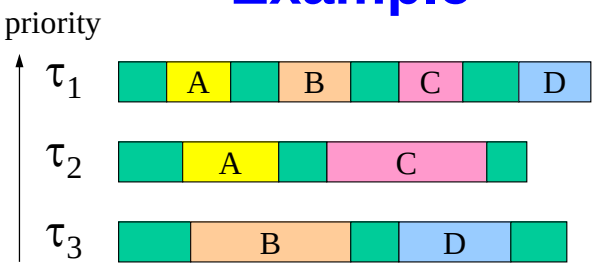
If there are m_i distinct semaphores that can block a task τ_i, then τ_i can be blocked at most m_i times, independently of the number of critical sections it uses.

Bounding blocking times

- If n_i is the number of tasks with priority less than τ_i
- and m_i is the number of semaphores on which τ_i can be blocked, **then**

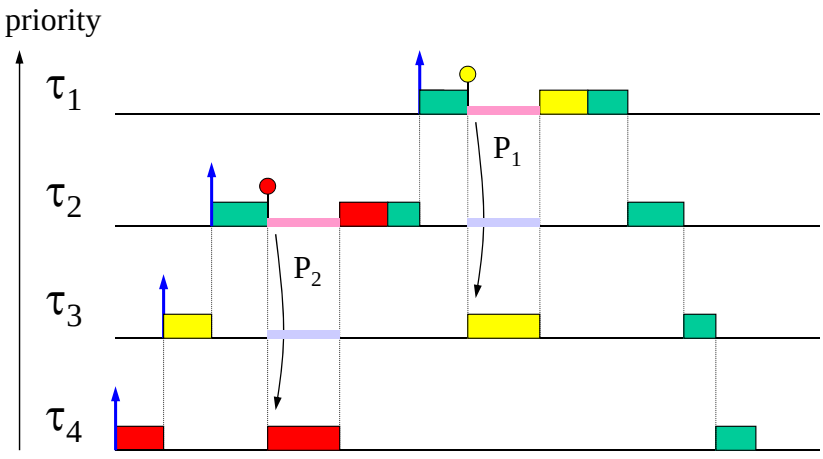
Theorem: τ_i can be blocked at most for the duration of $\alpha_i = \min(n_i, m_i)$ critical sections.

Example



- τ_1 can be blocked once by τ_2 (on A_2 or C_2) and once by τ_3 (on B_3 or D_3)
- τ_2 can be blocked once by τ_3 (on B_3 or D_3)
- τ_3 cannot be blocked

Schedule with PIP



Remarks on PIP

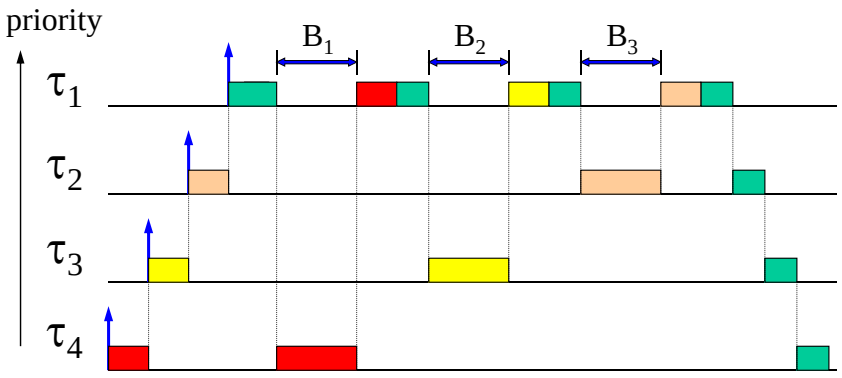
ADVANTAGES

- It is transparent to the programmer.
- It bounds priority inversion.

PROBLEMS

- It does not avoid deadlocks and chained blocking.

Chained blocking with PIP



Theorem: τ_i can be blocked at most once by each lower priority task.

Priority Ceiling Protocol

- Can be viewed as PIP + access test.
- A task can enter a CS only if it is free and there is no risk of chained blocking.

To prevent chained blocking, a task may stop at the entrance of a free CS (***ceiling blocking***).

Resource Ceilings

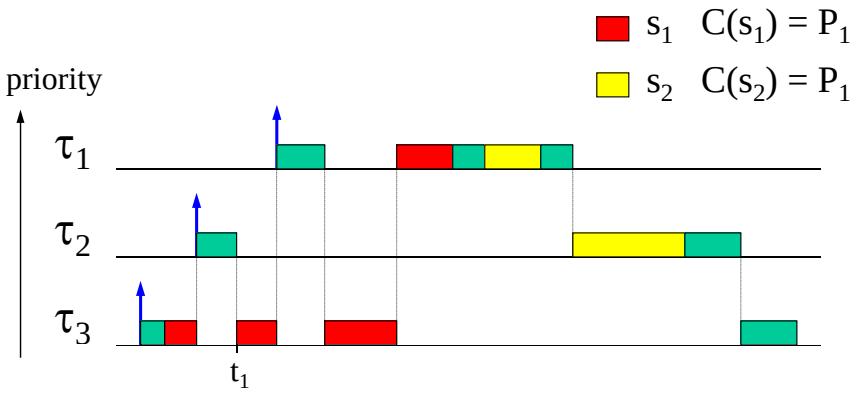
- Each semaphore s_k is assigned a ceiling:

$$C(s_k) = \max \{P_j : \tau_j \text{ uses } s_k\}$$

- A task τ_i can enter a CS only if

$$P_i > \max \{C(s_k) : s_k \text{ locked by tasks} \neq \tau_i\}$$

Schedule with PCP



t_1: τ_2 is blocked by the PCP, since $P_2 < C(s_1)$

PCP properties

Theorem 1

Under PCP, each task can block at most once.

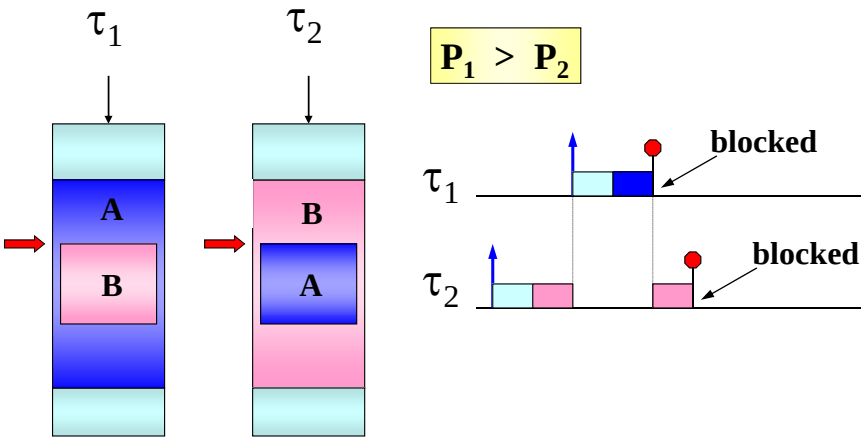
Theorem 2

PCP prevents chained blocking.

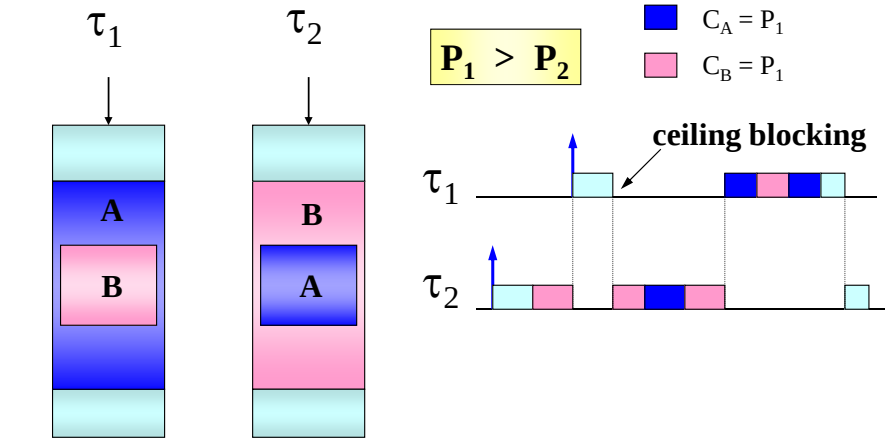
Theorem 3

PCP prevents deadlocks.

Typical Deadlock



Deadlock avoidance with PCP



PCP problems

1. It is not transparent to the programmer: semaphores need ceilings
2. It is pessimistic (like HLP): it can create unnecessary blocking

Questions

1. If a task uses several critical sections, can it be blocked on the second one?
2. Each task has a nominal priority (P_i) and a dynamic priority ($p_i \geq P_i$) changed by the protocol to prevent priority inversion. If τ_i blocks because

   $$P_i \leq \max \{C(s_k) : s_k \text{ locked by tasks} \neq \tau_i\}$$

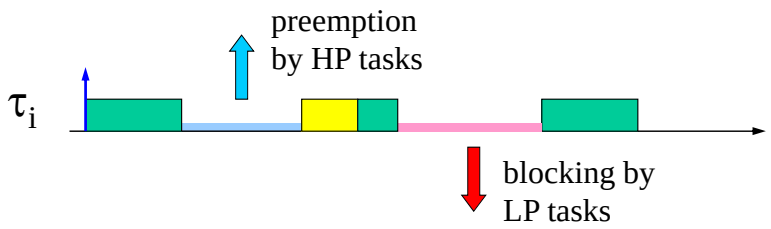
   can it be that

   $$p_i > \max \{C(s_k) : s_k \text{ locked by tasks} \neq \tau_i\} ?$$

Guarantee with resource constraints

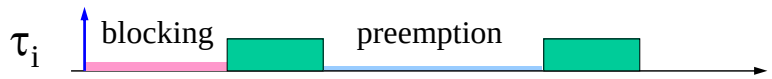
- We select a scheduling algorithm and a resource access protocol.
- We compute the maximum blocking times (B_i) for each task.
- We perform the guarantee test including the blocking terms.

Guarantee with RM



$$\forall i \quad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i\left(2^{1/i} - 1\right)$$

Response Time Analysis



$$\begin{cases} R_i^0 = B_i + C_i \\ R_i^s = B_i + C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^{(s-1)}}{T_k} \right\rceil C_k \end{cases}$$

iterate while $R_i^s > R_i^{(s-1)}$

Resource Sharing under EDF

The protocols analyzed so far have been originally developed for fixed priority scheduling schemes. However:

- NPP can also be used under EDF
- PIP has been extended under EDF by Spuri (1997).
- PCP has been extended under EDF by Chen-Lin (1990)
- In 1990, Baker proposed a new protocol that works both under fixed and dynamic priorities.

Stack Resource Policy [Baker 1990]

This protocol satisfies the same PCP properties:

- it avoids unbounded priority inversion;
- it prevents deadlocks and chained blocking;
- each task can be blocked at most once;

In addition:

- SRP allows using multi-unit resources;
- it can be used under fixed and dynamic priorities;
- it allows tasks to share the same stack space.

Stack Resource Policy

For each task τ_i, we have to specify

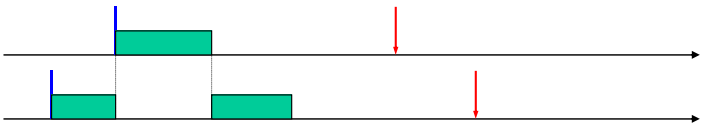
- A priority p_i (static or dynamic):
  - RM: $p_i \propto 1/T_i$
  - DM: $p_i \propto 1/D_i$
  - EDF: $p_i \propto 1/d_i$
- A preemption level: $\pi_i \propto 1/D_i$
- A set of resource requirements: $\forall R_k \quad \mu_i(R_k)$

  $\mu_i(R_k)$ specifies how many units of R_k are used by τ_i

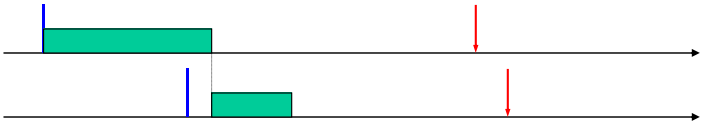
Preemption Level

$\pi_i \propto 1/D_i$

- Note that, under EDF, a task τ_i can preempt another task τ_k only if $\pi_i > \pi_k$ (that is, if $D_i \geq D_k$)



- No preemption can occur if $\pi_i \leq \pi_k$ (that is, if $D_i \geq D_k$):



Stack Resource Policy

Each resource R_k

- specifies the total number of units available: $N(R_k)$
- keeps track of the current available units: $n(R_k)$
- is assigned a dynamic ceiling: $C_R(n_k)$

$$C_R(n_k) = \max\{0, \pi_i : n(R_k) < \mu_i(R_k)\}$$

NOTE: $C_R(n_k)$ increases only when a resource is locked
$C_R(n_k)$ decreases only when a resource is released

Ceiling Property

Lemma

If $\pi_i > C_R(n_k)$ then there exist enough units of R

1. to satisfy the requirements of τ_i
2. to satisfy the requirements of all tasks that can make preemption on τ_i

Stack Resource Policy

Finally, a system ceiling is defined as:

$$\Pi_s = \max\{C_R(n_k)\}$$

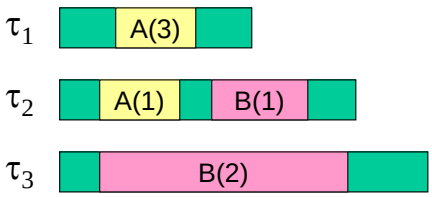
SRP preemption rule

A ready task τ_i can preempt the executing task τ_{exe} if and only if

$$p_i > p_{exe} \quad \text{and} \quad \pi_i > \Pi_s$$

Computing Resource Ceilings

	N_R
R_A	3
R_B	2



	D_i	π_i	μ_A	μ_B
τ_1	10	3	3	0
τ_2	15	2	1	1
τ_3	20	1	0	2

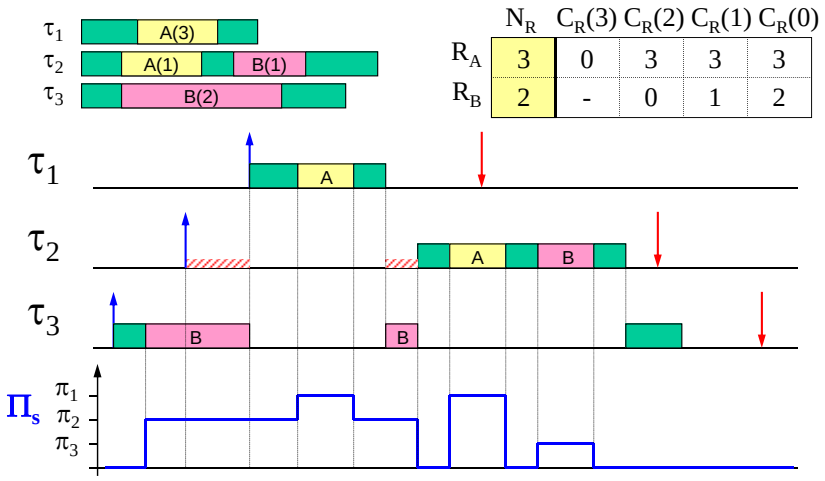
Computing Resource Ceilings

	N_R
R_A	3
R_B	2

	D_i	π_i	μ_A	μ_B
τ_1	10	3	3	0
τ_2	15	2	1	1
τ_3	20	1	0	2

	$C_R(3)$	$C_R(2)$	$C_R(1)$	$C_R(0)$
R_A	0	3	3	3
R_B	-	0	1	2

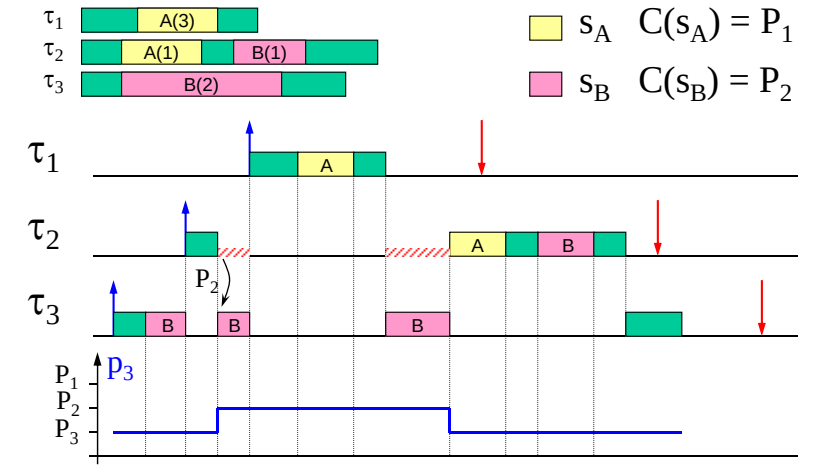
Schedule with SRP



	N_R	$C_R(3)$	$C_R(2)$	$C_R(1)$	$C_R(0)$
R_A	3	0	3	3	3
R_B	2	-	0	1	2

A task blocks when attempting to preempt

Schedule with PCP



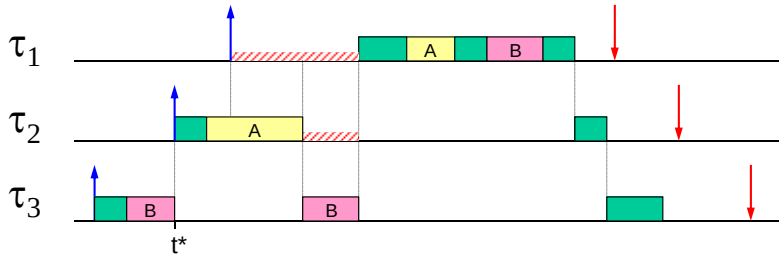
A task is blocked when accessing a resource

SRP Properties

Theorem 1

Under SRP, each task can block at most once.

Consider the following scenario where τ_1 blocks twice:



This is not possible, because τ_2 could not preempt τ_3 because, at time t*, $\pi_2 < \Pi_s$

SRP Properties

Theorem 2

If $\pi_i > \Pi_s$ then τ_i will never block once started.

Proof

Since $\Pi_s = \max\{C_R(n_k)\}$, then there are enough resources to satisfy the requirements of τ_i and those of all tasks that can preempt τ_i .

Question

If a task can never block once started, can we get rid of the wait / signal primitives?

SRP Properties

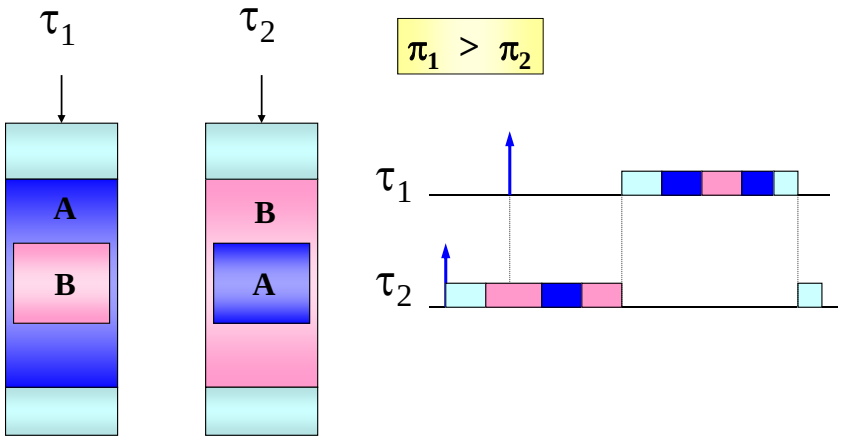
Theorem 3

SRP prevents deadlocks.

Proof

From Theorem 2, if a task can never block once started, then no deadlock can occur.

Deadlock avoidance with SRP



Schedulability Analysis under EDF

When $D_i = T_i$

$$\forall i \qquad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq 1$$

B_i can be computed as under PCP and refers to the length of longest critical section that can block τ_i.

Schedulability Analysis under EDF

When $D_i \leq T_i$

A task set is schedulable if $U < 1$ and $\forall L \in D$

$$\forall i \qquad B_i + \sum_{k=1}^{n} \left\lfloor \frac{L + T_k - D_k}{T_k} \right\rfloor C_k \leq L$$

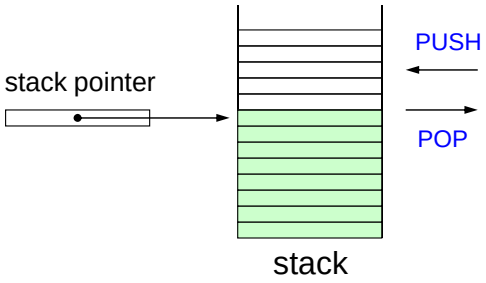
where $D = \{d_k \mid d_k \leq \min(H, L^*)\}$

$$H = lcm(T_1, \ldots, T_n) \qquad L^* = \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}$$

Stack Sharing

Each task normally uses a private stack for

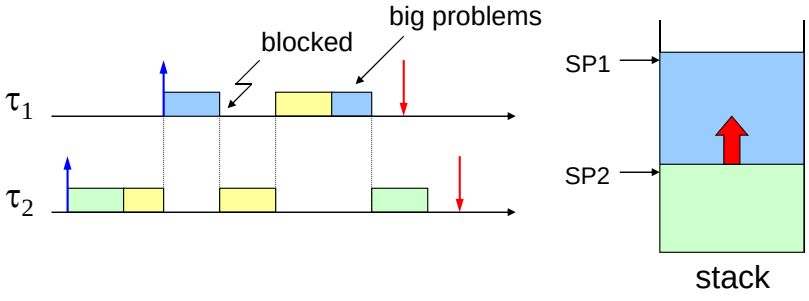
- saving context (register values)
- managing functions
- storing local variables



Stack Sharing

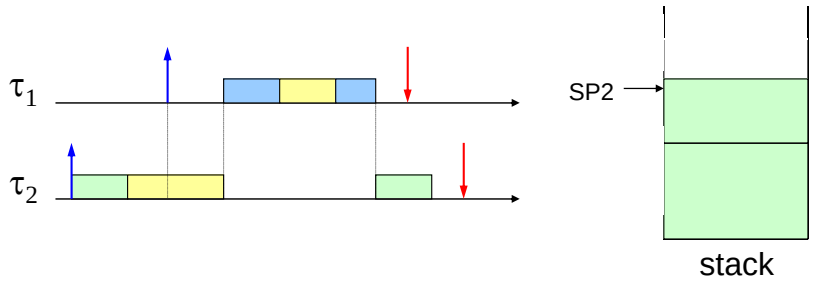
Why stack cannot be normally shared?

Suppose tasks share a resource: A



Stack Sharing

Why stack can be shared under SRP?



Saving Stack Size

To really save stack size, we should use a small number of preemption levels.

100 tasks
10 Kb stack per task
→ stack size = 1 Mb

10 preemption levels
10 tasks per group
→ stack size = 100 Kb

stack saving = 90 %

Non preemptive scheduling

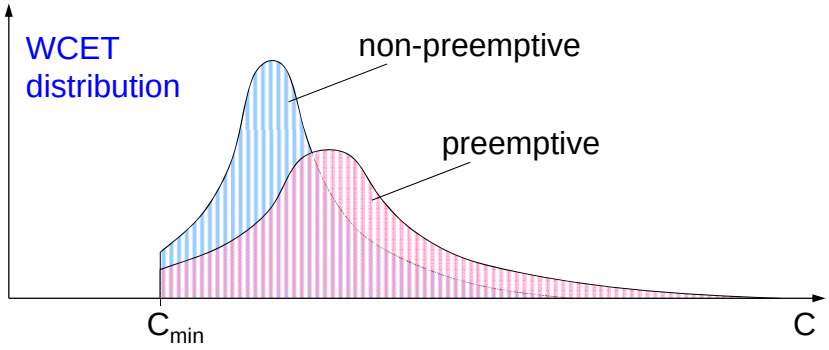
Problem Statement

Preemption has a cost, often not negleagible:

- Scheduler cost: due to the time taken by the scheduler to suspend the running task, switch the context, and dispatch the new incoming task.
- Pipeline cost: due to the time taken to flush the pipeline when the task is interrupted and the time taken to refill the pipeline when the task is resumed.
- Cache-related cost: due to the time taken to reload the cache lines evicted by the preempting task.

Influence on WCETs

- As a consequence, WCETs estimations for preemptive tasks are
  - higher
  - less predictable (highly variable)



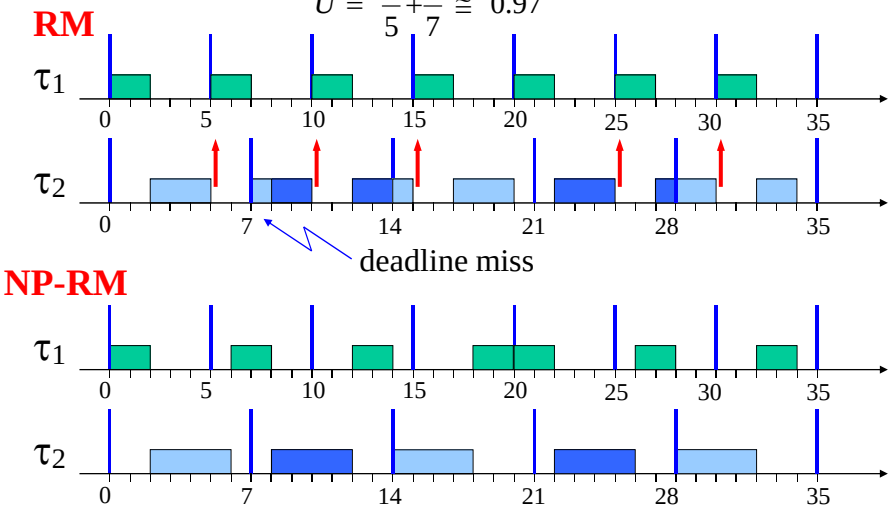
Advantages of NP scheduling

- It reduces context-switch overhead:
  - making WCETs smaller and more predictable.
- It simplifies the access to shared resources:
  - No semaphores are needed for critical sections
- It reduces stack size:
  - Task can share the same stack, since no more than one task can be in execution
- It allows achieving zero I/O Jitter:
  - finishing_time – start_time = C_i (constant)

Advantages of NP scheduling

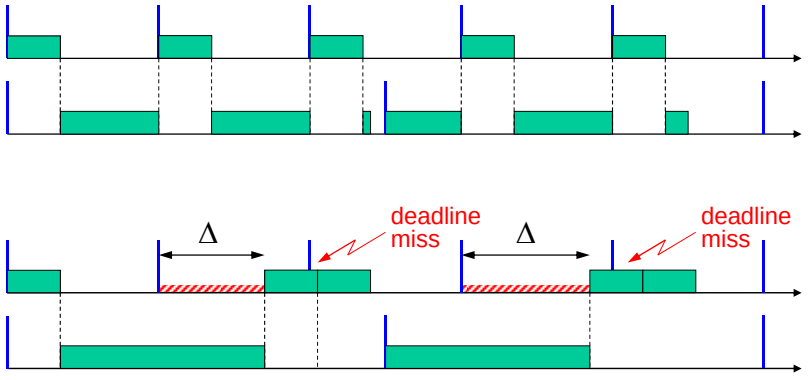
In fixed priority systems can improve schedulabiilty:

$$U = \frac{2}{5} + \frac{4}{7} \cong 0.97$$



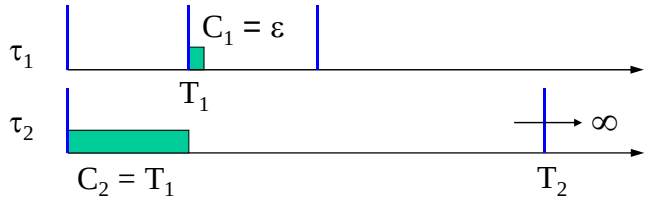
Disdvantages of NP scheduling

- In general, NP scheduling reduces schedulability introducing blocking delays in high priority tasks:



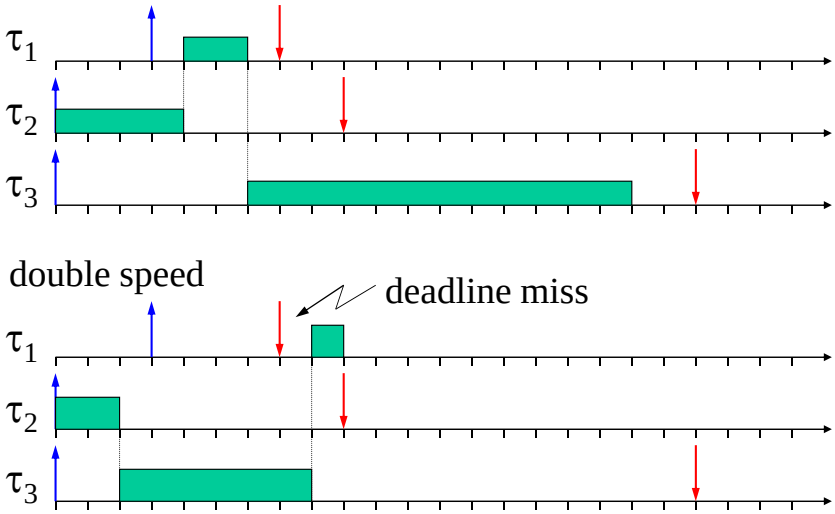
Disdvantages of NP scheduling

- The utilization bound under non preemptive scheduling drops to zero:



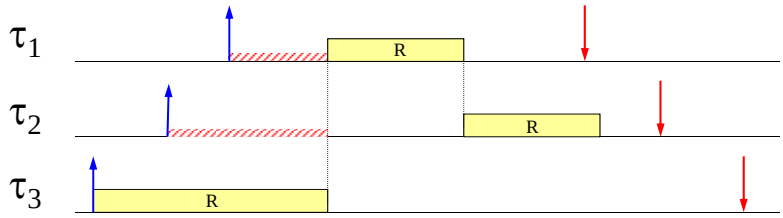
$$U = \frac{\varepsilon}{T_1} + \frac{C_2}{\infty} \rightarrow 0$$

Non preemptive scheduling anomalies



Non-preemtive analysis

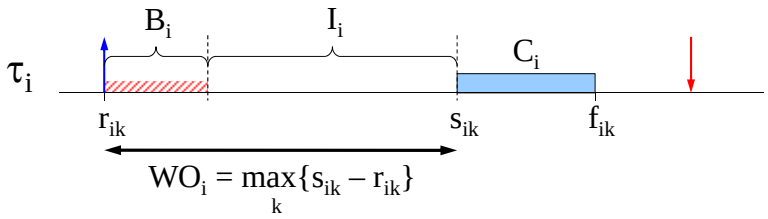
It is a special case of preemptive scheduling where all tasks share a single resource for their entire duration.



The max blocking time for task τ_i is given by the largest C_k among the lowest priority tasks:

$$B_i = \max\{C_j : P_j < P_i\}$$

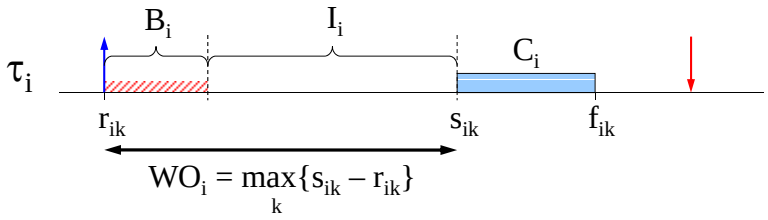
Response time analysis



WO_i = Worst-case occupied time: due to blocking from lp(i) and interference from hp(i)

NOTE: the end of WO_i cannot coincide with the activation of a higher priority task τ_h, which would increase WO_i by C_h

Response time analysis



$$WO_i = B_i + \sum_{k=1}^{i-1} \left(\left\lceil \frac{WO_i}{T_k} \right\rceil + 1 \right) C_k$$

$$R_i = WO_i + C_i$$

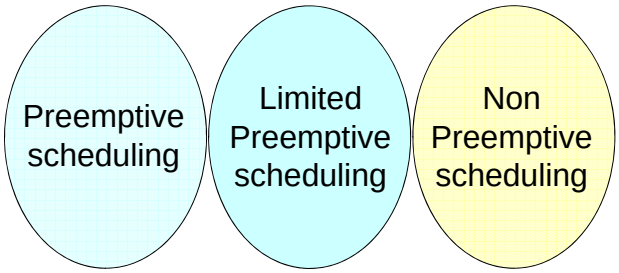
Response time analysis

$$\begin{cases} WO_i^{(0)} = B_i + \sum_{k=1}^{i} C_k \\ WO_i^{(s)} = B_i + \sum_{k=1}^{i-1} \left(\left\lceil \frac{WO_i^{(s-1)}}{T_k} \right\rceil + 1 \right) C_k \end{cases}$$

Stop when $WO_i^{(s)} = WO_i^{(s-1)}$

$$R_i = WO_i + C_i$$

Taking advantage of NP scheduling



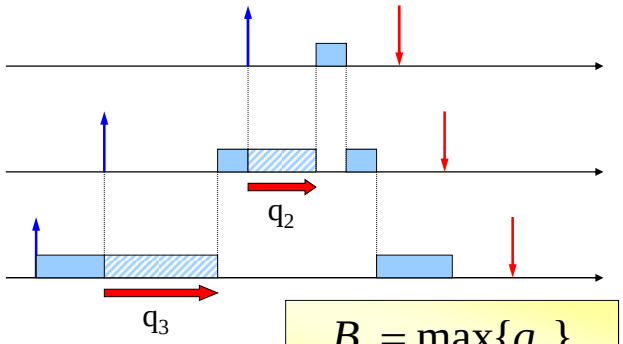
Trade-off solutions

- Deferred preemptions
- Fixed preemption points
- Preemption thresholds

Deferred Preemption

Each task can defer preemption up to q_i

NP regions are floating in the code (i.e., NP regions can start at any time)



$$B_i = \max_{P_j < P_i} \{q_j\}$$

Interesting problem

Given a preemptively feasible task set, reduce preemptions as much as possible still preserving schedulability.

⇒ Reducing context switch costs and WCETs

This means finding the longest non-preemptive chunk $\mathbf{Q_i}$ for each task that can still preserve schedulability.

- Under EDF ⇒ Baruah - ECRTS 2005
- Under Fix. Pr. ⇒ Yao et. al. - RTCSA 2009

A simple bound for Q_i

Q_i is related with the maximum blocking time that can be tolerated by higher priority tasks.

Let β_i be the maximum blocking time that can be tolerated by τ_i, called blocking tolerance.

Then, it must be $B_i \leq \beta_i$

where $B_i = \max_{P_j < P_i} \{q_j\}$

Hence: $\max_{P_j < P_i} \{q_j\} \leq \beta_i$

A simple bound for Q_i

$$\forall i \quad \max_{P_j < P_i}\{q_j\} \le \beta_i$$

$$\begin{cases} i = 1 & max\{q_2, q_3, q_4\} \le \beta_1 \\ i = 2 & max\{q_3, q_4\} \le \beta_2 \\ i = 3 & q_4 \le \beta_3 \end{cases}$$

$$\begin{cases} i = 1 & q_2 \le \beta_1 \\ i = 2 & q_3 \le min\{\beta_1, \beta_2\} \\ i = 3 & q_4 \le min\{\beta_1, \beta_2, \beta_3\} \end{cases}$$

A simple bound for Q_i

$$\begin{cases} i = 1 & Q_1 = \infty \\ i \ge 2 & Q_i = min\{Q_{i-1}, \beta_{i-1}\} \end{cases}$$

Deriving β_i from the utilization test

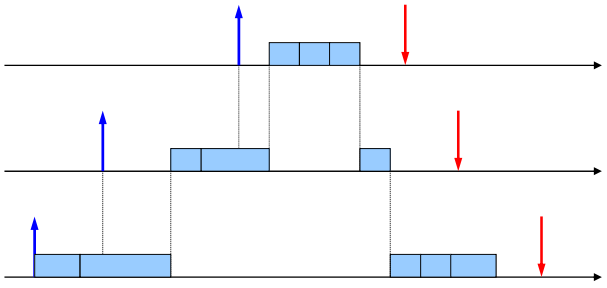
$$\left(\sum_{k=1}^{i}\frac{C_k}{T_k}\right)+\frac{B_i}{T_i}\le U_{\text{lub}}(i)$$

$$\left(\sum_{k=1}^{i}U_k\right)+\frac{\beta_i}{T_i}=U_{\text{lub}}(i)$$

$$\beta_i = T_i\left[U_{\text{lub}}(i)-\sum_{k=1}^{i}U_k\right]$$

Fixed Preemption Points (FPP)

- Each task τ_i is divided in m_i chunks: $q_{i,1}$ ... q_{i,m_i}
- It can only be preempted between chunks



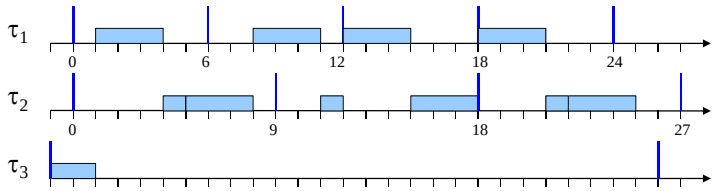
$$B_i = \max_{P_j<P_i}\{q_j^{\max}\}$$

Example

Let: τ_1 be fully non preemptive: $q_{11} = C_1 = 3$

τ_2 consisting of 2 NP chunks: $q_{21} = 1$, $q_{22} = 3$, $C_2 = 4$

τ_3 be fully non preemptive: $q_{31} = C_3 = 2$

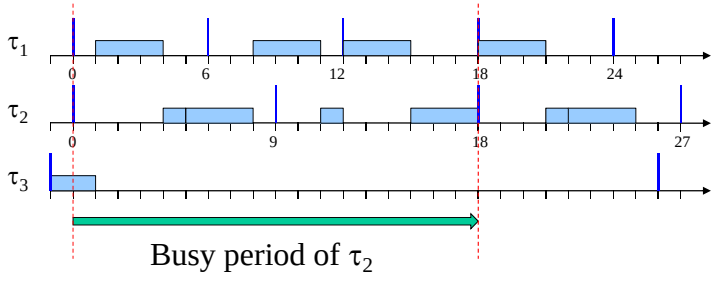


Note that:

- The worst case response time of τ_2 does not occur in the first instance.
- The interference on τ_2 is larger than $B_2 + C_1$.

Response Time Analysis (FPP)

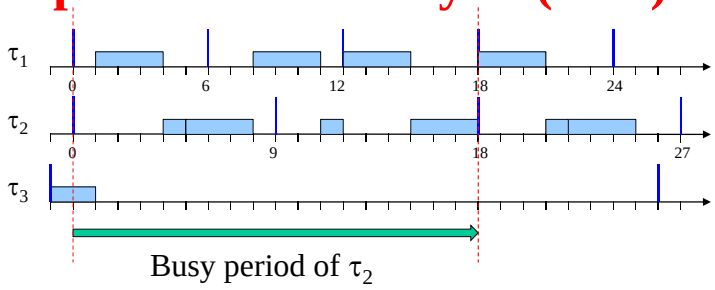
- Must be carry out up to the busy period of each task.



Level-i busy period

It is the interval in which the processor is busy executing tasks with priority higher than or equal to P_i, including blocking times.

Response Time Analysis (FPP)



Busy period of τ_2

Level-i busy period

It can be computed as the shortest interval that satisfies:

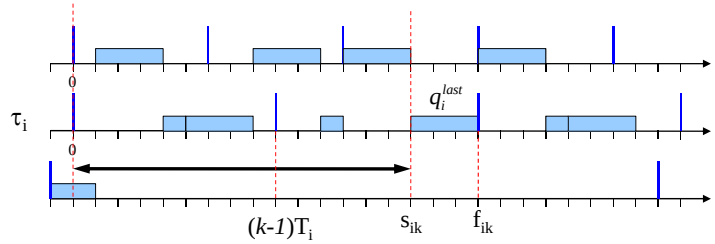
$$L_i = B_i + \sum_{h:P_h \geq P_i} \left\lceil \frac{L_i}{T_h} \right\rceil C_h$$

up to job N_i:

$$N_i = \left\lceil \frac{L_i}{T_i} \right\rceil$$

Initial value can be: $L_i^{(0)} = B_i + \sum_{h:P_h \geq P_i} C_h$

Response Time Analysis (FPP)



$$s_{ik} = B_i + (k-1)C_i + \left(C_i - q_i^{last}\right) + \sum_{h:P_h > P_i} \left(\left\lfloor \frac{s_{ik}}{T_h} \right\rfloor + 1 \right) C_h$$

$$f_{ik} = s_{ik} + q_i^{last}$$

$$R_{ik} = f_{ik} - (k-1)T_i$$

$$R_i = \max_{k \in [1, N_i]} \{R_{ik}\}$$

NOTE:

$$\begin{cases} s_{ik}^{(0)} = (k-1)T_i + \left(C_i - q_i^{last}\right) \\ N_i = \left\lceil \frac{L_i}{T_i} \right\rceil \end{cases}$$

Response Time Analysis (FPP)

for (i=1 to n) {

$N_i = \lceil L_i / T_i \rceil$

$s_{ik}^{(0)} = (k-1)T_i + \left(C_i - q_i^{last}\right)$

$k = 1$

do {

$$s_{ik} = B_i + (k-1)C_i + \left(C_i - q_i^{last}\right) + \sum_{h:P_h>P_i} \left(\left\lceil \frac{s_{ik}}{T_h} \right\rceil + 1 \right) C_h$$

$f_{ik} = s_{ik} + q_i^{last}$

$R_{ik} = f_{ik} - (k-1)T_i$

if ($R_{ik} > R_i$) **then** $R_i = R_{ik}$

if ($R_i > D_i$) **then** *return(UNFEASIBLE)*

k++

} **while** ($k \le N_i$)

}

return(FEASIBLE)

Special cases

- Fully non preemptive scheduling

$$\begin{cases} q_i^{last} = C_i \\ B_i = \max\limits_{P_j<P_i} \{C_j\} \end{cases}$$

- Deferred Preemption

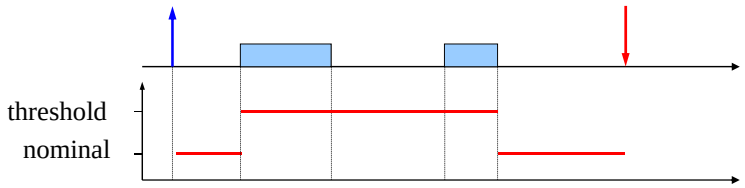
$$\begin{cases} q_i^{last} = 0 \\ B_i = \max\limits_{P_j<P_i} \{Q_j\} \end{cases}$$

Preemption Thresholds (PT)

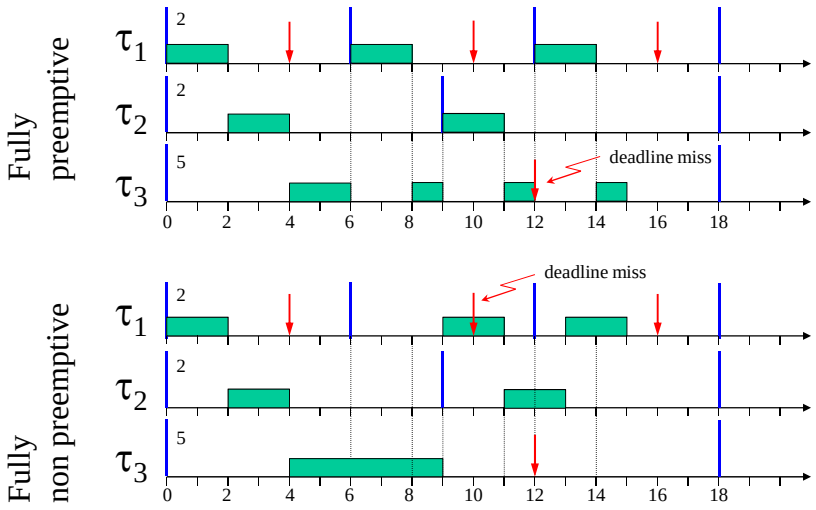
Each task has two priorities:

- P_i nominal priority: used to enqueue the task in the ready queue and to preempt
- θ_i threshold priority: used for task execution

threshold priority ≥ nominal priority

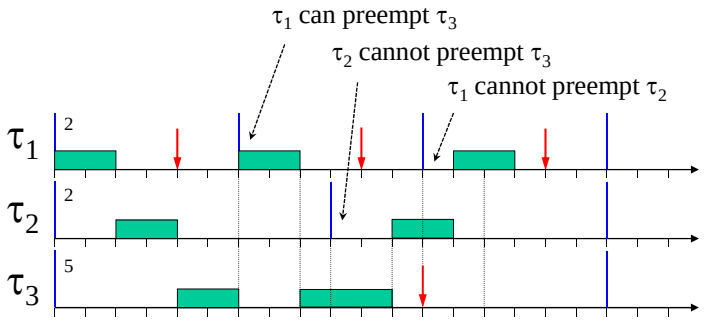


Unfeasible task set



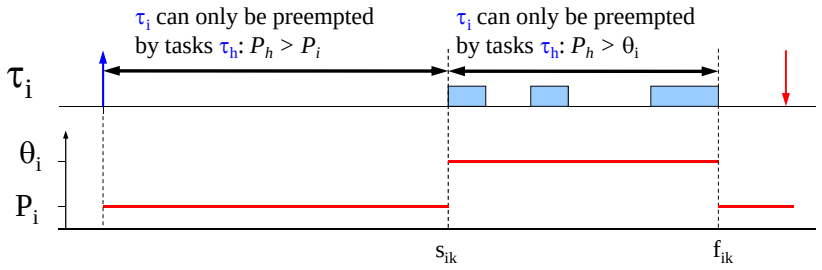
But feasible with preemption thresholds

P_i	θ_i
3	3
2	3
1	2



NOTE:
The same feasible schedule is obtained by splitting τ_3 in two non preemptive chuncks: $q_{31} = 2$, $q_{32} = 3$

Response time analysis (PT)



$$s_{ik} = B_i + (k-1)C_i + \sum_{h:P_h>P_i} \left(\left\lfloor \frac{s_{ik}}{T_h} \right\rfloor + 1 \right) C_h$$

$$f_{ik} = s_{ik} + C_i + \sum_{h:P_h>\theta_i} \left(\left\lceil \frac{f_{ik}}{T_h} \right\rceil - \left\lfloor \frac{s_{ik}}{T_h} \right\rfloor + 1 \right) C_h$$

Remarks

- Preemption Thresholds are easy to specify, but it is difficult to predict the number of preemptions and where they occur $\Rightarrow$ large preemption overhead
- Deferred Preemption allows bounding the number of preemptions but it is difficult to predict where they occur. Note that the analysis assumes $q_i^{last} = 0$
- Fixed Preemption Points allow more control on preemptions and can be selected on purpose (e.g., to minimize overhead, stack size, and reduce WCETs).
- A large final chunk in τ_i reduces the interference from hp-tasks (hence R_i), but creates more blocking to hp-tasks.