Real-Time Scheduling

Giorgio Buttazzo

Scuola Superiore Sant'Anna, Pisa

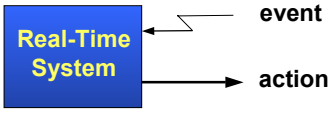
E-mail: buttazzo@sssup.it

Goal

Provide some background of RT theory that you can apply for implementing RT control applications (using Shark):

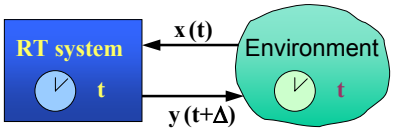
- Terminology and models
- Basic results on periodic scheduling
- Aperiodic task handling
- Inter-task communication
- Overload and QoS management

Real-Time system



A computing system able to respond to events within precise timing constraints.

Real-Time system



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

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

Typical applications

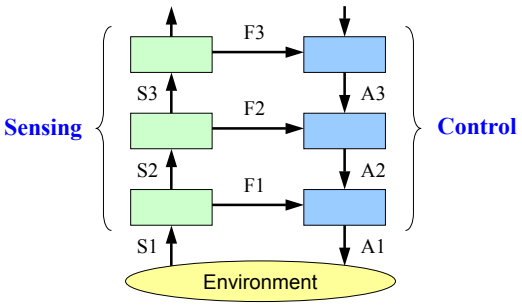
- automotive
- multimedia systems
- robotics
- small embedded devices
  - ⇒ cell phones
  - ⇒ digital TV
  - ⇒ videogames
  - ⇒ intelligent toys

Implications

- Timing constraints are imposed by the dynamics of the environment.
- The tight interaction with the environment requires the system to react to events within precise timing constraints.

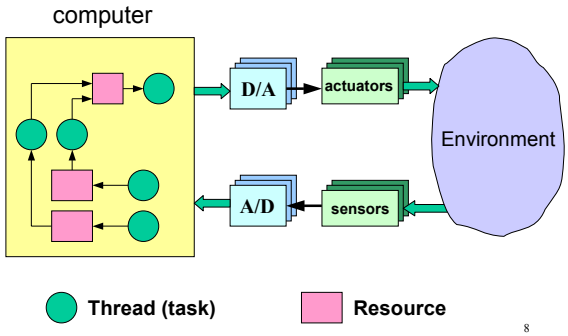
➡ The operating system is responsible for enforcing such constraints on task execution.

Multi-level feedback control



7

Software Vision



8

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

The resulting SW can be very efficient, but …

9

Disadvantages

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

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

10

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

11

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**

12

Typical misconception

"RT system = Fast system"

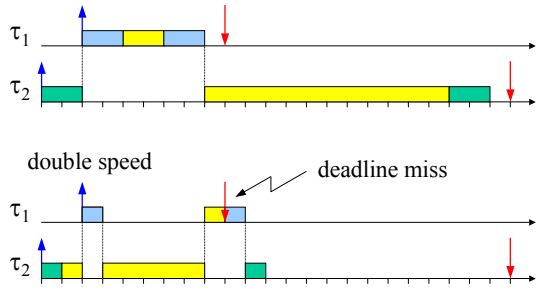
It is not worth studying RT theory, because any timing constraint can be handled by a sufficiently fast computer.

Answers

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

13

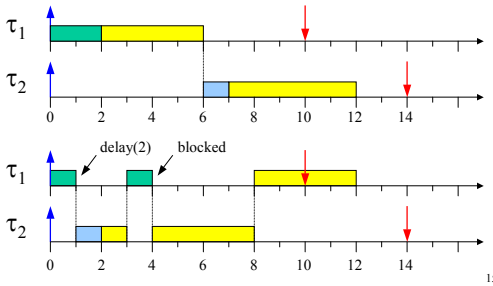
Increasing speed may not always work



14

Never use DELAY

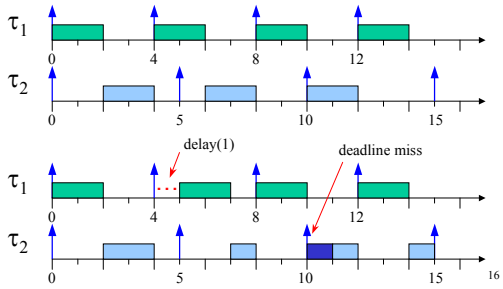
A **delay(Δ)** may cause a delay longer than Δ.



15

Never use DELAY

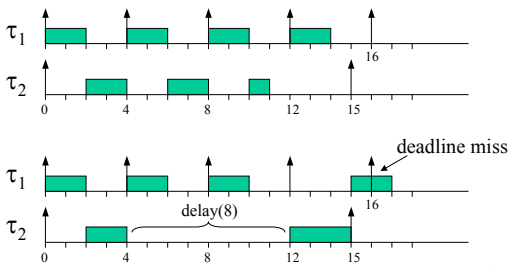
A **delay** in a task may also increase the response time of other tasks (example for fixed priorities):



16

Never use DELAY

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



17

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

18

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

A safe approach:
- ♦ use predictable kernel mechanisms
- ♦ analyze the system to predict its behavior

19

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
  - ⇒ overload management

20

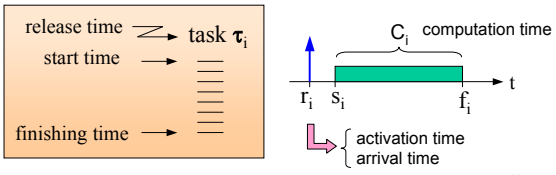
Let's review the main scheduling results

21

Terminology

- **Task** (or thread)

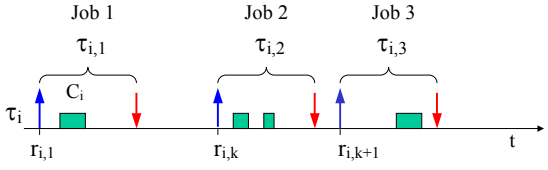
  is a sequence of instructions that in the absence of other activities is continuously executed by the processor until completion.



22

Tasks and jobs

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



23

Activation modes

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

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

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

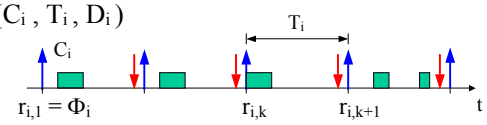
  the task is activated upon the arrival of an event or through an explicit invocation of the activation primitive.

24

Periodic task model

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

$\tau_i (C_i , T_i , D_i)$

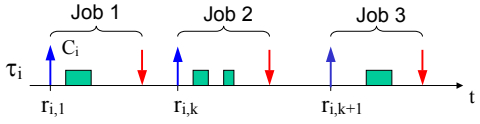


$$r_{i,k} = \Phi_i + (k-1)\,T_i$$
$$d_{i,k} = r_{i,k} + D_i$$

often $D_i = T_i$

Aperiodic task model

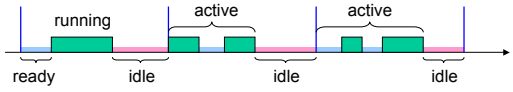
- **Aperiodic:** $r_{i,k+1} > r_{i,k}$
- **Sporadic:** $r_{i,k+1} \geq r_{i,k} + T_i$



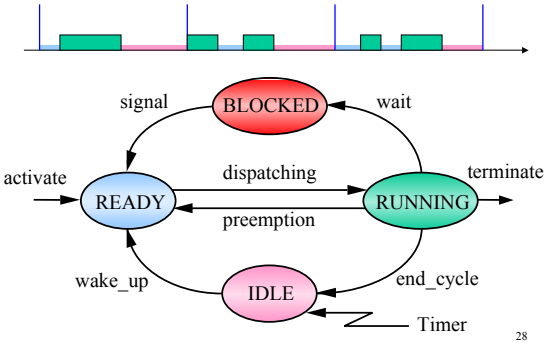
OS support for periodic tasks

task τ_i

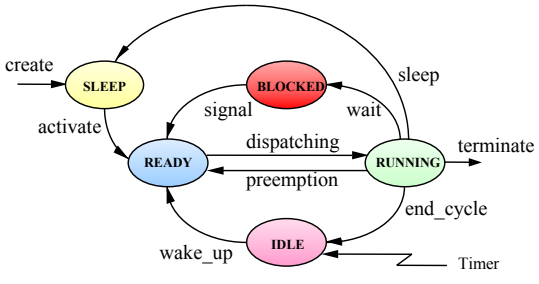
```
while (condition) {
    ...
    wait_for_next_period();
}
```



The IDLE state



SLEEP state



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.

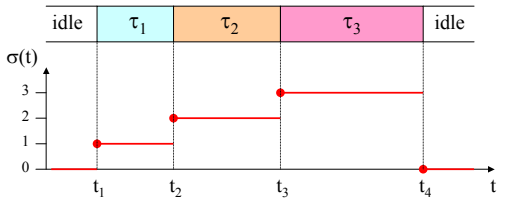
Given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a schedule is a mapping $\sigma : \mathbf{R}^+ \rightarrow \mathbf{N}$ such that $\forall t \in \mathbf{R}^+, \exists t_1, t_2 :$

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

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

31

A sample schedule

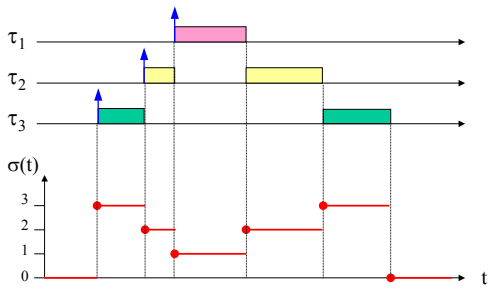


At time t_1, t_2, t_3, e t_4 a **context switch** is performed.

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

32

A preemptive schedule



33

Definitions

- A schedule σ is said to be **feasible** if all the tasks are able to complete within a set of constraints.
- A set of tasks Γ is said to be **schedulable** if there exists a feasible schedule for it.

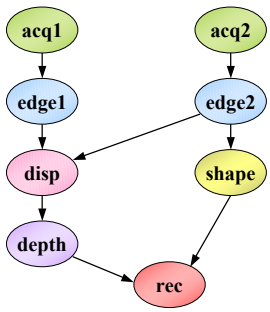
34

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.

35

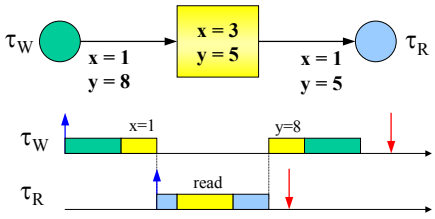
Precedence graph



36

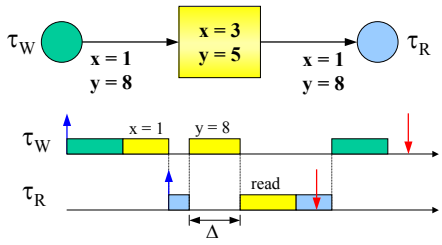
Resource constraints

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



Mutual exclusion

However, mutual exclusion introduces extra delays:



Timing constraints

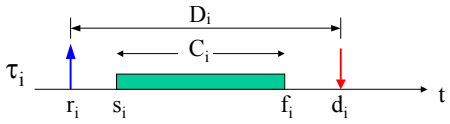
Can be explicit or implicit.

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

Examples

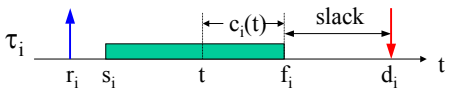
- open the valve **in** 10 seconds
- send the position **within** 40 ms
- read the altimeter **every** 200 ms

Real-Time tasks



$\mathbf{r_i}$	release 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: $\max(0, L_i)$

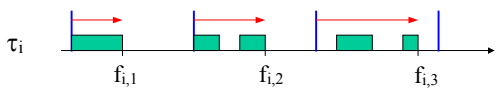
Residual wcet: $c_i(t)$ $c_i(r_i) = C_i$

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

Jitter

It is the time variation of a periodic event:

Finishing-time Jitter

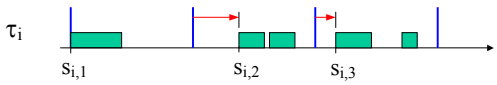


Absolute: $\max_k (f_{i,k} - r_{i,k}) - \min_k (f_{i,k} - r_{i,k})$

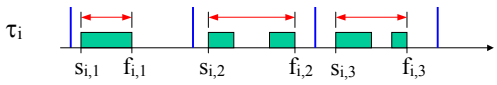
Relative: $\max_k \left| (f_{i,k} - r_{i,k}) - (f_{i,k-1} - r_{i,k-1}) \right|$

Other types of Jitter

Start-time Jitter



Completion-time Jitter (I/O jitter)



43

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.

44

Typical HARD tasks

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

Typical SOFT tasks

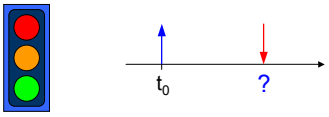
– reading data from the keyboard
– user command interpretation
– message displaying
– graphical activities

45

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

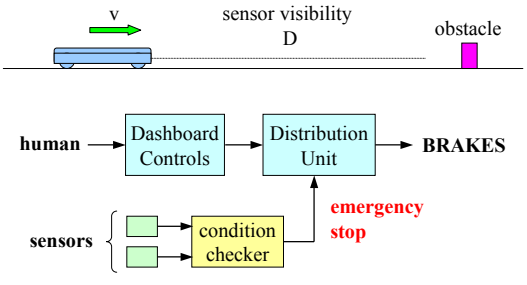
Example

What's the time validity of a sensory data?



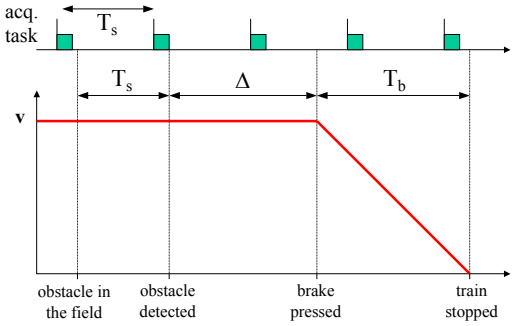
46

Example: automatic breaking



47

Worst-case reasoning



48

D = sensor visibility

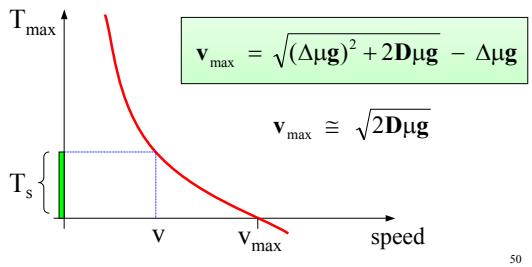
$$v(T_s + \Delta) + X_b < D$$

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

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

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

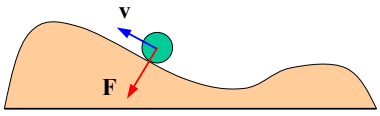
$$T_s < \frac{D}{v} - \frac{v}{2\mu g} - \Delta$$



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

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

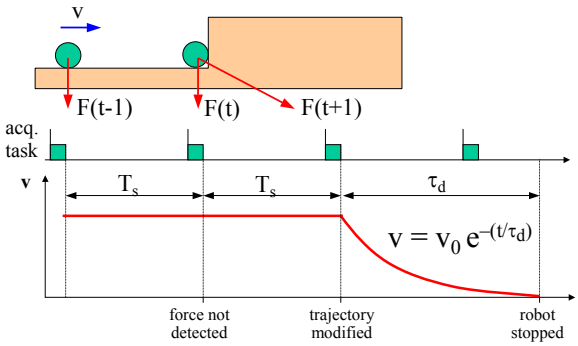
Esempio2: contour following



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:

$$L = vT_s + x_f$$

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

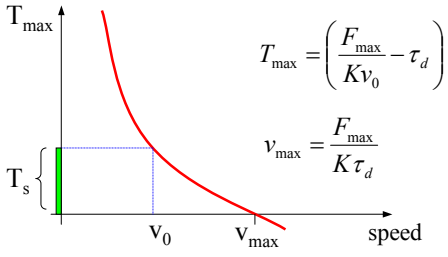
$$L = v(T_s + \tau_d)$$

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

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

Condition on the sampling period:

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

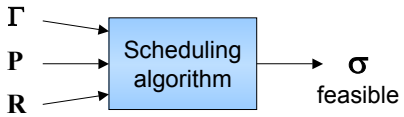


$$T_{max} = \left(\frac{F_{max}}{Kv_0} - \tau_d\right)$$

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

The general scheduling problem

Given a set **Γ** of n tasks, a set **P** of m processors, and a set **R** of r resources, find an assignment of **P** and **R** to **Γ** which produces a feasible schedule.



55

Complexity

- In 1975, Garey and Johnson showed that the general scheduling problem is NP hard.
- However, polynomial time algorithms can be found under particular conditions.

56

Complexity

It's important to find polynomial time algorithms.

number of tasks $n = 30$
elementary step = $1\mu s$

- Alg. 1: $O(n)$ **30 μs**
- Alg. 2: $O(n^6)$ **12 min**
- Alg. 3: $O(6^n)$ **7 billions of years**

57

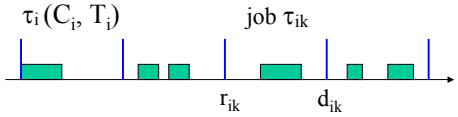
Simplifying assumptions

- Single processor
- Omogeneous task sets
- Fully preemptive tasks
- Simultanoeus activations
- No precedence constraints
- No resource constraints

58

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 $d_{ik} = r_{ik} + D_i$

60

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

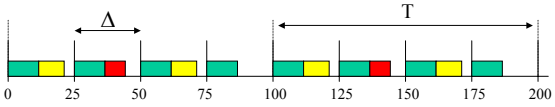
- The time axis is divided in intervals of equal length (***time slots***).
- Each task is statically allocated in a slot in order to meet the desired request rate.
- The execution in each slot is activated by a timer.

Example

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

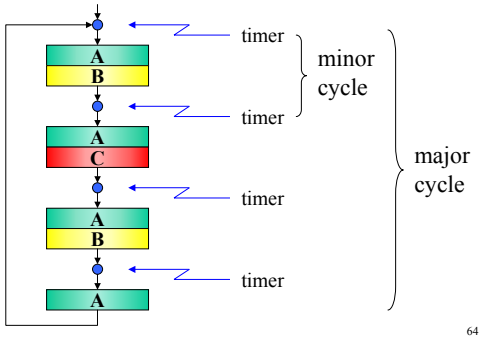
Δ = GCD (minor cycle)

T = lcm (major cycle)



Guarantee: $\begin{cases} C_A + C_B \le \Delta \\ C_A + C_C \le \Delta \end{cases}$

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

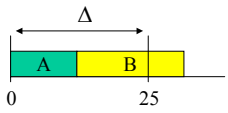
What do we do during task overruns?

- 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

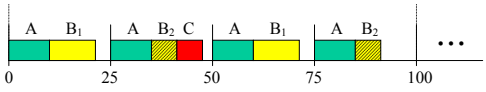
If one or more tasks need to be upgraded, we may have to re-design the whole schedule again.

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: $C_A + C_{B1} \le \Delta$ and $C_A + C_{B2} + C_C \le \Delta$

Expandibility

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

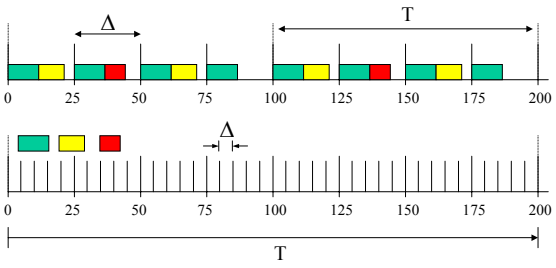
task	T	T
A	25 ms	25 ms
B	50 ms	**40 ms**
C	100 ms	100 ms
	before	**after**

minor cycle: $\Delta = 25$ $\Delta = 5$

major cycle: $T = 100$ $T = 200$

(40 sync. per cycle!)

Example



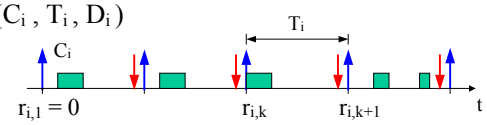
Priority Scheduling

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.

Priority Assignments

$\tau_i (C_i , T_i , D_i)$



$D_i = T_i$

- **Rate Monotonic (RM):**

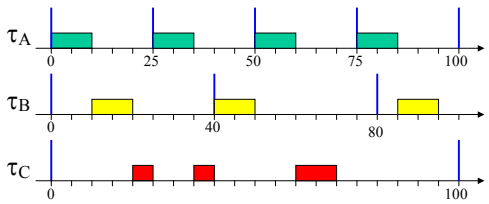
  $p_i \propto 1/T_i$ (static)

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

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

Rate Monotonic (RM)

- Each task is assigned a fixed priority proportional to its rate.



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**

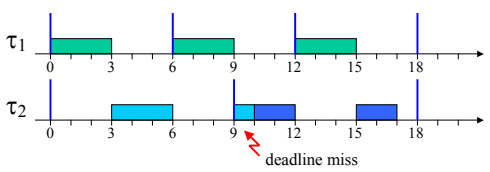
A necessary condition

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

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

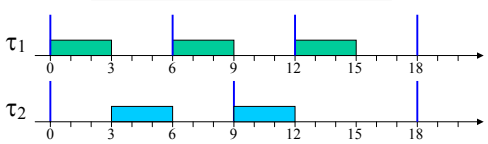
An unfeasible RM schedule

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



Utilization upper bound

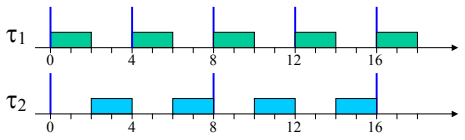
$$U_p = \frac{3}{6} + \frac{3}{9} = 0.833$$



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

A different upper bound

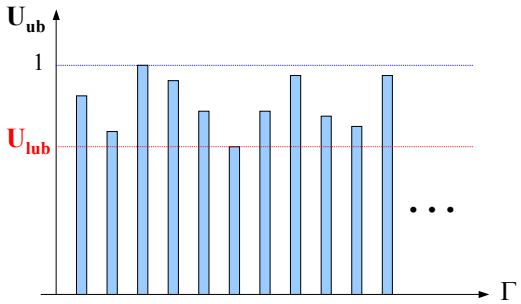
$$U_p = \frac{2}{4} + \frac{4}{8} = 1$$



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

79

The least upper bound



80

A sufficient condition

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

NOTE

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

81

Basic results

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

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

82

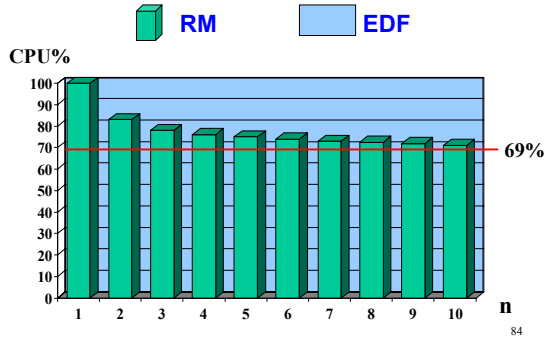
RM bound for large *n*

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

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

83

Schedulability bound

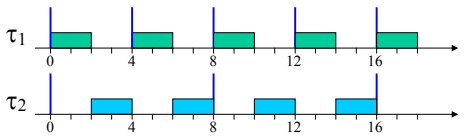


84

A special case

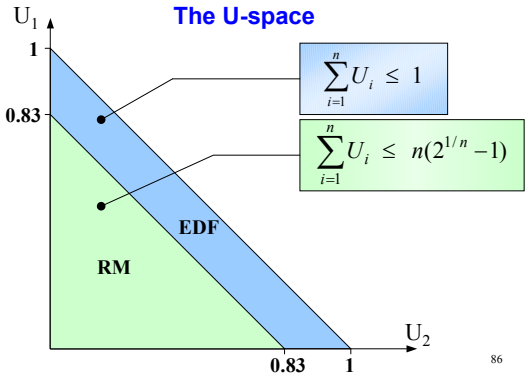
If tasks have harmonic periods $U_{lub} = 1$.

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



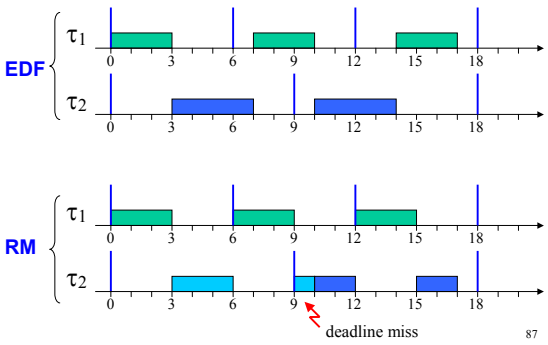
85

Schedulability region



86

Schedule



87

RM Optimality

RM is **optimal** among all fixed priority algorithms:

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.

88

EDF Optimality

EDF is **optimal** among all algorithms:

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

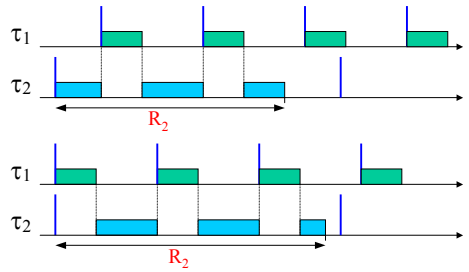
⇕

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

89

Critical Instant

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



90

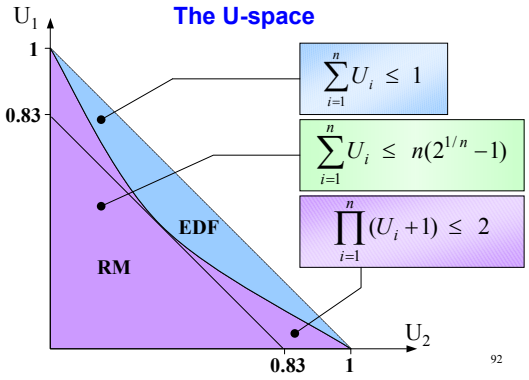
The Hyperbolic Bound

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

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

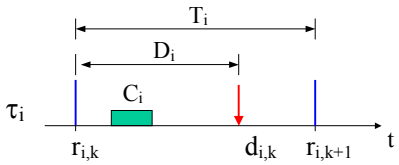
91

Schedulability region



92

Extension to tasks with D < T

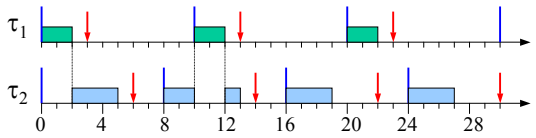


Scheduling algorithms

- Deadline Monotonic: $p_i \propto 1/D_i$ (static)
- Earliest Deadline First: $p_i \propto 1/d_i$ (dynamic)

93

Deadline Monotonic



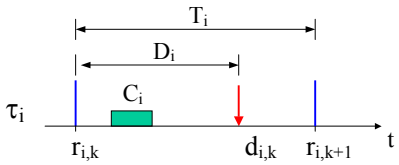
Problem with the Utilization Bound

$$U_p = \sum_{i=1}^{n}\frac{C_i}{D_i} = \frac{2}{3}+\frac{3}{6} = 1.16 > 1$$

but the task set is schedulable.

94

How to guarantee feasibility?



- Fixed priority: Response Time Analysis (RTA)
- EDF: Processor Demand Criterion (PDC)

95

Response Time Analysis [Audsley '90]

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

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

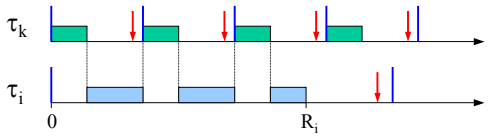
- compute its response time as

$$R_i = C_i + I_i$$

- verify if $R_i \le D_i$

96

Computing the 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 the 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 until $R_i^s > R_i^{(s-1)}$

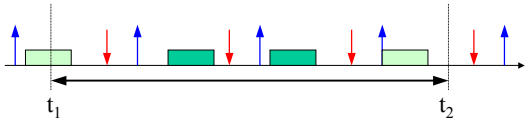
Processor Demand Criterion

[Baruah, Howell, Rosier 1990]

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

$$\forall t_1, t_2 > 0, \quad g(t_1, t_2) \leq (t_2 - t_1)$$

Processor Demand

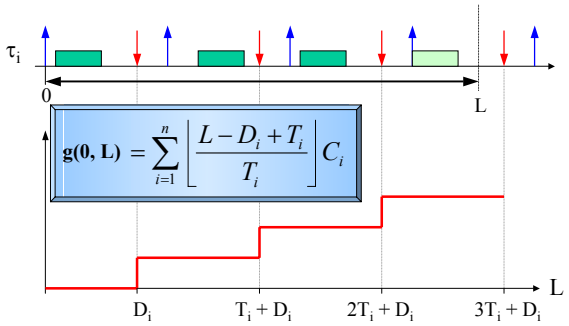


The demand in $[t_1, t_2]$ is the computation time of those jobs started at or after t_1 with deadline less than or equal to t_2:

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

Processor Demand

For synchronous task sets we can only analyze intervals [0,L]



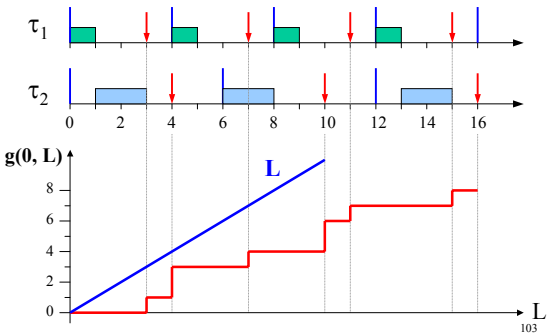
Processor Demand Test

$$\forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i \leq L$$

Question

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

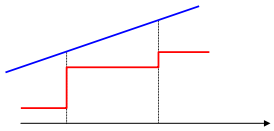
Example



103

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:

$$H = lcm(T_1, \ldots, T_n)$$

104

Bounding complexity

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

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

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

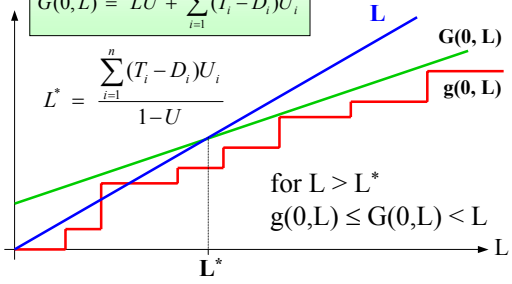
$$= LU + \sum_{i=1}^{n} (T_i - D_i) U_i$$

105

Limiting L

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

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



106

Processor Demand Test

A set of *n* periodic tasks with D ≤ T is schedulable by EDF if and only if

$$U < 1 \quad \text{AND} \quad \forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i \le L$$

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

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

107

Summarizing: RM vs. EDF

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

108

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.

Advantages of EDF

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

- Less overhaed due to preemptions;
- More flexible behavior in overload situations;
- More uniform jitter control;
- Better aperiodic responsiveness.

Runtime overhead

Two different types of overhead are considered:

1. **Overhead for job release**

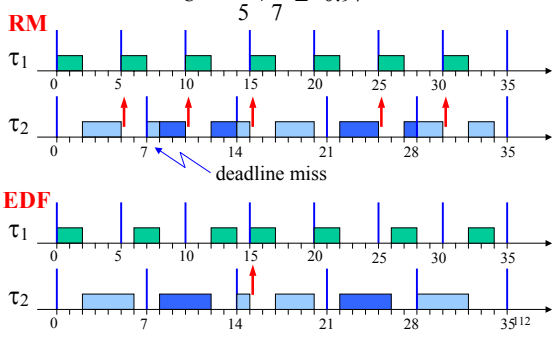
   ⇒ EDF has more than RM, because the absolute deadline must be updated at each job activation

2. **Overhead for context switch**

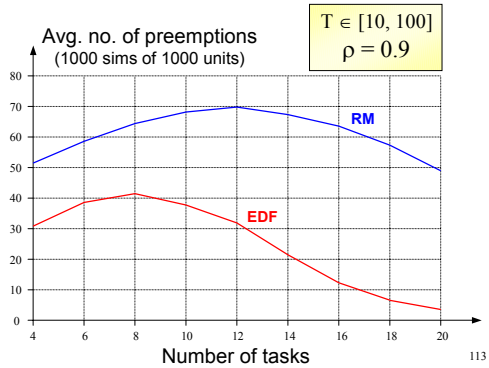
   ⇒ RM has more than EDF because of the higher number of preemptions

Preemptions

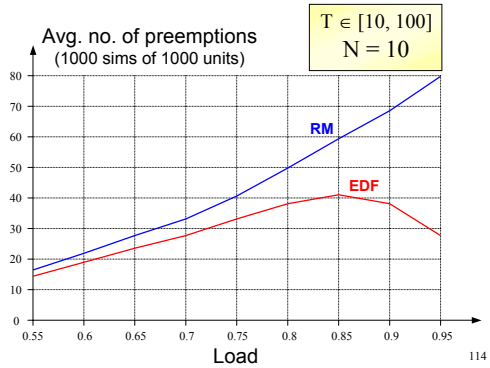
$$U = \frac{2}{5} + \frac{4}{7} \cong 0.97$$



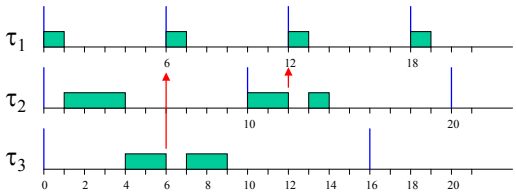
Preemptions



Preemptions



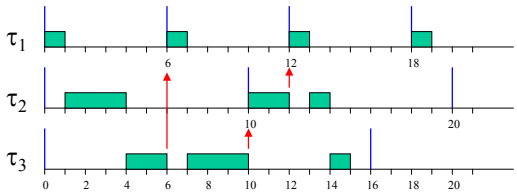
Example with RM



Under **RM**, preemptions increase as computation times increase

115

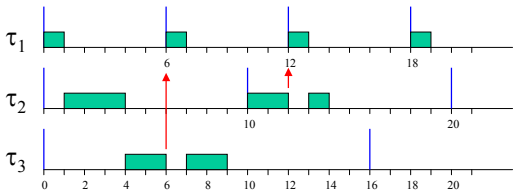
Example with RM



Under **RM**, preemptions increase as computation times increase

116

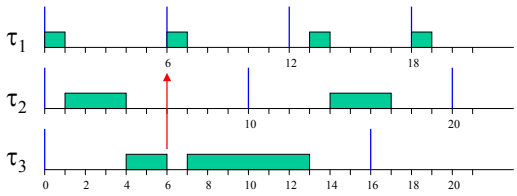
Example with EDF



Under **EDF**, preemptions may decrease as computation times increase

117

Example with EDF



Under **EDF**, preemptions may decrease as computation times increase

118

Robustness under overloads

Two situations are considered:

1. Permanent overload

⇒ This occurs when U > 1

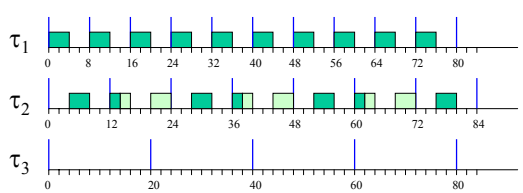
2. Transient overload

⇒ This occurs when some job executes more than expected

119

RM under permanent overload

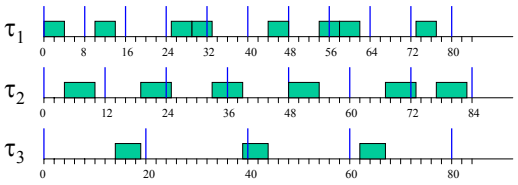
$$U = \frac{4}{8} + \frac{6}{12} + \frac{5}{20} = 1.25$$



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

EDF under permanent overload

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

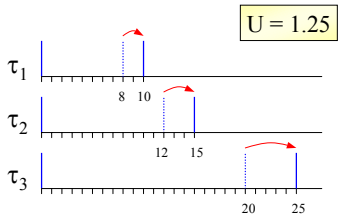


- All tasks execute at a slower rate
- No task is blocked

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

122

Big misconceptions

~~EDF is not predictable during overloads because we don't know which tasks are going to miss their deadlines.~~

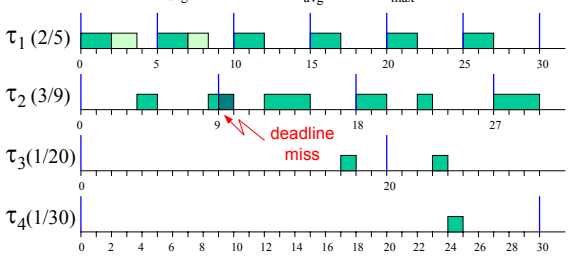
~~RM is predictable during overloads because the tasks that miss their deadlines are low priority tasks.~~

We now show that this is not true

123

RM during transient overruns

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



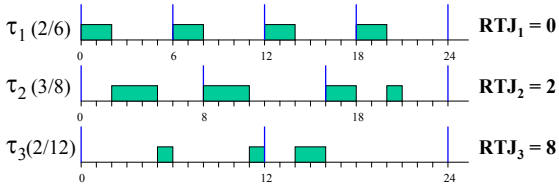
Who is missing its deadline is not the lowest priority task

Another misconception

RM reduces jitter during task execution more than EDF

125

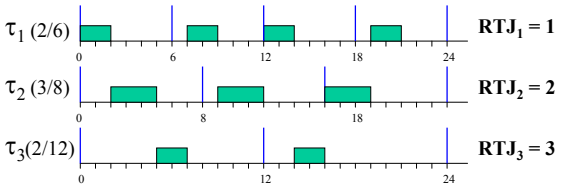
Jitter under RM



τ_3 experiences a very high jitter

126

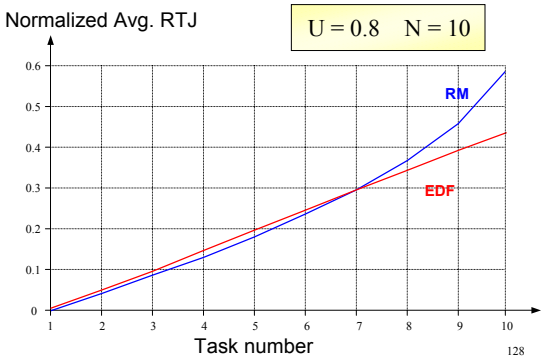
Jitter under EDF



For a little increase of RTJ_1, RTJ_3 decreases a lot

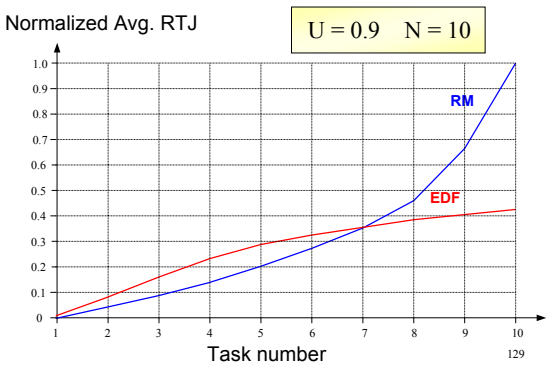
127

Jitter experiments



128

Jitter experiments

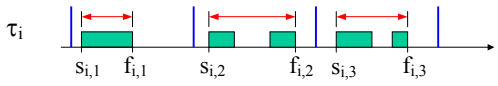


129

Another result on Jitter

Input-Output Jitter

$$IOJ_i = \max (f_{i,k} - s_{i,k}) - \min (f_{i,k} - s_{i,k})$$

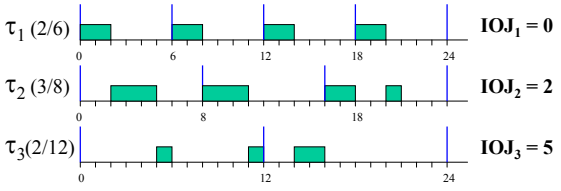


Theorem (Cervin 03)

$$IOJ_i^{EDF} \leq IOJ_i^{RM}$$

130

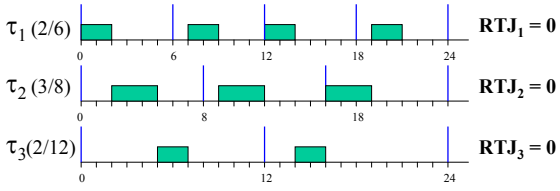
I/O Jitter under RM



All tasks except τ_1 experience I/O jitter

131

I/O Jitter under EDF



All tasks experience no I/O Jitter

132

Handling Aperiodic Tasks

Handling Criticality

- Aperiodic tasks with **HARD** deadlines must be guaranteed under worst-case conditions.
- Off-line guarantee is only possible if we can bound interarrival times (**sporadic tasks**).
- Hence **sporadic tasks** can be guaranteed as periodic tasks with $C_i = WCET_i$ and $T_i = MIT_i$

$$\begin{bmatrix} WCET & = & \text{Worst-Case Execution Time} \\ MIT & = & \text{Minimum Interarrival Time} \end{bmatrix}$$

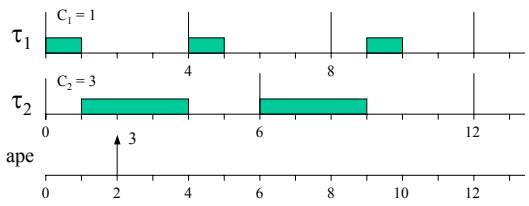
134

SOFT aperiodic tasks

- Aperiodic tasks with **SOFT** deadlines should be executed as soon as possible, but without jeopardizing HARD tasks.
- We may be interested in
  - → minimizing the average response time
  - → performing an on-line guarantee

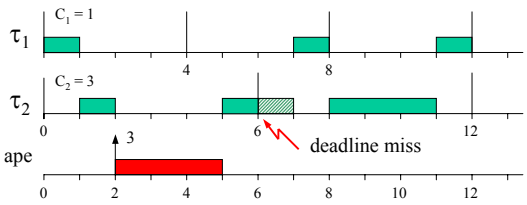
135

Periodic Scheduling
(EDF)



136

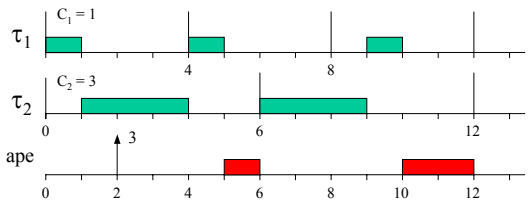
Immediate service



Response Time = 3

137

Background service



Response Time = 10

138

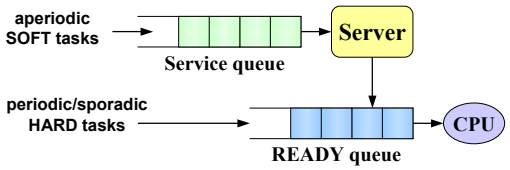
Aperiodic Servers

- A server is a kernel activity aimed at controlling the execution of aperiodic tasks.
- Normally, a server is a periodic task having two parameters:

C_s capacity (or budget)
T_s server period

To preserve periodic tasks, no more than C_s units must be executed every period T_s

Aperiodic service queue



- The server is scheduled as any periodic task.
- Priority ties are broken in favor of the server.
- Aperiodic tasks can be selected using an arbitrary queueing discipline.

Fixed-priority Servers

- Polling Server
- Deferrable Server
- Sporadic Server
- Slack Stealer

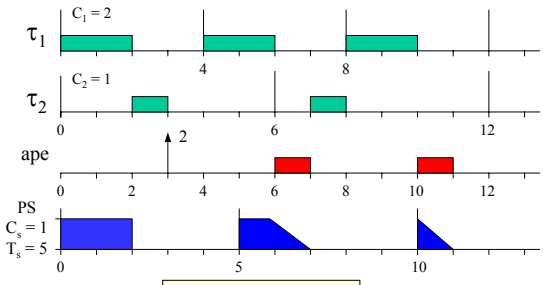
Dynamic-priority Servers

- Dynamic Polling Server
- Dynamic Sporadic Server
- Total Bandwidth Server
- Tunable Bandwidth Server
- Constant Bandwidth Server

Polling Server (PS)

- At the beginning of each period, the budget is recharged at its maximum value.
- Budget is consumed during job execution.
- When the server becomes active and there are no pending jobs, C_s is discharged to zero.
- When the server becomes active and there are pending jobs, they are served until $C_s > 0$.

RM + Polling Server



Response Time = 8

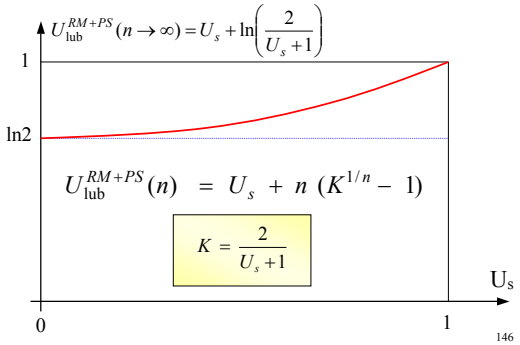
PS properties

- In the worst-case, the PS behaves as a periodic task with utilization $U_s = C_s/T_s$.
- Aperiodic tasks execute at the highest priority if $T_s = \min(T_1, \ldots, T_n)$.
- Liu & Layland analysis gives that:

$$U_{lub}^{RM+PS}(n) = U_s + n\left[\left(\frac{2}{U_s+1}\right)^{1/n} - 1\right]$$

145

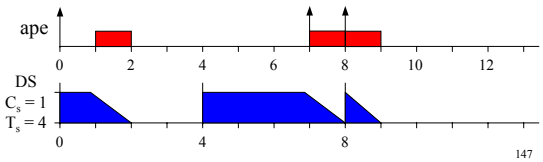
RM + PS schedulability



146

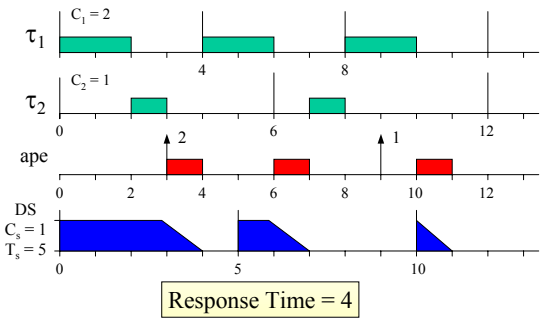
Deferrable Server (DS)

- Is similar to the PS, but the budget is not discharged if there are no pending requests.
- Keeping the budget improves responsiveness, but decreases the utilization bound.



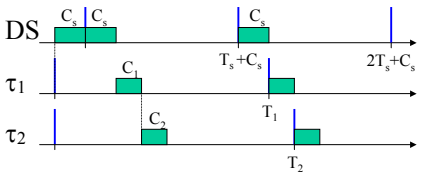
147

RM + Deferrable Server



148

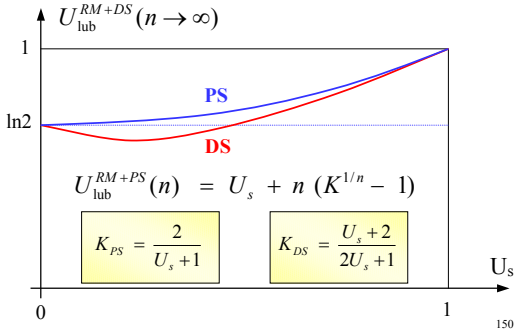
Analysis of RM + DS



$$U_{lub}^{RM+DS}(n) = U_s + n\left[\left(\frac{U_s+2}{2U_s+1}\right)^{1/n} - 1\right]$$

149

RM + DS schedulability



150

Designing server parameters

- Determine U_s^{max} from $U_p \leq n\left[\left(\frac{U_s+2}{2U_s+1}\right)^{1/n} - 1\right]$
- Define $U_s \leq U_s^{max}$
- Define $T_s = \min(T_1, \ldots, T_n)$
- Compute $C_s = U_s T_s$

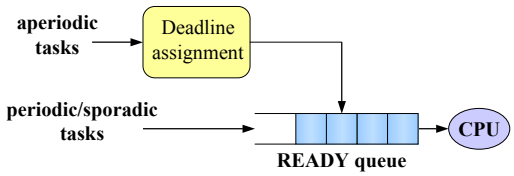
151

Total Bandwidth Server (TBS)

- It is a dynamic priority server, used along with EDF.
- Each aperiodic request is assigned a deadline so that the server demand does not exceed a given bandwidth U_s.
- Aperiodic jobs are inserted in the ready queue and scheduled together with the HARD tasks.

152

The TBS mechanism



- Deadlines ties are broken in favor of the server.
- Periodic tasks are guaranteed *if and only if*

$$U_p + U_s \leq 1$$

153

Deadline assignment rule

- Deadline has to be assigned not to jeopardize periodic tasks.
- A safe relative deadline is equal to the minimum period that can be assigned to a new periodic task with utilization U_s:

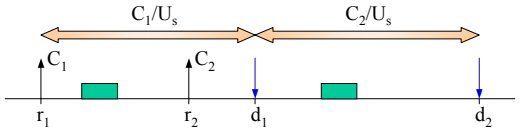
$$U_s = C_k/T_k \Rightarrow T_k = d_k - r_k = C_k/U_s$$

- Hence, the absolute deadline can be set as:

$$d_k = r_k + C_k/U_s$$

154

Deadline assignment rule

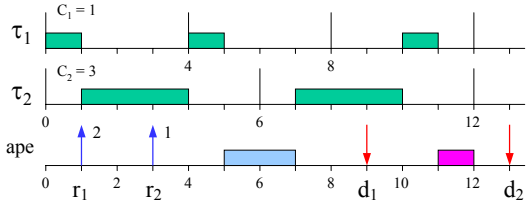


- To keep track of the bandwidth assigned to previous jobs, d_k must be computed as:

$$d_k = \max(r_k, d_{k-1}) + C_k/U_s$$

155

EDF + TBS schedule



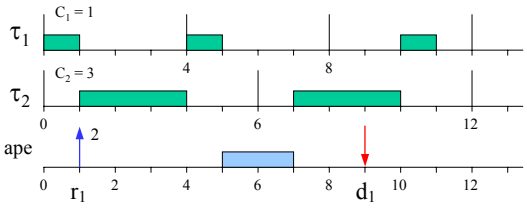
$$U_s = 1 - U_p = 1/4$$

$$\begin{cases} d_1 = r_1 + C_1/U_s = 1 + 2 \cdot 4 = 9 \\ d_2 = \max(r_2, d_1) + C_2/U_s = 9 + 1 \cdot 4 = 13 \end{cases}$$

156

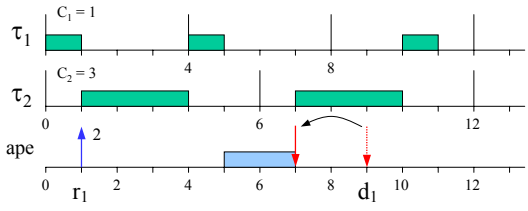
Improving TBS

- What's the minimum deadline that can be assigned to an aperiodic job?



Improving TBS

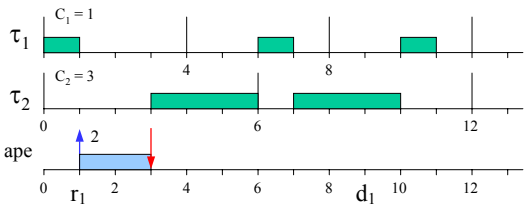
- If we freeze the schedule and advance d_1 to 7, no task misses its deadline, but the schedule is not EDF:



Feasible schedule ≠ EDF

Improving TBS

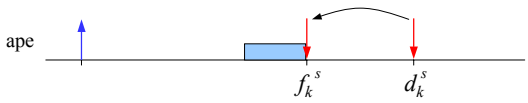
- Clearly, advancing the deadline now does not produce any enhancement:



Computing the deadline

- In general, the new deadline has to be set to the finishing time of the current job:

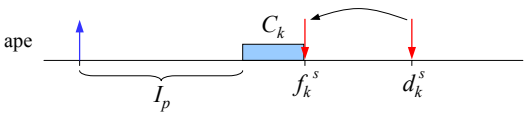
$$\begin{cases} d_k^0 = \max(r_k, d_{k-1}^0) \\ d_k^{s+1} = f_k^s = f_k(d_k^s) \end{cases}$$



Computing the deadline

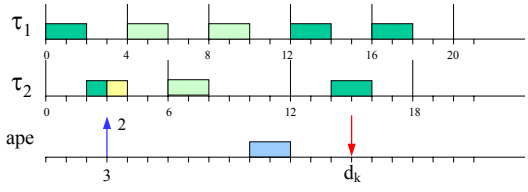
- The actual finishing time can be estimated based on the periodic interference:

$$f_k^s = C_k + I_p(r_k, d_k^s)$$



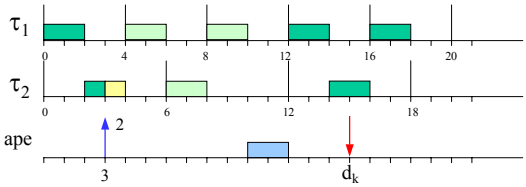
Periodic Interference

$$\begin{cases} U_p = 1/2 + 1/3 = 5/6 \\ U_s = 1 - U_p = 1/6 \end{cases} \qquad \begin{cases} C_k = 2 \\ d_k = 3 + 2/U_s = 15 \end{cases}$$



$$I_p(t, d_k^s) = I_a(t, d_k^s) + I_f(t, d_k^s)$$

Computing interference

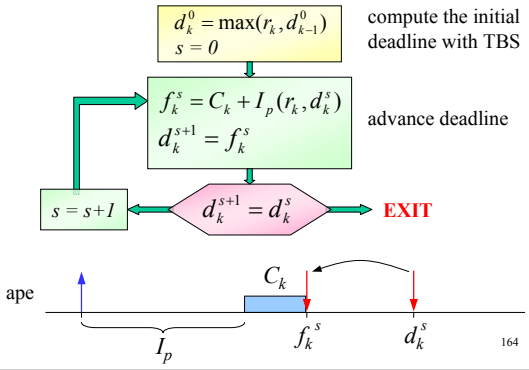


$$I_a(t,d_k^s) = \sum_{\tau_i \text{ active}} c_i(t)$$

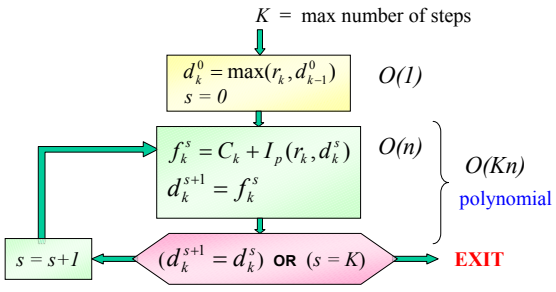
$next_i(t)$ = next release time of task τ_i after t

$$I_f(t,d_k^s) = \sum_{i=1}^{n} \left(\left\lceil \frac{d_k^s - next_i(t)}{T_i} \right\rceil - 1 \right) C_i$$

The Optimal Server

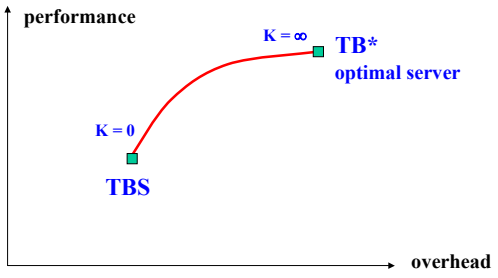


Tunable Bandwidth Server TB(K)

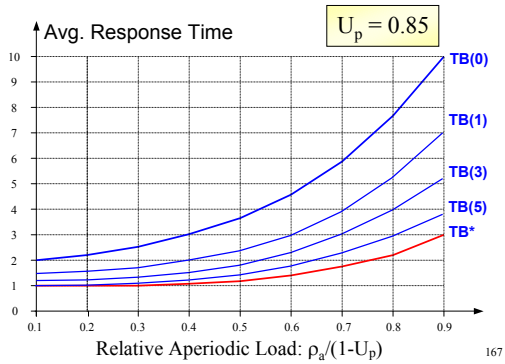


TB(0) = TBS **TB(∞) = TB***

Tuning performance vs. overhead

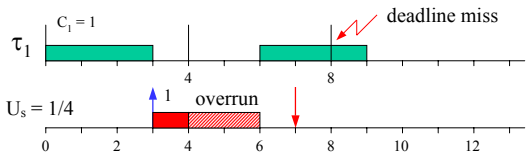


Aperiodic responsiveness



Problems with the TBS

- Without a budget management, there is no protection against execution overruns.
- If a job executes more than expected, hard tasks could miss their deadlines.

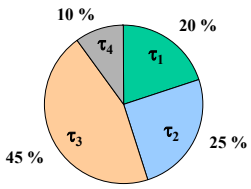


Solution: task isolation

- In the presence of overruns, only the faulty task should be delayed.
- Each task τ_i should not demand more than its declared utilization ($U_i = C_i/T_i$).
- If a task executes more than expected, its priority should be decreased (i.e., its deadline postponed).

Bandwidth partitioning

- Ideally, each task should be assigned a given bandwidth and never demand more.



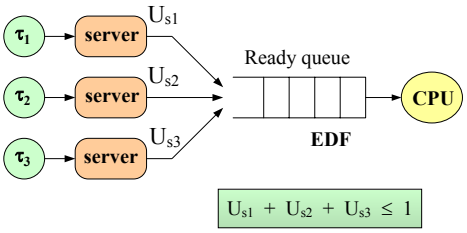
Questions

- **What do we do if a task overruns?**
  - Only that task should be delayed.
- **Consequences**
  - if the task is hard => exception
  - if the task is soft => QoS degradation

Achieving isolation

- Isolation among tasks can be achieved through a **bandwidth reservation**.
- Each task is managed by a dedicated server having bandwidth U_s .
- The server assigns priorities (or deadlines) to tasks so that they do not exceed the reserved bandwidth.

Implementation



Constant Bandwidth Server (CBS)

- It assigns deadlines to tasks as the TBS, but keeps track of job executions through a budget mechanism.
- When the budget is exhausted it is immediately replenished, but the deadline is postponed to keep the demand constant.

CBS parameters

Given by the user

- Maximum budget: Q_s
- Server period: T_s

$U_s = Q_s / T_s$ (server bandwidth)

Maintained by the server

- Current budget: c_s (initialized to 0)
- Server deadline: d_s (initialized to 0)

Basic CBS rules

- **Arrival of job J_k ⇒ assign d_s**

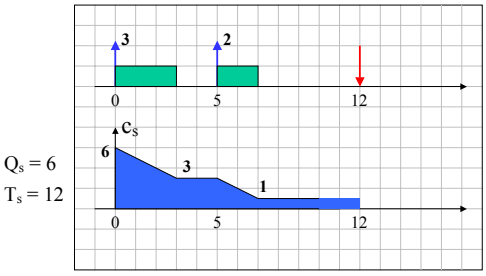
$$\textbf{if } (r_k + c_s/U_s \le d_s) \textbf{ then } \text{recycle } (c_s, d_s)$$

$$\textbf{else } \begin{cases} d_s = r_k + T_s \\ c_s = Q_s \end{cases}$$

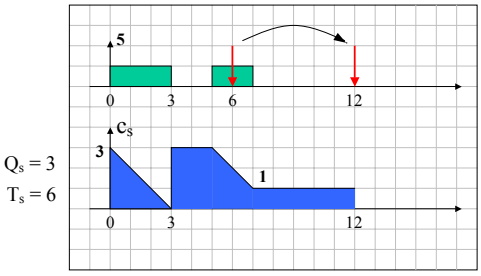
- **Budget exhausted ⇒ postpone d_s**

$$\begin{cases} d_s = d_s + T_s \\ c_s = Q_s \end{cases}$$

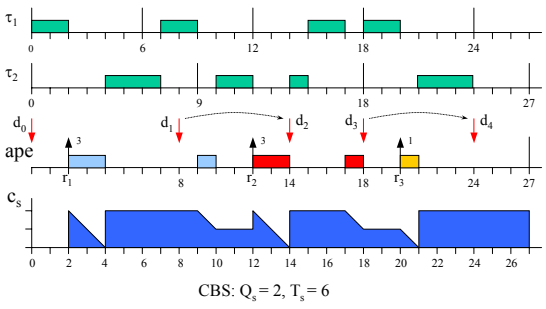
Deadline assignment



Budget exhausted



EDF + CBS schedule



CBS: $Q_s = 2$, $T_s = 6$

CBS properties

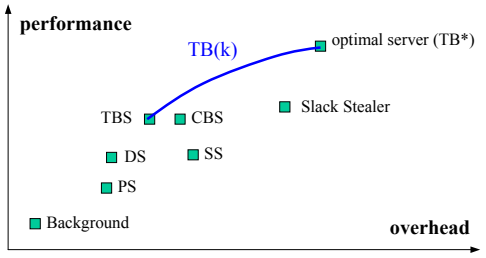
- **Bandwidth Isolation**

  If a task τ_i is served by a CBS with bandwidth U_s then, in any interval Δt, τ_i will never demand more than $U_s \Delta t$.

- **Hard schedulability**

  A hard task τ_i (C_i, T_i) is schedulable by a CBS with $Q_s = C_i$ and $T_s = T_i$, iff τ_i is schedulable by EDF.

Selecting the most suitable service mechanism



It depends on the price (overhead) we want to pay to reduce task response times

181

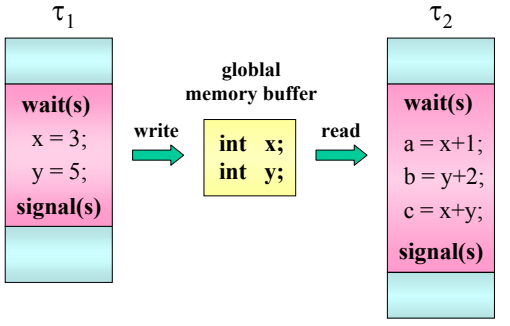
Inter-task communication mechanisms

- Shared memory
- Message passing ports
- Asynchonous buffers

Handling shared resources

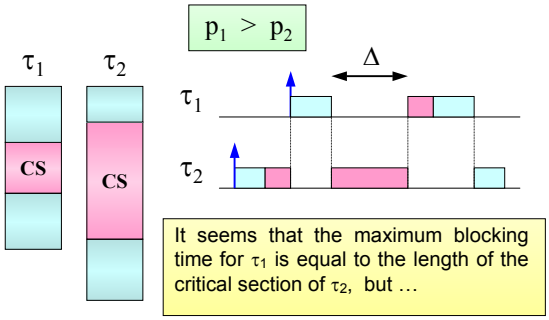
Problems caused by mutual exclusion

Critical sections



184

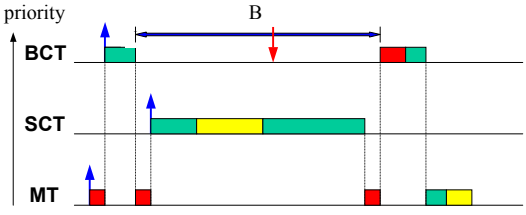
Blocking on a semaphore



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

185

Conflict on a critical section



186

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.

187

Resource Access Protocols

Under fixed priorities

- Non Preemptive Protocol (NPP)
- Highest Locker Priority (HLP)
- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)

Under EDF

- Stack Resource Policy (SRP)

188

Non Preemptive Protocol

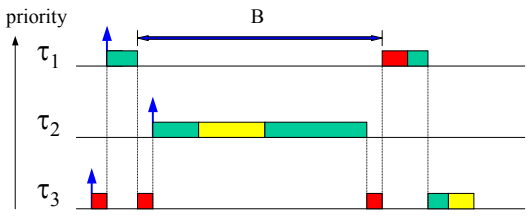
- Preemption is forbidden in critical sections.
- Implementation: when a task enters a CS, its priority is increased at the maximum value.

ADVANTAGES: simplicity

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

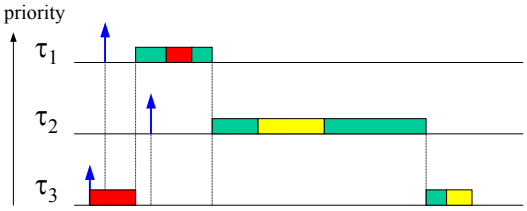
189

Conflict on critical section



190

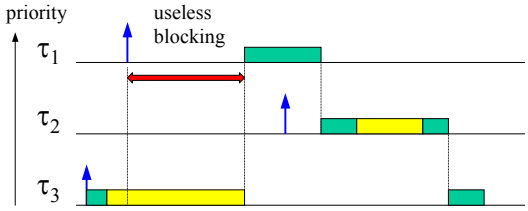
Schedule with NPP



$P_{CS} = \max\{P_1, \ldots P_n\}$

191

Problem with NPP



τ_1 cannot preemt, although it could

192

Highest Locker Priority

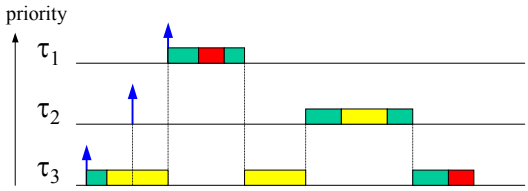
A task in a CS gets the highest priority among the tasks that use it.

FEATURES:

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

193

Schedule with HLP

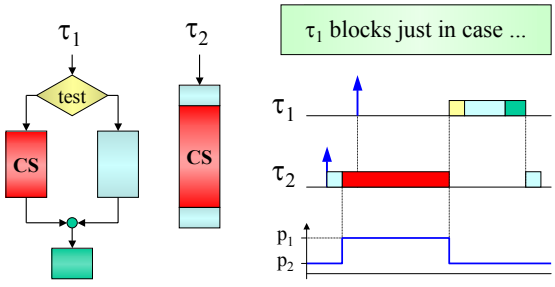


$P_{CS} = \max \{P_k \mid \tau_k \text{ uses CS}\}$

τ_2 is blocked, but τ_1 can preempt within a CS

194

Problem with HLP



195

Priority Inheritance Protocol

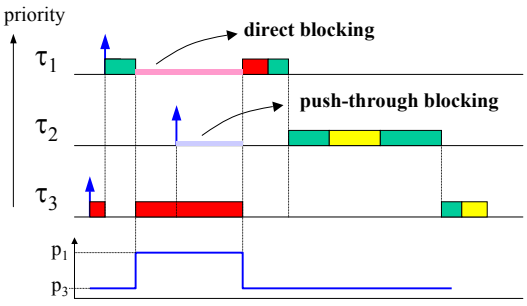
[Sha, Rajkumar, Lehoczky, 90]

- 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.

$P_{CS} = \max \{P_k \mid \tau_k \text{ blocked on CS}\}$

196

Schedule with PIP



197

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

198

Identifying blocking resources

- A task τ_i can be blocked by those semaphores used by lower priority tasks and
  - directly shared with τ_i (direct blocking) or
  - shared with tasks having priority higher than τ_i (push-through blocking).

Theorem: τ_i can be blocked at most once by each of such semaphores

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

199

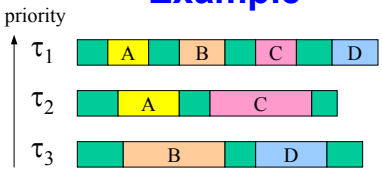
Bounding blocking times

- If **n** is the number of tasks with priority less than τ_i
- and **m** 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 **min(n,m)** critical sections

200

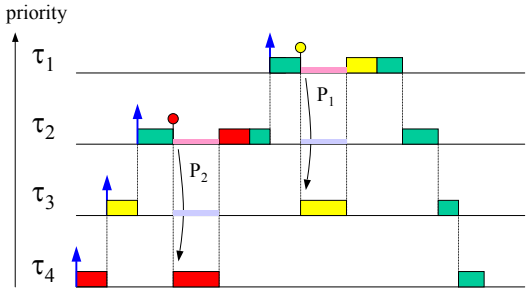
Example



- 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

201

Schedule with PIP



202

Remarks on PIP

ADVANTAGES

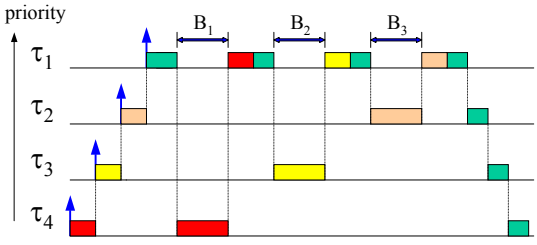
- It is transparent to the programmer.
- It bounds priority inversion.

PROBLEMS

- It does not avoid deadlocks and chained blocking.

203

Chained blocking with PIP



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

204

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

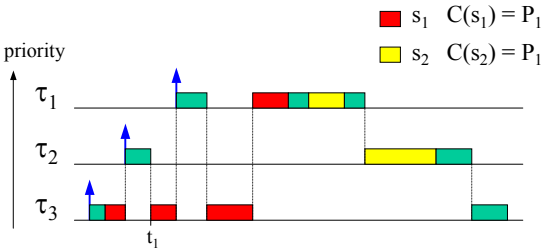
- Each semaphore s_k is assigned a ceiling:

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

- 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 $P_2 < C(s_1)$

Remarks on PCP

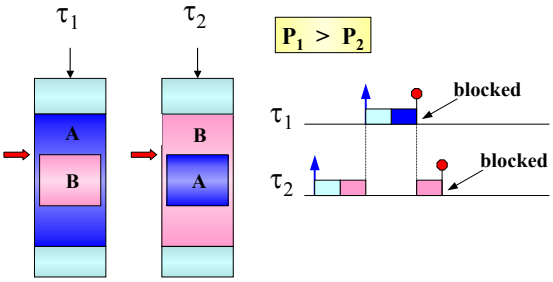
ADVANTAGES

- Blocking is reduced to only one CS
- It prevents deadlocks

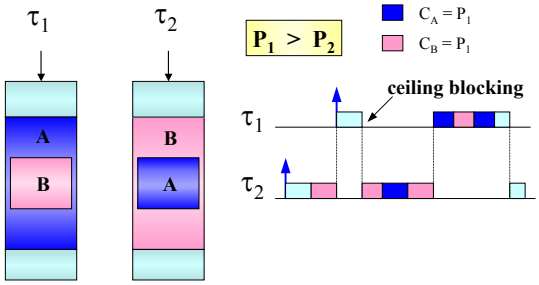
PROBLEMS

- It is not transparent to the programmer: semaphores need ceilings

Typical Deadlock



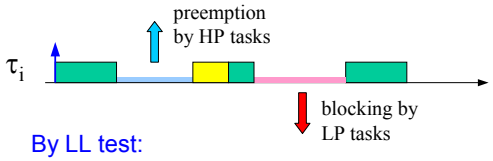
Deadlock avoidance with PCP



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.

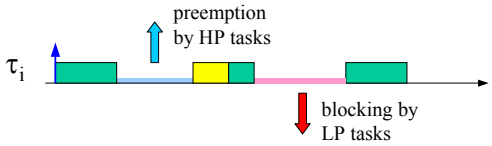
Guarantee with RM (D = T)



By LL test:

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

Guarantee with RM (D ≤ T)



By RTA test: $\forall i \quad R_i \leq D_i$

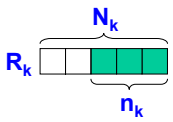
$$R_i = C_i + B_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Stack Resource Policy [Baker 1990]

- It works both with fixed and dynamic priority
- It limits blocking to 1 critical section
- It prevents deadlock
- It supports multi-unit resources
- It allows stack sharing
- It is easy to implement

Stack Resource Policy [Baker 90]

- For each resource R_k:
  - ⇒ Maximum units: N_k
  - ⇒ Available units: n_k



- For each task τ_i the system keeps:
  - ⇒ its resource requirements: $\mu_i(R_k)$
  - ⇒ a priority p_i: RM $p_i \propto 1/T_i$ EDF $p_i \propto 1/d_i$
  - ⇒ a static preemption level: $\pi_i \propto 1/D_i$

Stack Resource Policy [Baker 90]

Resource ceiling

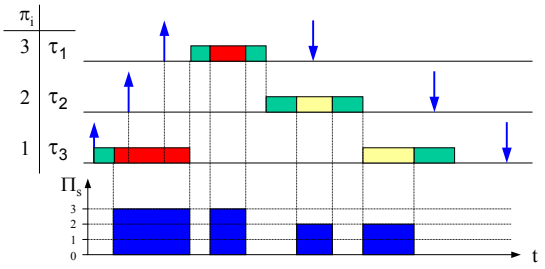
$$C_k(n_k) = \max_j \left\{ \pi_j : n_k < \mu_j(R_k) \right\}$$

System ceiling $\Pi_s = \max_k \{C_k(n_k)\}$

SRP Rule

A job cannot preempt until p_i is the highest and $\pi_i > \Pi_s$

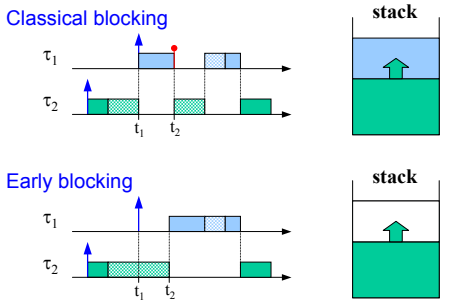
Example



SRP: Notes

- Blocking always occurs at preemption time
- A task never blocks on a wait primitive (semaphore queues are not needed)
- Semaphores are still needed to update the system ceiling
- Early blocking allows stack sharing

SRP: Stack sharing

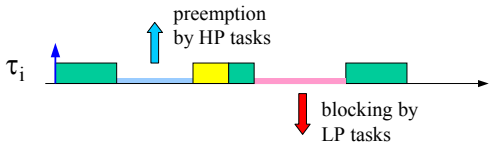


SRP: Stack sharing

- If tasks can be grouped in **M** subsets with the same preemption level, then tasks within a group cannot preempt each other.
- Then the stack size is the sum of the stack memory needed by **M** tasks.
- If we have 100 tasks with 10 preemption levels, and each task requires 10 Kb of stack, then

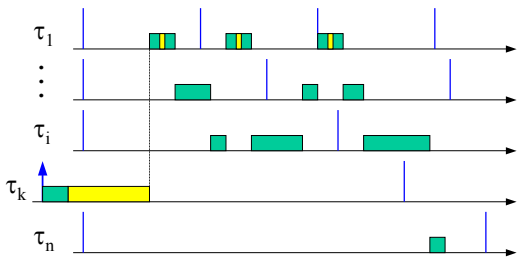
Stack size = { **1 Mb** without SRP ; **100 Kb** under SRP (90% less)

EDF Guarantee ($D_i = T_i$)



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

EDF Guarantee: PD test ($D_i \leq T_i$)



Tasks are ordered by decreasing preemption level

EDF Guarantee: PD test ($D_i \le T_i$)

$$\forall i \quad \forall L: \; D_i \le L \le \max(D_n, L_i^*)$$
$$U < 1 \quad \text{AND} \quad g_i(0,L) \le L$$

$$\begin{cases} g_i(0,L) = B_i + \sum_{k=1}^{i} \left\lceil \frac{L - D_k + T_k}{T_k} \right\rceil C_k \\ L_i^* = \frac{B_i + \sum_{k=1}^{i} (T_k - D_k) U_k}{1 - \sum_{k=1}^{i} U_k} \end{cases}$$

Message passing paradigm

Every task operates on a private memory space, exchanging messages through channels:

Channel: logical link by which two tasks can communicate.

Message: set of data having a predefined format.

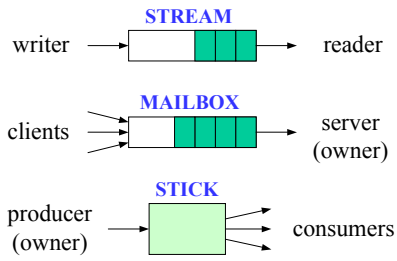
Communication Ports

- The operating system provides the channel abstraction through the **port** construct.
- A task can use a port to exchange messages by means of two primitives:

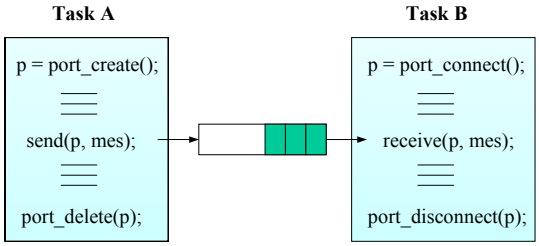
send sends a message to a port

receive receives a message from a port

Port types



Using a port



NOTE: Task A is the owner and must start first.

Port primitives

- ***port_create****(name, size, num, type, access);*

  name: identification string
  size: message size (in bytes)
  num: maximum number of messages
  type: STREAM, MAILBOX, STICK
  access: READ, WRITE
  It returns a port identifier.

- ***port_delete****(port_id);*

  Deletes the specified port.

Port primitives

- ***port_connect****(name, size, type, access);*

  name: stringa di identificazione
  size: message size (in bytes)
  type: STREAM, MAILBOX, STICK
  access: READ, WRITE
  It returns a port identifier.

- ***port_disconnect****(port_id);*

  Deletes the specified port.

229

Port primitives

- ***port_send*** *(port_id, msg_ptr, sync)*

  sends the message pointed by *msg_ptr* to the port identified by *port_id*.
  sync = BLOCK blocks on a full buffer
  sync = NON_BLOCK returns 0

- ***port_receive****(port_id, msg_ptr, sync)*

  receives a message from *port_id* and copies it into the buffer pointed by *msg_ptr*.
  sync = BLOCK blocks on an empty buffer
  sync = NON_BLOCK returns 0

230

A task creates a port to send messages of 6 bytes. The port can keep up to 8 messages.

```
TASK    writer(void)
{
PORT    p;
char    mes[6];

    p = port_create("door", 6, 8, STREAM, WRITE);
    while (condition) {
        build_message(mes);
        port_send(p, mes, BLOCK);
        task_endcycle();
    }
    port_delete(p);
}
```

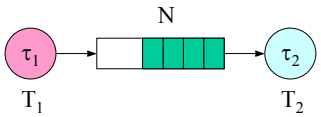
231

A task connects to an already opened port to receive messages of 2 bytes.

```
TASK    reader(void)
{
PORT    q;
char    data[2];

    q = port_connect("door", 2, STREAM, READ);
    while (condition) {
        if (port_receive(q, data, NON_BLOCK))
                action1();
        else    action2();
        task_endcycle();
    }
    port_disconnect(q);
}
```

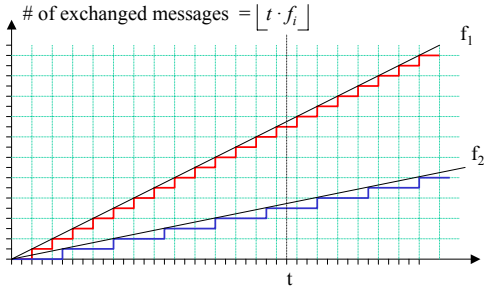
Periodic task communication



- If $T_1 < T_2$, τ_1 puts more messages than τ_2 can read.
- When the buffer is full, τ_1 must proceed with the same rate of τ_2.

233

Exchanged messages



234

Buffer Saturation

If $T_1 < T_2$, the buffer saturates when:

$$\left\lfloor \frac{t}{T_1} \right\rfloor - \left\lfloor \frac{t}{T_2} \right\rfloor > N$$

Hence, the tasks proceed at their proprer rate while:

$$\left\lfloor \frac{t}{T_1} \right\rfloor - \left\lfloor \frac{t}{T_2} \right\rfloor < \frac{t}{T_1} - \frac{t}{T_2} + 1 < N$$

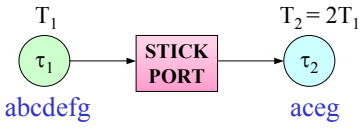
That is, while:

$$t < (N-1)\frac{T_1 T_2}{T_2 - T_1}$$

STICK Ports

- If $T_1 < T_2$, τ_1 overwrites previous messages.

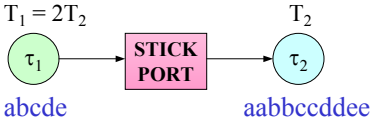
Example:



STICK Ports

- If $T_1 > T_2$, τ_2 reads the same message more than once (messages are not consumed).

Example:



Blocking on STICK ports

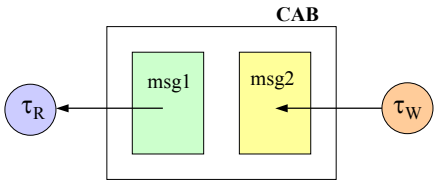
- STICK ports use a semaphore to avoid simultaneous accesses to the internal buffer.
- Long messages may cause long blocking delays on the semaphore.
- A more efficient solution avoids blocking through a buffer replication mechanism.

Cyclic Asynchronous Buffer

- It is a mechanism for exchanging messages among periodic tasks with different rates.
- It avoids memory conflicts by replicating the internal buffers.
- *State message* semantics: messages are overridden by senders and are not consumed by receivers.

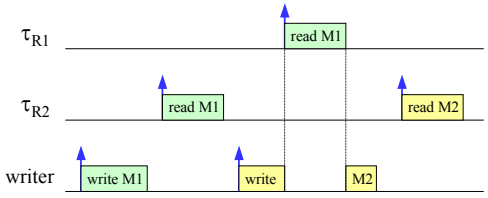
Simultaneous accesses

If a writer task τ_W arrives while a task τ_R is reading, the new message is written in a new buffer:



Reading from a CAB

Once written, a message becomes available to the next reader:

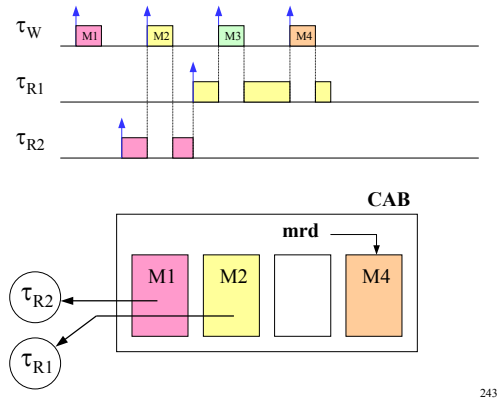


241

Accessing a CAB

- CABs are accessed through a memory pointer.
- Hence, a reader is not forced to copy the message in its memory space.
- More tasks can simultaneously read the same message.
- At each instant, a pointer (**mrd**) points to the most recent message stored in the CAB.

242



243

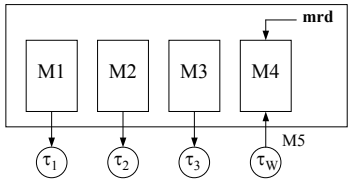
Dimensioning a CAB

If a CAB is used by N tasks, to avoid blocking, it must have at least N +1 buffers.

- The (N+1)-th buffer is needed for keeping the most recent message in the case all the other buffers are used.

244

Inconsistency with N buffers



- Assume all buffers are used and τ_W overwrites the most recent message (M4) with M5.
- If (while τ_W is writing) τ_1 finishes and requests a new message, it finds the CAB inconsistent.

245

Writing in a CAB

```
    . . .
p = cab_reserve(cab_id);
<copy message in *p>
cab_putmes(cab_id, p);
    . . .
```

246

Reading from a CAB

```
    . . .
p = cab_getmes(cab_id);
<process message with *p>
cab_unget(cab_id, p);
    . . .
```

247

CAB Primitives

- ***cab_create****(cab_name, buf_size, max_buf);*

  creates a CAB with *max_buf* buffers
  with size *buf_size* bytes.
  It returns a global CAB identifier.

- ***cab_delete****(cab_id);*

  deletes the specified CAB.

248

CAB Primitives

- ***cab_reserve****(cab_id)*

  returns a pointer to write in a free buffer

- ***cab_putmes****(cab_id, pointer)*

  releases the pointer after a write operation

- ***cab_getmes****(cab_id)*

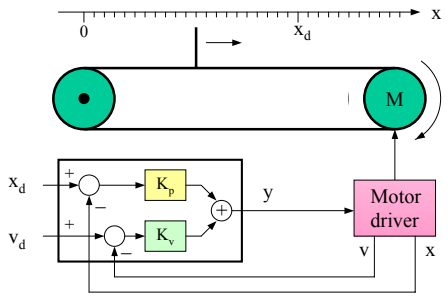
  returns a pointer to the most recent message

- ***cab_unget****(cab_id, pointer)*

  releases the pointer after a read operation

249

Position control



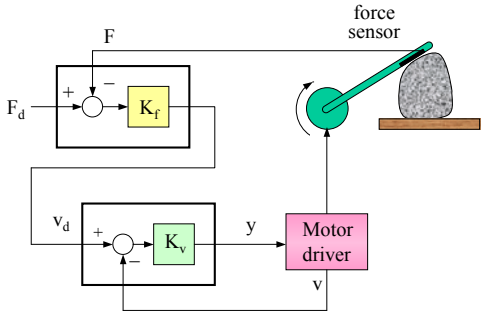
250

PD regulator

```
TASK    pos_control()
{
int     xd, vd, x, v;
float   y, Kp, Kv;
    while (1) {
        get_gains(&Kp, &Kv);
        get_setpoint(&xd, &vd);
        read_sensors(&x, &v);
        y = Kp*(xd - x) - Kv*(vd - v);
        output(y);
        task_endcycle();
    }
}
```

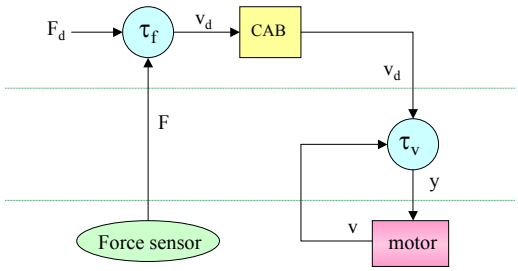
251

Two-level control loop



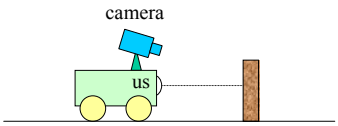
252

Using CABs



253

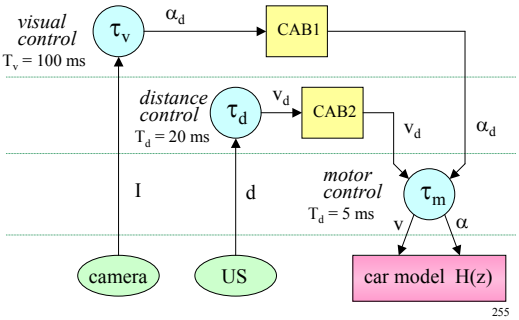
Using visual feedback



- An ultrasound sensor is used for speed control
- The camera is used for stear control

254

Three-level control



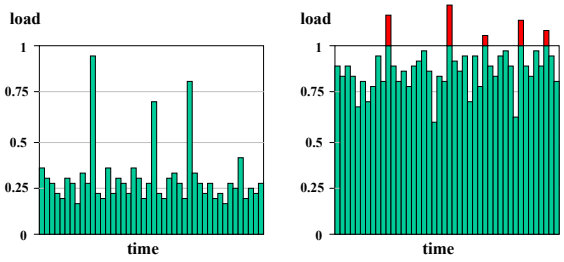
255

Overload management

Examples of load

System designed under worst-case assumptions

System designed under average-case assumptions



257

Transient vs. permanent overload conditions

Transient overload: $\rho_{avg} < 1$, $\rho_{max} > 1$

Possible causes

- ⇒ Arrival of aperiodic activities
- ⇒ Exceptions raised by the kernel
- ⇒ Malfunctioning of input devices
- ⇒ Task with variable execution
- ⇒ Sporadic overruns

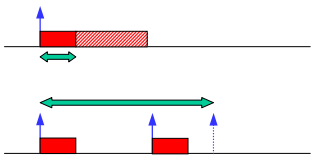
258

Types of overruns

- A task is said to be in overrun if the time demanded for execution exceeds the expected value according to which the task has been guaranteed.
- There are two types of overrun:

Execution overrun
A job executes more than expected

Activation overrun
A job arrives before the time it is expected



Transient vs. permanent overload conditions

Permanent overload: $\rho_{avg} > 1$

Possible causes

⇒ Activation of a new periodic task
⇒ Increase in the task frequencies
⇒ Increase in the task quality (execution times)
⇒ Changes in the environment
⇒ Bad system design

Overload management methods

- **Activation overruns of aperiodic tasks**
  - ⇒ **Value-based scheduling**
    - tasks are assigned values and executed accordingly
  - ⇒ **Admission control**
    - least importance tasks are rejected
    - important tasks receive full service
- **Activation/Execution overruns**
  - ⇒ **Resource Reservation**
    - Tasks cannot use more than reserved
- **Permanent overload of periodic tasks**
  - ⇒ **Performance degradation**
    - all tasks are executed
    - but with reduced requirements

Value-based scheduling

- If $\rho > 1$, no all tasks can finish within their deadline.
- To avoid domino effects, the load is reduced by rejecting the least important tasks.
- To do that, the system must be able to handle tasks with both timing constraints and importance values.

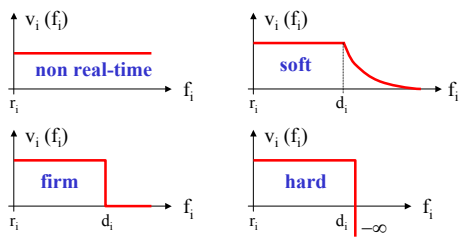
How to assign values

A task τ_i can be assigned a value v_i according to different criteria. Those most common are:

$v_i = V_i$	arbitrary constant
$v_i = C_i$	computation time
$v_i = V_i/C_i$	value density

Value as a function of time

In a real-time system, the value of a task depends on its completion time and criticality:



Performance evaluation

- The performance of a scheduling algorithm A on a task set T can be evalutated through its ***Cumulative Value***:

$$\Gamma_A(T) = \sum_{i=1}^{n} v_i(f_i)$$

- Note that: $\Gamma_A(T) < \Gamma_{max}(T) = \sum_{i=1}^{n} V_i$

Optimality under overloads

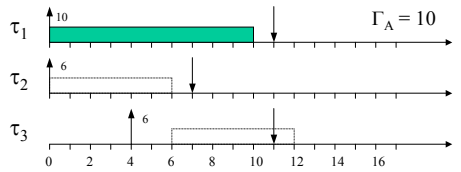
$$\Gamma^*(T) = \max_A \Gamma_A(T)$$

The performance of an algorithm can be evalutated with respect to Γ^*.

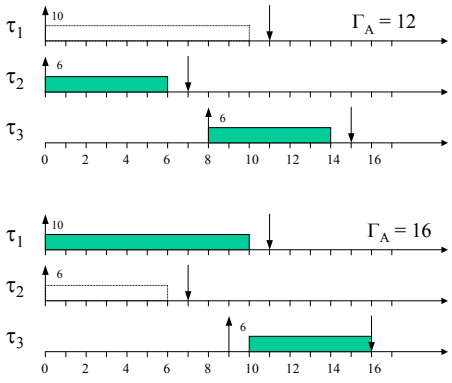
In overload conditions, there are no optimal **on-line** algorithms able to guarantee a cumulative value equal to Γ^*.

Proof (assume: $V_i = C_i$)

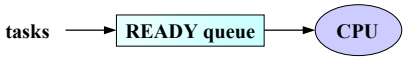
To maximize Γ_A we should know the future.



If at time t = 0 r_3 is not know, we cannot select the task that maximizes the cumulative value.

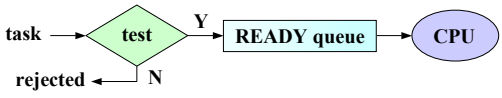


Best-effort scheduling



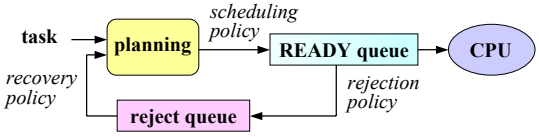
- Tasks are always accepted in the system.
- Performance is controlled through a suitable (value-based) priority assignment.
- **Problem:** domino effect.

Admission control



- Every task is subject to an acceptance test which keeps the load ≤ 1.
- It prevents domino effects, but does not take values into account.
- Low efficiency due to the worst-case guarantee (tasks may be unnecessarily rejected).

Robust scheduling



- Task scheduling and task rejection are controlled by two separate policies.
- Tasks are scheduled by deadline, rejected by value.
- In case of early completions, rejected tasks can be recovered by a reclaiming mechanism.

271

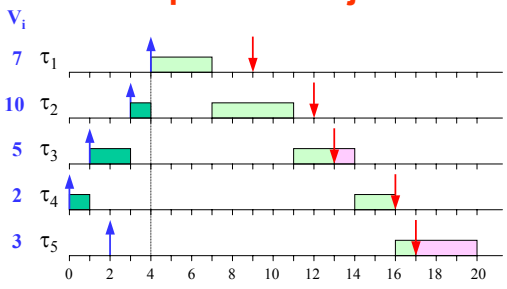
Robust EDF

- **Scheduling Policy** ⇒ **EDF**
- **Rejection policy**

  when an overload is detected, reject the least value task which can bring the load below 1.
- **Recovery policy**
  - keep rejected tasks by decreasing values;
  - when there is enough spare time, re-accept the highest value task which is still feasible.

272

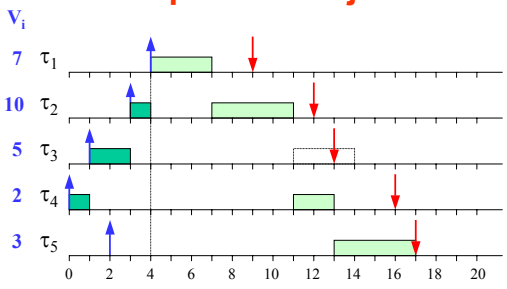
Example: task rejection



at time t = 4 ⇒ τ_3 rejected

273

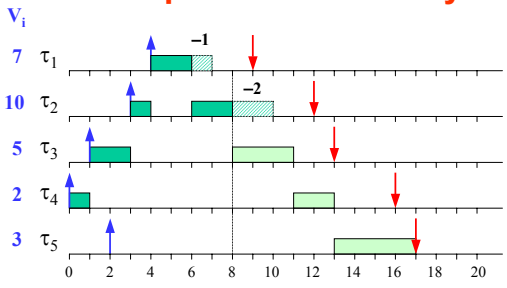
Example: task rejection



at time t = 4 ⇒ τ_3 rejected

274

Example: task recovery



at time t = 8 ⇒ τ_3 can be recovered

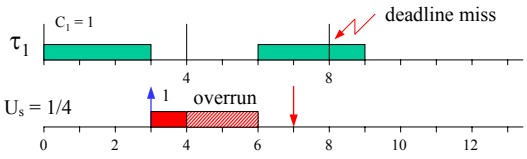
275

Resource Reservation

Handling sporadic overruns

Problems with overruns

- Without a budget management, there is no protection against execution overruns.
- If a job executes more than expected, hard tasks could miss their deadlines.



277

Solution: Temporal Isolation

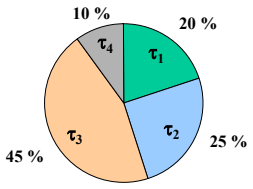
- The execution of a task should not affect the guarantee performed on the other tasks.
- Each task τ_i receives a fraction U_i of the processor (its *bandwidth*) and behaves as it were executing alone on a slower processor of speed U_i.

Temporal isolation ⇨ **bandwidth reservation**, **bandwidth enforcement**

278

Bandwidth reservation

- Ideally, each task should be assigned a given bandwidth and never demand more.



- However, tasks are subject to **overruns** or the reserved bandwidth can be **insufficient** for the task.

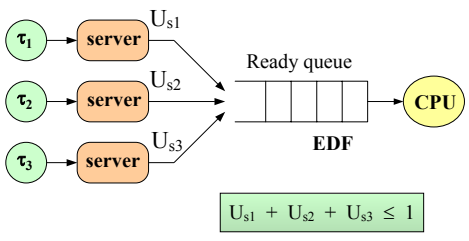
279

Bandwidth enforcement

- It is a mechanism needed for degrading the QoS when a task demands more than the reserved bandwidth.
- If a task executes more than expected, its priority should be decreased (i.e., its deadline postponed).
- When a task experiences an overrun, only that task is delayed, so that the guarantee performed on the other tasks is preserved.

280

Implementation



281

Handling permanent overload

Performance Degradation

The load can be decreased not only by rejecting tasks, but also by reducing their performance requirements.

This can be done by:

- reducing precision of results
- skipping some jobs;
- relaxing timing constraints.

283

Reducing precision

In many applications, computation can be performed at different level of precision: the higher the precision, the longer the computation. Examples are:

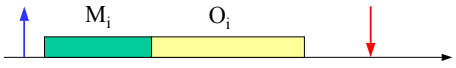
- binary search algorithms
- image processing and computer graphics
- neural learning

284

Imprecise computation

In this model, each task $\tau_i (C_i, D_i, w_i)$ is divided in two portions:

- a **mandatory** part: $\tau^m_i (M_i, D_i)$
- an **optional** part: $\tau^o_i (O_i, D_i)$



w_i is an importance weight

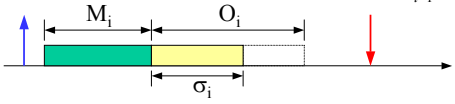
285

Imprecise computation

In this model, a schedule is said to be:

- **feasible**, if all mandatory parts complete in D_i
- **precise**, if also the optional parts are completed.

error: $\varepsilon_i = O_i - \sigma_i$ **average error**: $\varepsilon_a = \sum_{i=1}^{n} w_i \varepsilon_i$



GOAL: minimize the average error

286

Job skipping

Periodic load can also be reduced by skipping some jobs, once in a while.

Many systems tolerate skips, if they do not occur too often:

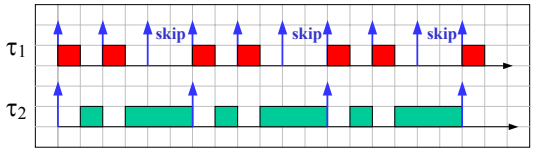
- multimedia systems (video reproduction)
- inertial systems (robots)
- monitoring systems (sporadic data loss)

287

Example

The system is overloaded, but tasks can be schedulable if t_1 skips one instance every 3:

$$U_p = \frac{1}{2} + \frac{4}{6} = 1.17 > 1$$



288

FIRM task model

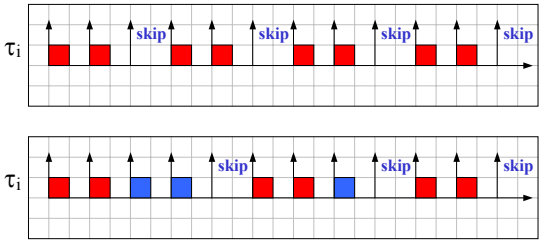
- Every job can either be executed within its deadline, or completely rejected (skipped).
- A percentage of task instances must be guaranteed off line to finish in time.
- Each task τ_i is described by (C_i, T_i, D_i, S_i):

  S_i is the minimum number of jobs that must be executed between two consecutive skips.

- Every instance can be **red** or **blue**:
  - **red** instances must finish within their deadline
  - **blue** instances can be aborted
- If a **blue** instance is aborted, the next S_i-1 instances must be **red**.
- If a **blue** instance is completed within its deadline, the next instance is still **blue**.
- The first S_i-1 instances of every task must be **red**.

Example

$C_i = 1 \quad T_i = 2 \quad D_i = 2 \quad S_i = 3$



Equivalent utilizazion factor

$$U_p^* = \max_{L \geq 0} \left\{ \frac{\sum_{i=1}^{n} g_i(0,L)}{L} \right\}$$

$$g_i(0,L) = \left(\left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_i S_i} \right\rfloor \right) C_i$$

Schedulability Analysis

A sufficient condition

Theorem: *A set of firm periodic tasks is schedulable if*

$$U_p^* \leq 1$$

A necessary condition

Theorem: *A set of firm periodic tasks is not schedulable if*

$$\sum_{i=1}^{n} \frac{C_i(S_i - 1)}{T_i S_i} > 1$$

NOTE: the sum represents the utilization of the computation that must take place.

Bandwidth saving

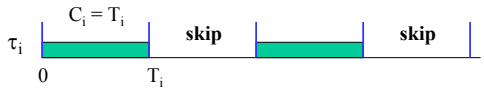
- In general, skipping jobs of periodic tasks causes a bandwidth saving:

$$\Delta U = U_p - U_p^*$$

- Such a bandwidth can be used for
  - improving aperiodic responsiveness (by increasing their reserved bandwidth);
  - accepting a larger number of periodic tasks.

295

Not always skips save bandwidth:



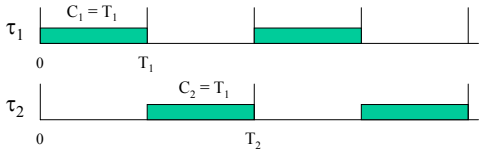
In this case: $U_p^* = 1$

In fact, for $L = T_i$ we have $g_i(0,L) = C_i = T_i$

Hence: $\frac{g_i(0,L)}{L} = \frac{T_i}{T_i} = 1$

296

However, notice that:



In this case we still have: $U_p^* = 1$

In fact: $g(0, T_1) = T_1$ e $g(0, T_2) = T_2$

Hence: $\frac{g(0,T_1)}{T_1} = \frac{g(0,T_2)}{T_2} = 1$

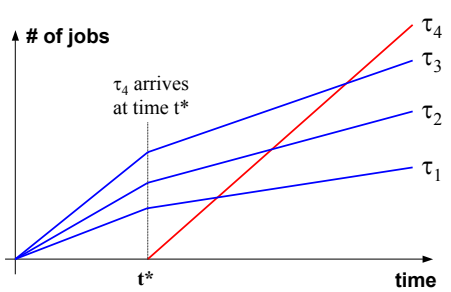
297

Relaxing timing constraints

- The idea is to reduce the load by increasing deadlines and/or periods.
- Each task must specify a range of values in which its period must be included.
- Periods are increased during overloads, and reduced when the overload is over.

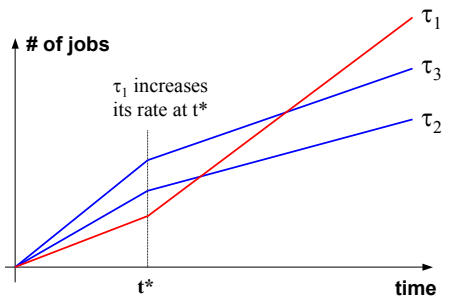
298

Overload handling due to a new arrival



299

Overload handling due to an increased rate



300

Example

task	C_i	T_{i0}	T_{min}	T_{max}
τ_1	10	20	20	25
τ_2	10	40	40	50
τ_3	15	70	35	80

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} = 0.96$$

301

Load adaptation

If τ_4 arrives with: $\mathbf{C_4 = 5,\ T_4 = 30}$ the system is not schedulable any more:

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} + \frac{5}{30} = 1.13$$

However, there exists a feasible schedule within the specified ranges:

$$U_p = \frac{10}{23} + \frac{10}{50} + \frac{15}{80} + \frac{5}{30} = 0.99$$

302

Elastic task model

- Tasks' utilizations are treated as elastic springs and can be changed by period variations.
- The resistance of a task to a period variation is controlled by an **elastic coefficient E_i**:

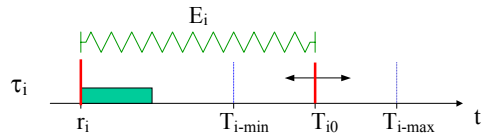
  ⇒ the greater E_i the greater the elasticity

303

Elastic task model

- A periodic task τ_i is characterized by:

  $(C_i, T_{i0}, T_{i\text{-}min}, T_{i\text{-}max}, E_i)$
- The actual period $T_i \in [T_{i\text{-}min}, T_{i\text{-}max}]$



304

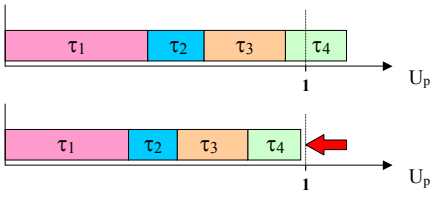
Special cases

- A task with $T_{min} = T_{max}$, is equivalent to a hard task.
- A task with $E_i = 0$ can intentionally change its period but does not allows the system to do that.

305

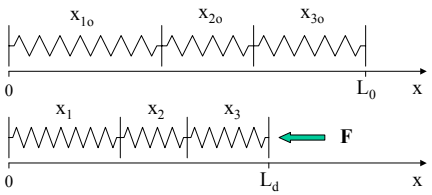
Compression algorithm

During overloads, utilizations must be compressed to bring the load below one.



306

The linear spring analogy



$$\begin{cases} F = k_1(x_{1o} - x_1) \\ F = k_2(x_{2o} - x_2) \\ F = k_3(x_{3o} - x_3) \end{cases} \qquad \begin{cases} x_1 + x_2 + x_3 = L_d \\ x_{1o} + x_{2o} + x_{3o} = L_0 \end{cases}$$

307

Solution without constraints

Summing the equations, we have:

$$F\left(\frac{1}{k_1}+\frac{1}{k_2}+\frac{1}{k_3}\right) = (x_{1o}+x_{2o}+x_{3o}) - (x_1+x_2+x_3)$$
$$= (L_0 - L_d)$$

That is:

$$F = \frac{(L_0 - L_d)}{\frac{1}{k_1}+\frac{1}{k_2}+\frac{1}{k_3}}$$

308

Solution without constraints

Substituting F in the equations, we have:

$$F = k_1(x_{1o} - x_1) = \frac{(L_0 - L_d)}{\frac{1}{k_1}+\frac{1}{k_2}+\frac{1}{k_3}}$$

That is:

$$x_1 = x_{1o} - (L_0 - L_d)\frac{1/k_1}{\frac{1}{k_1}+\frac{1}{k_2}+\frac{1}{k_3}}$$

309

Solution without constraints

$$x_i = x_{io} - (L_0 - L_d)\frac{K_{//}}{k_i} \qquad K_{//} = \frac{1}{\sum_{i=1}^{n}\frac{1}{k_i}}$$

And defining: $E_i = 1/k_i$

$$x_i = x_{io} - (L_0 - L_d)\frac{E_i}{E_s} \qquad E_s = \sum_{i=1}^{n} E_i$$

310

Period computation

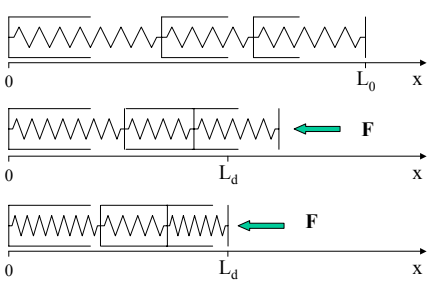
$$U_i = U_{io} - (U_0 - U_d)\frac{E_i}{E_s}$$

And then: $T_i = \frac{C_i}{U_i}$

311

Solution with constraints

Iterative solution:



312

Other use of elastic tasks

- Quickly find new period configurations during negotiation.
- Dynamically adjust the rates to fully utilize the processor.

Problem Understanding

- Real-Time applications are usually guaranteed based on worst-case execution times (WCETs).
- However, a precise WCET estimation is very difficult to achieve (due to interrupts, prefetch, cache, and DMA mechanisms).
- A wrong WCET estimate may cause the following problems:

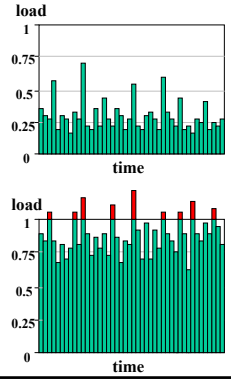
Predictability vs. efficiency

Over-estimation of WCETs

- high predictability
- low efficiency

Under-estimation of WCETs

- high efficiency
- low predictability



Adaptive solution

- The user does not specify any WCET.
- The system:
  - monitors tasks' execution times to estimate the actual load ($\mathbf{U_e}$);
  - adapts the task rates to keep the processor utilization close to a desired value ($\mathbf{U_d}$).

Task importance must be taken into account while performing rate adaptation.

Execution Time Estimator

- Monitored parameters (for each task t_i):

c_i current average execution time

C_i current worst-case execution time

- Building a prediction:

$$Q_i = c_i + k\,(C_i - c_i)$$

k = balancing factor

Load as a feedback