Embedded Systems - Real-Time Scheduling

part 6

Exclusive access to shared resources

The problem of accessing shared resources and the resulting blocking
Priority inversion as a consequence of blocking
Basic techniques to grant exclusive access to shared resources
Some specific synchronization protocols
Priority Inheritance Protocol – PIP
Priority Ceiling Protocol – PCP
Stack Resource Protocol- SRP

Previous lecture – On-line scheduling of periodic tasks

Fixed priorities scheduling

The Rate-Monotonic criterion – Optimality – CPU utilization upper bound

The Deadline-Monotonic and arbitrary fixed priorities criteria
– worst-case response time analysis

Dynamic priorities scheduling

The Earliest Deadline First criterion – Optimality – CPU utilization upper bound

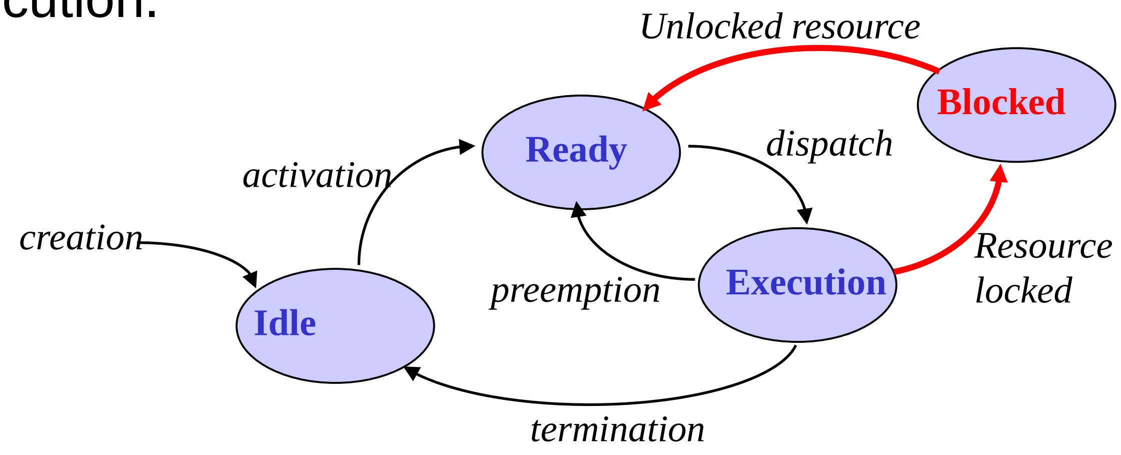
Earliest Deadline First *versus* Rate-Monotonic scheduling

Other dynamic priorities criteria – *Least Slack First, First Come First Served*

Shared resources with exclusive access

Tasks: the blocking state

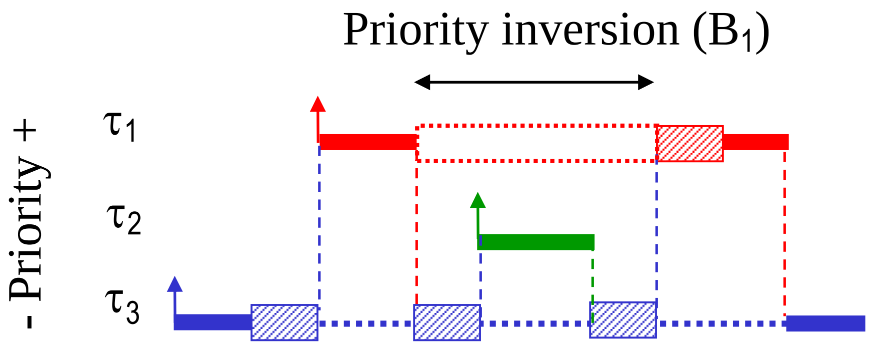
When a task that is executing tries to access a shared resource (e.g. a communication port, a buffer in memory, generally a **critical region**) that is locked by a ready task (necessarily with lower priority) the former task is said to be **blocked**. When the resource is unlocked, the blocked task becomes again ready for execution.



The priority inversion phenomenon

In a real-time system with preemption and **independent** tasks, when a **task executes** it means it has the **highest priority** at that instant.

However, when tasks can access shared resources in exclusive mode, the situation changes. When the executing task becomes blocked, the **task that blocks has lower priority**. When the blocking task executes (and any other of intermediate priority) there is a **priority inversion**.



The priority inversion phenomenon

The **priority inversion** is an inevitable phenomenon in the presence of exclusive access to shared resources (intrinsic to blocking)

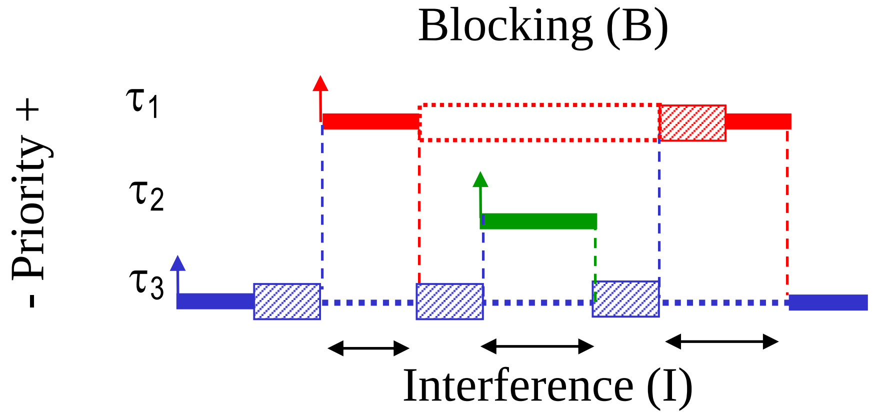
However, it is fundamental to **limit and quantify its impact** in the worst-case so that the **schedulability** of the task set can analyzed.

Thus, the techniques used to grant exclusive access to each resource (**synchronization primitives**) should allow bounding the period of priority inversion and be analyzable, i.e., allow **quantifying the worst-case blocking** that each task can suffer.

Accounting for the priority inversion

Once the blocking that a task can suffer is upper bounded **(B)**, the most common way to account for it in the schedulability analysis is to consider that the **task executes for a longer time** **(C+B)**

Note the difference between **blocking**, which is added once per instance, and **interference,** which can occur multiple times per instance.



Techniques to control accessing shared resources

Synchronization primitives

Global mechanisms (affect all tasks)

- **Interrupts disabling** (*disable / enable* ou *cli / sti*)
- **Preemption disabling** (*no_preemp / preempt*)

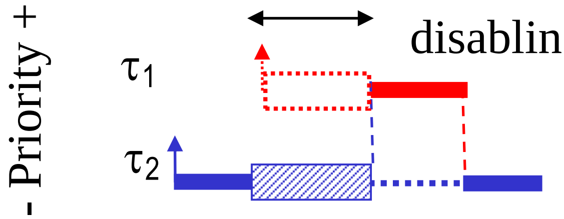
Mechanisms with reduced scope (affect just a subset of tasks)

- Use of ***locks*** or ***atomic flags*** (*mutexes* – although this expression is also often used to refer to semaphores – *lock / unlock*)
- Use of **semaphores** (counter+list – *P / V* or *wait / signal*)

Global control techniques

Interrupts disabling

- **All activities** in the system **are blocked**! (not only other tasks, independently of using a shared resource or not, but also interrupt servicing including the system tick).
- This technique is very easy to apply but should only be used with very short critical regions (e.g., access to a variable)
- Each task can only be blocked once and for the duration of the **largest critical region** among the tasks of lower priority (or shorter relative deadline for EDF) even when no resources are used !!



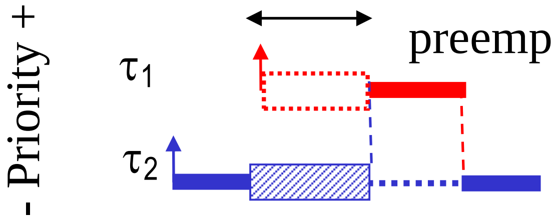
Blocking due to interrupt disabling (B_1)

e.g.	S_1	S_2	S_3
τ_1	**1**	**2**	**0**
τ_2	**0**	**9**	**3**
τ_3	**8**	**7**	**0**
τ_4	**6**	**5**	**4**

Global control techniques

Preemption disabling

- **All tasks** in the system **get blocked**! (interrupt servicing, including the tick, is unaffected)
- Easy to **implement** but needs to be at the **kernel level** (still causes unnecessary blocking)
- Each task can only be **blocked once** and for the duration of the **longer critical region** among the tasks of lower priority (or shorter relative deadline for EDF) even when no resources are used !!



Blocking due to preemption disabling (B_1)

e.g.	S_1	S_2	S_3
τ_1	**1**	**2**	**0**
τ_2	**0**	**9**	**3**
τ_3	**8**	**7**	**0**
τ_4	**6**	**5**	**4**

Impact in schedulability (fixed priorities)

Schedulability analysis (RM)

1. $\forall_{1\le i\le n}\ \sum_{k=1}^{i}\frac{C_k}{T_k}+\frac{\boldsymbol{B_i}}{\boldsymbol{T_i}}\ \le i\left(2^{\frac{1}{i}}-1\right)$

2. $\sum_{i=1}^{n}\frac{C_i}{T_i}+\max_{\boldsymbol{i}}\left(\frac{\boldsymbol{B_i}}{\boldsymbol{T_i}}\right)\ \le n\left(2^{\frac{1}{n}}-1\right)$

3. $Rwc_i=C_i+\boldsymbol{B}_i+\sum_{k=1}^{i-1}\left\lceil\frac{Rwc_i}{T_k}\right\rceil * C_k$

e.g.	C_i	T_i	B_i
τ_1	**5**	**30**	**9**
τ_2	**15**	**60**	**8**
τ_3	**20**	**80**	**6**
τ_4	**20**	**100**	**0**

←

e.g.	S_1	S_2	S_3
τ_1	**1**	**2**	**0**
τ_2	**0**	**9**	**3**
τ_3	**8**	**7**	**0**
τ_4	**6**	**5**	**4**

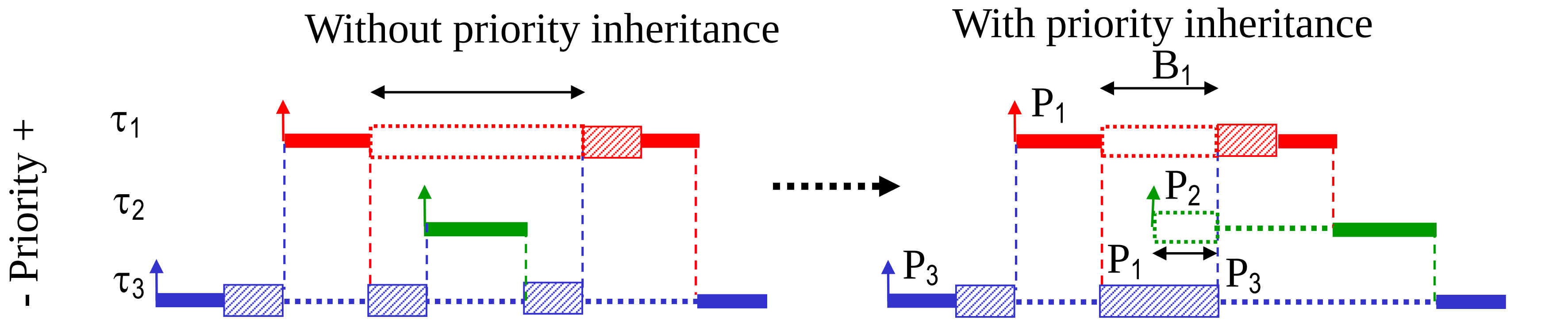
Techniques to control accessing shared resources

Using locks or semaphores

- Generally, they affect only the tasks involved in sharing resources.
- Harder implementation and at the kernel level but more efficient (because of the above)
- However, the duration of the blockings is highly dependent on the **specific protocol** used to **manage the semaphores**
- In this case, it is particularly important that the protocol allows **avoiding**:
    - **Undetermined blocking**
    - **Chained blocking**
    - **Deadlocks**

PIP – Priority Inheritance Protocol

- The blocking task (lower priority) **inherits the priority** of the blocked task (higher priority).
- **Limits the duration of the blocking periods** by avoiding that tasks with intermediate priority execute while the blocking task (with lower priority) is actually blocking a higher priority task.



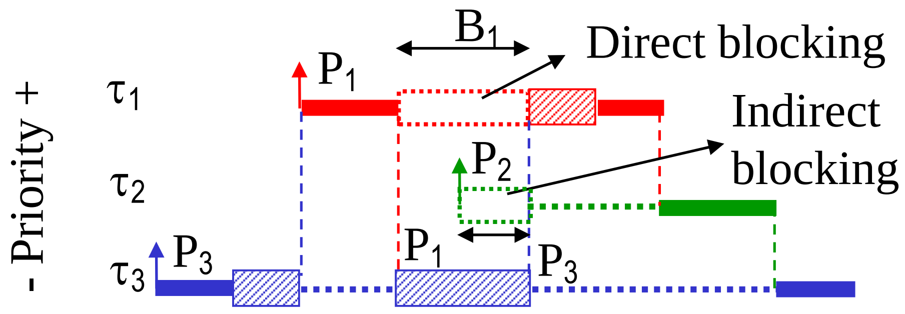
PIP – Priority Inheritance Protocol

- To bound the **blocking time** (B) note that a task can be blocked by any task with **lower priority**:

  1. With which it shares a resource (direct) or
  2. That can block a task with higher priority (indirect).

- Note further that, in the absence of nested resource accesses:

  1. Each task can only block another task once
  2. Each task can only be blocked once in each semaphore



e.g.	S_1	S_2	S_3
τ_1	**1**	**2**	**0**
τ_2	**0**	**9**	**3**
τ_3	**8**	**7**	**0**
τ_4	**6**	**5**	**4**

PIP – Priority Inheritance Protocol

Schedulability analysis (RM)

1. $\forall_{1\le i\le n} \sum_{k=1}^{i}\frac{C_k}{T_k}+\frac{\boldsymbol{B_i}}{\boldsymbol{T_i}} \le i\left(2^{\frac{1}{i}}-1\right)$

*(just computing **B$_i$** varies)*

2. $\sum_{i=1}^{n}\frac{C_i}{T_i}+\max_{i}\left(\frac{\boldsymbol{B_i}}{\boldsymbol{T_i}}\right) \le n\left(2^{\frac{1}{n}}-1\right)$

3. $Rwc_i = C_i + \boldsymbol{B}_i + \sum_{k=1}^{i-1}\left\lceil\frac{Rwc_i}{T_k}\right\rceil * C_k$

e.g.	C_i	T_i	B_i
τ_1	**5**	**30**	**17**
τ_2	**15**	**60**	**13**
τ_3	**20**	**80**	**6**
τ_4	**20**	**100**	**0**

←

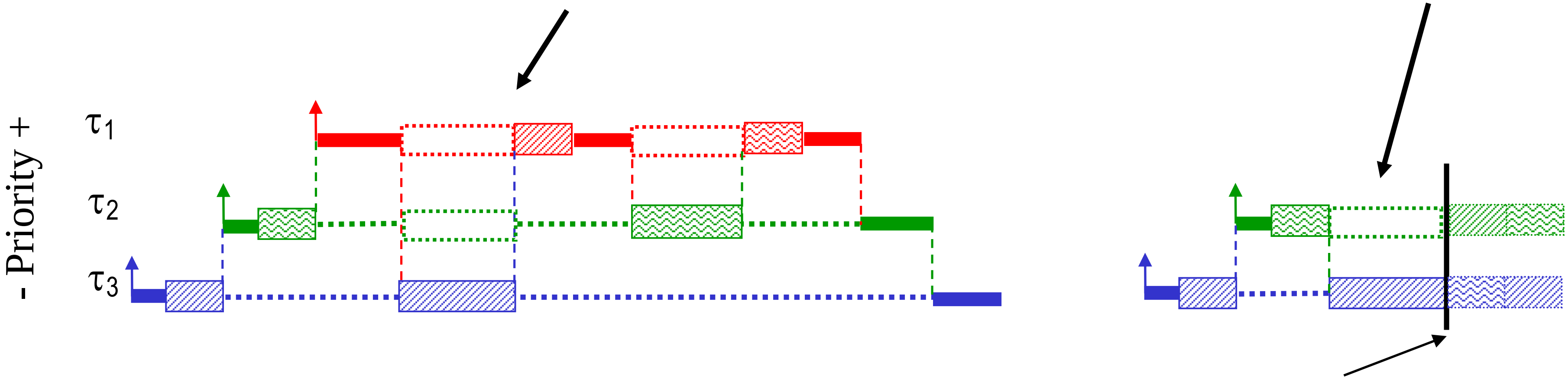
e.g.	S_1	S_2	S_3
τ_1	**1**	**2**	**0**
τ_2	**0**	**9**	**3**
τ_3	**8**	**7**	**0**
τ_4	**6**	**5**	**4**

PIP – Priority Inheritance Protocol

+ Relatively easy to implement (requires an extra field in the TCB, i.e., the inherited priority)

+ It is transparent for the programmer (each task uses local info, only)

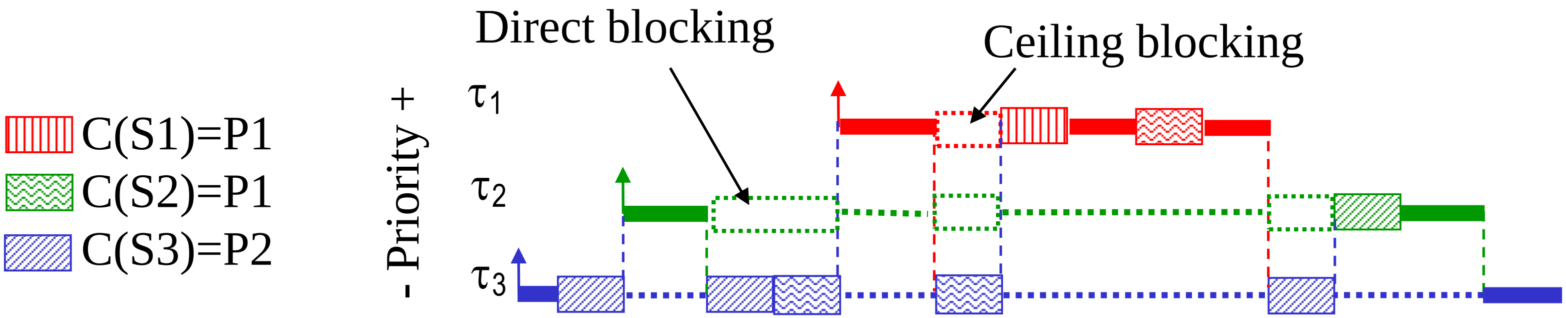
– Suffers from **chained blocking** and, mainly, is ***not deadlock-free***



Deadlock due to nesting in resources access.

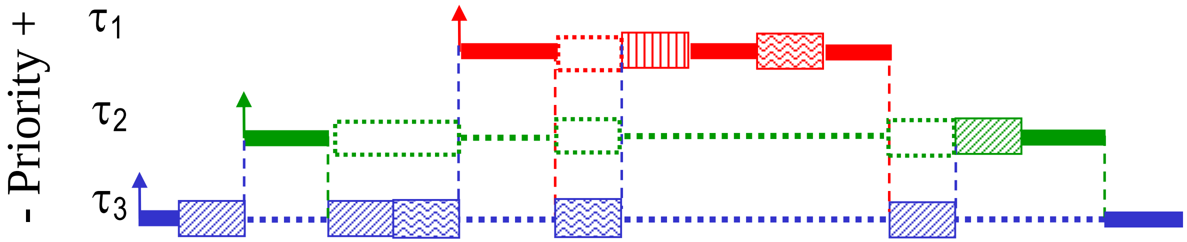
PCP – Priority Ceiling Protocol

- **Extension to PIP** but with an extra rule to control the **access to free semaphores** (to assure that all needed semaphores are free)
- A **priority ceiling** is defined for each semaphore taking the value of the highest priority among the tasks that use it.
- A task can **acquire a semaphore** if it is **free** **and** its **priority is higher than the ceilings of all** semaphores currently locked by other tasks.



PCP – Priority Ceiling Protocol

- PCP allows a task **acquiring** a semaphore if it is **free** and only when **all the remaining semaphores** that that task might need **are free**.
- Thus, each task can be **blocked only once**.
- To bound the **blocking time** (B) note that a task can be blocked by any other task with **lower priority** that uses the same semaphore or that uses a semaphore which **ceiling is at least equal to its priority**



e.g.	S_1	S_2	S_3
τ_1	**1**	**2**	**0**
τ_2	**0**	**9**	**3**
τ_3	**8**	**7**	**0**
τ_4	**6**	**5**	**4**

PCP – Priority Ceiling Protocol

Schedulability analysis (RM)

*(just computing **B$_i$** varies)*

Note: in this example, B values are equal to those of the global mechanisms. This is a coincidence. With PCP there could be higher priority tasks with B=0.

1. $\forall_{1\le i\le n}\ \sum_{k=1}^{i}\frac{C_k}{T_k}+\frac{\boldsymbol{B_i}}{\boldsymbol{T_i}}\le i\left(2^{\frac{1}{i}}-1\right)$

2. $\sum_{i=1}^{n}\frac{C_i}{T_i}+\boldsymbol{\max_i}\left(\frac{\boldsymbol{B_i}}{\boldsymbol{T_i}}\right)\le n\left(2^{\frac{1}{n}}-1\right)$

3. $Rwc_i=C_i+\boldsymbol{B}_i+\sum_{k=1}^{i-1}\left\lceil\frac{Rwc_i}{T_k}\right\rceil * C_k$

e.g.	C$_i$	T$_i$	B$_i$
τ_1	5	30	9
τ_2	15	60	8
τ_3	20	80	6
τ_4	20	100	0

e.g.	S$_1$	S$_2$	S$_3$
τ_1	1	2	0
τ_2	0	9	3
τ_3	8	7	0
τ_4	6	5	4

PCP – Priority Ceiling Protocol

+ Shorter blockings than with PIP, **free from chained blocking**, ***deadlock-free***,

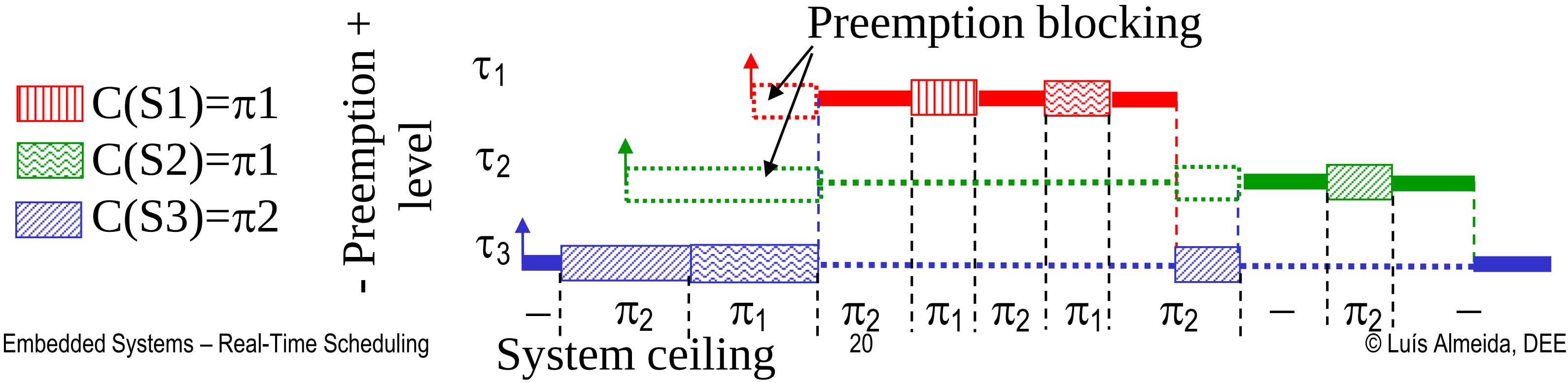
– Harder to implement (in the TCB, a new field is needed to hold the inherited priority and another one for the semaphore in which the task is blocked – to facilitate inheritance transitivity. Moreover, a new structure is needed to hold the current state of the semaphores with the respective ceilings and id of the task that is currently using them – to facilitate inheritance)

– It is not transparent for the programmer (the semaphore ceilings are not local to the tasks)

(There is also an **EDF version** in which the blocking tasks inherit the deadline of the blocked tasks and the semaphore ceilings use the relative deadlines (*preemption level*))

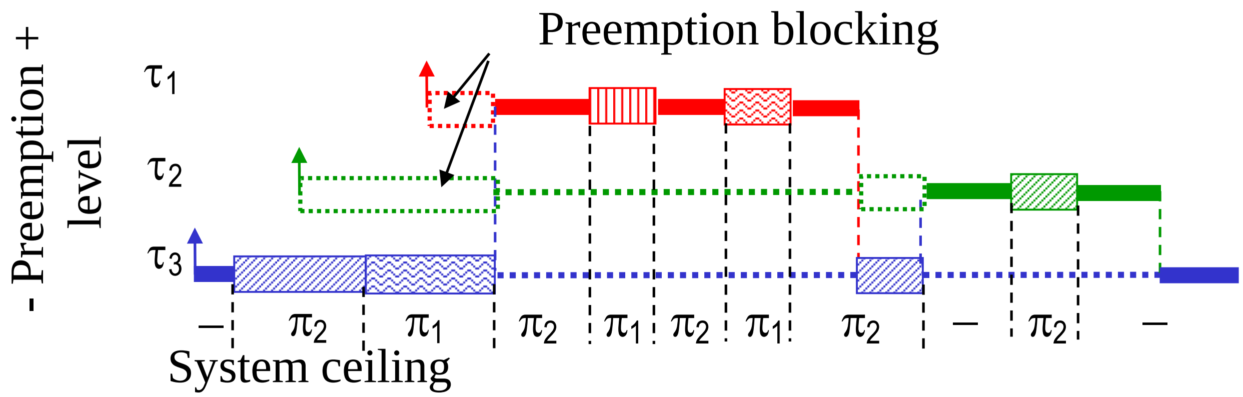
SRP – Stack Resource Policy

- Similar to PCP but with a rule on the actual **execution release** (which guarantees that all needed semaphores are free)
- Similarly based on the concept of **priority ceiling.**
- Defines the **preemption level** (π) as the capacity of a task to cause preemption to other (it is a static parameter).
- A task can **start execution** only if its **preemption level is higher than that of the executing task** and **higher than the ceilings** of all currently locked semaphores (**system ceiling**).



SRP – Stack Resource Policy

- SRP allows a task to start execution only when all the resources it might need are free.
- The **blocking upper bound** (B) is similar to that of PCP, just occurs in a different moment (task release).
- Each task can be blocked just once by any task with **lower preemption level** that uses a semaphore whose **ceiling is at least equal to its own preemption level.**



e.g.	S_1	S_2	S_3
τ_1	**1**	**2**	**0**
τ_2	**0**	**9**	**3**
τ_3	**8**	**7**	**0**
τ_4	**6**	**5**	**4**

SRP – Stack Resource Policy

Schedulability analysis (RM)

*(**B** similar to PCP, but SRP is also intrinsically prepared for EDF)*

1. $\forall_{1\le i\le n}\ \sum_{k=1}^{i}\frac{C_k}{T_k}+\frac{\boldsymbol{B_i}}{\boldsymbol{T_i}} \le i\left(2^{\frac{1}{i}}-1\right)$

2. $\sum_{i=1}^{n}\frac{C_i}{T_i}+\max_{\boldsymbol{i}}\left(\frac{\boldsymbol{B_i}}{\boldsymbol{T_i}}\right) \le n\left(2^{\frac{1}{n}}-1\right)$

3. $Rwc_i = C_i + \boldsymbol{B}_i + \sum_{k=1}^{i-1}\left\lceil\frac{Rwc_i}{T_k}\right\rceil * C_k$

Schedulability analysis (EDF)

1. $\forall_{1\le i\le n}\ \sum_{k=1}^{i}\frac{C_k}{T_k}+\frac{\boldsymbol{B_i}}{\boldsymbol{T_i}} \le 1$

2. $\sum_{i=1}^{n}\frac{C_i}{T_i}+\max_{\boldsymbol{i}}\left(\frac{\boldsymbol{B_i}}{\boldsymbol{T_i}}\right) \le 1$

e.g.	C_i	T_i	B_i
τ_1	5	30	9
τ_2	15	60	8
τ_3	20	80	6
τ_4	20	100	0

←

e.g.	S_1	S_2	S_3
τ_1	1	2	0
τ_2	0	9	3
τ_3	8	7	0
τ_4	6	5	4

SRP – Stack Resource Policy

+ Shorter blockings than with PIP, **free from chained blocking**, ***deadlock-free***,

+ **Less preemptions** than with PCP, intrinsically adapted to **fixed or dynamic priorities,** and to resources with multiple units (e.g., an array of buffers)

+ Since there are no blockings during task execution, tasks **do not need an individual stack** (a single stack can be used for all tasks leading to substantially lower memory requirements)

– Harder to implement (preemption test is more complex, needs keeping the system ceiling, and worse if using resources with multiple units)

– Not transparent to the programmer (semaphore ceilings...)

Wrapping up

- **Exclusive access** to shared resources: the **blocking**

- **Priority inversion (blocking)**: need to bound it and analyze it

- **Basic techniques** to synchronize the access to shared resources
  - Interrupt disabling and preemption disabling (**global scope**)

- Techniques based on **semaphores** (reduced scope)

  - ***Priority Inheritance Protocol – PIP***

  - ***Priority Ceiling Protocol – PCP***

  - ***Stack Resource Protocol- SRP***

Embedded Systems - Real-Time Scheduling

part 7

Scheduling aperiodic tasks

Running together periodic and aperiodic tasks

Using servers to run aperiodic tasks

Servers with fixed priorities

Servers with dynamic priorities

Putting together periodic and aperiodic tasks

Periodic tasks

Adequate, for example, to situations in which it is necessary to keep track of physical entities, normally continuous, or produce regularly a certain value or actuation. Frequently without control of relative phase (offset) → Worst-case sporadic behavior

Aperiodic tasks

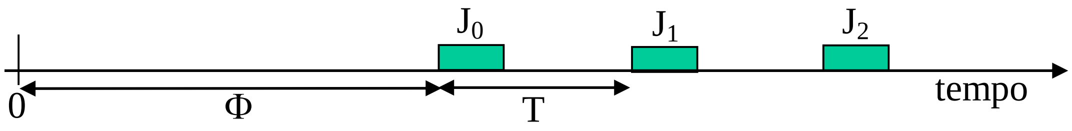
Adequate to situations in which we cannot bound the minimum interval (or it is too short) between consecutive activations, such as alarms, operator interfaces, or other asynchronous events.

Hybrid systems

Applications composed of periodic/sporadic and aperiodic tasks. It is the most common case in real applications.

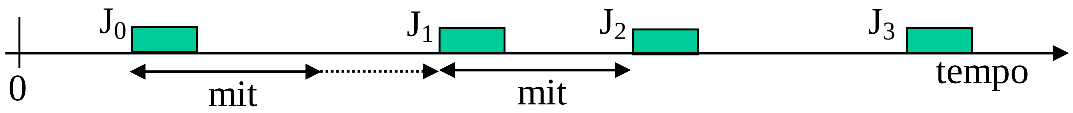
Putting together periodic and aperiodic tasks

- **Periodic tasks**



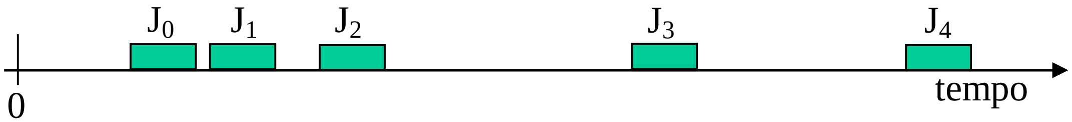
nth instance activated at $a_n = n*T+\Phi$ (well known worst-case)

- **Sporadic tasks**



In the worst-case, reverts to a periodic task with períod = *mit*

- **Aperiodic tasks**



Characterized stocastically, only

  - How to **limit interference** over periodic/sporadic tasks?
  - How to provide the **best Quality-of-Service** possible?

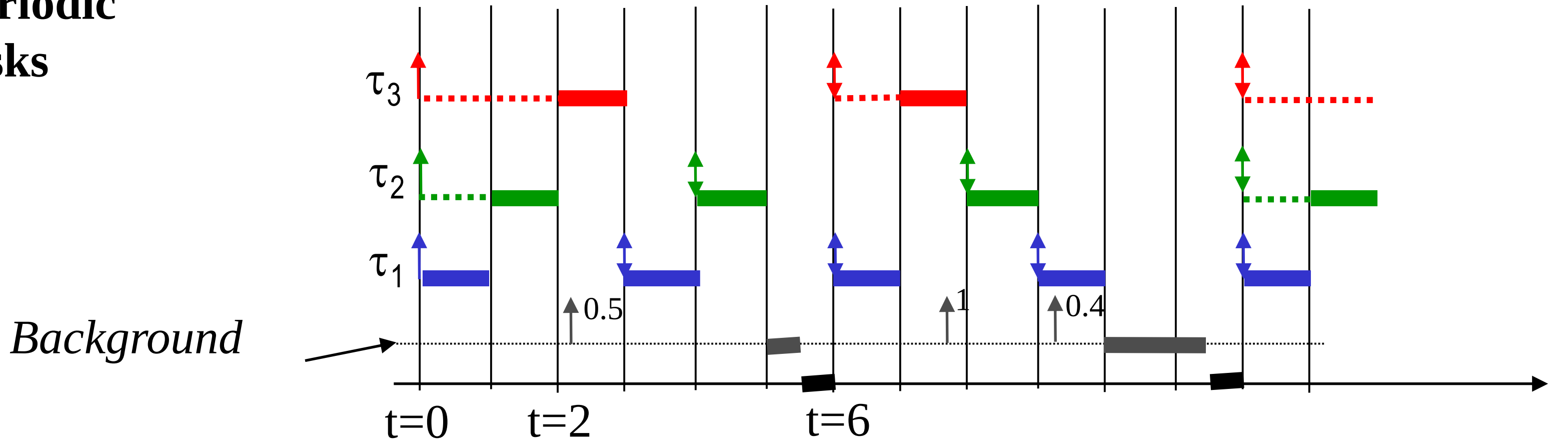
Execution in the background

A common and simple way to combine both types of tasks consists in giving **absolute priority** to **periodic tasks** and execute **aperiodic ones** in the intervals of CPU time **left free by the periodic subsystem.** (note: in these slides, by “periodic” we mean periodic/sporadic)

We say that the aperiodic tasks are **executed in the *background*** (lowest priority level in the system)

Periodic tasks

τ_i	T_i	C_i
1	3	1
2	4	1
3	6	1

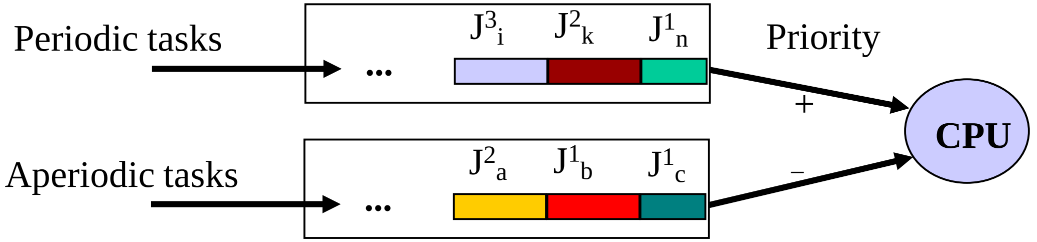


Execution in the background

Executing aperiodics in the background is **easy to implement** and **does not interfere with the periodic subsystem** (might cause small interference due to interrupt service routines)

Conversely, aperiodic tasks can suffer **large service delays** depending on the periodic load (can be computed considering aperiodic tasks as the lowest priority ones)

Poor performance for **real-time aperiodic tasks (alarms)** but **suitable for non-real-time aperiodic tasks (file transfers)**.

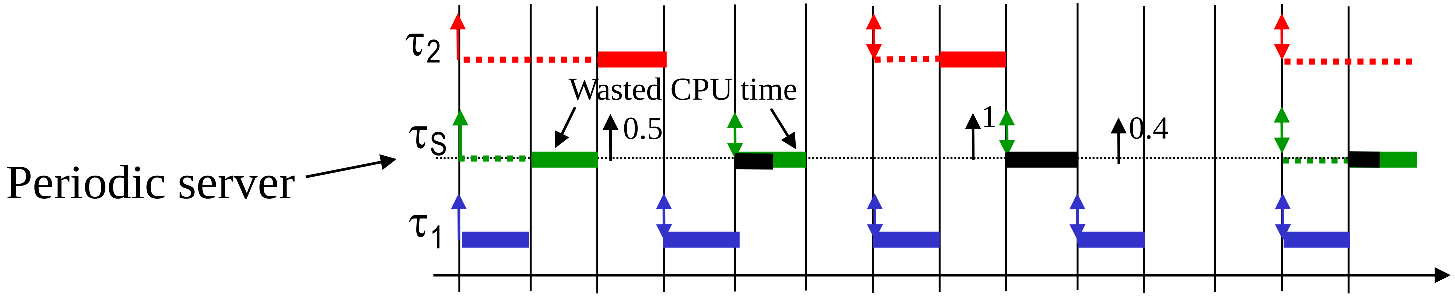


Using servers to run aperiodic tasks

One way to **improve the service** to aperiodic tasks (when background execution is not enough to meet the real-time constraints) consists in using a special **periodic task** whose purpose is just to **execute active aperiodic tasks**.

This task is called **periodic server** and it is characterized by a period T_S and a capacity C_S

This way, it is possible to insert the server in the periodic subsystem with the **required priority level** to obtain the desired service level.



Using servers to run aperiodic tasks

There are **several types** of servers to run aperiodic tasks, either with **fixed or dynamic priorities**, that vary in terms of:

- **Impact** on the schedulability of the periodic subsystem
- **Average response time** to aperiodic execution requests
- Computing and memory **overhead** and implementation cost.

Examples

- **Fixed priorities:** Periodic Server, polling server, deferrable server, priority exchange server, sporadic server, ...
- **Dynamic priorities:** adapted versions of the fixed priority servers, total bandwidth server (TBS), constant BW server (CBS), ...

Worst-case response time to aperiodic requests

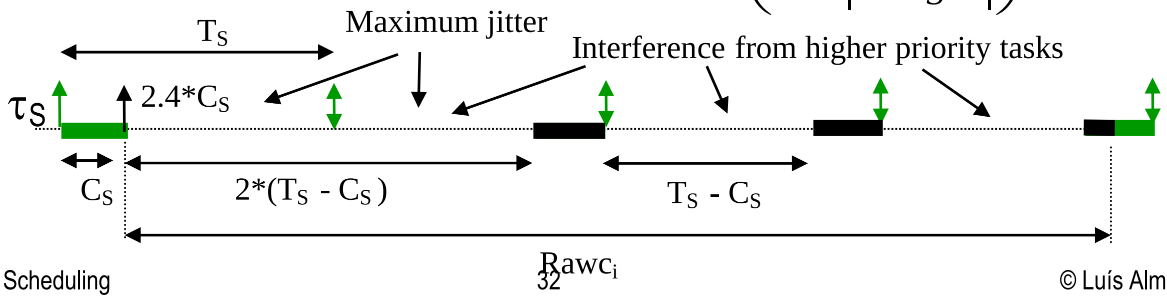
Worst-case service to aperiodic requests (tasks)

(similar for all servers that can be modeled as a periodic task under full load)

Consider that:

- The server is a periodic task τ_S (C_S,T_S)
- Suffers maximum jitter at the instant of the aperiodic request
- Suffers maximum interference delay in all subsequent instances

$$Rawc_i = Ca_i + (T_S - C_S) * \left(1 + \left\lceil \frac{Ca_i}{C_S} \right\rceil\right)$$



Worst-case response time to aperiodic requests

Worst-case service to aperiodic requests

(cont.)

If for the same server there are *Na* aperiodic requests queued waiting for service (scheduled according to a certain criterion – consistent with index *k*), the schedulability test for aperiodic task/request *i* is:

(consider that all *i*-1 previous aperiodic requests are queued for execution, e.g., arrived in a burst just before, when request *i* arrives)

$$\forall_{i=1..Na} \; Rawc_i = Ca_i + (T_S - C_S) * \left(1 + \left\lceil \frac{\sum_{k=1}^{i} Ca_k}{C_S} \right\rceil \right) < Da_i$$

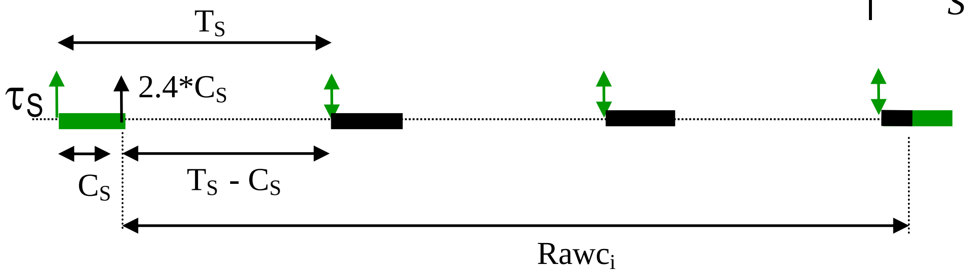
Worst-case response time to aperiodic requests

Worst-case service to aperiodic requests

(cont.)

If the server has the highest priority in a fixed priorities system (or in time-triggered slotted systems such as TDMA) there is no more the interference delay that affects the subsequent instances, and the worst-case response time is given by:

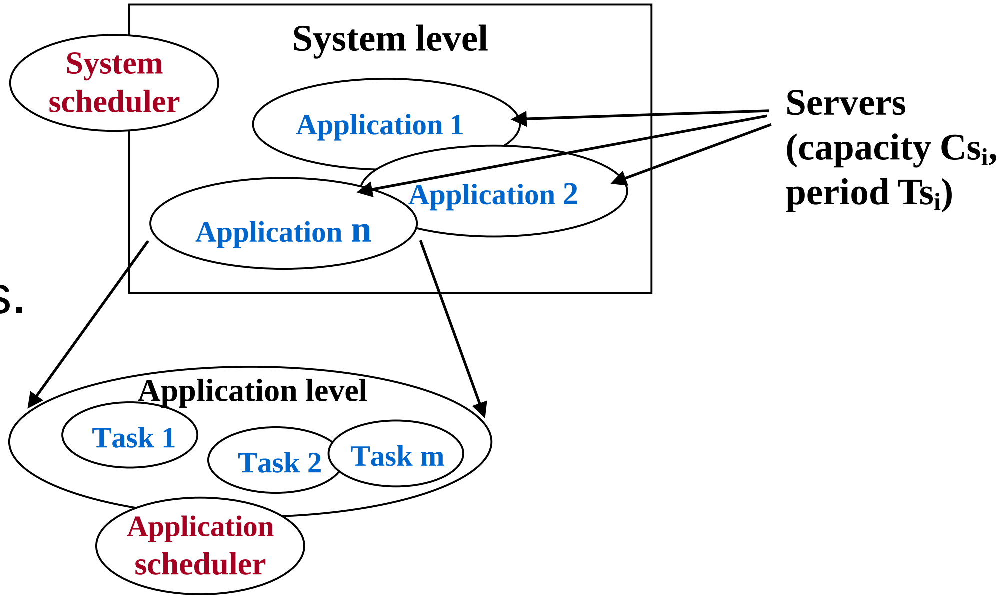
$$Rawc_i = Ca_i + (T_S - C_S) * \left\lceil \frac{Ca_i}{C_S} \right\rceil$$



Hierarchical scheduling

Taking a more general perspective, we can think of a system that runs **different applications** that must be guaranteed independently. One way to **bound the mutual interference** across applications is to **run each application within a server**.

Thus, we get two levels, the **system level** that manages the servers and the **server level** that manages the application specific tasks.



Hierarchical scheduling

In a **hierarchical framework** we have two problems:

- (**application schedulability**) Given a server and an application, is the application schedulable inside the server?
  - e.g., using the worst-case response time analysis within a server (previous slides), analyzing whether
    application demand ≤ server supply
- (**server design**) Given an application, which is the server requiring the minimum CPU that allows meeting the application requirements?
  - e.g., using an optimization technique (considering server context switch cost, otherwise Ts →0) determine
    min(server supply): supply ≥ application demand

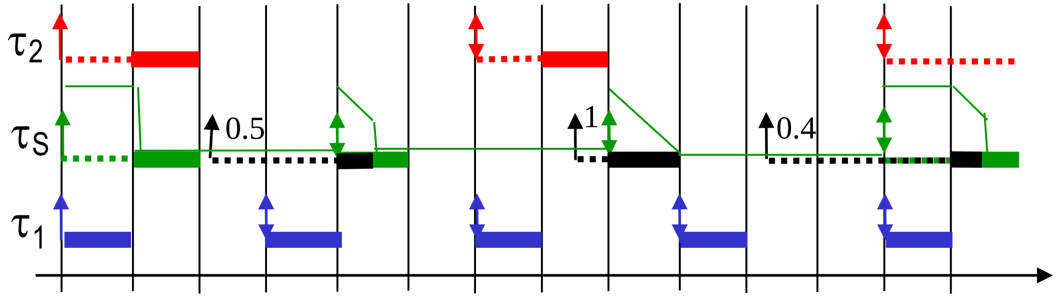
Polling server - PS

This **fixed priorities server** improves the periodic server by transferring unused capacity to lower priority tasks

It is still equivalent to executing a periodic task. **Aperiodic requests** are executed **during the intervals of time assigned to the server** by the periodic tasks scheduler.

Note that, even if there are no requests to execute, the server capacity is used up anyway. The capacity is replenished every period.

Polling server (1,4)



Polling server - PS

Implementing a polling server is **rather simple**, requiring a single queue for the aperiodic requests and a control of the capacity used thus far.

The **average response time** to the aperiodic requests is **improved** with respect to background execution because it is possible to execute the aperiodic tasks at a higher priority level. However, there are still intervals of unavailability corresponding to the server period.

The **impact** on the schedulability of the periodic subsystem is equivalent to that of the corresponding virtual periodic task. For example, using RM + PS

$$U_p(n \text{ periodic tasks}) + U_S \leq (n+1)\left(2^{1/(n+1)} - 1\right)$$

Polling server - PS

Note: The Liu & Layland's least upper bound was determined independently of the tasks utilizations. However, **fixing the utilization** of the task with the **highest priority** it is possible to improve the bound.

Thus, assigning the **highest priority to the server** and a utilization of **$U_S=C_S/T_S$** , the Liu&Layland's bound becomes:

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

and

$$U_p + U_S \xrightarrow[n\to\infty]{} U_S + \ln\left(\frac{2}{U_S+1}\right)$$

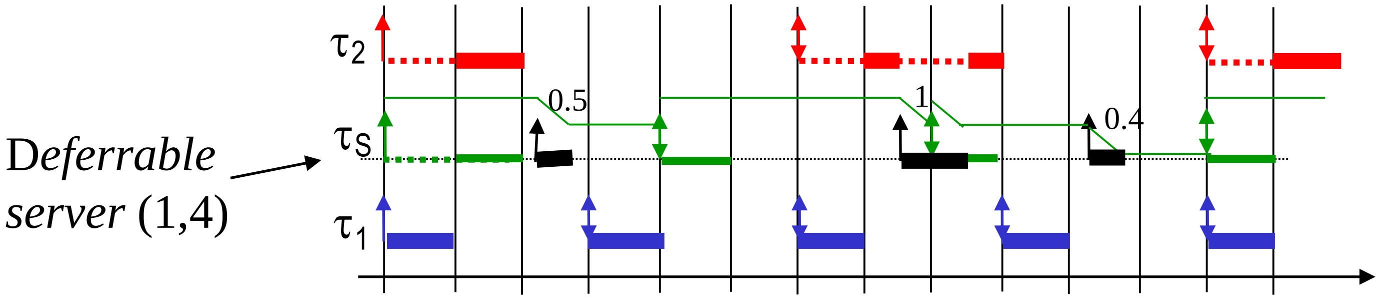
(Check the book of Giorgio Buttazzo)

Deferrable server - DS

This **fixed priorities server** improves the polling server by **maintaining the unused capacity available** until of the period

Therefore, an aperiodic request can get immediate service if the server is active and still has capacity in that period.

The server capacity is fully replenished every server period.



Deferrable server - DS

The **complexity** to implement a deferrable server is similar to that of a Polling Server.

The **average response time** to the aperiodic requests is **improved** with respect to the PS because it is possible to use the server capacity even after the intervals assigned by the periodic scheduler, thus eliminating the unavailability periods of the PS between successive replenishments.

Nevertheless, this server has a **negative impact** on the schedulability of the periodic subsystem due to the possible deferred execution. With RM+DS and assigning the server the highest priority we get:

$$U_p + U_S \le U_S + n\left(\left(\frac{U_S + 2}{2U_S + 1}\right)^{1/n} - 1\right)$$

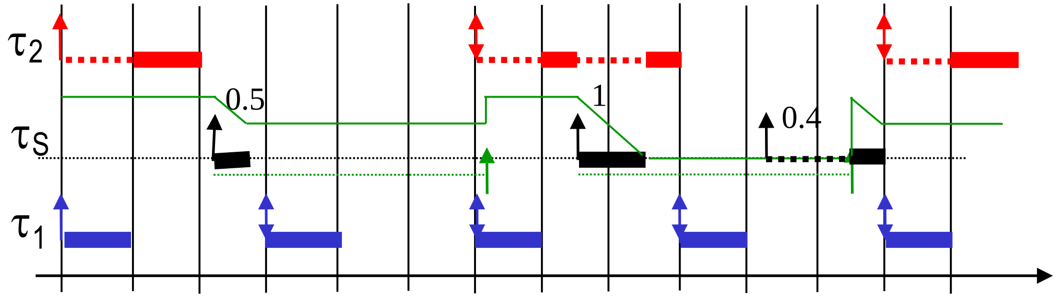
(Check the book of Giorgio Buttazzo)

Sporadic server - SS

This **fixed priorities server** improves the deferrable server by enforcing **a sporadic** (instead of periodic) **execution**

It does not **penalize schedulability** of the periodic subsystem at the expense of a **slightly longer response time** to aperiodic requests.

In this case, if the **execution** of the server **is deferred so is the replenishment** to enforce the **server bandwidth**. Basically, the amount consumed is always replenished a period later.



Sporadic server (1,4)

Sporadic server - SS

The complexity of implementing a sporadic server **is higher** than for a DS given the need to maintain a structure with the replenishment instants and the amounts to replenish.

The **average response time** to the aperiodic requests is **similar** to that of a DS.

The **impact** on the schedulability of the periodic subsystem is similar to that of a normal periodic task, similarly to the PS, due to the bandwidth enforcement (sporadic behavior).

Using RM+SS and assigning the highest priority to the server we get

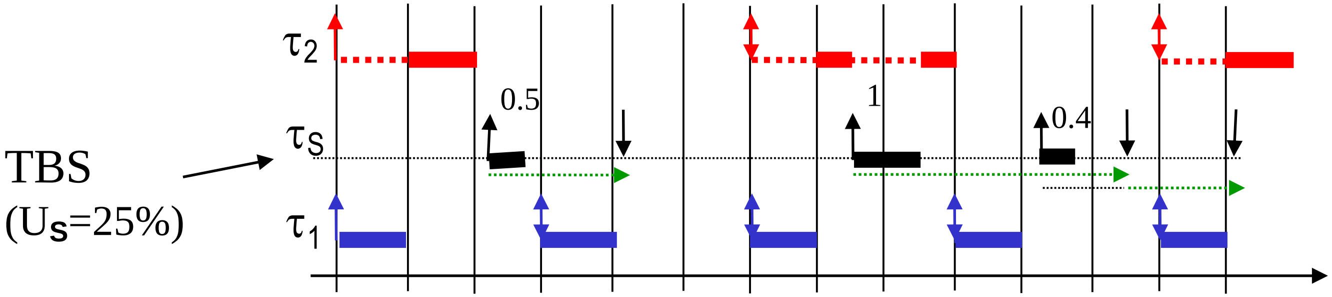
$$U_p + U_S \le U_S + n\left(\left(\frac{2}{U_S+1}\right)^{1/n} - 1\right)$$

Total bandwidth server - TBS

The Total Bandwidth Server (TBS) is a **dynamic priorities server** to handle aperiodic requests in an EDF system, **using a given CPU bandwidth independently of the arrival pattern**, i.e., it bounds the impact of the aperiodic requests on the periodic subsystem.

When a new request arrives at instant r_k, a deadline d_k, is assigned

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



Total bandwidth server - TBS

The **complexity** of implementing a TBS is relatively small since it is only necessary to compute a deadline every time a request arrives and then it is inserted in the ready tasks queue, together with the periodic tasks.

The **average response time** to aperiodic requests is **shorter** than with dynamic priorities versions of the previous servers.

The **impact** on the schedulability of the periodic subsystem is similar to that of a periodic task with the same bandwidth as that of the server. Using EDF+TBS

$$U_p + U_S \leq 1$$

The TBS is **vulnerable to overruns** since there is **no control over the actual execution time** (no capacity control).

Constant bandwidth server - CBS

The Constant Bandwidth Server (CBS) is a **dynamic priorities server** designed to solve the robustness problem of the TBS enforcing **bandwidth isolation**.

This is achieved using a **capacity management scheme** (Q_S ,T_S).

When a request arrives at instant r_k, and the remaining server capacity at that instant is $c_S(r_k)$, the deadline d_k is computed as:

if $r_k + c_S(r_k)/U_S < d_{k-1}$ **then** $d_k = d_{k-1}$

or else $d_k = r_k + T_S$ and $c_S(r_k) = Q_S$

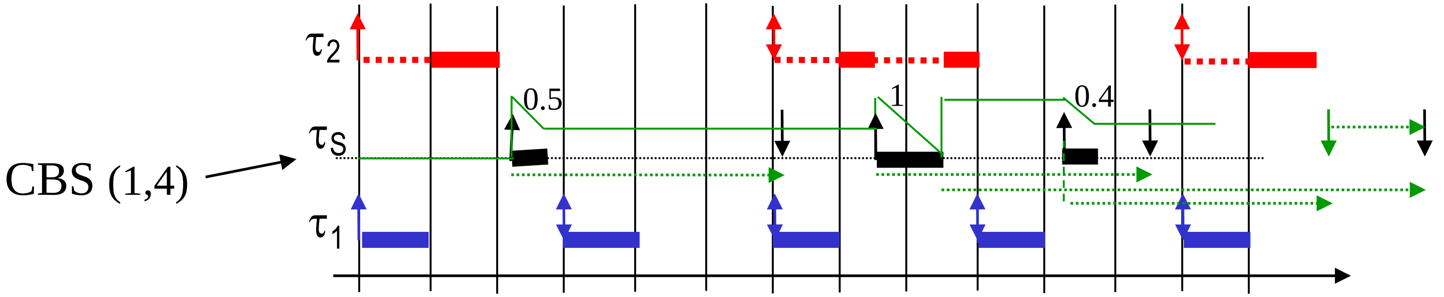
When the server capacity is exhausted (c_S=0) it is immediately replenished but d_k is postponed to enforce the server bandwidth

$$d_k = d_k + T_S \quad \text{and} \quad c_S = Q_S$$

Constant bandwidth server - CBS

The deadline assignment scheme used by CBS **enforces the server bandwidth** independently of the aperiodic arrival pattern and the **load that the server actually executes**.

If a task executing inside a CBS **executes for longer** than expected, its **deadline is automatically postponed**, which in practice corresponds to lowering the task priority



Constant bandwidth server - CBS

The **complexity** of implementing a CBS is higher than that of the TBS due to the required capacity management. Anyway, there is still **one single ready queue**, for periodic and aperiodic tasks, managed by deadlines.

The **average response time** to aperiodic requests is **similar** to that of the TBS.

The **impact** on the schedulability of the periodic subsystem is equal to that of a task with the same parameters than the server.

Using EDF+CBS

$$U_p + U_S \leq 1$$

Constant bandwidth server - CBS

The main motivation to use a CBS resides in the **bandwidth isolation that it provides**.

If a task is served by a CBS with bandwidth U_S, in any interval Δt multiple of its period, such task (and server) will never require more than $\Delta t * U_S$ of CPU.

Any task τ_i (C_i,T_i) schedulable under EDF is equally schedulable running within a CBS with $\mathbf{Q_S=C_i}$ and $\mathbf{T_S=T_i}$

Therefore, a CBS can be used for:

- Protecting a system of tasks ***overruns***
- Guaranteeing a **minimal service** to soft real-time tasks
- **Reserve bandwidth** for any activity.

Wrapping up

- Executing together **periodic** and **aperiodic tasks**
  - Basic technique
    - Executing aperiodic tasks in the **background**
  - Using **servers** to run **aperiodic tasks**
    - **Fixed priority** servers
      - Periodic Server – (is a periodic task)
      - Polling Server - PS
      - Deferrable Server - DS
      - Sporadic Server - SS
    - **Dynamic priority** servers
      - Total Bandwidth Server – TBS
      - Constant Bandwidth Server - CBS

part 8

Further issues in real-time scheduling

Non-preemptive scheduling
Practical implementation aspects

Non-preemptive scheduling

Non-preemptive scheduling consists in executing tasks completely, **not allowing their suspension** for the execution of other higher priority tasks

Main features (advantages):

- **Very simple to implement** since it does not need saving the task state (they execute without interruption).
- The stack can be shared among all tasks and thus the **total stack requirement is minimal** (same as the maximum stack requirement among the individual requirements of all tasks)
- The access to **shared resources** does not need any **specific protocol** (tasks already execute with mutual exclusion)

Non-preemptive scheduling

Main features (disadvantages):

- Implies a **schedulability penalty** that is as important as there are long execution times in the task set (longer blocking times).
- Such penalty becomes critical when simultaneously there are tasks that require a fast activation rate (short deadlines).

The impact of non-preemption is similar to a **blocking in the access to a shared resource, i.e., the CPU**. This allows using most of the tests presented before for systems with preemption and shared resources.

In this case,

$$\forall_i \ \ \boldsymbol{B_i} = \max_{k \in lpe(i)} \boldsymbol{C_k}$$

lpe(i) → set of tasks with lower or equal priority to task i

Non-preemptive scheduling

Beyond considering the blocking time associated to non-preemption, there are still a few more **adaptations** that need to be done in the response time-based schedulability analysis.

Computing Rwc$_i$ with **fixed priorities**:

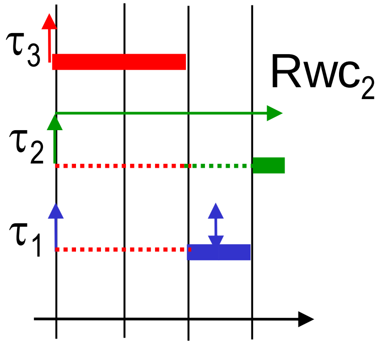
$$\forall_i \; Rwc_i = I_i + C_i$$

The iterative procedure is executed just **over I$_i$** since, after starting, task i will execute until completion. However, **I$_i$** should also include any higher priority activation that may occur at the **last instant** (⌊ ⌋ +1)

$$I_i = \sum_{k \in hp(i)} \left(\left\lfloor \frac{I_i}{T_k} \right\rfloor + 1 \right) * C_k$$

$$I_i(0) = B_i + \sum_{k \in hp(i)} C_k$$

$$I_i(m+1) = B_i + \sum_{k \in hp(i)} \left(\left\lfloor \frac{I_i(m)}{T_k} \right\rfloor + 1 \right) * C_k$$

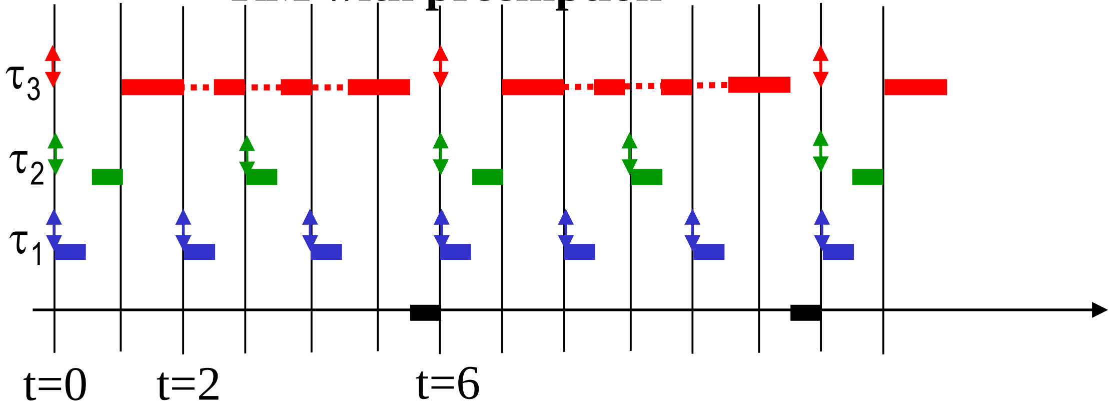


Non-preemptive scheduling

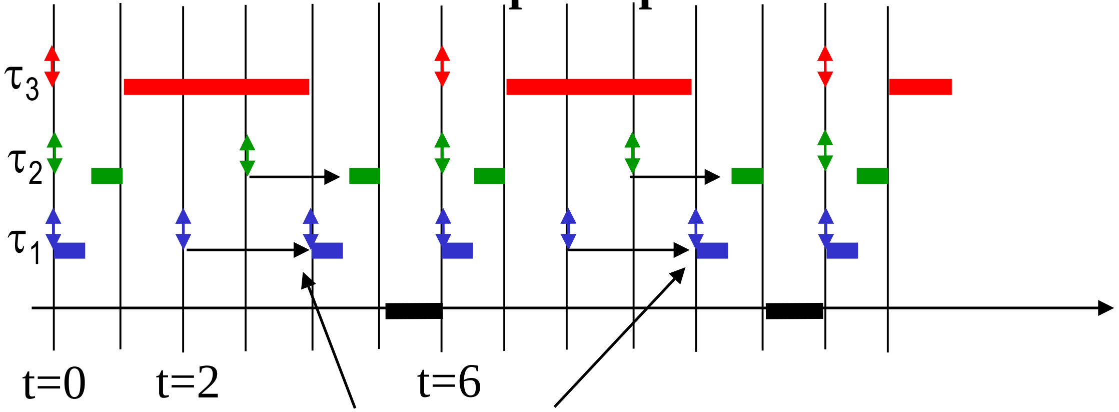
Task set

τ_i	T_i	C_i
1	2	0.5
2	3	0.5
3	6	3

RM with preemption



RM without preemption

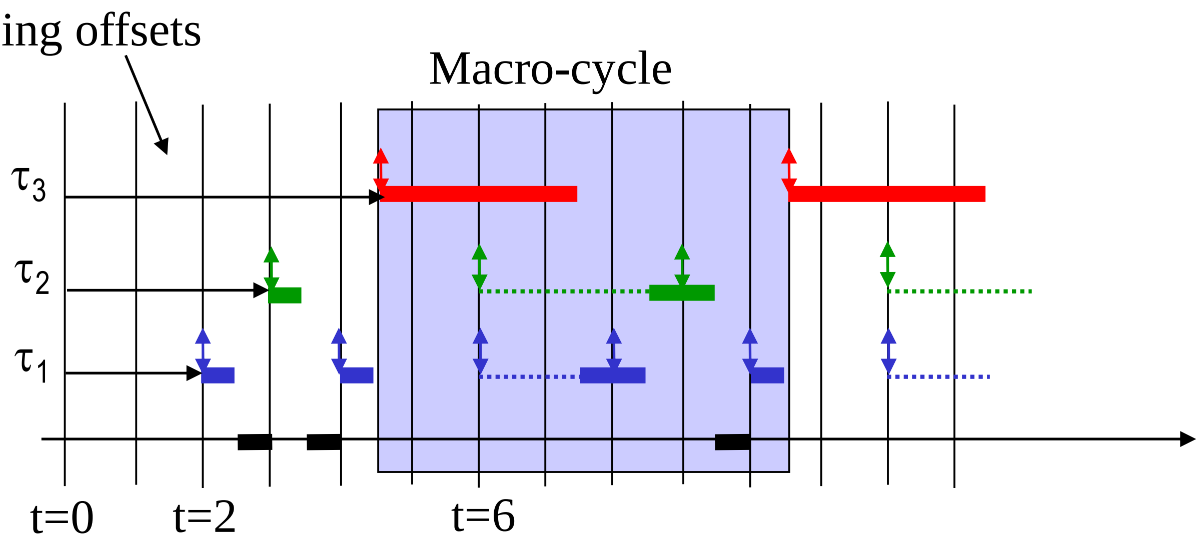


Blocking and deadline miss

Non-preemptive scheduling

Task set

τ_i	T_i	C_i	O_i
1	2	0.5	**2**
2	3	0.5	**3**
3	6	3	**4.5**



Using offsets might be particularly helpful in scheduling without preemption, frequently **turning schedulable** a system that was not.

Aspects of practical implementation

When building **real applications** there are a few extra aspects that should be considered since they might influence the accuracy of the schedulability tests:

- Overhead of **kernel internal mechanisms** (e.g., *tick handler*)
- Overhead of **context switching**
- The tasks **worst-case execution times (WCET)**
- Execution of other **interrupt service routines**
- **Deviations** in the tasks **activation instants**

Tick handling

Determining the tick handling overhead

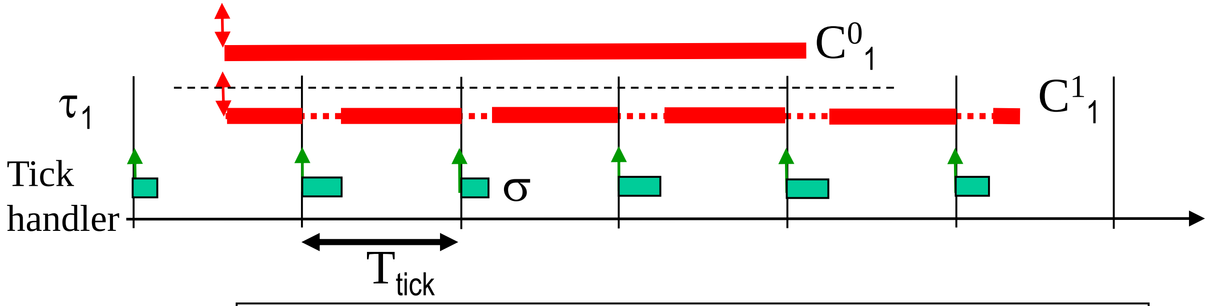
- Serving the **system clock tick** (when it exists) requires CPU time beyond that needed for the execution of the tasks.
- It is, often, the **highest priority** activity executed by the system and in a simplified way it can be modeled as a **periodic task.**
- The associated **overhead** (σ) may have a substantial impact on the efficiency of the system since it represents a fraction of CPU bandwidth that is stolen from that available to execute actual tasks.

Tick handling

Determining the tick handling overhead *(cont.)*

- It can be **measured directly** using a **timer** that is read at the start and end of the tick handler (plus a small correction to account for entering or leaving the handler). This requires **kernel instrumentation**.
- or **indirectly** using a **long function** that is executed with and without the tick handling interrupts (period T_{tick}) and measuring the execution time difference (C^0_1 and C^1_1 respectively). This is done from the **user space**. In this case,

$$\sigma = \frac{C_1^1 - C_1^0}{\left\lceil \frac{C_1^1}{T_{tick}} \right\rceil}$$



This latter technique returns average σ, only!

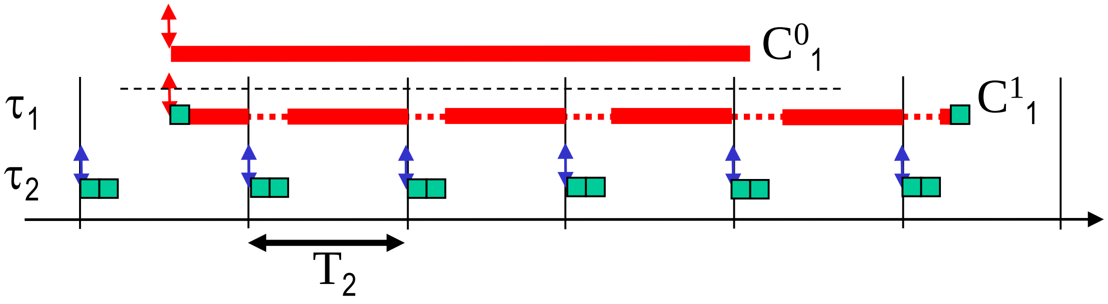
Context switching

Determining the context switch overhead

- **Context switching** also requires some CPU time beyond the tasks code execution.
- Again, this overhead (δ) can be **measured directly** with a **timer**, **instrumenting the kernel**, or **indirectly** using a **pair of tasks**, a long one ($T_{\mathbf{1}}$) and a short one ($T_{\mathbf{2}}$), with higher priority and without actual code. Then we measure the execution time of $T_{\mathbf{1}}$ alone ($C^0{}_1$) and together with $T_{\mathbf{2}}$ ($C^1{}_1$).

  In this case,

$$\boldsymbol{\delta = \frac{C_1^1 - C_1^0}{\left\lceil \frac{C_1^1}{T_2} \right\rceil}}$$

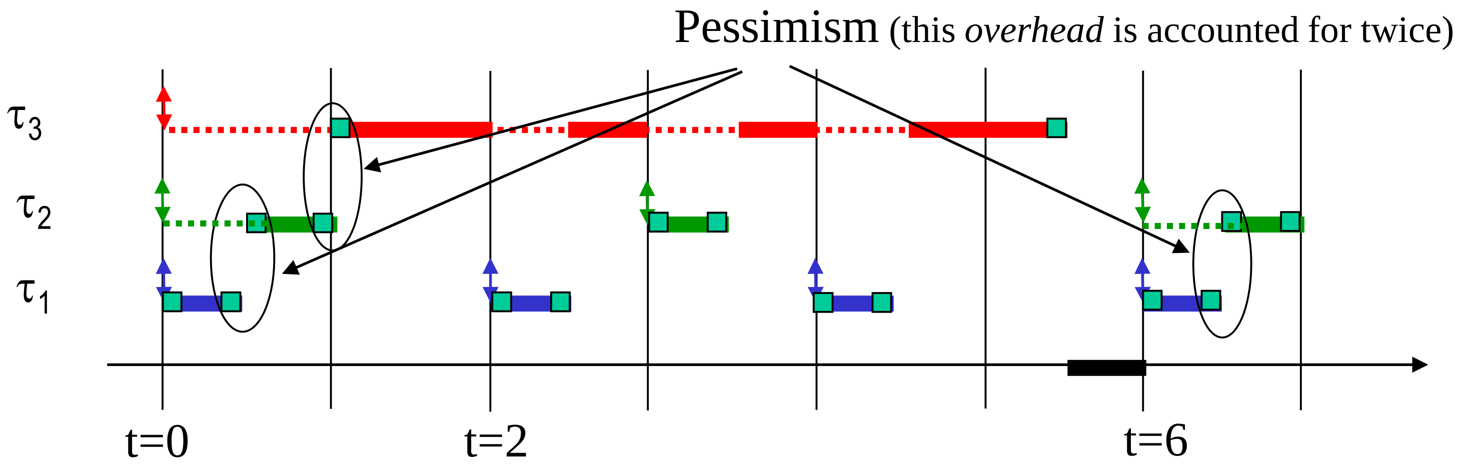


This latter technique returns average δ, only!

Context switching

Accounting for the context switch overhead

- A **simple** (but pessimistic) way to take the context switch overhead (δ) into account consists in **adding** this term to the tasks **execution time**.


Pessimism (this *overhead* is accounted for twice)

Worst-case execution times

Determining the tasks worst-case execution times

- This is normally carried out **analyzing the source code**, to determine the **longest execution path**, according to the input values.
- Then the **object code** is analyzed to determine the number of CPU clock cycles needed to execute the instructions in the longest path.

Note that the **task execution time** can **vary** from instance to instance due to **cycles** and **conditionals**, dependence on the input data and state, and even system interference by means of data/instruction caches and pipelines.

Worst-case execution times

Determining the tasks worst-case execution times (cont.)

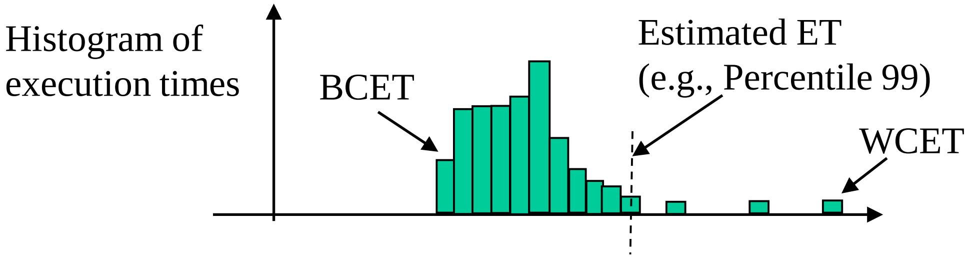
- It is also possible to **execute a task in isolation** in a controlled way, providing it with **adequate input data** and measure the actual execution in the final platform. However, this **experimental method** is **not accurate** and might not reveal the worst-case execution time.

Current **complex processors** use pipelines and caches (data and instructions) that improve the average execution time substantially increasing the difference between best and worst-cases. For this case, there are **specific analysis** that **bound** the **maximum number** of **cache misses** and **pipeline flushes** according to the effective instruction sequence and set of activities executed concurrently, reducing the pessimism of the WCET value.

Worst-case execution times

Determining the tasks worst-case execution times (cont.)

- It is also possible to use **stochastic analysis** of the execution times.
- The purpose is to determine the probability distribution of the execution times and to use an **estimate for the maximum time** that covers a high percentage (e.g., 99%) of the occurrences.
- Often, mainly when the worst-case is rare and much larger than the average, this technique allows reducing substantially the WCET pessimism and increasing the CPU utilization (**higher efficiency**)



Interrupt service routines

Impact of other interrupt service routines

- Generally, **interrupt handling** is carried out with a **higher priority level** with respect to that at which tasks are executed, and this is set by HW.
- Thus, in **fixed priorities systems**, the impact can be directly computed considering **ISRs as higher priority tasks** in the schedulability analysis.
- In a **dynamic priorities system**, a **hierarchical** approach must be used, with fixed priorities in the higher level and dynamic priorities in the lower level. For example, using the **processor demand** approach, a processor **supply bound function** *sbf(t)* can be defined that considers the worst-case time taken by the interrupts as unavailability time. Then the load condition becomes

$$\boldsymbol{h(t) \leq sbf(t) \quad \forall_{t\in[0\;L[}}$$

Interrupt service routines

Impact of other interrupt service routines (cont.)

- However, since ISRs are aperiodic, how to bound their impact?
  - The usual approach is to separate interrupt handling in two levels, **fast interrupt handlers**, tied to HW interrupts, and the **slow interrupt handlers**, which are aperiodic tasks activated by the fast handlers. The idea is to reduce to the minimum the length of Fast ISRs and use servers to handle Slow ISRs.
    - This may still suffer from overloads in the FISRs. To avoid this, it is possible to **enable / disable HW interrupts** selectively for each source, also **according to a server** policy, to bound their impact in the global system.

Release jitter

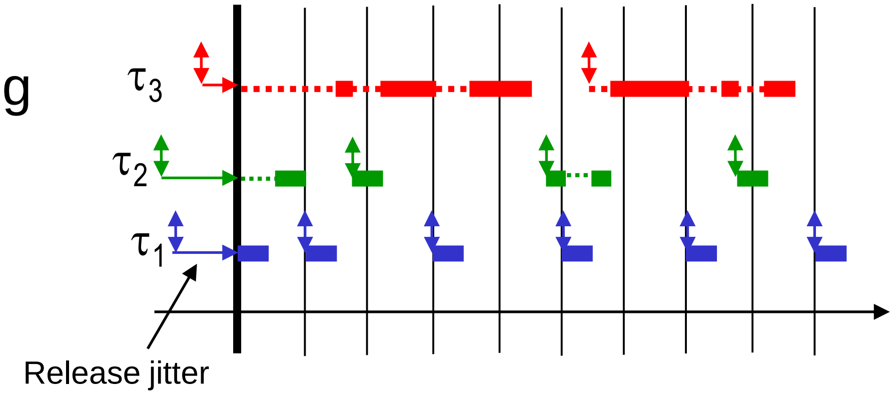
Impact of jitter in the tasks release instants

- A periodic task can suffer **jitter** in its activation instants if it is not triggered directly by a timer, e.g., when a task is activated by the termination of another one, or by an external interrupt, or by a message received through a communication port. In this case, the activation instant can be variably deferred with respect to the desired periodic instant – this is called **release jitter**
- Any existing release jitter can lead to a deferred execution that may result in a negative impact on the system schedulability and thus must be taken into account in the analysis.

Release jitter

Impact of jitter in the tasks release instants

- Under release jitter, the worst-case interference occurs when the following instances of all tasks are anticipated by an interval corresponding to such maximum deferral.



Computing Rwc$_i$ with preemption in fixed priorities, with release jitter

$$\forall_i\ Rwc_i = (J_i +)I_i + C_i \quad \text{and} \quad I_i = \sum_{k \in hp(i)} \left\lceil \frac{Rwc_i + J_k}{T_k} \right\rceil * C_k$$

$$Rwc_i(0) = \sum_{k \in hp(i)} C_k + C_i$$

$$Rwc_i(m+1) = \sum_{k \in hp(i)} \left\lceil \frac{Rwc_i(m) + J_k}{T_k} \right\rceil * C_k + C_i$$

Wrapping up

When **implementing real-time systems** there are some other scheduling-related issues that must be taken into account

- **Non-preemptive** scheduling has a negative impact on the schedulability but results in much simpler systems.
- Overhead of **kernel internal mechanisms** (e.g., *tick handler*)
- Overhead of **context switching**
- The tasks **worst-case execution times (WCET)**
- Execution of other **interrupt service routines**
- **Deviations** in the tasks **activation instants**