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

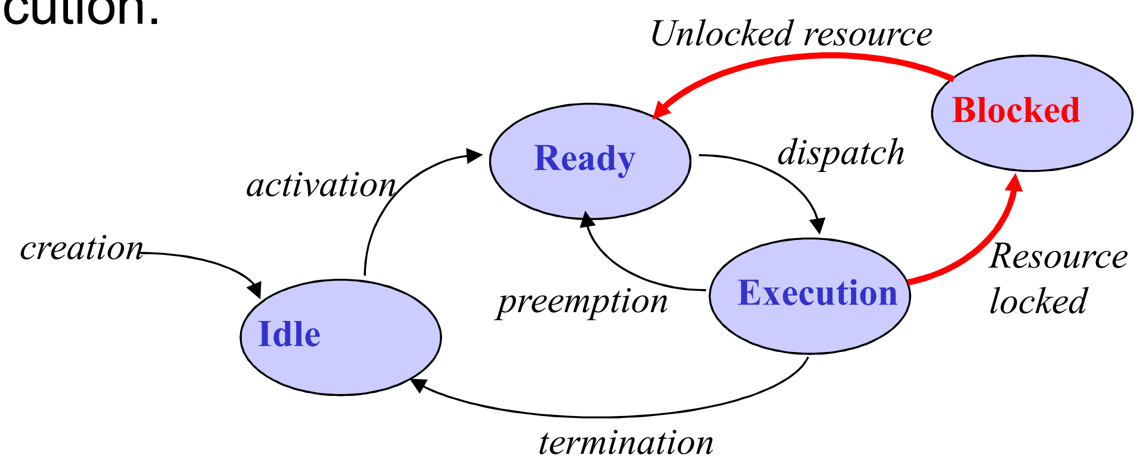
Earliest Deadline First *versus* Rate-Monotonic scheduling

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

Shared resources with exclusive access

Tasks: the blocking state

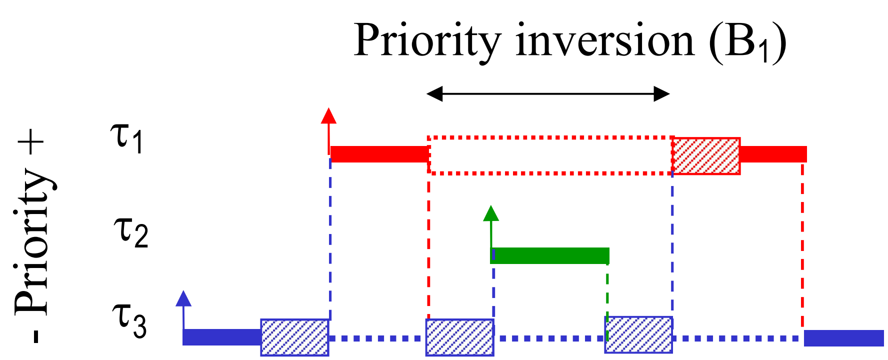
When a task that is executing tries to access a shared resource (e.g. a communication port, a buffer in memory, generally a **critical region**) that is locked by a ready task (necessarily with lower priority) the former task is said to be **blocked**. When the resource is unlocked, the blocked task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



The priority inversion phenomenon

In a real-time system with preemption and **independent** tasks, when a **task executes** that's because 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)

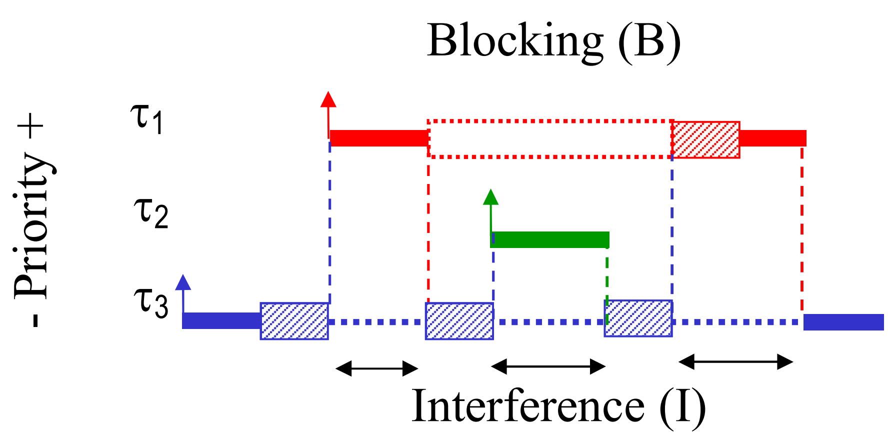
However, it is fundamental to **limit and quantify its impact** in the worst-case so that the **schedulability** of the task set can analyzed.

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

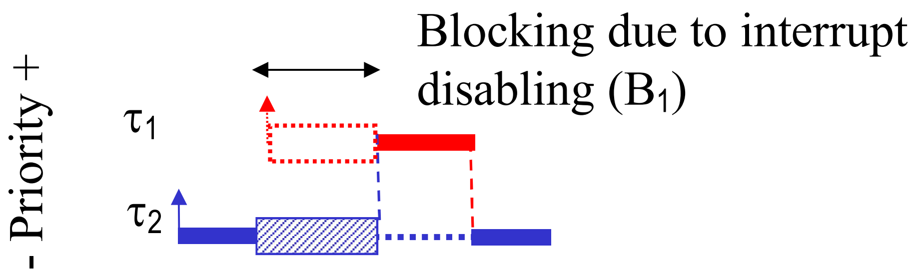
Synchronization primitives

- **Interrupts disabling** (*disable / enable*)
- **Preemption disabling** (*no_preemp / preempt*)
- Use of ***locks*** or ***atomic flags*** (*mutexes* – although this expression is also often used to refer to semaphores – *lock / unlock*)
- Use of **semaphores** (e.g., *wait / signal*)

Techniques to control accessing shared resources

Interrupts disabling

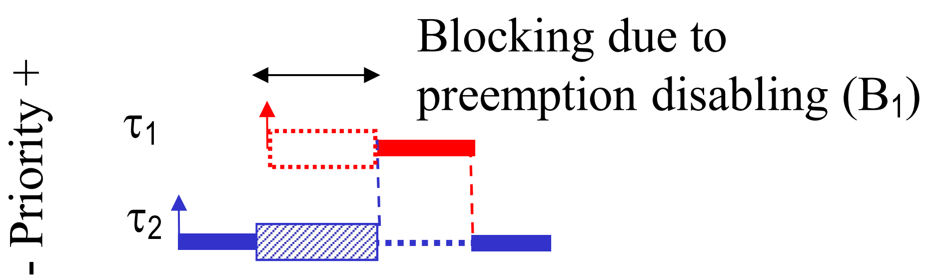
- **All activities** in the system **are blocked**!
  (not only other tasks, independently of using a shared resource or not, but also interrupt servicing including the system tick).
- This technique is very easy to apply but should only be used with very short critical regions (e.g. access to a variable)
- 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 !!



Techniques to control accessing shared resources

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 !!



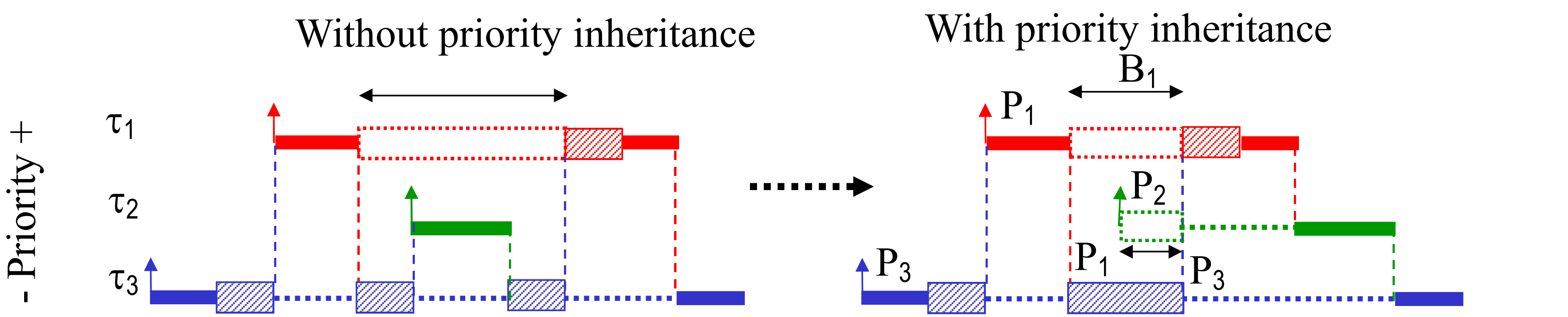
Techniques to control accessing shared resources

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**

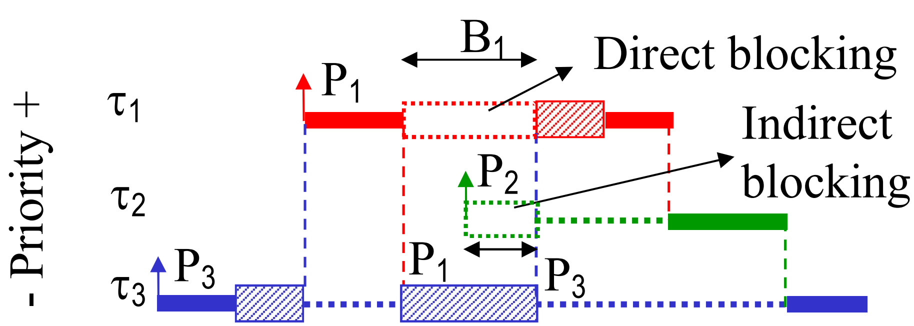
PIP – Priority Inheritance Protocol

- 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 <u>absence of nested resources accesses</u>:
    1. Each task can only block another task once
    2. Each task can only be blocked once in each semaphore



PIP – Priority Inheritance Protocol

Schedulability analysis (RM)

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

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

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

←

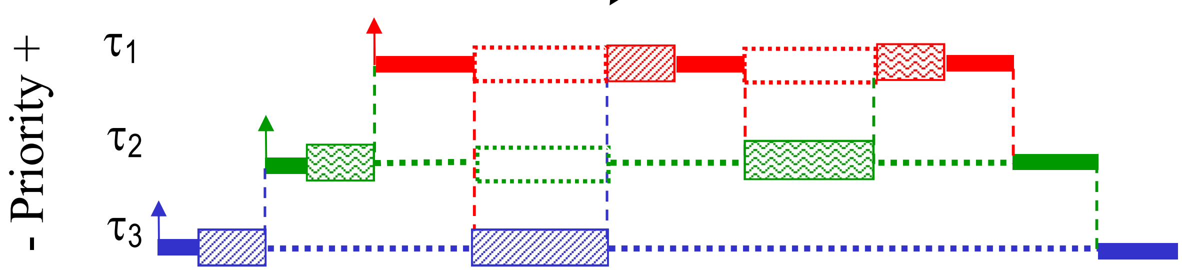
e.g.	S_1	S_2	S_3
τ_1	1	2	0
τ_2	0	9	3
τ_3	8	7	0
τ_4	6	5	4

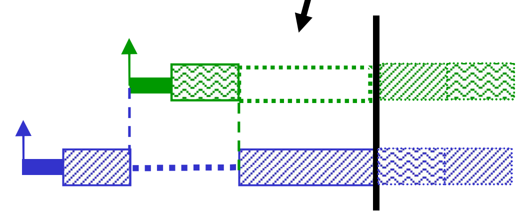
PIP – Priority Inheritance Protocol

+ 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***

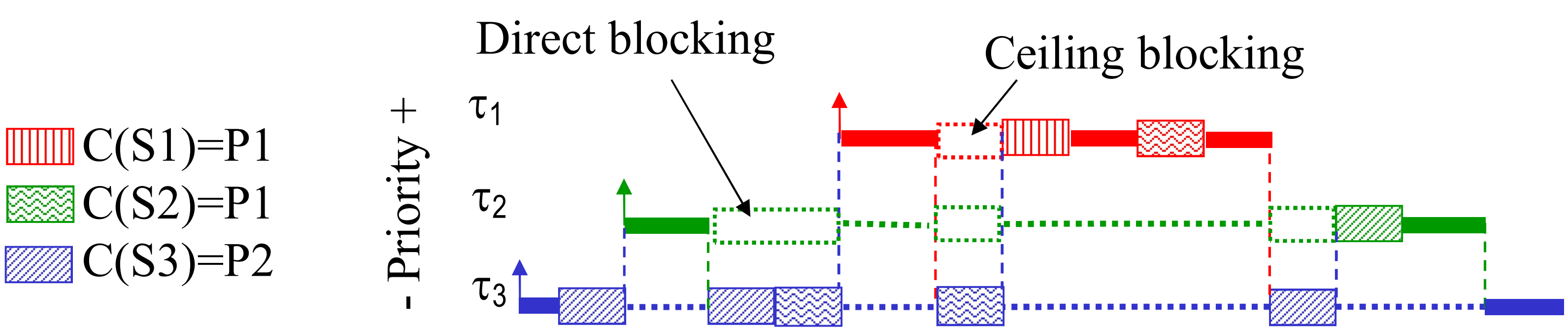




Deadlock due to nesting in resources access.

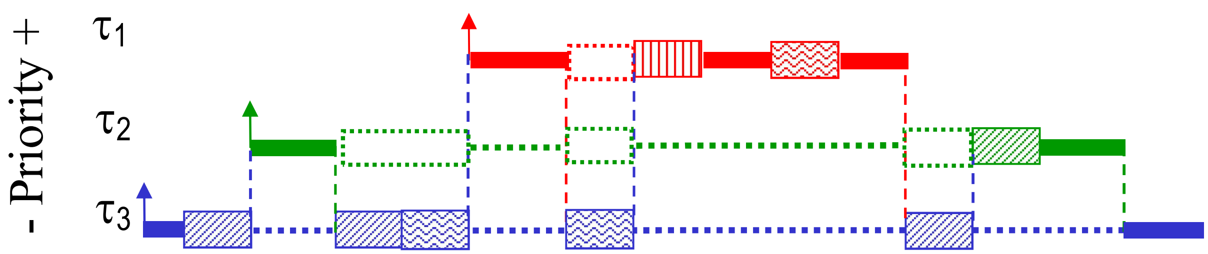
PCP – Priority Ceiling Protocol

- **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 a 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 $\boldsymbol{B_i}$ varies)*

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

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

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***,

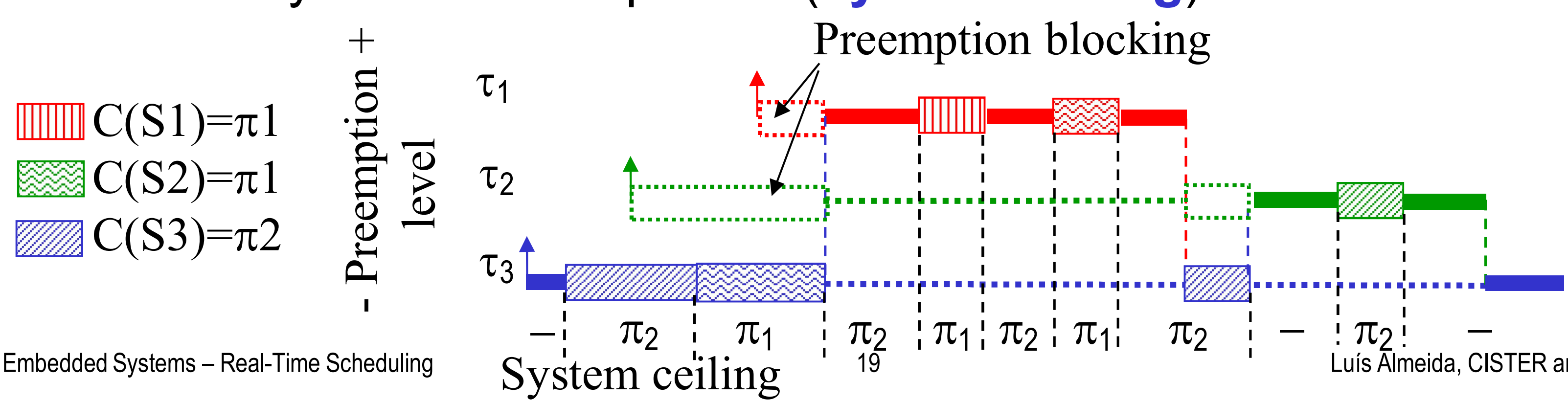
– Harder to implement (in the TCB, a new field is needed to hold the inherited priority and another one for the semaphore in which the task is blocked – to facilitate inheritance transitivity. Moreover, a new structure is needed to hold the current state of the semaphores with the respective ceilings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

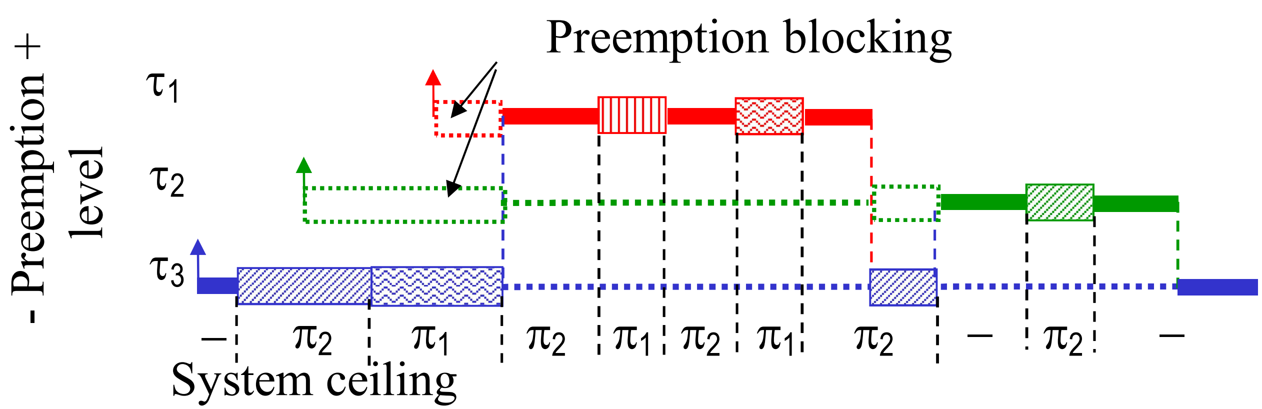
SRP – Stack Resource Policy

- 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 equals 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) *(just computing B_i varies)*

1. $\sum_{i=1}^{n} \frac{C_i}{T_i} + \max_{i} \left(\frac{B_i}{T_i} \right) \leq n \left(2^{\frac{1}{n}} - 1 \right)$
2. $Rwc_i = C_i + B_i + \sum_{k=1}^{i-1} \left\lceil \frac{Rwc_i}{T_k} \right\rceil * C_k$

Schedulability analysis (EDF)

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

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 even more 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
- Techniques based on **semaphores**
  - ***Priority Inheritance Protocol – PIP***
  - ***Priority Ceiling Protocol – PCP***
  - ***Stack Resource Protocol- SRP***