Real-Time Systems

Lecture 7

Exclusive access to shared resources

Exclusive access to shared resources
Priority inversion as a consequence of blocking
Basic techniques to enforce exclusive access to shared resources:
Priority Inheritance Protocol – PIP
Priority Ceiling Protocol – PCP
Stack Resource Protocol- SRP

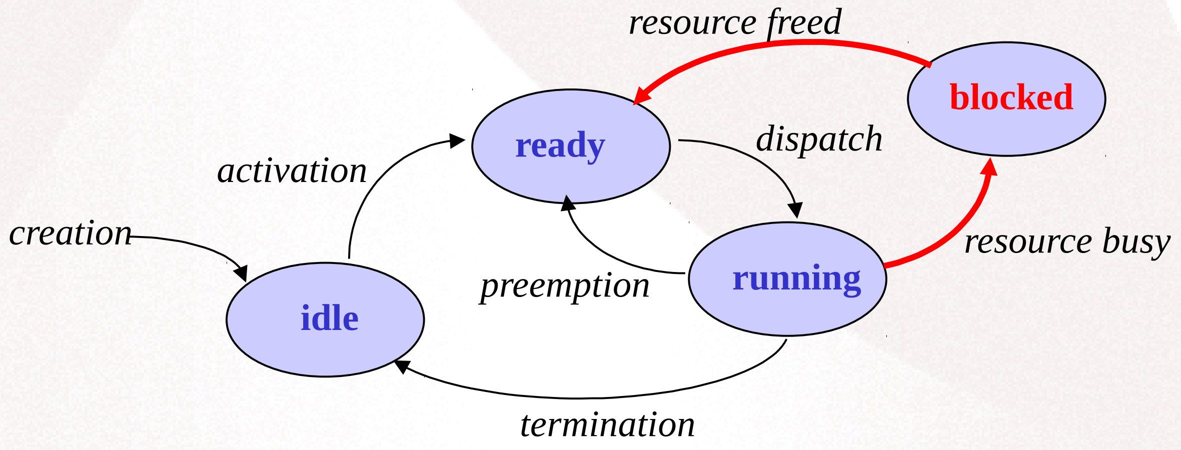
Last lecture (6)

- *On-line* scheduling with *dynamic priorities*
- The *EDF - Earliest Deadline First* criteria
- EDF *schedulability* analysis:
  - CPU utilization
  - CPU load method
- Optimality of EDF and comparison with RM:
  - Schedulability level, number of preemptions, jitter and response time
- Other dynamic priority criteria:
  - *LLF (LST) - Least Laxity (Slack) First*
  - *FCFS - First Come First Served*

Shared resources with exclusive access

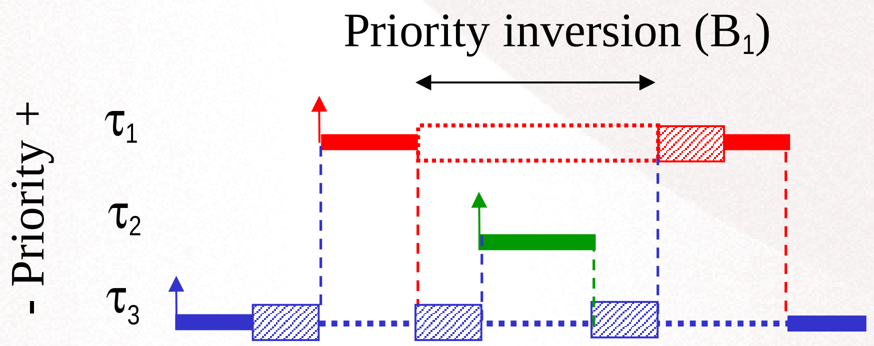
Tasks: the Blocked state

When a running task tries to access a shared resource (e.g. a buffer, a communication port) that is already taken (i.e. in use) by another task, the first one is **blocked**. When the resource becomes free, the blocked task becomes again ready for execution. To handle this scenario the state diagram is updated as follows:



The priority inversion phenomenon

- On a real-time system with **preemption** and **independent** tasks, the **highest priority** ready task is always the one in **execution**
- However, when tasks share resources with exclusive access, the case is different. An **higher priority task may be blocked** by another (**lower priority**) task, whenever this latter one owns a resource needed by the first one. In such scenario it is said that the higher priority task is **blocked**.
- When the blocking task (and eventually other tasks with intermediate priority) execute, there is a **priority inversion**.



The priority inversion phenomenon

- The **priority inversion** is an unavoidable phenomenon on the presence of shared resources with exclusive access.

- However, in real-time systems, it is of utmost importance **bound and quantify its worst-case impact**, to allow reasoning about the **schedulability** of the task set.

- Therefore, the techniques used to guarantee the exclusive access to the resources (**synchronization primitives**) must restrict the area of the priority inversion and be **analyzable**, i.e., allow the **quantification** of the maximum **blocking time** that each task may experience in any shared resource.

Techniques to allow exclusive access

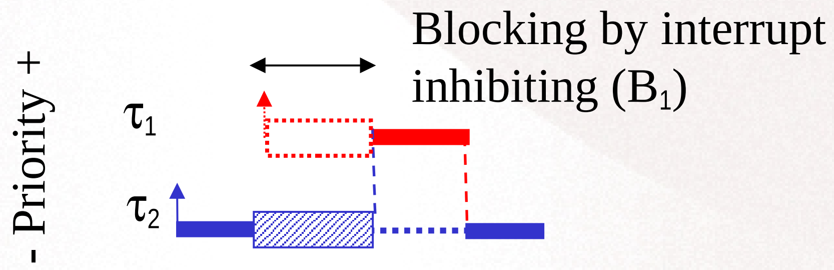
Synchronization primitives

- Disable **Interrupts**
  - *disable / enable* or *cli / sti*
- Inhibit the **preemption**
  - *no_preemp / preempt*
- Use of ***locks*** or ***atomic flags*** (*mutexes* – though this term is also used to designate semaphores – *lock / unlock*)
- Use of **semaphores**
  - Counter + task list – *P / V* ou *wait / signal*

Techniques to allow exclusive access

Interrupt inhibit

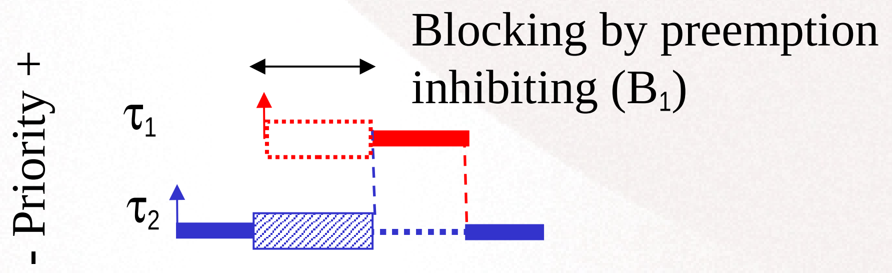
- **All other system activities are blocked**, not just other tasks, but also interrupt service routines, including the **system tick handler**.
- This technique is very easy to implement but should only be used with **very short critical regions** (e.g. access to a elemental variable)
- Each task can only be **blocked once** and for the **maximum duration of the critical region of lower priority tasks** (or smaller relative deadline for EDF), even if these don't use any shared resource!!



Techniques to allow exclusive access

Preemption inhibiting

- **All other tasks are blocked**. However, contrarily to disabling the interrupts, in this case the **interrupt service routines**, including the **system tick**, are **not blocked**!
- Very easy to implement but not efficient, as it causes unnecessary blocking.
- Each task can only be **blocked once** and for the **maximum duration of the critical region of lower priority tasks** (or smaller relative deadline for EDF), even if these don't use any shared resource!!



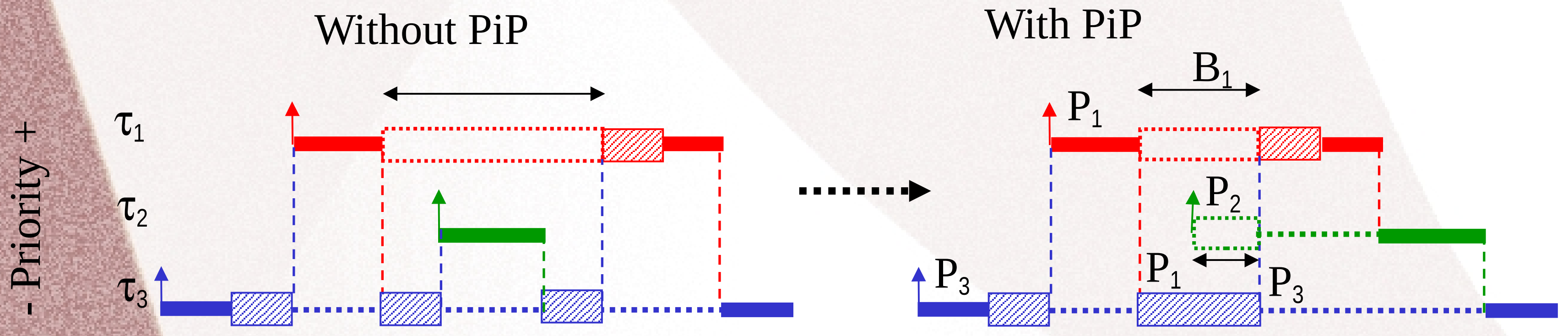
Techniques to allow exclusive access

Locks or semaphores

- **These techniques only block tasks that actually use the resources!**
- Costly but more efficient implementation
- However, the blocking duration depends on the **specific protocol** used to manage the semaphores
- These protocols must **prevent**:
  - **Indeterminate blocking**
  - **Chain blocking**
  - **Deadlocks**

PIP – Priority Inheritance Protocol

- The blocking task (lower priority) **temporarily inherits the priority** of the blocked task (the one with higher priority).
- Limits the blocking duration, **preventing** the execution of **intermediate priority tasks** while the blocking tasks owns the critical region. The priority of the blocking tasks returns to its nominal value when it exist the critical region.

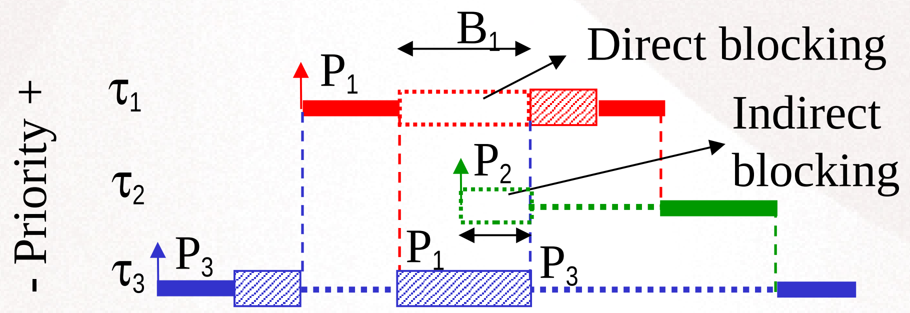


PIP – Priority Inheritance Protocol

To bound the **blocking time** (B) it is important to note that a task can be blocked by any lower priority task which:

- Shares a resource with it (**direct blocking**), or
- Can block a task with higher priority (***push-through or indirect blocking***)

- Note also that in the absence of chained accesses:
  - Each task can block any other task just once
  - Each task can block only once in each resource



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

Utilization test (RM)

$$\forall_{1\leq i\leq n} \sum_{h:P_h>P_i} \frac{C_h}{T_h} + \frac{C_i+B_i}{T_i} \leqslant i\left(2^{\frac{1}{i}}-1\right)$$

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

$\longleftarrow$

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

Response Time Analysis
(Fixed Priority)

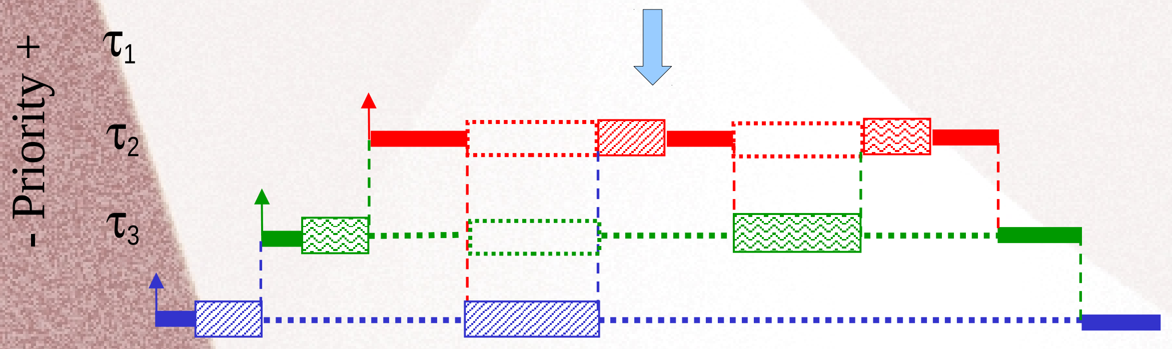
$$R_i^{(0)} = C_i + B_i$$

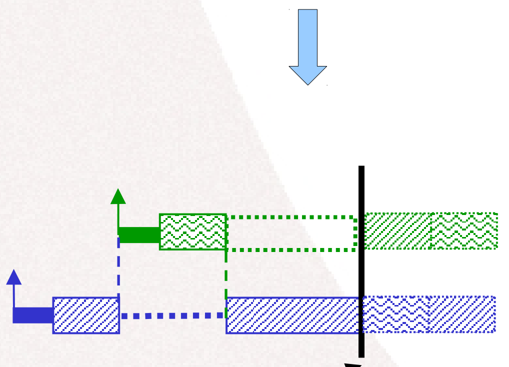
$$R_i^{(s)} = C_i + B_i + \sum_{h:P_h>P_i} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h$$

PIP – Priority Inheritance Protocol

Properties:

- Relatively easy to implement
  - One additional field on the TCB, the inherited priority
- Transparent to the programmer
  - Each task only uses local information
- Suffers from **chain blocking and does not prevent deadlocks**



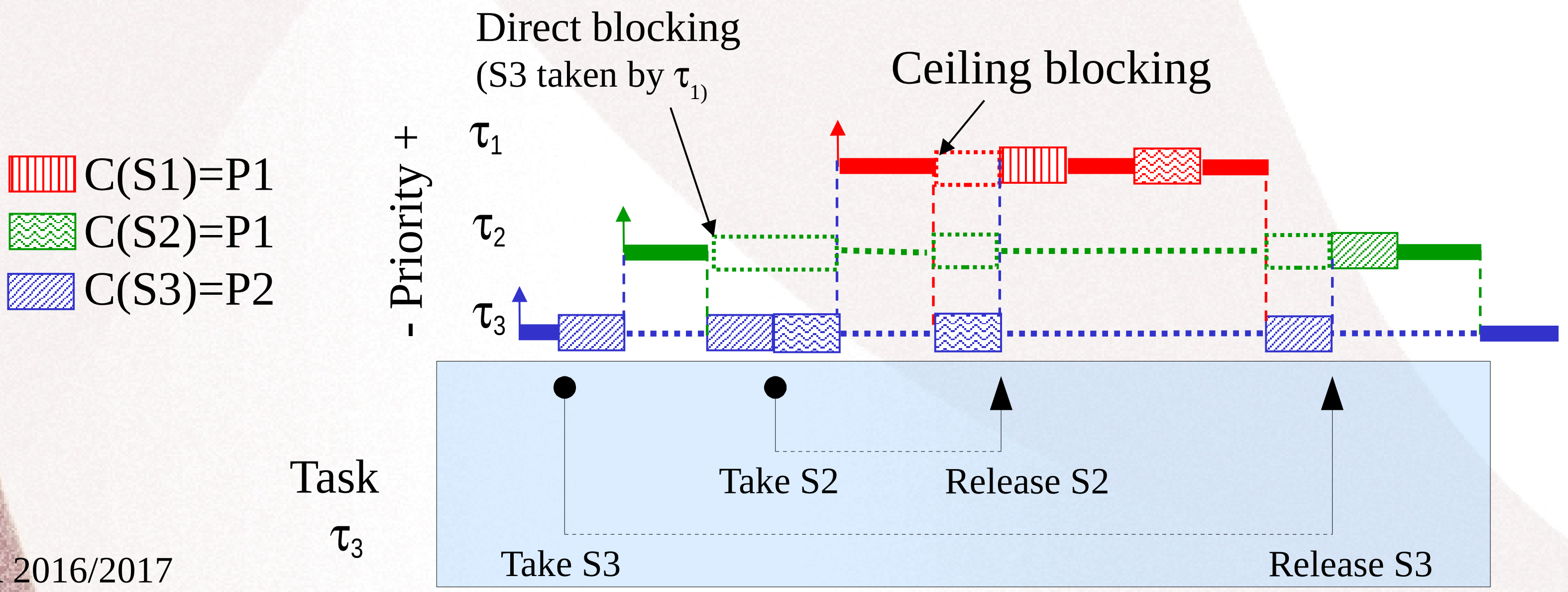


Deadlock due to resource nesting

PCP – Priority Ceiling Protocol

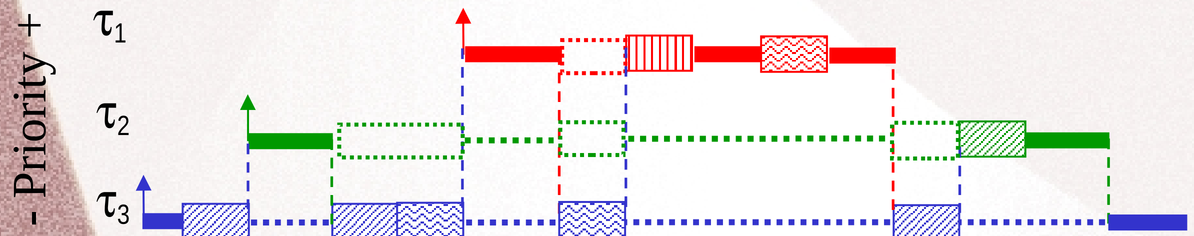
Extension of PIP with one additional rule about access to free semaphores, inserted to guarantee that all required semaphores are free.

- For each semaphore is defined a **priority *ceiling***, which equals the priority of the maximum priority task that uses it.
- A task can only **take a semaphore** if this one **is free** and if its **priority** is **greater than the ceilings** of all semaphores currently taken.



PCP – Priority Ceiling Protocol

- The PCP protocol only allows the access to the first semaphore when **all other semaphores that a task needs are free**
- To bound the **blocking time** (B) note that a task can be blocked **only once** by **lower priority tasks**. Only lower priority tasks that use semaphores which have a **ceiling at least equal** to the higher priority task can cause blocking.
- Note also that each task can only be blocked **blocked once**



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

$$\forall_{1 \le i \le n} \sum_{h: P_h > P_i} \frac{C_h}{T_h} + \frac{C_i + B_i}{T_i} \leqslant i\left(2^{\frac{1}{i}} - 1\right)$$

$$R_i^{(0)} = C_i + B_i$$

$$R_i^{(s)} = C_i + B_i + \sum_{h: P_h > P_i} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h$$

Same equations as for PiP!

Only the computation of ***B$_i$* varies**

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**

$\longleftarrow$

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

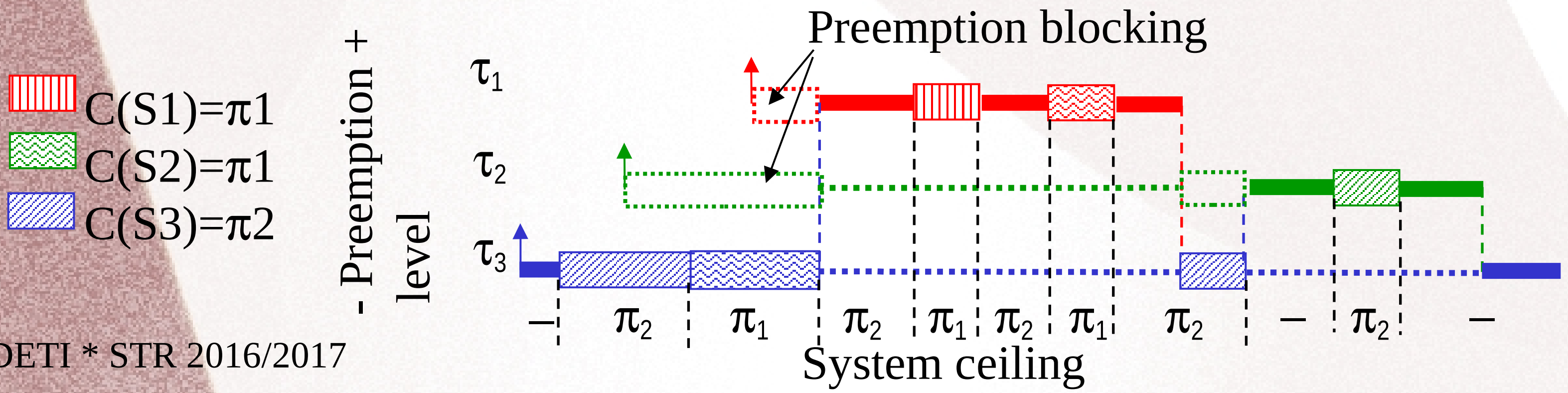
Properties:

- 🙂 Smaller blocking than PIP, **free of chain blocking and deadlocks**
- 🙁 **Much harder to implement than PiP**. On the TCB it requires one additional field for the inherited priority and another one for the semaphore where the task is blocked. To facilitate the transitivity of the inheritance it also requires a structure to the semaphores, their respective ceilings and the identification of the tasks that are using them
- 🙁 Moreover, it is not transparent to the programmer as the semaphore ceilings are not local to the tasks

There is one **version for EDF** in which all the blocking tasks inherit the deadline of the blocked ones and the semaphore ceilings use the relative deadlines to establish a preemption level.

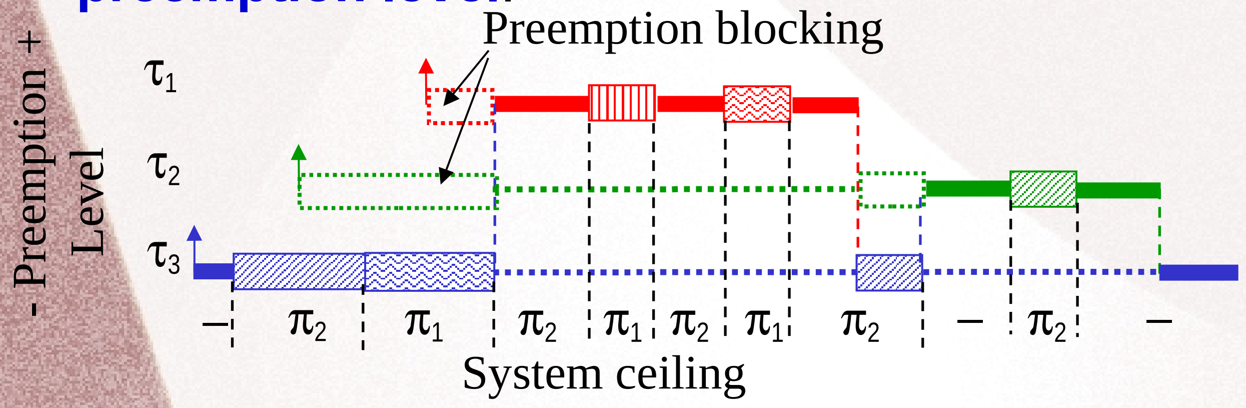
SRP – Stack Resource Policy

- Similar to PCP, but with one rule about the **beginning of execution, to guarantee** that all required semaphores are **free**
- Uses also the concept of priority ***ceiling***
- Defines the **preemption level** (π) as the capacity of a task to cause preemption on another one (static parameter).
- A task may only **start executing** when its own **preemption level** is **higher than the one of the executing task** and also **higher than the ceilings of all the semaphores in use** (system ceiling).



SRP – Stack Resource Policy

- The SRP protocol only allows that a task starts executing when all resources that it needs are free
- The upper bound of the **blocking time** (B) is equal to the one of the PCP protocol, but it occurs in a different time - **at the beginning of the execution** instead of at the shared resource access.
- Each task can **block only once** by any task with a lower preemption level that uses a **semaphore whose ceiling is at least equal to its 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 (Fixed. Prio.) **Schedulability analysis (EDF)**

$$\forall_{1 \le i \le n} \sum_{h:P_h > P_i} \frac{C_h}{T_h} + \frac{C_i + B_i}{T_i} \leqslant i\left(2^{\frac{1}{i}} - 1\right)$$

$$R_i^{(0)} = C_i + B_i$$

$$R_i^{(s)} = C_i + B_i + \sum_{h:P_h > P_i} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h$$

Only varies the way ***B_i***

is computed

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

Properties:

- ☺ Smaller blockings than PiP, **free of chain blockings and deadlocks**
- ☺ **Smaller number of preemptions than PCP**, intrinsic compatibility with fixed an dynamic priorities and to resources with multiple units (i.e., that allow more than one concurrent access, e.g. buffer arrays)
- ☹ The hardest to implement (preemption test much more complex, requires computing the system ceiling, etc.)
- ☹ Not transparent to the programmer (semaphore ceilings, etc.)

Summary of lecture 7

- Access to **shared resources**: blocking
- The **priority inversion**: need to bound and analyze
- **Basic techniques** to allow exclusive access to shared resources
  - Disable interrupts, preemption
- **Advanced techniques** to allow exclusive access to shared resources
  - ***The Priority Inheritance Protocol – PIP***
  - ***The Priority Ceiling Protocol – PCP***
  - ***The Stack Resource Protocol - SRP***