Real Time Operating Systems

Shared Resources

Luca Abeni

Credits: Luigi Palopoli, Giuseppe Lipari, and Marco Di Natale

Scuola Superiore Sant'Anna

Pisa -Italy

Interacting Tasks

- Until now, we have considered only independent tasks
  - A job never blocks or suspends
  - A task only blocks on job termination
- In real world, jobs might block for various reasons:
  - Tasks exchange data through shared memory $\rightarrow$ mutual exclusion
  - A task might need to synchronize with other tasks while waiting for some data
  - A job might need a hardware resource which is currently not available
  - ...

Interacting Tasks - Example

- Consider as an example three periodic tasks:
  - τ_1 reads the data from the sensors and applies a filter. The results of the filter are stored in memory
  - τ_2 reads the filtered data and computes some control law (updating the state and the outputs); both the state and the outputs are stored in memory
  - Finally, τ_3 reads the outputs from memory and writes on the actuator device
- All three tasks access data in the shared memory
- Conflicts on accessing this data in concurrency could make the data structures inconsistent

Task Interaction - Paradigms

- Interactions between tasks:
  - Private Resources - Client / Server paradigm
  - Shared Resources
- Private Resources
  - A *Resource Manager* (server task) per resource
  - Interaction via IPC
- Shared Resources
  - Must be accessed in *mutual exclusion*
  - Interaction via mutexes, semaphores, condition variables, . . .
- We will focus on shared resources (extensions to IPC based communication is possible)

Resources and Critical Sections

- Shared data structure representing a *resource* (hw or sw)
- Piece of code accessing the data structure: *critical section*
  - Critical sections on the same resource must be executed in *mutual exclusion*
  - Therefore, each data structure should be *protected* by a mutual exclusion mechanism;
- In this lecture, we will study what happens when resources are protected by mutual exclusion semaphores (mutexes)

Key Concepts

- Task
  - Schedulable entity (thread or process)
  - Flow of execution
  - In OO terminology each task implements an active object
  - Informally, it is an active entity that can perform operations on private or shared data
- Protected Objects
  - Encapsulating shared information (Resources)
  - Passive object (data) shared between different tasks
  - The execution of operations on protected objects is mutually exclusive (this is why they are protected)

Shared Resources - Definitions

- Shared Resource S_i
  - Used by multiple tasks
  - Protected by a *mutex* (mutual exclusion semaphore)
  - S_i can indicate either the resource or the mutex
- System / Application:
  - Set $\mathcal{T}$ of N periodic (or sporadic) tasks: $\mathcal{T} = \{\tau_i : 1 \leq i \leq N\}$
  - Set $\mathcal{S}$ of M shared resources: $\mathcal{S} = \{S_i : 1 \leq i \leq M\}$
  - Task τ_i *uses* resource S_j if it accesses the resource (in a critical section)
- k-th critical section of τ_i on S_j: $cs^k_{i,j}$
- Length of the longest critical section of τ_i on S_j: $\xi_{i,j}$

Posix Example

```
pthread_mutex_t s;
...
pthread_mutex_init(&s, NULL);
...
void *tau1(void * arg) {
    pthread_mutex_lock(&s);
    <critical section>
    pthread_mutex_unlock(&s);
};
...
void *tau2(void * arg) {
    pthread_mutex_lock(&s);
    <critical section>
    pthread_mutex_unlock(&s);
};
```

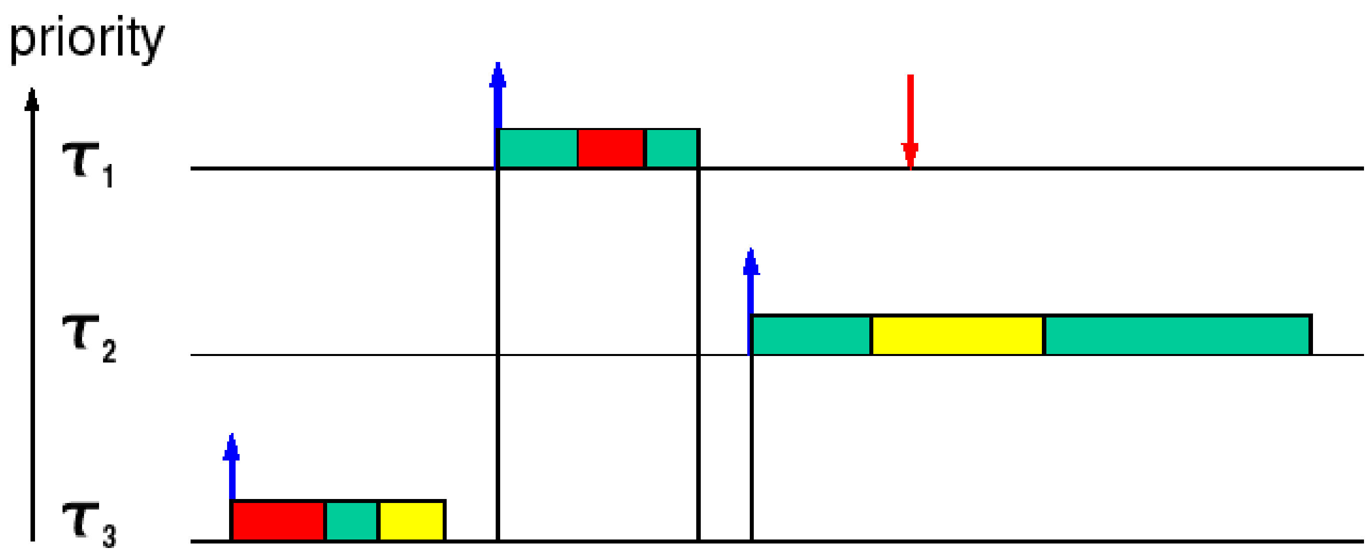
Blocking Time

- Mutual exclusion on a shared resource can cause *blocking time*
  - When task τ_1 tries to access a resource S already held from task τ_2, τ_1 blocks
  - Blocking time: time between the instant when τ_1 tries to access S (and blocks) and the instant when τ_2 releases S (and τ_1 unblocks)
- This blocking condition can be particularly bad in priority scheduling if a high priority tasks wants to access a resource that is held by a lower priority task
  - A low priority task executes, while a high priority one is blocked...
  - ...Schedulability guarantees can be compromised!

Blocking Time - Example

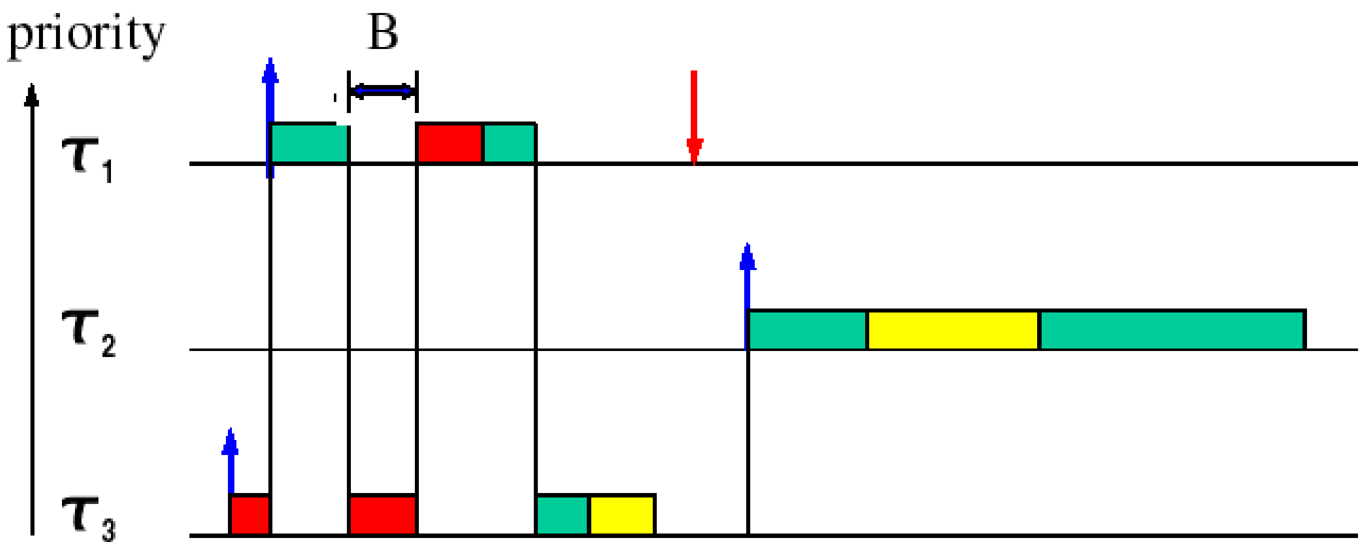
- A task incurs a blocking condition depending on the interleaving of the schedule

No conficts in this case



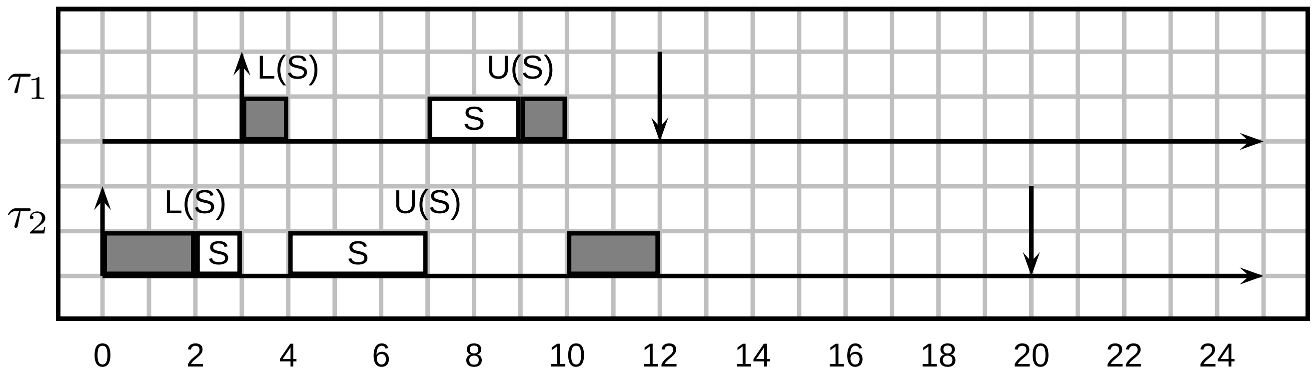
Blocking Time - Example

Blocking time in this case



Blocking and Priority Inversion

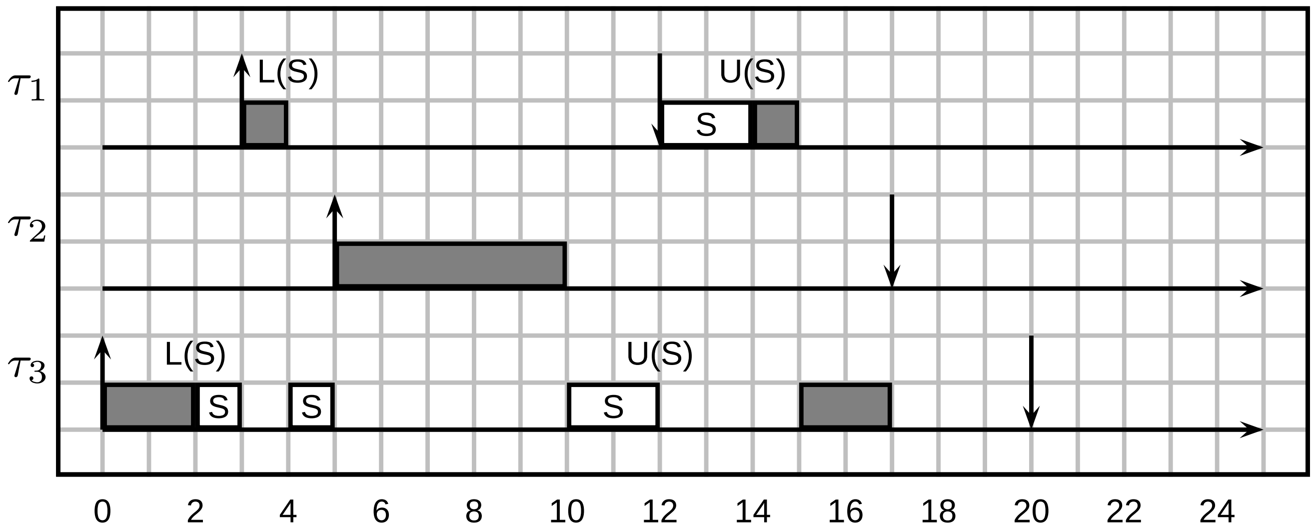
- Consider the following example, where $p_1 > p_2$.



- From time 4 to 7, task τ_1 is blocked by a lower priority taskτ_2; this is a *priority inversion*.
- This priority inversion is not avoidable; in fact, τ_1 must wait for τ_2 to leave the critical section.
- However, in some cases, the priority inversion could be too large.

Example of Priority Inversion

- Consider the following example, with $p_1 > p_2 > p_3$.



- Here, priority inversion is very large: from 4 to 12.
- Problem while τ_1 is blocked, τ_2 arrives and preempts τ_3 before it can leave the critical section.
- Other medium priority tasks could preempt τ_3 as well...

What Happened on Mars?

- This is not only a theoretical problem. It may happen in real cases.
- Most (in)famous example: Mars Pathfinder
  - A small robot, the Sojourner rover, was sent to Mars to explore the martian environment and collect useful information. The on-board control software consisted of many software threads, scheduled by a fixed priority scheduler. One high priority thread and one low priority thread were using the same software data structure through a shared semaphore. The semaphore was actually used by a library that provided high level communication mechanisms among threads, namely the `pipe()` mechanism. At some instant, it happened that the low priority thread was interrupted by medium priority threads while blocking the high priority thread on the semaphore.

    At the time of the Mars Pathfinder mission, the problem was already known. The first accounts of the problem and possible solutions date back to early '70s. However, the problem became widely known in the real-time community since the seminal paper of Sha, Rajkumar and Lehoczky, who proposed the Priority Inheritance Protocol and the Priority Ceiling Protocol to bound the time a real-time task can be blocked on a mutex semaphore.

More Info

A more complete (but maybe biased) description of the incident can be found here:
`http://www.cs.cmu.edu/~rajkumar/mars.html`

Dealing with Priority Inversion

- Priority inversion can be reduced...
  - ...But how?
  - By introducing an appropriate *resource sharing protocol* (concurrency protocol)
- Some protocols permit to find an *upper bound for the blocking time*
  - Non Preemptive Protocol (NPP) / Highest Locking Priority (HLP)
  - Priority Inheritance Protocol (PI)
  - Priority Ceiling Protocol (PC)
  - Immediate Priority Ceiling Protocol (Part of the OSEK and POSIX standards)
- mutexes (not generic semaphores) must be used

Non Preemptive Protocol (NPP)

- The idea is very simple *inhibit preemption when in a critical section*. How would you implement that?

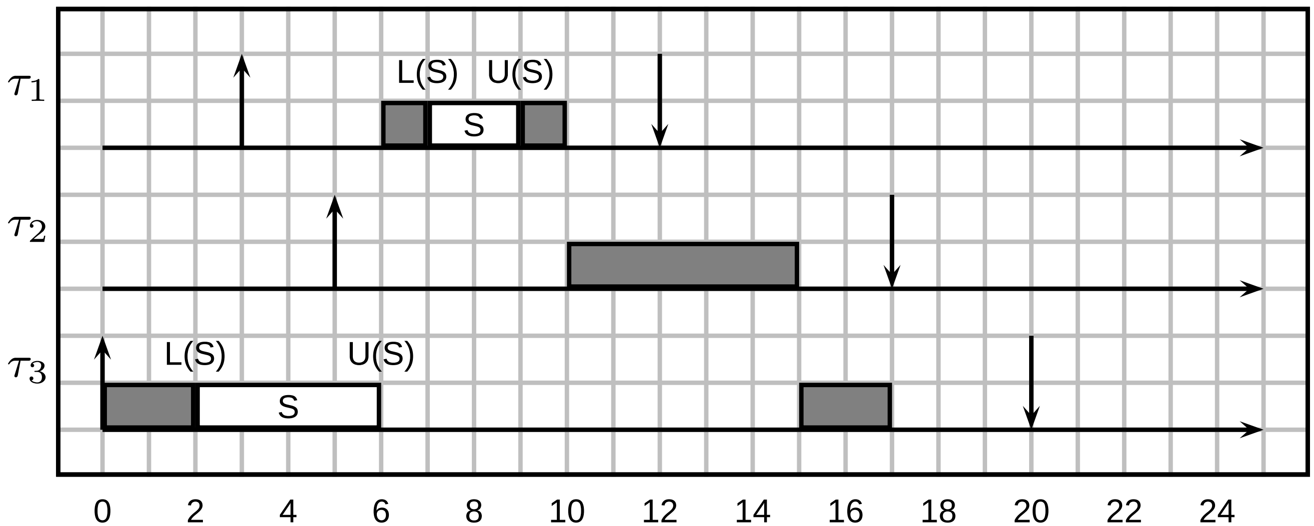
- Advantages: *simplicity*
- Drawbacks: tasks which are not involved in a critical section suffer blocking

Non Preemptive Protocol (NPP)

- The idea is very simple *inhibit preemption when in a critical section*. How would you implement that?
- Raise the task's priority to the maximum available priority when entering a critical section
- Advantages: *simplicity*
- Drawbacks: tasks which are not involved in a critical section suffer blocking

NPP Example

- Consider the following example, with $p_1 > p_2 > p_3$.

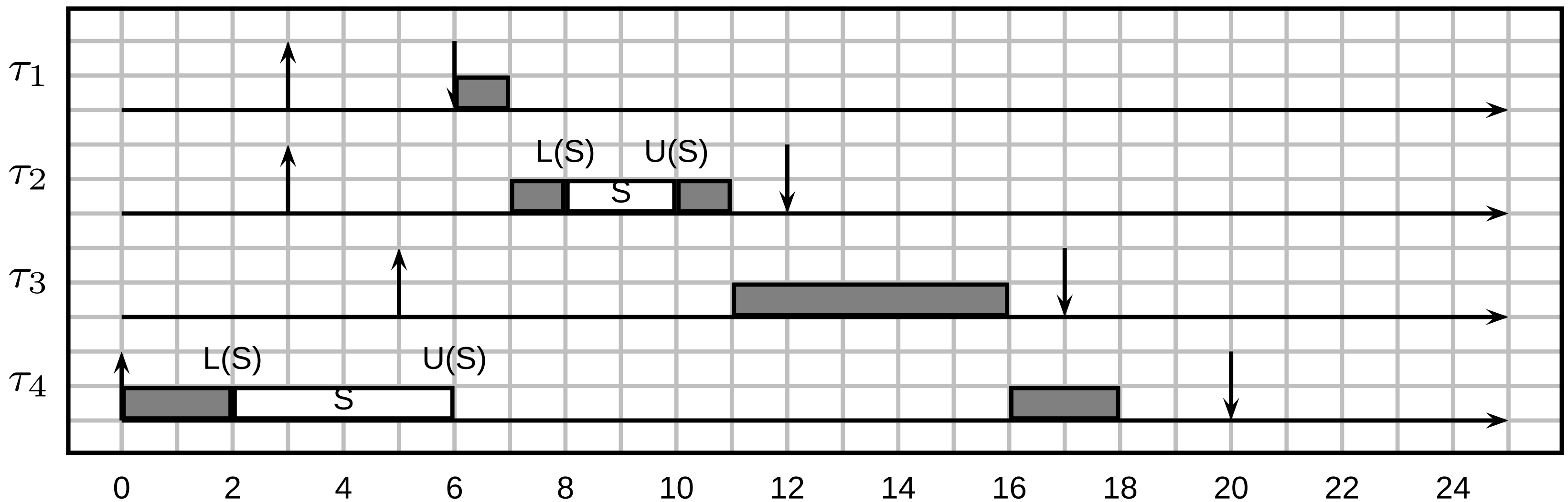


Some Observations

- The blocking (priority inversion) is bounded by the length of the critical section of task τ_3
- Medium priority tasks (τ_2) cannot delay τ_1
- τ_2 has a blocking time, even if it does not use any resource
  - *Indirect blocking*: due to the fact that τ_2 is *in the middle between* a higher priority task τ_1 and a lower priority task τ_3 which use the same resource.
  - This blocking time must be computed and taken into account in the formula as any other blocking time.
- What's the maximum blocking time B_i for τ_i?

A Problem with NPP

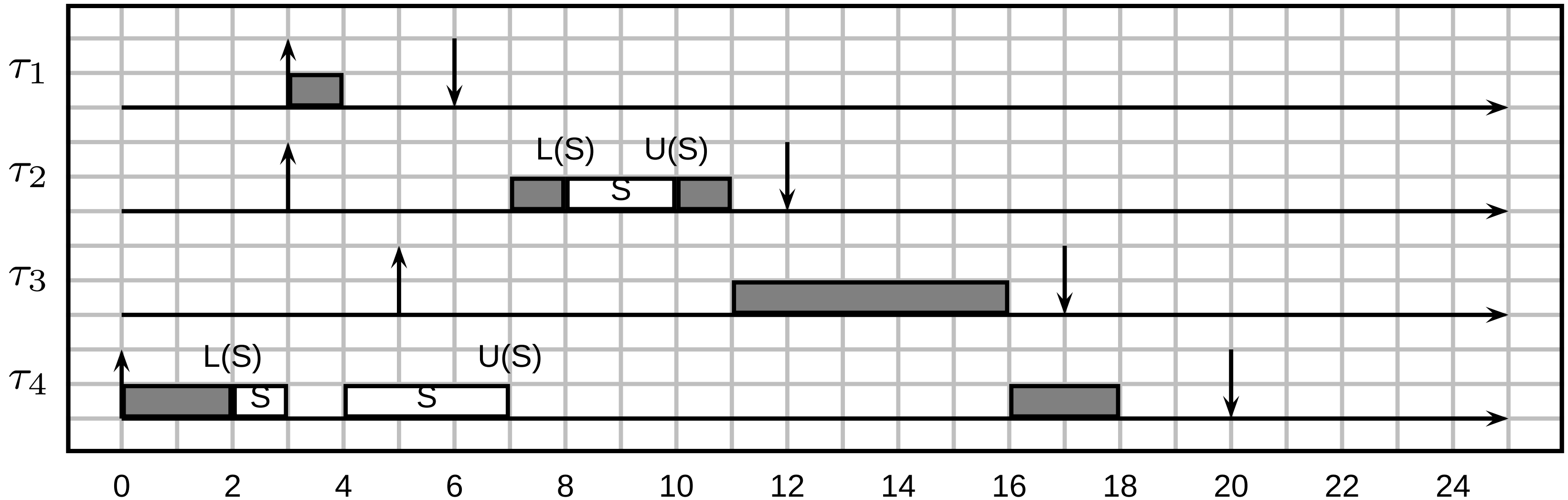
- Consider the following example, with $p_1 > p_2 > p_3 > p4$.



- τ_1 misses its deadline (suffers a blocking time equal to 3) even though it does not use any resource!!
- Solution: raise τ_3 priority to the maximum *between tasks accessing the shared resource* (τ_2' priority)
  - HLP

HLP

- So....



- This time, everyone is happy
- Problem: we must know in advance which task will access the resource

Blocking Time and Response Time

- NPP introduces a blocking time on **all** tasks bounded by the *maximum lenght of a critical section used by lower priority tasks*
- How does blocking time affect the response times?
- Response Time Computation:

$$R_i = C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

  - R_i is the response time of τ_i
  - B_i is the blocking time from lower priority tasks
  - $\sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$ is the preemption from higher priority tasks

Response Time Computation - I

Task	C_i	T_i	$\xi_{i,1}$	D_i
τ_1	20	70	0	30
τ_2	20	80	1	45
τ_3	35	200	2	130

Response Time Computation - II

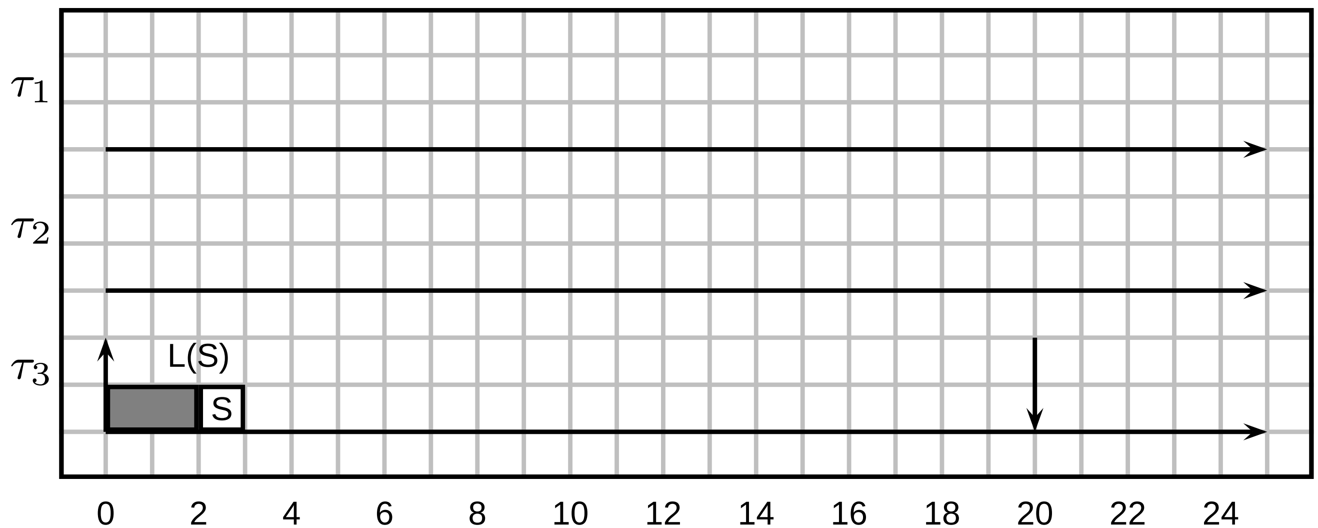
Task	C_i	T_i	$\xi_{i,1}$	D_i	B_i
τ_1	20	70	0	30	2
τ_2	20	80	1	45	2
τ_3	35	200	2	130	0

Response Time Computation - III

Task	C_i	T_i	$\xi_{i,1}$	D_i	B_i	R_i
τ_1	20	70	0	30	2	20+2=22
τ_2	20	80	1	45	2	20+20+2=42
τ_3	35	200	2	130	0	35+2*20+2*20=115

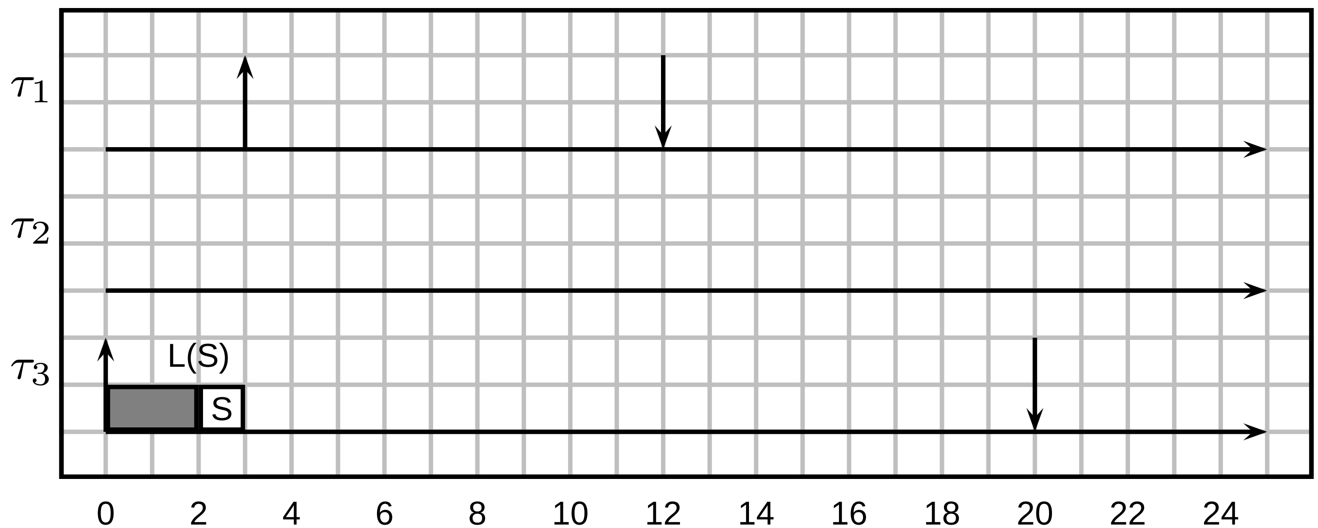
The Priority Inheritance protocol

- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



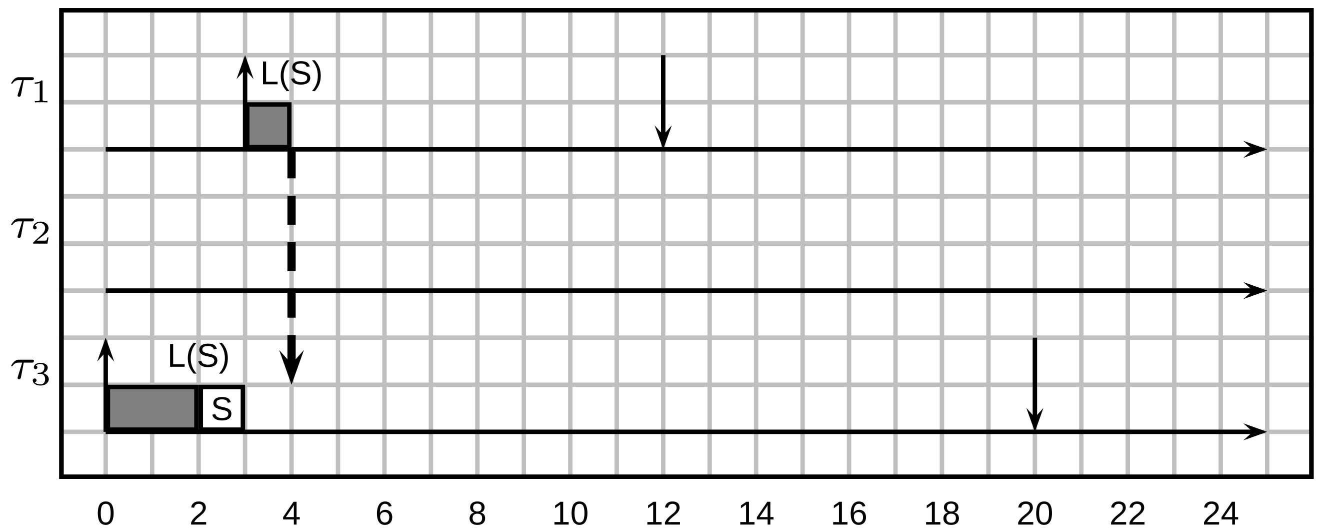
The Priority Inheritance protocol

- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



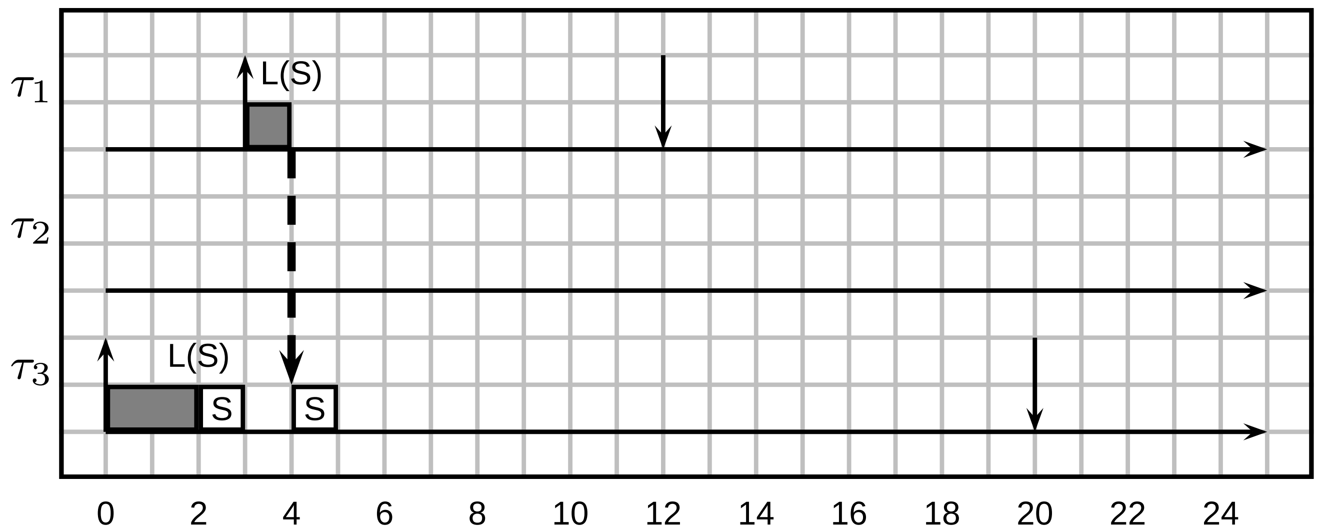
The Priority Inheritance protocol

- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



The Priority Inheritance protocol

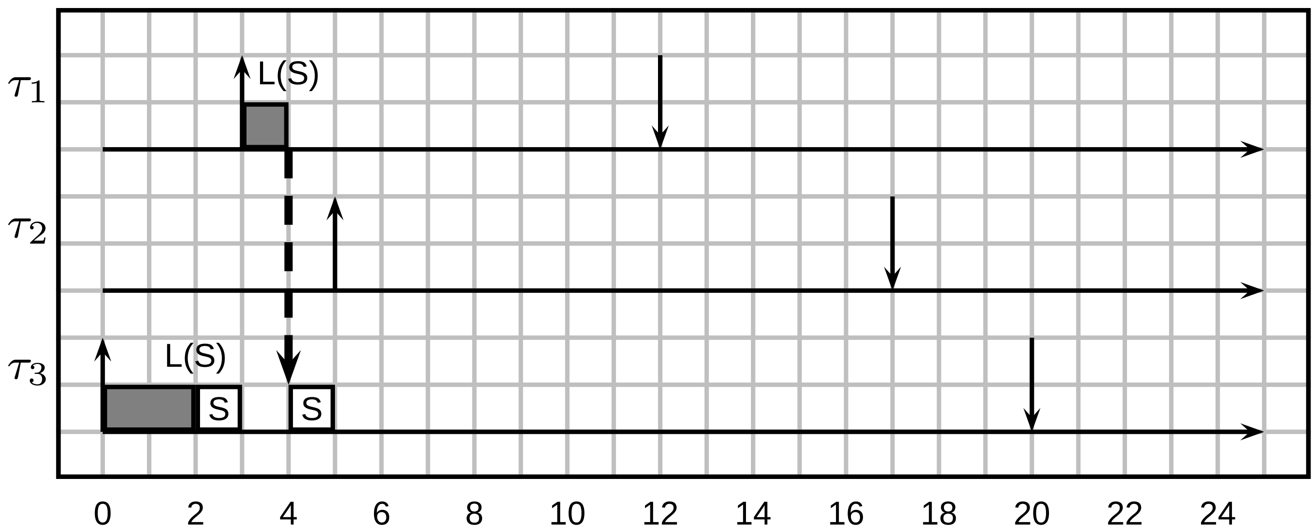
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



  - Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

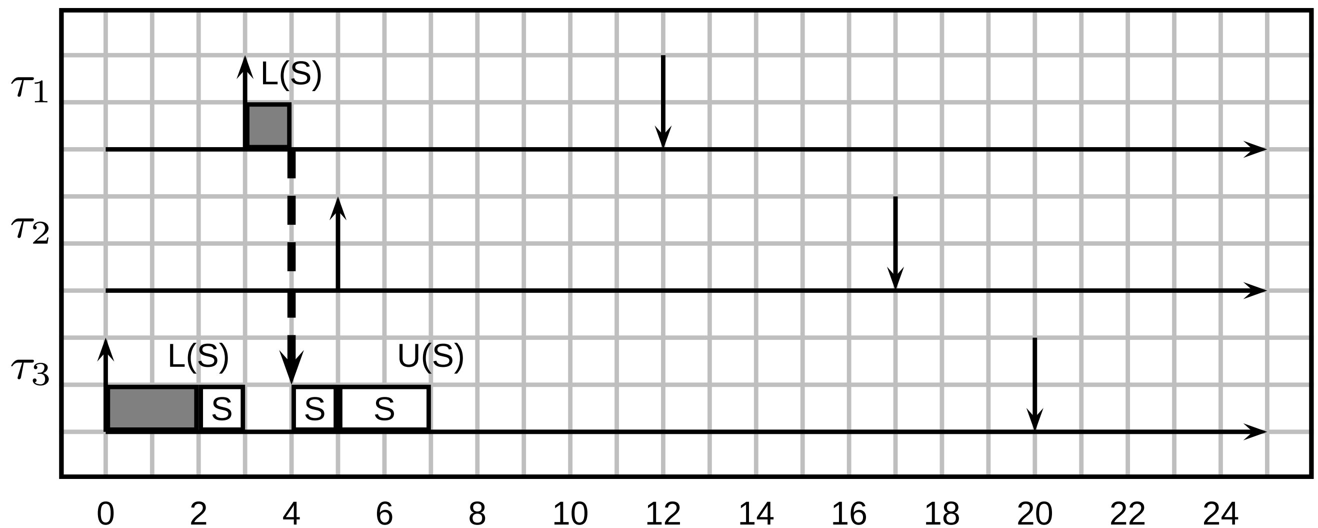
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



  - Task τ_3 inherits the priority of τ_1
  - Task τ_2 cannot preempt τ_3 $(p_2 < p_1)$

The Priority Inheritance protocol

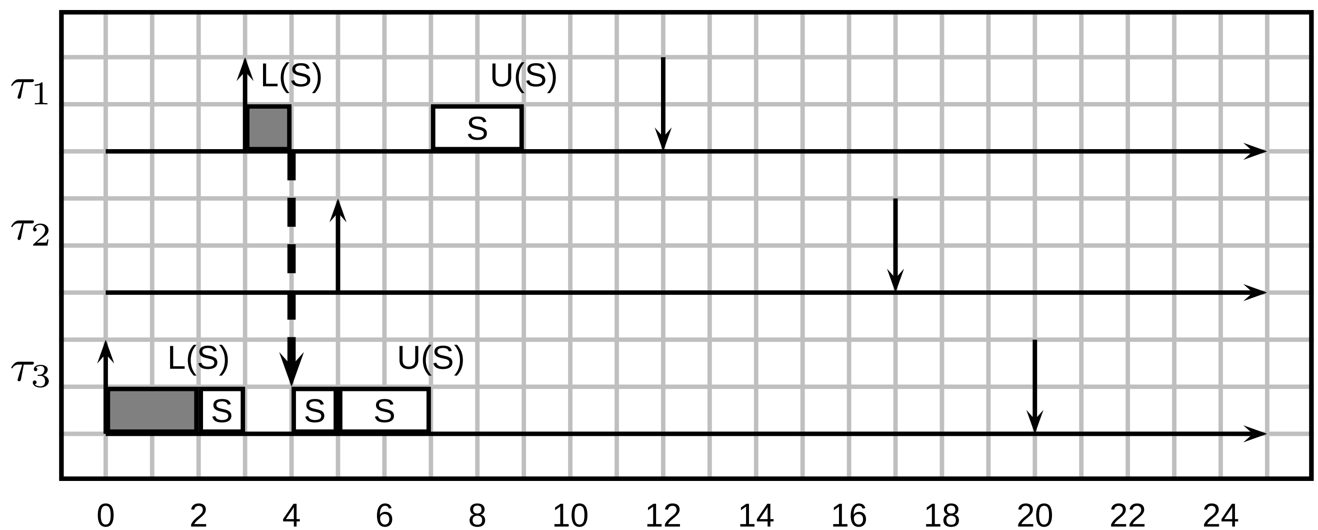
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



- Task τ_3 inherits the priority of τ_1
- Task τ_2 cannot preempt τ_3 ($p_2 < p_1$)

The Priority Inheritance protocol

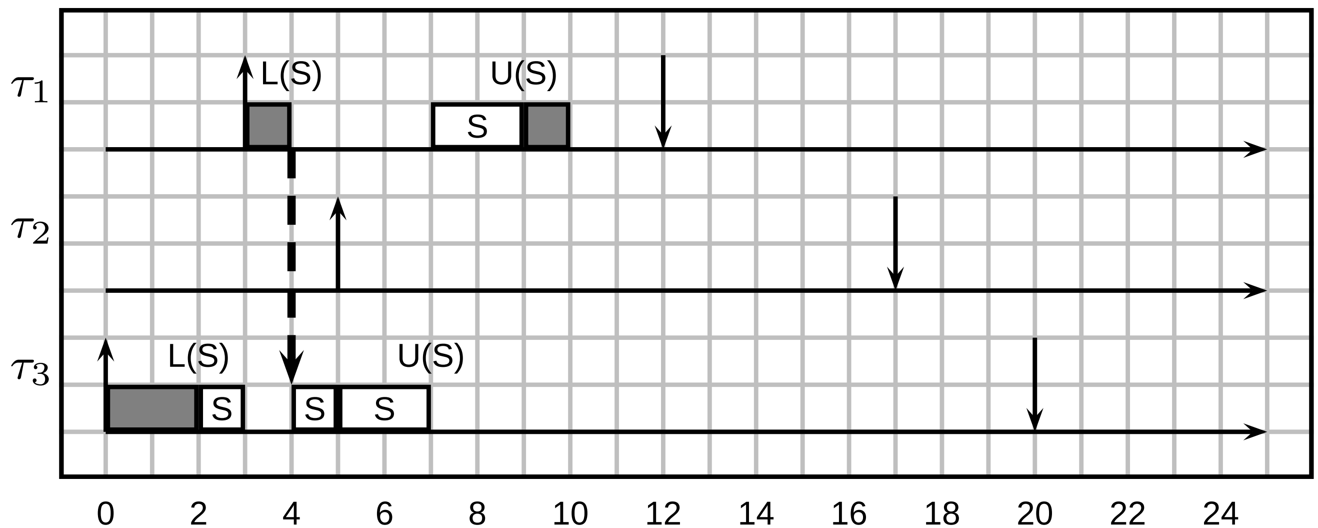
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



  - Task τ_3 inherits the priority of τ_1
  - Task τ_2 cannot preempt τ_3 ($p_2 < p_1$)

The Priority Inheritance protocol

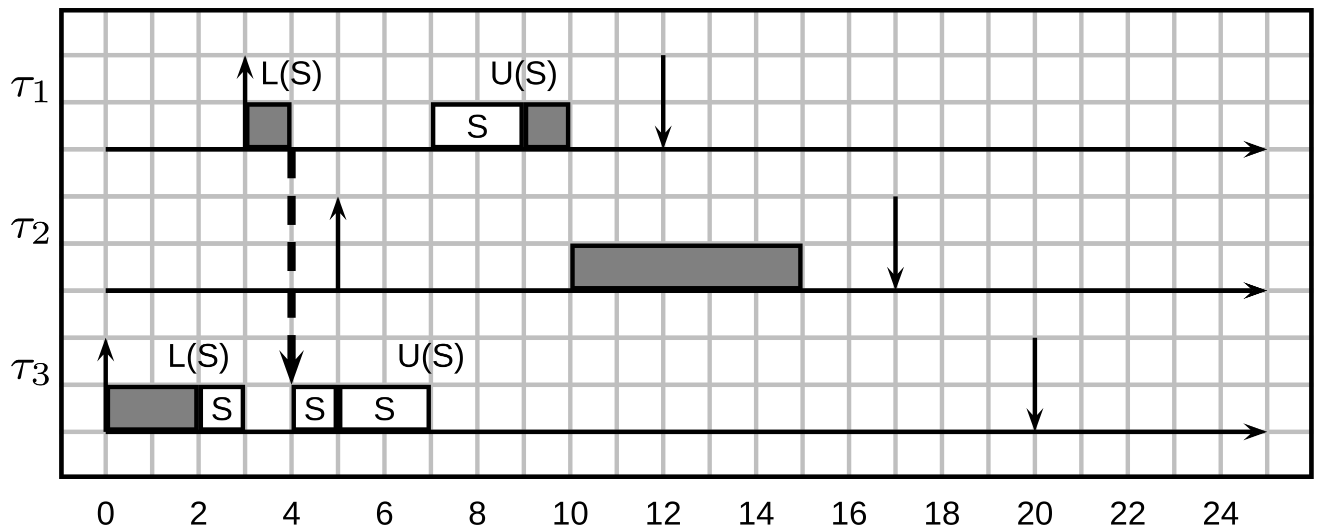
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



- Task τ_3 inherits the priority of τ_1
- Task τ_2 cannot preempt τ_3 ($p_2 < p_1$)

The Priority Inheritance protocol

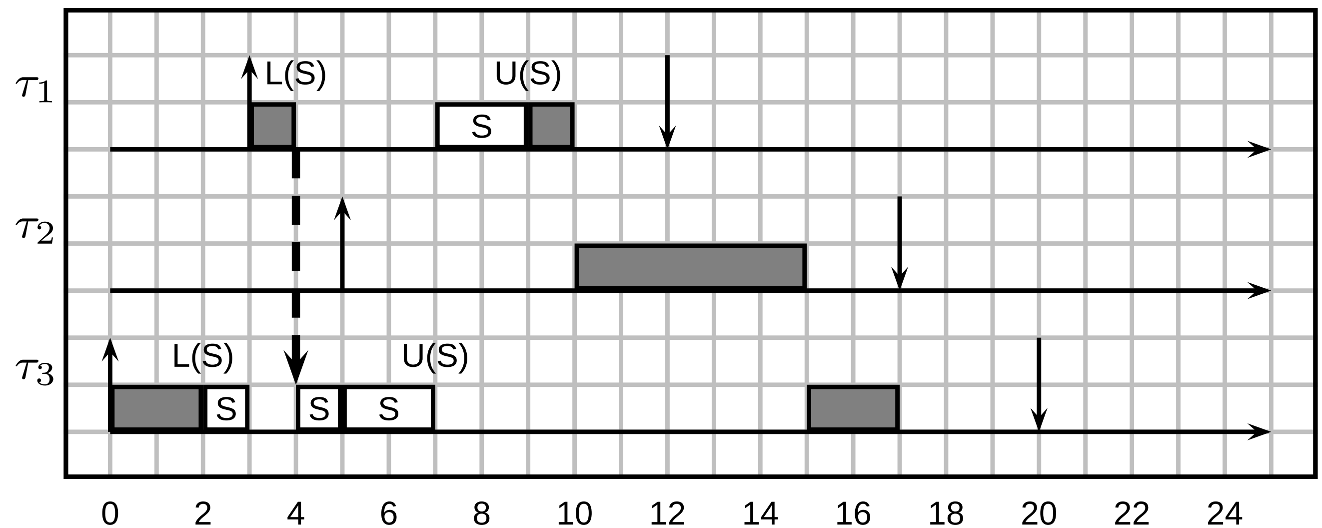
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



  - Task τ_3 inherits the priority of τ_1
  - Task τ_2 cannot preempt τ_3 ($p_2 < p_1$)

The Priority Inheritance protocol

- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



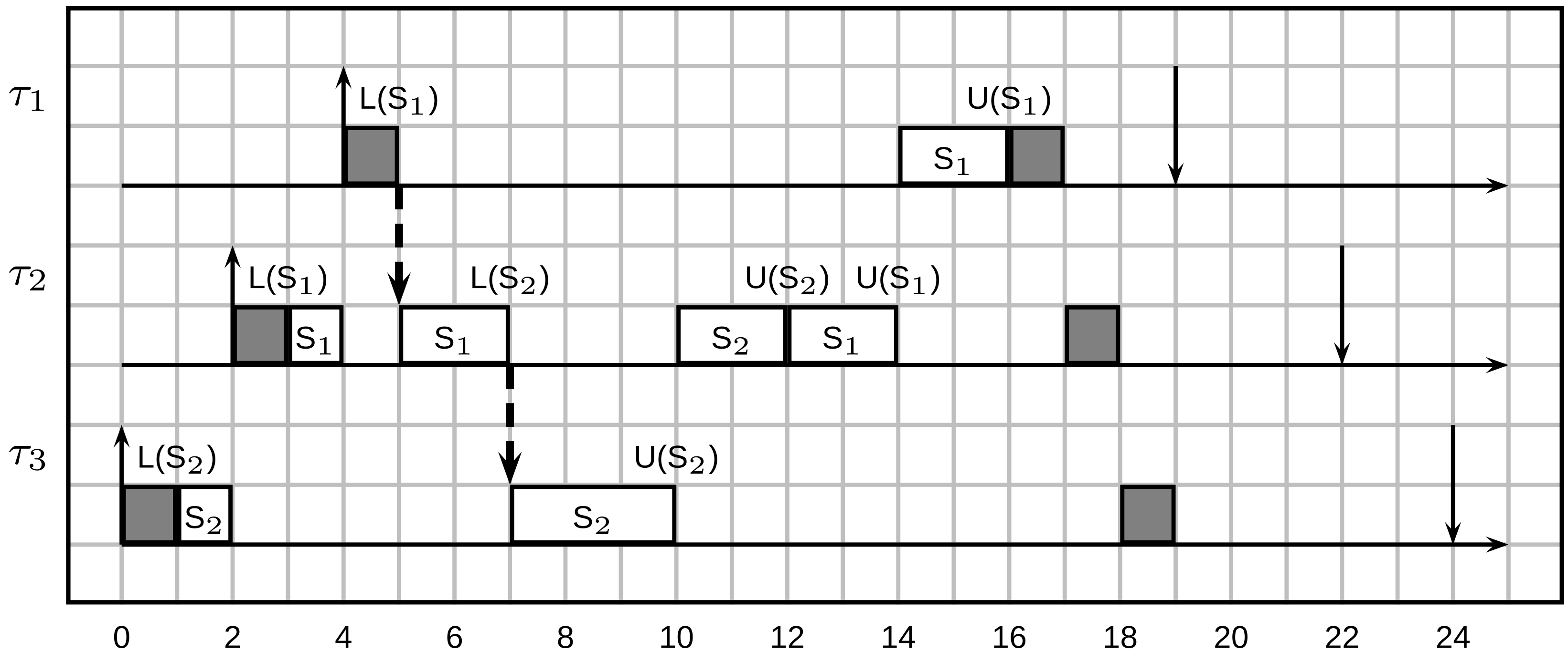
  - Task τ_3 inherits the priority of τ_1
  - Task τ_2 cannot preempt τ_3 ($p_2 < p_1$)

Nested critical sections

- Critical sections can be nested:
  - it means that, while a task τ is accessing a resource S_1, it can lock a resource S_2.
- When critical sections are nested, we can have *multiple inheritance*

Multiple inheritance

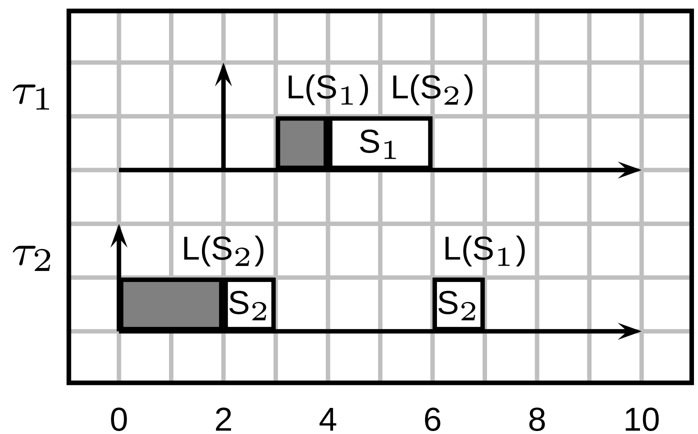
- Task τ_1 uses resource S_1; Task τ_2 uses S_1 and S_2 nested inside S_1; Task τ_3 uses only S_2



- At time $t = 7$ τ_3 inherits the priority of τ_2, which at time 5 had inherited the priority of τ_1. Hence, the priority of τ_3 is p_1.

Deadlock problem

- Nested critical sections $\rightarrow$ possible deadlock
  - Two tasks can be blocked waiting for each other
- The priority inheritance protocol *does not* automatically avoid deadlocks, as shown in the following example (τ_1 uses S_2 inside S_1, while τ_2 uses S_1 inside S_2)



- While τ_1 is blocked on S_2, which is held by τ_2, τ_2 is blocked on S_1 which is held by τ_1: **deadlock!**

Deadlock avoidance

- In the previous example, the priority inheritance protocol does not help (why should it?)
- To avoid deadlocks, it is possible to use a strategy for nested critical section
  - The problem is due to the fact that resouces are accessed in a random order by τ_1 and τ_2
  - One possibility is to decide an order a-priori *before writing the program*. For example that resources must be accessed in the order given by their index (S_1 before S_2 before S_3, and so on)
  - With this rule, task τ_2 is not legal because it accesses S_1 inside S_2, violating the ordering
  - If τ_2 accesses the resources in the correct order (S_2 inside S_1, the deadlock is automatically avoided)

Some PI Properties

- Summarising, the main rules are the following:
  - If a task τ_i blocks on a resource protected by a mutex semaphore S, and the resource is locked by task τ_j, then τ_j *inherits* the priority of τ_i
  - If τ_j itself blocks on another semaphore by a task τ_k, then τ_k inherits the priority of τ_i (*multiple inheritance*)
  - If τ_k is blocked, the chain of blocked tasks is followed until a non-blocked task is found that inherits the priority of τ_i
  - When a task unlocks a semaphore, it returns to the priority it had when locking it

Maximum Blocking Time for PI

- We only consider *non nested* critical sections...
  - In presence of multiple inheritance, the computation of the blocking time becomes very complex
  - Non nested critical sections $\rightarrow$ multiple inheritance cannot happen, and the computation of the blocking time becomes simpler
- Two important theorems:
  - **Theorem 1** if PI is used, a job can be blocked only once on each different semaphore
  - **Theorem 2** if PI is used, a job can be blocked by a lower priority task for at most the duration of one critical section
- $\Rightarrow$ a job can be blocked more than once, but only once per each resource and once by each lower priority task

Blocking Time Computation

- We must build a *resource usage table*
  - A task per row, in decreasing order of priority
  - A resource per column
  - Cell (i, j) contains $\xi_{i,j}$, i.e. the length of the longest critical section of task τ_i on resource S_j, or 0 if the task does not use the resource
- A task can be blocked only by lower priority tasks:
  - Then, for each task (row), we must consider only the rows below (tasks with lower priority)
- A task can be blocked only on resources that it uses directly, or used by higher priority tasks (*indirect blocking*):
  - For each task, only consider columns on which it can be blocked (used by itself or by higher priority tasks)

Example - 1

	S_1	S_2	S_3	B
τ_1	2	0	0	?
τ_2	0	1	0	?
τ_3	0	0	2	?
τ_4	3	3	1	?
τ_5	1	2	1	?

- Let's start from B_1
- τ_1 can be blocked only on S_1. Therefore, we must consider only the first column, and take the maximum, which is 3. Therefore, $B_1 = 3$.

Example - 2

	S_1	S_2	S_3	B
τ_1	2	0	0	3
τ_2	0	1	0	?
τ_3	0	0	2	?
τ_4	3	3	1	?
τ_5	1	2	1	?

- τ_2 can be blocked on S_1 (*indirect blocking*) and on S_2
- Consider all cases where two distinct lower priority tasks in $\{\tau_3, \tau_4, \tau_5\}$ access S_1 and S_2, sum the two contributions, and take the maximum;
  - τ_4 on S_1 and τ_5 on S_2: $\rightarrow 5$
  - τ_4 on S_2 and τ_5 on S_1: $\rightarrow 4$

Example - 3

	S_1	S_2	S_3	B
τ_1	2	0	0	3
τ_2	0	1	0	5
τ_3	0	0	2	?
τ_4	3	3	1	?
τ_5	1	2	1	?

- τ_3 can be blocked on all 3 resources
- The possibilities are:
  - τ_4 on S_1 and τ_5 on S_2: $\rightarrow 5$;
  - τ_4 on S_2 and τ_5 on S_1 or S_3: $\rightarrow 4$;
  - τ_4 on S_3 and τ_5 on S_1: $\rightarrow 2$;
  - τ_4 on S_3 and τ_5 on S_2 or S_3: $\rightarrow 3$;

Example - 4

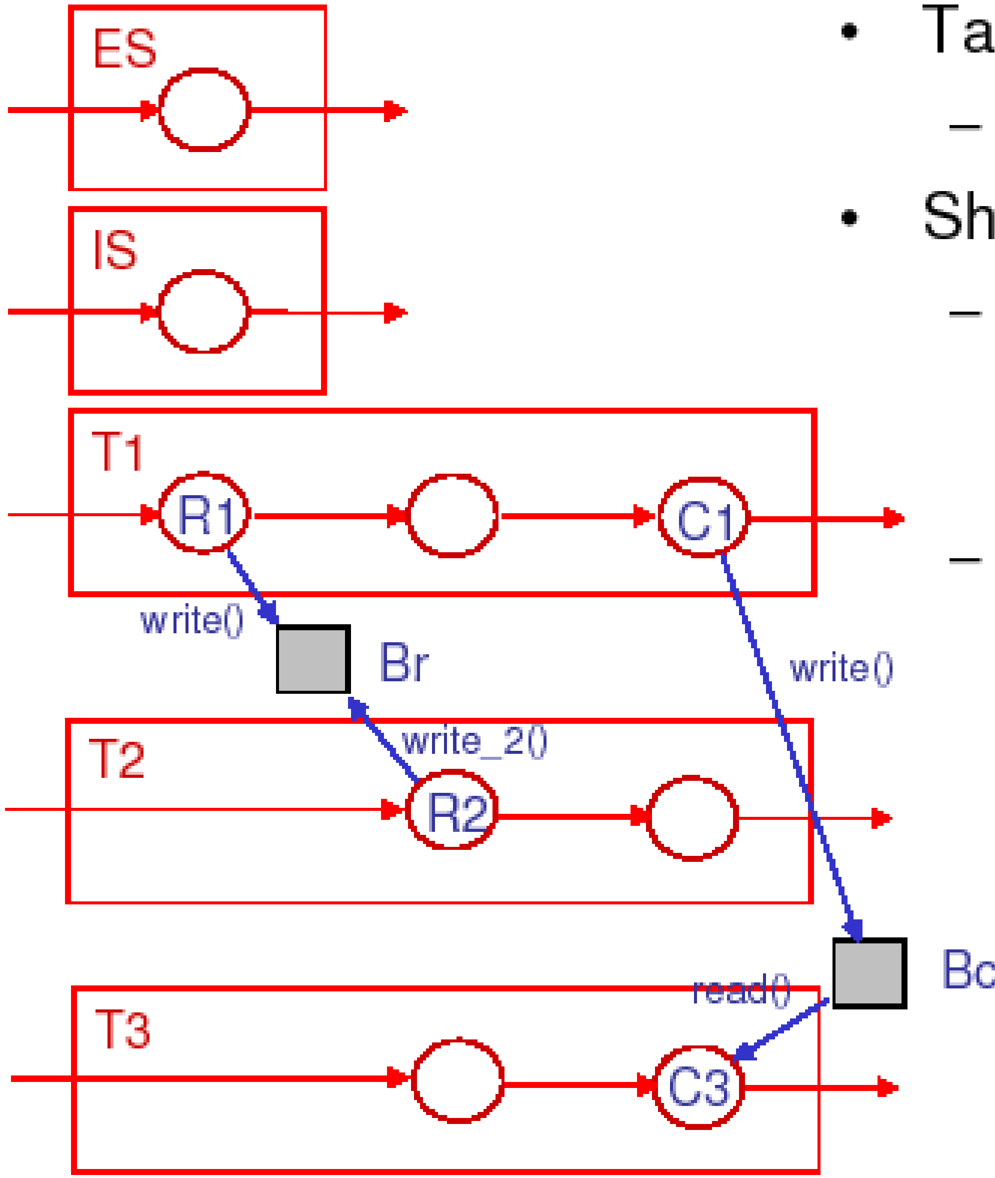
	S_1	S_2	S_3	B
τ_1	2	0	0	3
τ_2	0	1	0	5
τ_3	0	0	2	5
τ_4	3	3	1	?
τ_5	1	2	1	?

- τ_4 can be blocked on all 3 resources. We must consider all columns; however, it can be blocked only by τ_5.
- The maximum is $B_4 = 2$.
- τ_5 cannot be blocked by any other task (because it is the lower priority task!); $B_5 = 0$;

Example: Final result

	S_1	S_2	S_3	B
τ_1	2	0	0	3
τ_2	0	1	0	5
τ_3	0	0	2	5
τ_4	3	3	1	2
τ_5	1	2	1	0

An example



- Task
  - 5 Tasks
- Shared resources
  - Results buffer
    - Used by R1 and R2
    - R1 (2 ms) R2 (20 ms)
  - Communication buffer
    - Used by C1 and C3
    - C1 (10 ms) C3 (10 ms)

Example of blocking time computation

	C	T	D	$\xi_{1,i}$	$\xi_{2,i}$
ES	5	50	6	0	0
IS	10	100	100	0	0
τ_1	20	100	100	2	10
τ_2	40	150	130	20	0
τ_3	100	350	350	0	10

Table of resource usage

	$\xi_{1,i}$	$\xi_{2,i}$	B_i
ES	0	0	?
IS	0	0	?
τ_1	2	10	?
τ_2	20	0	?
τ_3	0	10	?

Computation of the blocking time

	$\xi_{1,i}$	$\xi_{2,i}$	B_i
ES	0	0	0
IS	0	0	0
τ_1	2	10	?
τ_2	20	0	?
τ_3	0	10	0

- Task ES and IS do not experience any blocking since neither do they use shared resource (direct blocking) nor are there tasks having higher priority that do so (indirect blocking)
- Task τ_3 does not experience any blocking time either (since it is the one having the lowest priority)

Computation of the blocking time

	$\xi_{1,i}$	$\xi_{2,i}$	B_i
ES	0	0	0
IS	0	0	0
τ_1	2	10	30
τ_2	20	0	?
τ_3	0	10	0

- For task τ_1 we have to consider both columns 1 and 2 since it uses both resources
- The possibilities are:
  - τ_2 on S_1 and τ_3 on S_2: $\rightarrow 30$;

Computation of the blocking time

	$\xi_{1,i}$	$\xi_{2,i}$	B_i
ES	0	0	0
IS	0	0	0
τ_1	2	10	30
τ_2	20	0	10
τ_3	0	10	0

- For task τ_2 we have to consider column 2 since it is associated to the only resource used by tasks having both higher and lower priority than τ_2 (τ_2 itself uses resource 1 which is not used by any other task with lower priority)
- The possibilities are:
  - τ_3 on S_2: $\rightarrow 10$;

The response times

	C	T	D	$\xi_{1,i}$	$\xi_{2,i}$	B_i	R_i
ES	5	50	6	0	0	0	5+0+0=5
IS	10	100	100	0	0	0	10+0+5=15
τ_1	20	100	100	2	10	30	20+30+20=70
τ_2	40	150	130	20	0	10	40+10+40=90
τ_3	100	350	350	0	10	0	100+0+200=300

Response Time Analysis

- We have seen the schedulability test based on response time analysis

$$R_i = C_i + B_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$$

- There are also other options
- For instance we can apply the following sufficient test: The system is schedulable if

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

Time Demand Approach

- In a task set $\mathcal{T}$ composed of independent and periodic tasks, τ_i is schedulable (for all possible phasings) **iff**

$$\exists 0 \leq t \leq D_i : W_i(0,t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h \leq t$$

- Introducing blocking times B_i, $\tau_i \in \mathcal{T}$ is schedulable **if** exists $0 \leq t \leq D_i$ such that

$$W_i(0,t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h \leq t - B_i$$

Time Demand Approach - 2

- As usual, we can define
  - $W_i(t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h$
  - $L_i(t) = \frac{W_i(t)}{t}$
  - $L_i = \min_{0 \leq t \leq D_i} L_i(t) + \frac{B_i}{t}$
- The task set is schedulable if $\forall i, L_i \leq 1$
- Again, we can compute L_i by only considering the scheduling points