Resource Sharing Protocols

Advanced Real Time Operating Systems

Luca Abeni

`luca.abeni@santannapisa.it`

Again on Preemptable Kernels

- Preemptable Linux kernel $\rightarrow$ reduces L^N
  - Is it just a hack?
- Theoretical foundation: spinlocks end up using NPP
  - Oh, no! Real-time jargon, once again!
  - So, what is NPP?
- Latencies can still be high... Why?
  - Once again, theory can explain...
- Two possible ways around: HLP and PI!
  - HLP? PI? WTH!!!

Reconciliating Practice and (RT) Theory

- Latency: can be modelled as a blocking time
- RT Theory $\rightarrow$ lot of work on blocking times
  - Mainly seen as due to priority inversion
  - In OS kernels, blocking times due to someting different...
  - ...But to re-use RT theory, let's see them as priority inversion due to kernel critical sections!
- Non-preemptable (monolithic) kernels: the kernel is a critical section!
- Preemptable kernels: fine-grained critical sections inside the kernel
  - Issue: they affect even tasks not using syscalls / IRQs!

Dealing with Priority Inversion

- Priority inversion can be reduced...
  - ...But how?
  - By introducing an appropriate *resource sharing protocol* (concurrency protocol)
- Provides 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/spinlocks (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

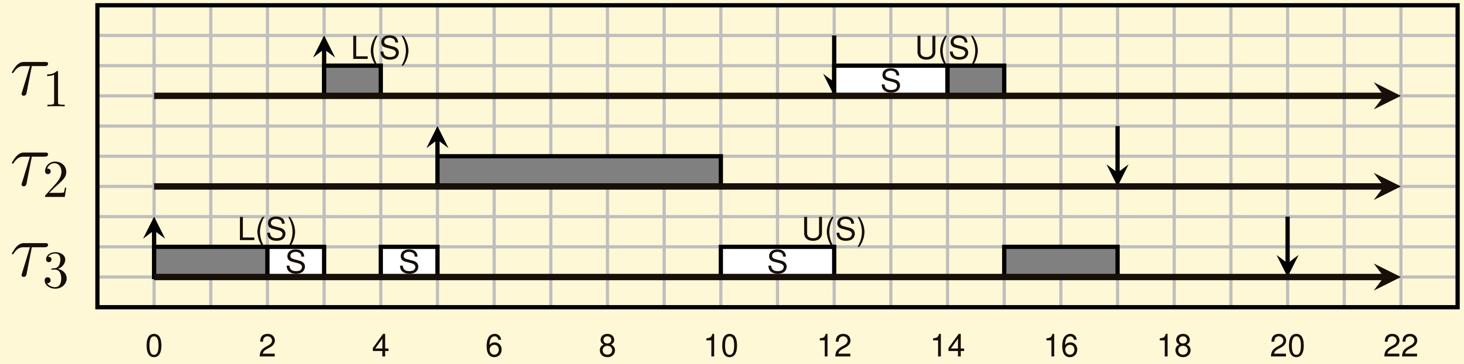
Non Preemptive Protocol (NPP)

- 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

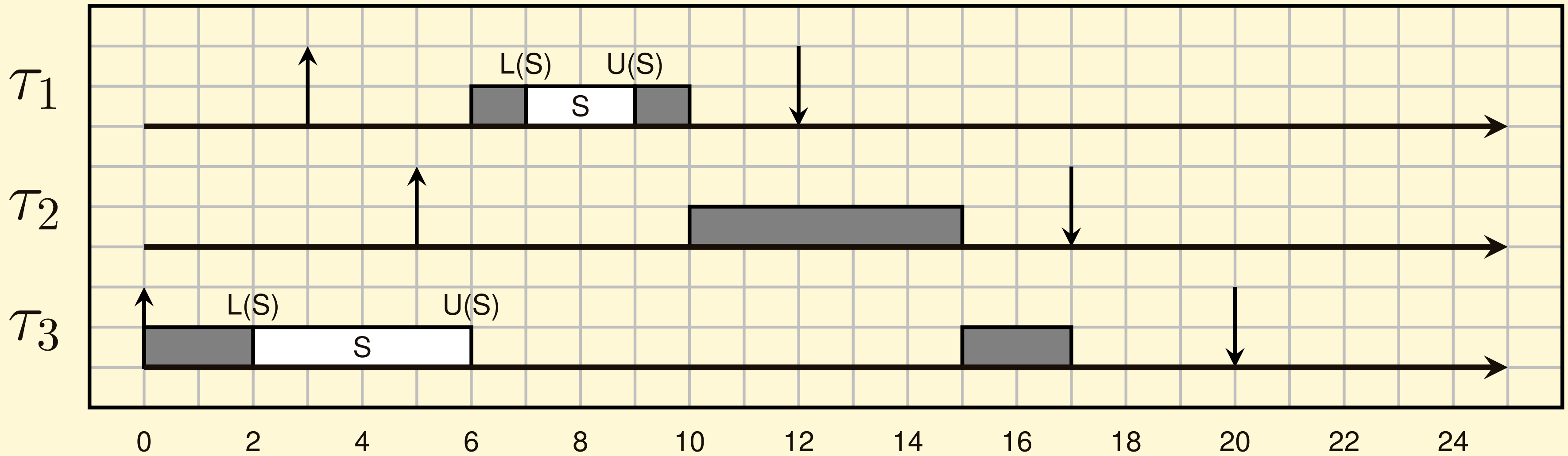
- Advantages: *simplicity*
- Drawbacks: tasks which are not involved in a critical section suffer blocking

NPP Example

- Remember the previous example...



- Using NPP, we have:

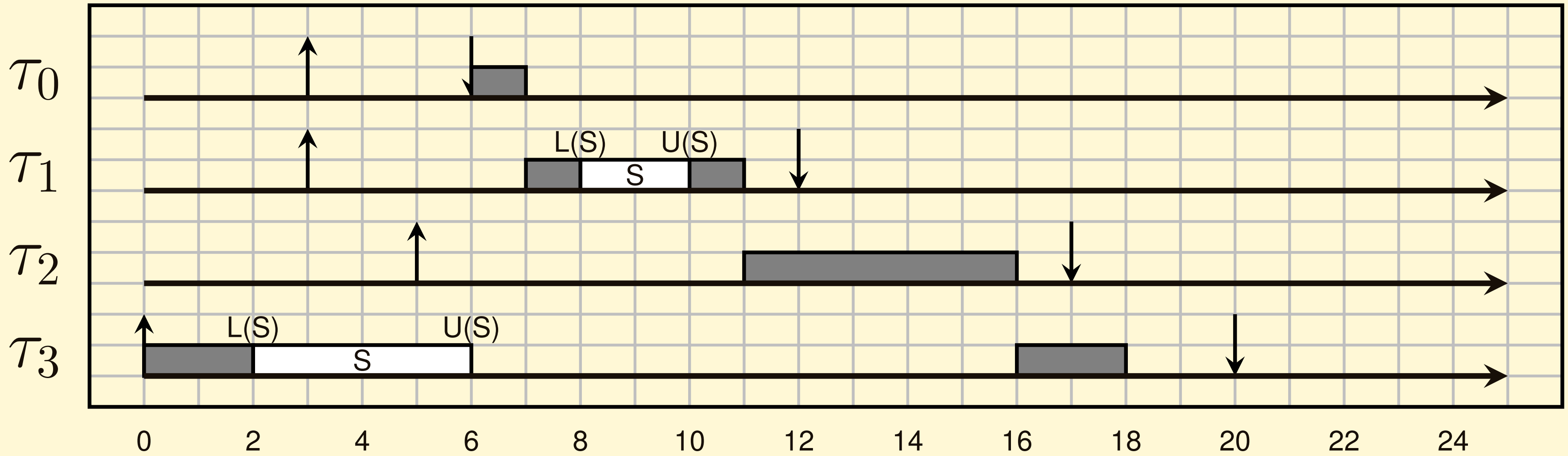


Some Observations

- 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 experiences some blocking, but it does not use any resource
  - *Indirect blocking*: 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
  - Must be computed and taken into account in the admission test as any other blocking time
- What's the maximum blocking time B_i for τ_i?

A Problem with NPP

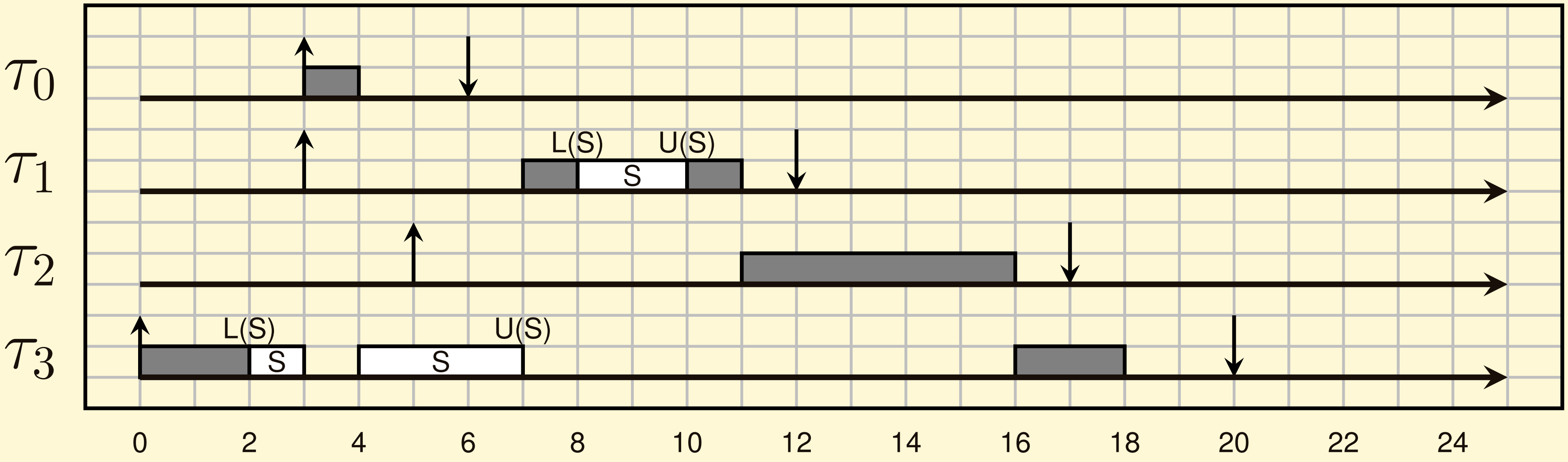
- Consider the following example, with $p_0 > p_1 > p_2 > p_3$.



- τ_0 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_1' priority)

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:

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

  - 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 interference 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

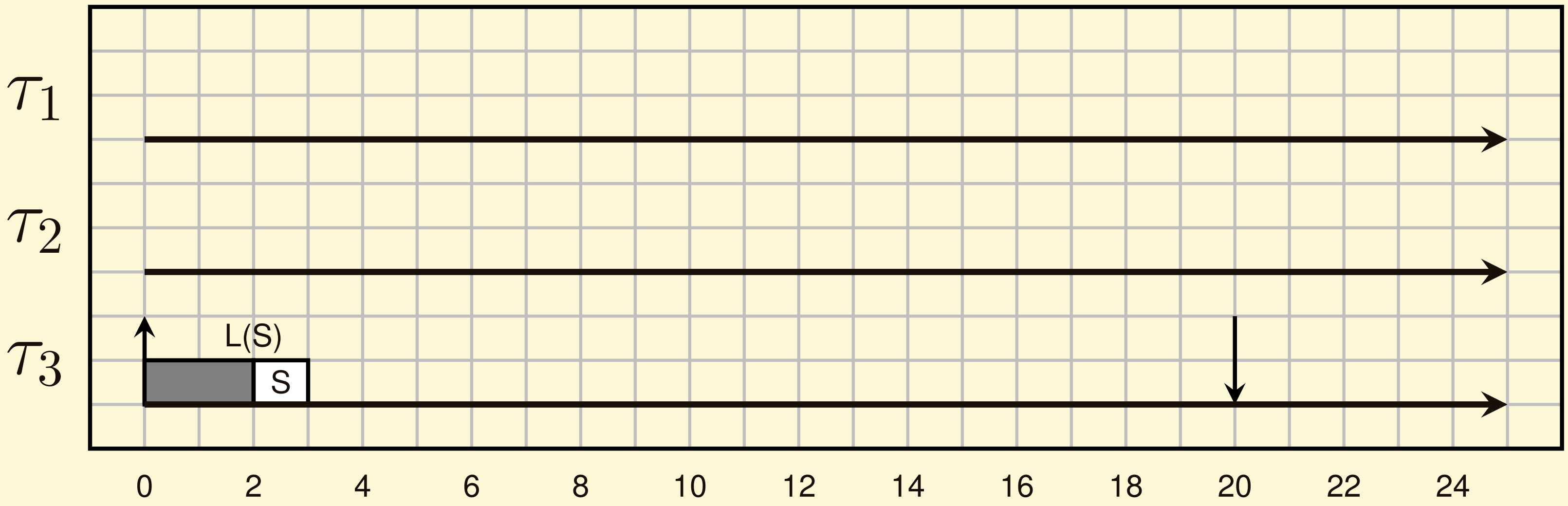
Task	C_i	T_i	$\xi_{i,1}$	D_i	B_i
τ_1	20	70	0	30	2
τ_2	20	80	1	45	2
τ_3	35	200	2	130	0

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

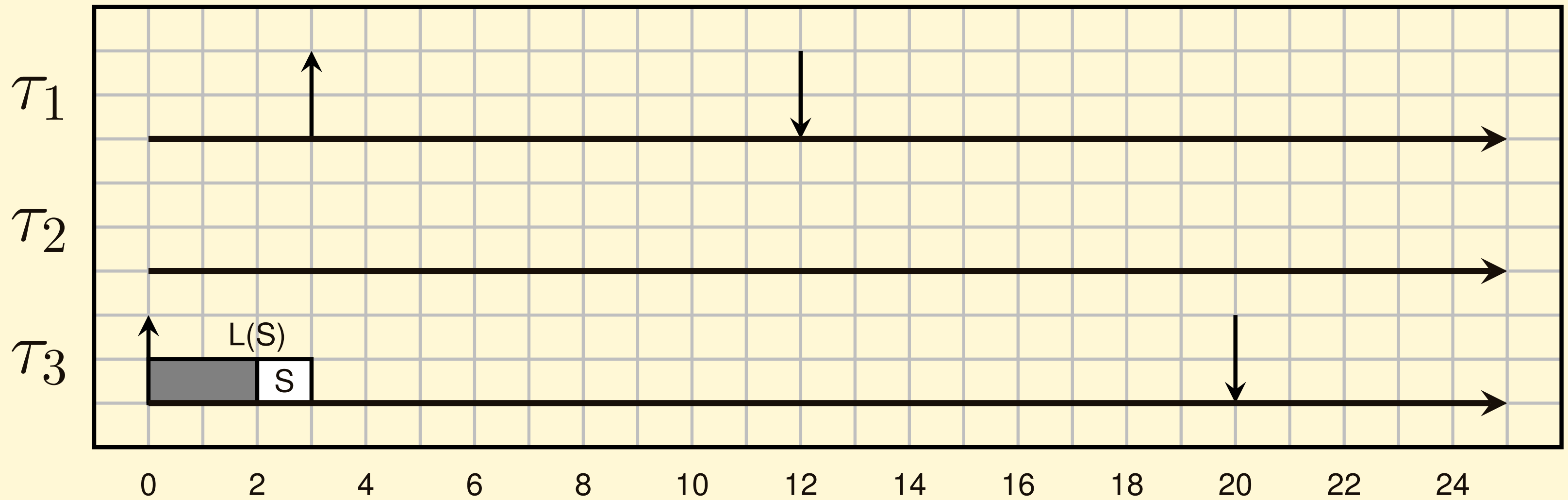
The Priority Inheritance protocol

- 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



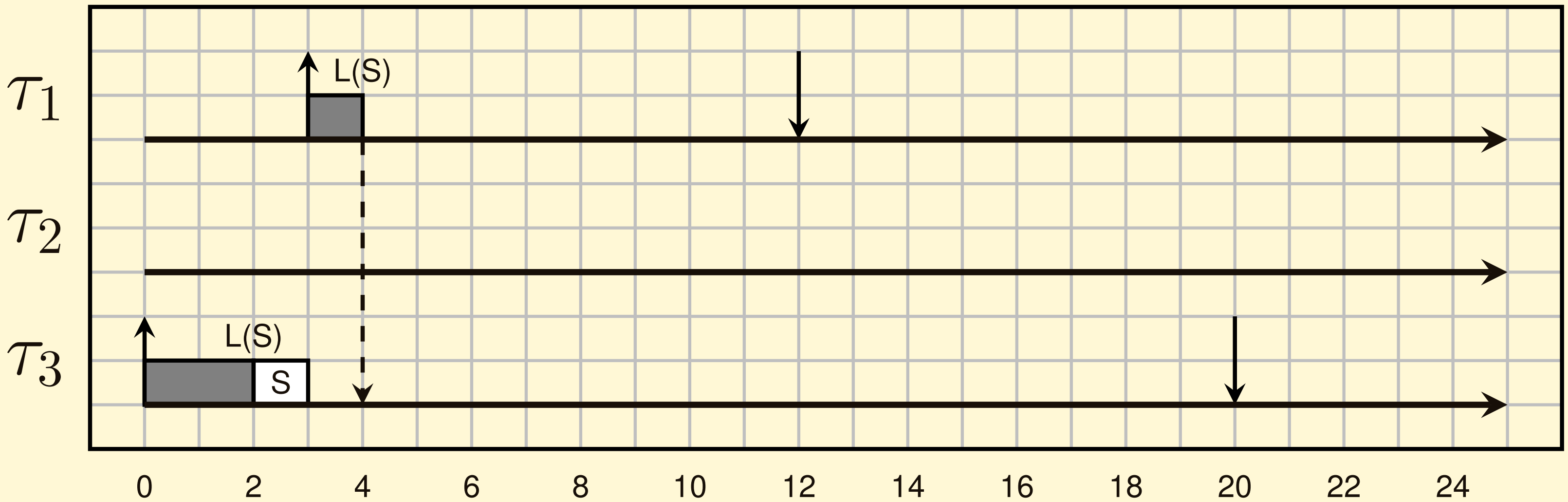
The Priority Inheritance protocol

- 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



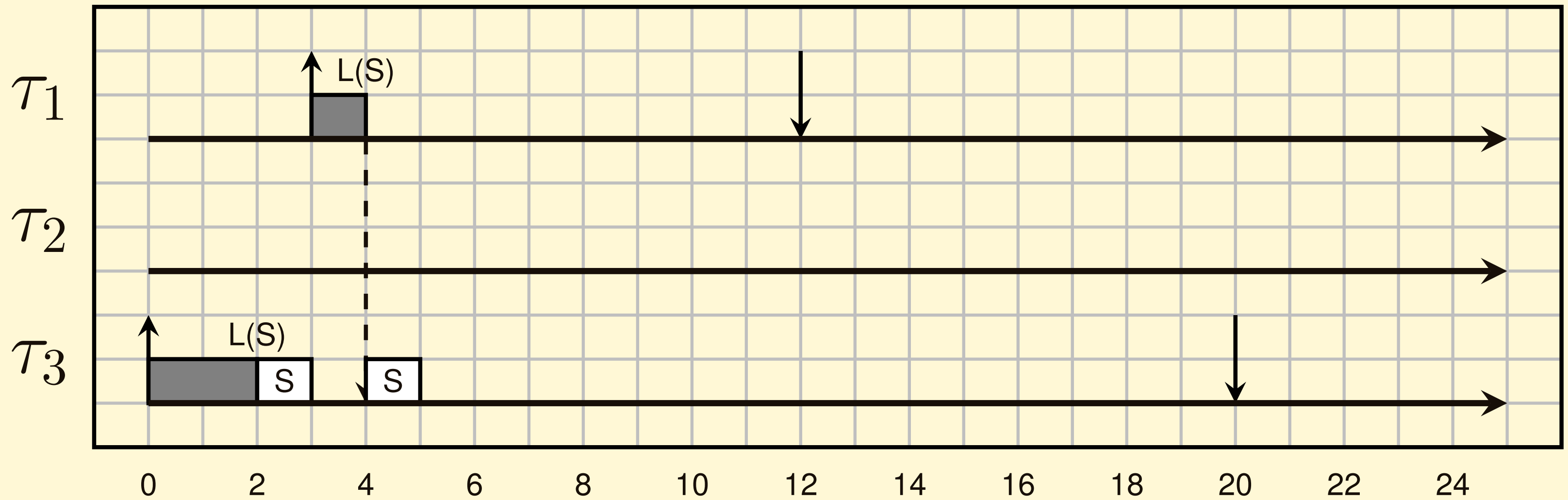
The Priority Inheritance protocol

- 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

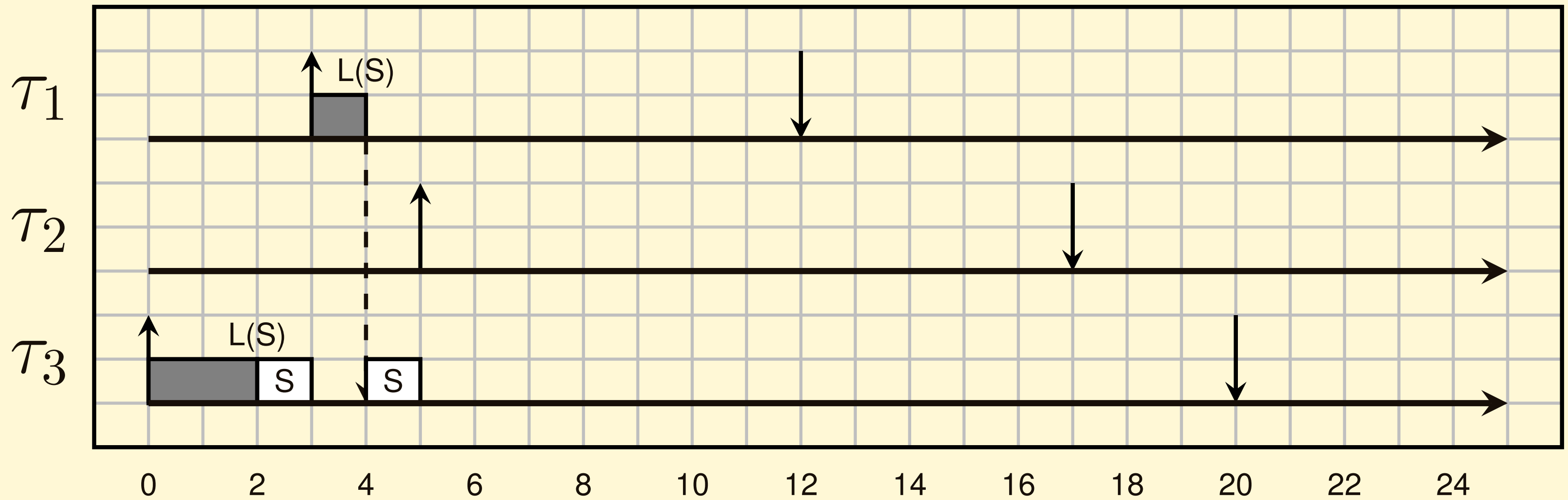
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



  - Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

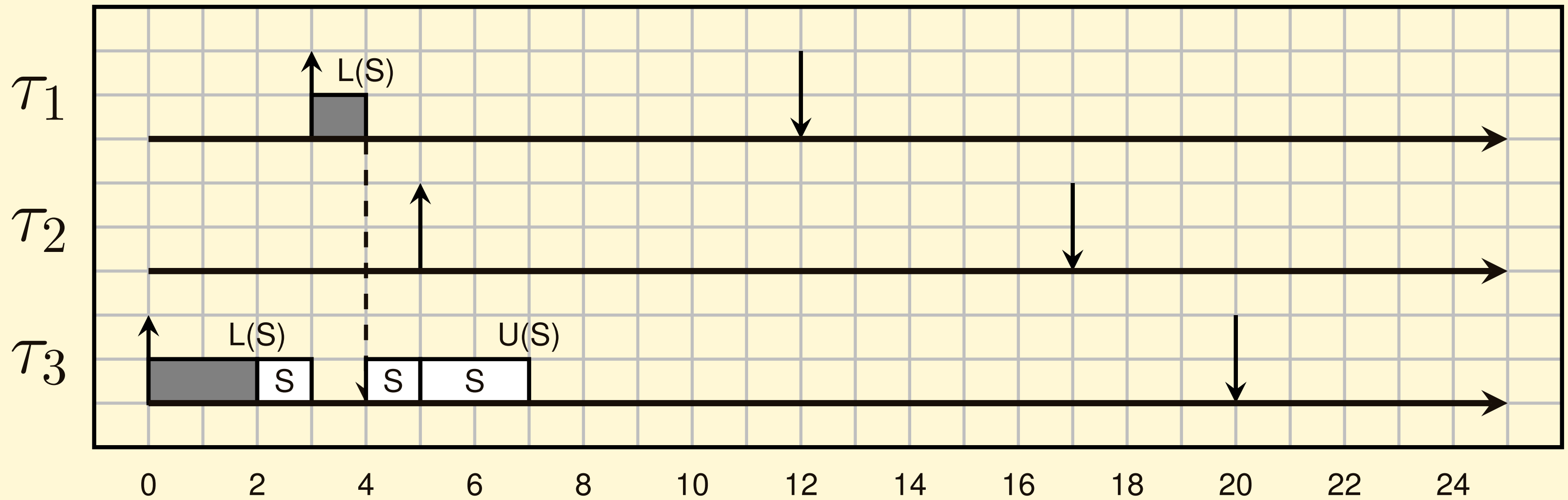
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

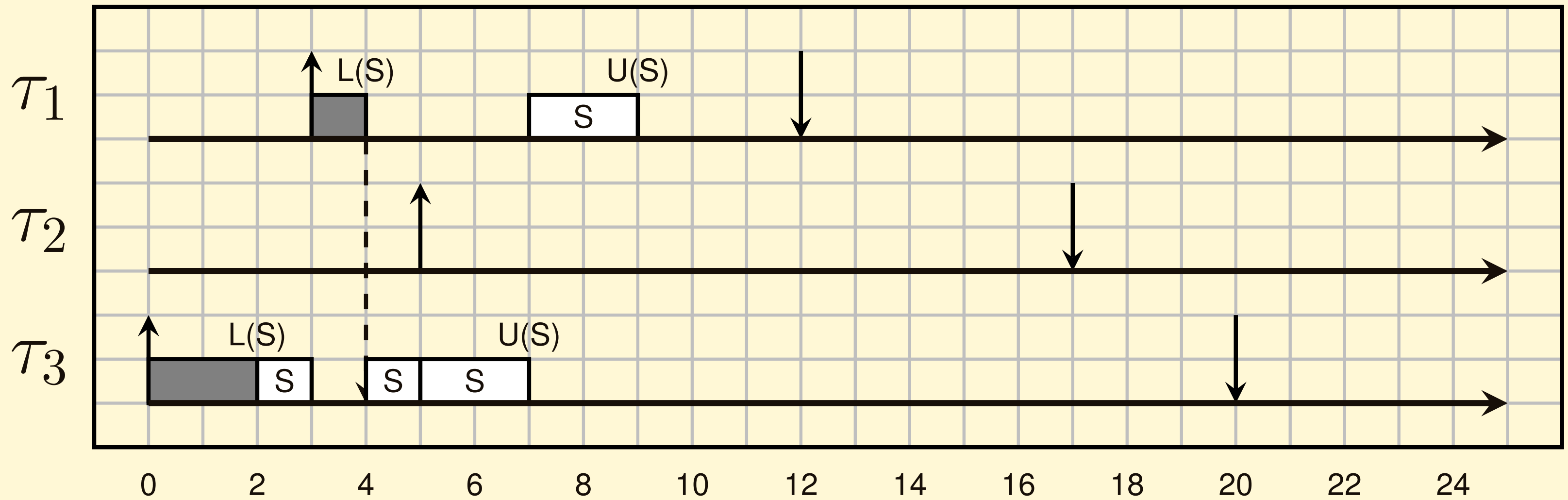
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

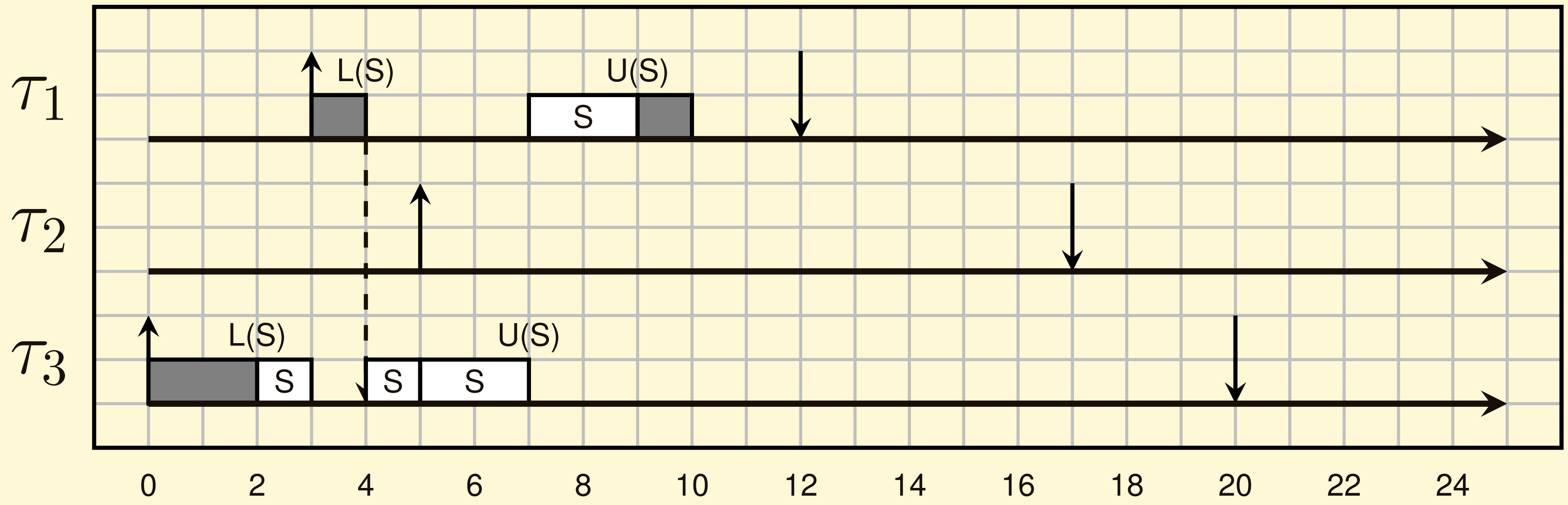
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

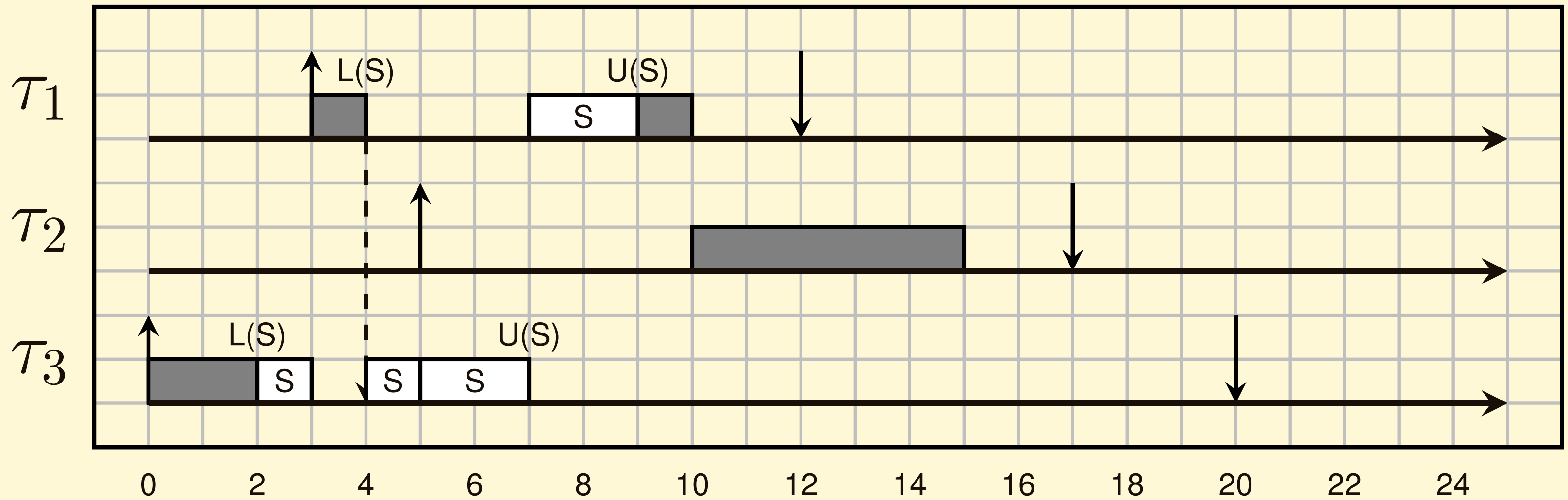
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

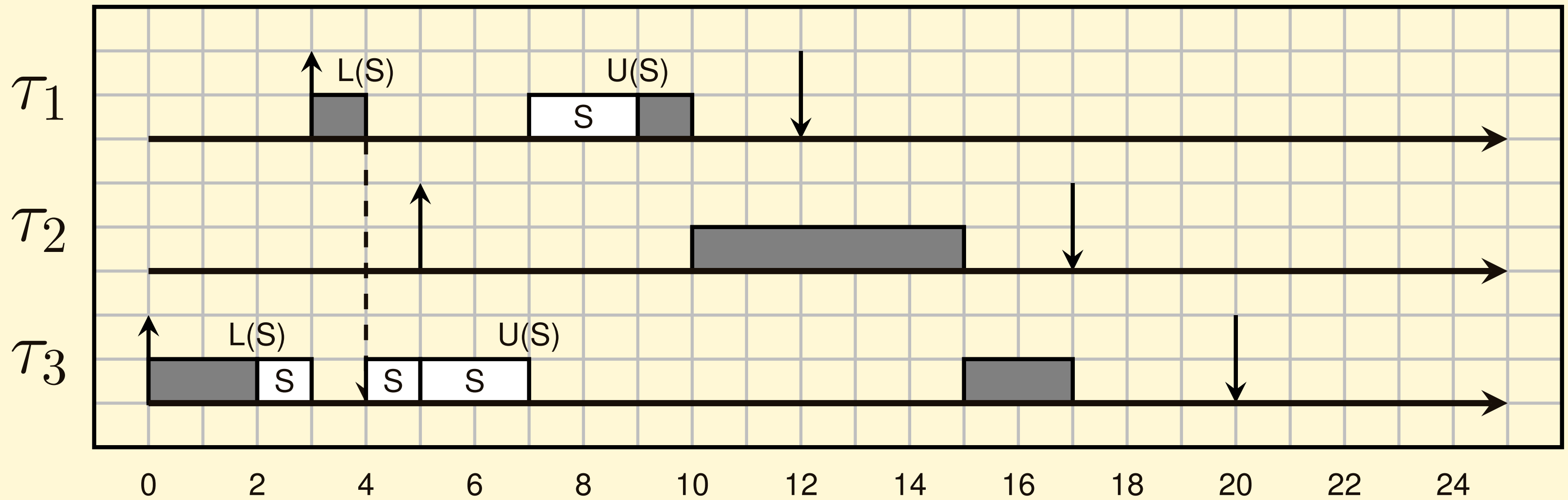
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



- 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 ($p_2 < p_1$)

Some PI Properties

- Summarising, the main rules are the following:
  - If a task τ_i blocks on a resource protected by a mutex 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 mutex 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 mutex, it returns to the priority it had when locking it