Real Time Operating Systems

Shared Resources

Luca Abeni

`luca.abeni@unitn.it`

Interacting Tasks

- Until now, 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

- Example: control application composed by three periodic tasks
  - τ_1 reads the data from the sensors and applies a filter. The results 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
  - τ_3 reads the outputs and writes on an actuator
- All of the three tasks access data in shared memory
- Conflicts on accessing this data concurrently
  - $\Rightarrow$ The data structures can become inconsistent

Task Intraction Paradigms - Private Resources

- How to handle interactions between tasks?
  - Private Resources $\rightarrow$ Client / Server paradigm
  - Shared Resources
- Something like "processes vs threads"
- Let's start with processes...
- Private Resources
  - A *Resource Manager* (server task) per resource
  - Tasks needing to access a resource send a message to its manager
  - Interaction via IPC
  - Example: the X server

Task Intraction Paradigms - Shared Resources

- What about threads?
- Shared Resources
  - Must be accessed in *mutual exclusion*
  - Interaction via mutexes, semaphores, condition variables, . . .
- Real-Time analysis presente here: will focus on shared resources
  - We will use mutexes, not semaphores
  - Extensions to IPC based communication are 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;
- This is ok for enforcing data consistency...
- ...But what is the effect on real-time performance?
  - Assume that resources are protected by mutual exclusion semaphores (mutexes)
  - Why Mutexes and not semaphores? ...

Remember... - Some Definitions

- Task
  - Schedulable entity (thread or process)
  - Flow of execution
    - Object Oriented terminology: task $\equiv$ active object
    - Informally speaking: task $\equiv$ active entity that can perform operations on private or shared data
- Now, we need to model the “private or shared data”...
  - As said, focus on shared data

Key Concepts - Protected Objects

- Shared data: protected by mutexes ⇒ protected objects
- Protected Objects
  - Encapsulate shared information (Resources)
  - Passive objects (data) shared between different tasks
  - Operations on protected objects are mutually exclusive (this is why they are “protected”!)
- As said, protected by mutexes
  - Locking a mutex, a task “owns” the associate resource...
  - ...So, I can ask: “who is the owner of this resource”?

Shared Resources - Definition

- Shared Resource S_i
  - Used by multiple tasks
  - Protected by a *mutex* (*mut*ual *ex*clusion semaphore)
  - $1 \leftrightarrow 1$ relationship between resources and mutexes
    - Convention: S_i can be used to indicate either the resource or the mutex
- The system model must be extended according to this definition
  - Now, the system is not limited to a set of tasks...

Shared Resources - System Model

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

Blocking Time - 1

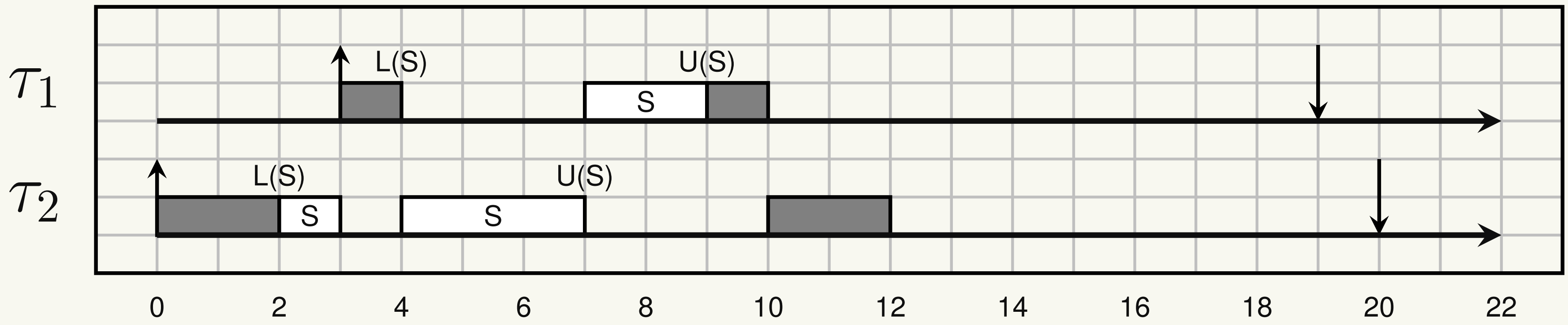
- Mutual exclusion on a shared resource can cause *blocking time*
  - When task τ_i tries to access a resource S already held from task τ_j, τ_i blocks
  - Blocking time: time between the instant when τ_i tries to access S (and blocks) and the instant when τ_j releases S (and τ_i unblocks)
- This is needed for implementing mutual exclusion, and cannot be avoided
  - The problem is that this blocking time might become unpredictable/too large...

Blocking Time - 2

- Blocking times 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!
- Schedulability tests must account for blocking times!
- Blocking times must be deterministic (and not too large!!!)

Blocking and Priority Inversion

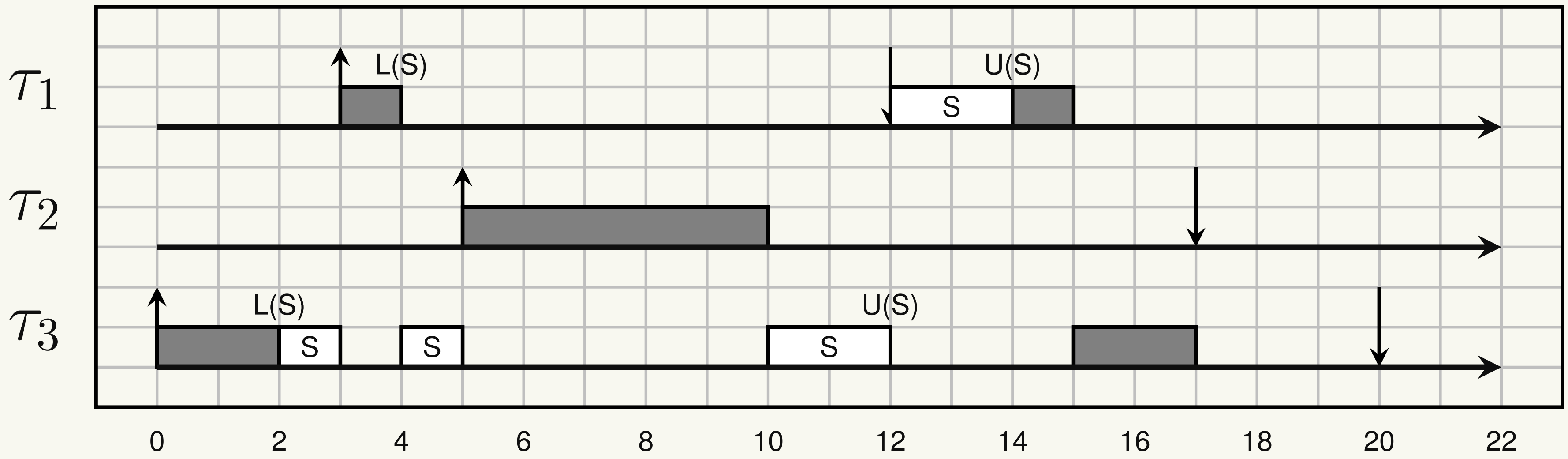
- Consider the following example, where $p_1 > p_2$.



- 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?

- Not only a theoretical problem; it happened for real
- 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. A high priority thread and a low priority thread were using the same software data structure (an “information bus”) protected by a mutex. The mutex 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 a medium priority thread while blocking the high priority thread on the mutex.
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.

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)
- 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 (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

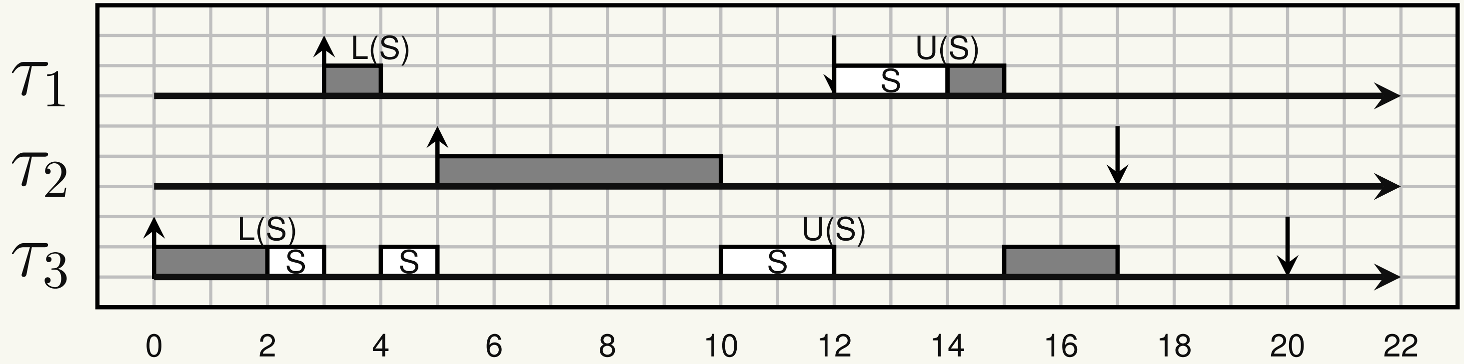
Non Preemptive Protocol (NPP)

- 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

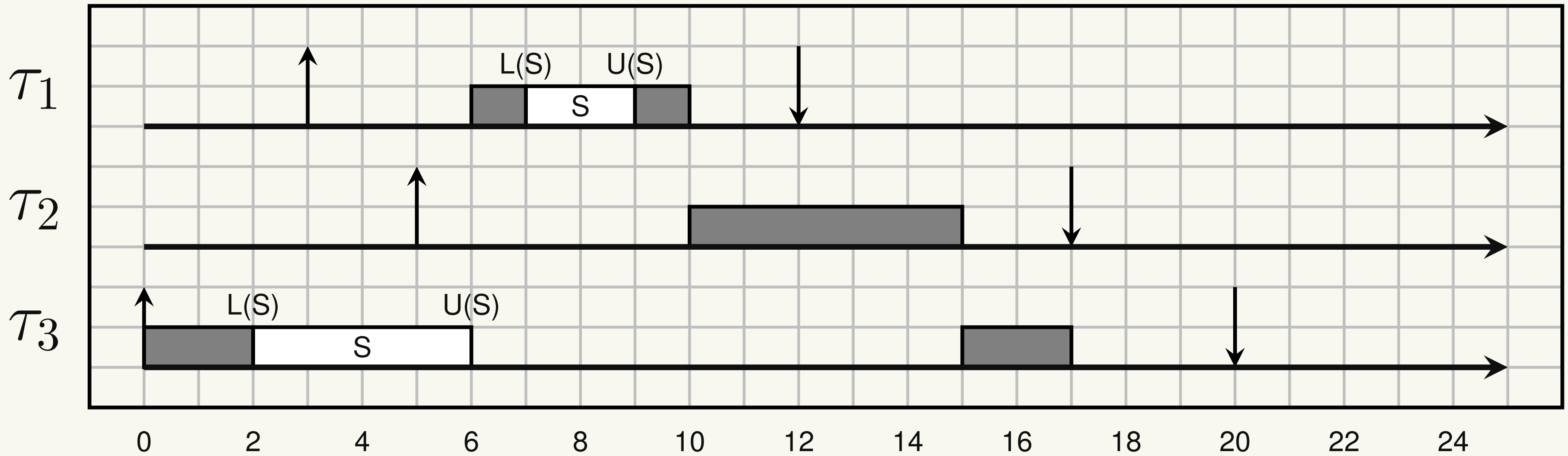
- Advantages: *simplicity*
- Drawbacks: tasks which are not involved in a critical section suffer blocking

NPP Example

- Remember the previous example...



- Using NPP, we have:

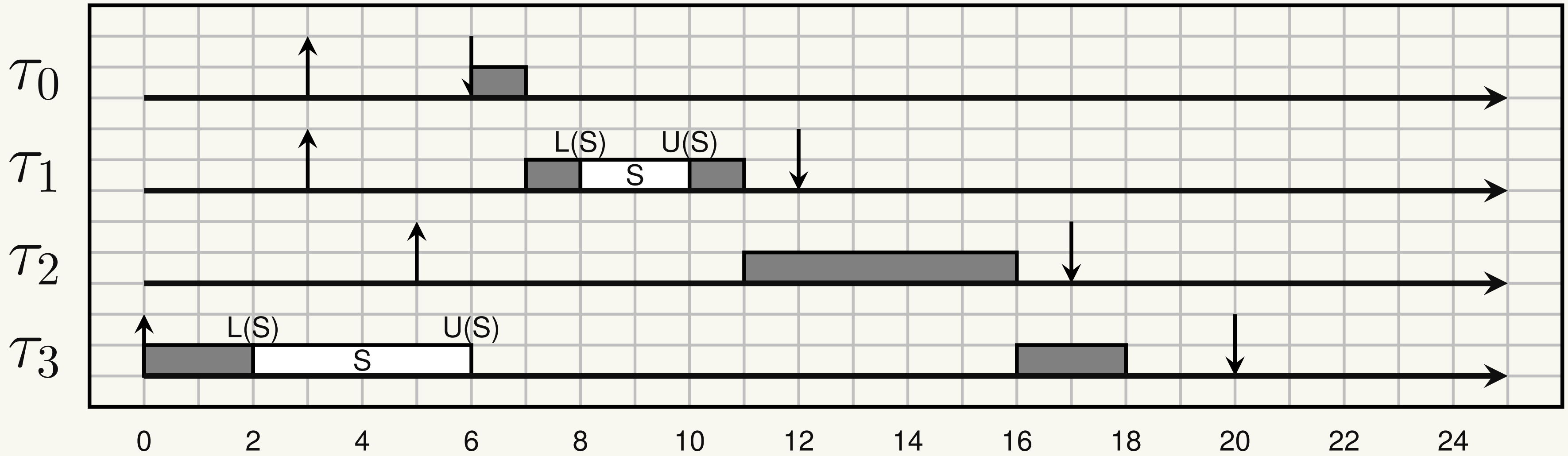


Some Observations

- 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

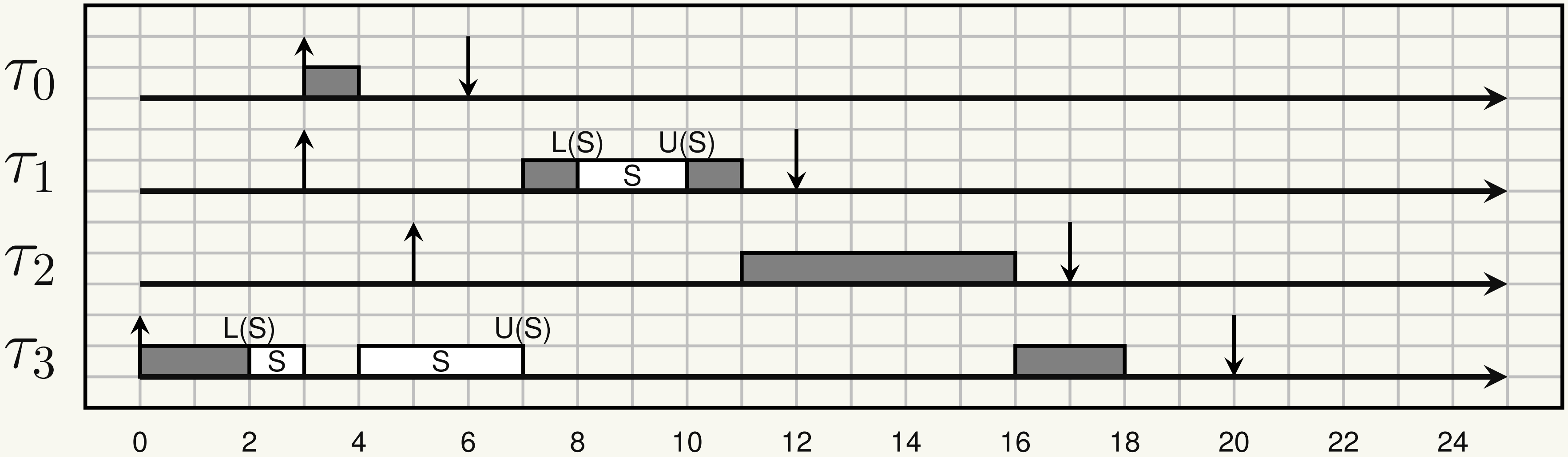
- Consider the following example, with $p_0 > p_1 > p_2 > p_3$.



- 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

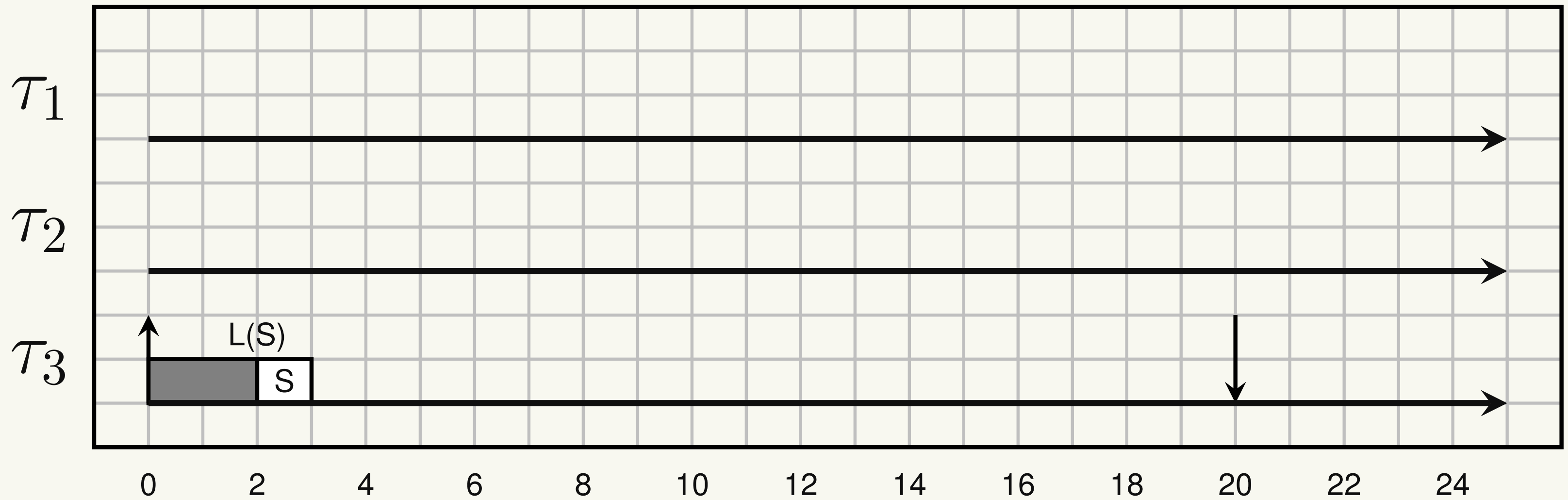
Task	C_i	T_i	$\xi_{i,1}$	D_i	B_i
τ_1	20	70	0	30	2
τ_2	20	80	1	45	2
τ_3	35	200	2	130	0

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

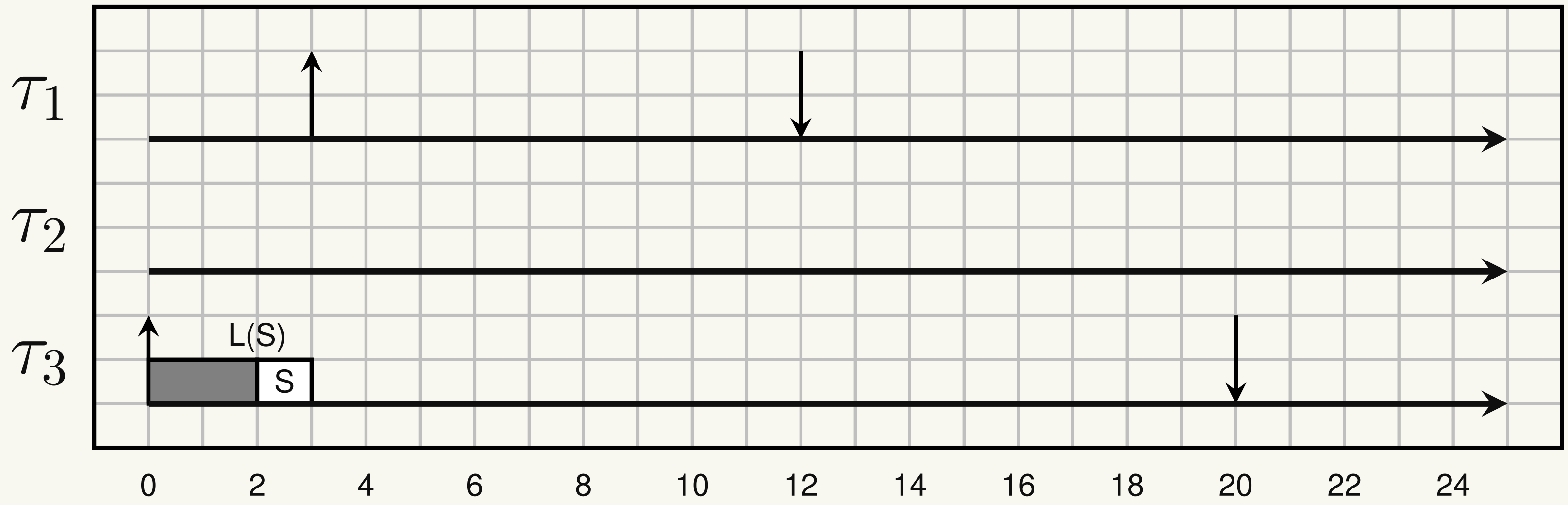
The Priority Inheritance protocol

- 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



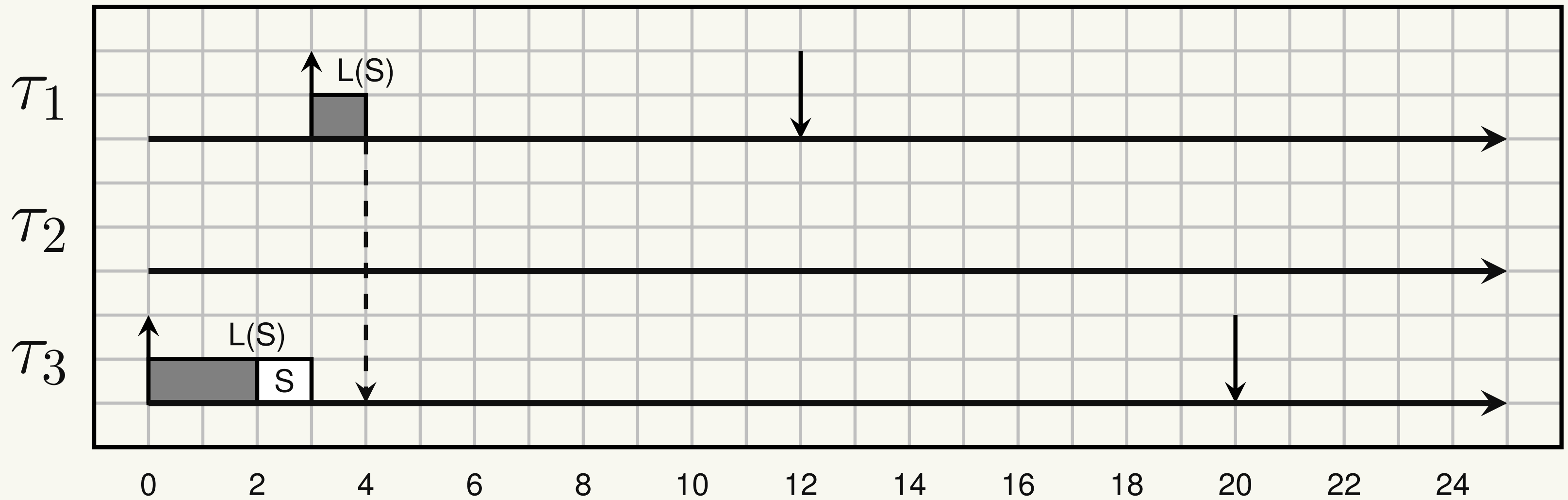
The Priority Inheritance protocol

- 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



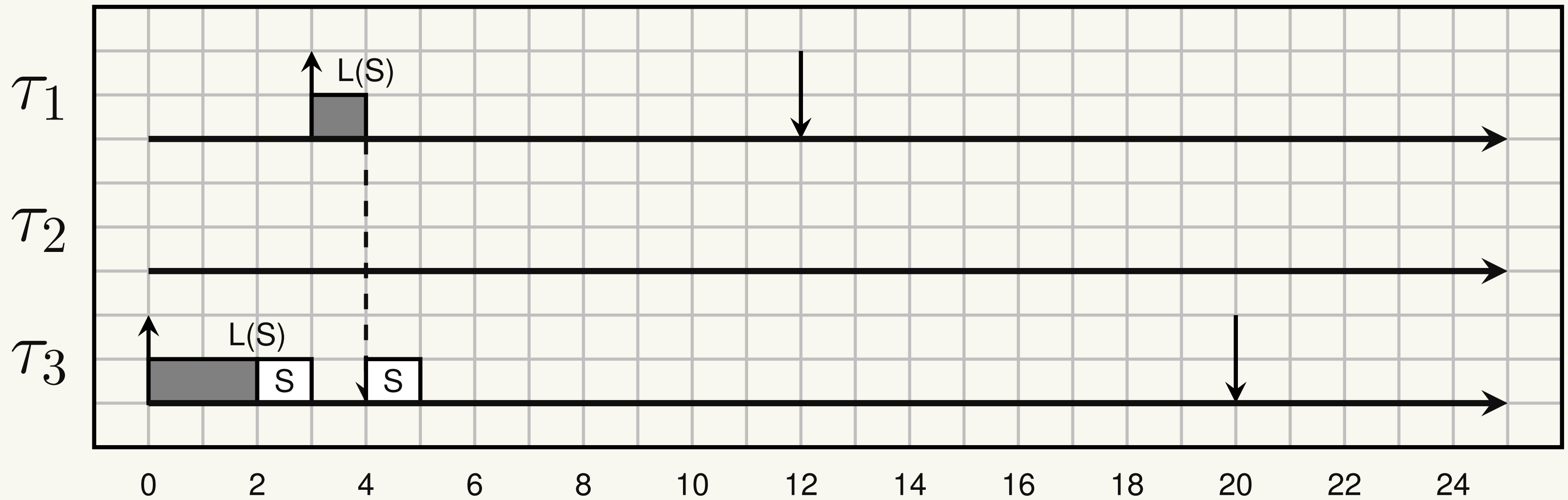
The Priority Inheritance protocol

- 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

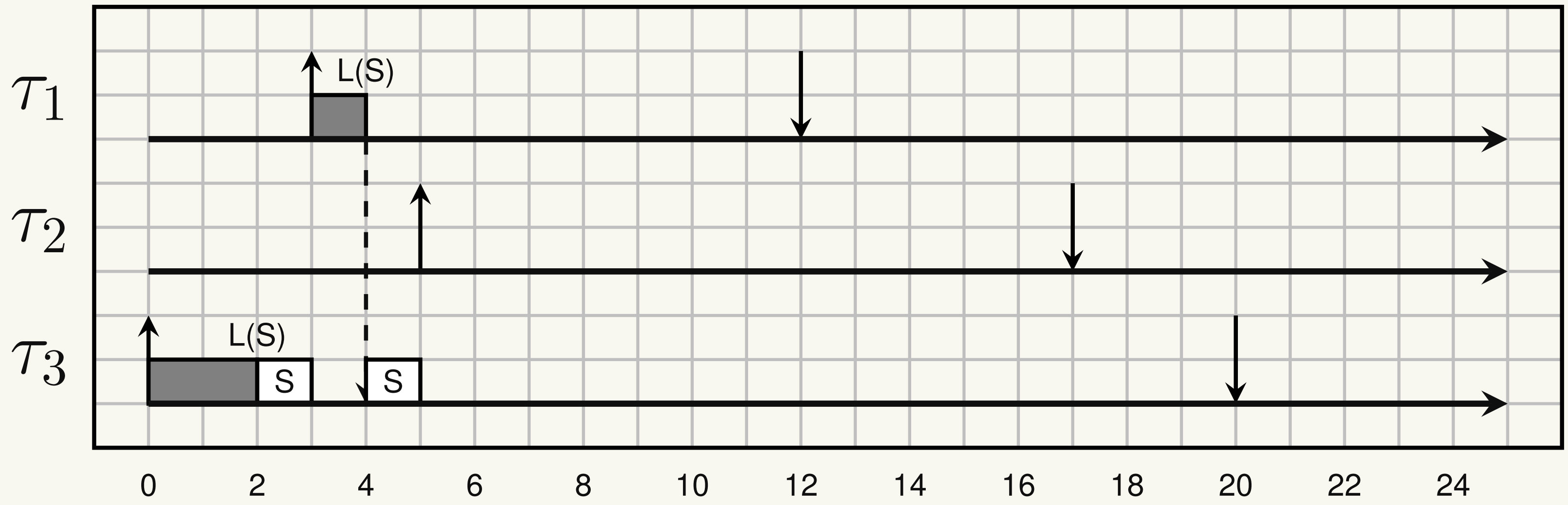
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



- Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

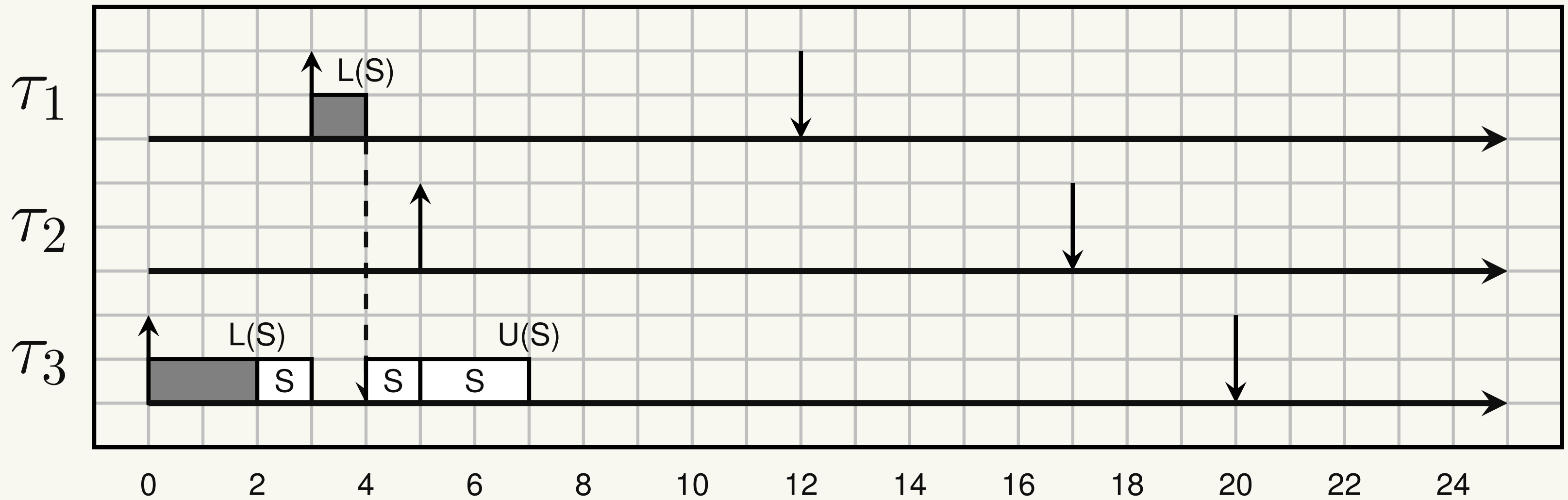
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

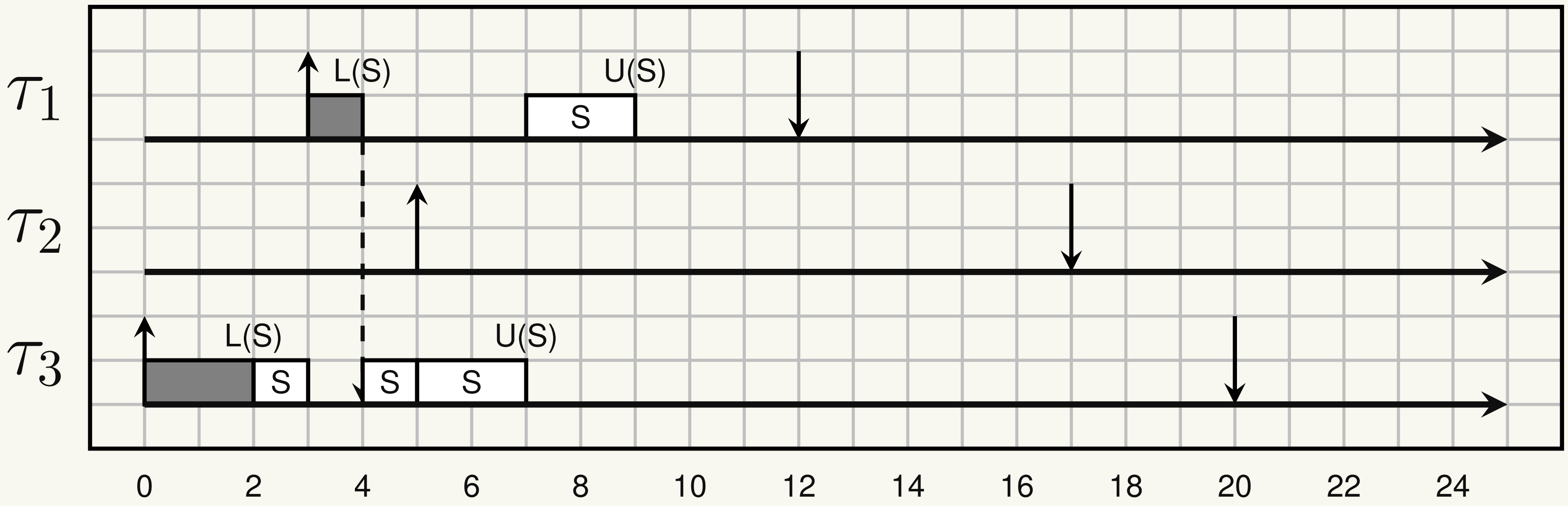
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

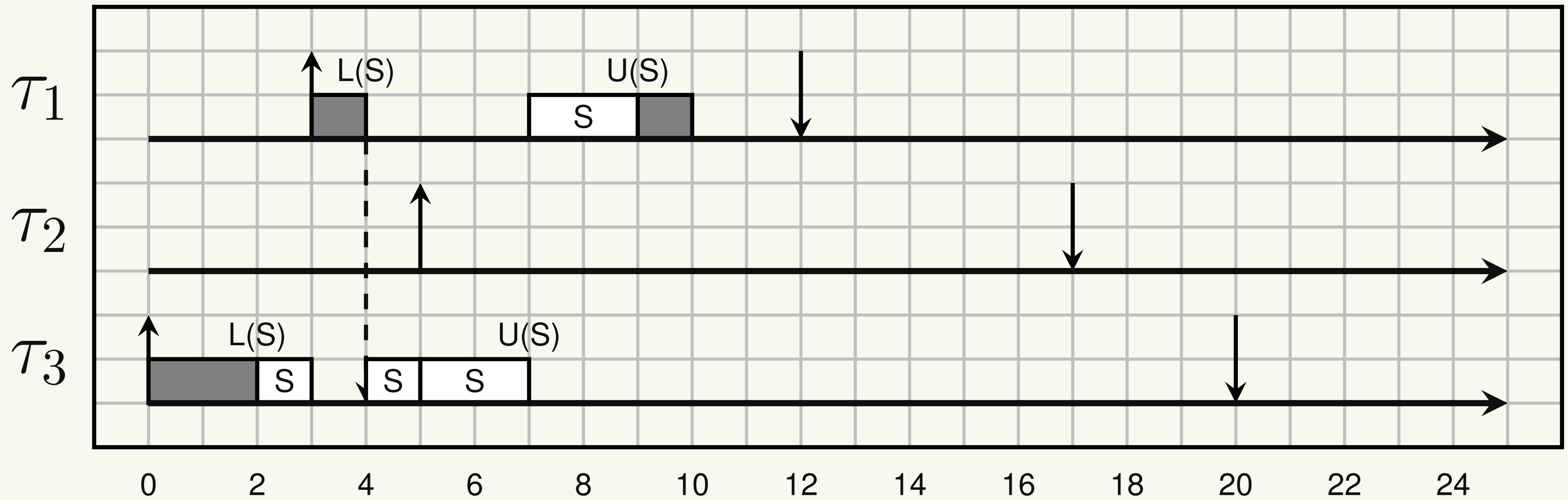
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

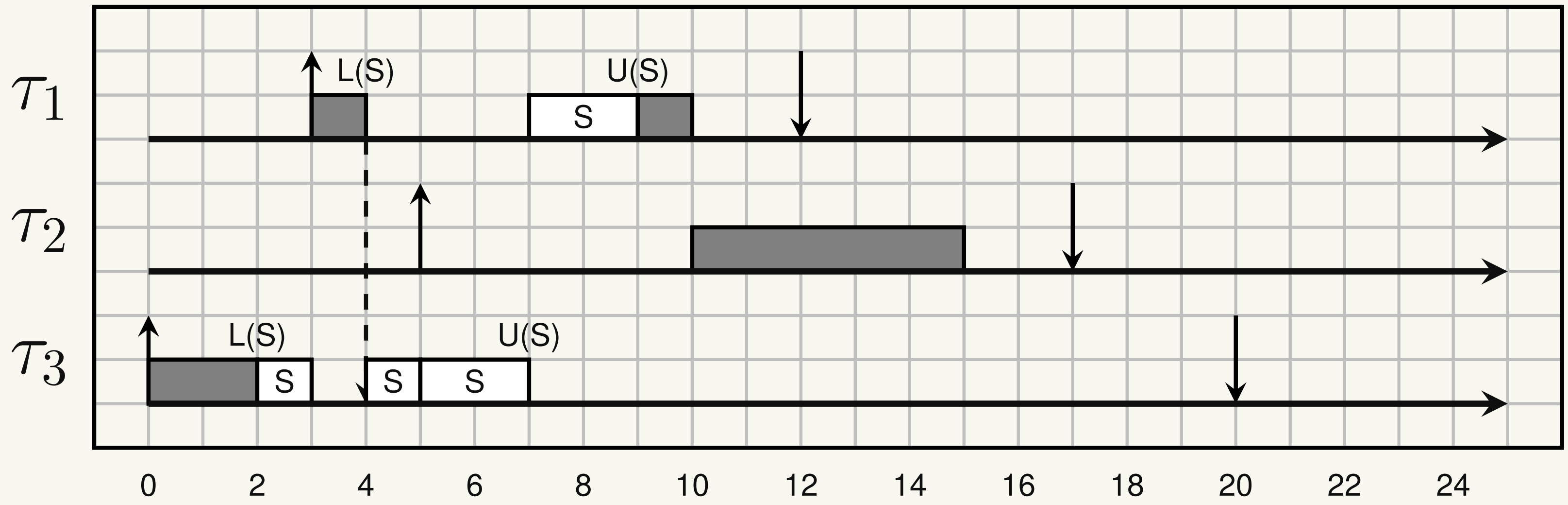
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

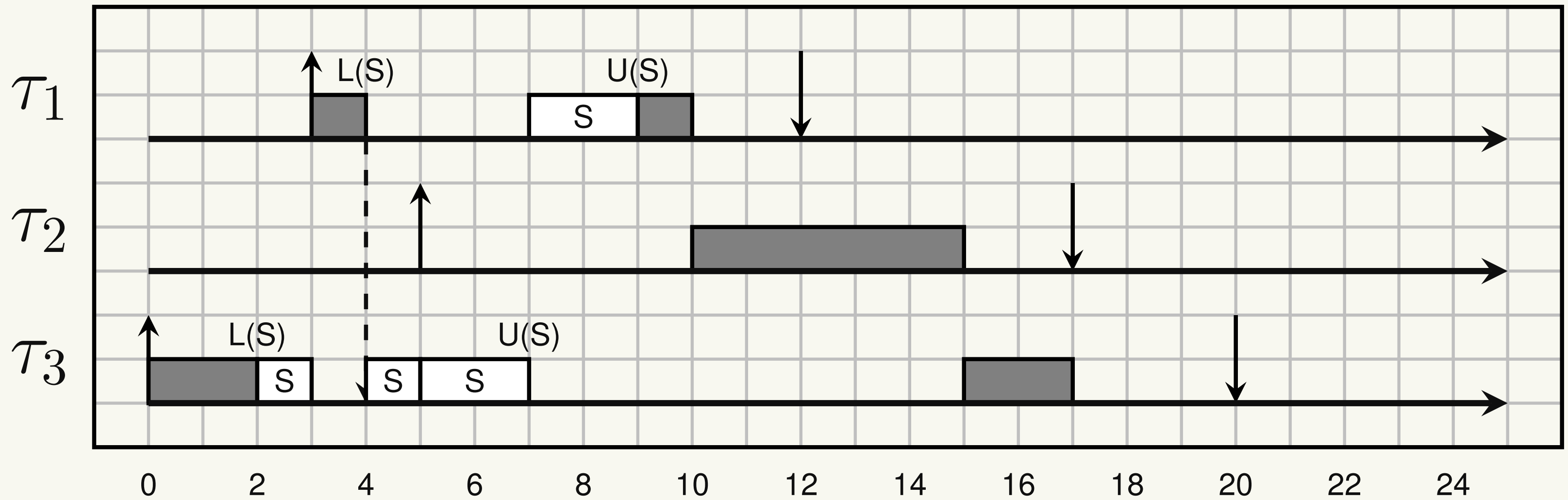
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



  - 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

Maximum Blocking Time for PI - 1

- 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
- The maximum blocking time can be computed based on two important properties
  - They provide an upper bound on the number of times a task can block

Maximum Blocking Time for PI - 2

- Two important theorems:
  - **Theorem 1** if PI is used, a task block only once on each different critical section
  - **Theorem 2** if PI is used, a task can be blocked by another lower priority task for at most the duration of one critical section
- $\Rightarrow$ a task can be blocked more than once, but only once per each resource and once by each task

Blocking Time Computation - 1

- 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
- Blocking times can be computed based on this table

Blocking Time Computation - 2

- How to use the resource usage table?
  - Let's recall the 2 PI properties...
- 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 block only on resources directly used, 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
  - 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

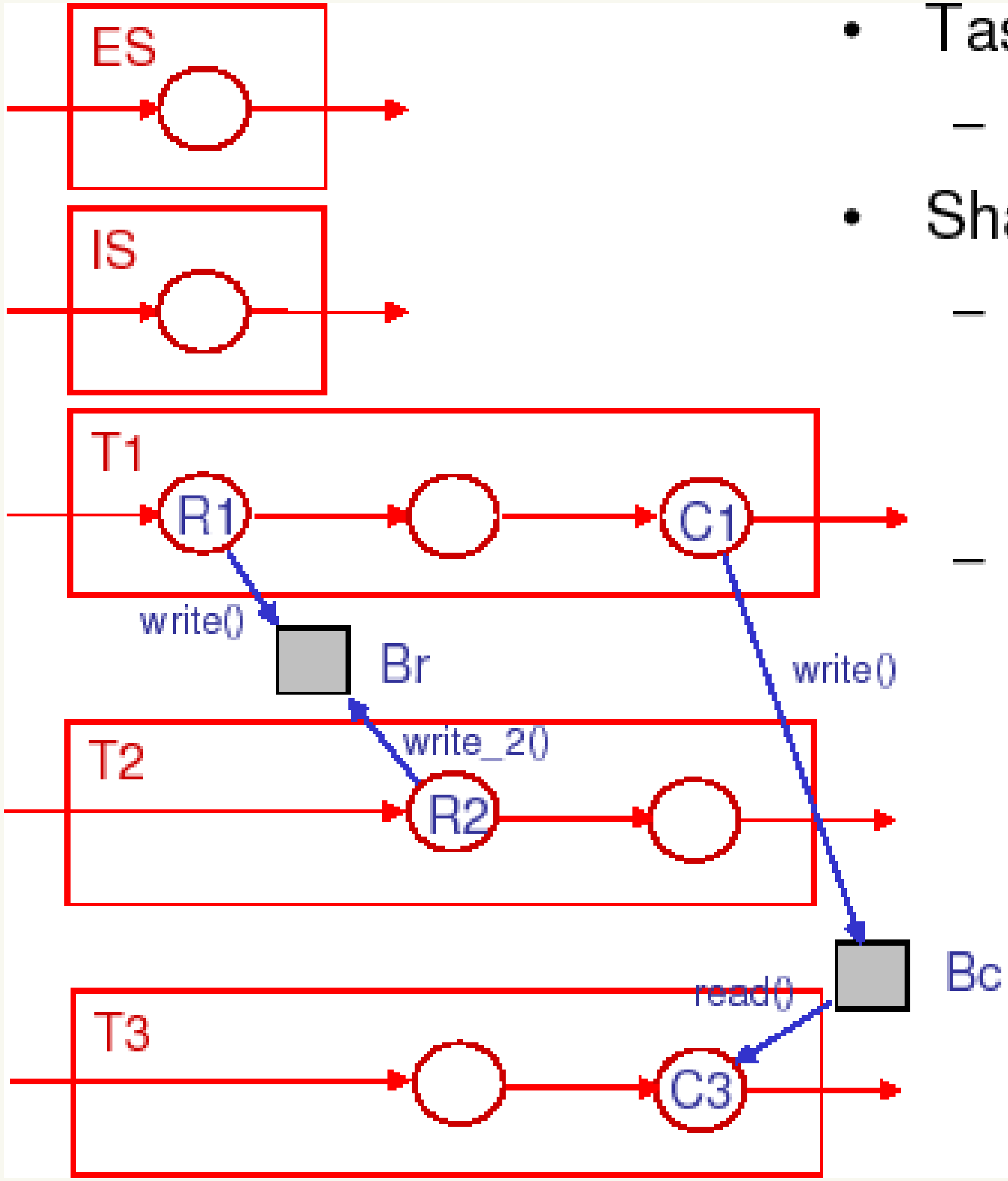
	S_1	S_2	S_3	B
τ_1	2	0	0	3
τ_2	0	1	0	5
τ_3	0	0	2	5
τ_4	3	3	1	?
τ_5	1	2	1	?

- 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, consider column 2: it represents 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 other tasks 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 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**

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

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

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 $\forall i, L_i \leq 1$

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