Sistemi in tempo reale

Shared resources

Giuseppe Lipari

Scuola Superiore Sant'Anna

Pisa -Italy

Interacting tasks

- Until now, we have considered only independent tasks
  - a task never blocks or suspends
  - it can only be suspended when it finishes its istance (job)
- However, in reality, many tasks exchange data through shared memory
- Consider as an example three periodic tasks:
  - One reads the data from the sensors and applies a filter. The results of the filter are stored in memory.
  - The second task 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, a third periodic task 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.

Resources and critical sections

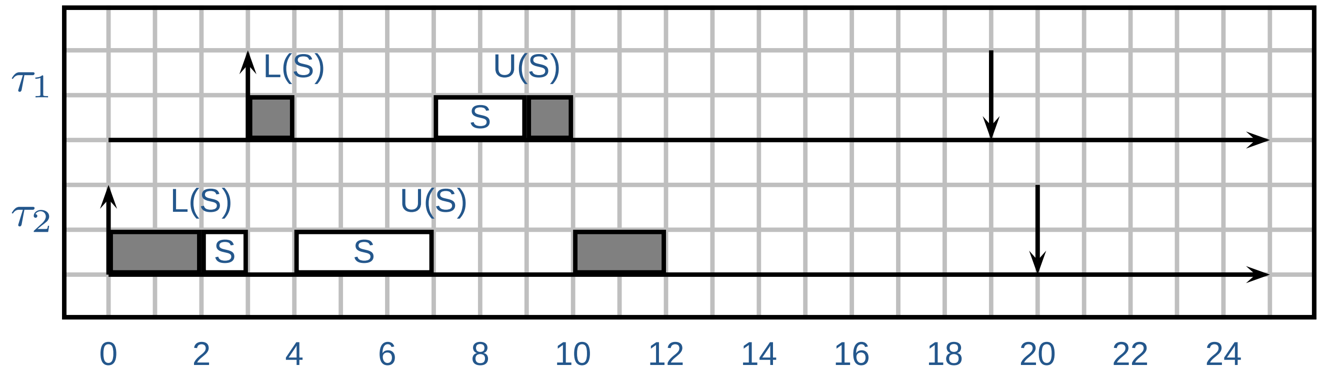
- The shared data structure is called *resource*;
- A piece of code accessing the data structure is called *critical section*;
- Two or more critical sections on the same resource must be executed in *mutual exclusion*;
- Therefore, each data structure should be *protected* by a mutual exclusion mechanism;
- In this lecture, we will study what happens when resources are protected by mutual exclusion semaphores.

Notation

- The resource and the corresponding mutex semaphore will be denoted by symbol S_j.
- A system consists of:
  - A set of N periodic (or sporadic) tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$;
  - A set of shared resources $\mathcal{S} = \{S_1, \ldots, S_M\}$;
  - We say that a task τ_i *uses* resource S_j if it accesses the resource with a critical section.
  - The k-th critical of τ_i on S_j is denoted with $cs_{i,j}(k)$.
  - The length of the longest critical section of τ_i on S_j is denoted by $\xi_{i,j}$.

Blocking and priority inversion

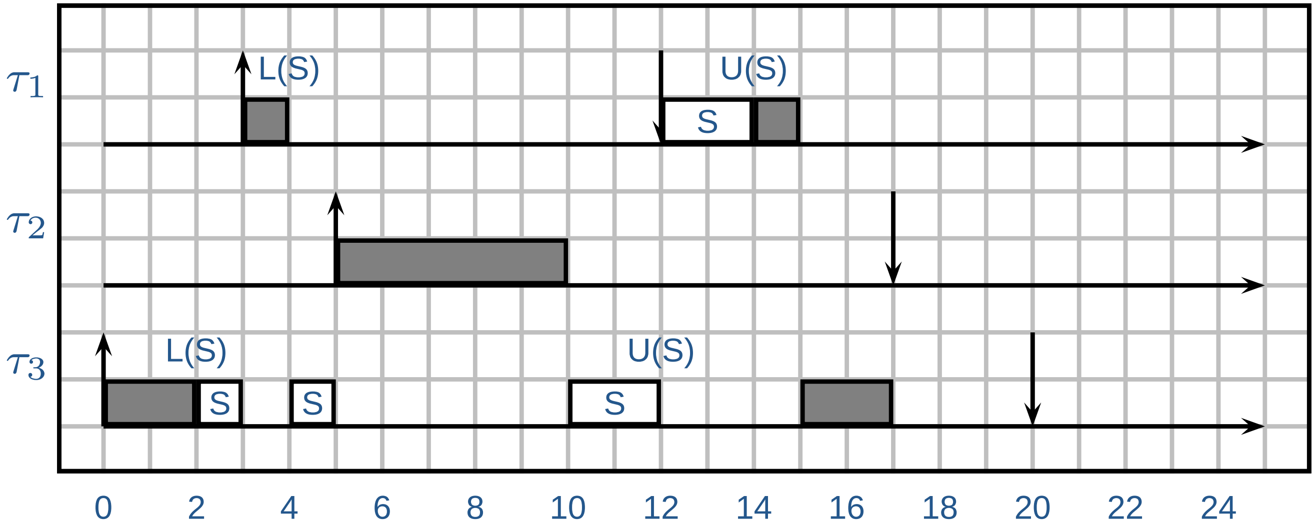
- A blocking condition happens when a high priority tasks wants to access a resource that is held by a lower priority task.
- Consider the following example, where $p_1 > p_2$.



- From time 4 to 7, task τ_1 is blocked by a lower priority taskτ_2; this is a *priority inversion*.
- 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 $p_1 > p_2 > p_3$.



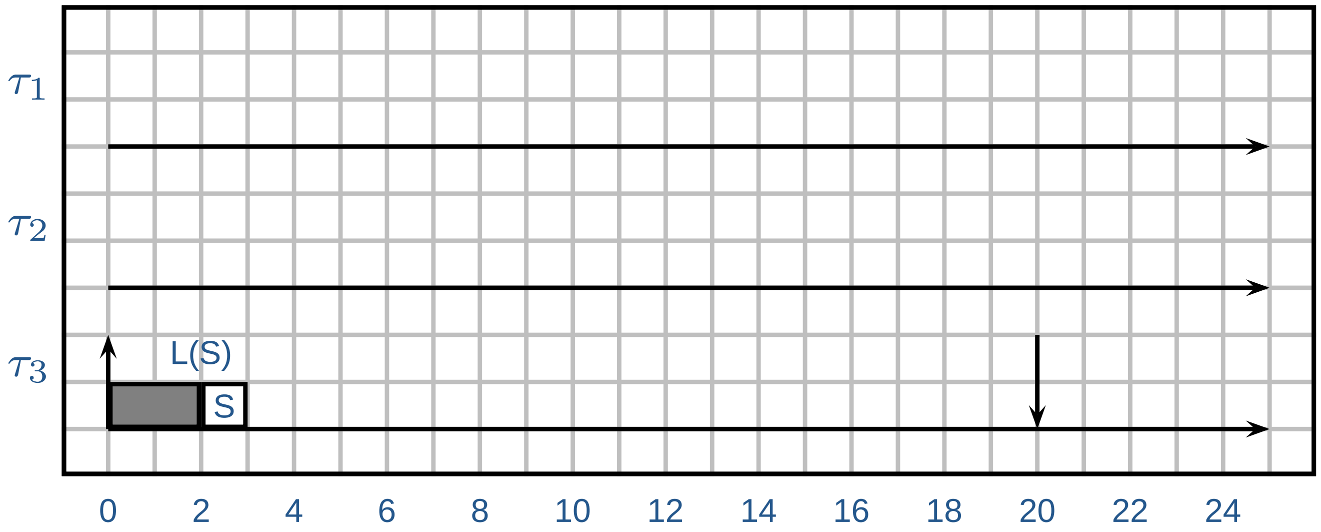
- This time the priority inversion is very large: from 4 to 12.
- The problem is that, while τ_1 is blocked, τ_2 arrives and preempt τ_3 before it can leave the critical section.
- If there are other medium priority tasks, they could preempt τ_3 as well.
- Potentially, the priority inversion could be unbounded!

The Mars Pathfinder

- This is not only a theoretical problem. It may happen in real cases.
- The most famous example of such problem was found during the Mars Pathfinder mission.
  - 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.

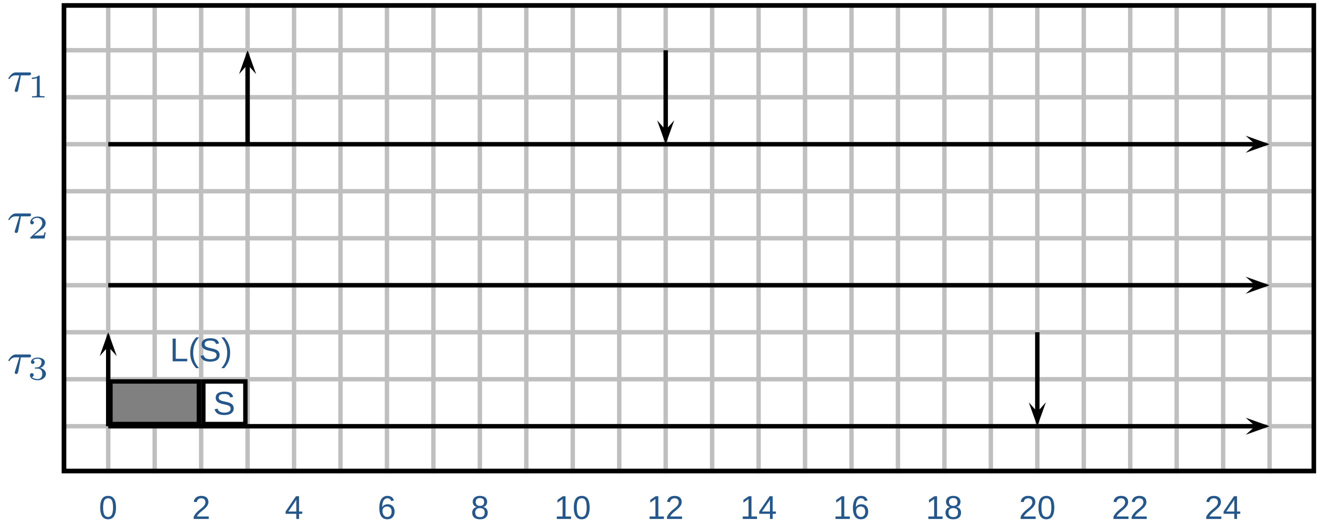
The Priority Inheritance protocol

- The solution of the problem is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



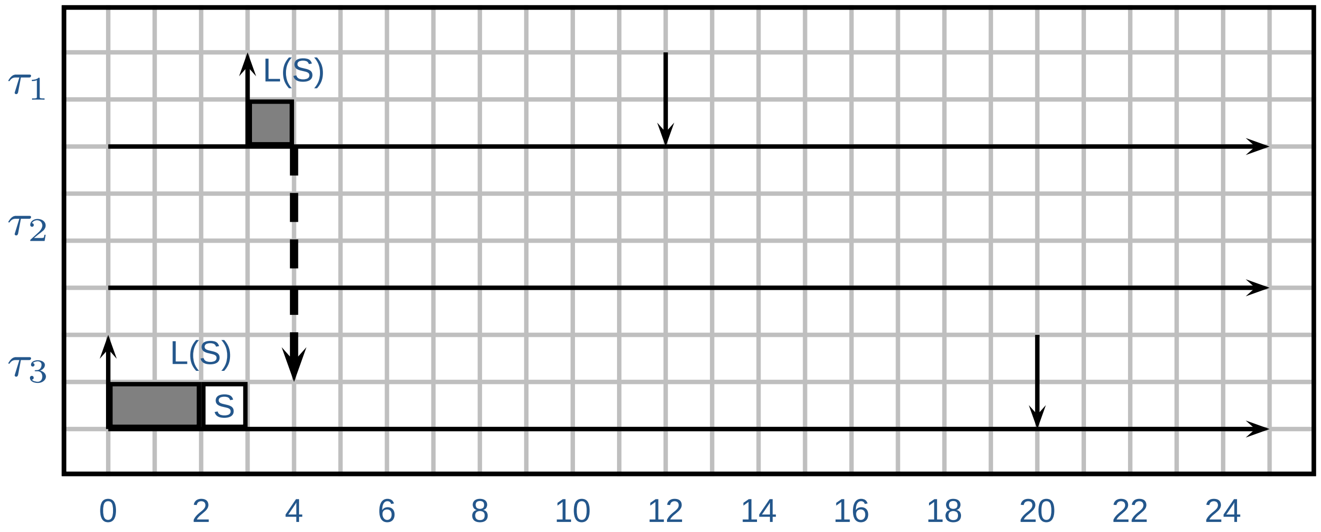
The Priority Inheritance protocol

- The solution of the problem is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



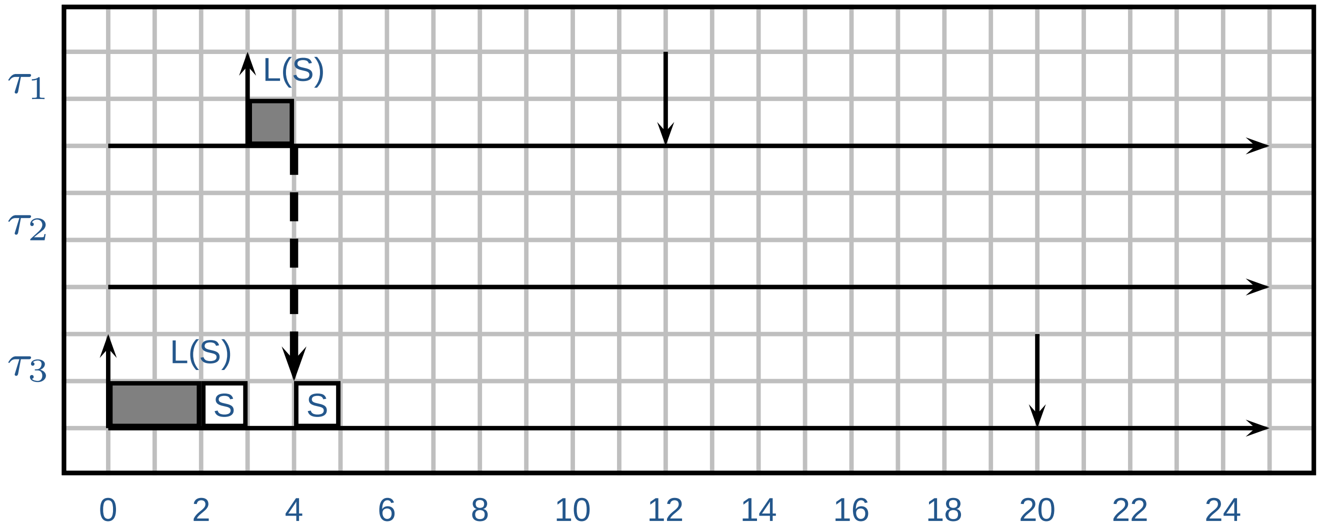
The Priority Inheritance protocol

- The solution of the problem is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



The Priority Inheritance protocol

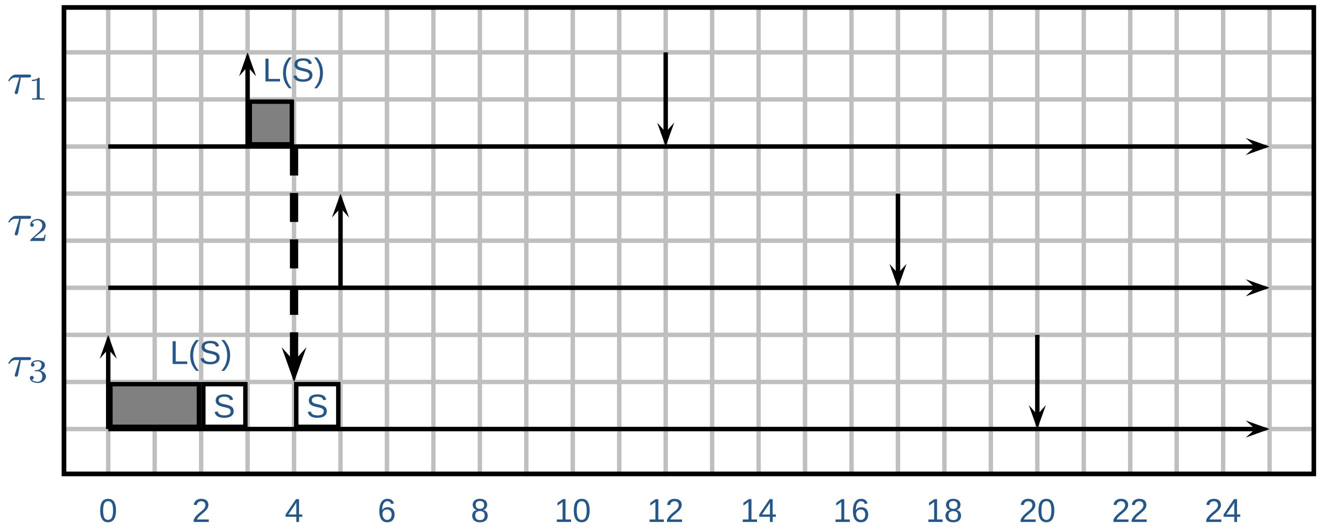
- The solution of the problem is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



  - Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

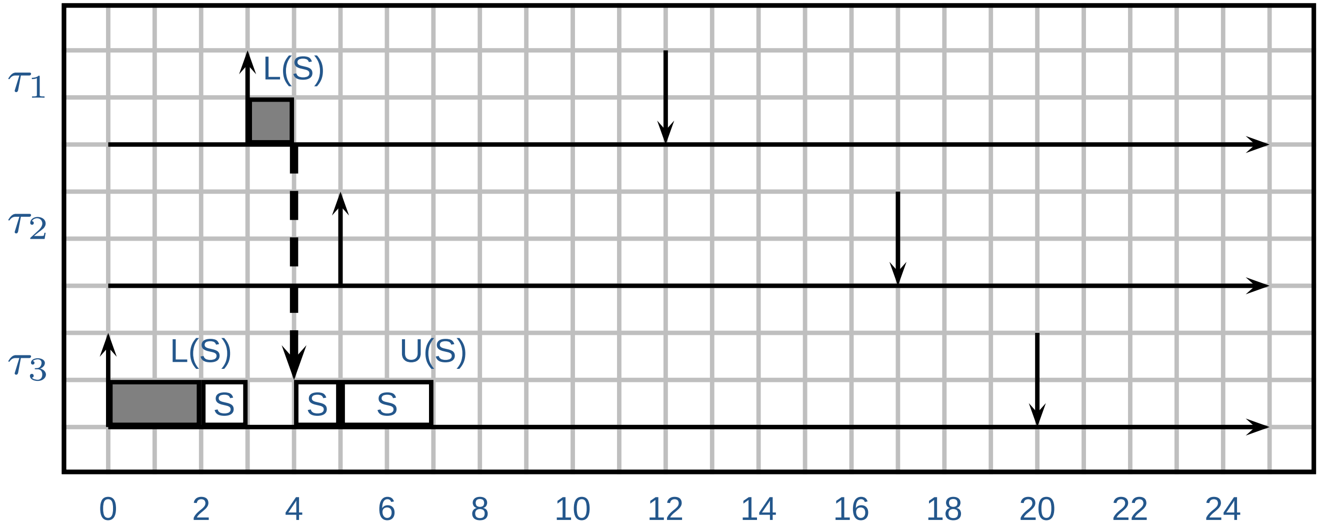
- The solution of the problem is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



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

The Priority Inheritance protocol

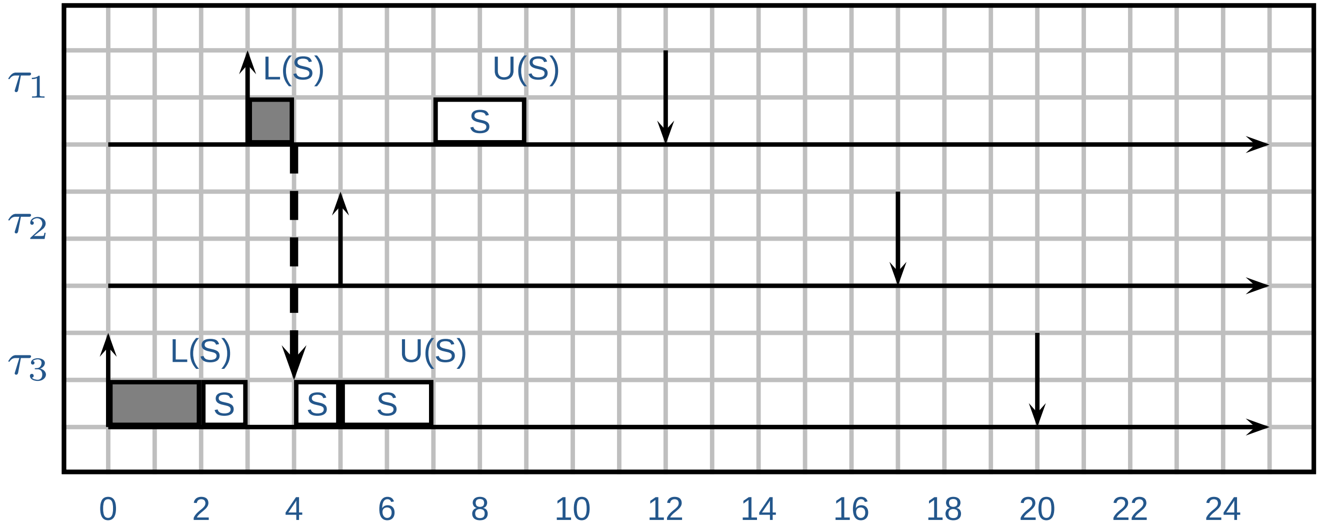
- The solution of the problem is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



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

The Priority Inheritance protocol

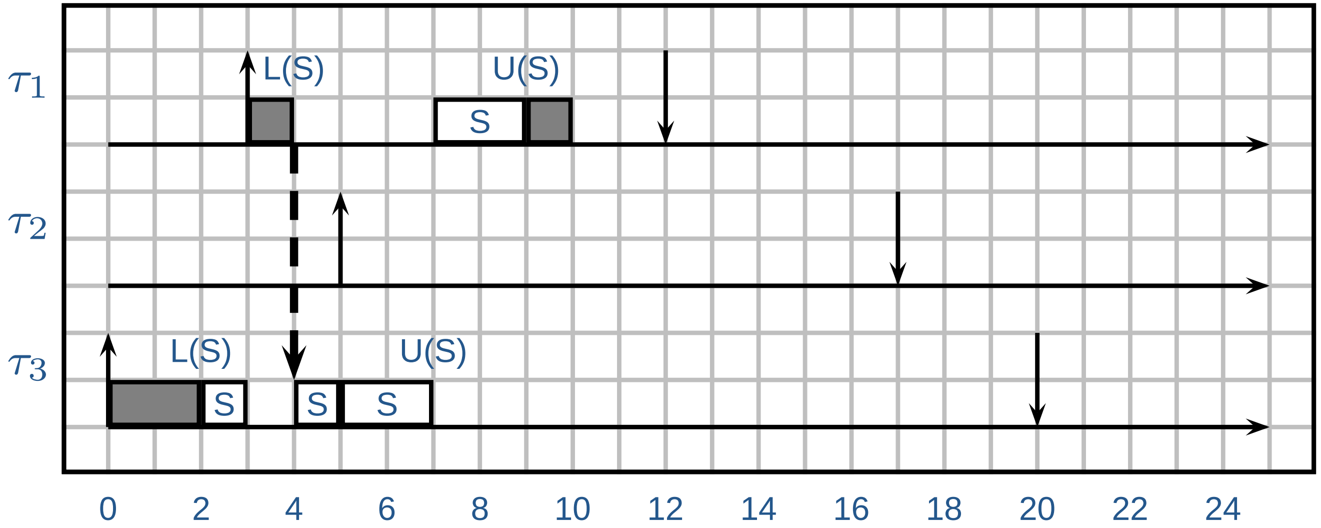
- The solution of the problem is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



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

The Priority Inheritance protocol

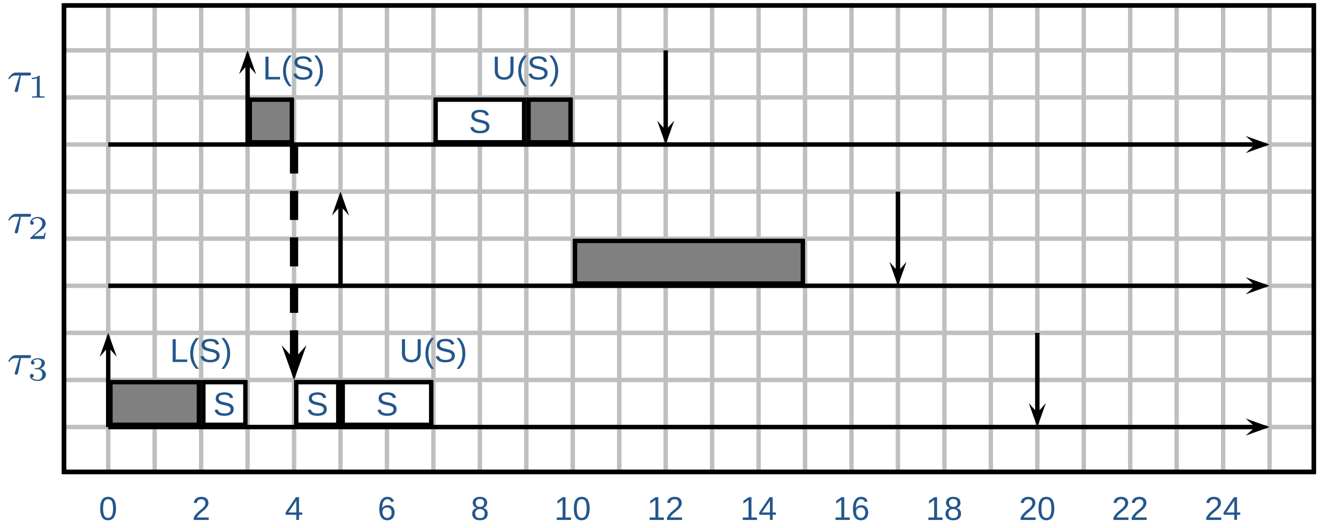
- The solution of the problem is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



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

The Priority Inheritance protocol

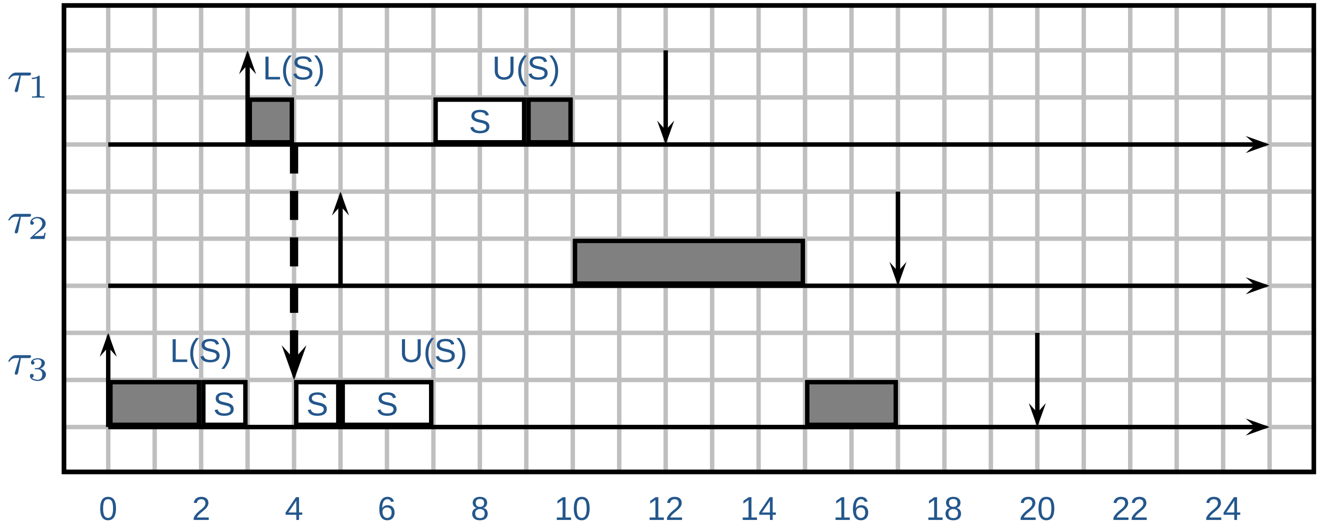
- The solution of the problem is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



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

The Priority Inheritance protocol

- The solution of the problem is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



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

Comments

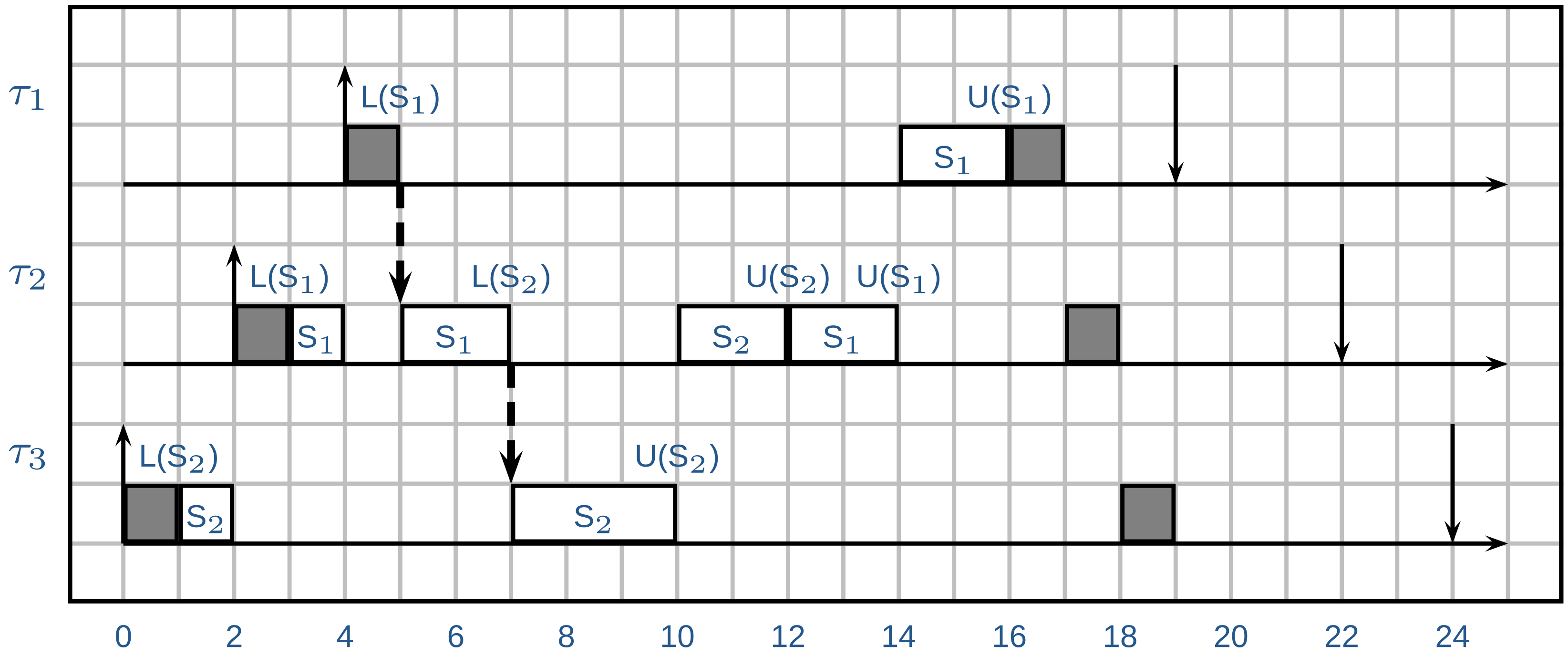
- The blocking (priority inversion) is now bounded to the length of the critical section of task τ_3
- Tasks with intermediate priority τ_2 cannot interfere with τ_1
- However, τ_2 has a blocking time, even if it does not use any resource
  - This is called *indirect blocking* and it is due to the fact 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.
- It remains to understand:
  - What is the maximum blocking time for a task
  - How we can account for blocking times in the schedulability analysis
- From now on, the maximum blocking time for a task τ_i is denoted by B_i.

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

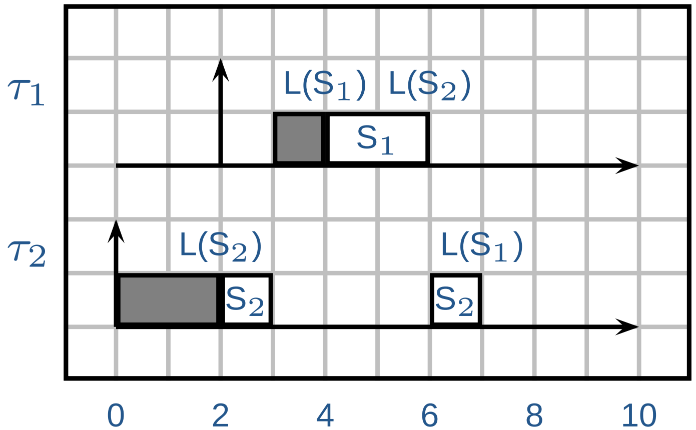
- Task τ_1 uses resource S_1; Task τ_2 uses S_1 and S_2 nested inside S_1; Task τ_3 uses only S_2.
- $p_1 > p_2 > p_3$;



- At time $t = 7$ task 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

- When using nested critical section, the problem of deadlock can occur; i.e. two or more tasks can be blocked waiting for each other.
- The priority inheritance protocol *does not solve* automatically the problem of deadlock, as it is possible to see in the following example.
    - Task τ_1 uses S_2 inside S_1, while task 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.
- To avoid deadlock, 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).

The Priority Inheritance Protocol

- Summarising, the main rules are the following;
  - If a task τ_i is blocked 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 is itself blocked 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.

Computing the maximum blocking time

- We will compute the maximum blocking time only in the case of non nested critical sections.
  - Even if we avoid the problem of deadlock, when critical sections are nested, the computation of the blocking time becomes very complex due to multiple inheritance.
  - If critical section are not nested, multiple inheritance cannot happen, and the computation of the blocking time becomes simpler.
- To compute the blocking time, we must consider the following two important theorems:
  - **Theorem 1** Under the priority inheritance protocol, a task can be blocked only once on each different semaphore.
  - **Theorem 2** Under the priority inheritance protocol, a task can be blocked by another lower priority task for at most the duration of one critical section.
- This means that we have to consider that a task can be blocked more than once, but only once per each resource and once by each task.

Blocking time computation

- We must build a *resource usage table*.
  - On each row we, put a task in decreasing order of priority;
  - On each column we put a resource (the order is not important);
  - On each cell (i, j) we put $\xi_{i,j}$, 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, we must consider only those column on which it can be blocked (used by itself or by higher priority tasks).

Example of blocking time computation

	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 of blocking time computation

	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	?

- Now τ_2: it can be blocked on S_1 (*indirect blocking*) and on S_2. Therefore, we must consider the first 2 columns;
- Then, we must consider all cases where two distinct lower priority tasks between τ_3, τ_4 and τ_5 access S_1 and S_2, sum the two contributions, and take the maximum;
- 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: $\rightarrow 4$;
- The maximum is $B_2 = 5$.

Example of blocking time computation

	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	?

- Now τ_3;
- It can be blocked on all 3 resources. We must consider all columns;
- The possibilities are:
  - τ_4 on S_1 and τ_5 on S_2: $\longrightarrow 5$;
  - τ_4 on S_2 and τ_5 on S_1 or S_3: $\longrightarrow 4$;
  - τ_4 on S_3 and τ_5 on S_1: $\longrightarrow 2$;
  - τ_4 on S_3 and τ_5 on S_2 or S_3: $\longrightarrow 3$;
- The maximum is $B_3 = 5$.

Example of blocking time computation

	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	?

- Now τ_4;
- It 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