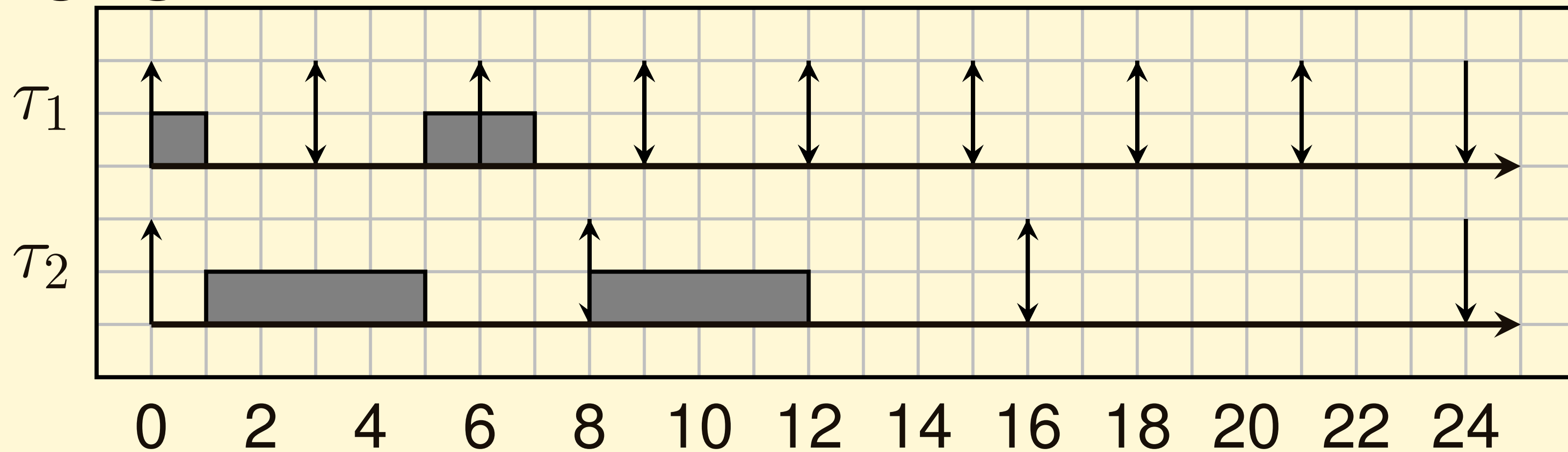
Real-Time Operating Systems

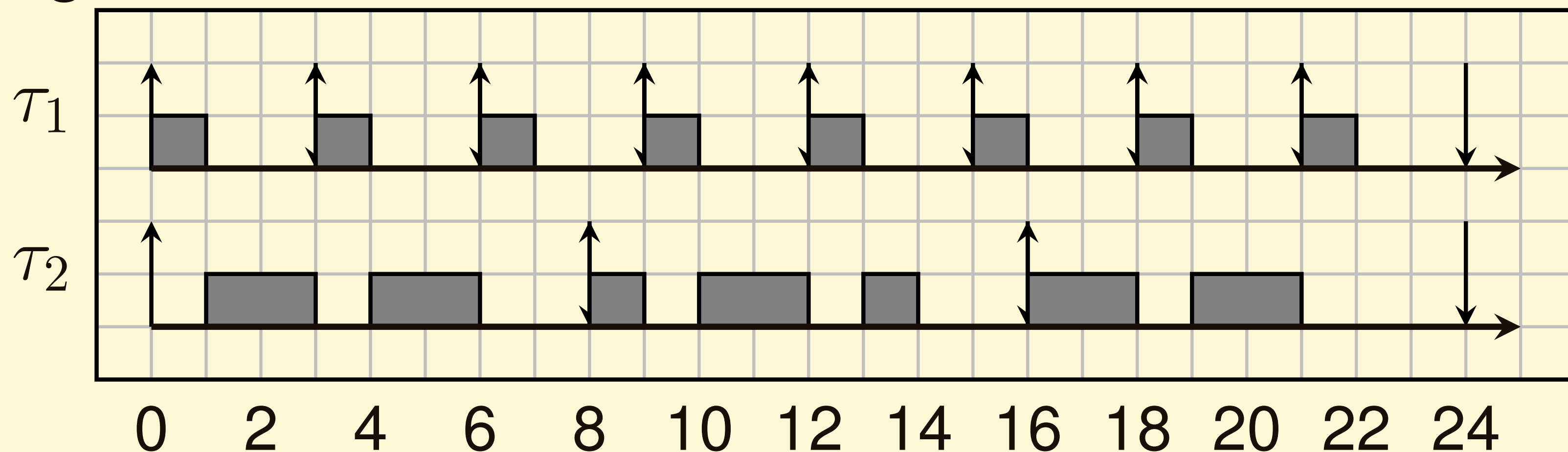
Luca Abeni
luca.abeni@santannapisa.it

RT Scheduling: Why?

- The task set $\mathcal{T} = \{(1,3),(4,8)\}$ is not schedulable by FCFS



- $\mathcal{T} = \{(1,3),(4,8)\}$ is schedulable with other algorithms

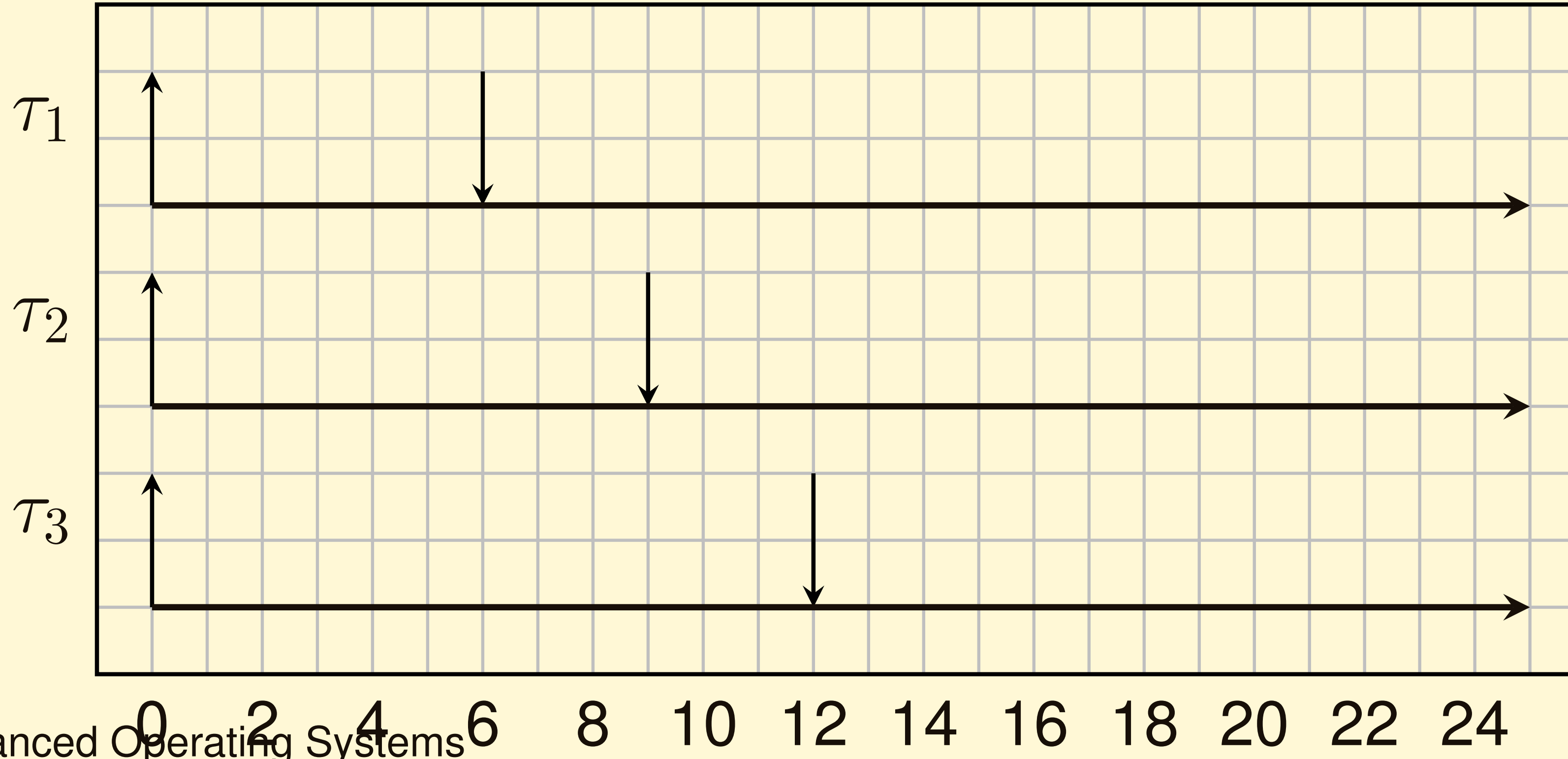


Fixed Priority Scheduling

- Very simple *preemptive* scheduling algorithm
  - Every task τ_i is assigned a fixed priority p_i
  - The active task with the highest priority is scheduled
- Priorities are integer numbers: the higher the number, the higher the priority
  - In the research literature, sometimes authors use the opposite convention: the lowest the number, the highest the priority
- In the following we show some examples, considering periodic tasks, constant execution times, and deadlines equal to the period

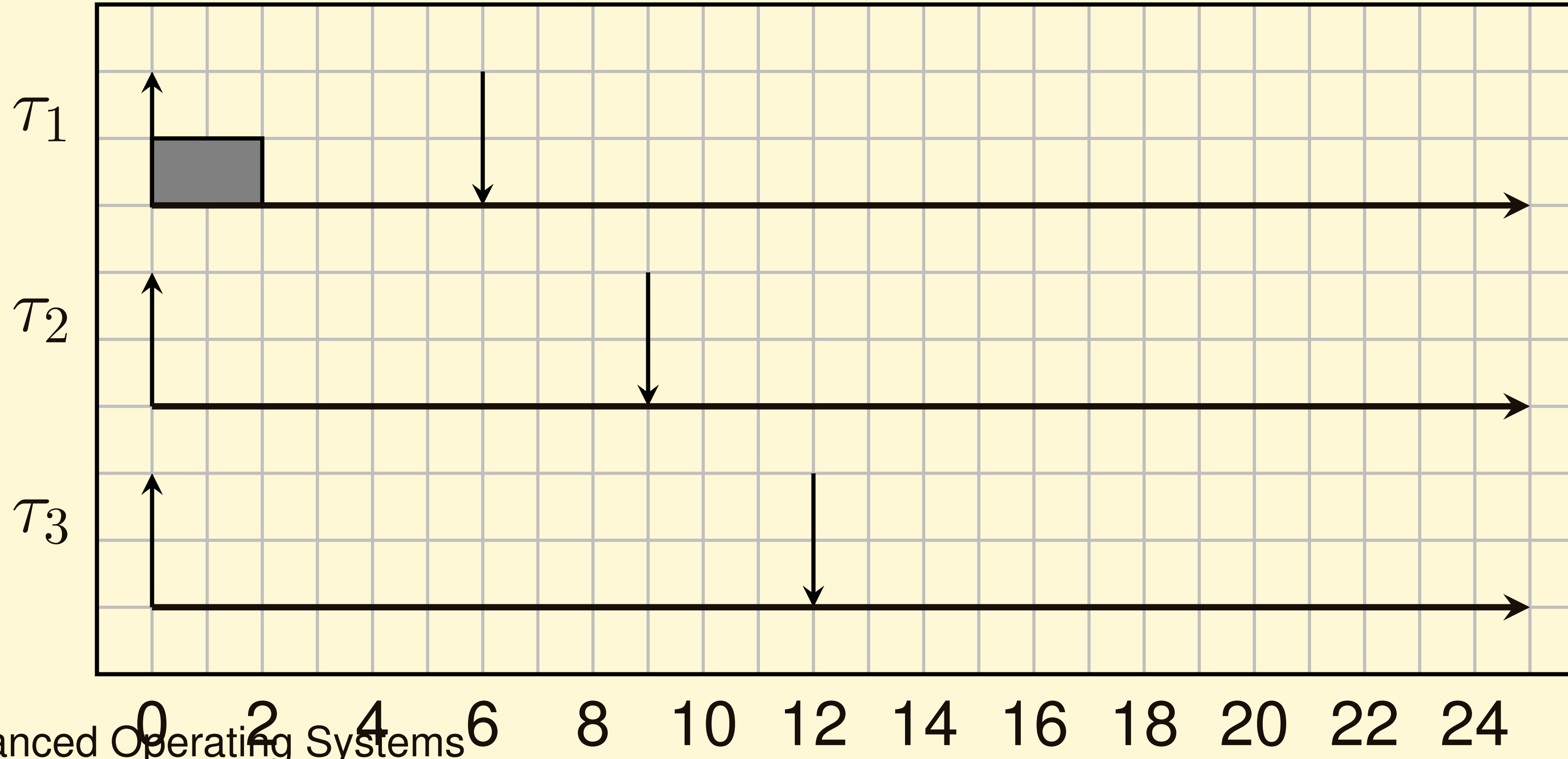
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



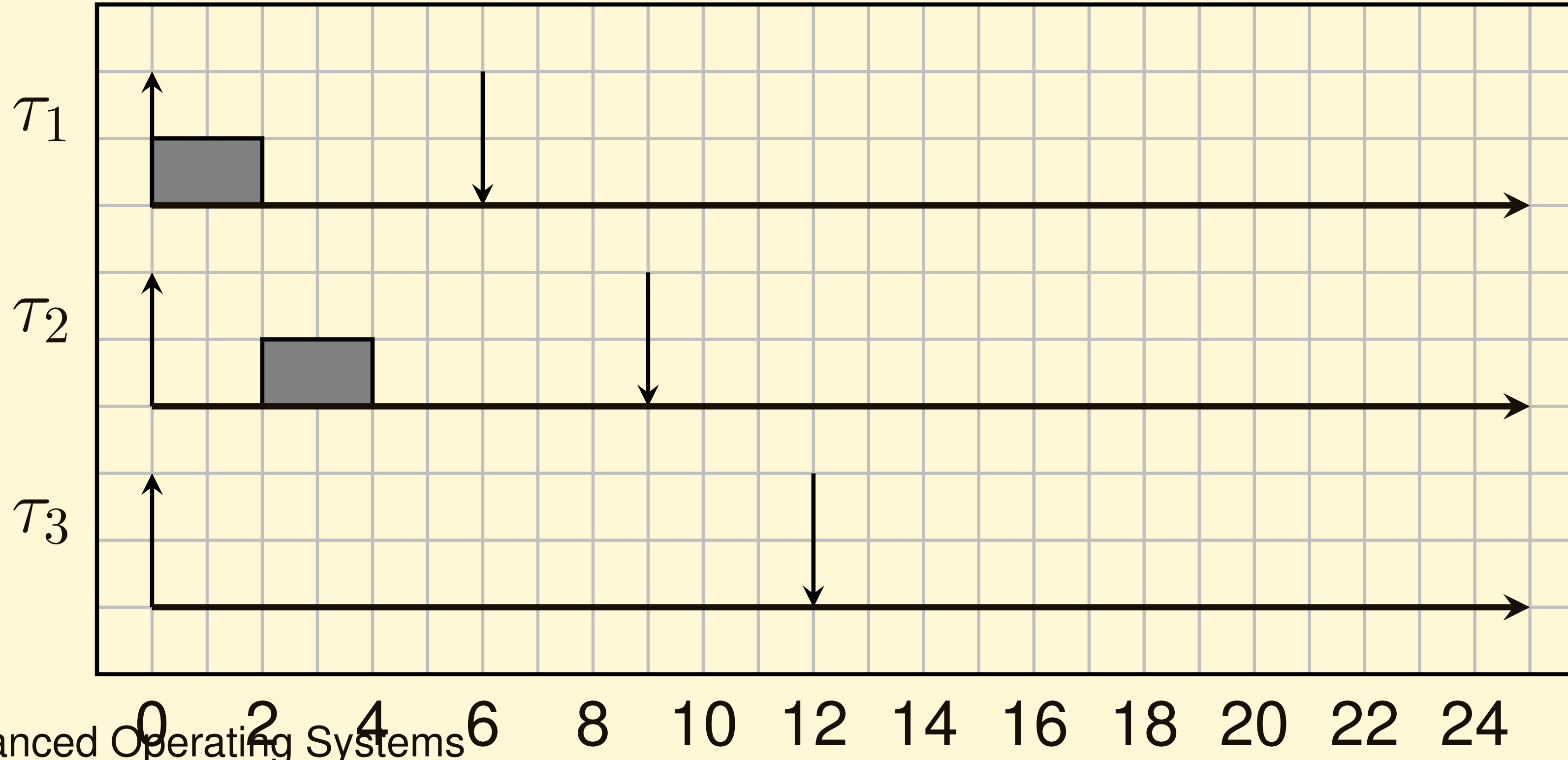
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



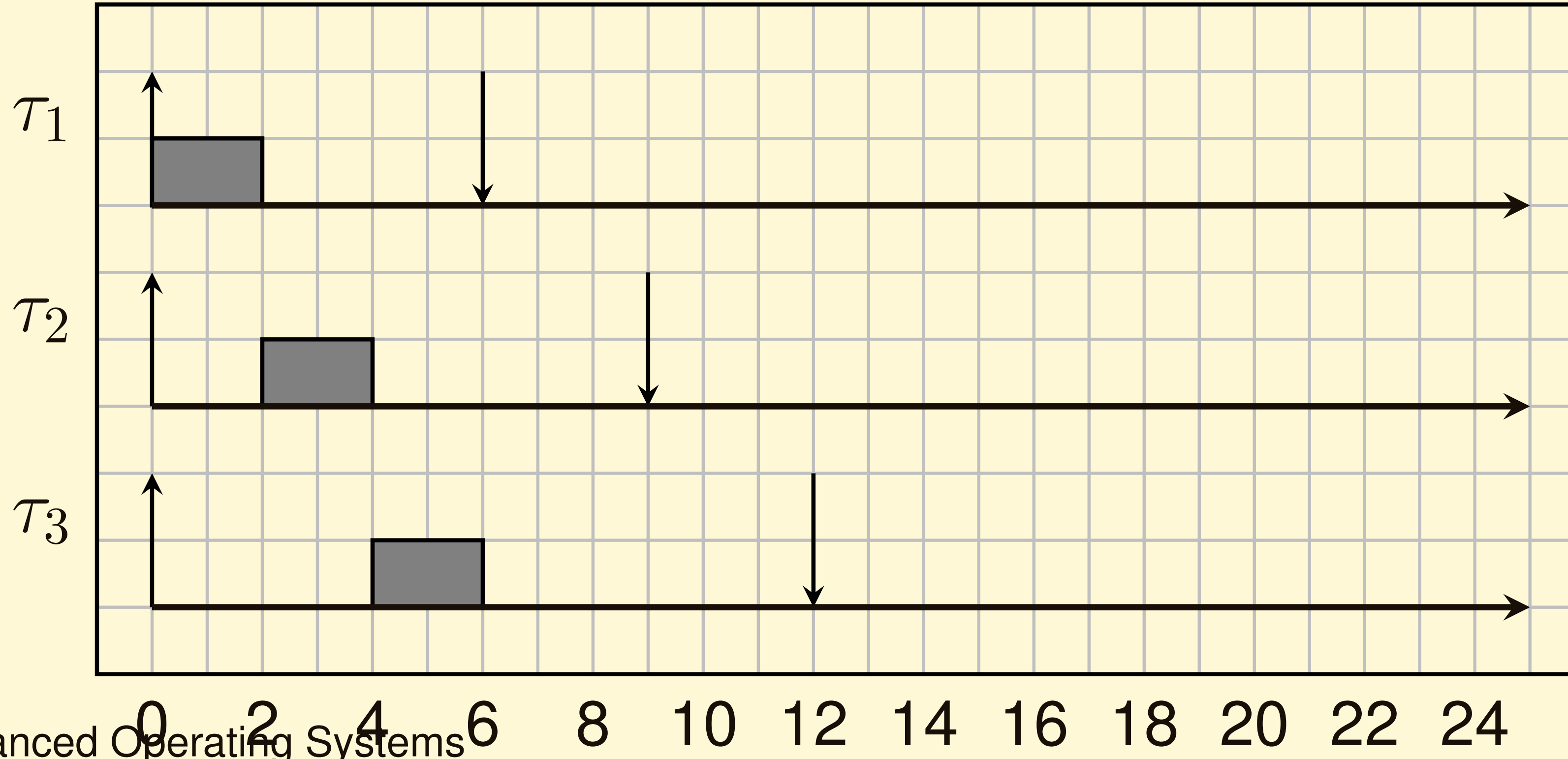
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



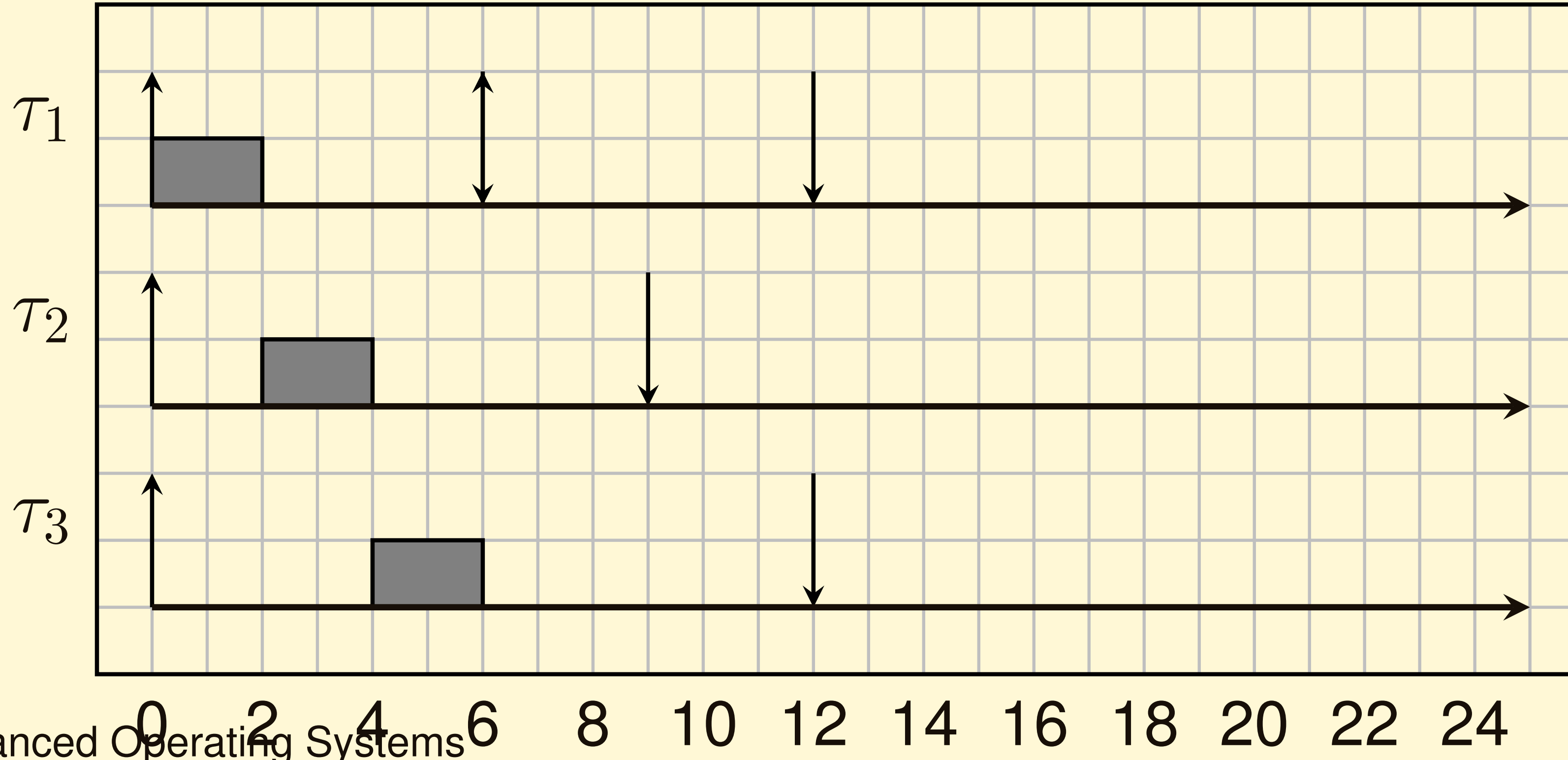
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



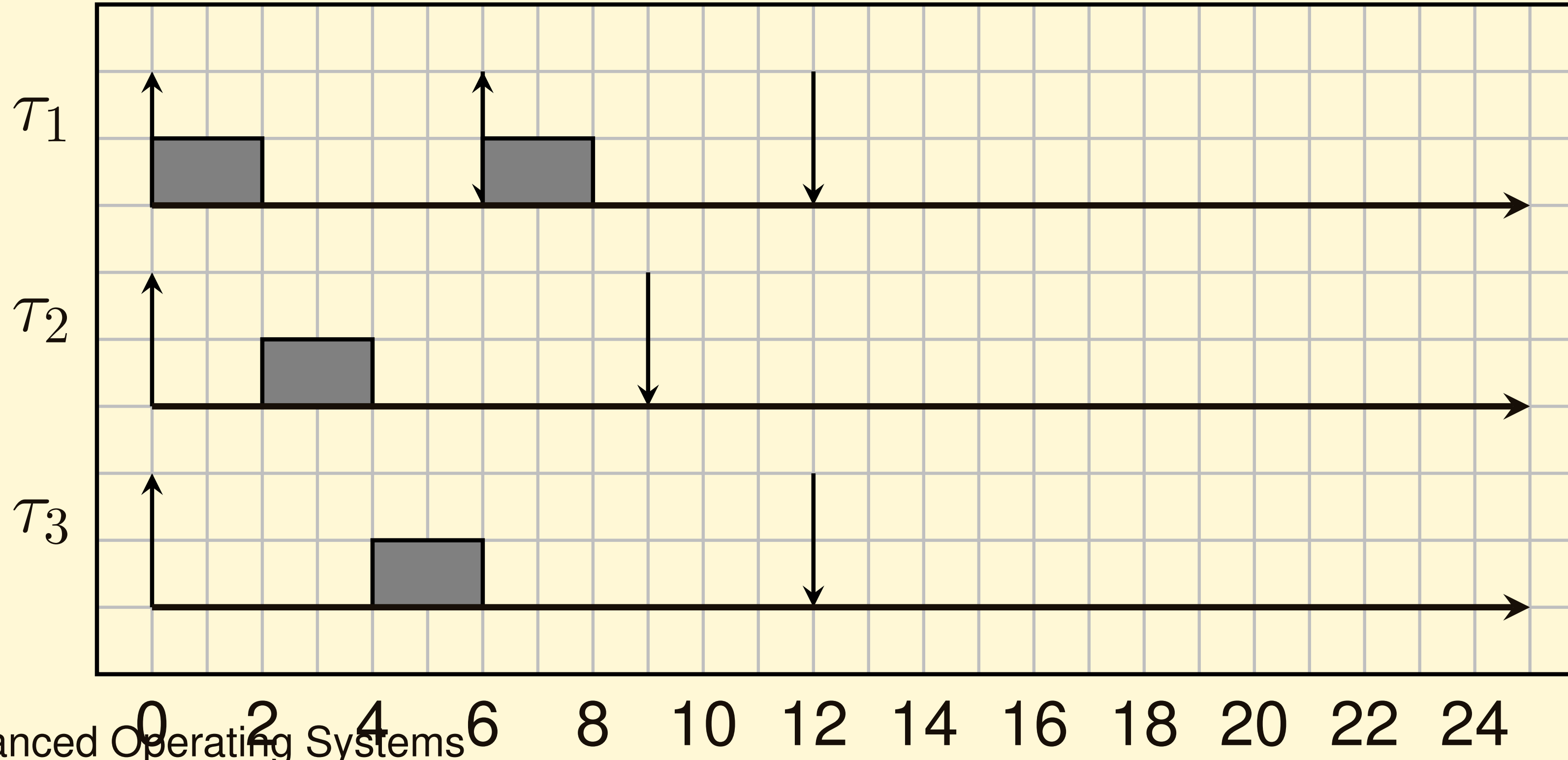
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



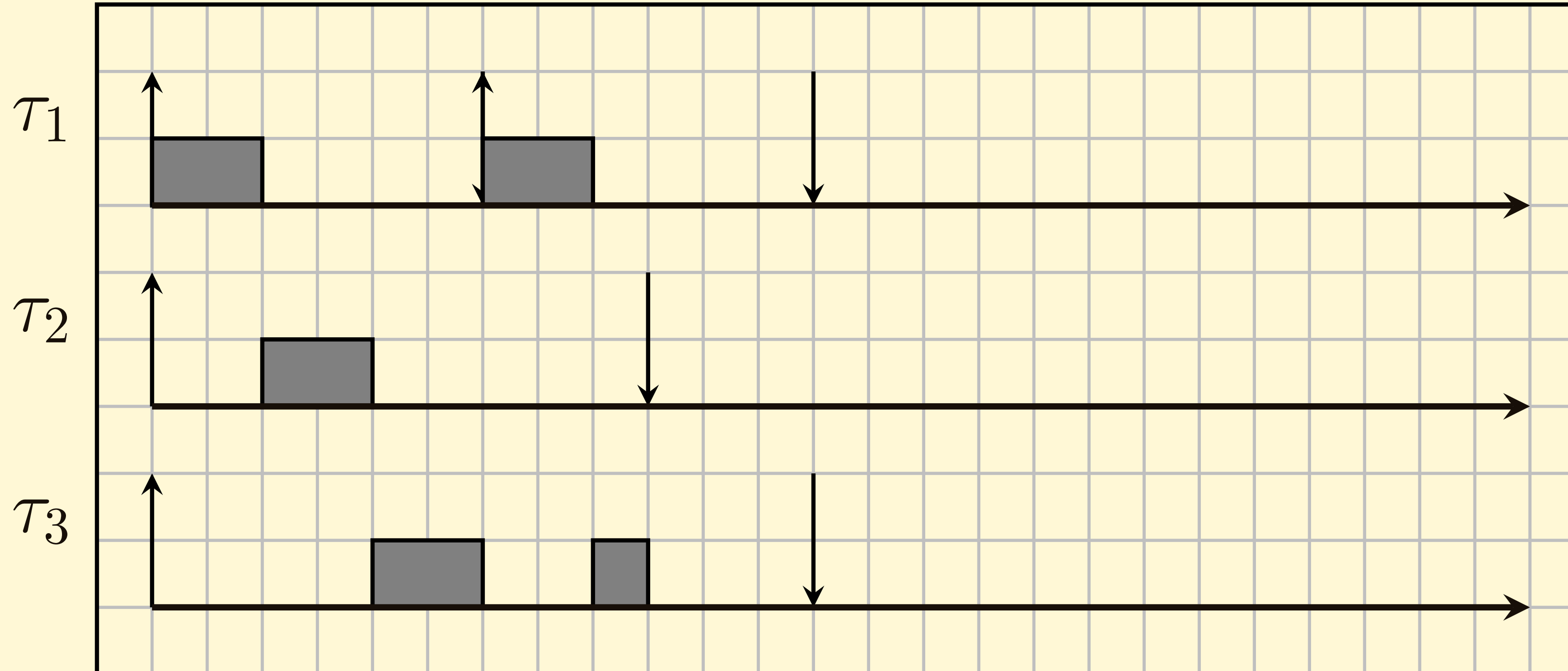
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



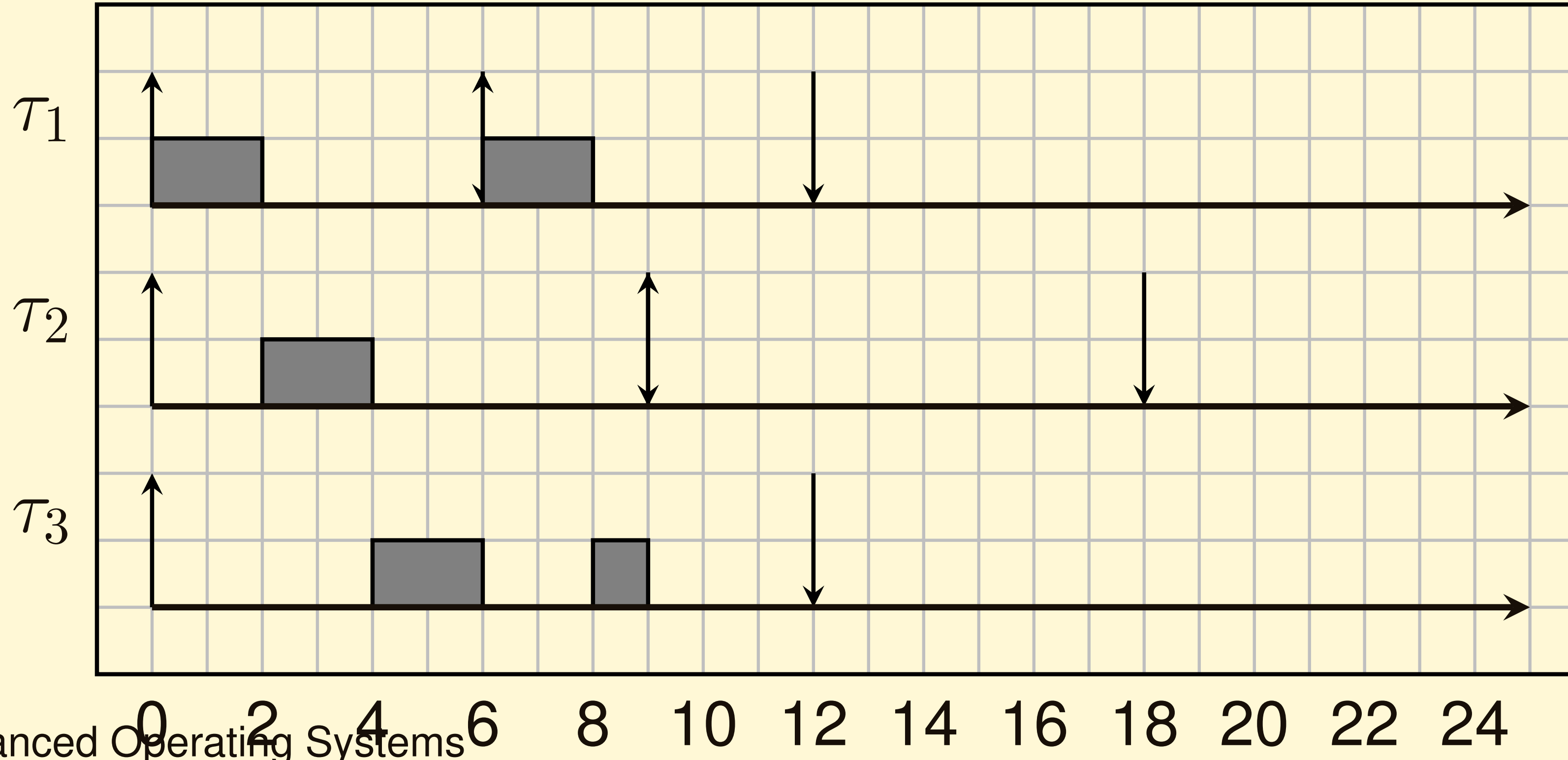
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



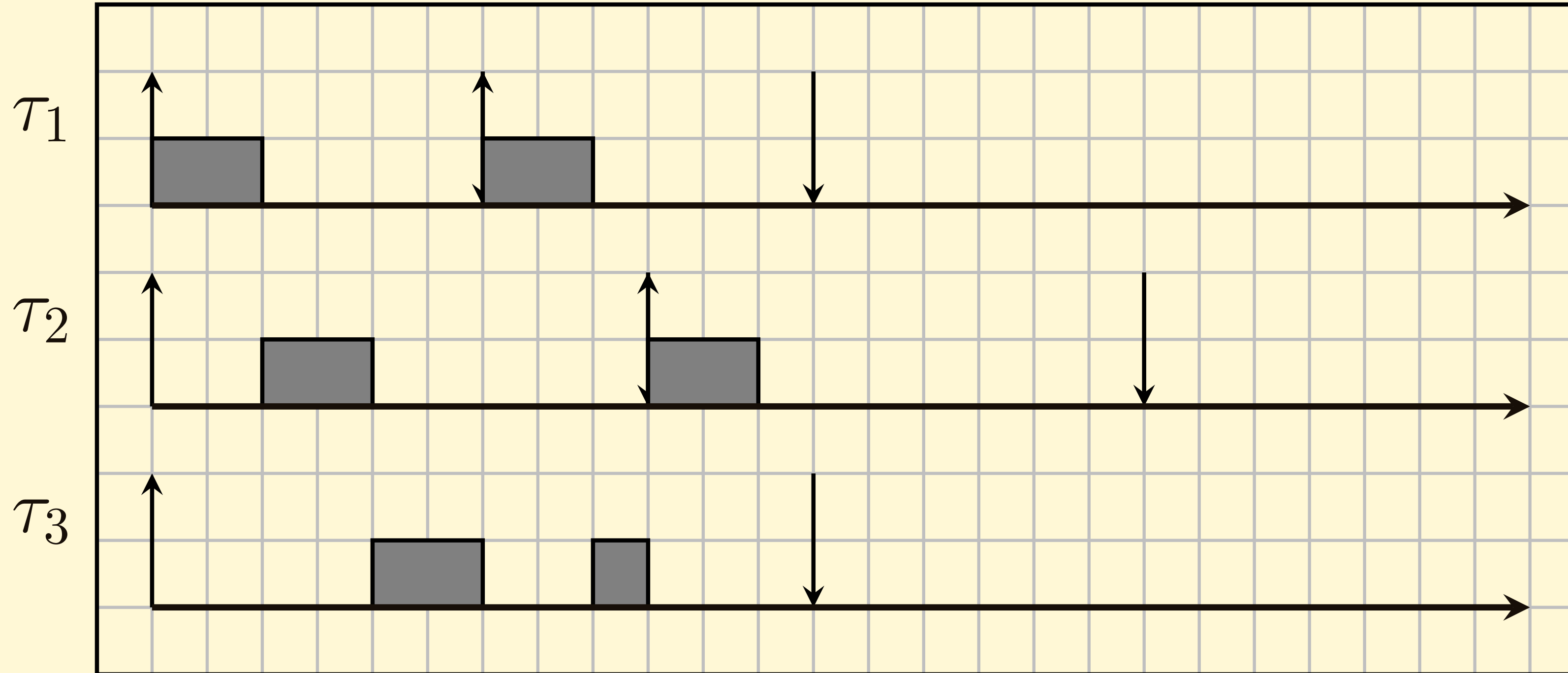
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



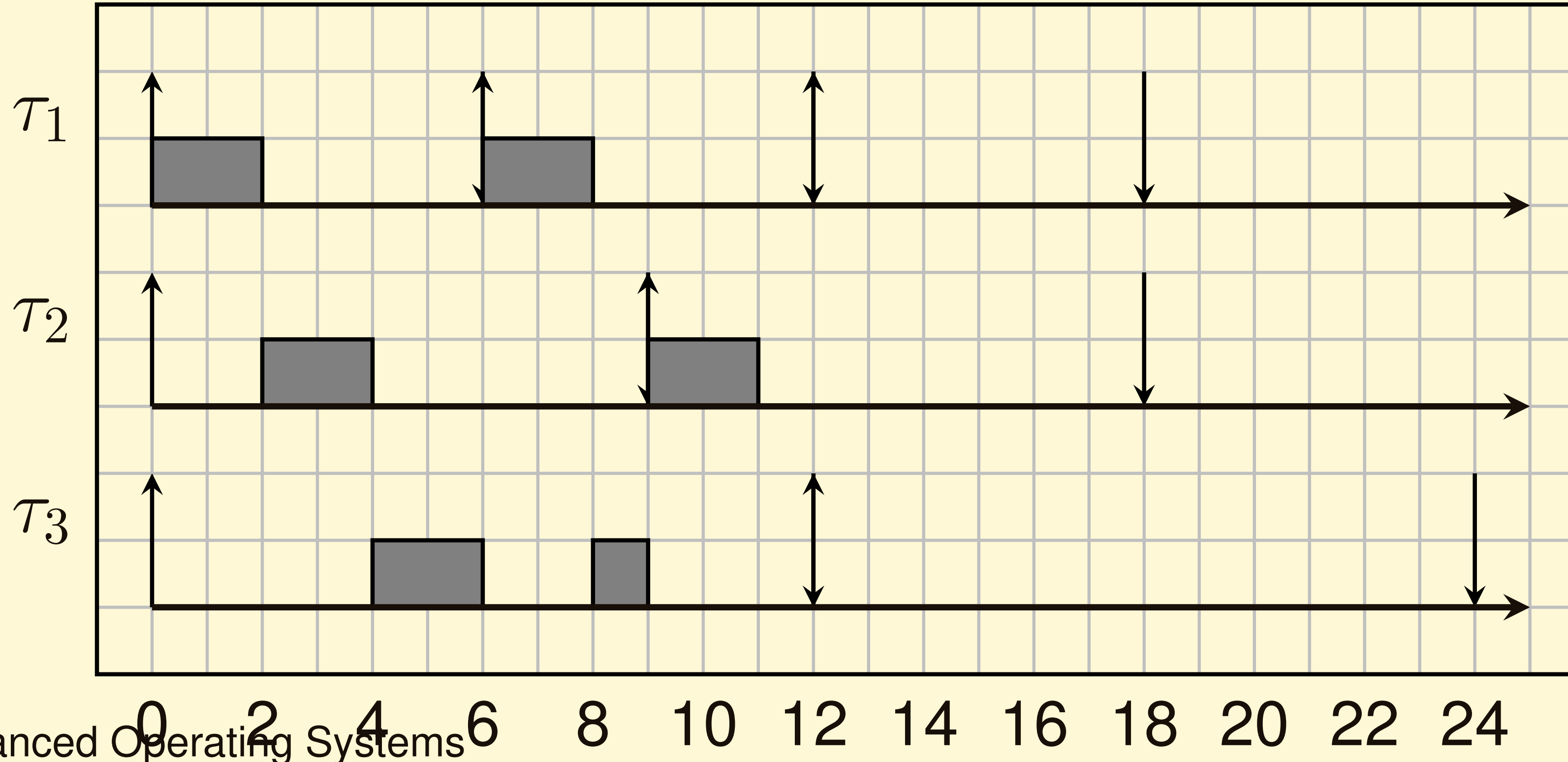
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



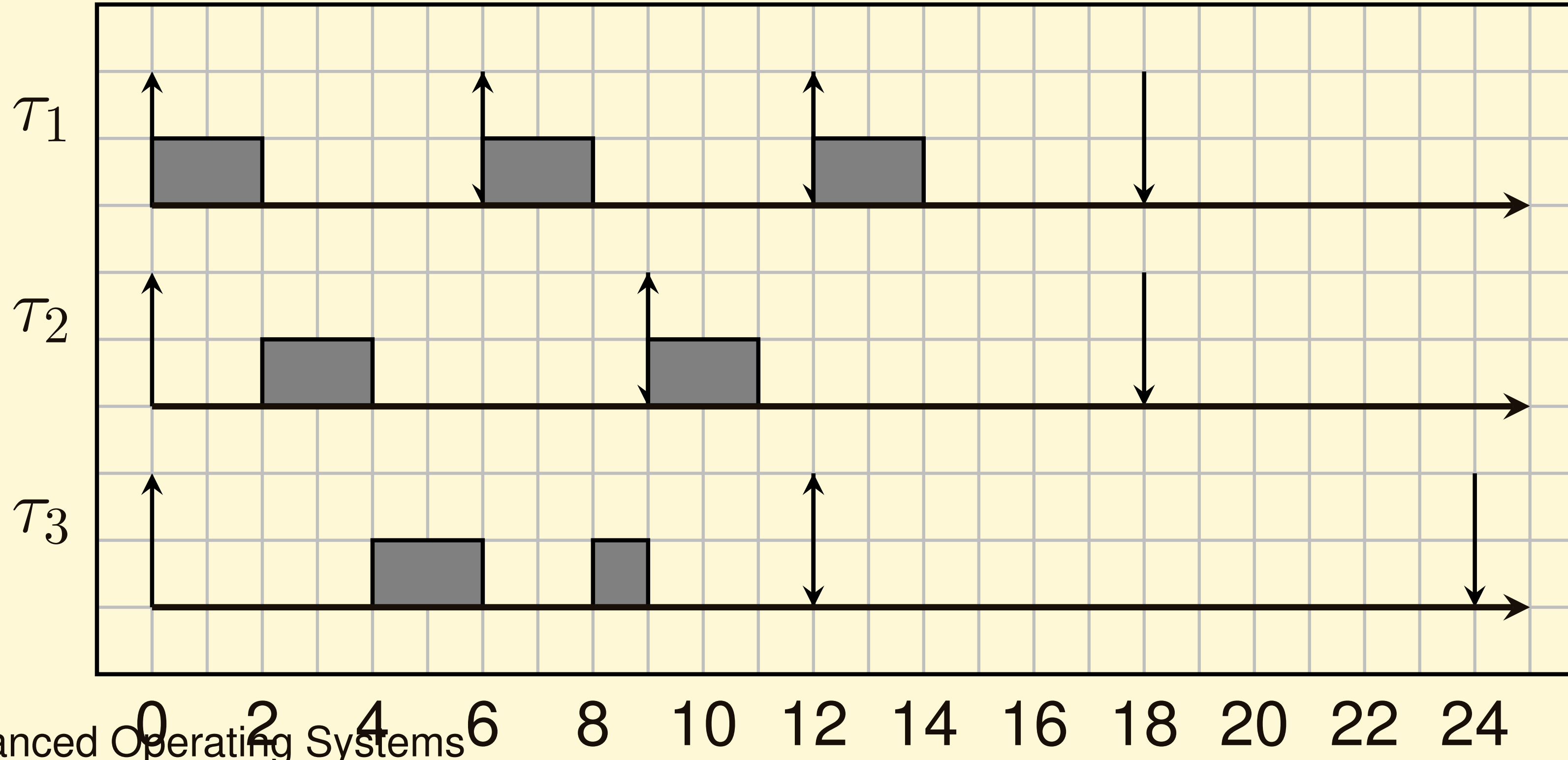
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



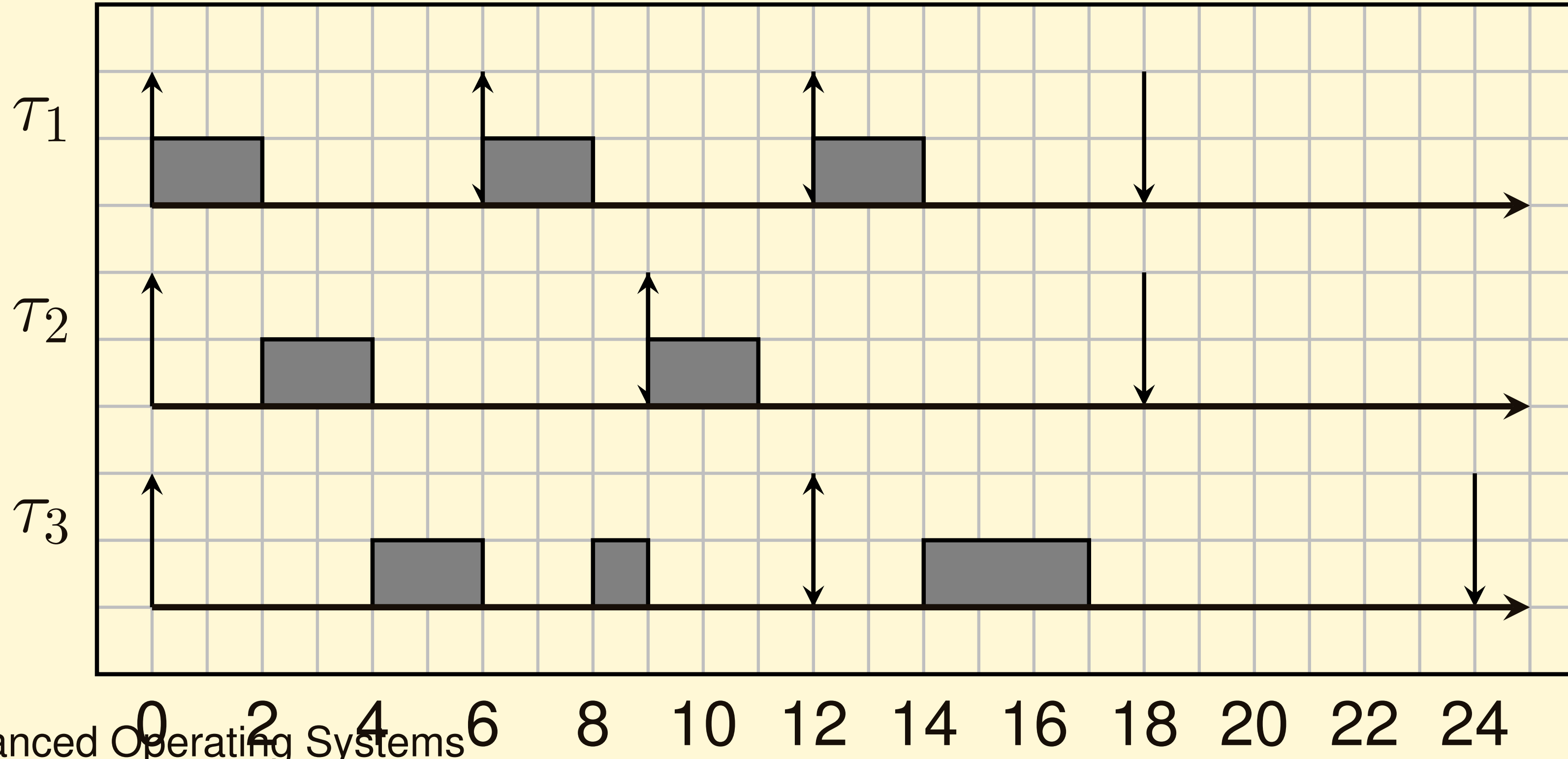
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



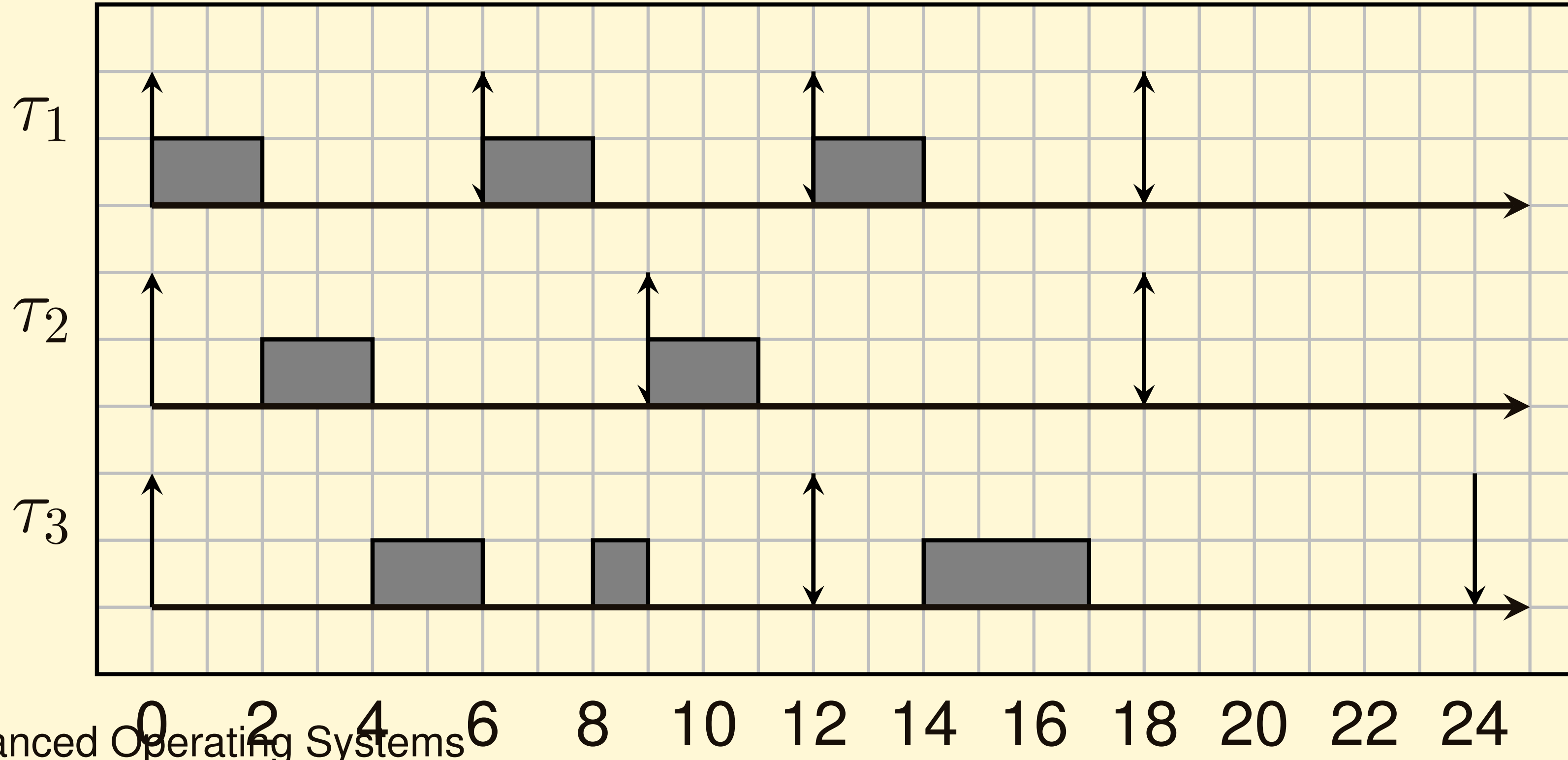
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



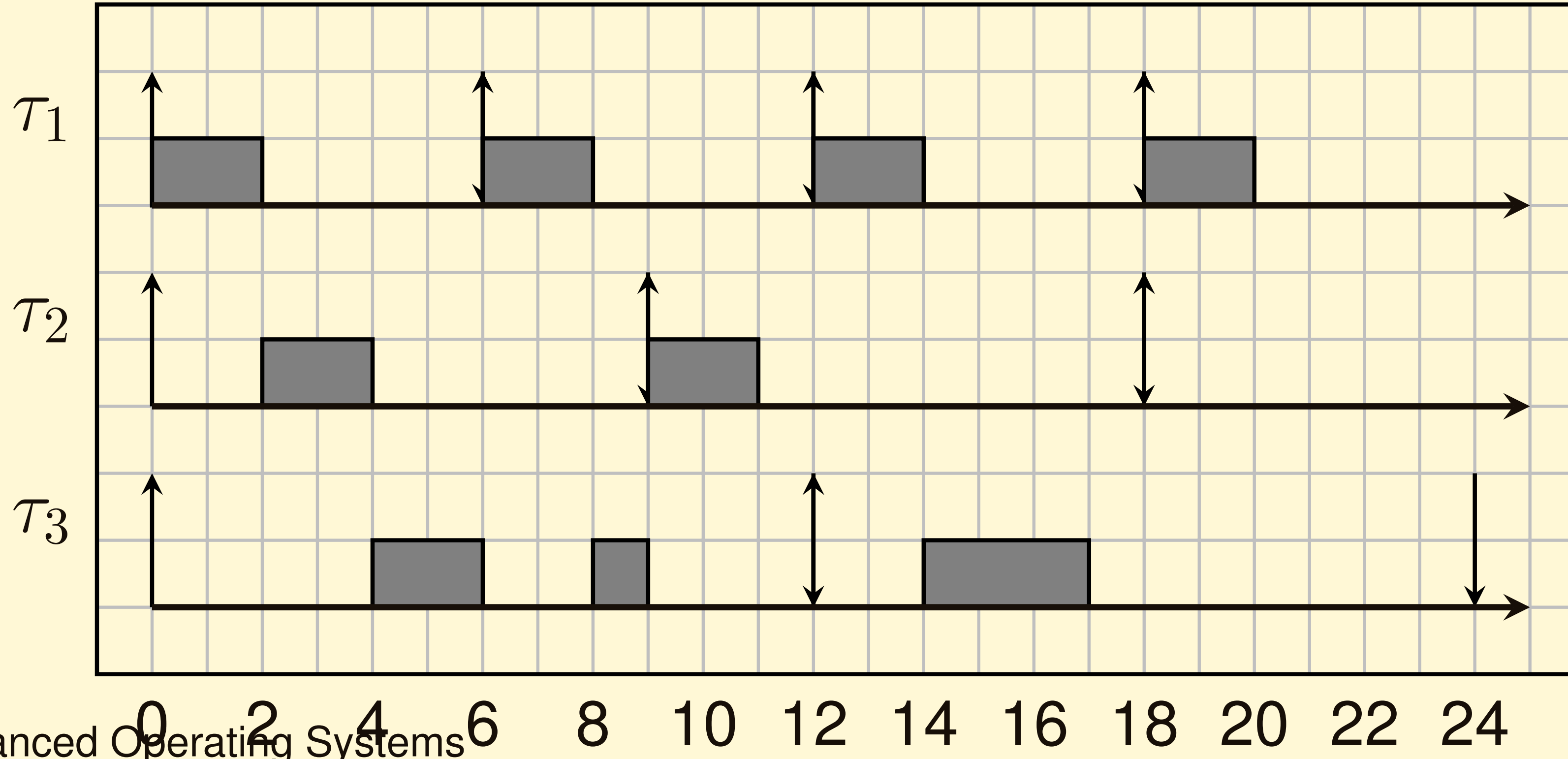
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



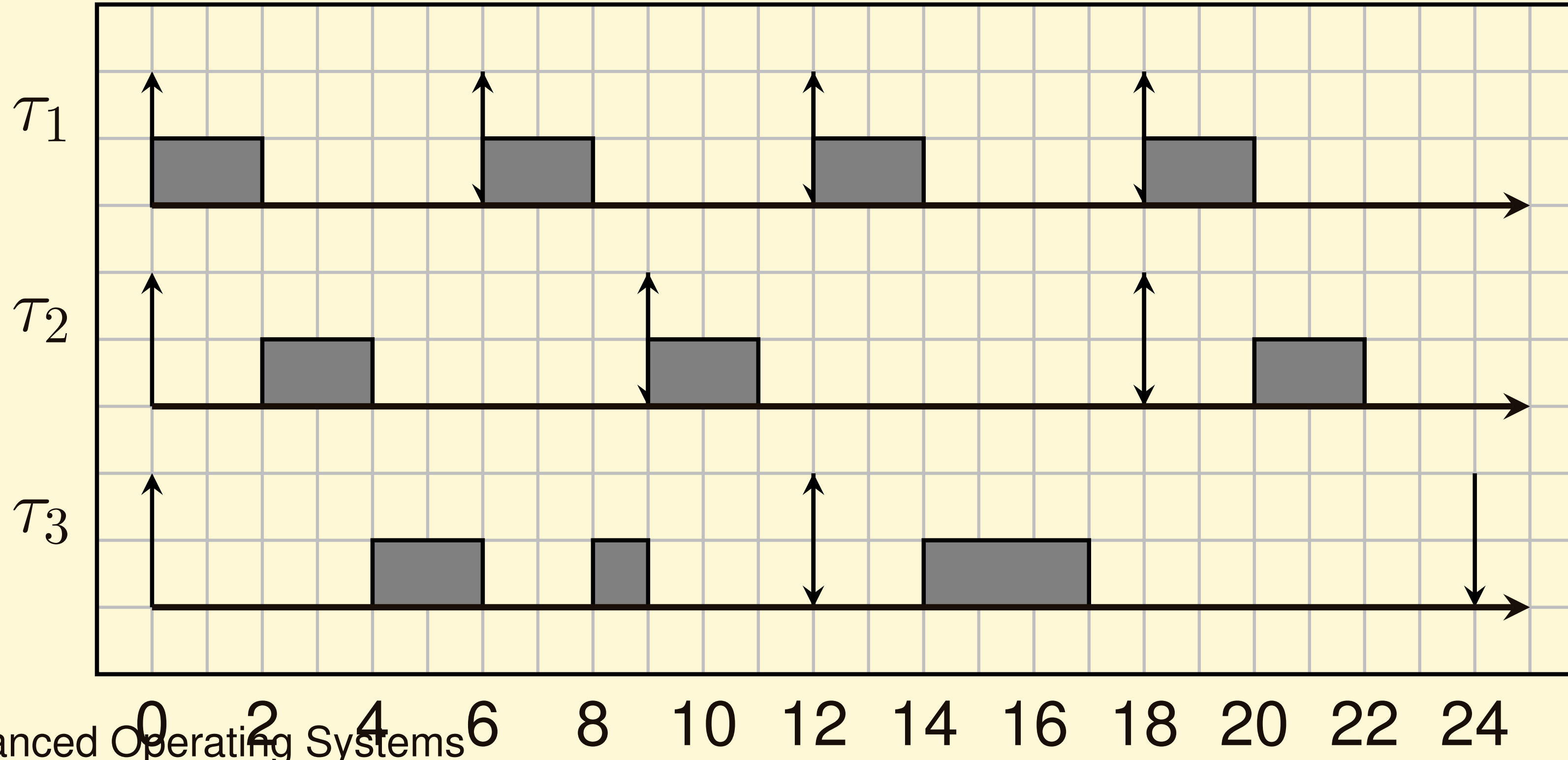
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



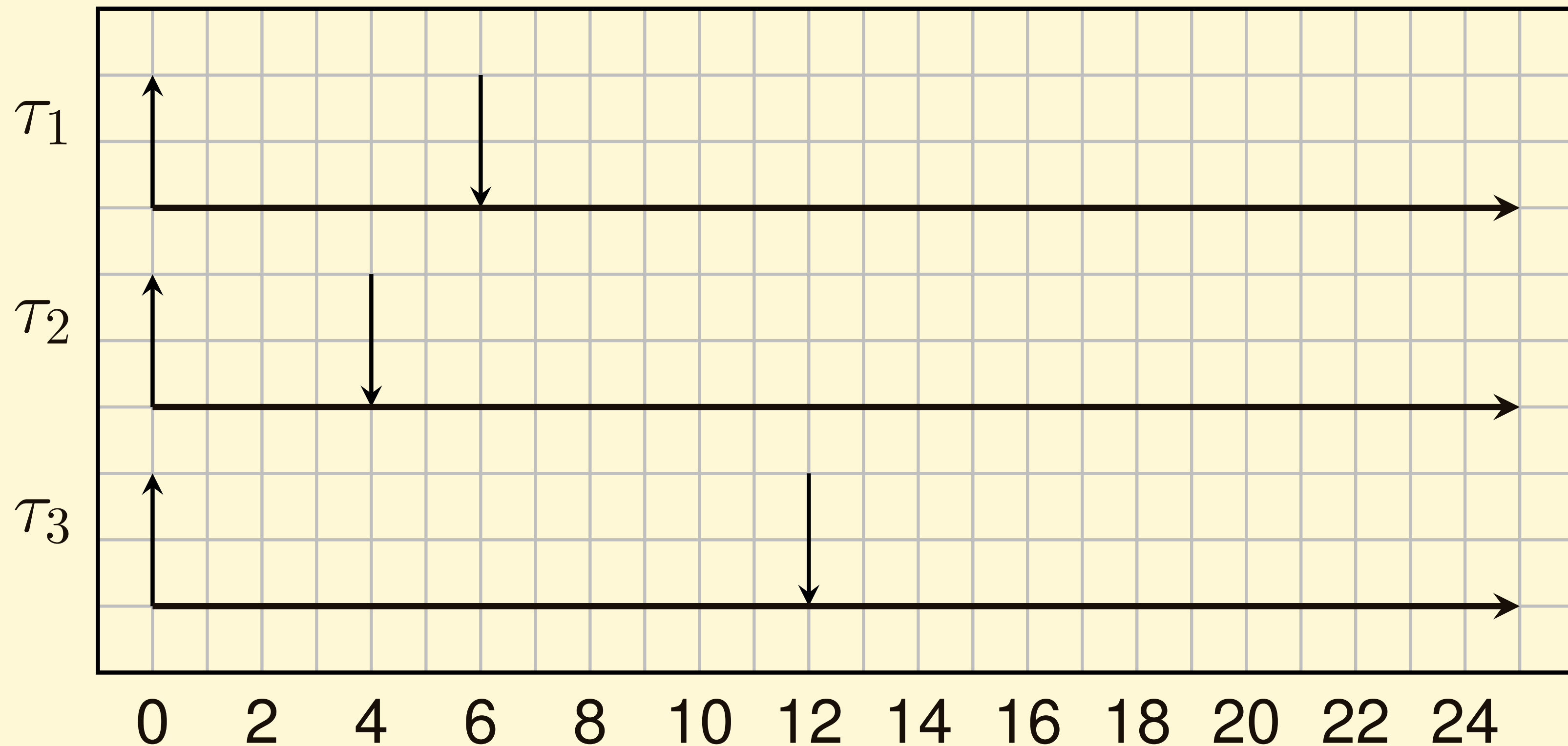
Example of Schedule

- Consider the following task set: $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$. Task τ_1 has priority $p_1 = 3$ (highest), task τ_2 has priority $p_2 = 2$, task τ_3 has priority $p_3 = 1$ (lowest)



Another Example (non-schedulable)

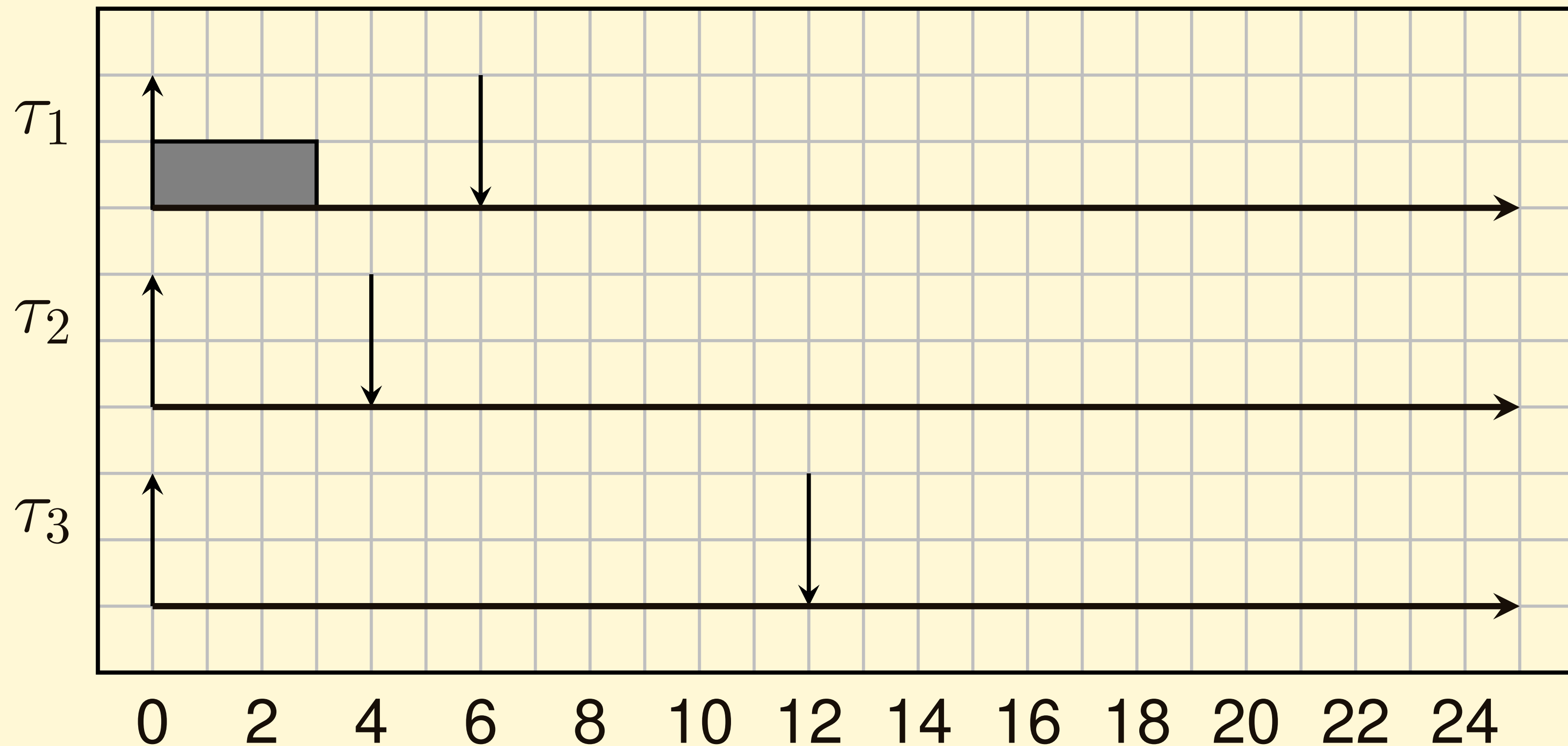
- Consider the following task set: $\tau_1 = (3, 6, 6)$, $p_1 = 3$, $\tau_2 = (2, 4, 8)$, $p_2 = 2$, $\tau_3 = (2, 12, 12)$, $p_3 = 1$



In this case, task τ_2 misses its deadline!

Another Example (non-schedulable)

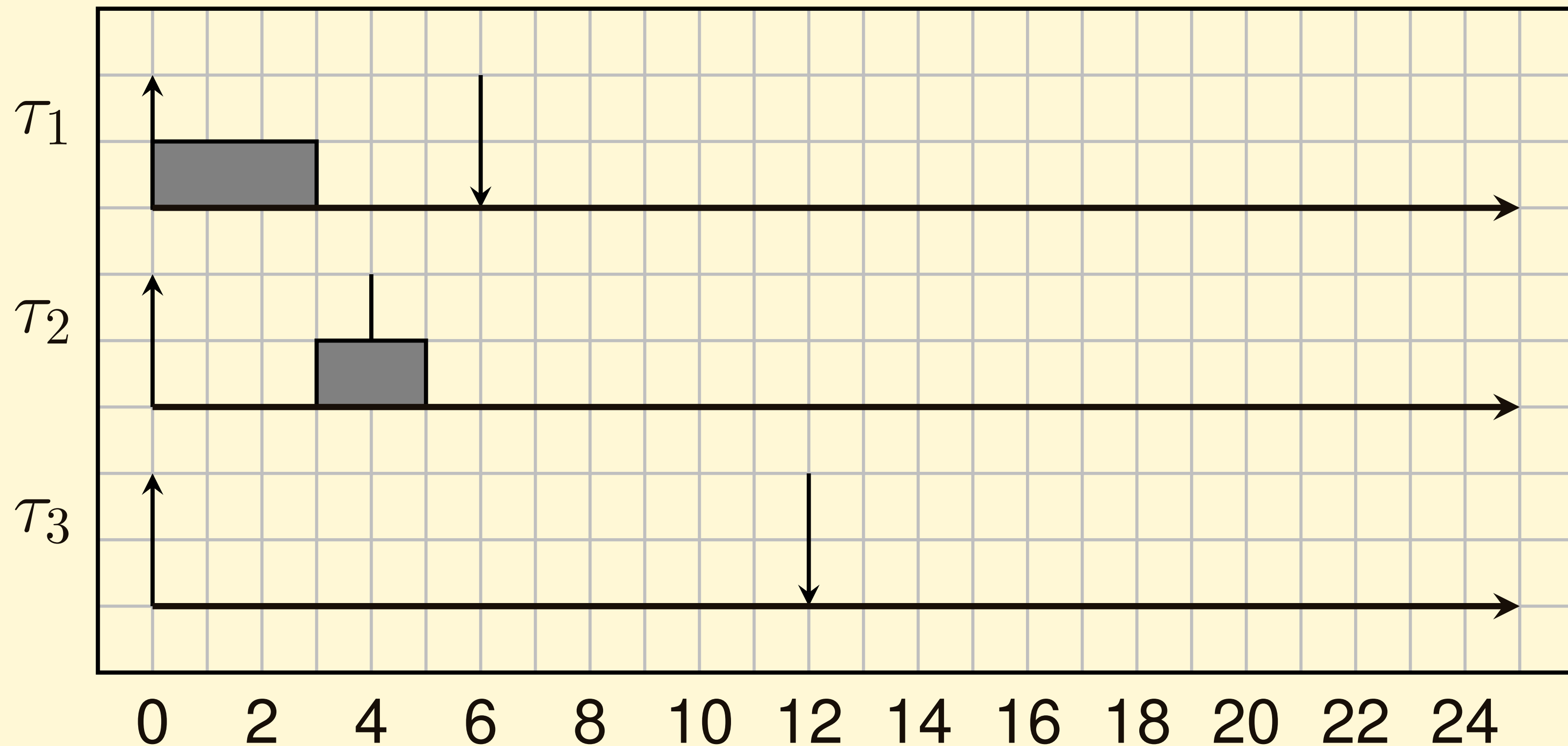
- Consider the following task set: $\tau_1 = (3, 6, 6)$, $p_1 = 3$, $\tau_2 = (2, 4, 8)$, $p_2 = 2$, $\tau_3 = (2, 12, 12)$, $p_3 = 1$



In this case, task τ_2 misses its deadline!

Another Example (non-schedulable)

- Consider the following task set: $\tau_1 = (3, 6, 6)$, $p_1 = 3$, $\tau_2 = (2, 4, 8)$, $p_2 = 2$, $\tau_3 = (2, 12, 12)$, $p_3 = 1$



In this case, task τ_2 misses its deadline!

Notes about Priority Scheduling

- Some considerations about the schedule shown before:
  - The response time of the task with the highest priority is minimum and equal to its WCET
  - The response time of the other tasks depends on the *interference* of the higher priority tasks
  - The priority assignment may influence the schedulability of a task set
    - Problem: how to assign tasks' priorities so that a task set is schedulable?

Response Time Analysis

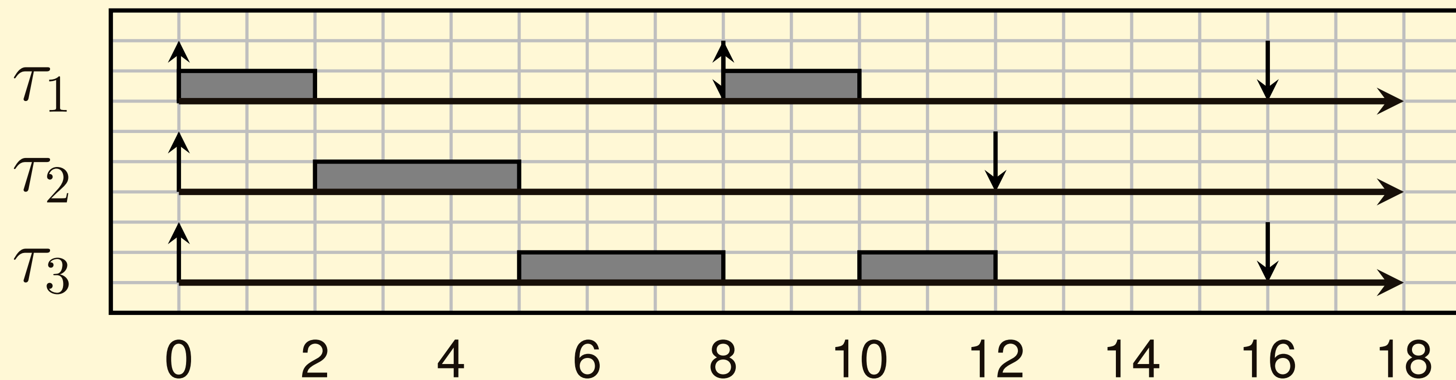
- Necessary and sufficient test: compute the *worst-case response time* for every task
- For every task τ_i:
  - Compute worst case response time R_i for τ_i
    - Remember? $R_i = \max_j \{\rho_{i,j}\}$; $\rho_{i,j} = f_{i,j} - r_{i,j}$
  - If $R_i \leq D_i$, then the task is schedulable
  - otherwise, the task is not schedulable
- No assumption on the priority assignment
  - Algorithm valid for arbitrary priority assignments
  - Not only RM / DM...
- Periodic tasks with no offsets, or sporadic tasks

The Critical Instant

- Tasks ordered by decreasing priority ($i < j \rightarrow p_i > p_j$)
- No assumptions about tasks offsets
  - $\Rightarrow$ Consider the *worst possible offsets combination*
  - A job $J_{i,j}$ released at the *critical instant* experiences the maximum response time for τ_i: $\forall k, \rho_{i,j} \geq \rho_{i,k}$
    - Simplified definition (jobs deadlines should be considered...)
  - **Theorem:** The critical instant for task τ_i occurs when job $J_{i,j}$ is released at the same time with a job in every high priority task
- **If all the offsets are 0, the first job of every task is released at the critical instant!!!**

Worst Case Response Time

- Worst case response time R_i for task τ_i depends on:
  - Its execution time...
  - ...And the execution time of higher priority tasks
    - Higher priority tasks can *preempt* task τ_i, and increase its response time

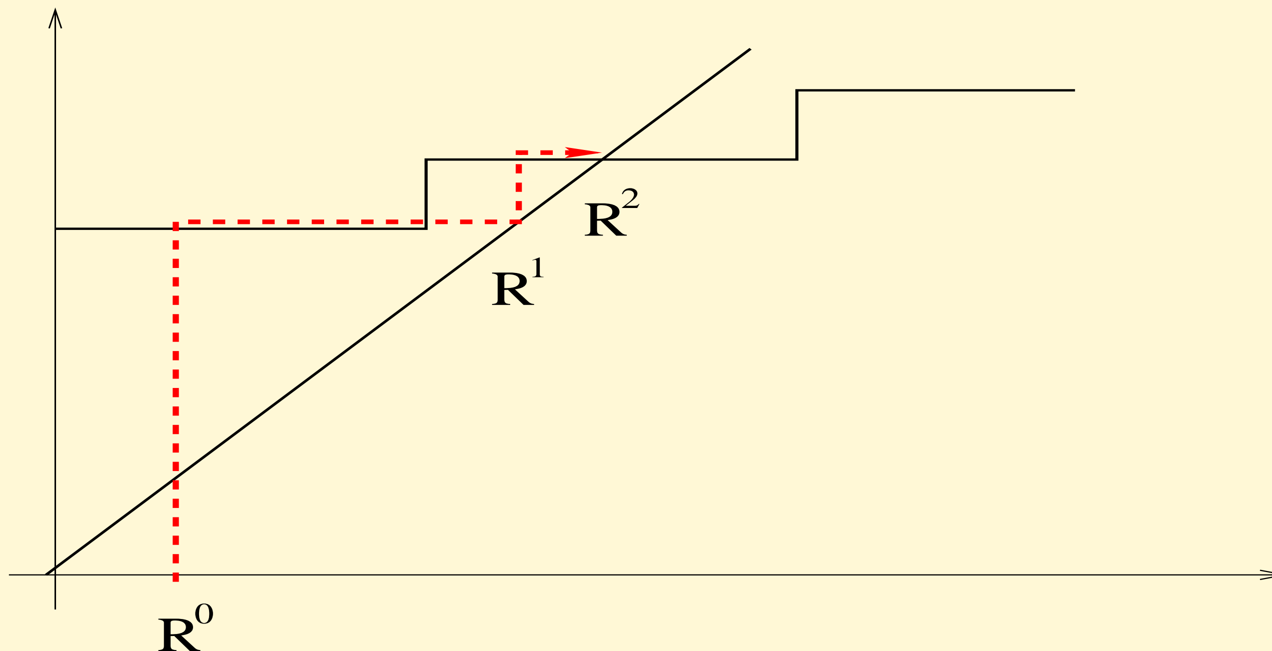


- $R_i = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$

Computing the Response Time - I

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

- Urk!!! $R_i = f(R_i)$... How can we solve it?
- There is no closed-form expression for computing the worst case response time R_i
- We need an iterative method to solve the equation



Computing the Response Time - II

- Iterative solution
  - $R_i = \lim_{k\to\infty} R_i^{(k)}$
  - $R_i^{(k)}$: worst case response time for τ_i, at step k
- $R_i^{(0)}$: first estimation of the response time
  - We can start with $R_i^{(0)} = C_i$
  - $R_i^{(0)} = C_i + \sum_{h=1}^{i-1} C_h$ saves 1 step

$$
\begin{aligned}
R_i^{(0)} &= C_i(+\sum_{h=1}^{i-1} C_h) \\
R_i^{(k)} &= C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i^{(k-1)}}{T_h} \right\rceil C_h
\end{aligned}
$$

Computing the Response Time - III

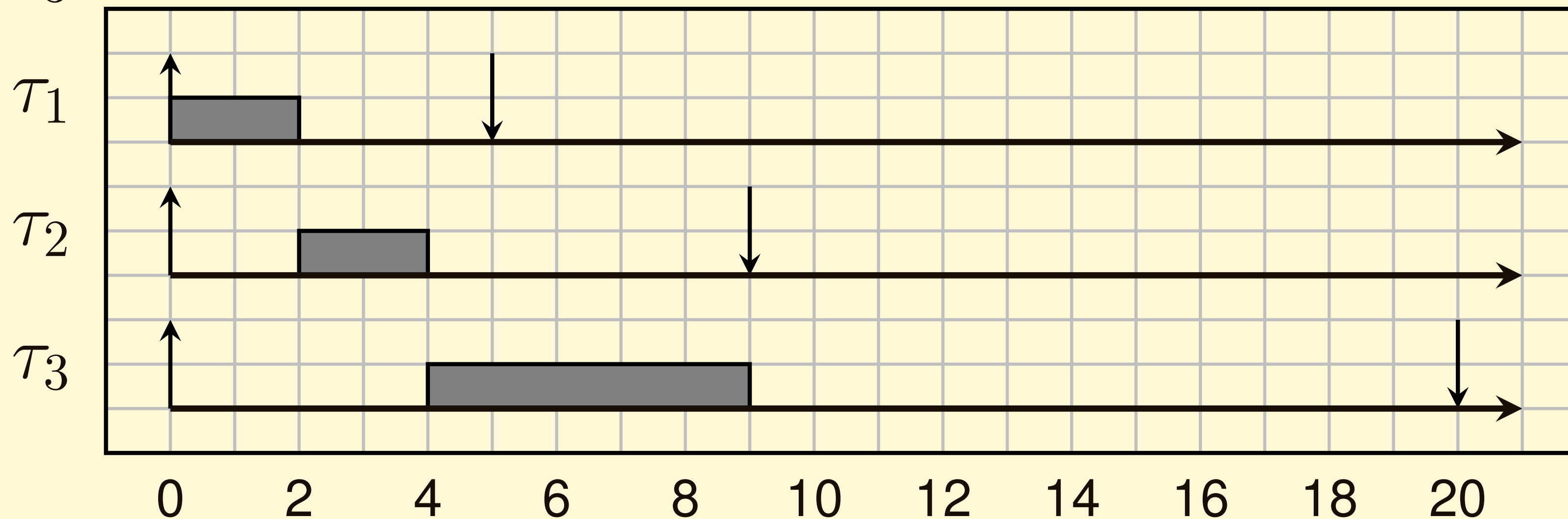
- Problem: are we sure that we find a valid solution?
- The iteration stops when:
  - $R_i^{(k+1)} = R_i^{(k)}$ *or*
  - $R_i^{(k)} > D_i$ (non schedulable);
- This is a standard method to solve non-linear equations in an iterative way
- If a solution exists (the system is not overloaded), $R_i^{(k)}$ converges to it
- Otherwise, the "$R_i^{(k)} > D_i$" condition avoids infinite iterations

Example

Task set: $\tau_1 = (2,5)$, $\tau_2 = (2,9)$, $\tau_3 = (5,20)$; $U = 0.872$

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

$$R_3^{(0)} = C_3 + 1 \cdot C_1 + 1 \cdot C_2 = 9$$

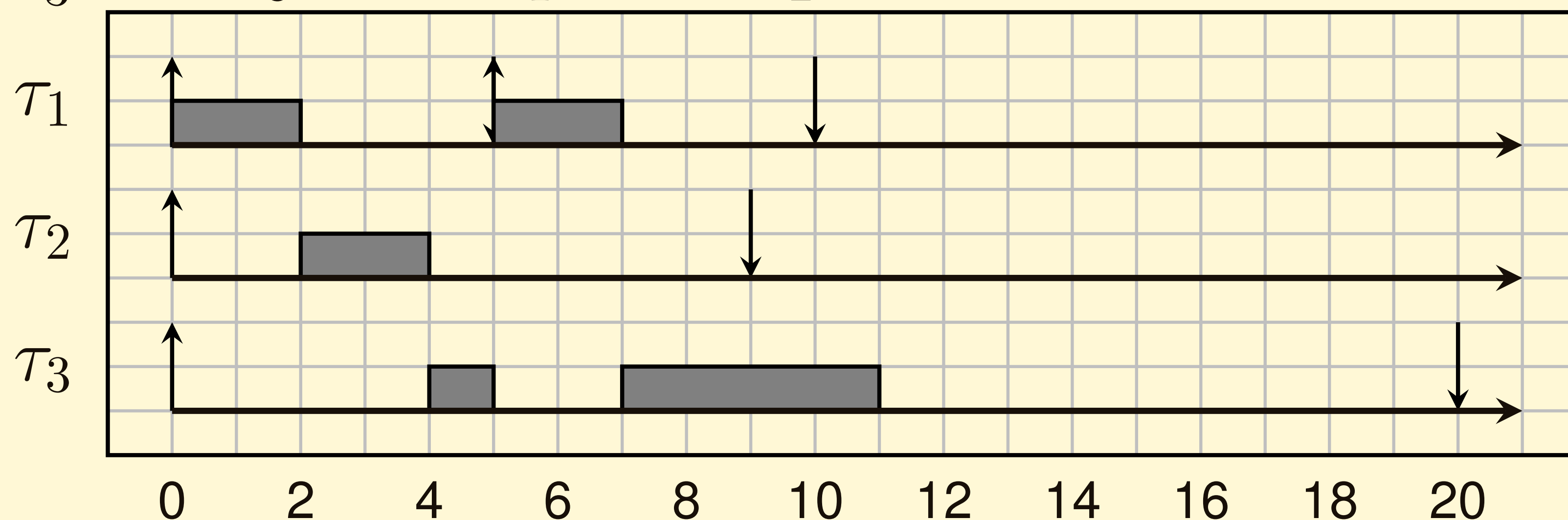


Example

Task set: $\tau_1 = (2, 5)$, $\tau_2 = (2, 9)$, $\tau_3 = (5, 20)$; $U = 0.872$

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

$$R_3^{(1)} = C_3 + 2 \cdot C_1 + 1 \cdot C_2 = 11$$

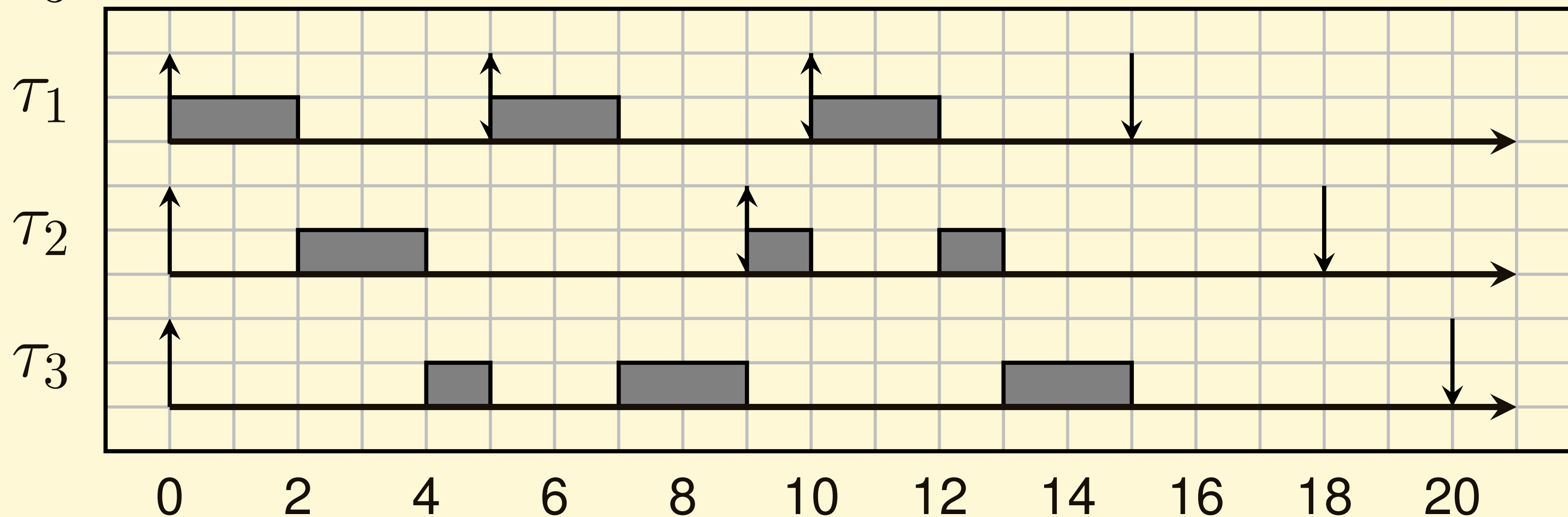


Example

Task set: $\tau_1 = (2,5)$, $\tau_2 = (2,9)$, $\tau_3 = (5,20)$; $U = 0.872$

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

$R_3^{(2)} = C_3 + 3 \cdot C_1 + 2 \cdot C_2 = 15$

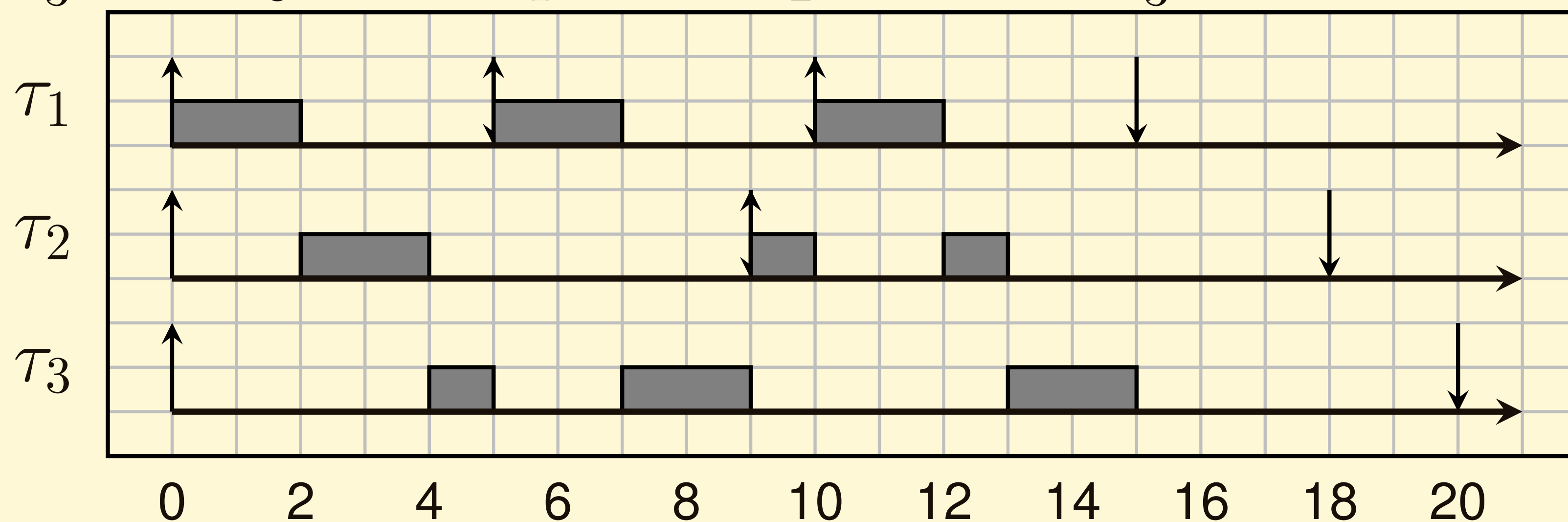


Example

Task set: $\tau_1 = (2,5)$, $\tau_2 = (2,9)$, $\tau_3 = (5,20)$; $U = 0.872$

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

$$R_3^{(3)} = C_3 + 3 \cdot C_1 + 2 \cdot C_2 = 15 = R_3^{(2)}$$



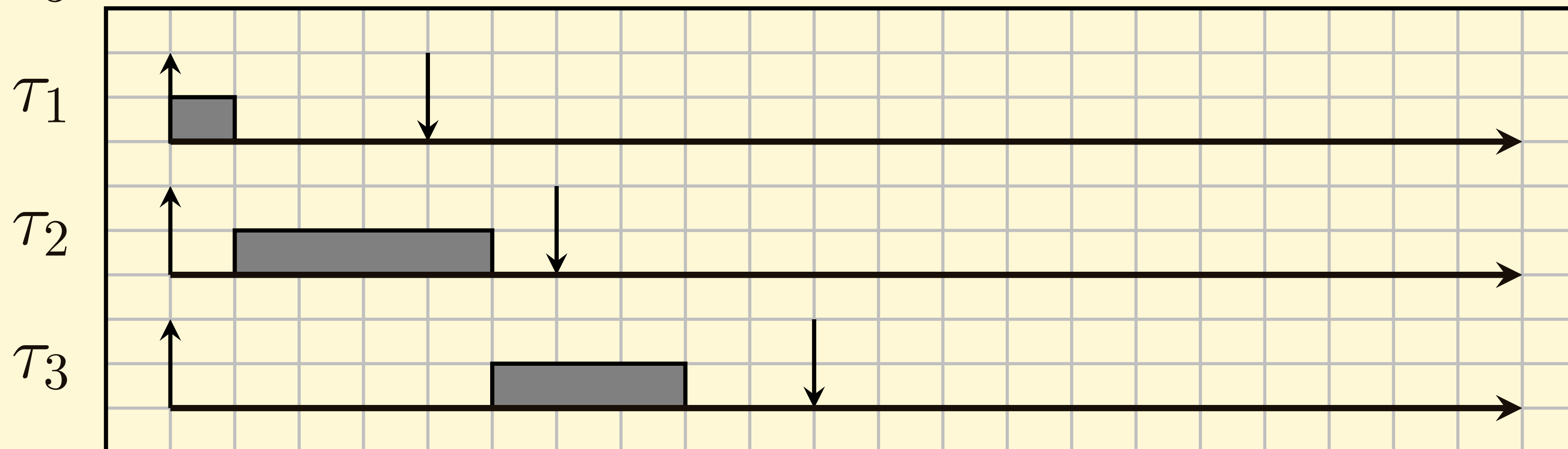
Another Example with DM

What about different priority assignments and deadlines different from periods?

$\tau_1 = (1, 4, 4), p_1 = 3$, $\tau_2 = (4, 6, 15), p_2 = 2$,
$\tau_3 = (3, 10, 10), p_3 = 1; U = 0.72$

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

$R_3^{(0)} = C_3 + 1 \cdot C_1 + 1 \cdot C_2 = 8$



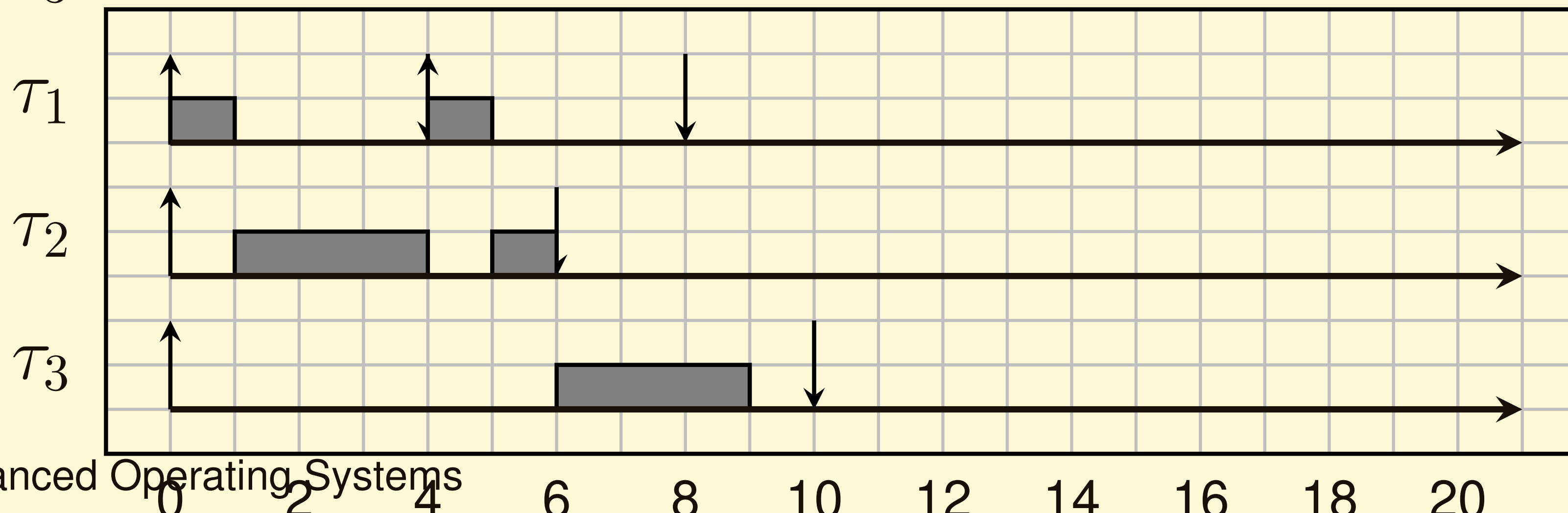
Another Example with DM

What about different priority assignments and deadlines different from periods?

$\tau_1 = (1, 4, 4), p_1 = 3$, $\tau_2 = (4, 6, 15), p_2 = 2$,
$\tau_3 = (3, 10, 10), p_3 = 1; U = 0.72$

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

$R_3^{(1)} = C_3 + 2 \cdot C_1 + 1 \cdot C_2 = 9$



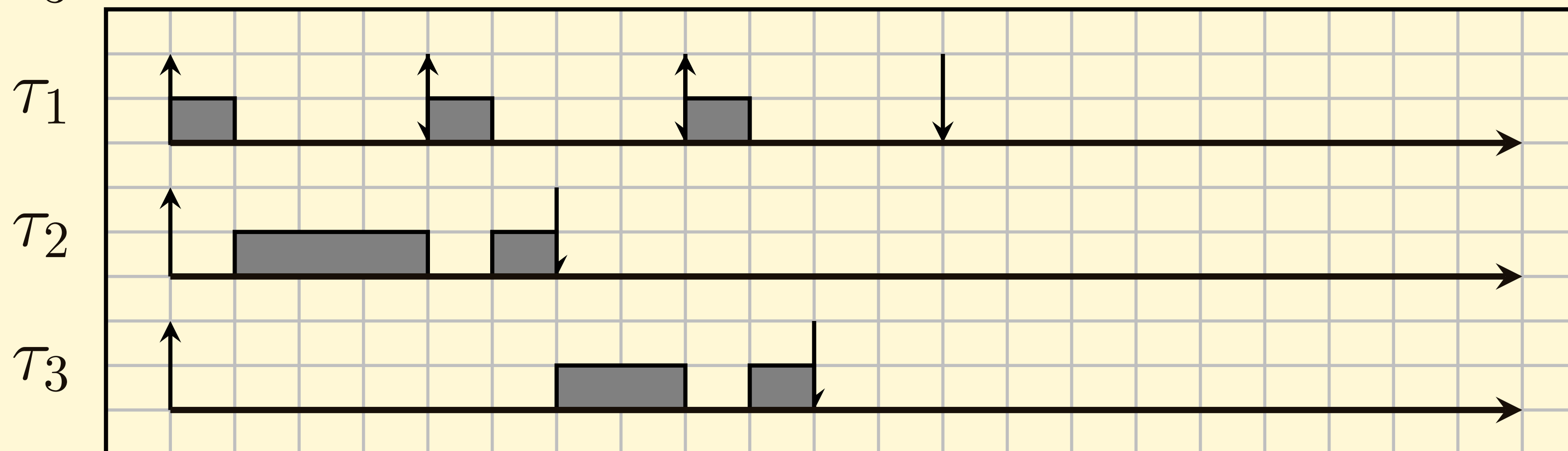
Another Example with DM

What about different priority assignments and deadlines different from periods?

$\tau_1 = (1, 4, 4), p_1 = 3$, $\tau_2 = (4, 6, 15), p_2 = 2$,
$\tau_3 = (3, 10, 10), p_3 = 1; U = 0.72$

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

$R_3^{(2)} = C_3 + 3 \cdot C_1 + 1 \cdot C_2 = 10$



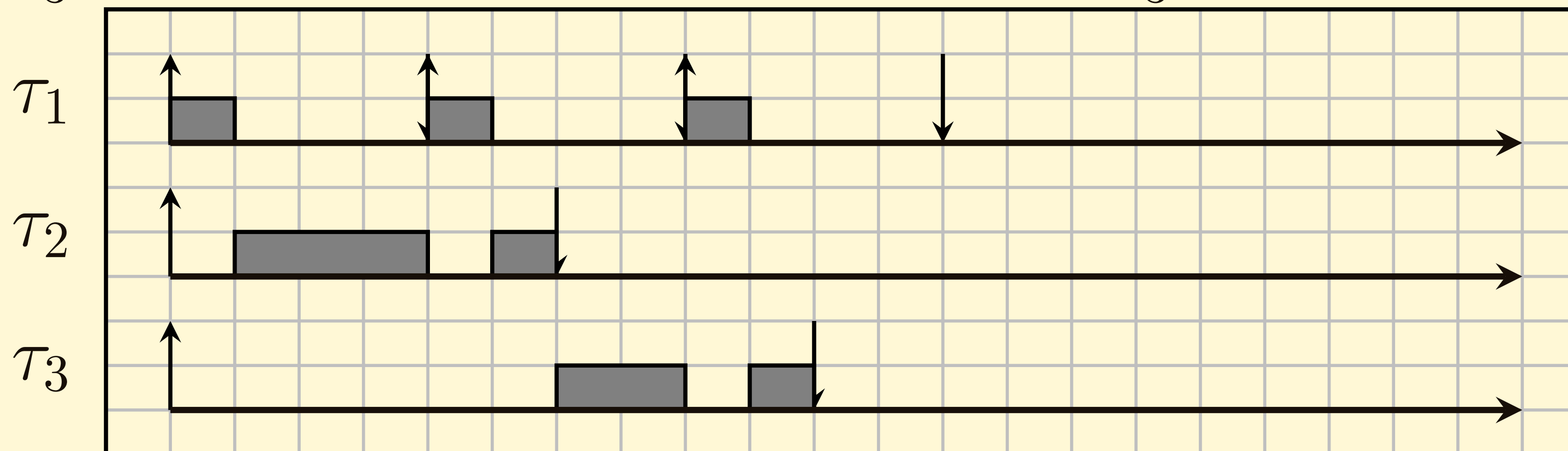
Another Example with DM

What about different priority assignments and deadlines different from periods?

$\tau_1 = (1,4,4), p_1 = 3$, $\tau_2 = (4,6,15), p_2 = 2$,
$\tau_3 = (3,10,10), p_3 = 1; U = 0.72$

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

$R_3^{(3)} = C_3 + 3 \cdot C_1 + 1 \cdot C_2 = 10 = R_3^{(2)}$



Considerations

- The response time analysis is an efficient algorithm
  - In the worst case, the number of steps N for the algorithm to converge is exponential
    - Depends on the total number of jobs of higher priority tasks in the interval $[0, D_i]$:

$$N \propto \sum_{h=1}^{i-1} \left\lceil \frac{D_h}{T_h} \right\rceil$$

    - If s is the minimum granularity of the time, then in the worst case $N = \frac{D_i}{s}$;
  - However, such worst case is very rare: usually, the number of steps is low.

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 $\Rightarrow$ 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

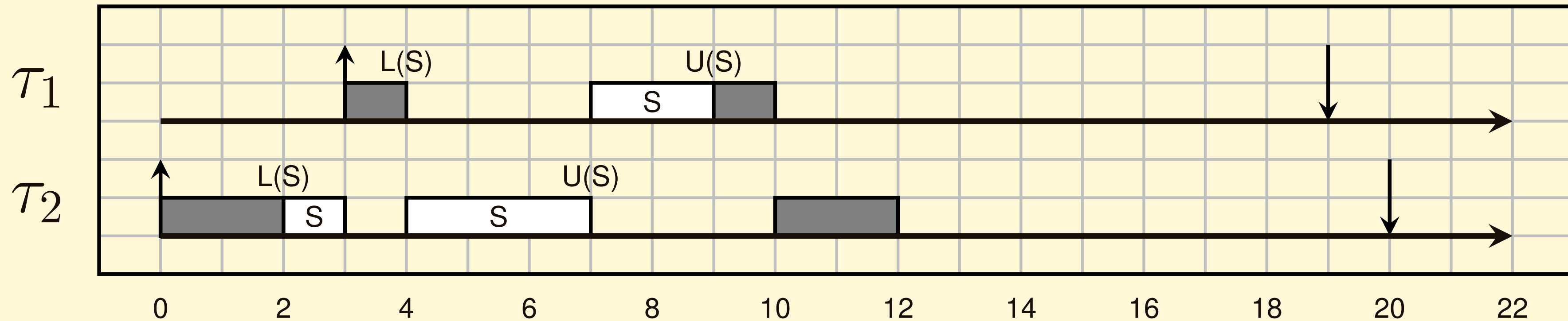
- Mutual exclusion on a shared resource can cause *blocking time*
    - When task τ_i tries to access a resource S already held from task τ_j, τ_i blocks
    - Blocking time: time between the instant when τ_i tries to access S (and blocks) and the instant when τ_j releases S (and τ_i unblocks)
- This 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

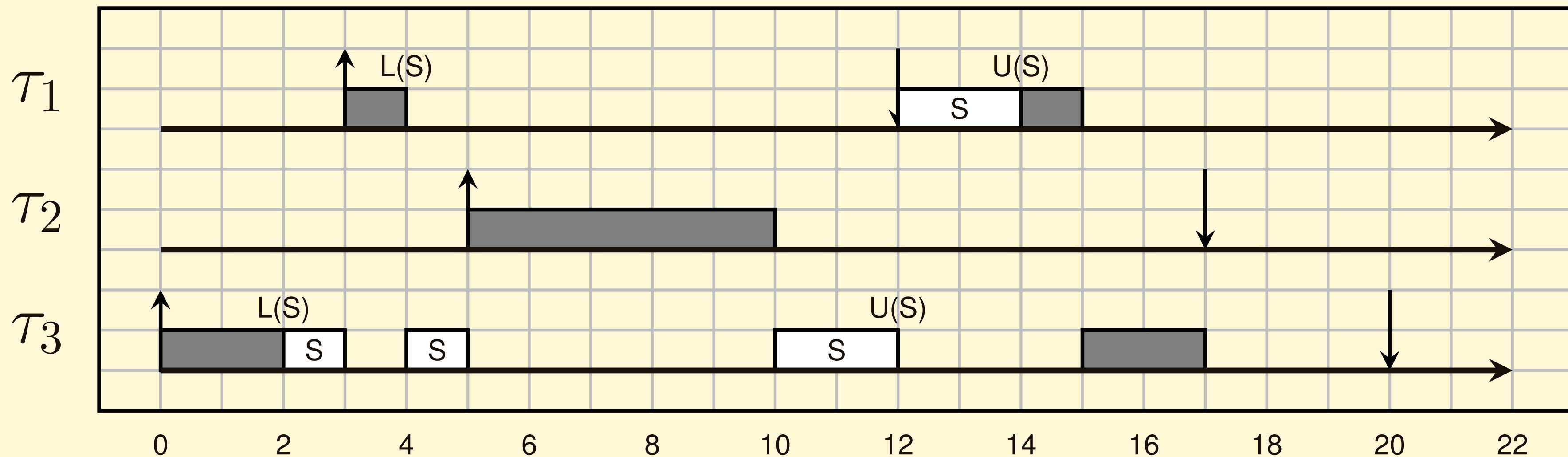
- Consider the following example, where $p_1 > p_2$.



- 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/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

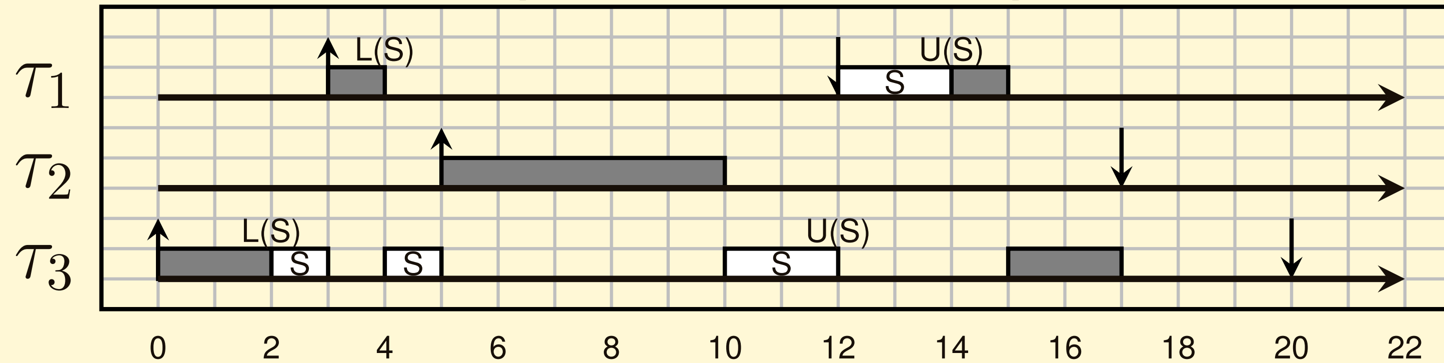
Non Preemptive Protocol (NPP)

- 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

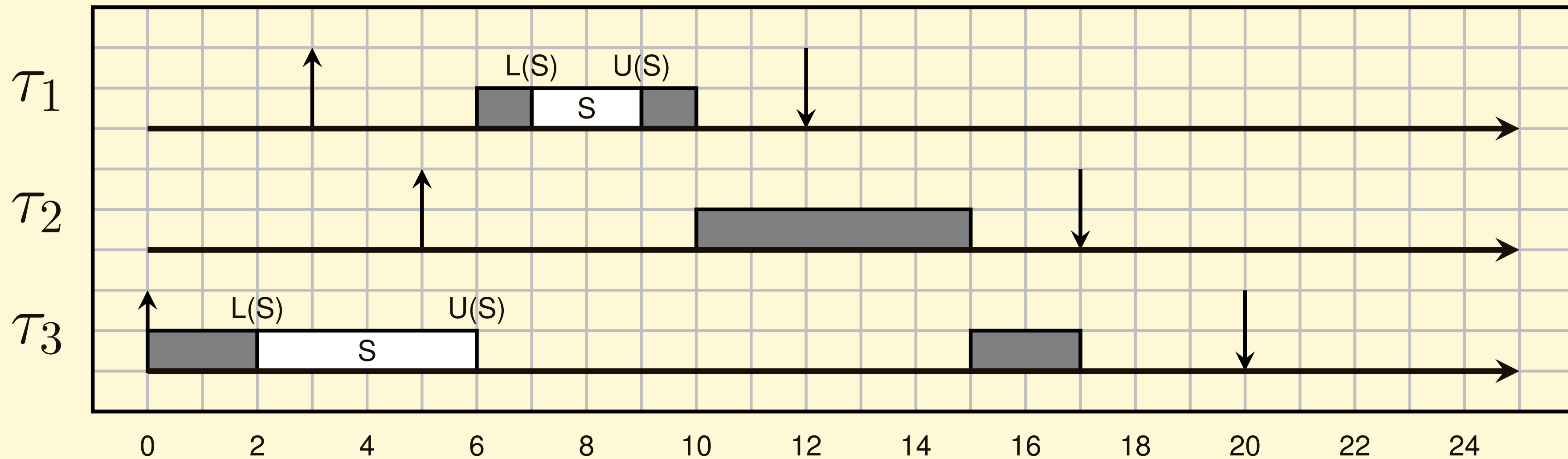
- Advantages: *simplicity*
- Drawbacks: tasks which are not involved in a critical section suffer blocking

NPP Example

- Remember the previous example...



- Using NPP, we have:

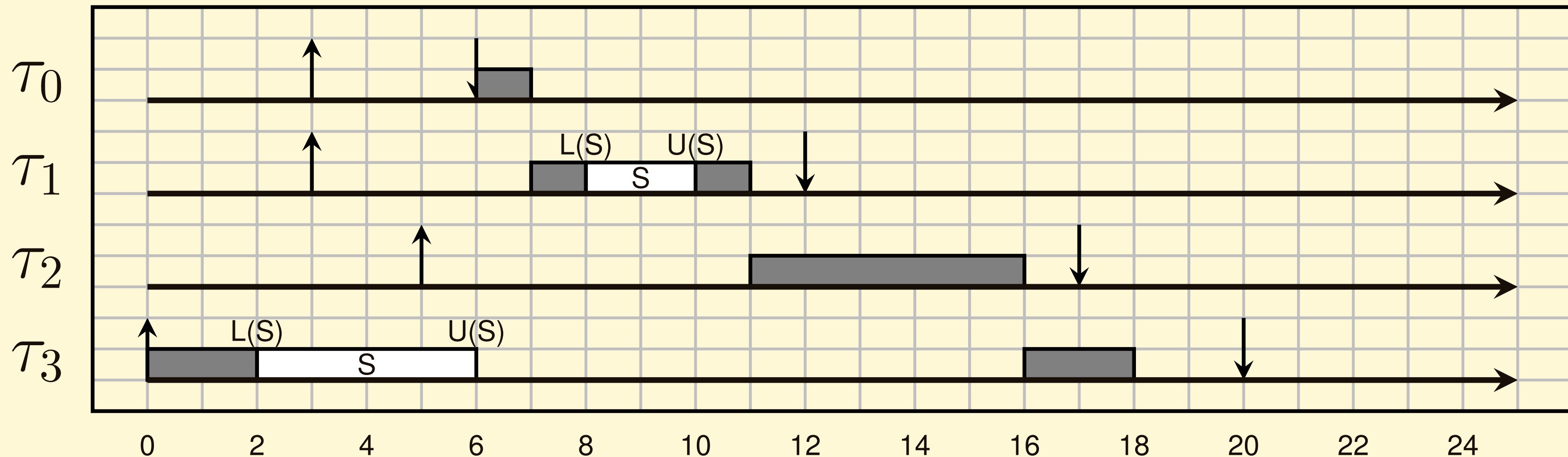


Some Observations

- 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

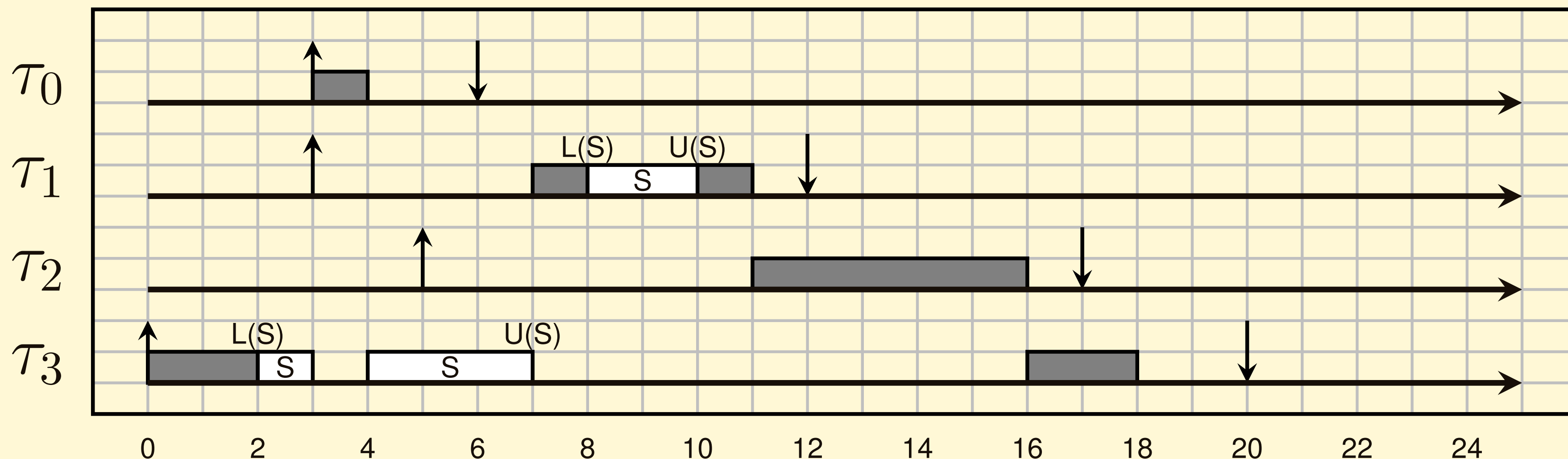
- Consider the following example, with $p_0 > p_1 > p_2 > p_3$.



- 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

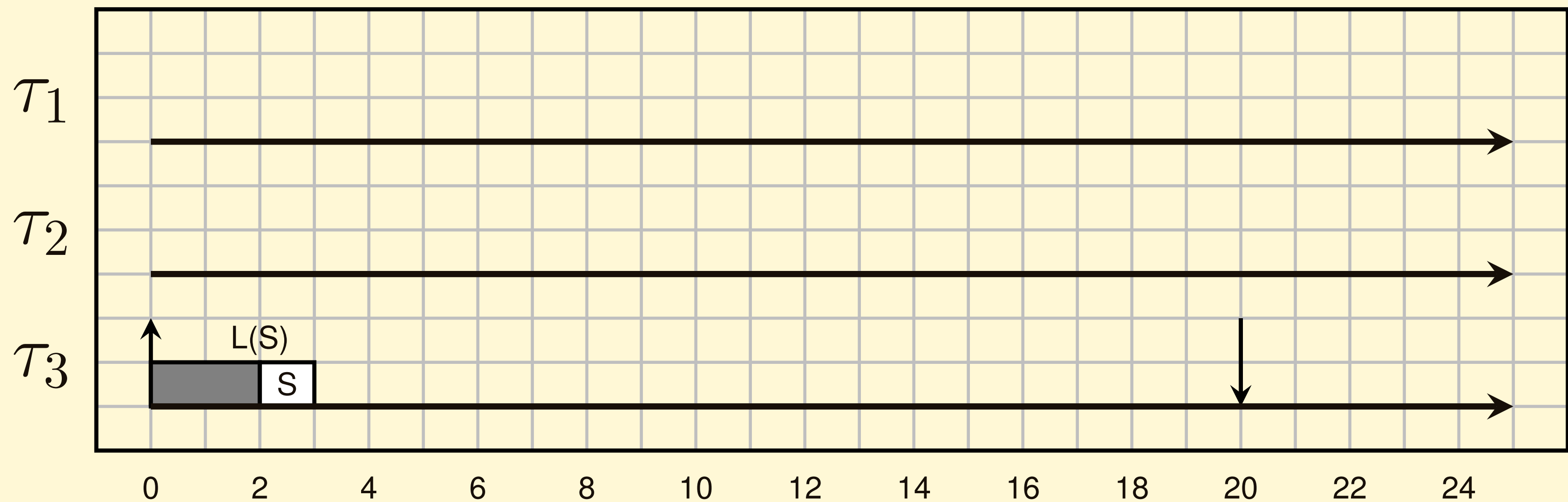
Task	C_i	T_i	$\xi_{i,1}$	D_i	B_i
τ_1	20	70	0	30	2
τ_2	20	80	1	45	2
τ_3	35	200	2	130	0

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

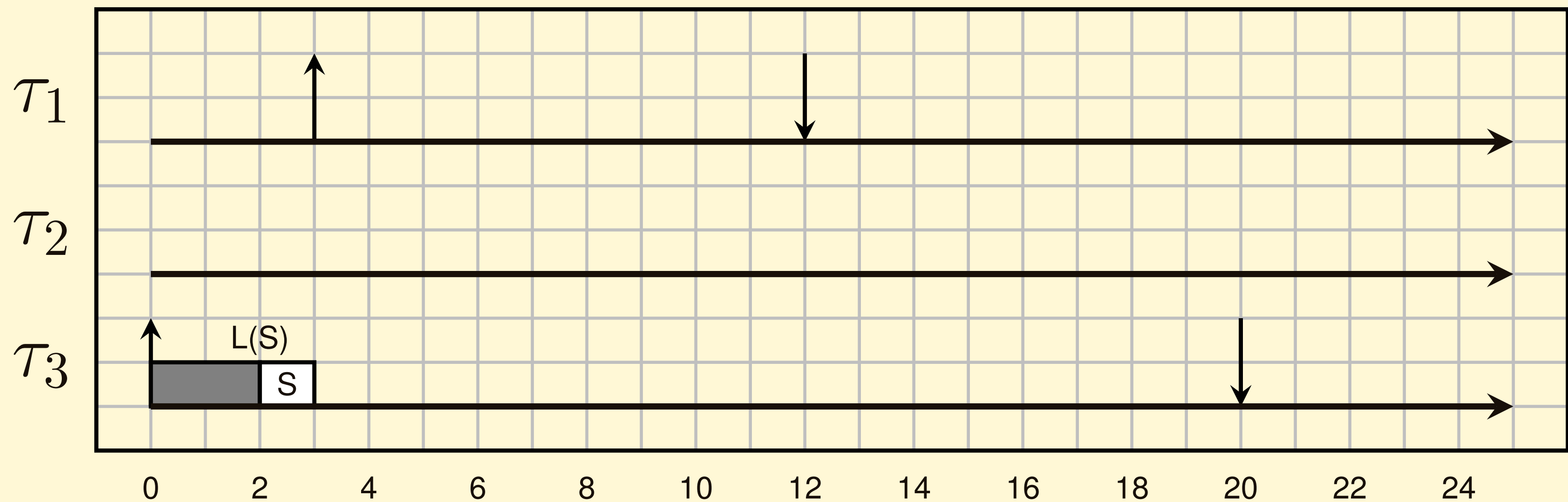
The Priority Inheritance protocol

- 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



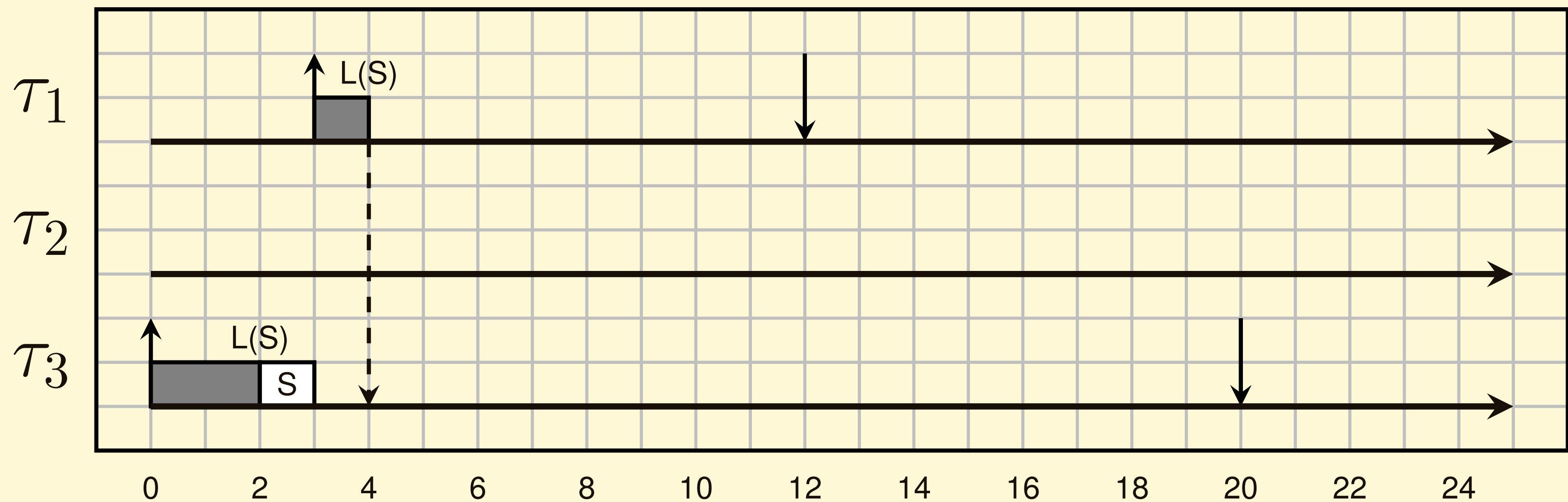
The Priority Inheritance protocol

- 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



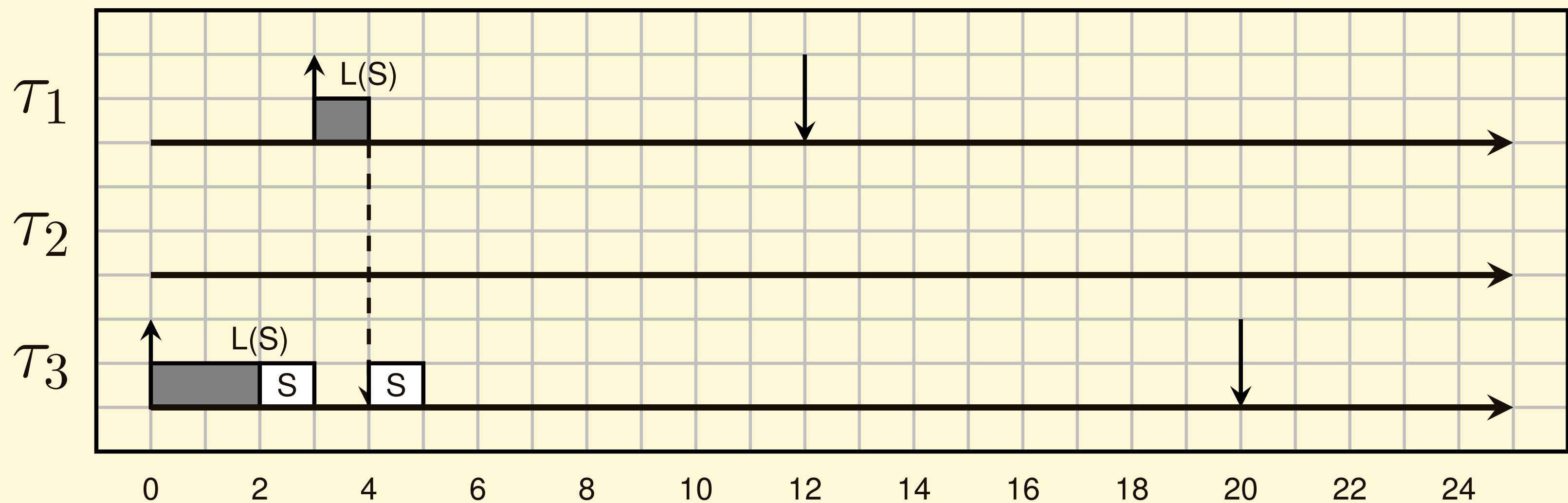
The Priority Inheritance protocol

- 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

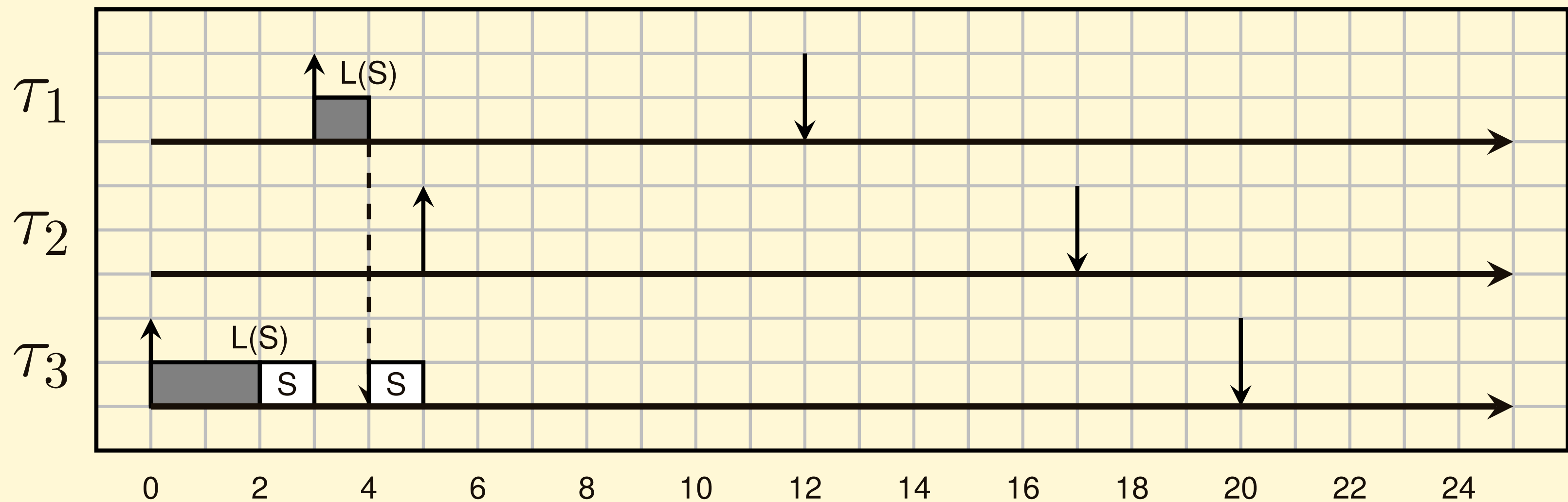
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



- Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

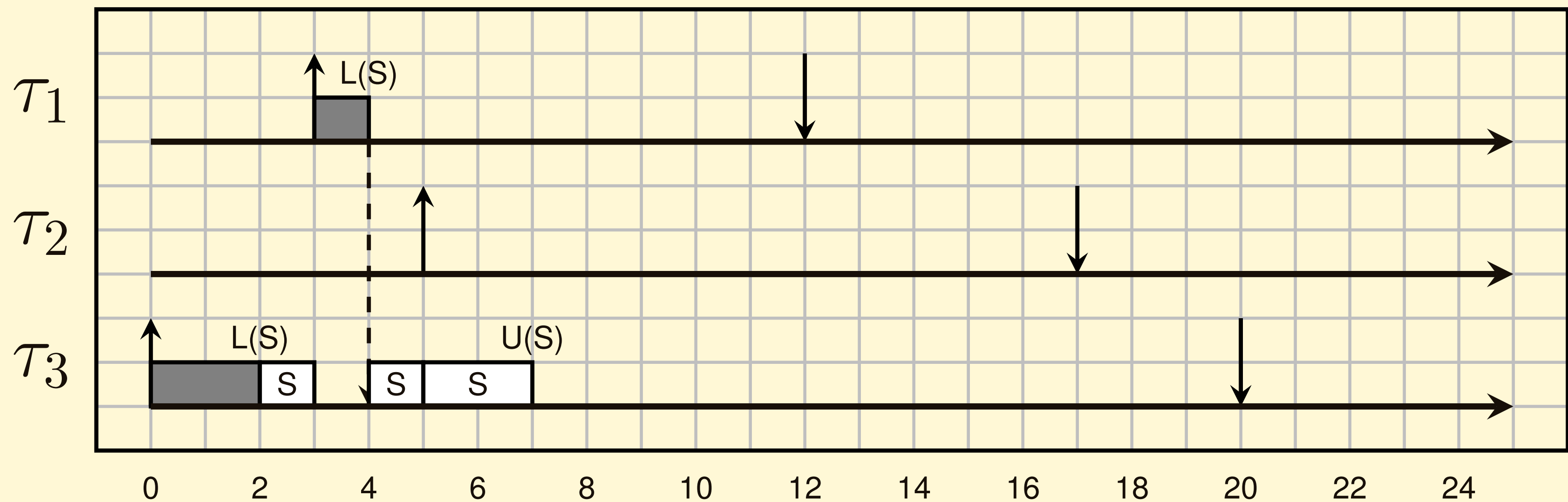
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

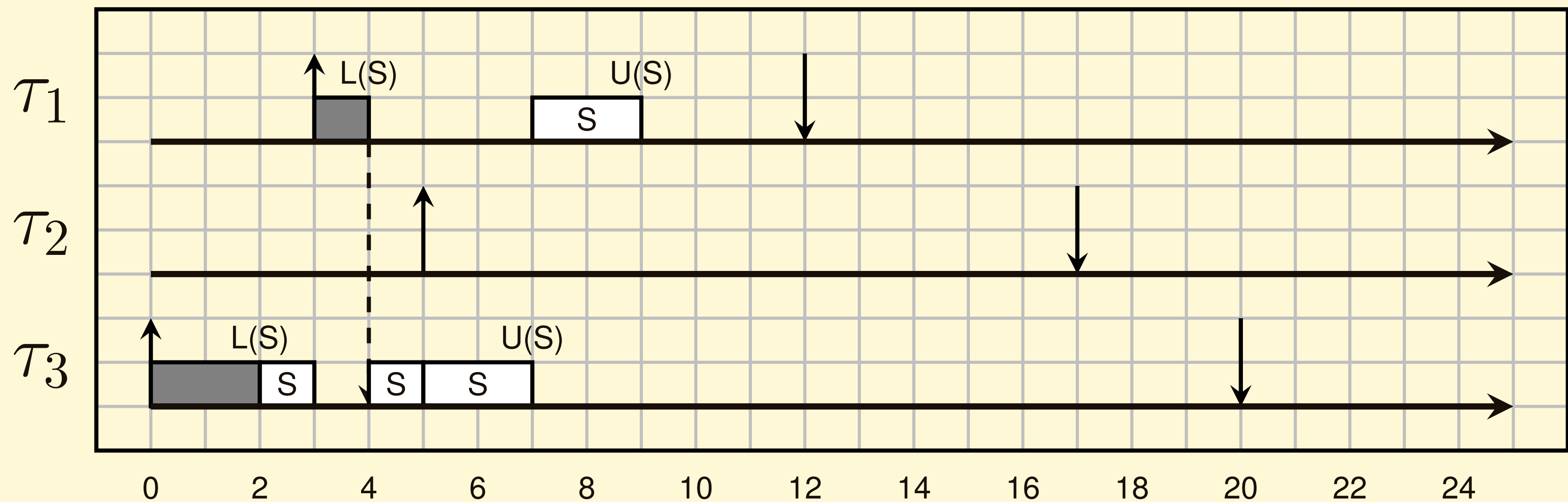
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

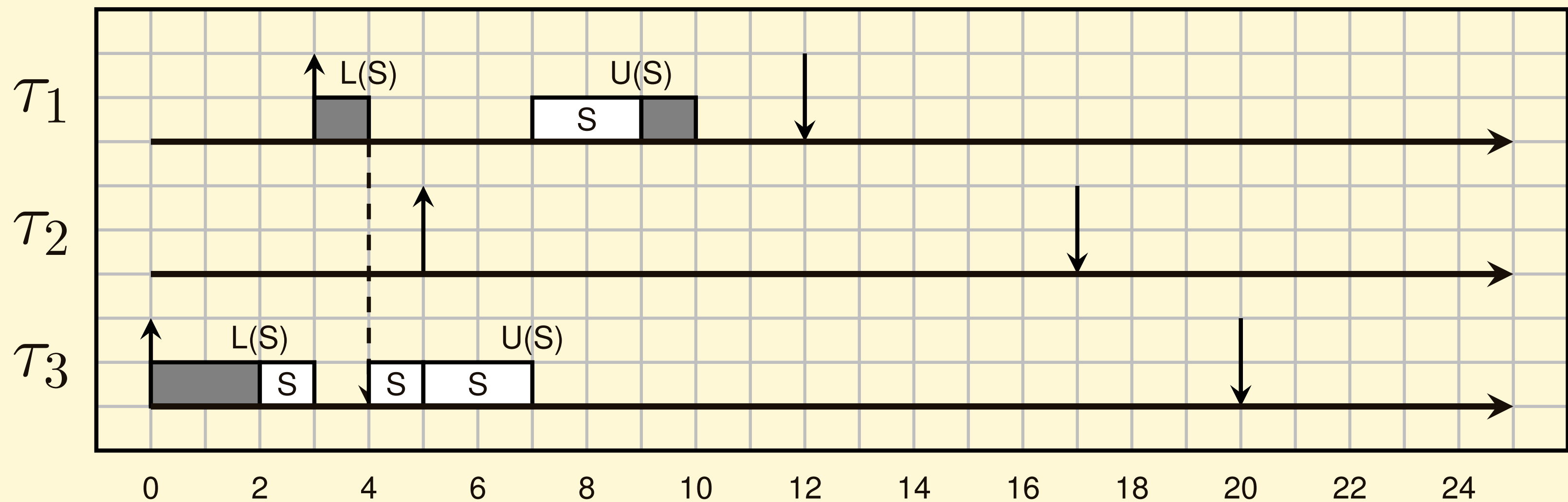
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

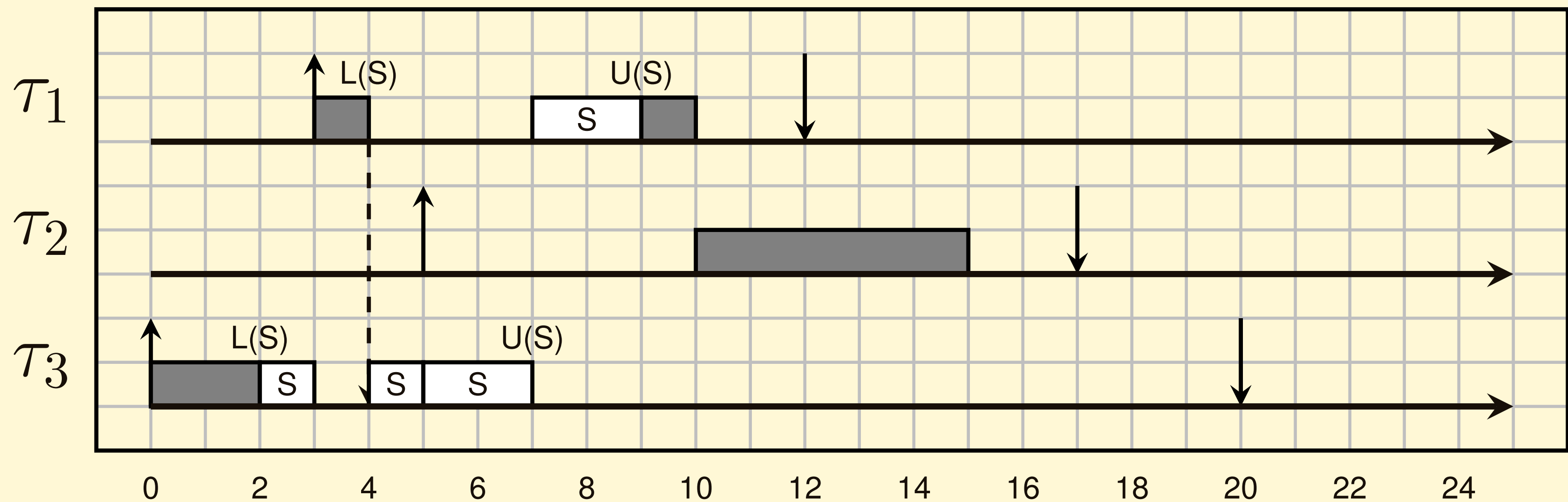
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



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

The Priority Inheritance protocol

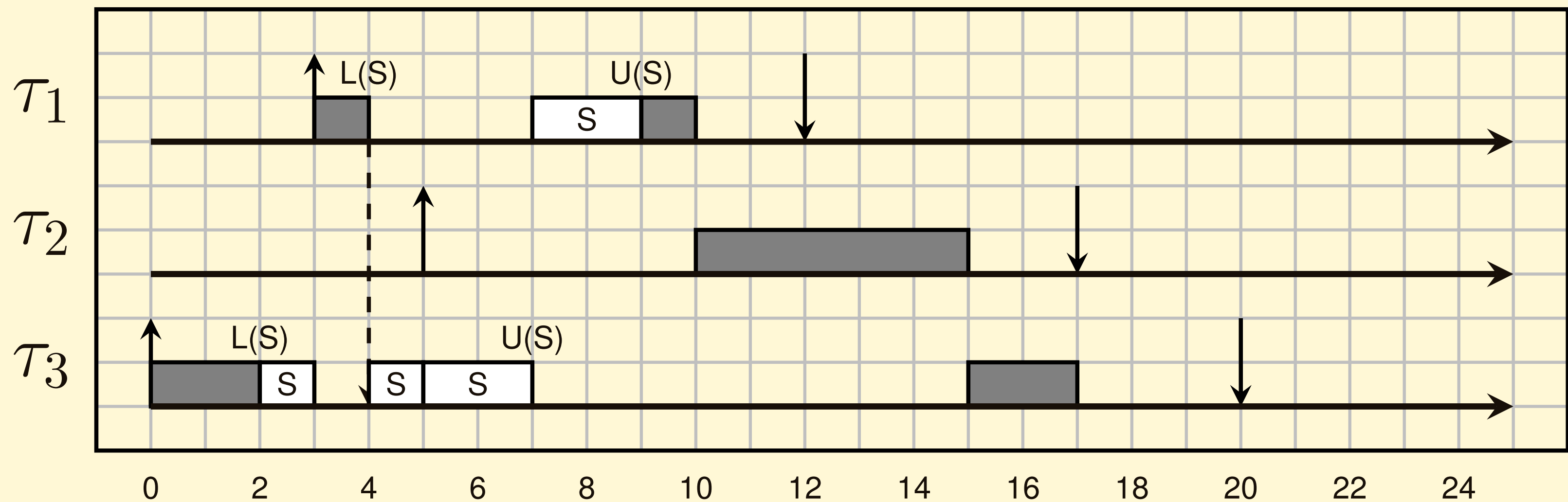
- Another possible solution to the priority inversion:
  - a low priority task τ_3 blocking an higher priority task τ_1 *inherits* its priority
  - $\rightarrow$ medium priority tasks cannot preempt τ_3



- 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

Real-Time Operating Systems

- Real-Time operating system (RTOS): OS providing support to Real-Time applications
- Real-Time application: the correctness depends not only on the output values, but also on the time when such values are produced
- Operating System:
  - Set of computer programs
  - Interface between applications and hardware
  - Control the execution of application programs
  - Manage the hardware and software resources

Different Visions of an OS

- An OS manages resources to provide services...
- ...hence, it can be seen as:
  - A Service Provider for user programs
    - Exports a programming interface...
  - A Resource Manager
    - Implements schedulers...

Operating System Services

- Services (Kernel Space):
  - Process Synchronisation, Inter-Process Communication (IPC)
  - Process / Thread Scheduling
  - I / O
  - Virtual Memory

RT-POSIX API?

Task Scheduling

- *Kernel*: core part of the OS, allowing multiple tasks to run on the same CPU
  - Task set $\mathcal{T}$ composed by N tasks running on M CPUs ($M < N$)
  - All tasks τ_i have the illusion to run in parallel
  - Temporal multiplexing between tasks
- Two core components:
  - *Scheduler*: decides which task to execute
  - *Dispatcher*: actually switches the CPU context (context switch)

Synchronization and IPC

- The kernel must also provide a mechanism for allowing tasks to communicate and synchronize
- Two possible programming paradigms:
  - Shared memory (threads)
  - Message passing (processes)

Programming Paradigms

- Shared memory (threads)
  - The kernel must provide mutexes + condition variables
  - Real-time resource sharing protocols (PI, HLP, NPP, ...) must be implemented
- Message passing (processes)
  - Interaction models: pipeline, client / server, ...
  - The kernel must provide some IPC mechanism: pipes, message queues, mailboxes, RPC, ...
  - Some real-time protocols can still be used

Real-Time Scheduling in Practice

An adequate scheduling of system resources removes the need for over-engineering the system, and is necessary for providing a predictable QoS

- Algorithm + Implementation = Scheduling
- RT theory provides us with good algorithms...
- ...But which are the prerequisites for correctly implementing them?

Theoretical and Actual Scheduling

- Scheduler, IPC subsystem, ... $\rightarrow$ must respect the theoretical model
  - Scheduling is simple: fixed priorities
  - IPC, HLP, or NPP are simple too...
  - But what about (for example) timers?
- Problem:
  - Is the scheduler able to select a high-priority task as soon as it is ready?
  - And the dispatcher?

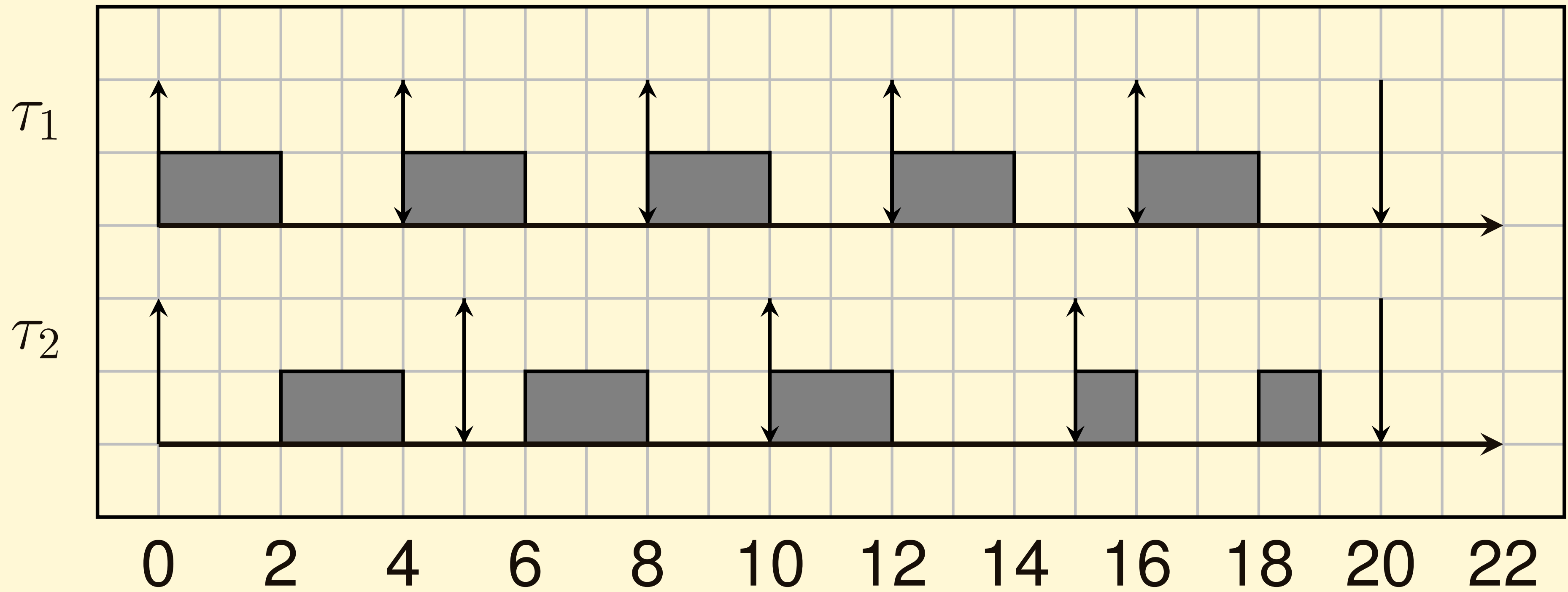
Periodic Task Example

- Consider a periodic task

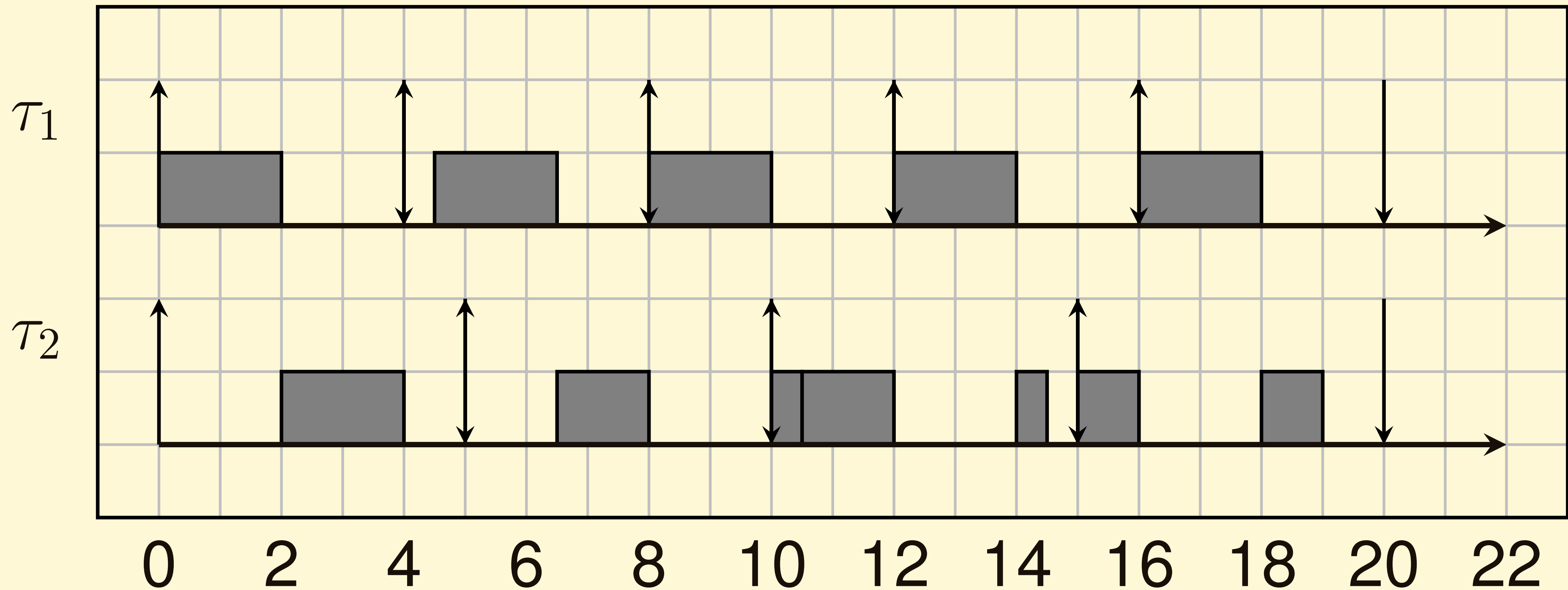
```
/* ... */
while(1) {
  /* Job body */
  clock_nanosleep(CLOCK_REALTIME,
                  TIMER_ABSTIME, &r, NULL);
  timespec_add_us(&r, period);
}
```

- The task expects to be executed at time `r` ($= r_0 + jT$)...
- ...But is sometimes delayed to $r_0 + jT + \delta$

Example - Theoretical Schedule



Example - Actual Schedule



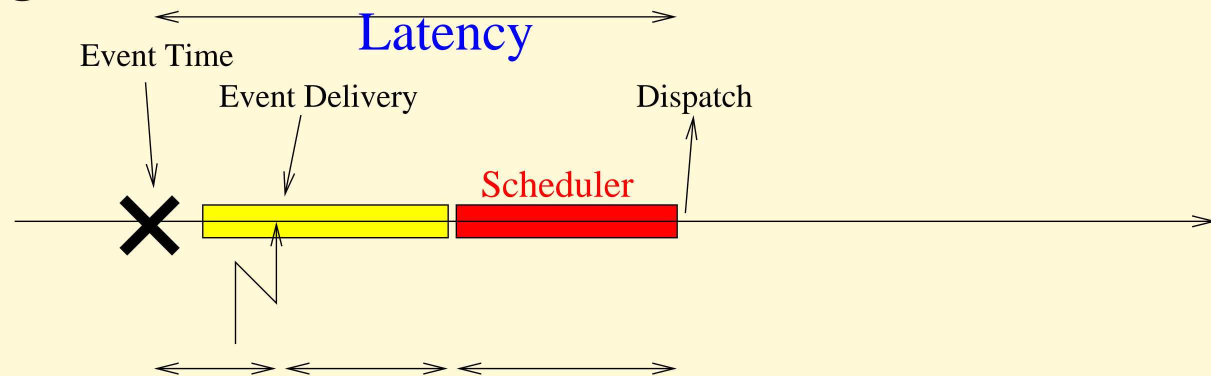
- What happens if the 2nd job of τ_1 arrives a little bit later???
  - The 2nd job of τ_2 misses a deadline!!!

Kernel Latency

- The delay δ in scheduling a task is due to *kernel latency*
- Kernel latency can be modelled as a blocking time
  - $\sum_{k=1}^{N} \frac{C_k}{T_k} \leq U_{lub} \rightarrow \forall i,\ 1 \leq i \leq n, \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i+\delta}{T_i} \leq U_{lub}$
  - $R_i = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h \rightarrow R_i = C_i + \delta + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$
  - $\exists 0 \leq t \leq D_i : W_i(0,t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h \leq t \rightarrow$
    $\exists 0 \leq t \leq D_i : W_i(0,t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h \leq t - \delta$

Kernel Latency

- Scheduler $\rightarrow$ triggered by internal (IPC, signal, ...) or external (IRQ) events
- Time between the triggering event and dispatch:
  - Event generation
  - Event delivery (interrupts may be disabled)
  - Scheduler activation (nonpreemptable sections)
  - Scheduling time



Theoretical Model vs Real Schedule

- In real world, high priority tasks often suffer from blocking times coming from the OS (more precisely, from the kernel)
  - Why?
  - How?
  - What can we do?
- To answer the previous questions, we need to recall how the hardware and the OS work...