Sistemi in tempo reale

Fixed Priority scheduling of Periodic tasks

Luigi Palopoli

Credits: Giuseppe Lipari and Marco Di Natale

Scuola Superiore Sant'Anna

Pisa -Italy

Task Model

Mathematical model of a task

- A task τ_i is a (infinite) sequence of jobs (or instances) $J_{i,k}$.
- Each job $J_{i,k} = (a_{i,k}, c_{i,k}, d_{i,k})$ is characterized by:
  - an activation time (or arrival time) $a_{i,k}$;

    $\rightarrow$ It is the time when the job is *activated* by an event of by a condition;
  - a computation time $c_{i,k}$;

    $\rightarrow$ It is the time it takes to complete the job;
  - an absolute deadline $d_{i,k}$

    $\rightarrow$ it is the absolute instant by which the job must complete.
  - the job finishes its execution at time $f_{i,j}$;

    $\rightarrow$ the response time of job $J_{i,j}$ is $\rho_{i,j} = f_{i,j} - a_{i,j}$;

    $\rightarrow$ for the job to be correct, it must be $f_{i,j} \leq d_{i,j}$.

Task model

A task can be:

- *periodic*: has a regular structure, consisting of an infinite cycle, in which it executes a computation and then suspends itself waiting for the next periodic activation. An example of pthread library code for a periodic task is the following:

```
void * PeriodicTask(void *arg)
{
  <initialization>;
  <start periodic timer, period = T>;
  while (cond) {
    <read sensors>;
    <update outputs>;
    <update state variables>;
    <wait next activation>;
  }
}
```

Model of a periodic task

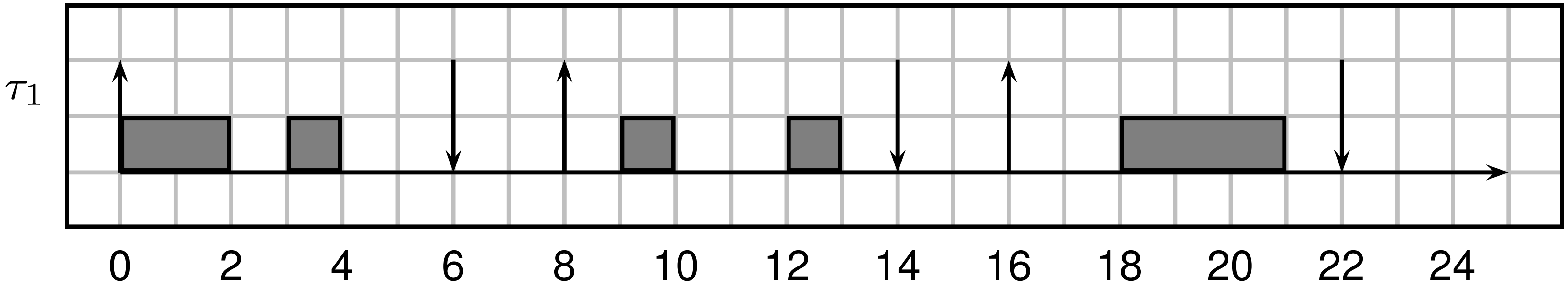
From a mathematical point of view, a periodic task $\tau_i = (C_i, D_i, T_i)$ consists of a (infinite) sequence of jobs $J_{i,k}$, $k = 0, 1, 2, \ldots$, with

$$
\begin{aligned}
& a_{i,0} &=& \; 0 \\
\forall k > 0 \quad & a_{i,k} &=& \; a_{i,k-1} + T_i \\
\forall k \geq 0 \quad & d_{i,k} &=& \; a_{i,k} + D_i \\
& C_i &=& \; \max\{k \geq 0 | c_{i,k}\}
\end{aligned}
$$

- T_i is the task's period;
- D_i is the task's relative deadline;
- C_i is the task's worst-case execution time (WCET);
- R_i is the worst-case response time: $R_i = max_j\{\rho_{i,j}\}$;
  - for the task to be schedulable, it must be $R_i \leq D_i$.

Graphical representation

In this course, the tasks will be graphically represented with a GANNT chart. In the following example, we graphically show periodic task $\tau_1 = (3, 6, 8)$.



Notice that, while job $J_{i,0}$ and $J_{i,3}$ execute for 3 units of time (WCET), job $J_{i,2}$ executes for only 2 units of time.

Hyperperiod

- A task set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ is periodic if it consists of periodic tasks only.
- The *hypeperiod* of a periodic task set is the least common multiple (lcm) of the task's periods;

$$H(\mathcal{T}) = \text{lcm}_{\tau_i \in \mathcal{T}}(T_i)$$

- The patterns of arrival repeats every hypeperiod. In practice, if two tasks arrive at the same time t, they will arrive at the same time $t + kH$, for every integer number $k \geq 0$;
- Sometimes, the hyperperiod is defined also for task sets consisting or periodic and sporadic tasks. The meaning is slightly different.

Offsets

- A periodic task can have an *initial offset* ϕ_i
- The offset is the arrival time of the first instance of a periodic task;
- Hence:

$$\begin{aligned} a_{i,0} &= \phi_i \\ a_{i,k} &= \phi_i + kT_j \end{aligned}$$

- In some case, offsets are set to a value different from o to avoid all tasks starting at the same time.

Timeline Scheduling for periodic tasks

Timeline scheduling

- very popular in military and avionics systems
- also called cyclic executive or cyclic scheduling
- examples
  - air traffic control
  - Space Shuttle
  - Boeing 777

The idea

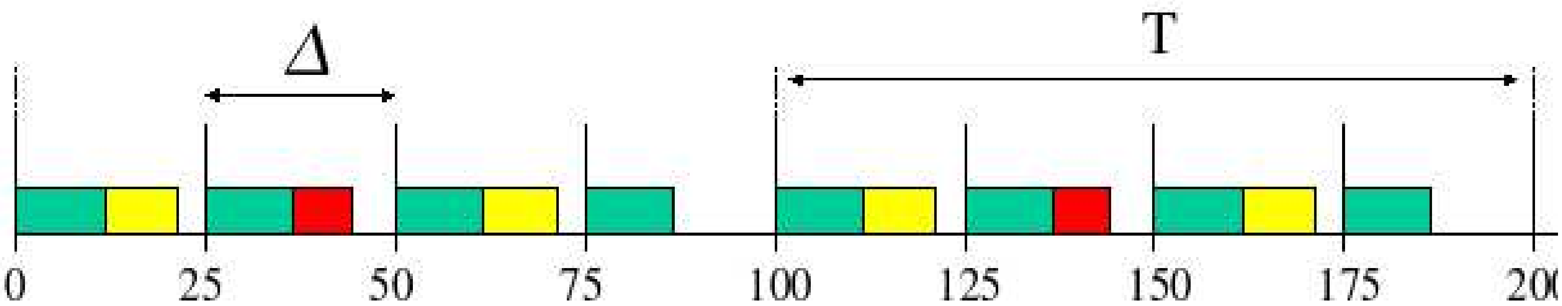
- the time axis is divided time slots
- slots are statically allocated to the tasks
- a timer activates execution (allocation of a slot)

Example

task	f	T
A	40 Hz	25 ms
B	20 Hz	50 ms
C	10 Hz	100 ms

Δ = gcd (minor cycle)

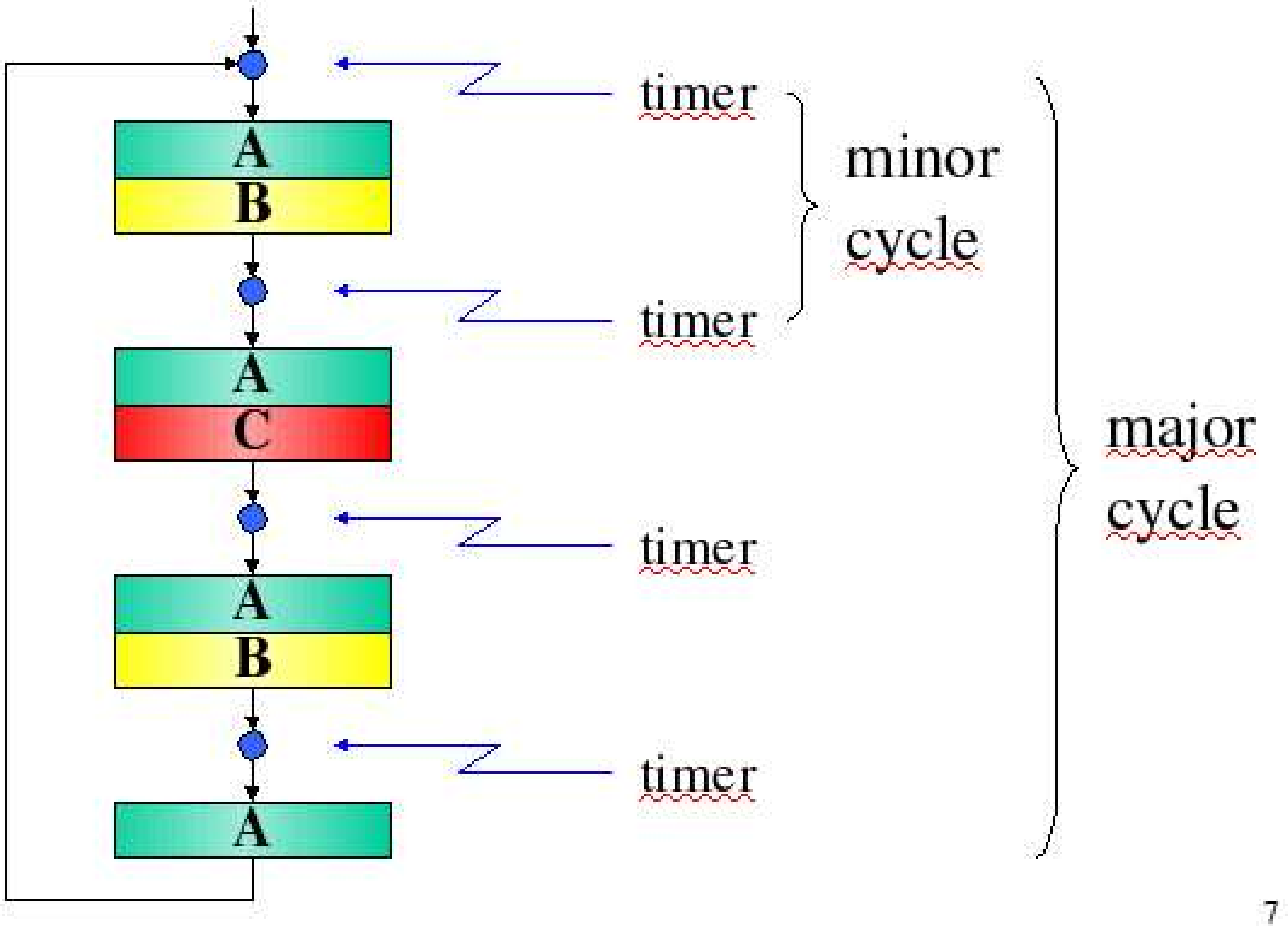
T = lcm (major cycle)



guarantee:
$$\begin{cases} C_A + C_B \leq \Delta \\ C_A + C_C \leq \Delta \end{cases}$$

6

Implementation



Advantages

- simple implementation (no real-time operating system is required)
- common address space
- run-time overhead
- jitter control

Drawbacks

- it is not robust during overloads
- it is difficult to expand the schedule
- it is not easy to handle aperiodic activities
- all process periods must be a multiple of the minor cycle time
- it is difficult to incorporate processes with long periods
- any process with a sizable computation time will need to be split into a fixed number of fixed sized procedures

Overload Management

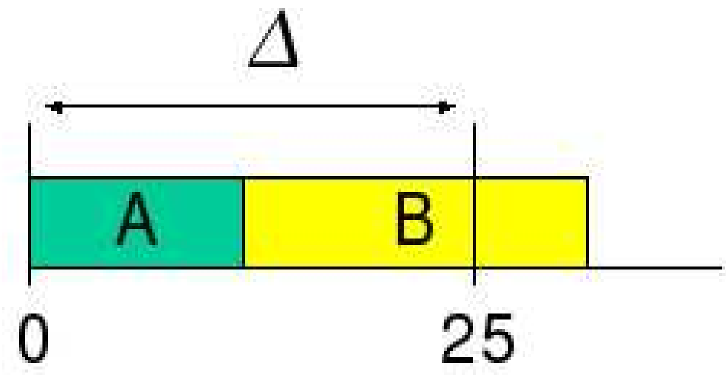
what do we do during task overruns?

- let the task continue
  - we can have a domino effect on all the other tasks (timeline break)
- abort the task
  - the system can remain in inconsistent states

Extensibility

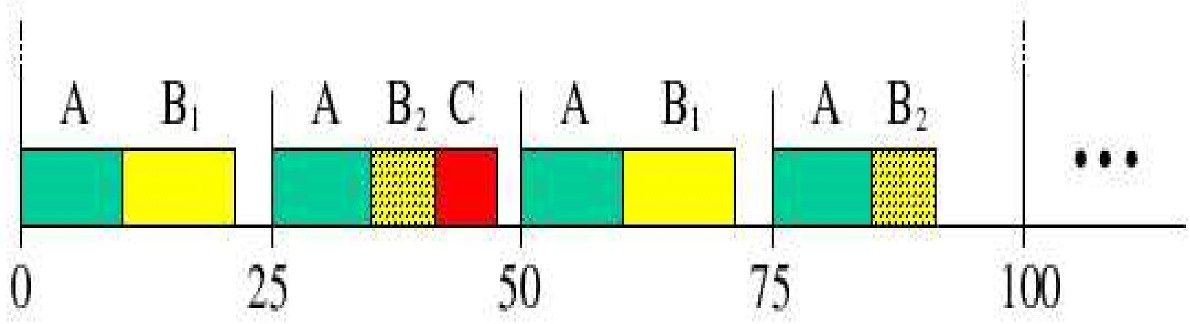
- if one or more tasks need to be upgraded, we may have to re-design the whole schedule again

Example: B is updated but $C_A + C_B > \Delta$



Extensibility

- We have to split B into two subtasks (B_1, B_2) and recompute the schedule.



guarantee: $\begin{cases} C_A + C_{B1} \le \Delta \\ C_A + C_{B2} + C_C \le \Delta \end{cases}$

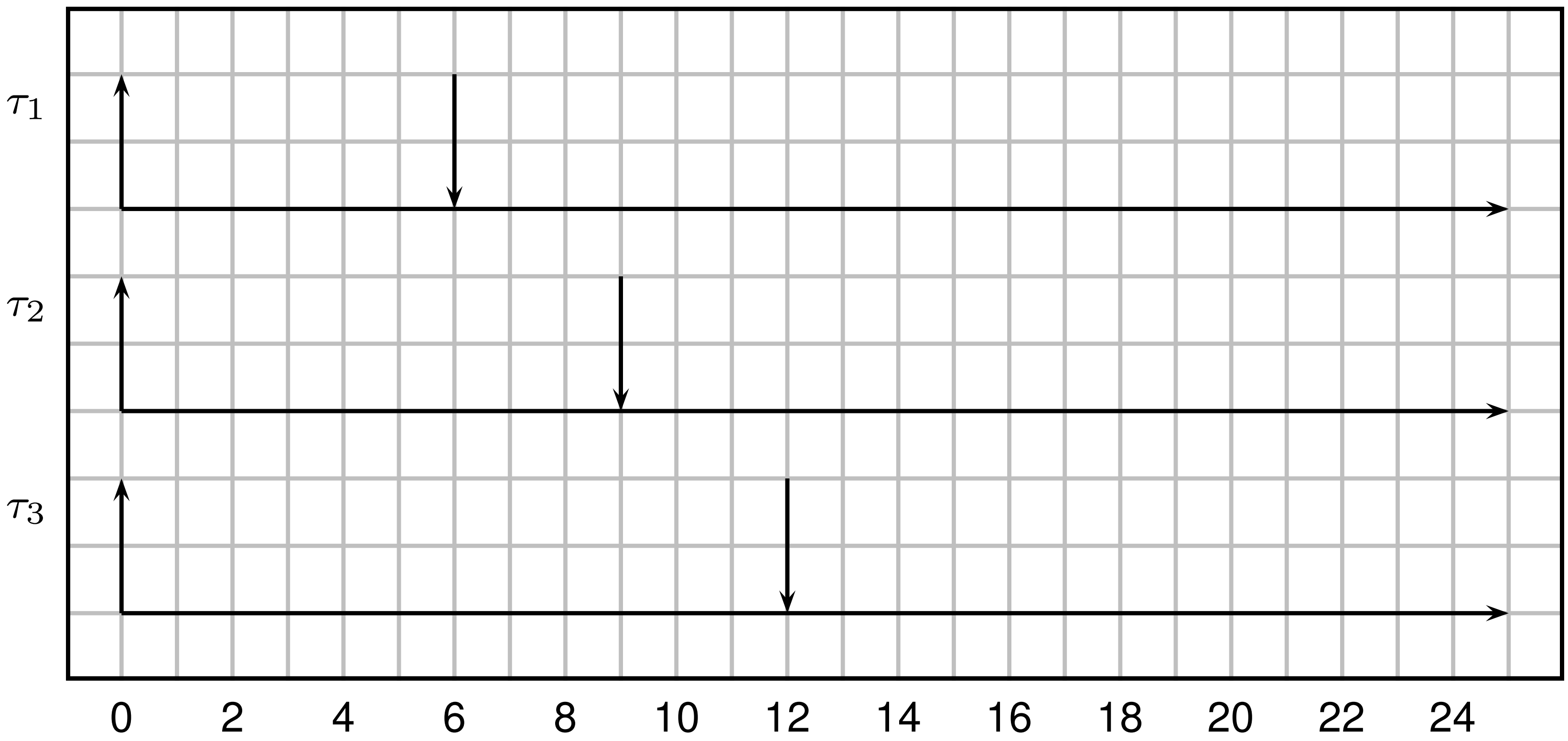
Fixed priority Scheduling

The fixed priority scheduling algorithm

- very simple 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, and constant execution time equal to the period.

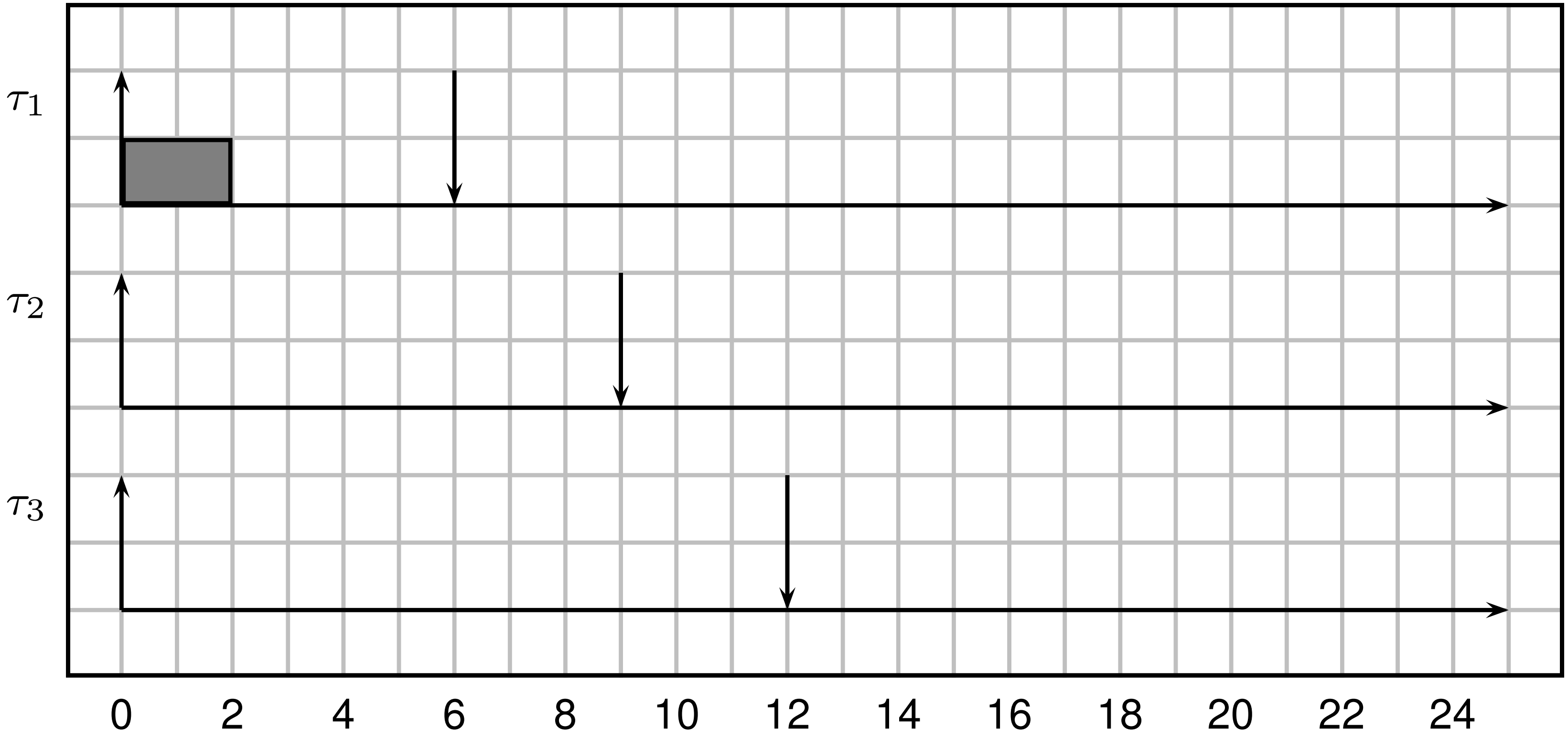
Example of schedule

- 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).



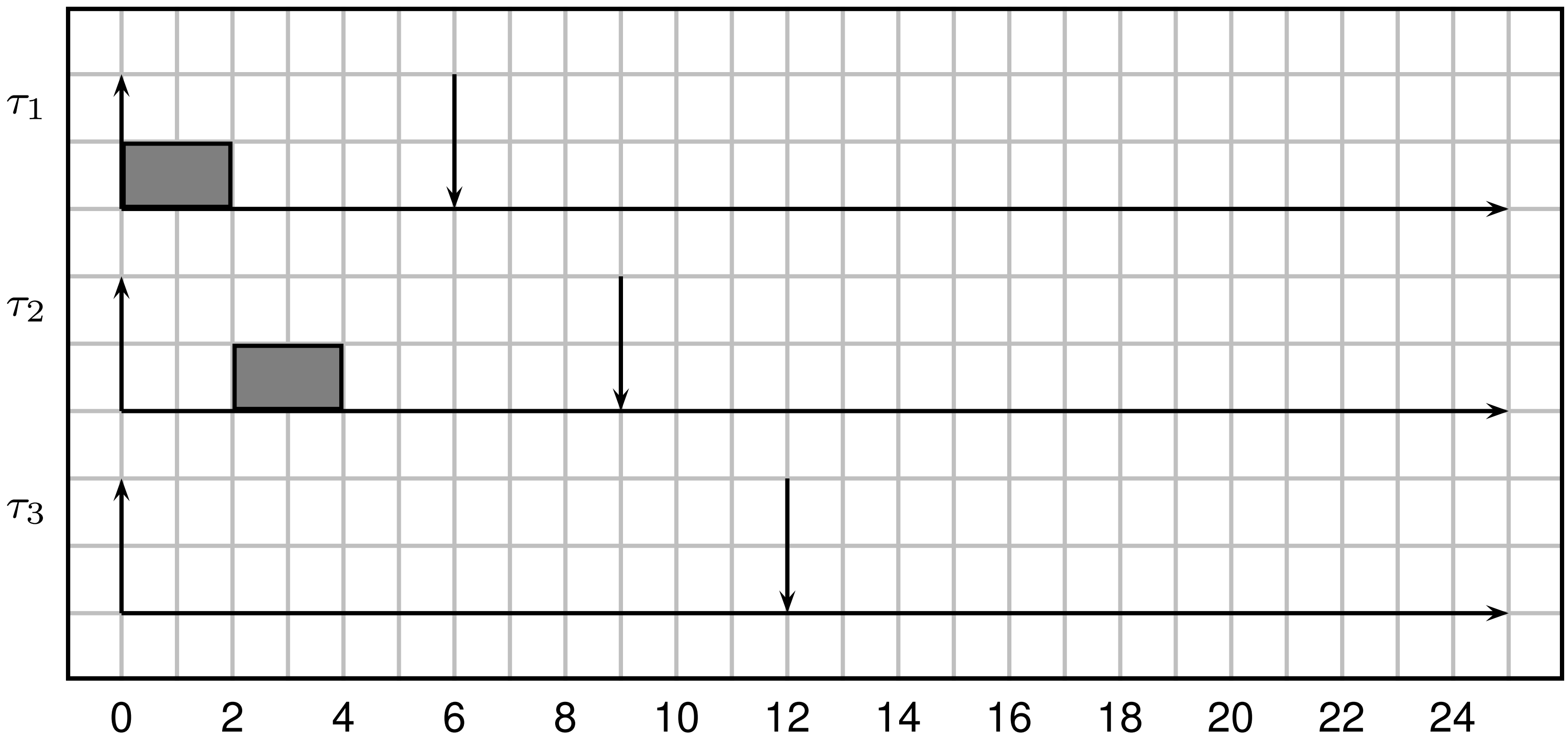
Example of schedule

- 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).



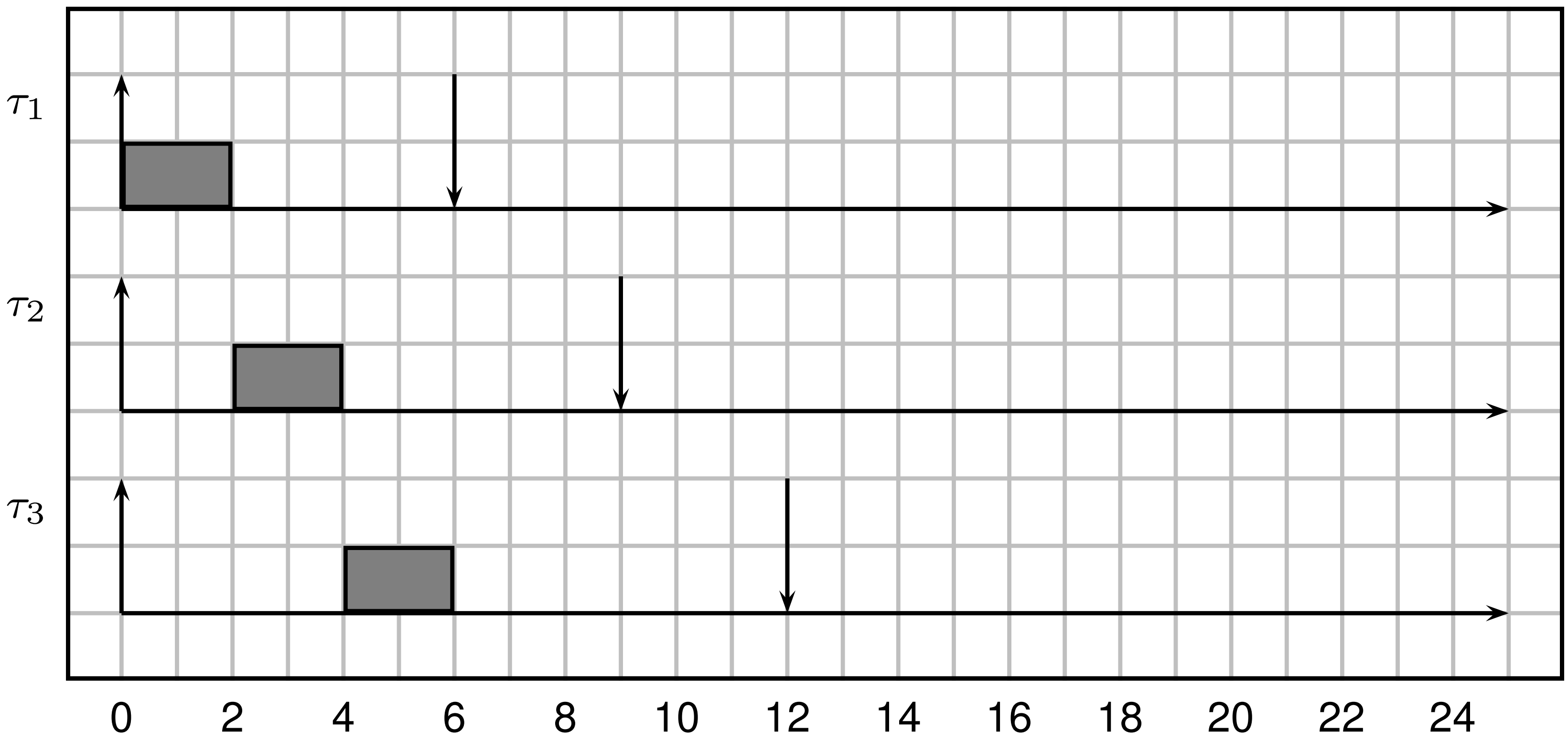
Example of schedule

- 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).



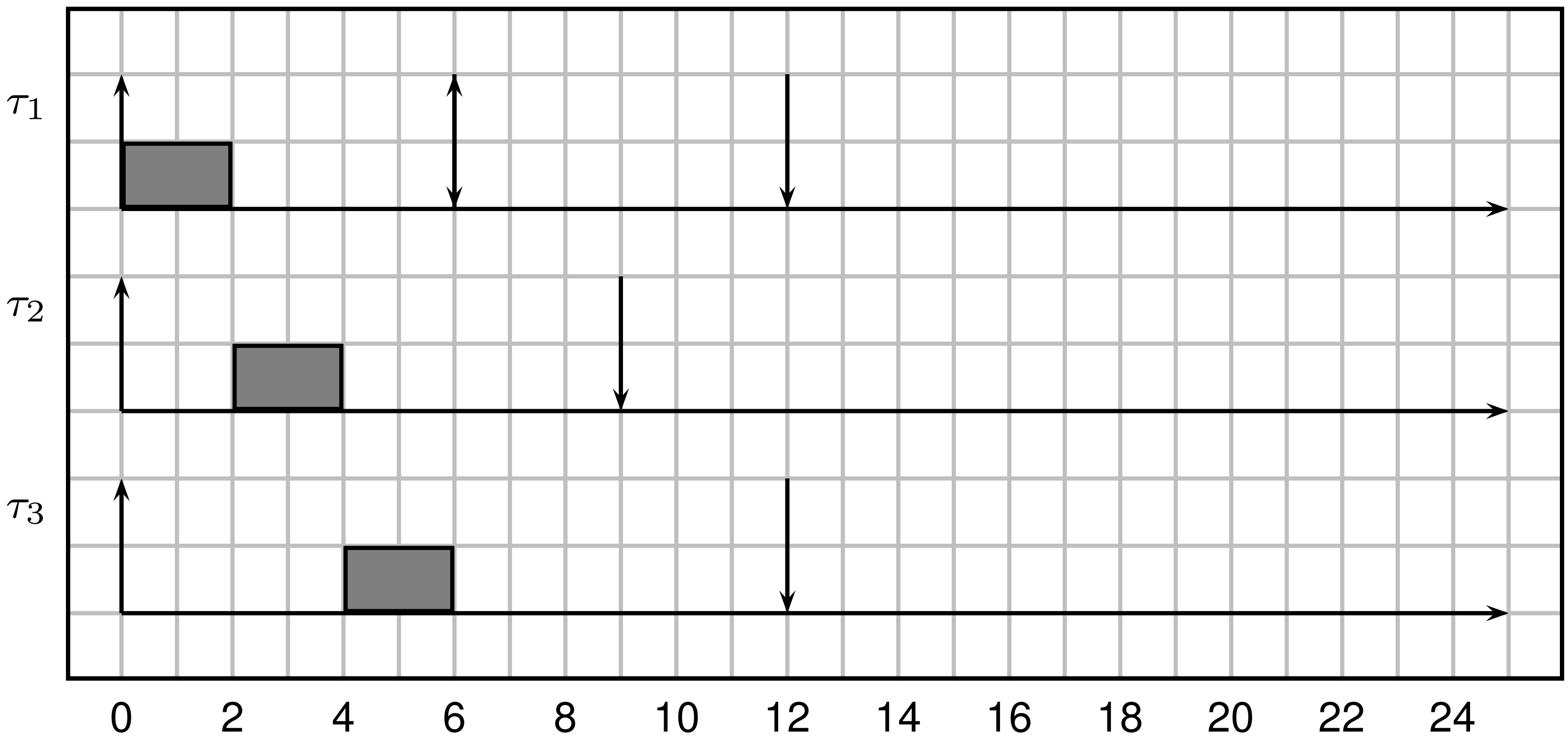
Example of schedule

- 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).



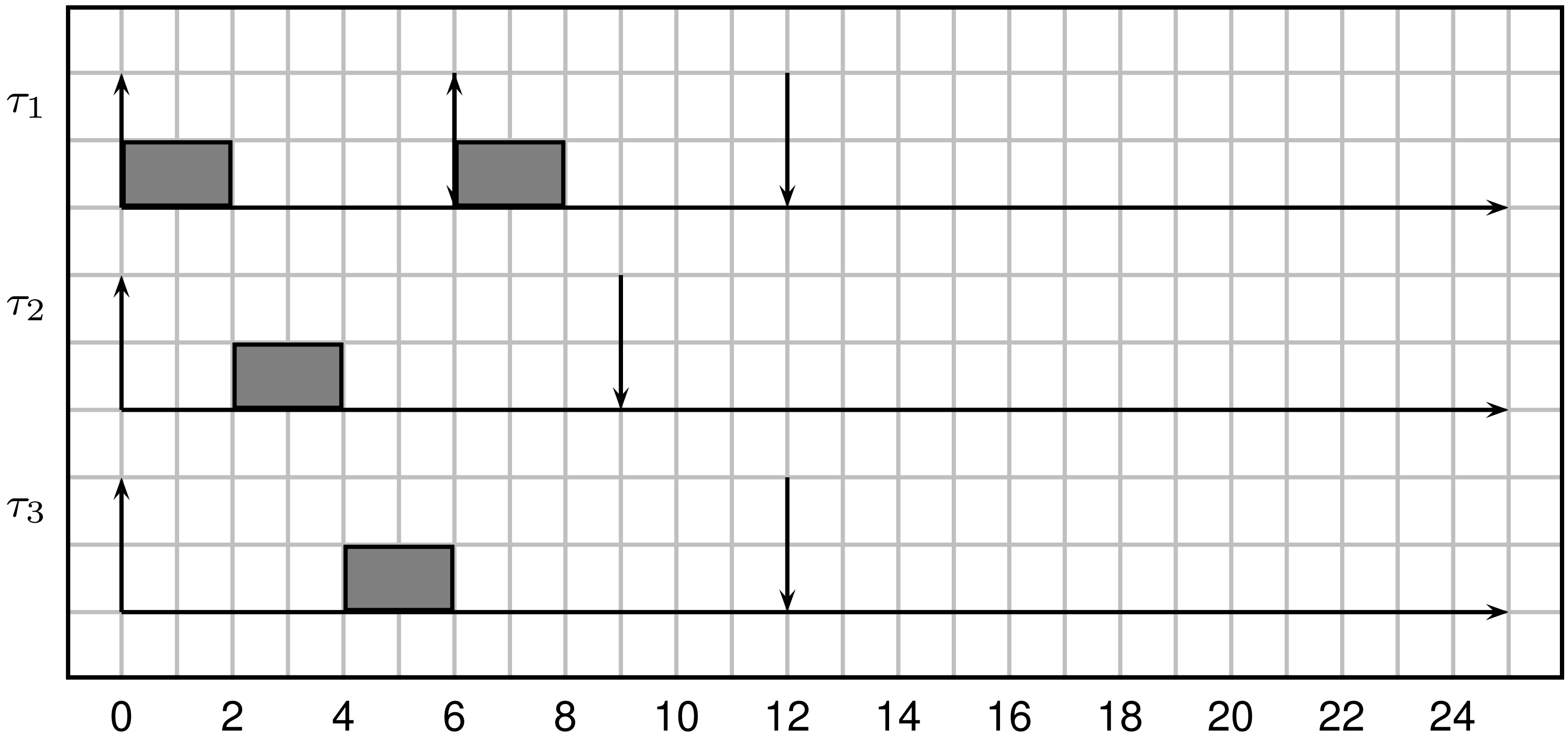
Example of schedule

- 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).



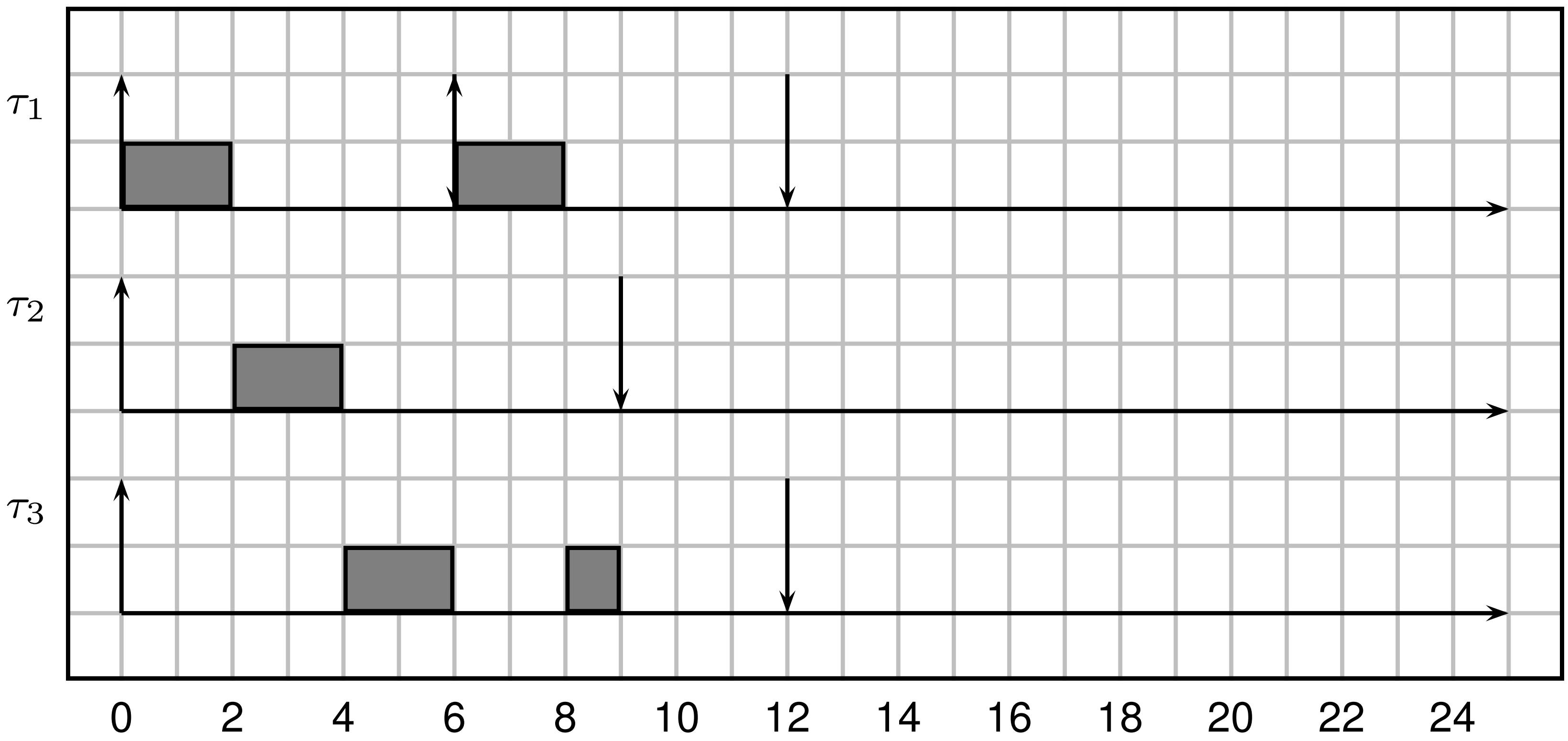
Example of schedule

- 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).



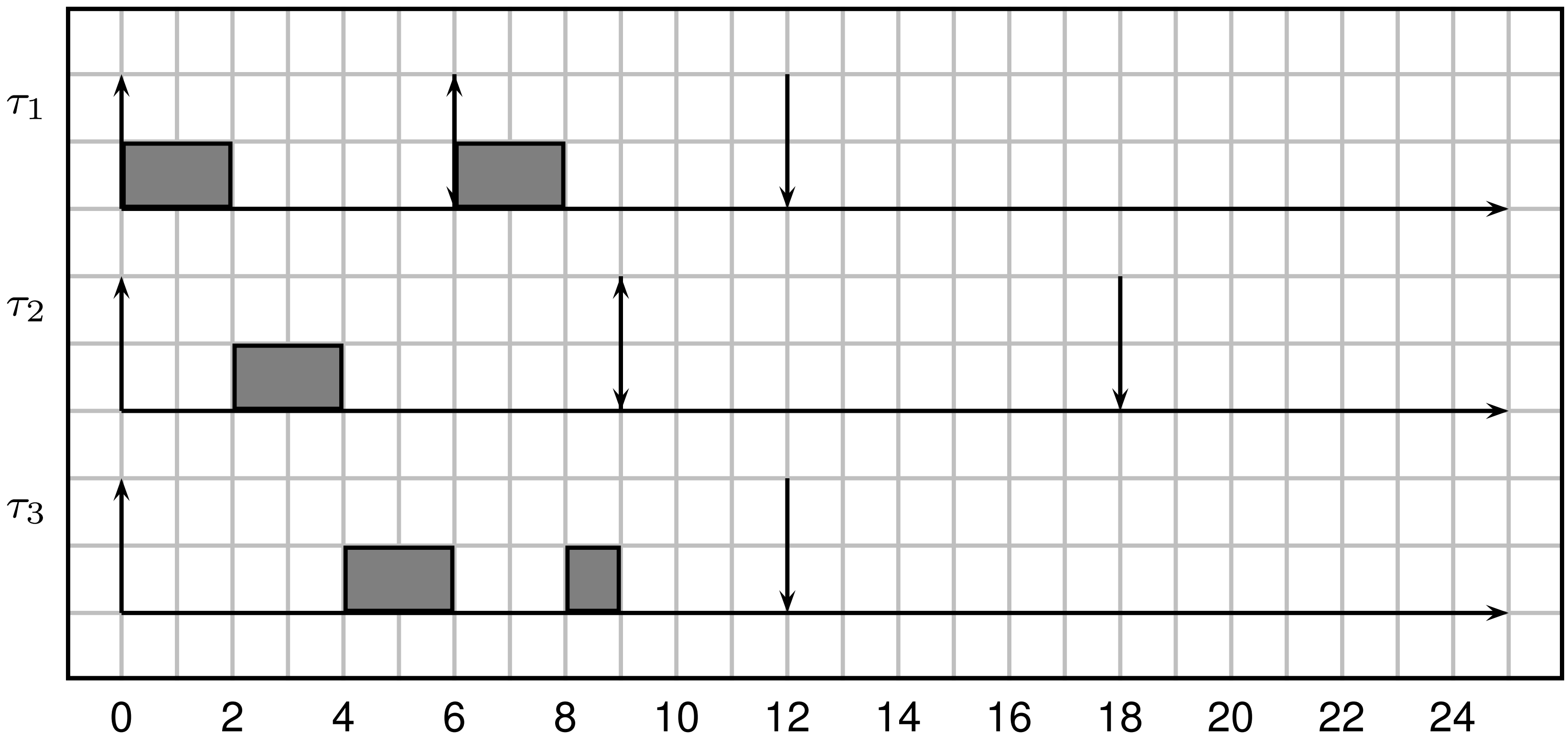
Example of schedule

- 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).



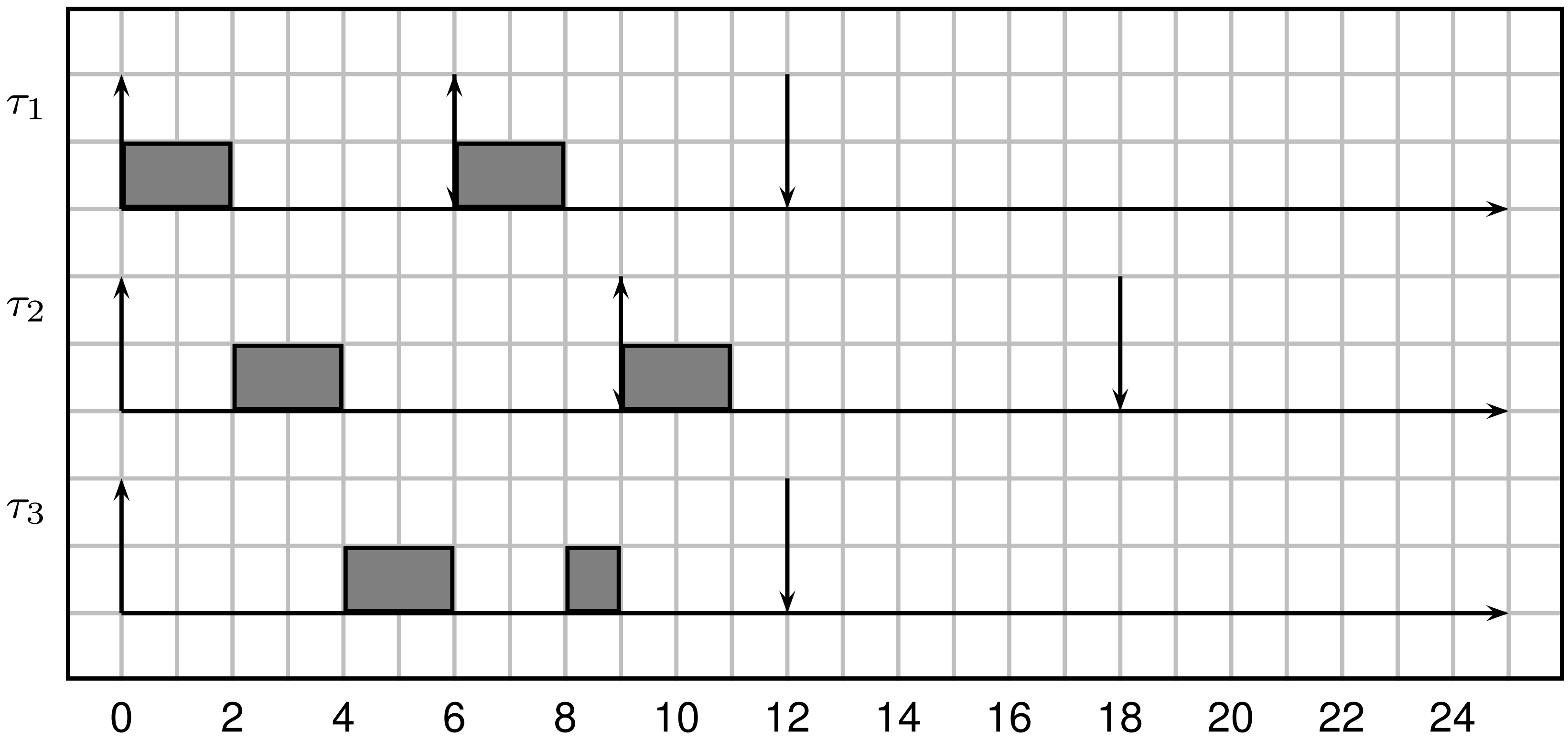
Example of schedule

- 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).



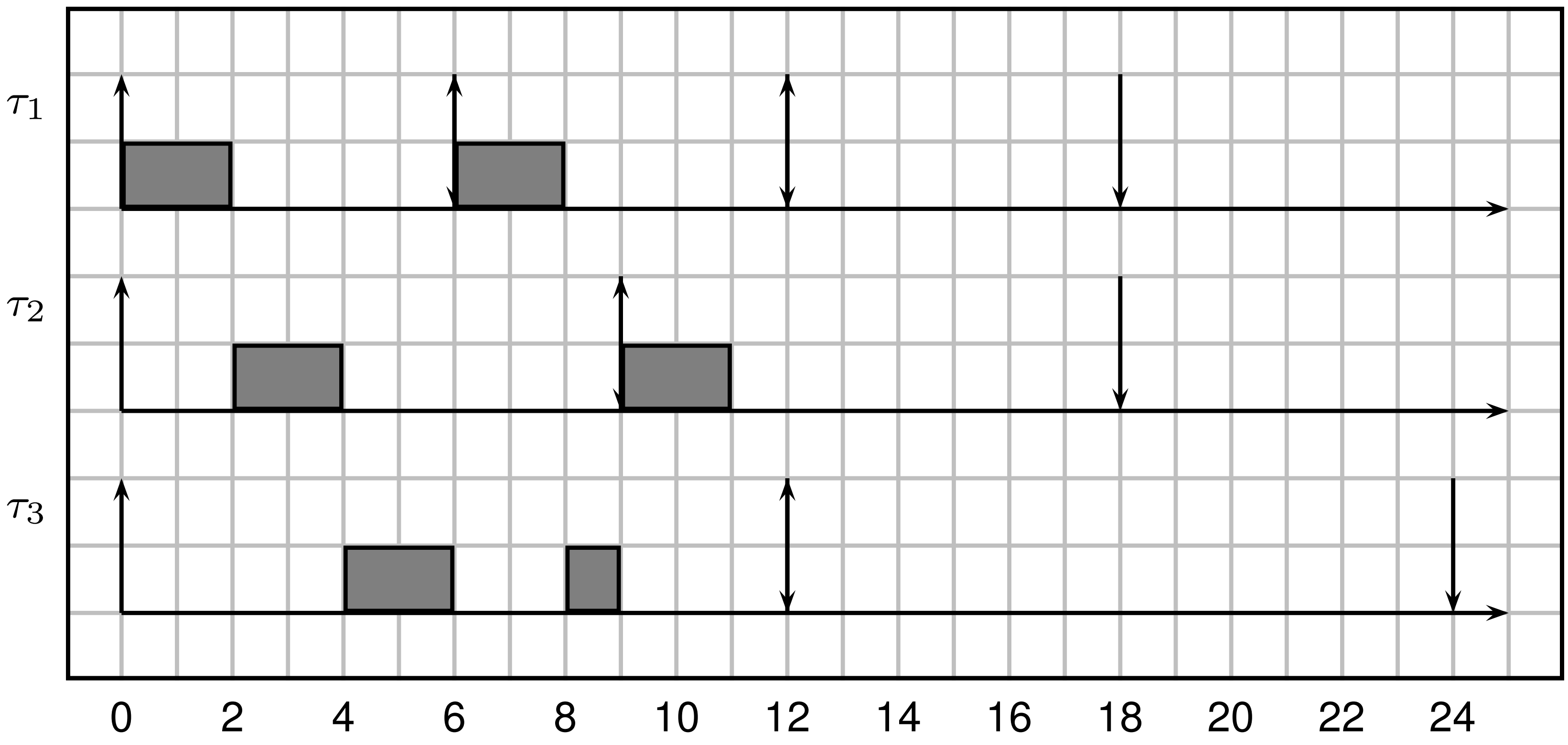
Example of schedule

- 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).



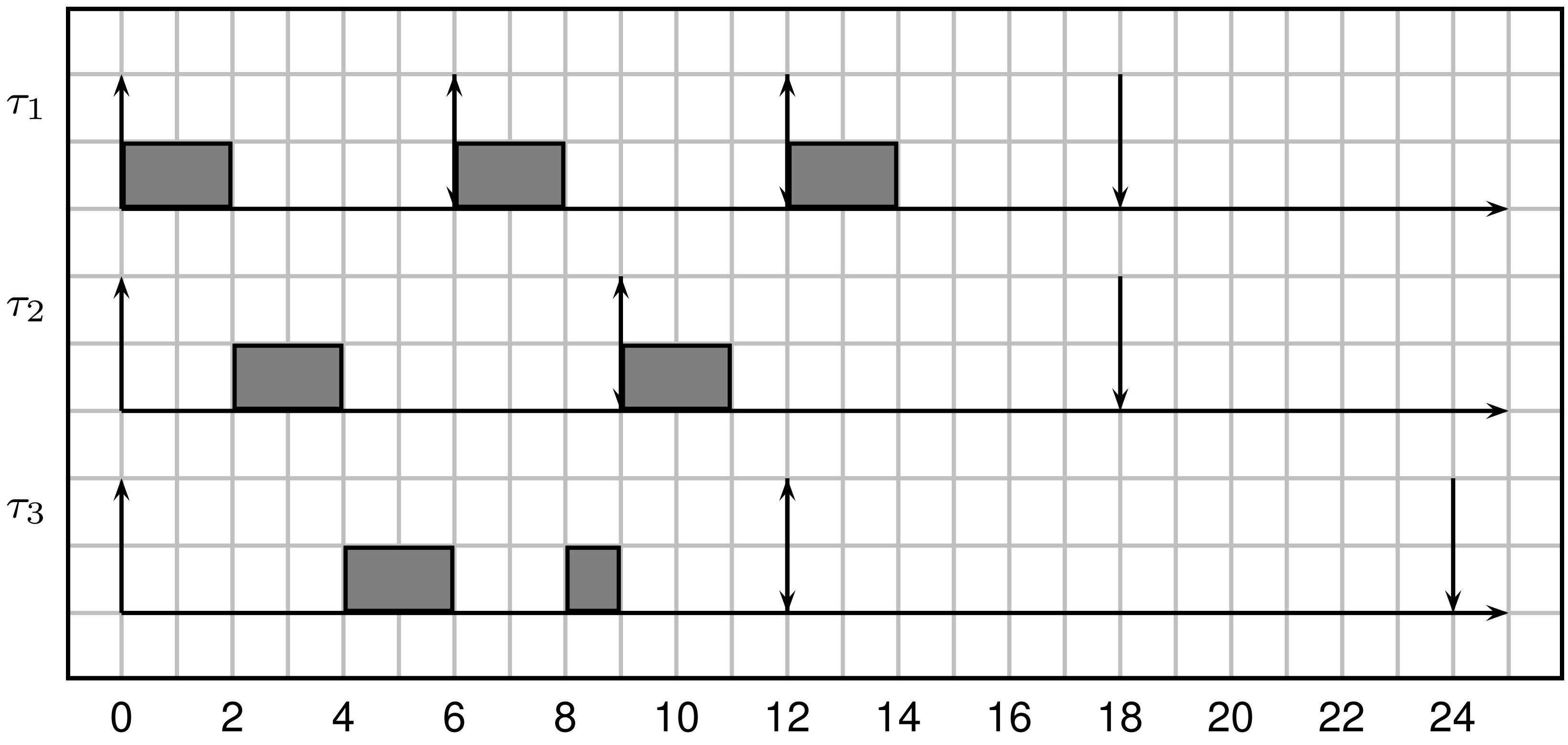
Example of schedule

- 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).



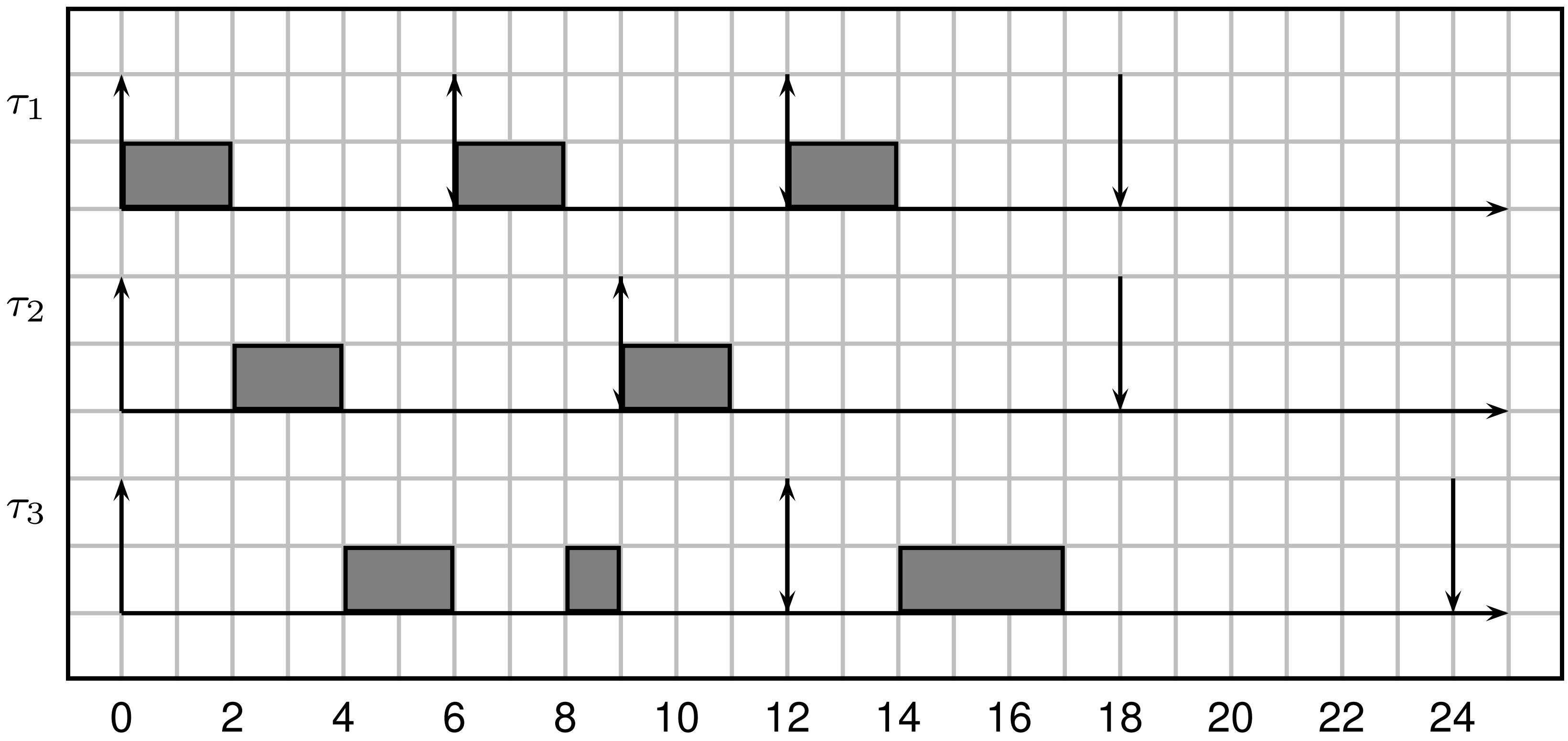
Example of schedule

- 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).



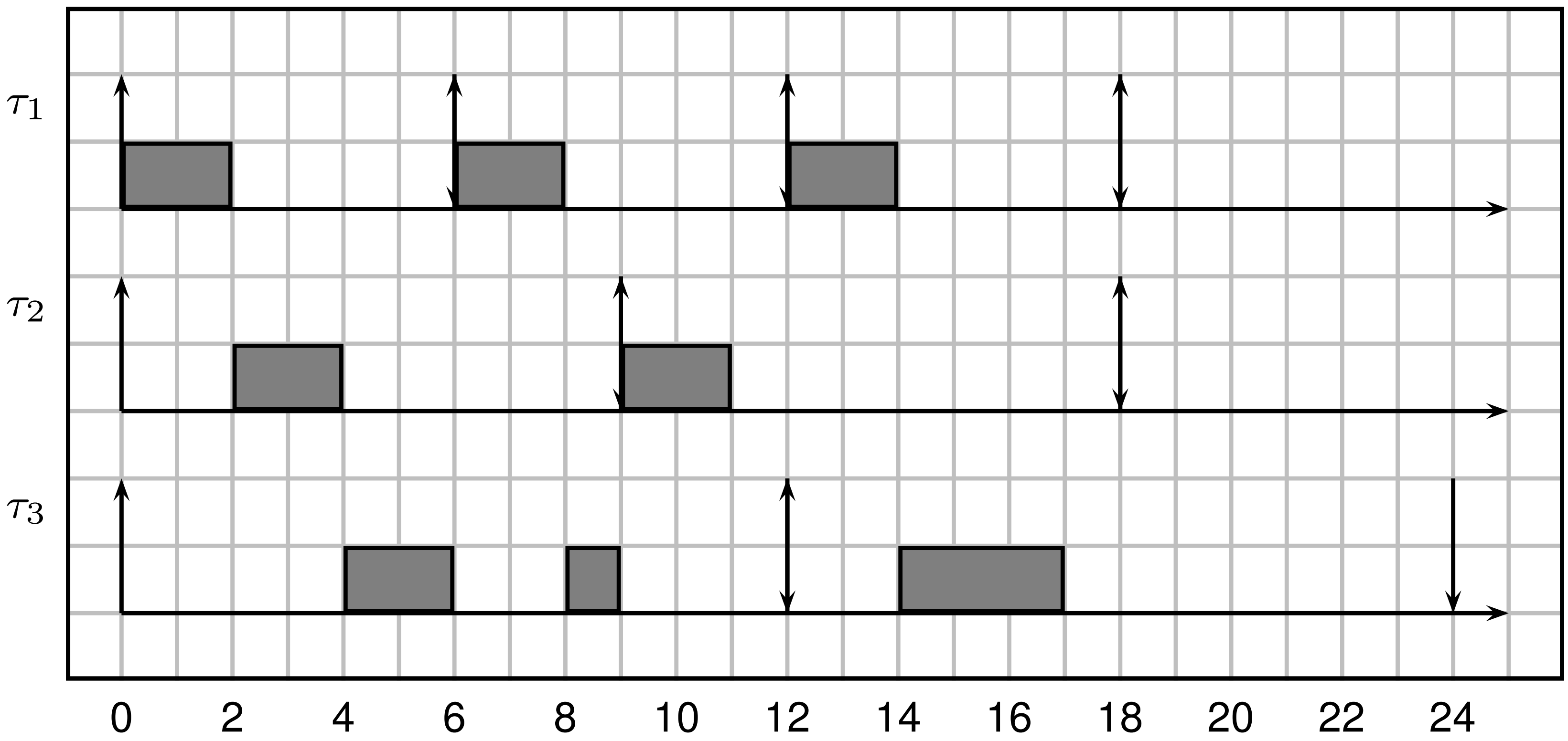
Example of schedule

- 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).



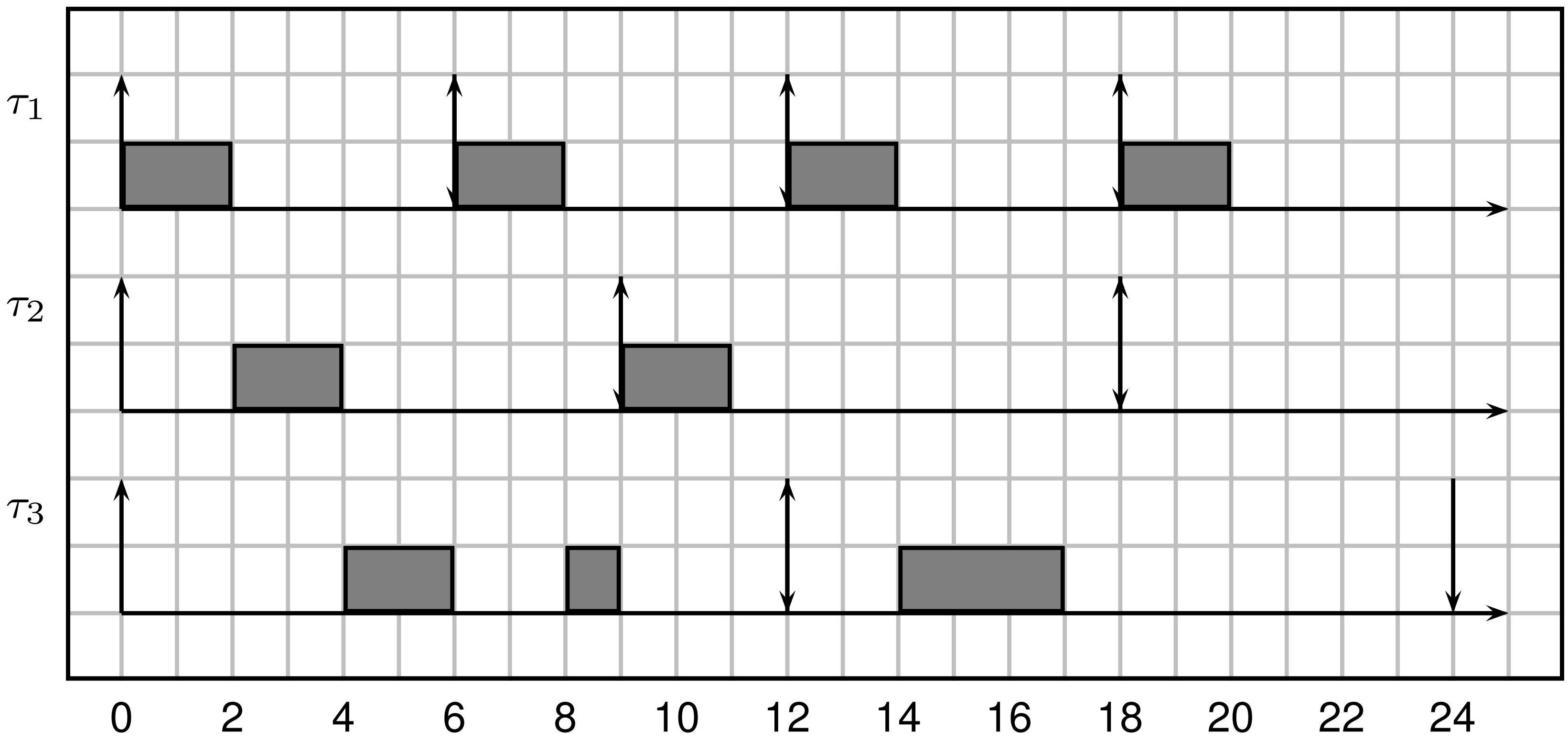
Example of schedule

- 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).



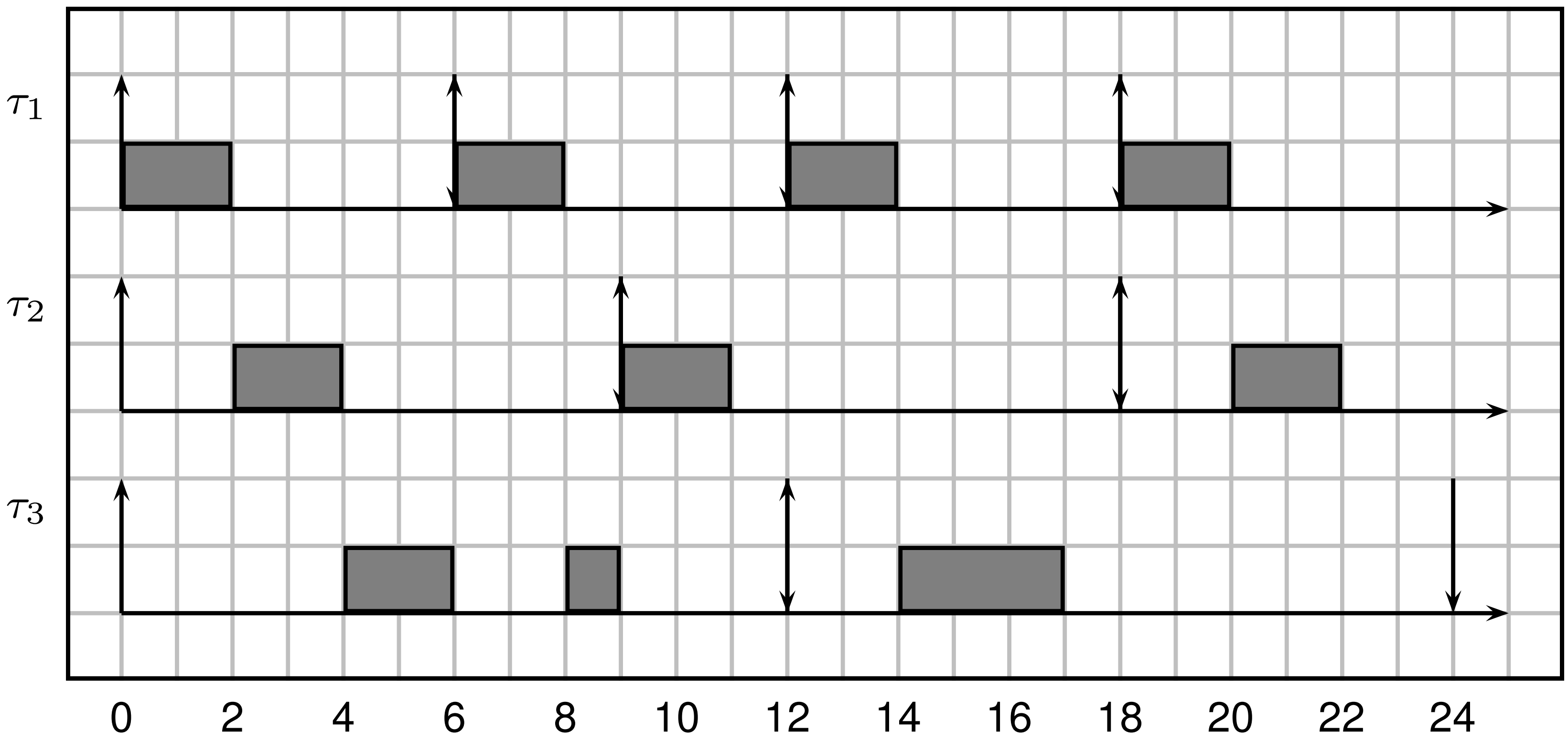
Example of schedule

- 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).



Note

- 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.

Priority assignment

Priority assignment

- Given a task set, how to assign priorities?
- There are two possible objectives:
  - Schedulability (i.e. find the priority assignment that makes all tasks schedulable)
  - Response time (i.e. find the priority assignment that minimize the response time of a subset of tasks).
- By now we consider the first objective only
- An *optimal* priority assignment *Opt* is such that:
  - If the task set is schedulable with another priority assignment, then it is schedulable with priority assignment *Opt*.
  - If the task set is not schedulable with *Opt*, then it is not schedulable by any other assignment.

Optimal priority assignment

- Given a periodic task set with all tasks having deadline equal to the period ($\forall i, \;\; D_i = T_i$), and with all offsets equal to 0 ($\forall i, \;\; \phi_i = 0$):
  - The best assignment is the *Rate Monotonic* assignment
  - Tasks with shorter period have higher priority
- Given a periodic task set with deadline different from periods, and with all offsets equal to 0 ($\forall i, \quad \phi_i = 0$):
  - The best assignement is the *Deadline Monotonic* assignment
  - Tasks with shorter relative deadline have higher priority
- For sporadic tasks, the same rules are valid as for periodic tasks with offsets equal to 0.

Presence of offsets

- If instead we consider periodic tasks with offsets, then *there is no optimal priority assignment*
  - In other words,
    - → if a task set $\mathcal{T}_1$ is schedulable by priority O_1 and not schedulable by priority assignment O_2,
    - → it may exist another task set $\mathcal{T}_2$ that is schedulable by O_2 and not schedulable by O_1.
  - For example, $\mathcal{T}_2$ may be obtained from $\mathcal{T}_1$ simply changing the offsets!

Scheduling analysis

Analysis

- Given a task set, how can we guarantee if it is schedulable of not?
- The first possibility is to *simulate* the system to check that no deadline is missed;
- The execution time of every job is set equal to the WCET of the corresponding task;
  - In case of periodic task with no offsets, it is sufficient to simulate the schedule until the *hyperperiod* $(H = lcm_i(T_i))$.
  - In case of offsets, it is sufficient to simulate until $2H + \phi_{\max}$.
  - If tasks periods are prime numbers the hyperperiod can be very large!

Utilization analysis

- In many cases it is useful to have a very simple test to see if the task set is schedulable.
- A sufficient test is based on the *Utilization bound*:
  - The *utilization least upper bound* for scheduling algorithm $\mathcal{A}$ is the smallest possible utilization U_{lub} such that, for any task set $\mathcal{T}$, if the task set's utilization U is not greater than U_{lub} ($U \leq U_{lub}$), then the task set is schedulable by algorithm $\mathcal{A}$.

Utilisation

- Each task uses the processor for a fraction of time

$$U_i = \frac{C_i}{T_i}$$

- the total processor utilisation is

$$U_p = \sum_i \frac{C_i}{T_i}$$

- this is a measure of the processor's load

Necessary condition

- if $U_p > 1$ the task set is not surely schedulable
- however, if $U_p < 1$ the task may not be schedulable

Utilization bound for RM

- We consider n periodic (or sporadic) tasks with relative deadline equal to periods.
- Priorities are assigned with Rate Monotonic;
- $U_{lub} = n(2^{1/n} - 1)$
  - U_{lub} is a decreasing function of n;
  - For large n: $U_{lub} \approx 0.69$

Schedulability test

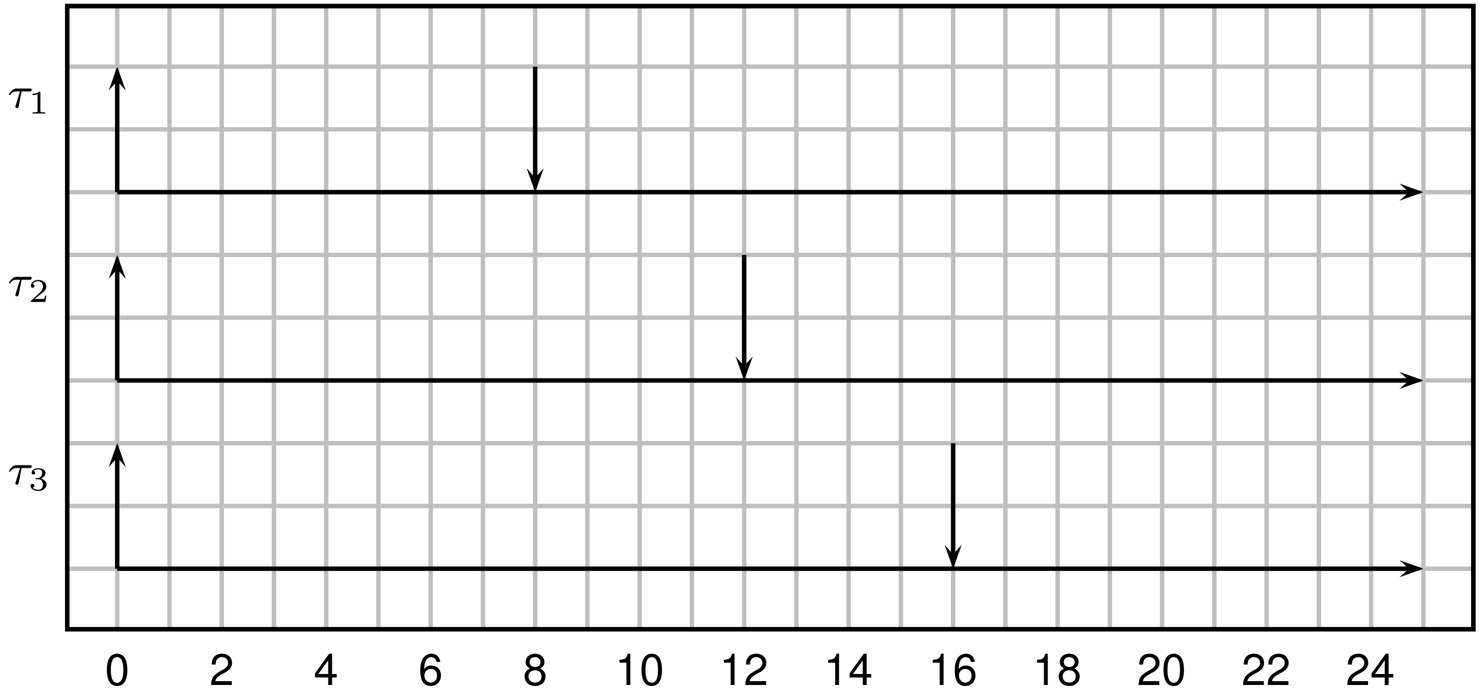
- Therefore the schedulability test consist in:
  - Compute $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$;
  - if $U \leq U_{lub}$, the task set is schedulable;
  - if $U > 1$ the task set is not schedulable;
  - if $U_{lub} < U \leq 1$, the task set may or may not be schedulable;

Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16);$

$$U = 0.75 < U_{lub} = 0.77$$

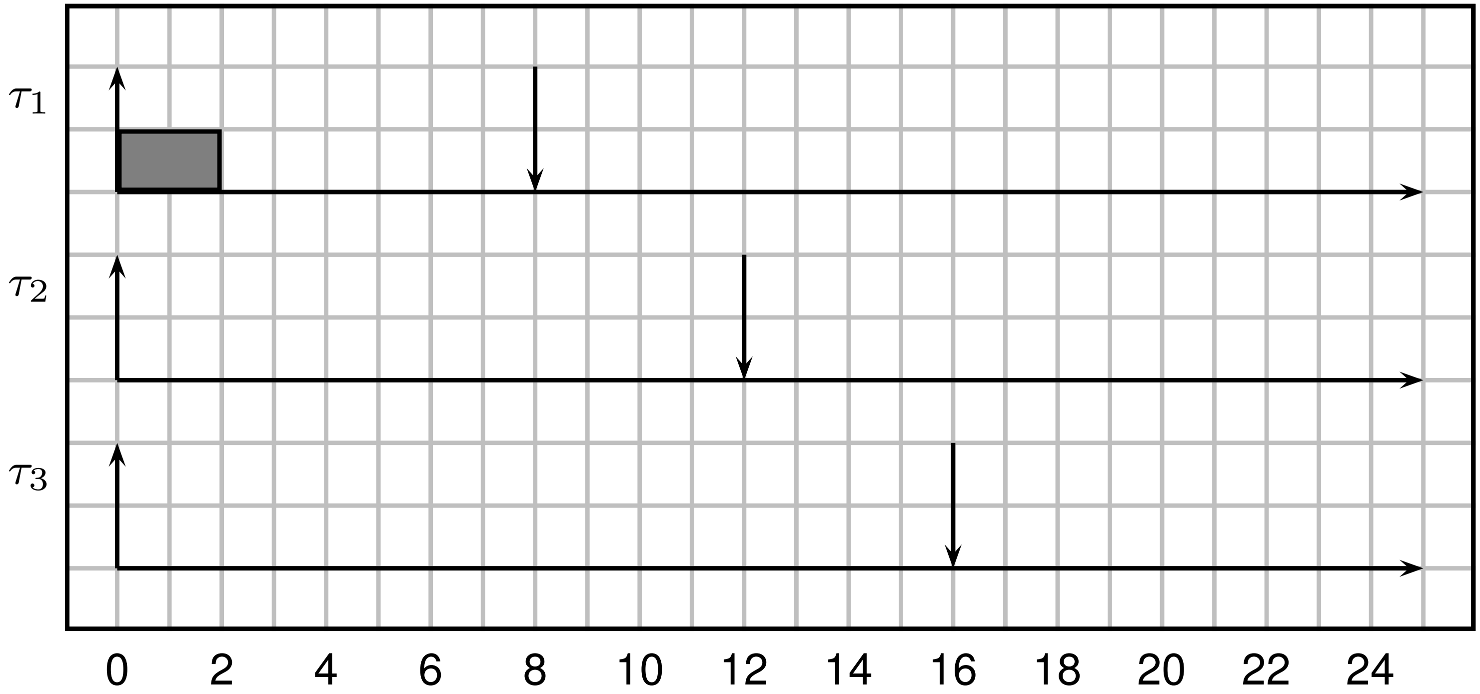


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16);$

$$U = 0.75 < U_{lub} = 0.77$$

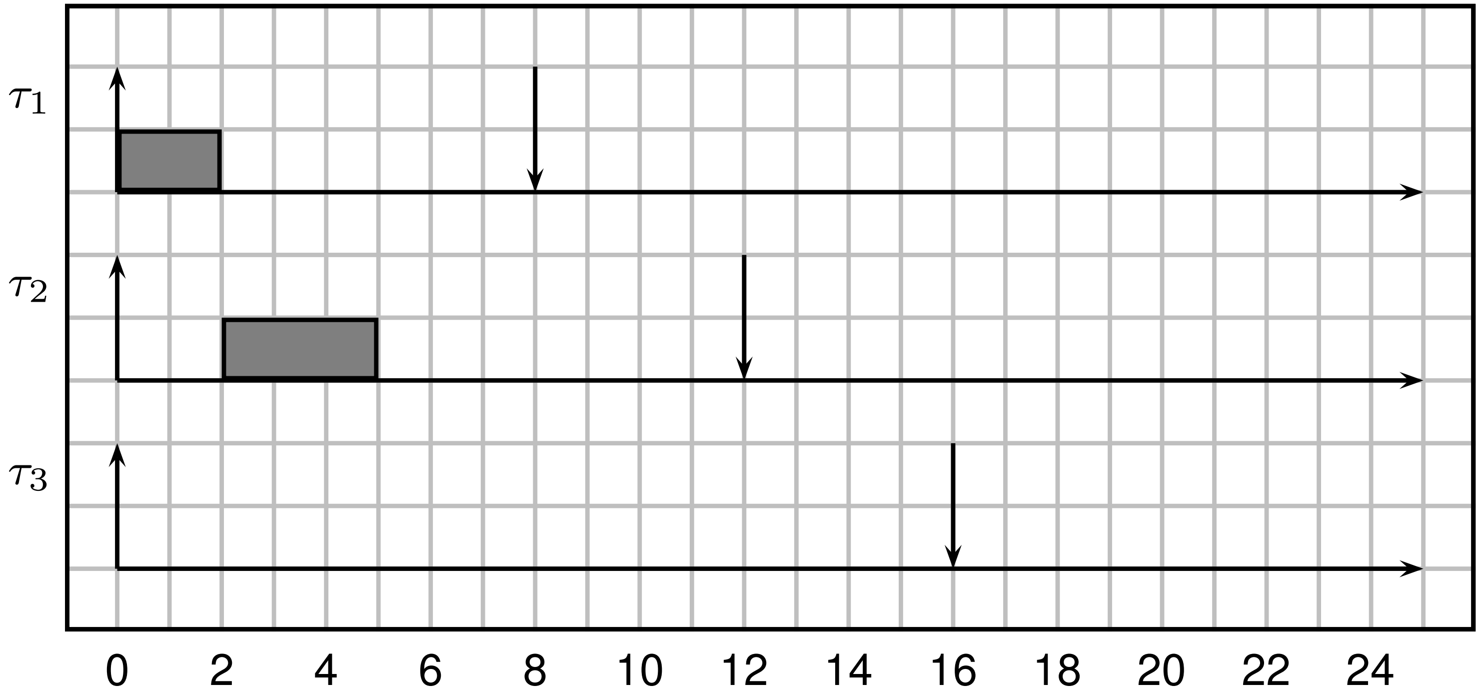


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16);$

$$U = 0.75 < U_{lub} = 0.77$$

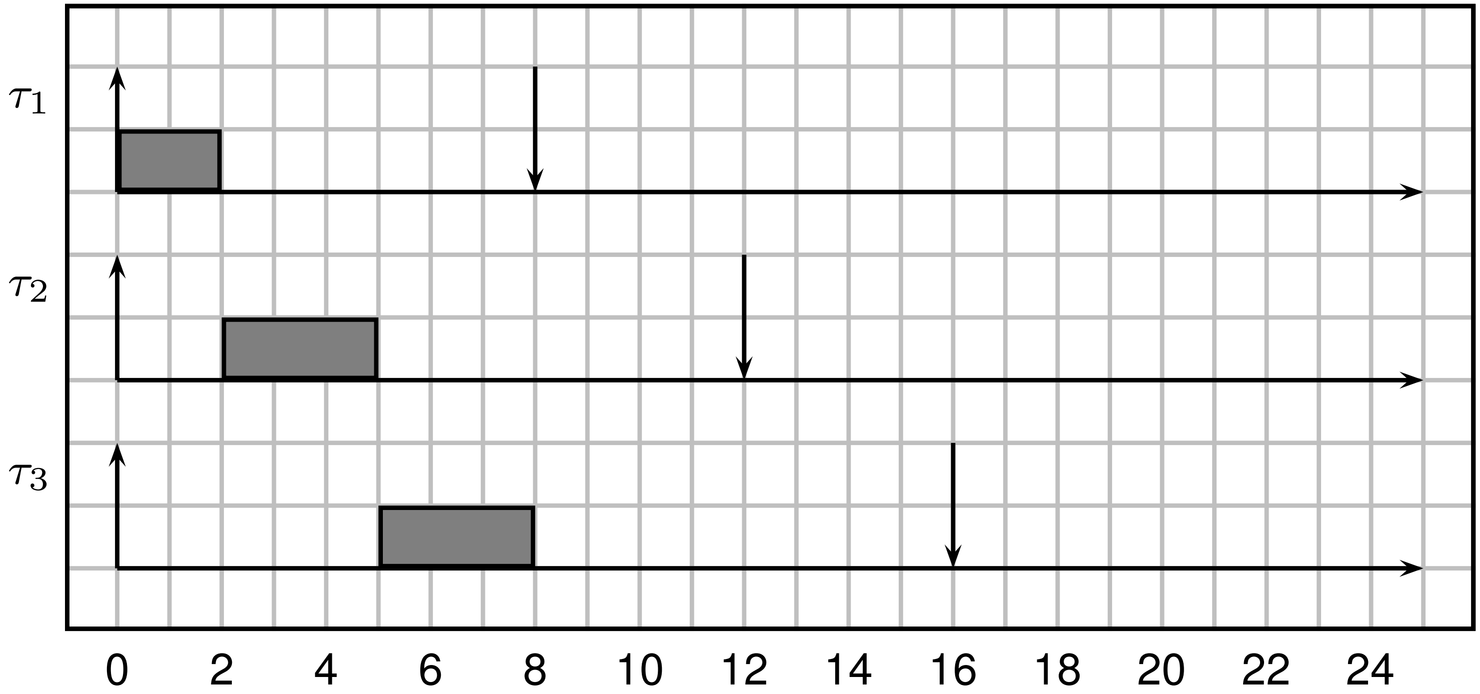


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16)$;

$$U = 0.75 < U_{lub} = 0.77$$

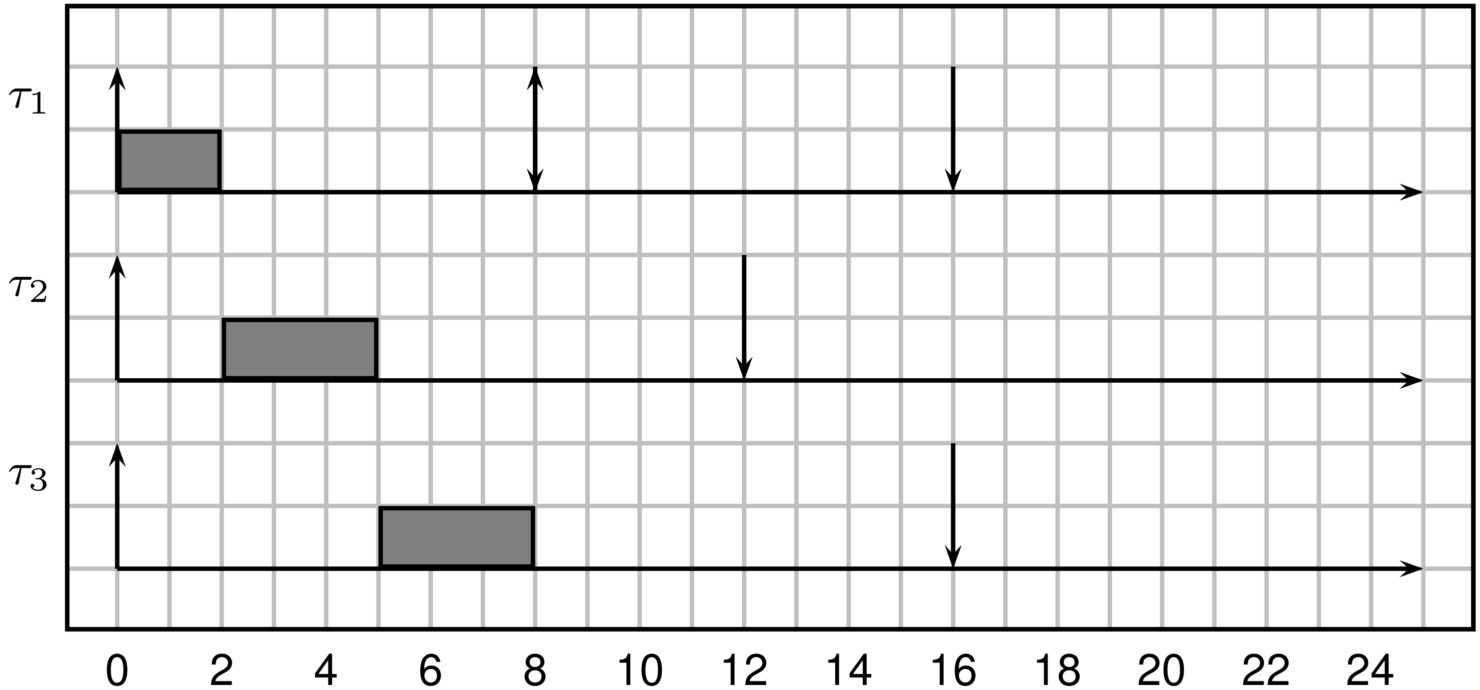


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16);$

$$U = 0.75 < U_{lub} = 0.77$$

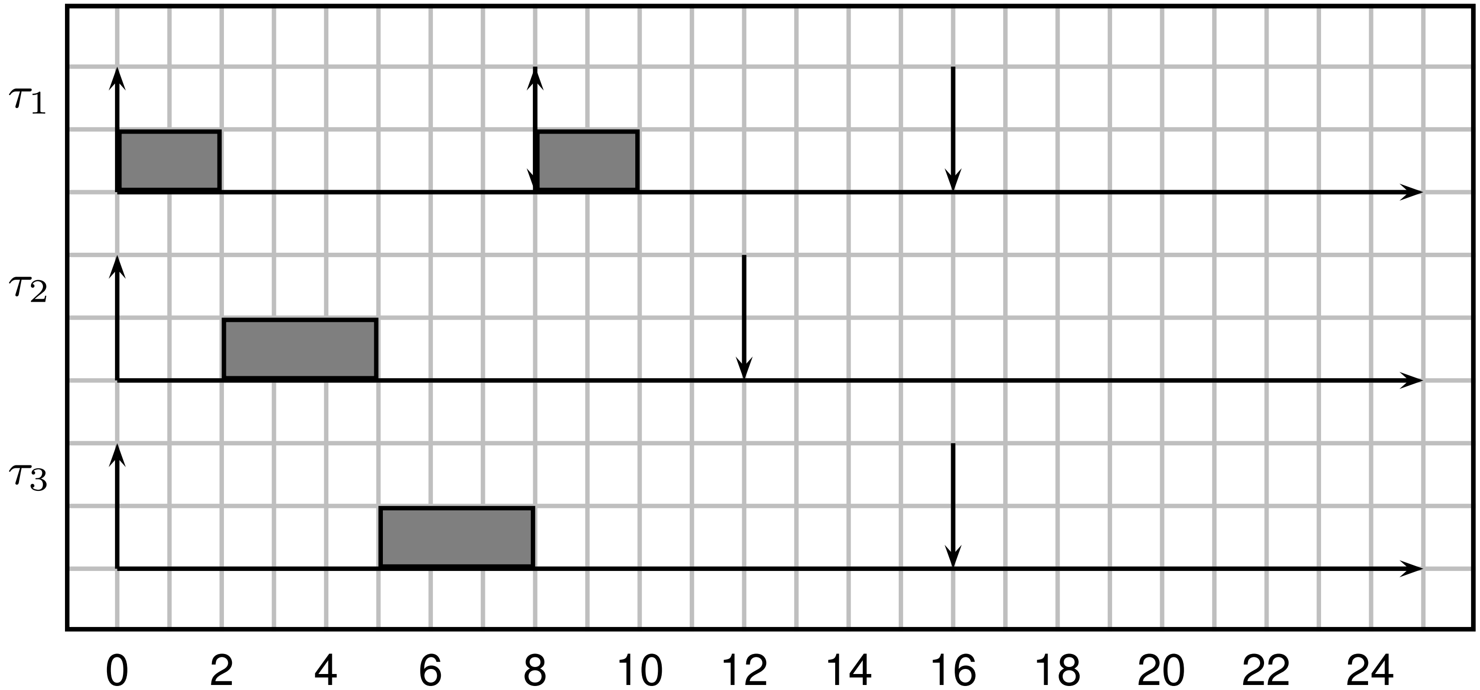


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2,8), \tau_2 = (3,12), \tau_3 = (4,16);$

$$U = 0.75 < U_{lub} = 0.77$$

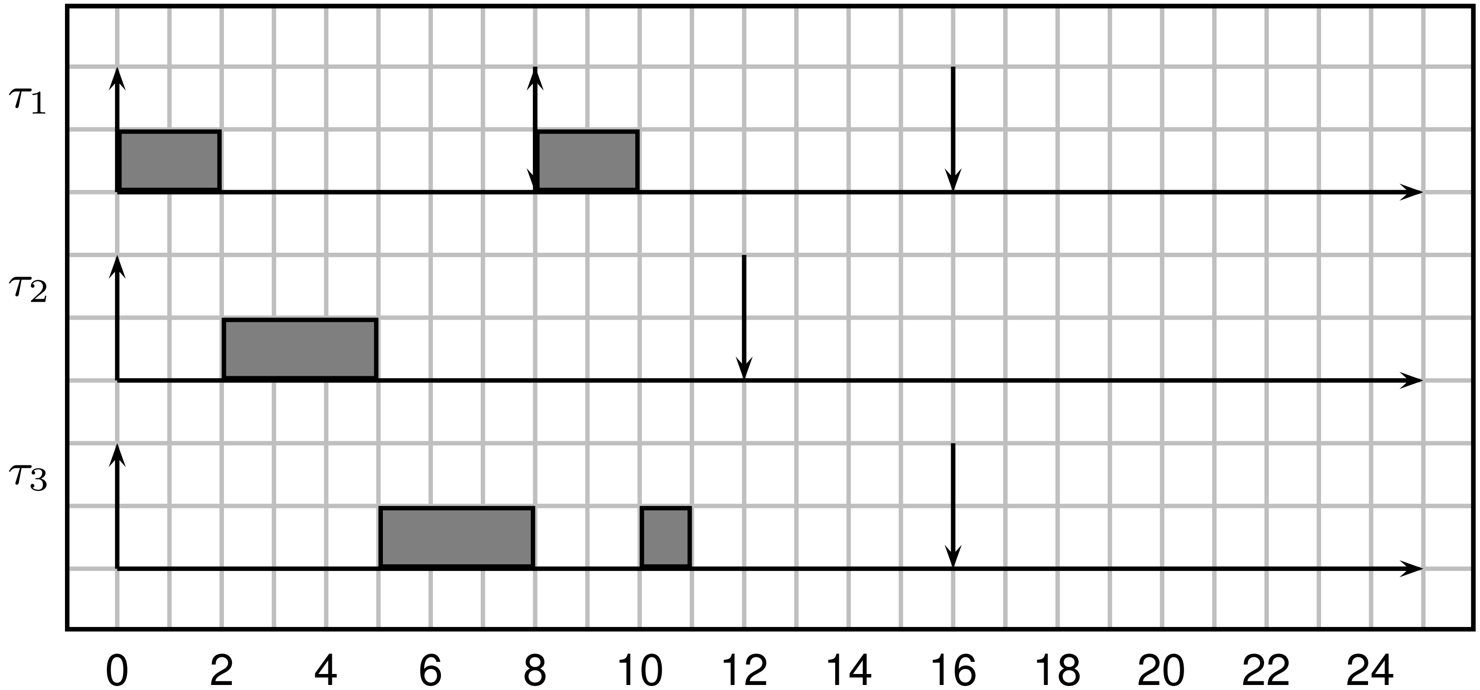


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16);$

$$U = 0.75 < U_{lub} = 0.77$$

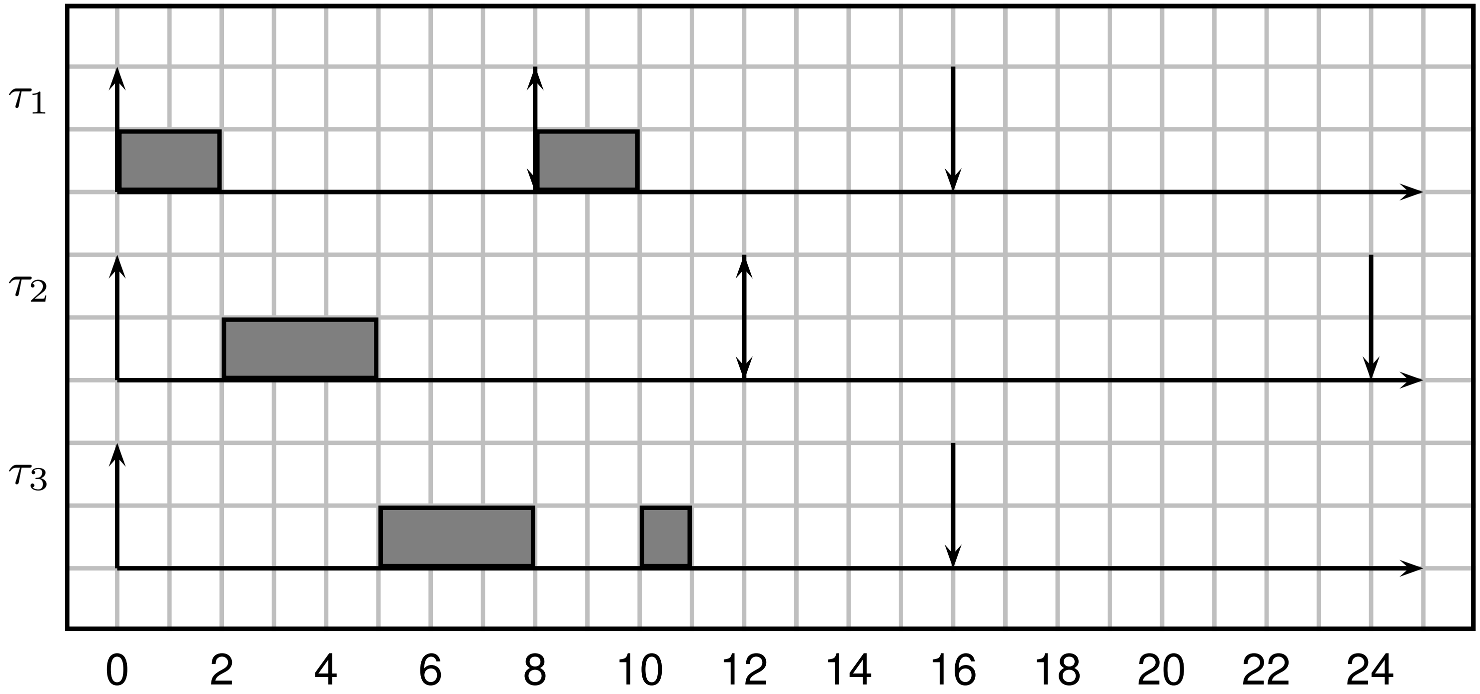


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16);$

$$U = 0.75 < U_{lub} = 0.77$$

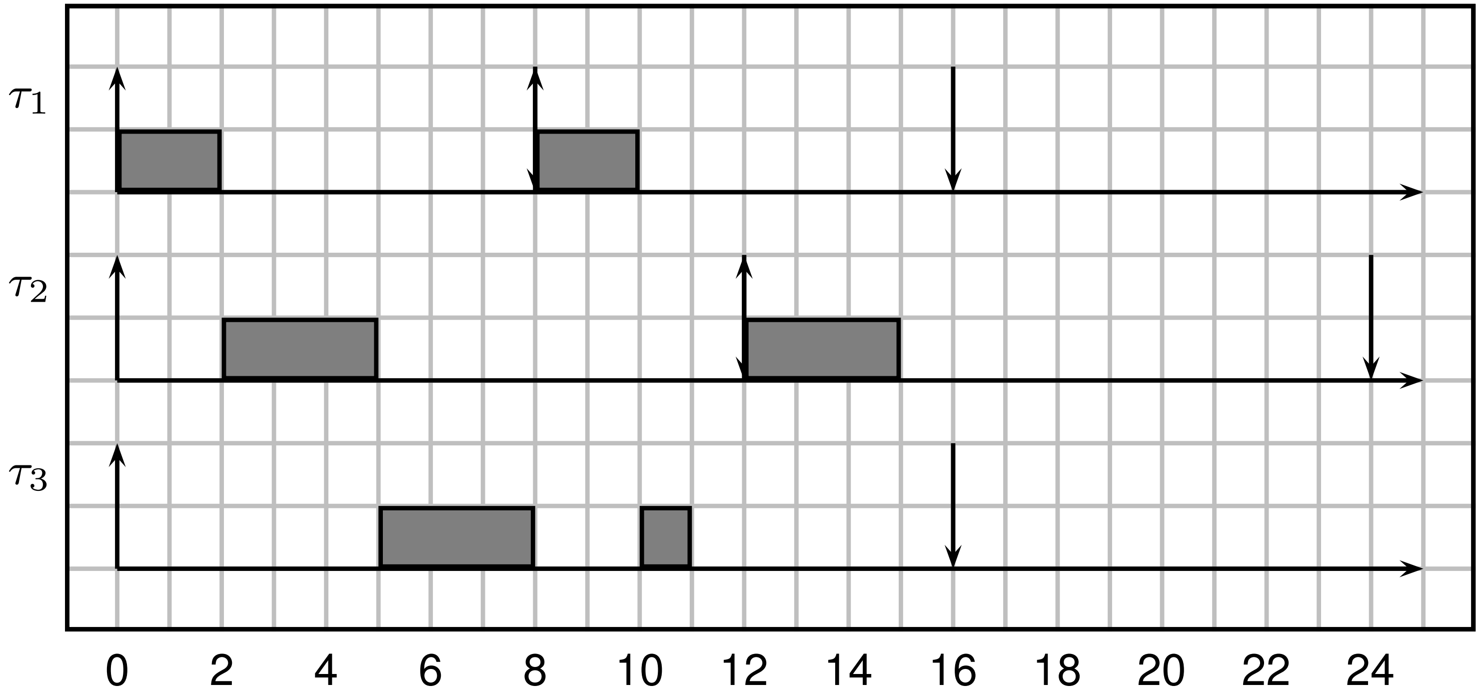


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16)$;

$$U = 0.75 < U_{lub} = 0.77$$

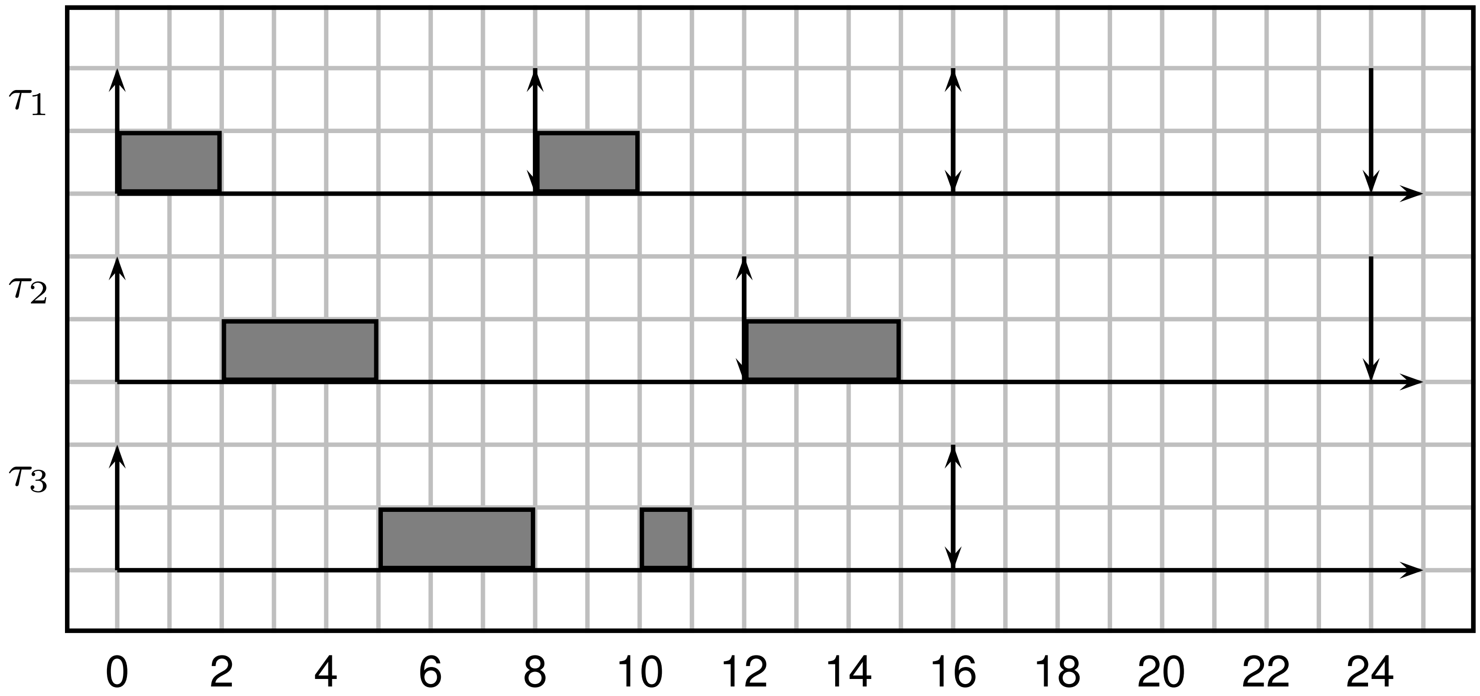


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16);$

$$U = 0.75 < U_{lub} = 0.77$$

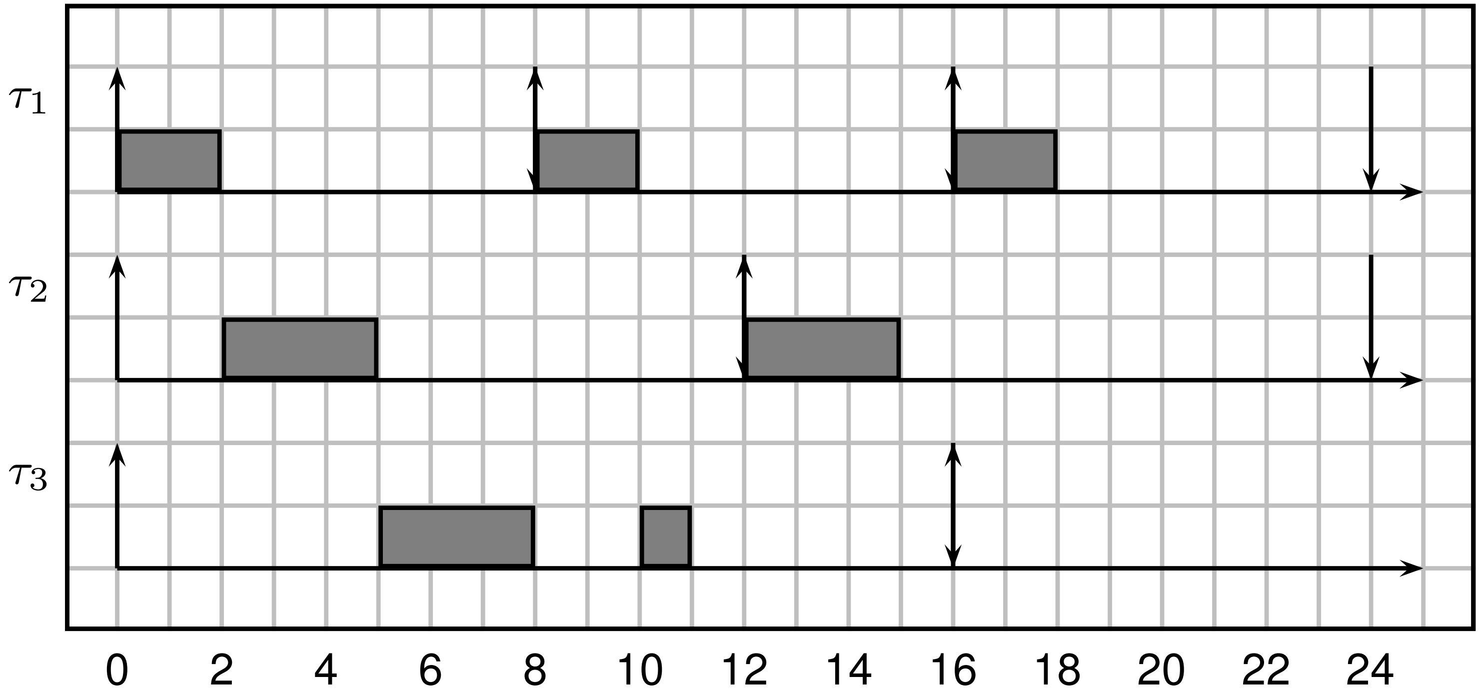


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16);$

$$U = 0.75 < U_{lub} = 0.77$$

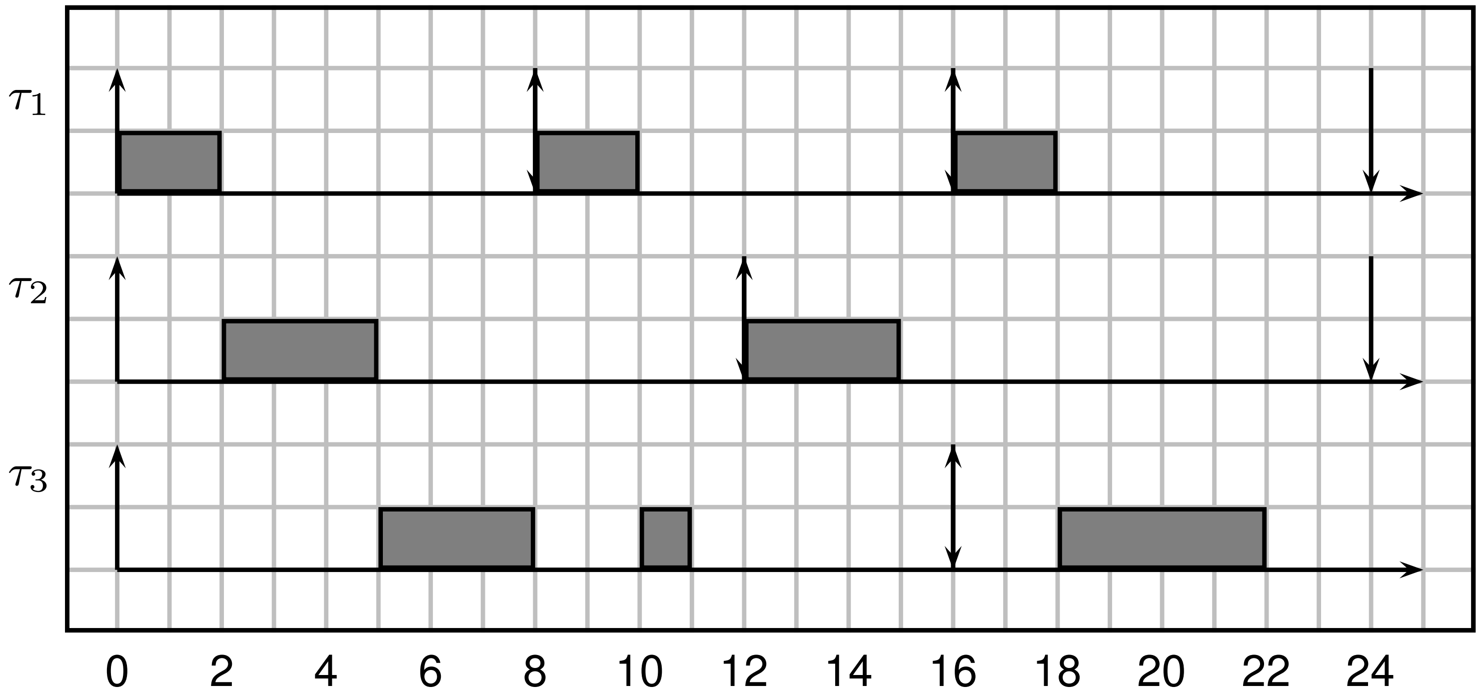


Example

Example in which we show that for 3 tasks, if $U < U_{lub}$, the system is schedulable.

$\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16);$

$$U = 0.75 < U_{lub} = 0.77$$



Pessimism

- The bound is very Conservative: most of the times, a task set with $U > U_{lub}$ is schedulable by RM.
- A particular case is when tasks have periods that are *harmonic*:
  - A task set is *harmonic* if, for every two tasks τ_i, tau_j, either P_i is multiple of P_j or P_j is multiple of P_i.
- For a harmonic task set, the utilization bound is $U_{lub} = 1$.
- In other words, Rate Monotonic is an *optimal* algoritm for harmonic task sets.

A necessary and sufficient test

Response time analysis

- A necessary and sufficient test is obtained by computing the *worst-case response time* (WCRT) for every task.
- For every task τ_i:
  - Compute the WCRT R_i for task τ_i;
  - If $R_i \leq D_i$, then the task is schedulable;
  - else, the task is not schedulable; we can also show the situation that make task τ_i miss its deadline!

- To compute the WCRT, we do not need to do any assumption on the priority assignment.
- The algorithm described in the next slides is valid for an arbitrary priority assignment.
- The algorithm assumes periodic tasks with no offsets, or sporadic tasks.

Response time analysis - II

- The *critical instant* for a set of periodic real-time tasks, with offset equal to 0, or for sporadic tasks, is when all jobs start at the same time.
- **Theorem:** The WCRT for a task corresponds to the response time of the job activated at the critical instant.
- To compute the WCRT of task τ_i:
  - We have to consider its computation time
  - and the computation time of the higher priority tasks (*interference*);
  - higher priority tasks can *preempt* task τ_i, and increment its response time.

Response time analysis - III

- Suppose tasks are ordered by decreasing priority. Therefore, $i < j \rightarrow prio_i > prio_j$.
- Given a task τ_i, let $R_i^{(k)}$ be the WCRT computed at step k.

$$R_i^{(0)} = C_i + \sum_{j=1}^{i-1} C_j$$

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

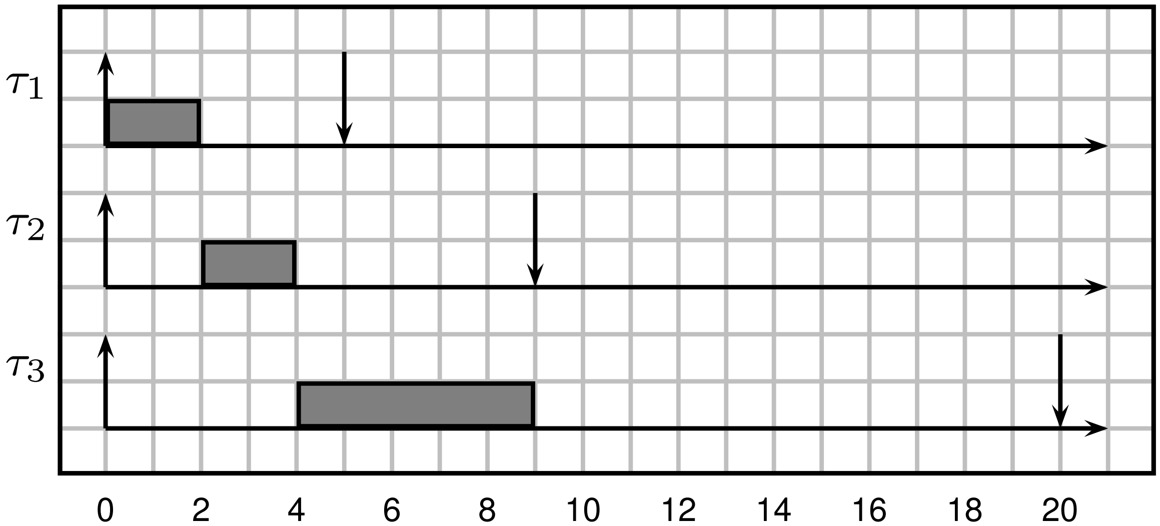
- The iteration stops when:
  - $R_i^{(k)} =_i^{(k+1)}$ *or*
  - $R_i^{(k)} > D_i$ (non schedulable);

Example

Consider the following task set: $\tau_1 = (2,5)$, $\tau_2 = (2,9)$, $\tau_3 = (5,20)$; $U = 0.872$.

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

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

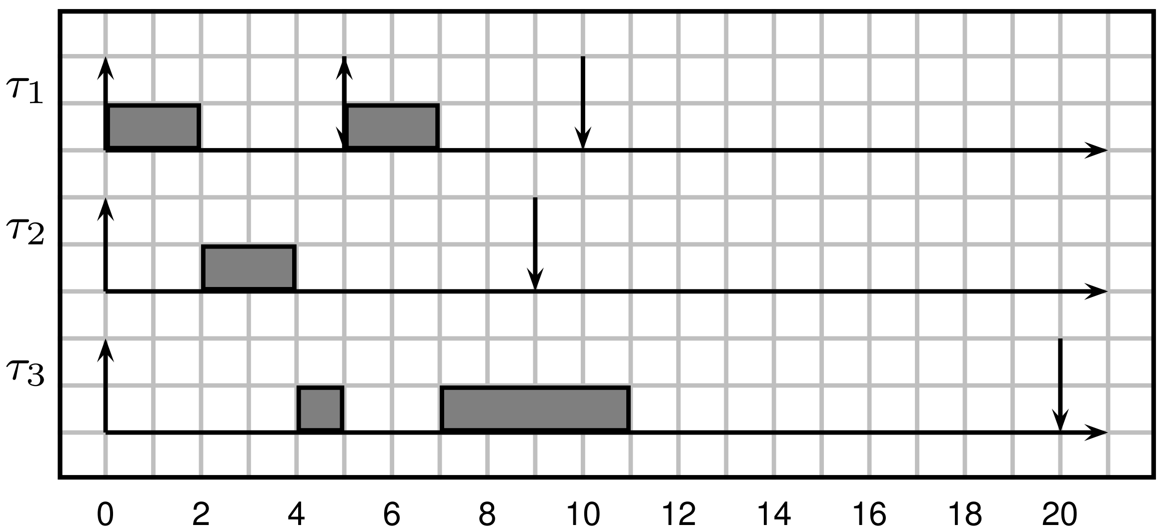


Example

Consider the following task set: $\tau_1 = (2,5)$, $\tau_2 = (2,9)$, $\tau_3 = (5,20)$; $U = 0.872$.

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

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

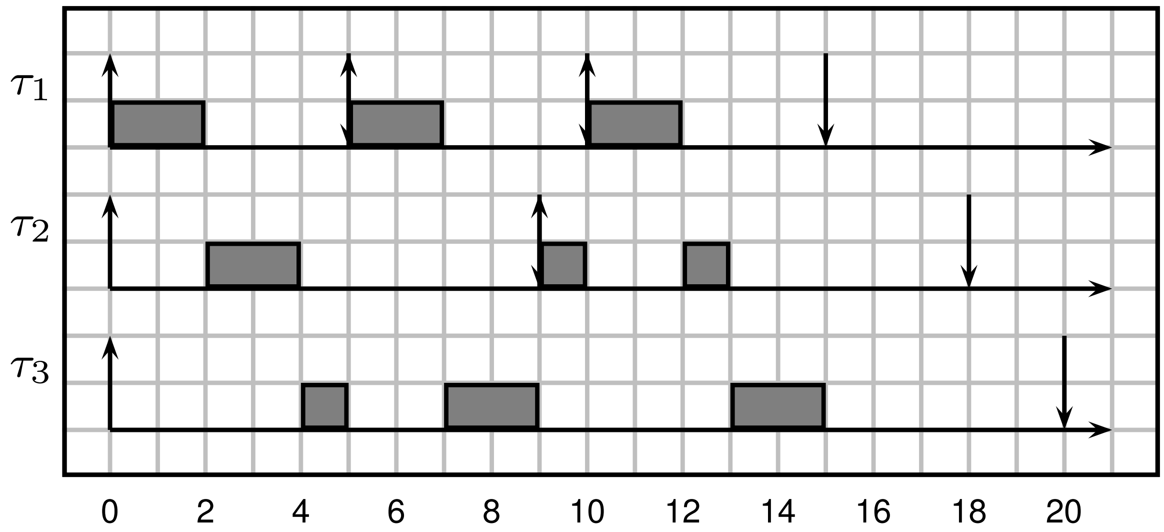


Example

Consider the following task set: $\tau_1 = (2,5)$, $\tau_2 = (2,9)$, $\tau_3 = (5,20)$; $U = 0.872$.

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

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

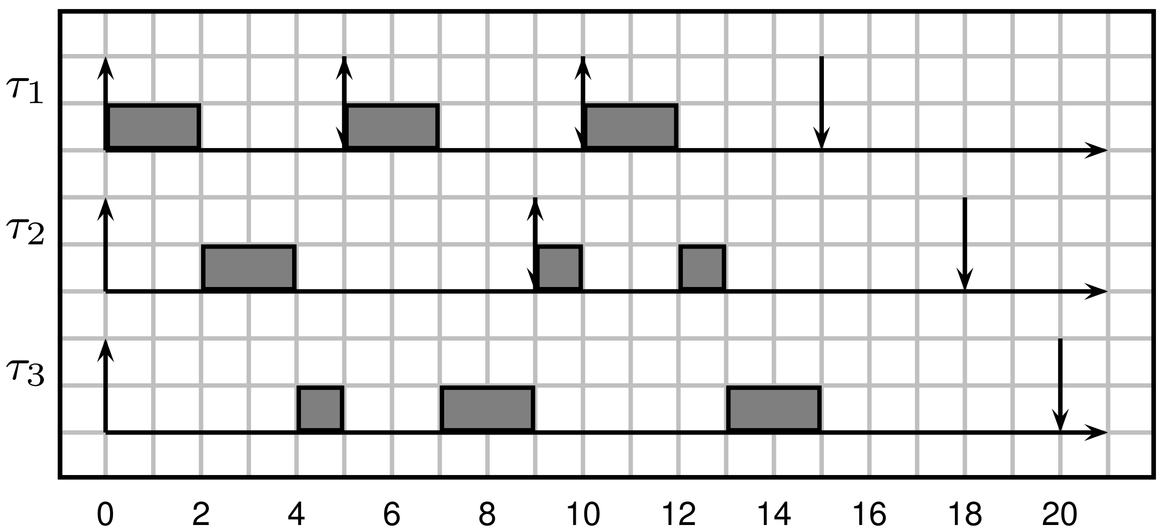


Example

Consider the following task set: $\tau_1 = (2, 5)$, $\tau_2 = (2, 9)$, $\tau_3 = (5, 20)$; $U = 0.872$.

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

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



Interacting Tasks

Interacting tasks

- 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.

Posix Esample

```
    sem_t s;
    ...
    sem_init(&s,1);
    ...
    void * tau1(void * arg) {
        sem_wait(&s);
        <critical section>
        sem_post(&s);
};
...
    void * tau1(void * arg) {
        sem_wait(&s);
        <critical section>
        sem_post(&s);
};
```

Implications of resource sharing

Notation

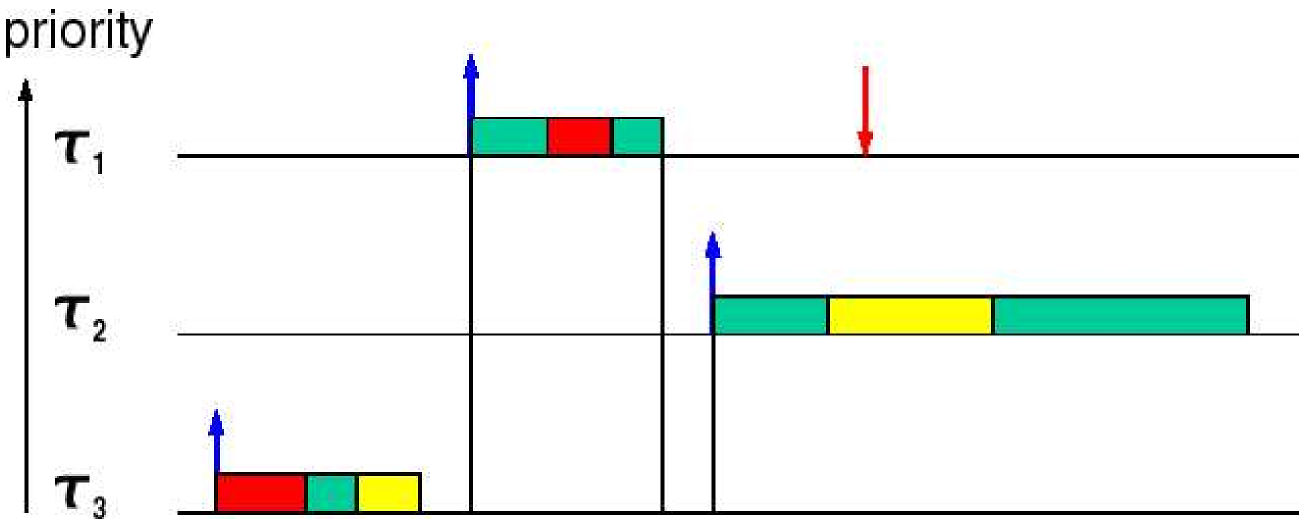
- The resource and the corresponding mutex semaphore will be denoted by symbol S_j.
- A system consists of:
  - A set of N periodic (or sporadic) tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$;
  - A set of shared resources $\mathcal{S} = \{S_1, \ldots, S_M\}$;
  - We say that a task τ_i *uses* resource 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 time

- A first important implocation of resource sharing is blocking time
- A blocking condition happens when a high priority tasks wants to access a resource that is held by a lower priority task.
- A task incurs a blocking condition depending on the interleaving of the schedule

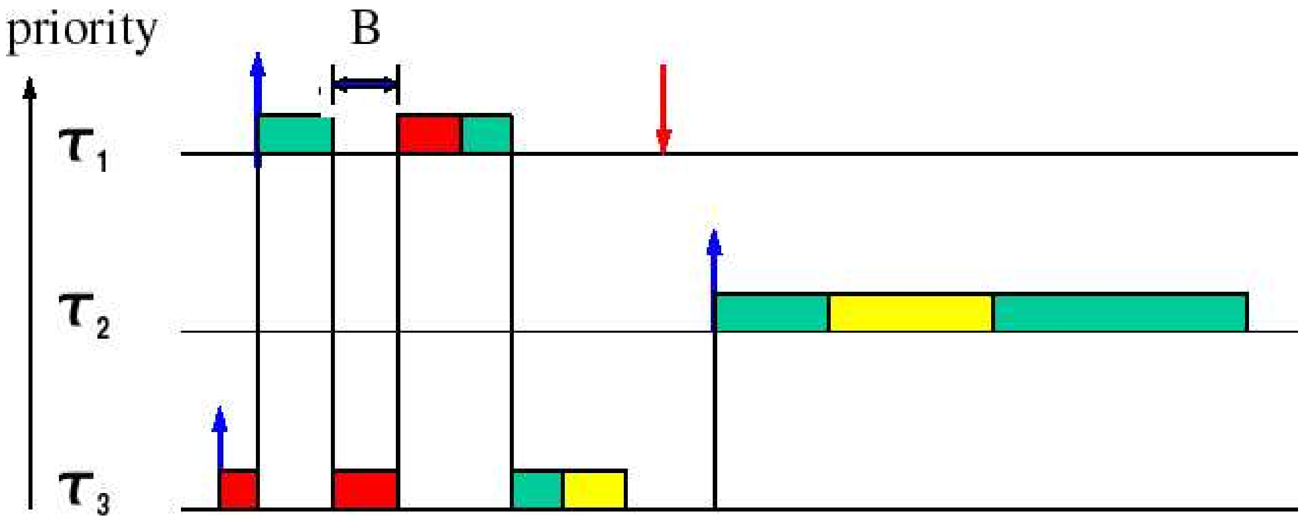
Example

No conficts in this case



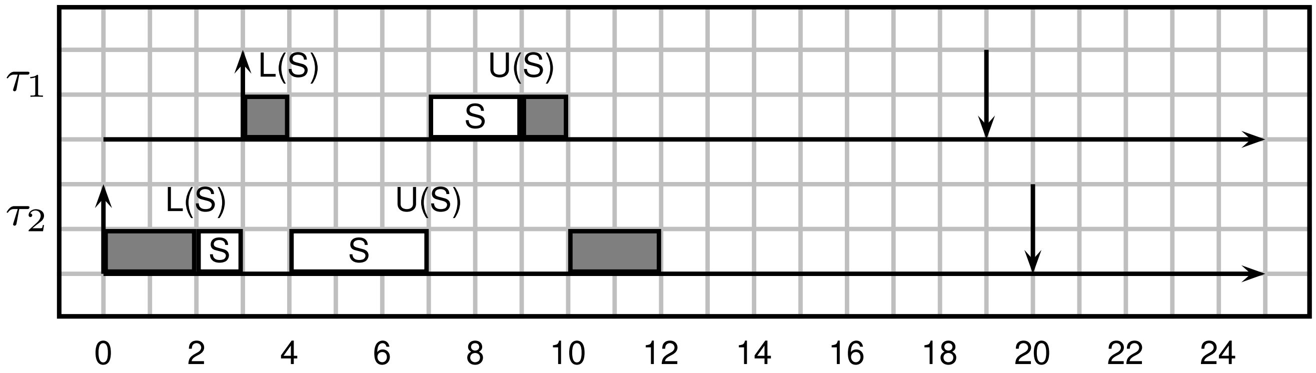
Example

Blocking time in this case



Blocking and priority inversion

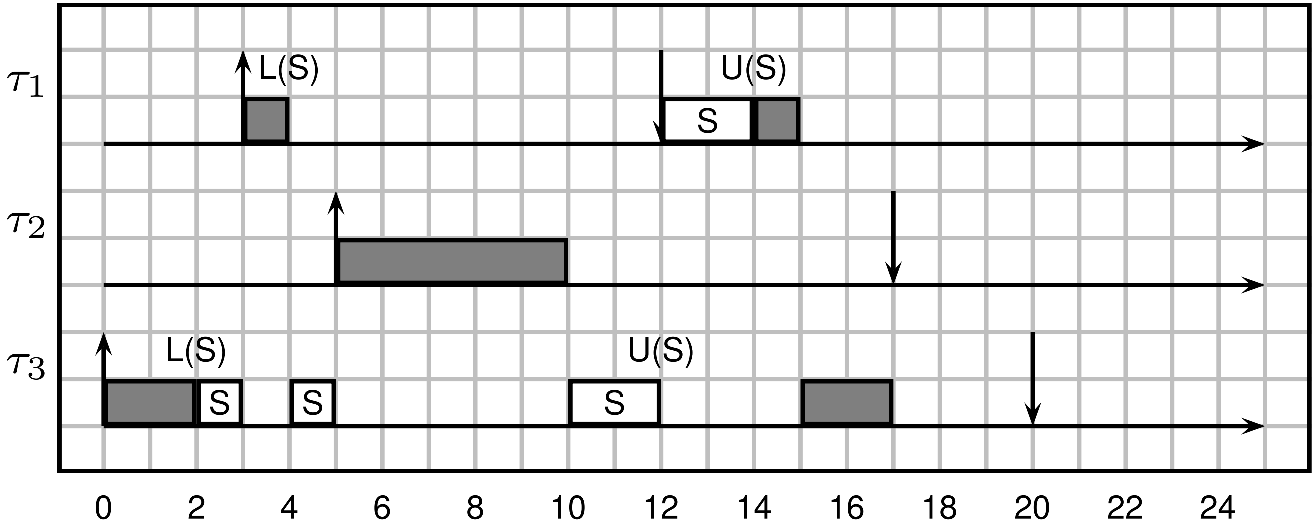
- Consider the following example, where $p_1 > p_2$.



- From time 4 to 7, task τ_1 is blocked by a lower priority taskτ_2; this is a *priority inversion*.
- 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!

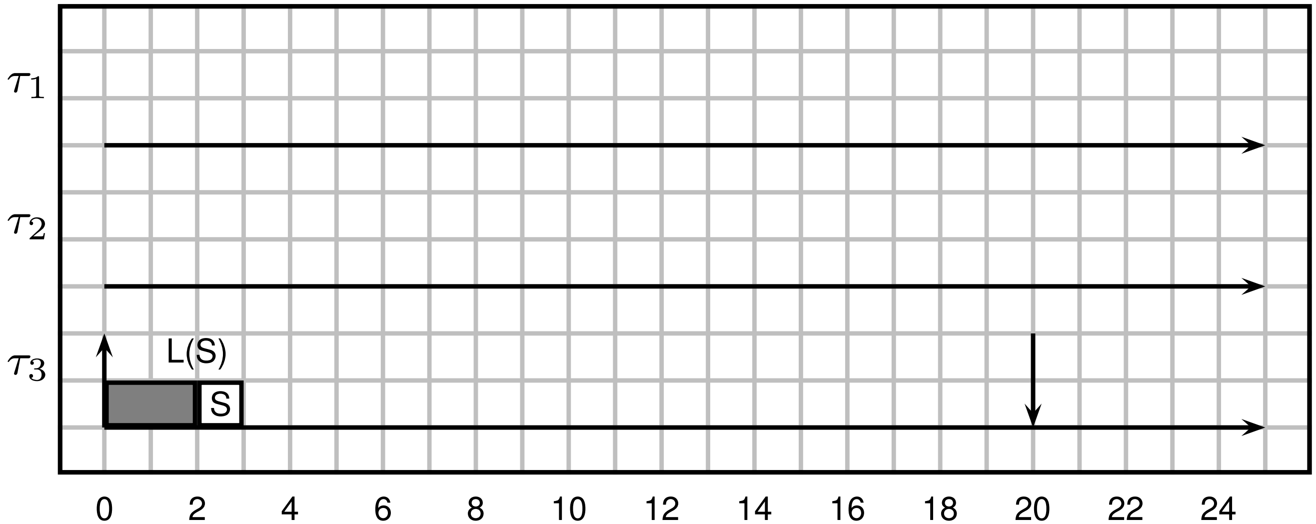
Dealing with priority inversion

- The only way to deal with priority inversion is by the introduction of an appropriate concurrency protocol
  - Non Preemptive protocol
  - Priority inheritance protocol (PIP)
  - Priority Ceiling Protocol (PCP)
  - Immediate Priority Ceiling Protocol (Part of the OSEK and POSIX standards)

The priority Inheritance Protocol

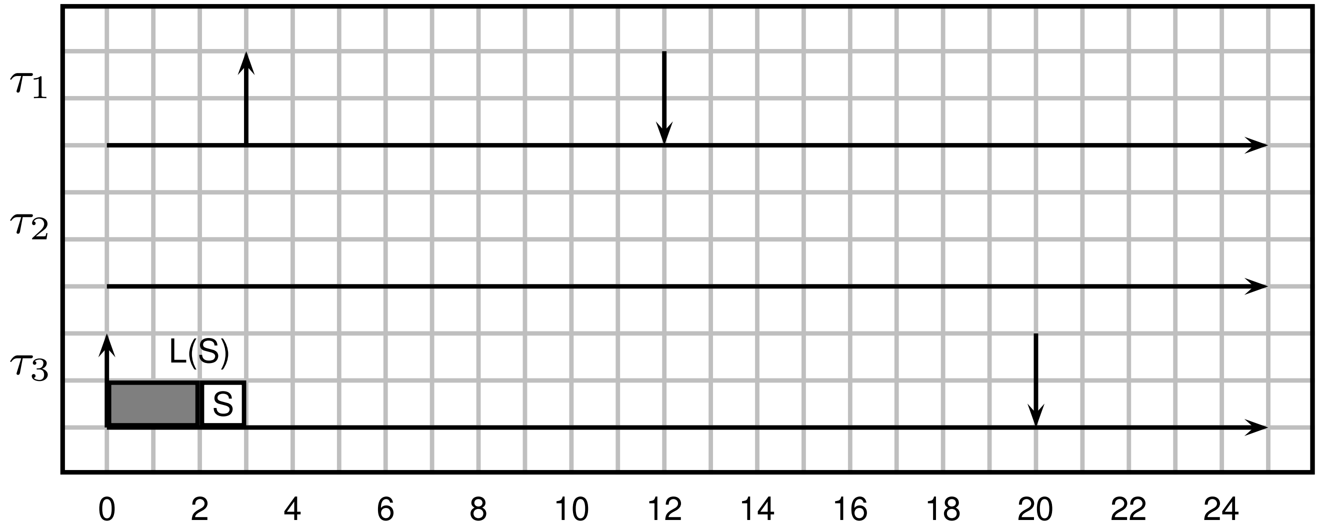
The Priority Inheritance protocol

- The solution to the problem of priority inversion 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:



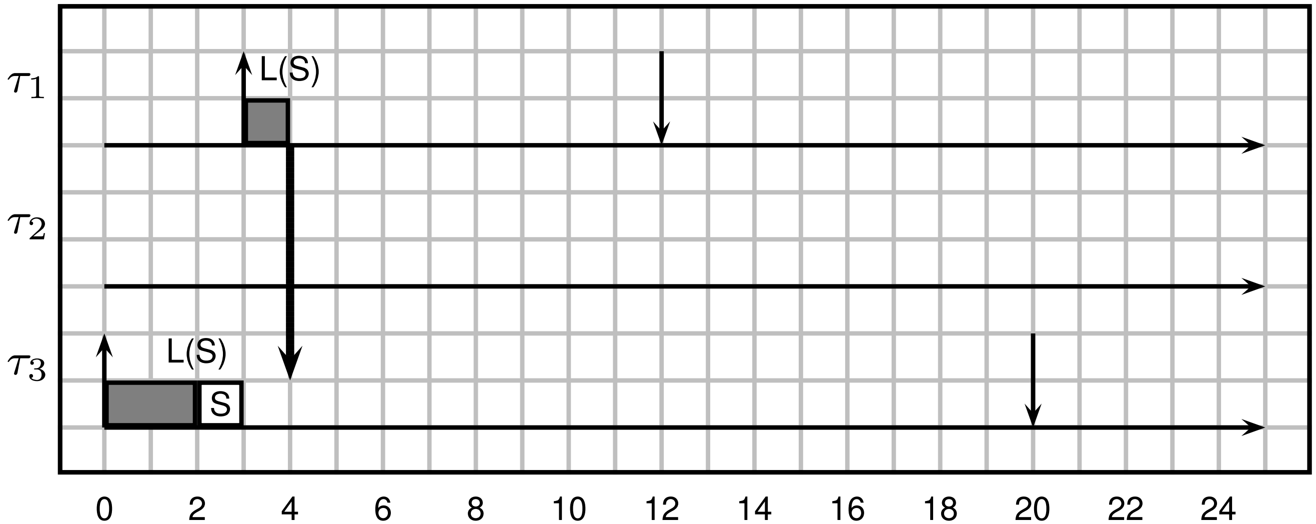
The Priority Inheritance protocol

- The solution to the problem of priority inversion 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:



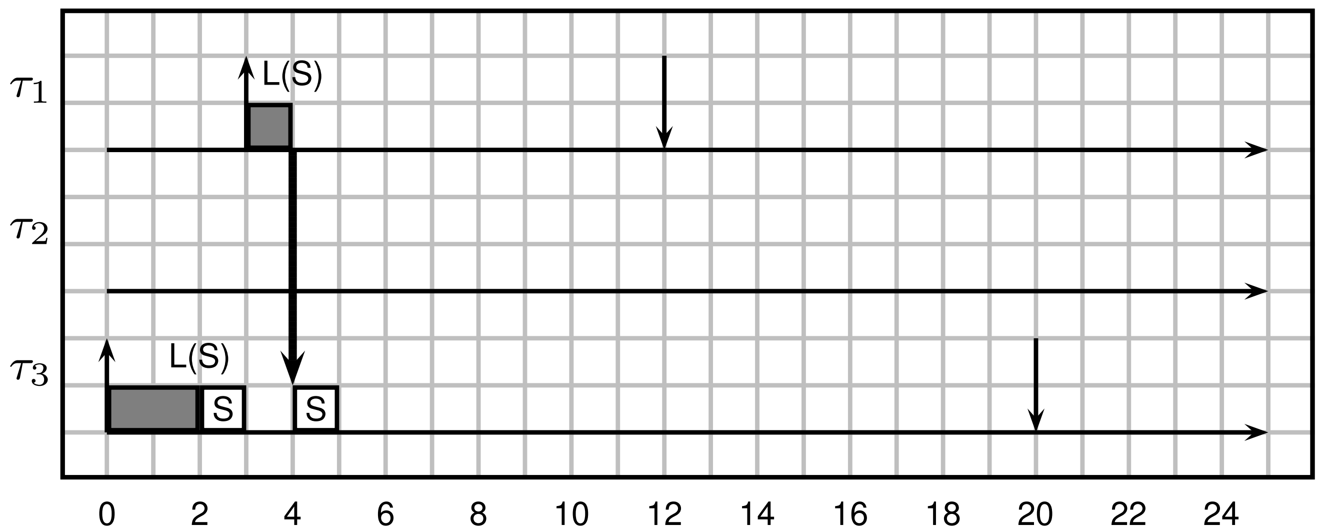
The Priority Inheritance protocol

- The solution to the problem of priority inversion 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

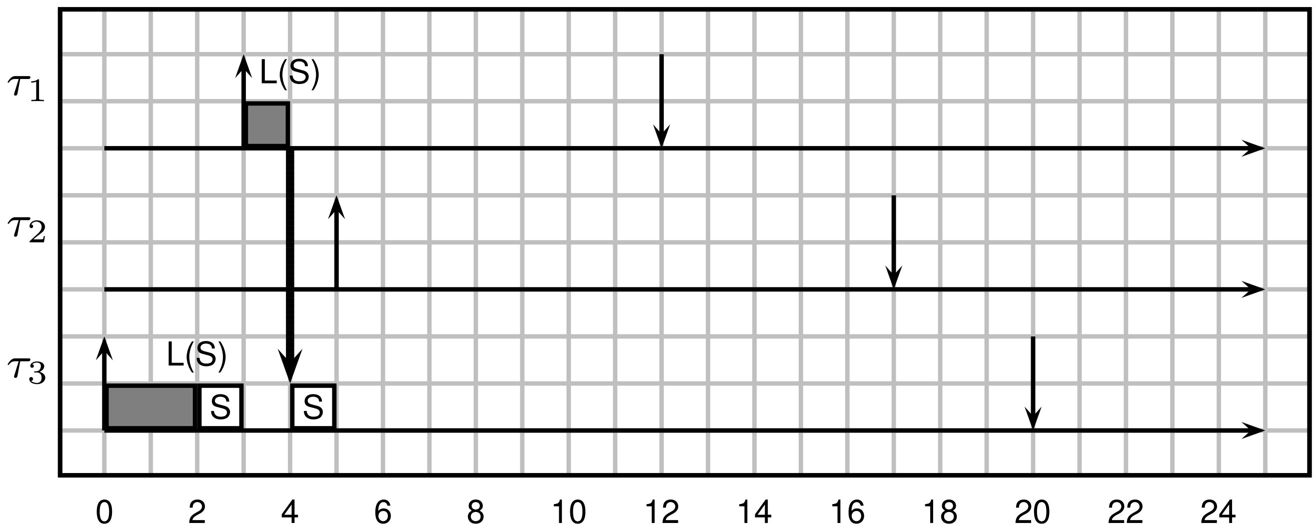
- The solution to the problem of priority inversion is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



  - Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

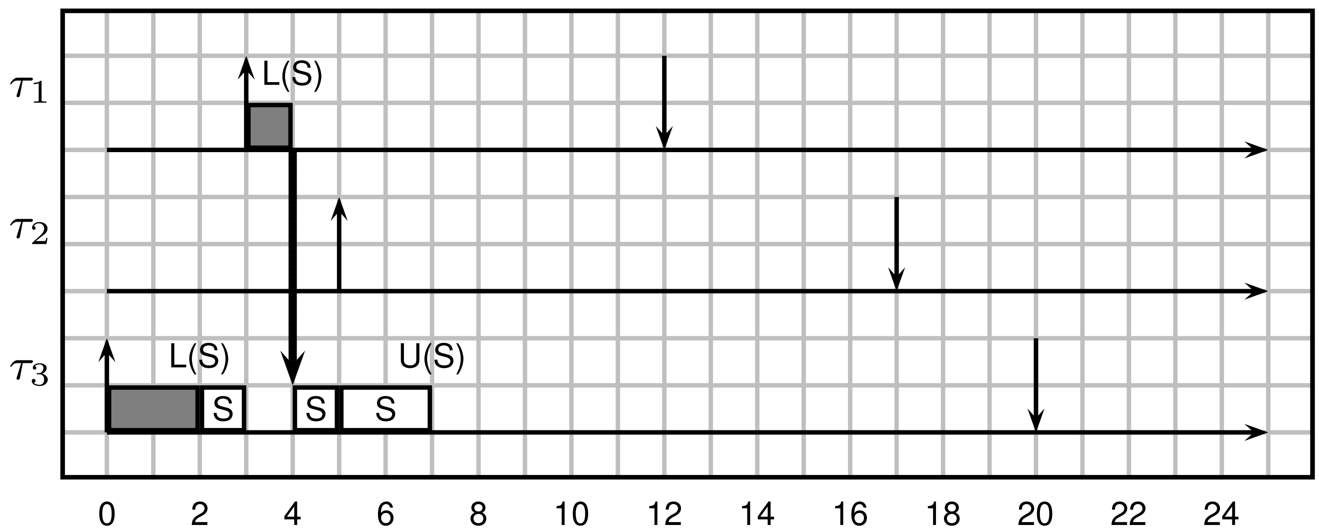
- The solution to the problem of priority inversion is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



  - Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

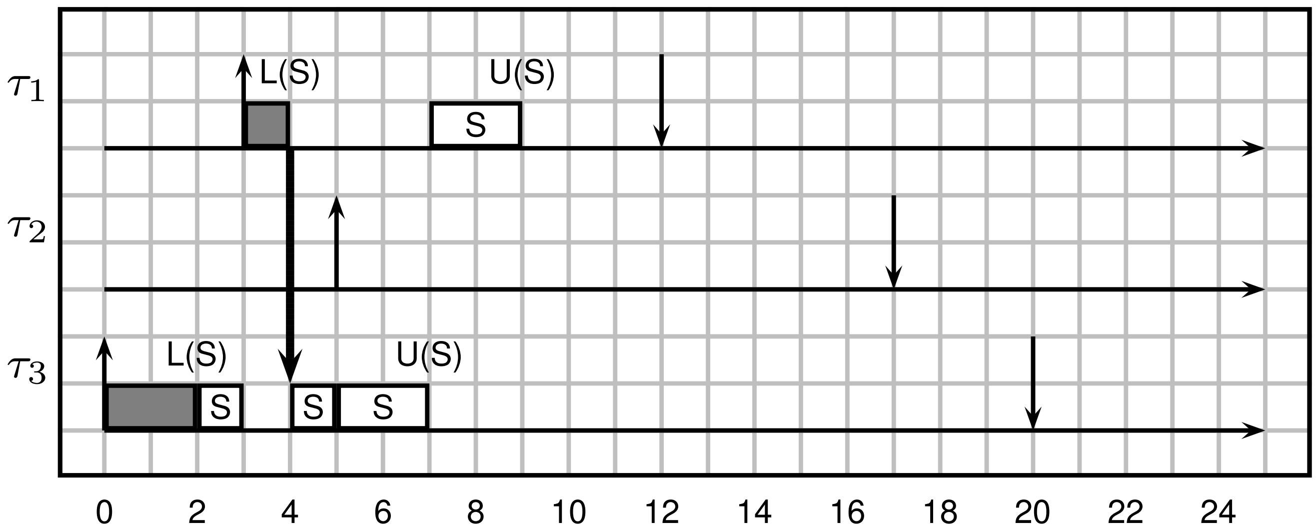
- The solution to the problem of priority inversion is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



  - Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

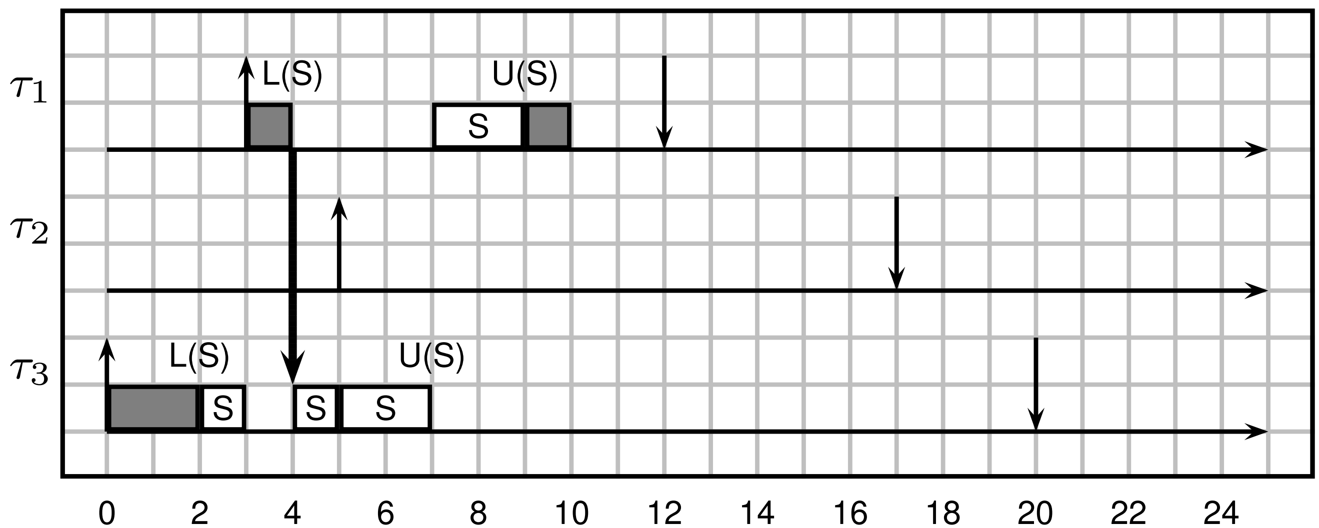
- The solution to the problem of priority inversion is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



  - Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

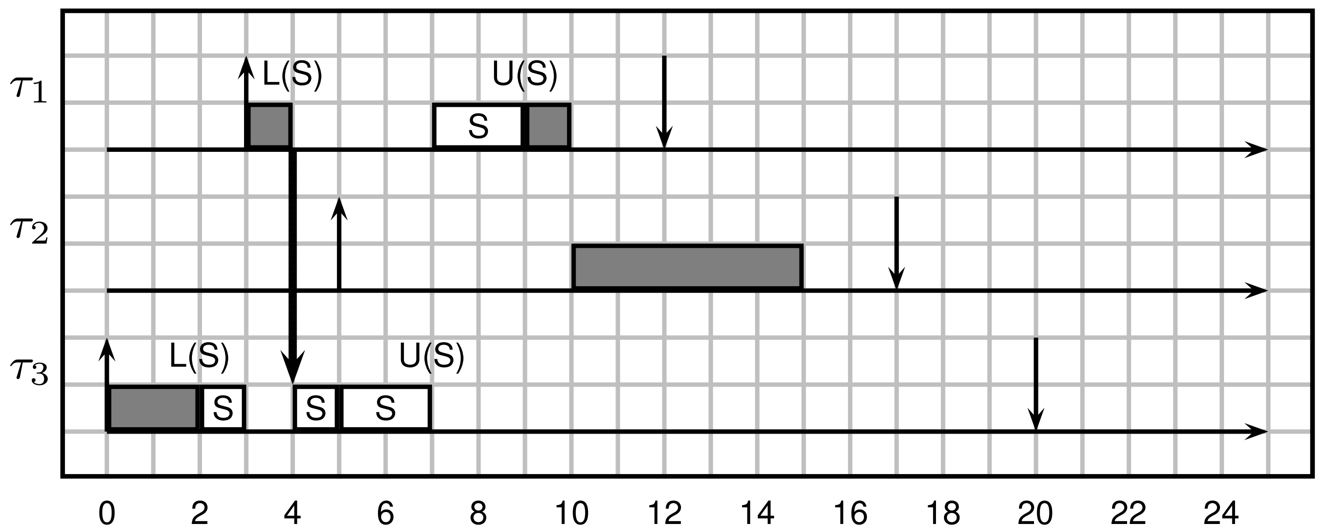
- The solution to the problem of priority inversion is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



  - Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

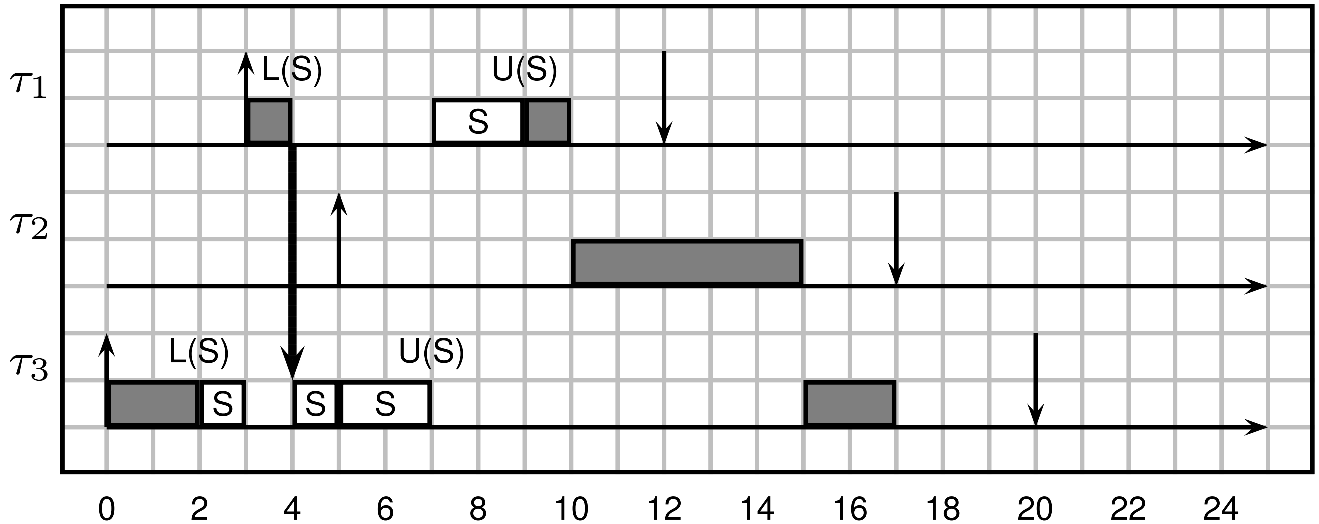
- The solution to the problem of priority inversion is rather simple;
  - While the low priority task blocks an higher priority task, it *inherits* the priority of the higher priority task;
  - In this way, every medium priority task cannot make preemption.
  - In the previous example:



  - Task τ_3 inherits the priority of τ_1

The Priority Inheritance protocol

- The solution to the problem of priority inversion 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

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*
  - 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.

Computing the maximum blocking time

- We will compute the maximum blocking time only in the case of non nested critical sections.
- 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: $\rightarrow 4$;

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: $\rightarrow 5$;
  - 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: $\rightarrow 3$;
- 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

Schedulability analysis

Response time analysis

- In the previous example we have seen the test based on response time analysis

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

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

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