Uniprocessor real-time scheduling

Julien Forget

Université Lille 1

Ecole d'Été Temps Réel - 30 Août 2017

Overview

At the end of this session, you should be (more) proficient at:

- Modeling a real-time system, using the classic periodic task model;
- Programming a set of periodic tasks;
- Testing the schedulability of a task set.

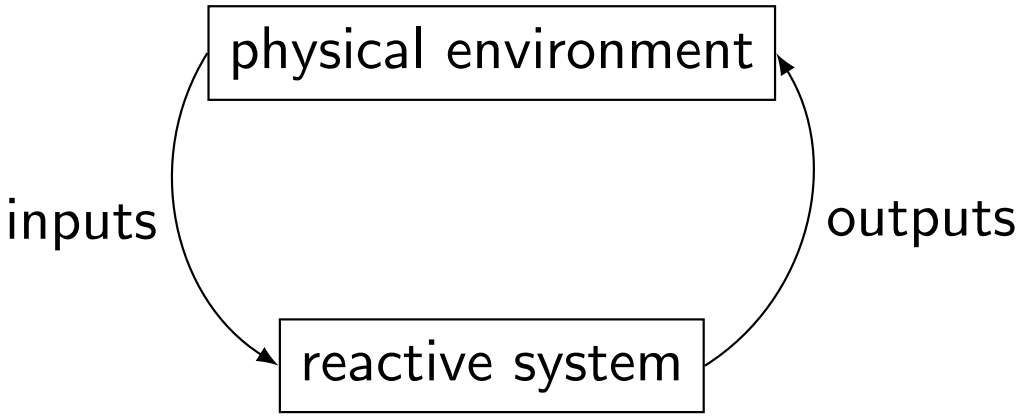
Embedded system

Definition

A special-purpose computer system designed to perform one or a few dedicated functions. Usually embedded as part of a complete device including hardware and mechanical parts. (Wikipedia)

- Dedicated $\Rightarrow$ optimized, reduced resources;
- Handle a system in its (annoying) physical environment.

Reactive system



- React to inputs:
  1. Acquire inputs on sensors;
  2. Compute;
  3. Produce values on actuators.
- Actions impact the environment, thus subsequent inputs;
- Response time must be *bounded*.

Real-time system

Definition

Real-time systems must guarantee response within strict time constraints, often referred to as "deadlines".
(Wikipedia)

- A sub-class of reactive systems;
- Usually embedded;
- Often critical.

Classify these systems

Discuss which systems are embedded, reactive, critical, real-time:

- Airplane flight control system;
- Data base;
- ABS;
- Compiler;
- Nuclear power plant control system;
- MP3 player;
- GSM communication;
- Video processing on chip.

Real-time systems classes

- *Hard* real-time: missing a deadline has catastrophic consequences;
- *Soft* real-time: missing a deadline degrades the behaviour (notion of quality of service or performance level).

Hard or soft real-time?

- Mpeg Video decoder;
- Airplane flight control system;
- Car cruise control;
- GSM communication.

Outline

1. Real-time model
2. Scheduling
3. Fixed-task priority
4. Fixed-job priority
5. Shared resources
6. Data-dependencies
7. Conclusion

Event-driven vs time-driven

- Event-driven system:
  - Wait for input;
  - Compute;
  - Produce output;

  ⇒ Must produce output before occurrence of next event.
- Sampled system:
  - Acquire input at regular intervals of time;
  - Compute;
  - Produce output;

  ⇒ Must produce output before start of next interval.
- Inter-arrival time of events is often bounded:

⇒ When so, both models are not that much different (*sporadic tasks*).

Period

Periods are the prime real-time constraints:

- Period (T_i): repetition interval of a task;
- Choosing a period:
  - Upper-bounded:

Period

Periods are the prime real-time constraints:

- Period (T_i): repetition interval of a task;
- Choosing a period:
  - Upper-bounded: system dynamics (e.g. airplane stability);

Period

Periods are the prime real-time constraints:

- Period (T_i): repetition interval of a task;
- Choosing a period:
  - Upper-bounded: system dynamics (e.g. airplane stability);
  - Lower-bounded:

Period

Periods are the prime real-time constraints:

- Period (T_i): repetition interval of a task;
- Choosing a period:
  - Upper-bounded: system dynamics (e.g. airplane stability);
  - Lower-bounded: sensor/actuator capabilities and processor load;

Period

Periods are the prime real-time constraints:

- Period (T_i): repetition interval of a task;
- Choosing a period:
  - Upper-bounded: system dynamics (e.g. airplane stability);
  - Lower-bounded: sensor/actuator capabilities and processor load;
  - Choose a period close to

Period

Periods are the prime real-time constraints:

- Period (T_i): repetition interval of a task;
- Choosing a period:
  - Upper-bounded: system dynamics (e.g. airplane stability);
  - Lower-bounded: sensor/actuator capabilities and processor load;
  - Choose a period close to upper-bound to reduce load:

Period

Periods are the prime real-time constraints:

- Period (T_i): repetition interval of a task;
- Choosing a period:
  - Upper-bounded: system dynamics (e.g. airplane stability);
  - Lower-bounded: sensor/actuator capabilities and processor load;
  - Choose a period close to upper-bound to reduce load:
    - Reduce hardware cost;
    - Reduces energy consumption.
- Related attributes: release date and deadline.

Deadline

- Deadline (D_i): bounds the accepted response time of a task;
- $D_i = T_i$:
  - Most classic model;
  - A task invocation must complete before its next invocation;
- $D_i \leq T_i$:
  - In some cases related to system dynamics, e.g. actuator physical reaction time;
  - Often used as an implementation "trick": influence schedule by increasing task priority (e.g. see data-dependencies).
- $D_i \geq T_i$: rarely used, but studied in litterature.

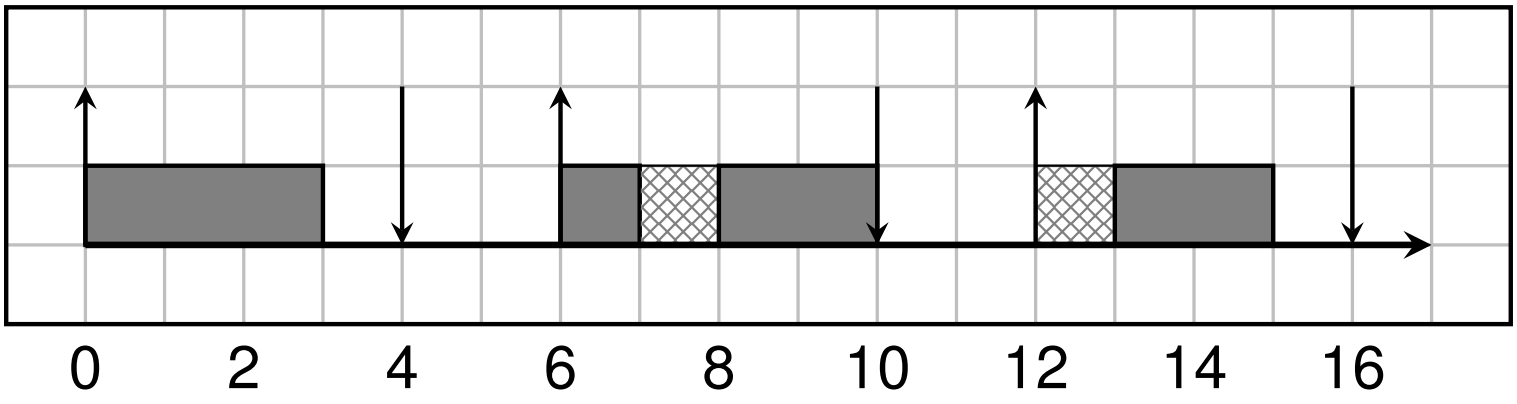
Offset

- Offset (O_i): wait before the initial release of a task;
- Rarely related to system dynamics;
- Again often used as an implementation "trick": delay the point where a task becomes ready (e.g. see data-dependencies).

Worst-Case Execution Time (WCET)

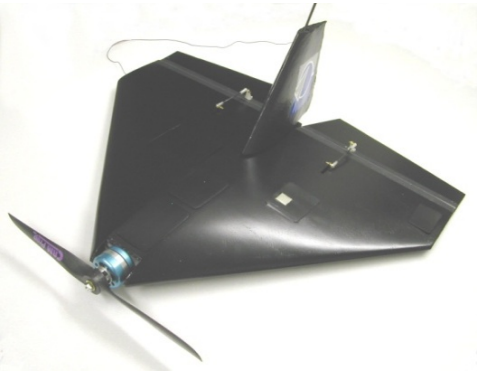
- WCET (C_i): execution-time of a task;
- Exact execution time hard to predict;
- Instead, provide a *safe* but *pessimistic* upper-bound;
- Not an actual constraint: execution time will usually (hopefully) be shorter;
- Used mainly for schedulability analysis.

Task model



- Task $\tau_i(C_i, T_i, D_i O_i)$:
  - $\tau_{i.k}$: the kth ($k \geq 0$) invocation or *job* of τ_i;
  - $o_{i.k} = O_i + kT_i$: release date of $\tau_{i.k}$;
  - Sporadic tasks: $o_{i.k} \geq O_i + kT_i$;
  - $d_{i.k} = o_{i.k} + D_i$: absolute deadline of $\tau_{i.k}$.
- In this example: :
  - $\tau_1(3, 6, 4, 0)$;
  - Deadline constraints are met;
  - $\tau_{1.1}$ and $\tau_{1.2}$ are *preempted*;
  - $\tau_{1.2}$ executes for less than C_1.

Impact of deadline misses: UAV control



- GPS (input): 1 frame every 250 ms.
  - Deadline miss ⇒ frame lost, i.e. current position lost.
- Inertial measurement unit (input): 1 message every 20ms.
  - Deadline miss ⇒ loss of control.
- Radio-command (input): 5 inputs updated every 20ms.
  - Deadline miss ⇒ non-fluid control.
- Servo-command (output): 5 outputs updated every 20ms.
  - Deadline miss ⇒ non-fluid control.

Impact of deadline misses: UAV control (2)

- Data for artificial ground control (output): every 50ms
  - Deadline miss ⇒ flight display delayed.
- Navigation (trajectory): same as GPS.
  - Deadline miss ⇒ wrong trajectory.
- Failure detection (internal): every 200ms
  - Deadline miss ⇒ crash with motors on.
- Attitude regulation (output): once every 3 IMU cycle, i.e. 60ms
  - Deadline miss ⇒ loss of control.

A real-time task in C

```
void* periodic_task1(void* args) {
  //initialization
  wait_for_release();

  while (1) {
    task_functionality();
    wait_next_activation();
  }
}
```

- Wait for release if $O_i \neq 0$ or task startup not synchronized (OS-dependent);
- "Immediate loop" bug: don't forget to wait for next activation!
- This is a very classic structure:
  - Yet, not enforced by the OS or compiler;
  - In some RTOS, programmer only provides the `task_functionality` function, repeated automatically by the OS.

Invoking tasks

```
int main() {
  create_task(periodic_task1,args1,RTparams1);
  create_task(periodic_task2,args2,RTparams2);

  start_scheduler();
  (or) while(1);

  // This should never be reached
  return 1;
}
```

- Arguments of task creation:
  - `periodic_taski`: pointer to task function (see previous slide);
  - `argsi`: the arguments of `periodic_taski`;
  - `RTparamsi`: the real-time parameters;
- Task startup:
  - Linux-based OS: immediate (at task creation) ⇒ need to synchronize initialize releases;
  - Others: when starting the scheduler (`start_scheduler`).
- Warning: in Linux-based OS, termination of `main()` terminates tasks ⇒ prevent termination with `while(1)`.

Outline

1. Real-time model
2. Scheduling
3. Fixed-task priority
4. Fixed-job priority
5. Shared resources
6. Data-dependencies
7. Conclusion

A two-fold problem

Real-time scheduling

Scheduling is the method by which work is assigned to resources. In real-time systems, the scheduler also must ensure that processes can meet deadlines. (Wikipedia)

This involves:

- Designing a *scheduling policy*: an algorithm that dictates how to assign work to resources;
- Designing a *schedulability analysis*: a test that verifies, *before execution*, that the policy will *definitely* satisfy the real-time constraints of the system.

Scheduling: OS vs RTOS

Scheduler objective:

General-purpose OS	RTOS
• Reduce mean response time • Equity between processes • Independent processes	• Ensure bounded response time • $\neq$ deadlines means $\neq$ urgency • Collaborating processes
Be fast, on average	*Be in-time, always*

For a RT system, faster is NOT better.

Scheduling in the development cycle

- Schedulability analysis always occurs off-line;
- *Off-line scheduling*:
  - Execution order is planned before execution;
  - Assumes a repetitive planning;
  - The (possibly large) planning is embedded;
  - Only requires a dispatcher.
- *On-line scheduling*:
  - Scheduler takes decisions on-line;
  - No-need for a pre-computed schedule;
  - Schedulability can still be analyzed off-line.

Feasibility, schedulability

Feasibility

A schedule for a given task set is *feasible* iff:

- Each job starts after its release date;
- Each job completes before its absolute deadline.

Schedulability

A task set is *schedulable* with a given scheduling policy iff the schedule produced by the policy for this task set is feasible.

Optimality

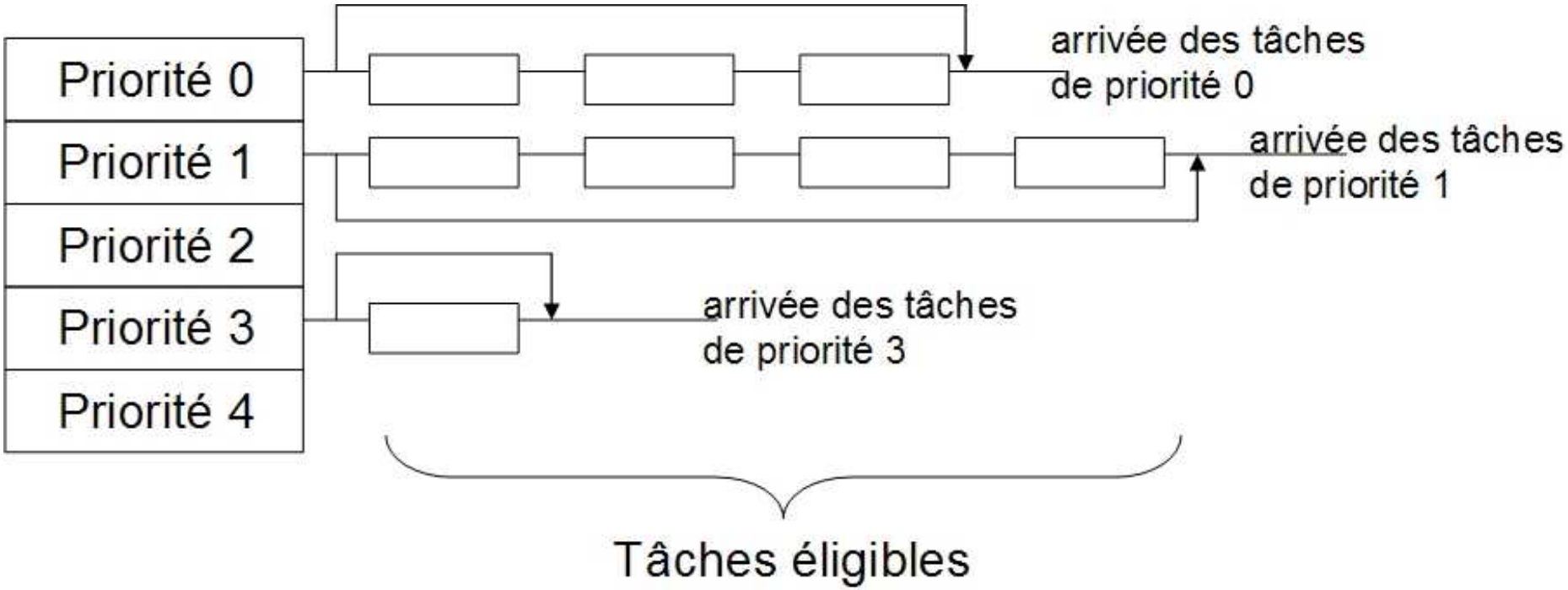
Definition

A scheduling policy $\mathcal{P}$ is *optimal* iff: if a task set is schedulable by some policy, then it is schedulable by $\mathcal{P}$.

Warning: Optimality is defined with respect to a certain class of policies, e.g. RM is optimal for fixed-task priority, on-line, preemptive, ...

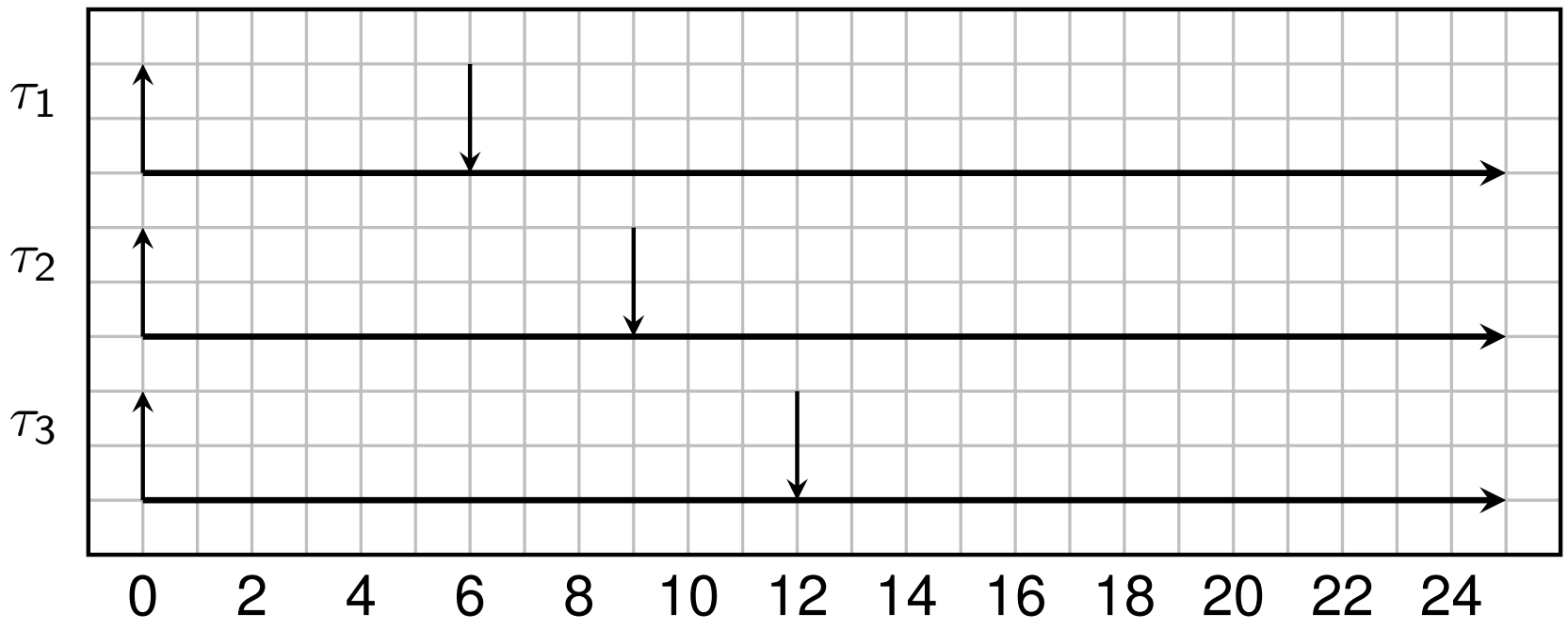
Reminder: priority-based scheduling

- Each task has a priority;
- The task with the highest priority is scheduled first;
- Several tasks can share the same priority ⇒ Round-robin/FIFO for each priority level;
- RT policies are usually derived from this general policy.



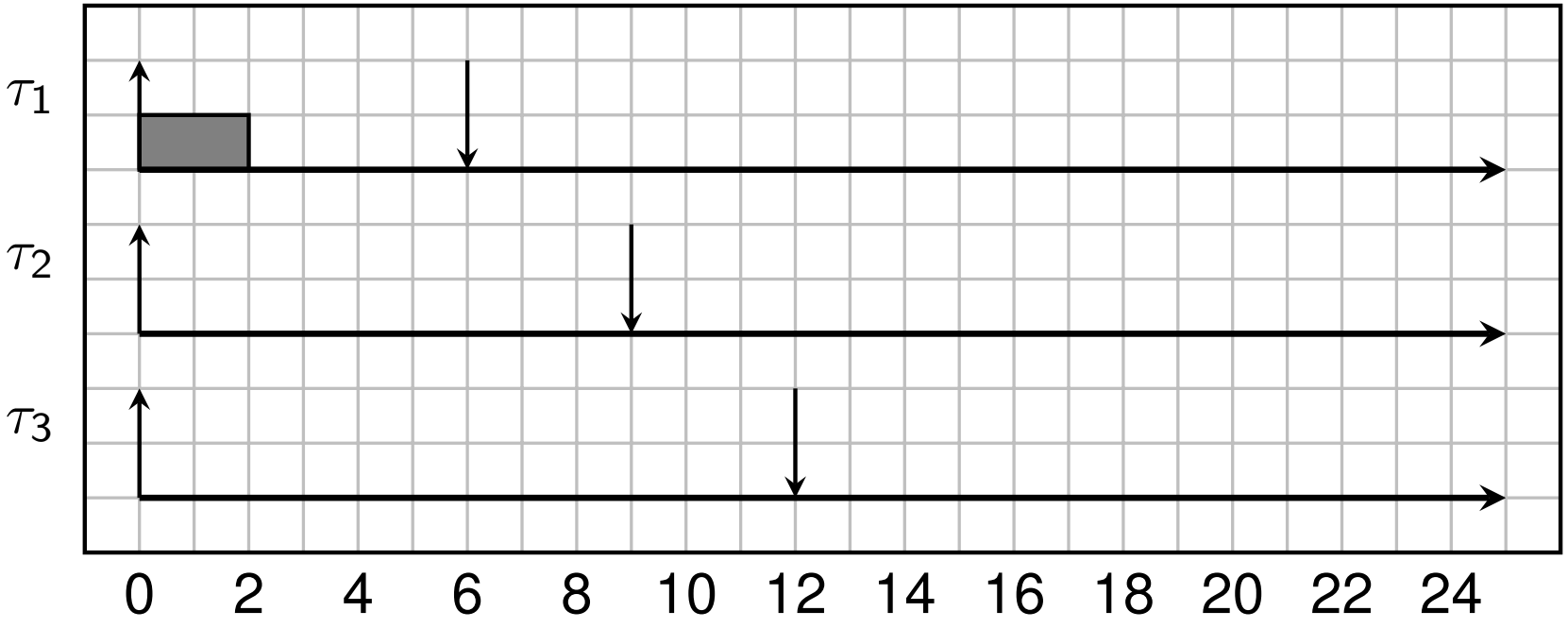
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



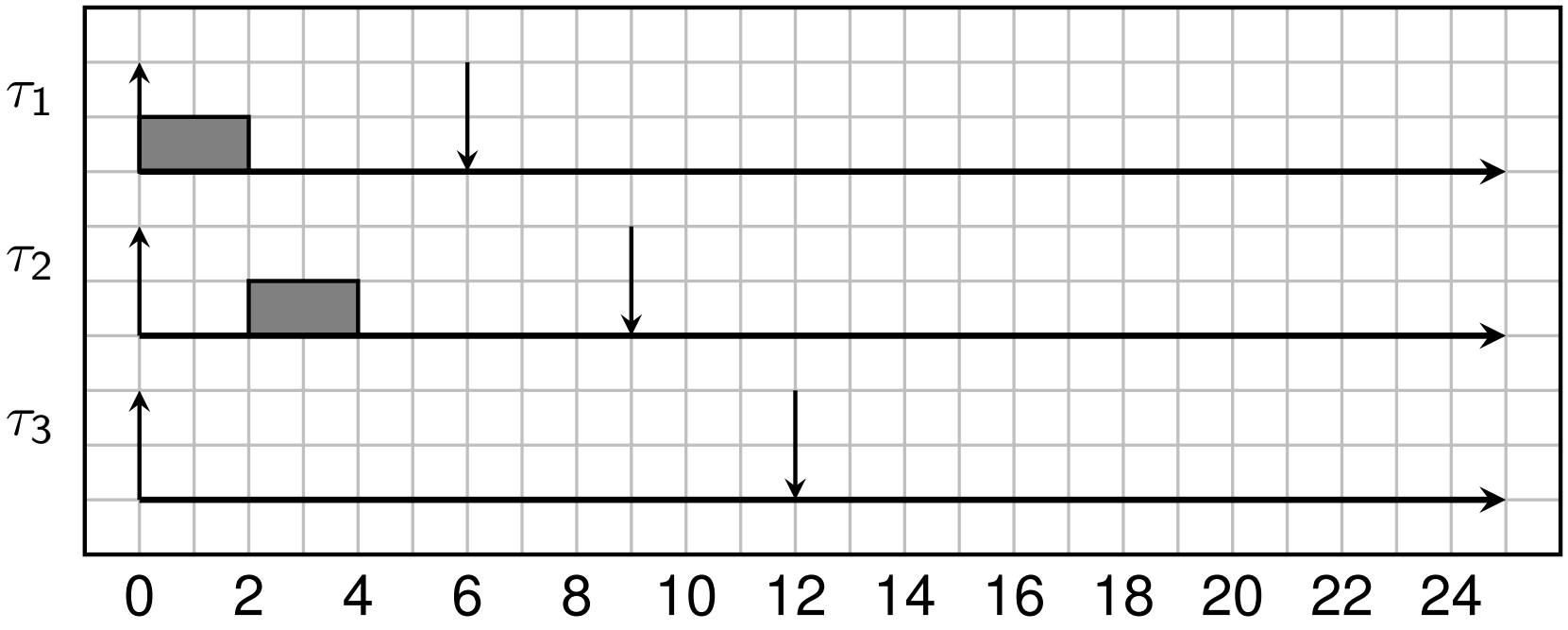
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



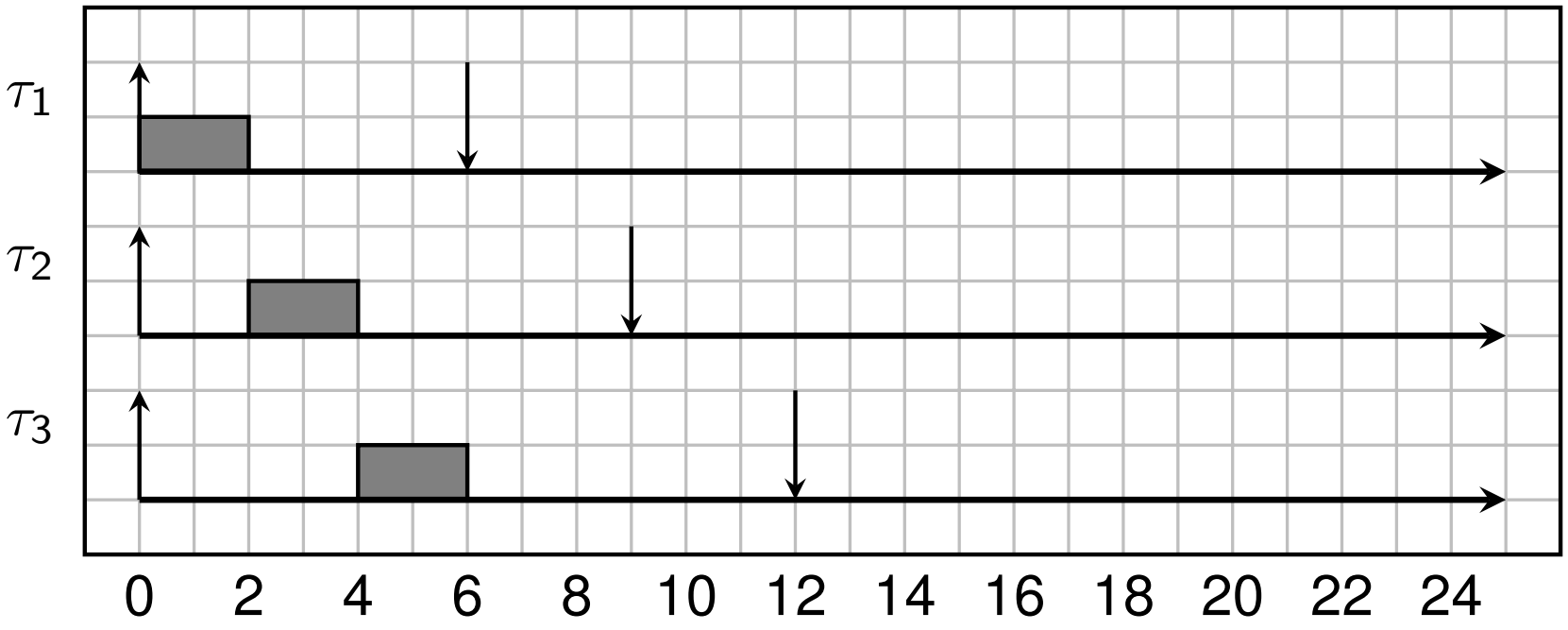
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



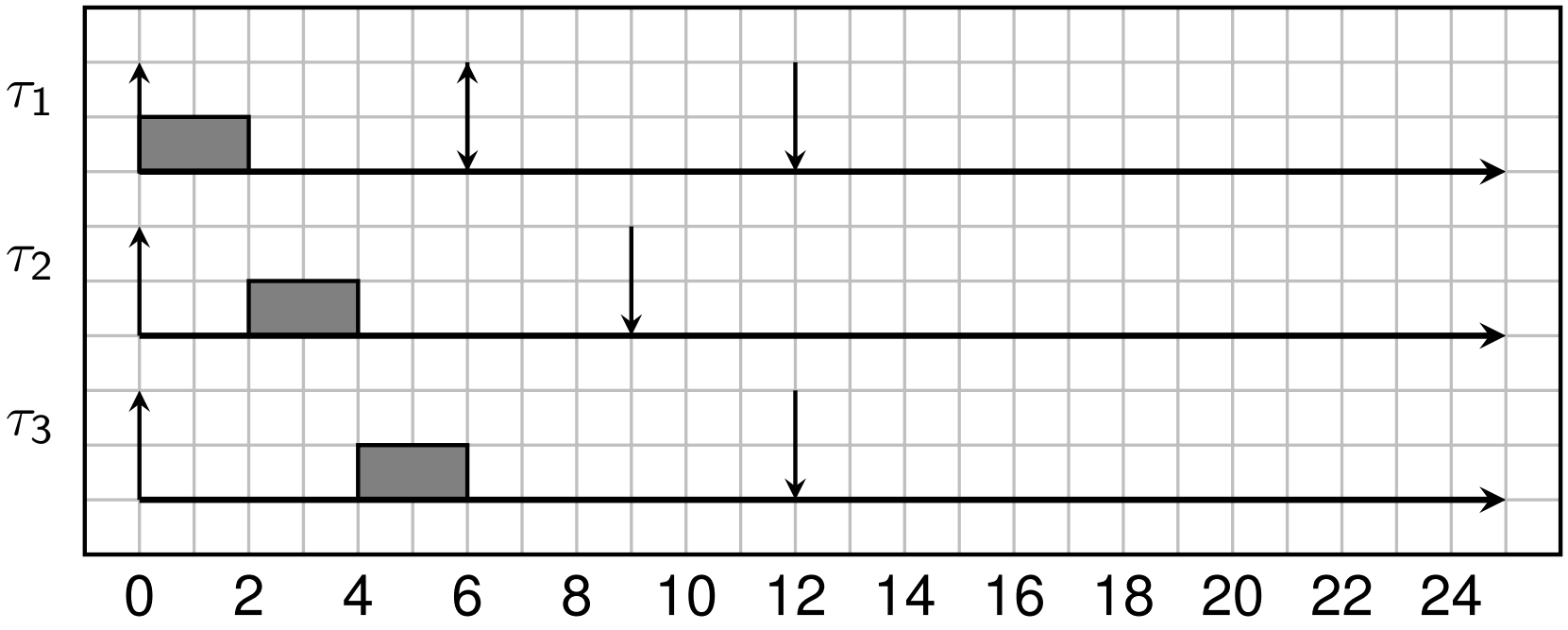
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



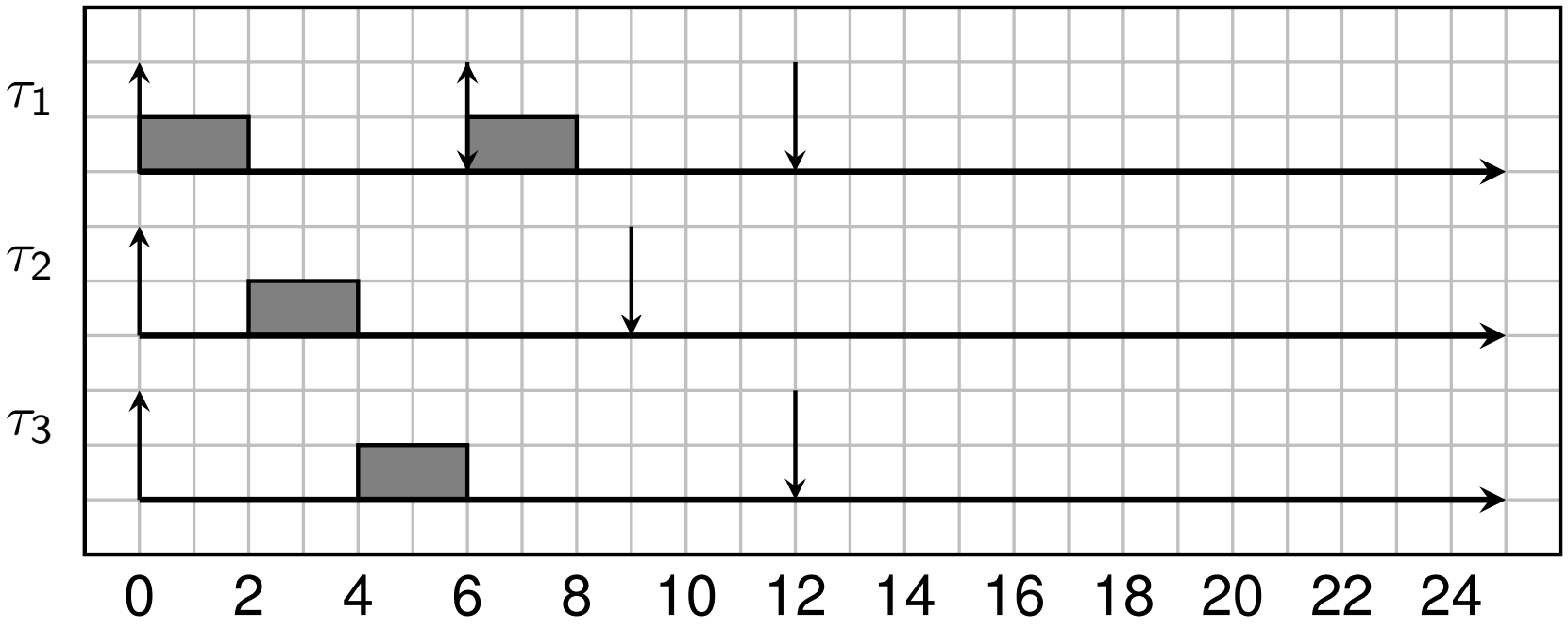
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



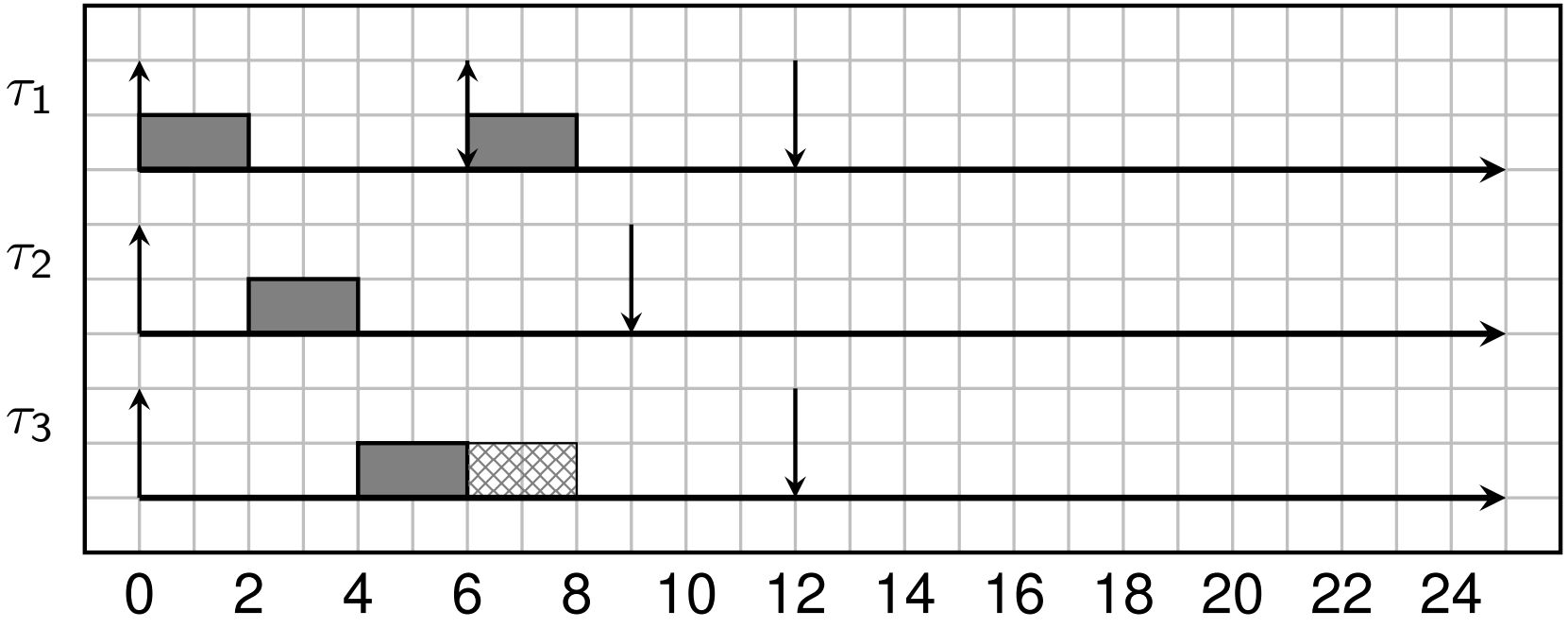
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



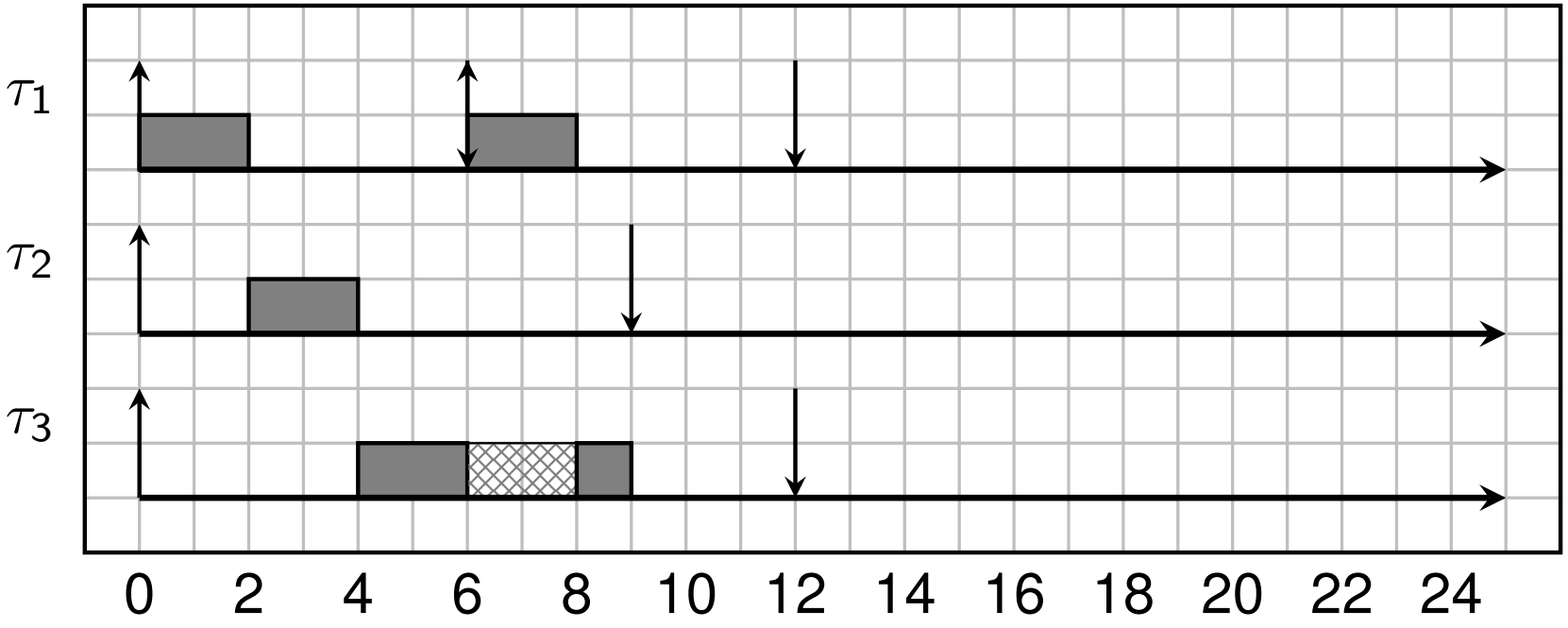
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



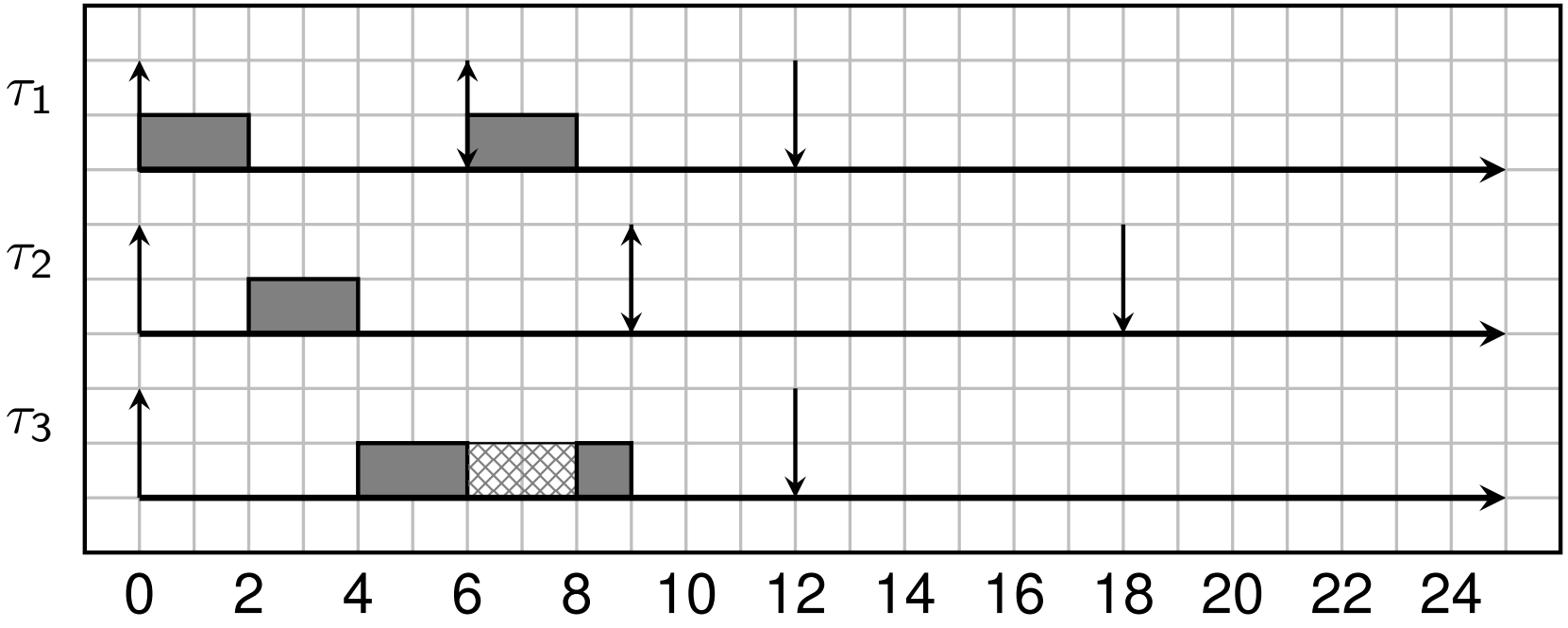
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



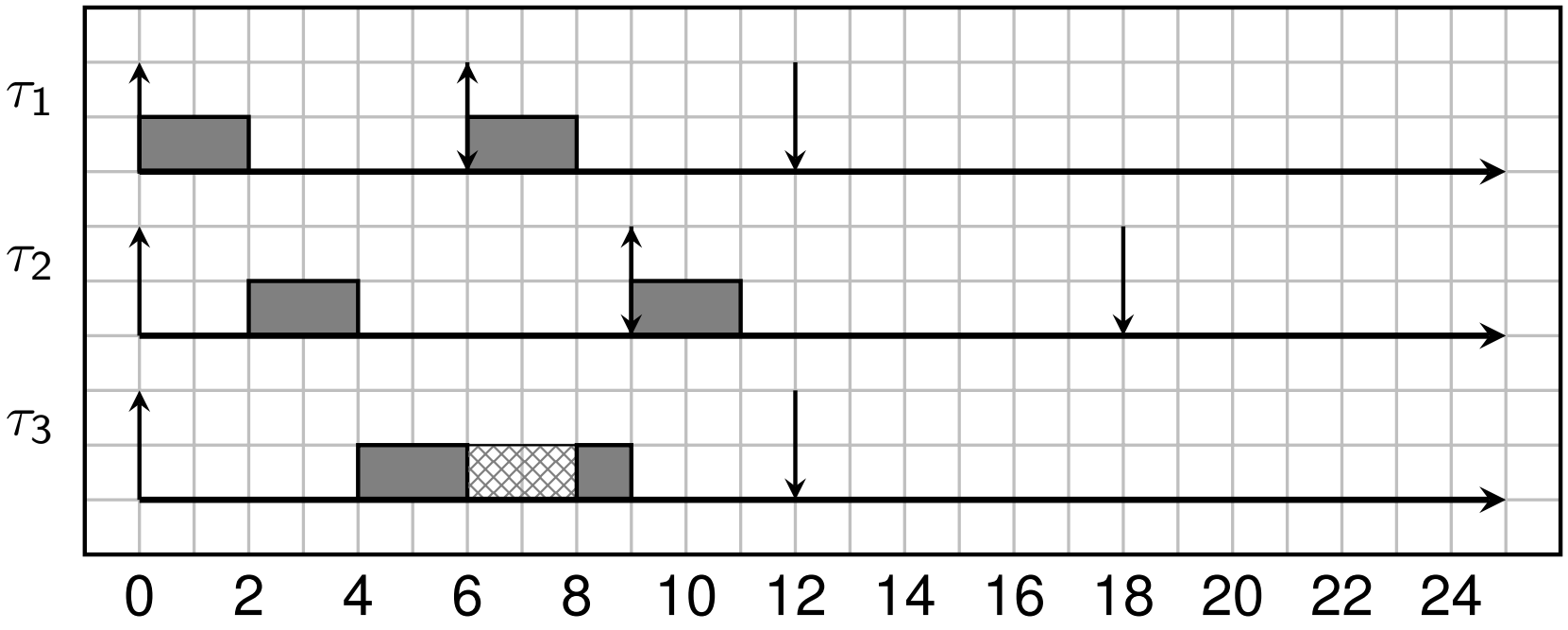
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



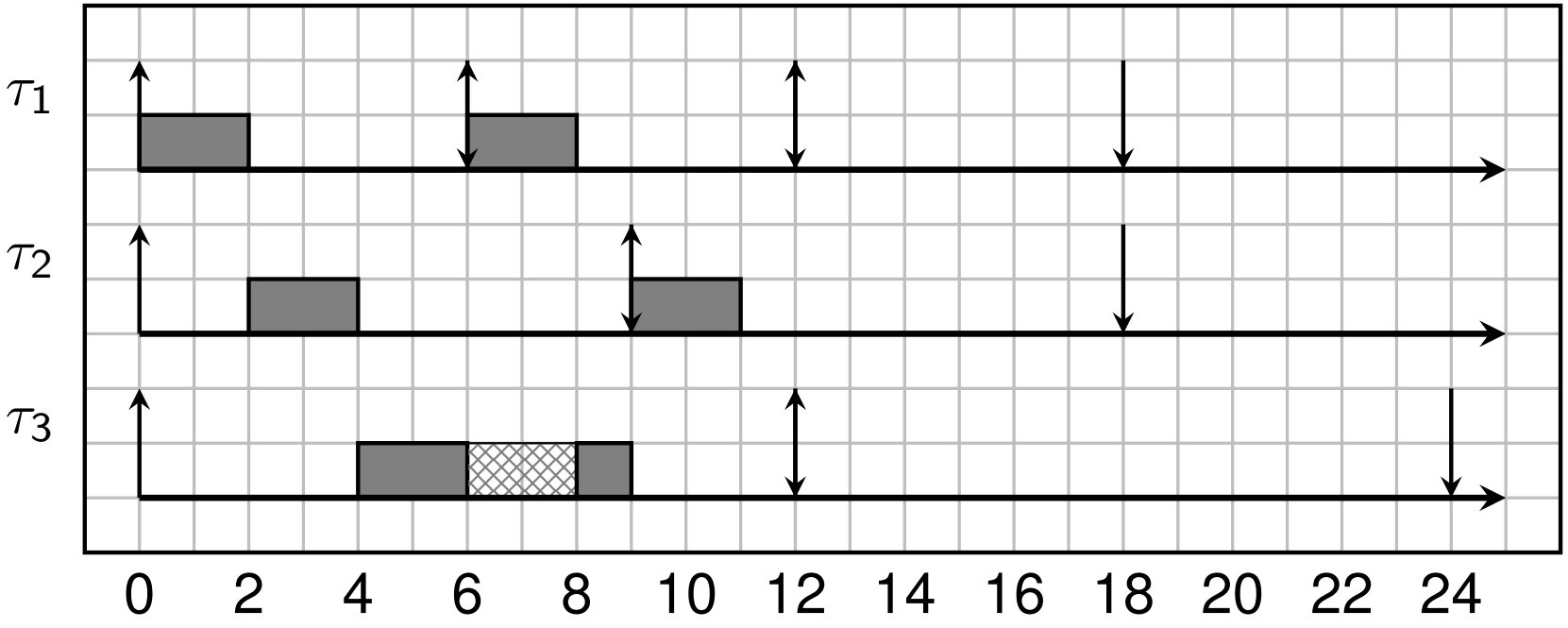
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



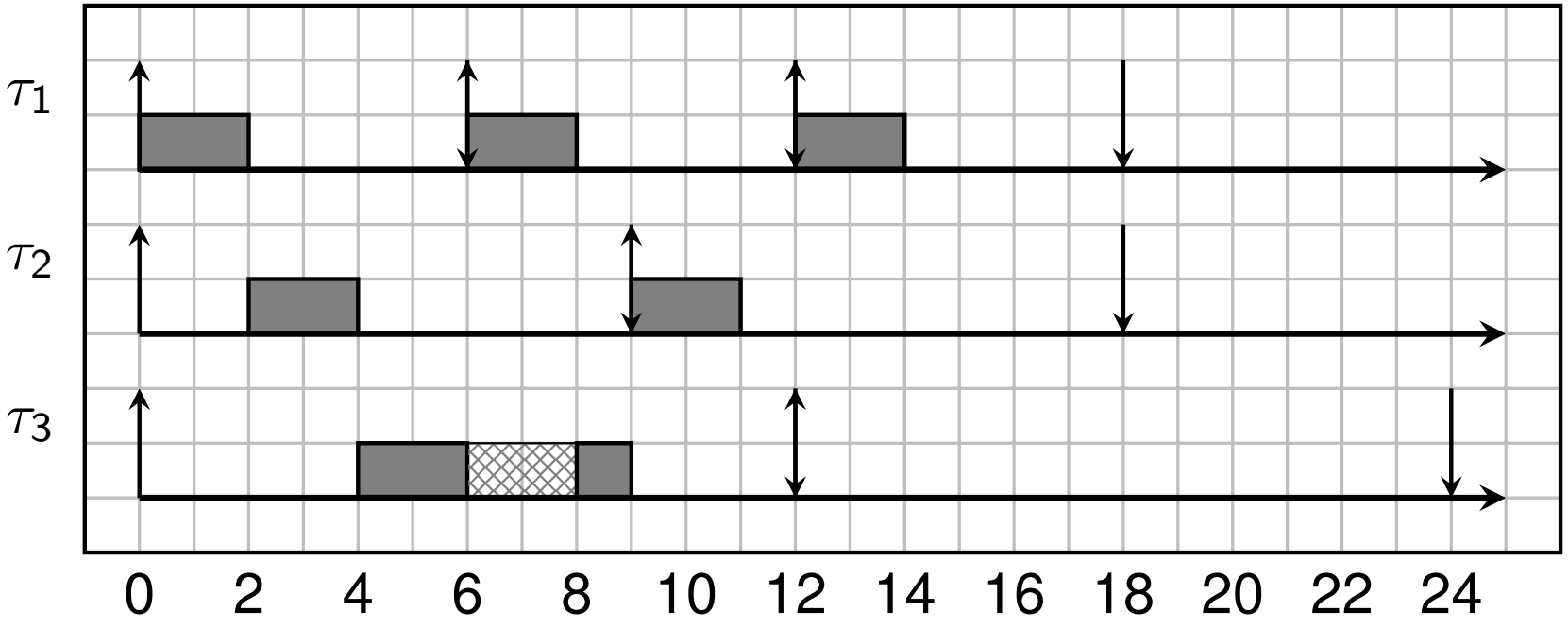
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



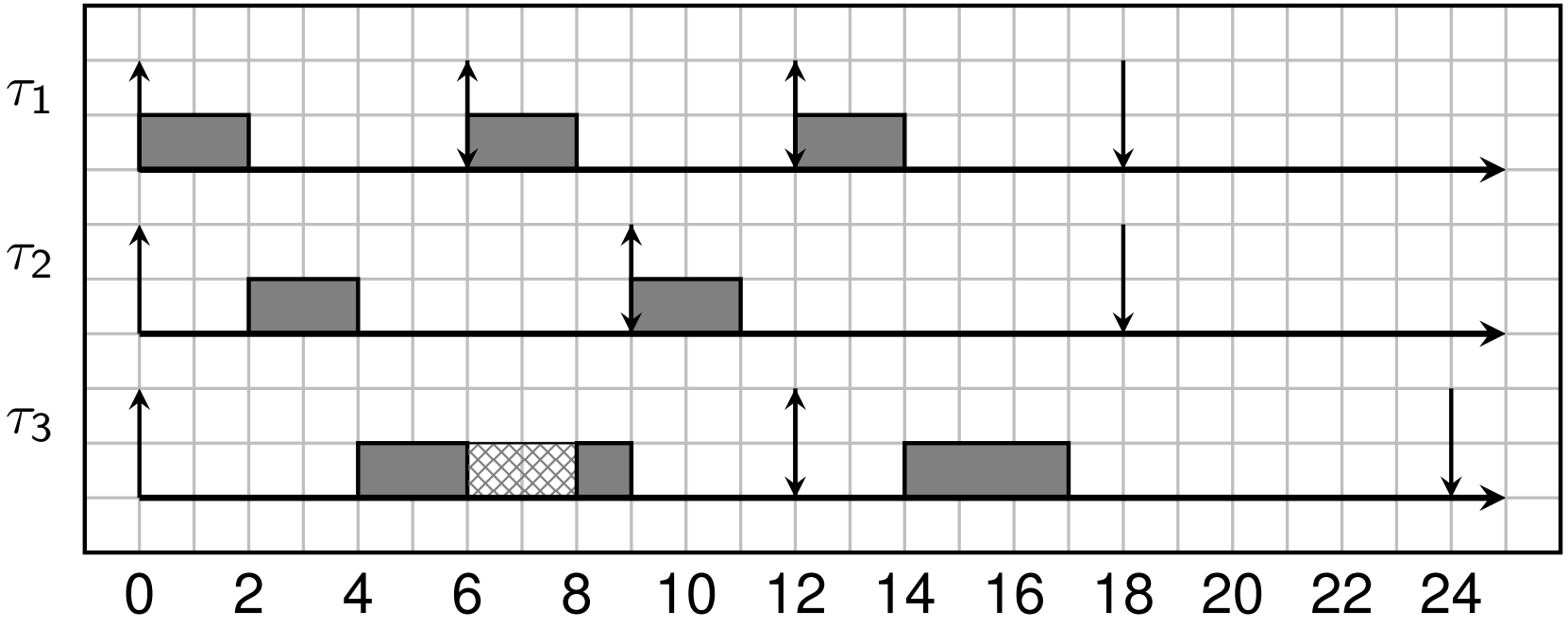
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



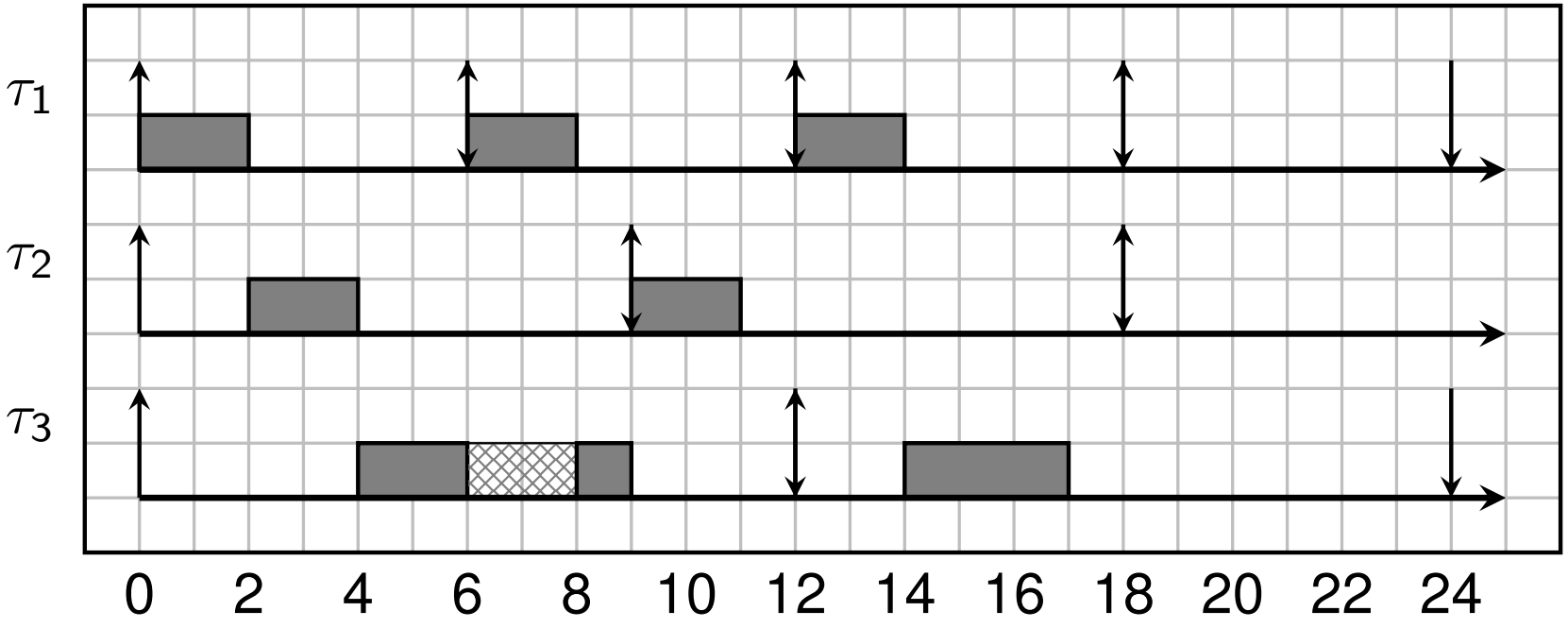
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



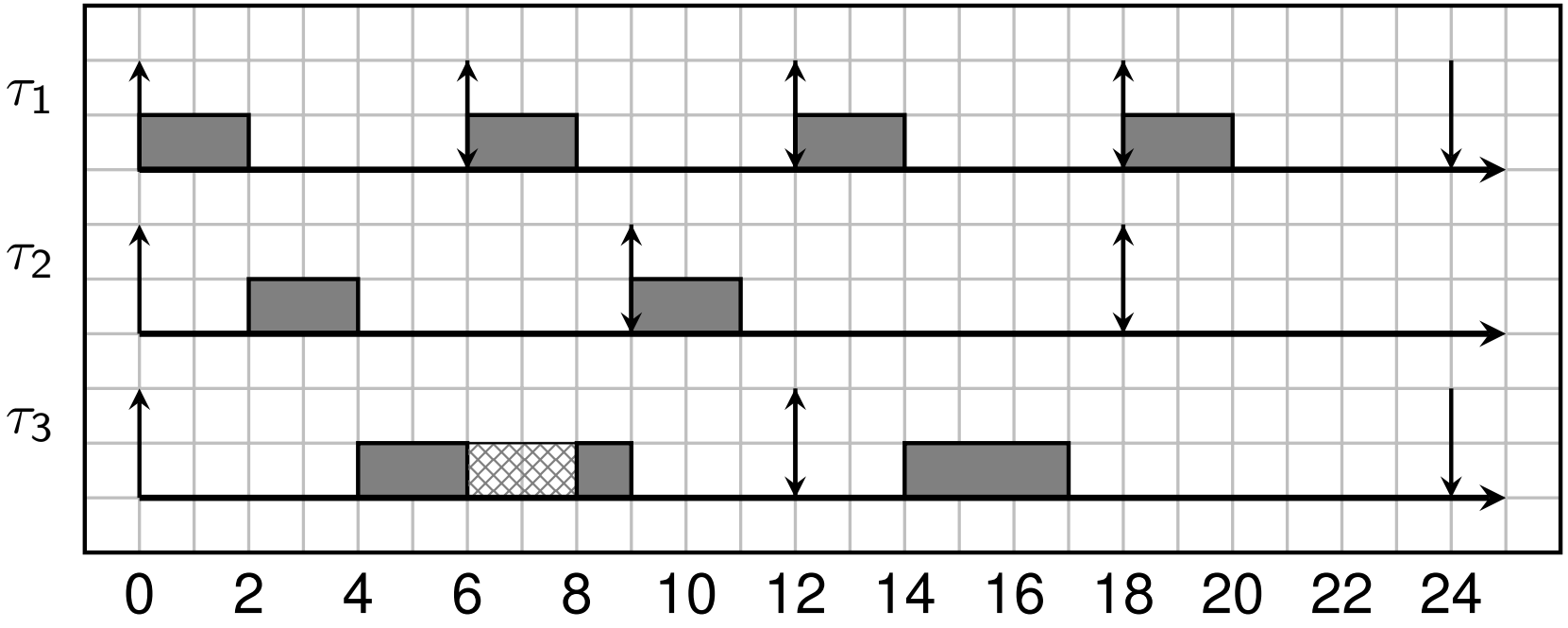
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



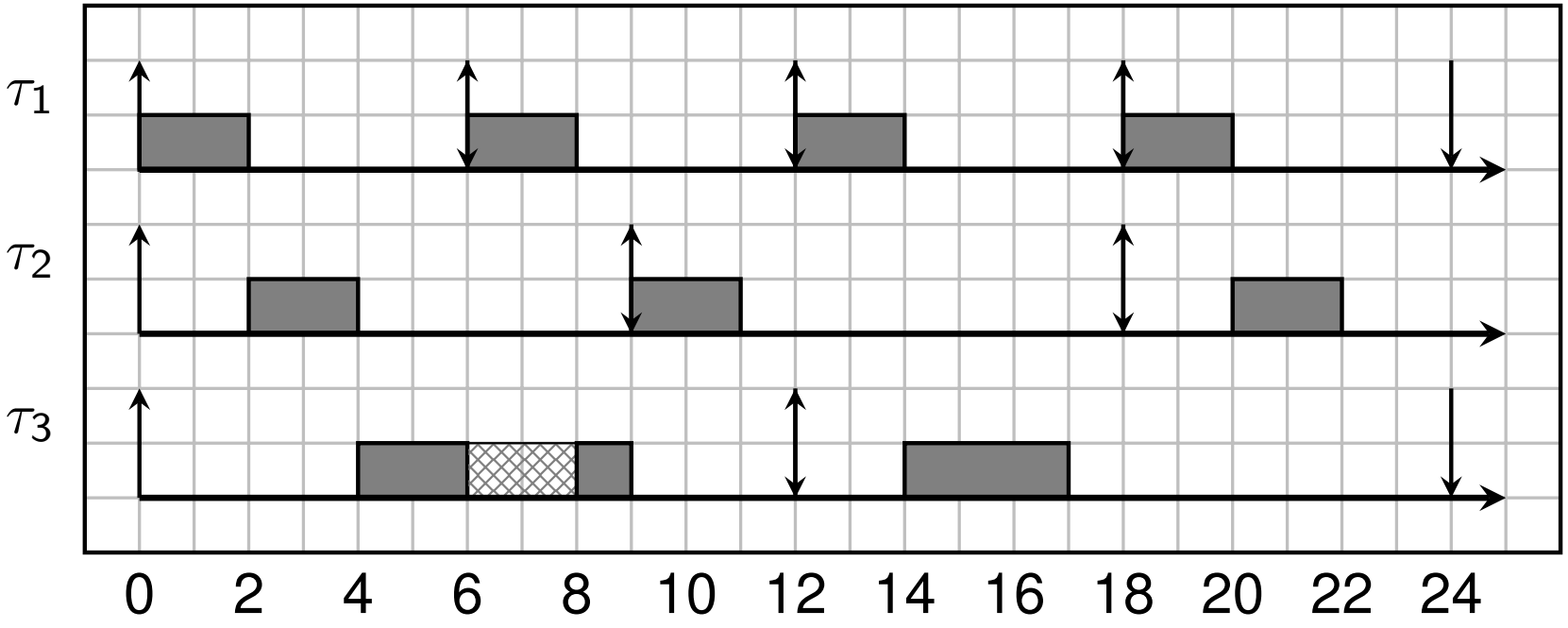
Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



Example

- Let p_i denote the priority of task τ_i;
- Let $\tau_1 = (2, 6, 6)$, $\tau_2 = (2, 9, 9)$, $\tau_3 = (3, 12, 12)$;
- Let $p_1 = 3$ (highest), $p_2 = 2$, $p_3 = 1$.



Outline

1. Real-time model
2. Scheduling
3. Fixed-task priority
4. Fixed-job priority
5. Shared resources
6. Data-dependencies
7. Conclusion

Fixed-task priority policies

- Fixed-task priority: the priority of a task remains the same for the whole execution;
- When $O_i = 0$ and $D_i = T_i$:
  - *Rate Monotonic* is an optimal policy;
  - Shorter period $\Rightarrow$ higher priority.
- When $O_i = 0$ and $D_i \leq T_i$:
  - *Deadline Monotonic* is an optimal policy;
  - Shorter relative deadline $\Rightarrow$ higher priority.
- These policies are very simple to implement;
- In most RTOS, priority is assigned by the programmer, at task creation.

Schedulability analysis: Simulation

- Since $O_i = 0$ and all tasks are periodic, the same schedule is repeated every *hyperperiod*;
- Hyperperiod: $HP = lcm(\{T_i\})$;

$\Rightarrow$ Check schedulability by simulating the execution until date HP.

Example

Compare the hyperperiod of these task sets:

- $T_1 = 8,\ T_2 = 12,\ T_3 = 24$

Schedulability analysis: Simulation

- Since $O_i = 0$ and all tasks are periodic, the same schedule is repeated every *hyperperiod*;
- Hyperperiod: $HP = lcm(\{T_i\})$;

$\Rightarrow$ Check schedulability by simulating the execution until date HP.

Example

Compare the hyperperiod of these task sets:

- $T_1 = 8,\ T_2 = 12,\ T_3 = 24 \Rightarrow HP = 24$

Schedulability analysis: Simulation

- Since $O_i = 0$ and all tasks are periodic, the same schedule is repeated every *hyperperiod*;
- Hyperperiod: $HP = lcm(\{T_i\})$;

$\Rightarrow$ Check schedulability by simulating the execution until date HP.

Example

Compare the hyperperiod of these task sets:

- $T_1 = 8,\ T_2 = 12,\ T_3 = 24 \Rightarrow HP = 24$
- $T_1 = 7,\ T_2 = 12,\ T_3 = 25$

Schedulability analysis: Simulation

- Since $O_i = 0$ and all tasks are periodic, the same schedule is repeated every *hyperperiod*;
- Hyperperiod: $HP = lcm(\{T_i\})$;

$\Rightarrow$ Check schedulability by simulating the execution until date HP.

Example

Compare the hyperperiod of these task sets:

- $T_1 = 8,\ T_2 = 12,\ T_3 = 24 \Rightarrow HP = 24$
- $T_1 = 7,\ T_2 = 12,\ T_3 = 25 \Rightarrow HP = 2100$

Schedulability analysis: Simulation

- Since $O_i = 0$ and all tasks are periodic, the same schedule is repeated every *hyperperiod*;
- Hyperperiod: $HP = lcm(\{T_i\})$;

$\Rightarrow$ Check schedulability by simulating the execution until date HP.

Example

Compare the hyperperiod of these task sets:

- $T_1 = 8,\ T_2 = 12,\ T_3 = 24 \Rightarrow HP = 24$
- $T_1 = 7,\ T_2 = 12,\ T_3 = 25 \Rightarrow HP = 2100$

- Worst-case: tasks have co-prime periods $\Rightarrow$ HP is *exponential* w.r.t. number of tasks;
- Practical case: tasks have harmonic periods $\Rightarrow$ HP is equal to highest period.

Schedulability analysis: Utilization Bound

- Let $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ the *Utilization Ratio*;
- The *utilization bound* U_{lub} is a value such that: $U \leq U_{lub} \Rightarrow$ feasible.

Theorem, Liu& Layland 1973

Consider n periodic (or sporadic) tasks with $D_i = T_i$, whose priorities are assigned in RM order. Then:

$$U_{lub} = n(2^{1/n} - 1)$$

- U_{lub} decreases with n;
- For $n = 2$, $U_{lub} = 0.83$, for large n, $U_{lub} \simeq 0.69$.

Schedulability analysis: Utilization Bound

- Let $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ the *Utilization Ratio*;
- The *utilization bound* U_{lub} is a value such that: $U \leq U_{lub} \Rightarrow$ feasible.

Theorem, Liu& Layland 1973

Consider n periodic (or sporadic) tasks with $D_i = T_i$, whose priorities are assigned in RM order. Then:

$$U_{lub} = n(2^{1/n} - 1)$$

- U_{lub} decreases with n;
- For $n = 2$, $U_{lub} = 0.83$, for large n, $U_{lub} \simeq 0.69$.

Warning:

- This test is sufficient, but not necessary!

Schedulability analysis: Utilization Bound

- Let $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ the *Utilization Ratio*;
- The *utilization bound* U_{lub} is a value such that: $U \leq U_{lub} \Rightarrow$ feasible.

Theorem, Liu& Layland 1973

Consider n periodic (or sporadic) tasks with $D_i = T_i$, whose priorities are assigned in RM order. Then:

$$U_{lub} = n(2^{1/n} - 1)$$

- U_{lub} decreases with n;
- For $n = 2$, $U_{lub} = 0.83$, for large n, $U_{lub} \simeq 0.69$.

Warning:

- This test is sufficient, but not necessary!
- It is very simple to find schedulable task sets with $U_i = 1$;
- Experiments show that the actual average bound is around 88%.

Schedulability analysis: Response Time Analysis

- Response-time: duration between job release and job completion;
- Main idea: check that worst-case response-time is lower than deadline for every task;
- RTA is a *necessary* and *sufficient* schedulability test;
- RTA does not require to explore the complete hyperperiod.

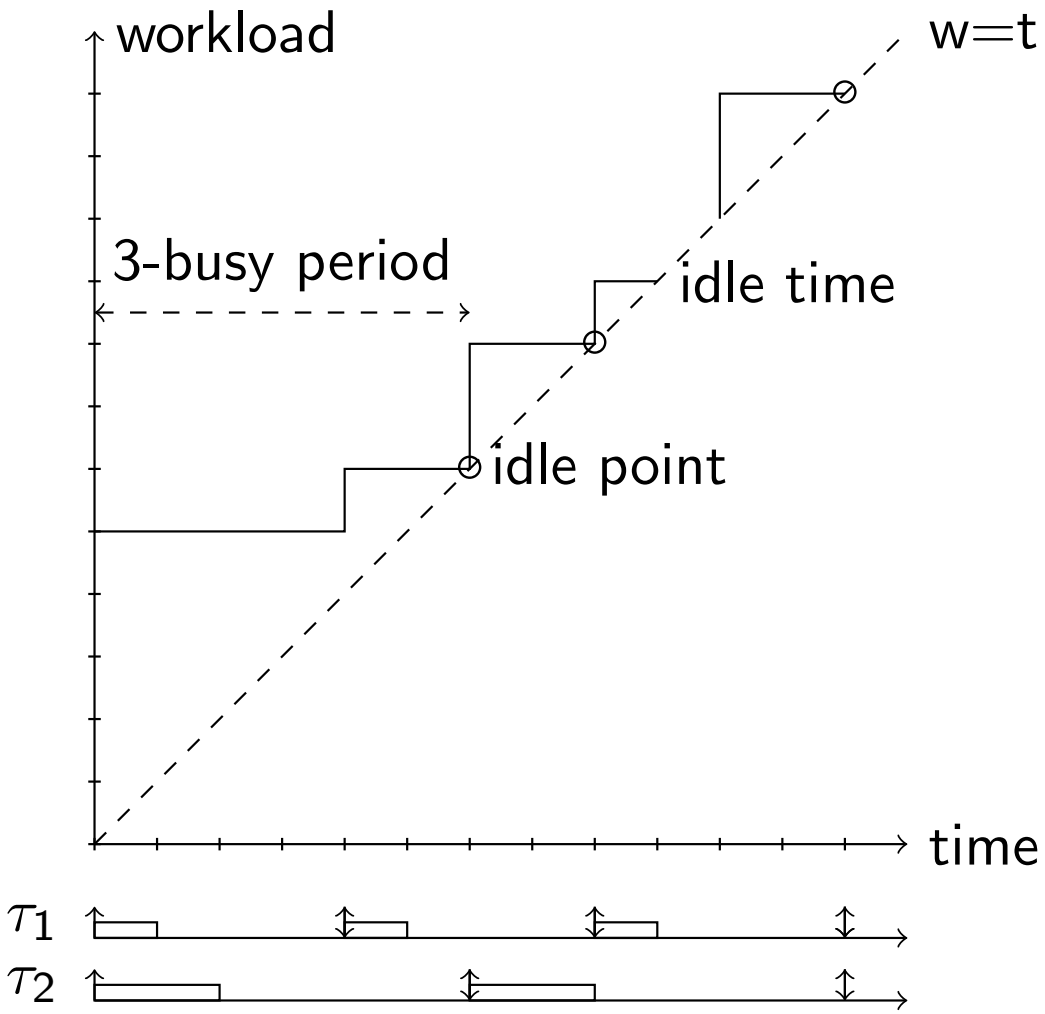
Theorem

The worst-case response time of a task of priority P_k is equal to the longest k-busy period.

Busy-period

Definition

k-busy period: processor executes tasks of priority higher than k.



Computing an $i-$busy period

- Inductive definition (Pseudo-polynomial):

$$B_{i,0} = C_i$$

$$B_{i,k+1} = C_i + \sum_{j \in hp(i)} \lceil \frac{B_{i,k}}{T_i} \rceil C_j$$

- $hp(i)$ tasks with priority higher than i;
- Stop when:
  - Reaching a fixed-point, i.e. $B_{i,k+1} = B_{i,k}$;
  - $B_{i,k} > D_i$.

Computing an $i-$busy period

- Inductive definition (Pseudo-polynomial):

$$B_{i,0} = C_i$$

$$B_{i,k+1} = C_i + \sum_{j \in hp(i)} \lceil \frac{B_{i,k}}{T_i} \rceil C_j$$

- $hp(i)$ tasks with priority higher than i;
- Stop when:
  - Reaching a fixed-point, i.e. $B_{i,k+1} = B_{i,k}$;
  - $B_{i,k} > D_i$.

NB:

- This works for any optimal fixed-priority policy;
- The scheduling policy only impacts the definition of $hp(i)$;
- We can use this test for RM and DM.

RTA for tasks with offset ($O_i \neq 0$)

- *Critical instant*: a date where all tasks are released simultaneously;
- *The longest busy period is initiated by the critical instant*;
- When $O_i = 0$, a critical instant occurs at date 0;
- This lies at the heart of RTA for case without offset;
- When $O_i \neq 0$, the critical instant may not exist, so RTA requires to study all busy periods up to $max(O_i) + 2HP$.

Audsley's scheduling policy ($O_i \neq 0$)

- *DM is not optimal when* $O_i \neq 0$;
- Audsley proposed a different policy for that case:

for Each priority *lvl*, lowest first **do**
  assigned $\leftarrow$ *false*
  for Each unassigned task τ_k **do**
    if τ_k schedulable with priority *lvl* **then**
      $P_k \leftarrow$ *lvl*; *assigned* $\leftarrow$ *true*
      break the current loop
    end if
  end for
  if assigned=false **then** the system is not feasible
  end if
end for

Audsley's scheduling policy

- When testing the schedulability of τ at level *lvl*:
  - The relative priorities of higher priority tasks does not matter;
  - We can compute the *interference* of higher priority tasks similarly to how we computed the busy period

  $\Rightarrow$ There is no need for an actual schedule.
- The policy performs $n(n+1)/2$ "partial" schedulability tests;
- This is both a scheduling policy and a schedulability test;
- This general principle has been applied to several other scheduling problems (see OPA, Optimal Priority Assignment).

Outline

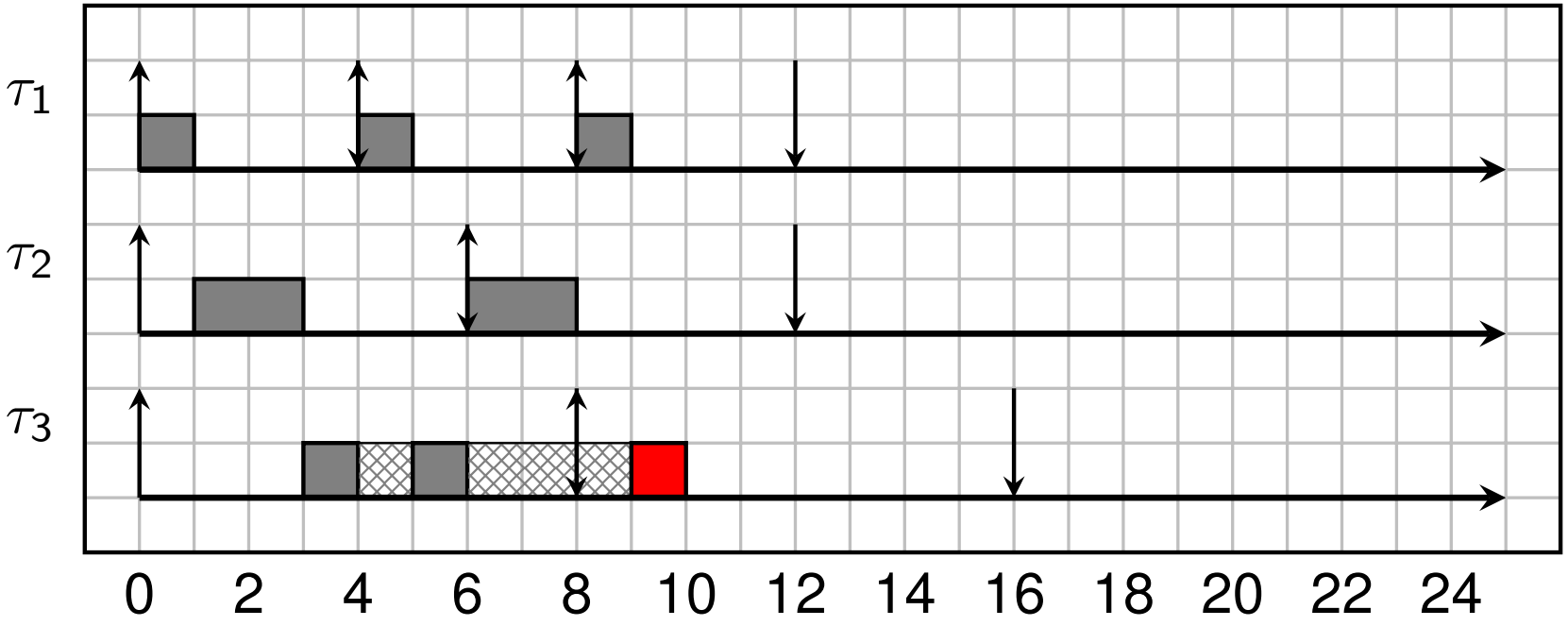
1. Real-time model
2. Scheduling
3. Fixed-task priority
4. Fixed-job priority
5. Shared resources
6. Data-dependencies
7. Conclusion

Fixed-job priority policies

- Fixed-job priority:
  - The priority of a job remains unchained during its execution;
  - The priorities of two jobs of the same task can differ.
- *Earliest-Deadline-First* is an optimal policy for a large class of problems: $O_i \geq 0$ and $D_i \leq T_i$;
- The *job* with the shortest absolute deadline has the highest priority;
- In a RTOS:
  - Priorities are assigned to tasks (not jobs);
  - So, EDF requires to change dynamically task priorities;
  - Not all RTOS allow this.

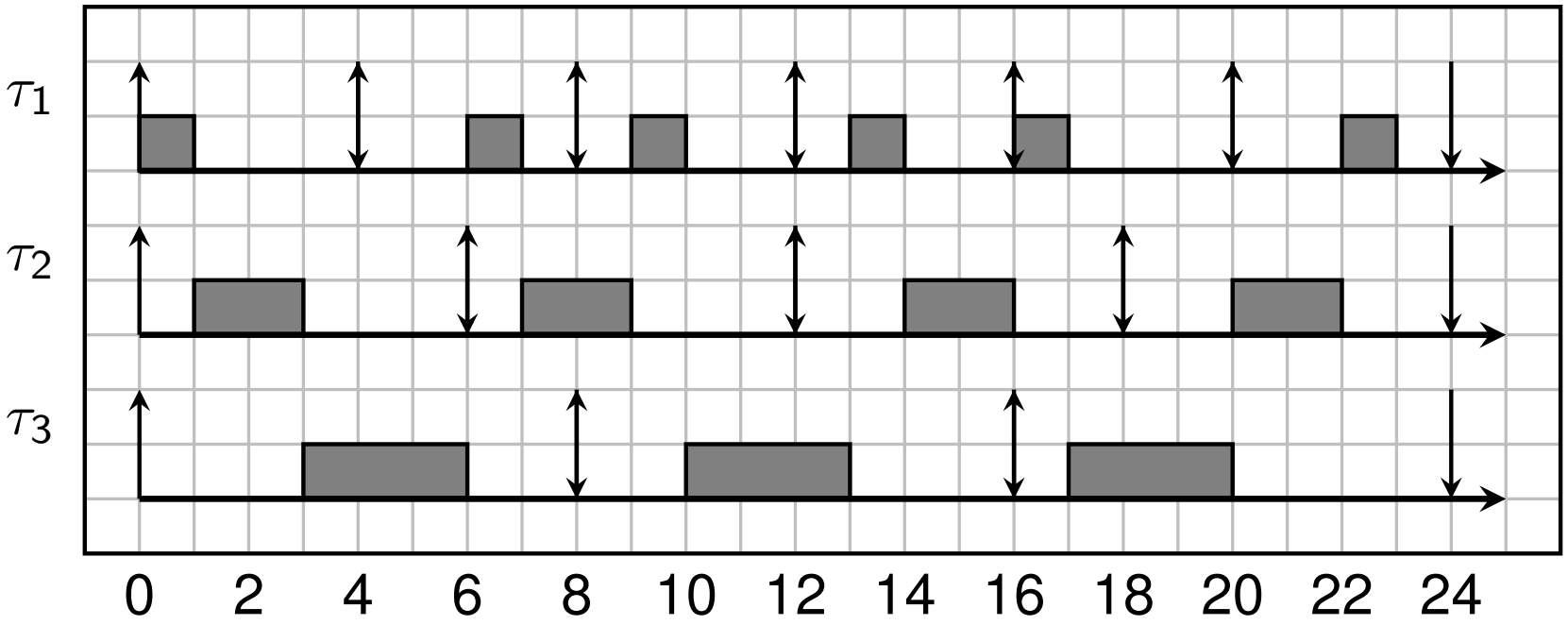
Example, with RM

- $\tau_1 = (1, 4)$, $\tau_2 = (2, 6)$, $\tau_3 = (3, 8)$;
- Utilization is quite high: $U = \frac{1}{4} + \frac{2}{6} + \frac{3}{8} = \frac{23}{24}$
- There is a deadline miss!



Same example, with EDF

- $\tau_1 = (1, 4)$, $\tau_2 = (2, 6)$, $\tau_3 = (3, 8)$;
- The task set is schedulable.



Utilization bound

When $D_i = T_i$, the system is schedulable with EDF iff:

$$U \leq 1$$

- Extremely simple and efficient test;
- EDF enables higher utilization ratio than RM.

Processor demand

When $D_i \leq T_i$ and $O_i = 0$:

- The worst-case response time does *not* occur at the critical instant;
- Let $h(t)$ denote the amount of work to complete before time t (*processor demand*):

$$h(t) = \sum_{i=1}^{n} \lfloor \frac{t + T_i - D_i}{T_i} \rfloor C_i$$

- Task set is schedulable iff:

$$\forall t, h(t) \leq t$$

- No need to study the processor demand for all t:
  1. $h(t)$ only increases at job deadlines;
  2. We can find a bound on the values of t that must be checked.

Efficient processor demand analysis

- Two known bounds:
  - L_a:
$$L_a = max\{D_1, \ldots, D_n, \frac{\sum_{i=1}^{n}(T_i - D_i)C_i/T_i}{1-U}\}$$
  - L_b computed as the least fixed-point of:
$$w^0 = \sum_{i=1}^{n} C_i$$
$$w^{j+1} = \sum_{i=1}^{n} \lceil \frac{w^j}{T_i} \rceil C_i$$

$\Rightarrow$ Check that $h(t) \leq t$ for each deadline until $min(L_a, L_b)$.

Fixed-job vs fixed-task priorities

- *Implementation*: DM/RM are easier to implement than EDF;
- *Run-time overhead*:
  - EDF requires to compute priorities at run-time;
  - EDF causes fewer preemptions.
- *Task jitter (response time variation)*:
  - RM/DM cause low jitter on high priority tasks;
  - EDF causes lower average jitter.
- *Overload*: When $U \geq 1$, EDF causes more deadline misses than RM/DM;
- *Processor utilization*: EDF enables higher processor utilization (up to 100%).

Outline

1. Real-time model
2. Scheduling
3. Fixed-task priority
4. Fixed-job priority
5. Shared resources
6. Data-dependencies
7. Conclusion

Shared resource pitfalls

- So far, we considered *independent tasks*;
- In reality, tasks often share *resources* (e.g. data);
- A piece of code accessing the shared resource is called a *critical section*;
- Two or more critical sections on the same resource must be executed in mutual exclusion;
- Each shared resource should be protected by a mutual exclusion mechanism: semaphore, mutex, etc.
- Mutual exclusion mechanisms are a notable source of bugs in computer systems:

Shared resource pitfalls

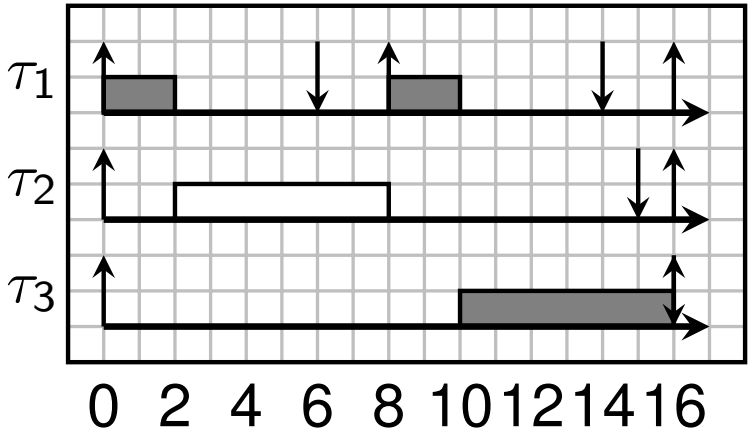
- So far, we considered *independent tasks*;
- In reality, tasks often share *resources* (e.g. data);
- A piece of code accessing the shared resource is called a *critical section*;
- Two or more critical sections on the same resource must be executed in mutual exclusion;
- Each shared resource should be protected by a mutual exclusion mechanism: semaphore, mutex, etc.
- Mutual exclusion mechanisms are a notable source of bugs in computer systems: deadlocks, starvation, ...

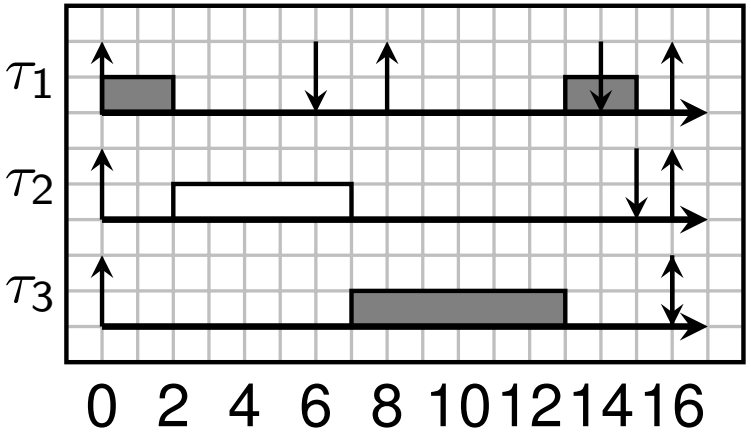
Shared resource pitfalls

- So far, we considered *independent tasks*;
- In reality, tasks often share *resources* (e.g. data);
- A piece of code accessing the shared resource is called a *critical section*;
- Two or more critical sections on the same resource must be executed in mutual exclusion;
- Each shared resource should be protected by a mutual exclusion mechanism: semaphore, mutex, etc.
- Mutual exclusion mechanisms are a notable source of bugs in computer systems: deadlocks, starvation, ...
- Real-time systems introduce their own nasty bugs related to scheduling:
  - Priority inversion;
  - Scheduling anomalies.

Scheduling anomaly

- $\{\tau_1(2,8,6), \tau_2(6,16,15), \tau_3(6,16,16)\}$
- Tasks in grey share resources;
- On the left: schedulability analysis succeeds;
- On the right: τ_2 takes less than WCET, causing a deadline miss.

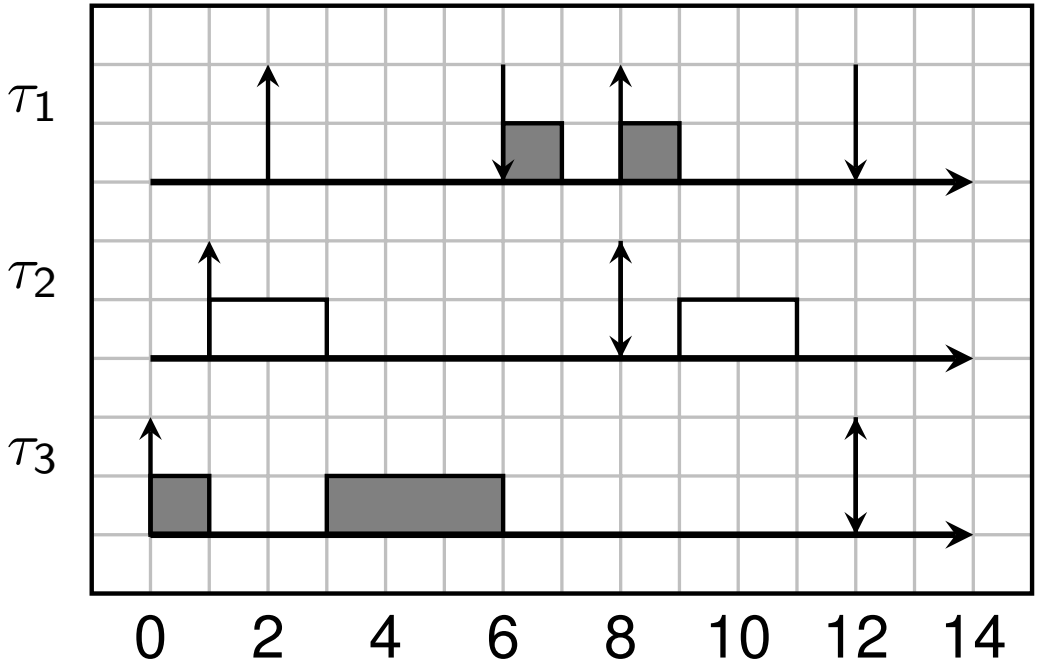




Priority inversion

- Tasks in grey share resources;
- τ_2 is running while the highest priority task τ_1 is waiting for τ_3 to complete its critical section;

$\Rightarrow$ This causes τ_1 to miss its deadline.



Priority Inheritance Protocol

- Principle: When a task accesses a shared resource, it inherits the priority of the highest priority task that shares this resource until the end of its critical section;

⇒ This completely prevents priority inversion;

- Several variations:
  - Priority Inheritance Protocol (the original): high blocking times;
  - Priority Ceiling Protocol: lower blocking time, complex implementation (priority inheritance), unnecessary context switches;
  - Stack Resource Policy a.k.a. Immediate Priority Ceiling: resolves previous drawbacks.

Stack Resource Policy: preemption level

The *preemption level* π_i of a task τ_i is:

- For RM/DM: equal to the priority of τ_i ($\pi_i = p_i$);
- For EDF: the inverse of the relative deadline ($\pi_i = 1/D_i$).

Stack Resource Policy: resource ceiling

Definition

The *resource ceiling* of a resource S_k is the highest preemption level of the tasks accessing it (at any time).

$$ceil(S_k) = \max_i \{\pi_i | \tau_i \text{ uses } S_k\}$$

Definition

The *system ceiling* at a given time is the maximum of the resource ceiling of resources locked at that time.

$$\Pi(t) = \max_k \{ceil(S_k) | S_k \text{ is locked at } t\}$$

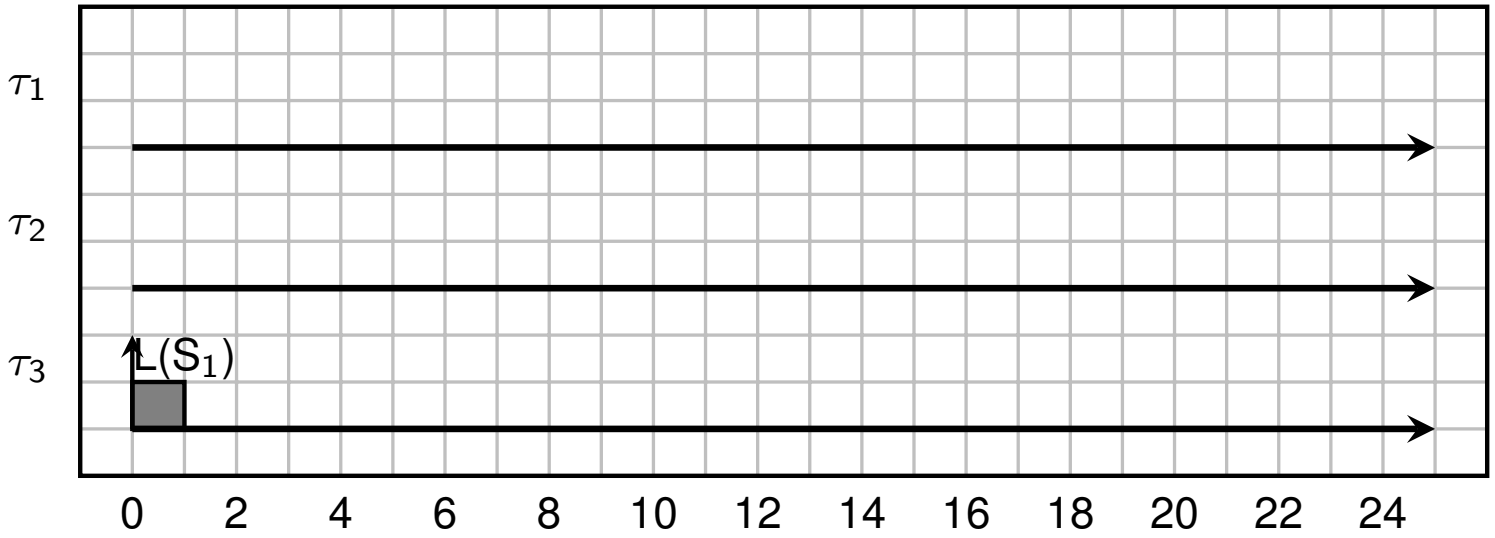
Stack Resource Policy: the protocol

At time t, task τ_i can start executing iff:

- It is the highest priority task;
- Its preemption level is greater than the current system ceiling: $\pi_i > \Pi(t)$

Stack Resource Policy: example

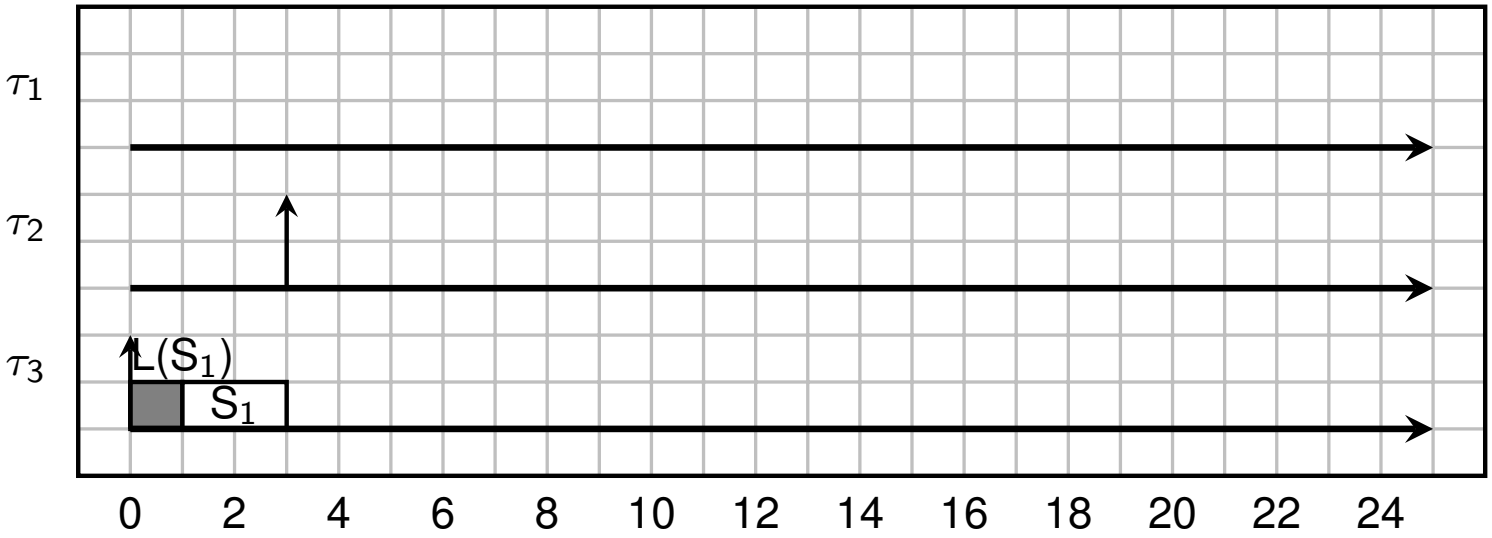
- $p_1 > p_2 > p_3$;
- S_1 used by $\tau_1, \tau_3 \Rightarrow ceil(S_1) = p_1$;
- S_2 used by $\tau_1, \tau_2 \Rightarrow ceil(S_2) = p_1$.



- Task τ_3 raises the sys ceiling Π to p_1

Stack Resource Policy: example

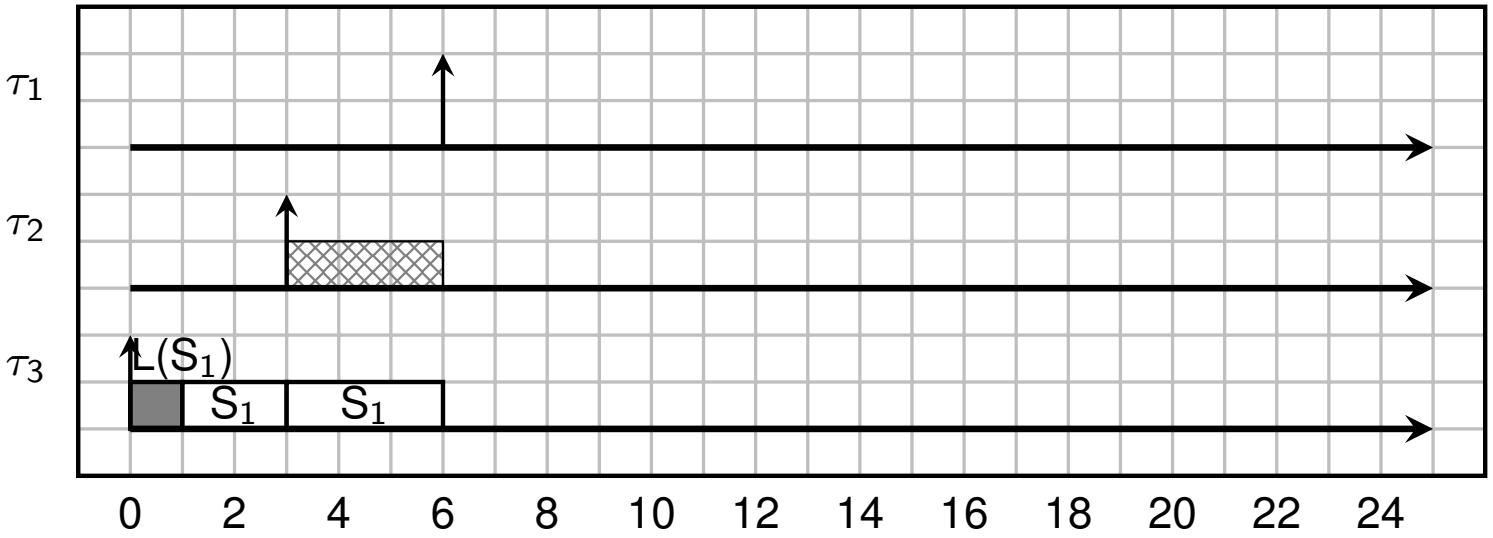
- $p_1 > p_2 > p_3$;
- S_1 used by $\tau_1, \tau_3 \Rightarrow ceil(S_1) = p_1$;
- S_2 used by $\tau_1, \tau_2 \Rightarrow ceil(S_2) = p_1$.



- Task τ_3 raises the sys ceiling Π to p_1
- Task τ_2 cannot start because $p_2 < \Pi = p_1$

Stack Resource Policy: example

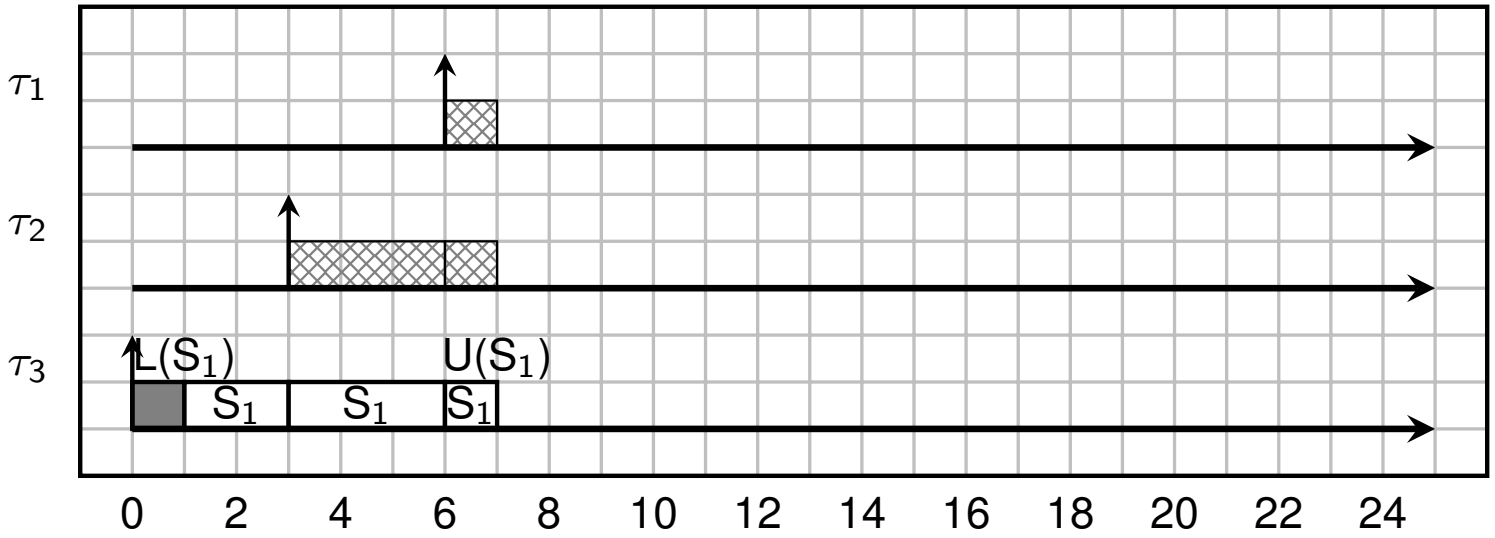
- $p_1 > p_2 > p_3$;
- S_1 used by $\tau_1, \tau_3 \Rightarrow ceil(S_1) = p_1$;
- S_2 used by $\tau_1, \tau_2 \Rightarrow ceil(S_2) = p_1$.



- Task τ_3 raises the sys ceiling Π to p_1
- Task τ_2 cannot start because $p_2 < \Pi = p_1$
- Task τ_1 cannot start because $p_1 = \Pi = p_1$

Stack Resource Policy: example

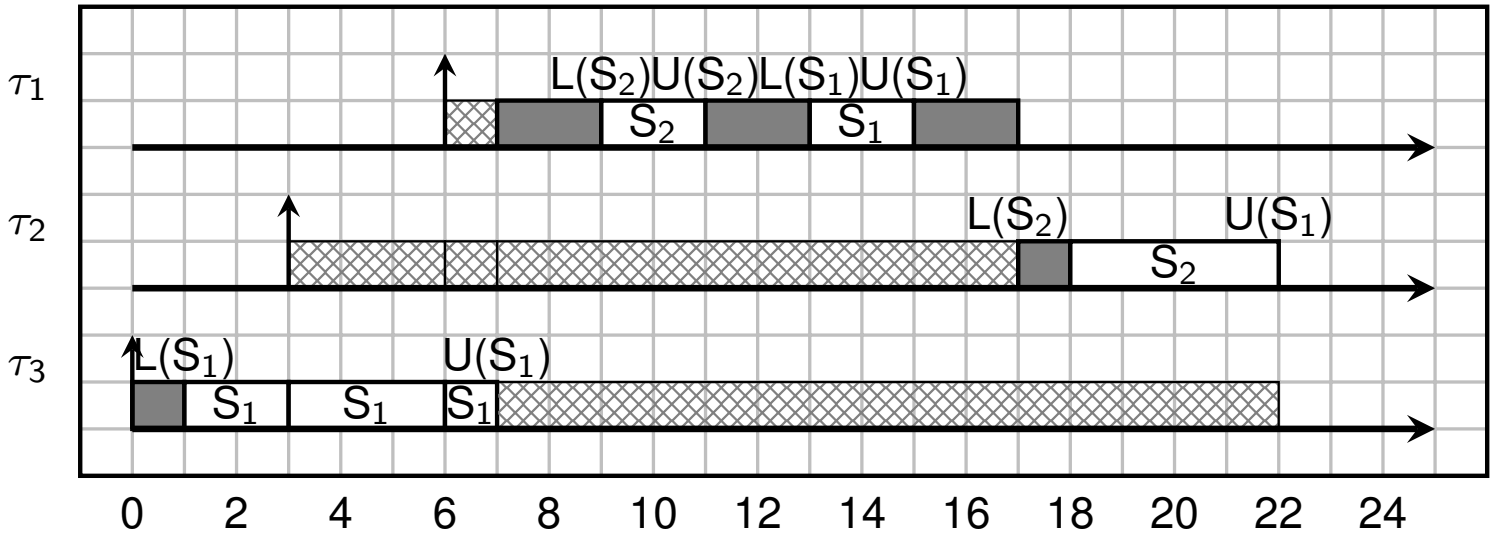
- $p_1 > p_2 > p_3$;
- S_1 used by $\tau_1, \tau_3 \Rightarrow ceil(S_1) = p_1$;
- S_2 used by $\tau_1, \tau_2 \Rightarrow ceil(S_2) = p_1$.



- Task τ_3 raises the sys ceiling Π to p_1
- Task τ_2 cannot start because $p_2 < \Pi = p_1$
- Task τ_1 cannot start because $p_1 = \Pi = p_1$
- When task τ_3 unlocks, the sys ceiling goes down, and all other tasks can start executing

Stack Resource Policy: example

- $p_1 > p_2 > p_3$;
- S_1 used by $\tau_1, \tau_3 \Rightarrow ceil(S_1) = p_1$;
- S_2 used by $\tau_1, \tau_2 \Rightarrow ceil(S_2) = p_1$.



- Task τ_3 raises the sys ceiling Π to p_1
- Task τ_2 cannot start because $p_2 < \Pi = p_1$
- Task τ_1 cannot start because $p_1 = \Pi = p_1$
- When task τ_3 unlocks, the sys ceiling goes down, and all other tasks can start executing

Stack Resource Policy: properties

SRP has several useful properties:

- A task can be blocked (waiting for a resource) at most once, at the begining of its execution;
- There are no deadlocks;
- Simple implementation (no priority inheritance).

Outline

1. Real-time model
2. Scheduling
3. Fixed-task priority
4. Fixed-job priority
5. Shared resources
6. Data-dependencies
7. Conclusion

Data communications

- Most common shared resource in a real-time system: *shared data*;
- Shared data: a task (called *consumer*) requires data produced by another task (called *producer*) to perform its computations;
- *Register-based communication*:
  - Consume the last value produced;
  - Semantics varies depending on execution order;
  - No impact on task order;
  - Often used in the automotive domain.
- *Causal communication*:
  - Producer must end before start of consumer;
  - Deterministic semantics;
  - Used in avionics systems;
  - $\Rightarrow$ Introduces precedence constraints.

Precedence constraints

- Representation: A Directed Acyclic Graph, with vertex=task, edge=precedence (why a DAG?)

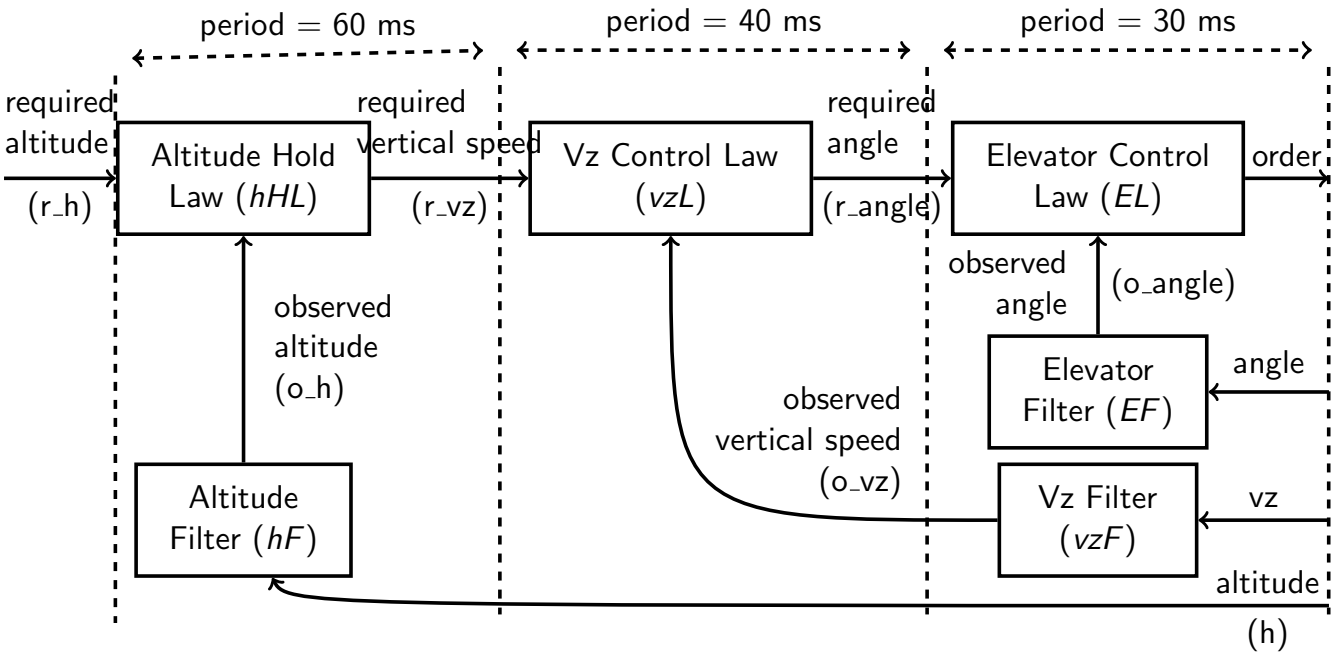
Precedence constraints

- Representation: A Directed Acyclic Graph, with vertex=task, edge=precedence (why a DAG?);
- $\tau_i \rightarrow \tau_j$: τ_i must end before τ_j starts;
- When $T_i = T_j$: simple precedence constraint, $\tau_{i.p} \rightarrow \tau_{j.q}$ for all p, q;
- When $T_i \neq T_j$: extended precedence constraint, not all $\tau_{i.p} \rightarrow \tau_{j.q}$ *does not hold for all* p, q.

Extended precedence constraints: example

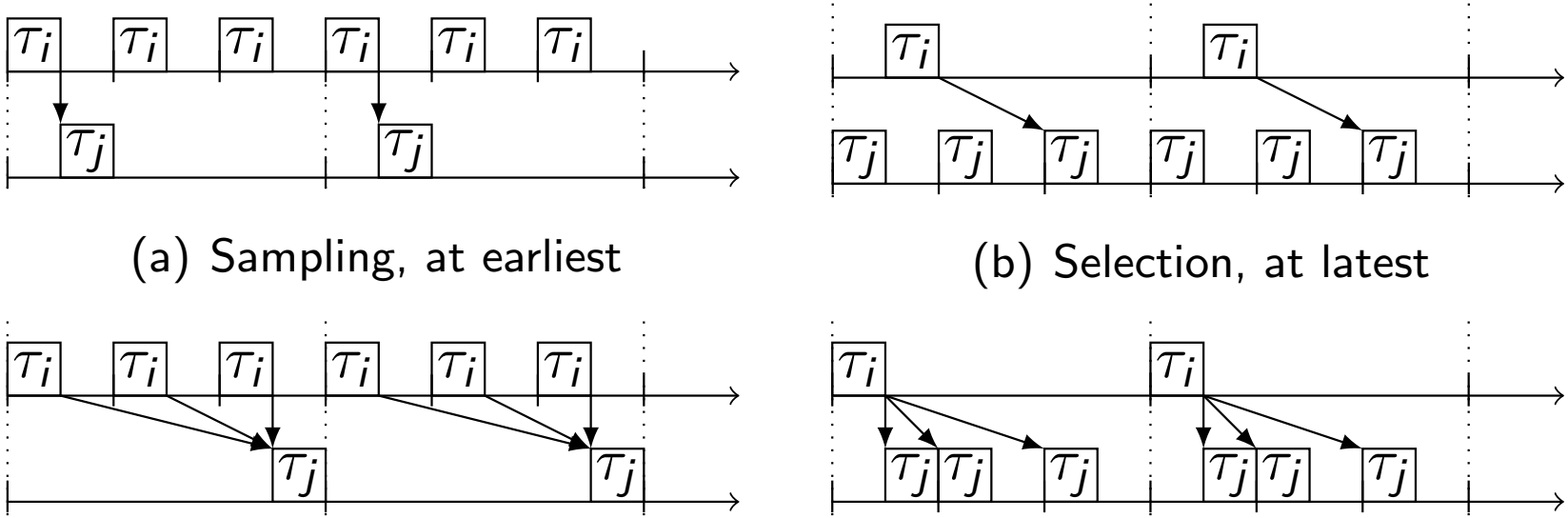
Longitudinal flight control system of an aircraft:

- Inputs: current surface angle, current vertical speed, current altitude, altitude required by the pilot;
- Outputs: control surfaces;
- Several multi-rate computation chains.



Some communication patterns

- Extended precedence constraints usually follow repetitive patterns;
- Repetition period$=HP$.



(a) Sampling, at earliest

(b) Selection, at latest

(c) Array gathering

(d) Array scattering

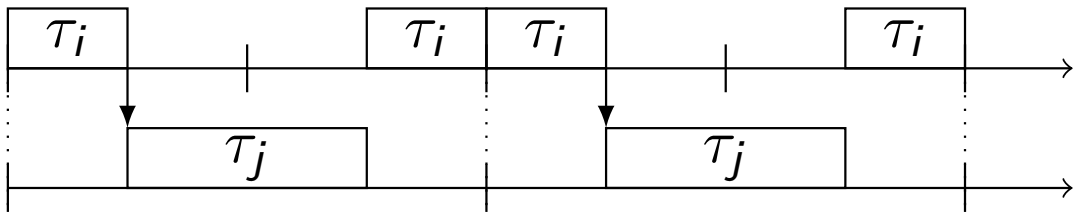
Precedence encoding, simple precedence constraints

A simple clever technique to schedule dependent tasks:

1. Encode $\tau_i \rightarrow \tau_j$ in real-time attributes as follows:
   - $D_i^* = min(D_i, \min_{\tau_j \in succs(\tau_i)}(D_j^* - C_j))$
     - $\Rightarrow$ $P_i > P_j$;
       - Anyway, τ_i must end early enough for τ_j to end before D_j;
   - $O_j^* = max(O_j, \max_{\tau_i \in preds(\tau_j)}(O_i^*))$
     - $\Rightarrow$ τ_j cannot start before τ_i is released.
2. Resulting problem $\Leftrightarrow$ Original problem;
3. Schedule encoded task set with a deadline-based scheduler;

$\Rightarrow$ *Optimal* policy with DM/EDF.

Precedence encoding, extended precedence constraints

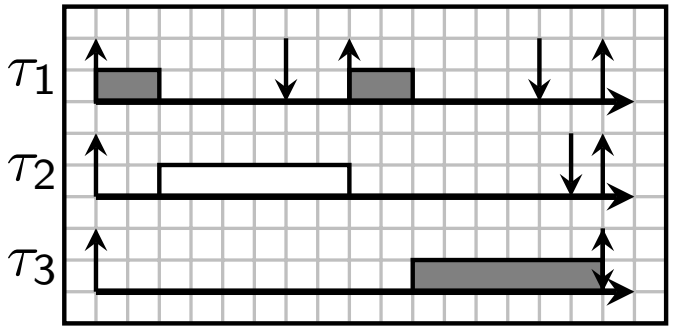


- $\mathcal{S}_1 = \{(O_i = 0, C_i = 2, D_i = T_i, T_i = 4),$
  $(O_j = 0, C_j = 4, D_j = 6, T_j = 8)\}$
- $\tau_i \rightarrow \tau_j$;
- $D^*_{i.n} = 2$ for $n \in \mathbb{N}$: not schedulable (EDF);
- $D^*_{i.n} = 2$ for $n \in 2\mathbb{N}$ and $D^*_{i.n} = 4$ for $n \in 2\mathbb{N} + 1$: schedulable (EDF);

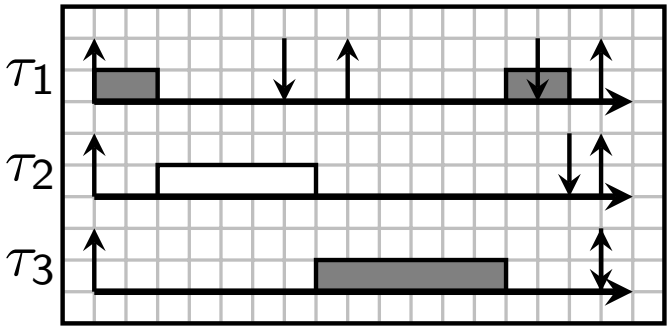
$\Rightarrow$ Compute (periodic) job deadline patterns.

Scheduling anomalies

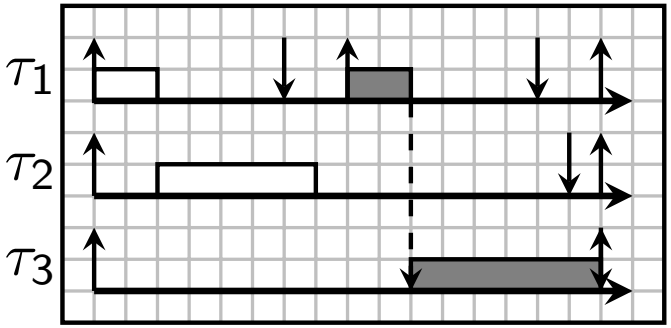
- No anomalies with precedence constraints;
- Precedence constraint enforces relative order of τ_i and τ_j.



(e) Reminder: schedulability analysis



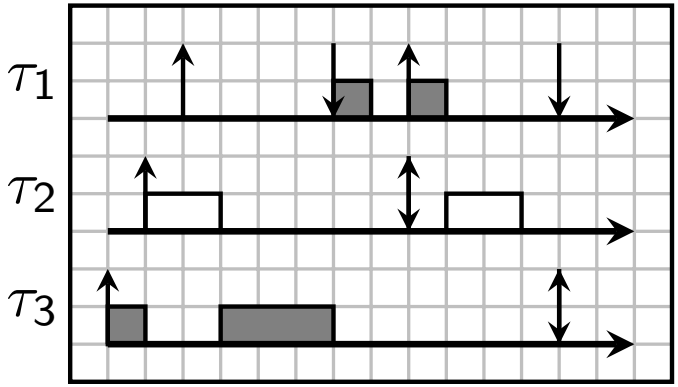
(f) Reminder: anomaly



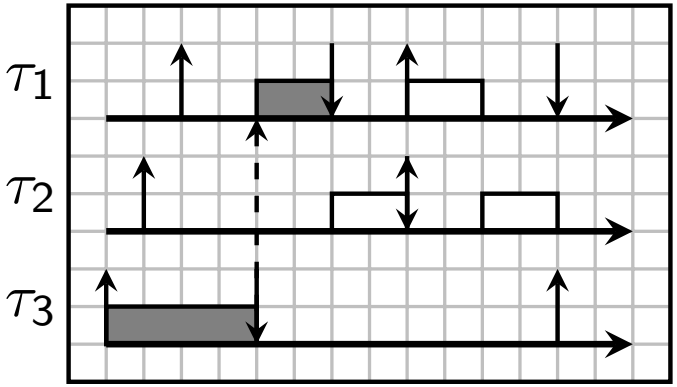
(g) $\tau_{1.1} \rightarrow \tau_{3.0}$

Priority inversion

- No priority inversion with precedence encoding;
- Precedence encoding ⇒ deadline inheritance ⇒ priority inheritance.



(h) Reminder: priority inversion



(i) Encoding: no priority inversion

Outline

1. Real-time model
2. Scheduling
3. Fixed-task priority
4. Fixed-job priority
5. Shared resources
6. Data-dependencies
7. Conclusion

Summary

To be a proficient real-time developper (on a mono-processor platform) you must be able to:

- Design the software architecture of a real-time system:
  - Choose task periods;
  - Evaluate the impact of deadline misses.
- Program the resulting real-time task set:
  - Define and invoke the different tasks;
  - Pay attention to shared resources;
  - Implement task communications/synchronizations
- Choose the correct scheduling policy;
- Test the schedulability of the task set.

Acknowledgements

This presentation borrows elements from:

- *Emmanuel GROLLEAU (LIAS/ISAE-ENSMA)*, Ordonnancement et ordonnançabilité monoprocesseur, Ecole d'Eté Temps Réel (ETR'2011), Brest, 2011
- *Giuseppe LIPARI (Univ. Lille 1)*, Real-time scheduling: from hard to soft, Ecole d'Eté Temps Réel (ETR'2015), Rennes, 2015