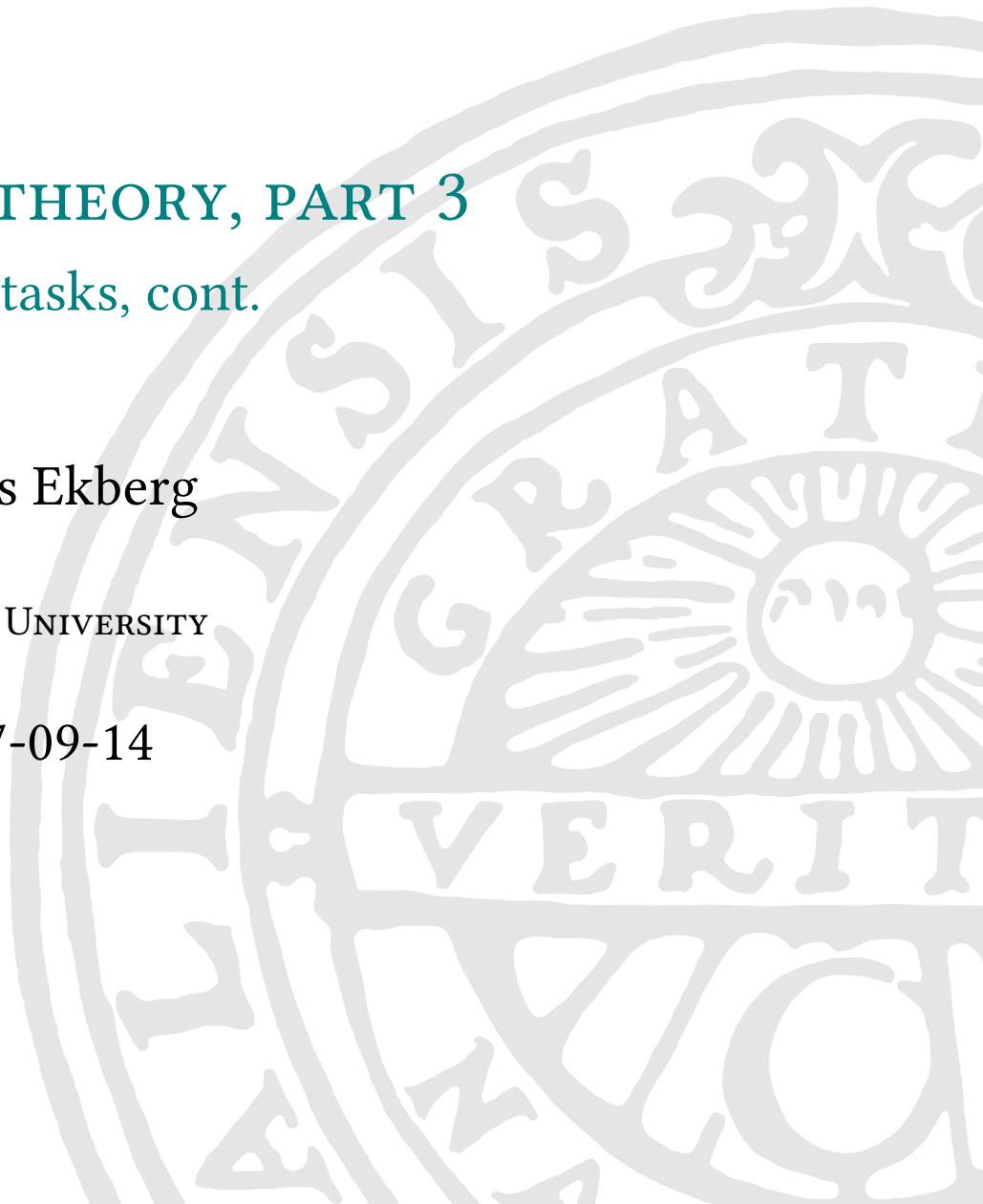
Scheduling theory, part 3

Periodic tasks, cont.

Pontus Ekberg

Uppsala University

2017-09-14



Recap

Synchronous periodic tasks

A periodic task τ_i generates an unbounded sequence of identical jobs. It is specified as $\tau_i = (C_i, D_i, T_i) \in \mathbb{N}^3$, where

- C_i is the worst-case execution time (WCET),
- D_i is the *relative* deadline, and
- T_i is the period.

Recap

Synchronous periodic tasks

A periodic task τ_i generates an unbounded sequence of identical jobs. It is specified as $\tau_i = (C_i, D_i, T_i) \in \mathbb{N}^3$, where

- C_i is the worst-case execution time (WCET),
- D_i is the *relative* deadline, and
- T_i is the period.

Last time

We scheduled task sets $\mathcal{T}$ of synchronous periodic tasks

- with a (static) cyclic executive, and
- on the fly with EDF.

Today's topic

We will look into the following today.

- The *Fixed-Priority* (FP) scheduling algorithm
  - Including schedulability analysis
- Some variations and extensions of the task model:
  - Sporadic tasks
  - Asynchronous periodic tasks
  - Jitter
  - Resource sharing (which you have already seen)
  - Preemption overhead

Another scheduling algorithm

Fixed-Priority (FP) scheduling

Preprocessing: *Assign a static priority to every task τ_i.*

Scheduling rule: *Choose among the ready jobs to execute the job from the task with the highest priority (ties can be broken in various ways).*

Another scheduling algorithm

Fixed-Priority (FP) scheduling

Preprocessing: *Assign a static priority to every task τ_i.*

Scheduling rule: *Choose among the ready jobs to execute the job from the task with the highest priority (ties can be broken in various ways).*

We write $\tau_i \prec \tau_j$ to mean that τ_i has higher priority than τ_j.

Another scheduling algorithm

Fixed-Priority (FP) scheduling

Preprocessing: *Assign a static priority to every task τ_i.*

Scheduling rule: *Choose among the ready jobs to execute the job from the task with the highest priority (ties can be broken in various ways).*

We write $\tau_i \prec \tau_j$ to mean that τ_i has higher priority than τ_j.

Warning: Some authors use higher numbers for higher priorities, and others do it the other way around.

Let's try it...

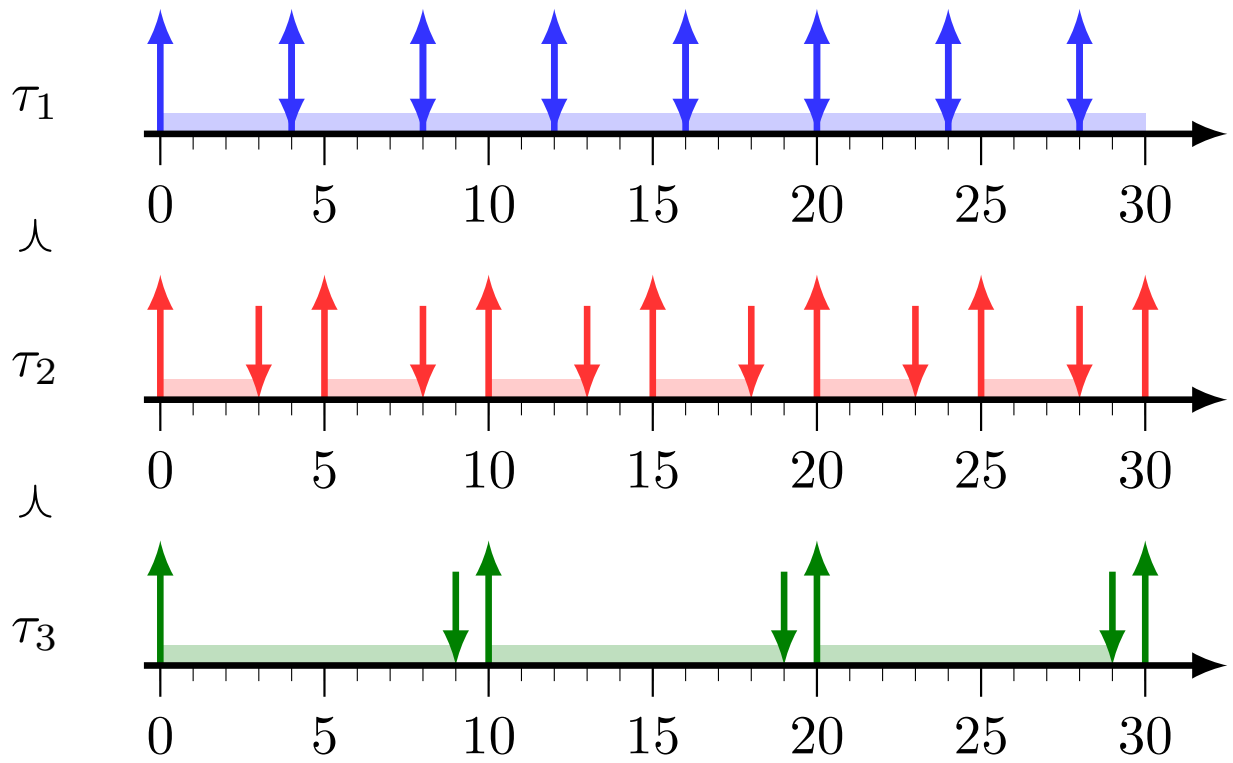
Schedule these tasks

$$\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1, 4, 4), (2, 3, 5), (3, 9, 10)\}$$

Let's try it...

Schedule these tasks

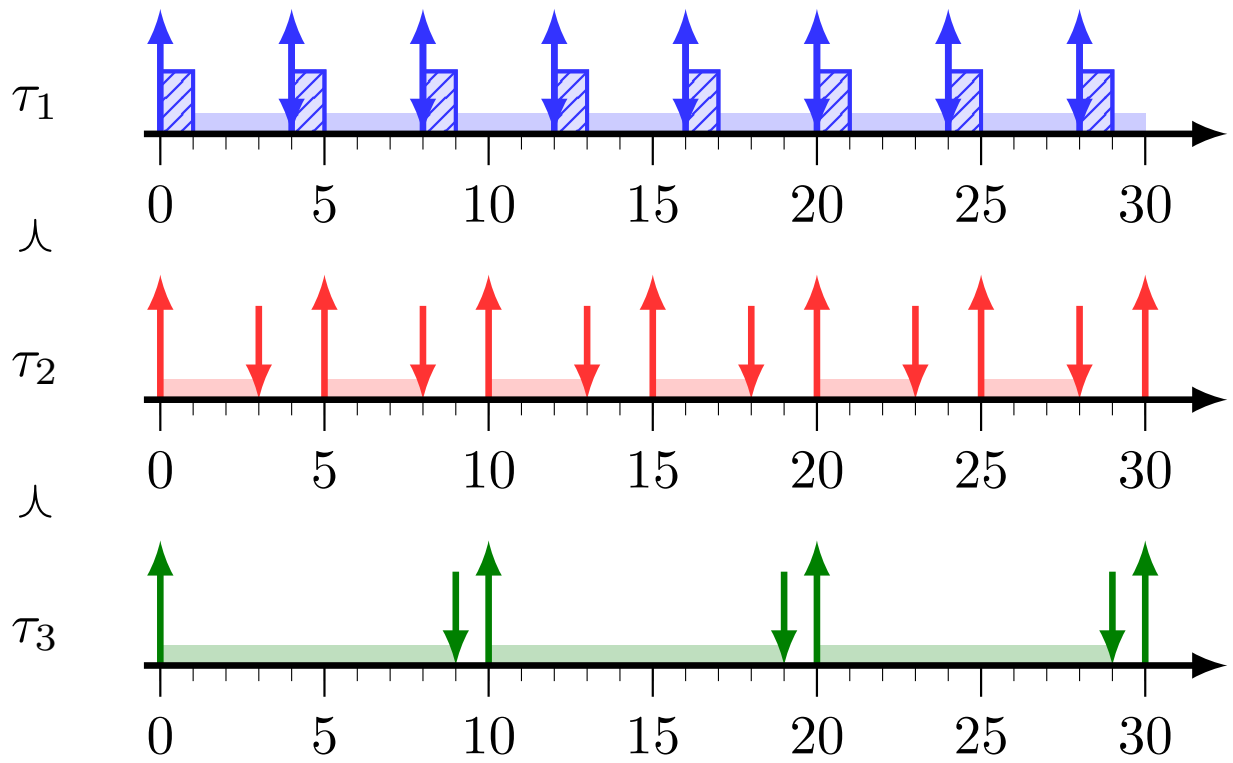
$$\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1, 4, 4), (2, 3, 5), (3, 9, 10)\}$$



Let's try it...

Schedule these tasks

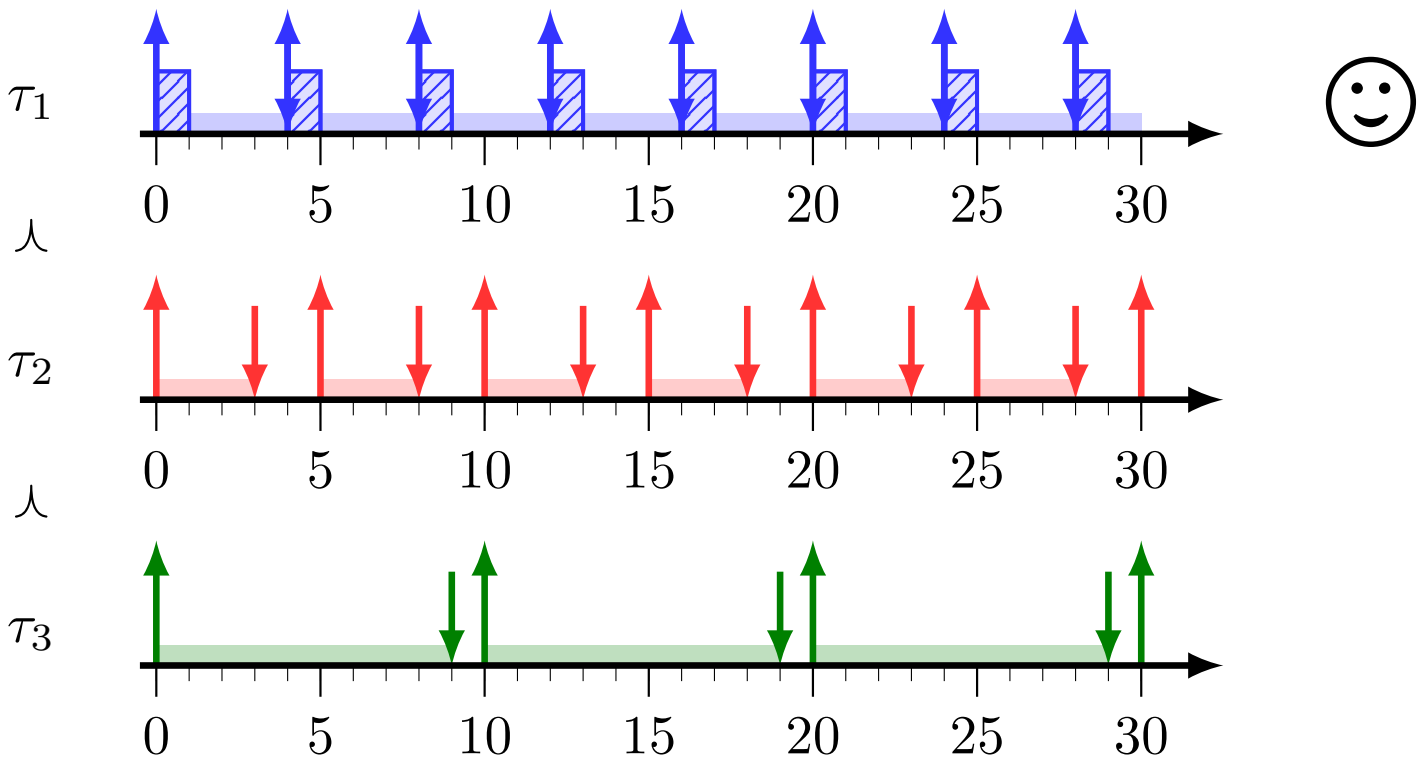
$$\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1, 4, 4), (2, 3, 5), (3, 9, 10)\}$$



Let's try it...

Schedule these tasks

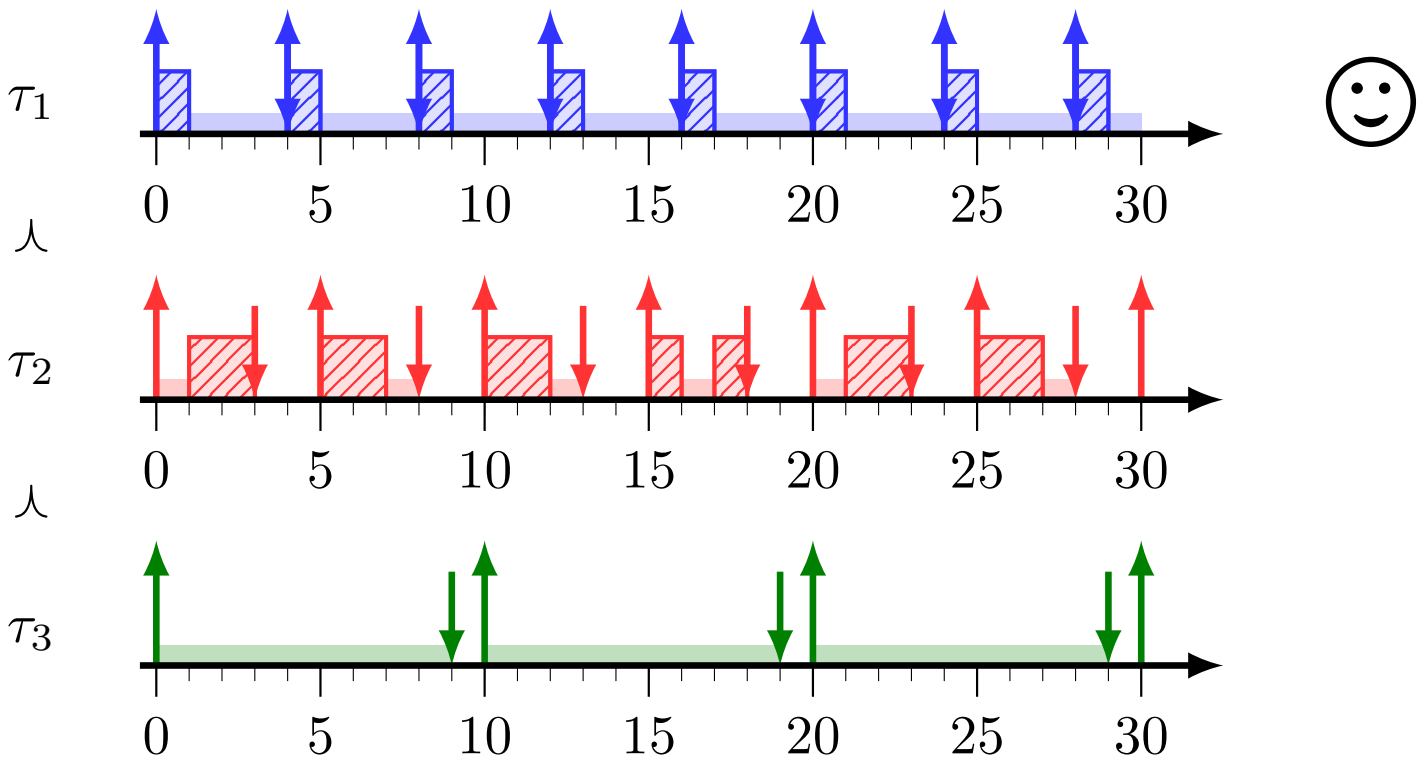
$$\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1, 4, 4), (2, 3, 5), (3, 9, 10)\}$$



Let's try it...

Schedule these tasks

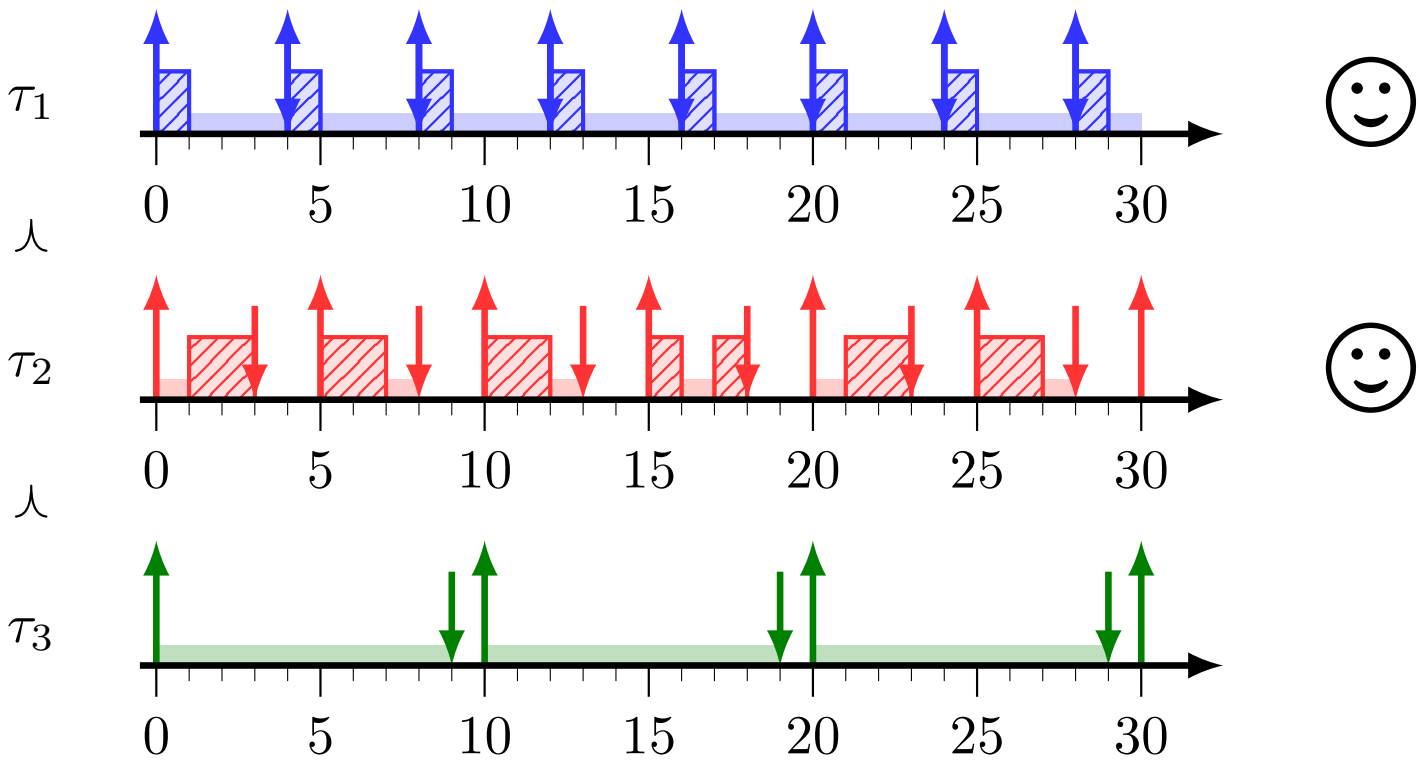
$$\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1, 4, 4), (2, 3, 5), (3, 9, 10)\}$$



Let's try it...

Schedule these tasks

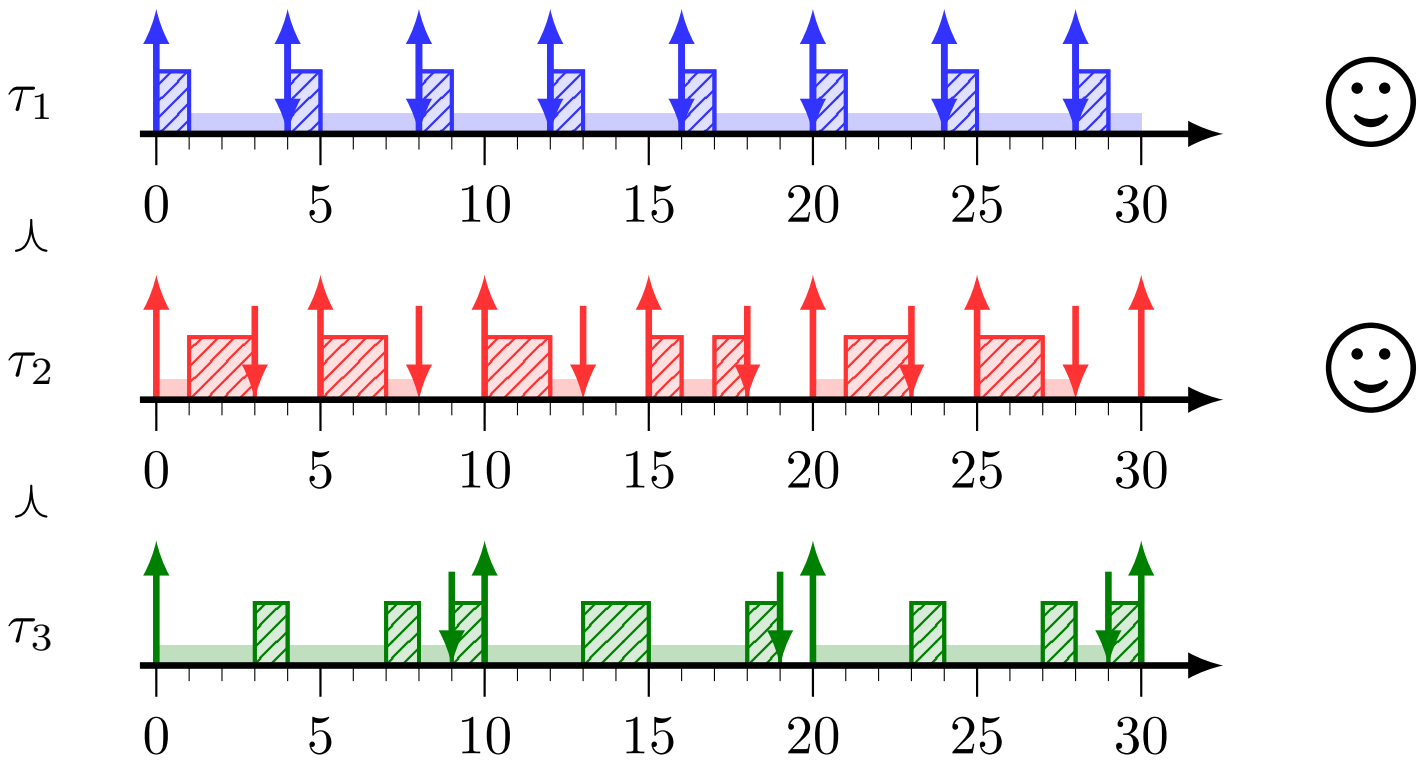
$$\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1, 4, 4), (2, 3, 5), (3, 9, 10)\}$$



Let's try it...

Schedule these tasks

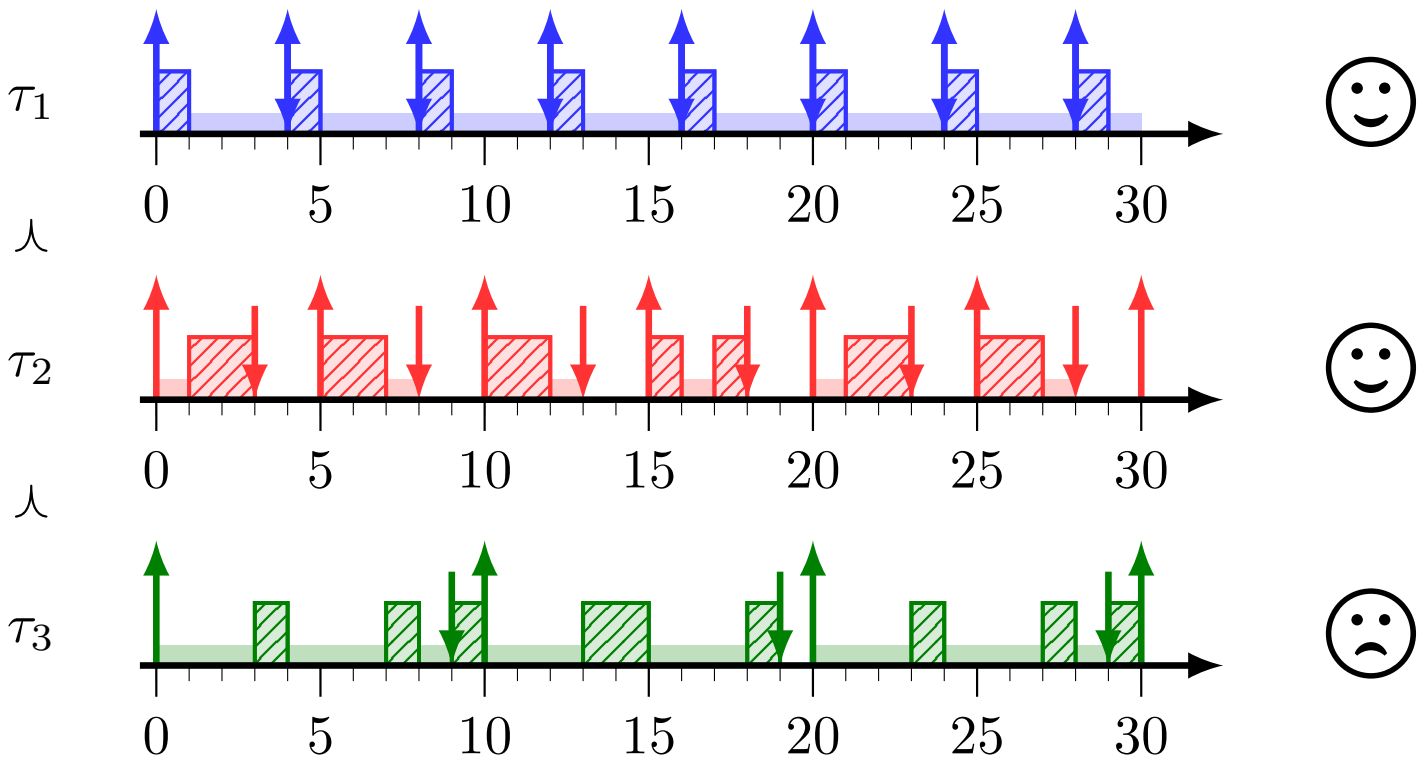
$$\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1, 4, 4), (2, 3, 5), (3, 9, 10)\}$$



Let's try it...

Schedule these tasks

$$\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1, 4, 4), (2, 3, 5), (3, 9, 10)\}$$



Questions

Are some priority orderings better than others?

Questions

Are some priority orderings better than others? **YES!**

Questions

Are some priority orderings better than others? **YES!**

For any feasible task set $\mathcal{T}$, does there always exist some priority ordering with which $\mathcal{T}$ is FP-schedulable?

Questions

Are some priority orderings better than others? **YES!**

For any feasible task set $\mathcal{T}$, does there always exist some priority ordering with which $\mathcal{T}$ is FP-schedulable? **NO!**

Questions

Are some priority orderings better than others? **YES!**

For any feasible task set $\mathcal{T}$, does there always exist some priority ordering with which $\mathcal{T}$ is FP-schedulable? **NO!**

Fixed-priority scheduling is *suboptimal*.

Determining FP-schedulability

The worst case response time (WCRT)

The worst case response time R_i of task τ_i is the longest possible duration between the release time and finishing time of any job from τ_i under a given scheduling policy.

Determining FP-schedulability

The worst case response time (WCRT)

The worst case response time R_i of task τ_i is the longest possible duration between the release time and finishing time of any job from τ_i under a given scheduling policy.

Observation

A task set $\mathcal{T}$ is schedulable under the given scheduling policy iff $R_i \leqslant D_i$ for all $\tau_i \in \mathcal{T}$.

Determining FP-schedulability

The worst case response time (WCRT)

The worst case response time R_i of task τ_i is the longest possible duration between the release time and finishing time of any job from τ_i under a given scheduling policy.

Observation

A task set $\mathcal{T}$ is schedulable under the given scheduling policy iff $R_i \leqslant D_i$ for all $\tau_i \in \mathcal{T}$.

Response times are fairly complicated to compute under EDF, but it is more straightforward under FP.

Calculating response times

An example task set

$$\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1, 4, 4), (3, 6, 6), (3, 20, 20)\},$$
with $\tau_1 \prec \tau_2 \prec \tau_3$.

Calculating response times

An example task set

$\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1, 4, 4), (3, 6, 6), (3, 20, 20)\}$,
with $\tau_1 \prec \tau_2 \prec \tau_3$.

Challenge

Calculate the response time of *the first job* from each task.

Iteratively refining our estimate

Idea

1. Make an *initial underestimation* R_i^1 of the response time (for example, $R_i^1 = C_i$).
2. *Refine the estimate* R_i^k to R_i^{k+1} by including the interference from higher priority tasks we can see in that interval of length R_i^k.
3. If $R_i^k = R_i^{k+1}$, then the estimate is the response time. Otherwise, *repeat* step (2).

Some insights!

A recurrence relation

The response time R_i of the *first job* from τ_i is the *smallest positive fixed point* to the recurrence relation

$$R_i = C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j,$$

where $\mathrm{hp}(\tau_i)$ denotes the set of tasks with higher or equal priority than τ_i (excluding τ_i itself).

Some insights!

A recurrence relation

The response time R_i of the *first job* from τ_i is the *smallest positive fixed point* to the recurrence relation

$$R_i = C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j,$$

where $\mathrm{hp}(\tau_i)$ denotes the set of tasks with higher or equal priority than τ_i (excluding τ_i itself).

For a task set with *constrained deadlines*, the response time of the first job of τ_i is the worst-case response time of τ_i.

Response Time Analysis (RTA)

Theorem (Joseph & Pandya, 1986)

For a task set with *constrained deadlines* under preemptive FP-scheduling on a single processor, the worst case response time R_i of τ_i is the smallest positive fixed point to the recurrence relation

$$R_i = C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j.$$

Response Time Analysis (RTA)

Theorem (Joseph & Pandya, 1986)

For a task set with *constrained deadlines* under preemptive FP-scheduling on a single processor, the worst case response time R_i of τ_i is the smallest positive fixed point to the recurrence relation

$$R_i = C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j.$$

Using this method we can test FP-schedulability in *pseudo-polynomial time* because we only care if $R_i \leqslant D_i$.

Response Time Analysis (RTA)

Theorem (Joseph & Pandya, 1986)

For a task set with *constrained deadlines* under preemptive FP-scheduling on a single processor, the worst case response time R_i of τ_i is the smallest positive fixed point to the recurrence relation

$$R_i = C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j.$$

Using this method we can test FP-schedulability in *pseudo-polynomial time* because we only care if $R_i \leqslant D_i$.

This FP-schedulability problem is NP-complete (RTSS 2017).

How to pick a priority ordering?

How to pick a priority ordering?

Rate-Monotonic ordering

Under the *Rate-Monotonic (RM)* priority ordering, tasks with shorter periods get higher priorities.

How to pick a priority ordering?

Rate-Monotonic ordering

Under the *Rate-Monotonic (RM)* priority ordering, tasks with shorter periods get higher priorities.

Deadline-Monotonic ordering

Under the *Deadline-Monotonic (DM)* priority ordering, tasks with shorter relative deadlines get higher priorities.

RM/DM optimality

Theorem (Liu & Layland, 1973)

RM is an *optimal priority ordering* for synchronous periodic task sets with *implicit deadlines* executed on a single preemptive processor.

RM/DM optimality

Theorem (Liu & Layland, 1973)

RM is an *optimal priority ordering* for synchronous periodic task sets with *implicit deadlines* executed on a single preemptive processor.

Theorem (Leung & Whitehead, 1982)

DM is an *optimal priority ordering* for synchronous periodic task sets with *constrained deadlines* executed on a single preemptive processor.

RM/DM optimality

Theorem (Liu & Layland, 1973)

RM is an *optimal priority ordering* for synchronous periodic task sets with *implicit deadlines* executed on a single preemptive processor.

Theorem (Leung & Whitehead, 1982)

DM is an *optimal priority ordering* for synchronous periodic task sets with *constrained deadlines* executed on a single preemptive processor.

(The latter subsumes the former.)

A simpler (but famous) test!

Theorem (Liu & Layland, 1973)

A task set $\mathcal{T}$ of n tasks with *implicit deadlines* is FP-schedulable with *RM priority ordering* on a single preemptive processor if

$$U(\mathcal{T}) \leqslant n(2^{\frac{1}{n}} - 1).$$

A simpler (but famous) test!

Theorem (Liu & Layland, 1973)

A task set $\mathcal{T}$ of n tasks with *implicit deadlines* is FP-schedulable with *RM priority ordering* on a single preemptive processor if

$$U(\mathcal{T}) \leqslant n(2^{\frac{1}{n}} - 1).$$

Sufficient, but not necessary!

A simpler (but famous) test!

Theorem (Liu & Layland, 1973)

A task set $\mathcal{T}$ of n tasks with *implicit deadlines* is FP-schedulable with *RM priority ordering* on a single preemptive processor if

$$U(\mathcal{T}) \leqslant n(2^{\frac{1}{n}} - 1).$$

Sufficient, but not necessary!

$$\lim_{n \to \infty} n(2^{\frac{1}{n}} - 1) = \ln(2) \approx 0.693$$

A simpler (but famous) test!

Theorem (Liu & Layland, 1973)

A task set $\mathcal{T}$ of n tasks with *implicit deadlines* is FP-schedulable with *RM priority ordering* on a single preemptive processor if

$$U(\mathcal{T}) \leqslant n(2^{\frac{1}{n}} - 1).$$

Sufficient, but not necessary!

$$\lim_{n \to \infty} n(2^{\frac{1}{n}} - 1) = \ln(2) \approx 0.693$$

Compare to $U(\mathcal{T}) \leqslant 1$ for EDF.

Another common task model

The sporadic task model

A *sporadic* task τ_i is also given by a triple $(C_i, D_i, T_i) \in \mathbb{N}^3$, where

- C_i is the worst-case execution time (WCET),
- D_i is the relative deadline, and
- T_i is the *minimum inter-arrival time* (but often just called "the period").

Another common task model

The sporadic task model

A *sporadic* task τ_i is also given by a triple $(C_i, D_i, T_i) \in \mathbb{N}^3$, where

- C_i is the worst-case execution time (WCET),
- D_i is the relative deadline, and
- T_i is the *minimum inter-arrival time* (but often just called "the period").

Each sporadic task can generate an infinite number of job sequences!

A job sequence to end them all

The synchronous arrival sequence (SAS)

Out of all the job sequences that a set of sporadic task can generate, the hardest to schedule (both for EDF and FP) on a single preemptive processor is the one where all tasks

1. release their first jobs at the same time, and
2. then release jobs as early as possible.

A job sequence to end them all

The synchronous arrival sequence (SAS)

Out of all the job sequences that a set of sporadic task can generate, the hardest to schedule (both for EDF and FP) on a single preemptive processor is the one where all tasks

1. release their first jobs at the same time, and
2. then release jobs as early as possible.

This is the sequence generated by a set of synchronous periodic tasks!

A job sequence to end them all

The synchronous arrival sequence (SAS)

Out of all the job sequences that a set of sporadic task can generate, the hardest to schedule (both for EDF and FP) on a single preemptive processor is the one where all tasks

1. release their first jobs at the same time, and
2. then release jobs as early as possible.

This is the sequence generated by a set of synchronous periodic tasks!

The tests for synchronous periodic tasks are also valid for sporadic tasks! (But you can't do the static schedule.)

Yet another task model...

The asynchronous periodic task model

An *asynchronous periodic* task τ_i is given by a quadruple $(O_i, C_i, D_i, T_i) \in \mathbb{N}^4$, where

- O_i is the *offset* (the arrival time of the first job),
- C_i is the worst-case execution time (WCET),
- D_i is the relative deadline, and
- T_i is the the period.

Sometimes asynchronous periodic tasks are just called "periodic tasks".

Analyzing asynchronous tasks

Observation

Because the synchronous arrival sequence (SAS) is always the hardest to schedule, an asynchronous periodic task set *can never be harder* to schedule for EDF or FP than the corresponding synchronous periodic task set.

Analyzing asynchronous tasks

Observation

Because the synchronous arrival sequence (SAS) is always the hardest to schedule, an asynchronous periodic task set *can never be harder* to schedule for EDF or FP than the corresponding synchronous periodic task set.

The tests for synchronous periodic (or sporadic) tasks are still *sufficient* for asynchronous periodic tasks!

Analyzing asynchronous tasks

Observation

Because the synchronous arrival sequence (SAS) is always the hardest to schedule, an asynchronous periodic task set *can never be harder* to schedule for EDF or FP than the corresponding synchronous periodic task set.

The tests for synchronous periodic (or sporadic) tasks are still *sufficient* for asynchronous periodic tasks!

Exact tests for asynchronous periodic tasks are generally more involved and computationally harder.

An extension to the model: jitter

Jitter

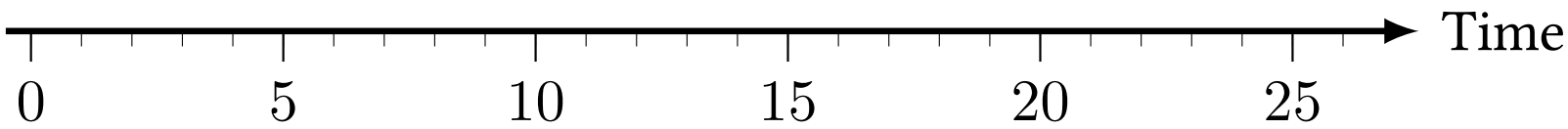
A task is said to experience *jitter* if there is some variation in the time it takes for its jobs to become available for execution after their arrival.

An extension to the model: jitter

Jitter

A task is said to experience *jitter* if there is some variation in the time it takes for its jobs to become available for execution after their arrival.

$$\tau_i = (C_i, D_i, T_i, \mathcal{J}_i) = (2, 6, 10, 3)$$

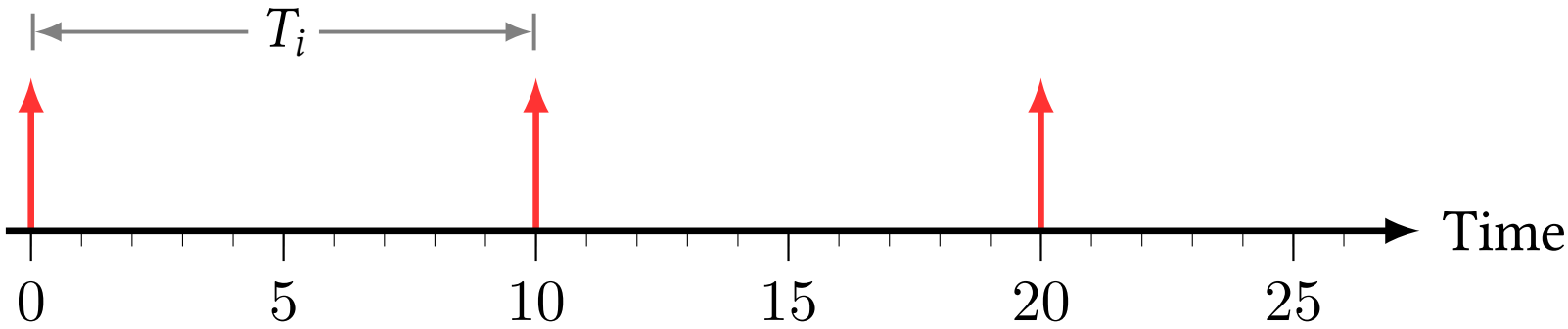


An extension to the model: jitter

Jitter

A task is said to experience *jitter* if there is some variation in the time it takes for its jobs to become available for execution after their arrival.

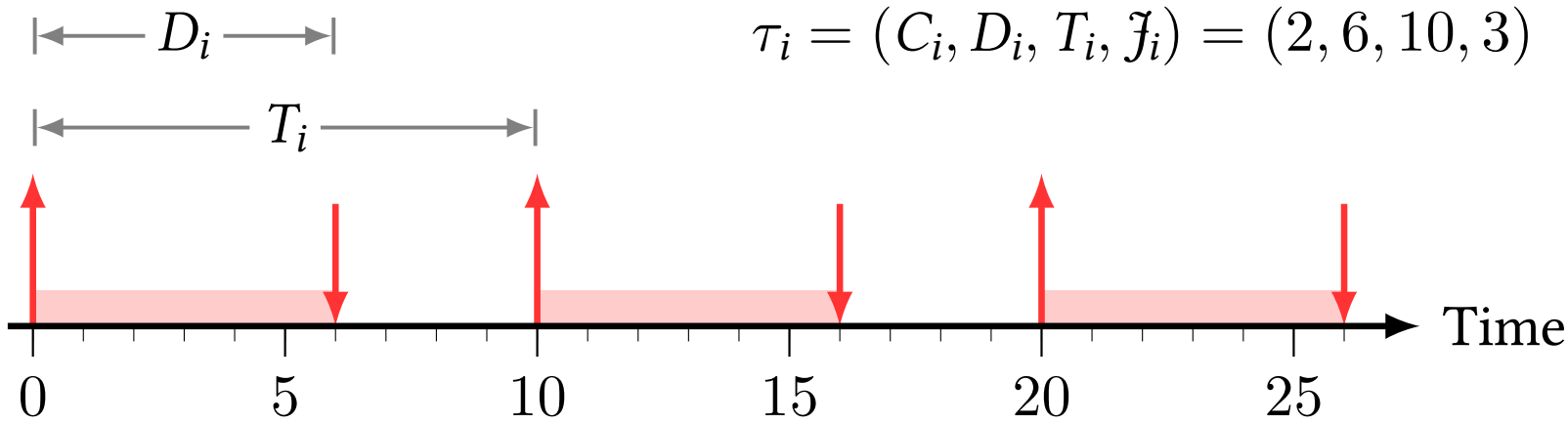
$$\tau_i = (C_i, D_i, T_i, \mathcal{J}_i) = (2, 6, 10, 3)$$



An extension to the model: jitter

Jitter

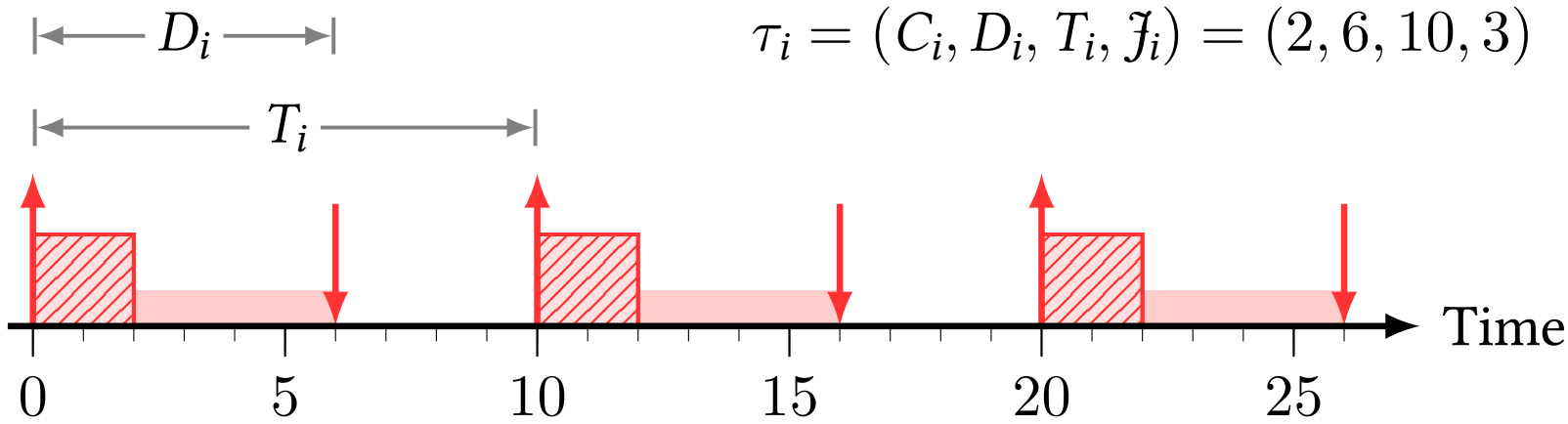
A task is said to experience *jitter* if there is some variation in the time it takes for its jobs to become available for execution after their arrival.



An extension to the model: jitter

Jitter

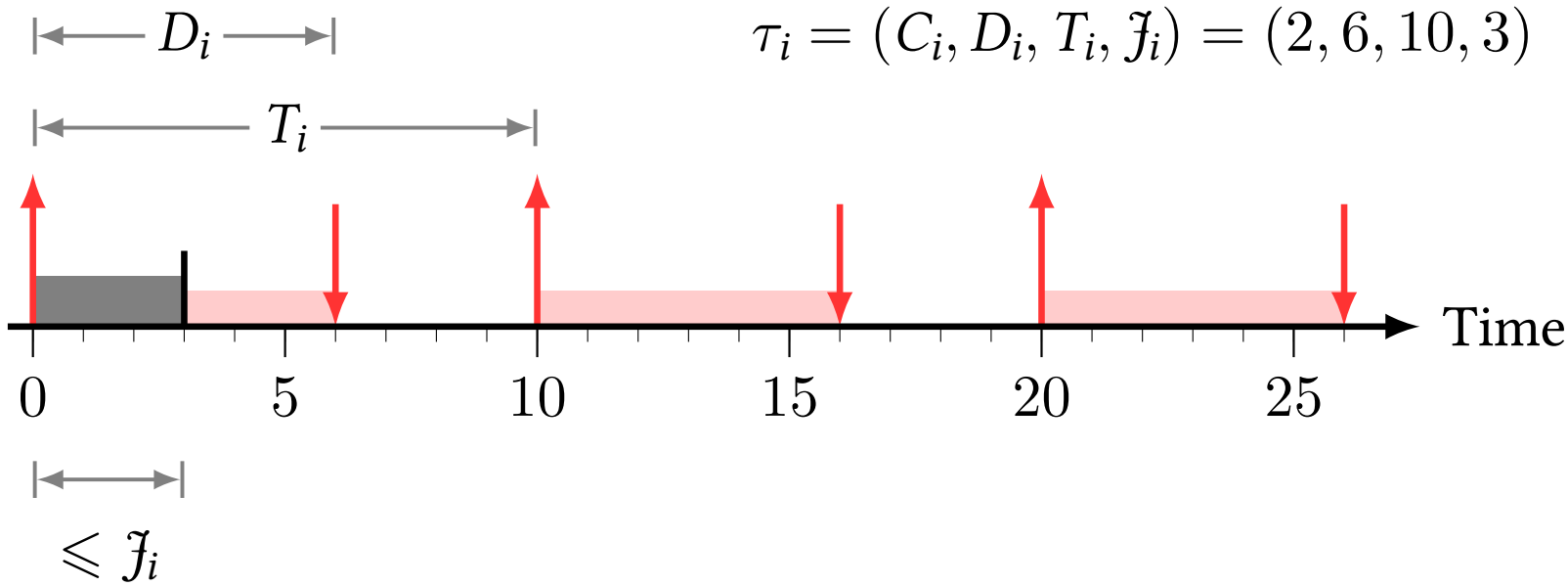
A task is said to experience *jitter* if there is some variation in the time it takes for its jobs to become available for execution after their arrival.



An extension to the model: jitter

Jitter

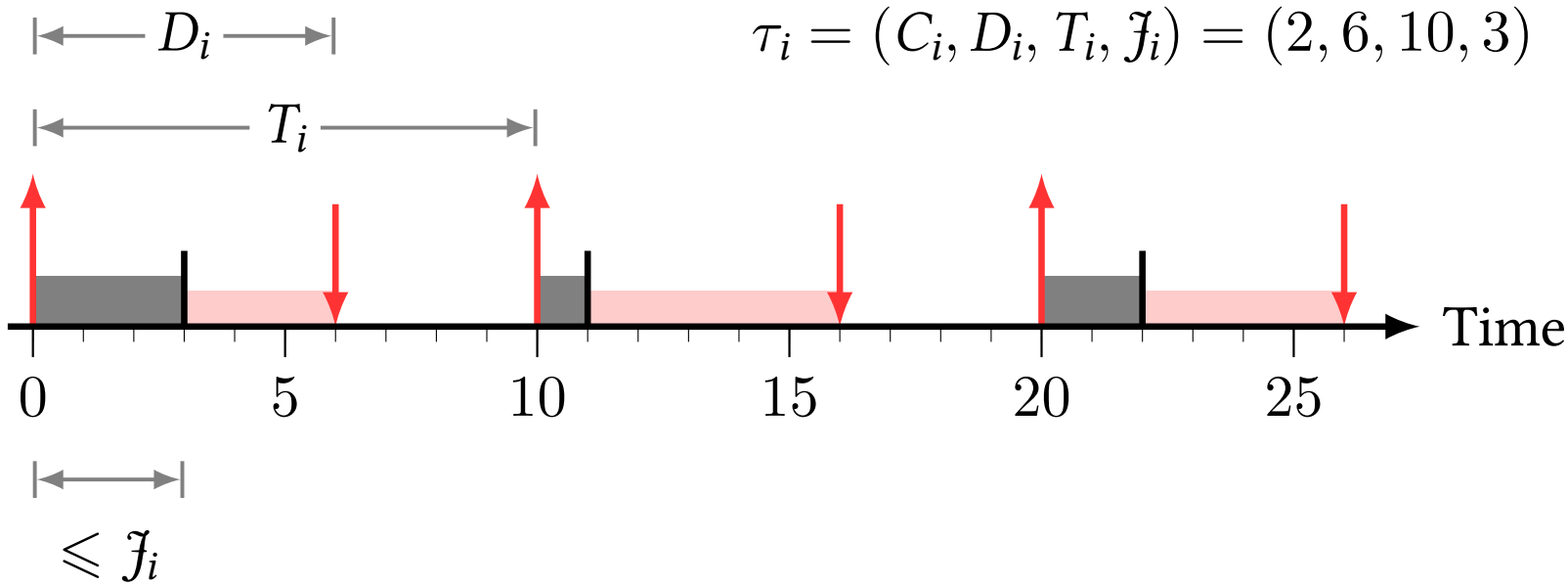
A task is said to experience *jitter* if there is some variation in the time it takes for its jobs to become available for execution after their arrival.



An extension to the model: jitter

Jitter

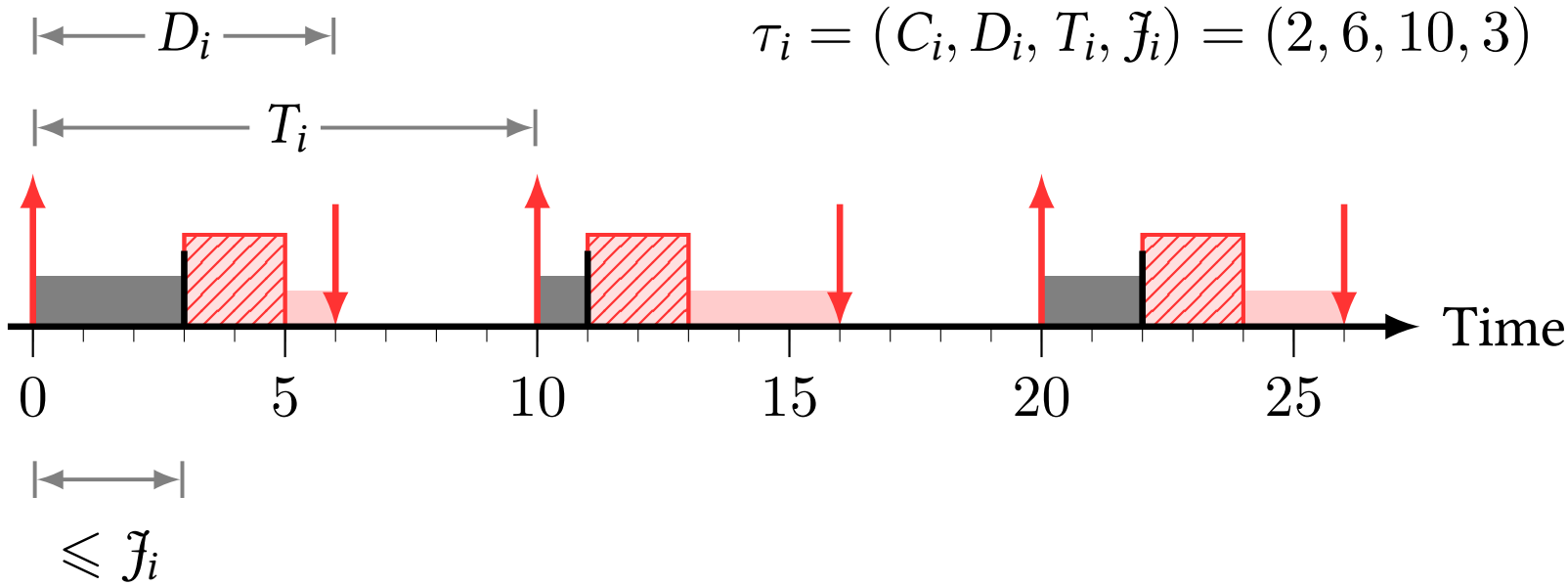
A task is said to experience *jitter* if there is some variation in the time it takes for its jobs to become available for execution after their arrival.



An extension to the model: jitter

Jitter

A task is said to experience *jitter* if there is some variation in the time it takes for its jobs to become available for execution after their arrival.



Extending RTA with jitter

For a task set with *constrained deadlines and jitter* under preemptive FP-scheduling on a single processor, the worst case response time R_i of τ_i is *at most*

$$R_i = \mathcal{J}_i + w_i,$$

where w_i is the smallest positive fixed point to

$$w_i = C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \left\lceil \frac{w_i + \mathcal{J}_j}{T_j} \right\rceil \cdot C_j.$$

Extending RTA with jitter

For a task set with *constrained deadlines and jitter* under preemptive FP-scheduling on a single processor, the worst case response time R_i of τ_i is *at most*

$$R_i = \mathcal{J}_i + w_i,$$

where w_i is the smallest positive fixed point to

$$w_i = C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \left\lceil \frac{w_i + \mathcal{J}_j}{T_j} \right\rceil \cdot C_j.$$

Note 1: Exact for sporadic tasks, but only sufficient for periodic tasks.

Extending RTA with jitter

For a task set with *constrained deadlines and jitter* under preemptive FP-scheduling on a single processor, the worst case response time R_i of τ_i is *at most*

$$R_i = \mathcal{J}_i + w_i,$$

where w_i is the smallest positive fixed point to

$$w_i = C_i + \sum_{\tau_j \in \text{hp}(\tau_i)} \left\lceil \frac{w_i + \mathcal{J}_j}{T_j} \right\rceil \cdot C_j.$$

Note 1: Exact for sporadic tasks, but only sufficient for periodic tasks.

Note 2: DM might not be the optimal priority ordering here.

Accounting for resource sharing

Recap: With a given resource sharing protocol (PIP, PCP, etc.), task τ_i may experience up to B_i time units of *blocking due to priority inversion*. (See Wang's previous slides and Buttazzo's book for details on how to compute B_i.)

Accounting for resource sharing

Recap: With a given resource sharing protocol (PIP, PCP, etc.), task τ_i may experience up to B_i time units of *blocking due to priority inversion*. (See Wang's previous slides and Buttazzo's book for details on how to compute B_i.)

For a task set with *constrained deadlines and blocking* under preemptive FP-scheduling on a single processor, the worst case response time R_i of τ_i is the smallest positive fixed point to

$$R_i = C_i + B_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j.$$

Preemption overhead

So far we have assumed that preemptions are *free*, but in reality they are not. Preemptions may inflate execution times due to

- Context-switch overheads
  - Saving current execution state
  - Loading new execution state
  - …
- Tainting cache states
- Other microarchitectural features
- …

Counting preemptions

A nice property

For both EDF and FP, there can be *at most one* new preemption per job release.

Counting preemptions

A nice property

For both EDF and FP, there can be *at most one* new preemption per job release.

If we can estimate an upper bound on the total cost of doing one preemption, then a safe (but likely pessimistic) approach is to simply inflate every WCET by this cost.

Counting preemptions

A nice property

For both EDF and FP, there can be *at most one* new preemption per job release.

If we can estimate an upper bound on the total cost of doing one preemption, then a safe (but likely pessimistic) approach is to simply inflate every WCET by this cost.

Reducing and managing preemptions is an active area of research, see Buttazzo's book (chapter 8) for some more details.

Highlights

- FP scheduling is more commonly found than EDF in RTOSs.
- FP is suboptimal: It can not schedule all feasible task sets.
- The priority ordering matters greatly.
  - Rate-monotonic (RM) is optimal for implicit deadlines.
  - Deadline-monotonic (DM) is optimal for constrained deadlines.
- With response time analysis (RTA) we can find WCRTs of tasks
  - We get a schedulability test by checking if $R_i \leqslant D_i$ for all τ_i.
  - Test runs in pseudo-polynomial time with constrained deadlines.
    - Which is quite fast in practice.
    - More complicated for arbitrary deadlines (we didn't cover this).
    - Test can be extended to handle things like jitter.
- Liu & Layland's utilization bound offers a simpler sufficient test.
  - Only for RM and implicit deadlines.
- Sporadic tasks are often more flexible than periodic tasks.
  - Because the SAS is the worst-case job sequence, (most) schedulability tests are the same as for synchronous periodic tasks.
- Asynchronous periodic tasks have offsets for the first job.
  - Tests for sporadic tasks are still sufficient.