Embedded Systems

Lecture 8
Deterministic Scheduling

© Michele Magno

D-ITET Center for Project-Based Learning

Credits: Lothar Thiele

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

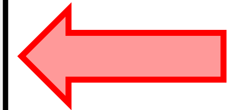
Where We are

Hardware-Software

Real-Time

Special

0. Introduction into Embedded Systems
1. Hardware-Software Architecture and Software Development
2. Hardware-Software Interfaces – (GPIO), Interrupt, and Clock
3. Hardware-Software Interfaces - Serial Interfaces
4. No Lecture
5. Hardware-Software Interfaces - Timer, ADC
6. Real-Time Systems
7. Dynamic Scheduling and Real-Time Operating Systems
8. Deterministic Scheduling
9. Low Power Design
10. Computational Units
11. Implementation Strategies & Project Kick-off
12. Project Q&A



Basic Terms and Models

Basic Terms (Recap)

Real-time systems

- *Hard:* A real-time task is said to be hard, if missing its deadline may cause catastrophic consequences on the environment under control. Examples are sensory data acquisition, detection of critical conditions, actuator servoing.

- *Soft:* A real-time task is called soft, if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior. Examples are command interpreter of the user interface, displaying messages on the screen.

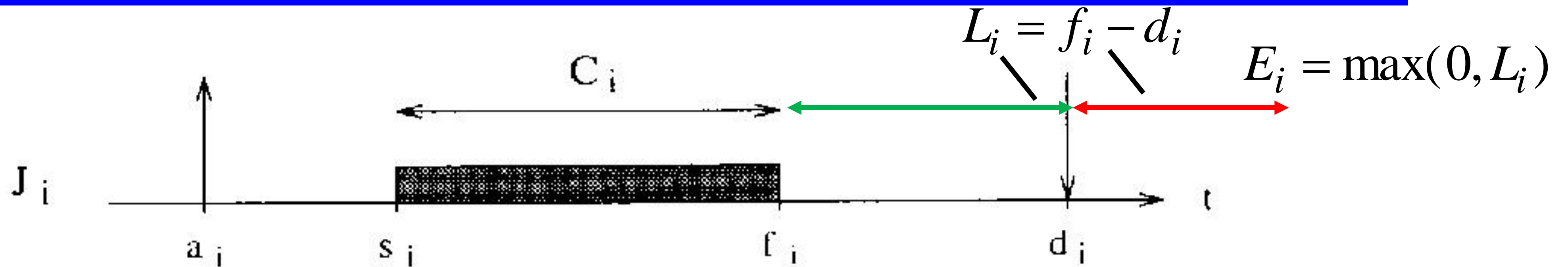
Hard Real-Time Computing Systems, page 26

Schedule and Timing (Summary)

- A schedule is said to be *feasible*, if all task can be completed according to a set of **specified constraints**. (e.g., deadlines, priorities, execution orders) (CPU, memory, I/O,)
- A set of tasks is said to be *schedulable*, **if** there exists at least one algorithm that can produce a feasible schedule.
- *Arrival time* a_i or *release time* r_i is the time at which a task **becomes ready** for execution.
- *Computation time* C_i is the time necessary to the processor for executing the task without interruption.
- *Deadline* d_i is the time at which a task should be completed.
- *Start time* s_i is the time at which a task starts its execution.
- *Finishing time* f_i is the time at which a task finishes its execution.
- A *preemptive schedule* is a schedule in which the running task can be arbitrarily suspended at any time, to assign the CPU to another task according to a predefined scheduling policy.

Hard Real-Time Computing Systems, page 27

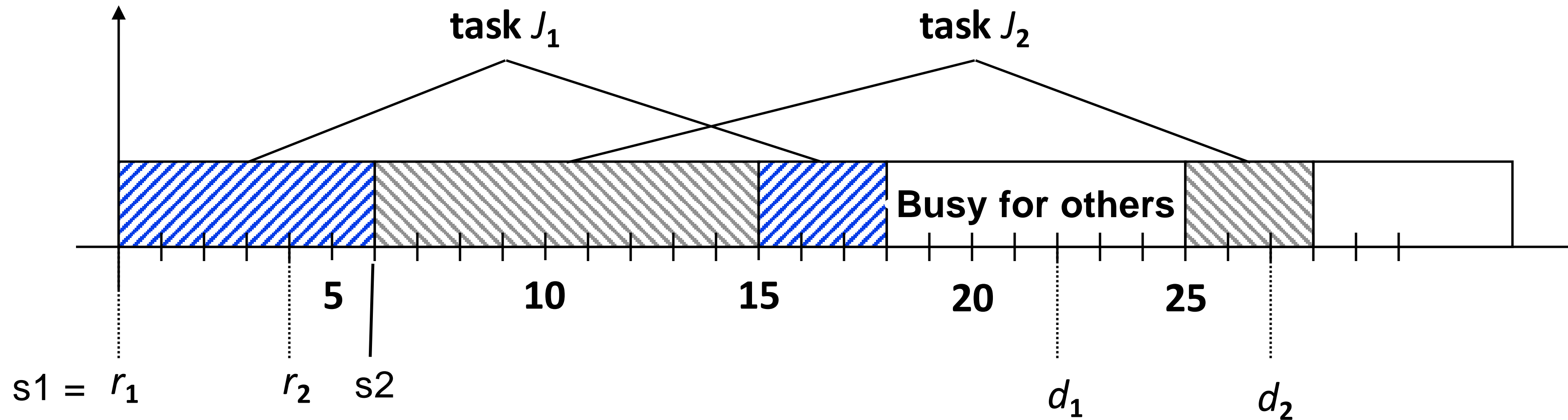
Schedule and Timing (Summary)



- Using the above definitions, we have $d_i \geq a_i + C_i$
- *Lateness* $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline; note that **if a task completes before the deadline, its lateness is negative.**
- *Tardiness or exceeding time* $E_i = \max(0, L_i)$ is the time a task stays active after its deadline.
- *Laxity or slack time* $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its activation to complete within its deadline.

Hard Real-Time Computing Systems, page 27

Example for Real-Time Model



Computation times: $C_1 = 9$, $C_2 = 12$

Start times: $s_1 = 0$, $s_2 = 6$

Finishing times: $f_1 = 18$, $f_2 = 28$ - **Deadlines: $d_1 = 24$, $d_2 = 27$**

Lateness: $L_1 = -4$, $\mathbf{L_2 = 1}$

Tardiness: $\mathbf{E_1 = 0}$, $E_2 = 1$

Laxity: $X_1 = 13$, $X_2 = 11 = 27 - 12 - 4$

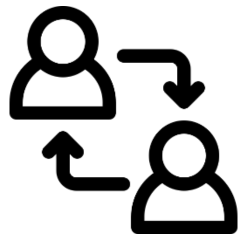
Metrics to Compare Schedules

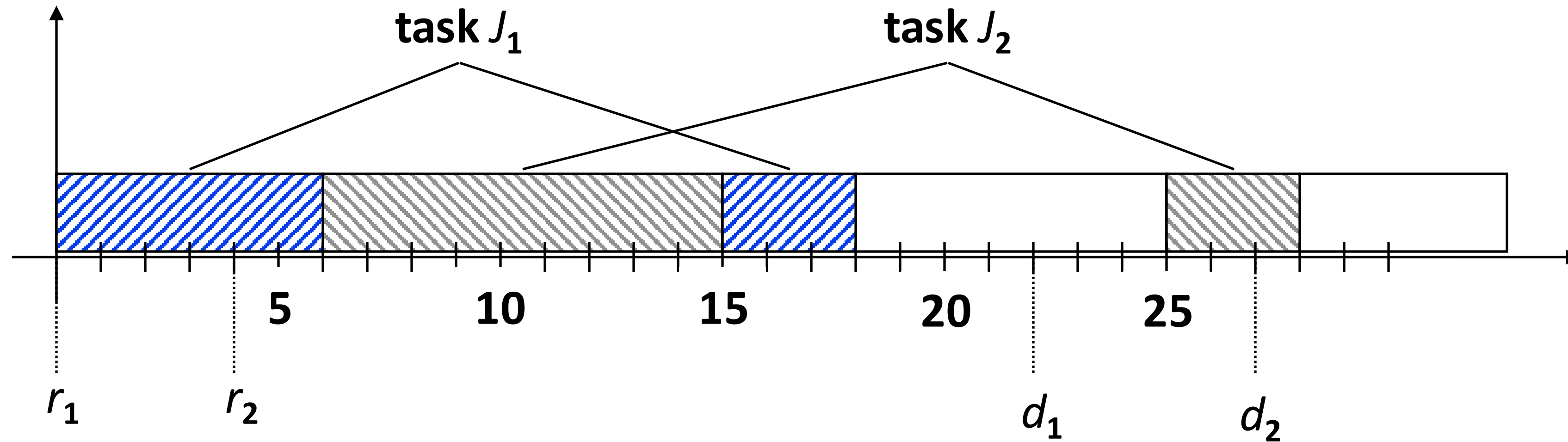
- Average response time: $$\overline{t_r} = \frac{1}{n}\sum_{i=1}^{n}(f_i - r_i)$$
- Total completion time: $$t_c = \max_i(f_i) - \min_i(r_i)$$
- Weighted sum of response time: $$t_w = \frac{\sum_{i=1}^{n} w_i (f_i - r_i)}{\sum_{i=1}^{n} w_i}$$
  - Wi: Weight of task reflecting its importance or priority (e.g., higher weight for critical tasks).
- Maximum lateness: $$L_{\max} = \max_i(f_i - d_i)$$
- Number of late tasks: $$N_{\text{late}} = \sum_{i=1}^{n} miss(f_i)$$

$$miss(f_i) = \begin{cases} 0 & \text{if } f_i \le d_i \\ 1 & \text{otherwise} \end{cases}$$

Hard Real-Time Computing Systems, page 40

Metrics Example





Average response time: $\overline{t_r} = \frac{1}{2}(18 + 24) = 21$

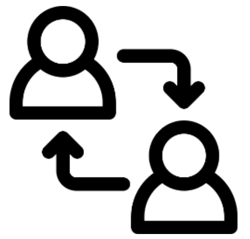
Total completion time: $t_c = 28 - 0 = 28$

Weighted sum of response times: $w_1 = 2, w_2 = 1: \quad t_w = \frac{2 \cdot 18 + 24}{3} = 20$

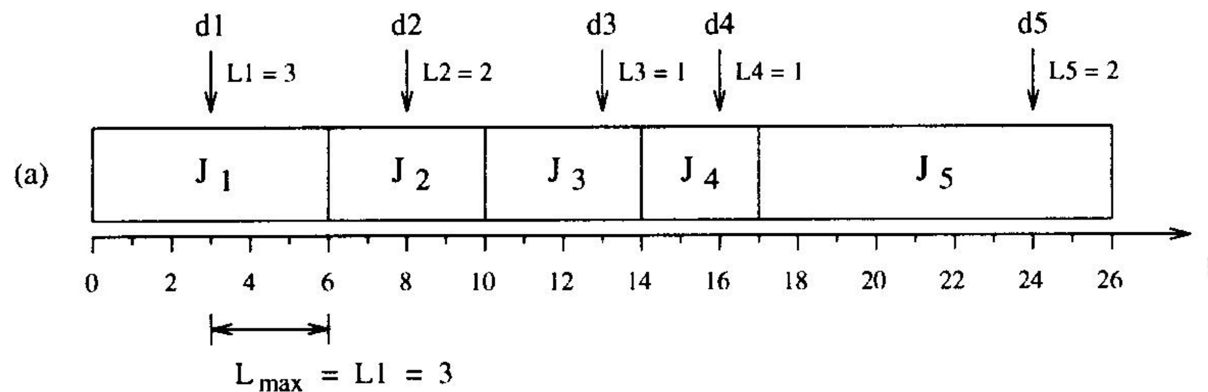
Number of late tasks: $N_{\text{late}} = 1$

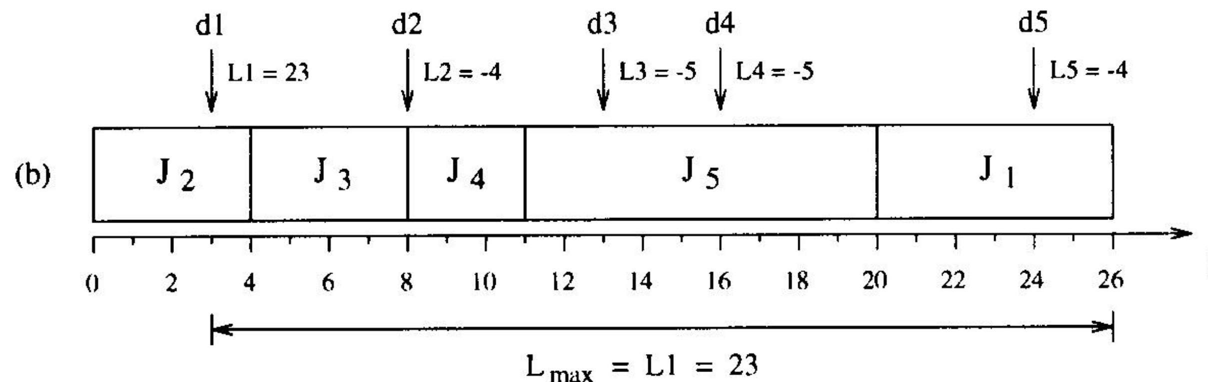
Maximum lateness: $L_{\max} = 1$

Assuming the weights – not realistic



- Explore the trade-offs in scheduling policies for embedded systems by comparing two different schedules: one that minimizes **maximum lateness**, and another with a higher **maximum lateness**.

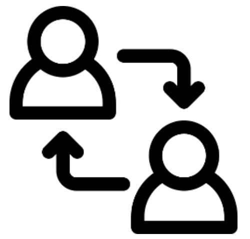




Examine the Gantt Charts for both schedules.

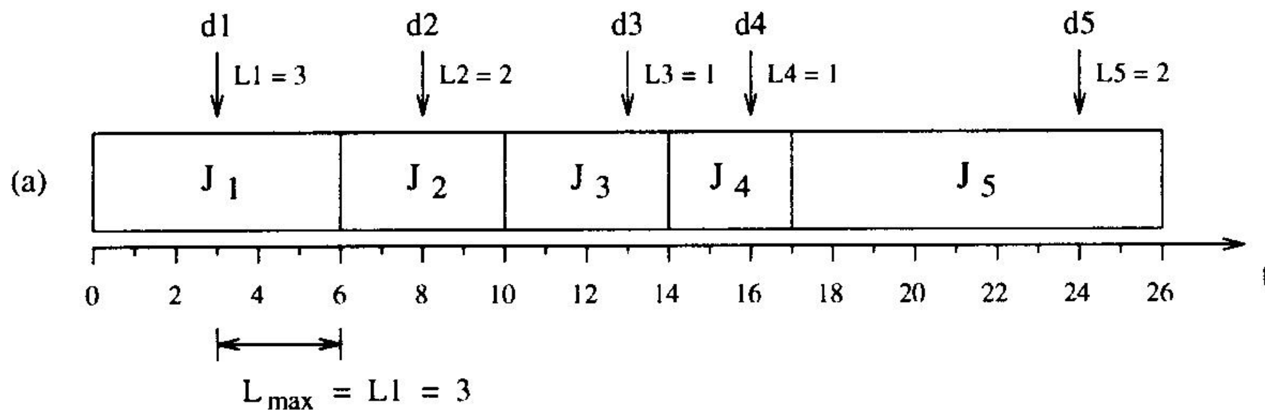
Identify task completion times and compare them to deadlines. Which one miss more deadlines?

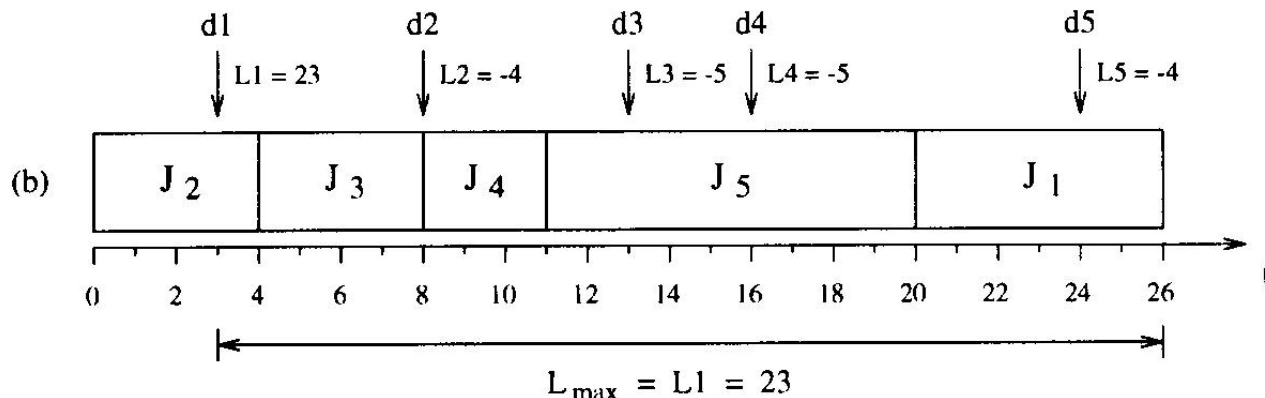
Metrics and Scheduling Example



In schedule (a), the *maximum lateness is minimized*, but all tasks miss their deadlines.

In schedule (b), the maximal lateness is larger, but only one *task misses* its deadline.





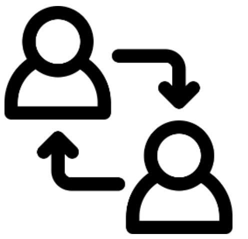
Discuss Trade-offs:

- What are the implications of minimizing maximum lateness in Schedule (a)?
- How does minimizing missed deadlines in Schedule (b) affect overall system performance?

Decision Point: Based on the system's requirements (e.g., hard vs. soft real-time constraints), which schedule would you prefer?

Hard Real-Time Computing Systems, page 41

Trade-Off Analysis



Factor	Schedule (a)	Schedule (b)
Maximum Lateness	Smallest across tasks (minimized).	Larger for some tasks.
Number of Missed Deadlines	All tasks miss their deadlines.	Only one task misses its deadline.
Applicability	Soft real-time systems (e.g., multimedia).	Hard real-time systems (e.g., avionics).
Predictability	High (small, uniform lateness).	Lower (due to larger lateness).
Critical Task Reliability	May fail to meet deadlines for all tasks.	Prioritizes deadlines for key tasks.

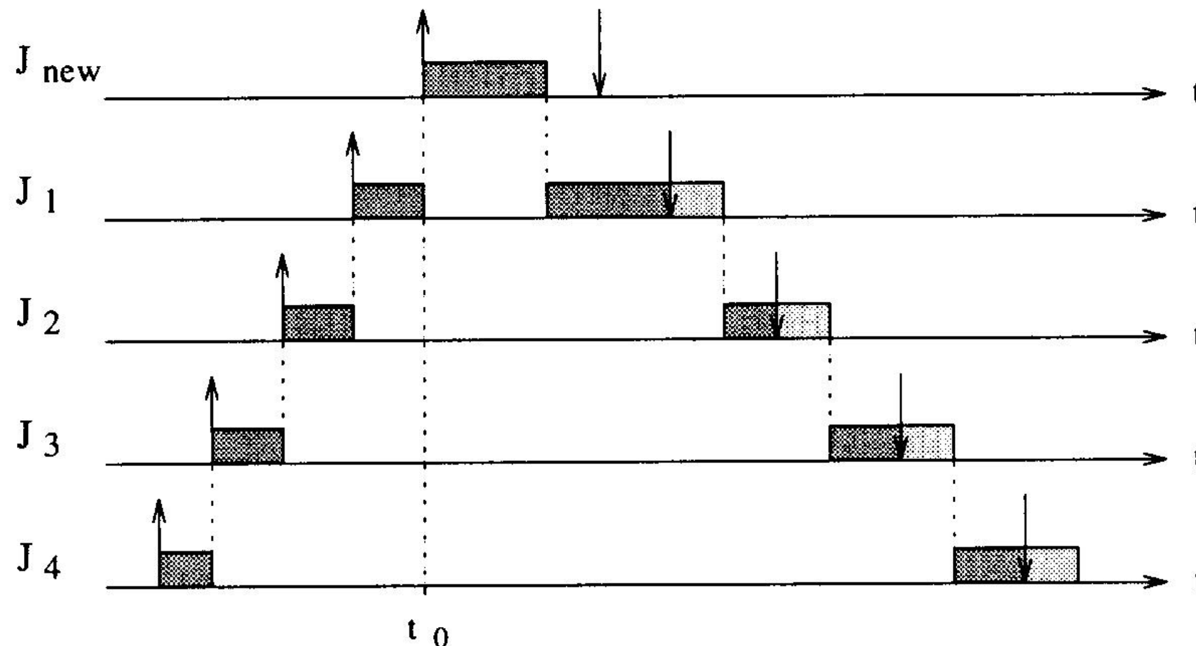
Classification of Scheduling Algorithms

- With *preemptive algorithms*, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy.
- With a *non-preemptive algorithm*, a task, once started, is executed by the processor until completion.

- *Static algorithms* are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.
- *Dynamic algorithms* are those in which scheduling decisions are based on dynamic parameters that may change during system execution.

Classification of Scheduling Algorithms

- An algorithm is said *optimal* if it minimizes some given cost function defined over the task set.
- An algorithm is said to be *heuristic* if it tends toward but does not guarantee to find the optimal schedule.
- *Acceptance Test:* The runtime system decides whenever a task is added to the system, whether it can schedule the whole task set without deadline violations.



Example for the „*domino effect*“, if an acceptance test wrongly accepted a new task.

Real-Time Scheduling of Aperiodic Tasks

Overview Aperiodic Task Scheduling

Scheduling of *aperiodic tasks* with real-time constraints.

Aperiodic tasks in real-time systems are tasks that arrive **unpredictably and do not have regular intervals** between activations. Their scheduling must accommodate real-time constraints to ensure **deadlines are met** while balancing system resources and performance.

- Table with some known algorithms:

	Equal arrival times non preemptive	Arbitrary arrival times preemptive
Independent tasks	EDD (Jackson)	EDF (Horn)
Dependent tasks	LDF (Lawler)	EDF* (Chetto)

Hard Real-Time Computing Systems, page 77

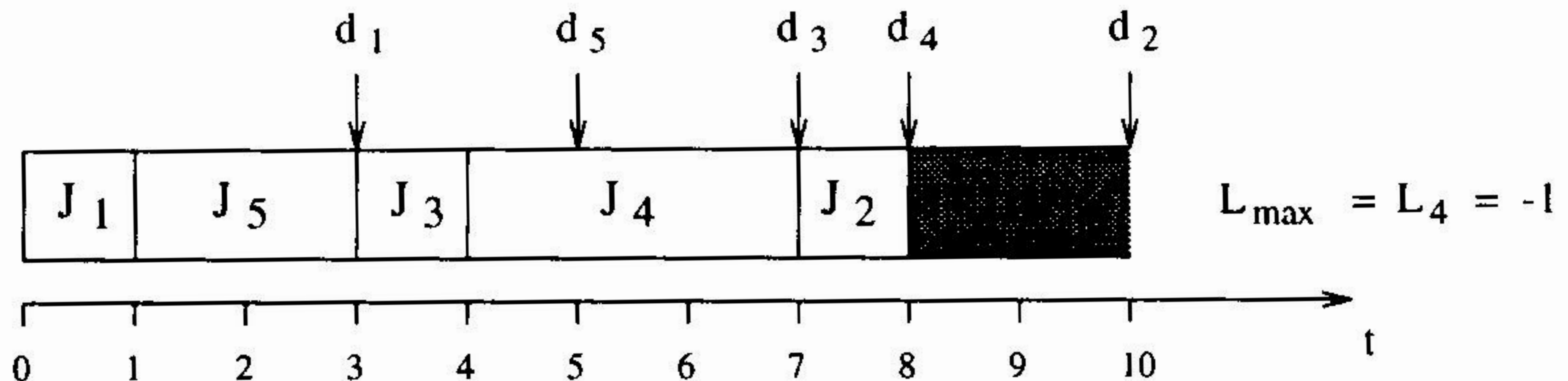
Earliest Deadline Due (EDD)

Jackson's rule: Given a set of *n* tasks. Processing in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness.

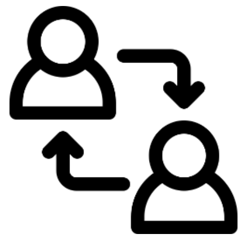
Earliest Deadline Due (EDD)

Example 1:

	J_1	J_2	J_3	J_4	J_5
C_i	1	1	1	3	2
d_i	3	10	7	8	5



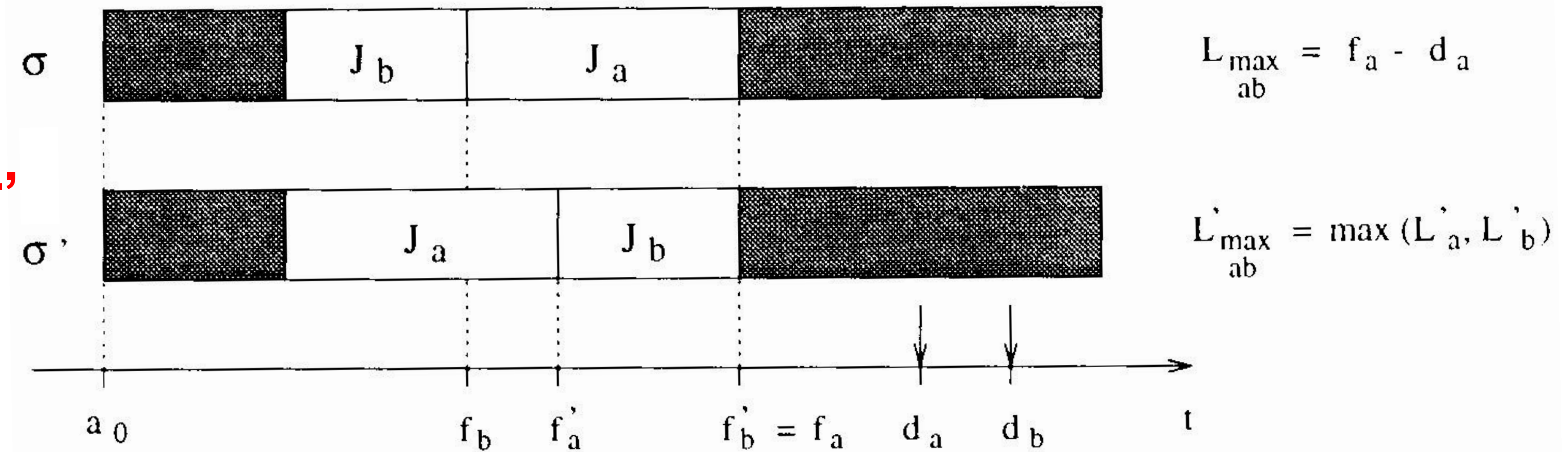
Earliest Deadline Due (EDD)



Jackson's rule: Given a set of *n* tasks. Processing in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness.

Proof concept:

Assuming σ and σ' Two random schedule



Is any a EDD?

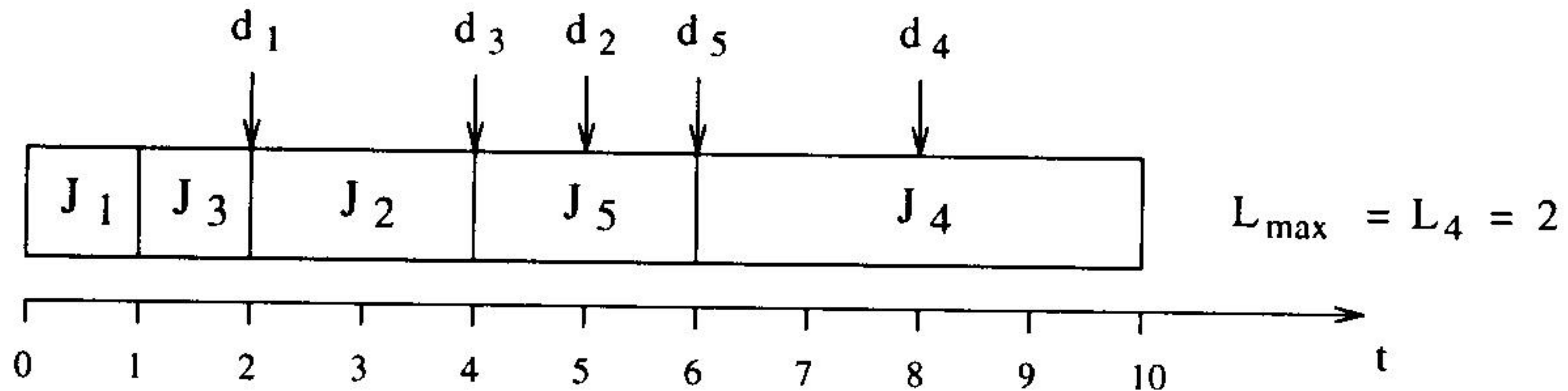
Is EDD (if any) optimal for Lateness?

Hard Real-Time Computing Systems, page 55,56

Earliest Deadline Due (EDD)

Example 2:

	J_1	J_2	J_3	J_4	J_5
C_i	1	2	1	4	2
d_i	2	5	4	8	6



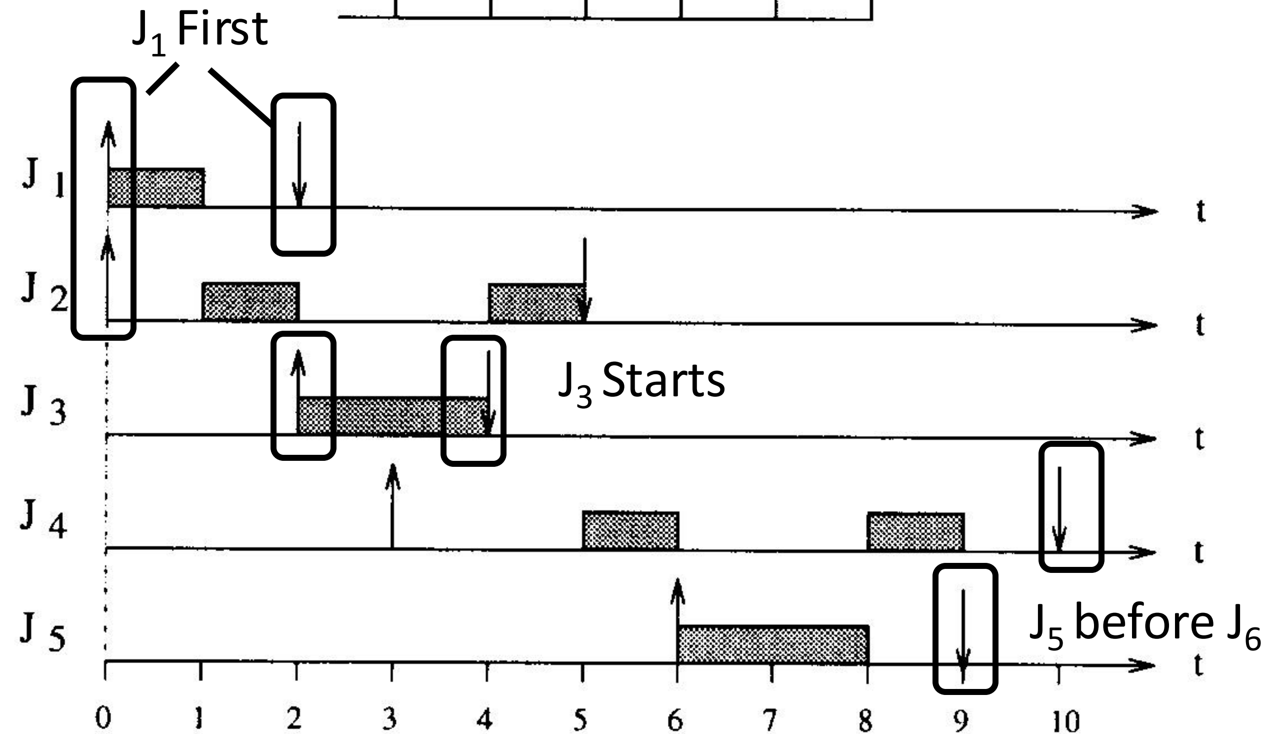
Earliest Deadline First (EDF)

Horn's rule: Given a set of *n* independent tasks with arbitrary arrival times, any algorithm that at any instant executes a task with **the earliest absolute deadline among the ready tasks is optimal** with respect to **minimizing the maximum lateness.**

Earliest Deadline First (EDF)

Example:

	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
C_i	1	2	2	2	2
d_i	2	5	4	10	9



Earliest Deadline First (EDF)

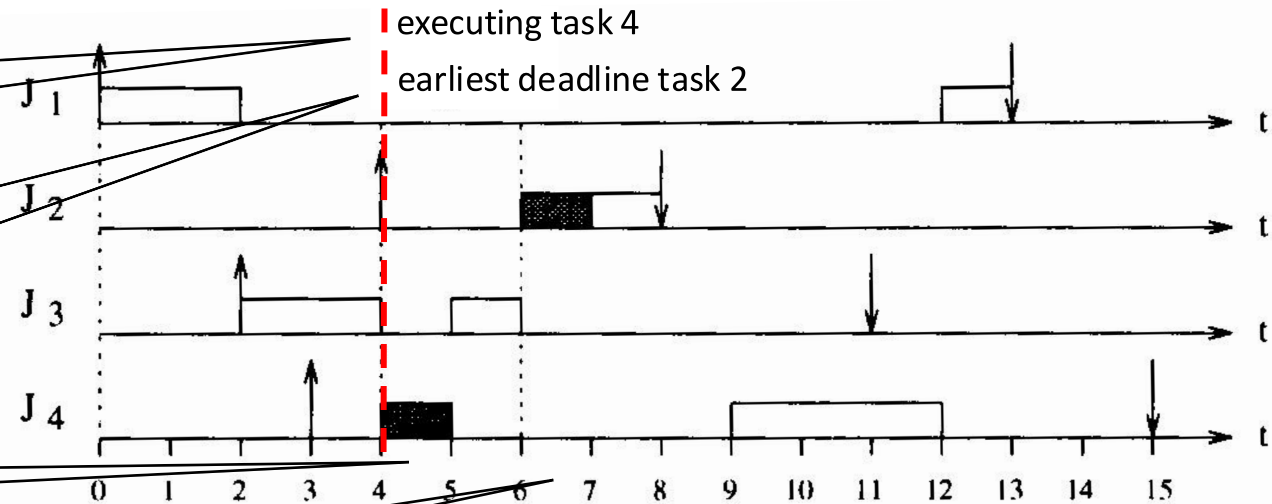
Horn's rule: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among the ready tasks is optimal with respect to minimizing the maximum lateness.

Concept of proof:

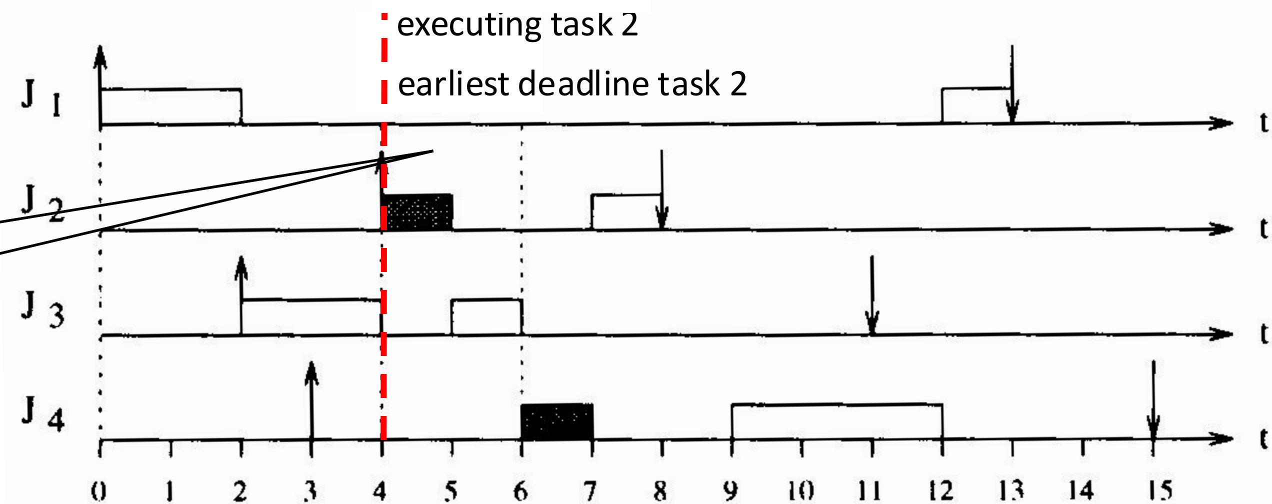
For each time interval $[t, t+1)$ it is verified, whether **the actual running task is the one with the earliest absolute deadline**. If this is not the case, the task with the **earliest absolute deadline is executed in this interval instead**. This operation cannot increase the maximum lateness.

Earliest Deadline First (EDF)

Till t=4 Earliest Deadline First (EDF)



(a)



(b)

Hard Real-Time Computing Systems, page 60

Earliest Deadline First (EDF)

remaining worst-case execution time of task k

Acceptance test:

- worst case finishing time of task i: $f_i = t + \sum_{k=1}^{i} c_k(t)$
- EDF guarantee condition: $\forall i = 1,\ldots,n \quad t + \sum_{k=1}^{i} c_k(t) \le d_i$
- algorithm:

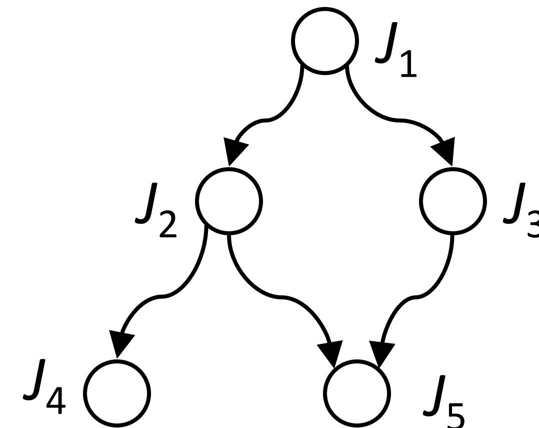
```
Algorithm: EDF_guarantee (J, J_new)
{       J'=J∪{J_new};  /* ordered by deadline */
        t = current_time();
        f_0 = t;
        for (each J_i∈J') {
                f_i = f_i-1 + c_i(t);
                if (f_i > d_i) return(INFEASIBLE);
        }
        return(FEASIBLE);
}
```

Hard Real-Time Computing Systems, page 64

What happen with dependences? Precedence Constraints

- *Precedence relations* between tasks can be described through an *acyclic directed graph* G where tasks are represented by nodes and precedence relations by arrows. G induces a partial order on the task set.

- There are different *interpretations* possible:
  - **All successors** of a task are activated (*concurrent task execution*). We will use this interpretation in the lecture.
  - One successor of a task is activated: *non-deterministic choice*.

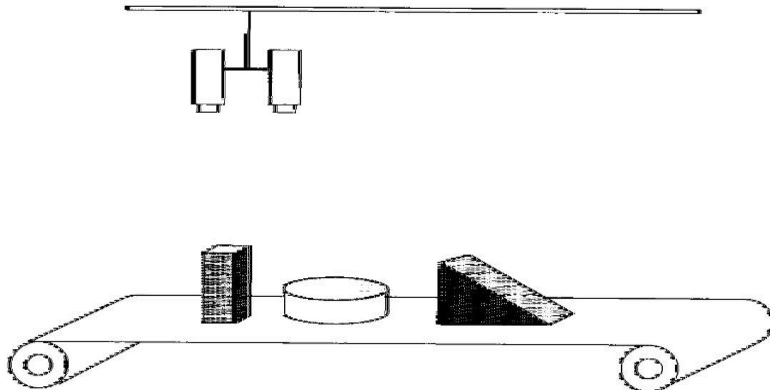


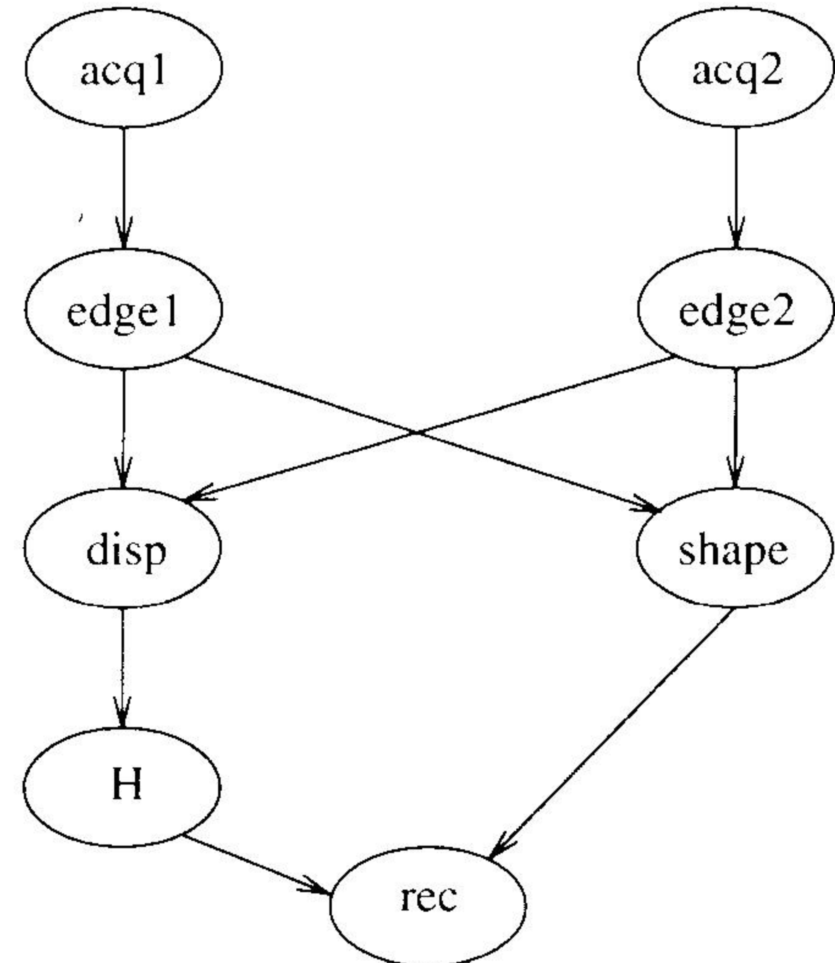
Hard Real-Time Computing Systems, page 29

Precedence Constraints

Example for concurrent activation:

- Image acquisition $acq1 \quad acq2$
- Low level image processing $edge1 \; edge2$
- Feature/contour extraction $shape$
- Pixel disparities $disp$
- Object size H
- Object recognition rec





Earliest Deadline First (EDF*) - With Precedence Constrains

- The problem of *scheduling a set of n tasks with precedence constraints* (concurrent activation) **can be solved in polynomial time complexity** if tasks are preemptable.

- The *EDF** algorithm determines a *feasible schedule* in **the case of tasks with precedence constraints if there exists one**.

- By **the modification it is guaranteed** that if *there exists a valid schedule* at all then
  - a task starts execution not earlier than its release time and not earlier than the finishing times of its predecessors (a task cannot preempt any predecessor)
  - all tasks finish their execution within their deadlines

Hard Real-Time Computing Systems, page 73

Earliest Deadline First (EDF*) With Precedence Constrains

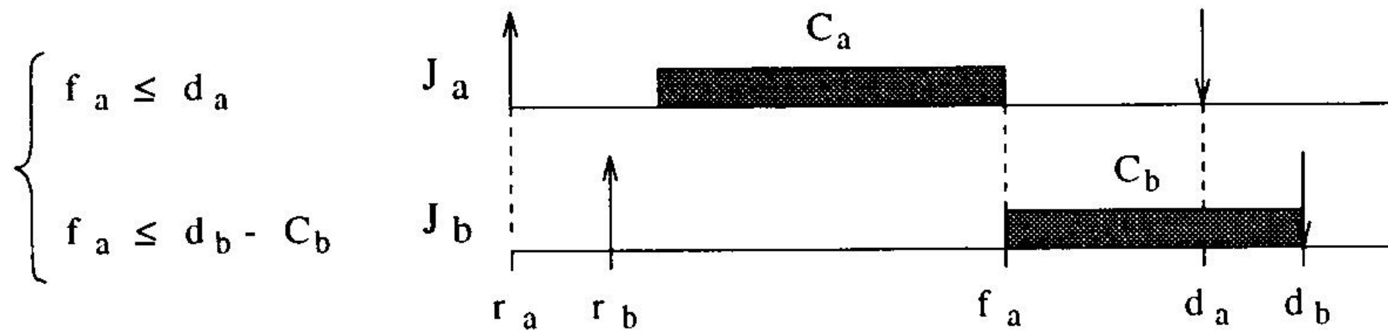
Goal of Chetto algo: Transform dependents tasks J in independent tasks J.*

Modification of deadlines:

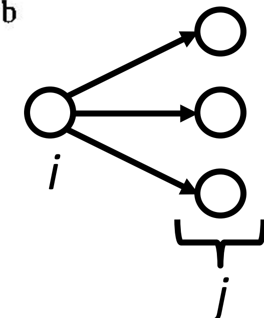
- Task must finish the execution time within its deadline.
- Task must not finish the execution later than the maximum start time of its successor.

task b depends on task a: $J_a \rightarrow J_b$

To satisfy the deadline of J_b, it must not start later than $d_b - C_b$.



- *Solution:* $$d_i^* = \min\left(d_i, \min\left(d_j^* - C_j : J_i \rightarrow J_j\right)\right)$$



In our example the deadline of Ja must be smaller or equal to d_b-C_b.

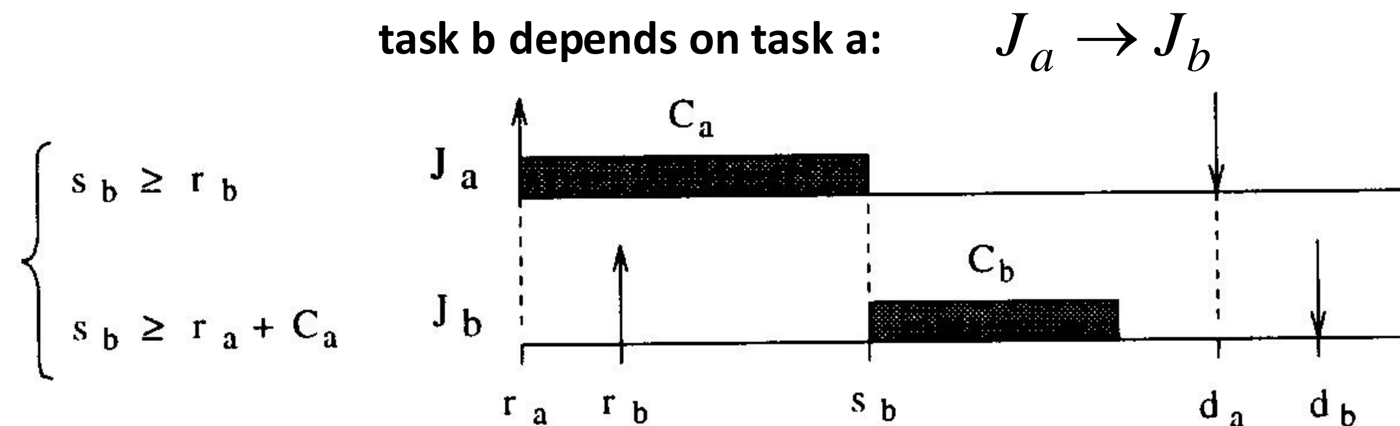
Hard Real-Time Computing Systems, page 73

Earliest Deadline First (EDF*)

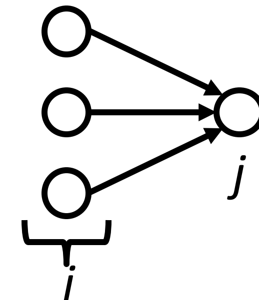
Modification of release times:

- Task must start the execution not earlier than its release time.
- Task must not start the execution earlier than the minimum finishing time of its predecessor.

task b depends on task a: $J_a \rightarrow J_b$



- *Solution:* $$r_j^* = \max\left(r_j, \max\left(r_i^* + C_i : J_i \rightarrow J_j\right)\right)$$



In out example Jb must have a release time that is larger or equal to ra + Ca.

Just a question that arises very often: Can EDF Guarantee the feasibility?

By modifying release times and deadlines it is not yet guaranteed that the subsequent EDF will find a feasible schedule. EDF may still fail to satisfy all deadlines. **But if it fails, we know that there exists no feasible schedule**

Earliest Deadline First (EDF*) (Self-Study Content)

Algorithm for modification of release times:

1. For any initial node of the precedence graph set $r_i^* = r_i$
2. Select a task j such that its release time has not been modified but the release times of all immediate predecessors i have been modified. If no such task exists, exit.
3. Set $r_j^* = \max\left(r_j, \max\left(r_i^* + C_i : J_i \to J_j\right)\right)$
4. Return to step 2

Algorithm for modification of deadlines:

1. For any terminal node of the precedence graph set $d_i^* = d_i$
2. Select a task i such that its deadline has not been modified but the deadlines of all immediate successors j have been modified. If no such task exists, exit.
3. Set $d_i^* = \min\left(d_i, \min\left(d_j^* - C_j : J_i \to J_j\right)\right)$
4. Return to step 2

Earliest Deadline First (EDF*) (Self-Study Content)

Proof concept:

- Show that if there exists a feasible schedule for the modified task set under EDF then the original task set is also schedulable. To this end, show that the original task set meets the timing constraints also. This can be done by using $r_i^* \geq r_i$, $d_i^* \leq d_i$; we only made the constraints stricter.
- Show that if there exists a schedule for the original task set, then also for the modified one. We can show the following: If there exists no schedule for the modified task set, then there is none for the original task set. This can be done by showing that no feasible schedule was excluded by changing the deadlines and release times.
- In addition, show that the precedence relations in the original task set are not violated. In particular, show that
  - a task cannot start before its predecessor and
  - a task cannot preempt its predecessor.

Real-Time Scheduling of Periodic Tasks

Overview

Periodic tasks are a core component of many real-time systems, such as those used in automotive, aerospace, or industrial automation.

Goals of Real-Time Scheduling

- The primary goal is to ensure that all tasks meet their deadlines while maximizing system utilization. This requires:
- **Feasibility Analysis**: Determining if all tasks can be scheduled without missing deadlines.

Table of some known *preemptive scheduling algorithms for periodic tasks*:

	Deadline equals period	Deadline smaller than period
static priority	RM (rate-monotonic)	DM (deadline-monotonic)
dynamic priority	EDF	EDF*

Model of Periodic Tasks (recap)

- *Examples:* sensory data acquisition, low-level actuation, control loops, action planning and system monitoring.
- When an *application* consists of several concurrent periodic tasks with individual timing constraints, the OS has to guarantee that each periodic instance is regularly activated at its proper rate and is completed within its deadline.

- Definitions:

  Γ : denotes a set of periodic tasks

  τ_i : denotes a periodic task

  $\tau_{i,j}$: denotes the j-th instance of task i

  $r_{i,j},\ s_{i,j},\ f_{i,j},\ d_{i,j}$: denote the release time, start time, finishing time, absolute deadline of the jth instance of task i

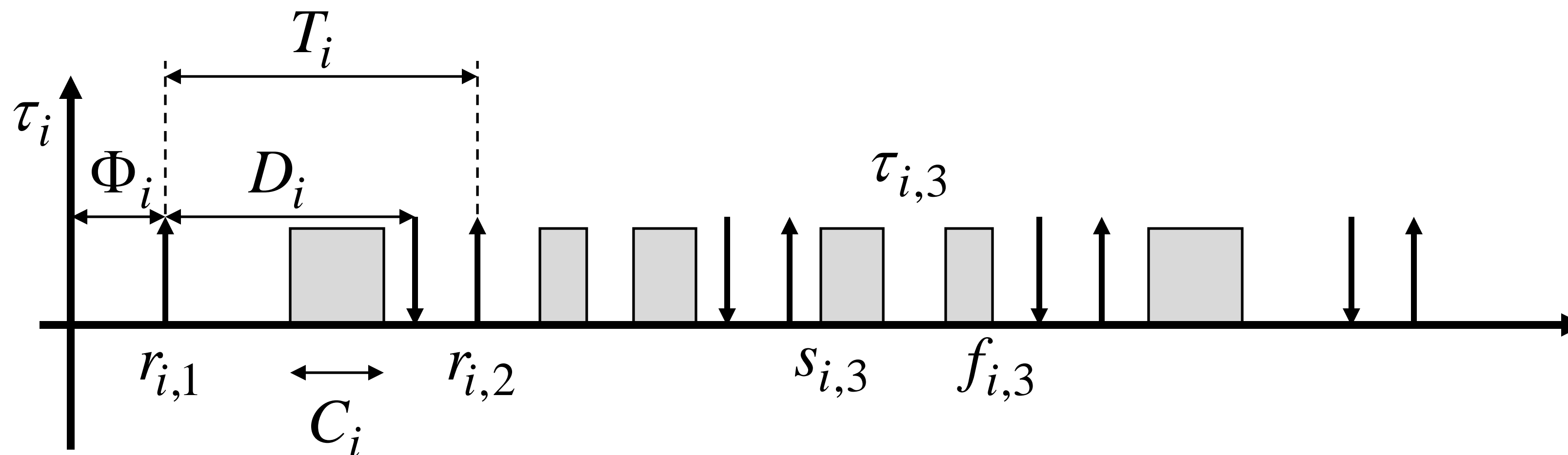
  Φ_i : denotes the phase of task i (release time of its first instance)

  D_i : denotes the relative deadline of task i

  T_i : denotes the period of task i

Model of Periodic Tasks (recap)

- *The following hypotheses are assumed on the tasks (continued):*
  - All periodic tasks are *independent*; that is, there are no precedence relations and no resource constraints.
  - *No task can suspend itself*, for example on I/O operations.
  - All tasks are *released as soon as they arrive*.
  - All *overheads* in the OS kernel are assumed to be *zero*.
  - *Example:*



Model of Periodic Tasks (recap)

- *The following hypotheses are assumed on the tasks:*
  - The instances of a periodic task are *regularly activated at a constant rate*. The interval T_i between two consecutive activations is called period. The release times satisfy

$$r_{i,j} = \Phi_i + (j-1)T_i$$

  - All instances have the *same worst case execution time* C_i
  - All instances of a periodic task have the *same relative deadline* D_i. Therefore, the absolute deadlines satisfy

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$

  - Often, the relative deadline equals the period $D_i = T_i$ (*implicit deadline*), and therefore

$$d_{i,j} = \Phi_i + jT_i$$

Rate Monotonic Scheduling (RM)

- *Assumptions:*
  - Task priorities are assigned to tasks before execution and do not change over time (**static priority assignment**).
  - **RM is intrinsically preemptive**: the currently executing job is preempted by a job of a task with higher priority.
  - Deadlines equal the periods $D_i = T_i$.

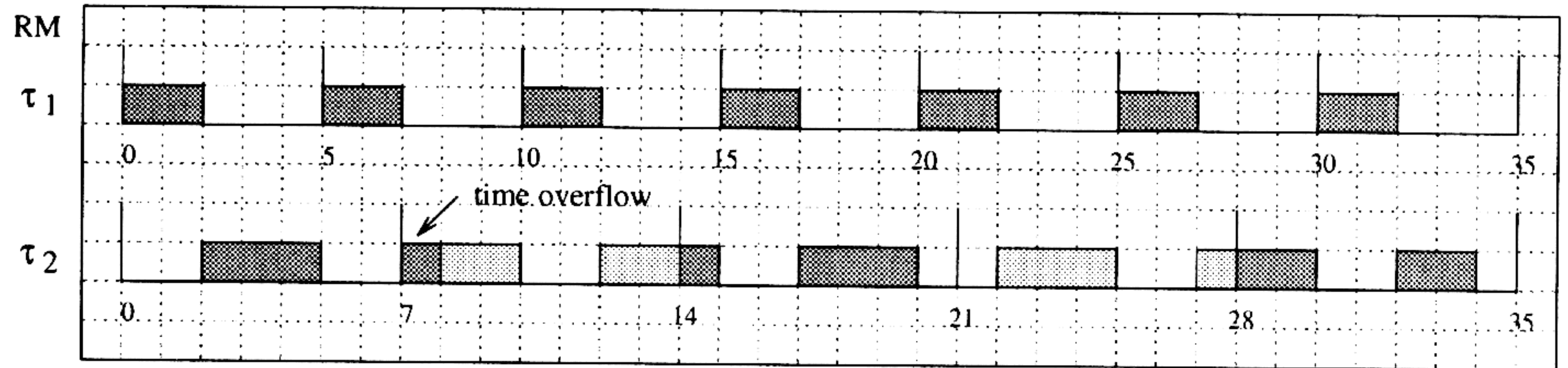
Rate-Monotonic Scheduling Algorithm: Each task is assigned a priority. Tasks with higher request rates **(that is with shorter periods) will have higher priorities**. Jobs of tasks with higher priority interrupt jobs of tasks with lower priority.

Hard Real-Time Computing Systems, page 86

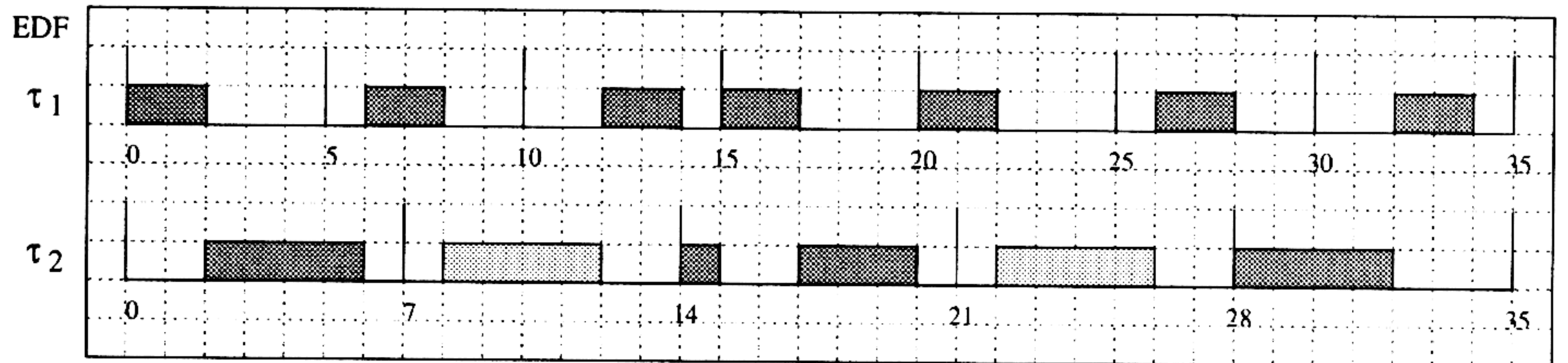
Periodic Tasks

Example: 2 tasks, deadlines = periods, utilization = 97%

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$



(a)



(b)

Rate Monotonic Scheduling (RM)

Optimality: RM is **optimal among all fixed-priority assignments** in the sense that no other fixed-priority algorithm can schedule a task set that cannot be scheduled by RM.

- The *proof* is done by considering several cases that may occur, but the main ideas are as follows:
  - A *critical instant* for any task occurs whenever the task is released simultaneously with all higher priority tasks. **The tasks schedulability can easily be checked at their critical instants**. If all tasks are feasible at their critical instant, then the task set is schedulable in any other condition.
  - Show that, given two periodic tasks, if the schedule is feasible by an arbitrary priority assignment, then it is also feasible by RM.
  - Extend the result to a set of n periodic tasks.

Hard Real-Time Computing Systems, page 87

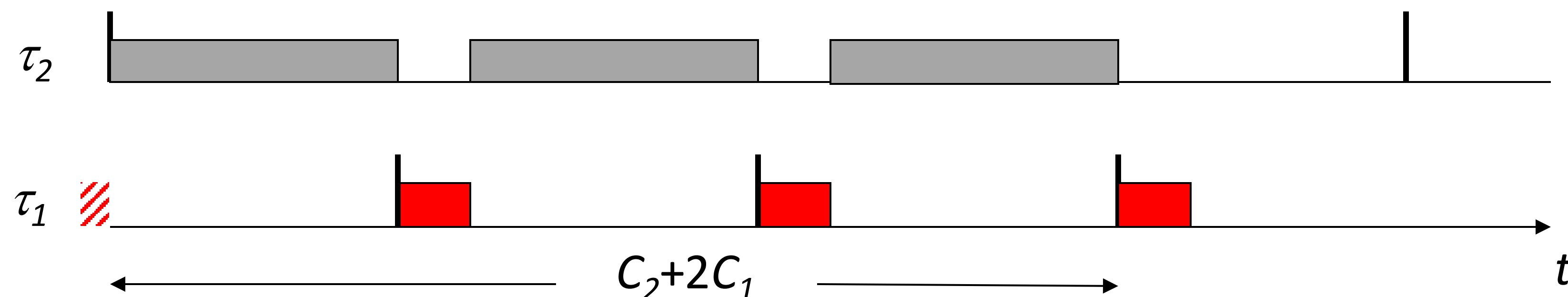
Proof of Critical Instance

Definition: A critical instant of a task is the time at which the release of a job will produce the largest response time. In other words, the largest difference between release time and finishing time.

Lemma: For any task, the critical instant occurs if a job is simultaneously released with all higher priority jobs.

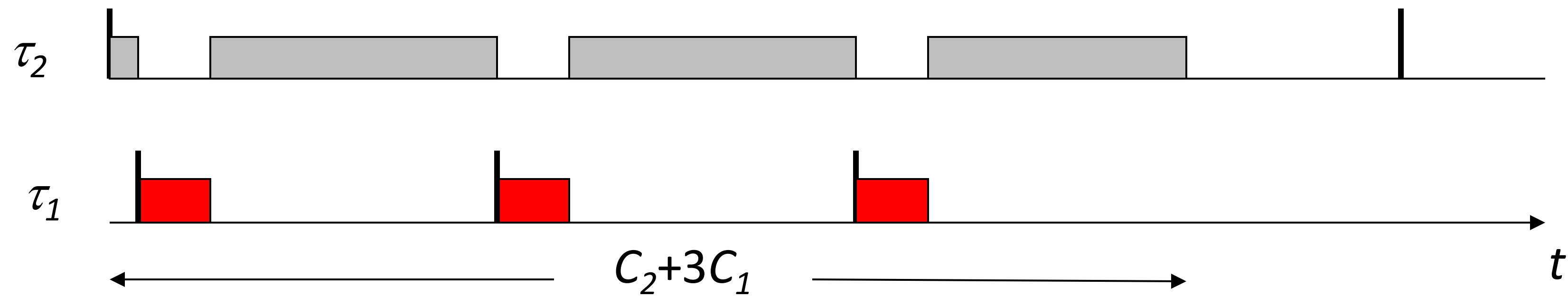
Proof sketch: Start with 2 tasks τ_1 and τ_2 .

Response time of a job of τ_2 is delayed by jobs of τ_1 of higher priority:



Proof of Critical Instance

Delay may increase if τ_1 starts earlier:



Maximum delay achieved if τ_2 and τ_1 start simultaneously.

Repeating the argument for all higher priority tasks of some task τ_2 :

> The worst-case response time of a job occurs when it is released simultaneously with all higher-priority jobs.

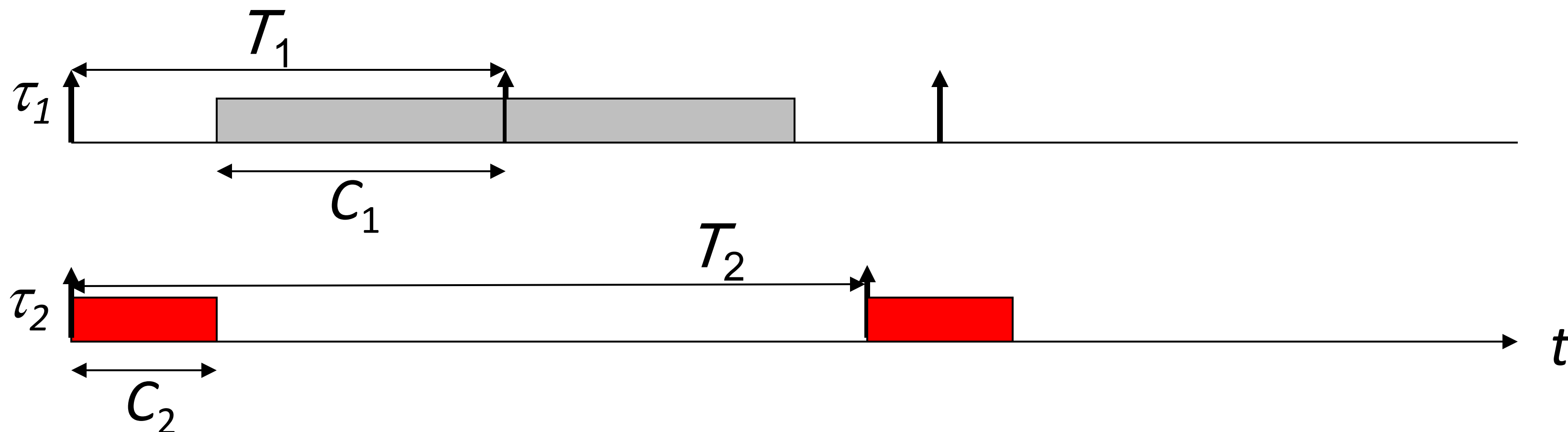
Proof of RM Optimality (2 Tasks)

We have two tasks τ_1, τ_2 with periods $T_1 < T_2$.

Define $F = \lfloor T_2/T_1 \rfloor$: the number of periods of τ_1 **fully** contained in T_2

Consider two cases A and B:

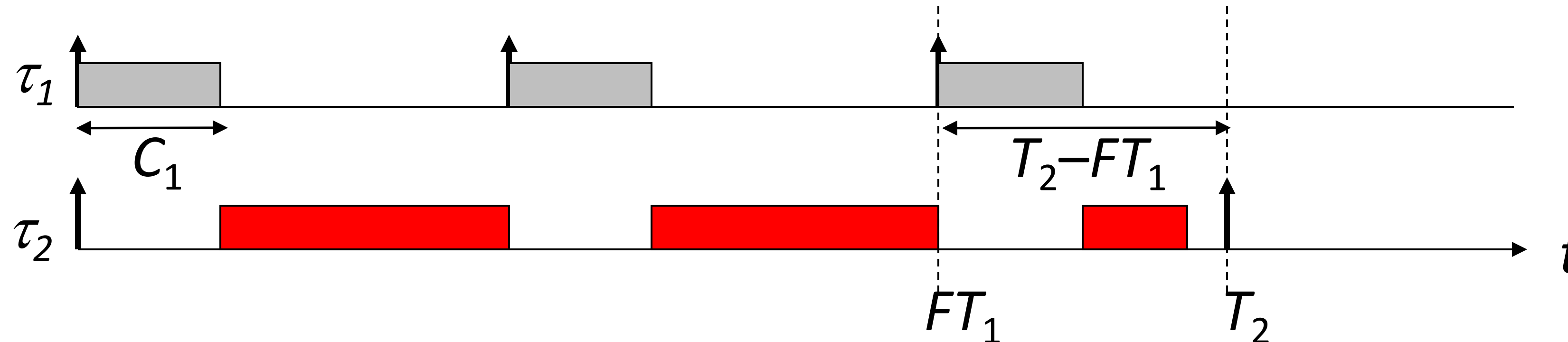
Case A: Assume RM is not used → prio(τ_2) is highest:



Schedule is feasible if $C_1 + C_2 \leq T_1$ and $C_2 \leq T_2$ (A)

Proof of RM Optimality (2 Tasks) (Self-Study Content)

Case B: Assume RM **is** used → prio(τ_1) is highest:



Schedulable is feasible if

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2 \text{ and } C_1 \leq T_1 \qquad \text{(B)}$$

We need to show that (A) ⇒ (B): $C_1 + C_2 \leq T_1 \Rightarrow C_1 \leq T_1$

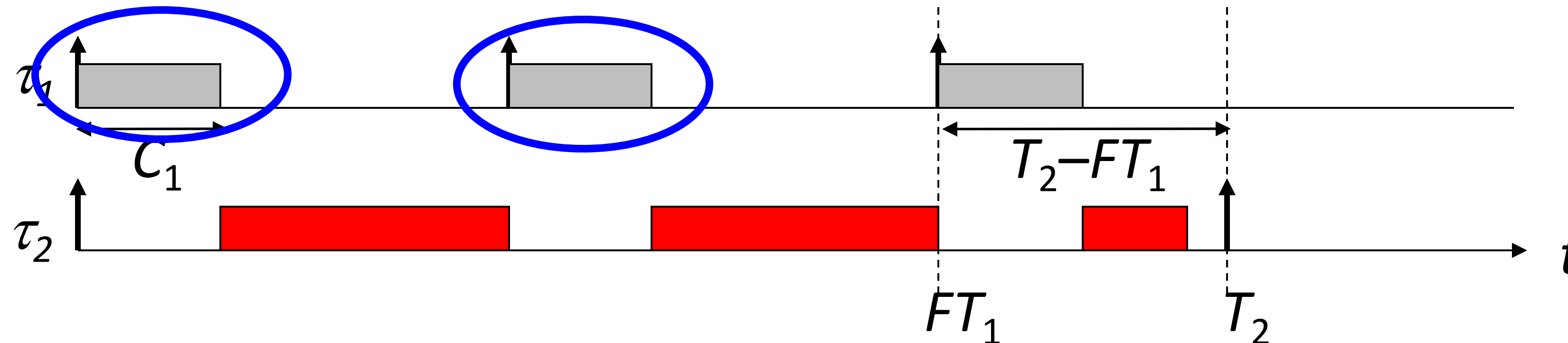
$$C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FC_1 + FC_2 \leq F\,T_1 \Rightarrow$$

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq F\,T_1 + \min(T_2 - FT_1, C_1) \leq \min(T_2, C_1 + FT_1) \leq T_2$$

Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

Proof of RM Optimality (2 Tasks) (Self-Study Content)

Case B: Assume RM **is** used → prio(τ_1) is highest:



Schedulable is feasible if

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2 \text{ and } C_1 \leq T_1 \quad \text{(B)}$$

We need to show that (A) ⇒ (B): $C_1 + C_2 \leq T_1 \Rightarrow C_1 \leq T_1$

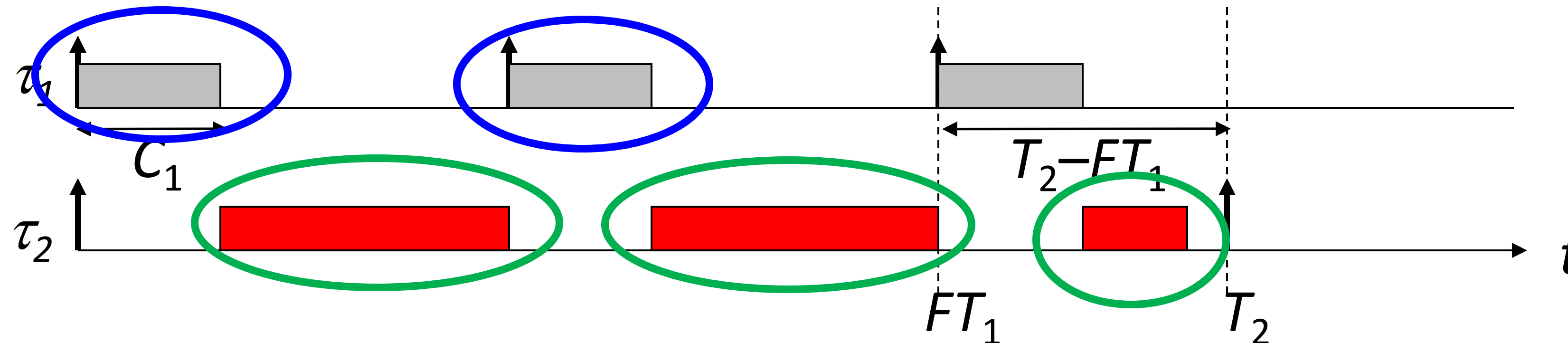
$$C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FC_1 + FC_2 \leq F T_1 \Rightarrow$$

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq F T_1 + \min(T_2 - FT_1, C_1) \leq \min(T_2, C_1 + FT_1) \leq T_2$$

Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

Proof of RM Optimality (2 Tasks) (Self-Study Content)

Case B: Assume RM **is** used → prio(τ_1) is highest:



Schedulable is feasible if

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2 \text{ and } C_1 \leq T_1 \quad \text{(B)}$$

We need to show that (A) ⇒ (B): $C_1 + C_2 \leq T_1 \Rightarrow C_1 \leq T_1$

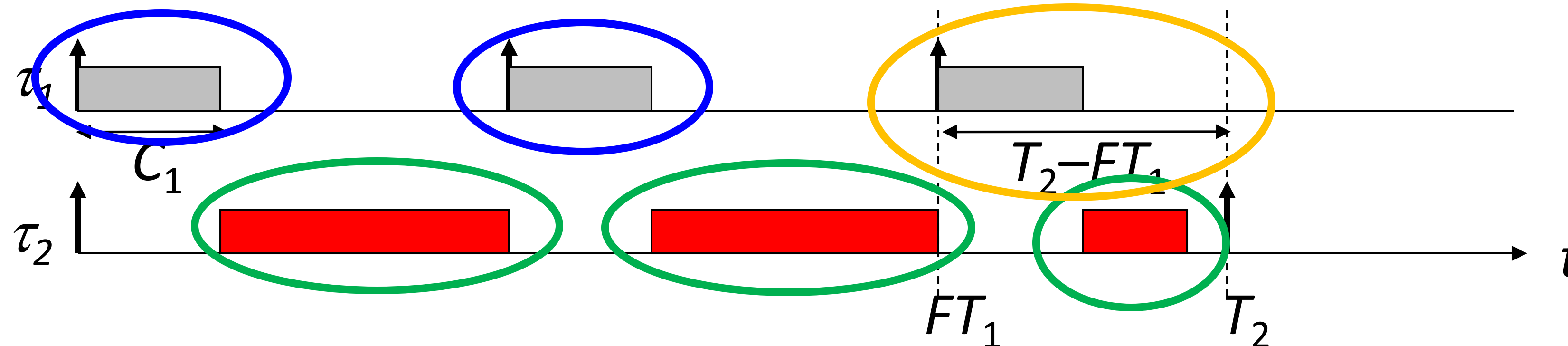
$$C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FC_1 + FC_2 \leq F\,T_1 \Rightarrow$$

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq F\,T_1 + \min(T_2 - FT_1, C_1) \leq \min(T_2, C_1 + FT_1) \leq T_2$$

Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

Proof of RM Optimality (2 Tasks) (Self-Study Content)

Case B: Assume RM **is** used → prio(τ_1) is highest:



Schedulable is feasible if

$$FC_1+C_2+\min(T_2-FT_1, C_1) \le T_2 \text{ and } C_1 \le T_1 \quad \text{(B)}$$

We need to show that (A) ⇒ (B): $C_1+C_2 \le T_1 \Rightarrow C_1 \le T_1$

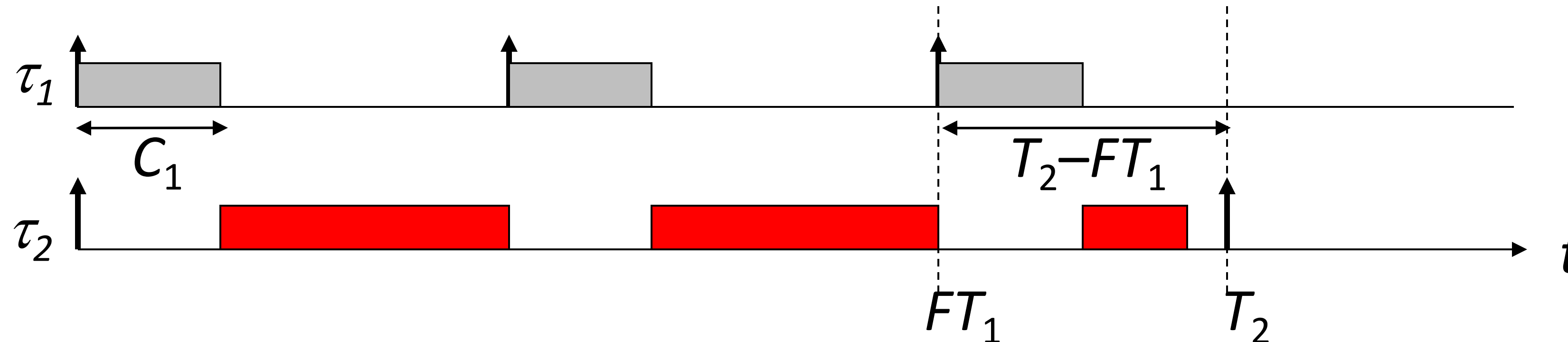
$$C_1+C_2 \le T_1 \Rightarrow FC_1+C_2 \le FC_1+FC_2 \le FT_1 \Rightarrow$$

$$FC_1+C_2+\min(T_2-FT_1, C_1) \le FT_1 + \min(T_2-FT_1, C_1) \le \min(T_2, C_1+FT_1) \le T_2$$

Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

Proof of RM Optimality (2 Tasks) (Self-Study Content)

Case B: Assume RM **is** used → prio(τ_1) is highest:



Schedulable is feasible if

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2 \text{ and } C_1 \leq T_1 \qquad \text{(B)}$$

We need to show that (A) ⇒ (B): $C_1 + C_2 \leq T_1 \Rightarrow C_1 \leq T_1$

$$C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FC_1 + FC_2 \leq F\,T_1 \Rightarrow$$

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq F\,T_1 + \min(T_2 - FT_1, C_1) \leq \min(T_2, C_1 + FT_1) \leq T_2$$

Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

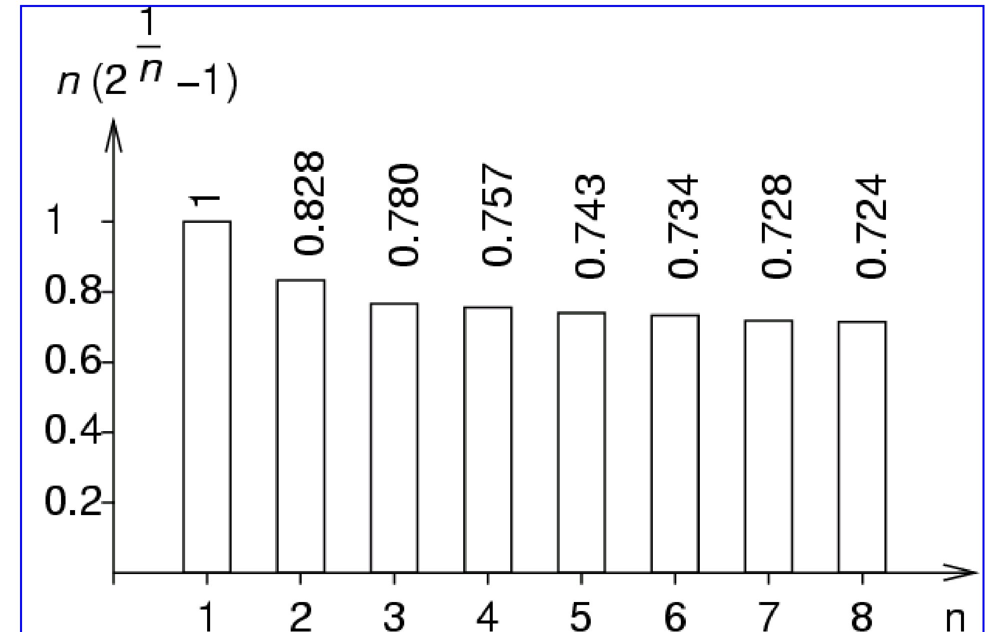
Rate Monotonic Scheduling (RM)

Schedulability analysis: A set of periodic tasks (n) is schedulable with RM if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

This condition is sufficient but not necessary.

The term $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ denotes the *processor utilization factor U* which is the fraction of processor time spent in the execution of the task set.

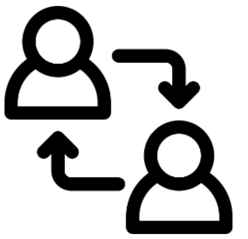


For $n = 2$, the RMS utilization bound is:

$$U_{bound} = 2 \times (2^{1/2} - 1) \approx 2 \times 0.4142 = 0.8284$$

A theoretical upper limit on CPU utilization under Rate-Monotonic Scheduling (RM).

Example to do together.



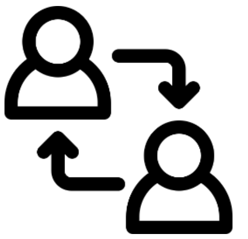
- **Question:**
  Suppose you have two tasks:

• **Task 1:** C1=1ms, T1=4 ms

• **Task 2:** C2=1 ms, T2=5 ms

- Calculate the total CPU utilization and determine if these tasks are schedulable under RM.
- **Calculate:**
  - Calculate the individual utilizations.
  - Sum them up.
  - Compare the total with the RM utilization bound for n=2 (number of tasks).

Solution



Step 1: Calculate Individual Utilizations

For **Task 1**:

$$U_1 = \frac{C_1}{T_1} = \frac{1}{4} = 0.25$$

$U_{total} = U_1 + U_2 = 0.25 + 0.20 = 0.45$

For **Task 2**:

$$U_2 = \frac{C_2}{T_2} = \frac{1}{5} = 0.20$$

For $n = 2$, the RMS utilization bound is:

$$U_{bound} = 2 \times (2^{1/2} - 1) \approx 2 \times 0.4142 = 0.8284$$

Task 1 and **Task 2** are schedulable under RM.

Proof of Utilization Bound (2 Tasks) (Self-Study Content)

We have two tasks τ_1, τ_2 with periods $T_1 < T_2$.
Define $F = \lfloor T_2/T_1 \rfloor$: number of periods of τ_1 **fully** contained in T_2

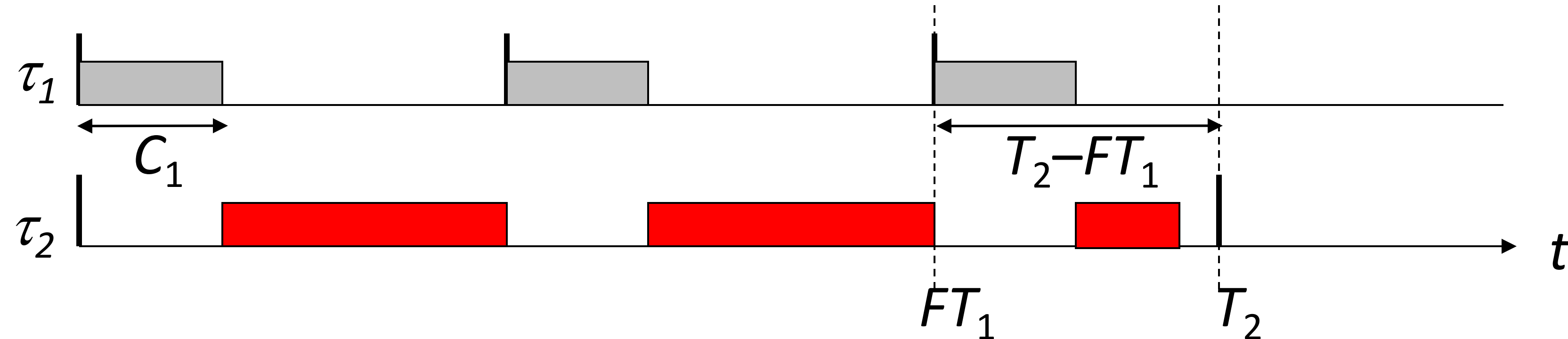
Proof Concept: **Compute upper bound on utilization *U*** such that the task set is still schedulable:

- assign priorities according to RM;
- compute upper bound U_{up} by increasing the computation time C_2 to just meet the deadline of τ_2; we will determine this limit of C_2 using the results of the RM optimality proof.
- minimize upper bound with respect to other task parameters in order to find the utilization below which the system is definitely schedulable.

Hard Real-Time Computing Systems, page 87

Proof of Utilization Bound (2 Tasks)
(Self-Study Content)

As before:



Schedulable if $FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2$ and $C_1 \leq T_1$

Utilization:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{T_2 - FC_1 - \min\{T_2 - FT_1, C_1\}}{T_2}$$

$$= 1 + \frac{C_1(T_2 - FT_1) - T_1 \min\{T_2 - FT_1, C_1\}}{T_1 T_2}$$

Proof of Utilization Bound (2 Tasks) (Self-Study Content)

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{T_2 - FC_1 - \min\{T_2 - FT_1, C_1\}}{T_2}$$

$$= 1 + \frac{C_1(T_2 - FT_1) - T_1 \min\{T_2 - FT_1, C_1\}}{T_1 T_2}$$

Proof of Utilization Bound (2 Tasks) (Self-Study Content)

Minimize utilization bound w.r.t C_1:

- If $C_1 \leq T_2 - FT_1$ then U decreases with increasing C_1
- If $T_2 - FT_1 \leq C_1$ then U decreases with decreasing C_1
- Therefore, minimum U is obtained with $C_1 = T_2 - FT_1$:

$$U = 1 + \frac{(T_2 - FT_1)^2 - T_1(T_2 - FT_1)\}}{T_1 T_2}$$

$$= 1 + \frac{T_1}{T_2}((\frac{T_2}{T_1} - F)^2 - (\frac{T_2}{T_1} - F))$$

We now need to minimize w.r.t. $G = T_2/T_1$ where $F = \lfloor T_2/T_1 \rfloor$ and $T_1 < T_2$. As F is integer, we first suppose that it is independent of $G = T_2/T_1$. Then we obtain

$$U = \frac{T_1}{T_2}((\frac{T_2}{T_1} - F)^2 + F)) = \frac{(G-F)^2 + F}{G}$$

Proof of Utilization Bound (2 Tasks) (Self-Study Content)

Minimizing *U* with respect to *G* yields

$$2G(G-F)-(G-F)^2-F=G^2-(F^2+F)=0$$

If we set *F* = 1, then we obtain

$$G=\frac{T_2}{T_1}=\sqrt{2} \qquad U=2(\sqrt{2}-1)$$

It can easily be checked, that all other integer values for *F* lead to a larger upper bound on the utilization.

Deadline Monotonic Scheduling (DM)

- **Assumptions** are as in rate monotonic scheduling, but *deadlines may be smaller than the period*, i.e.

$$C_i \leq D_i \leq T_i$$

Algorithm: Each task is assigned a priority. Tasks with smaller relative deadlines will have higher priorities. Jobs with higher priority interrupt jobs with lower priority.

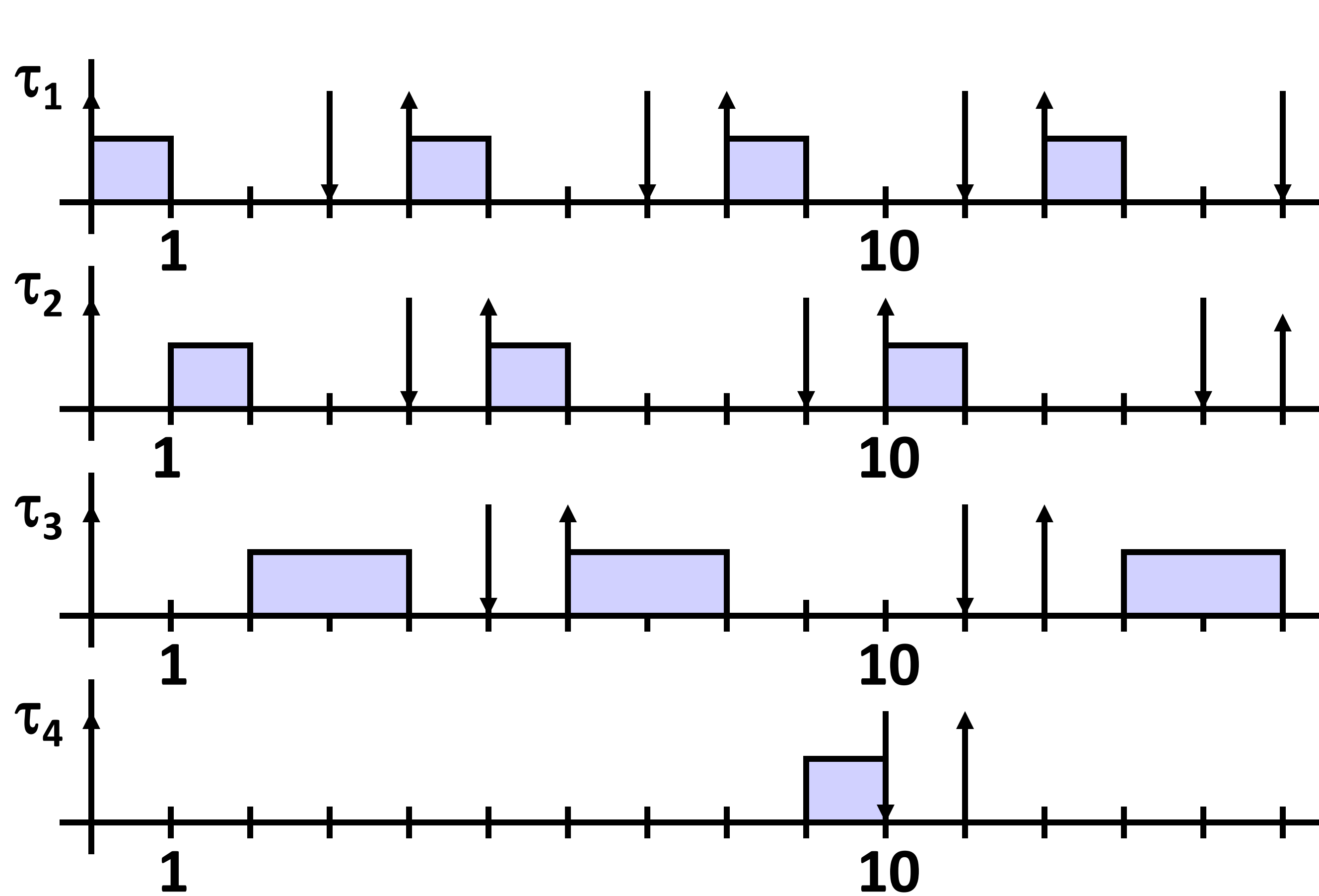
- *Schedulability Analysis:* A set of periodic tasks is schedulable with DM if

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{1/n} - 1\right)$$

This condition is sufficient but not necessary (in general).

Deadline Monotonic Scheduling (DM) - Example

$U = 0.874$ $\qquad \sum_{i=1}^{n} \frac{C_i}{D_i} = 1.08 > n\left(2^{1/n} - 1\right) = 0.757$



	C_i	D_i	T_i
τ_1	1	3	4
τ_2	1	4	5
τ_3	2	5	6
τ_4	1	10	11

Deadline Monotonic Scheduling (DM)

There is also a *necessary and sufficient schedulability test* which is computationally more involved. It is based on the following observations:

- The *worst-case processor demand* occurs when all tasks are released simultaneously; that is, at their critical instances.
- For each task *i*, the sum of its processing time and the *interference* imposed by higher priority tasks must be less than or equal to D_i .
- A measure of the *worst case interference* for task i can be computed as the sum of the processing times of all higher priority tasks released before some time t where tasks are ordered according to $m < n \Leftrightarrow D_m < D_n$:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j$$

Deadline Monotonic Scheduling (DM)

- The *longest response time* R_i of a job of a periodic task i is computed, at the critical instant, as the sum of its computation time and the interference due to preemption by higher priority tasks:

$$R_i = C_i + I_i$$

- Hence, the schedulability test needs to compute the smallest R_i that satisfies

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

  for all tasks i. **Then,** $R_i \leq D_i$ **must hold for all tasks i.**
- It can be shown that this *condition is necessary and sufficient*.

Deadline Monotonic Scheduling (DM)

The longest response times R_i of the periodic tasks *i* can be computed iteratively by the following algorithm:

```
Algorithm: DM_guarantee (Γ)
Sort tasks according to: m < n <=> Dm < Dn
{       for (each τi∈Γ){
                I = 0;
                do {
                        R = I + Ci;
                        if (R > Di) return(UNSCHEDULABLE);
                        I = ∑j=1,…,(i-1)⌈R/Tj⌉ Cj;
                } while (I + Ci > R);
        }
        return(SCHEDULABLE);
}
```

DM Example – Sufficient and Necessary Proof

	C_i	D_i	T_i
τ_1	1	3	4
τ_2	1	4	5
τ_3	2	5	6
τ_4	1	10	11

Example:

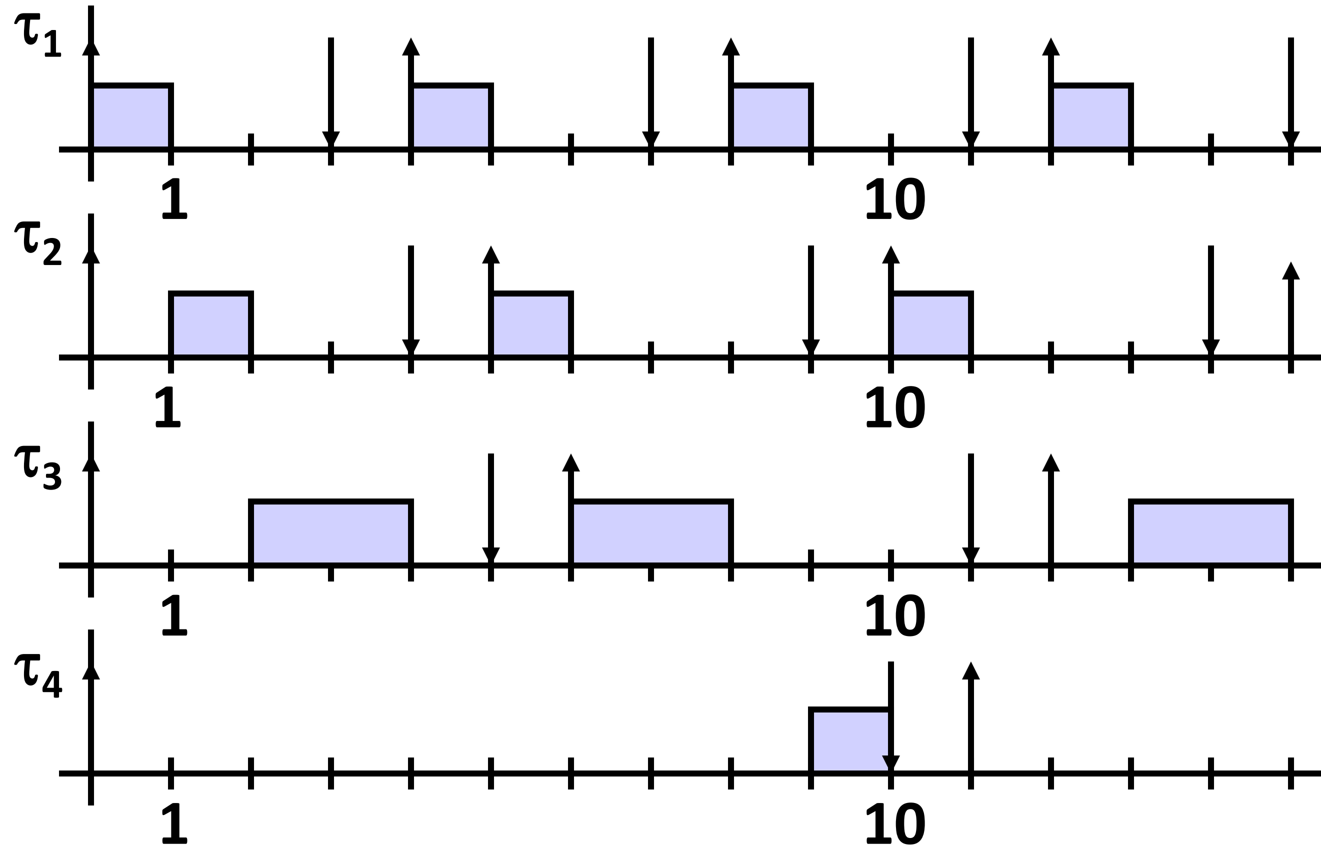
- ***Deadlines sorted: $D_1<D_2<D_3<D_4$***
- Task 1: I = 0,
  - Step 1 R = I + C_1 = 1; if (1>3=false); I = 0 (no predecessor task j); while (0+1>1=false) → exit
- Task 2: I =0,
  - Step 1 R = I + C_2 = 1; if (1>4=false); I = $\lceil 1/4 \rceil 1$ = 1; while(1+1>1=true)
  - Step 2 R = I + C_2 = 2 ; if (2>4=false); I = $\lceil 2/4 \rceil 1$ = 1; while (1+1>2=false) → exit
- Task 3: I =0,
  - Step 1. R = I + C_3 = 0 + 2 = 2; if(2>5=false); I = $\lceil 2/4 \rceil 1 + \lceil 2/5 \rceil 1$ = 2; while(2+2>2=true)
  - Step 2 R = I + C_3 = 2 + 2 = 4; if(4>5=false); I = $\lceil 4/4 \rceil 1 + \lceil 4/5 \rceil 1$ = 2; while(2+2>4=false) → exit
- Task 4: I = 0,
  - Step 1 R = I + C_4 = 0 + 1 = 1; if(1>10=false); I = $\lceil 1/4 \rceil 1 + \lceil 1/5 \rceil 1 + \lceil 1/6 \rceil 2$ = 4; while(4+1>1=true)
  - Step 2 R = I + C_4 = 4 + 1 = 5; if(5>10=false); I = $\lceil 5/4 \rceil 1 + \lceil 5/5 \rceil 1 + \lceil 5/6 \rceil 2$ = 5; while(5+1>5=true)
  - Step 3 R = I + C_4 = 5 + 1 = 6; if(6>10 = false); I = $\lceil 6/4 \rceil 1 + \lceil 6/5 \rceil 1 + \lceil 6/6 \rceil 2$ = 6; while(6+1>6=true)
  - Step 4 R = I + C_4 = 6 + 1 = 7; if(7>10 = false); I = $\lceil 7/4 \rceil 1 + \lceil 7/5 \rceil 1 + \lceil 7/6 \rceil 2$ = 8; while(8+1>7=true)
  - Step 5 R = I + C_4 = 8 + 1 = 9; if(9>10 = false); I = $\lceil 9/4 \rceil 1 + \lceil 9/5 \rceil 1 + \lceil 9/6 \rceil 2$ = 9; while(9+1>9=true)
  - Step 6 R = I + C_4 = 9 + 1 = 10; if(10>10 = false); I = $\lceil 10/4 \rceil 1 + \lceil 10/5 \rceil 1 + \lceil 10/6 \rceil 2$ = 9; while(9+1>10=false) → exit

Schedulable with DM

DM Example

$U = 0.874$

$$\sum_{i=1}^{n} \frac{C_i}{D_i} = 1.08 > n\left(2^{1/n} - 1\right) = 0.757$$



EDF Scheduling (earliest deadline first)

- *Assumptions:*
  - **Dynamic priority assignment**
  - intrinsically preemptive

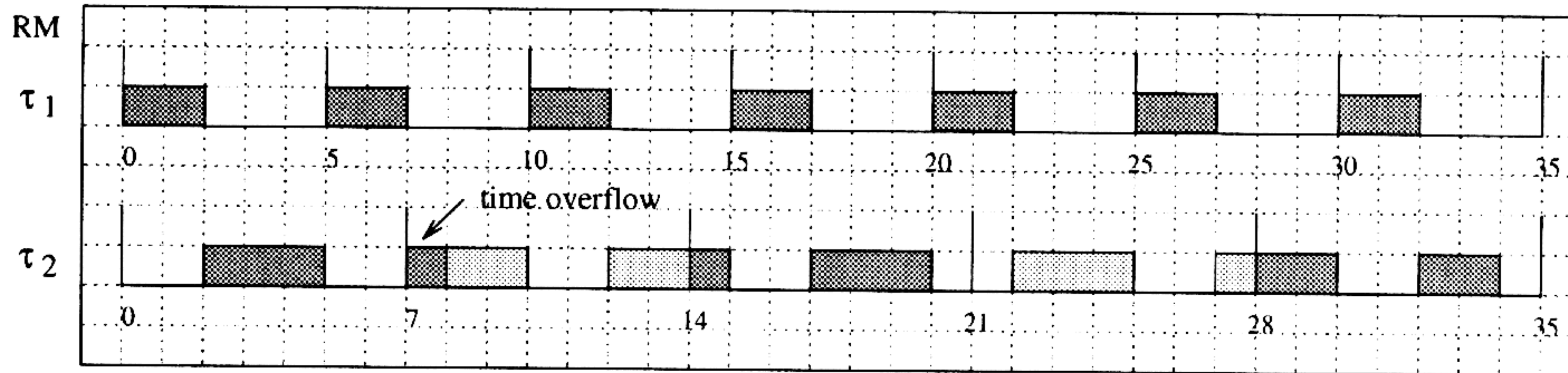
- *Algorithm:* The currently executing task is preempted whenever another periodic instance with earlier deadline becomes active.

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$

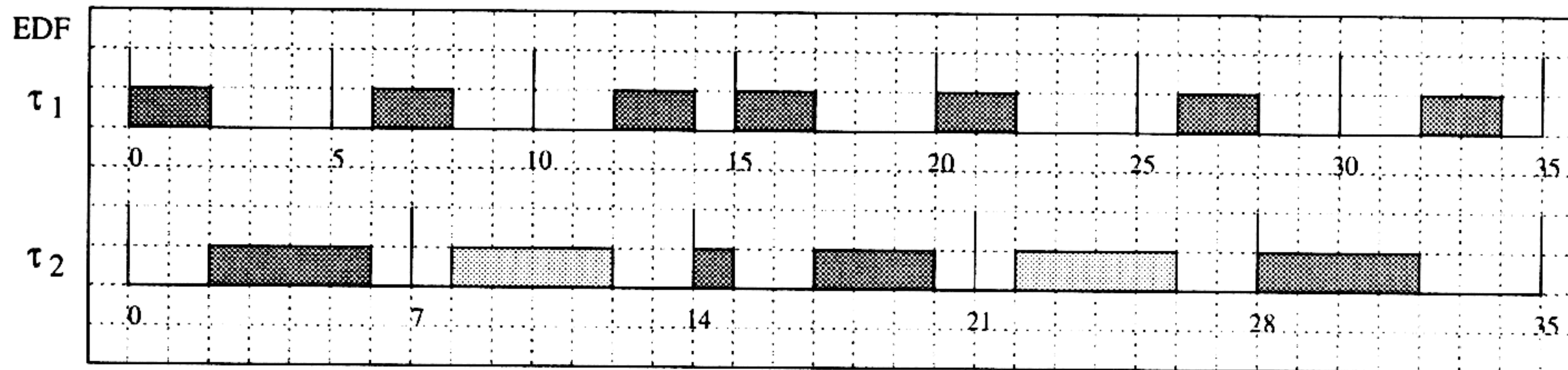
- *Optimality:* No other algorithm can schedule a set of periodic tasks if the set that can not be scheduled by EDF.
- The proof is simple and follows that of the aperiodic case.

Periodic Tasks

Example: 2 tasks, deadlines = periods, utilization = 97%



(a)



(b)

EDF Scheduling

A *necessary and sufficient schedulability test* for $D_i = T_i$:

A set of periodic tasks is schedulable with EDF if and only if $\sum_{i=1}^{n} \frac{C_i}{T_i} = U \leq 1$

The term $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ denotes the *average processor utilization.*

EDF Scheduling

- *If the deadline was missed* at t_2 then define t_1 as a time before t_2 such that (a) the processor is continuously busy in [t_1, t_2] and (b) the processor only executes tasks that have their arrival time AND their deadline in [t_1, t_2].
- *Why does such a time t_1 exist?* We find such a t_1 by starting at t_2 and going backwards in time, always ensuring that the processor only executed tasks that have their deadline before or at t_2 :
  - Because of EDF, the processor will be busy shortly before t_2 and it executes on the task that has deadline at t_2.
  - Suppose that we reach a time such that shortly before the processor works on a task with deadline after t_2 or the processor is idle, then we found t_1: We know that there is no execution on a task with deadline after t_2 .
    - But it could be in principle, that a task that arrived before t_1 is executing in [t_1, t_2].
    - If the processor is idle before t_1, then this is clearly not possible due to EDF (the processor is not idle, if there is a ready task).
    - If the processor is not idle before t_1, this is not possible as well. Due to EDF, the processor will always work on the task with the closest deadline and therefore, once starting with a task with deadline after t_2 all task with deadlines before t_2 are finished.

What Did You Learn?

- ✓ Schedule and Timing Terms and Metrics
- ✓ Aperiodic Task Scheduling
  - ✓ Equal Arrival Times, Non-Preemptive,
    - ✓ Independent Tasks: EDD
  - ✓ Arbitrary Arrival Times, Preemptive
    - ✓ Independent Tasks: EDF, Dependent Tasks: EDF*
- ✓ Preemptive Periodic Task Scheduling
  - ✓ Static Priority
    - ✓ Deadline=Period: RM, Deadline ≤ Period: DM
  - ✓ Dynamic Priority
    - ✓ Deadline=Period: EDF, Deadline ≤ Period: EDF*