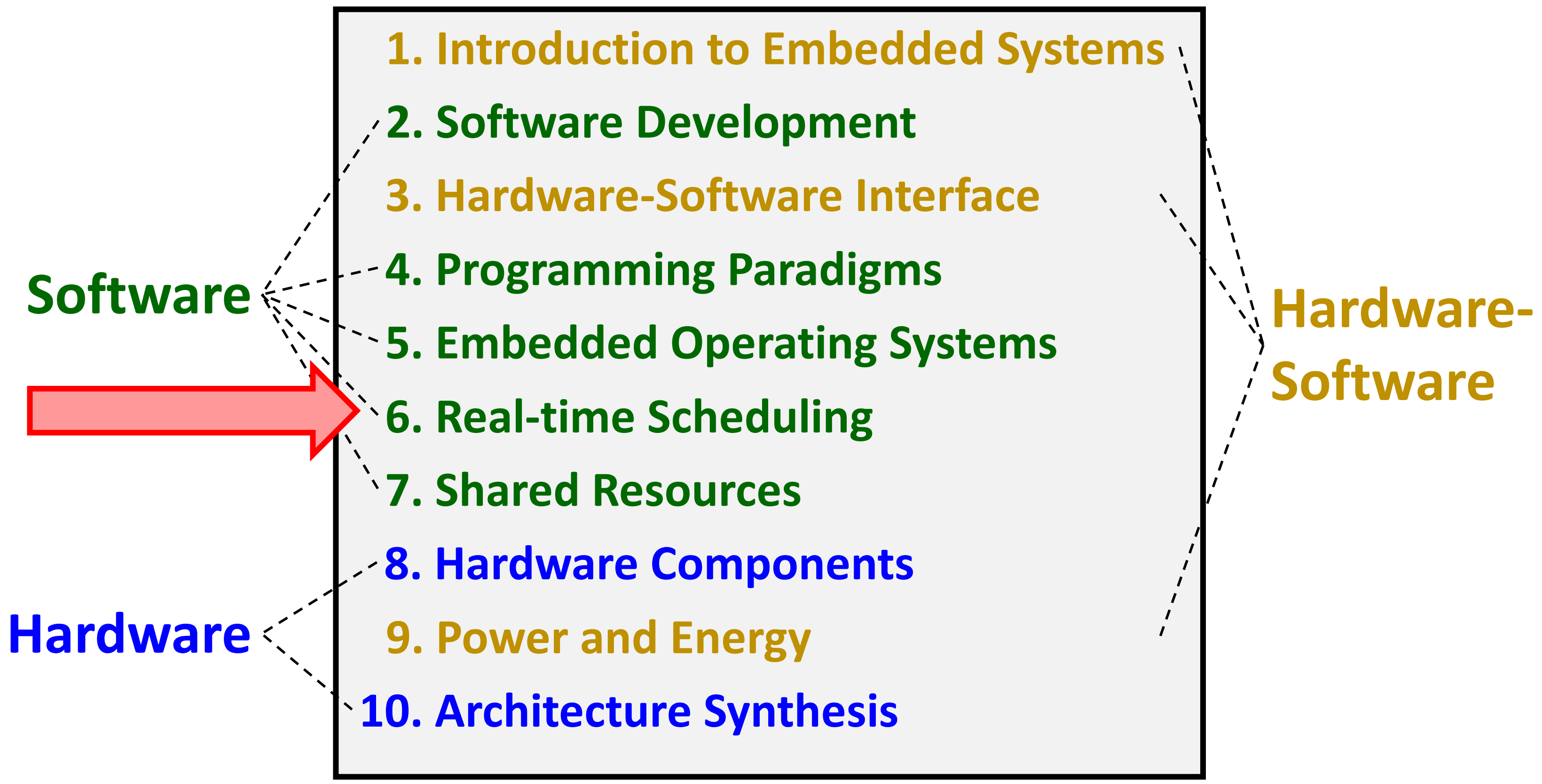
Introduction to Embedded Systems

6. Real-Time Scheduling

Prof. Dr. Marco Zimmerling

Where we are …

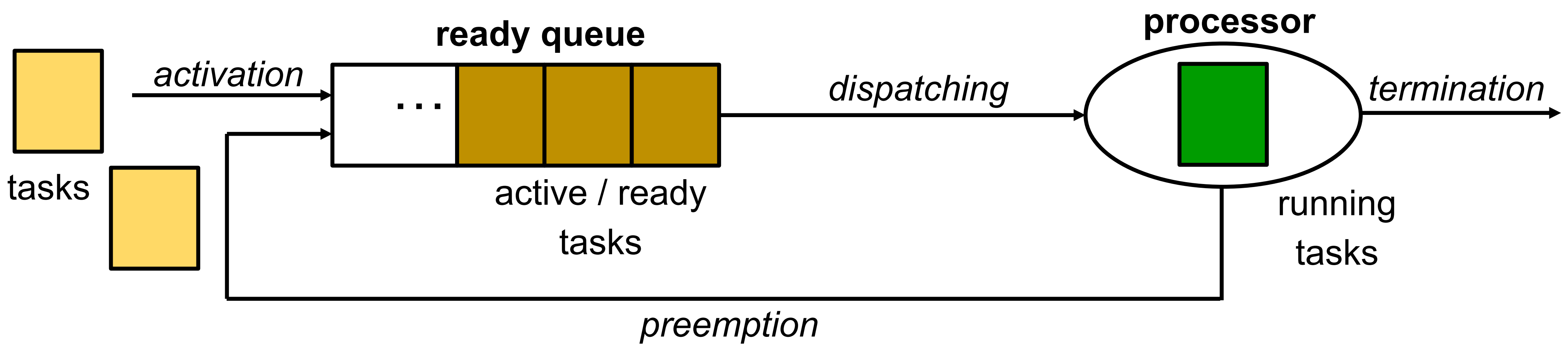


1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis

Terminology and Models

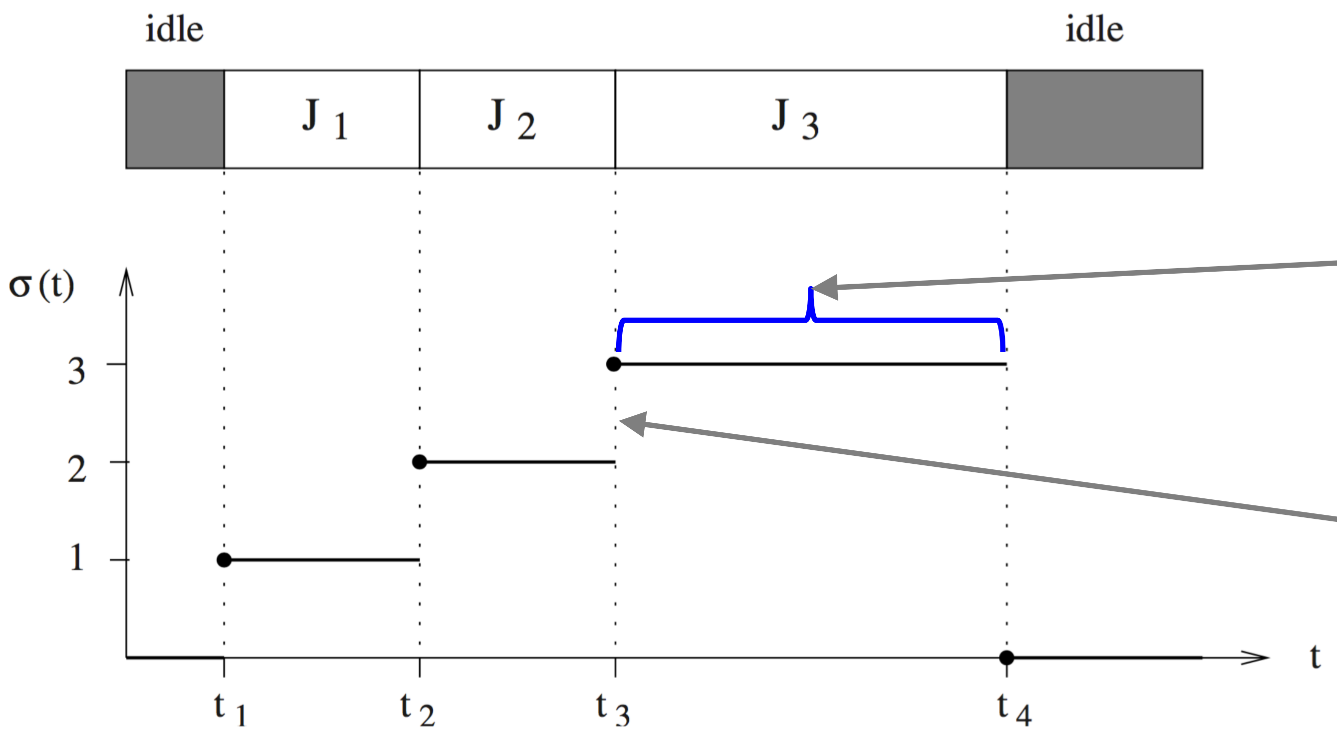
Basic Scheduling Concepts

- A *task* is a computation that is executed by the processor in a sequential fashion.
- A *scheduling algorithm* determines the order in which tasks that can overlap in time are executed on the processor.
- The operation of suspending the running task and inserting it into the ready queue is called *preemption*.



Definition of Schedule

- Given a set of tasks, $J = \{J_1, \ldots, J_n\}$, a *schedule* is an assignment of tasks to the processor, so that each task is executed until completion.
- Formally, a schedule is an integer step function $\sigma : \mathbb{R}^+ \rightarrow \mathbb{N}$, where
  - $\sigma(t) = k$ means that the processor *executes* task J_k at time t, and
  - $\sigma(t) = 0$ means that the processor is *idle* at time t.



time slice: interval $[t_i, t_{i+1})$ in which $\sigma(t)$ is constant

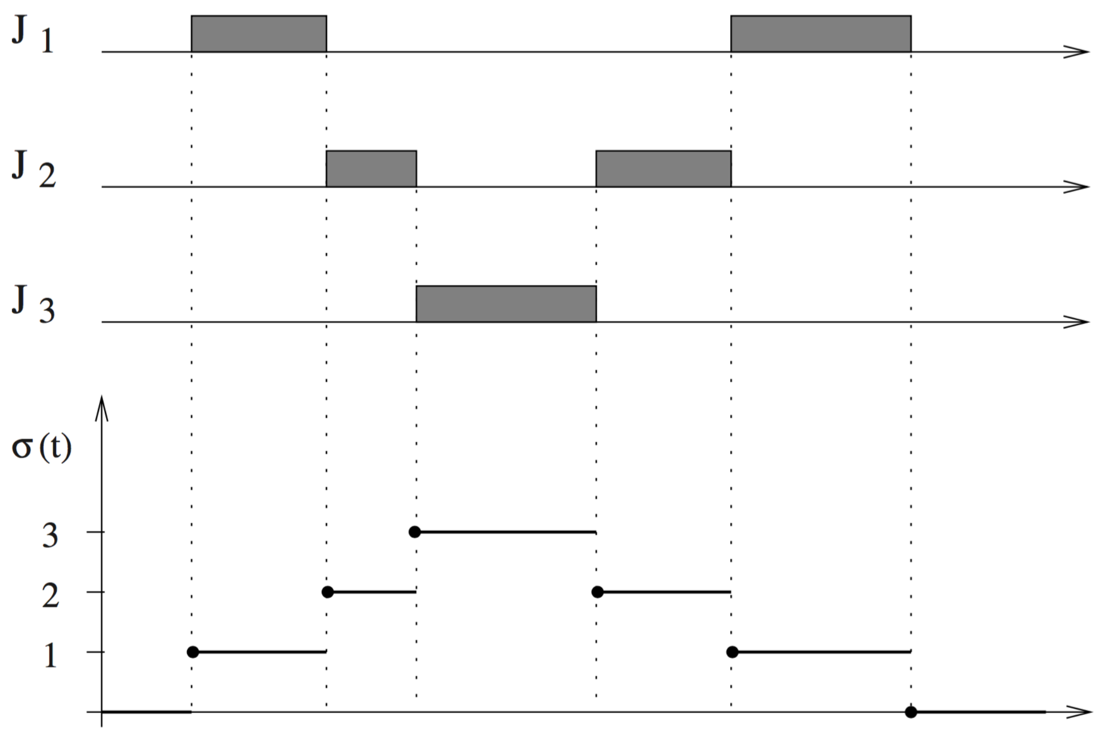
context switch: time t at which $\sigma(t)$ changes

Important Attributes

- A *preemptive* schedule is a schedule in which the running task can be arbitrarily suspended at any time to assign the processor to another task.

- A schedule is said to be *feasible* if all tasks can be completed according to a set of specified constraints.

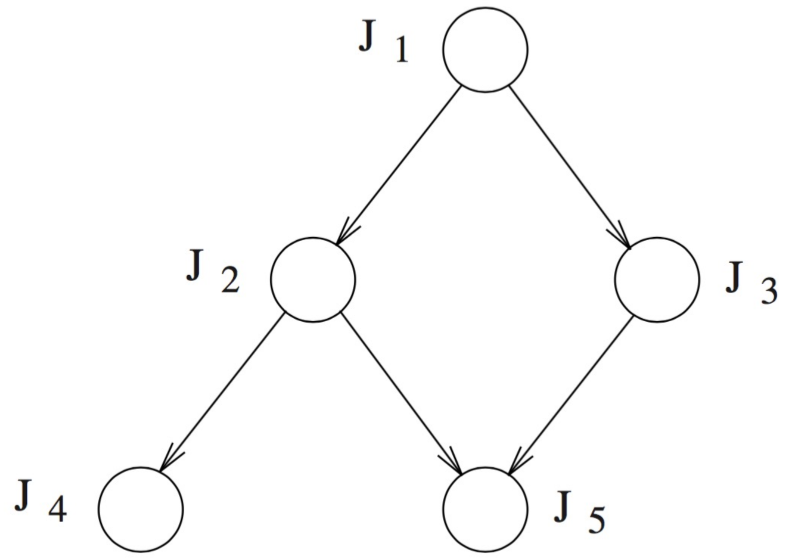
- A set of tasks is said to be *schedulable* if there exists at least one scheduling algorithm that can produce a feasible schedule.



Example of a *preemptive* schedule

Types of Task Constraints

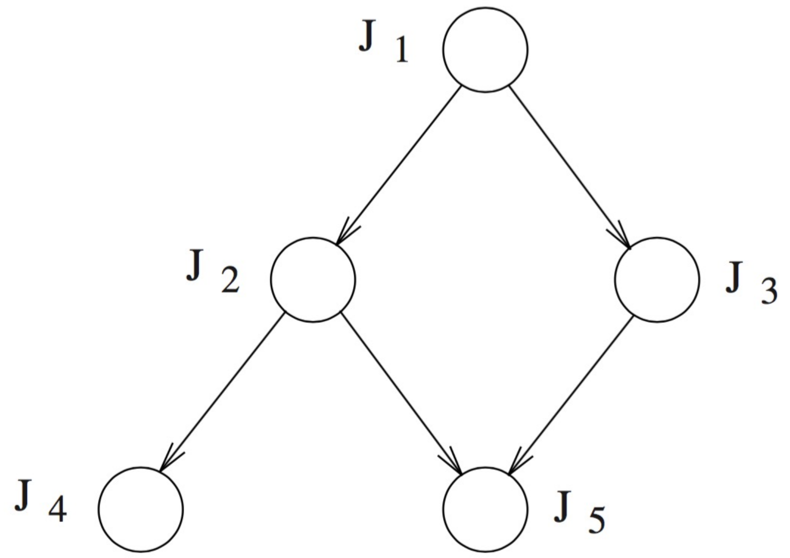
- *Timing constraints*: tasks need to complete before given deadlines
- *Precedence constraints*: tasks need to execute in a given order



- *Resource constraints*: tasks need to execute on given resources (*e.g.*, data structure, piece of a program, memory area, peripheral device)

Types of Task Constraints

- *Timing constraints*: tasks need to complete before given deadlines
- *Precedence constraints*: tasks need to execute in a given order



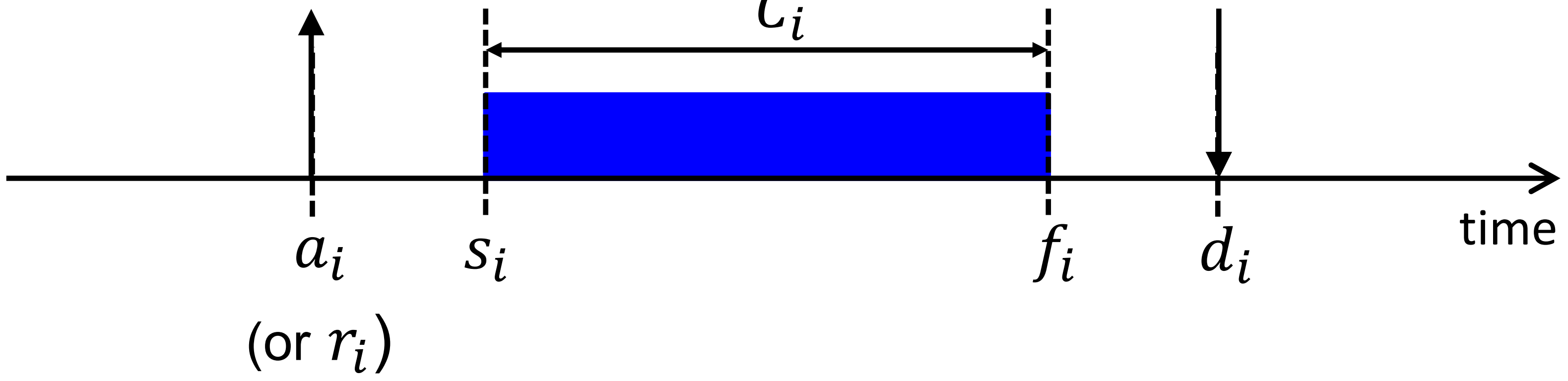
- *Resource constraints*: tasks need to execute on given resources (*e.g.*, data structure, piece of a program, memory area, peripheral device)

Tasks with Timing Constraints

- A typical timing constraint on a task is a *deadline*, representing the time before which the task should complete its execution.
- A task is called a *real-time task* if it is subject to a specified deadline.
- Depending on the consequences of a missed deadline, real-time tasks can be classified into two main categories:
  - A real-time task is said to be *hard* if missing its deadline may cause catastrophic consequences on the system under control. Examples include sensing, actuation, and control tasks in a safety-critical system (e.g., airplane, car, power plant, pace makers).
  - A real-time task is said to be *soft* if missing its deadline does not cause any serious damage and has still some utility for the system; that is, a deadline miss is considered a performance issue, not an issue of correct behavior. Examples are tasks related to user interactions on a smartphone or to convenience functions in a car (e.g., seat warmer).

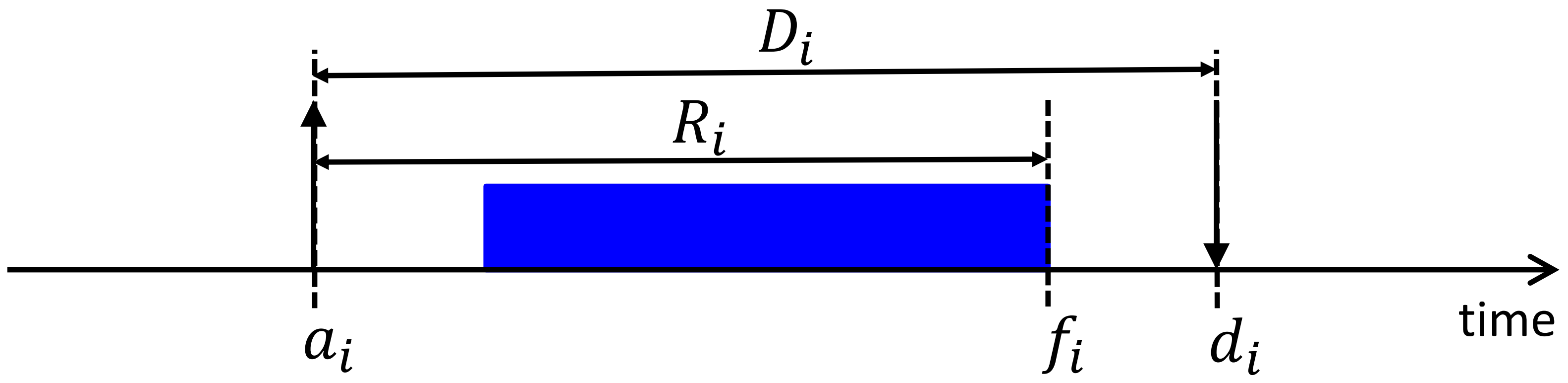
Parameters of Real-Time Tasks (1)

- *Arrival time* a_i or *release time* r_i is the time at which a task becomes ready for execution.
- *Start time* s_i is the time at which a task starts its execution.
- *Execution time* C_i is the time needed by the processor to execute a task without interruption.
- *Finishing time* f_i is the time at which a task finishes its execution.
- *Absolute deadline* d_i is the time by which a task should be completed.

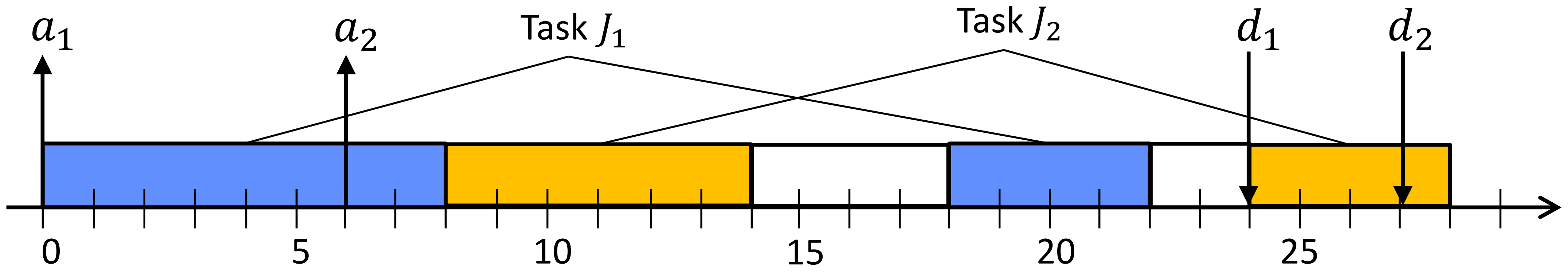


Parameters of Real-Time Tasks (2)

- *Relative deadline* D_i is the difference between the absolute deadline and the arrival time of a task, that is, $D_i = d_i - a_i$.
- *Response time* R_i is the difference between the finishing time and the arrival time of a task, that is, $R_i = f_i - a_i$.
- *Lateness* $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline, that is, $L_i \leq 0$ if a task completes within its deadline.
- *Tardiness* $E_i = \max(0, L_i)$ is the time a task stays active after its deadline.
- *Laxity* $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its execution in order to complete within its deadline.



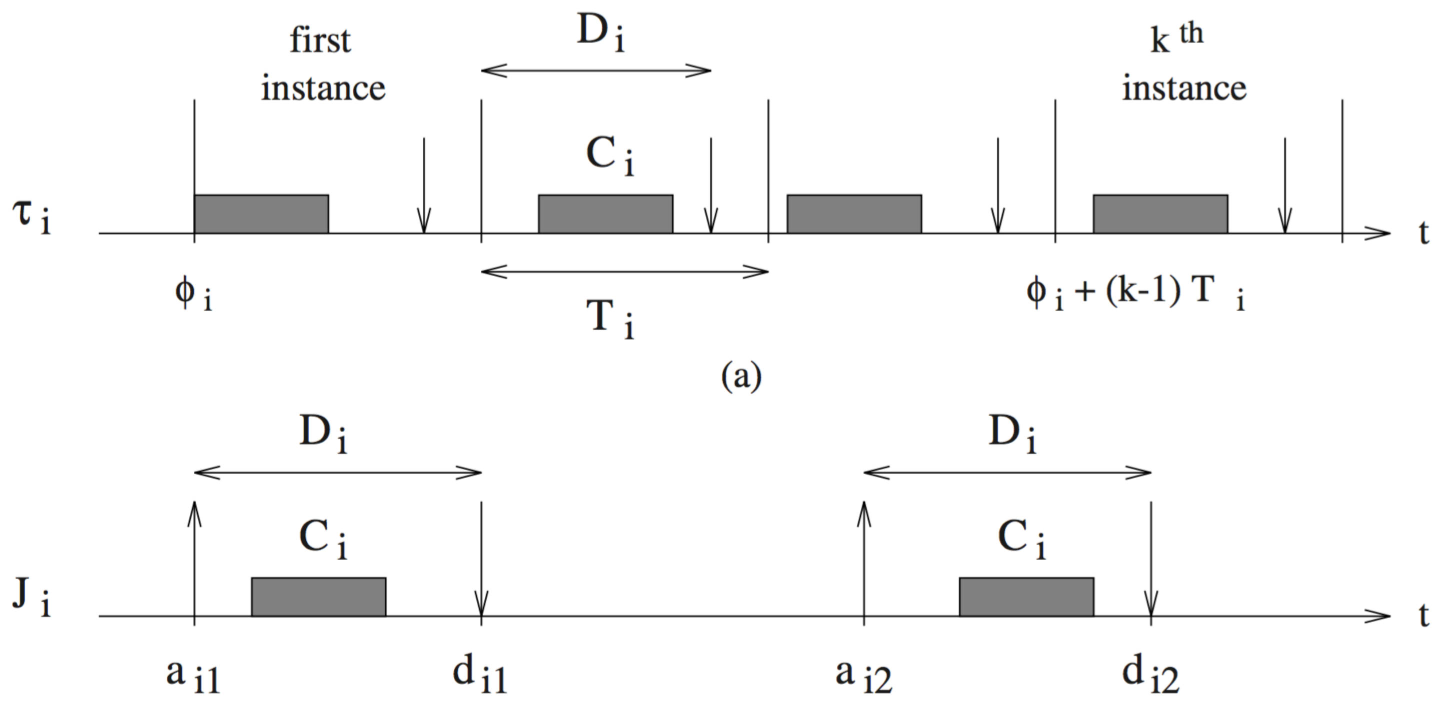
Example: Parameters of Real-time Tasks



- Execution times: $C_1 = 12, C_2 = 10$
- Start times: $s_1 = 0, s_2 = 8$
- Finishing times: $f_1 = 22, f_2 = 28$
- Lateness: $L_1 = -2, L_2 = 1$
- Tardiness: $E_1 = 0, E_2 = 1$
- Laxity: $X_1 = 12, X_2 = 11$

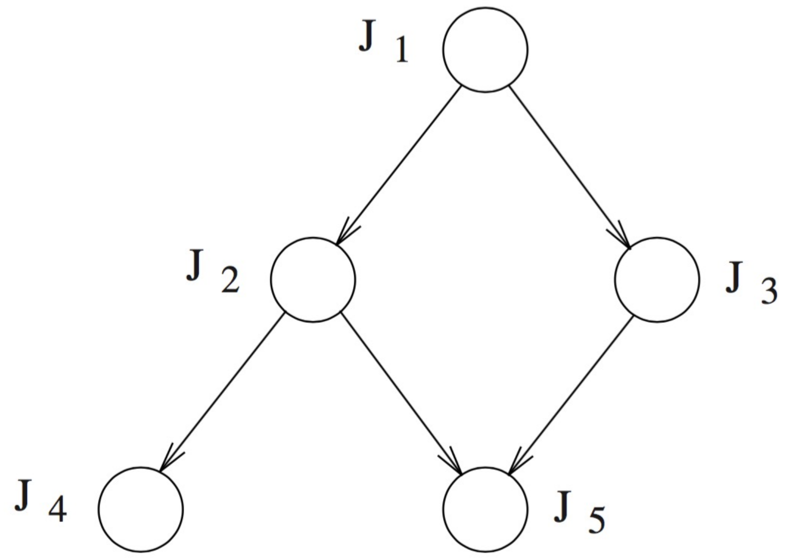
Periodic and Aperiodic Tasks

- A *periodic task* τ_i consists of an infinite sequence of identical activities, called instances or jobs, that are regularly activated with a constant *period* T_i. The arrival time of the first instance is called *phase* Φ_i.
- We use τ_i to denote a periodic task and J_i to denote an aperiodic task. An aperiodic task J_i can arrive (or can be released) at *any point in time*.



Types of Task Constraints

- *Timing constraints*: tasks need to complete before given deadlines
- *Precedence constraints*: tasks need to execute in a given order



- *Resource constraints*: tasks need to execute on given resources (*e.g.*, data structure, piece of a program, memory area, peripheral device)

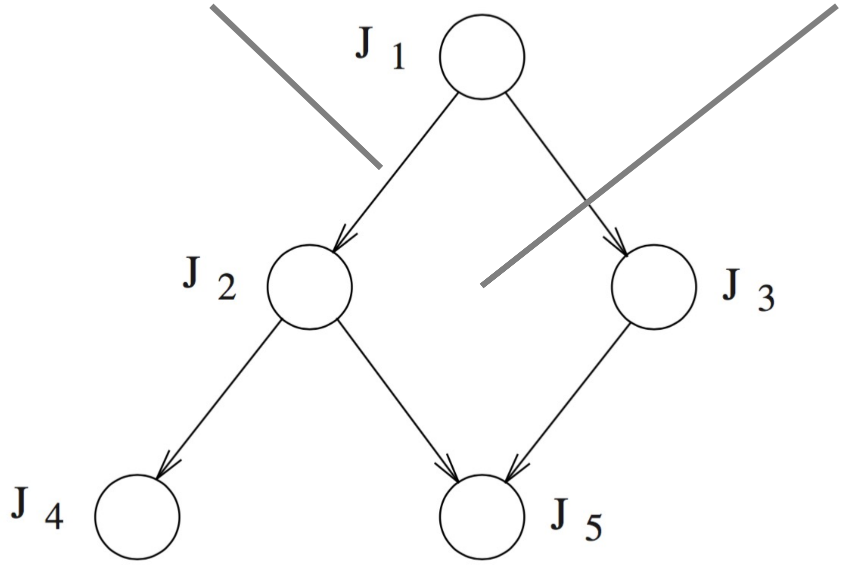
Tasks with Precedence Constraints

- Precedence constraints between tasks can be described through a *directed acyclic graph (or DAG)* G, where tasks are represented by nodes and precedence constraints by arrows. G induces a partial order on the task set.

- Different interpretations are possible:
  - *Concurrent task execution*: all successors of a task are activated. We will use this interpretation in the lecture.
  - *Non-deterministic choice*: one successor of a task is activated.

J_1 must finish before J_2 can start to execute

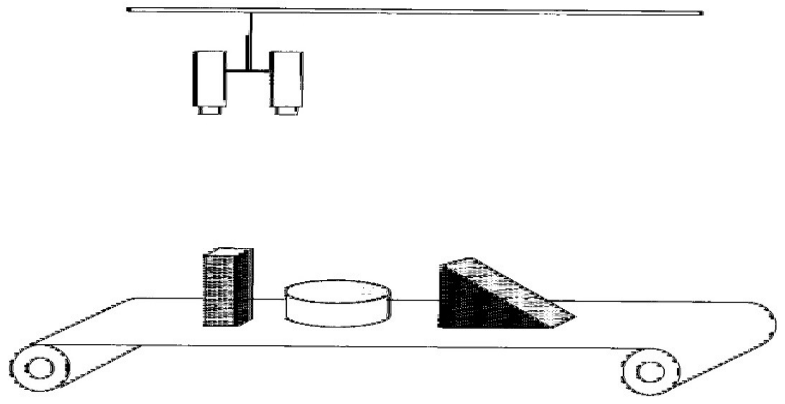
J_2 and J_3 can execute in any order

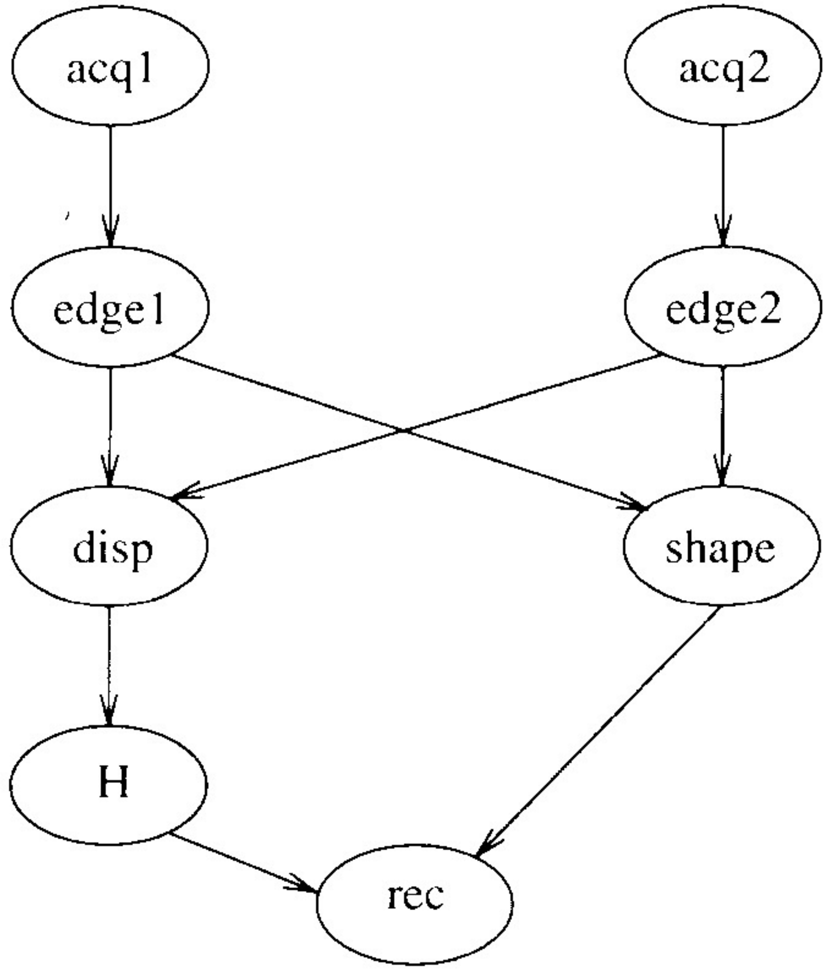


Example: Concurrent Activation

Object recognition with two cameras:

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





Classification of Scheduling Algorithms (1)

- Using a *preemptive* algorithm, the running task can be interrupted at any time to assign the processor to another active task.
- Using a *non-preemptive* algorithm, a task, once started, is executed by the processor until completion. Interrupts, which are typically very short, are still allowed and can preempt tasks. A task, however, cannot preempt another task.

- *Static* algorithms are those in which scheduling decisions are based on fixed parameters that are assigned to tasks before their activation. For example, this includes all time-triggered algorithms from Chapter 4 on programming paradigms.
- *Dynamic* algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system operation. For example, this includes all event-triggered algorithms from Chapter 4 on programming paradigms.

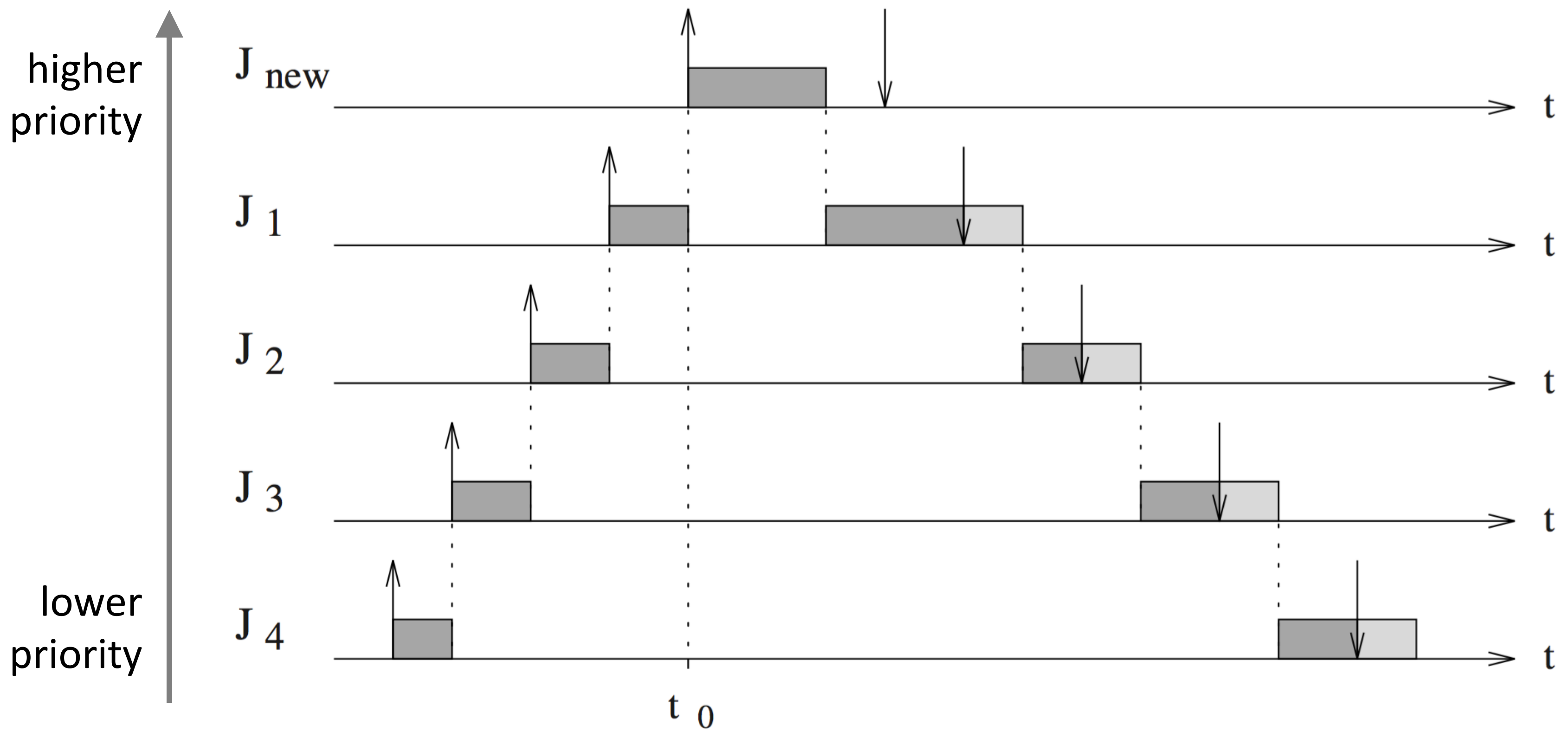
Classification of Scheduling Algorithms (2)

- An algorithm is used *offline* if it is executed on the entire task set before any task activation. The generated schedule can be stored in a table and then executed at runtime by a dispatcher.
- An algorithm is used *online* if scheduling decisions are taken at runtime every time a new task enters the system or when a running task terminates.

- An algorithm is said to be *optimal* it it minimizes a given cost function defined over the task set.
- An algorithm is said to be *heuristic* if it tends toward the optimal schedule, but does not guarantee finding it.

Schedulability Analysis

- In hard real-time applications, feasibility of the schedule should be guaranteed in advance (*i.e.*, before task execution).
  - Can be checked offline if task set is fixed and known a priori.
  - Must be checked online if tasks can be created at runtime (*acceptance test*).



Domino effect: if task J_{new} were accepted at time t_0, all other (previously schedulable) tasks would miss their deadline.

Metrics to Evaluate Schedules

- Average response time: $\overline{t_r} = \frac{1}{n}\sum_{i=1}^{n}(f_i - a_i)$

- Total completion time: $t_c = \max_i(f_i) - \min_i(a_i)$

- Weighted sum of response times: $t_w = \frac{\sum_{i=1}^{n} w_i(f_i - a_i)}{\sum_{i=1}^{n} w_i}$

- Maximum lateness: $L_{max} = \max_i(f_i - d_i)$

- Number of late tasks: $N_{late} = \sum_{i=1}^{n} miss(f_i)$

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

Metrics to Evaluate Schedules

- Average response time: $\overline{t_r} = \frac{1}{n}\sum_{i=1}^{n}(f_i - a_i)$

- Total completion time: $t_c = \max_i(f_i) - \min_i(a_i)$

- Weighted sum of response times: $t_w = \frac{\sum_{i=1}^{n} w_i(f_i - a_i)}{\sum_{i=1}^{n} w_i}$

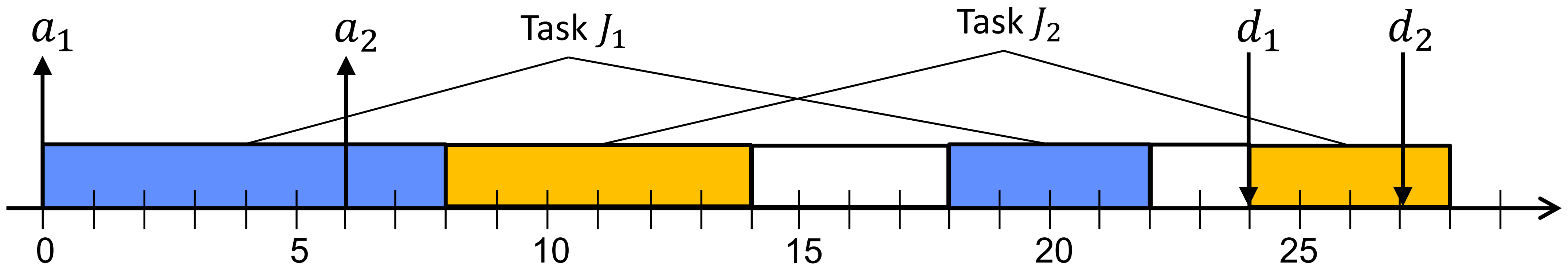
- Maximum lateness: $L_{max} = \max_i(f_i - d_i)$

- Number of late tasks: $N_{late} = \sum_{i=1}^{n} miss(f_i)$

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

Only these metrics are useful to evaluate real-time schedules, as they involve task deadlines.

Example: Metrics

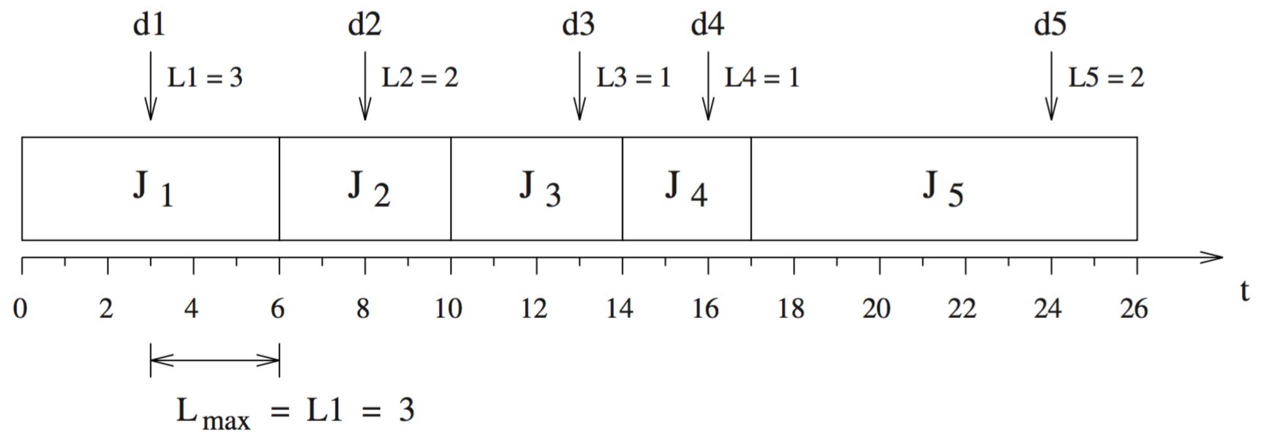


- Average response time: $\overline{t_r} = \frac{1}{n}\sum_{i=1}^{n}(f_i - a_i) = \frac{1}{2}(22 + 22) = 22$
- Total completion time: $t_c = \max_i(f_i) - \min_i(a_i) = 28 - 0 = 28$
- Weighted sum of response times: $t_w = \frac{2\times 22 + 1\times 22}{3} = 22$ for $w_1 = 2, w_2 = 1$
- Maximum lateness: $L_{max} = \max_i(f_i - d_i) = \max_2(-2,1) = 1$
- Number of late tasks: $N_{late} = \sum_{i=1}^{n} miss(f_i) = 0 + 1 = 1$

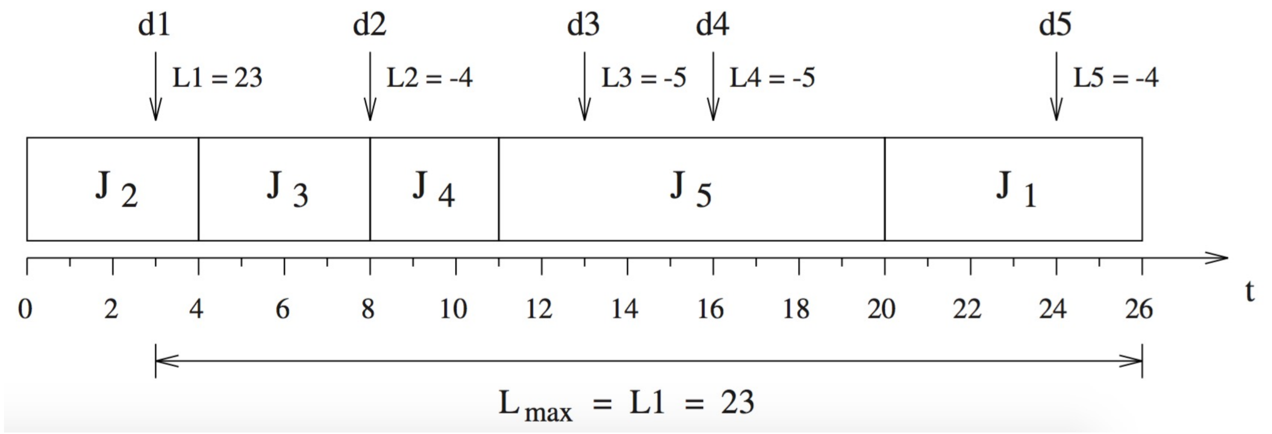
where $miss(f_i) = \begin{cases} 0 \text{ if } f_i \le d_i \\ 1 \text{ otherwise} \end{cases}$

Example: Maximum Lateness vs. Deadline Misses

(a)



(b)



- Schedule in (a) *minimizes maximum lateness*, but all tasks miss deadline.
- Schedule in (b) has higher maximum lateness, but *only one deadline miss*.

Real-Time Scheduling of Aperiodic Tasks

Overview Aperiodic Task Scheduling

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

- Table with some known algorithms:

		Equal arrival times non preemptive	Arbitrary arrival times preemptive
(*independent* = without precedence constraints)	**Independent tasks**	EDD (Jackson)	EDF (Horn)
(*dependent* = with precedence constraints)	**Dependent tasks**	LDF (Lawler)	EDF* (Chetto)

Earliest Deadline Due (EDD)

- Scheduling of aperiodic tasks J_i with *equal arrival times*
  - We assume that all tasks arrive at time $t = 0$ (*i.e.*, $a_i = r_i = 0$ for all tasks J_i).
  - There are no further constraints on the tasks (e.g., no precedence constraints).
  - Thus, each task J_i is fully characterized by execution time C_i and relative deadline D_i.
  - Note: Preemption is not an issue if all tasks arrive at the same time! Hence, EDD is effectively a *non-preemptive* real-time scheduling method.

- *Jackson's rule*: Given a set of n independent tasks, any algorithm that executes the tasks in order of non-decreasing deadline is optimal with respect to minimizing the maximum lateness of the task set.

Example: Feasible Schedule Produced by EDD

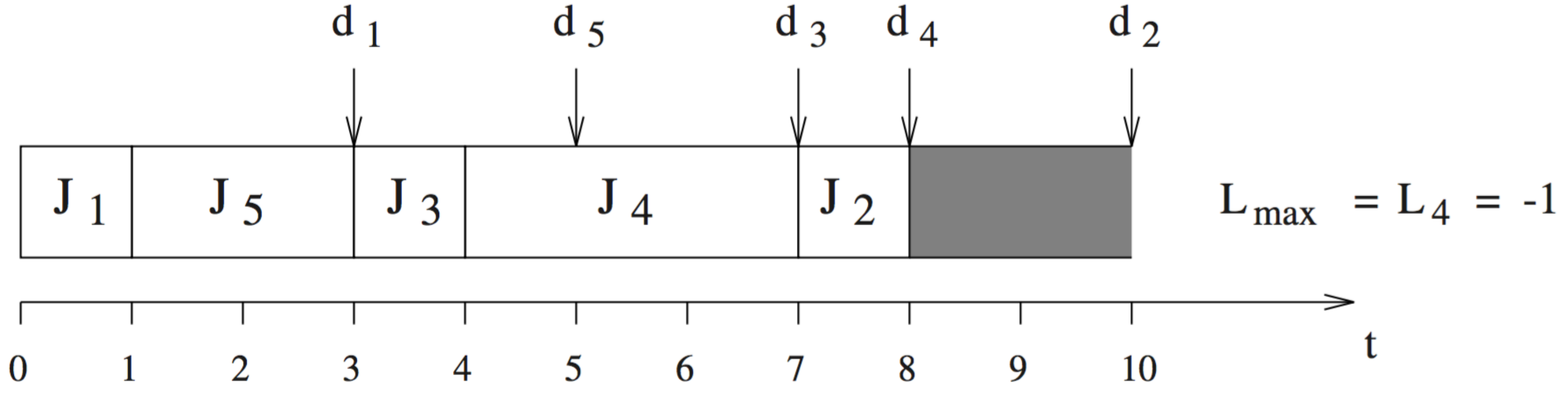
- *Jackson's rule*: Given a set of n independent tasks, any algorithm that executes the tasks in order of non-decreasing deadline is optimal with respect to minimizing the maximum lateness of the task set.

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

Example: Feasible Schedule Produced by EDD

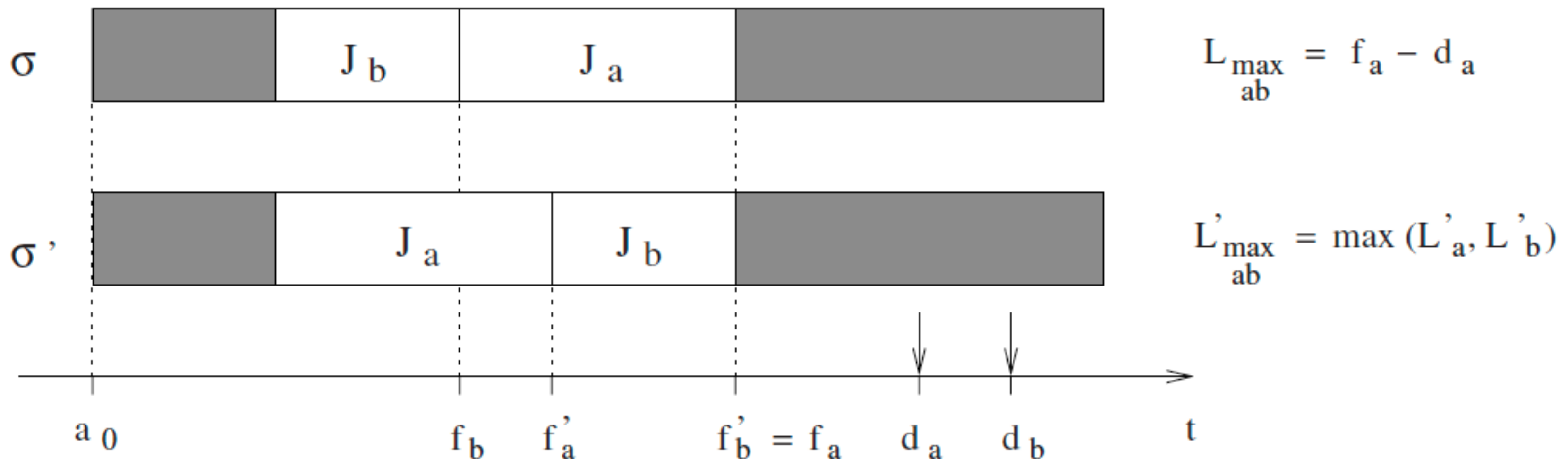
- *Jackson's rule*: Given a set of n independent tasks, any algorithm that executes the tasks in order of non-decreasing deadline is optimal with respect to minimizing the maximum lateness of the task set.

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



Optimality of EDD: Proof Sketch

- Let σ be a schedule produced by an algorithm different from EDD. Then there exist tasks J_a and J_b, with $d_a \le d_b$, such that J_b immediately precedes J_a in σ.
- Let σ' be a schedule derived from σ where J_a and J_b are exchanged. It can be shown that any such exchange cannot increase the maximum lateness of the task set.


σ
J b
J a
σ '
J a
J b
$L_{\max_{ab}} = f_a - d_a$
$L'_{\max_{ab}} = \max(L'_a, L'_b)$
a 0
f b
f' a
f' b = f a
d a
d b
t

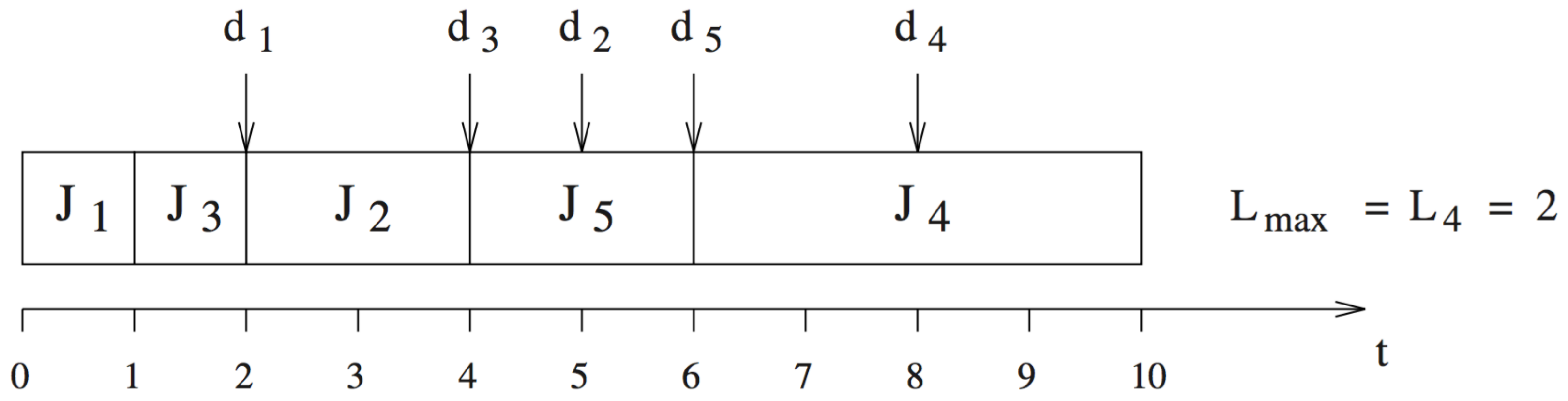
if $(L'_a \ge L'_b)$ then $L'_{\max_{ab}} = f'_a - d_a < f_a - d_a$

if $(L'_a \le L'_b)$ then $L'_{\max_{ab}} = f'_b - d_b < f_a - d_a$

in both cases: $L'_{\max_{ab}} < L_{\max_{ab}}$

Example: Infeasible Schedule Produced by EDD

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



EDD Schedulability Test

- To guarantee that scheduling a set of tasks using EDD produces a feasible schedule, we need to show that in the worst case all tasks can complete before their deadlines, that is, $f_i \leq d_i$ for all tasks J_i.

- If tasks $J_1, J_2, \dots, J_n$ are ordered by increasing deadline, we have

$$f_i = \sum_{k=1}^{i} C_k$$

- Thus, the EDD schedulability test can be performed (offline) by verifying for each task J_i

$$\sum_{k=1}^{i} C_k \leq d_i$$

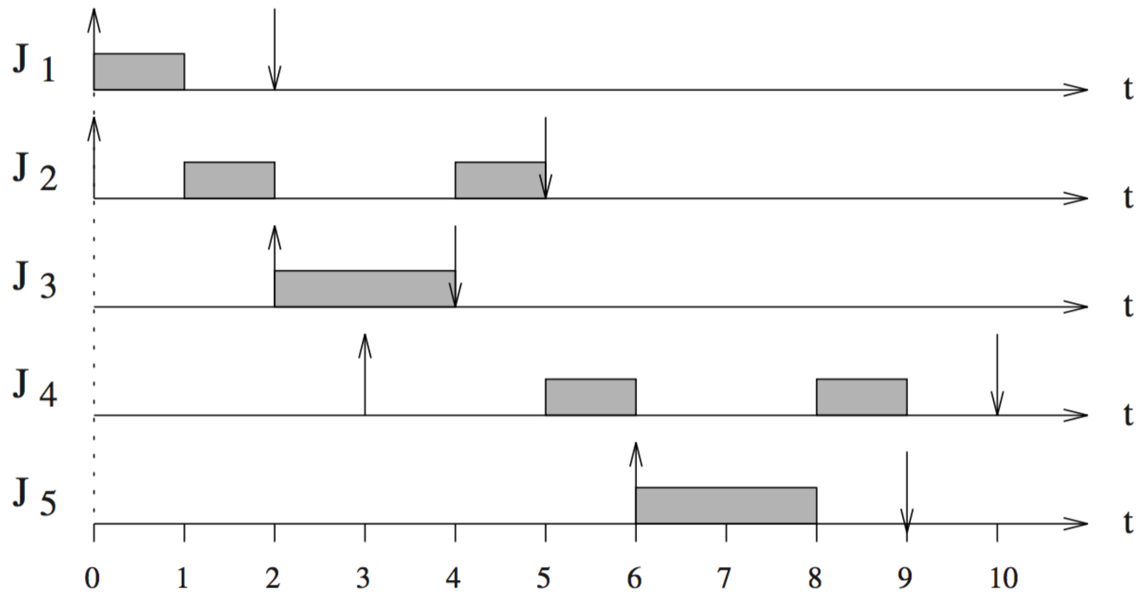
Earliest Deadline First (EDF)

- Scheduling of aperiodic tasks J_i with *arbitrary arrival times*
  - Tasks J_i arrive dynamically, so preemption is an important factor!
  - There are no further constraints on the tasks (e.g., no precedence constraints).
  - Thus, each task J_i is characterized by its execution time C_i, relative deadline D_i, and two parameters only known at runtime: arrival time a_i and absolute deadline d_i.

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

Example: Feasible Schedule Produced by EDF

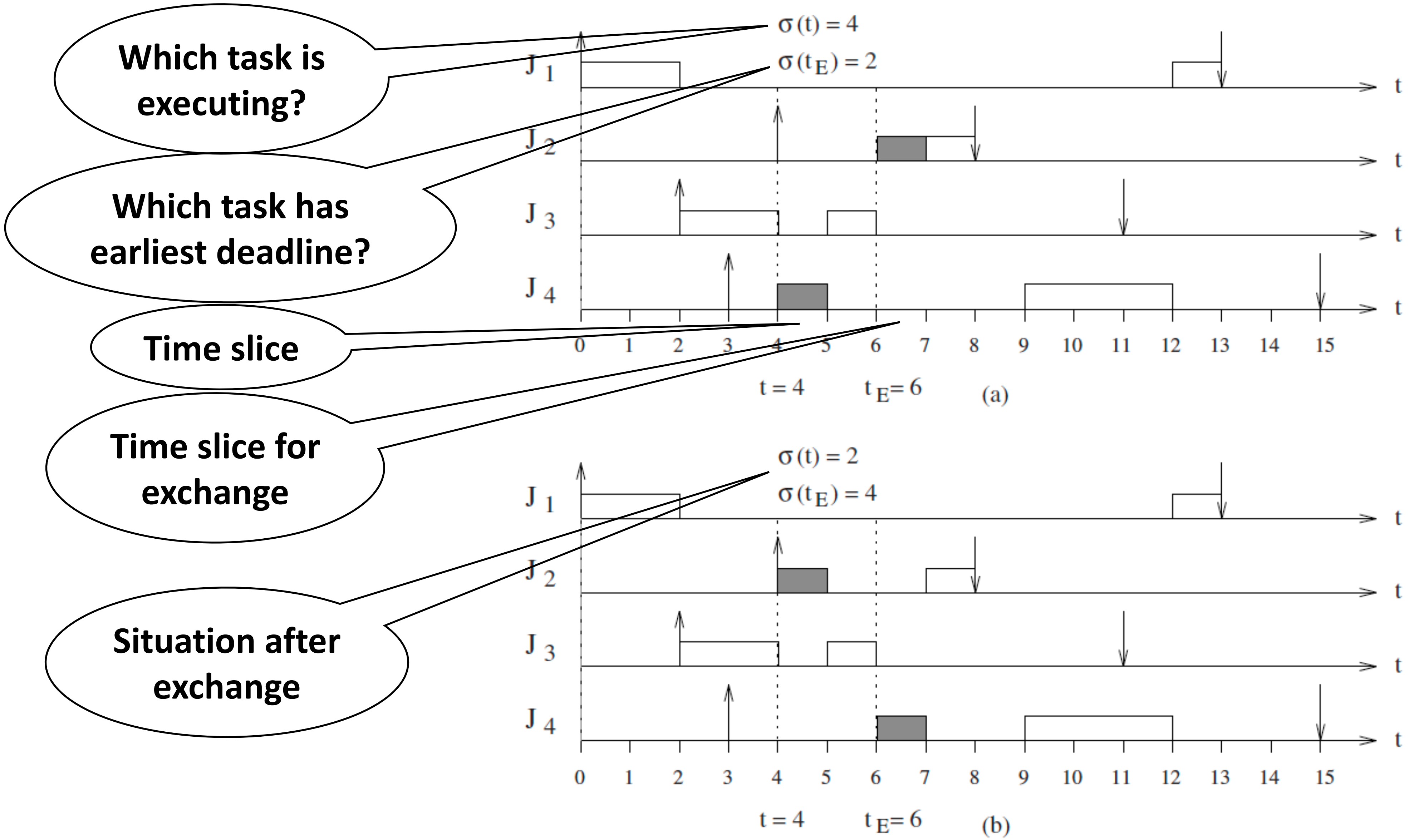
	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
C_i	1	2	2	2	2
d_i	2	5	4	10	9



Optimality of EDF: Proof Sketch (1)

- The proof is similar to the one for EDD. But rather than exchanging complete tasks, one now needs to exchange pieces of tasks using time slices.
- Let σ be a schedule produced by an algorithm different from EDF. Then there exists a time slice $[t, t+1)$ in which the executing task, denoted $\sigma(t)$, is not the task with the earliest absolute deadline among all ready tasks, denoted $E(t)$.
- The basic idea is to exchange this time slice with the next time slice in which $E(t)$ is executed in the current schedule. It can be shown that any such exchange cannot increase the maximum lateness of the task set.
- The next slide shows an example of such an exchange of time slices.

Optimality of EDF: Proof Sketch (2)



EDF Schedulability/Acceptance Test: Approach

- Similar to EDD, but the test must be done *online* whenever a new task J_{new} enters the system. Thus, assuming the current set of tasks J is schedulable, we need to check if $J' = J \cup J_{new}$ is also schedulable.
- If tasks $J_1, J_2, \dots, J_n$ are ordered by increasing deadline, the worst-case finishing time of task J_i at time t is given by

$$f_i = t + \sum_{k=1}^{i} c_k(t)$$

- Here, $c_k(t)$ is the *remaining worst-case execution* time of task J_k. It is initially equal to C_k, but may have a lower value at time t when J_{new} arrives since task J_k (and others) may have been partially executed.
- Thus, the EDF schedulability/acceptance test performed online at time t amounts to verifying for each task $J_i \in J'$

$$t + \sum_{k=1}^{i} c_k(t) \leq d_i$$

EDF Schedulability/Acceptance Test: Algorithm

```
edf_schedulability_test(J, $J_{new}$){
    J' = J ∪ {$J_{new}$}; /* ordered by increasing deadline */
    $f_0$ = get_current_time();
    for each $J_i$ ∈ J'{
        $f_i$ = $f_{i-1}$ + $c_i$(t);
        if ($f_i$ > $d_i$){
            return NOT_SCHEDULABLE;
        }
    }
    return SCHEDULABLE;
}
```

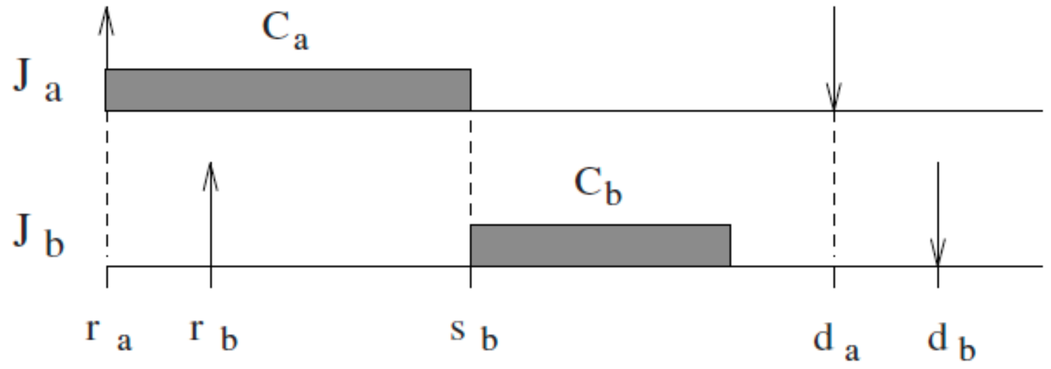
EDF with Precedence Constraints (EDF*)

- Scheduling of aperiodic tasks J_i with *precedence constraints*
  - Tasks J_i arrive dynamically, so they have arbitrary arrival times as for EDF above.
  - Again, each task J_i is characterized by its execution time C_i, relative deadline D_i, and two parameters only known at runtime: arrival time a_i and absolute deadline d_i.
  - In addition, a precedence graph G is given that specifies precedence constraints, where $J_a \rightarrow J_b$ means that task J_a is an immediate predecessor of task J_b.
  - Recall: all successors of a task in the precedence graph are activated (i.e., released)!

- *EDF**: Given a set J of n dependent tasks with arbitrary arrival times, transform this task set into a set J^* of n independent task by adequate modifications of the tasks' release times and deadlines. Then, the task set J^* is scheduled using the standard EDF algorithm. The modifications ensure that J is schedulable and the precedence constraints are satisfied if and only if J^* is schedulable.

Modification of Release Times

- Given tasks J_a and J_b with $J_a \rightarrow J_b$, in any valid schedule that meets precedence constraints the following two conditions must be the satisfied:
  - $s_b \geq r_b$: task J_b must start its execution not earlier than its release time
  - $s_b \geq r_a + C_a$: task J_b must starts its execution not earlier than the minimum finishing time of its immediate predecessor, task J_a

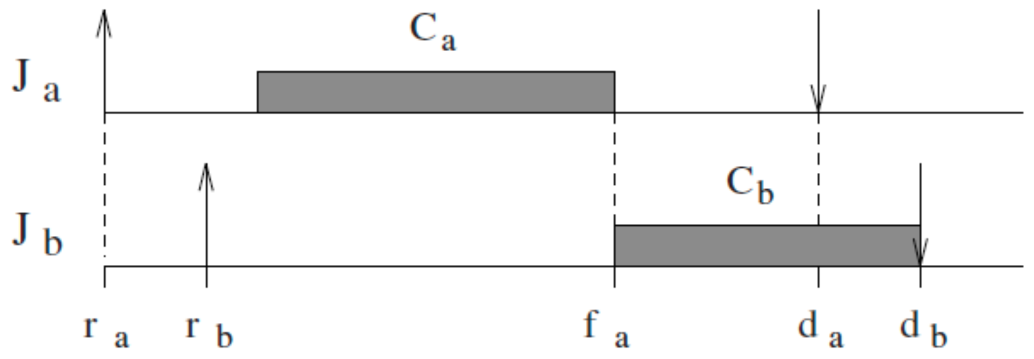


- Thus, the release time r_b of task J_b can be replaced by the new release time

$$r_b^* = \max(r_b, r_a + C_a)$$

Modification of Deadlines

- Given tasks J_a and J_b with $J_a \rightarrow J_b$, in any valid schedule that meets precedence constraints the following two conditions must be the satisfied:
  - $f_a \leq d_a$: task J_a must finish the execution before its deadline
  - $f_a \leq d_b - C_b$: task J_a must finish its execution not later than the maximum start time of its immediate successor, task J_b

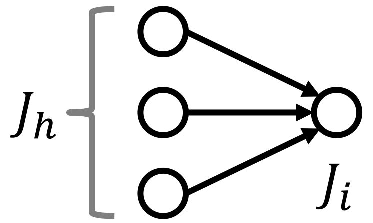


- Thus, the deadline d_a of task J_a can be replaced by the new deadline

$$d_a^* = \min(d_a, d_b - C_b)$$

Algorithm for Modifications

- *Modification of release times*:
  1. For any initial node of the precedence graph, set $r_i^* = r_i$.
  2. Select a task J_i such that its release time has not been modified the release times of all immediate predecessors J_h have been modified. If no such task exists, exit.
  3. Set $r_i^* = \max[r_i, \max(r_h^* + C_h : J_h \rightarrow J_i)]$.
  4. Return to step 2.



- *Modification of deadlines*:
  1. For any terminal node of the precedence graph, set $d_i^* = d_i$.
  2. Select a task J_i such that its deadline has not been modified but the deadlines of all immediate successors J_k have been modified. If no such task exists, exit.
  3. Set $d_i^* = \min[d_i, \min(d_k^* - C_k : J_i \rightarrow J_k)]$.
  4. Return to step 2.