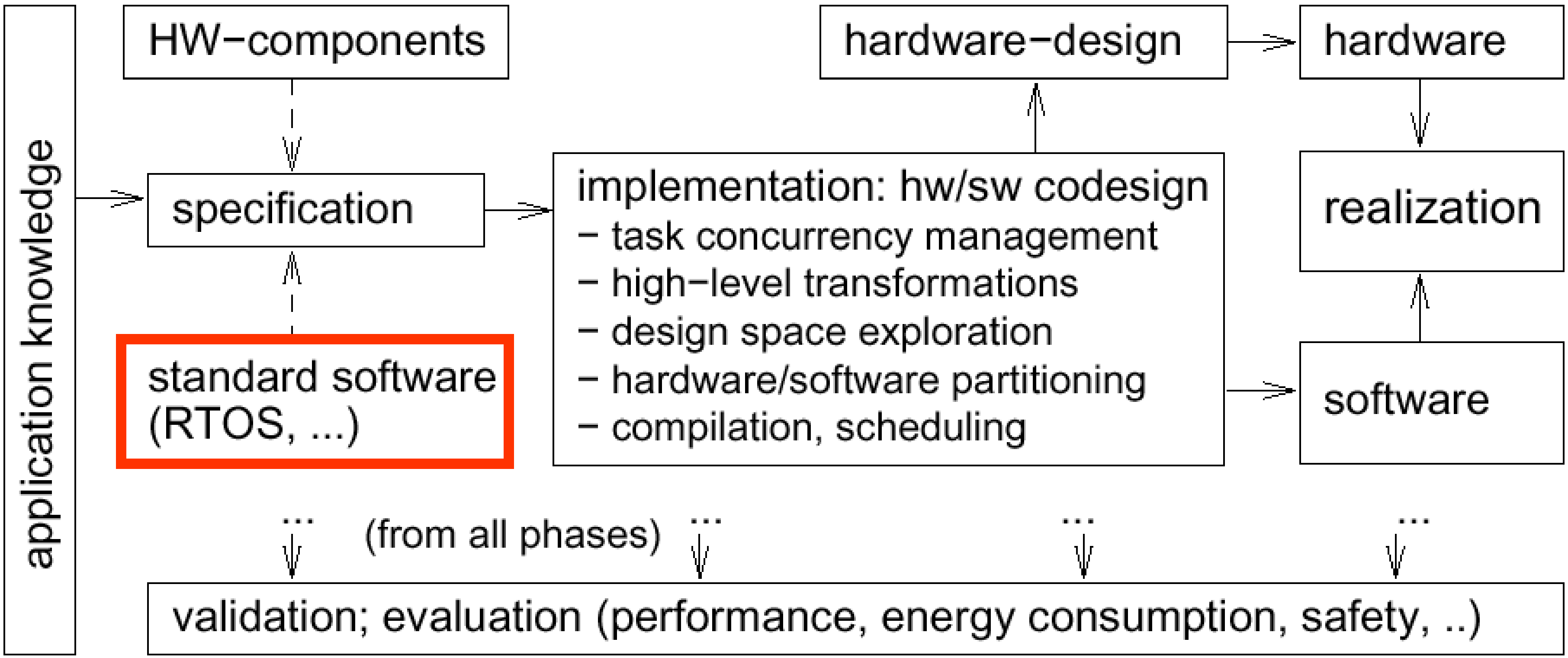
Embedded Systems 14

Overview of embedded systems design



Scheduling

- Support for multi-tasking/multi-threading – several tasks to run on shared resources
- **Task** ~ process – sequential program
- **Resources**: processor(s) + memory, disks, buses, communication channels, etc.
- Scheduler assigns **shared resources** to tasks for **durations of time**
- Most important resource(s) – processor(s)
- **Scheduling** – mostly concerned with processor(s)
  - **Online** – scheduling decisions taken when input becomes available
  - **Offline** – schedule computed with complete input known
- Other shared resources with exclusive access complicate scheduling task

Point of departure: Scheduling general IT systems

- In general IT systems, not much is known about the set of tasks a priori
  - The set of tasks to be scheduled is dynamic:
    - new tasks may be inserted into the running system,
    - executed tasks may disappear.
    - Tasks are activated with unknown activation patterns.
  - The power of schedulers thus is inherently limited by lack of knowledge – only online scheduling is possible

Scheduling processes in ES: The difference in process charaterization

- Most ES are “closed shops”
  - Task set of the system is known
  - at least part of their activation patterns is known
    - periodic activation in, e.g., signal processing
    - maximum activation frequencies of asynchronous events determinable from environment dynamics, minimal inter-arrival times
  - Possible to determine bounds on their execution time (WCET)
    - if they are well-built
    - if we invest enough analysis effort
- Much better prospects for **guaranteeing response times** and for delivering *high-quality* schedules!

Scheduling – Time

- **Time** – aspect of the controlled plant/environment
- Embedded real-time systems have **deadlines** for their reactions, dictated by their environment
  - **hard deadline**: must be met, otherwise system is faulty, examples: airbag, ABS,
  - **soft deadline**: missing it decreases value of the result, examples: video transmission
- Difference between
  - **speed** – being fast on the average - and
  - **punctuality** – being always on time
    Example: DB trains

Scheduling processes in ES: Differences in goals

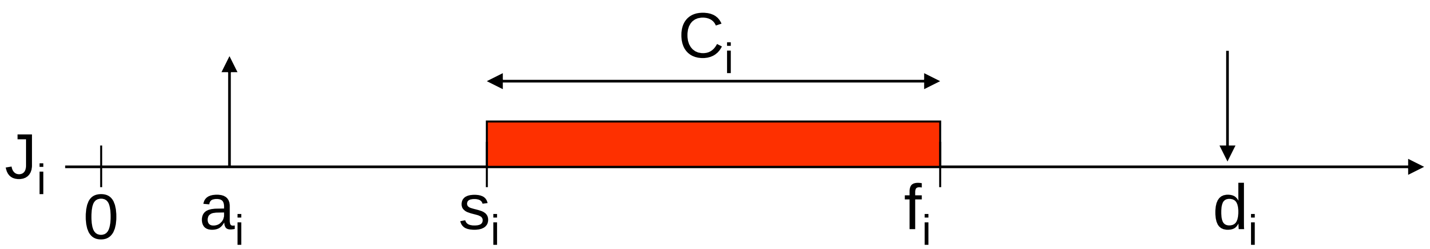
- In classical OS, quality of scheduling is normally measured in terms of performance:
  - Throughput, reaction times, … in the average case
- In ES, the schedules do often have to meet stringent quality criteria under all possible execution scenarios:
  - A task of an RTOS is usually connected with a **deadline**.
    Standard operating systems do not deal with deadlines.
  - Scheduling of an RTOS has to be predictable.
  - Real-time systems have to be designed for peak load.
    Scheduling for meeting deadlines should work for all anticipated situations.

Constraints for real-time tasks

- Three types of constraints for real-time tasks:
  - Timing constraints
  - Precedence constraints
  - Mutual exclusion constraints on shared resources
- Typical timing constraints: Deadlines on tasks
  - Hard: Not meeting the deadline can cause catastrophic consequences on the system
  - Soft: Missing the deadline decreases performance of the system, but does not prevent correct behavior

Timing constraints and schedule properties

- Timing parameters of a real-time task J_i:
  - **Arrival time a_i**: time at which task becomes ready for execution
  - **Computation time C_i**: time necessary to the processor for executing the task without interruption
  - **Deadline d_i**: time before which a task should complete
  - **Start time s_i**: time at which a tasks starts its execution
  - **Finishing time f_i**: time at which task finishes its execution
  - **Lateness L_i**: $L_i = f_i - d_i$, delay of task completion with respect to deadline
  - **Exceeding time E_i**: $E_i = \max(0, L_i)$
  - **Slack time X_i**: $X_i = d_i - a_i - C_i$, maximum time a task can be delayed on its activation to complete within its deadline



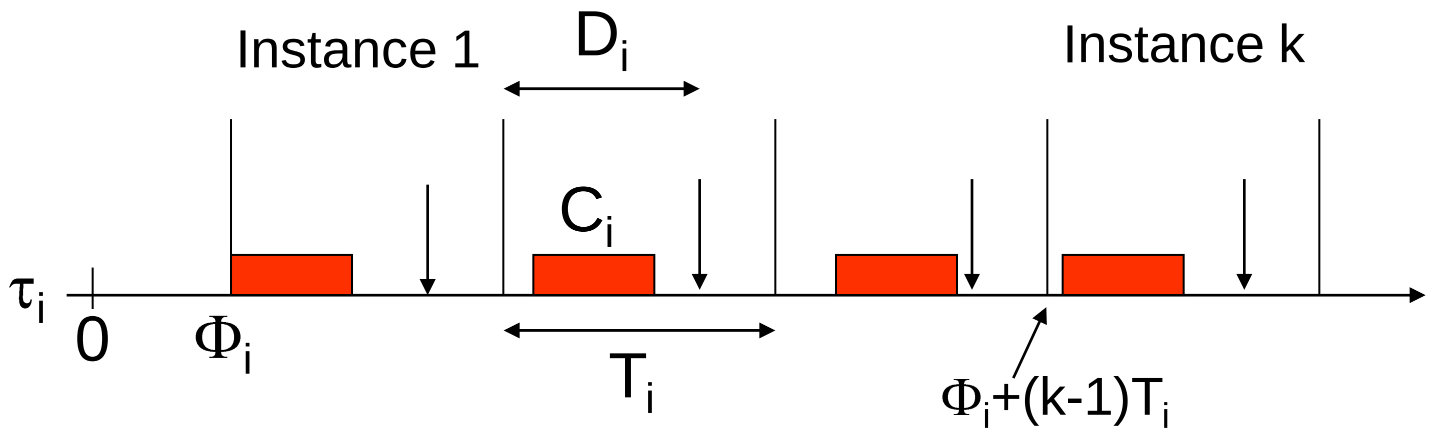
Timing parameters

- Additional timing related parameters of a real-time task J_i:
  - **Criticality:** parameter related to the consequences of missing the deadline
  - **Value v_i:** relative importance of the task with respect to other tasks in the system

  - **Regularity of activation**:
    - Periodic tasks: Infinite sequence of identical activities (instances, jobs) that are regularly activated at a constant rate, here abbreviated by τ_i
    - A-periodic tasks: Tasks which are not recurring or which do not have regular activations, here abbreviated by J_i

Timing constraints of periodic tasks

- **Phase** Φ_i: activation time of first periodic instance
- **Period** T_i: time difference between two consecutive activations
- **Relative deadline** D_i: time after activation time of an instance at which it should be complete



Scheduling - Basic definitions

Given a set of tasks $\{J_1, J_2, \ldots, J_n\}$.

What do we require from a schedule?

- Every processor is assigned to at most one task at any time.
- Every task is assigned to at most one processor at any time.
- All the scheduling constraints are satisfied.

Def.: A (single-processor) schedule is a function $\sigma : \mathbb{R}^+ \rightarrow \mathbb{N}$ such that

$\forall\, t \in \mathbb{R}^+ \ \exists\, t_1 < t_2 \in \mathbb{R}^+ .\ t \in [t_1, t_2)$ and $\forall\, t' \in [t_1, t_2)\ \sigma(t) = \sigma(t')$.

In other words: σ is an integer step function and $\sigma(t) = k$, with $k > 0$, means that task J_k is executed at time t, while $\sigma(t) = 0$ means that the CPU is idle.

- A schedule is feasible, if all tasks can be completed according to a set of specified constraints.
- A set of tasks is schedulable if there exists at least one feasible schedule.
- **Schedulability test**

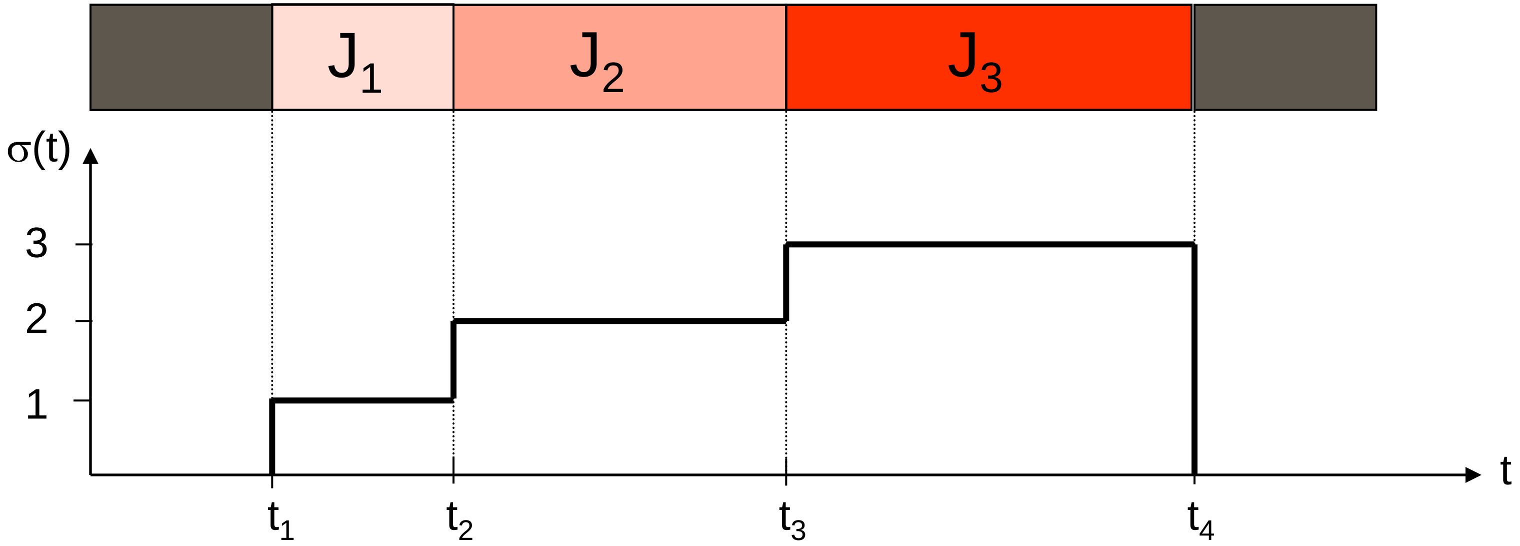
Scheduling Algorithms

- Classes of scheduling algorithms:
  - Preemptive, non-preemptive
    - Task may be interrupted or always runs to completion
  - Off-line / on-line
    - Schedule works on actual and incomplete or on complete information
  - Optimal / heuristic – solutions must be optimal or sub-optimal are determined by using heuristics to reduce effort
  - One processor / multi-processor

- We start with single-processor scheduling.

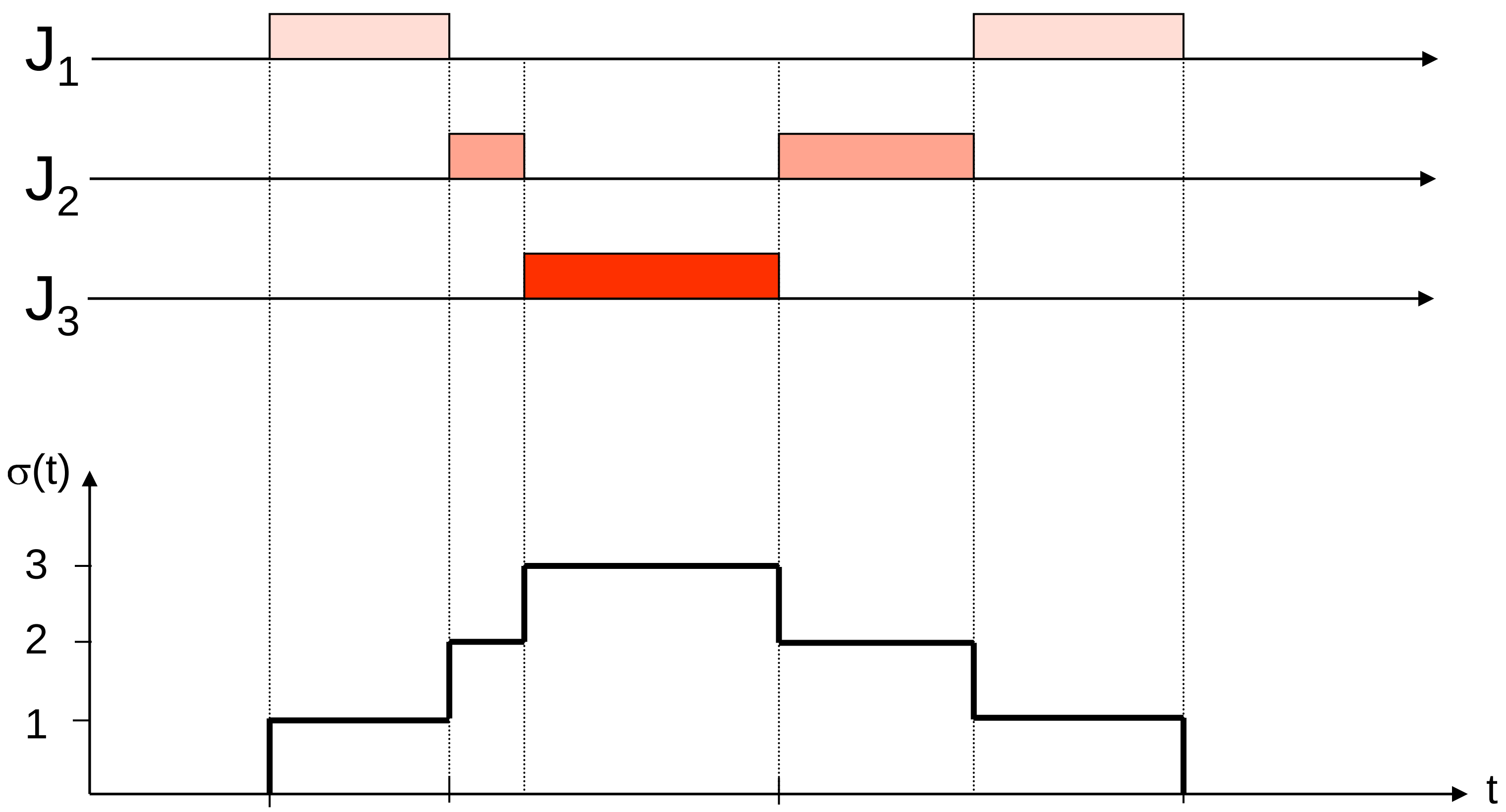
Example

- Non-preemptive schedule of three tasks J_1, J_2, and J_3:



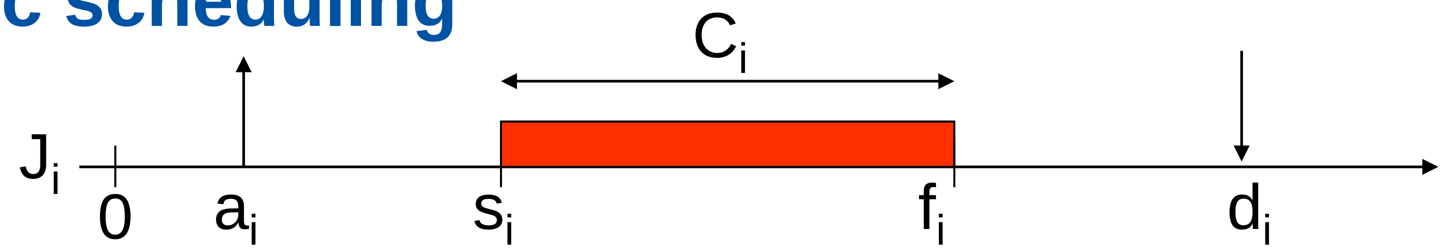
Example

- Preemptive schedule of three tasks J_1, J_2, and J_3:



Scheduling non-periodic tasks

A-periodic scheduling



- Given:
  - A set of a-periodic tasks $\{J_1, \ldots, J_n\}$ with
    - arrival times a_i, deadlines d_i, computation times C_i
    - precedence constraints
    - resource constraints
  - Class of scheduling algorithm:
    - Preemptive, non-preemptive
    - Off-line / on-line
    - Optimal / heuristic
    - One processor / multi-processor
    - …
  - Cost function:
    - Minimize maximum lateness (soft RT)
    - Minimize maximum number of late tasks (feasibility! – hard RT)
- Find:
  Optimal / good schedule according to given cost function

A-periodic scheduling

- Not all combinations of constraints, class of algorithm, cost functions can be solved efficiently.
- If there is some information on restrictions wrt. class of problem instances, then this information should be used!

- Begin with simpler classes of problem instances, then more complex cases.

Case 1: Aperiodic tasks with synchronous release

- A set of (a-periodic) tasks $\{J_1, \ldots, J_n\}$ with
  - arrival times $a_i = 0 \; \forall \; 1 \leq i \leq n$, i.e. "synchronous" arrival times
  - deadlines d_i,
  - computation times C_i
  - no precedence constraints, no resource constraints, i.e. "independent tasks"
- non-preemptive
- single processor
- Optimal
- Find schedule which minimizes maximum lateness (variant: find feasible solution)

Preemption

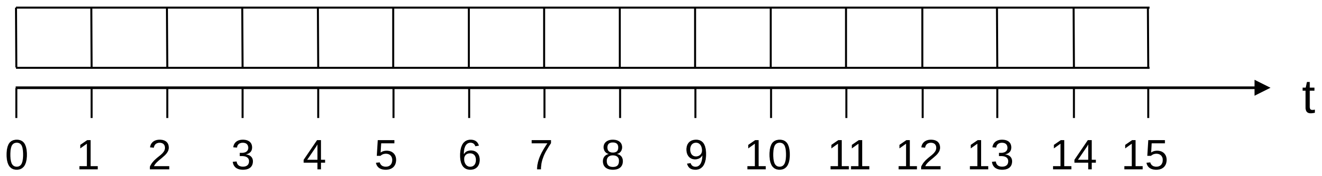
- **Lemma**:
  If arrival times are synchronous, then preemption does not help, i.e. if there is a preemptive schedule with maximum lateness L_{max}, then there is also a non-preemptive schedule with maximum lateness at most L_{max}.

EDD – Earliest Due Date

EDD: execute the tasks in order of non-decreasing deadlines

- Example 1:

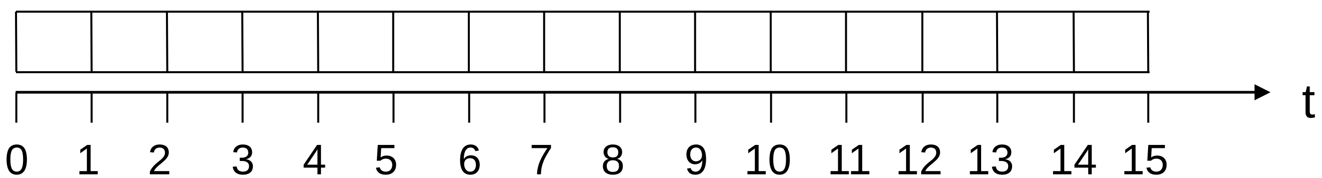
	J_1	J_2	J_3	J_4	J_5
C_i	1	1	1	3	2
d_i	3	10	7	8	5



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



EDD (3)

- Theorem (Jackson '55):
  Given a set of n independent tasks with synchronous arrival times, any algorithm that executes the tasks in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness.

- Remark: Minimizing maximum lateness includes finding a feasible schedule, if it exists. The reverse is not necessarily true.

EDD

- Complexity of EDD scheduling:
  - Sorting n tasks by increasing deadlines

  $\Rightarrow$ O(n log n)

- Test of Schedulability:
  If the conditions of the EDD algorithm are fulfilled, schedulability can be checked in the following way:

  - Sort task wrt. non-decreasing deadline.
    Let w.l.o.g. $J_1, \ldots, J_n$ be the sorted list.
  - Check whether in an EDD schedule $f_i \leq d_i \; \forall \, i = 1, \ldots, n$.
  - Since $f_i = \sum_{k=1}^{i} C_k$, we have to check

    $$\forall \, i = 1, \ldots, n \quad \sum_{k=1}^{i} C_k \leq d_i$$

- Since EDD is optimal, non-schedulability by EDD implies non-schedulability in general.

Optimality Proofs for Scheduling Algorithms

Claim: Scheduling algorithm *A* is optimal

- Feasibilty: If exists a feasible schedule S by some scheduling alg.

  Then: there exists a feasible schedule S_A as obtained by *A*
- Optimality: If exists a schedule *S* optimal w.r.t property *Q* by some scheduling alg.

  Then: there exists a schedule S_A as obtained by *A* with property no worse than *Q*.
- Proof technique:
- Transform schedule *S* into a schedule S_A
  - Preserving feasibility
  - Not impairing property *Q*
- Show this for each transformation step:
  - Select a task/slice of a task in *S* violating the criterion of *A*
  - Move it to a position satisfying this criterion

Case 2: aperiodic tasks with asynchronous release

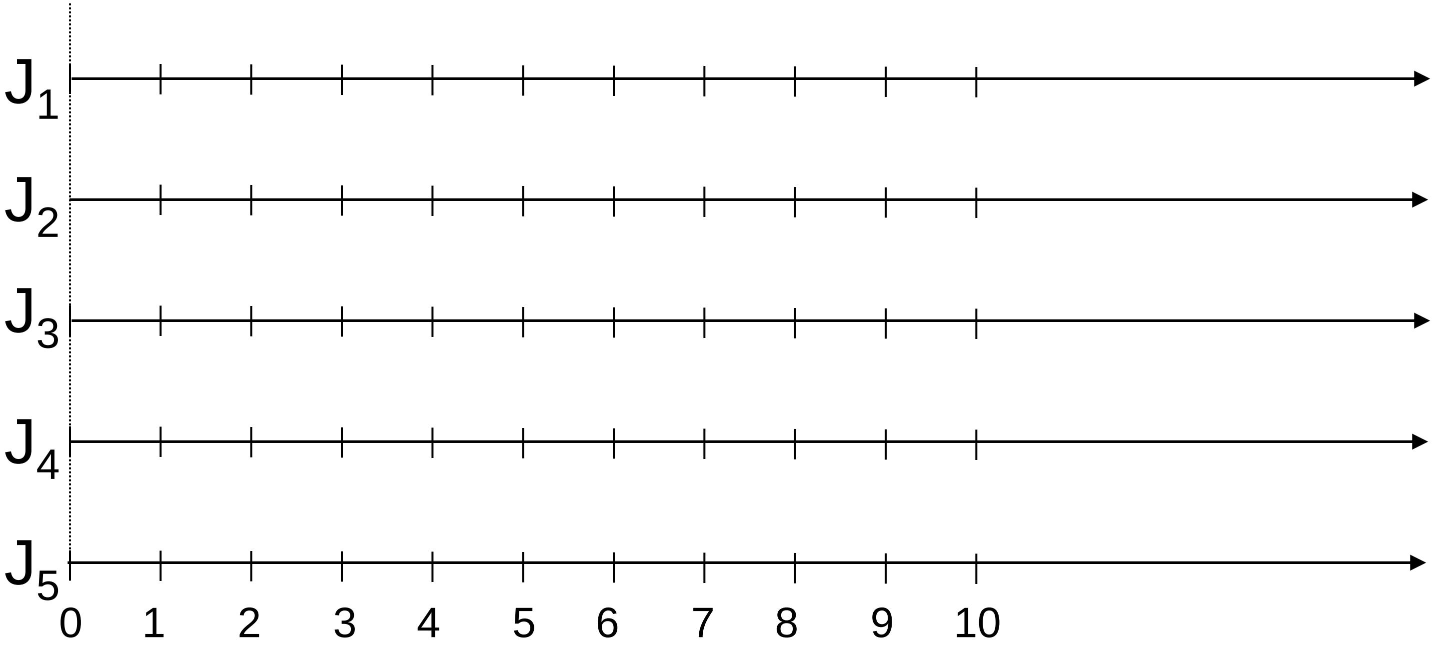
- A set of (a-periodic) tasks $\{J_1, \ldots, J_n\}$ with
  - **arbitrary** arrival times a_i
  - deadlines d_i,
  - computation times C_i
  - no precedence constraints, no resource constraints, i.e. "independent tasks"
- **preemptive**
- Single processor
- Optimal
- Find schedule which minimizes maximum lateness (variant: find feasible solution)

EDF – Earliest Deadline First

- At every instant execute the task with the earliest absolute deadline among all the ready tasks.
- Remark:
  1. If a new task arrives with an earlier deadline than the running task, the running task is immediately preempted.
  2. Here we assume that the time needed for context switches is negligible – we'll later see that this is unrealistic.

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



EDF

- **Theorem (Horn '74):**
  Given a set of n independent tasks with **arbitrary arrival times**, any algorithm that at every instant executes the task with the **earliest absolute deadline** among all the ready tasks is optimal with respect to minimizing the maximum lateness.

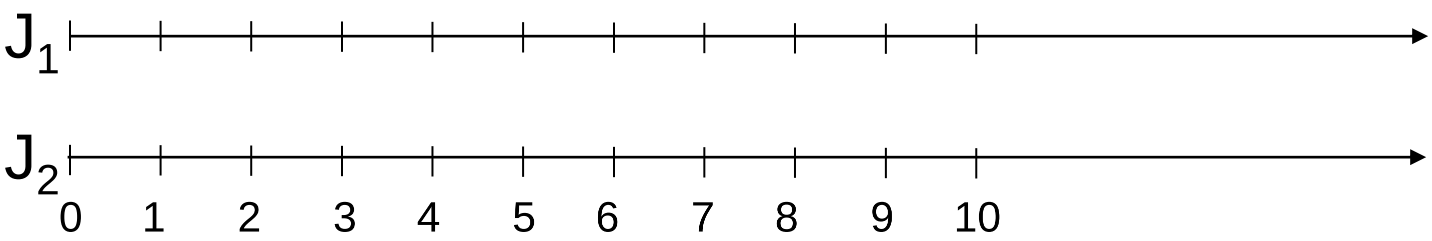
Non-preemptive version

- Changed problem:
  - A set of (a-periodic) tasks $\{J_1, \ldots, J_n\}$ with
    - **arbitrary** arrival times a_i
    - deadlines d_i,
    - computation times C_i
    - no precedence constraints, no resource constraints, i.e. "independent tasks"
  - **Non-preemptive** instead of preemptive scheduling!
  - Single processor
  - Optimal
  - Find schedule which minimizes maximum lateness (variant: find feasible solution)

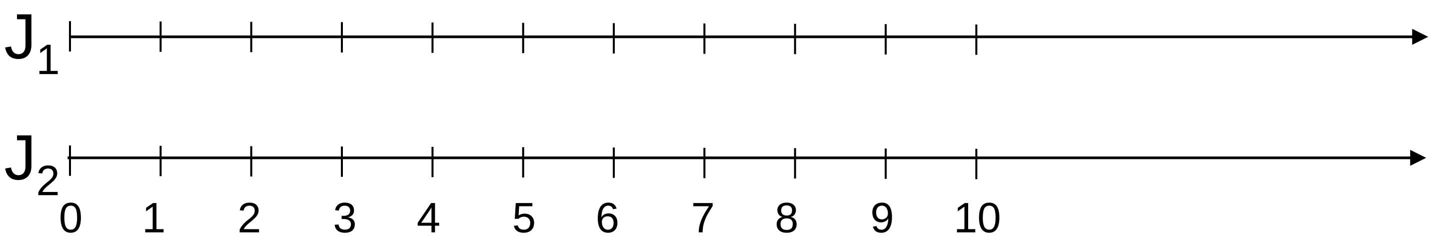
Example

	J_1	J_2
a_i	0	1
C_i	4	2
d_i	7	5

- Non-preemptive EDF schedule:



- Optimal schedule:



Example

- Observation:
  - In the optimal schedule the processor remains idle in intervall [0,1) although task J_1 is ready to execute.
- If arrival times are not known a-priori, then no on-line algorithm is able to decide whether to stay idle at time 0 or to execute J_1.

- **Theorem** (Jeffay et al. '91): EDF is an optimal *non-idle* scheduling algorithm also in a non-preemptive task model.

Non-preemptive scheduling: better schedules through the introduction of idle times

- Assumptions:
  - Arrival times known a priori.
  - Non-preemptive scheduling
  - "Idle schedules" are allowed.
- Goal:
  - Find feasible schedule

- Problem is NP-hard.

- Possible approaches:
  - Heuristics
  - Branch-and-bound

Bratley's algorithm

- Bratley's algorithm
  - Finds feasible schedule by branch-and-bound, if there exists one
  - Schedule derived from appropriate permutation of tasks $J_1, \ldots, J_n$
  - Starts with empty task list
  - Branches: Selection of next task (one not scheduled so far)
  - Bound:
    - Feasible schedule found at current path -> search path successful
    - There is some task not yet scheduled whose addition causes a missed deadline -> search path is blind alley

Bratley's algorithm

- Example:

	J_1	J_2	J_3	J_4
a_i	4	1	1	0
C_i	2	1	2	2
d_i	7	5	6	4

Bratley's algorithm

- Due to exponential worst-case complexity only applicable as off-line algorithm.

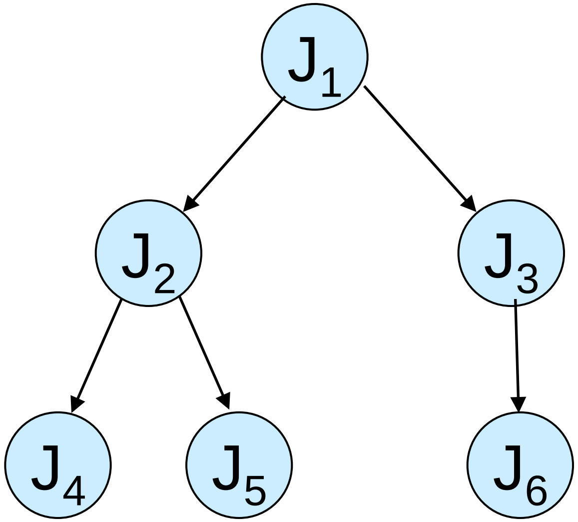
- Resulting schedule stored in task activation list.
- At runtime: dispatcher simply extracts next task from activation list.

Case 3: Scheduling with precedence constraints

- Non-preemptive scheduling with non-synchronous arrival times, deadlines and precedence constraints is NP-hard.
- Here:
  - Restrictions:
    - Consider synchronous arrival times (all tasks arrive at 0)
    - Allow preemption.
  - 2 different algorithms:
    - Latest deadline “first” (LDF)
    - Modified EDF
- Precedences define a partial order
- Scheduling determines a compatible total order
- Method: Topological sorting

Example

	J_1	J_2	J_3	J_4	J_5	J_6
a_i	0	0	0	0	0	0
C_i	1	1	1	1	1	1
d_i	2	5	4	3	5	6



Example

- One of the following algorithms is optimal. Which one?

Algorithm 1:

1. Among all **sources** in the precedence graph select the task T with **earliest** deadline. Schedule T **first**.
2. Remove T from G.
3. Repeat.

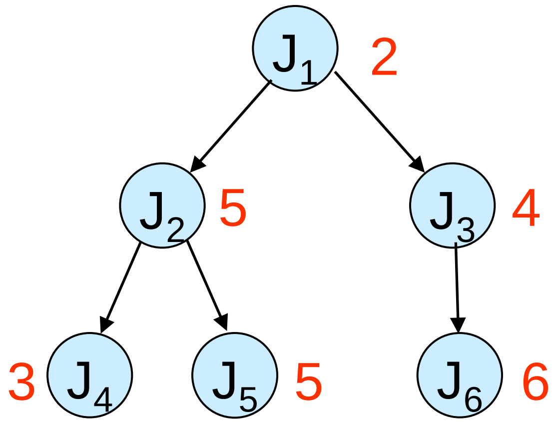
Algorithm 2:

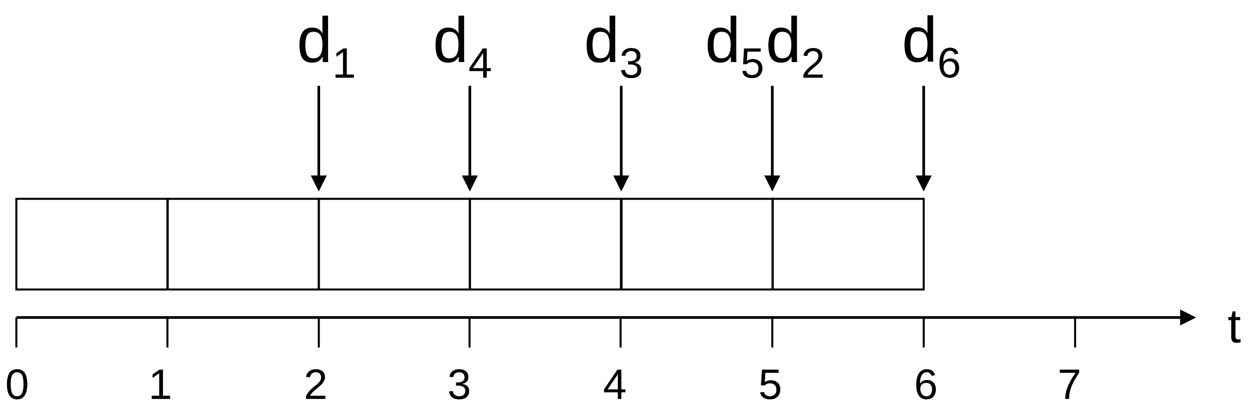
1. Among all **sinks** in the precedence graph select the task T with **latest** deadline. Schedule T **last**.
2. Remove T from G.
3. Repeat.

Example (continued)

- Algorithm 1:

	J_1	J_2	J_3	J_4	J_5	J_6
a_i	0	0	0	0	0	0
C_i	1	1	1	1	1	1
d_i	2	5	4	3	5	6

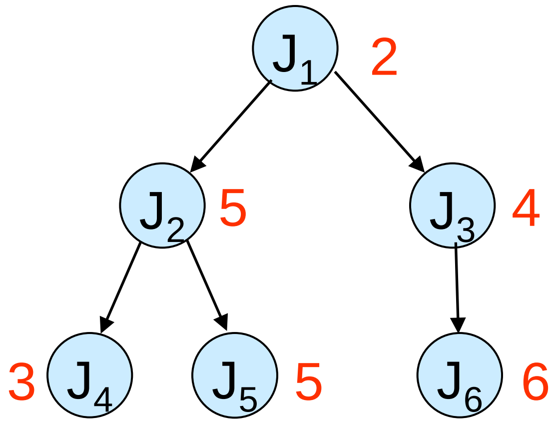


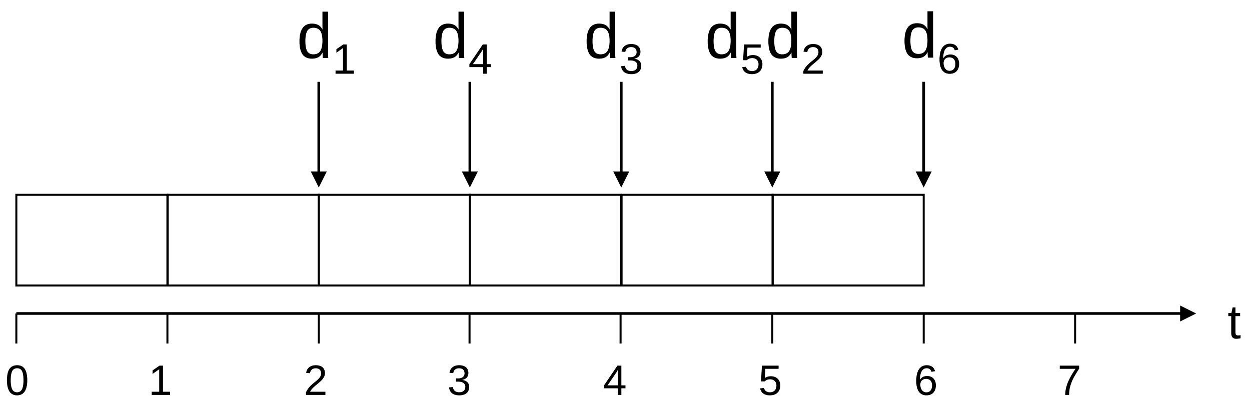


Example (continued)

- Algorithm 2:

	J_1	J_2	J_3	J_4	J_5	J_6
a_i	0	0	0	0	0	0
C_i	1	1	1	1	1	1
d_i	2	5	4	3	5	6





Example (continued)

- Algorithm 1 is **not** optimal.
- Algorithm 1 is the generalization of EDF to the case with precedence conditions.

- Is Algorithm 2 optimal?
- Algorithm 2 is called Latest Deadline First (LDF).

- **Theorem (Lawler 73):**
  LDF is optimal wrt. maximum lateness.

Proof of optimality

LDF

- LDF is optimal.
- LDF may be applied only as off-line algorithm.

- Complexity of LDF:
  - O(|E|) for repeatedly computing the current set Γ of tasks with no successors in the precedence graph G = (V, E).
  - O(log n) for inserting tasks into the ordered set Γ (ordering wrt. d_i).
  - Overall cost: O(n * max(|E|,log n))