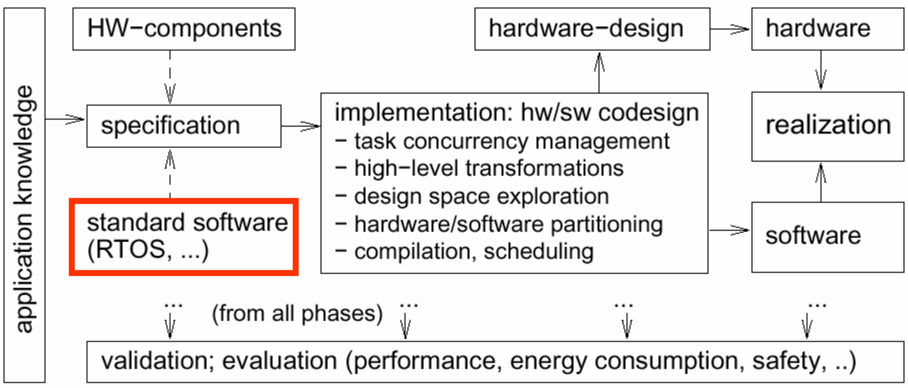
Embedded Systems 14

Overview of embedded systems design



Point of departure: Scheduling general IT systems

- In general IT systems, not much is known about the computational processes a priori
  - The set of processes to be scheduled is open:
    - New software may be inserted into the running system
    - Software is run with "random" activation patterns
  - The power of schedulers thus is inherently limited by lack of knowledge

Scheduling processes in ES: The difference in process charaterization

- Most ES are "closed shops"
  - We know all computational processes which may potentially enter the system
  - We can determine at least part of their activation pattern
    - Regular activation in, e.g., signal processing
    - Maximum activation frequencies of asynchronous events determinable from environment dynamics
  - We should be able to determine their worst-case exectuion time (WCET)
    - If they are well-built
    - If we invest enough analysis effort
- Consequently, we have much better prospects for delivering high-quality schedules!

Scheduling processes in ES: Differences in goals

- In classical OS, quality of scheduling is normally measured in terms of performance:
  - Throughput, reaction times, … in 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.
    - There are hard deadlines which have to be fulfilled under all circumstances and
    - "soft deadlines" which should be fulfilled if possible
  - 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.

Scheduling - Basic definitions

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

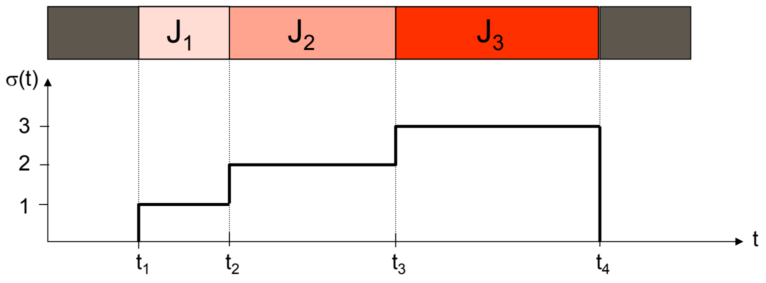
$\forall t \in \mathbb{R}^+ \exists t_1 < t_2 \in \mathbb{R}^+ . t \in [t_1, t_2)$ and $\forall t' \in [t_1, t_2)$ $\sigma(t) = \sigma(t')$.

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.

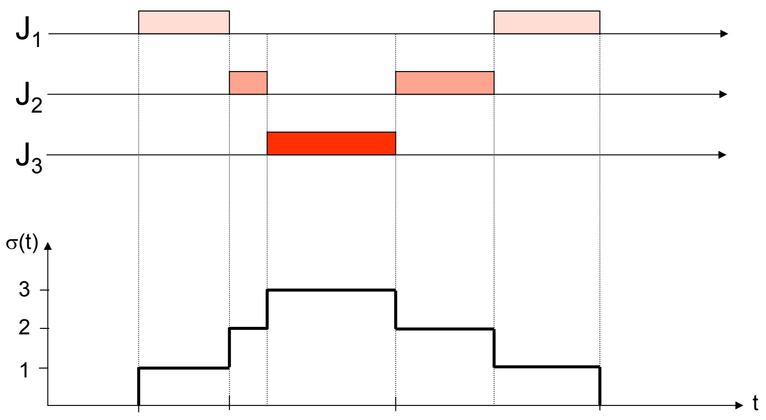
Example

- 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:



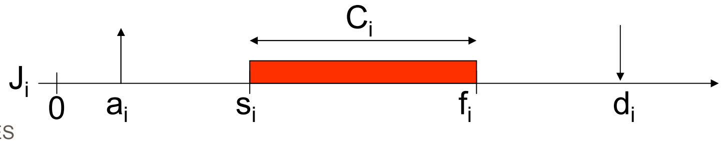
Constraints for real-time tasks

- 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 parameters

- 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 be complete to avoid damage to the system
  - **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



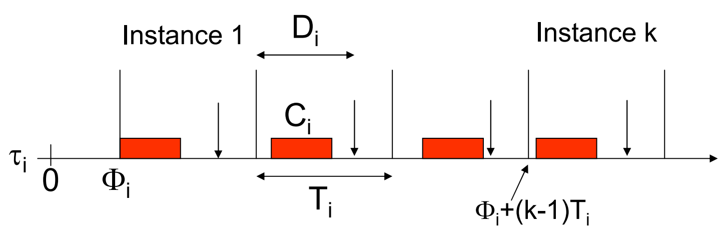
Timing parameters

- Additional timing related parameters of a real-time task J_i:
  - **Criticalness:** 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
    - Aperiodic tasks: Tasks which are not recurring or which do not have regular activations, here abbreviated by J_i

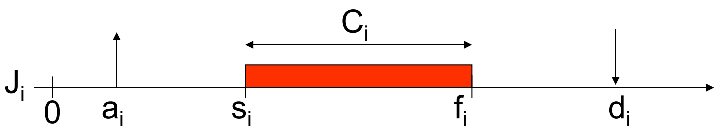
Timing parameters of periodic tasks

- 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 non-periodic tasks

A-periodic scheduling



- Given:
  - A set of non-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

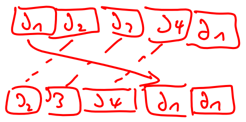
- Not all combinations of constraints, class of algorithm, cost functions can be solved efficiently.
- If there is some information on restrictions wrt. class of problem instances, then this information should be used!
- Begin with simpler classes of problem instances, then more complex cases.

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 L_{max}.



- Consider an arbitrary schedule with max. lateness L_{max}
- Repeat until no more preempted tasks:
  - Select preempted task J that finishes last
  - Reshuffle schedule s.t. all J-slices occur directly before the last J-slice

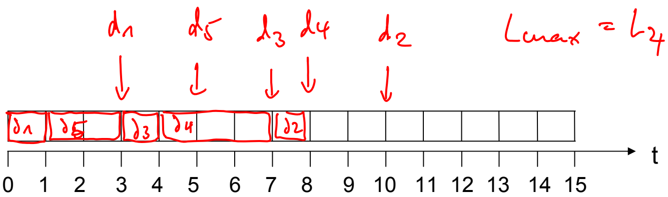
Transformation does not affect lateness of J, may improve lateness of other tasks.

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



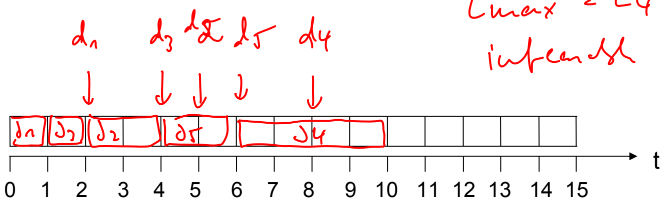
$L_{max} = L_4 = -1$

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

$L_{max} = L_4 = 2$
infeasible schedule

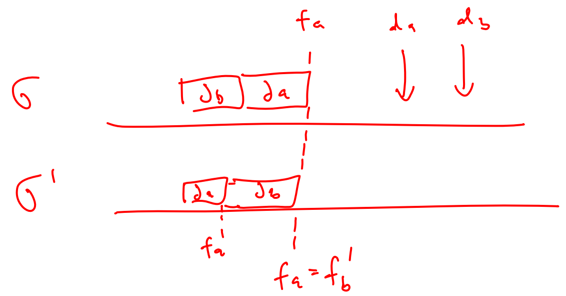


EDD (3)

- 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.

  - Let σ be an arbitrary schedule.
  - We transform σ into a given EDD-schedule σ_{EDD} without increasing L_{max}.
  - W.l.o.g. σ is non-preemptive.
  - Suppose $\sigma \neq \sigma_{EDD}$



$$L'_{max} = \max\{ f'_a - d_a,\ f'_b - d_b \}$$

$$f'_a - d_a < f_a - d_a$$

$$f'_b - d_b = f_a - d_b \le f_a - d_a$$

$$\Rightarrow L'_{max} \le L_{max}$$

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 $f_i \le d_i\ \forall\ i = 1, \ldots, n$.
  - Since $f_i = \sum_{k=1}^{i} C_k$, we have to check
    $\forall\ i = 1, \ldots, n\ \ \sum_{k=1}^{i} C_k \le d_i$

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

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)

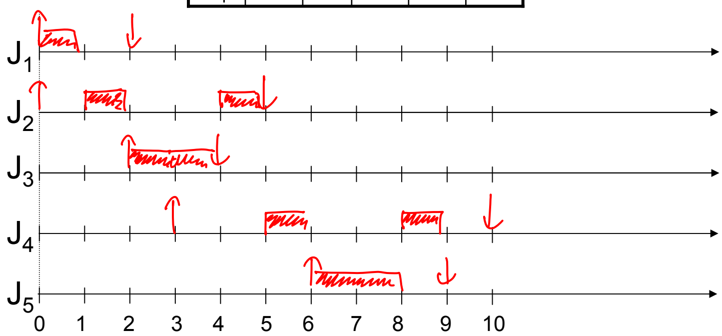
EDF – Earliest Deadline First

- 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.

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

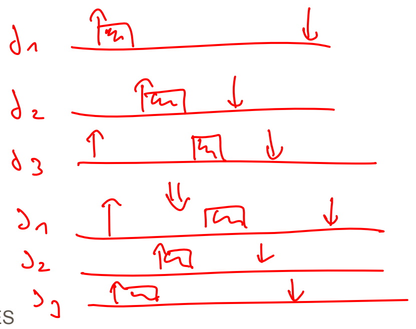
$L_{max} = 0$



EDF

- **Theorem (Horn '74):**
  Given a set of n independent task 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.

Same idea as for Jackson's theorem.



$f_1' - d_1 \le f_3 - d_3$

$f_3' - d_3 \le f_3 - d_3$

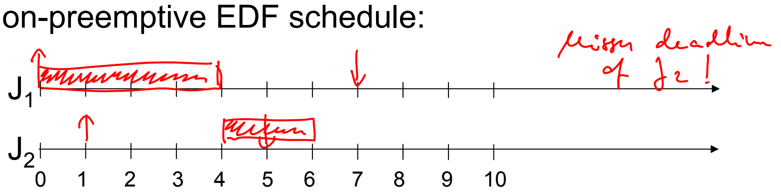
Non-preemptive version

- 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)

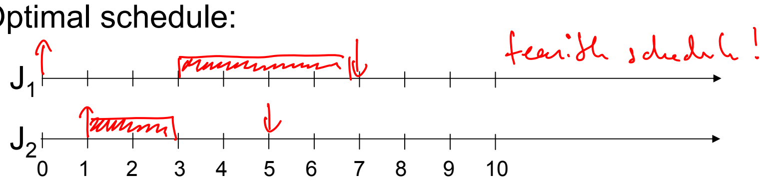
Example

	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 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

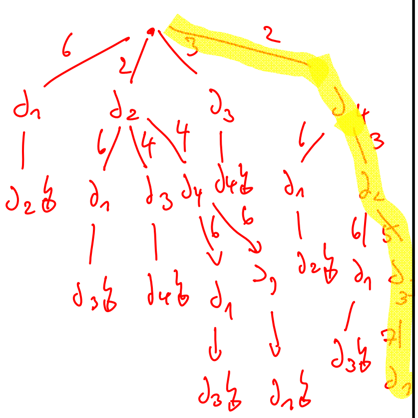
Bratley's algorithm

- 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

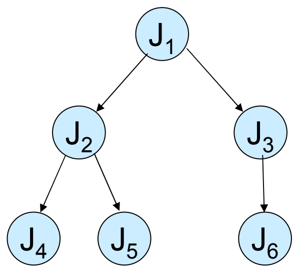
- Due to exponential worst-case complexity only applicable as off-line algorithm.
- Resulting schedule stored in task activation list.
- At runtime: dispatcher simply extracts next task from activation list.

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

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?

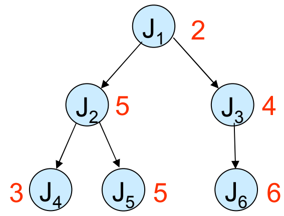
Algorithm 1:

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.

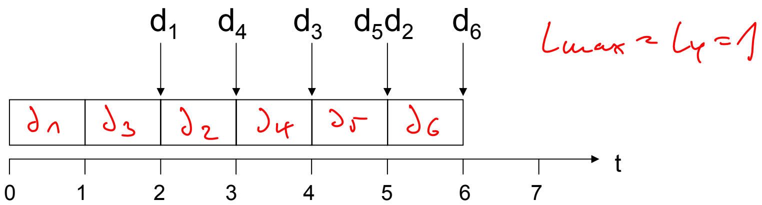
Algorithm 2:

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.

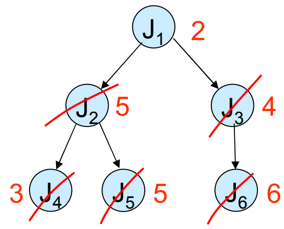
Example (continued)



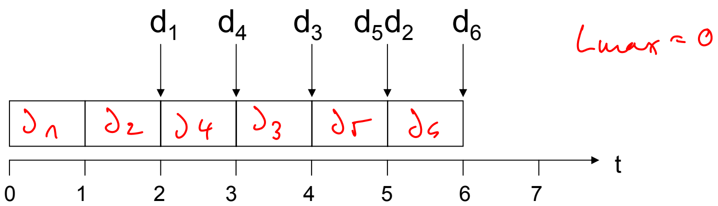
- Algorithm 1:



Example (continued)



- Algorithm 2:



Example (continued)

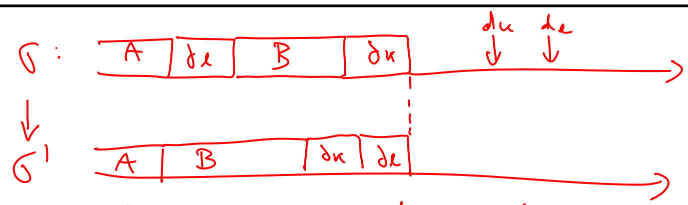
- Algorithm 1 is **not** optimal.
- Algorithm 1 is the generalization of EDF to the case with precedence conditions.

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

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

Proof of optimality

- Let $J = \{J_1, \dots, J_n\}$ be the set of tasks
- Let $T \subseteq J$ be the subset of tasks without successors
- Let J_l be the task in T with the latest deadline
- Consider a precedence-consistent schedule σ, where the last task is J_k with $k \neq l$, $d_k < d_l$
- Clearly, $J_k \in T$.
- We show that moving J_l to the end of the schedule: 1) does not contradict precedence, 2) does not increase L_{max}.



1. σ' does not contradict precedence constraints
→ Only J_l is postponed; J_l has no successors.

2. $L'_{max} = \max\{L'_{max}(A), L'_{max}(B), L'_k, L'_l\}$
- $L'_{max}(A) = L_{max}(A)$ not moved.
- $L'_{max}(B) < L_{max}(B)$ starts earlier.
- $L'_k < L_k$ starts earlier.
- $L'_l = \sum_{i=1}^{n} C_i - d_l < \sum_{i=1}^{n} C_i - d_k = L_k$

⇒ $L'_{max} \le L_{max}$

Remove J_l from J and repeat argument for remaining n-1 tasks.

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))