NTNU | Norwegian University of Science and Technology

Operating Systems

Lecture 13: Real-time scheduling

Michael Engel

Real-time computer systems

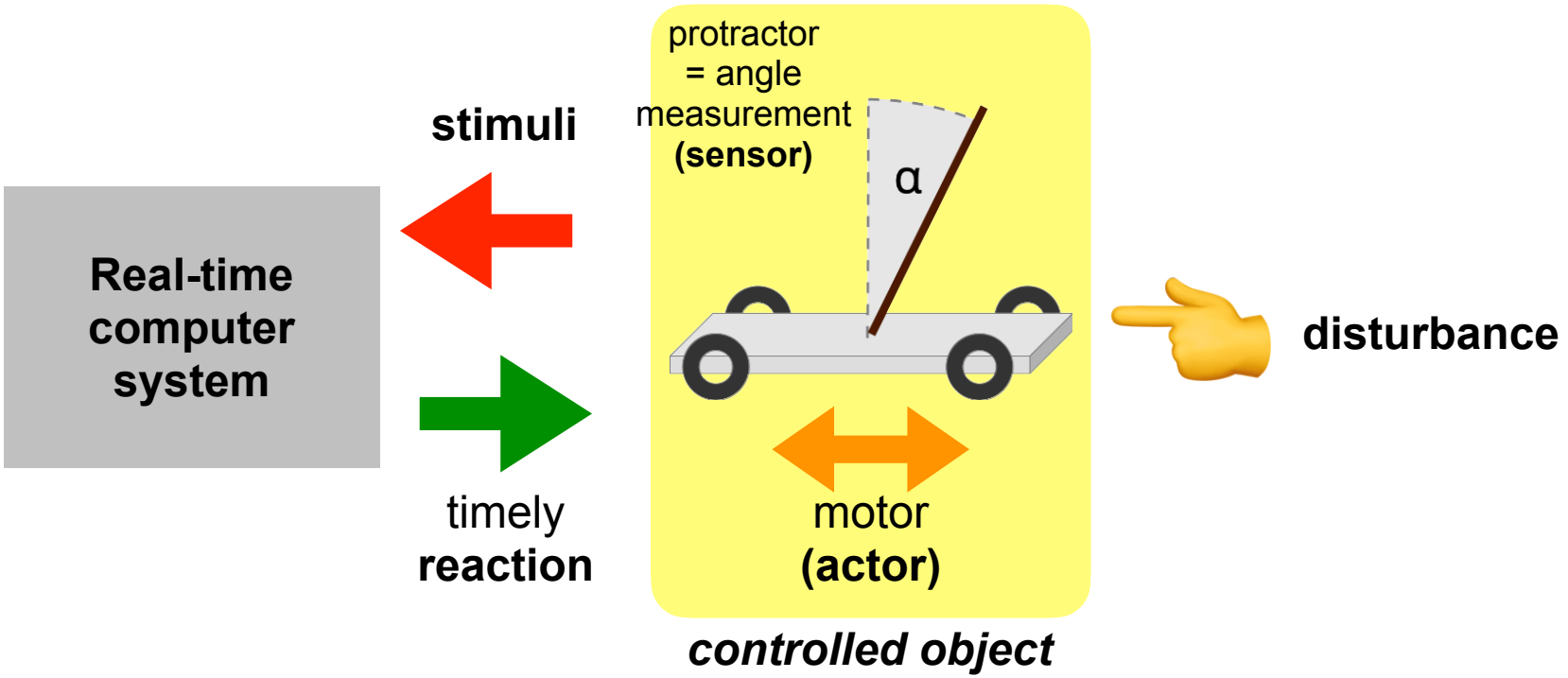
- What's this all about?

> „A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced."

Hermann Kopetz [1]

Example: "inverted pendulum"

Objective: angle α should be = 0°



The reaction time of the computer system (time passing between stimulus and reaction) and its variation ("*jitter*") should be minimized

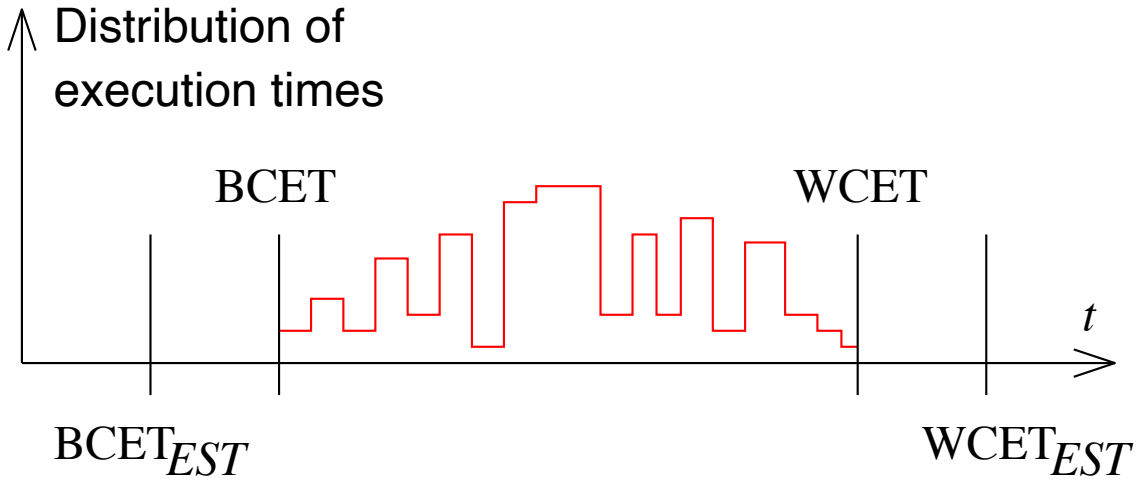
Deadlines

- Often defined by the technical system to be controlled

- Deadline classification:
  - **soft**: the obtained result (the reaction of the system) is useful even if it was obtained after the deadline has passed
  - **firm**: the result is useless after the deadline has passed
  - **hard**: if the deadline passes without a system reaction, damage can occur
- A real-time system is considered "hard" if at *least one of its deadlines* is hard. Otherwise, the system is "soft"
  - For hard real-time systems, it has to be guaranteed that *all deadlines are kept*. This requires different development approaches and system structures.

How long does a program run?

- Runtimes of programs vary due to:
  - different inputs
  - hardware states when the program starts
  - interrupts, process switching, power management, …



The estimated $WCET_{EST}$ has to be ***guaranteed*** larger or equal to the real WCET.

However, the difference between the two should be as small as possible ("*tight bounds*")

- Especially relevant: ***Worst Case Execution Time*** (WCET)

Trigger

… to initiate computation ("*task*") can be realized in different ways:

- ***Event-triggered* real-time systems**
  - A relevant change of the state of the controlled object (an *event*) was observed via sensor readings
  - Scheduling of the tasks at runtime
  - High overhead for tests under high load
  - Behavior is difficult to predict → soft real-time systems

Trigger

… to initiate computation ("*task*") can be realized in different ways:

- **Time-triggered real-time systems**
  - Fixed points in time to execute calculations are planned in advance (*offline scheduling*). Their execution is *periodic*
  - Resource utilization is higher than with event-triggered systems, since the calculation always has to consider the ***worst case execution time*** (WCET)
  - High energy consumption since the system is permanently active
  - Lower test effort required
  - *Guarantees are possible* → **hard real-time systems**

Example: OSEKtime

Objectives of the OSEKtime OS [2]:

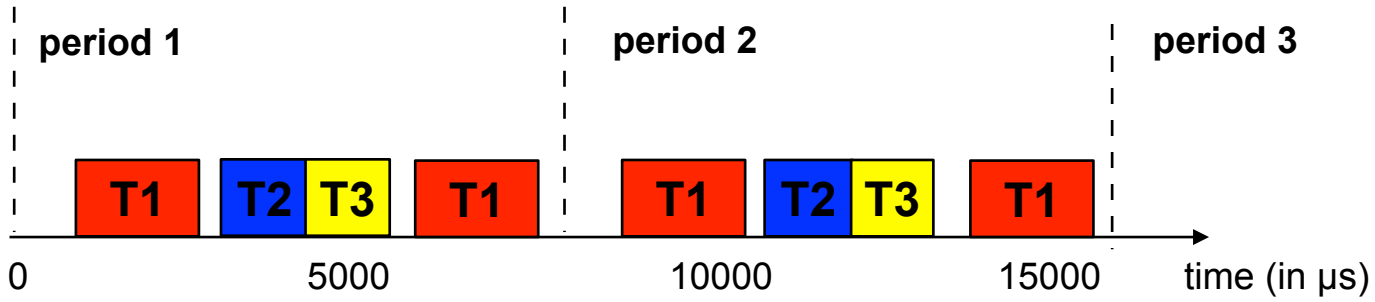
- ***Safe realization*** of "*x-by-wire*" applications, e.g. fly-by-wire, steer-by-wire, brake-by-wire, eGas
  - Guaranteed predictable behavior
    - support for time-triggered applications

    → OSEKtime operating system specification (version 1.0: 2001)
  - Global coordination of *embedded control units* (ECUs):
    - global time!

    → FTCom specification
- Compatibility with "classical" OSEK-OS tasks
  - Support for event-driven applications

OSEKtime scheduler

- **Offline scheduling** :
  - A **dispatch table** controls the *periodic activation* of tasks:



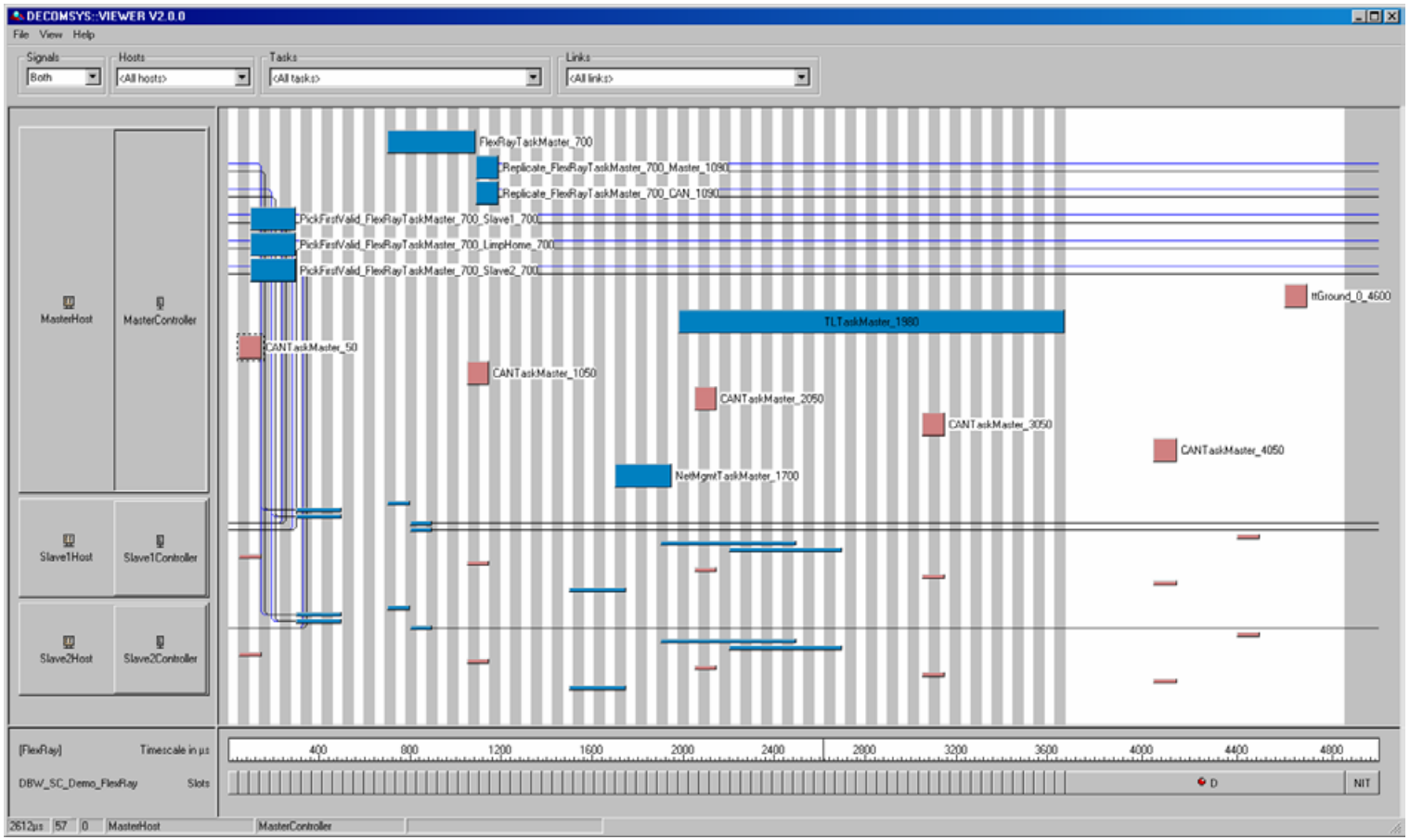
Task	Starting time
T1	1000 µs
T2	3000 µs
T3	4000 µs
T4	5000 µs

Dispatch table the for example. A complete pass through the table is called ***dispatcher round***

- The **dispatcher** is invoked by a timer interrupt
- Only the dispatcher can activate tasks
- Safety mechanism: **deadline monitoring**

Offline scheduling

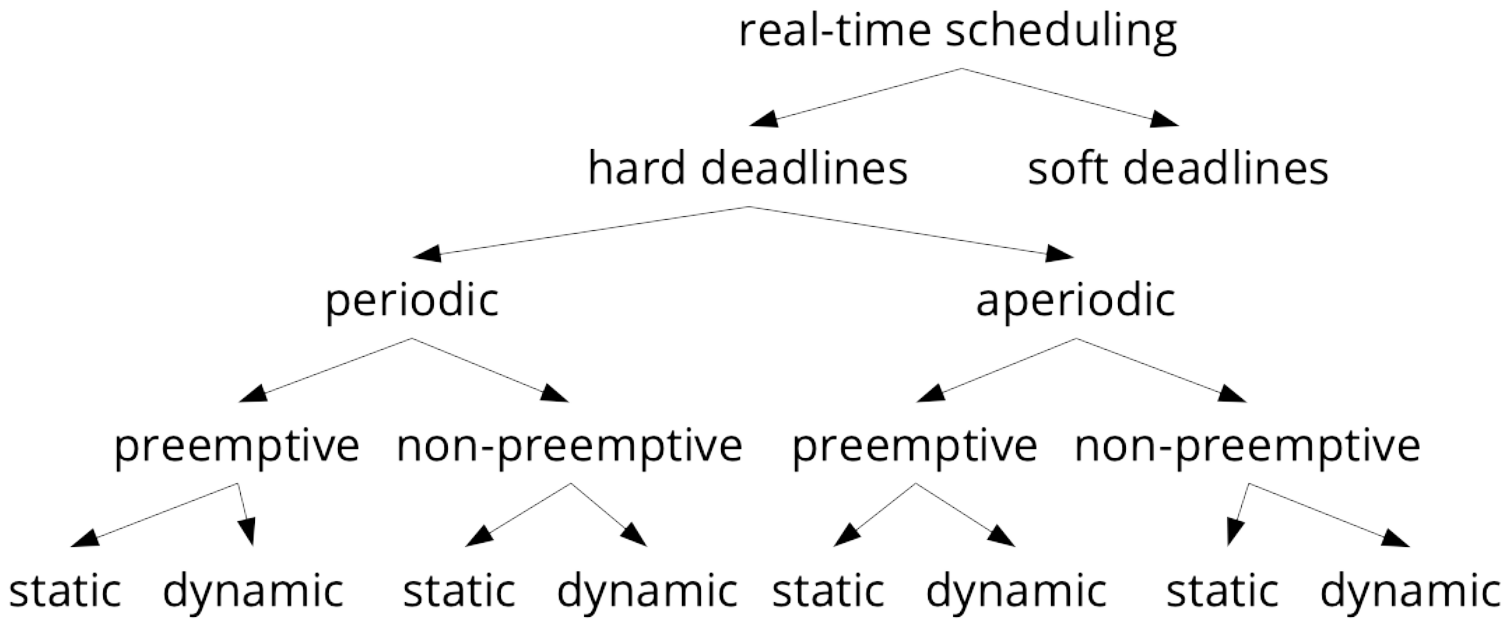
- Tools support developers when scheduling tasks at ***design time***



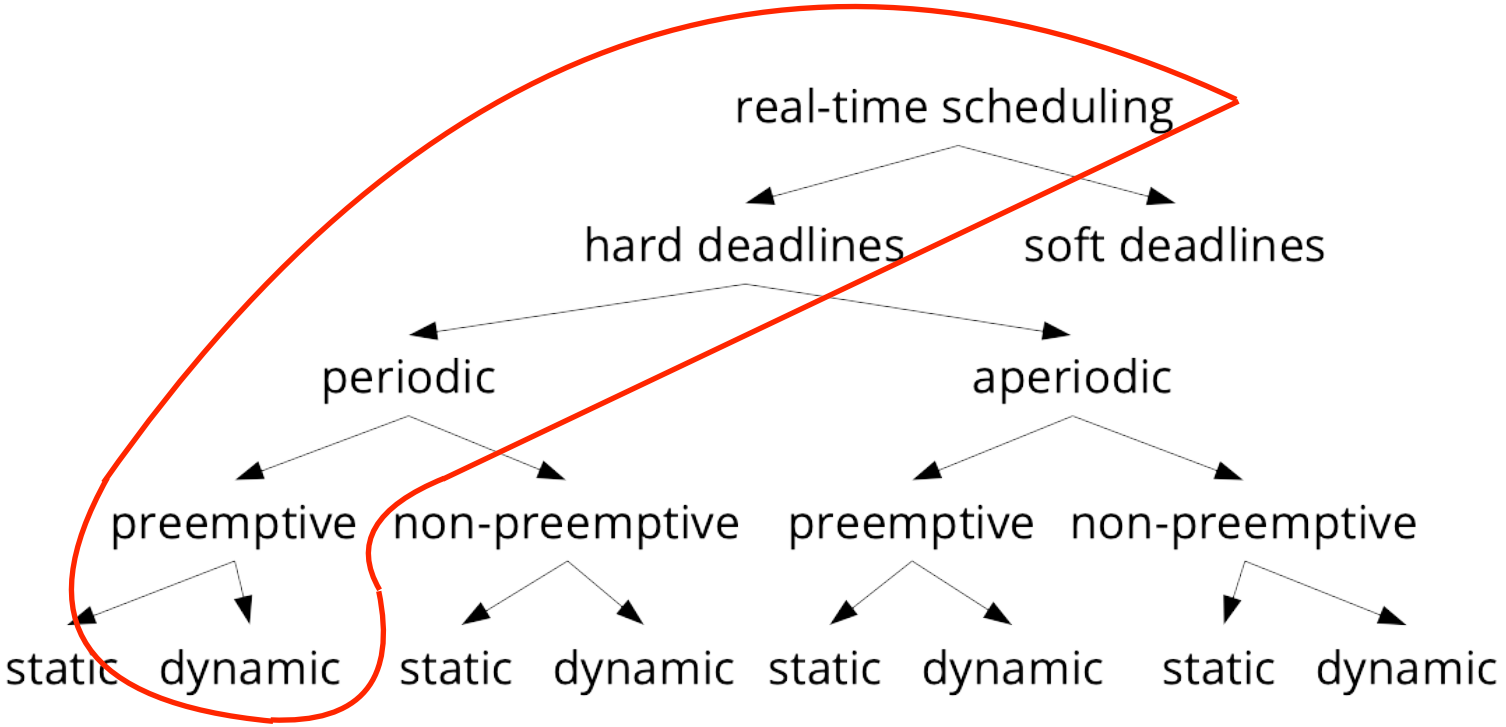
TimeCore

Real-time scheduling

- Objective:
  obtain ***guarantees*** that hard deadlines are kept
- Taxonomy of scheduling approaches [3, chapter 6]:



Rate-monotonic scheduling (RM)



- Rate-monotonic (RM) scheduling is a scheduling strategy for preemptive, periodic tasks with hard deadlines
- The scheduler works at runtime (using fixed priorities)

RM assumptions (Liu & Layland 1973 [4])

A1. All tasks are *preemptible* at any time
Preemption costs (duration) are negligible

A2. Only *compute time* is a relevant resource
The overhead for memory, I/O accesses and other resources is negligible

A3. All tasks are *independent*
There is no required order of execution between tasks

A4. All tasks are *periodic*

A5. The *relative deadline* of a task is equal to its period

Example: car headlight controller

…everything is periodic!

- For each task $\tau_i=(C_i,T_i)$ we know its WCET C_i and period T_i, but not its ***phase*** ϕ_i

Indicators:

- $\tau_5 = (1,500)$

Stepper motor control:

- $\tau_4 = (1,5)$

Status messages:

- $\tau_1 = (2,20)$

Control messages:

- $\tau_2 = (2,10)$

Network management:

- $\tau_3 = (4,50)$

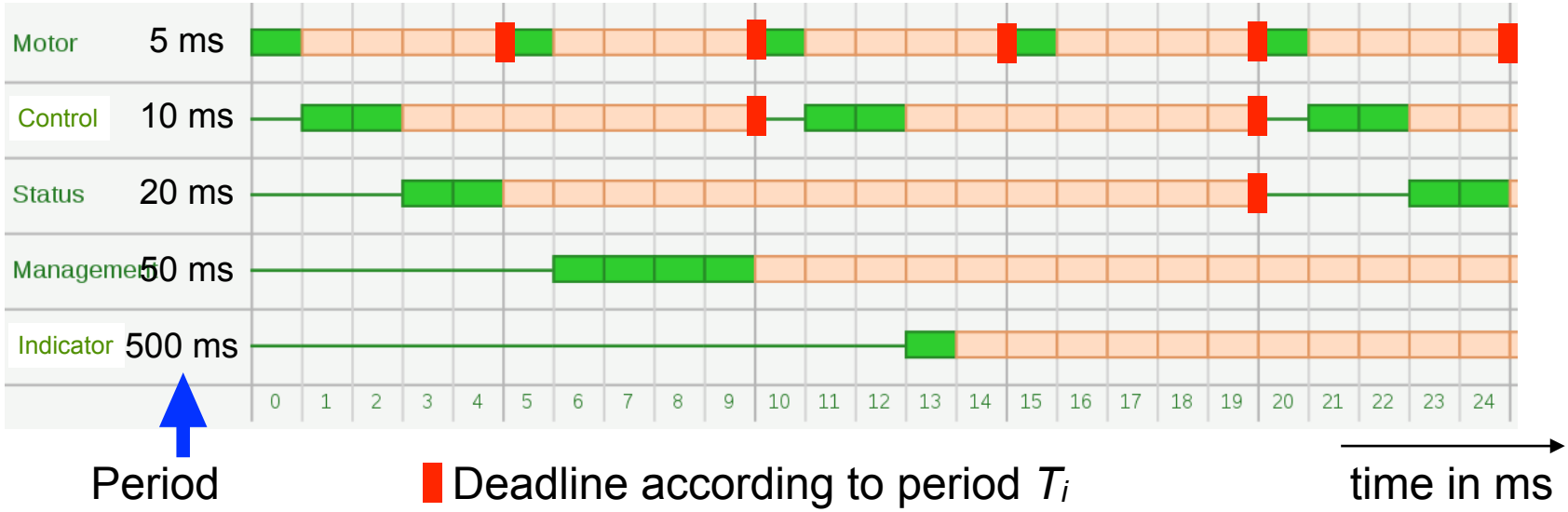
CAN bus

RM algorithm

- The priority grows monotonously with the event rate (=frequency)
- Thus: short period → high priority
- Tasks with high priority ***preempt*** tasks with low priority
- Example:

A practical implementation of RM scheduling requires only an operating system with a ***preemptive fixed priority scheduler***

Gantt diagram for $\phi_i = 0$



Schedulability analysis

- Question: ***are the deadlines kept for all tasks?***
  - The schedule can only be calculated if all tasks are completely time-triggered. In our example, the phases can have arbitrary values
- Necessary condition: the ***utilization U*** of the system is less than or equal to 1:

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

Assumption: Uniprocessor

U: system load
m: number of tasks

- Example: $\tau_1 = (1,5)$, $\tau_2 = (2,20)$, $\tau_3 = (2,10)$, $\tau_4 = (4,50)$, $\tau_5 = (1,500)$

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i} = \frac{1}{5} + \frac{2}{20} + \frac{2}{10} + \frac{4}{50} + \frac{1}{500} = 0{,}582 \leq 1$$

But… is this sufficient?

The “70% rule” [4]

- Rule: ***no deadline violations if the following condition holds:***

$$U \leq m \cdot \left(2^{1/m} - 1\right)$$

U: system load
m: number of tasks

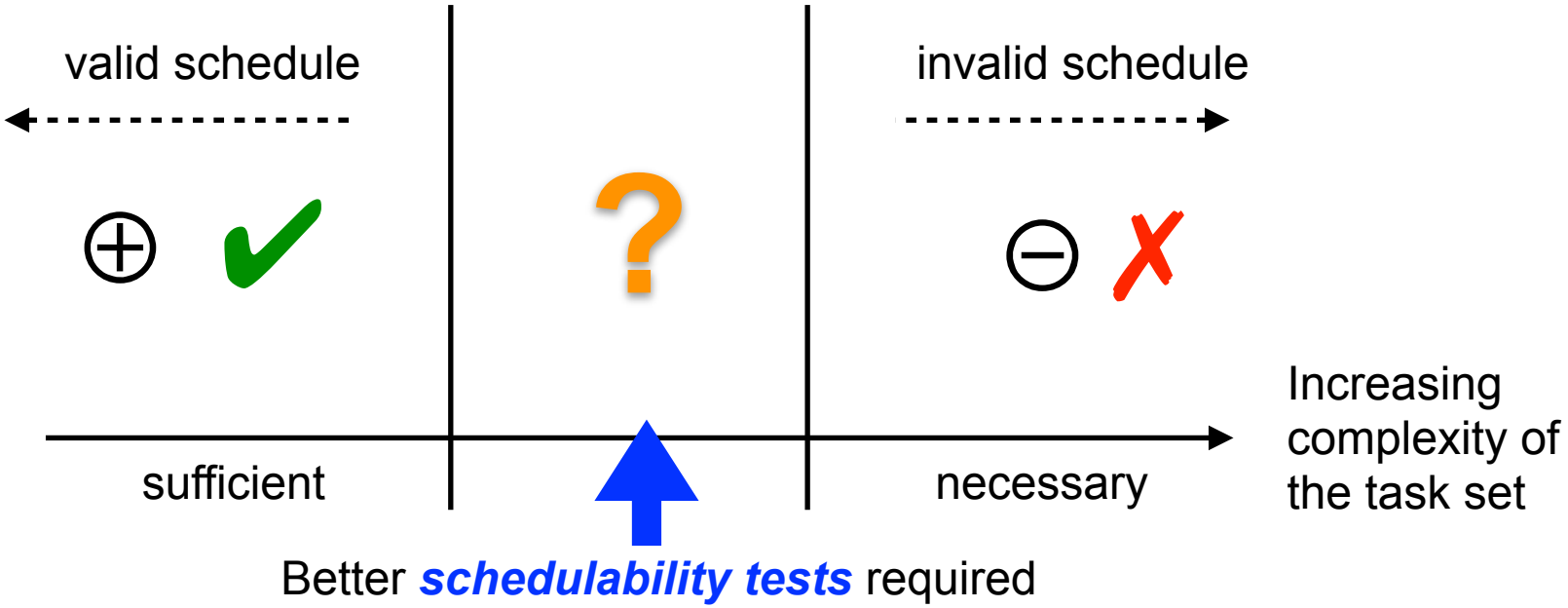
- For large values of *m*, this converges against **ln(2) ≈ 0,6931**, i.e. ca. **70%**
- Advantage: simple test, low overhead
- Example 1: $U = 58.2\%$, $m = 5$
  - $m \cdot (2^{1/m} - 1) = 74.35\%$, condition fulfilled → no deadline violation ✔
- Example 2: $\tau_1 = (2,5)$ instead of $\tau_1 = (1,5)$, thus $U = 78.2\%$, $m = 5$
  - $m \cdot (2^{1/m} - 1) = 74.35\%$, condition ***not*** fulfilled
    → **possible deadline violation**
- Disadvantage: *no conclusion* if the condition is not fulfilled

Tip: apply *L'Hôpital's rule*

Sufficient and necessary conditions

- Sufficient condition positive
- e.g. $U \leq m \cdot \left(2^{1/m} - 1\right)$
- Schedule is ***valid***

- Necessary condition negative
- e.g. $U \leq 1$ does not hold
- Schedule is ***invalid***



Ideal case is an "exact test": sufficient and necessary condition

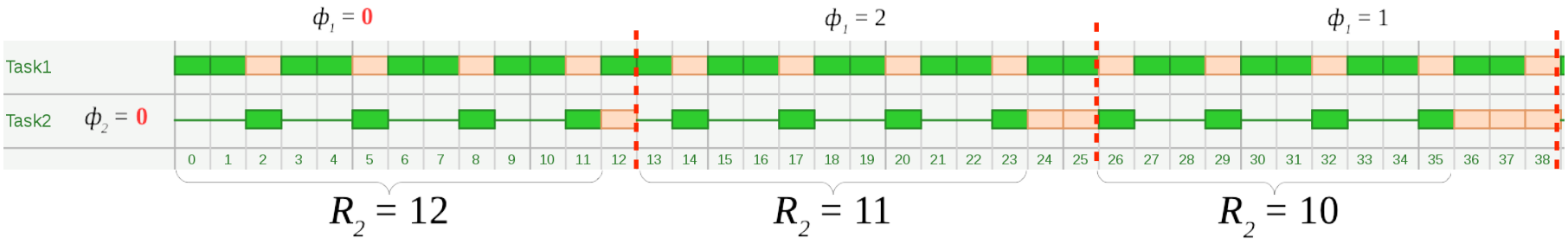
Exact test: response time analysis [5]

- If the response time R_i for all tasks is less than or equal to the period T_i, all deadlines are kept
- For the largest possible delay $\phi_i = 0$:

  All higher prioritized tasks are ready at the start of the period

Condition (necc. and suff.):

$$\forall i \in \{1, \ldots, m\} : R_i \leq T_i$$



- Calculate R_i:

$$R_i = C_i + I_i = C_i + \sum_{j \in hp_i} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

I_x: "Interference" – delay caused by tasks with higher priority

hp_x: indexes of the tasks that have a higher priority than task x

$\lceil x \rceil$: rounding up to next integer

Exact test: iterative solution

- Calculate R_i using fixed point iteration:
  - Terminate if $R_i^{n+1} = R_i^n$ or $R_i^{n+1} > T_i$

$$R_i^{n+1} = C_i + \sum_{j \in hp_i} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j$$

  - Pseudo code of the tests for all tasks:

```
for (each task τi) {
    I = 0
    do {
        R = I + Ci
        if (R > Ti) return false // deadline violation
        I = Σ_{j∈hpi} ⌈R/Tj⌉ · Cj
    } while (I + Ci > R)
}
return true // all deadlines are kept
```

Rate-monotonic scheduling is "optimal"

- We need to show:

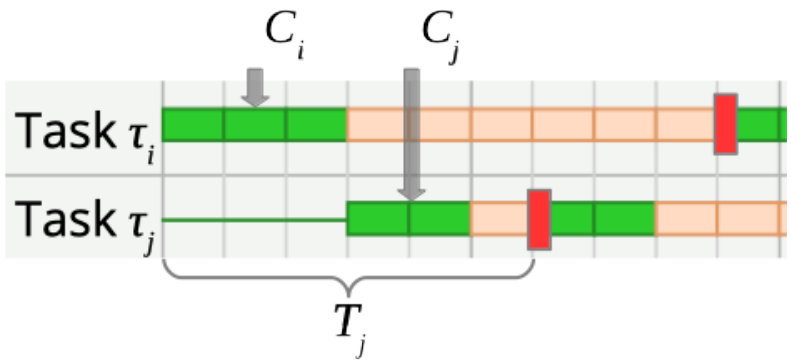
RM is an optimal scheduling algorithm for fixed priorities. I.e., if any algorithmus can find a valid schedule, RM can also find one.

- Proof by contradiction: we assume…
  algorithm A finds a valid schedule, but RM does not

$$\neg(A \Rightarrow B) \Leftrightarrow A \wedge \neg B$$

  - In schedule A: prio(τ_i) = prio(τ_j) + 1 and $T_i > T_j$ (different to RM)

$C_i + C_j \leq T_j$ holds since the schedule is valid and τ_i has a higher priority



Rate-monotonic scheduling is "optimal"

- We need to show:

RM is an optimal scheduling algorithm for fixed priorities. I.e., if any algorithmus can find a valid schedule, RM can also find one.

- Proof by contradiction: we assume…
  algorithm A finds a valid schedule, but RM does not
  - In schedule A: prio(τ_i) = prio(τ_j) + 1 and $T_i > T_j$ (different to RM)

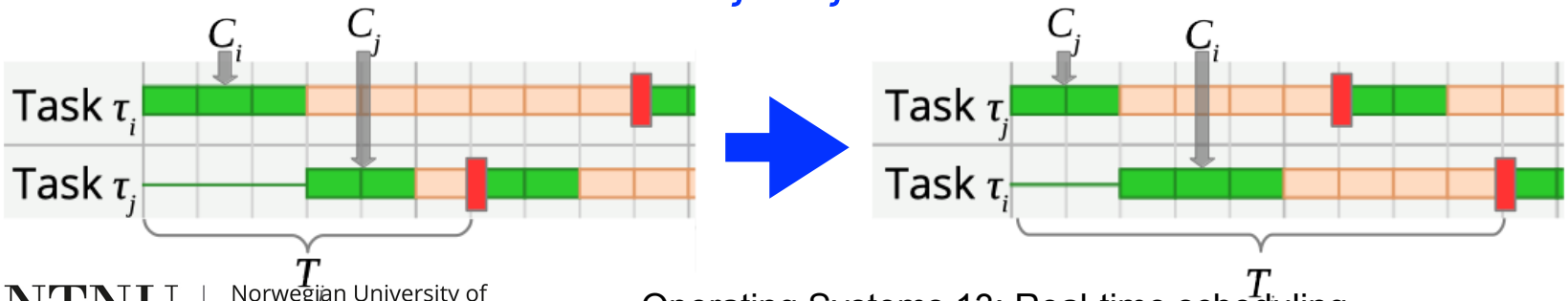
$\neg(A \Rightarrow B) \Leftrightarrow A \wedge \neg B$

$C_i + C_j \leq T_j$ holds since the schedule is valid and τ_i has a higher priority

What is the effect of swapping the priorities (only) of these two tasks?

τ_j can be scheduled, since it now has higher priority

τ_i can also be scheduled since $C_i + C_j \leq T_j < T_i$



Rate-monotonic scheduling is "optimal"

- We need to show:

RM is an optimal scheduling algorithm for fixed priorities. I.e., if any algorithmus can find a valid schedule, RM can also find one.

- Proof by contradiction: we assume…
  algorithm A finds a valid schedule, but RM does not
  - In schedule A: prio(τ_i) = prio(τ_j) + 1 and $T_i > T_j$ (different to RM)

$\neg(A \Rightarrow B) \Leftrightarrow A \wedge \neg B$

$C_i + C_j \leq T_j$ holds since the schedule is valid and τ_i has a higher priority

What is the effect of swapping the priorities (only) of these two tasks?

τ_j can be scheduled, since it now has higher priority

τ_i can also be scheduled since $C_i + C_j \leq T_j < T_i$

We obtain an RM schedule by applying a finite number of these swaps

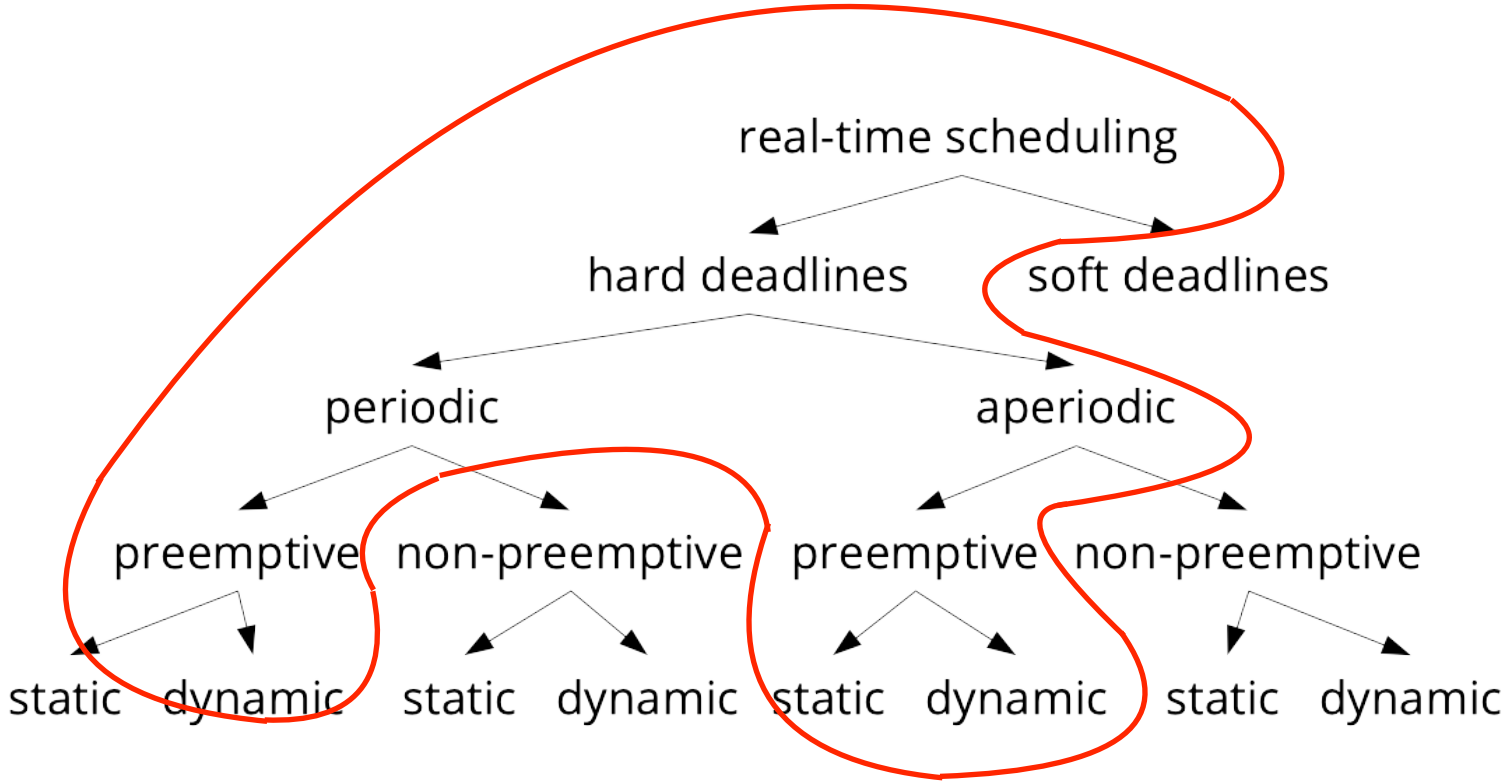
*This is also a valid schedule → **contradiction** → RM is optimal!*

RM scheduling: conclusion

- RM is easy to apply and optimal for fixed priorities
  - the OS only needs to provide a "fixed priority" scheduler
- Response time analysis enables an exact schedulability test
  - Important for hard real-time systems: mathematical guarantee!
- In many cases, the 70% rule is sufficient

- Attention: 
  - Assumptions A.1-5 must hold!
    - uniprocessor, no task dependencies, …
  - WCET estimation difficult for modern processors
    - memory hierarchies, out-of-order execution, DRAM access times, ...
  - In any case, the *complete system* has to be analyzed

Example: Earliest Deadline First



- **Earliest Deadline First (EDF)** scheduling is a scheduling strategy for preemptive, periodic and aperiodic tasks with hard deadlines. The priorities are assigned dynamically (at runtime).

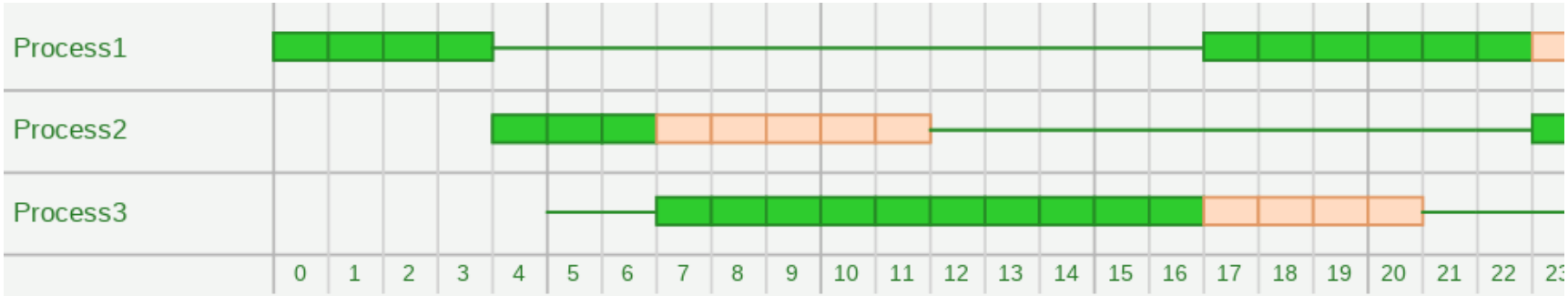
EDF algorithm

- Tasks which are ready are sorted in order of their ***absolute*** deadlines
- If the first task in the list has an earlier deadline than the currently running task, the running task is preempted immediately!

In general, deadlines are specified as relative times

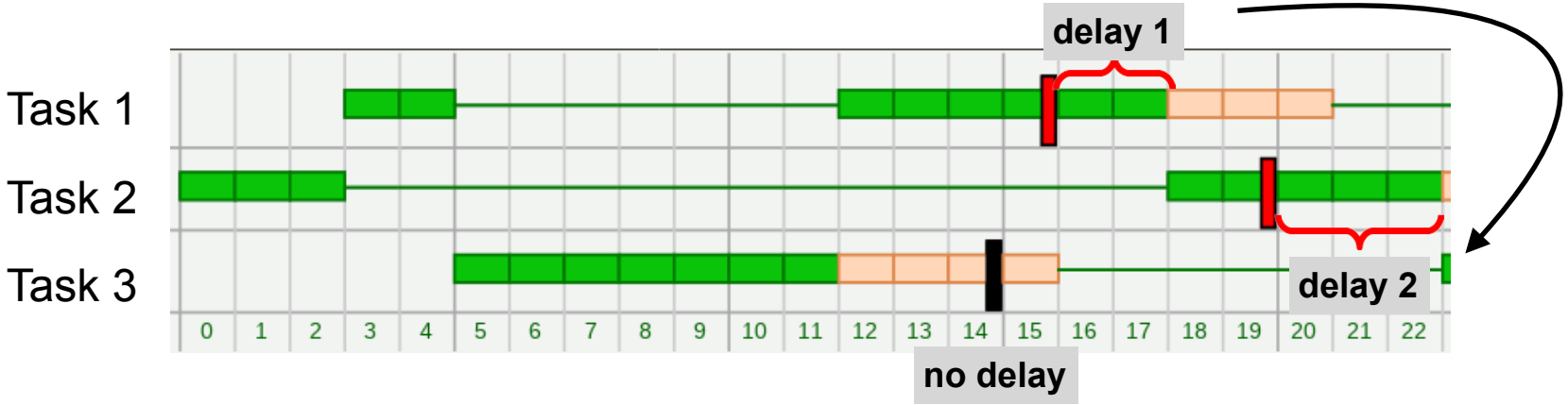
Example:

Process name	Arrival	CPU burst	IO burst	Deadline
Process1	0	10	3	33
Process2	4	3	5	24
Process3	5	10	3-5	24



Optimality of EDF

- EDF minimizes the ***maximum delay*** of tasks



- If a schedule exists which is able to keep all deadlines, then EDF also keeps all deadlines → **EDF is optimal**
  - …for independent tasks with dynamic priorities
- Especially for **periodic** tasks the following holds:
  If $U \leq 1$, then EDF always finds a valid schedule
  (without missing deadlines!)

Proof in [6]

EDF-Scheduling: Conclusion

- Simply **optimal** for periodic as well as aperiodic task sets
  - Achieves a higher utilization than RM scheduling by using dynamic priorities
- ***Attention***: 
  - EDF is usually only implemented in special ***real-time operating systems***
  - No information about the number and duration of deadline misses can be obtained
  - Less predictable than e.g. RM
  - Response times can vary significantly: "***jitter***"
  - In overload situations: "***domino effect***"

Outlook: Extending the strategies

- Working with **sporadic tasks**
  - Limited arrival rate, but no strict period
- Consideration of task dependencies
- Increase CPU utilization
  - **mixed-criticality** systems
  - restriction to "**harmonic tasks**"
    - periods are integer multiples of each other
- **Modus changes**
  - e.g. indicator/stepper motor becomes active
- Handle [temporary] overload
- Adaptation to [heterogeneous] multiprocessor systems

References

[1] Kopetz, Hermann: Real-Time Systems: Design Principles for Distributed Embedded Applications (2nd. ed.). Springer Publishing Company, Inc., 2011. https://doi.org/10.1093/comjnl/29.5.390

[2] Automotive Open System Architecture – http://www.autosar.org

[3] Peter Marwedel. 2021. Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems (4th ed.). Springer Publishing Company, Incorporated. Open access: https://www.springer.com/gp/book/9783030609092

[4] C. L. Liu and J. W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. J. ACM20, 1 (January 1973), 46-61. DOI=http://dx.doi.org/10.1145/321738.321743

[5] M. Joseph and P. Pandya. 1986. Finding response times inreal-time systems, BCS Computer Journal, 29 (5): 390–395, DOI=https://doi.org/10.1093/comjnl/29.5.390

[6] G. C. Buttazzo. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers, USA, 1997