TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

BETRIEBSSYSTEME UND SICHERHEIT

mit Material von Olaf Spinczyk, Universität Osnabrück

Schedulingstrategien für Echtzeitsysteme

https://tud.de/inf/os/studium/vorlesungen/bs

HORST SCHIRMEIER

Inhalt

- Echtzeitsysteme
- Beispiel: OSEKtime
- Echtzeit-Scheduling-Strategien
- *Rate Monotonic Scheduling*
- *Earliest Deadline First Scheduling*
- Ausblick

Inhalt

- **Echtzeitsysteme**
- Beispiel: OSEKtime
- Echtzeit-Scheduling-Strategien
- *Rate Monotonic Scheduling*
- *Earliest Deadline First Scheduling*
- Ausblick

Echtzeitcomputersysteme

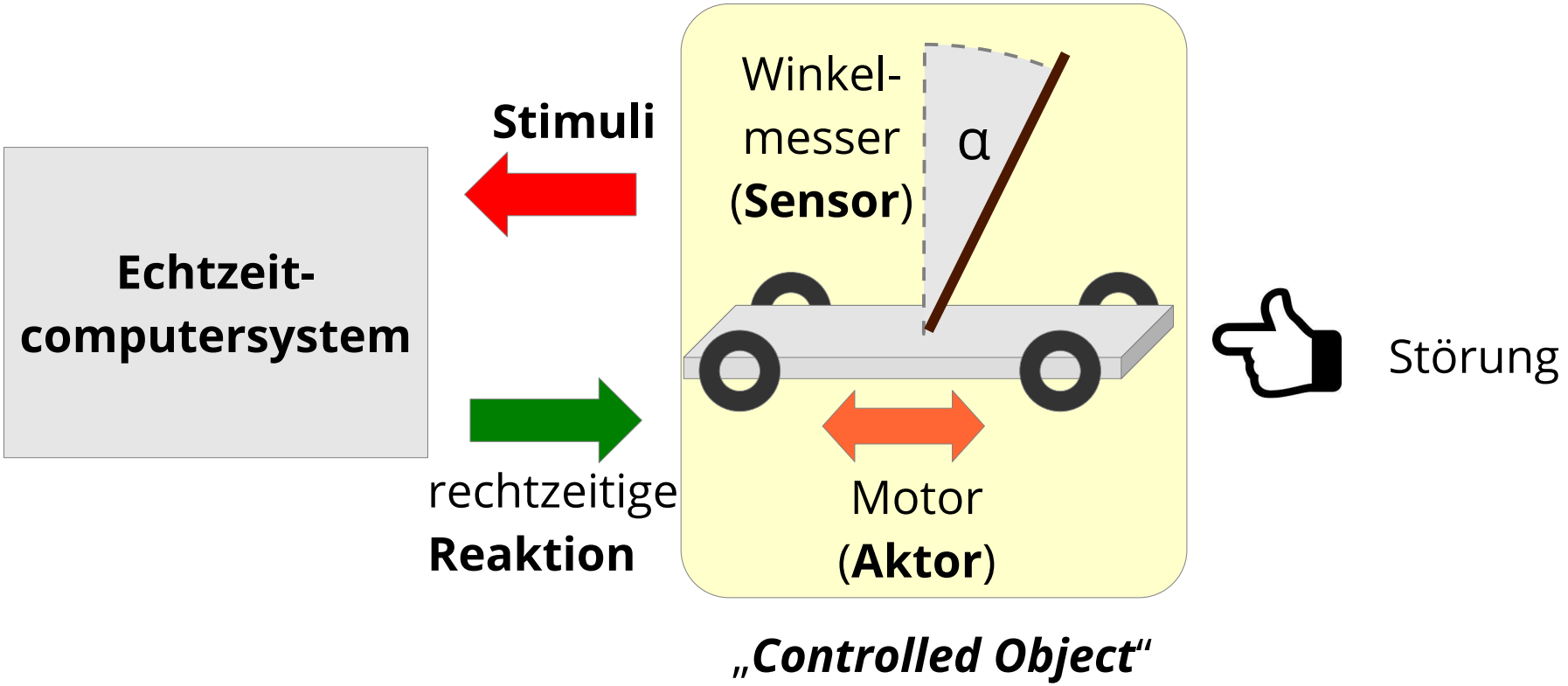
- Was ist das?

> *„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]

Beispiel „Inverses Pendel“

Ziel: α soll 0° betragen



„Controlled Object“

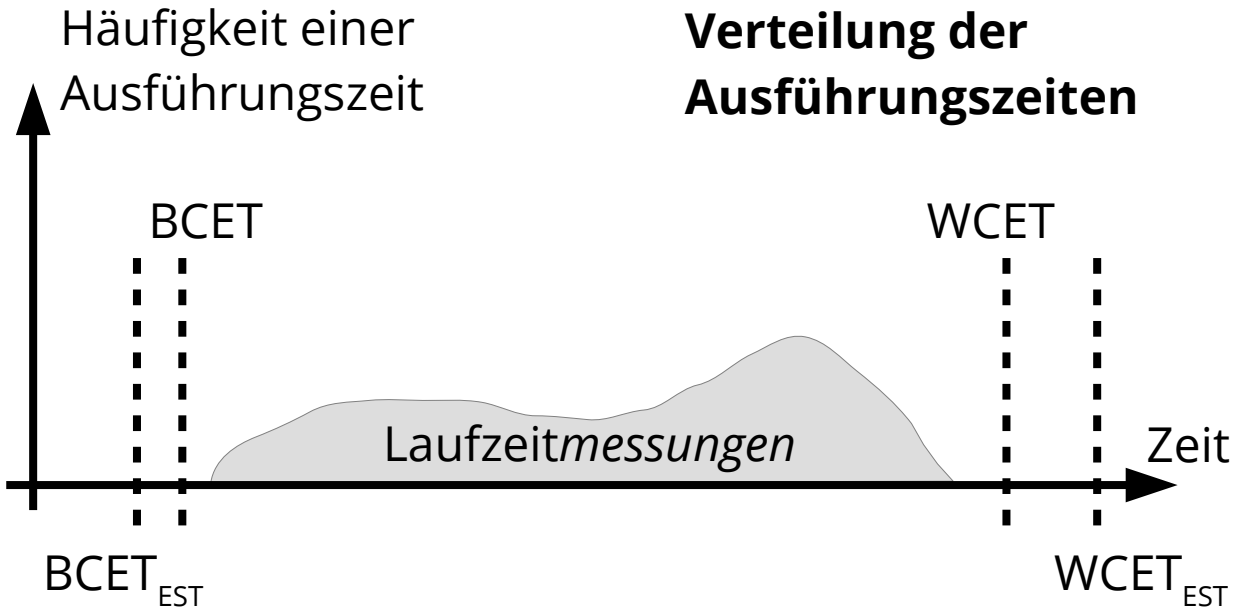
Die Reaktionszeit des Computersystems (Zeitraum zwischen dem Stimulus und der Reaktion) sowie der Schwankung (*„Jitter“*) sollten minimal sein.

Fristen („*deadlines*“)

- Sind durch das gesteuerte technische System vorgegeben
- Werden wie folgt klassifiziert:
  - ***soft:*** Auch wenn die Frist verstrichen ist, ist das berechnete Ergebnis (die Reaktion) noch nützlich.
  - ***firm:*** Wenn die Frist verstrichen ist, ist das Ergebnis wertlos.
  - ***hard:*** Wenn die Frist ohne Reaktion verstrichen ist, entsteht ein Schaden.
- Ein Echtzeitsystem gilt als „hart“, wenn mindestens eine Frist hart ist. Sonst gilt es als „weich“.
  - Bei harten Echtzeitsystemen muss die Einhaltung der Fristen garantiert werden. Das impliziert andere Entwicklungsmethoden und Systemstrukturen.

Wie lange rechnet ein Programm?

- Laufzeiten variieren: Unterschiedliche Eingabeparameter, Hardwarezustände beim Start, Unterbrechungen, Prozesswechsel, Power Management, …



Die geschätzte $WCET_{EST}$ muss garantiert größer oder gleich der wahre WCET sein. Der Abstand sollte aber möglichst gering sein (engl. *tight bounds*).

- Besonders wichtig: ***Worst Case Execution Time*** **(WCET)**

Auslöser („*trigger*“) …

… für die Berechnung („***Task***“) können unterschiedlich realisiert werden:

- **Ereignisgesteuerte Echtzeitsysteme**
  - Eine relevante Zustandsänderung (ein Ereignis – „*Event*“) des kontrollierten Objekts wurde an einem Sensor festgestellt.
  - Einplanung („*Scheduling*“) der Tasks erfolgt zur Laufzeit.
  - Hoher Aufwand für Tests im Hochlastbetrieb.
  - Vorhersagen schwierig → **weiche Echtzeitsysteme**

Auslöser („*trigger*“) …

… für die Berechnung („***Task***“) können unterschiedlich realisiert werden:

- **Zeitgesteuerte Echtzeitsysteme**
  - Feste Zeitpunkte für Berechnungen werden vorab geplant („*offline scheduling*“). Die Ausführung erfolgt periodisch.
  - Höherer Ressourcenverbrauch, da der Kalkulation die ***Worst Case Execution Time*** zugrunde liegen muss.
  - Hoher Energieverbrauch, kontinuierlich aktiv
  - Weniger Testaufwand
  - Garantien möglich → **harte Echtzeitsysteme**

Inhalt

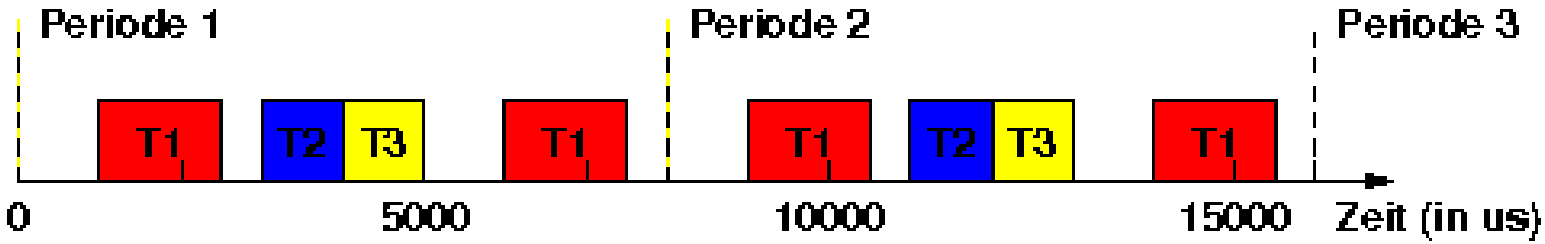
- Echtzeitsysteme
- **Beispiel: OSEKtime**
- Echtzeit-Scheduling-Strategien
- *Rate Monotonic Scheduling*
- *Earliest Deadline First Scheduling*
- Ausblick

OSEKtime [2]: Ziele

- Sichere Realisierung von ***„X-By-Wire“*** Anwendungen (*Steer-by-wire*, *Brake-by-wire*, eGas)
  - Garantiertes vorhersagbares Verhalten
    - Unterstützung für zeitgesteuerte Anwendungen
    - → OSEKtime Betriebssystem-Spezifikation (Version 1.0: 2001)
  - Globale Koordinierung im Steuergerätenetz
    - Globale Zeit!
    - → FTCom-Spezifikation
- Kompatibilität mit „klassischen“ OSEK-OS-*Tasks*
    - Unterstützung für ereignisgesteuerte Anwendungen

OSEKtime: *Scheduler*-Arbeitsweise

- ***Offline Scheduling***: Eine ***Dispatcher*-Tabelle** steuert die periodische Aktivierung von Tasks:



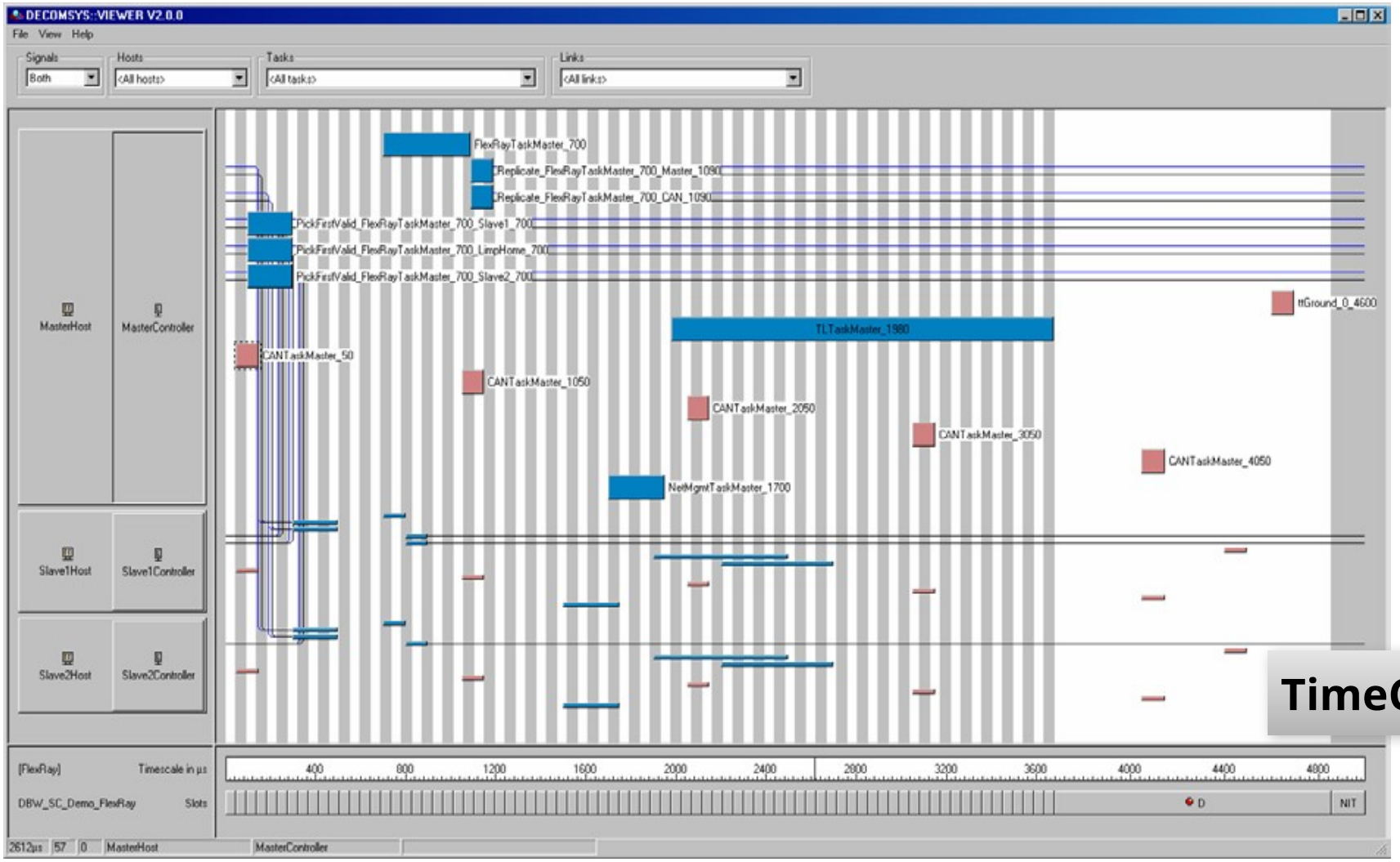
Task	Startzeitpunkt
T1	1000 µs
T2	3000 µs
T3	4000 µs
T1	6000 µs

Zum Beispiel passende *Dispatcher*-Tabelle.
Ein kompletter Durchlauf heißt *„Dispatcher Round“*.

- Für die Ausführung des ***Dispatchers*** sorgt ein *Timer-Interrupt*.
- Nur der *Dispatcher* darf *Tasks* aktivieren.
- Sicherheitsmechanismus: ***Deadline Monitoring***

Offline Scheduling

- Werkzeuge helfen dem Entwickler die *Tasks* anzuordnen.



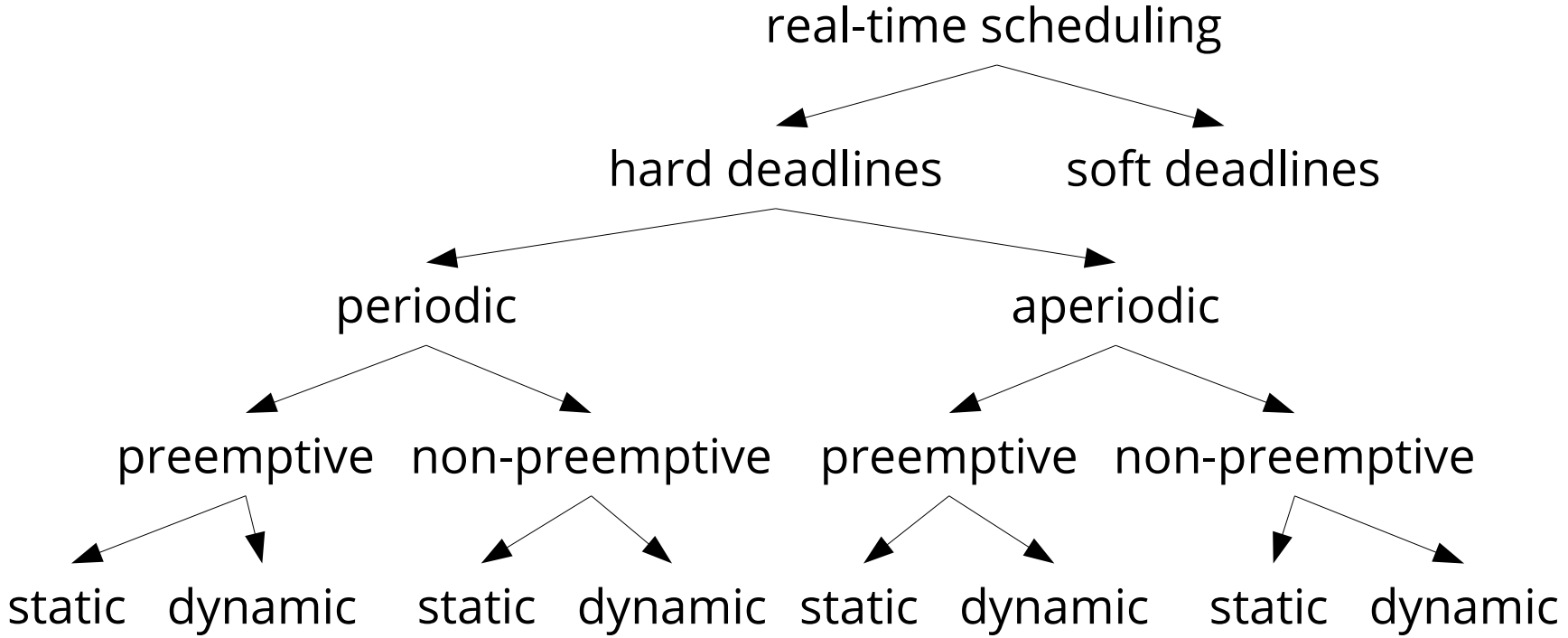
TimeCore

Inhalt

- Echtzeitsysteme
- Beispiel: OSEKtime
- **Echtzeit-Scheduling-Strategien**
- *Rate Monotonic Scheduling*
- *Earliest Deadline First Scheduling*
- Ausblick

Echtzeit-Scheduling

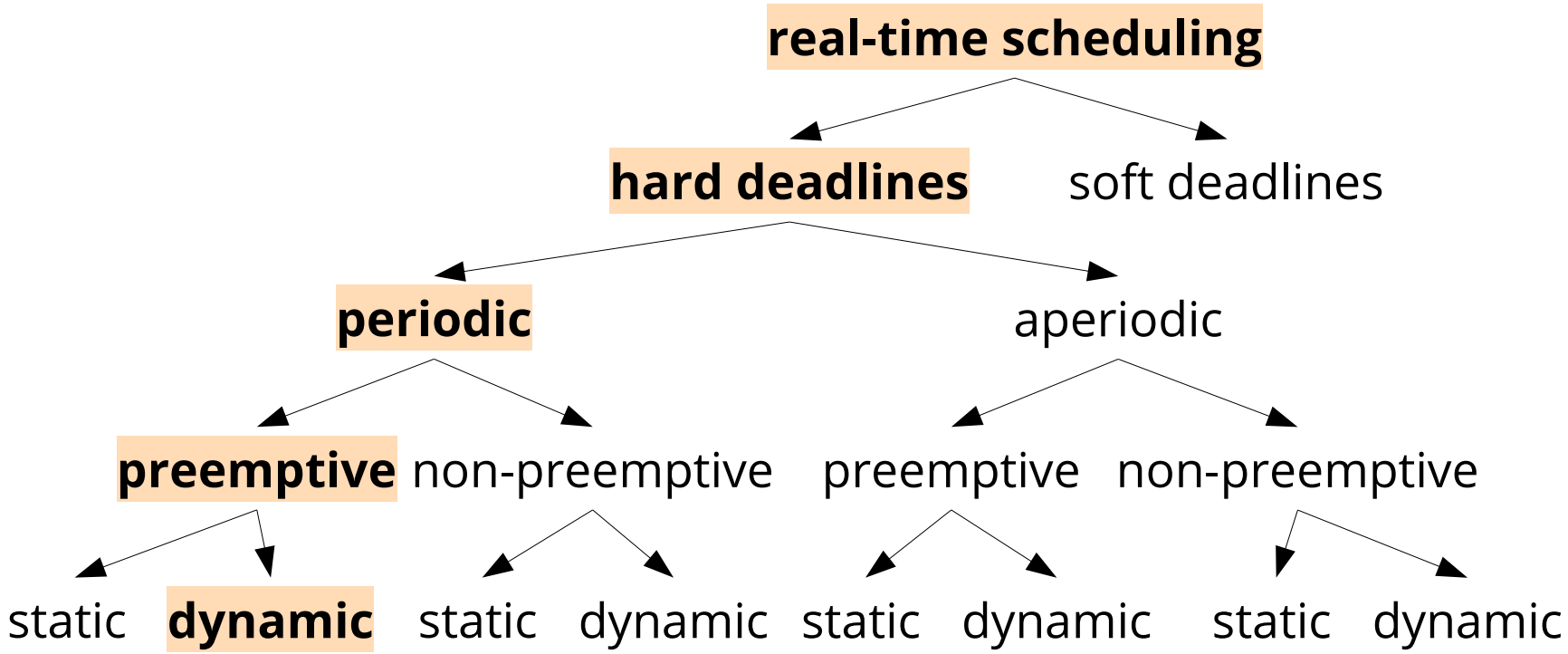
- Soll mathematische Garantien für die Einhaltung der harten Fristen geben.
- Taxonomie [3, S. 239]



Inhalt

- Echtzeitsysteme
- Beispiel: OSEKtime
- Echtzeit-Scheduling-Strategien
- ***Rate Monotonic Scheduling***
- *Earliest Deadline First Scheduling*
- Ausblick

Beispiel: *Rate-Monotonic Scheduling*



- ***Rate-Monotonic (RM)*** *Scheduling* ist eine *Scheduling*-Strategie für präemptive, periodische Tasks mit harten Fristen. Der Scheduler arbeitet zur Laufzeit (mit festen Prioritäten).

RM-Annahmen (Liu & Layland 1973 [4])

A1. Alle Tasks sind jederzeit **verdrängbar**.
Die Kosten der **Verdrängung** (Dauer) sind **vernachlässigbar**.

A2. Nur benötigte **Rechenkapazität** ist relevant.
Der Bedarf an Speicher, E/A und anderen Ressourcen ist vernachlässigbar.

A3. Alle Tasks sind **unabhängig**.
Es gibt keine Reihenfolgeabhängigkeiten.

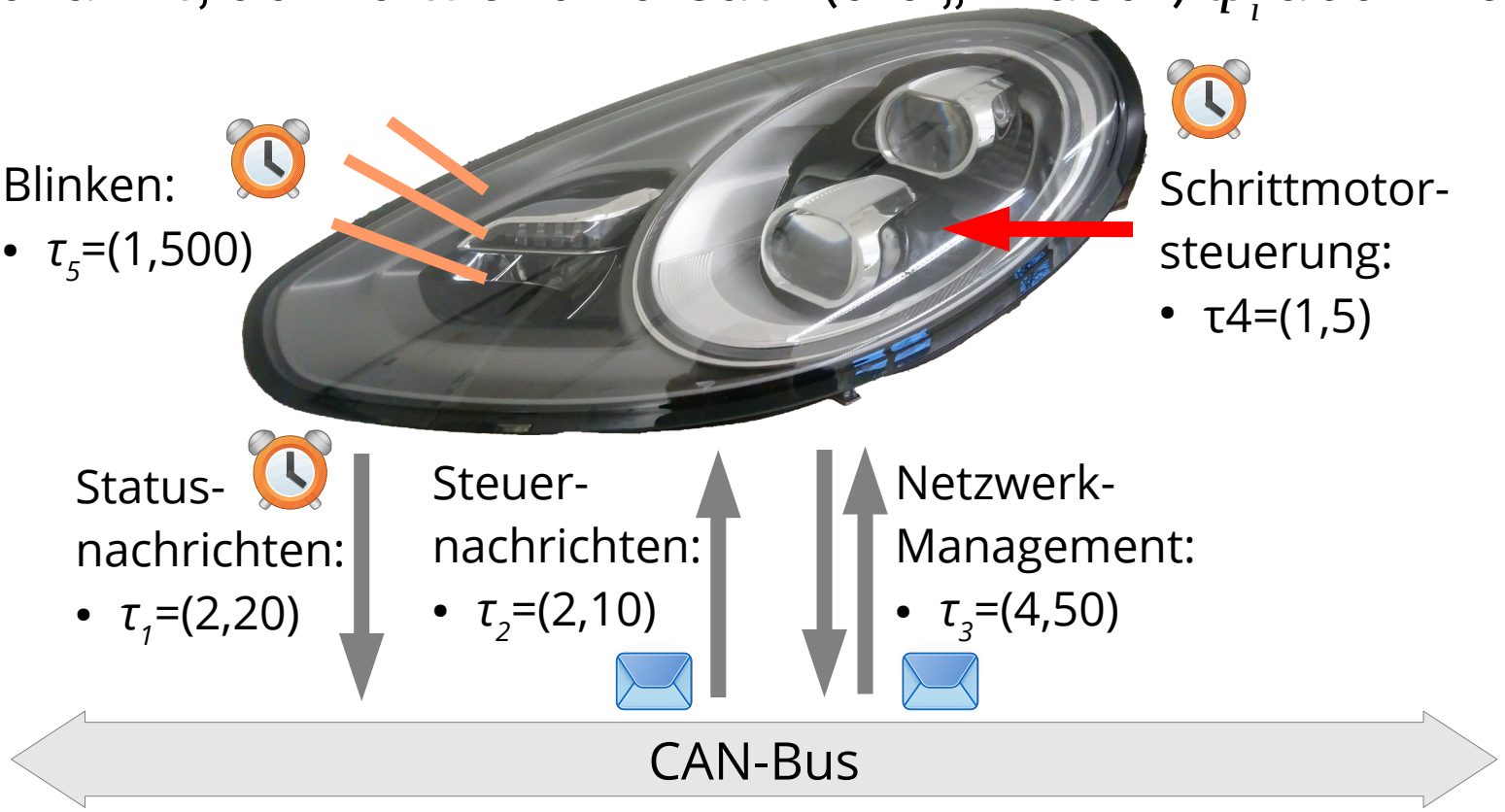
A4. **Alle** Tasks sind **periodisch**.

A5. Die **relative Frist** eines Tasks entspricht seiner **Periode**.

Beispiel: KFZ-Scheinwerfersteuergerät

... alles ist periodisch!

- Für jeden Task $\tau_i=(C_i,T_i)$ sind die WCET C_i und die Periode T_i bekannt, der zeitliche Versatz (die „Phase“) ϕ_i aber nicht.

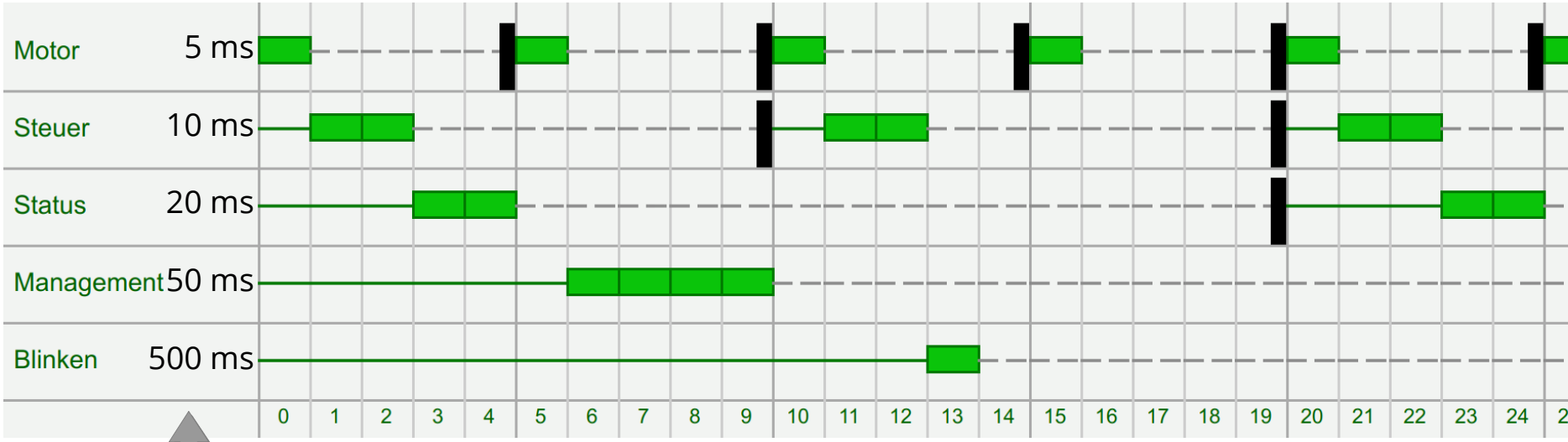


RM-Algorithmus

- Die Priorität wächst monoton mit der Ereignisrate (=Frequenz)
  - Also: Kleine Periode → hohe Priorität
- Hochpriore Tasks verdrängen niedrigpriore
- Beispiel:

Um RMS praktisch umzusetzen, benötigt man lediglich ein Betriebssystem mit einem präemptiven *"Fixed Priority"*-Scheduler.

Gantt-Diagramm für $\phi_i = 0$



Schedulability Analysis (Einplanbarkeitstest)

- Zu beantworten: Werden die Fristen aller Tasks eingehalten?
  - Den Ablaufplan kann man nur berechnen, falls die Tasks komplett zeitgesteuert arbeiten. In unserem Beispiel sind die Phasen beliebig.

- Gelten muss: Die Auslastung (*Utilization*) des Systems *U* ist kleiner oder gleich 1.

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

Annahme: Uniprozessor

U: Systemauslastung

m: Anzahl der Tasks

- Beispiel: τ_1=(1,5), τ_2=(2,20), τ_3=(2,10), τ_4=(4,50), τ_5=(1,500)

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

Soweit, so gut, aber reicht das?

Die „70%-Regel“ [4]

- Besagt: **Keine Fristverletzung**, wenn folgende Bedingung gilt:

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

U: Systemauslastung
m: Anzahl der Tasks

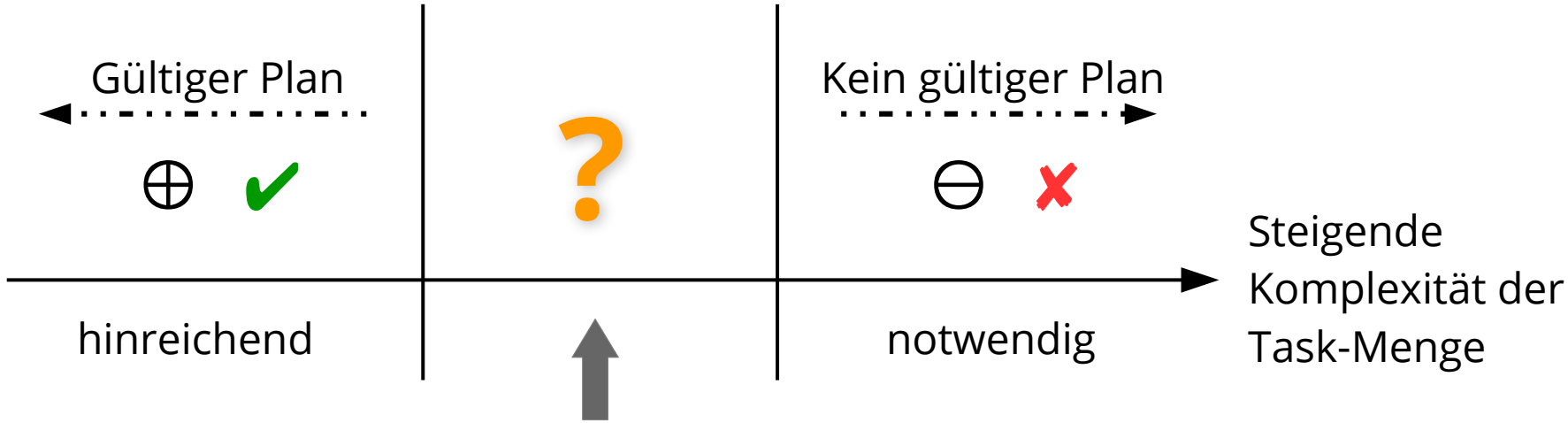
- Für große *m* konvergiert die Grenze gegen ln(2) ≈ 0,6931, also rund 70%.

Grenzwertberechnung: Regel von de L'Hospital

- **Vorteil:** Einfacher Test, schnell zu berechnen
  - Beispiel 1: *U*=58,2%, *m*=5
    - $m \cdot (2^{1/m}-1)$=74,35%, Bedingung erfüllt → keine Fristverletzung ✔
  - Beispiel 2: τ_1=(2,5) statt τ_1=(1,5), somit *U*=78,2%, *m*=5
    - $m \cdot (2^{1/m}-1)$=74,35%, Bedingung **nicht** erfüllt → **vielleicht** Fristverletzung
- **Nachteil:** Keine Aussage, falls die Bedingung nicht erfüllt ist

Hinreichende und notwendige Bedingungen

- **Hinreichende** Bed. **positiv**
  - z. B. $U \leq m \cdot \left(2^{1/m} - 1\right)$
  - → Ablaufplan ist **gültig**
- **Notwendige** Bed. **negativ**
  - z. B. $U \leq 1$ gilt nicht
  - → Ablaufplan ist **ungültig**



Bessere ***Schedulability*-Tests** erforderlich

Idealfall ist "exakter Test": Hinreichende *und* notwendige Bedingung

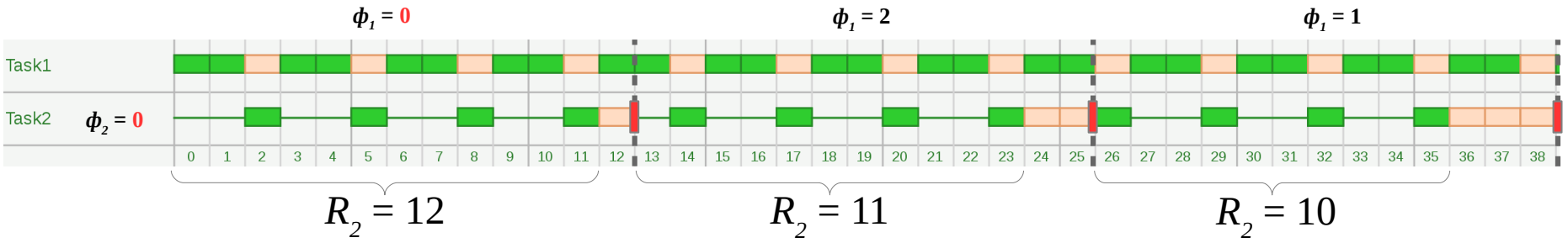
Exakter Test: Antwortzeitanalyse [5]

- Wenn die Antwortzeit R_i für alle Tasks kleiner oder gleich der Periode T_i ist, werden keine Fristen verletzt.

Bedingung (notw. *und* hinr.)

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

- Fall größtmöglicher Verzögerung $\phi_i = 0$: Bereits zu Beginn der Periode sind auch alle höherprioren Tasks rechenbereit.



- Berechnung von 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" – Verzögerung durch höherpriore Tasks

hp_x: Indizes der Tasks mit höherer Priorität als x

$\lceil x \rceil$: Ganzzahlige Aufrundung

Exakter Test: Iterative Lösung

- Berechnung von R_i durch Fixpunktiteration:
  - Abbruch, sobald $R_i^{n+1} = R_i^n$ oder $R_i^{n+1} > T_i$

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

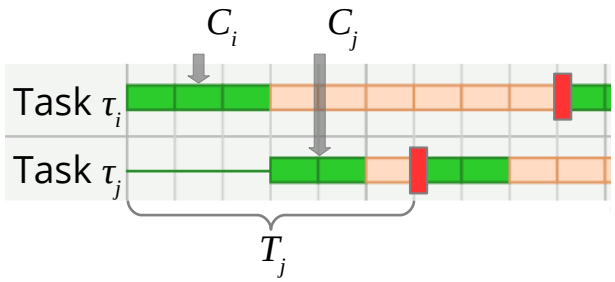
  - Pseudocode des Tests für alle Tasks:

```
for (each task τ_i) {
    I = 0
    do {
        R = I + C_i
        if (R > T_i) return false // Frist wird verletzt
        I = Σ_{j∈hp_i} ⌈R / T_j⌉ · C_j
    } while (I + C_i > R)
}
return true // alle Fristen werden eingehalten
```

Rate-Monotonic Scheduling ist „optimal“

- Zu zeigen: RM ist ein *optimaler* Scheduling-Algorithmus für *feste* Prioritäten. D.h., falls ein Algorithmus einen gültigen Ablaufplan liefert, tut RM dies auch.
- Widerspruchsbeweis: Angenommen … $\neg(A \Rightarrow B) \Leftrightarrow A \wedge \neg B$

  Algorithmus A liefert gültigen Plan und RM *nicht*.
  - Im Plan von A: prio(τ_i) = prio(τ_j) + 1 und $T_i > T_j$ (anders als RM)
    - $C_i + C_j \leq T_j$ gilt, da der Plan gültig ist und τ_i die höhere Priorität hat



Rate-Monotonic Scheduling ist „optimal“

- Zu zeigen:

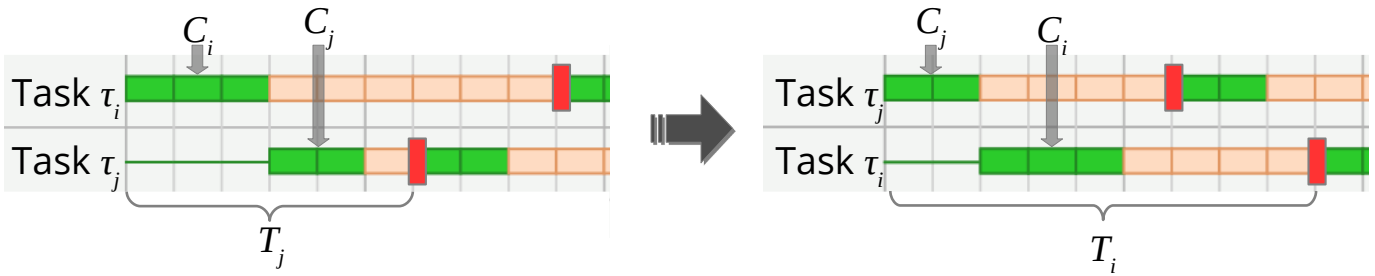
RM ist ein *optimaler* Scheduling-Algorithmus für *feste* Prioritäten. D.h., falls ein Algorithmus einen gültigen Ablaufplan liefert, tut RM dies auch.

- Widerspruchsbeweis: Angenommen …

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

Algorithmus A liefert gültigen Plan und RM *nicht*.

  - Im Plan von A: prio(τ_i) = prio(τ_j) + 1 und $T_i > T_j$ (anders als RM)
    - $C_i + C_j \leq T_j$ gilt, da der Plan gültig ist und τ_i die höhere Priorität hat
  - Wie wirkt sich die Vertauschung der Prioritäten (nur) dieser Tasks aus?
    - τ_j planbar, da jetzt mit höherer Priorität; τ_i auch planbar, da $C_i + C_j \leq T_j < T_i$



Rate-Monotonic Scheduling ist „optimal“

- Zu zeigen: RM ist ein *optimaler* Scheduling-Algorithmus für *feste* Prioritäten. D.h., falls ein Algorithmus einen gültigen Ablaufplan liefert, tut RM dies auch.
- Widerspruchsbeweis: Angenommen … $\neg(A \Rightarrow B) \Leftrightarrow A \wedge \neg B$

  Algorithmus A liefert gültigen Plan und RM *nicht*.
  - Im Plan von A: prio(τ_i) = prio(τ_j) + 1 und $T_i > T_j$ (anders als RM)
    - $C_i+C_j \leq T_j$ gilt, da der Plan gültig ist und τ_i die höhere Priorität hat
  - Wie wirkt sich die Vertauschung der Prioritäten (nur) dieser Tasks aus?
    - τ_j planbar, da jetzt mit höherer Priorität; τ_i auch planbar, da $C_i+C_j \leq T_j < T_i$
    - Durch eine endliche Zahl solcher Vertauschungen kommt man zu RMS
    - **Auch** ein gültiger Plan → Widerspruch zur Annahme → **RM ist optimal!**

RM-*Scheduling*: Fazit

- RM ist **einfach** anzuwenden und **optimal** bei festen Prioritäten
  - Betriebssystem benötigt lediglich "*Fixed Priority*"-Scheduler
- Antwortzeitanalyse ermöglicht exakten Einplanbarkeitstest
  - Wichtig für harte Echtzeitsysteme: Mathematische **Garantie**!
- In vielen Fällen reicht die 70%-Regel.

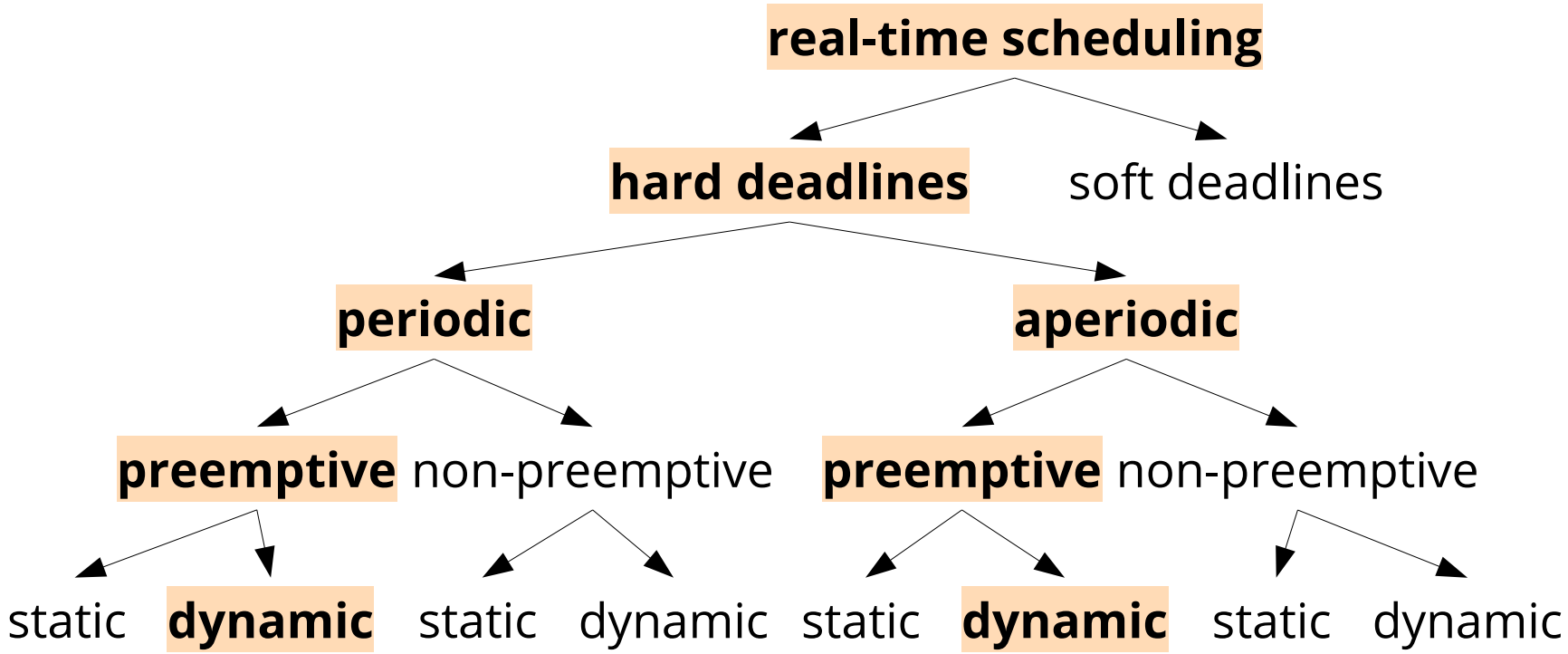
Aber Vorsicht:

- Annahmen A.1-5 müssen gelten!
  - Uni-Prozessor, keine Task-Abhängigkeiten, …
- WCET-Bestimmung bei modernen Prozessoren problematisch
  - Speicherhierarchien, *Out-of-Order-Execution*, DRAM-Zugriffszeiten, …
- Immer das *gesamte* System betrachten

Inhalt

- Echtzeitsysteme
- Beispiel: OSEKtime
- Echtzeit-Scheduling-Strategien
- *Rate Monotonic Scheduling*
- ***Earliest Deadline First Scheduling***
- Ausblick

Beispiel: *Earliest Deadline First*



- ***Earliest Deadline First* (EDF)** *Scheduling* ist eine Scheduling-Strategie für präemptive, periodische und aperiodische Tasks mit harten Fristen. Die Prioritäten werden dynamisch (zur Laufzeit) vergeben.

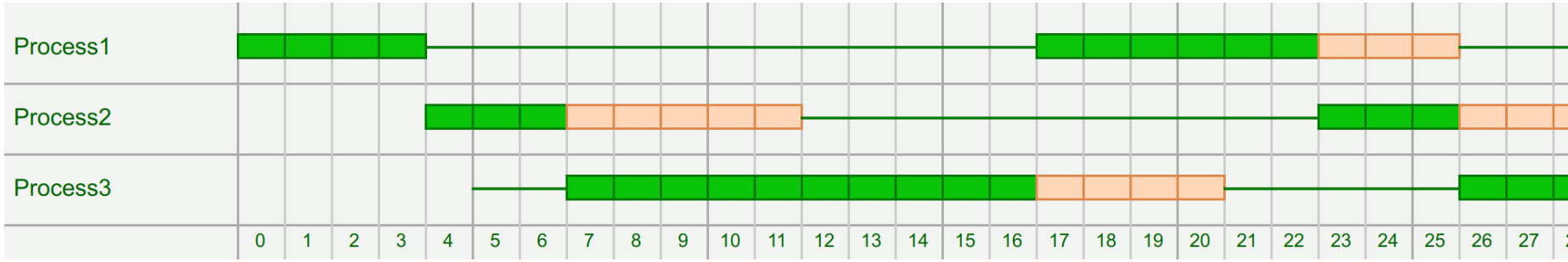
EDF-Algorithmus

- Rechenbereite Tasks sind in der Reihenfolge der **absoluten** Zeitpunkte ihrer Fristen sortiert.
- Falls der erste Task der Liste eine frühere Frist als der gerade laufende Task hat, wird dieser **sofort** verdrängt.

Spezifiziert werden die Deadlines aber i.d.R. relativ.

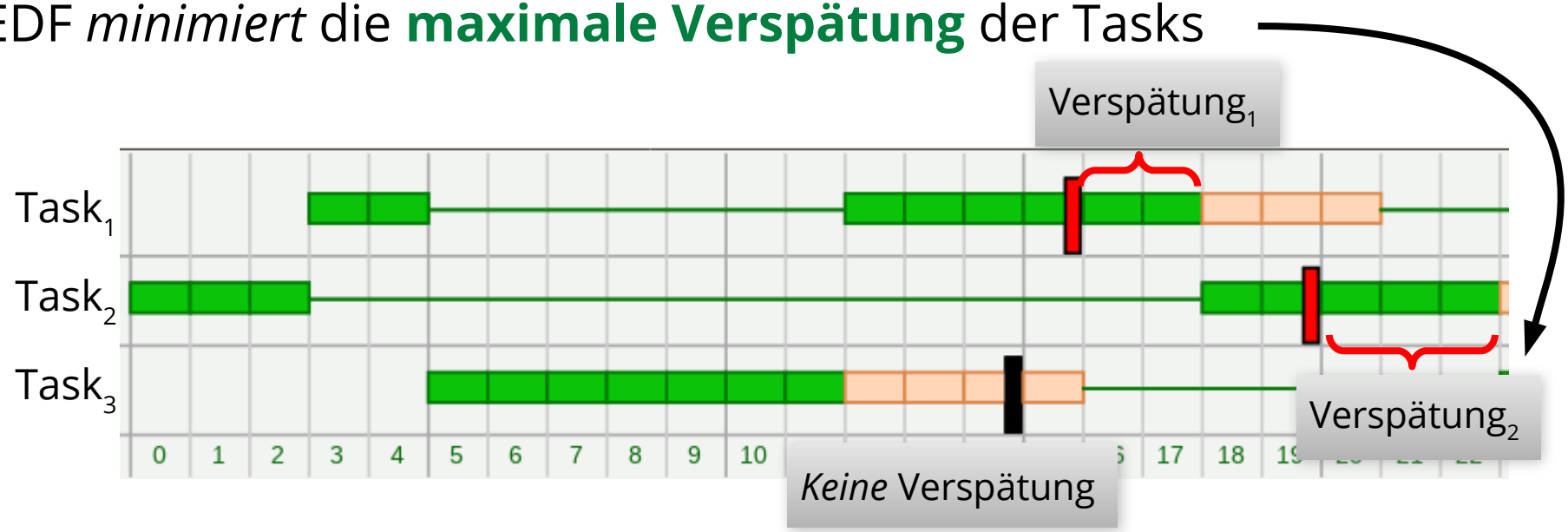
Beispiel:

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



Optimalität von EDF

- EDF *minimiert* die **maximale Verspätung** der Tasks



- Wenn ein Zeitplan existiert, der alle Fristen einhält, so tut dies auch EDF → **EDF ist optimal**
  - … für unabhängige Tasks mit dynamischen Prioritäten
- Speziell für **periodische** Tasks gilt: Wenn U ≤ 1, findet EDF einen gültigen Ausführungsplan (ohne *Deadlines* zu verpassen!).

Beweis in [6]

EDF-Scheduling: Fazit

- Schlicht **optimal** für periodische und aperiodische *Task-Sets*
  - Höhere Auslastung als RM-*Scheduling* durch dynamische Prioritäten

Aber:

- Meist nur in speziellen „Echtzeitbetriebssystemen“ implementiert
- Keine Aussage über Anzahl und Summe der Fristüberschreitungen
- Weniger vorhersagbar als zum Beispiel RM
  - Antwortzeiten können stark variieren: „Jitter“
  - In Überlastsituationen: „Dominoeffekt“

Inhalt

- Echtzeitsysteme
- Beispiel: OSEKtime
- Echtzeit-Scheduling-Strategien
- *Rate Monotonic Scheduling*
- *Earliest Deadline First Scheduling*
- **Ausblick**

Ausblick: Erweiterung der Strategien

- Umgang mit **sporadischen Tasks**
  - Limitierte Ankunftsrate, aber keine strikte Periode
- Beachtung von Task-Abhängigkeiten
- Steigerung der CPU-Auslastung
  - **Gemischtkritische („mixed-critical“) Systeme**
  - Einschränkung auf “**harmonische** Tasks”
    - Perioden sind ganzzahlige Vielfache voneinander
- **Moduswechsel**
  - z. B. Blinker/Schrittmotor wird aktiv
- [Vorübergehende] Überlast
- Anpassung an [heterogene] Multiprozessorsysteme

Literatur

[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. 2010. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems* (2nd ed.). Springer Publishing Company, Incorporated.

[4] C. L. Liu and J. W. Layland. 1973. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. J. ACM 20, 1 (January 1973), 46-61. http://dx.doi.org/10.1145/321738.321743

Literatur

[5] M. Joseph and P. Pandya. 1986. *Finding response times in real-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.