Echtzeitverarbeitung

R. Kaiser, K. Beckmann, R. Kröger

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Sommersemester 2022

6. Echtzeitplanung

https://school-time.co/

Inhalt

Modellbildung

Hochschule RheinMain

- Ausgangspunkt: Modellierung eines realen Anwendungssystems
  - Zuordnung von Rechensystem-Prozessen zu technischen Prozessen
  - Eingangsgrößen und Zielgrößen
  - Informationsfluss zwischen Prozessen
  - Zeitbedingungen
  - Betriebsmittelbedarf

Prozesse in der Echtzeitplanung

- Prozess: Ausführung eines sequenziellen Programms auf einem Prozessor (s.o.)
- Ausführung **endet** nach endlich vielen Schritten

→ Prozess entspricht endlicher Ausführungsfolge von Maschinenbefehlen

- Aufgabe der *Echtzeitplanung* (engl. *real-time scheduling*):Den Prozessen den Prozessor so zuzuteilen, dass alle von der Anwendung herrührenden Zeitbedingungen eingehalten werden
- Dabei ist der Prozess die kleinste einplanbare Einheit

Unterbrechbarkeit und Periodizität

- Bezogen auf den Startzeitpunkt spricht man von einem:
  - *periodischen* Prozess, falls er jeweils nach Ablauf einer festen Zeitspanne (der *Periode*) erneut gestartet werden soll,
  - *aperiodischen* oder *sporadischen* Prozess, sonst.
- Abhängig von der Anwendung ist zu unterscheiden zwischen Prozessen,
  - deren Ausführung zwischen Start und Beendigung nicht unterbrochen werden darf (nicht-unterbrechbar, engl. *non-preemptive*),
  - deren Ausführung i.d.R. nach jeder Anweisung unterbrochen werden kann (unterbrechbar, engl. *preemptive*)

Mehrfachinstanziierung

- Prozess als „Typ”
- Prozessinstanzen als konkrete Prozesse, die mit privaten Eingangsvariablen und Zielvariablen ausgestattet sind

Prozessumschaltung

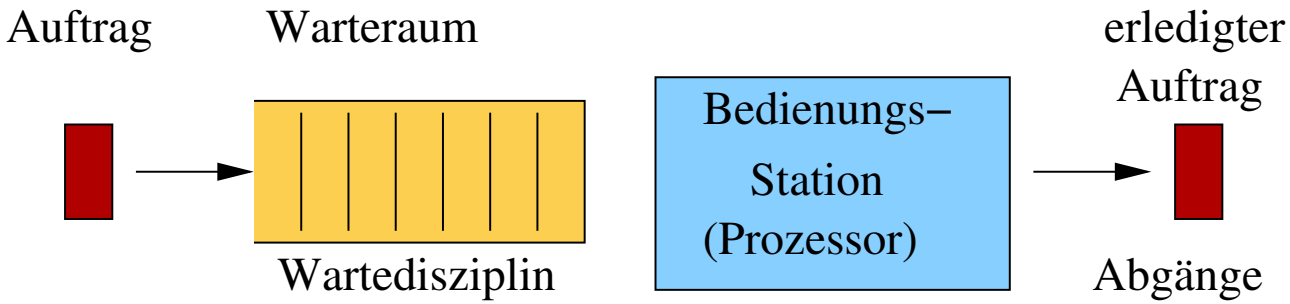
- Prozessumschaltung (engl. *context switch*): Änderung in der Zuordnung des gerade ausgeführten Prozesses
- Start eines Prozesses und Prozessumschaltung können vom Rechensystem oder vom technischen System ausgehen
- Dauer der Prozessumschaltung wird in den Planungsverfahren i.d.R. nicht berücksichtigt oder als Konstante den Ausführungszeiten zugeschlagen

Dauer der Prozessausführung

- abhängig von:
  - Leistungsfähigkeit des Prozessors
  - Peripherie des Prozessors
  - Eingangsdaten
  - Verfügbarkeit der Betriebsmittel
  - Verzögerung durch Erledigung wichtigerer Aufgaben (z.B. Interrupt-Bearbeitung)

Begriffe aus der Bedientheorie (1)

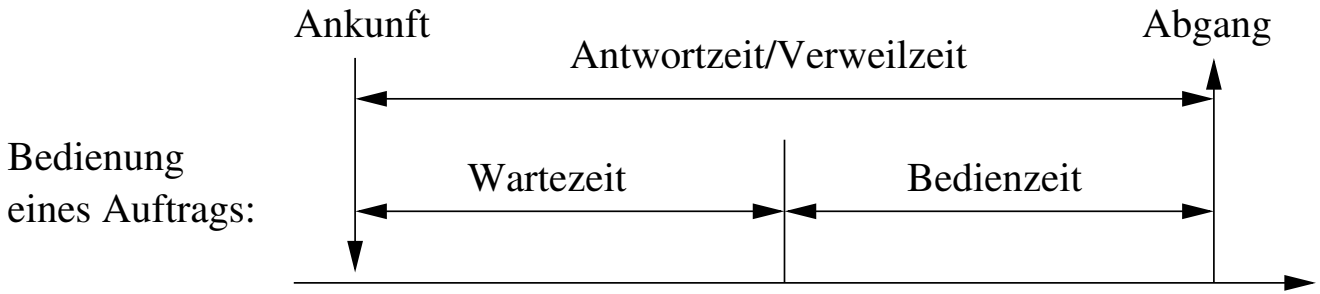
Hochschule RheinMain



- **Auftrag:** Einheit zur Bearbeitung (hier:Prozessinstanz)
- **Bedienzeit**:Zeitdauer für die reine Bearbeitung eines Auftrags durch die Bedienstation (hier: den Prozessor)
- **Wartedisziplin**: z.B. (einfache):
  - FCFS (*First-Come-First-Served*) oder FIFO (*First-In-First-Out*)
  - LIFO (*Last-In-First-Out*)
  - Random (zufällige Auswahl)
- Ankünfte und Bedienungszeiten werden bei Bedienungsmodellen häufig durch stochastische Prozesse modelliert
- komplexere Bedienmodelle können mehrere Bedienstationen (Multiprozessorsystem), mehrere Warteräume, mehrphasige Bedienung sowie die Rückführung teilweise bearbeiteter Aufträge enthalten.

Begriffe aus der Bedientheorie (2)

Hochschule RheinMain



- **Antwortzeit:** Zeitdauer vom Eintreffen eines Auftrags bis zur Fertigstellung
  Bei Dialogaufträgen Zeitdauer von der Eingabe eines Benutzers (z.B. Drücken der Return-Taste) bis zur Erzeugung einer zugehörigen Ausgabe (z.B. auf dem Bildschirm)
  Auch **Verweilzeit** genannt
- **Wartezeit**: Antwortzeit - Bedienzeit
- **Durchsatz**: Anzahl erledigter Aufträge pro Zeiteinheit
- **Auslastung**: Anteil der Zeit im Zustand „belegt”
- **Fairness**: „Gerechte” Behandlung aller Aufträge, z.B. alle rechenwilligen Prozesse haben gleichen Anteil an der zur Verfügung stehenden Rechenzeit

Relevante Größen

- Für eine Prozessausführung P_i *sporadischer* Prozesse sind folgende Zeitpunkte bzw. Zeitspannen relevant:
  - **Bereitzeit** (*ready time*) r_i: frühester Zeitpunkt, zu dem der Prozessor an P_i zugeteilt werden darf.
  - **Ausführungszeit** / **Bedienzeit** (*execution time*) Δe_i: reine Rechenzeit auf dem Prozessor; zur Planung wird meistens die für alle möglichen Eingangsdaten längste Zeitdauer zugrunde gelegt (schwierig zu bestimmen !!) (WCET = *Worst Case Execution Time*)
  - **Frist** (*deadline*) d_i: zu diesem Zeitpunkt muss die Ausführung von P_i abgeschlossen sein.
  - **Startzeit** (*starting time*) s_i: Beginn der Ausführung von P_i
  - **Abschlusszeit** / **Abgangszeit** (*completion time*) c_i: Beendigung der Ausführung von P_i

Zur Veranschaulichung

Hochschule RheinMain

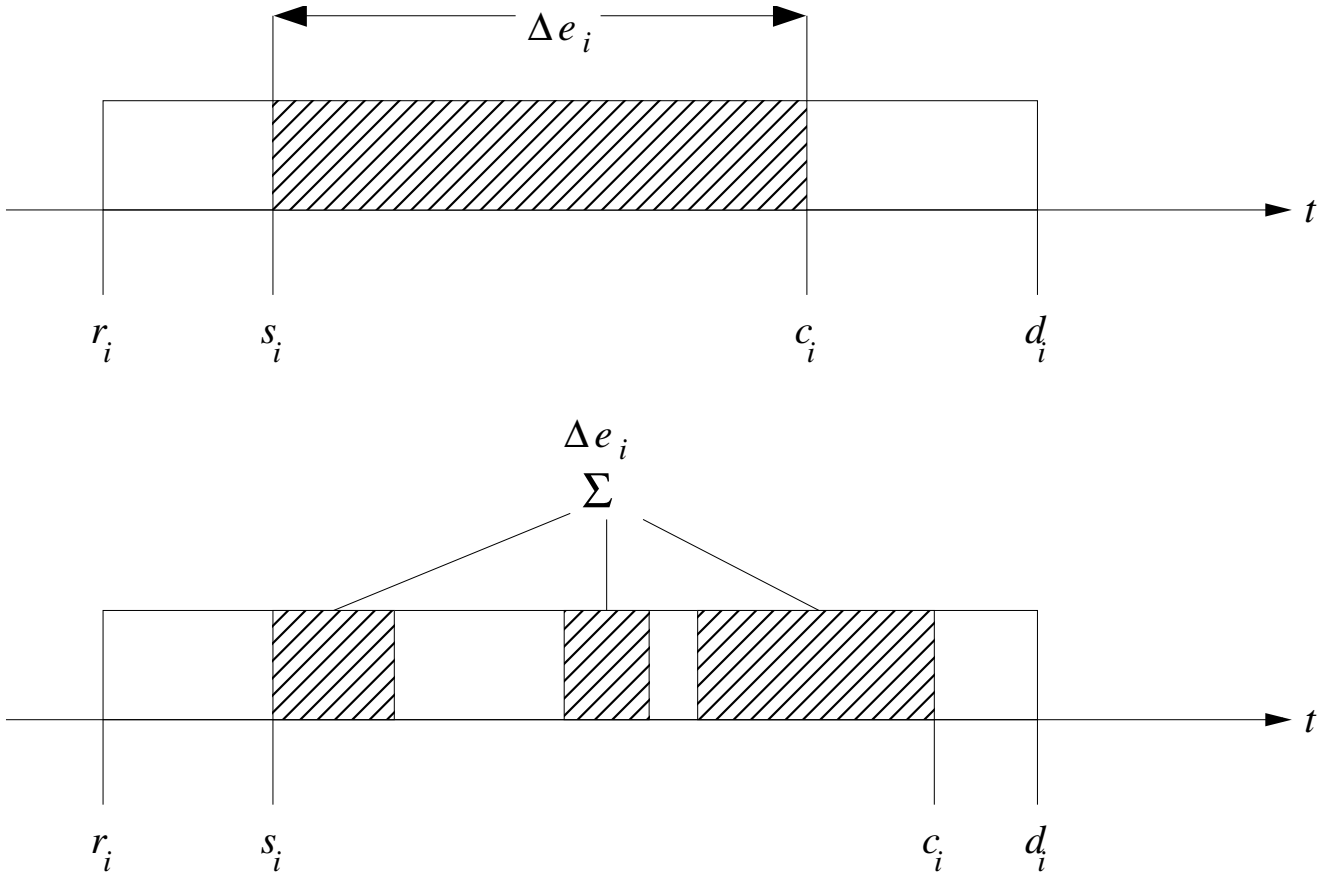


Abbildung: Zeitdiagramm einer ununterbrochenen (oben) und einer unterbrochenen (unten) Ausführung eines Prozesses P_i

Relevante Größen bei periodischen Prozessen

- Ein periodischer Prozess P_i wird mit einer bestimmten Frequenz f_i regelmäßig bereit gesetzt.
- Die Periode $\Delta p_i = \frac{1}{f_i}$ definiert den Rahmen seiner j-ten Ausführung P_i^j
- Ausgehend von der ersten Bereitzeit r_i^1 sind alle folgenden Bereitzeiten festgelegt durch

$$r_i^j = (j-1) \cdot \Delta p_i + r_i^1 \; \forall \; j \geq 1$$

- Das Ende einer Periode ist gleichzeitig eine Frist für die Prozessausführung:

$$d_i^j = j \cdot \Delta p_i + r_i^1 = r_i^{j+1} \; \forall \; j \geq 1$$

Zur Veranschaulichung

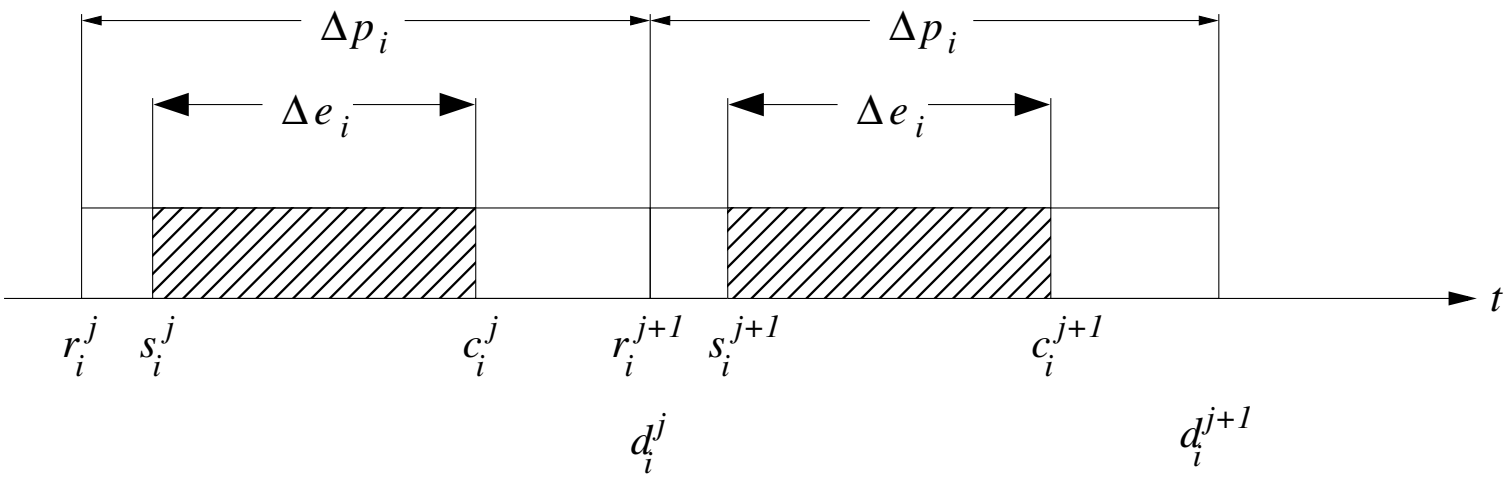


Abbildung: j-te und $(j+1)$-te ununterbrochene Ausführung von P_i^j und P_i^{j+1}.

Aufgaben der Echtzeitplanung

- Die Anwendung gibt Bereitzeiten, Ausführungszeiten, Fristen und Perioden vor
- Die Echtzeitplanung legt Start- und Abschlusszeiten und bei unterbrechbaren Prozessen die Folge von Unterbrechungen fest
- Im folgenden wird ausschließlich Prozessorzeit als Betriebsmittel betrachtet. Ein analoges Vorgehen ist auch für Speicher- und Kommunikations-Betriebsmittel (z.B. Busbelegung oder Netzwerkbelegung) möglich

Varianten der Echtzeitplanung

- *Statisches Scheduling*: Alle Daten für die Planung sind vorab bekannt, die Planung erfolgt durch eine *Offline-Analyse*
- *Dynamisches Scheduling*: Daten für die Planung fallen zur Laufzeit an und müssen zur Laufzeit verarbeitet werden
- *Explizite Planung*: Dem Rechensystem wird ein vollständiger Ausführungsplan (*Schedule*) übergeben, der zur Laufzeit befolgt wird (Umfang kann extrem groß werden)
- *Implizite Planung*: Dem Rechensystem werden nur die Planungsregeln übergeben

Phasen der Echtzeitplanung

Hochschule RheinMain

1. Einplanbarkeitsanalyse (*schedulability analysis*, S-Test)
2. Planerstellung (*schedule construction*)
3. Prozessorzuteilung (*dispatching*)

Phase 1: Einplanbarkeitsanalyse

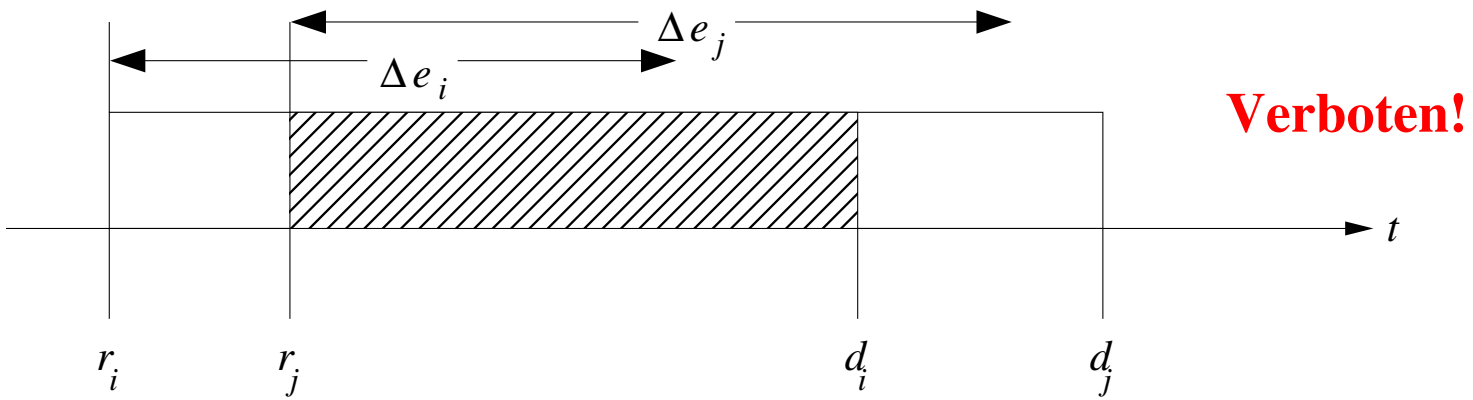
Hochschule RheinMain

- Unter Berücksichtigung der Planungsgrößen wird geprüft, ob überhaupt ein brauchbarer Ausführungsplan existiert
- Prozess P_i charakterisiert durch $(r_i, \Delta e_i, f_i$ bzw. $\Delta p_i, d_i)$ mit f_i als Frequenz bei periodischen Prozessen, bzw. als obere Schranke bei sporadischen Prozessen
- Notwendige Bedingungen:
  - $\forall i : \Delta e_i < d_i - r_i < \frac{1}{f_i}$
  - $\sum f_i \Delta e_i \leq$ verfügbare Ressourcen

Brauchbarkeit eines Plans

Hochschule RheinMain

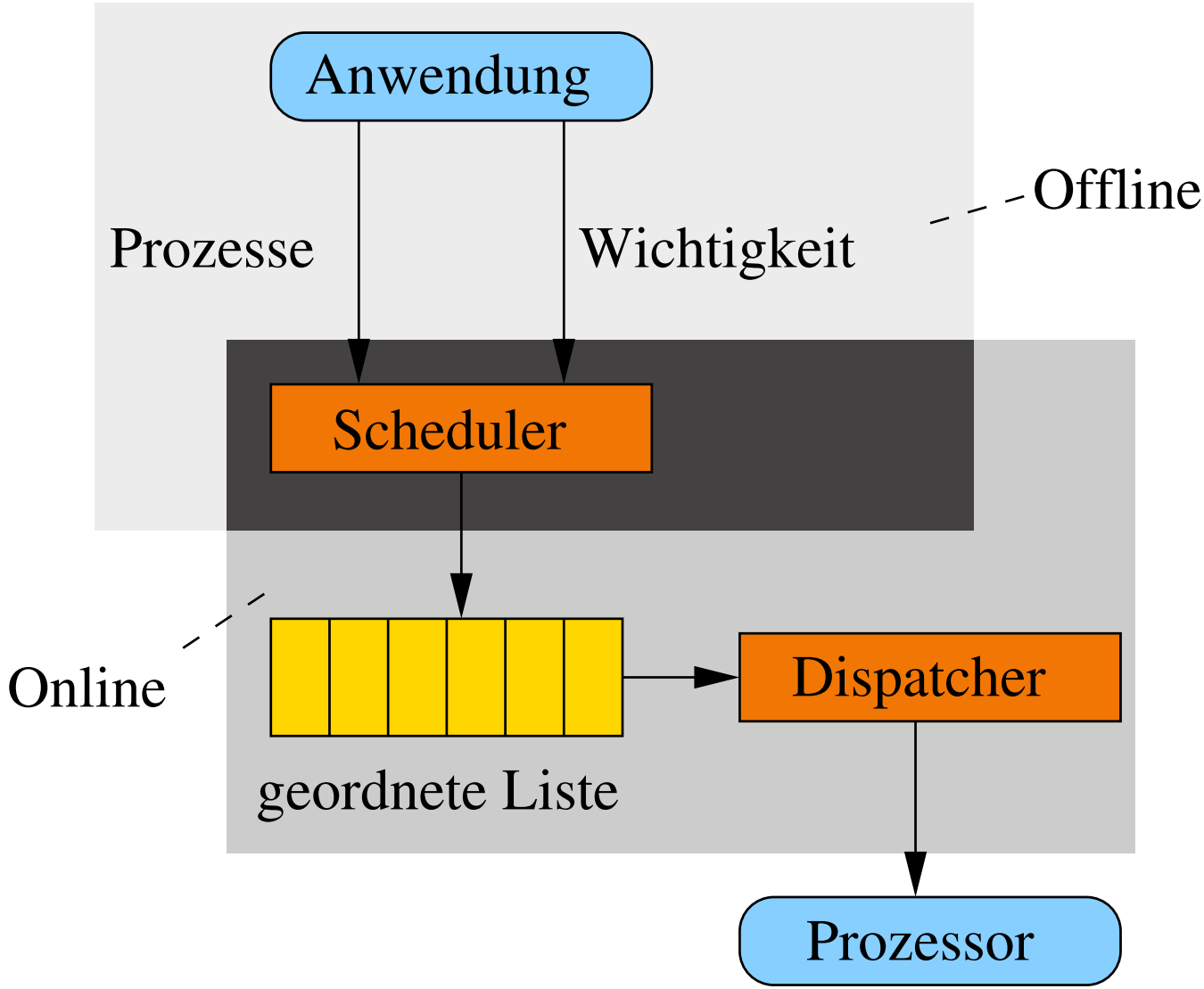
- Ein Plan (*Schedule*) heißt **brauchbar** (*feasible*) für eine Prozessmenge $\{P_1, P_2, ..., P_n\}$, falls bei gegebenen $(r_i, \Delta e_i, f_i, d_i)$ die Startzeit s_i und die Abschlusszeit c_i so gewählt sind, dass
  - alle Zeitbedingungen für jeden einzelnen Prozess eingehalten werden: $\forall i : \Delta e_i < d_i - r_i < \frac{1}{f_i}$
  - genügend Ressourcen vorhanden sind: $\sum f_i \Delta e_i \leq$ verfügbare Prozessorzeit
  - keine überlappenden Ausführungszeiten entstehen: $\forall i, j \; d_i < d_j : d_i \leq d_j - \Delta e_j$



Phase 2: Planerstellung

- Der Plan umfasst alle Informationen, die zur Laufzeit notwendig sind, um die eingeplanten Prozesse einem Prozessor zuzuordnen
- Eigenschaften statischer bzw. dynamischer Planung
- statisch:
  - \+ statische, nicht-unterbrechbare Prozesse
  - \+ Offline Einplanbarkeitsanalyse
  - \+ Einplanbarkeit bewiesen
  - \+ Plan z.B. in Form einer Prozesstabelle mit nach Startzeiten geordneten Einträgen
  - \+ Minimaler Aufwand zur Laufzeit
  - \- Bestimmung WCET
  - \- Komplexität des Verfahrens
- dynamisch:
  - \+ Mix aus periodischen und sporadischen Prozessen
  - \+ Prozesse unterbrechbar
  - \+ harte und weiche Zeitbedingungen
  - \+ Plan liefert Wichtigkeit und Prioritäten
  - \+ Online Analyse und Planerstellung
  - \- Aufwand zur Laufzeit
  - \- Einplanen des Zusatzaufwandes

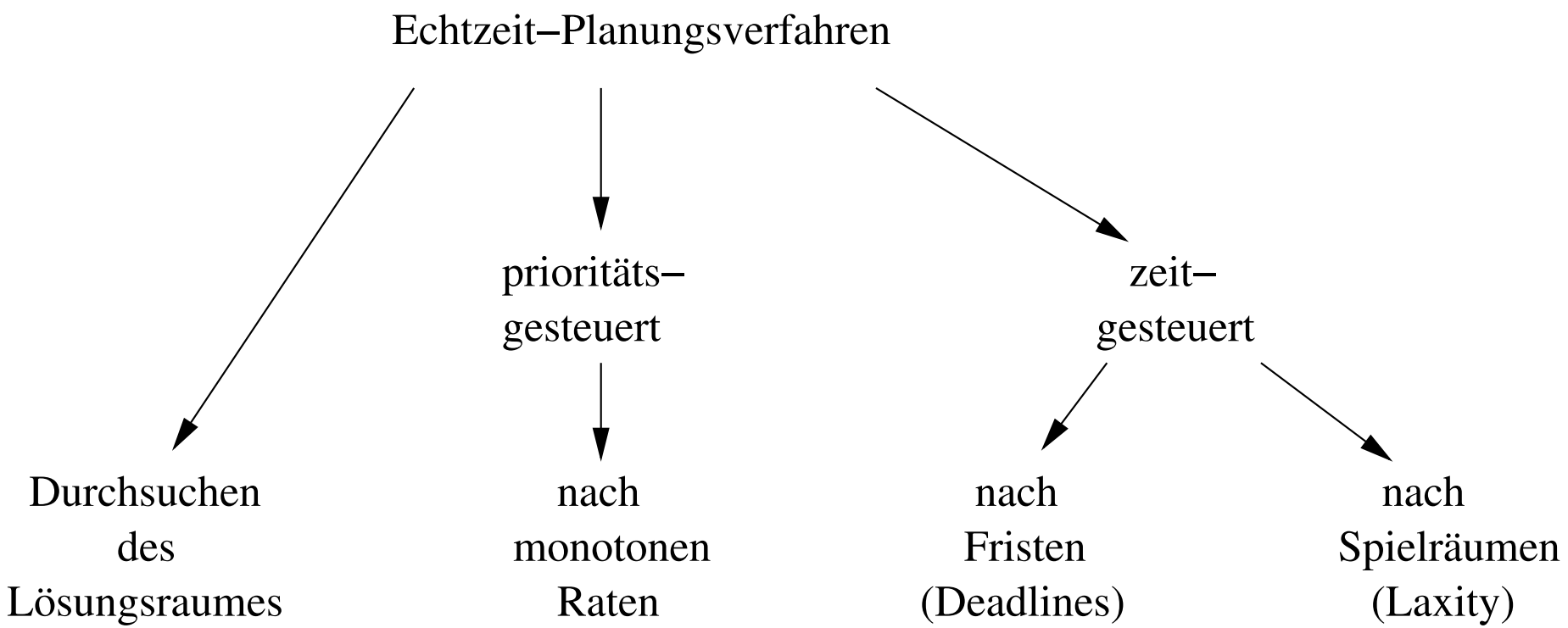
Phase 3: Prozessorzuteilung (Dispatching)



Gütekriterien für Planungsverfahren

- Ein *statisches* Planungsverfahren heißt *optimal*, falls es zu jeder beliebigen Prozessmenge einen Plan findet, sofern ein solcher existiert
- Ein *dynamisches* Planungsverfahren heißt *optimal*, falls es zu jeder beliebigen Prozessmenge einen brauchbaren Plan liefert, wenn ein statisches Verfahren in Kenntnis der gesamten Eingabedaten auch einen brauchbaren Plan geliefert hätte

Klassifizierung der Echtzeit-Planungsverfahren



Planen durch Suchen

Hochschule RheinMain

- Durchsuchen des Lösungsraumes ohne Beachtung von Strategien oder Heuristiken stellt das „einfachste" Planungsverfahren dar (*Branch-and-Bound*-Verfahren)

(A) Es werden zunächst alle möglichen Kombinationen von nacheinander ausgeführten Prozessen ohne Berücksichtigung von Bereitzeiten und Fristen generiert

→ Für n Prozesse existieren $n!$ Möglichkeiten
Darstellung als Baum der Tiefe n

→ Aufwand des Planungsverfahrens: $O(n!)$

Einbeziehen von Bereitzeiten und Fristen

- Nicht alle nach (A) erzeugten Pläne sind brauchbar, wenn Bereitzeiten und Fristen berücksichtigt werden

(B) Heuristik: Einplanen eines Prozesses so früh wie möglich, d.h.

$$s - r \rightarrow min$$

  - Äste des Lösungsbaumes werden nicht weiter geführt, wenn sich eine Fristüberschreitung ergeben würde
  - **Satz**: Wenn es brauchbare Pläne für eine Prozessmenge gibt, so wird ein solcher durch (B) gefunden

Beispiel

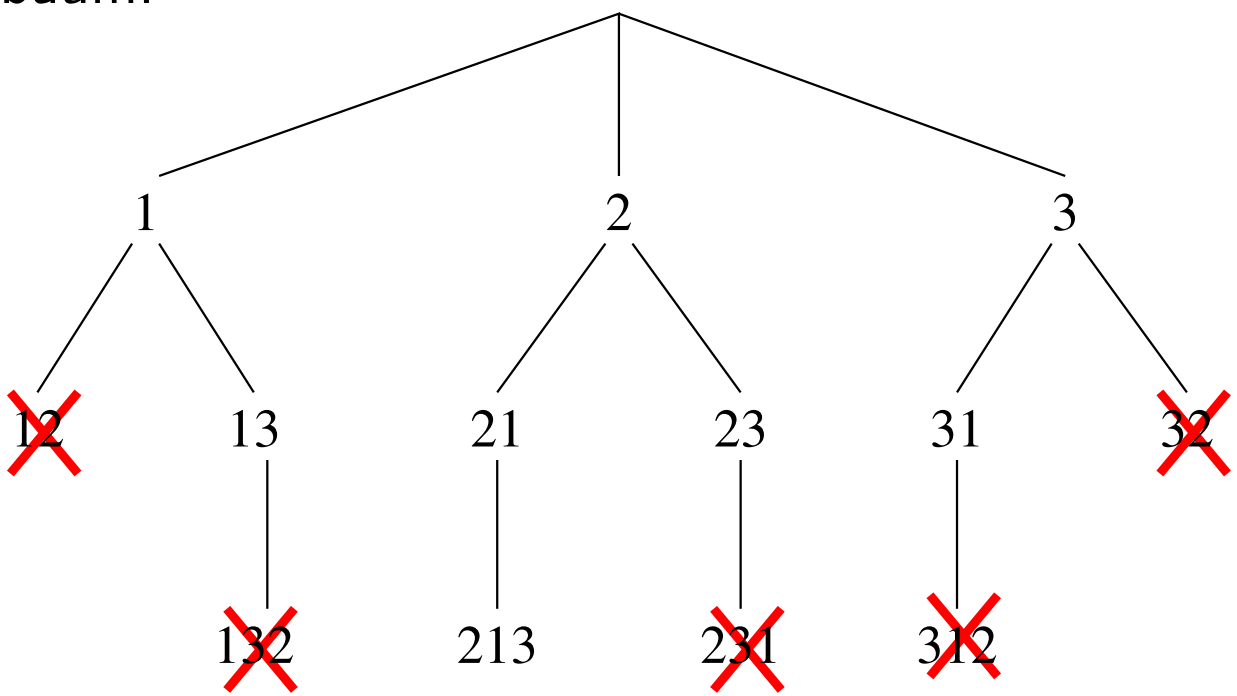
Hochschule RheinMain

Gegeben: Prozessmenge mit 3 Prozessen

Prozess i	Bedienzeit Δe_i	Frist d_i
1	13	19
2	3	8
3	6	24

Alle Aufträge seien zur Zeit Null bekannt

- Lösungsbaum:



Planen nach Fristen

Hochschule RheinMain

- Strategie: *Earliest Deadline First* (EDF)
  - Der Prozessor wird demjenigen Prozess i zugeteilt, dessen Frist d_i den kleinsten Wert hat (am nächsten ist)
  - Wenn es keinen rechenbereiten Prozess gibt, bleibt der Prozessor untätig (d.h. „idle")
- EDF ist eine der verbreitetsten zeitbasierten Strategien
- EDF ist anwendbar auf nicht-unterbrechbare und unterbrechbare Prozesse
- EDF ist anwendbar in statischen und dynamischen Planungsverfahren
- EDF ist bei nicht-unterbrechbaren Prozessen optimal, wenn alle Bereitzeiten gleich sind
- Falls EDF in diesem Fall keinen brauchbaren Plan liefert, gibt es keinen

Beispiel (EDF nicht-unterbrechbar)

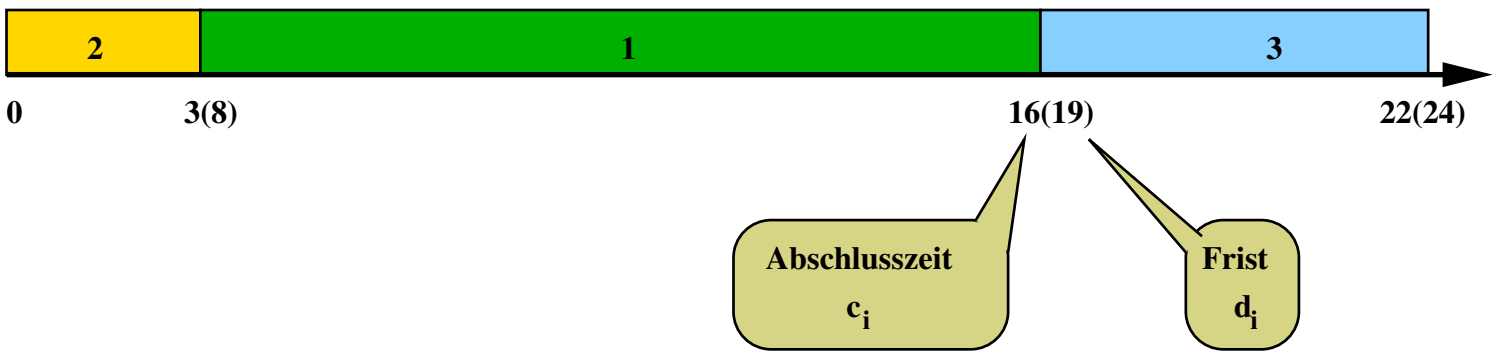
Hochschule RheinMain

Gegeben: Prozessmenge mit 3 Prozessen

Prozess i	Bedienzeit Δe_i	Frist d_i
1	13	19
2	3	8
3	6	24

Alle Aufträge seien zur Zeit Null bekannt

- Aus EDF resultierender Schedule:



EDF bei unterbrechbaren Prozessen

- Motivation: nicht alle Bereitzeiten sind identisch
- Ansatz: Zerlegen der Ausführungszeit eines Prozesses in mehrere Teilintervalle
- Variante EDF_p:
  - $Ready(t)$ sei zum Zeitpunkt t die nach Fristen geordnete Liste aller noch nicht fertig abgearbeiteten Prozesse
  - Bestimme zum Zeitpunkt t den Prozess, der die nächste Deadline besitzt
  - Ordne den Prozessor zu für das Minimum aus
    - Restbedienzeit des Prozesses
    - kleinster Zeitpunkt, zu dem einer der bisher nicht betrachteten Prozesse bereit wird
  - Plane zu diesem Zeitpunkt neu
- EDF ist auch für unterbrechbare Prozesse optimal

Beispiel (EDF unterbrechbar)

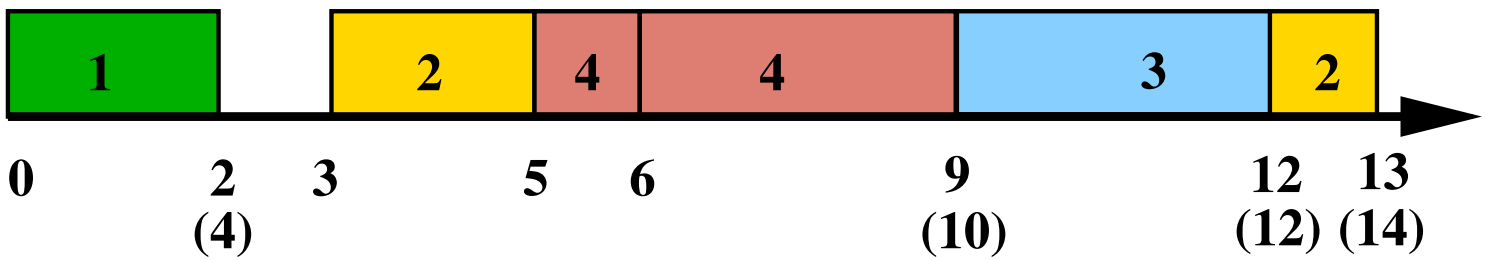
Hochschule RheinMain

Gegeben: Prozessmenge mit 4 Prozessen

Prozess i	Bedienzeit Δe_i	Bereitzeit r_i	Frist d_i
1	2	0	4
2	3	3	14
3	3	6	12
4	4	5	10

Alle Aufträge seien zur Zeit Null bekannt

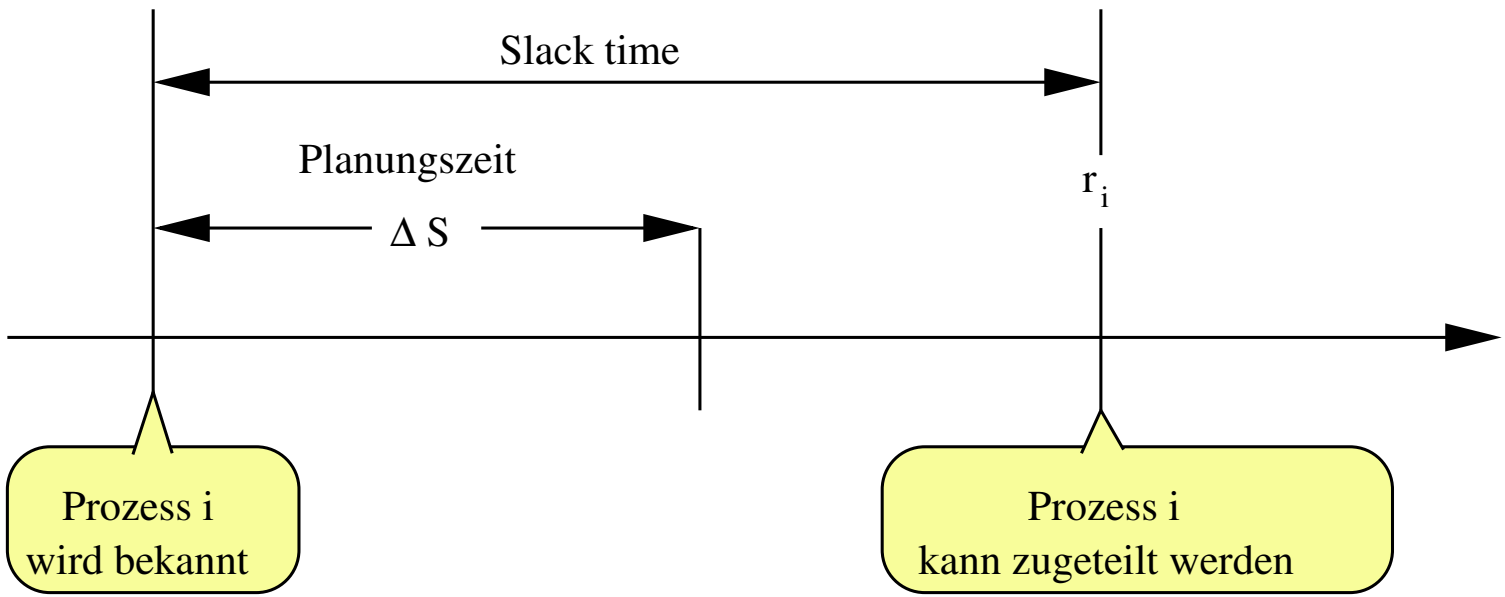
- Aus EDF_p resultierender Schedule:



EDF bei dynamischem Scheduling(1)

Hochschule RheinMain

- Voraussetzung:
  - Alle Prozessgrößen (Bereitzeit, Ausführungszeit, Frist) werden hinreichend früh bekannt, so dass der Prozess noch eingeplant werden kann
- Oft wird unterstellt, dass der Zeitaufwand für die Planung und Prozessorzuteilung vernachlässigt werden kann
- Bei Berücksichtigung der Planungszeit:



EDF bei dynamischem Scheduling(2)

- Vorgehensweise ansonsten wie bei EDF_p
- Dynamische EDF-Planung ist ebenfalls optimal
- Dynamische EDF-Planung ist sogar dann optimal, wenn die Ausführungszeiten nicht bekannt sind, d.h. wenn es überhaupt einen brauchbaren Plan gibt, wird auch einer durch EDF gefunden

EDF bei periodischen Prozessen

- Voraussetzung:
  - Perioden aller Prozesse bekannt
- Bei periodischen Prozessen $P = \{P_1, P_2, ..., P_n\}$ bezeichnet die Auslastung U (Utilisation) die Summe der Zeitanteile, die die Ausführungszeit an der Periode des jeweiligen Prozesses hat:

$$U = \sum_{i \in P} \frac{\Delta e_i}{\Delta p_i} = \sum_{i \in P} \Delta e_i \cdot f_i$$

- Einplanbarkeit ist nur gegeben, wenn $0 \leq U \leq 1$
- Es gilt auch: Wenn $U \leq 1$, wird durch EDF *immer* ein brauchbarer Plan für periodische Prozesse gefunden

Zusammenfassung der Vorteile von EDF[1] (1)

- EDF ist anwendungsorientiert
- Erweiterungen, Modifikationen und Berücksichtigung zusätzlicher Prozesse möglich
- Einfacher Einplanbarkeitstest
- Antwortzeiten können dynamisch garantiert werden
- Überlast kann frühzeitig erkannt werden
- EDF-Planungsalgorithmus hat lineare Komplexität
- EDF-Planungsalgorithmus hat beschränkte Ausführungszeit
- EDF-Planungsalgorithmus ist leicht implementierbar
- EDF-Planungsalgorithmus erlaubt Behandlung periodischer, sporadischer und untereinander in Vorrang-Relation stehender Prozesse durch gemeinsames Verfahren
- EDF erreicht beste Prozessorauslastung unter Beachtung der Planbarkeit

[1] nach W. Halang

Vorteile von EDF (2)

- EDF ist i.w. „non-preemptive", d.h. Unterbrechung der Bedienung ist nur notwendig, wenn sich die Situation ändert (z.B. neuer Prozess kommt, blockierter Prozess wird rechenwillig, ...)
- Wenn ein neuer Prozess hinzukommt, bleibt die Reihenfolge unter den anderen unverändert
- EDF hat i.W. eine sequentielle Ausführung der Prozesse zur Folge
- EDF vermeidet Ressourcen-Konflikte und Deadlocks
- EDF erlaubt die Planung unterbrechbarer und nicht-unterbrechbarer Prozesse

Probleme bei EDF

- Um EDF auf Basis prioritätsbasierter Scheduler zu implementieren, muss das Betriebssystem das dynamische Ändern der Prioriätszuordnung ermöglichen (daher auch *dynamic priority scheduling*)
  Einige Betriebssysteme (z.B. OSEK) unterstützen dies nicht
- Überlastsituationen ($U > 1$) führen unweigerlich zum Verpassen von Fristen. Alle Tasks („kritische“ wie „unkritische“) sind davon gleichermaßen betroffen.
  Keine „graceful degradation“

Planen nach Spielräumen

- Das Planen nach Fristen ist für Mehrprozessorsysteme bei so früh wie möglicher Vergabe an rechenwillige nicht unterbrechbare Prozesse selbst bei gleichen Bereitzeiten nicht optimal!

Beispiel: 2 Prozessoren

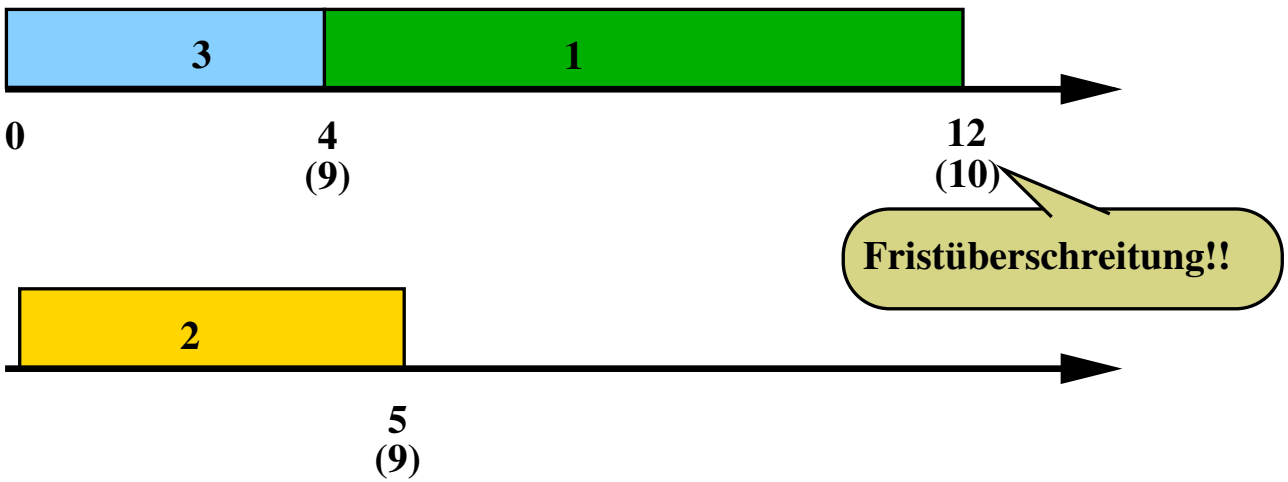
Hochschule RheinMain

Gegeben: Prozessmenge mit 3 Prozessen

Prozess i	Bedienzeit Δe_i	Frist d_i
1	8	10
2	5	9
3	4	9

Alle Aufträge seien zur Zeit Null bekannt

- Aus EDF resultierender Schedule:



- Das Planen nach Fristen berücksichtigt nicht, bis zu welchem Zeitpunkt man mit der Ausführung eines Auftrags spätestens begonnen haben muss

Least Laxity First (LLF)-Algorithmus

Hochschule RheinMain

- Def.: *Spielraum* (engl. *Laxity*)

$$\Delta lax_i = (d_i - r_i) - \Delta e_i$$

- Notwendige Bedingung für die Einplanbarkeit:

$$r_i \leq s_i \leq r_i + \Delta lax_i$$

- Strategie: Least Laxity First (LLF):
  - Gegeben sei eine Menge nicht unterbrechbarer Prozesse $P = \{P_1, P_2, ..., P_n\}$ mit gleicher Bereitzeit für M Prozessoren
  - LLF wählt denjenigen Prozess i aus, dessen Spielraum Δlax_i den kleinsten Wert hat (am kürzesten ist)
- LLF ist für Mehrprozessorsysteme optimal bei nicht-unterbrechbaren Prozessen mit gleicher Bereitzeit

Beispiel: 2 Prozessoren

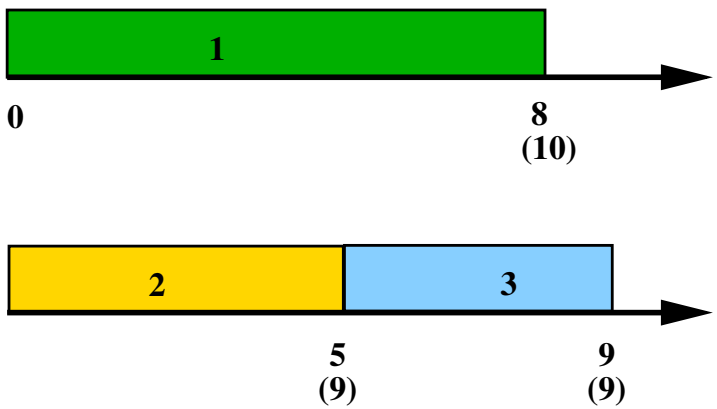
Hochschule RheinMain

Gegeben: Prozessmenge mit 3 Prozessen

Prozess i	Bedienzeit Δe_i	Frist d_i	Spielraum lax_i
1	8	10	2
2	5	9	4
3	4	9	5

Alle Aufträge seien zur Zeit Null bekannt

- Aus LLF resultierender, brauchbarer Schedule:



- Das Problem, nicht unterbrechbare Prozesse brauchbar auf einem Mehrprozessorsystem einzuplanen, ist NP-vollständig. (Gilt sogar bei nur zwei Prozessoren und Gleichheit aller Bereitzeiten und Fristen)

LLF bei beliebigen Bereitzeiten

- Das Planen nach Spielräumen bei beliebigen Bereitzeiten ist nicht optimal!
- Genauer:
  Satz: Kein Planungsverfahren für mindestens 2 Prozessoren kann optimal sein, ohne die Bereitzeiten der einzuplanenden Prozesse zu kennen

Planen nach monotonen Raten

Vorbemerkung

- Planen nach monotonen Raten erzeugt keinen expliziten Plan, sondern es entsteht ein impliziter Plan durch **feste** Vergabe von Prioritäten an Prozesse:

$$prio : P \rightarrow \mathbb{Z}$$

- Daher auch *fixed priority scheduling*
- Die meisten Echtzeit-Betriebssysteme unterstützen prioritätsbasiertes, unterbrechendes Scheduling
- Daher sind die Voraussetzungen für die Anwendung unmittelbar gegeben

Rate Monotonic Scheduling (RMS)

Hochschule RheinMain

- Es werden ausschließlich unterbrechbare, periodische Prozesse betrachtet
- Die *Rate* eines Prozesses i bezeichnet die Anzahl der Perioden in einem Betrachtungszeitraum bzw. die Frequenz bei unbegrenzter Zeit
- Strategie: Rate Monotonic Scheduling (RMS)
  - Entsprechend den Raten werden den Prozessen Prioritäten durch die Prioritätszuordnungsfunktion $rms(i)$ mit folgenden Eigenschaften zugeordnet:

$$rms(i) < rms(j) \iff \frac{1}{\Delta p_i} < \frac{1}{\Delta p_j}$$

  - Im Weiteren wird angenommen, dass die Prozesse entsprechend ihrer Priorität nummeriert sind:

$$\forall i, j : i < j \iff rms(i) < rms(j)$$

Kritischer Zeitpunkt, kritisches Intervall

- Def.: *Kritischer Zeitpunkt* eines Prozesses i ist diejenige Bereitzeit r_i^k, die abhängig von den Bereitzeiten aller übrigen Prozesse die Abschlusszeit c_i^k maximal werden lässt. Entsprechend heißt $[r_i^k, c_i^k]$ *kritisches Intervall*
- Satz: Für einen Prozess $i \in P$ ist ein kritischer Zeitpunkt r_i^k immer dann gegeben, wenn die Bereitzeiten aller höher priorisierten Prozesse auf r_i^k fallen.
  (Wenn alle Bereitzeiten höher priorisierter Prozesse mit der Bereitzeit r_i^k zusammenfallen, erhalten diese Prozesse – deren Perioden alle kürzer sind als die vom Prozess i – die maximale Rechenzeit im Intervall $[r_i^k, c_i^k]$)

Eigenschaften von RMS

- Das Planen nach monotonen Raten nimmt keine Rücksicht auf die „Wichtigkeit" von Prozessen. Das widerspricht der intuitiven Neigung, den wichtigen Prozessen Vorrang zu geben
- Dennoch ist die Prioritätenvergabe nach monotonen Raten die beste Methode zur Erzeugung brauchbarer Pläne bei periodischen Prozessen

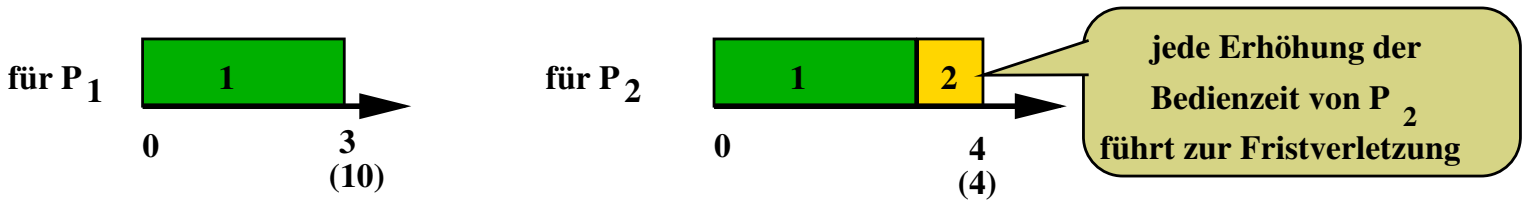
Beispiel

Hochschule RheinMain

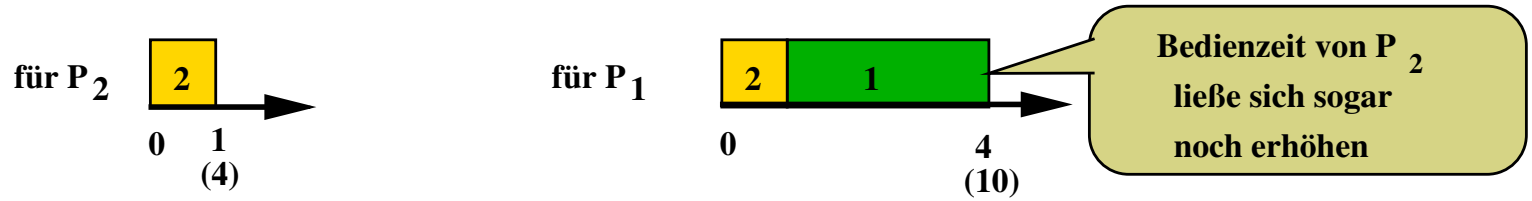
Gegeben: Prozessmenge mit 2 Prozessen

Prozess i	Bedienzeit Δe_i	Periode Δp_i	$\Rightarrow$ krit. Intervall	
1	3	10	Prio: $[0,3]$; RMS: $[0,4]$	$\longleftarrow$ **„wichtiger"**
2	1	4	Prio: $[0,4]$; RMS: $[0,1]$	

- Aus vorgegebener Priorisierung resultierender Schedule:



- Aus RMS resultierender Schedule:
  (wg. $\frac{1}{10} < \frac{1}{4}$ bekommt P_2 höhere Priorität)



Beispiel (Forts)

Hochschule RheinMain

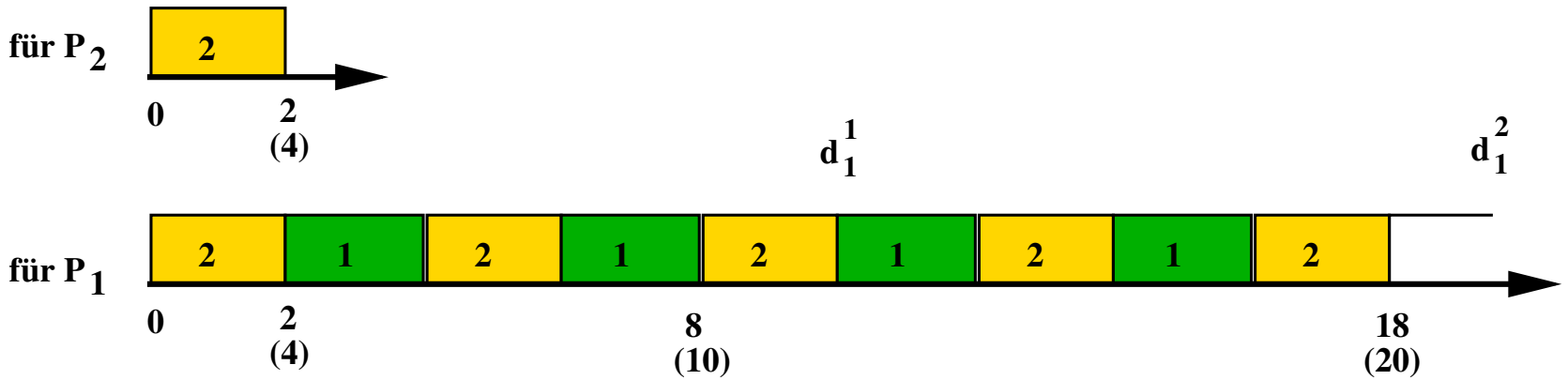
Gegeben: Prozessmenge mit 2 Prozessen

Prozess i	Bedienzeit Δe_i	Periode Δp_i	$\Rightarrow$ krit. Intervall
1	4	10	$[0, 8]$ RMS
2	2	4	$[0, 2]$ RMS

$\longleftarrow$ **„wichtiger"**

$\Uparrow$ **Jeweils um 1 erhöhte Bedienzeiten**

- Aus RMS resultierender Schedule:



Eigenschaften von RMS (2)

- Das Planen nach monotonen Raten ist nicht optimal
- Bevorzugt wird der zum aktuellen Zeitpunkt höchstpriore Prozess, obwohl durch seine Verzögerung die näherliegenden Fristen anderer Prozesse noch gehalten werden könnten

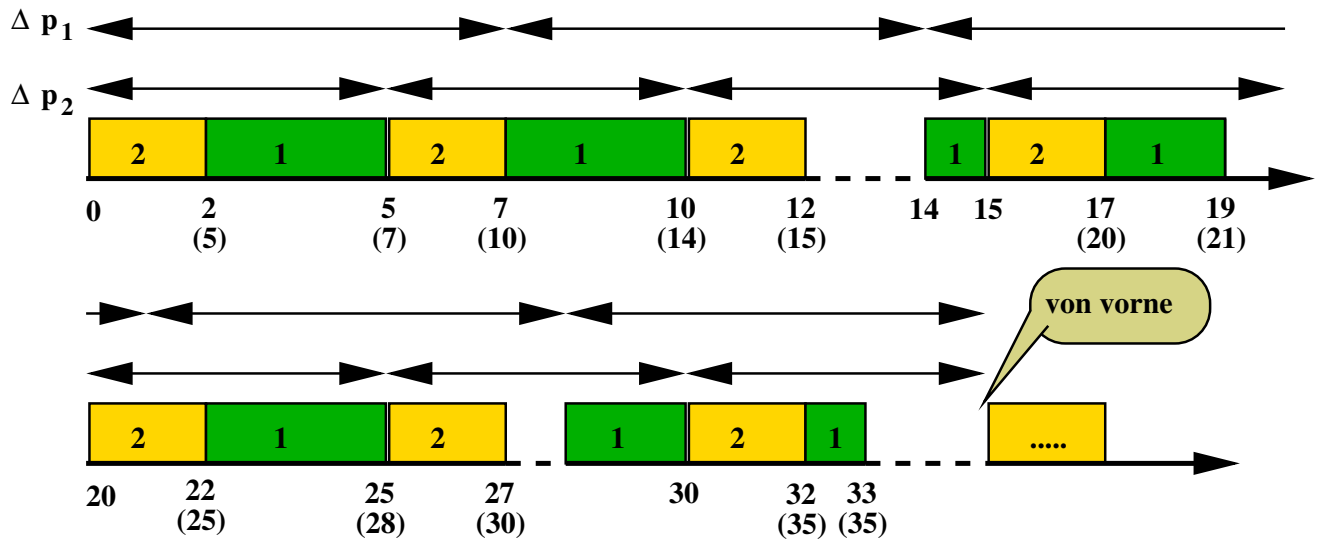
Beispiel

Hochschule RheinMain

Gegeben: Prozessmenge mit 2 Prozessen

Prozess i	Bedienzeit Δe_i	Periode Δp_i	$\Rightarrow$ krit. Intervall
1	3	7	$[0, 5]$ RMS
2	2	5	$[0, 2]$ RMS

- brauchbarer Plan existiert, da krit. Intervalle in Periode passen
- Aus RMS resultierender Schedule [2]:

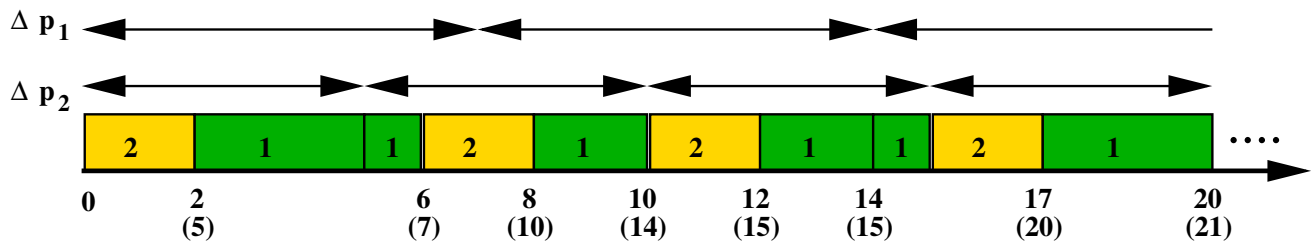


[2] (wg. $\frac{1}{7} < \frac{1}{5}$ bekommt P_2 höhere Priorität)

Beispiel (Forts.)

Hochschule RheinMain

- Die Auslastung U ist im Beispiel: $U = \frac{3}{7} + \frac{2}{5} = 0,82857...$
- Jede weitere Erhöhung der Bedienzeit führt jedoch zu einer Fristüberschreitung (z.B. für $\Delta e_1 = 4$)
- In diesem Fall ist $U = \frac{4}{7} + \frac{2}{5} = 0,9714... < 1$!
- EDF würde wegen $U \leq 1$ auch in diesem Fall einen Schedule finden:



Eigenschaften von RMS (3)

- Man kann den Prozessor nicht so gut auslasten wie im Fall, wenn man auch Fristen berücksichtigt
- Es gibt für RMS eine Grenze für die Prozessorauslastung U, für die gesichert ein Plan gefunden wird
- U_{min} bezeichnet diese Auslastung, für die RMS optimal ist, d.h. immer einen Plan findet, sofern einer existiert
- Für n Prozesse gilt:

$$U_{min} = n \cdot \left(2^{\frac{1}{n}} - 1\right)$$

- $n = 2 \Rightarrow U_{min} = 0.828427...$
- $n \to \infty \Rightarrow U_{min} = ln(2) = 0,693...$
- Im obigen Beispiel war $U_{min} = 0.82857...$, und RMS hatte trotzdem noch einen brauchbaren Schedule gefunden

Eigenschaften von RMS (4)

- Von Überlastsituationen ($U > 1$) sind zuerst nur die niederprioren Tasks betroffen

$\rightarrow$ robuster als EDF

Planen nach Zeitscheiben

- Wie bei RMS werden ausschließlich unterbrechbare, periodische Prozesse betrachtet.
- Bei bekannter Prozessorauslastung durch einzelne Tasks können individuell angepasste Zeitscheiben als Planungsgrundlage verwendet werden (*Time-Slice-Scheduling*, TSS)
- Jeder Task wird eine Zeitscheibe zugeordnet. Das Verhältnis der Dauern der Zeitscheiben zueinander entspricht dem Verhältnis der Prozessorauslastungen durch die jeweilige Task.
- Zeitscheiben werden ohne weitere Priorisierung im RR-Verfahren bearbeitet.
- Mischung aus RR und periodischem FCFS mit Unterbrechung

Beispiel

Hochschule RheinMain

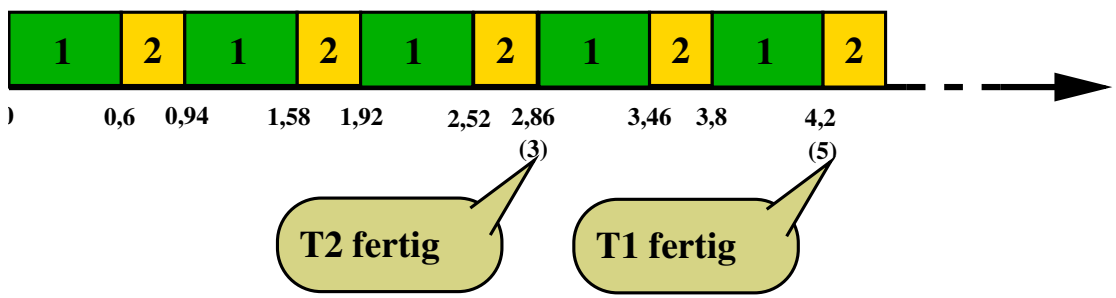
Gegeben: Prozessmenge mit 2 Prozessen

Prozess i	Bedienzeit Δe_i	Periode Δp_i	Auslastung U_i
1	3	5	$U_1 = \frac{3}{5} = 60\%$
2	1	3	$U_2 = \frac{1}{3} = 33,3\% \approx 34\%$

$$\sum_i U_i = 94\% \leq 100\%$$

- Annahme: Zeiteinheiten in ms, Zeitscheiben auf Basis 1% = 0,01 ZE

→ Resultierende Zeitscheiben: TS1 = 0,6; TS2 = 0,34



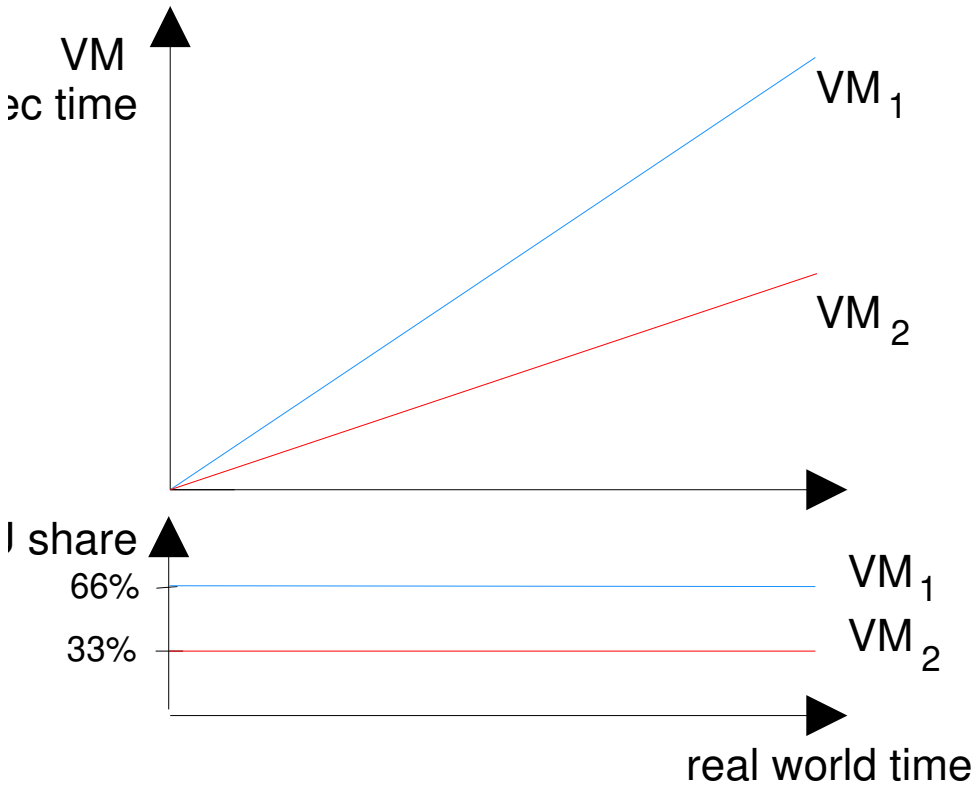
- Bei zu groben Zeitscheiben nicht optimal,
  da keine exakte Abbildung der Zeitverhältnisse der Tasks
- Bei beliebig feinen Zeitscheiben TSS → optimal,
  dann aber Effizienz → 0

Planen nach garantierten Zeitanteilen (1)

- *Guaranteed Percentage* (GP) , (auch „proportional share“ oder „Pfair“) Scheduling: eng verwandt mit TSS, Grundlage ist auch hier die bekannte Prozessorauslastung durch Tasks
- Task erhält einen prozentualen Anteil der Gesamtprozessorleistung, die dem Anteil ihrer Auslastung an der Gesamtauslastung entspricht.

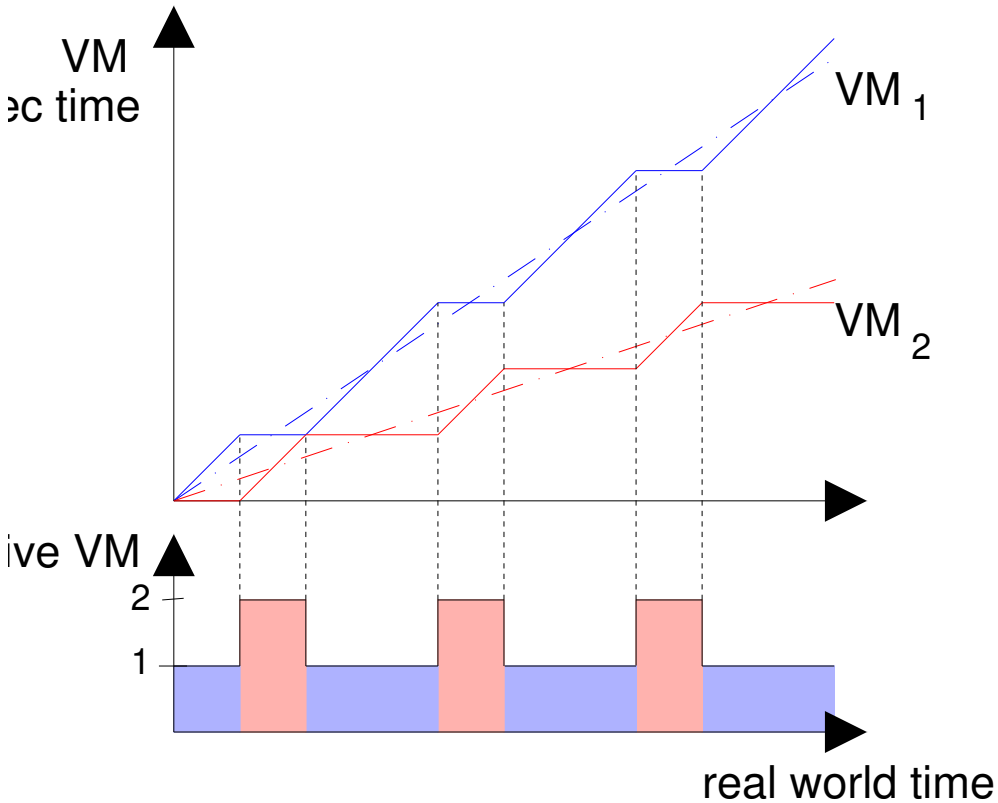
Planen nach garantierten Zeitanteilen (2)

- Mögliche Sichtweise: Jede Task erhält einen virtuellen Prozessor, der genau ihre Leistungsanforderungen abdeckt



Planen nach garantierten Zeitanteilen (3)

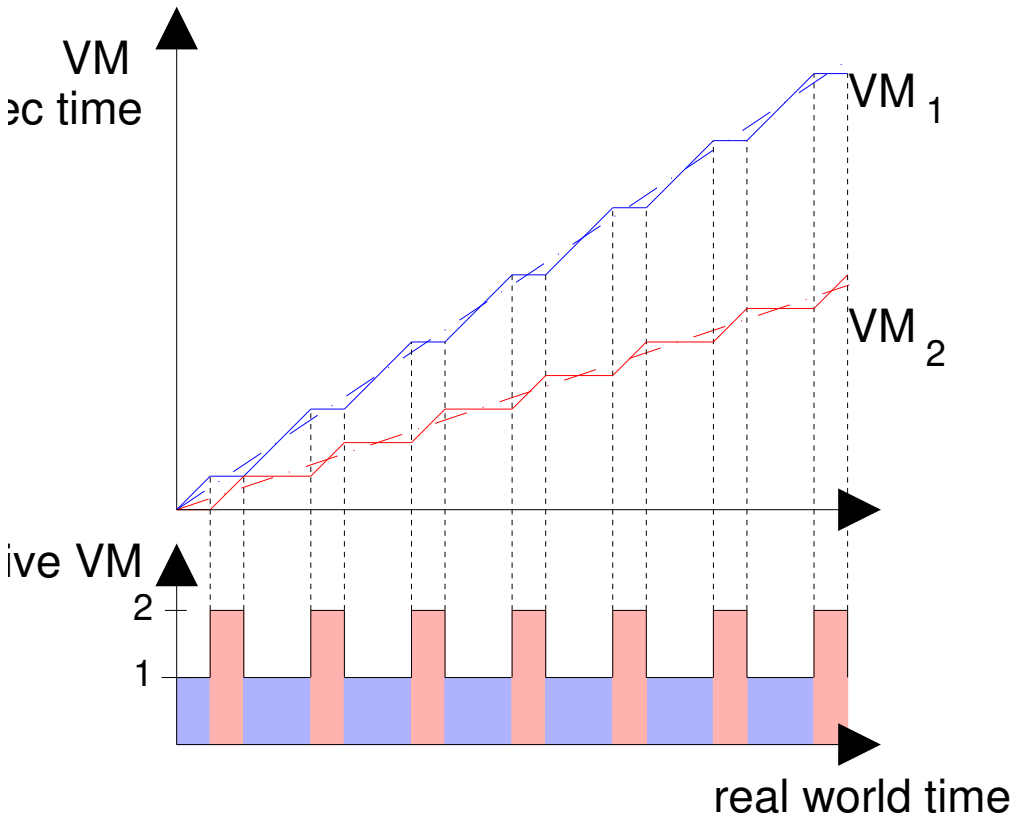
- Verwendung von homogenen Zeitscheiben ggf. verbunden mit paralleler Ausführung in Hardware.



Planen nach garantierten Zeitanteilen (4)

Hochschule RheinMain

- Erfordert für Optimalität noch feinere Zeitscheiben als TSS, daher Overhead noch größer



Vergleich der eingeführten Verfahren

Hochschule RheinMain

Verfahren	*Ein-Proz.-Systeme*	*Mehr-Proz.-Systeme*	*unterbrechbare Prozesse*	*nicht-unterbrechb. Prozesse*
Planen durch Suchen	x	x		optimal, aber nicht von prakt. Relevanz. NP-vollständig
Planen nach Fristen (EDF)	x		optimal für spor.u. period.Prozesse in stat.u.dyn. Verfahren	optimal bei Prozessen mit gleichen Bereitzeiten
Planen nach Spielräumen (LLF)		x	optimal in stat. Verfahren; nicht optimal in dyn. Verfahren	NP-vollständig
Planen nach monotonen Raten (RMS)	x		nur für period. Prozesse; optimal bis Auslastungs-grenze U_{min} ; einfach realisierbar	
Planen nach Zeitscheiben (TSS)	x		nur period. Prozesse; einfach realisierbar, aber Overhead	

Bewertung

Hochschule RheinMain

- Scheduling-„Theorie" lässt Problembereiche außer Acht:
  - genaue Bestimmung der Ausführungszeit
  - Unabhängigkeit der Ausführungszeit von den Prozessen selbst
    - ★ mittelbare: indirekte Abhängigkeiten über Betriebssysteme, ...
    - ★ unmittelbare:
      - Vorrangbeziehung: Anstoß eines Prozesses durch anderen
      - Fluss der Daten von einem Prozess zu anderem
      - Wechselseitiger Ausschluss durch Nutzung von Betriebsmitteln, die z.B. durch Semaphore geschützt sind
  - Trennung in nur sporadische oder nur periodische Prozesse nicht praxisgerecht
- Die Realität kümmert sich daher kaum um eine korrekte Analyse, auch nicht in sicherheitskritischen Anwendungen (wenig beruhigend !)

Prozesssynchronisation

- Problemstellung
  - Gegeben: Nebenläufige (konkurrente) Prozesse
  - Ziel:Vermeidung ungewollter gegenseitiger Beeinflussung
  - Ziel:Unterstützung gewollter Kooperation zwischen Prozessen:
    1. Gemeinsame Benutzung von Betriebsmitteln (Sharing)
    2. Übermittlung von Signalen
    3. Austausch von Nachrichten

→ Fazit: Mechanismen zur Synchronisation und Kommunikation von Prozessen sind notwendig

Beispiel (1)

Hochschule RheinMain

- Gemeinsame Benutzung Speicherbereiches (Hier: Datum „Kontostand")
- Ungewollte gegenseitige Beeinflussung

Prozess 1

```
/* Gehaltsueberweisung */
z = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
```

Prozess 2

```
/* Dauerauftrag Miete */
x = lies_kontostand();
x = x - 800;
schreibe_kontostand(x);
```

- (s.o.) Prozesse arbeiten „quasi-gleichzeitig" → können an belieber Stelle unterbrochen und später fortgeführt werden

Beispiel (2)

Hochschule RheinMain

Mögliche Ausführungsreihenfolge der Anweisungen

```
/* Gehaltsueberweisung */ /* Dauerauftrag Miete */
z = lies_kontostand();
                          x = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
                          x = x - 800;
                          schreibe_kontostand(x);
```

- Pech, die Gehaltsüberweisung ist „verloren gegangen"
- Bei anderen Reihenfolgen werden die beiden Berechnungen „richtig" ausgeführt, oder es geht der Dauerauftrag verloren

Definitionen(1)

Hochschule RheinMain

- Annahme: Prozesse mit Lese/Schreib-Operationen auf Betriebsmitteln
- **Def.**: Zwei nebenläufige Prozesse heissen im Konflikt zueinander stehend oder überlappend, wenn es ein Betriebsmittel gibt, das sie gemeinsam (lesend und schreibend) benutzen, ansonsten heißen sie unabhängig oder disjunkt
  (N.B.: Die Scheduling Theorie, wie sie bisher vorgestellt wurde, geht von unabhängigen Prozessen aus)
- **Def.**: Folgen von Lese / Schreib-Operationen der verschiedenen Prozesse heißen zeitkritische Abläufe (engl. *race conditions*), wenn die Endzustände der Betriebsmittel (Endergebnisse der Datenbereiche) abhängig von der zeitlichen Reihenfolge der Lese / Schreib-Operationen sind

Definitionen (2)

Hochschule RheinMain

- **Def.**: Verfahren zum wechselseitigen Ausschluss (engl. *mutual exclusion*): Ein Verfahren, das verhindert, dass zu einem Zeitpunkt mehr als ein Prozess auf ein Betriebsmittel zugreift
- **Def.**: Kritischer Abschnitt oder kritischer Bereich (engl. *critical section* oder *critical region*): der Teil eines Programms, in dem auf gemeinsam benutzte Betriebsmittel (z.B.Datenbereiche) zugegriffen wird
- Ein Verfahren, das sicherstellt, dass sich zu keinem Zeitpunkt zwei Prozesse in ihrem kritischen Abschnitt befinden, vermeidet zeitkritische Abläufe
- Kritische Abschnitte realisieren sogenannte komplexe unteilbare oder atomare Operationen

(Nicht-)atomare Operationen (1)

Hochschule RheinMain

Nicht atomar

```
/* Gehaltsueberweisung */
z = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
```

- Auch Operationen, die in Hochsprache (hier: C) atomar zu sein scheinen, sind es auf Maschinenebene u.U. nicht!
- Architekturabhängig

Jetzt atomar??

```
/* Gehaltsueberweisung 2.0 */
int kontostand;

kontostand += 1000;
```

⇒ Maschinencode:

```
lw r2,kontostand
addu r3,r2,#1000
sw r3,kontostand
```

→ 3 Instruktionen: **unterbrechbar!!**

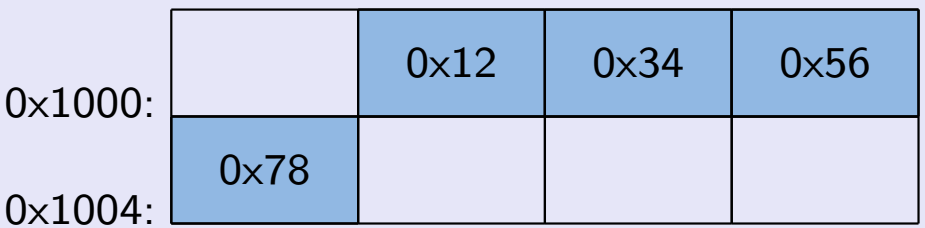
(Nicht-)atomare Operationen (2)

Hochschule RheinMain

- Sogar einzelne Maschinenbefehle können u.U. nicht-atomar sein

Beispiel

```
la  r2,0x1001
sw  0x12345678,kontostand
```



- Zugriff auf ungerade Adresse erfordert mehrere Buszyklen

→ Unterbrechbar!

„Guter" Algorithmus zum wechselseitigen Ausschluss

- Anforderungen:
  1. Zu jedem Zeitpunkt darf sich nur ein Prozess in seinem kritischen Abschnitt befinden (Korrektheit, Basisforderung)
  2. Es dürfen keine Annahmen über die Ausführungsgeschwindigkeiten oder die Anzahl der Prozessoren gemacht werden
  3. Kein Prozess, der sich nicht in seinem kritischen Abschnitt befindet, darf andere Prozesse blockieren (Fortschritt)
  4. Alle Prozesse werden gleich behandelt (Fairness)
  5. Kein Prozess darf unendlich lange warten müssen, bis er in seinen kritischen Abschnitt eintreten kann

Wechselseitiger Ausschluss mit aktivem Warten

Generelles Vorgehen

```
enter_crit();      /* Prolog */
<statement>;
...              /* critical section */
<statement>;
leave_crit();      /* Epilog */
```

- Prolog/Epilog-Paar
- Aktives Warten auf Eintritt in den kritischen Abschnitt (engl. *busy waiting*) (Aktives Warten für einen längeren Zeitraum verschwendet Prozessorzeit)
- Alle Prozesse müssen sich an das Vorgehen halten (nicht forcierbar!)

Mögliche Lösungsansätze

- Ansätze zum wechselseitigen Ausschluss:
  1. Sperren aller Unterbrechungen
  2. Sperrvariablen
  3. Striktes Alternieren
  4. Peterson-Algorithmus
  5. Test-and-set-Instruktion
- Nur die letzten beiden sind brauchbar ...

Sperren aller Unterbrechungen

- Der Prozessor kann nur dann zu einem anderen Prozess wechseln, wenn eine Unterbrechung auftritt

→ Einfachste Lösung:

  - Jeder Prozess sperrt vor Eintritt in seinen kritischen Abschnitt alle Unterbrechungen (*disable interrupts*)
  - Jeder Prozess lässt die Unterbrechungen am Ende seines kritischen Abschnitts wieder zu (*enable interrupts*)

- Korrektheitsforderung wird erfüllt
- Dennoch unbrauchbar für allgemeine Benutzerprozesse: keine Garantie, dass Unterbrechungen Jemals wieder zugelassen werden
- Wird aber häufig *innerhalb* von Betriebssystemen verwendet (*trusted code*)
- Generell nicht brauchbar bei Mehrprozessorsystemen

Sperrvariablen (1)

Hochschule RheinMain

- Jedem kritischen Abschnitt wird eine Sperrvariable zugeordnet
- Wert 0 bedeutet „frei", Wert 1 bedeutet „belegt"
- Jeder Prozess prüft die Sperrvarable vor Eintritt in den kritischen Abschnitt
- Ist sie 0, so setzt er sie auf 1 und tritt in den krit. Abschnitt ein
- Ist sie 1, so wartet er, bis sie den Wert 0 annimmt
- Am Ende seines kritischen Abschnitts setzt der Prozess den Wert der Sperrvariablen auf 0 zurück

Prozess 1

```
while(sperrvar) {}
sperrvar = 1;
/* kritischer Bereich */
....
sperrvar = 0;
```

Prozess 2

```
while(sperrvar) {}
sperrvar = 1;
/* kritischer Bereich */
....
sperrvar = 0;
```

Sperrvariablen (2)

Hochschule RheinMain

Mögliche Ausführungsreihenfolge der Anweisungen

```
while(sperrvar) {}
                                    while(sperrvar) {}
sperrvar = 1;
                                    sperrvar = 1;
/* kritischer Bereich */
....                                /* kritischer Bereich */
sperrvar = 0;                       ....
                                    sperrvar = 0;
```

- Gleicher Fehler wie beim Konto-Beispiel:Zwischen Abfrage der Sperrvariablen und folgendem Setzen kann der Prozess unterbrochen werden
- Damit ist es möglich, dass sich beide Prozesse im kritischen Abschnitt befinden (Korrektheitsbedingung verletzt !!)

Striktes Alternieren

Hochschule RheinMain

Prozess 1

```
while (1) {
  while (dran != 1) { }
  /* kritischer Bereich */
  dran = 2;
  /* unkritischer Ber. */
}
```

Prozess 2

```
while (1) {
  while (dran != 2) { }
  /* kritischer Bereich */
  dran = 1;
  /* unkritischer Ber. */
}
```

- Gemeinsame Variable „dran" gibt an, welcher Prozess den kritischen Bereich betreten darf (Anfangswert z.B. 1)
- Prozesse wechseln sich ab: 2,1,2,1,...
- Lösung erfüllt Korrektheitsbedingung, aber
- Fortschrittsbedingung (3) kann verletzt sein, wenn ein Prozess wesentlich langsamer als der andere ist
- Fazit: keine ernsthafte Lösung.

Lösung von Peterson

Hochschule RheinMain

- Historische Vorläufer:
  - Anfang der 60er Jahre viele Lösungsansätze, nur wenige erfüllten alle genannten Bedingungen
  - Erste korrekte Software-Lösung für 2 Prozesse: Algorithmus von Dekker
- Neue Lösung zum wechselseitigen Ausschluss:
  - Lösung von Peterson (1981) (im folgenden betrachtet)
  - Basiert (ebenfalls) auf unteilbaren Speicheroperationen und aktivem Warten, ist aber einfacher
  - Prolog: `enter_crit()`, Epilog: `leave_crit()`
- Weitere Lösungen für mehr als zwei Prozesse von:
  - Dijkstra, Peterson, Knuth, Eisenberg/McGuire, Lamport (hier nicht weiter diskutiert)

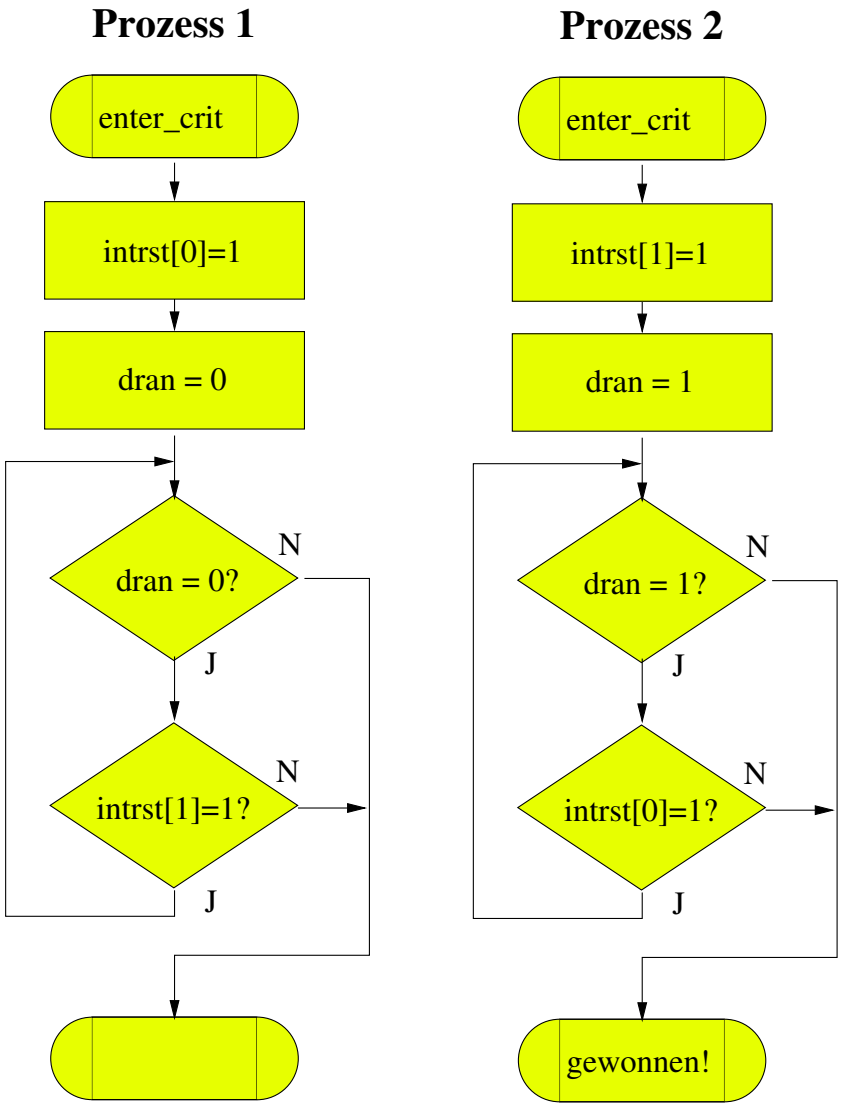
Peterson-Algoritmus (1)

Hochschule RheinMain

```
#define N    2                      /* Anzahl der Prozesse              */

/* gemeinsame Variablen                                                 */
volatile int dran;                  /* Wer kommt dran?                  */
volatile bool interested[N];        /* Wer will, anfangs alle false     */

/* Prolog-Operation, vor Eintritt in den
   kritischen Bereich ausfuehren    */
void enter_crit(int process) {      /* process: wer tritt ein: 0,1      */
   int other;                       /* Nummer des anderen Prozesses     */
   other = 1 - process;
   interested[process] = true;      /* zeige eigenes Interesse          */
   dran = process;                  /* setze Marke, unteilbar!          */
   while(dran==process && interested[other]);
                                    /* ev. Aktives Warten !!!           */
}

/* Epilog-Operation, nach Austritt aus dem
   krit. Bereich ausfuehren */
void leave_crit(int process) {      /* process: wer verlaesst: 0,1      */
   interested[process] = false;     /* Verlasse krit. Bereich           */
}
```

Peterson-Algorithmus (2)

Hochschule RheinMain

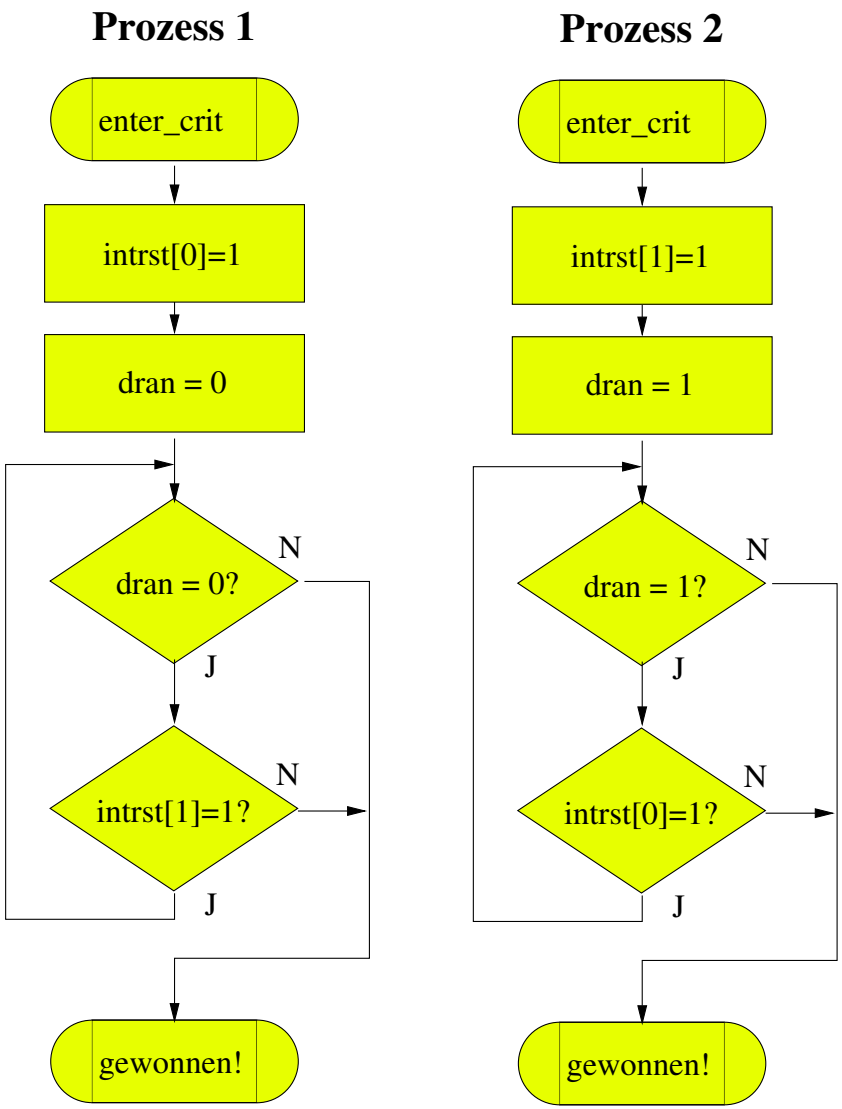


Anfangszustand

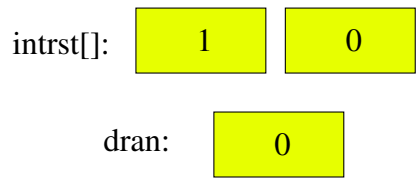
intrst[]:	0	0
dran:	0	

Peterson-Algorithmus (3)

Hochschule RheinMain

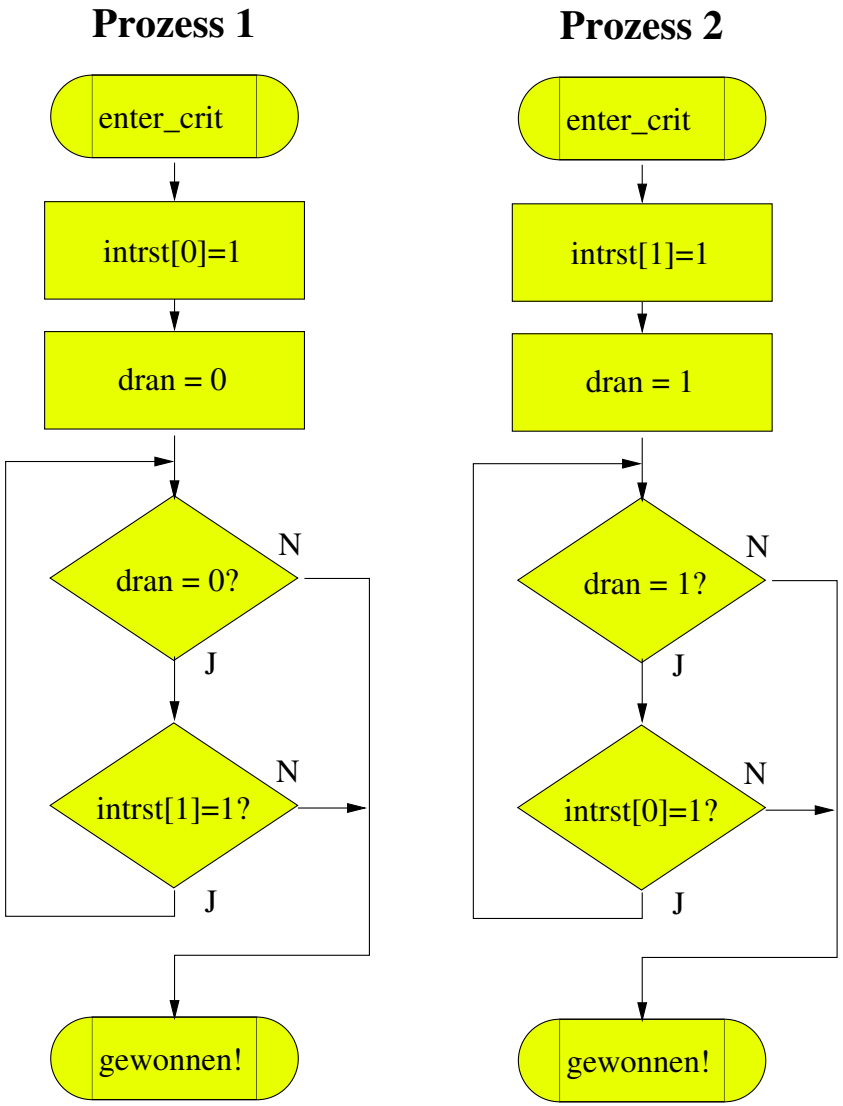


Prozess 1 betritt kritischen Abschnitt, Prozess 2 nicht

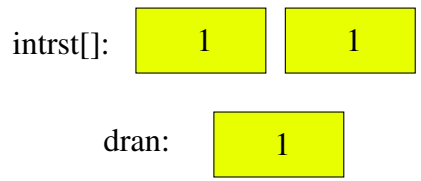


Peterson-Algorithmus (4)

Hochschule RheinMain



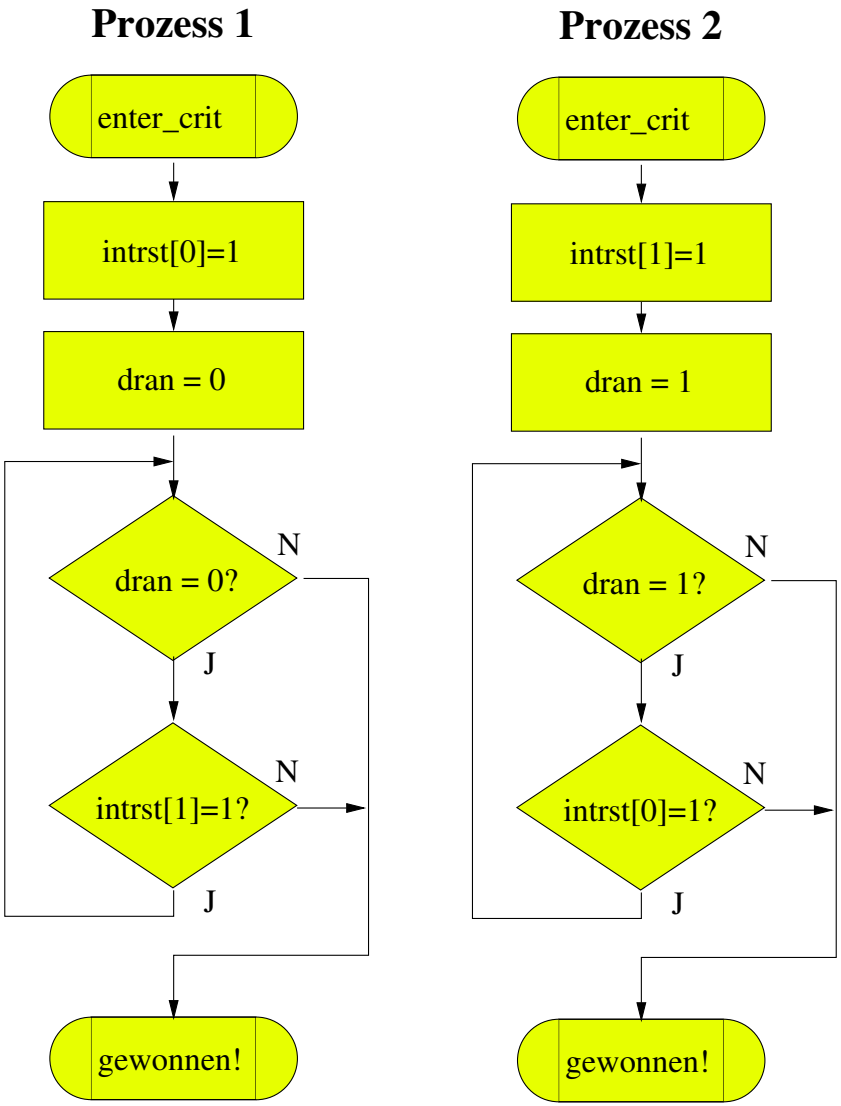
Prozess 1 ist in kritischem Abschnitt, Prozess 2 versucht, ihn zu betreten



Prozess 2 wartet, bis intrst[0] = 0

Peterson-Algorithmus (5)

Hochschule RheinMain



Kritischer Fall:
beide versuchen gleichzeitig,
den kritischen Abschnitt
zu betreten

intrst[]: 1 1

dran: 0 oder 1

Schreibzugriff auf „dran'' ist atomar. Nur einer von beiden kann „dran'' erfolgreich setzen.

Wer Erfolg hat, muss hier warten!

Test-and-Set-Befehl (1)

Hochschule RheinMain

- Algorithmen, die nur auf atomaren Speicherzugriffen basieren sind
  - Komplex → fehlerträchtig
  - Auf eine feste Teilnehmerzahl beschränkt
- Zudem besteht bei einigen Verfahren die Gefahr des „Verhungerns" (engl. *starvation*)

→ Notwendigkeit einer effizienten Lösung

- Lösung durch Hardware-Unterstützung:
- Einführung eines Maschinenbefehls „Test-and-Set": unteilbares Lesen einer Speicherzelle mit anschließendem Schreiben eines Wertes in die Zelle (Speicherbus wird zwischendurch **nicht** freigegeben)

Test-and-Set-Befehl (2)

- Jedem kritischen Abschnitt wird eine Sperrvariable „flag" zugeordnet
- Wert 0 bedeutet „frei", Wert 1 bedeutet „belegt"
- Auf Test-and-Set basierende Sperrvariablen mit aktivem Warten heißen auch *spin locks*
- Große Bedeutung in Multiprozessor-Betriebssystemen
- Typische Assembler-Routinen für Prolog und Epilog mit Test-and-Set-Instruktion „TAS":

```
enter_crit:
    tas reg,flag      | kopiere flag in reg, setze flag = 1
    cmp reg,#0        | war flag vorher 0?
    jnz enter_crit    | nein -> Sperre war belegt -> warten
    ret               | ja -> darf einreten, zurueck zum Aufrufer

leave_crit:
    mov flag,#0       | loesche flag -> setze Sperre zurueck
    ret               | zurueck zum Aufrufer
```

- (Im Gegensatz zur Darstellung hier erfolgt die Realisierung i.d.R. über Makros, da Prozeduraufrufe zur Laufzeit zu großen Overhead darstellen)

Wechselseitiger Ausschluss mit passivem Warten

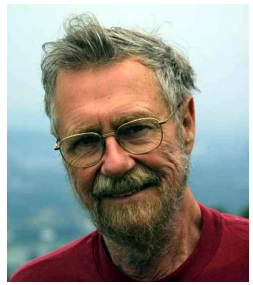
- Bisher:
  - Prolog-Operationen zum Betreten des kritischen Abschnitts führen zum aktiven Warten, bis der betreffende Prozess in den krititschen Abschnitt eintreten kann
  - Unerwünscht: Lediglich auf Multiprozessor-Systemen kann kurzzeitiges aktives Warten zur Vermeidung eines Prozesswechsels sinnvoll sein (spin locks)
- Ziel: Vermeidung von verschwendeter Prozessorzeit
- Vorgehensweise:
  - Prozesse blockieren, wenn sie nicht in ihren kritischen Abschnitt eintreten können
  - Ein solcher blockierter Prozess wird durch den aus seinem kritischen Abschnitt austretenden Prozess entblockiert

Einfache Primitive

- Einfachste Primitive: SLEEP() und WAKEUP():
  - SLEEP() blockiert den ausführenden Prozess, bis er von einem anderen Prozess geweckt wird
  - WAKEUP(process) weckt den Prozess process. Der ausführende Prozess wird dabei nie blockiert
- Häufig wird als Parameter von SLEEP und WAKEUP ein Ereignis (Speicheradresse einer beschreibenden Struktur) verwendet, um die Zuordnung treffen zu können
- Diese Primitive können auch der allgemeinen ereignisorientierten Kommunikation dienen

Semaphoren

Hochschule RheinMain

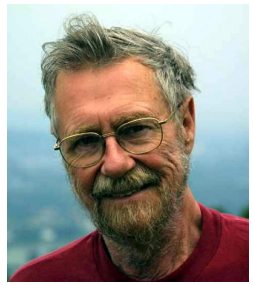


- 1965 von Edsger W. Dijkstra eingeführt
- Supermarkt-Analogie:
  - Kunde darf den Laden nur mit einem Einkaufswagen betreten
  - Es steht nur eine bestimmte Anzahl von Einkaufswagen bereit
  - Sind alle Wagen vergeben, müssen neue Kunden warten, bis ein Wagen zurückgegeben wird

- Semaphor besteht aus:
  - einer **Zählvariablen**, die begrenzt, wieviele Prozesse augenblicklich ohne Blockierung passieren dürfen
  - einer **Warteschlange** für (passiv) wartende Prozesse

Semaphoren

Hochschule RheinMain



- 1965 von Edsger W. Dijkstra eingeführt
- Supermarkt-Analogie:
  - Kunde darf den Laden nur mit einem Einkaufswagen betreten
  - Es steht nur eine bestimmte Anzahl von Einkaufswagen bereit
  - Sind alle Wagen vergeben, müssen neue Kunden warten, bis ein Wagen zurückgegeben wird

- Semaphor besteht aus:
  - einer **Zählvariablen**, die begrenzt, wieviele Prozesse augenblicklich ohne Blockierung passieren dürfen
  - einer **Warteschlange** für (passiv) wartende Prozesse

Operationen auf Semaphoren

Hochschule RheinMain

- **Initialisierung**
  - Zähler auf initialen Wert setzen
  - „Anzahl der freien Einkaufswagen"


- **Operation P()**: Passier(-Wunsch)
  - Zähler $= 0 \rightarrow$ Prozess in Warteschlange setzen, blockieren
  - Zähler > 0: Prozess kann passieren
  - In beiden Fällen wird der Zähler (ggf. nach dem Ende der Blockierung) erniedrigt
  - P steht für „proberen" (niederländisch für „testen")
- **Operation V()**: Freigeben
  - Zähler wird erhöht
  - Falls es Prozesse in der Warteschlage gibt, wird einer de-blockiert (und erniedrigt den Zähler dann wieder, s.o.)
  - V steht für „verhogen" (niederländisch für „erhöhen")

Operationen auf Semaphoren

- P() und V()sind atomare Operationen:
  - Das Überprüfen, Dekrementieren und Sich-Schlafen-Legen des Aufrufers bei P() sowie das inkrementieren und ggf. Wecken eines Prozesses bei V() ist jeweils eine Transaktion, die nur vollständig „in einem Zuge" ausgeführt werden kann
- Kein Prozess wird bei der Ausführung der V()-Operation blockiert

Implementierung von Semaphoren

- I.d.R. als Systemaufrufe
- Intern Nutzung für Sich-Schlafen-Legen und Aufwecken
- Wesentlich ist die Unteilbarkeit von P() und V():
  - Auf Einprozessorsystemen: durch Sperren aller Unterbrechungen während der Ausführung von P() und V(). Zulässig, da nur wenige Maschineninstuktionen zur Implementierung nötig sind.
  - Auf Multiprozessorsystemen: Jedem Semaphor wird eine Test-and-Set-Sperrvariable mit aktivem Warten vorgeschaltet. So kann stets höchstens ein Prozessor den Semaphor manipulieren
  - Man beachte: Unterscheide zwischen aktivem Warten auf den Zugang zum Semaphor, der einen kritischen Abschnitt schützt (einige Instruktionen, Mikrosekunden) und Aktivem Warten auf den Zugang zum kritischen Abschnitt selbst (problemabhängig, Zeitdauer nicht vorab bekannt oder begrenzt)

Binärsemaphore und Zählsemaphore

- Semaphore, die nur die Werte 0 und 1 annehmen, heißen *binäre Semaphore*
- ansonsten heißen sie *Zählsemaphore*
- Binäre Semaphore dienen zur Realisierung des wechselseitigen Ausschlusses
- Ein mit $n > 1$ initialisierter Zählsemaphor kann zur Kontrolle der Benutzung eines in n Exemplaren vorhandenen Typs von sequentiell wiederverwendbaren Betriebsmitteln verwendet werden
- Semaphore sind weit verbreitet, innerhalb von Betriebssystemen und zur Programmierung nebenläufiger Anwendungen
- Programmierung mit Semaphoren ist oft fehleranfällig, wenn mehrere Semaphore benutzt werden müssen (Deadlocks)

Anwendung: Wechselseitiger Ausschluss

Hochschule RheinMain

Wechselseitiger Aussschluss

```
semaphore sem = 1;   /* Init. mit 1      */
...
P(sem);              /*   Prolog         */
     <statement> ;   /* krit. Abschnitt */
      ...
     <statement> ;   /* krit. Abschnitt */
V(sem);              /*   Epilog         */
...
```

- Der mit 1 initialisierte Semaphor `sem` wird von allen beteiligten Prozessen benutzt
- Jeder Prozess klammert seinen kritischen Abschnitt mit `P(sem)` zum Eintreten und `V(sem)` zum Verlassen

Anwendung: Betriebsmittelvergabe

Hochschule RheinMain

- Es seien `n` Exemplare vom Betriebsmitteltyp `bm` vorhanden
- Jedes BM dieses Typs sei sequentiell benutzbar

Betriebsmittelvergabe

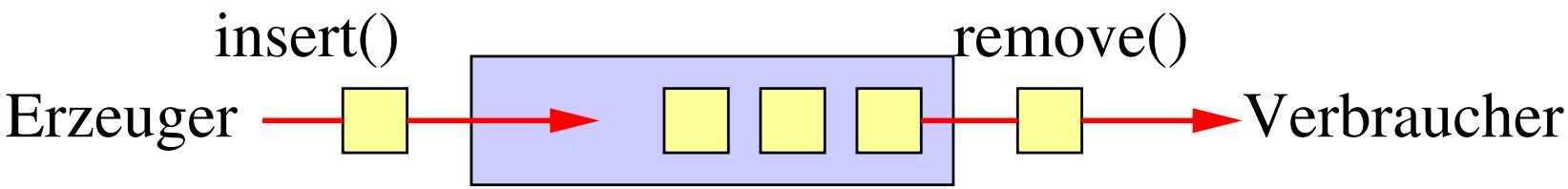
```
semaphore bm = n;

P(bm);          /*   Beantragen     */

    Benutze das Betriebsmittel

V(bm);          /*   Freigeben      */
```

- Dem BM-Typ `bm` wird ein Semaphor `bm`, zugeordnet
- Beantragen eines BM durch `P(bm)`, evtl. blockieren, bis ein BM verfügbar wird
- Nach der Benutzung freigeben des BM durch `V(bm)`
  (Es kann damit von einem weiteren Prozess benutzt werden)

Beispiel: Erzeuger/Verbraucher-Problem

Hochschule RheinMain

- Gegeben:
  - Datenobjekte (z.B. Nachrichten) fester Größe
  - Pufferspeicher von begrenzter Größe (n Objekte)
- Aufgabe
  - Verschiedene Prozesse legen Daten mit Hilfe einer Funktion `insert()` im Puffer ab („Erzeuger"), bzw. entnehmen sie mit einer Funktion `remove()` („Verbraucher")
  - `remove()` soll warten, wenn keine Objekte im Puffer sind
  - `insert()` soll warten, wenn kein Platz mehr im Puffer ist



Lösung mit Semaphoren

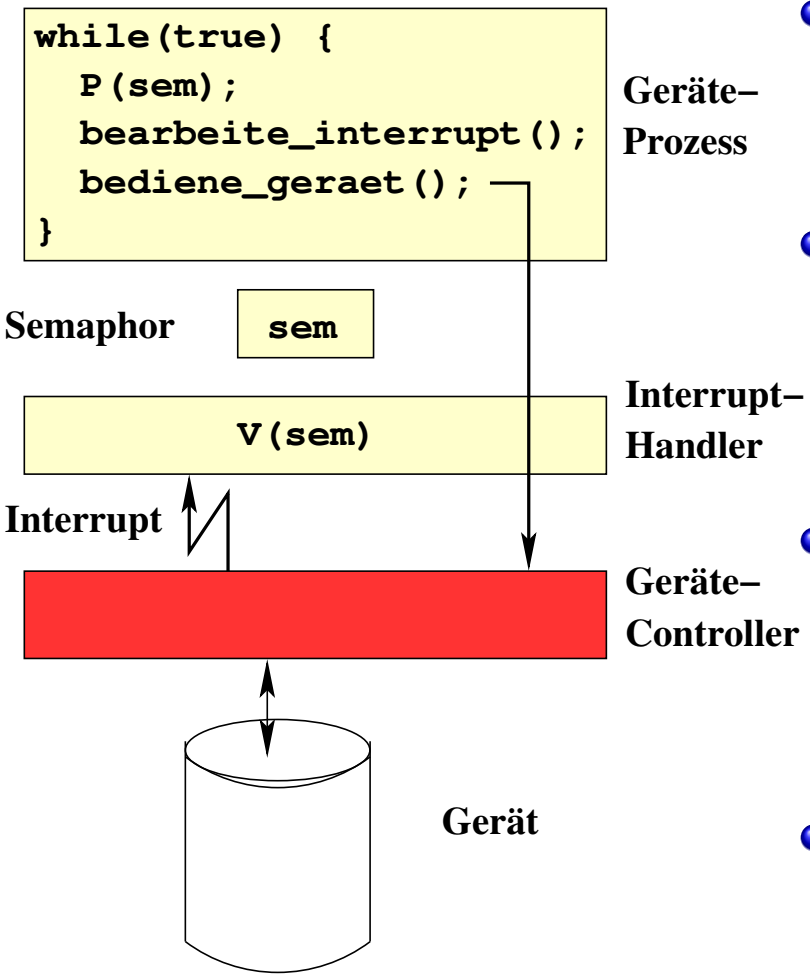
Hochschule RheinMain

Erzeuger/Verbraucher

```
/* Initialisierung der Semaphore:
 * empty = Puffergroesse,
 * full=0, mutex=1
 */

void insert(int item) {
    P(&empty);
    P(&mutex);
/* item in Puffer stellen */
    V(&mutex);
    V(&full);
}

int remove(void) {
    P(&full);
    P(&mutex);
/* vorderstes Item aus
   Puffer holen */
    V(&mutex);
    V(&empty);
}
```

- `full`:zählt belegte Einträge im Puffer, verhindert Entnahme aus leerem Puffer
- `empty`:verwaltet freie Plätze im Puffer, verhindert Einfügen in vollen Puffer
- `mutex`:schützt den kritischen Bereich vor gleichzeitigem Betreten (binär-Semaphor: nimmt nur Werte 1/0 an)

Anwendung: Verstecken von Interrupts



- Einem E/A-Gerät wird ein Geräteprozess und ein mit 0 initialisierter Semaphor zugeordnet
- Nach dem Start des Geräteprozesses führt dieser eine P-Operation auf dem Semaphor aus (und blockiert)
- Im Falle eines Interrupts des Gerätes führt der Interrupt-Handler eine V-Operation auf dem Semaphor aus
- Der Geräteprozess wird entblockiert (und dadurch rechenwillig) und kann das Gerät bedienen

P() und V() in Echtzeitsystemen

- Die meisten (alle?) (Echtzeit-)Betriebssysteme bieten Dienste zum Umgang mit Semaphoren
- In der Regel heißen diese aber *nicht* P() und V()
- Beispiele:
  - POSIX Threads Interface (pthreads): Mutex-Funktionen `phtread_mutex_xxx()` implementieren binäre Semaphoren
  - UNIX System V Release 4 (SVR4): Funktionen `semget()`, `semop()`, `semctl()` arbeiten auf *Mengen* von semaphoren
  - Windows NT: Zählsemaphore als Systemobjekte, Funktionen: `CreateSemaphore()`, `ReleaseSemaphore()`, `OpenSemaphore()`, `WaitForSingleObject()`

Planen und Synchronisation

Überblick

- Berücksichtigung von Abhängigkeiten zwischen Prozessen, die durch Synchronisation entstehen

1. Problem der Prioritätsinversion (*Priority Inversion Problem*)
2. Prioritätsvererbungsprotokoll (*Priority Inheritance Protocol*)
3. Prioritätsanhebungsprotokoll (*Priority Ceiling Protocol*)

Problem der Prioritätsinversion

- Problem der Prioritätsinversion tritt bei prioritätsbasiertem Scheduling unterbrechbarer Prozesse in Zusammenhang mit kritischen Abschnitten auf
- Szenario:
  - Prozess H habe hohe Priorität, Prozess L habe niedere Priorität. Scheduling-Regel: wenn H rechenwillig ist, soll H ausgeführt werden
  - Es werde H rechenwillig, während L sich in seinem kritischen Abschnitt befindet. H wolle ebenfalls in seinen kritischen Abschnitt eintreten
  - Der höher priore Prozess H kann nicht weiter ausgeführt werden, da L den kritischen Abschnitt noch nicht freigegeben hat
  - Die Dauer der Verzögerung hängt nicht nur von L ab, sondern von allen Prozessen, die aufgrund ihrer Priorität vor H ausgeführt werden
- Es muss also der nieder priore Prozess ausgeführt werden!
  Diese Situation heißt Prioritätsinversionsproblem.

Praktische Relevanz....

- Siehe Mike Jones: „What Happened on Mars?" (`www.cs.cmu.edu/afs/cs/user/raj/www/mars.html`)

„The Mars Pathfinder mission was widely proclaimed as "flawless" in the early days after its July 4th, 1997 landing on the Martian surface. ... But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. ...
Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.
Pathfinder contained an "information bus", which you can think of as a shared memory area ... Access to the bus was synchronized with mutual exclusion locks (mutexes).
The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus .. When publishing its data, it would acquire a mutex ... If an interrupt caused the information bus thread to be scheduled while this mutex was held, ..., this would cause it to block on the mutex, waiting until the meteorological thread released the mutex ... The spacecraft also contained a communications task that ran with medium priority.
Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running... After some time had passed, a watchdog timer would go off ... and initiate a total system reset.
This scenario is a classic case of priority inversion."

Prioritätsvererbungsprotokoll

- *Priority Inheritance Protocol*
  - Jeder Prozess läuft mit seiner ihm zugewiesenen Priorität, es sei denn, er befindet sich in einem kritischen Abschnitt und blockiert einen höher prioren Prozess
  - Wenn ein Prozess in einem kritischen Abschnitt einen höher prioren Prozess blockiert, erbt er die höchste Priorität aller durch ihn blockierten Prozesse
  - Verlässt ein Prozess einen kritischen Abschnitt, innerhalb dessen er eine höhere Priorität ererbt hat, so wird er auf seine Priorität zum Zeitpunkt des Eintritts in den krit. Abschnitt zurückgesetzt
- Prioritätsvererbung ist transitiv und kann mehrfach innerhalb eines krit. Abschnitts erfolgen
- Das Prioritätsvererbungsprotokoll kann Deadlocks nicht ausschließen und löst daher nicht alle Probleme

Prioritätsanhebungsprotokoll (1)

Hochschule RheinMain

- *Priority Ceiling Protocol*
  - Jedem kritischen Abschnitt s wird eine Prioritätsgrenze $ceil(s)$ zugeordnet. Dieser Wert entspricht der höchsten Priorität aller Prozesse, die diesen kritischen Abschnitt jemals betreten wollen (**Achtung**: muss damit vorab bestimmbar sein!)
  - Ein Prozess darf seinen kritischen Abschnitt nur betreten, wenn er von keinem anderen Prozess, der sich bereits in kritischen Abschnitten befindet, verzögert werden kann. Maßgeblich ist die aktuelle Prioritätsgrenze, die von allen übrigen Prozessen, die kritische Abschnitte gesperrt haben, bestimmt wird:

$$actceil(i) = max\{ceil(s) | s \in locked\_crit\_sect(i)\}$$

  - Entsprechend darf ein Prozess i einen kritischen Abschnitt nur betreten, wenn seine aktuelle Priorität (d.h. anfängliche oder ererbte) so groß ist, dass gilt:

$$actprio(i) > actceil(i)$$

Prioritätsanhebungsprotokoll (2)

- Protokoll:
  - Wenn ein Prozess einen krit. Abschnitt beansprucht und seine Priorität unter der Ceiling-Priorität liegt, so wird sie vorübergehend auf die Ceiling-Priorität angehoben
  - Wenn ein Prozess den krit. Abschnitt freigibt, fällt seine Priorität auf den ursprünglichen Wert zurück
- Das Prioritätsanhebungsprotokoll vermeidet Deadlocks
- Die Verzögerungszeit des jeweils höchstpriorisierten Prozesses ist durch den längsten krit. Abschnitt bestimmt, den er mit irgendeinem niedriger priorisierten Prozess gemeinsam hat

WCET-Analyse

- In der Echtzeitplanung wird die Kenntnis der (als konstant angenommenen) Ausführungszeit eines Prozesses vorausgesetzt
- Tatsächlich hängt diese ab von
  - Programmcode des Prozesses
  - Zustand des Prozesses zum Startzeitpunkt
  - Geschwindigkeit des Prozessors
  - Zustand des Prozessors
    (Pipeline, Cache(s), Tag-RAM für Sprungvorhersage, ...)
- Bestimmung der Ausführungszeit ist komplex
- Ausführungszeit ist nicht konstant: variiert zwischen verschiedenen Prozessinstanzen

Ad-hoc Methode

Hochschule RheinMain

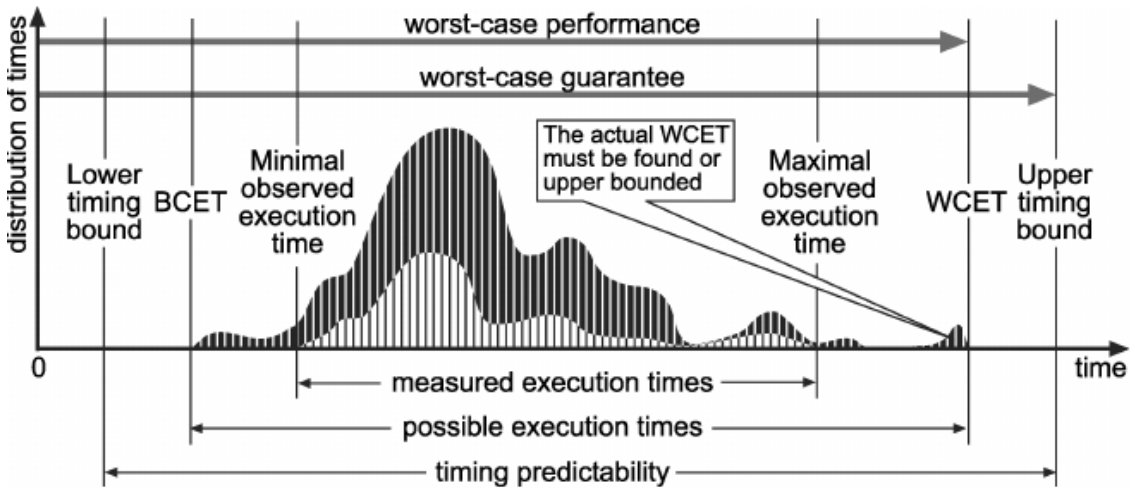
In der Vergangenheit (und leider auch heute noch) anzutreffen: End-to-End Messungen:

- Laufzeit wird experimentell bestimmt
- „Sicherheitsaufschlag“ hinzuaddiert

\+ Einfach

– keine Garantie, dass der worst case wirklich beobachtet wurde

- Werte sind i.A. nicht zufällig verteilt



[3]

[3] [Wilhelm et al, 2008]

Statische WCET-Analyse

- Forderung nach einer oberen Schranke der Ausführungszeit führt auf das Halteproblem (Turing)

→ Generell nicht für alle Programme lösbar

- Prinzipiell könnte mit einen Modell der Hardware (des Prozessors) der gesamte Zustandsraum eines Programmes „durchexerziert“ werden
- Der längste dabei auftretende Pfad liefert die WCET
- Probleme:
  - Explosion des Zustandsraums
  - Komplexität des Modells
  - Nur off-line möglich
  - Berücksichtigen von Unterbrechungen?
  - Ggf. zu pessimistisch
- Bei einigen Sicherheitsstandards (z.B. EASA) verbindlich

Dynamische WCET-Analyse

- Argumentation: Moderne Hardware zu komplex für Modellierung
  - Fehlerhaftes Modell wird bei statischer Analyse nicht erkannt, liefert aber ggf. falsche Ergebnisse
- Experimenteller Ansatz
  - Messen der WCET von Basic Blocks
  - Anhand des Kontrollflussgraphen alle möglichen Reihenfolgen der Basic Blocks (Ausführungspfade) konstruieren
  - Längster Pfad liefert WCET
- Probleme:
  - Berücksichtigen von Unterbrechungen?
  - Cache(s)?

Cache: empirische Sicht (1)

Hochschule RheinMain

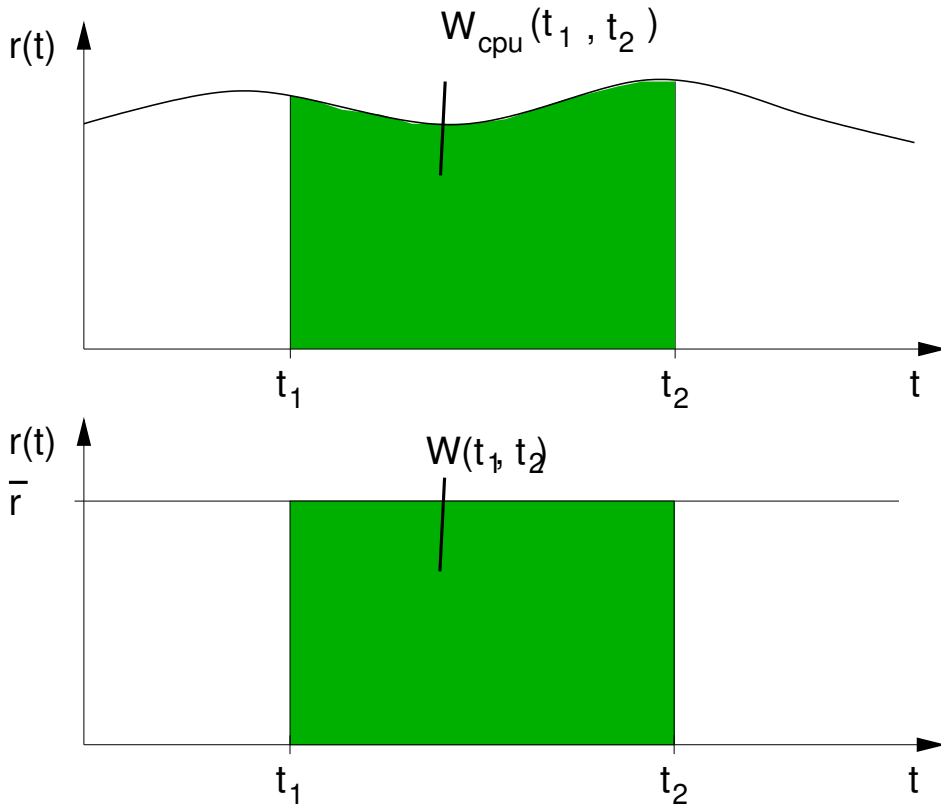
Prozessor führt mit einer Fortschrittsrate $r(t)$ Programmcode aus

- Rechenleistung, oder Fortschittsrate: $r(t)$
- Verrichtete Arbeit: $W_{cpu}(t)$

$$W_{cpu}(t_1, t_2) = \int_{t_1}^{t_2} r(\tau) d\tau$$

- Mittlere Rechenleistung (d.h. $r(t) = \bar{r}$):

$$W(t_1, t_2) = (t_2 - t_1) \cdot \bar{r}$$



Umschaltverluste

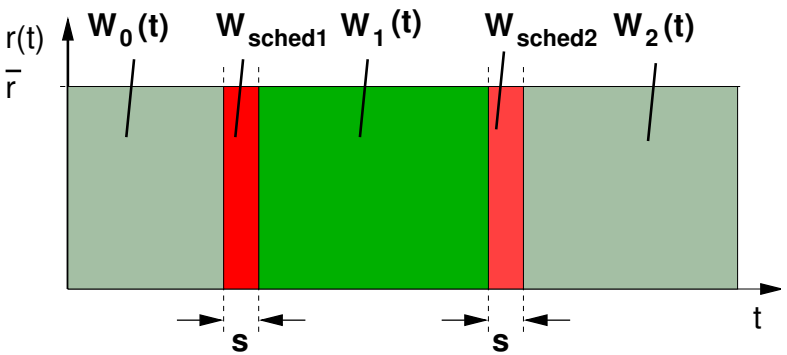
Hochschule RheinMain

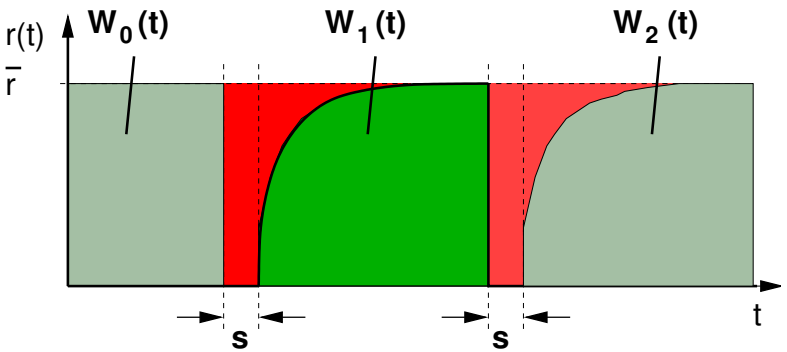
- Durch Prozesswechsel entstehen Verluste:
  - Zeitscheiben anderer Prozesse (kein Verlust)
  - Laufzeit des Schedulers (Verlust)
- Annahme: Scheduler-Laufzeit konstant ($= s$)

$\Rightarrow$ Verlust pro Scheduler-Ausführung:

$$W_s = s \cdot \bar{r}$$

- Bei Prozesswechsel[4]: weitere Verluste
- Verluste können Prozessen zugeordnet werden





[4] (N.B.: Prozesswechsel $\neq$ Scheduler-Aufruf)

Cache-bedingte Umschaltkosten

Hochschule RheinMain

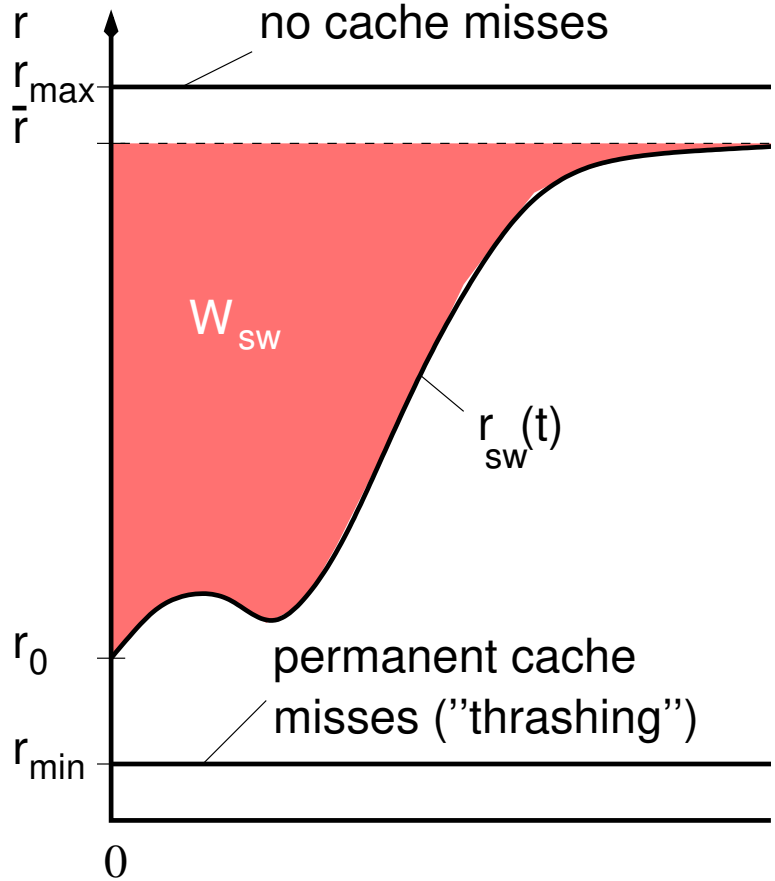
Bedingt durch Cache/TLB-Misses wird der Prozessor vorübergehend „langsamer“

⇒ Kosten je Prozesswechsel:

$$W_{sw}(t) = t \cdot \bar{r} - \int_0^t r_{sw}(\tau) d\tau$$

- Relativer Verlust:

$$O_{sw}(t) = 1 - \frac{1}{t} \int_0^t \frac{r_{sw}(\tau)}{\bar{r}} d\tau$$



Prozesswechsel bei $t = 0$

Nutzlastanteil

Hochschule RheinMain

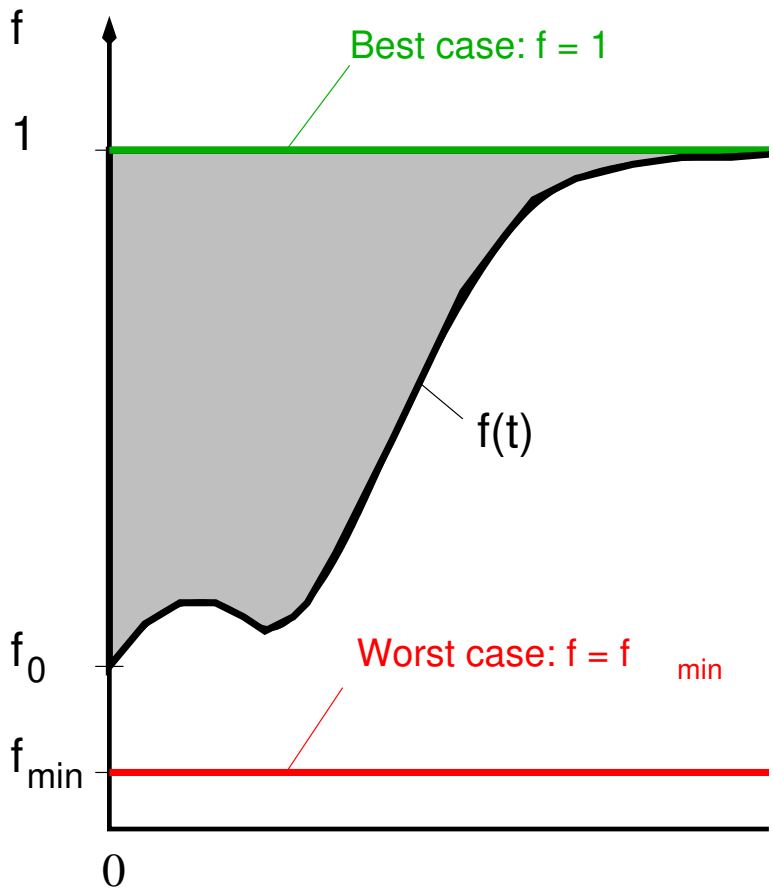
- Verhältnis von genutzter zu gesamter Rechenleistung:

$$f(t) := \frac{r_{sw}(t)}{\bar{r}}$$

- Damit:

$$O_{sw}(t) = 1 - \frac{1}{t} \int_0^t f(\tau) d\tau$$

- $f(t)$ ist i.A. unbekannt, aber
  - best case: $f(t) = 1$
  - worst case: $f(t) = f_{min} > 0$
  - Real: dazwischen ..



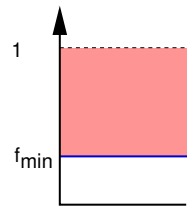
Prozesswechsel bei $t = 0$

Nutzlastanteil approximieren

Hochschule RheinMain

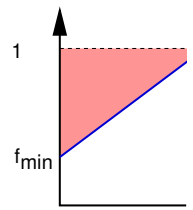
- Annahme eines durch eine Funktion $f(t)$ beschreibbaren Verlaufes des Nutzlastanteils, z.B.:
  - "cache flooding" (worst case) ...

$$f_{flood}(t) = \begin{cases} f_{min}, & 0 \leq t < t_s \\ 1, & t \geq t_s \end{cases}$$



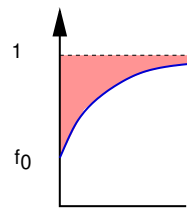
  - ... oder linear ...

$$f_{lin}(t) = \begin{cases} f_{min} + \frac{1-f_{min}}{t_s} \cdot t, & 0 \leq t < t_s \\ 1, & t \geq t_s \end{cases}$$



  - ... oder Exponentialfunktion ...

$$f_{exp}(t) = 1 + (f_0 - 1) \cdot e^{-kt}$$



  - ....

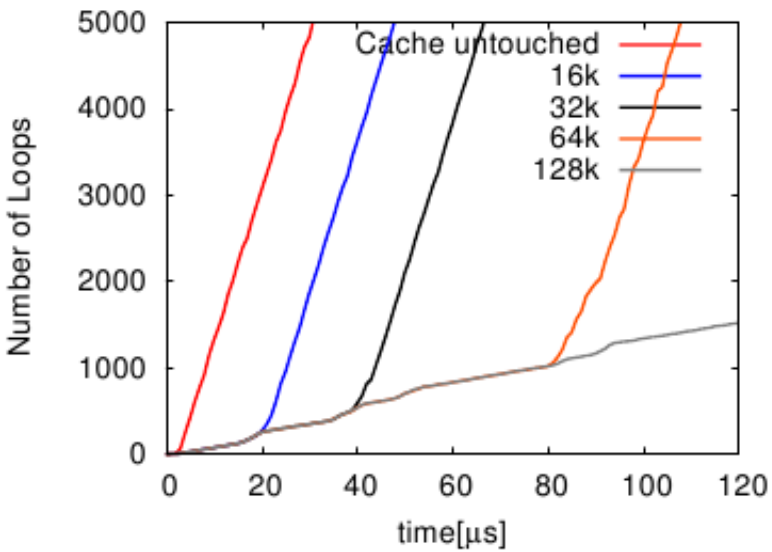
$\Rightarrow$ Task-bezogene Schätzung der Verluste möglich

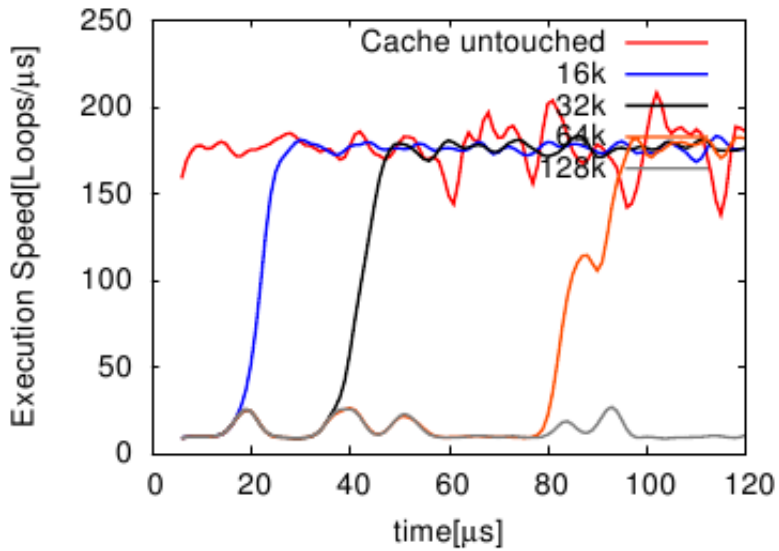
- Wahl von $f(t)$ entsprechend Anforderungen an die Task, z.B.
  - „harte“ Echtzeit $\rightarrow$ wähle $f_{flood}(t)$
  - „weiche“ Echtzeit $\rightarrow$ wähle $f_{lin}(t)$ or $f_{exp}(t)$

„Kalibrieren“ der Nutzlastfunktion

Hochschule RheinMain

- Ermitteln konkreter Werte für (t_s, f_{min}, etc.)
  - Caches in definierten Zustand bringen (invalidate / read-fill / write-fill)
  - Intensive Lese- oder Schreibzugriffe auf variable großen, zuvor ungecacheten Scpeicherbereich („working set“)
  - Zeit für vorgegebene (variable) Anzahl Zugriffe messen
- Plattform: Intel Celeron 2,5 GHz



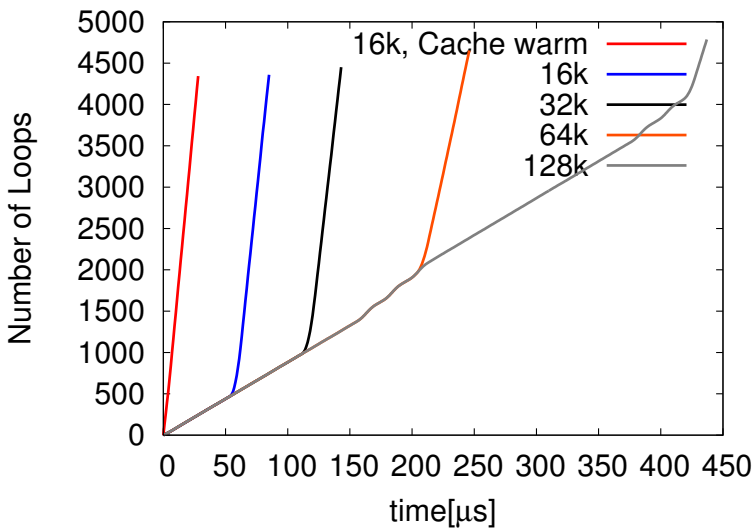


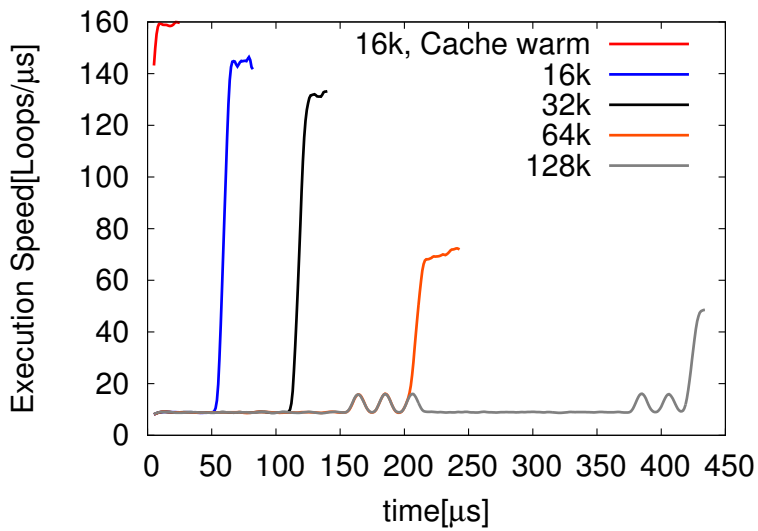
- Gemessen: Arbeit
- Daraus Abgeleitet: Leistung

„Kalibrieren“ der Nutzlastfunktion

Hochschule RheinMain

- Ermitteln konkreter Werte für (t_s, f_{min}, etc.)
  - Caches in definierten Zustand bringen (invalidate / read-fill / write-fill)
  - Intensive Lese- oder Schreibzugriffe auf variable großen, zuvor ungecacheten Scpeicherbereich („working set“)
  - Zeit für vorgegebene (variable) Anzahl Zugriffe messen
- Plattform: i.MX6 (SabreLight)



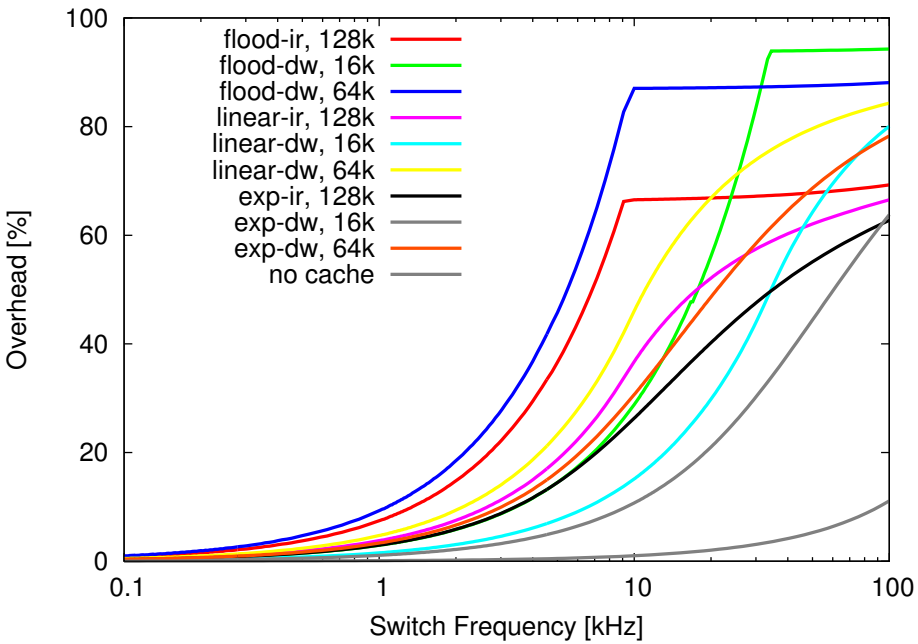


- Gemessen: Arbeit
- Daraus Abgeleitet: Leistung

Anwendung der Ergebnisse

Hochschule RheinMain

- Aus den Messungen konkrete Werte für f_{min} und t_s ableitbar
- Damit: Simulation eines Zeitscheiben Schedulers (TSS)
- Cachesimulationen: linear, exponential und flood (worst case)
- Parametrisiert mit f_{min}, t_s wie bei i.MX6 SabreLight gemessen



⇒ Zeigt Beziehung zwischen Zeitscheibe ↔ Umschaltkosten

⇒ Bei SabreLight: Umschaltfrequenzen über ≈ 2-3 kHz führen zu inakzeptablem Overhead!

⇒ Celeron: schon ab ≈ 1 kHz inakzeptabel

- 2 Tasks, 50% Zeitscheiben, Angenommene Scheduler Ausführungszeit: $1\mu s$