Übersicht

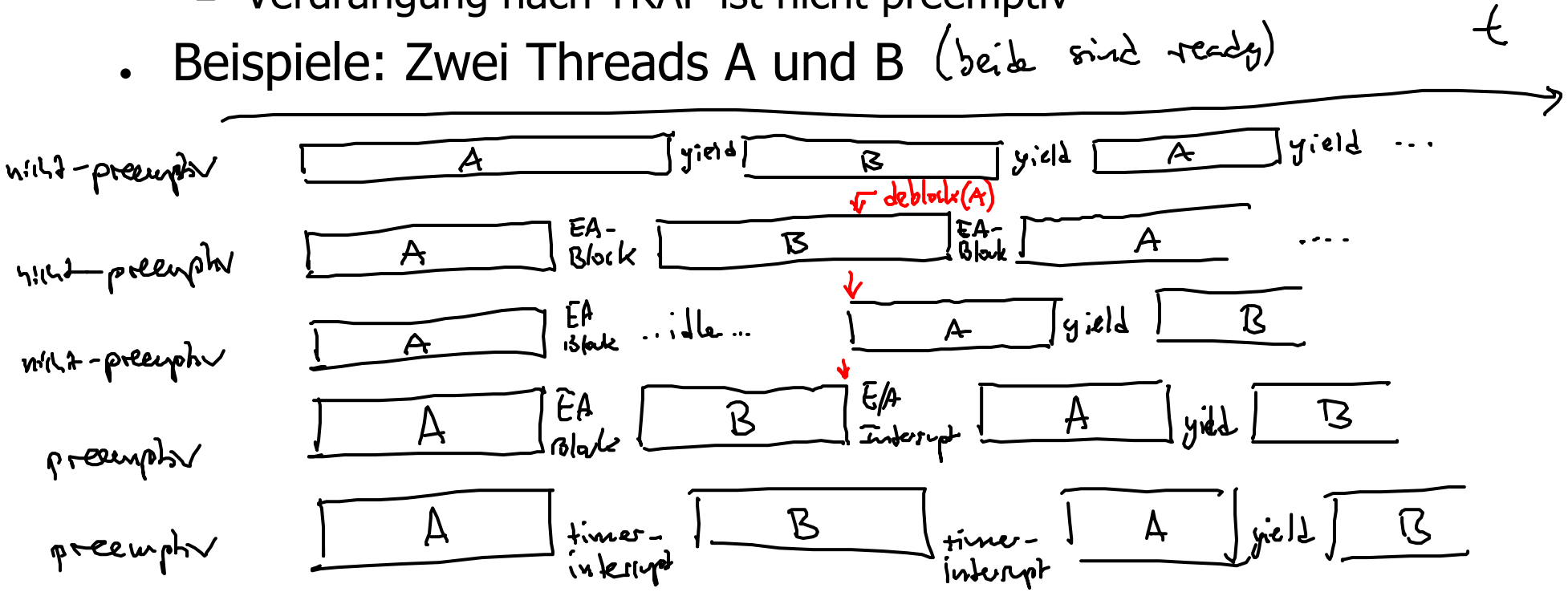
- Einführung
- Anforderungen und Thread-Typen
- Zustandsmodelle
- *Monoprozessor-Scheduling*
  - Einfache Scheduling-Verfahren: FCFS, SJF, RR usw.
- Echtzeit-Scheduling
- Multiprozessor-Scheduling
- Implementierungsaspekte

Qualitätskriterien

- Wahl der Scheduling-Strategie hängt nicht nur von der Betriebsform ab
- Qualitätsmaße, nach denen man entscheiden kann:
  - CPU-Auslastung: Wie lange pro Zeiteinheit führt die CPU Anwendungsinstruktionen aus und wie lange ist sie untätig?
  - Durchsatz: Wieviele Aufträge werden pro Zeiteinheit fertiggestellt?
  - Turnaround: Wie lange dauert es, bis ein Thread im Durchschnitt wieder auf den Prozessor kommt? (Zeit zwischen zwei `assign`-Übergängen)
  - Wartezeit: Wie lange verweilt ein Thread im Durchschnitt in der Bereit-Liste?
  - Antwortzeit: Wie lange dauert die Reaktion eines Threads auf eine Benutzereingabe?
  - Realzeit: Hält das System gegebene Echtzeitschranken ein?
- Qualitätsmaße zur Bewertung der folgenden Strategien

Einschub: preemptiv vs. nicht-preemptiv

- preemptives vs. nicht-preemptives Scheduling:
  - Ein Scheduling-Verfahren heisst preemptiv, wenn ein laufender Thread durch eine asynchrone Unterbrechung vom Prozessor genommen wird (Nehmer/Sturm, S. 104)
  - Verdrängung nach TRAP ist nicht preemptiv
- Beispiele: Zwei Threads A und B



First-Come First-Served (FCFS)

- Nicht-preemptives Schedulingverfahren
- Prozessorzuteilung in der Reihenfolge des Auftragseingangs
- Einfache schlangenbasierte Implementierung
  - Jeder neue Auftrag (Thread) wird "hinten" eingereiht
  - Bei **`assign`** wird jeweils der "vorderste" Thread auf dem Prozessor genommen
- Kontextwechsel findet nur statt, wenn ...
  - ... ein Thread eine blockierende Betriebssystemfunktion (z.B. E/A) aufruft, oder
    - Neu-Einreihung des Threads nach Beendigung der Blockierung
  - ... ein Thread den Prozessor freiwillig abgibt
    - Sofortige Neu-Einreihung des Threads
- FCFS erzielt hohe CPU-Auslastung
  - Alle anderen Kriterien werden nicht optimiert

Beispiel FCFS

- Drei Threads mit Ausführungszeiten ihrer CPU-Bursts:

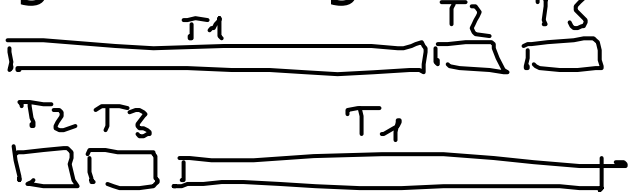
  T1 : 24 ms
  T2 : 3 ms
  T3 : 3 ms

  alle anfangs lauffähig

- Zwei verschiedene Ausführungsreihenfolgen:

  Reihenfolge 1: T1, T2, T3

  Reihenfolge 2: T2, T3, T1



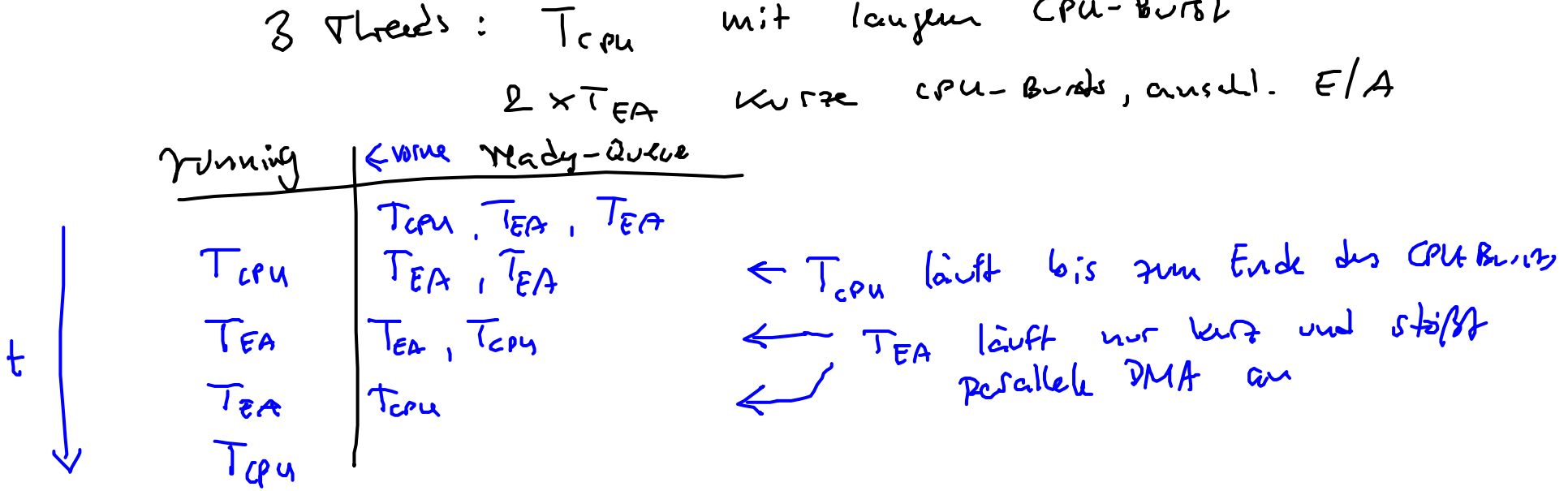
- Mittlere Wartezeit hat hohe Varianz:

  Wartezeit für Reihenfolge 1: (0 + 24 + 27) / 3 = 17 ms

  Reihenfolge 2: (0 + 3 + 6) / 3 = 3 ms

Konvoi-Effekt

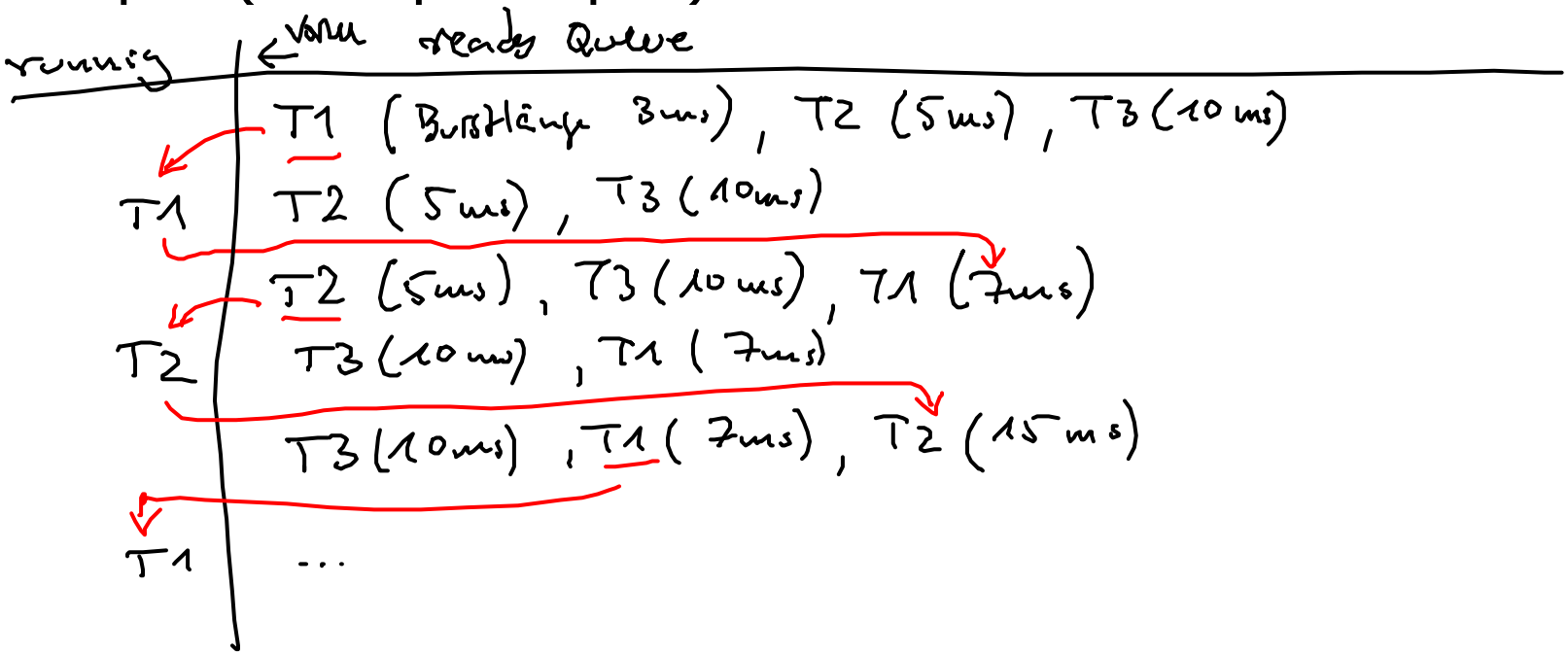
- Threads laufen "im Konvoi" über den Prozessor
  - Überholungen kaum möglich, aber manchmal sinnvoll
  - Auslastung der CPU hoch, Auslastung des Gesamtsystems niedrig
- Beispiel: Kombination von Threads mit langen CPU-Bursts und Threads mit viel E/A



- Parallelität des Systems wird kaum genutzt

Shortest Job First (SJF)

- Prozessorzuteilung in der Reihenfolge wachsender CPU-Burst-Länge
  - Thread mit dem kürzesten CPU-Burst erhält den Prozessor
  - Bei mehreren Threads mit gleichlangen CPU-Bursts wird FCFS angewendet
- Beispiel (nicht-preemptiv):



Diskussion SJF

- Verfahren vermeidet den Konvoi-Effekt
- Verfahren minimiert die durchschnittliche Wartezeit
- Verfahren ist nur bedingt realisierbar:
  - Länge des nächsten CPU-Bursts oft unbekannt
  - Länge des nächsten CPU-Bursts wird in der Praxis approximiert
- Approximationsverfahren:
  - Bilde einen gewichteten Mittelwert aus letzter Schätzung und dem gemessenen Wert des letzten CPU-Bursts

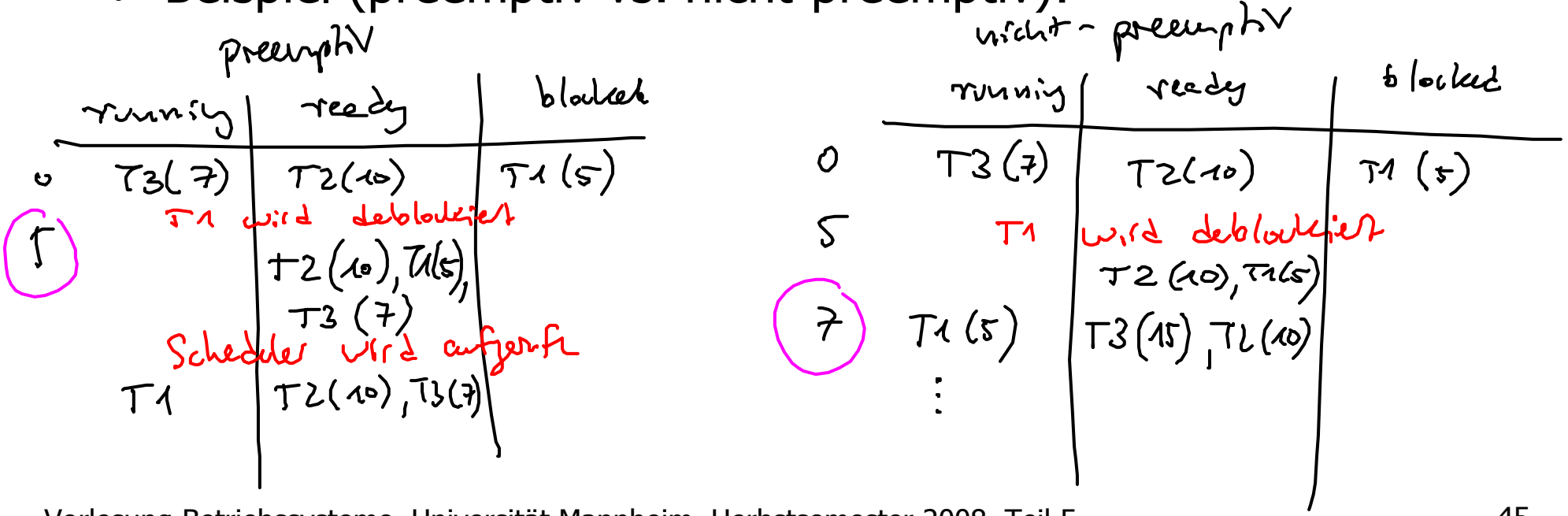
start mit der ersten Schätzung: $Burst_{geschätzt,\emptyset}$ Dann berechne jeweils:

$$Burst_{geschätzt,n+1} = \alpha \cdot Burst_{gemessen,n} + (1-\alpha)\, Burst_{geschätzt,n}$$

  - Faktor α zwischen 0 und 1 erlaubt die Gewichtung der Vergangenheit
    - Werte nahe 1 legen hohes Gewicht auf die gemessene Vergangenheit (schlecht bei hoher Varianz)

SJF preemptiv

- Bisher betrachtet: nicht-preemptives SJF
  - Bei Deblockierung eines Threads wird dieser unbeachtet hinten in die bereit-Schlange eingestellt
- Bei preemptiven SJF wird bei der Deblockierung der Scheduler erneut aufgerufen
  - Verbessert die Antwortzeit
- Beispiel (preemptiv vs. nicht-preemptiv):



Round Robin (RR)

- Ziel: Gleichmäßige Aufteilung der Rechenzeit auf alle bereiten Threads
- Sehr verbreitete preemptive Schedulingstrategie:
  - Jedem Thread wird ein definiertes Zeitquantum (Zeitscheibe) zugewiesen
  - Thread bleibt so lange auf dem Prozessor bis
    - der Thread eine blockierende Systemoperation aufruft, oder
    - die eigene Zeitscheibe abläuft.
  - Rechenbereite Threads werden ansonsten mit einer FIFO-Schlange verwaltet
  - Im `assign` wird jeweils der Kopf der Warteschlange auf den Prozessor genommen
- RR kann als preemptives FCFS angesehen werden

Diskussion

- RR ist Grundlage der meisten Multi-Tasking-Betriebssysteme
  - Rechenleistung wird möglichst gleichmäßig auf alle laufenden Anwendungen verteilt
  - Optimal, wenn alle Threads ihr Zeitquantum voll ausschöpfen
  - E/A-intensive Threads werden benachteiligt (bleiben jeweils nur kurz auf dem Prozessor)
- Wahl der Zeitscheibe ist kritisch
  - Bei zu kleinem Wert:
    - Threads kommen auf den Prozessor und fliegen gleich wieder runter
    - Caches werden nie ganz warm
  - Bei zu großem Wert degeneriert RR zu FCFS
    - Reaktionszeit wird sehr hoch
- Typische Werte der Zeitscheibe: 10 bis 20 ms

Prioritätsbasiertes Scheduling

- Threads werden gemäß einer zugewiesenen Priorität ausgewählt
  - Prioritätswerte werden aus einem gegebenen Ganzzahl-Intervall ausgewählt
  - Meistens bedeuten kleine Nummern hohe Priorität
- Statische Verfahren:
  - Priorität eines Threads wird zum Erzeugungszeitpunkt festgelegt
  - Wert wird nicht weiter verändert
  - Statisches prioritätsbasiertes Scheduling wird meist beim Echtzeitbetrieb gewählt
- Dynamische Verfahren:
  - Priorität wird zur Laufzeit dynamisch angepasst

Instanzen und Varianten

- Man kann fast alle Schedulung-Verfahren als prioritätsbasierte Verfahren ansehen
  - Beispiel: SJF kann als dynamisches prioritätsbasiertes Verfahren angesehen werden:
    - Priorität umgekehrt proportional zur geschätzten Länge des nächsten CPU-Bursts
- Bei einem Kontextwechsel wird immer der Thread mit höchster Priorität ausgewählt
  - Nicht-preemptive Verfahren: Kontextwechsel erst bei Aufruf einer blockierenden Systemoperation oder freiwilliger Aufgabe des Prozessors
  - Preemptive Verfahren: Kontextwechsel auch dann, wenn ein Thread mit höherer Priorität bereit wird
    - Durch Deblockierung aufgrund beendeter E/A, Erzeugung eines neuen Threads mit höherer Priorität oder timer interrupt

Aushungerung (Starvation)

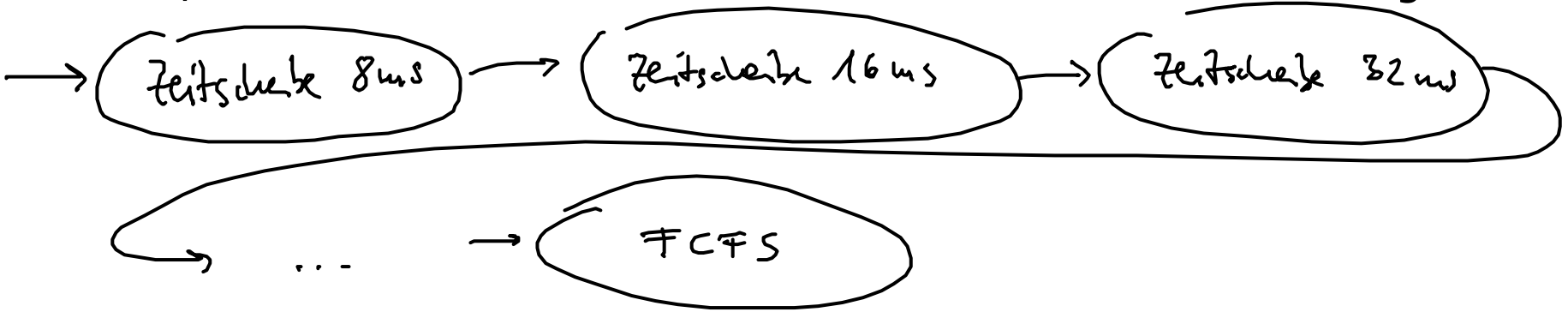
- Bei reinen prioritätsbasierten Verfahren besteht die Gefahr der Aushungerung von Threads
  - Aushungerung (starvation): Ein Thread kommt nie auf den Prozessor
  - Grund: Threads mit höherer Priorität schöpfen die komplette Rechenleistung des Prozessors aus
- Abhilfe:
  - Bei statischen Verfahren schwierig
    - Neue Prioritätszuordnung und Neustart des Systems
  - Bei dynamischen Verfahren einfacher
    - Aging-Techniken: Prioritäten werden mit der Zeit niedriger, um auch Threads mit kleiner Priorität schlussendlich den Prozessor zuzuteilen

Multilevel-Scheduling

- Kombination mehrerer Scheduling-Verfahren miteinander
  - Gängige Kombination beispielsweise:
    - Zuerst alle dialogorientierten oder zeitkritischen Threads gemäß FCFS
    - Dann alle unkritischen Threads nach RR
- Implementierung durch mehrere Bereit-Listen
  - Benötigt zusätzliches Auswahlverfahren zwischen den Listen
    - Üblich: Priorisierung der Listen oder auch ein Zeitmultiplexen der Listen
- Scheduling-Variante: Feedback-Scheduling
  - Der aktuelle Systemzustand (insbesondere die Vergangenheit eines Threads) geht in die Scheduling-Entscheidung ein
  - Beispiel: Aging = prioritätsbasiertes Feedback-Scheduling

Multilevel-Feedback-Scheduling

- Kombination von Multilevel- und Feedback-Scheduling
- Gängig ist folgende Variante:
  - Es gibt Bereit-Schlangen mit zunehmender Zeitscheibe
  - Am Ende folgt eine FCFS-Schlange
  - Threads, die vor Ablauf ihrer Zeitscheibe (d.h. innerhalb eines CPU-Bursts) vom Prozessor fliegen, kommen in eine Schlange mit längerer Zeitscheibe
  - Threads werden beginnend bei Schlange mit kürzester Zeitscheibe bedient
  - E/A-intensive Threads werden nicht mehr so stark benachteiligt



Übersicht

- Einführung
- Anforderungen und Thread-Typen
- Zustandsmodelle
- Monoprozessor-Scheduling
- *Echtzeit-Scheduling*
- Multiprozessor-Scheduling
- Implementierungsaspekte

Echtzeit-Scheduling

- Beim Echtzeit-Scheduling (real time scheduling) geht es wesentlich um die Einhaltung von Zeitvorgaben
- Unterscheidung in strikte und schwache Echtzeitsysteme:
  - Strikte Echtzeitsysteme (hard real time): Verletzung einer Zeitvorgabe ist katastrophal
    - Unter allen Umständen zu vermeiden (Verlust an Menschenleben, hohe Sachschäden, etc.)
    - Beispiel: Industriesteuerungen, Flugzeugsteuerungen
  - Schwache Echtzeitsysteme (soft real time): Verletzung einer Zeitvorgabe ist lästig, aber nicht katastrophal
    - Beispiel: Multimedia-Übertragungen, bei denen leichte Verzerrungen oder Verzögerungen auftreten können
- Zeitvorgaben müssen konkret für jede Anwendung spezifiziert werden

Formalisierung von Zeitvorgaben

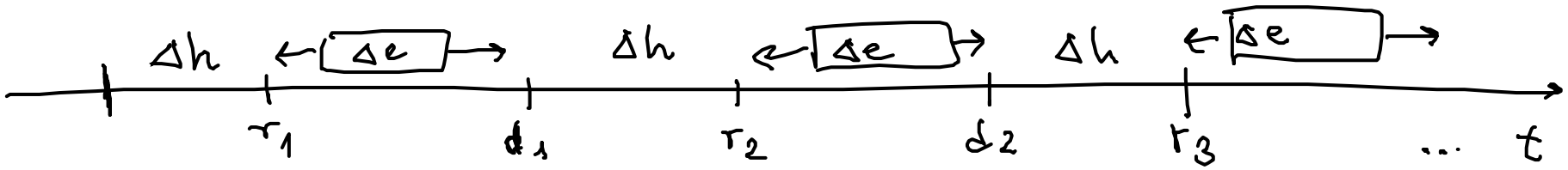
- Echtzeitanwendung besteht aus einer Menge an Aktivitäten
- Jede Aktivität ist durch drei Kenngrößen charakterisiert:
  - Bereitzeit (r, ready time): frühstmöglicher Ausführungsbeginn der Aktivität
  - Frist (d, deadline): spätester Zeitpunkt für die Beendigung einer Aktivität
  - Ausführungszeit (Δ e, execution time): worst-case-Abschätzung für das zur vollständigen Ausführung der Aktivität notwendige Zeitintervall

- Bestimmung dieser drei Werte ist für das Echtzeit-Scheduling essentiell

Periodische Aktivitäten

- Normalerweise sollte gelten: $\Delta e \leq d\text{-}r$
  - Worst-case-Abschätzung von Δe in der Praxis schwierig und führt zudem zu einer schlechten Gesamtauslastung des Systems
  - Es muss aber jederzeit genügend Rechenzeit für alle Aktivitäten zur Verfügung stehen
- Periodische Aktivitäten haben ähnliche Kenngrößen:
  - Periode (Δp): Frequenz der Aktivität
  - Phase (Δh): Versatz des Ausführungsbeginns relativ zum Anfang jeder Periode
    - Definiert die Bereitzeit innerhalb der Periode
  - Ausführungszeit (Δe)

$$\Delta p = d_i - r_i$$



Berechnung weiterer Zeiten

- Aus den Angaben zu Δp, Δh und Δe kann man Bereitzeit und Frist der k-ten Ausführung der periodischen Aktivität ermitteln:
  - $r_k = k \cdot \Delta h + (k-1) \Delta p$
  - $d_k = k \cdot \Delta h + k \cdot \Delta p$
- Jede Aktivität wird vom Scheduler zu einem bestimmten Zeitpunkt s_k (Startzeitpunkt) eingeplant
  - Aus s_k und Δe kann man die späteste Abschlusszeit c_k berechnen
  - Bei nicht-preemptiven Verfahren: $c_k = s_k + \Delta e$
  - Bei preemptiven Verfahren: $c_k \geq s_k + \Delta e$
    - Unter Umständen wurden andere Aktivitäten zwischengeschoben

Zeitgesteuerte Systeme

- Zeitgesteuerte Ausführung: Alle Scheduling-Entscheidungen werden durch das Voranschreiten von Zeit ausgelöst
  - Scheduler reagiert ausschließlich auf eintreffende Timer Interrupts
  - Kleinste Periode muss ein ganzzahliges Vielfaches der Frequenz des Timer-Interrupts sein
- Timer-Interrupt ist der „Tick", der das System voranschreiten lässt
  - Alle Zugriffe auf externe Geräte finden direkt statt
    - Sensoren werden beispielsweise periodisch explizit abgefragt
    - keine Interrupt-gesteuerte Geräteverwaltung notwendig
  - Zugriffskonflikte wurden im voraus aufgelöst
- Sehr konservatives Systemmodell für Echtzeitsysteme

Ereignisgesteuerte Systeme

- Schedulingentscheidungen basieren auf internen und externen Ereignissen
  - Ereignisse werden in der Regel durch Interrupts kommuniziert
  - Externe Ereignisse: Interrupts durch externe Geräte
  - Interne Ereignisse: Werden durch andere Prozesse ausgelöst, zum Beispiel durch Signale (softwaretechnisches Gegenstück zu Interrupts)
  - Scheduling-Verfahren, die ohne Timer-Interrupt auskommen, sind prinzipiell ereignisgesteuert
- Ereignisgesteuerte Systeme sind viel flexibler als zeitgesteuerte Systeme, aber schwerer vorhersagbar
  - Deswegen sind strikte Echtzeitsysteme meist zeitgesteuert

Statisches Offline-Scheduling

- Für eine gegebene strikte Echtzeitanwendung sind eine Menge von periodischen Aktivitäten mit Periode, Phase, Ausführungszeit gegeben
- In welcher Reihenfolge sollen die Aktivitäten abgearbeitet werden?
  - Alle möglichen Reihenfolgen zu betrachten, ist unmöglich (exponentieller Aufwand)
    - Insbesondere unmöglich zur Laufzeit
- Lösung: Scheduling-Reihenfolge offline berechnen und fest in das System hineingiessen
  - Scheduler muss jetzt nur noch in einer Tabelle nachschauen, wer als nächstes dran ist
  - Offline Scheduling meist in Verbindung mit zeitgesteuerten Systemen