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

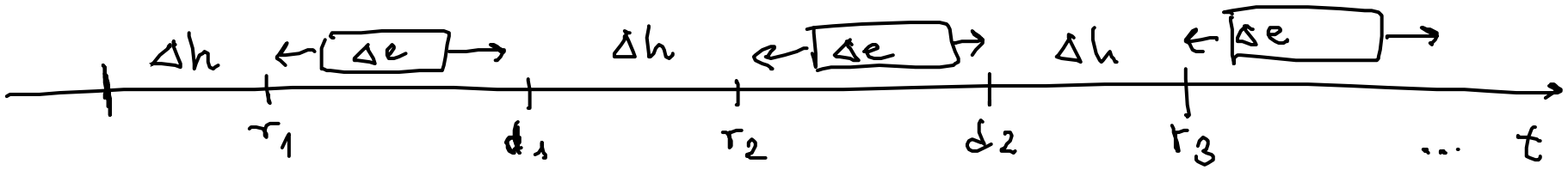
- Echtzeitanwendung besteht aus einer Menge an Aktivitäten
- Jede Aktivität ist durch drei Kenngrößen charakterisiert:
  - Bereitzeit (r, ready time): frühstmöglicher Ausführungsbeginn der Aktivität
  - Frist (d, deadline): spätester Zeitpunkt für die Beendigung einer Aktivität
  - Ausführungszeit (Δ 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

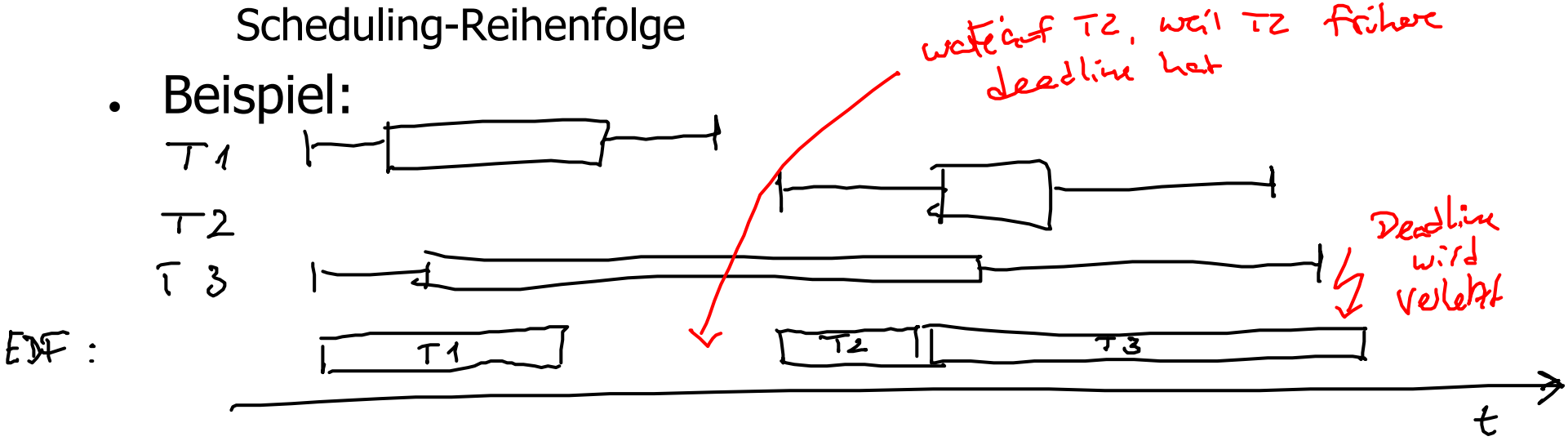
- Schedulingentscheidungen basieren auf internen und externen Ereignissen
  - Ereignisse werden in der Regel durch Interrupts kommuniziert
  - Externe Ereignisse: Interrupts durch externe Geräte
  - Interne Ereignisse: Werden durch andere Prozesse ausgelöst, zum Beispiel durch Signale (softwaretechnisches 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

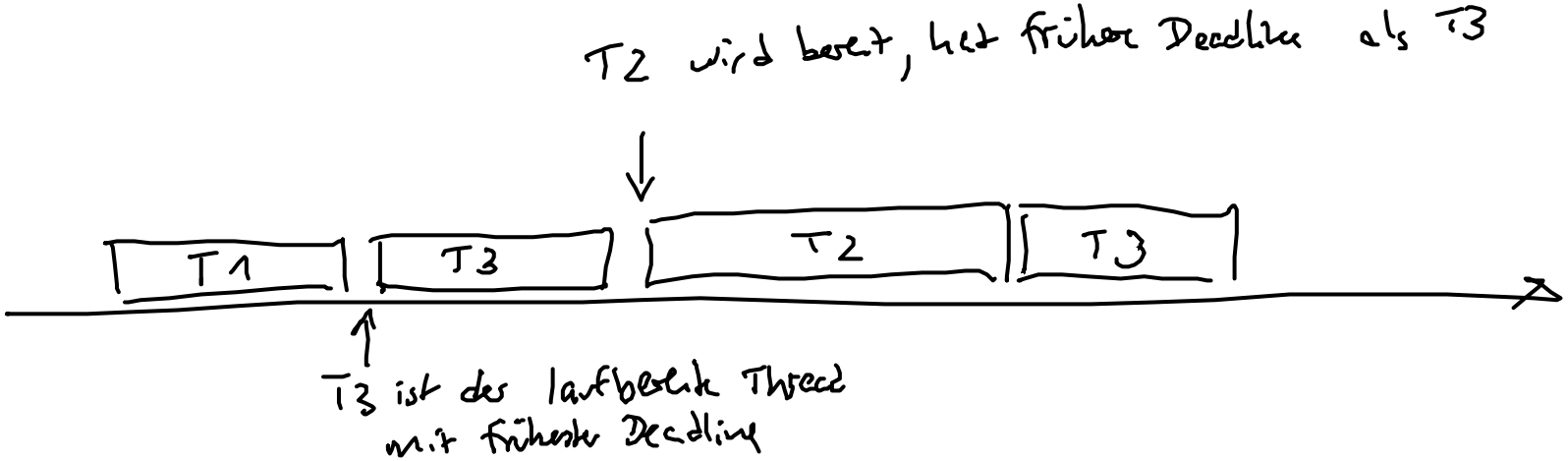
Earliest Deadline First (EDF)

- Analog zu SJF: Der Scheduler wählt immer die Aktivität mit der nächsten in der Zukunft liegenden Frist
  - Ein Prozessor bleibt untätig, solange die Bereitzeit des nächsten auszuführenden Threads noch nicht erreicht wurde
- Nicht-preemptive Variante:
  - Threads geben den Prozessor nur freiwillig oder aufgrund einer blockierenden Systemfunktion ab
  - Nicht-preemptive Variante findet nicht immer eine mögliche Scheduling-Reihenfolge
- Beispiel:



preemptives EDF

- Scheduler wird aktiv, sobald ein Thread mit näher in der Zukunft liegender Deadline rechenbereit wird
  - Beispiel: Ein Thread deblockiert nach einer E/A
- Preemptives EDF findet eine korrekte Abarbeitungsreihenfolge des Beispiels von eben:



- Man kann zeigen: preemptives EDF findet **immer** eine Abarbeitungsreihenfolge, die alle Deadlines einhält, wenn diese Reihenfolge existiert

Rate-Monotonic-Scheduling (RMS)

- preemptives, prioritätsbasiertes Scheduling-Verfahren für periodische strikte Echtzeitsysteme
  - Prioritäten werden in Abhängigkeit der Periode der einzelnen Aktivitäten vergeben
    - hohe Frequenz (= kleinste Periode) = höchste Priorität
    - geringste Frequenz (= größte Periode) = niedrigste Priorität
- Verfahren widerspricht der Intuition: Wichtigkeit hat eigentlich nichts mit der Periode zu tun sondern mit der nächsten Deadline
- Aber:
  - Zuordnung verzögert hochfrequente Aktivitäten minimal
  - Verfahren führt jedoch zu einer starken Zerstückelung der niederfrequenten Aktivitäten (diese werden oft unterbrochen zugunsten von Threads mit höherer Priorität)

Vorteile von RMS

- RMS ist relativ einfach handhabbar:
  - Übersetzung von Aktivitäten in Prioritäten ist direkt und einfach
  - Es gibt sogar ein Kriterium, mit dem man vorab bestimmen kann, ob ein gegebenes System seine Echtzeitvorgaben einhalten wird
- Im einfachsten Fall: Überprüfung der Gesamtbelastung
  - Gegeben n Aktivitäten mit Periode Δp und Ausführungszeit Δe
  - Gesamtbelastung $U = \sum_{k=1}^{n} \frac{\Delta e_k}{\Delta p_k}$
  - Kriterium: Falls $U < n \cdot \left(2^{\frac{1}{n}} - 1\right) \approx 0,693$

    gilt: RMS liefert immer eine funktionierende Ausführungsreihenfolge
- Falls Auslastung höher, muss man genauer hinschauen

Diskussion: Best Effort Scheduling

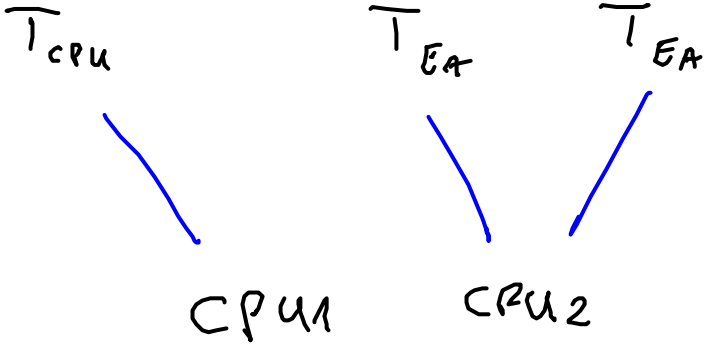
- EDF und RMS finden in der Praxis Anwendung im Bereich strikter Echtzeitsysteme
  - Nur dort lohnt sich der Aufwand für die schwierigen Worst-case-Abschätzungen der Ausführungszeiten
  - Dort werden Systeme auch nicht so häufig verändert
    - Jede Veränderung zieht eine neue Scheduling-Analyse des Gesamtsystems nach sich
- Im Bereich schwacher Echtzeitsysteme geht man von hinreichend leistungsfähiger Hardware aus
  - Man testet das System unter diversen künstlichen Lastbedingungen und hofft, dass das System auch unter realen Bedingungen die Echtzeitvorgaben mit hoher Wahrscheinlichkeit einhält
  - Das nennt man dann Best Effort Scheduling
  - Best Effort Scheduling liefert offensichtlich keine verbindlichen Garantien

Übersicht

- Einführung
- Anforderungen und Thread-Typen
- Zustandsmodelle
- Monoprozessor-Scheduling
- Echtzeit-Scheduling
- *Multiprozessor-Scheduling*
  - *Scheduling-Fragestellungen bei mehreren Prozessoren*
- Implementierungsaspekte

Multiprozessorsysteme

- Multiprozessorsysteme erlauben prinzipiell die echt parallele Ausführung mehrerer Threads
  - Wenn immer genügend Rechenlast in Form unabhängiger Threads bereit steht, vereinfacht dies die Scheduling-Problematik
  - Kenngrößen wie Antwortzeit, Durchsatz und die Einhaltung von Zeitschranken verbessern sich
- Bei unabhängigen Threads kann man jedes Monoprozessor-Scheduling-Verfahren auf Multiprozessoren übertragen
  - Oft werden sogar explizite Nachteile einzelner Verfahren aufgehoben
    - Beispiel: Konvoi-Effekt bei FCFS



Probleme

- Problem: Wirklich unabhängige Threads sind selten
  - Direkte Abhängigkeiten durch Synchronisation (siehe Kapitel 6)
  - Implizite Abhängigkeiten durch Zugriff auf gemeinsame Ressourcen (z.B. das gleiche E/A-Gerät)
  - Anzahl der wirklich rechenbereiten Threads unterschreitet oft die Anzahl der vorhandenen Prozessoren
  - Nominale Leistung eines Multiprozessorsystems wird häufig nicht voll ausgeschöpft
- Besonders schwierig: eng kooperierende Threads möglichst gleichzeitig ausführen
  - Ziel: hohen Parallelitätsgrad direkt in die Anwendung bringen
  - Dazu müssen kooperierende Threads explizit gekennzeichnet werden
  - Scheduler muss möglichst die ganze Thread-Gruppe gleichzeitig auf die Prozessoren bringen
    - Probleme wie bei Monoprozessorscheduling

Thread-Prozessor-Zuordnung

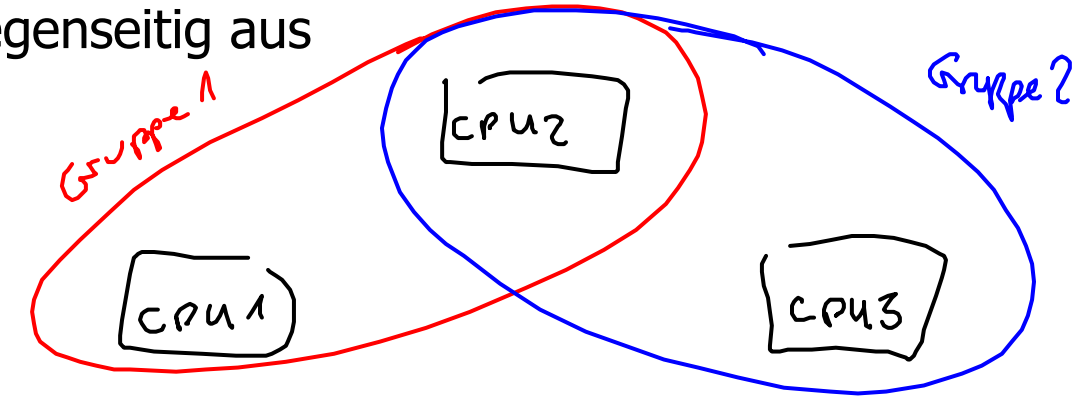
- Soll ein Thread immer auf denselben Prozessor kommen oder nicht?
  - Vorteile einer statischen Zuordnung: Warmlaufphase der Caches verkürzt sich
  - Nachteile: Scheduler ist in Auswahl eingeschränkt
- Bei dynamischer Zuordnung von Thread zu Prozessor spricht man von **symmetrischem Multiprozessorbetrieb**
  - Vorteil: Scheduler ist keinerlei Einschränkungen unterworfen
  - Nachteil: Nutzung von Kontextinformationen in Caches ist unrealistisch
- Bei dynamischer Zuordnung kann im Extremfall sogar eine Systemleistung erzielt werden, die schlechter als die eines Monoprozessors ist
  - Beispiel: Ein Thread, der von CPU zu CPU "springt" und dauernd kalte Caches vorfindet

Implementierung: Load Sharing

- Einfachste Form der Implementierung: Verwendung einer zentralen Bereit-Liste, aus der der Scheduler auswählt
  - Bei Kontextwechsel wird der jeweils nächste Thread auf den entsprechenden Prozessor genommen
  - Systemlast wird auf alle Prozessoren gleichmäßig verteilt (load sharing)
  - Da nur eine Liste vorhanden ist, kann prinzipiell auch jedes Verfahren für Monoprozessorscheduling verwendet werden
- Problem: Verwaltung der zentralen Liste
  - Als besonders geschützte Datenstruktur
    - Zugriff der Prozessoren muss wechselseitig ausgeschlossen geschehen
    - Ergibt Synchronisationsaufwand
  - Verwaltung auf einem dezidierten Prozessor
    - Ergibt hohen Kommunikationsaufwand zwischen den Prozessoren

Gruppen-Scheduling

- Kooperierende Threads werden durch eine zentrale Bereit-Liste nicht unterstützt
- Beim Gruppen-Scheduling werden eng kooperierende Threads als Einheit betrachtet
  - Thread-Gruppe wird nur zum Einsatz gebracht, wenn gleichzeitig genügend viele Prozessoren „frei" sind
  - Vorteil: Garantierte Parallelität in der Anwendung
  - Nachteil: Bei ungünstiger Verteilung von Thread-Anzahl und Prozessoranzahl können einzelne Prozessoren lange untätig bleiben
    - Extremes Beispiel: Zwei Thread-Gruppen, die für sich jeweils mehr als die Hälfte der Prozessoren benötigen, schliessen sich gegenseitig aus



Übersicht

- Einführung
- Anforderungen und Thread-Typen
- Zustandsmodelle
- Monoprozessor-Scheduling
- Echtzeit-Scheduling
- Multiprozessor-Scheduling
- *Implementierungsaspekte*
  - *Implementierung des Dispatchers für KL-Threads*
  - *Realisierung von UL-Thread-Packages*

Implementierung Dispatcher

- Threads werden wesentlich durch den Dispatcher implementiert
  - Dispatcher implementiert an einer einzigen Stelle im Betriebssystem die Zustandsübergangsoperationen für Threads
- Synchronisationsbedarf bei Zugriff auf gemeinsame Datenstrukturen
  - Manipulation der Prozesslisten muss wechselseitig ausgeschlossen geschehen
  - Auf Monoprozessorsystemen geht das am einfachsten durch Ausschalten der Interrupts während der Manipulation der Datenstrukturen
- Problem: Was ist der "Kontext" beim Kontextwechsel?
  - Insbesondere der PC muss erstmal gesucht werden (siehe Übung).

Globales Design des Dispatchers

- Was ist, wenn im `assign` die Bereit-Schlange leer ist?
  - Am einfachsten: Einführung eines Leerlaufprozesses, der immer bereit ist
  - Man muss aufpassen, dass dieser Prozess auch nur dran kommt, wenn kein anderer Prozess bereit ist
- Invariante: Es soll immer genau ein Prozess im Zustand rechnend sein
  - Daraus folgt: Nach Entzug des Prozessors durch den Dispatcher bei `resign` und `block` muss **zwingend** ein `assign` aufgerufen werden
  - Beim Hochfahren des Systems muss ein initialer Ur-Prozess angelegt werden, der alle weiteren Prozesse/Threads startet

Beispiel: Verwendung des Dispatchers

- Blockierendes `write(Data)` zum Schreiben von Daten auf Platte
  - Laufzeitfunktion (z.B. aus Bibliothek):

```
write(Adress a) {
  <Parameter a kommunizieren>
  TRAP(write); // System Call
}
```

  - Interrupt Handler für `TRAP(write)`:

```
syscall_write {
  <Adresse a herausfinden, Ziel d berechnen>
  <Schreiben von Adresse a nach Block d
      mittels DMA anstossen>
  block();
  assign();
}
```

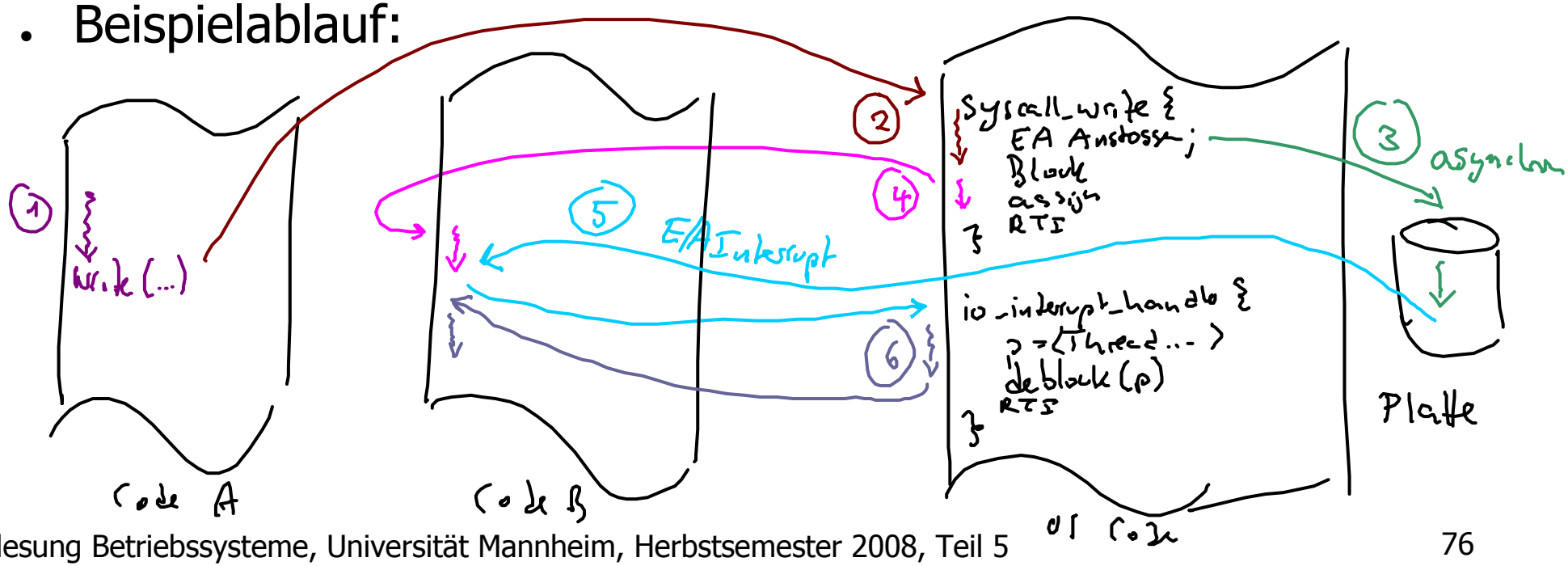
  - Fehlt nur noch der E/A-Interrupt-Handler, der den Prozess deblockiert

Beispiel (Fortsetzung)

- E/A-Interrupt-Handler:

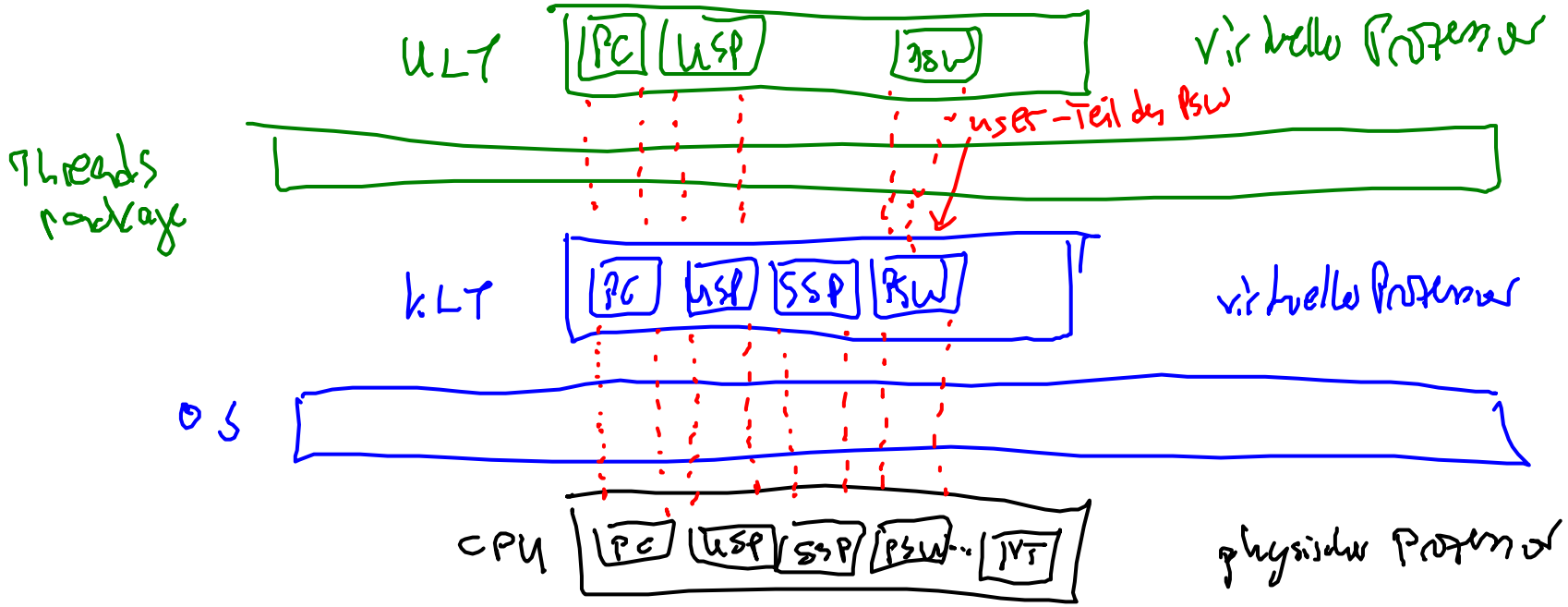
```
io_interrupt_handler {
  p = <Thread, zu dem die E/A gehörte>;
  deblock(p);
}
```

- Bei Bedarf kann nach dem `deblock` auch ein `resign;` `assign;` aufgerufen werden (für preemptives Scheduling)
- Beispielablauf:



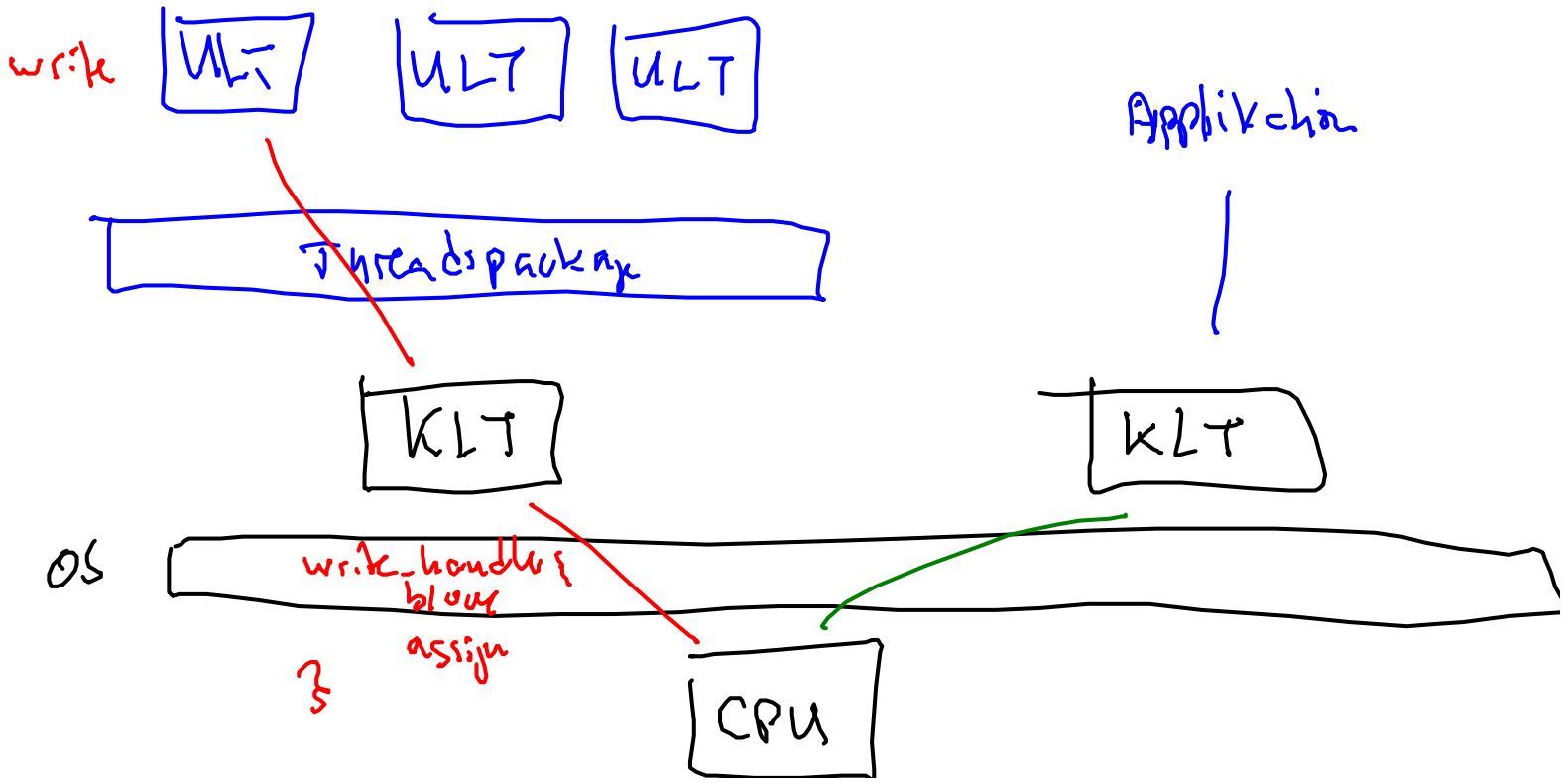
Realisierung UL-Threads

- Implementierung analog zum Kernel-Dispatcher
  - Realisierung als User-Prozess auf einem KL-Thread ist wie die Realisierung des Dispatchers für Monoprozessormaschinen
- Dispatcher-Operationen müssen den **User Level Kontext** speichern und laden
  - Kontext der virtuellen Prozessoren nimmt nach "oben" hin ab



Virtueller Monoprozessor

- Problem: Wenn ein UL-Thread eine blockierende E/A aufruft, steht das ganze Package



Diskussion

- Erste Lösung:
  - Umsetzung blockierender E/A auf nicht-blockierende E/A (steht manchmal zur Verfügung)
  - Interrupt-Mechanismus auf User-Ebene (z.B. UNIX-Signals)
- Zweite Lösung:
  - Verwenden von mehreren KL-Threads
  - Das ist wie die Implementierung von KL-Threads auf mehreren CPUs (siehe Übung)

Zusammenfassung

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

Ausblick

- Gliederung der Vorlesung:
  1. Einführung und Formalia
  2. Auf was baut die Systemsoftware auf?
     Hardware-Grundlagen
  3. Was wollen wir eigentlich haben?
     Laufzeitunterstützung aus Anwendersicht
  4. Verwaltung von Speicher: Virtueller Speicher
  5. Verwaltung von Rechenzeit: Virtuelle Prozessoren (Threads)
  6. Synchronisation paralleler Aktivitäten auf dem Rechner
  7. Implementierungsaspekte