Betriebssysteme
Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 5: Threads

Felix C. Freiling
Lehrstuhl für Praktische Informatik 1
Universität Mannheim

Motivation

- Adressraum und Prozessorzeit sind die beiden fundamentalen Abstraktionen, die Systemsoftware zur Verfügung stellt
- Adressraum für einen Prozess wurde im vergangenen Kapitel implementiert
  - Prozess = Adressraum + Thread
- Jetzt geht es um die Implementierung und das Management von Threads
  - Threads als zentrale Abstraktion von Prozessorzeit

Positionsbestimmung

- 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)**
     - **2 Wochen!**
  6. Synchronisation paralleler Aktivitäten auf dem Rechner
  7. Implementierungsaspekte

Übersicht

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

Threads

- Threads (Kontrollflüsse) beschreiben sequentielle Aktivitäten in einem System
  - Threads sind Abstraktionen von Prozessorzeit
  - Threads können als **virtuelle Prozessoren** betrachtet werden
  - Realisierung von Threads basiert auf dem Multiplexen mehrerer Threads auf einem physischen Prozessor
  - Threads besitzen immer ein zugeordnetes sequentielles Programm
  - Programmausführung beginnt an einer vordefinierten Adresse
- Mit der Einrichtung eines Adressraums wird immer ein (erster) Thread für diesen Adressraum generiert
  - Klassischer Prozess = Adressraum + 1 Thread
  - In vielen Fällen reicht das auch aus
  - Bei komplexen Anwendungen kann es sinnvoll sein, mehrere Threads in einem Adressraum zu generieren

Mehrere Threads (Teams)

- Warum macht es Sinn, mehrere Threads in einem Adressraum einzusetzen?
  - Wenn ein Thread an einer E/A wartet, kann ein anderer Thread andere Aufgaben der Anwendung bearbeiten
  - Bei Multiprozessorsystemen kann man eine Anwendungsaufgabe parallel auf mehreren Prozessoren bearbeiten
  - Man kann "anwendungsnäher" Programmieren, wenn die Anwendung selbst inhärent parallele Aktivitäten beinhaltet
- Ergibt ein nebenläufiges Verarbeitungsmodell
  - Nebenläufigkeit (*concurrency*) = Vorhandensein von mehreren unabhängigen gleichzeitigen Aktivitäten
  - Verwandte Begriffe: paralleles System und verteiltes System
    - paralleles System = "abhängige" gleichzeitige Aktivitäten
    - verteiltes System = geographische Verteiltheit

Natürliche Nebenläufigkeit

- Bei entsprechenden Anwendungen ist das nebenläufige Verarbeitungsmodell ein sehr natürliches Programmiermodell
  - Mit der Zunahme an Thread-Realisierungen in Betriebssystemen und Laufzeitpaketen nimmt das Modell an Bedeutung zu
- Beispiel: Wetterbeobachtung
  - Anwendung mit graphischer Bedienoberfläche
  - Zwei unabhängige Handlungsstränge:
    1. Berechnungen zur Messwertverarbeitung und Wettervorhersage, die kontinuierlich weiterlaufen
    2. Graphische Bedienoberfläche, mit der Daten ausgelesen und visualisiert werden können
  - Jeder Kontrollfluss für sich ist streng sequentiell
  - Wie programmiert man diese Anwendung am besten?

Zwei Kontrollflüsse

- Es gibt zwei nahezu unabhängige Kontrollflüsse, die für sich einfach (sequentiell) programmiert werden könnten
- Kontrollfluss 1: Anwendung - Dauerrechnen!

```
Compute() {
  while (1) {
    ... // Dauerrechnen
  }
}
Thread 1: Compute();
```

- Kontrollfluss 2: Bedienoberfläche - Reaktion auf Benutzereingaben

```
GUI() {
  while (1) {
    e = ReceiveEvent();
    ProcessEvent(e);
  }
}
Thread 2: GUI();
```

Sequentielle Programmierung

- Wenn man keine Threads hätte, wie würde man dann die Anwendung schreiben?
  - Zerteilung der Funktion **`Compute()`** in kleinere abgeschlossene Teile **`ComputeStep()`**
  - Regelmäßige Überprüfung, ob eine Benutzereingabe vorliegt
- Adaptiertes Beispiel:

```
while (1) {
  ComputeStep();       // ein Berechnungsschritt
  if (QueryEvent()) { // Ereignis angekommen?
    e = ReceiveEvent();
    ProcessEvent(e);
  }
}
```

Bemerkungen

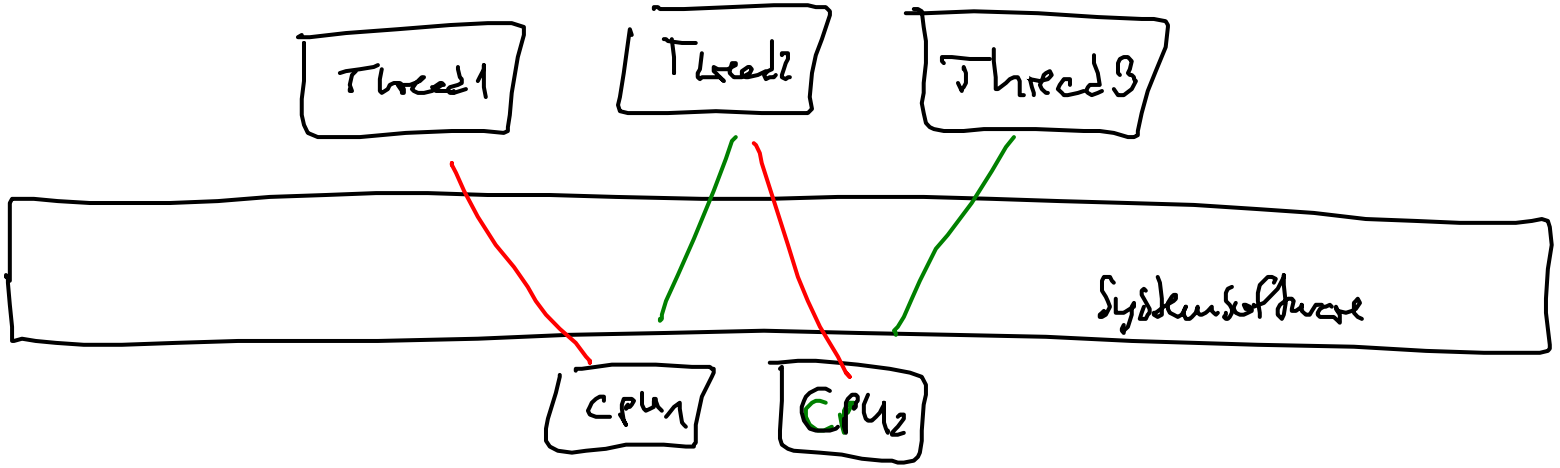
- Die Berechnung muss in kleinere Teilschritte zerlegbar sein
  - In vielen Fällen unnatürlich, wenn nicht sogar unmöglich
- Die Berechnung wird so periodisch auch dann unterbrochen, wenn keine Benutzereingabe vorliegt
  - Unterbrechung kann man sehr kurz machen, aber sie benötigt trotzdem einige Prozessorzyklen
- Reaktionszeit auf Benutzereingaben ist durch die Programmstruktur vorgegeben
  - Hängt von der Bearbeitungsdauer von `ComputeStep()` ab
  - Je länger die Bearbeitungsdauer, desto effizienter die Berechnung, aber desto langsamer die Reaktion
- Funktionen wie `QueryEvent()` dürfen auf keinen Fall blockieren, sonst steht die gesamte Berechung
- Programmstruktur wird schwer lesbar und wartbar

Vorteile von Threads

- Threads schaffen die Möglichkeit selbst auf einem Monoprozessorsystem eine unbeschränkte Zahl an virtuellen Prozessoren zu generieren
  - Man kann die Anwendungen auf diese virtuellen Prozessoren verteilen
  - Threads schaffen so die Möglichkeit, von der konkreten Prozessorzahl zu abstrahieren
- Bei der Anwendungsprogrammierung ist man nicht mehr an das sequentielle Ausführungsmodell gebunden
  - Sollte nach Möglichkeit inhärente Nebenläufigkeit einer Anwendung auch entsprechend ausprogrammieren
- Nicht nur softwaretechnische Vorteile
  - Auch die Leistungsfähigkeit eines Systems kann sich verbessern
  - Bereits auf Monoprozessormaschinen kann man Blockadezeiten einzelner Threads in anderen Threads ausnutzen

Virtuelle und physische Prozessoren

- Thread = virtueller Prozessor
- Team = mehrere Threads = virtueller Multiprozessor
- Im Idealfall steht jedem Thread ein eigener physischer Prozessor zur Verfügung
  - Feste 1:1-Zuordnung zwischen Thread und Prozessor
- Normalerweise existieren mehr Threads als physische Prozessoren
  - Systemsoftware muss die physischen Prozessoren im Zeitmultiplexbetrieb auf die virtuellen Prozessoren verteilen



Zeitmultiplexbetrieb

- Beispiel: ein physischer Prozessor, zwei virtuelle Prozessoren (Threads)
  - Systemsoftware weist abwechselnd beiden Threads den Prozessor zu
- Wechsel von einem zum anderen Thread: **Kontextwechsel**
  - Ablauf des aktuellen Threads wird unterbrochen
  - Prozessorkontext wird gesichert (Mehrzweckregister, Stackpointer, etc.)
  - Kontext des anderen Threads wird auf den Prozessor geladen
  - Fortführung des Threads an der zuvor gesicherten Adresse
- Wenn mehr als ein Thread als potentieller Nachfolger zur Verfügung steht, muss eine Auswahlentscheidung getroffen werden
  - Aufgabe des sogenannten Schedulers
  - Es existieren viele verschiedene Scheduling-Strategien

Übersicht

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

Thread-Konzepte

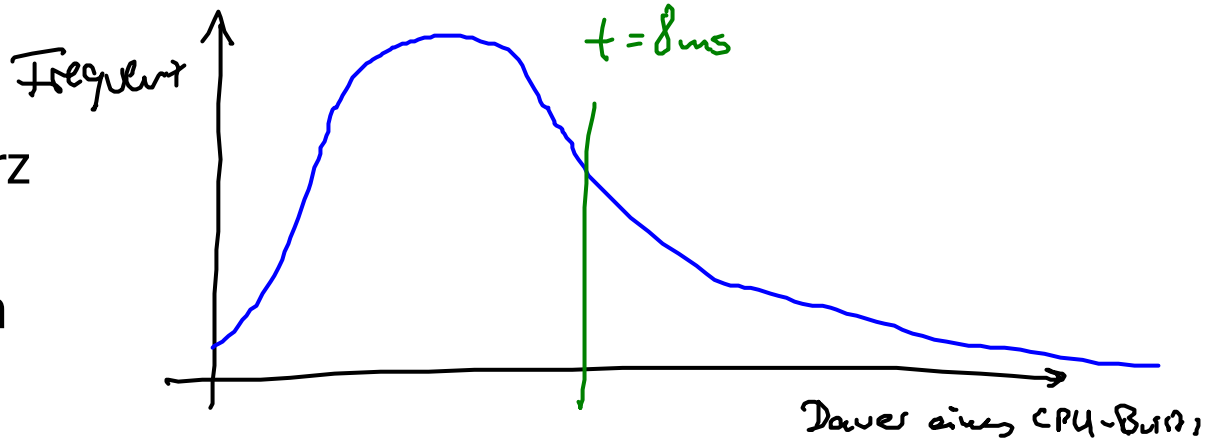
- Minimal sollte ein Betriebssystem folgende Funktionalität zur Verfügung stellen (einfaches Thread-Konzept):
  - Erzeugung eines neuen Adressraums mit einem einzelnen Thread
  - Zuweisung eines Programmes an den Thread
- Optimal sind leistungsfähige Thread-Konzepte:
  - Erzeugung neuer Threads zur Laufzeit
  - Zuweisung unterschiedlicher Programme an die verschiedenen Threads
- Realisierung meistens als Laufzeitbibliothek und/oder direkt im Betriebssystemkern
  - Wir werden in der Übung später eine kleine Thread-Bibliothek implementieren

Konkreter Nutzen von Threads

- Im Normalfall konkurrieren mehrere Threads um einen einzelnen Prozessor
  - Annahme: jeder Thread benötigt k Zeiteinheiten und läuft jeweils bis zum Ende durch
    - Erster Thread kommt sofort dran, zweiter Thread nach k Zeit usw.
    - Durchschnittliche Reaktionszeit: $\frac{1}{n} \sum_{i=1}^{n} (i-1) \cdot k = \frac{n-1}{2} \cdot k$
- In der Praxis hat man jedoch den Wechsel zwischen CPU- und I/O-Bursts
  - Threads in einem I/O-Burst werden dem Prozessor entzogen, es findet ein Kontextwechsel statt
  - Wenn E/A komplett asynchron abgearbeitet wird, ergibt sich eine mittlere Reaktionszeit von: $\frac{n-1}{2} t_{Burst}$

Vergleich

- Im ersten Fall: Verzögerung ist proportional zur Gesamtlaufzeit eines Threads
- Im zweiten Fall: Verzögerung ist proportional zur Länge eines CPU-Bursts
  - CPU-Bursts sind in der Regel sehr kurz
  - Beachtliches Maß an Nebenläufigkeit auch bei einem Prozessor erreichbar



- Unterbrechung eines Threads bei E/A ist also ein probates Mittel um die Reaktionszeit des Systems zu verbessern
  - Erst bei hoher Thread-Zahl fällt die Nebenläufigkeit negativ auf
  - Unterbrechung wegen E/A ist für den Scheduler eine wichtige Entscheidungsgrundlage für die Auswahl des nächsten Threads

CPU-Monopolisierung

- Wenn ein Thread auf dem physischen Prozessor läuft, dann läuft das Betriebssystem nicht
- Betriebssystem kann auf verschiedene Arten die Kontrolle zurückgewinnen
  1. Wenn der Thread eine blockierende E/A-Operation aufruft
  2. Durch freiwillige Abgabe des Prozessors durch den Thread
  3. Nach Eintreffen eines asynchronen Hardware-Interrupts
- Theoretisch könnte eine Anwendung durch Vermeidung aller drei Fälle das Ausführungsmonopol auf dem Prozessor erlangen
  - Beispiel: eine fehlerhafte Anwendung landet in einer Endlosschleife ohne E/A
- Systemsoftware verwendet Fall 3, um Monopolisierung zu vermeiden

Timer-Interrupts

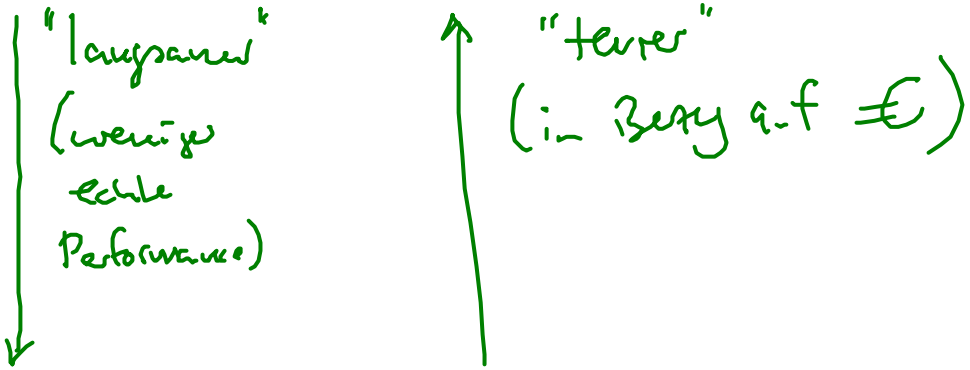
- Zusätzliche Hardware nötig: Timer-Baustein
  - Schaltkreis mit eingebauter Uhr
  - Löst periodisch einen Timer-Interrupt beim Prozessor aus
  - Periode kann konfiguriert werden
- Ebenfalls notwendig: (nicht-vertrauenswürdige) Anwendungen dürfen nur im User Mode laufen
  - Ausblenden der Timer-Interrupts bzw. Umkonfigurierung des Timers sind privilegierte Operationen
  - Manipulation der Interrupts sind nur mittels TRAP-Mechanismus möglich
  - Betriebssystem hat jeweils selbst wieder die Kontrolle und kann dem aktuellen Thread den Prozessor entziehen
- **preemptives Scheduling** = "ungewollter" Entzug des Prozessors durch das Betriebssystem
  - Fall 3: asynchrone Unterbrechung (timer interrupt, I/O interrupt, ...)

Schedulingziel

- Bei jedem Kontextwechsel muss der Scheduler erneut auswählen, welcher Thread als nächstes den Prozessor zugeteilt bekommt
- Auswahl des nächsten Prozesses hängt von der Betriebsform (Art des Einsatzes) des Rechners ab:
  - Dialogbetrieb: Schnelle Reaktion wichtig, d.h. unterbrochene Threads schnell wieder auswählen
  - Stapelbetrieb (Batchbetrieb): Hohe Prozessorauslastung wichtig, nur Threads auswählen, die gleich (und viel) rechnen wollen
  - Echtzeitbetrieb: Wähle die Threads so aus, dass jeder seine maximale Laufzeit einhalten kann
  - Hintergrundbetrieb: Egal, welcher Thread ausgewählt wird, Hauptsache alle werden am Ende oft genug bedient
- In der Praxis kommen Mischformen dieser Betriebsarten zum Einsatz

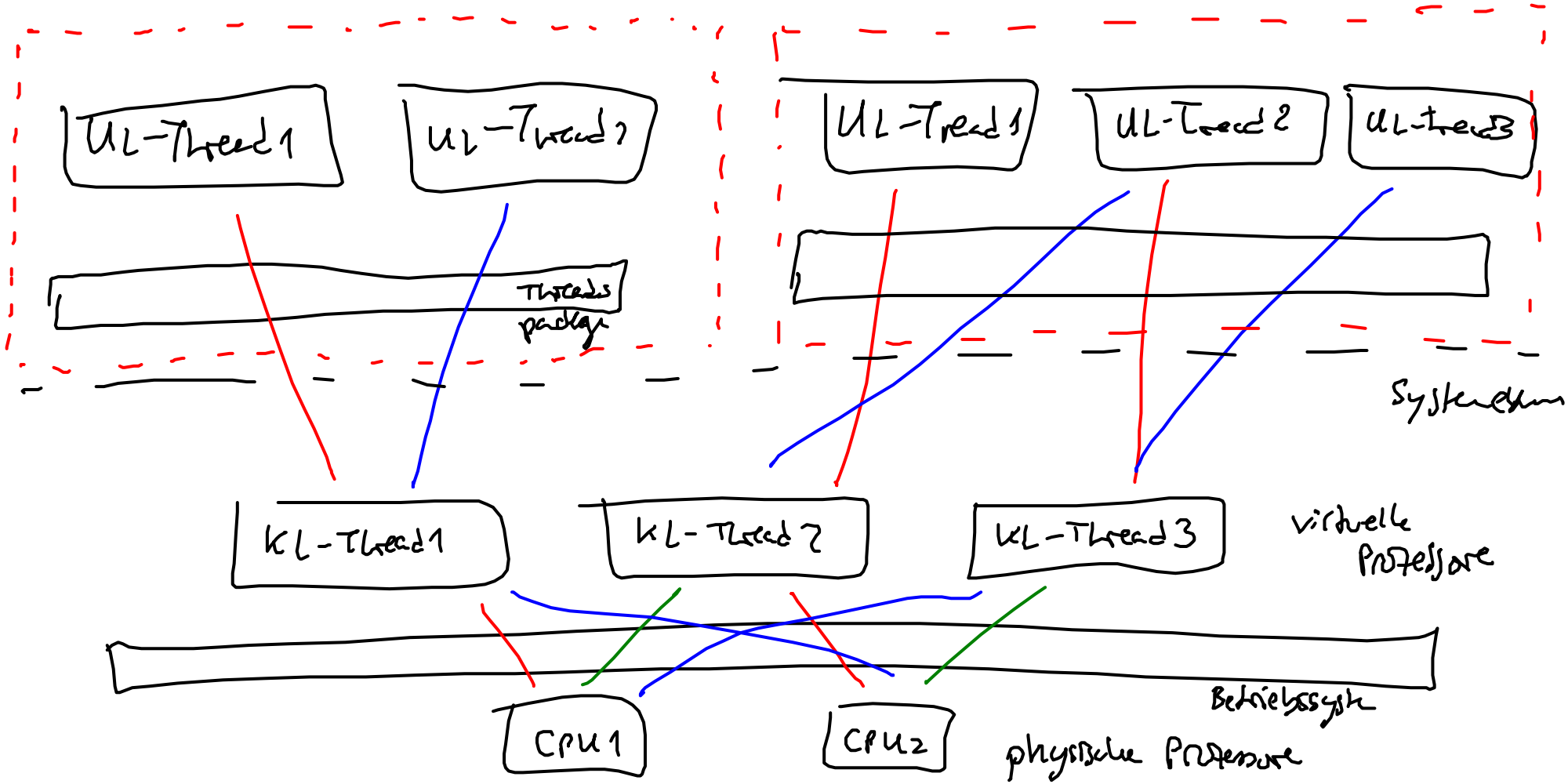
Thread-Typen

- Man unterscheiden abhängig von der Realisierungsform zwei Arten von Threads:
  - Kernel-Level-Threads (KL-Threads)
  - User-Level-Threads (UL-Threads)
- Die "klassischen" Threads (d.h. die Threads von Prozessen) sind KL-Threads
  - KL-Threads sind virtuelle Prozessoren auf physischen Prozessoren
  - UL-Threads sind virtuelle Prozessoren auf KL-Threads
- Prozessorhierarchie (analog zur Speicherhierarchie):
  - physische Prozessoren
  - KL-Threads
  - UL-Threads
  - UL-Threads
  - ...



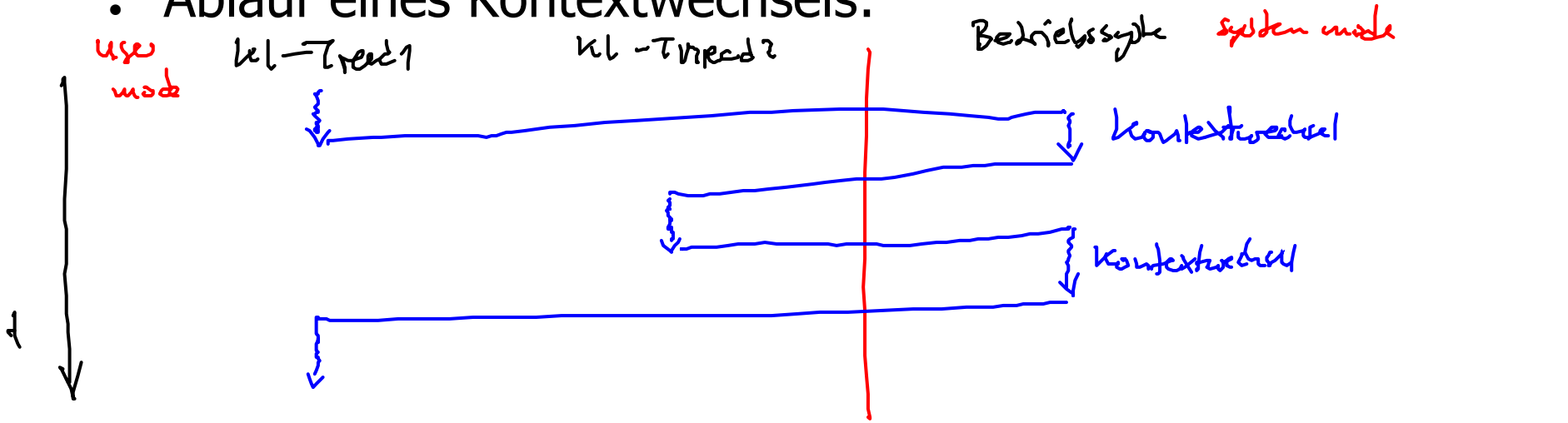
"Prozessorhierarchie"

- Team aus mehreren KL-Threads = virtueller Multiprozessor
  - Techniken sind auf unterschiedlichen Ebenen wiederverwendbar



Kernel-Level-Threads

- KL-Threads werden im System Mode verwaltet
  - Übergabe der Kontrolle von einem KL-Threads zum nächsten erfordert einen Aufruf des Betriebssystemkerns
- Ablauf eines Kontextwechsels:



- Kontextwechsel bei KL-Threads hat hohe Kosten:
  - TRAP ist sehr teuer
  - Bei Adressraumwechsel zusätzlich kalte Caches (z.B. TLB)
- KL-Threads deshalb auch *schwergewichtige* Threads

User-Level-Threads

- UL-Threads werden im User Mode verwaltet
  - Übergabe der Kontrolle von einem zum nächsten UL-Thread geschieht vollständig im User Mode (und deshalb sehr schnell)
  - UL-Threads deshalb auch *leichtgewichtige* Threads
- UL-Threads laufen auf KL-Threads
  - Bei einem KL-Thread: Analogie zu Monoprozessorsystem
  - Bei mehreren KL-Threads: Analogie zu Multiprozessorsystem
- Implementierung des Schedulers vollständig im User Space
  - Realisierung derselben Funktionalität, die auch im Betriebssystemkern für KL-Threads existiert
    - Kopie im User Space
  - Gewöhnlich als Laufzeitbibliothek zur Verfügung gestellt
    - UL-Thread-Package
- Wir **denken** vorerst **nur an KL-Threads**…

Bemerkung: Coroutinen

- Spezialfall von UL-Threads sind die sogenannten Coroutinen
  - Coroutinen sind eine Verallgemeinerung von Unterroutinen
  - Unterroutinen haben eine hierarchische Aufrufreihenfolge
    - A ruft B auf, B ruft C auf usw.
  - Coroutinen sind gleichberechtigt, ohne Hierarchie
    - A ruft B auf, B ruft wieder A auf
- Kontextwechsel wird in der Regel durch eine explizite Operation des Threads angestossen
  - Operation SWITCH(x): überträgt die Kontrolle an Coroutine x
  - Schedulingreihenfolge durch die Anwendung vorgegeben

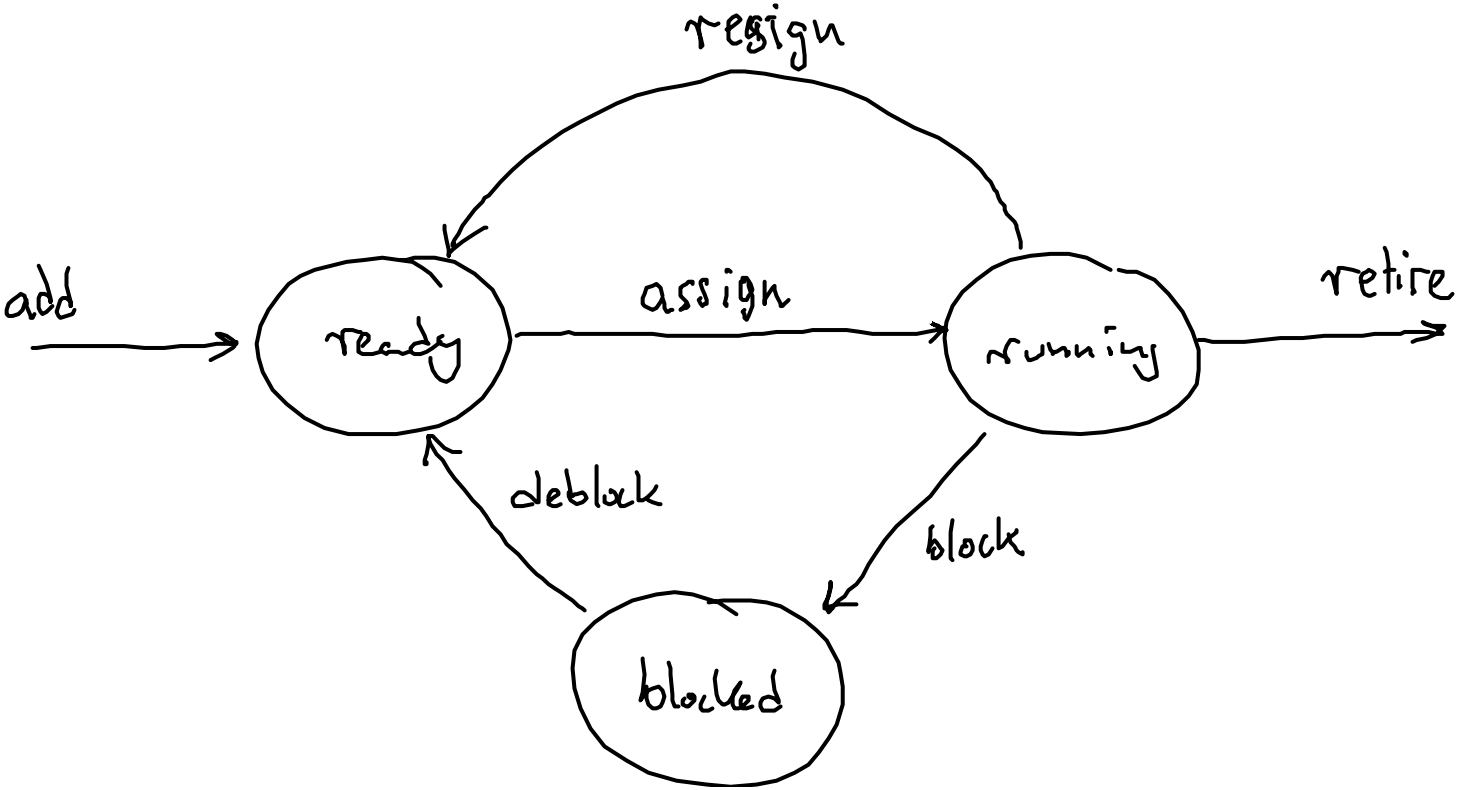
Übersicht

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

Zustände von Threads

- Der Scheduler muss eine Entscheidung treffen, welcher Thread als nächstes auf den Prozessor kommt
  - Zentrales Entscheidungsmerkmal: Der Zustand des Threads
- Einfaches Zustandsmodell unterscheidet drei Zustände:
  - Rechnend (running): Threads in diesem Zustand sind im Besitz des Prozessors
    - Bei mehreren Prozessoren gibt es mehrere Threads in diesem Zustand
  - Blockiert (blocked): Threads in diesem Zustand warten auf ein bestimmtes Ereignis, z.B. die Beendigung einer E/A-Operation
    - Es können jederzeit beliebig viele Threads in diesem Zustand sein
  - Bereit (ready): Threads in diesem Zustand sind potentiell ausführbar, haben zur Zeit aber leider nicht den Prozessor
    - Auch hier kann es jederzeit beliebig viele Threads in diesem Zustand geben

Zustandsübergänge eines Threads

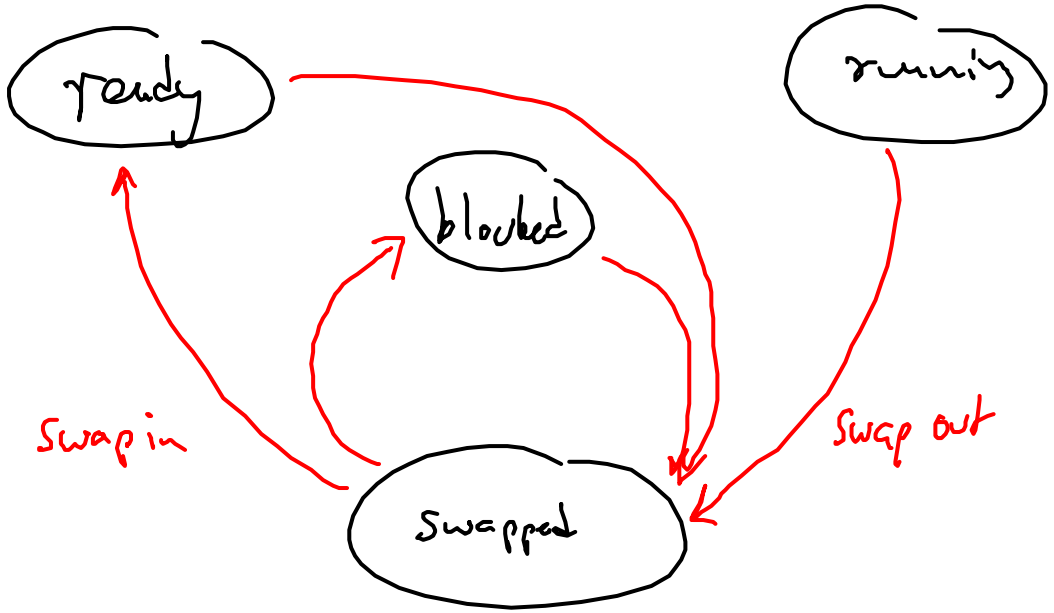


Bemerkungen

- Übergänge zwischen den Zuständen:
  - **`add`**: ein neuer Thread wird dynamisch erzeugt und in die Menge der bereiten Threads aufgenommen
  - **`assign`**: als Folge eines Kontextwechsels wird dem Thread der Prozessor zugeteilt
  - **`block`**: als Folge einer blockierenden Operation (z.B. E/A) oder eines Seitenfehlers wird dem Thread der Prozessor entzogen
  - **`deblock`**: der Grund der Blockierung eines Threads entfällt, der Thread wird wieder bereit zum rechnen und bewirbt sich erneut um den Prozessor
  - **`resign`**: ein laufender Thread wird der Prozessor vorzeitig entzogen, z.B. aufgrund eines Timer-Interrupts, der Thread bewirbt sich erneut um den Prozessor
  - **`retire`**: ein aktuell laufender Thread terminiert und gibt freiwillig den Prozessor ab

Zustände mit Swapping

- Wenn aufgrund von Speichermangel ein kompletter Prozess (Threads mit Adressraum) auf Externspeicher ausgelagert wird, hat man einen zusätzlichen Zustand:
  - ausgelagert (swapped)
- Zusätzliche Zustandsübergänge:
  - swap in, swap out



Threadkontrollblock (TCB)

- Der Zustand eines Threads wird bei den meisten Betriebssystemen in einem Threadkontrollblock beschrieben (Thread Control Block, TCB)
  - Manchmal auch Process Control Block (PCB) genannt
- Der TCB enthält alle für die Threadverwaltung notwendigen Informationen:
  - Eindeutige Identifikation (TID = Thread Identifier)
  - Speicherplatz zur Sicherung des Prozessorkontexts
  - ggf. Wartegrundes bei blockierten Threads
  - Adressrauminformationen, z.B. Verweis auf die Seitentabelle
  - Sonstige Zustandsinformationen und Statistiken für das Scheduling

Implementierung

- TCBs für alle Threads liegen in einer globalen Tabelle (Threadtabelle, früher Prozesstabelle)
- Betriebssystem organisiert Zustände durch verkettete Listen:
  - **Bereit-Liste**: alle Threads, die im bereit-Zustand sind
  - **Blockiert-Liste**: alle Threads, die blockiert sind
  - **Ausgelagert-Liste**: alle Threads, die ausgelagert sind
  - **Rechnend**: Zeiger auf den einen rechnenden Prozess
    - Bei Mehrprozessormaschinen gibt es entsprechend mehrere Zeiger (rechnend1, rechnend2, ...)
- Listen kann man auch als Warteschlangen auffassen
  - Man spricht dann von *ready queue* usw.

Dispatcher und Scheduler

- Die Durchführung der Zustandsübergänge übernimmt der **Dispatcher**
  - Modul des Betriebssystems, das die Zustandsübergangsfunktionen zur Verfügung stellt (Funktionen **`assign`**, **`resign`** etc.)
  - Kapselt Zugriffe auf die Prozesstabelle und die Warteschlangen
  - Muss mit Sonderfällen umgehen, z.B. Aufruf von **`assign`** ohne einen Prozess in der Bereit-Liste usw.
- **Scheduler** wird in der Operation **`assign`** vom Dispatcher aufgerufen
  - Short-Term-Scheduler: kurzfristige Entscheidung
    - Welcher der momentan bereiten Threads soll ausgewählt werden?
  - Long-Term-Scheduler: langfristige Entscheidung
    - Welcher Prozess soll ausgelagert werden
    - Wird nicht so häufig aufgerufen

Kontextwechsel

- Kontextwechsel = Wechsel des laufenden Threads auf dem Prozessor
  - Kontext = Prozessorkontext (Registerinhalte)
  - Prozessor kann auch ein virtueller Prozessor sein
- Bestandteile eines Kontextwechsels:
  - Unterbrechung des gerade laufenden Threads A
  - Sichern des Kontext von Thead A aus dem Prozessor in die Threadtabelle
  - Auswahl eines neuen Threads B aus der Bereitliste
  - Laden des Kontext von Thread B aus der Threadtabelle in den Prozessor
  - „Fortsetzung" von Thread B
- Der Code für einen Kontextwechsel gehört zu den kompliziertesten Teilen des Betriebssystemcodes
  - Mehr in der Übung

Kontextwechsel-Problematik

- Kontextwechsel stellen den wesentlichen Leistungsoverhead eines Betriebssystems dar
  - Jeder Kontextwechsel kostet Prozessorzyklen, ein Kontextwechsel sollte also möglichst effizient programmiert sein
  - Anzahl der Kontextwechsel über die Lebenszeit eines Systems sollte so klein wie möglich gehalten werden
- Reduktion der Kosten eines einzelnen Kontextwechsels
  - Optimierter Assemblercode, bei dem auf jede Anweisung geachtet wird
  - Wenn möglich, Kontextwechsel nur auf User-Ebene machen (UL-Threads)
- Reduktion der Anzahl der Kontextwechsel
  - Auswahl der richtigen Schedulingstrategie

Weitere Einflussfaktoren

- Gesamtgröße und Registerstuktur eines Prozessors
  - Je mehr Mehrzweckregister zur Verfügung stehen, desto mehr muss auch gesichert und wiederhergestellt werden
- Größe und Struktur des TLB
  - Indirekte Kosten durch kalten TLB nach Adressraumwechsel
  - Manche TLBs erlauben das Retten von Kontextinformationen über einen Adressraumwechsel hinweg
- Positionierung des L2-Cache
  - "Vor" der MMU: Cache speichert virtuelle Adressen
    - Muss bei Adressraumwechsel auch invalidiert werden
  - "Hinter" der MMU: Cache speichert physische Adressen
    - Muss nur bei Änderung der Kachelzuordnung oder Nachladen einer Seite invalidiert werden

Übersicht

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