Betriebssysteme

Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 7: Implementierungsaspekte

Prof. Dr. Felix C. Freiling
Lehrstuhl für Praktische Informatik 1
Universität Mannheim

Ankündigungen 4.12.2008

- Informationsveranstaltung Teleseminar 2009, 19.12.2008, 13:00-13:30 in A5 C116
- Terminplanung:
  - Donnerstag, 4.12.2008: Abschlussvorlesung
  - Freitag, 5.12.2008, 10:15: Fragestunde
- Prüfungstermin:
  - Freitag, 19.12.2008, 14:00 Uhr in A5 B144
  - Basiskurs schreibt 66 Minuten (3 Anmeldungen)
  - Rest schreibt 100 Minuten (31 Anmeldungen)

Motivation

- Nutzen von Synchronisationskonzepten nur bei richtiger Verwendung
  - Wir haben auch Lösungen zu Standardproblemen kennengelernt
    - Erzeuger/Verbraucher, Leser/Schreiber
- Chaos bei unvorsichtiger Verwendung
  - Wichtigste Art von Chaos: Verklemmung (Deadlock)
    - Gegenseitige Blockade, die nie aufgelöst wird
- Wir untersuchen heute Standard-Programmierfehler

- Abschluss: Wie baut man heute Betriebssysteme?

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)
  6. Synchronisation paralleler Aktivitäten auf dem Rechner
  7. **Implementierungsaspekte**
     - **Implementierung nebenläufiger Programme**
     - **Wie baut man Betriebssysteme heute?**

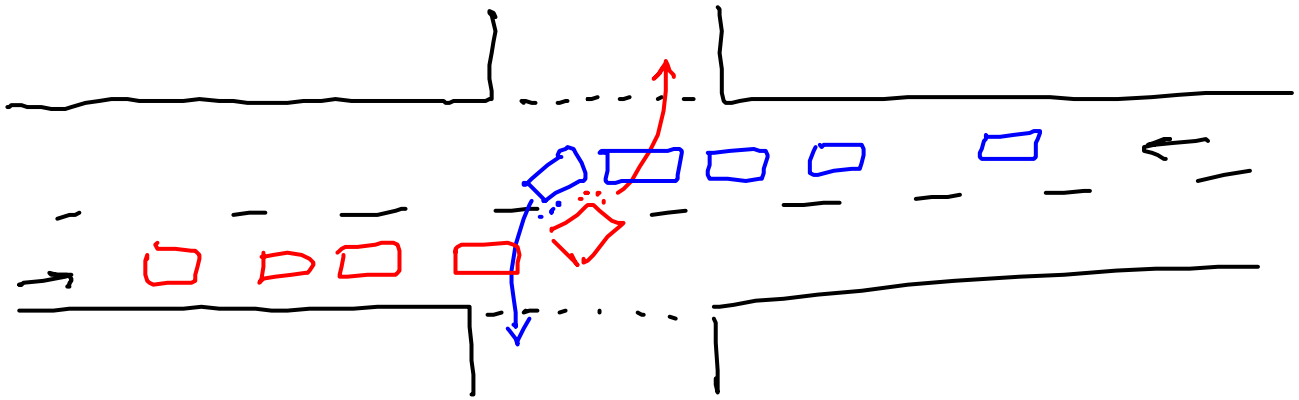
Übersicht

- Synchronisationsfehler:
  - Gegenläufige Schachtelung kritischer Abschnitte
  - Akkumulierende Belegung
  - Das Problem der speisenden Philosophen
  - Programmiertips

- Wie baut man Betriebssysteme heute?
  - Von monolithischen bis zu nebenläufigen Kernen

Synchronisationsfehler

- Ausbleibende oder inkorrekt verwendete Synchronisation kann zu Fehlern führen
  - Wesentliche Klasse von Fehlern: Verklemmungen
- Verklemmung (Deadlock)
  - Eine Gruppe von Prozessen wartet wechselseitig auf den Eintritt einer Bedingung, die nur durch Prozesse dieser Gruppe selbst hergestellt werden kann
- Beispiel: Strassenkreuzung

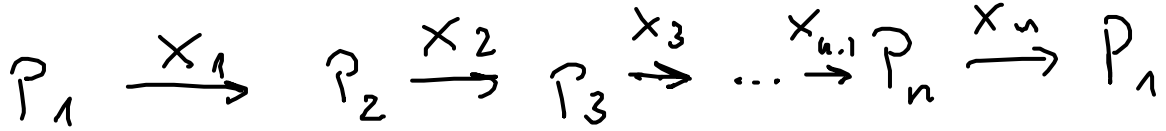


  - Annahme: Autos können nicht rückwärts fahren

Verklemmungen

- Eine Verklemmung tritt dann ein, wenn eine zirkuläre Wartebedingung eintritt
- Notation: Prozess A warten auf das Eintreten einer Bedingung X, die nur durch Prozess B hergestellt werden kann

- Eine Verklemmung tritt dann ein, falls gilt:



Synchronisationsfehler

- Aufgabe: Schreiben Sie ein minimales Programm mit Semaphoren, welches einen Deadlock hervorruft

Semaphore sem1 = 1, sem2 = 1

Thread1: P(sem1)
P(sem2)

Thread2: P(sem2)
P(sem1)

Semaphore mutex = 1
P(mutex)
P(mutex)

Semaphore foo = ∅
P(mutex)

- Denkübung: Schreiben Sie ein minimales Programm mit Monitoren, das einen Deadlock hervorruft.

Zeitabhängigkeit

- Typischerweise sind Synchronisationsfehler zeitabhängig
  - Sie treten nur bei einer bestimmten unglücklichen Aufeinanderfolge von Synchronisationsoperationen auf
    - Normalerweise Reihenfolge der Prozessaktivierung durch den Scheduler (wird als nichtdeterministisch angenommen)
  - Synchronisationsfehler sind deshalb in der Regel nicht einfach reproduzierbar
  - Können insbesondere nicht durch systematisches Testen gefunden werden
- Es gilt also: Aufpassen bei der nebenläufigen Programmierung
- Jetzt einige Beispiele für beliebte Probleme und Fehler, die passieren können

Übersicht

- Synchronisationsfehler:
  - **Gegenläufige Schachtelung kritischer Abschnitte**
  - Akkumulierende Belegung
  - Das Problem der speisenden Philosophen
  - Programmiertips

- Wie baut man Betriebssysteme heute?
  - Von monolithischen bis zu nebenläufigen Kernen

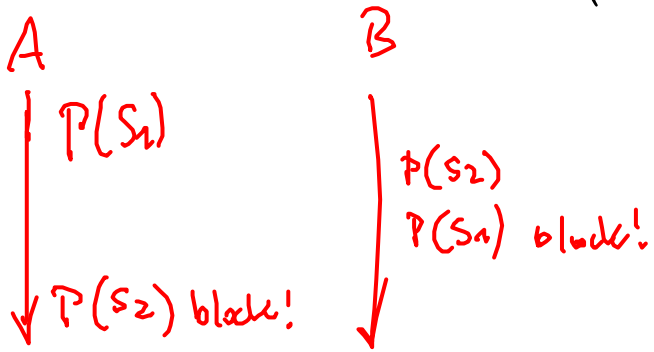
Geschachtelte kritische Abschnitte

- Ein Prozess benötigt zwei Ressourcen, um eine Operation auszuführen
  - Zweite Ressource kann erst nach erfolgreicher Belegung der ersten Ressource angefragt werden
- Konzept der geschachtelten kritischen Abschnitte
  - Beispiel: Zwei Rechner mit angeschlossenen Diskettenlaufwerken möchten eine Diskette kopieren

```
Semaphor floppyA(1), floppyB(1);
...
P(floppyA);
// äußerer kritische Abschnitt
P(floppyB);
// innerer kritischer Abschnitt
// Kopieroperation durchführen
V(floppyB);
V(floppyA);
```

Wo steckt der Fehler?

```
Semaphor s1(1), s2(1);
...
Prozess A                    Prozess B
...                          ...
P(s1);                       P(s2);
P(s2);                       P(s1);
// Kopieren                  // Kopieren
V(s2);                       V(s1);
V(s1);                       V(s2);
```



Bemerkungen

- Aus der lokalen Sicht der einzelnen Prozesse ist alles in Ordnung
- Verklemmung tritt auf, falls
  - Prozess A Semaphor s1 belegt hat und noch nicht Semaphor s2 belegt hat
  - Prozess B in diesem Moment Semaphor s2 belegt
- Typisches Beispiel für zeitabhängige Fehler
  - Problem tritt nicht immer auf, nur bei unglücklicher Wahl der Scheduling-Reihenfolge

Übersicht

- Synchronisationsfehler:
  - Gegenläufige Schachtelung kritischer Abschnitte
  - **Akkumulierende Belegung**
  - Das Problem der speisenden Philosophen
  - Programmiertips

- Wie baut man Betriebssysteme heute?
  - Von monolithischen bis zu nebenläufigen Kernen

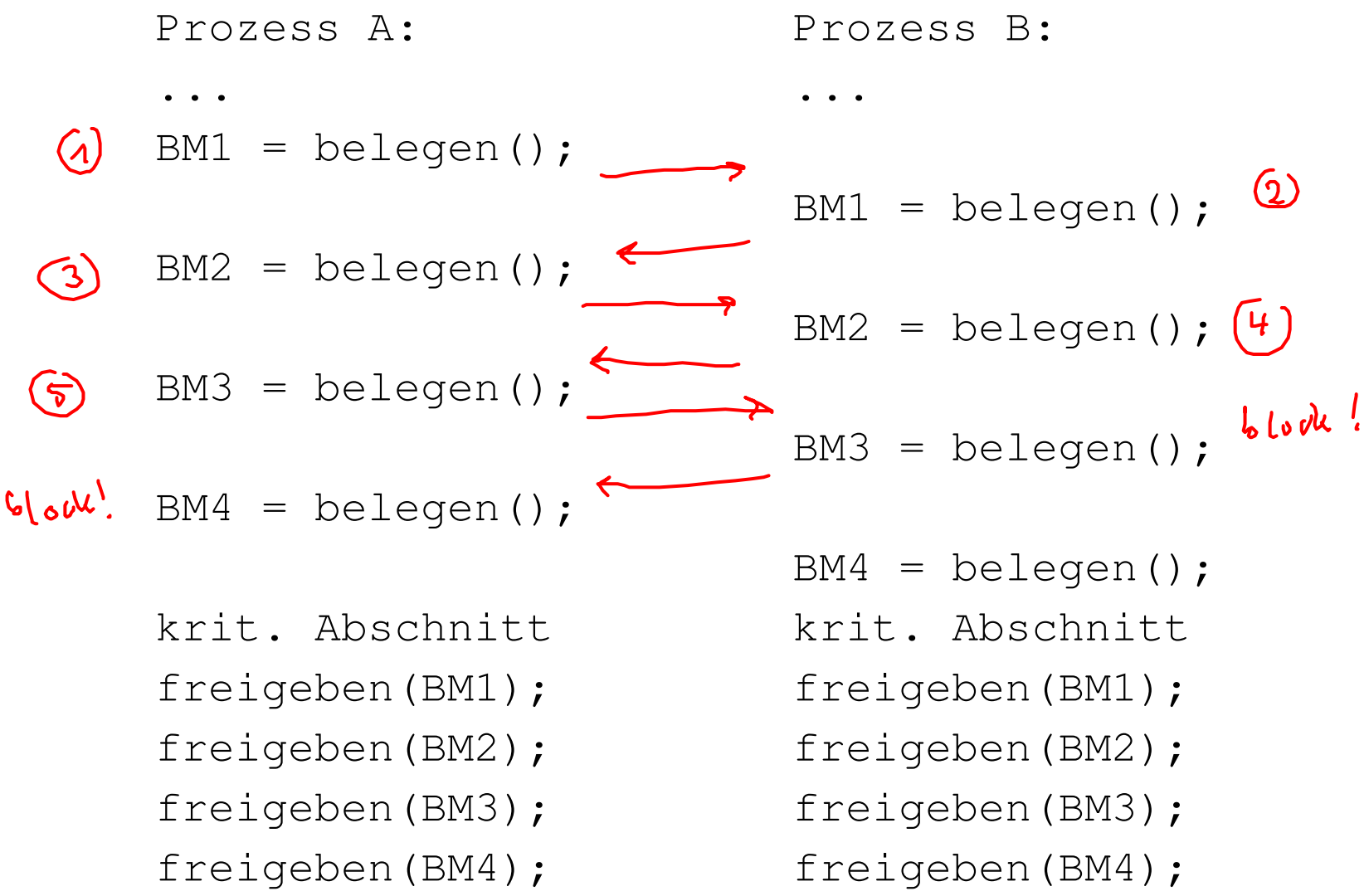
Akkumulierende Belegung

- Szenario:
  - Es gibt eine endliche Anzahl n an Ressourcen
    - Beispiel: Kacheln im Hauptspeicher
  - Eine Reihe von Prozessen benötigt eine bestimmte Anzahl k an Ressourcen, um eine Aufgabe durchführen zu können
    - Beispiel: einen neuen Prozess starten
  - Ressourcen können nur einzeln belegt und freigegeben werden
    - Operation `belegen()` **liefert freie Ressource**, falls verfügbar, **blockiert sonst**
- Typischer Code (hier n = 3, k = 2):

```
...
BM1 = belegen();
BM2 = belegen();
... // Aufgabe durchführen
freigeben(BM1);
freigeben(BM2);
```

Wo steckt der Fehler?

- Hier n = 5, k = 4:



Bemerkungen

- Deadlock, falls ...
  - Prozess A drei Betriebsmittel angefordert hat, dann unterbrochen wird, und anschliessend Prozess B zwei Betriebsmittel anfordert
  - Jetzt kann niemand mehr Ressourcen freigeben
- Paradoxie: Der Gesamtvorrat an Betriebsmitteln ist ausreichend für jede einzelne Anfrage allein
- Problem tritt auch dann auf, wenn 2k < n (siehe gleich)

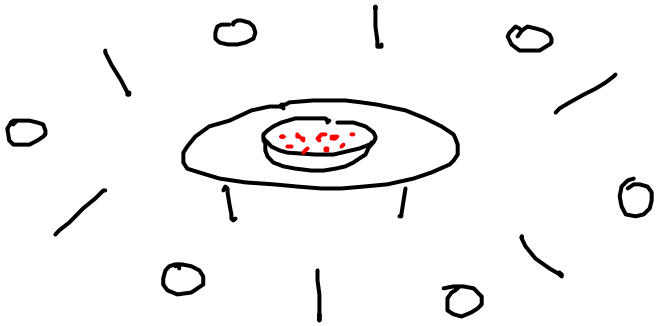
Übersicht

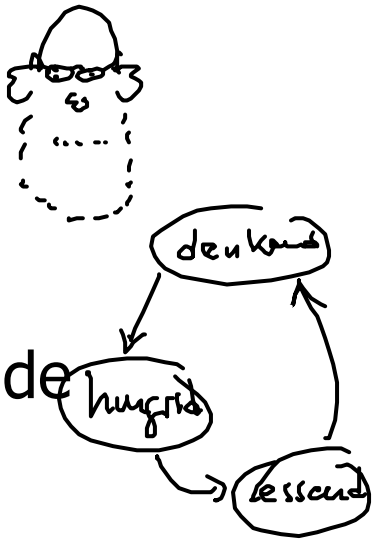
- Synchronisationsfehler:
  - Gegenläufige Schachtelung kritischer Abschnitte
  - Akkumulierende Belegung
  - **Das Problem der speisenden Philosophen**
  - Programmiertips

- Wie baut man Betriebssysteme heute?
  - Von monolithischen bis zu nebenläufigen Kernen

Dining Philosophers (Dijkstra)

- Szenario:
  - n Philosophen sind zyklisch denkend, hungrig, essend
  - Tisch mit einem niemals endenden Vorrat Reis
  - n Stäbchen (eins zwischen jeweils zwei benachbarten Philosophen)

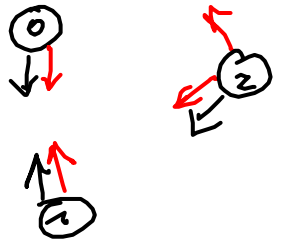




- Aufgabe: Schreibe einen Algorithmus, der folgende Bedingungen erfüllt
  - Kein Philosoph verhungert
  - Ein Stäbchen kann immer nur von einem Philosophen benutzt werden
  - Zum Essen braucht man zwei Stäbchen

Wo steckt der Fehler?

- Zum Essen nimmt Philosoph k
  - zuerst das linke Stäbchen (Position k) und dann
  - das rechte Stäbchen (Position (k+1) mod n)



```
Semaphor Stäbchen[n]

Essen (Platz k) {
  P(Stäbchen[k]);
  P(Stäbchen[(k+1) mod n]);
  // Guten Appetit
  V(Stäbchen[(k+1) mod n]);
  V(Stäbchen[k]);
}
```

Bemerkungen

- Lösung ist fast richtig:
  - Geschachtelte kritische Abschnitte sind richtig gesetzt
  - Maximal ein Philosoph isst an einem Teller
- Aber Verklemmung, falls
  - alle n Philosophen gleichzeitig ihr linkes Stäbchen nehmen
- Lösung:
  - Symmetrie an einer Stelle brechen
  - Lösung 1: Ein Philosoph nimmt zuerst rechtes und dann linkes Stäbchen
  - Alternativ: Erst "gerades" Stäbchen und dann "ungerades" Stäbchen nehmen
    - Mehr Parallelität als Lösung 1
  - Alternativ: Zugriff auf Stäbchen mittels Mutex kapseln
    - Wenig Parallelität, Problem mit Deadlock (Blockade im Mutex)

Übersicht

- Synchronisationsfehler:
  - Gegenläufige Schachtelung kritischer Abschnitte
  - Akkumulierende Belegung
  - Das Problem der speisenden Philosophen
  - **Programmiertips**

- Wie baut man Betriebssysteme heute?
  - Von monolithischen bis zu nebenläufigen Kernen

Korrekte Lösungen

- "Premature optimization is the root of all evil."

  (Donald Knuth)

  - Erst korrekte Lösung schaffen, dann eventuell über Optimierungen (auch bezüglich der Effizienz) nachdenken
- An Standardprobleme halten
  - Kann man das zu lösende Problem als Instanz eines Standardproblems ansehen?
    - Erzeuger/Verbraucher
    - Leser/Schreiber
- Lösungen auf Deadlocks hin untersuchen
  - Wann immer möglich: Deadlockvermeidung (Auflösung potentieller zyklischer Wartebedingungen)
  - Für kritische Systeme: Deadlockerkennungsalgorithmen einsetzen
    - Periodisch den Systemzustand auf Deadlocks überprüfen und ggf. zyklische Abhängigkeit sprengen

Safety und Liveness

- Safety = Lösung macht nie etwas Falsches
- Liveness = Lösung macht irgendetwas
- Kunst: Lösung muss Safety *und* Liveness erfüllen
  - Nur safe: Lösung macht nichts
  - Nur live: Lösung macht alles
- Bei der Konstruktion kann man auf zwei Arten vorgehen:
  - Top down: Safety first
    - Möglichst konservativ synchronisieren, Deadlocks in Kauf nehmen
  - Bottom up: Liveness first
    - Beginnen mit einem Programm vollkommen ohne Synchronisation
    - Anschliessend Synchronisationsoperationen einbringen
    - Anfangs Risiko von Inkonsistenzen
  - Mischformen möglich

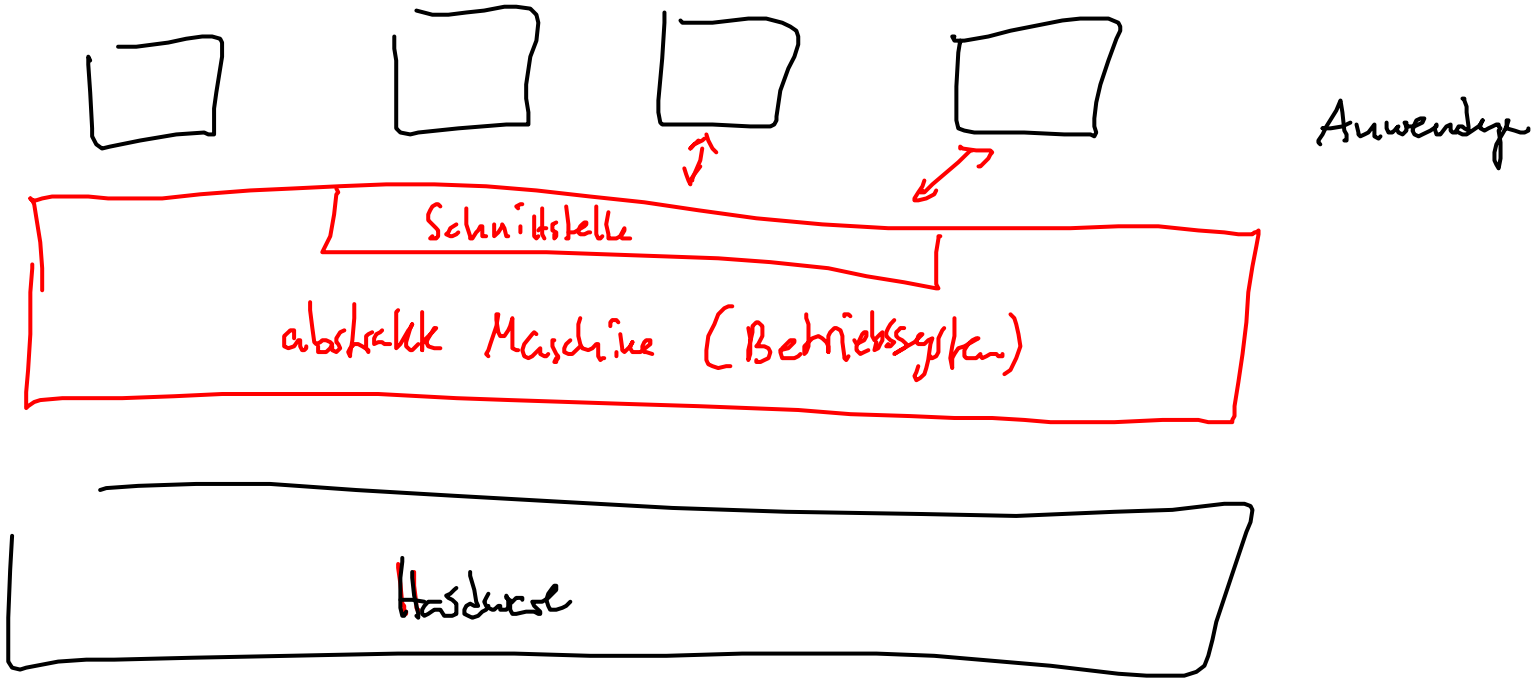
Übersicht

- Synchronisationsfehler:
  - Gegenläufige Schachtelung kritischer Abschnitte
  - Akkumulierende Belegung
  - Das Problem der speisenden Philosophen
  - Programmiertips

- **Wie baut man Betriebssysteme heute?**
  - Monolithischer Ansatz
  - Geschichteter Ansatz
  - Offene Systeme: Client/Server
  - Mikrokerne
  - Speichereinbettung der Kerne
  - Serielle vs. nebenläufige Kerne
  - Nichtblockierende Kerne

Monolithischer Ansatz

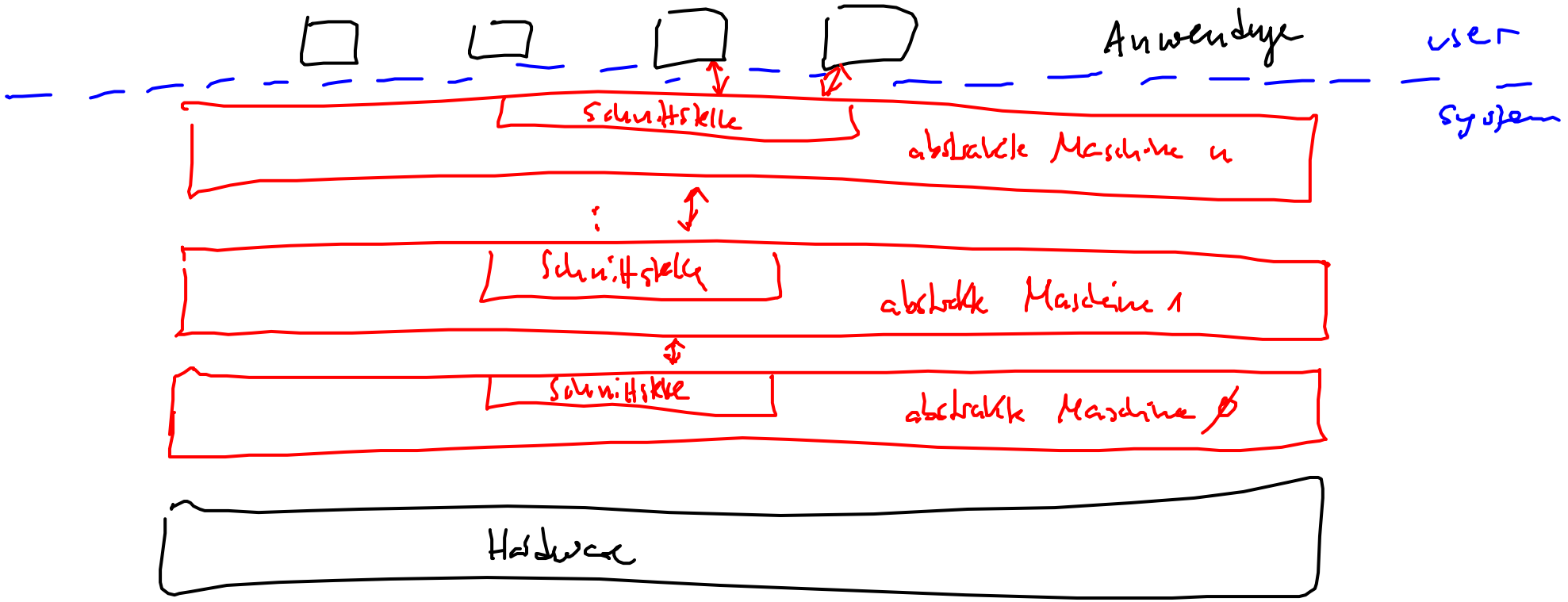
- Klassische Sichtweise: Betriebssystem ist eine Laufzeitplattform
  - Komfortable abstrakte Maschine auf unkomfortabler Hardware



- Problem: Was ist, wenn die Laufzeitplattform nicht die Dienste bietet, die die Anwendung erwartet?

Geschichteter Ansatz

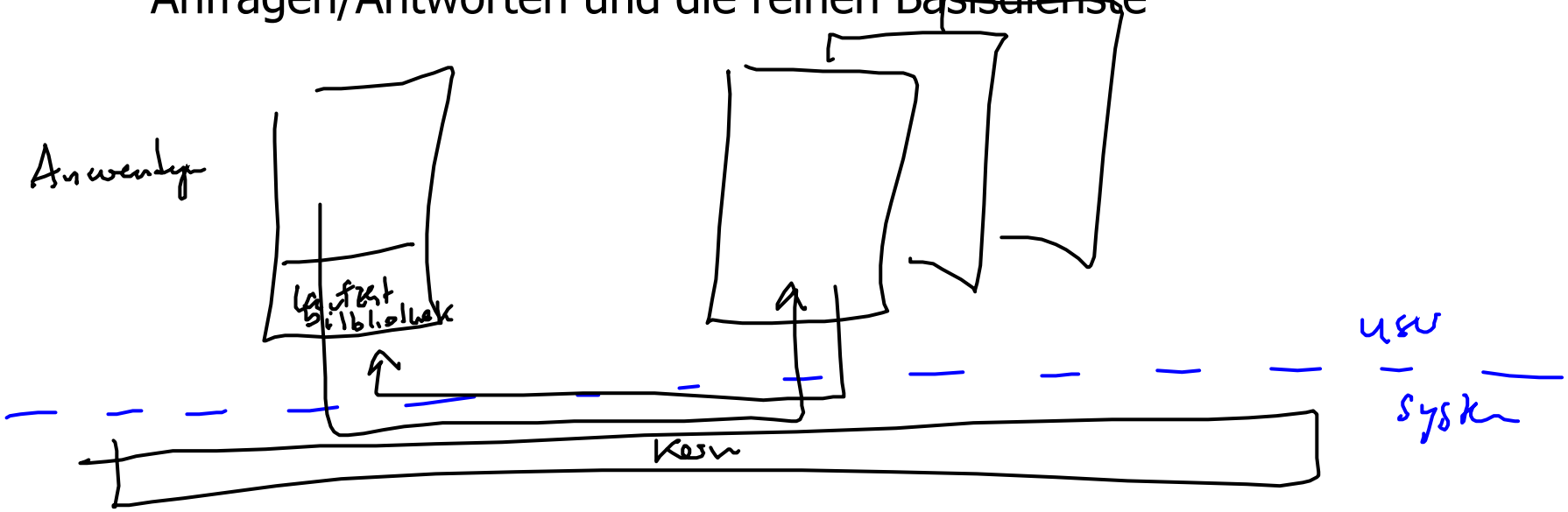
- Trennung von Funktionalität in einzeln überschaubare Schichten (Hierarchie)



- Vorteile: Reduktion von Komplexität, höhere Wartbarkeit und Erweiterbarkeit
  - Nachteile: Erweiterungen können weiter nur durch Spezialisten durchgeführt werden (sind immer im System Space)

Client/Server-Ansatz

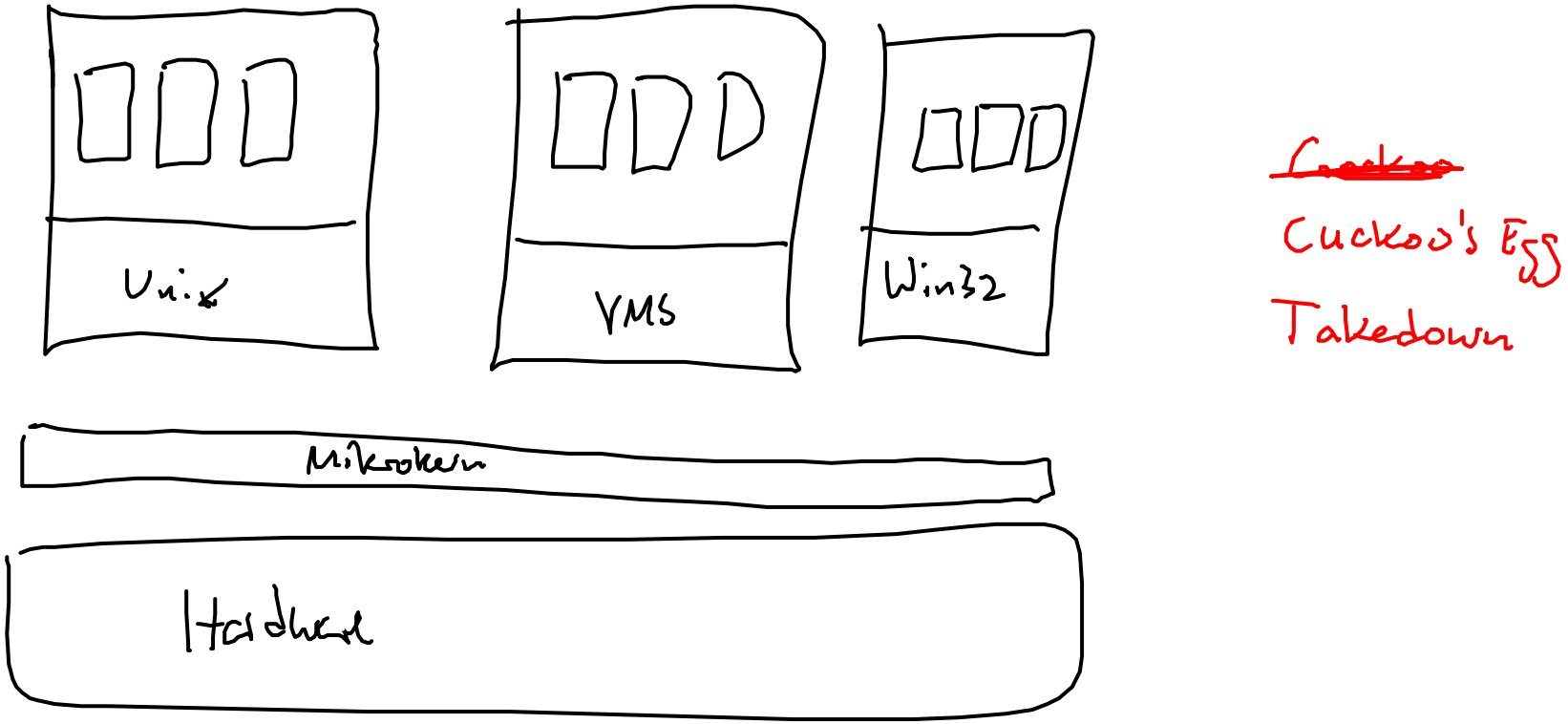
- Systemdienste sind als unabhängige Prozesse definiert, die untereinander Nachrichten austauschen
  - Bereits in Kapitel 3 vorgestellt
  - Eigentlicher Kern beschränkt sich auf das reine Vermitteln der Anfragen/Antworten und die reinen Basisdienste



- Erweiterungen können nun auch von Anwendungsprogrammierern durchgeführt werden

Mikrokerne

- Kern enthält nur Mechanismen für Adressräume, Threads und Basisschutzvorkehrungen (Autentisierung, Isolation)
  - Fliessender Übergang von Client/Server
  - Ganze Betriebssysteme können als Anwendung gesehen werden



Cuckoo's Egg
Takedown

Historisches

- 1950er: Monolithischer Ansatz
- 1960er/1970er: geschichtete Ansätze (Dijkstra, Liskov, Habermann)
- 1980er/1990er: Mikrokerne, beispielsweise:
  - V (Cheriton)
  - Amoeba (Mullender und Tanenbaum)
  - Mach (Acetta und andere)
  - Chorus (Zimmermann und andere)
  - L4 (Lietke und andere, TU Dresden, aktuell)
- Eigentliche Mikrokernidee bereits Ende der 1960er im IBM-System CP 67 (später VM) implementiert
  - Plattform für verschiedene Betriebssysteme für die IBM 360/370/390-Familie
- Auch monolithische Systeme wie Linux gehen in Richtung Modularisierung (Kernel-Module)

Serielle vs. nebenläufige Kerne

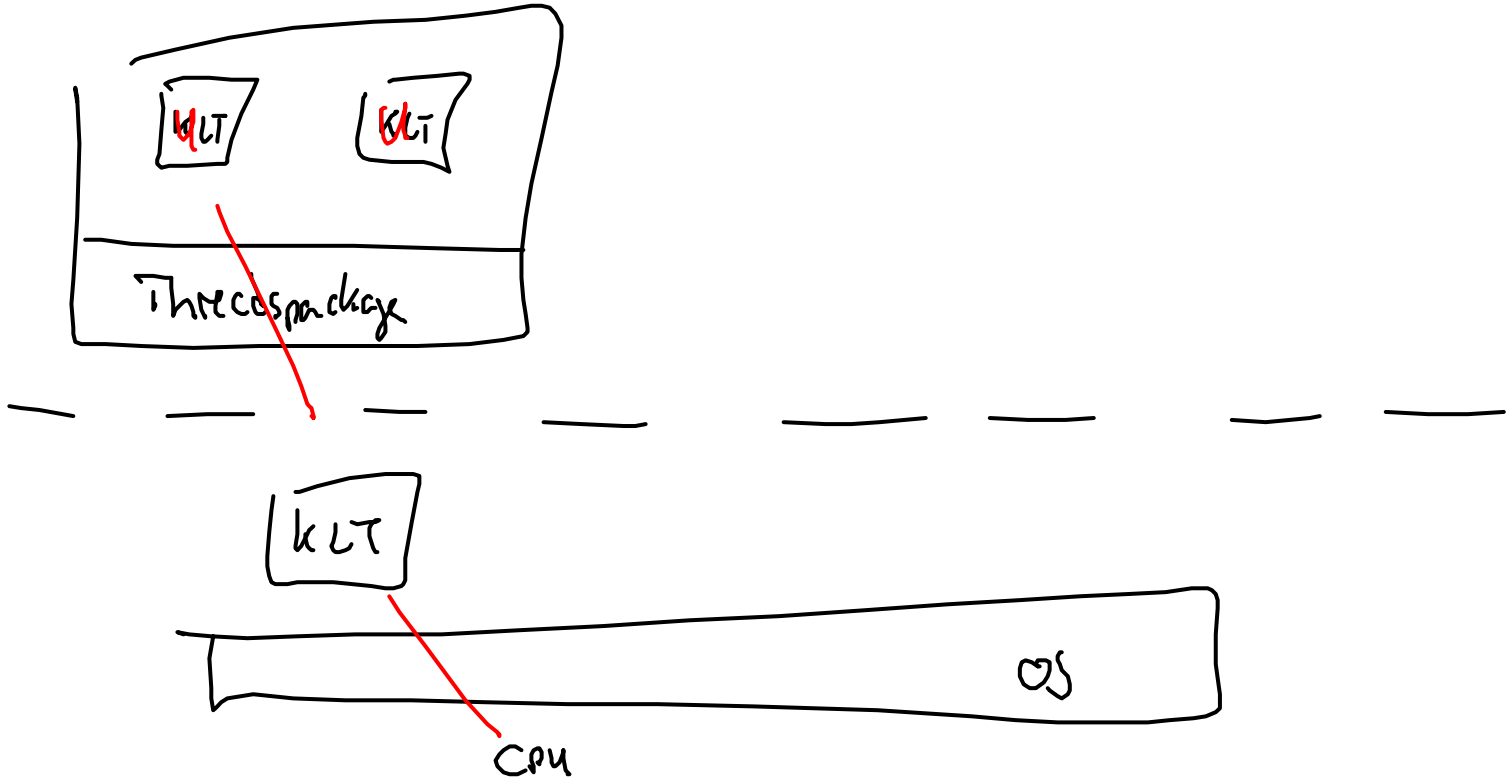
- Serieller Kern: Kernaufrufe sind ununterbrechbar
  - Immer nur ein Thread gleichzeitig im Kern
  - Vorteil: Man muss keine Thread Synchronisierung im Kern machen
- Probleme:
  - Was tun bei blockierenden Systemaufrufen?
  - Was tun bei asynchronen Interrupts?
- Antwort: Aufweichen der Ununterbrechbarkeit des Kerns
  - Notwendig bei der Umsetzung: Einhaltung von Regeln und Programmierdisziplin
- Umschalten zwischen mehreren Prozessen im Kern ist möglich, wenn
  - Blockierende Kernoperationen müssen Datenstrukturen in einem konsistenten Zustand hinterlassen
  - Asynchrone Interrupts dürfen im Kern arbeitende Threads nicht (sofort) verdrängen

Mikrokerne und Nebenläufigkeit

- Mikrokerne kommen dem Wunsch nach nebenläufigen Kernen sehr nahe
  - Kritische Abschnitte bezüglich der Synchronisation sind im Mikrokern gesammelt
    - Prozessormultiplexer
    - elementare Synchronisationsmechanismen (z.B. Semaphore)
  - Mikrokern muss streng synchronisiert sein
    - Mit Unterbrechungssperren und ggf. Spin Locks
- Über dem Mikrokern liegen "höhere Kernfunktionen"
  - Strenge Synchronisation dort nicht mehr notwendig bzw. effizient über Semaphore etc. möglich
- Nebenläufige Kerne auch "in einer Schicht" möglich
  - Wohlüberlegtes Setzen von Unterbrechungssperren
  - Heute Stand der Technik

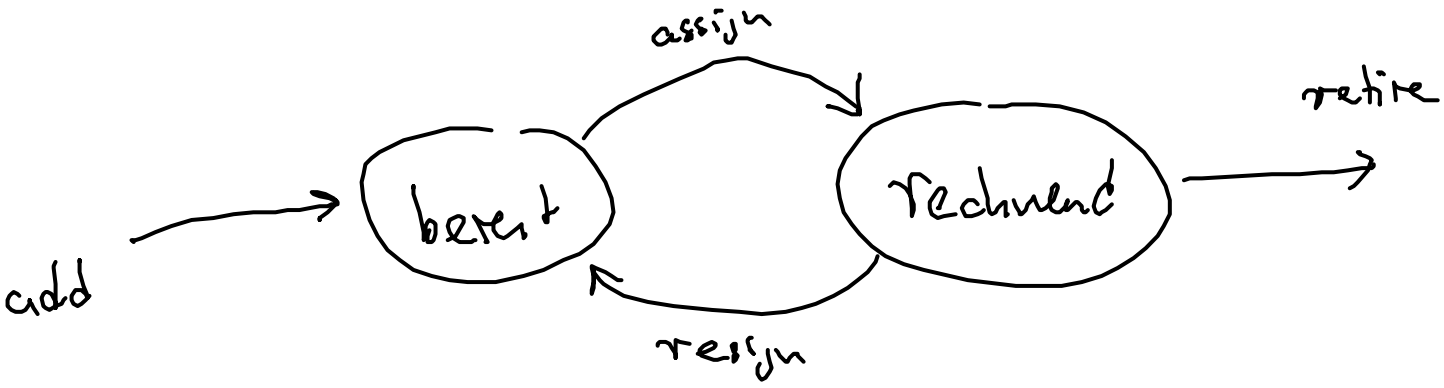
Probleme blockierender Threads

- Bei der Implementierung von User Level-Threads kann es zu folgendem Problem kommen
  - Blockade des KL-Threads führt zum Stillstand des kompletten Threadspackages



Nichtblockierende Kerne

- Lösung: Es gibt keine blockierenden Systemaufrufe
  - Prozesszustandsmodell vereinfacht sich



- Management von Blockaden wird auf höhere Ebene verlagert
  - Schafft mehr Flexibilität
  - Schwieriger zu programmieren

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)
  6. Synchronisation paralleler Aktivitäten auf dem Rechner
  7. Implementierungsaspekte

Lernziele im Rückblick

- Beschäftigung mit den Grundsätzlichen Problemen und Lösungsmechanismen klassischer Betriebssysteme
- Betrachtung ausgewählter Beispiele (meistens Unix/Linux-artig, Beispiel ULIX)
- Verständnis für die Konstruktionsprinzipien von Systemsoftware, um Anwendungen besser darauf abstimmen zu können, bzw. Anwendungsverhalten besser zu diagnostizieren
- Verständnis für die Herausforderungen in der nebenläufigen Programmierung

Ausblick

- Vertiefende Vorlesungen:
  - Vorlesung **Principles of Dependable Systems**, jeweils im Frühjahrssemester (leider aber nicht in 2009)
    - Vertiefung von nebenläufiger Programmierung inklusive Behandlung von Fehlern:
    - Themen: fehlertolerante verteilte Systeme, fehlertolerante Synchronisation, Sicherheitsprotokolle
  - Vorlesung **Forensische Informatik** (im Frühjahrssemester)
    - Vertiefung Dateisysteme, Spurensicherung in Betriebssystemen
  - Vorlesungen **Middleware-Technologien** (Prof. Becker)
    - Vertiefung der Thematik "Verteiltheit" sowie in Richtung Anwendungsebene
  - Vorlesung **Rechnernetze** (Prof. Effelsberg)
    - Vertiefung in Richtung Netzwerkprotokolle, Netzwerktechnologien