TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

BETRIEBSSYSTEME UND SICHERHEIT

mit Material von Olaf Spinczyk,
Universität Osnabrück

Multiprozessorsysteme

https://tud.de/inf/os/studium/vorlesungen/bs

HORST SCHIRMEIER

Inhalt

- Wiederholung
- Hardwaregrundlagen
- Anforderungen
- Synchronisation
- CPU-Zuteilung
- Zusammenfassung

Silberschatz, Kap. ...
--- ☹

Tanenbaum, Kap. ...
8: Multiprozessorsysteme

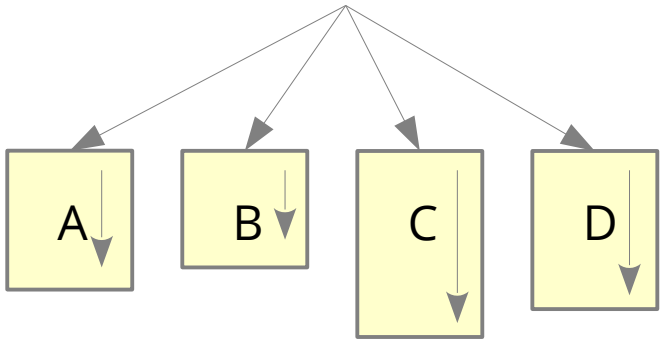
Inhalt

- **Wiederholung**
- Hardwaregrundlagen
- Anforderungen
- Synchronisation
- CPU-Zuteilung
- Zusammenfassung

Wiederholung

- Betriebssysteme …
  - verwalten Ressourcen und …
  - stellen den Anwendungen Abstraktionen zur Verfügung.
- Prozesse **abstrahieren** von der Ressource CPU

Nebenläufige Prozesse



Multiplexing der CPU

Konzeptionelle Sicht
- 4 unabhängige sequentielle Kontrollflüsse

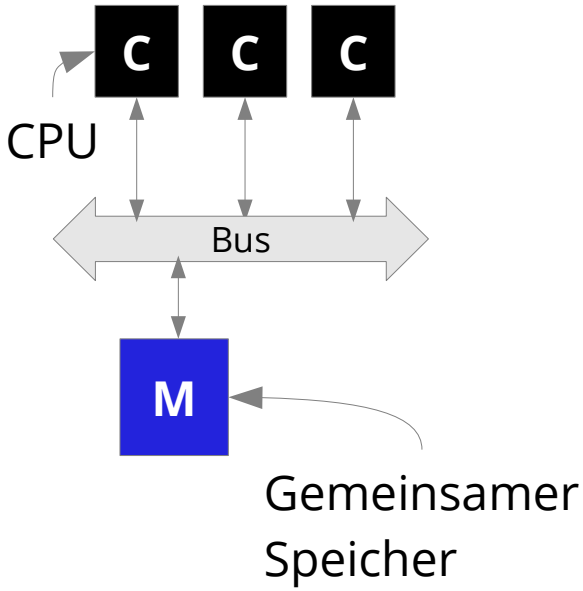
Realzeit-Sicht (Gantt-Diagramm)
- Zu jedem Zeitpunkt nur ein Prozess aktiv (**Uni-Prozessor-HW**)

Inhalt

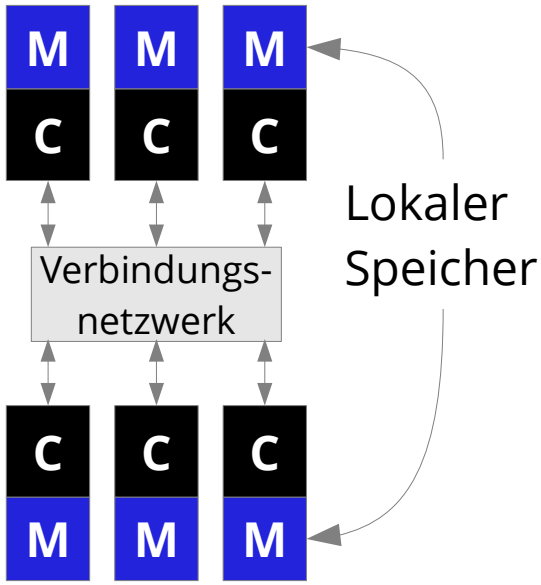
- Wiederholung
- **Hardwaregrundlagen**
- Anforderungen
- Synchronisation
- CPU-Zuteilung
- Zusammenfassung

Klassen paralleler Rechnersysteme*

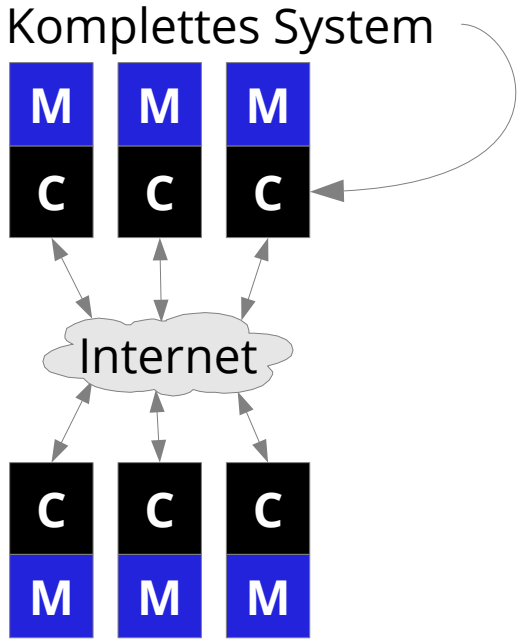
Multiprozessor-system



Multicomputer-system



Verteiltes System



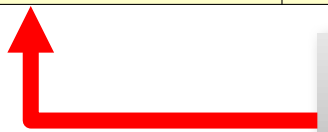
* Die Betrachtung beschränkt sich auf die sog. MIMD-Architekturen.

Klassen paralleler Rechnersysteme (2)

- Gegenüberstellung

  (nach Tanenbaum, „*Modern Operating Systems*“)

Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared, except maybe disc	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations



Im Folgenden wird es nur noch um Multiprozessorsysteme gehen.

Diskussion: Skalierbarkeit

- **Definition:** Eine parallele Rechnerarchitektur gilt als **skalierbar**, wenn die effektiv verfügbare Rechenleistung sich proportional zur Anzahl der eingebauten CPUs verhält.

- Ein **gemeinsamer Bus** für Speicherzugriffe und der **gemeinsame Speicher-Controller** werden bei Systemen mit vielen CPUs zum Flaschenhals.
  - Selbst das Holen von unabhängigen Instruktionen oder Daten kann zu Konkurrenzsituationen führen!

➔ Bus-basierte Multiprozessorsysteme skalieren schlecht
  - Trotz Einsatz von Caches typischerweise ≤ 64 CPUs
  - Parallele Systeme mit mehr CPUs sind Multi**computer** mit dediziertem Verbindungsnetzwerk und verteiltem Speicher
    - 2020: Fujitsu Fugaku: 7.299.072 Cores; 415,5 PetaFLOPS (=10^{15} FLOPS)
    - 2018: IBM Summit: 2.282.544 Cores; 122,3 PetaFLOPS

siehe top500.org

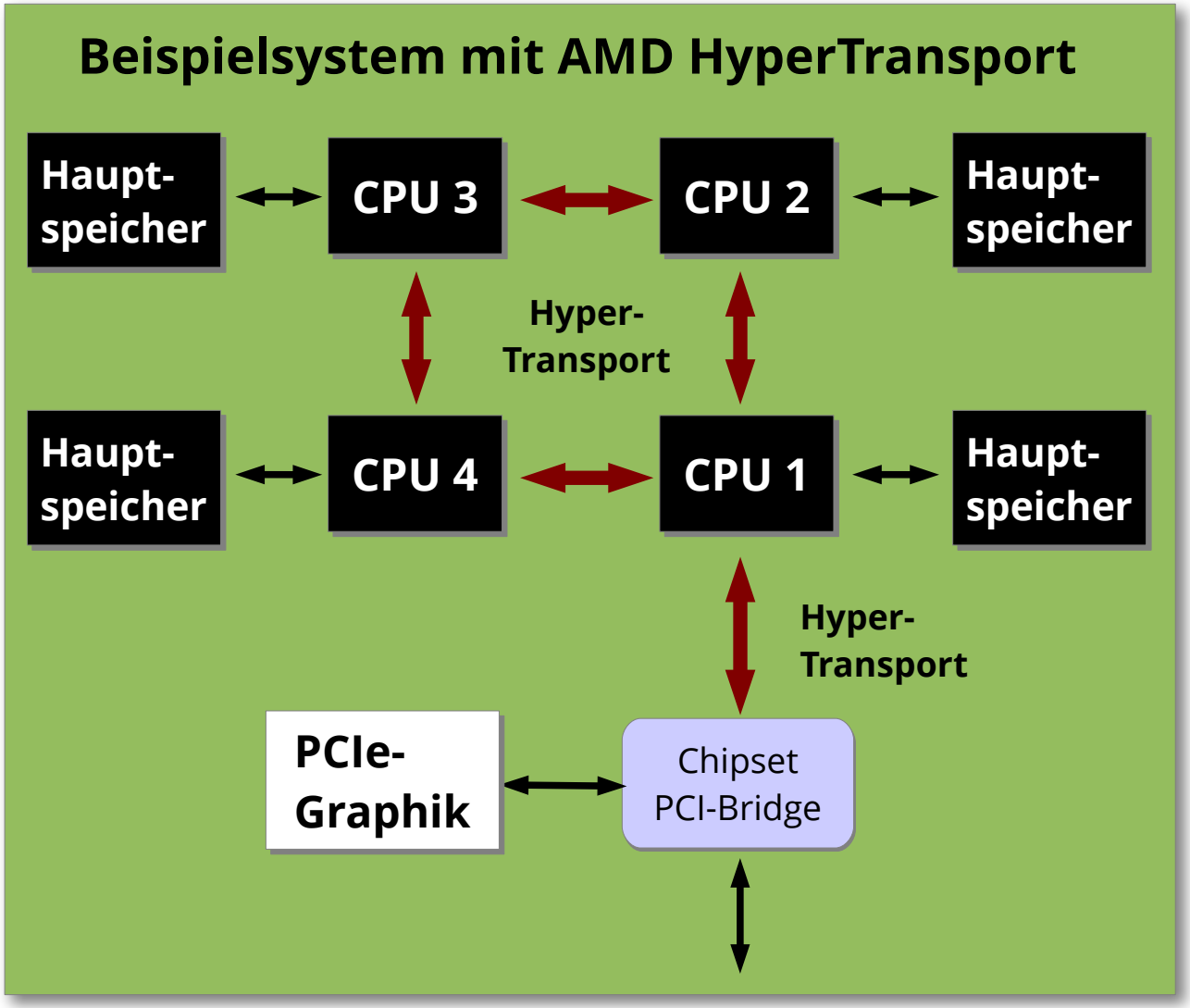
NUMA-Architekturen

(*Non-Uniform Memory Architecture*)

Die CPUs (u.U. mit mehreren Cores) kommunizieren untereinander via *HyperTransport*.

Globaler Adressraum: An andere CPUs angebundener Hauptspeicher kann adressiert werden, die Latenz ist jedoch höher.

Ansatz skaliert besser, da parallele Speicherzugriffe möglich sind.



Multiprozessorsysteme im Detail

- **Definition:** Ein **Multiprozessorsystem** ist ein Rechnersystem, in dem zwei oder mehr CPUs vollen Zugriff auf einen gemeinsamen Speicher haben.

- Die CPUs eines Mehrprozessorsystems können auch auf einem Chip integriert sein → ***Multicore*-CPU**
- CPUs weisen typischerweise *Caches* auf
- Rechnersysteme bestehen nicht nur aus CPU + Speicher
  - E/A-*Controller*!

- Offene Fragen
  - Wie erreicht man ***Cache*-Kohärenz**?
  - Werden Maschinen-Instruktionen weiterhin atomar ausgeführt?
  - Wer verarbeitet Unterbrechungen?

Diskussion: Konsistenz vs. Kohärenz

- **„Konsistenz“** bedeutet „in sich stimmig“ → nach innen
  - Hängt von der Konsistenzbedingung ab
    - Beispiel: Jedes Element einer einfach verketteten Liste wird genau einmal referenziert – das erste vom Listenkopf, der Rest von anderen Listenelementen.
  - Ein *Cache* wäre inkonsistent, wenn zum Beispiel dieselben Speicherinhalte mehrfach im *Cache* wären.

- **„Kohärenz“** bedeutet „Zusammenhalt“ → nach außen
  - *Cache*-Kohärenz ist eine Beziehung zwischen den verschiedenen *Caches* in einem Multiprozessorsystem.

Das MESI-Protokoll (1)

- … ist ein gängiges ***Cache*-Kohärenzprotokoll**, das die notwendige Abstimmung zwischen *Caches* in Multiprozessorsystemen implementiert.
- Jede ***Cache*-Zeile** wird um **2 Zustandsbits** erweitert:

  ***M**odified*: Daten nur in diesem Cache,
  lokale Änderung, Hauptspeicherkopie ungültig

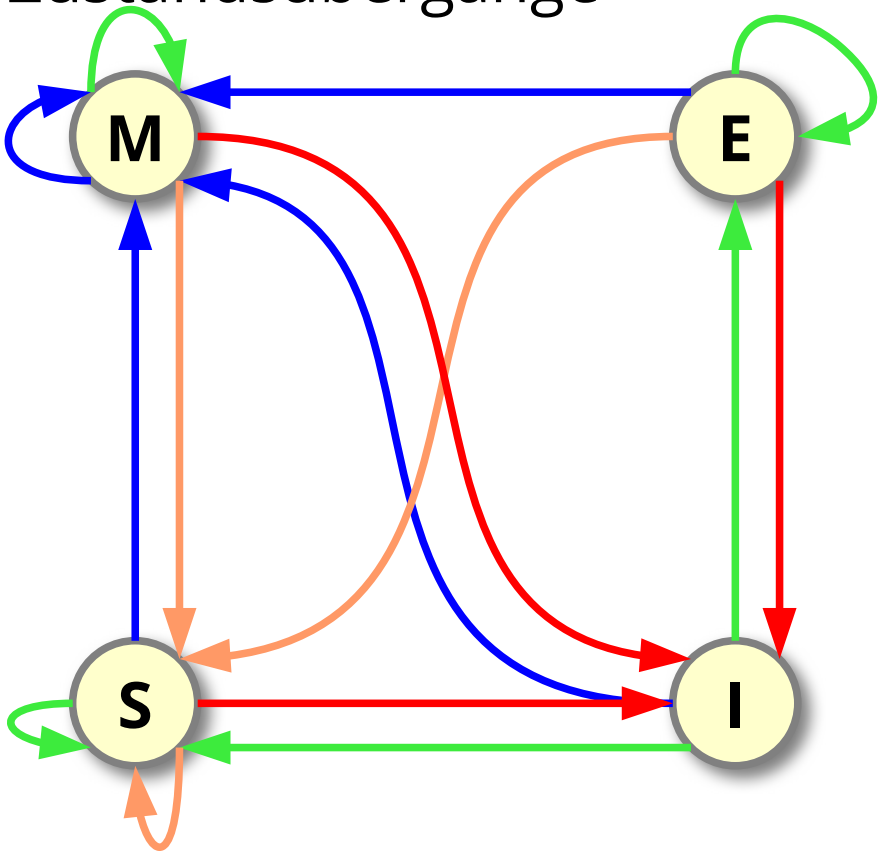
  ***E**xclusive*: Daten nur in diesem Cache,
  keine lokale Änderung, Hauptspeicherkopie gültig

  ***S**hared*: Daten sind in mehreren Caches,
  keine lokalen Änderungen, Hauptspeicherkopie gültig

  ***I**nvalid*: Der Inhalt der Cache-Zeile ist ungültig.

Das MESI-Protokoll (2)

- Zustandsübergänge



Legende:
lokaler Lesezugriff
lokaler Schreibzugriff
Lesezugriff durch andere CPU
Schreibzugriff durch andere CPU

Moderne CPUs nutzen auf der NUMA-Architektur Erweiterungen davon wie MOESI und MESIF.

- **Schnüffellogik** (*snooping logic*) liefert Informationen über Speicherzugriffe durch andere CPUs

Atomare Speicherzugriffe(?)

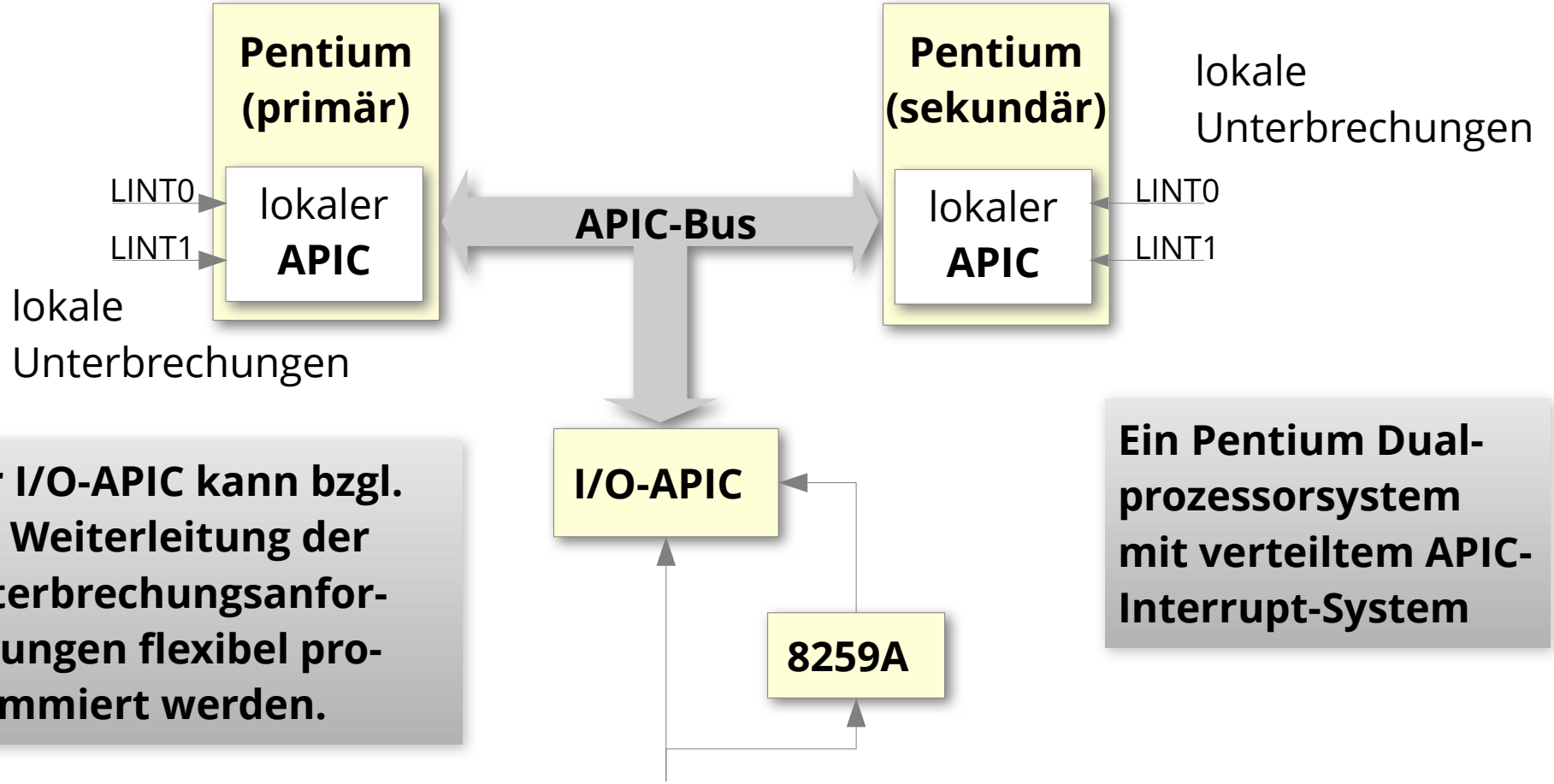
- Die **Bus-Arbitration** sorgt dafür, dass Speicherzugriffe sequentialisiert werden.
- **Aber**: Sonst (d.h. bzgl. Unterbrechungen) atomare Maschinenbefehle müssen in Multiprozessorsystemen nicht unbedingt atomar sein!
  - x86: `inc` führt zu zwei Speicherzugriffen
- **Hilfe**: Sperren des Busses
  - Spezielle Befehle mit Lese-/Modifikations-/Schreibzyklus: TAS, CAS, ...
  - x86: `lock`-Präfix

MP-Unterbrechungsbehandlung (1)

- Ein klassischer *Interrupt-Controller* priorisiert die Unterbrechungsanforderungen und leitet eine Anforderung an eine CPU weiter.
- **Multiprozessor-Interruptsysteme** müssen flexibler sein
  - Keine CPU sollte durch die Last durch Unterbrechungsbehandlung dauerhaft benachteiligt werden.
    - Nachteil für Prozesse auf dieser CPU
    - Keine Parallelverarbeitung von Unterbrechungen
  - Besser ist gleichmäßige Verteilung der Unterbrechungen auf CPUs
    - Statisch (feste Zuordnung von Unterbrechungsquelle zu CPU)
    - Dynamisch (z.B. in Abhängigkeit der aktuellen Rechenlast der CPUs)

Die Intel-APIC-Architektur

- Ein APIC-*Interrupt*-System besteht aus lokalen APICs auf jeder CPU und einem I/O-APIC



Der I/O-APIC kann bzgl. der Weiterleitung der Unterbrechungsanforderungen flexibel programmiert werden.

Ein Pentium Dual-prozessorsystem mit verteiltem APIC-Interrupt-System

MP-Unterbrechungsbehandlung (2)

... weitere Besonderheiten:

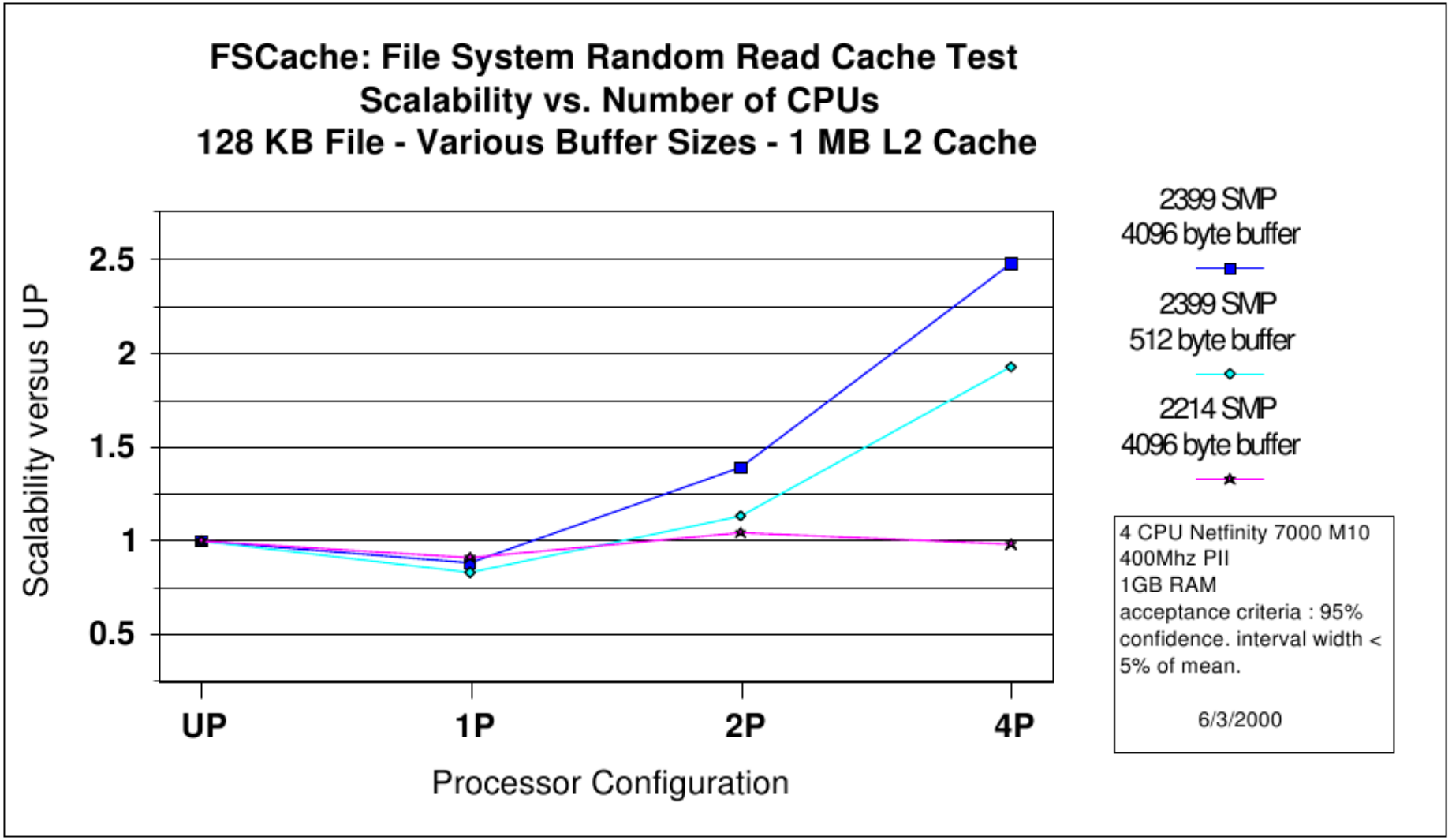
- Interprozessor-Unterbrechungen (IPI)
  - Prozessoren im Multiprozessorsystem können sich damit gegenseitig unterbrechen, z.B. aus Schlafzustand wecken.
- Maschinenbefehle zum Sperren und Erlauben von Unterbrechungen (z.B. **`cli`** und **`sti`**) wirken pro CPU
  - Problem für die Synchronisation konkurrierender Kontrollflüsse im Betriebssystem, z.B. für die Implementierung von Semaphore!

Inhalt

- Wiederholung
- Hardwaregrundlagen
- **Anforderungen**
- Synchronisation
- CPU-Zuteilung
- Zusammenfassung

Anforderungen: Skalierbarkeit

- … **der Systemsoftware** ist keine Selbstverständlichkeit:



Quelle: Ray Bryant and Bill Hartner, „SMP Scalability Comparisons of Linux® Kernels 2.2.14 and 2.3.99“

Weitere Anforderungen

- Ausnutzung **aller CPUs**
  - Eine CPU darf nicht leerlaufen, wenn laufbereite Prozesse existieren
- Beachtung spezieller **Hardwareeigenschaften**
  - Wechsel von Prozessen zu einer anderen CPU vermeiden
    - *Cache* ist „angewärmt“
  - Adressraum von Prozessen bei NUMA-Systemen lokal halten
- **E/A-Last** fair verteilen
  - Ggf. Prozessprioritäten beachten

- **Korrektheit**
  - Vermeidung von *Race Conditions* zwischen Prozessen auf unterschiedlichen CPUs → Synchronisation!

Inhalt

- Wiederholung
- Hardwaregrundlagen
- Anforderungen
- **Synchronisation**
- CPU-Zuteilung
- Zusammenfassung

Multiprozessorsynchronisation

- **Auf Prozessebene** durch passives Warten
  - Anwendung klassischer Abstraktionen wie Semaphore oder Mutex
- **Auf Betriebssystemebene** schwieriger; Beispiel:
  - **wait** und **signal** müssen per Definition unteilbar ausgeführt werden
    - Im Uniprozessorfall führen nur Unterbrechungen zu *Race Conditions*. Diese können leicht (für kurze Zeit) unterdrückt werden.
    - Im Multiprozessorfall reicht das Unterdrücken von Unterbrechungen nicht aus! Die anderen CPUs laufen unbeeinflusst weiter.

➔ **Multiprozessorsynchronisation** auf Kern-Ebene muss mit aktivem Warten (***spin locking***) realisiert werden

Spin Locking: Primitiven

lock- und *unlock*-Primitiven müssen mit unteilbaren Lese-/Modifikations-/Schreibinstruktionen implementiert werden:

- Motorola 68K: TAS (***Test-and-Set***)
  - Setzt Bit 7 des Zieloperanden und liefert den vorherigen Zustand in *Condition Code Bits*

```
acquire TAS   lock
        BNE   acquire
```

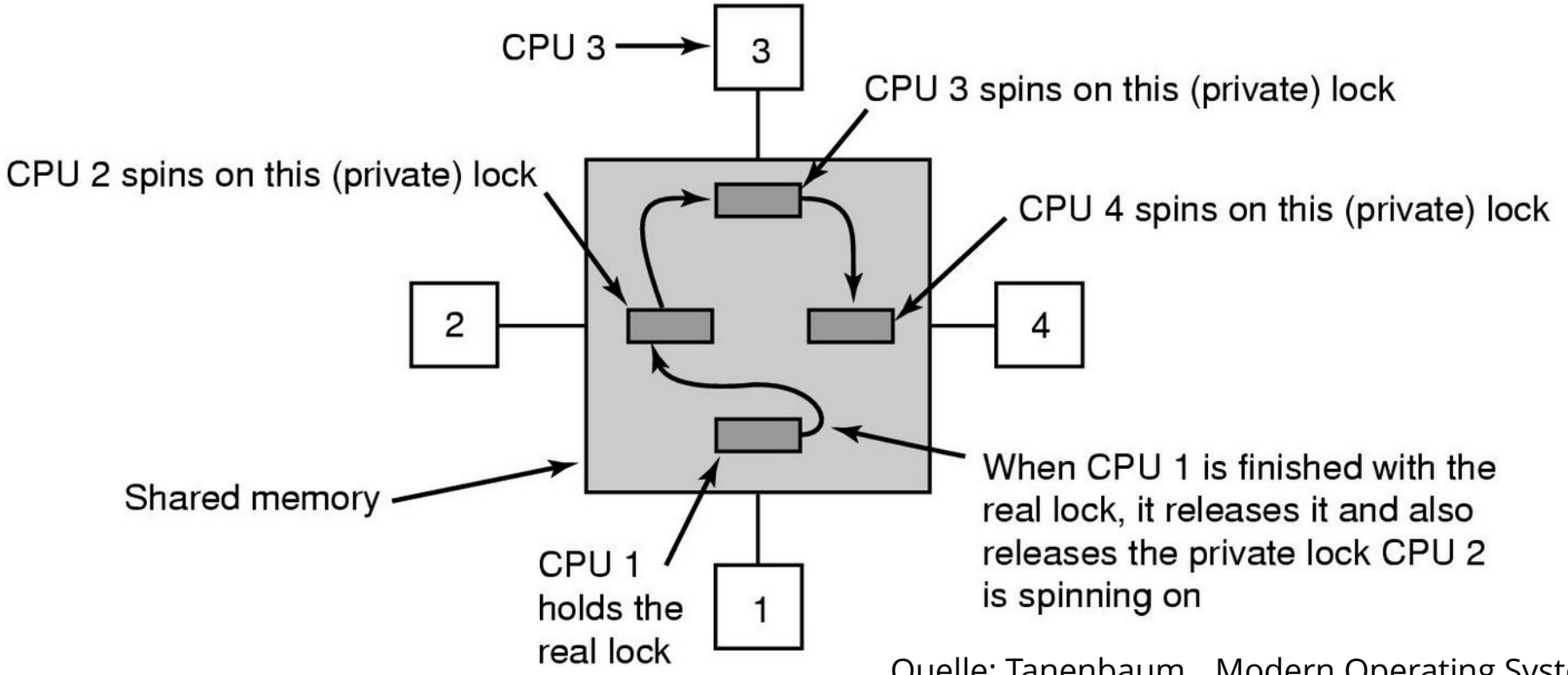
- Intel x86: XCHG (***Exchange***)
  - Tauscht den Inhalt eines Registers mit dem einer Variablen im Speicher

```
        mov   ax,1
acquire:xchg  ax,lock
        cmp ax,0
        jne   acquire
```

- PowerPC: LL/SC (*Load Linked/Store Conditional*)
- ...

Spin Locking: Effizienz

- Um ***Cache-Thrashing*** zu vermeiden, sollten **lokale Sperrvariablen** benutzt werden
  - (hohe Buslast durch viele konkurrierende Schreibzugriffe → MESI)



Quelle: Tanenbaum, „Modern Operating Systems“

Spin Locking: Granularität (1)

- Um Linux multiprozessortauglich zu machen, wurde der ***„Big Kernel Lock“*** (BKL) eingeführt.
  - Extrem **grobgranulares Sperren**: Nur ein Prozessor durfte den Linux-Kern betreten. Alle anderen mussten aktiv warten.
  - Linux 2.0 und 2.2 System skalierten daher sehr schlecht.
    - Faustregel: bis zu 4 CPUs
- Neuere Linux Systeme verwenden stattdessen viele „kleinere Sperren“ für Datenstrukturen innerhalb des Kerns.
  - **Feingranulares Sperren**
  - Mehrere Prozessoren können unterschiedliche Teile des Systems parallel ausführen.
  - Linux 2.4-, 2.6-, ... -Systeme skalieren erheblich besser

Spin Locking: Granularität (2)

Die ideale Sperrgranularität zu finden ist nicht einfach:

- Zu **grobgranular**:
  - Prozessoren müssen unnötig warten
  - Zyklen werden verschwendet
- Zu **feingranular**:
  - Auf dem Ausführungspfad eines Prozesses durch den Kern müssen evtl. viele Sperren reserviert und freigegeben werden.
    - Extra Aufwand – selbst wenn keine Konkurrenzsituation auftritt
  - Code wird unübersichtlich. Aufrufe von Sperrprimitiven müssen an diversen Stellen eingestreut werden.
  - Verwendung mehrerer Sperren birgt Gefahr der Verklemmung.

Spin Locking: Granularität (3)

Wie sieht es heute in **Linux** aus?

- feingranulares Sperren mit allen Vor- und Nachteilen
- es hilft nur eine andere Software-Architektur
  - käme Wegwerfen gleich

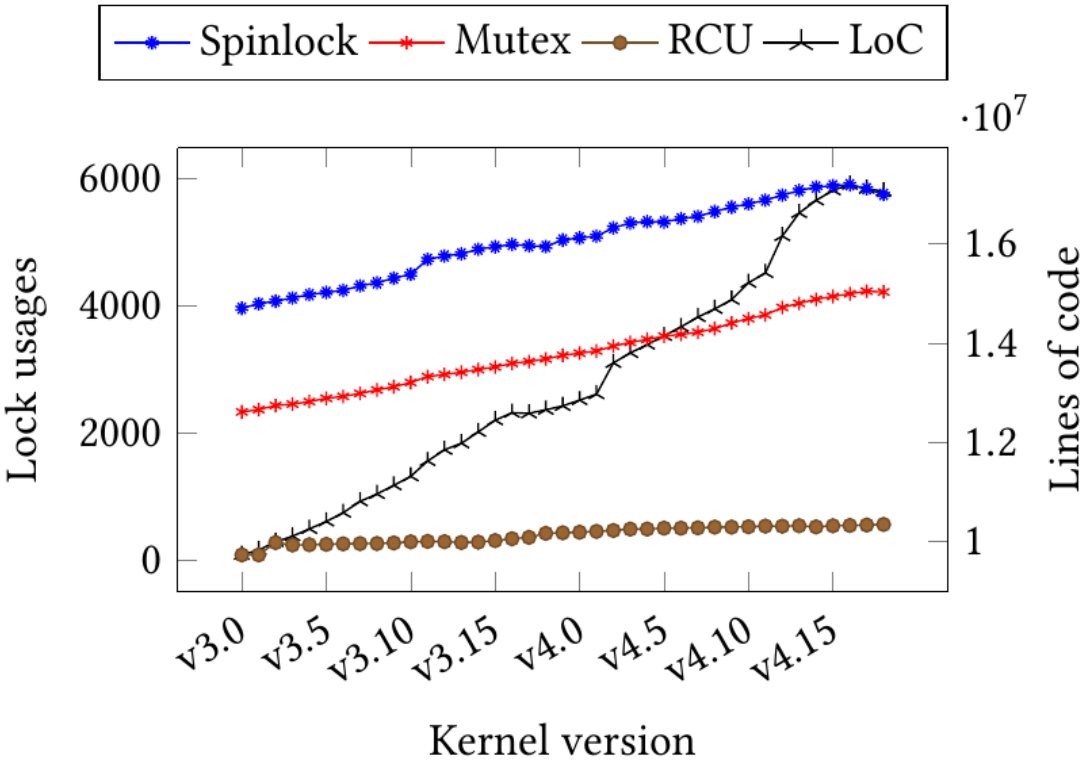


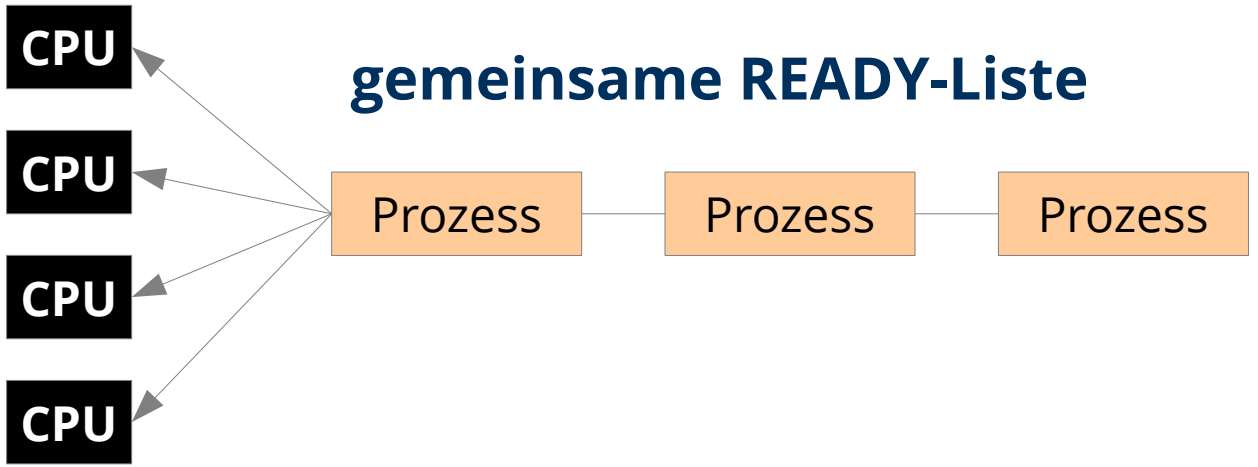
Figure 1. Increase of lock usage and lines of code (LoC) from Linux 3.0 to 4.18.

Quelle [1]

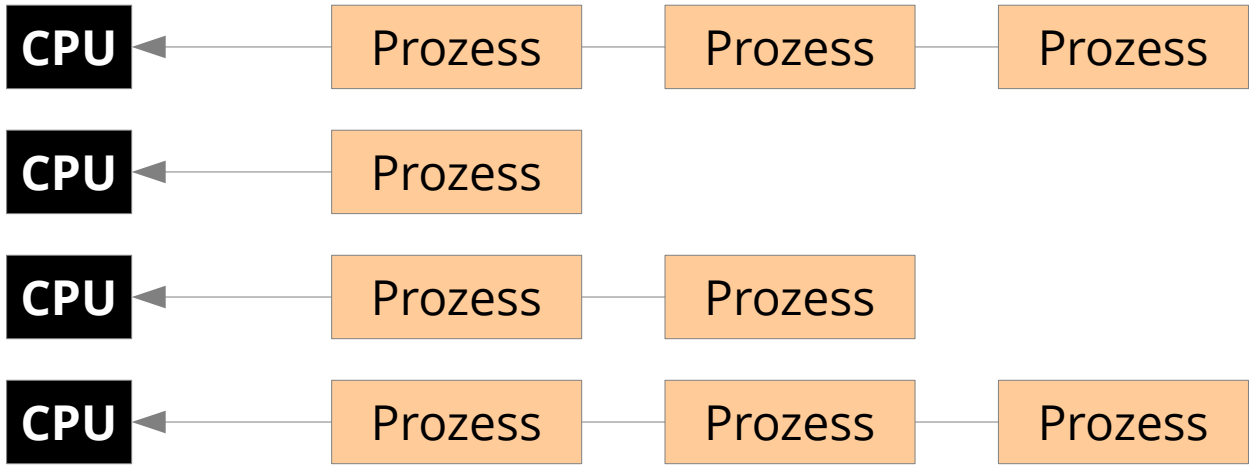
Inhalt

- Wiederholung
- Hardwaregrundlagen
- Anforderungen
- Synchronisation
- **CPU-Zuteilung**
- Zusammenfassung

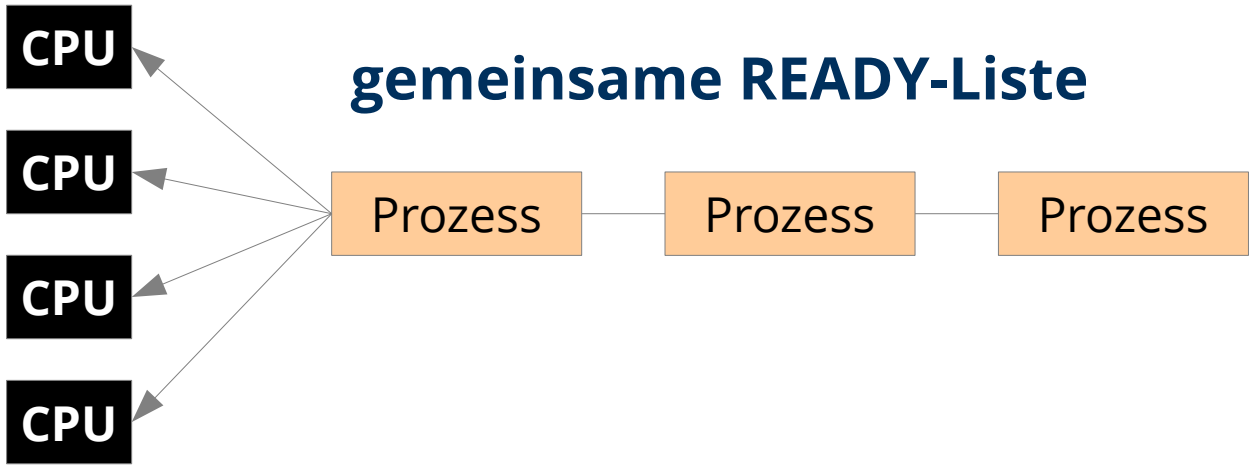
CPU-Zuteilung im Multiprozessor



oder ...



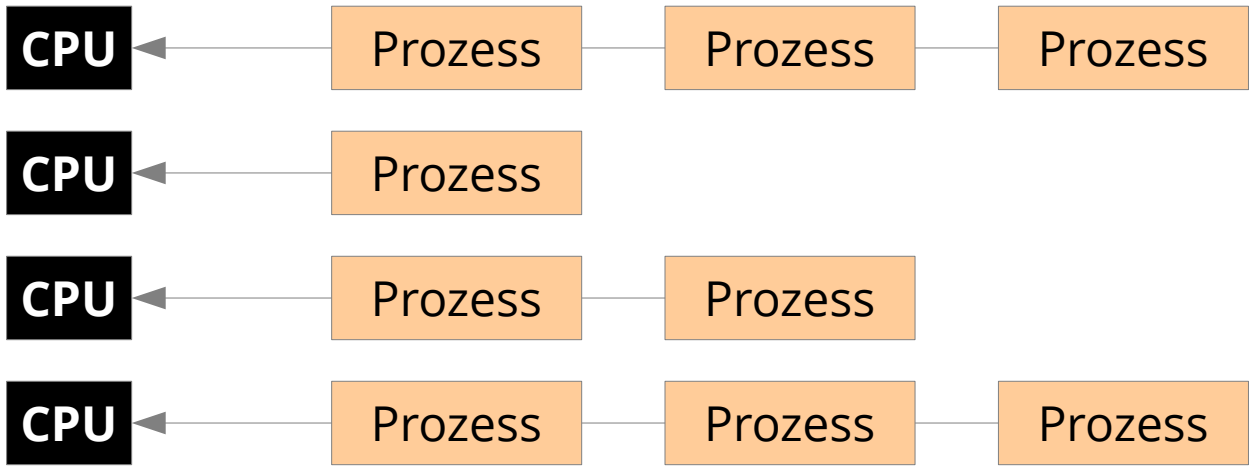
CPU-Zuteilung im Multiprozessor



- Automatischer Lastausgleich
  - Keine CPU läuft leer
- Keine Bindung von Prozessen an bestimmte CPU
- Zugriffe auf die READY-Liste müssen synchronisiert werden
  - Hoher Sperraufwand
  - Konfliktwahrscheinlichkeit wächst mit CPU-Anzahl!

CPU-Zuteilung im Multiprozessor

eine READY-Liste pro CPU



- Prozesse bleiben bei einer CPU
  - Bessere Ausnutzung der Caches
- Weniger Synchronisationsaufwand
- CPU kann leerlaufen
  - Lösung: Lastausgleich bei Bedarf
    - Wenn eine Warteschlange leer ist
    - Durch einen *Load Balancer*-Prozess

Moderne PC Betriebssysteme setzen heute getrennte READY-Listen ein.

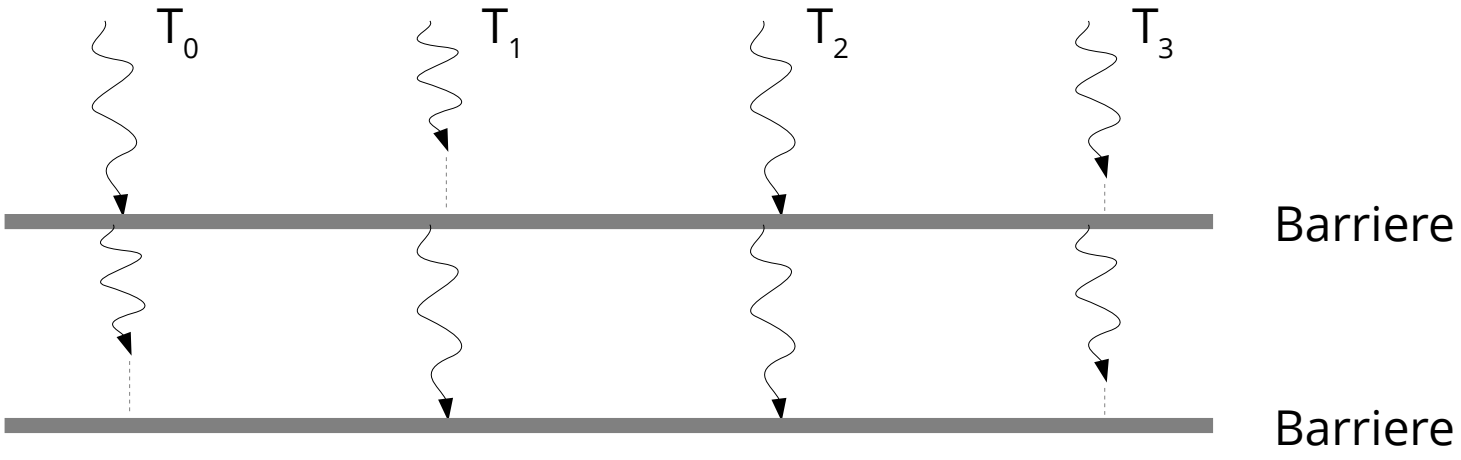
Scheduling paralleler Programme

… erfordert spezielle Strategien.

- Beispiel: ***Lock/Step*-Betrieb**

  (typisch für viele parallelen Algorithmen)

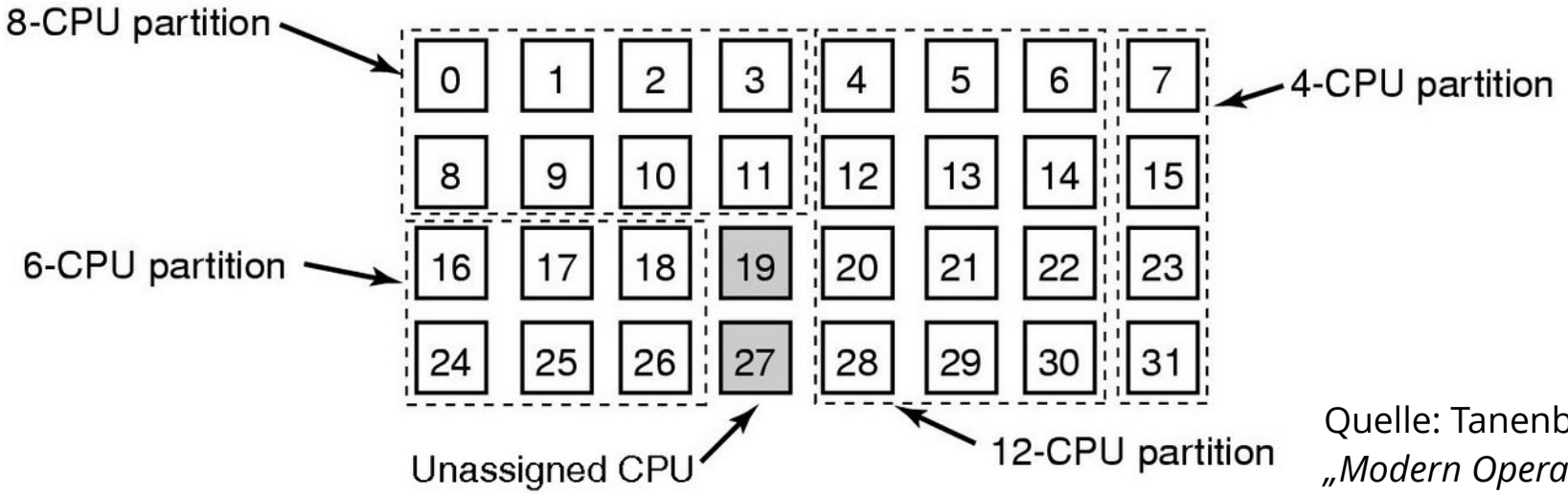
  1. Parallelen Berechnungsschritt durchführen
  2. **Barrierensynchronisation**
  3. wieder zu 1.



- Kooperierende Prozesse/Fäden sollten gleichzeitig laufen
  - Ansonsten müssen unter Umständen viele Prozesse auf einen einzelnen warten

Diskussion: *Space Sharing*

- ***Time Sharing***
  - Bei Uniprozessoren kann nur die Rechenzeit einer CPU auf Prozesse verteilt werden.
- ***Space Sharing***
  - Bei Multiprozessoren können auch Gruppen von Prozessoren vielfädigen Programmen zugeordnet werden:



Quelle: Tanenbaum, *„Modern Operating Systems“*

Gang-Scheduling

- CPU-Zuteilungsverfahren, das ***Time Sharing*** und ***Space Sharing*** kombiniert
  - Zusammengehörige Prozesse/Fäden werden als Einheit betrachtet.
    - Die „*Gang*“
  - Alle *Gang*-Mitglieder arbeiten im *Time Sharing* simultan.
  - Alle CPUs führen Prozesswechsel synchron aus.

Time slot	CPU 0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Es gibt verschiedene Algorithmen/Strategien wie BaG, AFCFS, usw., die entsprechende Pläne erzeugen.

Quelle: Tanenbaum, „Modern Operating Systems“

Inhalt

- Wiederholung
- Hardwaregrundlagen
- Anforderungen
- Synchronisation
- CPU-Zuteilung
- **Zusammenfassung**

Zusammenfassung

- Multiprozessorsysteme, Mehrrechnersysteme und Verteilte Systeme ermöglichen mehr Leistung durch Parallelverarbeitung …
  - für parallele Programme (HPC: *Number Crunching*, *Server*, …)
  - im Mehrbenutzerbetrieb
- Betriebssysteme für Multiprozessoren erfordern …
  - Prozessorsynchronisation beim Zugriff auf Systemstrukturen
  - Spezielle *Scheduling*-Verfahren
    - Eine vs. mehrere Bereitlisten mit Lastausgleich
    - *Gang*-Scheduling
- PC-Betriebssysteme müssen heute Multiprozessoren unterstützen, da *Multicore*-CPUs die Norm sind.

Literatur

[1] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. 2019. *LockDoc: Trace-Based Analysis of Locking in the Linux Kernel*. In Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19). Association for Computing Machinery, New York, NY, USA, Article 11, 1–15.
DOI: https://doi.org/10.1145/3302424.3303948