



(Grundlagen der) Betriebssysteme | E.1



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm



Bild von Mike by Pexels

E | Prozessverwaltung und Nebenläufigkeit (Grundlagen der) Betriebssysteme



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm

Überblick

Überblick der Themenabschnitte

- **A** – Organisatorisches
- **B** – Zahlendarstellung und Rechnerarithmetik
- **C** – Aufbau eines Rechnersystems
- **D** – Einführung in Betriebssysteme
- **E** – Prozessverwaltung und Nebenläufigkeit
- **F** – Dateiverwaltung
- **G** – Speicherverwaltung
- **H** – Ein-, Ausgabe und Geräteverwaltung
- **I** – Virtualisierung
- **J** – Verklemmungen
- **K** – Rechteverwaltung



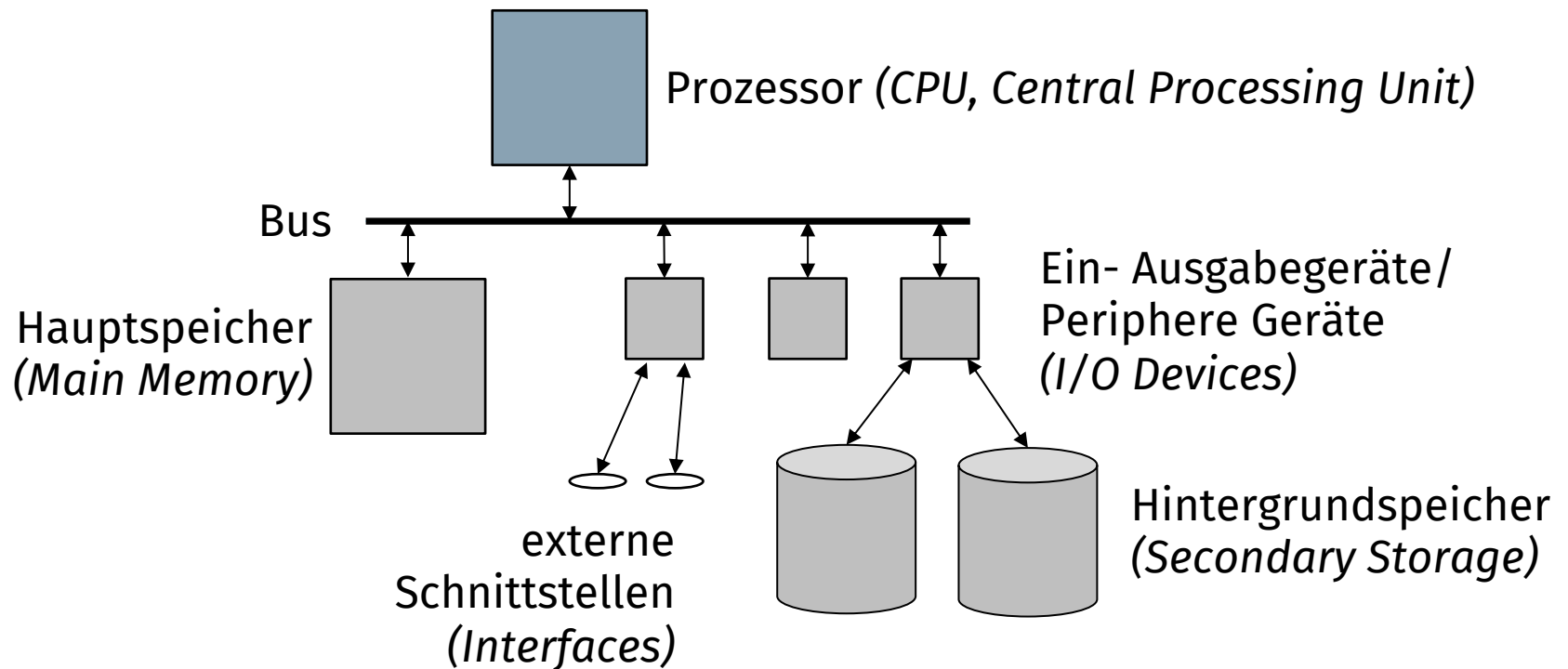
Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

Einordnung

Betroffene physikalische Ressourcen



Motivation

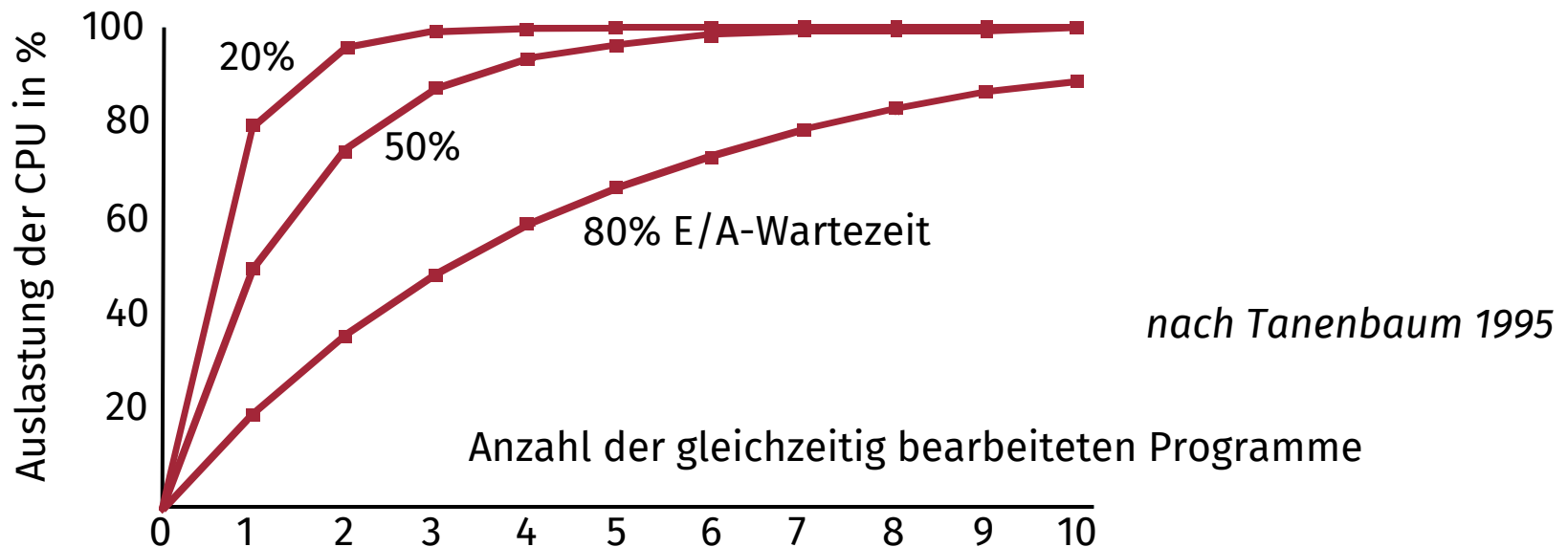
Unterstützung mehrerer gleichzeitig laufender Anwendungen

- Mehrprogramm-/**Mehrprozessbetrieb** (*Multitasking*)
 - nicht zusammengehörige Software muss nicht verbunden werden
 - einfachere Struktur großer Software-Systeme
- **Mehrbenutzerbetrieb** (*Multi-User Operation*)
 - mehrere Benutzer haben jeweils eigene laufende Anwendungen
 - ähnlich Mehrprozessbetrieb aber zusätzlich:
 - Verwaltung von Benutzern und deren Rechte

Motivation (2)

Mehrprozessbetrieb fördert Auslastung der CPU

- Wartezeiten werden von anderen Anwendungen genutzt



- Grafik für Prozesse, die 20%, 50% oder 80% ihrer Zeit warten

Terminologie

Programm

- Folge von Anweisungen und dazugehörigen Daten
 - hinterlegt z.B. als ausführbare Datei im Hintergrundspeicher

Prozess

- Programm, das sich in Ausführung befindet
 - inklusive aktuelle Daten
 - Programm kann **mehrfach** ausgeführt werden
 - mehrere gleichartige Prozesse mit evtl. unterschiedlichen aktuellen Daten

Prozess

Bestandteile

■ Speicher

- Bereiche für Code (Instruktionen) und Daten

■ Aktivitätsträger

- üblicherweise **ein** Aktivitätsträger
- startet Befehlsbearbeitung mit Erzeugung des Prozesses
- Kontext des Prozessors: Registerinhalte inkl. Programmzähler

◆ Prozess ist eine Art **virtueller Prozessor**

- führt Anweisungen des Programms aus

Prozess (2)

Bestandteile (fortges.)

- Schutzumgebung
 - Speicher und Bearbeitung ist vor anderen Prozessen geschützt
- Kontext für Ressourcenanforderungen
 - neben Speicher und Aktivitätsträger: Dateien, Semaphore, Queues, Pipes, Sockets, Geräteverbindungen etc.
 - Verwaltungseinheit für Ressourcen
- Prozesskontrollblock (*Process Control Block, PCB*)
 - Datenstruktur im Betriebssystem zur Prozessverwaltung

Prozesskontrollblock

Inhalte des PCB in UNIX/Linux:

- Prozessnummer (*Process Identifier, PID*)
- Metadaten
 - verbrauchte Rechenzeit, Erzeugungszeitpunkt
 - Eigentümer (User Identifier, UID, Group Identifier, GID) **Kap. F/K**
- Kontext (Register insbes. PC) **Kap. E**
- Speicherabbildung **Kap. G**
 - Speicherbereiche, Konfiguration der Schutzumgebung
- Wurzelverzeichnis, aktuelles Verzeichnis **Kap. F**
- Verwendete Ressourcen **Kap. F/H**
 - z.B. offene Dateien, offene Netzwerkverbindungen



(Grundlagen der) Betriebssysteme | E.2



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

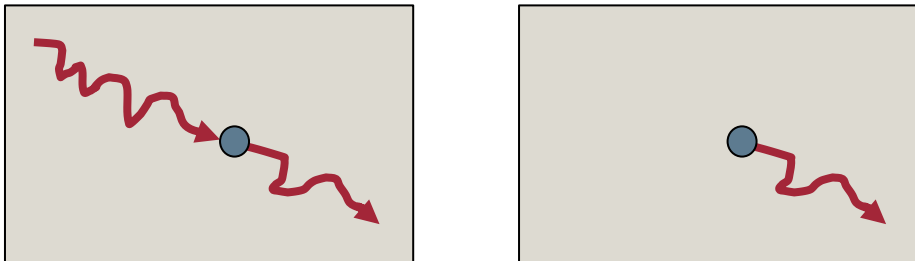
- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

Erzeugung von Prozessen

Beispiel: Linux

■ Systemaufruf `fork`

- aus einem Prozess werden zwei fast völlig identische Prozesse



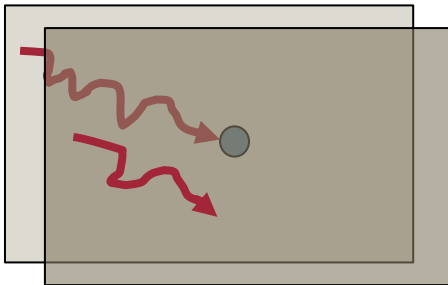
- Vater (Parent) und Kind (Child)
- Unterscheidung
 - `fork`-Aufruf im Vater liefert Kind-ID
 - `fork`-Aufruf im Kind liefert 0

Erzeugung von Prozessen (2)

Beispiel: Linux (fortges.)

■ Systemaufruf `exec`

- ein Prozess startet erneut mit anderem Programm



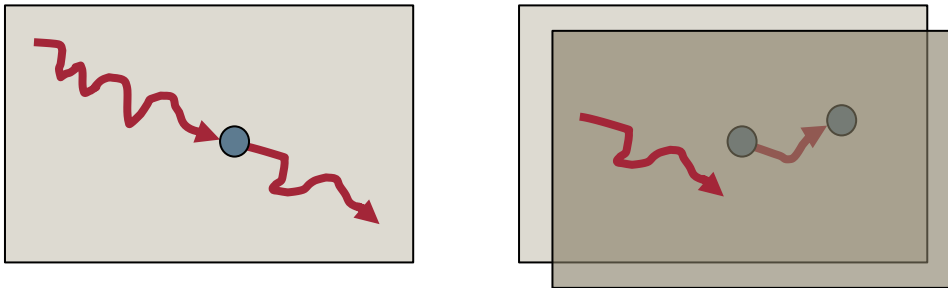
- Prozess bleibt bestehen
- führt neues Programm aus

Erzeugung von Prozessen (3)

Beispiel: Linux (fortges.)

■ Starten neuer Prozesse

- zunächst Aufruf von **fork**



- dann Aufruf von **exec**

■ Start oft durch spezielle Programme

- Shell, d.h. über Kommandozeile
- über graphische Benutzeroberfläche

Erzeugung von Prozessen (4)

Weitergabe von Informationen beim Erzeugen

- bei `fork` haben beide Prozesse dieselben Informationen
- bei `exec` werden übergeben:
 - Umgebungsvariablen
 - Name-Wert-Paare, typischerweise zur Konfiguration
 - Argumente
 - Liste von Strings als Parameter
 - Hinweis Java: `void main(String [] args)`
 - werden beim `exec` Systemaufruf übergeben und vom Betriebssystem weitergereicht

Shell

Textbasiertes Programm (Command Line Interface, CLI)

- Start und Terminierung von Prozessen
- Verwaltungsoperationen für das Gesamtsystem

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.17134.648]
(c) 2018 Microsoft Corporation. Alle Rechte vorbehalten.
C:\Users\Franz Hauck>
```

Cursor

Prompt

Shell (2)

Kommandozeile

```
C:\Users\Franz Hauck>md tmp ↵  
C:\Users\Franz Hauck>cd tmp ↵  
C:\Users\Franz Hauck\tmp> █
```

■ Beispiele für Kommandos

- **md** = erzeuge Unterverzeichnis (make directory)
- **cd** = wechsele in Verzeichnis (change directory)

■ Kommandos können Argumente besitzen

- z.B. Datei- oder Verzeichnisnamen
- mit Leerzeichen getrennt
- von Shell nach **fork** bei **exec** entsprechend übergeben

Shell (3)

Kommandozeile (fortges.)

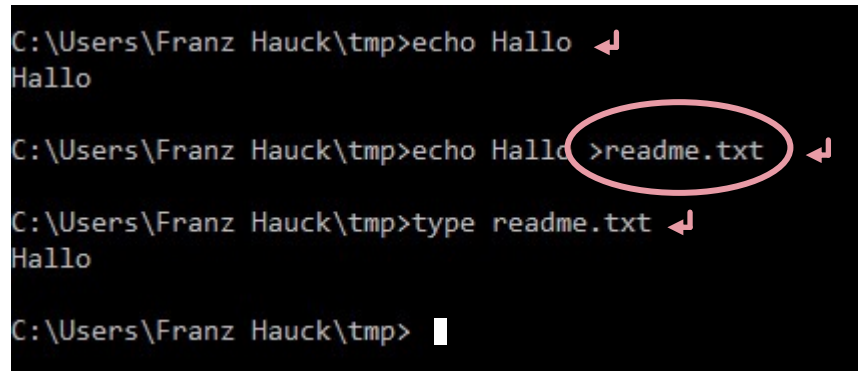
- Terminieren eines Prozesses durch **Strg+C (Ctrl+C)**
 - Terminierung kann vom Prozess unterbunden werden

```
C:\Users\Franz Hauck>c:\Programme\Microsoft Office\Office16\powerpnt ↵
```

- Kommando startet PowerPoint
 - Shell kann auch grafische Prozesse starten
 - hier Angabe des vollständigen Dateipfads zur Programmdatei
 - Shell ist konfiguriert, Programmdateien in bestimmten Verzeichnissen zu suchen
 - aber MS Office dort meist nicht registriert

Shell (4)

Kommandozeile (fortges.)



```
C:\Users\Franz Hauck\tmp>echo Hallo ↵
Hallo

C:\Users\Franz Hauck\tmp>echo Hallo >readme.txt ↵

C:\Users\Franz Hauck\tmp>type readme.txt ↵
Hallo

C:\Users\Franz Hauck\tmp> █
```

- echo-Kommando gibt Text aus
- Ausgabeumleitung in eine Datei mit „>“
- type-Kommando gibt Inhalt einer Datei aus

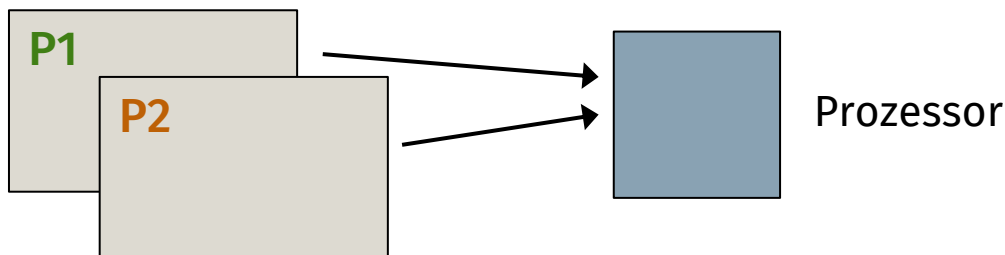
Umschalten zwischen Prozessen

Fokus Aktivitätsträger

- Befehlsstrang eines Prozesses
- Problem: mehr Aktivitätsträger als Prozessoren
 - Ressourcenzuteilung nötig

Umschaltung von einem Prozess zum anderen (Kontextwechsel)

- Annahme: nur ein Prozessor, zwei Prozesse
 - laufender Prozess muss seinen Kontext sichern: alle Register
 - neuer Prozess muss Kontext laden: alle Register



Umschalten zwischen Prozessen (2)

Abgabe der CPU in Prozess 1

```
...  
MOV R0, $10  
MOV R1, $11  
MOV R2, $12  
MOV CCR, $13  
MOV #L1cont, $14  
JMP Lsw2
```

L1cont:

...

- Abgabeinstruktionen für Prozess 2 analog
- \$10-\$14 Kontext Prozess 1
- \$20-\$24 Kontext Prozess 2

Umschalter nach Prozess 2

Lsw2:

```
MOV $20, R0  
MOV $21, R1  
MOV $22, R2  
MOV $23, CCR  
JMP [$24]
```

Abgabe in Prozess 2

```
...  
MOV R0, $20  
MOV R1, $21  
MOV R2, $22  
MOV CCR, $23  
MOV #L2cont, $24  
JMP Lsw1
```

L2cont:

...

Umschalten zwischen Prozessen (3)

Essenz des Umschaltens

- Wechsel der Registerinhalte
 - einschließlich PC
- Prozess muss Wechsel **nicht bemerken**
 - wird lediglich langsamer

Beispiel war stark vereinfacht

- Umschalter kann nur auf bestimmten Prozess umschalten
 - in der Realität vorherige **Suche** nach geeignetem Prozess
- **Ablaufumgebung** des Prozesses muss eingerichtet werden
 - z.B. Schutzumgebungen
 - z.B. Initialisierung von Betriebssystemvariablen (aktueller Prozess etc.)

Kooperatives Multitasking

Prozesse geben freiwillig CPU ab

- Umschaltung nicht transparent
- Beispiel: **yield**
 - Systemaufruf zur Abgabe des Prozessors
 - Auswahl des nächsten Prozesses durch das Betriebssystem
- Beispiel: **Co-Routinen**
 - Prozess gibt Prozessor ab und bestimmt nächsten Prozess
 - meist **yield**-Aufruf mit neuer Prozess-ID als Parameter
 - keine Fairness
 - Strategie in die Anwendung „eingebaut“

Umschaltung im Betriebssystem

Scheduler-Komponente im Betriebssystem

- verwaltet aktuelle Prozesse
- schaltet gemäß einer Strategie um

Eingriffsmöglichkeiten des Schedulers

- **Systemaufruf**
 - Prozesse wechselt freiwillig ins Betriebssystem
- **Unterbrechung**
 - Prozess wechselt „unfreiwillig“ ins Betriebssystem



Bild von Mike by Pexels

(Grundlagen der) Betriebssysteme | E.3



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

Prozesszustände

Ein Prozess befindet sich in einem der folgenden Zustände

- **Erzeugt** (New)

- Prozess ist erzeugt, besitzt aber noch nicht alle nötigen Ressourcen

- **Bereit** (Ready)

- Prozess besitzt alle nötigen Ressourcen und ist bereit zum Laufen

- **Laufend** (Running)

- Prozess wird auf einem realen Prozessor ausgeführt

Prozesszustände (2)

Ein Prozess befindet sich in einem der folgenden Zustände (fortg.)

■ Blockiert (Blocked/Waiting)

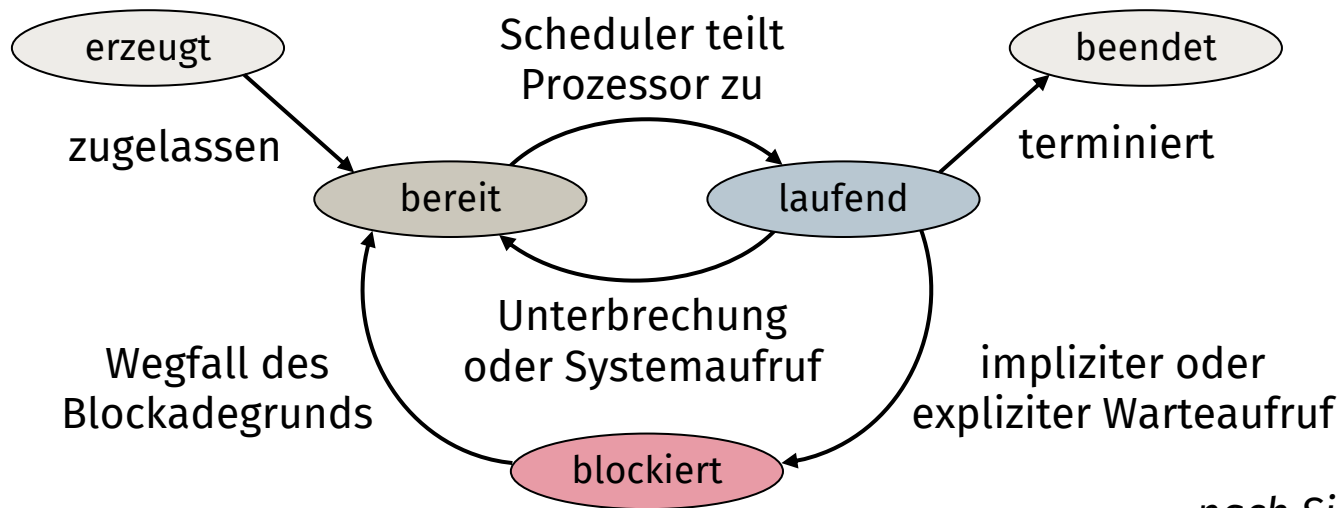
- Prozess wartet auf ein Ereignis und wird dazu blockiert
 - z.B. auf Fertigstellung einer I/O-Operation, Zuteilung einer Ressource, Empfang einer Nachricht

■ Beendet (Terminated)

- Prozess ist beendet, einige Ressourcen sind aber noch nicht freigegeben
- Prozess muss aus anderen Gründen im System verbleiben
 - z.B. bis sein Status dem Vaterprozess bekannt wird

Prozesszustände (3)

Zustandsdiagramm



nach Silberschatz 1994

- Scheduler ist Teil des Betriebssystems
 - nimmt Zuteilung des realen Prozessors vor
 - bestimmt laufende Prozesse aus der Menge der bereiten

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

Auswahlstrategien

Eingriffsmöglichkeiten im Detail

1. Zustandswechsel von **laufend** nach **blockiert** muss
 - z.B. Warten auf Platte, Netzwerk, Tastur, Maus etc. (I/O)
2. Zustandswechsel von **laufend** nach **bereit** kann
 - z.B. in oder nach einer Unterbrechung
 - z.B. in oder nach einem Systemaufruf
3. Zustandswechsel von **blockiert** oder **erzeugt** nach **bereit** kann
 - z.B. I/O-Ereignis eingetreten oder alle Ressourcen verfügbar
4. Zustandswechsel von **laufend** nach **terminiert** muss
 - in der Regel aktiver Vorgang aus Zustand laufend wg. Aufräumarbeiten im Prozess

Auswahlstrategien (2)

Strategien mit Auswahl in Kann-Situationen

- **verdrängende** bzw. **präemptive** Strategien (preemptive)
 - Prozess kann unfreiwillig abgelöst werden

Strategien ohne Auswahl in Kann-Situationen

- **nicht-verdrängende** bzw. **nicht-präemptive** Strategien (non-preemptive)
 - Prozess läuft bis er freiwillig den Prozessor abgibt

Kriterien für Scheduling-Strategien

CPU-Auslastung

- Möglichst zu 100% ausgelastete CPU

Durchsatz

- Möglichst hohe Zahl bearbeiteter Prozesse pro Zeiteinheit

Verweilzeit

- Möglichst geringe Gesamtzeit eines Prozesses im System

Wartezeit

- Möglichst kurze Gesamtzeit im Zustand „bereit“

Antwortzeit

- Möglichst kurze Reaktionszeit im interaktiven Betrieb



(Grundlagen der) Betriebssysteme | E.4



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

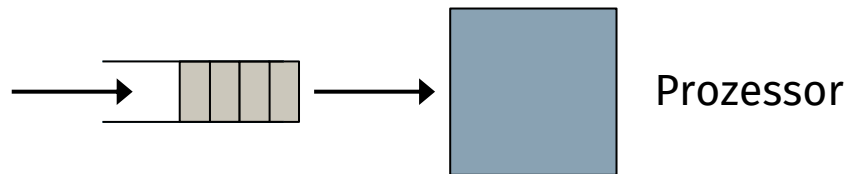
First Come, First Served

Der erste Prozess wird zuerst bearbeitet (*FCFS*)

- „Wer zuerst kommt...“
- nicht-präemptiv

Implementierung (Monoprozessor)

- Warteschlange für Prozesse im Zustand **bereit**
- Prozesse werden hinten eingereiht
- Prozesse werden vorne entnommen

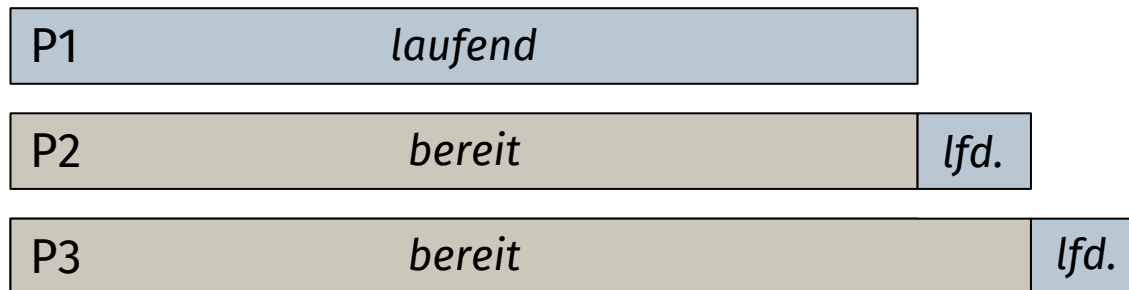


First Come, First Served (2)

Beispielablauf #1

- Prozess 1: 24
 - Prozess 2: 3
 - Prozess 3: 3
- } Zeiteinheiten

- Reihenfolge des Eintreffens: P1, P2, P3



- Mittlere Wartezeit: $(0 + 24 + 27)/3 = 17$

First Come, First Served (3)

Beispielablauf #2

- Prozess 1: 24
 - Prozess 2: 3
 - Prozess 3: 3
- } Zeiteinheiten

- Reihenfolge des Eintreffens: P2, P3, P1



- Mittlere Wartezeit: $(6 + 0 + 3)/3 = 3$

First Come, First Served (4)

Prozesse können blockieren

- Blockierung macht Prozessor frei
 - Neuzuteilung aus der Spitze der Warteschlange
- Deblockierung fügt Prozess ans Ende der Warteschlange wieder ein

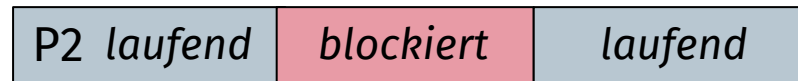
First Come, First Served (5)

Beispielablauf #3

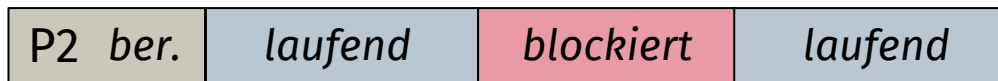
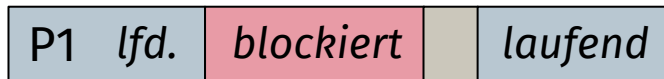
- Bedarf Prozess 1:



- Bedarf Prozess 2:



- Reihenfolge des Eintreffens: P1, P2



P1
P2

P2

P1

P2

First Come, First Served (6)

Bewertung

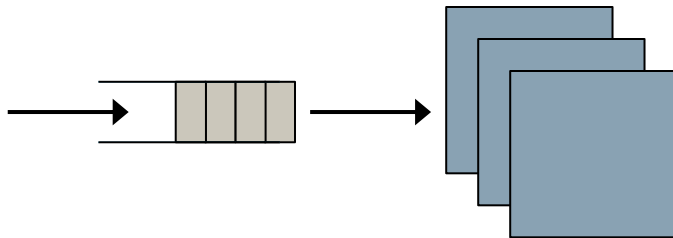
- fair (?)
- Wartezeiten nicht minimal
 - „Konvoi-Effekt“: Stau hinter lang laufenden Prozessen
- nicht für Time-Sharing- oder interaktiven Betrieb geeignet

First Come, First Served (7)

Implementierung für mehrere Prozessoren

■ für n Prozessoren:

- n ersten Einträge der Warteschlange werden laufend
- wird Prozessor frei, wird nächster aus Warteschlange zugeteilt

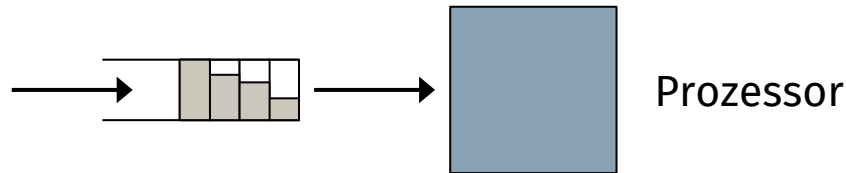


Shortest Job First

Kürzester aller bereiter Jobs wird ausgewählt (SJF)

■ **Bereit**-Warteschlange wird nach Länge der nächsten Rechenphase sortiert

- erster in Warteschlange wird zugeteilt



- Länge bezieht sich auf die anstehende Rechenphase bis zur nächsten Warteoperation (z.B. I/O)
- benötigt Kenntnis über tatsächliche Laufzeit oder Vorhersage (dann aber evtl. ungenau)
- Vorhersage der Länge z.B. durch Protokollieren der Länge bisheriger Rechenphasen (Mittelwert, exponentielle Glättung)

Shortest Job First (2)

Weitere Eigenschaften

- SJF optimiert die mittlere Wartezeit
 - bei ungenauer Vorhersage nicht optimal
- Varianten: **präemptiv** (PSJF) und **nicht-präemptiv**

Shortest Job First (3)

Beispielablauf SJF (nicht verdrängend)

- Bedarf Prozess 1:

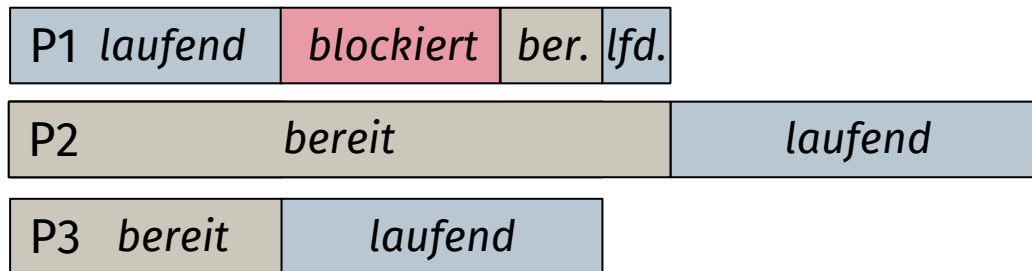
P1 <i>laufend</i>	<i>blockiert</i>	<i>lfd.</i>
-------------------	------------------	-------------
- Bedarf Prozess 2:

P2 <i>laufend</i>

- Bedarf Prozess 3:

P3 <i>laufend</i>

- Ablauf auf einem Monoprozessor



P1
P3
P2

P3
P2

P1
P2

Shortest Job First (4)

Beispielablauf PSJF (verdrängend)

- Bedarf Prozess 1:

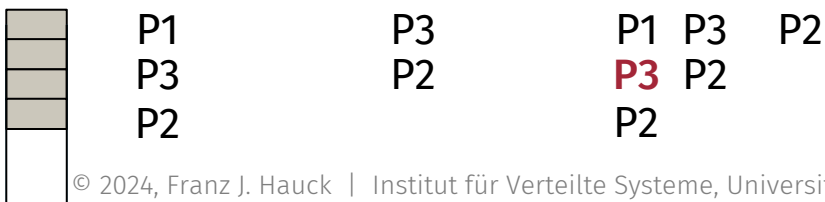
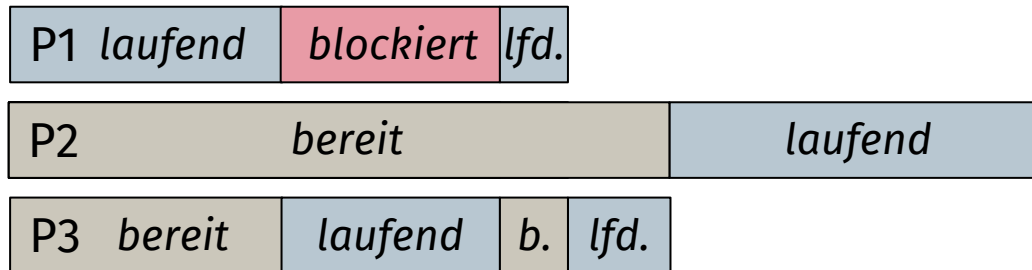
P1 <i>laufend</i>	<i>blockiert</i>	<i>lfd.</i>
-------------------	------------------	-------------
- Bedarf Prozess 2:

P2 <i>laufend</i>

- Bedarf Prozess 3:

P3 <i>laufend</i>

- Ablauf auf einem Monoprozessor

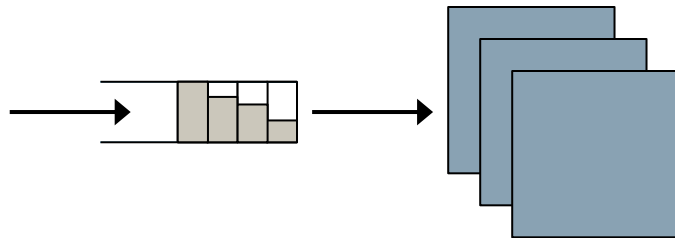


Shortest Job First (5)

Implementierung für mehrere Prozessoren

■ für n Prozessoren:

- n ersten Einträge der Warteschlange werden laufend
- wird Prozessor frei, wird nächster aus Warteschlange zugeteilt



■ in der verdrängenden Variante:

- bei Deblockierung: alle Prozesse zurück in die Warteschlange und neu zuteilen
- kann typischerweise stark optimiert werden



(Grundlagen der) Betriebssysteme | E.5



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

Highest Priority First

Auswahl des Prozesses mit höchster Priorität (*HPF*)

- Sortierung der **Bereit**-Warteschlange nach Prioritäten
 - Priorität typischerweise ganzzahliger Wert
 - unterschiedliche Ordnungen möglich
 - kleinerer Wert oder höherer Wert hat höhere Priorität
- SJF ist HPF
 - kürzere Rechenzeit entspricht höherer Priorität

Highest Priority First (2)

Varianten

- präemptiv – nicht präemptiv
 - mögliche Verdrängung des laufenden Prozesses, wenn andere Prozesse bereit werden
- statische Prioritäten – dynamische Prioritäten
 - SJF hat dynamische Prioritäten
 - in Echtzeitsystemen häufig statische Prioritäten
 - Vorhersagbarkeit der Reaktionszeiten

Highest Priority First (3)

Problem Aushungerung (*Starvation*)

- Prozess wird nie laufend, weil immer höher priorer Prozesse verfügbar

Lösung

- dynamische Anhebung der Priorität für lange wartende Prozesse (Alterung, *Aging*)

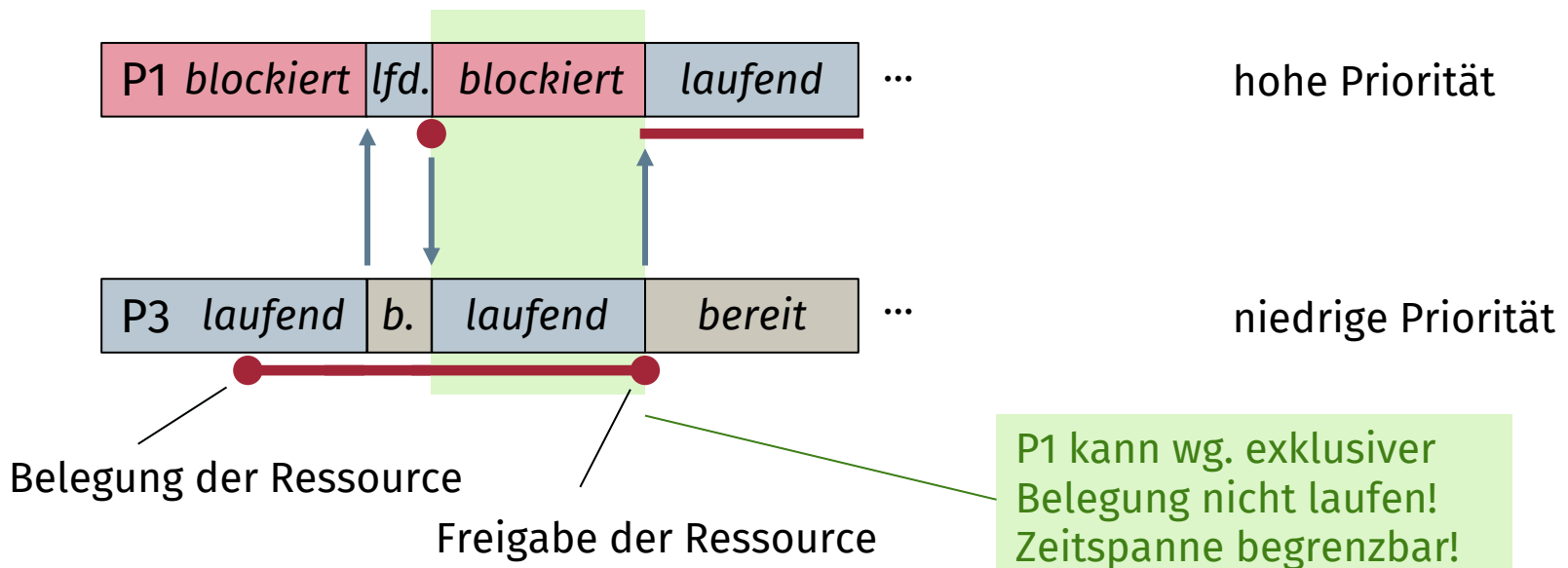
Highest Priority First (4)

Problem Prioritätenumkehr (*Priority Inversion*)

■ Szenario

- hochpriorer Prozess möchte exklusive Ressource, die ein niederpriorer Prozess momentan besitzt (z.B. Zugang zu Drucker)

■ normaler Ablauf



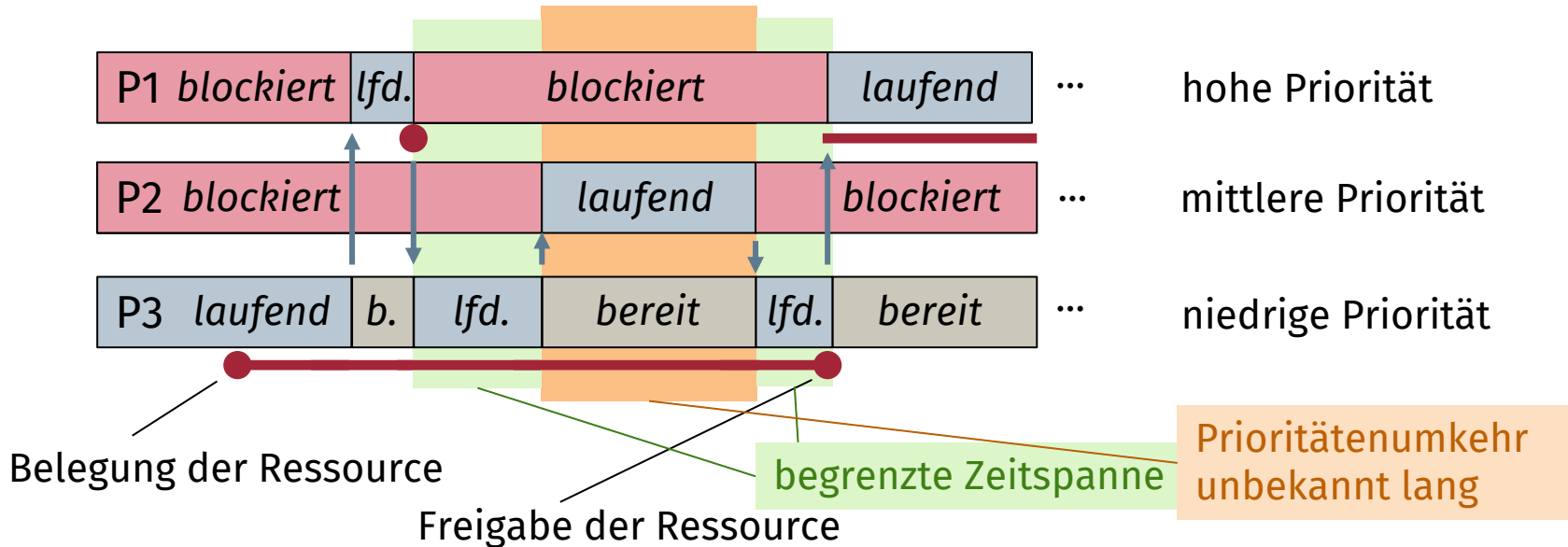
Highest Priority First (5)

Problem Prioritätenumkehr (*Priority Inversion*)

■ Szenario

- hochpriorer Prozess möchte exklusive Ressource, die ein niederpriorer Prozess momentan besitzt (z.B. Zugang zu Drucker)

■ **problematischer** Ablauf



Highest Priority First (6)

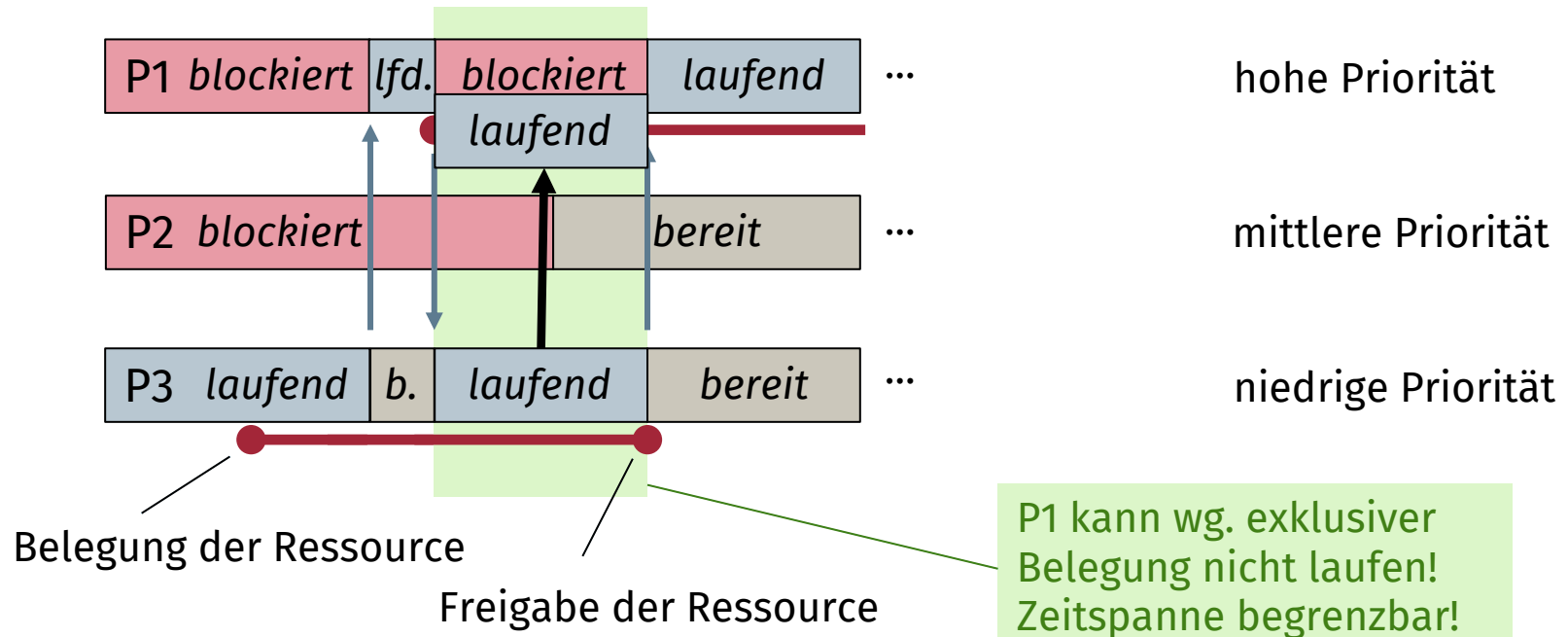
Zusammenfassung Prioritätenumkehr

- hochpriorer Prozess möchte exklusive Ressource
- niedrigpriorer Prozess besitzt diese
- kann sie aber nicht freigeben, weil mittelpriorer Prozess den niederprioreren verdrängt
- **Prioritätenumkehr**
 - im Beispiel: P2 läuft, obwohl P1 höhere Priorität hat
 - im Prinzip beliebig langer Zeitraum

Highest Priority First (7)

Lösung zur Prioritätenumkehr

- dynamisches Anheben der Priorität für Prozesse, die exklusive Ressourcen besitzen, auf die hochprioritäre Prozesse waren
 - im Beispiel: P3 bekommt temporär gleiche Priorität wie P1





(Grundlagen der) Betriebssysteme | E.6



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

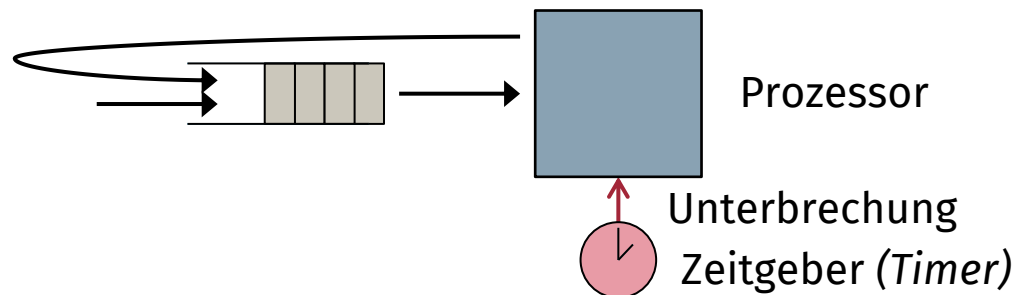
Round Robin

Zuteilung erfolgt reihum (RR)

- ähnlich wie FCFS aber mit automatischer Verdrängung nach bestimmtem Zeitintervall
 - Zeitquant (*Time Quantum*), Zeitscheibe (*Time Slice*)

Implementierung (Monoprozessor)

- Warteschlange für Prozesse im Zustand **bereit**
 - Prozesse werden hinten eingereiht, vorne entnommen
 - nach Ablauf der Zeitscheibe wird Prozess erneut eingereiht

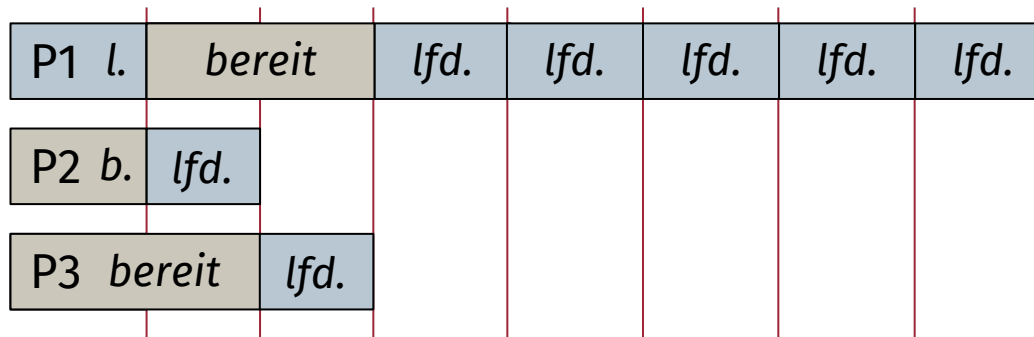


Round Robin (2)

Beispielablauf #1

- Prozess 1: 24
 - Prozess 2: 3
 - Prozess 3: 3
- } Zeiteinheiten

- Reihenfolge des Eintreffens: P1, P2, P3
- Zeitscheibe: 4 Zeiteinheiten



- Mittlere Wartezeit: $(6 + 4 + 7)/3 = 5,7$

Round Robin (3)

Abbruch der Zeitscheibe

- muss Prozess den Prozessor verlassen, wird Zeitscheibe abgebrochen
 - z.B. bei Blockierung oder Terminierung

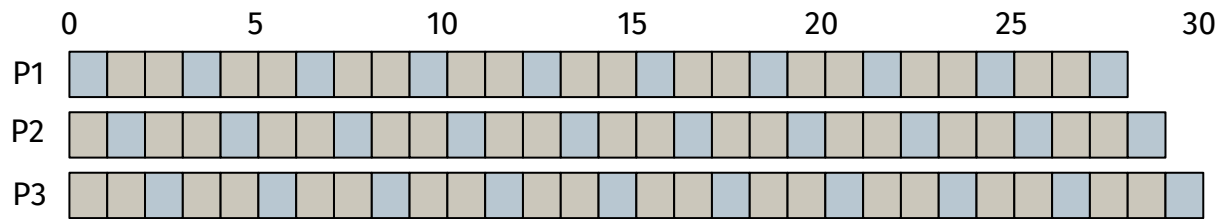
Länge der Zeitscheibe

- **kurze** Zeitscheibe: ständige Kontextwechsel
 - kostet Zeit
 - Prozessor dauernd mit Umschaltungen beschäftigt
- **lange** Zeitscheibe: Annäherung an FCFS
 - mit all den Nachteilen von FCFS
- Länge hat auch Einfluss auf mittlere Verweil- und Wartezeit

Round Robin (4)

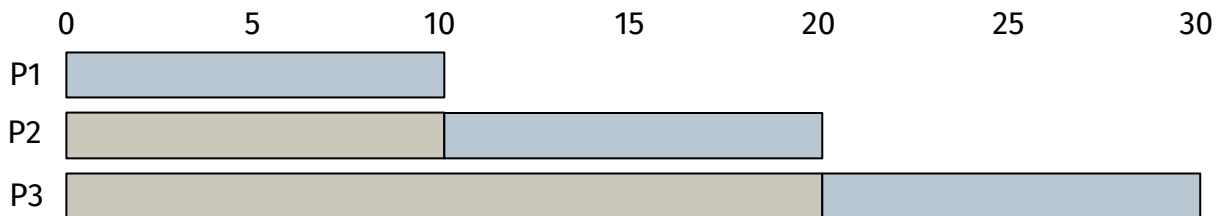
Beispiel Rechenzeit je 10 Zeiteinheiten

■ Beispielablauf Zeitscheibenlänge 1



- mittlere Verweilzeit: $(28 + 29 + 30)/3 = 29$ Zeiteinheiten
- mittlere Wartezeit: $(18 + 19 + 20)/3 = 19$ Zeiteinheiten

■ Beispielablauf Zeitscheibenlänge 10



- mittlere Verweilzeit: $(10 + 20 + 30)/3 = 20$ Zeiteinheiten
- mittlere Wartezeit: $(0 + 10 + 20)/3 = 10$ Zeiteinheiten

Round Robin (5)

Bewertung

- geeignet für Time-Sharing-Betrieb
 - gleichmäßige Verteilung der Rechenzeit
- fair (?)
- Wartezeit kann lang sein
 - hängt von Anzahl der bereiten Prozesse und der Zeitscheibenlänge ab
 - nicht gut für interaktiven Betrieb
- blockierende Prozesse benachteiligt
 - nutzen Zeitscheibe nicht aus, müssen wieder hinten anstehen

Strategien für Desktop/Laptop-Systeme

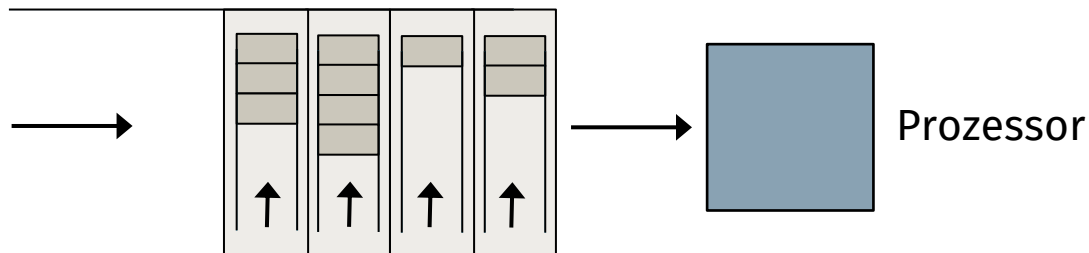
Ausgleich zwischen

- **langlaufenden** Prozessen
 - sollen viel Rechenzeit bekommen
- **interaktiven** Prozessen
 - sollen schnell laufend werden, wenn bereit
 - d.h. sobald Blockade-Ereignis eingetreten, soll Prozess aktiv werden
- **Beachte:** Prozesse können zwischen beiden Modi wechseln!
- ◆ Bisherige Strategien ungeeignet!

Multi-Level Queue Scheduling

Scheduling auf zwei Ebenen (MLQ)

- **erste Ebene:** Warteschlange mit Strategie (FCFS, SJF, HPF, RR, ...)



- **zweite Ebene:** Elemente in Warteschlange sind Warteschlangen
 - mit eigener Strategie, evtl. sogar pro Eintrag (FCFS, SJF, HPF, RR, ...)
 - Warteschlangen als Schedulingklassen

Multi-Level Queue Scheduling (2)

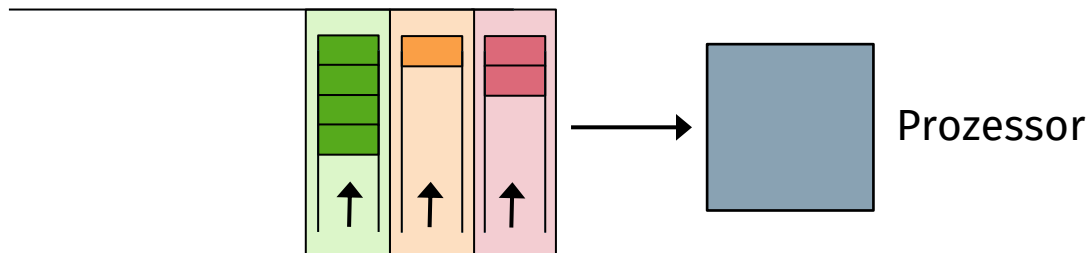
Beispiel Solaris

- **erste Ebene:** präemptives HPF
 - Schedulingklassen haben statische Prioritäten (absteigend):
 - **Systemprozesse**
 - systeminterne, kurze Prozesse z.B. für Unterbrechungsbehandlung
 - **Echtzeitprozesse**
 - Echtzeitanwendungen
 - **normale Prozesse**
- **zweite Ebene:** individuelle Strategie
 - **Systemprozesse:** FCFS
 - **Echtzeitprozesse:** präemptives HPF, dynamisch änderbare Prioritäten
 - **normale Prozesse:** MLFQ (siehe später)

Multi-Level Queue Scheduling (3)

Beispiel Solaris (fortges.)

- erste Ebene: präemptives HPF

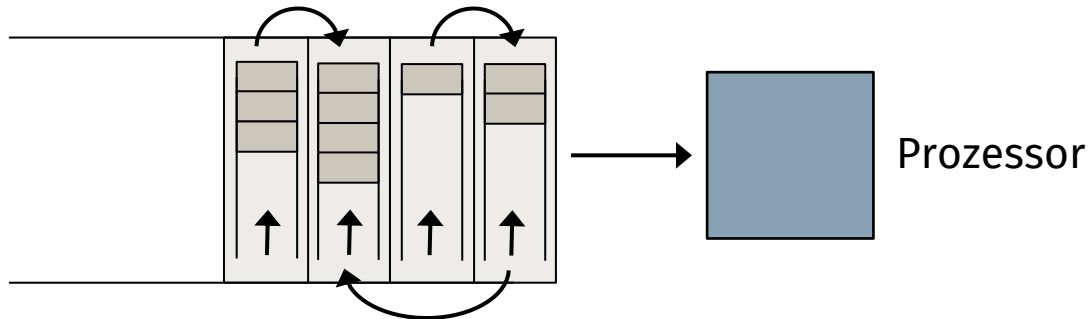


- zweite Ebene:
 - Systemprozesse: FCFS
 - Echtzeitprozesse: präemptives HPF
 - normale Prozesse: MLFQ

Multi-Level Feedback Queue Scheduling

Scheduling über Schedulingklassen hinweg (MLFQ)

- **erste Ebene:** präemptives HPF



- **zweite Ebene:** viele Klassen mit RR Strategie
 - evtl. unterschiedliche Zeitscheibenlänge
- mögliche Wechsel der Schedulingklasse
 - nach **Verdrängung**
 - bei **Deblockierung**
 - bei **Alterung**

Multi-Level Feedback Queue Scheduling (2)

Beispiel: Virtual Round Robin (VRR)

- 2 Scheduling-Klassen (Prioritäten)
 - bevorzugte Klasse
 - normale Klasse (hier starten alle Prozesse)
- Transfer zwischen Klassen/Prioritäten
 - **Verdrängung:** Prozess braucht Zeitscheibe auf
 - rechnet lang → zurück in normale Klasse
 - **Deblockierung:** Warteereignis eingetreten
 - interaktiv → Start aus bevorzugter Klasse
 - z.B. Zeitscheibe ist Rest aus vorheriger Zuteilung
 - bevorzugt interaktive Prozesse

Multi-Level Feedback Queue Scheduling (3)

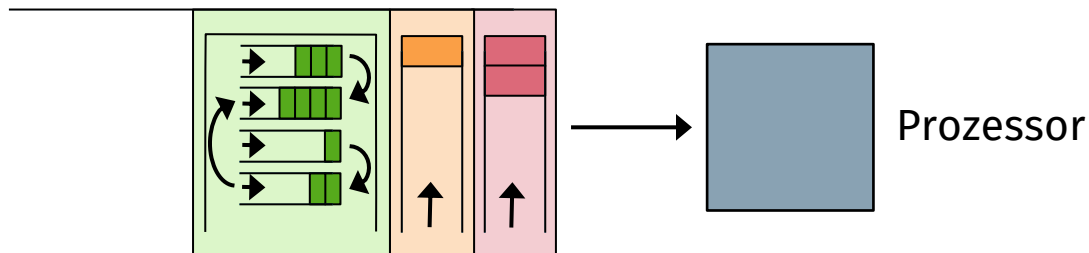
Beispiel: Solaris

- 60 Scheduling-Klassen (Prioritäten)
 - RR mit unterschiedlichen Zeitscheibenlängen
- Transfer zwischen Klassen/Prioritäten
 - **Verdrängung**: Prozess braucht Zeitscheibe auf
 - rechnet lang → **Priorität sinkt**
 - benachteiligt Langläufer
 - **Deblockierung**: Warteereignis eingetreten
 - interaktiv → **Priorität steigt**
 - bevorzugt interaktive Prozesse
 - **Alterung**: Prozess lange im Zustand bereit
 - alt → **Priorität steigt**
 - bevorzugt lange wartende Prozesse

Multi-Level Feedback Queue Scheduling (3)

Beispiel: Solaris (vervollständigt)

■ erste Ebene: HPF



■ zweite Ebene:

- Systemprozesse: FCFS
- Echtzeitprozesse: präemptives HPF
- normale Prozesse: MLFQ

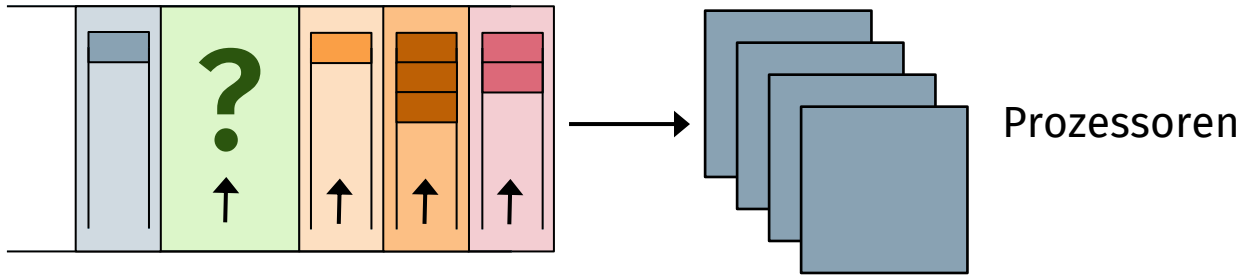
■ dritte Ebene:

- MLFQ RR Scheduling-Klassen

Linux Scheduling

MLQ Scheduling

- verschiedene Schedulingklassen ähnlich Solaris
- erste Ebene: HPF



- zweite Ebene:
 - **Systemprozesse:** FCFS
 - **Echtzeitprozesse:** EDF (*Earliest Deadline First*)
 - **Echtzeitprozesse:** präemptives HPF (*100 Prioritäten*)
 - **normale Prozesse:** CFS oder MuQSS (*40 Prioritäten*)
 - **Leerlaufprozess:** nur einer, ständig bereit

Linux Scheduling (2)

Scheduling-Varianten (*Policies*)

- **SCHED_DEADLINE** – Echtzeitprozesse mit Fristen
 - dynamische Prioritäten
- **SCHED_FIFO** – Echtzeitprozesse ohne Zeitscheiben
 - verdrängendes HPF
- **SCHED_RR** – Echtzeitprozess mit Zeitscheiben
 - verdrängendes HPF mit Zeitscheiben
- **SCHED_NORMAL** – normale Prozesse
- **SCHED_BATCH** – normale Prozesse ohne interaktive Phasen
- **SCHED_IDLE** – Variante falls keine bereiten Prozesse vorhanden
 - Leerlaufprozess, der typischerweise die CPU abschaltet

Linux Scheduling (3)

Completely Fair Scheduler (CFS)

- Seit 2007 im Linux-Kernel
- Idee
 - ideale Zuteilung der Prozessoren an vorhandene Prozesse
 - bei n Prozessen und p Prozessoren bekommt jeder p/n Anteile
 - zwischen zwei benachbarten Prioritätsebenen 10% Differenz
- Implementierung
 - Timer-Unterbrechung (1000 mal pro Sekunde)
 - verschiedene virtuelle Uhren zur Beobachtung der Abläufe
 - Lauf- und Blockiert-Zeiten
 - pro Prozessor eine eigene Bereit-Liste
- ◆ gut für Server und viele Cores geeignet

Linux Scheduling (4)

Multiple Queue Skiplist Scheduler (*MuQSS*)

- Patches von Con Kolivas für Desktops
 - Nachfolger des BFS (*Brain Fuck Schedulers*)
 - jetzt mit einer Bereitliste pro Prozessor
 - Einsatz von Skiplists
 - effiziente Listenstruktur zur Suche (mehrere Verkettungen)
- Idee
 - Zeitscheibe und virtuelle Frist pro Prozess (abhängig von Priorität)
 - frühere Frist zuerst (wie EDF)
- Anspruch
 - bessere Reaktionszeit für Desktop-Anwendungen



(Grundlagen der) Betriebssysteme | E.7



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

Prozesse und Aktivitätsträger

Bisher betrachtet:

- ein Prozess mit **einem** Aktivitätsträger (*Thread*)

Gute Gründe für mehrere Aktivitätsträger pro Prozess

- Aufteilung auf mehrere **gleichzeitige** Aufgaben
 - z.B. mehrere Fenster
 - z.B. Netzwerkaufgaben neben Benutzerinteraktion
 - z.B. viele gleichzeitige Anfragen in Server-Anwendungen
- **Ausnutzung** von Mehrkernprozessoren und -systemen
 - echte Parallelität bei nur einem Prozess

Prozesse und Aktivitätsträger (2)

Gute Gründe für mehrere Aktivitätsträger pro Prozess (fortges.)

- in der Regel **übersichtlicherer** Code
 - aber Problem der **Koordinierung** (siehe später)
- Prozess als Hülle für **gemeinsame** Ressourcen der Threads
 - gleiche Daten und gleicher Code (Speicher)
 - Schutzumgebung
 - Umschaltung zwischen Threads im gleichen Prozess ist **effizienter** als zwischen Threads verschiedener Prozesse
 - gemeinsam genutzte Dateien, Netzwerkverbindungen etc.

Kernel-level Threads

Moderne Betriebssysteme

- Systemaufrufe zum Erzeugen weiterer Aktivitätsträger
 - so genannte **Kernel-level Threads**
 - sind dem Betriebssystem (Kernel) bekannt
- Scheduling durch **Betriebssystem-Scheduler**
 - z.B. Linux, Windows, Solaris u.v.a.
- einige weitere Systemaufrufe für
 - Änderung der Prioritätsebenen
 - Anhalten und Löschen der Aktivitätsträger

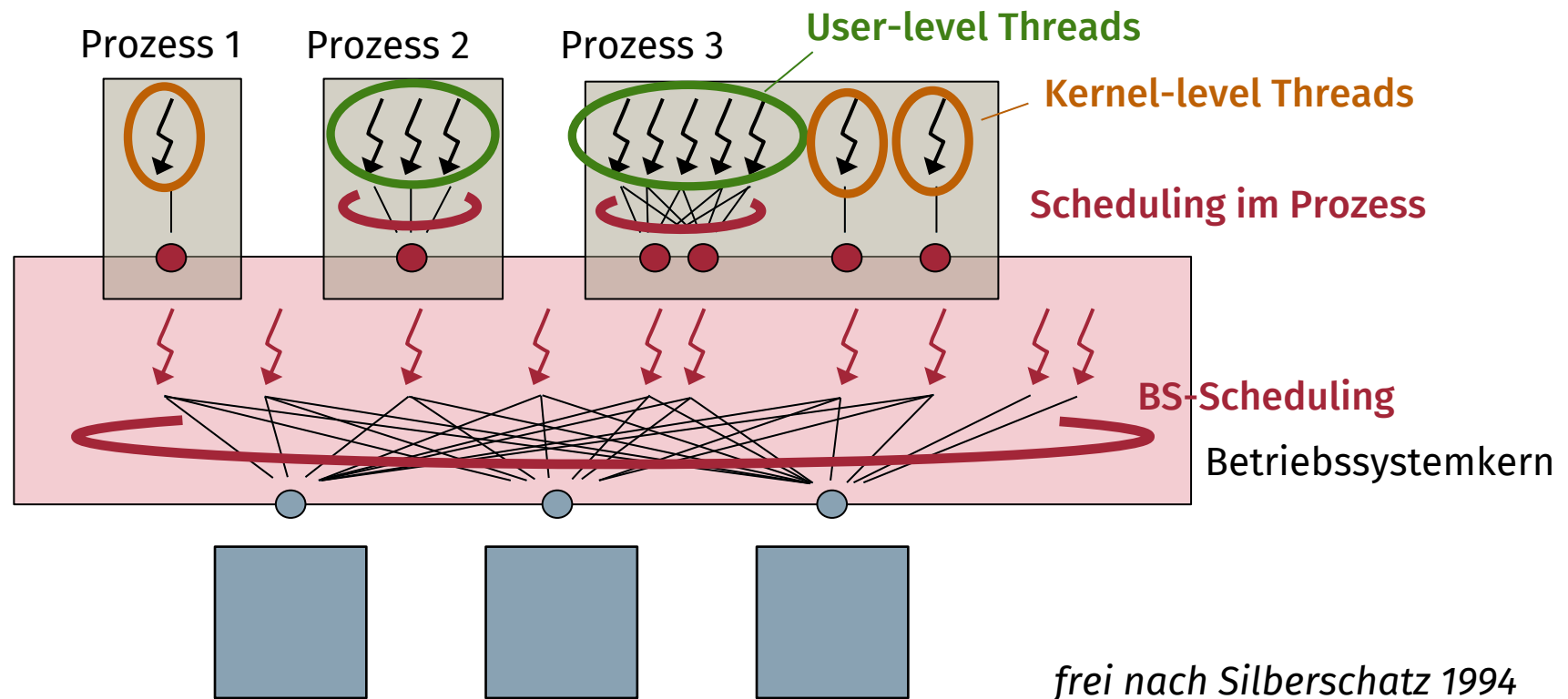
User-level Threads

Scheduling in der Anwendung

- Prozess (mit einem Thread) als virtueller Prozessor
 - im Prozess Umschaltung der Register zwischen verschiedenen virtuellen Aktivitätsträger
 - so genannte User-level Threads
- Prozess mit mehreren Threads als virtueller Mehrkernprozessor
 - Umschaltung der User-level Threads auf verschiedene Kernel-level Threads
 - ähnlich wie ein BS-Scheduler im Mehrprozessorsystem
- Beispiele
 - Java Green Threads, Windows Fibers etc.

Kernel- vs. User-level Threads

Blick auf ein modernes Betriebssystem



Kernel- vs. User-level Threads (2)

Blockierung

- blockierte User-level Threads **behindern** andere Threads
 - Aktivitätsträger bleibt im Betriebssystem „hängen“
 - bei Kernel-level Threads arbeiten alle unabhängig

Parallelität

- User-level Threads sind **nur so parallel** wie die Zahl der darunterliegenden Threads im Kernel

Kernel- vs. User-level Threads (3)

Umschaltung

- User-level Threads können **schneller** umgeschaltet werden
 - kein Wechsel ins Betriebssystem nötig

Scheduling-Strategie

- User-level Threads mit jeder **beliebigen** Strategie umschaltbar
 - auf Betriebssystem-Scheduler üblicherweise wenig Einfluss
- bei Kernel-level Threads **Fairness** notwendig
 - Prozesse mit wenig vs. Prozesse mit vielen Aktivitätsträgern



Bild von Mike by Pexels

(Grundlagen der) Betriebssysteme | E.8



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

Nebenläufigkeit

Mehrere Aktivitätsträger (Prozesse oder Threads)

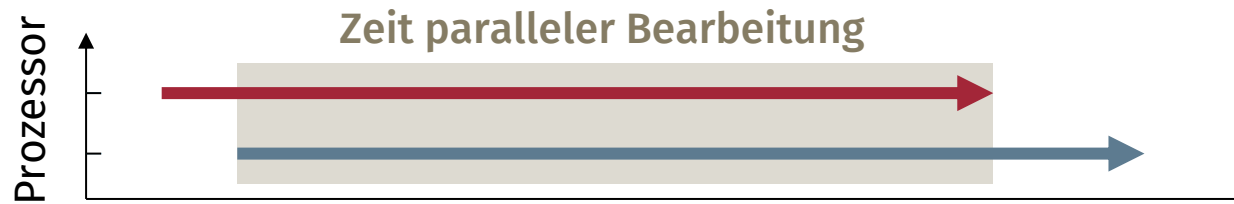
- nebenläufige Aktivitätsträger sind **unabhängig voneinander**
 - müssen **nicht** aufeinander **warten**
 - müssen sich **nicht synchronisieren**
- nebenläufige Aktivitätsträger können unabhängig voneinander ablaufen

Nebenläufigkeit (2)

Parallelität von Aktivitätsträgern (Prozessen oder Threads)

- ◆ nebenläufige Aktivitätsträger können **parallel** laufen
 - ◆ heißt gleichzeitige Ausführung der Anweisungen mehrerer Aktivitätsträger
 - nur auf Mehrprozessor- bzw. Mehrkernsystemen möglich

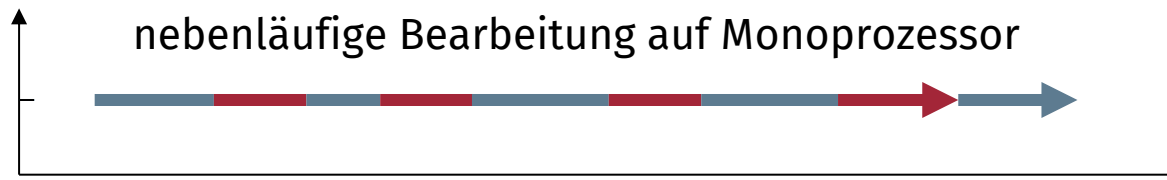
■ Beispiel:



Nebenläufigkeit (3)

Beispiel: Monoprozessorsysteme ohne Parallelität

- trotzdem so etwas wie „Scheinparallelität“
 - mehrere Aktivitätsträger, fair umgeschaltet durch Scheduler
 - beliebige unvorhersehbare zeitliche Durchmischung der Anweisungen möglich



- erscheint wie paralleler Ablauf, aber langsamer

Nebenläufigkeit (4)

Einfluss der Schedulingstrategie

- für Nebenläufigkeit muss Strategie **unabhängig** von auszuführenden Aktivitätsträger sein
 - sonst Ausführungen abhängig voneinander und damit nicht mehr nebenläufig
- **Gegenbeispiel:**
 - HPF mit statischen Prioritäten, wenige Aktivitätsträger
 - Ausführung evtl. **genau vorhersagbar**
 - Achtung: dazu muss Anzahl Prozessoren bekannt sein
 - Aktivitätsträger **nicht mehr unabhängig** voneinander
 - wird in Echtzeitsystemen stark ausgenutzt, um Nebenläufigkeit einzuschränken

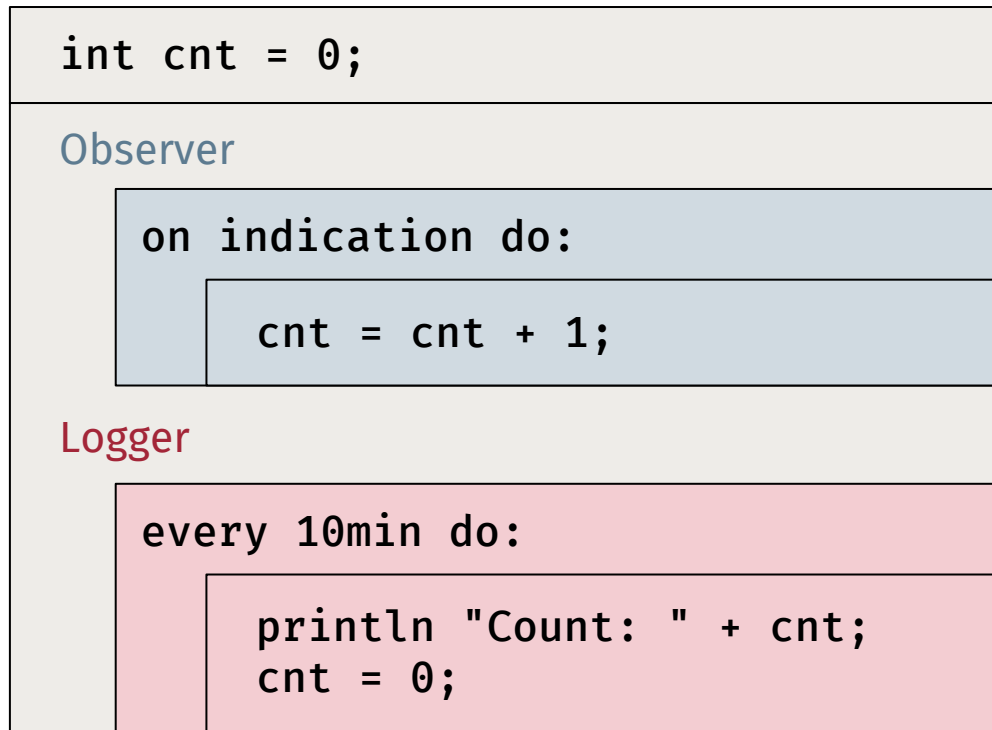
Probleme bei Nebenläufigkeit

Beispiel

- zwei Prozesse: **Beobachter** und **Protokollierer**
 - über eine Induktionsschleife sollen Autos gezählt werden
 - **Beobachter** erkennt Autos und zählt
 - **Protokollierer** soll alle 10min die in diesem Intervall aufgelaufenen Autos protokollieren

Probleme bei Nebenläufigkeit (2)

Mögliches Programm



Probleme bei Nebenläufigkeit (3)

Merkwürdige Effekte

- Fahrzeuge gehen „verloren“
- Fahrzeuge werden doppelt gezählt

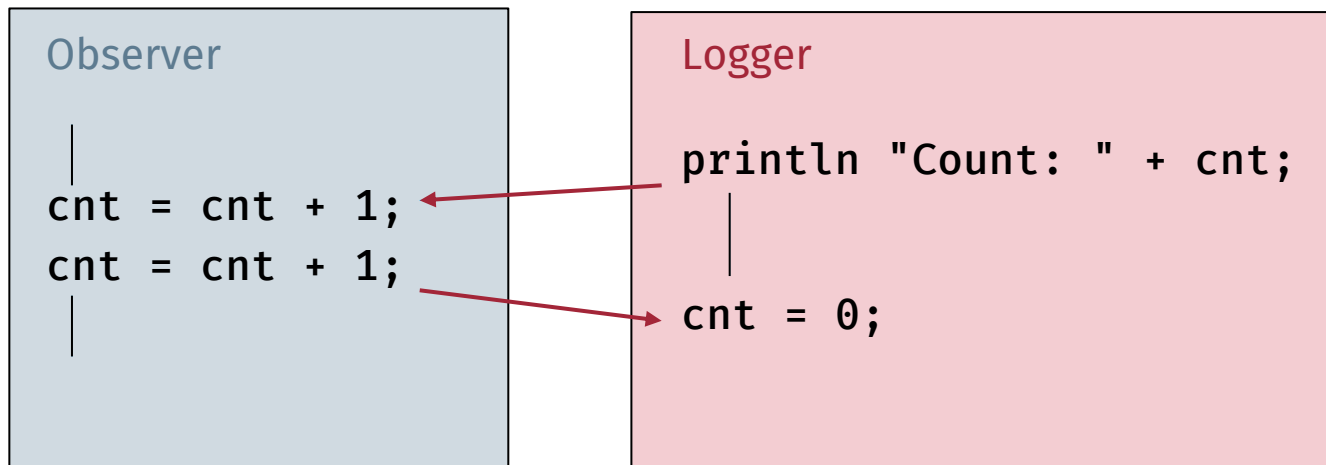
Ursachen

- Befehle einer Programmiersprache nicht unteilbar (atomar)
 - Abbildung auf mehrere Maschinenbefehle
- mehrere Anweisungen zusammen nie atomar
- ◆ (unerwartete) Prozesswechsel innerhalb einer Anweisung oder zwischen zwei zusammengehörigen Anweisungen können zu Inkonsistenzen führen

Probleme bei Nebenläufigkeit (4)

Fahrzeuge gehen „verloren“

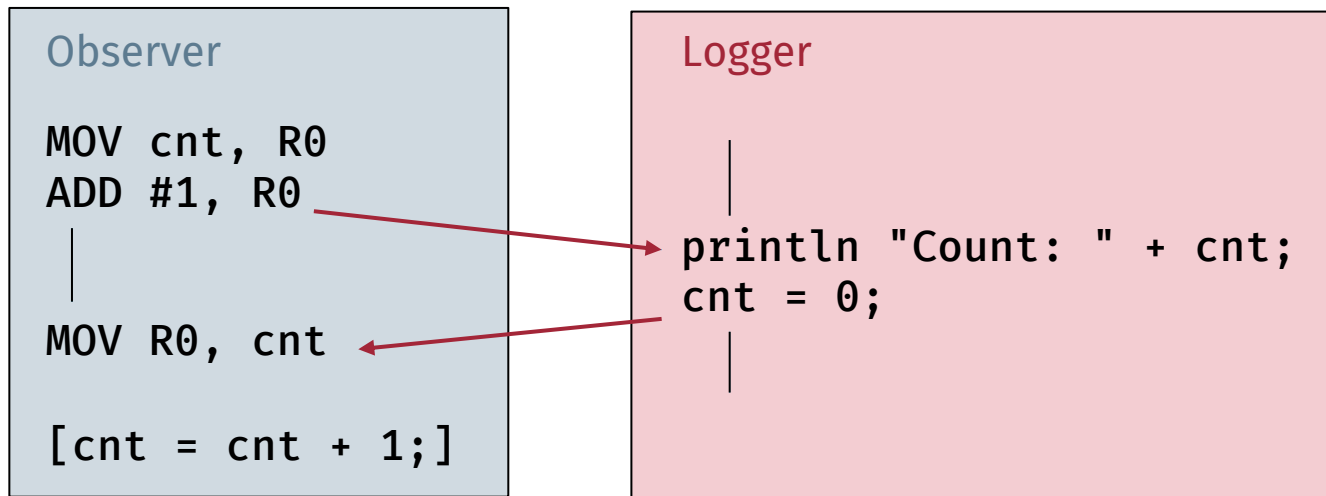
- Prozesswechsel nach dem Drucken im Protokollierer
 - Beobachter zählt weitere Fahrzeuge
 - nach Prozesswechsel löscht Protokollierer den Zähler



Probleme bei Nebenläufigkeit (5)

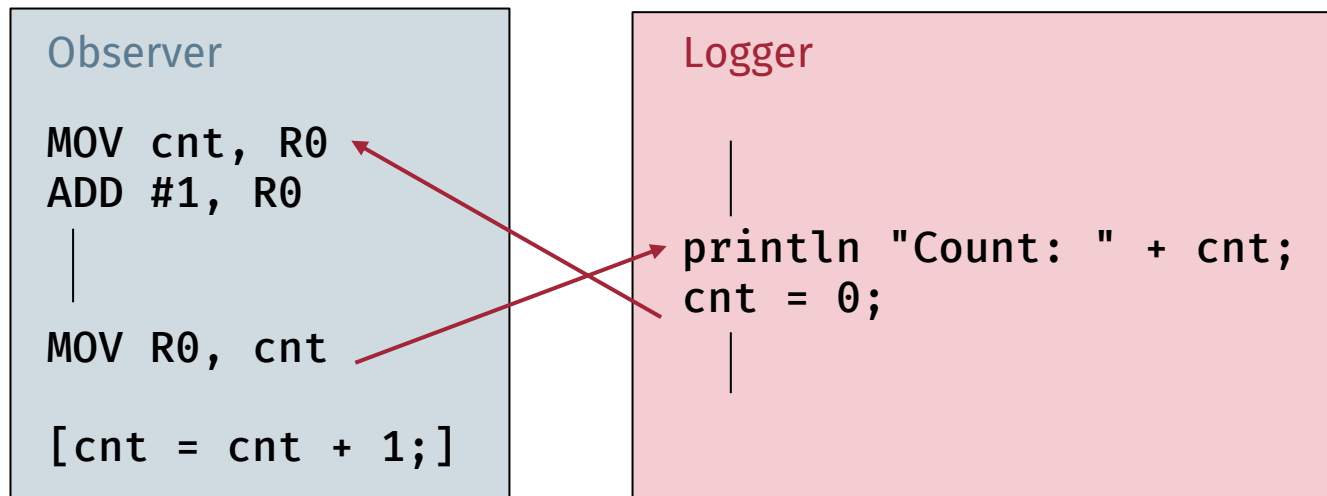
Fahrzeuge werden doppelt gezählt

- Beobachter will Zähler erhöhen
 - Zählerwert muss in Register geladen werden
 - nach Prozesswechsel löscht Protokollierer den Zähler
 - Beobachter speichert alten Zähler aus Register zurück



Probleme bei Nebenläufigkeit (6)

Prozesse sind nicht wirklich unabhängig



- Protokollierer darf nicht beliebig unterbrechen warten
 - entweder vorher laufen
 - oder warten bis Beobachter seine Inkrementierung beendet hat
- ◆ **Ursache:** gemeinsame Nutzung von Ressourcen (cnt-Variable)

Koordinierung

Koordinierung heißt Einschränkung der Nebenläufigkeit

- **Einschränkung** der zeitlichen Durchmischungen von nebenläufigen Threads/Prozessen
 - gezielte Aufgabe der Unabhängigkeit
- Einschränkungen sollten minimal sein
 - möglichst geringe Einschränkung über mögliche Parallelität

Ansatzpunkte

- **Umschaltungen** verhindern
 - Scheduler muss mit Koordinierung verknüpft werden
- Fortschritt verhindern durch **Anhalten** des Prozesses
 - funktioniert (weitgehend) unabhängig vom Scheduler

Koordinierung (2)

Maßnahmen zum geeigneten Anhalten

- gegenseitiger Ausschluss
- Sperren
- Semaphore
- Monitore
- ...



(Grundlagen der) Betriebssysteme | E.9



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

Gegenseitiger Ausschluss

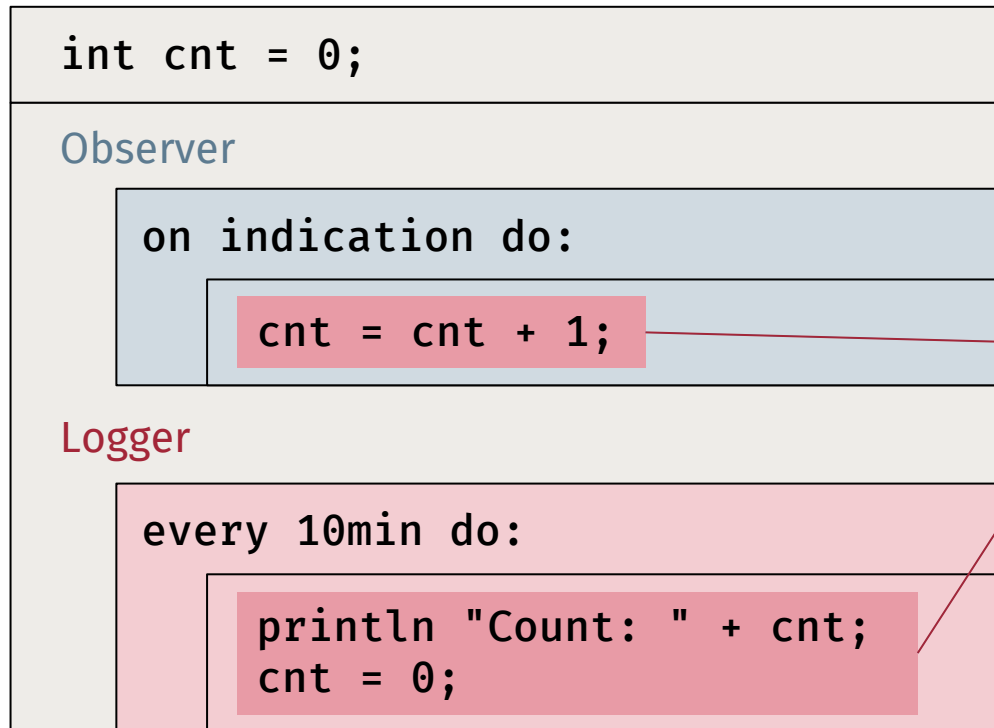
Nur ein Prozess/Thread in einem kritischen Abschnitt

- Befehlsabschnitt mit exklusivem Zugang zu bestimmten Daten oder Ressourcen
- alternative Bezeichnungen:
 - wechselseitiger Ausschluss, Mutual Exclusion, Mutex
- kritische Abschnitte erscheinen als zeitlich unteilbar

Gegenseitiger Ausschluss (2)

Lösung unseres Problems:

- zwei zusammengehörende kritische Abschnitte



nur genau ein Prozess
zu einem Zeitpunkt in
einem der Abschnitte

Gegenseitiger Ausschluss (3)

Wie kann der gegenseitige Ausschluss in kritischen Abschnitten erzielt werden?

- Vorkehrungen, dass nicht mehrere Prozesse gleichzeitig im kritischen Abschnitt sind
- Prozess kommt an einen kritischen Abschnitt
 - wartet bis alle anderen Prozesse den Abschnitt verlassen haben
- vor und nach kritischen Abschnitt muss es spezielle Anweisungen geben
 - meist benötigen diese Anweisungen gemeinsame Datenstrukturen

Algorithmus von Peterson

Zwei Prozesse (1981)

■ Voraussetzungen:

- faire Scheduling-Strategie
- gemeinsame Variablen für beide Prozesse
- Lesen und Schreiben von Variablen ist unteilbar (atomar)

■ Implementierung:

- ◆ kommt **ohne besondere Anweisungen** aus

■ Beispiel:

- zwei Prozesse P_0 und P_1 die regelmäßig jeweils kritische und unkritische Abschnitte durchlaufen

Algorithmus von Peterson (2)

Funktionsfähiges Beispiel

```
boolean ready0= false;  
boolean ready1= false;  
int turn= 0;
```

Warum funktioniert das?

```
while(true) {                                P0  
    ready0= true;  
    turn= 1;  
    while(ready1 && turn==1)  
        ;  
    ... // CRITICAL  
    ready0= false;  
    ... // uncritical  
}
```

```
while(true) {                                P1  
    ready1= true;  
    turn= 0;  
    while(ready0 && turn==0)  
        ;  
    ... // CRITICAL  
    ready1= false;  
    ... // uncritical  
}
```

Erster Ansatz

Leider nicht funktionsfähiges Beispiel

```
int turn= 0;
```

```
while(true) { P0  
    while(turn==1)  
        ;  
    ... // CRITICAL  
    turn= 1;  
    ... // uncritical  
}
```

```
while(true) { P1  
    while(turn==0)  
        ;  
    ... // CRITICAL  
    turn= 0;  
    ... // uncritical  
}
```

Erster Ansatz (2)

Lösung implementiert gegenseitigen Ausschluss

- immer nur ein Prozess im kritischen Abschnitt

Inhärentes Problem der Lösung

- nur alternierendes Betreten des kritischen Abschnitts durch P0 und P1 möglich
- Implementierung ist unvollständig

Zweiter Ansatz

Leider immer noch nicht funktionsfähiges Beispiel

```
boolean ready0= false;  
boolean ready1= false;
```

```
while(true) { P0  
    ready0= true;  
  
    while(ready1)  
        ;  
  
    ... // CRITICAL  
  
    ready0= false;  
  
    ... // uncritical  
}
```

```
while(true) { P1  
    ready1= true;  
  
    while(ready0)  
        ;  
  
    ... // CRITICAL  
  
    ready1= false;  
  
    ... // uncritical  
}
```

Zweiter Ansatz (2)

Lösung implementiert gegenseitigen Ausschluss

- immer nur ein Prozess im kritischen Abschnitt


Inhärentes Problem der Lösung

- **Verklemmung** möglich
 - richtige Bezeichnung in diesem Fall: **Live-Lock**
 - Prozesse arbeiten weiter, machen aber keinen Fortschritt


Zweiter Ansatz (3)

Beispielablauf für Verklemmung

Prozess P_0

```
ready0= true;  
|  
while ready1  
|  
;  
|  

```

Prozess P_1

```
|  
ready1= true;  
|  
while ready0  
|  
;  

```


Algorithmus von Peterson (3)

Funktionsfähiges Beispiel (Kombination der Ansätze)

```
boolean ready0= false;  
boolean ready1= false;  
int turn= 0;
```

```
while(true) {                                P0  
    ready0= true;  
    turn= 1;  
    while(ready1 && turn==1)  
        ;  
    ... // CRITICAL  
    ready0= false;  
    ... // uncritical  
}
```

```
while(true) {                                P1  
    ready1= true;  
    turn= 0;  
    while(ready0 && turn==0)  
        ;  
    ... // CRITICAL  
    ready1= false;  
    ... // uncritical  
}
```

Algorithmus von Peterson (4)

Beispielablauf zu möglicher Verklemmung

Prozess P₀

ready0= true;

turn= 1;

while(ready1 && turn==1)
;

... // CRITICAL

Prozess P₁

ready1= true;

turn= 0;

while(ready0 && turn==0)
;

Algorithmus von Peterson (5)

Algorithmus implementiert wechselseitigen Ausschluss

- lebendige (*live*) und sichere (*safe*) Implementierung
- `turn` entscheidet für den kritischen Fall des zweiten Ansatzes, welcher Prozess den kritischen Abschnitt betreten darf
 - in allen anderen Fällen ist `turn` unbedeutend

Probleme der Lösung

- aktives Warten
- nur für zwei Prozesse
 - erweiterbar für mehrere Prozesse: kompliziertere Bedingungen
- unterschiedliche Anweisungen pro Prozess vor einem kritischen Abschnitt



Grundlagen der Betriebssysteme | E.10



Franz J. Hauck | Institut für Verteilte Systeme, Univ. Ulm

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore

Spezielle Maschinenbefehle

Unterstützung für gegenseitigen Ausschluss

- Test-and-Set-Instruktion
- Swap-Instruktion

Test-and-Set-Instruktion

- Effekt
 - Setzen eines Bits im Speicher auf 1
 - Rückgabe des vorherigen Werts des Bits
 - Lese- und Schreib-Operation unteilbar (atomar)

Spezielle Maschinenbefehle (2)

Test-and-Set in der Praxis

- arbeitet auf einem Speicherwort
 - wird auf Wert 1 gesetzt
- Rückgabe durch CCR
 - bildet einen Vergleich des bisherigen Werts mit 0 nach
- Beispiel:

TAS <someVariable>

JEQ <wasZero>

JNE <wasNotZero>

Spezielle Maschinenbefehle (3)

Gegenseitiger Ausschluss mit Test-and-Set

```
word lock= 0;
```

```
while(true) { P0  
    L0: TAS lock  
        JNE L0  
  
    ... // CRITICAL  
  
    lock= 0;  
  
    ... // uncritical  
}
```

```
while(true) { P1  
    L1: TAS lock  
        JNE L1  
  
    ... // CRITICAL  
  
    lock= 0;  
  
    ... // uncritical  
}
```


Spezielle Maschinenbefehle (4)

Vorteil von Test-and-Set

- gleicher Code für jeden Prozess verwendbar


Nachteil

- aktives Warten

Hinweis zum Einsatz in der Praxis

- Vermeidung von überflüssigen Schreibzugriffen zur Entlastung des Speicherbuses

```
L0:  MOV lock, R0
      SUB #0, R0
      JNE L0
      TAS lock
      JNE L0
```



Spezielle Maschinenbefehle (5)

Swap-Instruktion

■ Effekt

- atomarer Tausch der Inhalte zweier Speicherzellen

■ Instruktion

SWP <someVariable>, <someOther>

Spezielle Maschinenbefehle (6)

Gegenseitiger Ausschluss mit Swap

```
word lock= 0;
```

```
while(true) {                                P0  
    word key= 1;  
    while( key==1 ) {  
        SWP key, lock  
    }  
    ... // CRITICAL  
    lock= 0;  
    ... // uncritical  
}
```

```
while(true) {                                P1  
    word key= 1;  
    while( key== 1 ) {  
        SWP key, lock  
    }  
    ... // CRITICAL  
    lock= 0;  
    ... // uncritical  
}
```

Kritik an bisherigen Verfahren

Spinlock

- bisherige Verfahren werden auch Spinlocks genannt
- aktives Warten auf Zugang zum kritischen Abschnitt

Nachteile

- Verbrauch von Rechenzeit ohne Nutzen
- Behinderung „nützlicherer“ Prozesse
- Abhängigkeit von der Scheduling-Strategie
 - z.B. schlechte Effizienz bei langen Zeitscheiben

Kritik an bisherigen Verfahren (2)

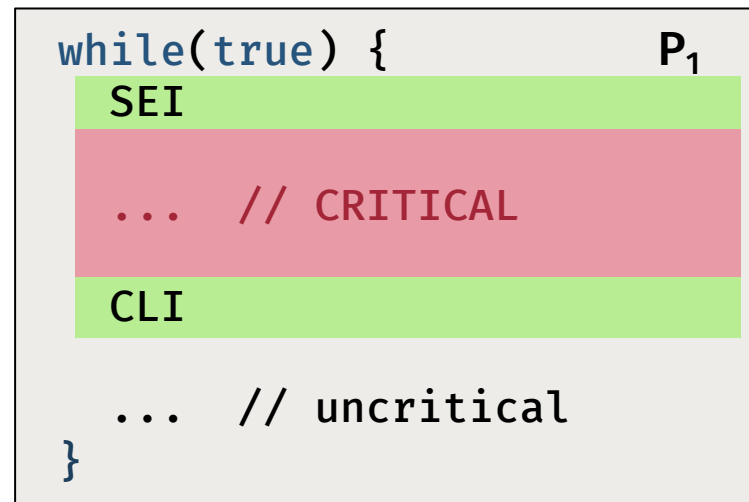
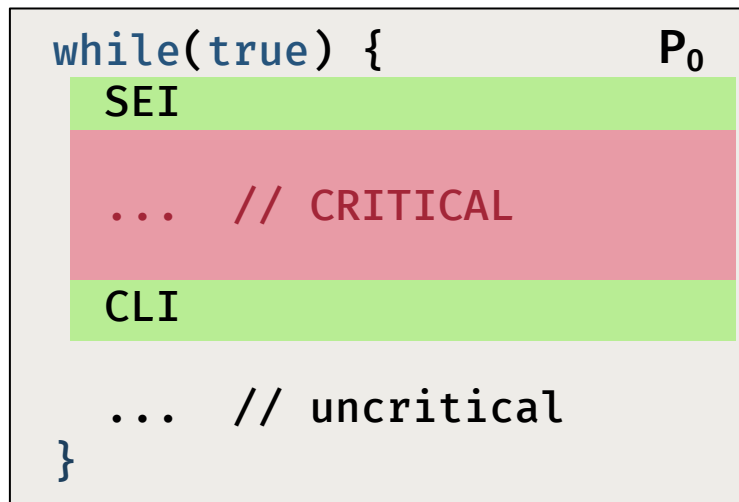
Einsatz von Spinlocks in der Praxis

- fast ausschließlich in Multiprozessorsystemen
 - Koordinierung über mehrere Prozessoren hinweg
- nur für kurze kritische Abschnitte effizient
 - z.B. Zugang zur Bereit-Warteschlange beim Scheduling

Sperrung von Unterbrechungen

Elegante Methode in Monoprozessorsystemen

- spezielle Instruktionen
 - SEI – sperrt Unterbrechungen
 - CLI – gibt Unterbrechungen wieder frei



- nur für sehr kurze kritische Abschnitte geeignet

Semaphore

Semaphore (griech. Zeichenträger)

- Systemdatenstruktur mit zwei Operationen (*Edsger W. Dijkstra*)
 - **P-Operation** (*proberen; passeren; wait; down*)
 - wartet bis Zugang frei
 - **V-Operation** (*verhogen; vrijgeven; signal; up*)
 - macht Zugang für anderen Prozess frei
 - eine private Integer-Variable zur internen Steuerung

Semaphore (2)

Implementierung als „Java-Klasse“

```
class Semaphore {  
    private int s;  
    public Semaphore( int init ) { this.s= init; }  
    public p() {  
        while( s<= 0 )  
            yield();  
        s--;  
    }  
    public v() {  
        s++;  
    }  
}
```

hier kann Abschnitt unterbrochen werden

kritische Abschnitte

Semaphore (3)

Gegenseitiger Ausschluss mit einem Semaphore

```
Semaphore sem= new Semaphore(1);
```

```
while(true) {                                P0  
    sem.p();  
    ... // CRITICAL  
    sem.v();  
    ... // uncritical  
}
```

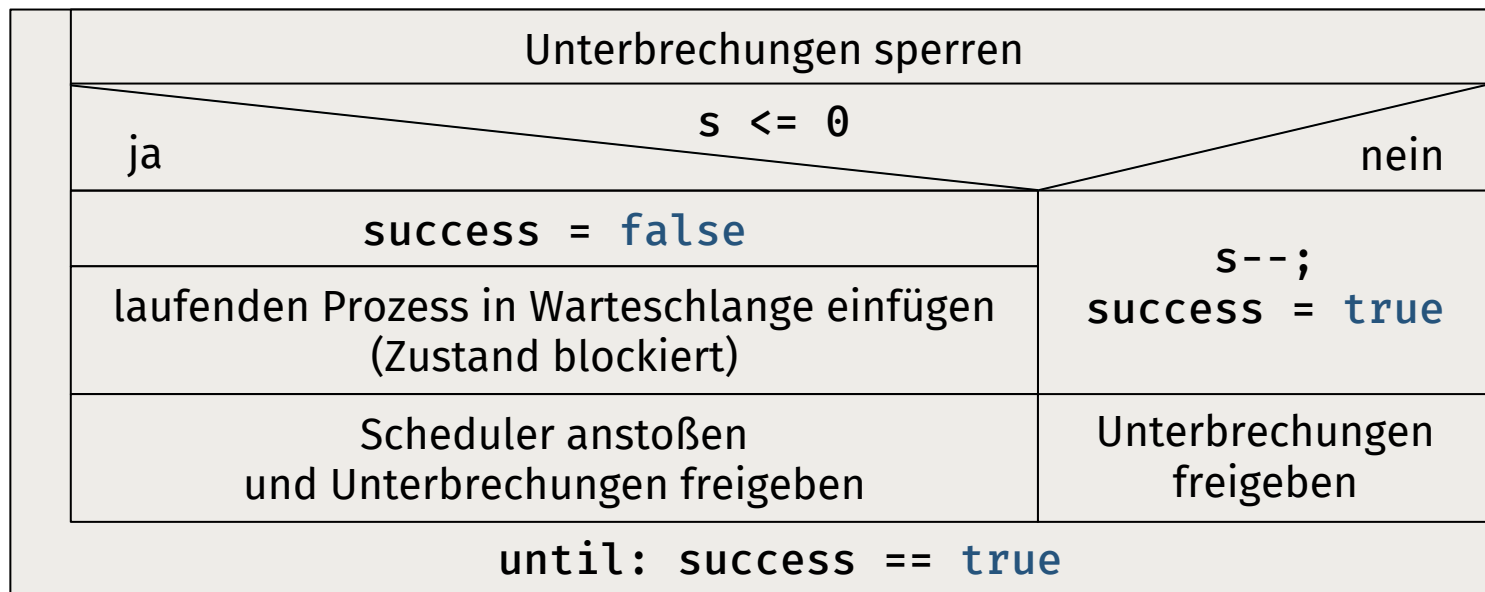
```
while(true) {                                P1  
    sem.p();  
    ... // CRITICAL  
    sem.v();  
    ... // uncritical  
}
```

- ◆ Wie lassen sich P- und V-Operationen **ohne aktives Warten** implementieren?

Semaphore (4)

Skizze einer Implementierung auf Monoprozessorsystem

■ Systemaufruf P-Operation



- `success` ist prozesslokale Variable
- jeder Semaphore besitzt eigene Warteschlange für blockierte Prozesse

Semaphore (5)

Skizze einer Implementierung auf Monoprozessorsystem

■ Systemaufruf V-Operation

Unterbrechungen sperren
$S++$
alle Prozesse aus Warteschlange in bereit-Zustand versetzen
Scheduler anstoßen und Unterbrechungen freigeben

- Prozesse probieren immer wieder, die P-Operation erfolgreich abzuschließen
- Scheduling-Strategie entscheidet über Reihenfolge und Fairness
- leichte Ineffizienz durch Aufwecken aller Prozesse
- mit Einbezug der Scheduling-Strategie effizientere Implementierungen möglich

Semaphore (6)

Vorteile einer Semaphor-Implementierung im Betriebssystem

- kein aktives Warten mehr
 - Ausnutzen der Blockierzeiten durch andere Prozesse
- Einbeziehen des Schedulers in die Semaphor-Operationen

Implementierung einer Synchronisierung

- S1 in P1 soll immer vor S2 in P2 stattfinden

```
Semaphore sem= new Semaphore(0);
```

```
...  
S1;  
sem.v();  
...
```

P₀

```
...  
sem.p();  
S2;  
...
```

P₁

Inhaltsüberblick

Prozessverwaltung und Nebenläufigkeit

- Einordnung und Motivation
- Prozesse
 - Repräsentation, Erzeugung, Prozesswechsel, Zustände
- Auswahlstrategien
 - FCFS, SJF, HPF, RR, MLQ, MLFQ, VRR, CFS und MuQSS
- Aktivitätsträger (Threads)
 - User-level und Kernel-level Threads
- Parallelität und Nebenläufigkeit
 - Koordinierung
 - gegenseitiger Ausschluss, Semaphore