



Technische
Universität
Braunschweig



Institute of Operating Systems
and Computer Networks
Reliable System Software



Betriebssysteme (BS)

Christian Dietrich

Wintersemester 2024



Teil A: Einführung

1 Einführung und Organisation

2 Exkurs: Systemnahe C-Programmierung

3 Grundlegende Konzepte

Teil B: Prozesse und Dateien

4 Dateien und Dateisysteme

5 Prozesse und Fäden

6 Unterbrechungen, Ausnahmen, Signale

7 Prozessverwaltung

Teil C: Interaktion und Kommunikation

8 Speicherbasierte Interaktionen

9 Betriebsmittelverwaltung, Synchronisation und Verklemmung

10 Interprozesskommunikation (IPC)

Teil D: Speicher und Zugriffsschutz

11 Speicherorganisation

12 Speichervirtualisierung



Technische
Universität
Braunschweig



Institute of Operating Systems
and Computer Networks
Reliable System Software



Betriebssysteme (BS)

Teil A Einführung

Christian Dietrich

Wintersemester 2024



1 Einführung und Organisation

- 1.1 Systemnahe Informatik → Betriebssysteme
- 1.2 Was ist (tut) ein Betriebssystem?
- 1.3 Ziele der Veranstaltung
- 1.4 Literatur
- 1.5 Aufbau und Organisation der Veranstaltung
- 1.6 Semesterüberblick
- 1.7 Ablauf und Inhalt der Übungen
- 1.8 Zusammenfassung

2 Exkurs: Systemnahe C-Programmierung

3 Grundlegende Konzepte

1.1 Systemnahe Informatik → Betriebssysteme



Prof. Dr.-Ing Christian Dietrich
dietrich@ibr.cs.tu-bs.de

Verlässliche Systemsoftware



Prof. Dr.-Ing Lars Wolf
wolf@ibr.cs.tu-bs.de

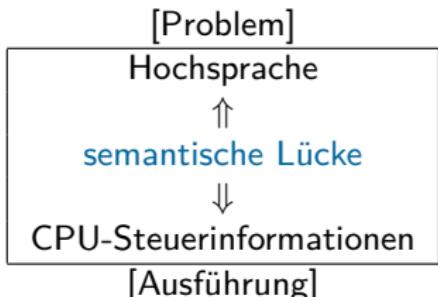
Connected and Mobile Systems



Prof. Dr.-Ing Sándor Fekete
fekete@ibr.cs.tu-bs.de

Algorithmik

Forschung und
Lehre in der
systemnahen
Informatik



- Informatisches Urproblem:
Schließen der Semantischen Lücke

 - Komplexitätsreduktion durch Hierarchie
 - *Divide et impera* – Teile und herrsche
 - Hierarchisch angeordnete **virtuelle Maschinen**
 - Problem schrittweise in einfachere Darstellungen bis auf die reale Maschine herunterbrechen.

 - Top-Down vs. Bottom-Up
 - Top-Down: Vom Problem zur Lösung
 - Bottom-Up: Erschaffen von Lösungsräumen
- Systeminfrastruktur



- Schrittweises Schließen der semantischen Lücke durch
 - Interpretation auf einer einfacheren (virtuellen) Maschine
 - Übersetzung für eine einfachere (virtuelle) Maschine

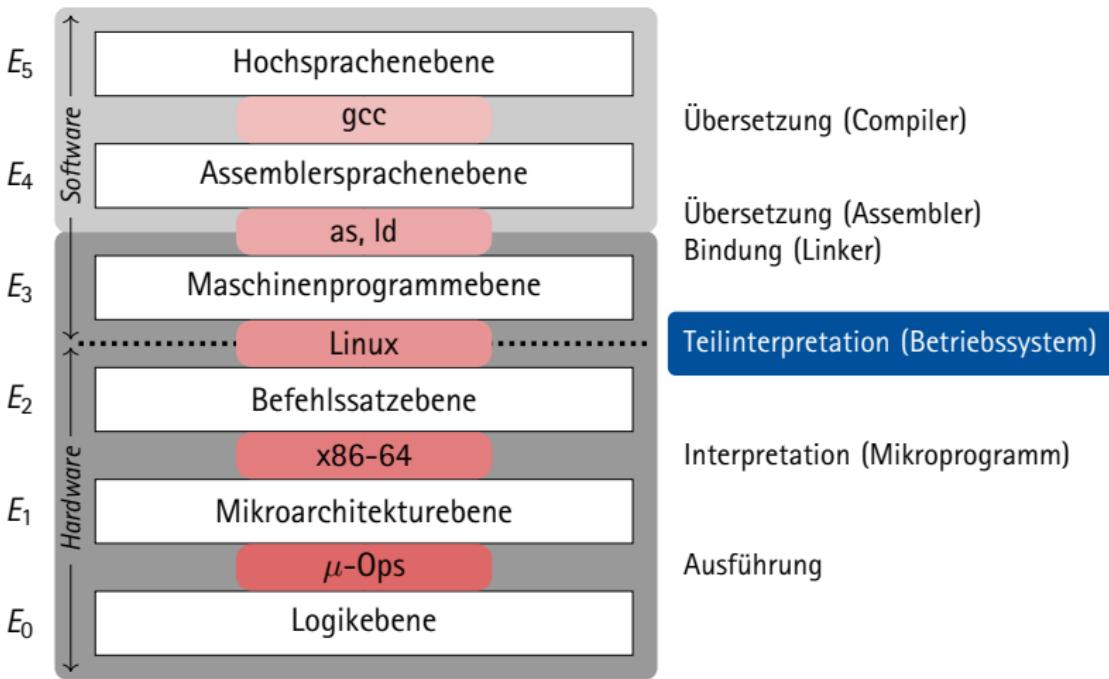


Ebene		
n	virtuelle Maschine M_n mit Maschinensprache S_n	Programme in S_n werden von einem auf einer tieferen Maschine laufenden Interpreter gedeutet oder in Pro- gramme tieferer Maschinen übersetzt
:	:	:
2	virtuelle Maschine M_2 mit Maschinensprache S_2	Programme in S_2 werden von einem auf M_1 bzw. M_0 lau- fenden Interpreter gedeutet oder nach S_1 bzw. S_0 über- setzt
1	virtuelle Maschine M_1 mit Maschinensprache S_1	Programme in S_1 werden von einem auf M_0 laufenden Interpreter gedeutet oder nach S_0 übersetzt
0	reale Maschine M_0 mit Maschinensprache S_0	Programme in S_0 werden direkt von der Hardware aus- geführt

- Übersetzung und Interpretation als Techniken, um (auch in Kombination) Programme auf HW zur Ausführung zu bringen.



Beispiel: Arbeitsplatzrechner



„Muss ich *das* wirklich im Detail wissen?“



Welche Lösung ist besser?

Gibt es überhaupt einen Unterschied?

Mathematik

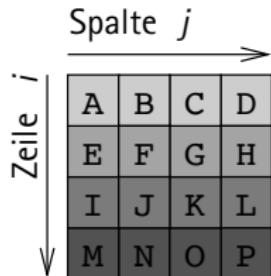
$$\mathbf{A} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \quad \text{sum}(\mathbf{A}) = \sum_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}} a_{ij}$$

Informatik (Hochsprache)

```
int sum = 0;
for (int j = 0; j < N; ++j) {           // Erst Spalten, dann Zeilen
    for (int i = 0; i < M; ++i) {
        sum += matrix[i][j];
    }
}
```

Informatik (Hochsprache, alternative Implementierung)

```
int sum = 0;
for (int i = 0; i < M; ++i) {           // Erst Zeilen, dann Spalten
    for (int j = 0; j < N; ++j) {
        sum += matrix[i][j];
    }
}
```



zeilenweise Abspeicherung/Aufzählung

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

spaltenweise Abspeicherung/Aufzählung

A	E	I	M	B	F	J	N	C	G	K	O	D	H	L	P
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Im Abstrakten (Programmsemantik) ist eines so gut wie das andere.
- Im Konkreten (Programmausführung) jedoch nicht!
 - Ein C-Compiler speichert Felder **zeilenweise** ab
 - Lösung 1 zählt das Feld jedoch **spaltenweise** auf

Faktor 10–20!

Folge der spaltenweisen Aufzählung

Zugegriffene Speicheradressen „springen“ um $N \times \text{sizeof char}$.

- Jeder Zugriff erfordert (bei nichtkleinem N) das Laden einer Cache-Line durch die HW.
- Ggf. zusätzlich (bei großem N) das Einlagern einer Speicherseite durch das BS.
- **Verletzung des Lokalitätsprinzips**, auf das Speicherstrategien optimiert sind.

- **Funktional** sind beide Lösungen identisch
 - Wegen der Kommutativität der Addition sind beide Verfahren korrekt
- **Nichtfunktional** zeigen sich erhebliche Unterschiede
 - Spaltenweise Bearbeitung um **Faktor 10-20** langsamer
- Die Ursache findet sich in den tieferen Schichten!
 - Wie organisiert der **Compiler** das Feld im Speicher? Wie den Zugriff?
 - Wie stellt das **Betriebssystem** Speicher bereit?
 - Wie läuft der Speicherzugriff auf der **Hardware** ab?

„De markationslinie“ zwischen technischer und praktischer Informatik

Die **Funktionsweise** eines Betriebssystems zu verstehen hilft,
Phänomene eines Rechensystems zu begreifen und besser einzuschätzen.

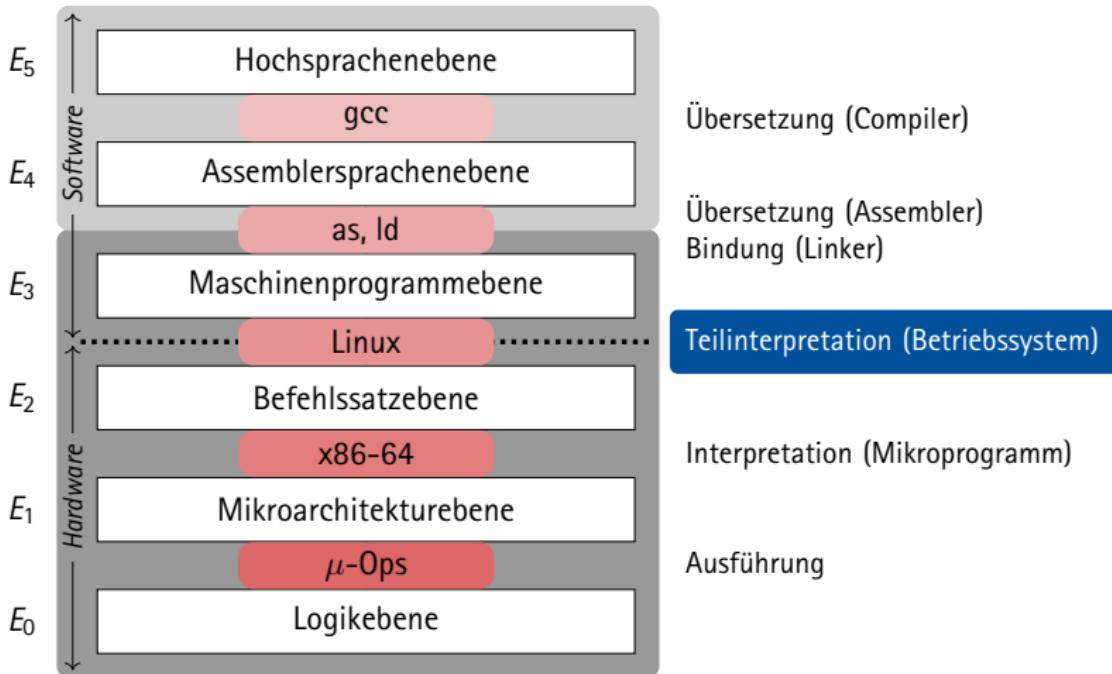
↪ **Wissen was dahinter steckt!**



Einordnung ins Informatikstudium

Mehrebenenmaschine

„Demarkationslinie“ zwischen technischer und praktischer Informatik

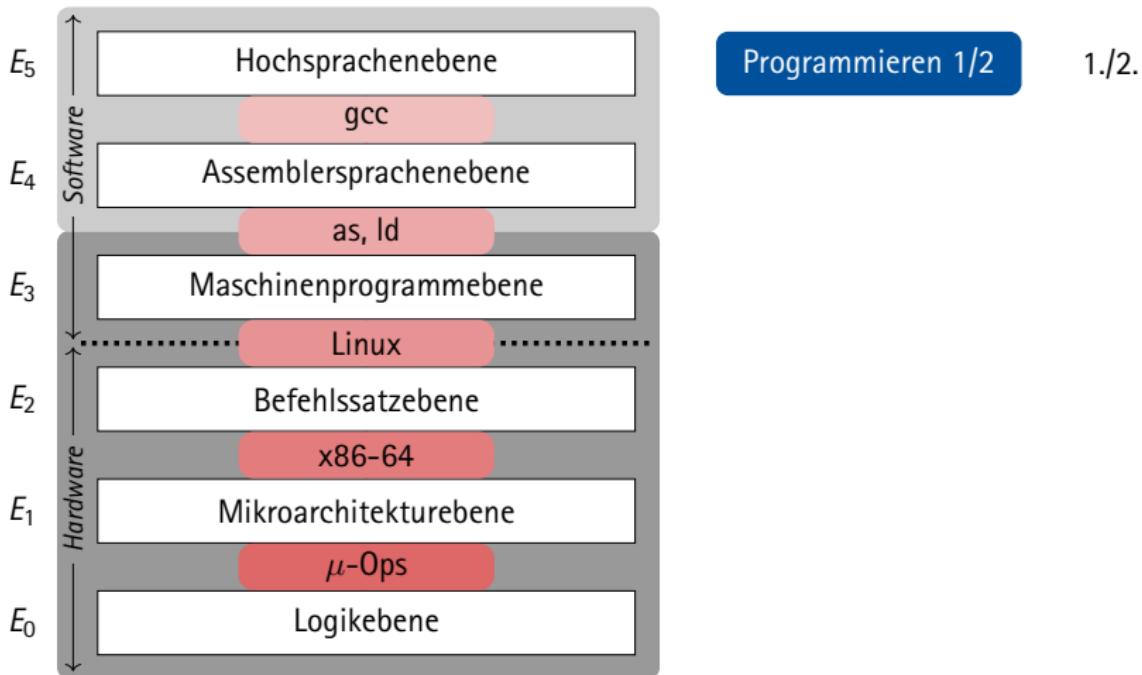




Einordnung ins Informatikstudium

Mehrebenenmaschine

„Markationslinie“ zwischen technischer und praktischer Informatik

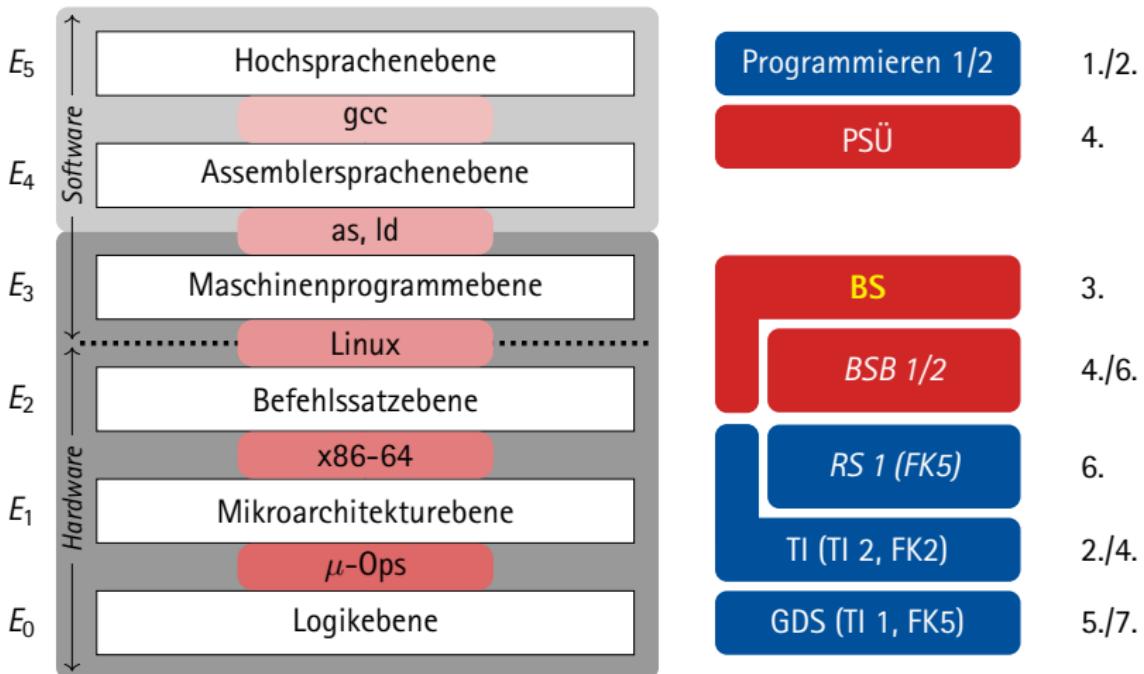




Einordnung ins Informatikstudium

Mehrebenenmaschine

„Markationslinie“ zwischen technischer und praktischer Informatik

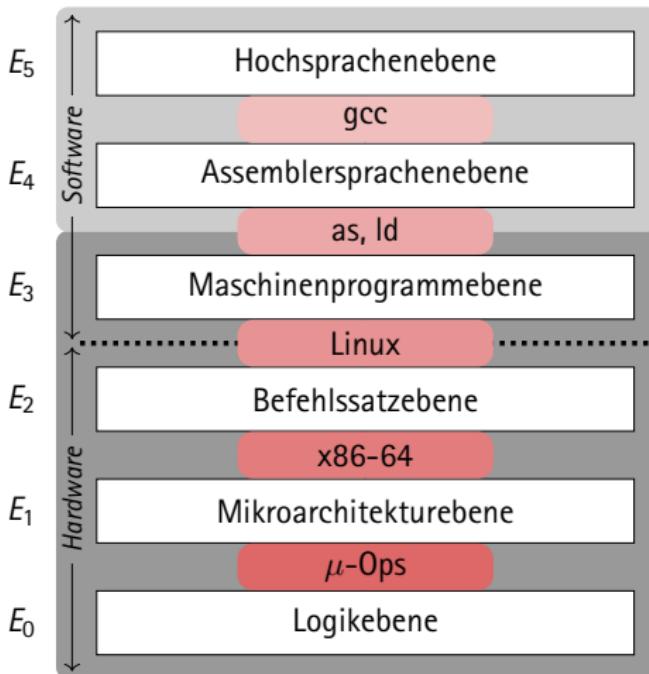




Einordnung ins Informatikstudium

Mehrebenenmaschine

„Demarkationslinie“ zwischen technischer und praktischer Informatik



Moderne
Rechnerarchitektur
verstehen und erklären



RS 1 (FK5)

6.

TI (TI 2, FK2)

2./4.



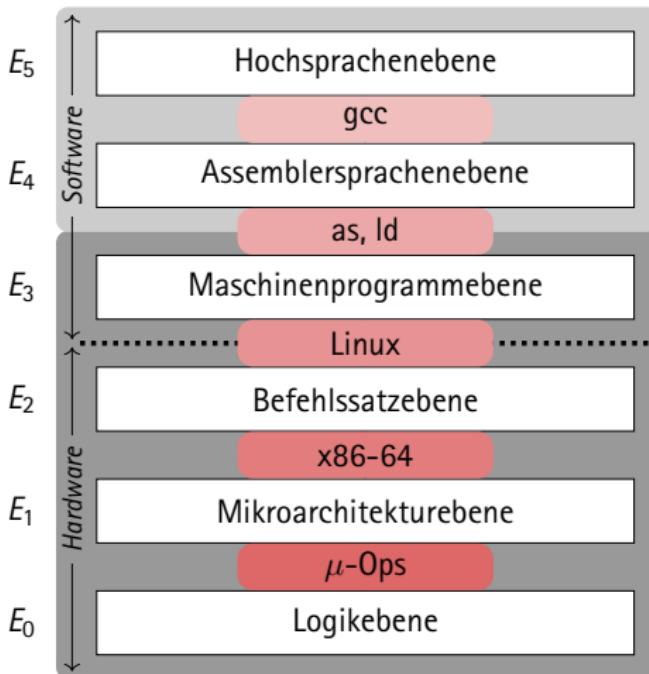
Von der Logikschaltung
zum Prozessor



Einordnung ins Informatikstudium

Mehrebenenmaschine

„Demarkationslinie“ zwischen technischer und praktischer Informatik



BS-Konzepte verstehen und verwenden



BS

3.

BSB 1/2

4./6.

BS-Konzepte vertiefen + entwerfen und implementieren



1.2 Was ist (tut) ein Betriebssystem?

Systemnahe Informatik ↽ Betriebssysteme

- Infrastruktursoftware für Rechensysteme
 - was Betriebssysteme sind, hat schon „Glaubenskriege“ hervorgerufen
 - das Spektrum reicht von „Winzlingen“ bis hin zu „Riesen“
 - simple Prozeduren \Leftrightarrow komplexe Programmsysteme
 - entscheidend ist, dass **Betriebssysteme nie dem Selbstzweck dienen**
- jedes Rechensystem wird durch ein Betriebssystem betrieben
 - Ausnahmen bestätigen die Regel...



- Betriebssysteme sind unerlässliches **Handwerkszeug** der Informatik
 - mit dem umzugehen ist zur Benutzung eines Rechensystems
 - das gelegentlich zu beherrschen, anzupassen und auch anzufertigen ist



- IBM: z/VM (vormals VM/CMS), z/OS



- DEC: VAX/VMS



- DOS (16-/32-Bit)-, NT- und CE-Linie



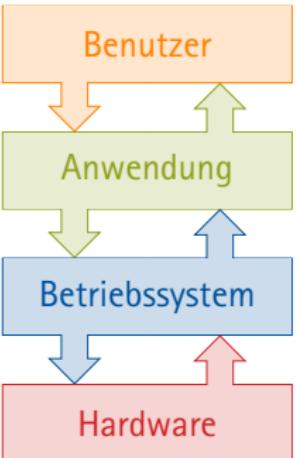
-     



Die **Funktionsweise** eines Betriebssystems zu verstehen hilft, **Phänomene** eines Rechensystems zu begreifen und besser einzuschätzen.

Phänomene hin oder her: Was ist (tut) denn nun ein Betriebssystem?

- Wir können drei Sichtweisen unterscheiden
 - Benutzersicht
 - Umgebung zum Verwenden von Anwendungen
 - Anwendungssicht
 - Funktionsbibliothek mit Abstraktionen zur Vereinfachung der Softwareentwicklung
 - Systemsicht
 - Softwareschicht zum Verwalten und Multiplexen der Hardwareressourcen
- Je nach Sicht unterscheidet sich das Verständnis

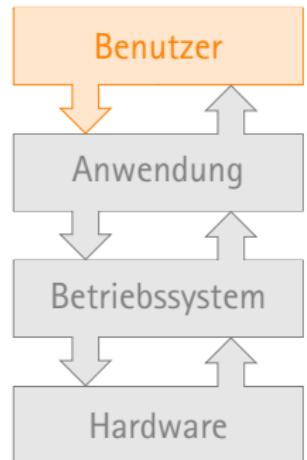


“ Ein Programm, das als Vermittler zwischen Rechnernutzer und Rechnerhardware fungiert. Der Sinn des Betriebssystems ist, eine Umgebung bereitzustellen, in der Benutzer bequem und effizient Programme ausführen können. ”

Silberschatz, Gagne und Galvin 2005: *Operating System Concepts* [30]

“ Be’triebs·sys·tem <n.; -s, -e; EDV> Programmzbündel, das die Bedienung eines Computers ermöglicht. ”

Wahrig-Burfeind 2002: *Universalwörterbuch Rechtschreibung* [37]



Was ist ein Betriebssystem?

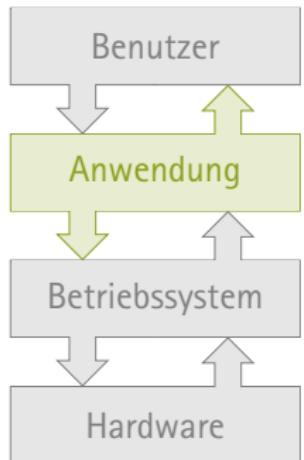
Literatur: Anwendungssicht

“ Eine **Softwareschicht**, die alle Teile des Systems verwaltet und dem Benutzer eine **Schnittstelle** oder eine virtuelle Maschine anbietet, die einfacher zu verstehen und **zu programmieren** ist [als die nackte Hardware]. ”

Tanenbaum 1997: *Operating Systems: Design and Implementation* [35]

“ Es ist das Betriebssystem, das die Kontrolle über das Plastik und Metall (**Hardware**) übernimmt und anderen **Softwareprogrammen** (Excel, Word, ...) eine **standardisierte Arbeitsplattform** (Windows, Unix, OS/2) schafft. ”

Ewert, Christoffer, Christoffer u. a. 2001: *FreeHand 10* [12]



Was ist ein Betriebssystem?

Literatur: Systemsicht

“ Der Zweck eines Betriebssystems [besteht] in der Verteilung von Betriebsmitteln auf sich bewerbende Benutzer. ”

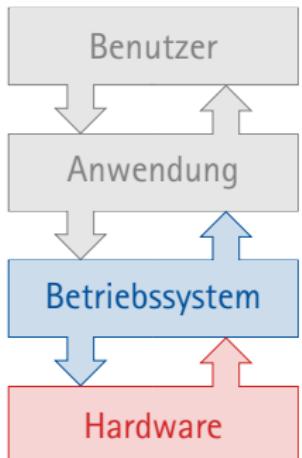
Hansen 1977: *Betriebssysteme* [14]

“ Eine Menge von Programmen, die die Ausführung von Benutzerprogrammen und die Benutzung von Betriebsmitteln steuern. ”

Habermann 1976: *Introduction to Operating System Design* [13]

“ Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen. ”

Deutsches Institut für Normung 1985: *Informationsverarbeitung – Begriffe* [9]

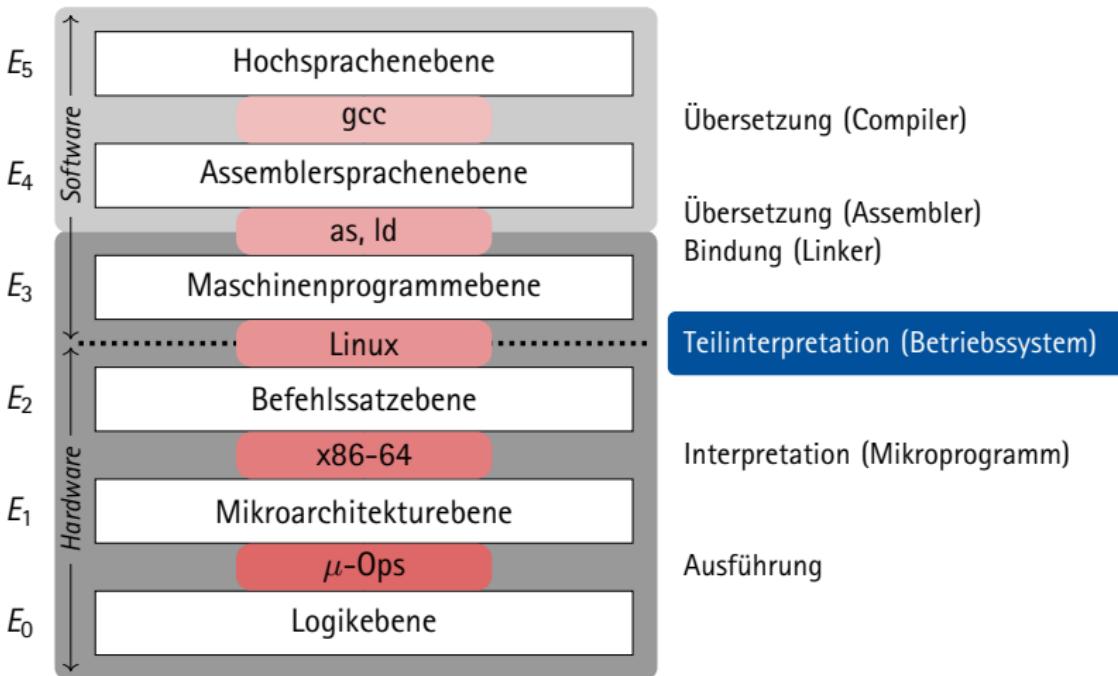




Was ist ein Betriebssystem?

Unsere Sicht in BS

Betriebssystem \mapsto Teilinterpretierende Virtuelle Maschine



- Das BS bietet dem Anwender (Programm) eine erweiterte Maschinenschnittstelle (**teilinterpretierende virtuelle Maschine, E_3**).
- Der Befehlssatz der HW (E_2) wird (weitestgehend) „durchgereicht“ und um zusätzliche **Systemaufrufe** (system calls) erweitert.
- Die Ressourcen der HW (Prozessor, Speicher, IO-Geräte, ...) werden durch **Multiplexing virtualisiert** (Mehrprogrammbetrieb).
- Virtuelle **Hardwareressourcen** werden durch **Schutzmechanismen** (räumlich und zeitlich) voneinander **isoliert** (Mehrbenutzerbetrieb).
- Virtuelle Hardwareressourcen werden repräsentiert durch die grundlegenden Konzepte (**Abstraktionen**) eines Betriebssystems.



1.3 Ziele der Veranstaltung

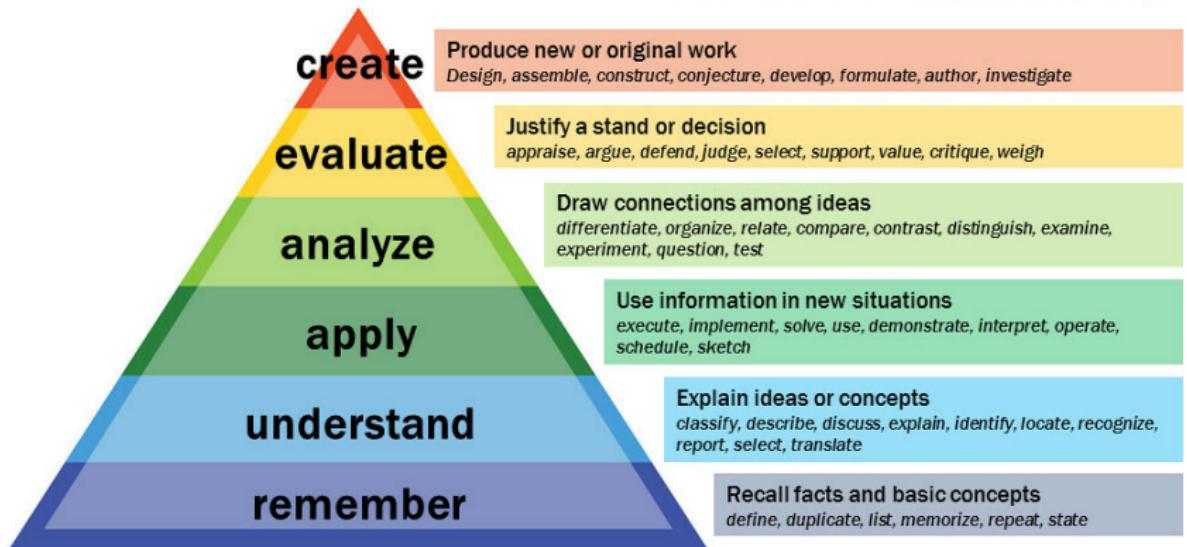


Lernziele: Grundlagen der Betriebssysteme

- **Kennen** der Rolle des BS, seiner grundlegenden Konzepte und Abstraktionen und deren Einordnung im Rechensystem
 - Ausgang: Programmieren I/II, (Technische Informatik)
 - Schwerpunkt: Vielzweckbetriebssysteme am Beispiel von UNIX
- **Verstehen** der technologischen Sprach- und Hardwaregrundlagen bei der Entwicklung systemnaher Software
 - Sprache – Compiler – **Betriebssystem** – Hardware
 - Das Prinzip der Mehrebenenmaschine verstehen und einschätzen können
- **Verwenden** der BS-Schnittstelle für systemnahe Software in C
 - Programmierübungen in **C** gegen eine **POSIX**-Schnittstelle (Linux)
 - **Praktische Erfahrungen** in systemnaher Softwareentwicklung machen



Bloom's Taxonomy



Source: Andrew W(T), CC BY-SA 2.0, via Wikimedia Commons.



Stufe	Verb	Beispiel im Kontext
1. Erinnern	nennen, aufzählen, wiedergeben	Kennt die grundlegenden Aufgaben eines Betriebssystems.
2. Verstehen	erklären, beschreiben, zuordnen	Kann erklären, wie ein Systemaufruf funktioniert und warum er nötig ist.
3. Verwenden	anwenden, benutzen, ausführen	Kann ein einfaches C-Programm gegen die UNIX API schreiben.
4. Analysieren	untersuchen, vergleichen, strukturieren	Kann den Unterschied zwischen Thread-Modellen analysieren.
5. Bewerten	bewerten, entscheiden, begründen	Kann neue Scheduling-Strategien hinsichtlich Fairness und Durchsatz bewerten.
6. Erzeugen	entwerfen, entwickeln, kombinieren	Entwirft ein eigenes Scheduling-Verfahren.

- Betriebssystemlehre erfolgt überwiegend am Beispiel von **UNIX**
 - **Warum UNIX?** (und nicht Windows/MacOS/iOS/<SuperOS>)
 - UNIX (von: UNICS für *Uniplexed Information and Computing Service*) ist der Ursprung vieler Betriebssystementwicklungen
 - insbesondere auch von Linux und Darwin (→ MacOS X)
 - Windows (NT) ist die bekannteste Ausnahme (aber ebenfalls stark beeinflusst)
 - Entwickelt ab 1969, nach Erfahrungen mit MULTICS [23] ↪ Vereinfachung
 - zunächst in Assembler, ab 1972 dann in **C** ↪ Portabilität
 - hohe Verbreitung durch frühe OpenSource-Politik der Bell Labs
 - **Prägend** für viele Begrifflichkeiten der systemnahen Informatik
 - Durch **POSIX** standardisiert auf Ebene der Benutzerschnittsstelle / API
- **UNIX** ist die **Referenz** für jede neue Betriebssystemplattform!
(Dies ist Segen und Fluch zugleich!)



Motivation: Warum UNIX?

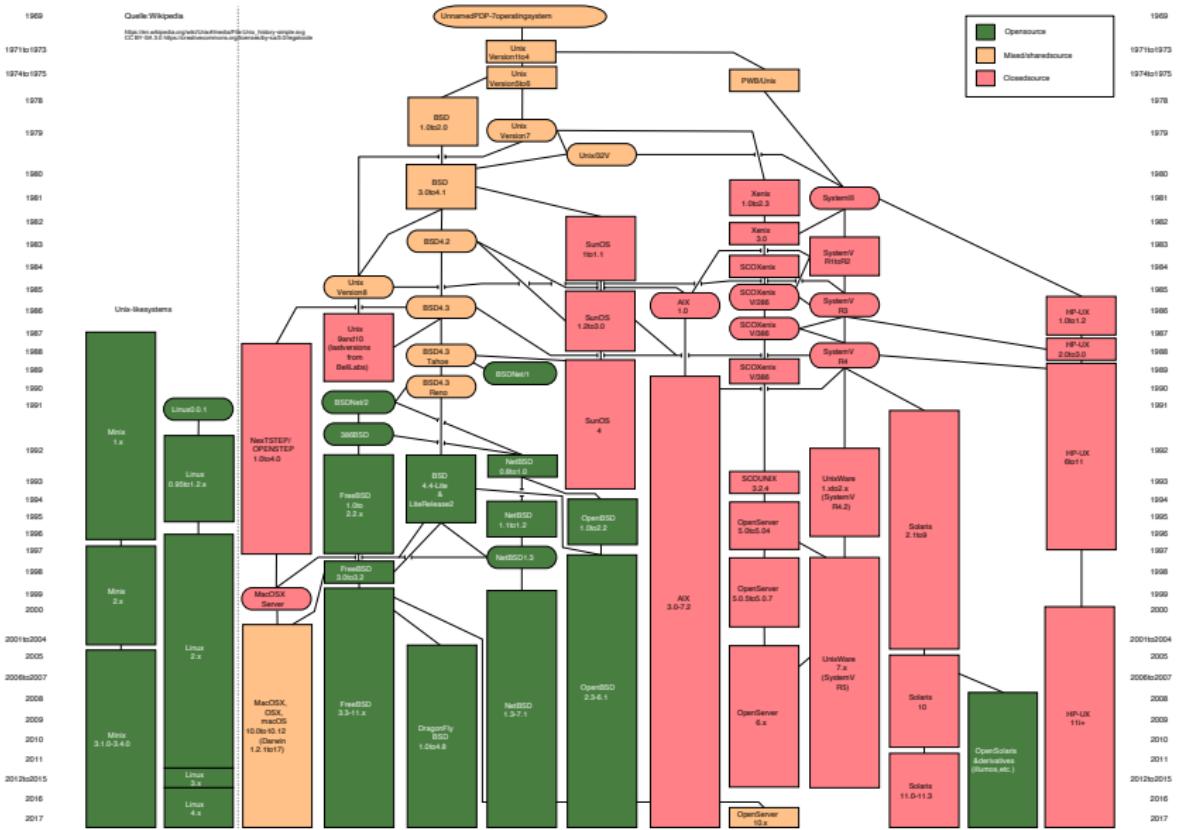


Dennis Ritchie (stehend) und Ken Thompson bei der UNIX-Portierung auf die PDP-11 an 2 Teletype 33 Terminals (1970)



Motivation: Warum UNIX?

Eine bewegte Geschichte!





Motivation: Warum UNIX?



✳️ UNIX – Realitätscheck

Vielzweck- vs Spezialzwecksysteme



Vielzweckbetriebssysteme
(general-purpose operating systems)



Spezialzweckbetriebssysteme
(special-purpose operating systems)



Quelle: Turley. „The Two Percent Solution“, 2002 [36]

- Systemnahe Softwareentwicklung erfolgt überwiegend in **C**
 - **Warum C?** (und nicht Java/Cobol/Scala/<Lieblingssprache>)
- C steht für eine Reihe hier wichtiger Eigenschaften
 - Laufzeiteffizienz (CPU)
 - Übersetzter C-Code läuft direkt auf dem Prozessor
 - Keine Prüfungen auf Programmierfehler zur Laufzeit
 - Einfachheit ("High-Level Assembler")
 - C ist eine Sprache mit sehr wenig Abstraktionen
 - Keine Klassen, keine komplexen Typen.
 - Direktheit (Maschinennähe)
 - C erlaubt den direkten Zugriff auf Speicher und Register
 - Portabilität
 - Es gibt für **jede** Plattform einen C-Compiler
 - C wurde „erfunden“ (1972), um das Betriebssystem UNIX portabel zu implementieren [17, 27]



→ **C** ist die **lingua franca** der systemnahen Softwareentwicklung!



1.4 Literatur



Die Veranstaltung folgt **keinem** klassischem Lehrbuch. Die folgenden Bücher sind daher als **Ergänzung** und Unterstützung der eigenen Mitarbeit in der Vorlesung empfohlen – **nicht als Ersatz!**

■ „Klassische“ Betriebssystemlehrbücher

- [1] Remzi H. Arpacı-Dusseau und Andrea C. Arpacı-Dusseau. *Operating Systems: Three Easy Pieces* (version 1.10). <https://pages.cs.wisc.edu/~remzi/OSTEP/>. 2023
- [32] William Stallings. *Operating Systems: Internals and Design Principles*. 9. Aufl. Pearson Education, 2018. ISBN: 978-0-13-467095-9
- [29] Abraham Silberschatz, Greg Gagne und Peter B. Galvin. *Operating System Concepts*. 10. Aufl. John Wiley & Sons, 2018. ISBN: 978-1-118-06333-0
- [34] Andrew S. Tanenbaum und Herbert Bos. *Modern Operating Systems*. 5. Aufl. Pearson Education, 2023. ISBN: 978-1-292-45966-0

■ Rechnerorganisation und Ebenenmodell

- [33] Andrew S. Tanenbaum. *Structured Computer Organization*. Fifth. Prentice Hall PTR, 2006. ISBN: 978-0131485211

■ Programmieren in C

- [8] Manfred Dausmann, Ulrich Bröckl, Dominic Schoop u. a. *C als erste Programmiersprache: Vom Einsteiger zum Fortgeschrittenen*. Vieweg+Teubner, 2010. ISBN: 978-3834812216. URL: <https://link.springer.com/content/pdf/10.1007%2F978-3-8348-9879-1.pdf>

1.5 Aufbau und Organisation der Veranstaltung



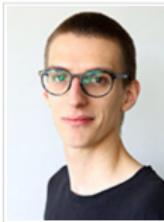
Beteiligte Personen

Vorlesung



Christian Dietrich

Übung



Sören Tempel

Tutoren für kleine Übungen

- Robin D'Andrea
- Steen Leeker
- Benjamin Zuidgeest
- Fangjie Lian
- Florian Lipp
- Florian Kokotz

- Vermittlung der grundlegenden Konzepte und Prinzipien
 - Essenz der fundamentalen Betriebssystemkonzepte
 - Einordnung in den Kanon der Informatik
 - **Grundlage für die Übung**
- Termin: Dienstags 9:45 - 11:15
- Ort: PK 11.1
 - **und** zeitgleich online via BBB
- Stud.IP: [Vorlesung ↗](#)

- Handout der Vorlesungsfolien wird über Stud.IP zur Verfügung gestellt
 - Als **zusammenhängendes** PDF-Dokument (Inhaltsverzeichnis, Querverweise)
 - Handout enthält (in geringem Umfang) zusätzliche Informationen
- Das Vorlesungsskript ist ein iteratives Projekt
 - Wird gelegentlich (vor der Vorlesung) aktualisiert
 - Fehler gefunden? Mail an dietrich@ibr.cs.tu-bs.de

Das Handout kann eine eigene Mitschrift nicht ersetzen!

- Ein Skript ist **kein** Lehrbuch.
- Folien sind nur die „Bildspur“ – und auch davon nur ein Teil.
~~> Skript als **Grundlage** für die eigene Mitschrift – **nicht Ersatz!**



- Große Übung (wöchentlich)
 - Termin: Freitags 09:45 - 11:15 ab 28.10.2024
 - Ort: PK 11.1
 - Die Tafelübung startet den Übungszyklus.
- Kleine Übungen
 - Individuelle Vorstellung der Abgaben
 - Termin: mehrere Termine, ab 27.10.2024
 - Ort: Informatikzentrum 146 (Plaza-Ebene)
 - Terminvergabe mittels Stud.IP
- Inhalte aus Vorlesung und gr. Übung
 - 6 Aufgaben mit Abgabe und individuellem Code-Interview
 - Partnerarbeit möglich (max. Zweiergruppen)
- Stud.IP: [Übung ↗](#)



Semesterplan (im StudIP)

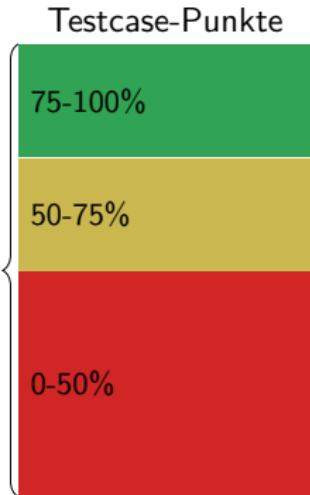
SW	Datum	V	BS	
			Di, 9:45-11:15	Fr, 9:45-11:15
1	Mo, 20. Okt 25			
1	Di, 21. Okt 25			
1	Mi, 22. Okt 25			
1	Do, 23. Okt 25			
1	Fr, 24. Okt 25			
1	Sa, 25. Okt 25			V1: Einführung
1	So, 26. Okt 25			
2	Mo, 27. Okt 25			
2	Di, 28. Okt 25	TÜ: Lilo		
2	Mi, 29. Okt 25			
2	Do, 30. Okt 25			
2	Fr, 31. Okt 25			
2	Sa, 1. Nov 25			
2	So, 2. Nov 25			
3	Mo, 3. Nov 25			
3	Di, 4. Nov 25	V2: Systemnahe Softwareentwicklung in C		
3	Mi, 5. Nov 25			
3	Do, 6. Nov 25			
3	Fr, 7. Nov 25		TÜ: Halde	
3	Sa, 8. Nov 25			
3	So, 9. Nov 25			
4	Mo, 10. Nov 25			
4	Di, 11. Nov 25	V3: Grundlegende Konzepte		
4	Mi, 12. Nov 25			
4	Do, 13. Nov 25			
4	Fr, 14. Nov 25		TÜ: C-Extended	
4	Sa, 15. Nov 25			
4	So, 16. Nov 25			
5	Mo, 17. Nov 25			
5	Di, 18. Nov 25	V4: Dateien und Dateisysteme		
5	Mi, 19. Nov 25			

Seite

- Prüfungsleistung: Schriftliche Klausur
 - Dauer: 90 Minuten, schriftlich
 - Termin: 16. Februar 2026 (vorläufig)
 - Programmieren ist essentiell zum bestehen (~50%)!
 - **Anmeldezeitraum des Prüfungsamts beachten!**
- Studienleistung: Hausaufgaben
 - Code-Abgabe für **jedes** der sechs Übungsblätter mit **mind. 50%** der automatischen Test-Case Punkte
 - Code-Interview wenn Sie gewürfelt werden ($p \approx 0.75, \sigma = 4.5$) erfolgreiches 15 Minuten Einzelgespräch mit einem Tutor 1× hervorragend kann 1× ungenügend ausgleichen
 - **Anmeldung:** Bis 27. Oktober (23:59)
StudIP → Teilnehmende → Gruppen
- Das Modul ist bestanden, wenn Klausur **und** Studienleistung bestanden sind.

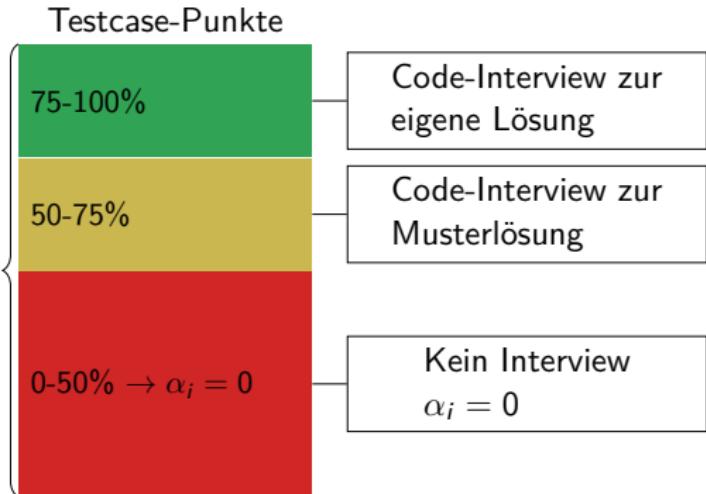
6 Hausaufgaben

- Abgabe im Gitlab
- CI/CD Testcases
- 2 Wochen Zeit



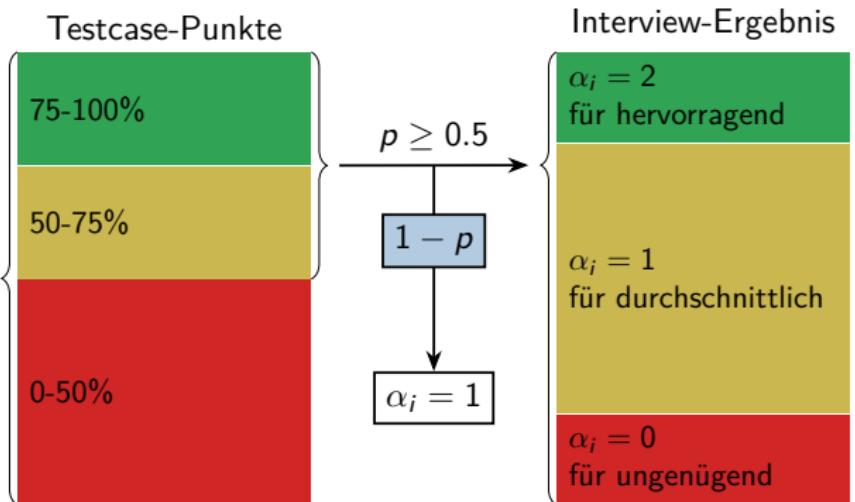
6 Hausaufgaben

- Abgabe im Gitlab
- CI/CD Testcases
- 2 Wochen Zeit



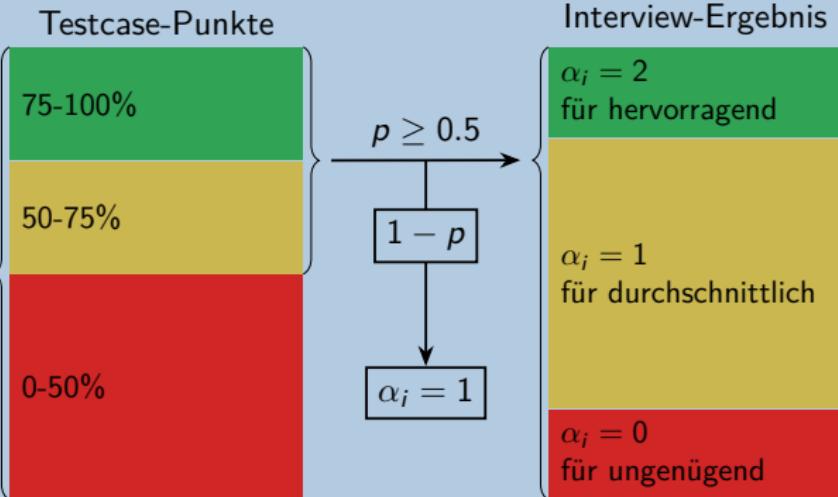
6 Hausaufgaben

- Abgabe im Gitlab
- CI/CD Testcases
- 2 Wochen Zeit



$$\text{Studienleistung} \iff \sum \alpha_i \geq 6$$

6 Hausaufgaben
– Abgabe im Gitlab
– CI/CD Testcases
– 2 Wochen Zeit





1.6 Semesterüberblick

1. Übungspartner:in
2. Kommiliton:innen
3. Stud.IP Forum
4. Übungstutoren
5. Übungsleiter (Sören Tempel) bs@ibr.cs.tu-bs.de
6. Dozent (Christian Dietrich) bs@ibr.cs.tu-bs.de

Reihenfolge beachten!

- Fragen zum Inhalt werden **nur im Forum** beantwortet
 - Studentische Antworten sind hier **sehr gerne gesehen**
 - Es gibt keine falschen Fragen – und keine Nachteile durch Antworten
 - Trauen Sie sich auch zu antworten – man lernt eine Menge dabei!
- Wir moderieren, korrigieren und ergänzen nach Bedarf. **Nur Mut!**



Semesterüberblick

Woche Di, 9:45

Fr, 9:45	Abgabe
VL 1 Einführung	—
31.10	—
TÜ: halde	lilo
TÜ: C-Extended	halde
TÜ: crawl	crawl
TÜ: Tooling	clash
TÜ: clash	clash
TÜ: Shellskripte	patric
TÜ: patric	clash
TÜ: Verlässliches C	patric
TÜ: ticker	ticker
TÜ: Probeklausur (PK)	ticker
TÜ: PK Besprechung	—
16.02.26 Schriftliche Klausur	—

Termine



1.7 Ablauf und Inhalt der Übungen

■ Große Übung

- Vorstellung der neuen Aufgabe
- Hinweise und Wissen zur Bearbeitung
- Beispiellösung verwandter Aufgaben
- Nachbesprechung Beispielklausur

■ Kleine Übung

- Vorstellung der Aufgabe in Einzelinterviews
- Fachdiskussion mit Kommilitonen / Gruppenpartner
- **Eigenen Computer mitbringen**





Semesterplan (im StudIP)

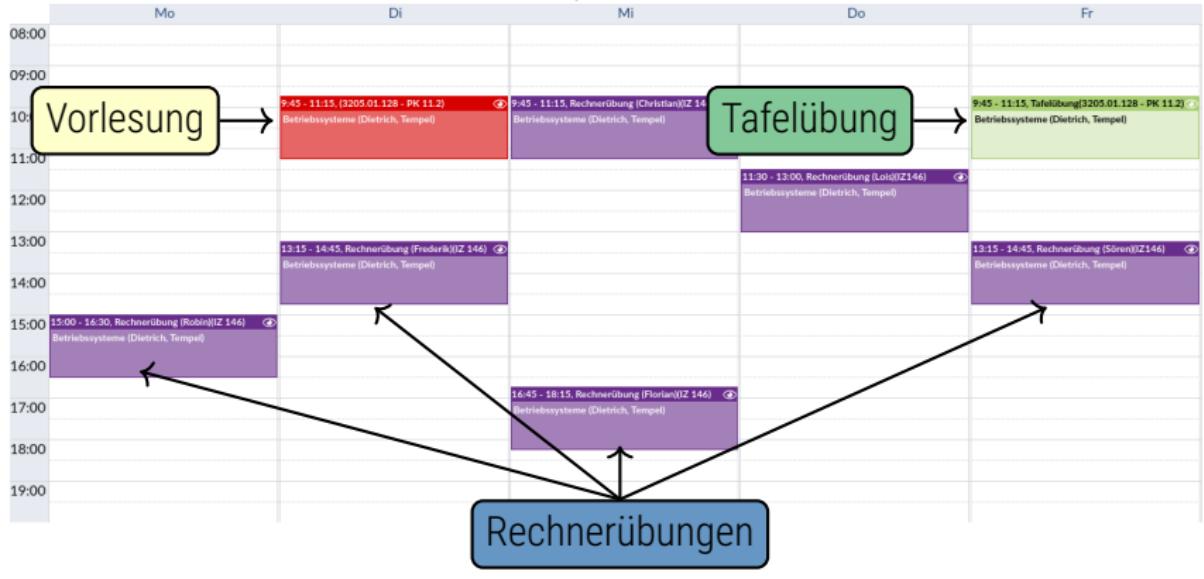
SW	Datum	BS	
		Di, 9:45-11:15	Fr, 9:45-11:15
1	Mo, 20. Okt 25		
1	Di, 21. Okt 25		
1	Mi, 22. Okt 25		
1	Do, 23. Okt 25		
1	Fr, 24. Okt 25		V1: Einführung
1	Sa, 25. Okt 25		
1	So, 26. Okt 25		
2	Mo, 27. Okt 25		
2	Di, 28. Okt 25	TÜ: Lilo	
2	Mi, 29. Okt 25		
2	Do, 30. Okt 25		
2	Fr, 31. Okt 25		
2	Sa, 1. Nov 25		
2	So, 2. Nov 25		
3	Mo, 3. Nov 25		
3	Di, 4. Nov 25	V2: Systemnahe Softwareentwicklung in C	
3	Mi, 5. Nov 25		
3	Do, 6. Nov 25		
3	Fr, 7. Nov 25		TÜ: Halde
3	Sa, 8. Nov 25		
3	So, 9. Nov 25		
4	Mo, 10. Nov 25		
4	Di, 11. Nov 25	V3: Grundlegende Konzepte	
4	Mi, 12. Nov 25		
4	Do, 13. Nov 25		
4	Fr, 14. Nov 25		TÜ: C-Extended
4	Sa, 15. Nov 25		
4	So, 16. Nov 25		
5	Mo, 17. Nov 25		
5	Di, 18. Nov 25	V4: Dateien und Dateisysteme	
5	Mi, 19. Nov 25		

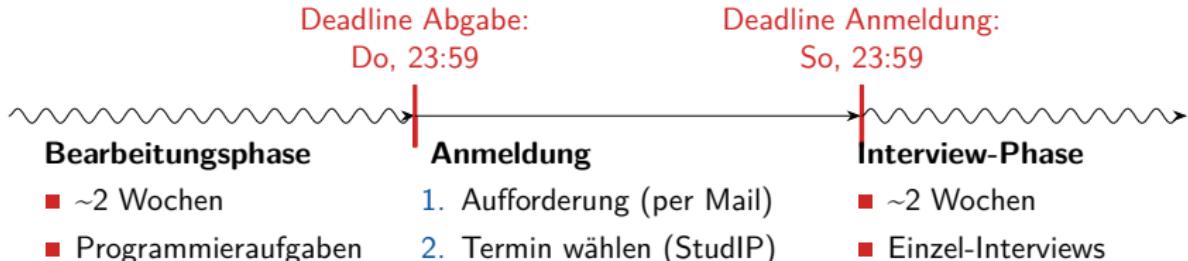
Seite



Übungstermine

Mein Stundenplan im WiSe 2024/25





- **Ziel der Studienleistung** ist Sie auf die Klausur vorzubereiten
 - Wer die Hausaufgabe selbst gelöst hat, für den soll das Interview keine Hürde.
 - Die Verantwortung für die Abgabe tragen **allein Sie!**
 - **KI:** "Wer beim Training betrügt ist dämlich." – Seien Sie nicht dämlich.

- Bearbeitung der Aufgaben (ca. 2 Wochen)
 - Sie dürfen die Aufgaben in Zweiergruppen bearbeiten
 - Die Nutzung von KI ist **verboten** (und dämlich)
 - Die Gitlab CI sagt Ihnen direkt Ihre Testcase-Punktzahl.
- Bearbeitung in Zweiergruppe (optional)
 - **Beide Partner müssen in Ihrem Git-Repository abgeben**
 - Partner kann für jede Abgabe variieren
 - PARTNER-Datei mit y-Nummer und Name des Gruppenpartners
 - Organisieren Sie sich selbst!
- Abgabe der Aufgabe
 - **Deadline:** Donnerstag 23:59 Uhr MEZ vor der nächsten Tafelübung.
 - **Jeder** muss eine Abgabe in seinem Repository haben.
 - Der letzte fristgerechte `git push` wird gewertet.

- Wer muss zum Code-Interview?
 - Wir sampeln mit ca $p \approx 0.75$.
 - Am Freitag erhalten viele eine E-Mail von uns (TU Account)
 - **Sie müssen Sich zu einem Interview-Termin anmelden!**
Separate StudIP Gruppe, Freitag 18:00 bis Sonntag 23:59
 - **Deadline:** Sonntag 23:59 MEZ nach der Abgabe
- Wie bereite ich mich auf das Interview vor?
 - Reflektieren Sie: Wissen, Verstehen, Verwenden.
 - Diskutieren Sie mit Komilitonen mögliche Fragen.
 - Verwenden Sie gerne KI um sich zu prüfen.



- Standard-Workload für 5 LP-Veranstaltung (laut NHG): 125–150 h

12 VL	a	1,5 h	18	h	VL Vor-/Nachbereitung	6	h
10 TÜ	a	1,5 h	15	h			
6 GÜ	a	1,5 h	9	h	6 Aufgaben	a	12 h
Beispielklausur			3	h			72 h
Klausur			1,5 h		Klausurvorbereitung		17 h
Kontaktzeit:				45	Selbststudium:		
							95 h

- Werte beziehen sich auf eine(n) **durchschnittliche(n) Studierende(n)**
 - Übungen sind **essentieller Teil** der Prüfungsvorbereitung
 - Etwa 90 von 140 Stunden Workload entfallen auf die Übung
 - Übungserfolg → Klausurerfolg
- **Nehmen Sie die Übung als Vorbereitung wahr!**



1.8 Zusammenfassung

Die **Funktionsweise** eines Betriebssystems zu verstehen hilft,
Phänomene eines Rechensystems zu begreifen und besser einzuschätzen.

→ **Wissen was dahinter steckt!**

- Wir werden dazu...
 1. einen „Hauch“ Rechnerorganisation und Softwaretechnik „einatmen“
 2. Betriebssystemabstraktionen kennen- und unterscheiden lernen
 3. Funktionen von Betriebssystemen im Detail untersuchen
 4. Aspekte der systemnahe Softwareentwicklung „erfahren“
- Zusammenhänge stehen im Vordergrund!
 - Leitfaden ist die ganzheitliche Betrachtung von Systemfunktionen
 - skizziert wird eine logische Struktur ggf. vieler Ausprägungsformen
 - klassischer Lehrbuchstoff wird ergänzt, weniger repetiert oder vertieft



1 Einführung und Organisation

2 Exkurs: Systemnahe C-Programmierung

- 2.1 Einordnung
- 2.2 Zeiger – Einführung
- 2.3 Zeiger und Funktionen
- 2.4 Felder – Einführung
- 2.5 Syntaktische Äquivalenz
- 2.6 Zeigerarithmetik
- 2.7 Felder und Funktionen
- 2.8 Strukturen, Bitfelder und Unions
- 2.9 Zusammenfassung

3 Grundlegende Konzepte



2.1 Einordnung

Worum geht es in diesem Kapitel?

- **Fokussierung** auf Sprachmerkmalen der Programmiersprache C
 - Blickwinkel: Systemnahe Softwareentwicklung in C
 - Schwerpunkt: Abbild und Adressierung von Daten im Speicher
 - Ziel: Systemnahen C-Code lesen und verstehen können
 - Es geht dabei nur am Rand um Betriebssysteme
 - Die UNIX-Betriebssystemschnittstelle ist jedoch „C-lastig“
 - Hardwarestrukturen lassen sich gut auf C-Datenstrukturen abbilden
- C ist domänenspezifische Sprache zur Beschreibung vieler BS-Strukturen



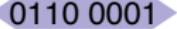
2.2 Zeiger – Einführung



Einordnung: Zeiger (*Pointer*)

■ **Literal:** 'a'

Darstellung eines Wertes

'a' ≡ 

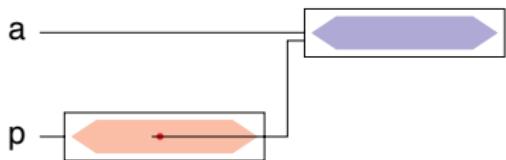
■ **Variable:** `char a;`

Behälter für einen Wert



■ **Zeiger-Variable:** `char *p = &a;`

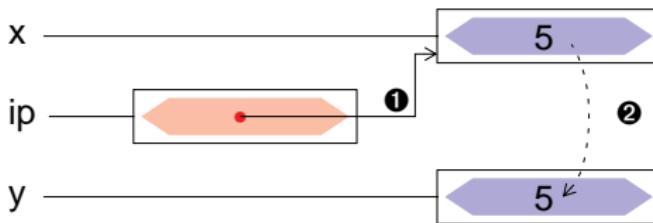
Behälter für eine Referenz
auf eine Variable



- Eine Zeigervariable (*Pointer*) enthält als Wert die **Adresse** einer anderen Variablen
 - Ein Zeiger verweist auf eine Variable (im Speicher)
 - Über die Adresse kann man **indirekt** auf die Zielvariable (ihren Speicher) zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
 - Funktionen können Variablen des Aufrufers verändern (*call-by-reference*)
 - Speicher lässt sich direkt ansprechen
 - Effizientere Programme
- Aber auch viele Probleme!
 - Programmstruktur wird unübersichtlicher
(welche Funktion kann auf welche Variablen zugreifen?)
 - Zeiger sind die **häufigste Fehlerquelle** in C-Programmen!

- **Zeigervariable** := Behälter für Verweise (\mapsto Adresse)
- Syntax (Definition): Typ ***Bezeichner;**
- Beispiel

```
int x = 5;  
  
int *ip;  
  
int y;  
  
ip = &x; ①  
  
y = *ip; ②
```



- Adressoperator: $\& x$ Der unäre $\&$ -Operator liefert die **Referenz** (\mapsto Adresse im Speicher) der Variablen **x**.
- Verweisoperator: $* y$ Der unäre $*$ -Operator liefert die **Zielvariable** (\mapsto Speicherzelle / Behälter), auf die der Zeiger **y** verweist (Dereferenzierung).
- Es gilt: $(* (\& x)) \equiv x$ Der Verweisoperator ist die Umkehroperation des Adressoperators.

Achtung: Verwirrungsgefahr (***Ich seh überall Sterne* ***)

Das $*$ -Symbol hat in C verschiedene Bedeutungen, **je nach Kontext**

1. Multiplikation (binär): $x * y$ in Ausdrücken
2. Typmodifizierer: `uint8_t *p1, *p2` in Definitionen und Deklarationen
`typedef char *CPTR`
3. Verweis (unär): $x = *p1$ in Ausdrücken

Insbesondere 2. und 3. führen zu Verwirrung

$\rightsquigarrow *$ wird fälschlicherweise für ein Bestandteil des Bezeichners gehalten.

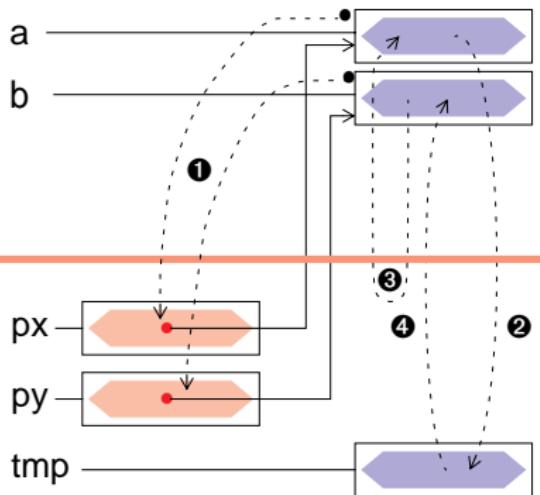


2.3 Zeiger und Funktionen

- Parameter werden in C immer *by-value* übergeben
 - Parameterwerte werden in lokale Variablen der aufgerufenen Funktion kopiert
 - Aufgerufene Funktion kann tatsächliche Parameter des Aufrufers nicht ändern
 - Das gilt auch für Zeiger (Verweise)
 - Aufgerufene Funktion erhält eine Kopie des Adressverweises
 - Mit Hilfe des *****-Operators kann darüber jedoch auf die Zielvariable zugegriffen werden und diese verändert werden
- ~~ **Call-by-reference**

■ Beispiel (Gesamtüberblick)

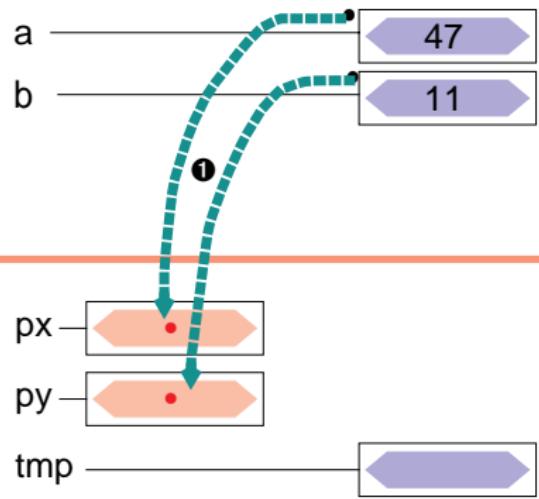
```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b); ①  
    ...  
}  
  
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
    *py = tmp; ④  
  
}
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap (&a, &b); ❶
```

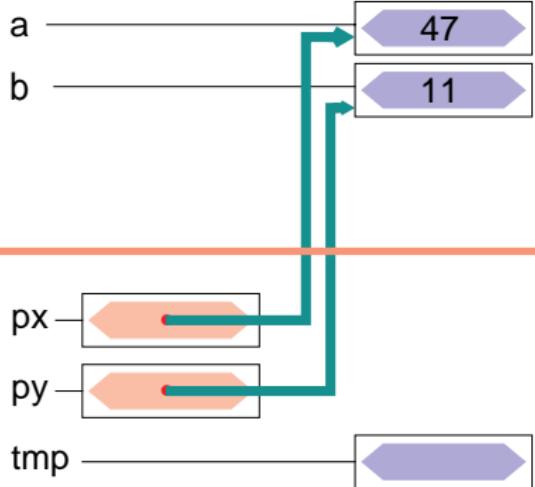
```
void swap (int *px, int *py)  
{  
    int tmp;
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap (&a, &b);
```

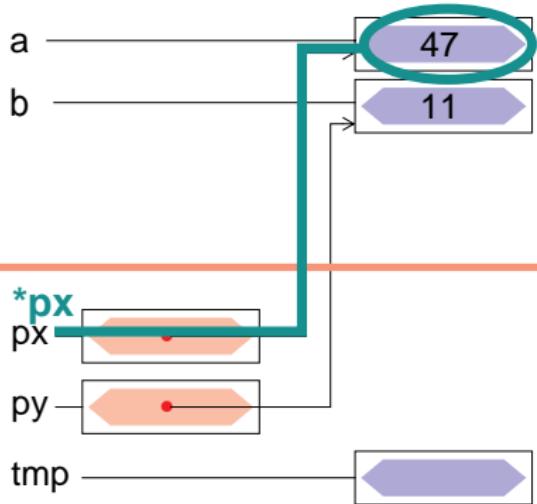
```
void swap (int *px, int *py)  
{  
    int tmp;
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap (&a, &b);
```

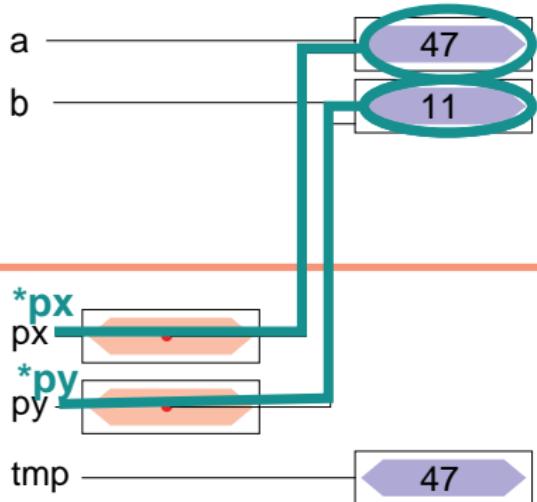
```
void swap (int *px, int *py)  
{  
    int tmp;  
    tmp = *px; ②
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap (&a, &b);
```

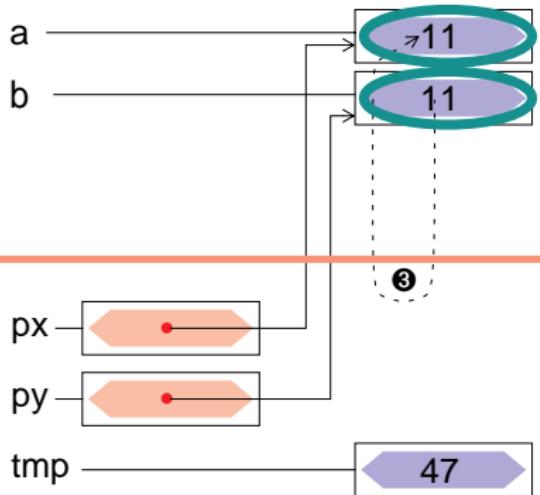
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap (&a, &b);
```

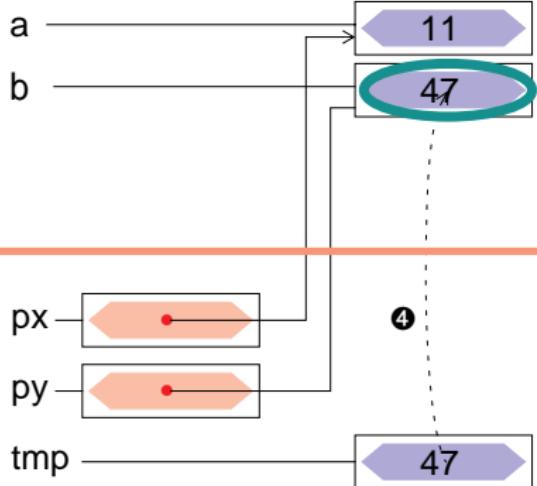
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③
```



■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap (&a, &b);
```

```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
    *py = tmp; ④  
}
```

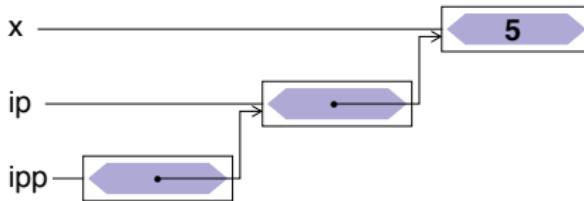




- Ein Zeiger kann auch auf eine Zeigervariable verweisen

```
int x = 5;
int *ip = &x;

int **ipp = &ip;
/* → **ipp = 5 */
```



- Wird vor allem bei der Parameterübergabe an Funktionen benötigt
 - Zeigerparameter *call-by-reference* übergeben
(z. B. `swap()`-Funktion für Zeiger)
 - Ein Feld von Zeigern übergeben



2.4 Felder – Einführung

- **Feldvariable** := Behälter für eine Reihe von Werten desselben Typs

- Syntax (Definition): *Typ Bezeichner [IntAusdruck];*

- *Typ* Typ der Werte

[=Java]

- *Bezeichner* Name der Feldvariablen

[=Java]

- *IntAusdruck* **Konstanter** Ganzzahl-Ausdruck, definiert die Feldgröße
(→ Anzahl der Elemente).

[≠Java]

Ab **C99** darf *IntAusdruck* bei **auto**-Feldern
auch **variabel** (beliebig, aber fest) sein.

- Beispiele:

```
static int prim[8 * 2];    // constant, fixed array size

void f(int n) {
    char a[MAX_PATH];      // constant, fixed array size
    char b[n];              // C99: variable, fixed array size
}
```



Feldinitialisierung

- Wie andere Variablen auch, kann ein Feld bei Definition eine **initiale Wertzuweisung** erhalten

```
uint8_t LEDs[4] = { RED0, YELLOW0, GREEN0, BLUE0 };
int prim[5]      = { 2, 3, 5, 7, 11 };
```

- Werden zu wenig Initialisierungselemente angegeben, so werden die restlichen Elemente **mit 0 initialisiert**

```
uint8_t LEDs[4] = { RED0 };           // => { RED0, 0, 0, 0 }
int prim[5]   = { 2, 3, 5 };          // => { 2, 3, 5, 0, 0 }
```

- Wird die explizite Dimensionierung ausgelassen, so bestimmt **die Anzahl** der Initialisierungselemente die Feldgröße

```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 }; // => uint8_t LEDs[4]
int prim[]     = { 2, 3, 5, 7, 11 };                // => int prim[5]
```



■ Syntax: *Feld [IntAusdruck]*

[=Java]

- Wobei $0 \leq \text{IntAusdruck} < n$ für $n = \text{Feldgröße}$

- **Achtung:** Feldindex wird nicht überprüft
~~ häufige Fehlerquelle in C-Programmen

[≠Java]

■ Beispiel

```
int prim[] = { 2, 3, 5, 7, 11 };
int res    = 0;

prim[3] = 4711;

for (int i = 0; i < 5; i++) {
    printf("%d, ", prim[i]);      // Output: 2, 3, 5, 4711, 11,
}

prim[5] = 13;      // UNDEFINED!!! Possibly => res = 13 ?
```

prim[5] = 13 bewirkt einen Zugriff auf den Speicher **hinter** **prim**. Die genauen Auswirkungen hängen auch von der Codegenerierung des Übersetzers ab. In diesem Fall wird *vermutlich* die Variable **res** überschrieben.

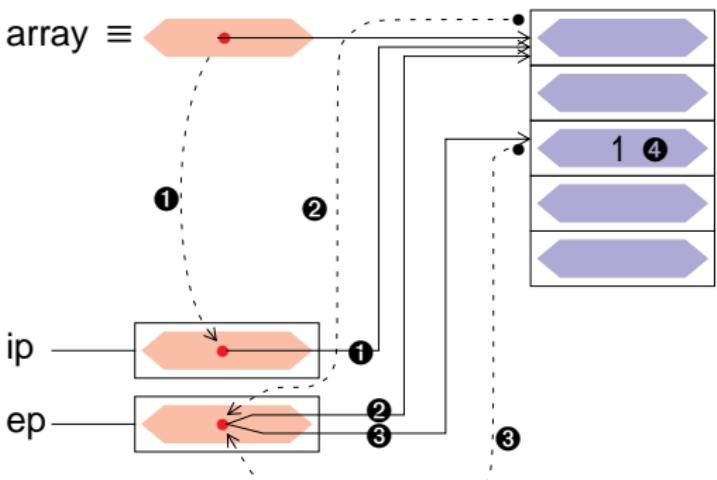


2.5 Syntaktische Äquivalenz

Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: $\text{array} \equiv \&\text{array}[0]$
 - Ein Alias – kein Behälter \rightsquigarrow Wert kann nicht verändert werden
 - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Gesamtüberblick)

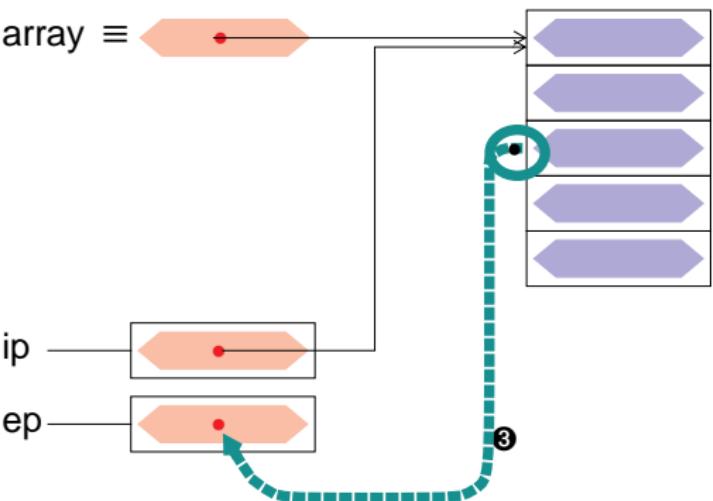
```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③  
  
*ep = 1; ④
```



Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: $\text{array} \equiv \&\text{array}[0]$
 - Ein Alias – kein Behälter \rightsquigarrow Wert kann nicht verändert werden
 - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

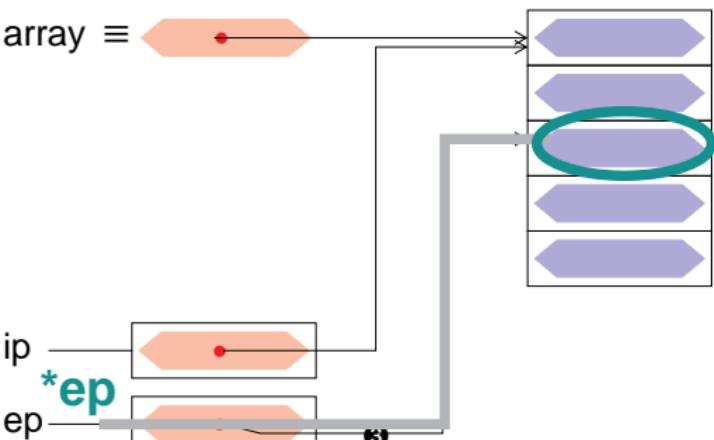
```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③
```



Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: $\text{array} \equiv \&\text{array}[0]$
 - Ein Alias – kein Behälter \leadsto Wert kann nicht verändert werden
 - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③  
  
*ep = 1; ④
```



- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: $\text{array} \equiv \&\text{array}[0]$
- Diese Beziehung gilt in beide Richtungen:
 - Ein Zeiger kann wie ein Feld verwendet werden
 - Insbesondere kann der **[]**-Operator angewandt werden
- Beispiel (vgl. \hookrightarrow 2-16)

 \hookrightarrow 2-16

```
int prim[] = { 2, 3, 5, 7, 11 };
int *p      = prim;

for (int i = 0; i < 5; i++) {
    printf("%d, ", p[i]);    // Output: 2, 3, 5, 7, 11
}
```



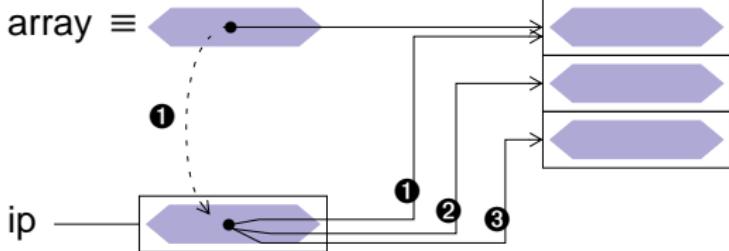
2.6 Zeigerarithmetik

Rechnen mit Zeigern

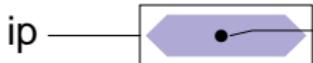
- Im Unterschied zu einem Feldbezeichner ist eine Zeigervariable ein Behälter \rightsquigarrow Ihr Wert ist veränderbar
- Neben einfachen Zuweisungen ist dabei auch **Arithmetik** möglich

```
int array[3];
int *ip = array; ①

ip++; ②
ip++; ③
```



```
int array[5];
ip = array; ①
```



$$(ip+3) \equiv \&ip[3]$$

Bei der Zeigerarithmetik wird immer die Größe des Objekttyps mit berücksichtigt.

■ Arithmetische Operationen

`++` Prä-/Postinkrement

~~ Verschieben auf das nächste Objekt

`--` Prä-/Postdekrement

~~ Verschieben auf das vorangegangene Objekt

`+, -` Addition / Subtraktion eines `int`-Wertes

~~ Ergebniszeiger ist verschoben um n Objekte

`-` Subtraktion zweier Zeiger

~~ Anzahl der Objekte n zwischen beiden Zeigern (Distanz)

■ Vergleichsoperationen: `<`, `<=`, `==`, `>=`, `>`, `!=`

~~ Zeiger lassen sich wie Ganzzahlen vergleichen und ordnen

❖ Felder sind Zeiger sind Felder – Zusammenfassung

- In Kombination mit Zeigerarithmetik lässt sich in C jede Feldoperation auf eine äquivalente Zeigeroperation abbilden.
- Für `int i, array[N], *ip = array;` mit $0 \leq i < N$ gilt:

$$\begin{array}{lllll} \text{array} & \equiv & \&\text{array}[0] & \equiv \quad \text{ip} \quad \equiv \quad \&\text{ip}[0] \\ \ast\text{array} & \equiv & \text{array}[0] & \equiv \quad \ast\text{ip} \quad \equiv \quad \text{ip}[0] \\ \ast(\text{array} + i) & \equiv & \text{array}[i] & \equiv \quad \ast(\text{ip} + i) \quad \equiv \quad \text{ip}[i] \\ & & \text{array}++ & \not\equiv \quad \text{ip}++ \end{array}$$

Fehler: array ist konstant!

- Umgekehrt können Zeigeroperationen auch durch äquivalente Feldoperationen dargestellt werden.
- Im Gegensatz zum Zeiger kennzeichnet der Feldbezeichner jedoch **keine Variable**. Er ist somit nicht durch Zuweisungen veränderbar.

- Zeichen sind in C Ganzzahlen (Integers)
 - `char` gehört zu den Integer-Typen (üblicherweise 8 Bit = 1 Byte)
- Repräsentation erfolgt durch den ASCII-Code
 - 7-Bit-Code \mapsto 128 Zeichen standardisiert
(die verbleibenden 128 Zeichen werden unterschiedlich interpretiert)
 - Spezielle Literalform durch Hochkommata
`'A'` \mapsto ASCII-Code von A
 - Nichtdruckbare Zeichen durch Escape-Sequenzen
 - Tabulator `'\t'`
 - Zeilentrenner `'\n'`
 - Backslash `'\\'`
- Zeichen \mapsto Integer \rightsquigarrow man kann mit Zeichen rechnen

```
char b = 'A' + 1;                   // b: 'B'  
  
int lower(int ch) {                // lower('X'): 'x'  
    return ch + 0x20;  
}
```

2-25



ASCII-Code-Tabelle (7 Bit)

ASCII \mapsto American Standard Code for Information Interchange

NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
00	01	02	03	04	05	06	07
BS	HT	NL	VT	NP	CR	SO	SI
08	09	0A	0B	0C	0D	0E	0F
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
10	11	12	13	14	15	16	17
CAN	EM	SUB	ESC	FS	GS	RS	US
18	19	1A	1B	1C	1D	1E	1F
SP	!	"	#	\$	%	&	,
20	21	22	23	24	25	26	27
()	*	+	/	-	*	/
28	29	2A	2B	2C	2D	2E	2F
0	1	2	3	4	5	6	7
30	31	32	33	34	35	36	37
8	9	:	:	<	=	>	?
38	39	3A	3B	3C	3D	3E	3F
@	A	B	C	D	E	F	G
40	41	42	43	44	45	46	47
H	I	J	K	L	M	N	O
48	49	4A	4B	4C	4D	4E	4F
P	Q	R	S	T	U	V	W
50	51	52	53	54	55	56	57
X	Y	Z	[\]	^	_
58	59	5A	5B	5C	5D	5E	5F
'	a	b	c	d	e	f	g
60	61	62	63	64	65	66	67
h	i	j	k	l	m	n	o
68	69	6A	6B	6C	6D	6E	6F
p	q	r	s	t	u	v	w
70	71	72	73	74	75	76	77
x	y	z	{		}	~	DEL
78	79	7A	7B	7C	7D	7E	7F

- Ein String ist in C ein Feld (Array) von Zeichen
 - Repräsentation: Folge von Einzelzeichen, terminiert durch (letztes Zeichen): **NUL** (ASCII-Wert 0)
 - Speicherbedarf: (Länge + 1) Bytes
- Spezielle Literalform durch doppelte Hochkommata:

"Hi!" \mapsto

'H'	'i'	'!'	0
-----	-----	-----	---

abschließendes 0-Byte

- Beispiel

```
#include <stdio.h>

char string[] = "Hello, World!\n";

int main(void) {
    printf("%s", string);
    return 0;
}
```



2.7 Felder und Funktionen

- Felder werden in C **immer** als Zeiger übergeben
~~~ *Call-by-reference*

[=Java]

```
static int prim[] = { 2, 3 };

void printints(int *array, unsigned n) {
    for (unsigned i = 0; i < n; i++)
        printf("%d --> %d, ", i + 1, array[i]);
    printf("\n");
}

void main() {
    printints(prim, 2);           // Output: 1 --> 2, 2 --> 3,
    int more[] = { 4711, 4812, 4913 };
    printints(more, 3);          // Output: 1 --> 4711, 2 --> 4812, 3 --> 4913
}
```

- Informationen über die Feldgröße gehen dabei verloren!
  - Die Feldgröße muss explizit als Parameter mit übergeben werden
  - In manchen Fällen kann sie auch in der Funktion berechnet werden (z. B. bei Strings durch Suche nach dem abschließenden **NUL**-Zeichen)

- Felder werden in C **immer** als Zeiger übergeben  
~~~ *Call-by-reference* [=Java]
- Wird der Parameter als **const** deklariert, so kann die Funktion die Feldelemente **nicht verändern** ↪ Guter Stil! [≠Java]

```
void printints(const int *array, unsigned n) {  
    ...  
}
```

- Um anzuzeigen, dass ein Feld (und kein „Zeiger auf Variable“) erwartet wird, ist auch folgende **äquivalente Syntax** möglich:

```
void printints(const int array[], unsigned n) {  
    ...  
}
```

- **Achtung:** Das gilt so nur bei Deklaration eines Funktionparameters
- Bei Variablendefinitionen hat **array[]** eine **völlig andere** Bedeutung (Feldgröße aus Initialisierungsliste ermitteln, ↪ 2-15)

- Die Funktion `int strlen(const char *)` aus der Standardbibliothek liefert die Anzahl der Zeichen im übergebenen String

```
void main() {  
    ...  
    const char *string = "hallo";      // string is array of char  
    unsigned len = strlen(string);    // len = 5  
    ...  
}
```

Dabei gilt: "hallo" \equiv ↗ h ↘ a ↘ l ↘ l ↘ o ↘ \0 ↘

2-26

- Implementierungsvarianten

Variante 1: Feld-Syntax

```
int strlen(const char s[]) {  
    int n = 0;  
    while (s[n] != '\0')  
        n++;  
    return n;  
}
```

Variante 2: Zeiger-Syntax

```
int strlen(const char *s) {  
    const char *end = s;  
    while (*end != '\0')  
        end++;  
    return end - s;  
}
```



2.8 Strukturen, Bitfelder und Unions

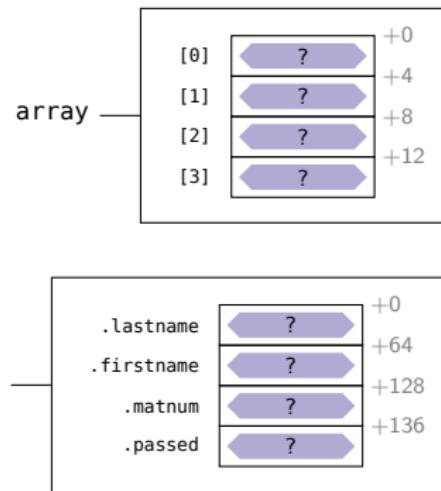
- Ein Feld ist eine Folge von Elementen des **festen Typs**
 - Elemente liegen hintereinander im Speicher.
 - Werden angesprochen durch ihren **Index**.
- Eine **Struktur (struct)** ist ein Folge von Elementen **beliebigen Typs**
 - Elemente liegen ebenfalls hintereinander im Speicher abgelegt.
 - Werden angesprochen durch ihren **Elementnamen**.

```
// Array definition
int array[4];
```



```
// Structure declaration
struct Student {
    char lastname[64];
    char firstname[64];
    long matnum;
    int passed;
};

// Variable definition
struct Student stud;
```



Strukturen: Variablendefinition und -initialisierung

- Analog zu einem Array kann eine Strukturvariable bei Definition elementweise initialisiert werden

→ 2-15

```
struct Student {  
    char lastname[64];  
    char firstname[64];  
    long matnum;  
    int passed;  
};
```

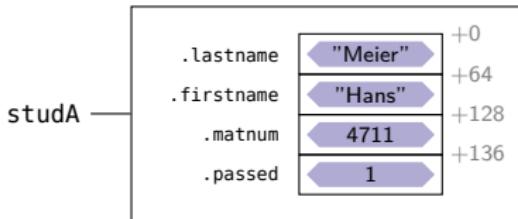
```
struct Student stud = { "Meier", "Hans",  
                        4711, 0 };
```

Die Initialisierer werden nur über ihre Reihenfolge, nicht über ihren Bezeichner zugewiesen.
~~> Potentielle Fehlerquelle bei Änderungen!

- Analog zur Definition von `enum`-Typen kann man mit `typedef` die Verwendung vereinfachen

```
typedef struct {  
    char lastname[64];  
    char firstname[64];  
    long matnum;  
    int passed;  
} student_t;
```

```
student_t studA = {"Meier", "Hans", 4711, 0};  
student_t studB = {"Müller", "Franz", 4812, 1};
```

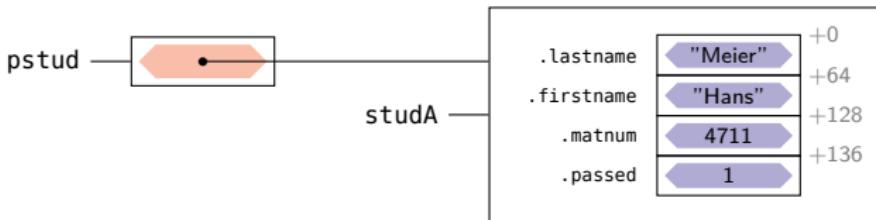


- Auf Strukturelemente wird mit dem `.`-Operator zugegriffen

[≈Java]

```
student_t studA = {"Meier", "Hans", 4711, 0};
```

```
studA.passed = 1; // student has passed BS course
```



- Bei einem Zeiger auf eine Struktur würde Klammerung benötigt

```
student_t studA = {"Meier", "Hans", 4711, 0};  
student_t *pstud = &studA;
```

Beachte: `.` hat eine höhere Priorität als `*`

```
(*pstud).passed = 1; // student has passed BS course
```

- Mit dem `->`-Operator lässt sich dies vereinfachen $s->m \equiv (*s).m$

```
student_t studA = {"Meier", "Hans", 4711, 0};  
student_t *pstud = &studA;  
  
pstud->passed = 1; // student has passed BS course
```

- Strukturelemente können auf Bit-Granularität festgelegt werden
 - Der Compiler fasst Bitfelder zu passenden Ganzzahltypen zusammen
 - Nützlich, um auf einzelne Bit-Bereiche eines Registers zuzugreifen
- Beispiel: Ein Hardwareregister (hier von einem AVR ATmega μ -Controller)
 - EICRA **External Interrupt Control Register A** Steuert Auslöser für externe Interrupt-Quellen INT0 und INT1.
[2, S. 83]

| | | | | | | | | |
|---|---|---|---|---|-------|-------|-------|-------|
| | | | | | ISC11 | ISC10 | ISC01 | ISC00 |
| 7 | 6 | 5 | 4 | 3 | R/W | R/W | R/W | R/W |

```
typedef struct {
    uint8_t ISC0      : 2;      // bit 0-1: interrupt sense control INT0
    uint8_t ISC1      : 2;      // bit 2-3: interrupt sense control INT1
    uint8_t reserved : 4;      // bit 4-7: reserved for future use
} EICRA_t;
```

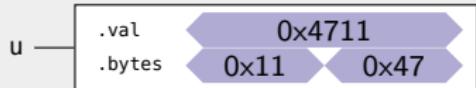


- In einer Struktur liegen die Elemente **hintereinander** im Speicher, in einer Union hingegen **übereinander**
 - Nutzung 1: Speicher sparen bei Variablen mit schnittfreien Lebenszeiten
 - Nutzung 2: Bitweise Typ-Casts; valide in C99, verboten in C++.
- Beispiel

```
union {
    uint16_t val;      // 16-bit unsigned integer
    uint8_t bytes[2]; // array of 2 8-bit unsinged integer
} u;

// little-endian machine
u.val = 0x4711;

uint8_t low   = u.bytes[0]; // ==> low   = 0x11
uint8_t high  = u.bytes[1]; // ==> high  = 0x47
```



Beispiel zeigt eine **little-endian Architektur** (z.B. Intel): An der niedrigsten Adresse befindet sich das niederwertigste Byte. Dieses ist genau umgekehrt der „natürlichen“ Schreibweise.



2.9 Zusammenfassung

Das Typsystem von C ermöglicht es, Speicher sehr direkt abzubilden:

- Zeiger verweisen direkt auf Variablen (Speicherzellen) im Speicher
 - Möglichkeit des **indirekten** Zugriffs auf den Wert
 - Grundlage für die Implementierung von *call-by-reference* in C
 - Wichtiges Element der **Maschinennähe** von C
 - **Häufigste Fehlerursache in C-Programmen**
- Felder repräsentieren gleichförmige Folgen von Speicherzellen
 - syntaktische Äquivalenz zwischen Zeigern und Feldern
 - Hauptspeicher \mapsto „großes Feld“
- Strukturen repräsentieren strukturierte Folgen von Speicherzellen
 - Zusammenfassung mehrere Objektattribute (vgl. einer Java-Klasse)
 - Kontrolle über die Abbildung im Speicher
 - Durch Bitfelder bis auf Sub-Byte-Granularität
 - Durch unions mehrere Typinterpretationen derselben Daten



1 Einführung und Organisation

2 Exkurs: Systemnahe C-Programmierung

3 Grundlegende Konzepte

3.1 Einordnung

3.2 Hierarchie Virtueller Maschinen

3.3 Virtualisierung der Hardwareressourcen

3.4 Aufbau und Architektur eines Betriebssystems

3.5 Betriebssystemkonzepte im Überblick

3.6 Zusammenfassung

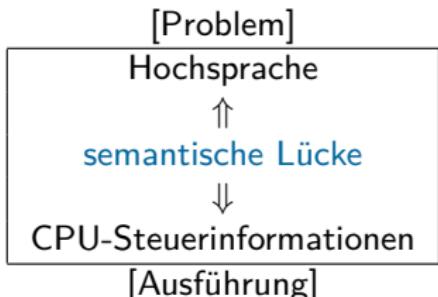
3.1 Einordnung

Worum geht es in diesem Kapitel?

- **Einordnen** des Betriebssystems und seiner Rolle in der Systemarchitektur.
 - Verständnis der Mehrebenenmaschine und partielle Interpretation.
 - Virtualisierung durch Multiplexing und Isolation.
 - Aufgabenteilung, Struktur und Architektur eines Betriebssystems.
- **Kennen** der grundlegenden Aufgaben und Abstraktionen.
 - Virtualisierung in UNIX durch Prozesse und Dateien
 - Prozessverwaltung: Prozessverhalten, Zustände, Ablaufplanung, Simultanverarbeitung, Betriebsmittel, Koordination
 - Speicherverwaltung: Speicherhierarchie, statische/dynamische Allokation, Platzierung, Hintergrundspeicher, Dateien, Verzeichnisse

→ Ziel ist der Überblick – Details in den folgenden Kapiteln

3.2 Hierarchie Virtueller Maschinen



- Informatisches Urproblem:
Schließen der Semantischen Lücke

- Komplexitätsreduktion durch Hierarchie
 - *Divide et impera* – Teile und herrsche
 - Hierarchisch angeordnete **virtuelle Maschinen**
 - Problem schrittweise in einfachere Darstellungen bis auf die reale Maschine herunterbrechen.

- Top-Down vs. Bottom-Up
 - Top-Down: Vom Problem zur Lösung
 - Bottom-Up: Erschaffen von Lösungsräumen

→ Systeminfrastruktur



⌘ Verschiedenheit zwischen Quell- und Zielsprache

Faustregel: { Quellsprache → höheres
Zielsprache → niedrigeres } Abstraktionsniveau

Semantische Lücke (*semantic gap*)

“ The difference between the complex operations performed by high-level constructs and the simple ones provided by computer instruction sets.

It was in an attempt to try to close this gap that computer architects designed increasingly complex instruction set computers. ”

<http://www.hyperdictionary.com/computing/semantic+gap>

→ Kluft zwischen gedanklich Gemeintem und sprachlich Geäußertem



- Schrittweises Schließen der semantischen Lücke durch
 - Interpretation auf einer einfacheren (virtuellen) Maschine
 - Übersetzung für eine einfachere (virtuelle) Maschine



| Ebene | | |
|-------|--|---|
| n | virtuelle Maschine M_n mit
Maschinensprache S_n | Programme in S_n werden von einem auf einer tieferen
Maschine laufenden Interpreter gedeutet oder in Pro-
gramme tieferer Maschinen übersetzt |
| : | : | : |
| 2 | virtuelle Maschine M_2 mit
Maschinensprache S_2 | Programme in S_2 werden von einem auf M_1 bzw. M_0 lau-
fenden Interpreter gedeutet oder nach S_1 bzw. S_0 über-
setzt |
| 1 | virtuelle Maschine M_1 mit
Maschinensprache S_1 | Programme in S_1 werden von einem auf M_0 laufenden
Interpreter gedeutet oder nach S_0 übersetzt |
| 0 | reale Maschine M_0 mit
Maschinensprache S_0 | Programme in S_0 werden direkt von der Hardware aus-
geführt |

- Übersetzung und Interpretation als Techniken, um (auch in Kombination) Programme auf HW zur Ausführung zu bringen.

Wichtig: Wir meinen nicht (nur) VMWare, VirtualBox oder KVM.

Hypervisor:
Maschinenmodell
entspricht einer
Hardwarearchitektur

■ Maschinenmodell der virtuellen Maschine

- **Speicher/Objekte:** Wie kann man Daten ablegen und wieder abrufen?
- **Befehle/Operationen:** Wie kann man Daten miteinander kombinieren?
- Ein „**Prozessor**“ kann dieses Modell implementieren.

→ Das Betriebssystem ist ein „Prozessor“, implementiert ein Maschinenmodell

je nach Standpunkt: **definiert**

■ Maschinenprogramme

- Eine Sequenz von Befehlen, gegen das Maschinenmodell geschrieben.
- Wird von einem Prozessor verarbeitet, um ein Ergebnis zu berechnen.

→ Das Betriebssystem arbeitet Maschinenprogramme ab

■ Sprache der virtuellen Maschine

~~ PSÜ

- Nicht jede Zeichenkette ist ein für jedes Maschinenmodell valides Programm.
- Syntaktische und Semantische Regeln, um valide Programme zu schreiben.

- Jede einzelne Ebene (d.h., Schicht) in der Hierarchie wird durch einen spezifischen **Prozessor** implementiert:
Kom|pi|la|tor lat. (Zusammenträger)

z. B. C/Pascal/Ada \mapsto ASM
ASM \mapsto Binärkode

- ein **Softwareprozessor**, transformiert in einer *Quellsprache* vorliegende Programme in eine semantisch äquivalente Form einer *Zielsprache*

In|ter|pret(er) lat. (Ausleger, Erklärer, Deuter)

- ein **Hard-, Firm- oder Softwareprozessor**, der die Programme direkt ausführt
~~ ausführbares Programm (*executable*) — z. B. JVM
- ggf. **Vorübersetzung** durch einen Kompilierer, um die Programme in eine für die Interpretation günstigere Repräsentation zu bringen — z. B. Java Bytecode

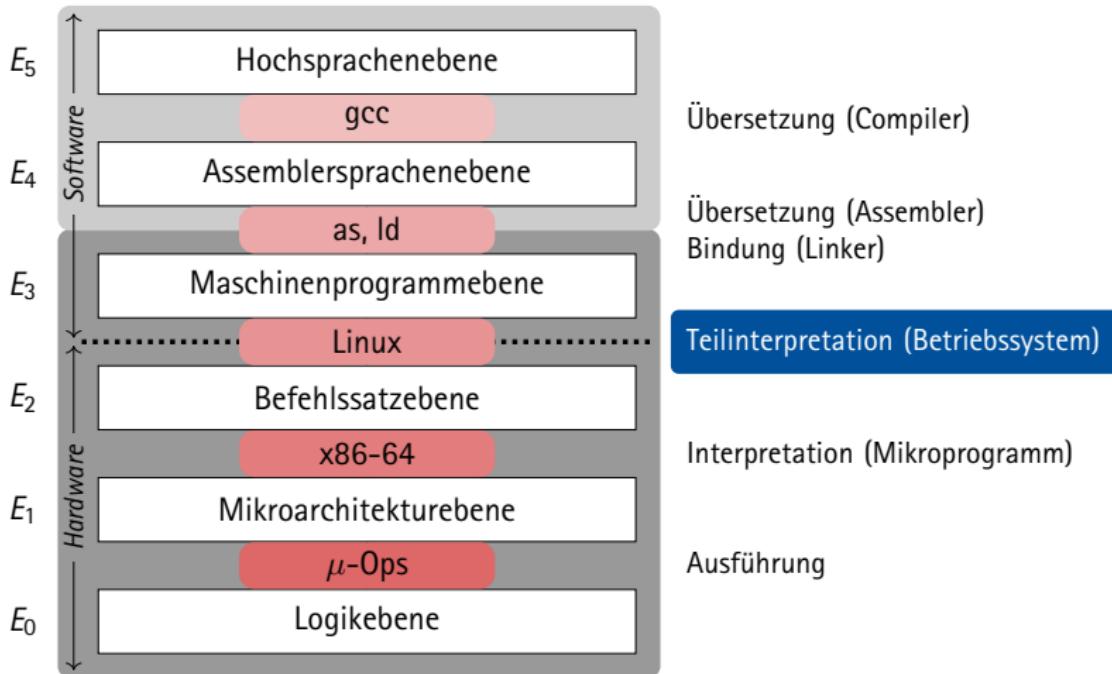
- Abstrakte/reale Prozessoren, die vor (statisch) oder zur (dynamisch) Ausführungszeit des Programms wirken, das sie verarbeiten



Beispiel: Informatikstudium

Mehrebenenmaschine

Übersetzung und Interpretation auf einem Linux-PC \hookrightarrow 3



Beispiel: Matritzenmultiplikation

Komplizieren: $E_6 \rightsquigarrow E_5$

- E_6 Lineare Algebra: Produktsummenformel

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

- E_5 C-Programm: Quellmodul `multiply.c`

```
1 typedef int Matrix [N][N];
2
3 void multiply(const Matrix a, const Matrix b, Matrix c) {
4     unsigned int i, j, k;
5     for (i = 0; i < N; i++) {
6         for (j = 0; j < N; j++) {
7             c[i][j] = 0;
8             for (k = 0; k < N; k++)
9                 c[i][j] += a[i][k] * b[k][j];
10        }
11    }
```

- insgesamt sechs mögliche Varianten (d.h., Schleifenanordnungen)
 - funktional gleich, nichtfunktional ggf. ungleich \hookrightarrow 1-9

Beispiel: Matritzenmultiplikation

Kompilieren: $E_5 \rightsquigarrow E_4$

E₄ Assembler-Programm: Zwischenmodul `multiply.s`

```
1 .file "multiply.c"          27    .p2align 3
2 .text                         28    .L3:
3 .p2align 4,,15                29    movl  (%ecx),%eax
4 .globl multiply                 30    addl  $400,%ecx
5 .type multiply,@function        31    imull (%edi,%edx,4),%eax
6 multiply:                      32    addl  $1,%edx
7   pushl %ebp                   33    addl  %eax,(%esi,%ebx,4)
8   movl  %esp,%ebp               34    cmpl  $100,%edx
9   pushl %edi                   35    jne   .L3
10  pushl %esi                  36    addl  $1,%ebx
11  pushl %ebx                  37    cmpl  $100,%ebx
12  subl  $4,%esp                38    jne   .L4
13  movl  16(%ebp),%esi          39    addl  $400,-16(%ebp)
14  movl  $0,-16(%ebp)           40    addl  $400,%esi
15 .L2:                           41    cmpl  $40000,-16(%ebp)
16  movl  8(%ebp),%edi           42    jne   .L2
17  xorl  %ebx,%ebx              43    addl  $4,%esp
18  addl  -16(%ebp),%edi         44    popl  %ebx
19  .p2align 4,,7                  45    popl  %esi
20  .p2align 3                     46    popl  %edi
21 .L4:                           47    popl  %ebp
22  movl  12(%ebp),%eax           48    ret
23  xorl  %edx,%edx              49    .size multiply, .-multiply
24  movl  $0,(%esi,%ebx,4)         50    .ident "GCC: (Debian 4.3.2-1.1) 4.3.2"
25  leal  (%eax,%ebx,4),%ecx          51    .section .note.GNU-stack,"",@progbits
26  .p2align 4,,7
```

You are **not** expected
to understand this!



Beispiel: Matritzenmultiplikation Assemblieren und Linken: $E_4 \rightsquigarrow E_3$

```
00000000 457f 464c 0101 0001 0000 0000 0000 0000  
00000020 0001 0003 0001 0000 0000 0000 0000 0000  
00000040 0114 0000 0000 0000 0034 0000 0000 0028  
00000060 0009 0006 0000 0000 0000 0000 0000 0000  
00001000 8955 57e5 5356 ec83 8b04 1075 45c7 00f0  
00001200 0000 8b00 087d db31 7d03 90f0 748d 0026  
00001400 458b 310c c7d2 9e04 0000 0000 0c8d 9098  
00001600 018b c181 0190 0000 af0f 9704 c283 0101  
00002000 9e04 fa83 7564 83e9 01c3 fb83 7564 81d1  
00002200 f045 0190 0000 c681 0190 0000 7d81 40f0  
00002400 009c 7500 83ae 04c4 5e5b 5d5f 00c3 0000  
00002600 4700 4343 203a 4428 6265 6169 206e 2e34  
00003000 2e33 2d32 2e31 2931 3420 332e 322e 0000  
00003200 732e 6d79 6174 0062 732e 7274 6174 0062  
00003400 732e 7368 7274 6174 0062 742e 7865 0074  
00003600 642e 7461 0061 622e 7373 2e00 6f63 6d6d  
00004000 6e65 0074 6e2e 746f 2e65 4e47 2d55 7473  
00004200 6361 006b 0000 0000 0000 0000 0000 0000  
00004400 0000 0000 0000 0000 0000 0000 0000 0000  
00004600 0000 0000 0000 0000 0000 0000 001b 0000  
00005000 0001 0000 0006 0000 0000 0000 0040 0000  
00005200 006d 0000 0000 0000 0000 0010 0000 0000  
00005400 0000 0000 0021 0000 0001 0000 0003 0000  
00005600 0000 0000 00b0 0000 0000 0000 0000 0000  
00006000 0000 0004 0000 0000 0000 0027 0000
```

```
0000620 0008 0000 0003 0000 0000 0000 00b0 0000  
0000640 0000 0000 0000 0000 0000 0000 0004 0000  
0000660 0000 0000 0000 002c 0000 0001 0000 0000  
0000700 0000 0000 00b0 0000 001f 0000 0000 0000  
0000720 0000 0000 0001 0000 0000 0000 0035 0000  
0000740 0001 0000 0000 0000 0000 0000 00cf 0000  
0000760 0000 0000 0000 0000 0000 0000 0001 0000  
0001000 0000 0000 0011 0000 0003 0000 0000 0000  
0001020 0000 0000 00cf 0000 0045 0000 0000 0000  
0001040 0000 0000 0001 0000 0000 0000 0001 0000  
0001060 0002 0000 0000 0000 0000 0000 027c 0000  
0001100 0080 0000 0008 0000 0007 0000 0004 0000  
0001120 0010 0000 0009 0000 0003 0000 0000 0000  
0001140 0000 0000 02fc 0000 0015 0000 0000 0000  
0001160 0000 0000 0001 0000 0000 0000 0000 0000  
0001200 0000 0000 0000 0000 0000 0000 0001 0000  
0001220 0000 0000 0000 0000 0004 fff1 0000 0000  
0001240 0000 0000 0000 0000 0003 0001 0000 0000  
0001260 0000 0000 0000 0000 0003 0002 0000 0000  
0001300 0000 0000 0000 0000 0003 0003 0000 0000  
0001320 0000 0000 0000 0000 0003 0005 0000 0000  
0001340 0000 0000 0000 0000 0003 0004 000c 0000  
0001360 0000 0000 006d 0000 0012 0001 6d00 6c75  
0001400 6974 6c70 2e79 0063 756d 746c 7069 796c  
0001420 0000
```

- E_3 Resultierendes Maschinenprogramm: Ladbares Modul `a.out`
 - Hexadezimalkode **ausführbar** – jedoch alleine kein ausführbares Programm!



Beispiel: Matritzenmultiplikation

Laden und Auflösen: $E_3 \rightsquigarrow E_{2/1}$

010101011000100111100101010111001010100010100111000001111011000000100
1000101111100101000100001100011101000101111000000000000000000000000000000000
00000000100010111101101000010000011000111011011000000110111110111110000
100100001000110101110100001001100000000010001011010001010000110000110001
1101001011000111000001001001110000000000000000000000000000000000000010001101
0000110010011000**10010000**100010110000000110000001110000011001000000000000001
00
00
00
00
00
110000011000000011000001111110110110010001110101110100011000000101000101
11110000100100000000000100
00
01110101101011101000001111000101000001000101101101011110010111110101111010111101
110000011

hervorgehobene Bitfolgen repräsentieren (durch `.p2align` aufgefüllte Nulloperationen)

- $E_{2/1}$ **Auflösung** des ausführbaren Hexadezimalkodes zur **Bitfolge**
 - Befehlsverarbeitung geschieht bitweise, *nicht* byte- oder wortweise
 - Die für einen Digitalrechner (hier: x86) letztendlich benötigte Form

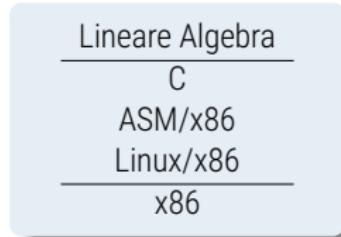
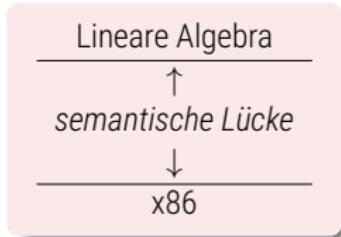


Beispiel: Zusammenfassung

Abstraktionshierarchie von

Sprachsystemen

- E_6 Modellsprache (Lineare Algebra) \rightsquigarrow 1 Produktsummenformel
 - E_5 Programmiersprache (C) \rightsquigarrow 5 Komplexschritte
 - E_4 Assembliersprache (ASM/x86) \rightsquigarrow 35 Elementarschritte
 - E_3 Maschinensprache (Linux/x86) \rightsquigarrow 109 Bytes Programmtext
 - E_2 Maschinensprache (x86) \rightsquigarrow 872 Bits Instruktionssstrom
- Eine einzelne komplexe und überwältigende Aufgabe
in mehrere kleine und handhabbare unterteilen

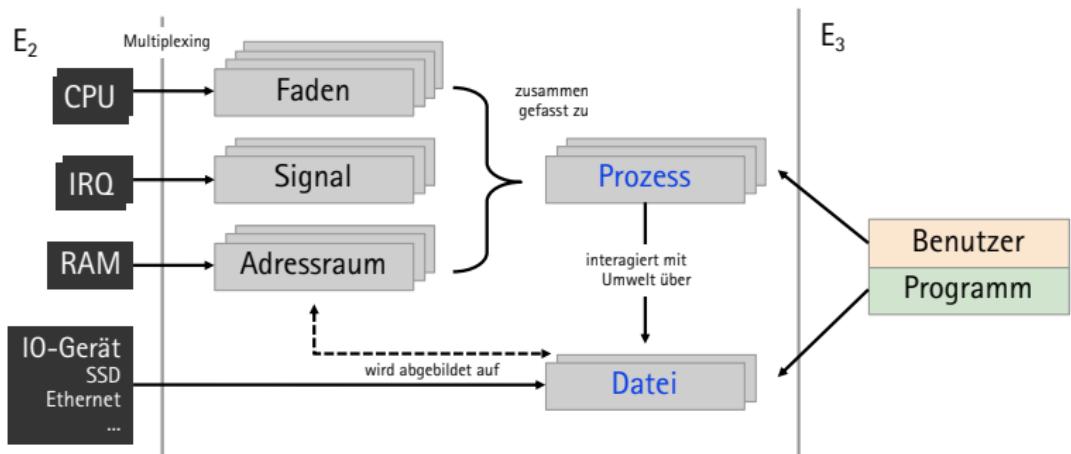


3.3 Virtualisierung der Hardwareressourcen

- Das BS bietet dem Anwender (Programm) eine erweiterte Maschinenschnittstelle (**teilinterpretierende virtuelle Maschine, E_3**).
- Der Befehlssatz der HW (E_2) wird (weitestgehend) „durchgereicht“ und um zusätzliche **Systemaufrufe** (system calls) erweitert.
- Die Ressourcen der HW (Prozessor, Speicher, IO-Geräte, ...) werden durch **Multiplexing virtualisiert** (Mehrprogrammbetrieb).
- Virtuelle **Hardwareressourcen** werden durch **Schutzmechanismen** (räumlich und zeitlich) voneinander **isoliert** (Mehrbenutzerbetrieb).
- Virtuelle Hardwareressourcen werden repräsentiert durch die grundlegenden Konzepte (**Abstraktionen**) eines Betriebssystems.

Multiplexing und Isolation durch Prozesse und Dateien

- Virtuelle Hardwareressourcen werden repräsentiert durch die grundlegenden Konzepte (**Abstraktionen**) eines Betriebssystems. **In UNIX sind das:**
 - Prozess** (Adressraum + Fäden) \mapsto virtueller „Computer“
 - Datei** \mapsto virtuelles „Peripheriegerät“
(auch persistenter Speicher \mapsto „echte Datei“)



“ In UNIX ist jedes Objekt entweder ein Prozess oder eine Datei! ”

Informatik Folklore

Multiplexing und Isolation durch Prozesse und Dateien

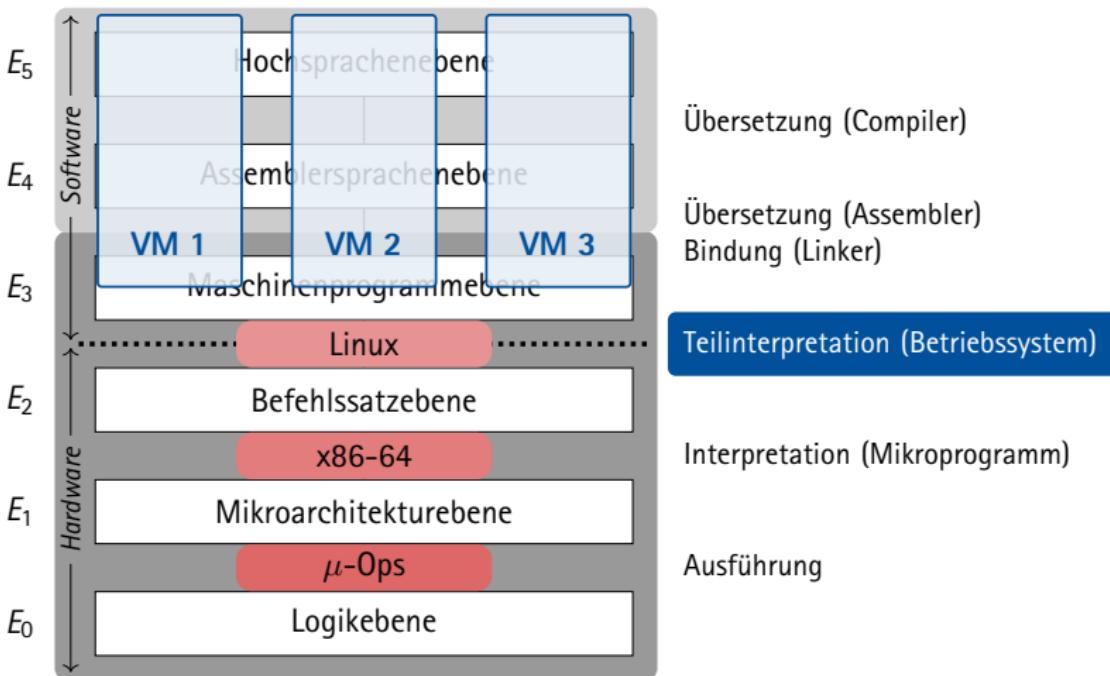
- Virtuelle **Hardwareressourcen** werden durch **Schutzmechanismen** (räumlich und zeitlich) voneinander **isoliert** (Mehrbenutzerbetrieb).
 - **Horizontale Isolation** **Prozesse** mit jeweils eigenem Adressraum + Fäden
 - **Vertikale Isolation** Aufteilung in **Benutzerobermodus** und **Systemmodus**
 - **Benutzerobermodus** CPU-Modus in dem einige E_2 -Instruktionen *nicht* durchgereicht, sondern durch das BS interpretiert werden
 - Manipulation von Adressräumen \mapsto räumliche Isolation, Speicher
 - Manipulation der Unterbrechungsbehandlung \mapsto zeitliche Isolation, CPU
 - **Systemmodus** Alle E_2 -Instruktionen stehen zur Verfügung
- Anwenderprozesse laufen immer im Benutzerobermodus
- Betriebssystemdienste laufen (teilweise) im Systemmodus

auch *nichtprivilegierter Modus*
oder *Ring 3* (Intel) genannt.

auch *Supervisormodus*,
privilegierter Modus
oder *Ring 0* (Intel) genannt.

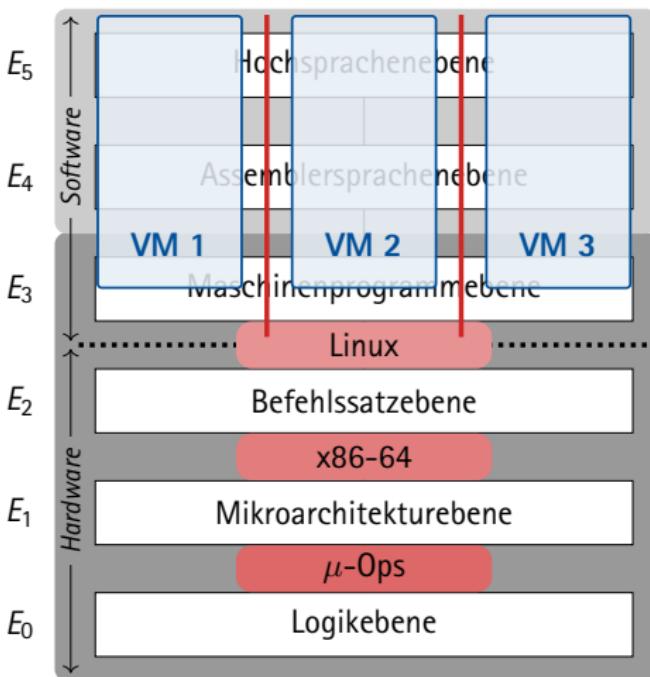
Multiplexing und Isolation durch Prozesse

Betriebssystem \mapsto Bereitsteller virtueller „Computer“ (Maschinen)



Multiplexing und Isolation durch Prozesse

Betriebssystem \mapsto Bereitsteller virtueller „Computer“ (Maschinen)



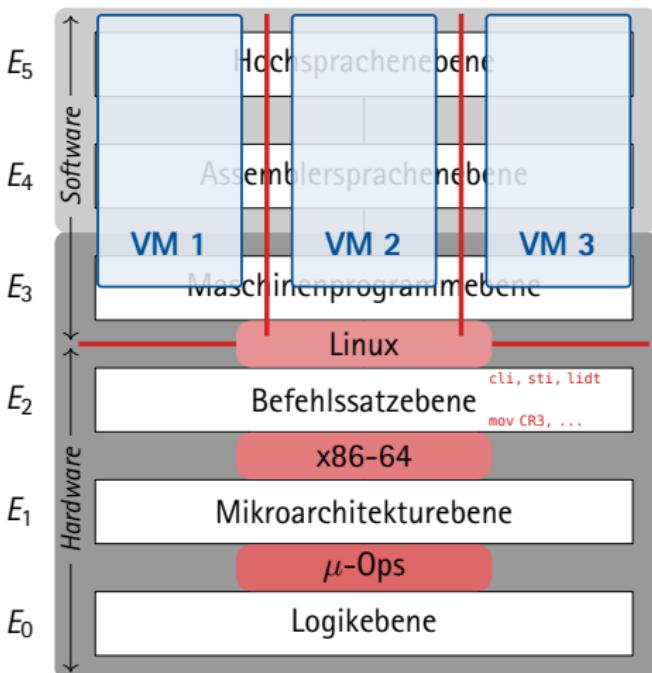
Horizontale Isolation
(zeitlich/räumlich)
unabhängiger virtueller
Maschinen (Prozesse) durch
IRQs, MPU/MMU (auf E_2)

Teilinterpretation (Betriebssystem)



Multiplexing und Isolation durch Prozesse

Betriebssystem \mapsto Bereitsteller virtueller „Computer“ (Maschinen)



Teilinterpretation (Betriebssystem)

Vertikale Isolation
(Benutzer-/Systemmodus)
durch Abschirmung der
E₂-Instruktionen für die
horizontale Isolation

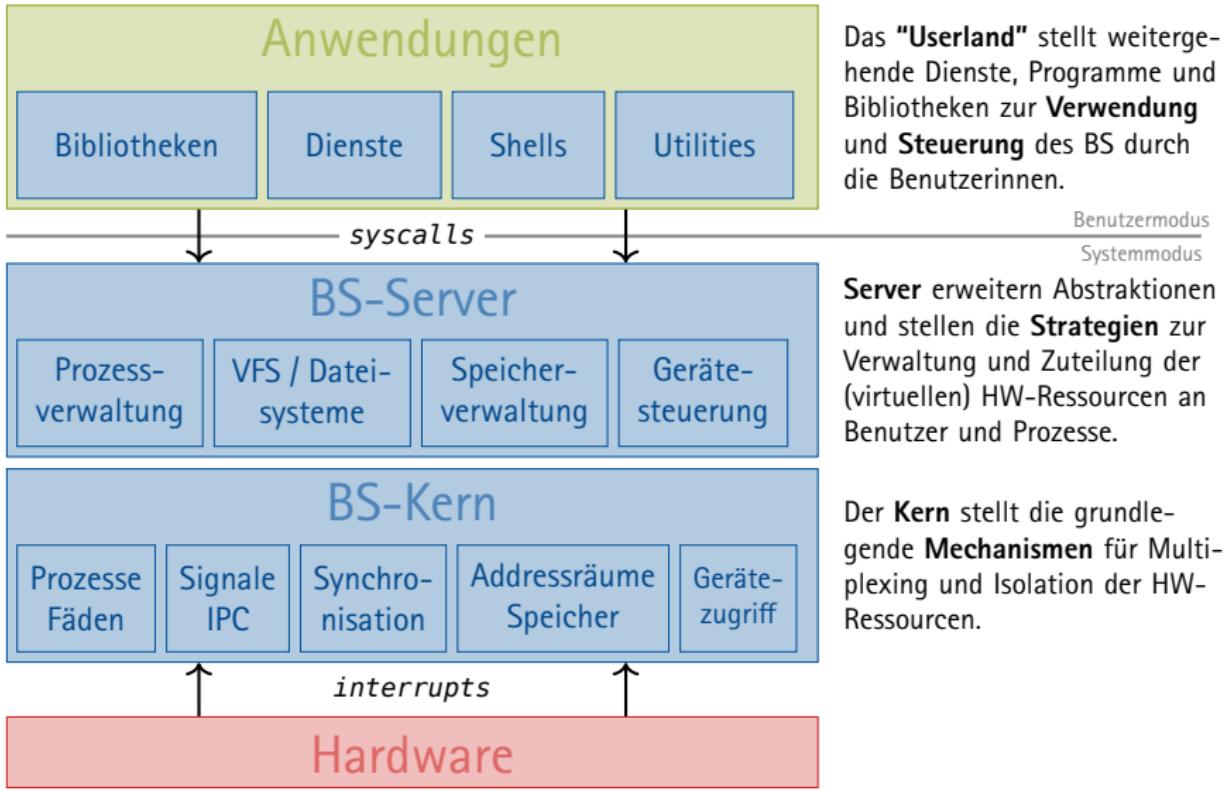


3.4 Aufbau und Architektur eines Betriebssystems

- Multiplexing und Isolation sind die **Kernaufgaben** eines Betriebssystems
 - Sie sind entsprechend im Betriebssystem **kern** (Kernel) verankert.
 - Der Kernel läuft immer im Systemmodus.
 - benötigt die privilegierten Befehle, um Isolation durchzusetzen
- **Verwaltung** der virtuellen HW-Ressourcen durch ergänzende **Server**
 - Prozessverwaltung, Speicherverwaltung
 - Dateisystem, Gerätesteuerung, Netzwerkzugriff
 - Im Systemmodus oder im Benutzermodus angesiedelt.
 - Privilegierte Befehle werden dafür i.d.R. nicht benötigt
- Weitere Dienste und Systemprogramme (**Utilities / Userland**) werden als Prozesse im Benutzermodus gestartet
 - Shell / Benutzeroberfläche (Systemzugang), wie **bash** oder Windows-Explorer
 - Hintergrunddienste (UNIX: Deamons), wie der Anmeldeprozess
 - Systemprogramme / Editoren (Systemsteuerung), wie **top**, **vim**, ...
 - Gemeinsame Bibliotheken (shared libraries), wie z.B. **libc**, **libgcrypt**, ...

Aufgabenteilung im Betriebssystem

Kern-Server-Userland

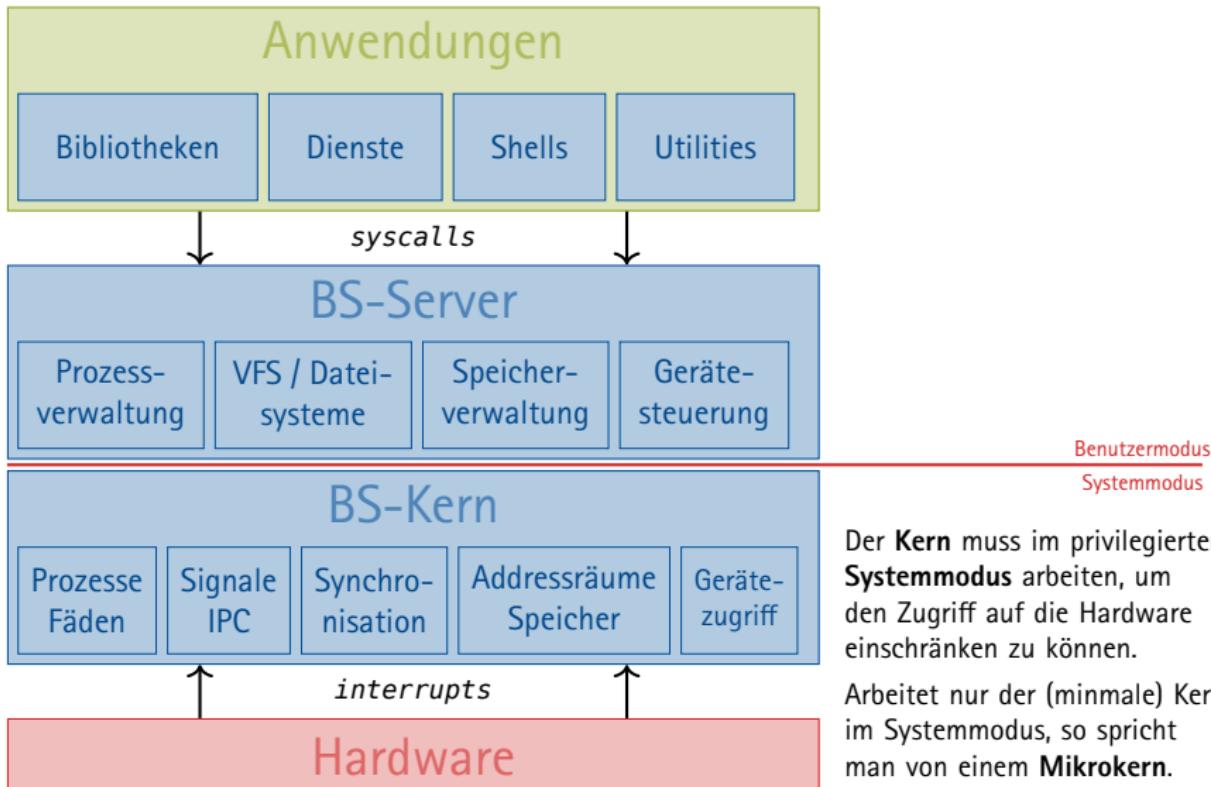


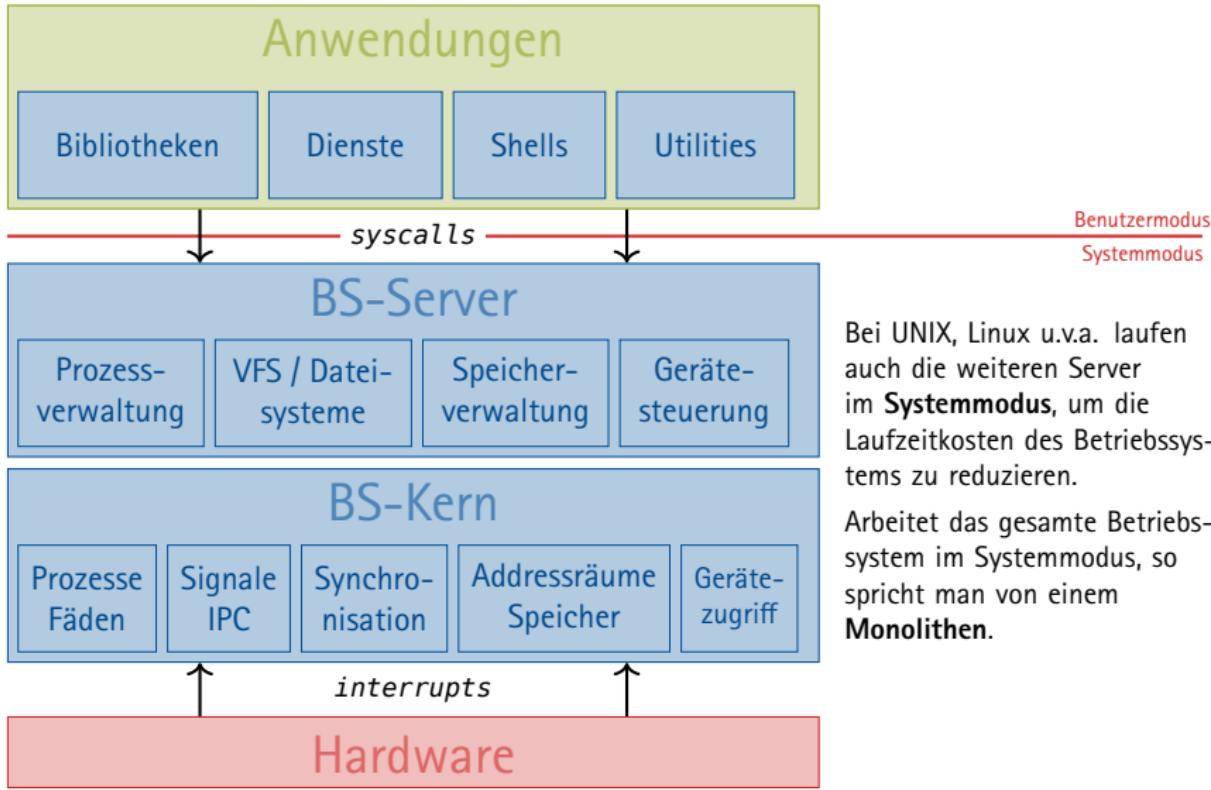
- Die grundlegenden Organisationsprinzipien bei der Aufteilung der Betriebssystemfunktionen bestimmen die **Betriebssystemarchitektur**.
 - Unser Bild der Aufgabenteilung entspricht einer **monolithischen Architektur**.
 - Andere Varianten, wie zum Beispiel die **Mikrokernarchitektur**, sind möglich.
 - Wesentliches Unterscheidungsmerkmal ist die Granularität der Schutzdomänen und Privilegierten *innerhalb* des Betriebssystems
- Die Architektur beeinflusst die Auslegung im Betriebssystem, nicht jedoch die Funktionalität der Systemfunktionen.
 - **funktional** transparent für Anwendung und Anwender
 - Unterschiede zeigen sich in **nichtfunktionalen** Eigenschaften, wie Robustheit, Geschwindigkeit, Angriffssicherheit oder Speicherbedarf

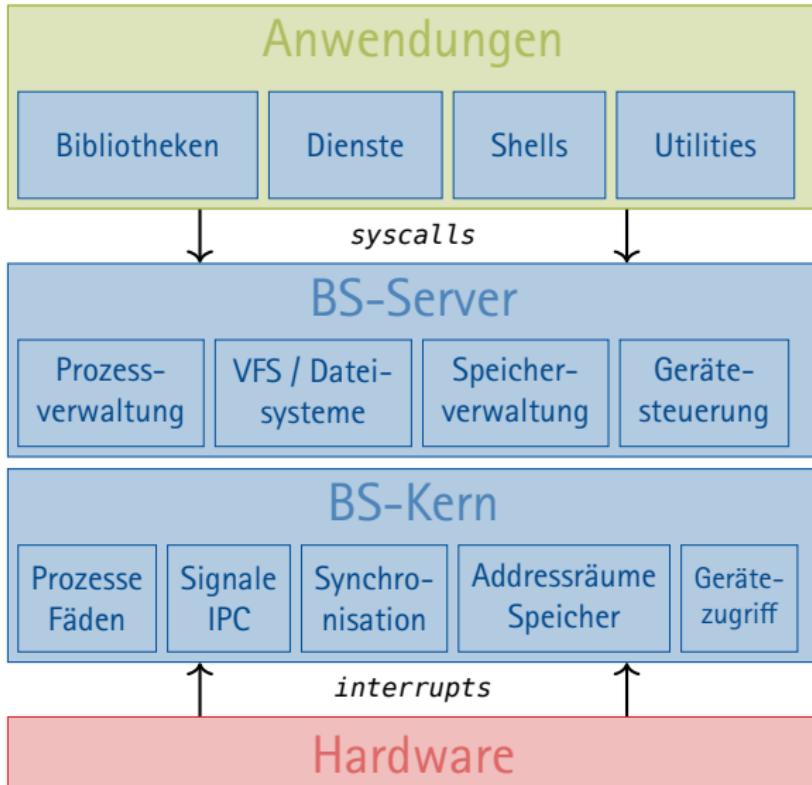
Die drei Prinzipien von Architektur

“ *Schönheit, Stabilität, Nützlichkeit — Venustas, Firmitas, Utilitas.* ”

Pollio 1996 (Original 27 v. Chr.): *De Architectura Libris Decem* [26]







Läuft das gesamte Rechensystem **einschließlich der Anwendungen** im **Systemmodus**, können Speicher- und Laufzeitkosten der Betriebssystemverwendung deutlich reduziert werden. Allerdings kann das Betriebssystem so keinerlei Schutz mehr sicher stellen.

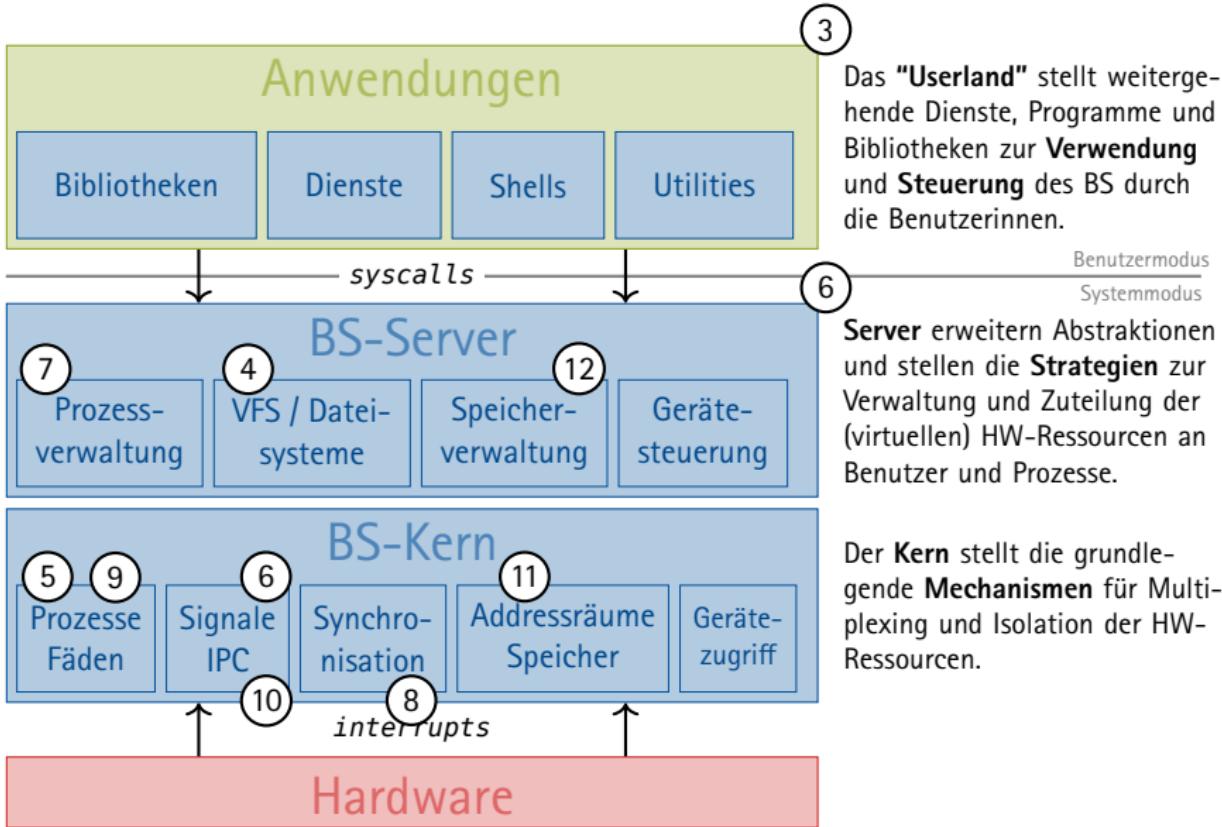
Diese Konfiguration findet sich häufig in eingebetteten Spezialzwecksystemen, wo die Anwendungen unter Kontrolle sind.



3.5 Betriebssystemkonzepte im Überblick



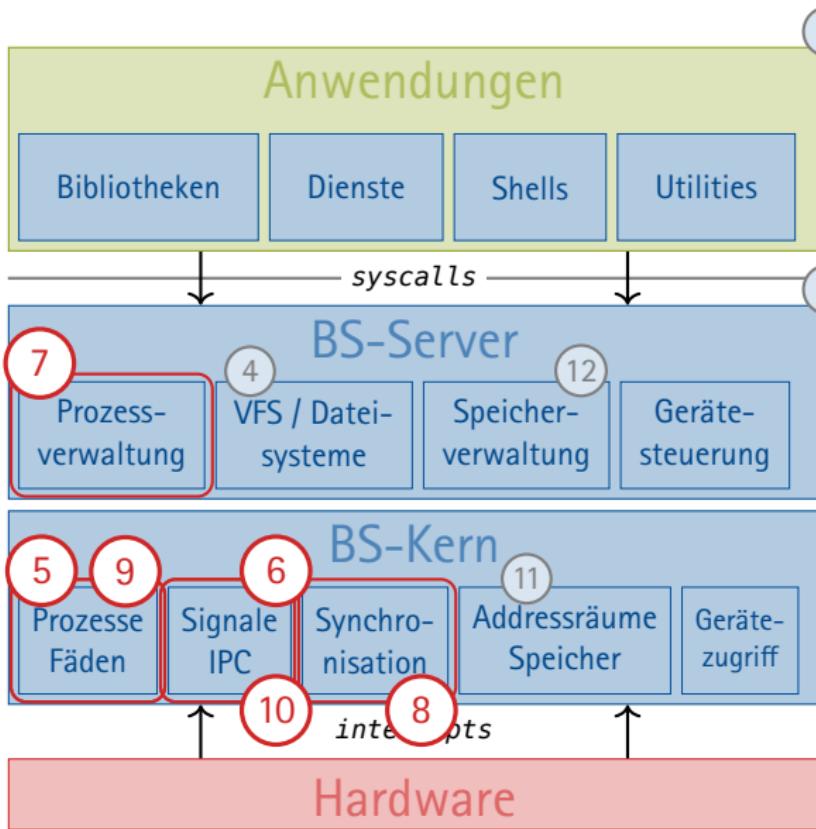
Themenbereiche in GBS





Themenbereiche in GBS

Prozesse und ihre Interaktionen



Das "Userland" stellt weitergehende Dienste, Programme und Bibliotheken zur **Verwendung** und **Steuerung** des BS durch die Benutzerinnen.

Benutzermodus
Systemmodus

Server erweitern Abstraktionen und stellen die **Strategien** zur Verwaltung und Zuteilung der (virtuellen) HW-Ressourcen an Benutzer und Prozesse.

Der Kern stellt die grundlegende **Mechanismen** für Multiplexing und Isolation der HW-Ressourcen.



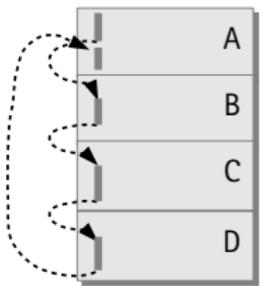
Multiplexing und Isolation durch Prozesse

→ [5]

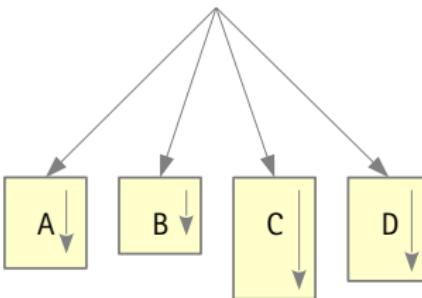
Programm \mapsto Eine Sequenz von Anweisungen für einen Prozessor

Prozess \mapsto Ein Programm in Ausführung auf einem Prozessor

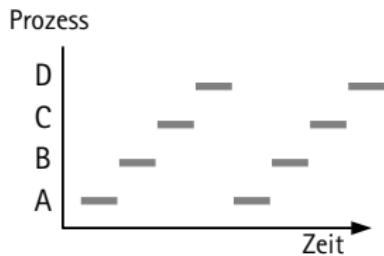
Mehrprogrammbetrieb



Nebenläufige Prozesse



Multiplexing der CPU



Technische Sicht

- 1 Instruktionszeiger
- 4 Prozessobjekte
- Kontextwechsel

Konzeptionelle Sicht

- 4 unabhängige Programmausführungen

Realzeit-Sicht

- Zu jedem Zeitpunkt ist nur ein Prozess aktiv
- Pseudoparallelität

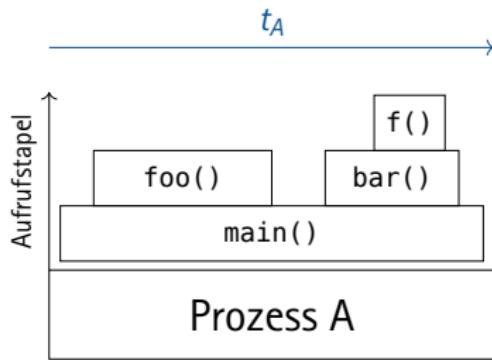


Multiplexing und Isolation durch Prozesse

→ [5]

Konzeptionelle Sicht:

Nebenläufige Prozesse
(logische Zeit auf virtueller CPU)



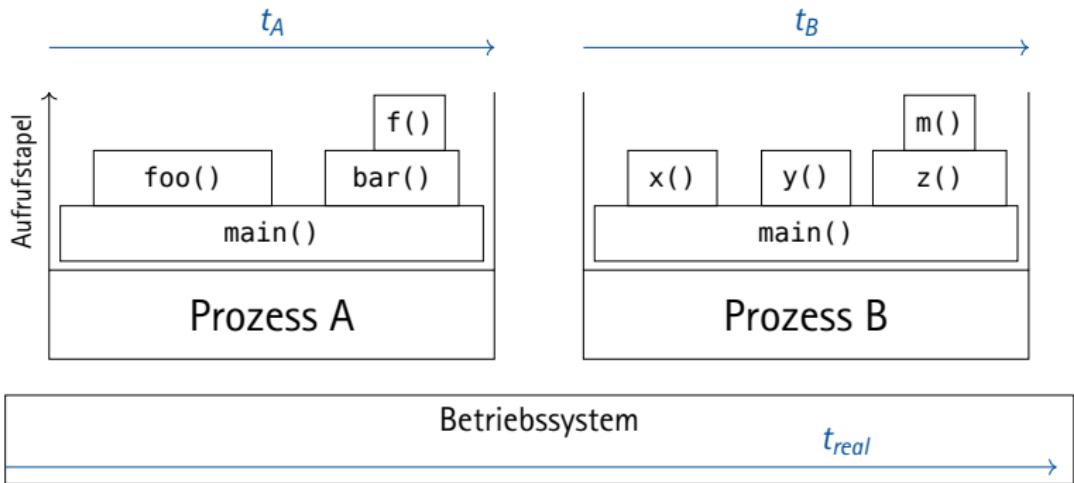


Multiplexing und Isolation durch Prozesse

→ [5]

Konzeptionelle Sicht:

Nebenläufige Prozesse
(logische Zeit auf virtueller CPU)



Realzeit-Sicht:

Betriebssystem multiplext CPU
(physische Zeit auf realer CPU)

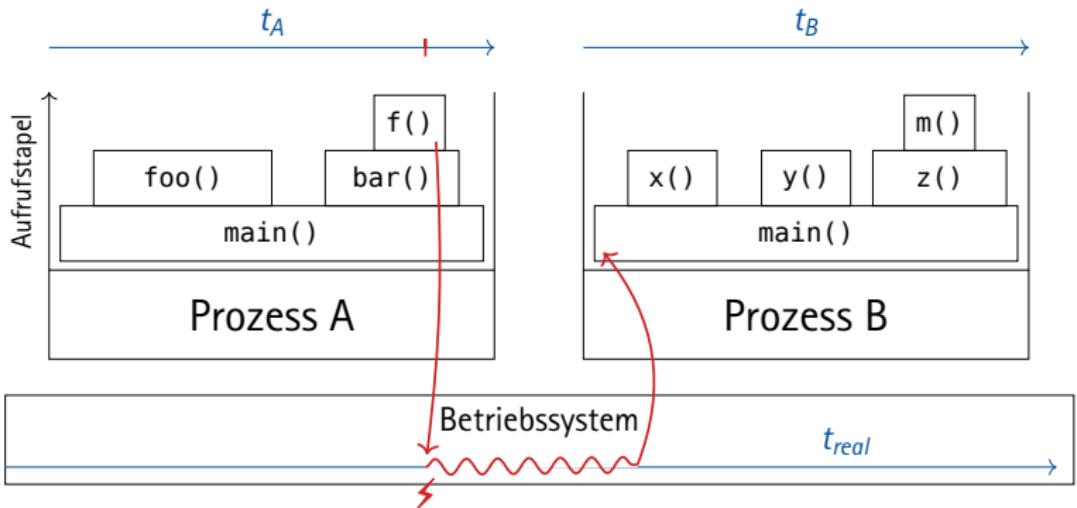


Multiplexing und Isolation durch Prozesse

→ [5]

Konzeptionelle Sicht:

Nebenläufige Prozesse
(logische Zeit auf virtueller CPU)



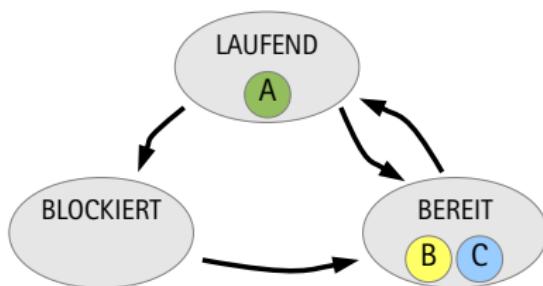
Realzeit-Sicht:

Betriebssystem multiplext CPU
(physische Zeit auf realer CPU)

Multiplexing und Isolation durch Prozesse

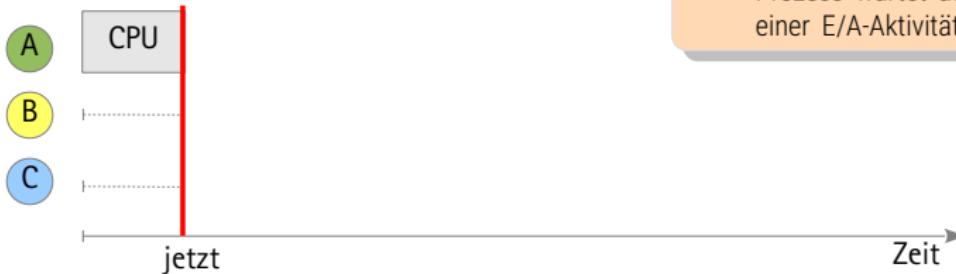
→ [5]

CPU-Virtualisierung: Prozessverhalten und -zustände



Prozesszustände (Fadenzustände)

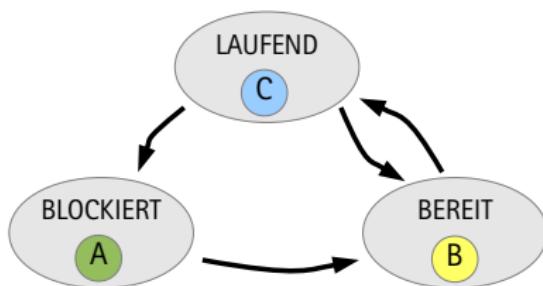
- LAUFEND
 - Prozess wird gerade ausgeführt
- BEREIT
 - Prozess kann ausgeführt werden
- BLOCKIERT
 - Prozess wartet auf Beendigung einer E/A-Aktivität



Multiplexing und Isolation durch Prozesse

→ 5

CPU-Virtualisierung: Prozessverhalten und -zustände



Kontextwechsel

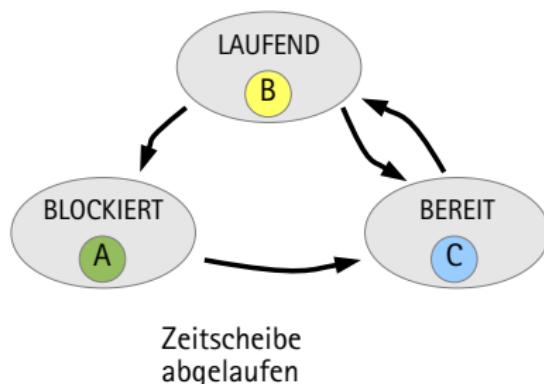
Prozess A hat einen E/A-Vorgang gestartet und ist in den Zustand BLOCKIERT übergegangen. Da A die CPU nun nicht benötigt, hat das Betriebssystem den Prozess C ausgewählt und von BEREIT in LAUFEND überführt. Es fand ein Kontextwechsel von A zu C statt.



Multiplexing und Isolation durch Prozesse

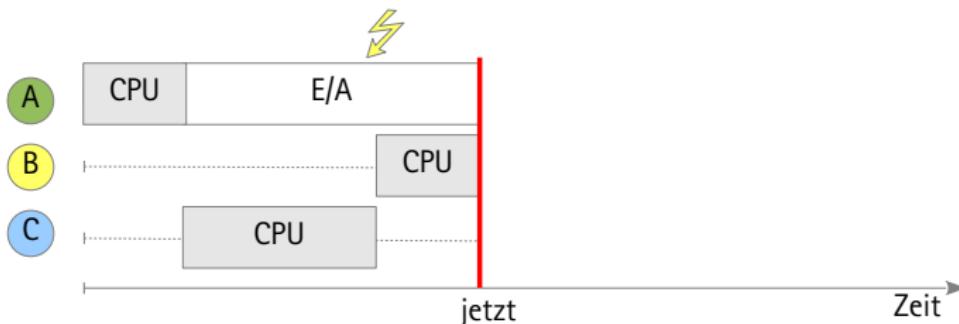
→ 5

CPU-Virtualisierung: Prozessverhalten und -zustände



Verdrängung

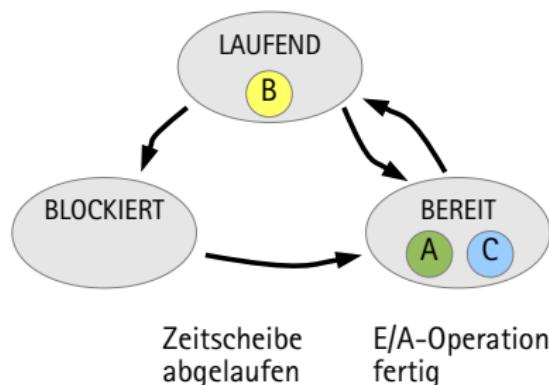
C hat die CPU zu lange „besessen“, wurde verdrängt und ist daher nun wieder im Zustand BEREIT. Damit kann jetzt endlich auch B bearbeitet werden und wird LAUFEND.



Multiplexing und Isolation durch Prozesse

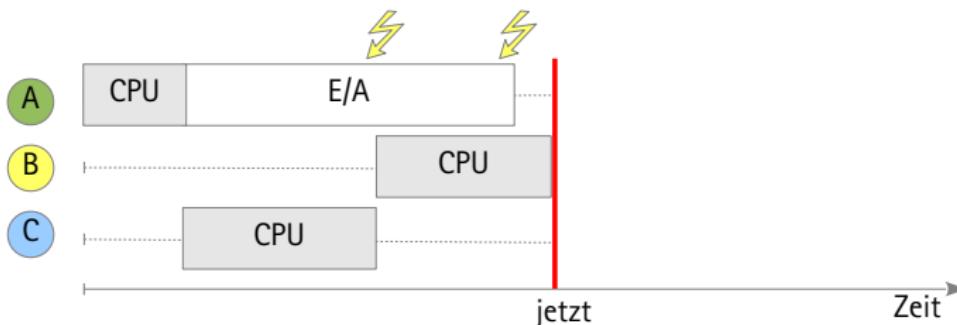
→ 5

CPU-Virtualisierung: Prozessverhalten und -zustände

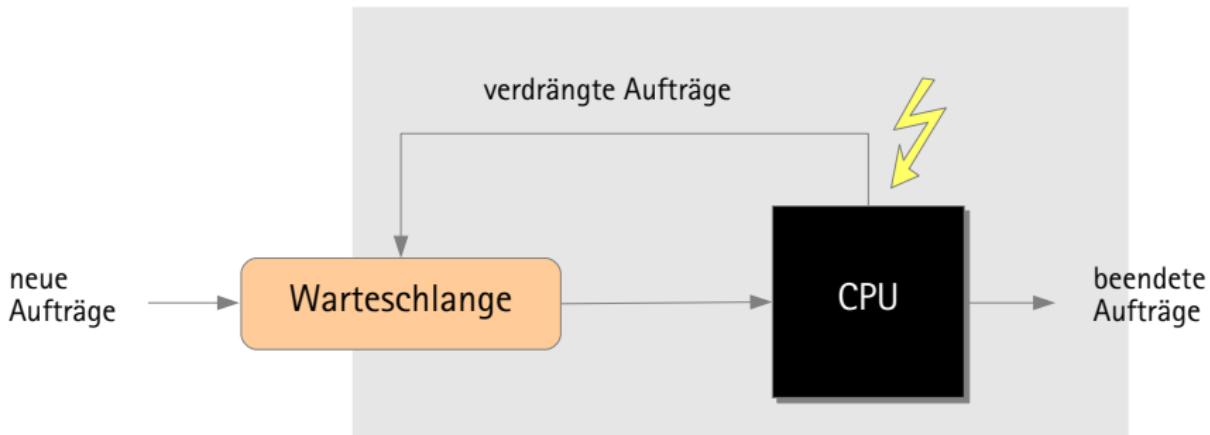


Bereitstellung

Die E/A-Operation von A ist nun abgeschlossen. Daraufhin wird A nun BEREIT und wartet auf die Zuteilung der CPU.

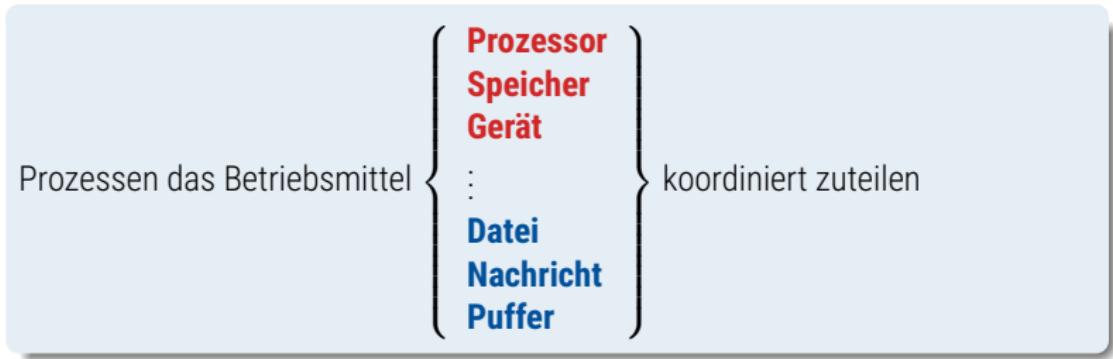


CPU-Virtualisierung: Prozessorzuteilung (*Scheduling*)

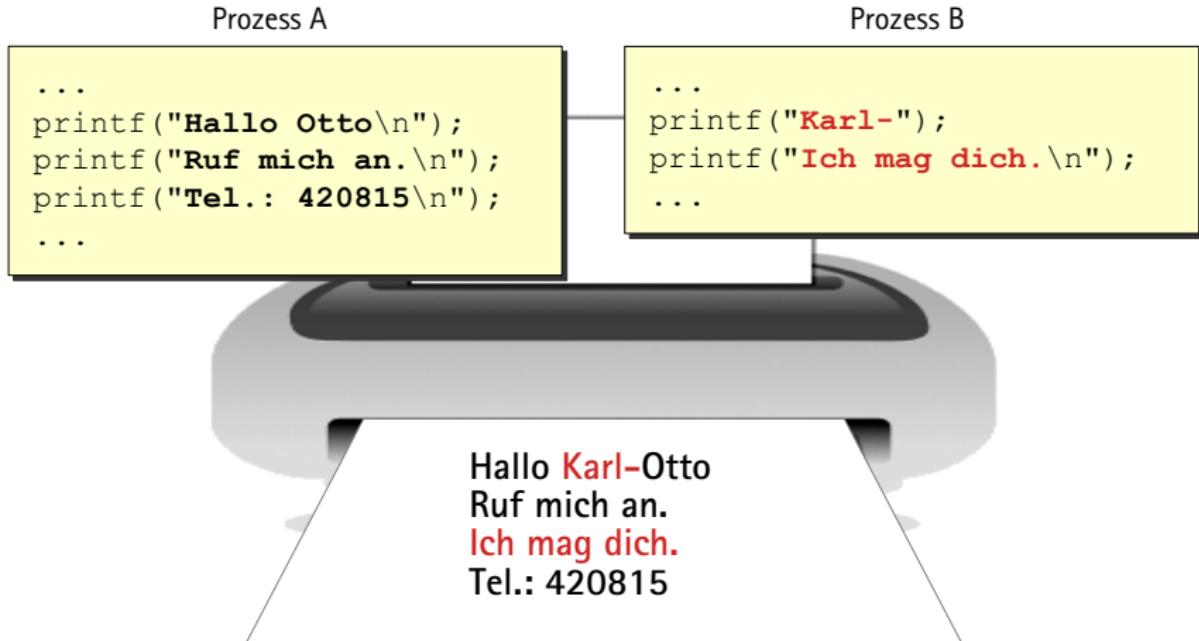


Ein **Scheduling-Algorithmus** charakterisiert sich durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse der Warteschlange zugeführt werden.

- Die **Bereitliste** (*ready list*) definiert einen **Ablaufplan** (*schedule*) zur Prozessorzuteilung, der zur Laufzeit fortgeschrieben wird.
 - Geordnet nach Ankunft, Termin, Dringlichkeit, ... → [Scheduling-Strategie](#)
- Ziele eines Planungsalgorithmus
 - benutzerorientiert? → z. B. kurze Antwortzeiten
 - systemorientiert? → z. B. hohe CPU-Auslastung
- Kein Planungsalgorithmus kann alle Bedürfnisse erfüllen!
- Allgemein bedeutet Planung innerhalb eines Betriebssystems:



Beispiel: unkoordinierter Druckerzugriff

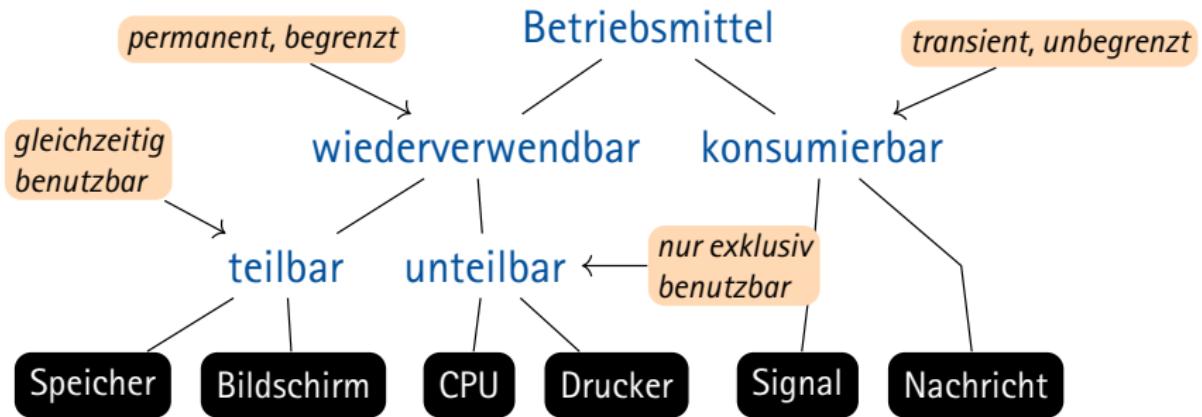


Simultanverarbeitung von Prozessen

Betriebsmittel

Allgemein: Zugriff auf Betriebsmittel \mapsto Synchronisation

\hookrightarrow 5-11



Ist die gemeinsame Benutzung (*sharing*) wiederverwendbarer Betriebsmittel oder eine logische Abhangigkeit von konsumierbaren Betriebsmitteln gegeben, so wird **Synchronisation** erforderlich \rightsquigarrow **Prozess muss gegebenenfalls „warten“!**

Beispiel: unkoordinierter Druckerzugriff \rightarrow unteilbares Betriebsmittel

- Druckerzugriff als **kritischer Abschnitt**
- Synchronisation durch **gegenseitigen Ausschluss**
- hier realisiert durch eine **Mutex**-Abstraktion des BS

Prozess A

```
...
lock(&printer_mutex);
printf("Hallo Otto\n");
printf("Ruf mich an.\n");
printf("Tel.: 420815\n");
unlock(&printer_mutex);
...
```

Prozess B

```
...
lock(&printer_mutex);
printf("Karl-");
printf("Ich mag dich.\n");
unlock(&printer_mutex);
...
```

Wenn sich einer der Prozesse A oder B zwischen **lock** und **unlock** befindet, kann der jeweils andere das **lock** nicht passieren und blockiert dort, bis der kritische Abschnitt durch **unlock** wieder freigegeben wird.

Simultanverarbeitung von Prozessen

Deadlocks

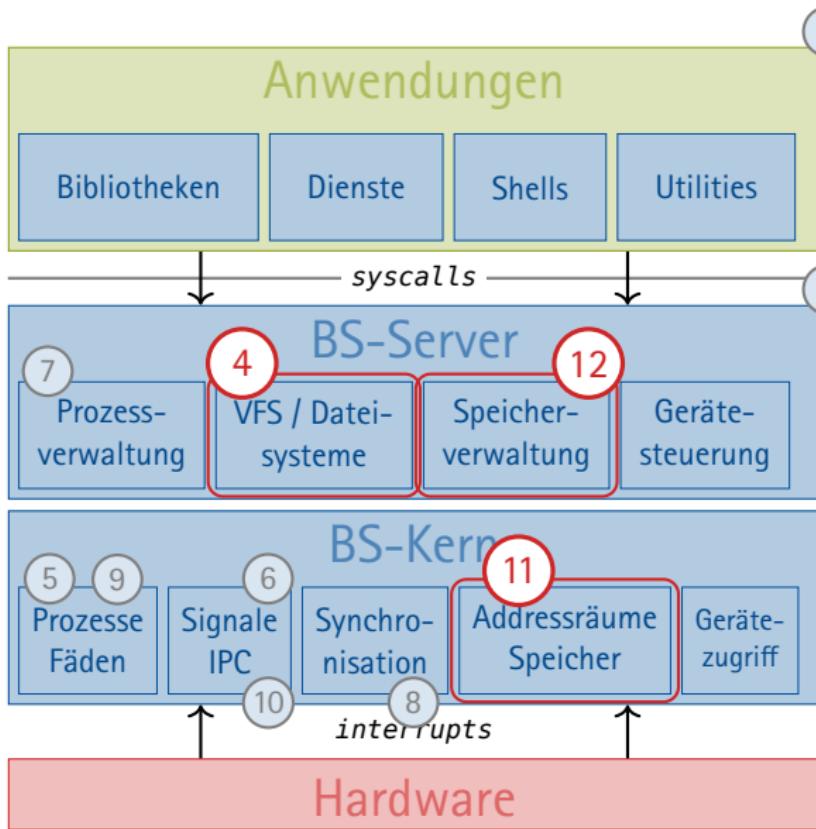
Gibt es Abhängigkeiten „im Kreis“, droht eine totale Verklemmung



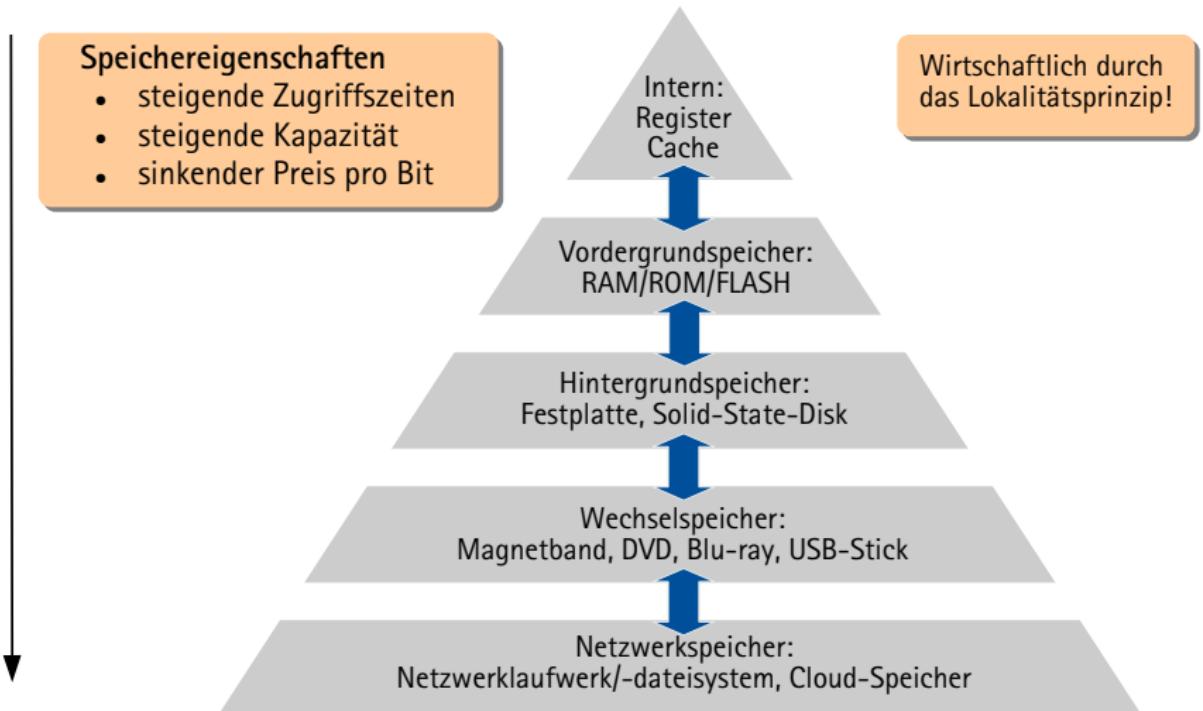


Themenbereiche in GBS

Speicher und seine Ausprägungen

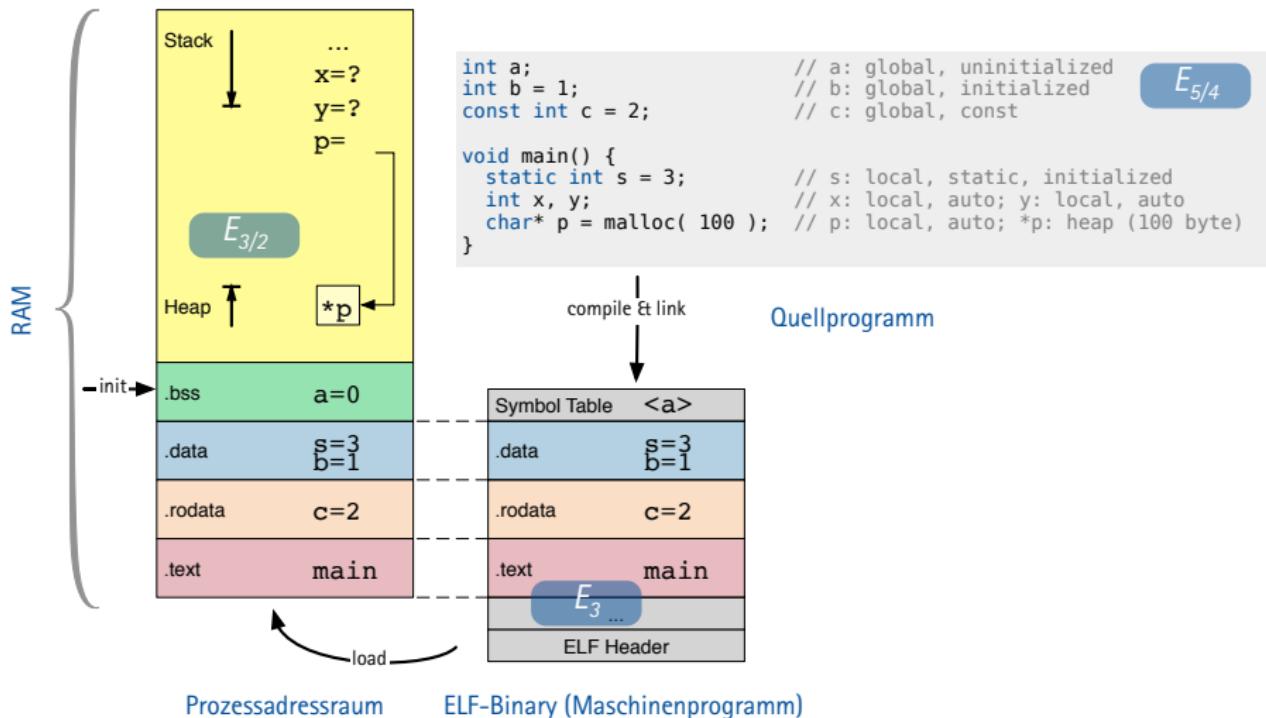


Die Speicherhierarchie eines Rechensystems





Speicherabbild: Vom Programm zum Prozess



Mehrprogrammbetrieb: Platzierung von Prozessen

■ Adressabbildung

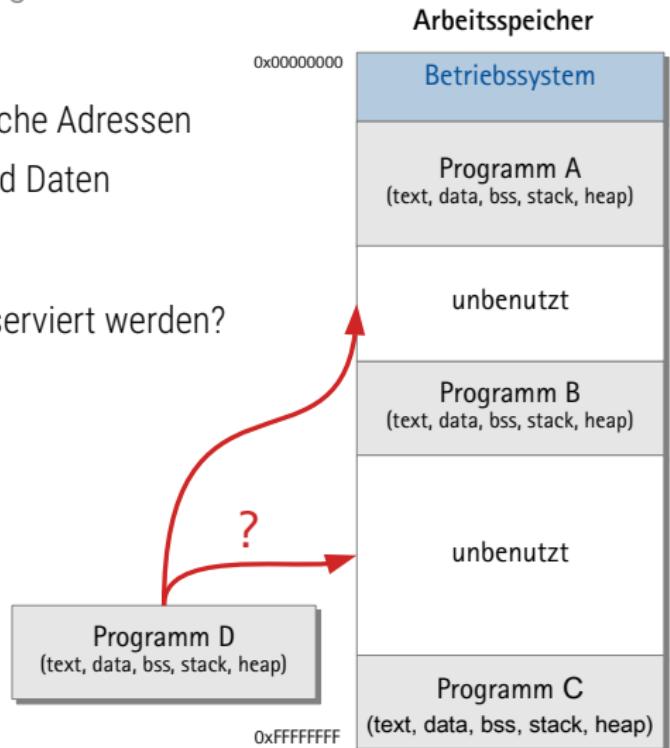
- Logische Adressen auf physikalische Adressen
- Gestattet Relokation von Code und Daten

■ Platzierungsstrategie

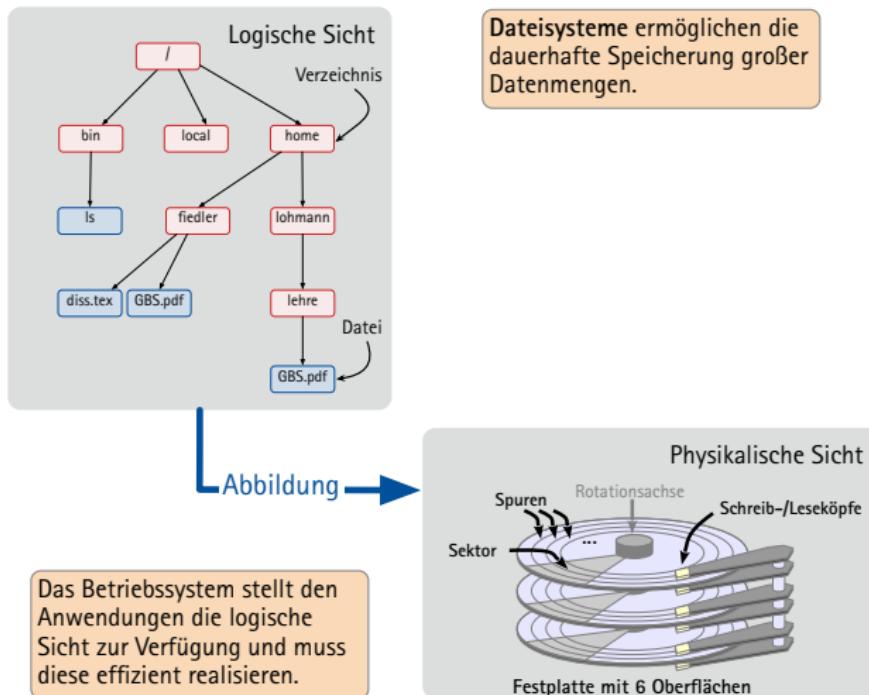
- In welcher Lücke soll Speicher reserviert werden?
- Kompaktifizierung verwenden?
- Wie minimiert man das Fragmentierungsproblem?

■ Ersetzungsstrategie

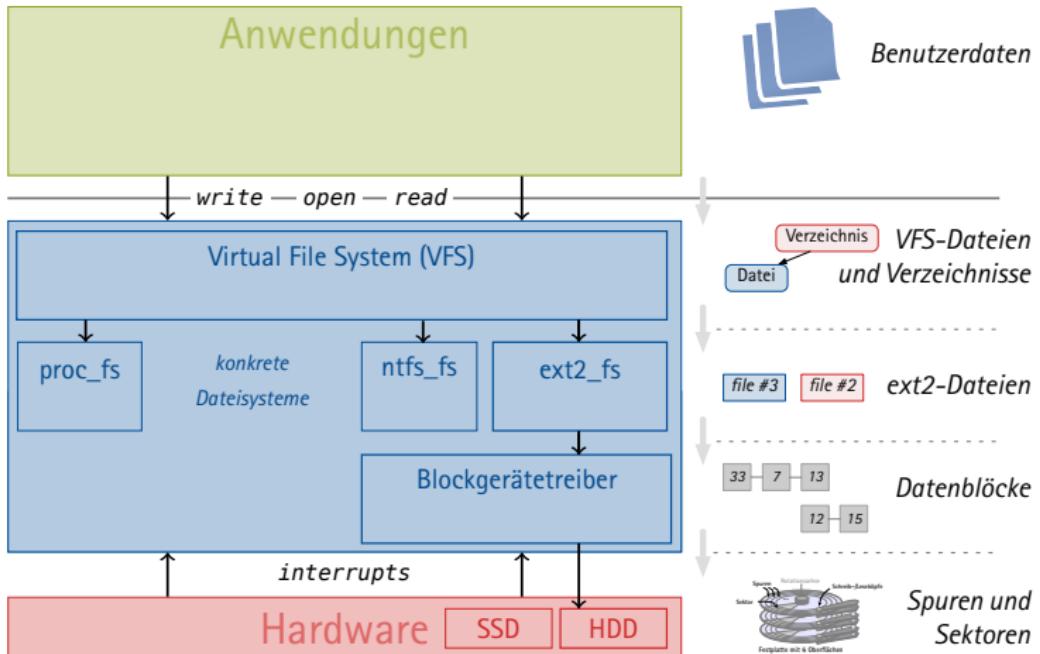
- Welcher Speicherbereich könnte sinnvoll ausgelagert werden?



Hintergrundspeicher \mapsto Dateien und Dateisysteme



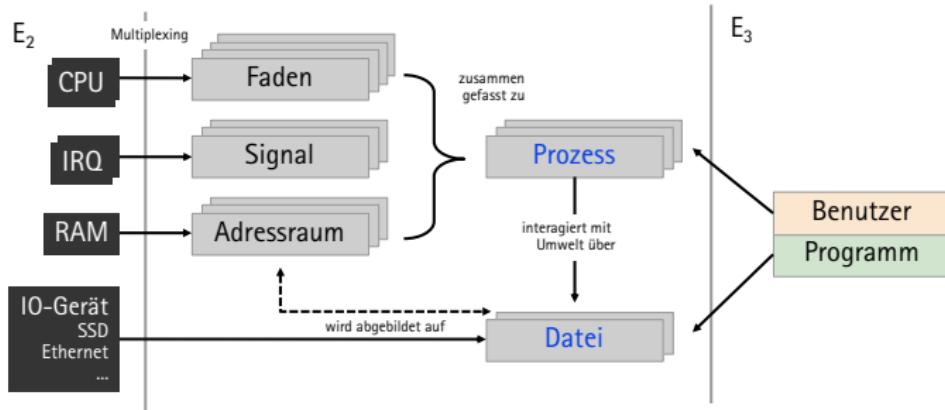
Hintergrundspeicher \mapsto Dateien und Dateisysteme



3.6 Zusammenfassung

Zusammenfassung: Grundlegende Konzepte

- **Betriebssystem** ↪ **Multiplexing** und **Isolation** der Hardware
 - Mehrprogramm- und Mehrbenutzerbetrieb
 - Verwaltung der Betriebsmitteln: CPU, Speicher, ...
 - Interne Aufgabenteilung in Kern (Mechanismen) und Server (Strategien/Dienste)
 - Planung der Zuteilung von Ressourcen an Benutzer und Prozesse
- Bereitstellung durch elementare **Abstraktionen** (Bsp. UNIX)





Teil A: Einführung

1 Einführung und Organisation

2 Exkurs: Systemnahe C-Programmierung

3 Grundlegende Konzepte

Teil B: Prozesse und Dateien

4 Dateien und Dateisysteme

5 Prozesse und Fäden

6 Unterbrechungen, Ausnahmen, Signale

7 Prozessverwaltung

Teil C: Interaktion und Kommunikation

8 Speicherbasierte Interaktionen

9 Betriebsmittelverwaltung, Synchronisation und Verklemmung

10 Interprozesskommunikation (IPC)

Teil D: Speicher und Zugriffsschutz

11 Speicherorganisation

12 Speichervirtualisierung



Technische
Universität
Braunschweig



Institute of Operating Systems
and Computer Networks
Reliable System Software



Betriebssysteme (BS)

Teil B Prozesse und Dateien

Christian Dietrich

Wintersemester 2024

4 Dateien und Dateisysteme

- 4.1 Einordnung
- 4.2 Dateien, Namen und Dateisysteme
- 4.3 Benutzersicht – Dateibaum
- 4.4 Systemsicht – Dateibaum
- 4.5 Anwendungssicht – Schnittstelle
- 4.6 Systemsicht – Offene Dateien
- 4.7 Systemsicht – Blockgeräte
- 4.8 Zusammenfassung

5 Prozesse und Fäden

6 Unterbrechungen, Ausnahmen, Signale

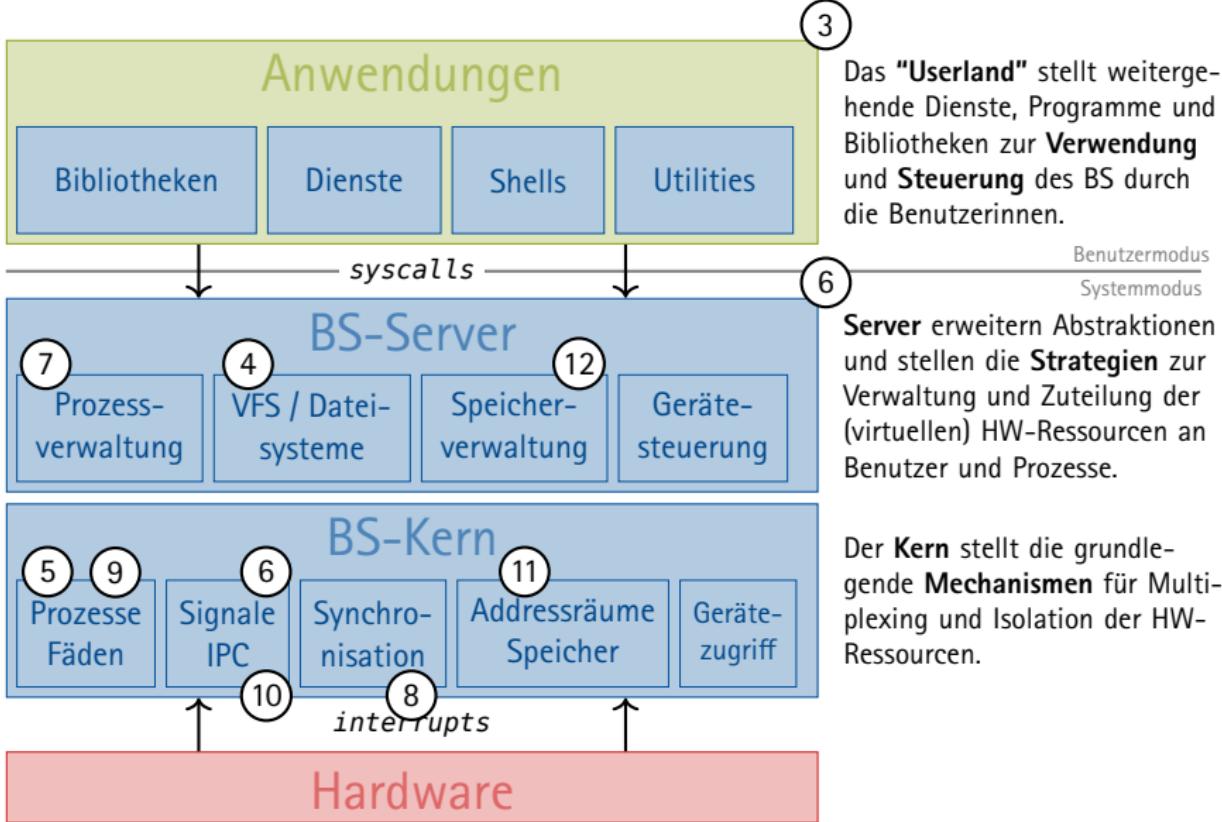
7 Prozessverwaltung



4.1 Einordnung

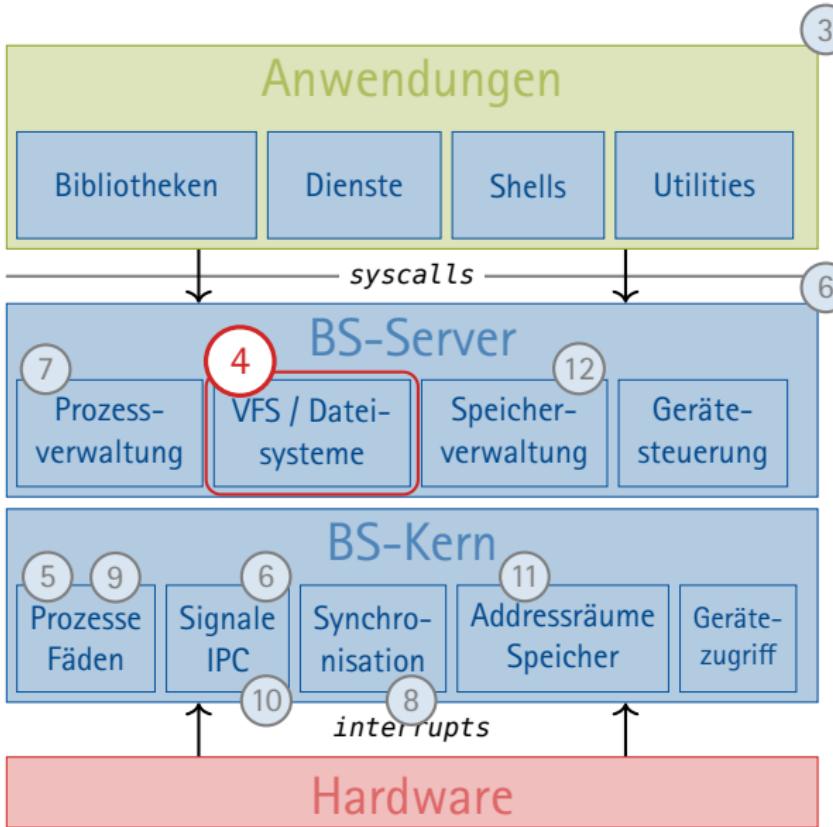


4 Dateien und Dateisysteme – Einordnung





4 Dateien und Dateisysteme – Einordnung





Worum geht es in diesem Kapitel?

■ **Kennen** der Konzepte Datei und Verzeichnis.

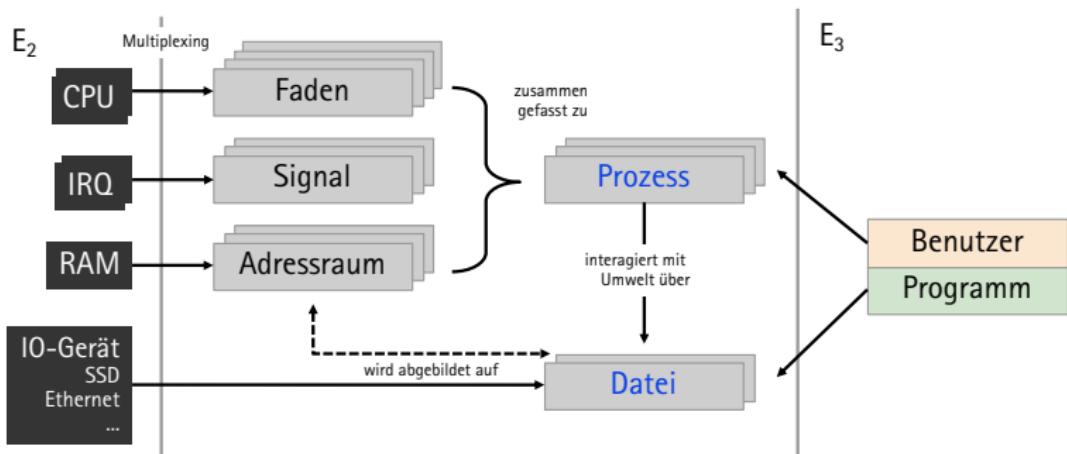
- Dateien als benannte Objekte, unechte Dateien, virtuelle Dateien
- Hierarchiebildung, Verzeichnisse, hard-/softlinks, Attribute
- Elementaroperationen, Dateien unter UNIX

■ **Verstehen** der Abbildung im und durch das Betriebssystem

- Dateisysteme, virtuelles Dateisystem (VFS), Dateideskriptoren
- Verzeichnisse \mapsto Dateien, Indexknoten
- Abbildung auf Blockgeräte, Freispeicherverwaltung, Blockallokation



- Virtuelle Hardwareressourcen werden repräsentiert durch die grundlegenden Konzepte (**Abstraktionen**) eines Betriebssystems. **In UNIX sind das:**
 - **Prozess** (Adressraum + Fäden) ↪ virtueller „Computer“
 - **Datei** ↪ virtuelles „Peripheriegerät“
(auch persistenter Speicher ↪ „echte Datei“)



„In UNIX ist jedes Objekt entweder ein Prozess oder eine Datei!“

4.2 Dateien, Namen und Dateisysteme

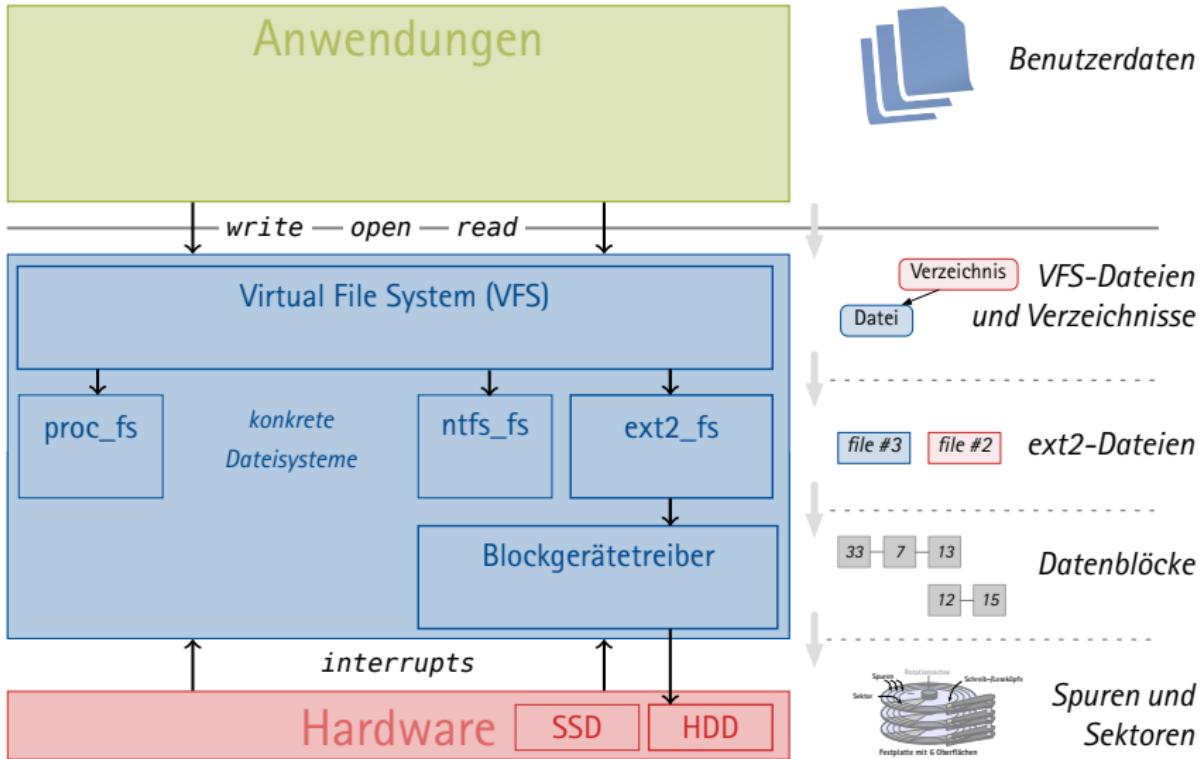


Was ist eine Datei?

- **Datei** (*file*): Kurzform von **Datenkartei** (EDV)
 - Bestand von sachlich **zusammenhängenden**, un/strukturierten **Daten**
 - in der Regel **dauerhaft** (→ persistent) im (Hintergrund-)Speicher vorrätig
 - wird durch **Prozesse** bei der Ausführung von Programmen **verarbeitet**
- Der Dateibegriff der EDV unterscheidet nur **echte Dateien**
 - **nicht ausführbare Datei**: Daten zur Verarbeitung durch ein Programm
 - **ausführbare Datei**: Instruktionen zur Ausführung durch einen Prozess
- Betriebssysteme verwenden zusätzlich **Pseudodateien** („unechte Dateien“)
 - **Gerätedatei**: repräsentiert ein physikalisches oder virtuelles Gerät (z.B. Festplatte, Drucker)
 - **procfs-Datei**: repräsentiert eine Zustandsvariable des BS-Kerns (z.B. Anzahl der Prozesse)
 - **Kommunikationskanal**: bietet Kommunikation zwischen Prozessen (IPC) (z.B. Pipe, Socket, Standardausgabe `stdout`)

- **Dateisystem** (*file system*): Prinzip zur **strukturierte Ablage** von Informationen auf einem **Datenträger**
 - Abbildung von **Dateien** und **Verzeichnissen** auf das Format des Datenträgers.
 - Bereitstellung von Metainformationen zur Einbindung durch ein Betriebssystem.
 - **Virtuelles Dateisystem:** Abstraktionsschicht des Betriebssystems für die Integration/Verwendung von Dateisystemen.
 - Virtualisierte Schnittstelle für den Zugriff auf konkrete Dateisysteme
 - Einbindung von Datenträgern durch **Montieren** (*mounten*)
 - Einbindung von **Pseudodateien** in den **Dateibaum**.
 - Überwachung und ggfs. Protokollierung des Zugriffs auf Dateien.
- Anwender und Anwendungen benutzen in der Regel nur das VFS.

4 Dateien und Dateisysteme – Überblick



Wie immer hängt das Verständnis von der Ebene der Betrachtung ab!

Benutzersicht (EDV)

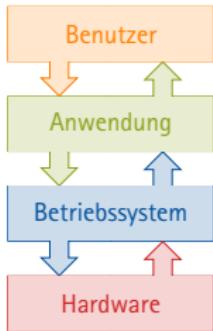
Wesentliches Merkmal ist die **Persistenz** der Daten:

- Sicherung und **Benennung** der Ergebnisse eines Programmablaufs auf dem **Hintergrundspeicher**.
- Spätere Weiterverwendung der Ergebnisse in anderem Programm.

Anwendungssicht

Wesentliches Merkmal ist die **Schnittstelle** zum Zugriff auf Daten:

- Einfachere Softwareentwicklung durch standardisierte Schnittstelle.
- Trennung von Programmtext und Daten(zugriffen).



Systemsicht

Wesentliches Merkmal ist die **Virtualisierung** des Zugriffs auf Daten:

- Multiplexing und Isolation von Geräten, Ressourcen, und Daten.
- Vereinfachte Einbettung weiterer Geräte und Ressourcen.



Benennung von Objekten – Namen

auch: Bezeichner

Namen

“ Name ist Schall und Rauch, umnebelnd Himmelsglut! ”

Goethe: Faust I

Benennung

“ Bezeichnung eines Gegenstands durch ein Wort oder mehrere Wörter.”

Wikipedia, <https://de.wikipedia.org/wiki/Benennung>

Namensverwaltung

Jedes Programm (auch das Betriebssystem) kann eigenen **Ressourcen** durch Zuordnung von **Zeichenketten** benennen und diese anderen Prozessen bekanntgeben.

Diese Zeichenketten müssen in der Regel eindeutig sein bezogen auf einen **Kontext**.

- Als BS-Konzept üblich (seit Multics [23]) ist die **Mehrwortbenennung**:
 - ein Wort ■ der **Name** relativ zu einem bestimmten **Kontext**
 - lokal (in seinem Kontext) eindeutig, global mehrdeutig
 - mehrere Wörter ■ der **Pfadname** im Namensraum zum benannten Ding
 - global eindeutige Bezeichnung des Namenskontextes
- Die **Mehrwortbenennung** sieht einen „Trenntext“ als **Separator** vor, den die Namensverwaltung im Betriebssystem definiert (z.B. `/` unter UNIX)
 - Alphabet und maximal Wortlänge der Namen gibt die Namesverwaltung vor
 - Die Namen selbst werden hingegen in der Regel nicht weiter interpretiert



4.3 Benutzersicht – Dateibaum

Benutzersicht: Dateibaum (*file tree*) Hierarchischer Namensraum

- Ein **Kontext** definiert einen **flachen Namensraum**.

→ Verzeichnis (*directory*)

- Durch Rekursion erhält man einen Namensraum **hierarchischer Struktur**.

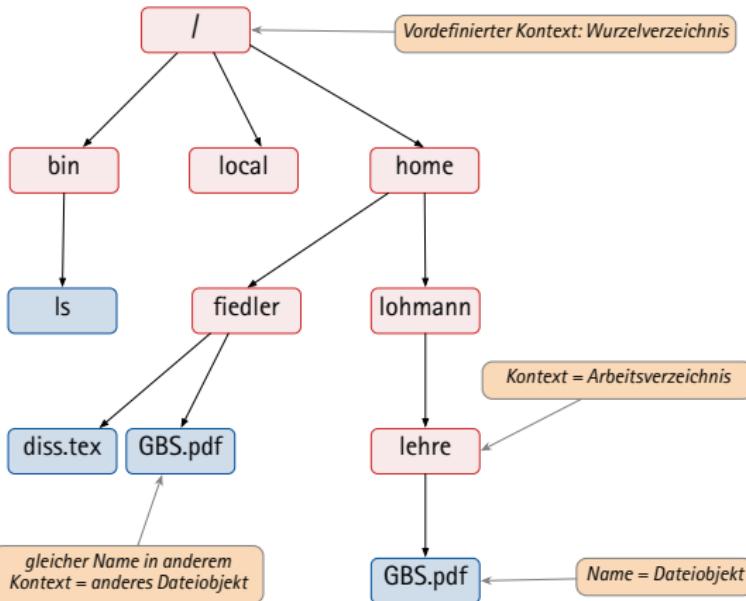
- Startkontext für den **Pfad** zu einer **Datei** ist das **Arbeitsverzeichnis** des Prozesses (implizit).

→ Pfade zur Datei **GBS.pdf** im Verzeichnis **lehre**:

- **GBS.pdf** oder **./GBS.pdf**
- **/home/lohmann/lehre/GBS.pdf**

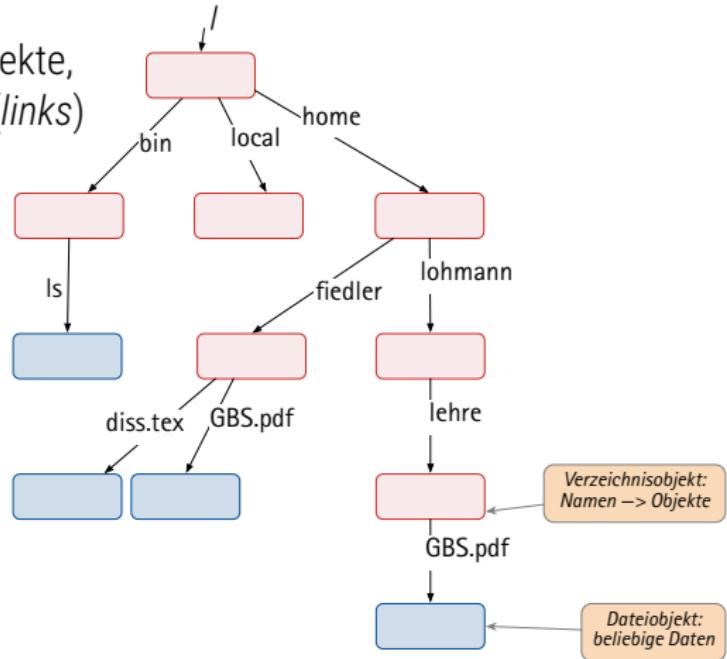
- mit **.** → Arbeitsverzeichnis
- mit **/** → Wurzelverzeichnis

(„dot“)
(„slash“)



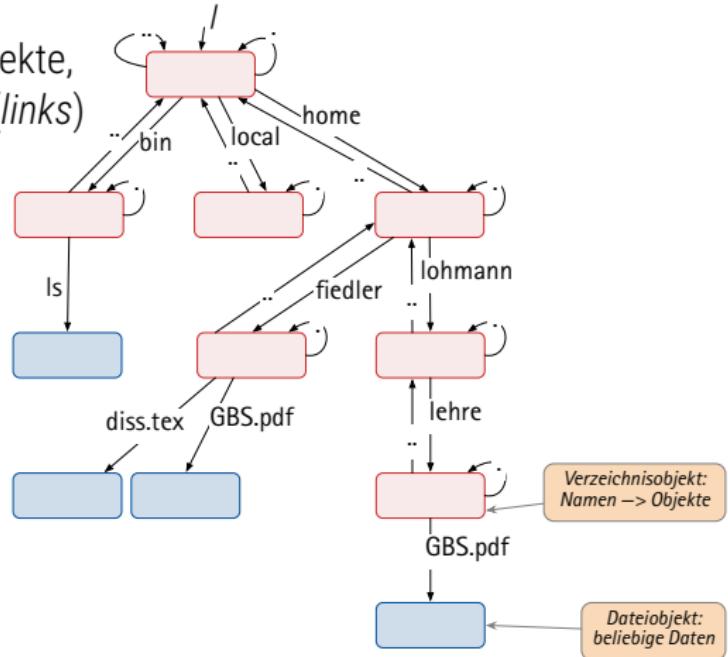
Benzersicht: Dateibaum (*file tree*) Hierarchischer Namensraum

- Tatsächlich sind nicht die Objekte, sondern die **Verknüpfungen** (*links*) zwischen ihnen benannt.
- Verzeichnis \mapsto Katalog von Verknüpfungen (Verweise)



Benzersicht: Dateibaum (*file tree*) Hierarchischer Namensraum

- Tatsächlich sind nicht die Objekte, sondern die **Verknüpfungen** (*links*) zwischen ihnen benannt.





Benutzersicht: Dateibaumgraph

hard links

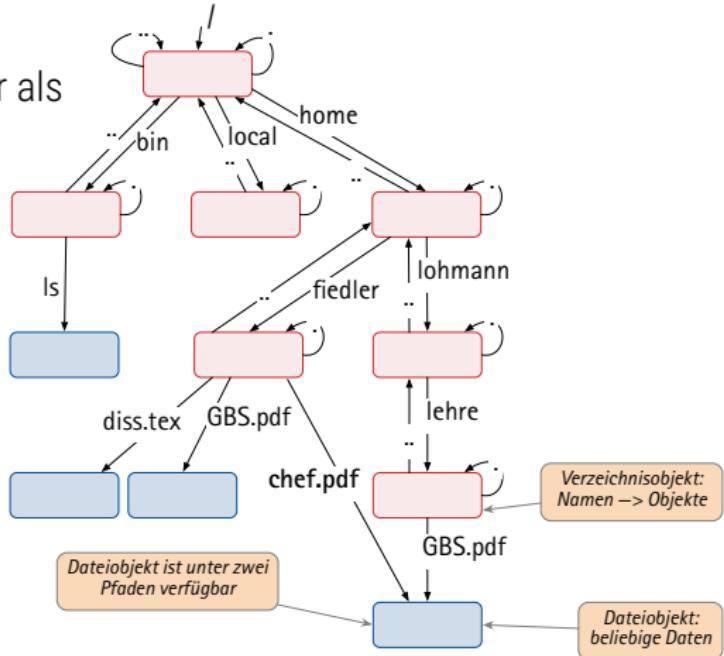
- Auf ein Dateiobjekt kann mehr als eine Verknüpfung verweisen.

- ↪ Anlegen eines **hard link**

```
cd /home/fiedler  
link ..../lohmamn/lehre/  
↳ GBS.pdf chef.pdf
```

- Hard links auf Verzeichnisse werden nicht unterstützt!*

- ↪ Dateibaum ist ein **azyklischer Graph**



Benutzersicht: Dateibaumgraph

hard links

- Auf ein Dateiobjekt kann mehr als eine Verknüpfung verweisen.

- ↪ Anlegen eines **hard link**

```
cd /home/fiedler  
link ..../lohmann/lehre/  
↳ GBS.pdf chef.pdf
```

- Hard links* auf Verzeichnisse werden nicht unterstützt!

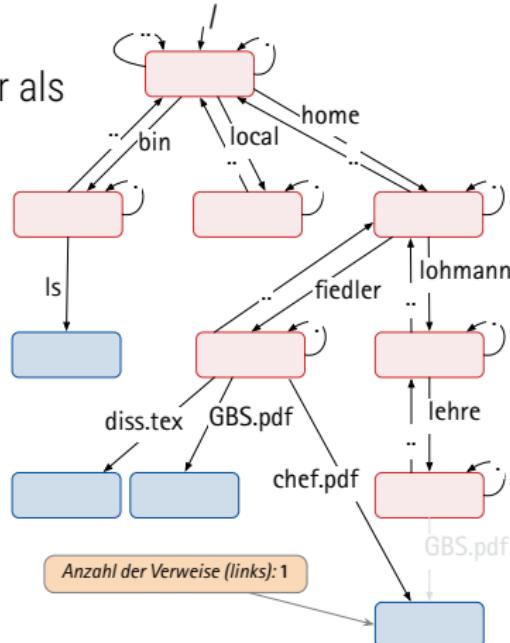
- ↪ Dateibaum ist ein **azyklischer Graph**

- Anzahl der *links* auf ein Dateiobjekt wird vermerkt

```
unlink ..../lohmann/lehre/GBS.pdf  
unlink chef.pdf
```

(reference counting)

- ↪ Freigabe erfolgt erst, wenn der letzte Verweis entfernt wurde



Benutzersicht: Dateibaumgraph

hard links

- Auf ein Dateiobjekt kann mehr als eine Verknüpfung verweisen.

- Anlegen eines **hard link**

```
cd /home/fiedler  
link ..../lohmann/lehre/  
↳ GBS.pdf chef.pdf
```

- Hard links* auf Verzeichnisse werden nicht unterstützt!

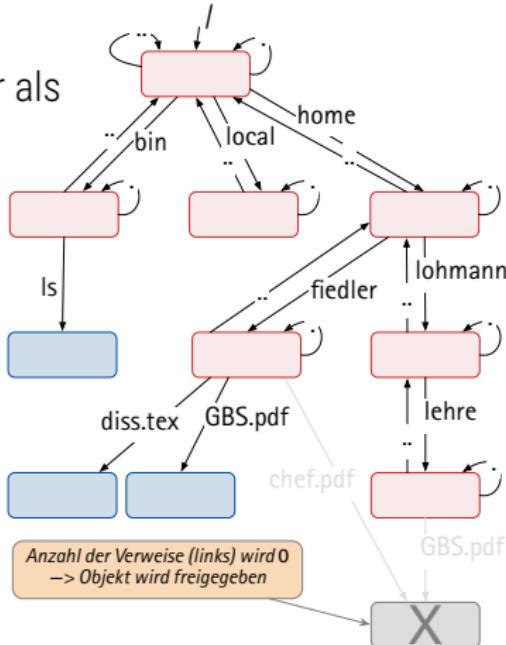
- Dateibaum ist ein **azyklischer Graph**

- Anzahl der *links* auf ein Dateiobjekt wird vermerkt

```
unlink ..../lohmann/lehre/GBS.pdf  
unlink chef.pdf
```

(reference counting)

- Freigabe erfolgt erst, wenn der letzte Verweis entfernt wurde





Benutzersicht: Dateigraph

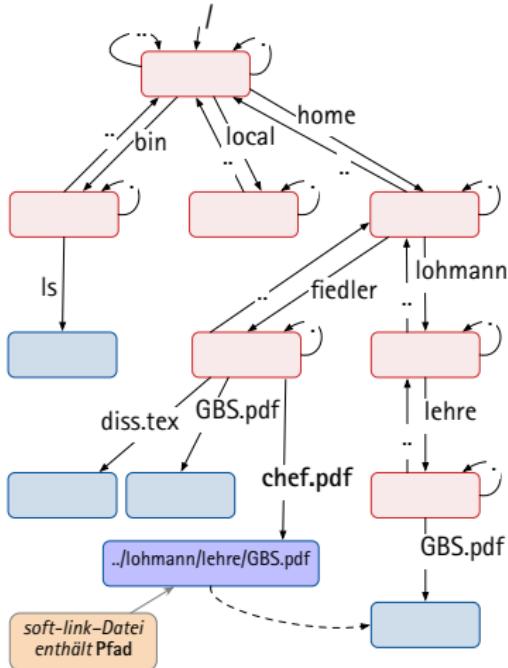
Symbolische Verknüpfungen *soft links*

- Alternative: Ein **soft link** verweist auf einen **Pfad**

- ↪ Soft link: Spezielle Datei
- ↪ Anlegen eines **soft link**

```
cd /home/fiedler  
ln -s ../lohmann/lehre/  
↳ GBS.pdf chef.pdf
```

- Soft links können auch auf Verzeichnisse oder in andere Dateisysteme verweisen.



Benutzersicht: Dateigraph

Symbolische Verknüpfungen *soft links*

- Alternative: Ein **soft link** verweist auf einen **Pfad**

↪ Soft link: Spezielle Datei

↪ Anlegen eines **soft link**

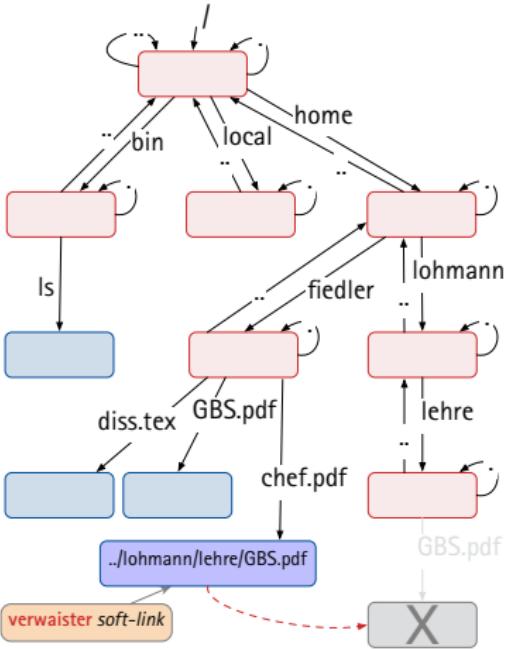
```
cd /home/fiedler  
ln -s ../lohmann/lehre/  
↳ GBS.pdf chef.pdf
```

- Soft links können auch auf Verzeichnisse oder in andere Dateisysteme verweisen.

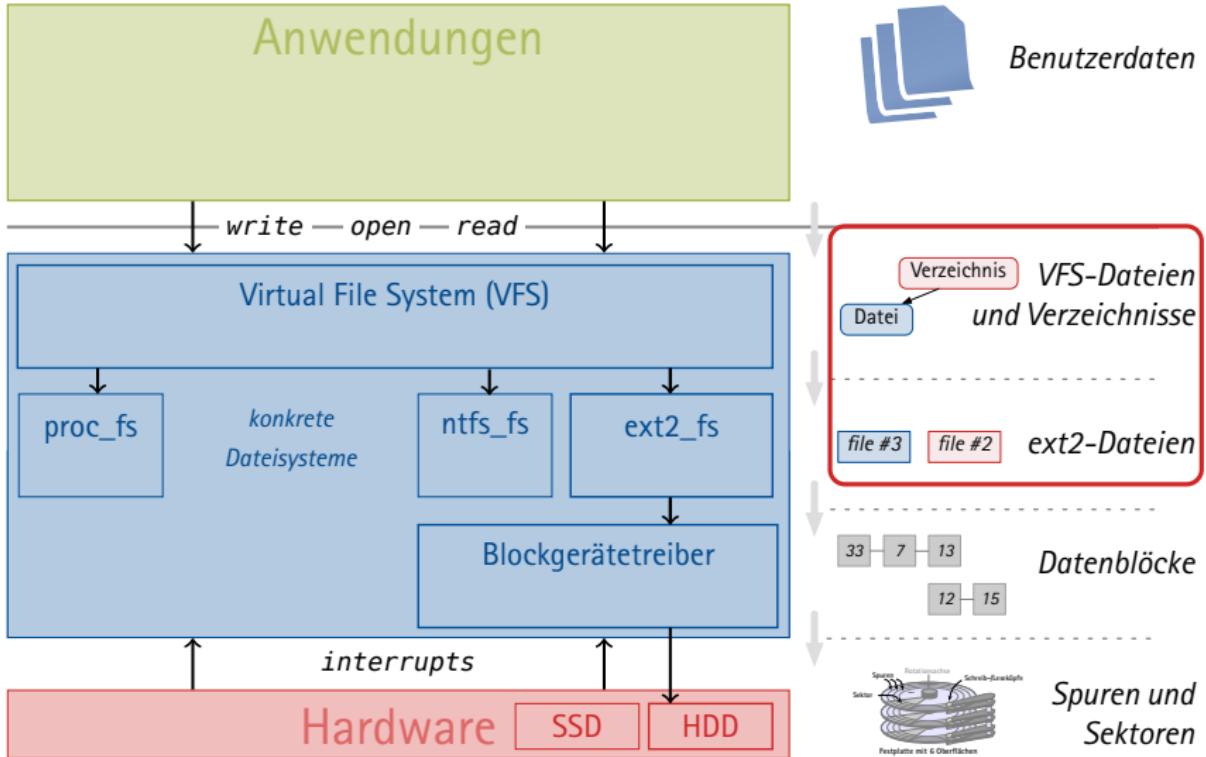
- Sie haben jedoch keinen Einfluss auf den *link counter*.

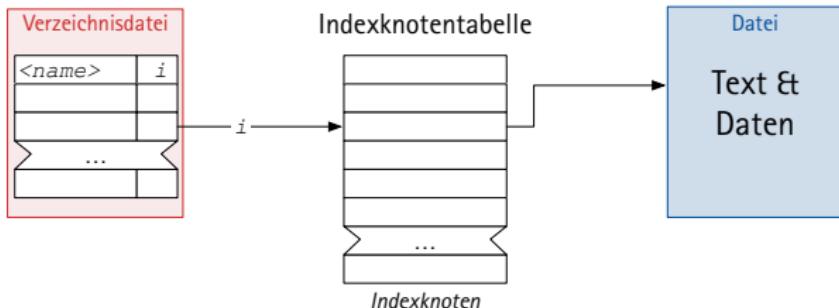
```
unlink ../lohmann/lehre/GBS.pdf
```

↪ Das Zielobjekt kann sich ändern oder der soft link kann **verwiesen**.



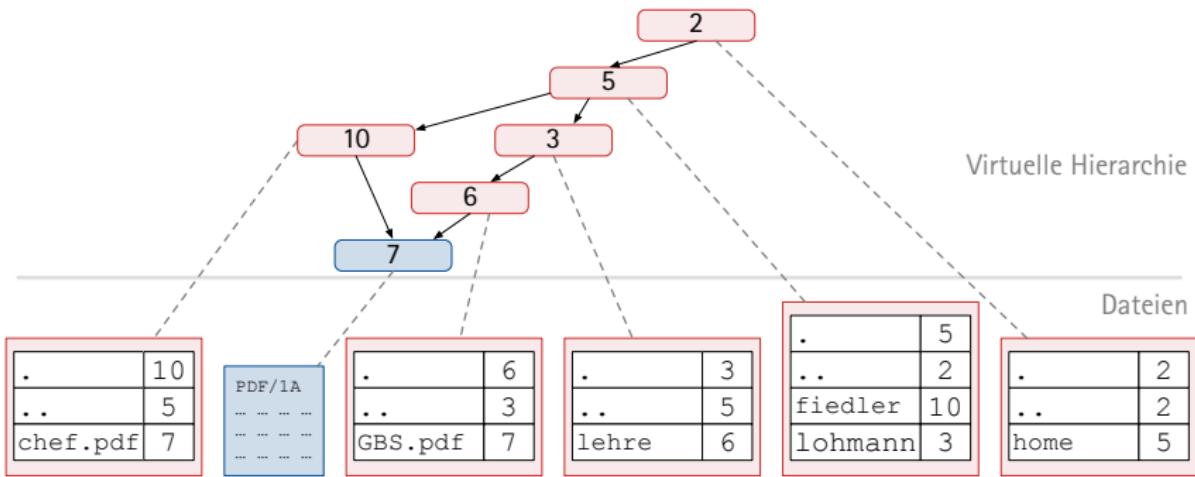
4.4 Systemsicht – Dateibaum





- Die **Indexknotentabelle** (*i-node table*) ist ein statisches Feld (array) von Indexknoten (*i-nodes*) und die **zentrale Datenstruktur**.
 - Indexknoten: Deskriptor des Objektes (Datei)
 - Indexknotennummer: **Eindeutige Referenz** des Objektes
- Ein **Verzeichnis** (*directory*) ist eine **Abbildungstabelle**, es übersetzt symbolisch repräsentierte Namen in Indexknotennummern
 - Spezielle Datei der Namensverwaltung des Betriebssystems
- Eine **Datei** (*file*) ist eine abgeschlossene Einheit zusammenhängender Daten beliebiger Repräsentation, Struktur und Bedeutung.

- Verzeichnisse definieren lediglich Paare: Name \mapsto Indexknotennummer
 - Sie werden von der Namensverwaltung als Dateien abgelegt und entsprechend über ihren Indexkonten referenziert (mit „/“ := Indexknoten 2).
 - Dateiattribute sind nicht im Verzeichnis, sondern im Indexknoten abgelegt.
 - Der Verzeichniseintrag kann für schnelleren Zugriff jedoch Kopien bereithalten.
- Der tatsächliche Namensraum ist flach! ↗ durchnummerierte Dateien





- Der **Indexknoten** aggregiert **Attribute** (Metadaten) eines (Datei-)Objektes:
 - Typ der Datei (reguläre Datei, Verzeichnisdatei, *soft link*, Gerät, Pipe, Socket)
 - Anzahl der Verzeichnisverweise (*link counter*)
 - Größe der Datei
 - Eigentümer der Datei (*user ID*)
 - Gruppenzugehörigkeit (*group ID*)
 - Rechte (jeweils für Eigentümer, Gruppe und Welt)
 - Zeitstempel (letzter Zugriff, letzte Änderung, letzte Attributänderung)
- Weitere Attribute sind implementierungsspezifisch für das **Dateisystem**
 - Zusätzliche Metadaten (z.B. „tags“ oder Rechteverwaltung über ACLs)
 - Insbesondere jedoch die **Objektdatenverweise** (z.B. Festplattenblöcke)
- Die Indexknotennummer ist **eindeutig** innerhalb des Dateisystems
 - Vergleichbar mit einer Adresse in einem Adressraum

Referenzzählung: Wird der
link counter null, wird die
Datei gelöscht. ↪ 4-16

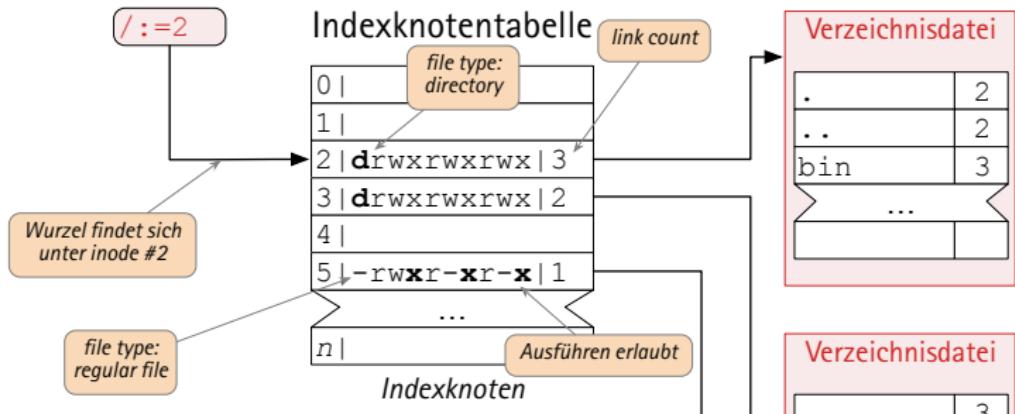
Name ist *kein* Attribut!

- **Namensbindung** (*name binding*) kommt zuerst, also die **Abbildung** der symbolischen Adresse in eine numerische Adresse
 - einen Pfadnamen mit einem Indexknoten assoziieren: creat⁽²⁾, link⁽²⁾
 - geschieht zum **Erzeugungszeitpunkt** eines Datei-/Verzeichnisnamens
 - diesen mit einem freien/belegten Indexknoten verknüpfen und
 - dann in ein Namensverzeichnis eintragen – unter Beachtung der Berechtigungen
- **Namensauflösung** (*name resolution*) kommt später, die **Umsetzung** der symbolischen Adresse in eine numerische Adresse
 - einen Indexknoten anhand eines Pfadnamens lokalisieren: open⁽²⁾
 - geschieht zum **Benutzungszeitpunkt** eines Datei-/Verzeichnisnamens
 - Für jeden einzelnen Namen im Pfad Indexknoten ermitteln,
 - Verzeichnisdatei laden und durchsuchen und
 - schließlich den Dateinamen (Blatt) auffinden – unter Beachtung der Berechtigungen

Berechtigungen auf der Verzeichnisdatei

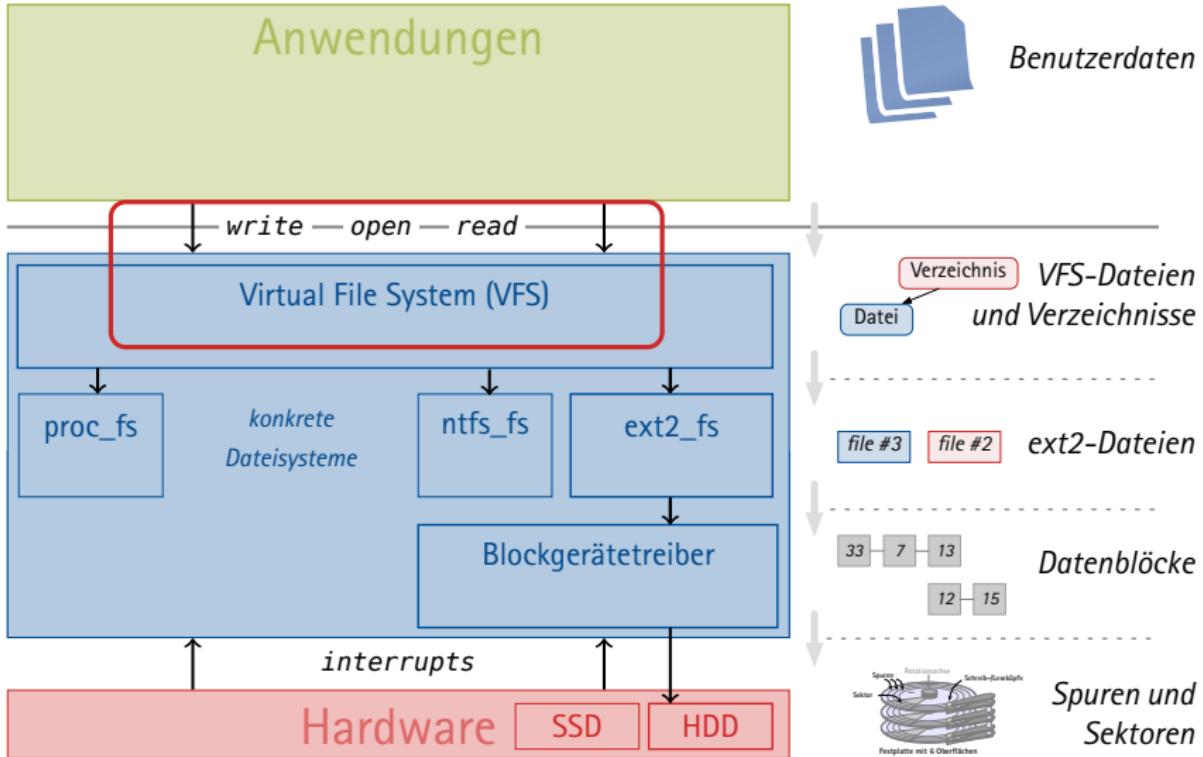
- Namensbindung erfordert Schreib- und Ausführungsrecht (-wx)
- Namensauflösung erfordert Ausführungsrecht (---x)
- Namensauflistung erfordert Leserecht (r---

Namensauflösung bei Ausführung von /bin/ls



1. Indexknoten für `/` ermitteln $\rightsquigarrow 2$
 - ist Verzeichnis (**d**) \rightsquigarrow Verzeichnisdatei lesen, darin
2. Indexknoten für `bin` ermitteln $\rightsquigarrow 3$
 - ist Verzeichnis (**d**) \rightsquigarrow Verzeichnisdatei lesen, darin
3. Indexknoten für `ls` ermitteln $\rightsquigarrow 5$
 - ist Datei (-) und ausführbar (x) \rightsquigarrow laden und ausführen

4.5 Anwendungssicht – Schnittstelle



■ Verzeichnisse bearbeiten: link⁽²⁾, symlink⁽²⁾, unlink⁽²⁾, rename⁽²⁾

```
int link(const char *oldpath, const char  
        ↴ *newpath);
```

erzeugt Verzeichniseintrag (*hard link*) für bestehende Datei ~ vergrößert *link count*

```
int symlink(const char *target, const  
            ↴ char *linkpath);
```

erzeugt neue *soft link*-Datei + Verzeichniseintrag

```
int unlink(const char *pathname);
```

entfernt Verzeichniseintrag ~ vermindert *link count*

```
int rename(const char *oldpath, const  
          ↴ char *newpath);
```

ändert/bewegt Verzeichniseintrag des Eintrags ~ *link count* bleibt gleich

■ Verzeichnisse erzeugen und löschen: mkdir⁽²⁾, rmdir⁽²⁾

```
int mkdir(const char *pathname, mode_t  
         ↴ mode);
```

erzeugt neue Verzeichnisdatei + Verzeichniseintrag

```
int rmdir(const char *pathname);
```

entfernt Verzeichnisdatei (Verzeichnis muss leer sein)

■ Verzeichnis auflisten: opendir⁽³⁾, readdir⁽³⁾, closedir⁽³⁾`DIR *opendir(const char *name);`öffnet ein Verzeichnis für readdir⁽³⁾`struct dirent *readdir(DIR *dirp);`liest nächsten Verzeichniseintrag; gibt **NULL** zurück, falls am Ende angekommen`int closedir(DIR *dirp);`schließt das mit opendir⁽³⁾ geöffnete Verzeichnis

```
struct dirent {  
    ino_t d_ino;           // i-node number  
    ...  
    char  d_name[256];   // filename  
};
```

Verzeichniseintrag kann weitere Elemente beinhalten, aber nur **d_ino** und **d_name** sind in POSIX spezifiziert

```
#include <stdio.h>  
#include <dirent.h>  
  
int main() {  
    DIR* dir = opendir( "." );    // öffne aktuelles Verzeichnis  
    struct dirent* e = NULL;  
    while( (e = readdir(dir)) )  // lese nächsten Eintrag bis  NULL  
        printf("%s\n", e->d_name); //  gebe ihn aus  
    closedir( dir );           // schließe Verzeichnis  
    return 0;  
}
```

Beispiel:

Auflisten aller Namen im aktuellen Verzeichnis.

■ Attribute auflisten (*i-node* auslesen): lstat⁽²⁾

```
int lstat(const char *pathname, struct stat *buf);
```

liest Attribute
(*i-node*-Daten)

```
struct stat {  
    dev_t      st_dev;          // device ID containing file  
    ino_t      st_ino;          // i-node number  
    mode_t     st_mode;         // protection  
    nlink_t    st_nlink;        // number of hard links  
    uid_t      st_uid;          // user ID of owner  
    gid_t      st_gid;          // group ID of owner  
    dev_t      st_rdev;         // device ID (if special file)  
    off_t      st_size;         // total size, in bytes  
    blksize_t   st_blksize;       // blocksize for filesystem IO  
    blkcnt_t   st_blocks;        // number of blocks allocated  
    struct timespec st_atim;    // time of last access  
    struct timespec st_mtim;    // time of last modification  
    struct timespec st_ctim;    // time of last status change  
};
```

Ergebnis von
lstat⁽²⁾: *i-node*-
Datenstruktur des
Betriebssystems
(VFS)

■ Dateien bearbeiten: open⁽²⁾/creat⁽²⁾, read⁽²⁾, write⁽²⁾, close⁽²⁾

```
int open(const char *path, int flags, mode_t m);
```

öffnet bestehende Datei; gibt
Dateideskriptor zurück

```
int creat(const char *path, mode_t mode);
```

erzeugt Datei +
Verzeichniseintrag; gibt
Dateideskriptor zurück

```
ssize_t read(int fd, void *buf, size_t count);
```

liest aus geöffneter Datei
(Deskriptor **fd**); gibt Anzahl der
gelesen Bytes zurück (0 bei
Dateiende)

```
ssize_t write(int fd, void *buf, size_t count);
```

schreibt in geöffnete Datei
(Deskriptor **fd**); gibt Anzahl der
gelesen Bytes zurück (0 bei
Dateiende)

```
int close(int fd);
```

schließt geöffnete Datei
(Deskriptor **fd**)

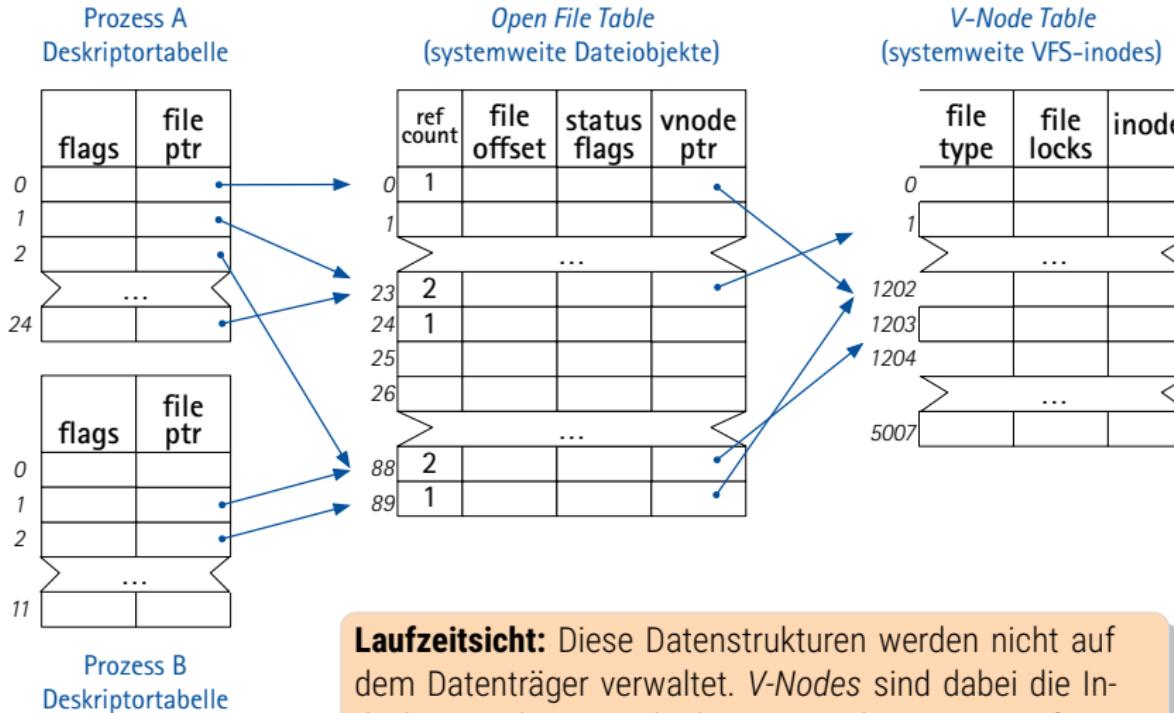
4.6 Systemsicht – Offene Dateien

- Das Lesen aus / Schreiben in eine Datei erfordert einen **Dateideskriptor**.
 - Nichtresidentes Betriebsmittel, repräsentiert eine prozesslokale **Befähigung** (Zugriffsrechte) für den Zugriff (z.B. mit `read(2)`) auf eine **geöffnete Datei**.
 - Anforderung mit: `open(2)`, `creat(2)`, `dup(2)`, ...
 - Freigabe mit: `close(2)` – sowie implizit bei Prozessbeendigung
 - Implizit startet ein Prozess mit drei geöffneten Deskriptoren für die Standard{ein|aus|fehleraus}gabe (`stdin := 0` | `stdout := 1` | `stderr := 2`)
 - Dateideskriptor verweist intern auf ein **Dateiobjekt**
 - Nichtresidentes Betriebsmittel, repräsentiert systemweit eine **geöffnete Datei**
 - Erzeugung: implizit, beim ersten Öffnen der Datei (Deskriptorerzeugung)
 - Freigabe: implizit, wenn der letzte Deskriptor freigegeben wurde
- **Deskriptortabelle** pro Prozess (*file descriptor table, fd-table*)
- Systemweite **Dateiobjekttabelle** (*open file table, of-table*)



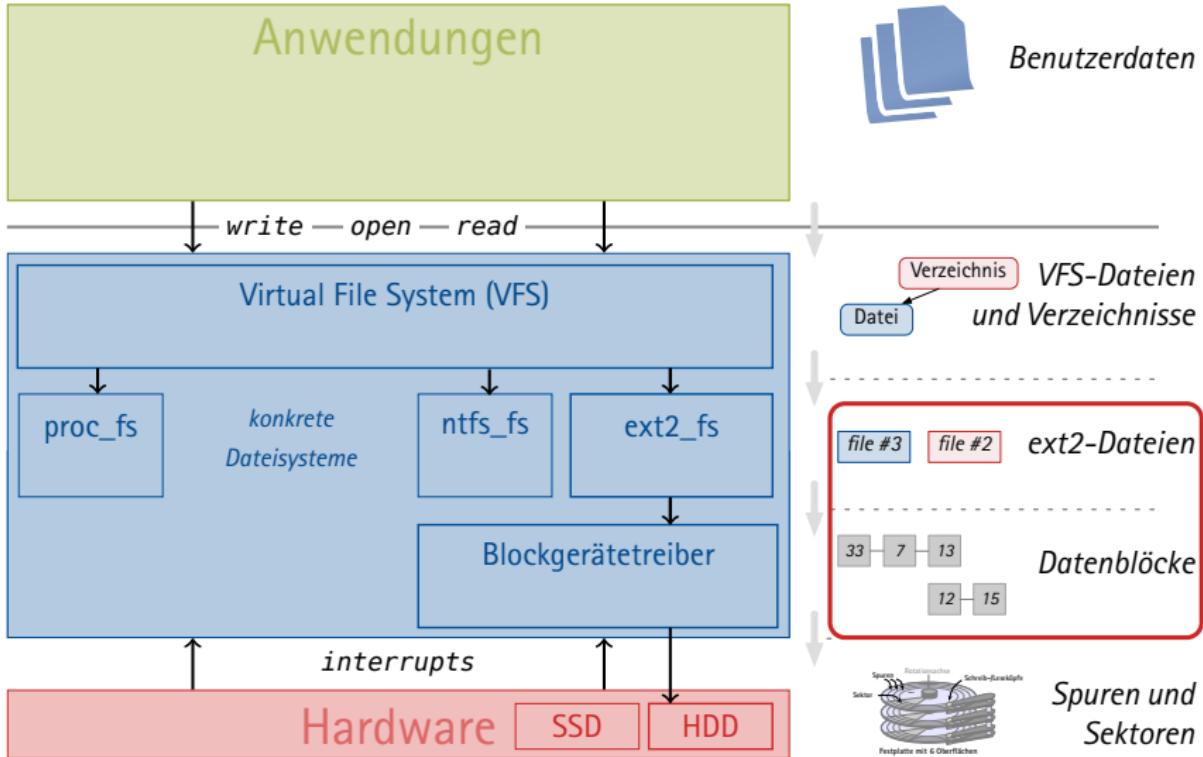
Systemsicht: Offene Dateien

Verwaltung im Kern

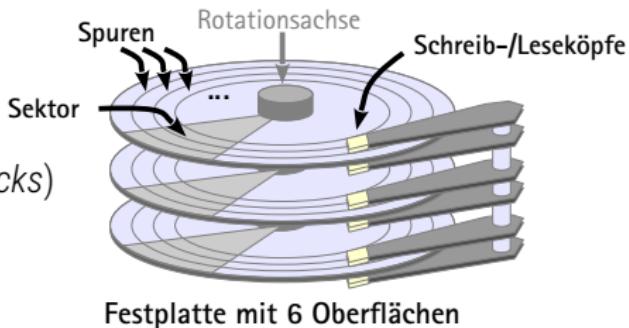


Laufzeitsicht: Diese Datenstrukturen werden nicht auf dem Datenträger verwaltet. *V-Nodes* sind dabei die Indexknoten des VFS, die bei einer echten Datei auf einen echten Indexknoten verweisen.

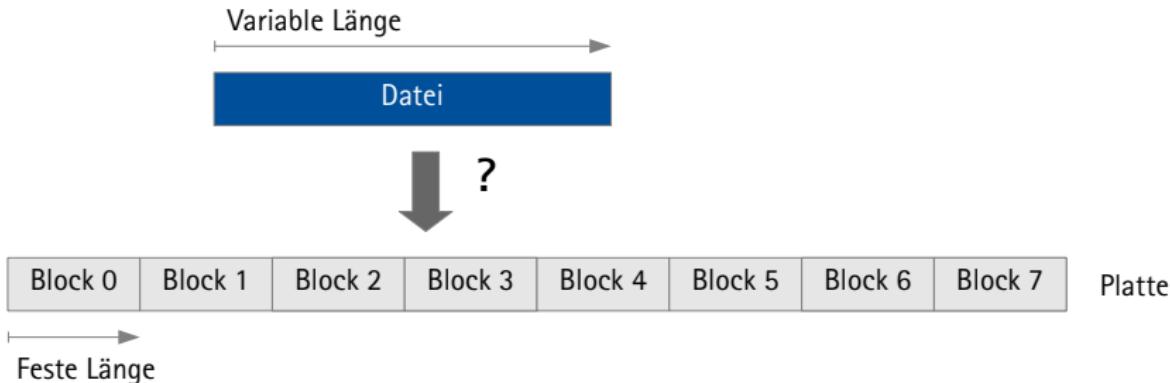
4.7 Systemsicht – Blockgeräte



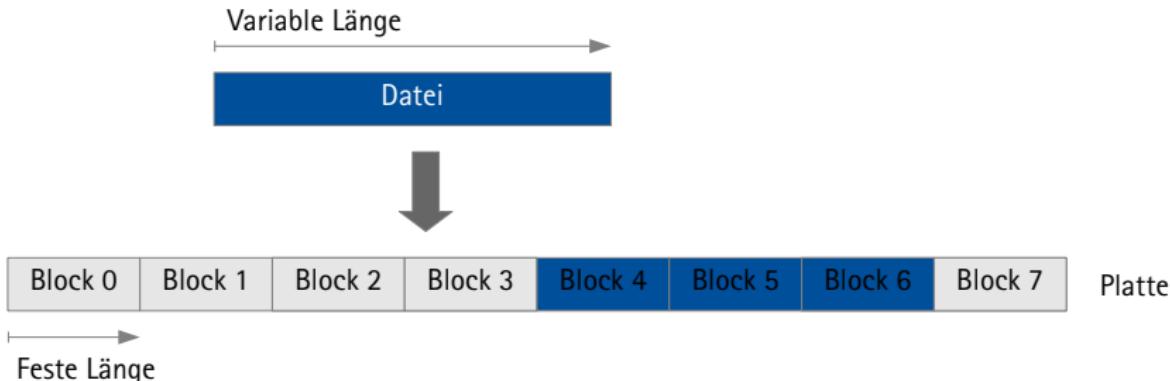
- **Dateisystem** (*file system*): Prinzip zur **strukturierte Ablage** von Informationen auf einem **Datenträger**
- Wesentliches internes Strukturelement ist der **Block**
 - Aufgabe: Abbildung von **Dateien** und **Verzeichnissen** auf Blöcke
 - Aufgabe: Allokation und Freigabe von Blöcken
- Benötigt: Verwaltungsinformationen
 - Tabelle der **Indexknoten**
 - Liste der **freien Blöcke**
 - Liste der fehlerhaften Blöcke (*bad blocks*)
 - Zentrale Metadaten (*super block*)



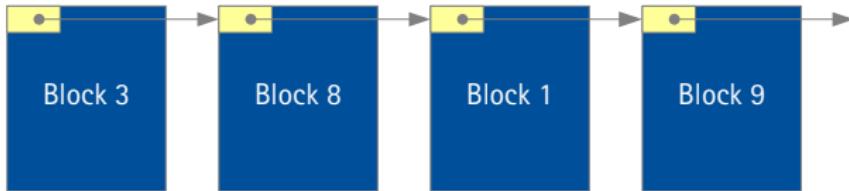
- Datenträgerspeicher wird in der Regel in Form von **Blöcken** verwaltet
 - Block: Element einer Datenträgeroperation
 - Typische Größe: 512 Byte bis 64 KiB
- Tatsächliche Datei muss auf eine **Folge von Blöcken** abgebildet werden
 - Es entsteht **Verschnitt** – nicht jeder Block kann voll genutzt werden
 - Speicherung und **Verwaltung** der Blockfolge erforderlich



- Am einfachsten ist die **kontinuierliche Speicherung** (z.B. CDFS)
 - Blockfolge: Blöcke mit aufsteigenden Blocknummern
 - Verwaltung: Tupel aus Startblock und Anzahl der Folgeblöcke
- Hoher Durchsatz, aber aufwändige Freispeicherverwaltung
 - Kontinuierliches Lesen bringt bei mechanischen Festplatten Vorteile aber Dateien können nicht ohne Weiteres vergrößert werden
 - Einsatz auf nichtmodifizierbaren Datenträgern (z.B. DVDs) sowie auf seriellen Datenträgern (z.B. Magnetbänder)



- Alternative: **verkettete Speicherung** (z.B. Commodore 64)
 - Blockfolge: Einfach verkettete Liste aus Blöcken
 - Verwaltung: Jeder Block enthält Zeiger des nächsten Blocks
- ~~> Größenänderung einfach, aber **Mischung von Nutz- und Verwaltungsdaten**
 - Nutzdatengröße ist keine Zweierpotenz mehr ~~ Ungünstig für *Pageing*
 - Zugriffszeit für wahlfreien Zugriff steigt mit der Dateigröße (linearer Suche)
 - Hohe Fehleranfälligkeit: Ein einzelner defekter Block zerstört gesamte Kette

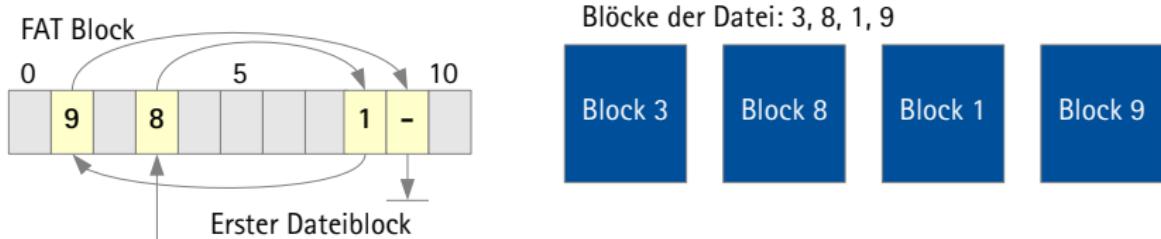


■ Variante: **verkettete Speicherung mit FAT** (z.B. MS-DOS)

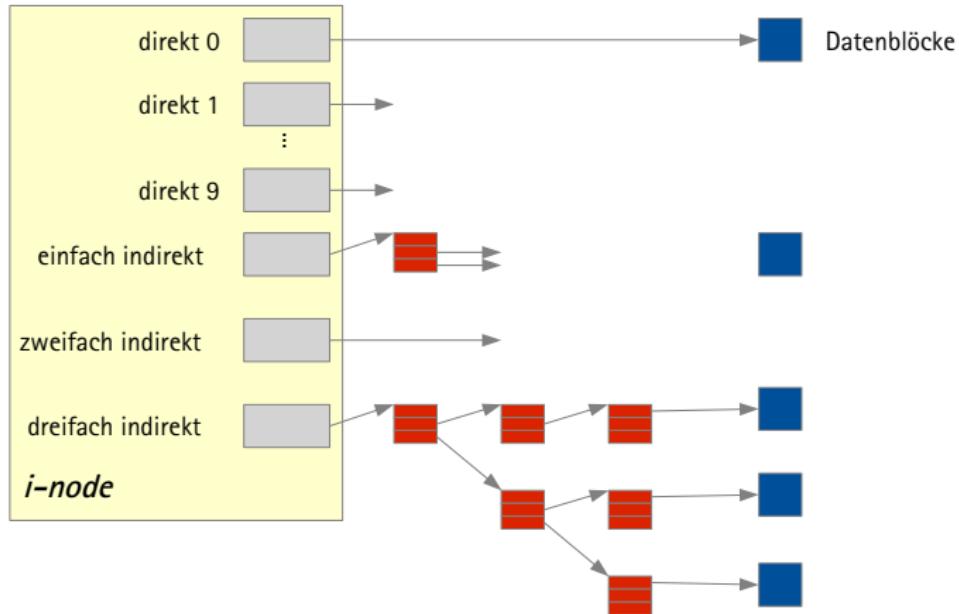
- Blockfolge: Einfach verkettete Liste aus Blöcken
- Verwaltung: Blockfolge wird in **seperaten Blöcken** gespeichert

~~> FAT-Ansatz (*File Allocation Table*)

- Kompletter Inhalt des Datenblocks ist nutzbar
- Einschränkung der Fehleranfälligkeit durch mehrfaches Speichern der FAT

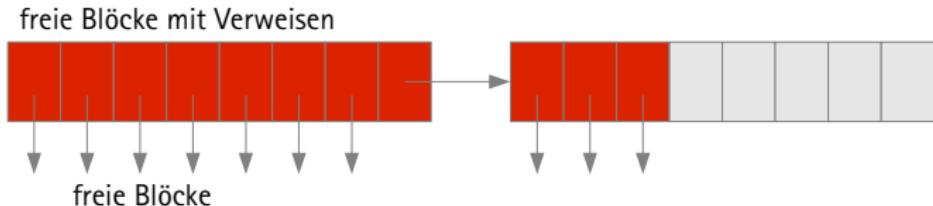


- Alternative **indiziertes Speichern** (z.B. Berkely FFS, ext2/3/4)
 - Blockfolge: Wird in speziellen Indexblöcken abgelegt
 - Verwaltung: Verweise auf Indexblöcke stehen im Indexknoten

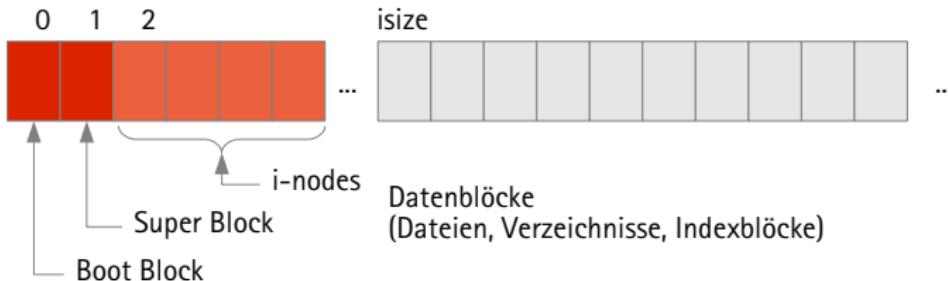


- Potentielle Nachteile der Indizierung
 - Zugriff auf Nutzdatenblöcke erfolgt indirekt $\rightsquigarrow Z$ zusätzliche Blockzugriffe
 - Anzahl der Indexeinträge beschränkt \rightsquigarrow Beschränkte Dateigröße F
 - Annahme: Blockgröße $B = 1024$ Byte, Blocknummer ist 32-Bit Wert
 - Bis zu 10 Blöcke direkt im *i-node* indiziert
 $\rightsquigarrow Z = 0$ für $F \leq 10$ KiB
 - Bis zu 2^8 Blöcke werden zusätzlich einfach-indirekt indiziert
 $\rightsquigarrow Z \leq 1$ für $F \leq 266$ KiB
 - Bis zu 2^{16} Blöcke werden zusätzlich zweifach-indirekt indiziert
 $\rightsquigarrow Z \leq 2$ für $F \leq 65.802$ KiB (≈ 64 MiB)
 - Bis zu 2^{24} Blöcke werden zusätzlich dreifach-indirekt indiziert
 $\rightsquigarrow Z \leq 3$ für $F \leq 16.843.018$ KiB ($\approx 16,8$ GiB)
- Keine oder geringe zusätzlichen Kosten bei kleinen Dateien.
- Sehr große Dateien ($F > 16,8$ GiB) durch Anpassung der Blockgröße B .

- Auch die **verfügaren Blöcke** müssen entsprechend verbucht werden
 - Verfahren ähnen der Verwaltung von freiem Hauptspeicher
- Variante: **Bitmaps** markieren für jeden Block, ob er belegt ist oder nicht
 - eine Metadatei des Dateisystems (Beispiel: NTFS \$Bitmap)
- Variante: **Verkette Listen** repräsentieren frei Blöcke
 - Verkettung kann in den freien Blöcken vorgenommen werden
 - Optimierung: Indizierung freier Blöcke (Beispiel: UNIX-Dateisysteme)

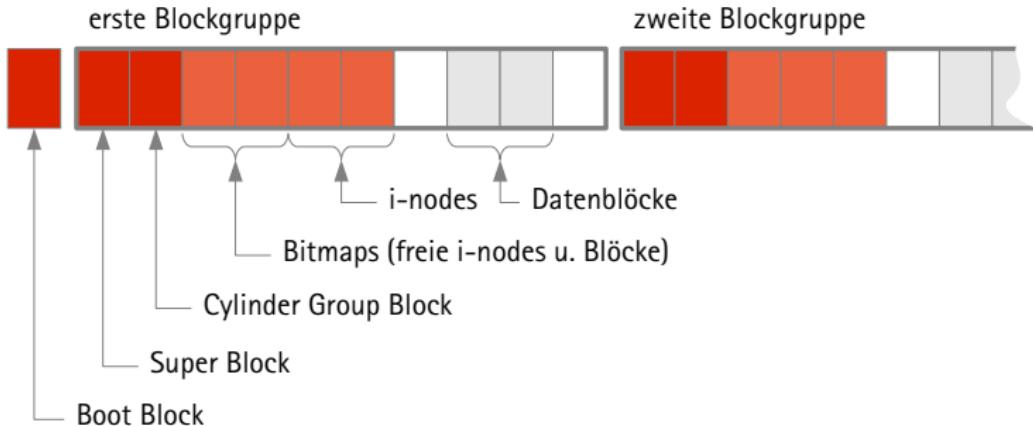


- Blockorganisation des **UNIX System V File System**



- **Boot Block** enthält Informationen zum Laden des Betriebssystems
 - eine spezielle Datei des Dateisystems (Beispiel: NTFS \$Bitmap)
- **Super Block** enthält Metadaten des Dateisystems
 - Anzahl der Blöcke und Anzahl der *i-nodes*
 - Anzahl und Liste der freien Blöcke und freien *i-nodes*
 - Attribute des Dateisystems, wie *modified* oder *read-only*

■ Blockorganisation des **Linux ext2/3/4 File System**



■ **Blockgruppe**: Eine Menge aufeinanderfolgender Blöcke

- Datei wird möglichst innerhalb einer Blockgruppe gespeichert
- Verteilung der Indexknoten über den Datenträger
- Kopie des Superblocks in jeder Blockgruppe

→ Kürzere Positionierungszeiten bei mechanischen Festplatten

4.8 Zusammenfassung



Zusammenfassung: Dateien und Dateisysteme

- **Datei:** abgeschlossene und benannte Einheit von **Daten**, intern repräsentiert durch einen **Indexknoten**.
- **Verzeichnis:** bildet **Namen** auf **Indexknotennummern** ab, Kontext für die (hierarchische) Namensauflösung.
- **Dateisystem:** bildet **Dateien** auf Blöcke eines Datenträgers ab und sorgt so für die **Persistenz** der Daten.
- Mit dem **VFS** **virtualisiert** das Betriebssystem Dateien und Dateisysteme.
 - Einheitliche Schnittstelle über alle Dateisysteme (open⁽²⁾, read⁽²⁾, close⁽²⁾, ...).
 - Einbindung von **Pseudodateien** (Pipe, Socket, tty, Gerät, Zustandsvariable).
 - Verwaltung **geöffneter Dateien** durch prozesslokale **Dateideskriptoren** und systemweite **Dateiobjekte**.

4 Dateien und Dateisysteme

5 Prozesse und Fäden

- 5.1 Einordnung
- 5.2 Anwendungssicht – Simultanverarbeitung
- 5.3 Anwendungssicht – Prozesszustände
- 5.4 Anwendungssicht – POSIX-Prozesse
- 5.5 Systemsicht – Prozesse, Fäden (*Threads*) und Fasern (*Fibers*)
- 5.6 Systemsicht – Prozesskontrollblöcke
- 5.7 Zusammenfassung

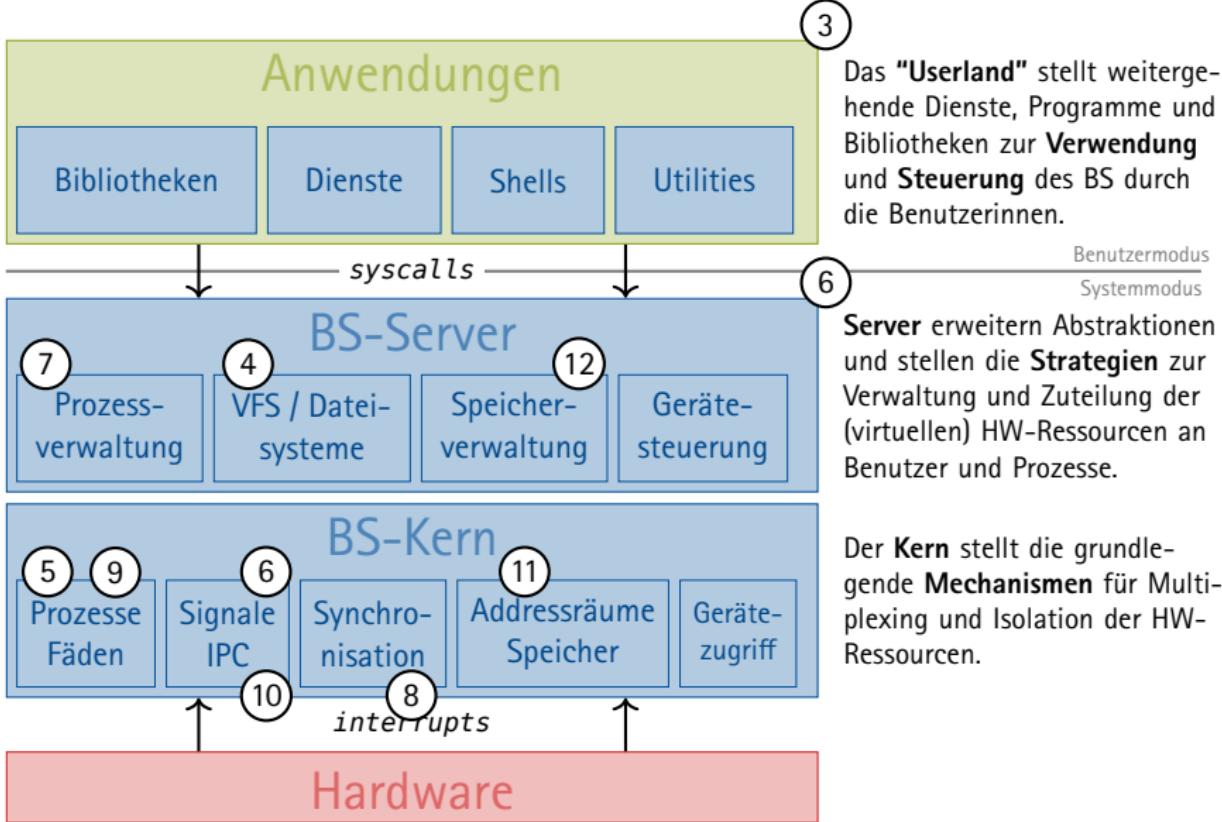
6 Unterbrechungen, Ausnahmen, Signale

7 Prozessverwaltung

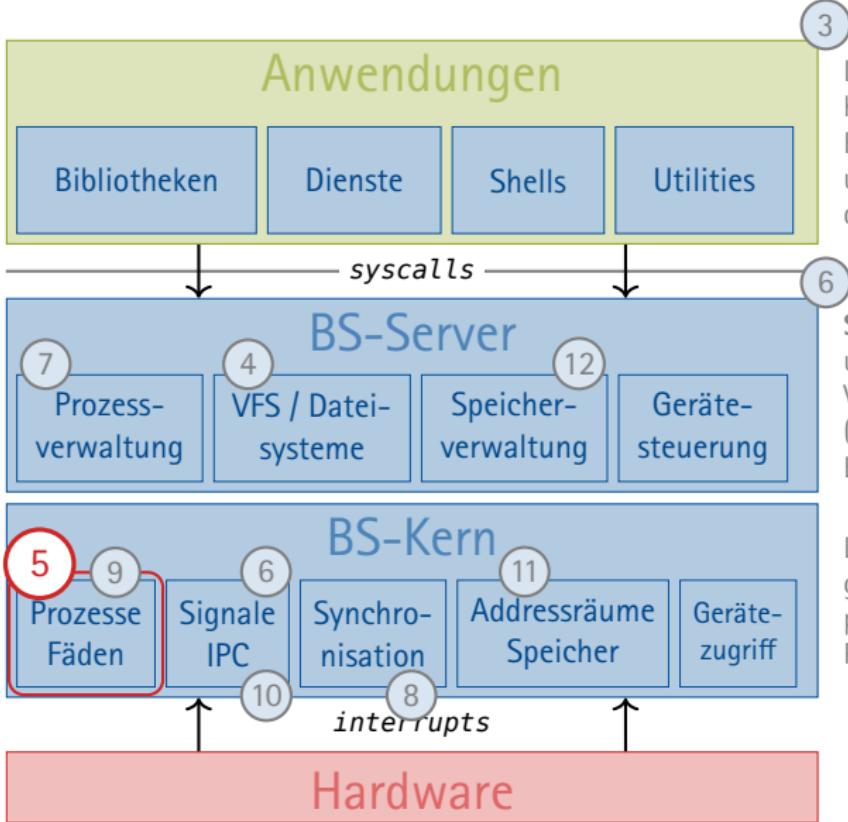


5.1 Einordnung

5 Prozesse und Fäden – Einordnung



5 Prozesse und Fäden – Einordnung



Worum geht es in diesem Kapitel?

■ **Erläutern** des Prozessbegriffs und seiner Ausprägungen in der Informatik und im Betriebssystem

- Unterscheidung Prozess, Programm, Prozessobjekt
- Simultanverarbeitung (*multiprogramming, multitasking, multithreading*)
- Betriebsmittel, Prozesszustände

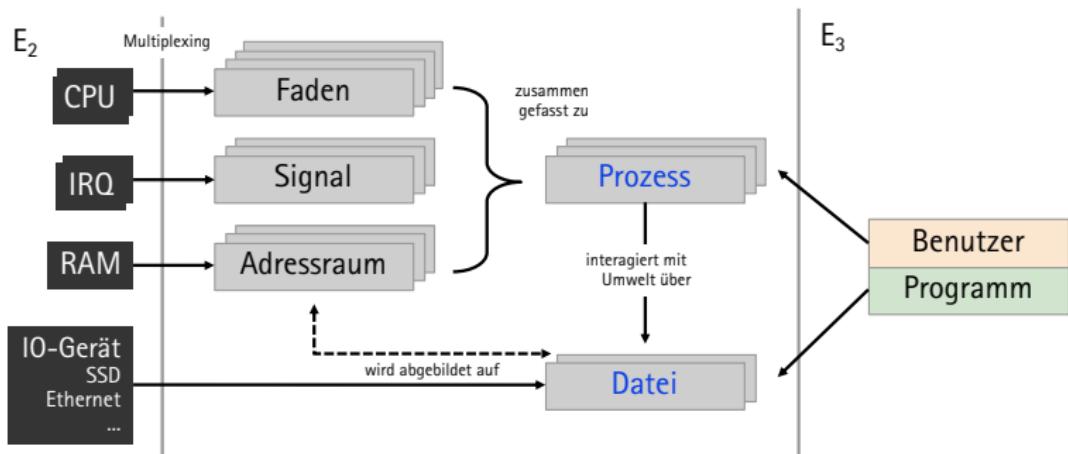
■ **Verstehen** des UNIX-Prozessmodells

- Prozessinkarnationen und Prozessobjekte \mapsto Prozesskontrollblock (PCB)
- Prozesserzeugung und Zerstörung, Prozessbaum, fork⁽²⁾ und exec⁽²⁾
- Gewichtsklassen: Prozesse und Fäden, Prozess als Virtualisierungskonzept

→ Prozess als zentrale Abstraktion des Betriebssystems



- Virtuelle Hardwareressourcen werden repräsentiert durch die grundlegenden Konzepte (**Abstraktionen**) eines Betriebssystems. **In UNIX sind das:**
 - **Prozess** (Adressraum + Fäden) ↪ virtueller „Computer“
 - **Datei** ↪ virtuelles „Peripheriegerät“
(auch persistenter Speicher ↪ „echte Datei“)



“ In UNIX ist jedes Objekt entweder ein Prozess oder eine Datei! ”

Informatik Folklore



Was ist ein Prozess? – Informatik

→

1-14

Das Verständnis hängt von der Ebene der Betrachtung ab!

Benutzersicht (EDV): Prozess ≡ Ablauf

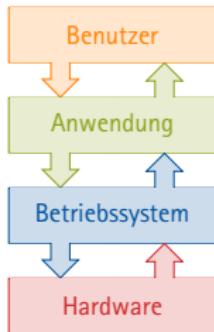
„Gerichteter Ablauf eines Geschehens“ [38]

- {Geschäfts | Datenverarbeitungs | Softwareentwicklungs}prozess
- Formal beschrieben (und dadurch standardisiert)

Anwendungssicht (Informatik): Prozess ≡ Programmablauf

Ein Programm in Ausführung durch einen Prozessor

- (inter)aktive Elemente des Rechensystems
- zur Ausführung gebracht durch das Betriebssystem



Systemsicht (Betriebssystem): Prozess ≡ Ausführungskontext

Systemobjekt, dass eine konkrete Prozessausführung beschreibt

- Umgebung für die Inkarnation eines Programmablaufs („Prozessor“ oder „virtuelle Maschine“ für den Programmablauf)
- Einheit der Zuteilung von Betriebsmitteln (CPU, Speicher, ...)



5.2 Anwendungssicht – Simultanverarbeitung

Prozess \equiv Programmablauf

Ein **Prozess** ist ein **Programm** in Ausführung durch einen **Prozessor**.

Programm \equiv Anweisungsfolge

Ein **Programm** spezifiziert eine Folge von **Anweisungen** für einen **Prozessor**.

- Das Programm ist statisch (passiv) \longleftrightarrow Der Prozess ist dynamisch (aktiv)
- Die Art einer Anweisung hängt von der betrachteten Abstraktionsebene ab (bzw. des „Prozessors“ dieser Ebene \rightsquigarrow kann z. B. auch ein Interpreter sein).

C-Programm

`var++;`

$E_5: \mathbf{1}$ C-Instruktion

Maschinenprogramm

$\xrightarrow{\quad}$
`movl 4(%esp), %eax
addl $1, eax,
movl eax, 4(%esp)`

$E_3: \mathbf{3}$ Maschinenbefehle

Mikroprogramm

$\xrightarrow{\quad} \dots$

$E_1: \textcolor{red}{n} \mu\text{-Ops}$

\hookrightarrow Eine Anweisung ist damit **nicht zwingend unteilbar auf E_1** (atomar)!



(*multiprocessing*)

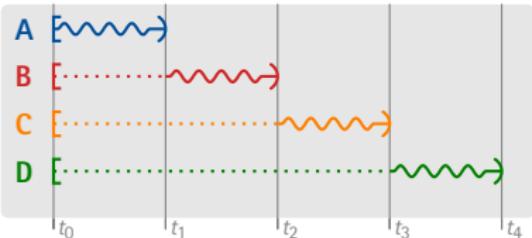
Das Betriebssystem bringt Programme zur Ausführung, indem dazu **Prozesse** erzeugt, bereitgestellt und begleitet werden.

- Prozesse sind **das** Mittel, um (pseudo/quasi) gleichzeitige Programmabläufe stattfinden zu lassen. ~~> **Parallelität**
 - *multiprogramming* ~~> mehrere Programme
 - *multitasking* ~~> mehrere Aufgaben mehrerer Programme
 - *multithreading* ~~> mehrere Fäden eines oder mehrerer Programme
- Ablauf: Teil eines einzelnen oder mehrerer Programme
 - zum Ablaufstart *lastet* das BS einen Prozess ein → *dispatching*
 - hierzu *plant* das BS Prozesse entsprechend ein → *scheduling*
- Üblich ist das **Zeitteilverfahren** (*time-sharing*; CTSS [6])

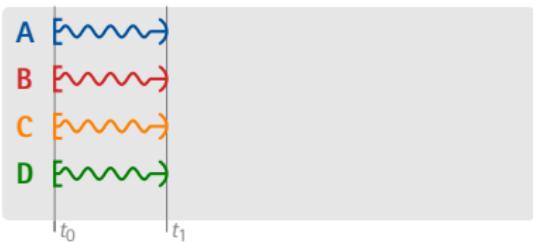
Mögliche Abläufe von 4 Prozessen

A, B, C und D auf einem Betriebssystem, abhängig von der Ablaufplanung und den verfügbaren Prozessoren.

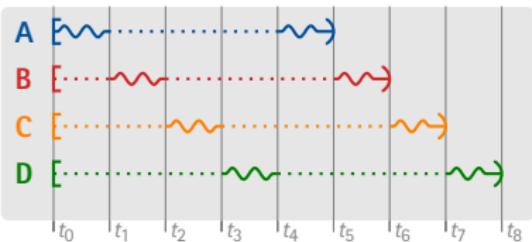
(a) sequentiell (1 Prozessor)



(b) echt parallel (4 Prozessoren)



(c) pseudo-parallel (1 Prozessor)

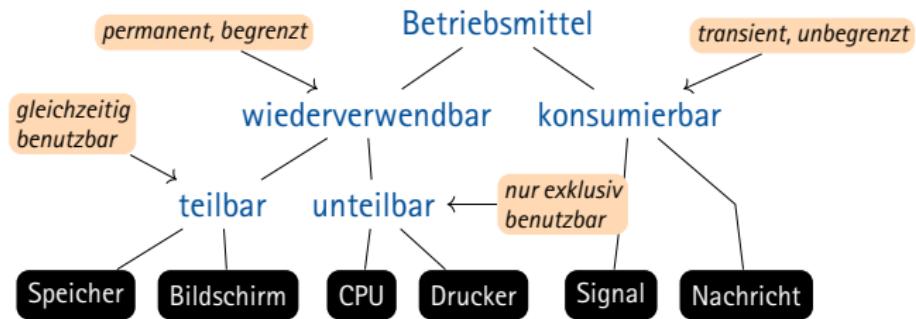


Nicht berücksichtigt ist hier der Overhead t_{CS} für das Einlasten eines Prozesses. Dieser beträgt im Fall (a) $4 \cdot t_{CS}$, im Fall (b) $1 \cdot t_{CS}$ und im Fall (c) sogar $8 \cdot t_{CS}$.

Simultanverarbeitung von Prozessen

Betriebsmittel

- Ein Programmablauf ist möglich, wenn
 1. er dem Betriebssystem explizit gemacht worden ist und
 2. alle von ihm benötigten **Betriebsmittel** (real/virtuell) verfügbar sind.
- Klassifikation von **Betriebsmitteln**: (vgl. [15, 16])



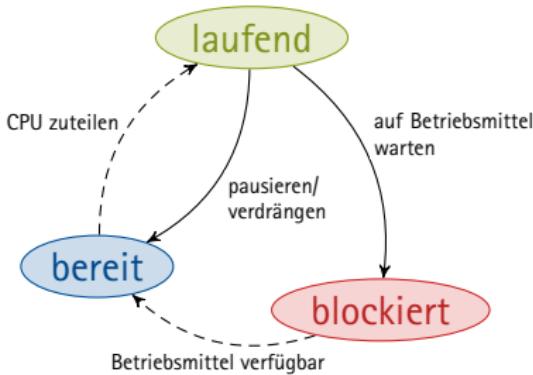
Ist die gemeinsame Benutzung (*sharing*) wiederverwendbarer Betriebsmittel oder eine logische Abhängigkeit von konsumierbaren Betriebsmitteln gegeben, so wird **Synchronisation** erforderlich \rightsquigarrow **Prozess muss gegebenenfalls „warten“!**

- **Wiederverwendbare Betriebsmittel** sind nur begrenzt verfügbar
 - Typisch für **Hardware**, wie CPU oder Speicher
 - Einige sind **aufteilbar** für eine **gleichzeitige Benutzung** durch **mehrere** Prozesse.
 - **Unteilbare** müssen **einzelnen Prozessen** zeitweise **exklusiv** zugeordnet werden.
 - **Konsumierbare Betriebsmittel** werden „generiert“ und „verbraucht“.
 - Typisch für **E/A-Operationen** → jede Art von Interaktion mit der Umwelt.
 - Das Ergebnis einer E/A-Operation ist ein konsumierbares Betriebsmittel.
- Prozesse sind in ihrem Fortschritt **logisch abhängig** von konsumierbaren Betriebsmitteln (z. B. Verfügbarkeit der Eingabedaten).
- Bei der Simultanverarbeitung liegen sie zusätzlich im **Wettstreit** um wiederverwendbare Betriebsmittel (z. B. CPU oder Speicher).



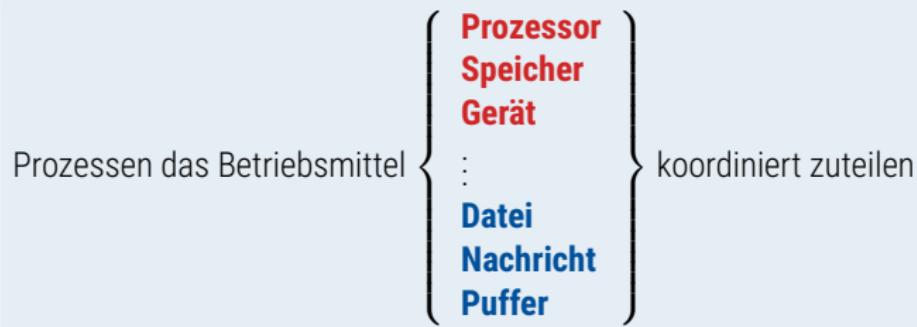
5.3 Anwendungssicht – Prozesszustände

- Während seiner Verarbeitung ist ein Prozess in einem der Zustände:
 - **blockiert**, so lange die benötigten Betriebsmittel nicht verfügbar sind.
 - **bereit**, wenn die benötigten Betriebsmittel verfügbar sind.
 - **laufend**, wenn ihm zusätzlich das Betriebsmittel CPU zugeteilt wurde.
- Technisch führt nur der **laufende Prozess** selber Zustandsübergänge aus
 - **explizit** —→ geht **selber** aus „laufend“ in anderen Zustand
 - **implizit** ----→ wird durch **andere** aus „blockiert“ oder „bereit“ geholt
- Konzeptionell bewirkt der **Planer** (*scheduler*) die Zustandsübergänge, sie definieren verschiedene Phasen der Prozessverarbeitung
 - **scheduling** beim Übergang in die Zustände „bereit“ oder „blockiert“
 - **dispatching** beim Übergang in den Zustand „laufend“



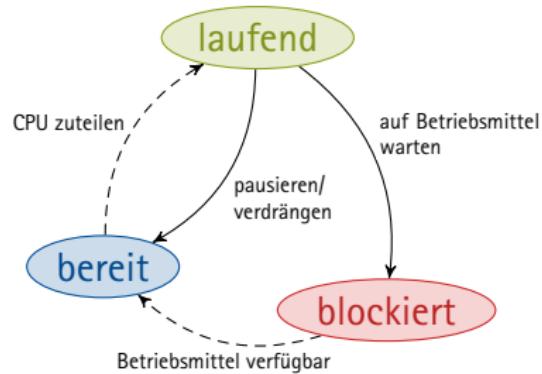
- Je **Rechenkern** (CPU) kann es zu einem Zeitpunkt stets nur einen laufenden, jedoch mehr als einen blockierten oder bereiten Prozess geben.
- Die **Bereitliste** (*ready list*) definiert den **Ablaufplan** (*schedule*) zur Prozessorzuteilung, der zur Laufzeit fortgeschrieben wird.
 - Elemente dieser Liste sind **Prozesskontrollblöcke**
 - Geordnet nach Ankunft, Zeit, Termin, Dringlichkeit, Gewicht, ...
- Allgemein bedeutet **Planung** innerhalb eines Betriebssystems

↪ 5-36



- Während seiner Verarbeitung ist ein Prozess in einem der Zustände:

- **blockiert**, so lange die benötigten Betriebsmittel nicht verfügbar sind.
- **bereit**, wenn die benötigten Betriebsmittel verfügbar sind.
- **laufend**, wenn ihm zusätzlich das Betriebsmittel CPU zugeteilt wurde.



- Weitere Zustände, in denen sich ein Prozess befinden kann:

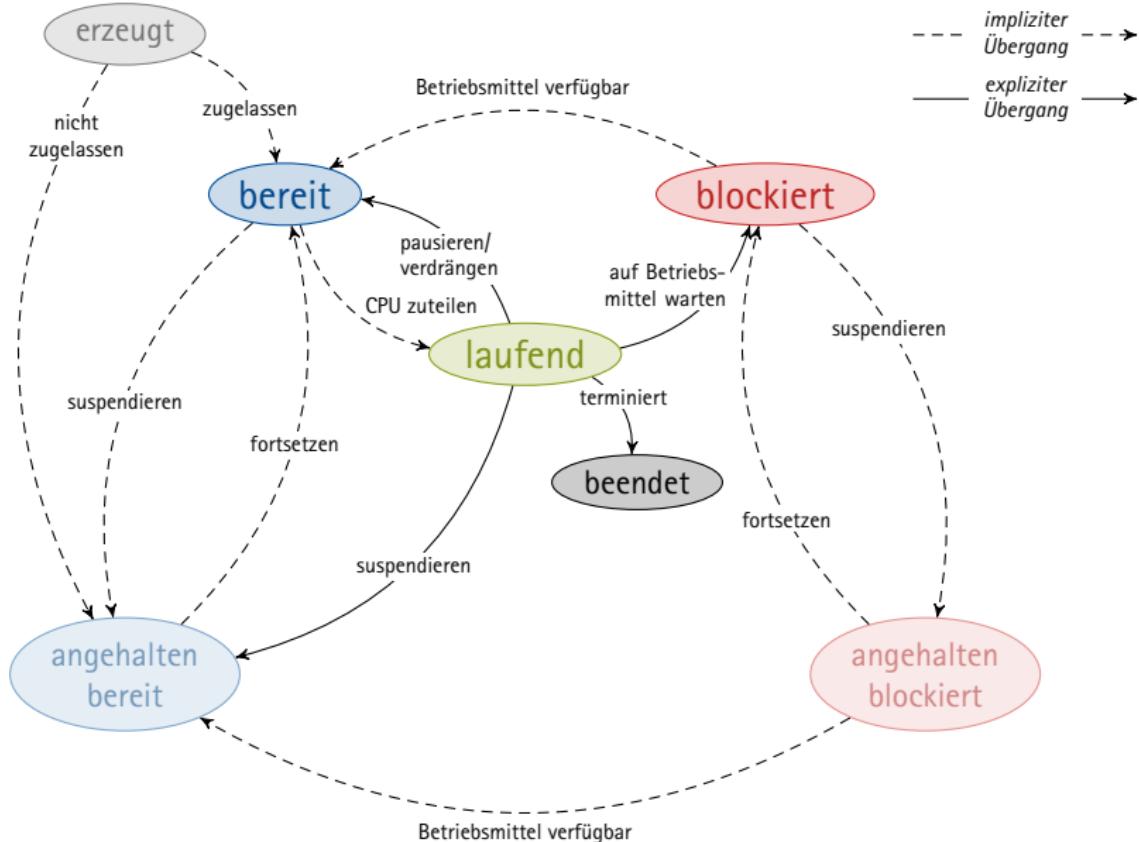
- **erzeugt**, wenn er dem Betriebssystem bekannt ist, aber noch nicht zur Verarbeitung zugelassen wurde.
- **beendet**, wenn er kein Programm mehr ausführt, aber dem Betriebssystem noch weiterhin bekannt sein muss.
- **angehalten**, wenn er aufgrund einer Überlastsituation oder auf Benutzerwunsch temporär von der Verarbeitung ausgeschlossen wurde.

„Zombie“



Erweiterte Prozesszustände

Zustandsübergänge





5.4 Anwendungssicht – POSIX-Prozesse



Prozess ≡ Programmablauf

konzeptioneller Prozessbegriff

Ein **Prozess** ist ein **Programm** in Ausführung **durch** einen **Prozessor**.

Programm ≡ Anweisungsfolge

Ein **Programm** spezifiziert eine Folge von **Anweisungen für** einen **Prozessor**.

Prozess ≡ Ausführungskontext

technischer Prozessbegriff

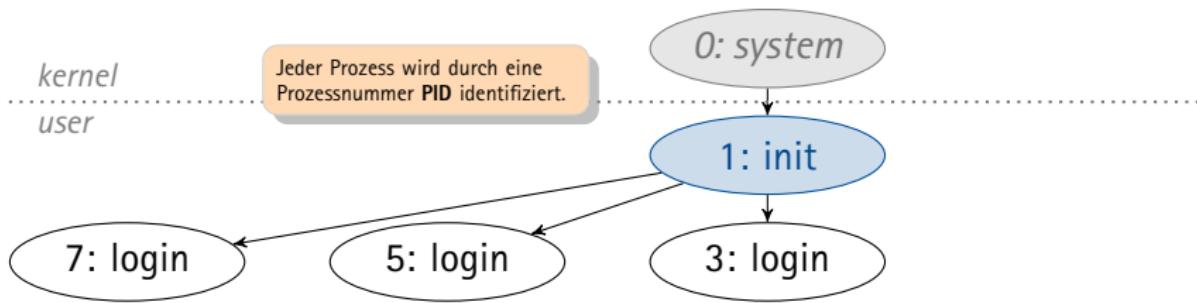
Ein **Prozess** definiert einen **Ausführungskontext** für einen Programmablauf.

- vom BS verwalteter **Ausführungskontext** (CPU, Speicher, Umgebung)
- seit Multics eng verknüpft mit einem **eigenen Adressraum** [23]
- zusammengefasst im **Prozesskontrollblock** (*process control block, PCB*) → 5-36

→ Prozesse sind **Objekte des Betriebssystems**

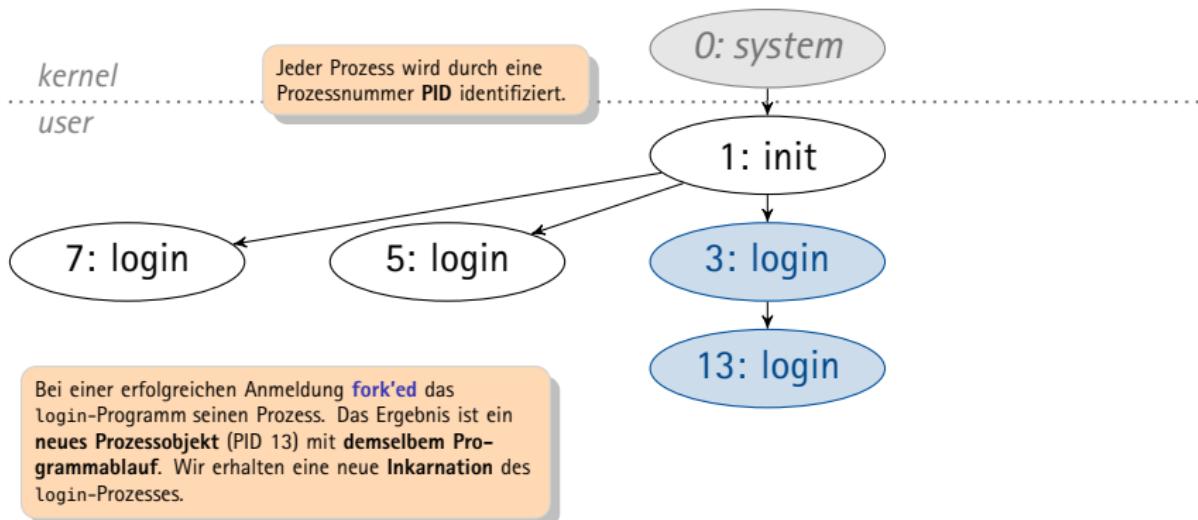
→ Der Prozess-Begriff ist im Betriebssystemkontext **mehrdeutig** :-(
■ steht für das **Prozessobjekt**, den **Programmablauf** oder **beides**
■ oft (aber nicht immer) gibt es eine 1:1-Beziehung zwischen Objekt und Ablauf

- Prozesse sind das **primäre Strukturierungsmittel** für Aktivitäten
 - hierarchisch angeordnet in einer Eltern–Kind-Beziehung
- Ein Kindprozess **entsteht**, indem sich ein bestehender Prozess **selber klon**t
 - **fork** erzeugt ein neues Prozessobjekt mit identischem Programmablauf
 - **exec** ersetzt den Programmablauf im eigenen Prozessobjekt

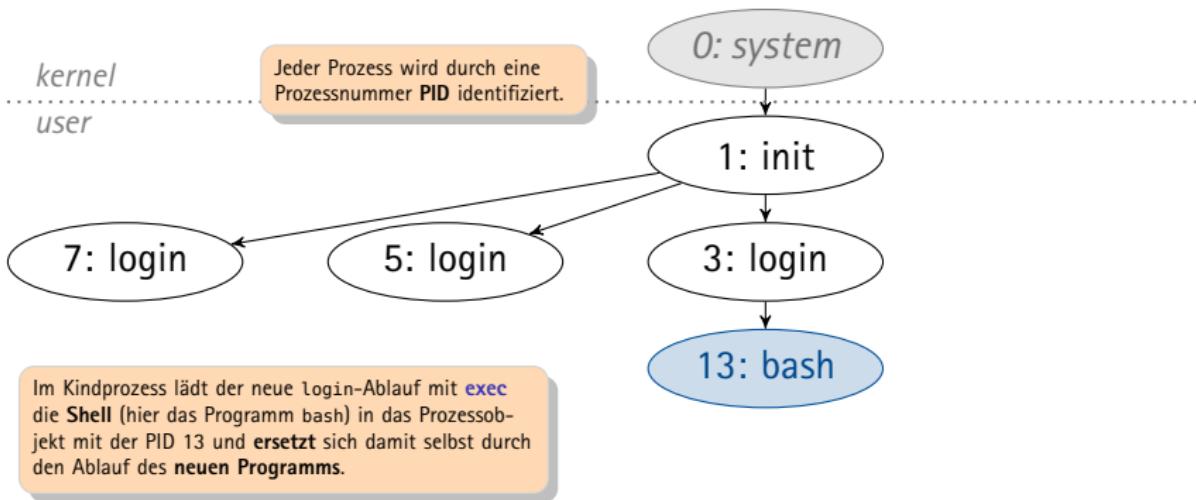


init ist das erste "echte" Prozessobjekt und hat die PID 1. Das init-Programm (z.B. systemd) liest seine Konfiguration und erzeugt für jedes Terminal einen **Kindprozess**, der über login die Einwahl erlaubt.

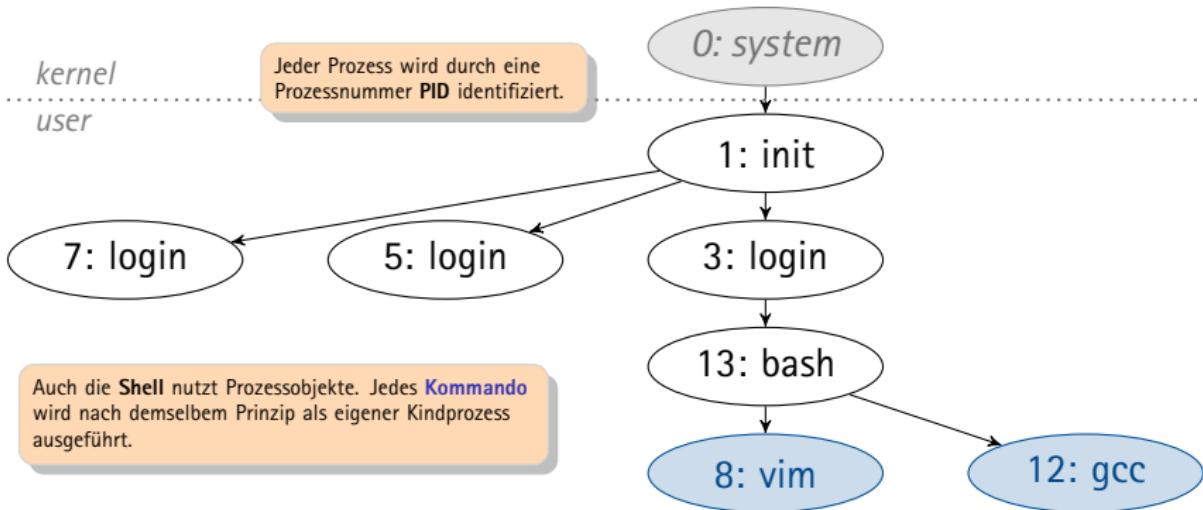
- Prozesse sind das **primäre Strukturierungsmittel** für Aktivitäten
 - hierarchisch angeordnet in einer Eltern–Kind-Beziehung
- Ein Kindprozess **entsteht**, indem sich ein bestehender Prozess **selber klon**t
 - **fork** erzeugt ein neues Prozessobjekt mit identischem Programmablauf
 - **exec** ersetzt den Programmablauf im eigenen Prozessobjekt



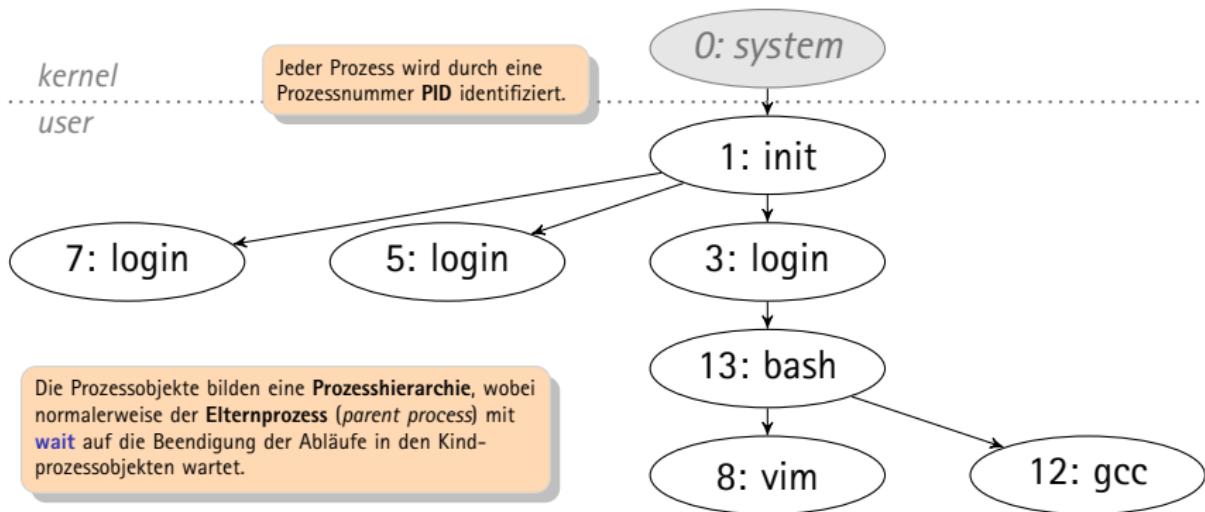
- Prozesse sind das **primäre Strukturierungsmittel** für Aktivitäten
 - hierarchisch angeordnet in einer Eltern–Kind-Beziehung
- Ein Kindprozess **entsteht**, indem sich ein bestehender Prozess **selber klon**t
 - **fork** erzeugt ein neues Prozessobjekt mit identischem Programmablauf
 - **exec** ersetzt den Programmablauf im eigenen Prozessobjekt



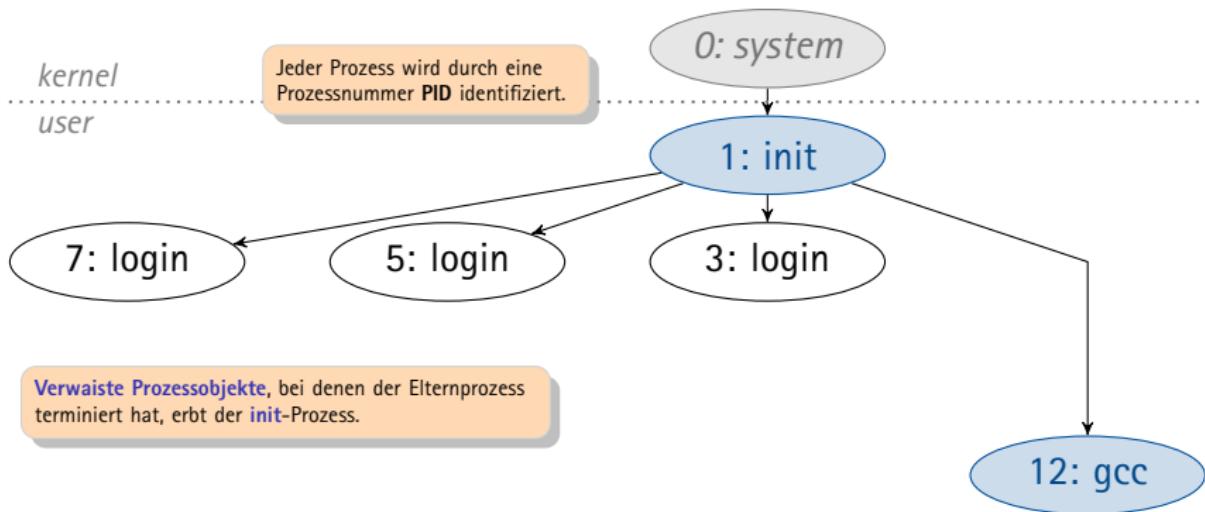
- Prozesse sind das **primäre Strukturierungsmittel** für Aktivitäten
 - hierarchisch angeordnet in einer Eltern–Kind-Beziehung
- Ein Kindprozess **entsteht**, indem sich ein bestehender Prozess **selber klon**t
 - **fork** erzeugt ein neues Prozessobjekt mit identischem Programmablauf
 - **exec** ersetzt den Programmablauf im eigenen Prozessobjekt



- Prozesse sind das **primäre Strukturierungsmittel** für Aktivitäten
 - hierarchisch angeordnet in einer Eltern–Kind-Beziehung
- Ein Kindprozess **entsteht**, indem sich ein bestehender Prozess **selber klon**t
 - **fork** erzeugt ein neues Prozessobjekt mit identischem Programmablauf
 - **exec** ersetzt den Programmablauf im eigenen Prozessobjekt



- Prozesse sind das **primäre Strukturierungsmittel** für Aktivitäten
 - hierarchisch angeordnet in einer Eltern–Kind-Beziehung
- Ein Kindprozess **entsteht**, indem sich ein bestehender Prozess **selber klon**t
 - **fork** erzeugt ein neues Prozessobjekt mit identischem Programmablauf
 - **exec** ersetzt den Programmablauf im eigenen Prozessobjekt



■ Prozessobjekt erzeugen und freigeben: fork⁽²⁾, exit⁽³⁾, kill⁽²⁾, wait⁽²⁾

```
pid_t fork(void);
```

erzeugt ein neues Prozessobjekt durch Verdopplung des aufrufenden Prozesses. Der neue *Kindprozess* ist eine Kopie des *Elternprozesses* mit eigener PID und eigenem Adressraum.

Beide kehren (pseudo-)parallel aus fork⁽²⁾ zurück mit dem Rückgabewert 0 im Kind und der PID des Kindprozesses im Elter.

```
void exit(int status);
```

beendet den aufrufenden Prozess „von innen“ mit dem Rückgabewert **status** und markiert das Prozessobjekt zur Freigabe. Die mit dem Prozessobjekt verbundenen Ressourcen werden freigegeben, das Prozessobjekt selber, sobald sein Rückgabewert mit wait⁽²⁾ vom Elternprozess abgeholt wurde.

```
int kill(pid_t pid, int
        ↴ sig);
```

beendet den Prozess **pid** „von außen“. Mit **sig == SIGTERM** wird der Prozess aufgefordert, sich zu beenden; mit **sig == SIGKILL** wird er zwangsläufig beendet.

```
pid_t wait(int *status);
```

wartet auf die Beendigung eines Kindprozesses, speichert dessen exit⁽³⁾-status in **status**, und gibt das damit verbundene Prozessobjekt endgültig frei. Rückgabewert ist die PID des Kindprozesses. Das Prozessobjekt eines beendeten Prozesses, bleibt als **Zombie** bestehen, bis der Rückgabewert vom Elternprozess mit wait⁽²⁾ „eingesammelt“ wurde.

Beispiel: fork⁽²⁾, wait⁽²⁾, exit⁽³⁾

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/wait.h>
4
5 int main () {
6     pid_t pid;
7     pid = fork(); /* Prozess wird dupliziert!
8                           Beide laufen an dieser Stelle weiter. */
9     if (pid > 0) {
10        printf("Elter: Warte auf Kind %d\n", pid);
11        int status;
12        if (wait(&status) == pid && WIFEXITED(status))
13            printf ("Elter: Exit Status des Kindes: %d\n", WEXITSTATUS(status));
14        else
15            printf ("Elter: Kind nicht normal beendet :-(\n");
16    }
17    else if (pid == 0) {
18        printf(" Kind: Beende mich in 10 Sekunden mit 42\n");
19        sleep(10);
20        printf(" Kind : Jetzt :-\n");
21        exit(42);
22    }
23    else
24        printf("Oh, ein Fehler!\n");
25 }
```

■ Neues Programm ausführen: exec⁽³⁾-Familie

```
int execlp(const char  
    ↴ *filename, const  
    ↴ char *arg, ...);
```

ersetzt im eigenen Prozessobjekt das aufrufende Programm und seine Ausführung durch das Programm **filename**, das neu ausgeführt wird. execve⁽²⁾ kehrt nur im Fehlerfall zum Aufrufer zurück; die neue Programmausführung erbt das Prozessobjekt inkl. PID und geöffneten Dateien → 4-33.

■ Das neue Programm übernimmt die **Dateideskriptoren** des Prozessobjektes

- Darunter die Deskriptoren 1–3 für die Standard{ein|aus|fehleraus}gabe (**stdin** := 0 | **stdout** := 1 | **stderr** := 2)
 - **stdin** zum Lesen von Benutzereingaben (Tastatur, wenn Terminal-Fenster aktiv)
 - **stdout** für Textausgaben des Programms (Terminal-Fenster)
 - **stderr** seperater Kanal für Fehlerausgaben des Progamm (Terminal-Fenster)
- Vor dem Aufruf von exec⁽³⁾ kann so die Umgebung angepasst werden

Auf diese Weise implementiert beispielsweise die UNIX-Shell die E/A-Umlenkung mit **<**, **>**, und **|**

Beispiel: fork⁽²⁾, exec⁽³⁾

Eine Mini-Shell

```
1 // ... includes ...
2 int main () {
3     pid_t pid;
4     char cmd[100], arg[100];
5     while (1) {
6         printf ("Kommando?\n");
7         scanf ("%99s %99s", cmd, arg);
8         pid = fork(); /* Prozess wird dupliziert!
9                         Beide laufen an dieser Stelle weiter. */
10        if (pid > 0) {
11            printf("Elter: Warte auf Kind %d\n", pid);
12            int status;
13            if (wait(&status) == pid && WIFEXITED(status))
14                printf ("Elter: Exit Status des Kindes: %d\n", WEXITSTATUS(status));
15            else
16                printf ("Elter: Kind nicht normal beendet :-(\n");
17        }
18        else if (pid == 0) {
19            printf(" Kind: Starte das Kommando\n");
20            execvp(cmd, cmd, arg, NULL);
21            printf (" Kind: exec fehlgeschlagen\n");
22            exit(ENOENT); // "Datei nicht gefunden"
23        }
24        else
25            printf("Oh, ein Fehler!\n");
26    }
27 }
```

- Die Trennung zwischen `fork()` und `exec()` ist eine **UNIX-Besonderheit** (Trennung zwischen Prozess**erzeugung** und Programmaus**führung**)
 - andere Betriebssysteme verwenden eine `forkexec()`-artige Primitive
 - Beispiel Windows: `CreateProcess()` erzeugt einen Kindprozess, der direkt ein angegebenes Programm ausführt
- **Vorteil:** Ausführendes Programm kann Umgebung im **Kindprozess** anpassen
 - Rechte anpassen, Standardein-/ausgabe vor dem `exec` umleiten, ...
 - Kompakte, flexible Schnittstelle: fork⁽²⁾ und execve⁽²⁾ in Summe **3 Parameter**
 - Vergleich Windows: `CreateProcess()` bekommt **mehr als 25 Parameter!**

```
BOOL CreateProcessA( 10 Parameter...
    LPCSTR           lpApplicationName,
    LPSTR            lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL             bInheritHandles,
    DWORD            dwCreationFlags,
    LPVOID           lpEnvironment,
    LPCSTR           lpCurrentDirectory,
    LPSTARTUPINFOA   lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

... + 17 weitere ausgelagert in diese Struktur!

```
// environment for new process
typedef struct _STARTUPINFOA {
    ...
    DWORD  dwFlags;           // 10 further fields
    WORD   wShowWindow;
    WORD   cbReserved2;
    LPBYTE lpReserved2;
    HANDLE hStdInput;        // stdin
    HANDLE hStdOutput;       // stdout
    HANDLE hStdError;        // stderr
} STARTUPINFOA, *LPSTARTUPINFOA;
```

■ Nachteil: Die Idee des **Klonens** ist für viele Abstraktionen „schwierig“

■ Was tun bei unteilbaren Betriebsmitteln?

↪

5-11

■ Was ist der Zustand konsumierbarer Betriebsmitteln?

■ Was tun bei mehrfädigen Programmen?

↪

5-28

↪ fork⁽²⁾ ist nur bei „sehr einfachen“ Prozesse wohldefiniert!

■ Dokumentation muss sehr viele Sonder-/Spezialfälle beschreiben

“ The received *wisdom suggests that* Unix’s unusual combination of *fork()* and *exec()* for process creation *was an inspired design*. In this paper, we argue that *fork was a clever hack for machines and programs of the 1970s that has long outlived its usefulness and is now a liability.* [...]”

Fork is an anachronism: a relic from another era that is out of place in modern systems where it has a pernicious and detrimental impact. ””

Baumann, Appavoo, Krieger u. a. 2019: „A Fork() in the Road“ [3]

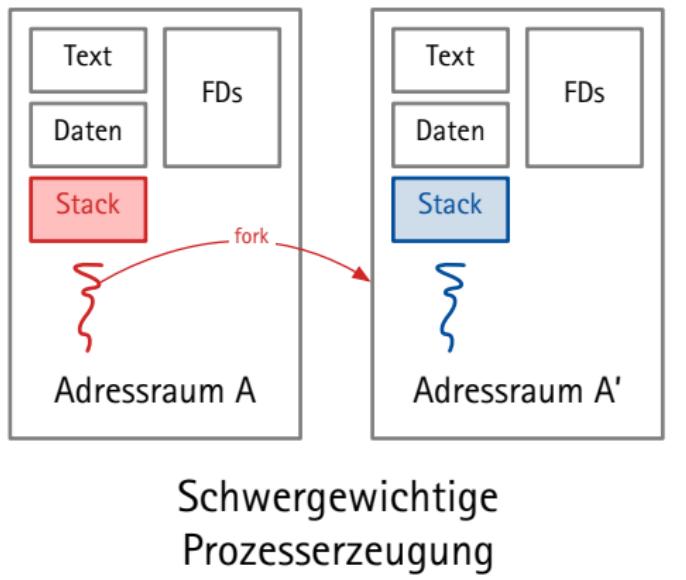


5.5 Systemsicht – Prozesse, Fäden (*Threads*) und Fasern (*Fibers*)

- UNIX-Prozesse sind **schwergewichtige** Ausführungskontexte
 - eigener Adressraum, eigene Dateideskriptortabelle, Befähigungen, Signale...
 - Segmente für Text (*code*), Daten (*data*) und zwei Stapel (*user/kernel-stack*)
 - CPU-Zustand der aktuellen Ausführung (bei verdrängten Prozessen)
- Multiplexing und Virtualisierung des „**vollständigen Computers**“
 - entsprechend **aufwändig** sind Erzeugung, Ein- und Auslastung von Prozessen
 - **ungünstig für parallele Abläufe** innerhalb eines Programms

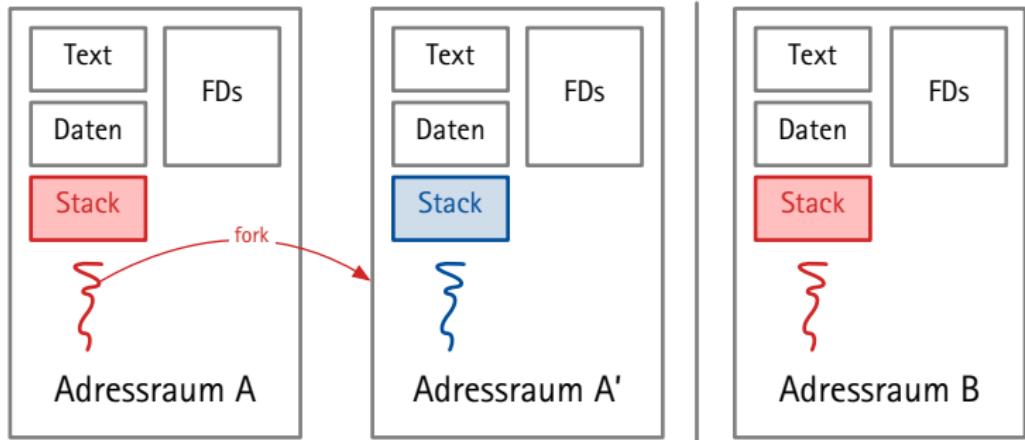
Das **Gewicht** eines Prozessobjekts ist ein bildlicher Ausdruck für die Größe seines Kontextes – und damit der Zeit und der Menge an Betriebsmitteln, die benötigt werden, einen Prozess zu erzeugen, einzulasten, auszulasten und zu zerstören. „Teuer“ sind diesbezüglich insbesondere alle Operationen, die Adressräume betreffen.

Vergleich: Erzeugung und Ausführung paralleler Abläufe



- UNIX-Prozesse sind **schwergewichtige** Ausführungskontexte
 - eigener Adressraum, eigene Dateideskriptortabelle, Befähigungen, Signale...
 - Segmente für Text (*code*), Daten (*data*) und zwei Stapel (*user/kernel-stack*)
 - CPU-Zustand der aktuellen Ausführung (bei verdrängten Prozessen)
- Multiplexing und Virtualisierung des „**vollständigen Computers**“
 - entsprechend **aufwändig** sind Erzeugung, Ein- und Auslastung von Prozessen
 - **ungünstig für parallele Abläufe** innerhalb eines Programms
- Variante: **Fäden** (*threads*) als **leichtgewichtige(re)** Ausführungskontexte
 - Nur noch eigener CPU-Zustand und eigene Stapel (*user/kernel-stack*)
 - Adressraum, Dateideskriptoren, Segmente, ... wird mit anderen Fäden **geteilt**
- Multiplexing und Virtualisierung (**nur** der **CPU**)
 - Erzeugung, Ein- und Auslastung von Fäden ist **deutlich weniger aufwändig**
 - einfachere **Verwendung echter Parallelität** innerhalb eines Programms

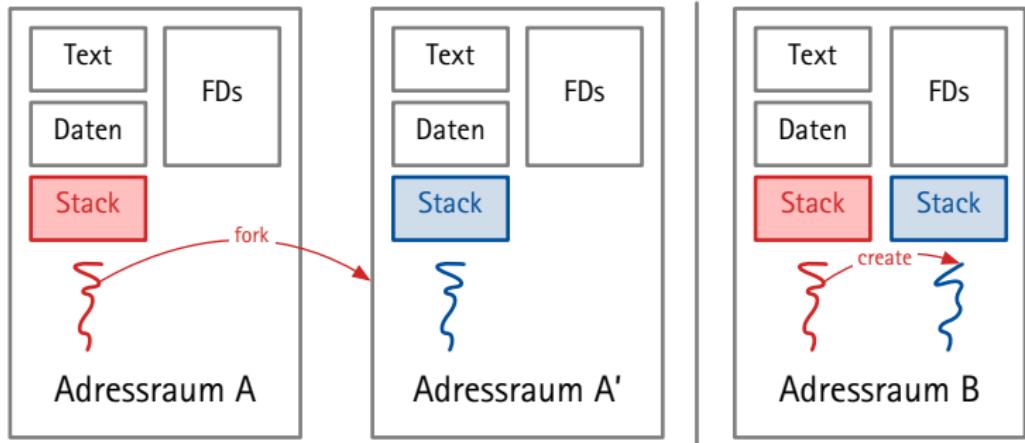
Vergleich: Erzeugung und Ausführung paralleler Abläufe



Schwergewichtige
Prozesserzeugung

Leichtgewichtige
Prozesserzeugung

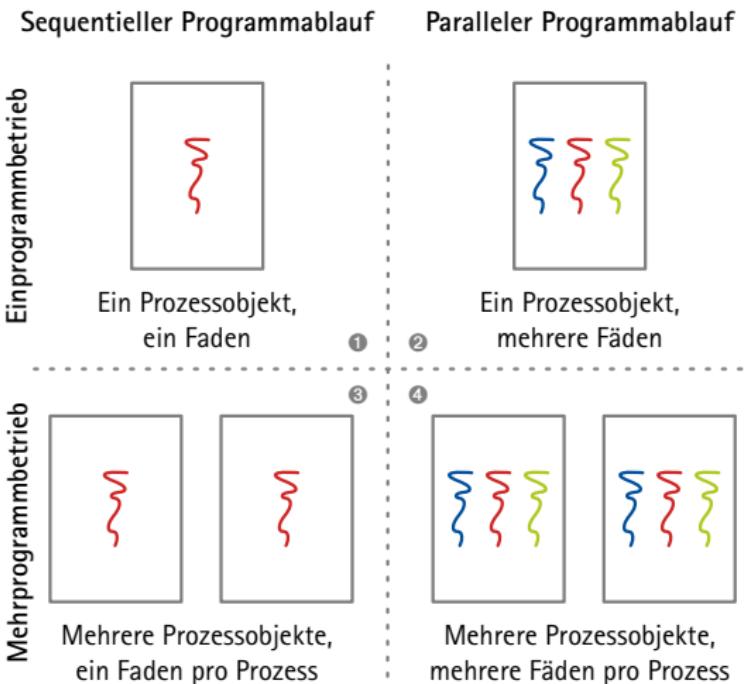
Vergleich: Erzeugung und Ausführung paralleler Abläufe



Schwergewichtige
Prozesserzeugung

Leichtgewichtige
Prozesserzeugung

- Fäden ermöglichen eine effiziente(re) Parallelisierung von Programmen



Der klassische UNIX-Prozess entspricht Fall ③ und ermöglicht (nur) die prozessbasierte Parallelisierung von Programmen. Modernere Betriebssysteme (wie Linux) unterstützen Fall ④.

Abbildung angelehnt an [31]

■ **Prozess**, auch **schwergewichtiger Prozess**, hier Prozessobjekt

- **BS-verwalteter** Ausführungskontext für ein Programm
- „virtueller Computer“, virtueller Adressraum/Geräte/CPUs
- gemeinsam benutzt von einem oder mehreren **Fäden** des Programms

■ **Faden (Thread)**, auch **leichtgewichtiger Prozess**

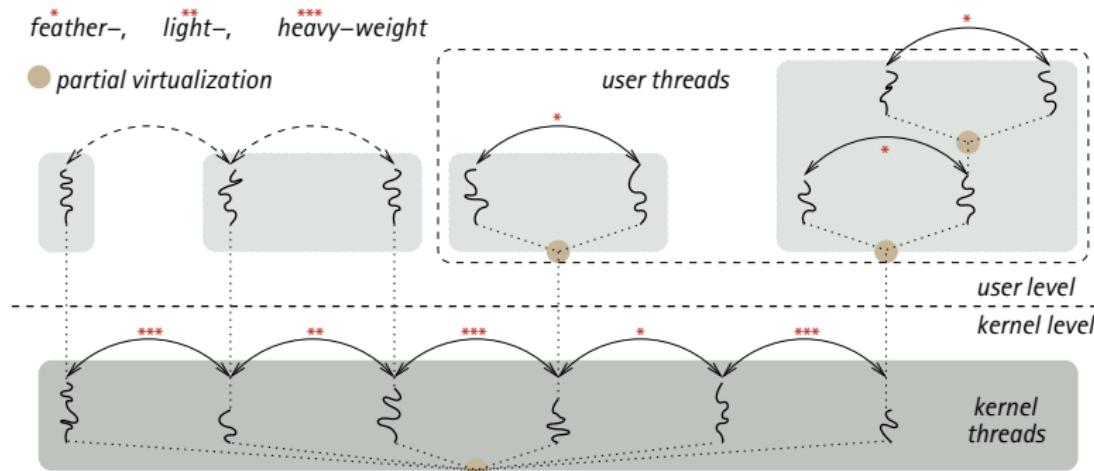
- vom **BS verwalteter** CPU-Kontext für einen Programmstrang, manifest im Anwendungs- wie Kernadressraum $\mapsto \text{kernel thread}$
- virtualisiert die CPU auf der Maschinenebene L3, „virtueller Prozessor“
- ausgeführt in einem konkreten Prozessadressraum

■ **Faser (Fiber)**, auch **federgewichtiger Prozess**

- von der **Anwendung verwalteter** CPU-Kontext für einen Programmstrang manifest nur im Anwendungsadressraum $\mapsto \text{user thread}$
- virtualisiert die virtuelle CPU (\mapsto Faden) oberhalb der Maschinenebene E_3 , \leadsto **dem BS nicht bekannt**
- ausgeführt in einem konkreten Faden

Gewichtsklassen von Prozessobjekten

Kontextwechselaufwand



- *** Schwergewichtiger Wechsel: $\text{CPU}_{\text{user}} + \text{CPU}_{\text{kernel}} + \text{Adressraum}$
- ** Leichtgewichtiger Wechsel: $\text{CPU}_{\text{user}} + \text{CPU}_{\text{kernel}}$
- * Federgewichtiger Wechsel: $\text{CPU}_{\text{user/kernel}}$
- Partielle Virtualisierung der CPU ist auf **jeder Ebene** möglich.
- Oberhalb von E_3 entzieht sich diese jedoch der **Kenntnis des BS**.



5.6 Systemsicht – Prozesskontrollblöcke

- Der **Prozesskontrollblock** (*process control block*, PCB) bündelt alle zur partiellen Virtualisierung relevanten Attribute eines Prozesses
 - Systemobjekt, in dem typischerweise folgende Daten verbucht sind:
 - Adressraum, Speicherbelegung, Laufzeitkontext, ..., **geöffnete Dateien**
 - Verarbeitungszustand, Blockierungsgrund, Dringlichkeit, Termin
 - Name, Domäne, Zugehörigkeit, Befähigung, Zugriffsrechte, Identifikationen
 - zentrale **Informations- und Kontrollstruktur** im Betriebssystem
- Pro Prozessor verwaltet das Betriebssystem einen **Prozesszeiger**
 - identifiziert das aktuell **laufende** Prozessobjekt → 5-14
 - so, wie der Befehlszähler der CPU den laufenden Befehl adressiert, zeigt der Prozesszeiger des Betriebssystems auf den gegenwärtigen Prozess
 - beim Prozesswechsel (*dispatch*) wird der Prozesszeiger weitergeschaltet
- Referenziert von außen durch eine **Prozessidentifikation** (PID)
 - oft ein Index in die systemweite interne **Prozesstabelle** (vgl. → 4-33)
 - **Achtung:** Unter UNIX werden PIDs „recycled“, sie sind eindeutig nur so lange das Prozessobjekt existiert!



5.7 Zusammenfassung

- **Prozess:** Ein **Programm in Ausführung** auf einem Prozessor
- **Prozess(objekt):** Ein **Ausführungskontext** für einen Programmablauf, verwaltet (in Form eines **PCB**) durch das BS, systemweit referenziert durch seine **PID**, oft verbunden mit einem eigenen **Adressraum**
- **Faden:** Ein **leichtgewichtiger Ausführungskontext** durch die gemeinsame Nutzung von Ressourcen (insb. Adressraum)
 - In UNIX sind Prozesse das zentrale **Strukturierungsmittel** für Aktivitäten
 - entstehen durch **klonen** mit **fork⁽²⁾** aus einem bestehenden Prozesses
 - können mit **exec⁽³⁾** zur Laufzeit ihr Programm wechseln
 - werden ergänzt durch **leichtgewichtige Prozesse** (Fäden)
 - Partielle Virtualisierung der CPU ist auf jeder Ebene möglich

4 Dateien und Dateisysteme

5 Prozesse und Fäden

6 Unterbrechungen, Ausnahmen, Signale

6.1 Einordnung

6.2 Systemsicht – Teilinterpretation

6.3 Systemsicht – Ausnahmen und Unterbrechungen

6.4 Systemsicht – E_2 -Ausnahmen

6.5 Systemsicht – E_3 -Ausnahmen: Signale

6.6 Anwendungssicht – Signale: Nebenläufigkeit

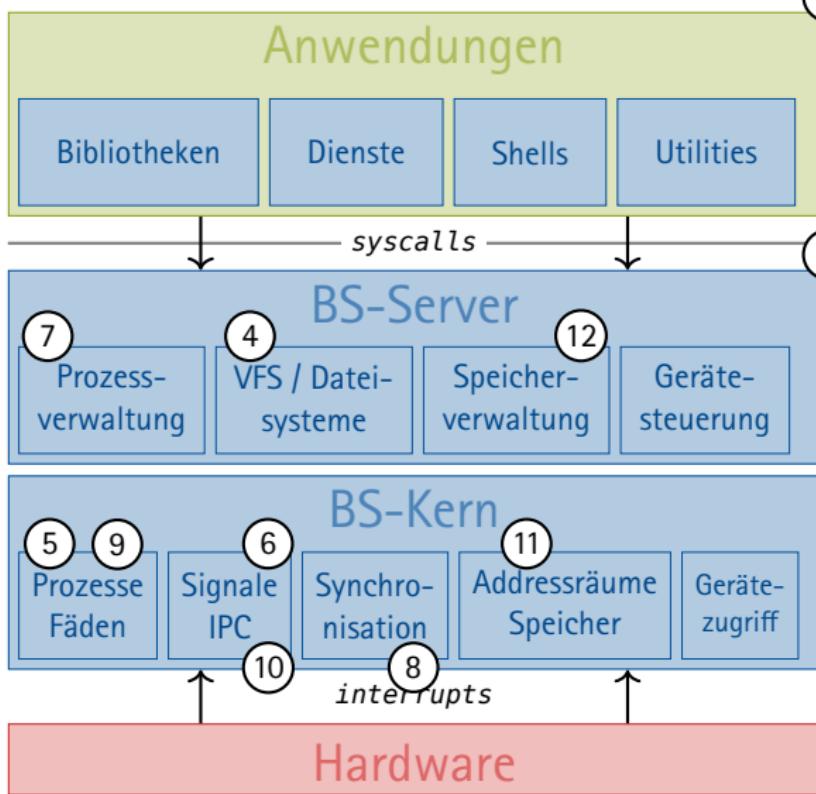
6.7 Zusammenfassung

7 Prozessverwaltung

6.1 Einordnung

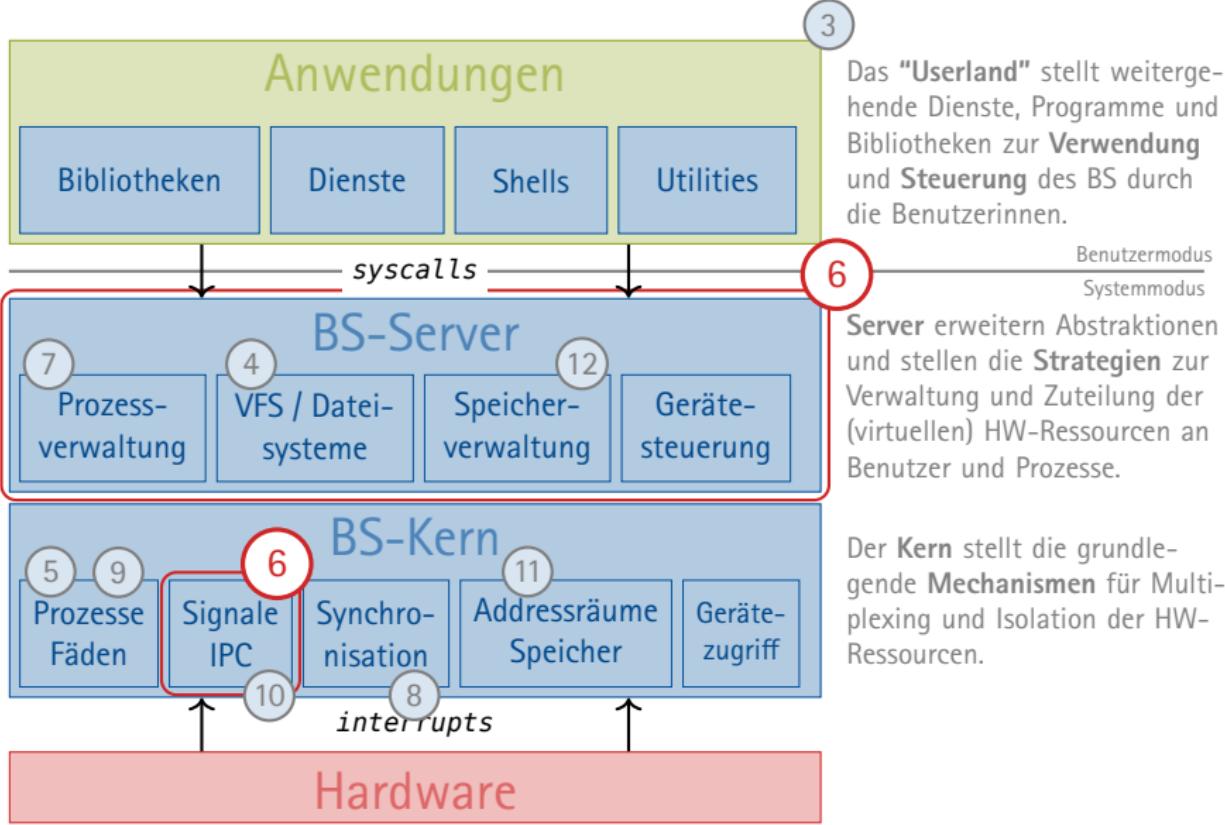


6 Unterbrechungen, Ausnahmen, Signale – Einordnung





6 Unterbrechungen, Ausnahmen, Signale – Einordnung





6 Unterbrechungen, Ausnahmen, Signale – Einordnung

Worum geht es in diesem Kapitel?

- **Verstehen** der verschiedenen Ausnahmearten und ihrer Behandlung durch das Betriebssystem
 - Unterscheidung *syscall, trap, interrupt*
 - CPU-Ausnahmen als Aktivierungssignale für das Betriebssystem
 - Teilinterpretation im Detail
 - Unterdrückung, Verzögerung und Synchronisation
- **Erläutern** der Analogie zwischen UNIX-Signalen und CPU-Ausnahmen
 - Unterscheidung synchrone/asynchrone Signale
 - Interprozesskommunikation mit Signalen
 - Nebenläufigkeit durch Signale

→ Nur durch Ausnahmen behält das Betriebssystem die Kontrolle

Auch hier hängt das Verständnis von der Ebene der Betrachtung ab!

Benutzersicht: Seltene Ereignisse

- „Interrupts are an unpleasant fact of life!“ [35]
- „Die Ausnahme bestätigt die Regel“ (*exceptio probat regulam*) [Cicero]

Anwendungssicht: Signal vom Betriebssystem

Rückruf des Betriebssystems an ein Programm in Ausführung.

- aufgrund eines **synchronen** Ereignisses, dass **unmittelbar** mit der Programmausführung zusammen hängt
- aufgrund eines **asynchronen** Ereignisses, dass **nicht/mittelbar** mit der Programmausführung zusammen hängt



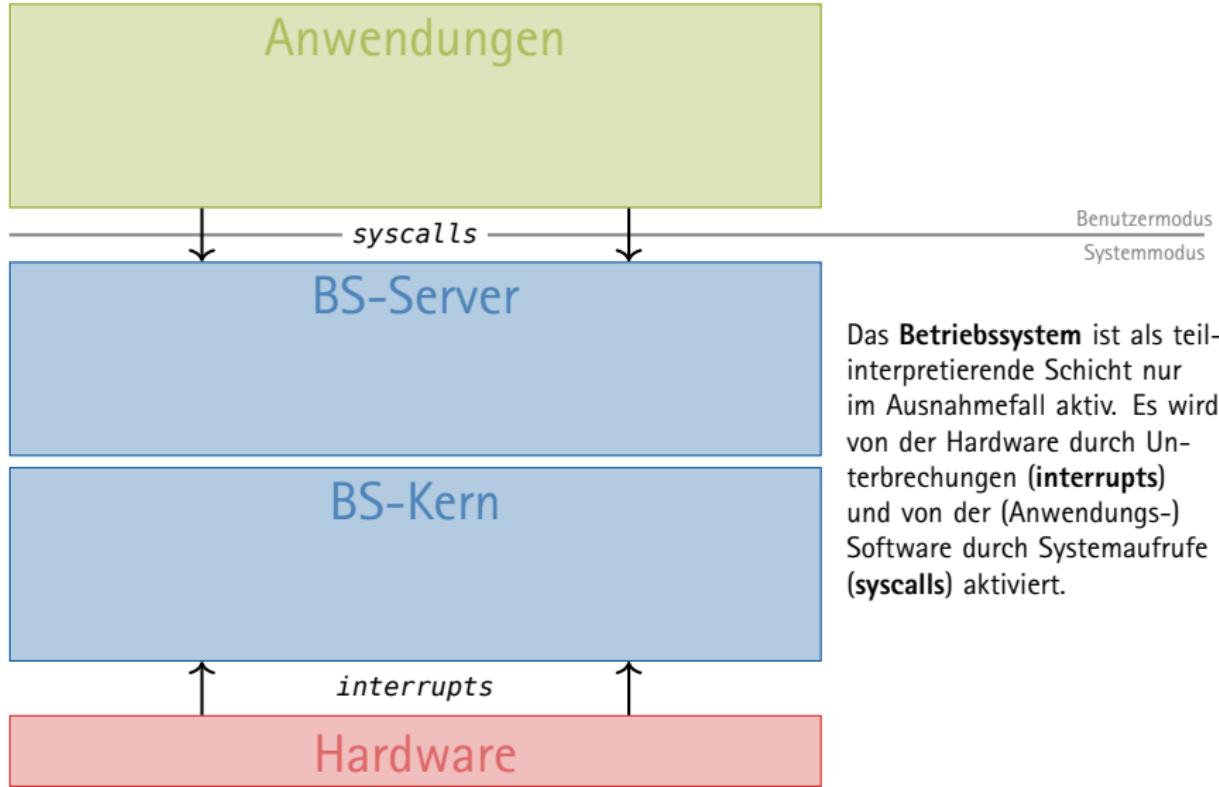
Systemsicht: Aktivierung des Betriebssystems

Aufruf des durch das Betriebssystem realisierten Interpreters

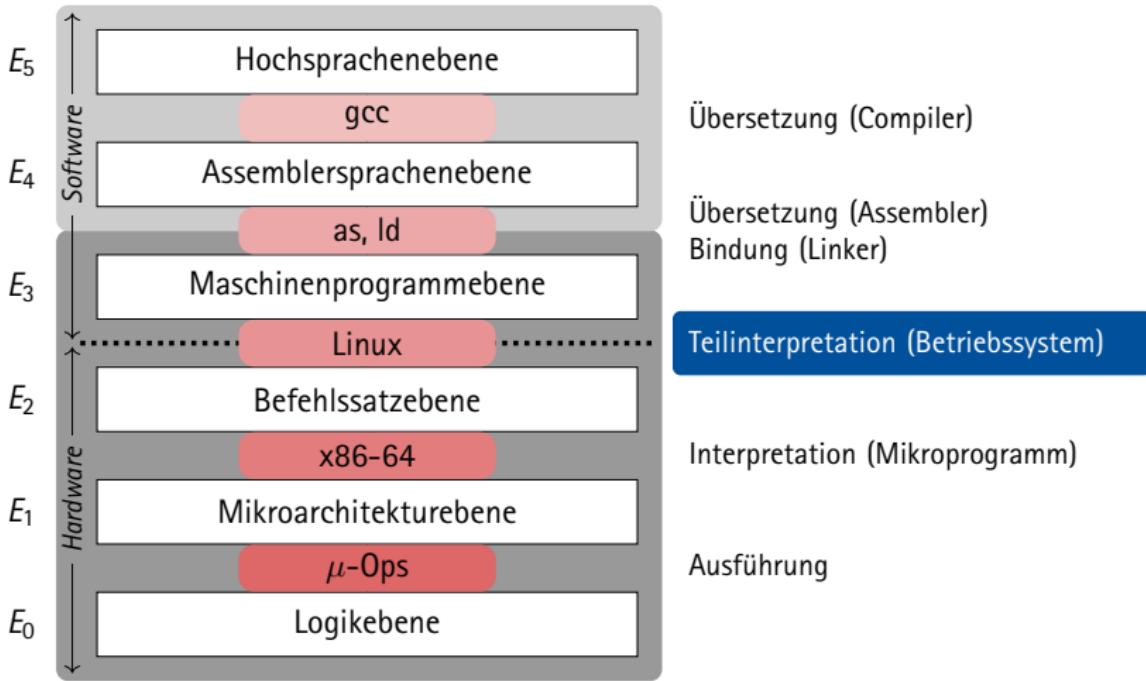
- synchron durch den laufenden Prozess (implizit oder explizit)
- asynchron durch ein Signal der nebenläufigen Hardware

6.2 Systemsicht – Teilinterpretation

⌘ Aktivierung des Betriebssystems durch Ausnahmen



Teilinterpretation durch das Betriebssystem



- Befehle der Maschinenprogrammebene E_3 sind
 - „normale“ Befehle der Befehlssatzebene $E_2 \rightsquigarrow$ werden vom Prozessor ausgeführt
 - **unkritische Befehle**, die in jedem Arbeitsmodus des Prozessors ausführbar sind
 - „unnormale“ Befehle der Befehlssatzebene $E_2 \rightsquigarrow$ werden vom BS ausgeführt
 - **privilegierte Befehle**, die nur im Systemmodus ausführbar sind
- Die „unnormalen“ Befehle stellen Prozesse, Dateien, (Schutz), ... bereit
 - Interpreter dieser E_3 -Befehle ist das Betriebssystem
 - Verwendet (benötigt) zur Umsetzung privilegierte E_2 -Befehle

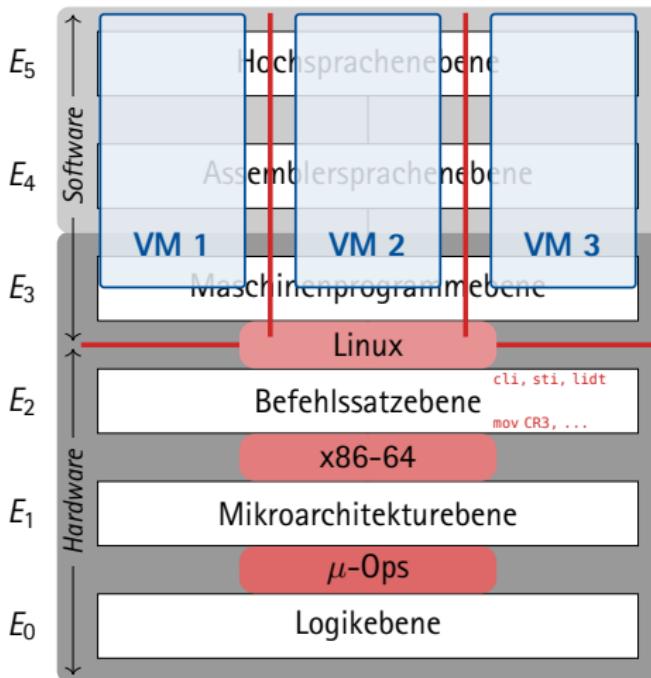
→ Das Betriebssystem ist **immer nur ausnahmsweise aktiv!**

- es muss von **außerhalb aktiviert** werden
 - **explizit**, durch einen **synchronen** Systemaufruf (*syscall*)
 - **implizit**, durch eine **synchrone** Programmunterbrechung (*trap*)
 - **implizit**, durch eine **asynchrone** Programmunterbrechung (*interrupt*)
- In allen Fällen **deaktiviert es sich selbst** (durch Einlastung eines Prozesses)



Teilinterpretation durch das Betriebssystem

Struktur



Teilinterpretation (Betriebssystem)

Vertikale Isolation
(Benutzer-/Systemmodus)
durch Abschirmung der
E₂-Instruktionen für die
horizontale Isolation

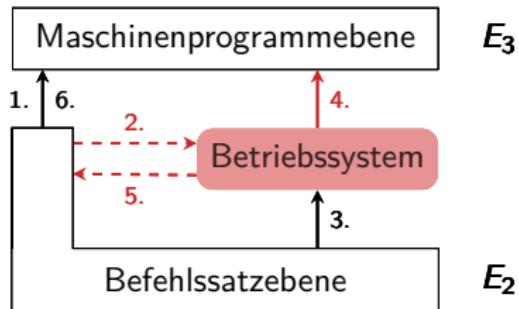


Für den Instruktionssatz auf E₃ gilt: $S_3 = S_2 \setminus S_{2,\text{priv}} \cup \text{Syscalls}$

❖ Teilinterpretation durch das Betriebssystem

Ablauf

1. Die Befehlssatzebene interpretiert das Maschinenprogramm befehlsweise.
2. Sie setzt dessen Ausführung aus, bei
 - Systemaufruf/Ausnahme (synchron)
 - Unterbrechung (asynchron)und **aktiviert** das Betriebssystem
3. Sie interpretiert nun das **Betriebssystemprogramm** befehlsweise.
4. Das ausgeführte **Betriebssystemprogramm** *interpretiert* das unterbrochene Maschinenprogramm befehlsweise.
~~~ Rekursiv – ggfs. interpretiert sich das Betriebssystem selbst partiell!
5. Das **Betriebssystemprogramm** instruiert die Befehlssatzebene, die Ausführung des Maschinenprogramms wieder aufzunehmen.
6. Die Befehlssatzebene interpretiert das Maschinenprogramm befehlsweise.

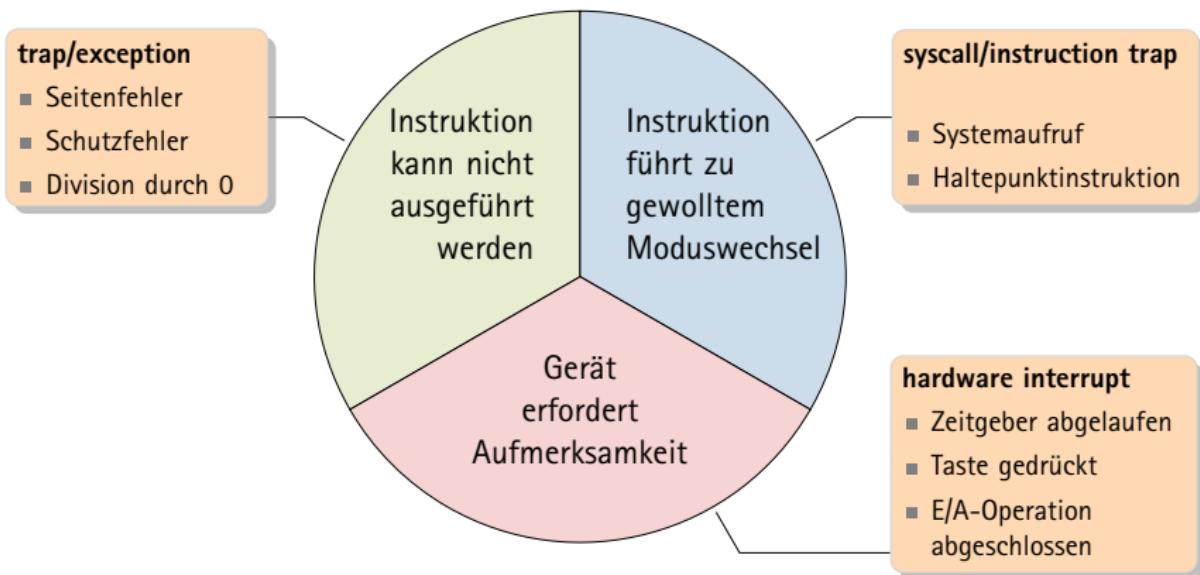


## 6.3 Systemsicht – Ausnahmen und Unterbrechungen



# Ausnahmen und Unterbrechungen

- Grob können drei Fälle von Ausnahmen unterschieden werden





Ausnahmesituationen der Befehlssatzebene  $E_2$  führen immer zu einem Wechsel in den Systemmodus und Aktivierung des Betriebssystems.

**syscall** explizite Aktivierung durch die Anwendung **synchron** zum Kontrollfluss (quasi ein „Prozeduraufruf mit Moduswechsel“)

**trap** implizite Aktivierung durch die CPU **synchron** zum Kontrollfluss, wenn  $E_2$ -Befehl nicht ausführbar ist (Ausnahmebehandlung)

- Division durch 0, Seitenfehler (*page fault*), Schutzverletzung
- privilegierter Befehl im Benutzermodus, unbekannter Befehl
- **nicht unterdrückbar/verzögerbar**  $\rightsquigarrow$  muss behandelt werden

**interrupt** implizite Aktivierung durch die CPU **asynchron** zum Kontrollfluss aufgrund eines externes Ereignisses (von „eigenständiger“ Hardware)

- Zeitscheibe abgelaufen, Netzwerkpaket eingetroffen, Taste gedrückt
- *Inter-Processor-Interrupt (IPI)* durch Programm auf anderem Kern
- **unterdrückbar/verzögerbar** durch **privilegierte  $E_2$ -Operationen**

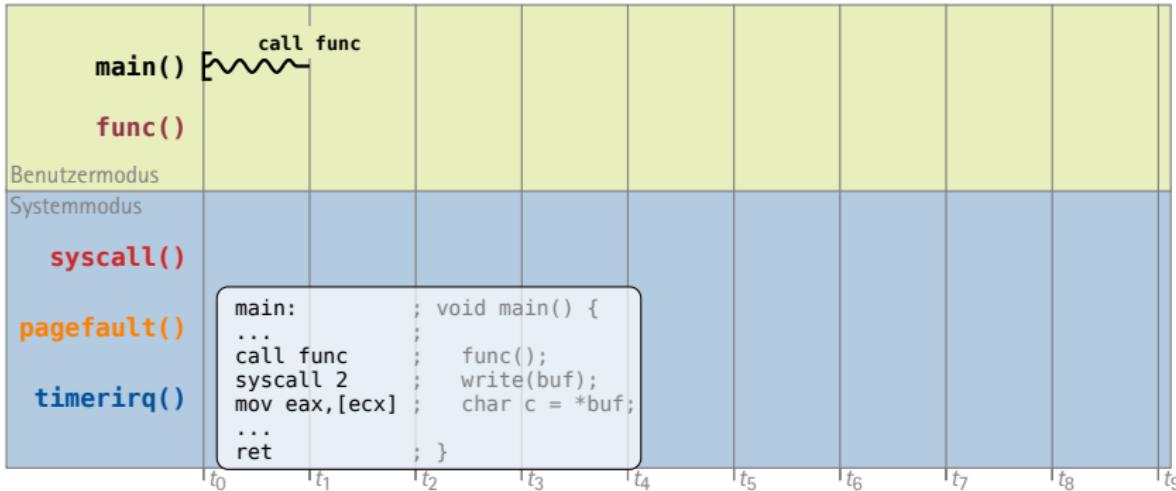
Für die Maschinenprogrammebene  $E_3$  sind **Signale** der analogen Mechanismus  $\hookrightarrow$

6-28



# Ausnahmen und Unterbrechungen

Beispielablauf

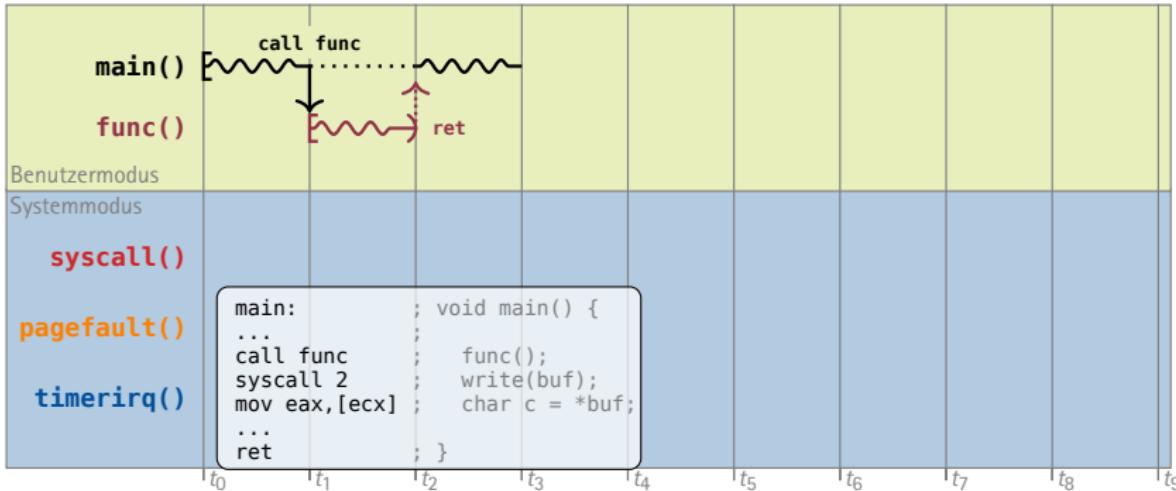


- $t_1$  synchroner, expliziter Prozedurruf, Rücksprung implizit an Stelle nach dem `call`.
- $t_3$  synchroner, expliziter Systemruf, Rücksprung implizit an Stelle nach dem `syscall`.
- $t_5$  synchroner, impliziter Systemruf (Trap), hier: `buf` ausgelagert. Betriebssystem behebt Fehlerursache, Rücksprung implizit an Fehlerstelle  $\leadsto$  Befehl wird wiederholt.
- $t_{??}$  asynchrone, implizite Unterbrechung (Zeitgeber abgelaufen). An beliebiger Stelle möglich, Rücksprung implizit an Stelle der Unterbrechung.



# Ausnahmen und Unterbrechungen

Beispielablauf

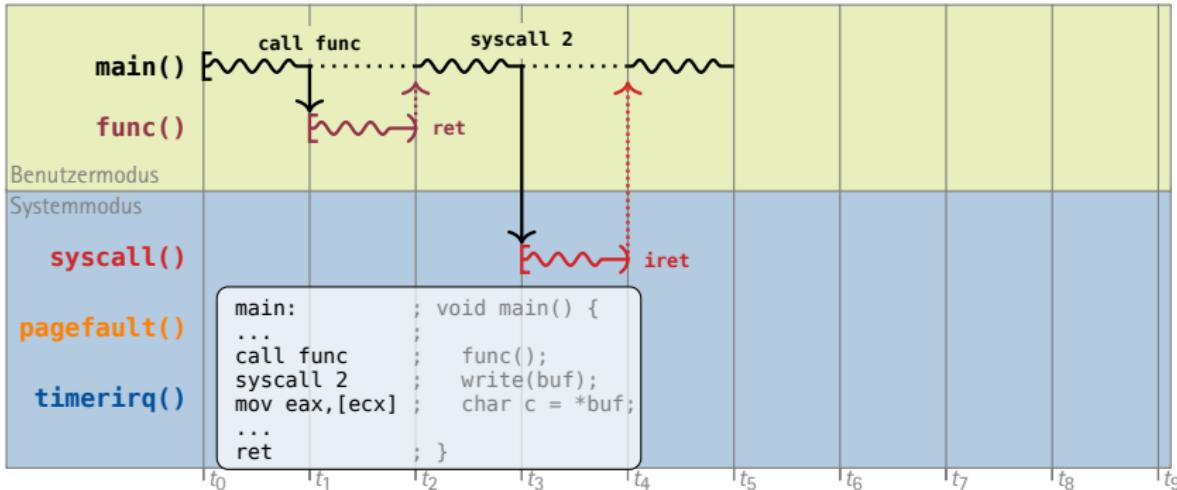


- $t_1$  synchroner, expliziter Prozedurruf, Rücksprung implizit an Stelle nach dem `call`.
- $t_3$  synchroner, expliziter Systemruf, Rücksprung implizit an Stelle nach dem `syscall`.
- $t_5$  synchroner, impliziter Systemruf (Trap), hier: `buf` ausgelagert. Betriebssystem behebt Fehlerursache, Rücksprung implizit an Fehlerstelle  $\rightsquigarrow$  Befehl wird wiederholt.
- $t_{??}$  asynchrone, implizite Unterbrechung (Zeitgeber abgelaufen). An beliebiger Stelle möglich, Rücksprung implizit an Stelle der Unterbrechung.



# Ausnahmen und Unterbrechungen

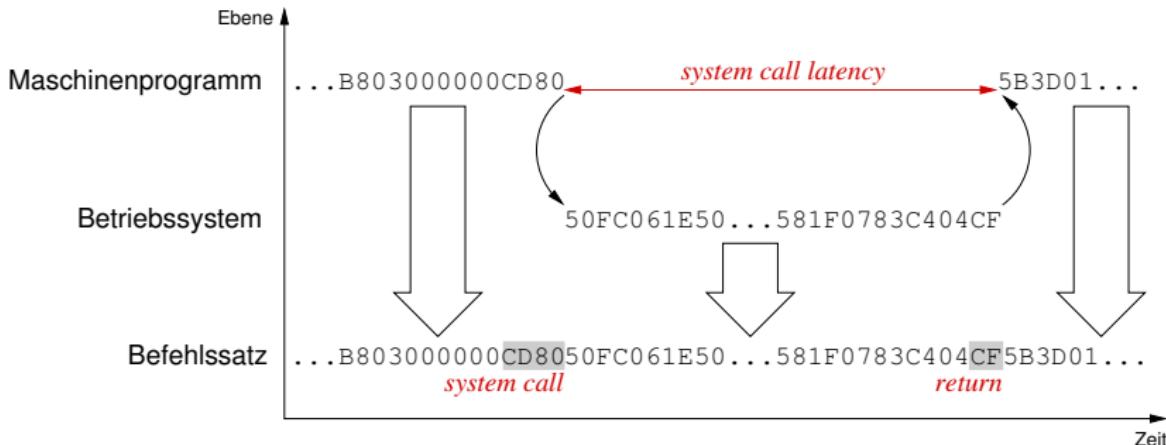
Beispielablauf



- $t_1$  synchroner, expliziter Prozedurruf, Rücksprung implizit an Stelle nach dem `call`.
- $t_3$  synchroner, expliziter Systemruf, Rücksprung implizit an Stelle nach dem `syscall`.
- $t_5$  synchroner, impliziter Systemruf (Trap), hier: `buf` ausgelagert. Betriebssystem behebt Fehlerursache, Rücksprung implizit an Fehlerstelle  $\leadsto$  Befehl wird wiederholt.
- $t_{??}$  asynchrone, implizite Unterbrechung (Zeitgeber abgelaufen). An beliebiger Stelle möglich, Rücksprung implizit an Stelle der Unterbrechung.

# Ablauf eines Systemrufs

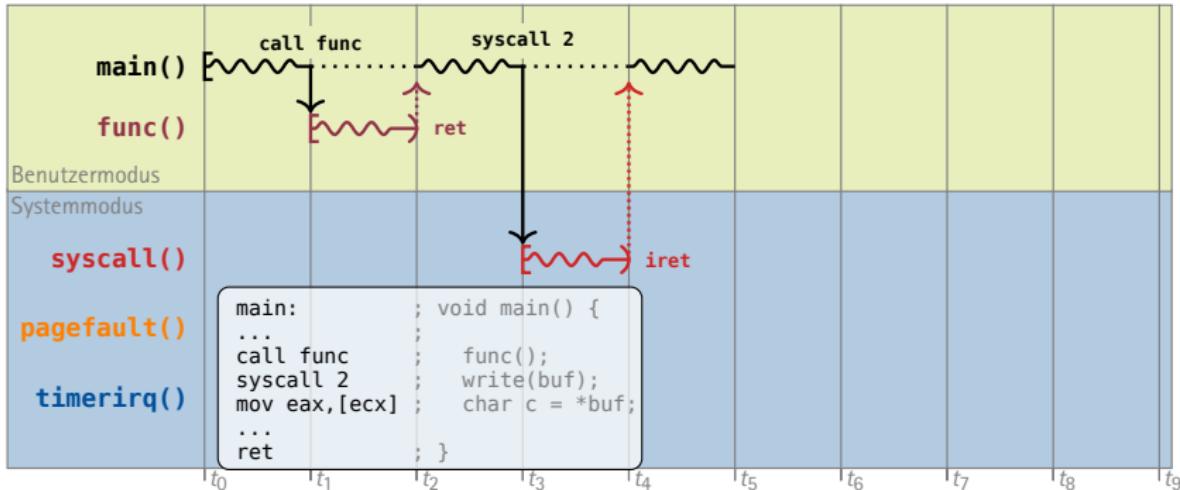
syscall bei IA-32: int-Instruktion



1. Ausführung eines Maschinenprogramms (Instruktionsstrom)
2. Auslösung eines Systemaufrufs durch den Prozessor
3. Verzweigung zum Betriebssystem
4. Behandlung des Systemaufrufs durch das Betriebssystem
5. Beendigung des Systemaufrufs
6. Fortsetzung des Maschinenprogramm

# Ausnahmen und Unterbrechungen

Beispielablauf

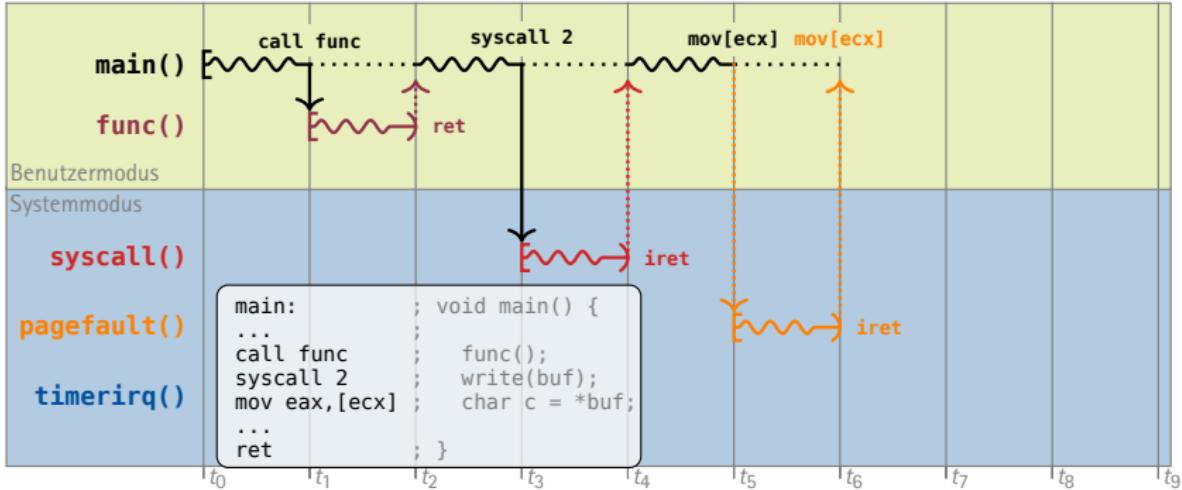


- $t_1$  synchroner, expliziter Prozeduraufruf, Rücksprung implizit an Stelle nach dem `call`.
- $t_3$  synchroner, expliziter Systemruf, Rücksprung implizit an Stelle nach dem `syscall`.
- $t_5$  synchroner, impliziter Systemruf (Trap), hier: `buf` ausgelagert. Betriebssystem behebt Fehlerursache, Rücksprung implizit an Fehlerstelle  $\leadsto$  Befehl wird wiederholt.
- $t_{??}$  asynchrone, implizite Unterbrechung (Zeitgeber abgelaufen). An beliebiger Stelle möglich, Rücksprung implizit an Stelle der Unterbrechung.



# Ausnahmen und Unterbrechungen

Beispielablauf

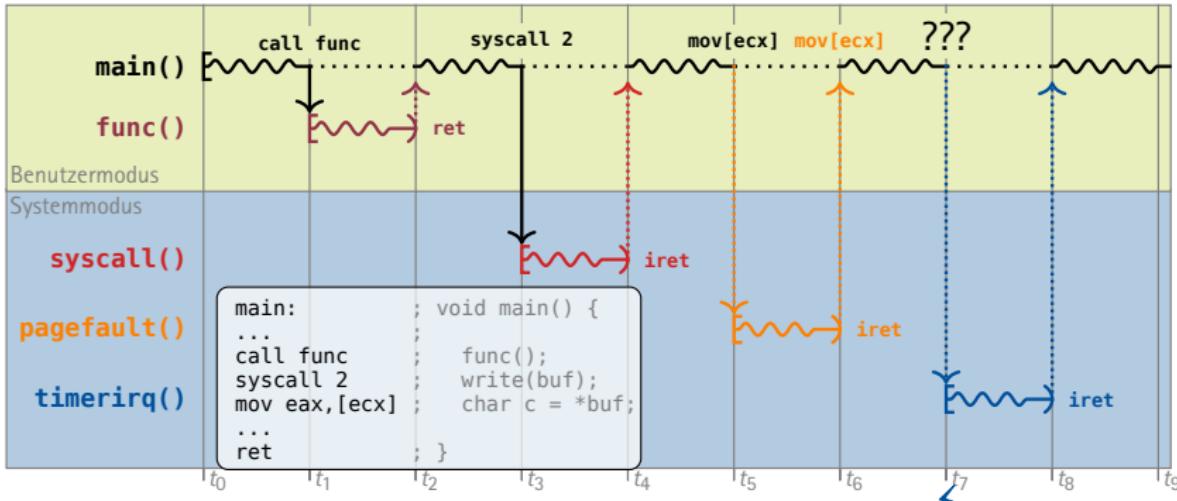


- $t_1$  synchroner, expliziter Prozedurruff, Rücksprung implizit an Stelle nach dem `call`.
- $t_3$  synchroner, expliziter Systemruff, Rücksprung implizit an Stelle nach dem `syscall`.
- $t_5$  synchroner, impliziter Systemruff (Trap), hier: `buf` ausgelagert. Betriebssystem behebt Fehlerursache, Rücksprung implizit an Fehlerstelle  $\leadsto$  Befehl wird wiederholt.
- $t_{??}$  asynchrone, implizite Unterbrechung (Zeitgeber abgelaufen). An beliebiger Stelle möglich, Rücksprung implizit an Stelle der Unterbrechung.



# Ausnahmen und Unterbrechungen

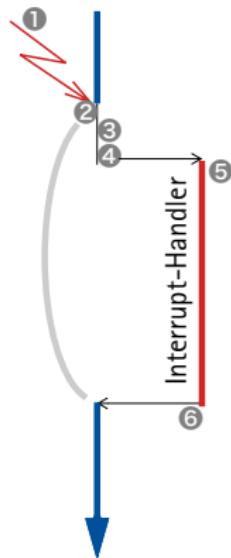
Beispielablauf



- $t_1$  synchroner, expliziter Prozedurruff, Rücksprung implizit an Stelle nach dem `call`.
- $t_3$  synchroner, expliziter Systemruff, Rücksprung implizit an Stelle nach dem `syscall`.
- $t_5$  synchroner, impliziter Systemruff (Trap), hier: `buf` ausgelagert. Betriebssystem behebt Fehlerursache, Rücksprung implizit an Fehlerstelle  $\leadsto$  Befehl wird wiederholt.
- $t_{??}$  asynchrone, implizite Unterbrechung (Zeitgeber abgelaufen). An beliebiger Stelle möglich, Rücksprung implizit an Stelle der Unterbrechung.

# Ablauf einer Unterbrechung (*Interrupt*)

- ① Gerät signalisiert Interrupt
  - Anwendungsprogramm wird „unmittelbar“ unterbrochen (vor dem nächsten Maschinenbefehl)
- ② Die Zustellung weiterer Interrupts wird gesperrt
  - Zwischenzeitlich auflaufende Interrupts werden gepuffert (maximal einer pro Quelle!)
- ③ Registerinhalte werden gesichert (z. B. im Stapel)
  - PC und Statusregister automatisch von der Hardware
  - Vielzweckregister üblicherweise manuell in der ISR
- ④ Aufzurufender Behandler (*interrupt service routine, ISR*) wird ermittelt
- ⑤ ISR wird ausgeführt
- ⑥ ISR terminiert mit einem „return from interrupt“-Befehl
  - Registerinhalte werden restauriert
  - Zustellung von Interrupts wird freigegeben
  - Das Anwendungsprogramm wird fortgesetzt



- Für die Anwendung ist die Behandlung einer Ausnahme im allgemeinen **zwingend** und prozessorabhängig.
  - **Aufwerfen** (*raising*) einer Ausnahme erfolgt entweder durch einen realen Prozessor (CPU) oder einen abstrakten Prozessor (Betriebssystem).
    - die CPU wirft Ausnahmen der Hardware (*interrupt, trap, syscall*)
    - das BS wirft Ausnahmen der Software (UNIX/POSIX: Signale **SIG\***)
  - **Behandlung** (*handling*) einer Ausnahme erfolgt immer durch einen abstrakten Prozessor
    - CPU-Ausnahmen (*interrupt, trap, syscall*) behandelt das Betriebssystem auf  $E_2$
    - BS-Ausnahmen (Signale) behandelt das Maschinenprogramm (ein Prozess) auf  $E_3$
- Behandlung erfolgt durch einen entsprechenden **Ausnahmebehandler**.
  - *syscall* Ausführen der gewünschten Systemfunktion auf  $E_2$ , Rückkehr
  - *trap* Fehler beheben auf  $E_2$ , Wiederholen der Instruktion – oder *abort*
  - *interrupt* Verarbeiten des externen Ereignisses auf  $E_2$ , Rückkehr
  - **SIG\*** Behandeln auf  $E_3$ , Rückkehr, Wiederholen oder *abort*



- Der Ausnahmebehandler wird **aktiviert** über eine unbedingte Verzweigung
  - **CPU-Ausnahme:** Sprung über einen Vektor der *Interrupt Vector Table* (IVT)
    - enthält Adressen von Unterprogrammen (seltener auch direkt Instruktionen)
    - liegt systemweit im RAM oder ROM, spezielles CPU-Register enthält Basisadresse
    - Modifikation der IVT ist eine privilegierte  $E_2$ -Anweisung
  - **BS-Signal:** Sprung in eine zuvor beim BS registrierte Funktion
    - POSIX: mit sigaction<sup>(2)</sup> für den Prozess registriert
    - wird bei Auftreten eines Signals nach dem Einlasten des Prozesses ausgeführt

## 6.4 Systemsicht – $E_2$ -Ausnahmen

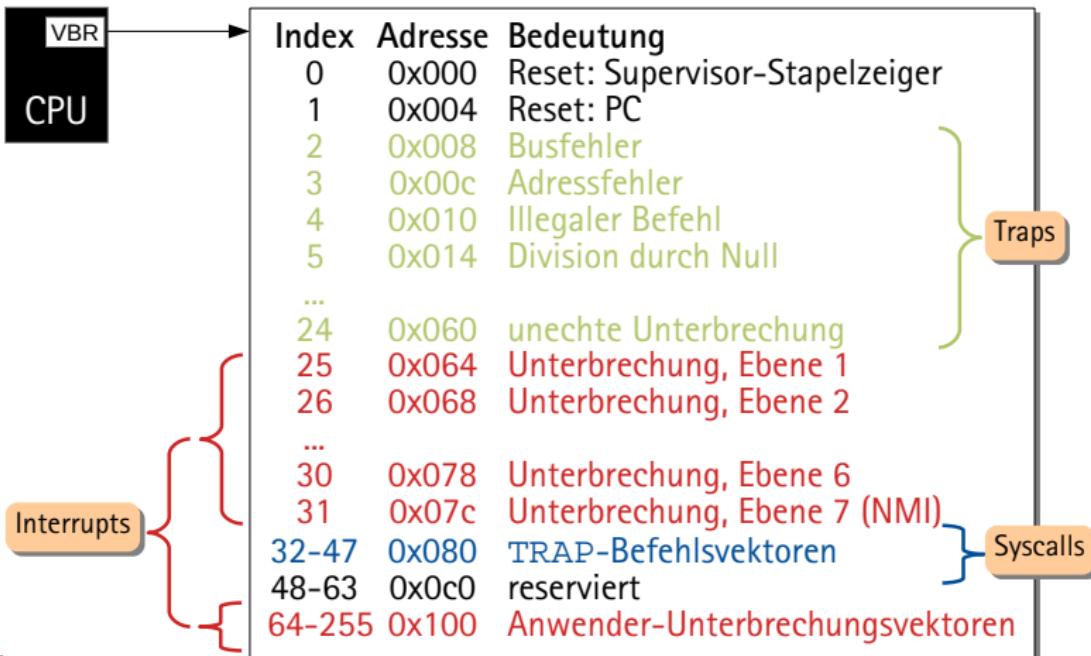
# 6 Unterbrechungen, Ausnahmen, Signale

Interrupt Vector

Table

- Beispiel: Vektortable des MC 68000 Prozessors

(Vector Base Register des Prozessors zeigt auf die Sprungtabelle)



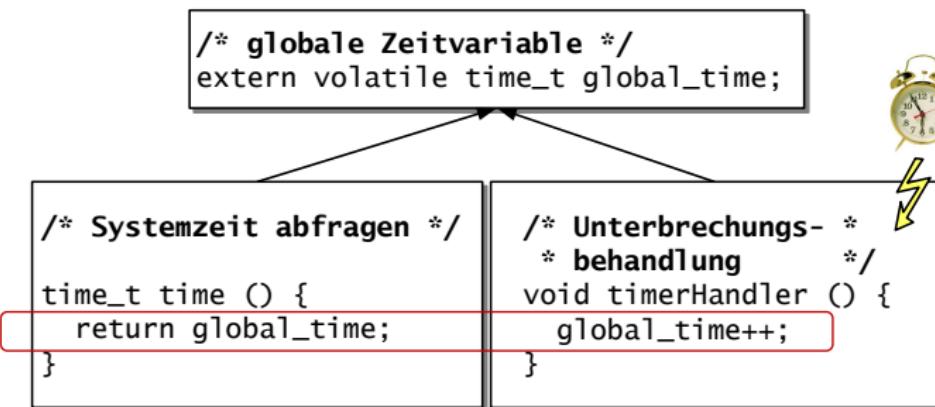
# Ausnahmen und Unterbrechungen: Synchronisation

- Ein Systemaufruf oder Trap ist Folge einer **bestimmten**  $E_2$ -Operation.
    - synchron zum Befehlsfluss, (im Prinzip) vorhersagbar
  - Eine asynchrone Unterbrechung (*interrupt*) der CPU ist **jederzeit** möglich.
    - Jederzeit  $\mapsto$  nach jeder  $E_2$ -Operation
- ↪ Problematisch bei gemeinsam verwendetem Zustand
- IRQ-Behandler unterbricht laufendes Programm und schreibt Daten in einem Puffer, aus dem ebendieses gerade liest.
  - Inkonsistenzen möglich!
- Ansatz: **Kritische Gebiete** erkennen und schützen
    - Während Kontrollfluss im kritischen Gebiet ist, Unterbrechung **verzögern**
    - Erkennung des kritischen Gebietes ist jedoch in der Regel **nicht trivial**

# Ausnahmen und Unterbrechungen: Synchronisation

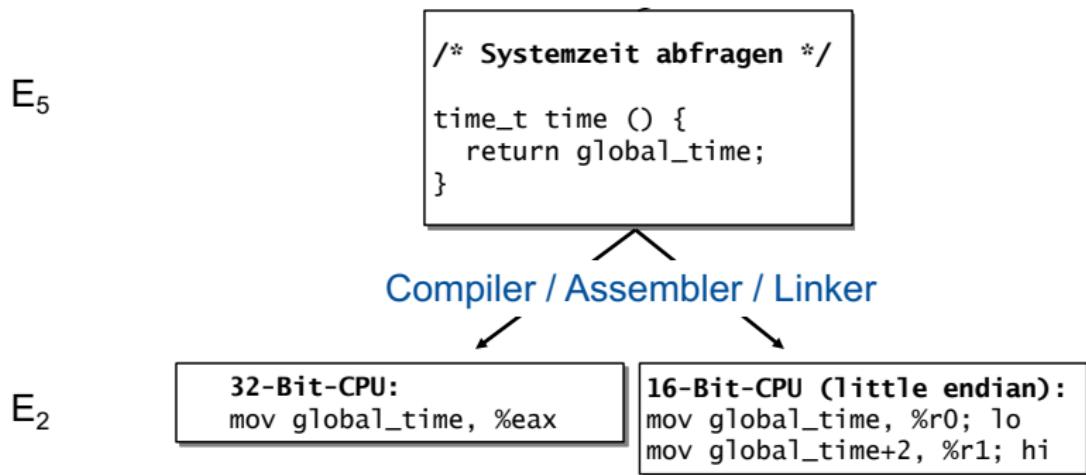
- Beispiel: Systemzeit – Sekunden seit Start des Rechners
  - Jede Sekunde sendet der Zeitgeber einen IRQ.
  - Behandler `timerHandler()` erhöht Systemzeit.
  - Anwendung kann mit `time()` die Systemzeit auslesen.

kritisches  
Gebiet



# Ausnahmen und Unterbrechungen: Synchronisation

- Annahme: Prozessor führt jede einzelne  $E_2$ -Instruktion **atomar** aus.
  - Eine Hochsprachen-Instruktion ( $E_5$ ) wird jedoch in der Regel auf mehrere Instruktionen der Befehlssatzebene ( $E_2$ ) abgebildet.
  - Diese Abbildung ist **hochgradig abhängig von Prozessorarchitektur und Compiler!**





# Ausnahmen und Unterbrechungen: Synchronisation

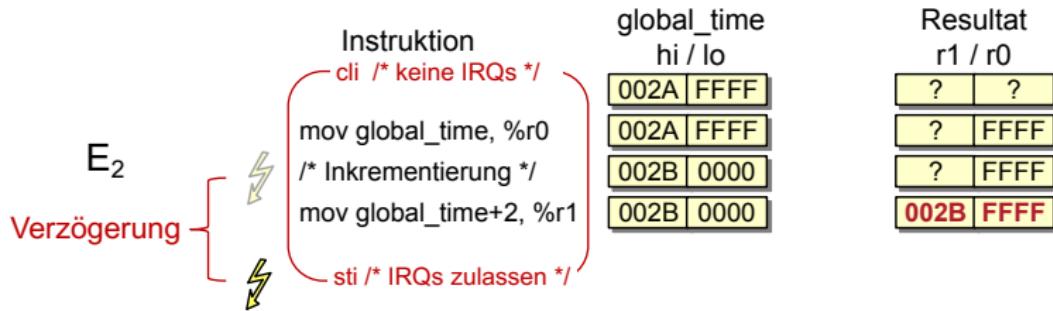
- Inkonsistenz ist möglich, wenn auf der 16-Bit CPU die Unterbrechung zwischen den beiden Leseoperationen eintrifft.

| Instruktion            | global_time |      | Resultat |      |
|------------------------|-------------|------|----------|------|
|                        | hi          | lo   | r1       | r0   |
| ?                      | 002A        | FFFF | ?        | ?    |
| E <sub>2</sub>         | 002A        | FFFF | ?        | FFFF |
| mov global_time, %r0   | 002B        | 0000 | ?        | FFFF |
| /* Inkrementierung */  | 002B        | 0000 | 002B     | FFFF |
| mov global_time+2, %r1 |             |      |          |      |

- Allerdings nur, wenn gerade ein Überlauf vom *lo*- ins *hi*-Word stattfindet.
  - Alle 65536 Sekunden könnte die Systemzeit für eine Sekunde inkonsistent sein
  - Extrem schwer zu reproduzierender Fehler!

# Ausnahmen und Unterbrechungen: Synchronisation

- Inkonsistenz ist möglich, wenn auf der 16-Bit CPU die Unterbrechung zwischen den beiden Leseoperationen eintrifft.

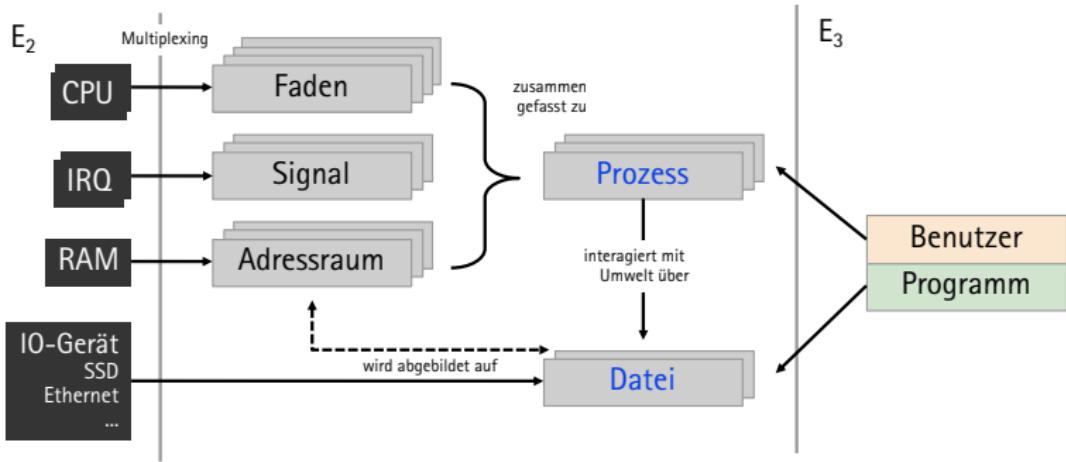


- Mögliche Lösung: **Unterbrechungsbehandlung sperren** (verzögern)
  - Einseitige Synchronisation:** Anwendung verzögert Unterbrechungsbehandlung
  - Verzögerung erhöht die **Unterbrechungslatenz**. Sie sollte **kurz** sein!
  - Sperren (`cli`) und Freigeben (`sti`) der Unterbrechungen sind privilegierte  $E_2$ -Operationen, da sonst die Monoplisierung der CPU möglich wäre.

## 6.5 Systemsicht – $E_3$ -Ausnahmen: Signale



- Virtuelle Hardwareressourcen werden repräsentiert durch die grundlegenden Konzepte (**Abstraktionen**) eines Betriebssystems. **In UNIX sind das:**
  - **Prozess** (Adressraum + Fäden) ↪ virtueller „Computer“ (Prozessor)
  - **Signal** ↪ virtuelle „Ausnahme“ dieses Prozessors (synchron wie asynchron)





## ■ **synchrones Signal**, ausgelöst durch die Aktivität des [eigenen Prozesses](#)

- „durchgereichter“  $E_2$ -Trap bei Ausführung einer Instruktion durch die [CPU](#)
  - *div/0* ([SIGFPE](#)), *segment violation* ([SIGSEGV](#)), *illegal instruction* ([SIGILL](#)), ...
- zusätzlicher  $E_3$ -Trap bei Ausführung eines syscalls durch das [Betriebssystem](#)
  - *pipe error* ([SIGPIPE](#)), *illegal syscall* ([SIGSYS](#)), ...
- wie Traps nicht unterdrückbar/verzögerbar  $\rightsquigarrow$  **Behandlung zwingend**

## ■ **asynchrones Signal**, ausgelöst durch [externes Ereignis / anderen Prozess\(or\)](#)

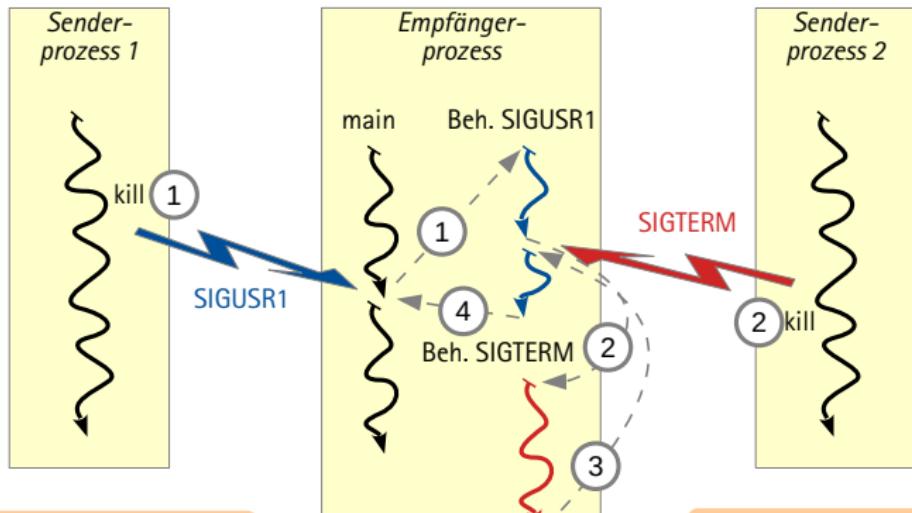
- „durchgereichtes“ asynchrones Ereignis des L2-Prozessors ([CPU](#), HW-Ereignis)
  - *Ctrl-C pressed* ([SIGINT](#)), *timer expired* ([SIGALRM](#)), ...
- zusätzliche asynchrone Ereignisse eines  $E_3$ -Prozessors ([anderer Prozess](#), BS)
  - *child stopped/terminated* ([SIGCHLD](#)), *terminate* ([SIGTERM](#)),  
*hangup* ([SIGHUP](#)), *user signal 1/2* ([SIGUSR1](#), [SIGUSR2](#)), ...
- wie IRQs unterdrückbar/verzögerbar  $\rightsquigarrow$  **Synchronisation erforderlich**
  - analog zum NMI gibt es auch nicht unterdrückbare Signale: [SIGKILL](#) und [SIGSTOP](#)

# Asynchrone Signale: Logische Sicht

IPC

- Signale zur einfachen Interprozesskommunikation (IPC)
  - Implizit durch das BS bei Kindprozessen (SIGCHLD, wait<sup>(2)</sup>)
  - Explizit mit anderem Prozess mit der kill<sup>(2)</sup>-Operation

→ 10



Signalisierung und Start der Behandlungsroutine erfolgen logisch gleichzeitig.

Behandlungsroutinen können selbst auch unterbrochen werden.

- Die Signalbehandlung erfolgt immer (erst) beim Verlassen des Kerns
  - wenn der  $E_3$ -Prozessor (Prozess) seine Arbeit (wieder) aufnimmt
- Genauer Ablauf ist abhängig vom Zustand des Prozesses:
  - **LAUFEND** (synchrone/asynchrone Signale)
    - Unmittelbarer Start des Behandlers
  - **BEREIT** (nur asynchrone Signale, z. B. zugestellt durch kill<sup>(2)</sup>)
    - Signal wird im Prozesskontrollblock vermerkt → 5-36
    - Behandlung erfolgt „unmittelbar“, wenn der Prozess eingelastet (LAUFEND) wird
  - **BLOCKIERT** (Prozess führt gerade blockierenden Systemaufruf durch)
    - Der E/A-Systemaufruf (z. B. read<sup>(2)</sup>) wird (intern) unterbrochen – falls möglich.
    - Der Prozess wird BEREIT gestellt ⇔ weiterer Ablauf siehe oben.
    - Beim Installieren des Signalbehandlers kann entschieden werden, ob der Systemaufruf fortgesetzt werden soll (**SA\_RESTART**) oder mit dem Rückgabewert **EINTR** abbricht.

## 6.6 Anwendungssicht – Signale: Nebenläufigkeit



- Asynchrone Signale erzeugen Nebenläufigkeit innerhalb des Prozesses

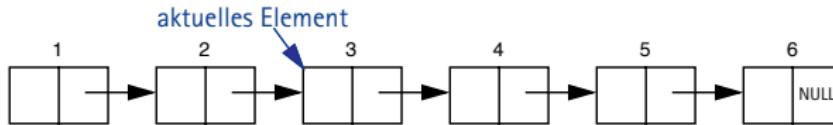
→ Problemstellung vergleichbar zu IRQs auf  $E_2$

→ 6-26

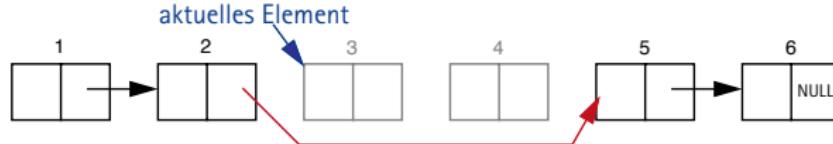
- Während der Ausführung könnte Zustands vorübergehend inkonsistent sein
- Problematisch, falls Signalbehandler auf denselben Zustand zugreift

- Beispiel:

- Prozess durchläuft gerade eine verkettete Liste (z. B. Freispeicherliste des Heaps)



- Prozess wird unterbrochen durch Signal: Behandler entfernt Elemente 3 und 4



Elementzeiger  
nun ungültig!

- Asynchrone Signale erzeugen Nebenläufigkeit innerhalb des Prozesses

↪ Problemstellung vergleichbar zu IRQs auf  $E_2$

↪ [6-26]

- Während der Ausführung könnte Zustands vorübergehend inkonsistent sein
- Problematisch, falls Signalbehandler auf **denselben Zustand** zugreift

- Dieses trifft für **viele Funktionen der C-Bibliothek** zu!

- Beispiel Heap: `malloc()`, `free()`, ...
  - Freispeicherverwaltung  $\mapsto$  Zustand
  - viele weitere Funktionen rufen intern `malloc()` auf!
- Beispiel E/A: `printf()`, `puts()`, `scanf()`, `perror()`, ...
  - Interne Puffer und Dateizeiger  $\mapsto$  Zustand
  - viele weitere Funktionen rufen intern `perror()` auf!

↪ Ein Signalbehandler darf **sehr viele** Funktionen nicht verwenden!

- Nur die unter signal-safety<sup>(7)</sup> dokumentierten Funktionen sind erlaubt.
- E/A niemals über `libc`-Funktionen, sondern über syscalls durchführen.



- Die Behandlung asynchroner Signale kann **unterdrückt (blockiert)** werden
  - Signale werden dann erst bei der Deblockierung zugestellt.
  - Analog zu IRQs und Unterbrechungssperren mit `cli/sti` auf  $E_2 \leftrightarrow$  6-26

## → Einseitige Synchronisation

- Einstellung pro Signal in der **Signalmaske** des Prozesses: `sigprocmask`<sup>(2)</sup>:

```
int sigprocmask(int how,
    ↴ const sigset_t *set,
    ↴ sigset_t *oldset);
```

Setzt (how: `SIG_BLOCK`) oder löscht (how: `SIG_UNBLOCK`) die in `sig` gesetzten Bits in der Signalmaske des Prozesses oder überschreibt diese komplett (how: `SIG_SET`). In `oldset` wird optional die alte Maske zurückgegeben.

- Beispiel:** Blockieren von `SIGUSR1` in einem kritischen Gebiet

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL);

... kritisches Gebiet

sigprocmask(SIG_UNBLOCK, &set, NULL);
```

## 6.7 Zusammenfassung



## Zusammenfassung: Unterbrechungen, Ausnahmen und Signale

- Prozessor aktiviert im bestimmten Situationen eine **Ausnahmebehandlung**
  - und **damit das Betriebssystem** im Systemmodus des Prozessors ( $E_2$ )
- Betriebssystem kontrolliert, **interpretiert** und behandelt die Ausnahmen
  - syscall** **synchron**, explizite Anweisung für Sprung in das Betriebssystem
  - trap** **synchron**, implizite Verzweigung bei fehlerhafter Anweisung
    - BS behebt Fehler transparent (z. B. bei *page fault*)
    - BS propagt Fehler zum auslösenden Prozess (z. B. *DIV/0*).  
In UNIX erfolgt dieses durch Zustellung eines **Signals**.
    - vorhersagbar, nicht unterdrückbar, nicht verzögerbar
- interrupt** **asynchron**, implizite Verzweigung bei externem Ereignis
  - BS bearbeitet IRQ transparent für die Anwendung
  - nicht vorhersagbar  $\rightsquigarrow$  ggfs. **Synchronisation** erforderlich
  - unterdrückbar/verzögern durch **privilegierte  $E_2$ -Operationen**

- Nur durch Interrupts erhält das BS **garantiert** immer wieder die Kontrolle!
- Signale sind das **Pendant** zu CPU-Ausnahmen auf Maschinenprogrammebene.

4 Dateien und Dateisysteme

5 Prozesse und Fäden

6 Unterbrechungen, Ausnahmen, Signale

## 7 Prozessverwaltung

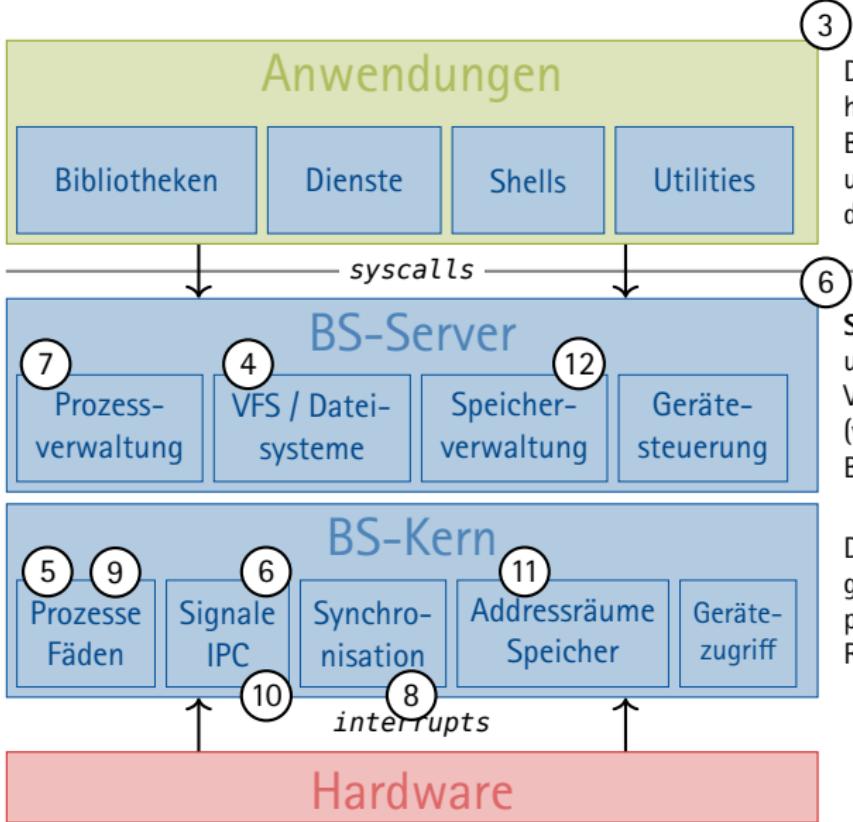
- 7.1 Einordnung
- 7.2 Systemsicht: Prozesseinplanung und -verarbeitung
- 7.3 Anwendungssicht: Betriebsarten
- 7.4 Systemsicht: Planungsebenen
- 7.5 Verfahren – Überblick
- 7.6 Verfahren – Kooperativ
- 7.7 Verfahren – Verdrängend
- 7.8 Verfahren – Probabilistisch
- 7.9 Verfahren – Mehrstufig
- 7.10 Exkurs: Echtzeitbetrieb
- 7.11 Verfahren – Gegenüberstellung
- 7.12 Beispiel aus der Praxis: Der Linux-Scheduler
- 7.13 Zusammenfassung



## 7.1 Einordnung

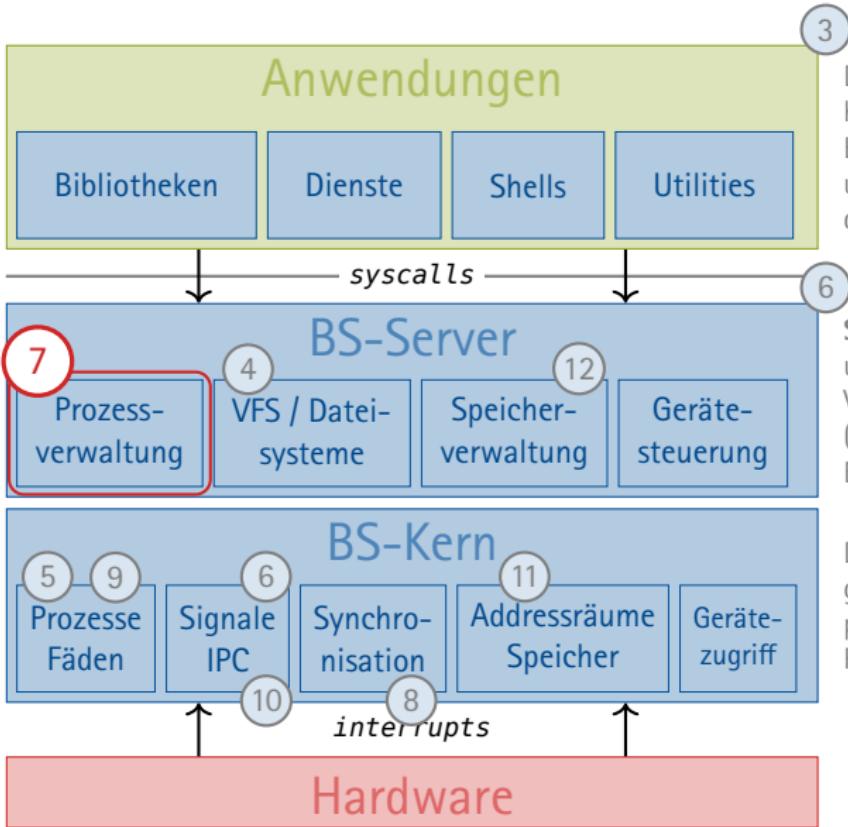


# 7 Prozessverwaltung – Einordnung





# 7 Prozessverwaltung – Einordnung

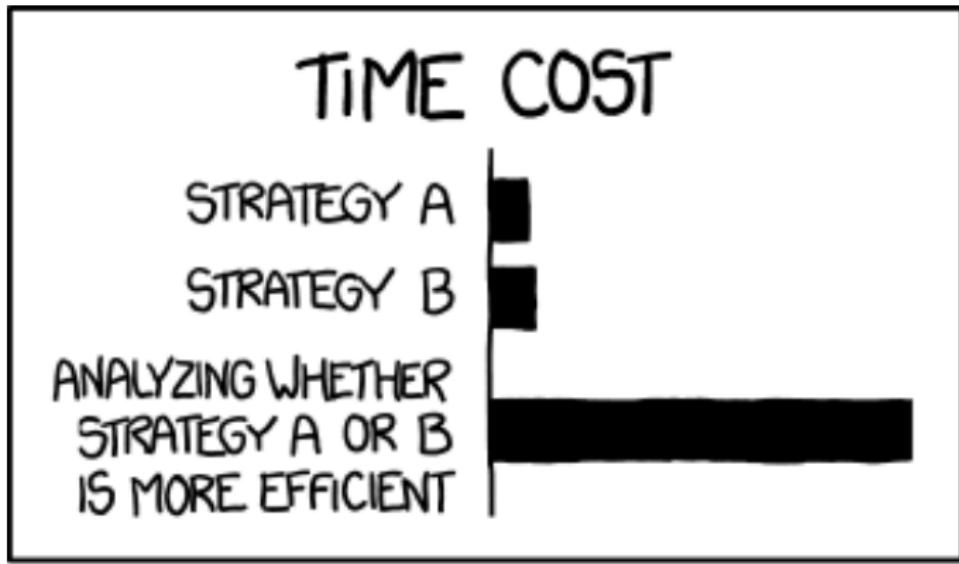


Das "Userland" stellt weitergehende Dienste, Programme und Bibliotheken zur **Verwendung** und **Steuerung** des BS durch die Benutzerinnen.

Benutzermodus  
Systemmodus

Server erweitern Abstraktionen und stellen die **Strategien** zur Verwaltung und Zuteilung der (virtuellen) HW-Ressourcen an Benutzer und Prozesse.

Der Kern stellt die grundlegende **Mechanismen** für Multiplexing und Isolation der HW-Ressourcen.



Quelle: <https://xkcd.com/1445>



Was „gute Planung“ ist hängt von der Ebene der Betrachtung ab.

## Benutzersicht: Wahrgenommenes Systemverhalten

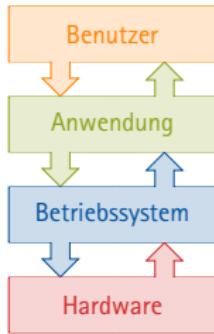
- Antwortzeit einer Systemanforderung – unter Wahrnehmungsschwelle
- Durchlaufzeit eines Prozesses – minimal
- Fairness der Ressourcenverteilung – nachvollziehbar

## Anwendungssicht: Transparenz und Verlässlichkeit

- Termintreue bei der Prozessaktivierung – verlässliche Einplanung
- Vorhersagbarkeit der Ressourcenzuteilung – Fortschrittsgarantie
- Dringlichkeiten berücksichtigen – Priorisierung von Aktivitäten

## Systemsicht: Auslastung der Systemressourcen

- Durchsatz des Rechensystems – Prozesse pro Zeiteinheit
- Hardwareauslastung – Kosten des Rechnerbetriebs minimieren
- Lastausgleich ermöglichen – Hardware gleichmäßig nutzen





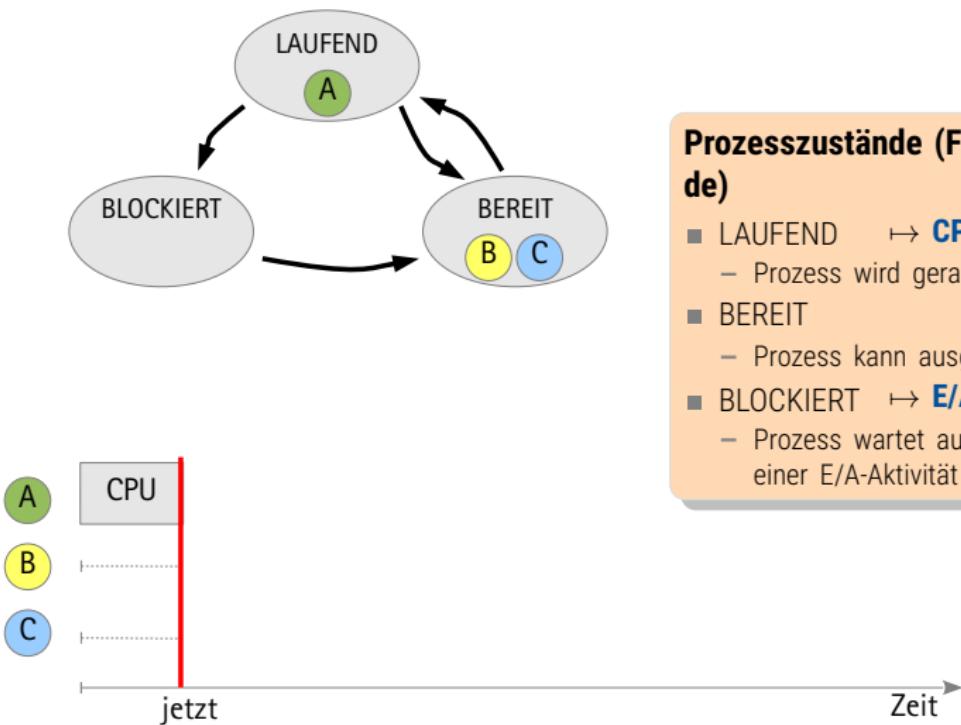
## 7.2 Systemsicht: Prozesseinplanung und -verarbeitung



# Prozessverarbeitung auf einem Betriebssystem

Wiederholung: Prozessverhalten und -zustände

← 5



## Prozesszustände (Fadenzustände)

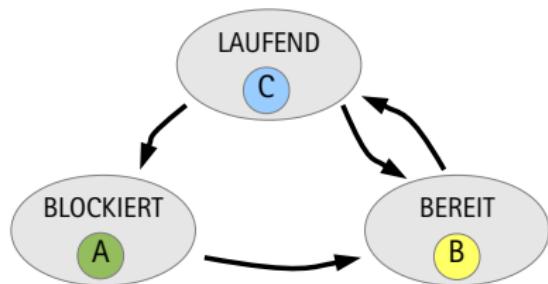
- LAUFEND  $\mapsto$  **CPU-Stoß**
  - Prozess wird gerade ausgeführt
- BEREIT
  - Prozess kann ausgeführt werden
- BLOCKIERT  $\mapsto$  **E/A-Stoß**
  - Prozess wartet auf Beendigung einer E/A-Aktivität



# Prozessverarbeitung auf einem Betriebssystem

Wiederholung: Prozessverhalten und -zustände

← 5



## Kontextwechsel

Prozess A hat seinen CPU-Stoß vorzeitig beendet (vor Ablauf der Zeitscheibe) und einen E/A-Stoß gestartet (BLOCKIERT).

Der **Planer** hat den Prozess C ausgewählt und von BEREIT in LAUFEND überführt.

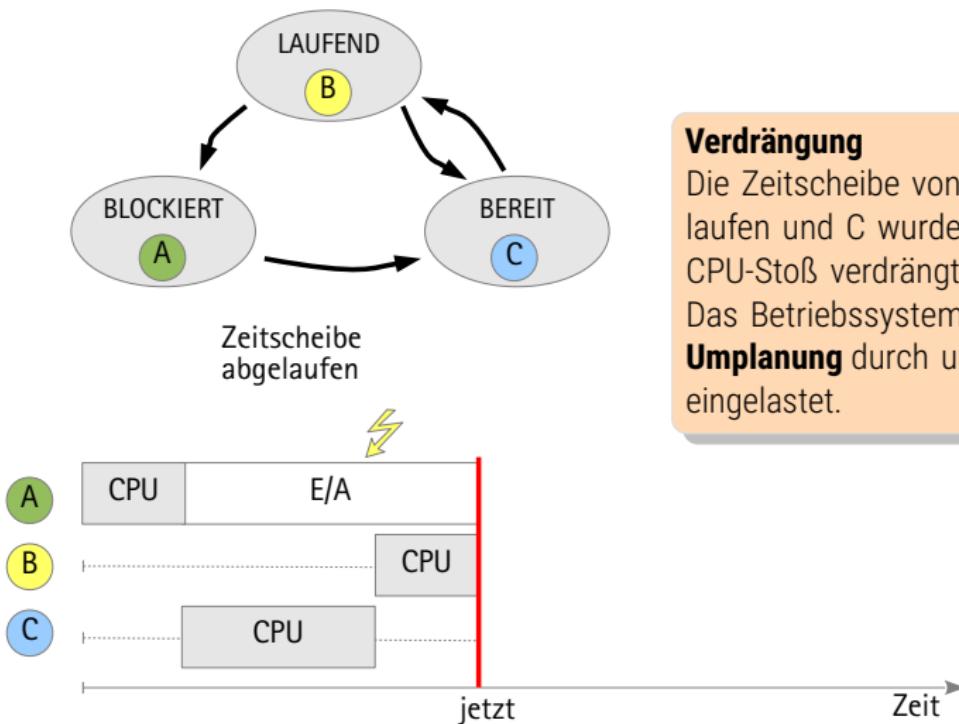




# Prozessverarbeitung auf einem Betriebssystem

Wiederholung: Prozessverhalten und -zustände

← 5

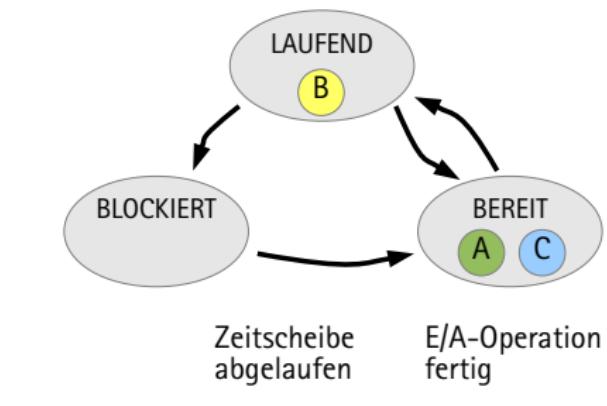




# Prozessverarbeitung auf einem Betriebssystem

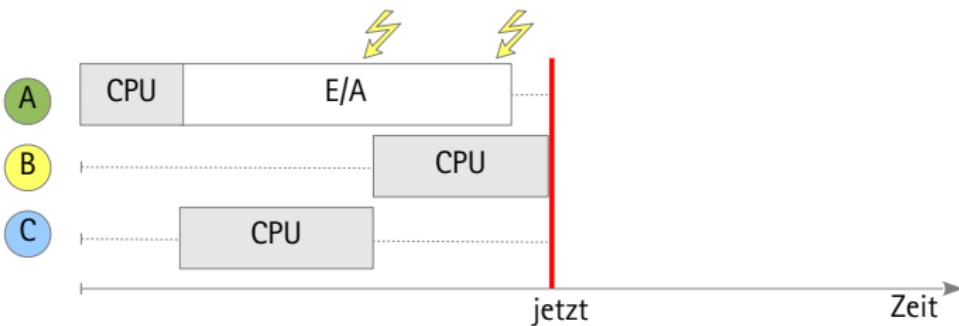
Wiederholung: Prozessverhalten und -zustände

← 5



## Bereitstellung

Der E/A-Stoß von A ist nun abgeschlossen. Daraufhin wird A nun wieder **eingeplant** (BEREIT) und wartet auf die Zuteilung der CPU für seinen nächsten CPU-Stoß.

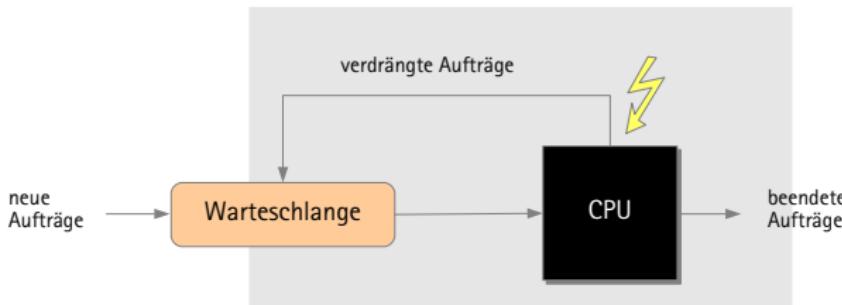


**Aufgabe (job)**  $\mapsto$  ausführbarer Arbeitsauftrag („Programm“)

**Prozess/Faden**  $\mapsto$  ausführende Einheit

- Planungsalgorithmen betrachten in der Regel **Aufgaben (jobs)**
  - Die CPU-Zuteilung erfolgt in der Regel an Prozesse und Fäden
    - Betrachtungsweise entstammt dem **Stapelbetrieb**
    - Zugelassene Aufgaben (*high-level scheduling*) wird instantiiert als (ein) Prozess, der den Auftrag abarbeitet und mit dem Ergebnis terminiert.
  - Betrachtungsweise im **Dialogbetrieb**
    - Zugelassene **Benutzer** (*high-level scheduling*) starten **interaktive Prozesse**
    - gekennzeichnet durch Wechsel von E/A und Berechnung:  
E/A-Stoß  $\longrightarrow$  CPU-Stoß  $\longrightarrow$  E/A-Stoß  $\longrightarrow$  CPU-Stoß ...
- Aufgabe  $\mapsto$  CPU-Stoß (*CPU burst*)

- Allgemein: Zuteilung eines Betriebsmittels an wartende Interessierte



„Betriebsmittel“  
sei im Folgenden  
immer die CPU.

- Der **Planungsalgorithmus** (*Scheduling*) definiert **Reihenfolge** und **Bedingungen** unter denen die Aufträge eingereiht und entnommen werden.
- Unter Einhaltung komplexer Randbedingungen wie Fairness, Durchsatz, Reaktionszeit, Ressourcennutzung  $\mapsto$  **Planungsziele**.
- Planungsziele können in der Regel nicht alle gleichzeitig erreicht werden. Ihre Gewichtung hängt ab von der **Betriebsart**. Es bleiben **Zielkonflikte**.



## 7.3 Anwendungssicht: Betriebsarten



# Planung: Betriebsarten eines Rechensystems operation modes

## ■ **Stapelbetrieb** (*batch scheduling*)

interaktionsloser / unabhängiger Programme

- nicht verdrängend
- verdrängend mit langen Zeitscheiben (Minimierung des Umplanungsaufwands)

Rechnen, Sortieren, Analysieren  
großer Datenmengen

Ziel: Durchsatz

## ■ **Dialogbetrieb** (*interactive scheduling*)

interaktionsreicher / abhängiger Programme

- ereignisgesteuerte, verdrängende Verfahren
- kurze Zeitscheiben (Minimierung der Antwortzeiten)

Textverarbeitung, Browser, Shell

Ziel: Antwortzeit

## ■ **Echtzeitbetrieb** (*real-time scheduling*)

zeitkritischer / abhängiger Programme

Sicherheitskritisches Steuergerät

- ereignisgesteuerte / zeitgesteuerte deterministische Verfahren
- Einhaltung **physischer Zeitschranken** ist das entscheidende Kriterium

Ziel: Rechtzeitigkeit



## 7.4 Systemsicht: Planungsebenen

### High-Level Scheduling [s-min]

„User“

System-Zugangskontrolle für Aufgaben (*job scheduling, admission scheduling*): Zugelassene Aufgaben werden initiiert zu interaktiven Prozessen oder Gruppen von interaktiven Prozessen.

### Intermediate-Level Scheduling [ms-s]

„Process“

CPU-Zugangskontrolle für Prozesse: Steuert Aktivierung und Deaktivierung (Ein- und Auslagerung) von Prozessen. Nur aktive Prozesse konkurrieren um Betriebsmittel.

### Low-Level Scheduling [ $\mu$ s-s]

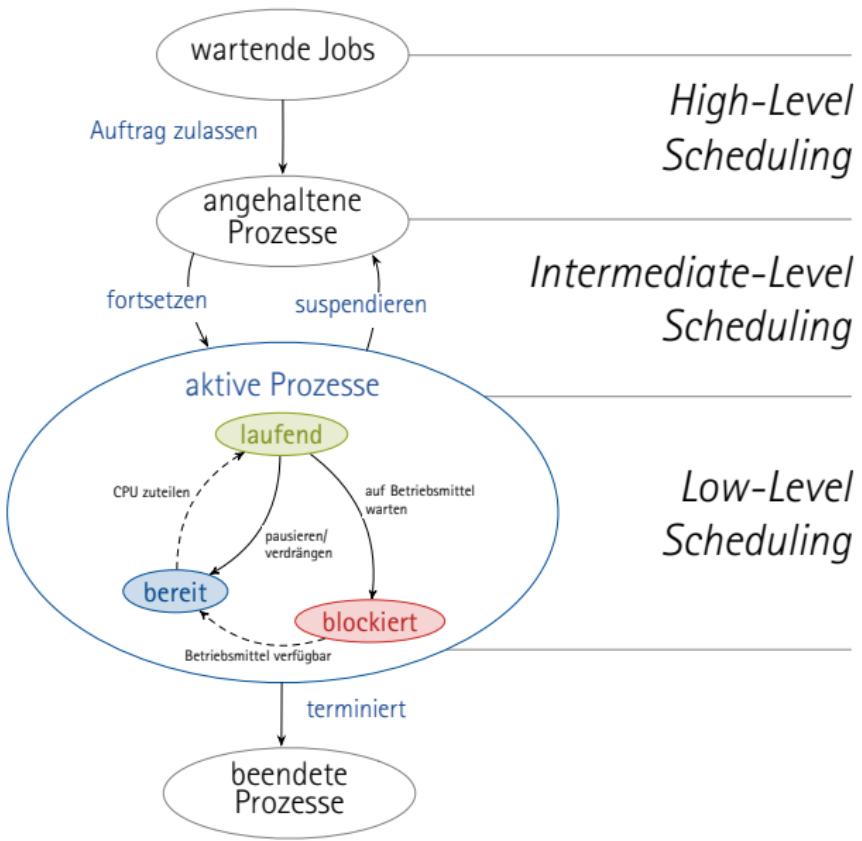
„Thread“

CPU-Zuteilungskontrolle für Fäden: Steuert die Einlastung konkreter Prozesse/Fäden über die Zustände BEREIT  $\leftrightarrow$  LAUFEND  $\leftrightarrow$  BLOCKIERT.



# Planung: Ebenen

scheduling levels





## 7.5 Verfahren – Überblick

- Planungsverfahren arbeiten zu verschiedenen **Zeitpunkten**
  - **online scheduling** → dynamisch, während der Ausführung
  - **offline scheduling** → statisch, vor der Ausführung
- unter der Annahme einer **Vorhersagbarkeit**
  - **deterministic scheduling** → bekannter, vorberechneter Prozesse
  - **probabilistic scheduling** → unbekannter Prozesse
- und eines **Kooperationsverhaltens**
  - **cooperative scheduling** → voneinander abhängiger Prozesse, welche die CPU freiwillig abgeben
  - **preemptive scheduling** → voneinander unabhängiger Prozesse, denen die CPU entzogen werden kann

UNIX-artige Systeme verwenden in der Regel einen Planer, der die Prozesse (Fäden) **online, probabilistisch** und **verdrängend** einplant.



# Klassische Planungs- und Auswahlverfahren

|                 |                                        |                                   |
|-----------------|----------------------------------------|-----------------------------------|
| kooperativ      | FCFS                                   | „Wer zuerst kommt, mahlt zuerst!“ |
|                 | ■ <i>first come, first served</i>      |                                   |
| verdrängend     | RR, VRR                                | „Reihum, jeder gegen jeden!“      |
|                 | ■ <i>round robin</i>                   |                                   |
|                 | ■ <i>virtual round robin</i>           |                                   |
| probabilistisch | SPN (SJF), SRTF, HRRN                  | „Die Kleinen nach vorne!“         |
|                 | ■ <i>shortest job first</i>            |                                   |
|                 | ■ <i>shortest remaining time first</i> |                                   |
|                 | ■ <i>highest response ration first</i> |                                   |
| mehrstufig      | MLQ, MLFQ, FSS                         | „Gesunder Mischkonsum“            |
|                 | ■ <i>multi-level queues</i>            |                                   |
|                 | ■ <i>multi-level feedback queues</i>   |                                   |
|                 | ■ <i>fair share scheduling</i>         |                                   |

Gütemerkmal ist im Folgenden die Fähigkeit zur Interaktion mit „externen Prozessen“, insbesondere dem Menschen ↗ Fokus liegt auf Antwortzeit.



## 7.6 Verfahren – Kooperativ



Fair, einfach zu implementieren (FIFO) ... und dennoch problematisch

Sortierkriterium  
der Warteliste

- Einplanung erfolgt nach der **Ankunftszeit** (*arrival time*)
  - und in der sich daraus ergebenden Reihenfolge wird auch verarbeitet
  - nicht-verdrängendes Verfahren, setzt kooperative Prozesse voraus
- gerechtes Verfahren
  - auf Kosten einer im Mittel höheren Antwortzeit und niedrigerem E/A-Durchsatz
  - suboptimal bei einem Mix von kurzen und langen CPU-Stößen

Prozesse mit  $\begin{cases} \text{langen} \\ \text{kurzen} \end{cases}$  CPU-Stößen werden  $\begin{cases} \text{begünstigt} \\ \text{benachteiligt} \end{cases}$

- Problem: **Konvoieffekt**       $\mapsto$  kurzer CPU-Stoß folgt einem langen



- Durchlaufzeit kurzer Prozesse im Mix mit langen Prozessen:

| Prozess     | Zeiten  |       |       |      |       | $T_q/T_s$ |
|-------------|---------|-------|-------|------|-------|-----------|
|             | Ankunft | $T_s$ | Start | Ende | $T_q$ |           |
| A           | 0       | 1     | 0     | 1    | 1     | 1.00      |
| B           | 1       | 100   | 1     | 101  | 100   | 1.00      |
| C           | 2       | 1     | 101   | 102  | 100   | 100.00    |
| D           | 3       | 100   | 102   | 202  | 199   | 1.99      |
| $\emptyset$ |         |       |       | 100  |       | 26.00     |

$T_s$  = Bedienzeit (auf dem Prozessor),  $T_q$  = Durchlaufzeit (Ankunft bis Ende)

- **normalisierte Durchlaufzeit:**  $(T_q/T_s)$ 
  - ideal für A und B, unproblematisch für D, **schlecht für C**
    - sie steht in einem extrem schlechten Verhältnis zur Bedienzeit  $T_s$
    - typischer Effekt im Falle von kurzen Prozessen, die langen folgen



## 7.7 Verfahren – Verdrängend



Verdrängendes FCFS, Zeitscheiben, CPU-Schutz.

- Prozesse werden nach ihrer **Ankunftszeit** ein- und in regelmäßigen Zeitabständen (periodisch) umgeplant
  - verdrängendes Verfahren, nutzt **periodische Unterbrechungen**
  - jeder Prozess erhält eine **Zeitscheibe** (*time slice*) zugeteilt
    - ~~ obere Schranke für den CPU-Stoß
- Verringerung der bei FCFS auftretenden Benachteiligung von Prozessen mit kurzen CPU-Stößen
  - die **Zeitscheibenlänge** bestimmt die Effektivität des Verfahrens
    - zu lang ~~ Degenerierung zu FCFS
    - zu kurz ~~ hoher Mehraufwand durch Kontextwechsel
  - Faustregel: etwas länger als die Dauer eines „typischen CPU-Stoßes“
- Problem: **Konvoieffekt**
  - „Kurzer“ Prozess folgt einem, der seine Zeitscheibe (Quantum) aufbraucht

Timer-  
Interrups

Zeitscheibe wird  
auch Quantum genannt



- da die Zuteilung der Zeitscheiben an Prozesse nach FCFS geschieht, werden kurze Prozesse nach wie vor benachteiligt:

**E/A-intensive Prozesse** schöpfen ihre Zeitscheibe selten voll aus

- sie beenden ihren CPU-Stoß freiwillig

Vor Ablauf  
der Zeitscheibe

**CPU-intensive Prozesse** schöpfen ihre Zeitscheibe meist voll aus

- sie beenden ihren CPU-Stoß unfreiwillig

durch  
Verdrängung

- unabhängig davon werden jedoch alle Prozesse immer reihum bedient

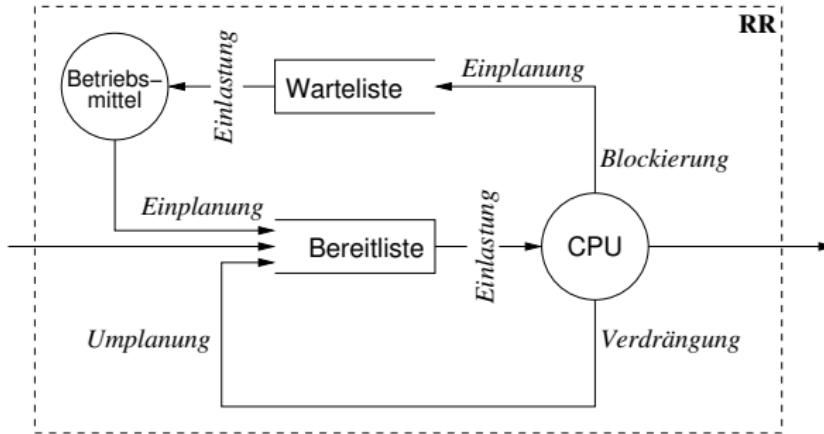
→ wird eine Zeitscheibe durch einen Prozess nicht ausgeschöpft, verteilt sich die CPU-Zeit zu Ungunsten E/A-intensiver Prozesse

- E/A-intensive Prozesse werden schlechter bedient
- E/A-Geräte sind schlecht ausgelastet
- Varianz der Antwortzeit E/A-intensiver Prozesse kann beträchtlich sein



# RR: Konvoieffekt

round robin

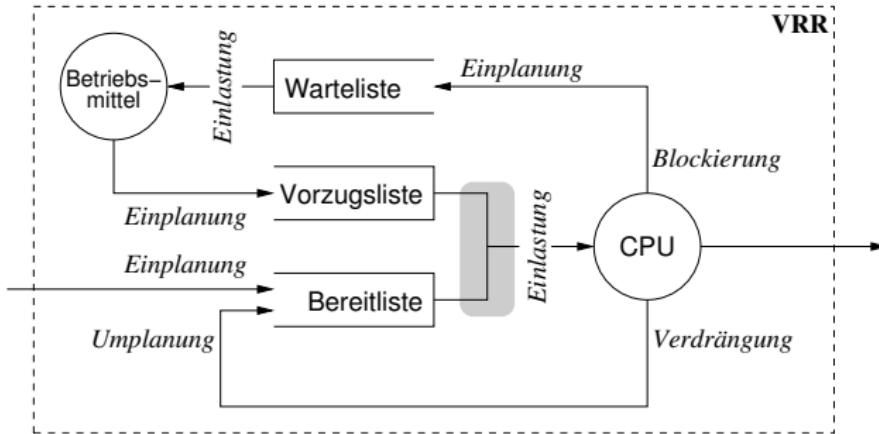


- **RR:** CPU-intensive und E/A-intensive Prozesse werden „gleich“ behandelt
  - Eine **gemeinsame** Bereitliste zur Zuteilung der CPU
  - E/A-intensive Prozesse bekommen (relativ) weniger CPU-Zeit
  - E/A-Geräte werden schlecht ausgelastet



# Alternative: VRR

**virtual round robin**



- **RR:** CPU-intensive und E/A-intensive Prozesse werden „gleich“ behandelt
  - Eine **gemeinsame** Bereitliste zur Zuteilung der CPU
  - E/A-intensive Prozesse bekommen (relativ) weniger CPU-Zeit
  - E/A-Geräte werden schlecht ausgelastet
- **VRR:** Nach E/A kommen Prozesse auf eine **Vorzugsliste**
  - diese wird von der CPU vorrangig bedient
  - Prozesse können nach E/A so den „Rest“ ihrer Zeitscheibe aufbrauchen

Vermindert Antwortzeit  
interaktiver Prozesse



*RR mit Vorzugswarteschlange und variablen Zeitscheiben, um interaktive ( $\mapsto E/A$ -intensive) Prozesse nicht zu benachteiligen.*

- auf E/A wartende Prozesse werden mit Beendigung ihres jeweiligen E/A-Stoßes bevorzugt eingeplant (bereit gestellt)
  - **Einplanung** mittels einer der Bereitliste vorgeschalteten **Vorzugsliste**
    - FIFO  $\rightsquigarrow$  evtl. Benachteiligung hoch-interaktiver Prozesse; daher...
    - aufsteigend sortiert nach dem **Zeitscheibenrest** eines Prozesses
  - **Umplanung** bei Ablauf der aktuellen Zeitscheibe
    - die Prozesse auf der Vorzugsliste werden zuerst eingelastet
    - sie bekommen die CPU für die Restdauer ihrer Zeitscheibe zugeteilt
    - bei Ablauf dieser Zeitscheibe werden sie in die Bereitsliste eingereiht
  - erreicht durch strukturelle Maßnahmen – nicht durch analytische
- **kein** voll-verdrängendes Verfahren
  - Einlastung erfolgt dennoch erst nach Ablauf der laufenden Zeitscheibe



## 7.8 Verfahren – Probabilistisch



Zeitreihen bilden, analysieren und verwerten: nicht verdrängend.

- jeder Prozess wird entsprechend der für ihn im Durchschnitt oder maximal **erwarteten Bedienzeit** eingeplant
  - Grundlage dafür ist *à priori Wissen* über die **Prozesslaufzeiten**:
    - Stapelbetrieb Programmierer setzen **Frist** (*time limit*)
    - Produktionsbetrieb Erstellung einer **Statistik** durch Probelaufe
    - Dialogbetrieb **Abschätzung** von CPU-Stoßlängen zur Laufzeit
  - Abarbeitung einer aufsteigend nach Laufzeiten sortierten Bereitsliste
    - Abschätzung erfolgt vor (statisch) oder zur (dynamisch) Laufzeit
- Verkürzung von Antwortzeiten und Steigerung der Gesamtleistung des Systems bei Benachteiligung längerer Prozesse
  - ein **Verhungern** (*starvation*) dieser Prozesse ist möglich

durch immer wieder neu eintreffende kurze Prozesse
- ohne Konvoi-Effekt – jedoch ist als praktikable Implementierung nur eine **näherungsweise Lösung** möglich



- ein **heuristisches Verfahren**, das für jeden Prozess den **Mittelwert** über seine jeweiligen CPU-Stoßlängen bildet
  - Erwarte Länge des nächsten CPU-Stoßes eines Prozesses:

Mittelwert über  
alle CPU-Stoßlängen

$$S_{n+1} = \frac{1}{n} \cdot \sum_{i=1}^n T_i$$

- Problem: **gleiche Wichtigung** aller CPU-Stöße
- jüngere CPU-Stöße machen die **Lokalität** eines Prozesse aus  
~~ sollten stärker berücksichtigt werden
- **Messung** der Dauer  $T_i$  eines CPU-Stoßes erfolgt bei Prozessumschaltung
  - Werte werden im Prozesskontrollblock aufakkumuliert
  - Kann durch Unterbrechungen (*interrupts*) verfälscht sein

→ 5-36

- Mittels eines **Dämpfungsfilters** (*decay filter*) kann der Einfluss weiter zurückliegenden CPU-Stöße begrenzt werden:

$$S_{n+1} = \alpha \cdot T_n + (1 - \alpha) \cdot S_n$$

- mit zuletzt gemessener ( $T_n$ ) und geschätzter ( $S_n$ ) CPU-Stoßlänge und konstantem **Wichtungsfaktor**  $\alpha$ , für den gilt:  $0 < \alpha < 1$
  - Wirkung des Wichtungsfaktors (Expansion der rekursiven Formel):
    - $S_{n+1} = \alpha T_n + (1 - \alpha)\alpha T_{n-1} + \dots + (1 - \alpha)^i \alpha T_{n-i} + \dots + (1 - \alpha)^n S_1$
  - Beispiel der Entwicklung für  $\alpha = 0.8$ :
    - $S_{n+1} = 0.8T_n + 0.16T_{n-1} + 0.032T_{n-2} + 0.0064T_{n-3} + \dots$
- Zurückliegende CPU-Stöße des Prozesses verlieren schnell an Gewicht



Hungerfreies SPN: nicht verdrängend

- Einplanung nach der (erwartet) **kürzesten Bedienzeit**, aber **Umplanung** unter Berücksichtigung der **Wartezeit**

Fairness!

- Einführung **dynamischer Prioritäten**
- regelmäßige periodische Neuberechnung der Prioritäten anhand:

$$\text{Prio} = \frac{\text{Wartezeit} + \text{Bedienzeit}}{\text{Bedienzeit}}$$

Durchlaufzeit

- Neuberechnung betrifft alle Einträge in der Bereitliste
- Es bleibt das Problem der Schätzung der Bedienzeit
- Der Anstieg der Wartenzeit wird als **Alterung** (*aging*) bezeichnet
  - Der Alterung entgegenwirken (*anti-aging*) beugt Verhungern (*starvation*) vor

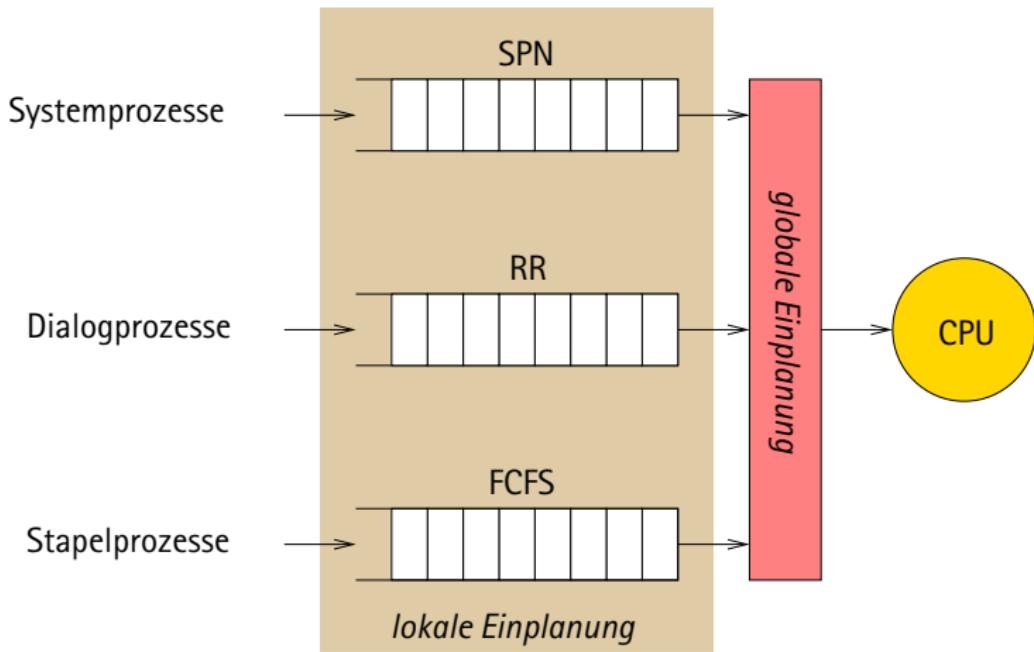


## 7.9 Verfahren – Mehrstufig



Unterstützt Mischbetrieb: Vorder- und Hintergrundbetrieb.

- Prozesse werden nach ihrem jeweiligen **Typ** eingeplant
  - Aufteilung der Bereitliste in separate („getypte“) Listen
    - z.B. für System-, Dialog- und Stapelprozesse
  - mit jeder Liste eine **lokale Einplanungsstrategie** verbinden
    - z.B. SPN, RR und FCFS
  - zwischen den Listen eine **globale Einplanungsstrategie** definieren
    - statisch
      - Liste einer bestimmten Prioritätsebene fest zuordnen
      - Hungergefahr für Prozesse tiefer liegender Listen
    - dynamisch
      - die Listen im Zeitmultiplexverfahren wechseln
      - z.B. 40 % System-, 40 % Dialog-, 20 % Stapelprozesse
- **Problem:** Typfestlegung erfordert Vorabwissen (statische Entscheidung)



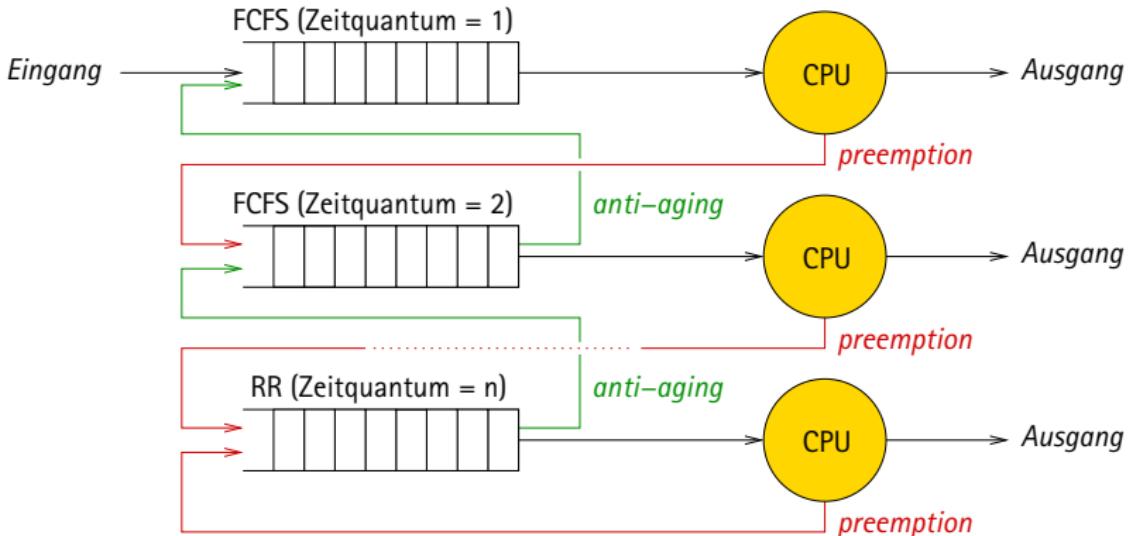


Begünstigt kurze/interaktive Prozesse, ohne Vorabwissen.

- Prozesse werden nach ihrer **Ankunftszeit** ein- und in regelmäßigen Zeitabständen (periodisch) umgeplant
- Hierarchie von Bereitlisten für **Prioritätsebenen**  $1, \dots, n$ 
  - Ebene 1: höchste Priorität, aber geringste Zeitscheibenlänge (Quantum  $q$ )
    - jeder Ebene ist eine Zeitscheibenlänge zugeordnet:  $q_1 < q_2 < \dots < q_n$
    - erstmalig eintreffende Prozesse steigen oben (Ebene 1) ein
    - Innerhalb einer Ebene gilt FCFS (Außnahme: RR auf der unterste Ebene  $n$ )
- **Bestrafung** (*penalization*)
  - Zeitscheibenablauf drückt den laufenden Prozess weiter nach unten
    - Prozesse mit langen CPU-Stößen fallen nach unten durch
- **Alterung** (*aging* und *anti-aging*)
  - nach unten durchfallende Prozesse finden seltener statt
    - durchgefallene Prozesse nach einer **Bewährungsfrist** wieder anheben



# MLFQ: Bestrafung und Bewährung

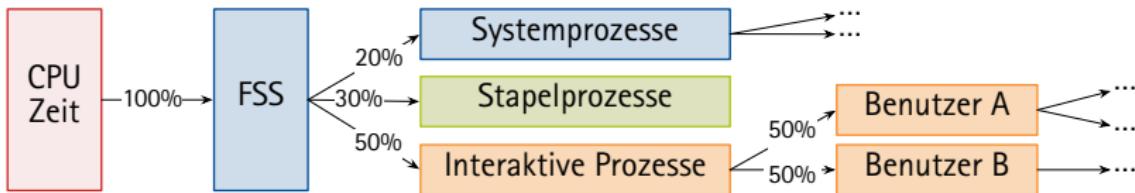


Feedback durch „Bestrafung“ bei Verdrängung und „Bewährung“ (*anti-aging*) bei freiwilliger CPU-Abgabe. Verbunden mit den unterschiedlichen Zeitscheibenlängen adaptiert MLFQ gut auf unbekannte Prozesse im **Vielzweckbetrieb**.



Gleichmäßige oder festgelegte („faire“) Aufteilung der Rechenzeit.

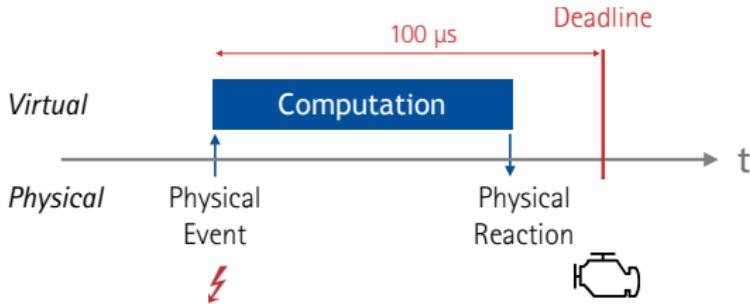
- Grundidee ist eine **hierarchisches Verteilung** der Systemressourcen
  - Oberste Ebene: Verteilung nach festem Schlüssel
  - Weitere Ebenen: problemspezifische Planer
- Ausgleich von Interessensgruppen / Mischbetrieb von Betriebsarten
  - Beispiel Cloud: Mehrere virtuelle Maschinen teilen sich einen echten Rechner
    - VM-Anteil an CPU/Speicher entsprechend der Bezahlung
  - Beispiel Mainframe: Sowohl interaktive als auch Stapel- und Systemprozesse
    - Aufteilung: 50 Prozent für interaktive Prozesse, Rest für Stapel-/Systemprozesse





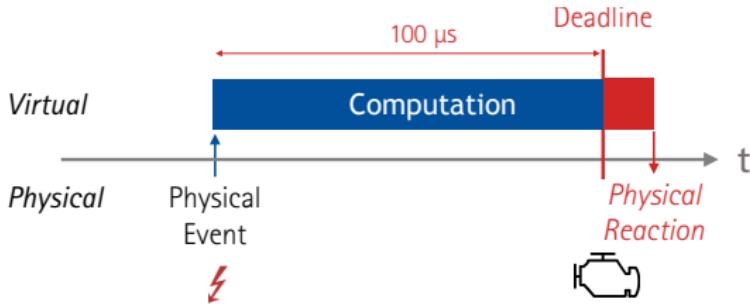
## 7.10 Exkurs: Echtzeitbetrieb

- Vorrangiges Ziel: Einhaltung von Zeitvorgaben  $\leftrightarrow$  Planungsziel!
  - bezogen auf physikalische Zeit
  - Zeitvorgaben durch die Anwendung: Einhaltung einer **maximalen Reaktionszeit** auf ein Ereignis (*worst-case response time, WCRT*)



- **Weiches** Echtzeitsystem:  
Zeitvorgaben sind möglichst einzuhalten  
(Überschreitung  $\rightsquigarrow$  Ergebnis verliert an Wert)
- **Festes** Echtzeitsystem:  
Zeitvorgaben sind einzuhalten  
(Überschreitung  $\rightsquigarrow$  Ergebnis ist wertlos)
- **Hartes** Echtzeitsystem:  
Zeitvorgaben sind **unbedingt einzuhalten**  
(Überschreitung  $\rightsquigarrow$  Katastrophe)

- Vorrangiges Ziel: Einhaltung von Zeitvorgaben     $\leftrightarrow$  Planungsziel!
  - bezogen auf physikalische Zeit
  - Zeitvorgaben durch die Anwendung: Einhaltung einer **maximalen Reaktionszeit** auf ein Ereignis (*worst-case response time, WCRT*)



- **Weiches** Echtzeitsystem:  
Zeitvorgaben sind möglichst einzuhalten  
(Überschreitung  $\rightsquigarrow$  Ergebnis verliert an Wert)
- **Festes** Echtzeitsystem:  
Zeitvorgaben sind einzuhalten  
(Überschreitung  $\rightsquigarrow$  Ergebnis ist wertlos)
- **Hartes** Echtzeitsystem:  
Zeitvorgaben sind **unbedingt einzuhalten**  
(Überschreitung  $\rightsquigarrow$  Katastrophe)



# Harte, feste oder weiche Echtzeit?







## 7.11 Verfahren – Gegenüberstellung



# Gegenüberstellung

|                 | FCFS | (V)RR | SPN | HRRN | SRTF |
|-----------------|------|-------|-----|------|------|
| kooperativ      | ✓    |       | (✓) | (✓)  |      |
| verdrängend     |      | ✓     |     |      |      |
| probabilistisch |      |       | ✓   | ✓    |      |
| deterministisch |      |       |     |      | ✓    |

keines  $\rightsquigarrow$  EZS [21]

- MLQ, MLFQ und FSS erlauben eine Kombination der Verfahren
  - Prozesseinplanung unterliegt einer **breit gefächerten** Einordnung 7-16
  - kooperativ/verdrängend, deterministisch/probablistisch, statisch/dynamisch
- Verfahren legen einen **Prozessvorrang** (Priorisierung) fest
  - nach Ankunftszeit FCFS, RR, VRR, (MLFQ)
  - nach Bedienzeit SPN, (MLFQ)
  - nache Bedien/Wartezeit HRRN
  - nach Restbedienzeit SRTF
- Bei probabilistischen Verfahren bleibt das **Problem der Abschätzung**.



# Gegenüberstellung

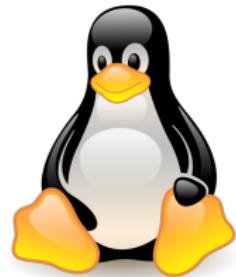
|      | präemptiv/<br>kooperativ   | Vorhersage<br>nötig | Implement.-<br>aufwand | Verhungern<br>möglich | Priorisierung<br>nach      | Auswirkung auf<br>Prozesse                       |
|------|----------------------------|---------------------|------------------------|-----------------------|----------------------------|--------------------------------------------------|
| FCFS | kooperativ                 | nein                | minimal                | nein                  | Ankunftszeit               | Konvoi-Effekt                                    |
| RR   | präemptiv<br>(Zeitgeber)   | nein                | klein                  | nein                  | Ankunftszeit               | Fair, aber benachteiligt<br>E/A-lastige Prozesse |
| VRR  | präemptiv<br>(Zeitgeber)   | nein                | mittel                 | nein                  | Ankunftszeit               | Fair, aber nicht<br>voll-präemptiv               |
| SPN  | kooperativ                 | ja                  | groß                   | ja                    | Bedienzeit                 | Benachteiligt<br>CPU-lastige Prozesse            |
| SRTF | präemptiv<br>(bei Eingang) | ja                  | größer                 | ja                    | Restbedien-<br>zeit        | Benachteiligt<br>CPU-lastige Prozesse            |
| HRRN | kooperativ                 | ja                  | groß                   | nein                  | Bedien-/<br>Wartezeit      | Gute Lastverteilung                              |
| MLFQ | präemptiv<br>(Zeitgeber)   | nein                | größer                 | ja                    | Ankunftszeit<br>Bedienzeit | Bevorzugt u.U.<br>E/A-lastige Prozesse           |

Angelehnt an [31]

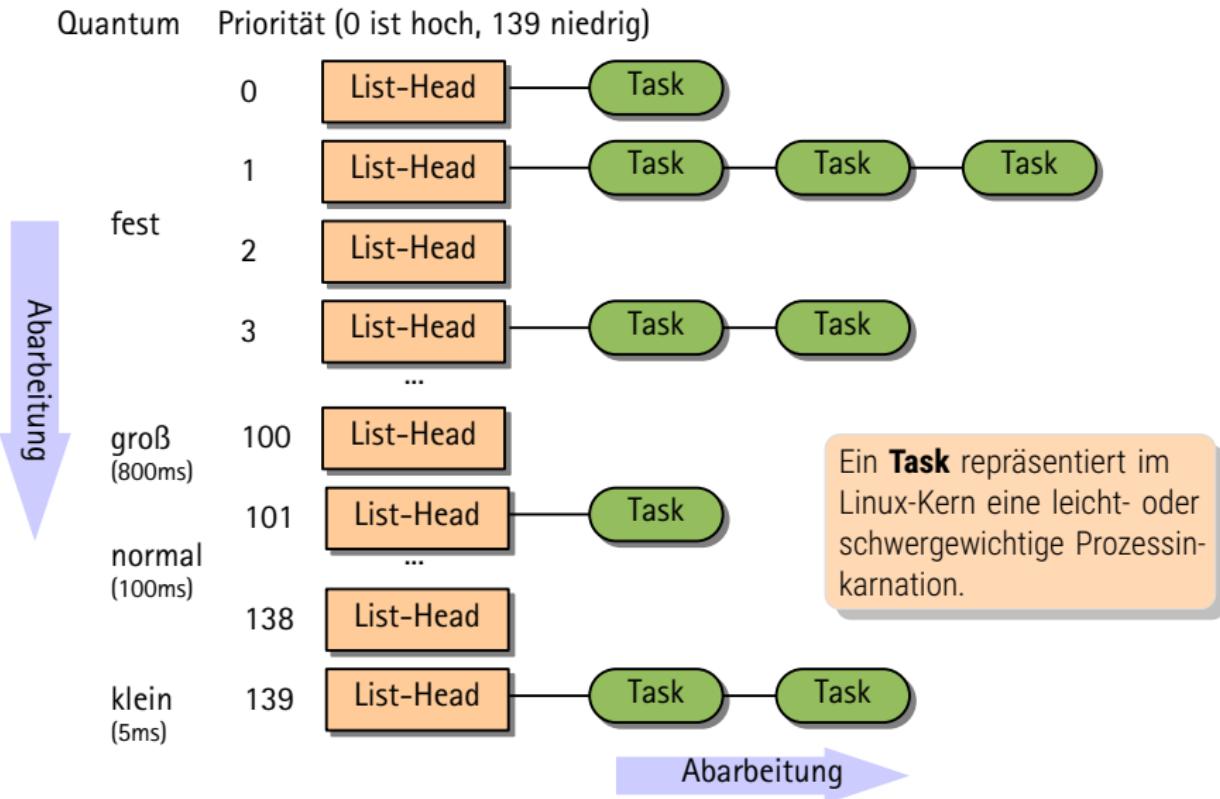


## 7.12 Beispiel aus der Praxis: Der Linux-Scheduler

- Kernel < 2.4: RR mit Prioritäten
  - eine gemeinsame Warteschlange
- Kernel  $\geq 2.4$ : O(n) Scheduler
  - immer noch eine gemeinsame Warteschlange
  - aber mehr Fairness durch **Scheduler-Epochen** (VRR mit Prioritäten)
  - extrem miese Leistung auf Mehrkernsystemen
- Kernel  $\geq 2.5$ : O(1) Scheduler
  - MLQ mit effizienter Epochenverwaltung durch *active* und *expired* Listen
  - erheblich bessere Leistung auf Mehrkernsystemen
  - Heuristiken zur Bevorzugung interaktiver Anwendungen
- Kernel  $\geq 2.6.23$ : Completely Fair Scheduler (CFS)
  - FSS bis hinunter zu einzelnen Prozessen
  - sehr schlank (und dadurch effizient)



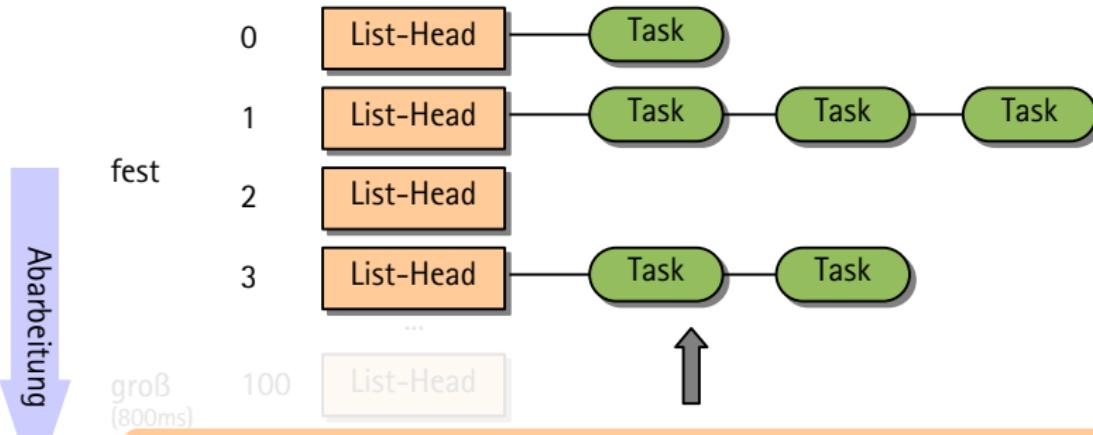
# Linux 0(1)-Scheduler: Spezielles MLQ mit Epochen





# Linux 0(1)-Scheduler: Spezielles MLQ mit Epochen

Quantum Priorität (0 ist hoch, 139 niedrig)



### Echtzeit-Tasks

SCHED\_FIFO nie verdrängt

SCHED\_RR verdrängt, wenn feste Zeitscheibe abgelaufen

Echtzeit-Tasks verdrängen jeden anderen normalen Task.

Durch die einfache Strategie kann das Verhalten in einer weichen Echtzeitumgebung recht gut vorhergesagt werden.

Abarbeitung



# Linux 0(1)-Scheduler: Spezielles MLQ mit Epochen

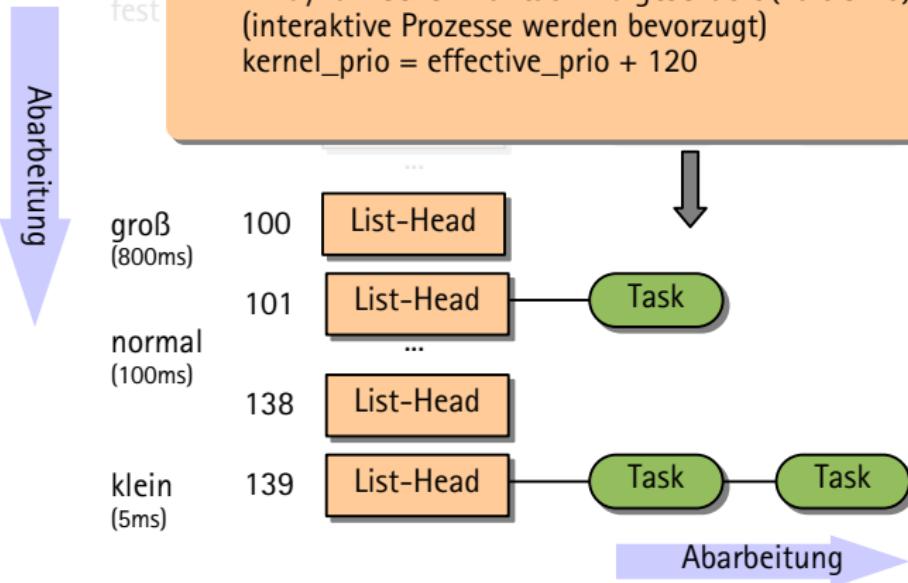
Quantum Priorität (0 ist hoch, 139 niedrig)

## Gewöhnliche Tasks

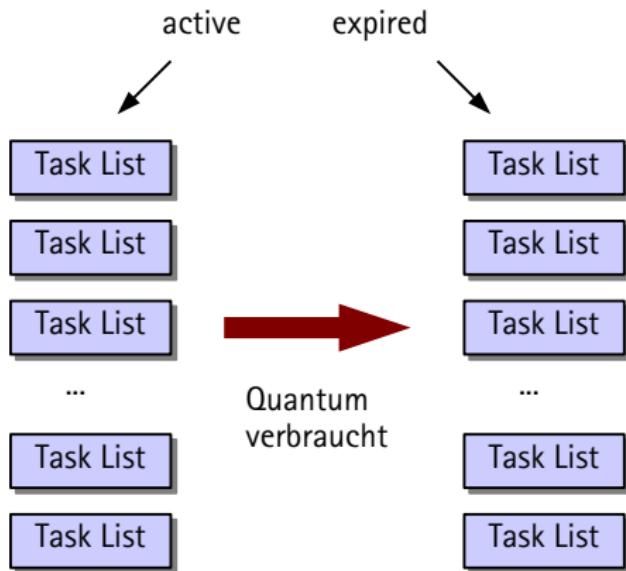
$\text{effective\_prio} = \text{static\_prio} + \text{dynamic\_prio}$

- statische Priorität entspricht dem nice value (-19 bis +20)
- dynamische Priorität wird geschätzt (-5 bis +5)  
(interaktive Prozesse werden bevorzugt)

$\text{kernel\_prio} = \text{effective\_prio} + 120$



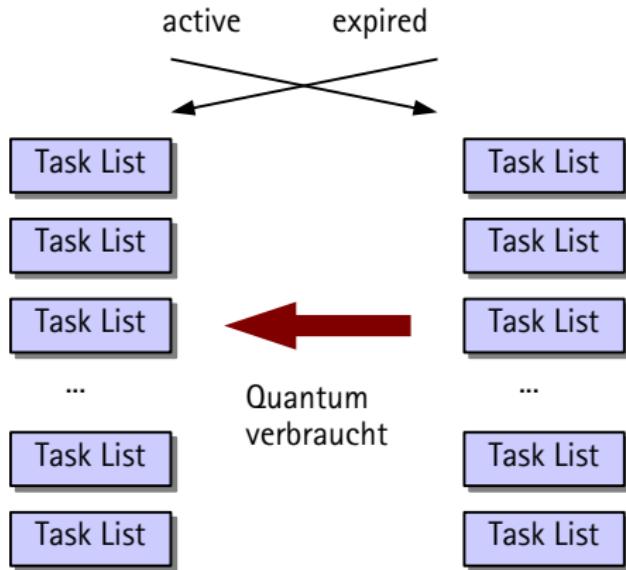
- Hat ein Task seine Zeitscheibe (Quantum) aufgebraucht, so kommt auf die *expired*-Liste.



Vergleichbar mit VRR:  
Solange ein interaktiver  
Task seine Zeitscheibe  
nicht aufgebraucht, bleibt  
er auf der *active*-Liste

# Linux O(1)-Scheduler: Epochenwechsel

- Hat ein Task seine Zeitscheibe (Quantum) aufgebraucht, so kommt auf die *expired*-Liste.
- Wurde die *active*-Liste komplett abgearbeitet, beginnt die nächste Epoche (durch Tauschen der Listen).



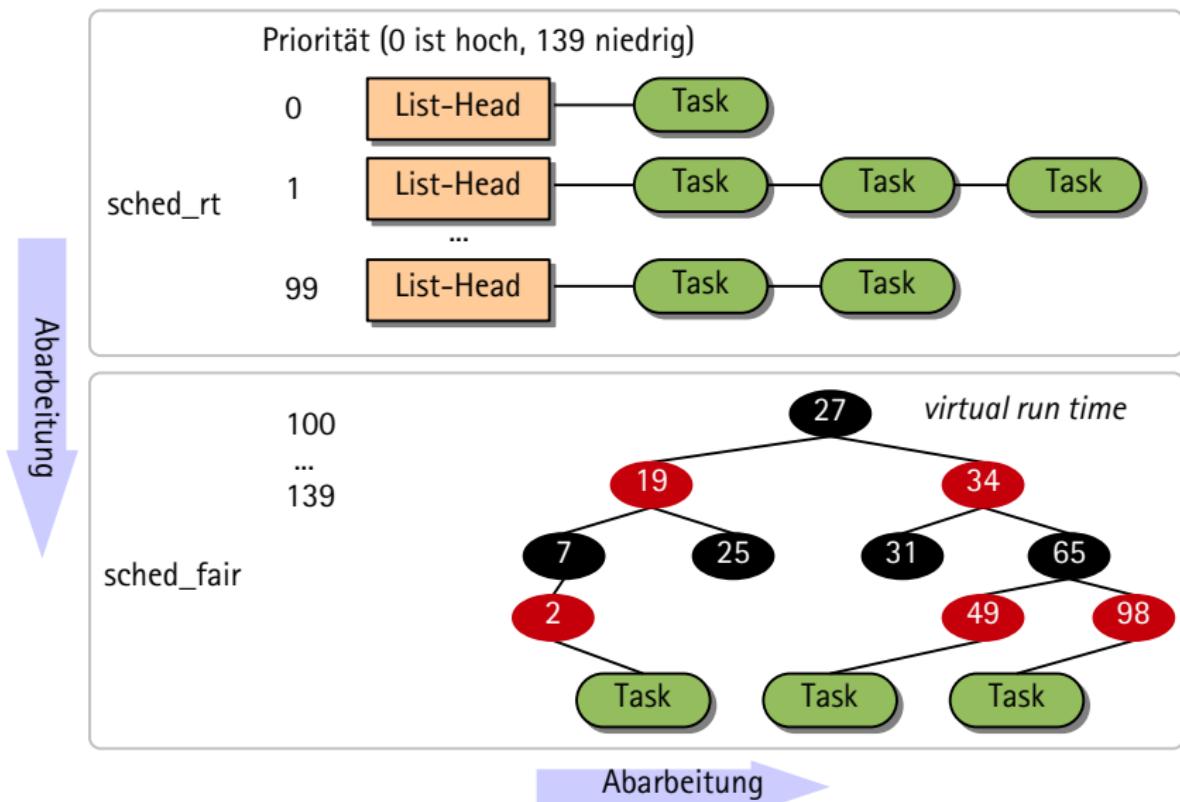
Vergleichbar mit VRR:  
Solange ein interaktiver  
Task seine Zeitscheibe  
nicht aufgebraucht, bleibt  
er auf der *active*-Liste



# Kernel $\geq$ 2.6.23: Completely Fair Scheduler (CFS)

- MLQ-Scheduler: Heuristiken zur **Bevorzugung** „wichtiger“ Prozesse
  - insbesondere interaktiver Prozesse
  - dadurch **kompliziert**, aufwändig, viele Randfälle zu beachten
- Neuer Ansatz: **FSS**, bis runter auf Prozessebene – der **CFS**-Scheduler
  - keine Unterschiede mehr (*completely fair*)!
  - bei  $n$  Prozessen bekommt jeder  $1/n$ -tel der CPU-Leistung
  - hierarchische Zuteilung aber weiter möglich (durch scheduling groups)
- CFS läuft nur im **SCHED\_NORMAL**-Bereich (Prioritäten 100–139)
  - „Echtzeit“-Prioritäten 0–99 laufen wie bisher (**SCHED\_RR** und **SCHED\_FIFO**)
  - Einplanungskriterium ist die **bislang zugeteilte CPU-Zeit**
  - Bereitliste als Rot-Schwarz-Baum, sortiert nach der zugeteilten CPU-Zeit
  - Dynamische Prioritäten (nice<sup>(2)</sup>) durch „schnellere/langsamere“ Uhren abgebildet

# Kernel $\geq$ 2.6.23: Completely Fair Scheduler (CFS)





## 7.13 Zusammenfassung



- Betriebssysteme treffen Zuteilungsentscheidungen auf **drei Ebenen**
  - **High-Level Scheduling** Zulassung von Prozessen zum System
  - **Intermediate-Level Scheduling** Aus-/Einlagerung von Prozessen
  - **Low-Level Scheduling** Zuteilung der CPU
- Wir haben *Low-Level-Scheduling*-Verfahren betrachtet
- Auswahl eines geeigneten Verfahrens ist eine Gratwanderung
  - **Benutzer-** und **systemorientierte** Kriterien
  - **Vorabwissen** über die zu Verarbeitenden Prozesse (→ **Betriebsart**)
  - **Komplexität** des Planungsverfahrens
- Reale Systeme kombinieren Verfahren – und passen diese immer wieder an
- Deterministische Planung (Echtzeitbetrieb) ist eine Domäne für sich



Technische  
Universität  
Braunschweig

Institute of Operating Systems  
and Computer Networks  
Reliable System Software



# Betriebssysteme (BS)

Teil C Interaktion und Kommunikation

**Christian Dietrich**

Wintersemester 2024



# Überblick: Teil C Interaktion und Kommunikation

## 8 Speicherbasierte Interaktionen

- 8.1 Einordnung
- 8.2 Motivierendes Szenario, Lost-Update Anomalie
- 8.3 Gegenseitiger Ausschluss mit Mutex
- 8.4 Mutex mit aktivem Warten: Spin Locks
- 8.5 Atomare Komplexbefehle: TAS, CAS, FAA
- 8.6 Mutex mit aktivem Warten: Zwischenfazit
- 8.7 Passives Warten mit Semaphor
- 8.8 Semaphor: Koordinierter Zugriff auf Betriebsmittel
- 8.9 Semaphor: Implementierung
- 8.10 Zusammenfassung

9 Betriebsmittelverwaltung, Synchronisation und Verklemmung

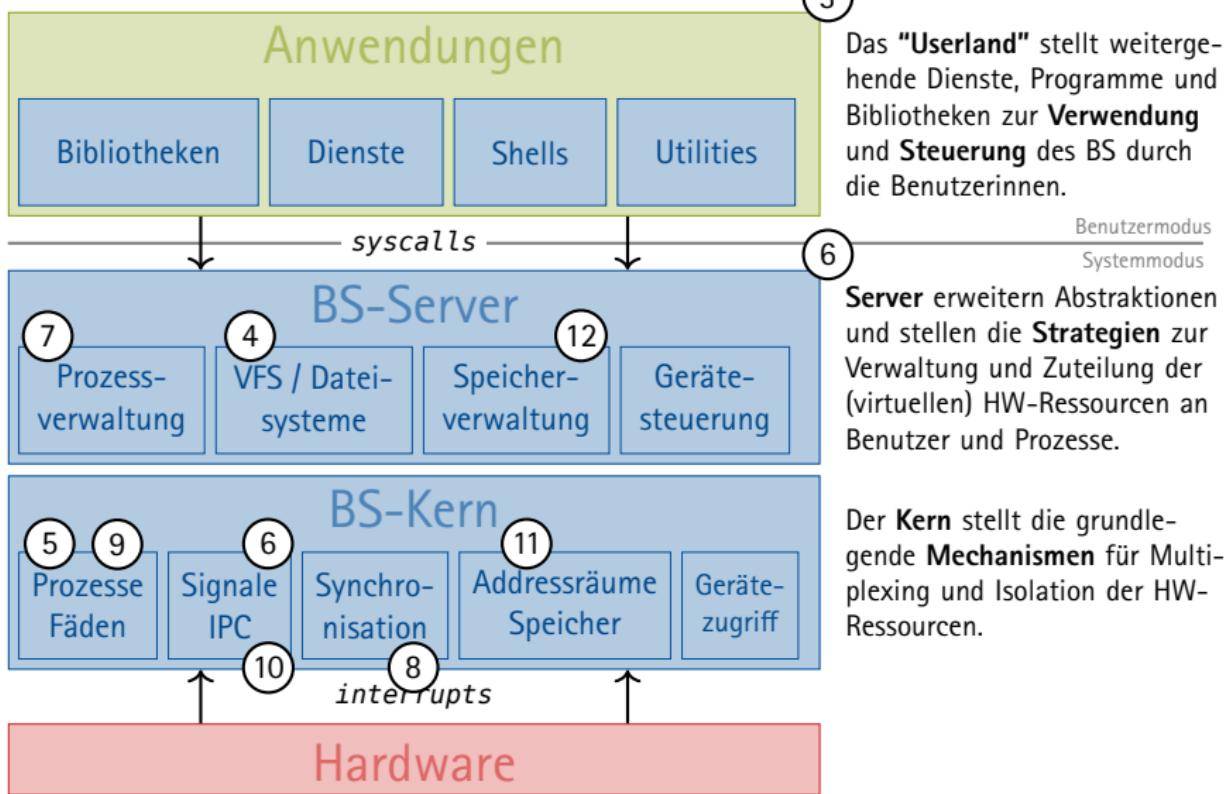
10 Interprozesskommunikation (IPC)



## 8.1 Einordnung

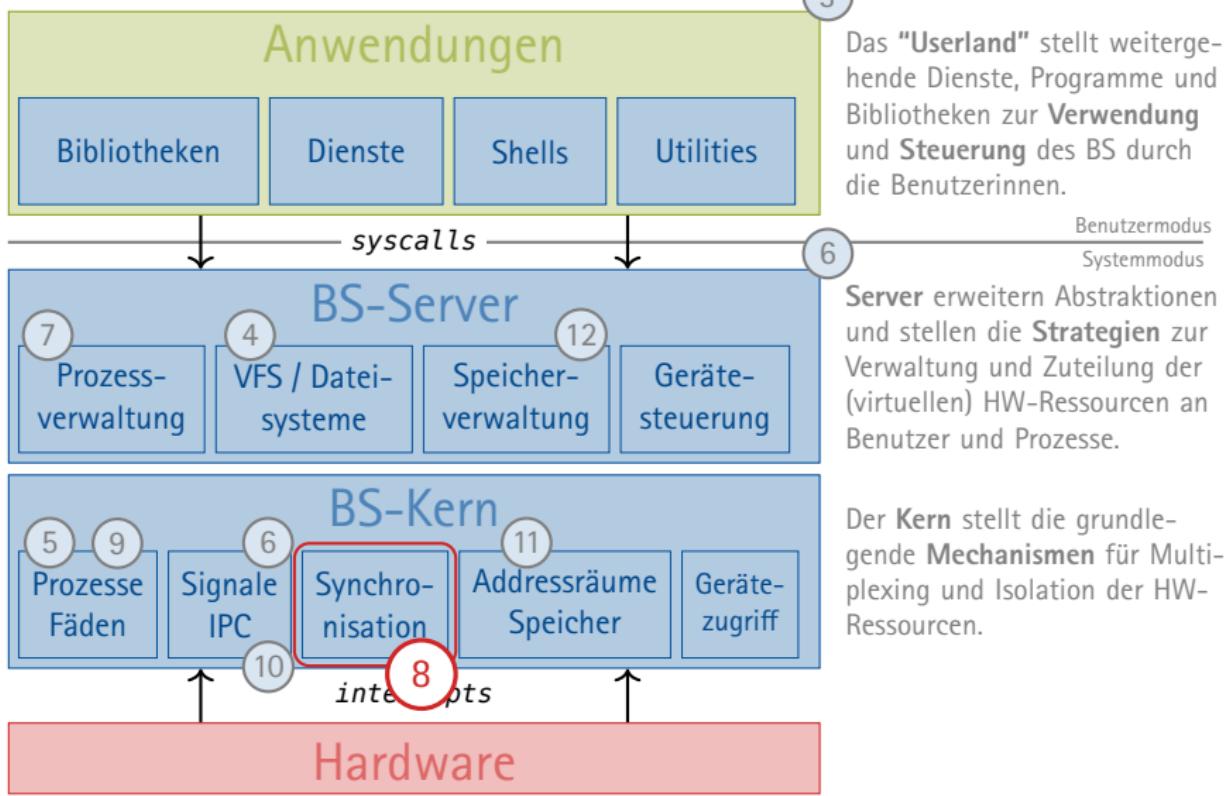


# 8 Speicherbasierte Interaktionen – Einordnung





# 8 Speicherbasierte Interaktionen – Einordnung



Worum geht es in diesem Kapitel?

- **Verstehen** des Interaktionsproblems bei gemeinsam genutzttem Speicher
  - Probleme und Fallstricke auf Softwarebasis, Sprachunterstützung
  - Hardwareunterstützung
  - Betriebssystemunterstützung
- **Erläutern** möglicher Lösungen im Detail
  - Prinzip des Wechselseitigen Ausschlusses
  - Aktives Warten mit *spin locks*
  - Implementierungsvarianten von *spin locks* – nichtfunktionale Auswirkungen
  - Atomare Komplexbefehle der Befehltssatzebene ( $E_2$ ): TAS, CAS, FAA
  - Passives Warten mit Semaphoren

→ Basismechanismen der Synchronisation



# 8.2 Motivierendes Szenario, Lost-Update Anomalie



## ■ **Ausgangsbasis:** Mehrere (pseudo/echt)-parallele Prozesse

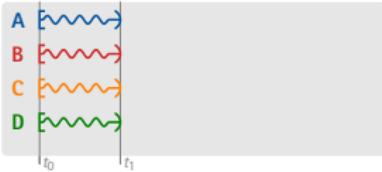
5-10

Mögliche Abläufe von 4 Prozessen **A**, **B**, **C** und **D** auf einem Betriebssystem, abhängig von der Ablaufplanung und den verfügbaren Prozessoren.

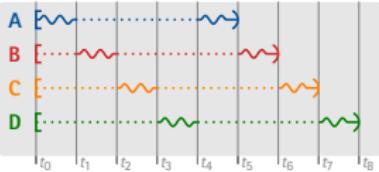
(a) sequentiell (1 Prozessor)



(b) echt parallel (4 Prozessoren)



(c) pseudo-parallel (1 Prozessor)

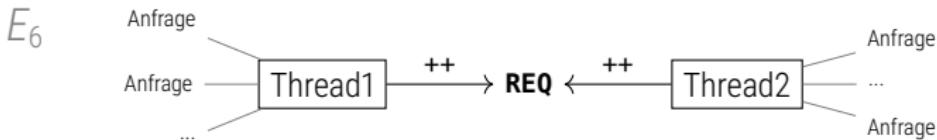


- **Voraussetzung:** Geteilter Zustand in **gemeinsamem Speicher**
  - Fäden (*threads*) in gemeinsamem Adressraum
  - Über Adressräume geteilter Speicher (*shared memory*)
- **Grundannahme:**  $E_2$ -Instruktionen (Befehlssatzebene) wirken atomar
  - bei Speicherzugriffen in **Maschinenwortbreite**

# Speicherbasierte Interaktion: Beispiel

Maschinensprache

- Webserver nimmt über zwei Fäden Anfragen entgegen
- Anfragen werden in globaler Variable **REQ** gezählt



E<sub>5</sub>    `volatile unsigned int REQ = 0; //shared: number of requests`

```
1 // thread 1                                // thread 2
2 do {                                         do {
3     waitForRequest(...);                      waitForRequest(...);
4     ...                                         ...
5     REQ++;                                     REQ++;
6 }
```

E<sub>2</sub>    7 } while (...); } while (...);  
; line 5 in thread 1 ; line 5 in thread 2  
`load REQ, r1` `load REQ, r1`  
`add #1, r1` `add #1, r1`  
`store r1, REQ` `store r1, REQ`

**Hochsprache:**  
1 Operation

**Maschinensprache:**  
3 Operation

# Speicherbasierte Interaktion: Beispiel

- Webserver nimmt über zwei Fäden Anfragen entgegen
- Anfragen werden in globaler Variable **REQ** gezählt

Mögliche Verschränkung



E<sub>5</sub>    `volatile unsigned int REQ = 0; //shared: number of requests`

```
1 // thread 1                                // thread 2
2 do {                                         do {
3     waitForRequest(...);                      waitForRequest(...);
4     ...                                         ...
5     REQ++;                                     REQ++;
6 }
```

Hochsprache:  
1 Operation

E<sub>2</sub> 7 } while (...); } while (...);
; line 5 in thread 1 ; line 5 in thread 2

```
load  REQ, r1    →
context switch
add   #1, r1    ←
store r1, REQ
```

Maschinensprache:  
3 Operation



# Speicherbasierte Interaktion: Beispiel

lost update!

- Webserver nimmt über zwei Fäden Anfragen entgegen
- Anfragen werden in globaler Variable **REQ** gezählt

E<sub>6</sub>



E<sub>5</sub>

```
volatile unsigned int REQ = 0; //shared: number of requests  
1 // thread 1 // thread 2  
2 do { do {  
3     waitForRequest(...); waitForRequest(...);  
4     ...  
5     REQ++; REQ++;  
6     ...  
7 } while (...); } while (...);  
; line 5 in thread 1 ; line 5 in thread 2
```

Hochsprache:  
1 Operation

E<sub>2</sub>

```
load REQ, r1  
add #1, r1  
store r1, REQ  
context switch  
load REQ, r1  
add #1, r1  
store r1, REQ
```

lost upd  
Maschinensprache:  
3 Operation



Variable **REQ** ↪ **gemeinsam genutztes, wiederverwendbares Betriebsmittel**

Ist die gemeinsame Benutzung (*sharing*) wiederverwendbarer Betriebsmittel oder eine logische Abhängigkeit von konsumierbaren Betriebsmitteln gegeben, so wird **Synchronisation** erforderlich ↠ **Prozess muss gegebenenfalls „warten“!**

- Synchronisation beim Zugriff auf eine Integer-Variable?
    - Variablenzugriffe, die nur **einen** Buszyklus (*load/store*) benötigen sind **atomar**.
  - Problem ist der **Lese-Verändere-Schreibe-Zugriff** (*read-modify-write cycle*)
    - **load:** Variable wird in Register gelesen                                       **read**
    - **add:**     **Kopie** im Register wird **verändert**                               **modify**
    - **store:** Variable wird mit **Kopie** überschrieben                               **write**
- Durch Arbeiten auf einer „Registerkopie“ mehr als ein Buszyklus notwendig
- **Lost-Update-Anomalie** möglich

Änderungen von **REQ** **nach** dem  
Lesen werden nicht berücksichtigt  
und „gehen verloren“.



## Atomare Aktion

Eine primitive oder komplexe Aktion, deren Einzelschritte nach außen sichtbar nur im Verbund stattfinden.

- Ansatz: Kritisches Zugriffsmuster als **atomare Aktion** auslegen

```
do {  
    waitForRequest(...);  
    ...  
    atomic {  
        REQ++;  
    }  
    ...  
} while (...);
```

**atomic** ist **kein** gültiges C-Konstrukt, sondern eine semantische Auszeichnung.

- Realisierung durch wechselseitigen Ausschluss (*mutual exclusion*)
  - $E_3$  Definition eines **kritisches Gebiets** auf der Maschinenprogrammebene
    - (Semantische) Unteilbarkeit einer Sequenz von  $E_2$ -Elementaroperationen herstellen
  - $E_2$  Verwendung einer **read-modify-write-Elementaroperation** der Befehlssatzebene
    - Falls vorhanden, die eleganteste Lösung – Löst das Problem direkt auf  $E_2$



### 8.3 Gegenseitiger Ausschluss mit Mutex

## ❖ E<sub>3</sub>-Realisierung: Wechselseitiger Ausschluss

- Zeitweise (im Wechsel) **exklusiven Zugang** zur gemeinsamen Variablen gewähren durch **wechselseitigen Ausschluss** (*mutual exclusion*)
- Ein Prozess, der auf gemeinsame Variable zugreift, ist für diese Zeit im **kritischen Bereich** (*critical region, critical section*)
- Kritischer Bereich wird „überwacht“ durch ein Sperrobject (**mutex**) mit zwei Elemetaroperationen **aquire/release** (oft auch **lock/unlock** bezeichnet)
  - **aquire( mutex )**      Kritisches Gebiet betreten. **Falls belegt, warten bis frei** und dann betreten
  - **release( mutex )**      Kritisches Gebiet verlassen und freigeben.

Es gilt:  $\sum \text{aquire(mutex)} - \sum \text{release(mutex)} \leq 1$

Zu jedem Zeitpunkt darf **maximal ein** in dem durch **mutex** bestimmten kritischen Bereich sein. Alle anderen Prozesse dürfen laufen, müssen aber vor Eintritt in den kritischen Bereich warten. (→ **Synchronisation**)



## ■ Ansatz: Verdrängung unterbinden

- Zum Beispiel durch Sperren der Unterbrechungen im kritischen Gebiet

```
void acquire(void){  
    asm ("cli"); // disable interrupts  
}  
void release(void){  
    asm ("sti"); // enable interrupts  
}
```

→ Abbildung von zweiseitiger Synchronisation auf einseitige Synchronisation

## ■ Möglich, aber **viele Nachteile:**

- Aufwändig: Erfordert **privilegierte Operationen** (Systemaufrufe → [6])
- Fehleranfällig: Prozesse könnten CPU **monopolisieren**  
Verlorene Ereignisse durch hohe Unterbrechungslatenz
- Grobgranular: **Unbeteiligte** Prozesse werden „bestraft“
- Eingeschränkt: **Funktioniert nicht** bei echter Parallelität :-(



## 8.4 Mutex mit aktivem Warten: Spin Locks

# Mutex: Realisierung durch *spin lock*

„Umlaufsperre“

## ■ Naive Implementierung mit binärer Schlossvariable

```
typedef volatile long lock_t; // 0 → free; 1 → taken; initial: free
```

```
void acquire(lock_t* lock){  
    while(*lock) // spin until lock release  
        ; // NOP  
    *lock = 1; // mark as taken  
}  
void release(lock_t* lock){  
    *lock = 0; // mark as free  
}
```

## ■ Anwendung auf unser Beispiel

```
volatile unsigned int REQ = 0; // shared: number of requests  
lock_t REQ_lock = 0; // access mutually exclusive
```

```
1 // thread 1 // thread 2  
2 do { do {  
3     waitForRequest(...); waitForRequest(...);  
4     ... ...  
5     acquire(&REQ_lock); acquire(&REQ_lock);  
6     REQ++; REQ++;  
7     release(&REQ_lock); release(&REQ_lock);  
8     ... ...  
9 } while (...); } while (...);
```

**Funktional** gesehen spielt es keine Rolle, ob die Fäden hier echt parallel oder pseudo-parallel ablaufen. Aus **nichtfunktionalen Gesichtspunkten** (Performanz, Auslastung, Energiebedarf) sind Lösungen mit aktivem Warten jedoch nur bei **echter Parallelität** sinnvoll einsetzbar.



# Mutex: Realisierung durch *spin lock*

Fehlerhaft!

- Ganz so einfach geht es leider nicht!
  - Das kritische Gebiet sei frei (`REQ_LOCK == 0`)
  - Beide Prozesse wollen gleichzeitig (ungünstig quasi-gleichzeitig) hinein
  - Test in *thread 2* erfolgt bevor *thread 1* die Schlossvariable setzen konnte

```
1 // thread 1                                // thread 2
2 void acquire(lock_t* lock){                void acquire(lock_t* lock){
3     while( *lock ) // spin                  while( *lock ) // spin
4     ;                                     // NOP          ;
5 }                                         }           // NOP
6                                                 context switch
7                                                 }           *lock = 1;      // mark as taken
8                                                 }           }
9                                                 }           }
10    *lock = 1;      // mark as taken
11 }
```

Das Ursprungsproblem,  
wiederholt in der „Lösung“...

→ Mehr als ein Prozess im kritischen Gebiet!

# ✳ Mutex: Realisierung durch *spin lock*

Fehlerhaft!

- Ganz so einfach geht es leider nicht!
  - Das kritische Gebiet sei frei (`REQ_LOCK == 0`)
  - Beide Prozesse wollen gleichzeitig (ungünstig quasi-gleichzeitig) hinein
  - Test in *thread 2* erfolgt bevor *thread 1* die Schlossvariable setzen konnte

```
1 // thread 1                                // thread 2
2 void acquire(lock_t* lock){                void acquire(lock_t* lock){
3     while( *lock ) // spin                  while( *lock ) // spin
4     ;                      // NOP           ;
5 }                                         }           // mark as taken
6                                                 context switch
7                                                 }           // mark as taken
8                                                 *lock = 1;    
```

Das Ursprungsproblem,  
wiederholt in der „Lösung“...

→ Mehr als ein Prozess im kritischen Gebiet!



# 8.5 Atomare Komplexbefehle: TAS, CAS, FAA

- Schlossvariable prüfen und setzen ist eine *read-modify-write*-Operation
- Unterstützung durch  $E_2$ -Elementaroperation: *test-and-set*

```
int TAS(long* ref) {  
    atomic {  
        long aux = *ref;  
        *ref = 1;  
    }  
    return aux;  
}
```

Pseudocode der TAS-Operation (Befehlssatzebene)

- TAS schreibt immer, aber die Wertveränderung erfolgt nur bedingt!
  - nur, wenn die Variable den Wert 0 enthielt
  - ansonsten wird der Wert 1 mit 1 überschrieben
- Operationsergebnis ist der Variablenwert vor der Operation
  - Hardware bewirkt gegenseitigen Ausschluss gleichzeitiger Buszugriffe
  - Synchronisation auch mit anderen Busmastern (andere CPU, DMA, ...)

→ **TAS** bewirkt einen atomare Lese-/Schreibzyklus



# Mutex: spin lock-Implementierung mit TAS

test-and-set

## ■ Implementierung mit TAS

```
typedef volatile long lock_t;    // 0 → free; 1 → taken; initial: free

void acquire(lock_t* lock){
    while( TAS(lock) ) // spin until lock release and mark as taken
        ; // NOP
    *lock = 1;          // mark as taken
}
void release(lock_t* lock){
    *lock = 0;          // mark as free
}
```

- Ein TAS-Befehl steht auf (fast) jedem (32-Bit) Prozessor zur Verfügung
  - GCC stellt dafür eine intrinsische Funktion (*compiler intrinsic*) bereit
  - Compiler übernimmt die Abbildung auf konkrete  $E_2$ -Instruktionen

```
// implementation (gcc)
#define TAS(lock) __sync_lock_test_and_set(lock, 1)
```

**Compiler intrinsics** sind als Funktionen „getarnte“ **nichtstandardisierte Erweiterungen** der Sprache, über die Anwendungsentwicklerinnen spezielle Features der Hardware zugänglich gemacht werden.



## ■ Spin-Lock-Implementierung mit TAS

```
void acquire(lock_t* lock){  
    while( TAS(lock) ) // spin until lock release and mark as taken  
        ; // NOP  
}
```

## ■ Funktioniert, aber sehr hohe Cache-Last bei Wettstreit im Multicore-Fall

- angenommen  $n$  Prozessoren wollen in das kritische Gebiet
- unbedingtes Schreiben von TAS löst massiven Datentransfer aus, entweder:
  - write-through cache: 1 Original schreiben und  $n - 1$  Kopien bei anderen Prozessoren aktualisieren
  - write-back cache:  $n - 1$  Kopien invalidieren und 1 Original zum auslösenden Prozessor bewegen
- Dazu kommt noch eine hohe Last auf dem Speicherbus
- auch unabhängige Prozesse (Prozessoren) werden ausgebremst!

→ In nichtfunktionaler Sicht skaliert diese Lösung schlecht!

:-)

- Weitere  $E_2$ -Elementaroperation: compare-and-swap

```
int CAS(long* var, long val, long new) {
    atomic {
        if(*var == val) { // Falls var erwarteten Wert hat
            *var = new;      // setze auf neuen Wert
            return 1;         // und gebe 1 zurück
        }
    }
    return 0;           // sonst gebe 0 zurück
}
```

Pseudocode der CAS-Operation

- Es wird nur geschrieben wenn `*var` den „erwarteten“ Wert hat
  - Operationsergebnis ist `true` (1), wenn geschrieben wurden, sonst `false` (0)
- CAS ist weniger verbreitet als TAS, aber auf vielen Plattformen verfügbar

```
// implementation (gcc)
#define CAS(var,val,new) __sync_lock_compare_and_swap(var,val,new)
```



## ■ Spin-Lock-Implementierung mit CAS

```
typedef volatile long lock_t;    // 0 → free; 1 → taken; initial: free

void acquire(lock_t* lock){
    while( !CAS(lock, 0, 1) ) // spin until lock release then mark as taken
        ; // NOP
}
void release(lock_t* lock){
    *lock = 0;      // mark as free
}
```

## ■ Es wird nur noch bei Belegung des Mutex geschrieben

- deutlich freundlicher bezüglich der Cachekohärenz :-)
- aber immer noch ungünstig viele Zugriffe auf den Speicherbus :-()

→ In nichtfunktionaler Sicht skaliert diese Lösung **immer noch nicht gut!**

# ✳ Mutex: Bessere Implementierung mit CAS

*spin on read*

- Spin-Lock-Implementierung mit CAS, **aber erst bei reeller Chance**

```
typedef volatile long lock_t;    // 0 → free; 1 → taken; initial: free

void acquire(lock_t* lock){
    do {
        while( *lock )           // first spin for lock release
        ;
    } while( !CAS(lock, 0, 1) ) // then try to acquire lock
}
void release(lock_t* lock){
    *lock = 0;                 // mark as free
}
```

- Weniger Wettstreit auf dem Speicherbus durch *spin on read*

- innere `while`-Schleife arbeitet auf dem Cache (ohne ihn zu ändern)
  - in dieser Zeit keine Datenzugriffe, keine Bus-Sperren

→ Diese Lösung skaliert erheblich besser

:-(

→ Allerdings ermöglichen alle bisherigen Lösungen ein **Verhungern**

:-|



# Mutex: Fairness mit „Bäckereialgorithmus“

*ticket lock*

- Bisherige Implementierungen ermöglichen **Verhungern (starvation)**
  - Bei hohem Wettstreit am Eingang gibt es keine Garantie „durch“ zu kommen
  - Reihenfolge ist „zufällig“ (durch externe Effekte bestimmt)
- Ansatz: „Bäckereialgorithmus“ → Am Eingang „Wartemarkte“ ziehen

```
typedef volatile struct {
    long next;      // next ticket    initial: 0
    long this;      // ticket being currently served  initial: 0
} lock_t;

void acquire(lock_t* lock){
    long self = FAA(&lock->next, 1); // get service number
    while( self > lock->this )        // wait until being served
    ;
}
void release(lock_t* lock){
    lock->this++; // next one, please!
}
```

FAA steht für *fecht-and-add*  
(Details nächste Folie)

Jeder wartet nur auf seinen  
unmittelbaren Vorgänger:  
Verwerfen des *ticket*  
führt zum **Deadlock**.

- Diese Lösung verhindert Verhungern und skaliert gut
- Voraussetzung: Keiner der Wartenden „steigt aus“

:)

:-|

- Weitere *E*<sub>2</sub>-Elementaroperation: *fetch-and-add*

```
long FAA(long* var, long add) {  
  
    atomic {  
        long aux = *var; // add atomically to var  
        *var += add;  
    }  
    return aux; // return old value  
}
```

Pseudocode der FAA-Operation

- Operationsergebnis ist Variablenwert vor der Erhöhung
- FAA ist weniger verbreitet als CAS und TAS
  - Einige Plattformen bieten nur eine *fetch-and-increment*-Operation an (erhöhe um 1), was hier aber ebenfalls reichen würde

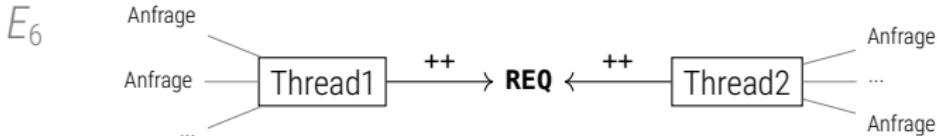
```
// implementation (gcc)  
#define FAA(var,add) __sync_lock_fetch_and_add(var,add)
```



# Direkte Verwendung von FAA

fetch-and-add

Mit **FAA** könnten wir unser Beispielproblem nun auch direkt lösen:



E<sub>5</sub>

```
volatile unsigned long REQ = 0; //shared: number of requests
1 // thread 1
2 do {
3     waitForRequest(...);
4     ...
5     FAA( &REQ, 1 )
6     ...
7 } while (...);
```

// thread 2  
do {  
 waitForRequest(...);  
 ...  
 FAA( &REQ, 1 )  
 ...  
} while (...);

Hochsprache:  
1 Operation

E<sub>2</sub>

; line 5 in thread 1  
faa #1, REQ

; line 5 in thread 2  
faa #1, REQ

→ Die mit Abstand **eleganteste und effizienteste** Lösung!  
Leider ist sie jedoch **nicht verallgemeinerbar**

Maschinensprache:  
1 Operation



# 8.6 Mutex mit aktivem Warten: Zwischenfazit



# Speicherbasierte Interaktionen: Zwischenfazit

- Prozess(or)-Interaktion über gemeinsamen Speicher
  - Konsistenzprobleme bei nichtatomarem Zugriff → kritisches Gebiet
  - Gemeinsamer Speicher → exklusives Betriebsmittel
- Synchronisation erforderlich
- Ideal ist die Verwendung atomarer Maschinenbefehle
  - einzelner *read/write-Befehl* ist auf der Maschinenebene  $E_2$  atomar
  - spezielle *read-modify-write*-Befehle auf  $E_2$  sind ebenfalls atomar
  - TAS, CAS, FAA – architekturabhängig, im Compiler durch *intrinsics* unterstützt
- Lösungen nicht verallgemeinerbar
- Alternative: Expliziter gegenseitiger Ausschluss (*mutual exclusion*, Mutex)
  - Mutex mit *spin lock*, *spin-on-read lock*, *ticket lock*
  - Protokoll mit `lock()`, `unlock()` und Schlossvariable, aktives Warten in `lock()`
  - ineffizient bei virtuellem Prozessor (Prozess „verbrät“ Zeitscheibe)
  - Implementierung kritisch ↽ nichtfunktionale Effekte durch Cache, Speicherbus

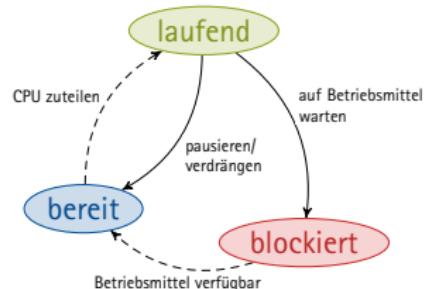


Bisherige Implementierungen verwenden **aktives Warten**

↔ Ineffizient!

- Grundsätzlich und immer ineffizient bei Pseudoparallelität (Uniprozessor)
  - während Eintretender wartet, *kann* Besitzer keinen Fortschritt erzielen
  - Eintretender „verbrät“ Zeitscheibe, wird ggfs. „bestraft“ (z. B. bei MLFQ → 7-34)
- Häufig aber nicht immer ineffizient bei echter Parallelität (Multiprozessor)
  - abhängig von den Eigenschaften des kritischen Gebiets (Dauer, Frequenz, Wettstreit, ...)

- **Ansatz:** Passives Warten (vgl. → 5-14 )
  - Ein vor dem kritischen Gebiet wartender Prozess wechselt in den **blockiert**-Zustand
    - und gibt damit dem Prozessor freiwillig ab
  - Der das kritische Gebiet verlassende Prozess versetzt Wartende in den **bereit**-Zustand
    - so dass sie den Eintritt erneut versuchen können
- Modifizieren der Prozesszustände → privilegiert Operation
- Implementierung der Synchronisationsprimitiven im Kern





# 8.7 Passives Warten mit Semaphor



# Semaphor

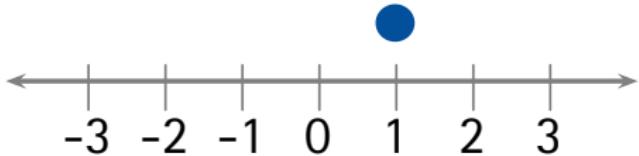
- Vorschlag von Dijkstra (1964) [11]: Semaphor
  - „Signalmast“, der den Einlass steuert
  - Primitive des BS für Erwerb/Abgabe von Betriebsmitteln
- Spezielle **ganzzahlige Variable** mit zwei Operationen
  - **P** (*prolaag*), auch **down**, *wait*, *acquire* – oder sem\_wait<sup>(3)</sup>
    - verringert den Wert des Semaphors  $s$  um 1
    - blockiere Prozess am Semaphor, falls  $s < 0$
  - **V** (*verhoog*), auch **up**, *signal*, *release* – oder sem\_post<sup>(3)</sup>
    - erhöht den Wert des Semaphors  $s$  um 1
    - stellt ggfs. am Semaphor blockierten Prozess **bereit**
- Ursprünglich definiert als      **binärer Semaphor**      mit  $s \in \{0, 1\}$
- Später verallgemeinert zu      **zählender Semaphor**      mit  $s \in \mathbb{Z}$       [10]





# Zählender Semaphor: Grundprinzip

*waiting:*



*init*

$s = 1$

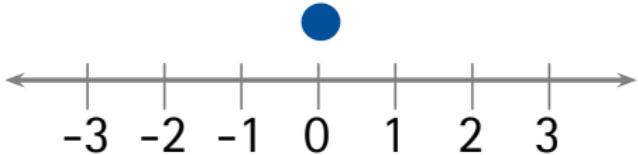
Semaphor kann mit beliebigem Wert initialisiert werden.

Dieser symbolisiert in der Regel der Anzahl der verfügbaren Betriebsmittel.



# Zählender Semaphor: Grundprinzip

*waiting:*



*init*

$s = 1$

*P1*

*down(s)*

Prozess P1 hat den Semaphore vermindert (und damit ein Betriebsmittel erworben).



# Zählender Semaphor: Grundprinzip

*waiting:*

P2



*init*

$s = 1$

*P1*

*down(s)*

*P2*

*down(s) -> block P2*

Prozess P2 hat den Semaphor vermindert. Da der Semaphorwert bereits 0 war, blockiert P2 und wird dazu in eine mit dem Semaphor verknüpfte Warteliste eingetragen.

# Zählender Semaphor: Grundprinzip



|             |                                              |
|-------------|----------------------------------------------|
| <i>init</i> | $s = 1$                                      |
| <i>P1</i>   | <i>down(s)</i>                               |
| <i>P2</i>   | <i>down(s)</i> $\rightarrow$ block <i>P2</i> |
| <i>P3</i>   | <i>down(s)</i> $\rightarrow$ block <i>P3</i> |

Auch Prozess P3 blockiert am Semaphor.

Negativer Zählerwert entspricht der Anzahl der wartenden Prozesse (implementierungsabhängig).



# Zählender Semaphor: Grundprinzip

*waiting:*

P3



Prozess P1 hat den Semaphor erhöht (ein Betriebsmittel abgegeben). Prozess P2 wurde der Warteliste entnommen (und hat das Betriebsmittel erhalten).

*init*       $s = 1$

P1            *down(s)*

P2            *down(s)*     $\rightarrow$  block P2

P3            *down(s)*     $\rightarrow$  block P3

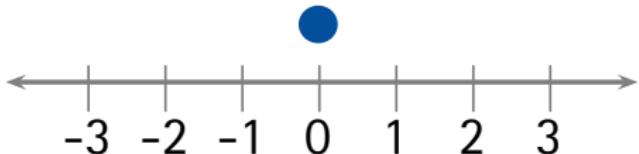
P1            *up(s)*       $\rightarrow$  unblock P2

Die Entnahmestrategie der Warteliste (hier: FIFO) ist durch Semaphor nicht definiert. Idealerweise entspricht sie der Strategie der Bereitliste des Planers  $\hookrightarrow$  7.



# Zählender Semaphor: Grundprinzip

*waiting:*



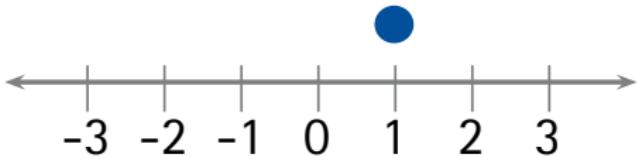
- |             |                                      |
|-------------|--------------------------------------|
| <i>init</i> | $s = 1$                              |
| <i>P1</i>   | <i>down(s)</i>                       |
| <i>P2</i>   | <i>down(s)</i> <i>-&gt; block P2</i> |
| <i>P3</i>   | <i>down(s)</i> <i>-&gt; block P3</i> |
| <i>P1</i>   | <i>up(s)</i> <i>-&gt; unblock P2</i> |
| <i>P2</i>   | <i>up(s)</i> <i>-&gt; unblock P3</i> |

Prozess P2 hat den Semaphor erhöht (ein Betriebsmittel abgegeben). Prozess P3 wurde der Warteliste entnommen (und hat das Betriebsmittel erhalten).



# Zählender Semaphor: Grundprinzip

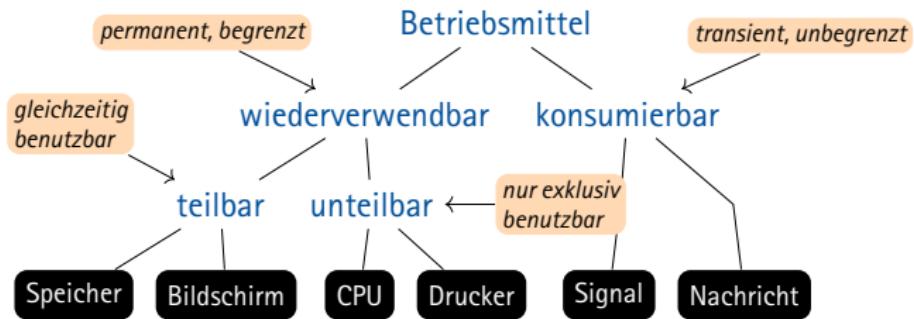
*waiting:*



- |             |                                              |
|-------------|----------------------------------------------|
| <i>init</i> | $s = 1$                                      |
| <i>P1</i>   | <i>down(s)</i>                               |
| <i>P2</i>   | <i>down(s)</i> $\rightarrow$ <i>block P2</i> |
| <i>P3</i>   | <i>down(s)</i> $\rightarrow$ <i>block P3</i> |
| <i>P1</i>   | <i>up(s)</i> $\rightarrow$ <i>unblock P2</i> |
| <i>P2</i>   | <i>up(s)</i> $\rightarrow$ <i>unblock P3</i> |
| <i>P2</i>   | <i>up(s)</i>                                 |

Prozess P2 hat den Semaphor ein weiteres Mal erhöht (ein Betriebsmittel produziert).

- Ein Programmablauf ist möglich, wenn
  1. er dem Betriebssystem explizit gemacht worden ist und
  2. alle von ihm benötigten **Betriebsmittel** (real/virtuell) verfügbar sind.
- Klassifikation von **Betriebsmitteln**: (vgl. [15, 16])



Ist die gemeinsame Benutzung (*sharing*) wiederverwendbarer Betriebsmittel oder eine logische Abhängigkeit von konsumierbaren Betriebsmitteln gegeben, so wird **Synchronisation** erforderlich  $\rightsquigarrow$  **Prozess muss gegebenenfalls „warten“!**



### 8.8 Semaphor: Koordinierter Zugriff auf Betriebsmittel

- **Wiederverwendbare Betriebsmittel** sind nur begrenzt verfügbar
    - Typisch für **Hardware**, wie CPU oder Speicher, aber auch **gemeinsame Variablen**
    - Einige sind **aufteilbar** für eine **gleichzeitige Benutzung** durch **mehrere** Prozesse.
    - **Unteilbare** müssen **einzelnen Prozessen** zeitweise **exklusiv** zugeordnet werden.
  - **Konsumierbare Betriebsmittel** werden „generiert“ und „verbraucht“.
    - Typisch für **E/A-Operationen** → jede Art von Interaktion mit der Umwelt.
    - Das Ergebnis einer E/A-Operation ist ein konsumierbares Betriebsmittel.
- Prozesse sind in ihrem Fortschritt **logisch abhängig** von konsumierbaren Betriebsmitteln (z. B. Verfügbarkeit der Eingabedaten).
- Bei der Simultanverarbeitung liegen sie zusätzlich im **Wettstreit** um wiederverwendbare Betriebsmittel (z. B. CPU oder Speicher).

Semaphore eignet sich in beiden Fällen als Koordinierungsprimitive!

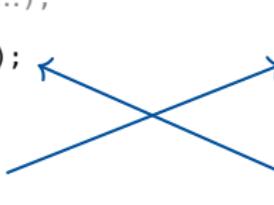
## ■ Koordinierter Zugriff auf ein **wiederverwendbares Betriebsmittel**

- gegenseitiger Ausschluss
  - Betreten des kritischen Gebiets
  - Verlassen des kritischen Gebiets
  - Initialisierung mit dem Wert 1
- Verwendung als binärer Semaphor
- down() – ggfs. passiv Warten
- up() – ggfs. Wartenden wecken
- das kritische Gebiet ist initial frei

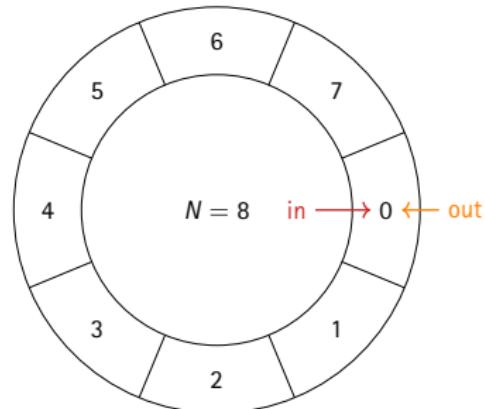
## ■ Anwendung auf unser Beispiel

```
volatile unsigned int REQ = 0; // shared: number of requests
semaphore_t REQ_mutex = 1; // semaphor for mutual exclusion
```

```
1 // thread 1                                // thread 2
2 do {                                         do {
3     waitForRequest(...);                     waitForRequest(...);
4     ...                                         ...
5     down(&REQ_mutex);                      down(&REQ_mutex);
6     REQ++;                                 REQ++;
7     up(&REQ_mutex);                       up(&REQ_mutex);
8     ...                                         ...
9 } while (...);                            } while (...);
```



- Koordinierter Zugriff auf **konsumierbare Betriebsmittel**
    - Erzeuger produziert Elemente → Verbraucher konsumiert Elemente
    - Entkopplung durch **Ringpuffer** (*bounded buffer*)
  - Erzeuger
    - legt Element im Puffer ab
    - **wartet**, falls Puffer voll
  - Verbraucher
    - entnimmt Element aus dem Puffer
    - **wartet**, falls Puffer leer
  - Semantisch **ein** Betriebsmittel, in der Realisierung **zwei**
    - Erzeuger: produziert **Elemente**, verbraucht **Pufferplätze**
    - Verbraucher: produziert **Pufferplätze**, verbraucht **Elemente**
- Realisierung mit **zwei zählenden Semaphoren**



# Ringpuffer-Implementierung mit zählender Semaphor

```
#define N 8                                // ring buffer with 8 elements
char Buffer[ N ];                          // and FIFO access
volatile long in = 0, out = 0;             // next index to insert, remove
semaphore_t elem = 0, slots = N;          // available elements, buffer slots
```

```
void insert( char item ){
    down(&slots);
    Buffer[FAA(&in, 1) % N] = item;
    up(&elem);
}
```

```
char remove( void ){
    char item;
    down(&elem);
    item = Buffer[FAA(&out, 1) % N];
    up(&slots);
    return item;
}
```

- remove() blockiert, falls Puffer leer.
- wird aufgeweckt von insert().

- insert() blockiert, falls Puffer voll.
- wird aufgeweckt von remove().

# Ringpuffer-Implementierung: Indexvariable

```
void insert( char item ){  
    down(&slots);  
    Buffer[FAA(&in, 1) % N] = item;  
    up(&elem);  
}  
  
char remove( void ){  
    char item;  
    down(&elem);  
    item = Buffer[FAA(&out, 1) % N]  
    up(&slots);  
    return item;  
}
```

- Auch der Zugriff auf die Indexvariablen `in` und `out` ist kritisch!
  - bei gleichzeitigem Zugriff mehrerer Erzeuger/Verbraucher (*lost update!*)
  - hier **elegant umschifft** mit der **FAA**-Primitive
- „Klassische“ Implementierung benutzt dazu einen dritten Semaphor
  - Binären Semaphor als Mutex für Veränderung der Indexvariablen
  - Faktisch ein „Nachbau“ von FAA mit expliziter Synchronisation

```
long FAA( long* var, long add ){  
    static semaphore_t mutex = 1;           // mutex object (static!)  
    down( mutex );                      // enter critical section  
    long aux = *var;  
    *var += add;  
    up( mutex );                        // leave critical section  
    return aux;  
}
```



## 8.9 Semaphor: Implementierung

- Implementierung von **up()** und **down()** bildet selber ein kritisches Gebiet!

```
typedef volatile int semaphore_t;

void down(semaphore_t* sema){
    atomic {
        *sema--;
        if( *sema < 0 )
            block( sema );
    }
}

void up(semaphore_t* sema){
    atomic {
        *sema++;
        if( *sema <= 0 )
            wakeup( sema );
    }
}
```

### Vielfältige Probleme – sind wir wieder am Anfang?

- *lost update* Gleichzeitiges Ausführen von **down()** könnte **mehr Prozesse** passieren lassen als der Semaphorwert (**sema**) erlaubt.
- *lost update* Gleichzeitiges Zählen kann Werte hinterlassen, die **nicht** der Anzahl der ausgeführten **up() / down()**-Operationen entspricht.
- *lost wakeup* Gleichzeitiges Auswerten der Wartebedingung in **down()** und Hochzählen in **up()** kann bewirken, dass **ein Prozess schlafen geht**, obwohl die Wartebedingung nicht mehr gilt.

- Implementierung von **up()** und **down()** bildet selber ein kritisches Gebiet!

```
typedef volatile int semaphore_t;

void down(semaphore_t* sema){
    atomic {
        *sema--;
        if( *sema < 0 )
            block( sema );
    }
}
```

```
void up(semaphore_t* sema){
    atomic {
        *sema++;
        if( *sema <= 0 )
            wakeup( sema );
    }
}
```

- Implementierung der Unteilbarkeit obliegt hier dem BS-Kern. Varianten:
  - **atomic** ↪ Unterbrechungssperren ↪ 8-12
    - falls **up()** auch aus dem Kontext von Unterbrechungsbehandlern aufrufbar sein soll
  - **atomic** ↪ nichtverdrängbares kritisches Gebiet (*non-preemptible critical section*, NPSC)
    - Faden im kritischen Gebiet bekommt den Prozessor nicht entzogen
  - **atomic** ↪ im Multiprozessorfall **zusätzlich**: *spin lock* ↪ ??
- Oder durch den geschickten Einsatz atomarer Befehle (knifflig!)



## 8.10 Zusammenfassung



# Zusammenfassung: Speicherbasierte Interaktionen

- **Ausgangsbasis:** (Pseudo-)parallele Prozesse mit gemeinsamem Zustand
  - **Problem:**
    - Nichtatomare, Zugriffe auf Zustandsvariablen
      - Schreibzugriffe können verloren gehen
      - *Lost-Update*-Anomalien
    - Zugriff als **kritisches Gebiet** auffassen ↵ Koordinierung
  - **Ansätze:**
    - Spezialbefehle, aktives Warten, passives Warten
      - TAS, CAS, FAA – hilfreich, aber nicht immer ausreichend
      - Aktiv Warten mit *spin locks* – nur bei echter Parallelität
      - Semaphore – passiv Warten mit Hilfe des Betriebssystems
- **Hohe Sensitivität** bezüglich **nichtfunktionaler Eigenschaften**
- Konzeptionell: Wann lohnt aktives, wann passives Warten?
  - Technisch: Implementierungsdetails haben großen Einfluss

## 8 Speicherbasierte Interaktionen

## **9 Betriebsmittelverwaltung, Synchronisation und Verklemmung**

- 9.1 Einordnung
- 9.2 Betriebsmittelverwaltung
- 9.3 Verklemmungen
- 9.4 Modellierung
- 9.5 Philosophenproblem
- 9.6 Betriebssystemstrategien
- 9.7 Zusammenfassung

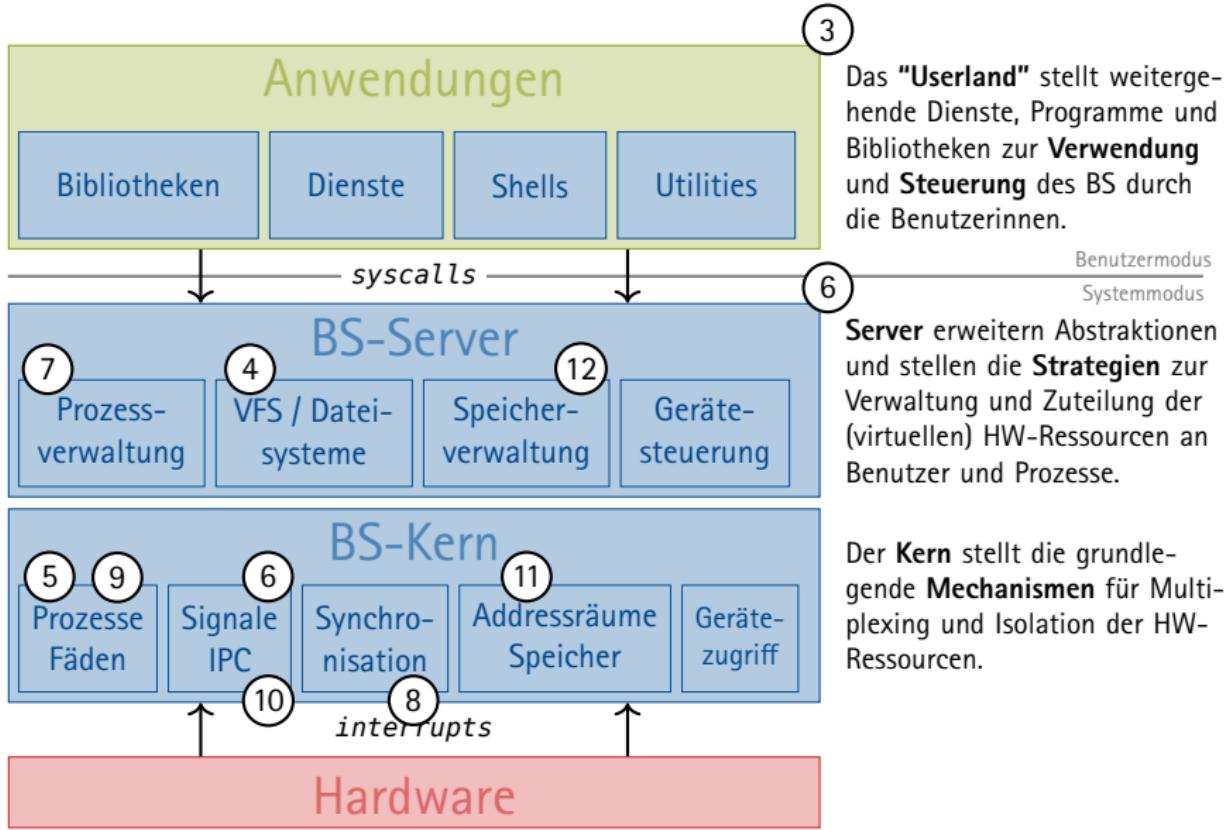
## 10 Interprozesskommunikation (IPC)



### 9.1 Einordnung

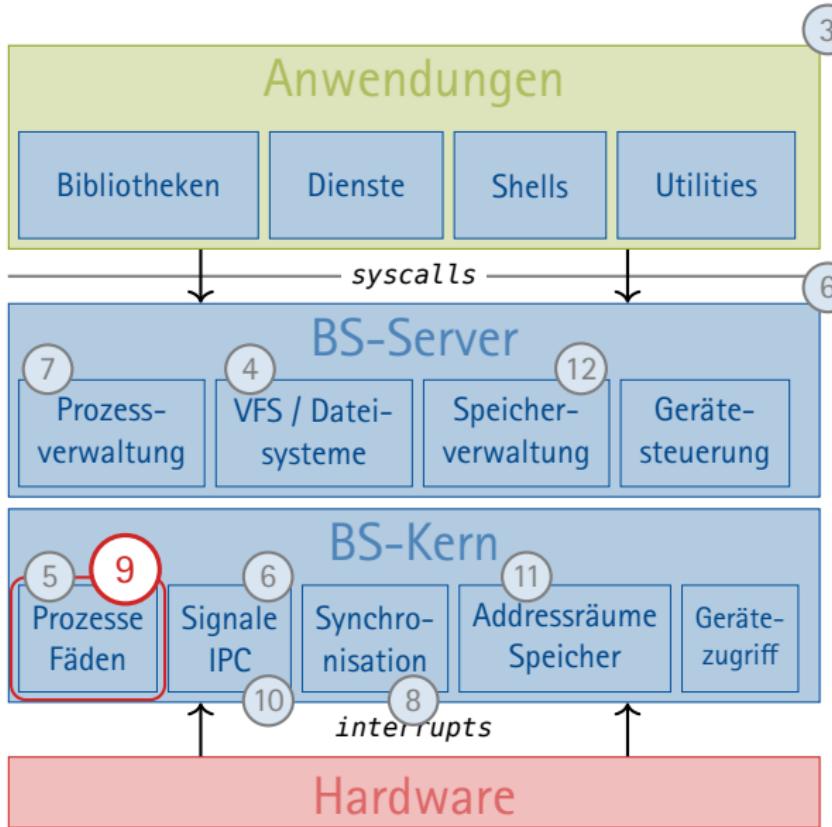


## 9 Betriebsmittelverwaltung, Synchronisation und Verklemmung – Einordnung





## 9 Betriebsmittelverwaltung, Synchronisation und Verklemmung – Einordnung



Das "Userland" stellt weitergehende Dienste, Programme und Bibliotheken zur **Verwendung** und **Steuerung** des BS durch die Benutzerinnen.

Benutzermodus  
Systemmodus

Server erweitern Abstraktionen und stellen die **Strategien** zur Verwaltung und Zuteilung der (virtuellen) HW-Ressourcen an Benutzer und Prozesse.

Der Kern stellt die grundlegende **Mechanismen** für Multiplexing und Isolation der HW-Ressourcen.



### Worum geht es in diesem Kapitel?

- **Verstehen** der Entstehung von Stillstand (*stalemate*) gekoppelter Prozesse, hervorgerufen durch eine ungünstige **Betriebsmittelzuteilung**
  - **Deadlock** („Totsperre“) – das kleinere Übel, da zumindest erkennbar
  - **Livelock** („Lebendsperre“) – das größere Übel, da kaum erkennbar
  - Definition und Entstehung
- **Erläutern** von Beispielen und Lösungen
  - Notwendige und Hinreichende Bedingungen für Deadlocks
  - Deadlock-Modellierung
  - Problem der speisenden Philosophen (und Lösungen)
  - Techniken der Vorbeugung, Vermeidung, Erkennung

### → Basismechanismen der Synchronisation

⌘ Stillstand durch „Tötliche Umarmung“ (*Deadly Embrace*)



Source: National Geographic

D.MAITLAND/WILDLIFE PHOTOGRAPHER OF THE YEAR 2008

### „Tötliche Umarmung“ (*deadly embrace*)

Eine Situation, in der gekoppelte Prozesse gegenseitig die Aufhebung einer Wartebedingung entgegensehen, diese aber durch Prozesse eben dieses Systems selbst aufgehoben werden müsste.

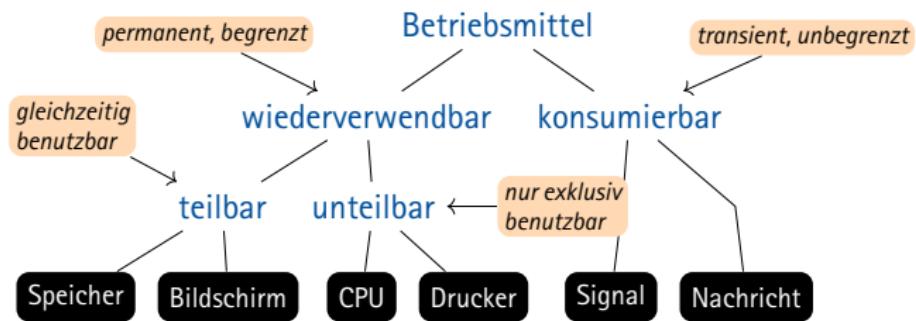
- Die Bedingung sagt etwas zur Verfügbarkeit eines **Betriebsmittels** aus
- Sie kann entstehen [5]:
  - obwohl kein einziger Prozess mehr als die insgesamt verfügbare Menge von Betriebsmitteln benötigt.
  - unabhängig davon, ob Betriebsmittelzuteilung in der Verantwortlichkeit des Betriebssystems oder des Anwendungsprogramms selbst liegt.
    - Stillstandssituation entsteht aus der **Verschränkung** der Prozessabläufe.
- Das Warten kann **passiv** (*deadlock*) oder **aktiv** (*livelock*) stattfinden.





### 9.2 Betriebsmittelverwaltung

- Ein Programmablauf ist möglich, wenn
  1. er dem Betriebssystem explizit gemacht worden ist und
  2. alle von ihm benötigten **Betriebsmittel** (real/virtuell) verfügbar sind.
- Klassifikation von **Betriebsmitteln**: (vgl. [15, 16])



Ist die gemeinsame Benutzung (*sharing*) wiederverwendbarer Betriebsmittel oder eine logische Abhängigkeit von konsumierbaren Betriebsmitteln gegeben, so wird **Synchronisation** erforderlich  $\rightsquigarrow$  **Prozess muss gegebenenfalls „warten“!**

- **Wiederverwendbare Betriebsmittel** sind nur begrenzt verfügbar
    - Typisch für **Hardware**, wie CPU oder Speicher
    - Einige sind **aufteilbar** für eine **gleichzeitige Benutzung** durch **mehrere** Prozesse.
    - **Unteilbare** müssen **einzelnen Prozessen** zeitweise **exklusiv** zugeordnet werden.
  - **Konsumierbare Betriebsmittel** werden „generiert“ und „verbraucht“.
    - Typisch für **E/A-Operationen** → jede Art von Interaktion mit der Umwelt.
    - Das Ergebnis einer E/A-Operation ist ein konsumierbares Betriebsmittel.
- Prozesse sind in ihrem Fortschritt **logisch abhängig** von konsumierbaren Betriebsmitteln (z. B. Verfügbarkeit der Eingabedaten).
- Bei der Simultanverarbeitung liegen sie zusätzlich im **Wettstreit** um wiederverwendbare Betriebsmittel (z. B. CPU oder Speicher).

“ Der Zweck eines Betriebssystems [besteht] in der **Verteilung von Betriebsmitteln** auf sich bewerbende Benutzer. ”

Hansen 1977: *Betriebssysteme* [14]

## ■ **Buchführung** über die im Rechensystem vorhandenen Betriebsmittel

- Art, Klasse
- Zugriffsrechte, Prozesszuordnung, Nutzungszustand und -dauer

## ■ **Steuerung** der Verarbeitung von Betriebsmittelanforderungen

- Entgegennahme, Überprüfung (z.B. der Zugriffsrechte)
- Einplanung der Nutzung durch Prozesse
- Zuteilung (Einlastung) von Betriebsmittel
- Freigabe von Prozessen benutzter Betriebsmittel

*request* ←

*grant* →  
*release* ←

## ■ **Betriebsmittelentzug**

- Zurücknahme der Betriebsmittel, die von einem „aus dem Ruder geratenen“ Prozess belegt werden
- Bei Virtualisierung reicht der Entzug des realen Betriebsmittels

*revoke* →

■ Durchsetzen der gewählten **Betriebsstrategie**,  
in Abhängigkeit von **Betriebsart** und **Planungszielen**:

→ 7-11

- Konfliktfreie Abarbeitung der anstehenden Aufträge
- Korrekte Bearbeitung der Aufträge in endlicher Zeit
- Gleichmäßige und maximale Auslastung der Betriebsmittel
- Hoher Durchsatz, geringe Durchlaufzeit, hohe Ausfallsicherheit
- ...

■ **Betriebsmittelzugang** frei von Verhungern / Verklemmung

- **Verhungern** Zeitweilige Benachteiligung **einzelner** Prozesse. Das Prozesssystem macht jedoch Fortschritt, steht nicht still.
- **Verklemmung** Irreversible  **gegenseitige Blockierung** von Prozessen. Das Prozesssystem macht keinen Fortschritt, steht still.

→ Gemeinsames Merkmal ist der **Wettstreit** um Betriebsmittel

→ Steuerndes Element ist die Form der **Zuteilung** der Betriebsmittel



## statisch

- vor Laufzeit oder vor einem Laufzeitabschnitt
- Anforderung **aller** (im Abschnitt) benötigten Betriebsmittel
- Zuteilung der Betriebsmittel erfolgt ggf. lange vor ihrer eigentlichen Benutzung
- Freigabe **aller** belegten Betriebsmittel mit Laufzeit(abschnitt)ende

## dynamisch

- zur Laufzeit, in beliebigen Laufzeitabschnitten
- Anforderung des jeweils benötigten Betriebsmittels **bei Bedarf**
- Zuteilung des jeweiligen Betriebsmittels erfolgt „im Moment“ seiner Benutzung
- Freigabe eines belegten Betriebsmittels, sobald **kein Bedarf** mehr besteht

→ Risiko einer nur **suboptimalen Auslastung** der Betriebsmittel

→ Risiko der **Verklemmung** von abhängigen Prozessen



### 9.3 Verklemmungen



## Verklemmung (*Deadlock*)

vgl. [22, S. 235]

Eine **Deadlock** liegt vor, wenn eine Gruppe von Prozessen wechselseitig auf den Eintritt von Bedingungen **wartet**, die nur durch andere Prozesse der Gruppe hergestellt werden können.

→ „Tödliche Umarmung“ gekoppelter Prozesse im **BLOCKED**-Zustand

## Lebendsperre (*Livelock*)

Ein **Livelock** ist eine Variante des Deadlock, bei der die Prozesse zwar nicht im **BLOCKED**-Zustand sind, jedoch dennoch keinen Fortschritt im Sinne der Programmausführung erzielen können.

→ „Tödliche Umarmung“ gekoppelter Prozesse im **READY/RUNNING**-Zustand

- Kennzeichnend ist, dass die Situation nur „von außen“ gelöst werden kann
  - beteiligte Prozesse können sie „aus eigener Kraft“ weder erkennen noch beheben
- **Livelocks** sind das **größere Übel**, da auch von außen kaum erkennbar
  - Programmzähler der beteiligten Prozesse verändert sich ständig



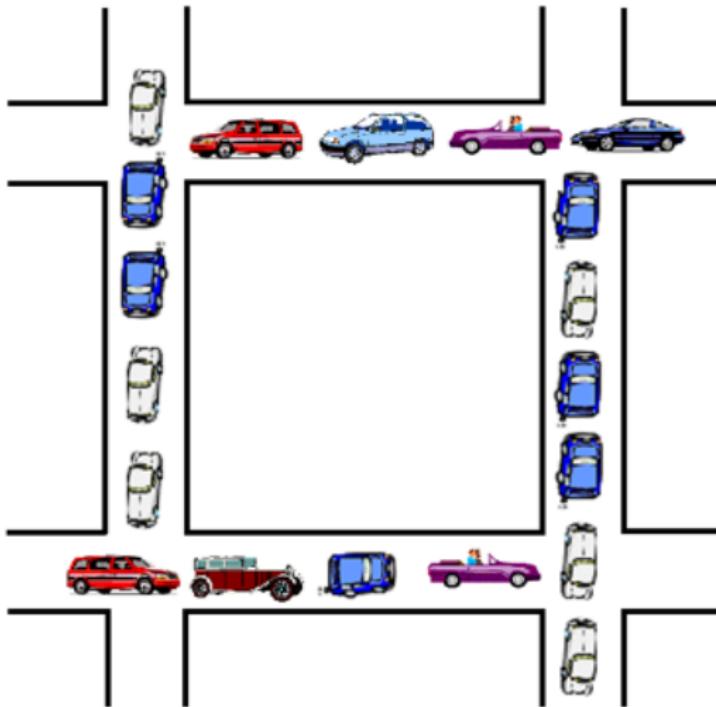
# Verklemmungen: Deadlocks und Livelocks

Beispiele

## ■ Verkehrs-Deadlock

- Betriebsmittel:  $N$  Plätze
- Prozesse:  $N$  Autos

Freigeben des Platzes  $n$  hängt ab von Verfügbarkeit des Platzes  $n + 1$





# Verklemmungen: Deadlocks und Livelocks

Beispiele

Alice und Bob haben verabredet, um 12:00 miteinander zu telefonieren...

12:00



Let's call Alice!



Let's call Bob!



12:05



Let's call Alice!



Let's call Bob!



12:10



Let's call Alice!



Let's call Bob!

...

...

Livelock



### 9.4 Modellierung



## Verklemmungen: Notwendige und hinreichende Bedingungen

- Für eine Deadlocksituation müssen **vier Bedingungen** erfüllt sein
  1. **Gegenseitiger Ausschluss** bei der Benutzung der Betriebsmittel
    - Betriebsmittel werden exklusiv vergeben.
  2. **Nachforderbarkeit** von Betriebsmitteln ist möglich
    - Ein Prozess, der bereits Betriebsmittel belegt, kann weitere nachfordern.
  3. **Unentziehbarkeit** der zugeteilten Betriebsmittel
    - Die Freigabe von Betriebsmitteln erfolgt ausschließlich und explizit nur durch den Prozess, der sie belegt hält.
  4. **Zirkuläres Warten** muss eingetreten sein
    - Es existiert eine geschlossene Kette von Belegungen und Anforderungen, an der mindestens 2 Prozesse und 2 Betriebsmittel beteiligt sind.
- Bedingungen 1–3 sind **notwendige** Bedingungen
- Bedingung 4 ist **notwendig und hinreichend**

Die **Behandlung** von Deadlocks orientiert sich daran, mindestens eine der obigen vier Bedingungen zu **invalidieren**.

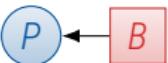
## ■ Benutzungsprotokoll für ein unteilbares Betriebsmittel

1. Anforderung (*request*)



Prozess *P* fordert (wartet auf) Betriebsmittel *B* (bekam es aber noch nicht zugeteilt, da belegt).

2. Zuteilung (*grant*)



Prozess *P* belegt Betriebsmittel *B* (wurde angefordert und zugeteilt).

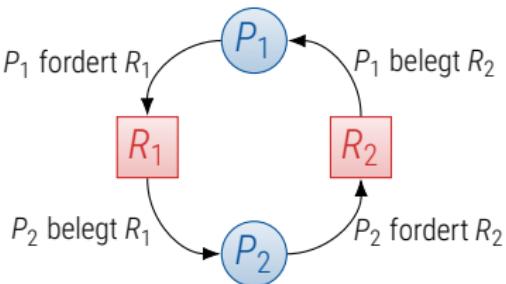
3. Benutzung (*<use>*)



Prozess *P* hat Betriebsmittel *B* weder angefordert noch belegt.

4. Freigabe (*release*)

## ■ Deadlock-Situation (geschlossene Kette $\mapsto$ zirkuläres Warten)

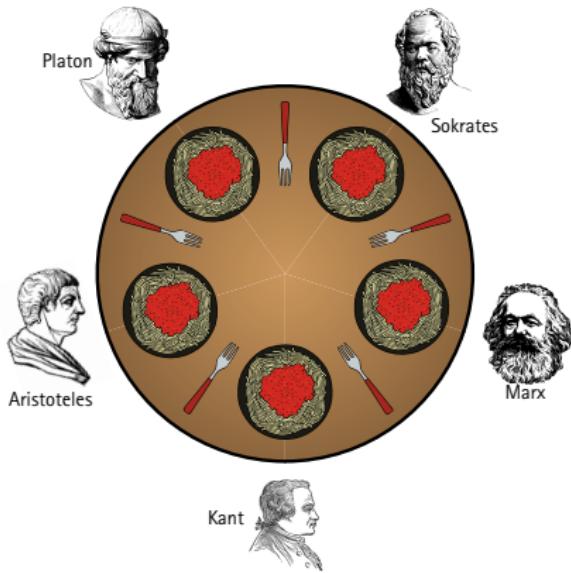
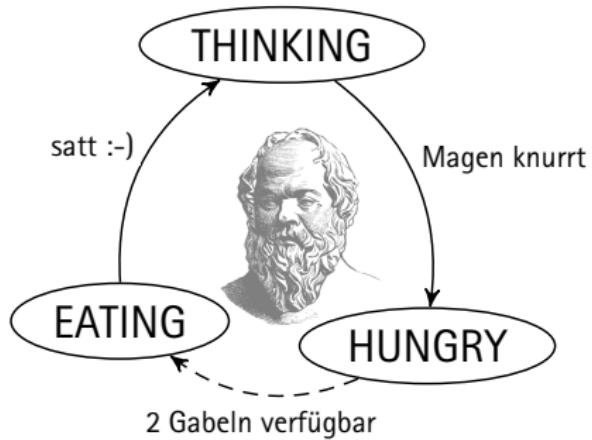




### 9.5 Philosophenproblem

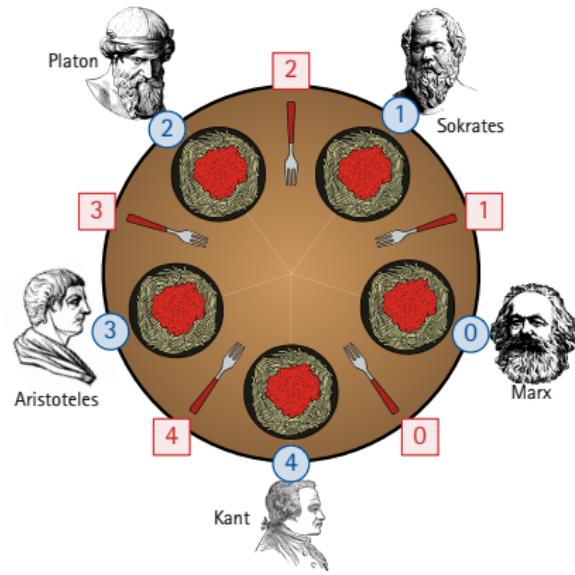
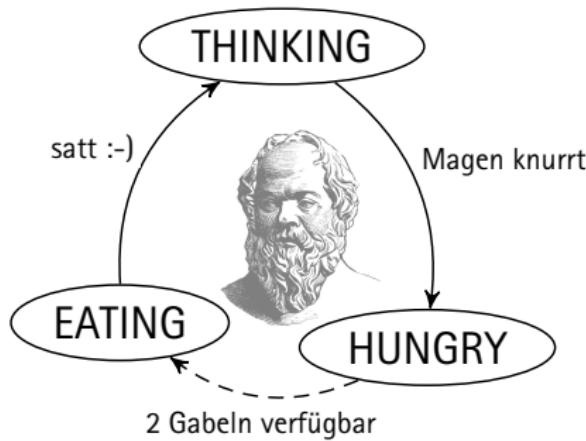
# Die speisenden Philosophen

- $N$  Philosophen sitzen um Tisch mit  $N$  Tellern Spaghetti und  $N$  Gabeln.
- Philosophen **denken**, werden jedoch ab und an **hungig** und möchten **essen**.
- Um essen zu können, benötigt ein Philosoph **zwei Gabeln** (rechts und links).



# Die speisenden Philosophen: Technische Sicht

- $N$  Prozesse (Philosophen) konkurrieren um  $N$  Betriebsmittel (Gabeln)
  - THINKING  $\mapsto$  Prozess  $n$  fordert und belegt keine Betriebsmittel
  - HUNGRY  $\mapsto$  Prozess  $n$  fordert Betriebsmittel  $n$  und  $(n + 1) \bmod N$
  - EATING  $\mapsto$  Prozess  $n$  belegt Betriebsmittel  $n$  und  $(n + 1) \bmod N$

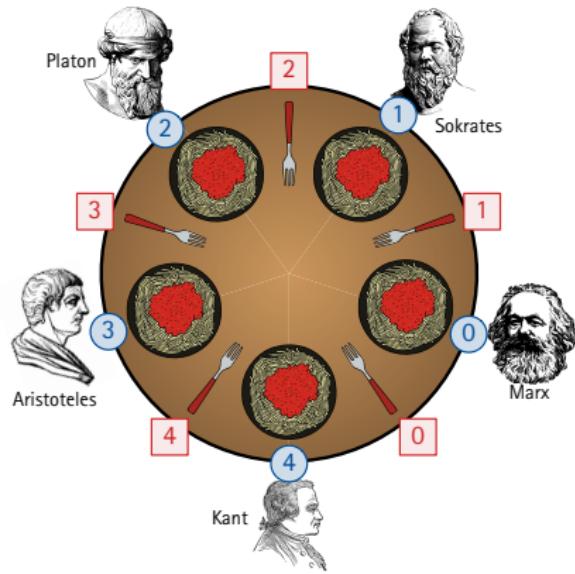


# Die speisenden Philosophen: Technische Sicht

- $N$  Prozesse (Philosophen) konkurrieren um  $N$  Betriebsmittel (Gabeln)
  - THINKING  $\mapsto$  Prozess  $n$  fordert und belegt keine Betriebsmittel
  - HUNGRY  $\mapsto$  Prozess  $n$  fordert Betriebsmittel  $n$  und  $(n + 1) \bmod N$
  - EATING  $\mapsto$  Prozess  $n$  belegt Betriebsmittel  $n$  und  $(n + 1) \bmod N$

Gesucht: Eine Lösung

- ohne Verklemmung (Deadlock)
- ohne unendliche Verzögerung (Livelock)
- unter guter Auslastung der Betriebsmittel



Lösungsskizze mit  $N$  Semaphoren  $\hookrightarrow$  8-30

```
1 #define N 5
2 semaphore_t fork[N] = {1}; // fork (left) of philosopher i, initial: 1
3
4 void philosopher(int i){ // philosopher thread function ( 0...N-1)
5     while( 1 ) { // until end of life, we...
6         think(); // work hard!           THINKING
7         // now we are hungry :-)   HUNGRY
8         down(&fork[ (i+1)%N ]); // take right fork
9         down(&fork[ i ]);       // take left fork
10
11
12         eat(); // miam, miam, Spaghetti EATING
13
14
15         up(&fork[ (i+1)%N ]); // that was good! Put right fork.
16         up(&fork[ i ]); // Put left fork.
17     }
18 }
```

# Philosophenproblem: Variante 1

Analyse

Lösungsskizze mit  $N$  Semaphoren  $\hookrightarrow$  8-30

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7
8         down(&fork[ (i+1)%N ]);
9         down(&fork[ i ]);
```

10

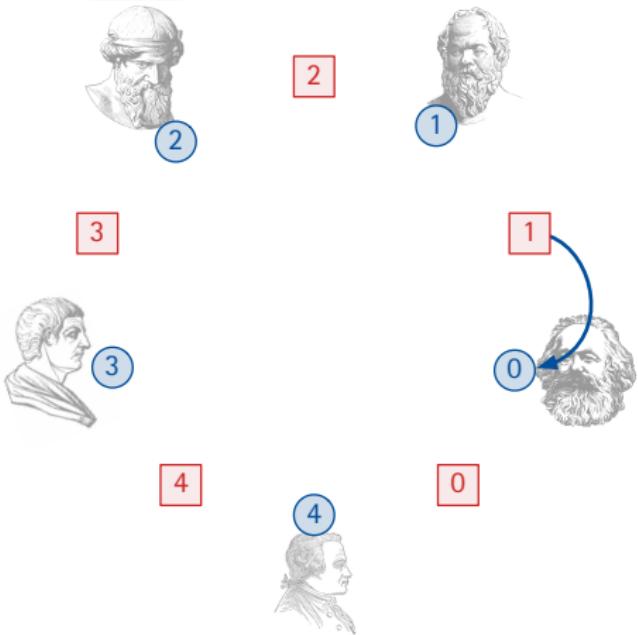
```
11     eat();
```

12

```
13
14         up(&fork[ (i+1)%N ]);
15         up(&fork[ i ]);
```

16
17 }

18 }



# Philosophenproblem: Variante 1

Analyse

Lösungsskizze mit  $N$  Semaphoren  $\hookrightarrow$  8-30

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7
8         down(&fork[ (i+1)%N ]);
9         down(&fork[ i ]);
```

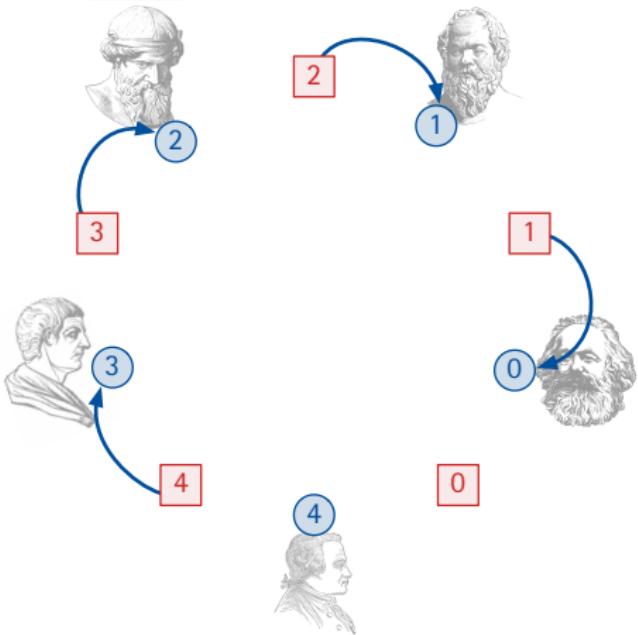
```
10
11         eat();
```

```
12
13         up(&fork[ (i+1)%N ]);
14         up(&fork[ i ]);
```

```
15     }
```



# Philosophenproblem: Variante 1

Analyse

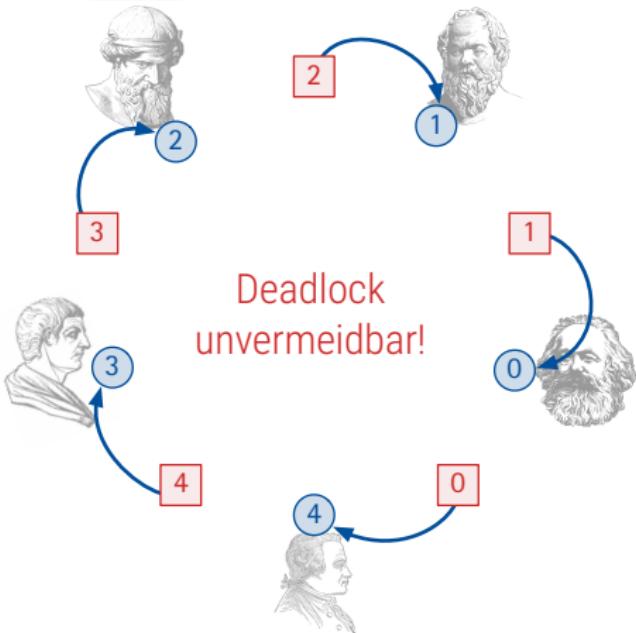
Lösungsskizze mit  $N$  Semaphoren  $\hookrightarrow$  8-30

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7
8         down(&fork[ (i+1)%N ]);
9         down(&fork[ i ]);
```

eat();

```
15     up(&fork[ (i+1)%N ]);
16     up(&fork[ i ]);
```

}



# Philosophenproblem: Variante 1

Analyse

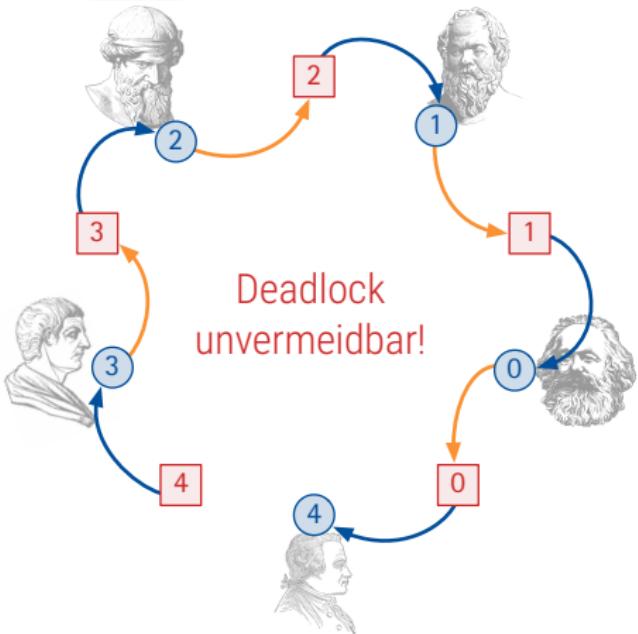
Lösungsskizze mit  $N$  Semaphoren  $\hookrightarrow$  8-30

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7
8         down(&fork[ (i+1)%N ]);
9         down(&fork[ i ]);
```

eat();

```
15     up(&fork[ (i+1)%N ]);
16     up(&fork[ i ]);
```

}

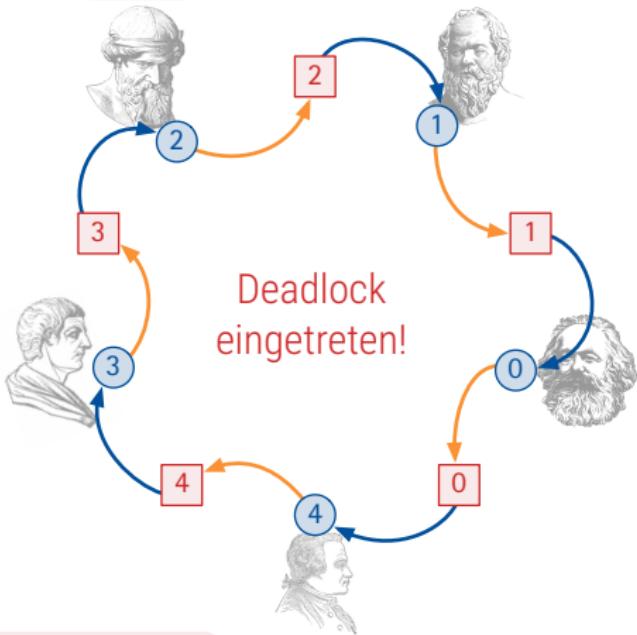


# Philosophenproblem: Variante 1

Analyse

Lösungsskizze mit  $N$  Semaphoren  $\hookrightarrow$  8-30

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7
8         down(&fork[ (i+1)%N ]); // Line 8
9         down(&fork[ i ]); // Line 9
10
11         eat();
12
13         up(&fork[ (i+1)%N ]); // Line 15
14         up(&fork[ i ]); // Line 16
15     }
16 }
17 }
```



Naive Lösung führt zu **Deadlock**, wenn alle Philosophen ihre rechte Gabel genommen haben.

**Was tun?**

# Philosophenproblem: Variante 2

mit atomarer Anforderung

Lösungsskizze mit  $N$  Semaphoren + Mutex  $\leftrightarrow$  8-30

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3 semaphore_t mutex = 1;
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7         down(&mutex);
8         down(&fork[ (i+1)%N ]);
9         down(&fork[ i ]);
```

```
10
11     eat();
```

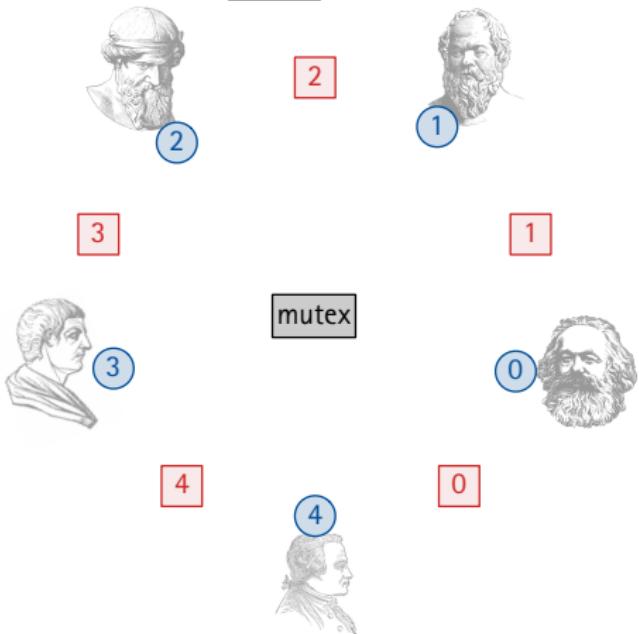
```
12
13     up(&mutex);
14     up(&fork[ (i+1)%N ]);
15     up(&fork[ i ]);
```

```
16 }
```

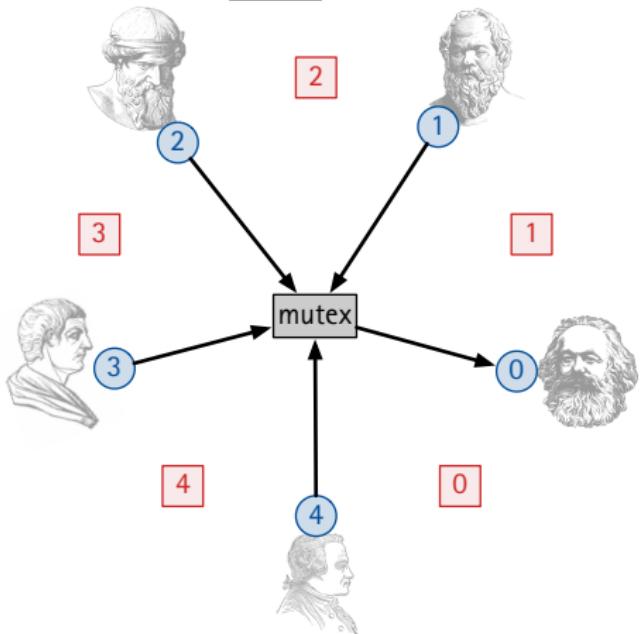
  

```
17 }
```



Lösungsskizze mit  $N$  Semaphoren + Mutex  $\leftrightarrow$  8-30

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3 semaphore_t mutex = 1;
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7         down(&mutex);
8         down(&fork[ (i+1)%N ]);
9         down(&fork[ i ]);
10
11         eat();
12
13         up(&mutex);
14         up(&fork[ (i+1)%N ]);
15         up(&fork[ i ]);
16     }
17 }
```



Lösungsskizze mit  $N$  Semaphoren + Mutex  $\leftrightarrow$  8-30

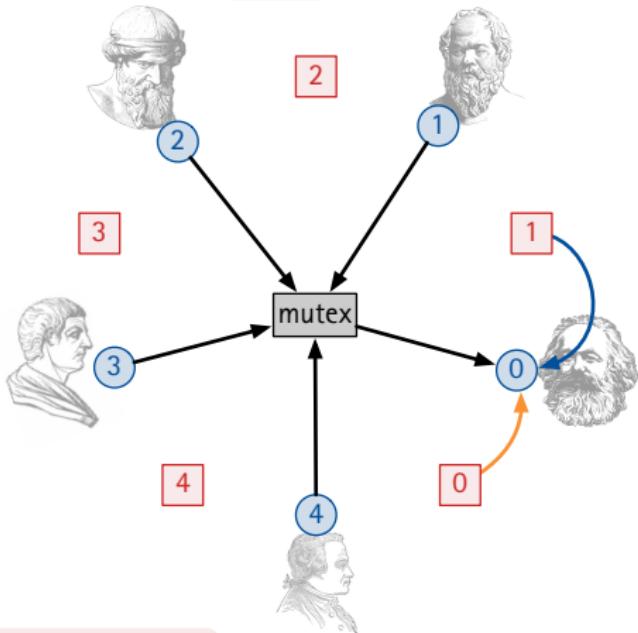
```
1 #define N 5
2 semaphore_t fork[N] = {1};
3 semaphore_t mutex = 1;
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7         down(&mutex);
8         down(&fork[ (i+1)%N ]);
9         down(&fork[ i ]);
```

```
10
11     eat();
```

```
12
13     up(&mutex);
14     up(&fork[ (i+1)%N ]);
15     up(&fork[ i ]);
```

  
16 }

Gegenseitiger Auschluss während der Betriebsmittelanforderung. Bricht Bedingung 2, aber **miese Auslastung** (nur noch ein Philosoph kann essen).

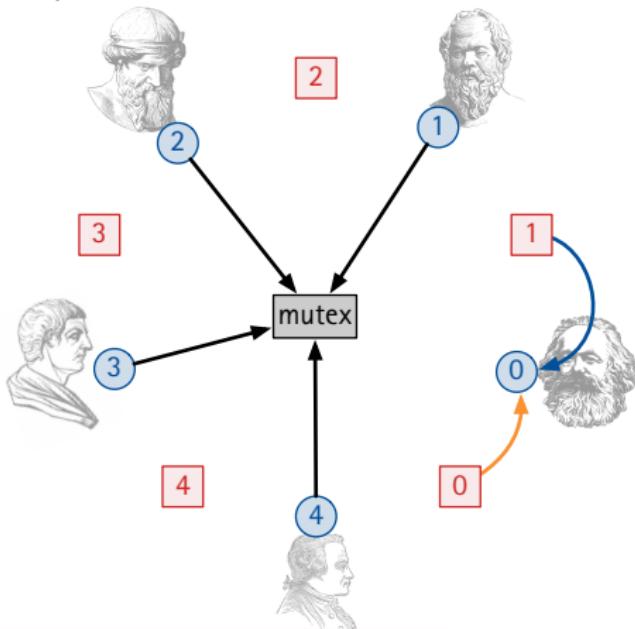


# Philosophenproblem: Variante 3

mit atomarer Anforderung

Bessere Lösungsskizze mit  $N$  Semaphoren + Mutex

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3 semaphore_t mutex = 1;
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7         down(&mutex);
8         down(&fork[ (i+1)%N ]);
9         down(&fork[ i ]);
10        up(&mutex);
11
12        eat();
13
14        up(&mutex);
15        up(&fork[ (i+1)%N ]);
16        up(&fork[ i ]);
17    }
18 }
```



Gegenseitiger Auschluss, bessere Variante mit potentiell höherer Auslastung:  
Mindestens 1 Philosoph kann essen, aber ob es tatsächlich 2 schaffen hängt davon ab, welcher Philosoph als nächstes mutex erhält.

# Philosophenproblem: Variante 4

mit Rückgabe

Lösungsskizze mit Rückgabe und Backoff

to back off:  
sich zurückhalten

```
1 #define N 5
2 semaphore_t fork[N] = {1};      // fork (left) of philosopher i, initial: 1
3
4 void philosopher(int i){
5     while( 1 ) {                // until end of life, we...
6         think();                // work hard!                      THINKING
7         while( 1 ){
8             down(&fork[ (i+1)%N ]); // now we are hungry           HUNGRY
9             if(trydown(&fork[ i ])) // take right fork
10                break;              // try to take left fork
11            else {                // success :-) --> go eating!
12                up(&fork[ (i+1)%N ]); // otherwise
13                sleep(1000);       // leave right fork
14            }
15        }                         // sleep for some time (msec)
16        eat();                     // next try
17        up(&fork[ (i+1)%N ]);    // miam, miam, Spaghetti       EATING
18        up(&fork[ i ]);          //
19    }
20}
21}
```

```
bool trydown(semaphore_t* sema){
    atomic {
        if( *sema <= 0 ) return false;
        *sema--;
        return true;
    }
}
```

Neue Operation: nichtblockierendes down()

# Philosophenproblem: Variante 4

mit Rückgabe

Lösungsskizze mit Rückgabe und Backoff

to back off:  
sich zurückhalten

```
1 #define N 5
2 semaphore_t fork[N] = {1};      // fork (left) of philosopher i, initial: 1
3
4 void philosopher(int i){
5     while( 1 ) {                // until end of life, we...
6         think();                // work hard!                      THINKING
7         while( 1 ){
8             down(&fork[ (i+1)%N ]); // now we are hungry            HUNGRY
9             if(trydown(&fork[ i ])) // take right fork
10                break;              // try to take left fork
11                // success :-) --> go eating!
12            else {               // otherwise
13                up(&fork[ (i+1)%N ]); // leave right fork
14                sleep(1000);        // backoff for some time (msec)
15            }
16            // next try
17            eat();                // miam, miam, Spaghetti          EATING
18            up(&fork[ (i+1)%N ]); // Bricht (im Prinzip) Bedingung 4: Der Prozess wartet nie bei Nachforderung und
19            up                                gibt Betriebsmittel vorzeitig wieder ab. Dadurch kein Zirkelschluss und gute
20            }                                Auslastung, aber Livelock möglich.
```

# Philosophenproblem: Variante 4

mit Rückgabe

Lösungsskizze mit Rückgabe und Backoff

to back off:  
sich zurückhalten

```
1 #define N 5
2 semaphore_t fork[N] = {1};      // fork (left) of philosopher i, initial: 1
3
4 void philosopher(int i){
5     while( 1 ) {                // until end of life, we...
6         think();                // work hard!                      THINKING
7         while( 1 ){
8             down(&fork[ (i+1)%N ]); // now we are hungry            HUNGRY
9             if(trydown(&fork[ i ])) // take right fork
10                break;              // try to take left fork
11                // success :-) --> go eating!
12            else {
13                up(&fork[ (i+1)%N ]); // otherwise
14                sleep(random(1000)); // leave right fork
15                sleep(random(1000)); // backoff for some random time (msec)
16            }
17        }                          // next try
18        eat();                     // miam, miam, Spaghetti          EATING
19        up(&fork[ (i+1)%N ]);      up Bricht (im Prinzip) Bedingung 4: Der Prozess wartet nie bei Nachforderung und
20    }                            gibt Betriebsmittel vorzeitig wieder ab. Dadurch kein Zirkelschluss und gute
21                                Auslastung, aber Livelock möglich. Abhilfe: Indeterministisches Backoff
```

# Philosophenproblem: Variante 1

Analyse

Lösungsskizze mit  $N$  Semaphoren  $\hookrightarrow$  8-30

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7
8         down(&fork[ (i+1)%N ]);
9         down(&fork[ i ]);
```

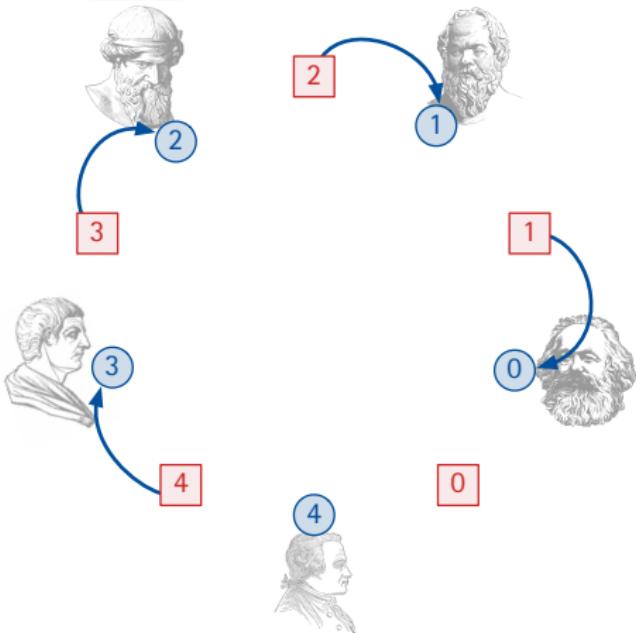
```
10
11         eat();
```

```
12
13         up(&fork[ (i+1)%N ]);
14         up(&fork[ i ]);
```

```
15     }
16 }
```



# Philosophenproblem: Variante 1

Analyse

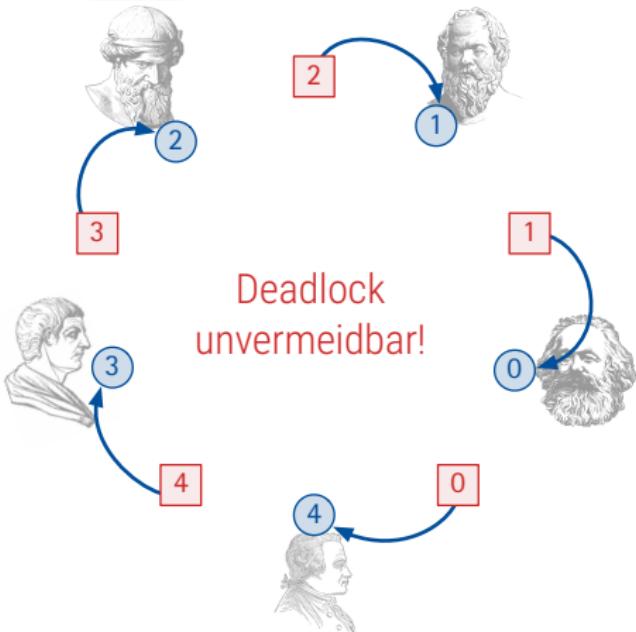
Lösungsskizze mit  $N$  Semaphoren  $\hookrightarrow$  8-30

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7
8         down(&fork[ (i+1)%N ]);
9         down(&fork[ i ]);
```

eat();

```
15     up(&fork[ (i+1)%N ]);
16     up(&fork[ i ]);
```

}



# Philosophenproblem: Variante 1.5

Einer rückwärts!

Lösungsskizze mit konstruktiver Vermeidung des Zirkelschlusses

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7         if( i < N-1 ) {
8             down(&fork[ (i+1)%N ]);
9             down(&fork[ i ]);
```

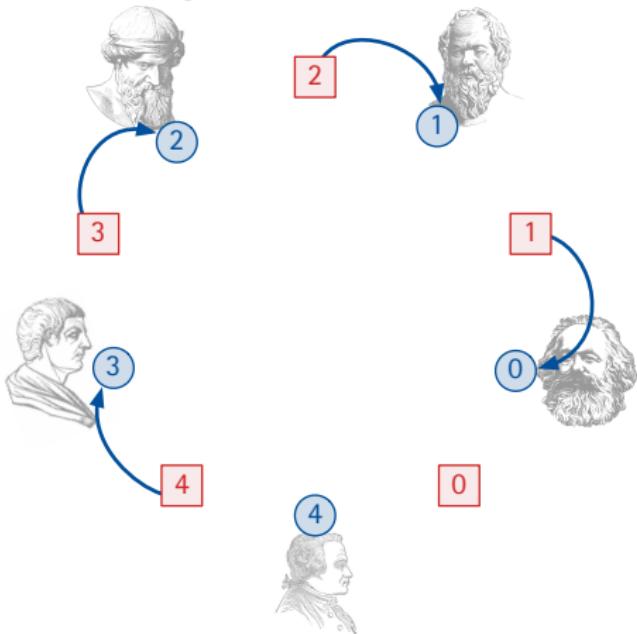
else {

```
10             down(&fork[ i ]);
```

down(&fork[ (i+1)%N ]);

```
11         }
12         eat();
13         up(&fork[ (i+1)%N ]);
14         up(&fork[ i ]);
```

```
15     }
16 }
```

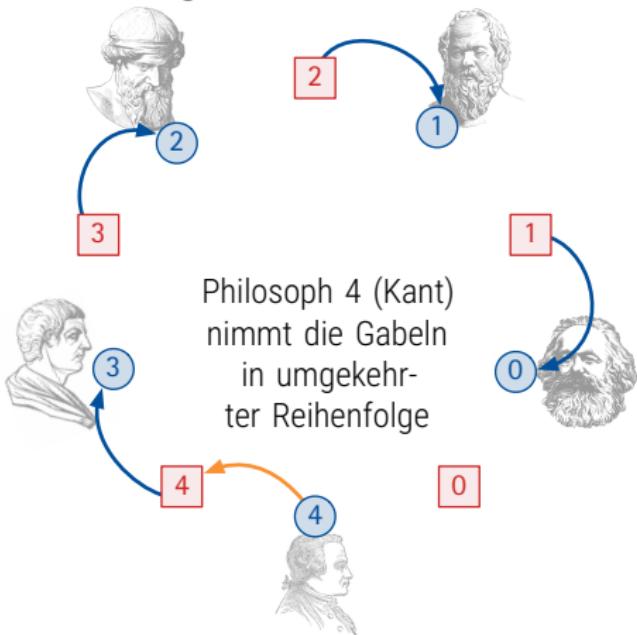


# Philosophenproblem: Variante 1.5

Einer rückwärts!

Lösungsskizze mit konstruktiver Vermeidung des Zirkelschlusses

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7         if( i < N-1 ) {
8             down(&fork[ (i+1)%N ]);
9             down(&fork[ i ]); 
10        } else {
11            down(&fork[ i ]); 
12            down(&fork[ (i+1)%N ]); 
13        }
14        eat();
15        up(&fork[ (i+1)%N ]);
16        up(&fork[ i ]); 
17    }
18 }
```

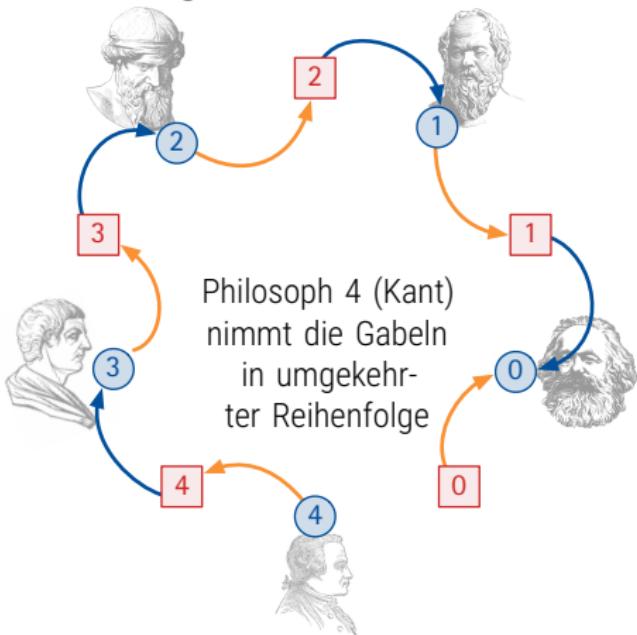


# Philosophenproblem: Variante 1.5

Einer rückwärts!

Lösungsskizze mit konstruktiver Vermeidung des Zirkelschlusses

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7         if( i < N-1 ) {
8             down(&fork[ (i+1)%N ]);
9             down(&fork[ i ]); 
10        } else {
11            down(&fork[ i ]); 
12            down(&fork[ (i+1)%N ]); 
13        }
14        eat();
15        up(&fork[ (i+1)%N ]);
16        up(&fork[ i ]); 
17    }
18 }
```

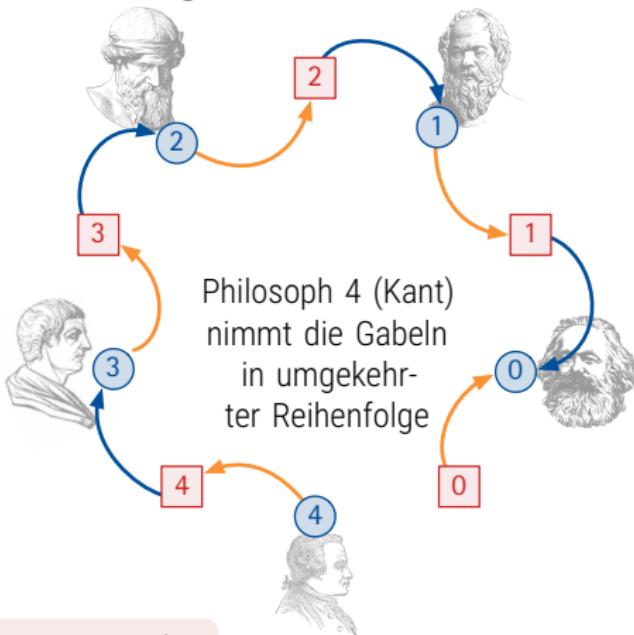


# Philosophenproblem: Variante 1.5

Einer rückwärts!

Lösungsskizze mit konstruktiver Vermeidung des Zirkelschlusses

```
1 #define N 5
2 semaphore_t fork[N] = {1};
3
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7         if( i < N-1 ) {
8             down(&fork[ (i+1)%N ]);
9             down(&fork[ i ]); 
10        } else {
11            down(&fork[ i ]); 
12            down(&fork[ (i+1)%N ]); 
13        }
14        eat();
15        up(&fork[ (i+1)%N ]);
16        up(&fork[ i ]); 
17    }
18 }
```



Bricht Bedingung 4, da nun kein zirkuläres Warten mehr eintreten kann und so, nach und nach, alle Philosophen zum Essen kommen.



Definition einer **Ordnung**, in der Betriebsmittel angefordert werden dürfen.

```
1 #define N 5
2 semaphore_t fork[N] = {1};      // ordered resources
3                                         // allocate in descending order!
4 void philosopher(int i){
5     while( 1 ) {
6         think();
7         down(&fork[max(i, (i+1)%N) ]); // aquire "higher" fork
8         down(&fork[min(i, (i+1)%N) ]); // aquire "lower" fork
9         eat();
10
11         up(&fork[ (i+1)%N ]);
12         up(&fork[ i ]);
13     }
14 }
```

Bezogen auf Ordnungsnummer,  
die hier dem Index entspricht

Verallgemeinerung von „Einer rückwärts“. Es wird eine **Ordnung** auf den Betriebsmitteln definiert. Nachforderungen sind nur für Betriebsmittel mit niedrigerer Ordnungsnummer erlaubt. Bricht Bedingung 4.

Im Beispiel fordert Philosoph 4 dadurch erst Gabel 4 (links) und dann Gabel 0 (rechts) an. Alle anderen in der bisherigen Reihung.

| Philosophen             | Idee: Deadlock...                                         | bricht Bedingung | Nachteile                                  |
|-------------------------|-----------------------------------------------------------|------------------|--------------------------------------------|
| Variante 1<br>(naiv)    | „wird schon nicht passieren“                              | <keine>          | Deadlock ist möglich                       |
| Variante 2/3<br>(Mutex) | <b>vorbeugen</b> (durch statische Zuteilung)              | Bedingung 2      | suboptimale Auslastung                     |
| Variante 4<br>(backoff) | <b>vermeiden</b> (durch dynamische Prüfung)               | Bedingung 4      | Livelock möglich; erfordert neue Primitive |
| Variante 5<br>(Ordnung) | <b>vorbeugen</b> (durch konstruktiv zyklus-freie Vergabe) | Bedingung 4      | Einschränkung der Programme                |

- Verklemmungsproblem angegangen (und teilweise gelöst)  
auf Ebene der einzelnen **Anwendung**

Was kann das Betriebssystem tun?



### 9.6 Betriebssystemstrategien



# Betriebssystemstrategien zum Umgang mit Verklemmungen

## 1. Ignorieren

- pragmatischer Ansatz: Deadlocks sind selten und liegen letztlich in der Verantwortung der Anwendung.

## 2. Vorbeugung (*deadlock prevention*)

- konstruktiver Ansatz:
  - indirekte Methoden
  - direkte Methoden
- Statisches Vergabeprotokoll stellt sicher, dass ein Deadlock nicht mehr auftreten kann.
  - ~~ heben eine notwendige Bedingung auf
  - ~~ heben die notwendige und hinreichende Bedingung auf

## 3. Vermeiden (*deadlock avoidance*)

- analytischer Ansatz: Betriebsmittelnachforderungen, die einen Deadlock verursachen würden, wird nicht stattgegeben.

## 4. Erkennen & Erholen (*deadlock detection*)

- optimistischer Ansatz: Deadlocks werden in Kauf genommen
  - Keine der vier Bedingungen wird aufgehoben
  - Stattdessen wird sporadisch nach Zirkeln im Wartegraphen gesucht und ein „Opfer“ muss Betriebsmittel abgeben (→ Prozesszerstörung oder Betriebsmittelentzug)



- Kopf in den Sand stecken
- Problem ignorieren
  - Wie oft passiert ein Deadlock?
  - Welche Konsequenzen hätte er?
  - Wie teuer wären Vorbeugung oder Vermeidung durch das Betriebssystem?

→ UNIX ignoriert Deadlocks!



- Konstruktiver Ansatz: Stelle sicher, dass Deadlocks unmöglich ist.
- Indirekte Methoden ↪ heben eine notwendige Bedingung auf
  - Nichtblockierende Synchronisation verwenden Bedingung 1
  - Alle Betriebsmittel unteilbar anfordern Bedingung 2
  - Betriebsmittel virtualisieren (und so entziehbar machen) Bedingung 3
- Direkte Methoden ↪ heben Bedingung 4 auf (zirkuläres Warten)
  - Lineare Ordnung über alle Betriebsmittel definieren, die zusichert, dass Betriebsmittel nur in aufsteigender oder absteigender Reihenfolge zuteilbar sind.

Die **konstruktive Vorbeugung** von Deadlocks erfolgt in der Regel auf **Kosten** der möglichen **Betriebsmittelauslastung** und setzt häufig **Vorabwissen** voraus. In der Praxis wird sie vor allem bei **Echtzeitsystemen** verwendet.

- *OSEK priority ceiling protocol [24]*
  - Direkte Methode, implementiert in OSEK/AUTOSAR (automotive RTOS)

- **Vorraussetzungen**

*real-time operating system*  
(Echtzeitbetriebssystem)

1. Einprozessorsystem (keine echte Parallelität der beteiligten Fäden)
2. Menge der Threads (OSEK: Tasks)  $t_1 \dots t_n \in T$  ist wohldefiniert
  - Threads haben statische Priorität  $prio : T \rightarrow \mathbb{N}$
  - Einplanung erfolgt rein prioritätenbasiert mit *run-to-completion*
3. Menge der exklusiven Betriebsmittel (OSEK: Ressourcen)  
 $r_1 \dots r_m \in \mathbb{N}$  ist wohldefiniert
4. Zur Laufzeit mögliche Ressourcenanforderungen sind wohldefiniert
  - Für jede Ressource  $r \in R$  sei  $T_r \subseteq T$  die Menge der Threads, die  $r$  anfordern

vor Laufzeit  
bekannt

keine Zeitscheiben

→ Voraussetzungen typisch für feste/harte Echtzeitsysteme

- Eine OSEK-Ressource ist vergleichbar mit einem Mutex ↪ 8-11
  - `GetResource( res )` – durch  $res$  abgesichertes kritisches Gebiet betreten
  - `ReleaseResource( res )` – durch  $res$  abgesichertes kritisches Gebiet verlassen
- Eine OSEK-Ressource hat eine feste Priorität (*ceiling priority*)
  - Priorität des **höchstprioren Threads**, der auf sie zugreifen kann
    - $prio : R \mapsto \mathbb{N}, prio(r) = \max(prio(t_1), \dots, prio(t_r))$  für  $t_1 \dots t_r \in T_r$
  - Thread hat diese Priorität während er im kritischen Gebiet ist:  $prio(t) = prio(r)$ 
    - also zwischen `GetResource( r )` und `ReleaseResource( r )`
- **Konsequenz:** `GetResource()` kann **niemals blockieren** ↪ **keine Deadlocks!**
  - Es läuft immer der höchstpriore Thread  $t \in T$  ↪ führt `GetResource( r )` aus
  - Nach `GetResource( r )` gilt:  $prio(t) = prio(r)$  ( $t$  hat *ceiling priority*)
  - Bis zum `ReleaseResource( r )` kann kein Thread  $u \in T_r$   $t$  verdrängen

↪ Wettstreit um dieselbe Ressource  $r$  ist **konstruktiv** ausgeschlossen

# Verklemmungen: 3. Vermeiden

Eingreifen, bevor die Gefahr entsteht

- Analytischer Ansatz: Erlaube keine Nachforderung, die zu Deadlock führt.
  - Prozesse und ihre Betriebsmittelanforderungen werden überwacht
    - jede Anforderung prüft auf einen möglichen unsicheren Zustand
    - sollte ein solcher möglich sein, wird die Zuteilung abgelehnt **Bedingung 4**
    - anfordernder Prozess wird suspendiert
  - Betriebsmittelzuteilung erfolgt nur im **sicheren Zustand**
    - im Falle einer Prozessfolge, die alle zukünftigen Anforderungen erfüllt
    - in Anbetracht aller aktuellen Belegungen und anstehenden Freigaben
- Vorabwissen erforderlich!

Die **dynamische Vermeidung** benötigt nicht nur **Vorabwissen** über möglichen **Betriebsmittelnachforderungen** aller Prozesse, auch die hohen Laufzeitkosten von Methode und Implementierung sind eine Herausforderung. In der Praxis haben analytische Verfahren deshalb **nur geringe Bedeutung**.

Bankiersalgorithmus

- **Bankiersalgorithmus** *banker's algorithm* [10]
  - Verfahren (Dijkstra, 1965) zur Erkennung des sicheren/unsicheren Zustands
- Voraussetzungen: Das System **kennt** die Menge der Betriebsmittel, die
  - jeder Prozess **möglichweise anfordert** (*maximum claim*, „Kreditrahmen“)
  - jeder Prozess **gegenwärtig hält** (*allocated*, „Kredit“)
  - **noch nicht** irgendeinem Prozess zugeteilt wurde (*available*, „Verfügungsrahmen“)
- Sicherer Zustand liegt vor, wenn Anforderung Folgendes nicht übersteigt:
  - ① den Kreditrahmen des anfragend Prozesses
  - ② den Verfügungsrahmen von Betriebsmitteln (der angefragten Klasse)

→ Scheitert ①, wird die Anforderung zurückgewiesen.

→ Scheitert ②, wird der anfordernde Prozess suspendiert.



# Verklemmungen: 4. Erkennen

Eingreifen, wenn es zu spät ist

- Optimistischer Ansatz: Erkenne und „repariere“ Deadlocks, wenn sie tatsächlich aufgetreten sind.
- Deadlocks werden stillschweigend in Kauf genommen
  - kein Versuch, eine der vier Bedingungen aufzuheben
  - stattdessen läuft eine **sporadische Suche** nach blockierten Prozessen
    - Betriebssystem führt Buch über Anforderungen und Belegungen
    - Der resultierende Wartegraph wird nach Zyklen abgesucht
- Erkannte Zyklen werden im Bedarfsfall durchbrochen
  - Entweder durch **Zerstörung** („killen“) eines einzelnen Prozesses.
  - Oder durch **Entzug eines Betriebsmittels** und **Zurückrollen** des besitzenden Prozesses (erfordert *Checkpointing* / Transaktionen).

„Schaden macht klug, aber zu spät“ (Sprichwort)

Die **optimistische Erkennung und Behebung** von Verklemmungen ist eine Gratwanderung zwischen Schaden und Aufwand. Zum mindestens die **Erkennung** findet in der Praxis durchaus Verwendung, oft als optionaler Systemdienst.



### 9.7 Zusammenfassung



Zusammenfassung: Betriebsmittelverwaltung,  
Synchronisation und Verklemmung

■ **Ausgangsbasis:** Betriebsmittelzuteilung durch das Betriebssystem

- Buchführung und Steuerung der Zuteilung
- statisch (vorab) oder dynamisch („on demand“)
- Prozesse im Wettstreit um Betriebsmittel

■ **Problem:** **Verklemmung** von Prozessen: **Deadlock** und **Lifelock**

- Gegeben, wenn **vier Bedingungen** gleichzeitig gelten
- 1. exklusive Belegung, 2. Nachforderung, 3. kein Entzug
- 4. zirkuläres Warten ist eingetreten

■ **Maßnahmen:** Aufheben einer der vier Deadlockbedingungen

- **Ignorieren, Vorbeugen** (konstruktiv),  
Vermeiden (analytisch), Erkennen & Erholen (optimistisch)
- Vermeiden und Erholen sind eher praxisirrelevant

→ **In der Praxis** ist das Deadlockproblem **auf Anwendungsebene** zu lösen

- Vorbeugen (nur) im Echtzeitbetrieb üblich ↗ geringere Auslastung
- Erkennen & Erholen beschränkt sich auf die Erkennung (*Debugging-Hilfe*)



# Ein letztes Beispiel :-)



If you can explain me deadlocks,  
you will have the job!



If you give me the job,  
I can explain you deadlocks.



...



Welcome to our team!



# Überblick: Teil C Interaktion und Kommunikation

8 Speicherbasierte Interaktionen

9 Betriebsmittelverwaltung, Synchronisation und Verklemmung

## **10 Interprozesskommunikation (IPC)**

10.1 Einordnung

10.2 Grundlagen

10.3 Nachrichtensemantiken

10.4 Client–Server: Architekturvarianten

10.5 Fernaufrufe (RPCs)

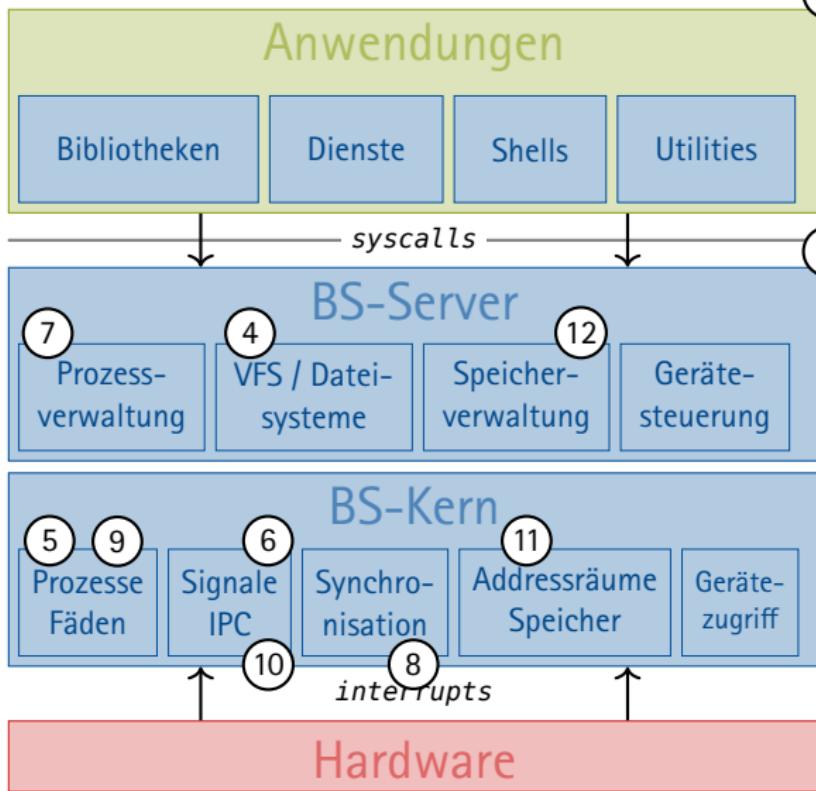
10.6 IPC unter UNIX: Signale, Pipes und Sockets

10.7 Zusammenfassung

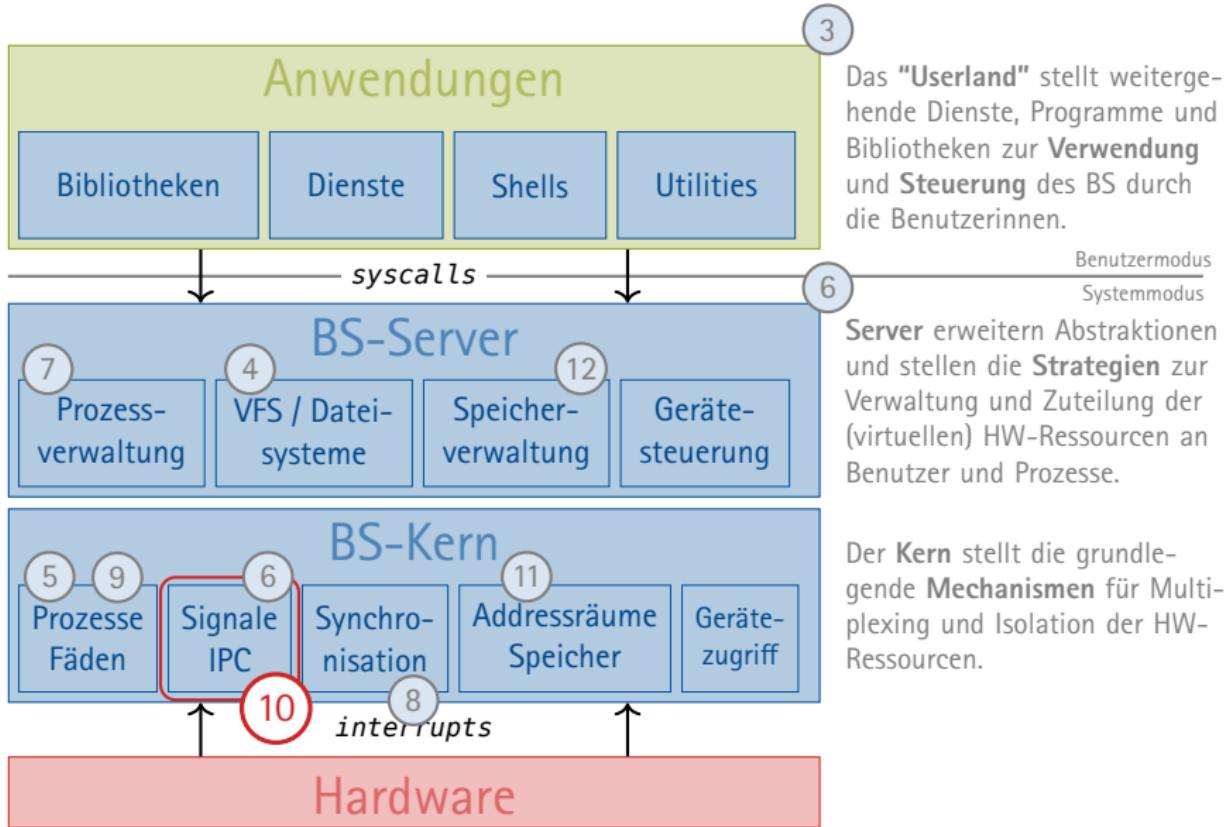
## 10.1 Einordnung



# 10 Interprozesskommunikation (IPC) – Einordnung



# 10 Interprozesskommunikation (IPC) – Einordnung





# 10 Interprozesskommunikation (IPC) – Einordnung

Worum geht es in diesem Kapitel?

■ **Verstehen** grundlegender Prinzipien der Interprozesskommunikation (*IPC*)

- **speichergekoppelt** – implizit über geteilten Adressraum → ??
- **nachrichtengekoppelt** – explizit über `send()` und `receive()`  
~~ Schwerpunkt der heutigen Vorlesung

■ **Erläutern** der Dimensionen nachrichtengekoppelter IPC

- Interaktion und Synchronität
- `send/receive`-Semantiken
  - no-wait send, synchronization send,
  - remote-invocation send, asynchronous remote-invocation send
- Verbindungsorientierte und verbindungslose Kommunikation
- Fernaufrufe (*remote procedure calls, RPCs*)

■ **Anwenden** einfacher IPC-Mechanismen unter POSIX

- IPC über Pipes und Sockets

→ Basismechanismen der nachrichtenorientierten Interaktion



# Prozesse und Synchronisation – Wiederholung

## ■ Prozesse können miteinander interagieren

### ■ Aufeinander warten $\mapsto$ **Synchronisation**

- Gemeinsame Benutzung eines wiederverwendbaren Betriebsmittels  $\mapsto$  5-11
- Logische Abhangigkeit von einem externen Ereignis (z. B. Kindprozess terminiert)

### ■ Daten austauschen $\mapsto$ **Kommunikation**

- Gezielter Informationsfluss zwischen Prozessen
- Nachrichten als konsumierbare Betriebsmittel

## ■ Wartemechanismen des Betriebssystems $\mapsto$ kontrollierten Interaktion

### ■ Warten auf asynchrone Signale (wait<sup>(2)</sup>, sigwaitinfo<sup>(2)</sup>)

$\mapsto$  6-28

### ■ Warten auf Semaphore (down(), POSIX: sem\_wait<sup>(3)</sup>)

$\mapsto$  8-30

### ■ Zyklisches Warten fuhrt zu Verklemmungen

$\mapsto$  9-15

## ~~> **Datenaustausch** zwischen Prozessen bislang nur am Rande betrachtet

### ■ Unter der Annahme von gemeinsamen Speicher

### ■ Insbesondere leichtgewichtige Prozesse (Faden) im selben Adressraum

## 10.2 Grundlagen

## Interprozesskommunikation (IPC)

Verständigung unter Prozessen  $\hookrightarrow$  5 eines Rechensystems mithilfe von Daten.

- Ein Prozesssystem bearbeitet gemeinsam **eine** Aufgabe
  - Parallelisierung, Latenzverbergung durch Ausführung „im Hintergrund“
  - Zugriff auf gemeinsame Daten und Informationen
- $\hookrightarrow$  Gekoppelte Prozesse  $\rightsquigarrow$  müssen miteinander kommunizieren können
- Kommunikation durch **gemeinsamen Speicher** (*shared memory*)
  - Nebenläufiges Schreiben/Lesen von gemeinsamem Speicher (write/read)
  - Erfordert explizite Synchronisation, z.B. mit Semaphore
- $\hookrightarrow$  8
- Kommunikation durch **Nachrichtenaustausch** (*message passing*)
  - Prozesse schicken und empfangen Nachrichten (send/receive)
  - Implizite Synchronisation über den Nachrichtenaustauschmechanismus
  - Ortstransparenz möglich – Transparente Kommunikation über Rechnergrenzen

- IPC über Nachrichten basiert auf zwei **Grundfunktionen** (Primitiven)
  - **send**( $E, n$ ):  $S \xrightarrow{n} E$  Sender  $S$  schickt Nachricht  $n$  an Empfänger  $E$
  - **receive**( $S, n$ ):  $E \xleftarrow{n} S$  Empfänger  $E$  empfängt Nachricht  $n$  von Sender  $S$
- Bereitgestellt durch das **Kommunikationssystem** (Betriebssystem)
- Unterschiede gibt es in der Semantik bezüglich
  - Synchronisation der Beteiligten
  - Adressierung von Sender  $S$  und Empfänger  $E, k : m$  Kommunikation
  - Pufferung und Nachrichtenformat (maximale Größe von  $n$ )

„Postdienstleister“

Kommunikationssystem  $\mapsto$  „beteiliger Dritter“

Technisch gesehen erfolgt die Kommunikation zwischen  $S$  und  $E$  **nicht direkt**, sondern immer über das **Kommunikationssystem**  $KS$  als vermittelndem Dritten:

- **send**( $E, n$ ):  $S \xrightarrow{E,n} KS$  Sender  $S$  übergibt Nachricht  $n$  für  $E$  an  $KS$
- **receive**( $S, n$ ):  $E \xleftarrow{S,n} KS$  Empfänger  $E$  erhält Nachricht  $n$  von  $S$  vom  $KS$

Erst diese Indirektion ermöglicht Eigenschaften wie Ortstransparenz, Asynchronität, ...  
Funktional transparent – aber nichtfunktional unter Inkaufnahme zusätzlicher Kosten!

## 10.3 Nachrichtensemantiken

## Ausprägungen

### ■ Dimension 1: Kommunikationsmuster (Interaktionen)

- **Meldung**  $\mapsto$  Einwegrnachricht – Sender benötigt kein Ergebnis

①  $S \xrightarrow{n} E$ : Nachricht

- **Auftrag**  $\mapsto$  Zweiwegrnachricht – Sender ist auf Ergebnis angewiesen

①  $S \xrightarrow{a} E$ : Auftrag (in Nachricht a)

②  $E$ : Ausführung des Auftrags

③  $E \xrightarrow{e} S$ : Ergebnis (in Antwortnachricht e)

### ■ Dimension 2: Zeitliche Kopplung (Synchronität)

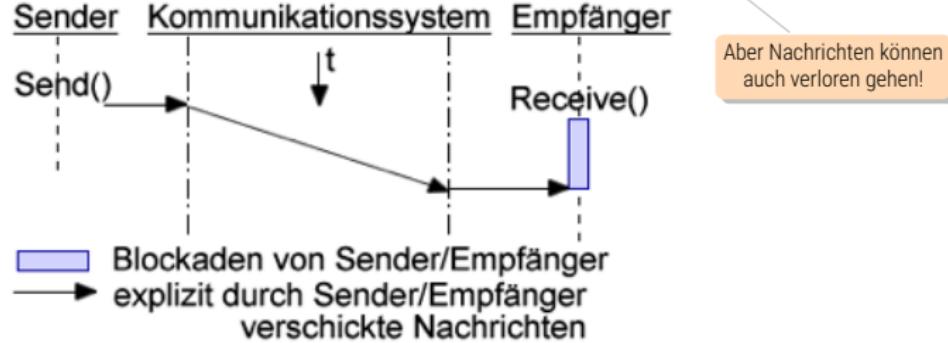
- **Asynchron**  $\mapsto$  Verarbeitung in S und E ist zeitlich entkoppelt

- **Synchron**  $\mapsto$  Verarbeitung in S und E ist zeitlich gekoppelt ([Rendezvous](#))

→ Vier verschiedene Nachrichtensemantiken

[20]

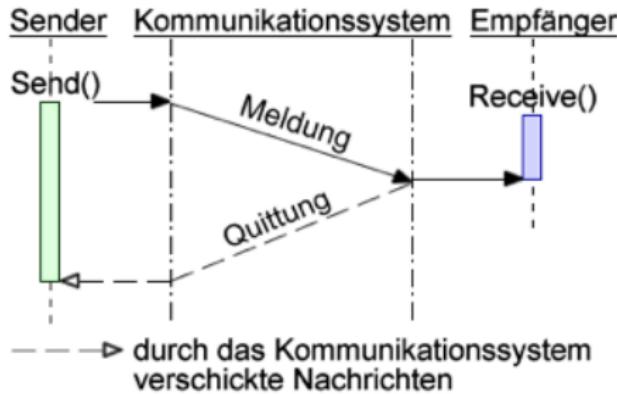
- **No-wait send:** asynchrone Meldung („fire-and-forget“)
  - Sender S liefert Nachricht an Kommunikationssystem und arbeitet dann weiter
  - Sender S „blockiert“, bis Nachricht das Kommunikationssystem erreicht hat
- Kommunikationssystem muss Nachrichten speichern können.
- Empfänger E ist unter Umständen im receive()-Aufruf blockiert. Falls nicht
  - KS verwirft die Nachricht
  - KS stellt die Nachricht an eine Warteschlange zu
- Sender und Empfänger sind zeitlich vollkommen entkoppelt



- **Synchronization send:** Synchrone Meldung mit Quittung („Rendevouz“)
    - Sender  $S$  liefert Nachricht an Kommunikationssystem
    - Sender  $S$  blockiert bis Nachricht den Empfänger  $E$  erreicht hat
  - Quittung wird vom **Kommunikationssystem** zugestellt
    - Bei Einreihen der Nachricht in die Warteschlange von  $E$
    - Sobald  $E$  die Nachricht mit `receive()` entgegen genommen hat
- Verhindert das zu schnelle Senden von Nachrichten

„Einwurfeinschreiben“

„Übergabeeinschreiben“

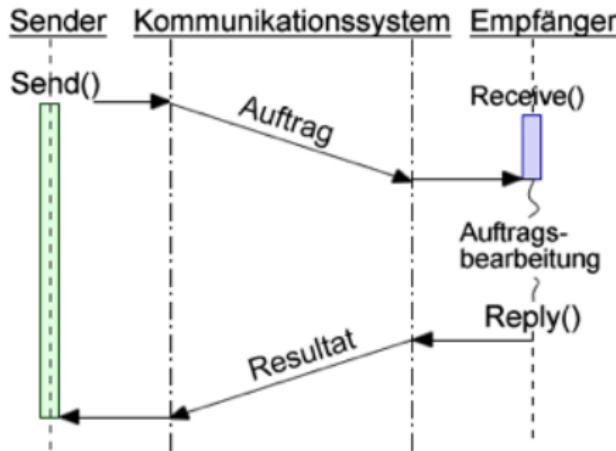


## ■ **Remote-invocation send:** Synchroner Auftrag mit Resultat

- Bearbeitung beim Empfänger ist Teil der Transaktion
- Sender S blockiert bis Resultat eintrifft
- **reply( S, n ):**  $E \xrightarrow{n} S$  zusätzliche Funktion zum Versenden der Antwort

## ■ Starke Beschränkung der Parallelität zwischen S und E

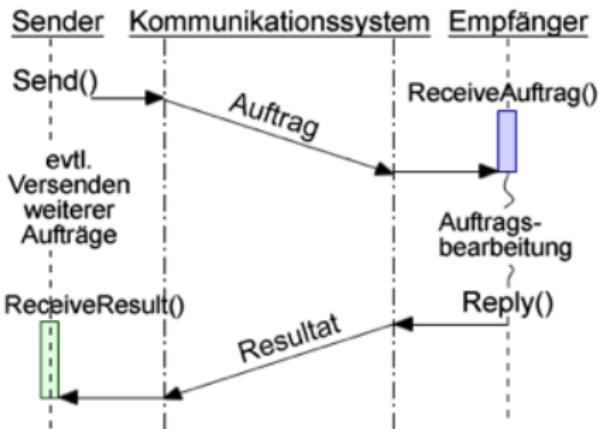
- Prinzip des **Fernauftrufs** (*remote procedure call*, RPC)



- **Asynchronous remote-invocation send:** Auftrag und Resultat in zwei unabhängigen Nachrichten; Verknüpfung über Auftragskennung.

- Sender S „blockiert“ bis Nachricht das Kommunikationssystem erreicht hat
- Sender S blockiert ggfs. später, wenn er ein Ergebnis braucht
- Sender S kann mehrere offene Aufträge haben
- Empfänger E kann Aufträge gezielt auswählen (Planung)

→ Asynchroner Fernaufruf mit zeitlicher Entkopplung:



Asynchronous remote-invocation send stellt zusätzliche Anforderungen an das Kommunikationssystem, insbesondere Session-Management (Auftragskennungen), Fehlerbehandlung, Zeitschränkenüberwachung und zusätzliche Pufferung sind erforderlich.

Es ist heute das vorwiegende Interaktionsmuster von Web-Anwendungen.

Wie finden Sender (*Client*) und Empfänger (*Server*) zusammen?

■ **Direkte Adressierung:** Empfänger ist ein Server-Prozess

- Process-ID (Signale)
- Kommunikationsendpunkt eines Prozesses (*Port, Socket*)

■ **Indirekte Adressierung:** Empfang und Server-Prozess(e) entkoppelt

- Kanäle (*Pipes*)
- Briefkästen (*Mailboxes*), Nachrichtenpuffer (*Message Queues*)
- Unter Linux durch (benannte) **Dateiobjekte** adressiert

■ **Zusätzliche Dimension:** Gruppenadressierung (1 : n)

- **Einzelruf** (*Unicast*) – an genau einen (Standard)
- **Gruppenruf** (*Multicast*) – an eine Auswahl
- **Rundruf** (*Broadcast*) – an „alle“

Falls einer Sequenz von Nachrichten/Daten übertragen werden soll:

- **Stromorientiert vs. Nachrichtenorientiert:** Granularität der Übertragung
  - stromorientiert (*stream-based*): Zeichenorientiert (Bytes)
  - nachrichtenorientiert (*message-based*): In festen Einheiten (Datagramme)
- **Verbindungsorientiert vs. Verbindungslos:** Sitzungsmanagement
  - verbindungsorientiert: über „stehende Verbindung“, Abbruch wird bemerkt
  - verbindungslos: jede Nachricht wird einzeln und unabhängig zugestellt
- **Sicher vs. Unsicher:** Transfergarantie, Ordnung
  - Sicher: Empfänger erhält Nachrichten in der Reihenfolge ihres Versandes
  - Unsicher: Verlust, Verdopplung und Vertauschung von Datagrammen möglich

## 10.4 Client–Server: Architekturvarianten

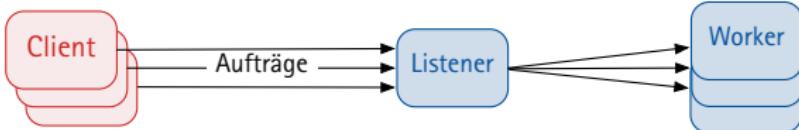
- *n* Clients, ein Server, **remote-invocation send** mit direkter Adressierung



Server wartet auf Nachricht und arbeitet Aufträge nacheinander ab

- Änderungen am Serverzustand implizit sequentialisiert :-)
- Keine Parallelität auf Serverseite, Client potentiell lange blockiert :-()

- Alternative: Server mit **Listener/Worker**-Struktur

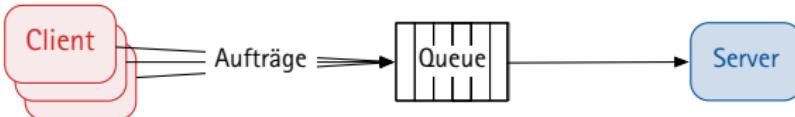


**Listener** wartet nur auf neue Nachrichten; erzeugt **Worker**

(eigener Faden oder Prozessklon mit fork<sup>(2)</sup>) zur Erledigung des Auftrags

- Parallelität auf Serverseite, Client weniger lange blockiert :-)
- Geheimer Serverzustand muss wieder explizit geschützt werden :-()
- Häufige Faden/Prozesserzeugung teuer :-()

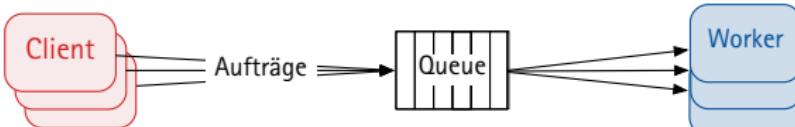
- Alternative: ein Server, **synchronization send** mit indirekter Adressierung



Server wartet auf Nachricht und arbeitet Aufträge nacheinander ab

- Änderungen am Serverzustand implizit sequentialisiert :-)
- Keine Parallelität auf Serverseite, aber Client blockiert nicht :-|

- Alternative: mehrere **Worker** (Entkopplung durch indirekte Adressierung)



Mehrere Server-Prozesse/Fäden arbeiten eingehende Aufträge parallel ab.

- Parallelität auf Serverseite, Client blockiert nicht :-)
- Fester „Pool“ an „Arbeitern“ effizienter als Erzeugung pro Auftrag :-)

- IPC über Nachrichten: **Kommunikation auf höherer Abstraktionsebene**

- Kein gemeinsamer Speicher → **Ortstransparenz**, strikte Isolation
  - **Explizite Kommunikation** → Trennung der Belange, Verständlichkeit
  - Lose Kopplung der Partner → Modularisierung, Asynchronität

- IPC über Nachrichten: **Kommunikation mit höherem Aufwand**

- Sprachlich oft unzureichend abgebildet → **Programmieraufwand**
  - Nachrichten müssen kopiert werden → Speicher und Laufzeit**kosten**
  - Nachrichtengröße oft beschränkt → Aufteilung großer Nutzlasten

- Im **lokalen Fall** kann das Betriebssystem viele dieser Kosten optimieren

- Nutzlast in CPU-Registern (*short IPC*) oder durch Übertragung einer ganzen Speicherseite (*zero copy*) über die Adressraumgrenze transferieren
  - Vor allem in Mikrokern-Betriebssystemen erforderlich [18, 19]

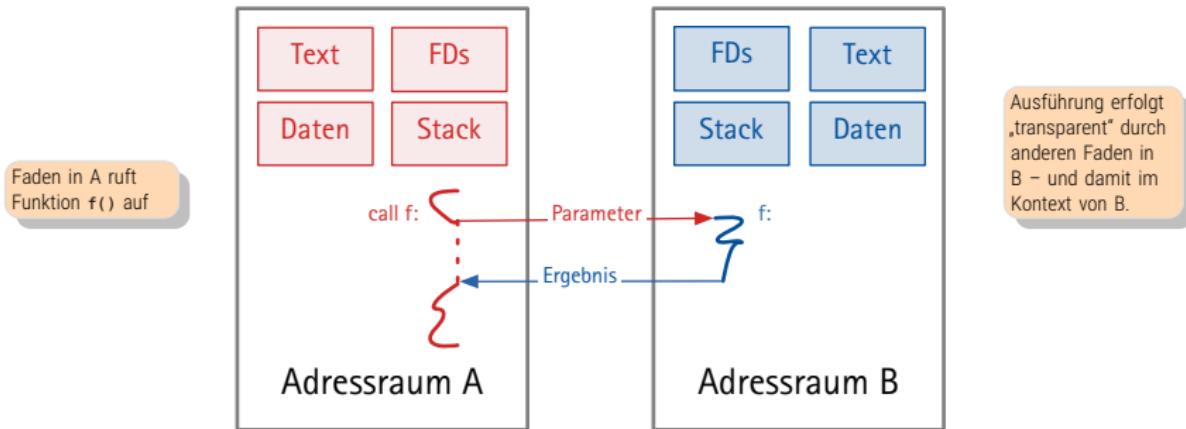
→ 3-24

→ **Ortstransparenz** und **Asynchronität** bleiben letztlich eine Illusion

- Begrenzt durch Pufferkapazität und Fehlerkorrektur des Kommunikationssystems
  - (Auch) asynchrone Aufrufe können blockieren oder fehlschlagen!

## 10.5 Fernaufrufe (RPCs)

- Grundidee: Adressraumübergreifender Prozederaufruf [4]
  - Programmierer nutzt die gewohnte Semantik des Unterprogrammaufrufs
  - Ausführung erfolgt jedoch in einem anderen Adressraum (Prozess)
  - „Transparent“ auch über das Netzwerk auf einem anderen Rechner (*Host*)

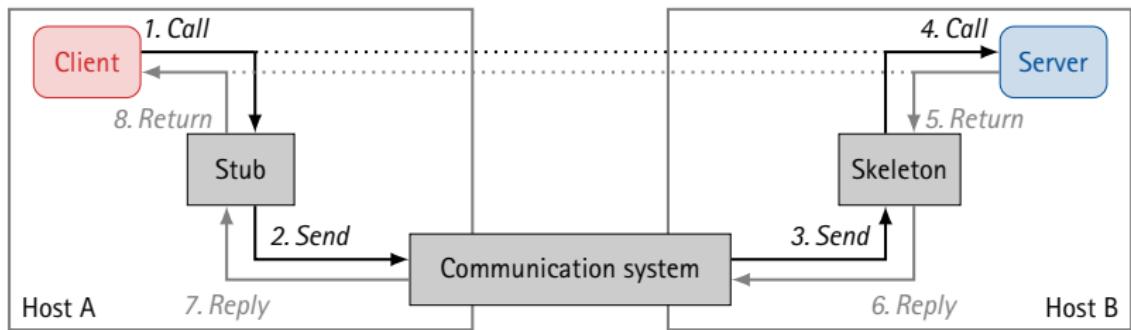


- Bereitstellung derselben Semantik wie bei lokalem Aufruf?
  - Bereitstellung der funktionale Semantik im Normalfall
  - Nichtfunktionale Eigenschaften unterscheiden sich jedoch gewaltig



# Realisierung: Stub und Skeleton

- Sprung in ein *lokales* Unterprogramm erfolgt durch Umsetzen des PCs
- Bei RPC wird nun in einen lokalen **Stellvertreter (stub)** gesprungen:
  - **Stub (Client Stub)** – Stellvertreter des Servers auf Client-Seite
    - Bezeichner und Signatur identisch zu aufzurufender Funktion auf Server-Seite
    - Verpacken der Aufrufparameter und Entpacken des Rückgabewerts
    - Umsetzung des Aufrufs in einen Nachrichtenaustausch
  - **Skeleton (Server Stub)** – Stellvertreter des Clients auf Server-Seite
    - Ruft die tatsächliche Funktionsimplementierung auf Server-Seite auf
    - Entpacken der Aufrufparameter und Verpacken des Rückgabewerts
    - Umsetzung der Aufrufrückkehr in einen Nachrichtenaustausch



- Der Code für die Stub und Skeleton wird üblicherweise generiert

```
// Client
int main() {
    ...
    int result = Print( "Hello", 7 );
    ...
    exit( result );
}
```

```
// Server
int main () { ... }

int Print( const char* s, int n ) {
    for( int i = 0; i < n; ++i )
        printf( "%d: %s\n", i, s );
    return 0;
}
```

1. Call

```
// Stub (client stub)
int Print( const char* s, int n ) {
    Msg* m = AllocMsg();
    PackStr( m, s ); // Par. verpacken
    PackInt( m, n );
    send( Server, m );
    receive( m );
    int res = UnpackInt( m );
    FreeMsg( m );
    return res;
}
```

4. Call

```
// Skeleton (server stub)
void Print_Skeleton() {
    Msg m = AllocMsg();
    while( 1 ) {
        receive( m ); // ↓ Par. entpacken
        const char* str = UnpackStr( m );
        int count      = UnpackInt( m );
        int res = Print( str, count );
        PackInt( m, res );
        reply( m ); // Ergebnis an Client
    }
}
```

2./3. Send

- Der Code für die Stub und Skeleton wird üblicherweise generiert

```
// Client
int main() {
    ...
    int result = Print( "Hello", 7 );
    ...
    exit( result );
}
```

```
// Server
int main () { ... }

int Print( const char* s, int n ) {
    for( int i = 0; i < n; ++i )
        printf( "%d: %s\n", i, s );
    return 0;
}
```

1. Call

```
// Stub (client stub)
int Print( const char* s, int n ) {
    Msg* m = AllocMsg();
    PackStr( m, s ); // Par. verpacken
    PackInt( m, n );
    send( Server, m );
    receive( m );
    int res = UnpackInt( m );
    FreeMsg( m );
    return res;
}
```

4. Call

```
// Skeleton (server stub)
void Print_Skeleton() {
    Msg m = AllocMsg();
    while( 1 ) {
        receive( m ); // ↓ Par. entpacken
        const char* str = UnpackStr( m );
        int count      = UnpackInt( m );
        int res = Print( str, count );
        PackInt( m, res );
        reply( m ); // Ergebnis an Client
    }
}
```

2./3. Send

- RPC „versteckt“ explizite IPC hinter lokalen Funktionsaufrufen
  - Unterprogramm wird **substituiert** durch **Stub** (identischer Bezeichner/Signatur)
  - Stub abstrahiert vom IPC-Mechanismus (Nachrichten)
    - Client muss nach Verlagerung einer Funktion „nur“ neu übersetzt werden.
- Auf Serverseite geringfügig mehr Aufwand (weniger Transparenz).
  - Server muss explizit Threads für Skeletons starten
- **Letztlich bleibt die Idee der Ortstransparenz dennoch eine Illusion!**
  - Sprache: Eingeschränkte Signaturen, *Call-by-reference* erfordert „Klimmzüge“
  - Schutz: Client muss sich beim Server authentifizieren (und umgekehrt)
  - Fehler: RPC kann Fehler provozieren, die lokal nicht möglich sind
  - Aufwand: Hoch, vor allem, wenn Client und Server auf demselben Host liegen

RPC hat hohe Bedeutung im klassischen Client–Server-Betrieb (Middlewares, DCOM, CORBA). Die lokale Windows-Netzwerkarchitektur (mit Domänencontroller) basiert auf MSRPC (DCOM), das Network File System (NFS) unter UNIX auf SunRPC.

Mit dem Aufkommen von (webbasierten) Diensten über das Internet gewannen Paradigmen für loser gekoppelte IPC an Popularität.

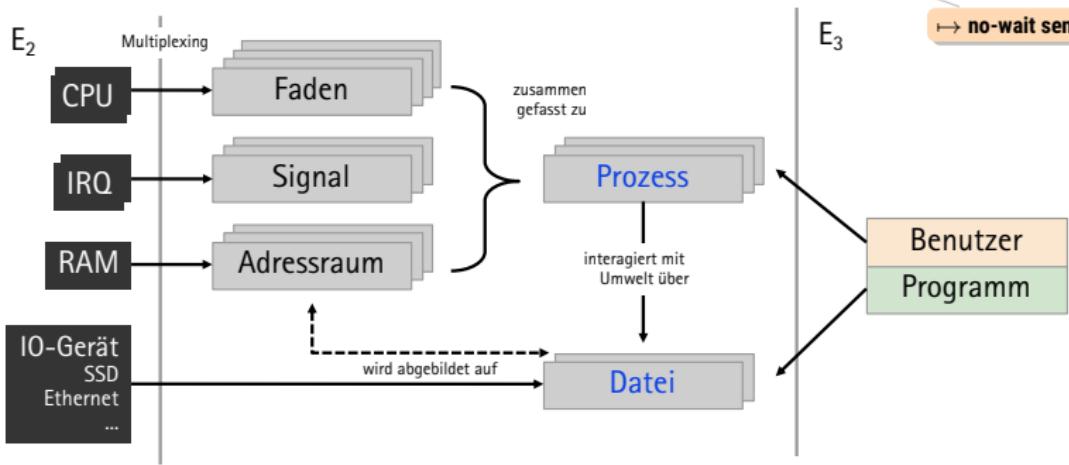
## 10.6 IPC unter UNIX: Signale, Pipes und Sockets

- POSIX Systeme stellen eine Vielzahl von IPC-Mechanismen bereit
  - Lokale Interprozesskommunikation
    - Signale
    - Pipes (FIFO-Kanäle)
    - Named Pipes (FIFO-Dateien)
    - Message Queues (Nachrichtenwarteschlangen)
  - Netzwerkübergreifende Interprozesskommunikation (auch lokal möglich)
    - Sockets
    - RPC (NFS/Sun RPC, D-BUS)
- Im Folgenden schauen wir uns einige davon etwas detaillierter an!

- Virtuelle Hardwareressourcen werden repräsentiert durch die grundlegenden Konzepte (**Abstraktionen**) eines Betriebssystems. **In UNIX sind das:**
  - **Prozess** (Adressraum + Fäden) ↪ virtueller „Computer“ (Prozessor)
  - **Signal**

↪ virtuelle „Ausnahme“ dieses Prozessors  
(synchron wie **asynchron**)

↪ no-wait send



CPU-Ausnahmen

**■ synchrones Signal**, ausgelöst durch die Aktivität des **eigenen Prozesses**

- „durchgereichter“  $E_2$ -Trap bei Ausführung einer Instruktion durch die **CPU**
  - *div/0 (SIGFPE), segment violation (SIGSEGV), illegal instruction (SIGILL)*, ...
- zusätzlicher  $E_3$ -Trap bei Ausführung eines syscalls durch das **Betriebssystem**
  - *pipe error (SIGPIPE), illegal syscall (SIGSYS)*, ...
- wie Traps nicht unterdrückbar/verzögerbar ↪ **Behandlung zwingend**

**■ asynchrones Signal**, ausgelöst durch **externes Ereignis / anderen Prozess (or)**

- „durchgereichtes“ asynchrones Ereignis des L2-Prozessors (**CPU, HW-Ereignis**)
  - *Ctrl-C pressed (SIGINT), timer expired (SIGALRM)*, ...
- zusätzliche asynchrone Ereignisse eines  $E_3$ -Prozessors (**anderer Prozess, BS**)
  - *child stopped/terminated (SIGCHLD), terminate (SIGTERM), hangup (SIGHUP), user signal 1/2 (SIGUSR1, SIGUSR2)*, ...
- wie IRQs unterdrückbar/verzögerbar ↪ **Synchronisation erforderlich**
  - analog zum NMI gibt es auch nicht unterdrückbare Signale: **SIGKILL** und **SIGSTOP**

# Asynchrone Signale als IPC-Mechanismus

Logische Sicht

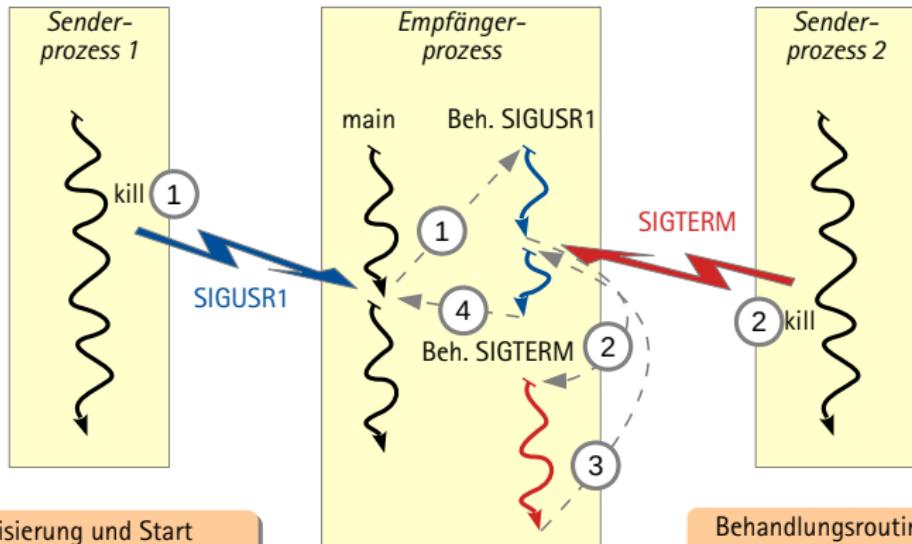
- Asynchrones Signal  $\mapsto$

No-wait send einer direkt adressierten leeren Nachricht

- Implizit durch das BS bei Kindprozessen (`SIGCHLD`, `wait(2)`)  $\mapsto$  `recv()`
- Explizit durch anderem Prozess mit der `kill(2)`-Operation

$\mapsto$  `send()`

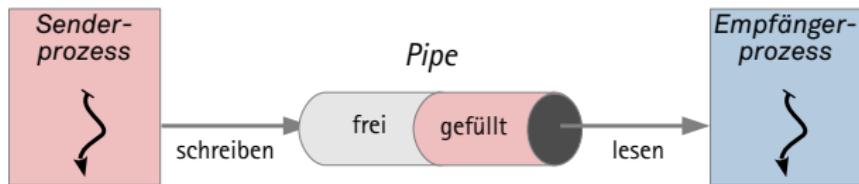
ohne/mit nur eingeschränkter Nutzlast (payload)



Signalisierung und Start der Behandlungsroutine erfolgen logisch gleichzeitig.

Behandlungsroutinen können selbst auch unterbrochen werden.

- FIFO-Kanal zwischen zwei Kommunikationspartnern
  - unidirektional: Sender → Empfänger
  - gepuffert (mit fester Puffergröße)
  - zuverlässig
  - stromorientiert (*stream-based*) ↽ Kommunikation auf Bytegranularität



- Operationen: Schreiben und Lesen
  - Ordnung der Daten bleibt erhalten (Bytestrom)
  - Sender blockiert bei voller Pipe (Schreiben), Empfänger bei leerer Pipe (Lesen)

→ Semantik entspricht **Synchronization send**

## ■ Unbenannte Pipe erzeugen: pipe<sup>(2)</sup>

```
int pipe(int pipefd[2]);
```

**erzeugt** ein neues Pipeobjekt, repräsentiert durch die beiden in `pipefd` zurückgegebenen **Dateideskriptoren**:

- `pipefd[0]` Deskriptor der **Leseseite** (read<sup>(2)</sup>) der Pipe
- `pipefd[1]` Deskriptor der **Schreibseite** (write<sup>(2)</sup>) der Pipe

Zur Verwendung als IPC-Mechanismus muss ein Ende an einen anderen Prozess weitergegeben werden (z. B. mit dup2<sup>(2)</sup> als Standardeingabe setzen vor einem fork<sup>(2)</sup>).

## ■ Benannte Pipe erzeugen: mkfifo<sup>(3)</sup>

```
int mkfifo(const char *  
          ↴ pathname, mode_t mode);
```

**erzeugt** ein benannte Pipe (FIFO-Spezialdatei) `pathname` mit Zugriffsrechten `mode`:

Für IPC muss `pathname` anschließend von Prozessen mit open<sup>(2)</sup> wie jede andere Datei zum Lesen mit read<sup>(2)</sup> oder Schreiben mit write<sup>(2)</sup> geöffnet werden. Sie verhält sich allerdings wie eine Pipe (FIFO).

↪ Pipes sind unter UNIX **Dateiobjekte**

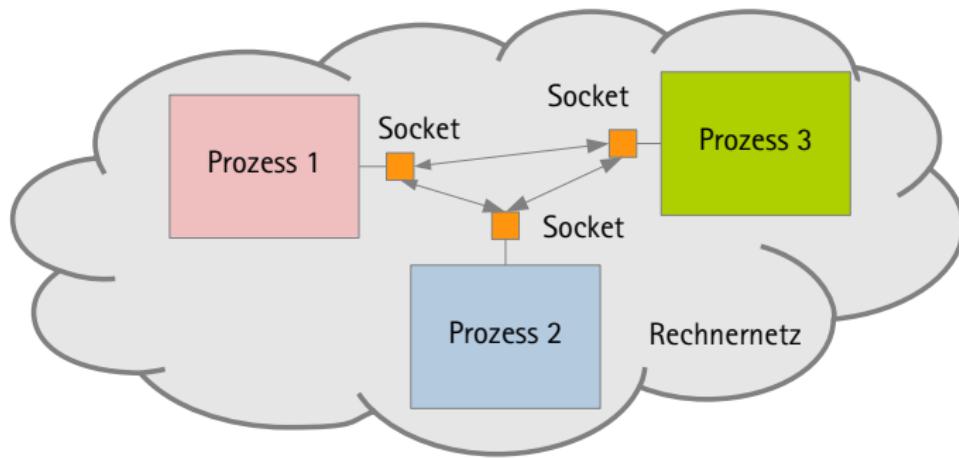
→ 4-32

■ Beispiel: Standardausgabe zweier Programme verknüpfen: **connect**

```
1 // USAGE: connect <prog1> <prog2>
2 #include <stdio.h>
3 #include <unistd.h>
4 enum { READ=0, WRITE=1 };
5
6 int main (int argc, char *argv[]) {
7     int res, fd[2];
8     if (pipe (fd) == 0) {           /* P
9         res = fork ();
10        if (res > 0) {             /* Elternprozess */
11            close (fd[READ]);      /* Leseseite schließen */
12            dup2 (fd[WRITE], 1);    /* Std-Ausgabe in Pipe */
13            close (fd[WRITE]);      /* Deskriptor freigeben */
14            execlp (argv[1], argv[1], NULL); /* Schreiber ausführen */
15        }
16        else if (res == 0) {        /* Kindprozess */
17            close (fd[WRITE]);      /* Schreibseite schließen */
18            dup2 (fd[READ], 0);    /* Std-Eingabe aus Pipe */
19            close (fd[READ]);      /* Deskriptor freigeben */
20            execlp (argv[2], argv[2], NULL); /* Leser ausführen */
21        }
22    }
23    perror("Problem!");
24 }
```

**connect** erwartet zwei Programmdateien, die in zwei Prozessen ausgeführt werden. Die Standardausgabe des ersten wird mit der Standardeingabe des zweiten verknüpft, genau wie bei Verwendung des Pipe-Symbols (`|`) in der Shell: **connect ls wc**  $\rightarrow$  **ls | wc**.

- Ein **Socket** ist ein allgemeiner Kommunikationsendpunkt im Rechnernetz
  - Bidirektional (im Gegensatz zu einer Pipe)
  - Gepuffert
- Sockets abstrahieren von Details des Kommunikationssystems
  - **Domäne** (Protokollfamilie), bestimmt mögliche **Typen** und **Protokolle**



- Sockets können in verschiedenen **Domänen** (*Domains*) angelegt werden
  - **UNIX Domain:** lokaler Rechner
    - Ein UNIX Domain Socket verhält sich wie eine bidirektionale *Pipe*
    - Anlage als Spezialdatei im Dateisystem möglich (analog zu einer *Named Pipe*)
    - **Killer Feature:** Übertragung von Dateideskriptoren an anderen Prozess
  - **Internet Domain:** rechnerübergreifend mit IP
    - die heute vorherrschende Domäne für Sockets
  - {Appletalk, DECNet, NetBIOS, ...} Domain
    - haben wegen der Dominanz von IP heute nur noch geringfügige Bedeutung
- Domäne definiert die möglichen **Protokolle**
  - z. B. Internet Domain: TCP/IP oder UDP/IP
- Domäne definiert die erforderliche Adressierung (**Adressfamilie**)
  - z. B. Internet Domain: IP-Adresse und Port-Nummer

- Der **Typ** definiert die Semantik der Nachrichtenübertragung.

Typische Sockettypen:

- stromorientiert, verbindungsorientiert und gesichert
- nachrichtenorientiert und ungesichert
- nachrichtenorientiert und gesichert

- Das **Protokoll** definiert die Umsetzung der Nachrichtenübertragung.

Protokolle der Internet Domain:

- TCP/IP-Protokoll
  - stromorientiert, verbindungsorientiert und gesichert
- UDP/IP-Protokoll
  - nachrichtenorientiert, verbindungslos, ungesichert
    - Nachrichten können verloren oder dupliziert werden
    - Nachrichten können in anderer Reihenfolge eintreffen
    - Paketgrenzen bleiben jedoch erhalten (Datagramm-Protokoll)

- Angabe von Protokoll und Typ ist in der Praxis oft redundant

- Protokoll definiert bereits (einzig) möglichen Typen – und umgekehrt

■ Socket erzeugen: socket<sup>(2)</sup>

```
int socket(int domain, int
    ↴ type, int protocol);
```

**erzeugt** ein neues Socket-Objekt vom Typ `type`: `SEQ_STREAM`, `SEQ_DGRAM`, ... in der Domäne `domain`: `AF_UNIX`, `AF_INET`, `AF_INET6`, ... und liefert seinen **Dateideskriptor** zurück.

■ Erzeugtem Socket eine Adresse zuweisen: bind<sup>(2)</sup>

```
int bind(int sockfd, const
    ↴ struct sockaddr *addr,
    ↴ socklen_t addrlen);
```

**bindet** das mit socket<sup>(2)</sup> erzeugte Socket `sockfd` an die Adresse `addr` seiner Domäne zu. Der Aufbau des in `addr` übergebenen Puffers ist familien spezifisch. Für die Internet-Adressfamilie (`AF_INET` mit IPv4):

```
struct sockaddr_in {
    sa_family_t sin_family; // = AF_INET
    in_port_t   sin_port;   // = port number
    in_addr_t   sin_addr;   // = IP number
};
```

Für IPv6 gibt es entsprechend `AF_INET6` und `struct sockaddr_in6`.

→ Sockets sind unter UNIX **Dateiobjekte**

→ 4-32

- Neben read<sup>(2)</sup> und write<sup>(2)</sup> gibt es jedoch zusätzlich noch spezielle Operationen

- Mit **Datagram Sockets** verbindungslos arbeiten: sendto<sup>(2)</sup> und recvfrom<sup>(2)</sup>
  - Kein Verbindungsaufbau notwendig
  - Senden und empfangen einzelner Datagramme (unsicher)

```
ssize_t sendto(int sockfd, const void *  
    ↳ buf, size_t len, int flags, const  
    ↳ struct sockaddr *dest_addr, socklen_t  
    ↳ addrlen);
```

```
ssize_t recvfrom(int sockfd, void *buf,  
    ↳ size_t len, int flags, struct sockaddr  
    ↳ *src_addr, socklen_t *addrlen);
```

**sendet** ein **einzelnes Datagramm** mit Daten an der Adresse **buf** und der Länge **len** an das durch **address** und **address\_len** beschriebene Ziel.

**empfängt** ein **einzelnes Datagramm** und legt es ab der Adresse **buf** mit der Maximallänge **len** im Speicher ab. Optional kann in **src\_addr** und **addrlen** die Adresse des Senders empfangen werden.

- Die maximale Größe der zu empfangenden/sendenden Nachricht ist (protokollspezisch) **beschränkt** auf die Größe eines Datagramms.

- Mit Stream Sockets verbindungsorientiert arbeiten  $\rightsquigarrow$  Client–Server–Prinzip
  - Client muss zunächst zum Server eine Verbindung aufbauen: connect<sup>(2)</sup>
  - Server muss dafür Verbindungen entgegen nehmen: listen<sup>(2)</sup>, accept<sup>(2)</sup>
  - Verbindung kann von jeder Seite mit close<sup>(2)</sup> beendet werden.
- **Client:** Verbindungsauftbau (bei stromorientierten Sockets)

```
int connect(int sockfd,  
           ↴ const struct sockaddr *  
           ↴ addr, socklen_t addrlen  
           ↴ );
```

**verbindet** das mit socket<sup>(2)</sup> erzeugte Socket `sockfd` mit dem Server-Endpunkt an der Adresse `addr`. Der Aufbau des in `addr` übergebenen Puffers ist familienspezifisch. Für die Internet-Adressfamilie (`AF_INET` mit IPv4):

```
struct sockaddr_in {  
    sa_family_t sin_family; // = AF_INET  
    in_port_t   sin_port;   // = port number  
    in_addr_t   sin_addr;  // = IP number  
};
```

Für IPv6 gibt es entsprechend `AF_INET6` und `struct sockaddr_in6`.

- **Server:** Auf Verbindungen vorbereiten: listen<sup>(2)</sup>

```
int listen(int sockfd, int  
         ↴ backlog);
```

**stellt** das mit socket<sup>(2)</sup> erzeugte und mit bind<sup>(2)</sup> gebundene Socket **sockfd** in den *listening*-Modus, mit dem es auf eingehende Verbindungsanfragen von Clients wartet.

- **Server:** Auf Verbindungen warten und annehmen: accept<sup>(2)</sup>

```
int accept(int sockfd,  
          ↴ struct sockaddr *addr,  
          ↴ socklen_t *addrlen);
```

**akzeptiert** eine Verbindungsanfrage auf dem im *listening*-Modus befindlichen Socket **sockfd** und gibt einen neuen Socket als **Dateideskriptor** zurück, über die Verbindung mit dem Client repräsentiert und über den der Datenaustausch stattfindet. Optional kann in **src\_addr** und **addrlen** die Adresse des verbundenen Clients empfangen werden.

- **Client/Server:** Datenaustausch über Verbindung durchführen

- Wie bei jedem Dateiobjekt: Mit read<sup>(2)</sup> und write<sup>(2)</sup>.
- Alternativ kann auch recv<sup>(2)</sup> und send<sup>(2)</sup> verwendet werden.

## ■ Beispiel: HTTPEcho – ein minimaler Webserver

```
1 #include <...>
2 #define PORT 6789
3 #define MAXREQ (4096*1024)
4
5 char buffer[MAXREQ], body[MAXREQ], msg[MAXREQ];
6
7 void errorexit(const char *msg) {
8     perror(msg);
9     exit(1);
10 }
11
12 int main(int argc, char *argv[]) {
13     int sockfd, newsockfd, n;
14     socklen_t clilen;
15     struct sockaddr_in serv_addr = {0}, cli_addr;
16
17     sockfd = socket(PF_INET, SOCK_STREAM, 0);
18     if( sockfd < 0 ) errorexit("ERROR opening socket");
19
20     serv_addr.sin_family = AF_INET;
21     serv_addr.sin_addr.s_addr = INADDR_ANY;
22     serv_addr.sin_port = htons(PORT);
23     if( bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
24         errorexit("ERROR on binding");
25
26     listen(sockfd, 5);
27     printf("HTTPEcho is listening on localhost:%d\n", PORT);
```

Socket erstellen

an PORT auf allen Interfaces binden

und in den *Listening*-Modus setzen

## ■ Beispiel: HTTPEcho – ein minimaler Webserver (Fortsetzung)

```
28 while (1) {  
29     clilen = sizeof(cli_addr);  
30     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);  
31     if( newsockfd < 0 ) errorexit("ERROR on accept");  
32  
33     n = read(newsockfd, buffer, sizeof(buffer)-1);  
34     if( n < 0 )  
35         errorexit ("ERROR reading from socket");  
36     buffer[n] = 0;  
37     snprintf (body, sizeof (body),  
38             "<html>\n<body>\n"  
39             "<h1>Hallo BS-Student :-)</h1>\n"  
40             "Die Anfrage deines Browsers war:\n"  
41             "<pre>%s</pre>\n"  
42             "</body>\n</html>\n", buffer);  
43     snprintf (msg, sizeof (msg),  
44             "HTTP/1.0 200 OK\n"  
45             "Content-Type: text/html\n"  
46             "Content-Length: %lu\n\n%s", strlen (body), body);  
47  
48     n = write(newsockfd,msg,strlen(msg));  
49     if( n < 0 ) errorexit("ERROR writing to socket");  
50  
51     close (newsockfd);  
52 }  
53 }
```

Auf Client warten und annehmen  
Anfrage lesen  
HTTP-konforme Antwort erstellen  
an Client ausliefern  
Verbindung beenden

## 10.7 Zusammenfassung

- Es gibt grundsätzlich zwei Arten der Interprozesskommunikation
  - speichergekoppelt (über gemeinsamen Speicher)
    - war Thema in der letzten Vorlesung (Synchronisation)
  - **nachrichtengekoppelt** (über send() und receive())
    - explizite Kommunikation über Nachrichten
    - potentiell auch über Rechnergrenzen
- Zu Berücksichtigen ist eine Vielzahl an Dimensionen und Varianten
  - Interaktionsmuster und Synchronität
  - Direkte vs. indirekte Adressierung
  - Stromorientiert vs Nachrichtenorientiert
  - Verbindungslos vs. Verbindungsorientiert
  - Sicherer vs. Unsicherer Transfer
  - Nachrichtensemantiken
  - Wie finden Partner zusammen
  - Granularität des Datenversands
  - Zuordnung mehrere Nachrichten
  - Zustellungsgarantien
- UNIX bietet mehrere IPC-Mechanismen
  - Signale, (Named) Pipes, Sockets (auch über Rechnergrenzen)
  - Durch das Internet sind IP-basierte Sockets zum verbreiteten Standard geworden



Technische  
Universität  
Braunschweig



Institute of Operating Systems  
and Computer Networks  
Reliable System Software



# Betriebssysteme (BS)

## Teil D Speicher und Zugriffsschutz

**Christian Dietrich**

Wintersemester 2024



# Überblick: Teil D Speicher und Zugriffsschutz

## 11 Speicherorganisation

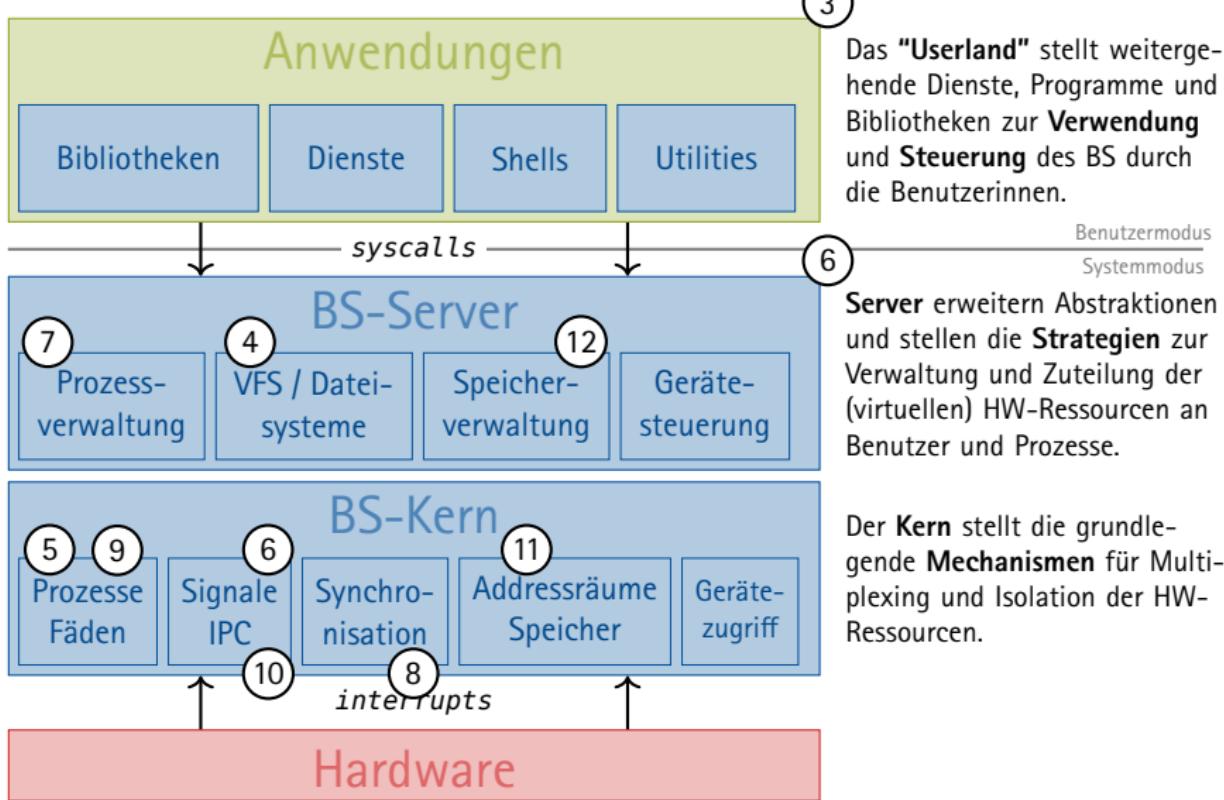
- 11.1 Einordnung
- 11.2 Grundlagen
- 11.3 Vom C-Programm zum Speicher ( $E_5 \rightsquigarrow E_3$ )
- 11.4 Freispeicherverwaltung
- 11.5 Adressraumlehre ( $E_3 \rightsquigarrow E_2$ )
- 11.6 Realer Adressraum
- 11.7 Relokation
- 11.8 Logischer Adressraum
- 11.9 Zusammenfassung

## 12 Speichervirtualisierung

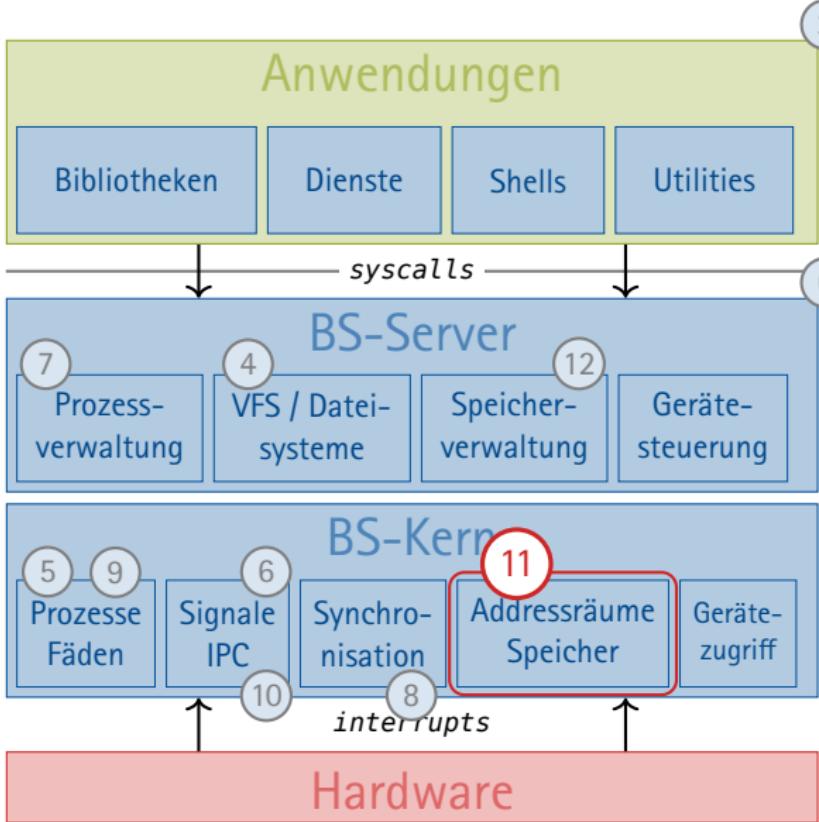


## 11.1 Einordnung

# 11 Speicherorganisation – Einordnung



# 11 Speicherorganisation – Einordnung



3

6

12

4

5

9

10

8

11

Das "Userland" stellt weitergehende Dienste, Programme und Bibliotheken zur **Verwendung** und **Steuerung** des BS durch die Benutzerinnen.

Benutzermodus  
Systemmodus

Server erweitern Abstraktionen und stellen die **Strategien** zur Verwaltung und Zuteilung der (virtuellen) HW-Ressourcen an Benutzer und Prozesse.

Der Kern stellt die grundlegende **Mechanismen** für Multiplexing und Isolation der HW-Ressourcen.

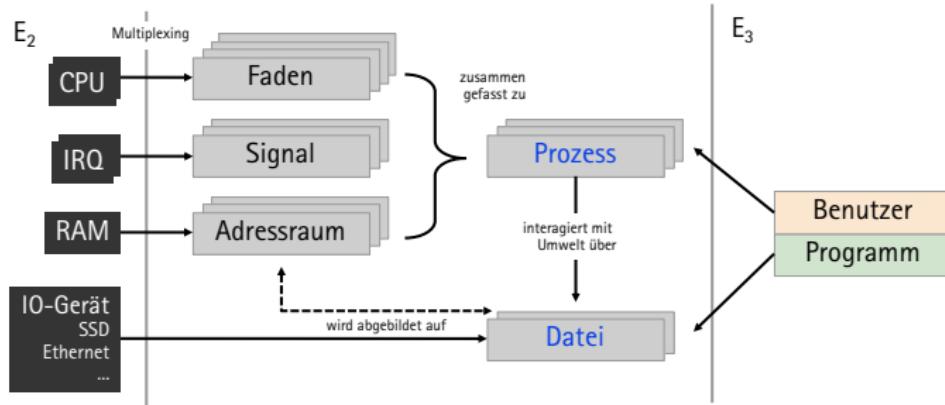
## Worum geht es in diesem Kapitel?

- **Verstehen** grundlegender Prinzipien und Konzepte der Speicherorganisation
    - **in der Mehrebenenmachine** – vom C-Programm zum Speicher im Prozess
    - **in der Speicherhierarchie** – vom Hauptspeicher bis zur Festplatte
  - **Erläutern** elementarer Strategien der Speicherverwaltung
    - Statische und dynamische Speicherverwaltung
    - Speicherverwaltung auf dem Stapel
    - Freispeicherverwaltung mit Listen, Bitmaps, Halbierungsverfahren
    - Fragmentierung und Verschnitt
  - **Diskutieren** der Adressraumlehre
    - realer, logischer und virtueller Adressraum
    - Laden von Programmen, Relokation
    - Logische Adressräume als Fundament der horizontalen Isolation (Schutz)
- Speicherverwaltung **systemorientiert** betrachten im Zusammenspiel von **Compiler**, **Betriebssystem** und **Hardware**.

## ■ Betriebssystem ↪ Multiplexing und Isolation der Hardware

- Mehrprogramm- und Mehrbenutzerbetrieb
- Verwaltung der Betriebsmitteln: CPU, Speicher, ...
- Interne Aufgabenteilung in Kern (Mechanismen) und Server (Strategien/Dienste)
- Planung der Zuteilung von Ressourcen an Benutzer und Prozesse

## ■ Bereitstellung durch elementare Abstraktionen (Bsp. UNIX)



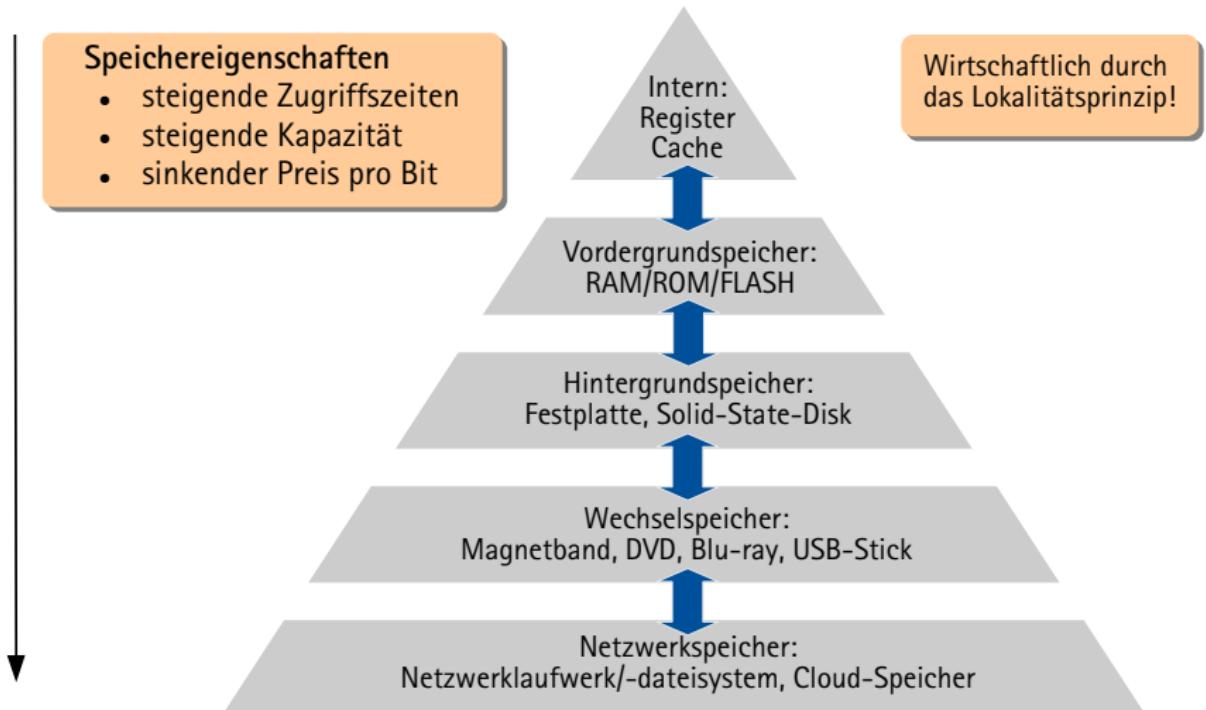
- Fokus bislang war Multiplexing des Prozessors durch Prozesse
- ↪ Nun: **Verwaltung von Speicher**



## 11.2 Grundlagen

# Verwaltung von Speicher: Speicherhierarchie

Die Speicherhierarchie eines Rechensystems



Unabhängig von der Ebene stellen sich immer dieselben Fragen

## ■ **Allokationsstrategie** (*allocation policy*)

- **Wann** soll benötigter Speicher angefordert werden?
  - Statisch – sobald man weiß, dass man ihn braucht?
  - Dynamisch – erst wenn man ihn wirklich braucht?

## ■ **Platzierungsstrategie** (*placement policy*)

- **Woher** soll benötigter Speicher genommen werden?
  - So, dass der Verschnitt minimal/maximal wird?
  - So, dass es am schnellsten möglich ist?

## ■ **Ladestrategie** (*fetch policy*)

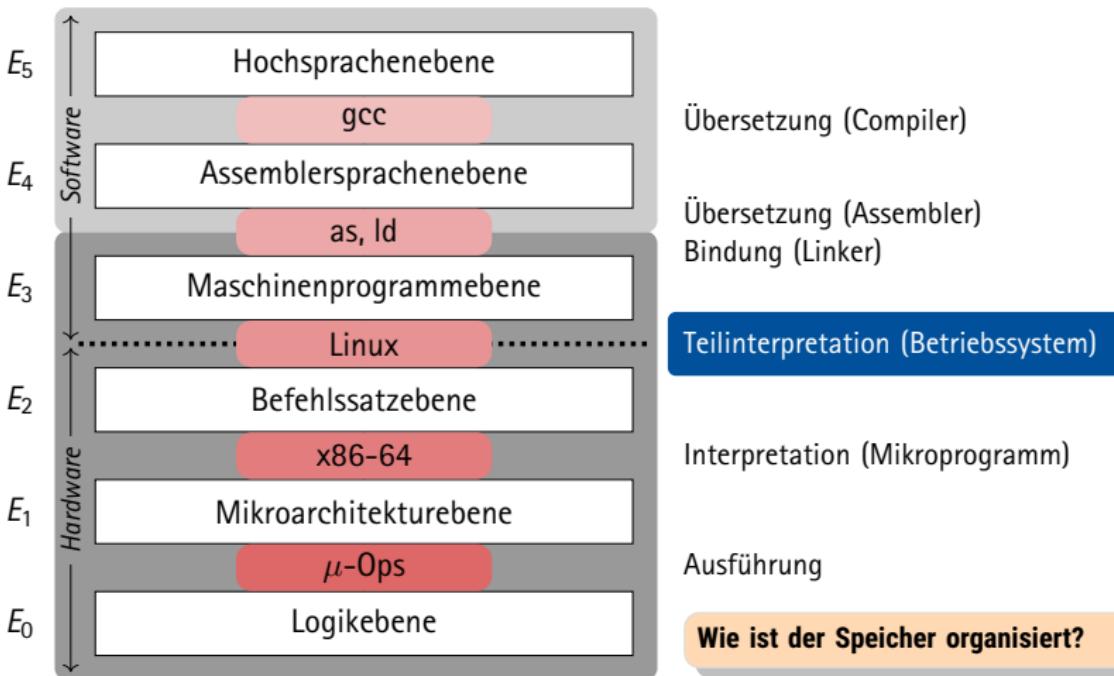
- **Wann** sind Speicherinhalte einzulagern?
  - Auf Anforderung oder im Vorraus?

## ■ **Ersetzungsstrategie** (*replacement policy*)

- **Welche** Speicherblöcke sind auszulagern, falls Speicher knapp wird?
  - Die ältesten?
  - Die am längsten ungenutzten?



## Übersetzung und Interpretation auf einem Linux-PC





## 11.3 Vom C-Programm zum Speicher ( $E_5 \rightsquigarrow E_3$ )



```
int a;                      // a: global, uninitialized
int b = 1;                   // b: global, initialized
const int c = 2;              // c: global, const

void main() {
    static int s = 3;          // s: local, static, initialized
    int x, y;                  // x: local, auto; y: local, auto
    char* p = malloc( 100 );   // p: local, auto; *p: heap (100 byte)
}
```

Wo kommt der Speicher für Variablen und Code her?

## ■ Statische Allokation – Reservierung beim Übersetzen / Linken $E_5 \rightsquigarrow E_3$

- Betrifft alle globalen/statischen Variablen, sowie den Code
- Allokation durch Platzierung in einer **Sektion**

.text – enthält den Programmcode

main()  
a

.bss – enthält alle mit 0 initialisierten Variablen

b,s

.data – enthält alle mit anderen Werten initialisierten Variablen

c

.rodata – enthält alle unveränderlichen Variablen

## ■ Dynamische Allokation – Reservierung zur Laufzeit des Programms $E_{3/2}$

- Betrifft lokale auto-Variablen und explizit angeforderten Speicher

Stack – enthält alle **aktuell lebendigen** auto-Variablen

x,y,p

Heap – enthält explizit mit **malloc()** angeforderte Speicherbereiche

\*p



# Speicherorganisation

$E_5 \rightsquigarrow E_3 \rightsquigarrow E_2$

Speicherabbild: Vom Programm zum Prozess

```
int a;                                // a: global, uninitialized
int b = 1;                             // b: global, initialized
const int c = 2;                         // c: global, const

void main() {
    static int s = 3;                   // s: local, static, initialized
    int x, y;                          // x: local, auto; y: local, auto
    char* p = malloc( 100 );           // p: local, auto; *p: heap (100 byte)
}
```

$E_{5/4}$

Quellprogramm

compile & link

| Symbol Table <a> |            |
|------------------|------------|
| .data            | s=3<br>b=1 |
| .rodata          | c=2        |
| .text            | main       |
| $E_3 \dots$      |            |
| ELF Header       |            |

ELF-Binary (Maschinenprogramm)

Beim Übersetzen und Linken werden alle **statisch allozierten Elementen** in entsprechenden Sektionen des **Maschinenprogramms** platziert; ihre Adresse und Größe wird in der **Symboltabelle** markiert.  
Optimierung: Mit 0 vorbelegte Variablen werden in der virtuellen .bss-Sektion platziert und belegen so keinen Speicher in der ELF-Datei.



# Speicherorganisation

$E_5 \rightsquigarrow E_3 \rightsquigarrow E_2$

Speicherabbild: Vom Programm zum Prozess

```
int a;                                // a: global, uninitialized
int b = 1;                             // b: global, initialized
const int c = 2;                         // c: global, const

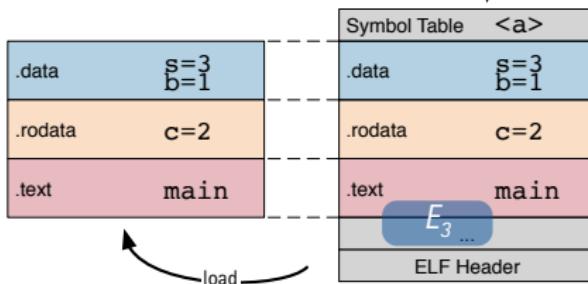
void main() {
    static int s = 3;                   // s: local, static, initialized
    int x, y;                          // x: local, auto; y: local, auto
    char* p = malloc( 100 );           // p: local, auto; *p: heap (100 byte)
}
```

$E_{5/4}$

Quellprogramm

compile & link

RAM



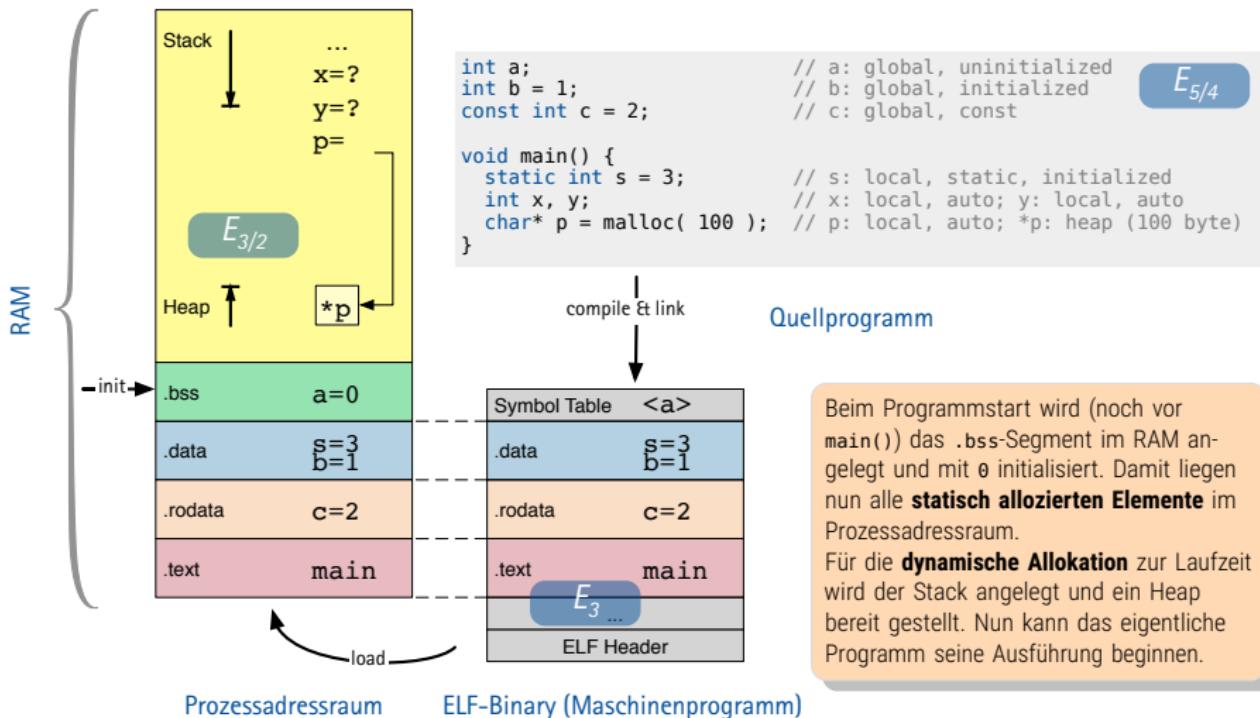
Mit exec<sup>(2)</sup> werden die Sektionen des Maschinenprogramms (alle statisch allozierten Elemente) in einen **Prozessadressraum** geladen (dabei ist ggfs. eine Relokation erforderlich  $\leftrightarrow [11-34]$ ). Sie sind nun im RAM hinterlegt und können vom Prozessor angesprochen werden.



# Speicherorganisation

$E_5 \rightsquigarrow E_3 \rightsquigarrow E_2$

Speicherabbild: Vom Programm zum Prozess



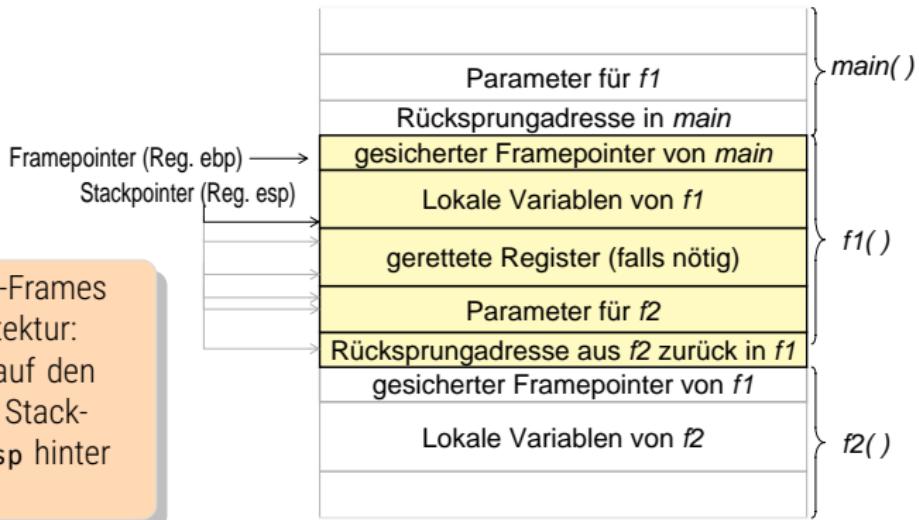
- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
  - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe beispielsweise mit malloc<sup>(3)</sup> und free<sup>(3)</sup>
  - `void* malloc( size_t n )` fordert einen Speicherblock der Größe  $n$  an.  
Rückgabe bei Fehler: 0-Zeiger (`NULL`)
  - `void free( void* pmem )` gibt einen zuvor mit `malloc()` angeforderten Speicherblock vollständig wieder frei
- Beispiel

```
#include <stdlib.h>
int* intArray( uint16_t n ) {    // alloc int[n] array
    return (int*) malloc( n * sizeof int );
}

void main() {
    int* array = intArray(100);    // alloc memory for 100 ints
    if( array ) {                // malloc() returns NULL on failure
        ...                      // if succeeded, use array
        array[99] = 4711;
        ...
        free( array );           // free allocated block (** IMPORTANT! **)
    }
}
```

# Dynamische Speicherallokation: Stack

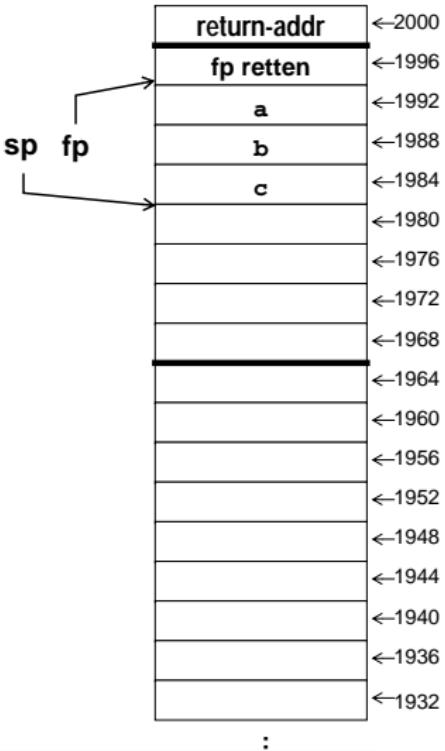
- Lokale Variablen, Funktionsparameter und Rücksprungadressen werden von Übersetzer und Prozessor auf dem **Stack** (Stapel, Keller) verwaltet
  - Stack „wächst“ (architekturabhängig) „von oben nach unten“
  - Prozessorregister [e]sp zeigt immer auf den nächsten freien Eintrag
- Die Verwaltung erfolgt in Form von **Stack-Frames**



# Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
  
    a = 10;  
    b = 20;  
  
    f1(a, b);  
    return(a);  
}
```

Stack-Frame für  
main erstellen  
 $\&a = fp-4$   
 $\&b = fp-8$   
 $\&c = fp-12$

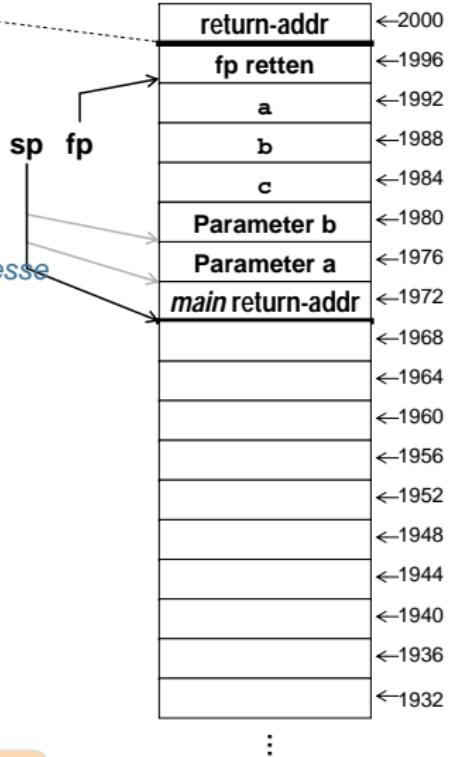


Beispiel hier für 32-Bit-Architektur (4-Byte `ints`), `main()` wurde soeben betreten

# Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```

Parameter  
auf Stack legen  
Bei Aufruf  
Rücksprungadresse  
auf Stack legen



main() bereitet den Aufruf von f1(int, int) vor

# Stack-Aufbau bei Funktionsaufrufen

```
int main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

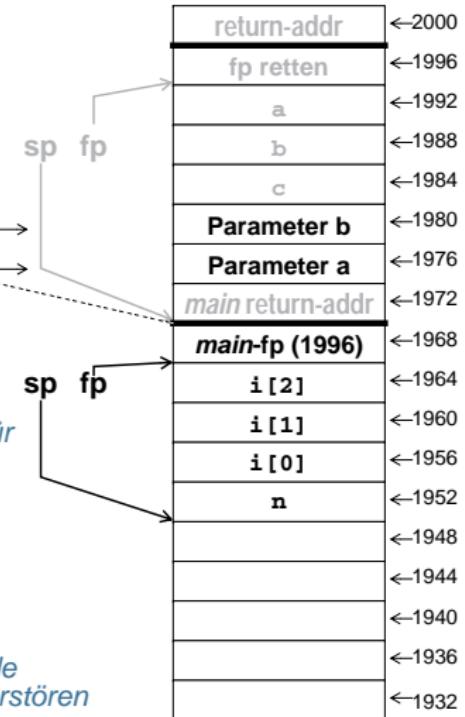
```
int f1(int x, int y) {
    int i[3];
    int n;
    x++;
    n = f2(x);
    return(n);
}
```

Stack-Frame für  
f1 erstellen  
und aktivieren

$\&x = fp+8$   
 $\&y = fp+12$   
 $\&(i[0]) = fp-12$   
 $\&n = fp-16$

i[4] = 20 würde  
return-Addr. zerstören

f1() wurde soeben betreten



# Stack-Aufbau bei Funktionsaufrufen

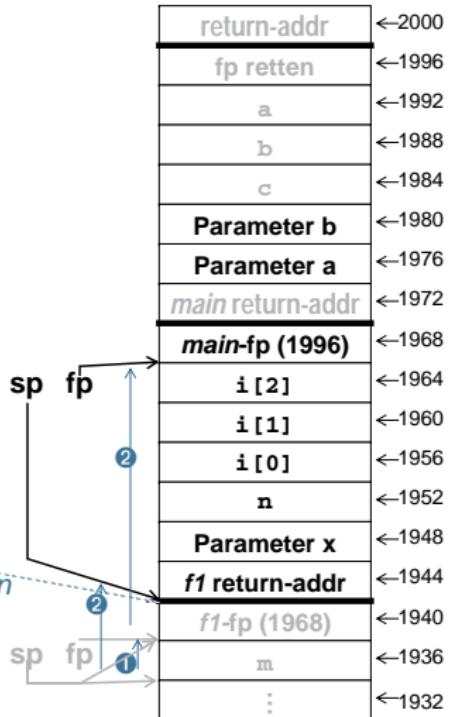
```
int main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}

int f1(int x, int y) {
    int i[3];
    int n;
    x++;
    n = f2(x);
    return(n);
}

int f2(int z) {
    int m;
    m = 100;
    return(z+1);
}
```

Stack-Frame von  
*f2* abräumen

- ①  $sp = fp$   
②  $fp = pop(sp)$



*f2()* bereitet die Terminierung vor (wurde von *f1()* aufgerufen und ausgeführt)

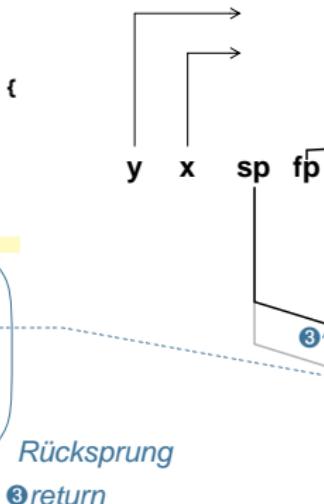
# Stack-Aufbau bei Funktionsaufrufen

```
int main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

```
int f1(int x, int y) {
    int i[3];
    int n;
    x++;
    n = f2(x);
    return(n);
}
```

```
int f2(int z) {
    int m;
    m = 100;
    return(z+1);
}
```

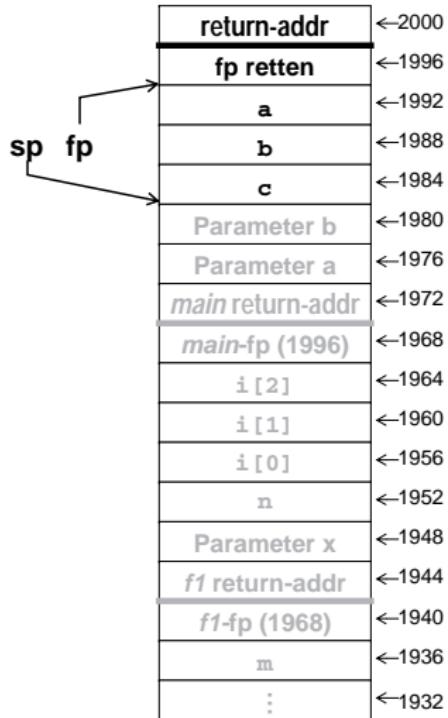
f2() wird verlassen



|                         |       |
|-------------------------|-------|
| return-addr             | ←2000 |
| fp retten               | ←1996 |
| a                       | ←1992 |
| b                       | ←1988 |
| c                       | ←1984 |
| <b>Parameter b</b>      | ←1980 |
| <b>Parameter a</b>      | ←1976 |
| <i>main</i> return-addr | ←1972 |
| <b>main-fp (1996)</b>   | ←1968 |
| i [2]                   | ←1964 |
| i [1]                   | ←1960 |
| i [0]                   | ←1956 |
| n                       | ←1952 |
| <b>Parameter x</b>      | ←1948 |
| <i>f1</i> return-addr   | ←1944 |
| <b>f1-fp (1968)</b>     | ←1940 |
| m                       | ←1936 |
| ⋮                       | ←1932 |

# Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
  
    a = 10;  
    b = 20;  
  
    f1(a, b);  
  
    return(a);
```



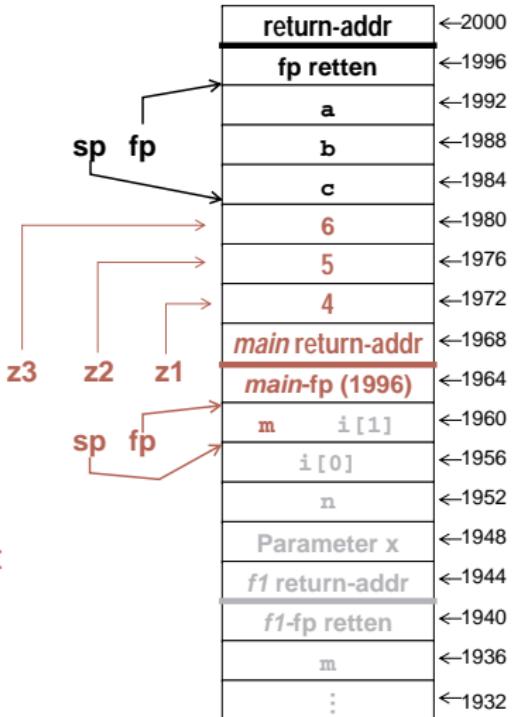
zurück in main()

# Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    f3(4, 5, 6);  
}
```

*Was wäre, wenn man nach f1 jetzt eine Funktion f3 aufrufen würde?*

```
int f3(int z1, int z2, int z3) {  
    int m;  
  
    return(m);  
}
```



m wird nicht initialisiert → „erbt“ alten Wert vom Stapel

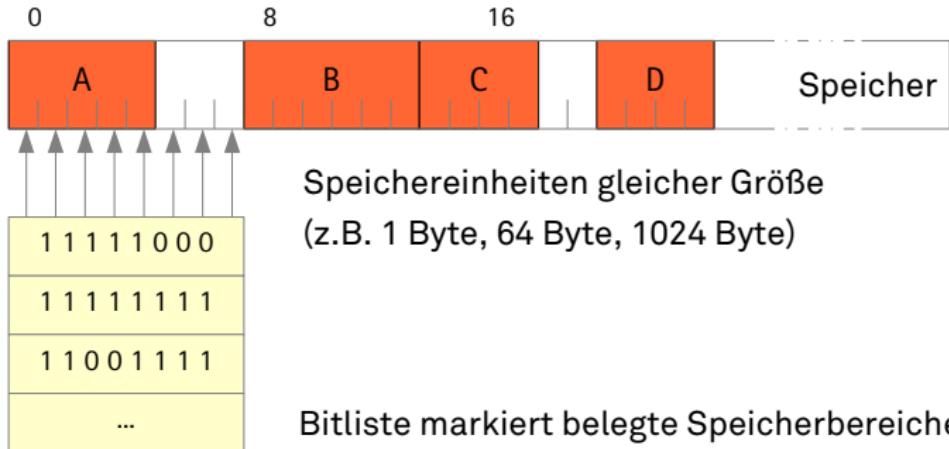


## 11.4 Freispeicherverwaltung

- Anforderung/Freigabe von **Stapelspeicher** ist **sehr effizient**
    - Anforderung von  $n$  Bytes:  $\rightarrow$  Vermindern des Stapelzeigers um  $n$
    - Freigabe von  $n$  Bytes:  $\rightarrow$  Erhöhen des Stapelzeigers um  $n$
  - Möglich (nur) durch Anforderungen/Freigabe in **LIFO-Reihenfolge**
    - Der zuletzt angeforderter Block wird immer zuerst freigeben
      - $\rightarrow$  konstruktiv kein Verschnitt oder Loch im Freispeicher
  - In anderen Fällen erforderlich: **Platzierungsstrategie**
    - Woher soll benötigter Speicher genommen werden?
    - Wie finde ich schnell passende Speicherblöcke?
    - Wie minimiere ich (internen/externen) Verschnitt?
- $\rightarrow$  Wesentlicher Punkt: **Freispeicherverwaltung**

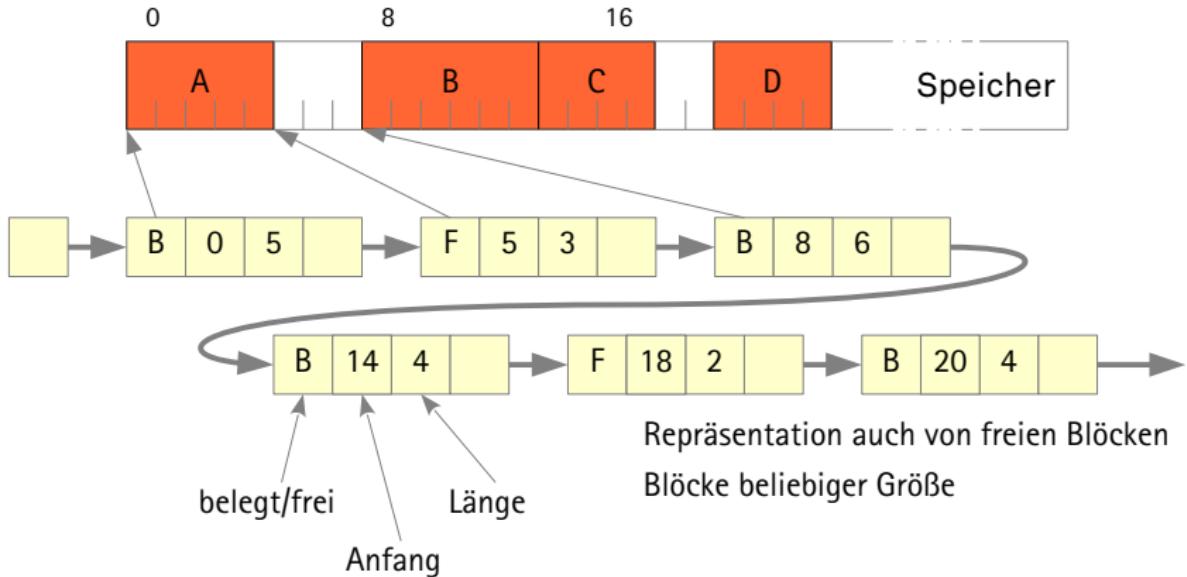


- Speichereinheiten werden durch je ein Bit als frei oder belegt markiert
  - Speicherblöcke sind Vielfaches der Einheitsgröße
  - Index des Bits  $\mapsto$  Blockadresse



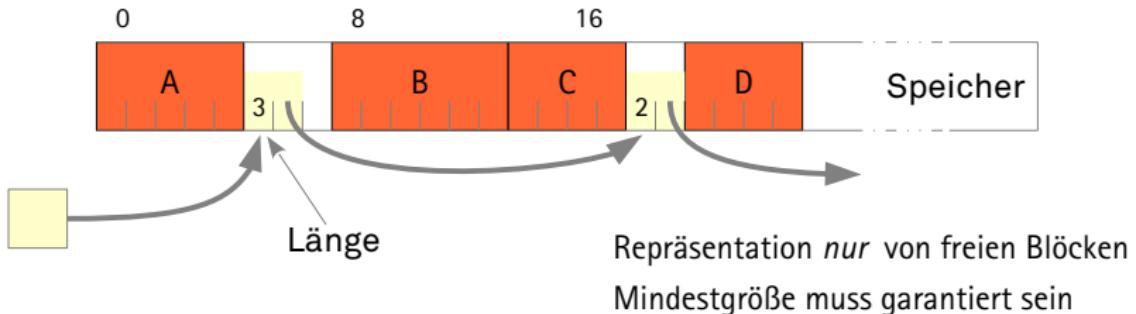
# Freispeicherverwaltung: Verkettete Liste (extern)

- Speicherblöcke werden extern verkettet
  - Repräsentation auch von freien Blöcken (Löchern)



# Freispeicherverwaltung: Verkettete Liste (intern)

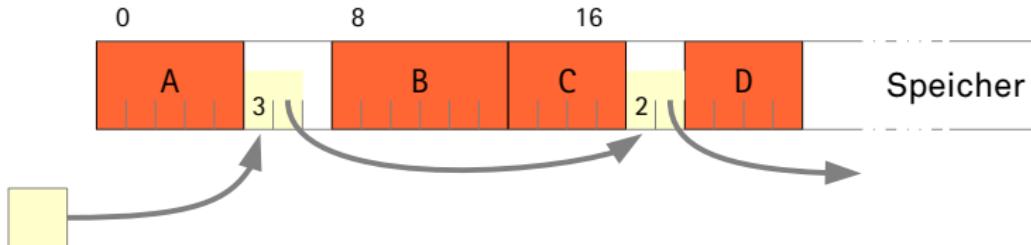
- Nur die freien Speicherblöcke (Löcher) werden verkettet
  - Verwaltung der Liste direkt im verwalteten Speicher



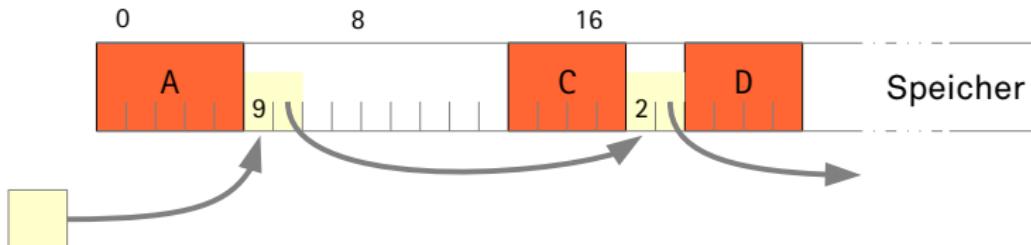


# Freispeicherverwaltung: Verschmelzung von Löchern

- Bei der Freigabe von Speicher Löcher verschmelzen
  - Verringerung der Fragmentierung



- Nach `free(B)` (Freigabe von B):



## Auf der Basis von unterschiedlich sortierten Löcherlisten

- **First Fit** - nimm erstes passendes Loch
  - schnell, aber Konzentration kleiner Lücken am Anfang der Löcherliste
  - Aufwand der Suche ist zu Beginn  $O(1)$ , kann jedoch zu  $O(n)$  degenerieren
- **Next Fit** - nimm nächstes passendes Loch
  - wie *First Fit*, aber starte Suche beim zuletzt zugewiesenen Loch
  - verteilt Entstehung kleiner Löcher gleichmäßiger
- **Best Fit** - nimm kleinstes, noch passendes Loch
  - minimiert Risiko, irgendwann kein genügend große Loch mehr zu finden
  - Aufwand der Suche ist immer  $O(n)$
- **Worst Fit** - nimm größtes passendes Loch
  - minimiert Risiko der Zerstückelung in zu kleine Löcher
  - Aufwand der Suche ist immer  $O(n)$

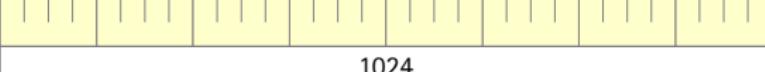
**Problem:** Mit der Zeit **zu kleine Löcher**  $\rightsquigarrow$  **externer Verschnitt**



# Freispeicherverwaltung: Halbierungsverfahren

Buddy

- Einteilung in dynamische Bereiche der Größe  $2^n$ 
  - Vermindert Fragmentierung durch bessere Verschmelzbarkeit
  - Effiziente Algorithmen für Anforderung und Freigabe ( $O(\log n)$ )

|                                                                                    | 0   | 128 | 256 | 384 | 512  | 640 | 768 | 896 | 1024 |
|------------------------------------------------------------------------------------|-----|-----|-----|-----|------|-----|-----|-----|------|
|  |     |     |     |     |      |     |     |     |      |
| Anfrage 70                                                                         | A   | 128 |     | 256 |      |     |     |     | 512  |
| Anfrage 35                                                                         | A   | B   | 64  |     | 256  |     |     |     | 512  |
| Anfrage 80                                                                         | A   | B   | 64  | C   | 128  |     |     |     | 512  |
| Freigabe A                                                                         | 128 | B   | 64  | C   | 128  |     |     |     | 512  |
| Anfrage 60                                                                         | 128 | B   | D   | C   | 128  |     |     |     | 512  |
| Freigabe B                                                                         | 128 | 64  | D   | C   | 128  |     |     |     | 512  |
| Freigabe D                                                                         |     | 256 |     | C   | 128  |     |     |     | 512  |
| Freigabe C                                                                         |     |     |     |     | 1024 |     |     |     |      |

Folge von Anforderungen und Freigaben über die Zeit. Durch fortlaufende **Halbierung** werden nur Speicherblöcke der Größe  $2^n$  herausgegeben; bei der Freigabe wird nur auf Größen  $2^n$  verschmolzen.

**Problem:** Aufrunden auf nächste Zweierpotenz  $\rightsquigarrow$  **interner Verschnitt**

# Freispeicherverwaltung: Fragmentierung und Verschnitt

Übererfüllung von Anforderungen oder zu kleine Restöcher durch Fragmentierung führen zu **Verschnitt** (Slack)  $\rightsquigarrow$  Verschwendungen.

- Ausprägung der **Fragmentierung** hängt von der Zuteilungsgranularität ab:
  - **Interne Fragmentierung**  $\mapsto$  feste Granulatgrößen: *Buddy, Bitliste*
    - die angeforderte Größe ist kleiner als der zugeteilte Block  
(z.B. *Buddy*: aufrunden auf nächste Zweierpotenz)
    - der „lokale“ Verschnitt ist durch Anforderer nutzbar  $\rightsquigarrow$  sollte es aber nicht sein!
  - $\hookrightarrow$  **Verschwendungen**, ist (durch das Betriebssystem) unvermeidbar
  - **Externe Fragmentierung**  $\mapsto$  beliebige Zuteilungsgrößen: *First Fit, Best Fit, ...*
    - die angeforderte Größe ist zu groß für jedes Loch  
(auch wenn *in Summe* noch genügend Speicher verfügbar ist)
    - der „globale“ Verschnitt ist ggfs. nicht mehr zuteilbar
  - $\hookrightarrow$  **Verlust**, wäre (durch das Betriebssystem) aufwändig vermeidbar
- Externe Fragmentierung kann durch **Verschmelzung** verringert und durch **Kompaktifizierung** aufgelöst werden

$\hookrightarrow$  11-21

$\hookrightarrow$  12-13



- Freispeicherverwaltung findet auf unterschiedlichsten Ebenen statt
  - **Innerhalb eines Prozesses**
    - Verwaltung des Haldenspeichers (*Heap*)
    - Dynamische Allokation von Speicher mit malloc<sup>(3)</sup> und free<sup>(3)</sup>
  - **Innerhalb des Prozessadressraums**
    - Verwaltung von Adressbereichen (logischer Adressraum) für Segmente
    - Dynamische Allokation mit mmap<sup>(2)</sup>, Laden dynamischer Bibliotheken
  - **Innerhalb des Betriebssystems**
    - Verwaltung des physischen Speichers (realer Adressraum)
    - Verwaltung des Hintergrundspeichers (*swap storage*)
  - **Innerhalb des Dateisystems**
    - Verwaltung der freien und belegten Blöcke des Datenträgers
    - Aufheben externer Fragmentierung durch Kompaktifizierung
- Zu beachten sind Aufwand, externer und interner Verschnitt
  - Die grundlegende Probleme und Strategien sind übertragbar



## 11.5 Adressraumlehre ( $E_3 \rightsquigarrow E_2$ )

- Ein laufender Prozess generiert eine **Folge von Adresszugriffen**
    - anhand der im Programmtext codierten Daten und Zugriffsmuster
    - abhängig von seiner **statischen Abbildung** durch Compiler und Linker
    - abhängig von den **dynamischen Eingabedaten** zur Laufzeit
  - Der **Wertevorrat** (Domäne) möglicher Adressen ist **immer beschränkt!**
    - obere Schranke: Adressbreite der Hardware (z. B. 32-Bit  $\mapsto$  4GiB)
    - untere Schranke: initiale Speicherallokation des Maschinenprogramms
    - zur Laufzeit: dazwischen dynamisch wachsend/schrumpfend
  - Der einem Prozess **zugebilligte** Wertevorrat definiert den **Adressraum**
    - in den der Prozess (logisch/physisch) **eingeschlossen** ist
    - durch Hardware, Betriebssystem – oder bereits konstruktiv auf der Maschinenprogrammebene (typsichere Sprachen, JVM)
- ☞ Implementierung von Horizontaler Isolation  $\hookrightarrow$  3-20

- **Realer Adressraum  $A_r$**   $E_2$  (Befehlssatzebene)
  - Durch **Prozessor und Rechensystem** definierter Wertvorrat von Adressen
    - $|A_r| = 2^n$  mit (üblicherweise)  $n \in \{16, 32, 48, 64\}$
    - Nicht jede **reale Adresse**  $ra \in A_r$  ist gültig:  $A_r$  kann Lücken aufweisen!
  - ~~> **Hauptspeicher HS**, adressierbar durch einen/mehrere Bereiche aus  $A_r$
- **Logischer Adressraum  $A_l$**   $E_{5,4,3}$  (Maschinenprogrammebene)
  - Durch ein **Programm P** definierter Wertvorrat von validen Adressen, der der ausführenden „virtuellen Maschine“ (Prozess  $p$  von  $P$ ) zugebilligt wird
    - „Prozessor“ der „virtuellen Machine“ übernimmt Abbildung:  $A_l \mapsto A_r$
    - Jede **logische Adresse**  $la \in A_l$  ist gültig:  $A_l$  enthält *konzeptionell* keine Lücken
  - ~~> **Arbeitsspeicher AS** des Prozesses  $p$ , auf **HS** abgebildet
- **Virtueller Adressraum  $A_v$**   $E_3$  (Maschinenprogrammebene)
  - Durch das **Betriebssystem** definierter (erweiterter) Wertvorrat an Adressen, der für einen logischen Adressraum zur Verfügung gestellt wird.
    - $A_v = A_l$ , aber  $A_v \mapsto A_r \vee HGS$  (Abbildung auch auf Hintergrundsspeicher, HGS)
    - Speicherzugriffe auf HGS werden durch das Betriebssystem **partiell interpretiert**
  - ~~> **Virtueller Arbeitsspeicher VAS**, des Prozesses  $p$ , durch das Betriebssystem **dynamisch** auf **HS oder HGS** abgebildet



## 11.6 Realer Adressraum

- Der **Adressraumbelegungsplan** (*memory map*) legt  $A_r$  fest, d.h., welche Hardwareeinheiten über welche Adressbereiche zugreifbar sind:

| Adressbereich       | Größe (KiB) | Verwendung                          |
|---------------------|-------------|-------------------------------------|
| 00000000–0009ffff   | 640         | RAM (System)                        |
| 000a0000–000bffff   | 128         | Video RAM                           |
| 000c0000–000c7fff   | 32          | BIOS Video RAM                      |
| 000c8000–000dffff   | 96          | keine                               |
| 000e0000–000effff   | 64          | BIOS Video RAM ( <i>shadow</i> )    |
| 000f0000–000fffff   | 64          | BIOS RAM ( <i>shadow</i> )          |
| 00100000–090fffff   | 147456      | RAM (Erweiterung)                   |
| 09100000–ffffdffff  | 4045696     | keine                               |
| ffffe0000–ffffeffff | 64          | SM-RAM ( <i>system management</i> ) |
| fffff0000–ffffffff  | 64          | BIOS ROM                            |

Toshiba Tecra 730CDT, 1996

- Wird durch den **Hersteller des Rechensystems** festgelegt (hier: Toshiba), nicht durch den Hersteller des Prozessors (hier: Intel).

- Der **Adressraumbelegungsplan** (*memory map*) legt A<sub>r</sub> fest, d.h., welche Hardwareeinheiten über welche Adressbereiche zugreifbar sind:

| Adressbereich       | Größe (KiB) | Verwendung                          |
|---------------------|-------------|-------------------------------------|
| 00000000–0009ffff   | 640         | RAM (System)                        |
| 000a0000–000bffff   | 128         | Video RAM                           |
| 000c0000–000c7fff   | 32          | BIOS Video RAM                      |
| 000c8000–000dffff   | 96          | keine                               |
| 000e0000–000effff   | 64          | BIOS Video RAM ( <i>shadow</i> )    |
| 000f0000–000fffff   | 64          | BIOS RAM ( <i>shadow</i> )          |
| 00100000–090fffff   | 147456      | RAM (Erweiterung)                   |
| 09100000–ffffdffff  | 4045696     | keine                               |
| ffffe0000–ffffeffff | 64          | SM-RAM ( <i>system management</i> ) |
| fffff0000–ffffffff  | 64          | BIOS ROM                            |

Toshiba Tecra 730CDT, 1996

- Der reale Adressraum ist **nicht zwingend linear** ↪ kann Lücken aufweisen.
- Zugriff auf eine Lücke in A<sub>r</sub> führt zu einem *bus error* Trap ↪ 6-15

- Der **Adressraumbelegungsplan** (*memory map*) legt  $A_r$  fest, d.h., welche Hardwareeinheiten über welche Adressbereiche zugreifbar sind:

| Adressbereich       | Größe (KiB) | Verwendung                          |
|---------------------|-------------|-------------------------------------|
| 00000000–0009ffff   | 640         | RAM (System)                        |
| 000a0000–000bffff   | 128         | Video RAM                           |
| 000c0000–000c7fff   | 32          | BIOS Video RAM                      |
| 000c8000–000dffff   | 96          | keine                               |
| 000e0000–000effff   | 64          | BIOS Video RAM ( <i>shadow</i> )    |
| 000f0000–000fffff   | 64          | BIOS RAM ( <i>shadow</i> )          |
| 00100000–090fffff   | 147456      | RAM (Erweiterung)                   |
| 09100000–ffffdffff  | 4045696     | keine                               |
| ffffe0000–ffffeffff | 64          | SM-RAM ( <i>system management</i> ) |
| fffff0000–ffffffff  | 64          | BIOS ROM                            |

Toshiba Tecra 730CDT, 1996

- Einige Adressbereiche unterliegen einem speziellen Verwendungszweck
- Ermöglicht den Zugriff auf Peripheriegeräte, ROM, Hardwarekonfiguration
  - Je nach Modus resultiert er in einem *access violation* Trap ↪ 6-15

- Der **Adressraumbelegungsplan** (*memory map*) legt  $A_r$  fest, d.h., welche Hardwareeinheiten über welche Adressbereiche zugreifbar sind:

| Adressbereich        | Größe (KiB) | Verwendung                          |
|----------------------|-------------|-------------------------------------|
| 00000000–0009ffff    | 640         | RAM (System)                        |
| 000a0000–000bffff    | 128         | Video RAM                           |
| 000c0000–000c7fff    | 32          | BIOS Video RAM                      |
| 000c8000–000dffff    | 96          | keine                               |
| 000e0000–000effff    | 64          | BIOS Video RAM ( <i>shadow</i> )    |
| 000f0000–000fffff    | 64          | BIOS RAM ( <i>shadow</i> )          |
| 00100000–090fffff    | 147456      | RAM (Erweiterung)                   |
| 09100000–ffffdfffff  | 4045696     | keine                               |
| ffffe0000–ffffefffff | 64          | SM-RAM ( <i>system management</i> ) |
| fffff0000–ffffffff   | 64          | BIOS ROM                            |

Toshiba Tecra 730CDT, 1996

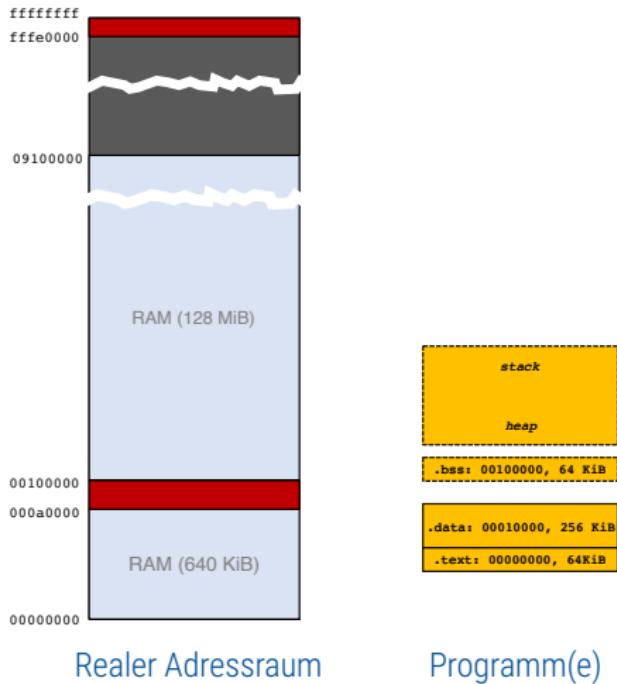
- Schließlich gibt es Adressbereiche, die der **allgemeinen Verwendung** dienen
- Hauptspeicher (*main memory*) für Maschinenprogramme und ihre Daten
  - Zugriff darauf *kann* einen Schutzfehler (*protection fault Trap*) liefern.



# Laden eines Programms

$E_2$ : Realer Adressraum

Laden eines Maschinenprogramms ( $E_3$ ) in den **realen Adressraum** ( $E_2$ )



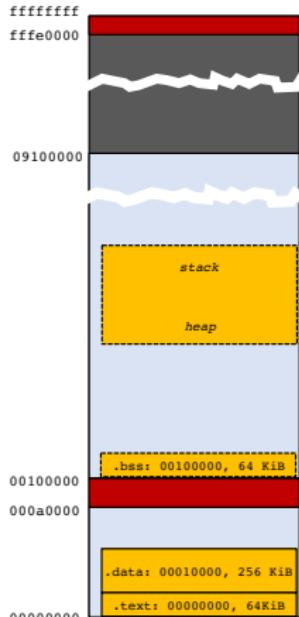
- Einprogrammbetrieb
  - Programm läuft exklusiv
  - Programm ist entsprechend des Adressraumbelegungsplans strukturiert
  - Symbole und Sektionen wurden bereits von [Compiler](#) und [Linker](#) überlappungsfrei auf  $A_r$  abgebildet



# Laden eines Programms

$E_2$ : Realer Adressraum

Laden eines Maschinenprogramms ( $E_3$ ) in den **realen Adressraum** ( $E_2$ )



- Einprogrammbetrieb
  - Programm läuft exklusiv
  - Programm ist entsprechend des Adressraumbelegungsplans strukturiert
  - Symbole und Sektionen wurden bereits von **Compiler und Linker** überlappungsfrei auf  $A_r$  abgebildet

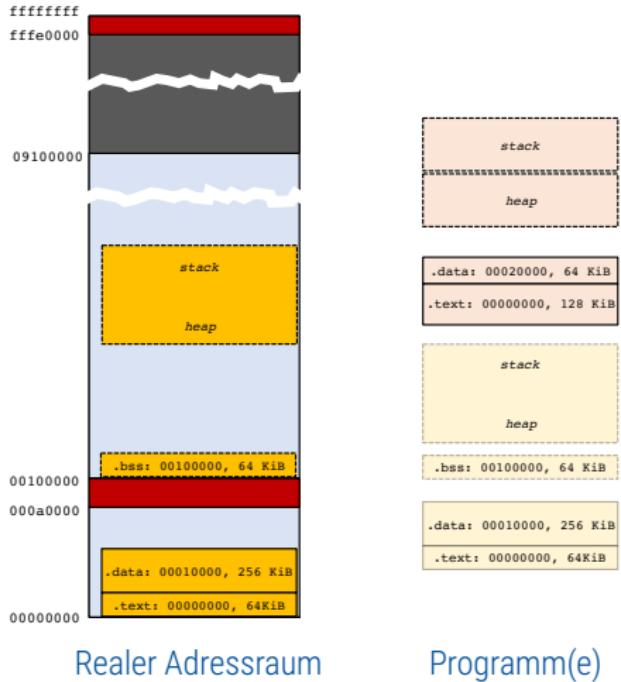
→ Einfaches Laden (Platzieren) im Speicher



# Laden eines Programms

$E_2$ : Realer Adressraum

Laden eines Maschinenprogramms ( $E_3$ ) in den **realen Adressraum** ( $E_2$ )



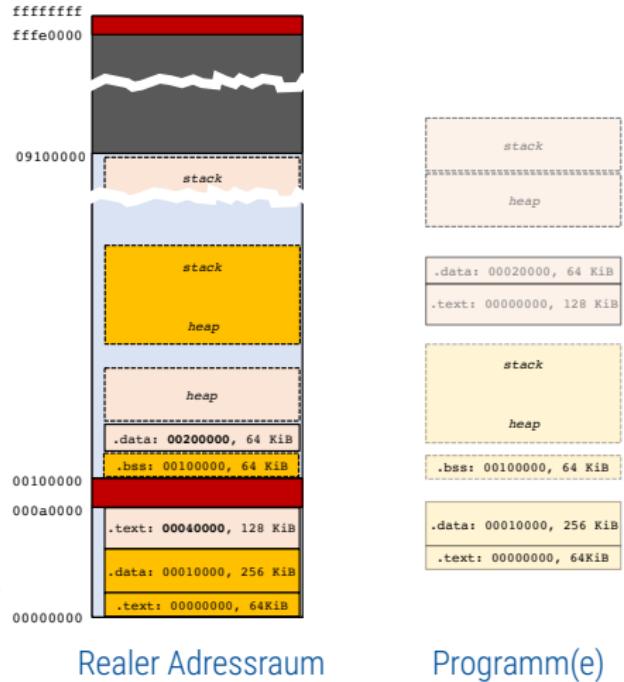
- Mehrprogrammbetrieb (dynamisch)
  - Neue Prozesse zur Laufzeit
  - Abbildung auf  $A_r$ , nicht mehr rein statisch möglich
  - Abbildung nur noch ein „Vorschlag“



# Laden eines Programms

$E_2$ : Realer Adressraum

Laden eines Maschinenprogramms ( $E_3$ ) in den **realen Adressraum** ( $E_2$ )



- Mehrprogrammbetrieb (dynamisch)
  - Neue Prozesse zur Laufzeit
  - Abbildung auf  $A_r$ , nicht mehr rein statisch möglich
  - Abbildung nur noch ein „Vorschlag“
- Erheblich mehr Arbeit für das BS
  - Speicherverwaltung und Platzierung
    - Freispeicherverwaltung
    - Platzierungsstrategie
    - Fragmentierung verhindern
  - **Relokation** der Symboladressen beim Laden erforderlich!



## 11.7 Relokation

Ladeadresse

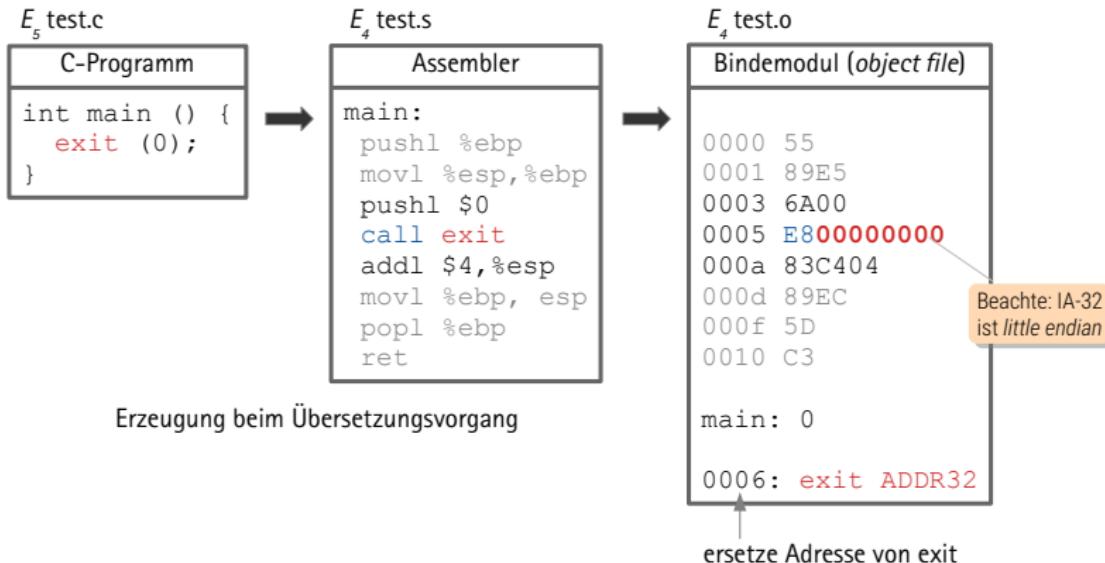
**Relokation** (Neuplatzierung) ist erforderlich, wenn die Sektionen eines Programms nicht an der beim Linken angenommenen Stelle in den Adressraum geladen werden können.

- **Variante A:** Beim Ladevorgang Code anpassen (*patchen*)
  - Anzupassen ist jeder Verweis (Zeiger) auf ein Symbol
  - Compiler und Linker exportieren Tabelle der Symbolreferenzen
  - Jede Symbolreferenz wird entsprechend der Ladeadresse angepasst
- **Variante B:** Positionsunabhängiger Code (*position-independent code, PIC*)
  - Zugriffe auf Code und Daten erfolgen immer Adressregister-indirekt
    - Bei Codezugriffen (soweit möglich) über den PC (PC-relative Sprünge)
    - Bei Datenzugriffen über ein festgelegtes Basisregister + Offset
    - Basisregister wird beim Laden auf Ladeadresse der Sektion gelegt
  - erfordert Unterstützung durch Compiler/Linker bei der Codegenerierung
- **Variante C:** HW-gestützte durch **Segmentierung**  $\mapsto$  Logischer Adressraum

# Laden eines Programs: Relokation

Variante A: Code anpassen

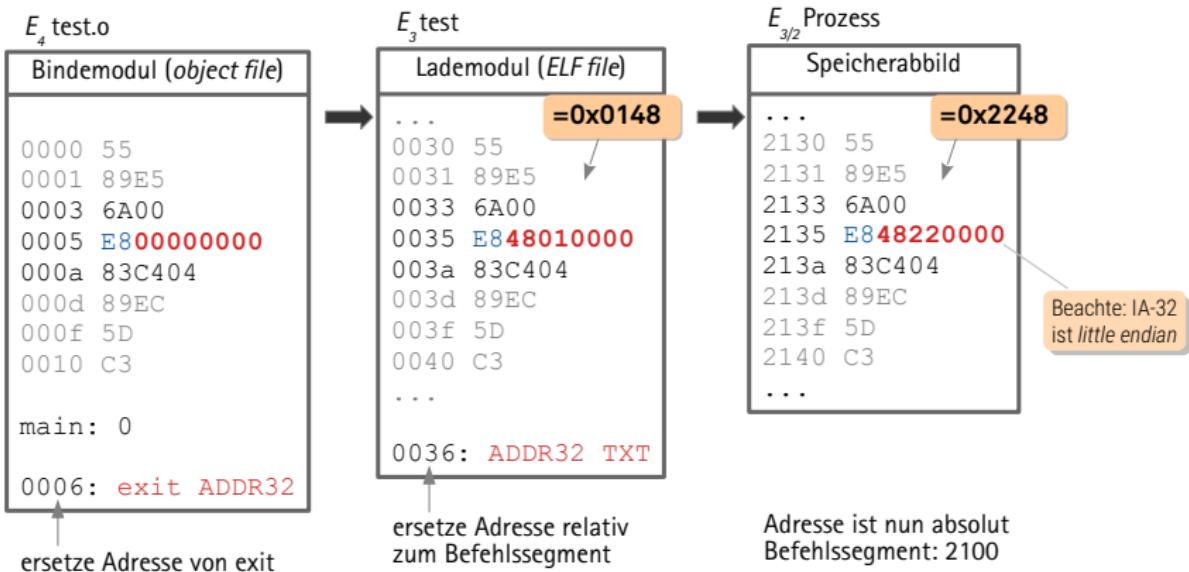
- Übersetzungsvorgang mit Relocationsinformationen (vgl. ↪ 11-12)
  - Für jedes Quellmodul (C-Datei) erzeugt der Compiler ein Bindemodul (*object file*)
    - Im Bindemodul wird für jede externe Referenz ein Relocationseintrag angelegt



# Laden eines Programs: Relokation

Variante A: Code anpassen

- Linken und Laden mit Relocationsinformationen (vgl. ↪ 11-12)
  - Linken fügt Code von `main()` und `exit()` zum `.text`-Segment zusammen
    - Wird zusammen mit den anderen Segmenten im ELF-Binary abgelegt
    - Relative Adresse (innerhalb von `.text`) von `exit()` ist nun bekannt
  - Beim Laden wird das `.text`-Segment im Speicher an einer Ladeadresse platziert
    - Absolute Adresse von `exit()` steht nun fest





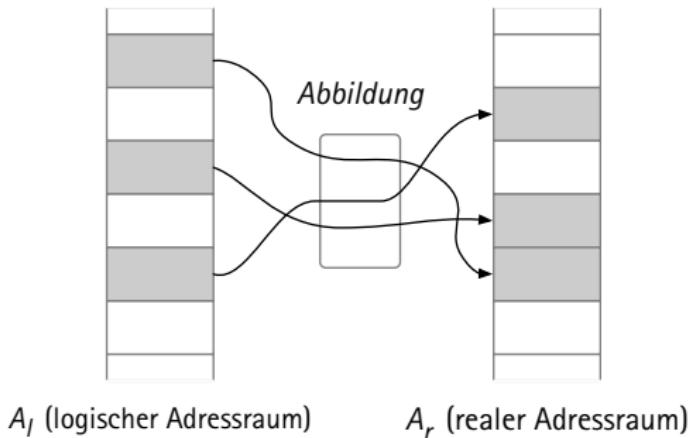
## 11.8 Logischer Adressraum

- **Realer Adressraum  $A_r$**   $E_2$  (Befehlssatzebene)
  - Durch Prozessor und Rechensystem definierter Wertvorrat von Adressen
    - $|A_r| = 2^n$  mit (üblicherweise)  $n \in \{16, 32, 48, 64\}$
    - Nicht jede reale Adresse  $ra \in A_r$  ist gültig:  $A_r$  kann Lücken aufweisen!
  - ~~ Hauptspeicher  $HS$ , adressierbar durch einen/mehrere Bereiche aus  $A_r$
- **Logischer Adressraum  $A_l$**   $E_{5,4,3}$  (Maschinenprogrammebene)
  - Durch ein **Programm  $P$**  definierter Wertvorrat von validen Adressen, der der ausführenden „virtuellen Maschine“ (Prozess  $p$  von  $P$ ) zugebilligt wird
    - „Prozessor“ der „virtuellen Machine“ übernimmt Abbildung:  $A_l \mapsto A_r$
    - Jede **logische Adresse  $la \in A_l$**  ist gültig:  $A_l$  enthält *konzeptionell* keine Lücken
  - ~~ **Arbeitsspeicher AS** des Prozesses  $p$ , auf  $HS$  abgebildet
- **Virtueller Adressraum  $A_v$**   $E_3$  (Maschinenprogrammebene)
  - Durch das **Betriebssystem** definierter (erweiterter) Wertvorrat an Adressen, der für einen logischen Adressraum zur Verfügung gestellt wird.
    - $A_v = A_l$ , aber  $A_v \mapsto A_r \vee HGS$  (Abbildung auch auf Hintergrundsspeicher, HGS)
    - Speicherzugriffe auf HGS werden durch das Betriebssystem **partiell interpretiert**
  - ~~ **Virtueller Arbeitsspeicher VAS**, des Prozesses  $p$ , durch das Betriebssystem **dynamisch** auf  **$HS$  oder  $HGS$**  abgebildet

- Die von einem Prozess  $p$  generierten **logischen Adressen** werden übersetzt
  - transparent durch eine prozessspezifische Abbildung  $p : A_l \mapsto A_r$
  - I. d. R. realisiert über die HW (z. B. **MMU**, *memory management unit*)
  - und dadurch auf Adressen im **realen Adressraum**  $A_r$  abgebildet.

Grundprinzip der Übersetzung  
von **logischen** in **physikalische**  
Adressen:  $p : A_l \mapsto A_r$

Abbildung erfolgt in Einheiten  
**gleicher Größe** (**Seiten**), oder...



$A_l$  (logischer Adressraum)

$A_r$  (realer Adressraum)

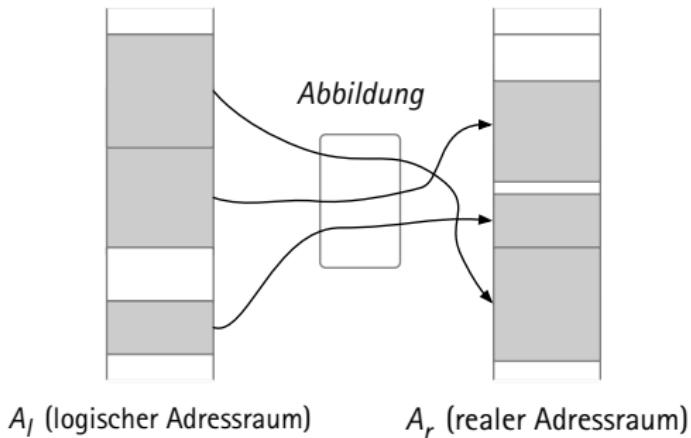
→ Grundlage des Speicherschutz zwischen Prozessen (**horizontale Isolation**),  
unter der Voraussetzung des Schutzes von  $p$  (**vertikale Isolation**) ← 3-20

- Die von einem Prozess  $p$  generierten **logischen Adressen** werden übersetzt
  - transparent durch eine prozessspezifische Abbildung  $p : A_l \mapsto A_r$
  - I. d. R. realisiert über die HW (z. B. **MMU**, *memory management unit*)
  - und dadurch auf Adressen im **realen Adressraum**  $A_r$  abgebildet.

Grundprinzip der Übersetzung  
von **logischen** in **physikalische**  
Adressen:  $p : A_l \mapsto A_r$

Abbildung erfolgt in Einheiten  
**gleicher Größe** (**Seiten**), oder...

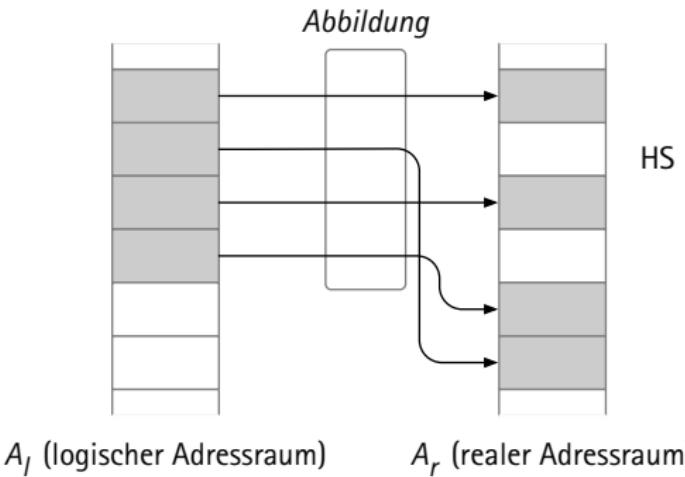
...oder in Einheiten **verschiede-**  
**ner Größe** (**Segmente**).



→ Grundlage des Speicherschutz zwischen Prozessen (**horizontale Isolation**),  
unter der Voraussetzung des Schutzes von  $p$  (**vertikale Isolation**) ↪ 3-20

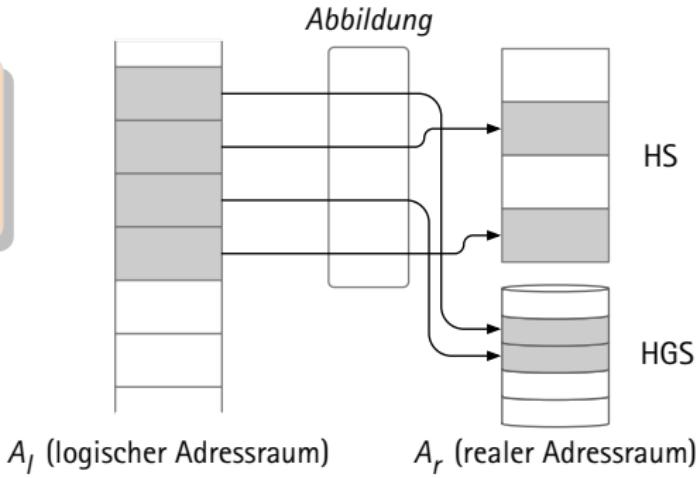
- Kleiner logischer  $\leftrightarrow$  größerer realer Adressraum (16/32-Bit Systeme)
  - Variante 8086:  $|A_l| = 2^{16} < |A_r| = 2^{20}$  (64 KiB  $\mapsto$  24 MiB)
  - Variante IA-32:  $|A_l| = 2^{32} < |A_r| = 2^{36}$  (4 GiB  $\mapsto$  64 GiB)
- Größerer logischer  $\leftrightarrow$  kleinerer realer Adressraum (32/64-Bit Systeme)
- Lineare logische  $\leftrightarrow$  unzusammenhängende reale Adressbereiche

Lineare (zusammenhängende) logische Adressbereiche werden abgebildet auf gestückelte reale Adressbereiche



- Lineare logische  $\mapsto$  **zweigeteilte** reale Adressbereiche
  - Zweiteilung in Hauptsspeicher (HS) und Hintergrundspeicher (HGS)
- Erfolgt ein Speicherzugriff auf den HGS, muss das BS aktiv werden
  - MMU signalisiert dieses durch einen Trap
  - Speicherzugriff wird vom BS partiell interpretiert  $\rightsquigarrow$  Speichervirtualisierung
    - Entsprechenden Bereich aus dem HGS in den HS „einlagern“
    - Dazu ggfs. erst anderen Bereich aus dem HS in den HGS „auslagern“

Lineare (zusammenhängende)  
logische Adressbereiche werden  
abgebildet auf **zweigeteilten** realen  
Speicher (z. B. RAM und Platte)



- Die Implementierung der Abbildung  $p : A_l \mapsto A_r$  erfolgt in der **MMU**
  - *Memory Management Unit* – früher ein Co-Prozessor, heute Teil der CPU
  - sobald aktiviert, wird **jede** Adresse auf dem Speicherbus transformiert
- Typische Implementierungsvarianten: Segmentbasiert und Seitenbasiert
  - **Segmentbasiert** – Strukturierung in Einheiten **verschiedener** Größe
    - **Umrechnung logischer Adressen** über die MMU auf **zusammenhängende** Bereiche aus  $A_r$  mit Basisregister:  $ra = la + seg_{base}$  für  $la < seg_{limit}$ , sonst *trap*.
    - ermöglicht einfache Implementierung von Relokation
  - **Seitenbasiert** – Strukturierung in Einheiten **gleicher** Größe
    - **Abbildung logischer Adressen** über die MMU auf **nichtzusammenhängende** Bereiche der Größe  $2^k$  mit Tabelle.  $la$  wird an Bit  $k$  aufgeteilt in Seitennummer  $la_{pn}$  und Versatz  $la_o$ :  $ra = page[la_{pn}] + la_o$  für  $page(la_{pn}) \neq \perp$ , sonst *trap*.
    - ermöglicht einfache Implementierung von virtuellem Speicher

Beides auch in Kombination, nicht nur für **horizontale Isolation**,  
sondern auch zur Implementierung von **virtuellem Speicher**

→ [12]



## 11.9 Zusammenfassung

- Vom C-Programm ( $E_5$ ) zum Speicher ( $E_{3/2}$ ) – mit Compiler, Linker und BS
    - statische und dynamische Allokation
    - Sektionen, Halden- und Stapelspeicher
    - Laden und Relokation
  - Freispeicherverwaltung
    - Mit Listen, Bitmaps, Halbierungsverfahren
    - Interner und externer Verschnitt
  - Adressraumlehre
    - realer Adressraum                       $\mapsto E_2$ , durch die Hardware definiert
    - logischer/virtueller Adressraum     $\mapsto E_3$ , linear, Abbildung durch das BS
    - Hardwareunterstützung (MMU)         $\mapsto$  durch (seitenorientierte) Segmentierung
- Für das Betriebssystem bleiben viele Herausforderungen



# Überblick: Teil D Speicher und Zugriffsschutz

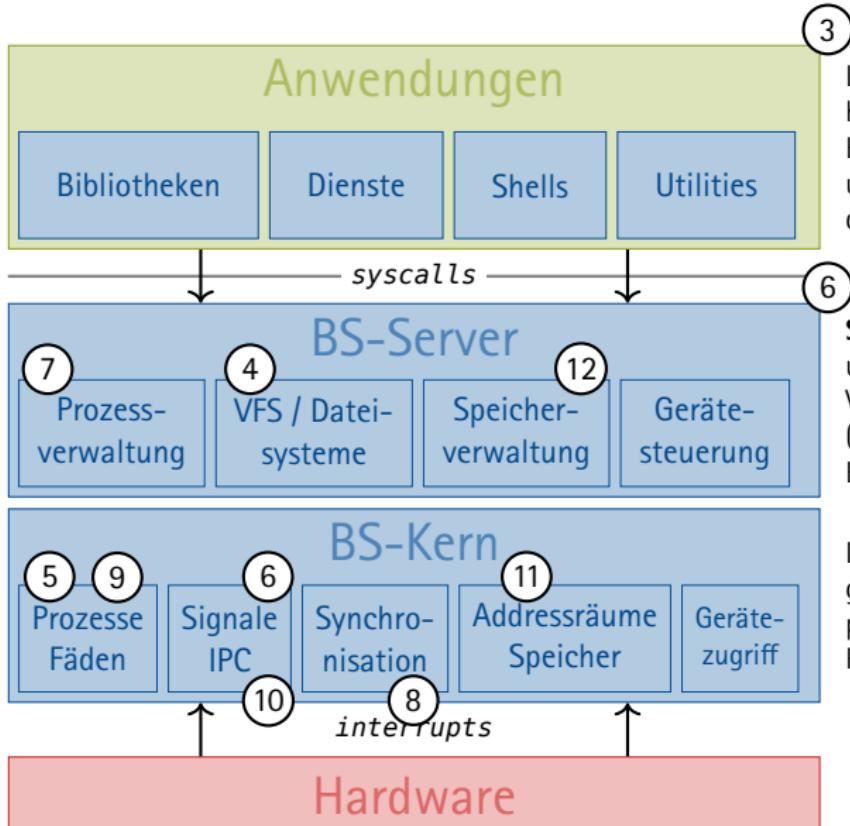
## 11 Speicherorganisation

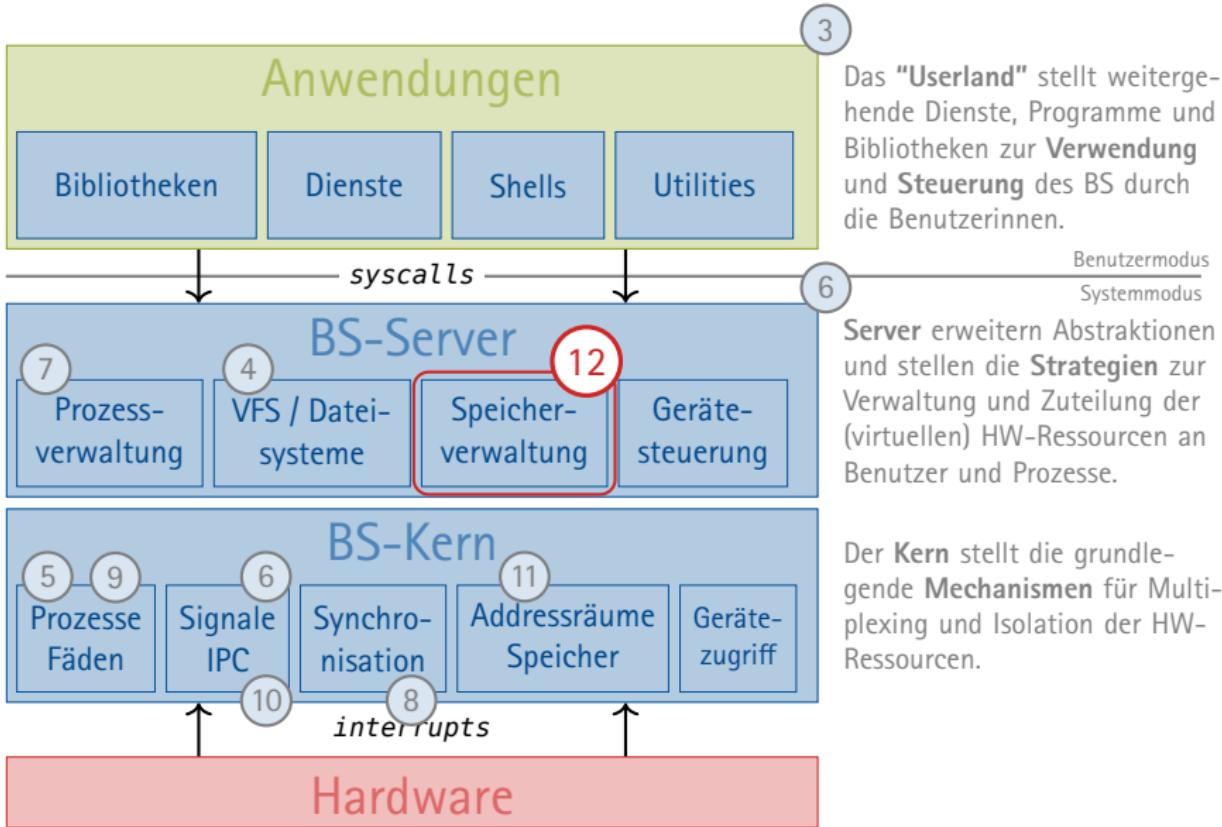
## **12 Speichervirtualisierung**

- 12.1 Einordnung
- 12.2 Segmentierung
- 12.3 Seitennummerierung
- 12.4 Gegenüberstellung
- 12.5 Virtueller Speicher
- 12.6 Entwicklungen
- 12.7 Zusammenfassung



## 12.1 Einordnung



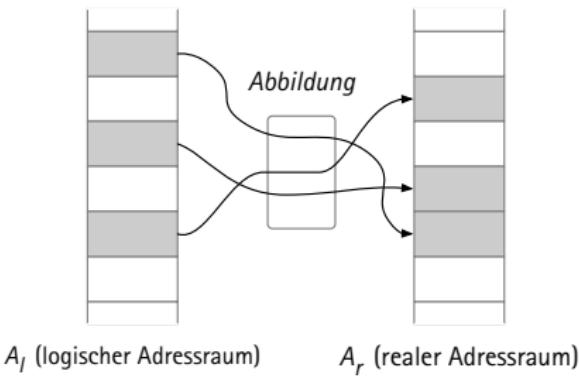


Worum geht es in diesem Kapitel?

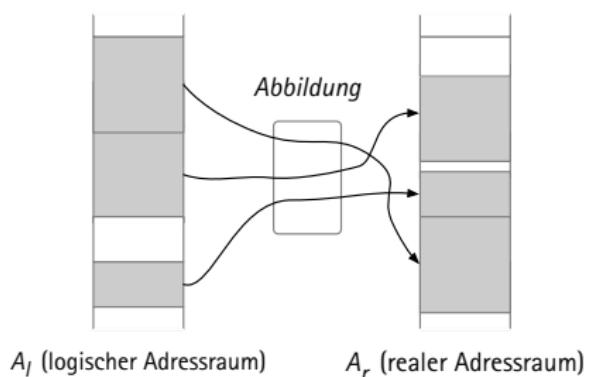
- **Verstehen** grundlegender Prinzipien und Konzepte der Speicherorganisation
    - Implementierung des logischen und virtuellen Adressraums
    - **Segmentierung** – Bereiche beliebiger Größe
    - **Seitennummerierung** – Bereiche fester Größe
  - **Erläutern** der jeweiligen Vorteile und Probleme
    - externe vs. interne Fragmentierung
    - Verschmelzung und Kompatifizierung
    - einstufige vs. mehrstufige Seitenabbildung
  - **Diskutieren** von Strategien der Speichervirtualisierung
    - Ladestrategie und Ersetzungsstrategie
    - Einzelanforderung vs. Vorausladen
    - Ersetzung mit LRU und *Second Chance*
- Multiplexing und **Virtualisierung** von Speicher durch das **Betriebssystem**



## Seitenbasiert



## Segmentbasiert





## 12.2 Segmentierung

- **Segmentierung** steht für eine Strukturierung des logischen Adressraums in Einheiten **verschiedener Größe**
- Einheiten werden als **Segment** bezeichnet
  - definieren eine lineare Folge von „Granulaten“ (Bytes, Seiten):
    - Bytes  $\mapsto$  zusammenhängend im **realen Adressraum**
    - Seiten  $\mapsto$  zusammenhängend in der **Seitentabelle** (zusätzliche Indirektion)
  - Idee:  $(E_5\text{-})\text{Modul} \mapsto (E_3\text{-})\text{Segment}$  [7, 23]
- Die vom Prozess generierte logische Adresse  $la$  bildet ein Paar  $(s, a)$ :
  - $s$  ist **Segmentname** (auch Segmentnummer)  $\rightsquigarrow$  1. Dimension
    - explizit gegeben (über den opcode)  $\mapsto$  zweikomponentige Adresse
    - implizit gegeben (durch Zugriffsart)  $\mapsto$  einkomponentige Adresse
    - Wertebereich für  $s = [0, 2^M - 1]$ ; bei IA-32:  $M = 13$
  - $a$  ist **Adresse**, auch **Versatz**, innerhalb des Segments  $S$   $\rightsquigarrow$  2. Dimension
    - Wertebereich für  $a = [0, 2^N - 1]$ ; bei IA-32:  $N = 32$
- Üblich sind numerische Werte für  $s$ , interpretiert als **Segmentindex**
  - Auswahl des **Segmentdeskriptors** aus einer **Segmenttabelle**

- **Segmentdeskriptor** von der **Hardware** vorgebener **Verbund** von **Attributen** zur partiellen Beschreibung von  $A_l \mapsto A_r$

- Basis Segmentanfangsadresse in  $A_r$ , Ausrichtung entspricht der Granularität (Bytes, Seiten)
- Limit Segmentlänge als Anzahl der Elemente (Granulate)
- Attribute Zugriffsrechte (lesen, schreiben, ausführen)  
Expansionsrichtung (aufwärts, abwärts)  
im Hauptspeicher? (*presence bit*)  
kürzlich verwendet? (*accessed bit*)

- Übliche Granularitäten

- 1 Maschinenwort „klassische“ Segmentierung
- 4/8 KiB, 4 MiB, 1 GiB seitenbasierte Segmentierung
- Kombinationen mehrstufige Segmentierung



## A<sub>1</sub>: Implementierung – Segmentierung

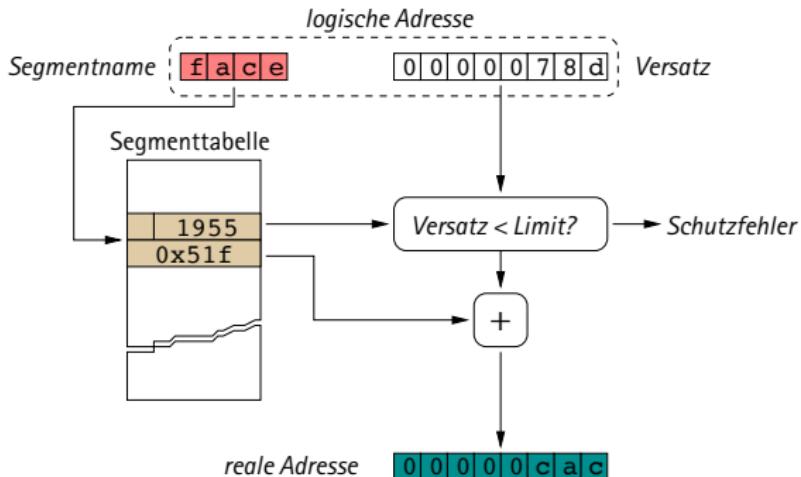
E<sub>3</sub> Logischer/Virtueller Adressraum

- Beispiel: Zugriff auf Adresse 0x78d im Segment 0xface

↪ **zweikomponentige Adresse**

Tabelleneintrag an 0xface enthält den **Segmentdeskriptor**.

Bei zweikomponentigen Adressen ist der verwendete **Segmentname** entweder im Opcode oder einem Segmentregister **explizit** gegeben.



- Adresse wird als **Versatz (offset)** zur Segmentbasisadresse interpretiert
  - Implementierung von **Relokation** ohne *Code-Patching* (→ 11–34, Variante C)

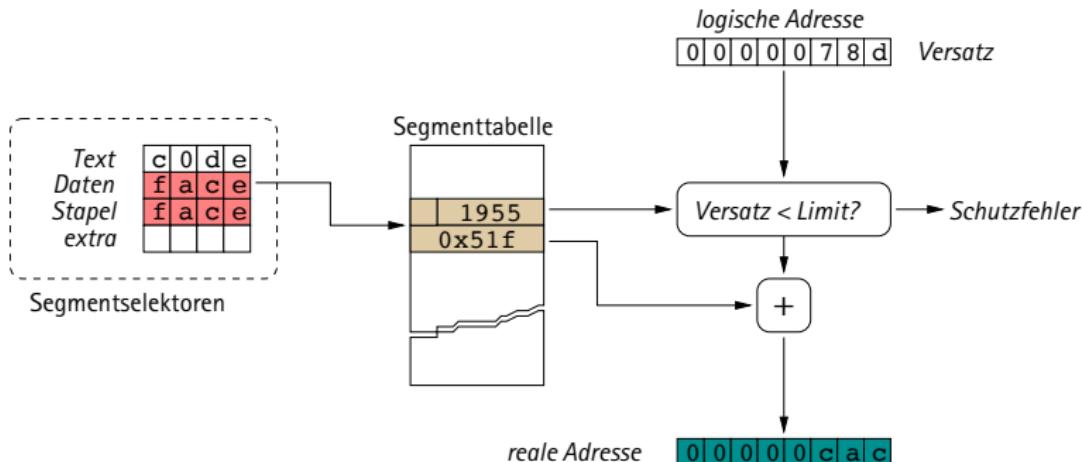


## A<sub>1</sub>: Implementierung – Segmentierung

E<sub>3</sub> Logischer/Virtueller Adressraum

- Beispiel: Datenzugriff auf Adresse 0x78d  $\rightsquigarrow$  Datensegment  
 $\mapsto$  **einkomponentige Adresse** (hier: Intel 8086)

Konnte so >64 KiB Speicher ansprechen



- Je nach Zugriffsart selektiert die MMU **implizit** das passende Segment
  - *instruction fetch*  $\mapsto$  Text-Segment (code)
  - *data fetch*  $\mapsto$  Daten-Segment (data)
  - *stack operation*  $\mapsto$  Stapel-Segment (stack)

Durch die **implizite** Segmentselektion bei einkomponentigen Adressen kann Segmentierung (und Relokation) transparent durchgeführt werden.



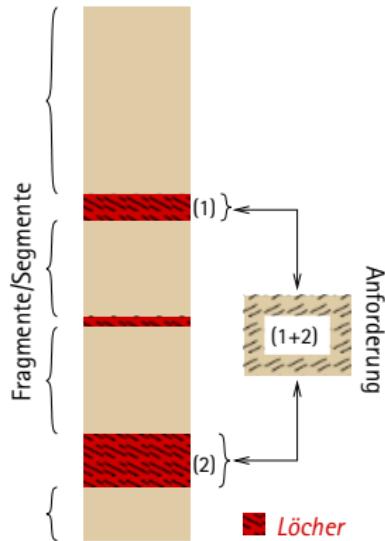
- Speicherverwaltung in Form von Segmenten orientiert sich am **Programm**
  - Problemnah: Compiler/Linker-Sektionen (Logische) Module      ↛ Segmente      ↛ Segmente
  - Ermöglicht transparente Relokation im realen Adressraum
- Die **Verwaltung** des realen Speichers  $A_r$  ist jedoch **aufwändig**
  - Zu platzierende Fragmente aus  $A_r$  sind eine Vielzahl von Bytes (Wörtern)
  - werden mit der MMU 1 : 1 auf lineare Adressbereiche in  $A_r$  abgebildet
  - führt zu Fragmentierung von  $A_r$

## → Externe Fragmentierung

→ 11-24

- Speicher ist fragmentiert, die angeforderte Größe zu groß für jedes Loch
- Summe der Größen aller freien Bereiche wäre ausreichend
- allerdings sind sie im Hauptspeicher nicht linear angeordnet.

- „Kommen und Gehen“ von Segmenten unterschiedlicher Größe
  - führt zu Freispeicher-Löchern in  $A_r$ ,
  - Komplexität der Suche nimmt zu
  - Anforderung kann nicht (linear) erfüllt werden
- **Verschnitt** ↪ ggf. nicht mehr zuteilbar
- **Verlust** ist durch das BS vermeidbar
  - mittels **Verschmelzung** verringrbar
  - mittels **Kompaktifizierung** auflösbar
  - jedoch mit hohem Aufwand verbunden



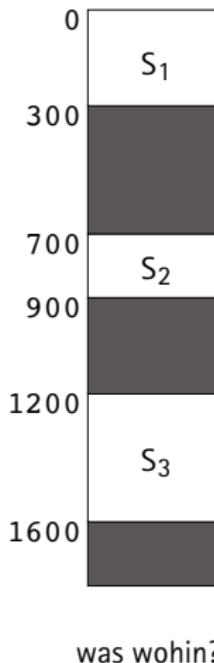
# ❖ Verringerung/Auflösung externer Fragmentierung

- **Verringerung:** Löcher mit benachbarten zusammen fassen ([Verschmelzung](#))
  - bei der Freigabe ↪ [11-21](#)
  - führt zu größeren Löchern (uns somit weniger Fragmentierung)
  - im Worst-Case jedoch keine Verbesserung
- **Auflösung:** Alle Löcher miteinander verschmelzen ([Kompaktifizierung](#))
  - Umlagern der Segmente im Hauptspeicher
    - durch direktes [Kopieren](#) im Hauptspeicher
    - oder Umweg über die Auslagerungsdatei (swapping)
  - Erfordert *im Prinzip* die [Relokation](#) der betroffenen Segmente ↪ [11-34](#)
    - Lage ändert sich hier jedoch nur in  $A_r$ , nicht in  $A_l$
    - ↪ Basisadresse im Segment anpassen reicht :-(
- **Umlagern ist teuer** ↪ Anzahl der Umlagerungsvorgänge gering halten
  - viele mögliche Varianten, komplexes Optimierungsproblem

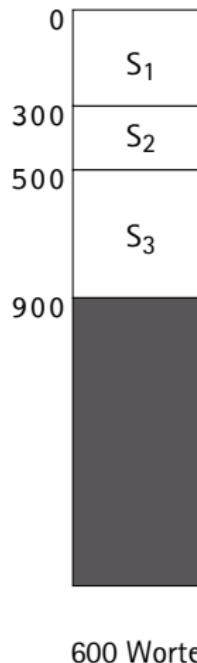


# Beispiel: Auflösung externer Fragmentierung

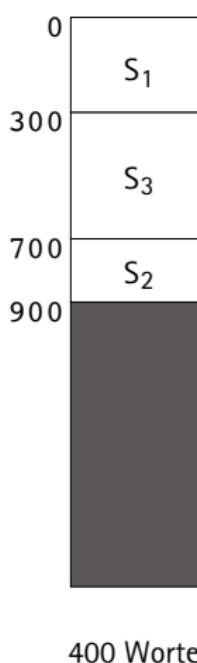
Kopieraufwand bei der Kompaktifizierung dreier Segmente – Varianten



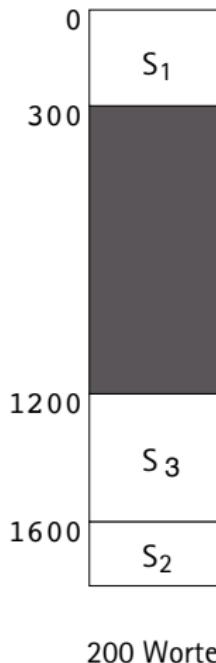
was wohin?



600 Worte



400 Worte

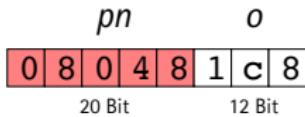


200 Worte



## 12.3 Seitennummerierung

- **Seitennummernierung** steht für eine Unterteilung des **Adressraums** in linear aufzählbare Einheiten **gleicher Größe**, bezeichnet als
  - **Kachel/Seitenrahmen** (*page frame*) im realen Adressraum
  - **Seite** (*page*) im logischen/virtuellen Adressraum
- Seiten werden durch eine **Seitentabelle** auf Seitenrahmen abgebildet
  - Einkomponentige lineare Adresse *la*, interpretiert als Bit-Tupel (*pn, o*)
    - *pn* ist **Seitennummer** (*page number*),  
indiziert in die Seitentabelle
    - *o* ist der **Versatz** (*Offset*) innerhalb der Seite
  - Typische Seitengröße: 4096 Byte ( $\sim$  12-Bit Offset)
- **Seitentabelle** (*page table*)
  - Array von **Seitendeskriptoren**
  - Seitennummer indiziert in diese Tabelle



- **Seitendeskriptor** von der **Hardware** vorgegebener **Verbund** von **Attributen** zur partiellen Beschreibung von  $A_l \mapsto A_r$ 
  - Kachel bzw. Seitenrahmen Seitenanfangsadresse in  $A_r$
  - Attribute Zugriffsrechte (lesen, schreiben, ausführen) im Hauptspeicher? (*presence bit*) kürzlich verwendet? (*accessed bit*)  
...
- Weitere Attribute werden vom Betriebssystem softwareseitig verwaltet
  - im **Schatten** der Seitentabelle (*shadow page table*)
  - für Verwaltungsaufgaben oder spezielle Funktionen

# ❖ A<sub>1</sub>: Implementierung – Seitenenorientiert, einstufig

## ■ **Einstufige Abbildung** (Seitentabelle enthält Deskriptoren)

```
real = table[pn].frame + o;
```

## ■ Eine Seitentabelle pro Prozess

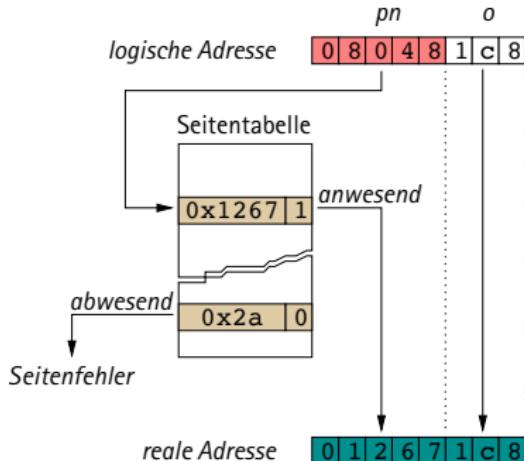
- Kann  $2^{20}$  Elemente beinhalten  
(bei 32-Bit Adressen und 12-Bit Seiten)
- Bei 32-Bit (4 Byte) pro Deskriptor:  
**4 MiB Speicherbedarf**
- **pro Prozess**, nur für die Tabelle!

## ■ Ansatz: Größe der Tabelle begrenzen

- Base/Limit Registerpaar in der MMU
- jedoch mit hohem Aufwand verbunden

→ Ergebnis wären verschieden große Seitentabellen

- Wachsen wäre möglich, Schrumpfen schwierig ↠ Fragmentierung :-(
- Externe Fragmentierung ↠ 12-12, aber diesmal im **logischen Adressraum A<sub>1</sub>**



# ❖ A1: Implementierung – Seitenenorientiert, zweistufig

## ■ Mehrstufige Abbildung (hier mit 2 Stufen)

## ■ Adressteil *pn* wird weiter unterteilt, z. B.

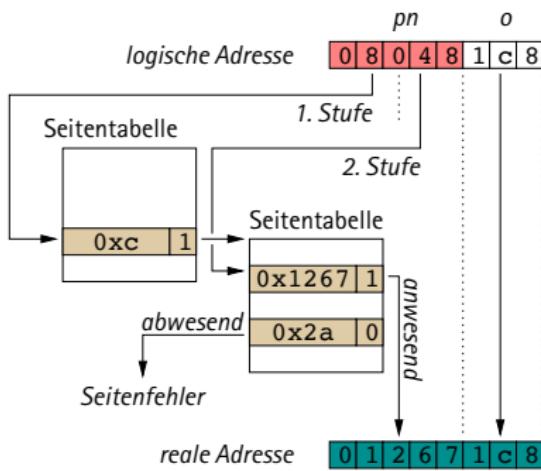
- 10-Bit *directory number*                    1. Stufe
- 10-Bit *page number*                        2. Stufe
- Bei 32-Bit (4 Byte) pro Deskriptor:  
4 KiB Speicherbedarf pro Tabelle
- pro Prozess (mindestens) 2 Tabellen

## ■ Ermöglicht gleich große Seitentabellen

- einfache Verwaltung
- aber zusätzliche Indirektionen :-(  
bei 64-Bit-Systemen bis zu 5-stufige Abbildung!)

## → Pro Stufe ein zusätzlicher Speicherzugriff durch den *table walk*

- Nur handhabbar durch *TLB (translation lookaside buffer)* mit hoher Trefferquote
- TLB cached Teile von  $p : A_l \mapsto A_r$  in HW  $\rightsquigarrow$  *table walk* nur bei *TLB-miss*

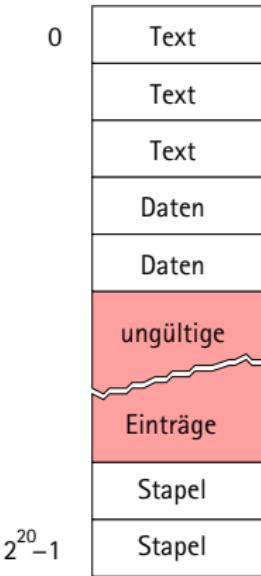


# A<sub>j</sub>: Implementierung – Seitennummeriert

Vergleich

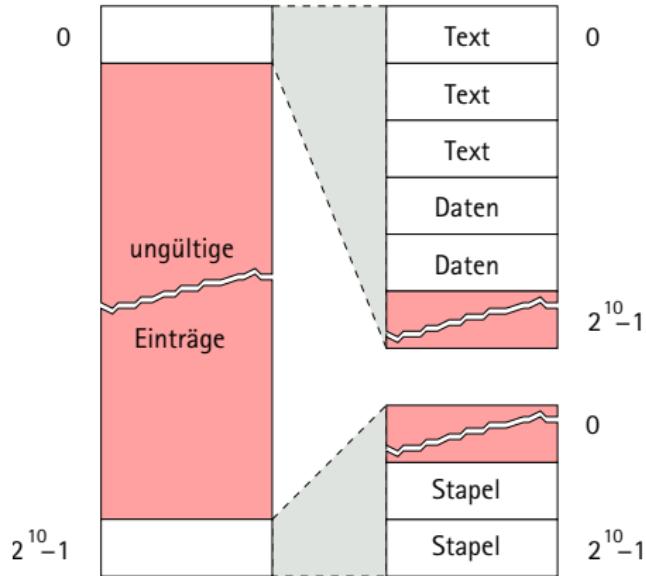
**Annahme:** UNIX-Prozess mit 12/8/8 KiB Text/Daten/Stapel

## ■ einstufige Tabelle



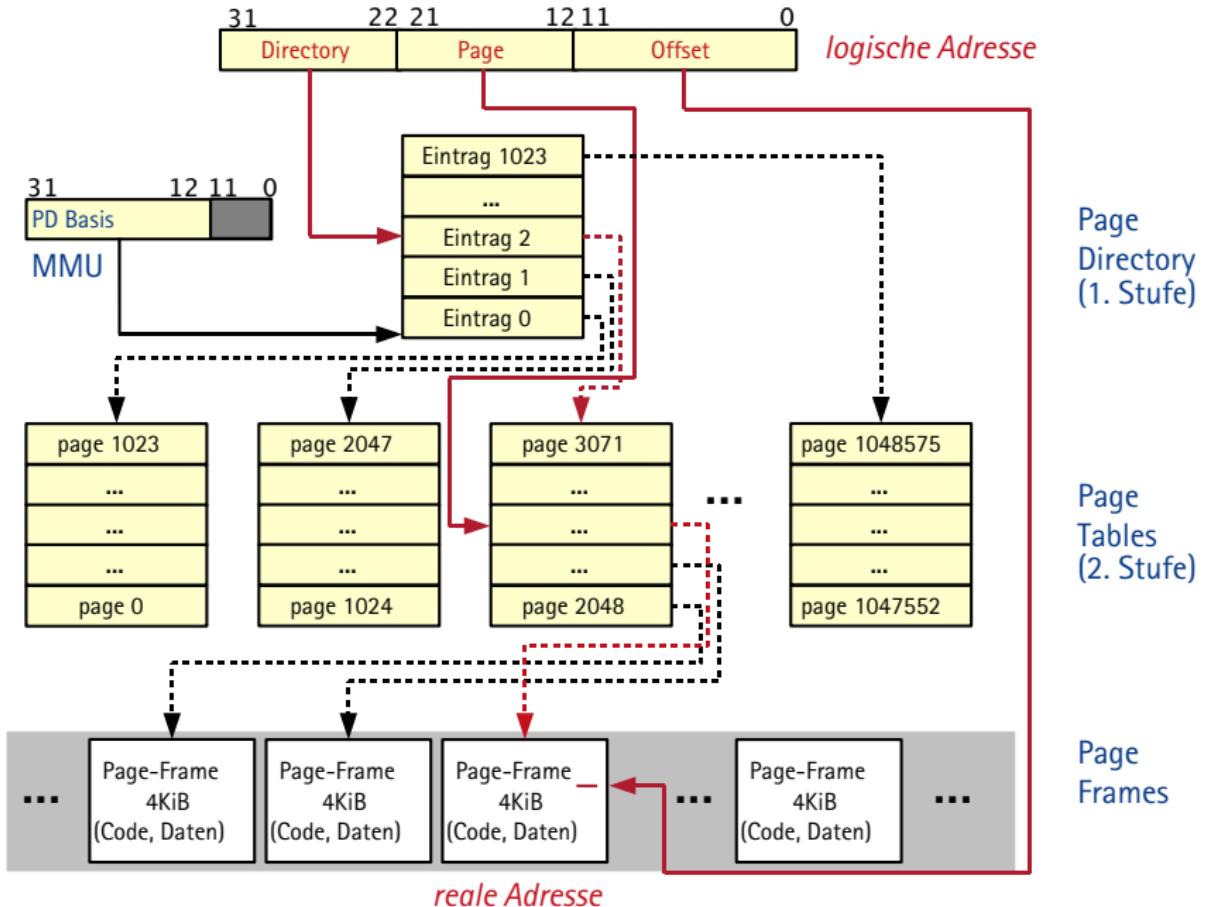
→  $2^{20} - 7$  ungültige Einträge  
■ ≈ 3,9 MiB auf 1 Tabelle

## ■ zweistufige Tabelle



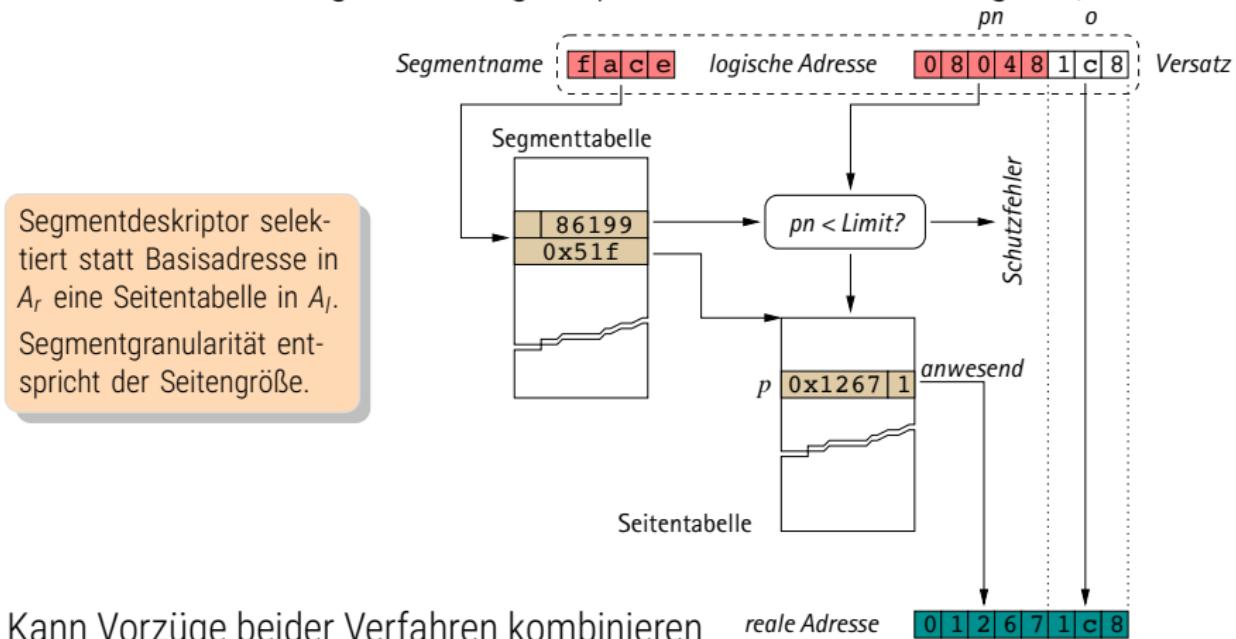
→ 3063 ungültige Einträge  
■ ≈ 11,9 KiB auf 3 Tabellen

# Beispiel: Zweistufige Abbildung auf IA-32



# A<sub>l</sub>: Variante – Segmentierte Seitennummerierung

- **Kombination** von Segmentierung in A<sub>l</sub> mit Seitennummerierung in A<sub>r</sub>



- Kann Vorzüge beider Verfahren kombinieren
  - Nur auf wenigen Architekturen verfügbar (insbesondere IA-32)
  - BS kann (logische) Segmentierung jedoch auch in Software realisieren



# Seitenorientierte Speicherverwaltung: Bewertung

- Speicherverwaltung in Form von Seiten orientiert sich an der **HW**
    - Verwaltung auf Ebene gleichgroßer Speicherstücke
    - Segment muss nicht mehr auf lineare Speicherbereiche  $A_r$  abgebildet werden
    - Vermeidet externe Fragmentierung (auf Kosten **interner Fragmentierung**)
  - Implementierung von  $p : A_l \mapsto A_r$  wird aufwändiger
    - Größere Deskriptortabellen, mehrstufige Deskriptoren
    - nur durch Hardwareunterstützung (TLB) „bezahlbar“
- Interne Fragmentierung
- 11-24
- Größe eines Segments ist nicht Vielfaches der Seitengröße
  - Pro Segment bleibt im Mittel eine halbe Speicherseite ungenutzt



# Seitennummerierung: Interne Fragmentierung

## ■ Seitennummerierter Adressraum

- Programmsegmente sind Vielfaches von Bytes
- Adressraum ist Vielfaches von Seiten
- jeweils letzte Seite eines Segments ist ggfs. nicht vollständig nutzbar

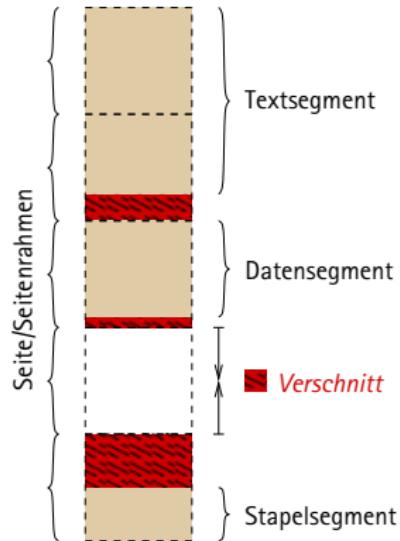
## ■ seitenlokaler Verschnitt

- vom Prozess logisch nicht beansprucht
- physisch dennoch zugewiesen und addressierbar

→ Verschwendungen, durch BS nicht vermeidbar!

- im Mittel eine halbe Seite/Segment

→ Dennoch in der Praxis ein **guter Kompromiss**





## 12.4 Gegenüberstellung



|                                    | <b>seitenorientiert</b>                                                                      | <b>segmentorientiert</b>                                                                           |
|------------------------------------|----------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| Größe/Granularität                 | linear adressierbarer Block<br>fester Größe, von der Hardware vorgegeben                     | linear adressierbarer Block beliebiger Größe, vom Programmierer festgelegt.                        |
| Präsenz im realen Adressraum $A_r$ | Menge von Seitenrahmen,<br>beliebig in $A_r$ verteilt                                        | Linear zusammenhängender Speicherblock in $A_r$                                                    |
| Fragmentierung                     | führt zu interner Fragmentierung                                                             | führt zu externer Fragmentierung                                                                   |
| Adressbildung                      | Zeiger wird intern aufgeteilt in Seitennummer und Offset.                                    | Zeiger repräsentiert Offset. Basis gegeben durch impliziten/expliziten Segmentselektor.            |
| Deskriptor-ermittlung              | erfordert (ggfs. mehrstufigen) Tabellenzugriff                                               | über den Segmentselektor                                                                           |
| Deskriptoren                       | Bilden Seitennummer aus $A_l$ , auf Seitenrahmennummer in $A_r$ ab: <i>Basis + Attribute</i> | Bilden Segmentnamen aus $A_l$ , auf Speicherblock in $A_r$ ab:<br><i>Basis + Limit + Attribute</i> |



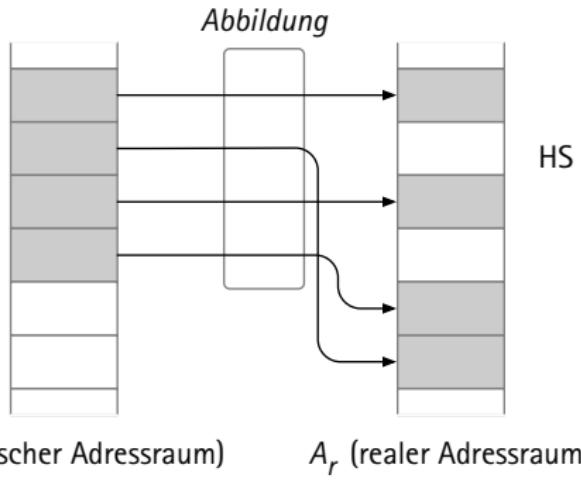
## 12.5 Virtueller Speicher

- **Realer Adressraum  $A_r$**   $E_2$  (Befehlssatzebene)
  - Durch Prozessor und Rechensystem definierter Wertvorrat von Adressen
    - $|A_r| = 2^n$  mit (üblicherweise)  $n \in \{16, 32, 48, 64\}$
    - Nicht jede reale Adresse  $ra \in A_r$  ist gültig:  $A_r$  kann Lücken aufweisen!
  - ~~ Hauptspeicher **HS**, adressierbar durch einen/mehrere Bereiche aus  $A_r$
- **Logischer Adressraum  $A_l$**   $E_{5,4,3}$  (Maschinenprogrammebene)
  - Durch ein Programm  $P$  definierter Wertvorrat von validen Adressen, der der ausführenden „virtuellen Maschine“ (Prozess  $p$  von  $P$ ) zugebilligt wird
    - „Prozessor“ der „virtuellen Machine“ übernimmt Abbildung:  $A_l \mapsto A_r$
    - Jede logische Adresse  $la \in A_l$  ist gültig:  $A_l$  enthält *konzeptionell* keine Lücken
  - ~~ Arbeitsspeicher **AS** des Prozesses  $p$ , auf **HS** abgebildet
- **Virtueller Adressraum  $A_v$**   $E_3$  (Maschinenprogrammebene)
  - Durch das **Betriebssystem** definierter (erweiterter) Wertvorrat an Adressen, der für einen logischen Adressraum zur Verfügung gestellt wird.
    - $A_v = A_l$ , aber  $A_v \mapsto A_r \vee HGS$  (Abbildung auch auf Hintergrundsspeicher, HGS)
    - Speicherzugriffe auf HGS werden durch das Betriebssystem **partiell interpretiert**
  - ~~ **Virtueller Arbeitsspeicher VAS**, des Prozesses  $p$ , durch das Betriebssystem **dynamisch** auf **HS oder HGS** abgebildet

- Lineare logische  $\mapsto$  unzusammenhängende reale Adressbereiche
  - Vermeidung externer Fragmentierung

↪ 11-24

Lineare (zusammenhängende) logische Adressbereiche werden abgebildet auf gestückelte reale Adressbereiche

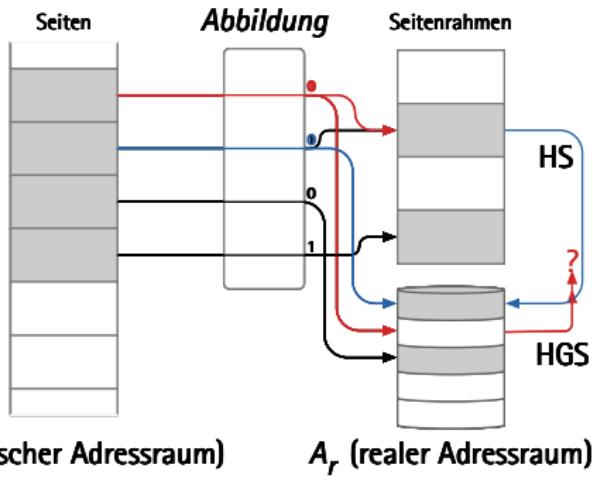


- Lineare logische  $\mapsto$  **zweigeteilte** reale Adressbereiche
  - Erweiterung des für Prozesse zur Verfügung stehenden Speichers
- Zweiteilung in Hauptsspeicher (HS) und Hintergrundspeicher (HGS)
  - Ein Speicherzugriff auf den HGS führt zu einem **Seitenfehler**
  - MMU sendet einen *Trap*, gesteuert durch **presence**-Bit im Deskriptor
- $\rightsquigarrow$  Speicherzugriff wird vom BS partiell interpretiert
- $\rightsquigarrow$  **Speichervirtualisierung**

Lineare (zusammenhängende) logische Adressbereiche werden abgebildet auf **zweigeteilten** realen Speicher (z. B. RAM und Platte)

Laden von Seiten aus dem HGS

Ersetzen von Seitrahmen im HS



A<sub>l</sub> (logischer Adressraum)

A<sub>r</sub> (realer Adressraum)

**Speichervirtualisierung** vergrößert den zur Verfügung stehenden Arbeitsspeicher in  $A$ , durch das **transparente Umlagern** (*swapping*) von Segmenten oder **Seiten** zwischen Haupt- und Hintergrundspeicher in  $A_r$ .

~~~ Möglich aufgrund es **Lokalitätsprinzips!**

■ Zu lösen sind dabei zwei zentrale Fragen:

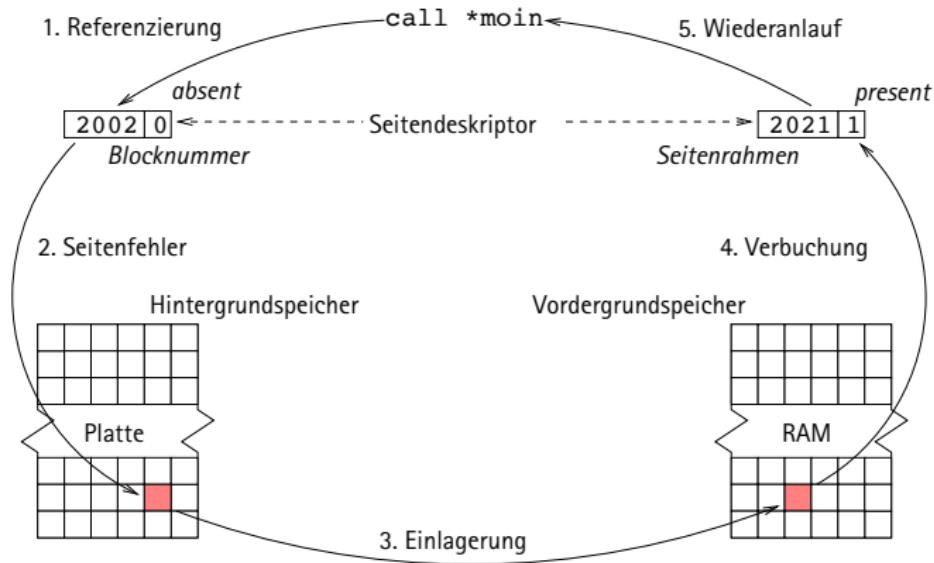
■ **Ladestrategie** (*fetch policy*)

- ~~~ Wann wird ein Seite in den Hauptspeicher geladen?
- „Erst bei Bedarf“ Einzelanforderung (**demand paging**)
 - „Vorausschauend“ Mehrfachanforderung (**anticipatory paging**)

■ **Ersetzungsstrategie** (*replacement policy*)

- ~~~ Welche Seite im Hauptspeicher ist (temporär) verzichtbar?
- Algorithmus FIFO, **LRU**, **second chance**, ...
 - lokale Suche innerhalb der Seitenrahmen des anfordernden Prozesses
 - **globale** Suche über die Seitenrahmen aller Prozesse

- Seitenfehler ⚡ bei Ausführen der Operation `call *moin`
- **Achtung:** Eine Operation kann **mehrere Seitenfehler** auslösen ↪ Folgefehler



- Seitenfehler ⚡ bei Ausführen der Operation `call *moin`
- **Achtung:** Eine Operation kann **mehrere Seitenfehler** auslösen ↪ **Folgefehler**
- **Fehler und Folgefehler ⚡ vermeiden**
 - ↪ erfordert Analyse / Wissen über künftige Zugriffe
 - aus Heuristik („raten“)
 - im *pager* auch Folgeseiten laden (insbesondere bei Arrays wirksam)
 - aus Aufzeichnung (*trace*) früherer Programmabläufe
 - insbesondere der Programmstart lässt sich so beschleunigen
 - durch partielle Interpretation der Operation im *pager*
 - gescheiterten Befehl dekodieren
 - Zieladressen aller Operanden/Stapel prüfen und einlagern

Linux

Windows

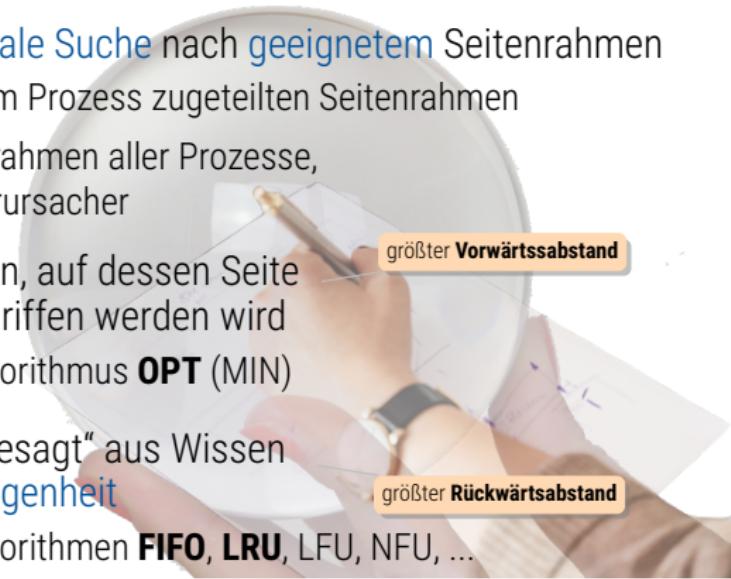
Betriebssystem emuliert
Dekodierung und Speicher-
operationen des Prozessors

- Seitenfehler bewirken eine **Prozessverzögerung** beim Speicherzugriff
 - Zugriffszeiten: $\text{HS} \approx 50 \text{ ns}$ $\text{HGS} \approx 10 \text{ ms}$ Faktor 200.000!
 - Erhöhen die **mittlere Zugriffszeit** (*mean access time, mat*) auf den Speicher in Abhängigkeit von
 - *p*, die **Seitenfehlerwahrscheinlichkeit** eines Speicherzugriffs
 - *rat*, der Zugriffszeit auf den HS (*RAM access time*)
 - *dat*, der Zugriffszeit auf den HGS (*disk access time*)

$$mat \approx (1 - p) \cdot rat + p \cdot dat, 0 \leq p \leq 1$$

- Sei $rat = 50 \text{ ns}$ und $dat = 10 \text{ ms}$, dann
 - $mat \approx 60 \text{ ns}$ mit $p = 1/10^6$ (übliche Annahme) Faktor 1,2
 - $mat \approx 50 \text{ ns}$ mit $p = 1/10^6$ und $dat = 0,1 \text{ ms}$ (SSD) Faktor 1
- Kosten **extrem** abhängig vom tatsächlichen Prozessverhalten!
 - $mat \approx 10.000 \text{ ns}$ mit $p = 1/10^2$ Faktor 2.000!

- Die Ladestrategie fordert, dass ein Seitenrahmen zur Aufnahme der einzulagernden Seite **verfügbar** sein muss.
 - Falls für den Prozess noch ein Rahmen frei ist, ist dies einfach.
 - Ansonsten ist ein Seitenrahmen freizumachen \mapsto **Ersetzungsstrategie**
- Erfordert eine lokale oder globale Suche nach geeignetem Seitenrahmen
 - lokal: Suche nur in den dem Prozess zugeteilten Seitenrahmen
 - global: Suche in den Seitenrahmen aller Prozesse, unabhängig vom Verursacher
- Optimal wäre der Seitenrahmen, auf dessen Seite am längsten nicht mehr zugegriffen werden wird
 - \hookrightarrow „**Glaskugel**“ Algorithmus **OPT** (MIN)
 - \hookrightarrow „**Buchführung**“ Algorithmen **FIFO, LRU, LFU, NFU, ...**
- Real wird die Zukunft „vorhergesagt“ aus Wissen über die Zugriffe in der **Vergangenheit**



FIFO Wähle die **älteste** Seite (*first in, first out*)

- betrachtet Nutzung nur eingeschränkt, Anomalien möglich
- wird in der Praxis kombiniert mit **second chance** \rightsquigarrow „LRU-artiger“

LRU [LFU] Wähle die **am längsten nicht mehr [am seltensten]** referenzierte Seite (*least recently [frequently] used*)

- beachtet das Lokalitätsprinzip
- erfordert Buchführung über die Zugriffe pro Seitendeskriptor (Zeitstempel/Zähler)
- aufwändig \rightsquigarrow in der Praxis nur (über Referenzbit) approximiert

... eine Vielzahl von Varianten unter eigenen Namen

- letztlich unterschiedlich gute Approximationen von LRU/LFU

- Jedem Seitendescriptor ist ein **Schieberegister** (*aging register*) zugeordnet
 - üblicherweise vom BS in Software verwaltet (in der *shadow page table*)
- Zeitgeber unterrichtet Prozesse periodisch
 - Bei jedem Ablauf des Zeitintervalls wird das **Referenzbit** *jeder* eingelagerten Seite in ihr Schieberegister übernommen.
 - Anschließend wird es zurückgesetzt.

Statusbit im Seitendescriptor;
wird von der MMU bei jedem Zugriff gesetzt.

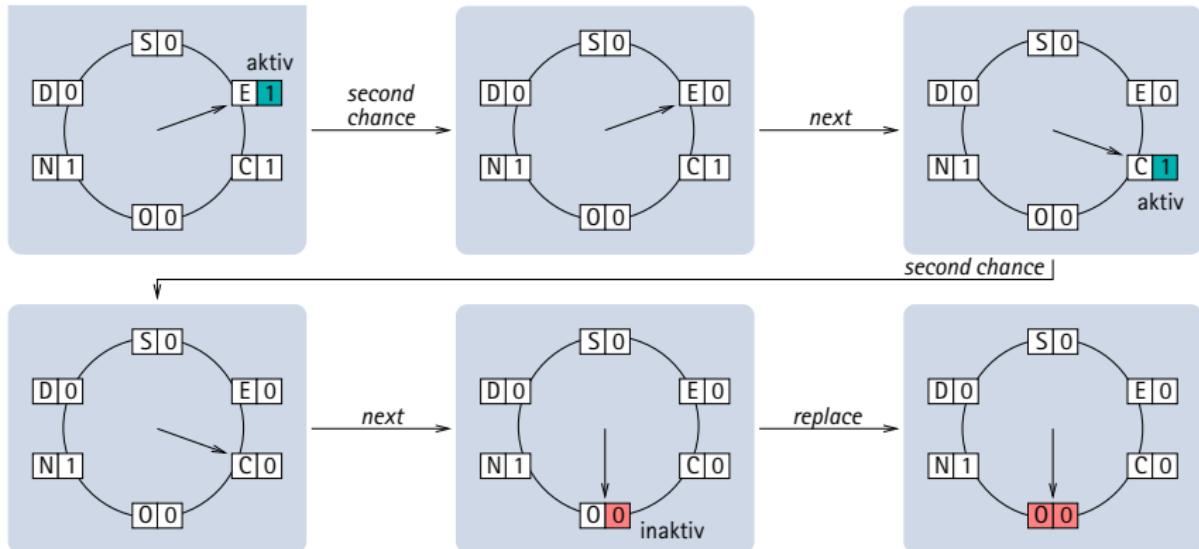
Aufwändig!

| <i>tick</i> | Ref. | <i>aging register</i> |
|--------------|------|-----------------------|
| | | 0 0 0 0 0 0 0 |
| <i>n</i> | 1 | 1 0 0 0 0 0 0 |
| <i>n + 1</i> | 1 | 1 1 0 0 0 0 0 |
| <i>n + 2</i> | 0 | 0 1 1 0 0 0 0 |
| <i>n + 3</i> | 1 | 1 0 1 1 0 0 0 |
| ⋮ | ⋮ | ⋮ |

- Registerinhalt (*age*) als Ganzzahl interpretiert liefert ein Maß für die Aktivität einer Seite
- mit abnehmendem Betrag, d. h. einer sinkenden Prozessaktivität, steigt die Ersetzungspriorität
- bei Seitenfehler/-ersetzung wird die **älteste** Seite gewählt

→ A_V: Ersetzungsstrategie – FIFO mit second chance

clock



- **Ansatz:** Suche umlaufend (FIFO/RR) Seitenrahmen mit inaktiver Seite
 - E aktiv, Referenzbit zurücksetzen, Seite im Hauptspeicher behalten
 - C aktiv, Referenzbit zurücksetzen, Seite im Hauptspeicher behalten
 - O inaktiv → Seite auslagern

⌘ A_v: Ersetzungsstrategie – FIFO mit second chance

- Arbeitet im Prinzip nach FIFO, berücksichtigt zusätzlich das Referenzbit
 - Berücksichtigt tatsächliche Nutzung erheblich besser als FIFO
 - „LRU-artig“, aber erheblich geringerer Aufwand als LRU
- Zur Seitenerersetzung wird der Rahmen der **ältesten eingelagerten** (FIFO) aber **unreferenzierten** (LRU) Seite gewählt
 - Seiten werden (umlaufend) in FIFO-Folge betrachtet. Ist das Referenzbit
 - 1** ↵ Seite erhält „zweite Chance“, Referenzbit zurücksetzen
 - 0** ↵ Rahmen gefunden, Seite wird ersetzt
 - Spätestens beim zweiten Umlauf wird ein passender Rahmen gefunden
- ~~ Entartet im schlimmsten Fall zu FIFO

Wenn im Betrachtungszeitraum
die Referenzbits **aller**
Seiten gesetzt (1) waren



12.6 Entwicklungen

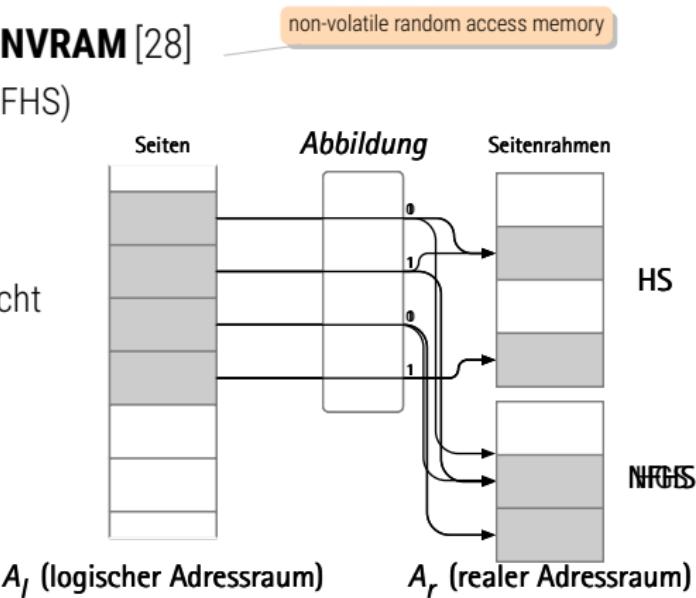
- Lade- und Ersetzungsstrategie des Betriebssystems waren lange **dominant** für die Leistungsfähigkeit eines Rechensystems – aber ihre Bedeutung sinkt!

- größere HS ↪ sinkende Seitenfehlerwahrscheinlichkeit
- schnellerer HGS (Flash) ↪ sinkende Seitenfehlerkosten (Faktor 100)

- Möglicher „*game changer*“: **NVRAM** [28]

- nichtflüchtiger Hauptspeicher (NFHS)
 - byteweise addressierbar
 - ca. 3–8 x langsamer als RAM
 - erheblich günstiger als Flash
- „ausgelagerte“ Seiten müssen nicht mehr zwingend geladen werden
- Ersetzung (nur) noch zur Zugriffszeitoptimierung

↪ HS wird Cache für HGS





12.7 Zusammenfassung



- **Organisation** segmentiert vs. seitennummeriert
 - externe Fragmentierung durch verschiedene große Segmente
 - interne Fragmentierung durch ungenutzten Verschnitt im Seitenrahmen
- **Speichervirtualisierung** erfolgt überwiegend **seitenbasiert**
 - Seite \mapsto Seitenrahmen/Disk-Block
 - Erfordert **Ladestrategie** und **Ersetzungsstrategie**
- **Seitenfehler** sind **richtig** teuer
 - Einzelanforderung vs. Vorausladen
 - abgemildert durch „gute“ Seitenaustrauschalgorithmen
- Die **Implementierung** von Virtuellem Speicher im Betriebssystem hat **größten Einfluss** auf Anwendungen mit höheren Speicherbedarf!



Technische
Universität
Braunschweig



Institute of Operating Systems
and Computer Networks
Reliable System Software



Betriebssysteme (BS)

Teil E Anhang

Christian Dietrich

Wintersemester 2024



- [1] Remzi H. Arpaci-Dusseau und Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces* (version 1.10). <https://pages.cs.wisc.edu/~remzi/OSTEP/>. 2023.
- [2] ATmega328PB 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash. Atmel Corporation. Okt. 2015.
- [3] Andrew Baumann, Jonathan Appavoo, Orran Krieger u. a. „A Fork() in the Road“. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: ACM, 2019, S. 14–22. ISBN: 978-1-4503-6727-1. doi: [10.1145/3317550.3321435](https://doi.org/10.1145/3317550.3321435).
- [4] Andrew D. Birrell und Bruce Jay Nelson. „Implementing Remote Procedure Calls“. In: *ACM Transactions on Computer Systems* 2.1 (Feb. 1984), S. 39–59. ISSN: 0734-2071. doi: [10.1145/2080.357392](https://doi.org/10.1145/2080.357392).
- [5] Edward G. Coffman, Jr., M. J. Elphick und Arie Shoshani. „System Deadlocks“. In: *Computing Surveys* 3.2 (Juni 1971), S. 67–78.
- [6] Fernando J. Corbató, Marjorie Merwin-Daggett und Robert C. Dalex. „An Experimental Time-Sharing System“. In: *Proceedings of the AIEE-IRE '62 Spring Joint Computer Conference*. ACM, 1962, S. 335–344. doi: [10.1145/1460833.1460871](https://doi.org/10.1145/1460833.1460871).
- [7] Robert C. Daley und Jack Bonnell Dennis. „Virtual Memory, Processes, and Sharing in MULTICS“. In: *Communications of the ACM* 11.5 (Mai 1968), S. 306–312. doi: [10.1145/363095.363139](https://doi.org/10.1145/363095.363139).



- [8] Manfred Dausmann, Ulrich Bröckl, Dominic Schoop u. a. *C als erste Programmiersprache: Vom Einsteiger zum Fortgeschrittenen*. Vieweg+Teubner, 2010. ISBN: 978-3834812216. URL: <https://link.springer.com/content/pdf/10.1007%2F978-3-8348-9879-1.pdf>.
- [9] Deutsches Institut für Normung. *Informationsverarbeitung – Begriffe*. Berlin, Köln, 1985.
- [10] Edsger Wybe Dijkstra. *Cooperating Sequential Processes*. Techn. Ber. EWD-123. (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996). Eindhoven, The Netherlands: Technische Universiteit Eindhoven, 1965.
- [11] Edsger Wybe Dijkstra. *Over seinpalen*. Manuskript EWD-74. (dt.) Über Signalmasten. Eindhoven, The Netherlands: Technische Universiteit Eindhoven, 1964 ca.
- [12] Birgit Ewert, Kerstin Christoffer, Uwe Christoffer u. a. *FreeHand 10*. Galileo Design, 2001. ISBN: 3-898-42177-5.
- [13] Arie Nicolaas Habermann. *Introduction to Operating System Design*. Science Research Associates, 1976. ISBN: 0-574-21075-X.
- [14] Per Brinch Hansen. *Betriebssysteme*. München: Carl Hanser Verlag, 1977. ISBN: 3-446-12105-6.
- [15] Richard Craig Holt. „On Deadlock in Computer Systems“. Diss. Ithaca, NY, USA: Cornell University, 1971.
- [16] Richard Craig Holt. „Some Deadlock Properties of Computer Systems“. In: *ACM Computing Surveys* 4.3 (Sep. 1972), S. 179–196.



- [17] Brian W. Kernighan und Dennis MacAlistair Ritchie. *The C Programming Language*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1978.
- [18] Jochen Liedtke. „Improving IPC by Kernel Design“. In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM Press, 1993. ISBN: 0-89791-632-8. doi: [10.1145/168619.168633](https://doi.org/10.1145/168619.168633).
- [19] Jochen Liedtke. „On μ -Kernel Construction“. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (Copper Mountain, CO, USA)*. SOSP'95. New York, NY, USA: ACM Press, Dez. 1995. ISBN: 0-89791-715-4. doi: [10.1145/224057.224075](https://doi.org/10.1145/224057.224075).
- [20] Barbara Liskov. „Primitives for Distributed Computing“. In: *Proceedings of the 7th Symposium on Operating Systems Principles (SOSP '79)*. 1979, S. 33–42.
- [21] Jane W. S. Liu. *Real-Time Systems*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 2000. ISBN: 0-13-099651-3.
- [22] Jürgen Nehmer und Peter Sturm. *Systemsoftware: Grundlagen moderner Betriebssysteme*. dpunkt.Verlag GmbH, 2001. ISBN: 3-898-64115-5.
- [23] Elliot I. Organick. *The Multics System: An Examination of its Structure*. MIT Press, 1972. ISBN: 0-262-15012-3.
- [24] OSEK/VDX Group. *Operating System Specification 2.2.3*. Techn. Ber. OSEK/VDX Group, Feb. 2005. URL: <http://portal.osek-vdx.org/files/pdf/specs/os223.pdf> (besucht am 29.09.2014).



- [25] Rob Pike. *Systems Software Research is Irrelevant*. Talk. CS Colloquium, Columbia University, 2000. URL: <http://herpolhode.com/rob/utah2000.pdf> (besucht am 09.12.2010).
- [26] Vitruv Marcus Vitruvius Pollio. *De Architectura Libris Decem*. Primus Verlag, 1996 (Original 27 v. Chr.)
- [27] Dennis MacAlistair Ritchie und Ken Thompson. „The Unix Time-Sharing System“. In: *Communications of the ACM* 17.7 (Juli 1974), S. 365–370. DOI: [10.1145/361011.361061](https://doi.org/10.1145/361011.361061).
- [28] M. Seltzer, V. Marathe und S. Byan. „An NVM Carol: Visions of NVM Past, Present, and Future“. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, S. 15–23. DOI: [10.1109/ICDE.2018.00011](https://doi.org/10.1109/ICDE.2018.00011).
- [29] Abraham Silberschatz, Greg Gagne und Peter B. Galvin. *Operating System Concepts*. 10. Aufl. John Wiley & Sons, 2018. ISBN: 978-1-118-06333-0.
- [30] Abraham Silberschatz, Greg Gagne und Peter Bear Galvin. *Operating System Concepts*. 7. John Wiley & Sons, Inc., 2005. ISBN: 0-471-69466-5.
- [31] William Stallings. *Operating Systems. Internals and Design Principles*. Sixth. Prentice Hall PTR, 2008. ISBN: 978-0136006329.
- [32] William Stallings. *Operating Systems: Internals and Design Principles*. 9. Aufl. Pearson Education, 2018. ISBN: 978-0-13-467095-9.



Referenzen (Forts.)

- [33] Andrew S. Tanenbaum. *Structured Computer Organization*. Fifth. Prentice Hall PTR, 2006. ISBN: 978-0131485211.
- [34] Andrew S. Tanenbaum und Herbert Bos. *Modern Operating Systems*. 5. Aufl. Pearson Education, 2023. ISBN: 978-1-292-45966-0.
- [35] Andrew Stuart Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Inc., 1997. ISBN: 0-136-38677-6.
- [36] Jim Turley. „The Two Percent Solution“. In: *embedded.com* (Dez. 2002). URL: <https://www.embedded.com/electronics-blogs/significant-bits/4024488/The-Two-Percent-Solution> (besucht am 10. 06. 2019).
- [37] Renate Wahrig-Burfeind. *Universalwörterbuch Rechtschreibung*. Deutscher Taschenbuch Verlag, Sep. 2002. ISBN: 3-423-32524-0.
- [38] Wikipedia. *Prozess*. <http://de.wikipedia.org/wiki/Prozess>. Nov. 2013.