

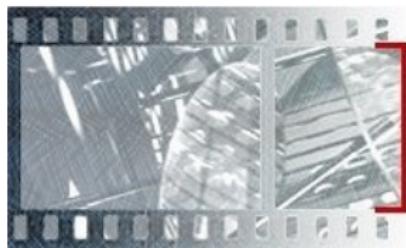
Betriebssysteme

R. Kaiser

(HTTP: <http://www.cs.hs-rm.de/~kaiser>
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2018/2019

0. Vorspann



<http://www.interaktiv-narrativ.org/media/vorspann.jpg>

Vorspann

- ① Einordnung der Veranstaltung
- ② Organisation der Veranstaltung
- ③ Materialien

Einordnung der LV in das Studium

- Pflichtveranstaltung des Bachelor-Studiums
- Baut auf Inhalten des 1. und 2. Semesters auf
 - ▶ Grundlagen der Informatik
 - ▶ Programmieren¹(!) für Praktikum
 - ▶ Rechnernetze und Telekommunikation
- Grundlage für LVen der Semester 4-7
 - ▶ Listenfächer, z.B.
 - ★ Embedded Systems
 - ★ Echtzeitverarbeitung
 - ★ Mobile Computing
 - ▶ Wahlprojekte
 - ▶ Praxisprojekt, Thesis

⇒ Alles, was Spaß macht :-)

¹in C (!!)

Einordnung der LV in das Studium

- Pflichtveranstaltung des Bachelor-Studiums
 - Baut auf Inhalten des 1. und 2. Semesters auf
 - ▶ Grundlagen der Informatik
 - ▶ Programmieren¹(!) für Praktikum
 - ▶ Rechnernetze und Telekommunikation
 - Grundlage für LVen der Semester 4-7
 - ▶ Listenfächer, z.B.
 - ★ Embedded Systems
 - ★ Echtzeitverarbeitung
 - ★ Mobile Computing
 - ▶ Wahlprojekte
 - ▶ Praxisprojekt, Thesis
- ⇒ Alles, was Spaß macht :-)

¹in C (!!)

Inhaltliche Einordnung

Hauptgebiete	Teilgebiete	Untersuchungsgegenstände (Beispiele)
Kern-Informatik	Theoretische Informatik	Automatentheorie Formale Sprachen Informations- und Codierungstheorie Komplexitätstheorie Künstliche Intelligenz
	Praktische Informatik	Betriebssysteme Softwaretechnik Verteilte Systeme Compilerbau
	Technische Informatik	Rechnerarchitektur Rechnernetze Embedded Systems
Angewandte Informatik	Wirtschaftsinformatik Medizinische Informatik Rechtsinformatik	Anwendungsspez. Programmsysteme Decision Support Systeme Krankenhausinformationssysteme Juristische Informationssysteme
	Informatik und Gesellschaft	Folgenabschätzung Ethik in der Informatik Informatik und Datenschutz

Worum geht es?

- Grundlegende Architekturprinzipien von Prozessoren und Rechnersystemen
- Algorithmen und Programme, die der internen Verwaltung eines Rechensystems dienen.
- Organisation und Koordination von (nebenläufigen) Abläufen.
- Optimale oder effiziente Verwaltung von Betriebsmitteln.
- Beispiele: primär UNIX, etwas Windows.
- Betriebssystemfunktionalität ist in vielen anderen Software Systemen enthalten (z.B. JVM, RDBMS, Ada Runtime, ...)

Organisation der Veranstaltung

- Für Angewandte Informatik PO2017, Technische Systeme, Duale Stg.: *Betriebssysteme* (BS)
 - ▶ 2-stündig (Mi 8:15)
 - ▶ Dozent: Hr. Kaiser
- Für Angewandte Informatik PO2010: *Betriebssysteme und Rechnerarchitekturen* (BS+RA)
 - ▶ Vorlesung BS: s.o.
 - ▶ + Vorlesung RA: 2-stündig (Di 8:15)
 - ▶ + Übungen RA: 2-stündig (Mi 10:00)
 - ▶ Dozent: Hr. Gergeleit
- Praktikum:
 - ▶ 2-stündig
 - ▶ 9 Gruppen (7xBS, 2xBS+RA)
 - ▶ Betreuung durch Fr. Dedi, Hr. Flothow, Hr. Heckmann, Hr. Kaiser, Hr. Thoss, Hr. Werner, Hr. Züpke
 - ▶ Erster Termin (BS+RA Gruppe A): Mo, 22.10.)

Organisation der Veranstaltung (2)

- Leistungsnachweis

- ▶ Prüfungsleistung

- ★ BS: 90 Minuten
 - ★ BS+RA: 120 Minuten
 - ★ Zum Bestehen müssen mind. 50% der möglichen Punkte erreicht werden

- ▶ Praktikum: separate Studienleistung

- ★ Bewertung von 8 (BS+RA: 10) Praktikumsaufgaben mit Punkten
 - ★ Ausgabe von Aufgabenzetteln im Intranet
 - ★ Abgabe für alle Teilnehmer zu fixem Termin (Nacht zu Montag 4:00)
 - ★ Automatisierte Plagiatsprüfung (genehmigtes Verfahren)
 - ★ Abnahmegespräche i.d.R. während Praktikumsterminen
 - ★ Zum Bestehen müssen mind. 50% der möglichen Punkte erreicht werden
 - ★ nicht Voraussetzung zur Teilnahme an der Klausur
 - ★ Praktikum geht mit Gewicht 0,3 in die Fachnote ein.

Organisation der Veranstaltung (3)

Vorlesungsstermine BS:

#	Vorlesung	Thema	Bemerkungen
1	Mi, 17. Okt 18	Einführung	
2	Mi, 24. Okt 18	Betriebssystemstrukturen	
3	Mi, 31. Okt 18	Prozesse und Threads	
4	Mi, 7. Nov 18	Prozesse und Threads	
5	Mi, 14. Nov 18	Scheduling	
6	Mi, 21. Nov 18	Prozesssynchronisation	
7	Mi, 28. Nov 18	Prozesssynchronisation	
8	Mi, 5. Dez 18	Prozesskommunikation	
9	Mi, 12. Dez 18	Deadlocks	Vertretung Hr Züpke
10	Mi, 19. Dez 18	Caches	
11	Mi, 9. Jan 19	Speicherverwaltung	
12	Mi, 16. Jan 19	Speicherverwaltung	
13	Mi, 23. Jan 19	Dateisysteme	
14	Mi, 30. Jan 19	Dateisysteme	
15	Mi, 6. Feb 19	Dateisysteme	



Organisation der Veranstaltung (4)

Praktikumstermine:

#	BS (PO17)+ITS				Thema	#	BS+RA (PO10)		Thema	
	AI:A-B	ITS	AI:C-D	AI:E-F			AI:A	AI:B		
1	Mi, 24. Okt 18	Mi, 24. Okt 18	Do, 25. Okt 18	Fr, 26. Okt 18	1. Dateien	1	Mo, 22. Okt 18	Mi, 24. Okt 18	1. Dateien	
2	Mi, 31. Okt 18	Mi, 31. Okt 18	Do, 1. Nov 18	Fr, 2. Nov 18	1. Dateien	2	Mo, 29. Okt 18	Mi, 31. Okt 18	1. Dateien	
	fällt aus		Fr, 9. Nov 18	Do, 8. Nov 18	2. Prozesse	3	Mo, 5. Nov 18	Mi, 7. Nov 18	2. Prozesse	
3	Mi, 14. Nov 18	Mi, 14. Nov 18	Do, 15. Nov 18	Fr, 16. Nov 18	3. Zeitmessungen	4	Mo, 12. Nov 18	Mi, 14. Nov 18	3. Zeitmessungen	
4	Mi, 21. Nov 18	Mi, 21. Nov 18	Do, 22. Nov 18	Fr, 23. Nov 18	4. Threads I	5	Mo, 19. Nov 18	Mi, 21. Nov 18	4. Threads I	
5	Mi, 28. Nov 18	Mi, 28. Nov 18	Do, 29. Nov 18	Fr, 30. Nov 18	5. Threads II	6	Mo, 26. Nov 18	Mi, 28. Nov 18	5. Threads II	
6	Mi, 5. Dez 18	Mi, 5. Dez 18	fällt aus		6. Papierübung I	7	Mo, 3. Dez 18	Mi, 5. Dez 18	6. MIPS-Assembler	
7	Mi, 12. Dez 18	fällt aus		Do, 13. Dez 18	Fr, 14. Dez 18	7. Signale	8	Mo, 10. Dez 18	Mi, 12. Dez 18	6. MIPS-Assembler
8	Mi, 19. Dez 18	Mi, 19. Dez 18	Do, 20. Dez 18	fällt aus		7. Signale	9	Mo, 17. Dez 18	Mi, 19. Dez 18	7. Signale
9	Mi, 16. Jan 19	Mi, 16. Jan 19	Do, 17. Jan 19	Fr, 18. Jan 19	8. Pipes	10	Mo, 14. Jan 19	Mi, 16. Jan 19	8. Pipes	
10	Mi, 23. Jan 19	Mi, 23. Jan 19	Do, 24. Jan 19	Fr, 25. Jan 19	9. Shared Memory	11	Mo, 21. Jan 19	Mi, 23. Jan 19	9. Shared Memory	
11	Mi, 30. Jan 19	Mi, 30. Jan 19	Do, 31. Jan 19	Fr, 1. Feb 19	10. Papierübung II	12	Mo, 28. Jan 19	Mi, 30. Jan 19	10. Cache-Simulator	
12	Mi, 6. Feb 19	Mi, 6. Feb 19	Do, 7. Feb 19	Fr, 8. Feb 19	11. Papierübung III	13	Mo, 4. Feb 19	Mi, 6. Feb 19	10. Cache-Simulator	

Ausfalltermine	Grund
Do, 6. Dez 18	Dies Academicus
Fr, 21. Dez 18	Weihnachtsferien
Mi, 7. Nov 18	FSR-Vollversammlung
Mi, 12. Dez 18	AI Gruppe B: Vertretung durch Hr. Zupke

Abgabetermine (Mo, 04:00)			
Bl.#	Thema	Termin	Hinweis
-	-	-	
1	Dateien	Mo, 5. Nov 18	
-	-	-	
2	Prozesse	Mo, 19. Nov 18	
3	Zeitmessungen	Mo, 26. Nov 18	
4	Threads I	Mo, 3. Dez 18	
5	Threads II	Mo, 10. Dez 18	
6	MIPS-Assembler	Mo, 17. Dez 18	Nur BS+RA (PO10)
7	Signale	Mo, 14. Jan 19	
8	Pipes	Mo, 21. Jan 19	
9	Shared Memory	Mo, 28. Jan 19	
-	-	-	
10	Cache-Simulator	Mo, 11. Feb 19	Nur BS+RA (PO10)

Materialien

① Folien zur Vorlesung

- ▶ werden als .pdf-Dateien kapitelweise im Intranet bereitgestellt.
- ▶ <http://wwwvs.cs.hs-rm.de/lehre/>
- ▶ oder <http://www.cs.hs-rm.de/~kaiser>

② Übungsblätter

- ▶ für Programmier- und Papierübungen des Praktikums
- ▶ werden als .pdf-Dateien kapitelweise im Intranet bereitgestellt (s.o.).

③ Arbeitsplatzrechner zum freien Üben

- ▶ Linux PC-Pools (Di nachmittags und wenn Pool frei)
- ▶ Aktuell: Ubuntu 18.0.4

④ Linux für zuhause

- ▶ sollten Sie seit dem 1. Semester haben !
- ▶ Virtuelle Maschine (z.B. VirtualBox oder VmWare)
- ▶ Images (z.B.) unter <https://www.osboxes.org/ubuntu/>

⑤ eLearning-Material

- ▶ für ausgewählte Themen verfügbar
- ▶ <http://wwwvs.cs.hs-rm.de/lehre/>

Materialien (2)

⑥ Lehrbücher (siehe auch Modulbeschreibung)

A.S. Tanenbaum, H. Bos: *Moderne Betriebssysteme*

Pearson 2014

ISBN 978-3868942705

69,95 €



W. Stallings: *Operating Systems: Internals and Design Principles*

Pearson 2014

ISBN 978-0133805918

72,86 €



A. Silberschatz, P. B. Galvin: *Operating System Concepts*

Wiley 2018

ISBN 978-1119127482

78,75 €



Materialien (3)



7 Lehrbücher zur Rechnerarchitektur

D. Patterson, L. Hennessy: *Rechnerorganisation und Rechnerentwurf: Die Hardware/Software-Schnittstelle*

De Gruyter 2018

ISBN 978-3110446050

69,95 €



A.S. Tanenbaum, T. Austin: *Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner*

Pearson 2014

ISBN 978-3868942385

47,99 €



Kapitel 1: Einführung



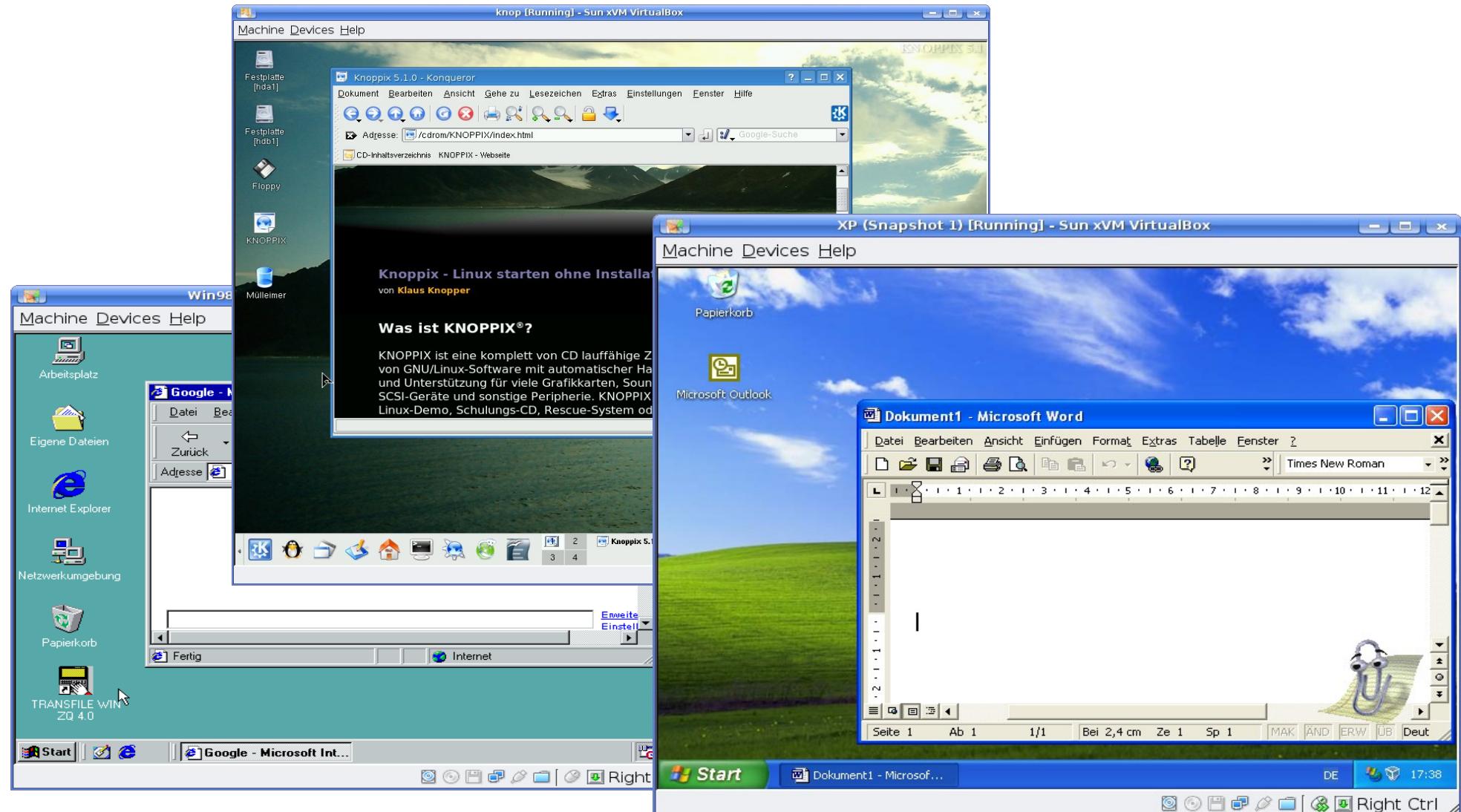
Kap. 1: Einführung

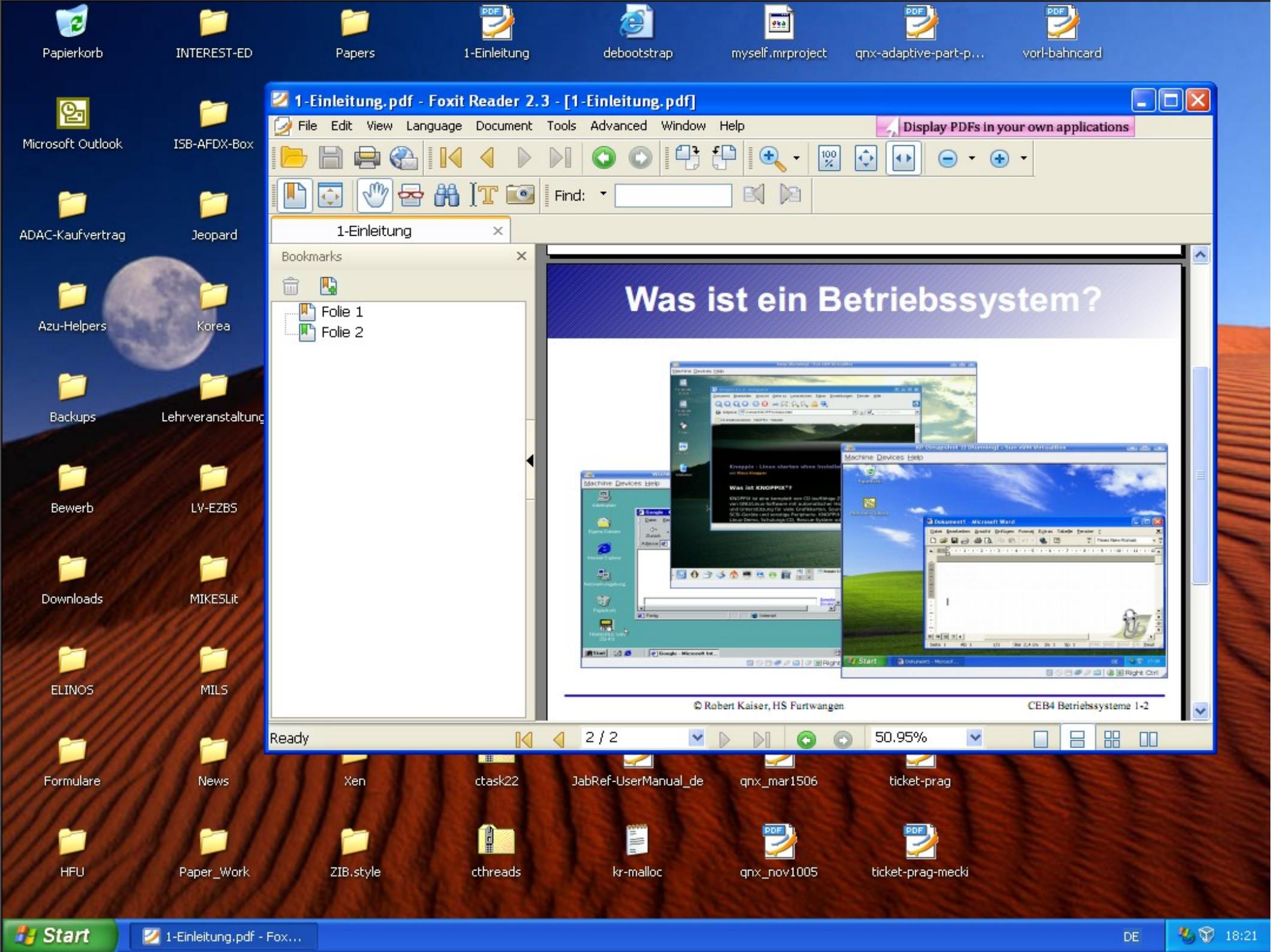
- 1.1 Was ist ein Betriebssystem?
- 1.2 Geschichte der Betriebssysteme
- 1.3 Geschichte von UNIX
- 1.4 Zusammenfassung

1.1 Was ist ein Betriebssystem?



Was ist ein Betriebssystem?





Windows

A fatal exception 0E has occurred at 0028:C0011E36 in 0XD UMM(01) + 00010E36. The current application will be terminated.

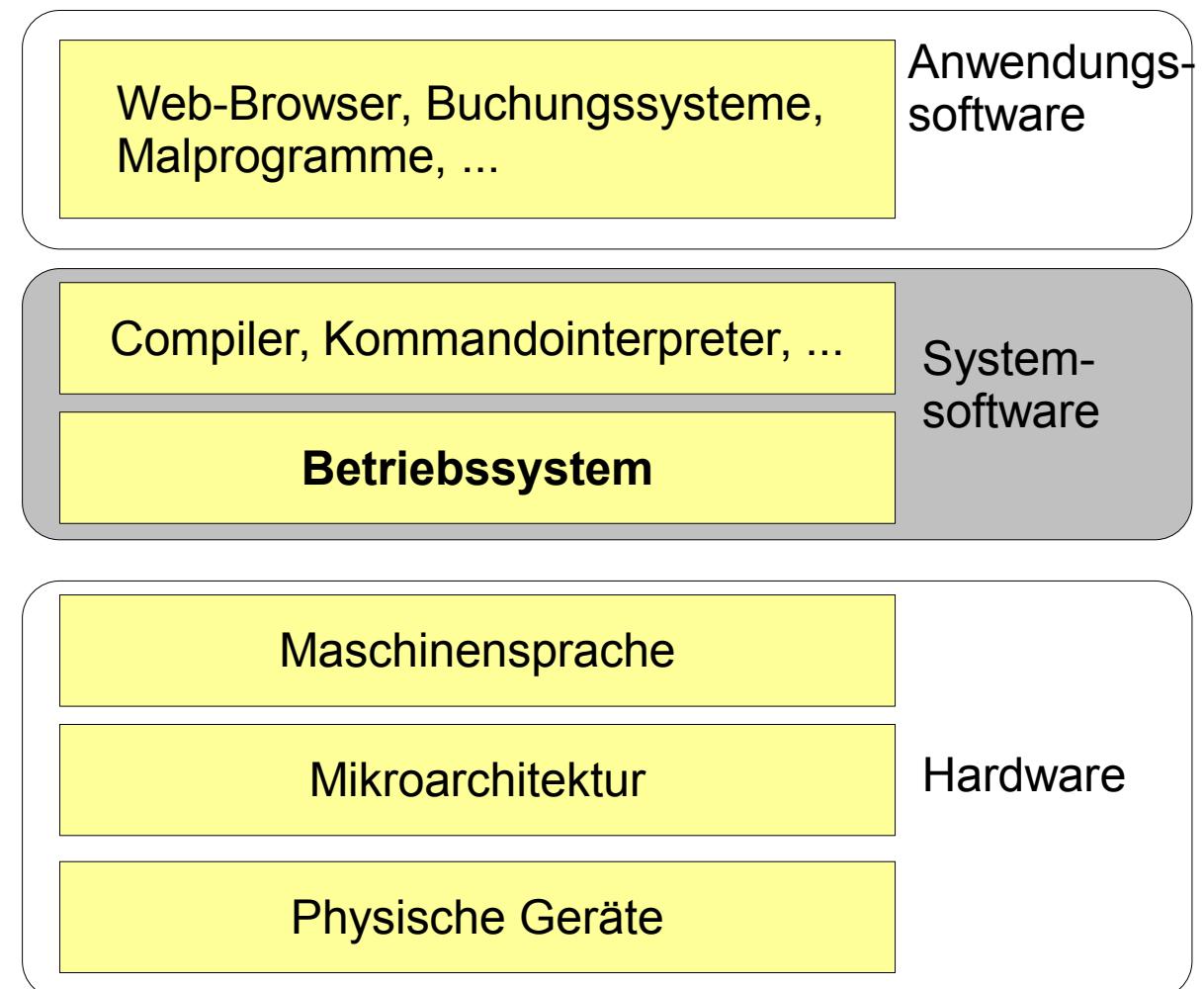
- * Press any key to terminate the current application.
- * Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue _

Was ist ein Betriebssystem?

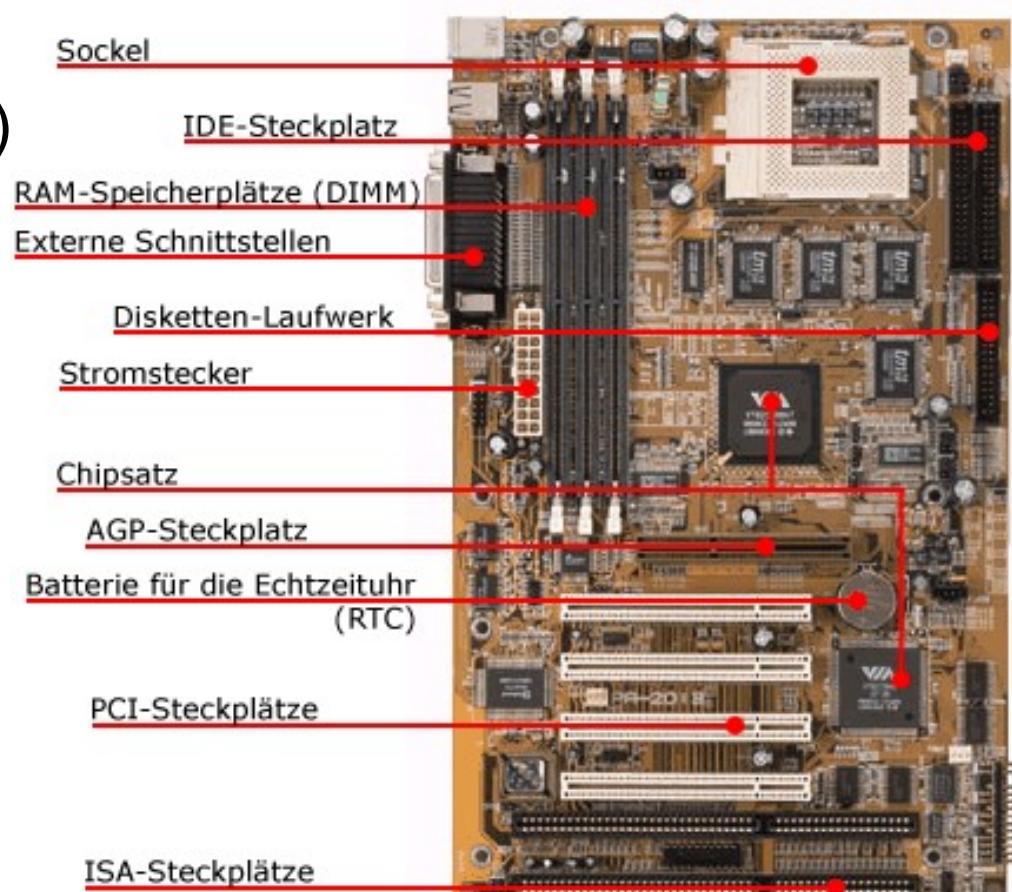
Allgemeines Schichtenmodell eines Rechensystems

- Anwendungssoftware
„Nutzen“ des Rechers
- Systemsoftware
Zum Betrieb des
Rechners erforderlich
(„notwendiges Übel?“)
- Betriebssystem:
grundlegendstes
Systemprogramm



Komponenten eines Rechners

- Prozessor(en) („CPUs“)
- Hauptspeicher („Core Memory“)
- Bildschirmschnittstelle (Grafikkarte, Terminal)
- Massenspeicher (Festplatte, Diskette, Flash-Speicher)
- Netzwerkschnittstelle(n)
- Uhren
- Soundchip
- Drucker, Webcam,



Nutzung der Komponenten

- Das Erstellen von Programmen zur Verwaltung und möglichst optimalen Nutzung dieser Komponenten ist **schwierig**.
- Aufgrund des rapiden technischen Fortschritts sind solche Programme zudem sehr kurzlebig.
- Beispiel: Festplatten-Zugriff
ATAPI – Befehlssatz (*):

- EXECUTE DEVICE DIAGNOSTIC
- IDENTIFY DEVICE
- INITIALIZE DEVICE PARAMETERS
- READ DMA
- READ MULTIPLE
- READ SECTOR(S)
- READ VERIFY SECTOR(S)
- SEEK
- SET FEATURES
- SET MULTIPLE MODE
- WRITE DMA
- WRITE MULTIPLE
- WRITE SECTOR(S)

Wer will schon mit solch einer Schnittstelle arbeiten, um Informationen langfristig zu speichern oder darauf zuzugreifen ?!?

(*) <http://www.t10.org/t13/project/d1153r18-ATA-ATAPI-4.pdf>

- System-Software stellt **Dienste** bereit, z.B.:
 - **Dateiverwaltung** (Dateien / Ordner öffnen / lesen / schreiben, Suchfunktionen, Zugriffsschutz, ...)
 - **Ein-/Ausgabemöglichkeiten** (Tastatur / Maus, Bildschirm / Drucker, ..)
 - **Programmierumgebung** (Editor, Compiler, Assembler, Linker, Debugger, Bibliotheken,..)
 - Evtl. **Mehrbenutzerfähigkeit** (Gemeinsame Nutzung „teurer“ Gerätschaften)
 - **Netzwerkzugang** (Internet, (W)LAN, ..)
 - ...

- Der Funktionsumfang (Befehlssatz) der realen Maschine wird erweitert
- Ziele dabei:
 - Komplexität der „nackten“ Hardware verstecken
 - Abstrakte Schnittstellen auf hohem Niveau
 - Leichter zur verstehen und auch langlebiger
- Die Vorstellung solcher Abstraktionen und ihrer Realisierung im Betriebssystem ist Gegenstand der Vorlesung.

- [Def.: Betriebsmittel (BM, engl. *resources*): Alle zuteilbaren und nutzbaren Hardware- und Software- Komponenten eines Rechensystems]
- Aufgabe des Betriebssystems: Geordnete, kontrollierte Zuteilung („Allokation“) der BM an die um sie konkurrierenden Programme.
- Gemeinsame Nutzung von Betriebsmitteln (z.B. Drucker)
- Schutz vor unberechtigter Benutzung, Geheimhaltung von Informationen, wenn ihr Benutzer dies wünscht
- Vermitteln im Falle von Konflikten
- Abrechnen der Kosten der Betriebsmittelnutzung (*accounting*)

Zusammenfassung

Definition (Betriebssystem):

- *Ein Betriebssystem ist ein Programm, das alle Betriebsmittel eines Rechensystems verwaltet und ihre Zuteilung kontrolliert und den Nutzern des Rechensystems eine virtuelle Maschine offeriert, die einfacher zu verstehen und zu programmieren ist, als die unterlagerte Hardware.*



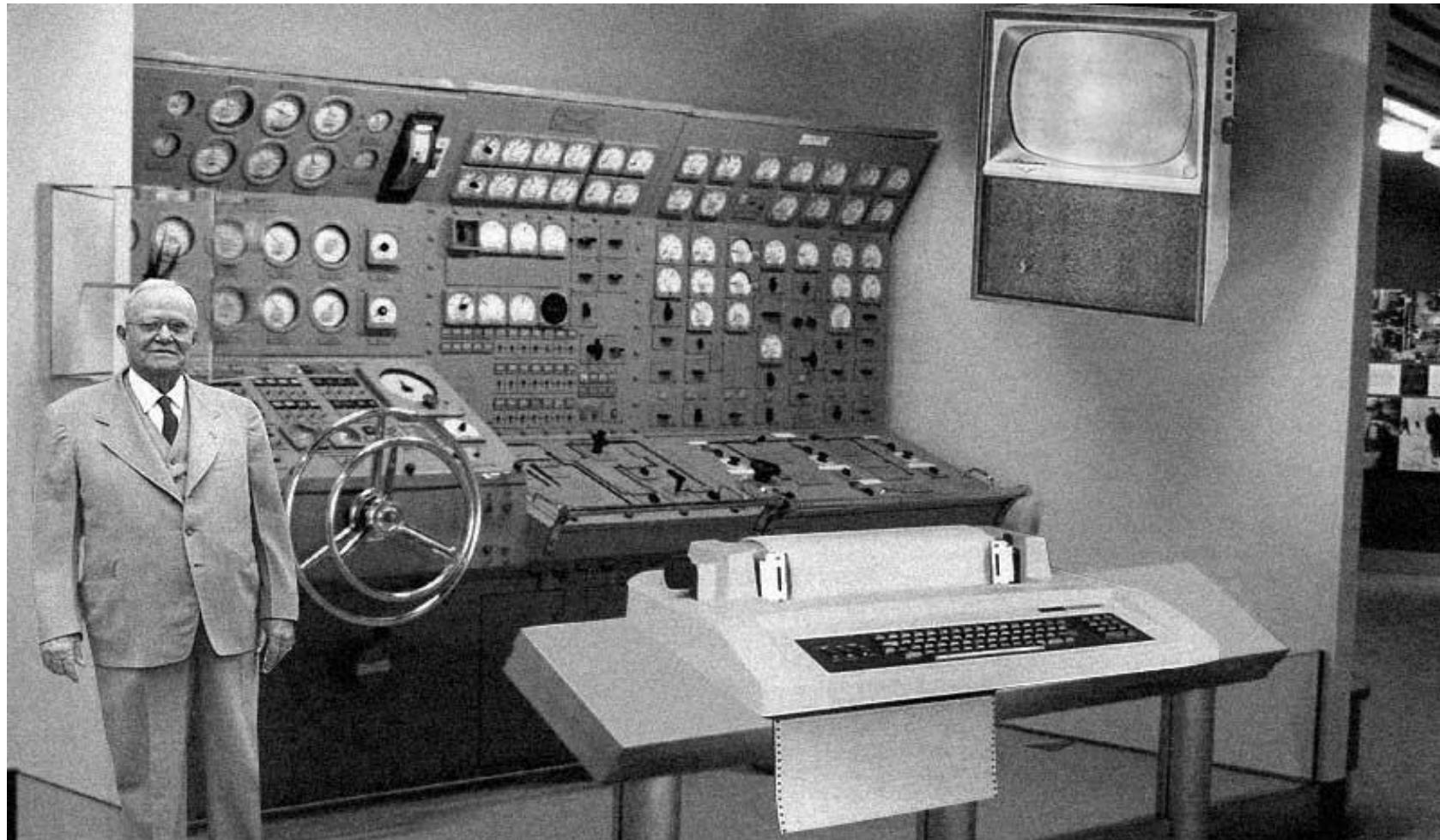
- Entwicklung in etwa parallel zu Rechnergenerationen:
 - 1.(1945-1955): Röhren und Steckkarten
 - 2.(1955-1965): Transistoren und Stapelverarbeitung
 - 3.(1965-1980): ICs und Mehrprogrammbetrieb
 - 4.(1980-1990): PCs und Netzwerkbetriebssysteme
 - 5.(1990-heute): ...?



1.2.1 Generation (1945-1955)

- USA: Howard Aiken (Harvard), John von Neumann (Princeton), J.P. Eckert, W. Mauchley, (UPenn)
- Deutschland: Konrad Zuse
- Eine Gruppe von Personen kümmert sich um Entwurf, Bau, Programmierung, Betrieb und Wartung des Rechners
- „Programmieren“ = Verdrahten von Steckkarten oder absolute Maschinensprache (keine Programmiersprachen, nicht einmal Assembler).
- Ab Anfang der 50er Jahre: Benutzung von Lochkarten als Ersatz für Steckkarten.

1. Generation (1945-1955)



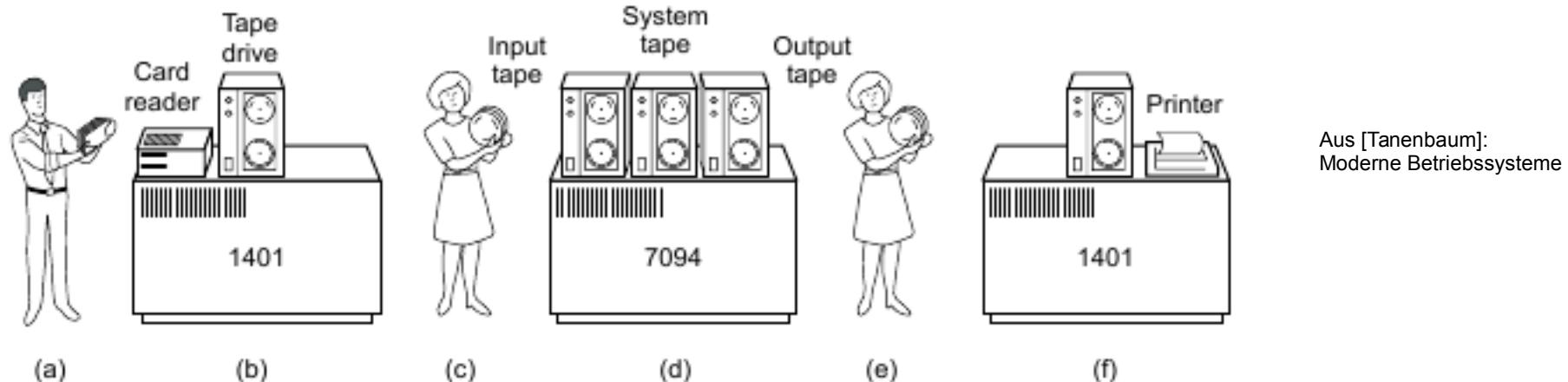
... Betriebssystem???



1.2.2. 2. Generation (1955-1965)

- Durch die Einführung der Transistorstechnologie werden Rechner zuverlässig genug, um an Kunden verkauft zu werden
- Unterscheidung zwischen Entwicklern, Herstellern, Operateuren, Programmierern und Wartungspersonal
- Zunächst Ausführung einzelner „Jobs“ (= Programm oder Menge von Programmen) in Form von Lochkartenstapeln mit hohem Anteil manueller Arbeiten
- Rationalisierung des Betriebs (engl.: *operating*) durch Einführung des „Stapelverarbeitungsbetriebs“ (*batch system*) ...

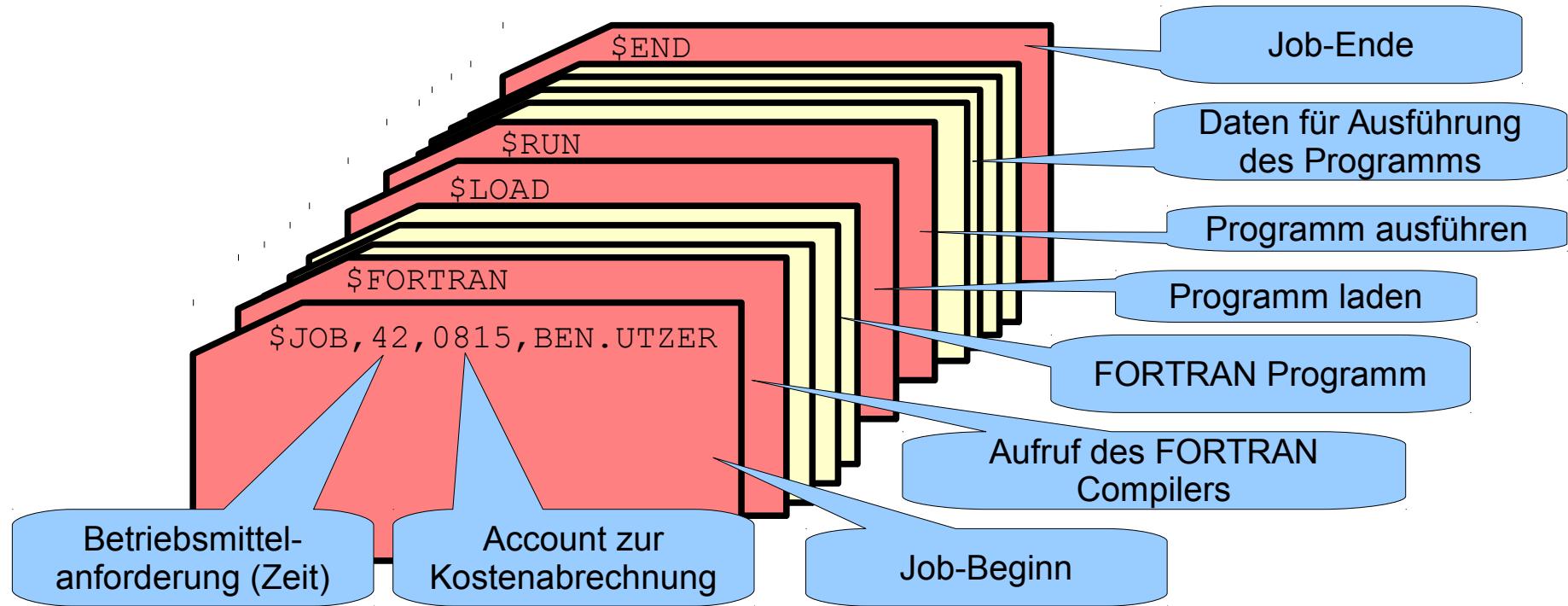
Stapelverarbeitung



Aus [Tanenbaum]:
Moderne Betriebssysteme

- (a) Programmierer bringen Lochkartenstapel zur Lesestation
(z.B. IBM 1401: relativ preiswerter Rechner für kommerzielle DV)
- (b) Lesestation sammelt Kopien (1:1) auf Magnetband
- (c) OperateurIn trägt Magnetband von Zeit zu Zeit zum Hauptrechner
- (d) Hauptrechner (z.B. IBM 7094 mit mehreren Bandstationen) erzeugt für das Eingabeband ein Ausgabeband mit Ergebnissen aller Programme. Die Abarbeitung geschieht eigenständig durch ein „Monitorprogramm“.
- (e) OperateurIn trägt Ausgabeband zurück
- (f) Drucken des Ausgabebandes unabhängig vom Hauptrechner

Stapelverarbeitung

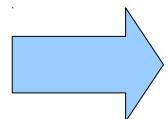


- Kontrollkarten sind Vorläufer der heutigen Job-Kontrollsprachen und der Kommandointerpreter
- Das Monitorprogramm zur Abarbeitung der Job-Folge ist der Vorläufer eines Betriebssystems



1.2.3 3. Generation (1965-1980)

- Einführung von *Rechnerfamilien* (Urvater IBM /360) mit gleichem Befehlssatz aber Unterschieden hinsichtlich Preis und Leistung => Rechner sind „softwarekompatibel“: Dasselbe Programm läuft auf allen Rechnern der Familie.
- Gleichermassen für wissenschaftliche wie kommerzielle Berechnungen geeignet
- Besseres Preis/Leistungsverhältnis durch integrierte Schaltkreise
- Anforderung: Die gesamte Software (auch das Betriebssystem) sollte auf allen Modellen der Familie lauffähig sein.
- Verschiedenste Anforderungen an das BS: E/A-orientierte Anwendungen vs. numerische Berechnungen, etc.



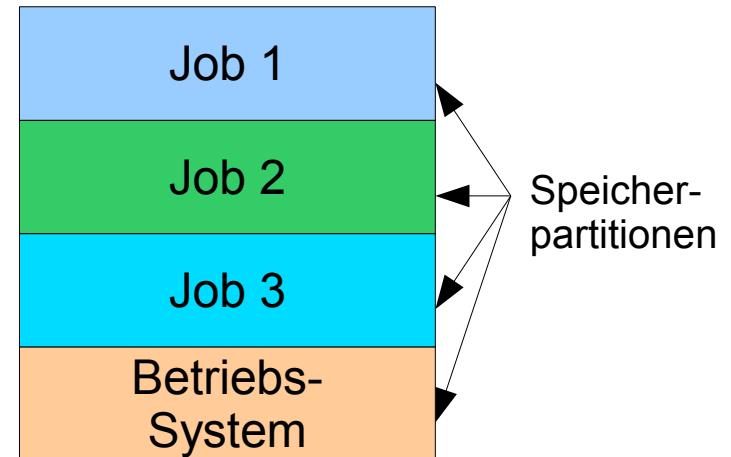
Komplexes Betriebssystem...

- Eigenschaften des Betriebssystems (OS /360)
 - Mehrere Zehnerpotenzen größer als frühere Stapelverarbeitungsprogramme
 - Millionen Zeilen Assemblercode
 - Von Tausenden von Programmierern entwickelt
 - Tausende von Fehlern
 - Fortlaufende Releases notwendig
 - Niemals „fehlerfrei“



Mehrprogrammbetrieb

- Neue Schlüsseltechnik (Multiprogramming, Multitasking)
- Ziel: Vermeiden von Wartezeiten für „teuren“ Prozessor während E/A-Vorgängen, besonders relevant in kommerziellen Anwendungen.
- Lösung:
- Gleichzeitiges Bereithalten mehrerer Jobs im Hauptspeicher („Partitionierung“)
- Anstatt auf E/A zu warten: Umschalten auf anderen Job
- ➔ Neue Anforderung: Schutz der Speicherbereiche (wurde später hinzugefügt)



- Variante: Timesharing
 - Drang nach kurzen Antwortzeiten
 - Jeder Benutzer hat über ein eigenes Terminal on-line Zugang zum System
 - Schnelle Reaktion des Systems auf Benutzeingaben (vorrangig behandelt), Stapelverarbeitung „im Hintergrund“
 - Speicherschutz unerlässlich (vgl. CTSS (MIT, [Corbato 1962]) auf modifizierter IBM 7094)

- Das Beispiel: MULTICS
 - MULTplexed Information and Computing System
 - Zu Beginn (1965) gemeinsames Projekt von MIT, Bell Labs, General Electric, brach auseinander wegen technischer Schwierigkeiten
 - Ziel: Hunderte von Nutzern gleichzeitig im Dialogbetrieb auf GE645.
 - Letztendlich nur im MIT und einigen andern Einrichtungen produktiv eingesetzt
 - Aber: bedeutender Einfluss auf nachfolgende Systeme (UNIX!)

- Weniger Speicher, geringere Leistung als Großrechner zu einem Bruchteil des Preises
- Beginn 1961: DEC PDP-1
 - 4K 18-bit Worte
 - \$120.00,- (5% vom Preis einer IBM 7094)
- Für bestimmte, nicht-numerische Aufgaben sogar schneller als eine 7094
- Schnelle Abfolge von (nicht zueinander kompatiblen) PDP-Rechnern bis zur PDP-11 (1970)

Aufkommen der Minicomputer



Ken Thompson und Dennis M. Ritchie an einer PDP-11 (ca. 1972)



1.2.4 4. Generation (1980-1990)

- Personal Computer als Individuen zugeordnete Werkzeuge („Workstation“)
- Getrieben durch LSI und VLSI-Entwicklung: preiswert, zugleich aber leistungsstark wie Mini- oder Großrechner
- Netzwerke für Kommunikation und Kooperation
- Hohe Grafikleistung, benutzerfreundliche Oberflächen (z.B. Apple Macintosh) erschließen neue Benutzergruppen, die keine eigentlichen Rechnerkenntnisse besitzen (müssen).
- Dominante Betriebssysteme: MS-DOS und UNIX
- Netzwerkbetriebssysteme erlauben Zugang zu anderen Rechnern, Dateitransfer, gemeinsame Nutzung von Informationen (z.B. TCP/IP Netzwerk Utilities, Network File System)

- Neue Anwendungen gekennzeichnet durch
 - steigende Komplexität
 - Probleme bei der Sicherheit
 - neu geforderte Funktionalitäten
 - Web-Anbindung
 - Multimedia
 - Sicherheit (security)
 - Sicherheit (safety)
 - Fehlertoleranz/Robustheit
 - Echtzeitfähigkeit
 - Skalierbarkeit

- Über lange Zeit regelmäßige Steigerungen der Rechenleistung durch höhere Taktfrequenzen
- Physikalische Grenzen werden hier allmählich erreicht
 - Trend zu on-Chip Parallelität (z.B. Multicore-Prozessoren)
 - neue Herausforderungen, auch für Betriebssysteme
- Netzwerke:
 - höhere Bandbreiten
 - mobile Knoten, drahtlose Netze
 - veränderliche Topologie



Aktuelle Themen

- Symmetrisches Multiprocessing (SMP): Das Betriebssystem unterstützt symmetrische Multiprozessorsysteme, (alle Prozessoren sind gleichwertig).
- Virtualisierung: Mehrere BS-Schnittstellen auf einem System:
 - Unterstützung für Vielfalt von Anwendungen, Ausführbarkeit ohne Änderung.
 - Konsolidierung (nicht nur) für Rechenzentren
 - Fehlerlokalität (*Multi Level Secure Systems*)
- Anwendungen treten in den Vordergrund, die unterlagerten Rechensysteme werden austauschbarer.
- Echtzeitfähigkeit (auch für nicht-technische Anwendungen, wie etwa multimediale Informationsströme und ihre Synchronität).



Aktuelle Themen

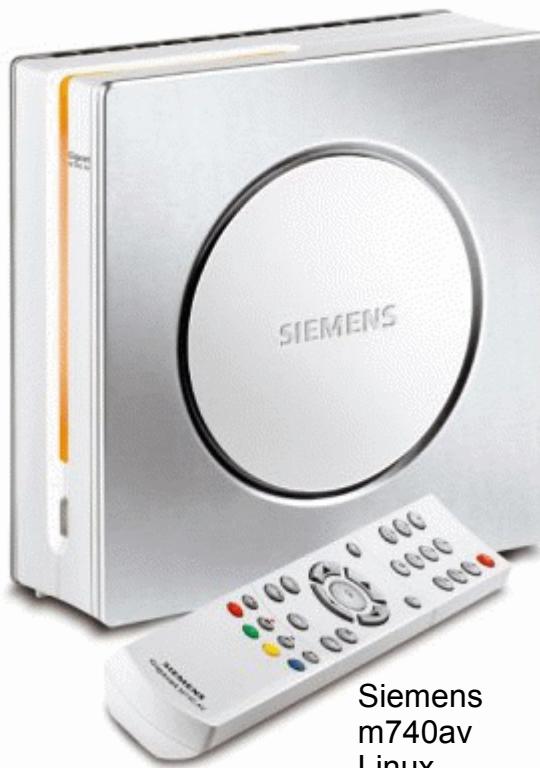
- Übergang zu verteilten Betriebssystemen: Das gesamte Netzwerk erscheint Benutzern wie ein einziges System.
- Grid Computing (Virtualisierung von Rechenleistung im Internet)
- Drang nach standardisierten Schnittstellen (APIs: Application Programming Interfaces):
- Hochverfügbarkeit
- Energieeffizienz („Green IT“)
- Unterstützung für die Administration großer Systeme, Netzwerk und Anwendungen (Integriertes System-, Netzwerk- und Anwendungs-Management).



Heutiges Spektrum an BS

- Mainframe-Betriebssysteme: Massendatenhaltung, hohe E/A-Raten, „legacy“ Anwendungen. (z.B. OS/390, BS2000)
- Server-Betriebssysteme: Bereitstellung gemeinsam genutzter Betriebsmittel und Dienste. (z.B. UNIX / Linux, Windows Server)
- Arbeitsplatzrechner- (Desktop-) Betriebssysteme: (z.B. Windows 95/98/2000/XP/Vista, Linux, MacOS)
- Echtzeit-Betriebssysteme: Gesicherte Einhaltung von Zeitschränken für die Verarbeitung. (z.B. VxWorks, QNX, RTLinux)
- Betriebssysteme für *eingebettete Systeme*: i.d. R. Produkte mit stark eingeschränktem Betriebsmittelumfang (Handys, PDAs, SmartCards, ABS, ...). Oft auch Echtzeiteigenschaften gefordert. (z.B. Symbian, PalmOS, Linux, ...)

Beispiele: Embedded Systems



DVB-T Fernsehempfänger

Siemens
m740av
Linux



Handspring Treo 600,
PalmOS

Handy



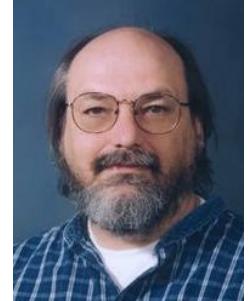
Linksys WRT54g,
Linux

WLAN-Router

1.3 Geschichte von UNIX



- 1965 - AT&T Bell Labs, General Electric und MIT beginnen mit Entwicklung des Betriebssystems MULTICS („MULTIplexed Information and Computing System“, s.o.): gute Ideen, wenig Erfolg
- 1969 – Ken Thompson entwickelt UNICS („UNiplexed Information and Computing System“) auf PDP-7 (4KB Speicher!)
- Ken Thompson, Dennis Ritchie und andere entwickeln UNIX für PDP-11
- Dateisystemaufbau und -Schnittstelle, Kommandointerpreter (Shell) und anderes entstammen MULTICS-Ideen
- 1973 – Reimplementierung in C (von Ritchie entworfen)
- 1974 – Meilenstein-Papier zu Unix von Ritchie und Thompson in CACM
- Einfachheit und Eleganz (später ACM Turing Award)





Kleine UNIX-Historie

- Schnelle Verbreitung im akademischen Bereich, da das System mit allen Quellen ausgeliefert wurde.
 - Version 6 in 1976
 - Bis 1977 an ca. 500 Institutionen vergeben
 - Version 7 (Portables UNIX) in 1978, mit Variante 32/V für DEC VAX
- Zahlreiche, divergierende Versionen
- Zeitweise Spaltung AT&T System V und Berkeley System Distribution (BSD) (s.u.), ab 1998: Versuch der Vereinheitlichung
- POSIX (Portable Operating System Interface for UNIX)
 - durch das anerkannte, unabhängige IEEE-Gremium fand POSIX Beachtung als Standard 1003.1
- Heute: Mehrere, weitgehend POSIX-konforme Versionen (Linux, *BSD, Solaris (Sun), AIX (IBM), ...)

- Univ. of California at Berkeley bekam früh AT&T V 6, dann 32/V
 - eigene Weiterentwicklungen als 1BSD, 2BSD, 3BSD (Berkeley Software Distribution) abgegeben,
- 1979: 3BSD für VAX (Neu: Virtual Memory, Demand Paging)
 - (u. A. durch William Joy, späterer SUN Mitbegründer)
- 4BSD-Entwicklung für VAX mit zahlreichen Verbesserungen
 - ab 1979 4BSD-Entwicklung mit Förderung durch DARPA (Defense Advanced Research Agency).
 - DARPA Net: Ursprung des Internet
 - Höhepunkt 4.2BSD in 1983: Neues Virtual Memory Interface, Device Driver Interface, Fast File System, Stabilere Signalverarbeitung, Socketbasierte Interprozesskommunikation, Netzwerkstandards: TCP/IP im Kern und Netzwerk-Utilities (rlogin, telnet, rcp, ftp, rsh).

- Von vielen Herstellern als Basis für Portierungen benutzt (MC 68020, NS32032, i80386).
 - Starke Rückwirkung auf AT&T System V Entwicklung sowie Eingang in Standardisierung POSIX durch IEEE Std. 1003 und X/OPEN.
- 4.3BSD/4.4BSD mit Quellcode heute allgemein zugänglich.
 - FreeBSD
 - OpenBSD
 - NetBSD
 - Dragonfly BSD
- MacOS X und Darwin basieren auf 4.4BSD

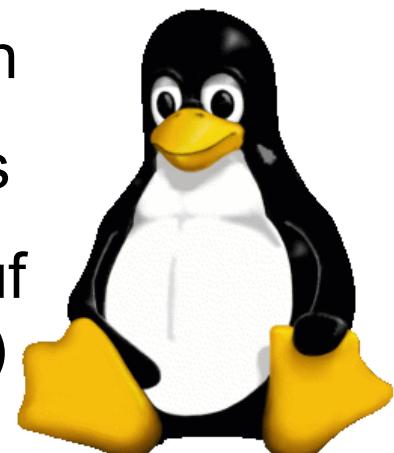




Das „kommerzielle“ UNIX

- AT&T Bell Labs:
 - Zusammenfassung versch. Varianten bis 1982 zu "UNIX System III"
 - ab 1983 System V als Produkt mit offiziellem Support und fortlaufenden Releases:
 - V.2 in 1984 (100.000 Installationen),
 - Release V.3 in 1987 mit viel Einfluss von 4.2BSD UNIX.
- Novell:
 - 1993: kauft den UNIX-Teil von AT&T (300 Mio \$)
 - SUN kauft sich 1994 von Novell frei (100 Mio \$).
 - 1995: Verkauf der UNIX System V-Quellen an SCO
 - Caldera übernimmt SCO (2001) und nennt sich SCO Group (2002)
 - 2003: SCO-Caldera beschuldigt verschiedene Linux-Firmen und IBM des Diebstahls geistigen Eigentums (Intellectual Property)

- 1991: Der Finnische Informatikstudent Linus Torvalds beginnt mit der Entwicklung eines freien UNIX-Kerns entsprechend der Single UNIX Specification
- Gewinnt im Server-Markt stark an Bedeutung
- Attackiert Windows im Desktop-Bereich
- Unterstützung durch führende Hersteller (z.B. IBM, HP, ...)
- Aufgrund der freien Verfügbarkeit und wachsender Betriebsmittel auch erfolgreich im embedded-Bereich
- Heute das populärste Betriebssystem nach Windows
- Große Vielfalt an Distributionen, oft zugeschnitten auf das Einsatzgebiet (Desktop / Server / Embedded / ..)



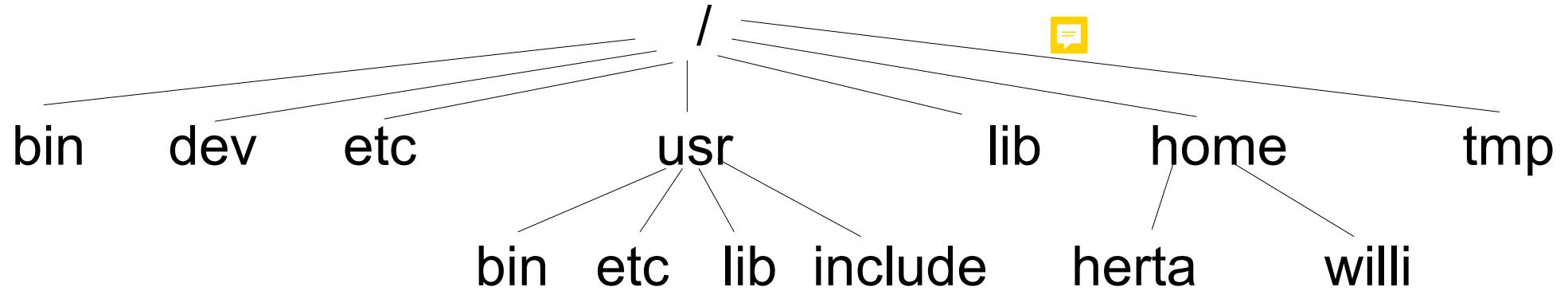
- Mehrbenutzer- und Mehrprogrammbetrieb (*multi-user / multi-tasking*)
- Hierarchisches Dateisystem
 - eine Wurzel ("/")
 - Physische Geräte sind Teil des Dateibaums
- Hohe Übertragbarkeit, dadurch verfügbar vom PDA bis zum Großrechner
- größtenteils in C geschrieben
- Mächtige Kommandosprache („Shell“): einfache Bausteine, aber flexible Verknüpfungsmöglichkeiten



Beispiel: Shell-Kommandos

- Gegeben: Datei mit Städtenamen (einer je Zeile)
- Frage: Wie oft kommt „Kirchveischede“ vor?
- Folgende einfachen Kommandos gibt es schon:
 - „grep“ sucht in seiner Eingabe nach Zeilen, die Suchbegriff enthalten
 - „wc“ (word counter) zählt Zeilen / Wörter / Buchstaben in seiner Eingabe
 - mit „|“ kann man Aus- und Eingabe zweier Kommandos verbinden ("Pipeline")
- Lösung: grep "Kirchveischede" datei | wc
- ➔ „Baukastenprinzip“

Dateibaum



- / Wurzel des Dateisystems
- /bin, /usr/bin - ausführbare Programme, Kommandos
- /dev Device-Dateien, direkter Zugriff auf angeschlossene Geräte (Scanner, Drucker, Platten...)
- /etc Konfigurationsdateien (Paßwörter, Netzkonfig, ...)
- /lib C-Bibliotheken
- /home Benutzerverzeichnisse
- /tmp Arbeitsverzeichnis für temporäre Dateien

- `cd` aktuelles Verzeichnis wechseln
- `rm` Datei löschen (*remove*)
- `mkdir` Verzeichnis anlegen
- `rmdir` Verzeichnis löschen (*remove directory*)
- `ps` Prozeßliste ausgeben
- `mv` Datei verschieben / umbenennen (*move*)
- `cat` Datei(en) ausgeben
- `ls` Dateien auf/*list*en ("`ls -l`" für mehr Infos)
- `man` Online-Manual abrufen (z.B. "`man ls`")
- `mount` Einbinden von Dateibäumen



1.4 Zusammenfassung

1. Sichtweise eines Betriebssystems:
 - Betriebsmittelverwalter
 - (erweiterte) virtuelle Maschine
2. Geschichtliche Entwicklung, dabei Begriffe:
 - Stapelbetrieb (Batch Processing)
 - Mehrprogrammbetrieb (Multiprogramming)
 - Spooling
 - Dialogbetrieb (Timesharing)
3. Geschichte von UNIX

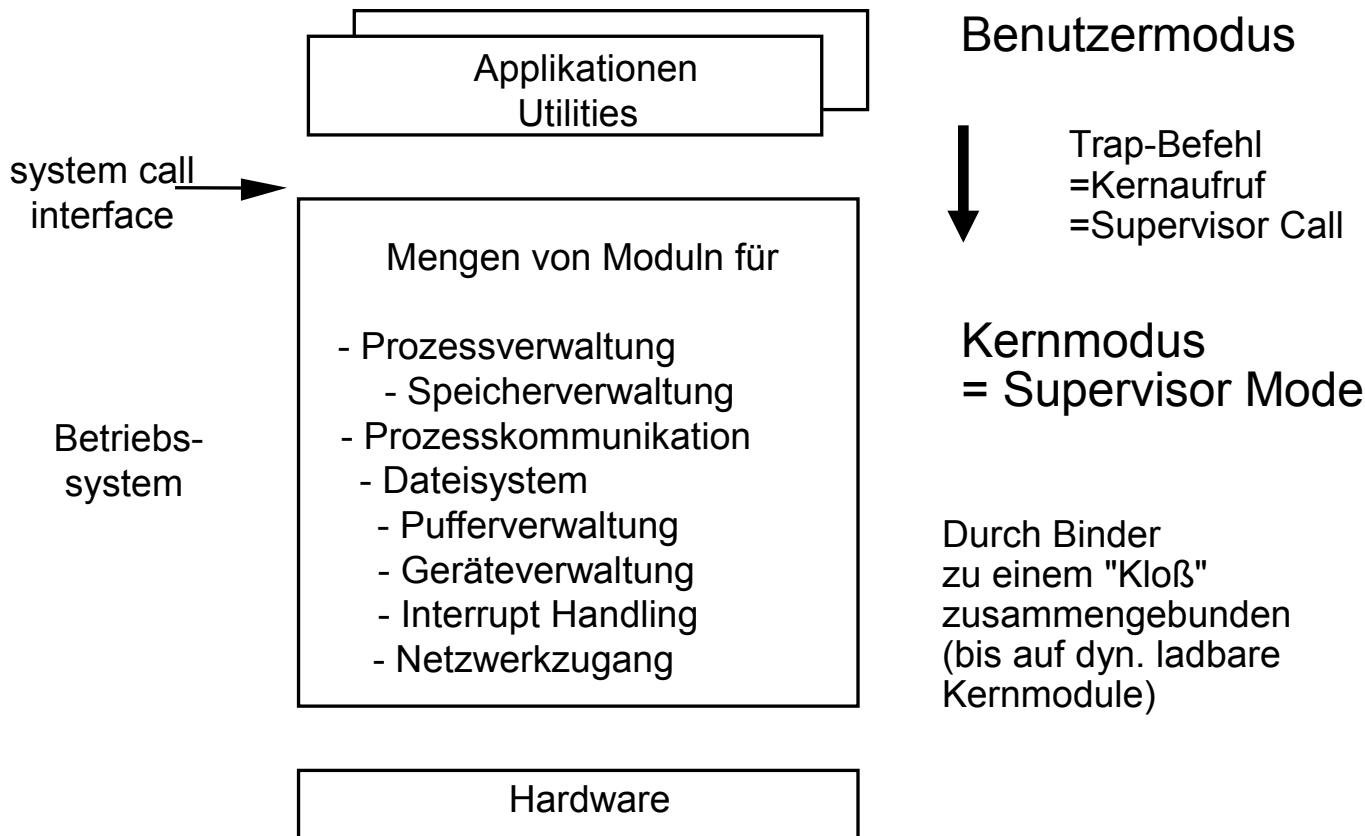
Kap. 2: **Betriebssystem-** **Strukturen**

- 2.1 Monolithische Systeme
- 2.2 Geschichtete Systeme
- 2.3 Virtuelle Maschinen
- 2.4 Client/Server-Strukturen (Microkernel)
- 2.5 Zusammenfassung

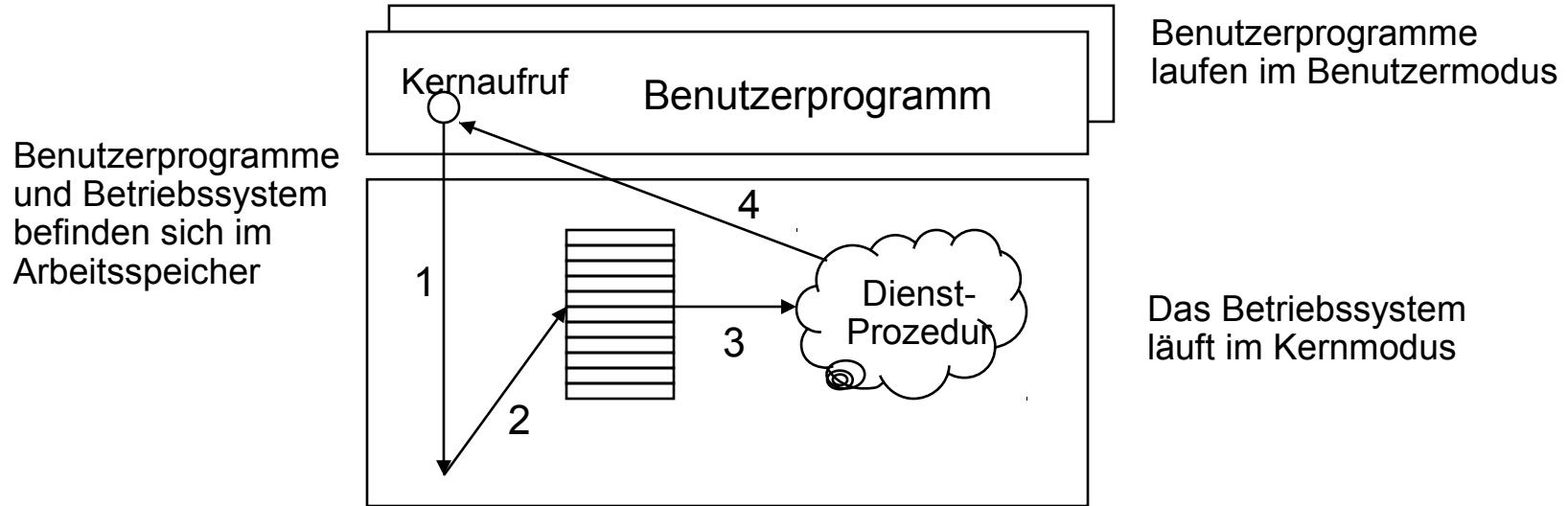
2.1. Monolithische Systeme



Vorwiegende Struktur aller kommerziellen Betriebssysteme:
z.B. UNIX



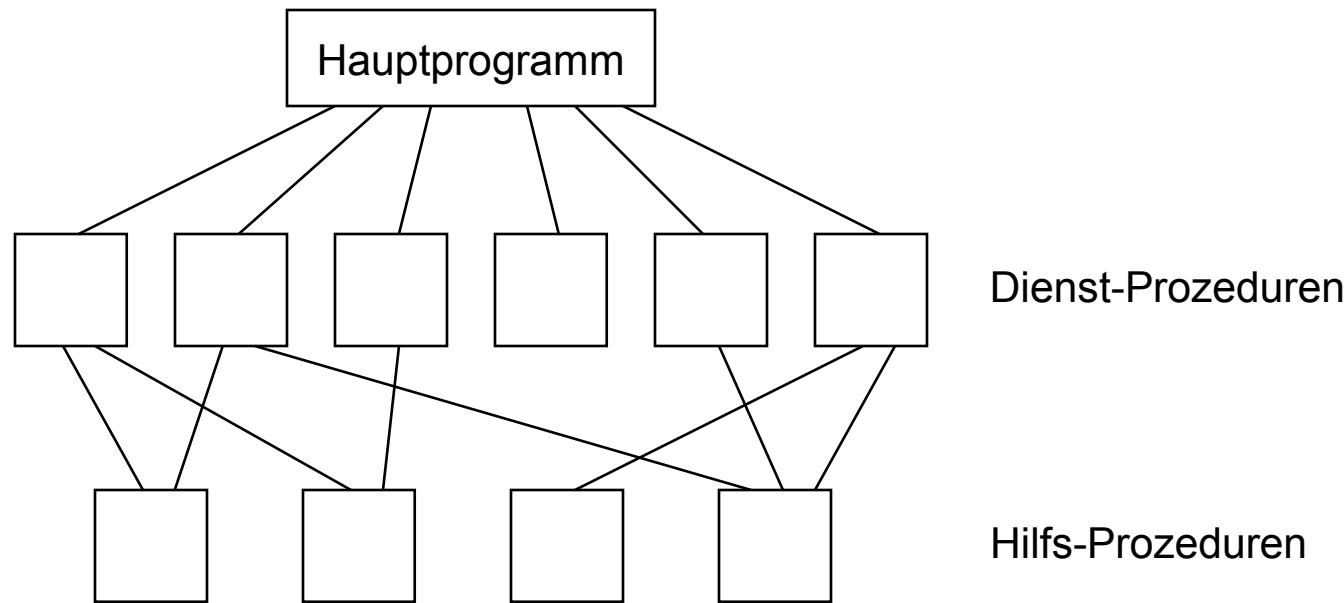
Durchführung eines Kernauftrufs



- 1: Benutzerprogramm springt über TRAP in den Kern und führt den Code selbst aus.
- 2: BS Code bestimmt die Nummer des angeforderten Dienstes.
- 3: BS Code lokalisiert Prozedur-Code für Systemaufruf und ruft sie auf.
- 4: Kontrolle wird an das Benutzerprogramm zurückgegeben.

Wichtig: Kern selbst ist passiv (Menge von Datenstrukturen und Prozeduren)

Innere Struktur eines monolithischen BS:



*Da der Betriebssystemkern passiv ist
und der Code aus einer Menge von Prozeduren besteht,
heißt ein solches Betriebssystem auch prozedurorientiert.*

Der UNIX Betriebssystemkern:

- monolithisch, aber portierbar
- Beispiel: 4.3BSD UNIX Kern (1987)
 - Lines of Code: 116 470 (nicht mehr !)
 - C - Anteil: 97.1 %
 - maschinenunabhängig: 41.5 %
 - maschinenabhängig: 58.5 %
 - davon Gerätetreiber: 35.5 %
 - Netzwerktreiber: 14.8 %

Neben dem Betriebssystemkern wird ein Großteil der UNIX-Systemfunktionalität durch sogenannte Dämon-Prozesse erbracht.

Vergleich Kernel:	SLOC (ohne Leerzeilen , ohne Kommentarzeilen)
Linux 1.0.0 (1994)	176.250
Linux 2.2.0 (1999)	1.800.847
Linux 2.6.0 (2003)	5.929.913
Linux 3.2 (2012)	14.998.651
Windows Server 2003 (Gesamtsystem)	ca. 50 Mio

(Vergleich zu sonstigen Codegrößen)



Project	No. of Files	eLOC
Linux Kernel 2.6.17	15,995	4,142,481
Firefox 1.5.0.2	10,970	2,172,520
MySQL 5.0.25	1973	894,768
PHP 5.1.6	1316	479,892
Apache Http 2.0.x	275	89,967

http://msquaredtechnologies.com/m2rsm/rsm_software_project_metrics.htm

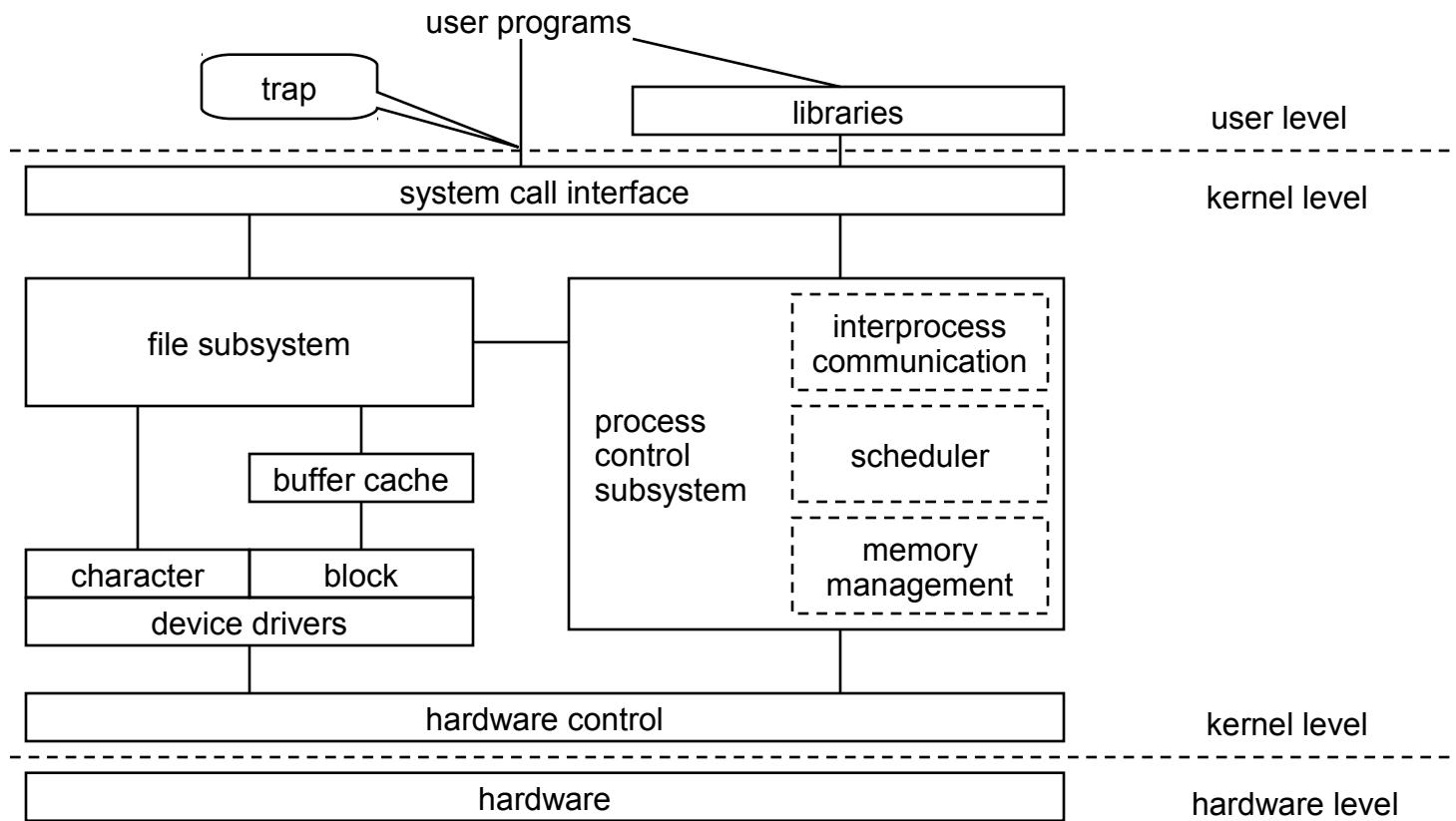
The effective lines of code (eLOC) are measured using the following method:

1. Get the number of lines of code
2. Subtract whitespace lines
3. Subtract comment lines
4. Subtract the lines that contains only block constructs

Beispiel: UNIX (2)



Blockdiagramm des Systemkerns:



aus [Bach]: The Design of the UNIX Operating System

Verallgemeinerung des monolithisches Ansatzes:

- BS als Hierarchie von Schichten (engl. layers).
- Jede Schicht abstrahiert von gewissen Restriktionen der darunterliegenden Schicht. Schicht benutzt Dienste der darunterliegenden Schicht.
- Erstes System: THE (Techn. Hochschule Eindhoven, Dijkstra, 1968, einfaches Stapelverarbeitungssystem in Pascal).

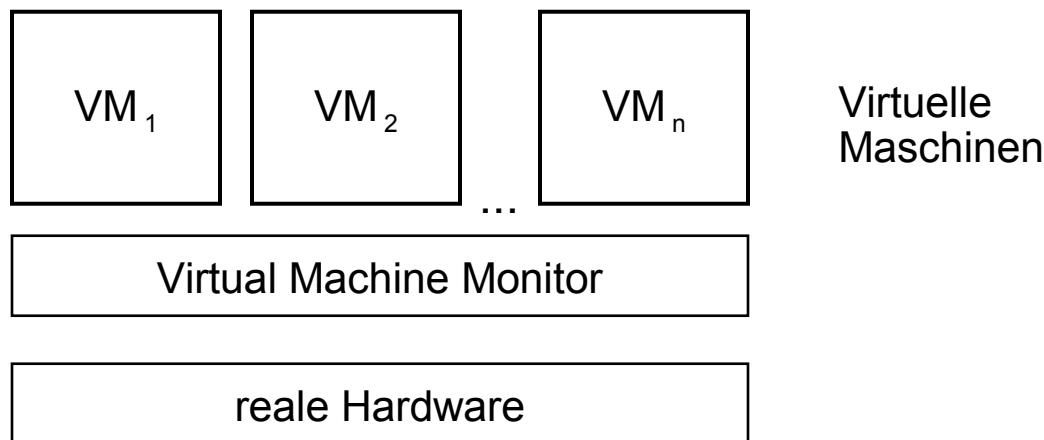
Schicht 5	Operateur
Schicht 4	Benutzerprogramme
Schicht 3	Ein- / Ausgabeverwaltung
Schicht 2	Operateur-zu-Prozess-Kommunikation
Schicht 1	Speicher- und Trommelverwaltung
Schicht 0	Prozessorvergabe und Multiprogramming

- Weitere Verallgemeinerung in MULTICS: "konzentrische (Schutz-) Ringe", verbunden mit nach innen zunehmender Privilegierung, kontrollierter Aufruf zwischen den Ebenen zur Laufzeit.

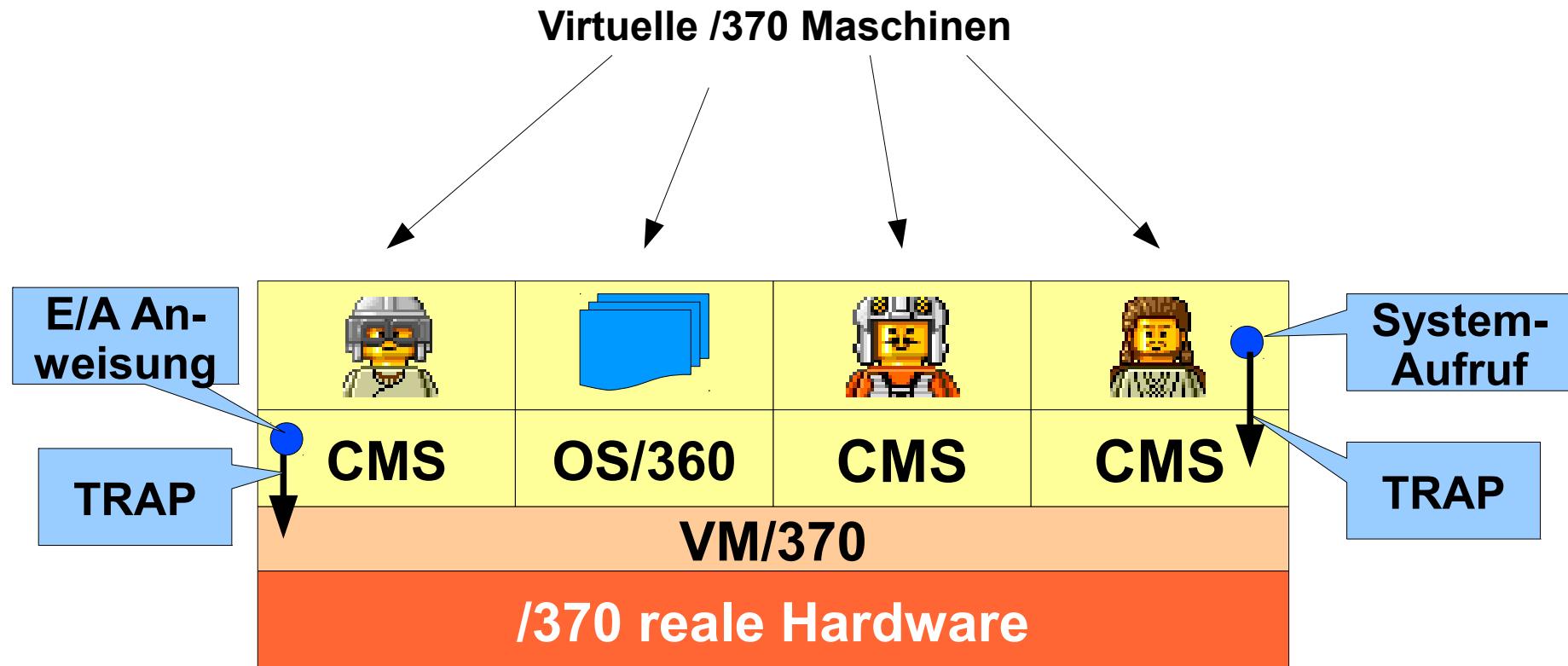
2.3. Virtuelle Maschinen



- Trennen der Funktionen Mehrprogrammbetrieb und erweiterte Maschine
- Virtualisierung durch "Virtual Machine Monitor":
virtuelle Maschinen als mehr oder weniger identische Kopien der
unterliegenden Hardware
- In jeder virtuellen Maschine: übliches Betriebssystem.



Beispiel: VM/370

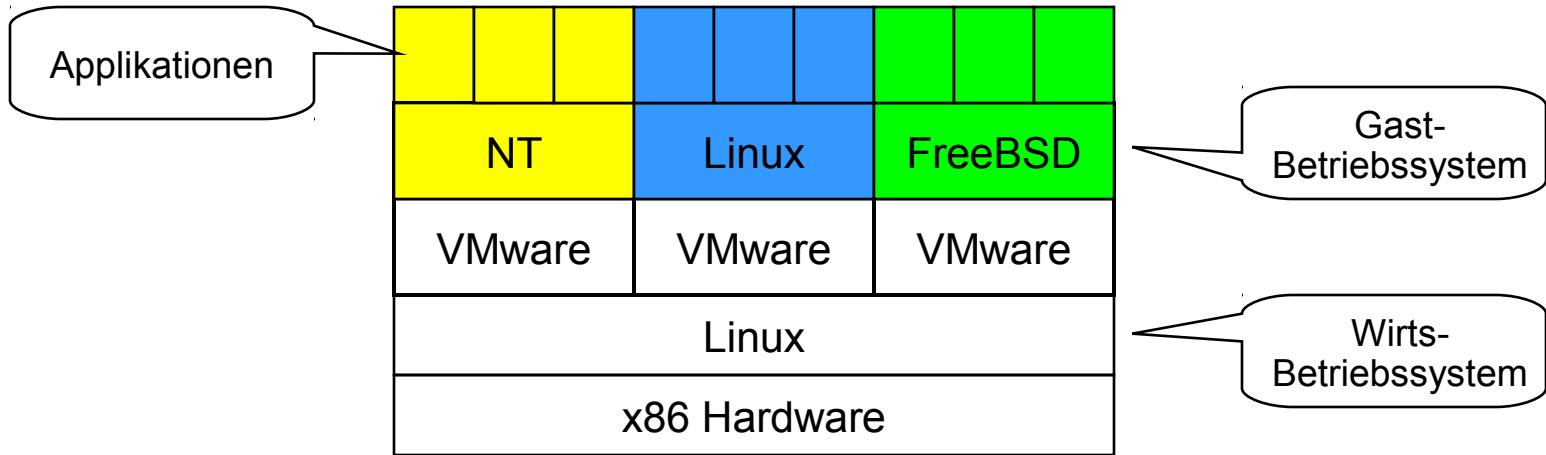


Beispiel: VM/370 (2)



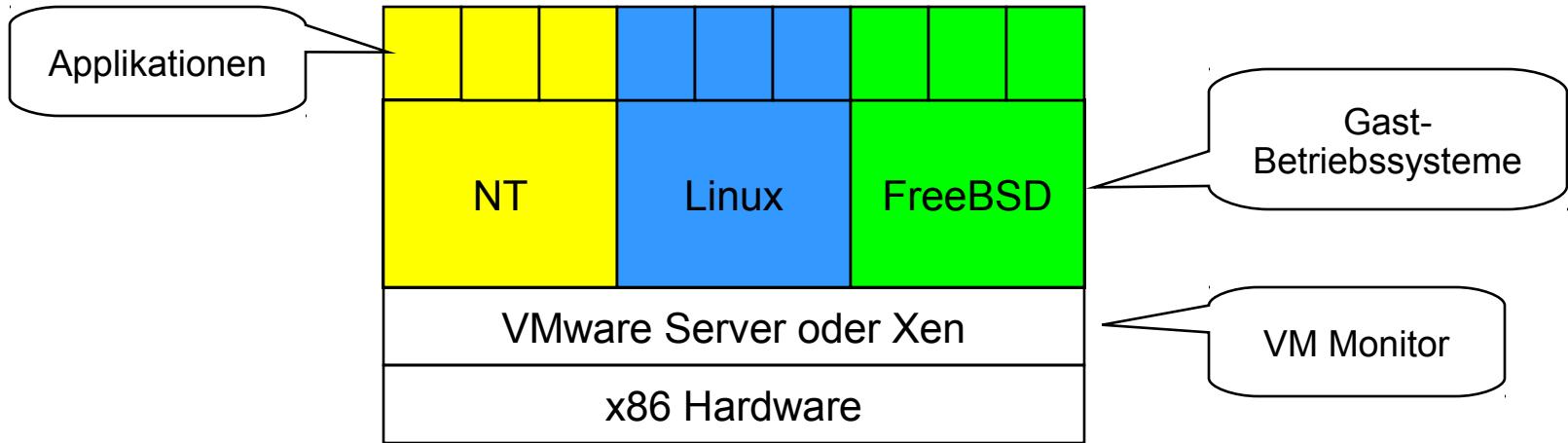
- Das offizielle IBM-Produkt für Tmesharing-Betrieb der /360, TSS/360, kam zu spät, war zu groß und zu langsam.
- In der Zwischenzeit: IBM Scientific Center Cambridge, Mass. Eigenentwicklung, wurde als Produkt (ursprünglich CP/CMS) akzeptiert, erlangte als VM/370 weite Verbreitung.
- Unterste Ebene: virtuelle Maschinen als identische Kopien der unterliegenden Hardware mit Nachbildung von Benutzer/Supervisor-Modi, I/O, Unterbrechungen, (Simulation mehrerer /370 Rechner).
- Betriebssysteme in virtuellen Maschinen:
z.B. ein Stapelverarbeitungssystem (OS/360) und eine Menge von Einbenutzer-Dialogsystemen (CMS, Conversational Monitor System) gleichzeitig möglich.
- Heute: z/VM: erlaubt z.B. 100 unabhängige Linux-Systeme auf einem IBM Mainframe.

Beispiel: VMware Workstation



- erlaubt beliebige Betriebssysteme für x86-Architektur auf Linux oder Windows
- Jedes Gastbetriebssystem kann abstürzen, ohne den Rest zu beeinflussen

Beispiele: VMware Server, Xen



- Xen: Paravirtualisierung: Gastsysteme müssen angepasst werden (Quellcode Voraussetzung)
- VMware Server: klassischer VM Monitor

2.4. Client/Server-Struktur (μ kern)



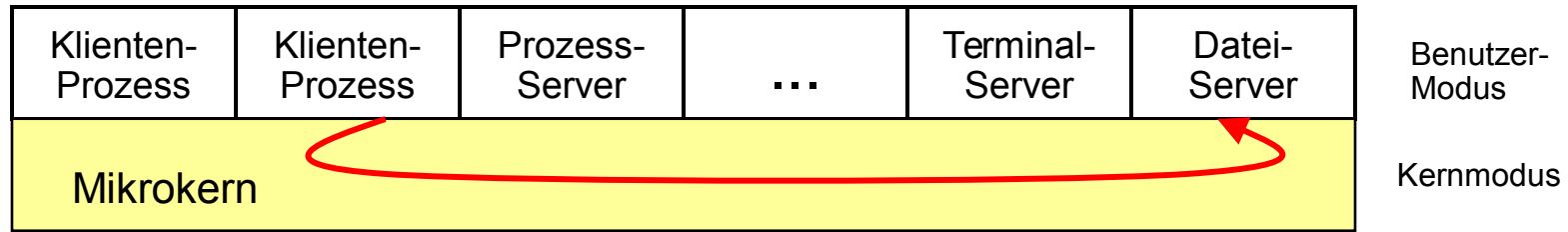
- Problem monolithischer Systeme: Kernelcode wird immer umfangreicher und komplexer, damit zwangsläufig auch fehlerträchtiger (z.B. Linux 2.6.x: ca. 5.7 Mio Zeilen)
- Aller Code, der im privilegierten Modus läuft, hat Zugriff auf alle Betriebsmittel und zählt damit immer zur „Trusted Code Base“.
- Nicht alle Anwendungen benötigen wirklich alle Dienste, die ein Kernel anbietet
- Art und Anzahl der Dienste werden aber durch den Kernel vorgegeben
- **Mikrokern**-Ansatz: Alle Funktionen, die für ihre Funktion nicht im privilegierten Modus arbeiten müssen, werden aus dem Kernel ausgelagert

Mikrokern-Ansatz:

- Dienste wie Dateisystem, Netzwerkprotokolle, Speicherverwaltung, Prozesssteuerung, sogar Gerätetreiber müssen nicht zwangsläufig im Kernel angesiedelt sein.
- „Server“-Prozesse, die wie Anwenderprogramme ohne besondere Privilegien arbeiten, übernehmen diese Aufgaben
- Prinzip: Trennung von Strategien und Mechanismen (*separation of policy and mechanism*)
- Der verbleibende Mikrokern bietet nur noch Dienste zur Kommunikation der Server untereinander an.
- Er *sollte* daher wesentlich weniger komplex sein → kleinere, bzw. feingranularere „Trusted Code Base“

Ansatz für moderne Betriebssysteme:

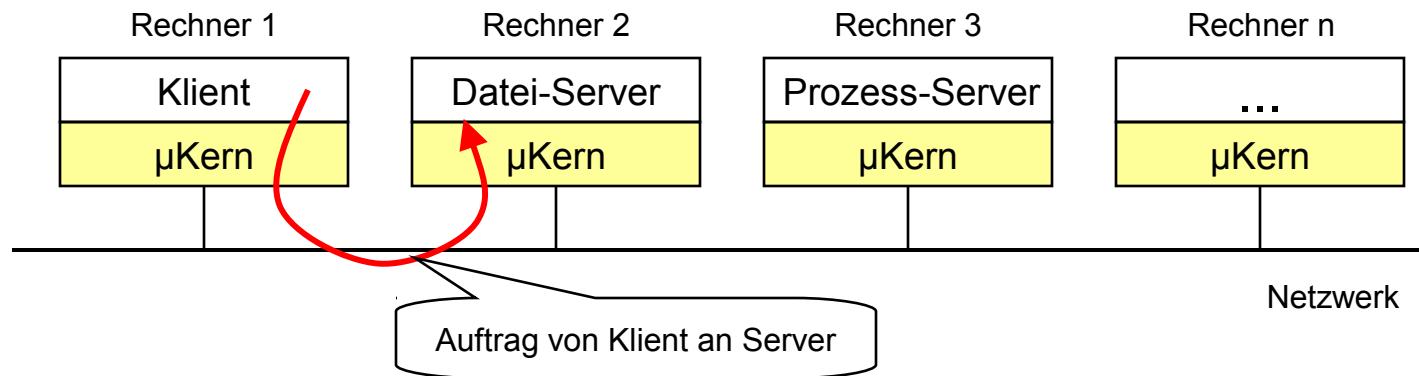
- Auslagerung großer Teile der Funktionalität eines BS-Kerns in Benutzerprogramme (=lauffähig im Benutzermodus).
- Übrig bleibt minimaler BS-Kern, als Mikrokern bezeichnet (= "Infrastruktur").



- Clients erhalten einen Dienst, indem sie Nachrichten an einen Serverprozess senden
- Mehrere Server können ihre Dienste in verschiedener, auf das jeweilige Ziel zugeschnittener Form anbieten
- Die Betriebssystemschnittstelle ist die Menge der Dienste, die ein Klient nutzt
- Jeder Klient kann „seine“ Schnittstelle selbst definieren
- Wie bei Virtualisierung sind mehrere BS-Schnittstellen in einem System möglich

Vorteile:

- Isolation einzelner "Systemteile" gegeneinander (Vermeidung von Fehlerausbreitung).
- Erweiterbarkeit, Anpassungsfähigkeit und flexible Konfigurierbarkeit insbesondere für Verteilte Systeme.



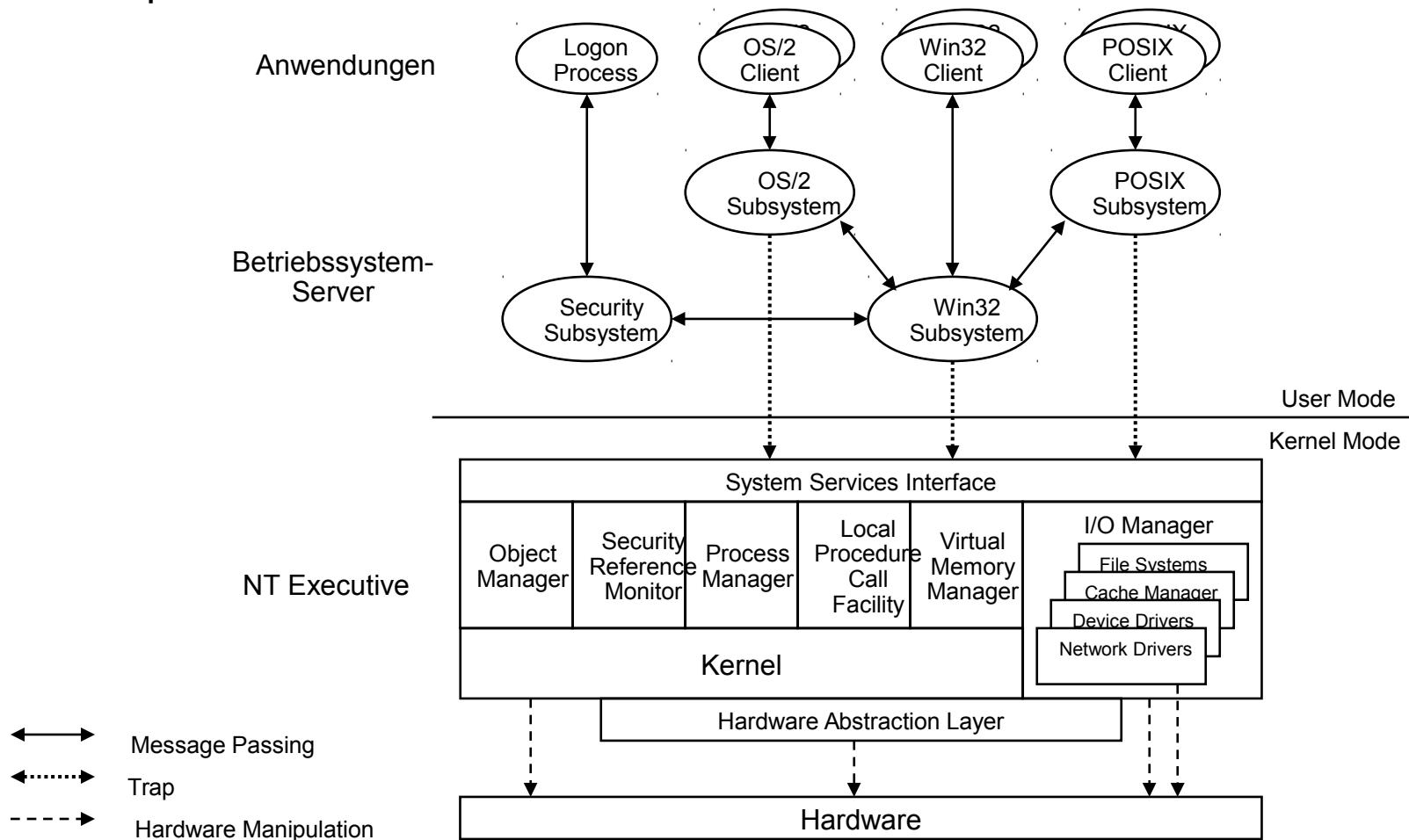
Beispiele: Mach, Chorus, Amoeba, Windows NT

Ein Betriebssystem, das auf einem über Nachrichten realisierten Beauftragungsprinzip beruht, heißt auch nachrichtenorientiert. Nachrichtenorientiertheit und Prozedurorientiertheit sind funktional gleichwertig, nachrichtenorientierte Systeme leiden aber häufig unter Ineffizienz.

Client/Server-Struktur (5)



Beispiel: Windows NT



1. Grundverständnis einer Betriebssystemschnittstelle
2. Strukturierungsprinzipien von Betriebssystemen:
 - Monolithische Struktur
 - Hierarchie von Schichten
 - Virtuelle Maschinen
 - Client/Server-Struktur (Microkernel)

Kap. 3: Prozesse und Threads

3.1 Prozessmodell

3.2 Implementierung von Prozessen

3.3 Threads

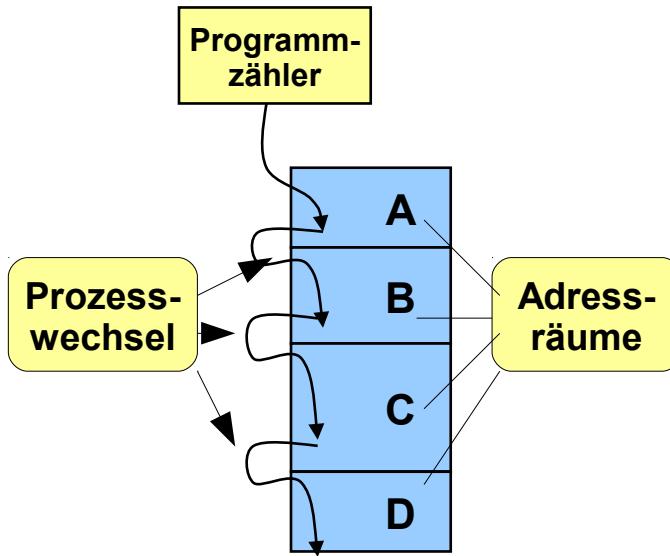
3.4 Zusammenfassung

Konzept des sequentiellen Prozesses:

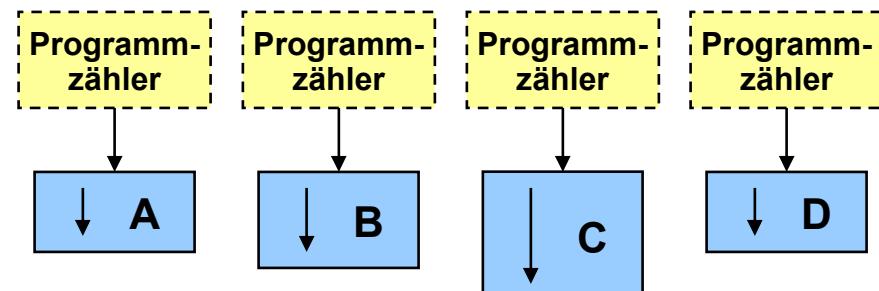
Def

- Def: Ein Prozess ist ein sich in Ausführung befindliches Programm (einschl. seiner aktuellen Werte des Programmzählers, der Register, Speichervariablen, Stack).
- Ein Prozess besitzt einen privaten Adressraum
 - Menge von (virtuellen) Adressen, von Prozess zugreifbar
 - Programm und Daten in Adressraum sichtbar.
- Verhältnis Prozess - Prozessor
 - Prozess besitzt konzeptionell einen eigenen virtuellen Prozessor
 - reale(r) Prozessor(en) zwischen den virtuellen Prozessoren umgeschaltet (Mehrprogrammbetrieb).
 - Umschaltungseinheit heißt Scheduler oder Dispatcher, Scheduling-Algorithmus legt Regeln fest.
 - Umschaltungsvorgang heißt Prozesswechsel oder Kontextwechsel (Context Switch).
 - Multicore-Prozessoren enthalten mehrere Prozessoren auf einem Chip

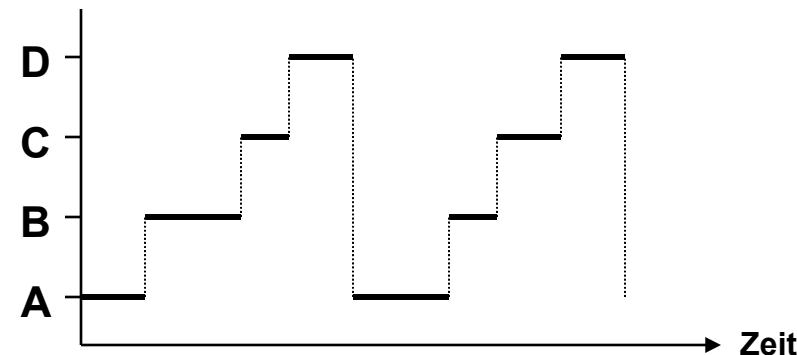
(a) Mehrprogrammbetrieb mit 4 Programmen



(b) Konzeptionelles Modell:
4 unabhängige, sequenzielle Prozesse



(c) Gantt-Diagramm



nur ein Programm ist zu jedem Zeitpunkt aktiv



Beachte:

- Programm:
 - statische Beschreibung eines sequentiellen Algorithmus
 - dasselbe Programm kann mehrmals (auch gleichzeitig!) innerhalb verschiedener Prozesse ausgeführt werden.
- Prozess:
 - Ausführungsgeschwindigkeit eines Prozesses ist nicht gleichmäßig und nicht reproduzierbar.
 - ⇒ Bei der Programmierung sind keine a-priori-Annahmen über den zeitlichen Verlauf zulässig.
 - ⇒ Bei zeitkritische Anforderungen, z.B. Realzeit-System, sind besondere Vorkehrungen im Scheduling-Algorithmus notwendig.

- In einfachen Systemen (z.B. Gerätesteuerung):
 - Menge der zur Laufzeit existierenden Prozesse häufig statisch
 - wird bei Systemstart erzeugt.
 - Allg. Fall:
 - Mechanismus notwendig, um Prozesse dynamisch (zur Laufzeit) durch andere Prozesse erzeugen zu können.
- Def** →
- Erzeugender Prozess heißt Elternprozess (*parent process*), erzeugte Prozesse heißen Kindprozesse (*child processes*).
 - Wiederholte Prozesserzeugung durch Kindprozesse
⇒ baumartig strukturierte Prozessmenge.
 - Prozess mit all seinen direkten und indirekten Nachfahren heißt Prozessfamilie.
 - Jede baumartig strukturierte Menge von Prozessen heißt Prozesshierarchie (ohne notwendigerweise durch Prozesserzeugung aus der Wurzel entstanden zu sein).

Motivation:

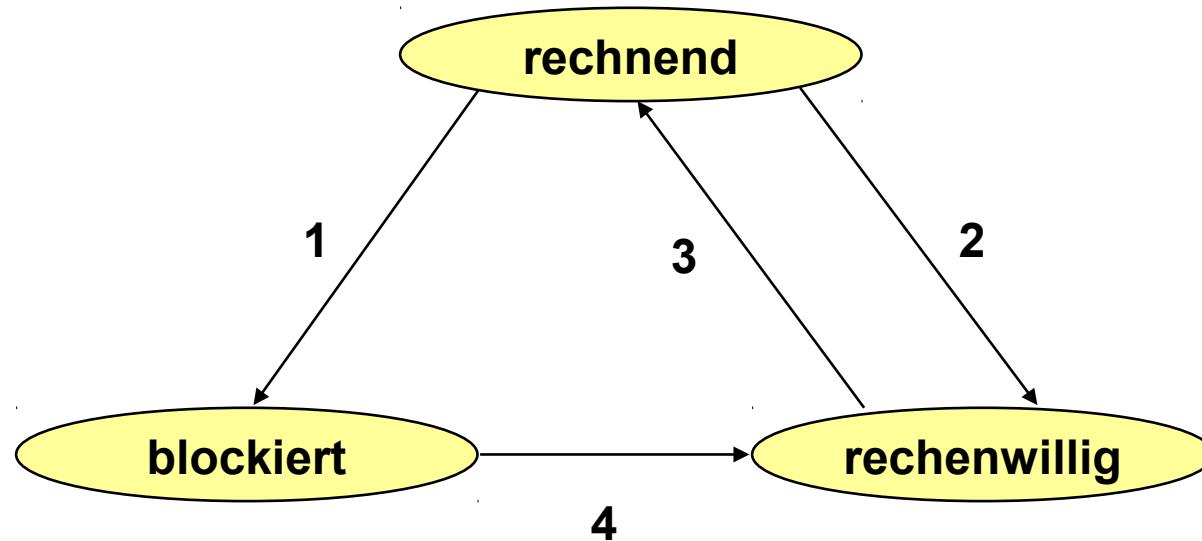
- Prozesse, obwohl unabhängige Einheiten, können aufgrund des Algorithmus logisch voneinander abhängig sein:
Beispiel
 UNIX shell: `cat datei1 datei2 datei3 | grep egon`
- In Abhängigkeit von den relativen Ausführungsgeschwindigkeiten kann ein Prozess warten müssen, bis eine Eingabe vorliegt.
- Allgemeiner sagt man:
 Er blockiert und wartet auf ein (für ihn externes) Ereignis.
- Einprozessor-System: Prozessor wird dann unmittelbar einem anderen Prozess zugeordnet. Entzug des Prozessors (Suspendierung) in diesem Fall problembegründet.
- Auch möglich: Scheduler entscheidet auf Prozesswechsel, obwohl der erste Prozess weiter ausgeführt werden könnte (Preemption).

Damit sinnvolle Prozesszustände:

- rechnend (oder aktiv): dem Prozess ist ein Prozessor zugeordnet, der das Programm vorantreibt.
- rechenwillig (oder bereit): Prozess ist ausführbar, aber Prozessor ist anderem Prozess zugeordnet (bzw. alle verfügbaren Prozessoren sind anderen Prozessen zugeordnet).
- blockiert (oder schlafend): Prozess wartet auf Ereignis. Er kann solange nicht ausgeführt werden, bis das Ereignis eintritt.

Gelegentlich noch folgende Zustände:

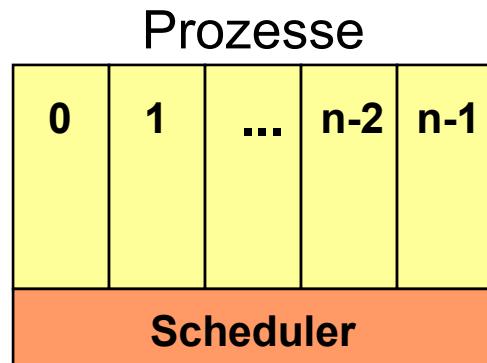
- initiiert: in Vorbereitung (Anfangszustand).
- terminiert: Prozess ist beendet (Endzustand).



Zustandsübergänge:

- 1 $\text{rechnend} \rightarrow \text{blockiert}$: Versetzung in den Wartezustand, (Warten auf Ereignis).
- 2 $\text{rechnend} \rightarrow \text{rechenwillig}$: Scheduler entzieht den Prozessor.
- 3 $\text{rechenwillig} \rightarrow \text{rechnend}$: Scheduler teilt Prozessor zu.
- 4 $\text{blockiert} \rightarrow \text{rechenwillig}$: Ereignis tritt ein.

- Das Prozessmodell vereinfacht die Beschreibung der Aktivität des Rechensystems.
- Die ineinander verwobene Aktivität des Systems wird durch eine Menge von sequentiellen Prozessen beschrieben.
- Die unterste Schicht eines Betriebssystems behandelt die Unterbrechungen und ist für das Scheduling verantwortlich. Der Rest des Systems besteht aus sequentiellen Prozessen.



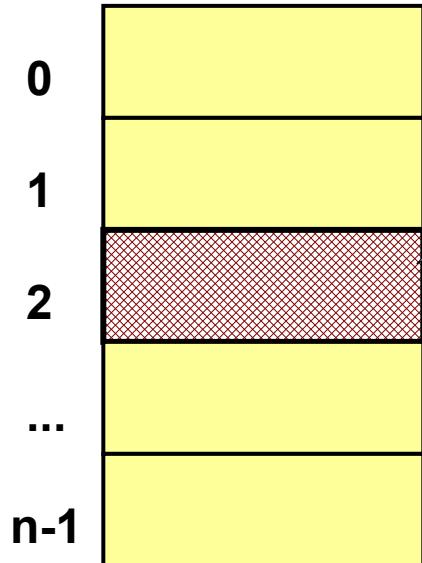
- Mechanismen zur Synchronisation und Kommunikation von Prozessen sind notwendig → Kap. 5
- Programmierung von Anwendungen aus mehreren nebenläufigen Prozessen heißt Concurrent Programming.
- Klass. Prozessmodell verfeinert durch Einführung sog. Leichtgewichtsprozesse (light weight processes, oder Threads = Fäden), die mehrere Aktivitätsträger in einem einzigen Adressraum darstellen → 3.3 .

3.2. Implementierung von Prozessen



Datenstrukturen im BS-Kern zur Prozessverwaltung:

Prozesstabelle
(process table)



Prozesskontrollblock
process control block
(PCB)
task control block (TCB)

PCB dient zum Speichern des gesamten
Zustandes eines Prozesses

Typische Felder in einem Prozesskontrollblock:

Prozessverwaltung

Register
Programmzähler
Programmstatuswort
Stack-Zeiger
Prozesszustand
Prozessnummer
Elternprozessnummer
Prozesserzeugungszeitpunkt
Terminierungsstatus
verbrauchte Prozessorzeit
Prozessorzeit der Kinder
Alarm-Zeitpunkt
Signalstatus
Signalmaske
unbearbeitete Signale
Zeiger auf Nachrichten
verschiedene Flags

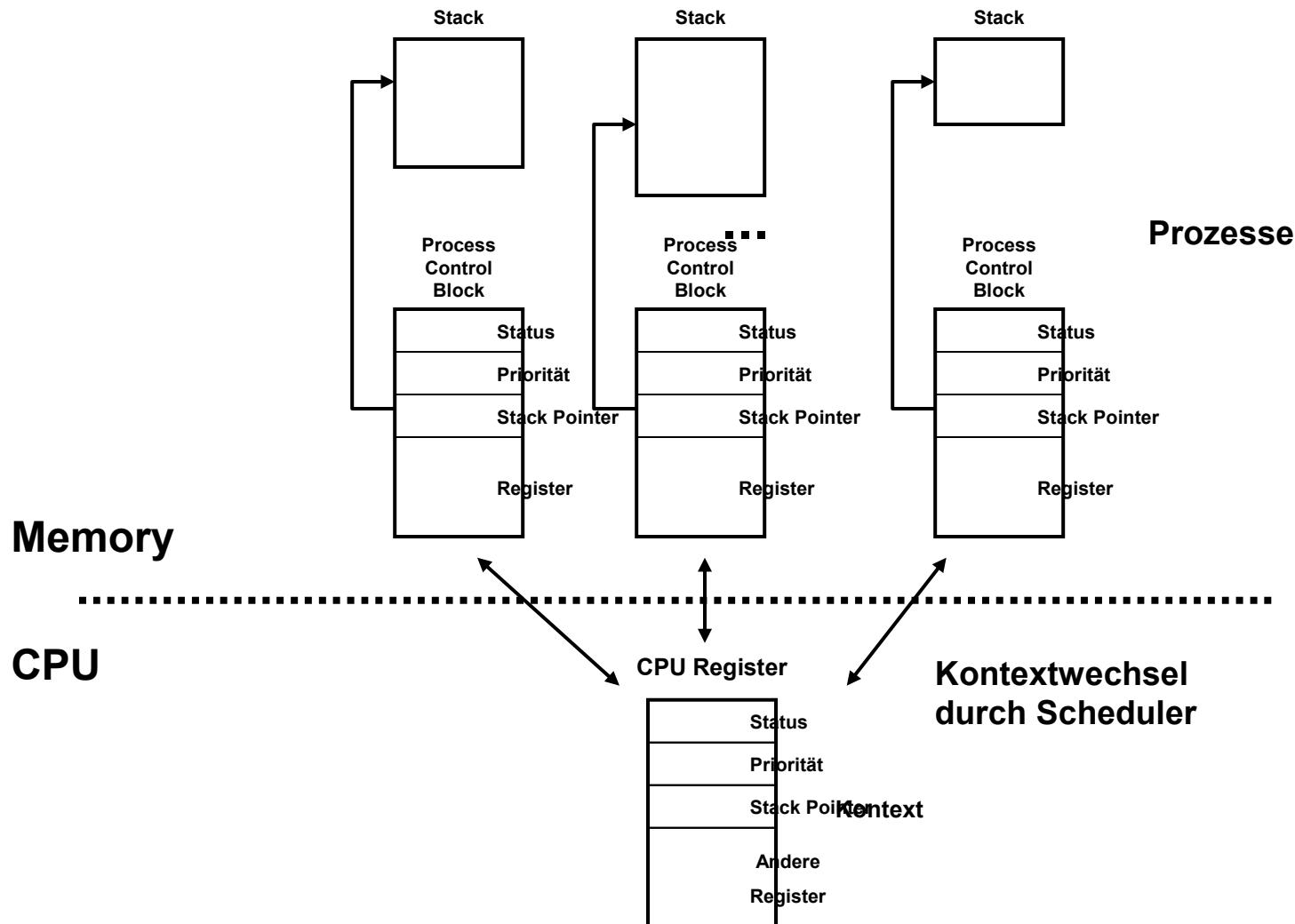
Speicherverwaltung

Zeiger auf Textsegment
Zeiger auf Datensegment
Zeiger auf BSS-Segment
Prozessgruppe
reale UID
effektive UID
reale GID
effektive GID
verschiedene Flags

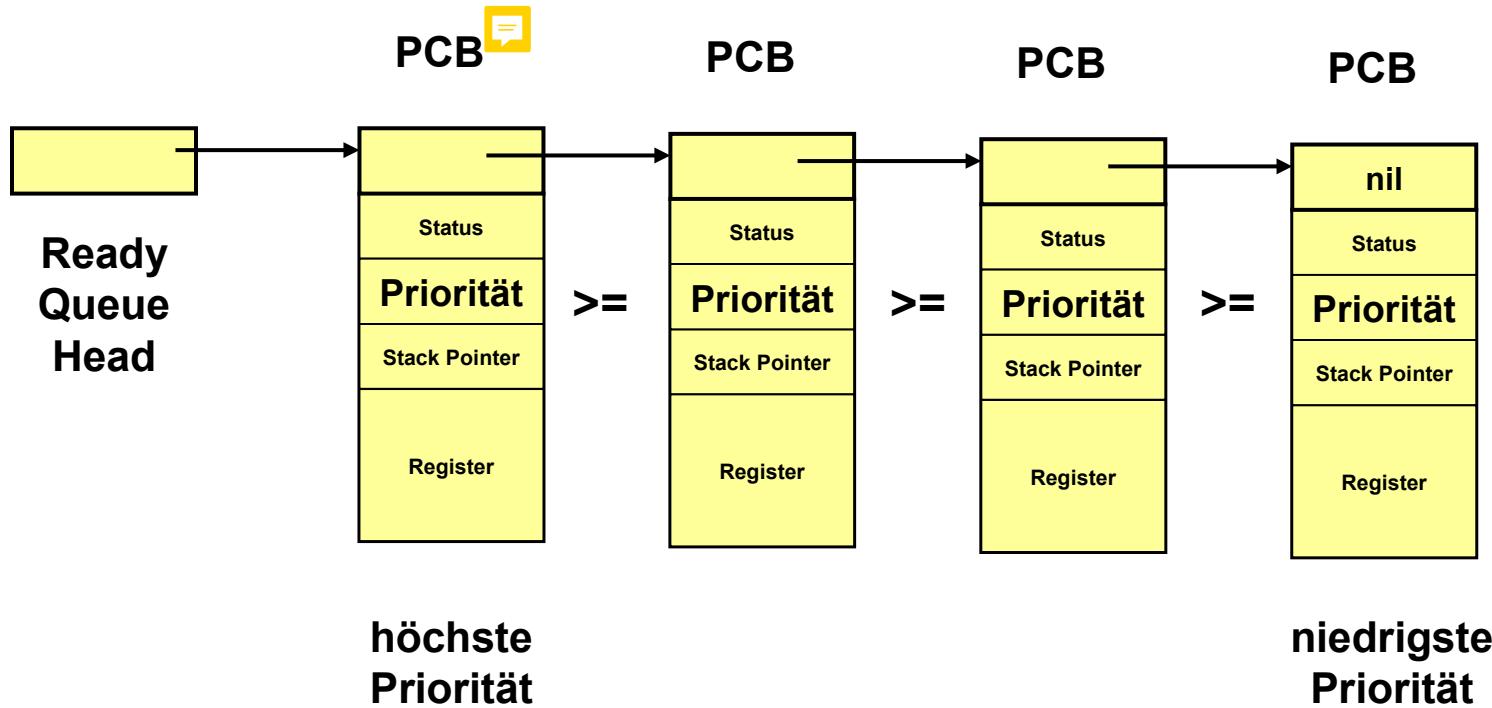
Dateisystem

Wurzelverzeichnis
aktuelles Verzeichnis
UMASK-Maske
offene Dateideskriptoren
effektive UID
effektive GID
Aufrufparameter
verschiedene Flags

zusätzlich: Zeiger zur Verkettung des PCB in Warteschlangen

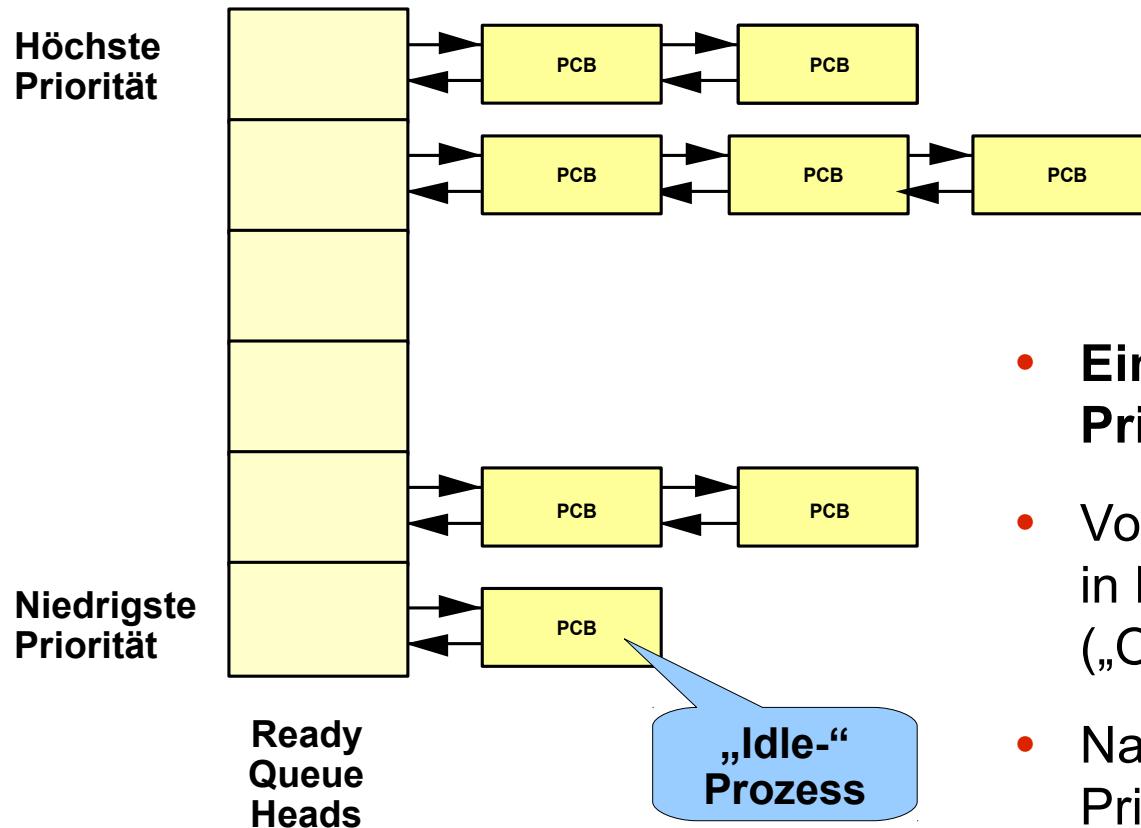


Einfache Struktur der Liste der rechenwilligen Prozesse
(Bereit-Liste oder Ready Queue):

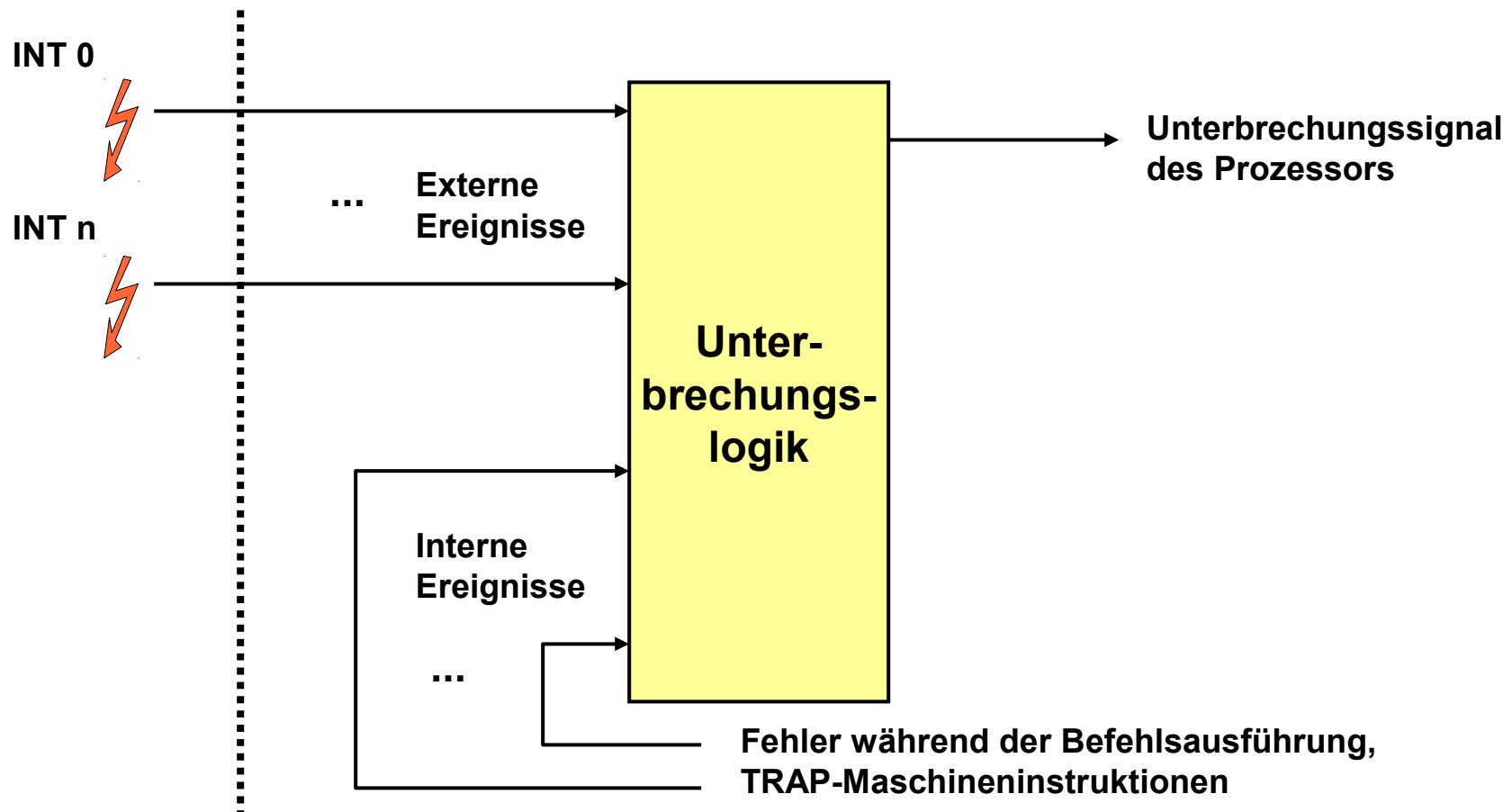


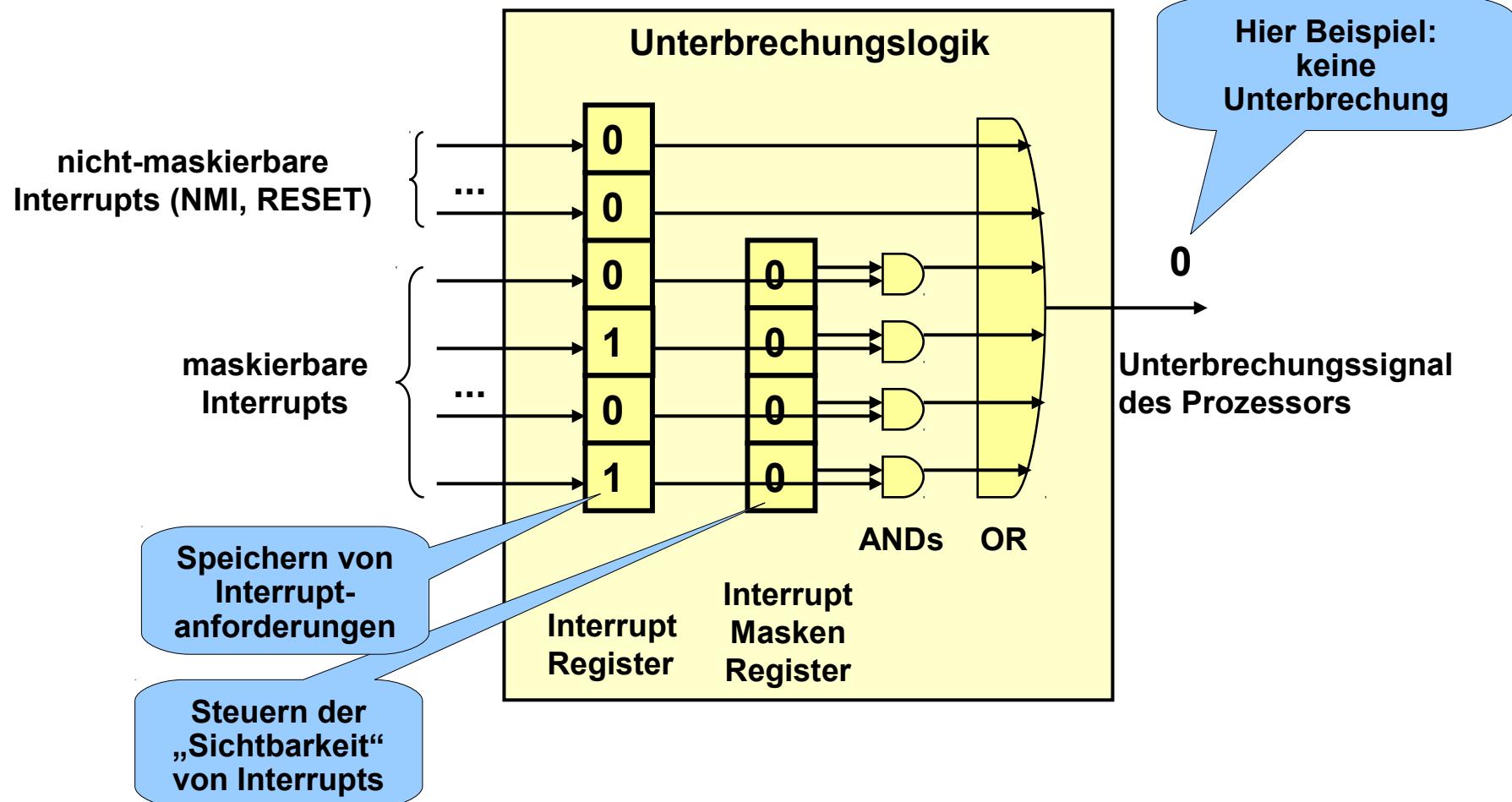
Bei gleicher Priorität: Einreihen nach „first in / first out“
==> Nachteil: Laufzeit abhängig von Prozessanzahl

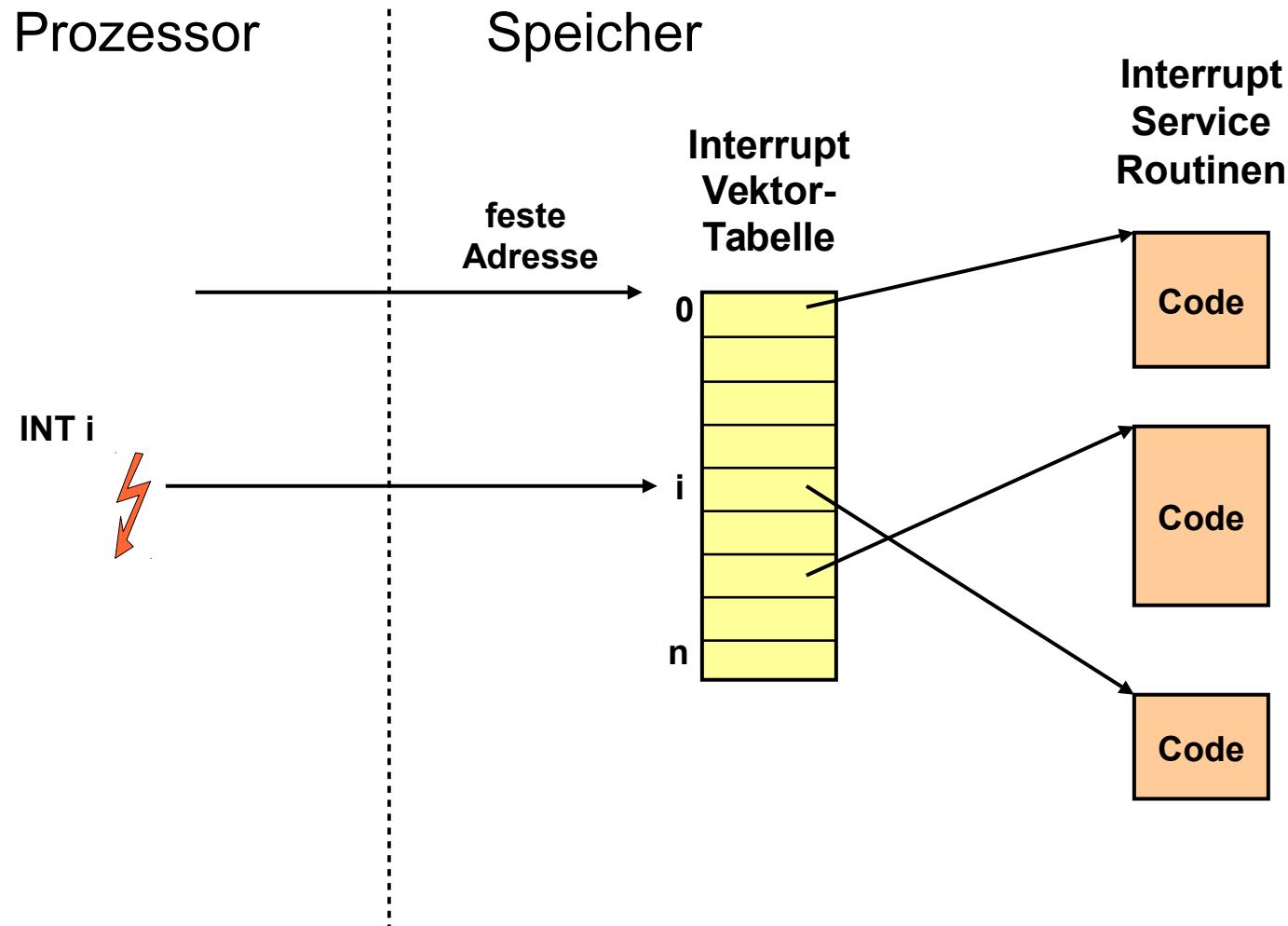
Typische Struktur der Liste der rechenwilligen Prozesse
(Bereit-Liste oder Ready Queue):



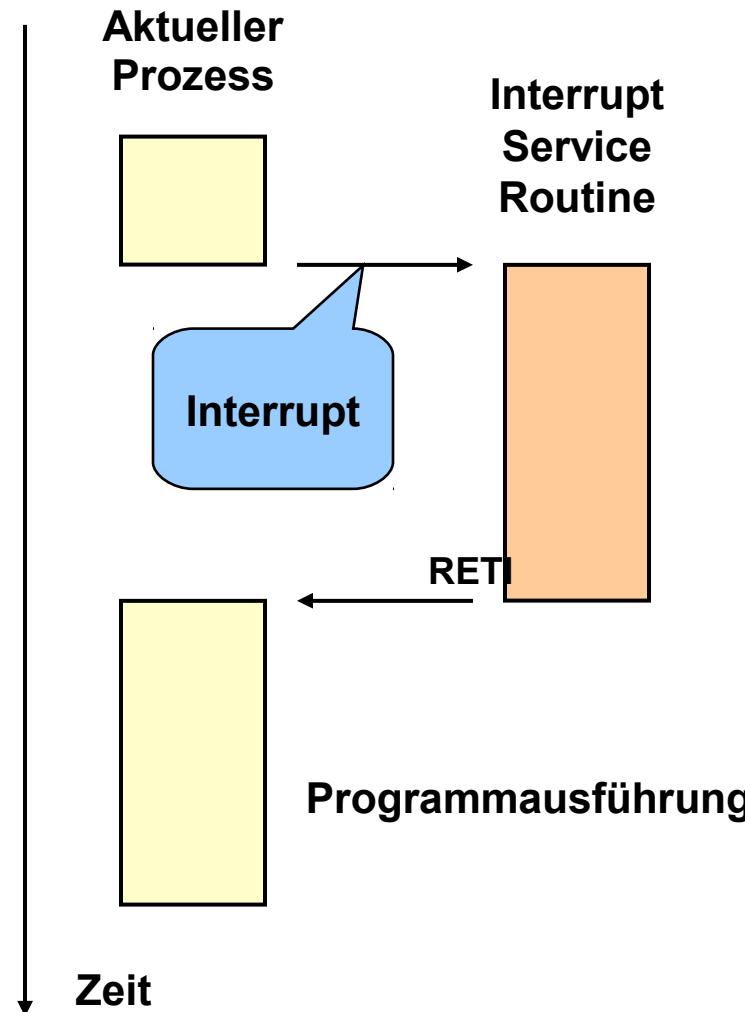
- **Eine Ready-Queue pro Prioritätsstufe**
- Vorteil: Schnelles Einreihen in konstanter Laufzeit („O(1)-Scheduler“)
- Nachteil: Feste Anzahl an Prioritäten







Prinzipieller Ablauf

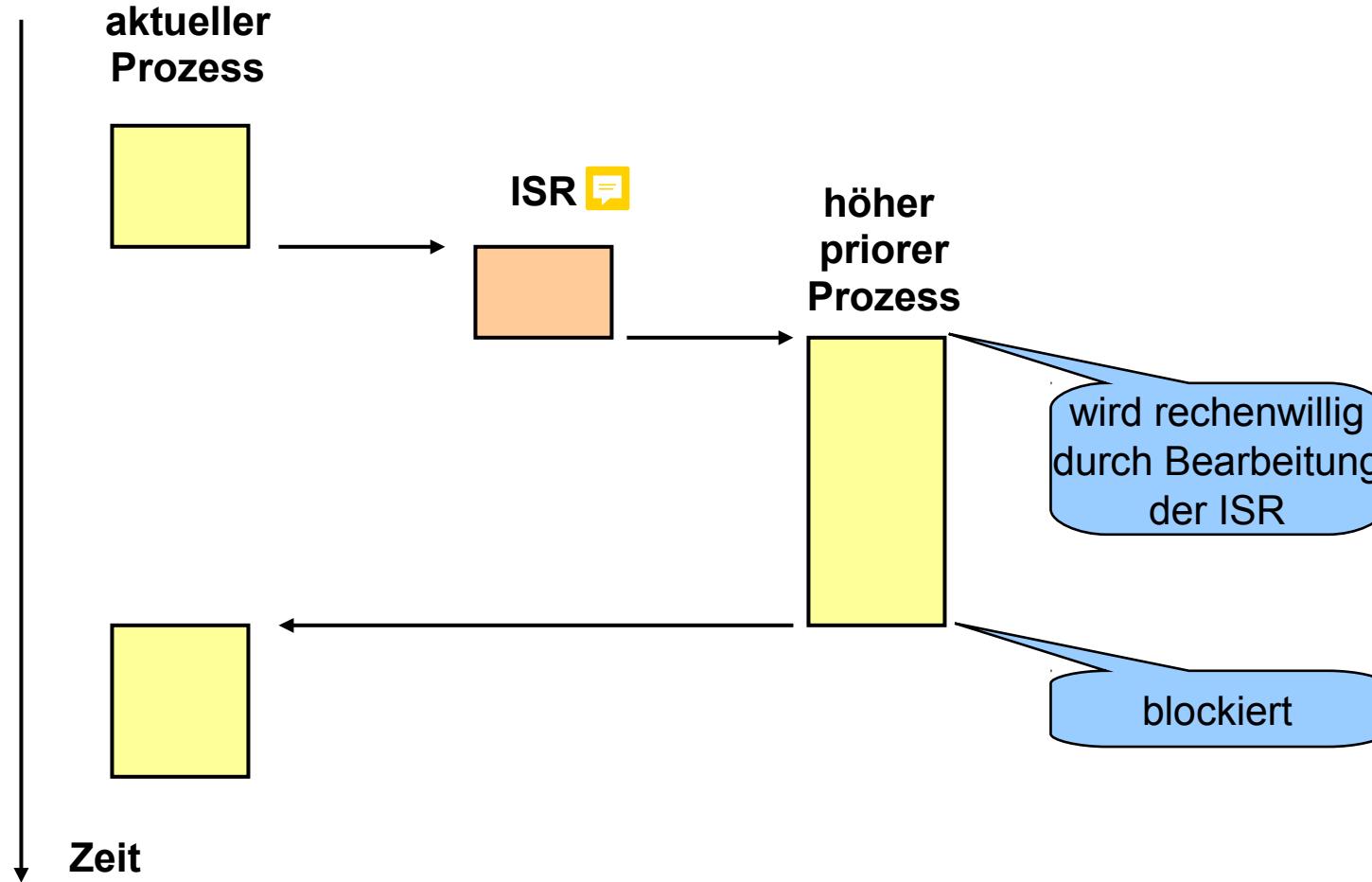




Typische Ausführungsschritte:

1. Programmzähler (u.a.) wird durch Hardware auf dem Stack abgelegt.
2. Hardware lädt den neuen Programmzählerinhalt aus dem Unterbrechungsvektor.
3. Eine Assembler-Routine rettet die Registerinhalte.
4. Eine Assembler-Routine bereitet den neuen Stack vor.
5. Eine C-Prozedur markiert den unterbrochenen Prozess als rechenwillig.
6. Der Scheduler bestimmt den Prozess, der als nächster ausgeführt werden soll.
7. Die C-Prozedur gibt die Kontrolle an die Assembler-Routine zurück.
8. Die Assembler-Routine startet den ausgewählten Prozess.

Damit Umschaltung zwischen Benutzerprogrammen:



Systemaufrufe zur Prozessverwaltung:

`pid_t fork(void)`

Erzeugen einer Kopie des Prozesses (Child), Parent erhält pid des Kindes zurück oder -1 bei Fehler, Kind erhält 0 als Ergebnis.

`int execve(char* name, char* argv[], char* envp[])`

Überlagern des ausgeführten Programms eines Prozesses (Code, Daten, Stack) durch neues Programm. Andere Varianten:

`exec1, execle, execlp, execv, execvp.`

`pid_t getpid(void)`

Rückgabe der eigenen Process Id.

`pid_t getppid(void)`

Rückgabe der Process Id des Elternprozesses.

Systemaufrufe zur Prozessverwaltung (2):

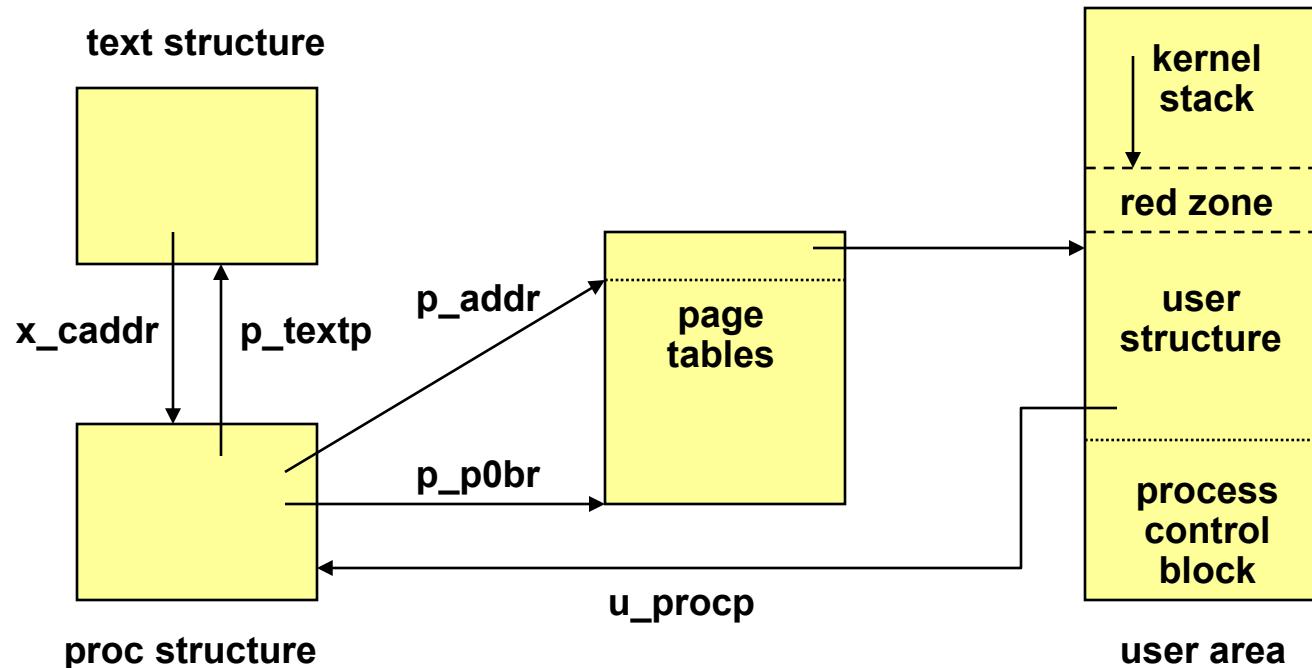
`exit(status)` **Beende den laufenden Prozess und stelle dem Parent den Exit-Status zur Verfügung.**

`pid_t wait(int* status)`
 Warten auf die Beendigung eines Kindprozesses. Dessen Id wird über den Rückgabewert, sein Status über `status` zurückgegeben.

`pid_t waitpid(pid_t pid, int* status, int opts)`
 Warten auf das Ende eines bestimmten Kindprozesses, dessen Id über den Parameter `pid` übergeben wird.

(wird im Praktikum vertieft)

Kerndatenstrukturen zur Repräsentierung eines Prozesses in 4.3BSD UNIX (Überblick):



aus Leffler et al: Design and Implementation of the 4.3BSD UNIX Operating System

proc structure (residenter Teil der Prozessbeschreibung):

Kategorie	Name	Bedeutung
scheduling	p_pri	current priority
	p_userpri	user priority based on p_cpu and p_nice
	p_nice	user-requested scheduling priority
	p_cpu	recent CPU usage
	p_slptime	amount of time sleeping
identifiers	p_pid	unique process identifier
	p_ppid	parent process identifier
	p_uid	user identifier
	p_textp	pointer to description of executable file
memory management	p_P0br	pointer to process page tables
	p_szpt	size of process page tables
	p_addr	location of user area
	p_swaddr	location of user area when swapped out
	p_wchan	event process is awaiting
synchronisation signals	p_sig	mask of signals pending delivery
	p_sigignore	mask of signals being ignored
	p_sigcatch	mask of signals being caught
	p_pgrp	process-group identifier
	p_rusage	pointer to rusage structure
resource accounting	p_quota	pointer to disk quota structure
	p_time	amount of real-time until timer expires
timer management		

aus Leffler et al: Design and Implementation of the 4.3BSD UNIX Operating System

Prozesszustände (process states):

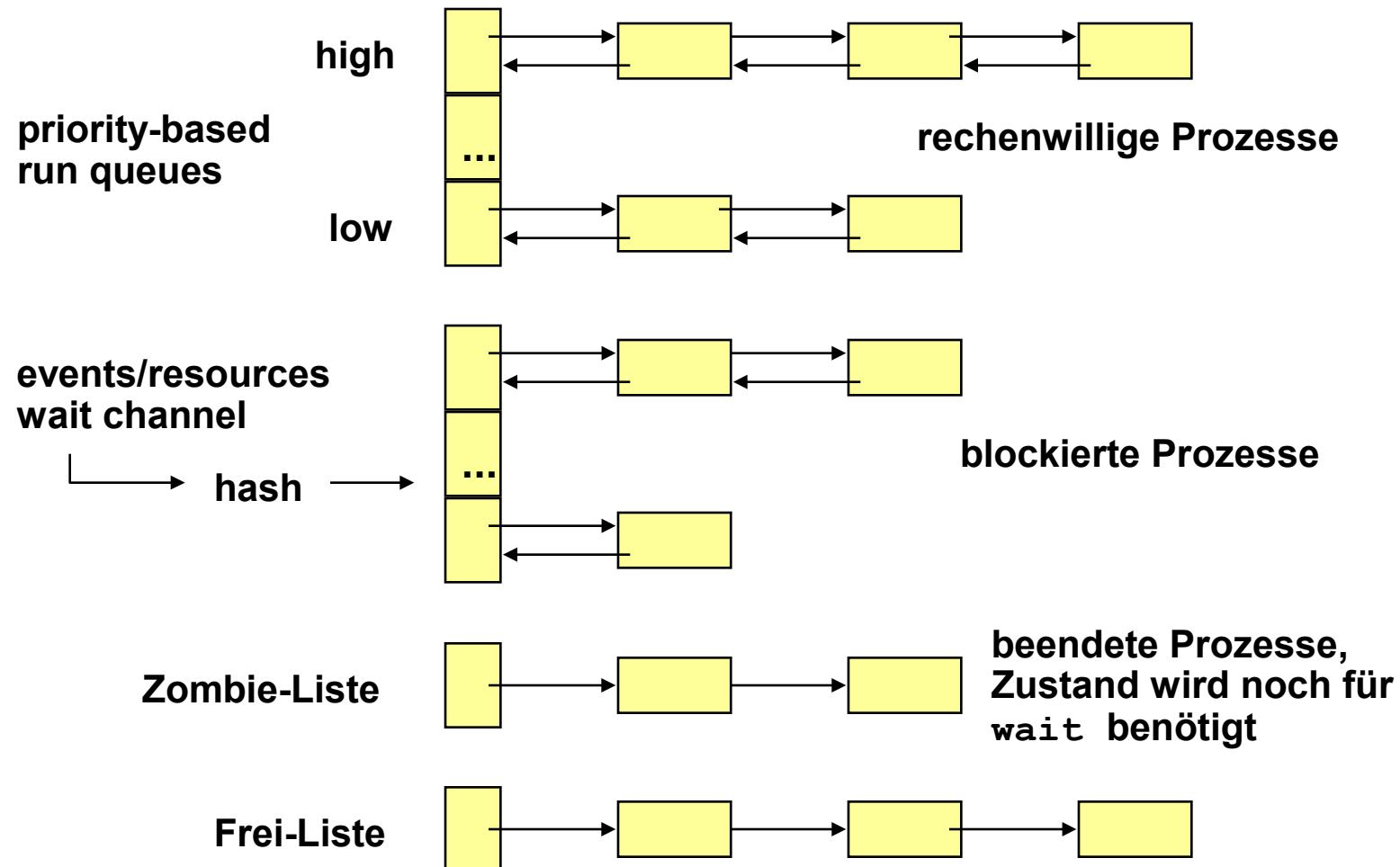
SSLEEP	schlafend = blockiert, wartend auf ein Ereignis
SRUN	runnable = rechenwillig und rechnend
SIDL	initiiert , transienter Zustand während Erzeugung
SZOMB	terminiert , transienter Zustand nach Beendigung
SSTOP	angehalten oder in trace mode

user structure

(auslagerbarer Teil der Prozessbeschreibung):

- **Prozessorzustand (Registerinhalte usw. = eigentlicher PCB, getrennt für user mode und kernel mode).**
- **Zustand bzgl. aktueller Systemaufrufe.**
- **Descriptor-Tabelle (Liste der geöffneten Objekte des Prozesses).**
- **Accounting-Information.**
- **Kernel Stack des Prozesses (Stack für user mode ist im Prozessadressraum).**

Organisation der Warteschlangen:



Wesentliche interne Kern-Routinen zur Prozessumschaltung:

swtch() Aufruf des Schedulers zur Auswahl des rechenwilligen Prozesses mit der höchsten Priorität und darin Aufruf von **resume()**.

resume(...)
Lade Prozessor-Register mit ausgewähltem Prozesskontrollblock und führe diesen Prozess fort.

sleep(wait channel, priority)
Freiwillige Abgabe des Prozessors bei Übergang nach blockiert und Zielpriorität bei Erwachen.

wakeup(wait channel)
Setze alle Prozesse rechenwillig, die auf das Ereignis blockiert warten.

Motivation

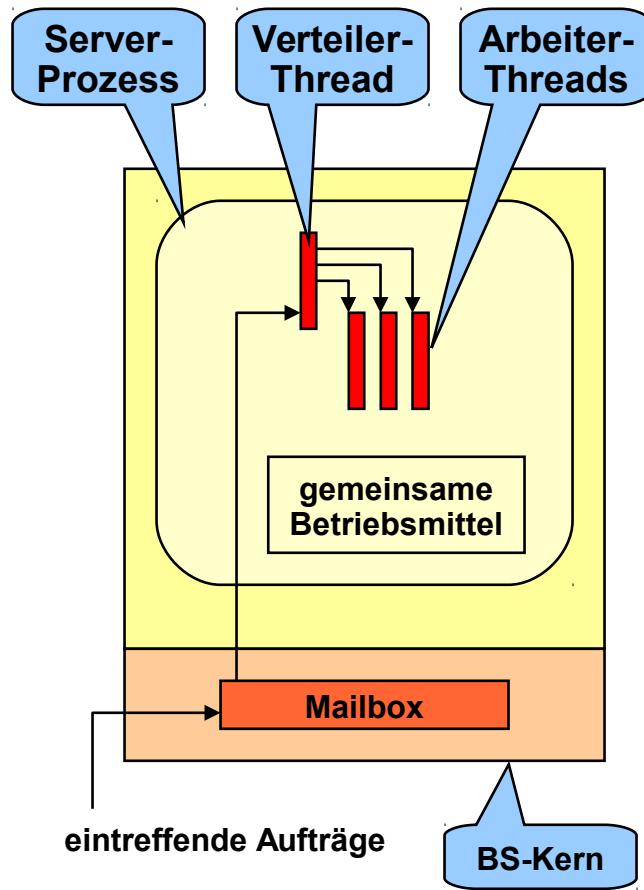
- Prozesserzeugung, Prozessumschaltung und Prozesskommunikation sind teuer
 - = rechenzeitaufwendig zur Laufzeit
 - Verluste auch durch Cache-Misses
- Wie nutzt man mehrere Prozessoren eines Multiprozessors für eine Applikation?
 - Z.B. mehrere kooperierende Prozesse auf verschiedenen Prozessoren
 - Muss vom Applikationsprogrammierer ausprogrammiert werden !
- Wie strukturiert man einen Server-Prozess, der Anforderungen von mehreren Klienten bedienen kann?
 - Ein Server-Prozess = keine Parallelität
 - Multiplexing für verschiedene Klienten von Hand
 - = komplexe Programmierung

Lösung

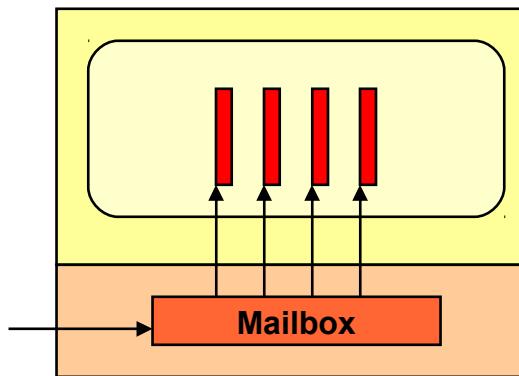
- Einführung von billiger Nebenläufigkeit in einem Prozessadressraum durch „Leichtgewichtsprozesse“, sogenannte Threads.

- Prozess (Einheit der BM-Verwaltung)
 - ausführbares Programm, das Code und globale Daten definiert.
 - privater Adressraum.
Code und Daten über Adressraum zugreifbar.
 - Menge von Betriebsmitteln
 - geöffnete Objekte, Betriebssystem-Objekte wie z.B. Timer, Signale, Semaphore
 - dem Prozess durch das BS als Folge der Programmausführung zugeordnet.
 - Menge von Threads.
- Threads (Aktivitätsträger)
 - Idee einer „parallel ausgeführten Programmfunktion“
 - Eigener Prozessor-Context (Registerinhalte usw.)
 - Eigener Stack (i.d.R. zwei, getrennt für user und kernel mode)
 - Eigener kleiner privater Datenbereich (Thread Local Storage)
 - Threads eines Prozesses nutzen gemeinsam Programm, Adressraum und alle Betriebsmittel.

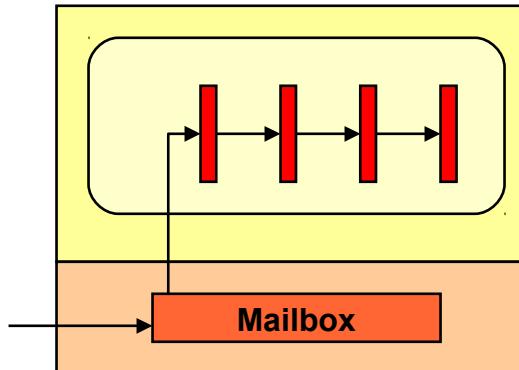
(a) Verteiler/Arbeiter-Modell



(b) Team-Modell



(c) Fließband-Modell



aus Tanenbaum: Moderne Betriebssysteme

- Nicht einheitlich
- Bsp: Java: Klasse Thread in `java.lang`
 - Z.B. Implementierung der Schnittstelle `java.lang.Runnable` und Implementierung von Methode `run()`.
 - Thread-Modell in JVM implementiert
- Bsp: C++: Boost Threads verbreitet
- Bsp: Windows
 - C-Schnittstelle für Windows API, u.a. `CreateThread(...)`
- Bsp: LinuxThreads (veraltet)
 - `clone()` system call
 - Erzeugung eines Prozesses mit Angabe detaillierter Flags, was gemeinsam mit erzeugendem Prozess genutzt werden soll
 - Gehört nicht zum Unix Standard
 - ⇒ Programme nicht portierbar

- POSIX Threads (Pthreads)
 - Verbreitete Standard-Programmierschnittstelle für Threading und entsprechende Synchronisation
 - Standardisiert in IEEE POSIX 1003.1c (1995)
 - Pthreads: Implementierungen dieses Standards
 - Auf vielen Systemen, insbesondere auch Multiprozessorsystemen, z.B. Linux, Solaris, MacOS X, FreeBSD
 - Neben C / C++ auch für andere Programmiersprachen: z.B. Fortran
 - Teilweise Bestandteil der libc
 - Ca. 50 Funktionen

• API-Aufrufe zum Thread-Management

(wird fortgesetzt für andere Funktionsbereiche wie datenbezogene Synchronisation)

```
#include <pthread.h>

int pthread_create(pthread_t * thread,
                  const pthread_attr_t * attr,
                  void * (*start_routine)(void *),
                  void *arg);
    Erzeugen eines Threads

void pthread_exit(void *retval);
    Sich selbst beenden

pthread_t pthread_self(void);
    Thread-Identifier des aktuellen Threads ermitteln

int pthread_join(pthread_t th, void **thread_return);
    Warten auf Beendigung eines Threads

int pthread_cancel(pthread_t thread);
    Beenden eines anderen Threads
```

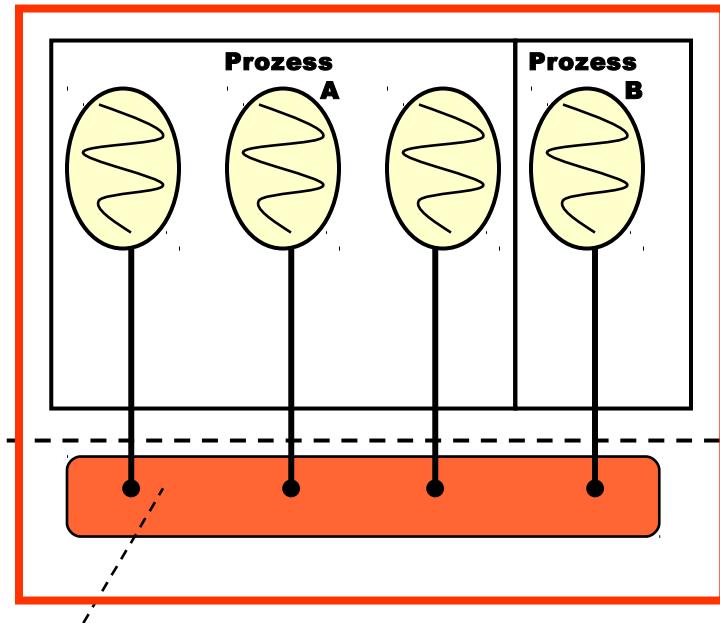
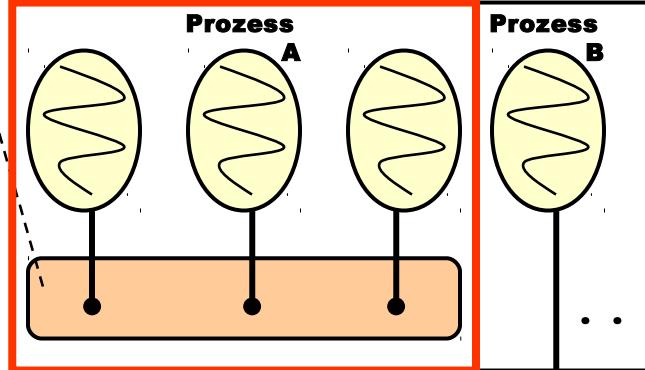


- Im BS-Kern:
genannt Kernel Threads:
 - Betriebssystem unterstützt Threads
 - Threads sind die Einheiten, denen Prozessoren zugeordnet werden (Schedulable Entities).
 - 1:1-Zuordnung
 - Beispiele: Windows NT, Mach, Chorus, Amoeba
- Nur in Thread Library:
genannt User-Level Threads
 - Programmierung mit Threads, aber BS-Scheduler kennt nur übliche Prozesse.
 - 1:n-Zuordnung (allg. m:n)
 - Beispiel:
POSIX Pthreads in OSF/DCE

Kernel Thread vs. User level Thread



Thread-Bibliothek

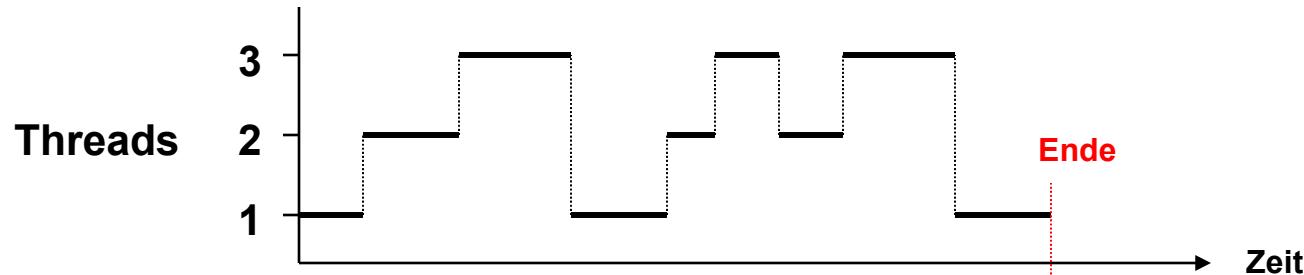


Prozess-Scheduler des Betriebssystems

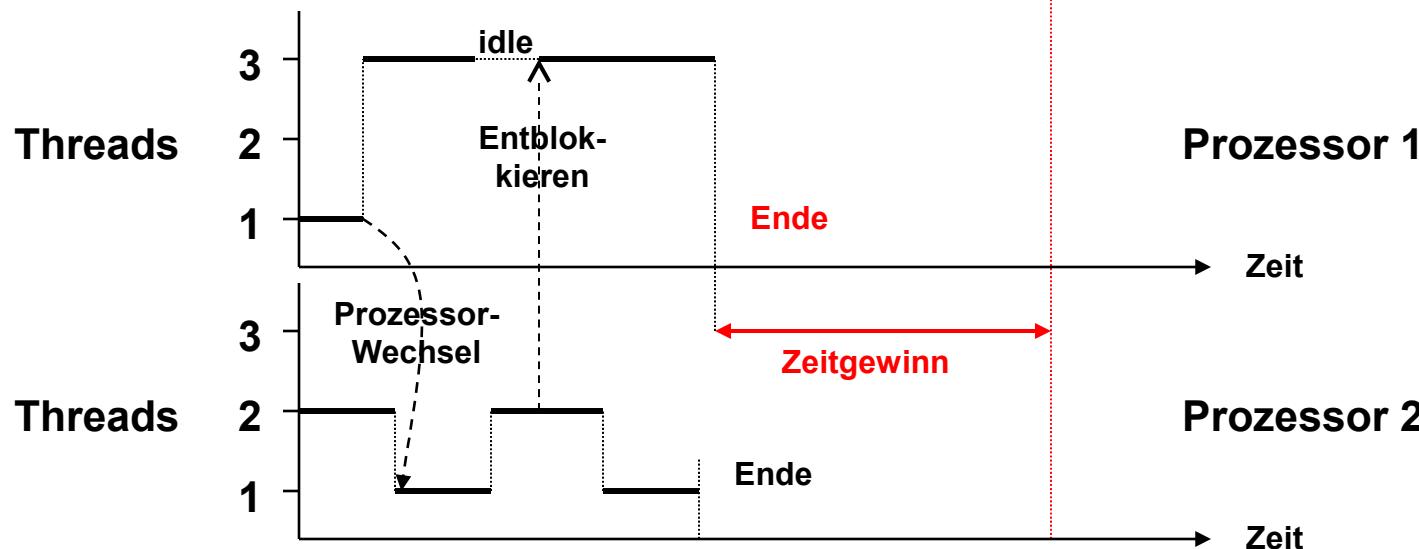
„Process Scope“

„System Scope“

Ein-Prozessor-System:



Zwei-Prozessor-System:





- Native POSIX Thread Library (NPTL)
- Federführung Red Hat
- Ziele:
 - Konformität zu POSIX Pthreads
 - Gute Multiprozessor-Performance
 - Niedrige Erzeugungskosten
 - Kompatibilität zu LinuxThreads
- 1:1-Implementierung
 - Kernel verwaltet Prozesse
 - `pthread_create` führt zu neuem Prozess unter Nutzung von `clone()`
 - Spezielle Kernel-Unterstützung und viel Optimierung im Kernel (z.B. sog. Futexes)
- Nutzung zusammen mit GNU libc (glibc)

Was haben wir in Kap. 3 gemacht?

- Konzept des sequentiellen Prozesses (Wichtig!).
- Strukturierung von Aktivität durch eine Menge von sequentiellen Prozessen, die zueinander nebenläufig ausgeführt werden.
- Betriebssystem bietet Anwendungsprogrammierern ein solches Prozesskonzept an der Dienstschnittstelle zur Strukturierung von Anwendungen.
- Das Betriebssystem kann Prozesse auch intern zur Strukturierung höherer Funktionalität nutzen.
- Ansätze besprochen, wie Betriebssystem das Prozesskonzept implementiert (prinzipiell und speziell am Beispiel UNIX).
- Thread-Konzept als performante „Leichtgewichtsprozesse“ vorgestellt.

Kap. 4: Scheduling

4.1 Einführung

4.2 Non-Preemptive Scheduling-Verfahren

4.3 Preemptive Scheduling-Verfahren

4.4 Scheduling in UNIX

4.5 Zusammenfassung

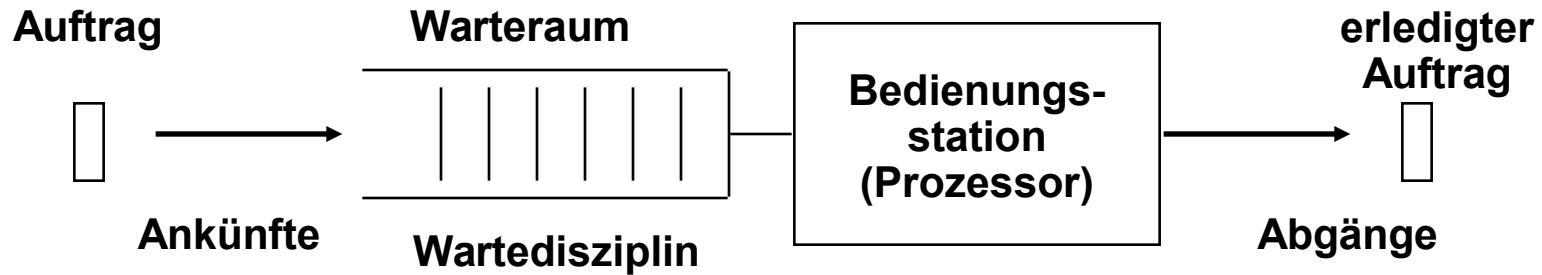
Grundlegende Begriffe:



- Wiederholung aus Kap. 3.1:
 - Scheduler oder Dispatcher: Umschalteinheit
 - Scheduling-Algorithmus: zugehöriger Algorithmus
 - Prozesswechsel oder Kontextwechsel: Umschaltungsvorgang, verursacht Kosten.
- Preemptives Scheduling: rechnende Prozesse können suspendiert werden (Prozessorenzug)
 - preemptive-resume: Fortsetzung ohne Verlust
 - preemptive-repeat: Beginn von vorne
- Non-preemptives Scheduling-Verfahren (Run-to-Completion, d.h. Prozess ist solange aktiv, bis er endet oder sich selbst blockiert).
- Non-preemptive-Scheduling-Verfahren sind für General Purpose Systeme mit interaktiven Benutzern nicht geeignet.

- Prioritäten-basierte Scheduling-Verfahren: ordnen Prozessen Prioritäten zu (relative Wichtigkeit).
- Prioritäten können extern vorgegeben sein, oder intern durch das Betriebssystem selbst bestimmt werden.
- Prioritäten können statisch sein, d.h. ändern sich während der Bearbeitung nicht, andernfalls dynamischen Prioritäten.
- Scheduling-Verfahren, die Prozessorzuteilung auf der Basis gewählter Zeitspannen mittels Uhrunterbrechungen steuern, heißen Zeitscheiben-basierte Scheduling-Verfahren.
- Mischformen sind möglich.

- Mehrstufiges Scheduling: Scheduler auf mehreren Ebenen
Nur die auf einer höheren Ebene durch den entsprechenden Scheduler ausgewählten Prozesse sind dem Scheduler der nächst niederen Ebene für sein Scheduling sichtbar. Scheduling der niederen Ebene wird dann häufig als Dispatching bezeichnet.
Typisches Anwendungsbeispiel: 2-stufiges Scheduling;
Der Scheduler der höheren Ebene transportiert Aufträge vom Hintergrundspeicher in den Arbeitsspeicher und zurück (Swapping); der Scheduler der niederen Ebene berücksichtigt ausschließlich Prozesse, die sich im Arbeitsspeicher befinden.



- Auftrag: Einheit zur Bearbeitung (z.B. Stapeljob, Dialogschritt).
- Bedienzeit: Zeitdauer für die reine Bearbeitung eines Auftrags durch die Bedienstation (den Prozessor).
- Wartedisziplin: einfache:
 - FCFS (First-Come-First-Served) oder FIFO (First-In-First-Out),
 - LIFO (Last-In-First-Out),
 - Random (zufällig ausgewählt).
- Ankünfte und Bedienungszeiten werden bei Bedienungsmodellen häufig durch stochastische Prozesse modelliert.
- komplexere Bedienmodelle können mehrere Bedienstationen (Multiprozessorsystem), mehrere Warteräume, mehrphasige Bedienung sowie die Rückführung teilweise bearbeiteter Aufträge enthalten.

Bedienung eines Auftrags:



- Antwortzeit: Zeitdauer vom Eintreffen eines Auftrags bis zur Fertigstellung.
bei Dialogaufträgen: Zeitdauer von einer Eingabe eines Benutzers (z.B. Drücken der return-Taste) bis zur Erzeugung einer zugehörigen Ausgabe (z.B. auf dem Bildschirm).
bei Stapelaufträgen auch Verweilzeit genannt.
- Wartezeit: Antwortzeit - Bedienzeit.
- Durchsatz: Anzahl der erledigten Aufträge pro Zeiteinheit.
- Auslastung: Anteil der Zeit im Zustand belegt.
- Fairness: "Gerechte" Behandlung aller Aufträge, z.B. alle rechenwilligen Prozesse haben gleichen Anteil an der zur Verfügung stehenden Prozessorzeit.

Ziele:

Nutzer-bezogen:

- Kurze Antwortzeiten bei interaktiven Aufträgen.
- Kurze Verweilzeiten für Stapelverarbeitungsaufträge.

Betreiber-bezogen:

- Hoher Durchsatz.
- Hohe Auslastung.
- Fairness in der Behandlung aller Aufträge.
- Geringer Aufwand für die Bearbeitung des Scheduling-Algorithmus selbst (Overhead!).

Vorabwissen über die Bedienzeit:

- Einige Verfahren verlangen Kenntnis der Bedienzeit eines Auftrags bei dessen Ankunft (vgl. Verfahren zur Maschinenbelegung, Operations Research).
- Nur realistisch für wiederkehrende Stapeljobs, nicht realistisch bei interaktiven Aufträgen.

Trennung von Strategie und Mechanismus:

- Ziel: höhere Flexibilität.
 - Parametrisierbarer Scheduling-Mechanismus auf niederer Ebene (im Betriebssystemkern).
 - Parameter können auf höherer Ebene (z.B. über Systemaufrufe in Benutzerprozessen) gesetzt werden, um applikationsbezogene Strategie zu implementieren.
- ⇒ Das Scheduling wird damit weiter auf niederer Ebene durchgeführt, aber von der Applikationsebene aus gesteuert.
- Beispiel: Ein Datenbank-Management-Prozess kann für seine Kindprozesse, die z.B. bestimmte ihm bekannte Anfragen bearbeiten oder interne Dienste durchführen, deren optimale Einplanung und Abfolge bewirken.

User-Level-Scheduler:

- Über die obige Forderung hinaus kann ein Applikationsprogramm dem Betriebssystem z.B. über einen Systemaufruf einen Scheduler übergeben, der für eine Teilmenge der Prozesse deren Scheduling durchführt (Höchstmaß an Flexibilität).
- Nur in wenigen Betriebssystemen vorhanden

4.2 Non-Preemptive Scheduling



(für Stapeljobs mit bekannten Bedienzeiten)

1. First-Come-First-Served (FCFS)
2. Shortest-Job-First (SJF)
3. Prioritäts-Scheduling (Prio)

4.2.1 First-Come-First-Served (FCFS)



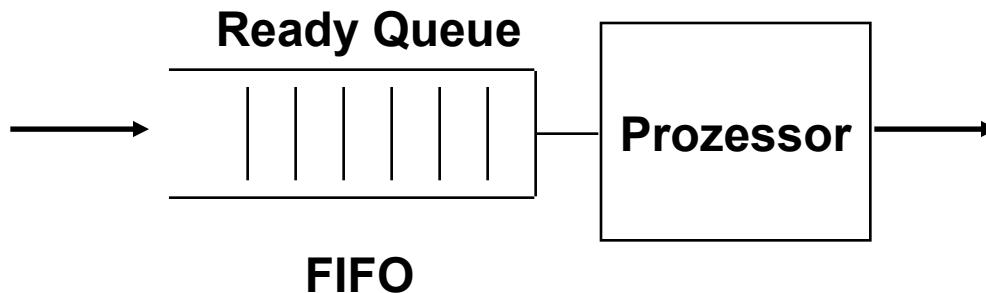
Algorithmus:

- Einfachst-Algorithmus: "Wer zuerst kommt, mahlt zuerst".
- Vollständige Bearbeitung jedes Auftrags, bevor ein neuer begonnen wird.

Implementierung:

- Die Ready-Queue wird als FIFO- Liste verwaltet.

Bedienungsmodell:



Prozess Bedienzeit

1
2
3

13
3
6

Alle Aufträge seien zur
Zeit Null bekannt.

Resultierender Schedule:

1	2	3
---	---	---

t=0

13 16

22

Prozess Wartezeit Antwortzeit

1 0 13
2 13 16
3 13+3=16 22

Durchschnittliche Wartezeit:
 $(13+16)/3 = 29/3$

Im Falle der Ausführungsfolge 3, 2, 1 hätte sich ergeben:
Durchschnittliche Wartezeit: $(6+9)/3 = 5$

Algorithmus:

- Von allen rechenwilligen Prozessen wird derjenige mit der kleinsten Bedienzeitanforderung ausgewählt.
- Bei gleicher Bedienzeitanforderung wird nach FCFS gewählt.
- Der Algorithmus SJF ist in dem Sinne optimal in der Menge aller möglichen Algorithmen, dass er die kürzeste mittlere Wartezeit für alle Aufträge sichert.
- Notwendigkeit der Kenntnis der Bedienzeit!

Prozess Bedienzeit

1	13
2	3
3	6

Alle Aufträge seien zur Zeit Null bekannt.

Resultierender Schedule:

2	3	1
---	---	---

t=0 3 9 22

Prozess Wartezeit Antwortzeit

1	3+6=9	22
2	0	3
3	3	9

Durchschnittliche Wartezeit: $(9+3)/3 = 4$

Algorithmus:

- Jeder Auftrag besitze eine statische Priorität.
- Von allen rechenwilligen Prozessen wird derjenige mit der höchsten Priorität ausgewählt.
- Bei gleicher Priorität wird nach FCFS ausgewählt.

Rechenbeispiel:

Prozess	Bedienzeit	Priorität
---------	------------	-----------

1	13	2
2	3	3
3	6	4

Alle Aufträge seien zur Zeit Null bekannt.

Resultierender Schedule:

3	2		1
---	---	--	---

t=0

6

9

22

4.3 Preemptive Scheduling



(realistisch für heutige Rechensysteme)

1. Round-Robin-Scheduling (RR)
2. Dynamisches Prioritäts-Scheduling
3. Mehrschlangen-Scheduling
4. Mehrschlangen-Feedback-Scheduling
5. Earliest-Deadline-First-Scheduling

Algorithmus:

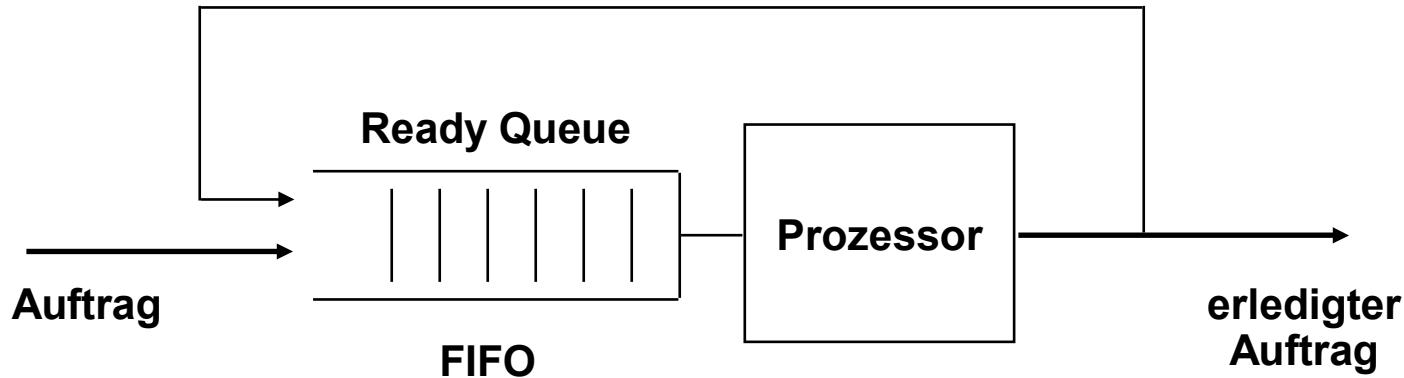
- Menge der rechenwilligen Prozesse linear geordnet.
- Jeder rechenwillige Prozess erhält den Prozessor für eine feste Zeitdauer q , die Zeitscheibe (time slice) oder Quantum genannt wird.
- Nach Ablauf des Quantums wird der Prozessor entzogen und dem nächsten zugeordnet (preemptive-resume).
- Tritt vor Ende des Quantums Blockierung oder Prozessende ein, erfolgt der Prozesswechsel sofort.
- Dynamisch eintreffende Aufträge werden z.B. am Ende der Warteschlange eingefügt.

Implementierung:

- Die Zeitscheibe wird durch einen Uhr-Interrupt realisiert.
- Die Ready Queue wird als lineare Liste verwaltet, bei Ende eines Quantums wird der Prozess am Ende der Ready Queue eingefügt.

Bedienungsmodell:

Ablauf des Quantums



Bewertung:

- Round-Robin ist einfach und weit verbreitet.
- Alle Prozesse werden als gleichwichtig angenommen und fair bedient.
- Einziger kritischer Punkt: Wahl der Dauer des Quantums.
 - Quantum zu klein \Rightarrow häufige Prozesswechsel, sinnvolle Prozessornutzung sinkt
 - Quantum zu groß \Rightarrow schlechte Antwortzeiten bei kurzen interaktiven Aufträgen.

Prozess Bedienzeit

1	13	Alle Aufträge seien zur Zeit Null bekannt.
2	3	Quantum sei $q = 4$.
3	6	

Resultierender Schedule:

1	2	3	1	3	1	1
t=0	4	7	11	15	17	21 22

Prozess Wartezeit Antwortzeit

1	$3+4+2=9$	22
2	4	7
3	$4+3+4=11$	17

Durchschnittliche Wartezeit: $(9+4+11)/3 = 8$

Grenzwertbetrachtung für Quantum q :

- $q \rightarrow \text{unendlich}$:
Round-Robin verhält sich wie FCFS.
- $q \rightarrow 0$:
Round-Robin führt zu sogenanntem processor-sharing:
jeder der n rechenwilligen Prozesse erfährt $1/n$ der
Prozessorleistung. (Kontextwechselzeiten als Null angenommen).

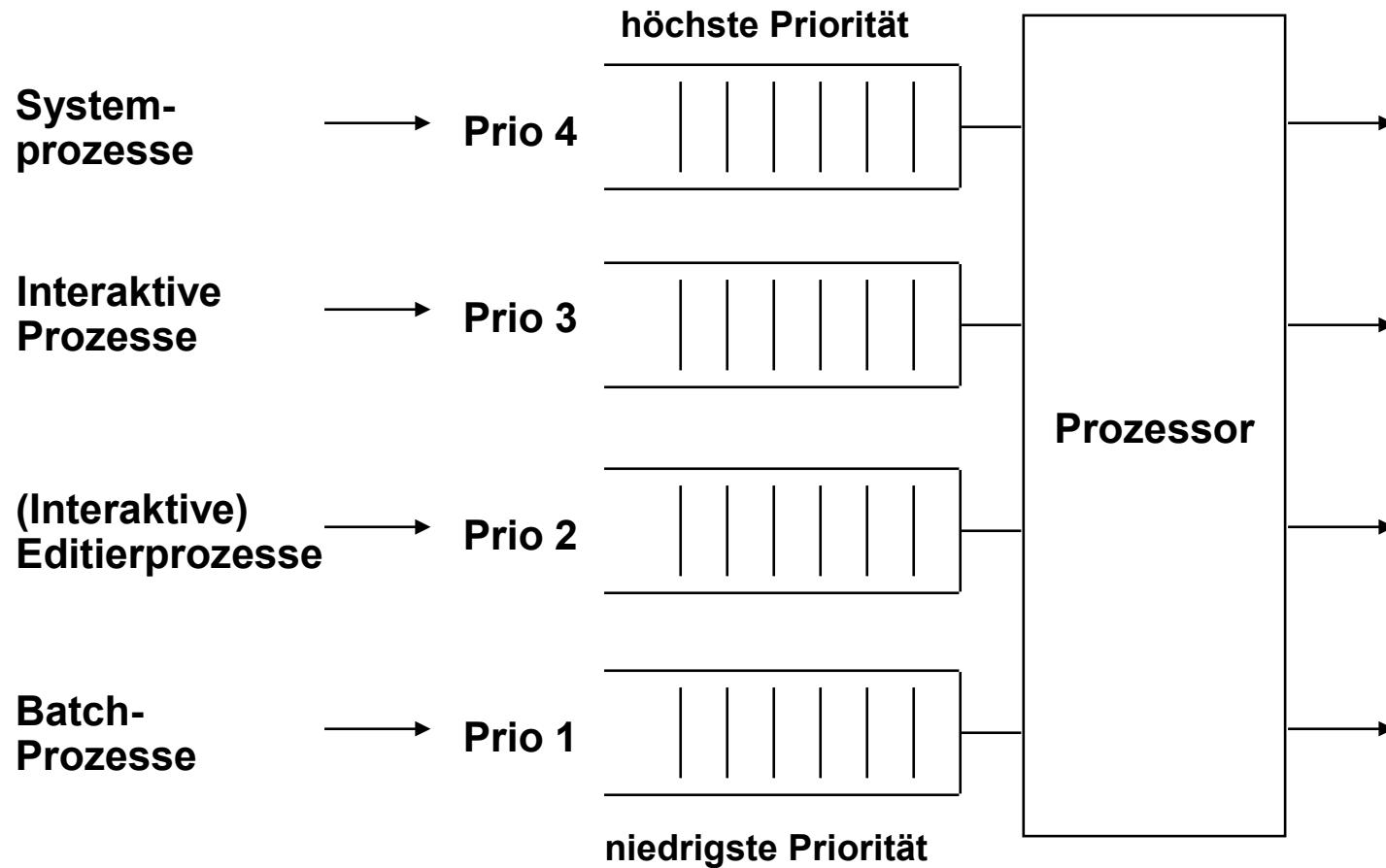
Algorithmus:

- Jedem Prozess ist ein statische Priorität zugeordnet.
- Prozesse werden gemäß ihrer Priorität in eine Warteschlange eingereiht.
- Es wird jeweils der rechenwillige Prozess mit der höchsten Priorität ausgewählt und bedient.
- Wird ein Prozess höherer Priorität rechenwillig (z.B. nach Beendigung einer Blockierung), so wird der laufende Prozess unterbrochen (preemption) und in die Ready Queue eingefügt.

Algorithmus:

- Prozesse werden statisch klassifiziert als einer bestimmten Gruppe zugehörig (z.B. interaktiv, batch).
- Alle rechenwilligen Prozesse einer bestimmten Klasse werden in einer eigenen Ready Queue verwaltet.
- Jede Ready Queue kann ihr eigenes Scheduling-Verfahren haben (z.B. Round-Robin für interaktive Prozesse, FCFS für batch-Prozesse).
- Zwischen den Ready Queues wird i.d.R. unterbrechendes Prioritäts-Scheduling angewendet, d.h.: jede Ready Queue besitzt eine feste Priorität im Verhältnis zu den anderen; wird ein Prozess höherer Priorität rechenwillig, wird der laufende Prozess unterbrochen (preemption).

Bedienungsmodell (Beispiel):



Prinzip:

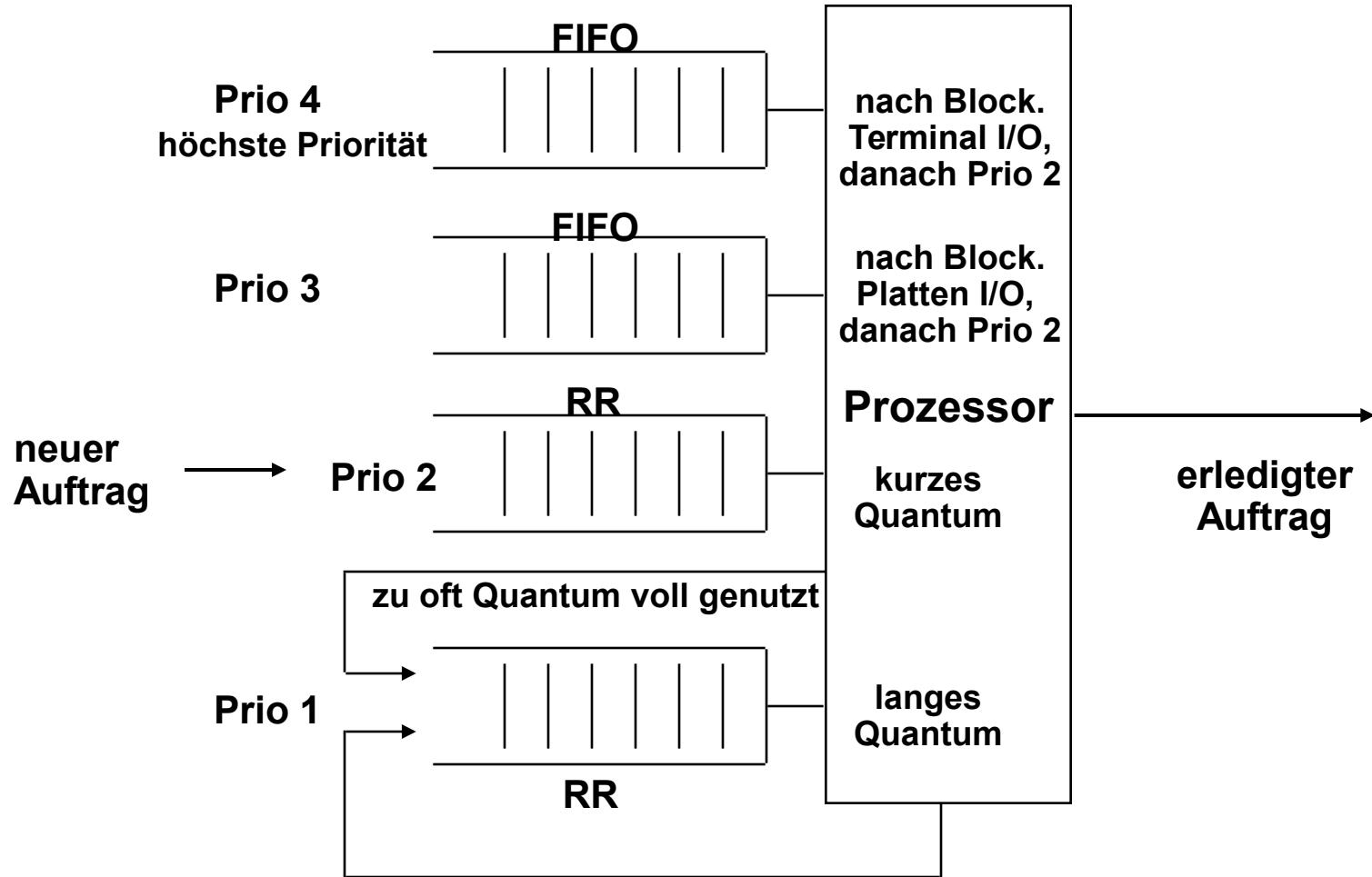
- Erweiterung des Mehrschlangen-Scheduling.
- Rechenwillige Prozesse können im Verlauf in verschiedene Warteschlangen eingeordnet werden (dynamische Prioritäten).
- Algorithmen zur Neubestimmung der Priorität wesentlich

Beispiel 1: Wenn ein Prozess blockiert, wird die Priorität nach Ende der Blockierung um so größer, je weniger er von seinem Quantum verbraucht hat (Bevorzugung von I/O-intensiven Prozessen).

Beispiel 2: Wenn ein Prozess in einer bestimmten Priorität viel Rechenzeit zugeordnet bekommen hat, wird seine Priorität verschlechtert (Bestrafung von Langläufern).

Beispiel 3: Wenn ein Prozess lange nicht bedient worden ist, wird seine Priorität verbessert (Altern, Vermeidung einer "ewigen" Bestrafung).

Bedienungsmodell:



- Mit wachsender Bedienzeit sinkt die Priorität, d.h. Kurzläufer werden bevorzugt, Langläufer werden zurückgesetzt.
- Wachsende Länge des Quanta mit fallender Priorität verringert die Anzahl der notwendigen Prozesswechsel (Einsparen von Overhead).
- Verbesserung der Priorität nach Beendigung einer Blockierung berücksichtigt I/O-Verhalten (Bevorzugung von I/O-intensiven Prozessen). Durch Unterscheidung von Terminal I/O und sonstigem I/O können interaktive Prozesse weiter bevorzugt werden.
- sehr flexibel.

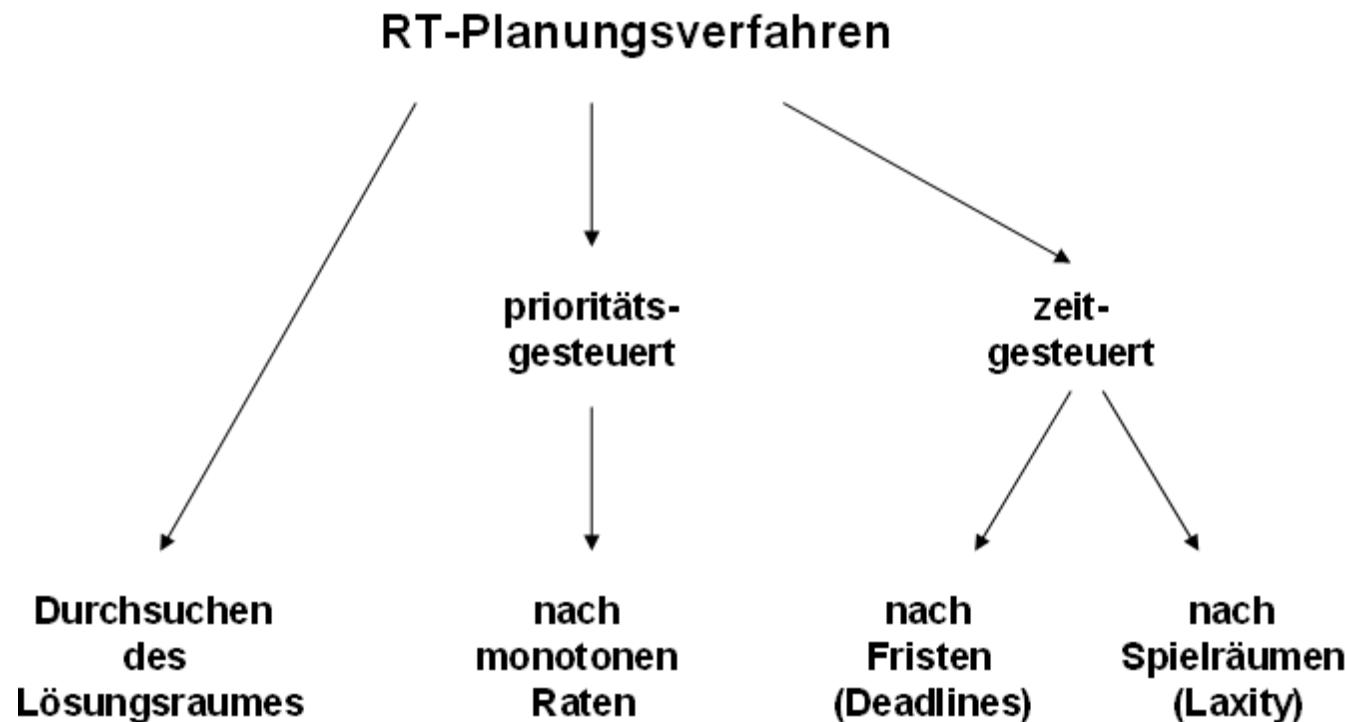
4.3.5 Echtzeit-Scheduling



- Scheduling in Realzeit-Systemen beinhaltet zahlreiche neue Aspekte. Hier nur erster kleiner Einblick.
- Varianten in der Vorgehensweise
 - *Statisches Scheduling:*
Alle Daten für die Planung sind vorab bekannt, die Planung erfolgt durch eine Offline-Analyse.
 - *Dynamisches Scheduling:*
Daten für die Planung fallen zur Laufzeit an und müssen zur Laufzeit verarbeitet werden.
 - *Explizite Planung:*
Dem Rechensystem wird ein vollständiger Ausführungsplan (Schedule) übergeben und zur Laufzeit befolgt (Umfang kann extrem groß werden).
 - *Implizite Planung:*
Dem Rechensystem werden nur die Planungsregeln übergeben.

Klassifizierung

4.3.5



- Gewisse Prozesse müssen häufig zyklisch ausgeführt werden.
- Hard-Realtime-Prozesse müssen unter allen Umständen ausgeführt werden (ansonsten sind z.B. Menschenleben bedroht).
- Zeitliche Fristen (Deadlines) vorgegeben, zu denen der Auftrag erledigt sein muss.
- Scheduler muss die Erledigung aller Hard-Realtime-Prozesse innerhalb der Fristen garantieren.
- Scheduling geschieht in manchen Anwendungssystemen statisch vor Beginn der Laufzeit (z.B. Automotive). Dazu muss die Bedienzeit-Anforderung (z.B. worst case) bekannt sein.
- Im Falle von dynamischem Scheduling ist das Earliest-Deadline-First-Scheduling-Verfahren verbreitet, das den Prozess mit der frühesten fälligen Frist zur Ausführung auswählt.

Ziel:

- Gute Antwortzeiten für interaktive Prozesse.

Vorgehensweise:

- Zweistufiges Scheduling:
 1. Hintergrundspeicher-Hauptspeicher-Swapping.
 2. Scheduling von im Hauptspeicher befindlichen Prozessen.

Anmerkungen:

- Im folgenden nur die untere (2.) Scheduling-Stufe betrachtet.
- UNIX-Varianten unterscheiden sich bzgl. des Scheduling. Im folgenden wird UNIX System V Rel. 2 betrachtet (relativ einfach, vgl. [Bach]).

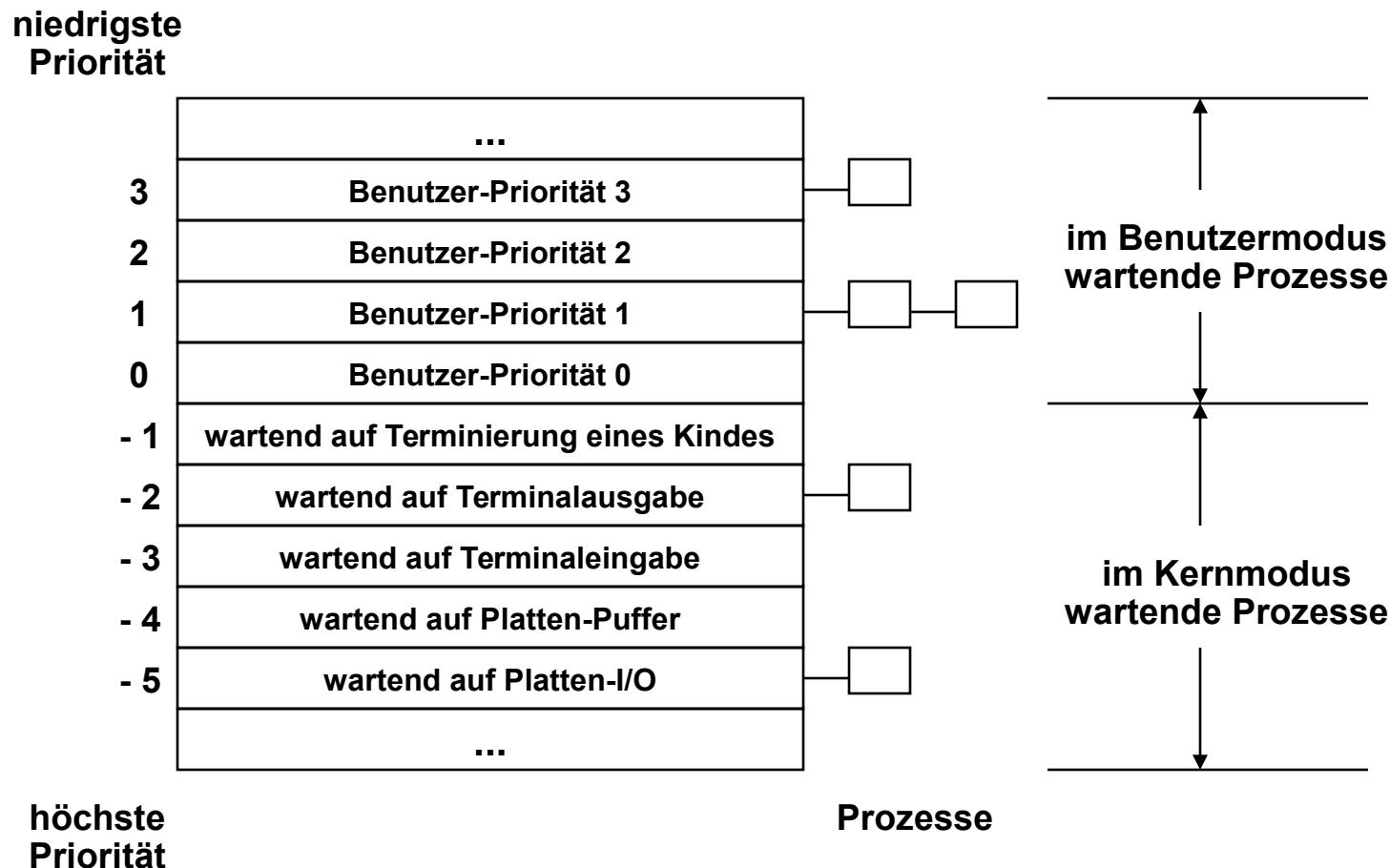
Mehrschlangen-Feedback-Scheduling:

- Rechenwillige Prozesse werden in Ready Queues gehalten. Blockierte Prozesse werden außerhalb geführt.
- Jede Ready Queue besitzt eine Priorität. Prioritäten sind ganze Zahlen. Je kleiner (negativer) der Wert, je vorrangiger der Prozess.
- Prozesse im Kernmodus haben negative Prioritäten, Prozesse im Benutzermodus haben nicht-negative Prioritäten.
- Jeder Benutzerprozess hat Basispriorität (bestenfalls 0). Die Basispriorität kann mittels `nice` verschlechtert werden ("be nice to *others*", nur super-user kann verbessern).
- Ein ausgewählter Prozess wird für die Dauer eines Quantums ausgeführt, typ. 100 msec (4-6 Ticks der Systemuhr), wenn er nicht vorher blockiert.
- Am Ende des Quantums wird ein Benutzerprozess an das Ende derselben Ready Queue zurückgestellt, d.h. Round-Robin auf jeder Benutzer-Prioritätsebene.

Feedback-Verfahren (vereinfacht):

- Bei jedem Uhr-Tick wird CPU-Nutzungszähler des aktuellen Prozesses inkrementiert.
- CPU-Nutzungszähler wird auf die Basispriorität des Prozesses addiert (d.h. die Priorität wird schlechter, und der Prozess wird in eine niedere priore Ready Queue verdrängt).
- Einmal je Sekunde wird aktuelle Priorität jedes Prozesses neu berechnet.
 - CPU-Nutzungszähler durch 2 dividieren (Alterungsmech. zum langsamen Vergessen der vergangenen CPU-Nutzung).
 - neue Priorität = Basispriorität + CPU-Nutzungszähler.
- Benutzerprozess, der im Verlauf eines Kernauftrufs im Kern blockiert, erhält, abhängig von der Art der Ereignisses, auf das er wartet, eine Kernpriorität zugeordnet, gemäß der er eingeordnet wird, wenn er wieder rechenwillig wird. Ziel: Schnelles Verlassen des Kerns, Ordnen der Ereignisse nach Wichtigkeit).

Resultierende Warteschlangenstruktur:



- Linux 1.2
 - Zyklische Liste, Round-Robin
- Linux 2.2
 - Scheduling-Klassen (Echtzeit, Non-Preemptive, Nicht-Echtzeit)
 - Unterstützung für Multiprozessoren
- Linux 2.4
 - $O(n)$ -Komplexität (jeder Task-Kontrollblock muss angefasst werden)
 - Round-Robin
 - Teilweiser Ausgleich bei nicht verbrauchter Zeitscheibe
 - Insgesamt relativ schwacher Algorithmus

- Linux 2.6
 - $O(1)$ -Komplexität (konstanter Aufwand für Auswahl unabhängig von Anzahl Tasks)
 - Run Queue je Priorität
 - Zahlreiche Heuristiken für Entscheidung I/O-intensiv oder rechenintensiv
 - Sehr viel Code
- Ab Linux Kernel 2.6.23: Completely Fair Scheduler (CFS)
 - Sehr gute Approximation von Processor Sharing
 - Task mit geringster *Virtual Runtime* (größter Rückstand) bekommt Prozessor
 - Zeit-geordnete spezielle Baumstruktur für Taskverwaltung ($\rightarrow O(\log n)$ -Komplexität)
 - Kein periodischer Timer-Interrupt sondern One-Shot-Timer

Was haben wir in Kap. 4 gemacht?

- Scheduling-Verfahren
 - Kenngrößen wie Antwortzeit, Auslastung und Fairness relevant
 - einfache Verfahren: FCFS, SJF, PRIO
 - unterbrechende Verfahren Round-Robin, statische Prioritäten, Mehrschlangen-Verfahren
 - flexible Feedback-Algorithmen, nach denen Prioritäten dynamisch neu berechnet werden
 - Beispiel UNIX/Linux

Kap. 5: Prozesssynchronisation

5.1 Einführung

5.2 Synchronisationsprimitive

5.3 Klass. Synchronisationsprobleme

5.4 Zusammenfassung

Problemstellung:

- **Gegeben:** Nebenläufige (konkurrente) Prozesse (vgl. Kap. 3, engl.: concurrent processes).
- **Ziel:** Vermeidung ungewollter gegenseitiger Beeinflussung.
- **Ziel:** Unterstützung gewollter Kooperation zwischen Prozessen:
 - Gemeinsame Benutzung von Betriebsmitteln (Sharing)
 - Übermittlung von Signalen
 - Nachrichtenaustausch

⇒ **Fazit:** Mechanismen zur Synchronisation und Kommunikation von Prozessen sind notwendig.

- Gemeinsame Benutzung eines Speicherbereiches
(Hier: Datum „Kontostand“)
- Ungewollte gegenseitige Beeinflussung

Prozess 1

```
/* Gehaltsüberweisung */  
z = lies_kontostand();  
z = z + 1000;  
schreibe_kontostand(z);
```

Prozess 2

```
/* Dauerauftrag Miete */  
x = lies_kontostand();  
x = x - 800;  
schreibe_kontostand(x);
```

Möglicher Ablauf

Prozess 1

```
/* Gehaltsüberweisung */  
z = lies_kontostand();  
z = z + 1000;  
schreibe_kontostand(z);
```

Prozess 2

```
/* Dauerauftrag Miete */  
x = lies_kontostand();  
x = x - 800;  
schreibe_kontostand(x);
```

Mögliche Ausführungsreihenfolge der Anweisungen in Prozess 1,2

```
/* Gehaltsüberweisung */          /* Dauerauftrag Miete */  
z = lies_kontostand();  
                                x = lies_kontostand();  
z = z + 1000;  
schreibe_kontostand(z);  
                                x = x - 800;  
                                schreibe_kontostand(x);
```

Pech, die Gehaltsüberweisung ist „verloren gegangen“ :-(

Bei anderen Reihenfolgen werden die beiden Berechnungen „richtig“ ausgeführt, oder es geht der Dauerauftrag verloren.

- Annahme: Prozesse mit Lese/Schreib-Operationen auf Betriebsmitteln
- **Def** Zwei nebenläufige Prozesse heißen im Konflikt zueinander stehend oder überlappend, wenn es ein Betriebsmittel gibt, das sie gemeinsam (lesend und schreibend) benutzen, ansonsten heißen sie unabhängig oder disjunkt.
- Folgen von Lese/Schreib-Operationen der verschiedenen Prozesse heißen zeitkritische Abläufe (engl. race conditions), wenn die Endzustände der Betriebsmittel (Endergebnisse der Datenbereiche) abhängig von der zeitlichen Reihenfolge der Lese/Schreib-Operationen sind.

Def

- **Verfahren zum wechselseitigen Ausschluss (engl. mutual exclusion):**
Verfahren, das verhindert, dass zu einem Zeitpunkt mehr als ein Prozess zugreift.
- **Bemerkung:** Ein Verfahren zum wechselseitigen Ausschluss vermeidet zeitkritische Abläufe. Es löst damit ein Basisproblem des Concurrent Programming.
- **Def**
• **Kritischer Abschnitt oder kritischer Bereich (engl. critical section oder critical region):** der Teil eines Programms, in dem auf gemeinsam benutzte Datenbereiche zugegriffen wird.
- **Bemerkung:** Ein Verfahren, das sicherstellt, dass sich zu keinem Zeitpunkt zwei Prozesse in ihrem kritischen Abschnitt befinden, vermeidet zeitkritische Abläufe.
- **Kritische Abschnitte realisieren sog. komplexe unteilbare oder atomare Operationen.**

(Nicht-)atomare Operationen



Nicht atomar

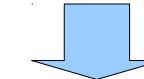
```
/* Gehaltsüberweisung */  
z = lies_kontostand();  
z = z + 1000;  
schreibe_kontostand(z);
```

Atomar ?

```
/* Gehaltsüberweisung 2.0 */  
int kontostand;  
  
kontostand += 1000;
```

Auch Operationen, die in Hochsprache (hier: C) atomar zu sein scheinen, sind es auf Maschinenebene u.U. nicht!

Teilweise architekturabhängig



ASM:

3
Instruktionen,
unterbrechbar!
!

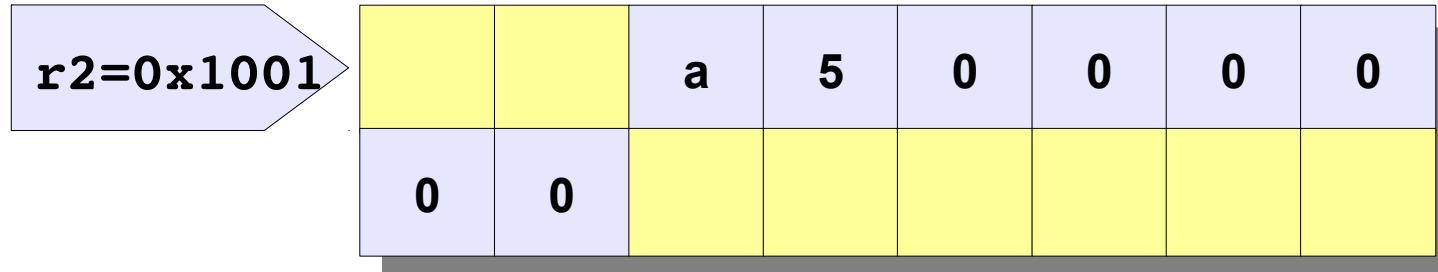
```
lw r2,kontostand  
addu r3,r2,#1000  
sw r3,kontostand
```

Selbst einzelne Maschineninstruktionen können u.U. nicht-atomar sein!

Beispiel:

ASM:
la r2,0x1001
sw 0xa5000000,(r2)

Ein Maschinenbefehl
=> atomar?



Ungerade Adresse => Mehrere Buszyklen zum Schreiben eines Datenwortes erforderlich => unterbrechbar!

Forderungen an einen guten Algorithmus zum wechselseitigen Ausschluss:

- (1) Zu jedem Zeitpunkt darf sich nur ein Prozess in seinem kritischen Abschnitt befinden (Korrektheit, Basisforderung).
- (2) Es dürfen keine Annahmen über die Ausführungsgeschwindigkeiten oder die Anzahl der unterliegenden Prozessoren gemacht werden.
- (3) Kein Prozess, der sich nicht in seinem kritischen Abschnitt befindet, darf andere Prozesse blockieren (Fortschritt).
- (4) Alle Prozesse werden gleich behandelt (Fairness).
- (5) Kein Prozess darf unendlich lange warten müssen, bis er in seinen kritischen Abschnitt eintreten kann (→ „Verhungern“, engl. starvation).

Gliederung

- 1. Wechselseitiger Ausschluss mit aktivem Warten**
- 2. Wechselseitiger Ausschluss mit passivem Warten**
- 3. Semaphore**
- 4. Höhere Synchronisationsprimitive**

Generelle Vorgehensweise:

```
enter_crit();      /* Prolog */  
/* critical section */  
<statement> ;  
...  
<statement> ;  
leave_crit();      /* Epilog */
```

Kritische
Sektion

- Prolog/Epilog-Paar
- Aktives Warten auf Eintritt in den kritischen Abschnitt (engl. busy waiting).
- Aktives Warten für einen längeren Zeitraum verschwendet Prozessorzeit.
- Alle Prozesse müssen sich an das Vorgehen halten.

1. **Sperren aller Unterbrechungen**
2. **Sperrvariablen**
3. **Striktes Alternieren**
4. **Lösung von Peterson (+)**
5. **Test-and-Set-Instruktion (+)**

+ brauchbar

Einfachste Lösung:

- **Jeder Prozess sperrt vor Eintritt in seinen kritischen Abschnitt alle Unterbrechungen (Disable Interrupts).**
- **Jeder Prozess lässt die Unterbrechungen am Ende seines kritischen Abschnitts wieder zu (Enable Interrupts).**

Bemerkungen:

- **Der Prozessor kann nur dann zu einem anderen Prozess wechseln, wenn eine Unterbrechung auftritt. Also ist für diese Lösung die Korrektheitsforderung erfüllt.**
- **Lösung ist unbrauchbar für allgemeine Benutzerprozesse, da nicht zugesichert werden kann, dass sie die Interrupts auch wieder zulassen (z.B. wegen Programmierfehler).**
- **Lösung wird häufig innerhalb des Betriebssystemkerns selbst eingesetzt, um wechselseitigen Ausschluss zwischen Kernroutinen zu gewährleisten (z.B. im alten Einprozessor-UNIX-Kern, um wechselseitigen Ausschluss mit einem Interrupt-Handler sicherzustellen).**
- **Lösung unbrauchbar im Falle eines Multiprozessor-Systems, da sich die Interrupt-Sperre i.d.R. nur auf einen Prozessor auswirkt.**

Einfacher, falscher Lösungsansatz:

- **Jedem krit. Abschnitt wird eine Sperrvariable zugeordnet:**
 - Wert 0 bedeutet "frei",
 - Wert 1 bedeutet "belegt".
- **Jeder Prozess prüft die Sperrvariable vor Eintritt in den kritischen Abschnitt:**
 - Ist sie 0, so setzt er sie auf 1 und tritt in den kritischen Abschnitt ein.
 - Ist sie 1, so wartet er, bis sie den Wert 0 annimmt.
- **Am Ende seines kritischen Abschnitts setzt der Prozess den Wert der Sperrvariablen auf 0 zurück.**

Prozess 1

```
while (sperrvar) { }
sperrvar = 1;
/* kritischer Bereich */
sperrvar = 0;
```

Prozess 2

```
while (sperrvar) { }
sperrvar = 1;
/* kritischer Bereich */
sperrvar = 0;
```

Bemerkungen:

- Prinzipiell gleicher Fehler wie bei Konto-Beispiel:
Zwischen Abfrage der Sperrvariablen und folgendem
Setzen kann der Prozess unterbrochen werden.
- Damit ist es möglich, dass sich beide Prozesse im
kritischen Abschnitt befinden
(Korrektheitsbedingung verletzt !!).
- Speicher (-wörter, -variablen) erlauben nur unteilbare Lese-
und Schreibzugriffe (Eigenschaft der Architektur des
Speicherwerks).

Prozess 1

```
while (sperrvar) { }
sperrvar = 1;
/* kritischer Bereich */
sperrvar = 0;
```

Prozess 2

```
while (sperrvar) { }
sperrvar = 1;
/* kritischer Bereich */
sperrvar = 0;
```

5.2.1.3 Striktes Alternieren



Prozess 0

```
int turn = 0;
```

Prozess 1

```
while (TRUE) {  
    while (turn != 0) /* Warte */  
        ;  
    critical_section();  
    turn = 1;  
    noncritical_section()  
}
```

```
while (TRUE) {  
    while (turn != 1) /* Warte */  
        ;  
    critical_section();  
    turn = 0;  
    noncritical_section()  
}
```

- **Gem. Variable „turn“ gibt an, welcher Prozess den kritischen Bereich betreten darf**
- **Warten wird aktiv durchgeführt.**
- **Prozesse wechseln sich ab: 0,1,0,1,...**
- **Lösung erfüllt Korrektheitsbedingung, aber**
- **Fortschrittsbedingung (3) kann verletzt sein, wenn ein Prozess wesentlich langsamer als der andere ist.**
- **Fazit: keine ernsthafte Lösung.**

5.2.1.4 Lösung von Peterson



- **Historische Vorläufer:**
 - Anfang der 60er Jahre viele Lösungsansätze, nur wenige erfüllten alle Bedingungen aus 5.1.
 - Erste korrekte Software-Lösung für 2 Prozesse: Algorithmus von Dekker.
- **Neue Lösung zum wechselseitigen Ausschluss:**
 - Lösung von Peterson (1981) (im folgenden betrachtet).
 - basiert ebenfalls auf unteilbaren Speicheroperationen und aktivem Warten, ist aber einfacher.
 - Prolog: `enter_region()`, Epilog: `leave_region()`
- **Weitere Lösungen für mehr als zwei Prozesse von:**
 - Dijkstra, Peterson, Knuth, Eisenberg/McGuire, Lamport (hier nicht weiter diskutiert).

Algorithmus

5.2.3

```

/* Algorithmus von Peterson fuer 2 Prozesse */

#define N      2                      /* Anzahl der Prozesse */

/* gemeinsame Variablen */

int turn;                            /* Wer kommt dran? */
bool interested[N];                  /* Wer will, anfangs alle false */

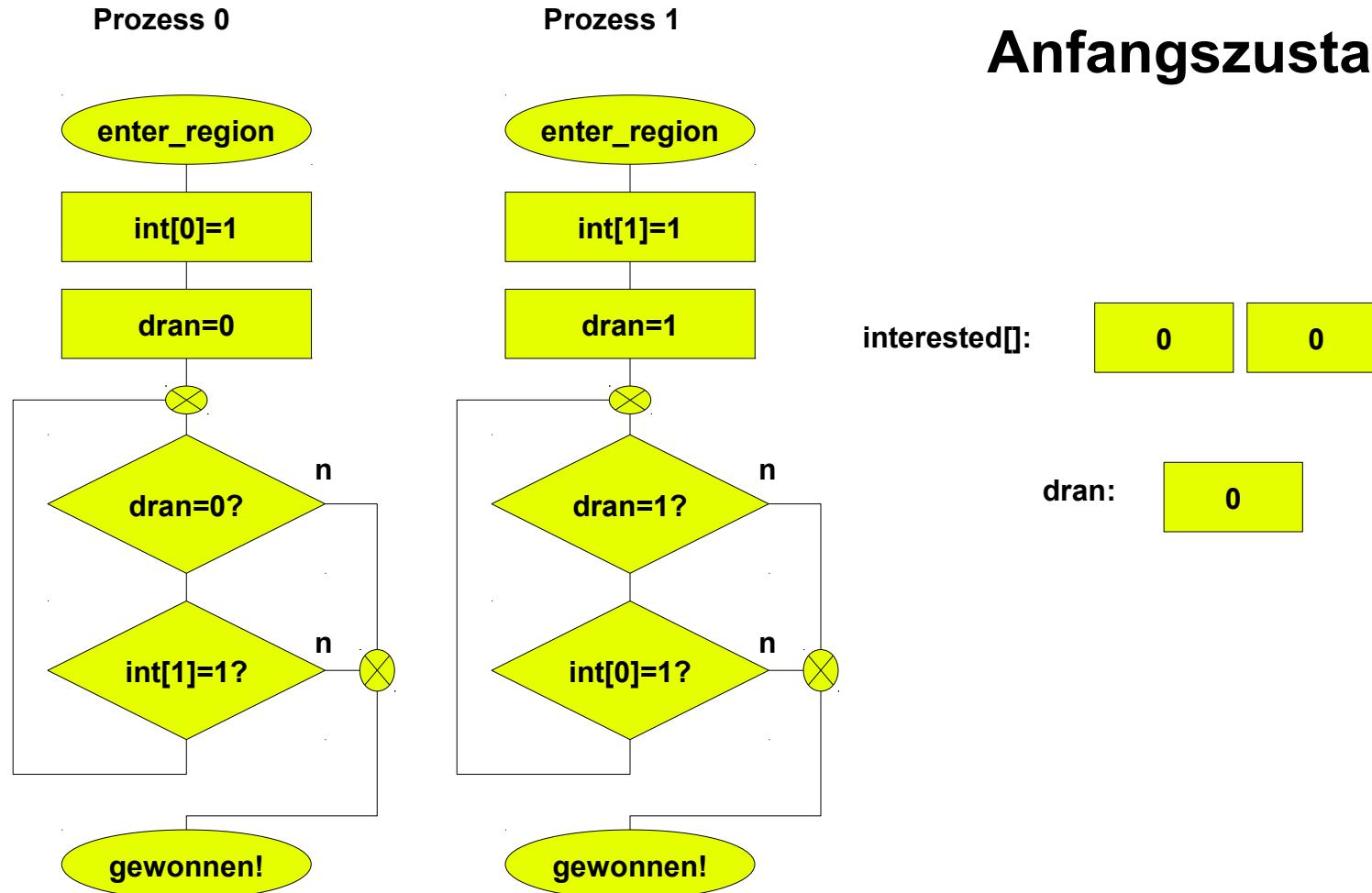
/* Prolog-Operation, vor Eintritt in den krit. Bereich ausfuehren */
void enter_region (int process) {    /* process: wer tritt ein: 0,1 */
    int other;                        /* Nummer des anderen Prozesses */
    other = 1-process;
    interested[process] = true;       /* zeige eigenes Interesse */
    turn = process;                  /* setze Marke, unteilbar! */
    while (turn==process && interested[other]) ;
        /* ev. Aktives Warten !!! */
}

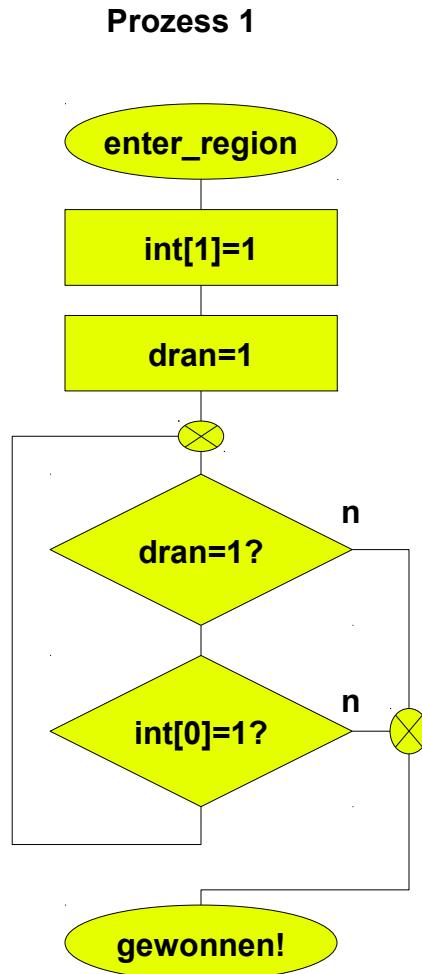
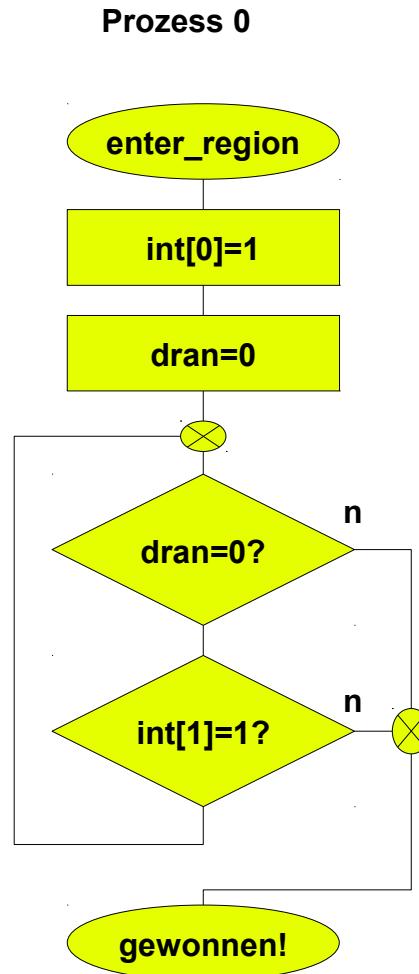
/* Epilog-Operation, nach Austritt aus dem krit. Bereich ausfuehren */
void leave_region (int process) {    /* process: wer verlaesst: 0,1 */
    interested[process] = false;      /* Verlassen des krit. Bereichs */
}

```



Algorithmus





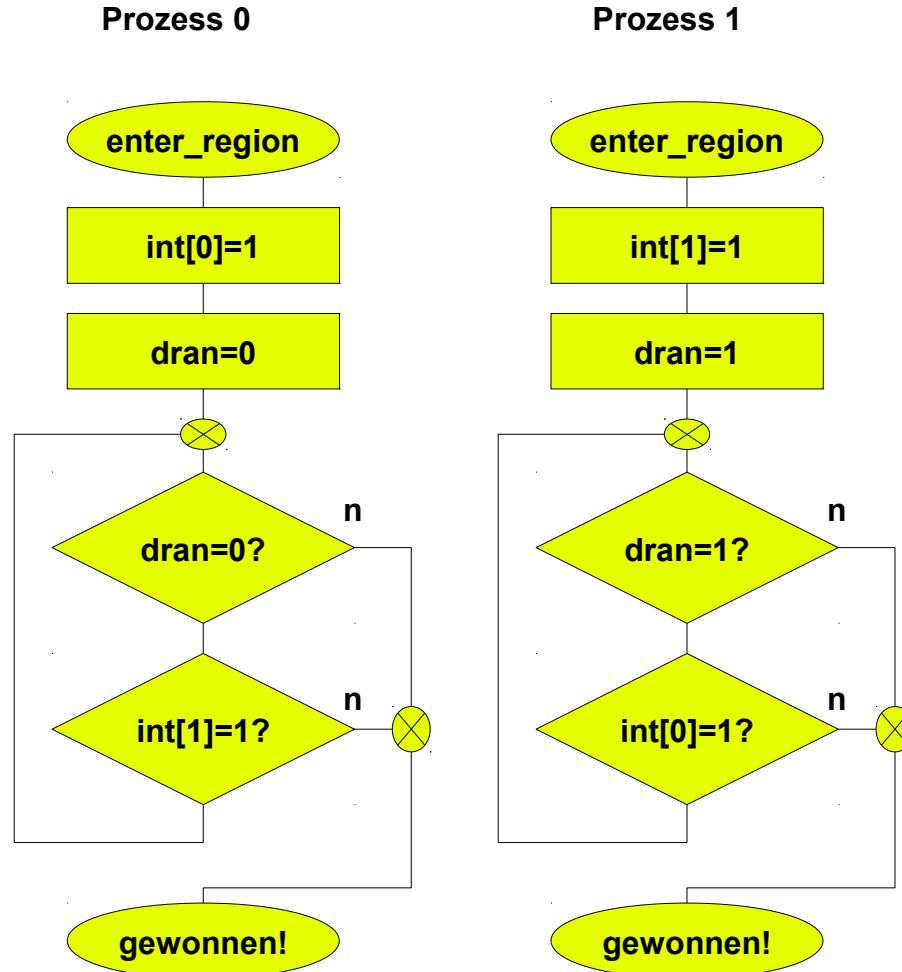
**Prozess 0 betritt krit. Sektion
Prozess 1 nicht**

interested[]:

1	0
---	---

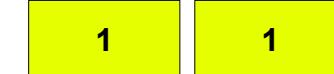
dran:

0



Prozess 0 ist in krit. Sektion, Prozess 1 versucht einzutreten

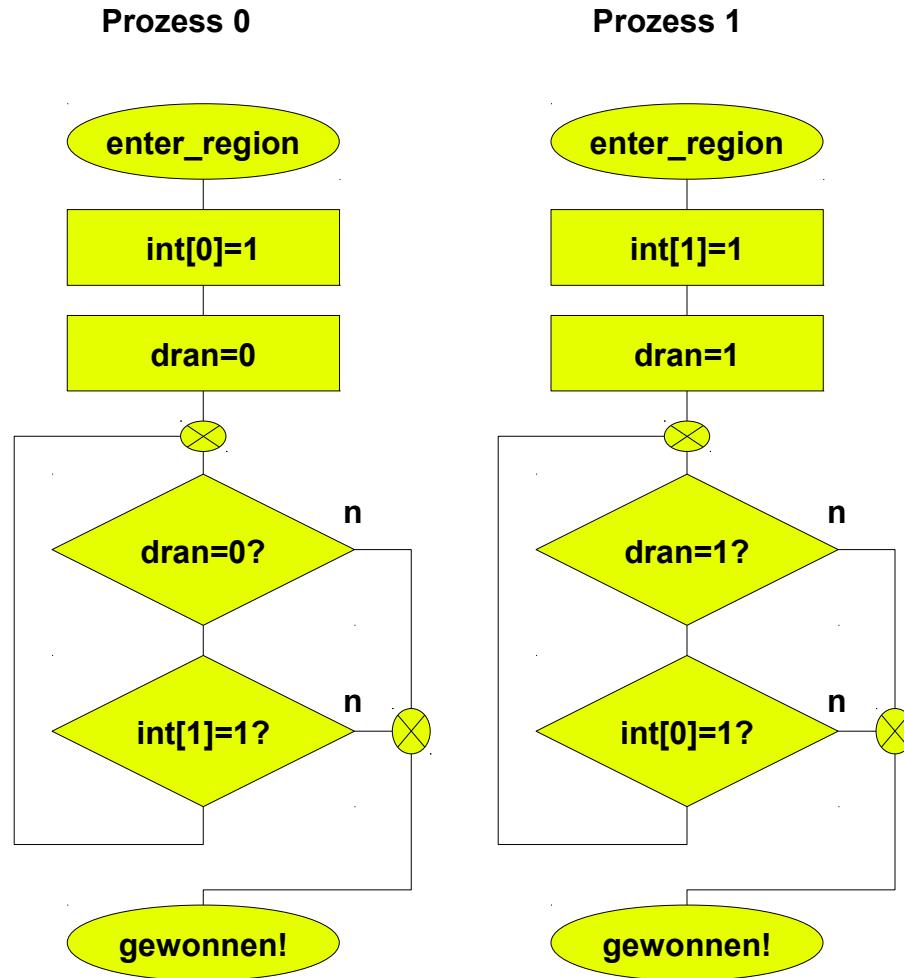
interested[]:



dran:

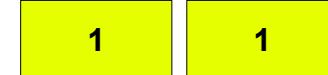


=> Prozess 1 wartet bis int[0] = 0.



Kritischer Fall:
Beide versuchen
gleichzeitig, krit.
Sektion zu betreten

interested[]:



dran:



Schreibzugriff auf dran ist atomar => nur einer von beiden kann dran erfolgreich setzen!
Wer Erfolg hat, muss hier warten!

5.2.1.5 Test-and-Set - Instruktion



- Algorithmen zu komplex, die nur auf atomaren Speicherzugriffen basieren (siehe oben).
⇒ Notwendigkeit für effiziente Lösung
- Lösung durch Hardware-Unterstützung:
Einführung eines Maschinenbefehls Test-and-Set:
unteilbares Lesen einer Speicherzelle mit anschließendem Schreiben eines Wertes in die Zelle
(Speicherbus wird dazwischen nicht freigegeben).
- Heute
 - Standard auf praktisch allen Architekturen (mit geringen Abweichungen)
 - notwendig für Multiprozessor-Systeme
 - häufig zusätzliche unteilbare Maschinen-Operationen zur Listenmanipulation (z.B. für Ready-Queue, MC680x0)
 - oder LOCK/UNLOCK-Paare, um kurze Folgen beliebiger Instruktionen atomar auszuführen (z.B. i860).

- Jedem kritischen Abschnitt wird eine Sperrvariable `flag` zugeordnet. Wert 0 bedeutet "frei", Wert 1 bedeutet "belegt".
- Auf Test-and-Set basierende Sperrvariablen mit aktivem Warten heißen auch spin locks. Hohe Bedeutung in Multiprozessor-Betriebssystemen.
- Typische Assembler-Routinen für Prolog und Epilog mit Test-and-Set - Instruktion `TSL`:

```
enter_region:
    tsl register,flag    | kopiere flag in register und setze flag auf 1
    cmp register,#0      | war flag vorher Null?
    jnz enter_region    | nein, d.h. Sperre war gesetzt, aktiv warten!
    ret                  | ja, darf Eintreten, zurueck zum Aufrufer

leave_region:
    mov flag,#0          | loesche flag, d.h. setze Sperre zurueck
    ret                  | zurueck zum Aufrufer
```

- Im Gegensatz zur Darstellung hier erfolgt die Realisierung i.d.R. über Makros, da Prozeduraufruf zur Laufzeit zu großen Overhead darstellt.

- Dieses Problem kann trotz korrekter Lösungen mit aktivem Warten (Peterson, Test-and-Set, s.o.) auftreten.
- Prozess H habe hohe Priorität, Prozess L habe niedere Priorität. **Scheduling-Regel:** wenn H rechenwillig ist, soll H ausgeführt werden.
- Es werde H rechenwillig, während L sich in seinem kritischen Abschnitt befindet. H wolle ebenfalls in seinen kritischen Abschnitt eintreten.
- Wird nun der höher priore Prozess H ausgeführt, so wartet er aktiv für immer, da L seinen kritischen Abschnitt nicht beenden kann.
- Es muss also der niedere priore Prozess ausgeführt werden ! Diese Situation heißt auch Prioritätsinversionsproblem.
- (Darüberhinaus tritt das Problem in Lösungen mit passivem Warten ebenfalls auf. Insbesondere in Echtzeitbetriebssystemen ist eine Lösung hierfür extrem wichtig.)

Bisher:

- Prolog-Operationen zum Betreten des kritischen Abschnitts führen zum Aktiven Warten, bis der betreffende Prozess in den krit. Abschnitt eintreten kann.
- Lediglich auf Multiprozessor-Systemen kann kurzzeitiges Aktives Warten zur Vermeidung eines Prozesswechsels sinnvoll sein (spin locks).

Ziel: Vermeidung von verschwendeter Prozessorzeit.

Vorgehensweise:

- Prozesse blockieren, wenn sie nicht in ihren kritischen Abschnitt eintreten können.
- Ein blockierter Prozess wird durch den aus seinem krit. Abschnitt austretenden Prozess entblockiert.

- Einfachste Primitive werden als SLEEP und WAKEUP bezeichnet.
- SLEEP() blockiert den ausführenden Prozess, bis er von einem anderen Prozess geweckt wird.
- WAKEUP(*process*) weckt den Prozess *process*. Der ausführende Prozess wird dabei nie blockiert.
- Häufig wird als Parameter von SLEEP und WAKEUP ein Ereignis (Speicheradresse einer beschreibenden Struktur) verwendet, um die Zuordnung treffen zu können, (vgl. 3.2 Beispiel UNIX(7) Kernroutinen).
- Diese Primitive können auch der allgemeinen ereignisorientierten Kommunikation dienen.

- NT Critical Sections erlauben den wechselseitigen Ausschluss von Threads eines Prozesses (nicht über einen Adressraum hinaus) mittels passivem Warten.
- `CRITICAL_SECTION cs;`
`void InitializeCriticalSection(LPCRITICAL_SECTION cs)`
`void DeleteCriticalSection(LPCRITICAL_SECTION cs)`
Definition, Initialisierung und Zerstörung einer Critical Section-Variable.
- `void EnterCriticalSection(LPCRITICAL_SECTION cs)`
Prolog zum Betreten eines kritischen Abschnitts
- `void LeaveCriticalSection(LPCRITICAL_SECTION cs)`
Epilog zum Verlassen eines kritischen Abschnitts
- `BOOL TryEnterCriticalSection(LPCRITICAL_SECTION cs)`

Versuch des Betretens eines kritischen Abschnitts ohne Blockierung.

- Der Begriff Mutex ist von mutual exclusion abgeleitet.
- Ein Mutex offeriert Operationen
 - *lock* als Prolog-Operation zum Betreten des kritischen Abschnitts
 - *unlock* als Epilog-Operation beim Verlassen des kritischen Abschnitts
- Mutexe können als Spezialfall von Semaphoren angesehen werden (vgl. 5.2.3).
- Gelegentlich wird angenommen, dass unlock *alle* wartenden Prozesse entblockiert und sich diese dann erneut um das Betreten des kritischen Abschnitts bewerben.

```
#include <pthread.h>

pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
Anlegen einer Mutex-Variablen (mehrere Varianten)

pthread_mutex_init(pthread_mutex_t *mutex,
                   const pthread_mutexattr_t *mutexattr);
Initialisieren einer Mutex-Variablen

int pthread_mutex_lock(pthread_mutex_t *mutex);
Lock anfordern

int pthread_mutex_trylock(pthread_mutex_t *mutex);
Lock anfordern, falls ohne Blockieren möglich

int pthread_mutex_unlock(pthread_mutex_t *mutex);
Lock freigeben

int pthread_mutex_destroy(pthread_mutex_t *mutex);
Lock zerstören
```

Windows NT offeriert Mutexe als Systemobjekte für den wechselseitigen Ausschluss zwischen Threads bei. Prozesse.

Mutex-Operationen:

- `HANDLE CreateMutex (` **Erzeugen neues Mutex.**
 `LPSECURITY_ATTRIBUTES security,`
 `BOOL FInitialOwner,`
 `LPSTR name);`
- `HANDLE OpenMutex (` **Öffnen exist. Mutex.**
 `DWORD access,`
 `BOOL inherit,`
 `LPCTSTR name);`
- `DWORD WaitForSingleObject (` **lock**
 `HANDLE hmutex,` **wird allg. zur Synch. verwendet**
 `DWORD Timeout);`
- `BOOL ReleaseMutex (` **unlock**
 `HANDLE hsem);`
- **Bemerkung:** NT Mutexe werden von Threads in "Besitz" genommen: Nur der Thread, der `WaitForSingleObject` erfolgreich ausführte, kann anschließend `ReleaseMutex` ausführen. Nach erfolgreichem lock sind weitere Wait-Aufrufe desselben Threads ebenfalls erfolgreich (Idempotenz).

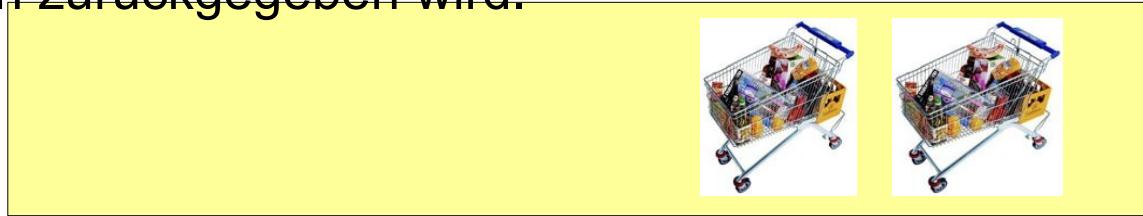
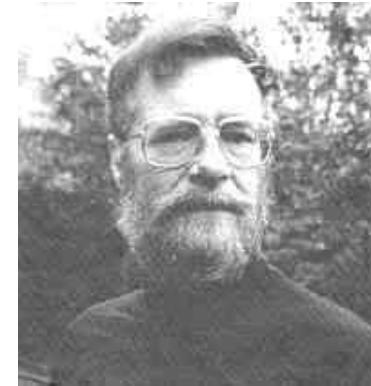
Semaphoren



1965 von Edsger W. Dijkstra eingeführt

Supermarkt-Analogie:

- Kunde darf den Laden nur mit einem Einkaufswagen betreten
- es steht nur eine bestimmte Anzahl von Einkaufswagen bereit
- sind alle Wagen vergeben, müssen neue Kunden warten, bis ein Wagen zurückgegeben wird.



Semaphor besteht aus

- einer **Zählvariablen**, die begrenzt, wieviele Prozesse augenblicklich ohne Blockierung passieren dürfen
- und einer **Warteschlange** für (passiv) wartende Prozesse

5.2.3 Semaphor-Operationen



!

Initialisierung

- Zähler auf initialen Wert setzen
- "Anzahl der freien Einkaufswagen"



Operation P () : Passier(-Wunsch)

- Zähler = 0: Prozess in Warteschlange setzen, blockieren
- Zähler > 0: Prozess kann passieren
- In *beiden* Fällen wird der Zähler (ggf. nach dem Ende der Blockierung) erniedrigt
- P steht für "proberen" (niederländisch für "testen")

Operation v () : Freigeben

- Zähler wird erhöht
- Falls es Prozesse in der Warteschlange gibt, wird einer de-blockiert (und erniedrigt den Zähler dann wieder, s.o.)
- v steht für „verhogen“ (niederländisch für "erhöhen")

P(s) oder DOWN(s):

- $i > 0 \Rightarrow i := i - 1$; Prozess fährt fort.
- $i = 0 \Rightarrow$ ausführender Prozess blockiert.
(Er legt sich am Semaphor schlafen).
- Wichtig: Atomarität bedeutet, dass Überprüfen des Wertes, Verändern des Wertes und sich Schlafen legen als eine einzige, unteilbare Operation ausgeführt werden.
Dieses vermeidet zeitkritische Abläufe.

V(s) oder UP(s):

$i := i + 1$;

Wenn es blockierte Prozesse gibt, wird einer von ihnen ausgewählt.
Er kann seine P-Operation jetzt erfolgreich beenden.
Das Semaphor hat in diesem Fall dann weiterhin den Wert Null, aber es gibt einen Prozess weniger, der an dem Semaphor schläft.
Erhöhen des Semaphor-Wertes und ev. Wecken eines Prozesses werden ebenfalls als unteilbare Operation ausgeführt.
Kein Prozess wird bei der Ausführung einer V-Operation blockiert.

- **Semaphore, die nur die Werte 0 und 1 annehmen, heißen binäre Semaphore, ansonsten heißen sie Zählsemaphore.**
- **Binäre Semaphore zur Realisierung des wechselseitigen Ausschlusses.**
- **Ein mit $n > 1$ initialisiertes Zählsemaphor kann zur Kontrolle der Benutzung eines in n Exemplaren vorhandenen Typs von sequentiell wiederverwendbaren Betriebsmitteln verwendet werden.**

Bemerkungen:

- **Semaphore sind weit verbreitet**
 - innerhalb von Betriebssystemen
 - zur Programmierung nebenläufiger Anwendungen
- **Programmierung mit Semaphoren ist oft fehleranfällig, wenn mehrere Semaphore benutzt werden müssen (vgl. 5.3).**
- **Mutex-locks (vgl. 5.2.2) können als binäre Semaphore angesehen werden.**

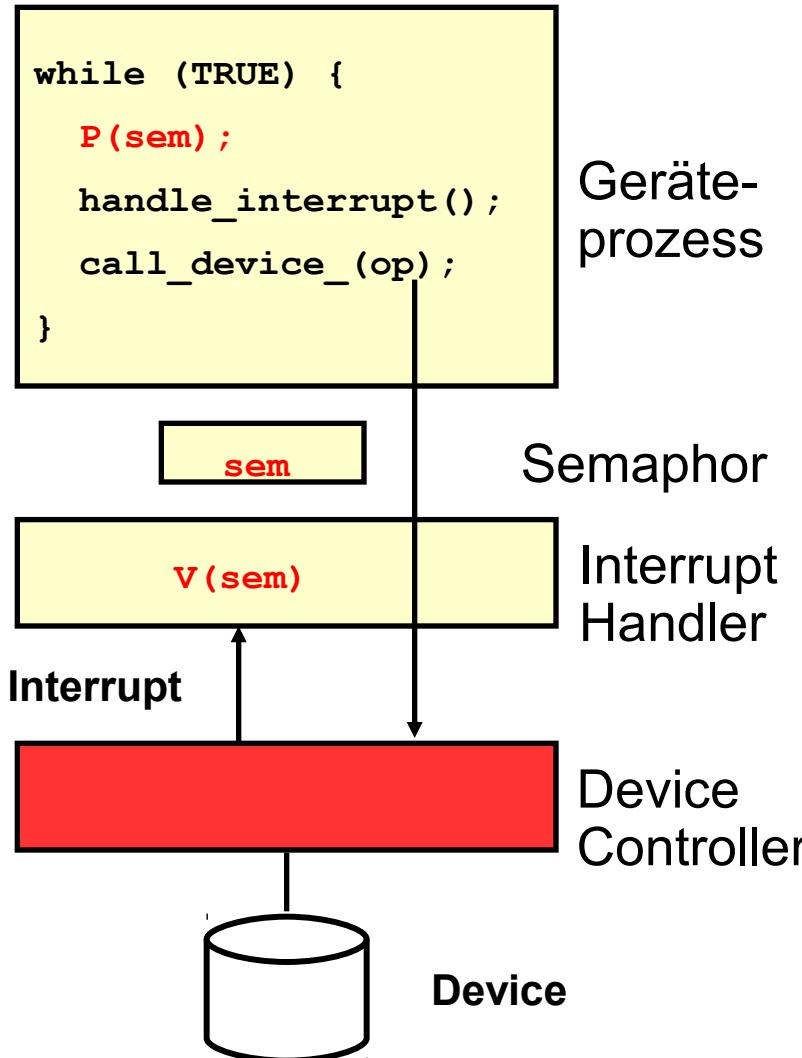
```
semaphore sem = 1; /* Init. mit 1 */

...
P(sem); /* Prolog */
<statement> ;
...
<statement> ;
V(sem); /* Epilog */
...
```

geklammerter
kritischer Abschnitt

- Das mit 1 initialisierte Semaphor `sem` wird von allen beteiligten Prozessen benutzt.
- Jeder Prozess klammert seinen kritischen Abschnitt, mit `P(sem)` zum Eintreten und `V(sem)` zum Verlassen.

Anwendung: Verstecken von Interrupts



- Jedem I/O-Gerät wird ein Geräteprozess und ein mit 0 initialisiertes Semaphor zugeordnet.
- Nach dem Start des Geräteprozesses führt dieser eine P-Operation auf dem Semaphor aus (und blockiert).
- Im Falle eines Interrupts des Gerätes führt der Interrupt Handler eine V-Operation auf dem Semaphor aus. Der Geräteprozess wird entblockiert (und dadurch rechenwillig) und kann das Gerät bedienen.

Anwendung: Betriebsmittelvergabe



- Es seien n Exemplare vom Betriebsmitteltyp bm vorhanden.
- Jedes BM dieses Typs sei sequentiell benutzbar.

```
semaphore bm = n;  
  
P (bm) ;          /*      Beantragen      */
```

Benutze das Betriebsmittel

```
V (bm) ;          /*      Freigeben      */
```

- Dem BM-Typ bm wird ein Semaphor bm zugeordnet.
- Beantragen eines BM durch $P(bm)$,
ev. mit Blockieren, bis ein BM verfügbar wird.
- Nach der Benutzung freigeben des BM durch $V(bm)$.
(Es kann damit von einem weiteren Prozess benutzt werden).

- Sei P eine Menge von kooperierenden Prozessen (Prozesssystem), und sei $<$ eine partielle Ordnung auf der Menge der Prozesse mit
 - $P_1 < P_2 : \Leftrightarrow$ Prozess P_1 muss vor P_2 ausgeführt werden;
 - $<$ wird Vorrangrelation genannt.In einem Graph wird $P_1 < P_2$ häufig durch $P_1 \rightarrow P_2$ dargestellt.
- Jeder Vorrangbeziehung $P_1 < P_2$ wird ein Semaphor s zugeordnet, auf das $P_1 \text{ v}(s)$ und $P_2 \text{ p}(s)$ ausführt.
- Alle Semaphore werden mit 0 initialisiert.
⇒ Alle Prozesse können gleichzeitig gestartet werden. Die Vorrangrelation zwischen ihnen wird dennoch korrekt durchgesetzt.

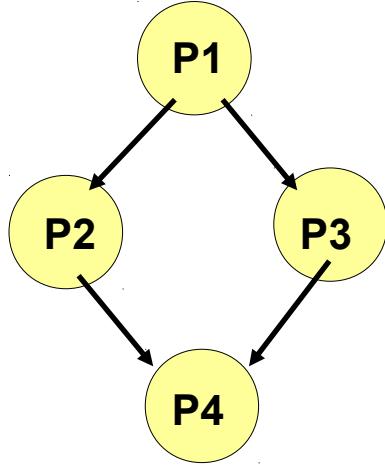
Beispiel: Vorrangrelation

5.2.3

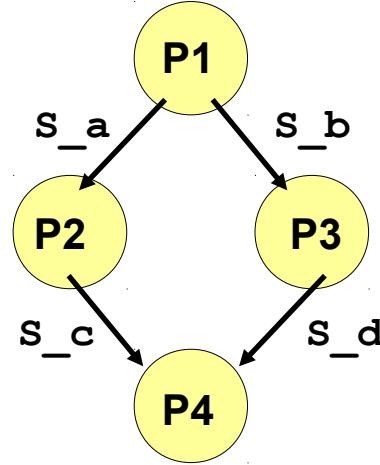


Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

gegebenes Prozesssystem:



Lösung:



Initialisieren aller
Semaphore mit 0

```
P1() {  
    ...work...  
  
    V(S_a);  
  
    V(S_b);  
    exit();  
}
```

```
P2() {  
    P(S_a);  
    ...work...  
  
    V(S_c);  
    exit();  
}
```

```
P3() {  
    P(S_b);  
    ...work...  
  
    V(S_d);  
    exit();  
}
```

```
P4() {  
    P(S_c);  
  
    P(S_d);  
    ...work...  
    exit();  
}
```



- **Üblich als Systemaufrufe.**
Intern Nutzung von Sich-Schlafen-Legen und Aufwecken.
- **Wesentlich ist die Unteilbarkeit der Implementierung von P() und V():**
 - **Auf Einprozessorsystemen:**
Unteilbarkeit kann durch Sperren aller Unterbrechungen während der Ausführung von P() und V() erreicht werden. Zulässig, da nur wenige Maschineninstuktionen zur Implementierung nötig sind.
 - **Auf Multiprozessorsystemen:**
Jedem Semaphor wird eine mittels Test-and-Set realisierte Sperrvariable mit Aktivem Warten vorgeschaltet. Hierdurch kann zu jedem Zeitpunkt nur höchstens ein Prozessor das Semaphor manipulieren.
 - **Beachte: Unterscheide zwischen Aktivem Warten auf den Zugang zum Semaphor, das einen kritischen Abschnitt schützt (einige Instruktionen, Mikrosekunden) und Aktivem Warten auf den Zugang zum kritischen Abschnitt selbst (problemabhängig, Zeitdauer nicht vorab bekannt oder begrenzt).**

- **System Calls zum Umgang mit Semaphoren**
(gehören zum System V IPC-Mechanismus)
- **UNIX-Semaphore sind in der Benutzung erheblich komplexer**
als oben beschrieben.
- **Es werden Gruppen von Semaphoren betrachtet.**
Mit einem Operationsaufruf kann eine Teilmenge dieser
Semaphoren atomar verändert werden, d.h. alle Operationen
werden ausgeführt oder keine und Blockierung.
- **Über Flags werden Varianten gesteuert, z.B.**
 - nicht blockierender Aufruf mit Fehlercode
(Vermeidung einer Blockierung)
 - undo aller Effekte von Semaphor-Operationen bei Beendigung
eines Prozesses (Vermeidung von Inkonsistenzen).



Operationen:

- `int semget(key_t key, int nsems, int semflag):`
liefert den zum externen `key` gehörenden Id der Semaphore-Gruppe und macht so eine exist. Gruppe zur Benutzung in einem Prozess zugänglich. Wird `key` auf `IPC_PRIVATE` gesetzt oder ein neuer Schlüssel verwendet und in `semflag` das `IPC_CREAT`-Bit gesetzt und existiert noch keine Gruppe zu `key`, so wird eine neue Semaphore-Gruppe mit `nsems` Semaphoren erzeugt.
- `int semop(int semid, struct sembuf *sops[], int nsops)`
zur Ausführung von Semaphore-Operationen auf der Semaphore-Gruppe `semid`. Die durchzuführenden Operationen werden in einem Vektor `sops` der Länge `nsops` von `sembuf`-Strukturen beschrieben:

```
struct sembuf {  
    ushort sem_num;      /* # des Sem. in Gruppe */  
    short  sem_op;       /* Op.: <0=P(), >0=V() */  
    short  sem_flg; }   /* IPC_NOWAIT, SEM_UNDO */
```

Für jedes Semaphore kann damit individuell die durchzuführende Operation festgelegt werden.

- `int semctl(int semid, int semnum, int cmd, union semun arg)`
Kontrolloperationen auf Semaphoren zum Lesen, Setzen, Löschen,

wird im Praktikum vertieft.

Semaphore-Beispiel

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY 424242
#define SEMPERM 0777
#define PROCS 5

struct sembuf op_lock = { 0, -1, 0 }; /* Sem 0 erniedr.*/
struct sembuf op_unlock = { 0, 1, 0 }; /* Sem 0 erhöhen */

int main(void) {
    int i, j, semid, status, initwert=1;
    if ((semid = semget(SEMKEY, 1, IPC_CREAT | SEMPERM)) < 0)
    {
        perror("semget");
        exit(1);
    }
    semctl(semid, 0, SETALL, &initwert);
    ...
}
```

**Ziel: 5 parallele Prozesse zählen
nacheinander sekundenweise
von 0...4 (keine Überschneidungen)**

Semaphore-Beispiel (2)

```
for (i=0; i < PROCS; i++) {  
    if (!fork()) {  
        if (semop(semid, &op_lock, 1) < 0) {  
            perror("semop op_lock"); exit(1);  
        }  
        for (j=0; j < 5; j++) {  
            fprintf(stderr,"Prozess %d mit j=%d\n",getpid(), j);  
            sleep(1);  
        }  
        if (semop(semid, &op_unlock, 1) < 0) {  
            perror("semop op_unlock"); exit(1);  
        }  
        exit(0);  
    }  
    for (j=0; j < PROCS; j++) wait(&status);  
    semctl(semid, 0, IPC_RMID, 0);  
    return 0;  
}
```

kritisch

Sohnprozess, 5x

- Windows NT offeriert Zählsemaphore als Systemobjekte für die Synchronisation von Threads beliebiger Prozesse.
- Operationen:
 - `HANDLE CreateSemaphore (` Erzeugen eines neuen Semaphors

`LPSECURITY_ATTRIBUTES security,`
`LONG initialValue,`
`LONG maxValue,`
`LPCTSTR name);`
 - `HANDLE OpenSemaphore (` Öffnen eines existierenden Semaphors

`DWORD access,`
`BOOL inherit,`
`LPCTSTR name);`
 - `DWORD WaitForSingleObject (` entspricht P(), aber allg. verwendbar.
wie auch WaitForMultipleObjects()

`HANDLE hsem,`
`DWORD Timeout);`
 - `BOOL ReleaseSemaphore (` entspricht V()

`HANDLE hsem,`
`LONG incrementValue,`
`LPLONG prevCount);`

Monitore:

- Vorschlag Hoare (1974), Brinch Hansen (1975):
- In Programmiersprache eingebettete Primitive zur einfacheren Entwicklung korrekter nebenläufiger Programme.
- Monitor besteht aus
 - Variablen und Datenstrukturen
(die das eigentliche Betriebsmittel repräsentieren)
 - Sichtbaren Operationen zur Manipulation der Daten.
 - Wechselseitige Ausschluss in der Ausführung der Operationen ist garantiert, d.h. zu jedem Zeitpunkt ist höchstens ein Prozess im Monitor aktiv.
- Code zur Sicherstellung des wechselseitigen Ausschlusses wird durch den Compiler erzeugt, z.B. auf der Basis von Semaphoren.
- Heutige Sicht:
 - Monitor entspricht einer Instanz eines abstrakten Datentyps mit automatischem wechselseitigen Ausschluss
 - Vergleiche Java „synchronized instance method“

- Programmierung von Synchronisationsvorgängen innerhalb von Monitoren mit internen Bedingungsvariablen (condition variables) mit Operationen WAIT und SIGNAL.
- WAIT(c) blockiert den Aufrufer, ein ev. wartender anderer Aufrufer einer Monitor-Operation kann nun den Monitor betreten.
- SIGNAL(c) weckt einen Prozess (aus der Sicht des Monitors zufällig aus der Menge der Wartenden ausgewählt), der Aufrufer muss den Monitor sofort verlassen (Annahme: letzte Anweisung, Brinch Hansen-Modell). Falls kein Prozess wartet, ist SIGNAL(c) ohne Wirkung (keine Zähler-Semantik!).
- Monitore haben kaum Eingang in Programmiersprachen gefunden (Gegenbeispiele: Mesa (Xerox), eingeschränkt Java synchronized)).

Gedachte Pascal-Erweiterung

```
monitor example

    integer i;
    condition c;

    procedure producer(x);
    ...
    end;

    procedure consumer(x)
    ...
    end;

end monitor;
```

- Synchronisation von Threads basierend auf Bedingung über aktuellem Wert einer Variable
- Benutzung immer im Zusammenhang mit assoziiertem Mutex
- Condition Variable
 - deklarieren vom Typ `pthread_cond_t`
 - `pthread_cond_init()` (Initialisieren)
 - `pthread_cond_destroy` (Zerstören)
- Warten und Signalisieren
 - `pthread_cond_wait(condition,mutex)` Blockieren bis Bedingung signalisiert wird
 - `pthread_cond_signal(condition)` Signalisieren der Bedingung (Wecken mindestens eines anderen Threads)
 - `pthread_cond_broadcast(condition)` Signalisieren der Bedingung (Wecken aller wartenden Threads)
 - Details: <https://computing.llnl.gov/tutorials/pthreads/>

Erläuterungen siehe Man Pages:

- `pthread_cond_wait()` blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking mutex when the thread is finished with it.
- The `pthread_cond_signal()` routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for `pthread_cond_wait()` routine to complete.
- The `pthread_cond_broadcast()` routine should be used instead of `pthread_cond_signal()` if more than one thread is in a blocking wait state.
- It is a logical error to call `pthread_cond_signal()` before calling `pthread_cond_wait()`.

Weitere, hier nicht besprochene Synchronisationsprimitive:

- Eventcounts: Reed, Kanodia (1979).
- Serializer: Atkinson, Hewitt (1979).
- Objekte mit Pfadausdrücken (path expressions) zur Festlegung von zulässigen Ausführungsfolgen der Operationen einschl. deren Nebenläufigkeit: Campbell, Habermann (1974).
- Read/Write-Locks (vgl. Datenbanken).
- Barrieren.
- fork/join.

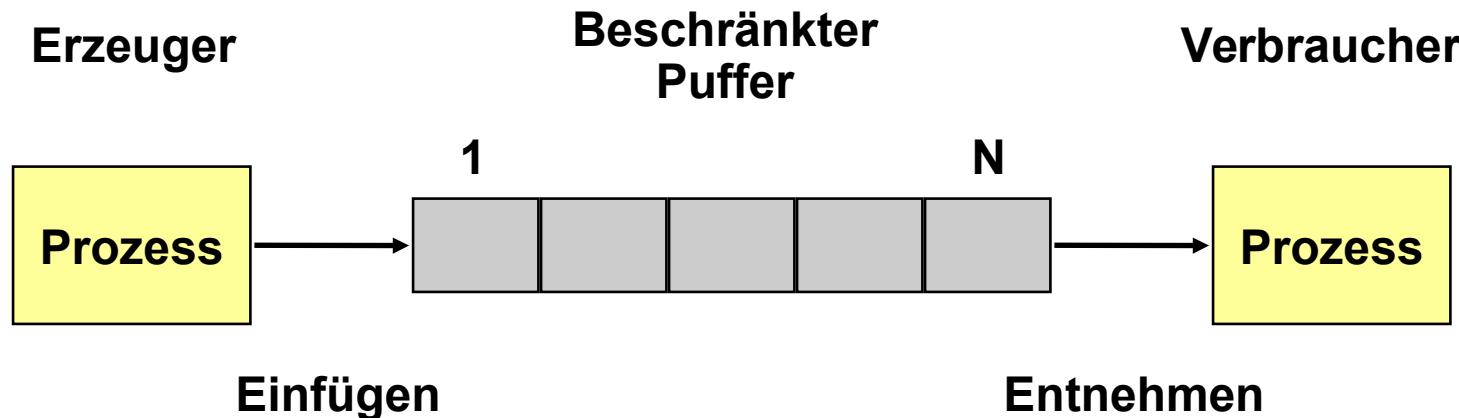
Literatur hierzu:

z.B. Tanenbaum, Peterson/Silberschatz, Maekawa et. al.
(siehe Literaturliste).

Gliederung

1. Erzeuger-Verbraucher-Problem
2. Philosophen-Problem
3. Leser-Schreiber-Problem

5.3.1. Erzeuger-Verbraucher-Problem



Probleme:

- **Erzeuger:** Will Einfügen, aber Puffer ist voll.
Lösung: Lege dich schlafen, lass dich vom Verbraucher wecken, wenn er ein Datum entnommen hat.
- **Verbraucher:** Will Entnehmen, aber Puffer ist leer.
Lösung: Lege dich schlafen, lass dich vom Erzeuger wecken, wenn er ein Datum eingefügt hat.

- Problem wird auch "Problem des beschränkten Puffers" (bounded buffer problem) genannt.
- Sieht einfach aus, enthält aber kritische Abläufe.
- Erweiterung: mehrere Erzeuger, mehrere Verbraucher.

Im weiteren werden folgende Lösungsansätze betrachtet:

1. Lösung mit Semaphoren
2. Lösung mit Monitoren
3. Lösung mit Nachrichtenaustausch

5.3.1.1. Lösung mit Semaphoren



```
#define N 100                                /* Kapazitaet des Puffers      */  
  
/* gemeinsame Variablen                      */  
  
semaphore mutex = 1;                          /* kontroll. kritischen Bereich */  
semaphore empty = N;                          /* zaehlt leere Eintraege      */  
semaphore full  = 0;                          /* zaehlt belegte Eintraege    */  
  
void producer(void) {                         /* Erzeuger                    */  
    int item;  
    while(TRUE) {  
        produce_item(&item);  
        down(&empty);  
        down(&mutex);  
        enter_item(item);  
        up(&mutex);  
        up(&full);  
    }  
}
```

Lösung mit Semaphoren (2)

```
void consumer(void) {                                /* Verbraucher */  
    int item;  
    while(TRUE) {  
        down(&full);                                /* belegter Eintrag vorhanden? */  
        down(&mutex);  
        remove_item(&item);  
        up(&mutex);  
        up(&empty);  
        consume_item(item);  
    }  
}
```

- **Beachte:** `down()` entspricht `P()`, `up()` entspricht `V()`.
- **Semaphore mutex** wird zur Durchsetzung des wechselseitigen Ausschlusses in der Benutzung des Puffers verwendet.
- **Semaphore empty** und **full** werden zur Synchronisation von Erzeuger und Verbraucher benutzt, um bestimmte Operationsreihenfolgen zu erreichen bzw. zu vermeiden.
Beispiel: Erzeuger: `down(&empty)`, Verbraucher: `up(&empty)` realisiert: Wenn der Puffer voll ist, muss zuerst ein Eintrag entnommen werden, bevor ein neuer eingefügt werden kann.
- Die Algorithmen für Erzeuger und Verbraucher gelten unverändert, wenn mehrere Erzeuger und/oder mehrere Verbraucher zugelassen werden.

5.3.1.2. Lösung mit Monitoren



```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure enter;
    begin
        if count=N then wait(full);
        enter_item;
        count:=count+1;
        if count=1 then signal(empty);
    end;

    procedure remove;
    begin
        if count=0 then wait(empty);
        remove_item;
        count:=count-1;
        if count=N-1 then signal(full);
    end;

    count:=0;
end monitor;
```

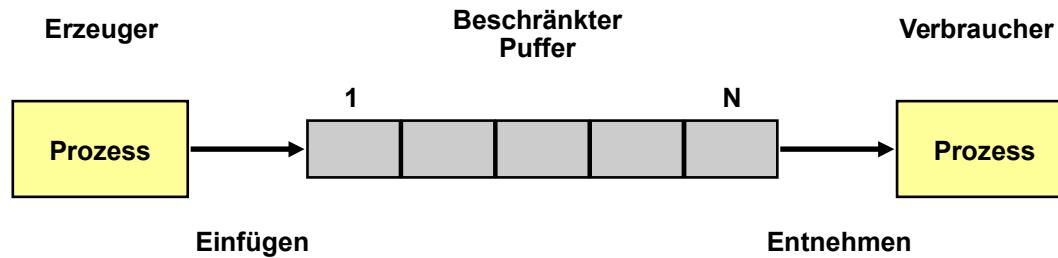
(hypothetische Pascal-Erweiterung)

**Der Erzeuger muß warten,
solange der Puffer voll ist**

**Ein wegen der "empty"-
Bedingung wartender Prozess,
könnte nun weitermachen**

**Der Verbraucher muß warten,
solange der Puffer leer ist**

**Ein wegen der "full"-Bedingung
blockierter Prozess könnte nun
weitermachen**



```
procedure producer;
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter;
  end
end;

procedure consumer;
begin
  while true do
  begin
    ProducerConsumer.remove;
    consume_item;
  end
end;
```

- **Beachte:** Die Ausführung der Operationen `enter` und `remove` ist wechselseitig ausgeschlossen (Monitor!).
- Die Variable `count` gibt die augenblickliche Anzahl der belegten Pufferplätze an.
- Die Variablen `full` und `empty` sind Bedingungsvariablen, keine Zähler.
- Die Algorithmen für Erzeuger und Verbraucher gelten ebenfalls unverändert, wenn mehrere Erzeuger und/oder mehrere Verbraucher zugelassen werden.

5.3.1.3. Lösung mit Nachrichtenaustausch



```
#define N      100          /* Kapazitaet des Puffers      */
#define MSIZE   4           /* Nachrichtengroesse          */

typedef int message[MSIZE];

void producer(void) {           /* Erzeuger                   */
    int item;
    message m;

    while(TRUE) {
        produce_item(&item);    /* erzeuge Eintrag            */
        receive(consumer, &m);   /* warte auf leere Nachricht */
        build_message(&m, item); /* erzeuge zu sendende Nachricht */
        send(consumer, &m);     /* sende Nachricht z Verbraucher */
    }
}
```

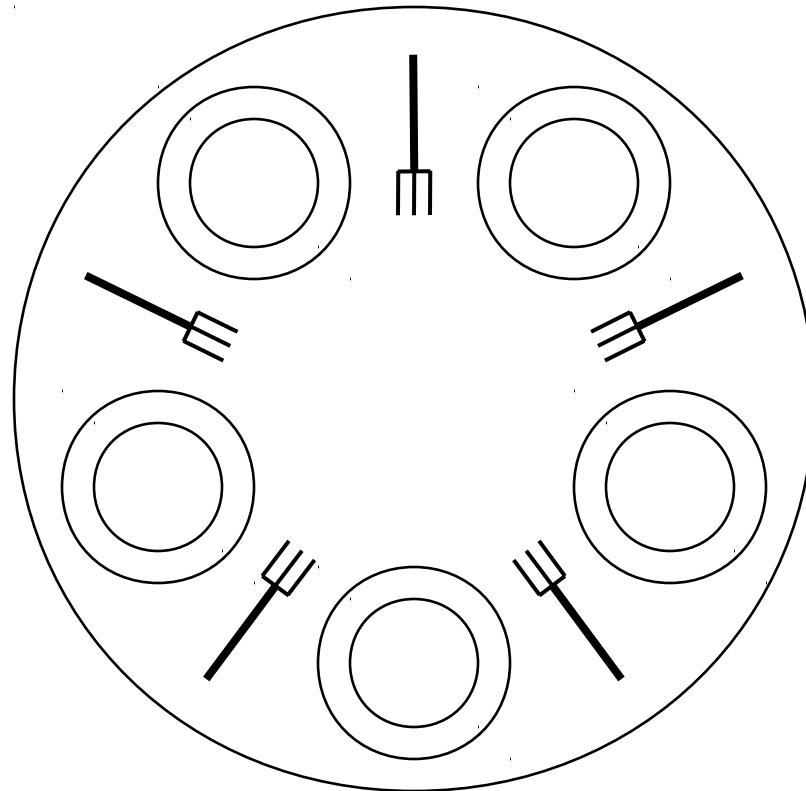
Lösung mit Nachrichtenaustausch (2)



```
void consumer(void) {                                /* Verbraucher */  
    int item, i;  
    message m;  
  
    for (i=0; i<N; i++)                            /* sende N leere Nachrichten */  
        send (producer, &m);  
    while(TRUE) {  
        receive(producer, &m);                      /* empfange Nachricht v Erzeuger */  
        extract_item(&m, &item);                     /* entnimm Eintrag */  
        send(producer, &m);                          /* sende leere Nachricht zurueck */  
        consume_item(item);                         /* verarbeite Eintrag */  
    }  
}
```

- **Beachte:** Kein gemeinsamer Speicher zwischen Erzeuger und Verbraucher.
- **Annahmen:** Alle Nachrichten haben dieselbe feste Länge. Gesendete, aber noch nicht empfangene Nachrichten werden innerhalb des Nachrichtensystems automatisch gepuffert.
- Es werden insgesamt N Nachrichtenhülsen (Umschläge) benutzt. Leere Nachrichten werden vom Verbraucher an den Erzeuger gesendet, der Erzeuger füllt leere Nachrichten mit erzeugten Einträgen und sendet "gefüllte Umschläge" an den Verbraucher.
- Die Algorithmen für Erzeuger und Verbraucher sind aufgrund der direkten Adressierung auf einen Erzeuger und einen Verbraucher zugeschnitten.

5.3.2. Philosophen-Problem



- Dijkstra (1965): dining philosophers problem.
- **5 Philosophen sitzen am runden Tisch. Jeder hat einen Teller, zwischen je zwei benachbarten Tellern liegt eine Gabel. Linke und rechte Gabel werden zum Essen benötigt. Nach dem Essen werden beide Gabeln abgelegt. Essen und Denken wechseln einander fortlaufend ab.**
- **Wie lautet der Algorithmus für jeden Philosophen?**

Eine zu einfache, fehlerhafte Lösung



```
#define N 5                                /* Anzahl der Philosophen      */

void philosopher(int i)          /* i:0..N-1, welcher Philosoph */
{
    while (TRUE) {
        think();                            /* Denken                      */
        take_fork(i);                      /* Greife linke Gabel          */
        take_fork((i+1)%N);                /* Greife rechte Gabel         */
        eat();                               /* Essen                       */
        put_fork(i);                        /* Ablegen linke Gabel         */
        put_fork((i+1)%N);                /* Ablegen rechte Gabel        */
    }
}
```

Annahme: Alle Philosophen greifen die linke Gabel durch `take_fork(i)`.
Dann sind alle für immer blockiert. Es liegt ein sogenannter Deadlock vor.

- Verbesserung: Nach dem erfolgreichen Aufnehmen der linken Gabel überprüfe, ob die rechte Gabel verfügbar ist. Falls nicht, lege die linke Gabel ab, warte eine Zeitlang, dann beginne von vorn.
- Dieser Ansatz vermeidet den alten Fehler (Deadlock), enthält aber einen neuen:
Alle Philosophen greifen gleichzeitig die linke Gabel, erkennen, dass die rechte nicht verfügbar ist, legen die linke wieder ab, warten gleich lang, greifen wiederum gleichzeitig die linke usw.
- Das Problem der endlosen Ausführung ohne Fortschritt wird Verhungern (Starvation) genannt.

Korrekte, aber unbefriedigende Lösung



```
#define N 5                                /* Anzahl der Philosophen      */
→ semaphore mutex = 1;

void philosopher (int i)      /* i:0..N-1, welcher Philosoph */
{
    while (TRUE) {
        think();                                /* Denken                      */
→         down(&mutex);
        take_fork(i);                          /* Greife linke Gabel          */
→         take_fork((i+1)%N);                /* Greife rechte Gabel         */
→         eat();                                /* Essen                       */
→         put_fork(i);                          /* Ablegen linke Gabel        */
→         put_fork((i+1)%N);                /* Ablegen rechte Gabel       */
→         up(&mutex)
    }
}
```

kritisch

down(&mutex);
take_fork(i); /* Greife linke Gabel */
take_fork((i+1)%N); /* Greife rechte Gabel */
eat(); /* Essen */
put_fork(i); /* Ablegen linke Gabel */
put_fork((i+1)%N); /* Ablegen rechte Gabel */
up(&mutex)

Semaphore schützt gesamten „Ess-Abschnitt“
Kein Deadlock, aber unbefriedigend:
nur ein Philosoph kann gleichzeitig essen !

```
#define N          5          /* Anzahl der Philosophen      */
#define LEFT    (i-1)%N    /* Nummer des linken Nachbarn von i  */
#define RIGHT   (i+1)%N    /* Nummer des rechten Nachbarn von i */
#define THINKING 0        /* Zustand: Denkend          */
#define HUNGRY   1        /* Zust: Versucht, Gabeln zu bekommen */
#define EATING    2        /* Zustand: Essend          */

/*
 * gemeinsame Variablen
 */

int state[N];          /* Zustaende aller Philosophen      */
semaphore mutex = 1;    /* fuer wechselseitigen Ausschluss */
semaphore s[N];         /* Semaphor fuer jeden Philosoph  */

void philosopher(int i) { /* i:0..N-1, welcher Philosoph      */
    while (TRUE) {
        think();          /* Denken                      */
        take_forks(i);    /* Greife beide Gabeln oder blockiere */
        eat();             /* Essen                      */
        put_forks(i);     /* Ablegen beider Gabeln      */
    }
}
```

```
void take_forks(int i) {          /* i:0..N-1, welcher Philosoph      */
    down(&mutex);                /* tritt in krit. Bereich ein      */
    state[i] = HUNGRY;            /* zeige, dass du hungrig bist      */
    test(i);                     /* versuche, beide Gabeln zu bekommen */
    up(&mutex);                  /* verlasse krit. Bereich          */
    down(&s[i]);                 /* bockiere, falls Gabeln nicht frei */
}

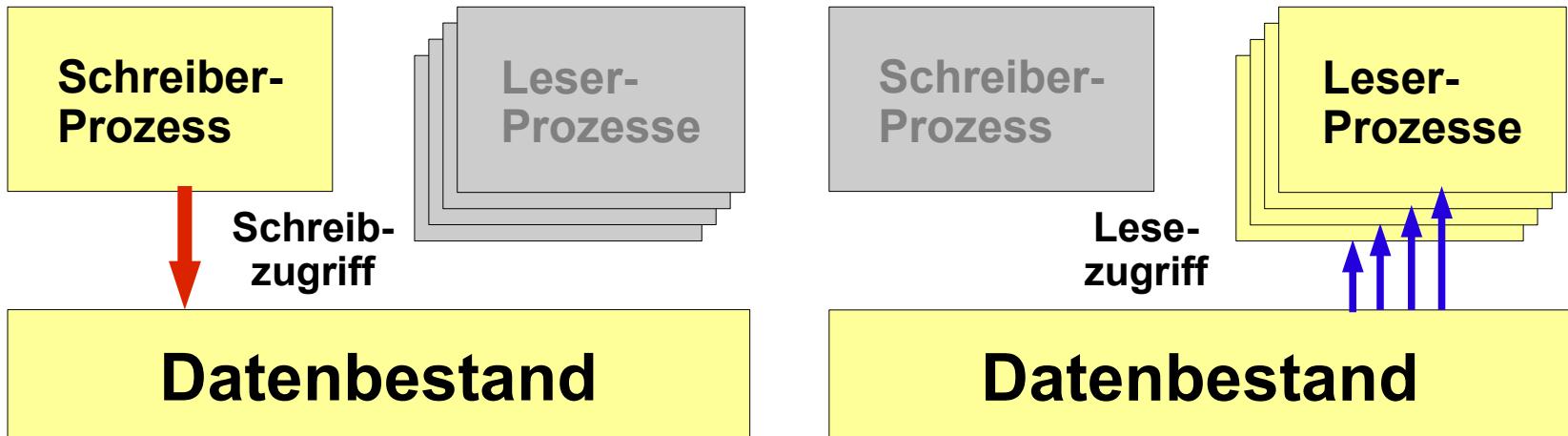
void put_forks(int i) {          /* i:0..N-1, welcher Philosoph      */
    down(&mutex);                /* tritt in krit. Bereich ein      */
    state[i] = THINKING;          /* zeige, dass du fertig bist      */
    test(LEFT);                  /* kann linker Nachbar jetzt essen ? */
    test(RIGHT);                 /* kann rechter Nachbar jetzt essen ? */
    up(&mutex);                  /* verlasse krit. Bereich          */
}

void test(int i) {               /* i:0..N-1, welcher Philosoph      */
    if (state[i]== HUNGRY && state[LEFT]!=EATING && state[RIGHT] !=EATING)
    {
        state[i]=EATING;          /* jetzt kann Phil i essen !      */
        up(&s[i]);                /* "sage es ihm"                  */
    }
}
```

- **Jeder Prozess führt die Prozedur `philosopher` als Hauptprogramm aus, die anderen Prozeduren sind gewöhnliche Unterprogramme, keine separaten Prozesse.**
- **Das array `state` speichert die aktuellen Zustände der Philosophen: essend, denkend, hungernd (versucht, Gabeln zu bekommen).**
- **Jeder Philosoph blockiert an einem ihm zugeordneten Semaphor `s[i]`, wenn die benötigten Gabeln nicht verfügbar sind.**
- **Das Semaphor `mutex` sichert den kritischen Abschnitt der Benutzung der Zustandsinformation.**

- **Die Lösung ist korrekt, sie enthält keinen Deadlock und kein Verhungern.**
- **Die Lösung lässt eine möglichst hohe Nebenläufigkeit zwischen den Philosophen zu.**
- **Die Lösung ist für eine beliebige Anzahl von Philosophen korrekt.**

5.3.3. Leser-Schreiber-Problem



Zu jedem Zeitpunkt dürfen entweder mehrere Leser oder ein Schreiber zugreifen.

Verboten: gleichzeitiges Lesen und Schreiben

Wie sollten Leser- und Schreiber-Programme aussehen?

Lösung mit Semaphoren



5.3.3



```
/* gemeinsame Variablen: */  
semaphore mutex = 1;           /* wechsels. Ausschluss fuer rc */  
semaphore db = 1;             /* Semaphor fuer Datenbestand */  
int rc = 0;                   /* readcount: Anzahl Leser */  
  
void reader(void) {           /* Leser */  
    while (TRUE) {  
        down(&mutex);           /* erhalten exkl. Zugriff auf rc */  
        rc=rc+1;                /* ein zusaetlicher Leser */  
        if (rc==1) down (&db);  /* falls 1.Leser, reserviere Daten */  
        up(&mutex);             /* freigeben exkl. Zugriff auf rc */  
  
        read_data_base();       /* lies Datenbestand */  
  
        down(&mutex);           /* erhalten exkl. Zugriff auf rc */  
        rc=rc-1;                /* ein Leser weniger */  
        if (rc==0) up (&db);   /* falls letzter Leser, Daten freigeb */  
        up(&mutex);             /* freigeben exkl. Zugriff auf rc */  
        use_data_read();        /* unkrit. Bereich */  
    }  
}
```

```
void writer(void) {                                /* Schreiber
    while (TRUE) {
        create_data();                            /* unkrit. Bereich
        down(&db);                                /* erhalten exkl. Zugriff auf Daten */
        write_data_base();                         /* schreib Datenbestand
        up(&db);                                 /* freigeben exkl. Zugriff auf Daten */
    }
}
```



- **Semaphore `mutex` sichert den kritischen Abschnitt in der Benutzung des Read-Counters `rc`.**
- **Semaphore `db` sichert den Zugriff auf den Datenbestand, so dass entweder mehrere Leser oder ein Schreiber zugreifen können. Der erste Leser führt eine P-Operation durch, alle weiteren inkrementieren nur `rc`. Der letzte Leser führt eine V-Operation auf dem Semaphore `db` aus, so dass ein wartender Schreiber Zugriff erhält.**
- **Lösung bevorzugt Leser gegenüber Schreibern. Neu eintreffende Leser erhalten Zugriff vor einem schon wartenden Schreiber, wenn noch mindestens ein Leser Zugriff hat.**

Was haben wir in Kap. 5 gemacht?

- **Interaktionen zwischen Prozessen können zu zeitkritischen Abläufen führen, d.h. Situationen, in denen das Ergebnis vom zeitlichen Ablauf abhängt. Zeitkritische Abläufe führen zu einem nicht reproduzierbaren Verhalten und müssen vermieden werden.**
- **Kritische Bereiche als Teile von Programmen, in denen mit anderen Prozessen gemeinsamer Zustand manipuliert wird, bieten die Möglichkeit des wechselseitigen Ausschlusses. Sie vermeiden damit zeitkritische Abläufe und erlauben komplexe unteilbare (atomare) Aktionen.**

- **Viele Primitive zur Synchronisation und Kommunikation von Prozessen wurden vorgeschlagen. Sie machen verschiedene Annahmen über die unterlagerten elementaren unteilbaren Operationen, sind aber im Prinzip gleichmächtig. Besprochen wurden insbesondere Semaphoren und Nachrichtenaustausch, die in aktuellen Systemen weit verbreitet sind.**
- **Es gibt eine Reihe von klassischen Problemen der Interprozesskommunikation, an denen die Nutzbarkeit neuer vorgeschlagener Primitive gezeigt wird. Von diesen wurden das Erzeuger-Verbraucher-Problem, das Philosophen-Problem und das Leser-Schreiber-Problem besprochen. Auch mit den heute üblichen Primitiven muss sorgfältig umgegangen werden, um inkorrekte Lösungen, Deadlocks und Starvation zu vermeiden.**

Kap. 6: Prozesskommunikation

6.1 Ereigniskommunikation

6.2 Nachrichtenkommunikation

6.3 Zusammenfassung

Bisher:

- Prozesse kooperieren durch gemeinsame Benutzung von Betriebsmitteln (z.B. Speicherbereichen) (Sharing-Paradigma)
- Synchronisationsmechanismen zur Vermeidung von zeitkritischen Abläufen
- Synchronisationsmechanismen auch zur Signalisierung von Ereignissen zwischen Prozessen geeignet (z.B. Fertig-Signale zur Durchsetzung einer Vorrangrelation zwischen Prozessen eines Prozesssystems)

Hier:

- Betriebssysteme können darüberhinaus spezielle Mechanismen für die Kommunikation (Signalisierung) von Ereignissen offerieren. (Diese Mechanismen können wiederum auch zur Synchronisation eingesetzt werden).
- Kommunikation von Ereignissen ist ein Spezialfall der Nachrichtenorientierten Kommunikation (vgl. 6.2).

- Ereignisse zeigen das Eintreten einer "wichtigen Begebenheit" an
 - z.B. das Erreichen eines bestimmten Zustands
 - das Eintreten eines Fehlers
 - den Ablauf einer vorgegebenen Zeitspanne.
- Ereigniskonzept als Übertragung des Hardware-Interrupt-Konzepts auf die Software-Ebene
- Ereignisse werden lediglich nummeriert, Zuordnung einer tatsächlichen Bedeutung kann ganz oder teilweise vorgegeben oder den Anwendungen überlassen sein.
- Bereitstellung eines effizienten Ereigniskonzepts ist insbesondere für Echtzeitbetriebssysteme von Bedeutung:
In Echtzeitanwendungen modellieren Ereignisse häufig reale Ereignisse in der Außenwelt.

Eigenschaften:

- UNIX unterscheidet zwei Arten von Signalen:
 - Fehler-Ereignisse, die bei der Programmausführung auftreten, z.B.:
 - Adressierung eines ungültigen Speicherbereichs (→ SIGSEGV)
 - Versuch der Ausführung einer ungültigen Instruktion (→ SIGILL)
 - Asynchrone Ereignisse, die von außerhalb des Prozesses stammen, z.B.:
 - Beendigung eines Sohn-Prozesses (→ SIGCHLD)
 - Terminalleitungsstreiber zeigt Ende der Übertragung an (→ SIGHUP)
 - Programmierter Timer läuft ab (→ SIGALRM)
- Das UNIX-Signalkonzept überträgt des Hardware-Interrupt-Konzepts auf die Software-Ebene:

Interrupt	\cong Signal
Interrupt Handler	\cong Signal Handler
Maskieren von Interrupts	\cong Maskieren von Signalen

- UNIX unterscheidet insgesamt MAXSIG=32 Signale (Wortlänge). Die Bezeichnung der Signale und ihr zugehöriger Index sind z.T. für BSD-UNIX und UNIX System V verschieden.
- Typische Signale (UNIX System V):

Signal	Wert	Bedeutung	Default-Aktion
SIGHUP	1	Hangup, Verbindungsabbruch	exit
SIGILL	4	ungültiger Maschinenbefehl	core dump
SIGABRT	6	Abbruch des Prozesses mit Dump	core dump
SIGFPE	8	Gleitkommafehler	core dump
SIGKILL	9	Kill-Signal, nicht abfangbar	exit
SIGSEGV	11	ungültige Speicheradresse	core dump
SIGPIPE	13	Schreiben in Pipe ohne Empfänger	exit (vgl. 6.2)
SIGALRM	14	Ablauf des Weckers	exit
SIGTERM	15	Aufforderung zur Terminierung	exit
SIGUSR1	16	Anwender-definierbar 1	exit
SIGUSR2	17	Anwender-definierbar 2	exit
SIGCHLD	18	Statusänderung Sohnprozess	ignore
SIGTSTP	24	Anwender-Stop von tty	stop
SIGCONT	25	Fortsetzung	ignore

Beispiel 1: Signalisierung in UNIX (3)



- Ein Prozess im Zustand rechnend kann durch den BS-Kern Signale senden und empfangen.
- Nicht-rechnende Prozesse können nur Signale von rechnenden Prozessen empfangen.
- Ein Prozess kann festlegen, dass er bestimmte Signale selbst durch einen Signal-Handler behandeln oder ignorieren will (s.u.). Einige Signale (wie z.B. SIGKILL) können nicht abgefangen werden.
- Ein Prozess kann nur Signale an Prozesse mit derselben realen oder effektiven User-Id senden (Prozessgruppe), insbesondere an sich selbst. Darüberhinaus können Prozesse mit der realen oder effektiven User-Id 0 (d.h. mit Superuser-Berechtigung) Signale an beliebige Prozesse senden.
- Der BS-Kern kann jedes Signal an jeden Prozess senden.
- Die Verarbeitung von anstehenden Signalen geschieht, wenn der Prozess aktiv ist und im Kernmodus ausgeführt wird.
- Das Betriebssystem hält die Signalmasken im Prozesskontrollblock des Prozesses (proc structure und user structure, vgl. 2.1).

Systemdienste zur Signalisierung:

- `int sigaction(int sig, struct sigaction * action,
 struct sigaction * oldaction);`

```
#include <signal.h>  
struct sigaction {  
    int sa_flags;                  /* Flags, POSIX: nur SA_NOCLDSTOP */  
    void (*sa_handler)();        /* Adresse einer eigenen Funktion */  
                                  /* oder SIG_IGN oder SIG_DFL */  
    sigset_t sa_mask;            /* zu maskierende Signale, wenn */  
                                  /* handler ausgefuehrt wird. */  
}
```

- - Zuordnung einer Aktion bei Auftreten des Signals `sig` und Speichern der bisherigen Arbeitsweise gemäß POSIX (auch BSD `sigvec()` oder System V `signal()`). Erfolgt i.d.R. nur einmal im Programm für jedes Signal, für das die Default-Aktion geändert werden soll.
- - Fehler `EINVAL`, falls `sig` `SIGKILL` oder `SIGSTOP` ist.
- - Im Falle von `sig==SIGCHLD` bewirkt `SA_NOCLDSTOP`, dass `SIGCHLD` erst gesendet wird, wenn alle Sohnprozesse beendet sind (Normalfall).
- - Nach Abschluss der erfolgreichen Behandlung erfolgt die Fortsetzung des Programms an der unterbrochenen Stelle.

Beispiel 1: Signalisierung in UNIX (5)



- `int kill(pid_t pid, int sig);`
Sendet das Signal `sig` an den Prozess `pid` (Mitglied derselben Prozessgruppe) oder an alle Prozesse dieser Gruppe, falls `pid` Null ist (auch System V `sigsendset()`). Weitere Möglichkeiten für Prozesse mit User-Id 0 (Superuser).
- `int sigpause(int sig);`
Das Signal `sig` wird aus der Signalmaske entfernt (zugelassen), und der Prozess blockiert, bis ein Signal empfangen oder der Prozess beendet wird.
- `int pause(void);`
Der Prozess blockiert, bis ein Signal empfangen oder der Prozess beendet wird.
- `int sighold(int sig);`
Das Signal `sig` wird der Signalmaske hinzugefügt und dadurch bis auf weiteres zurückgehalten (maskiert).

- `int sigrelse(int sig);`

Das Signal `sig` wird aus der Signalmaske entfernt und damit wieder zugelassen.

- `int sigignore(int sig);`

Das Signal `sig` wird auf die Disposition `SIG_IGN` gesetzt und damit bis auf weiteres vom aufrufenden Prozess ignoriert.

- `void alarm(int seconds);`

Das Signal `SIG_ALRM` wird nach Ablauf der übergebenen Zeitdauer zugestellt (Wecker stellen).

wird im Praktikum vertieft.

Motivation:

- Signalisierung von Ereignissen ist Spezialfall des Austausches von Nachrichten beliebiger Art (Paradigma der Kooperation durch Nachrichtenaustausch).
- Betriebssysteme besitzen i.d.R. entsprechende Mechanismen zur nachrichtenorientierten Kommunikation.
- Nachrichtenorientierte Kommunikation ist sowohl für lokale wie auch für verteilte Rechensysteme anwendbar.

Hier:

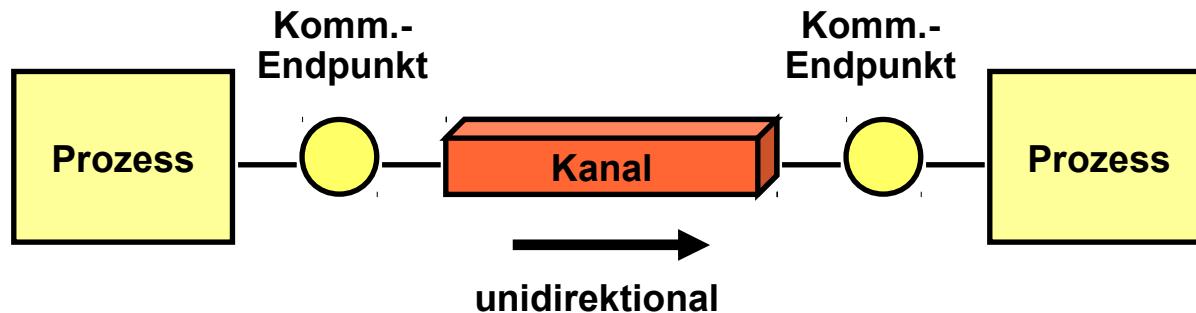
- Besprechung der Grundlagen eines solchen allgemeinen Nachrichten(austausch)systems zur Interprozess-kommunikation.

- Aufgabe
 - Mechanismus, so dass Prozesse einander Nachrichten zukommen lassen, ohne über gemeinsamen Speicher verfügen zu müssen.
- Operationen
 - SEND (*message*, ...) zum Senden einer Nachricht durch einen Sendeprozess oder einfach Sender.
 - RECEIVE (*message*, ...) zum Empfangen einer Nachricht durch einen Empfangsprozess oder Empfänger.
- Kopplung/Verbindung der Kommunikationspartner
 - notwendig, um die Kommunikation zu ermöglichen.
 - Verbindung heißt Kommunikationskanal oder Kanal.
 - Ein Prozess kann zu einem Zeitpunkt mehrere Kanäle aufrecht erhalten (i.d.R. zu verschiedenen Prozessen).
- Wichtige Entwurfsaspekte eines Nachrichtensystems
 - Adressierung, Pufferung, Nachrichtenstruktur

6.2.1 Kommunikationskanal



- Anzahl der Kommunikationsteilnehmer eines Kanals
 - Genau zwei: Regelfall.
 - Mehr als zwei: Gruppenkommunikation (Multicast-Dienst, Spezialfall: Broadcast, d.h. Rundsendung an alle), hier nicht weiter betrachtet.
- Richtung des Nachrichtenflusses eines Kanals
 - Kanal heißt gerichtet oder unidirektional, wenn ein Prozess ausschließlich die Sender-Rolle, der andere ausschließlich die Empfänger-Rolle ausübt, ansonsten ungerichtet oder bidirektional.



6.2.2 Adressierung



- Direkte Adressierung
 - Prozesse haben eindeutige Adressen.
 - Benennung ist explizit und symmetrisch:
sendender Prozess muss Empfänger benennen und umgekehrt.

SEND (P, *message*)

Sende eine Nachricht an Prozess P.

RECEIVE (Q, *message*)

Empfange eine Nachricht von Prozess Q.

- Asymmetrische Variante (z.B. für Server-Prozesse):
Sender benennt Empfänger, Empfänger (Server-Prozess) wird mit dem Empfang die Identität des Senders bekannt:

SEND (P, *message*)

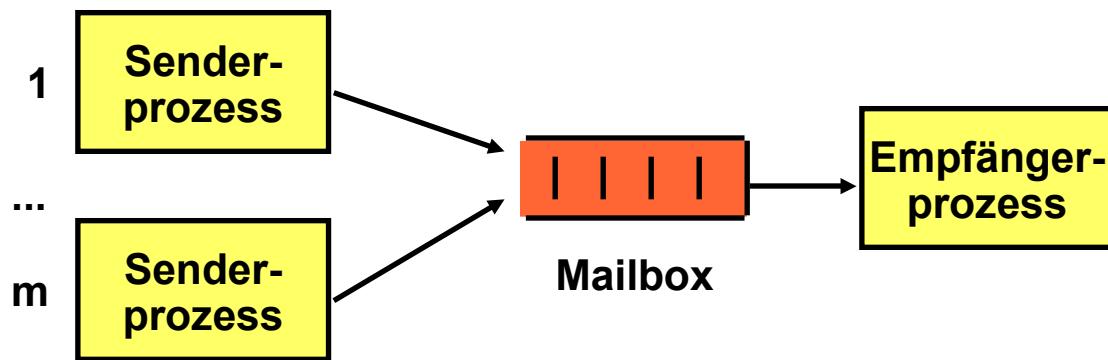
RECEIVE (*sender_id* , *message*)

- Indirekte Adressierung
 - Kommunikation erfolgt indirekt über zwischengeschaltete Mailboxes (Puffer für eine Anzahl von Nachrichten):

SEND (*mbox, message*)
Sende eine Nachricht an Mailbox *mbox*.

RECEIVE (*mbox, message*)
Empfange eine Nachricht von Mailbox *mbox*.
 - Vorteile:
 - Verbesserte Modularität.
 - Prozessmenge kann transparent restrukturiert werden, z.B. nach Ausfall eines Empfangsprozesses.
 - Erweiterte Zuordnungsmöglichkeiten von Sendern und Empfängern, wie z.B. $m:1$, $1:n$, $m:n$.

- Beispiel: m:1



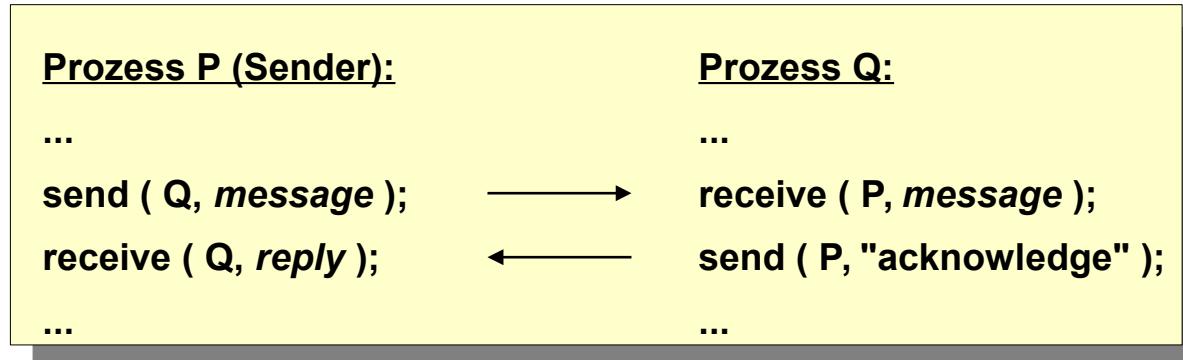
- Def. Kapazität
 - Kapazität eines Kanals bezeichnet Anzahl der Nachrichten, die vorübergehend in einem Kanal gespeichert werden können, um Sender und Empfänger zeitlich zu entkoppeln
 - Die Pufferungsfähigkeit eines Kanals wird i.d.R. durch einen Warteraum/Warteschlange im Betriebssystemkern erreicht.

- Keine Pufferung (Kapazität Null)
 - Sender wird blockiert, wenn SEND-Operation vor entsprechender RECEIVE-Operation stattfindet.
Wird dann die entsprechende RECEIVE-Operation ausgeführt, wird die Nachricht ohne Zwischenspeicherung unmittelbar vom Sender- zum Empfänger-Prozess kopiert.
 - Findet umgekehrt RECEIVE zuerst statt, so wird der Empfänger bis zum Aufruf der SEND-Operation blockiert.
 - Diese ungepufferte Kommunikationsform, die Sender und Empfänger zeitlich sehr eng koppelt, heisst Rendezvous oder synchroner Nachrichtenaustausch.
 - Beispiel: Hoare: Communicating Sequential Processes (CSP).
 - Beispiel: Kommunikation zwischen ADA-Tasks.
 - Synchroner Nachrichtenaustausch wird häufig als zu inflexibel angesehen.

- Beschänkte Kapazität
 - Kanal kann zu einem Zeitpunkt maximal N Nachrichten enthalten (Warteraum der Kapazität N).
 - Im Falle einer SEND-Operation bei nicht-vollem Warteraum wird die Nachricht im Warteraum abgelegt, und Sendeprozess fährt fort.
 - Ist der Warteraum voll (er enthält N gesendete aber noch nicht empfangene Nachrichten), so wird der Sender blockiert, bis ein freier Warteplatz vorhanden ist.
 - Ablegen der Nachricht kann durch Kopieren der Nachricht oder Speichern eines Zeigers auf die Nachricht realisiert sein.
 - Analog wird Empfänger bei Ausführung einer RECEIVE-Operation blockiert, wenn Warteraum leer.

- Unbeschränkte Kapazität
 - Kanal kann potentiell eine unbeschränkte Anzahl von Nachrichten enthalten.
 - SEND-Operation kann nicht blockieren.
 - Lediglich Empfänger kann bei Ausführung einer RECEIVE-Operation blockieren, wenn Warteraum leer.
 - Implementierung kann erfolgen, in dem Sender und Empfänger die Warteplätze beim Aufruf "mitbringen".

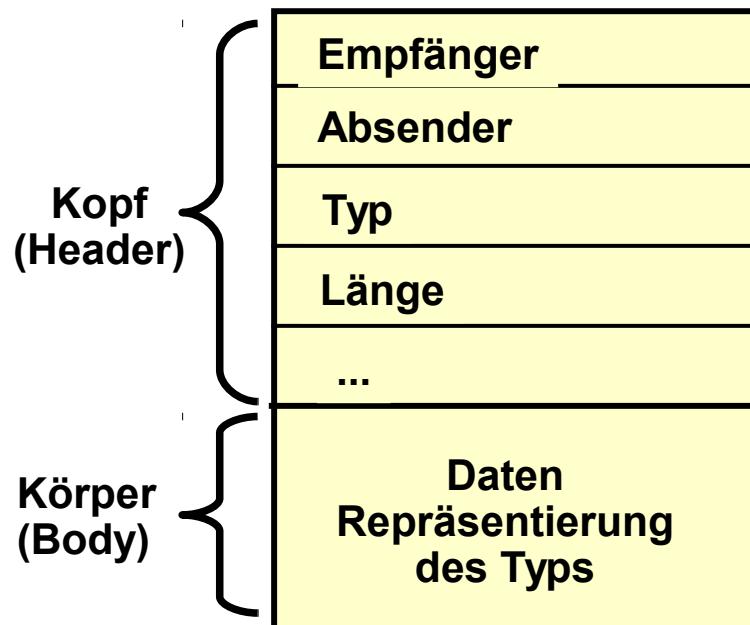
- Konsequenzen
 - Gepufferte Kommunikation (beschränkt oder unbeschränkt) bewirkt zeitlich lose Kopplung der Kommunikationspartner.
 - Nach Abschluss der SEND-Operation weiß der Sender nicht, dass der Empfänger die Nachricht erhalten hat. Er kennt i.d.R. auch keine maximale Zeitdauer dafür.
 - Wenn dieses Wissen wesentlich für den Sender ist, muss dazu eine explizite Kommunikation zwischen Sender und Empfänger durchgeführt werden:



6.2.4 Semantik von Nachrichten



- Typisierte Nachrichten
 - Nachrichten haben eine typisierte Struktur.
 - Typ ist Sender und Empfänger und z.T. dem Nachrichten-system bekannt und wird in Operationen verwendet.
 - Beispielhafter Aufbau einer Nachricht:



- Nachrichtencontainer
 - Nachrichten sind für Sender und Empfänger identifizierbare Einheiten fester oder variabler Länge. Nachrichtengrenzen bleiben erhalten.
 - Korrekte Interpretation der internen Struktur einer Nachricht obliegt den Kommunikationspartnern.
 - Beispiel: UNIX message queues.
- Bytestrom
 - Empfänger (und das Nachrichtensystem) sehen ausschließlich eine Folge von Zeichen (Bytestrom).
 - Übergebene Nachrichten verschiedener SEND-Operationen sind als Einheiten nicht mehr identifizierbar. Nachrichtengrenzen gehen verloren.
 - Beispiel: UNIX pipes.

6.2.5 Implementierung



- Klassische Implementierung
 - Speicher für zu puffernde Nachrichten liegt im BS-Kern.
 - Bei Ausführung von SEND wird die Nachricht aus dem sendenden Prozess(-adressraum) in den BS-Kern kopiert.
 - Bei Ausführung von RECEIVE wird die Nachricht aus dem BS-Kern in den empfangenden Prozess(-adressraum) kopiert.
 - Fazit: 2-maliges Kopieren. Aufwendig!
- Integration mit Speicherverwaltung
 - In modernen Betriebssystemen wird die Implementierung von Nachrichtenkommunikation mit dem Virtual Memory Management integriert (siehe Kap. 8, z.B. Mach).
 - Versenden von Nachrichten wird auf das Kopieren von Deskriptoren von Speicherbereichen beschränkt.
 - Die Speicherbereiche werden durch Markierung als *copy-on-write* nur im Notfall wirklich kopiert (lazy copying, vgl. Kap. 8).

6.2.6 Beispiel: UNIX

UNIX stellt mehrere Dienste zum Nachrichtenaustausch zwischen Prozessen zur Verfügung:

- Pipes
 - Ursprünglicher Mechanismus zum unidirektionalen Nachrichten-transport (Bytestrom) zwischen verwandten Prozessen
 - Vererbung der Kommunikationsendpunkte durch fork().
- Named Pipes oder FIFOs
 - Erweiterung auf nicht-verwandte Prozesse
 - Dateisystem-Namensraum als gemeinsamer Namensraum zur Benennung von Named Pipes
 - Repräsentierung von Named Pipes im Dateisystem.
- Message Queues
 - Bestandteil der System V IPC-Mechanismen
 - Message Queue ist komplexes, gemeinsam benutztes Objekt zum Austausch typisierter Nachrichten zwischen potentiell beliebig vielen Sende- und Empfangs-Prozessen.

Beispiel: UNIX (2)

6.2.6

- Sockets
 - In 4.2BSD UNIX zusammen mit TCP/IP eingeführtes Konzept zur allgemeinen, rechnerübergreifenden, bidirektionalen, nachrichtenorientierten Interprozesskommunikation
 - zur Programmierung von Client/Server-Kommunikation geeignet
 - weit verbreitet (wird in LV Verteilte Systeme behandelt).
- STREAMS
 - Mit UNIX System V Rel. 3 erstmals eingeführte Gesamtumgebung zur Entwicklung von geschichteten IPC-Protokollen.
 - Modularisierung durch verkettbare STREAMS-Module.
 - In UNIX System V Rel. 4 werden Pipes, Named Pipes, Terminal-Protokolle und Netzwerkkommunikation in der STREAMS-Umgebung bereitgestellt.

Eigenschaften:

- Anonyme Pipe definiert namenlosen unidirektionalen Kommunikationskanal, über den zwei Prozesse mit einem gemeinsamen Vorfahren einen Bytestrom kommunizieren können.
- Pipe besitzt eine Pufferkapazität von einer Seite (z.B. 4 Kb)
- wird von den Prozessen wie eine Datei behandelt.

Typischer Umgang:

- Vater legt Pipe an, erzeugt zwei Söhne durch fork(), die die geöffneten Enden der Pipe als File-Deskriptoren erben.
- Der Schreiber schließt (mittels close) die Lese-Seite, der Leser die Schreib-Seite, der Vater beide Seiten.
- Analog kann der Vater mit einem Sohn über eine Pipe kommunizieren.

Systemaufrufe:

- `int pipe(int pfds[2])`
Erzeugen einer anonymen geöffneten Pipe. Der File-Deskriptor `pfds[0]` kann zum Lesen, `pfds[1]` zum Schreiben benutzt werden.
- `int write(int fd, char* buf, unsigned length)`
Senden von `length` bytes in die durch `fd` bezeichnete pipe. Aufrufer blockiert, falls Pipe voll ist. Nicht-blockierendes Schreiben ist nach `fcntl` mit `O_NDELAY` möglich. Write erzeugt `SIGPIPE`-Signal, falls Leser-Seite geschlossen wurde.
- `int read(int fd, char* buf, unsigned length)`
Empfangen von maximal `length` bytes aus der durch `fd` bezeichneten pipe. Aufrufer blockiert, falls Pipe leer ist. (Blockierung kann analog zu `write` vermieden werden). Rückgabewert ist die tatsächlich gelesene Anzahl bytes, 0 bei Schließen der Pipe durch die Sender-Seite (Dateiende), -1 bei Fehler.
- `int close(int fd)`
Schließen eines Endes der Pipe zum Beenden der Kommunikation.

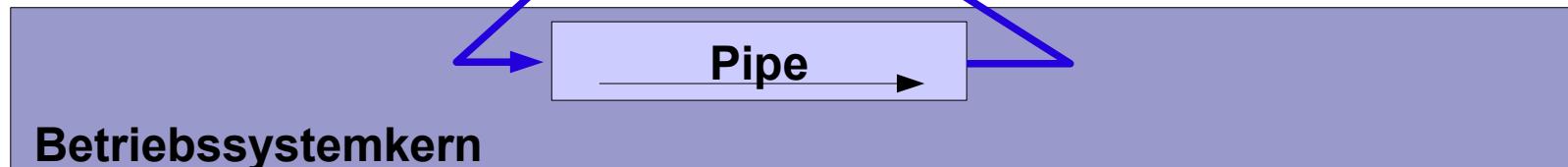
Beispiel: Anonyme Pipes (3)

Vaterprozess

```
pipe(fds);  
fork();  
close(fds[0]);  
write(fds[1], ...);
```

Sohnprozess

```
close(fds[1]);  
read(fds[0], ...);
```



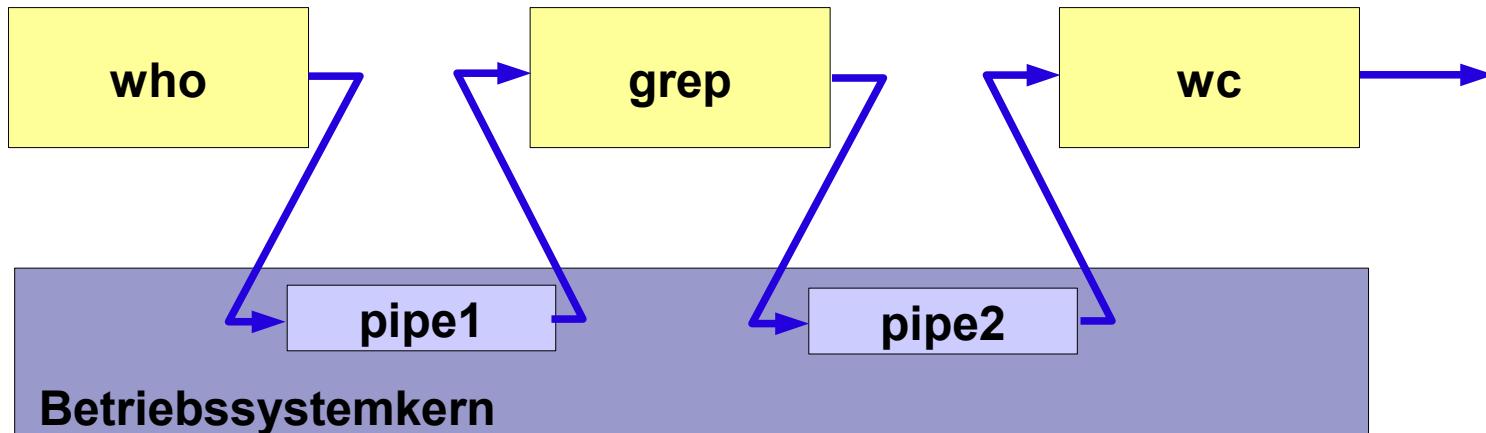
- Systemfunktion `int pipe(int fds[2]);`
erzeugt zwei File-Deskriptoren im übergebenen Vektor `fds`:
`fds[0]` ist zum Lesen geöffnet
`fds[1]` zum Schreiben
- Rückgabewert: 0 für ok, -1 für Fehler

Beispiel: Pipes in der Shell

Shell-Kommandozeile (wie oft ist „hansl“ auf dem Rechner angemeldet?)

```
who | grep "hansl" | wc -l
```

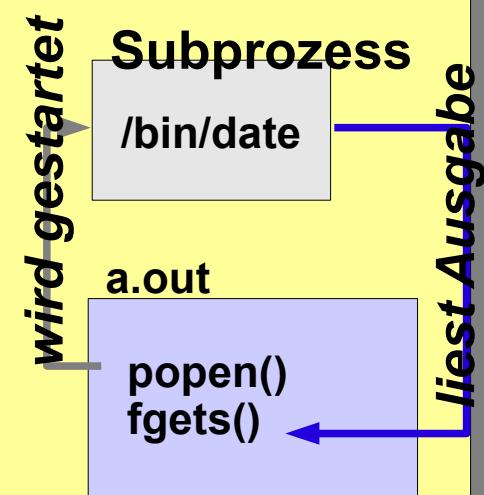
Dazu erzeugt die Shell 2 Pipes und 3 Sohn-Prozesse,
deren Standardein-/ausgabe-File-Deskriptoren sie wie folgt setzt:



popen()

```
#include <stdio.h>
#include <stdlib.h>
#define MAXZEILE 80
int main(void) {
    char zeile[MAXZEILE];
    FILE *fp;
    if ( (fp = popen("/bin/date", "r")) == NULL) {
        perror("Fehler bei popen");
        exit(1);
    }
    if (fgets(zeile, MAXZEILE, fp) == NULL) {
        perror("Fehler bei fgets");
        exit(2);
    }
    pclose(fp);
    printf("Ausgabe von 'date' ist: %s\n", zeile);
    return 0;
}
```

```
$ ./a.out
Ausgabe von 'date' ist: Di 26. Nov 17:41:19 CET 2013
```



- Mit `popen()` kann ein Kommando als Subprozess gestartet, in dessen Standardeingabe geschrieben (`"w"`) oder dessen Standardausgabe gelesen (`"r"`) werden kann (entweder/oder)
- Verwendung mit „Stream“-Funktionen (`fprintf()`, `fgets()`, ...)
- Schließen mit `pclose()`

Eigenschaften:

- Eine benannte Pipe, auch FIFO genannt, definiert einen Kommunikationskanal, über den mehrere Sender- und mehrere Empfänger-Prozesse, die nicht miteinander verwandt sein müssen, einen gemeinsamen Bytestrom kommunizieren können.
- Eine benannte Pipe besitzt einen Namen aus dem Dateinamensraum und eine Repräsentierung im Dateisystem (inode, Dateikontrollblock, vgl. Kap. 10) aber keine Datenblöcke, sondern lediglich einen Seiten-Puffer im Arbeitsspeicher.
- Für benannte Pipes existieren Zugriffsrechte wie bei Dateien, die beim Öffnen überprüft werden.
- In System V sind FIFOs durch ein spezielles vnode-Dateisystem `fifo` implementiert (vgl. Kap. 10).

Systemaufrufe:

- `int mknod(char* pfad, int modus, int dev)`
Mit mknod können allgemeiner beliebige Knoten im Dateisystem angelegt werden. Zum Erzeugen einer Named Pipe gibt pfad den (Datei-)Namen für das FIFO an, modus wird gebildet durch `S_IFIFO | <rechte>`, wobei die Bitmaske `<rechte>` die Lese-/Schreibrechte für user/group/others wie üblich kodiert (z.B. dürfen mit 0666 beliebige Prozesse lesen und schreiben), dev hat für FIFOs den Wert 0. (In V.4 existiert zusätzlich `mkfifo()`).
- `int open(char* pfad, int modus)`
Öffnen der Pipe mit Namen pfad zum Lesen (modus `O_RDONLY`) oder Schreiben (modus `O_WRONLY`) wie bei üblicher Datei. Öffnen blockiert, bis sowohl ein Leser als auch ein Schreiber vorhanden sind. Durch Setzen des Flags `O_NDELAY` wird nicht-blockierendes Lesen oder Schreiben ermöglicht.
- `read, write` und `close` wie bei anonymen Pipes.

Vertiefung erfolgt im Praktikum.

Beispiel: Named Pipes oder FIFOs (3)



6.2.6

```
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(void) {
    int fd;
    mkfifo("my_fifo", 0666);
    fd = open("my_fifo", O_WRONLY);
    write(fd, ...);
    ...
    /* Schließen des FIFO */
    unlink("my_fifo");
}
```

```
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(void) {
    int fd;
    ...
    fd = open("my_fifo", O_RDONLY);
    read(fd, ...);
    ...
}
```

(Fehlerbehandlung weggelassen)

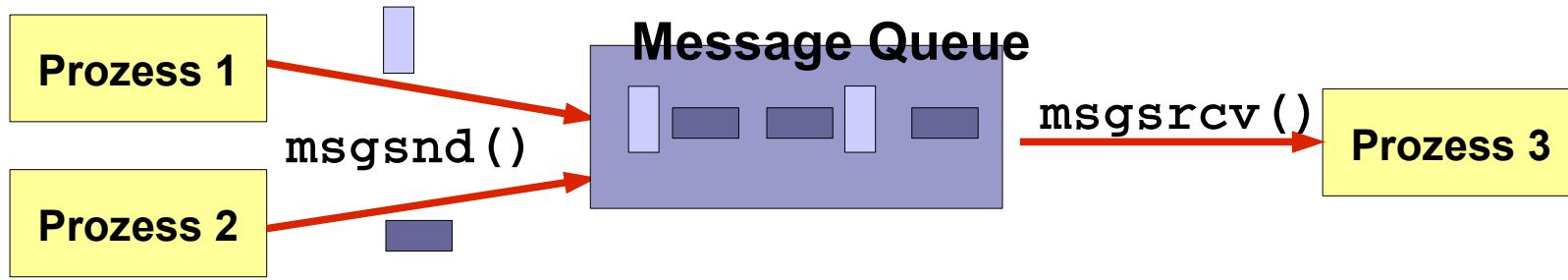
- `mkfifo()` erzeugt FIFO mit angegebenem Pfad / Zugriffsbits
- Benannte Pipe (FIFO) erscheint wie eine Datei im Dateibaum
- Kann daher von beliebigen Prozessen (nicht nur Vater/Sohn) auf dem Rechner „gesehen“ und mit den bekannten Dateioperationen genutzt werden (Zugriffsrechte vorausgesetzt)
- Schließen über Datei-Löschoperation `unlink()` (!)

Eigenschaften:

- Message Queues (Nachrichtenwarteschlangen) im Rahmen der UNIX System V IPC-Mechanismen eingeführt, Bedeutung gesunken.
- Message Queue definiert Kommunikationskanal, über den mehrere Sender- und mehrere Empfänger-Prozesse, die nicht miteinander verwandt sein müssen, typisierte Nachrichten variabler Länge kommunizieren können.
- Message Queues besitzen eindeutigen Bezeichner aus dem Key-Namensraum, geöffnete Message Queue wird intern über Integer-Id bezeichnet.
- Nachricht besteht aus einem Integer-Nachrichtentyp und variabel langem Nachrichteninhalt. Festlegung der Bedeutung von Typ und Inhalt ist den Prozessen vorbehalten.
- Message Queue hält Berechtigungen zur Zugriffskontrolle.
- Anzahl und Puffergröße der Message Queues werden bei der Systemgenerierung festgelegt.

Systemaufrufe:

- `int msgget(key_t key, int flag)`
Erzeugen einer neuen oder Öffnen einer existierenden Message Queue, liefert den internen Id der Message Queue.
- `int msgsnd(int id, struct msgbuf * buf, int msgsz, int msgflag)`
Senden einer Nachricht an die angegebene Message Queue.
- `int msgrcv(int id, struct msgbuf * buf, int msgsz, long msgtype, int msgflag)`
Empfangen einer Nachricht eines wählbaren oder beliebigen Typs von der angegebenen Message Queue. Der Aufrufer blockiert, falls keine Nachricht des gewünschten Typs vorhanden ist. Die Blockierung kann über ein Flag vermieden werden.
- `int msgctl(int id, int cmd, struct msqid_ds * buf)`
Ausführung einer Kontroll-Operation auf der Message Queue zum Auslesen von Verwaltungs-Informationen, Verändern der Berechtigungen sowie zum Zerstören der Message Queue.



- Eine Message Queue ist eine **verkettete** (Nachrichten-) **Liste**, die vom Kernel verwaltet wird
- Empfangsreihenfolge normalerweise „first-in-first-out“, eine Priorisierung der Nachrichten ist aber auch möglich

6.2.7 Beispiel: Windows NT



Windows NT stellt ebenfalls mehrere Dienste zum Nachrichtenaustausch zwischen Prozessen zur Verfügung:

- **Anonyme Pipes**
Semantik ähnlich wie UNIX anonyme Pipes.
- **Named Pipes**
geht über die UNIX-Semantik teilweise hinaus: bidirektionaler Kommunikationskanal für mehrere Sende-Prozesse/Threads und einen Empfänger (Server-Prozess), auch auf verschiedenen Windows-Rechnern im Netzwerk.
- **Mailslots**
Für jeden Mailslot gibt es einen Empfänger (den Erzeuger des Mailslots) und mehrere mögliche Sender. Wird derselbe Mailslot-Name für mehrere Mailslots auf verschiedenen Rechnern verwendet, so werden gesendete Nachrichten an allen diesen Mailslots zugestellt.

Was haben wir in Kap. 6 gemacht?

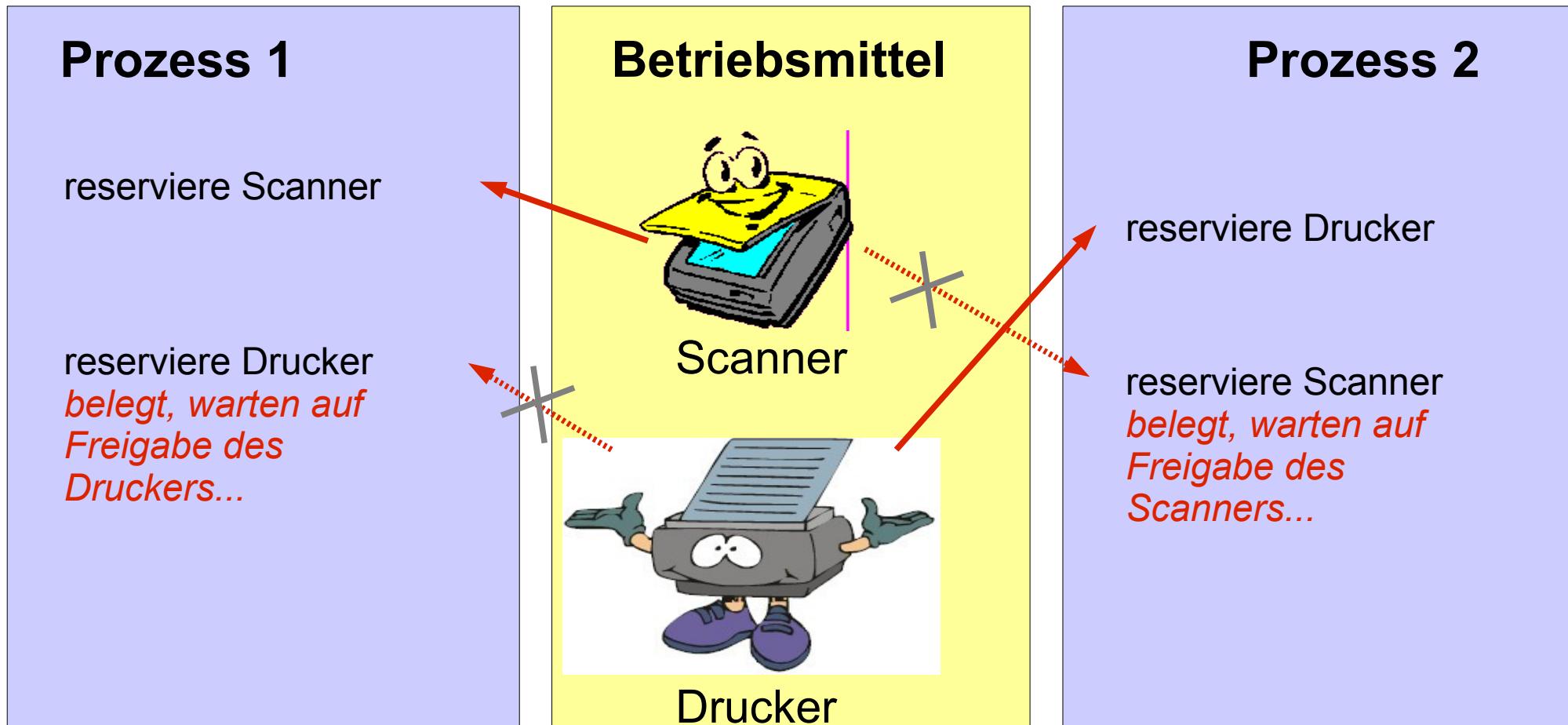
- Prozesse müssen kommunizieren können. Die grundlegenden Kommunikationsmöglichkeiten basieren auf der gemeinsamen Benutzung von Speicher (Sharing) sowie dem Nachrichtenaustausch.
- Kommunikation von Ereignissen (Signalisierung) wurde am Beispiel UNIX erläutert.
- Grundlagen der nachrichtenrichtenorientierten Kommunikation wurden vorgestellt und am Beispiel der UNIX Pipes, Named Pipes und Message Queues verdeutlicht.



Kap. 7: Deadlocks

- In diesem Kapitel wird ein grundlegendes Problem bei der Benutzung exklusiv benutzbarer Betriebsmittel behandelt, das sogenannte Deadlock-Problem.
- Ein Deadlock wird auch als Systemverklemmungszustand bezeichnet.
- Untersuchungen zu Deadlocks nehmen breiten Raum in der frühen Betriebssystem-Literatur ein.
- Ziel hier: Kennenlernen von Methoden zur Erkennung, Behebung, Vermeidung und Verhinderung von Deadlocks.

Hinführendes Beispiel



- Beide Prozesse sind blockiert und bleiben es für immer
- Eine solche Situation heißt **Deadlock** (genauere Definition folgt)
- Deadlocks treten auch in vielen anderen Situationen auf, z.B. beim Sperren von Datensätzen in Datenbanken

- 7.1 Betriebsmittel
- 7.2 Deadlocks
- 7.3 Ignorieren von Deadlocks
- 7.4 Deadlock-Erkennung und Behebung
- 7.5 Deadlock-Vermeidung
- 7.6 Deadlock-Verhinderung
- 7.7 Verwandte Fragestellungen
- 7.8 Zusammenfassung



Betriebsmittel

- Reservierbare Objekte (Objekte, auf die Zugriff erteilt werden kann) heißen **Betriebsmittel**.
- Diese können **Hard-** oder **Softwarekomponenten** sein:
 - CD-Brenner
 - Prozessor
 - ein Datensatz
 - eine Verwaltungsstruktur des Betriebssystems
 - ...
- Ein Betriebsmittel(typ) kann in mehreren identischen Instanzen oder Einheiten vorliegen
- Ein Betriebsmittel ist **unterbrechbar**, wenn es einem Prozess ohne nachteilige Auswirkungen entzogen werden kann, ansonsten heißt es **ununterbrechbar**.
- Beispiele:
 - Arbeitsspeicher → unterbrechbar (Prozess aus-/einlagern)
 - Drucker, DVD-Brenner: ununterbrechbar



Betriebsmittel (2)

7.1

- Schritte zur Benutzung eines Betriebsmittels:
 - **Anfordern** des BMs
 - **Benutzen** des BMs
 - **Freigeben** des BMs
- Falls BM angefordert wird aber nicht verfügbar ist, muss der anfordernde Prozess warten. Warten kann durch Blockieren oder durch Busy Waiting realisiert sein.
- Form einer BM-Anforderung ist in konkreten Systemen sehr unterschiedlich, u.U. auch BM-Typ-abhängig.
Typisch: open (bm) .
- Bewilligung einer Betriebsmittelanforderung heißt auch (Betriebsmittel-)Zuteilung. Prozess wird durch Zuteilung Inhaber des Betriebsmittels.



7.2 Deadlocks

Def

- Definition:

Eine Menge von Prozessen befindet sich in einem Deadlock-Zustand, falls jeder Prozess der Menge auf ein Ereignis wartet, das nur ein anderer Prozess der Menge auslösen kann.

Man sagt auch:

Die Prozesse sind in einen Deadlock verstrickt.

Da alle Prozesse warten, kann keiner jemals ein Ereignis erzeugen, auf das einer der anderen wartet. Alle Prozesse warten also für immer.

Voraussetzungen für Deadlocks

Coffman (1971): Voraussetzungen für Deadlocks:

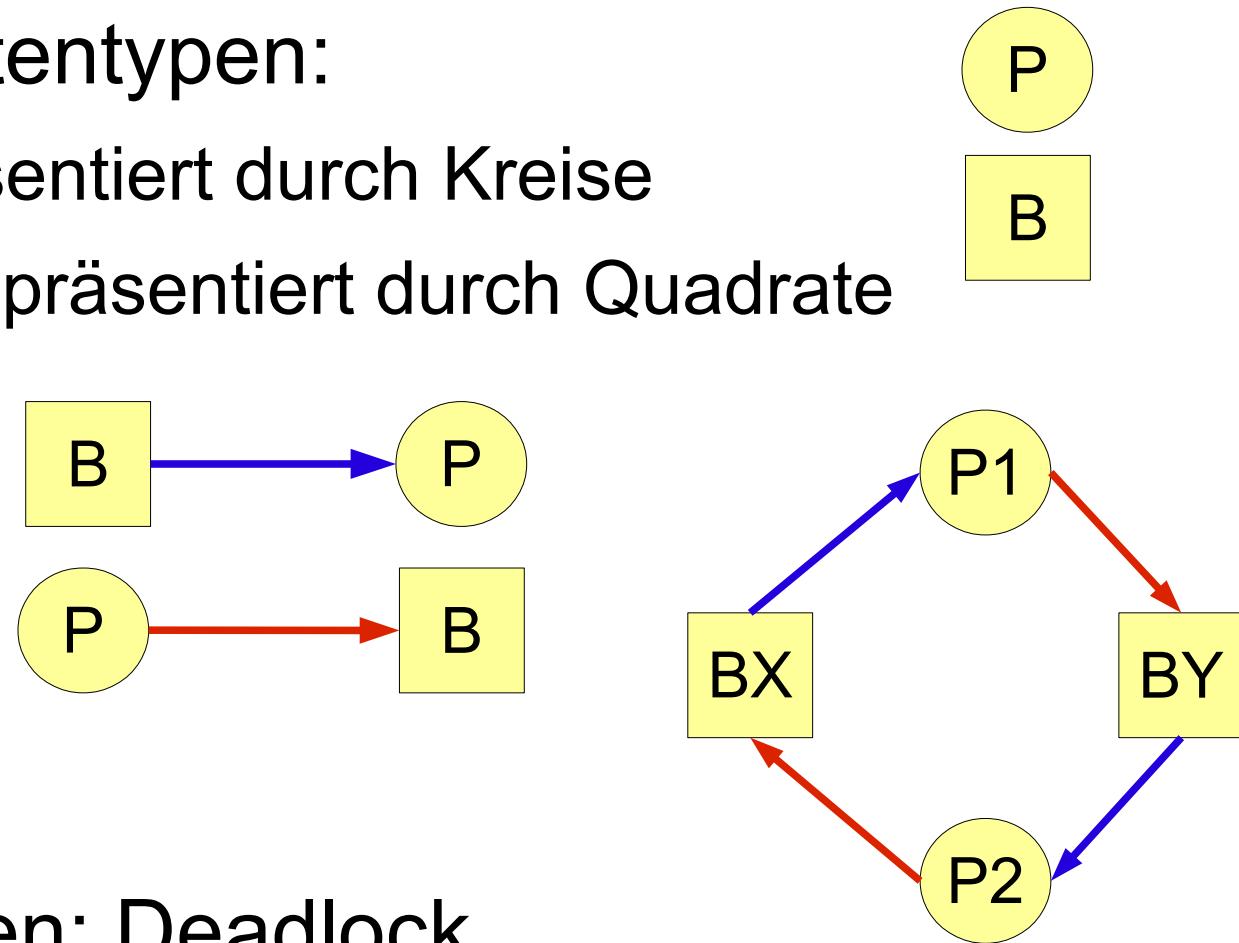
- **Wechselseitiger Ausschluß:**
Jedes Betriebsmittel ist entweder frei oder genau einem Prozess zugeteilt
- **Belegungs-/Anforderungs-Bedingung („Hold-and-wait“):**
Prozesse können zu bereits reservierten Betriebsmitteln noch weitere anfordern
- **Ununterbrechbarkeit:**
Einmal einem Prozess zugeteilte Betriebsmittel können nicht wieder ohne dessen Zustimmung (Freigabe) entzogen werden.
- **Zyklisches Warten:**
Es muss eine zyklische Kette von Prozessen geben, in der jeder Prozess auf ein Betriebsmittel wartet, das dem nächsten Prozess in der Kette gehört.

Alle vier Bedingungen gleichzeitig erfüllt => Ein Deadlock **ist möglich**
(falls eine nicht erfüllt ist => **kein Deadlock möglich**)

Belegungs-Anforderungs-Graphen



- Graphische Darstellung der Beziehung von Prozessen zu Betriebsmitteln (Holt, 1972)
- Es gibt zwei Knotentypen:
 - Prozesse, repräsentiert durch Kreise
 - Betriebsmittel, repräsentiert durch Quadrate
- Pfeile:
 - P belegt B
 - P wartet auf B
- Zyklus im Graphen: Deadlock



Beispiel

- Gegeben:
 - drei Prozesse A, B, C und
 - drei Betriebsmittel R, S, T

Prozess A

- Anforderung R
- Anforderung S
- Freigabe R
- Freigabe S

Prozess B

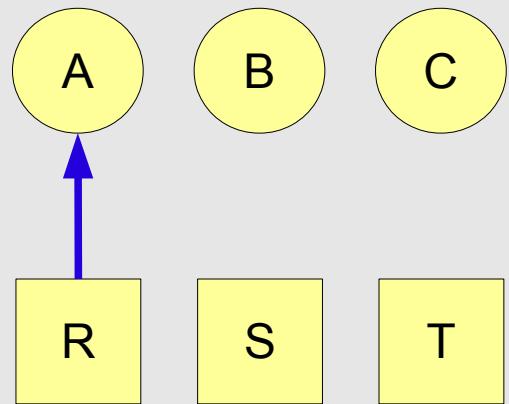
- Anforderung S
- Anforderung T
- Freigabe S
- Freigabe T

Prozess C

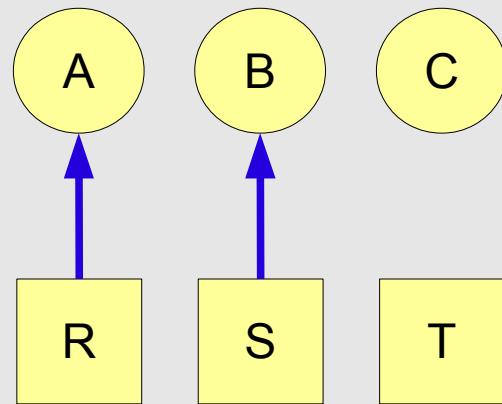
- Anforderung T
- Anforderung R
- Freigabe T
- Freigabe R

- Das Betriebssystem kann jeden (nicht blockierten) Prozess **jederzeit** ausführen
- Sequentielle Ausführung von A, B, C wäre unproblematisch (dann aber auch keine Nebenläufigkeit)
- Wie sieht es bei nebenläufiger Ausführung aus?

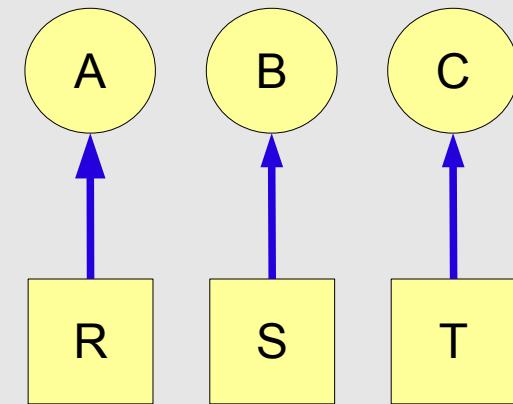
1. A fordert R an



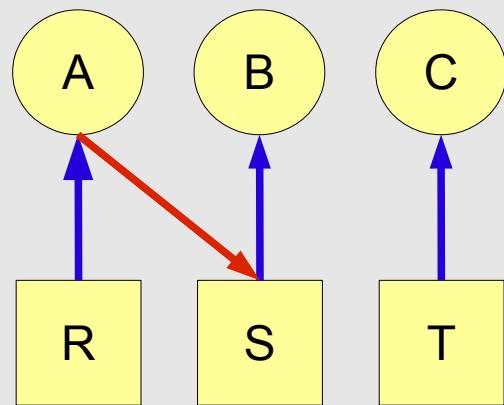
2. B fordert S an



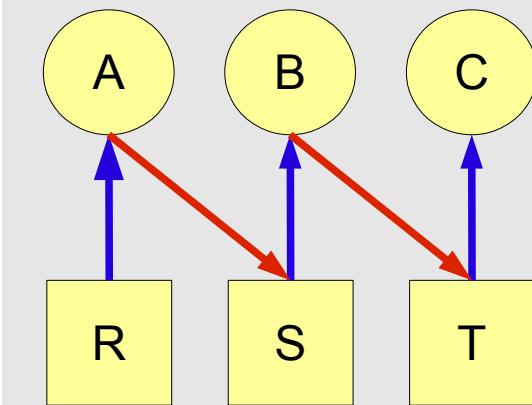
3. C fordert T an



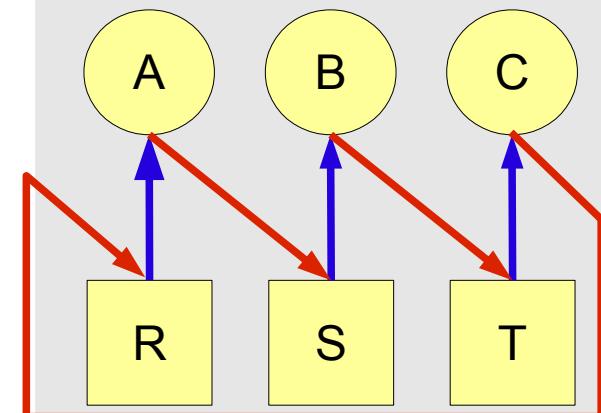
4. A fordert S an



5. B fordert T an



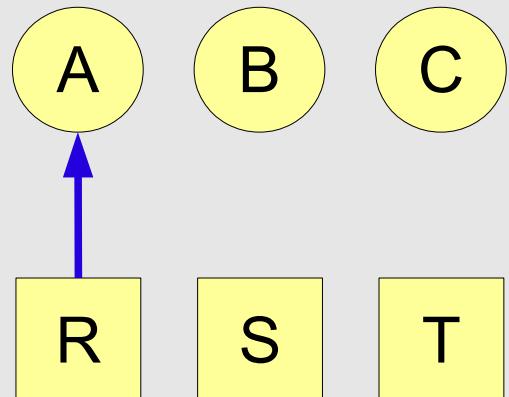
6. C fordert R an
Deadlock



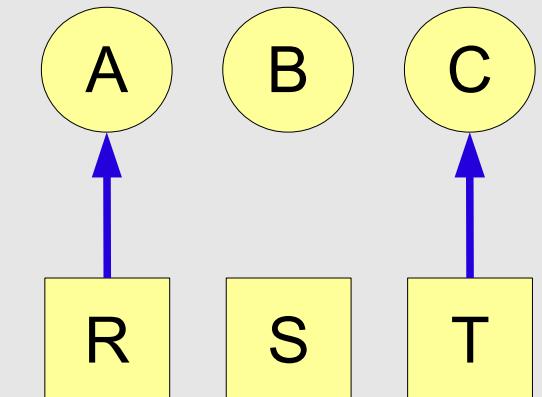
Ausführung II

(B wird zunächst suspendiert)

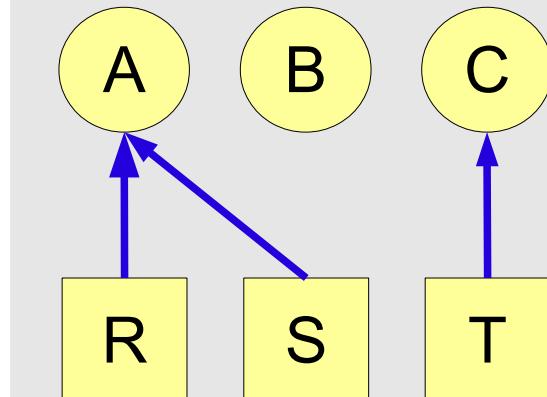
1. A fordert R an



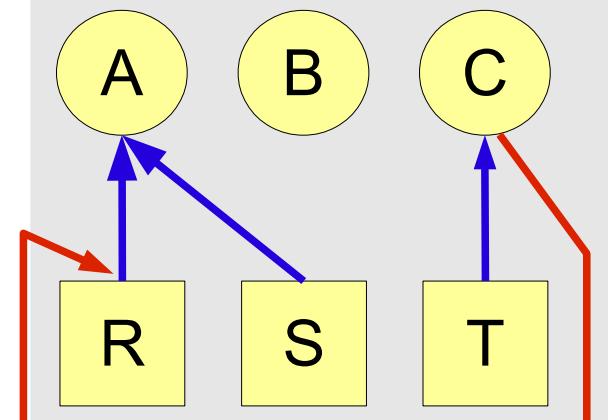
2. C fordert T an



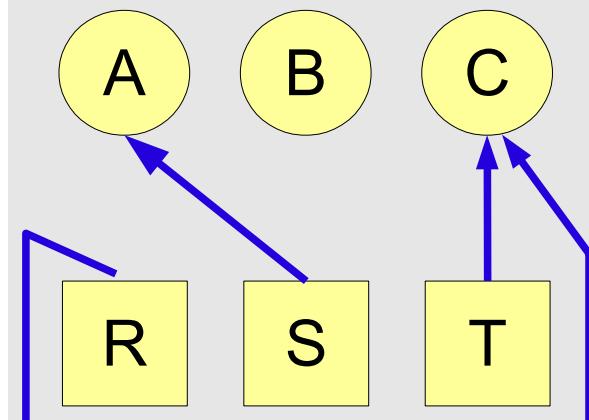
3. A fordert S an



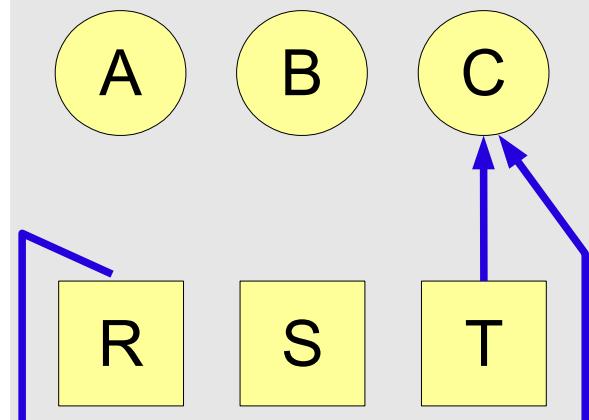
4. C fordert R an



5. A gibt R frei



6. A gibt S frei
kein Deadlock



Verfahren zur DL-Behandlung



- Mit Betriebsmittelzuteilungsgraphen („Belegungs/Anforderungs-Graphen“) lassen sich Deadlocks erkennen (→ Zyklus im Graph)
- Wie weiter verfahren?
 - **Ignorieren** („Vogel-Strauß-Verfahren“)
 - Deadlocks **erkennen und beheben**
 - **Verhinderung** durch Planung der Betriebsmittelzuordnung (*Deadlock Avoidance*)
 - **Vermeidung** durch Nichterfüllung (mindestens) einer der vier Voraussetzungen für Deadlocks (*Deadlock Prevention*)
 - Diese Strategien werden im folgenden untersucht

7.3 Ignorieren des Problems

- „Vogel-Strauß-Algorithmus“
- Ausdruck optimistischer Lebenshaltung:

„Deadlocks kommen in der Praxis sowieso nie vor“



- ...warum also dann Aufwand in ihre Vermeidung stecken?
- **Beispiel:**
 - UNIX-System mit z.B. 100 Einträge großer Prozesstabellen
 - 10 Programme versuchen gleichzeitig, je 12 Kindprozesse zu erzeugen
 - Deadlock nach 90 erfolgreichen `fork()`-Aufrufen (wenn keiner der Prozesse aufgibt)
 - Ähnliche Beispiele sind mit anderen begrenzt großen Systemtabellen möglich (z.B. inode-Tabelle)

...manchmal nicht so gut



- Engl.: Deadlock Detection and Resolution (Recovery)
- Vorgehensweise: Das Auftreten von Deadlocks wird vom Betriebssystem nicht verhindert. Es wird versucht, Deadlocks zu erkennen und anschließend zu beheben.
- Betrachtet werden im folgenden:
 1. Deadlock-Erkennung mit einem Betriebsmittel je Klasse (Einfacher Fall)
 2. Deadlock-Erkennung mit mehreren Betriebsmitteln je Klasse (Allgemeiner Fall)
 3. Verfahren zur Deadlock-Behebung

7.4.1 Deadlocks erkennen (Einfacher Fall)

- Vereinfachende Annahme: **Ein Betriebsmittel** je Betriebsmitteltyp
- **Vorgehen:**
 - erzeuge Belegungs-/Anforderungs-Graph
 - suche nach Zyklen
 - falls ein Zyklus gefunden wurde: Deadlock beheben (s.u.)
- **Wann** wird die Untersuchung durchgeführt?
 - bei **jeder** Betriebsmittelanforderung?
 - in **regelmäßigen** Zeitabständen?
 - wenn „**Verdacht**“ auf Deadlock besteht
(z.B. Abfall der CPU-Auslastung unter eine Grenze)

Beispiele: Sicher?



4 Prozesse, ein Betriebsmitteltyp (10 Stück vorhanden)

verfügbar: 10

Proz.	hat	max.
A	0	6
B	0	5
C	0	4
D	0	7

sicher!

z.B. sequentielle Ausführung von A, B, C, D in beliebiger Reihenfolge ist möglich

verfügbar: 2

Proz.	hat	max.
A	1	6
B	1	5
C	2	4
D	4	7

sicher!

C ist ausführbar,
(→ dann 4 verfügbar)
dann D, B, A möglich.

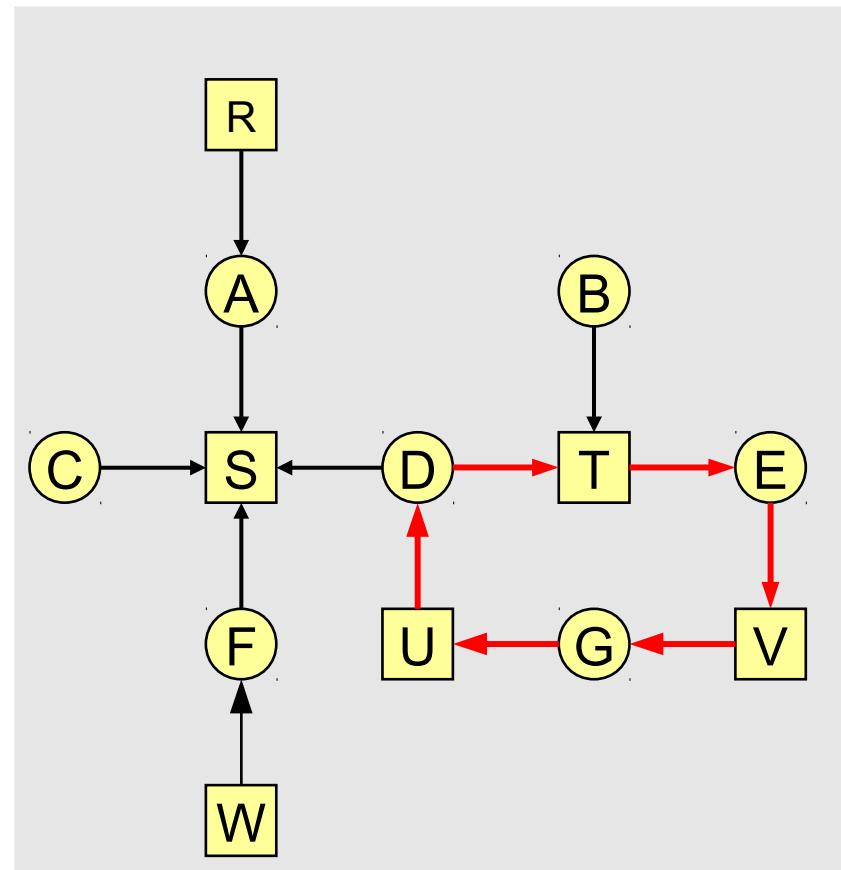
verfügbar: 1

Proz.	hat	max.
A	1	6
B	2	5
C	2	4
D	4	7

unsicher!

Differenz *max - hat* immer > *verfügbar*.
Deadlock, sobald irgendein Prozess auf sein Maximum zugeht

1. A belegt R und fordert S an.
2. B fordert T an.
3. C fordert S an.
4. D belegt U und fordert S und T an.
5. E belegt T und fordert V an.
6. F belegt W und fordert S an.
7. G belegt V und fordert U an.



Deadlocks erkennen

7.4.2

- Erweiterung: **Mehrere** (E_i -viele) Betriebsmittel je Betriebsmitteltyp i (z.B. mehrere Drucker)
- Prozesse P_1, \dots, P_n

Betriebsmittelvektor $E = (E_1, E_2, \dots, E_m)$ - Gesamtzahl der BM je Typ i

Verfügbarkeitsvektor $A = (A_1, A_2, \dots, A_m)$ - noch verfügbare BM je Typ i

Belegungsmatrix C : Zeile j gibt BM-Belegung durch Prozess j an
("Prozess j belegt C_{jk} Einheiten von BM k ")

Anforderungsmatrix R : Zeile j gibt BM-Anforderungen durch Prozess j an
("Prozess j fordert R_{jk} Einheiten von BM k ")

C_{11}	C_{12}	\dots	C_{1m}
C_{21}	C_{22}	\dots	C_{2m}
\dots	\dots	\dots	\dots
C_{n1}	C_{n2}	\dots	C_{nm}
R_{11}	R_{12}	\dots	R_{1m}
R_{21}	R_{22}	\dots	R_{2m}
\dots	\dots	\dots	\dots
R_{n1}	R_{n2}	\dots	R_{nm}

Erkennungsalgorithmus

7.4.2

- Zu Beginn sind alle Prozesse aus P unmarkiert
(Markierung heißt, dass der Prozess in keinem DL steckt)
-  Suche einen Prozess, der ungehindert durchlaufen kann, also einen unmarkierten Prozess P_i , dessen Zeile in der Anforderungsmatrix-Zeile R_i (komponentenweise) kleiner als oder gleich dem Verfügbarkeitsvektor A ist
- Kein passendes P_i gefunden? Dann **Ende**
- Gefunden? Dann kann P_i durchlaufen, gibt danach seine belegten Betriebsmittel zurück: $A = A + C_i$, wird markiert und es geht beim nächsten unmarkierten Prozess weiter
- Beim Ende des Verfahrens sind alle unmarkierten Prozesse an einem Deadlock beteiligt.

Beispiel

7.4.2

	Bandgeräte	Plotter	Scanner	CD-Brenner
$E = (4 \ 2 \ 3 \ 1)$ vorhanden				
$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$ Belegungen				
$A = (2 \ 1 \ 0 \ 0)$ verfügbar				
$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$ Anforderungen				

Ausführbar ist zunächst nur **P3**

$$\begin{aligned} \text{Freigabe von } C_3 &= (0 \ 1 \ 2 \ 0) \\ \Rightarrow A &= (2 \ 1 \ 0 \ 0) + (0 \ 1 \ 2 \ 0) \\ \Rightarrow A &= (2 \ 2 \ 2 \ 0) \end{aligned}$$

Nun ausführbar: **P2**
 (benötigt $R_2 = (1 \ 0 \ 1 \ 0)$)

Freigabe von $C_2 = (2 \ 0 \ 0 \ 1)$
 Danach: $A = (4 \ 2 \ 2 \ 1)$

Schließlich auch **P1** ausführbar
 $A = (4 \ 2 \ 3 \ 1)$

Alle Prozesse markiert,
 kein Deadlock aufgetreten.

7.4.3 Beheben von Deadlocks

7.4.2

Wie kann man auf erkannte Deadlocks reagieren?

- **Prozessunterbrechung**
 - Betriebsmittel zeitweise entziehen, anderem Prozess bereitstellen und dann zurückgeben
 - Kann je nach Betriebsmittel schwer oder nicht möglich sein
- **Teilweise Wiederholung (Rollback)**
 - System sichert regelmäßig Prozesszustände (Checkpoints)
 - Dadurch ist Abbruch und späteres Wiederaufsetzen möglich
 - Arbeit seit letztem Checkpoint geht beim Rücksetzen verloren und wird beim Neuaufsetzen wiederholt (ungünstig z.B. bei seit Checkpoint ausgedruckten Seiten)
 - Beispiel: Transaction Abort bei Datenbanken
- **Prozessabbruch**
 - Härteste, aber auch einfachste Maßnahme
 - Nach Möglichkeit Prozesse auswählen, die relativ problemlos neu gestartet werden können (z.B. Compilierung)



7.5 Verhindern von Deadlocks

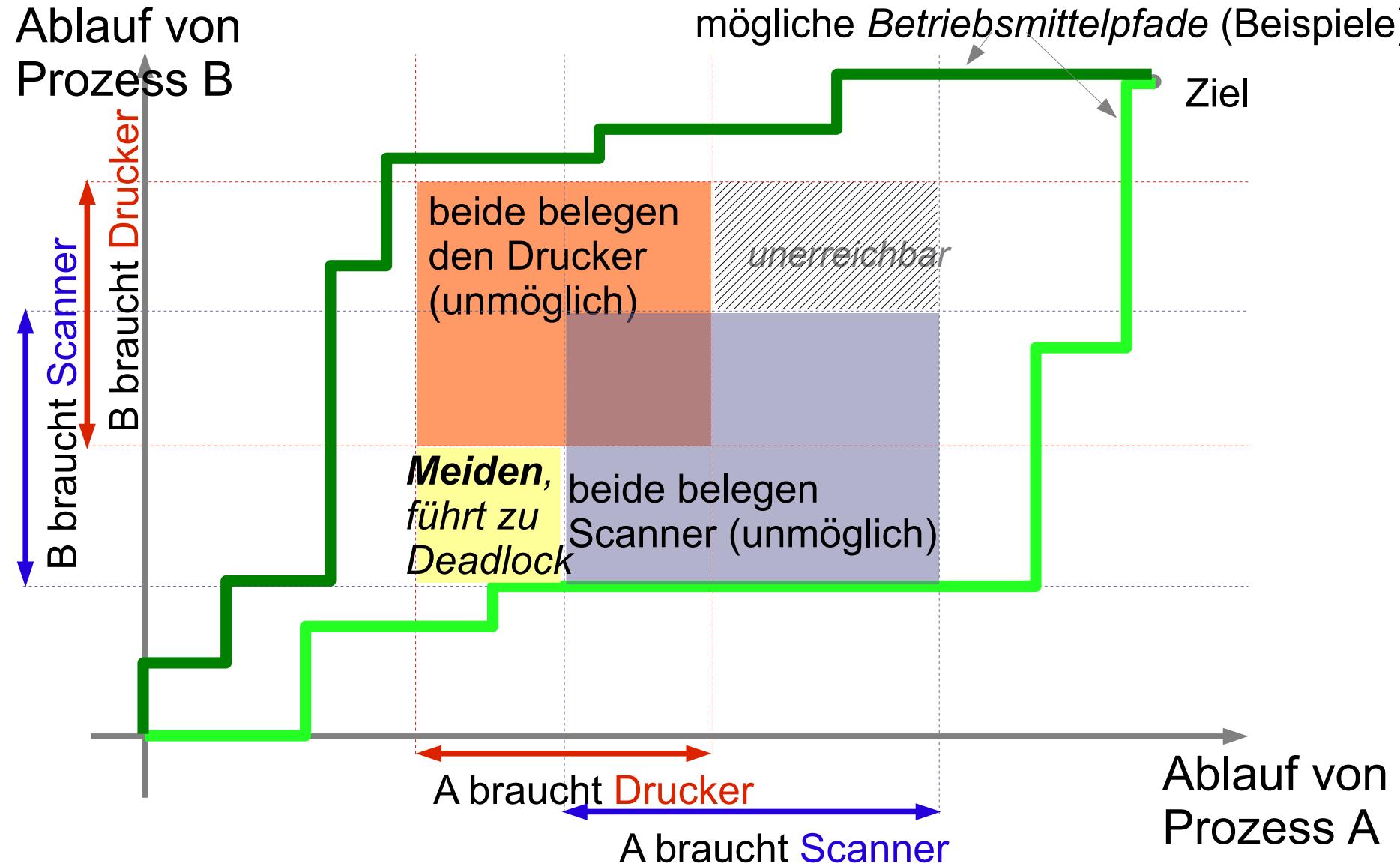
- Bisher: Erkennung von Deadlocks, gegebenenfalls „drastische“ Maßnahmen zur Auflösung
- Annahme bisher: Prozesse fordern alle Betriebsmittel „auf ein Mal“ an (vgl. 7.4.2).
- In den meisten praktischen Fällen werden BM jedoch nacheinander angefordert
- Betriebssystem muss dann dynamisch über Zuteilung entscheiden



7.5 Verhindern von Deadlocks

- Kann man **Deadlocks** durch „geschicktes“ Vorgehen bei der Betriebsmittelzuteilung **von vornherein verhindern**?
- Welche Informationen müssen dazu vorab zur Verfügung stehen?
- Im folgenden betrachtet
 1. Betriebsmittelpfade (Grafische Veranschaulichung)
 2. Sichere und unsichere Zustände
 3. Der vereinfachte Bankiersalgorithmus für eine BM-Klasse
 4. Der Bankiersalgorithmus für mehrere BM-Klassen

7.5.1 Betriebsmittelpfade



7.5.2 (Un-)Sichere Zustände

Def

- Ein Systemzustand ist **sicher**, wenn er
 - **keinen Deadlock** repräsentiert und
 - es eine geeignete Prozessausführungsreihenfolge gibt, bei der alle Anforderungen erfüllt werden
(die also *auch dann* nicht in einen Deadlock führt, wenn alle Prozesse gleich ihre max. Ressourcenanzahl anfordern)
- Sonst heißt der Zustand **unsicher**.
- Im folgenden: Datenstrukturen aus 7.4.2:
 - E Betriebsmittelvektor
 - A Verfügbarkeitsvektor
 - C Belegungsmatrix
 - R Anforderungsmatrix

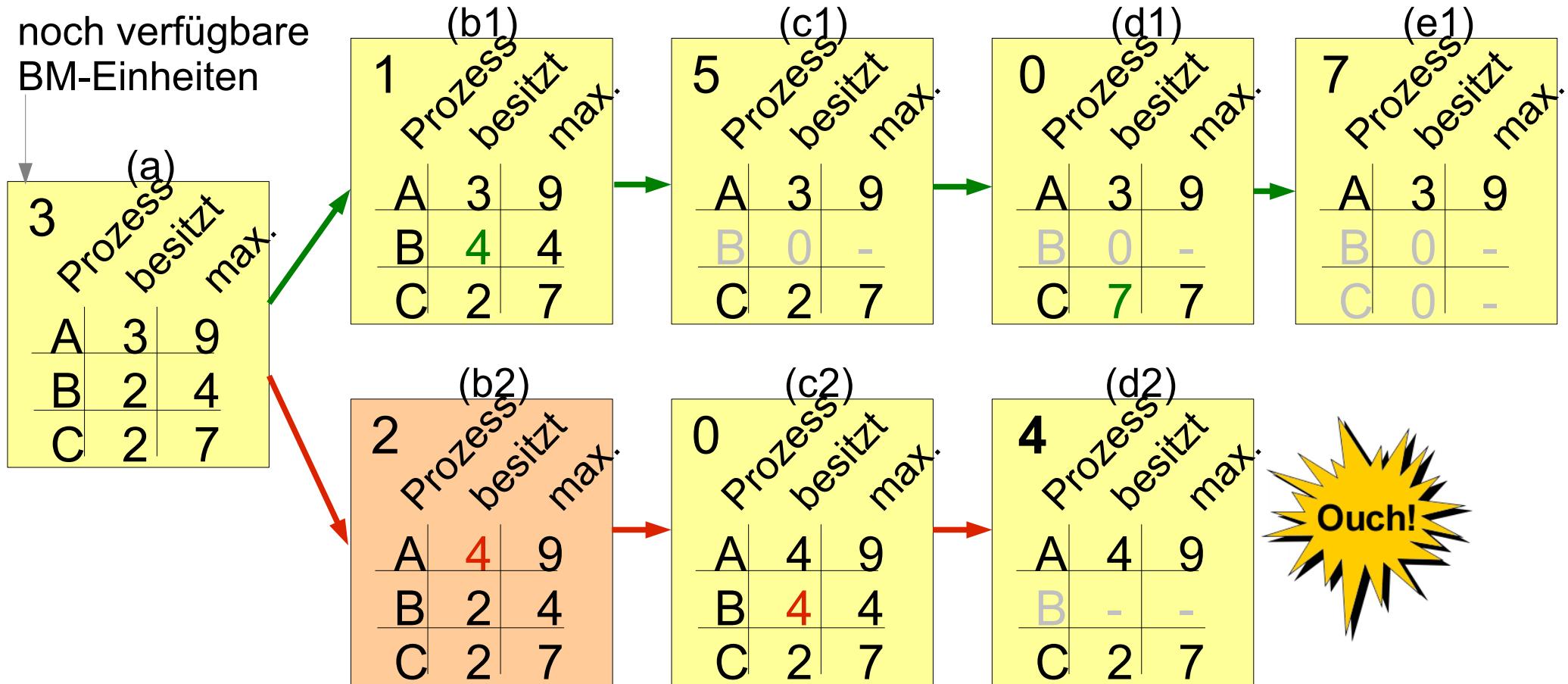
Was sagt uns das?

- Bei einem sicherem Zustand kann das System **garantieren**, dass alle Prozesse bis zum Ende durchlaufen können.
- Bei unsicherem Zustand ist das nicht garantierbar (aber auch nicht ausgeschlossen!).
 - Beispiel: Ein Prozess gibt ein BM zu einem „glücklichen“ Zeitpunkt kurzzeitig frei, wodurch eine Deadlock-Situation „zufällig“ vermieden wird. (→ „Glück“ nicht vorhersehbar)
 - „Unsicher“ bedeutet also *nicht* „Deadlock unvermeidlich“.

Beispiel

7.5.2

- 3 Prozesse A,B,C; jeweils mit BM-Besitz und max. Bedarf
- ein Betriebsmitteltyp, 10x vorhanden



- Zustand (a) ist sicher (es gibt eine DL-freie Lösung)
- (b2) ist **nicht** sicher (A und C brauchen je 5, frei sind nur 4)

7.5.3 Bankier-Algorithmus (1 BM-Klasse)



- Dijkstra (wer sonst? 1965):
- Ein **Bankier** kennt die **Kreditrahmen** seiner Kunden.
- Er geht davon aus, dass **nicht alle** Kunden **gleichzeitig** ihre Rahmen **voll** ausschöpfen werden.
- Daher hält er **weniger Bargeld** bereit als die **Summe** der Kreditrahmen.
- Gegebenenfalls **verzögert** er die **Zuteilung** eines Kredits, bis ein anderer Kunde zurückgezahlt hat.
- **Zuteilung** erfolgt **nur**, wenn sie "**sicher**" ist (also letztlich alle Kunden bis zu ihrem Kreditrahmen bedient werden können).
- Bankier = Betriebssystem, Bargeld = Betriebsmitteltyp, Kunden = Prozesse, Kredit = BM-Anforderung, z

Bankier-Algorithmus (2)

7.5.3

- Prüfe bei jeder Anfrage, ob die Bewilligung in einen sicheren Zustand führt:
 - **Teste** dazu, ob ausreichend Betriebsmittel bereitstehen, um **mindestens einen** Prozess **vollständig** zufrieden zu stellen.
 - Davon ausgehend, dass dieser Prozess nach Durchlauf seine Betriebsmittel freigibt: führe **Test** mit dem Prozess aus, der dann am nächsten am Kreditrahmen ist
 - usw., **bis alle** Prozesse positiv getestet sind;
 - Falls **ja**, kann die aktuelle Anfrage **bewilligt** werden.
 - **Sonst:** Anforderung **verschieben** (warten)

7.5.4 Verallgemeinerter Bankier-Alg.

7.5.3

- Mehrere Betriebsmittelklassen
- Daten wie bei „Deadlockerkennung“ (7.4.2)
- Matrizen mit belegten / angeforderten Betriebsmitteln
- Vektoren mit BM-Bestand, verfügbaren BM und belegten BM je Betriebsmitteltyp
 - E Betriebsmittelvektor
 - A Verfügbarkeitsvektor („Betriebsmittelrestvektor“)
 - C Belegungsmatrix
 - R Anforderungsmatrix

Verallgemeinerter Bankier-Algorithmus

7.5.4

- **Verfahren:**

- Zu Beginn sind alle Prozesse unmarkiert.
- **Suche** Zeile aus Anforderungsmatrix, die kleiner oder gleich dem Verfügbarkeitsvektor ist (falls keine existiert, wird das System in einen Deadlock laufen)
- **Markiere** zugehörigen Prozess und **addiere** seine Betriebsmittel (zugehörige Zeile in der Belegungsmatrix) zum Verfügbarkeitsvektor
- **Wiederhole** die letzten beiden Schritte,
 - bis alle Prozesse markiert sind (→Zustand ist sicher)
 - oder ein Deadlock entdeckt wurde (→Zustand unsicher)
- Falls sicher → Teile BM zu
- Falls unsicher → Anfordernden Prozess blockieren
- Bei Beendigung eines Prozesses werden alle verbleibenden Prozesse geweckt und die Anforderungen gemäß Verfahren neu bearbeitet

Beispiel

$$E = (6 \ 3 \ 4 \ 2) \quad \text{vorhanden}$$

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{zugewiesen}$$

$$A = (1 \ 0 \ 2 \ 0) \quad \text{verfügbar}$$

$$P = (5 \ 3 \ 2 \ 2) \quad \text{belegt}$$

$$R = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix} \quad \text{angefordert}$$

Sicher? Ja, Ausführungsfolge
 P4, P1, P5, ... ist möglich:

$$P4 \rightarrow A = (2 \ 1 \ 2 \ 1)$$

$$P1 \rightarrow A = (5 \ 1 \ 3 \ 2)$$

$$P5 \rightarrow A = (5 \ 1 \ 3 \ 2)$$

$$P2 \rightarrow A = (5 \ 2 \ 3 \ 2)$$

$$P3 \rightarrow A = (6 \ 3 \ 4 \ 2)$$

Beispiel

7.5.4

P2 fordere ein BM 3 an (rot)

Sicher? Ja (P4, P1, P5, P2, P3)

P4 \rightarrow A = (2 1 1 1)

P1 \rightarrow A = (5 1 2 2)

P5 \rightarrow A = (5 1 2 2)

P2 \rightarrow A = (5 2 3 2)

P3 \rightarrow A = (6 3 4 2)

also erhält P2 ein BM 3

$E = (6 \ 3 \ 4 \ 2)$ vorhanden

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ zugewiesen}$$

$A = (1 \ 0 \ 1 \ 0)$ verfügbar

$P = (5 \ 3 \ 3 \ 2)$ belegt

$$R = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix} \text{ angefordert}$$

Beispiel

$E = (6 \ 3 \ 4 \ 2)$ vorhanden

$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ zugewiesen

$A = (1 \ 0 \ 0 \ 0)$ verfügbar

$P = (5 \ 3 \ 4 \ 2)$ belegt

$R = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix}$ angefordert

Nun fordere auch P5 ein BM 3 an
 → dann würde $A = (1 \ 0 \ 0 \ 0)$

Sicher? *Nein!*
 → daher Anfrage von P5 blockieren

- In der Praxis gibt es mehrere **Probleme** beim Einsatz:
 - Prozesse können "maximale Ressourcenanforderung" selten im Voraus angeben
 - Anzahl der Prozesse ändert sich ständig
 - Ressourcen können verschwinden (z.B. durch Ausfall)

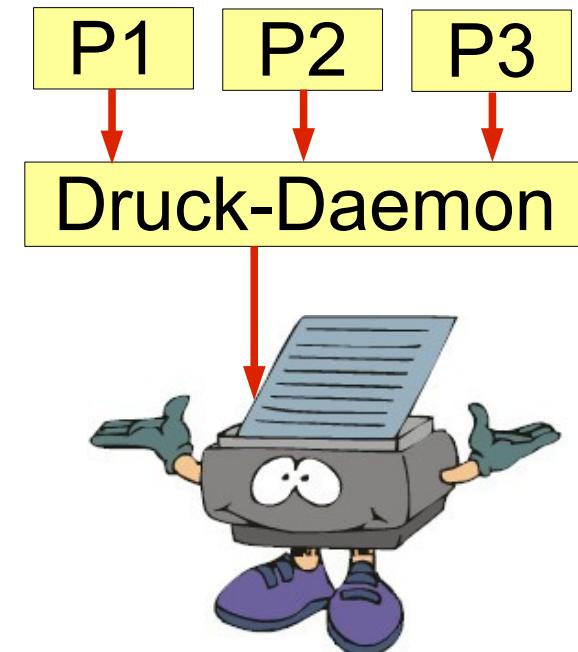


7.6 Deadlock-Vermeidung

- Deadlock-Verhinderung ist wenig praktikabel :-)
- Ansatz: **Vermeidung** mindestens einer der vier Deadlock-Voraussetzungen (vgl 7.2)
 - Wechselseitiger Ausschluss
 - Belegungs-/Anforderungsbedingung („Hold-and-Wait“, d.h. zu reservierten BM weitere anforderbar)
 - Ununterbrechbarkeit (kein erzwungener BM-Entzug)
 - zyklisches Warten

7.6.1 Wechselseitiger Ausschluß?

- Falls es keine exklusive Zuteilung eines Betriebsmittels an einen Prozess gibt, gibt es auch keine Deadlocks.
- **Beispiel:** Zugriff auf Drucker
- Einführung eines **Spool-Systems**, das
 - Druckaufträge von Prozessen (schnell) entgegennimmt
 - ggf. zwischenspeichert
 - und der Reihe nach auf dem Drucker ausgibt
- **Entkopplung** zwischen (konkurrierenden) Prozessen und dem (langsam) Betriebsmittel
- **Vermeidung** einer exklusiven Zuteilung des Betriebsmittels „Drucker“





7.6.2 Belegungs-/Anforderungsbed.?

- Vermeiden, dass neue Betriebsmittel-Anforderungen zu bereits bestehenden hinzukommen.
- „**Preclaiming**“: Alle Anforderungen zu Beginn der Ausführung stellen („alles oder nichts“)
- **Vorteil:** Wenn Anforderungen erfüllt werden, kann der Prozess bestimmt bis zum Ende durchlaufen (er hat ja dann alles, was er braucht)
- **Nachteil:**
 - Anforderungen müssen **zu Beginn bekannt** sein
 - Betriebsmittel werden unter Umständen **lange blockiert**
 - und können zwischenzeitlich nicht (sinnvoll) anders genutzt werden.
- **Beispiel:** Batch-Jobs bei Großrechnern.



7.6.3 Ununterbrechbarkeit?

- Hängt vom Betriebsmittel ab, aber
- „gewaltsamer“ Entzug ist in der Regel nicht akzeptabel
 - Drucker?
 - CD-Brenner?



7.6.4 Zyklische Wartebedingung?

- Wenn es kein zyklisches Auf-einander-warten gibt, entstehen auch keine Deadlocks
- Idee:
 - Betriebsmitteltypen **linear ordnen** und
 - nur in aufsteigender Ordnung Anforderungen annehmen (wenn mehrere Exemplare eines Typs gebraucht werden: alle Exemplare auf einmal anfordern)
 - „Drucker vor Scanner vor CD-Brenner vor ...“
- Dadurch entsteht **automatisch** ein **zyklenfreier** Belegungs-Anforderungs-Graph,
- wodurch Deadlocks ausgeschlossen sind.
- Tatsächlich praktikables Verfahren.

- Deadlock-Vermeidung durch Verhinderung (mindestens) einer der 4 Vorbedingungen eines Deadlocks ist möglich:

- Wechselseitiger Ausschluß → Spooling
- Belegungs-/Anforderungsbed. → Preclaiming
- Ununterbrechbarkeit *(BM-Entzug... besser nicht)*
- Zyklisches Warten → Betriebsmittel ordnen



7.7 Verwandte Fragestellungen

- Deadlocks bei der Benutzung von Semaphoren
(vgl. Kap. 3)
- Zwei-Phasen-Locking in Datenbanken
- Verhungern (Starvation), kein Deadlock, aber auch
kein Fortschritt für einen Prozess
(vgl. Philosophen-Problem)



7.8 Zusammenfassung

- Deadlocks sind ein Problem in jedem Betriebssystem aber auch in nebenläufigen Anwendungssystemen.
- Deadlocks treten auf, wenn einer Menge von Prozessen jeweils exklusiv ein Betriebsmittel zugeteilt ist, und alle ein Betriebsmittel anfordern, das bereits von einem anderen Prozess der Menge belegt ist. Alle Prozesse sind blockiert, keiner kann jemals fortgeführt werden.
- Deadlocks können durch vier notwendige Bedingungen charakterisiert werden:
 1. Bedingung des wechselseitigen Ausschlusses
 2. Belegungs- / Anforderungs-Bedingung
 3. Ununterbrechbarkeitsbedingung
 4. Zyklische Wartebedingung

Zusammenfassung (2)

- Deadlocks können vermieden werden (7.5), indem überprüft wird, ob der nachfolgende Zustand sicher ist oder nicht.
 - Zustand ist sicher, wenn es Folge von Ereignissen gibt, so dass alle Prozesse ihre Ausführung beenden können.
 - In unsicherem Zustand gibt es keine Garantie dafür.
 - Der Bankier-Algorithmus vermeidet Deadlocks, in dem er Anforderungen zurückstellt, die das System in einen unsicheren Zustand überführen würden.
- Deadlocks können durch konstruktive Verfahren verhindert werden (7.6)
 - z.B. durch Vorabbelegen aller Betriebsmittel
 - durch die geordnete Betriebsmittelbenutzung.

Heute: Kleiner Ausflug in die Rechnerarchitektur

Kap. 8: Ca..

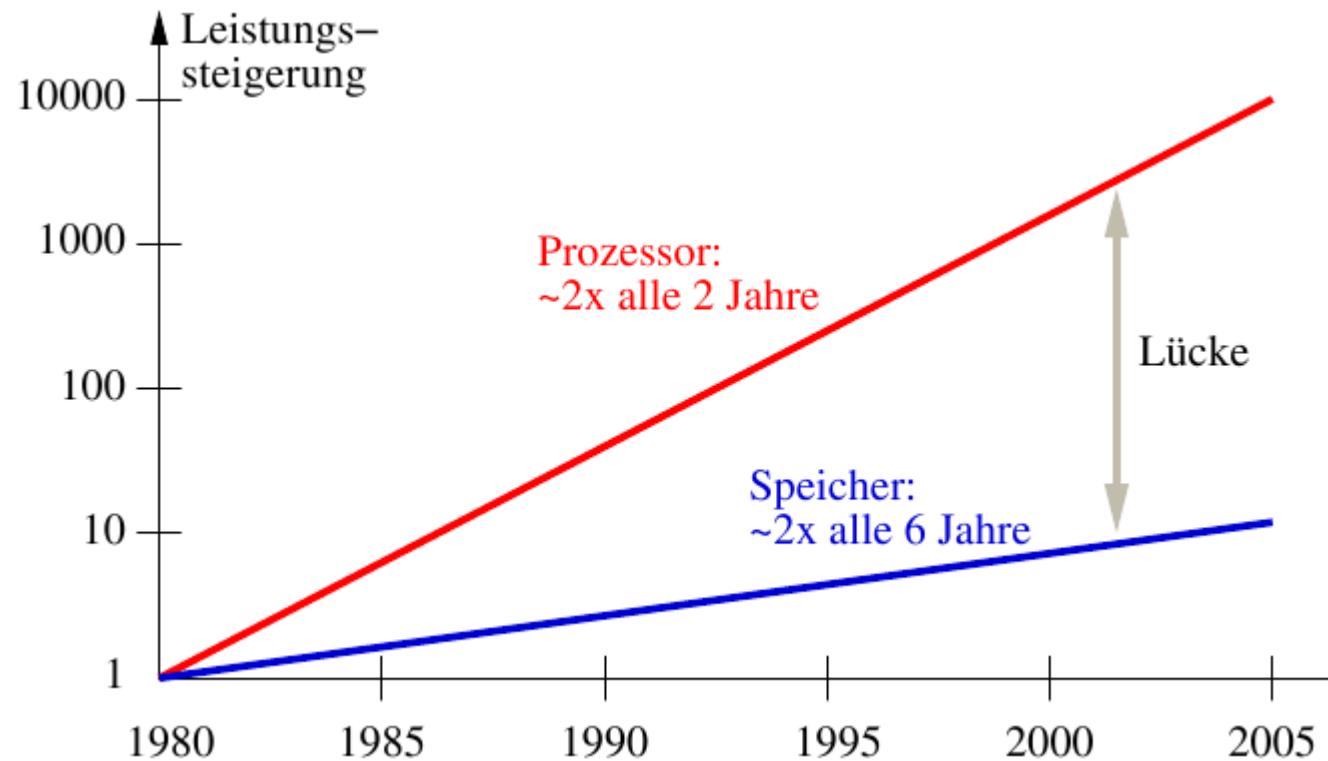


Cache !



Wozu Caches? (1)

- Prozessorgeschwindigkeit hat sich in den letzten 30 Jahren mehr als vertausendfacht
- Speicherzugriffsgeschwindigkeit ist zwar auch gestiegen, aber deutlich weniger:



- Prozessortakt > 10 x Speichertakt ist heute keine Seltenheit → Speicher wird zum „Flaschenhals“
- Skalare Prozessoren: 1 Befehl pro Takt (Superskalare: noch mehr ...)
- Wie soll das gehen?
- Ein Maschinenbefehl besteht mindestens aus
 - einem Speicherzugriff (Opcode fetch)
 - + evtl. auch noch mehrere Operanden
- Lösung: schneller, dafür kleiner Zwischenspeicher zw. Prozessor und Hauptspeicher: Cache⁽¹⁾

(1) Engl. *Cache*: Depot, geheimes Versteck

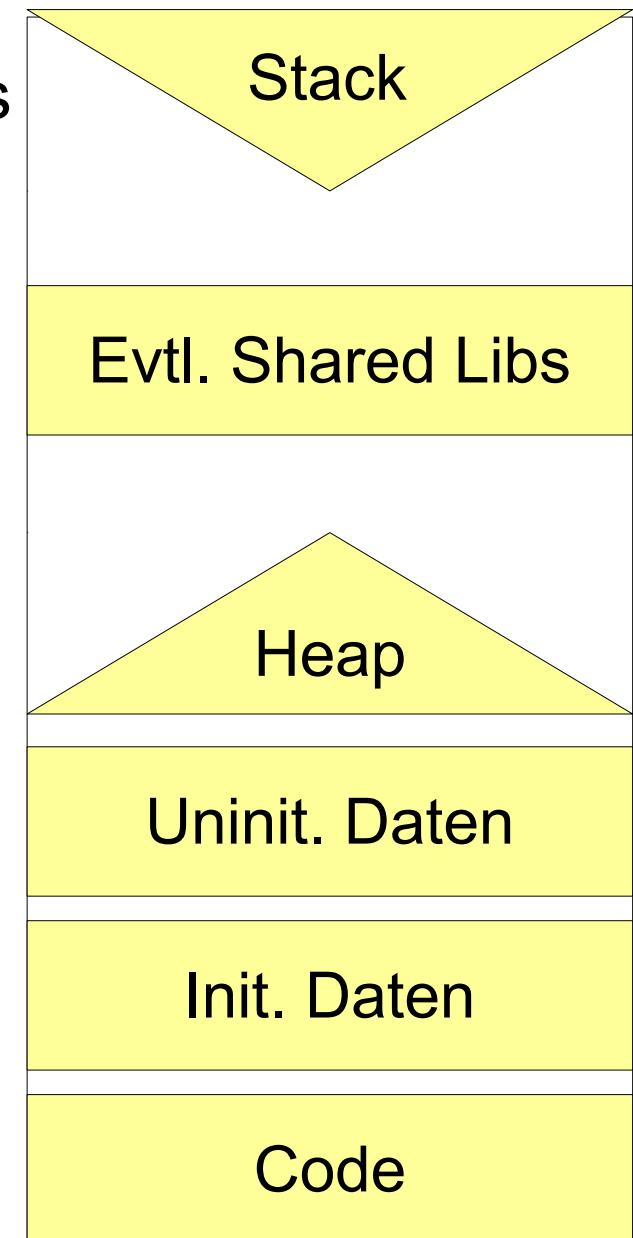
- Cache gehört zur Mikroarchitektur, nicht zur ISA, ist also aus Programmierersicht transparent
- Warum sich hier damit beschäftigen?
- Gründe:
 - Erzielte/erzielbare Performance hängt von Lokalitätseigenschaften (s.u.) der Programme ab
 - Durch geschicktes/ungeschicktes Coden kann die Performance u.U. um Größenordnungen variieren
 - Im Bereich der Betriebssysteme sind Caches manchmal doch nicht so ganz transparent
 - DMA, Debuggen, Laden von Programmen
- → Grundlegende Kenntnisse über die Arbeitsweise von Caches sind erforderlich



- **Räumliche Lokalität:** Zugriffe häufig auf Adresse in der Nähe bereits zuvor benutzer Adressen
- **Zeitliche Lokalität:** Zugriffe auf dieselbe oder benachbarte Adressen zeitlich nahe beieinander
- Trifft zu auf:
 - Befehlszugriffe: meistens (ausser bei Sprüngen)
 - Datenzugriffe: Programmabhängig
- Caches nutzen diese Eigenschaften:
 - Nachgefragte Daten und der sie umgebende Speicherblock (Cache-Zeile) werden möglichst lange zwischengespeichert
 - Wird das gleiche (oder ein benachbartes Datum) erneut nachgefragt, wird es aus dem Zwischenspeicher geliefert
 - → geht wesentlich schneller

Begriff: Working Set

- **Working set:** Gesamtheit der Speicherobjekte, auf die ein Prozess während eines gegebenen Zeitintervalls zugreift
- **Ziel:** möglichst viel vom Working Set eines Prozesses im Cache halten
- Programme arbeiten i.d. R. auf verschiedenen „Sektionen“:
 - Programmcode (`.text`)
 - Initialisierte Daten (`.data`)
 - Uninitialisierte Daten (`.bss`)
 - Heap (z.B. von `malloc()`)
 - Stack
 - Evtl. shared Libraries
- ➔ Working Set besteht aus >5-6 „Regionen“



- Programm greift auf ein Objekt (z.B. Variable) zu
- Zwei Möglichkeiten:

1. Cache Hit: Daten sind bereits im Cache → schnell

2. Cache Miss: Daten sind nicht im Cache, Hauptspeicherzugriff erforderlich → langsam



- Der Cache hält **Kopien** von im Speicher liegenden Objekten → Gefahr von Inkonsistenzen
- Cache ist **konsistent** → Alle Cache-Kopien und Original im Hauptspeicher haben gleichen Wert
- Cache ist **kohärent** → Cache und Hauptspeicher liefern für das gleiche Objekt den gleichen Wert
- Kohärenz betrifft das korrekte Arbeiten von Programmen
 - Inkohärenz führt zu Fehlern → vermeiden!
 - (Vorübergehende) Inkonsistenz ist tolerabel, solange sie nicht zu Inkohärenz führt
- Es gibt verschiedene Verfahrensweisen um Kohärenz sicherzustellen (sog. Kohärenzprotokolle)
→ hier nur Grundlagen behandelt

A) Bei Lesezugriff:

- **Cache Hit** → Daten aus dem Cache liefern, keine weiteren Aktionen
- **Cache Miss** → Daten werden aus dem Speicher geliefert, dabei in den Cache kopiert

B) Bei Schreibzugriff:

- **Cache Hit** → zwei Möglichkeiten:
 - 1) **Write Through**: Geschriebene Daten werden zugleich im Cache und im Speicher gespeichert (einfach aber langsam)
 - 2) **Copy-Back**: Datum wird zunächst nur im Cache gespeichert, zugehörige Cache-Zeile als „dirty“ markiert
→ System wird vorübergehend inkonsistent
- **Cache Miss** → auch zwei Möglichkeiten:
 - 1) **No Write Allocate**: Datum wird nur in Speicher geschrieben
 - 2) **Write Allocate**: Datum wird in Speicher und Cache geschrieben (zusammen mit umliegender Cachezeile)

- Effektive Wartezeit T_{eff} hängt ab von:
 - Wartezeit bei Zugriff auf Hauptspeicher (Cache miss): T_{miss}
 - Wartezeit bei Zugriff auf Cache (Cache hit): T_{hit}
 - Wahrscheinlichkeit eines Cache-Hit (*Trefferrate*): H

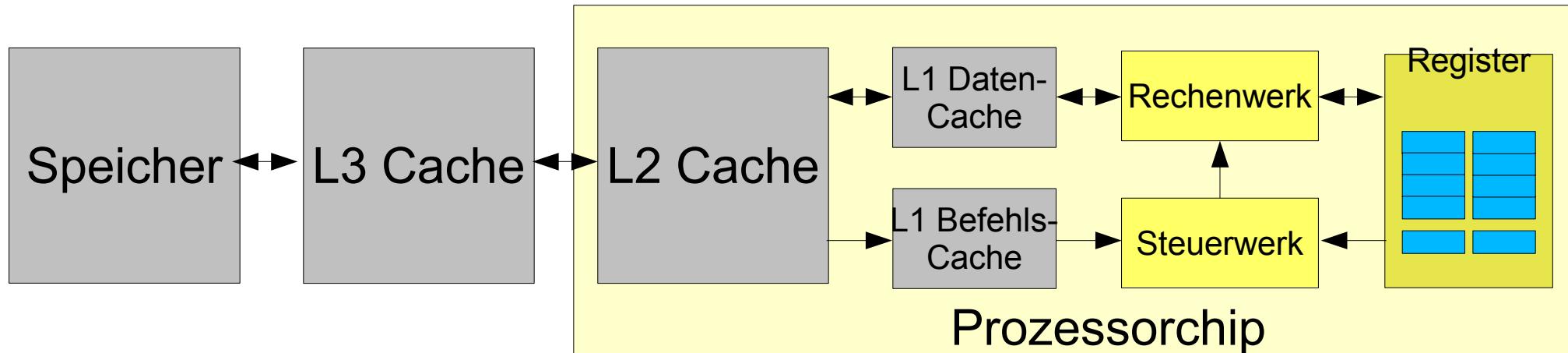
$$T_{\text{eff}} = H * T_{\text{hit}} + (1 - H) * T_{\text{miss}}$$

- Beispiel: $H = 80\%$, $T_{\text{hit}} = 0$, $T_{\text{miss}} = 20\text{ns}$:
 - $\Rightarrow T_{\text{eff}} = 0,8 * 0 + 0,2 * 20\text{ns} = 4\text{ns}$

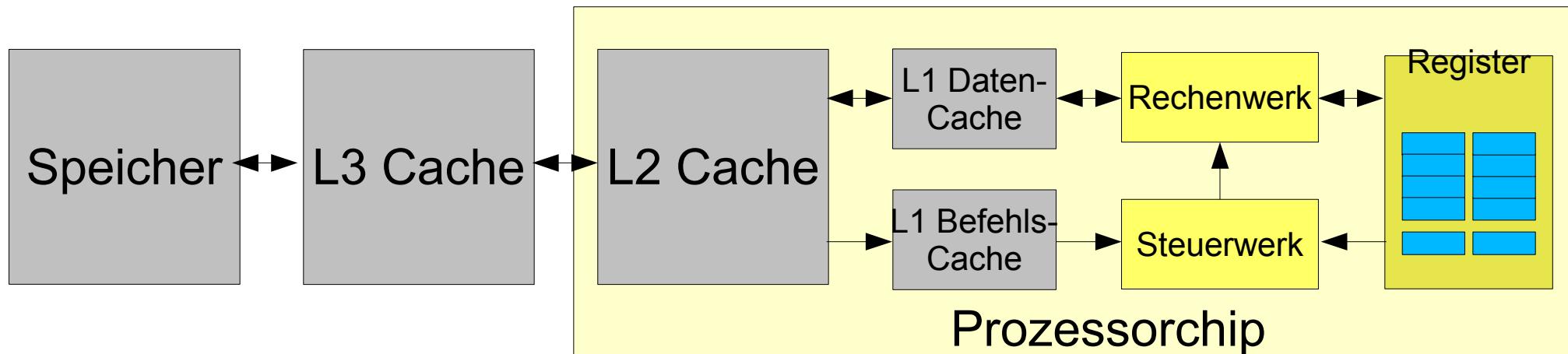
- z.B. Zweistufiger Cache (Level 1 und Level 2):

$$T_{\text{eff}} = H_{L1} * T_{L1\text{hit}} + (1 - H_{L1}) * (H_{L2} * T_{L2\text{hit}} + (1 - H_{L2}) * T_{\text{miss}})$$

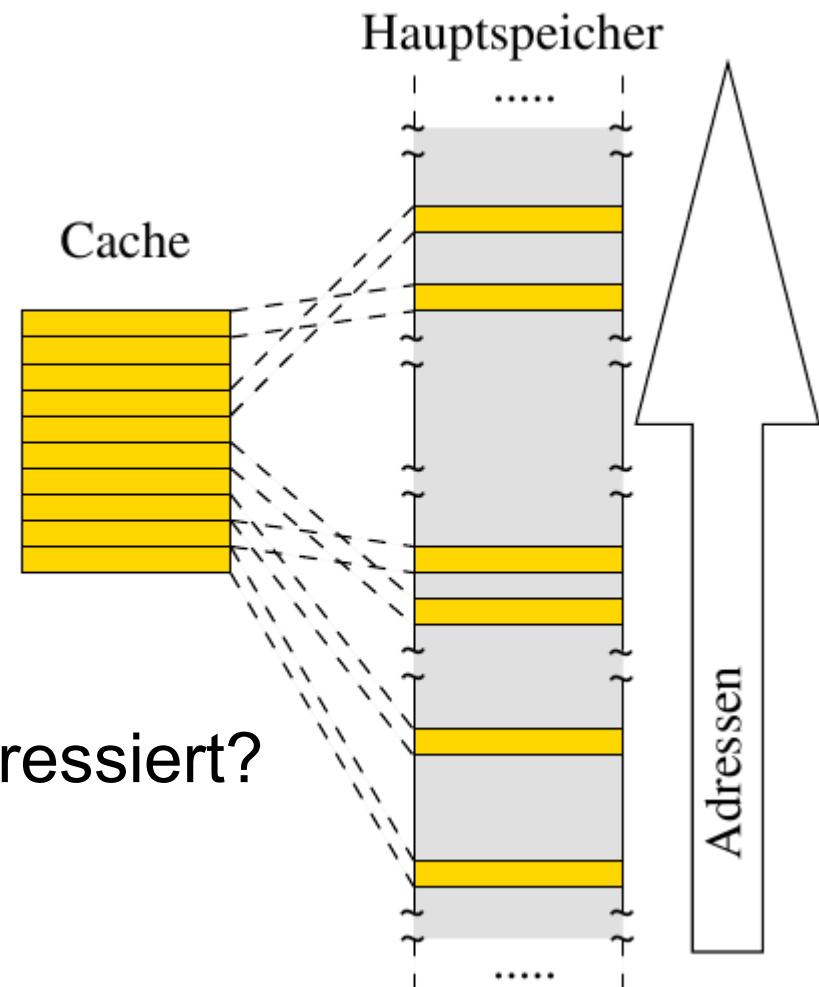
- Beispiel: $H_{L2} = 90\%$, $T_{L2\text{hit}} = 4\text{ns}$, $T_{\text{miss}} = 20\text{ns}$:
 $\Rightarrow T_{\text{eff}} = 0,8 * 0 + 0,2 * (0,9 * 4\text{ns} + 0,1 * 20\text{ns}) = 1,12\text{ns}$
- Caches heute meist mehrstufig:



- Heute bei PCs gängige Größen
 - L1 Cache: On-Chip, getrennt für Befehle und Daten, je 16-64kB
 - L2 Cache: On-Chip, gemeinsam für Befehle und Daten, 256kB bis 4MB
 - L3 Cache: Optional, Off-Chip
(z.B. schnelles ECL-RAM mit 5ns Zugriffszeit)



- Bisher betrachtete Speicher ordnen Adressen → Daten zu
- Cache: hält Kopien relativ kleiner⁽¹⁾ Hauptspeicher-Ausschnitte
- Zuordnung zwischen Speicher-objekt und Cache-Zeile ergibt sich aus der Zugriffsreihenfolge
- Kein Zusammenhang zwischen Adresse eines Objekts und dem Ort seiner Cache-Kopie
- Adresse (bzw. Index) einer Cachezeile ist ohne Bedeutung
- Wie aber werden Cache-Zeilen adressiert?



(1) Cache-Zeilen Größe: einige zehn bis hundert Byte



Assoziativspeicher (2)

- **Assoziativspeicher** (auch „inhaltsadressierter Speicher“)
(engl. *Content Addressable Memory* – CAM)
- **Assoziation von Wertepaaren**
(hier: Speicheradresse und Cache-Zeile)
(Auch der Mensch „speichert“ assoziativ)
- Ein Cache-Eintrag besteht aus:
 - Einem Identifikationsteil, dem *Tag* (Etikett)
 - Einem Datenbereich (Cache-Zeile),
der die eigentliche Kopie des Speicherinhaltes enthält
 - Einem Verwaltungsteil: z.B. Bits „V“ (Valid) und „D“ (Dirty)
 - V=1 → Cache-Zeile ist in Benutzung
 - D=1 → Inhalt der Cache-Zeile wurde verändert → ist inkonsistent

- Arbeitsweise:
 - Bei Zugriff auf ein Speicherobjekt: Adresse wird **gleichzeitig** mit den Tags **aller** Cache-Einträge verglichen
 - Existiert ein Eintrag mit übereinstimmendem Tag und $V = 1 \rightarrow \text{Cache Hit}$: Gültige Kopie ist im zugehörigen Datenbereich
 - Existiert kein solcher Eintrag $\rightarrow \text{Cache Miss}$

- Modellierung in C:

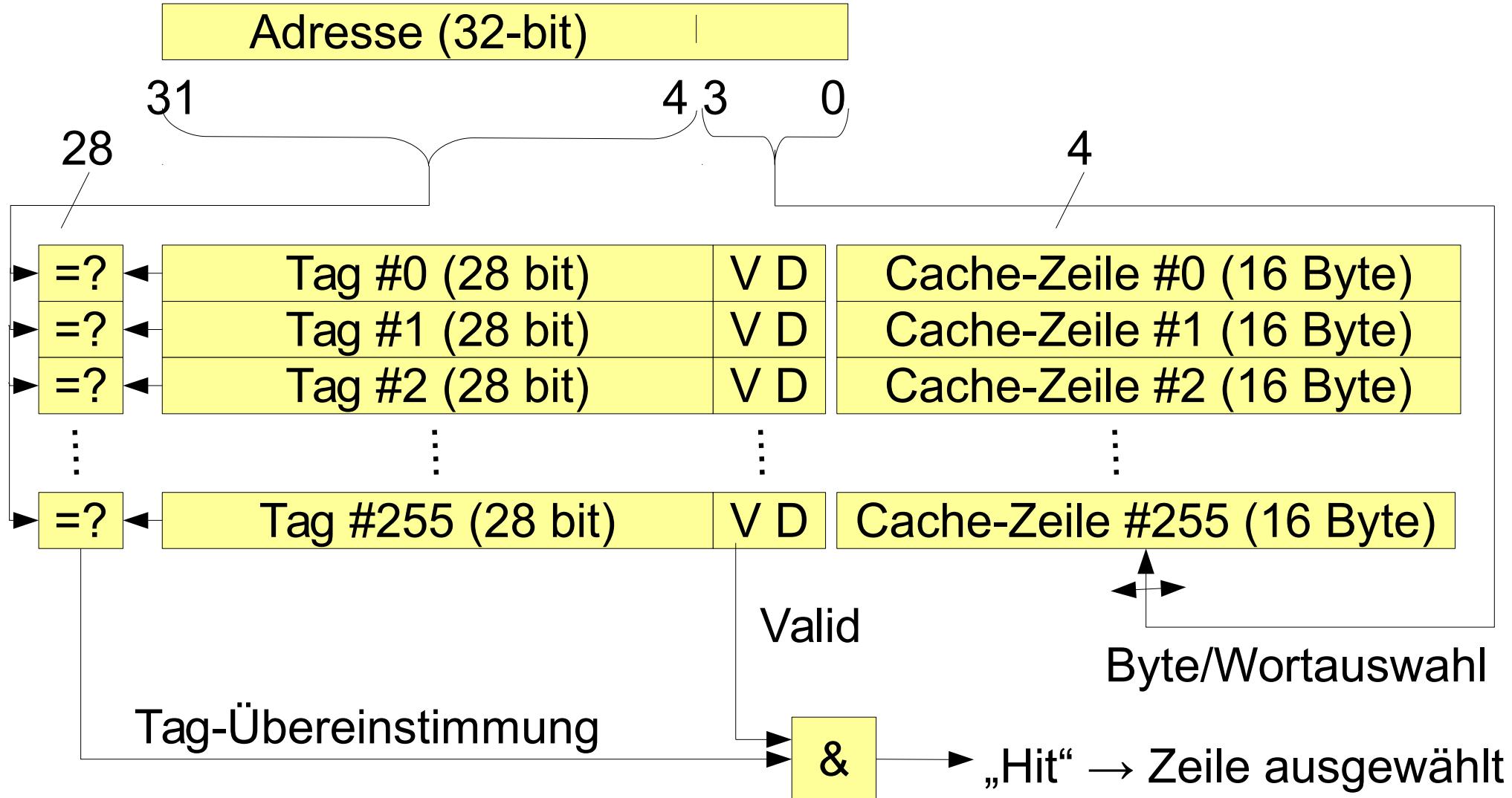
```
struct cache_eintrag {  
    void *tag;  
    int V, D;  
    unsigned int cache_zeile[CacheSize];  
};  
  
struct cache_eintrag cache[ANZ_CACHEZL];  
  
unsigned int *finde_eintrag(void *Adresse)  
{  
    for(i = 0; i < ANZ_CACHEZL; i++) {  
        if(cache[i].tag == Adresse && cache[i].V)  
            / * cache hit * /  
            return(&cache[i].cache_zeile[0]);  
    }  
    return(NULL); / * cache miss * /  
}
```

In Wirklichkeit: parallele Suche
in Hardware

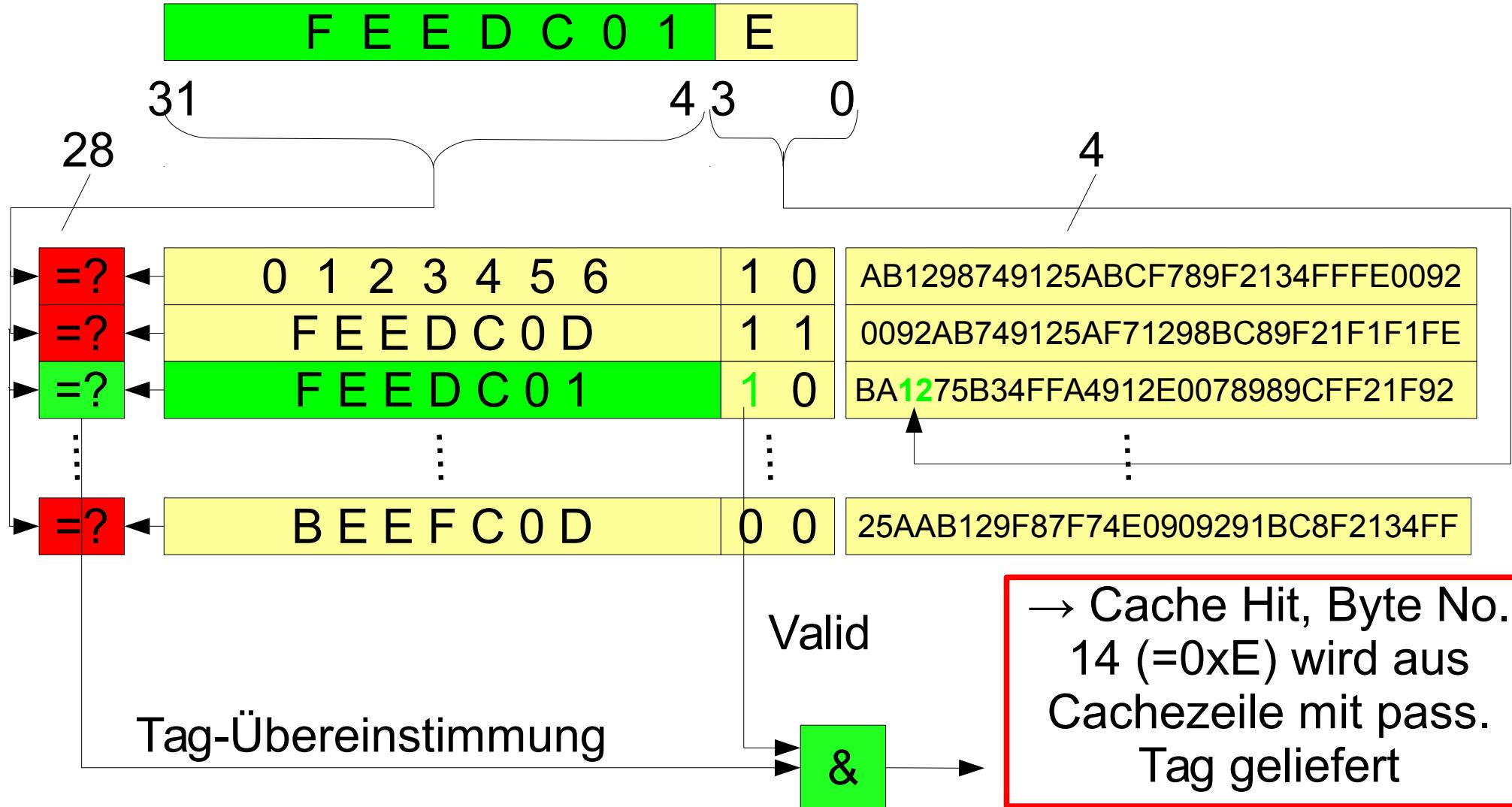
- Wenn alle Cachezeilen belegt sind und eine Kopie eines Objekts in einer Zeile neu angelegt werden soll, so muss eine bestehende Cachezeile überschrieben („verdrängt“) werden
- Nach welcher Strategie erfolgt diese Auswahl?
- Hier besprochene Möglichkeiten:
 - **Random**: Ersetzung nach dem Zufallsprinzip
 - Einfach, überraschend gutes Ergebnis
 - **FIFO** (first in first out): Die am längsten im Cache befindliche Zeile wird ersetzt
 - Immer noch einfach, aber keine gute Idee
 - **LRU** (least recently used): Die am längsten nicht mehr verwendete Zeile wird ersetzt
 - Gute Ergebnisse, aber aufwändig
 - **LFU** (least frequently used): Die am wenigsten häufig verwendete Zeile wird ersetzt
 - Aufwand und Ergebnis ähnlich LRU (d.h. gut aber aufwändig)

- Cache-Organisationsformen:
 1. Vollassoziativ: *fully associative*
 2. Direkt abbildend: *direct-mapped*
 3. Mehrfach assoziativ: *N-way set associative*
- Annahmen bei den folgenden Beispielen:
 - 32-bit Adressierung
 - 4K (4096) Byte Cache
 - Cache-Zeilengröße: 16 Byte
($\rightarrow 4096 : 16 = 256$ Cache-Einträge)

Vollassoziativer Cache: Aufbau

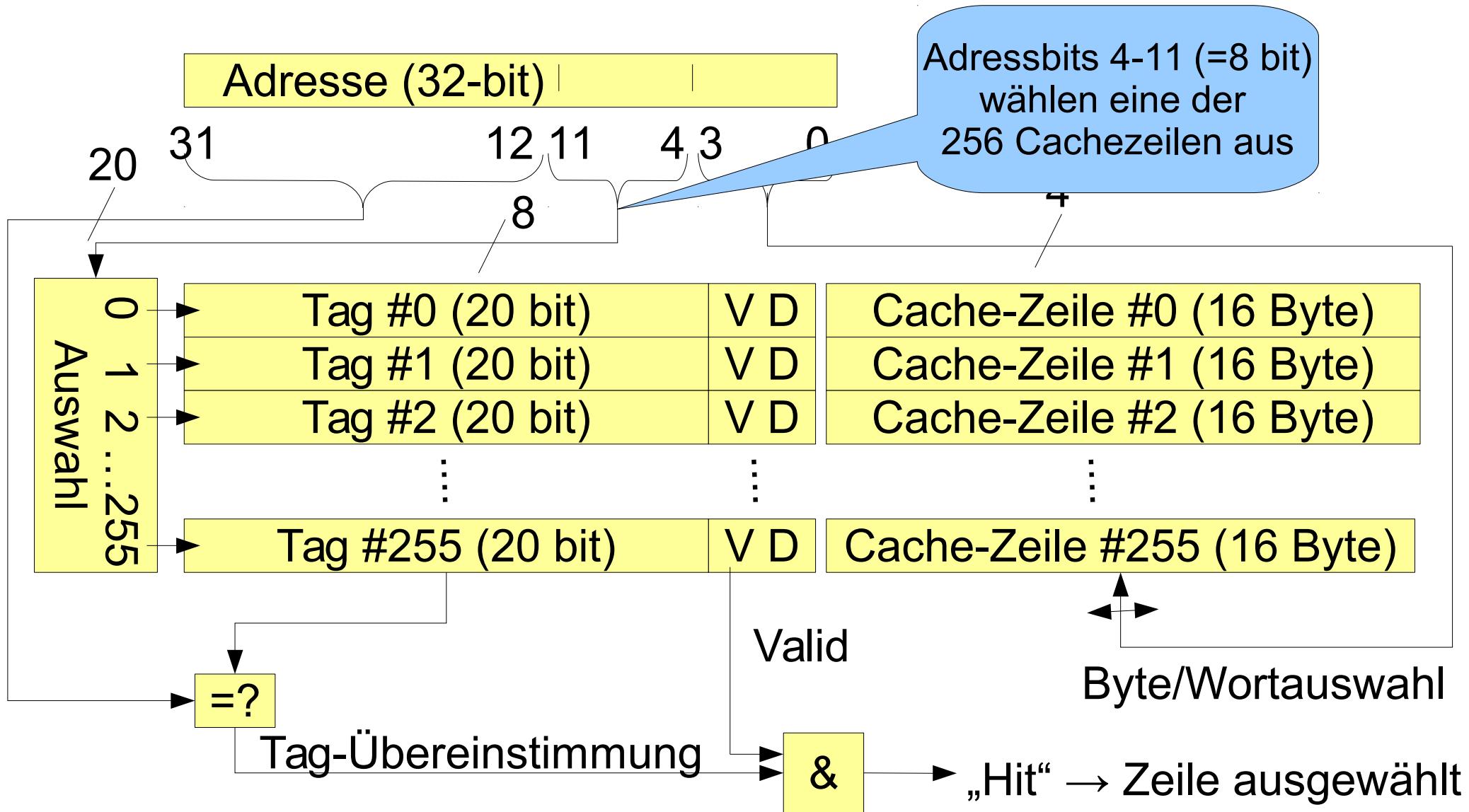


Vollassoziativer Cache: Beispiel

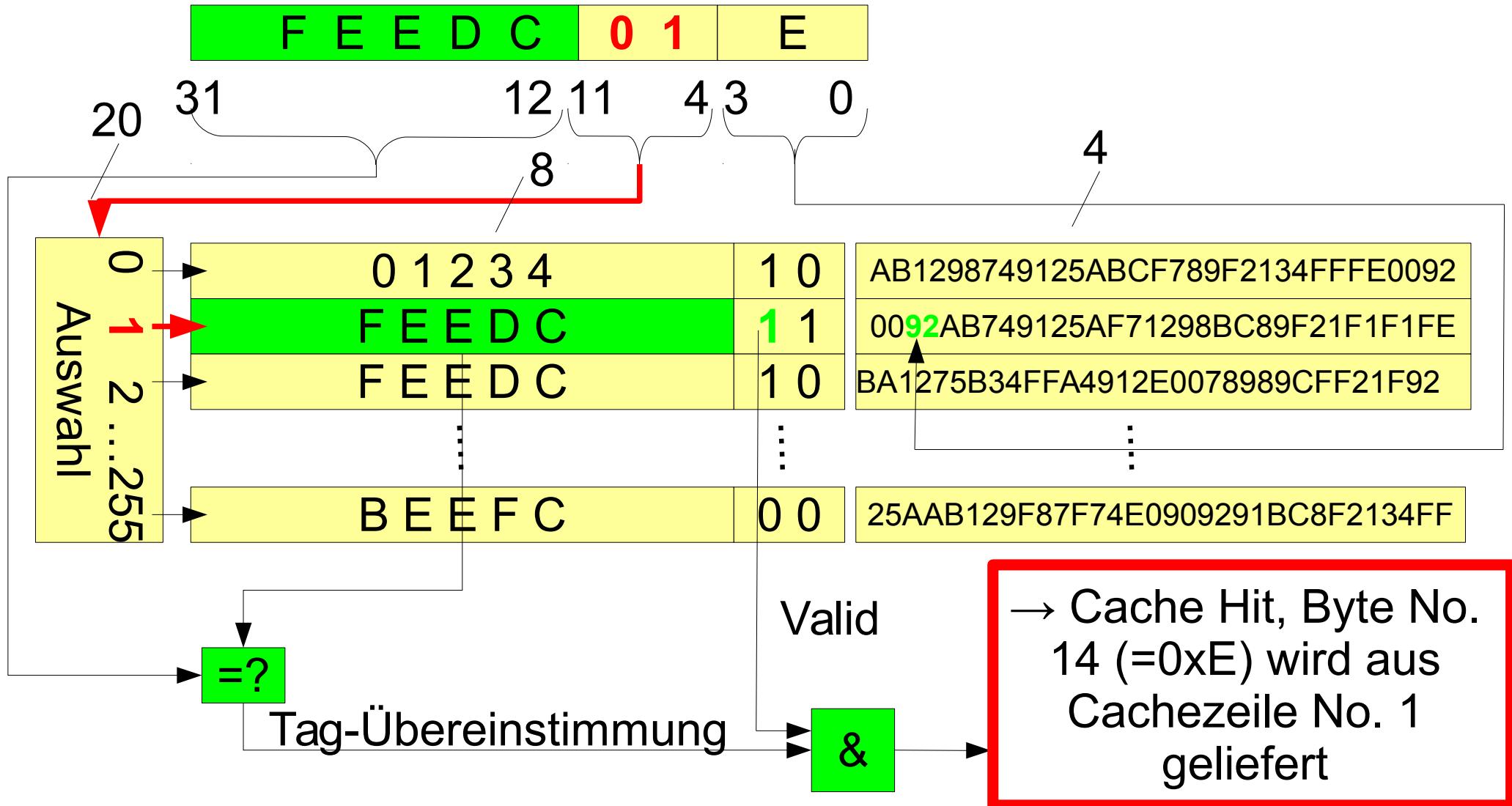


- Ein Objekt kann in eine beliebige Cache-Zeile kopiert werden
→ Freie Auswahl eines freien / zu verdrängenden Eintrags
- Identifikation der Zeile ausschließlich anhand des Tags
→ Konsequenzen:
 - Tags **aller** Zeilen müssen mit Adresse verglichen werden
 - Vergleich muss **gleichzeitig** auf allen Zeilen erfolgen
 - Für jede Zeile wird ein eigener Vergleicher benötigt
 - Jedes Tag darf maximal ein Mal vorkommen
- Erreicht höchste Trefferquote (wg. Eintrags-Wahlfreiheit)
- Große Anzahl an Vergleichern (Im Beispiel: 256 für einen 4K Cache) → sehr hoher Hardwareaufwand
- Beispiel:
 - TLB-Cache des MIPS R3000 / R4000: Vollassoziativer Cache mit 64 / 128 Einträgen

Direkt abbildender Cache: Aufbau



Direkt abbildender Cache: Beispiel

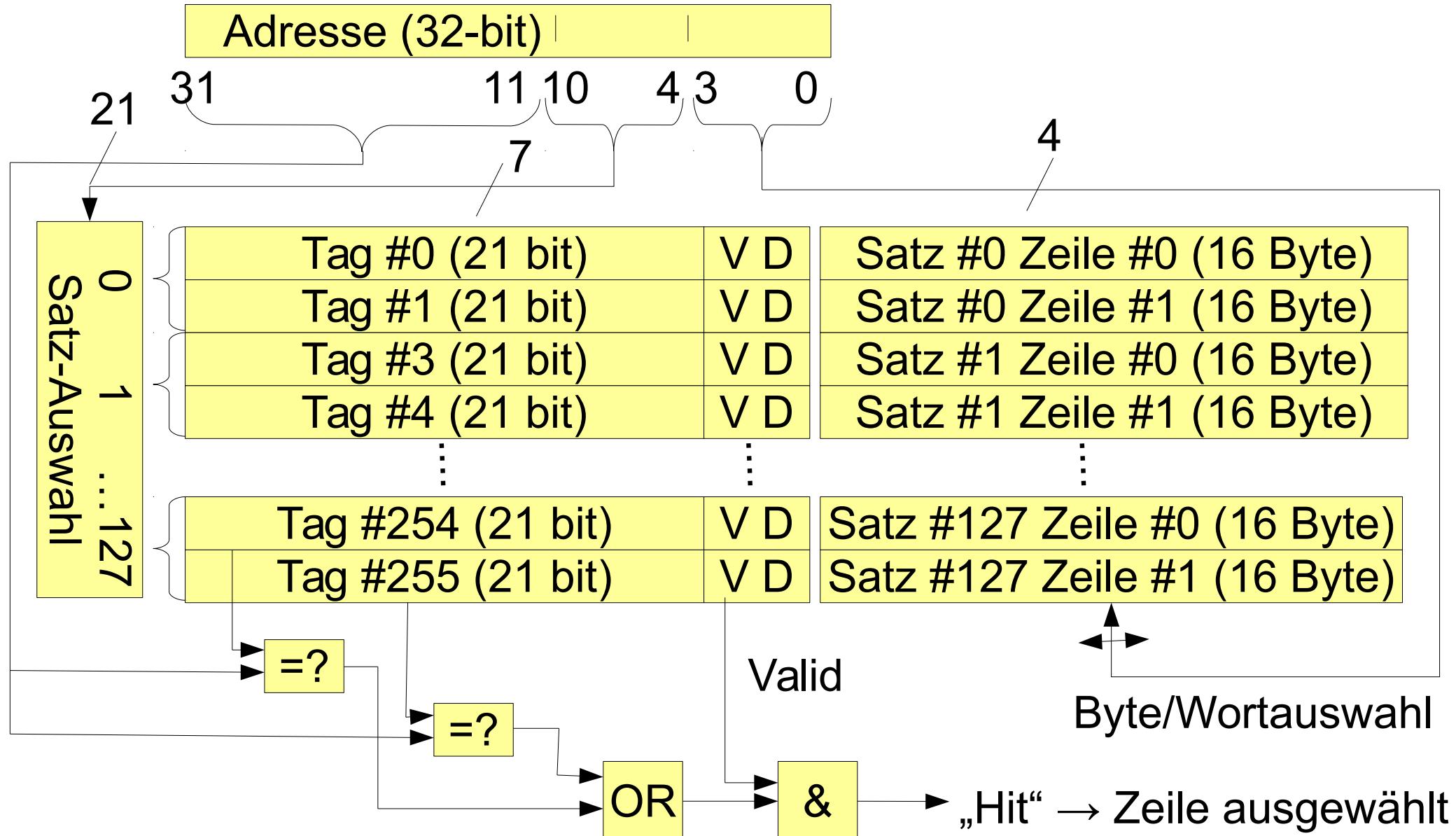


- Ein Teil der Adresse (im Beispiel: Bits 4 bis 11) wählt die Cachezeile aus
- Eindeutige Zuordnung ohne Wahlfreiheit, keine alternativen Verdrängungsstrategien möglich (aber auch keine nötig)
- Tag dient allein zur Hit/Miss Entscheidung
- **Konsequenzen:**
 - (+) Einfacher Aufbau (nur ein Vergleicher erforderlich)
 - (-) Schlechte Trefferquote
- **Beispiel:**
 - Ein Programm arbeitet in einer Schleife mit zwei Objekten, die in verschiedenen Cache-Zeilen liegen, deren Adressen sich aber in Bits 4 bis 11 nicht unterscheiden → Objekte verdrängen sich permanent gegenseitig (sog. „Cache Trashing“)

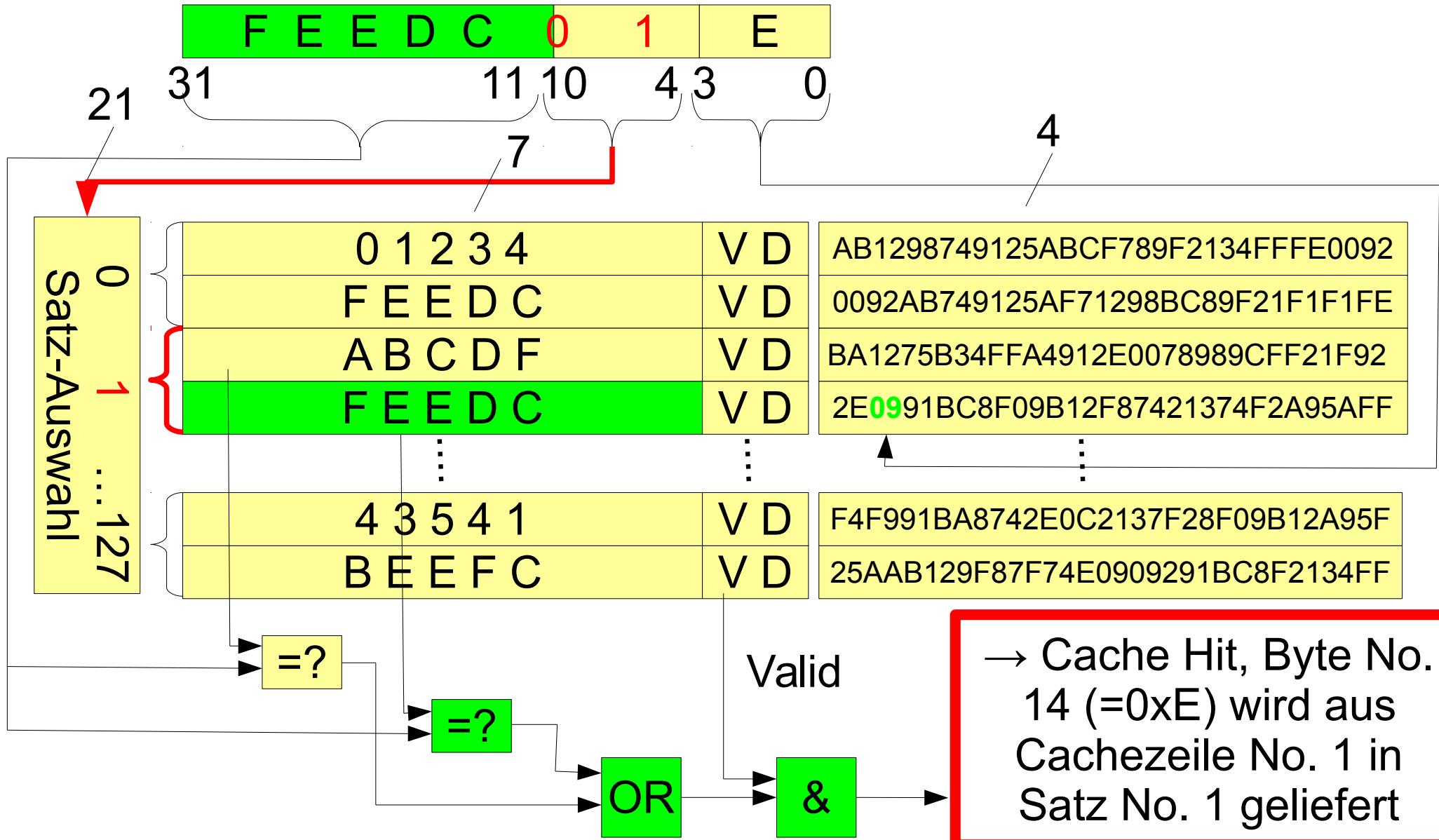
- Vollassoziativer Cache: Hohe Trefferquote, aber aufwändig
- Direkt abbildender Cache: Geringer Aufwand, aber schlechte Trefferquote (neigt zu „thrashing“)
- Kompromiss: Mehrfach assoziativer Cache^(*)
 - Zusammenfassen von je N ($N = 2, 4, 8, \dots$) Cache-Zeilen zu einem „Satz“ (engl. „set“)
 - Ein Teil der Adresse dient als Satznummer
 - Innerhalb eines Satzes gibt es N mögliche Cache-Zeilen („Wege“, engl. „ways“), die anhand ihres Tags unterschieden werden
- ➔ Für die Auswahl eines freien bzw. zu verdrängenden Cache-Eintrags stehen N Alternativen zur Verfügung
- Verdrängungsstrategien können –wenn auch eingeschränkt– umgesetzt werden

(*) engl. *N-way set associative cache*

z.B. zweifach assoziativer Cache: Aufbau



zweifach assoziativer Cache: Beispiel



- Deutliche Verbesserung der Trefferquote gegenüber direkt abbildendem Cache
- N Wege \rightarrow N Vergleicher werden benötigt
- Für $N = 1$ „degeneriert“ er zum direkt abbildenden Cache
- Für $N = <\text{Anzahl der Cache-Zeilen}>$ „degeneriert“ er zum vollassoziativen Cache
- Für Zwischenwerte von N : guter Kompromiss zwischen Aufwand und Trefferquote
- Heute der am meisten verwendete Cache
- (s.o.) Working Set üblicher Programme besteht aus $> 5\text{-}6$ Regionen
- N sollte \geq Anzahl der Regionen sein
(sonst \rightarrow Thrashing)

- Pentium 4:
 - L1 Datencache: 4-fach assoziativ (64 Byte Zeilengröße)
 - L1 Befehlscache: 8-fach assoziativ
 - L2 Cache: 8-fach assoziativ (64 Byte Zeilengröße)

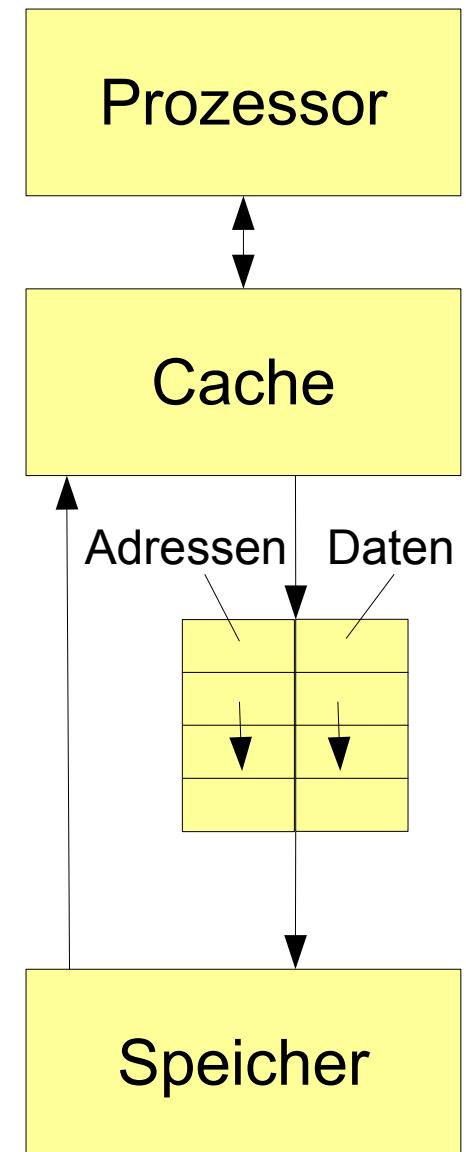


- Potenzielle Probleme im Zusammenhang mit Caches
 - „zerklüfteter“ Working Set oder ungünstige Adresslage von Variablen kann zu *Thrashing* führen → drastischer Performance-Einbruch
 - **Multitasking**: Prozesswechsel bedeutet i.d.R. auch kompletten Wechsel des Working Set
 - Nach Prozesswechsel ist der Prozessor langsamer (u.U. bis Faktor 30!)
 - DMA und Schreibzugriffe auf Codespeicher: evtl. explizites *flush* & *invalidate* erforderlich (s.o.)
 - **Multicore**: Vielfach gemeinsamer L2/L3 Cache: → gegenseitiges „ausbremsen“ der Cores, wenn auf verschiedenen Working sets gearbeitet wird.

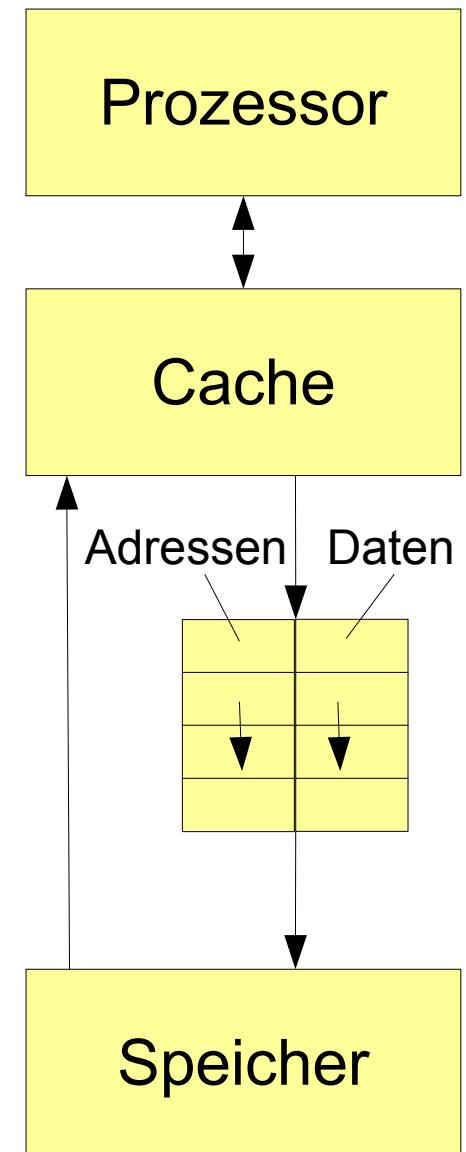
Schreib-Pufferspeicher (Write Buffer)



- Charakteristisches Verhalten von –z.B.– C-Programmen:
 - Etwa 10% „store“-Befehle, d.h. speichern von Daten
 - Solche Schreibzugriffe kommen häufig in schneller Folge („Bursts“) vor (z.B. wenn zu Beginn eines Unterprogramms Register gerettet werden)
- Insbesondere bei einem *write through* Cache muss der Prozessor hier auf den langsamen Hauptspeicher warten
- Abhilfe durch *Write Buffer*:
 - Ausstehende Schreibzugriffe (Adressen **und** zu schreibende Daten) werden in einen FIFO-Puffer zwischengespeichert
 - Prozessor kann sofort weiterarbeiten
 - Zwischgespeicherte Speicherzugriffe werden parallel dazu abgearbeitet



- Write Buffer finden sich z.B. bei ARM, PowerPC und MIPS-Prozessoren
- Potenzielles Problem: Lesezugriffe können Schreibzugriffe „überholen“
- z.B. bei Ein-/Ausgabe:
 - Gerät löst Interrupt aus, obwohl der bereits (per Schreibzugriff) abgeschaltet wurde
- Lösungswege:
 - **Software**: Puffer explizit „flushen“ (spezieller Maschinenbefehl)
 - **Hardware**: Jeder Lesezugriff wartet, bis der Puffer leer ist



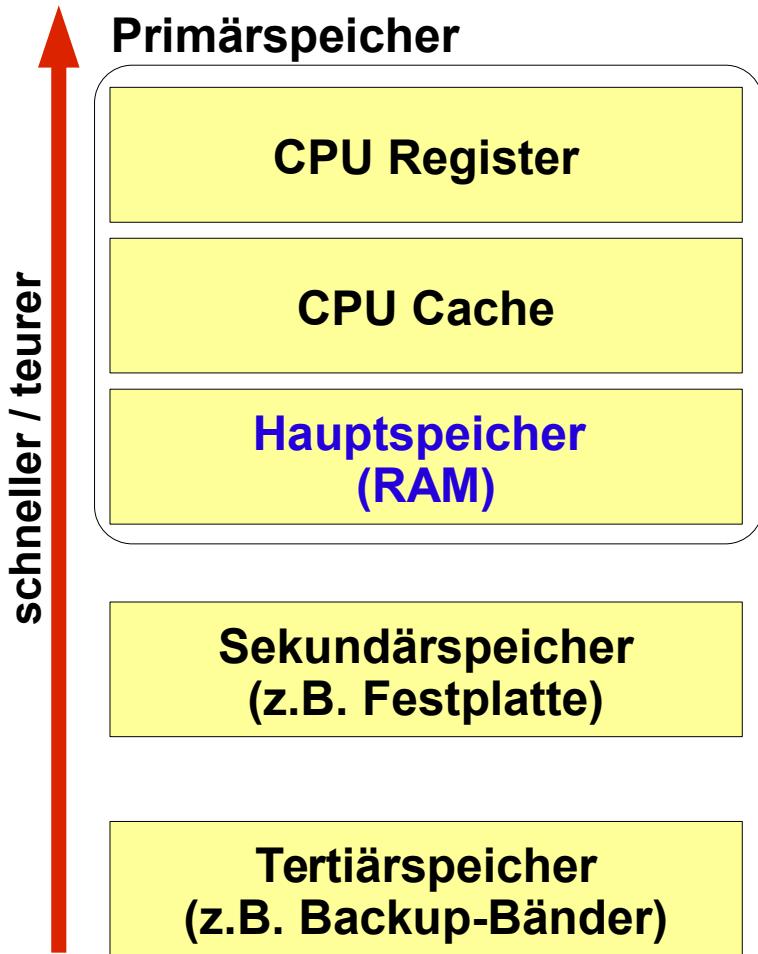


Kap. 9: **Speicherverwaltung**

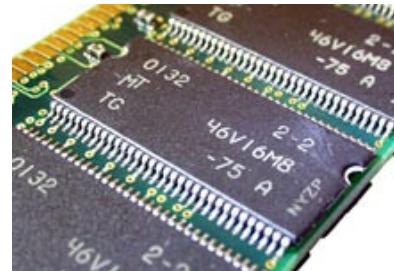
Ausgangspunkt

- Idealerweise sollte Speicher sein...
 - groß
 - schnell
 - nicht flüchtig („geht beim Ausschalten nicht verloren“)
- In der Realität (heute) nicht alle Ziele gleichzeitig zu akzeptablen Preisen mit einem Speichermedium zu erreichen
- Daher: Kombination verschiedener Speicherformen

Die Speicherhierarchie



direkter, wahlfreier Zugriff
durch den Prozessor,
sehr schnell



extern; wahlfreier
Zugriff auf den Inhalt



extern; langsam,
oft nur sequenzieller
Zugriff, hohe Kapazität



- **Schutz**
 - Prozesse sollten nicht unerlaubt auf fremde Speicherbereiche zugreifen können
- **Gemeinsame Nutzung**
 - Andererseits soll eine kontrollierte gemeinsame Nutzung von Speicherbereichen möglich sein (z.B. 10 gleichzeitige bash-Nutzer → Code nicht 10x laden)
- **Relokation**
 - Absolute Adressbezüge im Programmcode z.B. beim Laden in einen (anderen) konkreten Speicherbereich anpassen
- **Speicherorganisation**
 - Unterstützung von Programm-Modularisierung durch **Segmentierung**, abgestufter Schutz z.B. für Daten / Code
 - Ein-/Auslagern von Speicherbereichen zwischen Hauptspeicher und Sekundärspeicher („Festplatte“)

Def

Der Speicherverwalter ist der Teil des Betriebssystems, der den Arbeitsspeicher verwaltet.

Aufgaben:

- Verwaltung von freien und belegten Speicherbereichen.
- Zuweisung von Speicherbereichen an Prozesse, wenn diese sie benötigen (Allokation).
- Freigabe nach Benutzung oder bei Prozessende.
- Durchführung von Auslagerungen von Prozessen (oder Teilen) zwischen Arbeitsspeicher und Hintergrundspeicher (Platte) bei Speicherengpässen.

In diesem Kapitel wird eine Folge von einfachen bis hochentwickelten Speicherverwaltungsverfahren betrachtet.

- 9.1 Statische Speicherverwaltung
- 9.2 Swapping
- 9.3 Virtueller Speicher (vgl. RA)
- 9.4 Seiterersetzungsalgorithmen
- 9.5 Design-Probleme bei Paging-Systemen
- 9.6 Segmentierung
- 9.7 Beispiel: Speicherverwaltung in UNIX
- 9.8 Zusammenfassung

Statische Speicherverwaltung

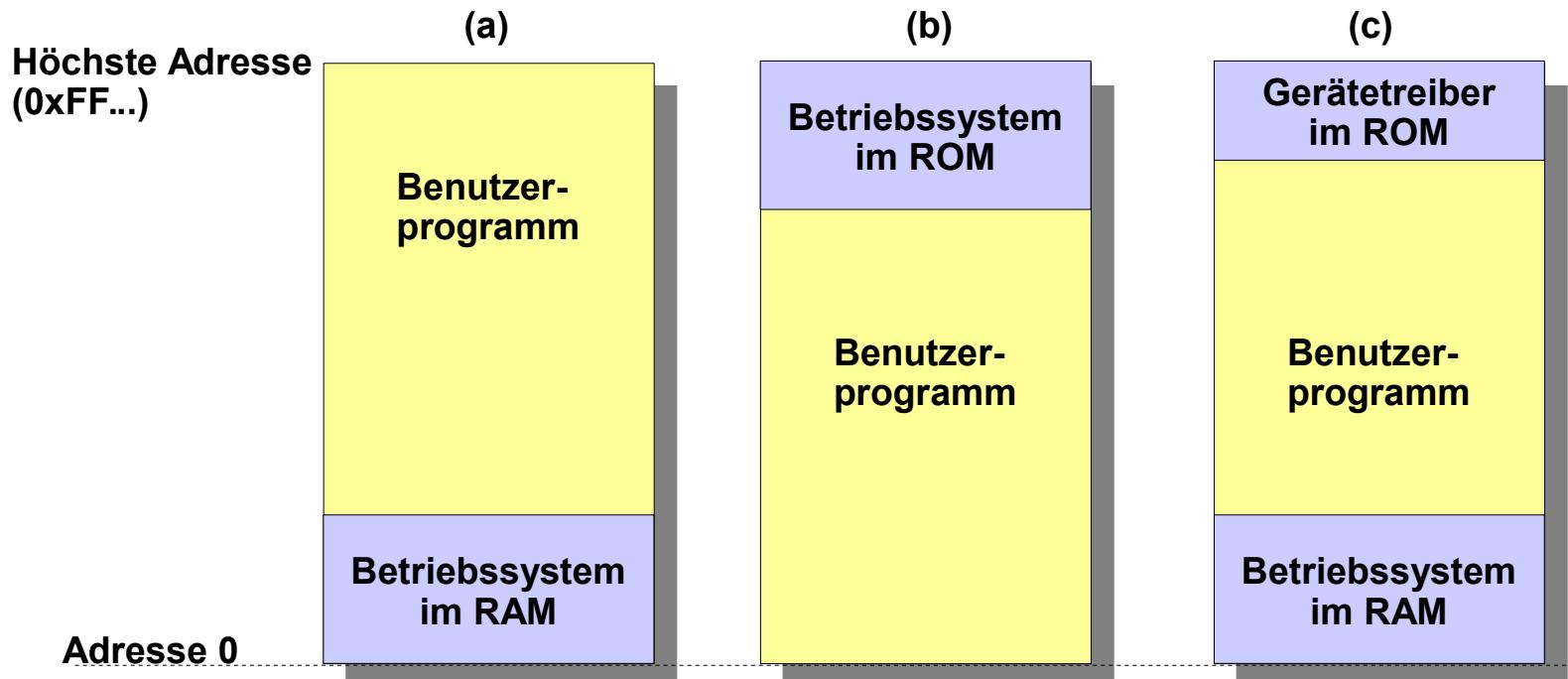
Def

- Verfahren mit fester Zuordnung von Speicher an Prozesse.
- Zuordnung ist keinen Veränderungen während der Laufzeit des Systems oder der Prozesse unterworfen.
- Nicht-statische Speicherverwaltung heißt auch dynamische Speicherverwaltung (ab Kap. 9.2).
- Im folgenden für den historischen Einprogramm- und Mehrprogrammbetrieb betrachtet.
- Heute noch in einfachen Echtzeitanwendungen üblich (z.B. Steuerung einfacher Maschinen).

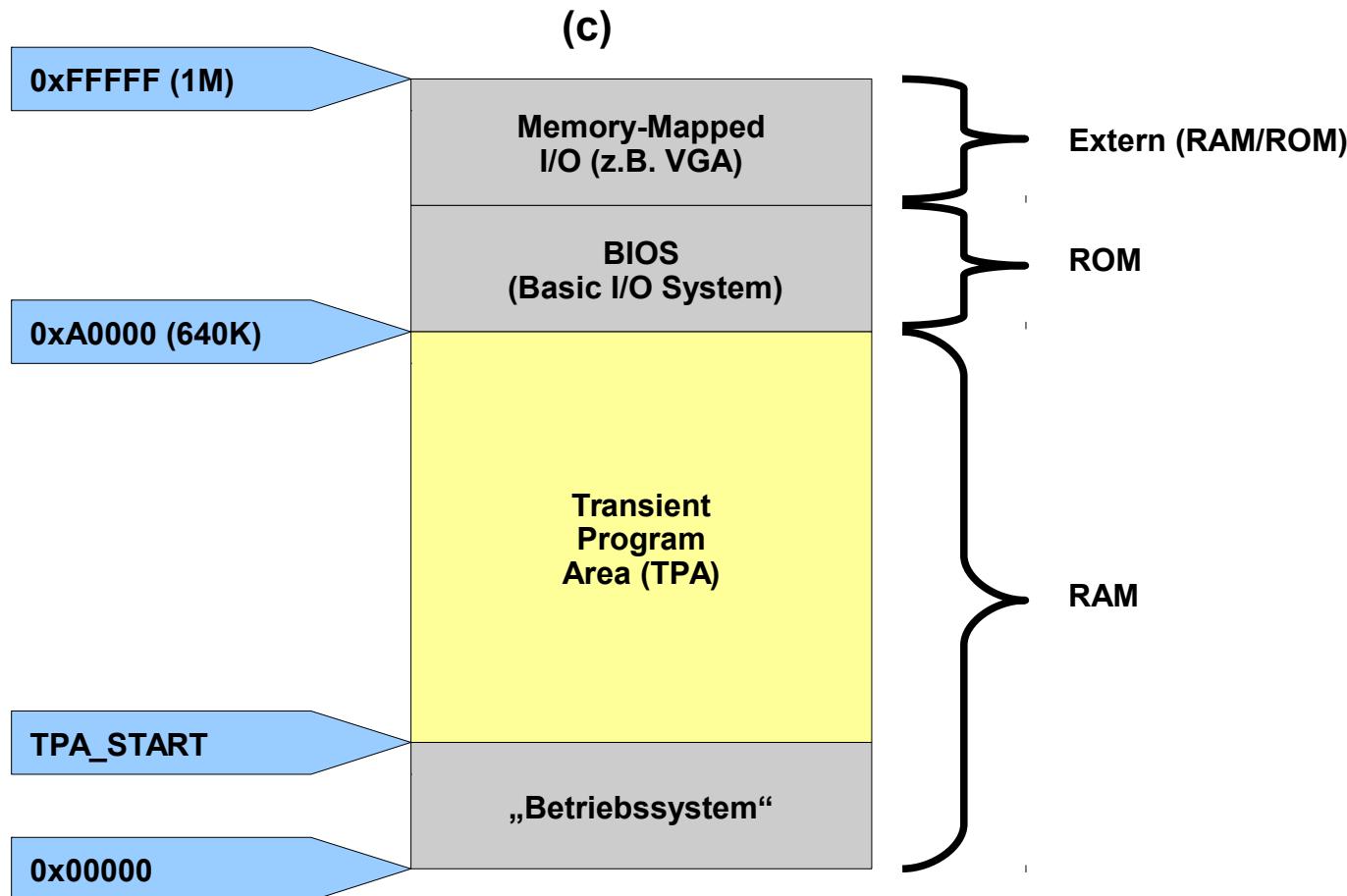
9.1.1 Einprogrammbetrieb



- Nur ein Prozess befindet sich zu einem Zeitpunkt im Speicher.
Zuweisung des gesamten Speichers an diesen Prozess.
- Einfache Mikrocomputer: Speicheraufteilung zwischen einem Prozess und dem Betriebssystem.
- Typische Alternativen:



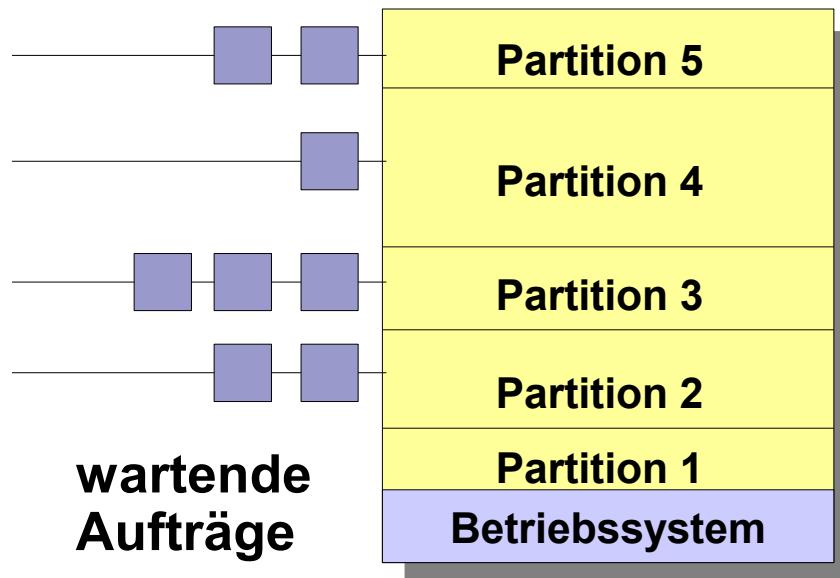
IBM PC (i8088):



9.1.2 Mehrprogrammbetrieb

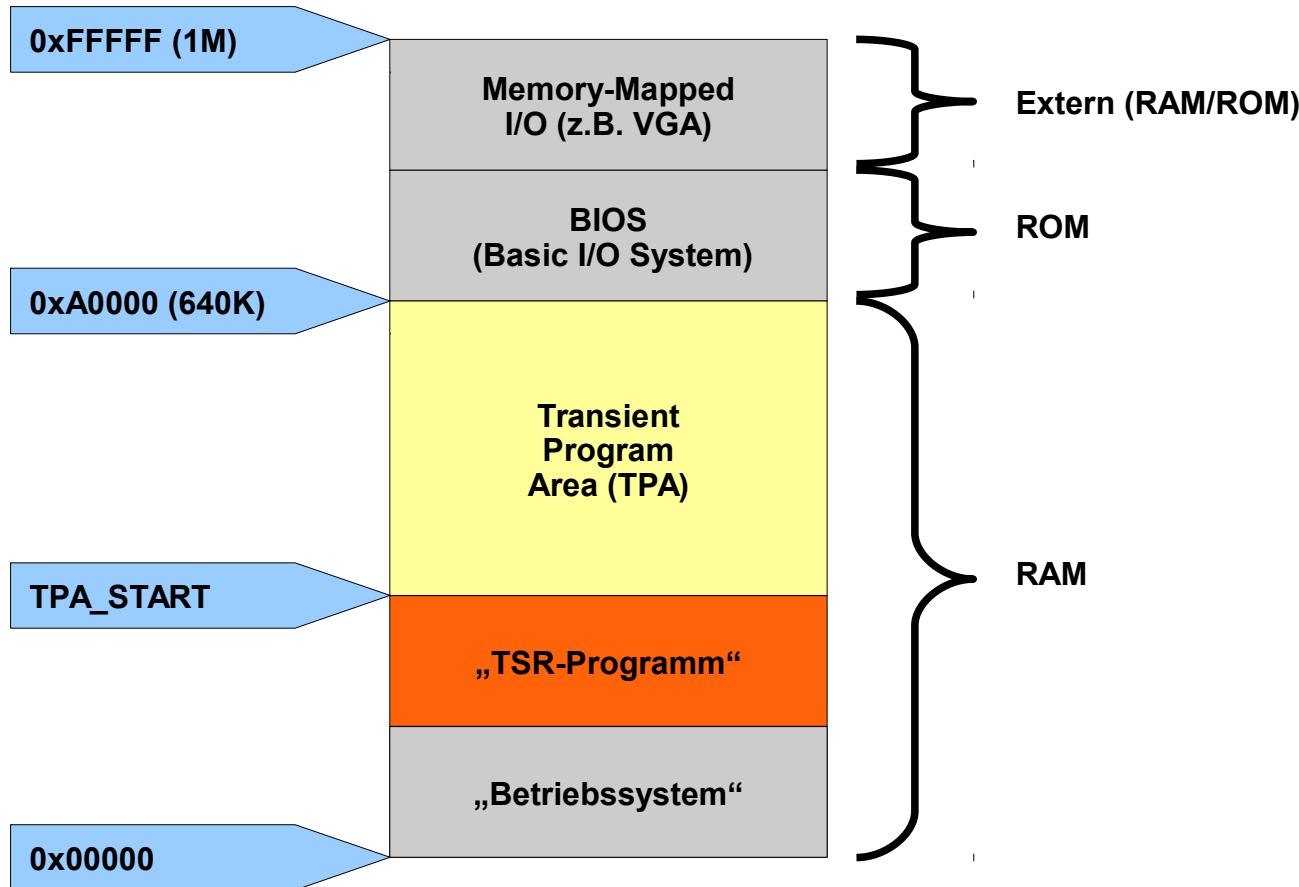


- Ziel: Bessere Ausnutzung der CPU (vgl. Kap. 1).
- Aufteilung des Speichers in feste Partitionen bei Systemstart. In jede Partition kann ein Programm geladen werden.
- Beispiel: IBM Mainframe /360 Betriebssystem OS/MFT (Multiprogramming with a Fixed number of Tasks).
- Programme können für eine Partition gebunden sein und sind dann nur in dieser lauffähig.

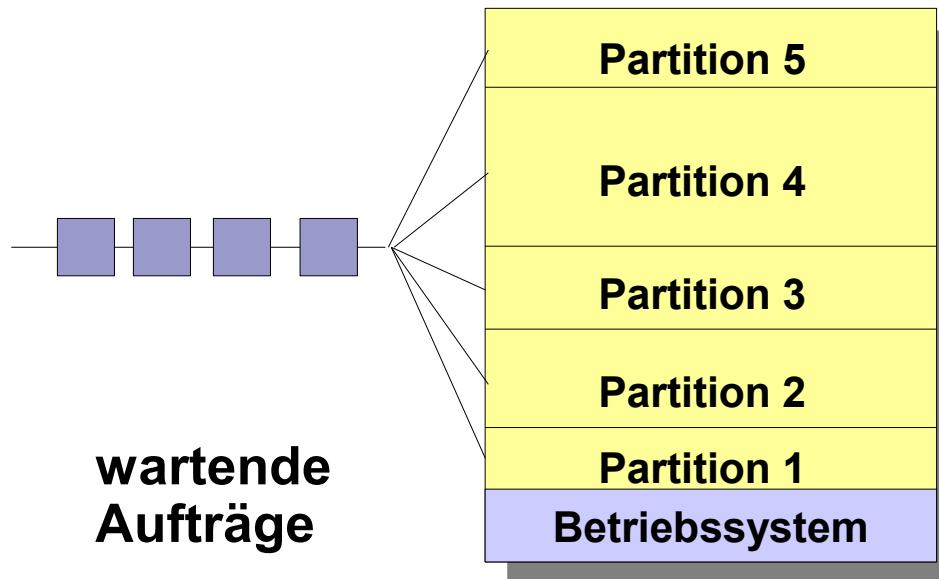


„Mehrprogrammbetrieb“ mit TSR^(*)-Programm

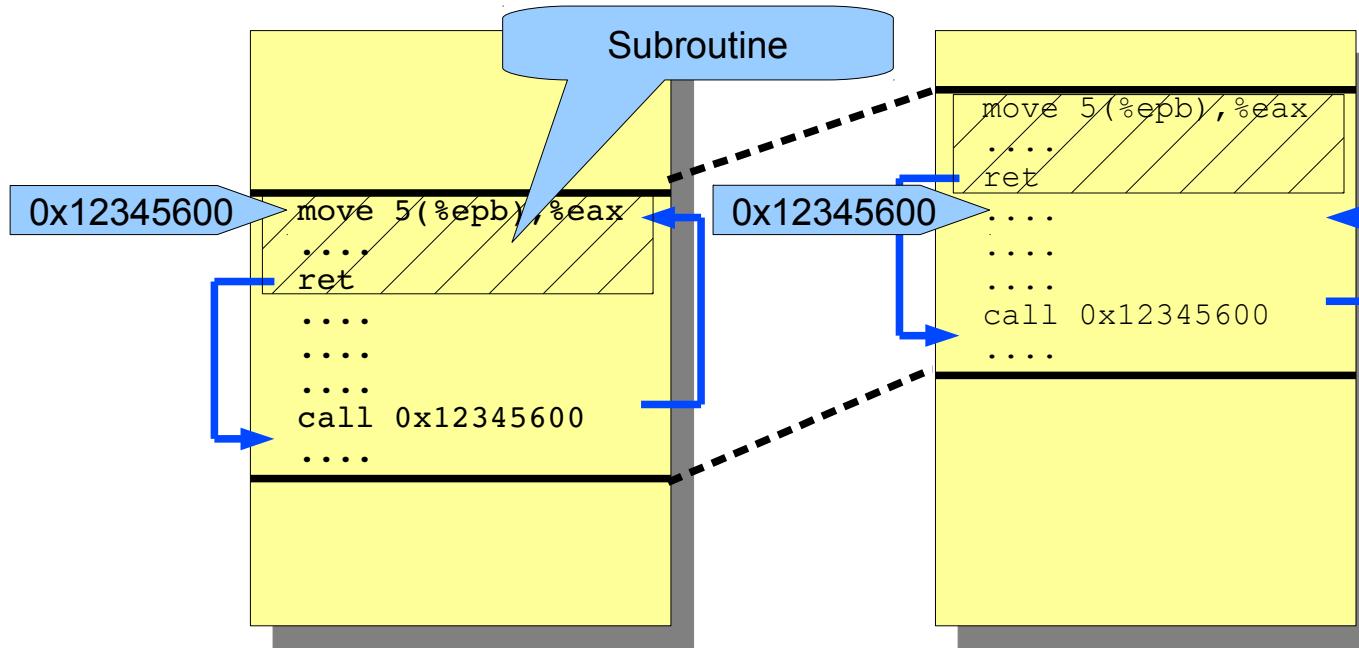
(*) terminate, stay resident



- Freie Auswahl einer Partition verlangt Lösung des sog. Relokationsproblems (Verschiebbarkeit), d.h. Anpassung aller (absoluten) Adressen des Programms in Abhängigkeit von der Ladeadresse.



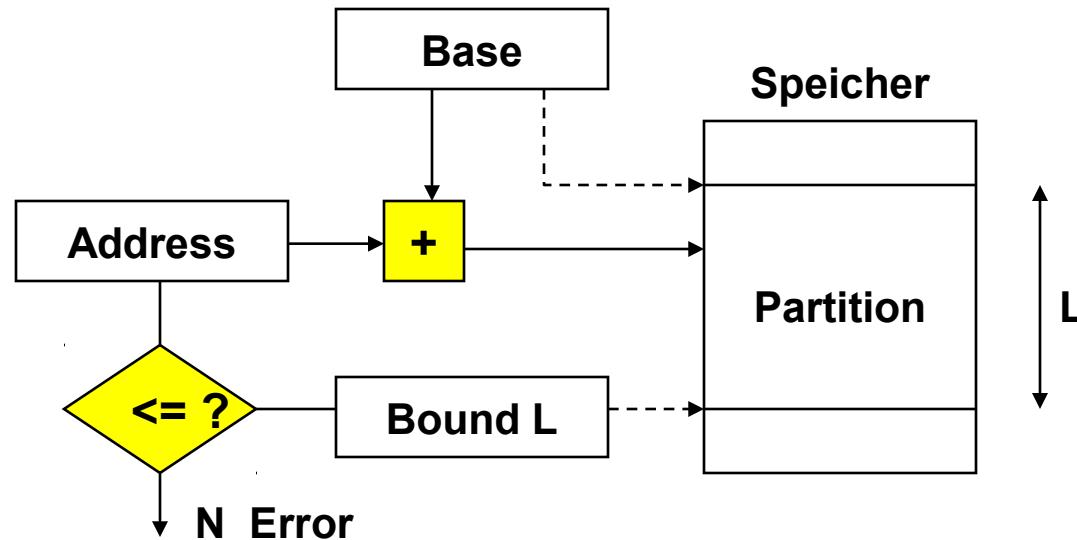
- In der Regel sind Programme nicht Positionsunabhängig
(„PIC“=position independent Code, „PID“ = position independent Data)



- Mögliche Lösung des Relocationsproblems: Loader addiert die Ladeadresse auf alle Adressen gemäß eines vom Binder erstellten Adressverzeichnisses des Programms
(Bsp.: OS/MFT).

- Mehrprogrammbetrieb führt auch zum Schutzproblem:
Programme können aufgrund der absoluten Adressierung Speicherbereiche anderer Benutzer lesen und schreiben (unerwünscht!).
 - Lösung des Schutzproblems in IBM /360:
 - Jedem 2kB-Block des Arbeitsspeichers wird ein 4-Bit Schutzcode (protection key) als Schloss zugeordnet.
 - Jeder Prozess besitzt einen ihm im Programmstatuswort (PSW) zugeordneten Schlüssel, der beim Zugriff auf einen Block passen muss, ansonsten Abbruch.
 - Nur das Betriebssystem kann Schutzcodes von Speicherblöcken und Schlüssel von Prozessen ändern.
- ⇒ Fazit: Ein Benutzerprozess kann weder andere Benutzerprozesse noch das Betriebssystem stören.

- Alternative Vorgehensweise zur Lösung des Schutzproblems wie des Relokationsproblems:
Ausstattung der CPU mit zwei zusätzlichen Registern, die Basisregister und Grenzregister genannt werden
(base and bound register).



- Weiterer Vorteil eines Basisregisters zur Relokation: Verschieblichkeit des Programms nach Programmstart wird möglich.

Einführung

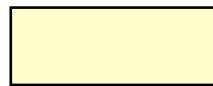
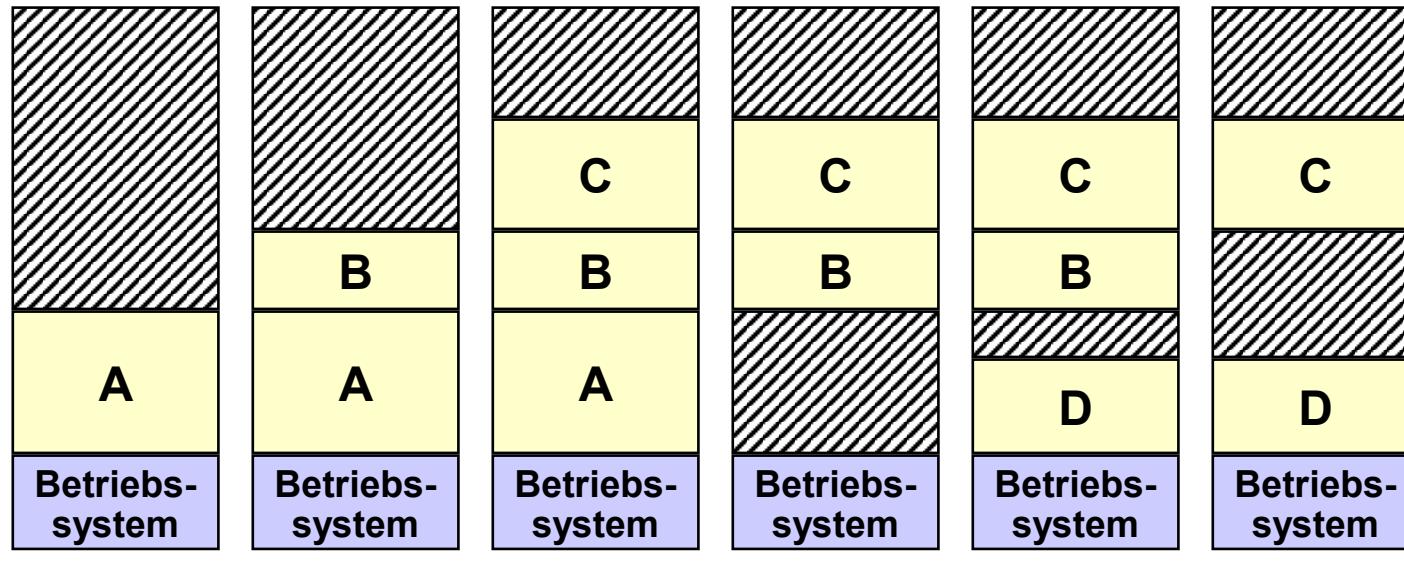
- Bei klassischen Timesharing-Systemen gibt es normalerweise nicht genügend Hauptspeicher, um Prozesse aller Benutzer aufzunehmen.
- Swapping beinhaltet das Verschieben von Prozessen vom Hauptspeicher auf Platte (Auslagern) und umgekehrt (Einlagern).
- Im Hauptspeicher zu jedem Zeitpunkt nur eine Teilmenge der Prozesse, diese sind aber vollständig repräsentiert (im Gegensatz zu virtuellem Speicher, vgl. 9.3).
- Die restlichen (möglicherweise auch rechenwilligen) Prozesse werden in einem Bereich im Hintergrundspeicher abgelegt. Dieser Bereich heißt Swap-Bereich (Swap Area).
- Beispiel: Im Unix vor Einführung der virtuellen Speicherverwaltung genutzt.

9.2.1 Variable Partitionen

- Mehrprogrammbetrieb kann in variablen Partitionen erfolgen, d.h.: Anzahl, Anfangsadresse und Länge der Partitionen und damit der eingelagerten Prozesse ändern sich dynamisch.
- Ein zusammenhängender Speicherbereich variabler Länge heißt auch Speichersegment oder einfach Segment.
- Ziele variabler Partitionen:
 - Anpassung an die tatsächlichen Speicheranforderungen
 - Vermeidung von Verschwendungen.

Def

Zeit →

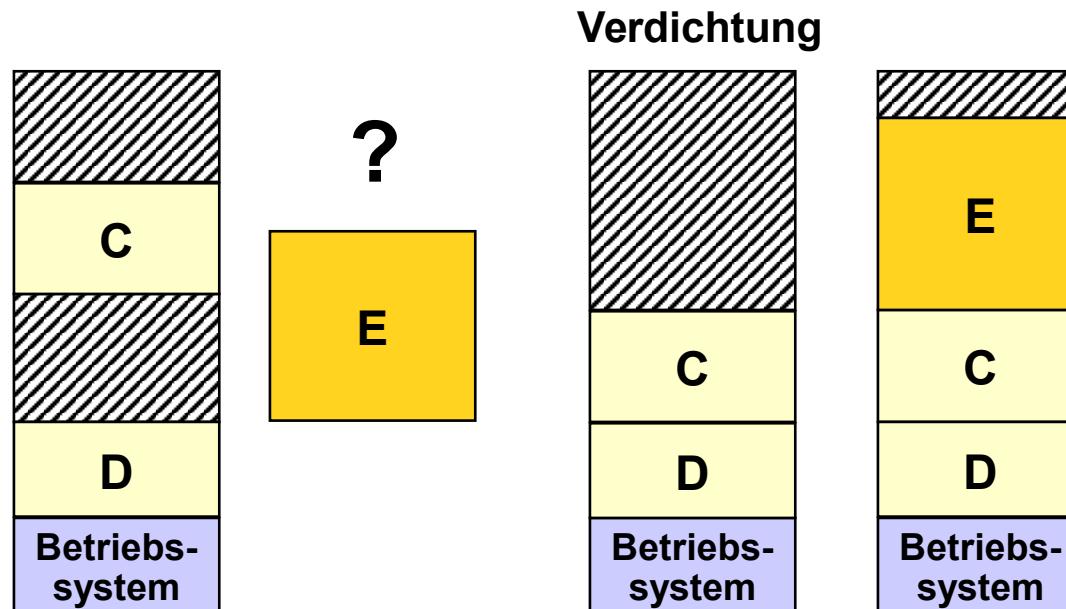


Partition



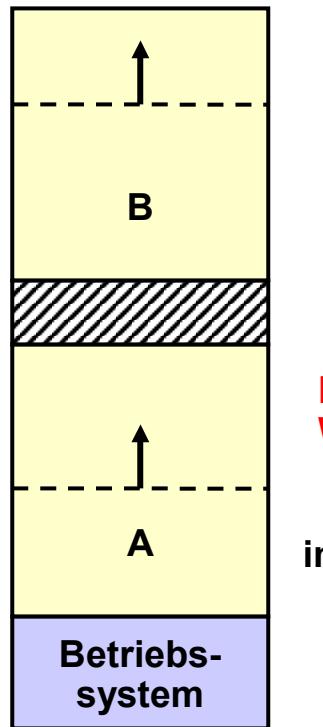
Freispeicher

- Zerstückelung von Freispeicher zwischen den belegten Segmenten wird externe Fragmentierung genannt.
- **Def** • Abhilfe: Alle unbelegten Speicherbereiche können durch Verschiebung der belegten Segmente zu einem einzigen Bereich zusammengefasst werden. (⇒ Speicherverdichtung oder Kompaktifizierung).
- Beispiel:



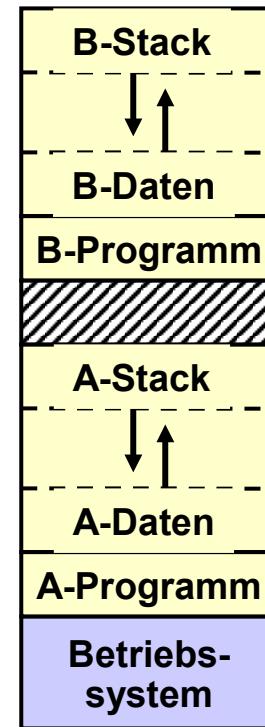
- Speicherbedarf eines Prozesses kann sich im Verlaufe ändern (i.d.R. wachsen).
- Prozess stellt dynamische Speicheranforderungen, z.B. zur Aufnahme von neu erzeugten Objekten auf der Halde (dem Heap).
- Behandlung dynamischer Speicheranforderungen:
 - Belegen angrenzender „Löcher“, falls vorhanden.
 - Verschiebung von anderen Prozessen, falls möglich.
 - Auslagern von anderen Prozessen.
 - „Raum zum Wachsen“: Bei der ersten Speicherbelegung vorab zusätzlichen Speicherplatz alloziiieren.

- Beispiele für "Raum zum Wachsen":



(a)
Wachsener Datenbereich
eines Prozesses

Raum zum
Wachsen
in Benutzung

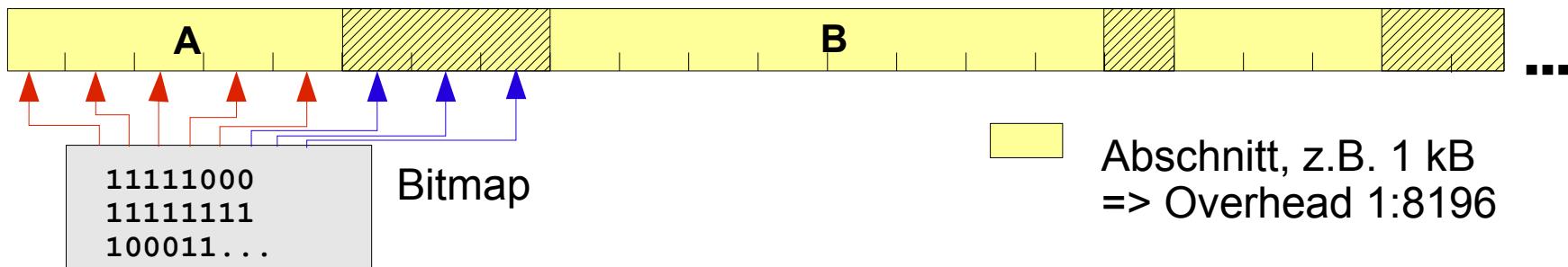


(b)
Wachsener Datenbereich
und wachsender Stack

Raum zum
Wachsen

- Speicherverwaltung mit Bitmaps (9.2.2).
- Speicherverwaltung mit verketteten Listen (9.2.3).
- Speicherverwaltung mit dem Buddy-System (9.2.4).

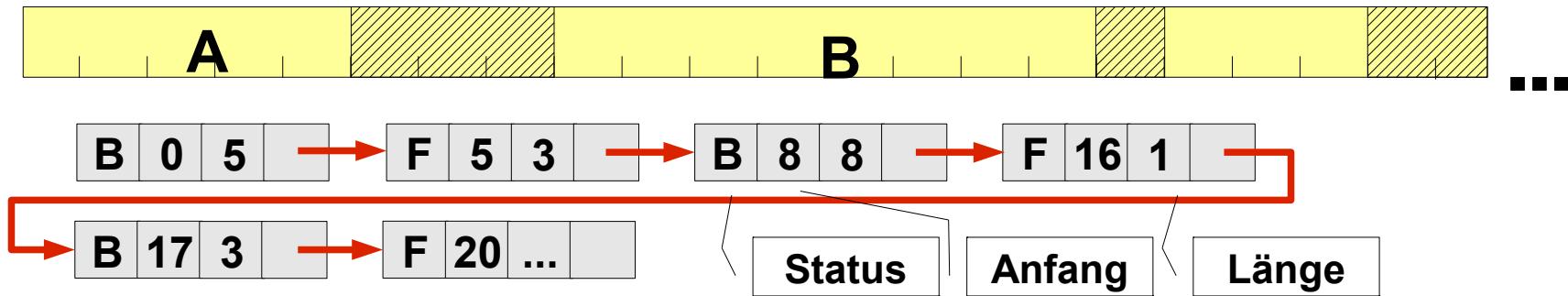
9.2.2 Speicherverw. mit Bitmaps



Arbeitsweise:

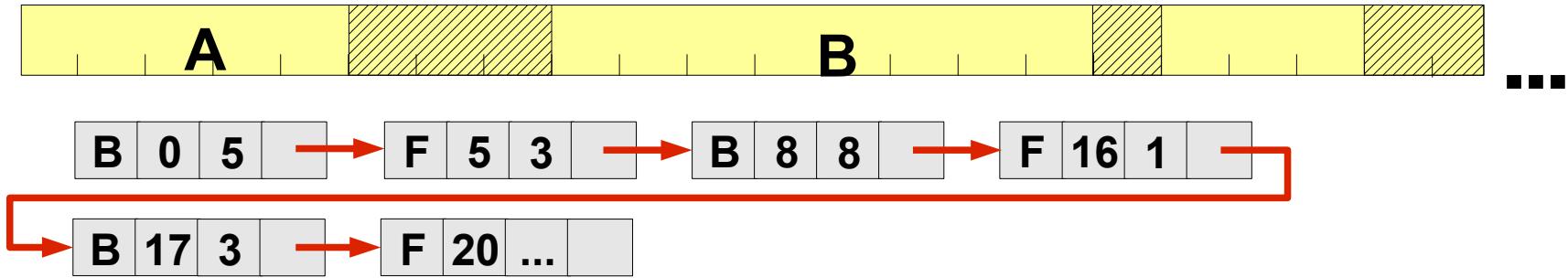
- Der zur Verfügung stehende Speicher wird in Einheiten fester Länge unterteilt (Granularität einige Worte bis einige KB).
- Jeder Speichereinheit wird ein Bit in einer Bitmap zugeordnet: $0 \simeq$ Einheit ist frei, $1 \simeq$ Einheit ist belegt.
- Je kleiner die Einheit, je größer die Bitmap.
- Je größer die Einheit, je mehr Speicher wird verschwendet, da die letzte „angebrochene“ Einheit voll alloziert werden muss.
- Hauptproblem: Belegen eines Speicherbereichs von k Einheiten erfordert Durchsuchen der Bitmap nach einer Folge von k Null-Bits. Aufwändig!
- Für Hauptspeicherverwaltung daher selten eingesetzt.

9.2.3 Verkettete Listen



- Jedem belegten und jedem freien Speicherbereich wird ein Listenelement zugeordnet.
- Segmente dürfen variabel lang sein.
- Jedes Listenelement enthält Startadresse und Länge des Segments, sowie den Status (B=belegt, F=frei).
- Gefundenes freies Segment wird (falls zu groß) aufgespalten.
- Freigegebenes Segment wird ggf. mit ebenfalls freien Nachbarsegmenten „verschmolzen“.

Verkettete Listen (2)



- Die freien Segmente können auch in separater Liste geführt werden („Freiliste“). Die Freiliste kann in sich selbst gehalten werden, d.h. in den verwalteten freien Bereichen.
(\Rightarrow kein zusätzlicher Speicher notwendig).
- Die Segmentliste kann nach Anfangsadressen geordnet sein. Vorteil: freiwerdendes Segment kann mit benachbartem freien Bereich zu *einem* freien Segment „verschmolzen“ werden.
- Freiliste kann alternativ nach der Größe des freien Bereichs geordnet sein. Vorteil: Vereinfachung beim Suchen nach einem freien Bereich bestimmter Länge.

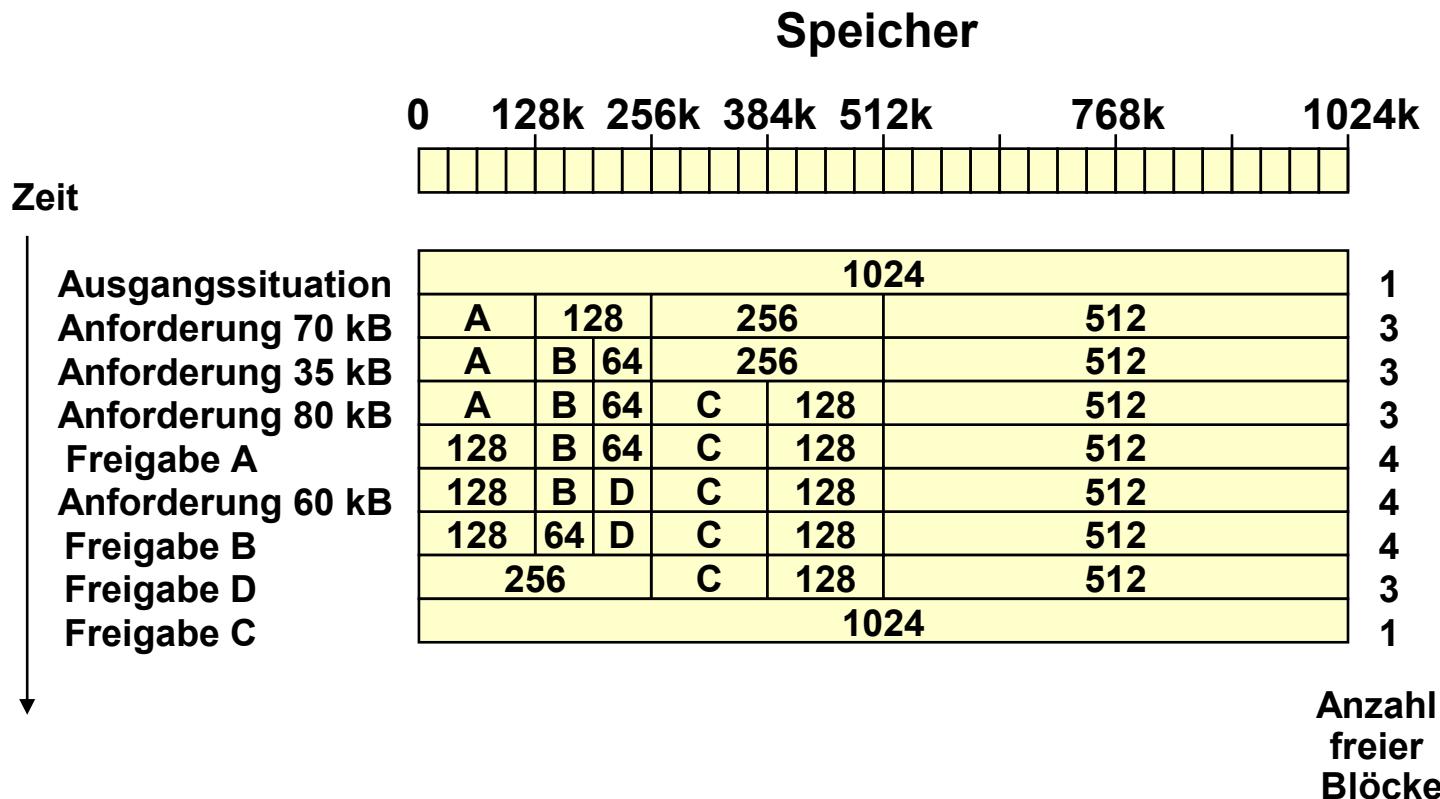
- Voraussetzungen:
 - Liste der Segmente nach Anfangsadressen geordnet.
 - Geforderte Speichergröße bekannt.
- **Def** **First Fit**: Durchsuche Freiliste, bis ein freies Segment hinreichender Größe gefunden ist. Zerlege den freien Bereich in ein Segment der geforderten Größe und ein neues freies Segment für den übrig bleibenden Rest. Kurze Suchzeit.
- **Rotating First Fit** oder **Next Fit**: Variante von First Fit. Anstelle von vorn zu suchen, beginne Suche an der nachfolgenden Stelle, an der bei letzten Durchlauf ein Segment belegt wurde.
- **Best Fit**: Durchsuche die gesamte Liste, wähle das kleinste für die Anforderung gerade ausreichende freie Segment und spalte dieses. Langsam. Neigt zur ungewollten Erzeugung vieler kleiner Freisegmente (Fragmentierung).
- **Worst Fit**: Wähle das größte freie Segment und spalte dieses. Vermeidet vieler kleiner Freisegmente, aber keine gute Idee.

Arbeitsweise:

- Freie und belegte Speicherbereiche haben ausschließlich Längen, die 2-er Potenzen sind.
- Da nur bestimmte Längen auftreten, wird von Speicherblöcken (anstelle von Segmenten) gesprochen.
- Die maximale Blocklänge L_{\max} entspricht der größten 2-er-Potenz kleiner gleich dem verfügbaren Speicher, z.B. $L_{\max} = 2^{22}$ bei 4 MB oder 6 MB.
- Es kann eine minimale Blocklänge größer $2^0 = 1$ Byte geben, z.B. $L_{\min} = 2^6 = 64$ Bytes.
- Eine Speicheranforderung wird auf die nächst mögliche 2-er-Potenz aufgerundet, z.B.: angefordert 70 KB, belegt 128 KB.
- Für jede der verwalteten Blocklängen zwischen L_{\min} und L_{\max} wird eine separate Freiliste gehalten.

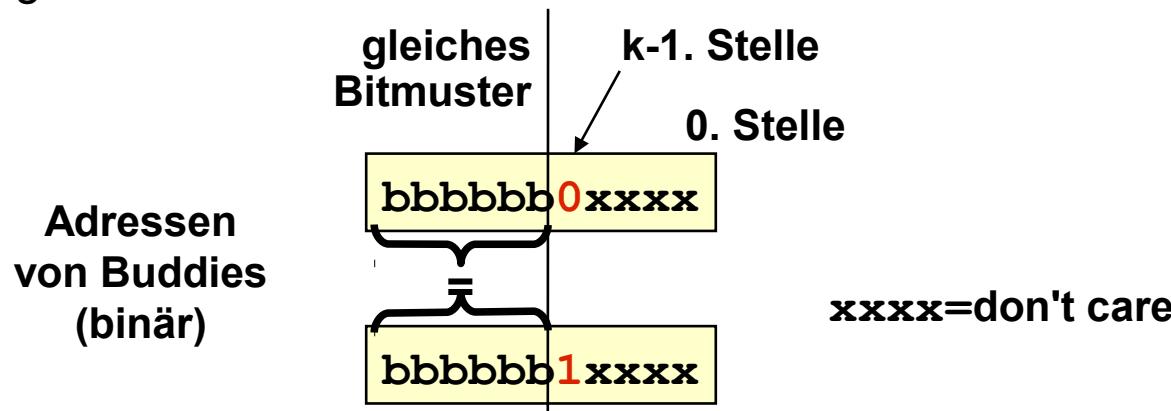
Arbeitsweise am Beispiel

9.2.4



- Kein freier Block der gesuchten Länge: Halbiere einen Block der nächsthöheren 2-er-Potenz in 2 freie Blöcke der gesuchten Länge. Diese werden Buddies genannt (deutsch: Kumpel).
- Freigabe eines Blocks der Länge 2^k : wenn sein Buddy bereits ein freier Block ist, mit diesem zu *einem* neuen freien Block der Länge 2^{k+1} vereinen.
- Es lassen sich **nicht** zwei beliebige benachbarte Blöcke gleicher Länge 2^k zusammenfassen!

Die Entscheidung ist einfach aufgrund der Adresse der freien Blöcke möglich:



Vorteile:

- Bei Speicheranforderung insgesamt schnelles Auffinden oder Erzeugen eines freien Blocks.
- Bei Freigabe muss nur *eine* Freiliste durchsucht werden. Falls Buddy bereits frei ist, einfaches Verschmelzen zu einem größeren Block.

Nachteile:

- Ineffizient in der Speicherausnutzung, da immer auf die nächste 2-er-Potenz aufgerundet werden muss.
- Diese Verschwendungen wird als interne Fragmentierung bezeichnet, da sie innerhalb des allozierten Speicherbereichs auftritt (im Gegensatz zur externen Fragmentierung).

Ergebnisse:

- Speicherverwaltung mit Bitmaps oder verketteten Listen führt zu externer Fragmentierung.

Def

- 50%-Regel (Knuth): Sind im Mittel n Prozesse im Speicher, so gibt es im Mittel $n/2$ freie Bereiche. Ursache ist, dass zwei benachbarte Freibereiche zu einem zusammengefasst werden.

Def

- Ungenutzte-Speicher-Regel: Sei s die mittlere Größe eines Prozesses, und k^*s sei die mittlere Größe eines Freibereichs für einen Faktor k , der abhängig vom Algorithmus ist. Dann gilt für den ungenutzten Anteil f des Speichers:

$$f = k / (k+2).$$

Beispiel: Sei $k = 0.5$. Dann werden 20% des Speicherplatzes für Freibereiche verbraucht.

Swapping:

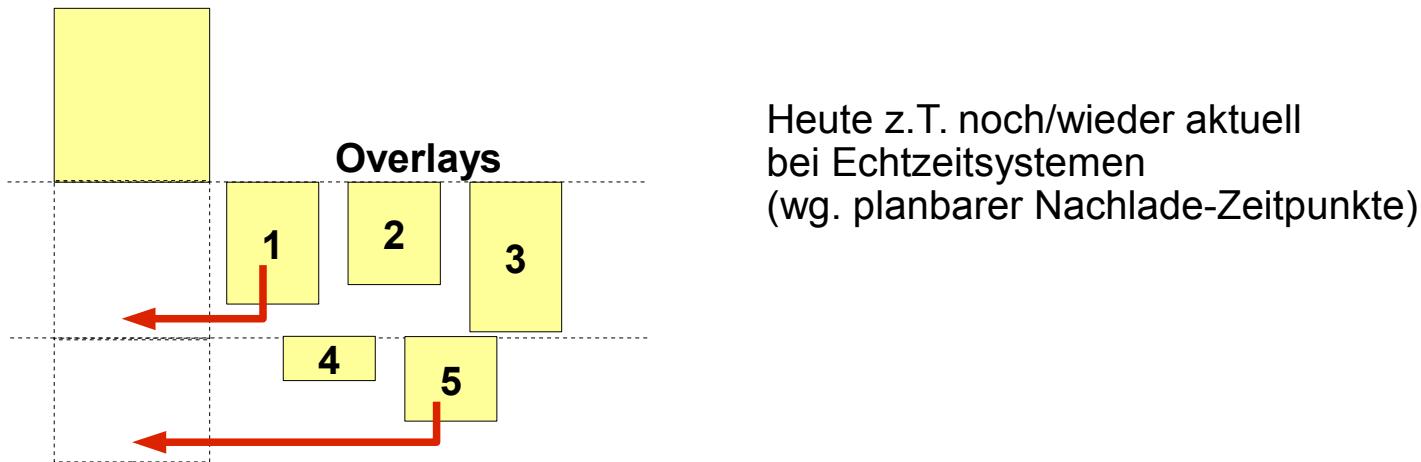
- Im Hauptspeicher wird zu jedem Zeitpunkt nur eine Teilmenge der Prozesse gehalten. Die restlichen Prozesse sind in einen Swap-Bereich auf Platte ausgelagert.
- Alle Prozesse im Hauptspeicher sind *vollständig* repräsentiert.
- Es wurden Verfahren zur Speicherverwaltung für derartige Systeme besprochen:
 - Verwaltung mittels Bitmaps
 - Verwaltung mittels verketteter Listen
 - Buddy-System
- Es wurden Strategien zur Speicherverwaltung besprochen:
 - First Fit
 - Rotating First Fit
 - Best Fit

Problem:

- Die Größe eines einzelnen Prozesses kann die Größe des insgesamt zur Verfügung stehenden Speichers übersteigen.
⇒ Swapping ist keine Lösung mehr.

Historische Lösung:

- Lösung der 60er Jahre arbeitet mit Overlays (Überlagerungen):
 - Overlay-Struktur des Programms wird vom Programmierer vorgeplant und vom Binder erzeugt.
 - Das Betriebssystem muss bei Bedarf Overlays dynamisch vom Hintergrundspeicher nachladen:



Heutige Lösung:

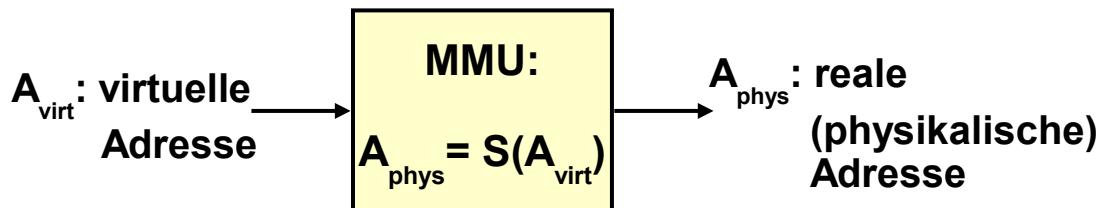
Def

- Betriebssystem hält „gerade in Benutzung“ befindlichen Teile eines Programms im Hauptspeicher, Rest auf Hintergrundspeicher.
- Dieser Ansatz wird virtueller Speicher genannt.
- Die von den Instruktionen eines Programms generierten Adressen werden virtuelle Adressen genannt. Sie bilden zusammen den (virtuellen) Adressraum des Prozesses, der das Programm ausführt.
- Speicherverwaltungseinheit (Memory Management Unit, MMU) wandelt virtuelle Adressen in reale (physikalische) Adressen um.
- Virtueller Speicher und Mehrprogrammbetrieb passen gut zueinander: Werden Teile eines blockierten Prozesses eingelagert, kann der Scheduler den Prozessor einem anderen (rechenwilligen) Prozess zuordnen.
- Paging: eindimensionaler virtueller Speicher. Heute am stärksten verbreitet.
- Segmentierung: zweidimensionaler virtueller Speicher.

- Grundbegriffe des Paging
 - Seiten (Pages)
 - Kachel oder Seitenrahmen (Page Frames)
 - Seitentabelle (Page Tables)
 - Seitenfehler (Page Fault)
- Prinzipielle Arbeitsweise einer MMU
- Realisierungsgesichtspunkte
 - Mehrstufige Seitentabellen
 - Hardware-Unterstützung mittels Assoziativspeicher
- Arbeitsweise einer MMU und verschiedene Formen der Übersetzung virtueller Adressen in reale unter Nutzung von mehrstufigen Seitentabellen und TLBs zum Caching von Übersetzungen wurden im Rechnerarchitekturteil der Vorlesung besprochen.

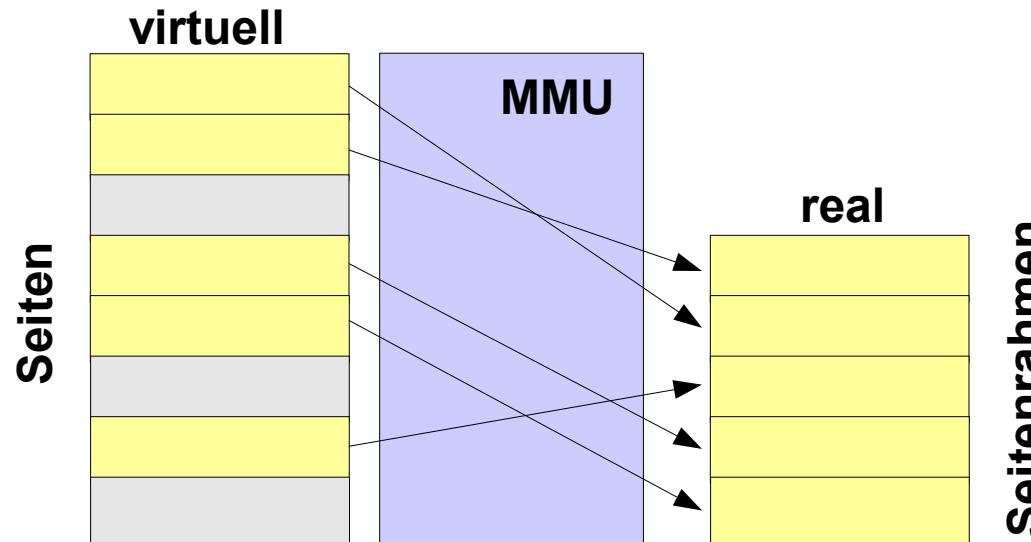
Grundbegriffe des Paging (1)

- Die von den Instruktionen eines Programms generierten Adressen werden virtuelle Adressen genannt. Sie bilden zusammen den (virtuellen) Adressraum des Prozesses, der das Programm ausführt.
- Speicherverwaltungseinheit (Memory Management Unit, MMU):
 - wandelt virtuelle Adressen in reale (physikalische) Adressen um
 - reale Adressen werden dann für den Zugriff zum Arbeitsspeicher benutzt
 - Abbildung wird abstrakt auch als Speicherfunktion bezeichnet

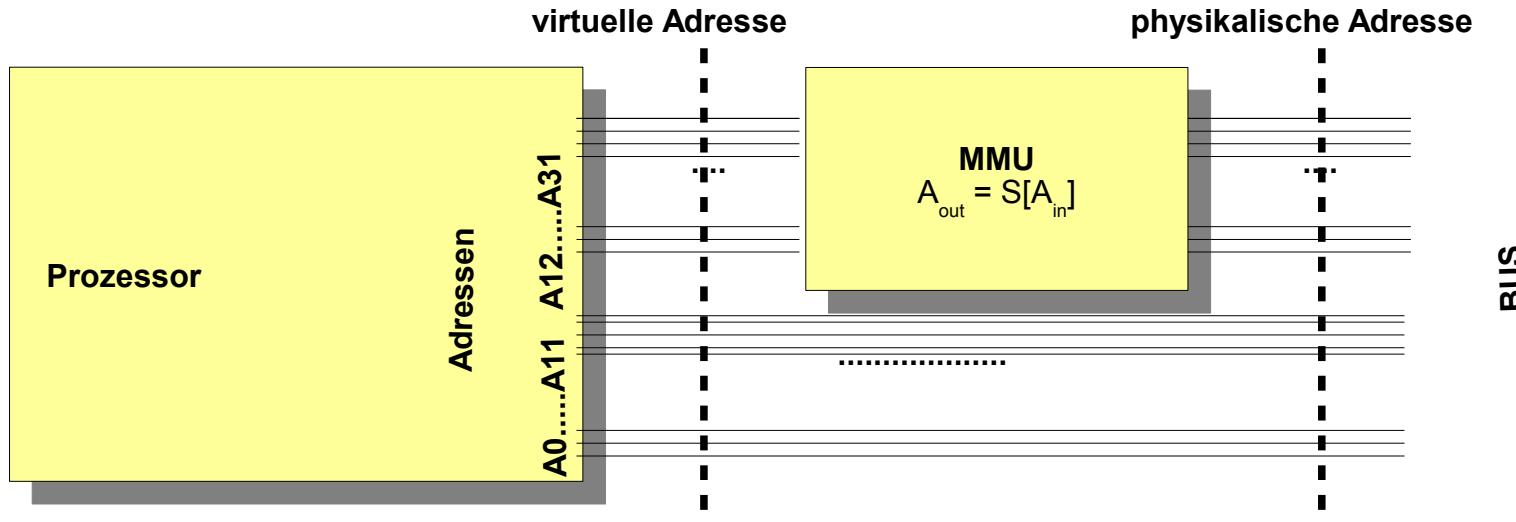


Def

- Virtueller Adressraum ist in Einheiten fester Länge unterteilt, diese heißen Seiten (Pages).
- Physikalischer Speicher wird ebenfalls in Einheiten dieser Länge unterteilt, diese heißen Kacheln oder Seitenrahmen (Page Frames).
- Transfer zwischen Hauptspeicher und Hintergrundspeicher erfolgt grundsätzlich in Einheiten von Seiten.



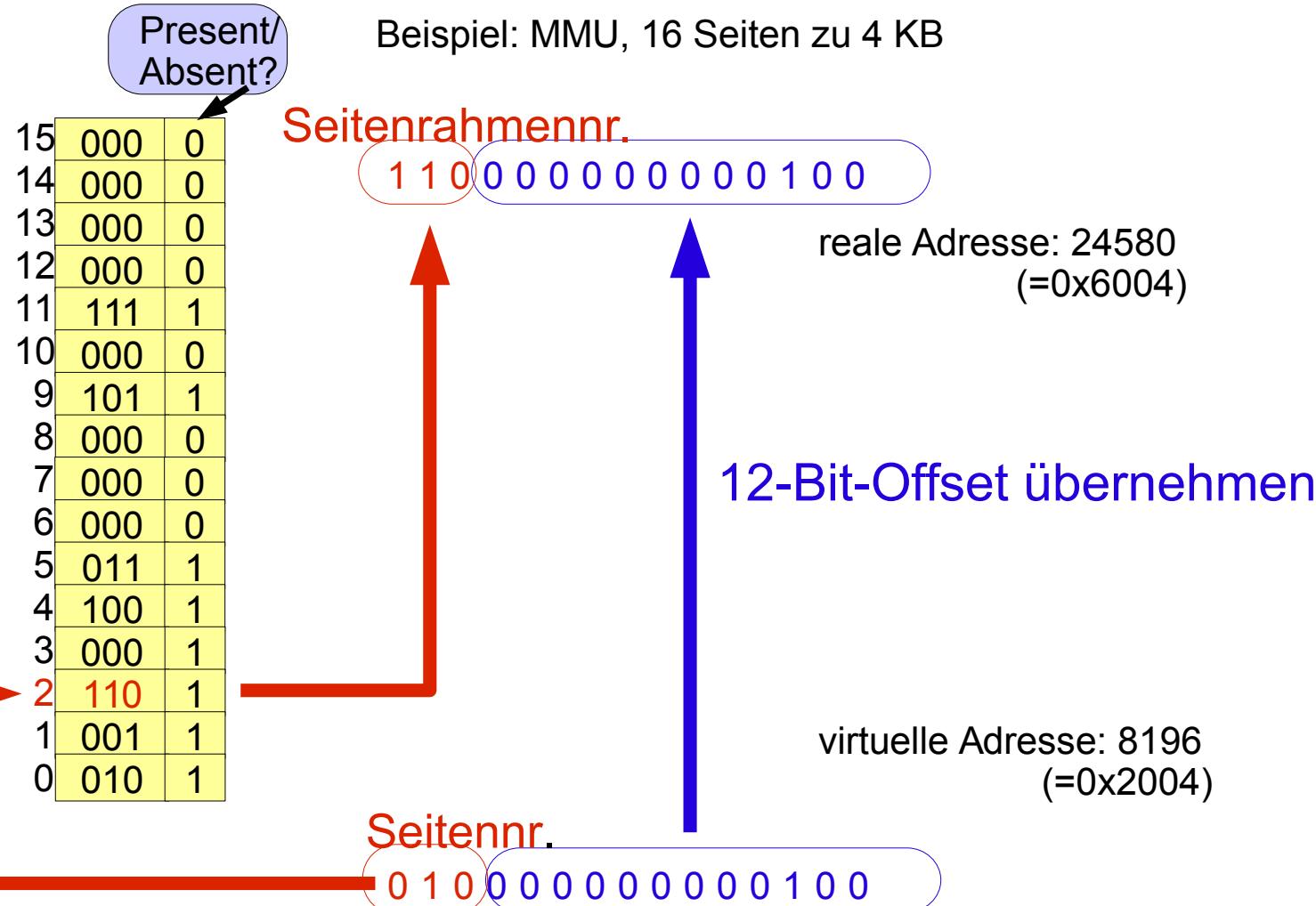
- Heute typische Größen:
 - virtueller Adressraum:
 2^{32} Bytes (4 GB) basierend auf 32-Bit Adressen,
 2^{48} Bytes (256 TB) basierend auf 64-Bit Adressen (AMD64, Intel64).
 - physikalischer Arbeitsspeicher:
i.d.R. kleiner, 2-6 GB für Arbeitsplatzrechner, Tendenz steigend.
 - Seiten / Seitenrahmen: typisch 4 KB oder 8 KB.



Def

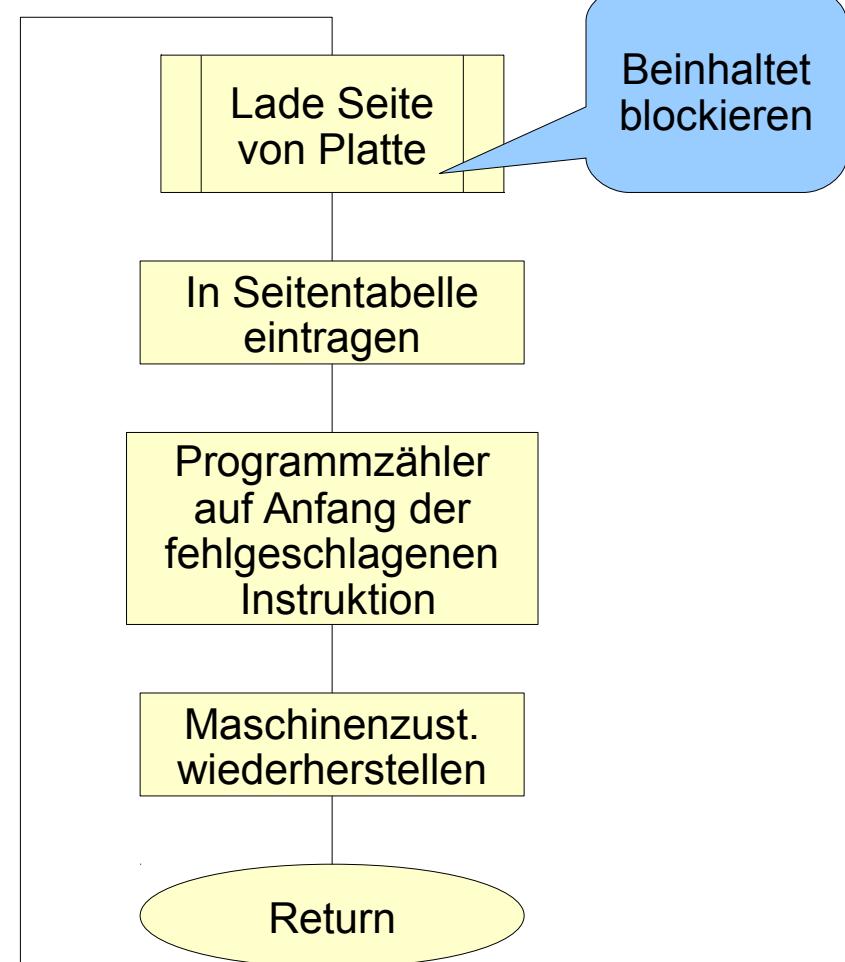
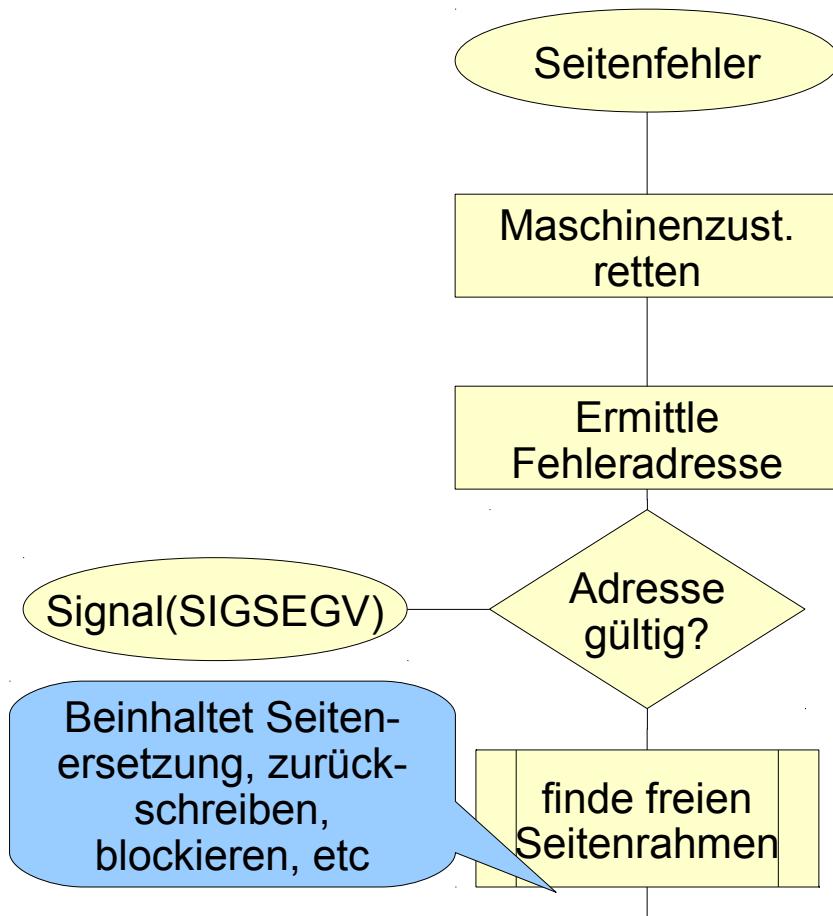
- Die abstrakte Speicherfunktion kann bildlich durch eine Tabelle repräsentiert werden, bezeichnet als Seitentabelle.
- Die Seitentabelle ordnet jeder Seite des virtuellen Adressraums einen Seitenrahmen des Hauptspeichers oder ein Kennzeichen zu, dass die Seite sich nicht im Hauptspeicher befindet (nicht eingelagert ist). Dieses Kennzeichen wird Present/Absent-Bit genannt.
- Benennt ein Programm eine virtuelle Adresse, deren zugehörige Seite nicht eingelagert ist, so erzeugt die MMU eine Unterbrechung der CPU. Diese Situation wird als Seitenfehler (Page Fault) bezeichnet.

MMU-Funktionsprinzip



- Behandlung eines Seitenfehlers besteht prinzipiell in der Einlagerung der fehlenden Seite vom Hintergrundspeicher in einen freien Seitenrahmen des Arbeitsspeichers.
- Einlagerung als Reaktion auf einen Seitenfehler wird als Demand Paging bezeichnet.
- Liegt kein freier Seitenrahmen vor, muss ein solcher gewonnen werden. I.d.R. hält das Betriebssystem einen Vorrat an freien Rahmen vor. (Algorithmen für die Auswahl zu ersetzender Seiten werden detailliert in 9.4 besprochen).
- Falls die zu ersetzende Seite während ihrer Lebenszeit im Speicher verändert wurde (als Dirty Page bezeichnet), wird sie auf die Platte zurückgeschrieben (Update). Der belegte Seitenrahmen ist danach frei.
- Falls die zu ersetzende Seite nicht verändert wurde (enthält z.B. Programmcode), ist kein Zurückschreiben erforderlich.

Behandlung eines Seitenfehlers (2)



Problem:

- Geschickte Auswahl von zu ersetzenen Seiten des Hauptspeichers:
Wird eine häufig benutzte Seite ausgelagert, ist die Wahrscheinlichkeit groß, dass diese schon bald wieder eingelagert werden muss, was Zusatzaufwand für das Betriebssystem und Blockierungszeiten für den betroffenen Prozess mit sich bringt.

Ziel:

- Gute Performance des Systems.

Seitenersetzungsalgorithmen (Page Replacement Algorithms) sind ein klassisches Problem der Betriebssysteme, vielfach theoretisch und experimentell untersucht.

In der BS-Literatur betrachtete Seitenersetzungsverfahren:

- Der optimale Seitenersetzungsalgorithmus.
- Not-Recently-Used (NRU).
- First-In, First-Out (FIFO) und Second-Chance.
- Clock-Algorithmus.
- Least-Recently-Used (LRU).
- Aging-Algorithmus.

Algorithmus:

- Wird Seitenrahmen benötigt, bestimme für alle eingelagerten Seiten, wieviele Instruktionen lang die Seite zukünftig durch die noch auszuführenden Instruktionen nicht referiert werden wird.
- Wähle die Seite zur Ersetzung aus, die am längsten nicht referiert werden wird.

Bemerkungen:

- Algorithmus schiebt den Seitenfehler, der dazu führt, dass die auszulagernde Seite wieder eingelagert werden muss, am weitesten in die Zukunft.
- Die benötigte Information über das zukünftiges Programmverhalten ist aber i.d.R. nicht vorhanden!
- Algorithmus ist daher nicht realisierbar, nur für Vergleichszwecke interessant.

Voraussetzungen:

- Verwendung der im Seitendeskriptor gespeicherten Status-Bits je Seite (vgl. VL Rechnerarchitektur):
 - R-Bit (Referenced: gesetzt, wenn auf die Seite lesend oder schreibend zugegriffen wird).
 - M-Bit (Modified: gesetzt, wenn auf die Seite schreibend zugegriffen wird).
- Status-Bits werden durch die Hardware aktualisiert.
- Gesetzte Status-Bits können durch das Betriebssystem (softwaremäßig) zurückgesetzt werden.

Algorithmus:

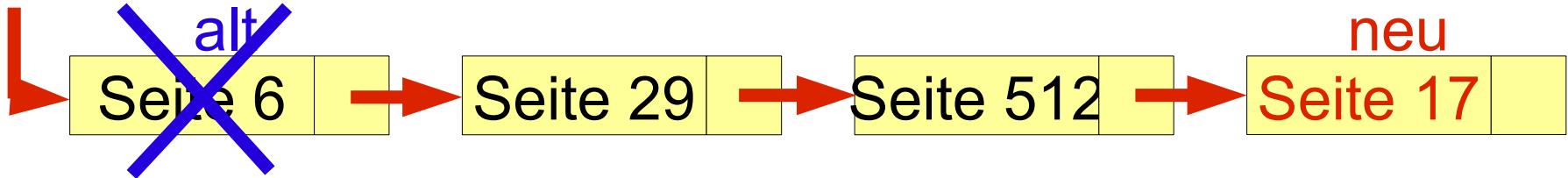
- Bei Prozessstart werden beide Status-Bits für alle Seiten des Prozesses zurückgesetzt.
- Periodisch, z.B. bei jeder Uhr-Unterbrechung, wird das Referenced-Bit aller Seiten durch das Betriebssystem zurückgesetzt. Dadurch können im folgenden referierte Seiten von nicht referierten unterschieden werden.
- Uhr-Unterbrechungen löschen das Modified-Bit *nicht*. Es wird weiter für die Entscheidung über das Zurückschreiben benötigt.
- Tritt ein Seitenfehler auf, teile alle eingelagerten Seiten in die folgenden Klassen entsprechend der aktuellen Werte des R- und des M-Bits ein:
 - Klasse 0: nicht referiert, nicht modifiziert.
 - Klasse 1: nicht referiert, modifiziert.
 - Klasse 2: referiert, nicht modifiziert.
 - Klasse 3: referiert, modifiziert.
- Wähle zufällig aus der kleinste nummerierten nicht-leeren Klasse eine Seite zur Ersetzung aus.

Bemerkungen:

- Einfach.
- Effiziente Implementierung.
- Nicht optimale, aber akzeptable Performance.

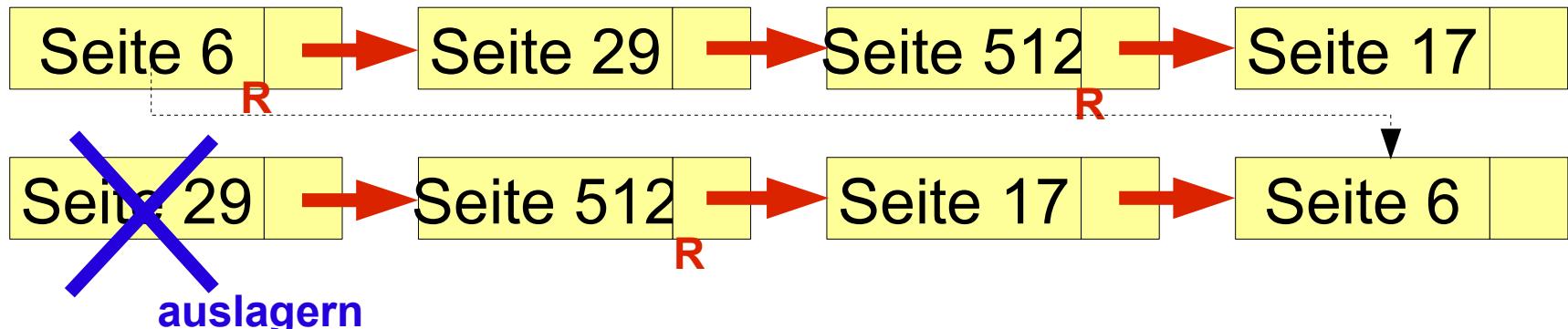
First In, First Out (FIFO):

- **Idee:** Die zuerst eingelagerte Seite wird auch zuerst wieder ausgelagert
- Verwaltung über eine Liste:
 - Bei **Einlagerung** wird Eintrag am Listenende angehängt
 - Bei **Seitenfehler**: Auszulagernde (älteste) Seite steht im Listenkopf, Listenkopf wird danach entfernt.



- Eher ungeschicktes Verfahren (die älteste Seite kann trotzdem ständig gebraucht werden, es würde dann bald wieder ein Seitenfehler für diese Seite erzeugt).

- **Idee:** Ähnlich FIFO, aber mit Beachtung des Referenced-Bits
- Bei **Seitenfehler:**
 - Vom Listenkopf ausgehend Seiten-Knoten durchlaufen:
 - Wenn R-Bit gelöscht: Seite auslagern, fertig
 - Wenn R-Bit gesetzt: löschen und Seite hinten anhängen
(→ Seite erhält eine „zweite Chance“)
 - Sollte überall R-Bit gesetzt sein, degeneriert Verfahren zu FIFO
(erster Eintrag ist mit gelöschtem R-Bit wieder vorne)

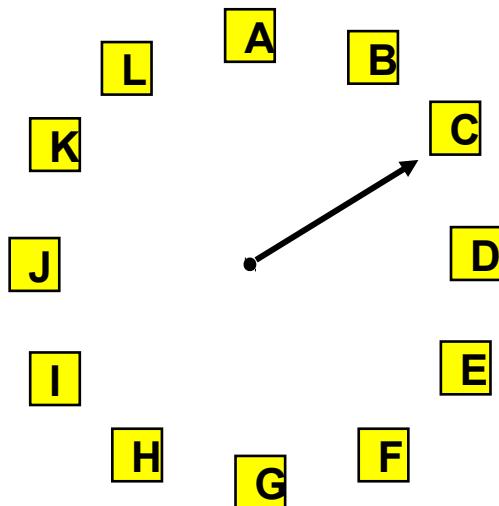


9.4.4 Clock-Algorithmus



Voraussetzungen:

- Alle Seiten werden als zyklische Liste in Form einer Uhr verwaltet.
- Der Zeiger der Uhr zeigt auf die älteste Seite.



Algorithmus:

1. Bei Seitenfehler wird die Seite überprüft, auf die der Uhrzeiger zeigt.
 2. Ist das Referenced-Bit nicht gesetzt, dann
 - Auslagern der Seite,
 - einzulagernde Seite an dieser Stelle einfügen
 - Zeiger der Uhr auf die nächste Seite verschieben.
 3. Ist das Referenced-Bit gesetzt, dann
 - Löschen des Referenced-Bit
 - Zeiger der Uhr auf die nächste Seite verschieben
 4. Wiederhole Schritt 3 , bis Seite mit einem nicht gesetzten Referenced-Bit gefunden wird
- Bemerkung:
- Der Clock-Algorithmus ist eine bzgl. der Implementierung verbesserte Form des Second-Chance Algorithmus.

Voraussetzungen:

- Gute Annäherung an den optimalen Algorithmus ist möglich, wenn man aus dem Zugriffsverhalten in der jüngeren Vergangenheit auf das Verhalten in der Zukunft schließen kann.
- Beobachtung: Seiten, die schon lange nicht mehr benötigt wurden, werden auch mit hoher Wahrscheinlichkeit für längere Zeit nicht mehr benötigt werden.

Algorithmus:

- Wenn ein Seitenfehler auftritt, ersetze die Seite, die am längsten unbenutzt ist (least-recently-used, LRU).

Bemerkungen:

- LRU ist realisierbar, Implementierung ist aber aufwändig.
- Implementierungsseite:
 - Die Menge aller Seiten im Speicher wird mit einer verketteten Liste verwaltet.
 - Die gerade benutzte Seite steht am Kopf der Liste, die am längsten nicht benutzte Seite am Ende.
 - Bei jedem Speicherzugriff wird die zugegriffene Seite in der Liste gesucht und an den Anfang gestellt (Umordnung).
 - Wähle die Seite am Ende der Liste als zu ersetzende Seite.
- Eine effiziente Implementierung ist nur mit spezieller Hardwareunterstützung möglich und nur in Ausnahmefällen sinnvoll (z.B. für die Verwaltung von TLBs mit wenigen Einträgen).

Ziel:

- Approximation von LRU in Software.

Algorithmus:

- Jeder Seite wird Software-Zähler der Länge L Bit zugeordnet (z.B. 8-Bit lang), initialisiert mit 0.
- Bei jeder Uhr-Unterbrechung für jede eingelagerte Seite:
 - Der Zähler der Seite wird um 1 Bit nach rechts verschoben (Aging = Altern).
 - Das Referenced-Bit der Seite ersetzt die führende Stelle des Zählers.
 - Das Referenced-Bit wird gelöscht. (Es wird durch die Hardware im Falle eines weiteren Zugriffs automatisch neu gesetzt).
- Wenn ein Seitenfehler auftritt, wird die Seite mit dem kleinsten Zählerwert ersetzt.

Aging-Algorithmus (2)

9.4.6

Beispiel:

Seite R Zähler

0	1	10000000	1	11000000	1	11100000
1	0	00000000	1	10000000	1	11000000
2	1	10000000	0	01000000	0	00100000
3	0	00000000	0	00000000	1	10000000
4	1	10000000	1	11000000	0	01100000
5	1	10000000	0	01000000	1	10100000

Nach Uhrtick: 1

2

3

0	1	11110000	0	01111000	1	10111100
1	0	01100000	1	10110000	0	01011000
2	0	00010000	1	10001000	1	11000100
3	0	01000000	0	00100000	0	00010000
4	1	10110000	0	01011000	0	00101100
5	0	01010000	0	00101000	0	00010100

4

5

6

Bemerkungen zum Verhältnis zu LRU:

- Information über das Zugriffsverhalten auf eine Seite während eines Uhr-Intervalls wird auf ein einziges Bit vergröbert.
- Information, die älter als L Zeitintervalle ist, wird als wertlos eingestuft (geht durch Rechts-Shifts verloren).
- Eine Seite, die während n Uhrticks nicht referiert wurde, besitzt n führende Nullen im Zähler.
- Die Auswertung des Zählers muss nur nach Ablauf dieses Zeitintervalls erfolgen, nicht bei jeder Speicherreferenz.
- Die Aufzeichnung von nur einem Bit je Zeitintervall erlaubt damit nicht die Unterscheidung einer früheren oder einer späteren Referenz in dem Intervall.
- Insgesamt ist die Approximation von LRU hinreichend gut.

Ziel:

- Kennenlernen von Aspekten, die beim Entwurf von Paging-Systemen besondere Beachtung finden, da sie starken Einfluss auf die Performance haben.

Überblick:

1. Working Set Modell
2. Lokale oder globale Seitenersetzung
3. Seitengröße
4. Implementierungsprobleme

9.5.1 Das Working Set Modell



Def

- Prozesse zeigen i.d.R. ein Lokalitätsverhalten.
- Die Menge der Seiten, die ein Prozess augenblicklich nutzt, wird sein Working Set (Arbeitsbereich) genannt (P. Denning, 1968).
- Ist der zugeordnete Hauptspeicher zu klein, um das Working Set aufzunehmen, so entstehen sehr viele Page Faults.
- Wenn nach wenigen Zugriffen die Seite wieder benötigt wird, die gerade verdrängt wurde, so sinkt die Ausführungsgeschwindigkeit auf einige Instruktionen je Seiteneinlagerung (typ. mehrere msec). Man bezeichnet eine solche Situation als Thrashing.

- Einlagern von Seiten, obwohl noch keine entsprechenden Seitenfehler vorliegen, wird als Prepaging bezeichnet (im Gegensatz zum üblichen Demand Paging).
- Ist das Working Set eines Prozesses bekannt, kann mittels Prepaging nach einem Prozesswechsel verhindert werden, dass der Prozess durch eine große Anzahl von Seitenfehlern seine Umgebung zunächst wieder aufbauen muss, in dem das Working Set vollständig eingelagert wird (Working Set Model, Denning, 1970).
- Durch das Working Set Modell kann die Seitenfehlerrate drastisch gesenkt werden.

- Ansatz zur Bestimmung des Working Sets:
 - Verwendung des Aging-Algorithmus (siehe 9.4.6).
 - Eine 1 in den höherwertigen n Bits des Alterungszählers kennzeichnet die zugehörige Seite als zum Working Set gehörig.
 - Wird eine Seite für n Uhrticks nicht referiert, so scheidet sie aus dem Working Set aus.
- Verbesserung des Clock-Algorithmus (siehe 9.4.4) ist möglich:
 - Seiten, die zum Working Set gehören, werden bei gelösctem Referenced Bit übersprungen.
 - Dieser Algorithmus wird wsclock (Working Set Clock) genannt.

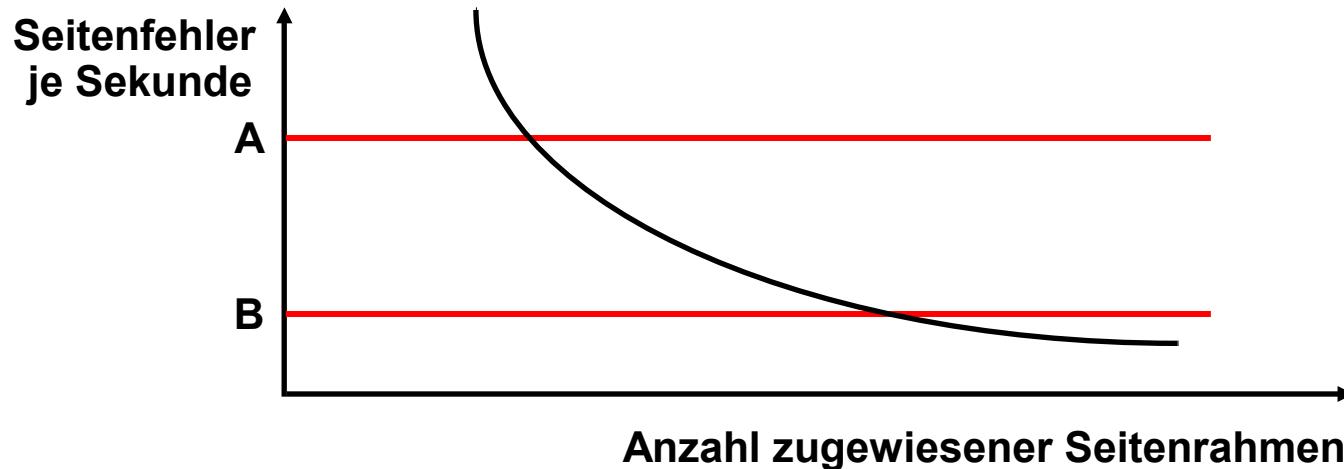
- Im Mehrprogrammbetrieb haben mehrere Prozesse Seiten im Speicher.
- Lokale Seitenersetzung: bei der Auswahl einer zu ersetzenenden Seite werden nur die Seiten des Prozesses selbst betrachtet
- Globale Seitenersetzung: *alle* im Speicher befindlichen Seiten, also auch die anderer Prozesse, werden in Betracht gezogen.
- Lokale Seitenersetzungs-Algorithmen ordnen jedem Prozess eine feste Anzahl von Seitenrahmen zu, globale Algorithmen alloziieren Seitenrahmen dynamisch.
- Globale Algorithmen arbeiten i.a. mit besserer Performance.
- Im Falle eines globalen Algorithmus muss das Betriebssystem dynamisch entscheiden, wie viele Seitenrahmen jeder Prozess zugewiesen bekommt. Einfacher Ansatz: Betrachtung des Working Set, verhindert aber kein Thrashing.

Mindestgröße:

- Wie viele Seiten kann eine einzige Maschineninstruktion referenzieren?
 1. Befehlscode lesen
 2. Quelloperand holen
 3. Zieloperand speichern
 - Operand/Maschineninstruktion i.d.R. > 1 Byte
 - Jeder kann (worst-case) über max. 2 Seiten verteilt sein
- Mindestens 6^(*) Seitenrahmen für einen Prozess, sonst u.U. „Thrashing innerhalb einer Instruktion“! ()

(*) Architekturabhängig: z.B. MIPS: 2 Seitenrahmen

- Bester Ansatz:
Kontrolle der Seitenfehlerfrequenz (Page Fault Frequency PFF)
über die Anzahl der zugeordneten Seitenrahmen.
- Typischer Verlauf:



- **Seitenfehlerrate zwischen A und B wird als akzeptabel angesehen. Steigt die Seitenfehlerrate über A, werden zusätzliche Seitenrahmen zugeordnet, sinkt sie unter B, werden Seitenrahmen entzogen.**

9.5.3 Seitengröße



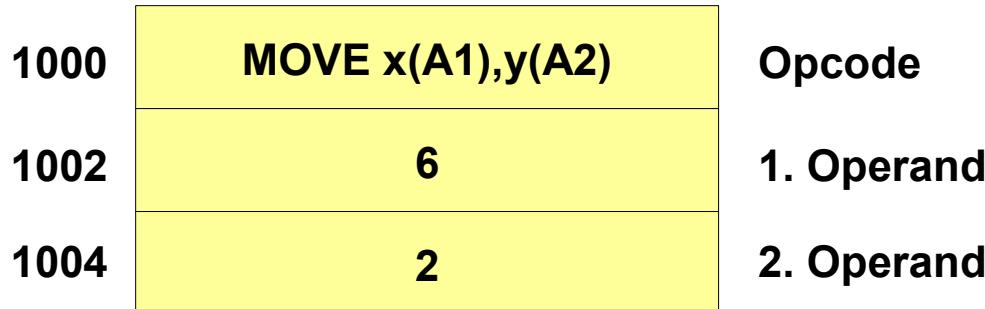
- Bestimmung der optimalen Seitengröße erfordert ein Ausbalancieren von verschiedenen widersprechenden Faktoren:
 - Geringe interne Fragmentierung spricht für eine kleine Seitengröße.
 - Kleine Seiten erfordern eine große Seitentabelle.
 - Kosten für die Ein-/Auslagerung sind stark von Positionierzeiten und Rotationsverzögerung der Platte beeinflusst. Spricht für große Seitengröße.
 - Hardware-Aufwand für Memory Management Unit.
- Heutige Seitengrößen: 512 B - 8 KB, typisch 4 KB

9.5.4 Implementierungsprobleme



Instruction Retry nach Seitenfehler:

- Eine Instruktion besteht aus mehreren Worten
- Beispiel (MC68000): MOVE 6(A1), 2(A2)



- Bei Seitenfehler kann PC=1000, 1002 oder 1004 sein
- „Wieviel von der Instruktion ist schon abgearbeitet?“
- „Wo/wie muss nach Seitenfehler fortgefahren werden?“
- ➔ Kann kompliziert werden ...

- Sperren von Seiten im Speicher
 - z.B. für Pufferbereiche während gestarteter I/O-Aufträge.
 - Entsprechende Seiten von blockierten Prozessen dürfen dann nicht ausgelagert werden (reserviert für DMA-Transfer).
 - Typische Operationen: pin / unpin auf Seiten.
- Memory Sharing
 - Gemeinsam benutzte Seiten verschiedener Prozesse (insbesondere Code-Seiten) müssen berücksichtigt werden.
 - Auslagern würde Seitenfehler für Partner verursachen.
 - I.d.R. werden spezielle Datenstrukturen mit Referenzzählern verwendet.

- Paging-Dämonen
 - Ein Dämon (Daemon) ist ein Hintergrundprozess, der nicht mit einem Benutzer in Interaktion tritt (hat kein Terminal).
 - Ziel: Vorhalten einer Anzahl freier Seitenrahmen, um bei Auftreten eines Seitenfehlers keine Verzögerung für das Auslagern einer Seite in Kauf nehmen zu müssen (Performance-Aspekt).
 - Der Page Daemon wird z.B. periodisch aktiviert, um die Speichersituation zu untersuchen:
 - Liegen zu wenige freie Rahmen vor, beginnt er, Seiten mithilfe des Seitenersetzungsalgorithmus zur Auslagerung auszuwählen.
 - Wurden sie modifiziert, veranlasst er das Zurückschreiben auf Platte.
 - Der ursprüngliche Inhalt als „frei“ geführter Seitenrahmen bleibt bekannt.
 - Erfolgt ein erneuter Zugriff auf die ursprüngliche Seite, bevor sie überschrieben wird, wird sie aus dem „Frei“-Pool zurückgewonnen.
 - Dieser Vorgang wird Page Reclaiming genannt.



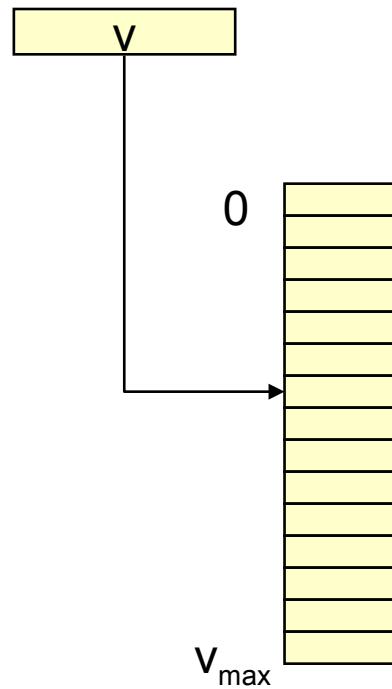
Ziel:

- Bisher wurden Paging-Systeme betrachtet, die einen sogenannten eindimensionalen virtuellen Speicher offerieren.
- Im folgenden wird sogenannter segmentierter virtueller Speicher besprochen.

Eindimensionale / Zweidimensionale virtuelle Adressräume:

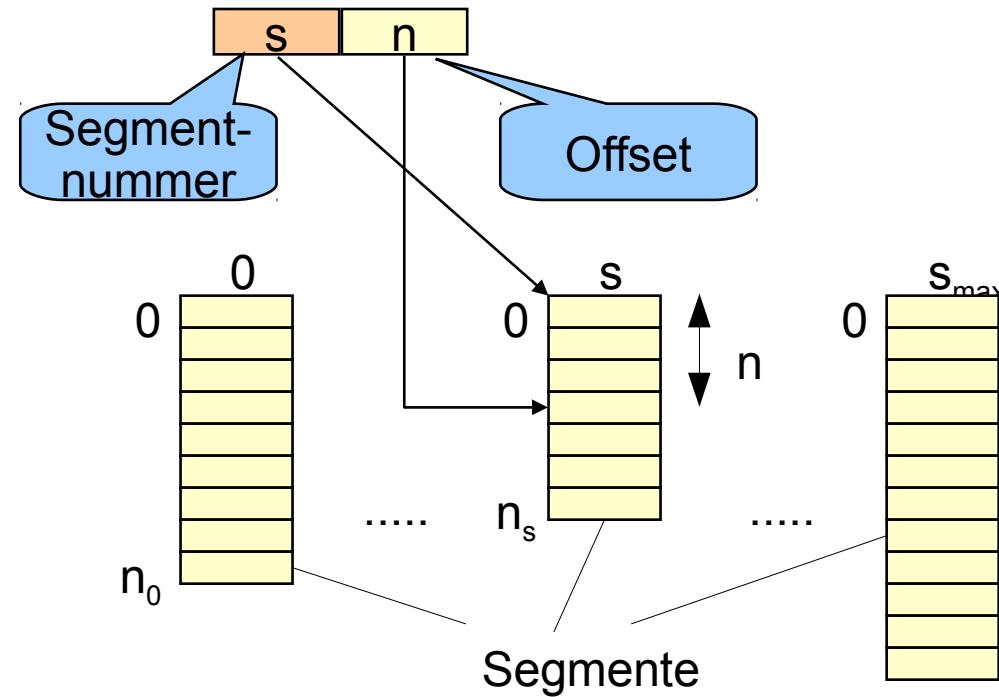
eindimensional:

virt. Adresse



zweidimensional:

virt. Adresse

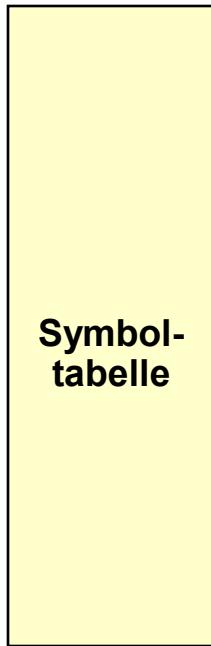


Def

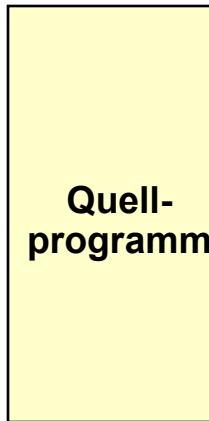
- Zweidimensionale (zweiteilige) virtuelle Adresse:
 - Segmentnummer, die ein Segment selektiert,
 - Adresse im selektierten Segment (Offset).
- Die Menge aller Segmente definiert den sog. Segmentraum.
- Ein Segment besitzt einen maximalen linearen Adressbereich $\{0, \dots, L_{\max}\}$. Jedes Segment hat damit die Eigenschaften eines (linearen) Adressraums.
- Verschiedene Segmente können unterschiedlich lang sein.
Die Länge jedes Segments liegt zwischen 0 und dem erlaubten Maximum L_{\max} .
- Die Segmentlänge darf sich i.d.R. während der Lebenszeit des Segments ändern.
- Ein Segment ist eine logische Speichereinheit und als solche dem Programmierer bewusst (sollte es zumindest sein).

Mögliche Segmente für einen Compiler-Lauf:

Segment
0



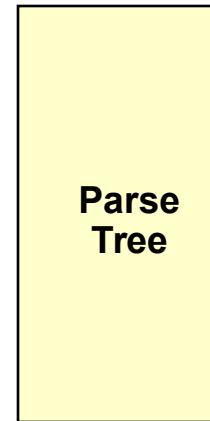
Segment
1



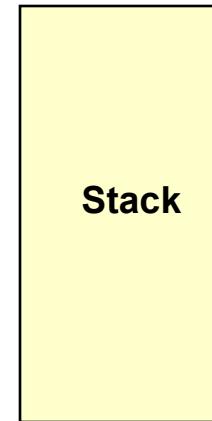
Segment
2



Segment
3



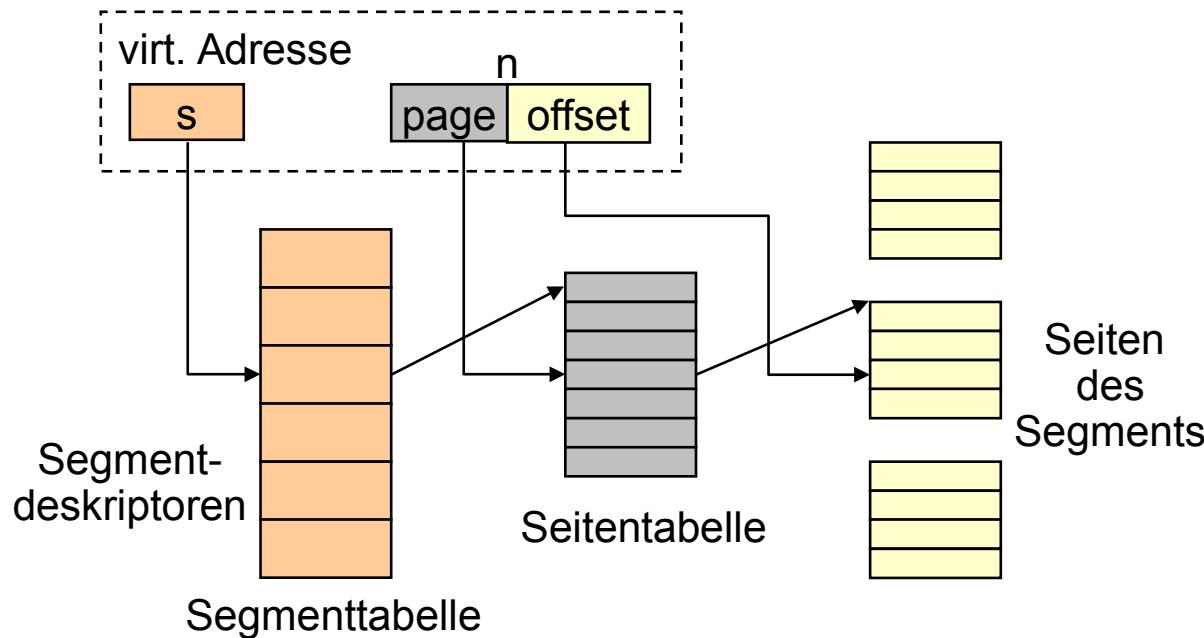
Segment
4



**Jedes Segment kann, unabhängig von
allen anderen, wachsen und schrumpfen**

Segmentierung mit Paging

- Ein eingelagertes Segment *muss nicht mehr* vollständig im Hauptspeicher sein.
- Jedes Segment besteht aus einer Folge von Seiten.
- Prinzipielle Adressierung:



Ziel:

- Kennenlernen der UNIX-Speicherverwaltung.

Überblick:

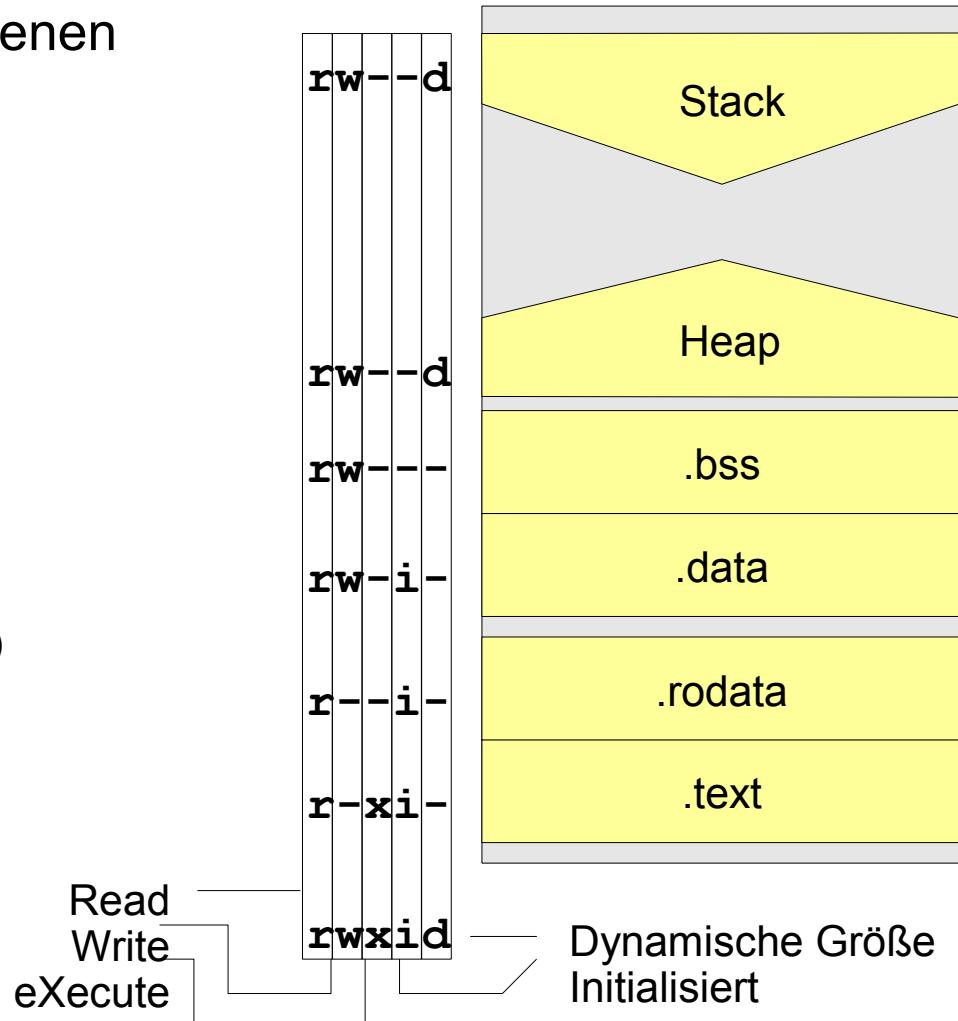
1. Speichermodell.
2. Systemaufrufe zur Speicherverwaltung.
3. Swapping.
4. Paging.

9.7.1 Speichermodell



9.7.1

- Jeder Prozess besitzt einen eigenen 32-bit virtuellen Adressraum.
- Der Adressraum enthält aus Benutzersicht (im Anwenderadressbereich) 3 Segmente
 1. Text-Segment
 - Programmcode (.text)
 - Nur-Lese Daten (.rodata)
 2. Daten-Segment
 - Initialisierte Daten (.data)
 - Nicht-initialisierte Daten (.bss)
 - Heap (für malloc() & Co.)
 3. Stack-Segment
 - für lokale Variablen etc.



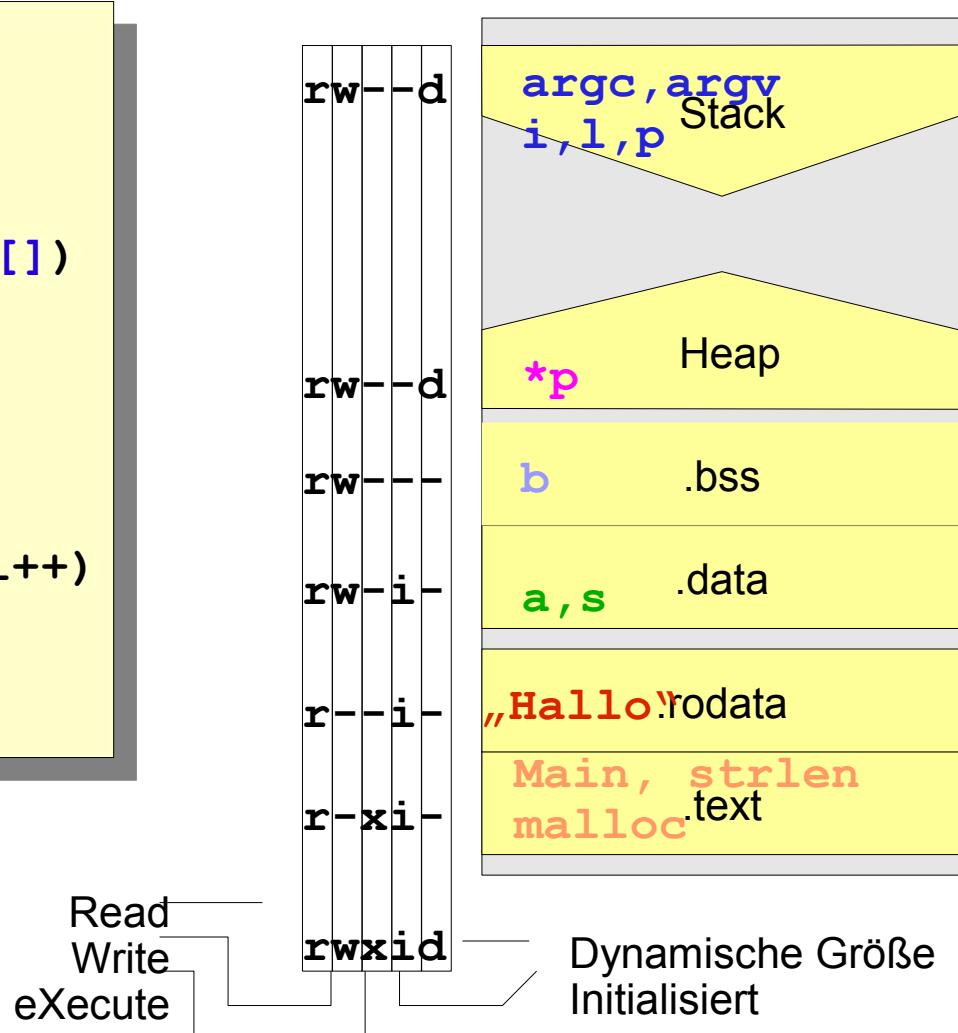
Speichermodell (2)

9.7.1

```

static int a = 5;
int b;
char *s = „Hallo“;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l+1);
    for(i = 0; i < l; i++)
        p[i] = s[i];
}
  
```

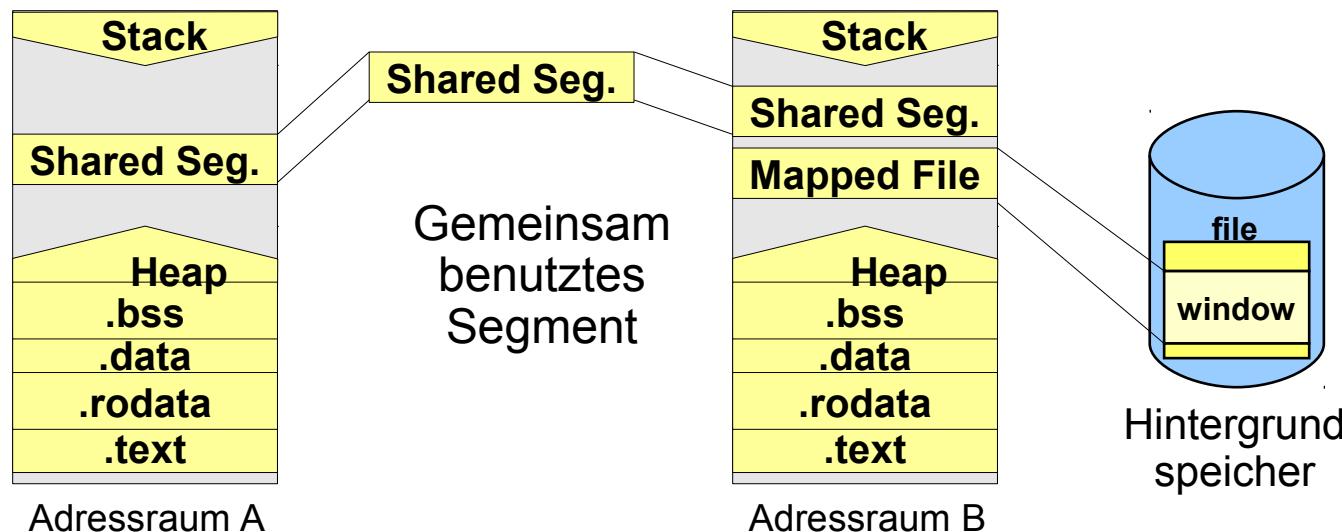


- Text-Segment kann von mehreren Prozessen gemeinsam benutzt werden, Daten und Stack sind privat.

Speichermodell (3)

9.7.1

- Zusätzlich kann ein Adressraum in neueren UNIX-Systemen ein oder mehrere Segmente der folgenden beiden Typen enthalten:
 - Gemeinsam benutztes Speichersegment, das gleichzeitig in mehrere Adressräume eingeblendet ist (z.B. als Shared Memory Data Segment oder Shared Library),
 - Mapped File-Segment zur Aufnahme eines Fensters eines Memory Mapped Files (abgebildete Datei, siehe Kapitel Dateisysteme). Auf ein Mapped File kann ohne explizite E/A-Operationen zugegriffen werden.



- Virtueller Adressraum wird in Regionen unterteilt.
- Eine Region ist ein zusammenhängender Bereich virtueller Adressen und kann ein identifizierbares Segment (Objekt) aufnehmen (Text-, Daten-, Stack-Segment und andere, s.o.).
- Einem zugeordneten Segment entspricht eine Folge von Seiten des Adressraums.
- Für jede Region können individuelle Schutzattribute festgelegt werden (read-only, read-write, read-execute). Diese übertragen sich auf die repräsentierenden Seiten.
- Gemeinsam genutztes Segment:
 - entspricht Regionen in verschiedenen Adressräumen, denen das Segment zugeordnet ist.
 - Anfangsadresse der Regionen kann in Adressräumen unterschiedlich sein.
(Achtung: Adressen im gemeinsam benutzten Segment sind dann i.d.R. bedeutungslos).

Überblick:

- Im POSIX-Standard existieren keine expliziten Festlegungen.
- Für die Anwendungsprogrammierung wird auf ANSI C-Standardbibliotheksfunktionen zur dyn. Speicherverwaltung zurückgegriffen.
- Im folgenden werden betrachtet:
 - klassische Dienste zur Behandlung der Standard-Segmente.
 - Dienste für Shared Memory Segmente.
 - Dienste für Memory-Mapped Files.
 - Dienste zur Adressraum-Manipulation.

Dienste für Memory-Mapped Files sowie zur Adressraum-Manipulation sind noch nicht auf allen heutigen UNIX-Varianten verfügbar.

- Zur absoluten bzw. relativen Veränderung der Größe des Datensegments (Wachsen oder Schrumpfen) existieren die Systemaufrufe `brk()` bzw. `sbrk()` (letzterer ist häufig eine Bibliotheksfunktion):

```
caddr_t brk(caddr_t addr); caddr_t sbrk(int incr);
```

- Auf Programmebene werden i.d.R. die ANSI C-Bibliotheksfunktionen `malloc()` und `free()` benutzt. Implementierung von `malloc()` greift z.B. auf `brk()` zurück, wenn die Heap-Größe erweitert werden muss.
- Die Größe des Stack-Segments wird nicht auf Programmebene verändert, sondern vom Betriebssystem implizit vergrößert, wenn sich der Stack zur Ausführungszeit als zu klein erweist.
- Die Größe des Text-Segments ist statisch und damit nicht veränderbar.

- Shared Memory Segmente sind Bestandteil der System V IPC-Mechanismen und stellen gemeinsam von mehreren Prozessen benutzbare Segmente mit festlegbaren Zugriffsattributen dar.
- Die Benutzung von Shared Memory Segmenten setzt i.d.R. die korrekte Synchronisation der Prozesse voraus (z.B. mittels Semaphoren).
- Shared Memory Segmente sind i.d.R. die effizienteste Form der Interprozesskommunikation.
- Maximale Anzahl von Shared Memory Segmenten im System (`SHMMNI`) und je Prozess (`SHMSEG`) und deren maximale und minimale Größe (`SHMMAX`, `SHMMIN`) werden bei der Konfiguration des BS-Kerns festgelegt.
- Shared Memory Segmente überleben, wie die anderen System V IPC-Objekte die sie erzeugenden Prozesse und sind ohne besondere Maßnahmen erst mit einem Neustart des Systems zerstört. Zur Information über die existierenden IPC-Objekte bzw. zu ihrer Zerstörung existieren die Kommandos `ipcs` bzw. `ipcrm`.
(Bitte am Ende jeder Praktikumssitzung zum Aufräumen nutzen !)

- Operationen:
 - Header-Dateien: `<sys/types.h>`, `<sys/ ipc.h>`, `<sys/shm.h>`
 - `int shmget(key_t key, int size, int shmflag):`

erzeugt oder öffnet ein Shared Memory Segment mit dem externen Schlüssel `key` und liefert den zugehörenden Identifier `shmid` des Segments zurück (oder -1 bei Fehler).

Die Behandlung von `key` und `shmflag` ist insgesamt analog wie bei Semaphoren. Ist insbesondere `key` auf `IPC_PRIVATE` gesetzt, so wird ein privates Segment (ohne öffentlichen `key`) erzeugt, das für verwandte Prozesse genutzt werden kann.

Wird ein Schlüssel verwendet und in `shmflag` das `IPC_CREAT`-Bit gesetzt und existiert noch kein IPC-Objekt zu `key`, so wird ein neues Segment der Mindestgröße `size` erzeugt.

Für das Öffnen eines existierenden Objekts darf das `IPC_CREAT`-Flag nicht gesetzt sein. Hat `size` einen Wert größer als die aktuelle Größe des Segments, so schlägt der Aufruf fehl, der Wert 0 akzeptiert das Segment in der aktuellen Größe.

Die Zugriffsrechte für das Segment werden durch einen entsprechenden `mode` in `shmflag` bei der Erzeugung mit festgelegt, typisch 0600 für Lesen und Schreiben durch alle Prozesse des Eigentümers des Segments oder 0666 für beliebige Prozesse. Beim Öffnen eines Segments gibt der Zugriffsmodus die geforderten Rechte an.

- `void * shmat(int shmid, void * addr, int flag)`

blendet das Segment `shmid` in den Adressraum des Aufrufers an der virtuellen Adresse `addr` ein (attach). Wird für `addr` der Wert `NULL` gewählt, so wird die Startadresse des Segments im Adressraum vom Betriebssystem festgelegt (Konflikt ausgeschlossen), ansonsten an der spez. Adresse, die i.d.R. innerhalb des Anwenderadressbereichs auf Seitengrenze ausgerichtet sein muss. Die getroffene Startadresse wird zurückgegeben (oder -1 bei Fehler). Ist in `flag` `SHM_RDONLY` gesetzt, so wird das Segment nur mit Leserecht in den Adressraum eingeblendet. `shmat` muss nach Ausführung von `shmget` und vor der eigentlichen Nutzung des Segments ausgeführt werden.

- `int shmdt(void * addr)`

blendet ein zuvor mittels `shmat` an die Adresse `addr` eingeblendetes Segment wieder aus dem Adressraum des Aufrufers aus (detach). Die Existenz des Shared Memory Segments bleibt hiervon unberührt.

- `int shmctl(int shmid, int cmd, struct shmid_ds * arg)`

ermittelt/ändert den Status eines Shared Memory Segments oder löscht es.
Kommandos (Werte für `cmd`) sind

- `IPC_STAT` (Abfragen des Status mit Ergebnis in einer Struktur `shmid_ds`, auf die `arg` zeigt).
- `IPC_SET` (Setzen der Eigentümer-UID/GID und der Zugriffsrechte bei passender effektiver UID).
- `IPC_RMID` (Löschen des Segments bei passender effektiver UID; wird effektiv, wenn der letzte Prozess `shmdt` ausführt, zwischenzeitlich ist kein `shmat` mehr möglich, Referenzzähler).
- `SHM_LOCK` (Sperren des Segments, nur durch Prozess mit root-Recht).
- `SHM_UNLOCK` (Entsperren des Segments, nur durch Prozess mit root-Recht).

- Einblenden einer Datei oder eines Gerätes in eine Region des Adressraums (Kombination von Systemfunktionen der Speicherverwaltung, des I/O-Systems und des Dateisystems)
- Eingeblendete, zuvor geöffnete Datei wird wie normales Shared Memory Datensegment zugegriffen, d.h. *ohne* Nutzung der Dienste `read`, `write`, `lseek`.
- Daten werden mithilfe des normalen Paging-Mechanismus aus der hinterlagerten Datei herangeschafft.
- Veränderte Seiten der Datei werden irgendwann, bedingt durch Seitenverdrängung, dorthin zurückgeschrieben, wenn der letzte Prozess die zugrundeliegende Datei schließt oder durch sync-Vorgänge.

- Vorteile:
 - Vermeiden von Kopiervorgängen.
 - Weniger Kontext-Umschaltungen, da I/O-Systemaufrufe wegfallen.
 - Bequemer Zugriff über Zeiger.
 - Veränderungen in der Datei sind für andere Prozesse, die dieselbe Datei eingeblendet haben, *unmittelbar* sichtbar.
- Die Anwendbarkeit des Einblendvorgangs für zahlreiche Geräteklassen erlaubt den Zugriff auf Hardware-Komponenten (z.B. Geräteregister, Graphik-Bildspeicher, usw.), auch aus einem Anwendungsprogramm heraus (hohe Flexibilität, hohe Performance).

- Operationen (Überblick):
 - Header-Dateien: `<sys/types.h>`, `<sys/mman.h>`
 - `caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset):`

blendet ein Fenster der durch `fd` angegebenen, geöffneten Datei (oder Gerät), beginnend in der Datei an der Stelle `offset`, in der Länge `len` in den Adressraum des Aufrufers an der virt. Adresse `addr` mit der Rechtefestlegung `prot` ein.

- `int munmap(caddr_t addr, size_t len):`

blendet eine zuvor eingeblendete Datei für den angegebenen Adressbereich aus.

- `int msync(caddr_t addr, size_t len, int flags):`

erlaubt in durch `flags` festlegbaren Varianten, den Inhalt der im angegebenen Adressbereich eingeblendeten Datei auf die Originaldatei zurückzuschreiben.

- In neueren UNIX-Varianten (wie System V R.4), die gewisse Echtzeitanforderungen erfüllen, wurden auch Funktionen zur Speicherkontrolle bereitgestellt, etwa um sicherzustellen, dass der BS-Kern einem Echtzeitprozess nicht unfreiwillig Seiten verdrängt und dadurch beim nächsten Zugriff (relativ lange) Pagein-Zeiten anfallen.
- Operationen
 - `memctl(...)`
 - `memprotect(...)`
 - `mincore(...)`
- Hier nicht weiter betrachtet

9.7.3 Swapping

- Das klassische Unix basierte nur auf Swapping
- Demand Paging wurde durch die BSD-Version 3 erstmals eingeführt. Ein heutiges UNIX enthält beide Mechanismen.
- Die Verlagerung zwischen Arbeitsspeicher und Hintergrundspeicher wird durch die obere Schicht des zweistufigen Schedulers durchgeführt. Dieser heisst Swapper (Prozess mit pid 0, unterliegt normalem Scheduling (mit hoher Priorität), läuft allerdings nur im Kernel Mode, in System V R.4 sched genannt).
- Der Swap Space ist mindestens eine Plattenpartition (oder auch eine Swap-Datei, UNIX System V R.4).
- Freispeicherverwaltung im Swap-Space geschieht nach dem First-Fit-Algorithmus mittels verketteter Listen, die alle Freibereiche aller Swap-Partitionen umfassen.

Überblick:

- Paging wird im folgenden am Beispiel System V R.4 skizziert (Details in [Goodheart, Cox], siehe Literaturliste). Paging in System V R.4 ist weitgehend an das 4.3BSD-UNIX angelehnt.
- UNIX wendet Demand Paging an für durch das Swapping "eingelagerte" Prozesse.
- Seiten der Segmente aller Adressräume werden dynamisch "on demand" ein- und ausgelagert. UNIX benutzt kein Prepaging.
- Das Paging wird gemeinsam durch den BS-Kern sowie durch einen Prozess erbracht, der als Pager Daemon bezeichnet wird (Prozess mit pid 2, in System V R.4 pageout genannt) und im Kernmodus läuft.
- Der Pager wird periodisch geweckt, um seine Dienste zu erbringen. Insbesondere ist es seine Aufgabe, eine bestimmte Menge freier Seitenrahmen vorzuhalten.

- Der Seitenersetzungsalgorithmus wird vom Page Daemon (pageout) ausgeführt.
- Der angewendete Algorithmus ist
 - global (betrifft also die Seiten aller Prozesse),
 - eine Verallgemeinerung des Clock-Algorithmus, genannt Two-Handed-Clock (Zwei-Zeiger-Uhr-Algorithmus),
 - verwendet Page Reclaiming.
- Der erste Zeiger der Uhr (fronthand) setzt das Referenced-Bit zurück, der zweite Zeiger (backhand) überprüft das Referenced-Bit und gewinnt freie Seitenrahmen, wenn das Referenced-Bit nicht gesetzt ist. Die Zeiger sind als Indizes in das Seitenverzeichnis implementiert.
- Es ist damit kein voller Umlauf des Zeigers wie bei normalem Clock-Algorithmus erforderlich, bis mit Sicherheit ein erster freier Rahmen gefunden wird. (Performance-Problem wird beseitigt, wenn unmittelbar nach einer Scan-Pause enormer Seitenrahmenbedarf auftritt).

- Damit zusätzliche Steuerungsmöglichkeit über den Abstand der beiden Zeiger der Uhr (handspread):

Werden beide Zeiger dicht beieinander gehalten, so wird das Referenced-Bit nur für sehr häufig benutzte Seiten schon wieder gesetzt sein, wenn der zweite Zeiger vorbeikommt.

Liegen beide Zeiger maximal auseinander (360° - 1 Seite), so arbeitet der Algorithmus nahezu wie der ursprüngliche Clock-Algorithmus.

- Der Pager wird im Normalfall alle 250 ms geweckt. Er überprüft die Anzahl der freien Seitenrahmen (Scan). Es ist sein Ziel, eine gewünschte Anzahl `desfree` (desired) von Seitenrahmen in der Freiliste vorzuhalten (default: 1/16 des Arbeitsspeichers).
- Ist die Anzahl der freien Rahmen größer als der Systemparameter `lotsfree` (default: 1/4 des Arbeitsspeichers), so legt er sich unmittelbar wieder schlafen.
- Ist die Anzahl der freien Rahmen kleiner als `desfree`, so wird der Pager bei jedem Clock-Tick aktiv.
- Sinkt die Anzahl unter `minfree` (default: 1/32 des Arbeitsspeichers), so setzt unfreiwilliges Swapping ein.
- Die Scan-Rate ist hoch (`fastscan` Seiten werden untersucht), wenn die Menge freier Rahmen bei Start des Pagers `destfree` ist und sinkt kontinuierlich auf einen unteren Wert (`slowscan`), gerade wenn `lotsfree` freien Rahmen erreicht sind (ab `lotsfree` ist sie 0).
- `descan` (zwischen `slowscan` und `fastscan`) legt die Anzahl der Seiten fest, für die der Pager bei einer Aktivierung das Referenced-Bit löscht.

Was haben wir in Kap. 8 gemacht?

- Konzepte der Speicherverwaltung (Wichtig!).
- Statische Speicherverwaltung als einfachste Verfahren.
- Swapping unterstützt Systeme, deren Prozesse mehr Speicher benötigen als insgesamt zur Verfügung steht, lagern dabei aber stets ganze Prozesse aus und ein.
- Freispeicherverwaltung im Arbeitsspeicher und auf der Platte geschieht durch Verfahren basierend auf Bitmaps, verketteten Listen oder dem Buddy-System.
- Virtueller Speicher unterstützt Systeme, deren Prozesse sehr groß werden können.
- Paging und entsprechende Hardware-Unterstützung wurde im Rechnerarchitekturteil der Vorlesung besprochen.

- Seitenersetzungsalgorithmen wurden vielfach untersucht. Clock-Algorithmus und Aging sind praktikabel und brauchbar, der optimale Algorithmus und LRU sind dagegen gut, aber nicht praktikabel.
- Modellierungstechniken, wie die besprochenen Distanzketten, können für eine Analyse hilfreich sein.
- In konkreten Paging-Systemen gewinnen weitere Aspekte an Bedeutung, etwa die Abwägung des Working Set Modells gegenüber reinem Demand Paging oder lokale gegenüber globaler Seitenersetzung.
- Abschließend wurden die in UNIX eingesetzten Verfahren zur Speicherverwaltung besprochen.



Kap. 10: **Dateisysteme**

Anforderungen an informationsverarbeitende Systeme:

- Speichern und Wiederauffinden sehr großer Mengen von Informationen
- Lebensdauer der Informationen länger als die der benutzenden Prozesse (Persistenz).
- Gemeinsame Nutzung von Informationen durch Prozesse (Sharing).

Def

Eine Datei (File) ist eine logische Einheit zur Speicherung von Informationen auf externen Speichermedien. Dateisysteme (File Systems) sind die Teilsysteme eines Betriebssystems, die der Bereitstellung von Dateien dienen.

- **Datenhaltungssysteme**

	Dateisysteme	Datenbank- systeme	Objekt- management systeme
Inhalt	universell	Massendaten weniger struktureller Typen	Beziehungen stehen im Vordergrund
Gespeicherte Information	passiv	passiv	aktiv
Semantik aufprägender Code	extern	extern	intern (Typen)
Zugriff	über Namen, einfache Navigation	komplexe assoziative Suchfunktionen	komplexe Such- und Navigations- funktionen

- 10.1 Dateien
- 10.2 Verzeichnisse
- 10.3 Implementierung von Dateisystemen
- 10.4 Sicherheit
- 10.5 Schutz
- 10.6 Zusammenfassung

UNIX wird beispielhaft in den jeweiligen Abschnitten behandelt.

Zunächst Einführung von Dateien aus Benutzersicht.

Gliederung

1. Benennung von Dateien
2. Dateistrukturen
3. Dateitypen
4. Dateizugriff
5. Dateiattribute
6. Dateioperationen
7. Memory-Mapped Files

10.1.1 Benennung von Dateien



- Datei als Abstraktion zur Speichern und Lesen von Information auf einem Hintergrundspeicher
- Benutzer muss *nicht* wissen, wie und wo die Information abgelegt wird, noch wie der Hintergrundspeicher (i.d.R. Platte) im Detail funktioniert.
- Dateinamen werden benutzt, um in Dateien abgelegte Information zu identifizieren und wiederaufzufinden:
 - Namensvergabe erfolgt bei Dateierzeugung durch den erzeugenden Prozess.
 - Datei und Dateiname bleiben bestehen, auch wenn der Prozess terminiert.
 - Dateiname kann von anderen Prozessen benutzt werden, um Zugang zu der gespeicherten Information zu bekommen.

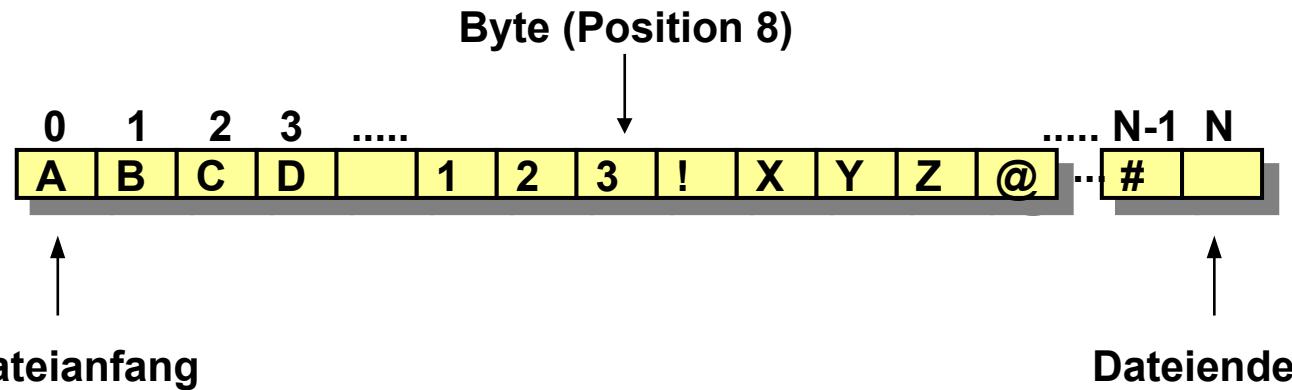
Def

- Der Dateinamensraum definiert die Menge der zulässigen Dateinamen.
- Regeln für die Konstruktion zulässiger Dateinamen stark systemabhängig:
 - Unterscheidung zwischen Groß- und Kleinbuchstaben (ja: UNIX, nein: MS-DOS).
 - Länge der zulässigen Dateinamen (BSD UNIX: 255 Zeichen, MS-DOS: 8 Zeichen).
 - Verwendung von Sonderzeichen.
 - Verwendung von Namenserweiterungen (File Extensions):
 - optional (Regelfall) oder erzwungen.
 - einfach (z.B. MS-DOS) oder mehrfach (UNIX).
 - Konventionen für die Verwendung
 - z.B. .c für C-Quelldateien
 - z.T. von verarbeitenden Programmen erzwungen

10.1.2 Dateistrukturen



Die Datei als unstrukturierte Folge von Bytes:



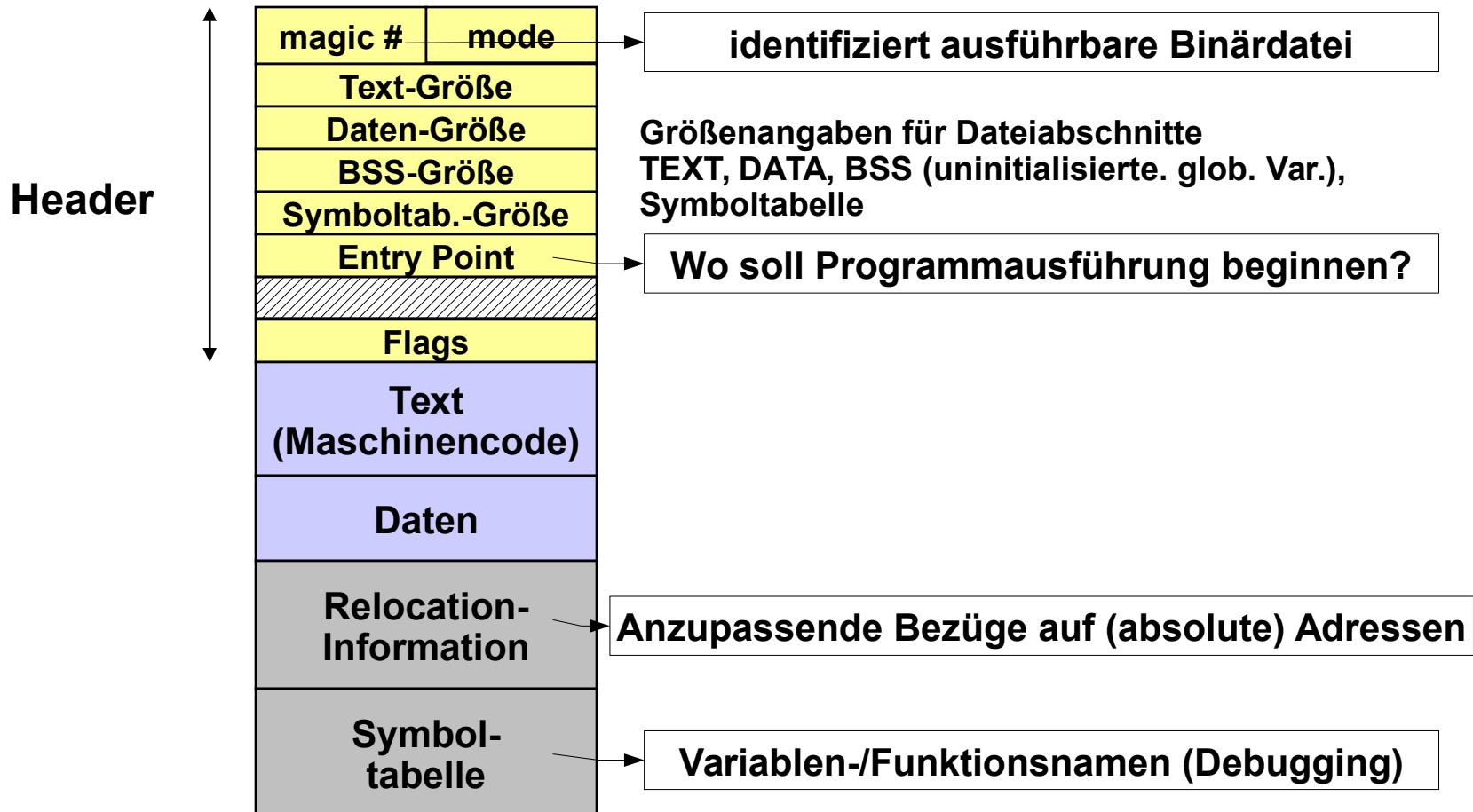
- Das Betriebssystem **weiß nichts** über den Inhalt oder dessen Struktur, sieht eine Datei ausschließlich als Container an.
- Vorteil: Maximale Flexibilität.
- Beispiele: UNIX, MS-DOS.

10.1.3 Dateitypen



- Gewöhnliche Dateien (Regular Files):
 - ASCII- („Text-“) Dateien:
 - Folge von Textzeilen variabler Länge, i.d.R. durch *betriebssystemabhängiges* Steuerzeichen getrennt (UNIX: „\n“, MacOS: „\r“, Windows: „\r\n“)
 - Vorteil: Mit Editor einfach manipulierbar.
 - Binär-Dateien:
 - Sonstige.
 - Verschiedene interne Formate, abhängig von Verwendung, häufig durch sog. Magic Numbers gekennzeichnet, z.B. als ausführbare Dateien.
- Verzeichnisse (Directories) (vgl. 10.2).
- Zeichenorientierte Spezialdateien (Character Special Files)
 - Repräsentierung serieller E/A-Geräte, z.B. Terminals (z.B. Linux: „/dev/ttys0“), Drucker, Netzwerk.
- Blockorientierte Spezialdateien (Block Special Files)
 - Repräsentierung blockorientierter Hintergrundspeichermedien, z.B. Platte (z.B. Linux: „/dev/sda1“), CDROM.

Format einer ausführbaren Datei (a.out, einfacher als ELF):



open()

```
#include <fcntl.h>
int open(char *pathname, int mode, int accessmode);
int open(char *pathname, int mode);
```

- **pathname** - Name oder Pfad der zu öffnenden Datei
- **mode** - wie soll die Datei geöffnet werden? (→ **fcntl.h**)
 - **O_RDONLY** nur lesen, **O_WRONLY** nur schreiben
 - **O_RDWR** lesen und schreiben
- Per Bit-ODER ("|") können verschiedene Flags hinzugefügt werden:
 - **O_APPEND** Schreibzugriffe: am Dateiende anhängen
 - **O_CREAT** Datei anlegen, falls noch nicht vorhanden
(in diesem Fall nur Variante *mit mode* erlaubt)
 - **O_EXCL** (mit **O_CREAT**) Fehler, falls Datei schon da
 - **O_TRUNC** löscht Dateiinhalt, falls schon vorhanden
 - **O_NDELAY** erlaubt nicht-blockierende E/A-Operationen
 - **O_SYNC** sofortiges Zurückschreiben bei Schreiboperationen
- **accessmode**: Bitmuster für Zugriffsrechte (bei **O_CREAT**)
- Ergebnis: „Dateideskriptor“-Wert oder -1 bei Fehler

Beispiel: open()

```
int fd1, fd2, fd3;

fd1 = open("test-1", O_RDONLY);
fd2 = open("test-2", O_WRONLY | O_APPEND);
fd3 = open("test-3", O_RDWR | O_CREAT | O_TRUNC, 0640);
```

- Datei **test-1** kann über Dateideskriptor **fd1** gelesen werden
- Datei **test-2** kann über Dateideskriptor **fd2** beschrieben werden. Dabei wird am Ende angehängt.
- Datei **test-3** kann über Dateideskriptor **fd3** geschrieben und gelesen werden.
 - Falls die Datei noch nicht existiert, wird sie angelegt (creat)
 - Falls sie schon existiert, wird der Inhalt gelöscht (trunc)
 - Rechte: **rw- r-- ---** (drei 3-Bit-Gruppen → oktal 0640)

110	100	000
6	4	0

```
#include <unistd.h>
int read(int fd, char *daten, unsigned anzahl);
int write(int fd, char *daten, unsigned anzahl);
int close(int fd);
```

- **read()**
 - liest bis zu **anzahl** Bytes vom Dateideskriptor **fd** in den Hauptspeicher ab Adresse **daten** ein
 - Rückgabewert: Anzahl tatsächlich gelesener Bytes (oder -1 bei Fehler)
- **write()**
 - schreibt (bis zu) **anzahl** Bytes ab Adresse **daten** auf **fd**
 - Rückgabewert: Anzahl tatsächlich geschriebener Bytes (oder -1 bei Fehler)
- **close()**
 - schließt Datei mit Deskriptor **fd** (-1 bei Fehler)

Exkurs: Fehlerbehandlung, perror()



- Viele Systemfunktionen liefern einen Wahrheitswert zurück (0 für „ok“, -1 für „Fehler“)
- Vorsicht, in „C“ ist 0 „falsch“ und nicht-Null „wahr“!
- Der genaue Fehlercode steht in der globalen `int`-Variablen `errno` (dazu: `#include <errno.h>`)
- Die Funktion `perror(char *)` gibt dann eine passende Fehlermeldung mit einem frei wählbaren **Begleittext** aus.
- Beispiel:

```
#include <errno.h>
int main(int argc, char *argv[]) {
    int ergebnis;
    ergebnis = rename(argv[1], argv[2]);
    if (ergebnis != 0) {
        perror("Fehler beim Umbenennen");
        return -1;
    }
    return 0;
}
```

\$ a.out gibts_nicht_irgendwas
Fehler beim Umbenennen: No such file or directory

Beispiel: Kopierprogramm (1)



```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    enum { BUFFSIZE=1000 };
    char buffer[BUFFSIZE];
    int lese_fd, schreib_fd, gelesen, geschrieben;
    if (argc != 3) { printf("Falscher Aufruf\n"); exit(EXIT_FAILURE); }
    lese_fd = open(argv[1], O_RDONLY);
    if (lese_fd < 0) {
        perror("Bei Oeffnen der Eingabedatei");
        exit(EXIT_FAILURE);
    }
    schreib_fd = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT, 0644);
    if (schreib_fd < 0) {
        perror("Bei Oeffnen der Ausgabedatei");
        exit(EXIT_FAILURE);
    }
}
```

*Programm sofort mit
Rückgabewert 1 beenden*

Beispiel: Kopierprogramm (2)

```
/* Fortsetzung ... */  
while (1) {  
    gelesen = read(lese_fd, buffer, BUFFSIZE);  
    if (gelesen == 0) {  
        break;  
    } else if (gelesen < 0) {  
        perror("Lesefehler");  
        break;  
    }  
    geschrieben = write(schreib_fd, buffer, gelesen);  
    if (geschrieben <= 0) {  
        perror("Schreibfehler");  
        exit(EXIT_FAILURE);  
    }  
    close(lese_fd);  
    close(schreib_fd);  
    return gelesen == 0 ? EXIT_SUCCESS : EXIT_FAILURE;  
}
```

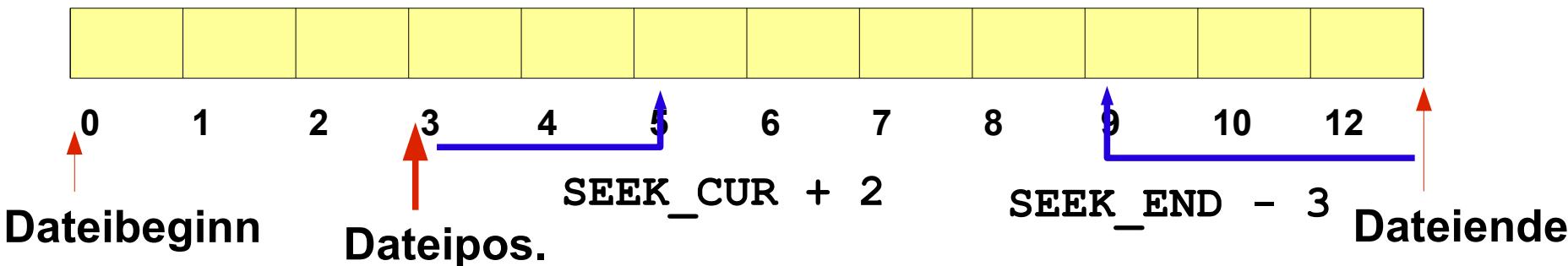
N.B: gelesen, nicht BUFFSIZE!



Direktpositionierung: lseek()

```
#include <unistd.h>
#include <sys/types.h>
int lseek(int fd, off_t offset, int basis);
```

- **lseek()** positioniert die aktuelle Dateiposition von **fd** auf den Wert **offset** gemäß der Einstellung von **basis**
- Werte für "basis" (**offset** darf auch negativ sein):
 - **SEEK_SET**: neue Position wird auf **offset** gesetzt
 - **SEEK_CUR**: neue Position ist aktuelle Position + **offset**
 - **SEEK_END**: neue Position ist Dateiende + **offset**
- Rückgabewert: neue Position (ab Anfang gezählt) oder -1 bei Fehler

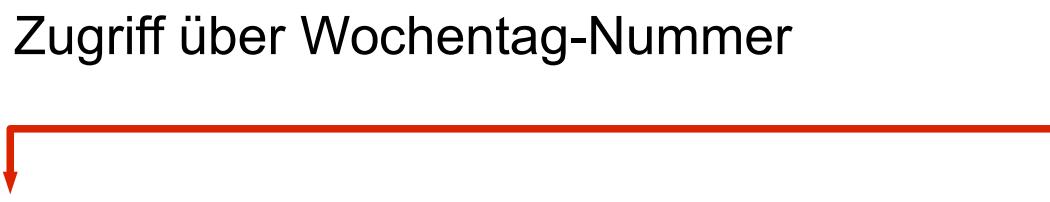


Beispiel: Direktzugriff

- struct-Typ beschreibt Messwerte-Datensatz

```
typedef struct mw {  
    char ableser[20];  
    float temperatur;  
} Messwert;
```

- Ziel:
 - Speichern Messwerte der jeweils letzten 7 Tage (rollierend)
 - Dateiposition aus Wochentag (0=Sonntag, 1=Montag, ...)
 - Direkter Zugriff über Wochentag-Nummer



Satz 0	Satz 1	Satz 2	Satz 3	Satz 4	Satz 5	Satz 6
Meier	Hinz	Kunz	Knüsel	Berger	Huber	Mayr
25.1	110.5	11.0	16.2	17.3	18.0	110.2

speichern() mit Direktzugriff

```
int speichern(int fd, Messwert *pm, int tag) {
    if (lseek(fd, tag*sizeof(Messwert), SEEK_SET) < 0) {
        perror("speichern (lseek)");
        return -1;
    }

    if (write(fd, pm, sizeof(Messwert)) < 0) {
        perror("speichern (write)");
        return -1;
    }
    return 0;
}
```

```
Messwert m;
enum { SONNTAG, MONTAG, DIENSTAG, MITTWOCH, ... };
...
fd = open("messwerte.dat", O_RDWR | O_CREAT, 0644);
err1 = speichern(fd, &m, SONNTAG);
err2 = lesen(fd, &m, MONTAG); /* wie speichern() */
```

stat(): Dateiattribute abfragen

```
int stat(char *file_name, struct stat *buf);
int fstat(int filedeskriptor, struct stat *buf);
```

- liefert Informationen zu der durch `file_name` angegebenen Datei in der angegebenen stat-Struktur (-1 bei Fehler).
- Identisch `fstat` bei bereits geöffneter Datei.

```
struct stat
{
    dev_t          st_dev;      /* Device */
    ino_t          st_ino;      /* INode */
    mode_t         st_mode;    /* Zugriffsrechte */
    nlink_t        st_nlink;   /* Anzahl harter Links */
    uid_t          st_uid;     /* UID des Besitzers */
    gid_t          st_gid;     /* GID des Besitzers */
    dev_t          st_rdev;    /* Typ (wenn INode-Gerät) */
    off_t          st_size;    /* Größe in Bytes */
    unsigned long  st_blksize; /* Blockgröße */
    unsigned long  st_blocks;  /* Allozierte Blocks */
    time_t         st_atime;   /* Letzter Zugriff */
    time_t         st_mtime;   /* Letzte (Inh.) Änderung */
    time_t         st_ctime;   /* Letzte Statusänderung */
};
```

Dateiattribute setzen (Auswahl)

- Zugriffsrechte ändern

- `int chmod(char *Pfad, mode_t Rechte);`
- Ergebnis: 0 für ok, -1 für Fehler

```
if ( chmod("meineDatei.txt", 0600) == 0 ) {  
    /* ok! */  
}
```

- Dateibesitzer / -gruppe ändern

- `int chown(char *path, uid_t owner, gid_t group);`
- Ergebnis: 0 für ok, -1 für Fehler

```
if ( chown("meineDatei.txt", 7, 27) == 0 ) {  
    /* ok! */  
}
```

- **Hinweis:** ID-Nummern für Eigentümer (uid) und Gruppe (gid) stehen z.B. in der Datei /etc/passwd; Angabe von -1: keine Änderung

- Zugriffs- und Modifikationszeitpunkte setzen (vgl. `<utime.h>`)

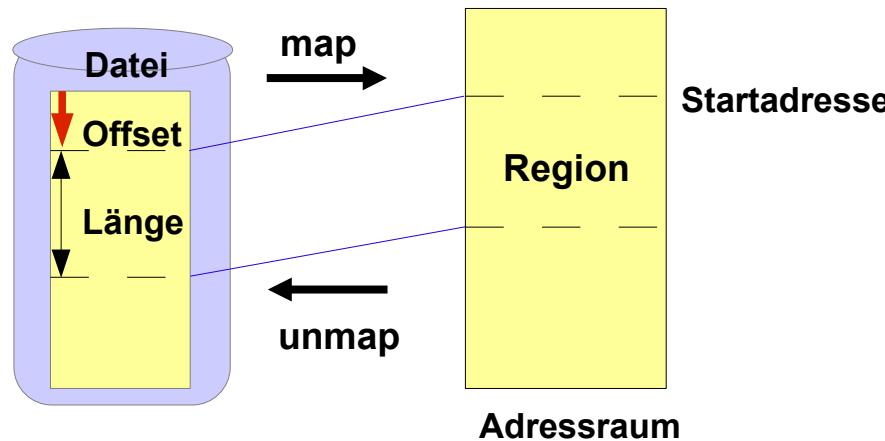
- `int utime(char *path, utimbuf *time);`

10.1.7 Memory-Mapped Files



Def

- Memory-Mapping von Dateien bezeichnet das Einblenden und Ausblenden von Dateien (oder Teil-Fenstern) in den Adressraum eines Prozesses
- Abstrakte Operationen: map und unmap



- Vorteil: Zugriff auf Dateien mit normalen Speicherbefehlen, kein `read` und `write` erforderlich.
- In neueren Systemen in Zusammenhang mit Virtual Memory Management zum Transport der Seiten der Datei realisiert.
- Beispiele: UNIX, Windows NT.

mmap()

```
#include <unistd.h>
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- **length** Bytes von Dateideskriptor **fd** ab Position **offset**
- sollte ab Adresse **start** eingeblendet werden
(**start == 0**: System wählt Adresse selbst)
- **prot** gibt Zugriffsart an (lesen, schreiben, ausführ.)
- **flags**: z.B. **MAP_SHARED**: Änderungen für andere sichtbar
- Ergebnis: Anfangsadresse oder -1 bei Fehler
- **int munmap(void *start, size_t length);**
 - Aufheben des Mappings
- Die ganze Wahrheit: **man mmap**

Beispiel: mmap()

```
...
int main(void) {
    int fd, laenge, i;
    Messwert *pmw; /* s.o.: „Direktzugriff“ */
    fd = open("messwerte.dat", O_RDWR, 0644);
    laenge = lseek(fd, 0, SEEK_END);

    pmw = mmap(0, laenge, PROT_READ|PROT_WRITE,
               MAP_SHARED, fd, 0);
    for (i=0; i < 3; i++) {
        printf("Ableser %s: %f Grad\n",
               pmw[i].ableser, pmw[i].temperatur);
        pmw[i].temperatur = pmw[i].temperatur * 2;
    }
    munmap(pmw, laenge);

    return 0;
}
```

Gliederung

1. Hierarchische Verzeichnissysteme
2. Pfadnamen
3. Verzeichnisoperationen
4. Montierbare Verzeichnisbäume

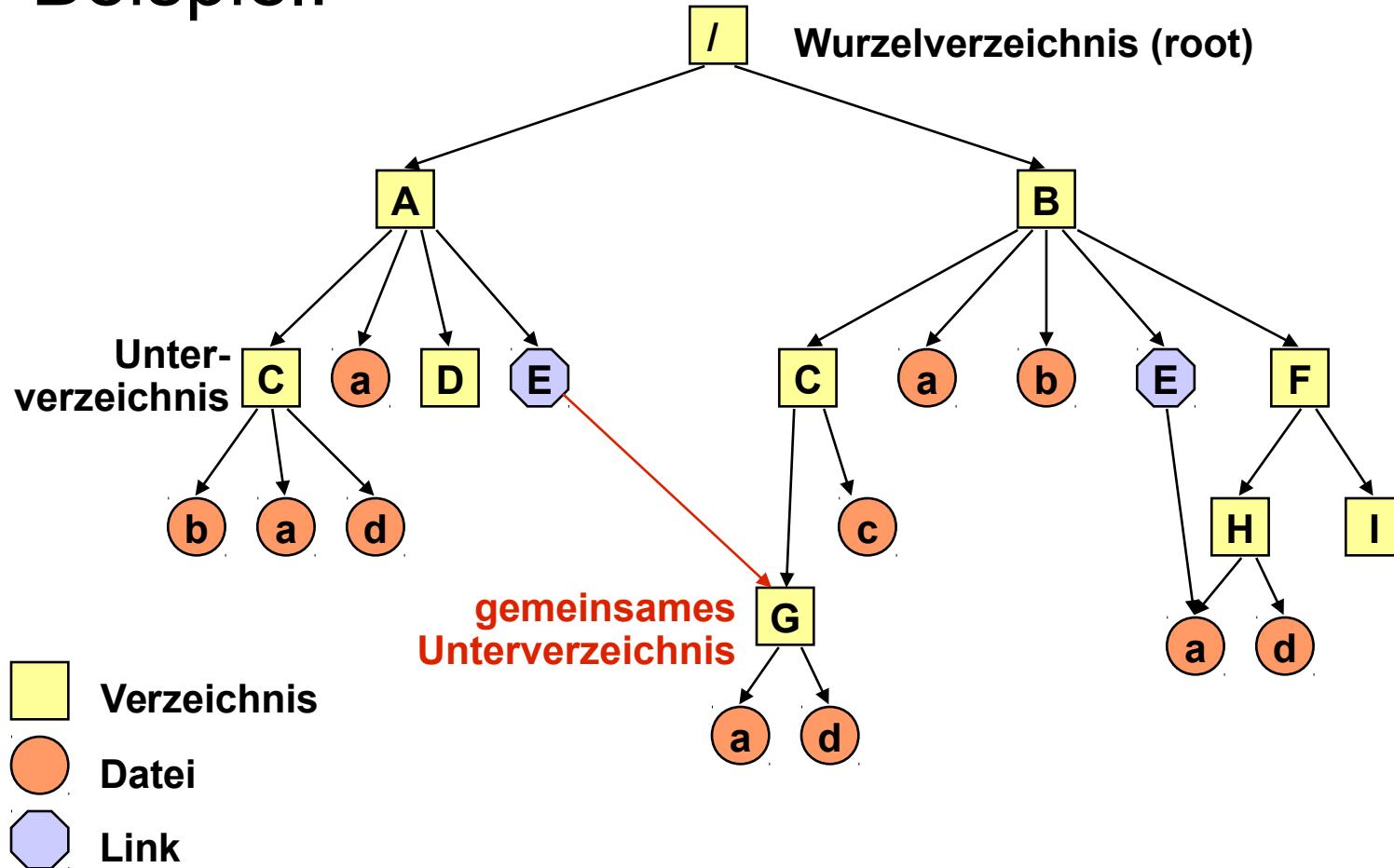
10.2.1 Hierarchische Verzeichnissysteme



Def

- Ein Verzeichnis (Ordner, Directory) dient der Strukturierung der Dateimenge eines Dateisystems und besteht aus einer Menge von Einträgen für Dateien und i.d.R. für weitere Verzeichnisse (Unterverzeichnisse).
- Alternativen:
 - Ein Verzeichnis für alle Dateien sämtlicher Benutzer.
Beispiel: CP/M.
 - Ein Verzeichnis für jeden Benutzer.
 - Hierarchischer Verzeichnisbaum für jeden Benutzer zur Strukturierung seiner Dateimenge (heute Standard).
Beispiel: MS-DOS.
 - Darüber hinaus Einführung zusätzlicher symbolischer Links zur flexiblen gemeinsamen Benutzung von Dateimengen.
Dateisystemstruktur wird damit zu einem azyklischen Graphen.
Beispiel: UNIX.

Beispiel:



Def

- Für hierarchische Verzeichnissysteme dienen Pfadnamen (*Path Names*) der Benennung von Dateien und Verzeichnissen.
- Alternativen:
 - Absolute Pfadnamen:
 - Beispiel UNIX: "/" Name des Wurzelverzeichnisses
 - Benennung vom Wurzelverzeichnis aus.
 - Beispiel UNIX: /usr/egon/uebung/mist.
 - Beispiel MS-DOS: "\\" als Separator.
 - Relative Pfadnamen:
 - Namen werden von einem Arbeitsverzeichnis ausgehend interpretiert (akt. Verzeichnis, Working oder Current Directory).
 - "." bezeichnet häufig das aktuelle Verzeichnis selbst.
 - ".." bezeichnet häufig das Vater-Verzeichnis.

Beispiel: UNIX System V.R4

/	bin	(binaries) wichtige Dienstprogramme, bereits für Hochfahren
	sbin	wichtige Dienstprogramme für Systemverwalter, (single user)
	dev	(devices) Gerätedateien
	etc	Systemverwaltungsprogramme und Systemdateien
	home	Heimatverzeichnisse der Benutzer
	mnt	(mount) für temporär einhängbare Dateisysteme (s.u.)
	opt	(optional) optionale Software-Pakete
	proc	(process) Pseudoverzeichnis des Prozessdateisystems
	tmp	für temporäre Dateien von Anwendungen
	usr	(user) wichtiges Verz. (Dateisystem), in Netzen häufig read-only mountbar
	bin	Dienst- und Anwendungsprogramme
	include	Header-Dateien
	lib	Bibliotheken, Programme, ... (Sammelverzeichnis)
	local	Komponenten, die spezifisch für die Installation sind
	share	nur einmal zu speichernde Information mehrerer Varianten
	ucb	Berkeley-Unix Komponenten
	var	ausgegliederter, veränderlicher Teil des usr-Verzeichnisses
	spool	Sammelverzeichnis für Spooling System, ...
	adm	(administrator) Accounting, Logging, ...

10.2.3 Verzeichnisoperationen



- Typische Verzeichnisoperationen sind
 - Create / Delete / Change Directory,
 - Opendir / Close / Read Directory,
 - Link, Unlink.

UNIX: `mkdir()`, `rmdir()`, `chdir()`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int mkdir(char *pathname, mode_t mode);
int rmdir(char *pathname);
int chdir(char *pathname);
```

- `mkdir()` legt Verzeichnis `pathname` mit Zugriffsrechten `mode` und Einträgen für „..“ und „..“ an (auch mit `mknod` möglich)
- `rmdir()` löscht das (bis auf die Einträge „..“ und „..“ leere!) Verzeichnis `pathname`
- `chdir()` setzt das aktuelle Verzeichnis für den ausführenden Prozess auf `pathname`
- Wieso ist der Verweis-Zähler für ein Verzeichnis mindestens 2?
\$ `mkdir Beispiel`
\$ `ls -l`
drwx----- 2 kaiser profs 4096 Apr 20 15:00 Beispiel

UNIX: **opendir()**, **readdir()**, **closedir()**



```
#include <dirent.h>
#include <sys/types.h>
DIR *opendir(char *pathname);
int closedir(DIR *dir);
struct dirent *readdir(DIR *dir);
```

- **opendir()** öffnet die Verzeichnisdatei **pathname** und gibt einen Zeiger auf **DIR** zurück (**NULL** bei Fehler)
- **closedir()** schließt eine Verzeichnisdatei;
Ergebnis ist 0 (ok) oder -1 (Fehler)
- **readdir()** liefert jeweils nächsten Verzeichniseintrag.
Bei Ende oder Fehler wird der **NULL**-Zeiger geliefert
- Die **struct dirent** enthält ein Feld **char d_name[]** mit dem Namen des betreffenden Verzeichniseintrags

Beispiel: mini-"ls" (Ausgabe)

Ziel: So etwas...

```
$ ./a.out /etc
[ /etc/sysconfig ]
[ /etc/X11 ]
/etc/fstab (1355 Bytes)
/etc/mtab (413 Bytes)
/etc/modules.conf (1049 Bytes)
/etc/csh.cshrc (561 Bytes)
/etc/bashrc (1497 Bytes)
/etc/gnome-vfs-mime-magic (8042 Bytes)
[ /etc/profile.d ]
/etc/csh.login (409 Bytes)
/etc/exports (2 Bytes)
/etc/filesystems (51 Bytes)
/etc/group (601 Bytes)
/etc/host.conf (17 Bytes)
/etc/hosts.allow (161 Bytes)
/etc/hosts.deny (347 Bytes)
...
```

Beispiel: mini-„ls“ (1)

```
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    DIR *dir;
    struct dirent *eintrag;
    struct stat statbuf;
    char pfadpuffer[PATH_MAX], *pfadp;

    if (argc != 2) {
        printf("Aufruf: %s verzeichnis\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    dir = opendir(argv[1]);
    if (dir == NULL) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
```

Beispiel: mini-„ls“ (2)

```
strcpy(pfadpuffer, argv[1]);
strcat(pfadpuffer, "/");
pfadp = pfadpuffer + strlen(pfadpuffer);
while (1) {
    eintrag = readdir(dir);
    if (eintrag == NULL) break;
    if (strcmp(eintrag->d_name, ".") == 0 || 
        strcmp(eintrag->d_name, "..") == 0) continue;
    strcpy(pfadp, eintrag->d_name);

    if (stat(pfadpuffer, &statbuf) == -1) {
        perror(pfadpuffer);
    } else if (S_ISDIR(statbuf.st_mode)) {
        printf("[%s]\n", pfadpuffer);
    } else {
        printf("%s (%ld Bytes)\n", pfadpuffer, statbuf.st_size);
    }
}
closedir(dir);
return EXIT_SUCCESS;
}
```

pfadp

/home/frieda/bsp.c\0

S_ISDIR(m) ist „wahr“, wenn m der st_mode-Wert eines Verzeichnisses ist (siehe: man stat)

UNIX: **link()**, **unlink()**

```
#include <unistd.h>
int link(char *oldpath, char *newpath) ;
int unlink(char *pathname) ;
```

- **link()** legt einen neuen Verzeichniseintrag **newpath** an, der auf dieselbe Datei (inode) wie der bestehende **oldpath** verweist, und erhöht den Referenzzähler im inode. (Keine Datei-Kopie!)
- Entsprechendes Shell-Kommando: **ln alter_eintrag neuer_eintrag**
- **unlink()** erniedrigt den Referenzzähler im zugehörigen inode und löscht den Verzeichniseintrag (sowie die Datei, falls der Referenzzähler auf 0 gefallen ist)
- Verwendet z.B. im Shell-Kommando „**rm**“

```
$ ls -l /usr/bin
-rwxr-xr-x  2 root  root  2003 Jun 23  2002 zdiff
-rwxr-xr-x  3 root  root  3029 Jun 23  2002 zegrep
-rwxr-xr-x  3 root  root  3029 Jun 23  2002 zfgrep
-rwxr-xr-x  1 root  root  1016 Jun 23  2002 zforce
-rwxr-xr-x  3 root  root  3029 Jun 23  2002 zgrep
-rwxr-xr-x  1 root  root  8408 Aug  4   2002 zic2xpm
-rwxr-xr-x  1 root  root  64344 Jun 24  2002 zip
```

- ls -l zeigt Anzahl der Verweise (Links) auf den inode einer Datei
- Option „-i“ zeigt zusätzlich inode-Nummer → so werden verschiedene Verweise auf denselben inode erkennbar:

```
$ cd /usr/bin; ls -l -i z*grep
376494 -rwxr-xr-x 3  root  root  3029 Jun 23  2002 zegrep
376494 -rwxr-xr-x 3  root  root  3029 Jun 23  2002 zfgrep
376494 -rwxr-xr-x 3  root  root  3029 Jun 23  2002 zgrep
377039 -rwxr-xr-x 1  root  root  1180 Jun 24  2002 zipgrep
```

```
#include <unistd.h>
int symlink(char *oldpath, char *newpath);
int readlink(char *path, char *buf, size_t bufsiz);
```

- **symlink()** legt symbolischen Verweis **newpath** an, der auf **oldpath** verweist (symbolischer Link, symlink)
(Shell-Kommando: `ln -s oldpath newpath`)
- **readlink()** liest Verweis aus Symlink **path** in Zeichenvektor **buf** (max. **bufsiz** Zeichen)
- Für beide Funktionen: Ergebnis 0 für „ok“, -1 für Fehler
- Unterschiede zu Hard-Links:
 - Verweis **per Namen**, nicht inode (ändert Referenzzähler nicht)
 - Auflösung zur Laufzeit nötig, evtl. mehrstufig (s.u.)
 - Verweise über Filesystem- und Partitionsgrenzen hinweg möglich (warum geht das mit harten Links nicht?)

Symbolische Links: ls -l

```
$ cd /usr/lib
$ ls -l sendmail
1rwxrwxrwx 1 root      root          16 Apr  6 10:21
sendmail -> ../sbin/sendmail

$ ls -l ../sbin/sendmail
1rwxrwxrwx 1 root      root          21 Feb 14 00:47
../sbin/sendmail -> /etc/alternatives/mta

$ ls -l /etc/alternatives/mta
1rwxrwxrwx 1 root      root          27 Apr  6 10:21
/etc/alternatives/mta -> /usr/sbin/sendmail.sendmail

$ ls -l /usr/sbin/sendmail.sendmail
-rwxr-sr-x 1 root      smmsp         818943 Mar 26 11:19
/usr/sbin/sendmail.sendmail
```

- Verweisziel wird bei symbolischen Links von `ls` angezeigt („->“)
- Typkennzeichen in ls-Ausgabe: „l“ (symLink)
- Hier: Zugriff auf `/usr/lib/sendmail` führt *letztlich* auf `/usr/sbin/sendmail.sendmail`

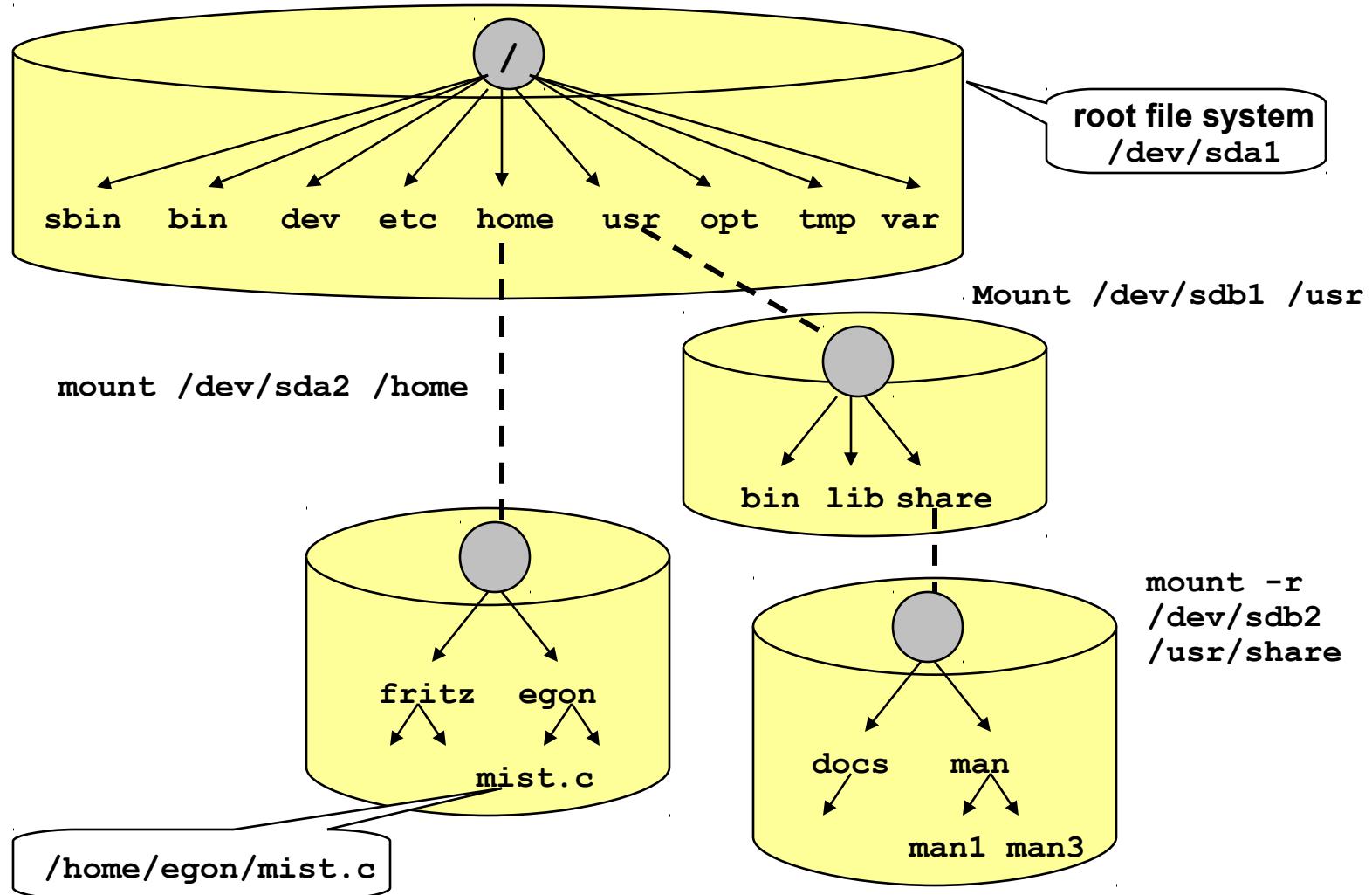
10.2.4 Montierbare Verzeichnisbäume

- Sichtbarer Dateiraum ist i.d.R. durch mehrere Dateisysteme auf mehreren Speichern (u.U. auf mehreren Rechnern) realisiert.
⇒ Mehr oder weniger Auswirkungen auf den Dateinamensraum.
- Alternativen:
 - Dateinamen spiegeln die verschiedenen Speicher wider.
Beispiel: Laufwerksbuchstaben in Windows:
C:\WINDOWS\system, H:\setup.exe
 - Transparenz im Dateinamensraum.
 - Gesamt-Dateiraum aus mehreren Dateisystemen mit jeweils eigenem Verzeichnisbaum durch Montieren (mount) zusammengesetzt.
 - Nach Montieren existiert ein einziger Dateinamensbaum.
 - Beispiel: UNIX, Utilities: mount / umount

Beispiel



10.2.4



Beispiel: /etc/fstab

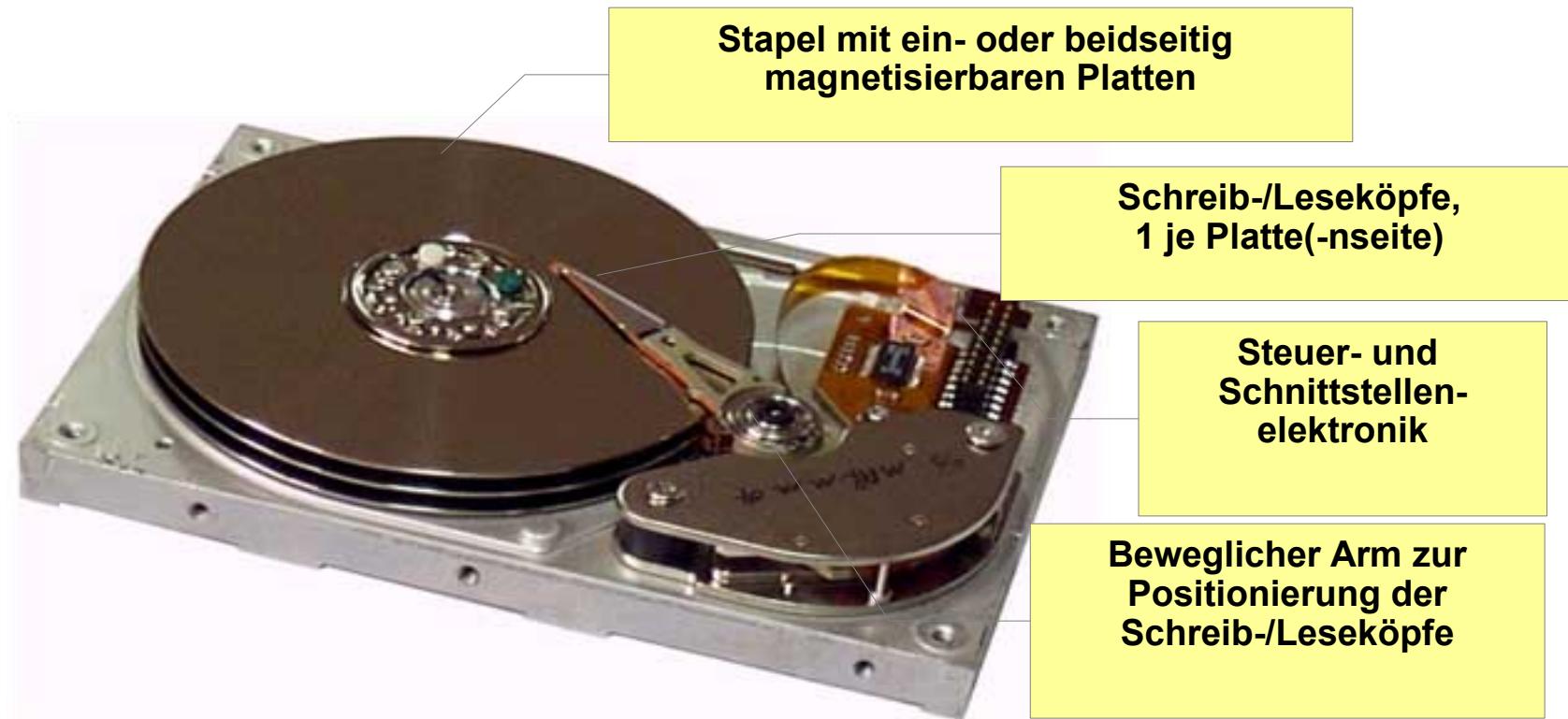
zu mountendes Gerät
Mountpoint
Dateisystem,
Dateisystem-Optionen,
Informationen für Datensicherung (dump),
Rangfolge bei Konsistenzprüfung

```
/dev/sda3  /          ext4 errors=remount-ro  0  1
/dev/sda2  none        swap  sw              0  0
/dev/sr0   /media/cdrom0 udf,iso9660 user,noauto 0  0
/dev/sda1  /win         vfat  defaults        0  0
/dev/sdb1  /local       ext4 defaults        0  0
```

Verbreitete Dateisysteme für Festplatten:

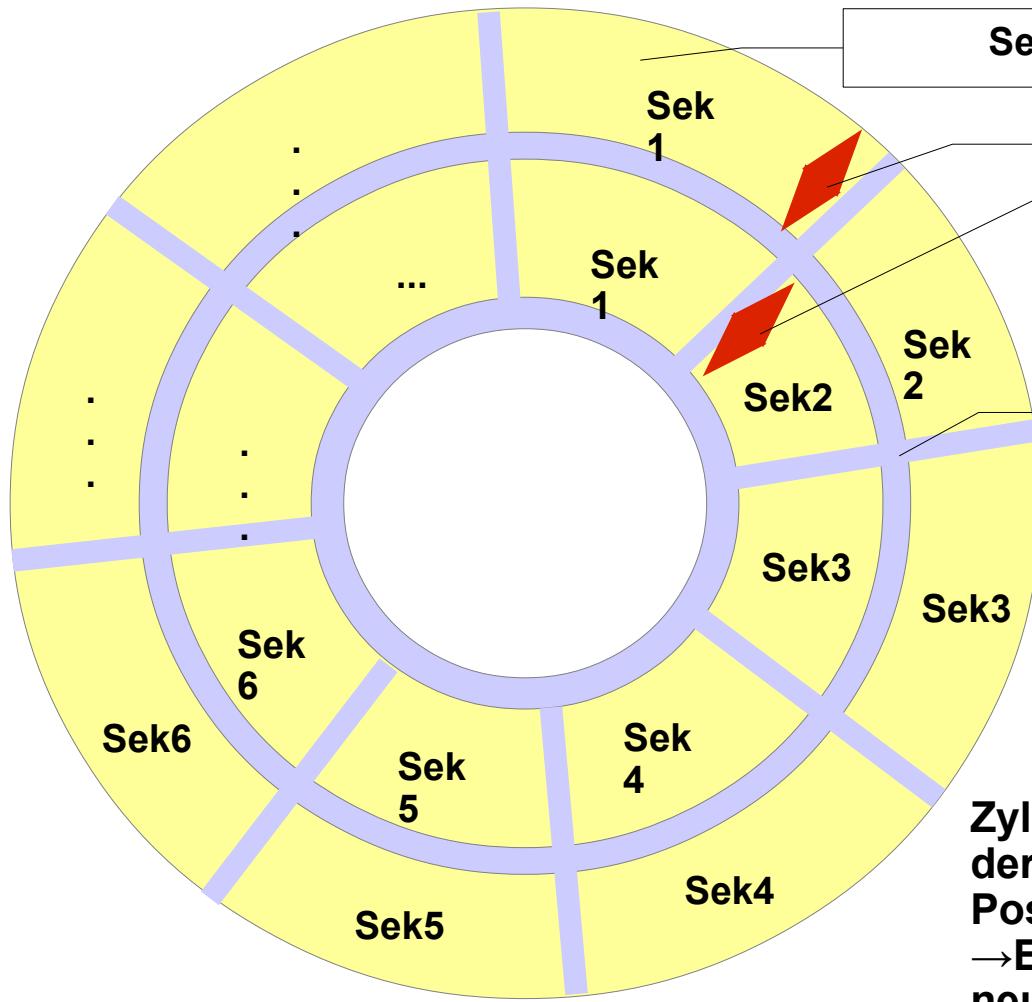
- BSD Fast File System (=UFS, historisch), Linux ext2 (verwandt)
- FAT16 (z.T. historisch), FAT32, NTFS (Windows)
- Linux Journaling File Systems (Logging-basiert):
JFS, XFS, ext3/4 (i.w. ext2+Journaling), ReiserFS
- Linux ab 2.6.28: ext4 (i.w. ext3-Kompat+Extents+größereFS+...)
- Vergleich: http://en.wikipedia.org/wiki/Comparison_of_file_systems
- Im folgenden Sichtweise eines Implementierers
- Gliederung
 1. Technische Gegebenheiten
 2. Implementierung von Dateien
 3. Implementierung von Verzeichnissen
 4. Plattenplatz-Verwaltung
 5. Repräsentierung im Hauptspeicher
 6. Zuverlässigkeit des Dateisystems
 7. Performance des Dateisystems

Speichermedium: Magnetplatte

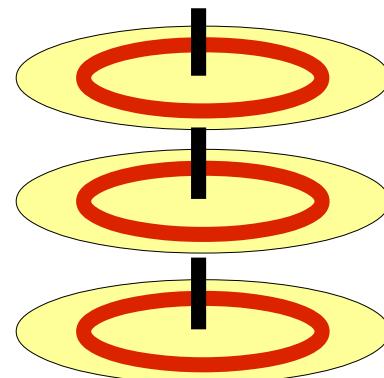


- 5400 bis über 10000 Umdrehungen/Minute
- Datenübertragungsraten: mehrere hundert Mbit/s
- Mittlere Positionierungszeit: ca. 6ms und weniger

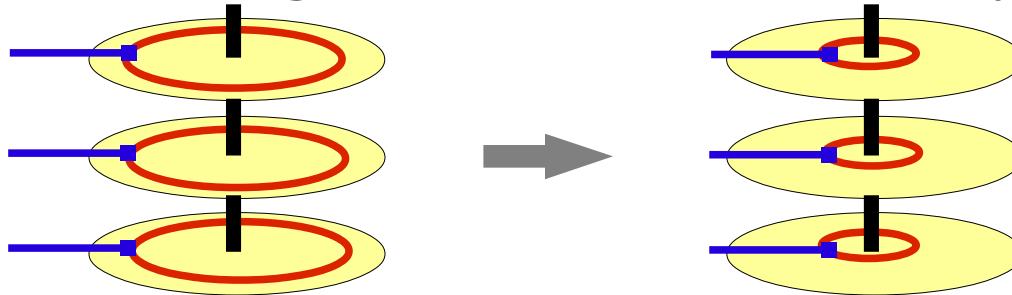
Magnetplatte: Datenorganisation



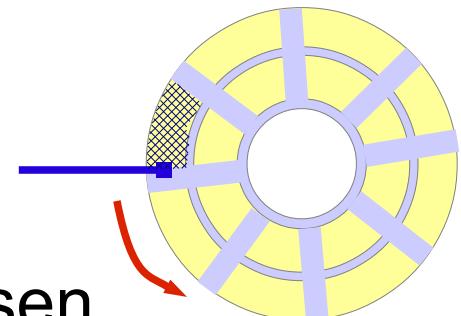
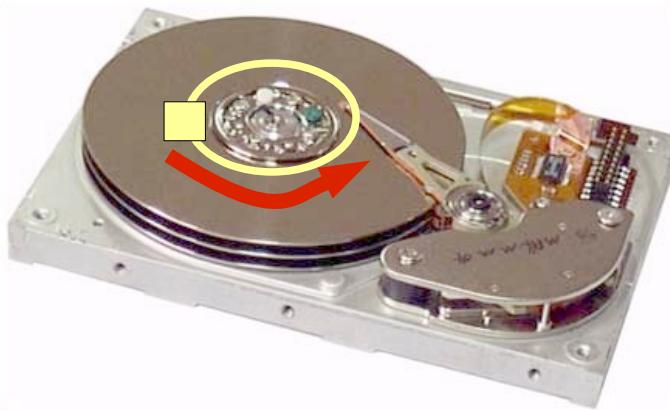
Zylinder (alle Spuren derselben relativen Position zusammen):
→ Erreichbar ohne neue Kopfbewegung



- **Positionierungszeit:** Arm über Ziel-Zylinder bewegen



- **Rotationsverzögerung**, bis gesuchter Sektor unter Kopf ist



- Zeit zur **Datenübertragung** beim Auslesen

- Beobachtungen

- Wahlfreiheit gilt nur bis zur Ebene der Blöcke (kleinste adressierbare Einheit), die Anwenderschnittstelle (s.o.) ermöglicht jedoch Byte-Adressierung
- Die Änderung eines einzelnen Bytes auf der Festplatte erfordert Lesen, Ändern und Zurückschreiben eines ganzen Blockes (gilt auch für Halbleiter-„Festplatten“ (SSD))
- Adressierung der Blöcke erfolgt prinzipiell über Zylinder-, Kopf- und Sektornummern (C/H/S)
- Heutige Festplatten verwenden jedoch logische Blockadressen (LBA), die ursprünglich folgendermaßen gebildet wurden:
$$\text{LBA} = (\text{C} * \text{NumHeads} + \text{H}) * \text{BlocksPerTrack} + \text{S}$$
- Hierdurch können auch Geometriedaten der Festplatte, Eigenschaften wie z.B. eine variable Anzahl `BlocksPerTrack`, etc. abstrahiert werden
→ bei heutigen Festplatten haben diese Parameter -sofern sie überhaupt angegeben werden- keinen Bezug zur Realität mehr
- Dennoch gilt, dass benachbarte Blöcke *mit hoher Wahrscheinlichkeit* ohne Kopfbewegungen „in einem Zug“ schneller gelesen / geschrieben werden können
- Moderne SSD-„Platten“ haben keine mechanischen Komponenten mehr
→ Verzögerungen durch Kopfbewegungen treten nicht auf
→ Datentransfergeschwindigkeit ist i.W. Durch die Schnittstelle und die Eigenheiten des Halbleiter-Speichermediums bestimmt.

Beispiel: Festplattendaten

Hard Disk Drive Device Technical Details

- Hard Disk Model: HITACHI HDS5C1010CLA382
- Disk Family: Deskstar 5K1000
- Form Factor: 3.5"
- Capacity: 1000 GB (1000 x 1 000 000 000 bytes)
- Number Of Disks: 2
- Number Of Heads: 4
- Rotational Speed: 5700 RPM
- Rotation Time: 10.53 ms
- Average Rotational Latency 5.26 ms
- Disk Interface: Serial-ATA/300
- Buffer-Host Max. Rate: 300 MB/seconds
- Buffer Size: 8192 KB
- Drive Ready Time (typical): 10 seconds
- Average Seek Time: 14 ms
- Track To Track Seek Time: 0.8 ms
- Width: 101.6 mm (4.00 inch)
- Depth: 147 mm (5.79 inch)
- Height: 26.1 mm (1.03 inch)
- Weight: 680 grams (1.50 pounds)
- Required Power For Spinup: 3300 mA
- Power Required (Seek): 7 W
- Power Required (Idle): 5 W
- Power Required (Standby): 2 W
- Manufacturer: Hitachi Global Storage Technologies

Grenze für Roh-Datenrate

Spitzen-Datenrate
(Schnittstellen-Eigenschaft
→ gilt auch für SSD-„Platten“)

Bedingt durch Kopfbewegung
→ Spitzenrate wird nicht dauerhaft
erreicht → Vorteil von SSD

**Will der Benutzer Sektoren, Spuren etc. selbst ansteuern,
Seine Daten in 512-Byte-Blöcke aufteilen müssen usw?
Wohl kaum. Er will ...**

- Mithilfe der in 10.1 beschriebenen Funktionen Dateien und Verzeichnisse bearbeiten und verwalten
- optimale **Hardwarenutzung**
(... natürlich ohne eigene Hardware-Kenntnisse)
- **einheitlichen Zugang** zu *vielen* (verschiedenen) Speichergerätearten (standardisierte Schnittstelle)

- Hauptproblem:

Verwaltung der Plattenblöcke und ihrer Zugehörigkeit zu einer Datei.

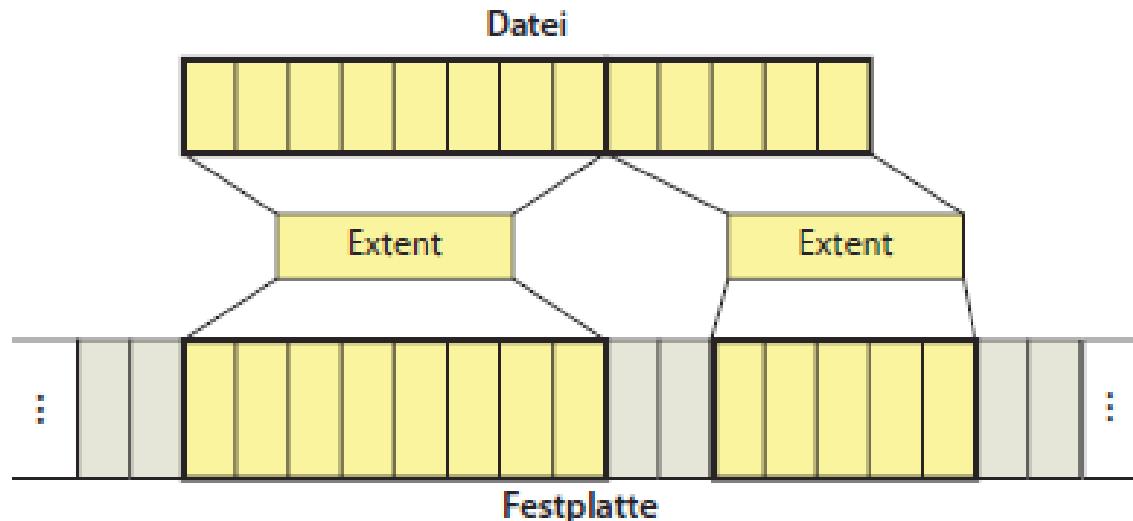
- Alternativen:

- Kontinuierliche Allokation.
- Allokation mittels einer verketteten Liste.
- Allokation mittels einer verketteten Liste und einem Index.
- Allokation mittels Index Nodes.

Datei 1	Datei 2	D.3	Datei 4	D.5	D.6
---------	---------	-----	---------	-----	-----

- Jeder Datei wird eine Menge zusammenhängender Plattenblöcke zugeordnet (beim Anlegen *reserviert*).
- Vorteile:
 - Einfach zu implementieren (nur die Adresse des ersten Blocks ist zu speichern).
 - Sehr gute Performance beim Lesen und Schreiben der Datei (minimale Kopfbewegungen).
- Nachteile:
 - Max. Größe der Datei muss zum Erzeugungszeitpunkt bekannt sein.
 - Externe Fragmentierung möglich.
 - Verdichtung extrem aufwendig.
- Anwendbarkeit:
 - Echtzeit-Anwendungen (Contiguous Files)
 - Write-Once Dateisysteme (CD/DVD, Logs, Backups, Versionierung).

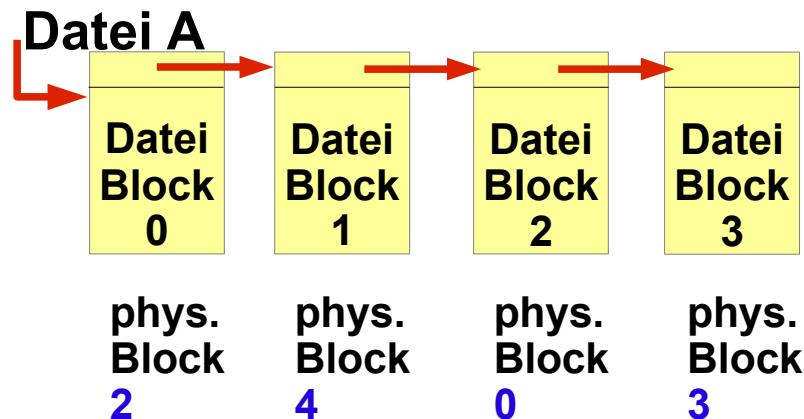
- Extent: Stück einer Datei wird als Folge zusammenhängender Plattenblöcke gespeichert
- Datei besteht aus Folge von Extents
- Vorteile kontinuierlicher Allokation bleiben i.w. erhalten
- Nachteile werden z.T. durch persistente Präallokation umgangen



www.heise.de

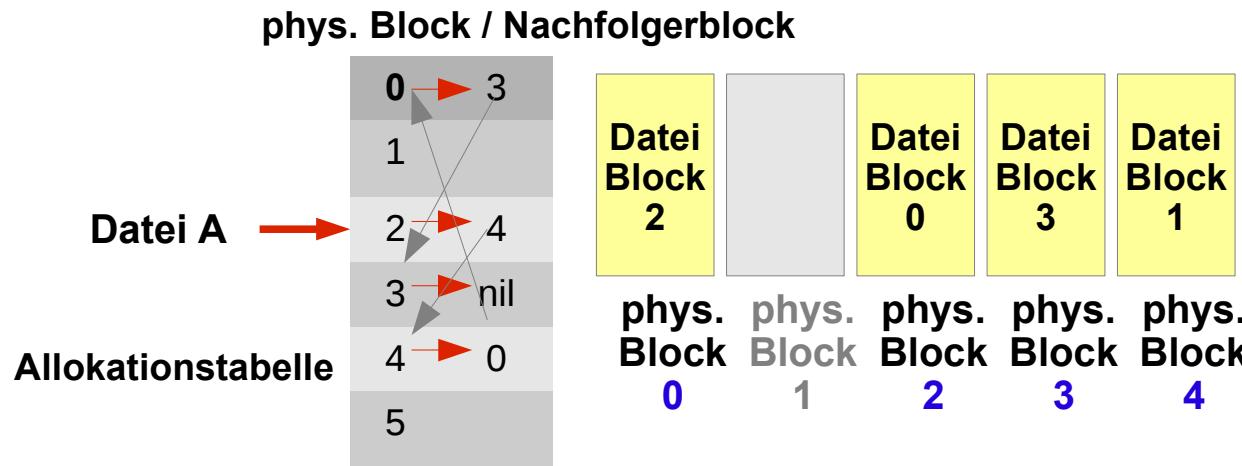
Allokation mittels verketteter Liste

- Datei: Speicherblöcke durch Verweise miteinander verkettet
- Jeder Block hat einen **Verweis auf Nachfolger-Block**.
- Verweis z.B. direkt am Beginn jedes Speicherblocks
- Verzeichniseintrag verweist auf ersten Block der Datei



- Vor-/Nachteile:
 - + keine externe Fragmentierung
 - Wahlfreier Zugriff sehr langsam
 - Es steht nicht der gesamte Datenblock für Daten zur Verfügung

- Speicherung der Verkettungsinformation in einer separaten Tabelle (Index oder File Allocation Table = FAT).

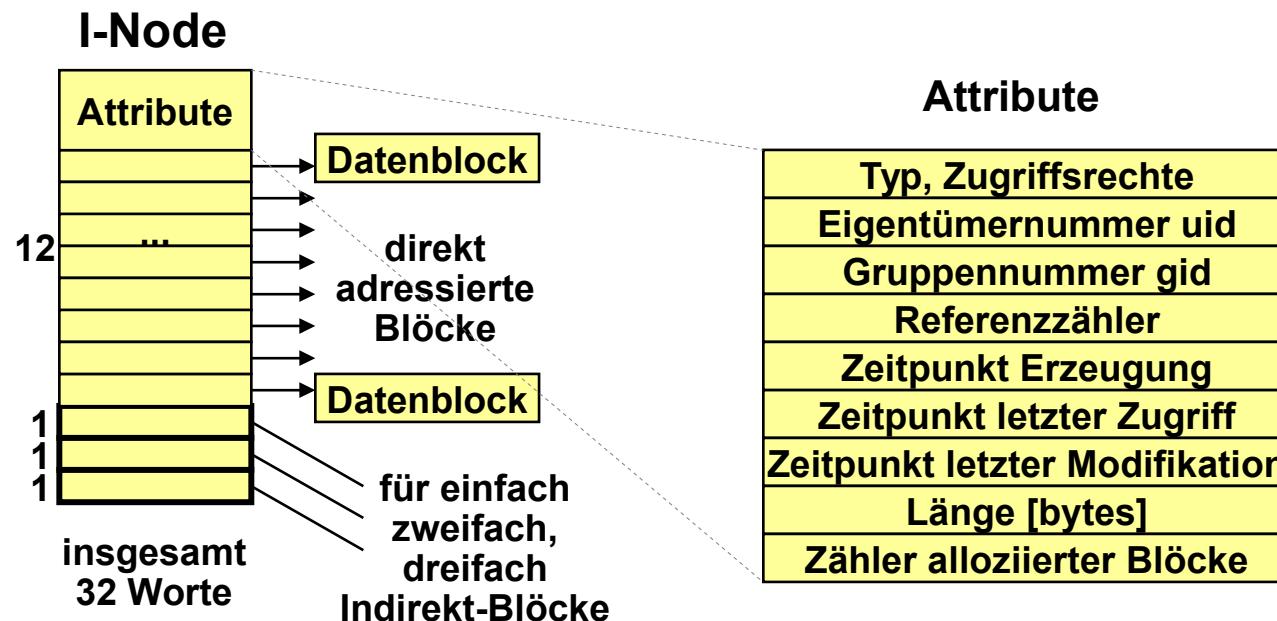


- Vorteile:
 - Gesamter Datenblock steht für Daten zur Verfügung.
 - Akzeptable Performance bei direktem Zugriff, da Index im Arbeitsspeicher gehalten werden kann.
- Nachteile:
 - Gesamte Tabelle muss im Arbeitsspeicher gehalten werden. Kann bei großer Platte sehr speicherplatzaufwändig sein.
 - Beispiel: MS-DOS FAT File System.

10.3.1

Def

- Ein I-Node (UNIX: Inode) oder Index Node ist ein Dateikontrollblock
 - enthält neben Attributen der Datei kleine Tabelle mit Adressen von zugeordneten Plattenblöcken.
 - Ursprung Beispiel: BSD UNIX Fast File System (ufs)

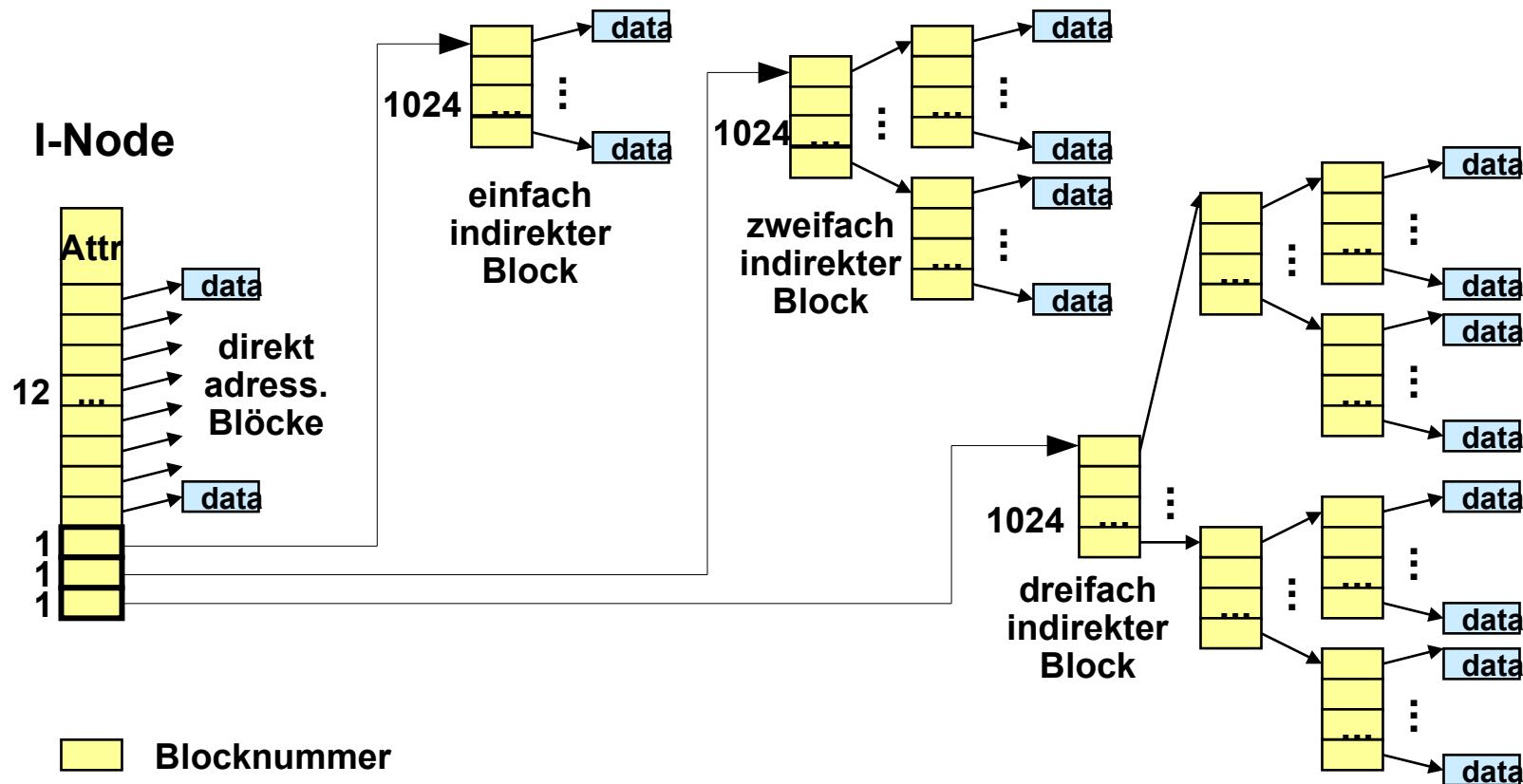


Allokation mittels Index Nodes (2)



10.3.1

- Nutzung von Indirekt-Blöcken



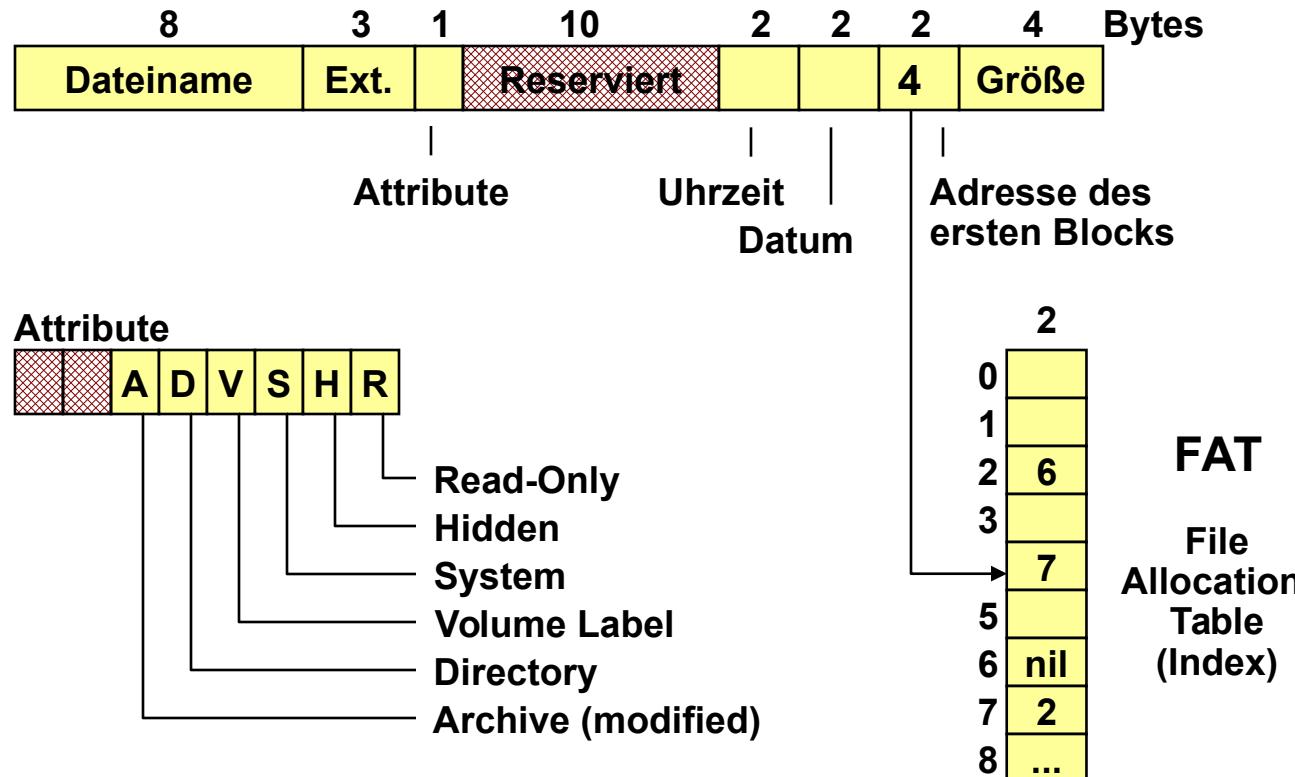
- Logische Blocknummern einer Datei
 - fortlaufend vergeben
 - beginnend bei den direkt adressierten Blöcken.
- Wenige Blockadressen im Inode selbst gespeichert.
 - ⇒ Nach Öffnen einer Datei und damit verbundenem Einlagern des Inodes in den Hauptspeicher stehen diese Blockadressen unmittelbar zur Verfügung.
 - ⇒ Schneller Zugriff zu kleinen Dateien.
- Für größere bis sehr große Dateien werden nach und nach einfach, zweifach und dreifach indirekte Blöcke zur Speicherung verwendet.
 - ⇒ Sinkende Zugriffsgeschwindigkeit bei wachsender Dateigröße.
- Beispiel: BSD UNIX Fast File System (ufs), ext2, ext3
 - Typische Blockgröße: 4 KB oder 8 KB (4 KB bei ext2 und ext3)
 - Direkt adressierte Blöcke: 12 (= 48 KB oder 96 KB).
 - 1-fach, 2-fach und 3-fach indirekte Blöcke vorgesehen.
 - Indirekte Blöcke (4KB) speichern bis zu 1024 ($=2^{10}$) weitere Verweise
⇒ Maximale Dateigröße:
$$(12 + 2^{10} + 2^{20} + 2^{30}) * 4 \text{ KB} = 48\text{KB} + 4\text{MB} + 4\text{GB} + 4\text{TB} \approx 4\text{TB}$$

- Hauptaufgabe des Verzeichnissystems:
Abbildung der Zeichenketten-Namen von Dateien in Informationen zur Lokalisierung der zugeordneten Plattenblöcke.
- Bei Pfadnamen werden die Teilenamen zwischen Separatoren schrittweise über eine Folge von Verzeichnissen umgewandelt.
- Verzeichniseintrag liefert bei gegebenem Namen (Teilenamen) die Information zum Auffinden der Plattenblöcke:
 - bei kontinuierlicher Allokation: die Plattenadresse der gesamten Datei oder des Unterverzeichnisses.
 - bei Allokation mit verketteter Liste mit und ohne Index: die Plattenadresse des ersten Blocks der Datei oder des Unterverzeichnisses.
 - bei Allokation mit Index Nodes: die Nummer des Inodes der Datei oder des Unterverzeichnisses.

Beispiel: MS-DOS

10.3.2

- Hierarchisches Verzeichnissystem.
- Allokation von Plattenblöcken mittels verketteter Liste und Index.
- Verzeichniseintrag:



- Hierarchisches Verzeichnissystem.
- Allokation von Plattenblöcken mittels Index Nodes.
- Verzeichniseintrag BSD UNIX Fast File System (ufs):

2	2	4	max. 255	padding auf 4	variabel lang
Länge Eintrag	Länge Name	Nummer des Inodes	Dateiname		

- Verzeichniseintrag klassisches UNIX System V (s5):

2	14
Nummer Inode	Dateiname

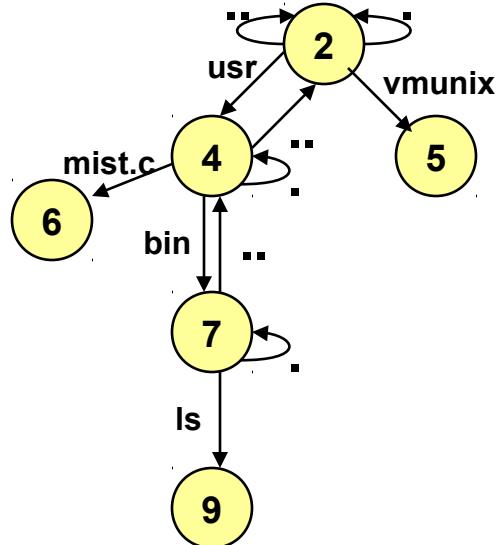
Beispiel: UNIX (2)



10.3.2

- Prinzip der Umsetzung eines Pfadnamens

Logische Dateisystemstruktur



Inode-Liste

aus Leffler et al: Design and Impl.
of the 4.3BSD Operating System

2	root	staff	.	2
	drwxr-xr-x	2
	andere Attr.	usr	4	
	Blockliste	vmunix	5	
		...		
3	frei			
4	root	staff	.	4
	drwxr-xr-x	2
	andere Attr.	bin	7	
	Blockliste	mist.c	6	
	root	source	...	
	-rwxr--r--	...		
	andere Attr.	egon	prof	
	Blockliste	-rw-----		
	root	staff	text	data
	drwxr-xr-x	int main(arg		
	andere Attr.	.	7	
	Blockliste	..	4	
	root	staff	ls	9
	drwxr-xr-x	...		
	andere Attr.	Blockliste		
8	frei			
9	root	staff	text	data
	-rwxr--r--	...		
	andere Attr.	Blockliste		

Datenblöcke
Directory /

Directory /usr

Datei /vmunix
Datei /usr/mist.c

Directory /usr/bin

Datei /usr/bin/ls

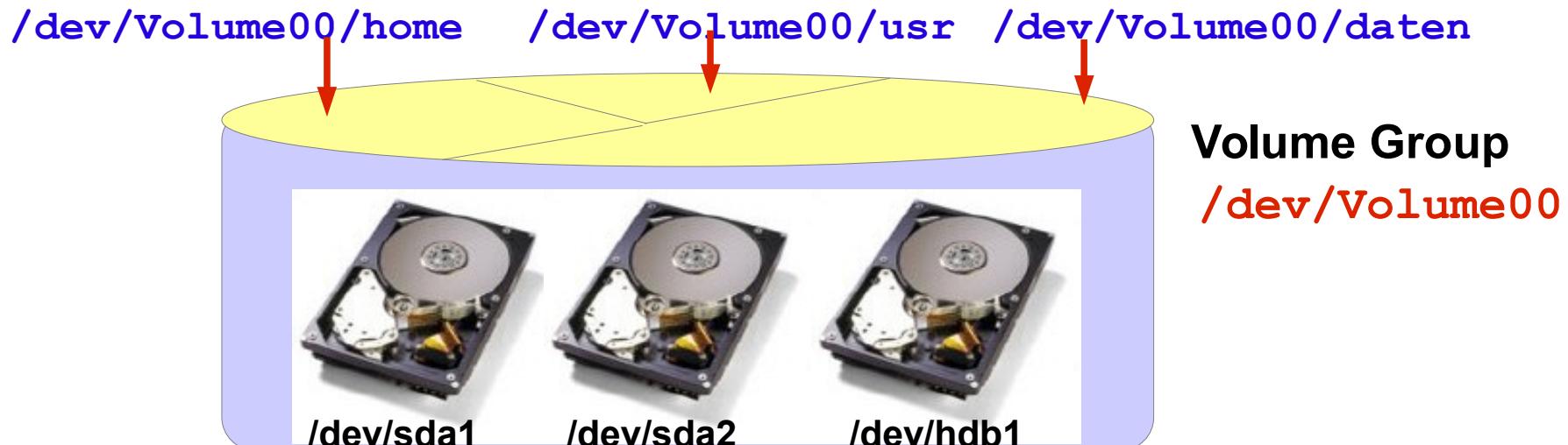
Hauptgesichtspunkte:

1. Logische Laufwerke.
2. Wahl der Blockgröße.
3. Dateisystemstruktur.
4. Quotas (Plattenplatz-Kontingentierung).

10.3.3.1 Logische Laufwerke

Def

- Heutige Betriebssysteme erlauben, Logical Volumes zu erzeugen, die sich über mehrere physische Laufwerke erstrecken.
- Vorteil: sehr große Dateisysteme möglich.
- Beispiel: Linux **Logical Volume Manager** (LVM)
- Physische Speichergeräte (*physical volumes*) werden zu **Laufwerksgruppen** (*volume groups*) zusammengefaßt
- Auf einer Laufwerksgruppe können **logische Laufwerke** eingerichtet (entspricht Partitionierung) und mit einem Dateisystem versehen werden.

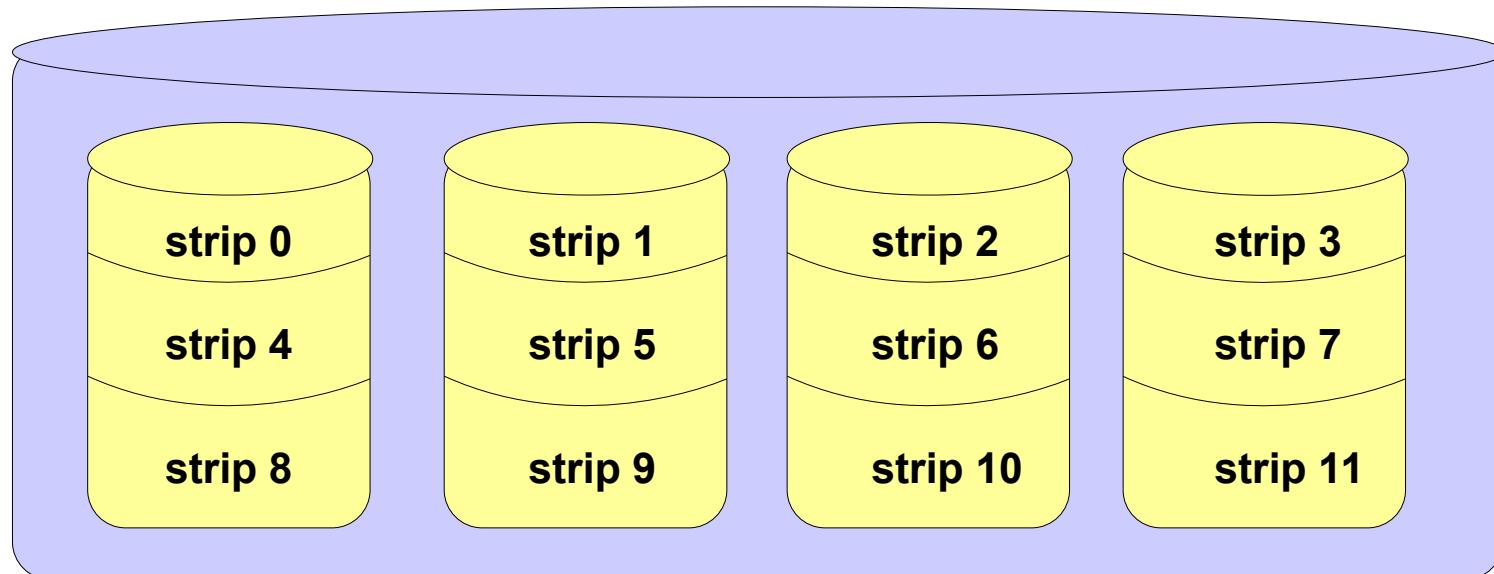


- Im laufenden Betrieb (!) ...
 - kann die Kapazität der Laufwerksgruppe durch **Hinzufügen weiterer physische Volumes** vergrößert werden
 - können **Daten** von alten Platten auf neue **verlagert** und die alten Platten außer Betrieb genommen werden
 - kann logischen Laufwerken mehr **Speicherplatz zugeordnet** werden oder Speicherplatz **entzogen** werden.
- LVM unterstützt „Filesystem Snapshots“
 - Beim Anlegen eines Snapshots wird ein neues logisches Laufwerk angelegt, das den **momentanen Zustand** seines zugehörigen Ursprungs-Laufwerks enthält (eingefrorene Sicht, keine Kopie)
 - Ermöglicht **konsistente Backups** über Snapshot-Laufwerk **trotz weiterlaufenden Betriebs** auf dem ursprünglichen Laufwerk

- Weitere Möglichkeit: RAID:
Redundant Array of Independent (Inexpensive) Disks
- **Viele (preisgünstige) Platten** zusammengeschaltet, sehen für den Rechner **wie eine (sehr große) Platte** aus.
- **Realisierungen:**
 - Hardware-RAID (spezieller Festplatten-Controller)
 - Software-RAID (Betriebssystem verwaltet mehrere angeschlossenen Platten als RAID)
- **Erhöhung der Datensicherheit** durch geschickte redundante Speicherung möglich
- In der Regel Austausch defekter Platten im laufenden Betrieb ohne Unterbrechung (oft auch „hot standby“-Platte)
- Verteilung der Daten auf die einzelnen Platten wird durch RAID level (RAID level 0 ... RAID level 6) definiert

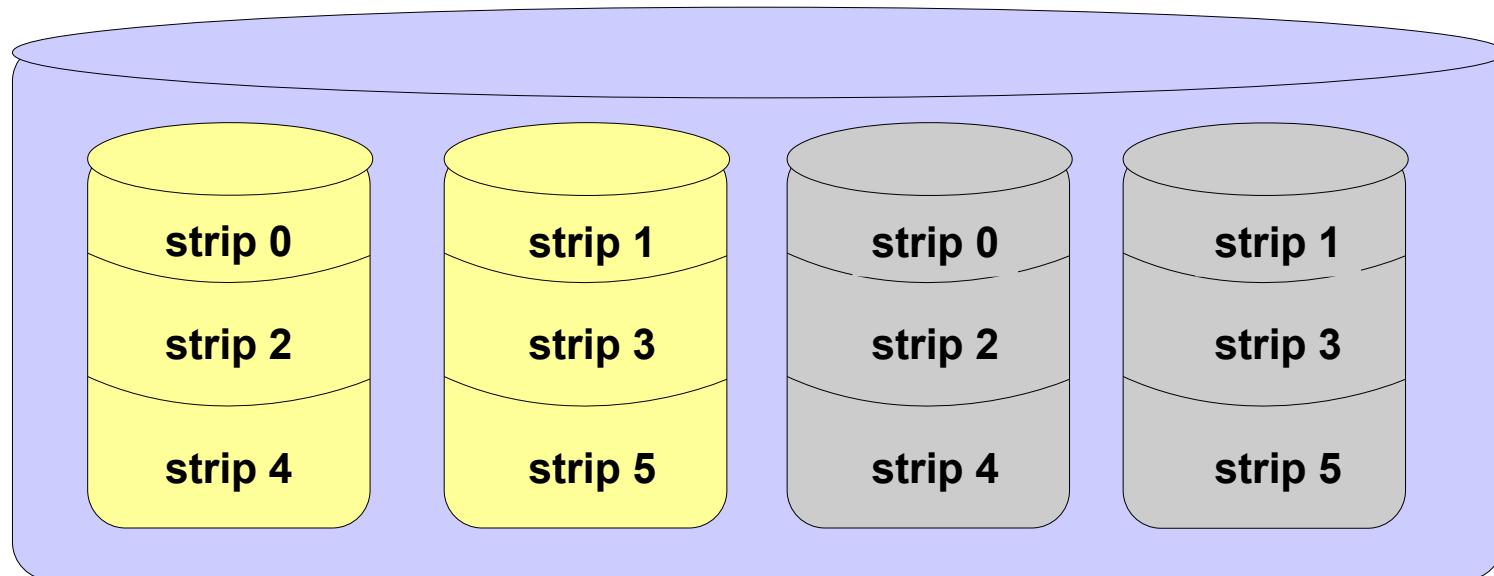
RAID 0 - „striping“

- RAID-Platte wird in „Streifen“ mit k Sektoren eingeteilt
- Streifen werden reihum auf den angeschlossenen Platten abgelegt.
- **keine Redundanz**, damit keine höhere Fehlertoleranz
- **Schneller Zugriff** besonders bei großen Dateien, da Platten parallel arbeiten können
- RAID-Kapazität: Summe der Plattenkapazitäten



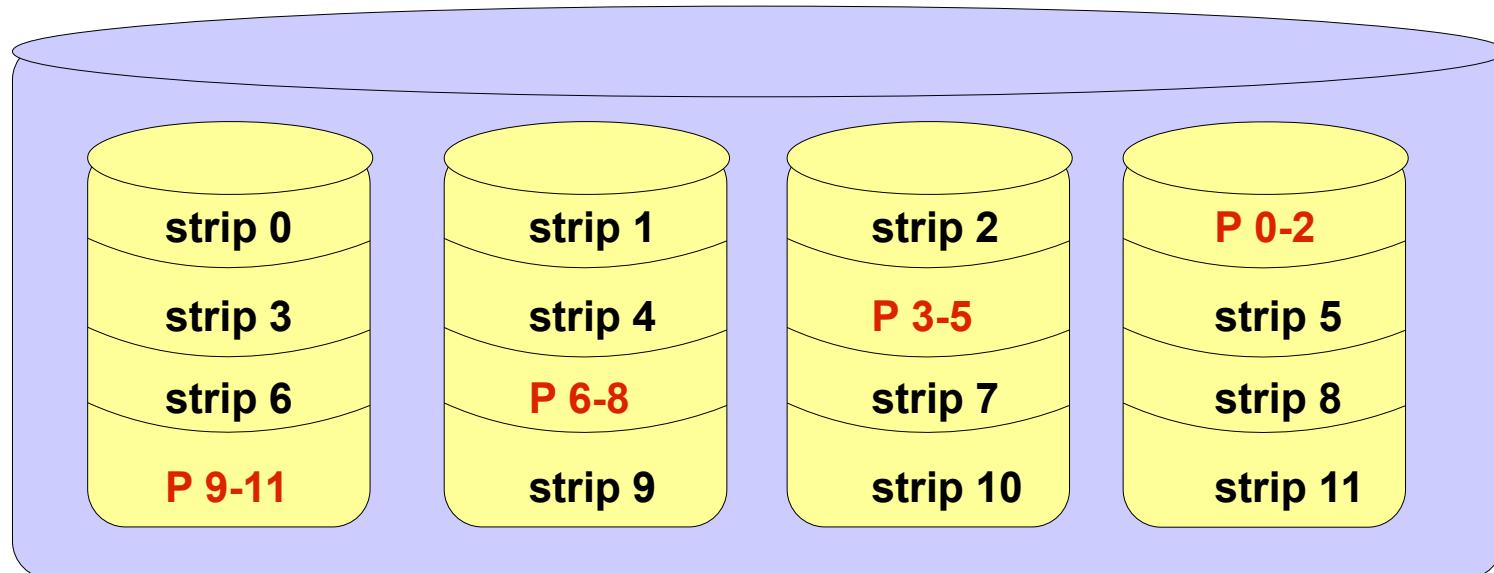
RAID 1 - „mirroring“

- Zu jeder Platte gibt es eine Spiegelplatte gleichen Inhalts
- **Fehlertoleranz:** Wenn eine Platte ausfällt, kann andere sofort einspringen (übernimmt Controller automatisch)
- **Schreiben:** etwas langsamer; **Lesen:** schneller durch Parallelzugriff auf beide zuständigen Platten
- Kapazität: Hälfte der addierten Plattenkapazitäten



- Paritätsinformation auf alle Platten verteilt.
- Beispiel: **P 0-2** enthält XOR-Verknüpfung über die Streifen 0, 1, 2
- XOR Verknüpfung ist „selbstinvers“:
 - Wenn gilt: $P = A \wedge B \wedge C$
 - dann ist: $A = B \wedge C \wedge P$
 - oder: $B = A \wedge C \wedge P$
 - oder: $C = A \wedge B \wedge P$
- Solange maximal eine (beliebige) Platte ausfällt, kann ihr Inhalt aus den Übrigen (im lfd. Betrieb) rekonstruiert werden (wieder per XOR)

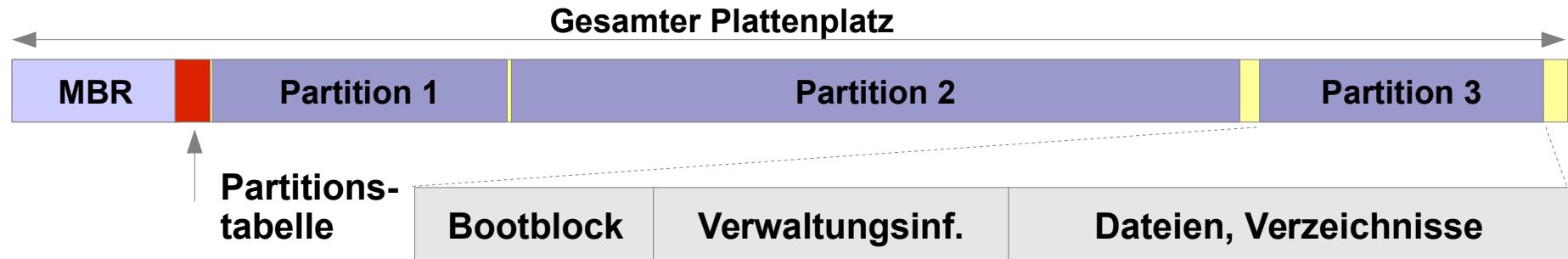
- + Fehlertolerant bei guter Kapazitätsnutzung
- + Leseoperationen schnell
- Schreiben aufwändiger



Klassischer Fall:

Def

- Aufteilung eines physischen Laufwerks in mehrere logische Laufwerke (= Partitionen).
- Partitionierungsinformation (Partition Table) wird auf dem physischen Laufwerk gespeichert.
- Jede Partition kann genau ein Dateisystem aufnehmen.
- Dienstprogramme:
 - MS-DOS: `fdisk`
 - UNIX: `fdisk` (`Linux`), `chpt`, u.a.



- **MBR** (master boot record) enthält ausführbaren Code, der beim Systemstart vom BIOS (basic input/output system) geladen und gestartet wird.
- Dieser Code identifiziert eine **Startpartition**, lädt und startet deren ersten Block (**Bootblock**), der seinerseits ggf. das Laden und Starten des Betriebssystems auslöst.
- Der Bootblock muß das Dateisystem des zu startenden Betriebssystems (zumindest eingeschränkt) verstehen, der Code im MBR kann unabhängig davon sein (z.B. Boot-Menü)
- Die Partitionstabelle beschreibt die Aufteilung der Platte in Partitionen (Anfang, Länge, Typ, ggf. „bootbar“-Flag)

Beispiel: Linux fdisk

Partitionierungstabelle der Festplatte /dev/sda (=erste SATA-Festplatte)

```
$ fdisk /dev/sda
```

```
...
```

```
Befehl (m für Hilfe): print
```

```
Festplatte /dev/sda: 240 Köpfe, 63 Sektoren, 2584 Zylinder  
Einheiten: Zylinder mit 15120 * 512 Bytes
```

Gerät	boot.	Anfang	Ende	Blöcke	Id	Dateisystemtyp
/dev/sda1		1	1163	8792248+	c	Win95 FAT32 (LBA)
/dev/sda2	*	2386	2584	1504440	c	Win95 FAT32 (LBA)
/dev/sda3		1164	1173	75600	83	Linux
/dev/sda4		1174	2385	9162720	f	Win95 Erw. (LBA)
/dev/sda5		1174	2385	9162688+	8e	Linux LVM

```
Partition table entries are not in disk order
```



MBR

sda4 ist eine sogenannte „erweiterte Partition“, die Unterpartitionen (hier: sda5) enthalten kann

10.3.3.2 Wahl der Blockgröße

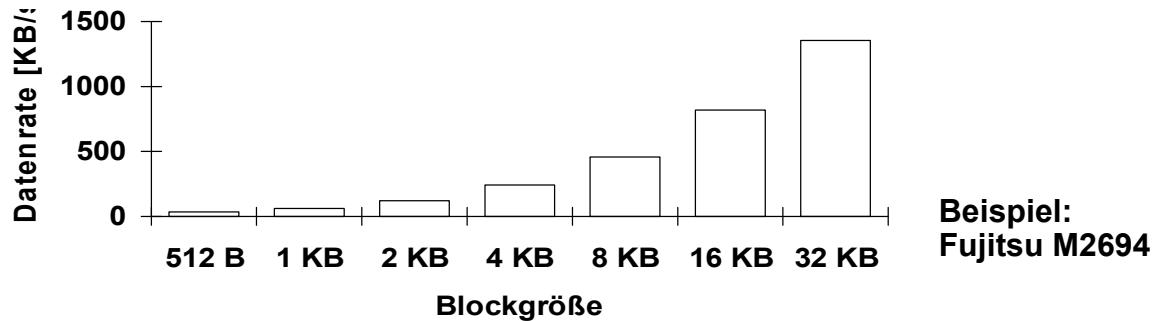


- Fast alle Dateisysteme bilden Dateien aus Plattenblöcken *fester* Länge.
- Plattenblock umfasst Folge von Sektoren mit aufeinanderfolgenden Adressen.
- Problem: Welches ist die optimale Blockgröße?
- Kandidaten aufgrund der Plattenorganisation sind:
 - Sektor 512 B (noch üblich).
 - Spur z.B. 256 KB.
- Untersuchungen an Dateisystemen in UNIX-ähnlichen Systemen zeigen:
 - Die meisten Dateien sind klein (< 10 KB) aber mit wachsender Tendenz.
 - In Hochschul-Umgebung im Mittel 1 KB (Tanenbaum).

- Eine große Allokationseinheit (z.B. Spur) verschwendet daher zuviel Platz.
- Beispielrechnung: Verschwendeter Platz
Daten basieren auf realen Dateien, Quelle [Leffler et al].

Gesamt [MB]	Overhead [%]	Organisation
775.2	0.0	nur Daten, byte-variabel lange Segmente
807.8	4.2	nur Daten, Blockgröße 512 B, int. Fragmentierung
828.7	6.9	Daten und Inodes, UNIX System V, Blockgröße 512 B
866.5	11.8	Daten und Inodes, UNIX System V, Blockgröße 1 KB
948.5	22.4	Daten und Inodes, UNIX System V, Blockgröße 2 KB
1128.3	45.6	Daten und Inodes, UNIX System V, Blockgröße 4 KB

- Eine kleine Allokationseinheit (z.B. Sektor) führt zu schlechter zeitlicher Performance (viele Blöcke = viele Kopfbewegungen).



	Fujitsu M2694	Seagate Barracuda 7200.12
Jahr	1994	2009
Kapazität	1 GB	160-1000 GB
Drehzahl	5400 min ⁻¹	7200 min ⁻¹
Transferrate	4 MB/s	125 MB/s
Positionierzeit (Mittel)	10,0 ms	8,5 ms
Positionierzeit (track-to-track / Max)	2,5 ms / 22,0 ms	?
Latenzzeit (1/2 Umdrehung)	5,6 ms	4,2 ms

- Kompromiss:
 - Wahl einer mittleren Blockgröße, z.B. 4 KB oder 8 KB.
 - Bei Sektorgröße 512 B entspricht ein Block von 4 KB Größe dann 8 aufeinanderfolgenden Sektoren.
 - Für Lesen oder Schreiben eines Blockes wird entsprechende Folge von Sektoren als Einheit gelesen oder geschrieben.
- Ab ca. 2010 für PC-Systeme und Notebooks vermehrt Festplatten mit 4 KB-Sektorgröße (Advanced Format)
 - Ca. 9% Kapazitätsgewinn
 - Kompatibilitätsprobleme (Lösung durch Emulation) können Performance-Nachteile haben

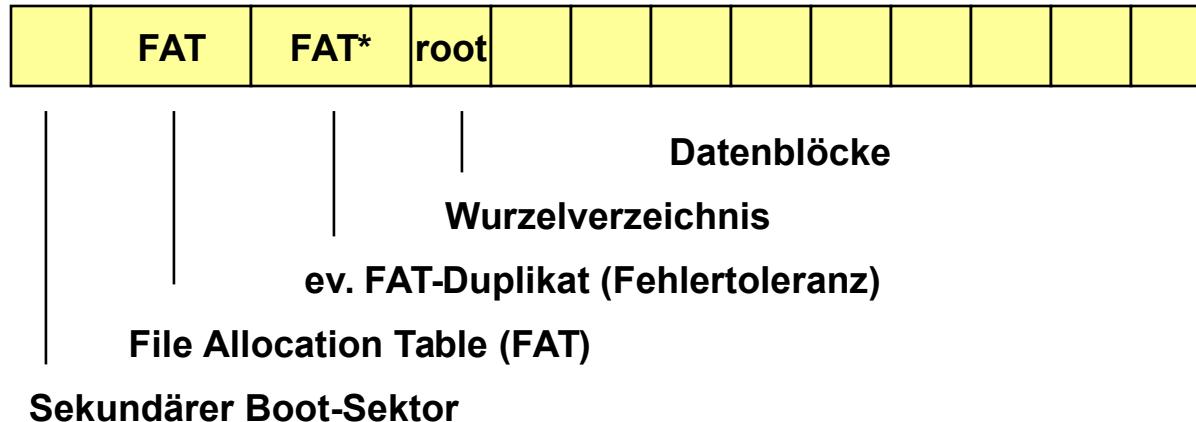


- Beispiel: BSD UNIX Fast File System (heute ähnlich ext2)
 - Einführung zweier Blockgrößen, genannt Block und Fragment als Teil eines Blocks (typ. heute 8 KB / 1 KB).
 - Eine Datei besteht aus ganzen Blöcken (falls nötig) sowie ein oder mehreren Fragmenten am Ende der Datei.
 - ⇒ Transfer großer Dateien wird effizient.
 - ⇒ Speicherplatz für kleine Dateien wird gut genutzt.
Empirisch wurde ein ähnlicher Overhead für ein 4KB/1KB BSD File System beobachtet wie für das 1 KB System V File System.

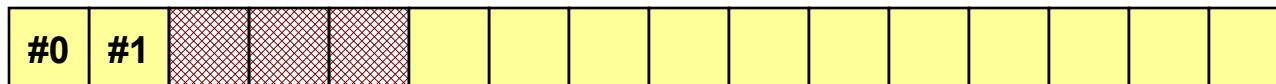
10.3.3.3 Dateisystemstruktur



- Die Struktur eines Dateisystems wird beim Erzeugen auf die Blockmenge eines logischen Laufwerks (Partition) aufgeprägt.
- Dienstprogramme zum Erzeugen:
 - MS-DOS: `format` (nicht zu verwechseln mit dem Formatieren eines Mediums, in MS-DOS low-level Formatierung genannt)
 - UNIX: `mkfs`, (`newfs`)
- Beispiel: MS-DOS



- Beispiel: klass. UNIX System V (s5)



I-Node-
Blöcke

Datenblöcke

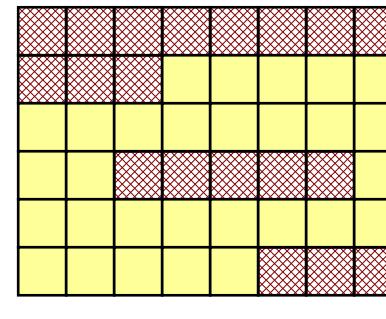
Superblock enthält Layoutbeschreibung:
Blockgröße (512B, 1K, 2k), Anzahl Inodes, Anzahl Blöcke,
Kopf der Freiliste, Kopf der Liste der freien Inodes

nicht Teil des Dateisystems, als Boot-Block reserviert

- Angewendete Methoden:
 - Verkettete Liste.
Beispiele: MS-DOS FAT, UNIX System V
 - Bitmap.
Vorteil: Größere freie Bereiche einfacher erkennbar.
Beispiel: BSD UNIX Fast File System

11
39
4711
2814
9
134
999
21

**Verkettete
Liste**



 **Belegt**  **Frei**

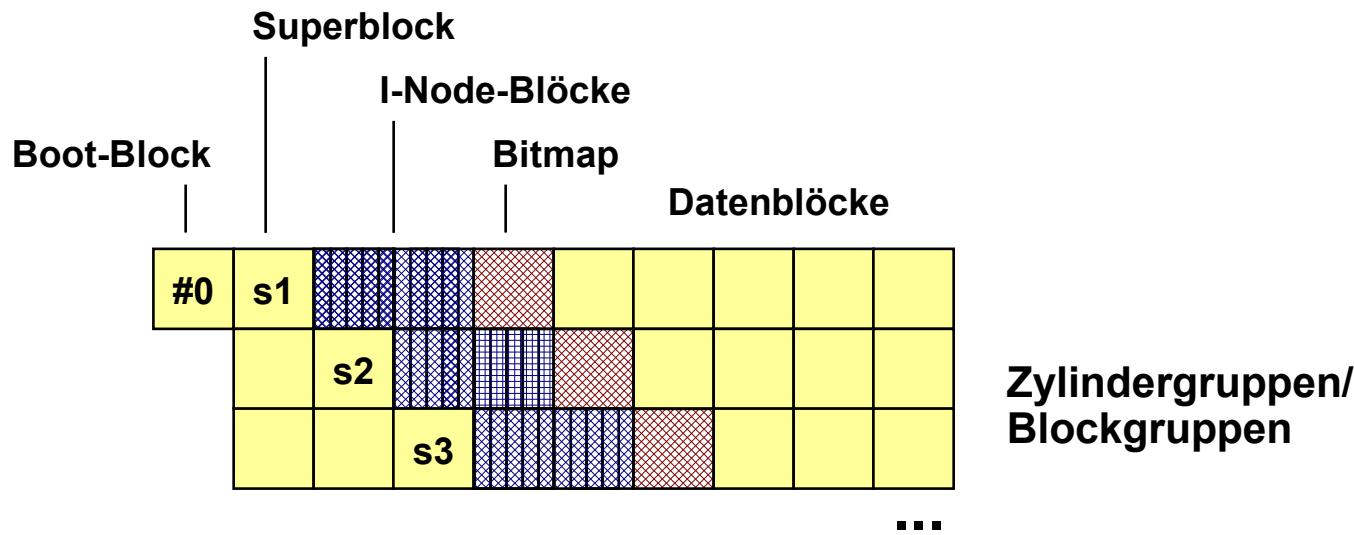
Bitmap

Beispiel: BSD Fast File System (ufs)

10.3.3.3



- Weitere Besonderheiten:
 - Einführung sogenannter Zylindergruppen/Bockgruppen, d.h. Mengen von aufeinanderfolgenden Zylindern/Blöcken innerhalb eines Dateisystems mit jeweils eigenen Verwaltungsstrukturen.
 - I-Node und zugehörige Datenblöcke sollen möglichst dicht beisammen bleiben (Performance).



Redundante Kopien **s1**, **s2**, ... des Superblocks in jeder Zylindergruppe an verschiedenen Stellen (Kopfpositionen) zur Verbesserung der Verfügbarkeit der Layoutinformation auch bei Plattenfehlern.

- Zuordnungstrategie:
 - Normalfall:
Vergrößerung einer Datei erfolgt bei Bedarf Block für Block.
 - Neuere Zuordnungsstrategie
in DEC OSF/1 Advanced File System:
In Folge 1, 2, 3, 6, 12, 32, 32, ... Blöcke von jeweils 8 KB zugewiesen.
⇒ Bei großen Dateien weniger Zuordnungsvorgänge notwendig.

- Ziel:

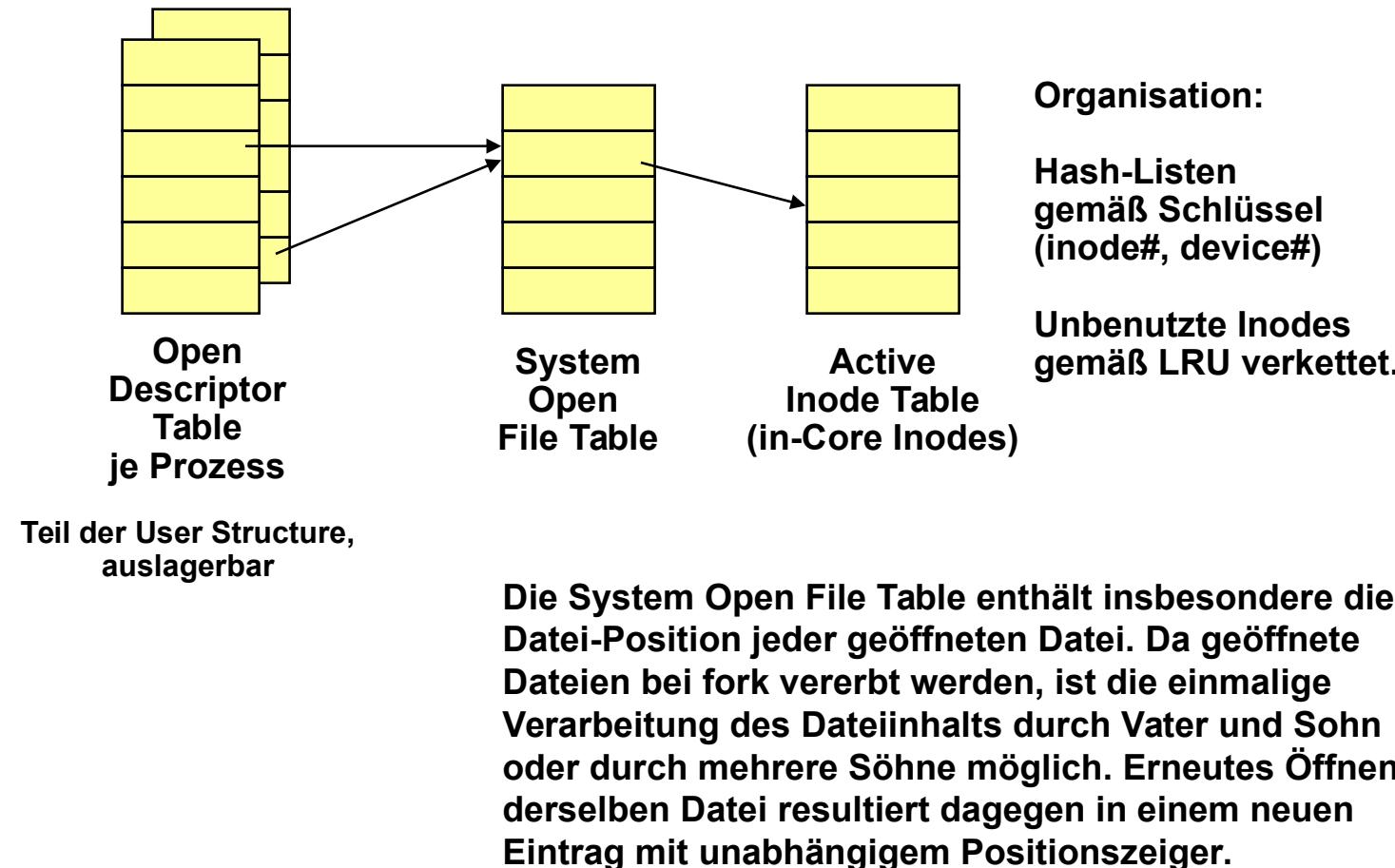
Vermeidung der Monopolisierung von Plattenplatz durch einzelne Benutzer in Mehrbenutzersystemen.

- Systemadministrator kann jedem Benutzer Schranken (Quotas) zuordnen für
 - maximale Anzahl von eigenen Dateien und
 - Anzahl der von diesen benutzten Plattenblöcke.
- Betriebssystem stellt sicher, dass diese Schranken nicht überschritten werden.

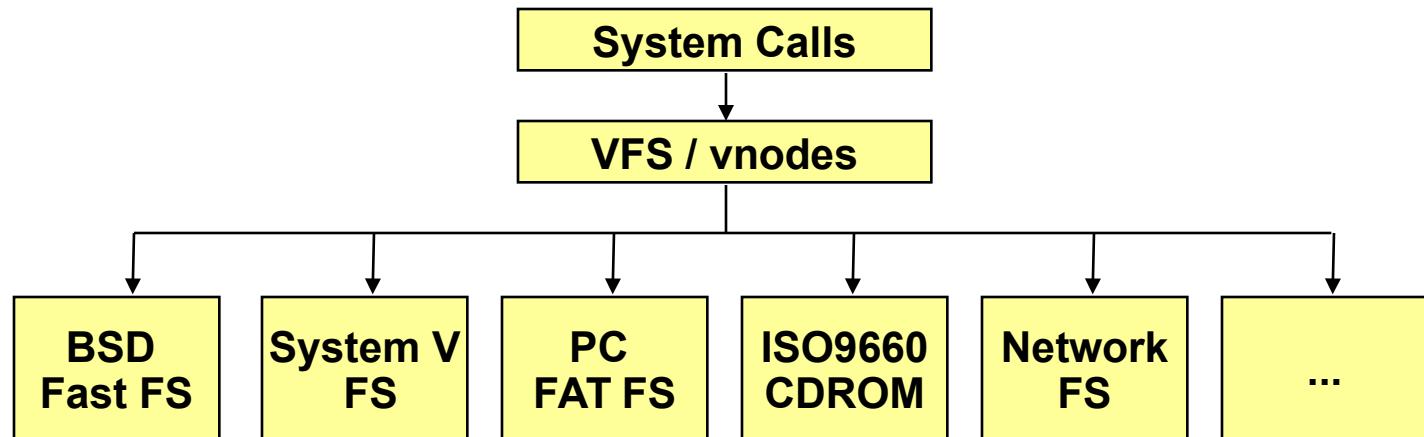
- Je Benutzer und Dateisystem werden verwaltet:
 - Weiche Schranke (Soft Limit) für die Anzahl der benutzten Blöcke (kurzfristige Überschreitung möglich).
 - Harte Schranke (Hard Limit) für die Anzahl der benutzten Blöcke (kann *nicht* überschritten werden).
 - Anzahl der aktuell insgesamt zugeordneten Blöcke.
 - Restanzahl von Warnungen.
Diese werden bei Überschreitung des Soft Limits beim Login beschränkt oft wiederholt, danach ist kein Login mehr möglich.
 - Gleiche Information für die Anzahl der benutzten Dateien (Inodes).

- Bisher wurde die Repräsentierung von Dateien und Verzeichnissen auf dem Hintergrundspeicher betrachtet.
- Geöffnete Dateien besitzen eine Repräsentierung im Arbeitsspeicher
 - mit Informationen des Dateikontrollblocks des Hintergrundspeichers
 - einige zusätzliche Informationen
 - Felder für die Verkettung der Deskriptoren
- Repräsentierung von Dateien im Hauptspeicher wird im folgenden am Beispiel BSD UNIX konkretisiert.

- Überblick:



- Innerhalb des BS-Kerns wurde neue Schnittstelle des Dateisystems eingezogen, das sogenannte vnode-Interface (virtual inode) des VFS (Virtual File System).
- Das vnode-Interface umfasst generische Operationen zum Umgang mit Dateien und Verzeichnissen bzw. Dateisystemen als ganzes.
- Die virtuelle Schnittstelle wird für jeden Dateisystemtyp implementiert.
- Das Interface ist auch auf Pseudo-Dateisysteme anwendbar, wie etwa System V /proc-Prozessdateisystem (jedem aktiven Adressraum entspricht eine Datei) oder RAM-Disk.

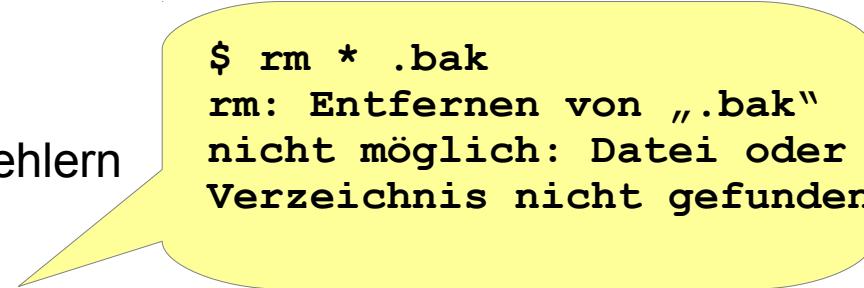


Hauptgesichtspunkte:

- Behandlung fehlerhafter Blöcke.
- Erzeugen und Verwalten von Backups.
- Dateisystemkonsistenz.

- Festplatten haben i.d.R. von Anfang an fehlerhafte Sektoren (z.B. aufgrund ungleichmäßiger Magnetisierung der Oberflächen).
- Fehlerhafte Sektoren werden beim Formatieren der Platte (Aufbau der Sektoren) festgestellt. (Im PC-Umfeld wird das Formatieren auch low-level-Formatierung genannt).
- Verzeichnis der fehlerhaften Sektoren wird Media Defect List genannt.
- Hardware-Lösung:
 - Media Defect List wird auf Platte selbst geführt.
 - Jedem defekten Sektor wird Ersatzsektor (i.d.R. auf einem zusätzlichen Zylinder) zugeordnet, der statt des defekten Sektors benutzt wird.
 - Problem: evtl. unvorhersehbare Kopfbewegungen
- Software-Lösung:
 - Es wird eine Datei konstruiert, die nie gelesen oder geschrieben wird und der alle defekten Blöcke zugeordnet werden.

- Regelmäßige Datensicherung ist wichtig: Schutz vor Datenverlust im Falle von Defekten ...
 - Plattencrash
 - Feuer, Überschwemmung, ...
- ... nicht-redundanter Laufwerke oder Fehlern
 - Sabotage / Viren
 - Benutzer- / Programmierfehler.
- Einheit der Datensicherung sind i.d.R. Dateisysteme.
- Dienstprogramme in UNIX:
dump/restore, backup (System V), tar
- Voller Dump - inkrementeller Dump.
- Dump-Strategie: z.B.
 - Voller Dump wöchentlich.
 - Inkrementeller Dump täglich.
 - Sicherungskopien in geeigneter Entfernung sicher aufbewahren
- Sicherung dauert lange → Problem mit aktiven Dateisystemen
 - Sicherung nachts / an Wochenenden
 - Volume Management (z.B. LVM, vgl. 10.3.3.1)



```
$ rm * .bak
rm: Entfernen von „.bak“
nicht möglich: Datei oder
Verzeichnis nicht gefunden
```

Def

- Konsistenz eines Dateisystems meint Korrektheit der inneren Struktur des Dateisystems
 - d.h. aller mit der Aufprägung der Dateisystemstruktur auf die Blockmenge verbundenen Informationen (Meta-Information des Dateisystems, UNIX: z.B. Superblock, Freiliste oder Bitmap).
- Beispiel einer Konsistenzregel:
 - Jeder Block ist entweder Bestandteil genau einer Datei oder eines Verzeichnisses, oder er ist genau einmal als freier Block bekannt.
- Verletzung der Konsistenz:
 - I.d.R. durch Systemzusammenbruch (z.B. aufgrund eines Stromausfalls) vor Abspeicherung aller modifizierten Blöcke eines Dateisystems.
- Überprüfung der Konsistenz:
 - Betriebssysteme besitzen Hilfsprogramme zur Überprüfung und ev. Wiederherstellung der Konsistenz bei ev. auftretendem Datenverlust.

- UNIX traditionell schwach in Bezug auf Sicherstellung der Konsistenz von Dateisystemen:
 - Dateisystem (z.B. ufs) wird *nicht* in atomaren Schritten von einem konsistenten Zustand in einen neuen konsistenten Zustand überführt. Eine solche Veränderung verlangt i.d.R. mehrere Schreibzugriffe.
 - Modifizierte Datenblöcke bleiben im Pufferspeicher (Block Buffer Cache, vgl. 10.3.6) und werden durch einen Dämonprozess spätestens nach 30 sec zurückgeschrieben.
 - Modifizierte Blöcke mit Meta-Informationen werden zur Verringerung der Gefahr der Inkonsistenz sofort zurückgeschrieben.
 - System Call `sync` existiert zur Einleitung eines sofortigen Zurückschreibens aller veränderten Blöcke (Forced Write).

- Übliche Dienstprogramme zur Überprüfung / Reparatur der Konsistenz eines nicht benutzten Dateisystems bei möglichem Datenverlust: `fsck` (File System Check), auch z.T. `ncheck` (inode check).
- In jüngerer Zeit neue Dateisystemtypen, als Journaled File Systems bezeichnet, die bzgl. der Meta-Informationen des Dateisystems ein Write-Ahead-Logging (analog Datenbanken) durchführen:
 - Konsistenz ist gewährleistet, z.B. bei Systemzusammenbruch aufgrund Stromausfall.
 - Kein `fsck` notwendig (Schneller Restart).
 - Beispiele: IBM AIX 3.2, DEC OSF/1 2.0, NT NTFS, Linux Reiser, Linux ext3.
- Zur Verbesserung der Verfügbarkeit von Daten im Falle von Plattenfehlern werden ebenfalls in jüngerer Zeit z.B. eingesetzt:
 - Spiegelplattenbetrieb (Disk Mirroring, vgl. 10.3.3.1).
 - RAID-Laufwerke (vgl. 10.3.3.1).

UNIX fsck: Blockprüfung

fsck führt verschiedene Konsistenzüberprüfungen durch:

- **Blocküberprüfung**
 - zwei Tabellen mit jeweils einem Zähler je Block
 - anfangs alle Zähler mit 0 initialisiert

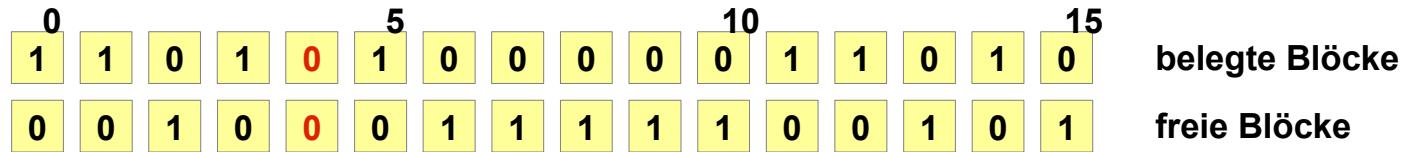
Blocknr	belegte Blöcke	freie Blöcke
0	1 1 0 1 1 5 0 0 0 0 10 1 1 0 1 1 15	0 0 1 0 0 0 1 1 1 1 0 0 1 0 1

- erste Tabelle: wie oft tritt jeder **Block in einer Datei** auf?
 - alle inodes lesen
 - für jeden verwendeten Block Zähler in Tabelle 1 aktualisieren
- zweite Tabelle: **freie Blöcke**
 - Für Blöcke in der Liste / Bitmap der freien Blöcke Zähler in Tabelle 2 aktualisieren
- **Konsistenz:** Für jeden Block ist Zählerstand aus Tab 1 und Tab 2 zusammen „1“

Blockprüfung: Fehler (1)

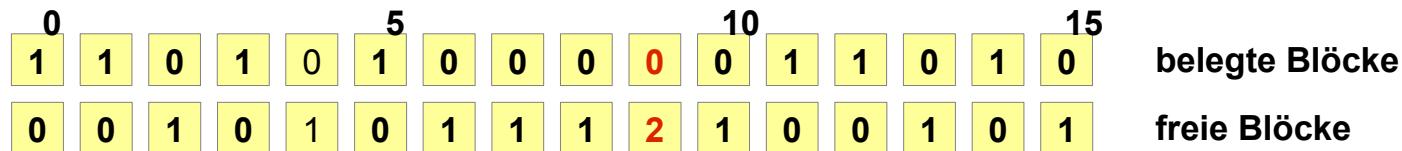
- Fehlender Block: Block 4 ist weder belegt noch frei?

- Maßnahme: Block zu freien Blöcken hinzunehmen



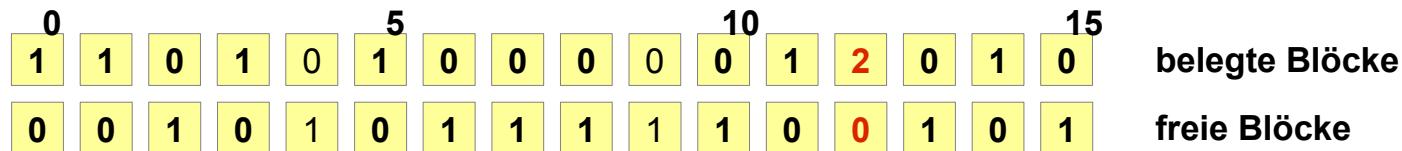
- Doppelter Block in Freiliste (Block 9)

- Maßnahme: Freiliste neu aufbauen



- Doppelter belegter Block (Block 12)

- Maßnahme: Block kopieren, Kopie-Block in eine der beiden betroffenen Dateien statt Block 12 einbauen



Maßnahmen zur Performance-Steigerung:

- Zylindergruppen.
 - Ziel: Vermeiden von weiten Kopfbewegungen.
 - Beispiel: BSD Fast File System (vgl. 10.3.3.3).
- Block Buffer Cache.
 - Ziel: Reduzierung der Anzahl der Plattenzugriffe.
 - Ein Teil des Arbeitsspeichers, (Block) Buffer Cache genannt, wird als Cache für Dateisystemblöcke organisiert.
 - Typische Hitrate: 85 % (4.3BSD UNIX).
 - Modifizierter LRU-Algorithmus zur Auswahl zu verdrängender Blöcke unter Berücksichtigung der Forderungen zur Verringerung der Gefahr von Inkonsistenz
 - Blöcke verbleiben im Cache, auch wenn die entsprechenden Dateien geschlossen sind.

- Dateinamens-Cache.
 - Ziel: Verringerung der Anzahl der Schritte bei der Abbildung von Dateipfadenamen auf Blockadressen.
 - Beispiel: BSD UNIX namei-Cache:
 - `namei()`-Routine zur Umsetzung von Pfadnamen auf inode-Nummern. benötigte vor Einführung von Caching ca. 25% der Zeit eines Prozesses im BS-Kern, auf <10% gesenkt.
 - Cache der n letzten übersetzten Teilnamen:
Typische Hitrate: 70-80 %.
Höchste Bedeutung für Gesamtperformance.
 - Cache des letzten benutzten Directory-Offsets:
Wenn ein Name im selben Verzeichnis gesucht wird, beginnt die Suche am gespeicherten offset und nicht am Anfang des Verzeichnisses (sequentialles Lesen eines Verzeichnisses ist häufig, z.B. durch das ls-Kommando).
Typische Hitrate: 5-15 %.
 - Gesamthitrate von ca. 85 % wird erreicht.

Gliederung

1. Einordnung und Begriffe
2. Angriffe auf die Sicherheit
3. Entwurfsprinzipien
4. Aufgabenbereiche
5. Beispiel: Benutzer-Authentifikation

10.4.1 Einordnung



- Das deutsche Wort Sicherheit beinhaltet:
 - Sicherheit gegen technische Fehler (technische Zuverlässigkeit) bei verschiedenen Betriebsbedingungen:
 - Normalbetrieb (Fehlertolerante Systeme).
 - Unfall, Feuer, Verbrechen (Back-up-Systeme).
 - Naturkatastrophen (Erweiterte Back-up-Lösungen).
 - Kriegsfall oder terroristische Angriffe (?).
 - Hauptziel: Ausschließen einer Gefährdung (Safety).
 - Sicherheit gegen menschliches Versagen wegen:
 - Überforderung.
 - Unaufmerksamkeit.
 - Fahrlässigkeit.
 - Mutwilligkeit.
 - Neben organisatorischen Aspekten häufig ein Problem der Mensch-Maschine-Schnittstelle.
 - Sicherheit gegen Ausspähung (im weiteren behandelt).
 - Sicherheit gegen Manipulation (im weiteren behandelt).

Def

- Sicherheit (Security) bezeichnet das allgemeine Problem, Information nur nach festgelegten Regeln (Policies) berechtigten (autorisierten) Personen lesend oder verändernd zugänglich zu machen.

Aspekte:

- politische
- juristische
- organisatorische
- technische (hier im Vordergrund).
- Privatsphäre (Privacy) als spezielles Sicherheitsproblem: Schutz des Individuums vor dem Missbrauch der über es gespeicherten Information.
- Schutzmechanismen (Protection Mechanisms) bezeichnen Verfahren in Rechensystemen, die dem Schutz von Informationen dienen, um Sicherheit zu erreichen.

- Verletzung von Sicherheit
 - durch Eindringlinge (Intruders).
 - Vorgang als Penetration bezeichnet.
- Passive Eindringlinge versuchen, Dateien zu lesen, ohne dass sie dazu autorisiert sind (Ausspähung).
- Aktive Eindringlinge versuchen nichtautorisierte Veränderungen an gespeicherter Information (Manipulation).

- Kategorien von Eindringlingen:
 - Gelegentliches Ausspionieren durch normale Benutzer.
 - Unterlaufen vorhandener Sicherheitsvorkehrungen durch Experten (persönliche Herausforderung, z.B. Hacker).
 - Betrug oder Diebstahl mit Rechnern als Hilfsmitteln (kriminelles Verhalten).
 - Kommerzielle oder militärische Spionage.
- Eindringlinge können außer Benutzern auch sein:
 - Betreiber.
 - Wartungspersonal.
 - Hersteller
 - Behörden (NSA)

10.4.3 Entwurfsprinzipien



- Erste systematische Behandlung durch Saltzer und Schroeder (1975) im Rahmen des MULTICS-Projekts am MIT.
- Regeln:
 - Der Entwurf des Systems sollte öffentlich zugänglich sein. Sicherheit, die auf Geheimhaltung vor möglichen Eindringlingen basiert, ist nie lange gewährleistet.
 - Jeglicher Zugriff muss explizit erlaubt werden. Default-Fall: kein Zugriff.
 - Ein einmal gewährter Zugriff muss wiederholt überprüft werden (Revalidierung).
 - Befristung oder Widerruf (Revocation) eines Rechts sollte möglich sein.
 - Jeder Prozess sollte soweit Zugriffsrechte wie nötig haben (Least Privilege Principle oder Need-to-Know Principle).

- Der Schutzmechanismus sollte einfach (\Rightarrow verständlich), einheitlich und in den untersten Ebenen des Systems verankert sein. Sicherheit nachträglich in ein existierendes Betriebssystem einzubauen, gilt als aussichtslos.
- Trennung von Schutzstrategie (Policy) und Mechanismus.
 - Strategie legt fest, wessen Daten vor wem geschützt werden sollen
 - Mechanismus legt fest, wie (mit welchen Mitteln) diese Festlegungen durchgesetzt werden.
- Akzeptanz der Mechanismen durch die Benutzer muss gegeben sein.

10.4.4 Aufgabenbereiche



- Authentifikation:
Authentifikation (oder Authentifizierung) bedeutet, Gewissheit zu erlangen (Validierung) über die Identität eines Subjekts (vgl. 10.4.5).
- Autorisation:
Autorisation beinhaltet die Verwaltung von Berechtigungen für Subjekte zur Durchführung legaler Aktionen (vgl. 10.5).
- Kryptographie:
Kryptographie beinhaltet die Verschlüsselung gespeicherter und übertragener Informationen einschließlich entsprechender Schlüsselverwaltungssysteme (hier nicht behandelt, vgl. andere Lehrveranstaltungen).
Aufgrund der Vernetzung der Systeme und der wirtschaftlichen Nutzung der Netze haben Verschlüsselungsverfahren eine hohe Bedeutung erlangt (Beachte auch die politische Diskussion und Stellungnahmen dazu, vgl. Informatik & Gesellschaft).

10.4.5 Beispiel: Benutzer-Authentifikation

- Authentifikation bedeutet, Gewissheit zu erlangen über eine vorgebene Identität eines Subjekts.
- Benutzer-Authentifikation beinhaltet, dass das Betriebssystem eine vorgegebene Identität eines Benutzers überprüfen kann, wenn er Zugang zum System verlangt.
- Prinzipiell sollte sich ein Benutzer auch über die Identität eines Rechensystems Gewissheit verschaffen können (gegenseitige Authentifikation).

- Passwörte.
 - Regeln beachten für brauchbare Passwörte.
 - Passwörte nicht in unverschlüsselter Form speichern.
 - Regelmäßiges Ändern von Passworten, z.T. über Alterungsmechanismus erzwungen.
 - Extreme Form: Einmal-Passwörte gemäß einer Liste von vereinbarten Passworten (Beispiel: TANs).
- Herausforderung / Antwort - Spiel (Challenge / Response).
- Physische/Biometrische Identifikation.
 - Chipkarte mit Password (PIN) oder SmartCards.
 - Bestimmung schwer fälschbarer physischer Merkmale:
 - Fingerabdrücke.
 - Sprachprobe.
 - Iris-Abbildung.
 - Unterschriftenanalyse.
 - Blutprobe (Akzeptanzproblem).

- Örtliche und zeitliche Einschränkungen
 - Benutzern werden Terminalleitungen und mögliche Zeiten zugeordnet, über die und während der ein Benutzer Zugang zum System erhalten kann.
- Rückruf
 - bei Einwahl über Modem mit vorab fest vereinbarter Telefonnummer.
- Fallen
 - Aufstellen von "Fallen" im System, mit denen Eindringlinge auf frischer Tat ertappt werden können.

- Mehrbenutzersystem
- Nur eingerichtete Benutzer können System nutzen
- Einrichten von Benutzern ist dem privilegierten Benutzer Systemadministrator (Super-User) vorbehalten.
 - In neueren Versionen (z.B. System V.R4) wurden spezielle Administratoren mit abgestuften Rechten eingeführt (vgl. auch Windows NT).
 - Einrichten geschieht durch spezielle Werkzeuge (z.B. adduser, SAM) oder durch Editieren der Datei `/etc/passwd`, in der alle (lokalen) Benutzer geführt werden.
- Benutzer
 - besitzt Namen (account)
 - systeminterne Kennung uid (Benutzernummer, User Identification), die Basis für Rechtsüberprüfungen ist.
Der Super-User besitzt die uid 0.
 - ist ein und mehreren Benutzergruppen zugeordnet, die in `/etc/group` geführt werden.

- Anmeldungsverfahren eines Benutzers (login)
 - verwendet Passwort-basierte Authentisierung.
Passwörter in verschlüsselter Form in der Datei /etc/passwd (bzw. für Gruppen in /etc/group) gespeichert.
 - Problem: Datei muss „world readable“ sein
 - Jeder kann sie kopieren und in aller Ruhe die enthaltenen Passwörter (per „brute force“) entschlüsseln
 - Heute in der Regel: Verschlüsselte Passwörter in „shadow“-Passworddatei ausgelagert, nur für root zugreifbar
- Nach erfolgreicher Authentisierung
 - besitzt der Benutzer die systeminterne Identität (uid, gid) entsprechend den Festlegungen in /etc/passwd
 - und die dort ebenfalls festgelegte initiale Anwendung (i.d.R. eine Shell) wird gestartet.

Zur Erinnerung:

- Schutzmechanismen beinhalten technische Verfahren in Rechensystemen, die dem Schutz von Informationen dienen, um Sicherheit zu erreichen.

Gliederung

1. Schutzumgebungen (Protection Domains)
2. Zugriffskontrollisten
3. Capabilities
4. Standardisierte Sicherheitskriterien

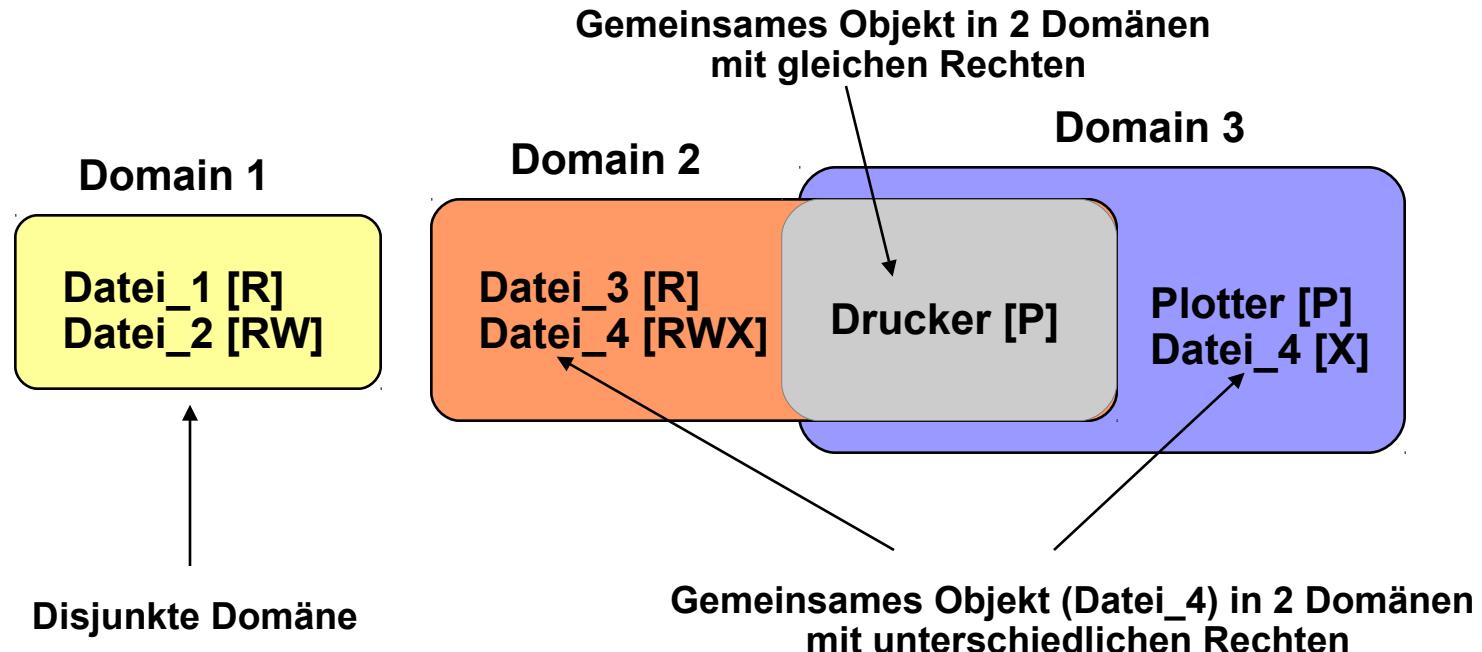
Def

- Objekte
 - sind die Einheiten, die geschützt werden sollen, z.B. Dateien.
 - besitzt einen eindeutigen Namen, über den auf das Objekt zugegriffen wird
 - Menge von Operationen, die auf dem Objekt ausgeführt werden können (Sicht von abstrakten Datentypen).
- Subjekte
 - sind die aktiven Einheiten, die auf Objekte zugreifen wollen.
 - Beispiele: Benutzer, Prozesse.
- Recht
 - beinhaltet die Erlaubnis zur Ausführung einer Operation auf einem Objekt.
- Schutzumgebung (Protection Domain, Domäne)
 - Menge von Paaren (Objekt, Recht).
- Einem Subjekt ist in jedem Augenblick eine Schutzumgebung zugeordnet, die die Menge der zugreifbaren Objekte und der auf ihnen ausführbaren Operationen beschreibt.
- Ein Subjekt kann prinzipiell seine Schutzumgebung wechseln.

Beispiel: Schutzumgebungen

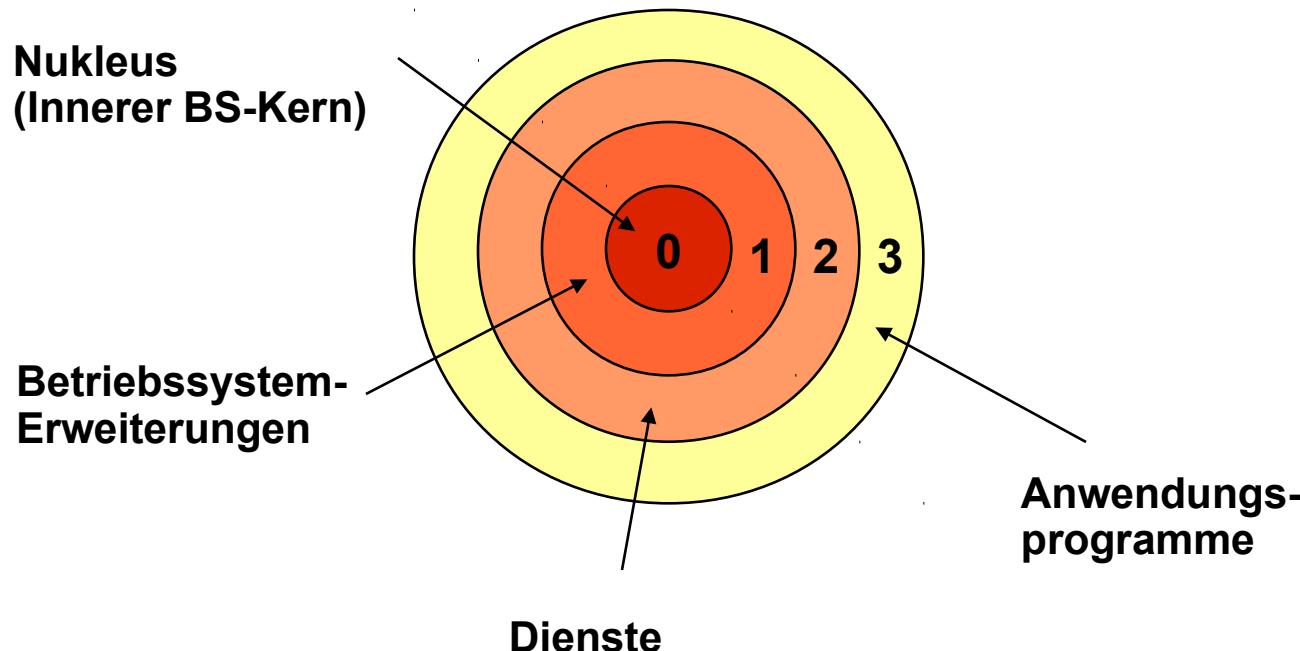


- Objekte:
 - Dateien mit den Operationen Read, Write, eXecute.
 - Drucker, Plotter mit der Operation Print.
- Domänen:



- Objekte: Dateien einschl. Spezialdateien (Geräte).
- Operationen: Read, Write, Execute.
- Subjekte: UNIX-Prozesse.
- Jedem Prozess zugeordnet:
 - reale Benutzer- und Gruppennummer (uid, gid):
 - Prozess ist "Stellvertreter" des realen Benutzers uid und der Gruppe gid im System.
 - effektive Benutzer- und Gruppennummer (euid, egid):
 - zur Überprüfung der Dateizugriffsrechte verwendet (s.u.).
 - Zugriffsumgebung:
 - Menge aller Dateien, deren Eigentümer oder Gruppe mit der effektiven Benutzer-/Gruppennummer des Prozesses übereinstimmt.
 - Rechte bzgl. Dateien sind durch Festlegungen im Inode gegeben.

- Verallgemeinerung der Zweiteilung Benutzermodus / Kernmodus.
- Ursprung: MULTICS protection rings, z.B. 16 Ringe.
- Beispiel: Intel '386 erlaubt 4 Ringe (Privilege Levels).



- Eine Schutzmatrix (oder Zugriffsmatrix) ist eine abstrakte Beschreibung einer Rechtssituation des Gesamtsystems.
- dient der Überprüfung der Rechtmäßigkeit eines beabsichtigten Zugriffs.

Objekte

		Datei_1	Datei_2	Datei_3	Datei_4	Drucker	Plotter
		Read	Read Write				
D o m ä n e n	Domäne 1	Read	Read Write				
	Domäne 2			Read Write	Read Write Execute	Print	
	Domäne 3				Execute	Print	Print

Menge der Rechte in einer Domäne
an einem Objekt

- Modellierung der Rechtmäßigkeit von Domänenwechseln durch Auffassen von Domänen als Objekte mit der Operation "enter".

		Objekte						Domäne Domäne Domäne		
		Datei_1	Datei_2	Datei_3	Datei_4	Drucker	Plotter	1	2	3
Domäne 1	Read	Read Write							Enter	Enter
			Read Write	Read Write Execute	Print					
				Execute	Print	Print				
Domäne 2										
Domäne 3										

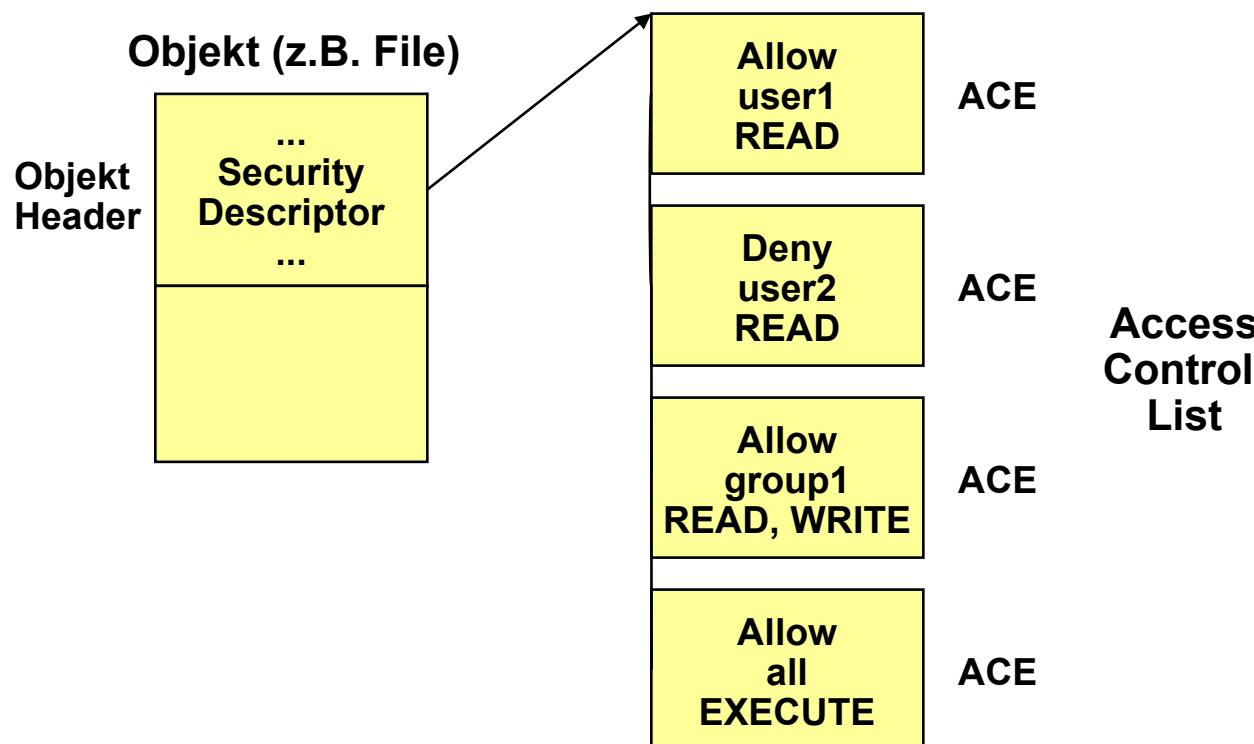
10.5.2 Zugriffskontrollisten (ACL)



- Information der abstrakten Schutzmatrix wird nicht als Matrix gespeichert (groß, dünn besetzt).
Hier betrachtet: *spaltenweise* Speicherung.
- Zugriffskontrolliste oder Access Control List (ACL)
 - Jedem Objekt zugeordnet
 - enthält alle Domänen, die auf dieses Objekt zugreifen dürfen, zusammen mit den jeweiligen Rechten.
- Bei Zugriff auf ein Objekt
 - Auswertung der Schutzumgebung
 - Entscheidung über die Zulässigkeit des beabsichtigten Zugriffs.
- Eigentümer eines Objekts kann die Zugriffskontrolliste jederzeit ändern
 - Rechteentzug (*Revocation*) einfach
 - Für bereits "geöffnete" Objekte kann der Rückruf eines Rechts schwierig oder unmöglich sein.

- Sehr einfache Form von Zugriffskontrollisten für Dateien (und Geräte usw.)
- Im Inode der Datei geführt
- Jeweils 3 Rechte-Bits (rwx für Lesen, Schreiben, Ausführen)
 - für den Eigentümer der Datei (Owner),
 - die Gruppe des Eigentümers (Group)
 - für alle andere Benutzer (Others)
- Als Zusatzprodukte sind allgemeine ACL-basierte Dateisysteme verfügbar.
 - Diese werden zur Erreichung einer C2-Sicherheitsstufe benötigt, vgl. 10.5.4.

- Windows NT verwaltet für jedes Systemobjekt (z.B. File, Thread, Event) eine Zugriffskontrolliste (ACL), die Bestandteil eines Security-Deskriptors des Objekts ist. Die Einträge der ACL werden Access Control Entries (ACE) genannt.

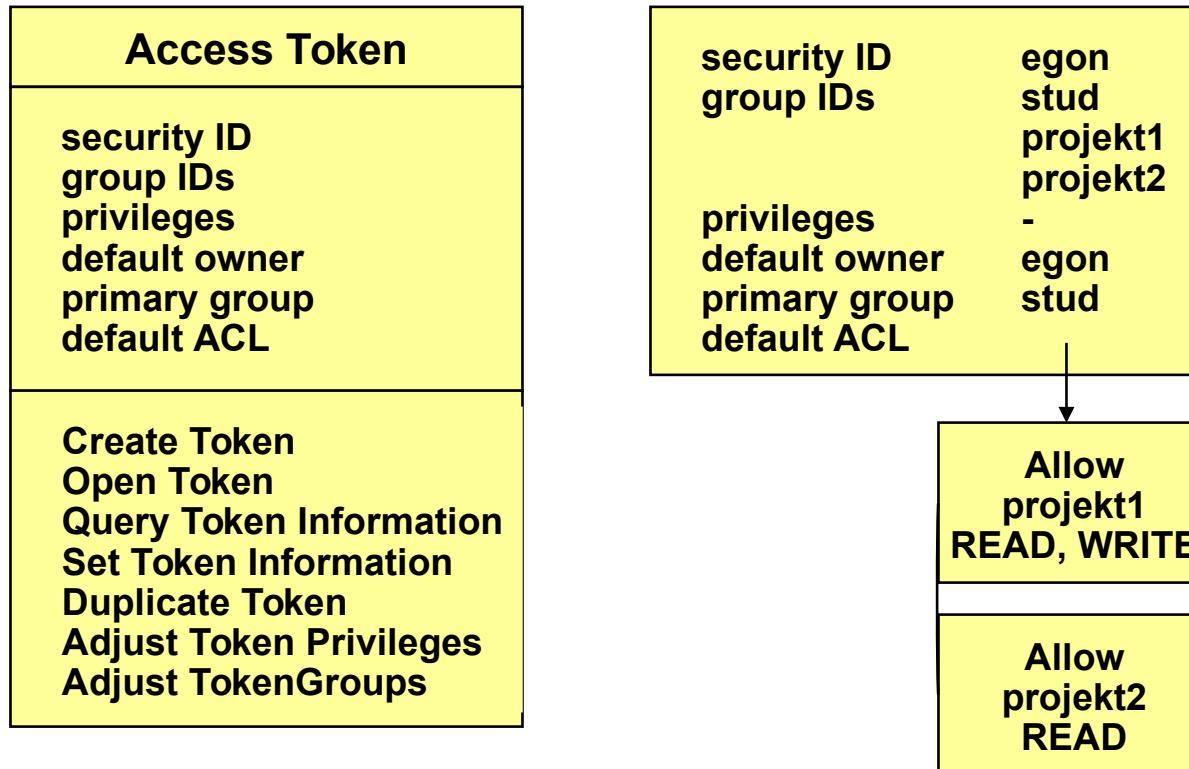


Beispiel: Windows NT (2)



10.5.2

- Windows NT erzeugt nach erfolgreicher Authentifizierung eines Benutzers ein Objekt vom Typ Access Token, das jedem Prozess des Benutzers fest zugeordnet wird.

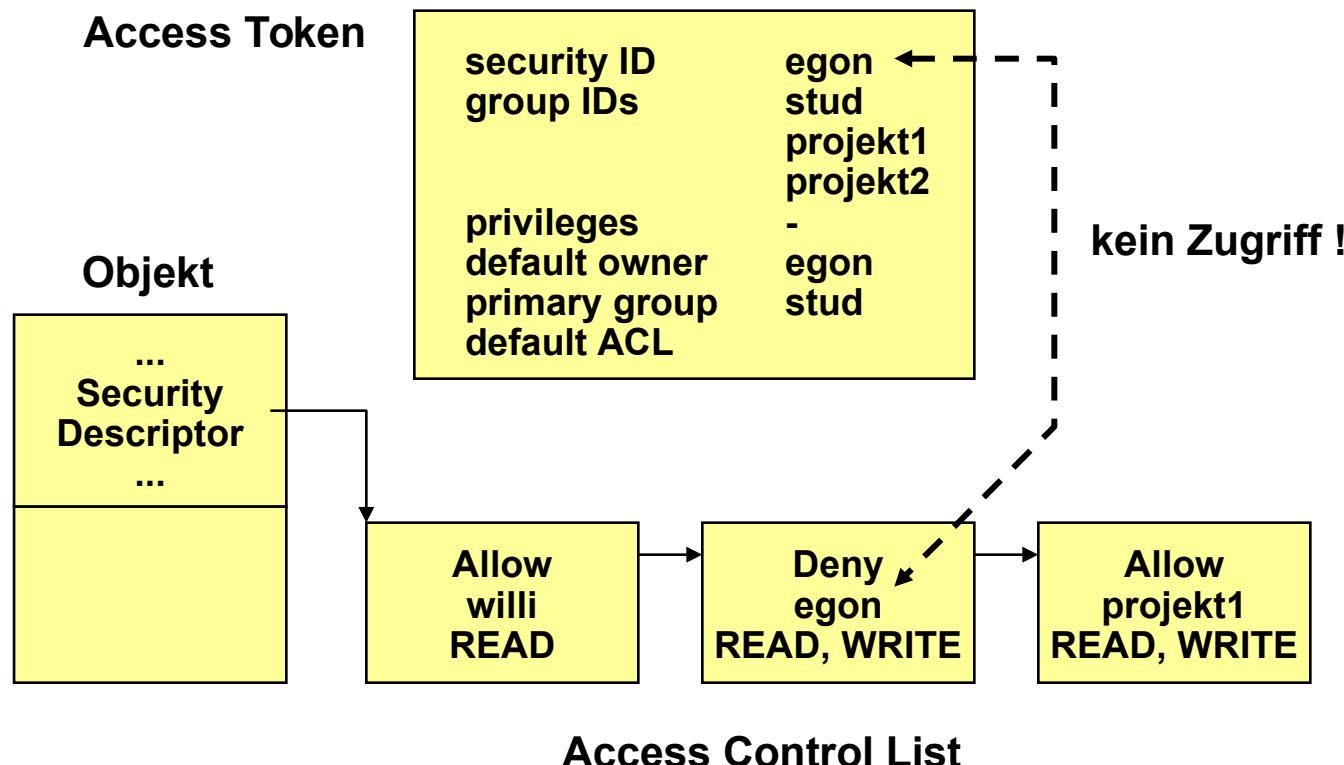


Beispiel: Windows NT (3)



10.5.2

- Windows NT überprüft beim Öffnen eines Objects die Access Control Liste des Objekts mit der Information des Access Token des zugreifenden Prozesses.



10.5.3 Capabilities



- Hier: zeilenweise Speicherung der Schutzmatrix.
- Jedem Prozess wird Capability-Liste (C-Liste) zugeordnet:
 - jedes Element heißt Capability
(Dennis, van Horn 1966; Fabry 1974)
 - Capability enthält Typfeld, Rechtefeld und Objektfeld mit Referenz auf das entsprechende Objekt
 - Liste benennt damit alle Objekte mit jeweiligen Zugriffsrechten.
- Capabilities werden i.d.R. durch ihre relative Position in der Capability-Liste identifiziert (analog UNIX File Deskriptoren).
- Beispiel einer typischen Capability-Liste:

	Typ	Rechte	Objekt
0	Datei	R W -	Zeiger auf Datei_3
1	Datei	R W X	Zeiger auf Datei_4
2	Drucker	P	Zeiger auf Drucker1

- Alternativen zur Sicherstellung der Unfälschbarkeit von Capabilities im Arbeitsspeicher:
 - Kennzeichnende Lösung (Tagged Memory): Jedes Speicherwort enthält Zusatzbit, das das Wort als Capability kennzeichnet. Tag-Bit kann nur im Kernmodus vom Betriebssystem geändert werden.
 - Aufteilende Lösung: Capabilities und Daten werden getrennt aufgehoben, C-Listen werden nur durch das Betriebssystem manipuliert. Ein Prozess referiert eine Capability durch Angabe des Index in der C-Liste.
Beispiele: Betriebssystem Hydra (Wulf, CMU);
Prozessor Intel iAPX 432.
 - Verschlüsselung: Capabilities werden im Benutzeradressraum aufbewahrt, aber mit einem für Benutzer geheimen Schlüssel verschlüsselt. Unversehrtheit kann vom Betriebssystem mit grosser Sicherheit überprüft werden.
Beispiel: Betriebssystem Amoeba (Tanenbaum).

- Klassifizierung für sogenannte Sichere Systeme der Informationstechnik basierend auf Kriterienkatalogen.
- Kriterienkataloge finden häufig bei der Beschaffung "Sicherer IT-Systeme" Anwendung.
- Ursprung US DoD "Orange Book"
 - TCSEC: Trusted Computer Systems Evaluation Criteria, 1983
- Deutschland:
 - Zuständigkeit heute:
Bundesamt für Sicherheit in der Informationstechnik (BSI)
 - ehemals Zentralstelle für Sicherheit in der Informationstechnik (ZSI)
 - Deutsche IT-Sicherheitskriterien (Grünbuch), 1989
 - Konzept der Trennung von Funktionalität und Prüftiefe (Qualitätsstufe)

- Europa:
 - Information Technology Security Evaluation Criteria - ITSEC, 1998
 - Common Criteria zur Bewertung
(Gegenseitige Anerkennung von Prüfzertifikaten), 1998
 - Common Criteria Version 2.1 als
weltweiter Standard ISO/IEC 15408
 - Teil 1: Introduction and General Model
 - Teil 2: Security Functional Requirements
 - Teil 3: Security Assurance Requirements
 - Unterscheidung Funktionalität und Vertrauenswürdigkeit
 - Vordefinierte Beispielklassen (Funktionalitätsklassen)
 - Vertrauenswürdigkeit unterscheidet zwischen Korrektheit und
Wirksamkeit (Stärke *niedrig*, *mittel* und *hoch*)
 - Bewertung des Vertrauens in die Korrektheit durch sechs
hierarchische Evaluationsstufen E1 (niedrig) bis E6 definiert.

- Grobe Zuordnung:

<i>ITSEC</i>	<i>TCSEC</i>
E0	D
F-C1, E1	C1
F-C2, E2	C2
F-B1, E3	B1
F-B2, E4	B2
F-B3, E5	B3
F-B3, E6	A1

- Für Klasse C2 (Orange Book) sind insbesondere mit technischen Mitteln innerhalb des Betriebssystems zu erfüllen:
 - Sichere Login-Prozedur zur Authentifizierung
 - Discretionary Access Control
 - Eigentümer eines Objekts hat freie Entscheidung, welche Rechte er welchem Subjekt oder Gruppe von Subjekten an dem Objekt zubilligt
 - wie bisher besprochen, z.B. auf der Basis eines ACL-Mechanismus durchgesetzt.
 - Auditing
 - Möglichkeit zur Erkennung und Protokollierung wichtiger sicherheitsrelevanter Ereignisse im System zusammen mit der Identität des verursachenden Benutzers.
 - Initialisierung aller Speicherbereiche
 - niemand kann Informationen eines früheren Benutzers einer Datenstruktur in Erfahrung bringen.

- Die Klasse B2 (Orange Book) ist i.w. für Anwendungen im militärischen Bereich sinnvoll:
 - Mandatory Access Control
 - Jeder Benutzer und jedes Objekt hat Sicherheitseinstufung (Clearance Level):
z.B. Vertraulich, Geheim, Streng Geheim.
 - Information darf nicht zu Benutzern mit geringerer Sicherheitstufe fließen.

Was haben wir in Kap. 9 gemacht?

- Konzepte von Dateisystemen.
- Verwalten von Mengen von Dateien und Verzeichnissen.
- Sicherheit und Schutz.
- In 10.1 und 10.2 zunächst eine äußere Benzersicht
 - Benennung von Dateien
 - Dateistrukturen und -Typen
 - Zugriffsarten
 - Dateiattribute
 - Moderne Dateisysteme unterstützen hierarchische Strukturierung beliebiger Tiefe.

- In 10.3 Dateisysteme aus der Sicht eines Implementierers
 - Zuordnung von Plattenblöcken
 - Freispeicherverwaltung
 - Realisierung von Verzeichnissen
 - Repräsentierung von Dateien im Hauptspeicher
 - Zuverlässigkeitssaspekte
 - Performance-steigernde Maßnahmen
- In 10.4 und 10.5: allgemeine Sicherheitsaspekte sowie Schutzmechanismen in Betriebssystemen
 - Authentifikation von Benutzern
 - Zugriffskontrollisten und Capabilities als Realisierungen einer abstrakten Schutzmatrixt.

Kap. 11: Input / Output

- Eine der Hauptaufgaben eines Betriebssystems ist die Überwachung und Steuerung aller E/A-Geräte (I/O-Geräte, I/O Devices):
 - Kommandos an Geräte geben
 - Unterbrechungen (Interrupts) bedienen
 - Fehlerbehandlung durchführen
- Das I/O-System eines Betriebssystems hat i.d.R. einen beträchtlichen Umfang (z.B. ca. 50% in 4.3BSD)
- Ziele eines I/O-Systems:
 - Einfachheit der Schnittstelle zwischen Geräten und dem Rest des Systems
 - Hohes Maß an Geräteunabhängigkeit der Schnittstelle

- 11.1 I/O-Hardware
- 11.2 I/O-Software
- 11.3 Plattentreiber
- 11.4 Uhrtreiber
- 11.5 Terminal-Treiber
- 11.6 Beispiel: UNIX
- 11.7 Zusammenfassung

I/O-Hardware wird hier nicht im einzelnen besprochen. Bzgl. Details vgl. auch Lehrbuch Tanenbaum.

In Stichworten:

- I/O-Geräte
 - Blockorientierte Geräte:
 - Speichern Informationen in Blöcken gleicher Größe mit jeweils eigener (Block-)Adresse
 - Beispiel: Festplatten
 - Zeichenorientierte Geräte:
 - Erzeugen und akzeptieren Zeichenströme
 - Keine Blockgrenzen, Adressen oder Suchoperationen
 - Beispiele: Terminals, Drucker, Mäuse, Netzwerkschnittstellen.
 - Sonstige Geräte:
 - z.B. Uhren

- Steuerwerke (Device Controller, Adapter):
 - I.d.R. spezialisierte kleine Rechner als Bindeglieder zwischen Systembus und Geräten
 - Beispiele für standardisierte Schnittstellen zwischen Controller und Geräten: USB, SCSI/SAS, IEEE 488
 - Bei Großrechnern: Intelligente I/O-Kanäle anstatt Controller, arbeiten eigenständig Kanalprogramme im Hauptspeicher ab
 - Controller sind über adressierbare Register zugreifbar
 - im allg. phys. Adressraum des Prozessors (z.B. MC 680x0)
 - oder in spez. I/O-Space (z.B. Intel 386)
 - Hierdurch erfolgt Initiierung von I/O-Operationen
- Direct Memory Access (DMA):
 - Eigenständiges Übertragen von Daten durch den Controller ohne Zuhilfenahme des Prozessors
 - Indirekte Performance-Beeinflussung durch Cycle Stealing

Gliederung

1. Ziele und Grobarchitektur
2. Unterbrechungsbehandlung
3. Gerätetreiber
4. Geräteunabhängige Software des BS-Kerns
5. I/O-Software im Benutzer-Modus

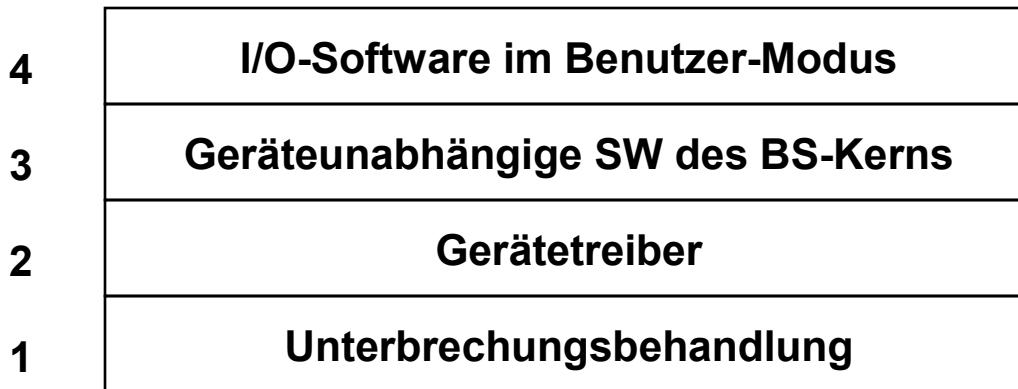
11.2.1 Ziele

- Geräteunabhängigkeit:
 - Schlüsselkonzept beim Entwurf
 - Ein Programm soll nicht wissen müssen, ob eine zugegriffene Datei z.B. auf einer Festplatte oder einem Diskettenlaufwerk gespeichert ist
- Einheitliches Benennungsschema für Dateien und Geräte
- Verwaltung gemeinsam und exklusiv benutzbarer Geräte:
 - Gemeinsam benutzbare Geräte erlauben die gleichzeitige Benutzung durch mehrere Prozesse (z.B. Magnetplatten)
 - Exklusiv benutzbare Geräte erlauben dies nicht (z.B. Drucker)

- Synchrone / Asynchrone I/O-Operationen:
 - synchron: Blockierung bis zum Abschluss der Operation
 - asynchron: Ende der Blockierung nach Einleitung der Operation
 - Für Anwendungsprogramme sichtbare E/A-Operationen können durch das Betriebssystem blockierend erscheinen (einfachere Programmierung), auch wenn physikalisches I/O asynchron und unterbrechungsgesteuert abläuft
- Fehlerbehandlung:
 - so nahe an der Hardware wie möglich
 - z.B. durch Wiederholung (Retry) bei transienten Fehlern

- Erreichung der Ziele durch geschichteten Entwurf der I/O-System-Software in vier Schichten (Layers):

Schicht



- **Diese Schichten werden im weiteren von unten nach oben behandelt.**

11.2.2 Unterbrechungsbehandlung



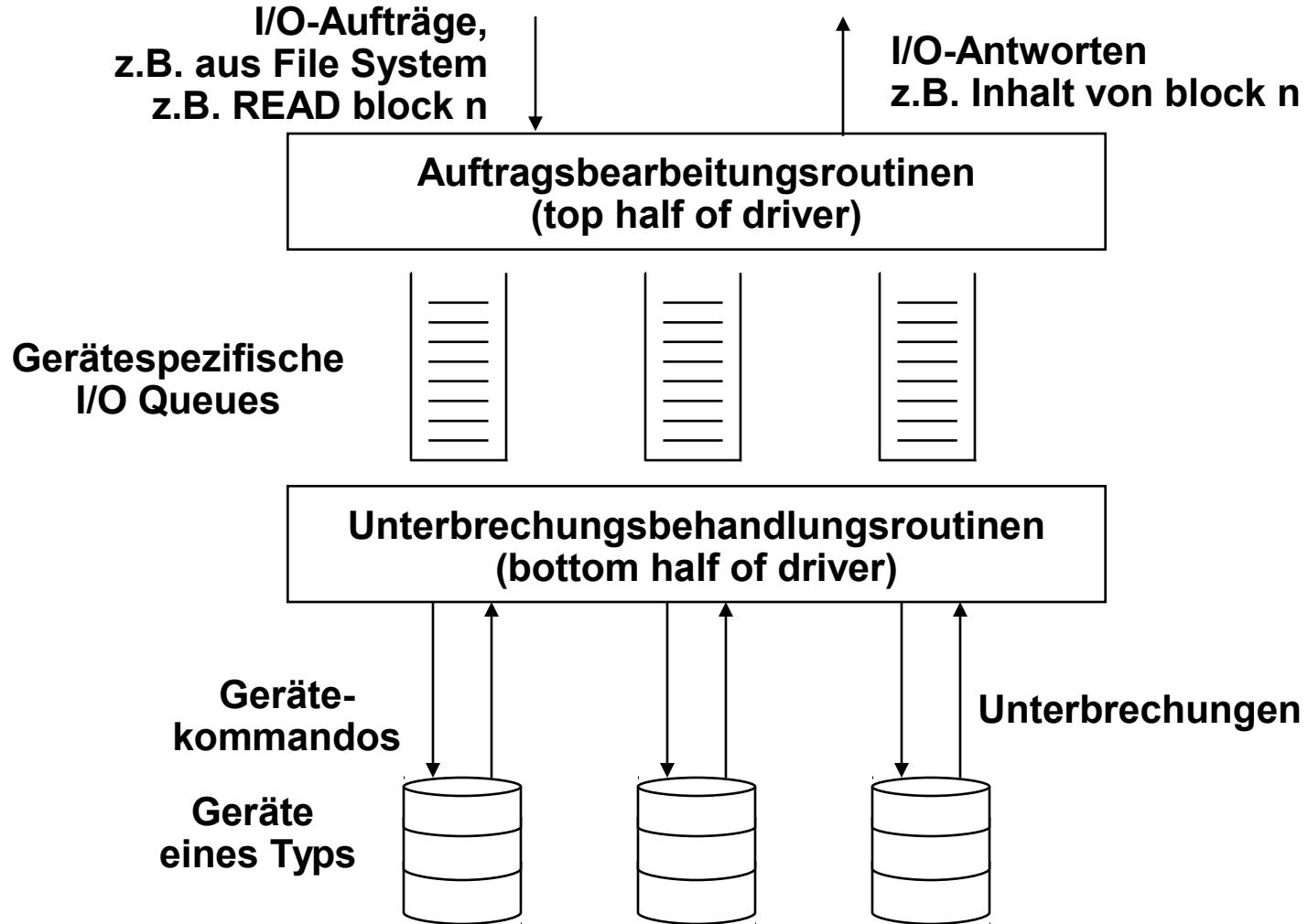
- Technisch: vgl. Vorlesungsteil Rechnerarchitektur
- Unterbrechungsbehandler (Interrupt Handler) sollten in einer möglichst niedrigen Schicht des Betriebssystems verborgen werden
- Gute Möglichkeit der Einkapselung:
 - Jeder Prozess, der I/O-Operation ausführt, blockiert bis zu deren Abschluss, z.B. durch:
 - P (Semaphor), oder
 - Wait (Condition), oder
 - Receive (Message).
 - Tritt eine Unterbrechung auf, die einer Fertigmeldung der I/O-Operation entspricht, entblockiert der Interrupt Handler den blockierten Prozess entsprechend:
 - V (Semaphor), oder
 - Signal (Condition), oder
 - Send (Message).

11.2.3 Gerätetreiber (Device Drivers)



- Gerätetreiber kapselt den gesamten geräteabhängigen Code für einen bestimmten Gerätetyp, d.h.:
 - Nur Gerätetreiber kennen Struktur der Controller-Register, setzen Kommandos in ihnen ab und überprüfen die korrekte Ausführung
 - z.B. nur der Plattentreiber muss sich um Sektoren, Spuren, Zylinder, Köpfe, Armbewegungen, Motore, Interleaving usw. kümmern
- Aufgabe eines Treibers ist das Bearbeiten von abstrakten Aufträgen der geräteunabhängigen Software-Schicht des Betriebssystem-Kerns (z.B. Lies Block n)
- Gerätetreiber bearbeitet i.d.R. mehrere Geräte eines Typs
- Gegebener abstrakter Auftrag muss durch den Treiber in eine konkrete Folge von Kommandos an einen Controller umgesetzt werden

- Typische Komponenten eines Gerätetreibers (z.B. UNIX):
 - Autokonfigurations- & Initialisierungsroutinen:
Einmalig benutzt während des Boot-Vorgangs zur Überprüfung der Existenz von Geräten eines bestimmten Typs sowie deren Initialisierung
 - I/O-Auftragsbearbeitungsroutinen
(obere Hälfte des Treibers):
I/O-Auftrag kann durch Ausführung eines System Calls eines Applikationsprogramms oder durch das virtuelle Speichersystem innerhalb des Betriebssystems entstehen
Treibercode ist i.d.R. synchron zur aufrufenden Seite (Auftraggeberseite) hin, d.h. Aufrufer können im Treiber blockieren
 - Unterbrechungsbehandlungsroutinen
(untere Hälfte des Treibers)
Treibercode ist i.d.R. asynchron zur Geräteseite hin
 - Dazwischen liegen gerätespezifische Warteschlangen für I/O-Kommandos an die vom Treiber verwalteten Geräte



11.2.4 Geräteunabh. SW des BS-Kerns

- Aufgaben:
 - Bereitstellung von Funktionen, die unabhängig von bestimmten I/O-Gerätetypen sind
 - Bereitstellung einer einheitlichen Schnittstelle zur I/O-Software in den Benutzerprozessen
- Typische Funktionen:
 - Bereitstellung einer einheitlichen Schnittstelle zur Einbindung von Gerätetreibern
 - Benennung von Geräten
 - Zugriffsschutz für Geräte
 - Bereitstellung von Blöcken mit einer festen geräteunabhängigen Blockgröße
 - Pufferung
 - Speicherverwaltung auf blockorientierten Geräten
 - Zuteilung und Freigabe von Geräten
 - Fehlermeldung

11.2.5 I/O-Software im Benutzermodus



- I/O-Bibliotheken
 - Funktionen, die zu Anwendungsprogrammen hinzugebunden werden
 - Beispiele: `write` und `printf` im C-Programm, die mit wenig bzw. mehr Aufwand zu einem WRITE system call führen
- Spooling-System
 - Verwaltung von Druckaufträgen für eine Menge von Druckern
 - realisiert durch einen speziellen Dämon-Prozess und ein spezielles (Spooling)-Verzeichnis
- E-mail-System
 - analog
- ...

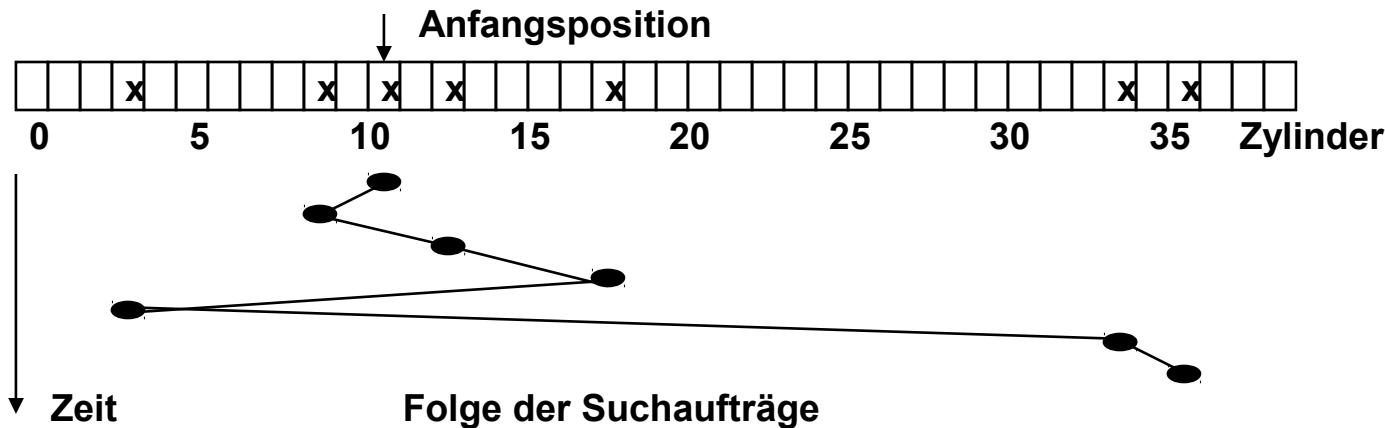
11.3 Plattentreiber



- Plattentreiber
 - sind Treiber für Magnetplatten
 - besitzen alle unter 11.2.3 besprochenen Eigenschaften
 - Im folgenden Plattenarm-Scheduling-Verfahren
 - in Plattentreibern bei der Auswahl des nächsten durchzuführenden Auftrags eines Plattenlaufwerks eingesetzt
 - Ausgangspunkt: Zugriffszeit zu einem Plattenblock ist bestimmt durch:
 - Positionierzeit des Plattenarms (Seek Time)
 - Latenzzeit (bis Block unter Schreib-/Lesekopf, im Mittel 1/2 Rotationszeit)
 - Transferzeit (Übertragung des Blocks)
- Positionierzeit hat i.d.R. einen hohen Anteil (vgl. 9.3.3.2)
- ⇒ Leistungssteigerung kann also durch Reduktion der mittleren Positionierzeit erreicht werden.

- Algorithmen, die für eine Menge von Platten-I/O-Aufträgen die mittlere Positionierzeit gegenüber FCFS verringern:
 - Shortest-Seek-First (SSF)
 - Fahrstuhl-Algorithmus

- Bei der Auswahl des nächsten I/O-Auftrags wird derjenige gewählt, der "am nächsten" ist, d.h. die geringsten Kopfbewegungen erfordert
- Beispiel:

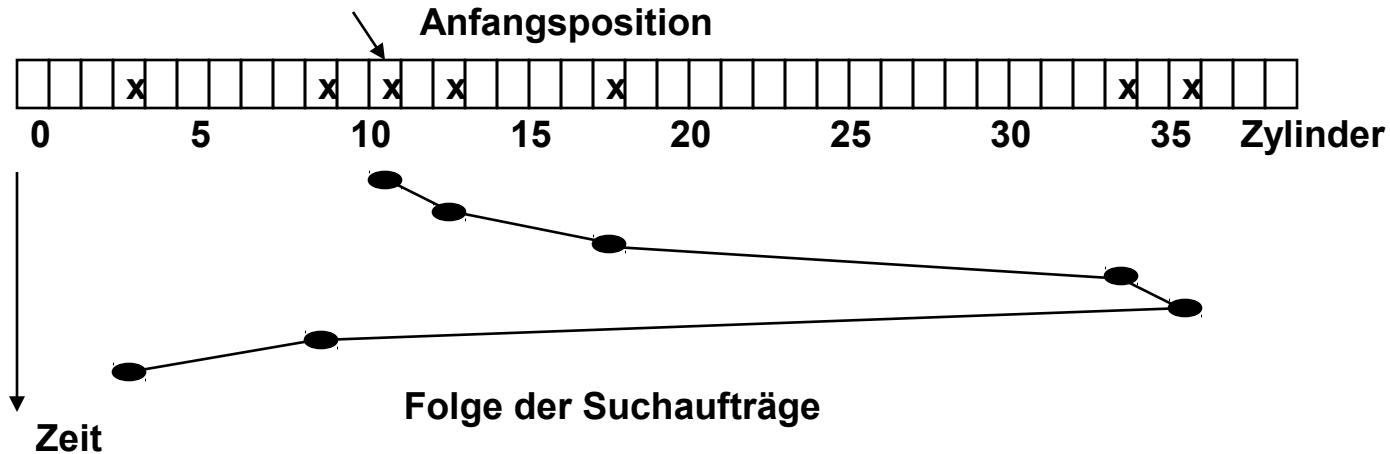


- SSF reduziert die Positionierzeiten gegenüber FCFS deutlich
- Aber: Shortest-Seek-First ist unfair:
 - Bei stark belasteter Platte tendiert der Arm dazu, sich in der Mitte der Platte aufzuhalten. Anforderungen bzgl. äußerer und innerer Zylinder werden nachteilig bedient.

Fahrstuhl-Algorithmus (Elevator Seek)



- Wie bei einem Aufzug bewegt sich der Plattenarm solange in eine Richtung, bis es in dieser Richtung keine Plattenaufträge mehr gibt, und wechselt dann die Richtung
- Beispiel:

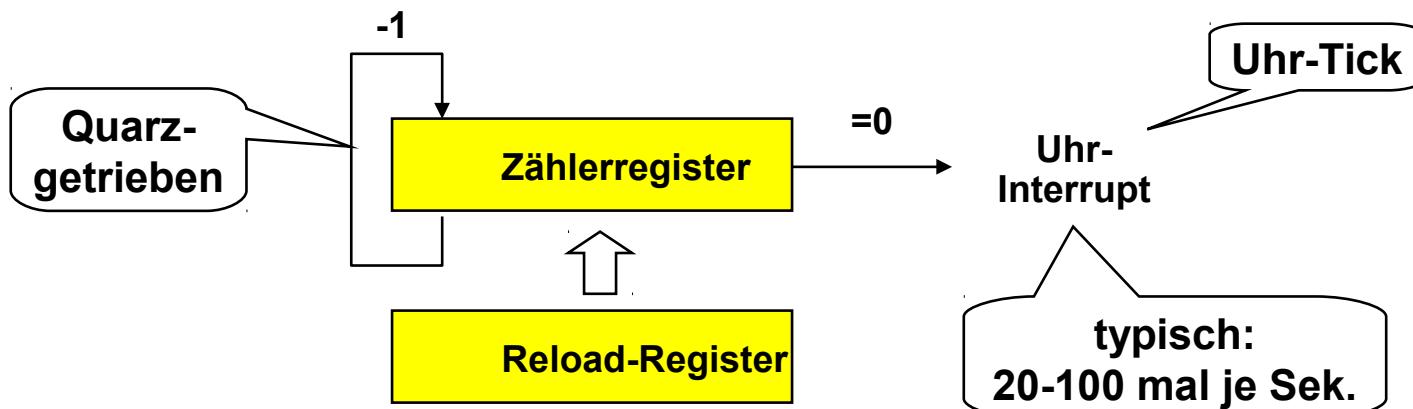


- Für jede gegebene Menge von Aufträgen ist die obere Grenze der Summe der Armbewegungen bekannt, nämlich zweimal die Anzahl der vorhandenen Zylinder

11.4 Uhrtreiber



- Hardware eines Timers besteht i.d.R. aus:
 - einem Oszillator fester Frequenz
 - einem Zähler
 - bei jeder Schwingung des Oszillators dekrementiert
 - bei Erreichen des Werts Null eine Unterbrechung erzeugt
 - einem Reload-Register
 - ladbarer Wert initialisiert automatisch den Zähler neu, wenn dieser den Wert Null erreicht
- Uhrtreiber erbringt im Falle eines Uhrticks durchzuführende Funktionen



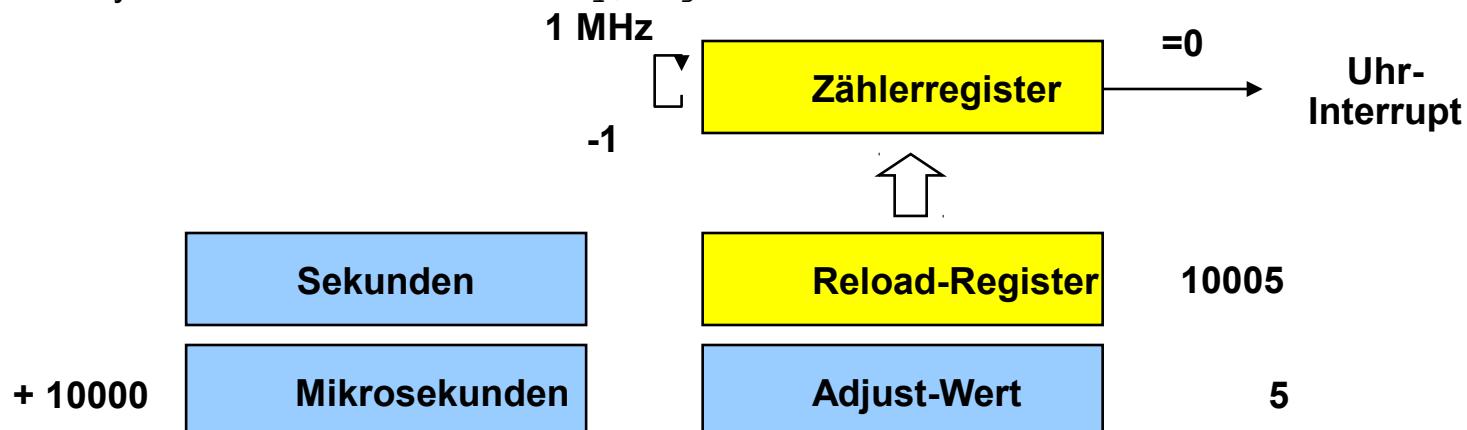
- Verwalten der Uhrzeit (Time-of-Day, Wall Time):

- Variable des Betriebssystemkerns
- Inkrementieren bei jedem Uhrtick
- Beispiel UNIX
 - zwei 32-Bit (oder 64-Bit) Integer-Variablen

Anzahl Sekunden seit 1.1.1970

Anzahl μ s (oder ns) in der aktuellen Sekunde

- typ. 100 Interrupts/s
- bei Interrupt werden Variablen um nominelle Anzahl μ s erhöht
- Korrekturwert (Adjust-Wert) für Ausgleich der Drift des Quarzes
- Systemdienste `settimeofday`, `adjtime`



- Quantum-Überwachung:
 - Bei Prozesswechsel wird ein Zähler mit der Dauer des dem Prozess zugewiesenen Quantums, gemessen in Uhrticks, initialisiert
 - Bei jedem Uhrtick wird der Zähler dekrementiert
 - Wird der Wert Null erreicht, ruft der Uhrtreiber den Scheduler zur Neuvergabe des Prozessors auf

- Alarm-Systemdienst (Interval Timer):
 - Ziel: Beauftragung des Betriebssystems, nach einer gewählten Zeitspanne einen Alarm (ein Signal, eine Nachricht o.ä.) zuzustellen.
 - Beispiel UNIX: `setitimer` system call
 - erlaubt Einplanen von SIG(VT)ALRM-Signalen nach Ablauf einer bestimmten Zeitspanne der realen Zeit oder der virtuellen Zeit des Prozesses, die nur fortschreitet, wenn der Prozess rechnend ist
 - anwendungsspezifische Reaktion kann mittels `sigaction` zugeordnet werden
 - Interval Timer verwaltet i.d.R. eine nach den Zeitpunkten geordnete Liste der Zeit-Alarne.

- Watchdog-Funktion (Watchdog Timer):
 - Ziel: Zeitüberwachung Betriebssystem-interner Vorgänge (Timeouts)
 - Vorgehensweise analog
- Accounting:
 - Ziel: "Faire" Abrechnung der CPU-Nutzung durch einen Prozess
 - Bearbeitung von Unterbrechungen wird i.d.R. dem gerade betroffenen Prozess zugeordnet (nicht dem Verursacher, Einfachheit)
 - Bei jedem Uhrtick wird ein CPU-Nutzungszähler im Prozesskontrollblock des sich gerade in Ausführung befindlichen Prozesses inkrementiert

- Profiling:
 - Ziel: Ermittlung statistischer Aussagen über den Zeitverbrauch bestimmter Programmteile (z.B. Erwartungswerte für die Ausführungs dauern von Prozeduren).
 - Dazu: Erstellen einer Häufigkeitsverteilung über das Antreffen des Programmzählers (Program Counters) in bestimmten Adressbereichen.
Beispiel: UNIX `profil` System Call
 - Mittels Analyse des Stacks (Call Graph) und Symbol Table des Programms sowie Anwendung des "Gesetzes der Großen Zahl" werden die gewünschten Erwartungswerte bestimmt
 - Bei jedem Uhrtick überprüft der Uhrtreiber, ob Profiling für den rechnenden Prozess eingeschaltet ist. Wenn ja, wird bestimmt, in welche Klasse der Häufigkeitsverteilung der aktuelle Programmzählerwert fällt, und der Zähler dieser Klasse wird inkrementiert.

11.5 Terminal-Treiber

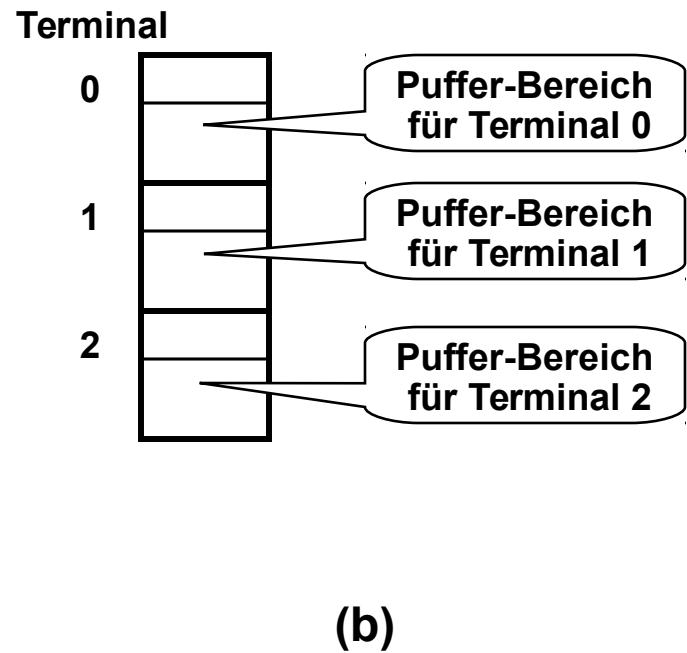
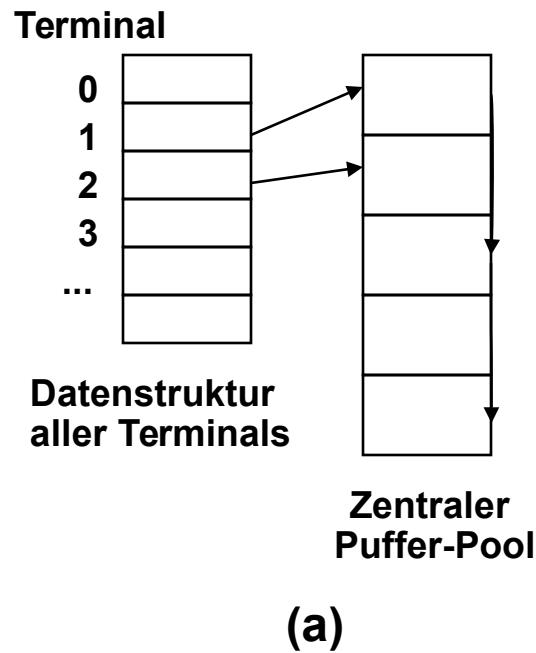


- Typen von Terminals:
 - Serielle Terminals bestehen aus Tastatur und Bildschirm sowie einer seriellen Schnittstelle (z.B. RS 232) zur zeichen-orientierten Kommunikation mit einem Rechner (Host)
 - Memory-Mapped Terminals
 - ✚ im phys. Adressraum sogenanntes Video-RAM vorhanden, das mittels üblicher Maschinenbefehle oder durch Befehle eines HW-Beschleunigers beschrieben werden kann
 - ✚ Video-Controller liest zyklisch den Inhalt des Speichers und erzeugt daraus die Video-Signale für den Bildschirm
 - ✚ Tastatur und Maus werden als separate Geräte mit eigenem Treiber eingesetzt
- Unterscheidung:
 - ✚ Zeichenorientierte Geräte
(z.B. der ursprüngliche IBM-PC, heute unüblich)
 - ✚ Bitmap-Terminals erlauben Pixel-Verarbeitung (heute Standard)
- X-Terminals sind dagegen keine Terminals, sondern eigenständige Rechner, die mittels des X-Protokolls mit einem Host kommunizieren, um entfernte graph. Ausgabe zu bewirken

- Eigenschaften:
 - Aufgabe: Sammeln von Eingaben der Tastatur und Übergabe an Benutzerprogramme, wenn diese vom Terminal lesen
 - I.d.R. bewirkt jeder Tastendruck eine Unterbrechung
 - Falls Tastennummern an den Treiber übergeben werden, erfolgt zunächst eine Umwandlung anhand einer Code-Tabelle z.B. in ASCII Code
 - Raw Mode-Betrieb:
 - Zeichenorientiert: Die Zeichen werden ihrer exakten Folge unverarbeitet weitergegeben (z.B. für Editoren sinnvoll)
 - Cooked Mode-Betrieb:
 - Zeilenorientiert: Treiber übernimmt das Editieren innerhalb einer Zeile (z.B. Backspace-Bearbeitung)
 - übergibt nur bearbeitete Zeilen an Anwendung (z.B. Shell)
 - Zeichen müssen gespeichert werden, bis Zeile vollständig
 - Modus kann i.d.R. mittels Systemaufruf gewählt werden

Beispiel: UNIX `ioctl` System Call

- Verbreitete Ansätze zur Pufferung von Zeichen:
 - Zentraler Puffer-Pool für alle Terminals mit Verkettung von Puffern fester Länge (z.B. 64 Zeichen) für jedes Terminal (a)
 - Privater Pufferbereich für jedes Terminal (b)



- Problembereiche:
 - Tastatureingaben
 - erzielen keine automatische Ausgabe (z.B. bei Passwort-Eingabe wichtig!).
 - Ausgabe von eingegebenen Zeichen erfolgt durch Software und wird Echoing genannt
 - Behandlung von Tabulatoren
 - Zeilenfortschaltung mittels Wagenrücklauf und/oder Zeilenvorschub
 - Behandlung von Sonderzeichen im Cooked Modus (z.B. Löschzeichen, Escape-Zeichen, Dateiende)

- Eigenschaften:
 - Terminal-Ausgabe i.d.R. einfacher als Behandlung von Eingabe.
 - Treiber für serielle Terminals und Treiber für Memory-Mapped Terminals sind dabei sehr unterschiedlich.
 - Serielles Terminal
 - Zuordnung eines Puffers (u.U. aus demselben Puffer Pool wie die für Tastatur-Treiber benutzten).
 - Ausgabe: Daten werden in einen oder mehrere Puffer kopiert
 - Treiber führt I/O zeichenweise über die serielle Schnittstelle mit zwischenzeitlichem Warten auf Unterbrechung durch
 - Memory-Mapped zeichenorientiertes Terminal
 - Ausgabe unmittelbar in Video-RAM
 - Notwendig: Verwaltung einer "aktuellen Position"
 - Behandlung von Sonderzeichen (wie z.B. Backspace)
 - Besondere Beachtung auch für Zeilenvorschub am Seitenende sowie die Cursor-Positionierung

11.6 Beispiel: Das UNIX I/O-System

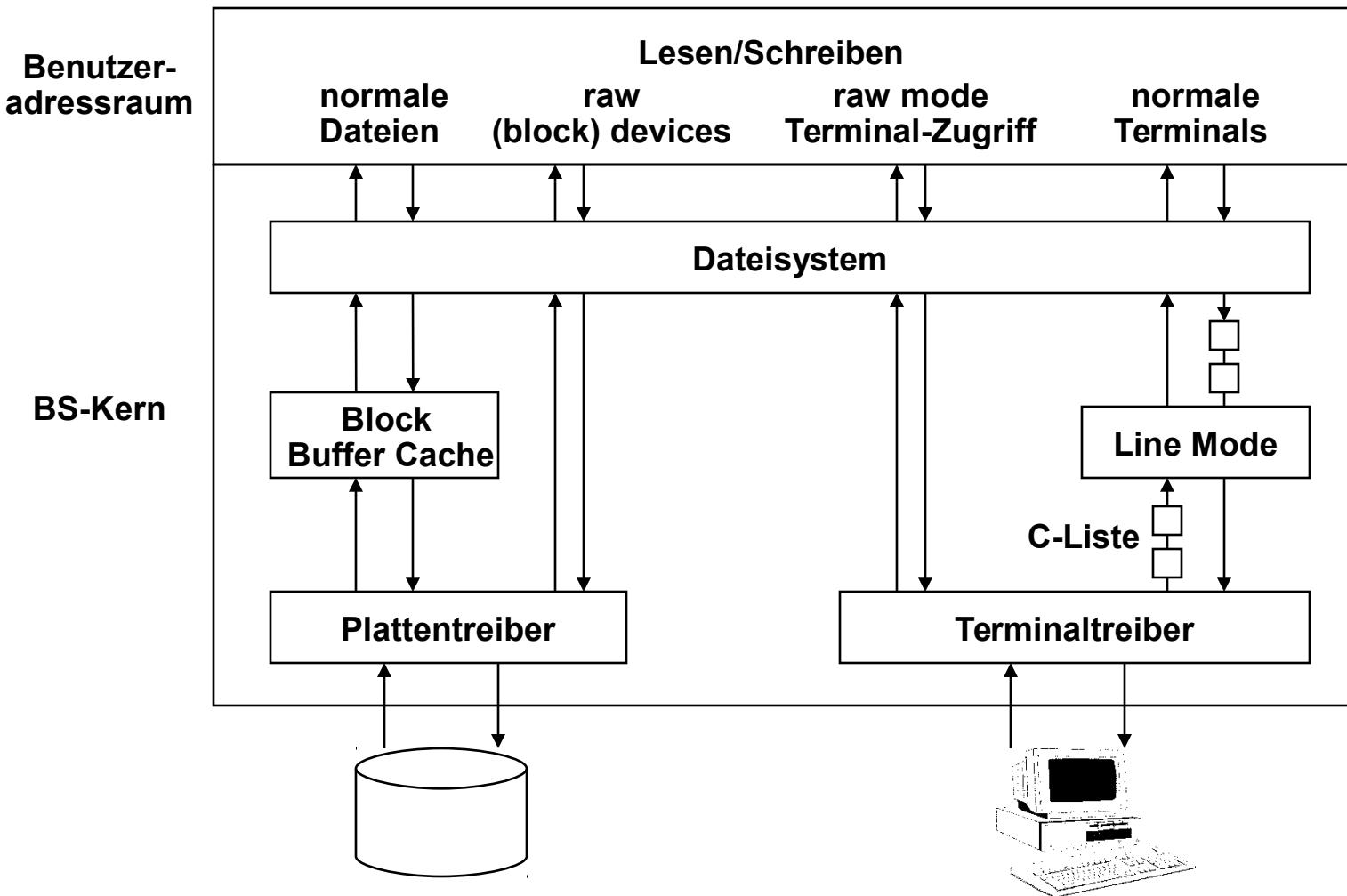


- BSD UNIX I/O-System besitzt alle in 11.2 beschriebenen Komponenten und prinzipiellen Eigenschaften.
- In jüngerer Zeit dynamisch ladbare Gerätetreiber
- Geräte haben Repräsentierung als Inode im Dateisystem mit Typ "special file".
- UNIX unterscheidet
 - blockorientierte Spezialdateien (Block Devices, z.B. Platten)
 - zeichenorientierte Spezialdateien (Character Devices, z.B. Terminals (ttys))
 - jeweils festgelegte Schnittstelle für Gerätetreiber dieser Klassen
- Inode einer Spezialdatei enthält
 - Major und Minor Device Number des Gerätes
 - Major Device Number selektiert den Gerätetyp und damit den zuständigen Treiber
 - Minor Device Number selektiert ein logisches Gerät (z.B. Platten-Partition oder bestimmtes Terminal)

- Für alle blockorientierten Geräte gilt:
 - Jeder Typ besitzt einen Eintrag in einer Tabelle (4.3BSD `bdevsw` block device switch).
 - Eintrag entspricht Treiber und enthält Adressen (Einsprungstellen) der Operationen des Treibers
 - Major Device Number eines Gerätes selektiert Eintrag
 - Jeder blockorientierte Treiber muss folgende Operationen bereitstellen:
 - `open`
 - `strategy` (Initiieren von `read` oder `write`)
 - `close`
 - `dump` (Hauptspeicherabzug bei Crash)
 - `psize` (Größe einer Partition in Blöcken)
 - Gemeinsame Pufferverwaltung für alle blockorientierten Geräte, den Block Buffer Cache (vgl. 9.3.6). Er reduziert die Anzahl der notwendigen Plattenzugriffe. Die Pufferverwaltung erfolgt gemäß LRU.

- Für alle zeichenorientierten Geräte gilt:
 - Jeder Typ besitzt einen Eintrag in einer Tabelle (4.3BSD `cdevsw` character device switch)
 - Character Device Typen werden auch verwendet, um "Raw Access" zu blockorientierten Geräten zu erhalten, d.h. unter Umgehung des Block Buffer Cache (z.B. für `fsck`, `dump`)
 - Jeder zeichenorientierte Treiber muss folgende Operationen bereitstellen:
 - `open`
 - `close`
 - `ioctl` (I/O control operation)
 - `mmap` (map device contents into memory)
 - `read`
 - `reset`
 - `select` (poll device for I/O readiness)
 - `stop`
 - `write`

- Gemeinsame Pufferverwaltung ohne Caching (unnötig) für alle wirklichen zeichenorientierten Geräte, die sogenannten C-Listen, je Zeichenstrom bestehend aus verketteten kleinen Blöcken mit bis zu 64 Zeichen.
- Ein Zeichenstrom kann im Treiber durch eine sogenannte Line Discipline vorverarbeitet werden (Cooked Character Stream) oder diese umgehen (Raw Character Mode).



11.7 Zusammenfassung



- I/O ist ein wichtiges, aber aufgrund der vielen spezifischen Eigenarten oft vernachlässigtes Thema
- Anteil des Betriebssystem-Codes für Ein-/Ausgabe ist beträchtlich
- In 11.1 wurden einige Eigenschaften von I/O-Hardware stichwortartig zusammengestellt
- In 11.2 wurden die vier Software-Schichten zur Ein-/Ausgabe allgemein besprochen
 - Unterbrechungsbehandlung
 - Gerätetreiber
 - geräteunabhängige Software im BS-Kern
 - I/O-Software auf Benutzerebene, wie z.B. I/O-Bibliotheken und Spooling-Dämonen
- In 11.3 bis 11.5 wurden einige Eigenschaften von Treibern für Platten (speziell Plattenarm-Scheduling-Verfahren) sowie für Uhren und Terminals besprochen
- In 11.6 wurde schließlich die Struktur der BSD UNIX I/O-Software als Beispiel behandelt