

Vorlesung *Betriebssysteme*

Dr. Felix Friedrich, Prof. Jürg Gutknecht

ETH Zürich

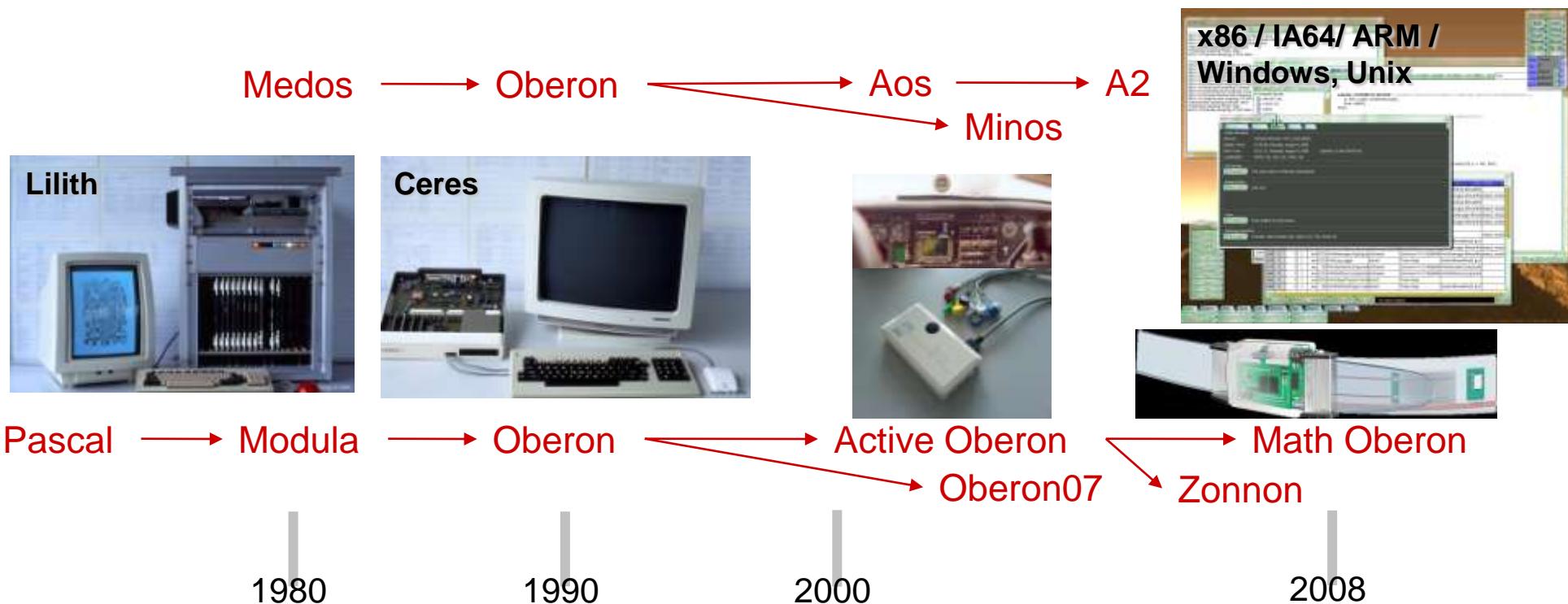
Frühjahrssemester 2009

Unsere Gruppe: Native Systems

<http://nativesystems.inf.ethz.ch>

Fokus

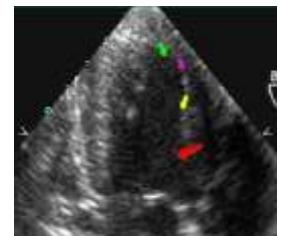
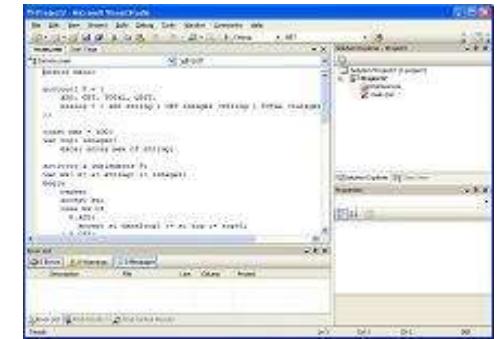
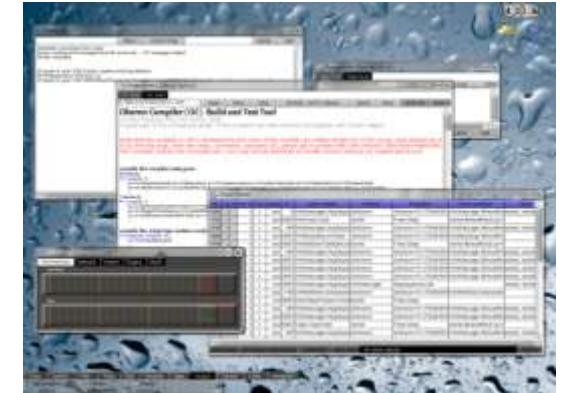
- Design und Implementation von möglichst einfachen Betriebssystem-Modellen, die Entwicklung und Einsatz heutiger und zukünftiger Applikationen optimal unterstützen
- Haupt-Gebiete
 - Programmiersprachen in der Tradition von Pascal / Modula
 - Betriebssysteme („lightweight“, „embedded“, „native“)



Native Systems

Projekte und Forschungsinhalt

- Neue Grafische Oberflächen und APIs
 - Aos / A2
- Neue Programmiermodelle
 - Active Oberon
 - Strukturierte Komponenten, persistente Laufzeitsysteme
 - Zonnon (auf .NET und MONO)
 - MathOberon
- Anwendungen
 - E-Health (Realtime OS): autonomes EKG, vollautomatisches Echokardiogramm
 - Steuerung von Industrieanlagen (In Kooperation mit Colortronics)
 - ONBASS (Fault Tolerant OS), Digital Arts ...
- Hardware-Software Codesign
 - „Supercomputer in the Pocket“:
Multi-Core FPGA + Active Oberon Variante + Runtime zum Einsatz in der autonomen Echokardiographie



Administratives

- **Vorlesungszeiten und Orte**
 - Mittwoch, 9.15 – 10.00, HG E3
 - Freitag, 10.15 – 12.00, HG E5
- **Übungszeiten**
 - Montag, 14.15 – 15.45,
 - Mittwoch, 15.15 – 16.45
- **Vorlesungshomepage:**
 - Aktuelle Informationen zu Vorlesung, Übungen, Prüfung
 - Folien und Übungsserien in pdf Form
 - Kommentierte Folien 1x wöchentlich
 - Informationen zu den praktischen Übungen
- **Chefassistent**
 - Sven Stauber

Vorlesungszeit: 16.2.2008-29.5.2008.

keine Vorlesungen:

Fr, 10.4.– So, 19.4. (Osterferien)
Fr, 1.5. (Tag der Arbeit)

1. Übung: 23.2. / 25.2.

keine Übungen:

Mo, 13.4., Mi. 16.4. (Osterferien)
Mo, 20.4. (Sechseläuten), Mi 22.4.

<http://nativesystems.inf.ethz.ch/WebHomeLecturesBetriebssysteme09>

Allgemeines zur Vorlesung

- Die Vorlesung ist obligatorischer Teil des Informatik Bachelorstudiums (2. Jahr)
- Das Thema sind konzeptuelle Grundlagen von Betriebssystemen, illustriert an gängigen Beispielen
- Vorausgesetzt werden gute Kenntnisse der Systemprogrammierung, z.B. entsprechend „Bryant & O'Hallaron, Computer Systems“
- Primäre Lehrziele sind Kenntnis und tieferes Verständnis des „State of the Art“ der Betriebssysteme im Zusammenhang
- Zu Beginn jeder Vorlesungsstunde werden Handouts zum jeweiligen Stoff verteilt.
- Die Vorlesung wird audiovisuell aufgezeichnet. Aufzeichnungen können im Internet heruntergeladen werden (=>Vorlesungshomepage).

Allgemeines zu den Übungen

- Übungen werden in Kleingruppen durch Assistenten bzw. Hilfsassistenten abgehalten.
- Sie dienen der Repetition und Ergänzung des Vorlesungsstoffes
- Sie werden wöchentlich als „Papierübungen“ gestellt. Neben theoretischen Fragen auch **praktische Übungen**.
- Lösungen in Schriftform können dem betreffenden Assistenten zur Korrektur abgegeben werden. Die regelmässige Abgabe dient zu Ihrer eigenen Kontrolle, sie ist **empfohlen** aber nicht verpflichtend.
- Lösungen deutsch oder englisch.

Zu den praktischen Übungen

- Die Übungsaufgaben werden schriftlich ausformuliert gestellt.
- Sie dienen der Illustration des Vorlesungsstoffs an realen Beispielen, sind aber trotzdem exemplarisch und eher einfach gehalten.
- Sie basieren auf einer Emulation des Betriebssystems A2 (Nachfolger des Active Object System) unter Windows, erhältlich per Download von der Vorlesungshomepage.
- Anleitung und Hilfestellung zu den Übungen auf der Website, im installierten Betriebssystem und natürlich in den Übungsgruppen.

Übungsgruppen

Zeit	Ort	Assistent
Montag 14-16	IFW A 34	Ulrike Glavitsch
Montag 14-16	IFW B 42	Sven Stauber
Montag 14-16	CAB G57	Philipp Bönhof
Mittwoch 15-17	CAB G 59	Florian Negele
Mittwoch 15-17 (Englisch)	CAB H 53	Roman Mitin

Allgemeines zur Prüfung

- Die Prüfung (Endterm) ist (hand-)schriftlich.
- Prüfungsstoff ist der gesamte bis zur Prüfung behandelte Stoff, inklusive (theoretischer und praktischer) Übungen.
- Es sind keine Hilfsmittel zugelassen.
- Falls eine Note von mindestes 4 erreicht wird, werden 6 Krediteinheiten gutgeschrieben, sonst keine.
- Eine gute Voraussetzung für den Erfolg ist der regelmässige & aufmerksame Besuch von Vorlesung & Übungen.

Termin:

Freitag, 5. Juni 2009

10.00-12.00 h

CAB G11/ CAB G61

Wiederholungsprüfung

für nicht bestandene

Prüflinge im September 09

Ziele, Erwartungen

Ziel der Vorlesung

- Prinzipielles Verständnis der Vorgänge in einem Betriebssystem.
- Aneignung von Standard-Lösungsstrategien.
- Überblick über und kritisches Beurteilen von bekannten Ansätzen.
- Einsicht in die Interna von Betriebssystemen.

Voraussetzung

- Programmierkenntnisse, am besten Systemprogrammierung (Literatur s.u.)

Begleitende Literatur

Stallings, William, Betriebssysteme, Prinzipien und Umsetzung, Prentice Hall/ Pearson Studium	Sehr ausführlich
Tanenbaum Moderne Betriebssysteme Hanser Studienbücher	Klassiker neue Auflage (2004) aktuell und sehr ausführlich
Glatz, Eduard, Betriebssysteme Grundlagen, Konzepte, Systemprogrammierung dpunkt.verlag	aktuell (1. Auflage 2006) ausführlich, mit Fokus auf Programmierung von Betriebssystemen (in C)
U. Baumgarten & H.J. Siegert Betriebssysteme, eine Einführung, Oldenbourg Verlag	aktuell (6. aktual. Aufl. 2007), eher knapp und prinzipiell
A. Silberschatz, P.B. Galvin, G.Gagne Operating System Concepts, 7th edition John Wiley & Sons	Klassiker, ausführlich und aktuell

Weiterführende Literatur

<p>R. Bryant and D. O'Halloran, Computer Systems: A Programmer's Perspective, Prentice-Hall, 2002</p>	Gut zur praktischen Vorbereitung: ausführliche Anleitung zur Systemprogrammierung
<p>R. Jones, R. Lins, Garbage Collection Algorithms for Automatic Dynamic Memory Management Wiley, 2003</p>	Spezialliteratur zum Thema Garbage Collection Standardreferenz Lit. zum Kapitel 4
<p>Russinovich, Solomon Microsoft Windows Internals Microsoft Press, 2005</p>	Hintergründe zu den Microsoft Windows Betriebssystemen
<p>Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel, 3rd Edition, O Reilly 2005. http://proquest.safaribooksonline.com/0596005652</p>	Hintergründe zum Linux Kern
<p>N.Wirth, J.Gutknecht Project Oberon – The design of an operating system and compiler. Vergriffen, download unter http://www-old.oberon.ethz.ch/WirthPubl/ProjectOberon.pdf</p>	Klassiker, interessant als Nebenlektüre. Beschreibt ein komplettes Betriebssystem incl. Sourcecode

Große Inhaltsübersicht

- Betriebssysteme
 - Ressourcenverwaltung
 - Sekundärspeicher
 - Hauptspeicher
 - Prozessoren
 - Systemarchitekturen
 - Kernstrukturen
 - Virtuelle Systeme
 - Überblick Fallstudien

aussen
↓
innen

Sicht des Benutzers
und des Programmierers

innen
↓
aussen

Sicht des Betriebssystem-
architekten

Kapitelverzeichnis und Literaturvergleich

Tanenbaum

- 1 Einführung
- 2 Prozesse & Threads
- 3 Deadlocks
- 4 Speicherverwaltung
- 5 Ein- und Ausgabe
- 6 Dateisysteme
- 7 Multimedia
 Betriebssysteme
- 8 Multiprozessorsysteme
- 9 IT-Sicherheit
- 10 Unix/Linux
- 11 Windows 2000
- 12 Entwurf von BS

Vorlesung Betriebssysteme

Einleitung

Kapitel 1. Bootstrapping

Kapitel 2. Filesysteme

Kapitel 3. I/O Subsysteme

Kapitel 4. Hauptspeicherverwaltung

Kapitel 5. Prozessorverwaltung

Kapitel 6. Betriebssystemarchitekturen

Kapitel 7. Fallstudien

Stallings

- 1 Hintergrund
- 2 Überblick über Betriebssysteme
- 3 Beschreibung und Steuerung von Prozessen
- 4 Threads, SMP und Mikrokernel
- 5 Wechselseitiger Ausschluss und Synchronisierung
- 6 Verklemmung und Verhungern
- 7 Speicherverwaltung
- 8 Virtueller Speicher
- 9 Scheduling bei Einprozessorsystem
- 10 Scheduling in Mehrprozessor systemen
- 11 E/A Verwaltung
- 12 Dateiverwaltung
- 13 Verteilte Verarbeitung, Client/Server, Cluster
- 14 Verwaltung verteilter Prozesse
- 15 Sicherheit

Glatz

- 1 Einführung
- 2 Grundlagen der Programmausführung und Systemprogrammierung
- 3 Prozesse und Threads
- 4 Synchronisation von Prozessen & Threads
- 5 Kommunikation von Prozessen und Threads
- 6 Ein- und Ausgabe
- 7 Speicherverwaltung
- 8 Dateisysteme
- 9 Programmierung
- 10 Sicherheit
- 11 Spezielle Technologien

Silberschatz

- 1 Introduction, OS Structures
- 2 Processes, Threads, CPU Scheduling,
 Process Synchronisation, Deadlocks
- 3 Main Memory, Virtual Memory
- 4 File Systems , Mass Storage Structure,
 I/O Systems
- 5 Protection, Security
- 6 Distributed Systems
- 7 Real Time Systems, Multimedia Systems
- 8 Linux System, Windows XP, Influential OSs
A Unix BSD, Mach System, Windows 2000

Motivation

- Warum beschäftigt man sich mit Betriebssystemen?
Genügt nicht die Expertise in Hardware-naher
Programmierung?
- Was sind die funktionalen Aufgaben eines
Betriebssystems und wo liegen die Schwierigkeiten?
- Wo liegen die Unterschiede in verschiedenen
Lösungsansätzen?
- (Wo) ist noch Forschungsbedarf?

Aufgaben eines Betriebssystems

Ein-Prozess Systeme (Batch Betrieb)

- Ausführen einer Applikation / eines Programms / eines Algorithmus
 - Startvorrichtung
Bootvorgang / Dynamisches Linken / Patchen von Adressen
- Vereinfachter Zugriff zur Hardware
 - Bibliotheken von Funktionen
Dateiverwaltung, Netzwerkservices etc.
 - Unterstützung der Programmiersprache (Runtime Support)
Garbage Collection, Module Loading
- Einheitlicher Zugriff zu unterschiedlicher Hardware
 - Einteilung des Addressraumes (E/A Controller, Speicher)
 - Verschiedene angeschlossene Geräte oder gar Bus-Systeme
 - ⇒ Gerätetreiber
- Benutzerschnittstelle
 - Textuelle / graphische Shell

~1955: kein Betriebssystem, Programme wurden ausschließlich manuell geladen

~1965: Stapelverarbeitung, Programme nacheinander von Lochkarten / Magnetbändern geladen

Aufgaben eines Betriebssystems

Mehrprozess-Systeme, Mehrbenutzer-Systeme, Multiprogramming

- Quasi-gleichzeitige **Rechenzeitzuteilung** an mehrere Prozesse
 - Beobachtung: CPU wartet oft auf Geräte. Nutze die Zeit für andere Prozesse.
Job Scheduling (Fairness, Priorisierung, Zeitscheibe), Preemption, Interprozesskommunikation (IPC), Fork
 - ⇒ **Gerätezuteilung** an mehrere Prozesse *DMA*
 - ⇒ **Speicherzuteilung** an mehrere Prozesse *Segmentation, Paging*
 - ⇒ Verwaltung von **Interrupts** *Kontextwechsel*
- **Systemzuteilung** an mehrere Nutzer
 - Teure Hardware wird gleichzeitig mehreren Nutzern zur Verfügung gestellt.
 - ⇒ Gegenseitiger **Schutz** der Nutzer und Prozesse: Verwaltung der **Betriebsmodi** des Prozessors, Speicherschutzmechanismen.
MMU, virtueller Addressraum, Segmentierung, Prozesskontext, Shared Memory, Message Passing, Zugriffsrechte

~1980: Einführung des Mehrprogrammbetriebs und Mehrbenutzerbetriebs

~1990: Rückkehr zu Einbenutzersystemen

Aufgaben eines Betriebssystems

Mehrprozessorsysteme / Multicore, Realtime

- Rechenzeitzuteilung auf mehreren Prozessoren
 - Verschiedene Rechnerarchitekturen, Cache-Kohärenz-Probleme
SMP, NUMA, Multicore, CPU-Scheduling
 - In Multi-Prozessor Systemen ist die **Synchronisation** von Prozessen schwieriger und wichtiger
Critical Section, Mutual Exclusion, Semaphore, Deadlock, Starvation, Monitor (Hoare),
- (Eingebettete) Realtime-Systeme
 - Systeme mit garantierten maximalen Antwortzeiten, unverzichtbar bei Steuerung von Geräten mit hohem Gefahrenpotential oder lebenserhaltenden Systemen.
Interaktive Systeme vs Realtime, Realtime Scheduling

~2000: zunehmende Vernetzung

~2005: zunehmende Parallelisierung, Multicore

Aufgaben eines Betriebssystems, zusammengefasst

Betriebssystem stellt bereit

- Abstraktion
 - API* für den Benutzer / Programmierer
 - Abstraktion von Geräten und Diensten
 - „Virtuelle Maschine“, Unabhängigkeit von jeweiliger Hardware
- Resourcenverwaltung und -Schutz
 - Zeit-Zuteilung  z.B. „batch processing“ ./ „time sharing“ / „real time“
Interrupts, Threads, Prozesse, DMA etc.
 - Raum-Zuteilung
Speicherverwaltung, Gerätezuteilung, DMA, etc.

Resource
Multiplexing

*API: Application Programming Interface
(Programmier-Schnittstelle zu den Applikationen)

Abgrenzung

Differenzierung des Begriffs Betriebssystem:

1. Betriebssystem im engeren Sinne

- System aus Sicht der Anwendungssoftware
- Software zwischen Hardware und Systemschnittstelle

2. Betriebssystem im weiteren Sinne

- System aus Sicht des Benutzers, d.h. Benutzerschnittstelle, Compiler, Dienstprogramme usw.

z.B. JIT Compiler
Thread-safe GUI etc.

Die Grenze ist unscharf.

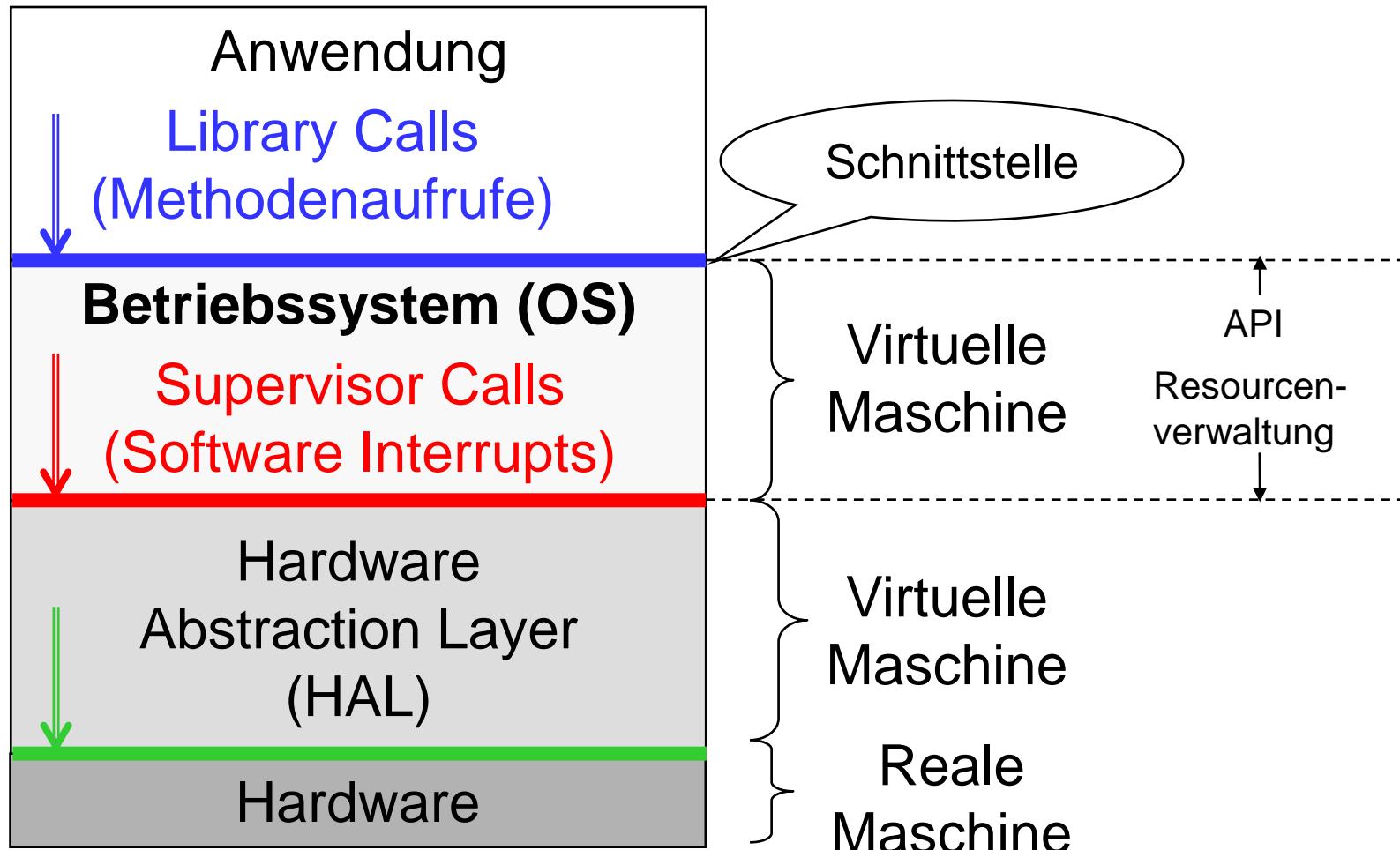
In dieser Vorlesung: weitgehend im engeren Sinne, so wie unter 1. bezeichnet.

Einordnung im Computersystem: Schichtenmodell

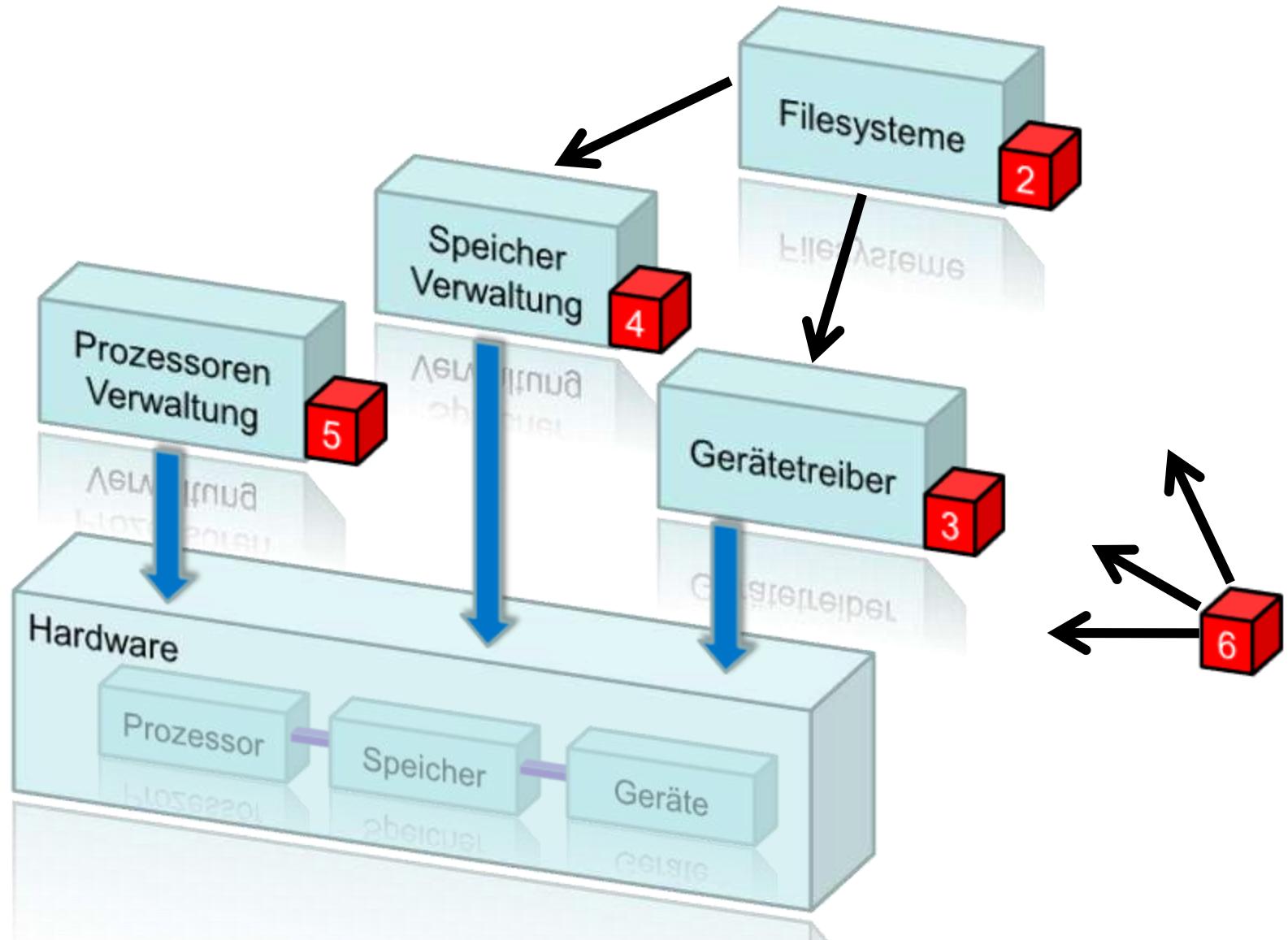
Application
Programming
Interface
(API)

Funktions
Tabelle

Instruktions
Satz (ISA) plus
Register



Einordnung im Computersystem: thematisch



Kapitel 1. Bootstrapping*

E. Glatz, Betriebssysteme, S. 567

- 1.1 Startup Prozedere
- 1.2 Disk Partitionierung
 - 1.2.1 Geometrie, Disk Hardware
 - 1.2.2 Partitionierung



*Bootstrap: Stiefelriemen

1.1. Startup Prozedere

- Systemstart
 - Kaltstart (Power ON / Reset Taste)
 - Warmstart (System Reset)
- Initialer Ladevorgang*
 - Die Hardware setzt den *Programmzähler (PC)* auf eine fixe Adresse im *Read Only Memory (EEPROM*)*

* Initial Program Load =IPL

** Electrically Erasable Programmable Read Only Memory

BIOS* (PC)



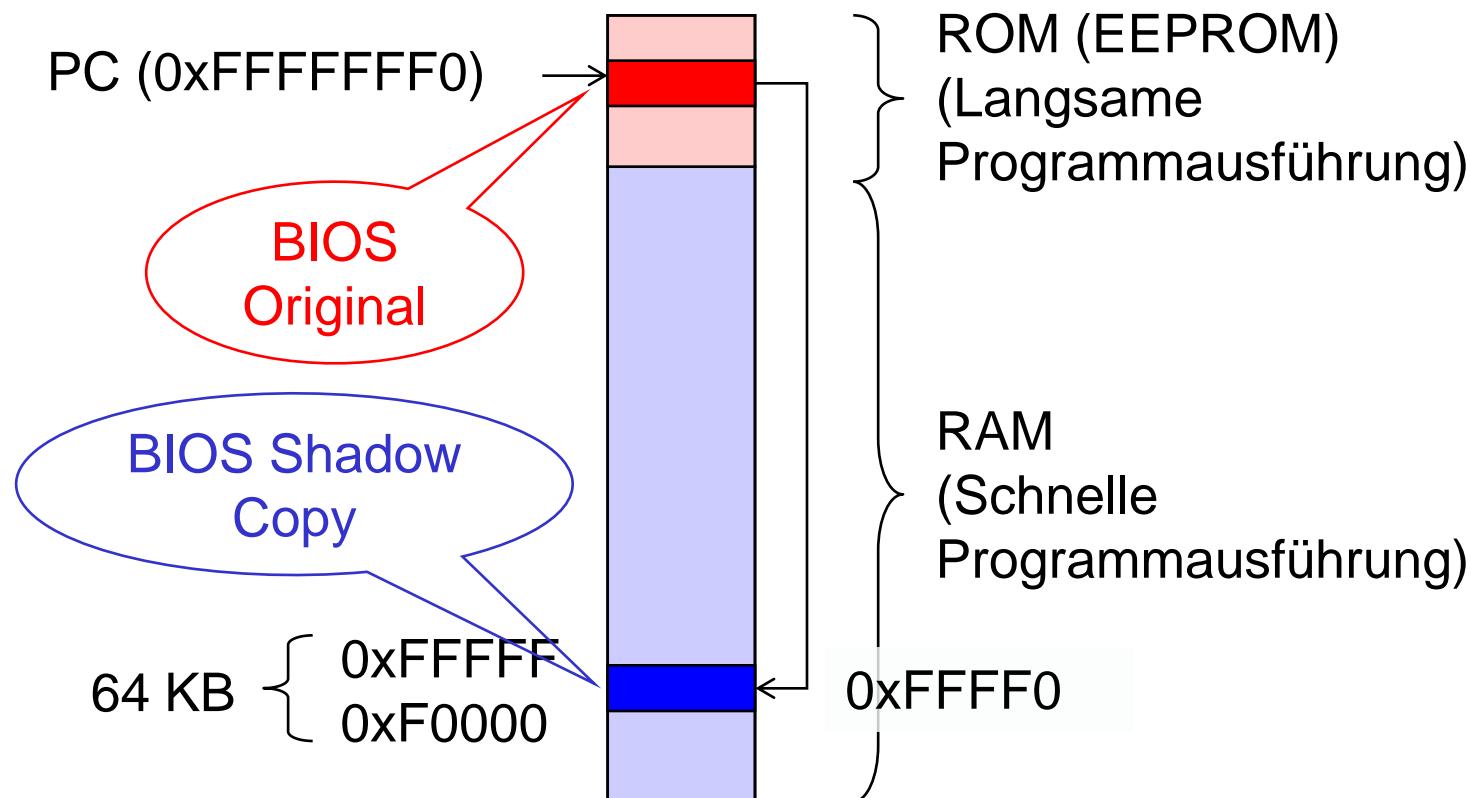
- BIOS Startup Programm (eingeführt 1981)
 - Power ON Self-Test (*POST*)
 - Initialisieren der Systemkonfiguration via CMOS** Speicher mit Batteriespeisung
 - Suchen eines Bootfiles und Laden in den Hauptspeicher
 - Sprung zum Eintrittspunkt in das Bootfile
 - Start im Real-Mode
 - Wechsel zum Protected-Mode durch das Betriebssystem
 - Nicht 64-bit tauglich

*Basic In- and Output-System

**Complementary Metal Oxide Semiconductor

BIOS HAL

- Shadowing im RAM



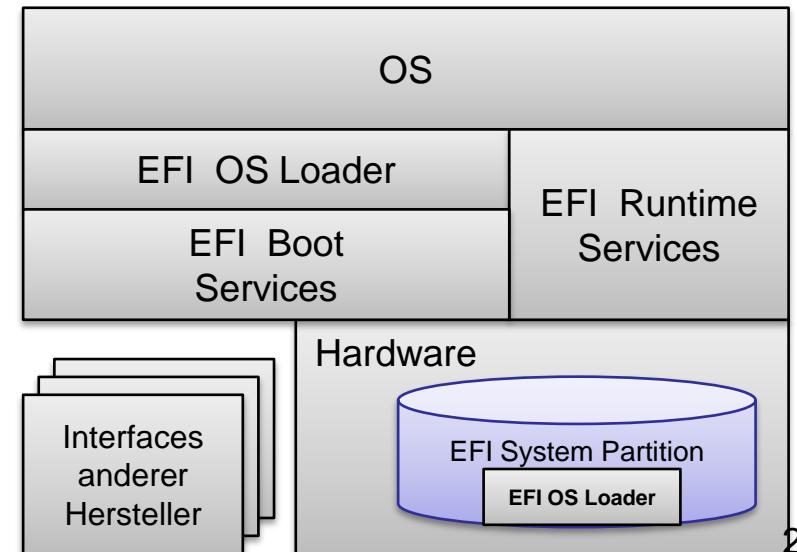
Bootfile Suchstrategie

- Boot Device Präferenzlisten
 - BIOS Boot Device Präferenzliste, z. B.
 - Hard Disk → CD Rom
 - CD Rom → Hard Disk
 - USB Memorystick → CD → Hard Disk → LAN
 - Hard Disk Boot Präferenzliste, z. B.
 - Primary Master ATA → ... → Secondary Slave ATA
→ SCSI
- Boot Device Suchstrategie
 - Traversiere Boot Devices gemäss Präferenzlisten bis ein gültiger *Master Boot Record (MBR)* gefunden wird

UEFI (Unified Extensible Firmware Interface)

bislang bekannt als EFI

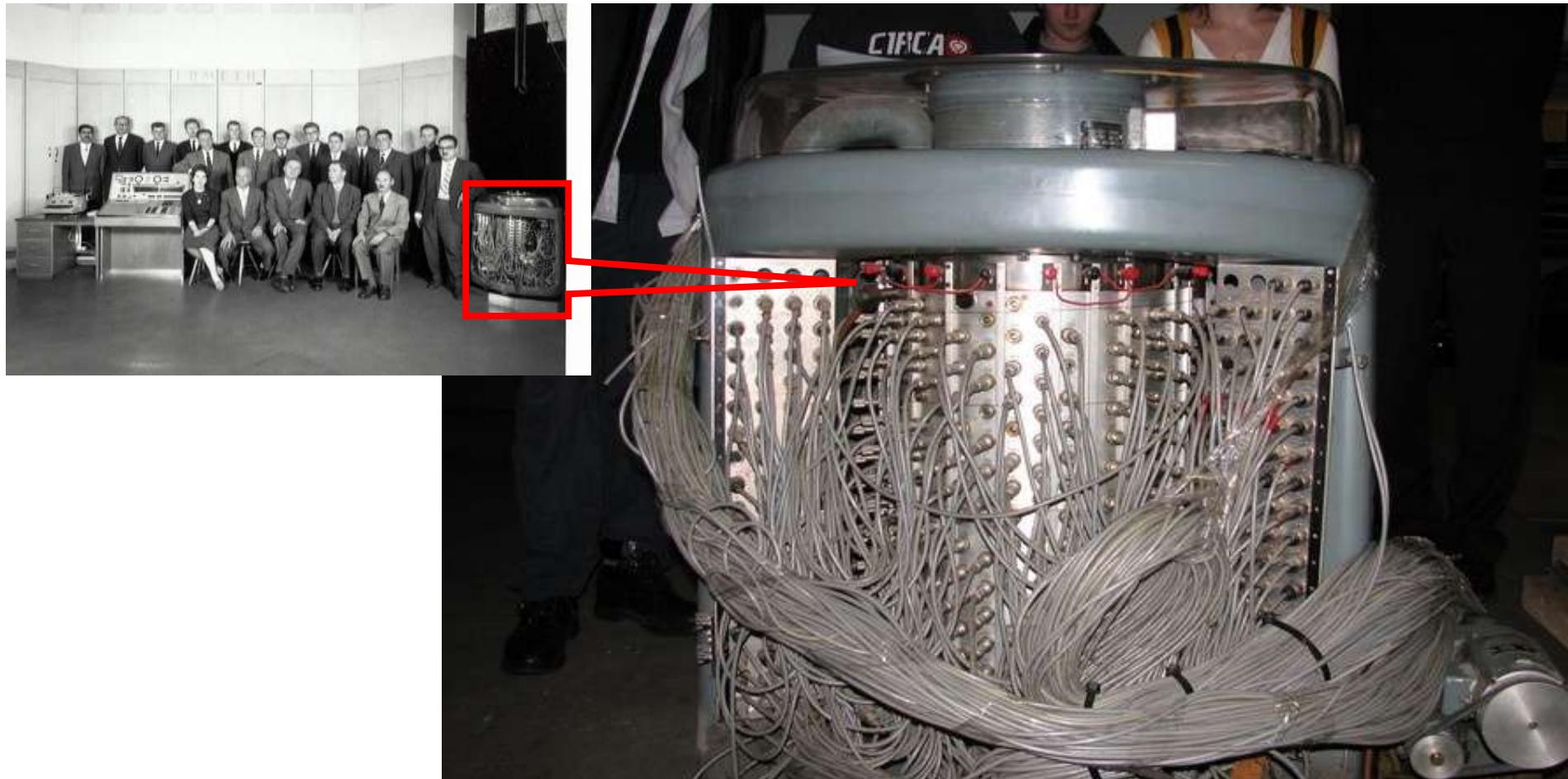
- Eingeführt anlässlich des Wechsels auf 64bit Systeme
 - 2000 Seiten Spezifikation, verantwortlich: Unified EFI Forum
- Software *Interface* zwischen Betriebssystem und Hardware
 - Ziel-Hardware-unabhängig
 - Device Drivers für das Pre-boot environment.
 - Ersetzen *nicht* die performanten OS-spezifischen Gerätetreiber.
 - UEFI definiert Bytecode für die Implementation hardware-unabhängiger Gerätetreiber
 - Bootmanager, Disk Support: Unterstützung der GUID Partitions Tabelle (s.u.), Dateisystem Support (FAT32), Textuelle und graphische Konsole
 - Erweiterungen: können von nicht-flüchtigen Speichern geladen und installiert werden
 - Unterstützung von Pre-Boot Applikationen
- Unterstützte Betriebssysteme
 - Linux seit 2000 (elilo)
 - HP/UX auf IA-64 seit 2002
 - MS Windows Server 2003 für IA64
 - Windows Vista vorr. nur in der 64-Bit Version
 - MacOS auf Intel-basierten Systemen



1.2. Partitionierung

- 1.2.1. Hardware
- 1.2.2. Partitionierung

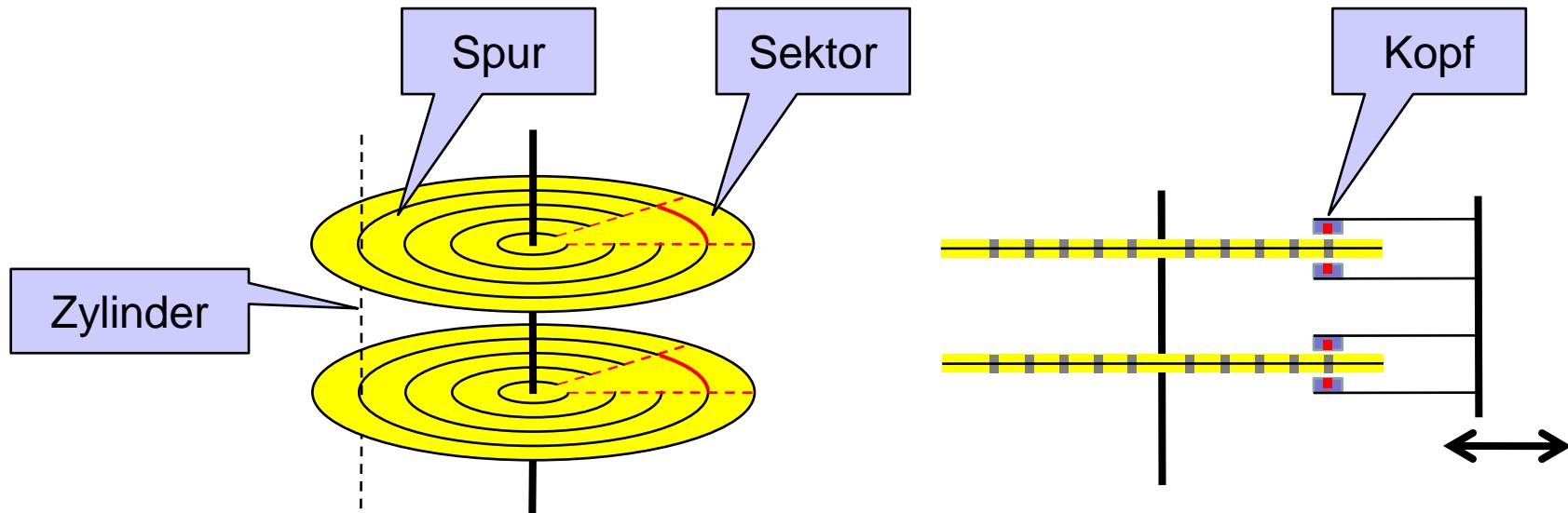
1.2.1. Hardware



Magnettrommel der Rechenanlage ERMETH (ETHZ) , Gewicht 1.5 Tonnen, in Betrieb 1957-1963
6000 U/Min, Zugriffszeit (av) 10ms, Abstand Magnetköpfe-Trommel: 1/100 mm
Speicherkapazität: 10000 Wörter zu 14 Dezimalziffern

Diskgeometrie

Tanenbaum Kap. 5.4.



(Logical) Block

Zylinder

Blockadresse

Blöcke im BIOS über INT13 ansprechbar – früher nur via CHS, heute auch LBA

Typisch: 512 Bytes

kleinste adressierbare Einheit

Menge Blöcke, die ohne Armbewegung angesprochen werden können

physikalisch: CHS (Cylinder, Head, Sector) oder

logisch: LBA (Logical Block Address)

$$\text{LBA} = ((\text{C} \times \text{HPC} + \text{H}) \times \text{SPT}) + \text{S} - 1$$

HPC : Leseköpfe/Zylinder (typisch: 8-255)

SPT : Sektoren/Spur (typisch: 63)

Sektoren nummeriert ab 1

Köpfe, Zylinder nummeriert ab 0

LBA nummeriert ab 0

Disks

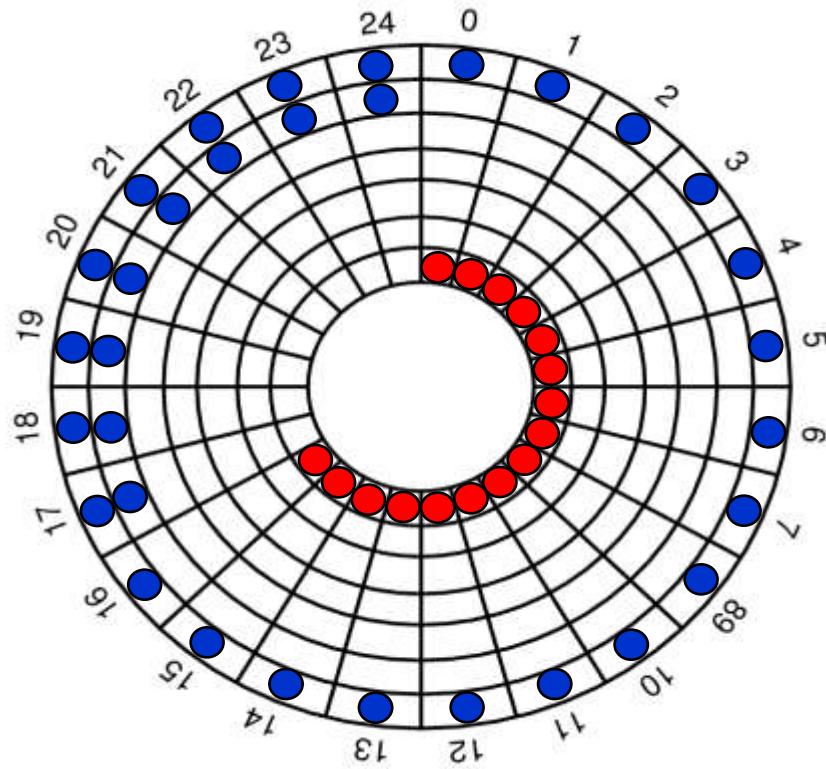
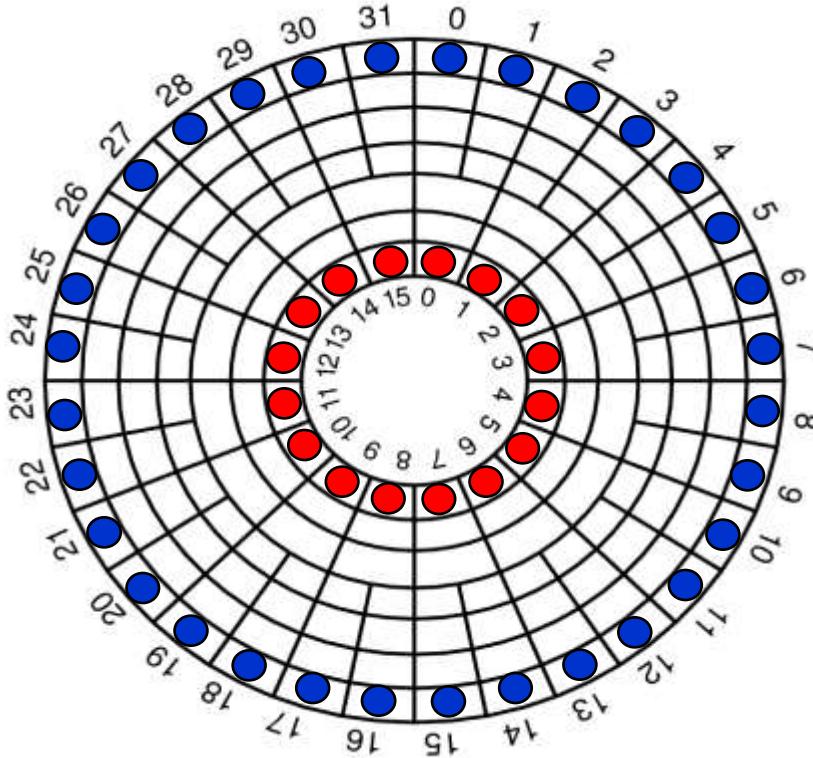
Disk Parameter

Parameter	IBM 360K Floppy Disk	WD 18300	Hitachi Deskstar 7K 1000 (derzeit (2.2009) CHF 109)
Anzahl Zylinder	40	1061	
Köpfe (Spuren/Zylinder)	2	12	10
Sektoren / Spur	9	281 (av)	
Sektoren / Disk	720	35.742.000	ca. 2^{30}
Bytes / Sektor	512	512	512
Kapazität	360KiB*	18.3GB*	1 TB*
Suchzeit (nebeneinander)	6 ms	0.8 ms	0.8 ms
Suchzeit (durchschnittlich)	77 ms	6.9 ms	8.5 ms
Rotationszeit (av.Latenz*2)	200 ms	8.33 ms	8.33 msec
Motor Start/Stop Zeit	250 ms	20 sec	
Transferzeit / Sektor (cached)	22 ms	17 µs	3.6 µsec 0.17 µsec (ATAPI**)

*beachte: binäre vs. metrische Einheit!

**ATAPI=Advanced Technology
Attachment with Packet Interface

Disk Geometrie



- Physikalische Geometrie mit 2 Zonen
- Mögliche zugehörige virtuelle Geometrie

SSD* vs. HDD**

- Vergleich (2/2009)

	SSD (OCZ Vertex Series SATA II 2.5" SSD, ca. 1000SFR)	HDD (WD Scorpio Black 2.5" 250GB 90 SFR)
Kapazität	250 GB 1GB ~ 4 SFR	250 GB 1 GB ~ 0.36 SFR
Schockfestigkeit	1500 G	350 G
Gewicht	77g	99 g
Zugriffszeit	< 0.1 ms	2-12 ms
Datenrate Lesen (MB/s)	bis zu 200 MB/s	bis zu 100 MB/s
Datenrate Schreiben (MB/s)	bis zu 160 MB/s	bis zu 100 MB/s
Stromverbrauch (Watt) (Standby / Leerlauf / Last)	0.5 / 2	0.25 / 0.8 / 2.5
Geräusch (dBA)	0	22-25
Haltbarkeit (MTBF)	1.5 Mio Stunden	k.A. (typischerweise < 1 Mio h)

- Folgerung: HDDs werden langfristig durch SSDs ersetzt werden

* solid state disk

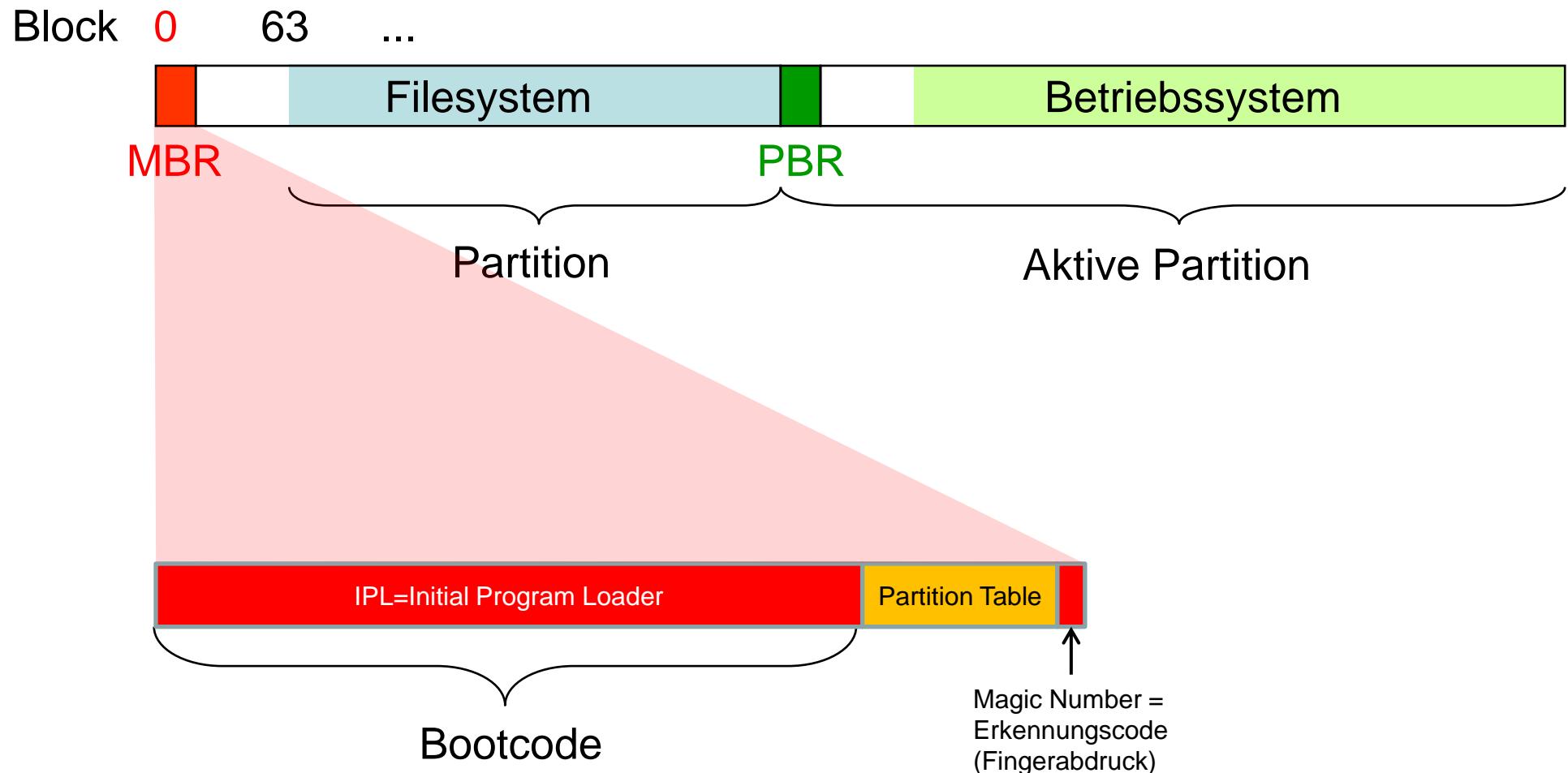
** hard disk drive

1.2.2. Partitionierung

Partitionen als logische Disk Drives

- Disk (physisches Laufwerk)
 - Menge disjunkter Partitionen (logische Laufwerke, Volumes)
- Partition
 - Serie aufeinanderfolgender Sektoren (Blöcke)
 - [start LBA, end LBA] (früher [start CHS, end CHS])
 - Typisiert
 - Filesystem / Betriebssystem
 - Partition Boot Record (PBR)
 - Data Volume
 - Device Driver
 - Swap Area
 - ...

PC Boot Disk Layout

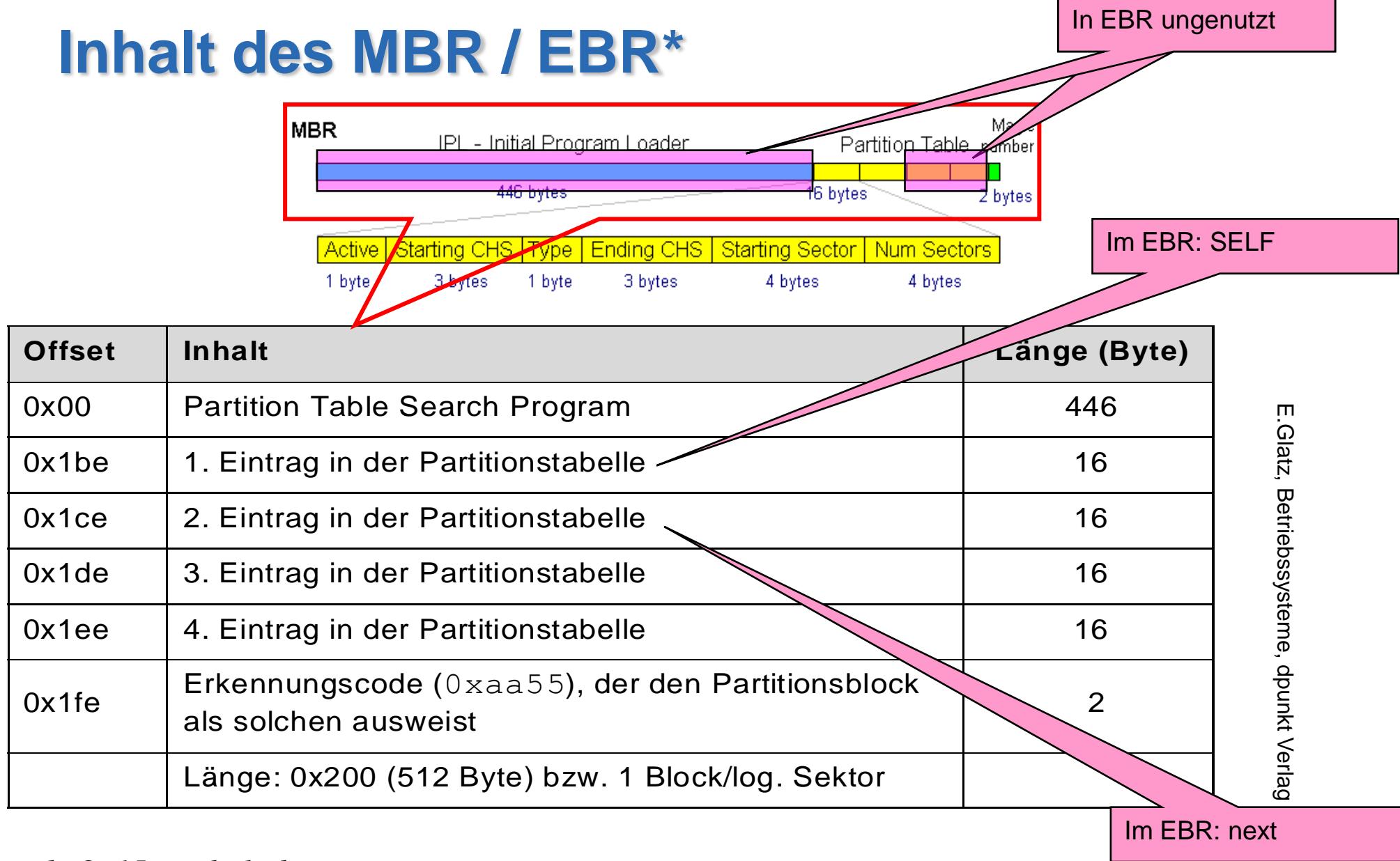


PC MBR

- Partitionstabelle
 - Maximal 4 Partitionen, davon maximal eine „erweiterte“ Partition mit maximal 24 Subpartitionen
 - Genau eine „aktive“ Partition als Boot Partition
- Bootcode
 - Inventarisiere alle Partitionen gemäss Partitionstabelle. Falls eine erweiterte Partition existiert, verfolge die Kette der Subpartitionen und ordne allen Volumes symbolische Namen zu
 - Bestimme die *aktive* Partition und lies den zugehörigen PBR in den Hauptspeicher
 - Springe zum Programmladdecode im PBR und führe ihn aus



Inhalt des MBR / EBR*

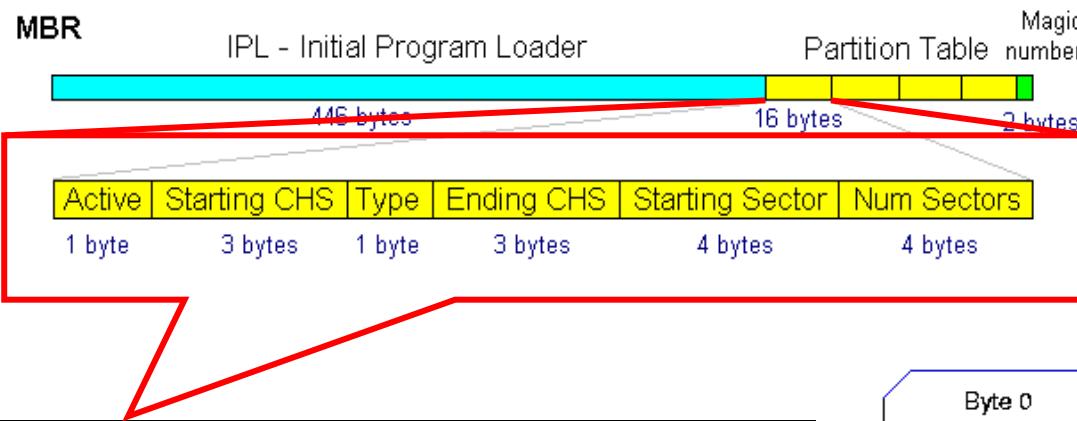


E.Glatz, Betriebssysteme, dpunkt Verlag

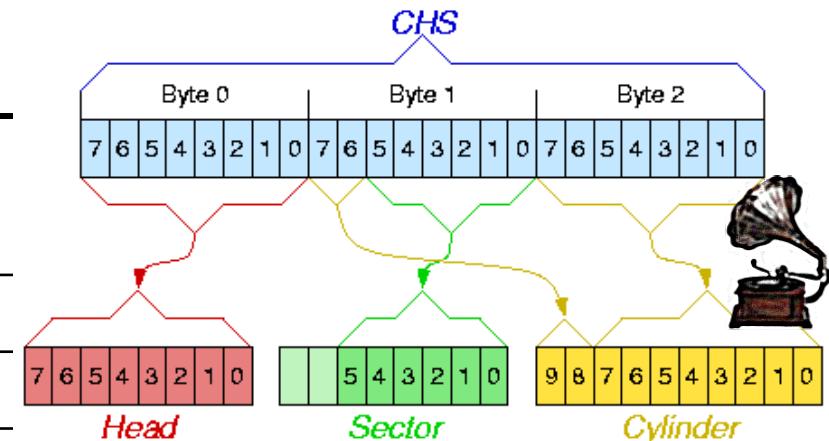
Tab. 8-15 Inhalt des MBR

*Master Boot Record / Partition Boot Record / Extended B.R.

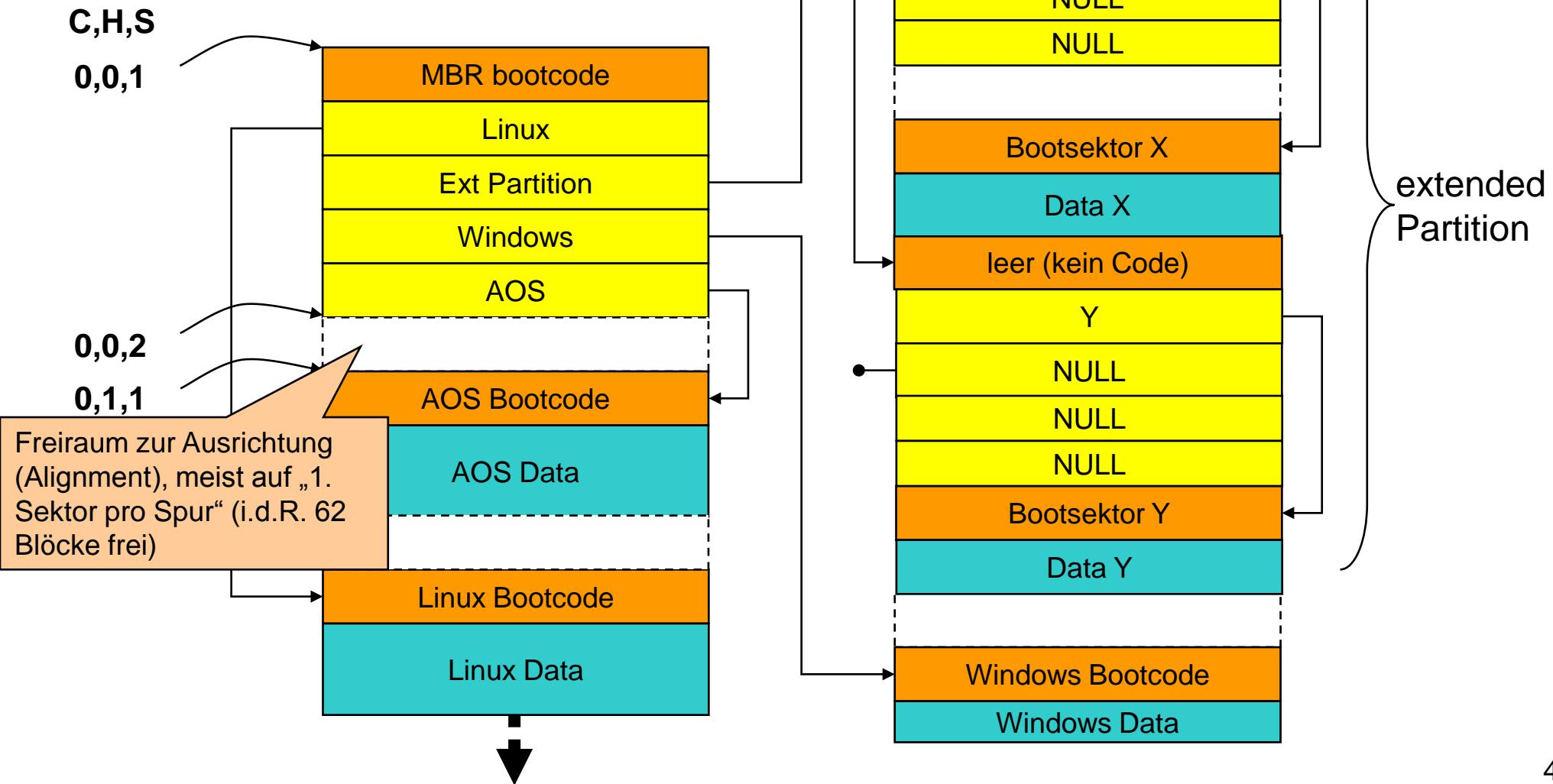
Eintrag in der Partitionstabelle



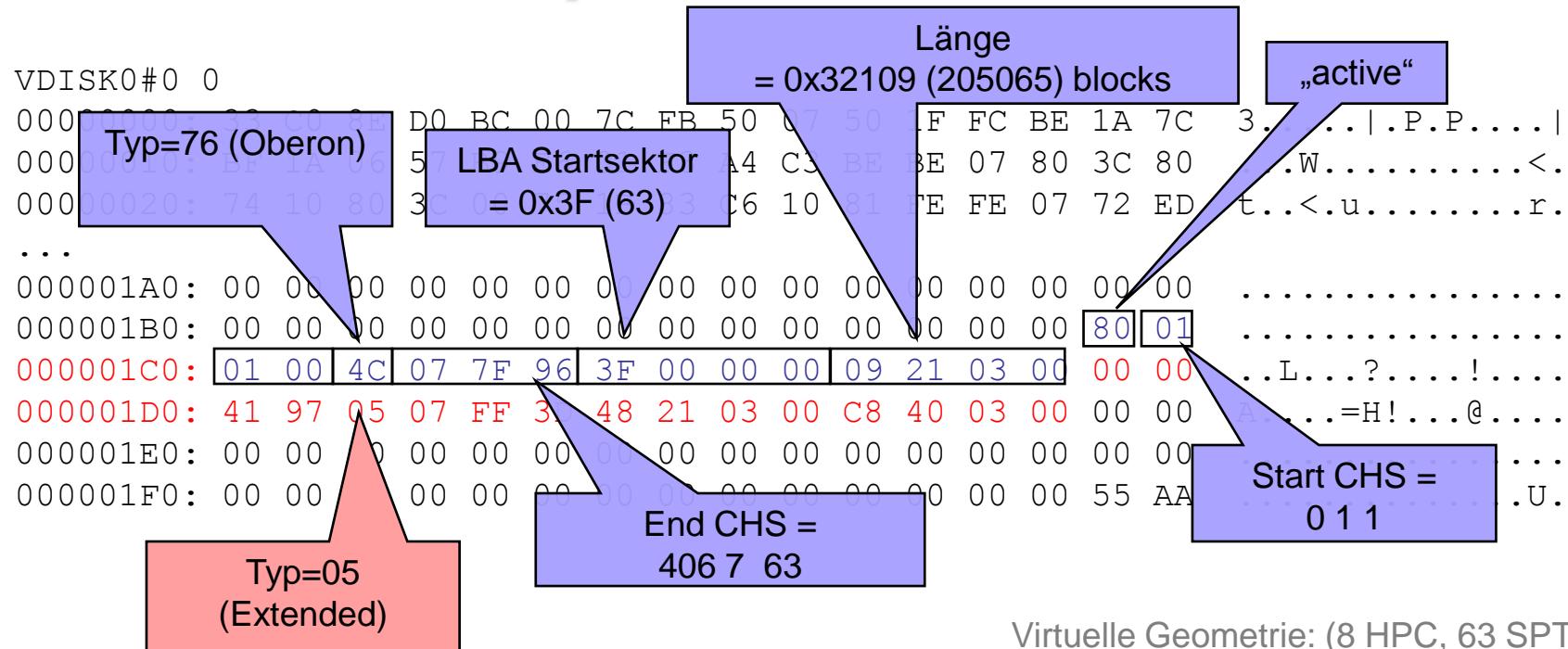
Adresse	Größe (Bytes)	Inhalt
0x00	1	bootable flag
0x01	3	CHS-Eintrag erster Sektor
0x04	1	Partitionstyp
0x05	3	CHS-Eintrag letzter Sektor
0x08	4	Startsektor [ab Anfang Festplatte oder erweiterte Partition]
0x0C	4	Anzahl der Blöcke in der Partition



Beispiel einer Partitionierung (abstrahiert)



Konkretes Beispiel: MBR



VDISK0 (Virtual Disk for file D:/VMTest/AosVM/Waos-f001.vmdk), Size: 205 MB, Open count: 0, CHS: 832x8x63

Partition	Start	End	Size	Type	Flags
VDISK0#0	0	419429	205 MB	256 (Whole disk)	[V]
VDISK0#1	63	205127	100 MB	76 (Native Oberon, Aos)	[V][P][B]
VDISK0#2	205128	418319	104 MB	5 (Extended)	[V][P]
VDISK0#3	205191	410255	100 MB	76 (Native Oberon, Aos)	[V]
VDISK0#4	410319	418319	4000 KB	76 (Native Oberon, Aos)	[V]
VDISK0#5	418320	418823	252 KB	-1 (Unallocated)	[P]

Tools zum Editieren von Partitionstabellen

Partition Tool

PhysicalDrive0 (Windows Disk), readonly, Size: 305243 MB, Open count: 0, CHS: 38913x255x63

Partition	Start	End	Size	Type	Flags
PhysicalDrive0#0	0	625137344	305243 MB	256 (Whole disk)	[V]
PhysicalDrive0#1	63	102398309	49999 MB	7 (NTFS, HPFS, QNX, Adv. Unix)	[V][F]
PhysicalDrive0#2	307194930	625137344	155245 MB	15 (Extended LBA)	[V]
PhysicalDrive0#3	307194993	368627489	29996 MB	11 (Win 95/98, FAT32)	[V]
PhysicalDrive0#4	102398310	307194929	99998 MB	-1 (Unallocated)	[F]
PhysicalDrive0#5	368627553	625137344	125249 MB	-1 (Unallocated)	

VDISK0 (Virtual Disk for file D:/AOSTest/aostest-flat.vmdk), Size: 512 MB, Open count: 0, CHS: 520x32x63

Partition	Start	End
VDISK0#0	0	104
VDISK0#1	63	41
VDISK0#2	411264	82
VDISK0#3	822528	104

Computer Management

File Action View Window Help

Volume Layout Type File System Status Capacity Free

(C:)	Partition	Basic	NTFS	Healthy (System)	48,83 GB	36,24 GB	/4 %	No	0%
DATA (D:)	Partition	Basic	FAT32	Healthy	29,28 GB	10,47 GB	35 %	No	0%

UID Operation Device Status

No operations

Show Details Refresh FSTools Pa

Mount Prefix Auto0 FileSystem

Unmount Force File System Info: n/a

Ready

Expert Partitioner

Device	Size	F	Type	Mount	Start	End	Us
/dev/sda	93.1 GB		TOSHIBA-MK1032GS		0	12160	
/dev/sda1	10.0 GB		HPFS/NTFS	/windows/C	0	1305	
/dev/sda2	71.2 GB		Extended				1306 10612
/dev/sda3	2.0 GB		Win95 FAT32 LBA	/windows/D	10613	10874	
/dev/sda4	9.8 GB		Win95 FAT32 LBA	/windows/E	10875	12160	
/dev/sda5	23.7 GB		HPFS/NTFS	/windows/F	1306	4408	
/dev/sda6	1.0 GB		Linux swap	swap	4409	4539	
/dev/sda7	29.9 GB		Linux native	/	4540	8455	
/dev/sda8	14.6 GB		Linux native	/home	8456	10369	
/dev/sda9	1.0 GB		unknown				10369 10500
/dev/sda10	878.5 MB		unknown				10501 10612

Create Edit Delete Resize

LVM... RAID... Crypt File... Expert...

Disk 0

Basic 298,09 GB Online

(C:) 48,83 GB NTFS Healthy (System)

97,65 GB Unallocated

DATA (D:)

29,29 GB FAT32 Healthy

122,31 GB Free space

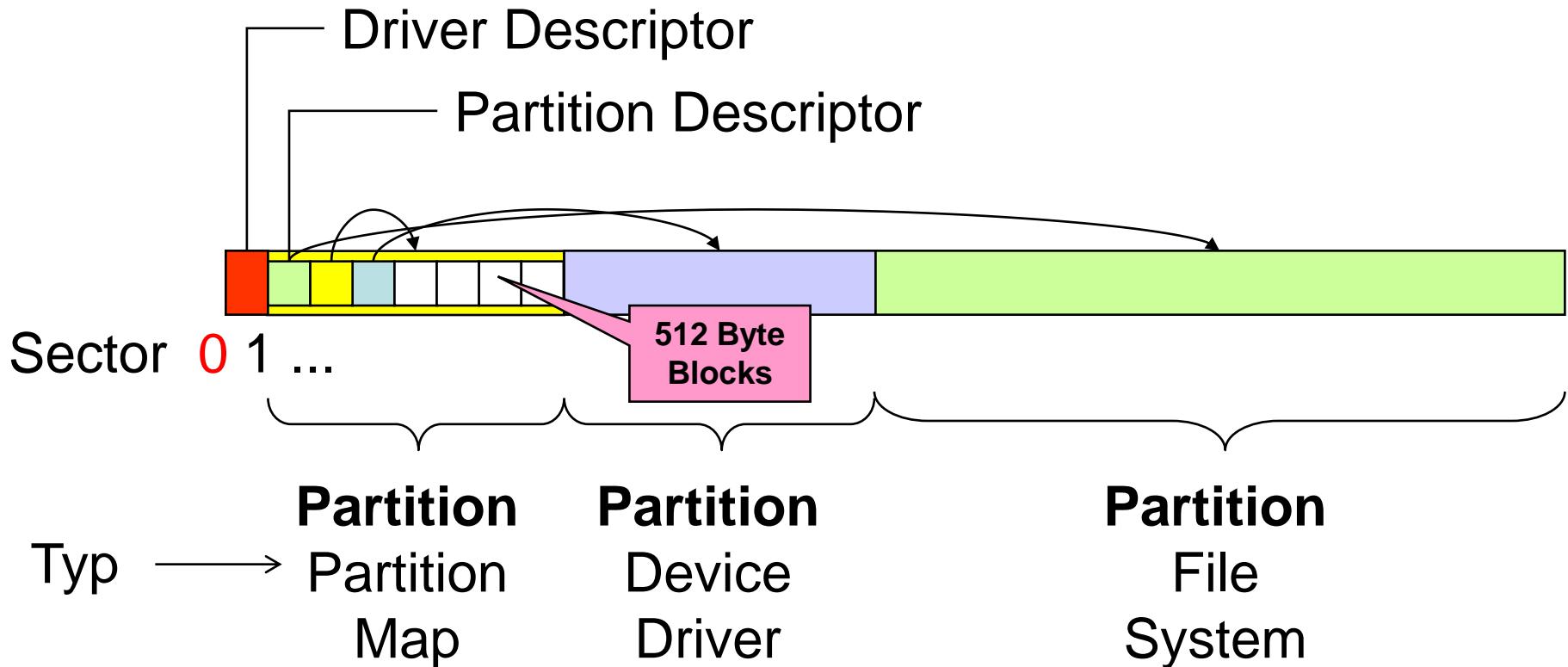
CD-ROM 0

DVD (X:)

No Media

■ Unallocated ■ Primary partition ■ Extended partition ■ Free space ■ Logical drive

Macintosh Disk Layout APM*



*Apple Partition Map

Beispiel: APM Entry Record

```
TYPE Partition= RECORD
```

```
    pmSig: Integer; {Partition Signature}
```

```
    pmSigPad: Integer; {reserved}
```

```
    pmMapBlkCnt: LongInt; {number of blocks in partition map}
```

```
    pmPyPartStart: LongInt; {first physikal block of partition}
```

```
    pmPartBlkCnt: LongInt; {number of blocks in partition}
```

```
    pmPartName: PACKED ARRAY [0..31] OF CHAR; {partition name}
```

```
    pmParType: PACKED ARRAY [0..31] OF CHAR; {partition type}
```

```
    pmLgDataStart: LongInt; {first logical block of data area}
```

```
    pmPartStatus: LongInt; {partition status information}
```

```
    pmDataCnt: LongInt; {number of blocks in data area}
```

```
    pmLgBootStart: LongInt; {first logical block of boot code}
```

```
    pmBootSize: LongInt; {size of boot code, in bytes}
```

```
    pmBootAddr: LongInt; {boot code load address}
```

```
    pmBootAddr2: LongInt; {reserved}
```

```
    pmBootEntry: LongInt; {boot code entry point}
```

```
    pmBootEntry2: LongInt; {reserved}
```

```
    pmBootCksum: LongInt; {boot code checksum}
```

```
    pmProcessor: PACKED ARRAY [0..15] OF Char; {processor type}
```

```
    pmPad: ARRAY [0..187] OF Integer; {reserved}
```

```
END;
```

Partitions-Information

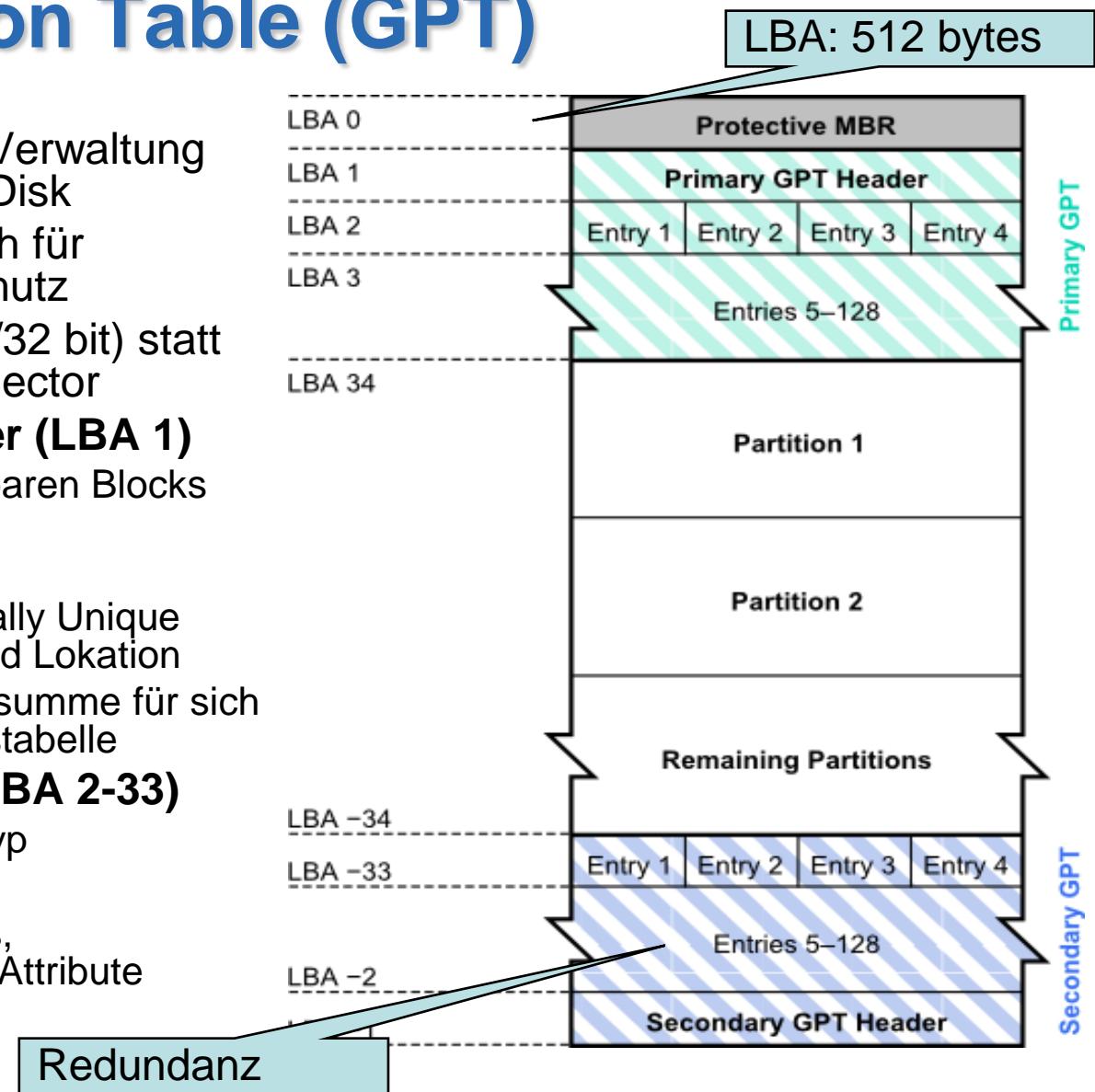
File-System bezogen

Boot-Code bezogen

Prozessor bezogen

GUID* Partition Table (GPT)

- GPT benutzt EFI zur Verwaltung und Bootloading von Disk
- **MBR Eintrag** nur noch für Kompatibilität und Schutz
- GPT benutzt LBA (64/32 bit) statt CHS Cylinder-Head-Sector
- **Partition table header (LBA 1)**
 - definiert die benutzbaren Blocks
 - Zahl und Größe der Partitionseinträge
 - Enthält GUID (Globally Unique Identifier), Größe und Lokation
 - Enthält CRC32 Prüfsumme für sich selbst und Partitionstabelle
- **Partitionseinträge (LBA 2-33)**
 - 16 Bytes Partitionstyp
 - 16 Bytes GUID
 - Start- und end LBAs, Partitionsname und Attribute



*Globally Unique IDentifier

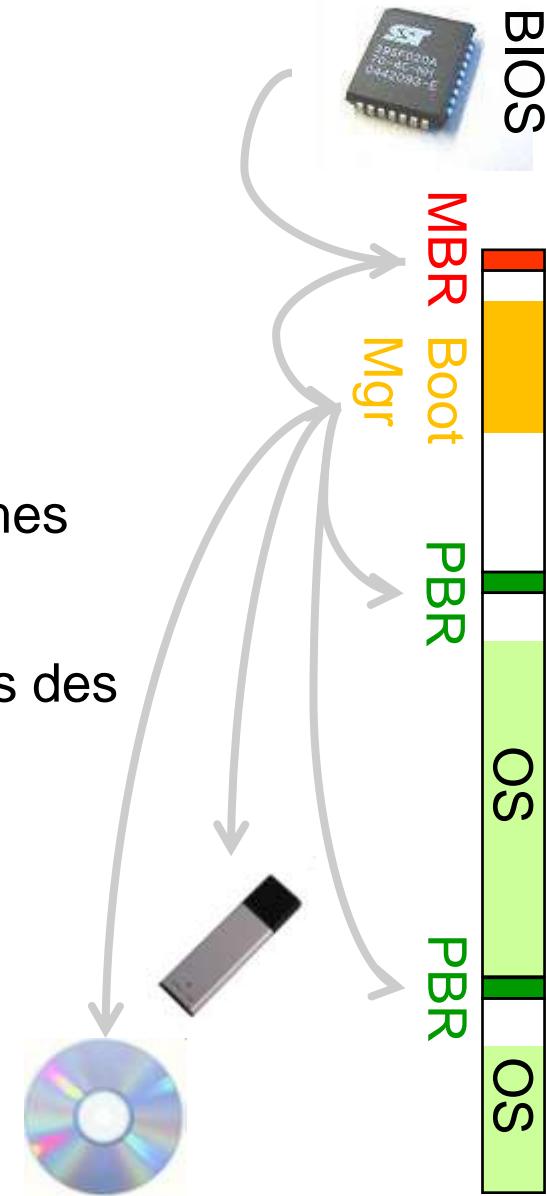
Bootmanager

- Ermöglichen die interaktive Spezifikation der aktiven Partition, z. B. In Dualbootsystemen
- Sind im MBR bzw. in einem bestehenden Betriebssystem integriert oder standalone als eigenes Betriebssystem ausgestaltet
- Boot Beispiel mit stand-alone Bootmanager
 - Phase 0: BIOS Startup Programm
 - Phase 1: Bootcode MBR
 - Phase 2: Bootcode PBR des Bootmanagers
 - Phase 3: Bootcode PBR des Betriebssystems

Booten mit Bootmanager

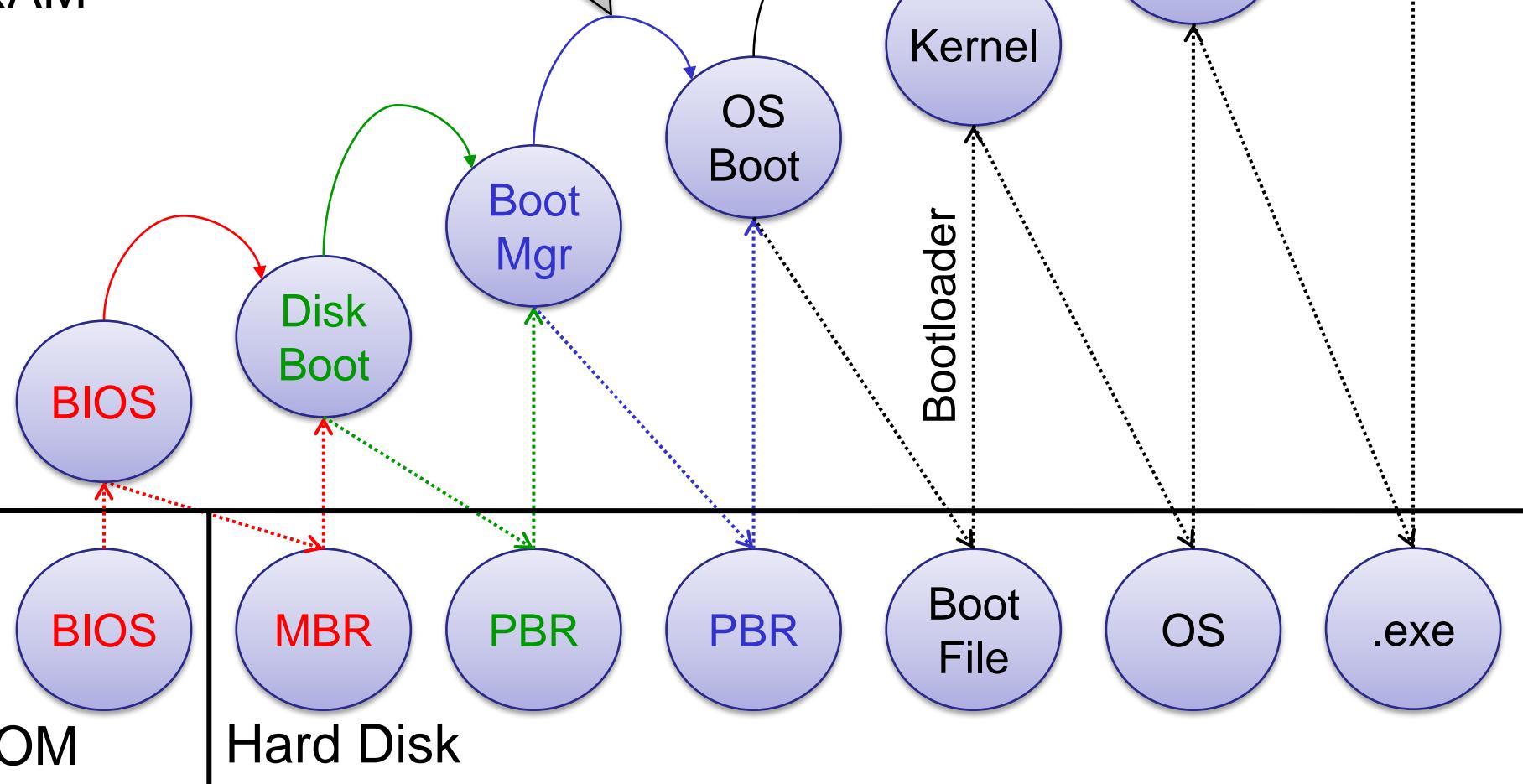
Unterteilung in drei Phasen

- Maschinen bezogen
 - System (BIOS) lokalisiert primäres bootbares Gerät
 - Auffinden des primären Bootvolumes
 - Übergabe der Kontrolle an den MBR des Bootvolumes
- Bootmanager bezogen
 - Code im MBR lädt Bootmanager von anderen Teiles des Bootvolumes nach
 - Textuelle oder graphische Selektion verschiedener Betriebssysteme
 - Benutzer selektiert die zu startende Partition
- Betriebssystem bezogen
 - PBR übergibt Kontrolle an das Betriebssystem



Kaskadierter Bootvorgang

RAM

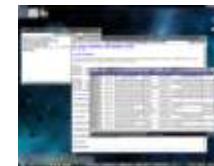


Nebenbemerkung: Prozessor Modes

beim Systemstart

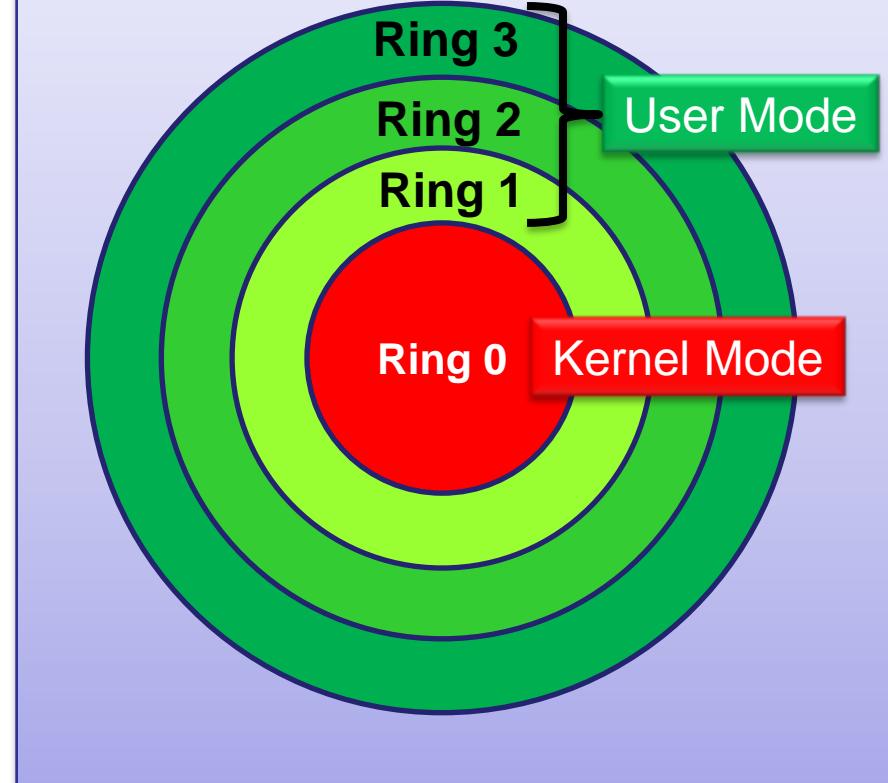


Real Mode



SO

Protected Mode

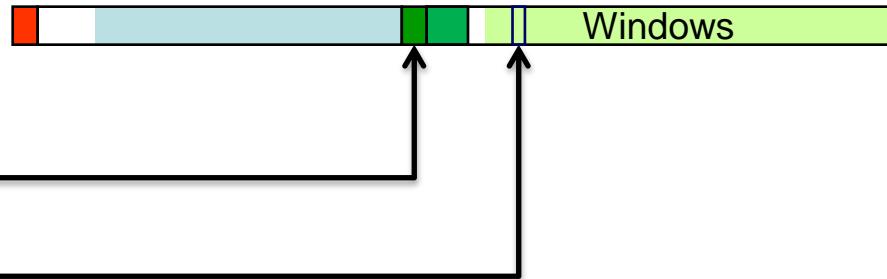


Anhang zum Kapitel 1: Beispiele

- Booten von Windows
- Booten von Linux
- Booten von A2

Booten von WinNT/2000/XP (I)

Tanenbaum, S.863, Glatz, S. 117



Initiale Phase

- Lade Windows Boot-Sektor
- Laden und Ausführen der Datei ntldr im Wurzelverzeichnis.

Boot Loader Phase

- Initialisiere 16 MB Adressumsetzungstabellen
- Mini Dateisystem, Laden der Bootkonfiguration
- Hardware Untersuchung Eintrag in Registry
- Laden des Hardware Profils und Initialisierung der Boot Geräte Treiber

Windows Registry

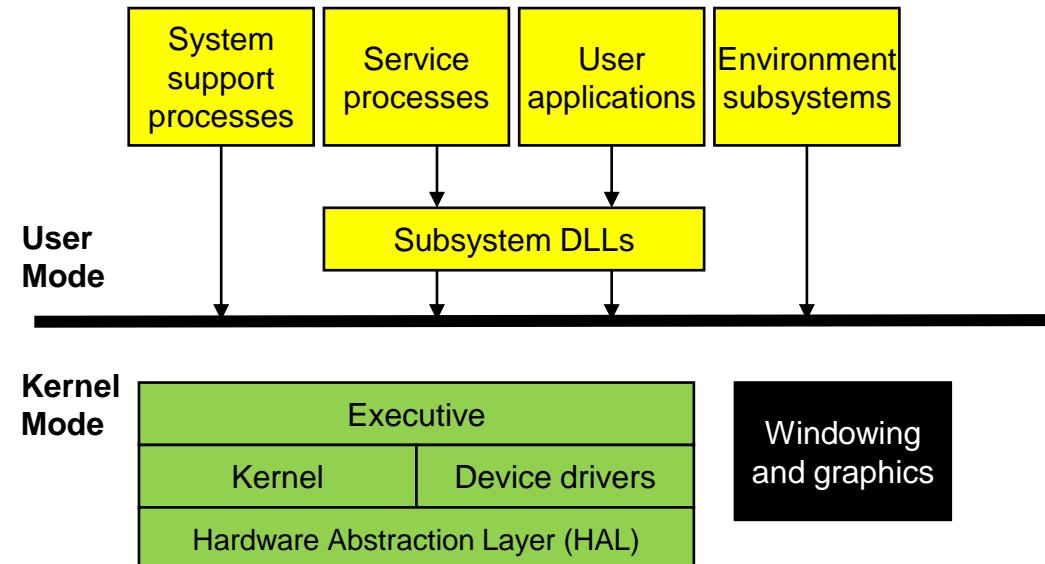
- enthält Konfigurations- und Statusinträge für Bootprozess, für den Loginprozess und das laufende System
- Strukturiert wie ein Filesystem:
 - Keys ~ Verzeichnisse
 - Werte ~ Files
 - Datentyp ~ Filetyp
 - Daten ~ Inhalt des File
- gespeichert in *Hives*

Booten von WinNT/2000/XP (II)

Tanenbaum, S.863, Glatz, S. 117

Kern Phase

- Systemkernstart, Kern-Initialisierung, Start des Idle-Prozesses, Einschalten der Interrupts,
- Boot Video Treiber, Startup („Splash“) Screen, Initialisierung der restlichen Treiber
- Session-Manager, Zero-Page Thread, Win32-Umgebung, Client/Server Runtime System
- Vaterprozess aller Dienste: Service Control Manager, startet Hintergrundprozesse im Benutzerraum (spezifiziert in Registry)



Logon Phase

- Anmelden der Benutzer durch Authentifizierungs-Manager

Windows x86-/x64 Boot Komponenten

Russinovich, Solomon: Microsoft Windows Internals, 4th edition, S. 251 ff.

Komponente	Ausführungsmodell	Aufgabe
MBR code	16 bit real-mode	Lade PBR
Boot sector	16 bit real-mode	Lese root directory (FS-Unterstützung!)
Ntdetect.com	16 bit real mode	Hardware Erkennung
Ntldr	16 bit real-mode → 32/64 bit protected mode, aktiviere Paging	boot.ini → Boot Menü Lade Ntoskrnl.exe, Bootvid.dll, Hal.dll, Boot Gerätetreiber
Ntbootdd.sys	protected mode	Gerätetreiber für Disk-I/O auf SCSI and ATA- Systemen
Ntoskrnl.exe	protected mode mit Paging	Initialisiere Executive Subsystem, Boot- und Start- Gerätetreiber, Vorbereitung für den Start von Applikationen, starte Smss.exe
Hal.dll	protected mode mit Paging	Kernel-mode DLL, Interface zwischen Hardware- Gerätetreibern und Ntoskrnl
smss.exe	Windows native application	Lade Windows Subsystem, Winlogon Prozess
winlogon.exe	Windows native application	starte Service Control Manager (SCM), starte Local Security Subsystem (LSASS), logon dialog
Service Control Manager	Windows native application	Lade und initialisiere Autostart -Gerätetreiber und Windows services

Bootkomponenten von Windows Vista

BIOS → Master Boot Record (Real Mode)	EFI → 32Bit executable (PE-File Format)
Boot Sector Windows Boot Manager (Bootmgr) lokalisiert in Boot\Bcd	Windows Boot Manager (Bootmgfw.efi) lokalisiert in EFI System Partition protected mode (ohne paging)
	ersetzt ntldr
liest Boot Configuration data (BCD) gespeichert wie eine Windows registry hive	
Suche Hibernation File	
winload.exe OS boot loader lädt den OS Kern und Boot Gerätetreiber	
ntoskrnl.exe	
smss.exe	
winlogon.exe → starte Services und das Login Interface	

Booten von Unix / Linux (I)

Assembler Code

- boot* wird von Platte geladen, kopiert sich selbst an hohe Speicheradresse
- boot liest Wurzelverzeichnis des Boot-Geräts, liest Kern ein und startet diesen,
 - boot muss Dateisystem verstehen oder Kern muss direkt in Sektoren gespeichert sein
- Startcode des Kerns in Assembler: Stack-Bereitstellung, Erkennung der CPU, Speicher, IRQ, MMU, Bus, Maus, APM**, geht in den Protected Mode
- **Sprung zum C-Code**

* Lilo oder Grub bei Linux

** Advanced Power Management

Booten von Unix / Linux (II)

C-Code

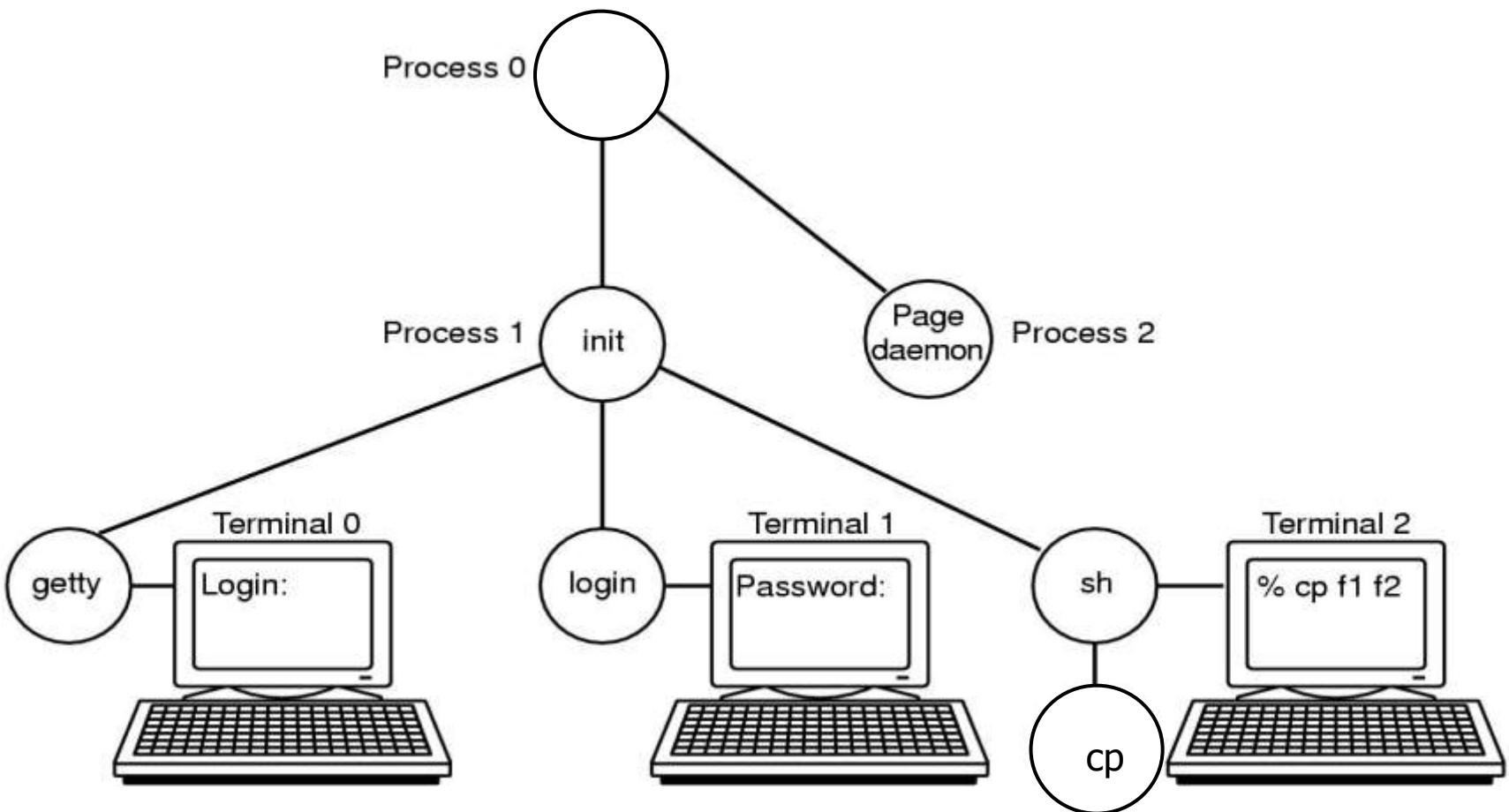
- Aufruf von `main`-Funktion (C-Code)
- Logische Initialisierung, Allokation der Kern-Datenstrukturen
- Autokonfiguration, Geräteliste, Gerätetreiber (dynamisch oder statisch, Abhängig von Unix-Version).
- Bereitstellung der Ablaufumgebung für Prozesse, Prozess läuft als Prozess 0 weiter: Einrichten der Echtzeituhr, Mounten des Wurzelverzeichnisses,
- Erzeugen von `init` (Prozess 1) und Page-Daemon (Prozess 2) durch Aufruf von `fork()`, Prozess 1 läuft im Benutzermodus weiter

Erster Start des Benutzermodus

- Prozess 1 ruft `exec()` auf und führt `/etc/init` aus
- `init`-Prozess liest `/etc/inittab` und startet Prozesse darin, darin ist auch das `login` Prozedere spezifiziert
- Prozess 0 wird zum Swapper Prozess

*Lilo oder Grub im Falle von Linux

Booten von Unix / Linux (III)



Reihenfolge der Prozesse beim Booten

Booten von A2 (I)

Bootcode in Assembler (BBL.Asm)

- Lade den Bootloader und Konfigurationsdaten
(Lokalisierung: nächste Folie)
- Lokalisiere und lade den Kern
- Einstellen des Video-Modes
- Eintritt in den Protected Mode
- Springe zu 32-bit Kerncode
(Oberon)

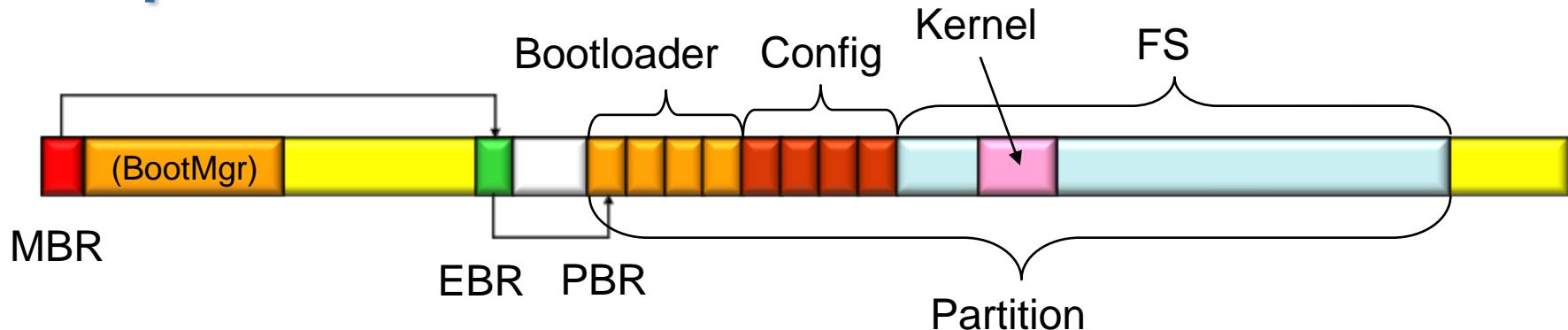
```
; Enter 80286 protected mode

smsw ax      ; store machine status word
or al,1      ; set Protection Enable bit (PE)
lmsw ax      ; load machine status word
jmp short kd0; flush instruction (prefetch) queue
...
; jump to kernel

mov esp,edx
pop edi      ; kpar1
pop esi      ; kpar0
pop eax      ; boot table linear address
db 066h      ; prefix followed by jmp opcode
retf         ; jump to kernel via Boot.Mod
```

Ausschnitt aus dem Oberon Bootloader

Beispiel: A2 Bootloader in erweiterter Partition



Booting Oberon

- Lade einen Block (512 Bytes) von Disk: 1. Teil des Bootloaders
- Reloziere Bootloader im Speicher
- Inspiziere Disk Parameter Tabelle
- Lade Rest des Bootstrap Loaders (+3 Sektoren)
- Lade Konfigurationsdaten (4 Sektoren)
- Lokalisiere und lade den Kern (Dateisystem!)
- Start des Kerns durch Einsprung in die gelinkten Modulbodies.

A2 Kern

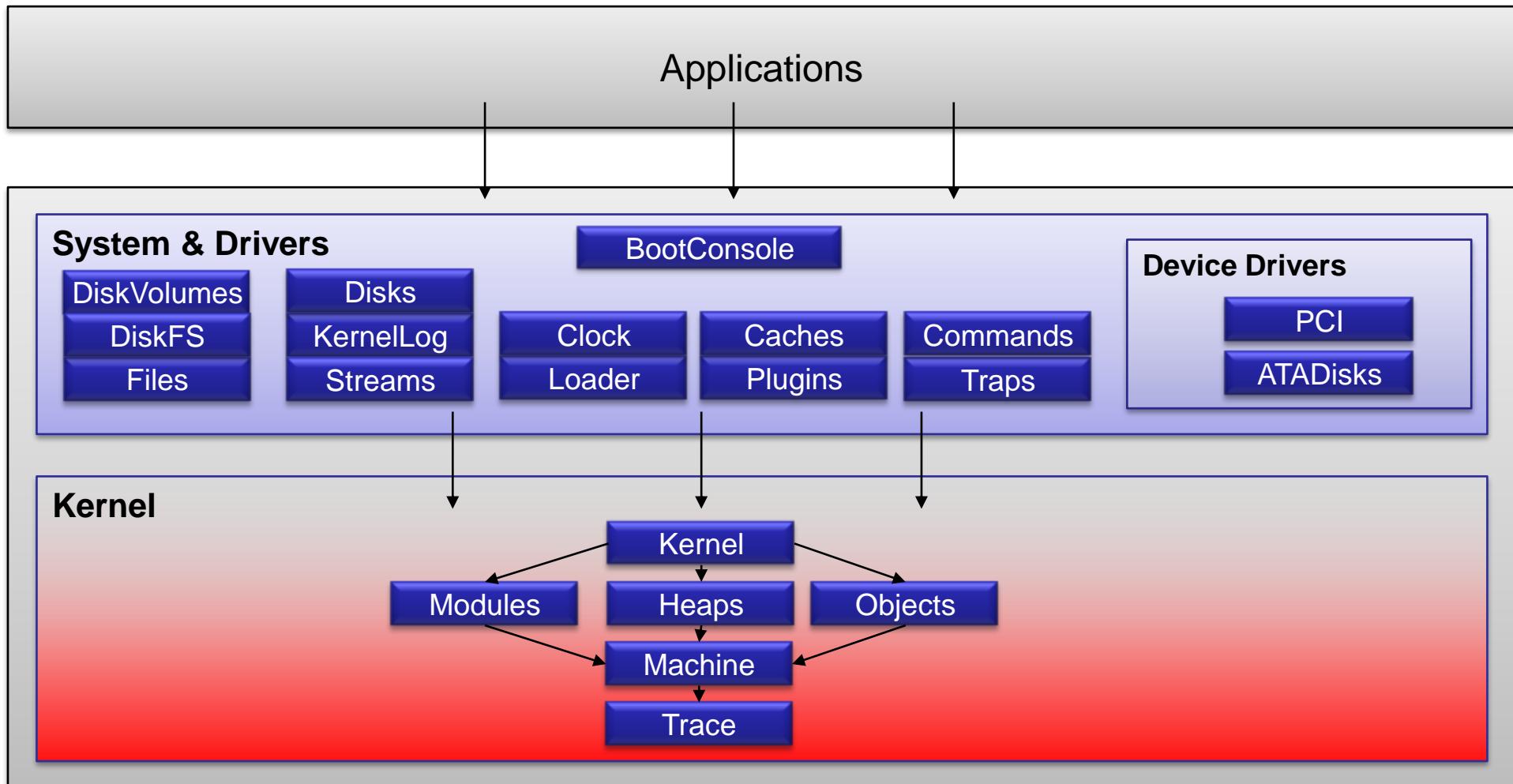
Active Oberon Code, modularer Kern

- **Machine:**
 - Abstrahiert die Hardware
 - Transfer der Konfigurationsdaten
 - Speicher- und Prozessorinitialisierung
 - Init der feingranularen Locks
 - Init des virtuellen Speichers
 - Initialisierung der Interrupts
- **Heaps:** Speichermanagement + GC
- **Modules:** Dynamischer Modullader
- **Objects:** Prozess-Management
- **Kernel:** Timer und Finalizer
- **(....)**
- **BootConsole**
 - lädt dynamisch konfigurierte Module nach

```
Idle = OBJECT
BEGIN {ACTIVE, SAFE, PRIORITY(-1)}
LOOP
  REPEAT
    Boot.SpinHint
  UNTIL maxReady > MinPriority;
  Yield
END
END Idle;
```

Beispiel eines aktiven Objektes
("Idle Thread" in Objects.Mod)

A2 Kern, Übersicht



Kapitel 2. Dateisysteme

E. Glatz, Betriebssysteme, S. 503, Tanenbaum, Betriebssysteme, Kap. 6

- 2.1. Aufgaben
 - 2.1.1. Dateien
 - 2.1.2. Verzeichnisse
- 2.2. Implementierung
 - 2.2.1. API
 - 2.2.2. Verwaltung freien Speichers
 - 2.2.3. Dateisystem-Diskstrukturen
 - 2.2.4. Verzeichnisse
 - 2.2.5. Beispiele
- 2.3 Fault Tolerance
- 2.4 Optimierungen
- 2.5 Virtuelle Filesysteme

2.1. Aufgaben eines Dateisystems

1. Speichern großer Menge an Informationen
 2. Information soll Prozess-/Applikations-terminierung überdauern
 3. Gleichzeitiger Zugriff verschiedener Prozesse / Applikationen
-
- Lösung: Persistentes Speichern der Daten in Einheiten: Dateien

2.1.1. Dateien

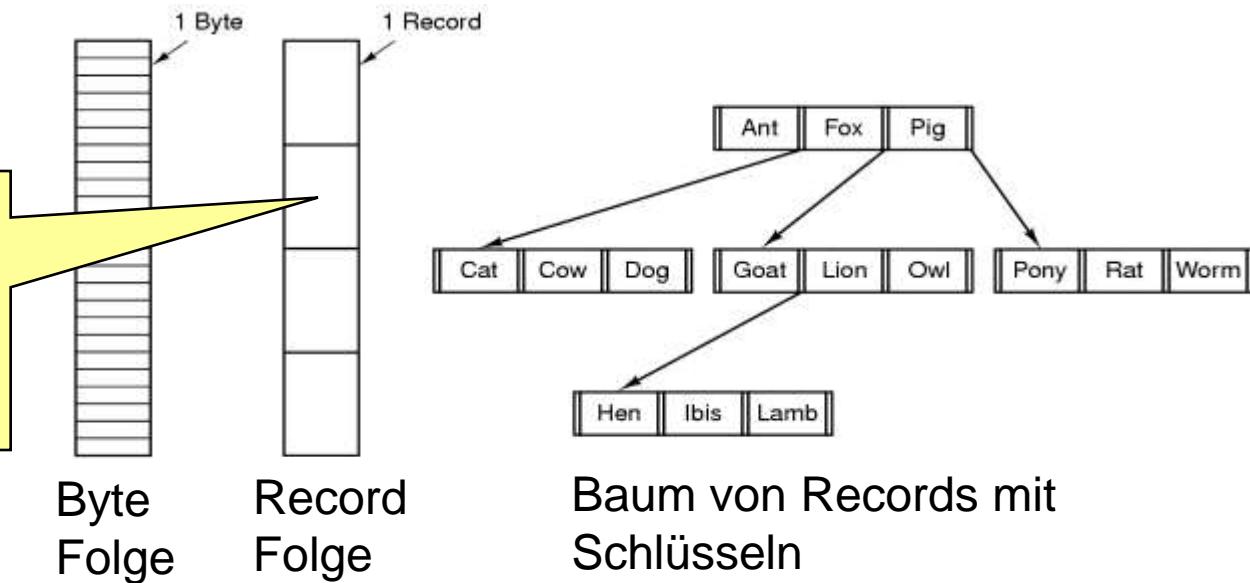
Struktur

Filetyp	Struktur	Einheit
Sequenziell	Sequenz	Byte
Index- Sequenziell	Geordnete Sequenz	Datensatz
Strukturierter Speicher	Hierarchie	Container oder Bytesequenz
...

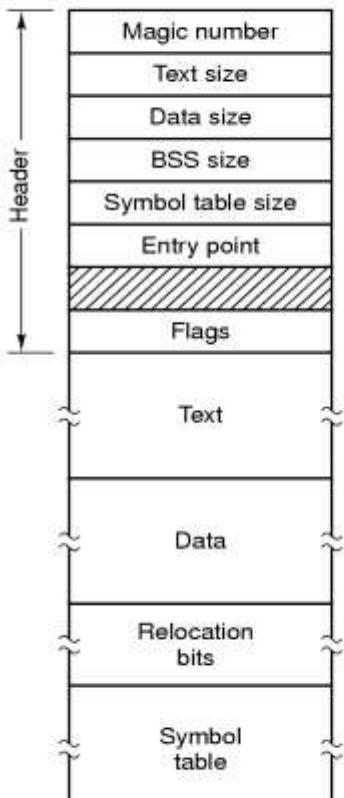
Illustration

z.B.

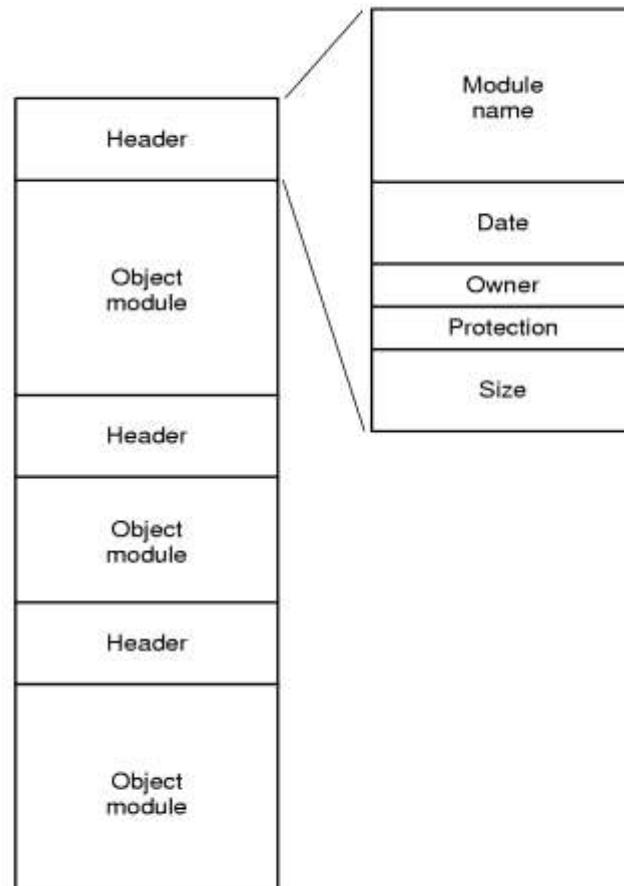
```
struct record {  
    unsigned int hasen_id  
    unsigned int anzahl_ostereier  
    unsigned int farbe  
    char liefern_an[20]  
}
```



Dateitypen



(a)



(b)

(a) Ausführbare Datei (b) Archiv

Dateinamen

- Konvention meist { char | '.' } ':' { char }



Beispiel	Bedeutung der Erweiterung
file.bak	Backup Datei
file.c	C Source Code
file.jpg	JPEG-Grafik
file.html	Hypertext Dokument
file.obj	Object File
file.mod	Oberon Module
file.pdf	Portable Document Format
file.txt	Text-Datei

Datei-Attribute

- Attribut = Eigenschaft, die mit einer Datei assoziiert ist.

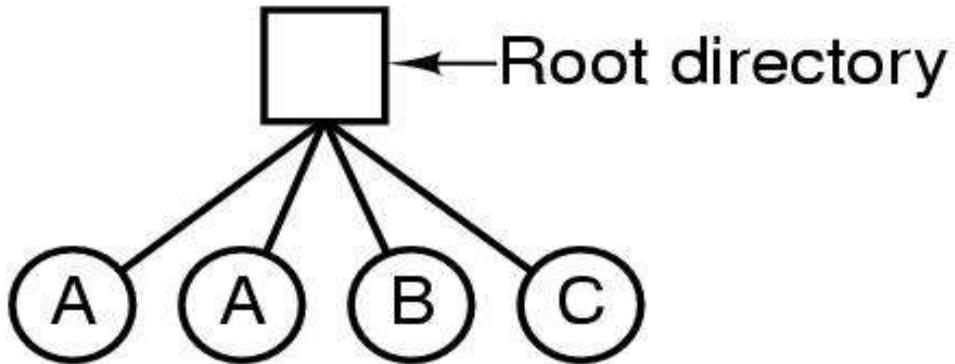
Beispiel	Bedeutung
Erstellungszeit	Datum, zu dem die Datei erstellt wurde
Erstellungsdatum	Uhrzeit, zu der die Datei erstellt wurde
Zeit letzte Änderung	Zeit des letzten Schreibzugriffs auf die Datei
Zeit letzter Zugriff	Zeit des letzten Lesezugriffs
Grösse	Dateilänge (in Bytes)
System Flag	Datei ist eine Systemdatei
Hidden Flag	Datei ist versteckt
Read-Only Flag	Datei Read-Only
Besitzer	Besitzer der Datei
Zugriffsrechte	Wer (Besitzer / Gruppe / Alle) kann auf die Datei zugreifen und wie (lesen / schreiben /ausführen)

Datei-Zugriff

- Sequentiell (historisch)
 - Lese Bytes vom Anfang zum Ende
 - Beliebiges Zurückspulen
 - Lediglich für Magnetbänder tauglich
- Random access
 - Lesen mit wahlfreiem Zugriff
 - Essentiell für Datenbanken
 - Positionierung
 - bei jeder Lese-Operation oder
 - durch Anwendung von Seek vor Read

2.1.2. Verzeichnisse

Benutzer-Dilemma (historisch)

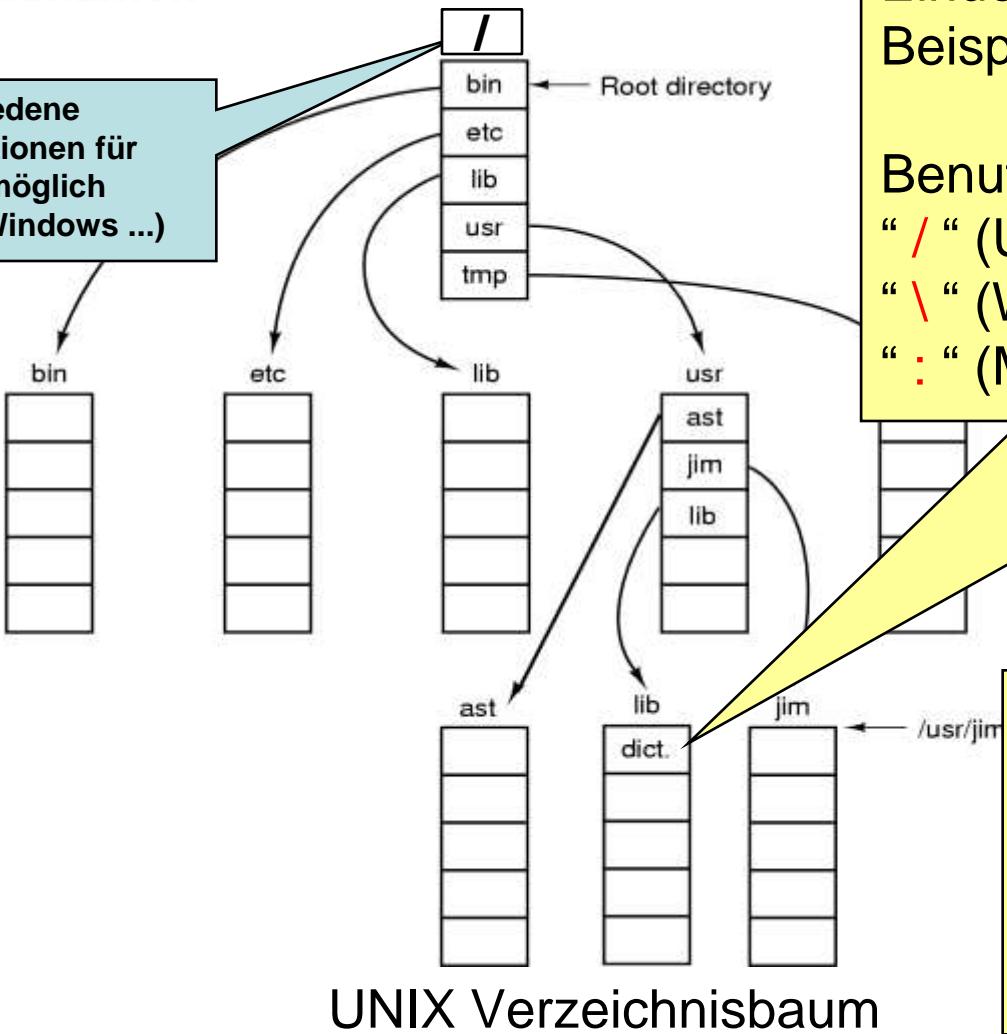


- Single Level Verzeichnis
 - 4 Dateien
 - von 3 verschiedenen Benutzern A, B, and C

Hierarchische Verzeichnisstruktur

Pfadnamen

Verschiedene Konventionen für Zugriff möglich (Linux/Windows ...)



Eindeutige Datei-Identifikation
Beispiel: bin/lib/dict

Benutzte Trennzeichen

“ / ” (Unix)

“ \ ” (Windows)

“ : ” (Macintosh, früher)

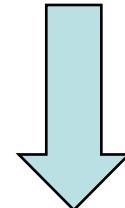
Zusätzlich Konvention:
Neben **absoluten** auch **relative**
Verzeichnispfade, dabei
„. “ : aktuelles Verzeichnis
„..“ : übergeordnetes Verzeichnis

Aufgaben des Filesystems: konzeptuell

- Repräsentation der Fileabstraktion als API
- Abbildung der Fileabstraktion auf Sekundärspeichermedien
- Verwaltung der Sekundärspeichermedien

Filemanagement

File

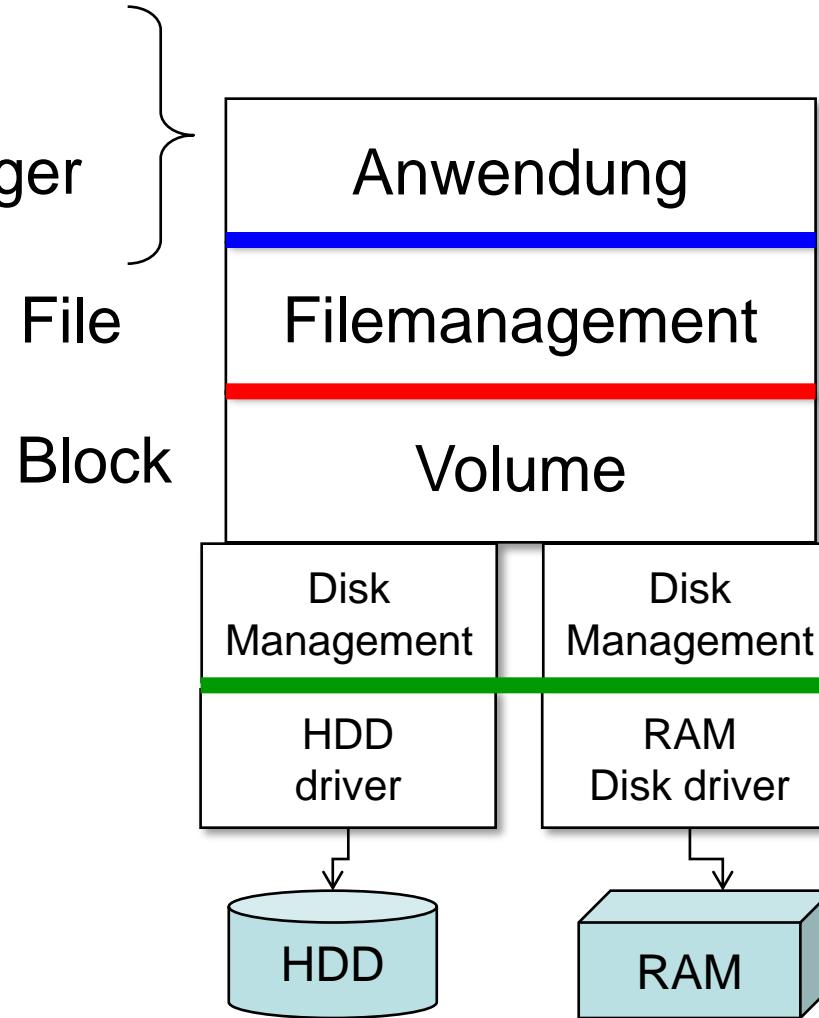


Diskmanagement

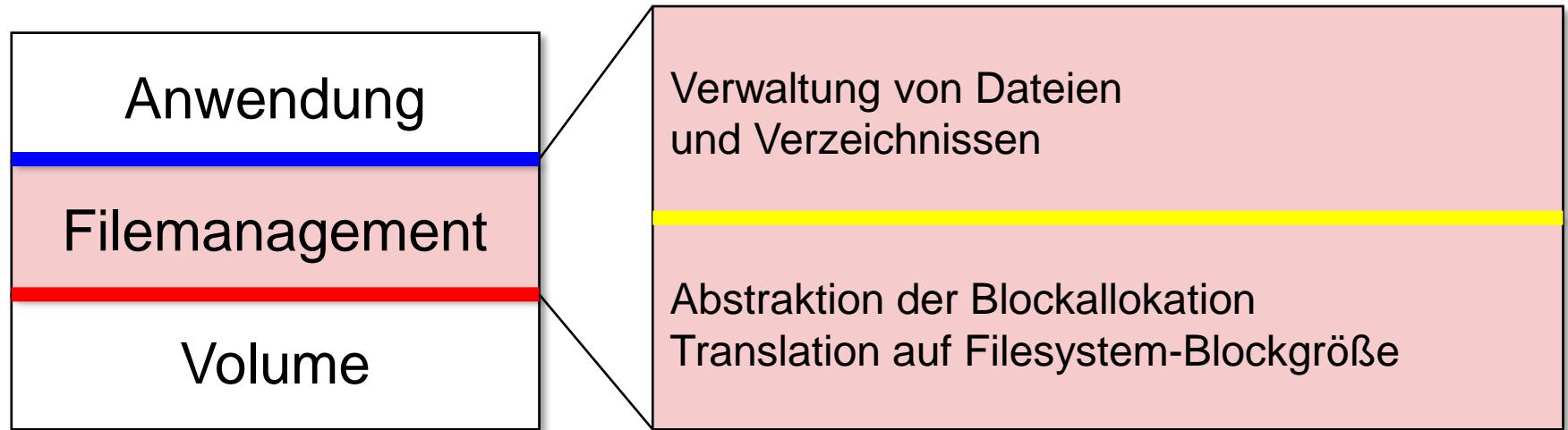
Serie von Diskblöcken

2.2. Implementation Filesystem als Subsystem

- Textsystem
- Datenbank
- Paging Manager
- ...



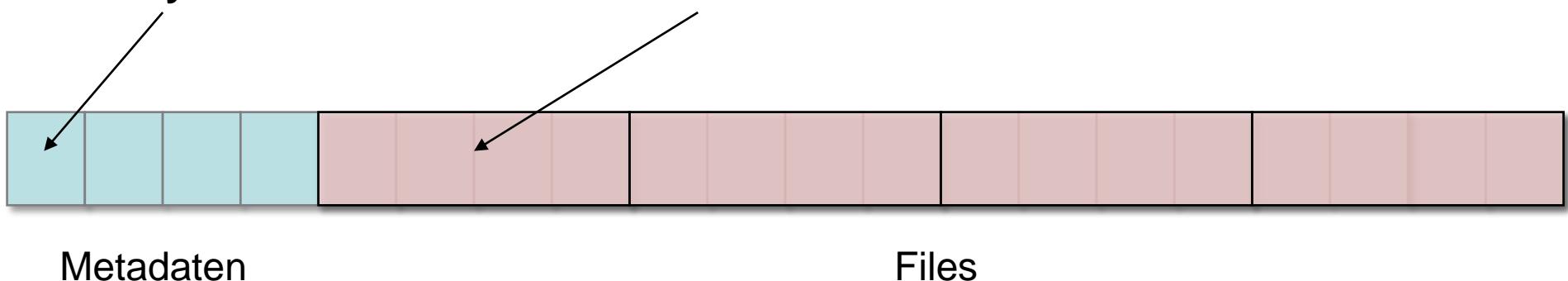
Unterteilung des File-Managements



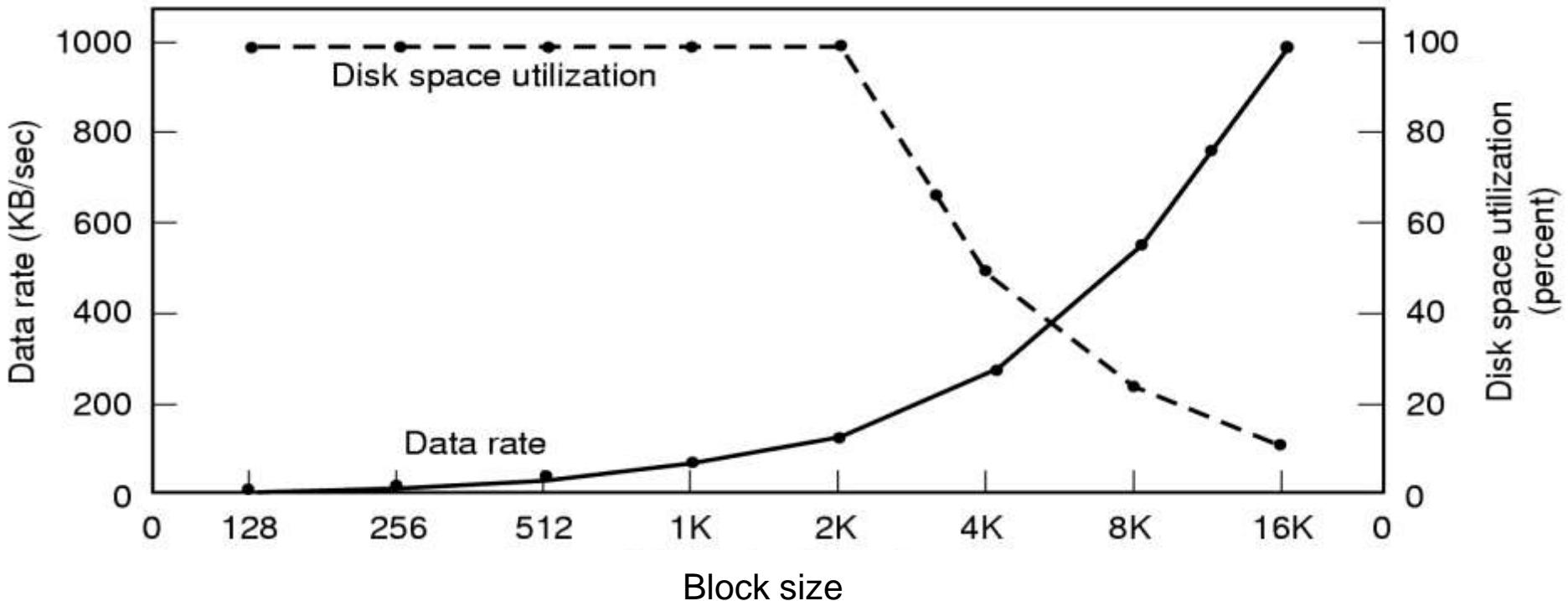
Beispiel:

Volume Block 512 Bytes

Filesystem Block (Cluster) 2 KB



Wahl der Blockgrößen



- Durchgezogene Linie: Datenrate (mittlere)
- Gepunktete Linie: Diskgrößeneffizienz (Filegröße 2KB angenommen)

2.2.1. API: Volume-Management API

- **Blockierung**
 - Einteilung in Blöcke gleicher Größe
 - Einheitliche Blockdarstellung über alle eingebundenen Partitionen
- **Operationen**
 - ReadBlock (in Hauptspeicher)
 - WriteBlock (ab Hauptspeicher)
 - GetSize

Device* = OBJECT (Plugin)

```
PROCEDURE Transfer*( op,block,num: LONGINT; VAR data: ARRAY OF CHAR;  
                      ofs: LONGINT; VAR res: LONGINT);
```

```
PROCEDURE GetSize*(VAR size, res: LONGINT);
```

...

```
END Device;
```

Beispiel: API von Disks.Device

Filesystem-Management API

- Verzeichnis (Directory)
 - Suchen
 - Eintragen
 - Umbenennen
 - Löschen
- Fileoperationen
 - Öffnen/ Internalisieren
 - Schliessen/ Externalisieren
 - Eigenschaften/ Attribute
- Zugriffsmechanismus
 - Positionieren
 - Lesen, Schreiben

Beispiel: A2 Filesystem API - FileSystem

FileSystem = OBJECT

Erzeugen

vol: Files.Volume;

PROCEDURE New0 (name: ARRAY OF CHAR): Files.File;

Suchen &
Öffnen

PROCEDURE Old0 (name: ARRAY OF CHAR): Files.File;

Löschen

PROCEDURE Delete0 (name: ARRAY OF CHAR; VAR key, res: LONGINT);

PROCEDURE Rename0 (old, new: ARRAY OF CHAR; f: Files.File; VAR res: LONGINT);

Aufzählen

PROCEDURE Enumerate0 (mask: ARRAY OF CHAR; flags: SET; enum:
Files.Enumerator);

Umbennnen

PROCEDURE FileKey (name: ARRAY OF CHAR): LONGINT;

PROCEDURE CreateDirectory0 (name: ARRAY OF CHAR; VAR res: LONGINT);

PROCEDURE RemoveDirectory0 (name: ARRAY OF CHAR; force:BOOLEAN;
VAR key, res: LONGINT);

PROCEDURE Finalize;

END FileSystem;

Beispiel: A2 Filesystem API - File

File = OBJECT

 fs: Files.FileSystem;

 PROCEDURE Set (VAR r: Files.Rider; pos: LONGINT);

Positionieren

 PROCEDURE Pos (VAR r: Files.Rider): LONGINT;

 PROCEDURE Read (VAR r: Files.Rider; VAR x: CHAR);

 PROCEDURE Write (VAR r: Files.Rider; x: CHAR);

Schreiben

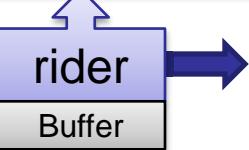
 PROCEDURE GetDate (VAR t, d: LONGINT);

 PROCEDURE GetName (VAR name: ARRAY OF CHAR);

 PROCEDURE Length (): LONGINT;

 PROCEDURE Register0 (VAR res: LONGINT);

END File;



Lesen

Externalisieren

Implementation

- Laufzeitdatenstrukturen
 - Verzeichnis (Namen & Attribute)
 - Files als Sequenzen ihres Elementtyps
 - Files als Sequenzen von Diskblocks
 - Pool freier Blöcke
- Darstellung der Laufzeitdatenstrukturen
 - Persistent auf Trägervolume
 - Internalisiert im Hauptspeicher
(konzentriert auf den aktuellen Fokus)

} Gefahr der
Inkonsistenz

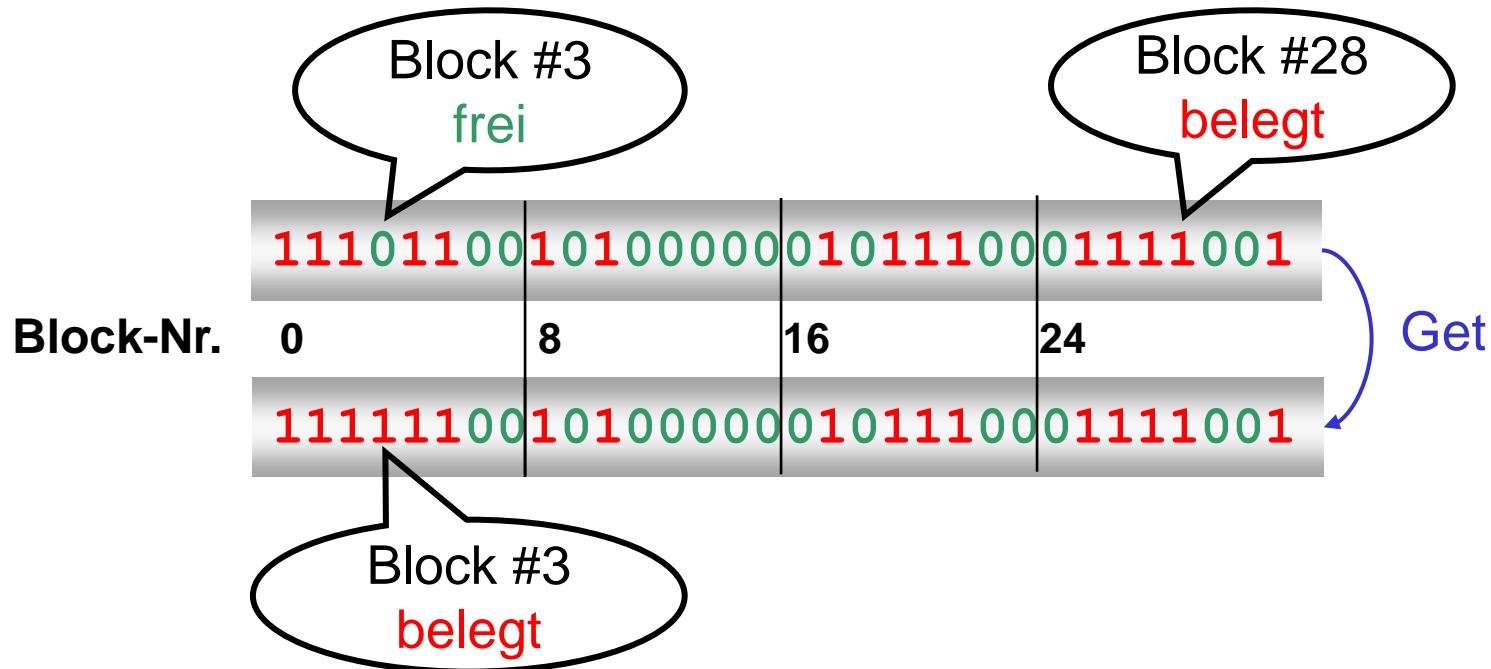
2.2.2. Verwaltung freien Speicherplatzes

Block-Pool Darstellung durch

- Bit-Tabellen
- Verkettete freie Abschnitte
- Liste freier Blöcke
- Indizierung: INodes (wie indizierte Dateien, s.u.)

Block Pool Darstellung: Bit-Tabellen

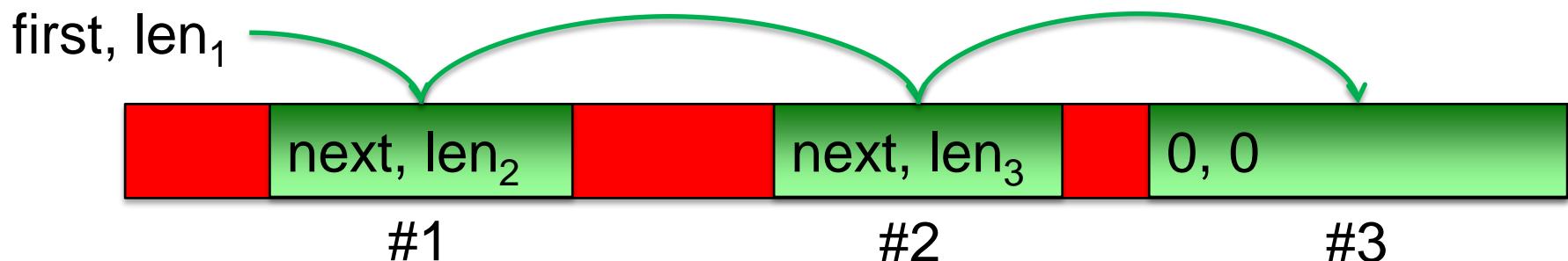
- Darstellung als „Bitmap“
 - 1 Bit pro Block off \Leftrightarrow free



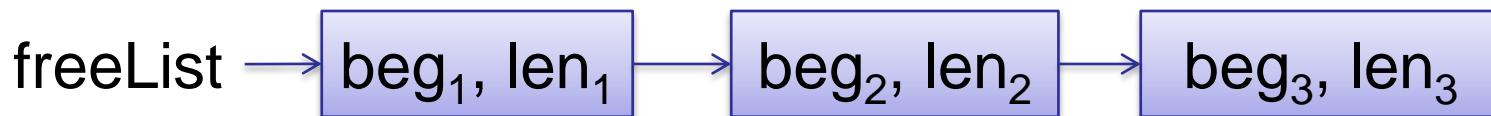
Beispiel: 16GB Platte, Blockgröße 512 Byte \rightarrow 4MB Bittabelle

Block Pool Darstellung: Verkettete freie Abschnitte

- Darstellung als Kette
 - Verkettung freier Stücke („Extents“) direkt im Volume (Verlinkung der Blöcke)



- Liste der freien Segmente
 - Im Hauptspeicher



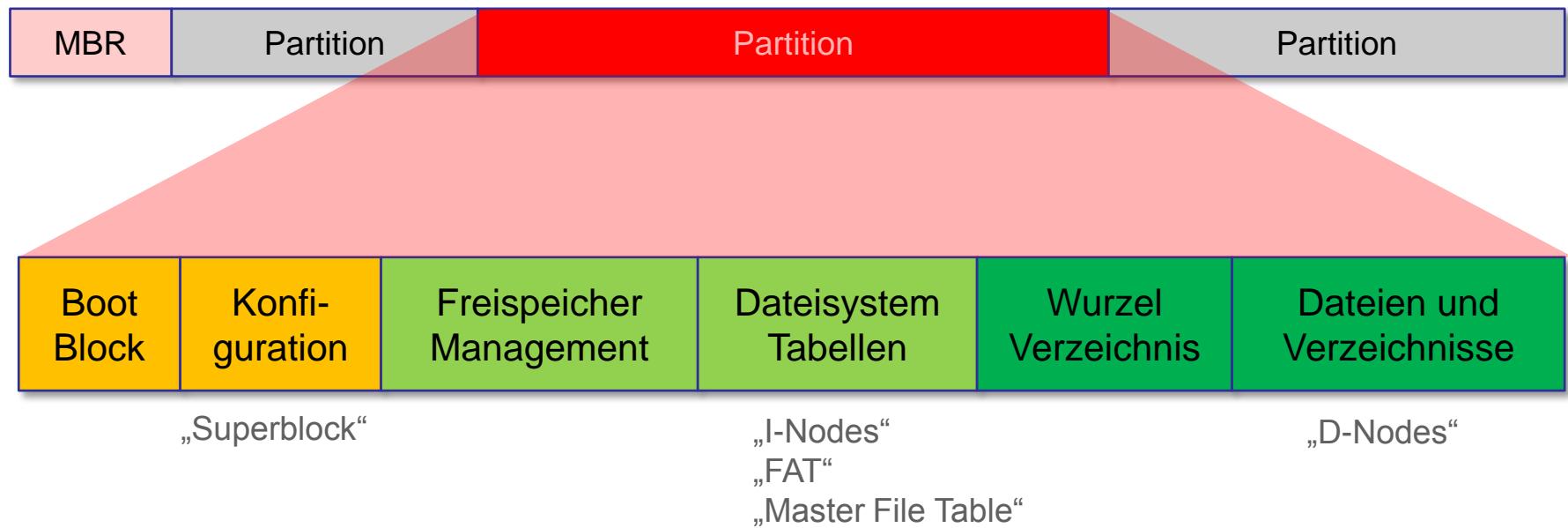
Block Pool Darstellung: Liste freier Blöcke

- Speicherung in freien Blöcken auf dem Volume

10	18	5	13	14	15	2	3	4	30
31	32	33	34	35	36	37			

- Jeder freie Block benötigt 4 Byte Speicherplatz
- (Teil der) Liste kann z.B. als LIFO-Stapel oder FIFO- Warteschlange im Hauptspeicher gehalten werden. Zur Optimierung kann im Hintergrund sortiert werden.
- Verfahren ist adaptiv, d.h. je weniger Platz, desto kürzer die Freelist.

2.2.3. File System Disk-Strukturen



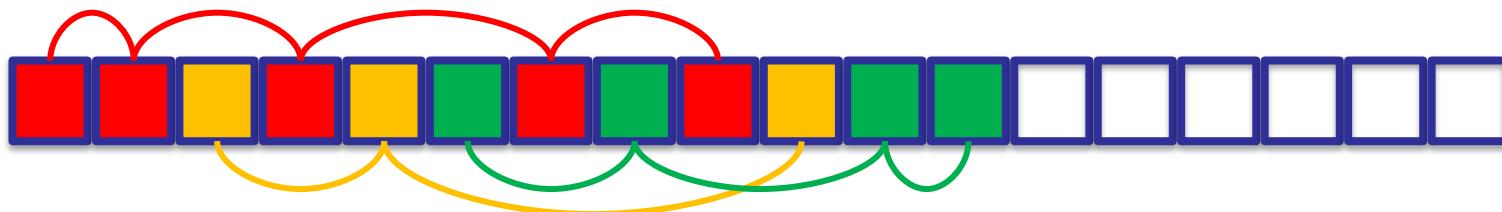
Dateisystemlayout, schematisch

Fragmentierung

- **Interne Fragmentierung** liegt vor, wenn der Speicherplatz in Blöcken nicht vollständig durch Nutzdaten belegt werden kann

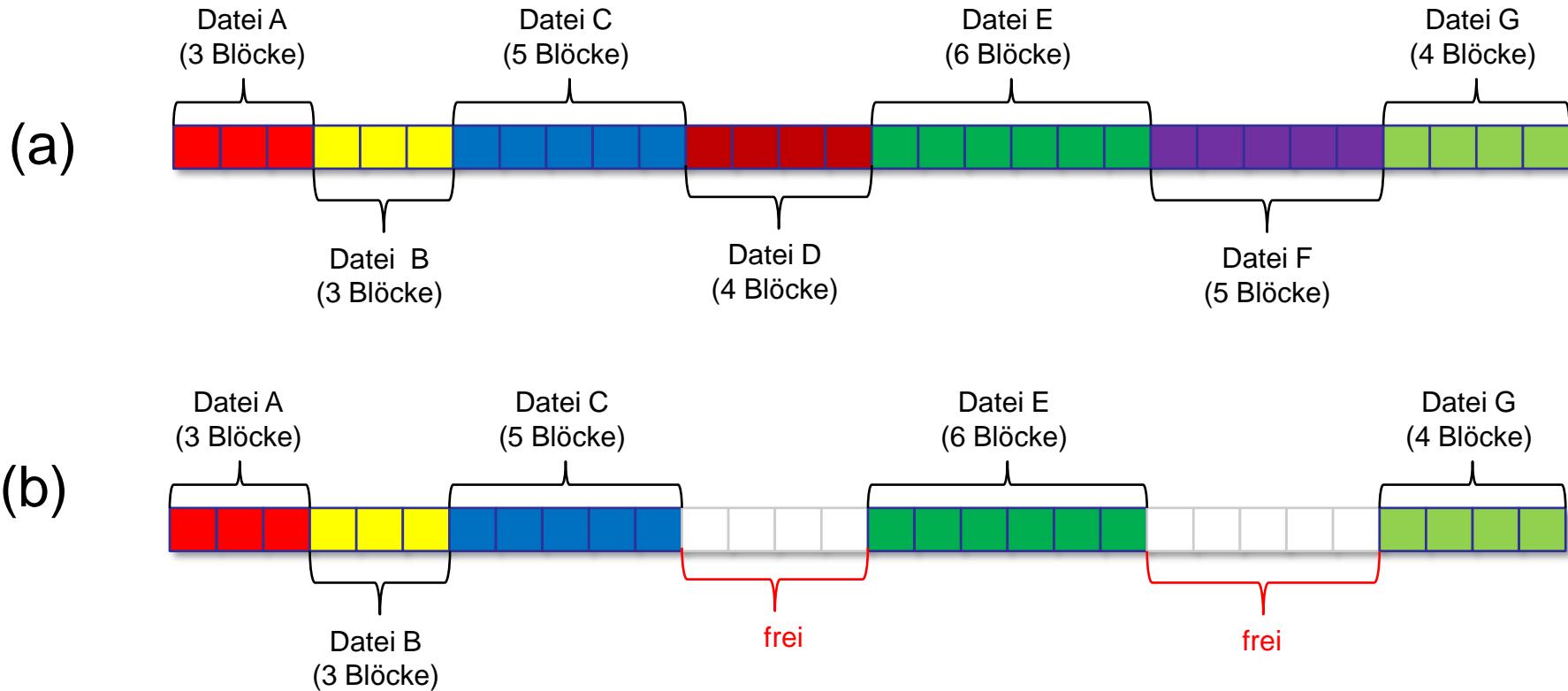


- **Externe Fragmentierung** liegt vor, wenn die Daten nicht in zusammenhängenden Blöcken alloziert werden können.



Files als Blocksequenzen (1)

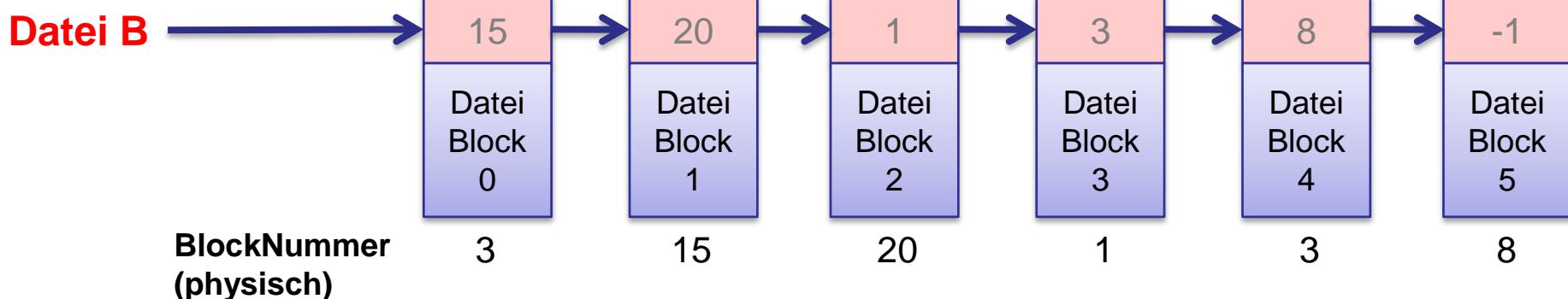
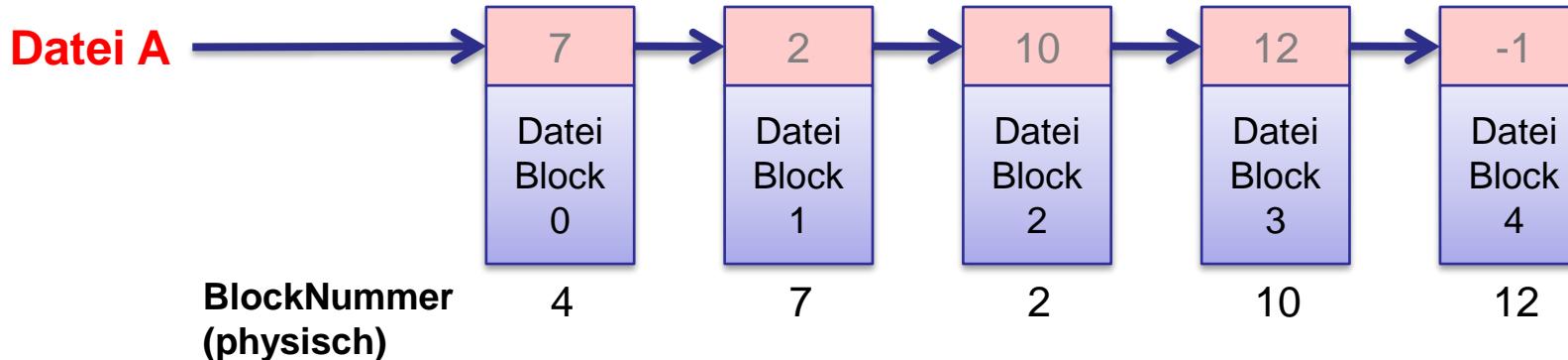
Zusammenhängende Allokation



- (a) **Zusammenhängende Allokation** von 7 Files
(b) Zustand nachdem D und F gelöscht wurden

Files als Blocksequenzen (2)

Verkettete Liste

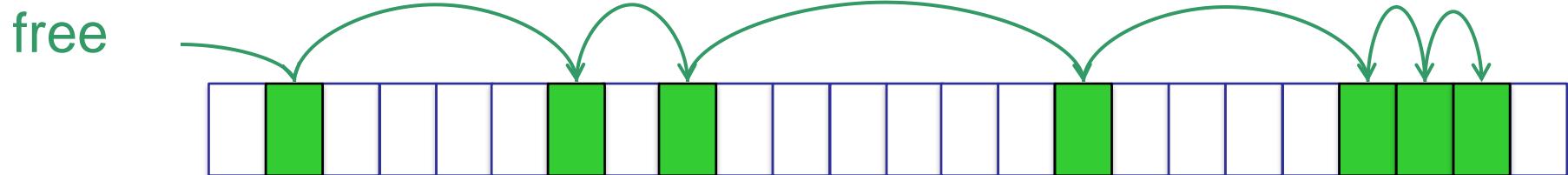


Dateien als **direkt verkettete Listen** von Blöcken im Volume

Files als Blocksequenzen (3)

Allokationstabellen (FAT)

- Pool der freien Blöcke



- Files analog

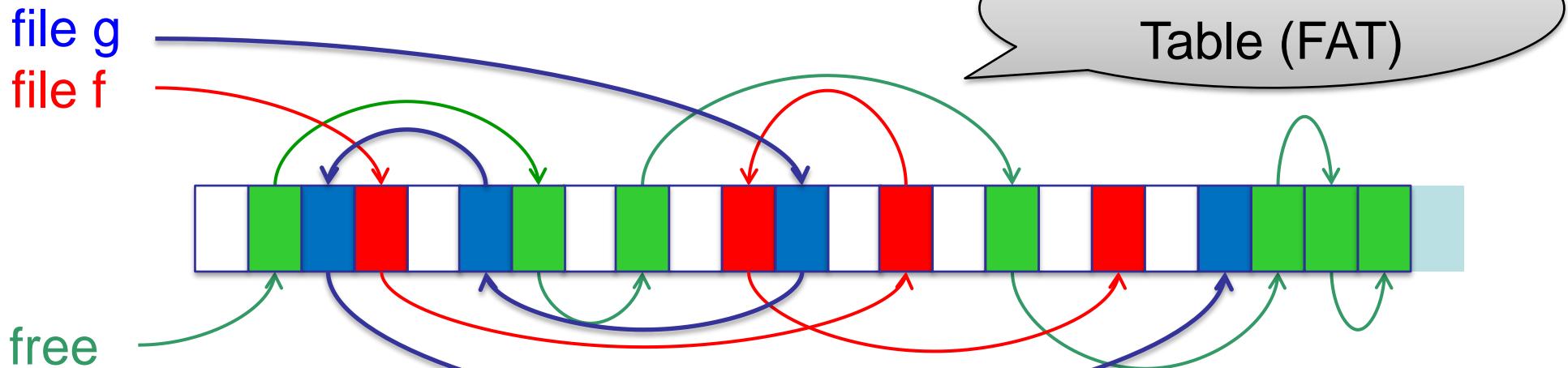
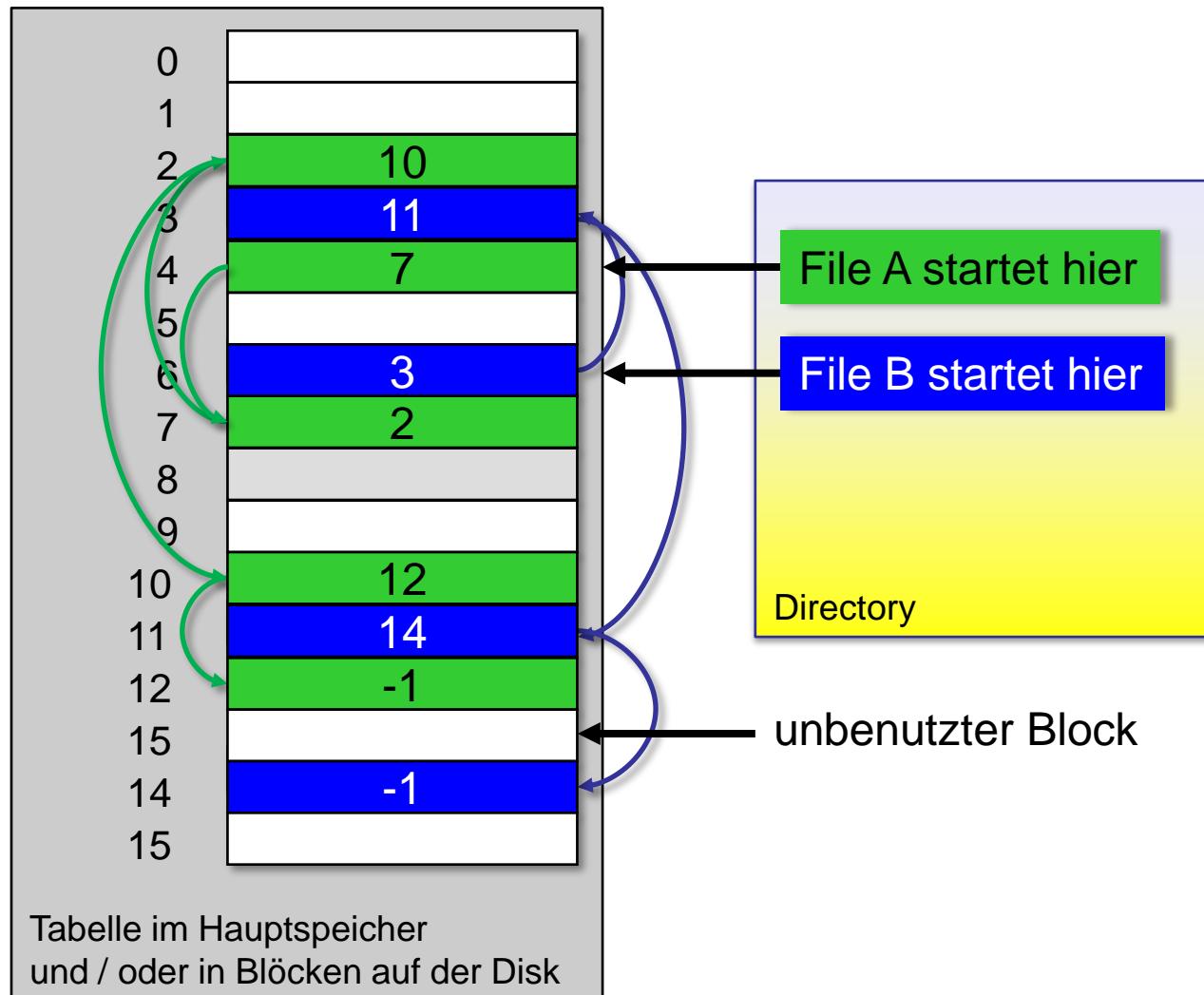


Illustration: FAT - File Allocation Table

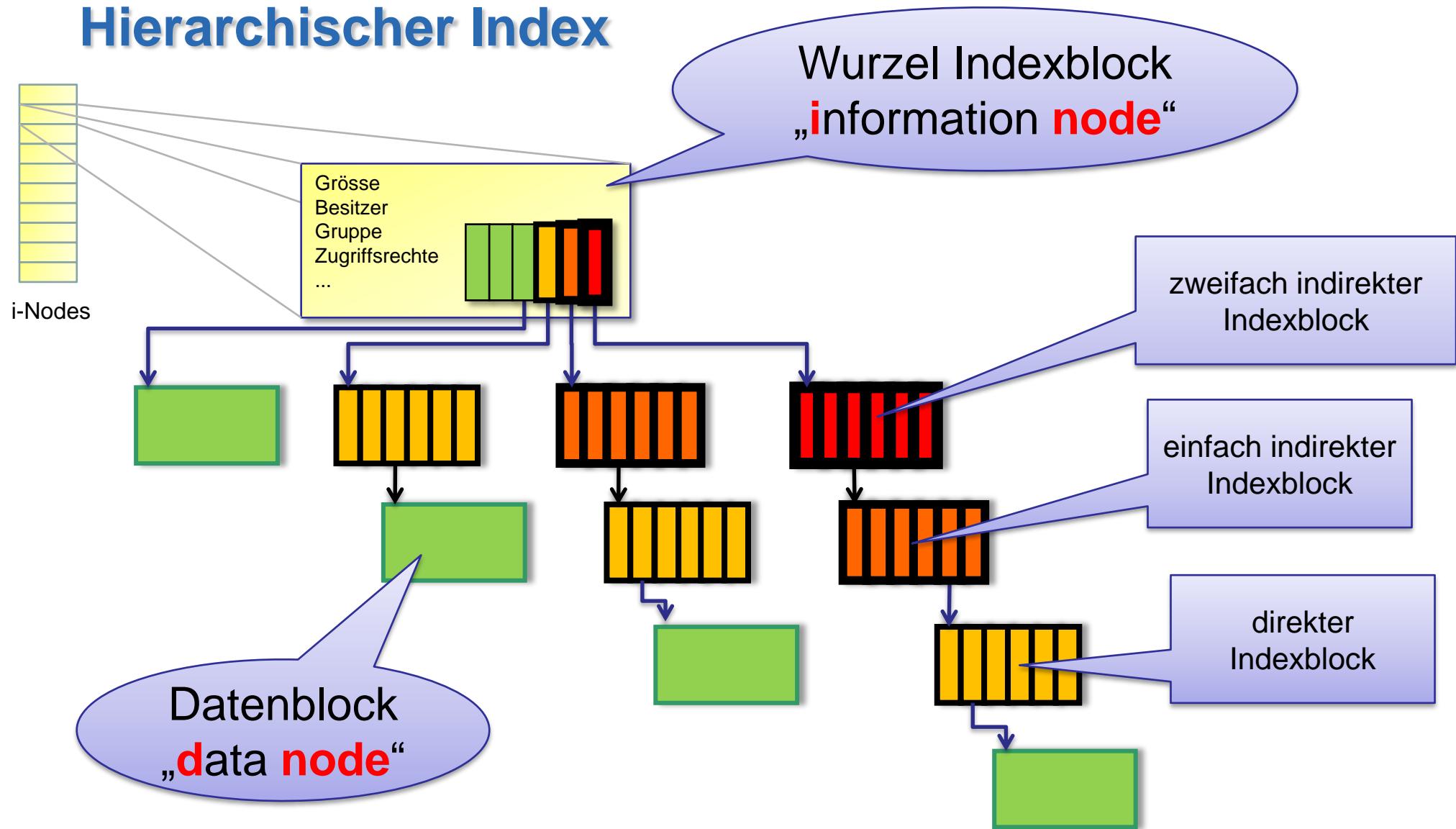


Platte 20GB
Blockgröße 1KB
→ 80 MB FAT !

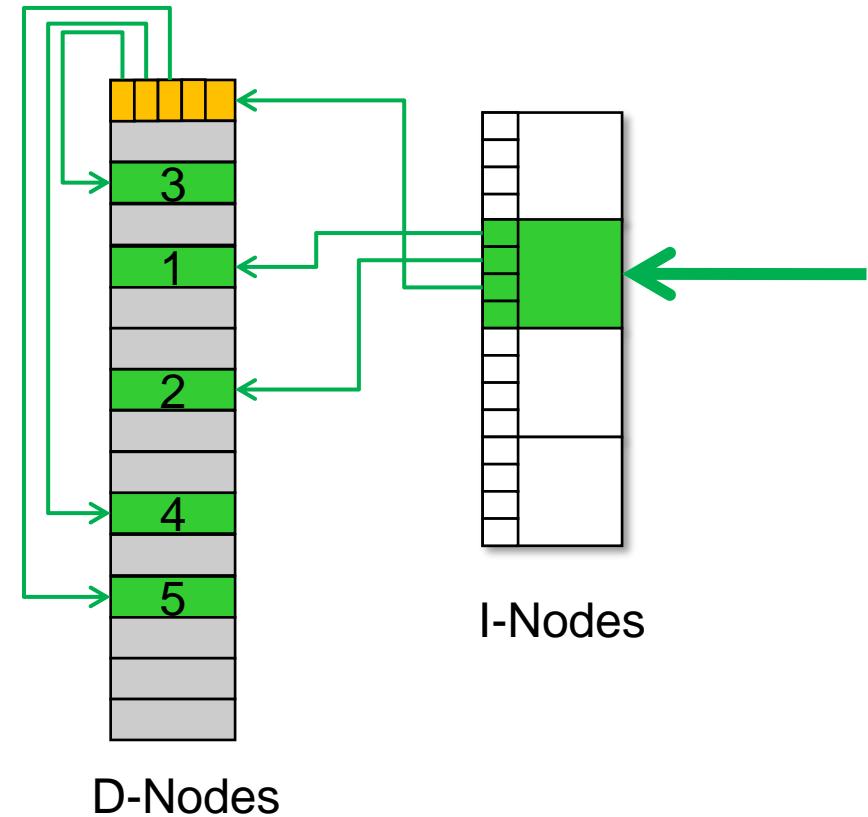
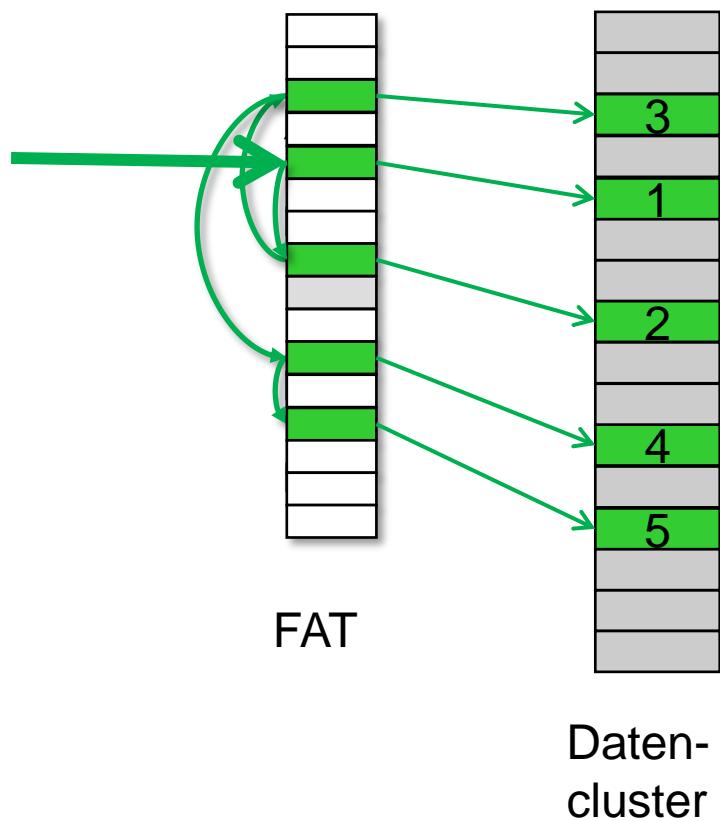
Verlinkte Liste einer Allokationstabelle

Files als Blocksequenzen (4)

Hierarchischer Index

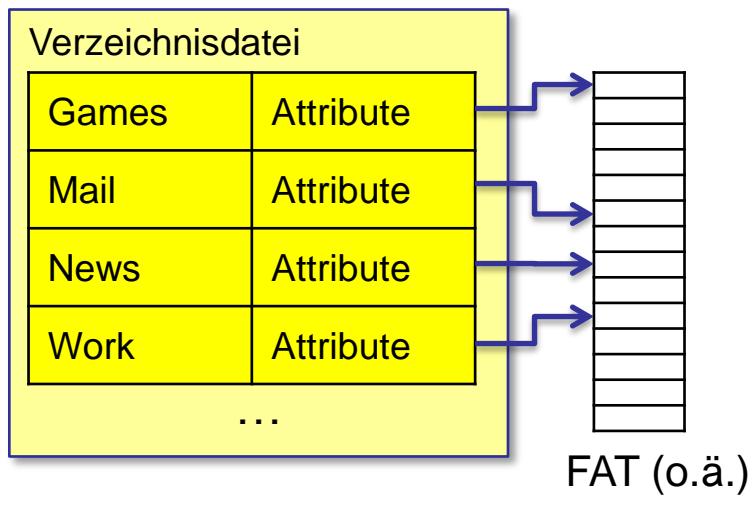


FAT vs. I-Nodes

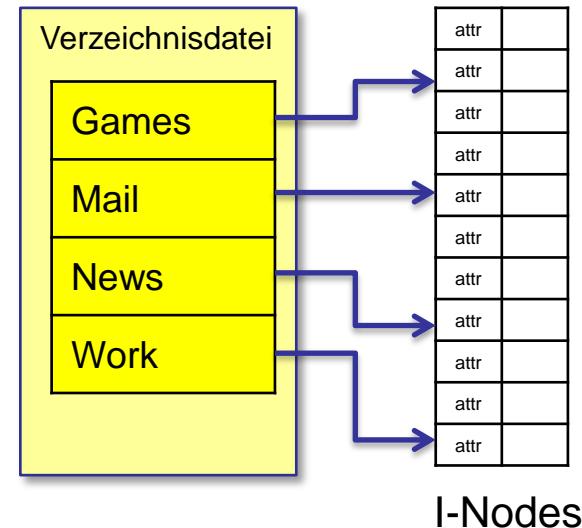


2.2.4. Implementation von Verzeichnissen

- Verzeichnis = Datei, die auf Dateien verweist*



(a)



(b)

Einfaches Verzeichnis (“directory”) mit Namenseinträgen fixer Länge

- (a) Adressen und Attribute im Verzeichniseintrag
- (b) Verzeichnis mit Verweisen auf I-Nodes

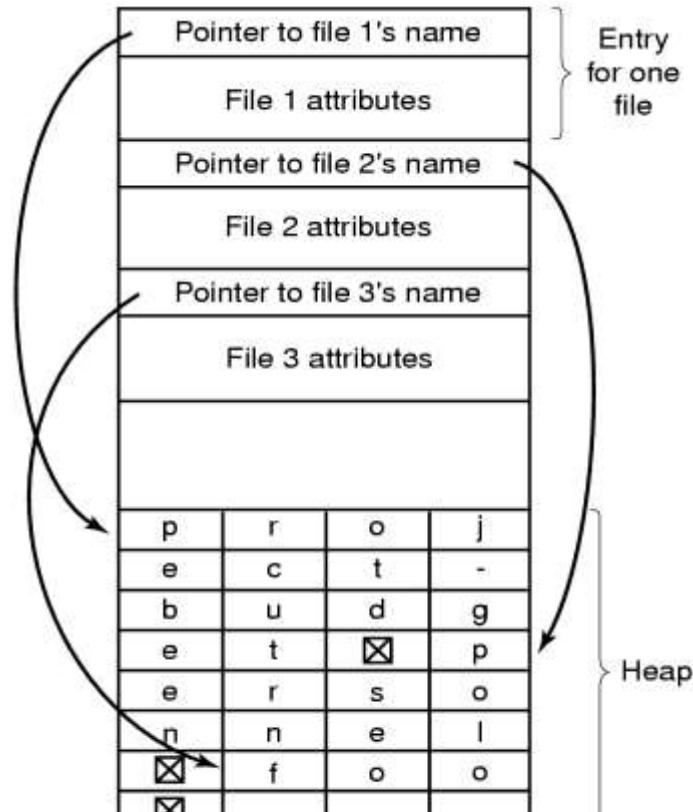
*Ausnahme: Root-Verzeichnis bei alten Systemen

Implementation von Verzeichnissen

Dateinamen mit verschiedener Länge

File 1 entry length			
File 1 attributes			
p	r	o	j
e	c	t	-
b	u	d	g
e	t	☒	
File 2 entry length			
File 2 attributes			
p	e	r	s
o	n	n	e
l	☒		
File 3 entry length			
File 3 attributes			
f	o	o	☒
⋮			

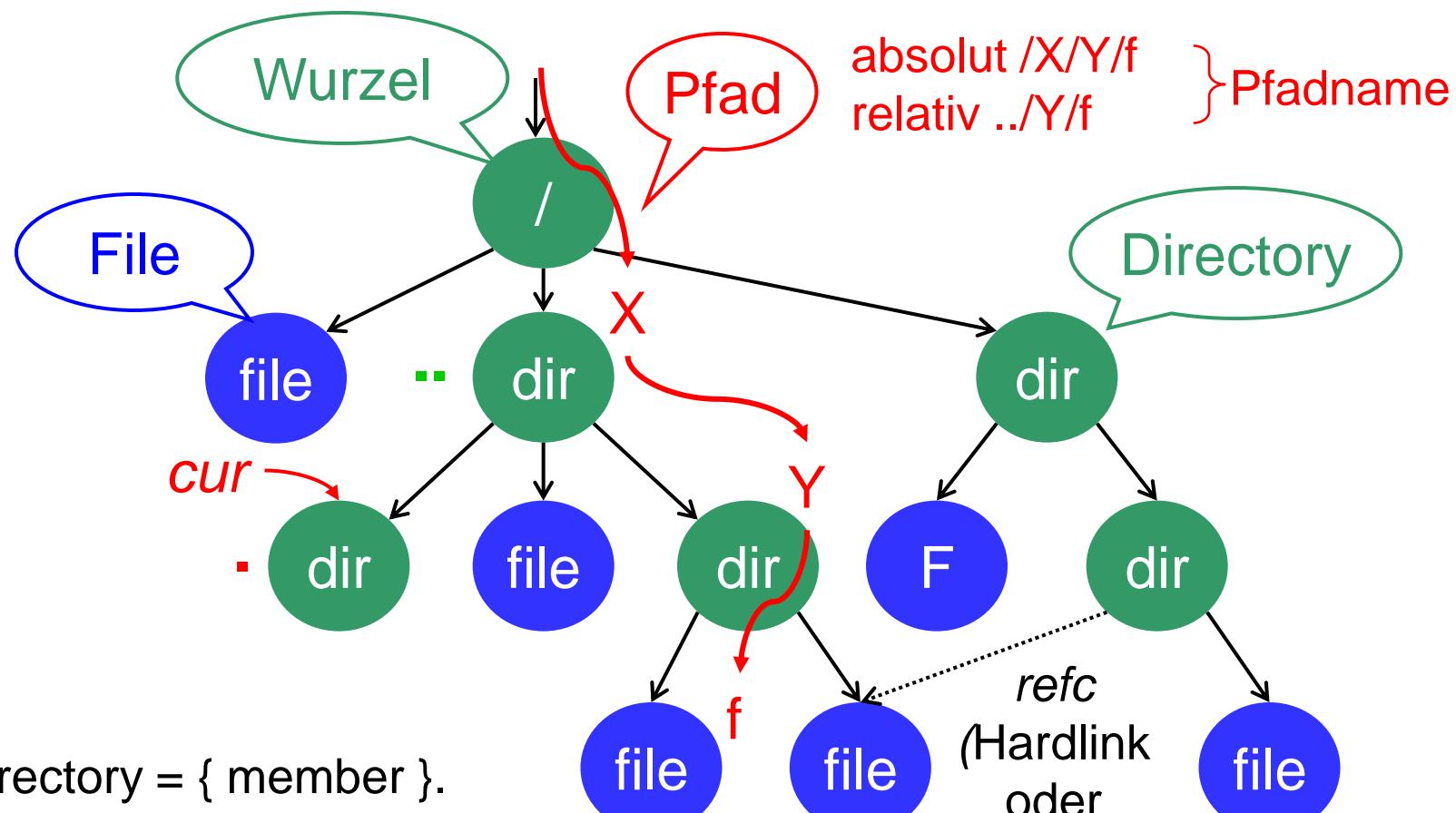
(a)



(b)

- Ansätze für lange Dateinamen
 - (a) In-line
 - (b) Auf einem Heap

Links*



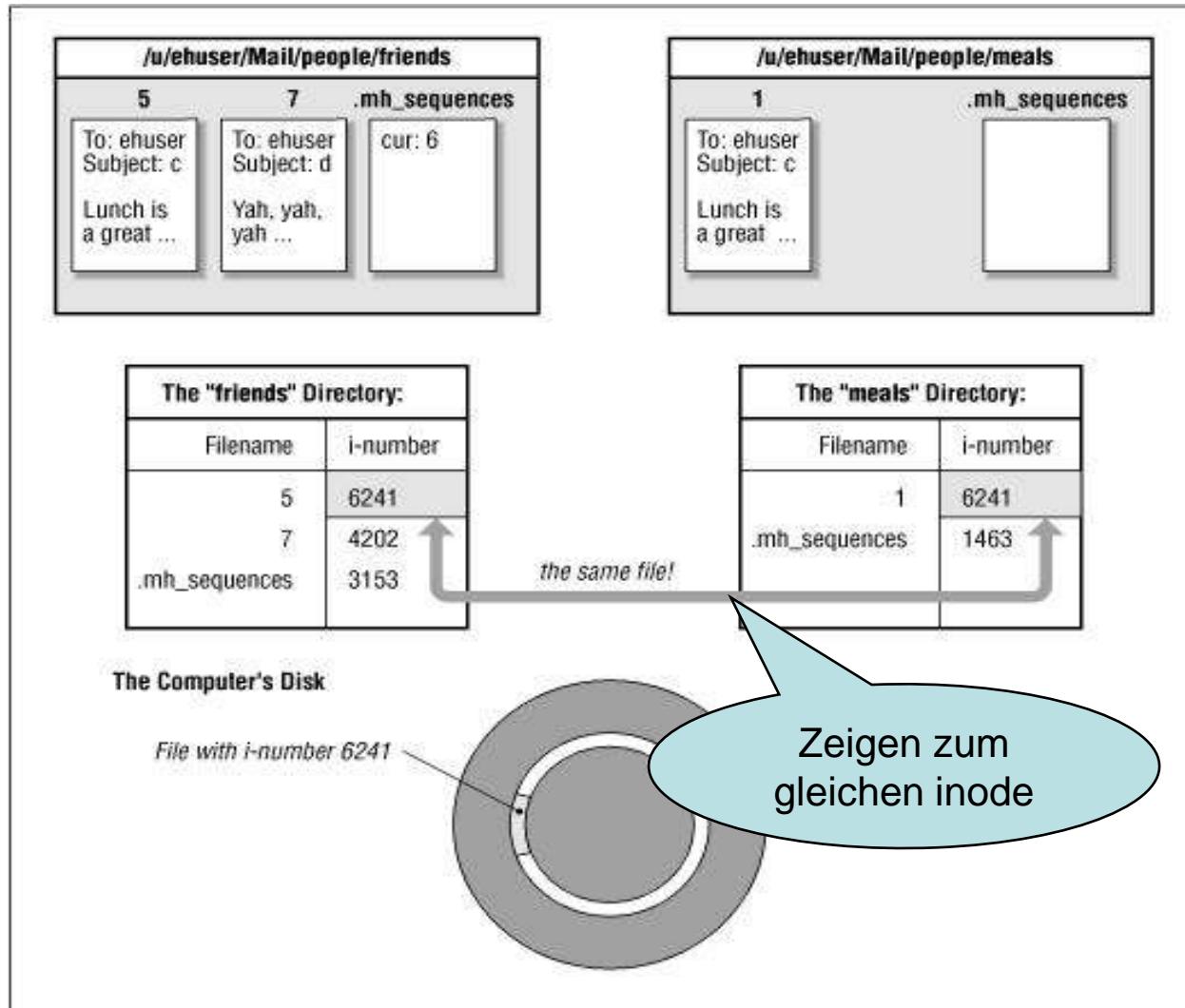
directory = { member }.

member = directory | file.

Wurzelverzeichnis = directory

*Links = Verknüpfungen

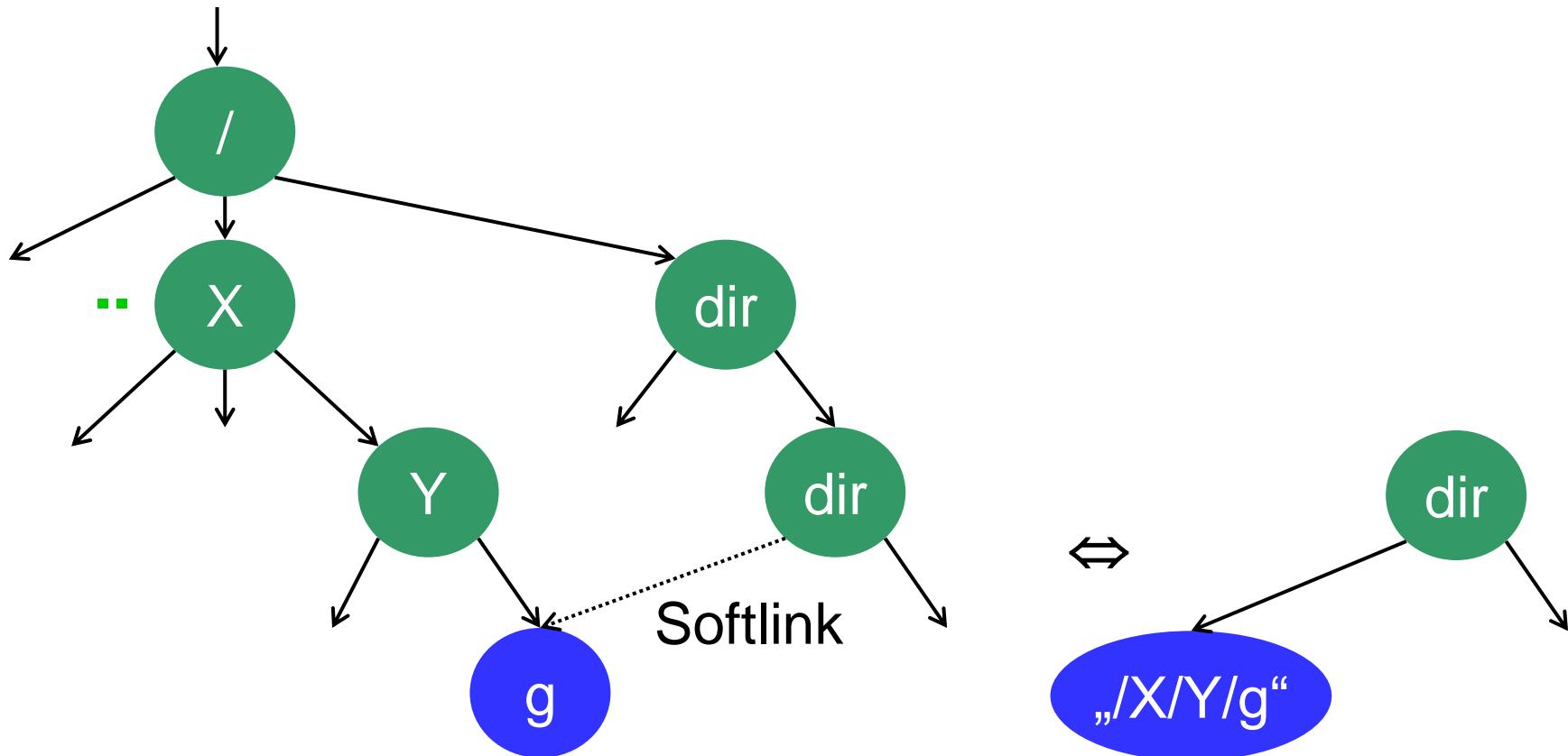
Hardlinks



- Zusätzliche Referenz auf dieselbe Datei
- Lösung für das Konsistenzproblem: Link count
- Hardlinks nur auf demselben Volume

Softlinks*

- Dargestellt durch File, welches (nur) den betreffenden Pfadnamen enthält



*auch „symbolic links“

2.2.5. Beispiele von Dateisystemen

- FAT Filesysteme
- NTFS Filesystem
- UNIX Filesysteme
- ZFS
- HFS+
- CDROM Filesystem
- Oberon Filesystem

FAT Filesysteme

- Konzepte
 - Verwendung einer Allokationstabelle
 - Files (verlinkt)
 - Freie Cluster (nicht verlinkt)
 - Defekte Cluster
 - Volumestruktur
 - FAT1 und FAT2 (Duplikat als Redundanz)
 - Bereich für das Wurzelverzeichnis
 - Datenbereich (Clusternummerierung*)
- Versionen mit verschiedenen Kapazitäten
 - FAT12
 - FAT16
 - FAT32
 - exFAT (64 bit)

*Cluster: Zusammenhängende Sektoren / logische Disk-Blöcke

FAT Filesysteme Volumestruktur



Bootsektor

FAT

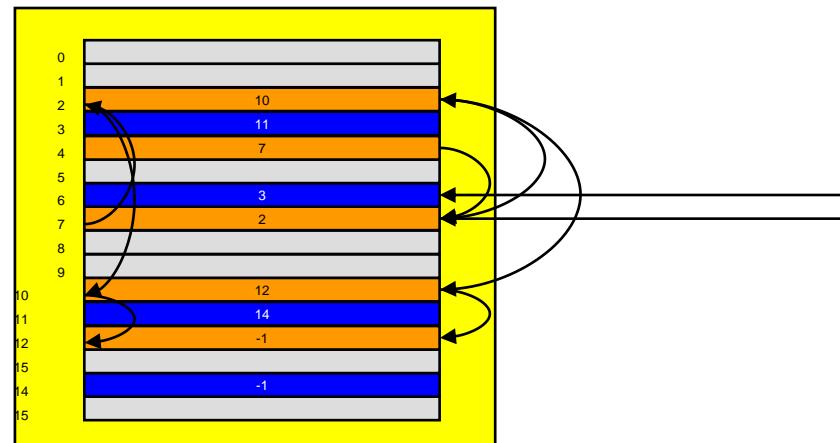
FAT Duplikat

Wurzelverzeichnis
(nicht bei FAT32)

Daten- und
Verzeichnisblöcke
(in diesem Beispiel: 4k Blöcke)

Bootcode und Grunddaten,
z.B.

- Bytes Pro Sektor
- Sektoren Pro Cluster/Block
- Anzahl FAT Kopien
- FAT Variante
- Flags etc.



Name	Größe	1. Block
...		
file.txt	20480	50
directory	16384	51
...		

FAT Filesysteme

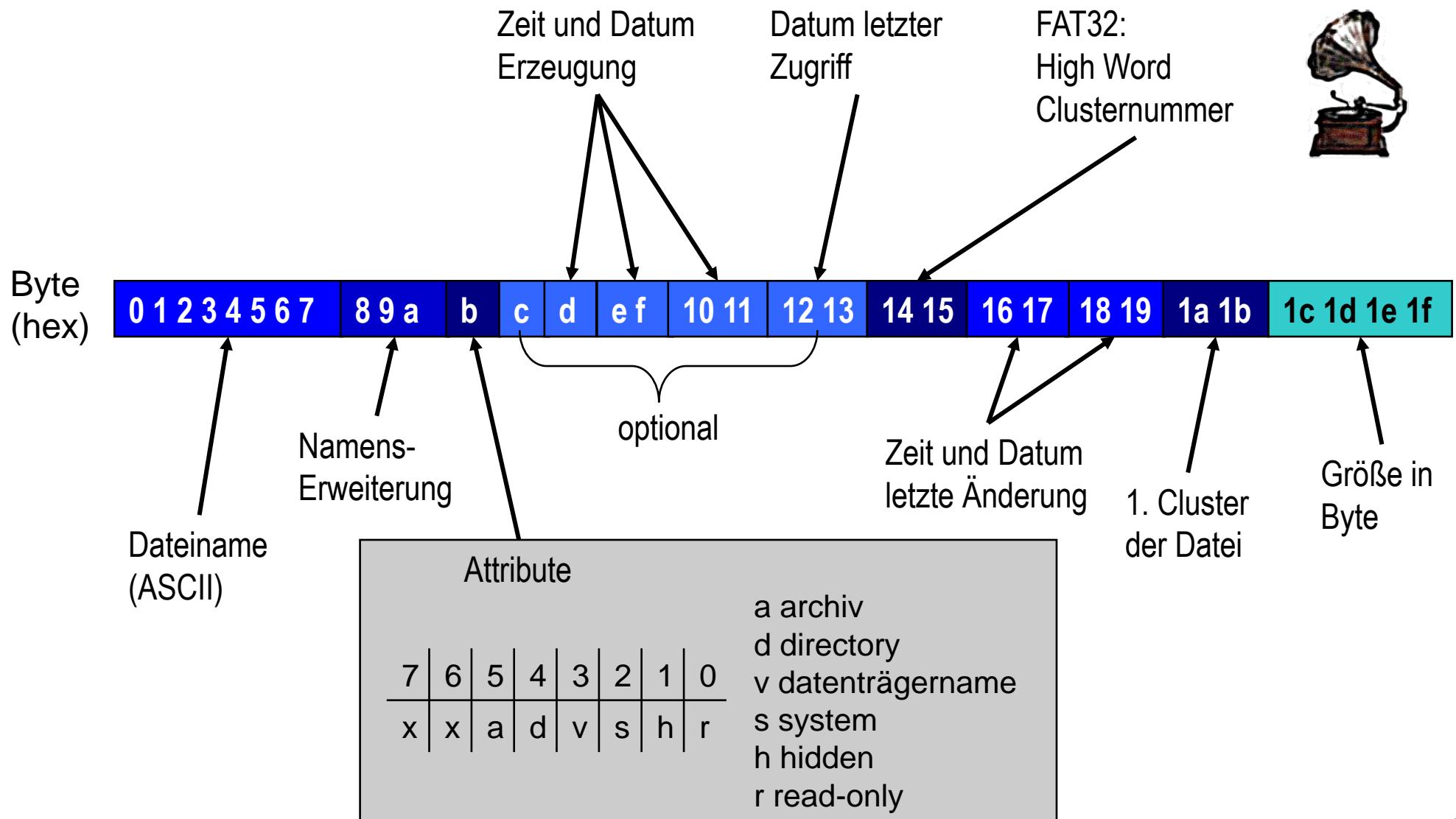
Kapazitäten

Cluster Grösse	FAT-12 <2080 Cluster 1980	FAT-16 <2^16 Cluster 1983	FAT32 <2^28 Cluster 1997	ExFAT 64 bit 2006
0.5 KB	2MB			
1KB	4MB			
2KB	8MB	128MB		
4KB	16MB	256MB	1TB (4GB)	(2^64 B)
8KB		512MB	2 TB (4GB)	(2^64 B)
16KB		1 GB	4TB (4GB)	(2^64 B)
32KB		2GB	8TB (4GB)	(2^64 B)

- Maximale Partitionsgrößen (Dateigrößen) zu Clustergrößen
- Leere Box: verbotene Kombination

* „Block“ und „Cluster“ oft synonym verwendet

FAT Eintrag im Verzeichnis



FAT: Spezielle Dateinamen

0x00	Kein Eintrag und keine nachfolgenden Einträge im Verzeichnis
0x05	Steht für Zeichencode 0xe5 („^“)
0x2e 0x20	Eigenes Verzeichnis „..“
0x2e 0x2e	Übergeordnetes Verzeichnis „..“
0xe5	Gelöschte Datei

VFAT*: Lange Dateinamen

Aufsteigende Nummer
für Zusatzeinträge;
letzter Zusatzeintrag, falls Bit 6
gesetzt

Checksumme über
Dateinamen

Checksumme über
Dateinamen

Null

Byte
(hex)



Zeichen 1-5 des
erweiterten Dateinamens
in Unicode-Kodierung

„Unmögliche“
Attributkombination
 $r=1, h=1, s=1, v=1$

Zeichen 6-11 des
erweiterten Dateinamens

Zeichen 12-13 des
erweiterten Dateinamens

*Virtual FAT

VFAT: Verkettung der Dateinamen

68	d o g	A O	C K											O		
3	o v e	A O	C K	t h e l a										O	z y	
2	w n f o	A O	C K	x j u m p										O	s	
1	T h e q	A O	C K	u i c k b										O	r o	
Bytes	T H E Q U I ~ 1	A N T S		Creation time	Last acc	Upp		Last write		Low	Size					

Beispiel: lange Dateinamen bei VFAT werden durch mehrere VFAT Einträge am Stück repräsentiert.

NTFS

Russinovich, Solomon: Microsoft Windows Internals, 4th edition, S. 717 ff.

■ Konzepte

- Wiederherstellbarkeit
 - Transaktionsprotokoll
„Atomic transactions“ mit Rollbacks
 - Redundanz der wichtigsten Metadaten
- Sicherheit
 - Zugriffsrechteverwaltung für Benutzer(gruppen)
- Redundanz und Fehlertoleranz
 - RAID Unterstützung
- Kompression und Verschlüsselung von Dateien und Verzeichnissen
- Mehrere Datenströme pro Datei: Attribute
- Sehr große Dateien möglich
(64 Bit basiert: bis zu 16 Exabyte)

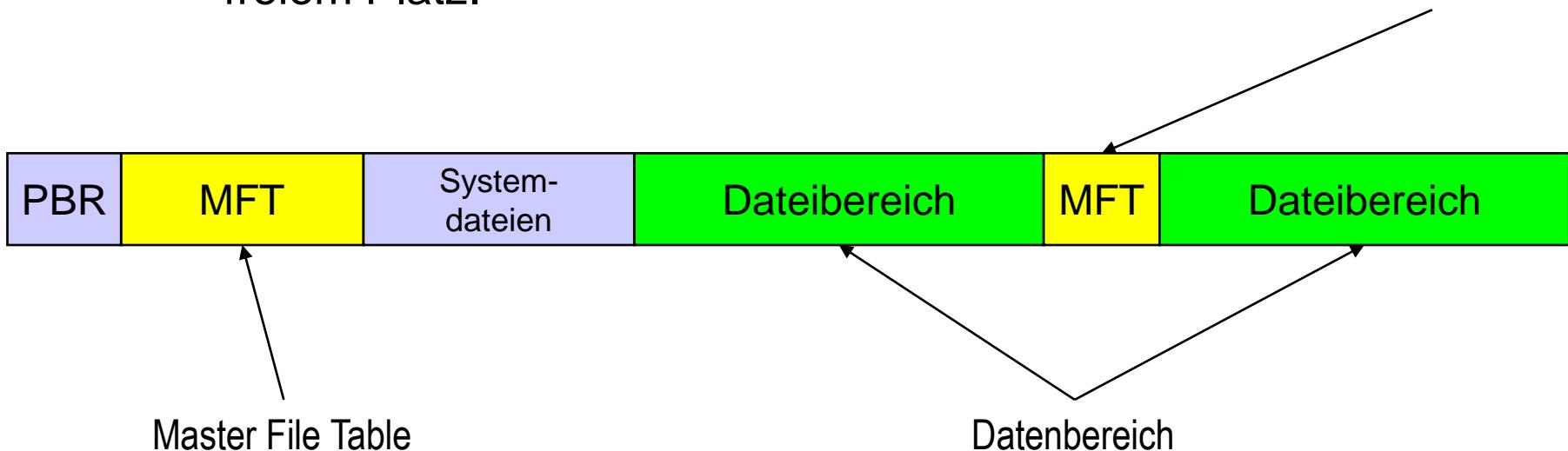
NTFS

Volume Struktur

Datenträger: logische Partition einer Festplatte mit maximal 2^{64} Byte. Besteht aus

- Master File Table,
- andere Dateisysteminformationen,
- Dateien und
- freiem Platz.

Kopie der
wichtigsten Einträge der
Master File Table



NTFS

Master File Table (MFT)

Datei	Eintrag	Bedeutung
0	\$Mft	MFT
1	\$MftMirr	MFT gespiegelt
2	\$LogFile	Log Datei (->Journaling FS)
3	\$Volume	Volume Datei
4	\$AttrDef	Attribut-Definitionen
5	\	Root Directory
6	\$Bitmap	Allokationstabelle
7	\$Boot	Boot Sektor
8	\$BadClus	Bad Cluster Datei
9	\$Secure	Sicherheitseinstellungen
10	\$UpCase	Uppercase Character Mapping
11	\$Extend	Extended Metadata Directory
12 ... 15		Ungenutzt
16		Benutzerdaten und Verzeichnisse
...		

Clustergrösse von NTFS kann variieren (abhängig von Partitionsgröße)
Jedoch: Grösse eines MFT-Eintrages fix bei 1KB

Verzeichnis = Datei, die die Positionen der Dateien in der MFT auflistet.

„everything on the disk is a file“

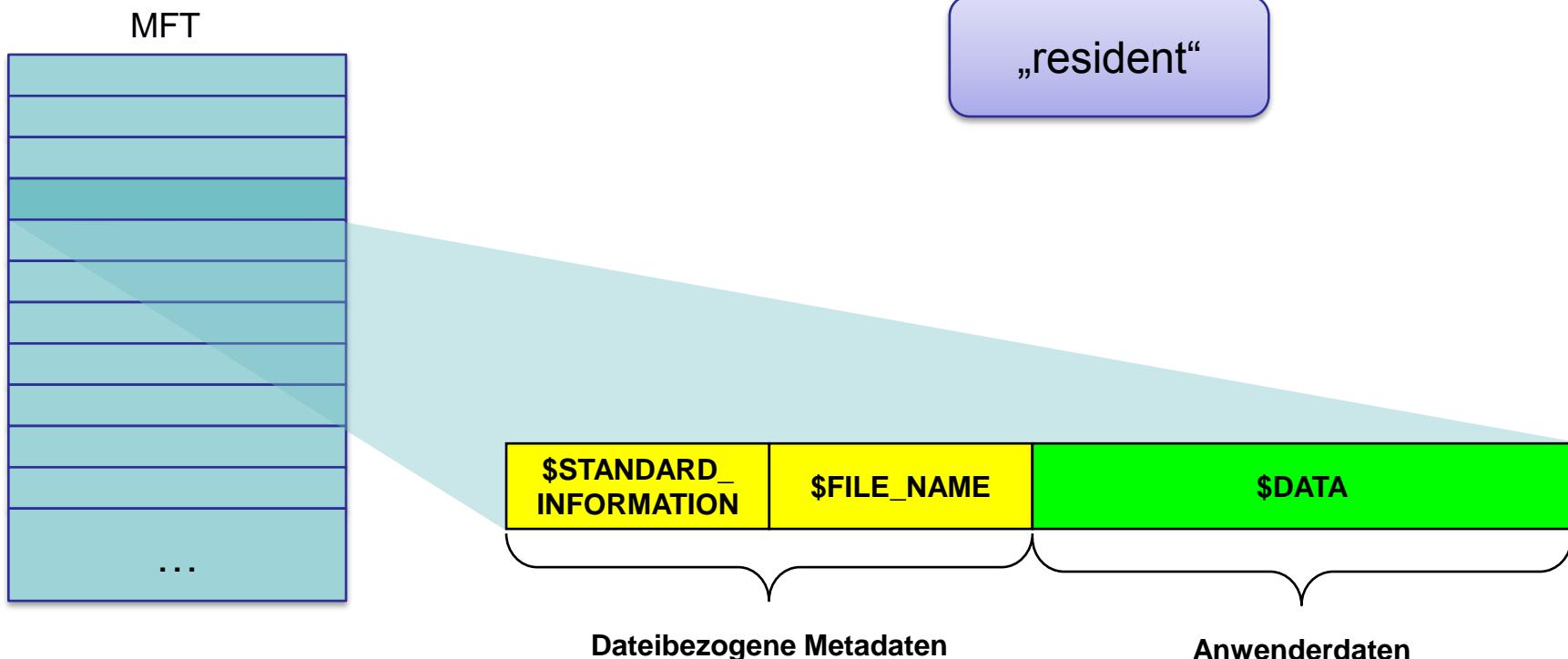
NTFS

Attribute

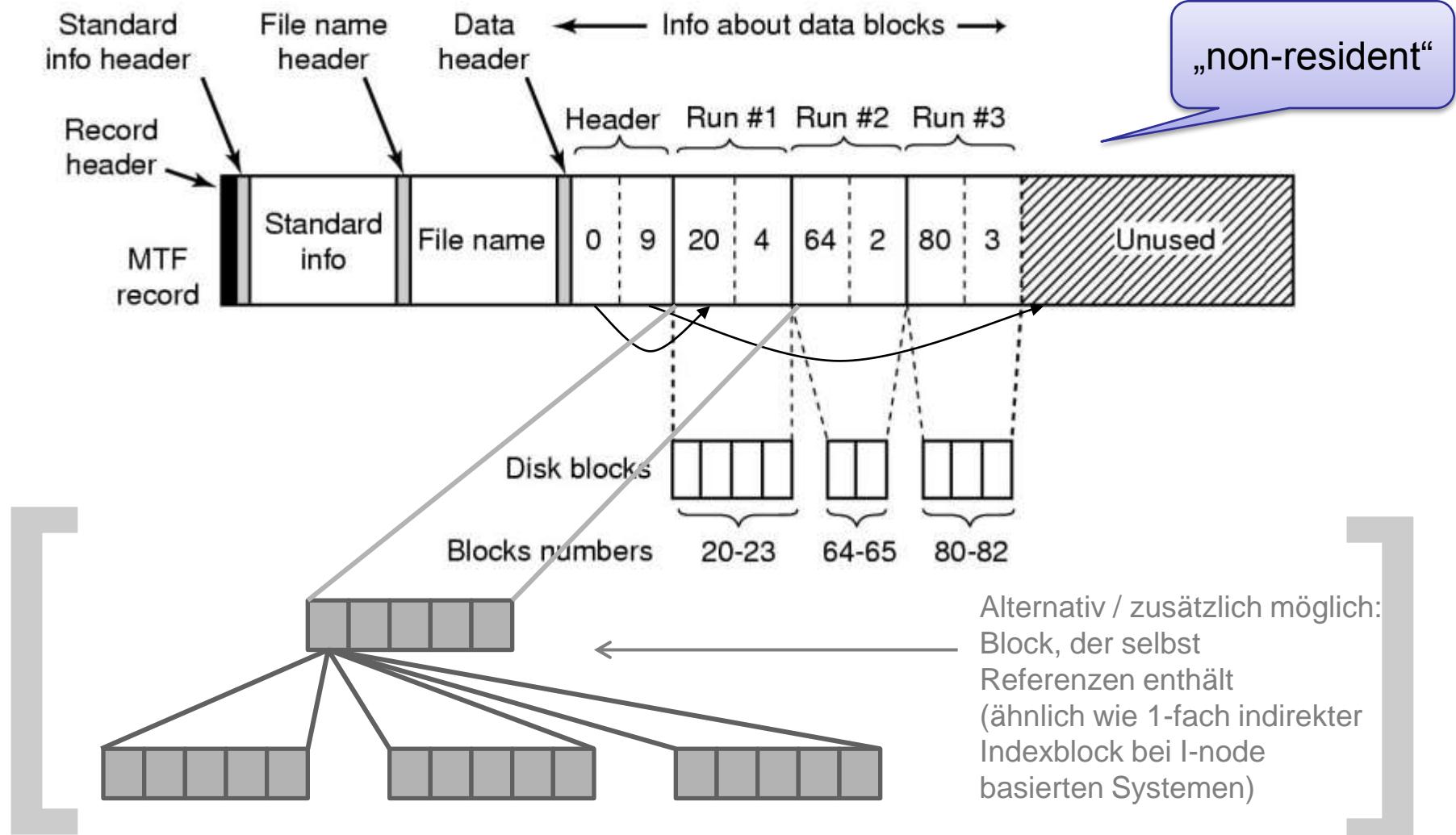
Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in \$Extend\$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in \$Volume)
Volume information	Volume version (used only in \$Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to \$LogFile
Data	Stream data; may be repeated

NTFS: Kleine Datei

- Bei kleinen Dateien wird der Dateiinhalt direkt im MFT Datensatz gespeichert

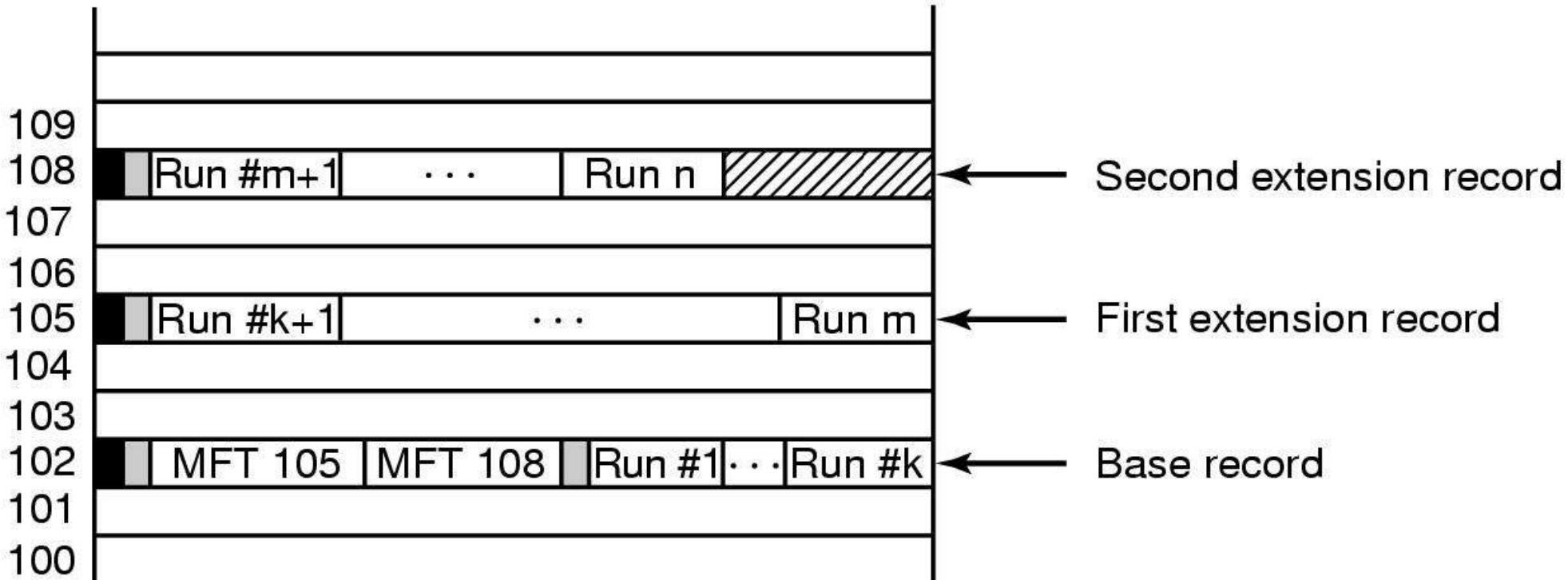


NTFS: Große Datei



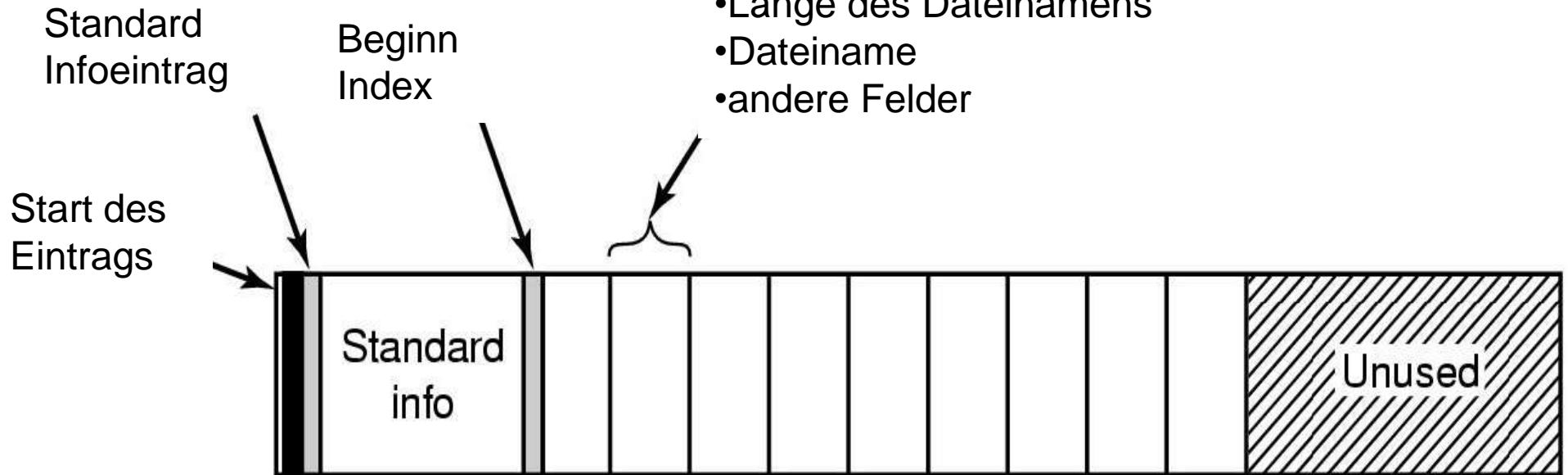
MFT Eintrag für ein großes File, 3 Sequenzen, 9 Blöcke

NTFS: Datei mit vielen Serien



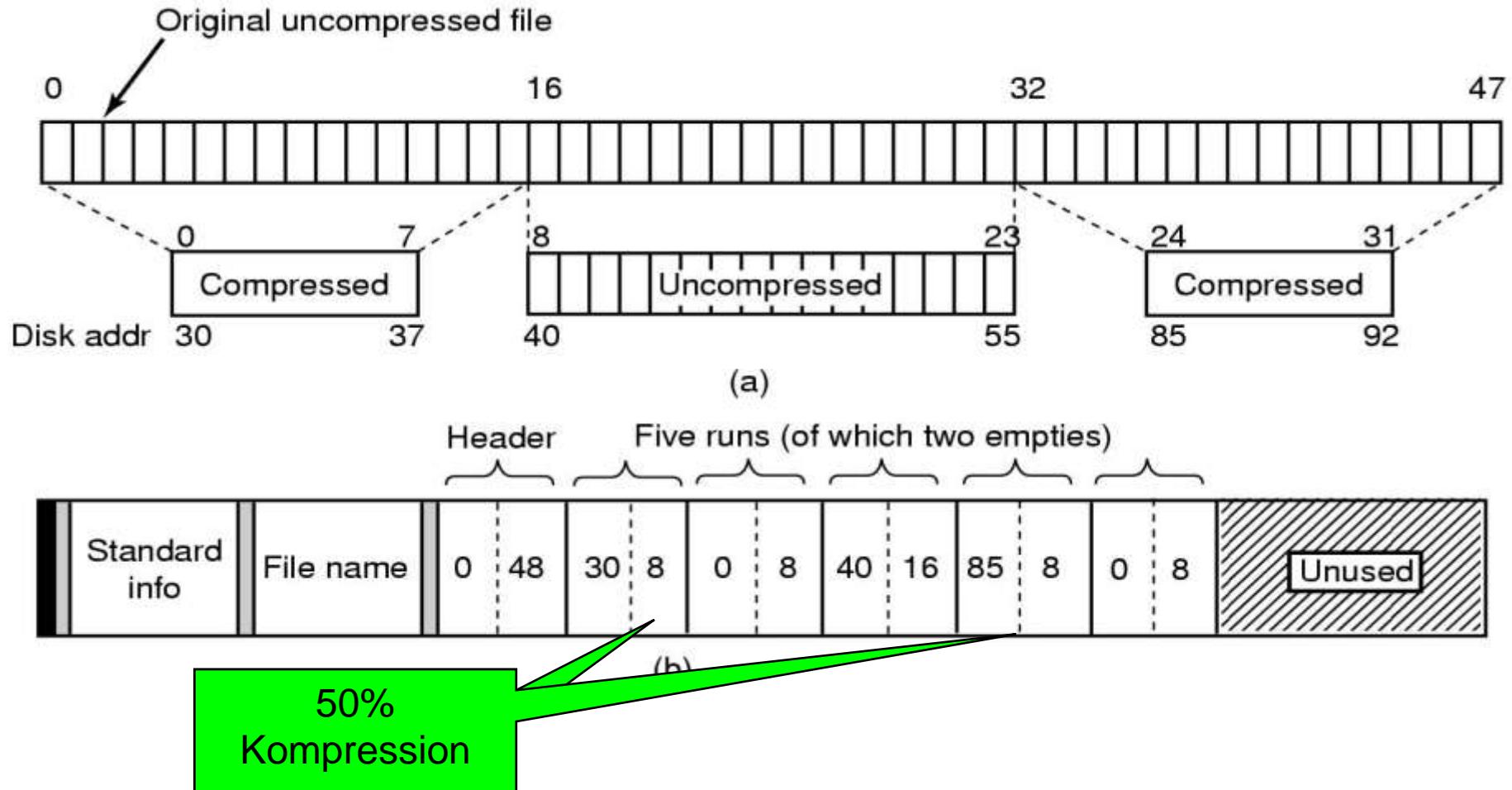
Ein File, das viele Einträge benötigt

NTFS: kleines Verzeichnis



MFT Eintrag für ein kleines Verzeichnis.

NTFS Dateikompression



(a) Beispiel einer 48-Block Datei, komprimiert zu 32 Blöcken

(b) MFT Eintrag

UNIX Filesysteme

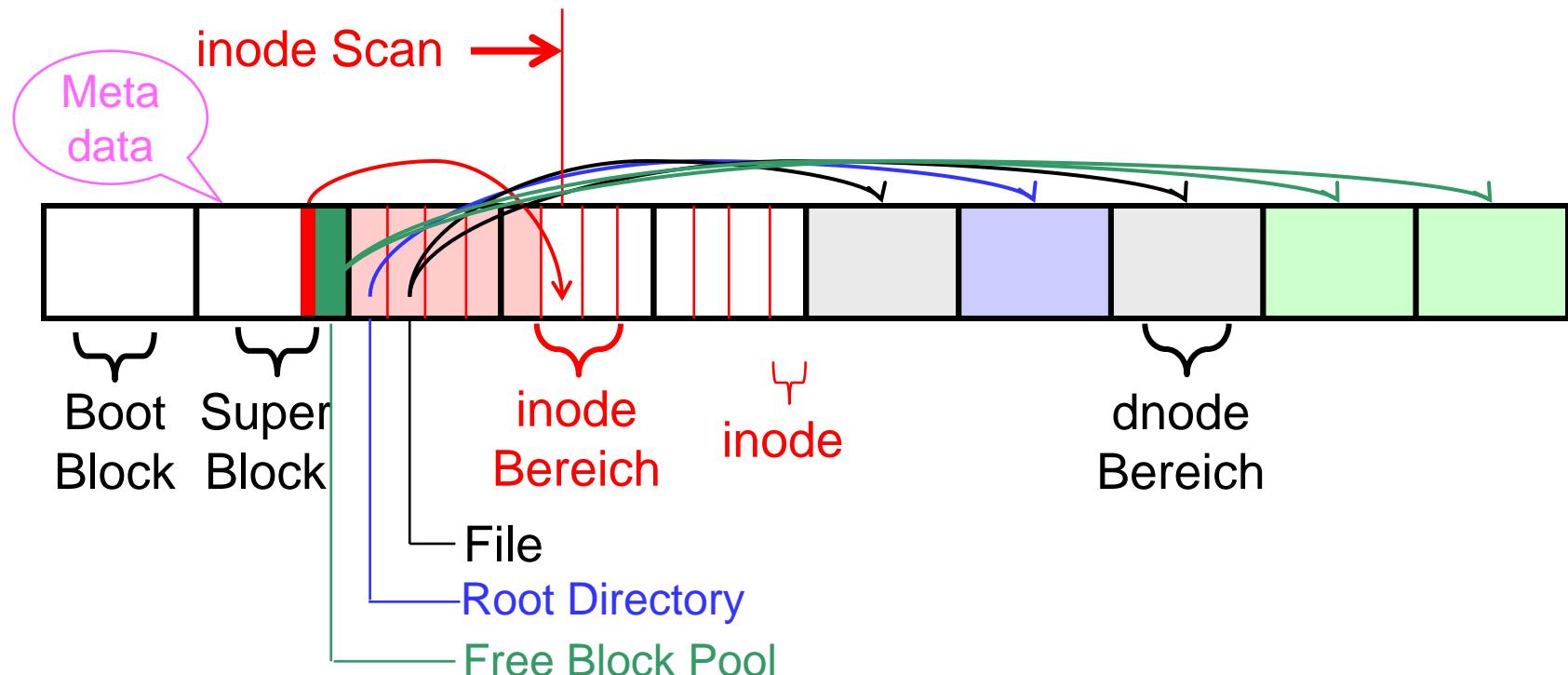
Konzept

- Verwendung von inodes / dhodes / vnodes* zur vereinheitlichten Beschreibung aller Ingredienzen des Filesystems („everything is a file“)
 - Verzeichnis
 - File
 - Spezielles File: Device (zeichenorientiert oder blockorientiert)
 - Pool der freien Blöcke
- Superblock zur Beschreibung der *Metadaten* des Trägervolumes

*virtual node

UNIX Filesysteme

Volume Layout



Unix Filesysteme

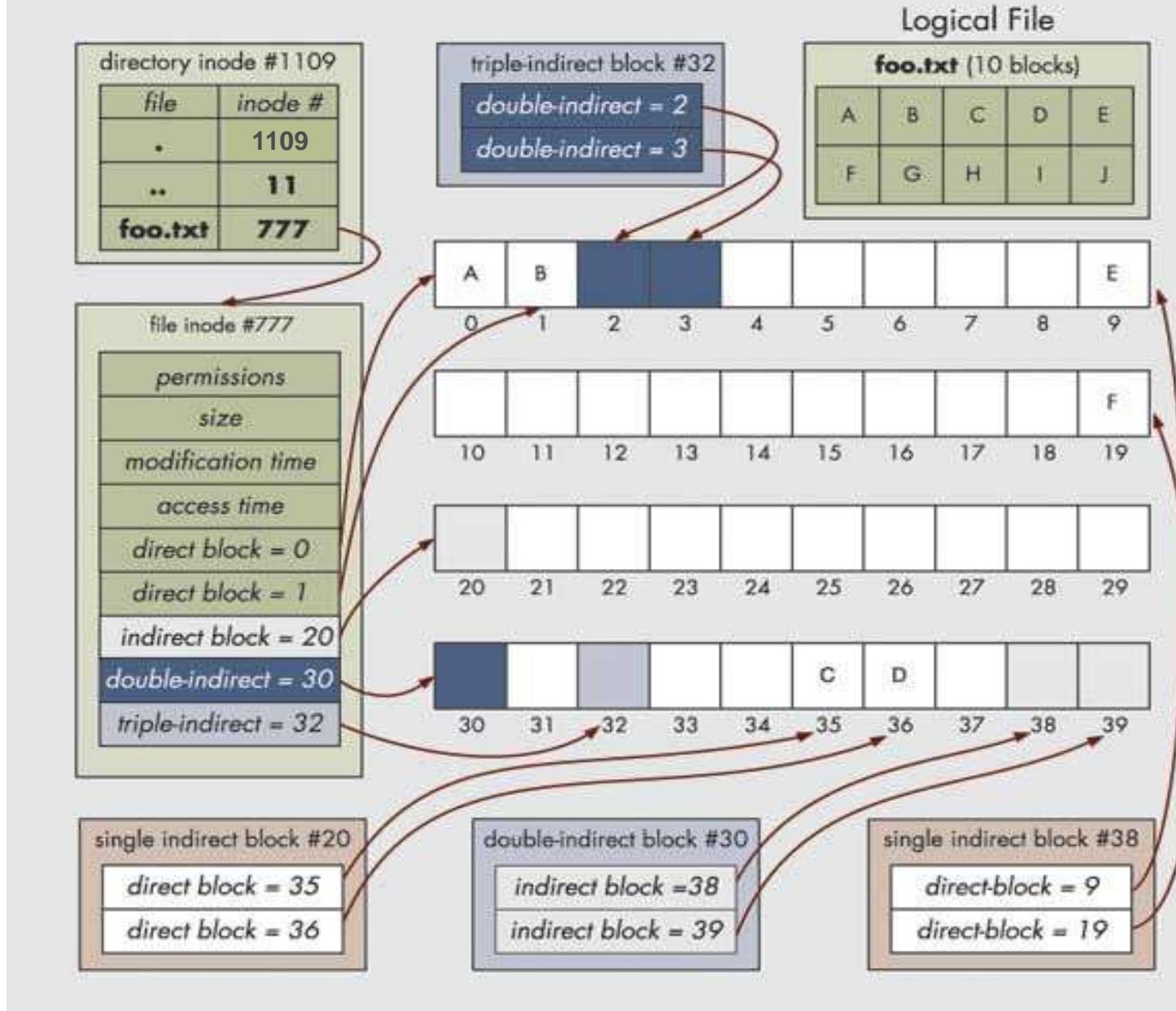
Metadaten

- Superblock
 - Anzahl Blöcke, Anzahl I-Nodes
 - Blockgrösse
 - Zeiger auf freien inode
 - Zeiger auf freien Datenblock
 - Statusbits (z. B. „inkonsistent“)
- inode (File Header)
 - Besitzer
 - Typ (File, Verzeichnis, Device, Pipe, etc.)
 - Zugriffsrechte
 - Zeit des letzten Zugriffs bzw. Modifikation
 - Anzahl der (Hard-)Links zu diesem inode
 - Datenblockzeiger
 - Filegrösse (bei regulären Files)
Devicenummer (bei Devices)

UNIX Filesystems

Beispiel

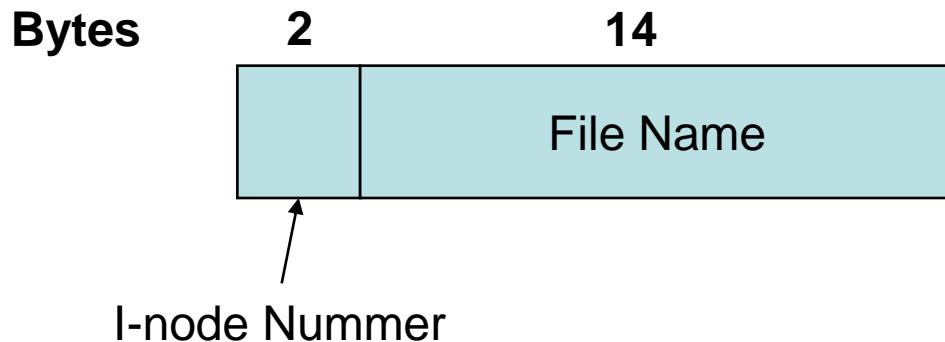
Steve Best, Journaling File Systems, Feature Story, October 2002



UNIX Filesysteme

Verzeichnis Einträge

- UNIX V7 Eintrag



- Ext2 FS Eintrag (lange Dateinamen)



Unix Filesysteme

Verzeichnisse

Root directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Looking up
usr yields
i-node 6

I-node 6
is for /usr

Mode	
size	
times	
132	

I-node 6
says that
/usr is in
block 132

Block 132
is /usr
directory

6	•
1	..
19	dick
30	erik
51	jim
26	ast
45	bal

/usr/ast
is i-node
26

I-node 26
is for
/usr/ast

Mode	
size	
times	
406	

I-node 26
says that
/usr/ast is in
block 406

Block 406
is /usr/ast
directory

26	•
6	..
64	grants
92	books
60	mbox
81	minix
17	src

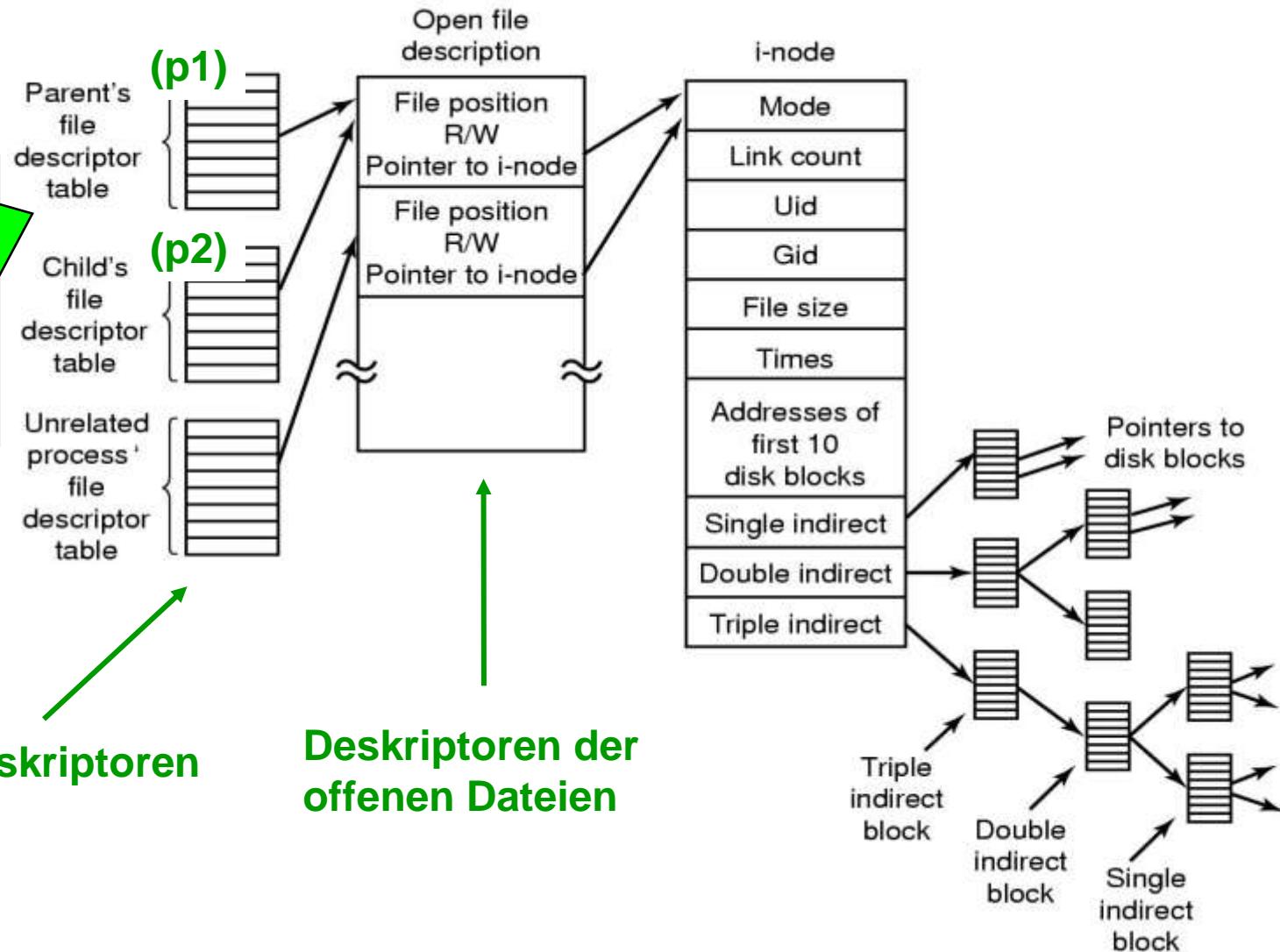
/usr/ast/mbox
is i-node
60

Lokalisieren der Datei /usr/ast/mbox

UNIX Filesysteme

Laufzeitdaten: Beschreibung offener Dateien

skript „s“ bestehend aus zwei Prozessen p1 und p2.
Ausführung von **s > x**

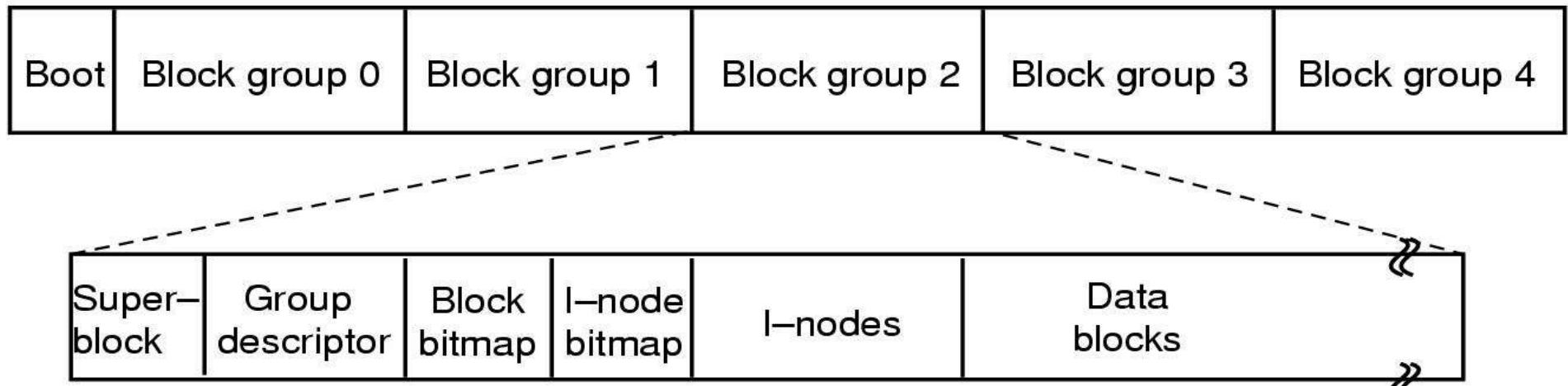


Linux Filesysteme

	MinixFS 1987	Ext FS 1992	Ext2/3 FS 1993	Ext4 FS 2006- 2008	ReiserFS (Vs. 3.6) 2001	XFS 1994
Max. FS Größe	64 MiB	2 GiB	2-16 TiB*	1 EiB (2^{60} B)	256 TiB	8 EiB
Max Dateigröße	64 MiB	2 GiB	16 GiB* - 2TiB*	16 GiB*- 16 TiB*	1 PiB (theor.)	8 EiB
Max Länge Filename	16/30	255	255	255	Blockgröße-64 (z.B. B=4KiB l=4032)	255
Var. Blockgrößen	nein	nein	ja	ja	ja	ja
Journaling	nein	nein	nein (Ext3:ja)	ja	ja	ja

* blockgrößenabhängig 121

Linux File System (Ext2-Ext4)



Layout des Linux Ext2/Ext3/Ext4 Filesystems.

- Ext3 = Ext2 + Journaling + H-Baum Verzeichnisindizes +..
- Ext4 = Ext3 + Extents + 48 Bit Block Nummern +..

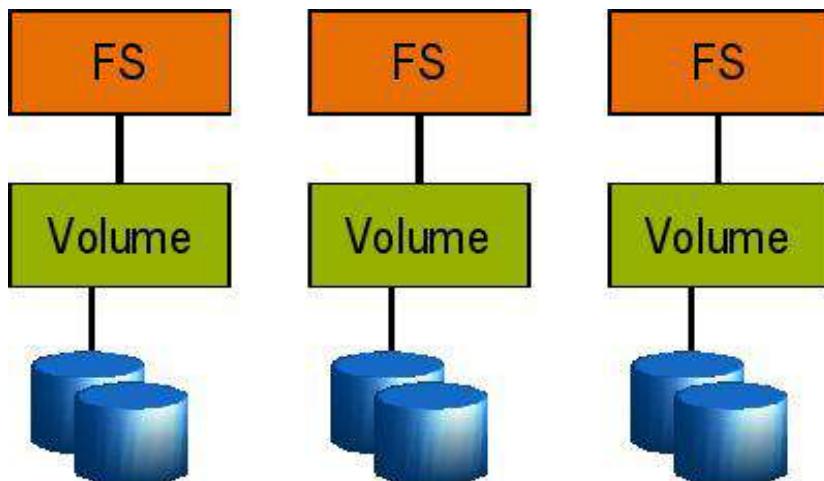
ZFS

- 128-bit Dateisystem mit integriertem Volume-Management
 - ursprünglich entwickelt für Sun Solaris 10, verfügbar unter OpenSolaris (z.B. Belenix), MacOS ab 10.5
 - Jeff Bonwick, Chefentwickler von ZFS:
"Populating 128-bit file systems would exceed the quantum limits of earth-based storage. You couldn't fill a 128-bit storage pool without boiling the oceans." (Aussage wahr für 1/100 Kubikkilometer, Übertreibung als PR-Gag).

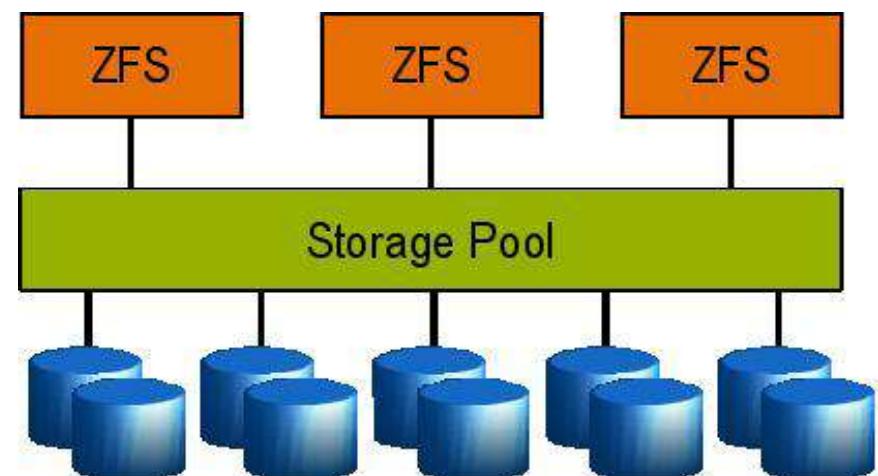
Wortlänge	128-Bit ($\approx 10^{38}$)
Max Anzahl Dateien	2^{48}
Max Größe des Dateisystems	16 Exbibytes ($16 \cdot 1024^6 = 2^{64} \approx 18 \cdot 10^{18}$)
Max Größe File	16 Exbibytes
max Anzahl Dateien pro Verzeichnis	2^{56} (limitiert durch Max. Anzahl Dateien auf 2^{48})

ZFS: Pools

- logische Einheit: Pool
 - beliebig viele logische Partitionen zusammengefasst
 - dynamisch vergrößerbar



traditionelle Aufteilung in Volume Manager und Dateisystem

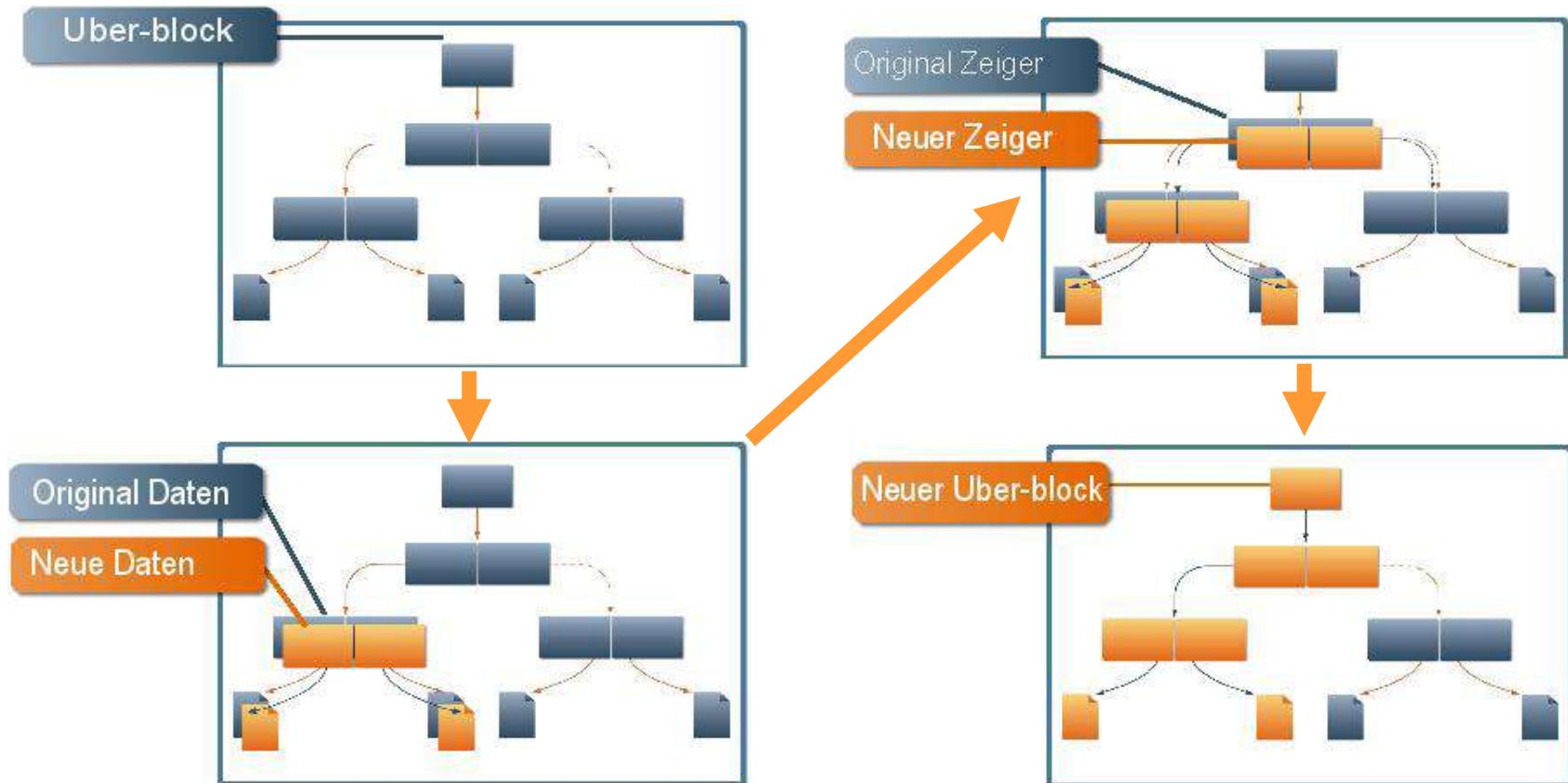


Storage Pools zur Abstrahierung des Speichers

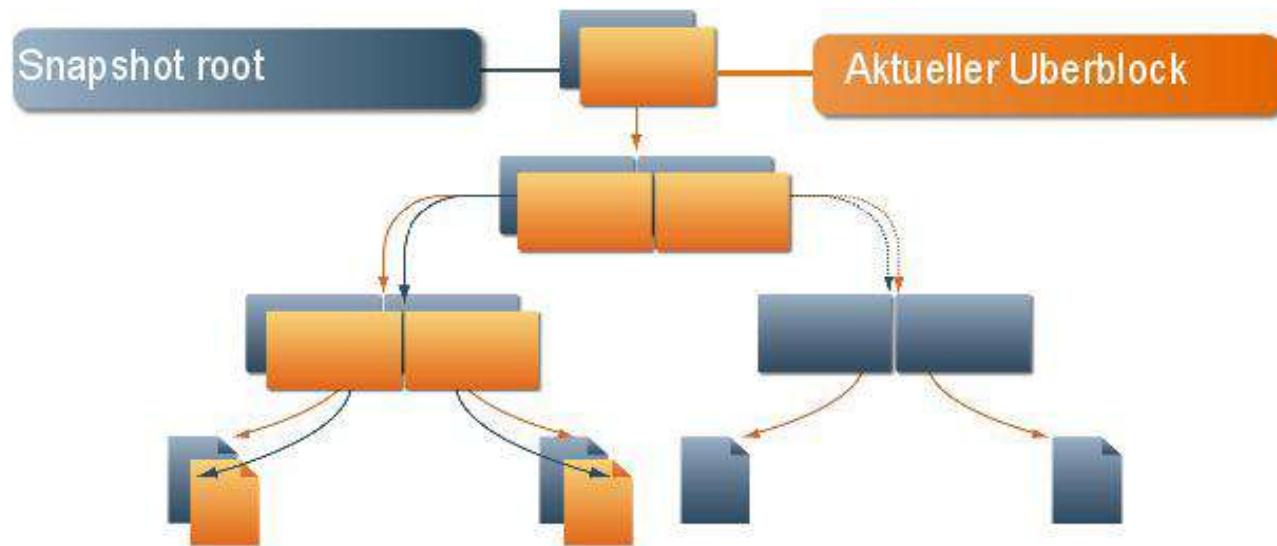
Abbildungen aus: Das OpenSolaris ZFS Dateisystem, Constantin Gonzalez, Sun Microsystems
http://mediacast.sun.com/users/constant/media/LT2007OpenSolarisZFSPaper_v1.1.pdf

ZFS: Copy On Write

- transactional update model
- garantierte Konsistenz des FS zu jeder Zeit

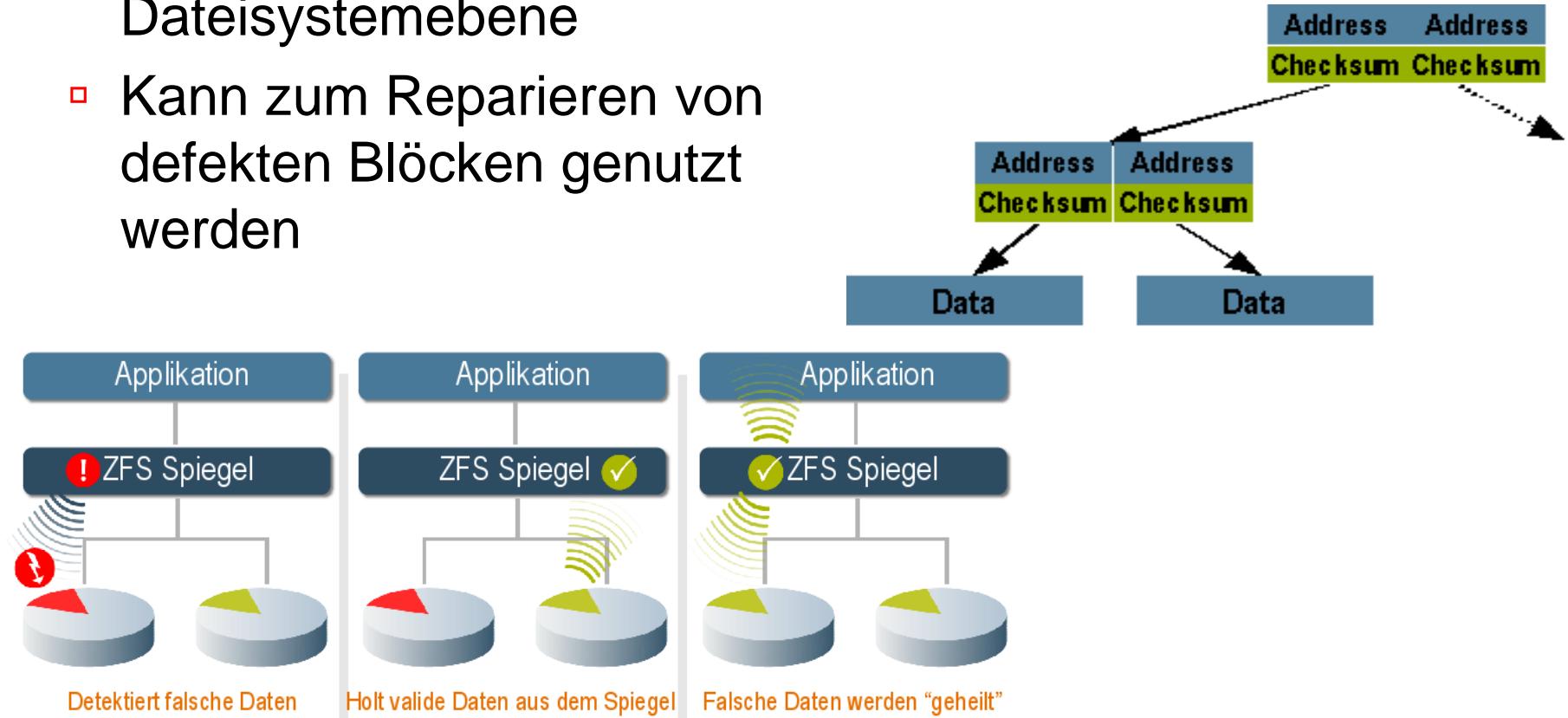


ZFS: Snapshots



ZFS: Checksummen

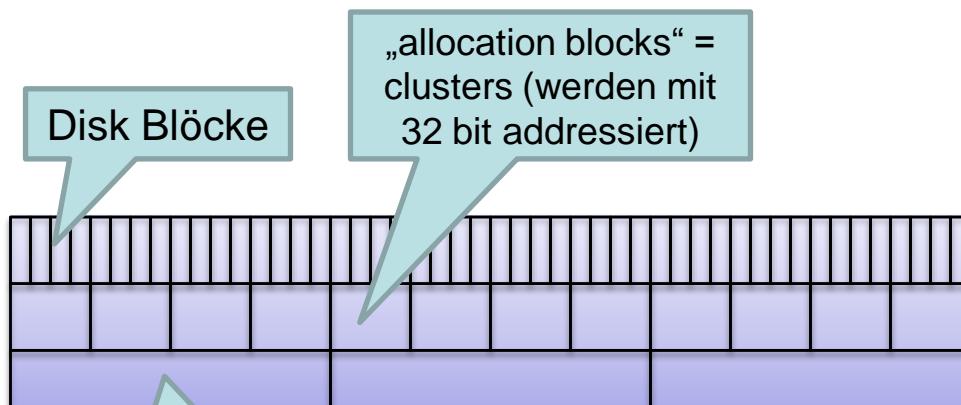
- Rekursive Bildung von Checksummen
 - Erkennung von (Meta-)Datenblockkorruption auf Dateisystemebene
 - Kann zum Reparieren von defekten Blöcken genutzt werden



HFS+

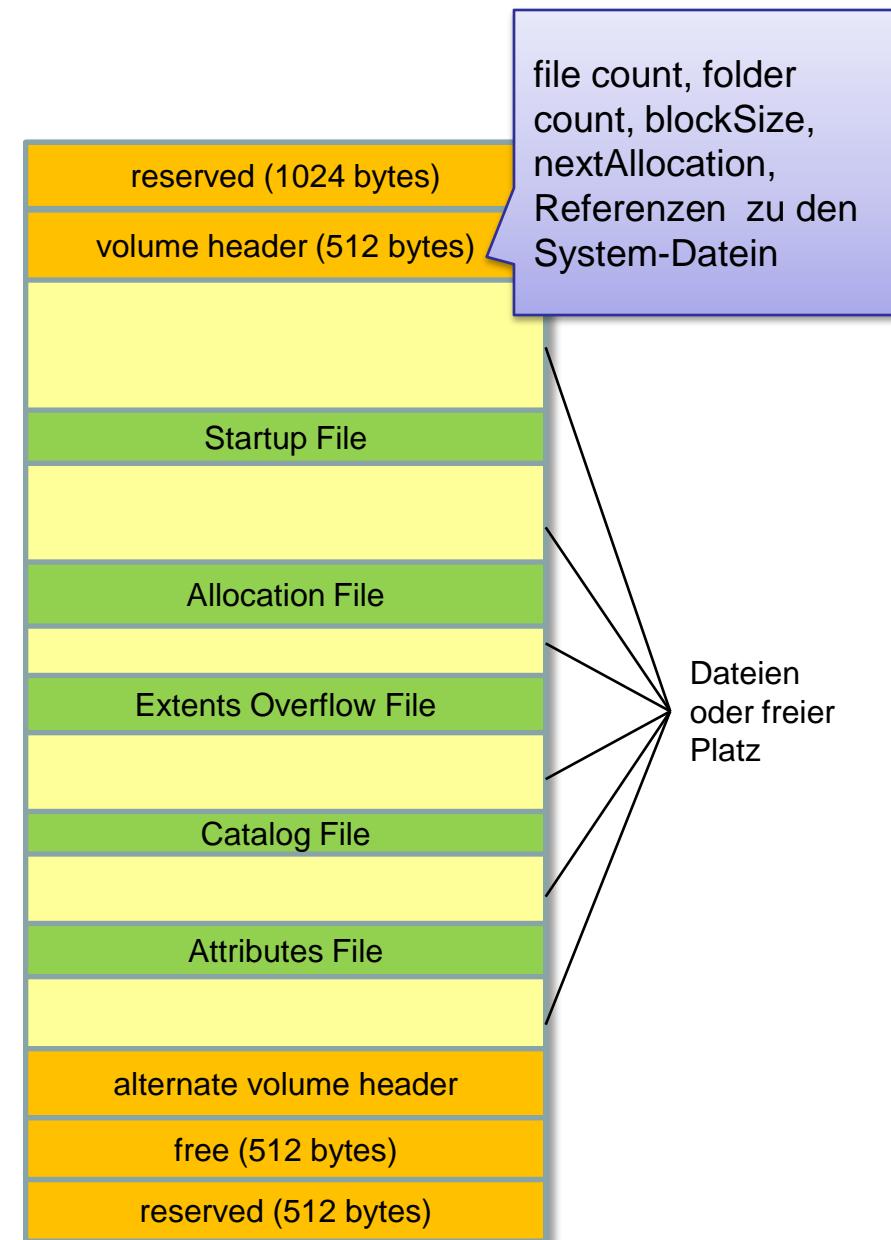
Übersicht

- Filesystem von MacOS X
- Journaling
- 64 Bit basiert, max. Filegröße 2^{63} Byte
- Dateinamen ≤ 255 Zeichen, Unicode
- Dateien in „Clumps“ abgespeichert, verwendet Extents



„clumps“
(Eine Größe pro Datei,
default Wert pro Volume)
Prä-Allokationseinheit

„allocation blocks“ =
clusters (werden mit
32 bit adressiert)



Dateien oder freier Platz

HFS+

Dateistruktur

- Catalog File: Datei mit B-Baum, in dem alle Files referenziert sind (nächste Folie).
- Erste 8 Extents einer Datei direkt in den Fork-Daten des Dateieintrags
- Weitere Extents im Extend Overflow File (B-Baum, sortiert nach Catalog Node ID (CNID))
- Dateien bestehen aus einem „resource fork“ (Konfigurationsdaten) und einem „data fork“ (Eigentliche Daten),

HFSPlusCatalogThread

parentID: CNID des übergeordneten Verzeichnisses

name: Unicode Name (255 Zeichen)

HFSPlusCatalogFile

flags

date

permissions

...

dataFork: HFSPlusForkData

resourceFork: HFSPlusForkData

HFSPlusForkData

clumpSize

totalBlocks

startBlock, blockCount (1)

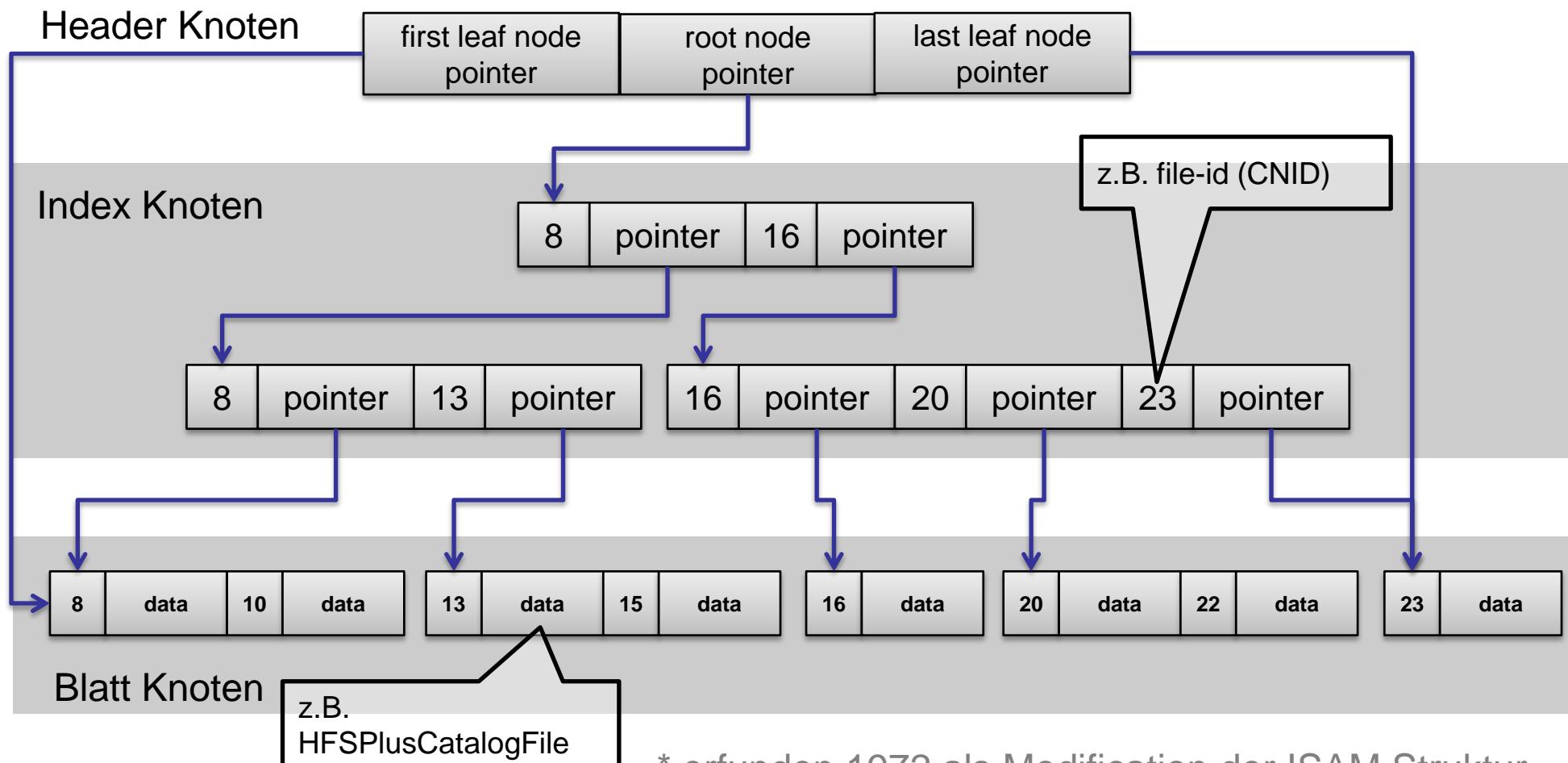
startBlock, blockCount (2)

....

startBlock, blockCount (8)

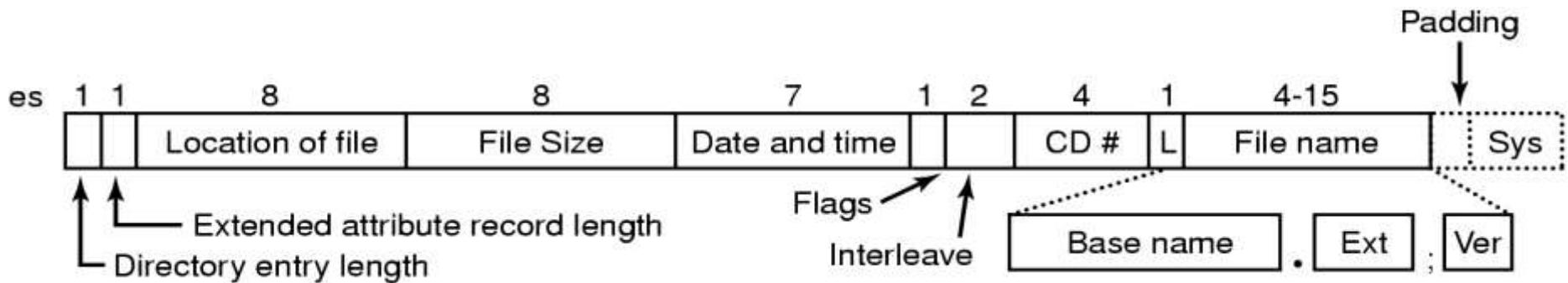
HFS+

Catalog File / Extends Overflow File: B+ Bäume *



* erfunden 1972 als Modifikation der ISAM Struktur

CD-ROM File System



ISO 9660 Directoryeintrag

Erweiterungen:

- Joliet (Microsoft)
- Rock-Ridge (Linux)

Oberon/A2 Filesystem

API, Laufzeitdaten

- API
 - Abstrakte Datentypen *File*, *Rider*
 - Öffnen eines Files (neu oder via Name)
 - Schliessen eines Files
 - Positionieren eines Riders in einem File
 - Lesen des nächsten Bytes via Rider
 - Schreiben des nächsten Bytes via Rider
- Laufzeit Datenstruktur
 - *File Handle* zur Identifikation des Files
 - *Block Puffer* zum blockweisen Lesen und Schreiben
 - *Riders* als Lese- und Schreibzustandsträger

A2- File

File Header

marker	
filename (32 Bytes)	
alen blen	
date time	
Sector Table (512 Bytes)	
ext	
data (3528 Bytes)	

$$\text{len} = \text{alen} * 4096 + \text{blen}$$

FileHeader = RECORD (DiskSector)

```
mark: LONGINT;
name: FileName;
alen, blen: LONGINT;
date, time: LONGINT;
sec: SectorTable;
ext: DiskAddr;
data: ARRAY SS-HS OF CHAR
END;
```

magic number

Datenblock

Data
(4096)

IndexBlock

Index
(4096)

Datenblock

Data
(4096)

IndexBlock

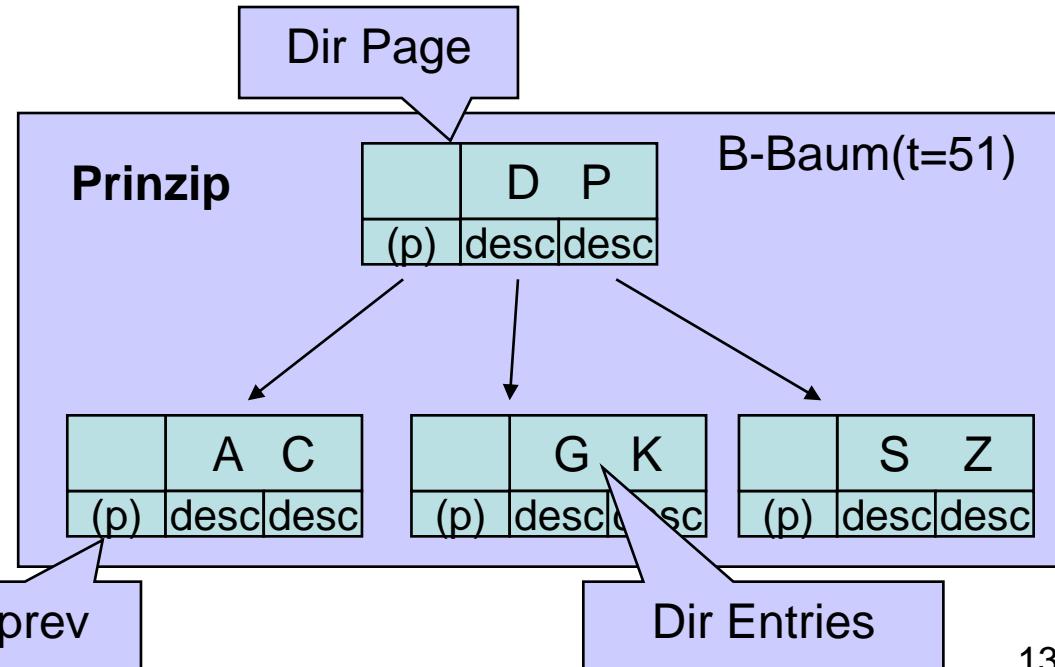
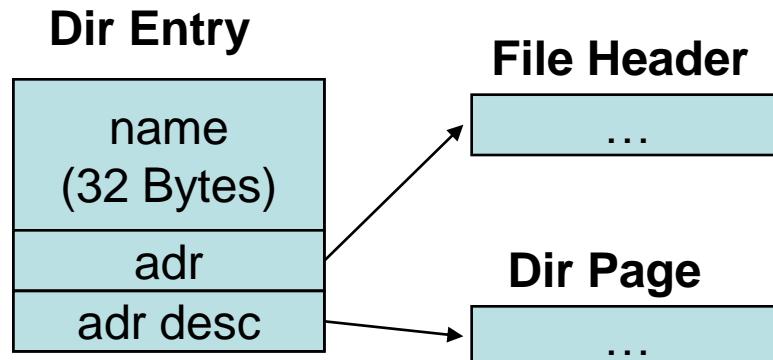
Index
(4096)

Data
(4096)

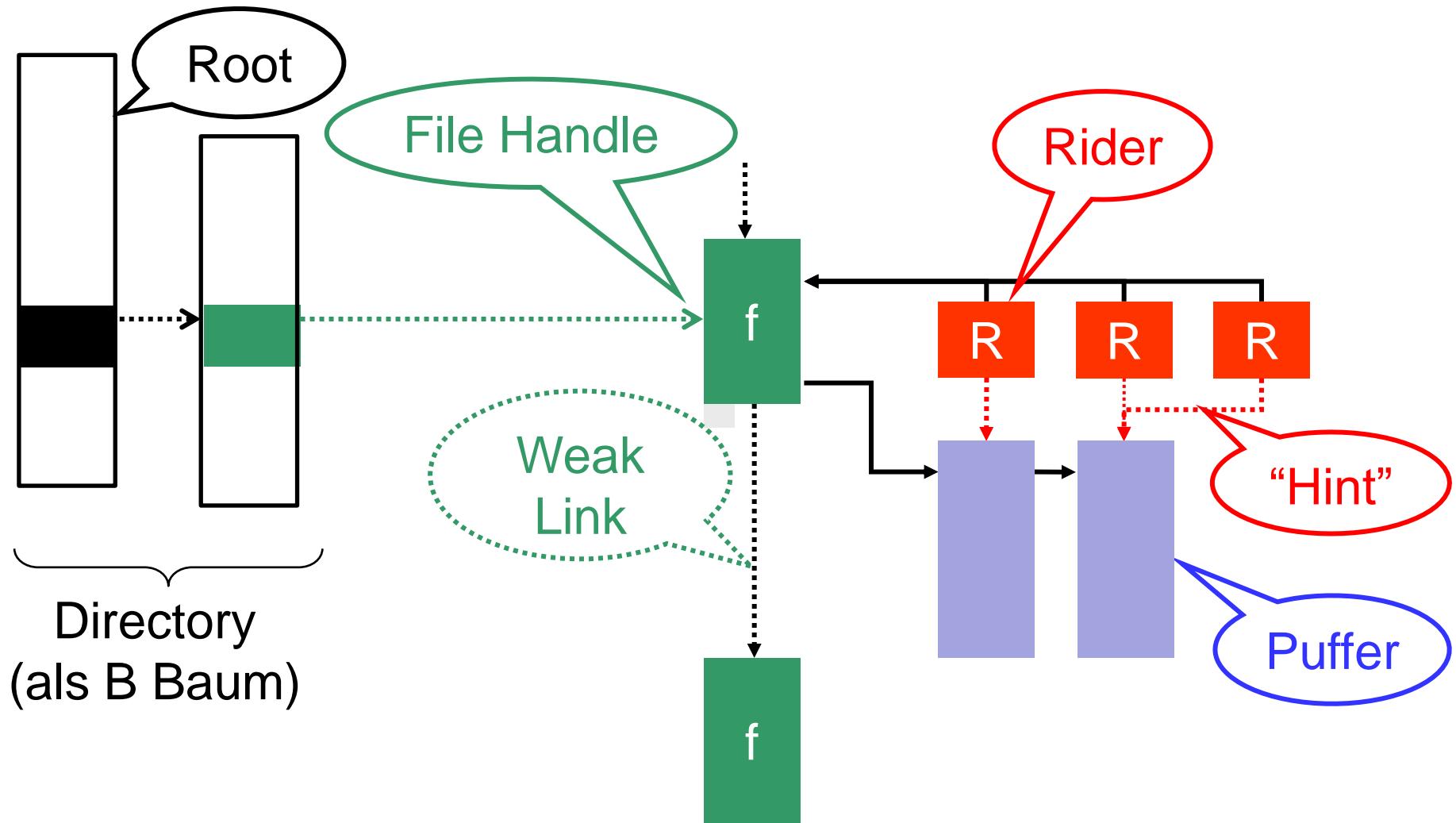
Index
(4096)

A2 – Verzeichnis

Implementation mit B-Baum



Oberon Filesystem Laufzeitstrukturen



2.3. Fault Tolerance

- Fehlerarten
- Konsistenz in Dateisystemen
 - Scavenging
 - Journaling

Das Thema **Datensicherung** gehört auch zum Block „Fault Tolerance“, wird in der Vorlesung jedoch nicht eingehend behandelt. Dazu z.B. Tannenbaum, S.446ff

Fehlerarten

- Hardwarefehler
 - Bit Failure Rate (pro Read/Write Operation): 10^{-15}
- Softwarefehler
 - Inkonsistenz der Diskdaten nach System Crash
 - Erzeugen eines Files
 - A1: Directoryeintrag hinzufügen
 - A2: inode aus free Liste nehmen
 - A3: Attribute im inode setzen
 - A4: Im Directoryeintrag auf inode verweisen
 - Daten schreiben
 - B1: Datenblock aus free Liste nehmen
 - B2: Daten in den Datenblock schreiben
 - B3: Im inode auf Datenblock verweisen

- Operation A {
- A1: Directoryeintrag hinzufügen
 - A2: inode aus free Liste nehmen
 - A3: Attribute im inode setzen
 - A4: Im Directoryeintrag auf inode verweisen
- Crash
- Operation B {
- B1: Datenblock aus free Liste nehmen
 - B2: Daten in den Datenblock schreiben
 - B3: Im inode auf Datenblock verweisen
- Crash

Gegenmassnahmen

- Cyclic Redundancy Check (CRC) pro Sektor
- Pseudofile BADSECTORS zur dauerhaften Absorption fehlerhafter Sektoren
- Redundanz zur Rekonstruktion der Filesystemdatenstruktur (Scavenging)
 - Repliziere kritische Blöcke (z. B. FAT, Superblock)
 - Markiere File Headerblöcke mit Fingerabdruck
 - Trage Filenamen in File Headerblöcke ein
 - Verkette Datenblöcke sequenziell
- Journaling zur (schnellen) Behebung von Inkonsistenzen
 - Logging aller Transaktionen (Metadaten)

Konsistenz in Dateisystemen

Typische Fehlerbereinigung – fsck/scandisk

BLOCK NUMBER																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	1
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	0

(a)

BLOCK NUMBER																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	1	0	0
0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1	0

(b)

BLOCK NUMBER																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	1	0	0
0	0	0	0	2	0	0	0	0	1	1	0	0	0	1	1	0

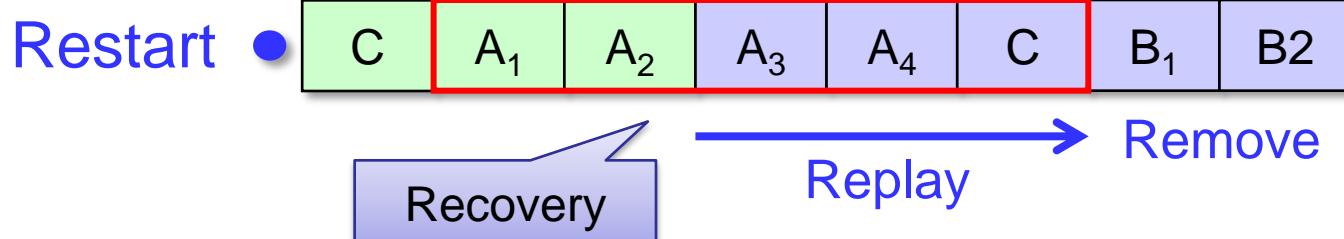
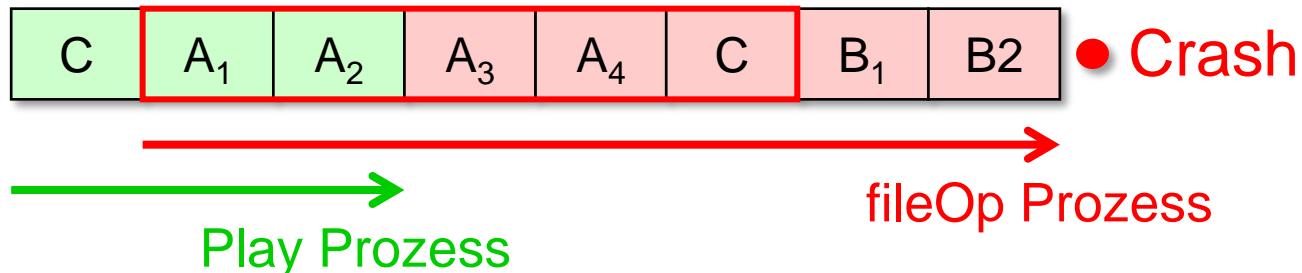
BLOCK NUMBER																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	2	1	1	1	0	0	1	1	1	1	0	0
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	0

Dateisystemzustände

- (a) konsistent
- (b) fehlender Block
- (c) Block zweifach in free-List
- (d) Block zweifach in Datenliste

Journaling / Log-basierte Dateisysteme

- Fileoperationen als Transaktionen
 - $\text{fileOp} = \{ A_1; A_2; \dots; A_n; \}$
- Journal
 - Implementiert z.B. als zirkulärer Puffer aus Blöcken auf der Disk
- Ablauf

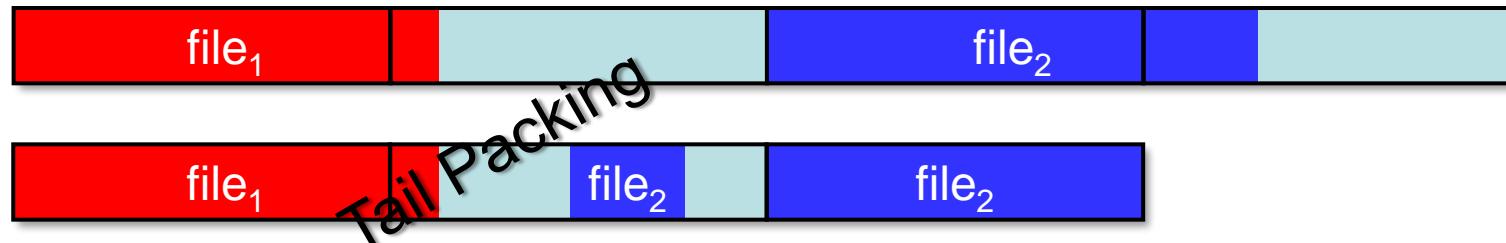


2.4. Optimierungen

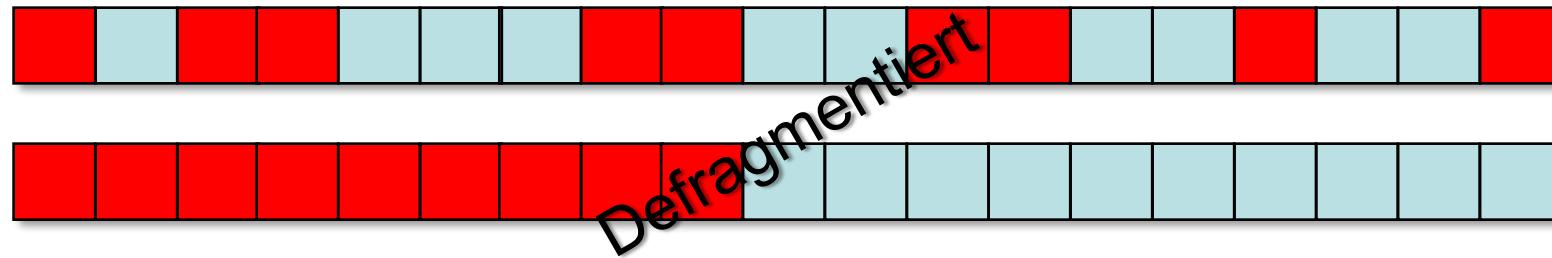
- Tail Packing / Subblock-Allokation
- Defragmentierung, Verwendung von Extents
- Caching
- Disk-Arm-orientiertes Scheduling
- RAID Systeme

Fragmentierung

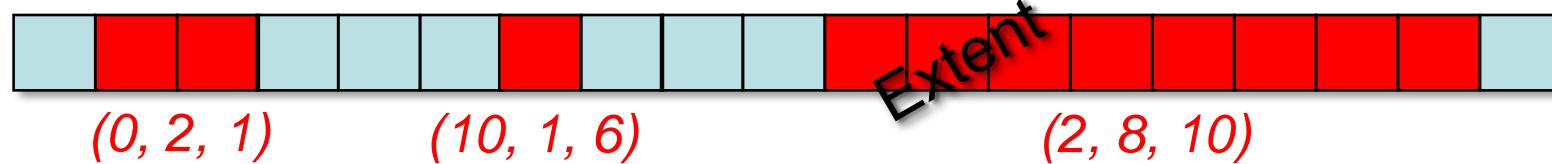
- Interne Fragmentierung \Rightarrow Speicherineffizienz



- Externe Fragmentierung \Rightarrow Zeitineffizienz

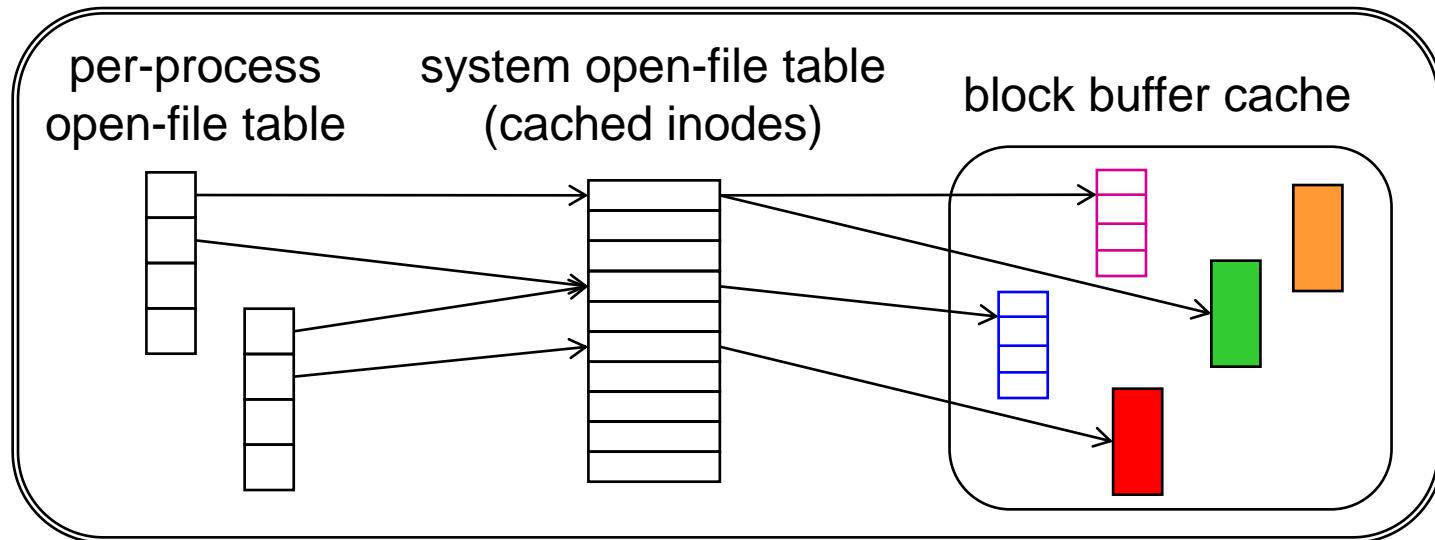


- Verwendung von Extents (*pos in file, length, start block*)

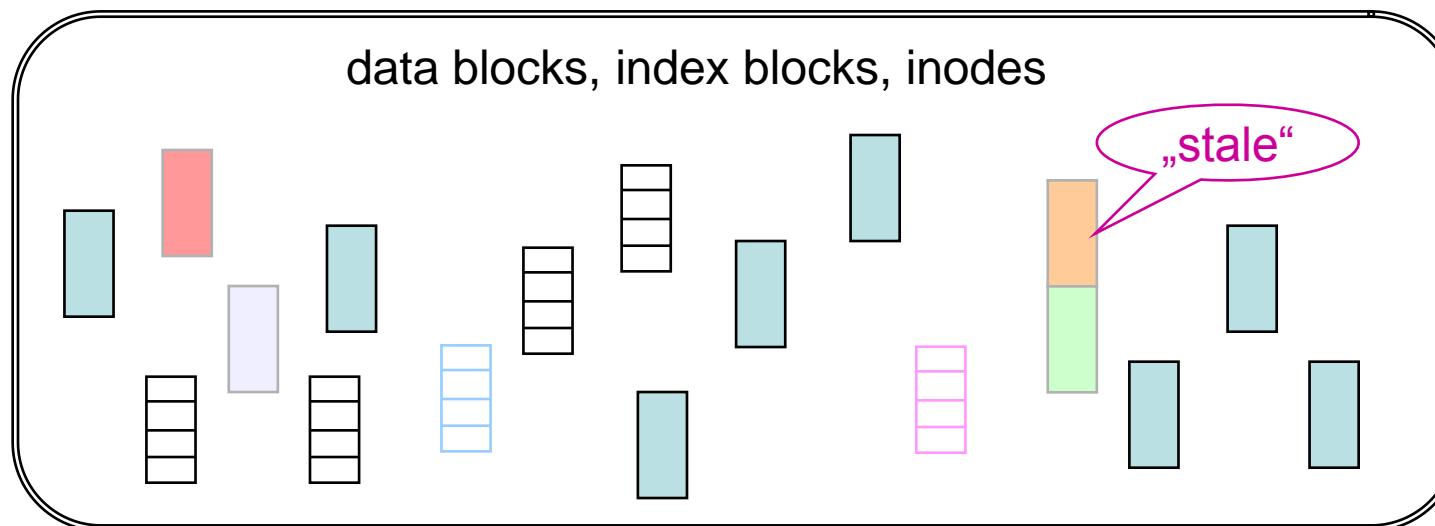


Caching

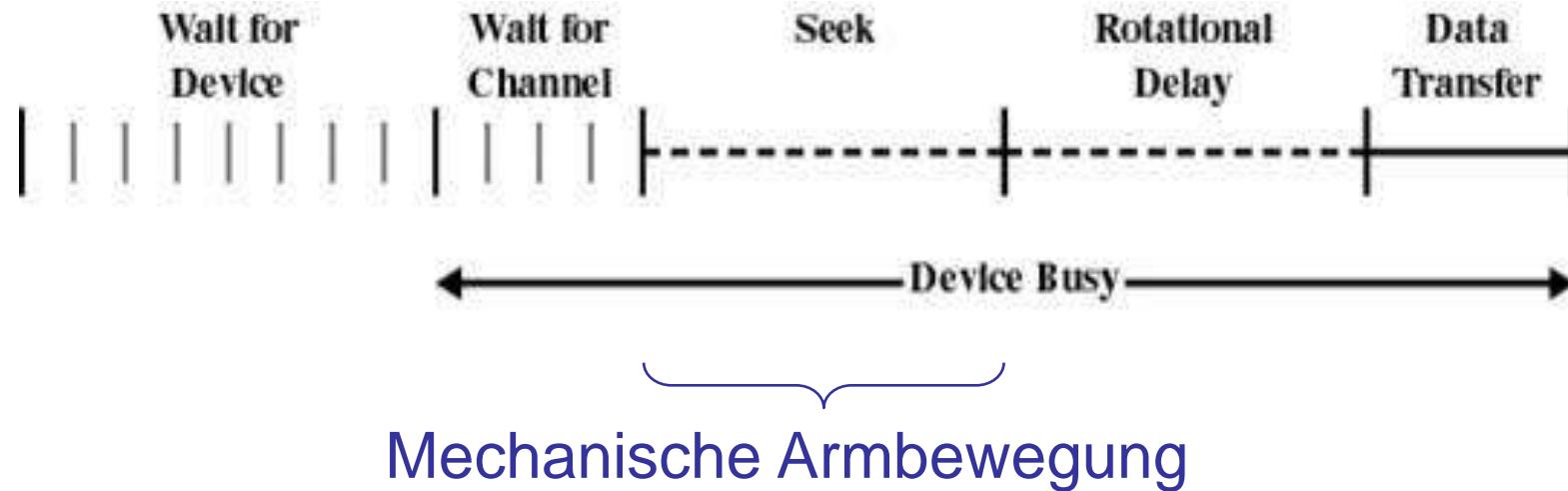
Hauptspeicher
(flüchtig)



Sekundärspeicher
(persistant)



Geometriefaktoren



$$\text{Gesamte Zugriffszeit} = T + \frac{1}{2r} + \frac{b}{rN}$$

T: Mittlere Positionierungszeit [s]

r: Rotationsgeschwindigkeit [U/s]

b: # Bytes

N: # Bytes / Spur

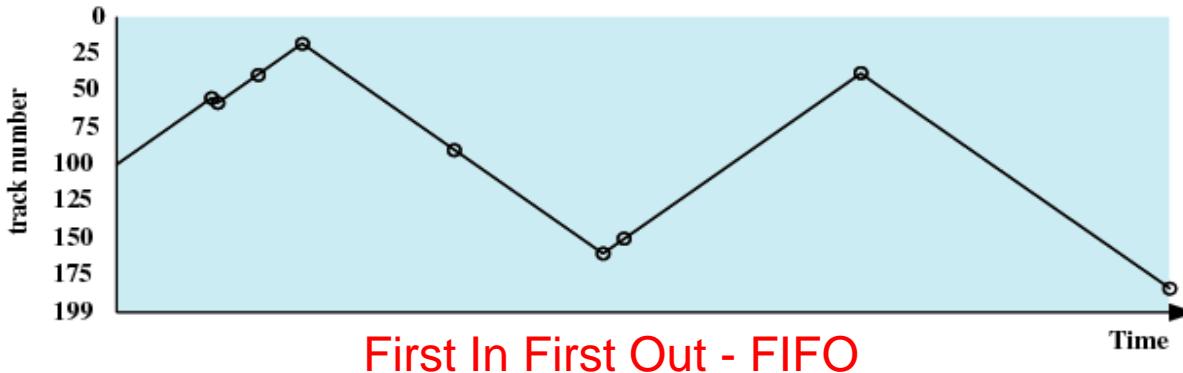
mittlere
Rotationsverzögerung

Übertragungszeit

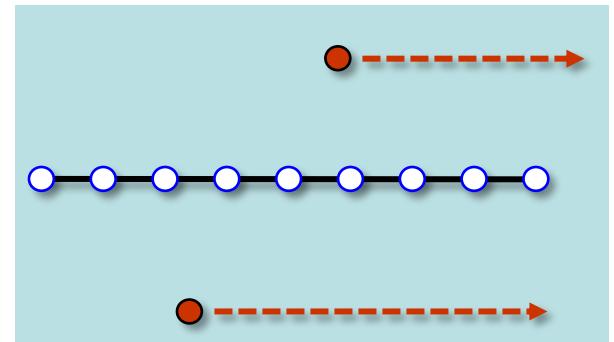
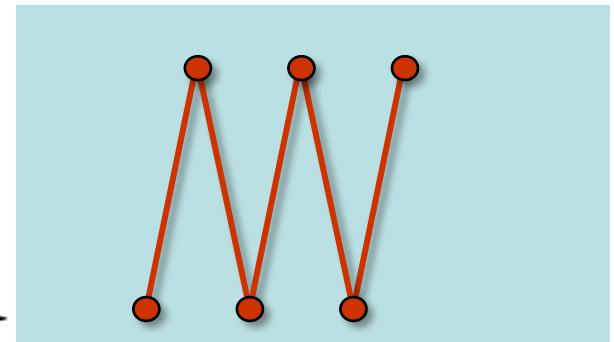
Optimierungsmöglichkeiten

Disk-Arm-orientiertes Scheduling

Beispiel



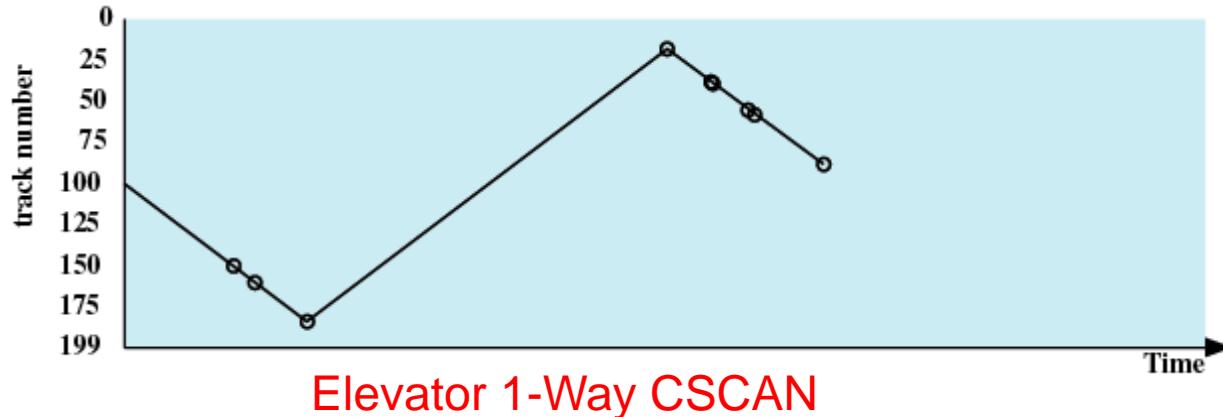
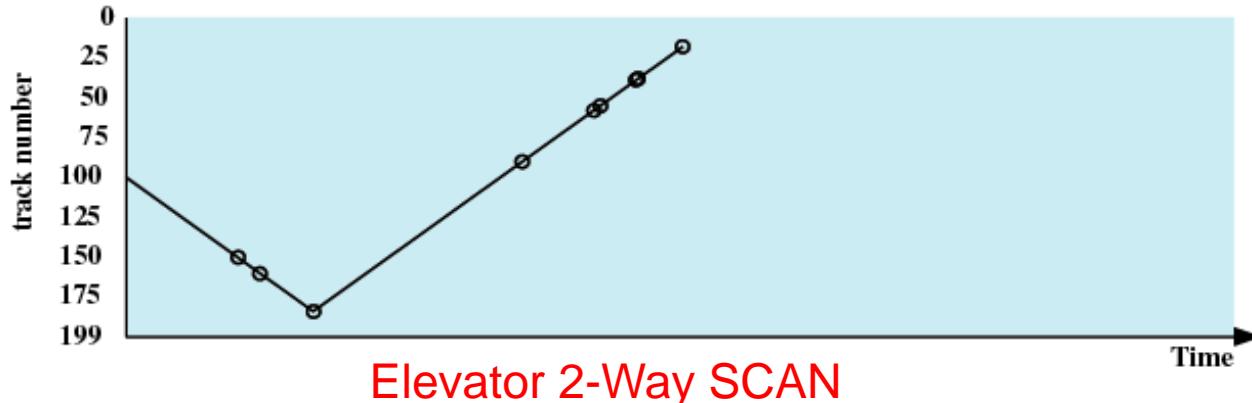
Worst Case Scenario



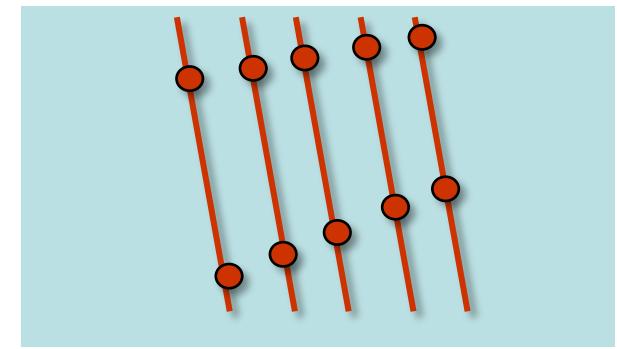
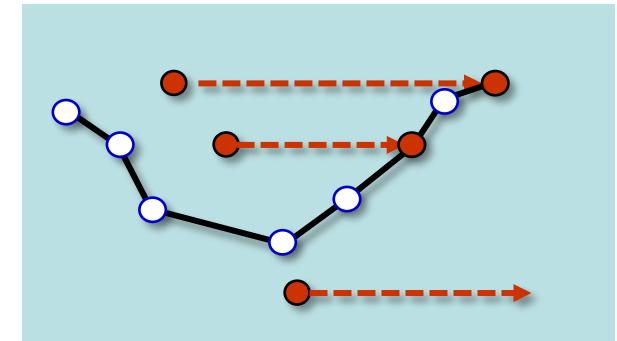
Optimierungsmöglichkeiten

Disk-Arm-orientiertes Scheduling

Beispiel



Worst Case Scenario



zusätzlich N-Step SCAN und FSCAN (ohne Abb): Mehrere (Teil)-Warteschlangen

Optimierungsmöglichkeiten (2)

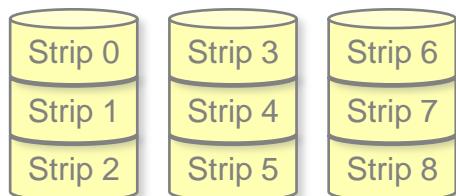
- Anwendungsorientierte Blockgrösse
 - Grosse Blöcke ⇒ Amortisierungspotential
 - Kleine Blöcke ⇒ Geringe interne Fragmentation
- Versetzte Blockallozierung
 - Block Nr. 1, 6, 2, 7, 3, 8, 4, 9, 5 auf Track
 - Block Nr. 1 entlang Zylinder versetzt
- Optimierte Platzierung von Files
 - Häufig zugegriffene Files im Zentrum des Zylinders
 - Replikation & Striping auf verschiedenen Harddisks

RAID Technologie

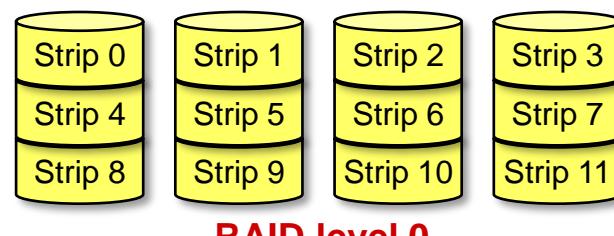
- Redundant Array of Independent Disks
 - Serie physischer Disks als ein logisches Volume
 - Effizienz dank parallelem Lesen/ Schreiben
 - Datensicherheit dank redundanter Speicherung
- Striping
 - Verteilung eines Files quer durch die Diskserie
- Redundanz
 - Replikation von Daten (Mirroring)
 - Hinzufügen von Redundanzinformation (Paritätsbits)
- Definierte Organisationsformen
 - Standard Levels 0 – 6 (1987: 0-5)
 - Darüber hinaus viele mehr
(vgl. <http://de.wikipedia.org/wiki/RAID>)

RAID Levels

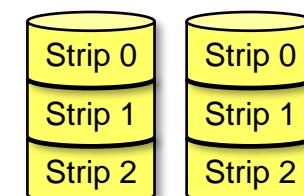
Just a Bunch of Disks



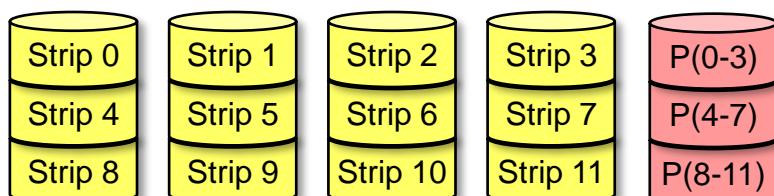
Striped Data



Mirrored Data

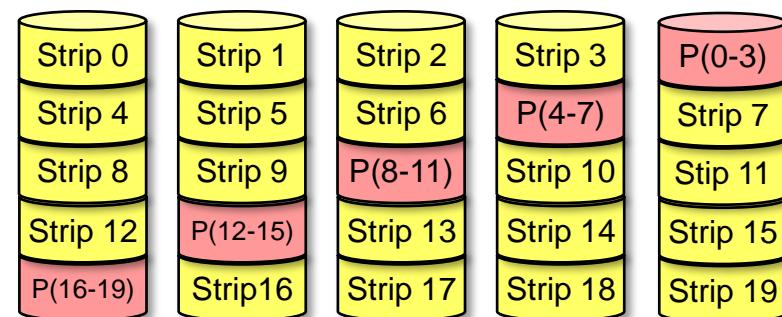


Striped Data + Parity



RAID level 4

Striped Data + Striped Parity

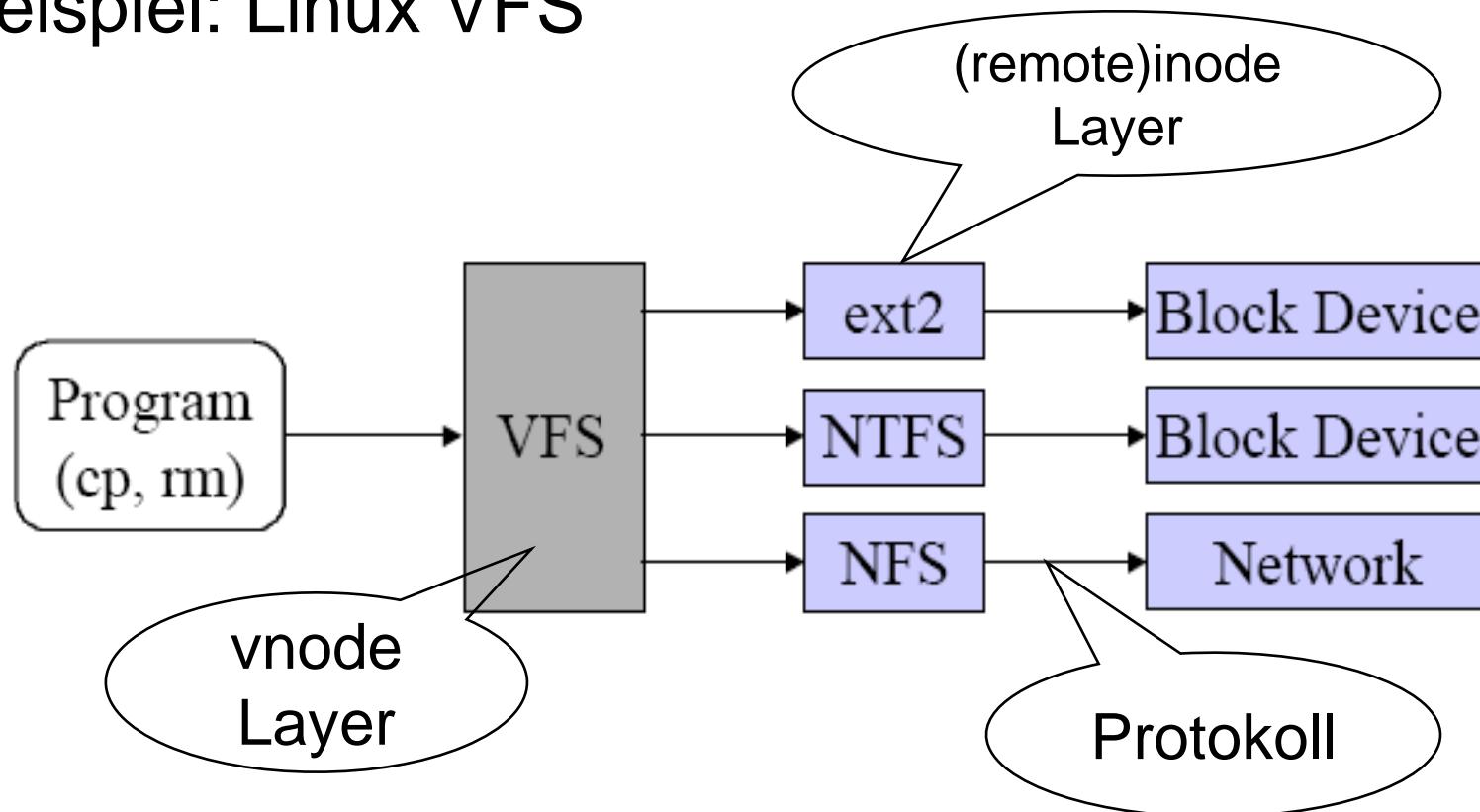


RAID level 5

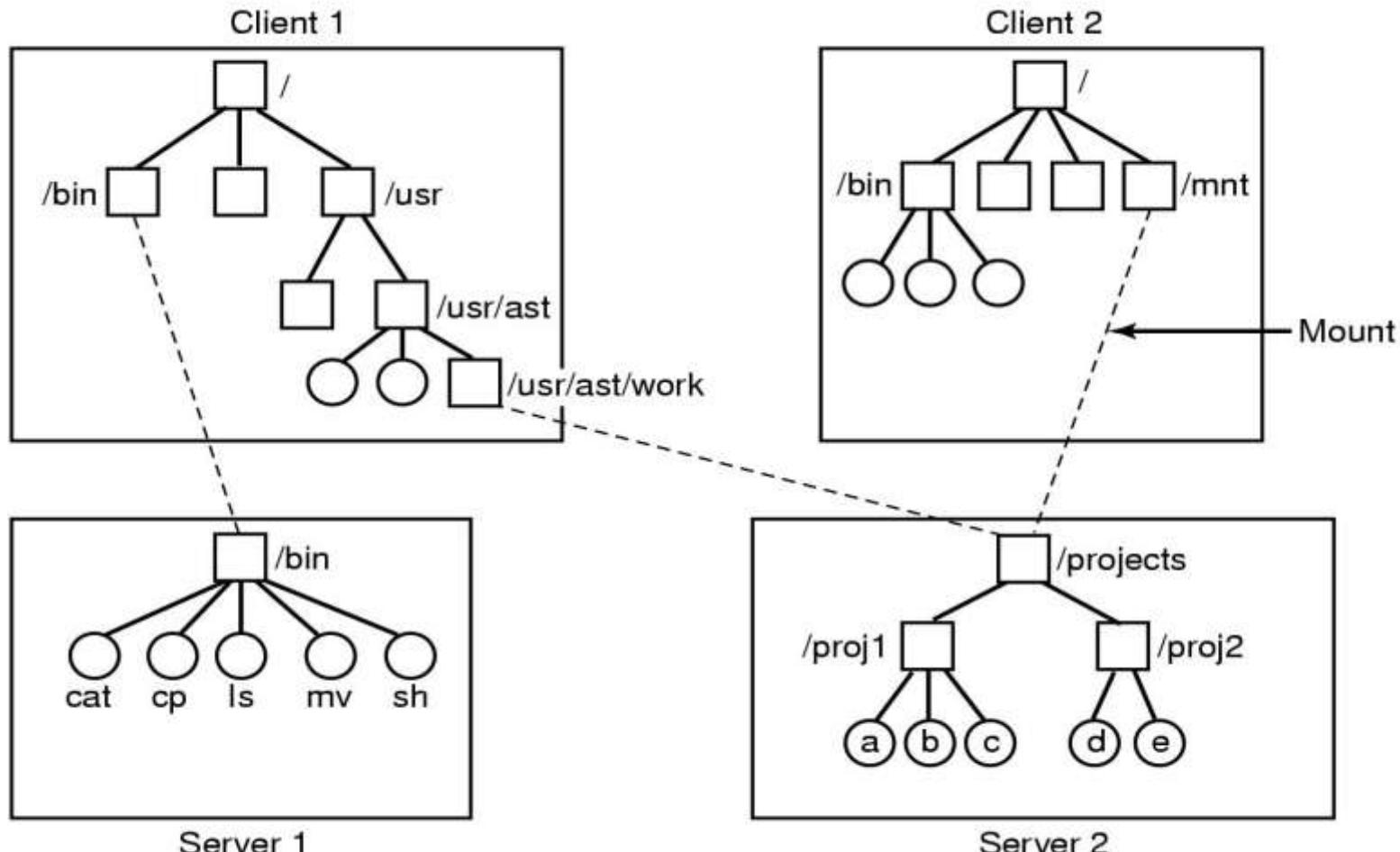
RAID level 2 & 3: synchronisierte Platten mit Fehlerkorrektur

2.5. Virtuelle Filesysteme

- Zusätzliche Abstraktionsschicht zur Unifikation verschiedener (ev. „verteilter“) Filesysteme
- Beispiel: Linux VFS

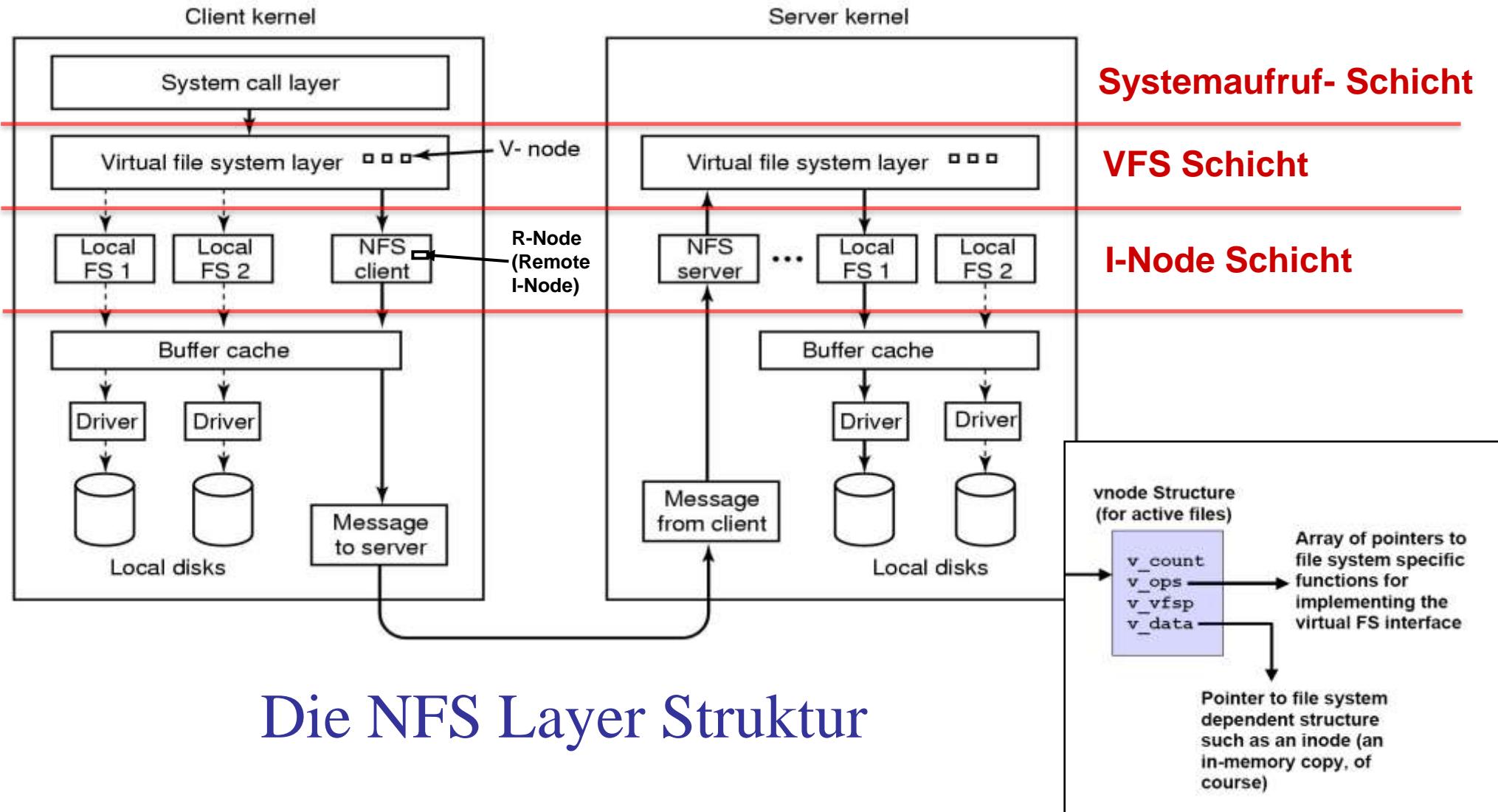


Anwendung: Network File System



- Beispiel Remote Mounted File Systems
- Rechtecke: Verzeichnisse, Kreise: Dateien

Network File System (2)



Kapitel 3. I/O Subsysteme

- 3.1. Hardware Unterstützung
- 3.2. Gerätetreiber Beispiele
- 3.3. Generische Struktur von Gerätetreibern

3.1. Hardware Unterstützung

- 3.1.1. I/O Hardware, Das Bussystem
- 3.1.2. Gerätezugriff: Ports, Memory Mapped I/O
- 3.1.3. Das Interruptkonzept
- 3.1.4. Direct Memory Access
- 3.1.5. Kernel Modus vs. User Modus

3.1.1. I/O Hardware

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

Tanenbaum: Moderne Betriebssysteme, <http://www.cs.vu.nl/~ast>

Unterscheidung in

- **blockorientierte**
und
- **zeichenorientierte**

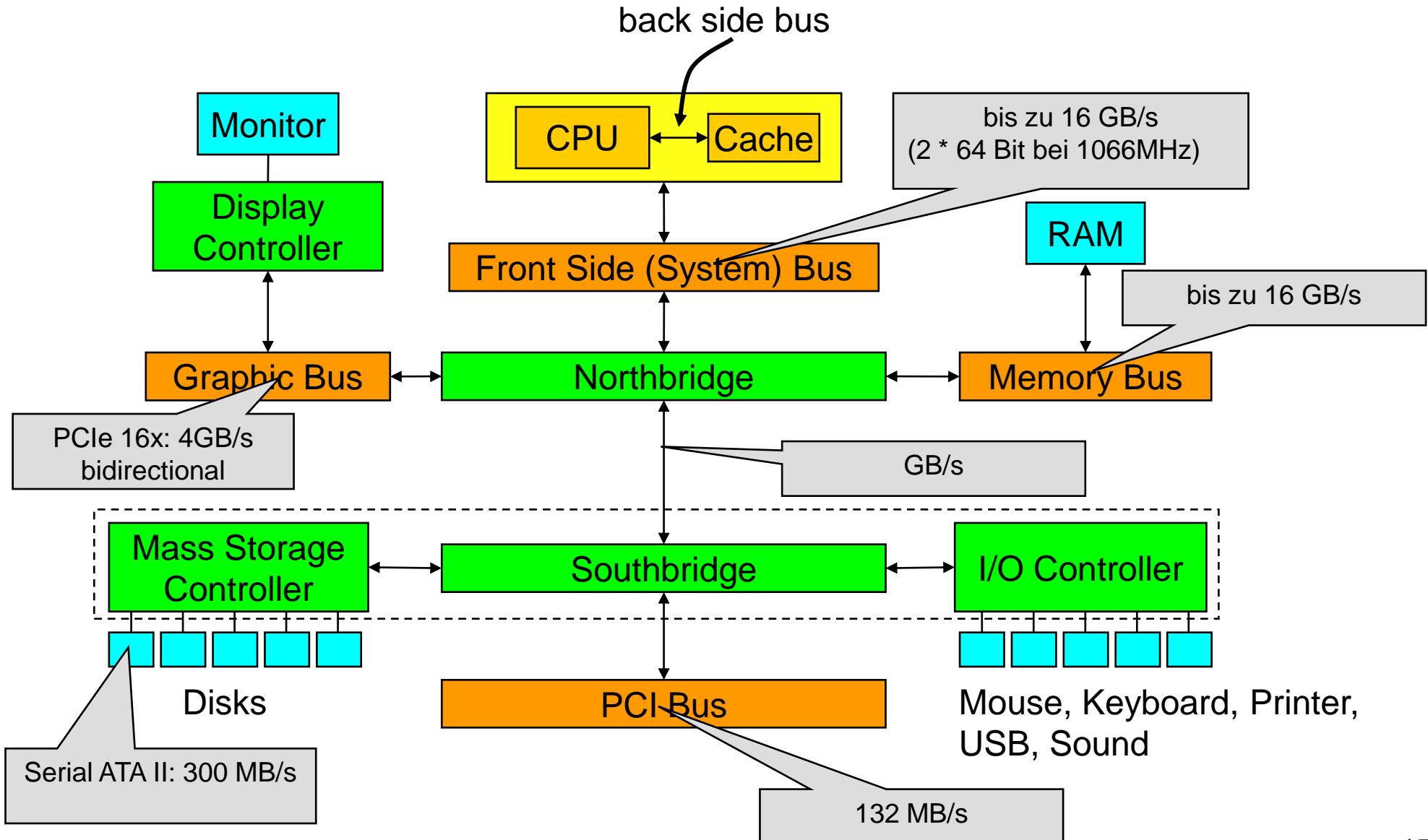
Geräte

Geräte, typische Datenraten (etwa im Jahre 2000)

Device Controllers

- I/O Geräte bestehen aus
 - einer ausführenden (evtl. mechanischen) Einheit
 - einer elektronischen Komponente
- Elektronische Komponente heißt Device Controller
 - kann evtl. mehrere Geräte verwalten
- Aufgaben des Controllers
 - Konversion serieller Bitstrom in Byte-Blöcke
 - Fehlerkorrektur (wenn nötig)
 - Verfügbarkeit via Hauptspeicher
 - Standardisierte Schnittstelle

Das PC Bussystem (2007)



3.1.2. Zugriff auf Geräte

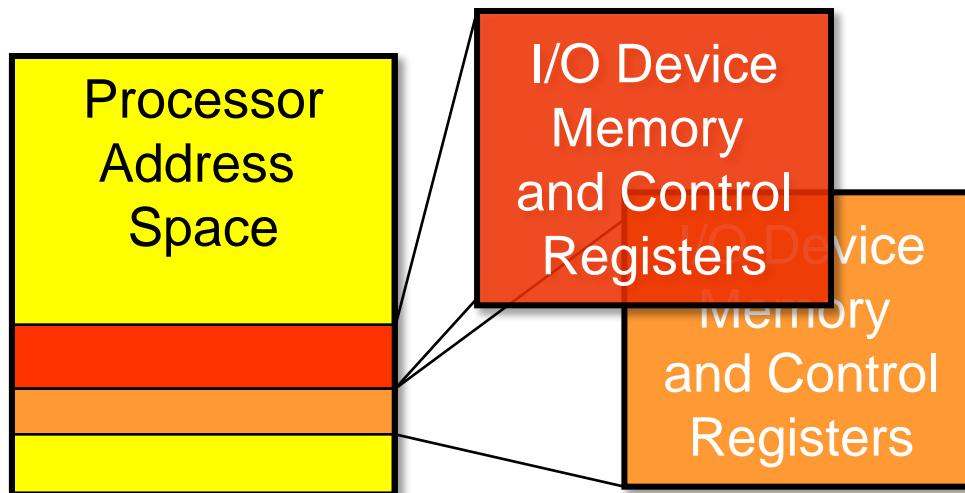
- Manipulation der **Kontrollregister** eines Geräts
 - zur Erteilung von Befehlen und zur Konfiguration
- **Datenpuffer** der Geräte, die das Betriebssystem lesen und schreiben kann
 - z.B. Videospeicher
- Zugriff
 - über spezielle **Port-Befehle** (Zugriff zum *Port-Memory*)
 - z.B. IN Reg, PortNo ; OUT Reg, PortNo
 - oder **Memory Mapped I/O**
- Methoden
 - **Programmgesteuerte Ein-/Ausgabe** (busy waiting*) vs.
 - Ein-/Ausgabe mit **Interrupts**** und **DMA** Betrieb

* geschäftiges Warten

** Unterbrechungen

Memory Mapped I/O

- Zu jedem I/O Gerät gehört ein definierter Abschnitt im Adressraum des Hauptspeichers
 - Konfigurations- und Zustandsregister
 - Datenregister
- Konfigurierung, Zustandsabfragen und Datentransferoperationen werden einheitlich als Speicherzugriffe modelliert



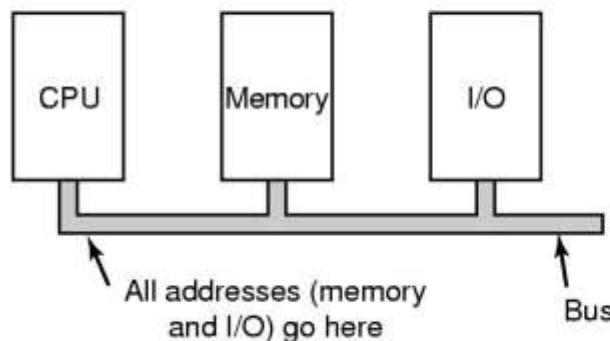
Portaddressierung vs. Memory-Mapped I/O

Portaddressierung

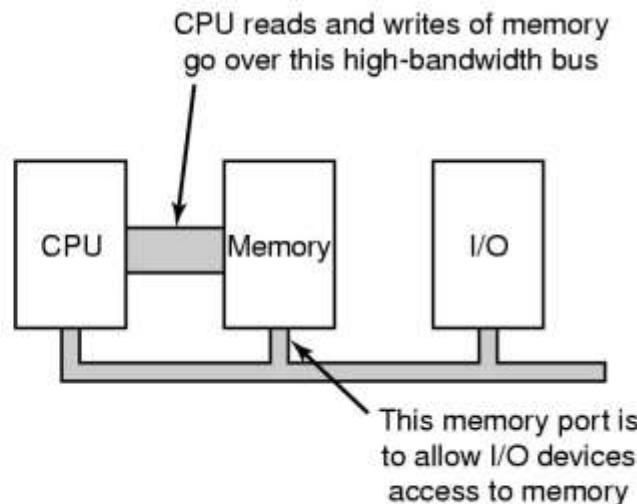
- Separate Befehle in Assembler
- Schutz von IN/OUT im Kernel-Mode
- Kein selektiver Schutz möglich
- Kein Caching-Problem
- Einfacher bei Systemen mit getrenntem Bus

Memory Mapped I/O

- Direkt verwendbar in Hochsprache
- Schutzmechanismus durch virtuellen Speicher
- Caching wäre fatal
- Komplizierter bei Systemen mit getrenntem Bus



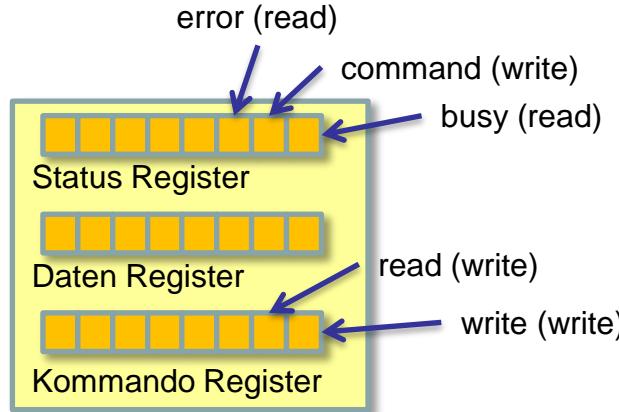
Konzept mit einem Bus



Dual-Bus Architektur

Tanenbaum: Moderne
Betriebssysteme,

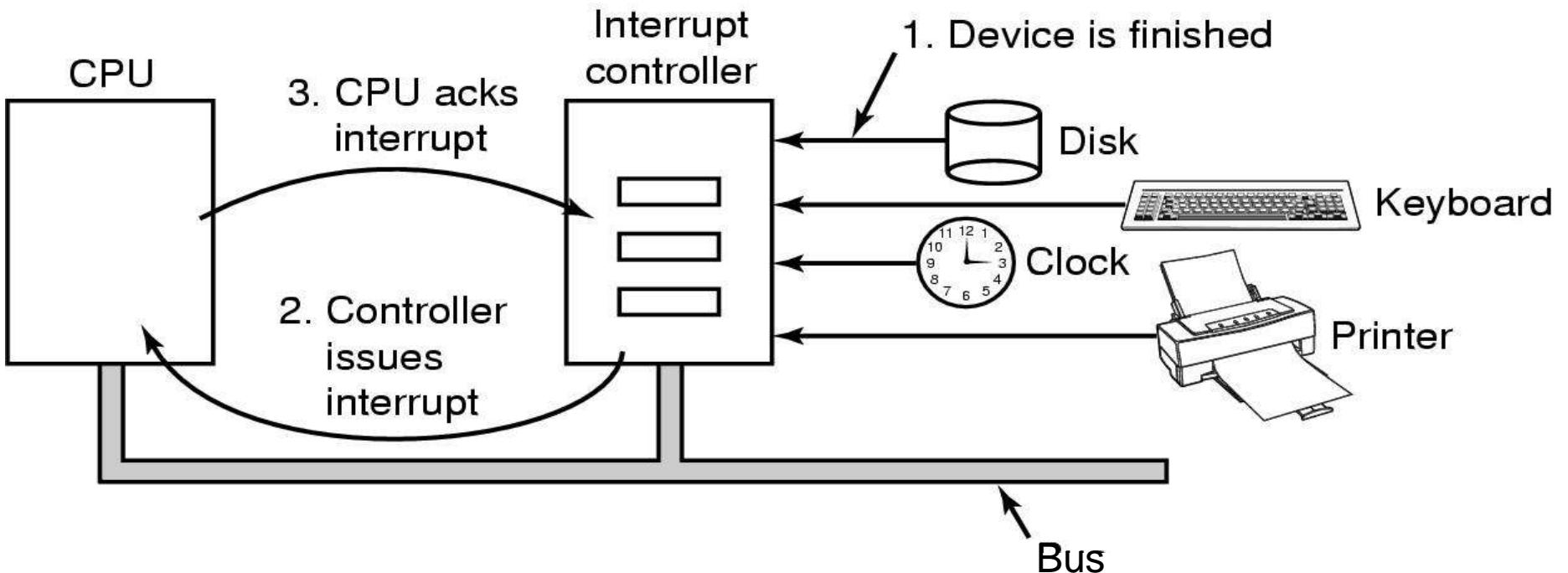
Programmgesteuerte Ein-/Ausgabe (Polling)



- **Beispiel:** Gerät mit drei Registern: Status, Daten und Kommando
- Ablauf Polling
 - Host liest Status bit *busy* bis frei
 - Host schreibt Daten in Daten-Register und setzt *write* bit im Kommando-Register
 - Host setzt *command* im Status-register
 - Gerät sieht *command* und setzt *busy* im Status register
 - Gerät schreibt Daten
 - Gerät setzt Status-bits *command* und *busy* zurück

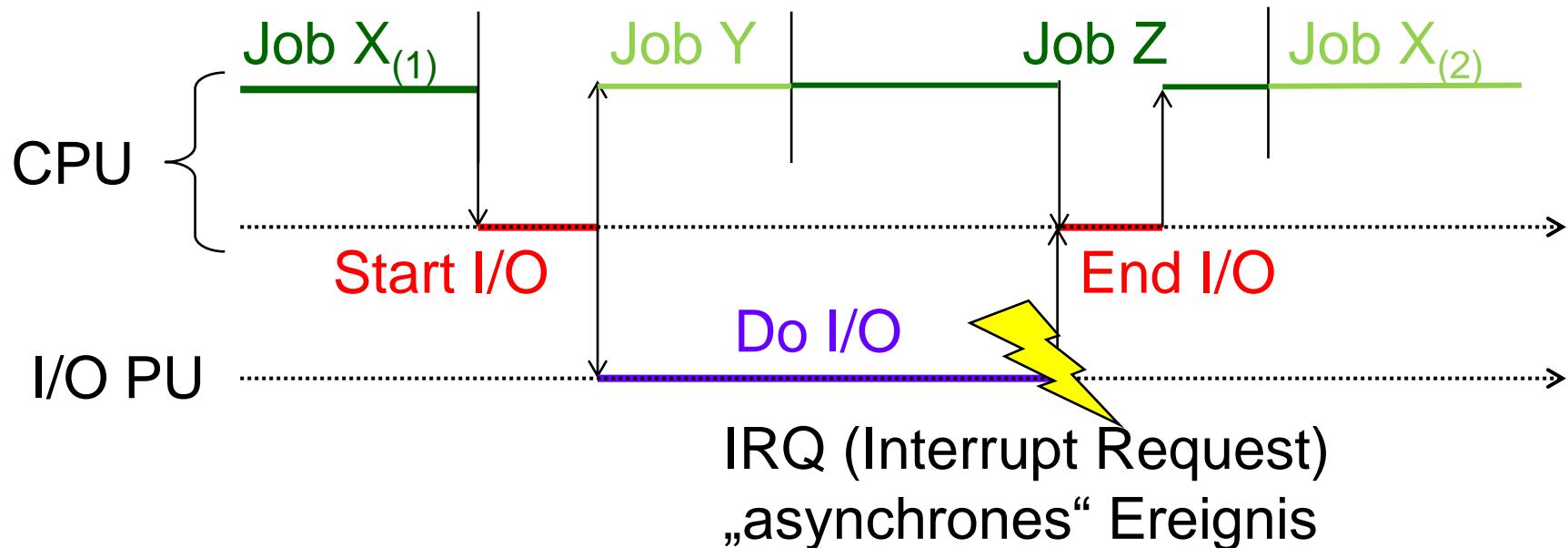
3.1.3. Das Interruptkonzept

- Interrupts unterbrechen die normale sequentielle Befehlsfolge des Prozessors
- Die meisten I/O Geräte sind langsamer als der Prozessor
 - Ohne Interrupts müsste der Prozessor angehalten werden, um auf ein Gerät zu warten („busy waiting“).

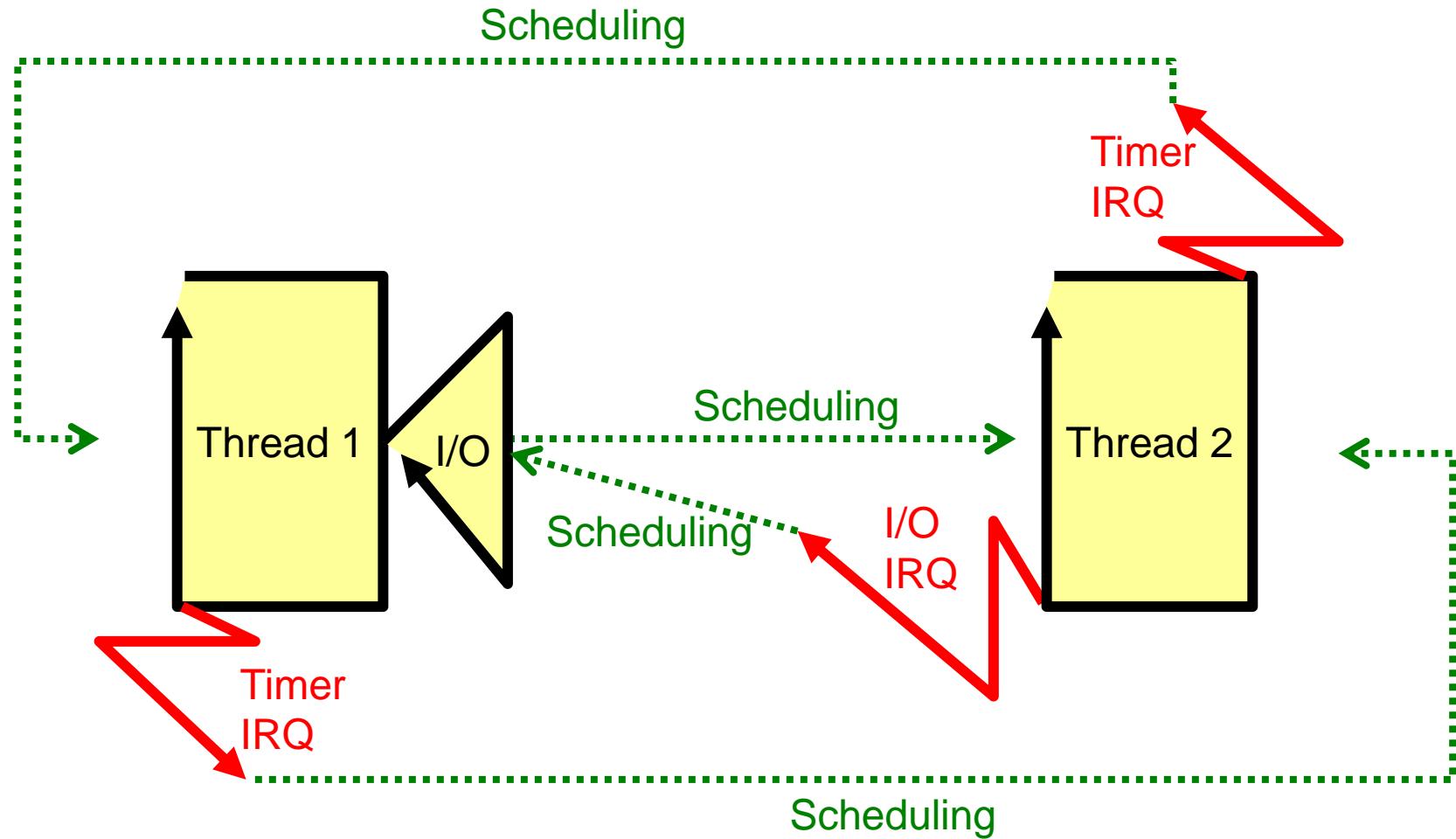


Interruptkonzept: Anwendung

- Delegierung von I/O Aufträgen an I/O Prozessoren
 - Inhärentes Parallelisierungspotenzial
 - Unterstützt durch asynchrone Hardware Interrupts

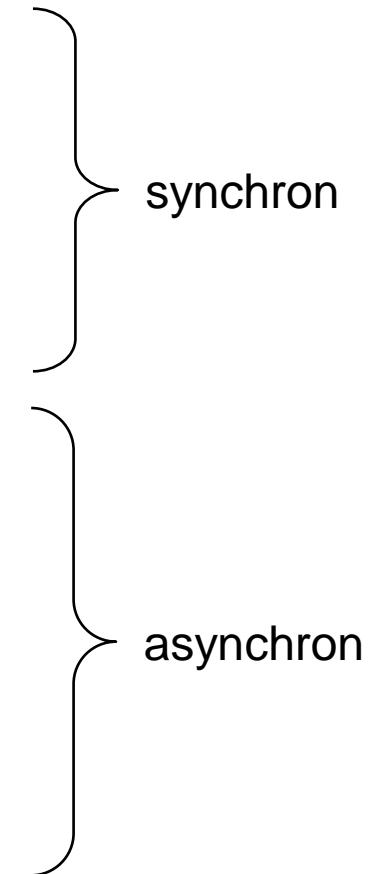


Ablaufbeispiel I/O-IRQ und Prozesswechsel



Interrupt Klassen

- Programm (Software Interrupt)
 - Erzeugt durch Bedingung in Befehlsausführung
- Fehlsituation (Exception)
 - Rechenoperation, ungültiger Befehl, Adressfehler ..
- Zeitgeber (Timer)
 - Erzeugt durch Zeitgeber innerhalb des Prozessors
- I/O
 - Erzeugt durch I/O Steuereinheit
- Hardwareausfall
 - Erzeugt durch Defekt



Die meisten Interrupts können
maskiert, also deaktiviert oder
verzögert werden

oft „unmaskable“

Interrupts

- Unterbrechung der normalen Ausführungssequenz

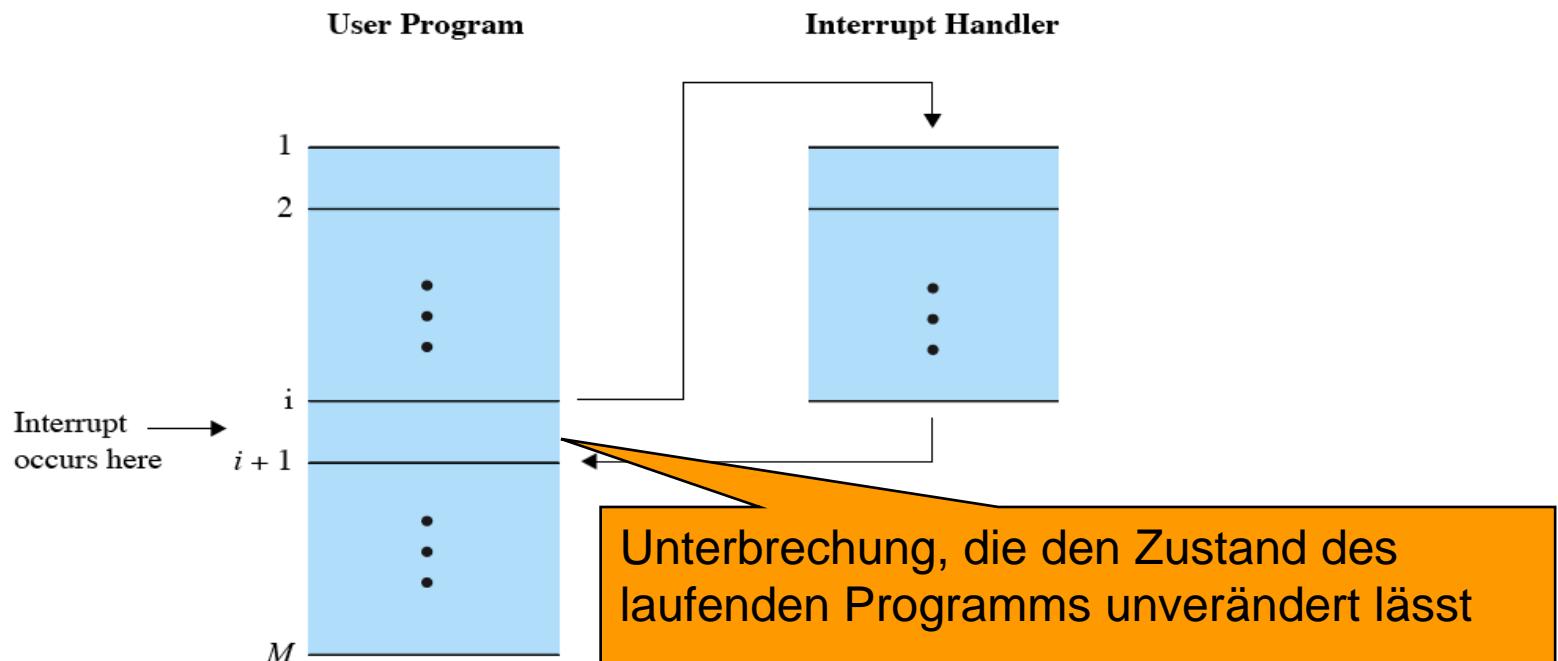


Figure 1.6 Transfer of Control via Interrupts

Interrupt Zyklus im Prozessor

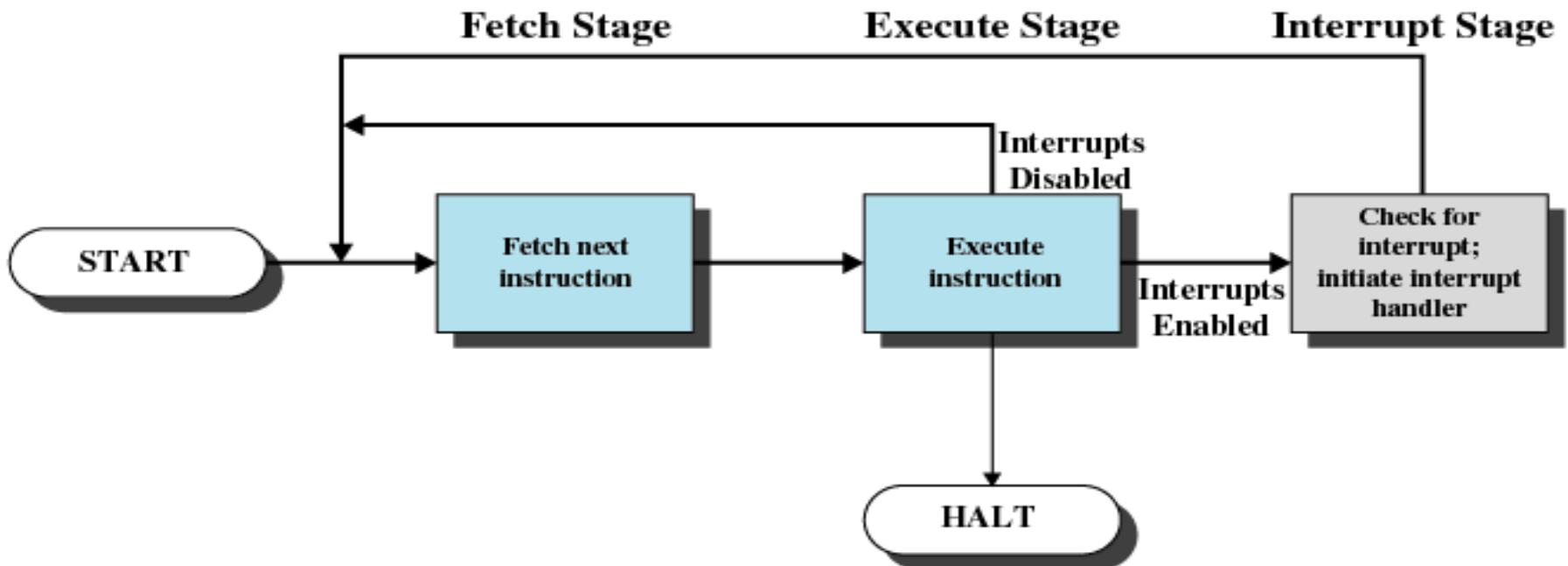


Figure 1.7 Instruction Cycle with Interrupts

Begriff des Prozessorzustandes

Prozessorzustand

- Processor Status Wort (PSW / PSR)
 - Processor Modus
 - Interrupt Masken
 - Condition Codes
- Spezielle Register
 - Programmzähler (PC / IP)
 - Stack Pointer (SP)
- Generelle Register
 - Integer Werte
 - Adressen (Pointer), Framepointer (FP)
- Weitere Control Register
 - Einstellungen zu Paging, Caching, Prozessor Modus ...
- Floating Point Register
- MMX, SIMD, Performance Monitoring ...

Interrupt-Verarbeitung

Hardware

- Gerät-controller oder andere Systemhardware löst Interrupt IRQ #n aus
- Prozessor beendet Ausführung der aktuellen Instruktion
- Prozessor signalisiert Bereitschaft für Interrupt
- Prozessor schreibt PSW und PC auf Stack oder in Spezialregister (architekturabhängig)
- Prozessor lädt Wert des Programmzählers, abhängig von der Interrupt Nummer n

Software

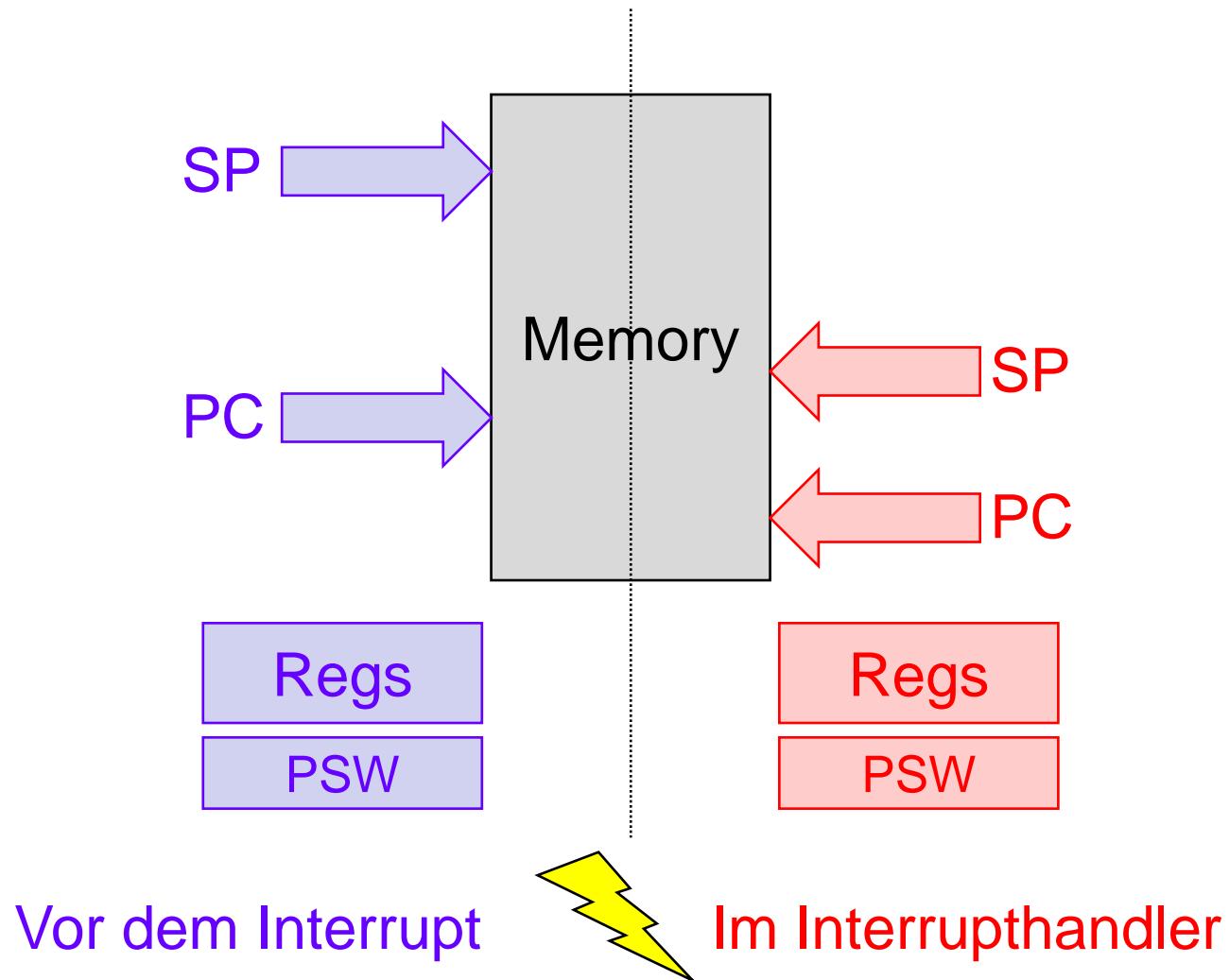
- *(First Level) Interrupt Handling*
Speichern des (nötigen) Rests der Prozessstatusinformationen
- Verarbeitung des Interrupts
(Möglicherweise als eigener Prozess => Second Level Interrupt Handler / DPC*)
- Wiederherstellen der Prozessstatusinformation
- Wiederherstellung des alten Prozessstatusworts und des Programmzählers

Interrupt Vector

#2	IRQ PC(2)
#1	IRQ PC(1)
#0	IRQ PC(0)

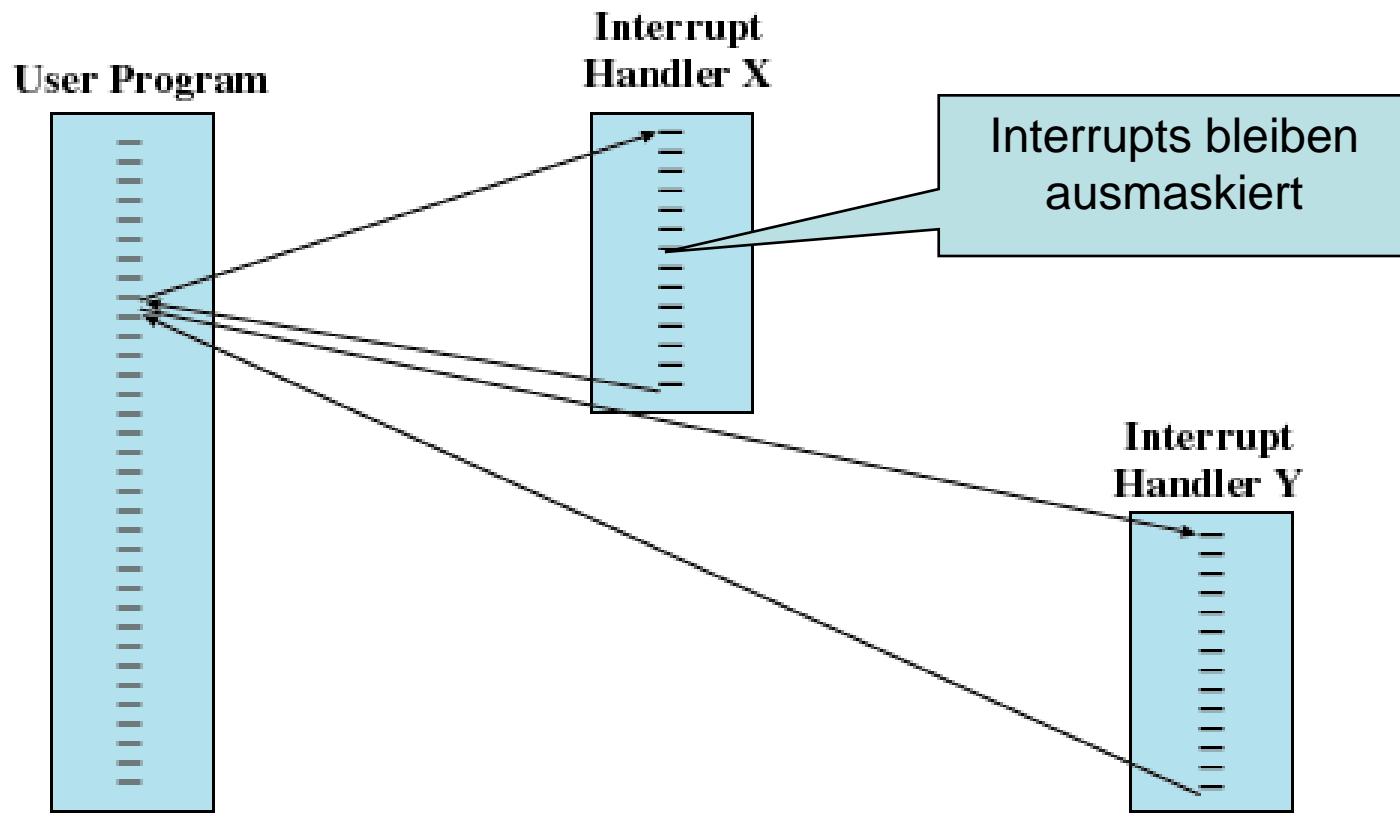
Interrupt Descriptor Table (IDT)

*Deferred Procedure Call



Mehrfach-Interrupts

- Strategie 1: Sequentielle Interrupt-Verarbeitung



(a) Sequential interrupt processing

Verschachtelte Interrupts

- Strategie 2: Prioritätsbasierte Zuteilung

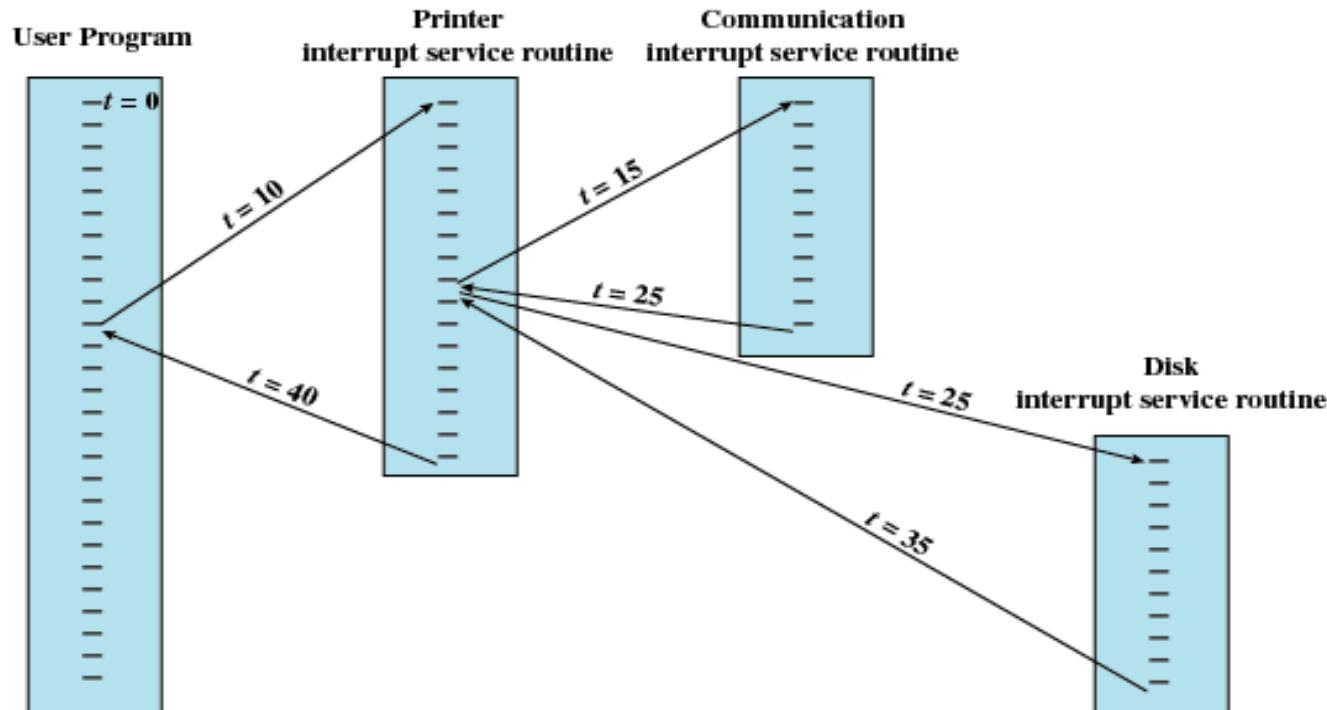
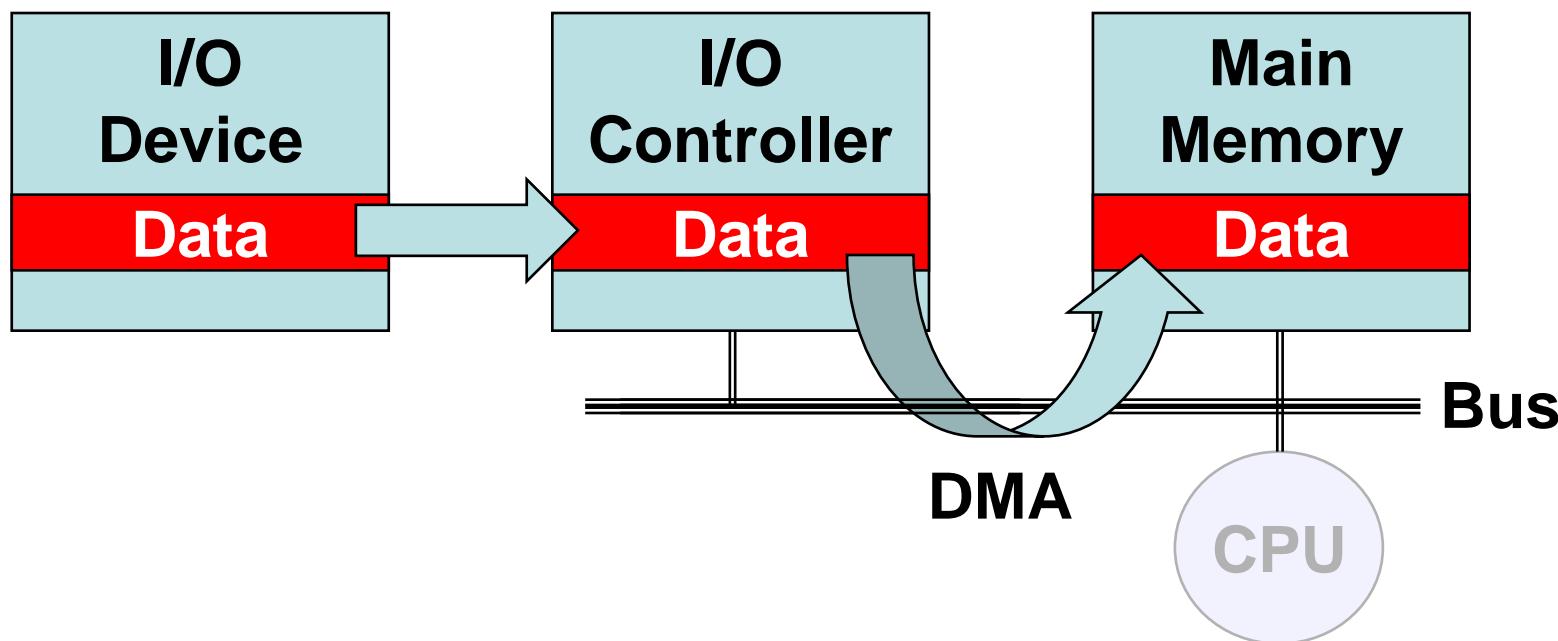


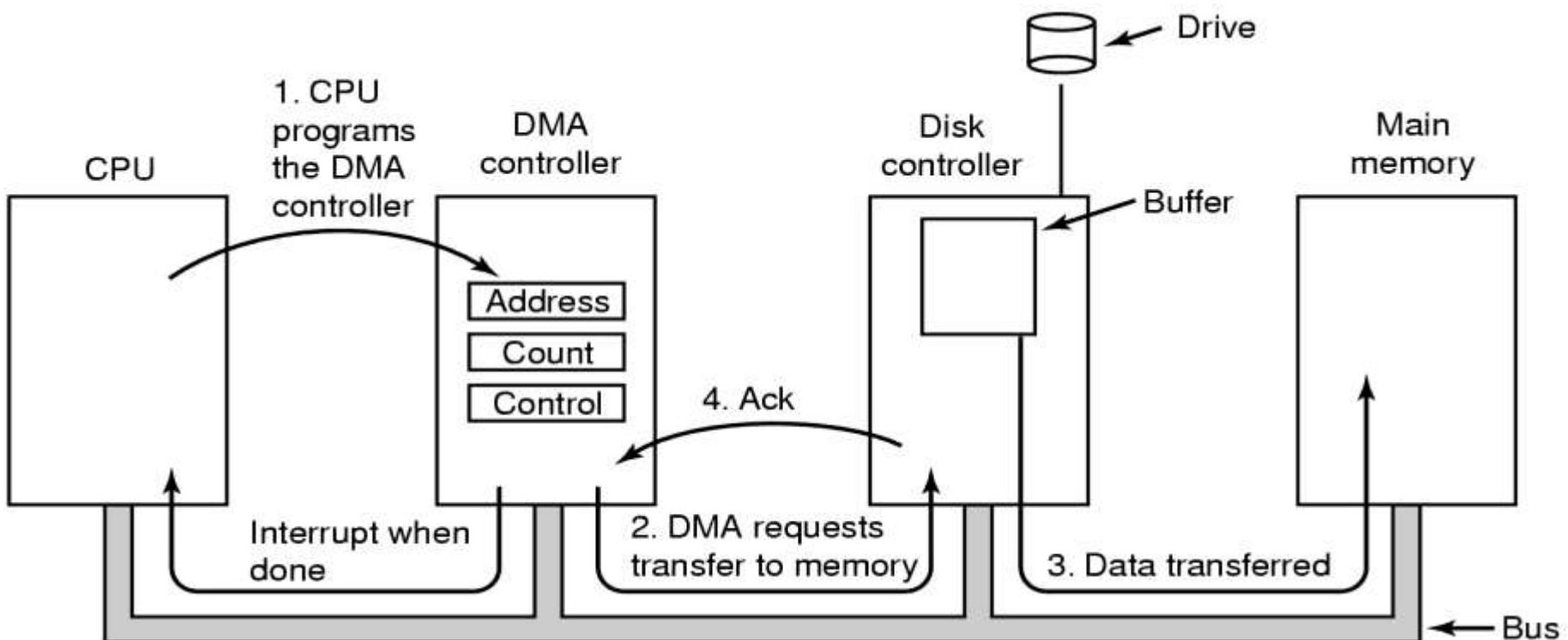
Figure 1.13 Example Time Sequence of Multiple Interrupts

3.1.4. Direct Memory Access (DMA)

- Möglichkeit Daten ohne Intervention eines Hauptprozessors vom I/O Gerät über den Bus in den Hauptspeicher zu transferieren



Beispiel: Lesen von Disk

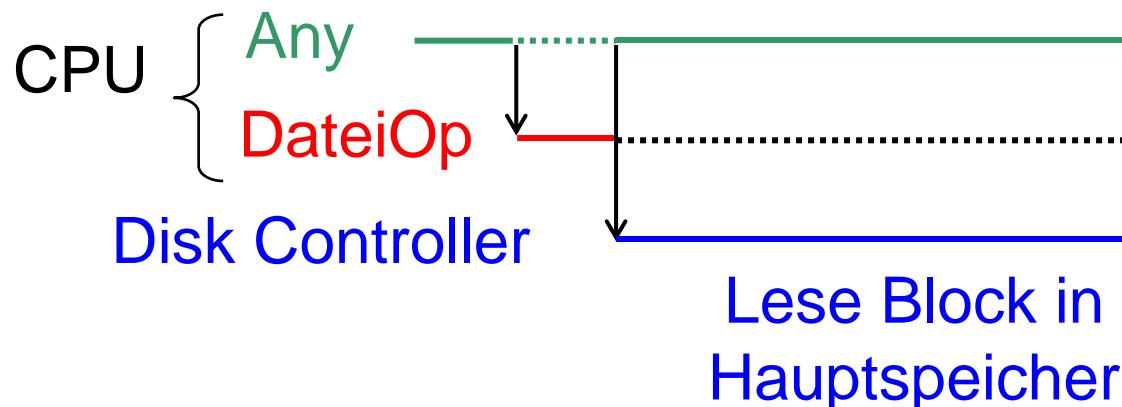


Blocktransfer

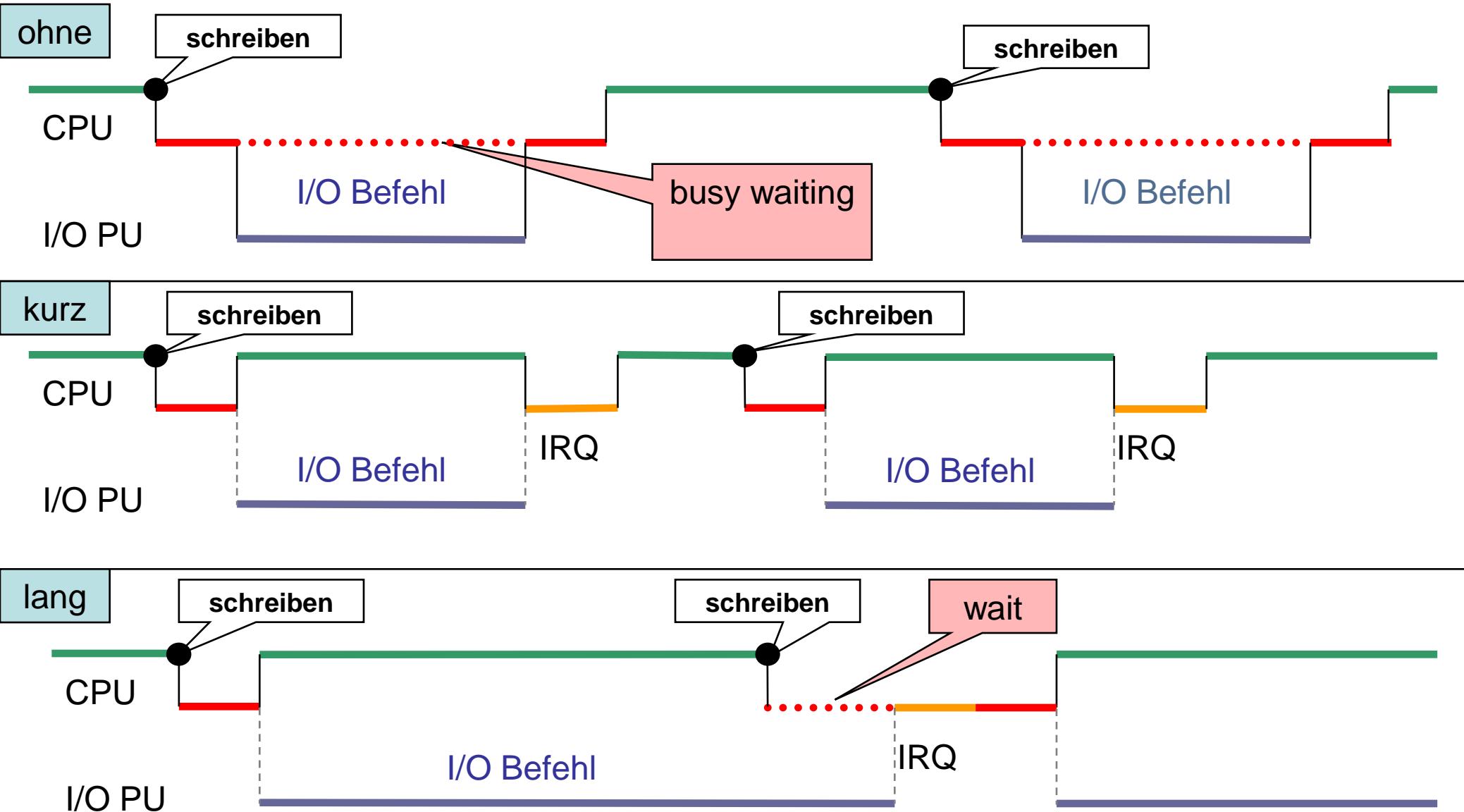
- Ablauf ohne DMA



- Ablauf mit DMA

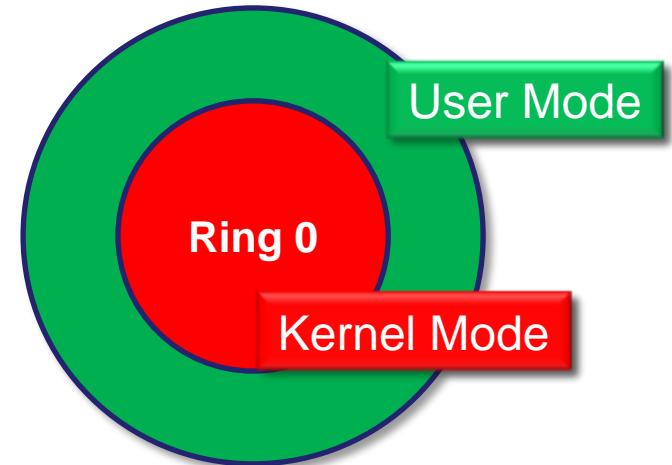


Programmfluss mit und ohne Interrupts



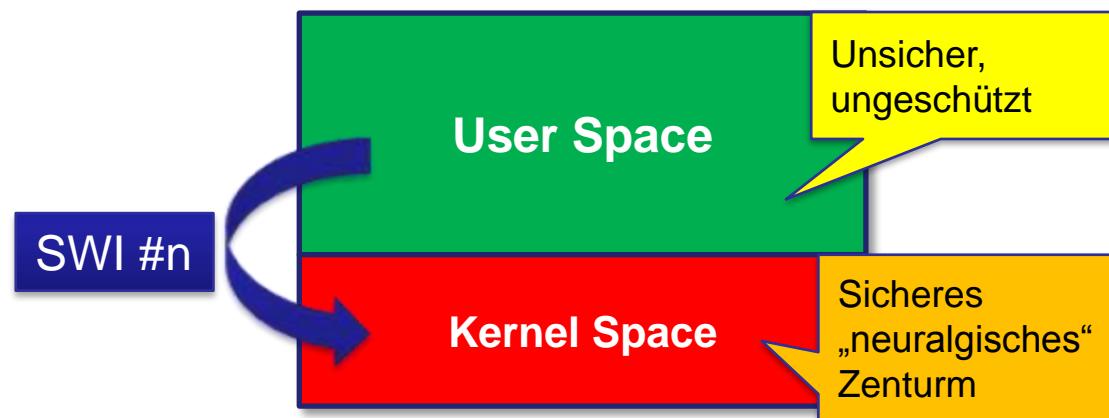
3.1.5. Kernel Modus vs. User Modus

- Kernel Mode (im PSW)
 - Privilegiert
 - Alle Instruktionen erlaubt
 - Zugriff zu allen Speicheradressen erlaubt
- User Mode (im PSW)
 - Eingeschränkter Instruktionssatz
 - Load PSW nicht erlaubt
 - Eingeschränkter Zugriff im Hauptspeicher
 - Systemkernbereich nicht zugreifbar
 - Realisiert durch Paging, Page Table im User-Mode nicht veränderbar



Software Interrupt (Supervisor Call)

- Durch Maschineninstruktion eingeleiteter Interruptvorgang
 - „Synchroner“ Interrupt
 - SWI #n führt zum PSW Austausch entsprechend Interrupttabelle #n wie im asynchronen Fall
- Führt vom „User Space“ in den „Kernel Space“

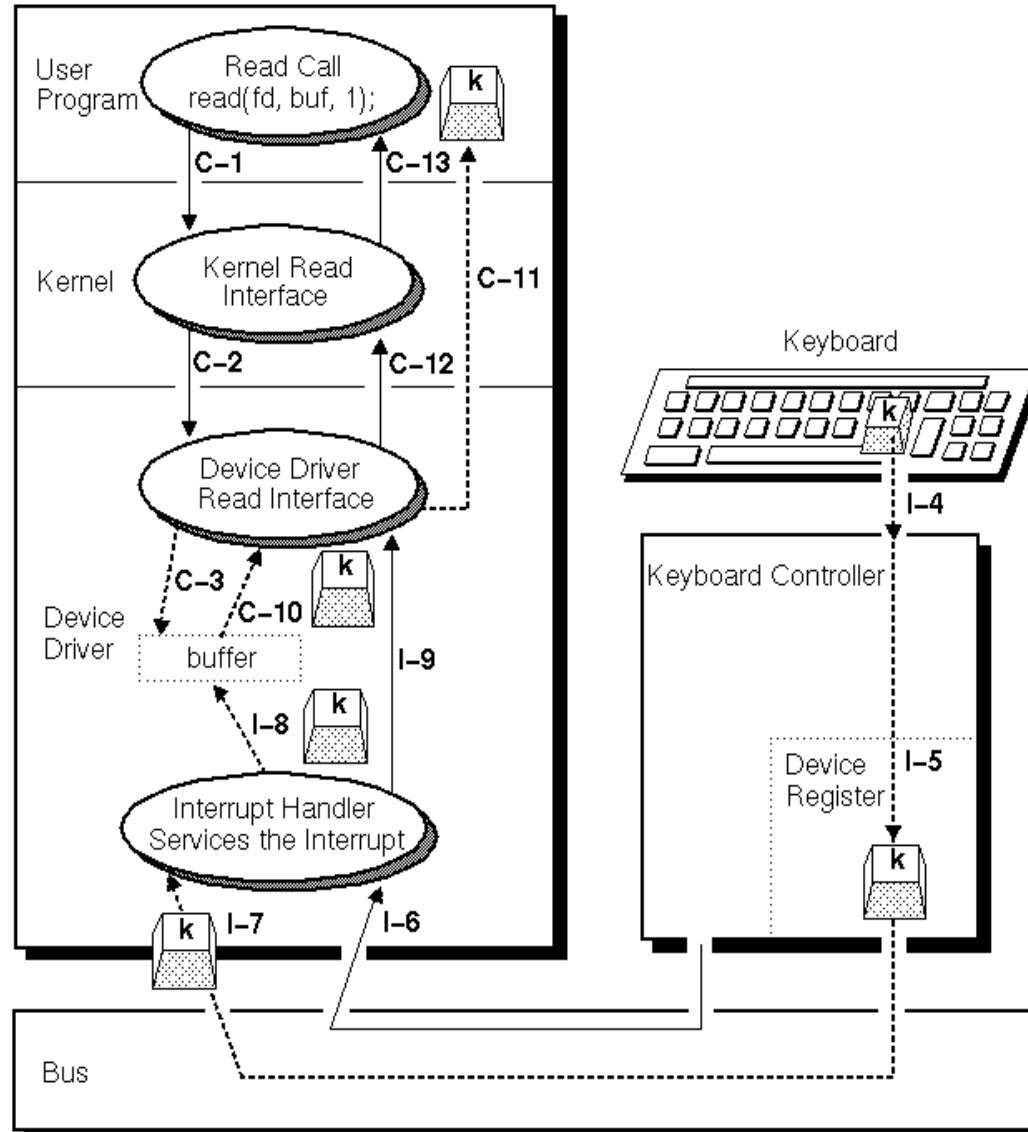


3.2. Driver Beispiele

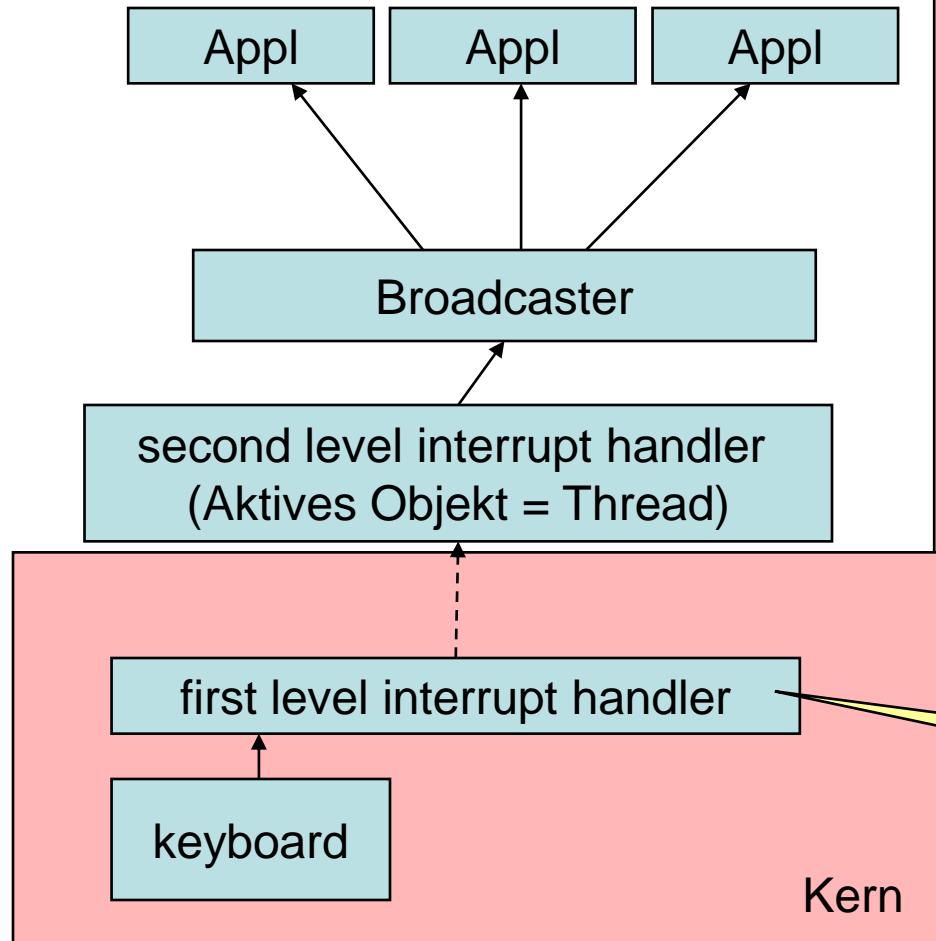
- 3.2.1. Keyboard Driver
- 3.2.2. Disk Driver



3.2.1. Keyboard Driver – Buffer Modell



Beispiel: Situierung eines Keyboard Drivers im A2-System, asynchrones Modell



```

PROCEDURE FieldInterrupt;
CODE {SYSTEM.i386} (* 3 bytes implicit code skipped: PUSH EBP; MOV EBP, ESP *)
    PUSHAD      ; save all registers (EBP = error code)
    ...
    MOV EBX, 32[ESP]
    LEA EAX, intHandler ; interrupt procedures
    MOV EAX, [EAX][EBX*4]

loop:           ; call all handlers for the interrupt
    ...
    CALL DWORD 4[EAX]          ; call handler
    CLI                      ; handler may have re-enabled interrupts
    ...
    JNE loop
    POPAD                   ; now EBP = error code
    POP EBP                 ; now EBP = INT
    POP EBP                 ; now EBP = caller EBP
    IRET
END FieldInterrupt;

```

EBX = int number
mittels Glue code
sichergestellt

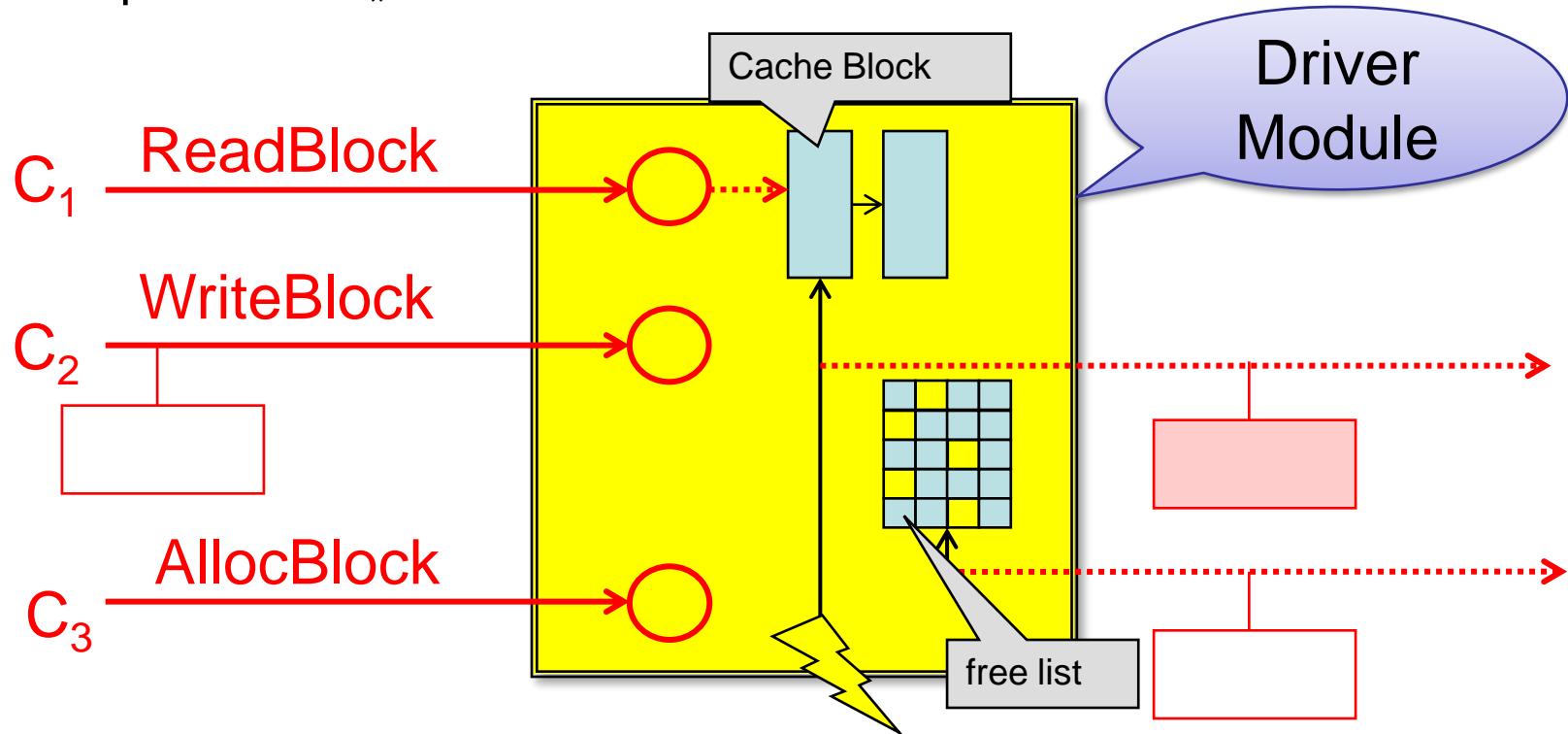
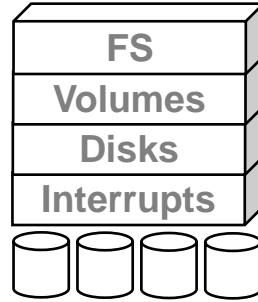
$p := \text{interrupt}[iNr].process;$
 $\text{Enter}(p);$ (* Eintrag in die Ready-Queue *)

3.2.2. Disk Driver

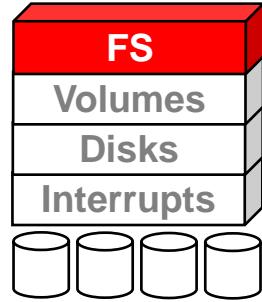
- Aufgaben
 - Implementation des Disksystem API
 - Allokation von Diskblöcken
 - Blockorientiertes Lesen und Schreiben
 - Verwaltung der Blockbelegungstabelle
 - Verwaltung aller aktuellen Lese- und Schreibaufträge
 - Schedulieren neuer Aufträge (z. B. Elevatoralgorithmus)
 - Erkennen der Terminierung von Operationen

Disk Driver als Monitor-Modul

- Gekapselte Datenstruktur plus Operationen
 - Implizite Initialisierung beim Laden
- „Shared Resource“ für alle Systemprozesse
 - Forcierung von „gegenseitigem Ausschluss“ zum Schutz gegen potentielle „Race Conditions“*



* „Wettlaufsituation“



Beispiel: Diskmanagement API

Abstract Filesystem Level

MODULE Files;

Volume = OBJECT

PROCEDURE AllocBlock (hint: LONGINT; VAR adr: LONGINT);

...

PROCEDURE ReadBlock (adr: LONGINT; VAR blk: BYTES);

PROCEDURE WriteBlock (adr: LONGINT; VAR blk: BYTES);

END;

```

PROCEDURE AllocBlock*(hint: Address; VAR adr: Address);
BEGIN {EXCLUSIVE}
... (* Fehlerbehandlung *)
adr := hint+1;
LOOP

...
IF (adr MOD 32) IN map[adr DIV 32] THEN
  INC(adr) (* Block in use *)
ELSE
  INCL(map[adr DIV 32], adr MOD 32); EXIT
END;

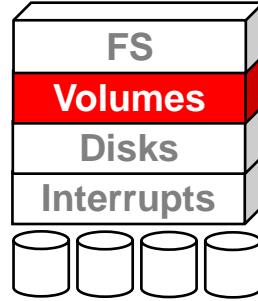
...
END;

...
END AllocBlock;

```

Beispiel: Diskmanagement API

Filesystem Implementation Level (Logical Device Driver)



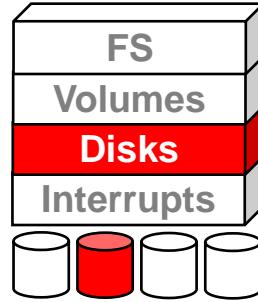
MODULE DiskVolumes

Volume = OBJECT (AosFS.Volume)

```
PROCEDURE AosDiskRead(adr: LONGINT; VAR adr: LONGINT);
PROCEDURE ReadBlock*(adr: LONGINT; VAR blk: ARRAY OF CHAR);
VAR res, block: LONGINT; buf: AosCaches.Buffer; valid: BOOLEAN;
BEGIN {EXCLUSIVE}
  ... (* Fehlerbehandlung *)
  block := startfs + (adr-1) * blocks;
  IF cache # NIL THEN
    cache.Acquire(dev, block, buf, valid);
    IF ~valid THEN dev.Transfer(AosDisks.Read, block, blocks, buf.data^, 0, res)
    ELSE res := AosDisks.Ok
  END;
  SYSTEMMOVE(SYSTEM.ADR(buf.data[0]), SYSTEM.ADR(blk[0]), blockSize);
  cache.Release(buf, FALSE, FALSE)
ELSE
  dev.Transfer(AosDisks.Read, block, blocks, blk, 0, res)
END;
...
END GetBlock;
```

Beispiel: Diskmanagement API

Device Driver Level (Physical Device Driver)



MODULE ATADisks;

Device* = OBJECT (Disks.Device)

...

PROCEDURE Transfer*(op, block, num: LONGINT; VAR data: ARRAY OF CHAR; ofs: LONGINT; VAR res: LONGINT);

...

END Device;

BEGIN{EXCLUSIVE}

...

(* SETUP DMA: Übertrage Adresse des Buffers *)

SYSTEM.PORTOUT(bmbase + Ofs_BMPRDT, command.prdtPhysAdr);

(* Setze Richtungsflag *)

IF command.read THEN ch := CHR(ASH(1, DMA_Read)); ELSE ch := 0X; END;

SYSTEM.PORTOUT(bmbase + Ofs_BMCmd, ch);

...

(* START DMA: *)

SYSTEM.PORTOUT(bmbase + Ofs_BMCmd, CHR(SYSTEM.VAL(LONGINT, s)));

....

REPEAT

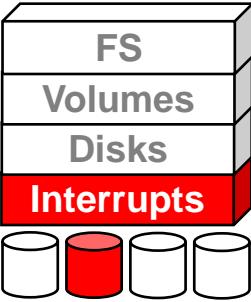
IF ~interrupt.Wait(ms) THEN ... END;

SYSTEM.PORTIN(bmbase + Ofs_BMStatus, ch); s := SYSTEM.VAL(SET, LONG(ch));

UNTIL ~(DMA_Busy IN s) OR AosKernel.Expired(t);

...

(* STOP DMA *) ...



Implementation des Interrupt Objekts

Interrupt = OBJECT

```
VAR int: LONGINT; interrupt, timeout: BOOLEAN; clock: Objects.Timer;
```

```
PROCEDURE HandleInterrupt;
BEGIN {EXCLUSIVE} interrupt := TRUE; END HandleInterrupt;
```

```
PROCEDURE HandleTimeout;
BEGIN {EXCLUSIVE} timeout := TRUE; END HandleTimeout;
```

```
PROCEDURE Wait(ms: LONGINT): BOOLEAN;
BEGIN {EXCLUSIVE}
    timeout := FALSE; Objects.SetTimeout(clock, SELF.HandleTimeout, ms); (* set or reset timeout *)
    AWAIT(interrupt OR timeout);
    Objects.CancelTimeout(clock); interrupt := FALSE;
    RETURN ~timeout
END Wait;
```

```
PROCEDURE &Init(irq: LONGINT);
BEGIN
    interrupt := FALSE; int := Machine.IRQ0 + irq; NEW(clock);
    Objects.InstallHandler(SELF.HandleInterrupt, int)
END Init;
```

```
END Interrupt;
```

3.3. Generische Treiberstruktur Ziele und Konzepte

- Ziele
 - Geräteneunabhängigkeit
 - Einheitliche Identifikation
- Konzepte
 - Synchroner vs. asynchroner Transport
 - Gemeinsame vs. exklusive Zuteilung
 - Buffer

Synchrone / Asynchrone Ein-/Ausgabe

- Blockierend: Prozess steht still bis I/O abgeschlossen ist
 - einfach zu benutzen und zu verstehen
 - offensichtliche Nachteile
- Nicht blockierend: I/O liefert derzeit vorhandene Information zurück
 - gibt sofort die Anzahl gelesener Bytes zurück (möglicherweise 0)
 - Häufig eingesetzt für einfache Netzwerkdienste
- Asynchron: Prozess läuft weiter während I/O stattfindet
 - I/O Subsystem signalisiert dem Prozess, wenn I/O Auftrag abgeschlossen ist.
 - sehr flexibel, kompliziert zu verstehen und zu benutzen

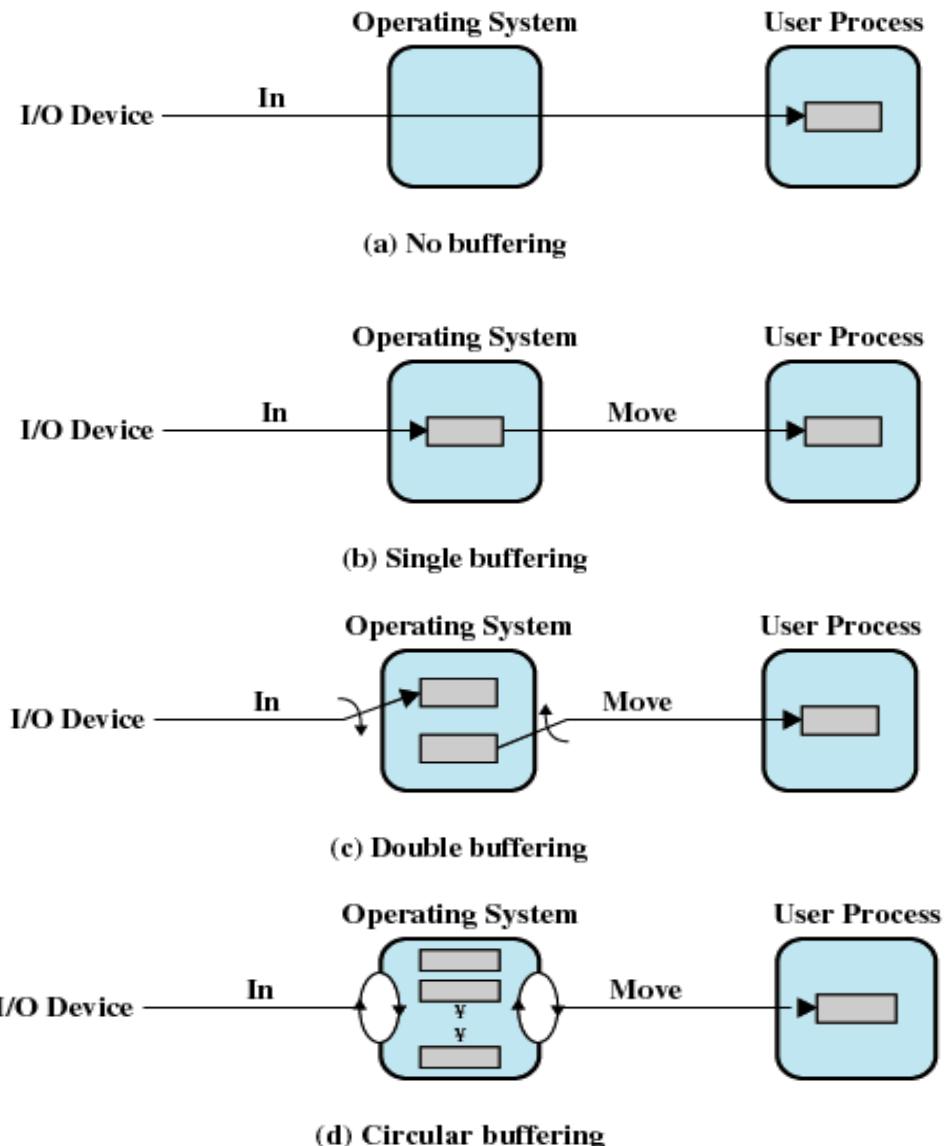
Buffering

Prozess blockiert / suspendiert während I/O Operation im OS

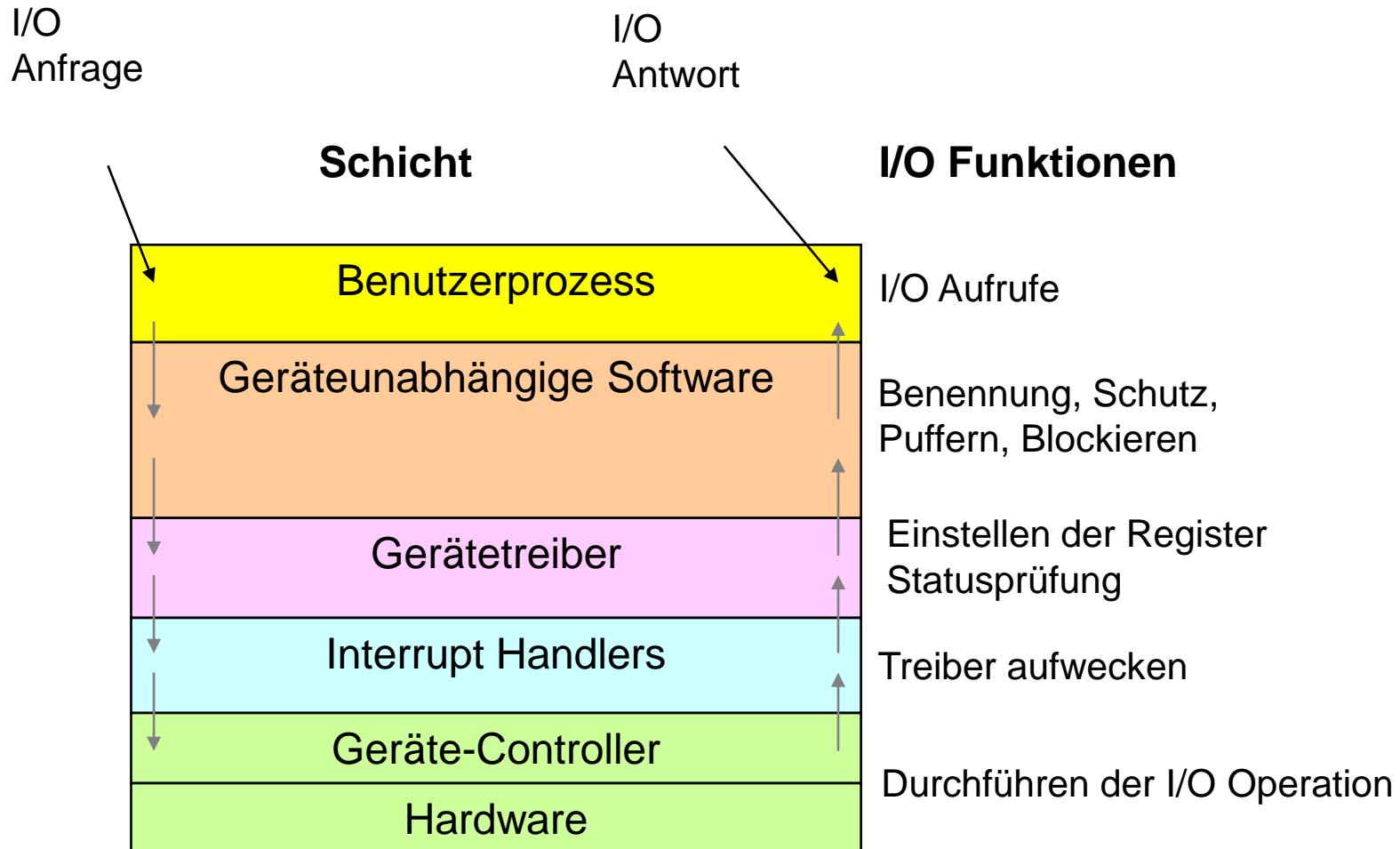
OS kann Puffer füllen, bevor dieser vom Prozess benötigt / ausgelesen wird

Prozess kann Puffer benutzen während das OS den anderen füllt oder leert

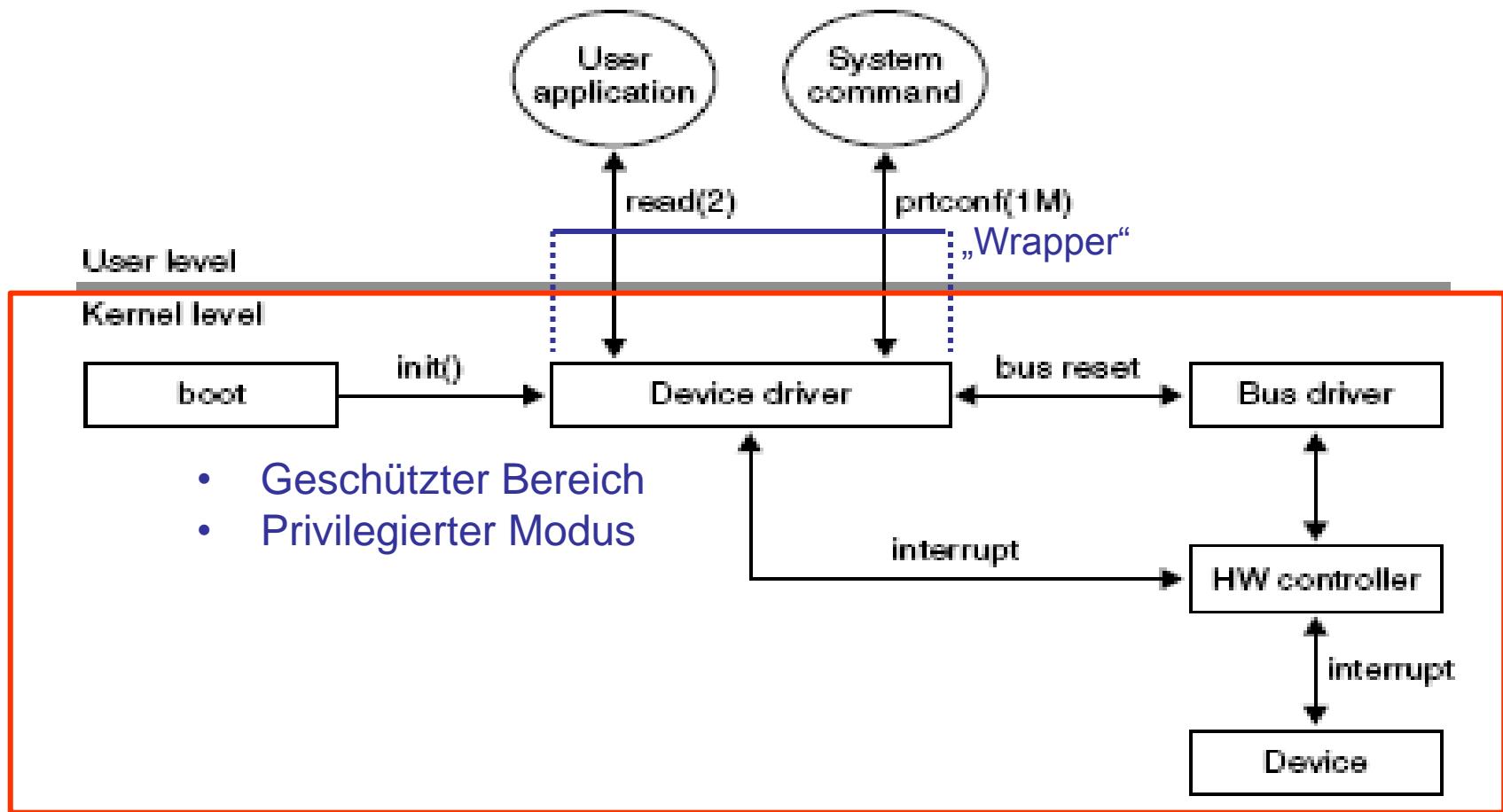
I/O Lese Operation kann / soll mit Prozess mithalten



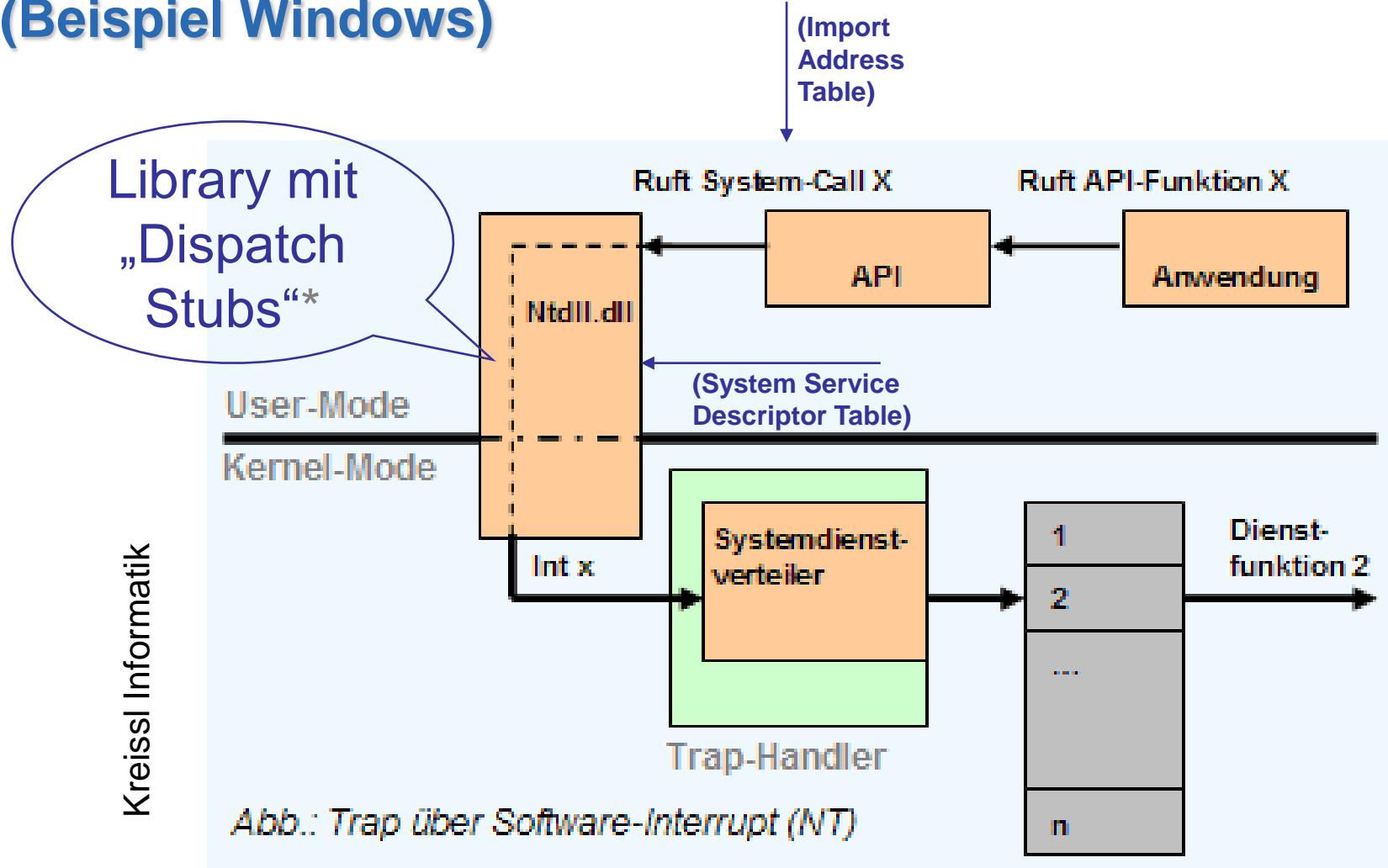
Schichten des I/O Systems



Device Driver Situierung



Kernel Calls via System Library (Beispiel Windows)

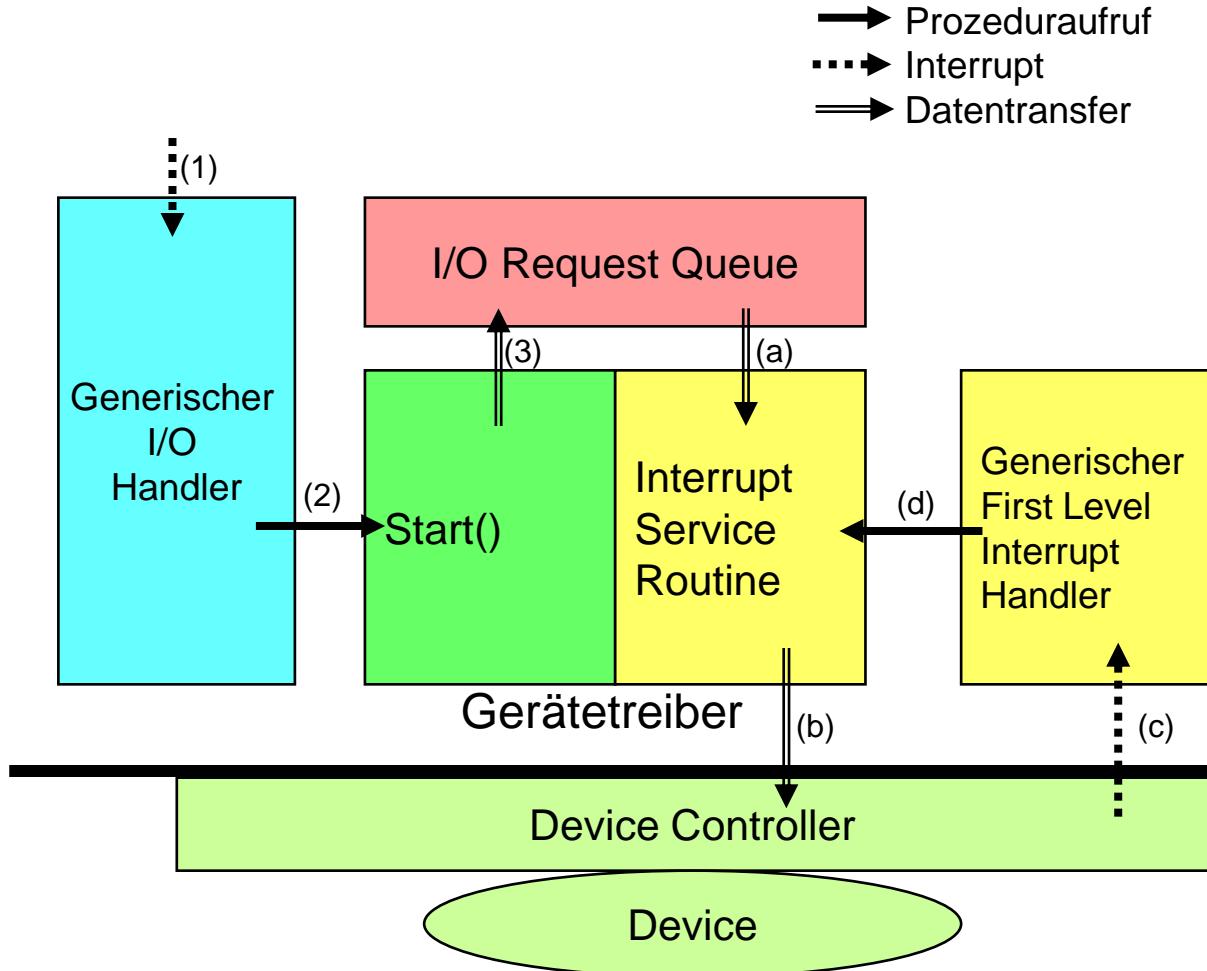


* Dispatch Stub = Ausführungsstumpf

Device Driver Steuerfluss

Beispiel: WriteToDisk

Idee: S. Wilbur, University College London



- (1) System Call zum Schreiben der Daten (via Trap) -> Kernel Mode
 - (2) Device Driver Start Methode, Argument: Buffer
 - (3) Auftrag in Warteschlange
-
- (a) Abholen des nächsten Auftrags
 - (b) Auftrag zum Schreiben der Daten
 - (c) Acknowledgement durch Device Controller (Transport fertig)
 - (d) Aufruf der ISR

Kapitel 4. Hauptspeicherverwaltung

- 4.1. Einprozess Speicherlayout
 - 4.1.1. Stackverwaltung
 - 4.1.2. Heapverwaltung
 - 4.1.3. Codebereichsverwaltung
 - 4.1.4. Laufzeitunterstützung für OOP
 - 4.1.5. Multi Thread Speicherlayout
- 4.2. Mehrprozess Speicherlayout
 - 4.2.1. Segmentierung
 - 4.2.2. Virtuelle Adressierung
 - 4.2.3. Demand Paging

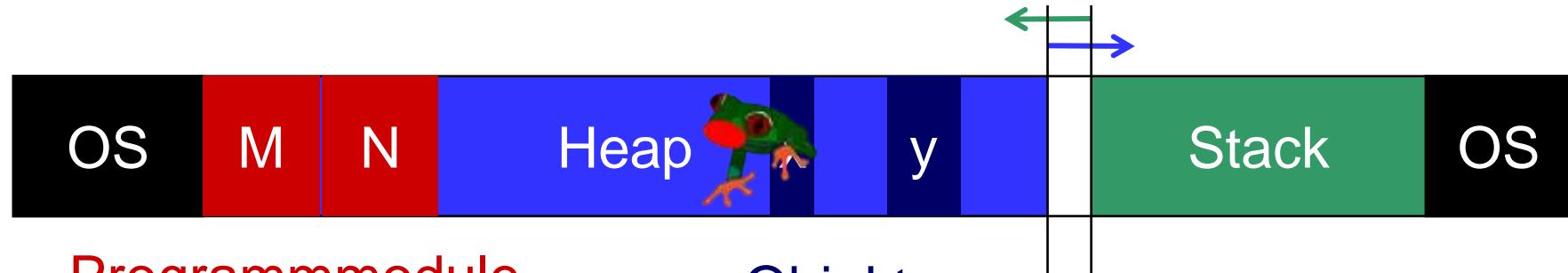
4.1. Einprozess Speicherlayout

- Prozess
 - Programm / Ausführungskontext für Aktivitäten
- Speicherbereiche
 - OS Bereich
 - Heapbereich
 - Programmmodul
 - Ausführbarer Code
 - Globale Daten
 - Dynamisch erzeugte Objekte
 - Stackbereich pro Aktivität („Thread“)
 - Procedure Activation Frame

Klassisches Speicherbild



Fliessende Grenze



Programmmodule
(Code und globale Daten)

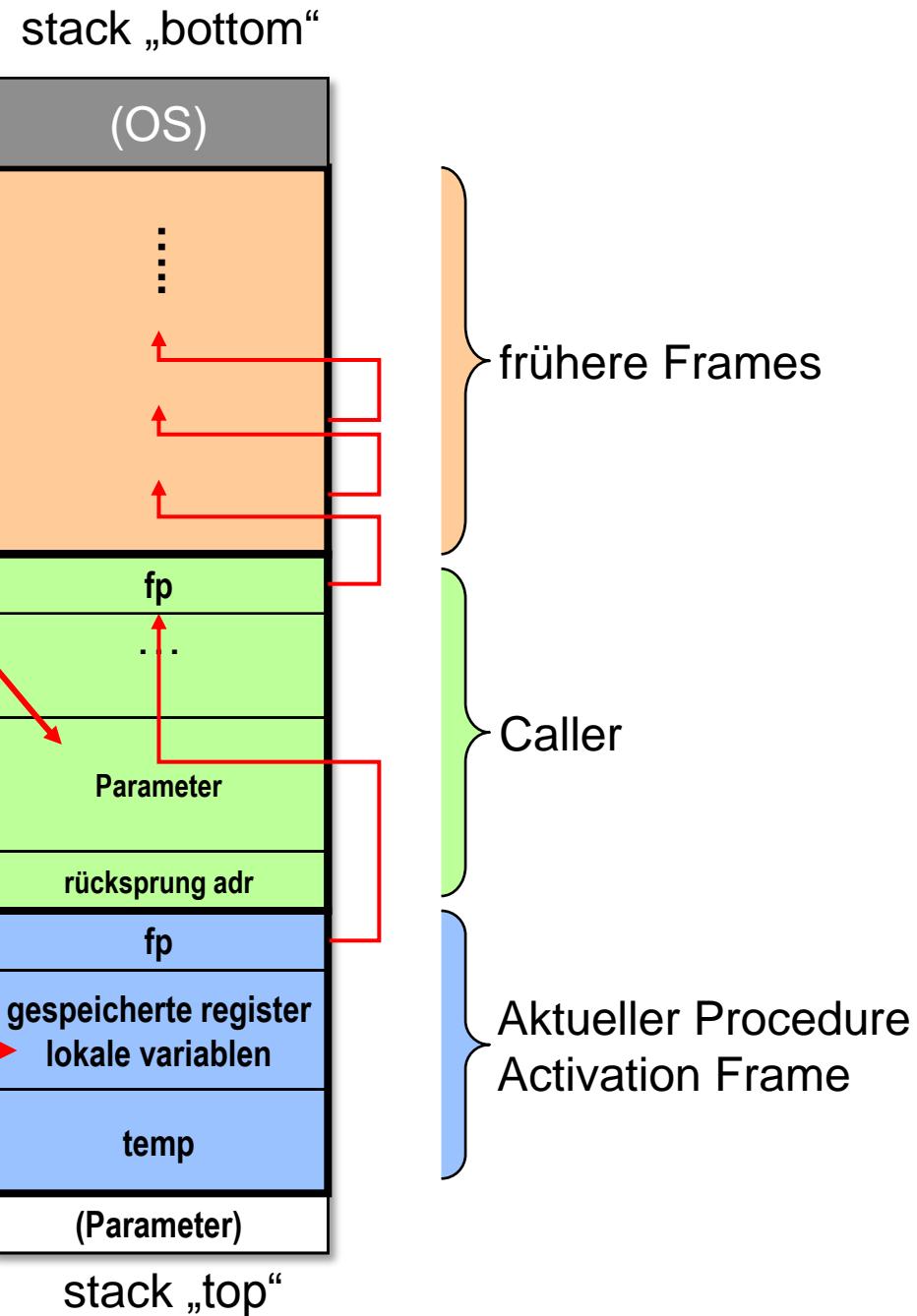
Objekte

Stack frame Struktur

```
procedure CallMe(a,b: integer);  
var c,d,e: char;  
....
```

frame pointer **FP**

stack pointer **SP**



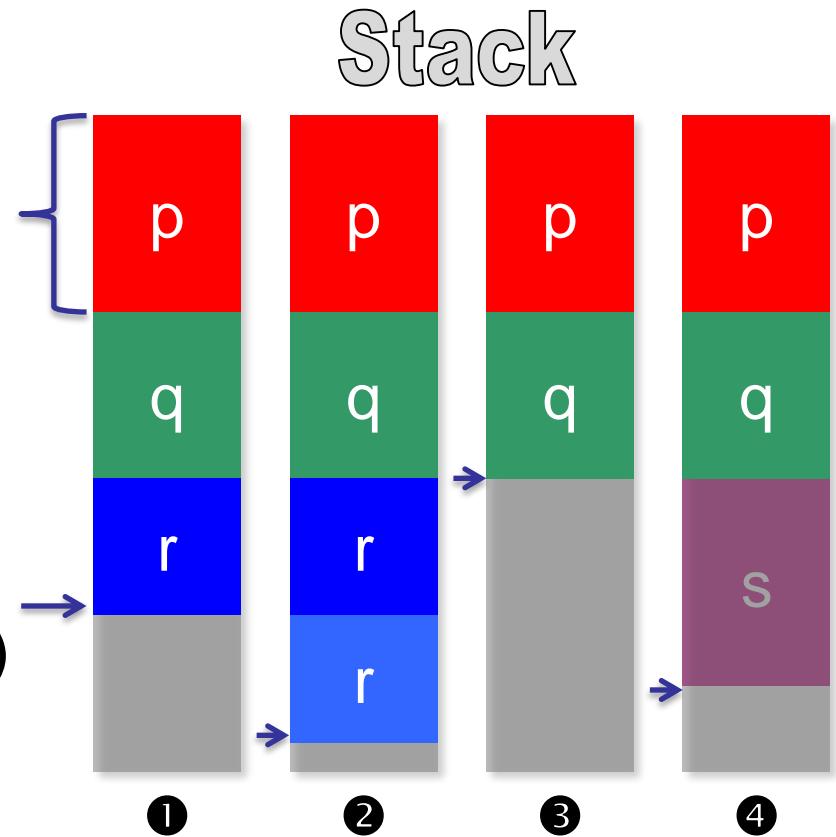
4.1.1. Stackverwaltung

- Prozeduren p, q, r, s
- Ablaufsszenario

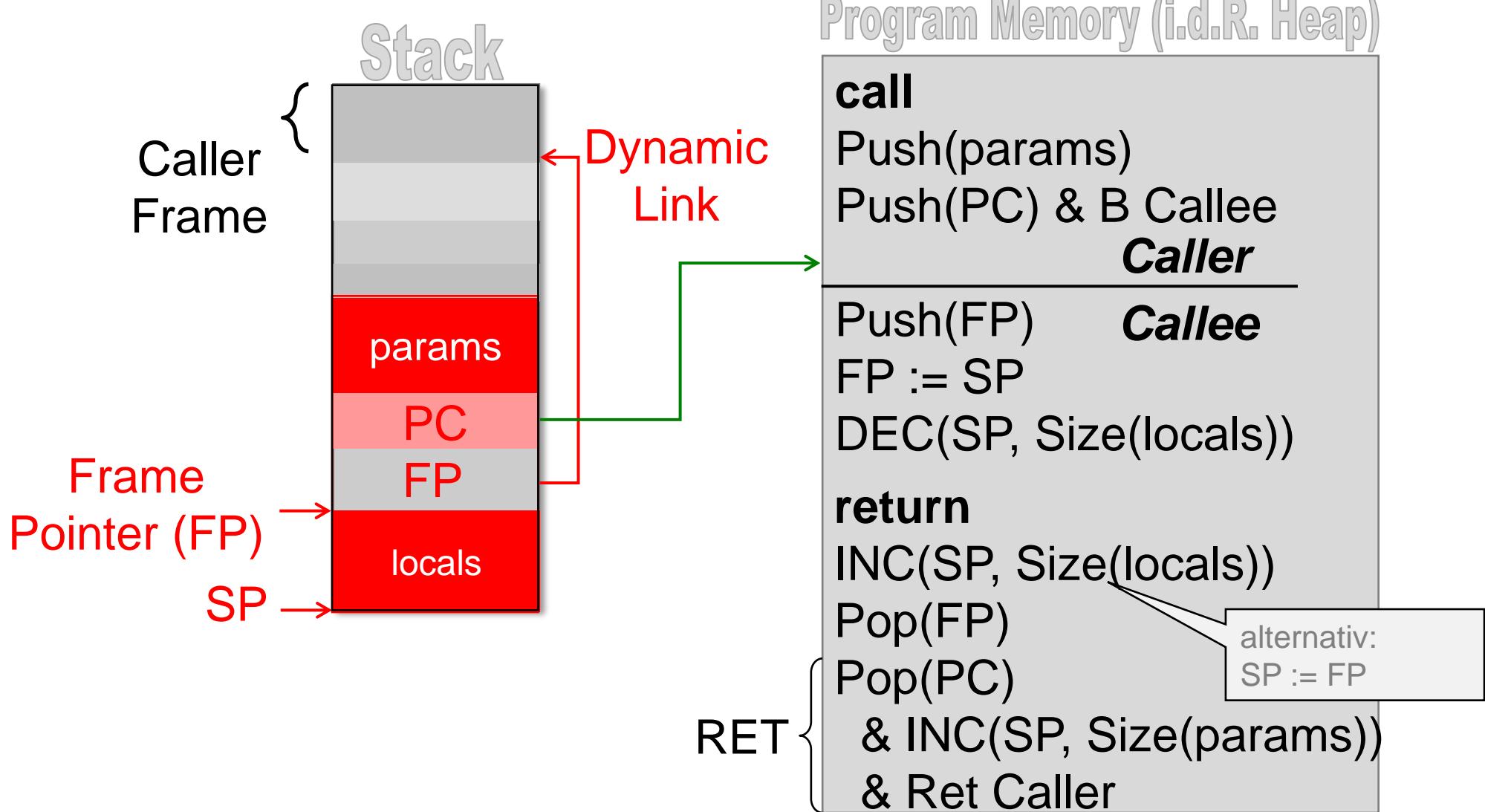
call p
call q
call r ①
call r ②
return r
return r ③
call s
call s ④
return q
return p

Procedure Activation Frame (PAF)

Stack
Pointer (SP)



Procedure Activation Frame

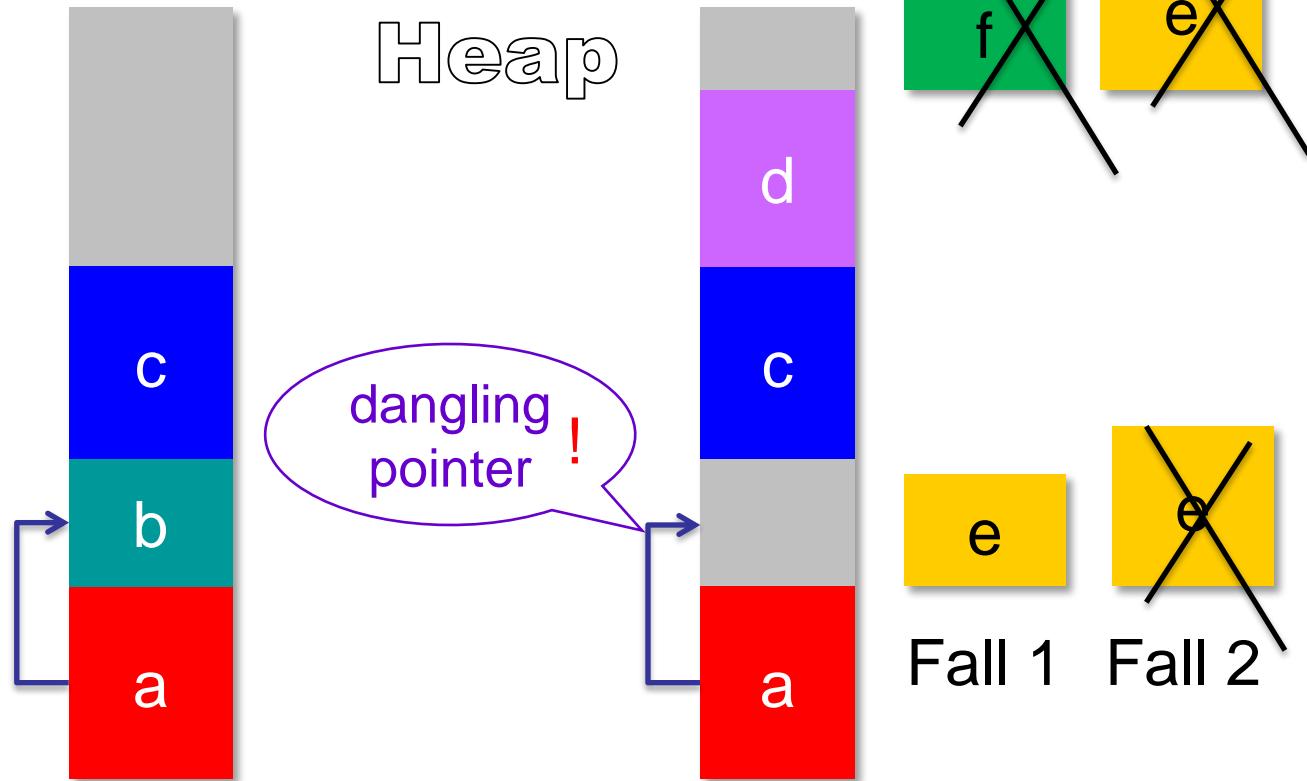


4.1.2. Heapverwaltung

- Objekte a, b, c, d, ...
- Allokationszenario

alloc(a)
alloc(b)
alloc(c)
dispose(b)
alloc(d)
alloc(e)
alloc(f)

pointer
(reference)



4.1.2.1. Heap Organisation

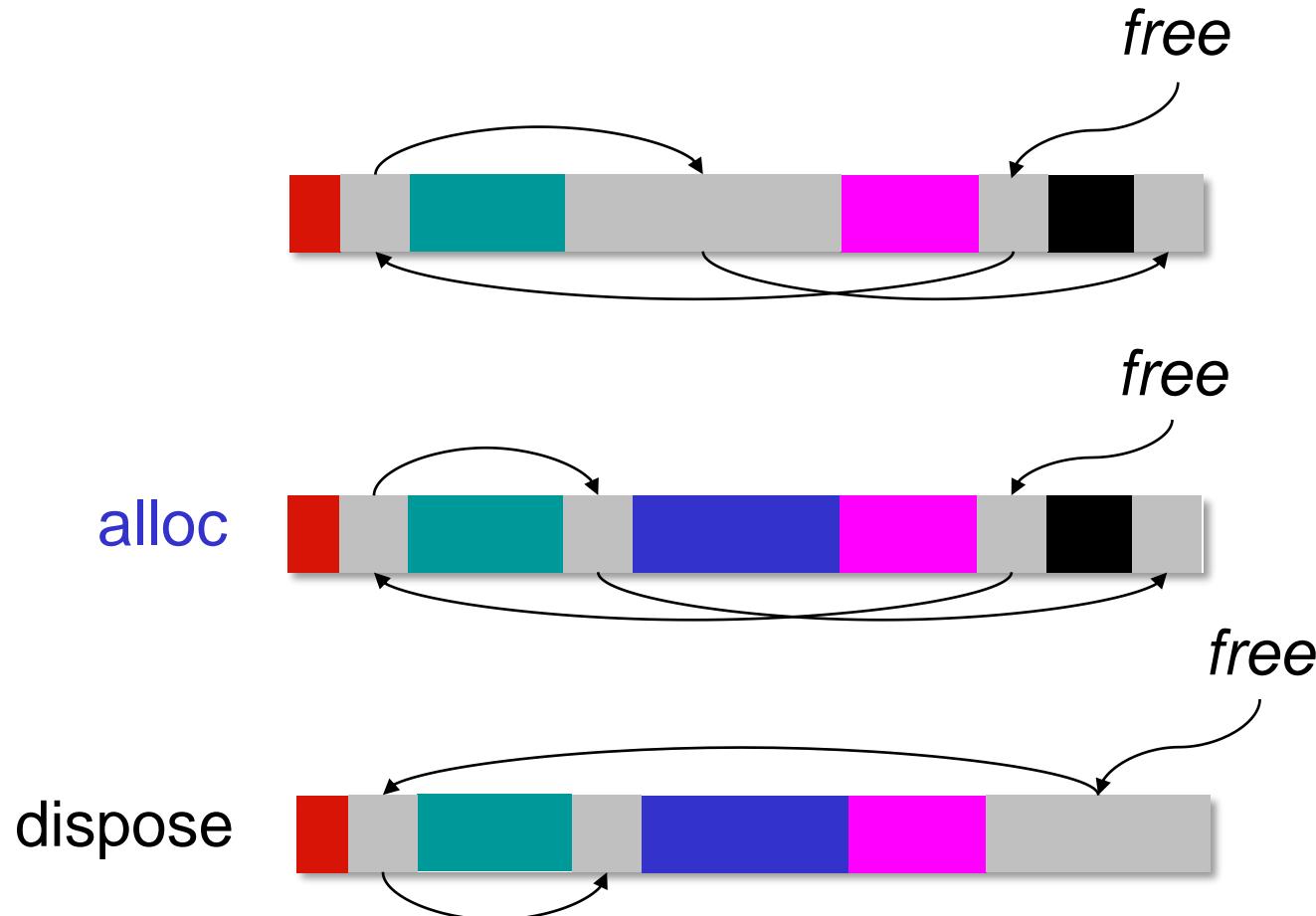
- Probleme
 - Begrenztheit des Objektheaps (■ f, Fall 1)
 - (Externe) Fragmentierung (■ e, Fall 2)
 - Dangling Pointers (a zeigt auf b)
 - Garbage*
- Lösungsansätze: Verwaltungsstrukturen
 - Freie Blockgrößen und Verkettung der freien Blöcke („free list“)
 - Standardblockgrößen und Belegungstabelle (Bitmap, Binärfolge mit 0 = frei, 1 = belegt)
 - Automatische Freigabe nicht mehr benutzter Blöcke durch das System („Garbage Collection“)

typische Probleme
der expliziten Allokation

Beispiel 1: free-Liste

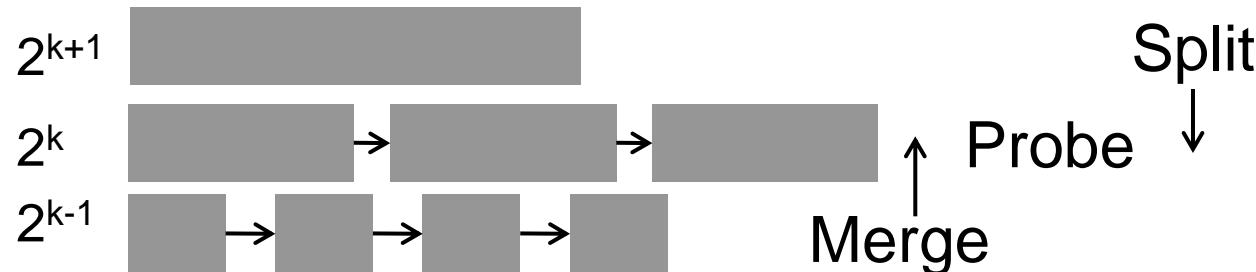
- Erzeugen der free-Liste durch Verkettung der Lücken
(Verkettung in den Blöcken)
- Allozierung neuer Blöcke durch Traversierung der free-Liste
 - Strategien
 - **first fit** erster verfügbarer, ausreichend großer Speicherblock
 - **best fit** kleinster verfügbarer, ausreichend großer Speicherblock
 - **next fit** nächster verfügbarer, ausreichend großer Speicherblock
 - **worst fit** größter verfügbarer Speicherblock
 - Abschneiden der benötigten Blockgrösse am Lückenende
- Freigeben nicht mehr benutzter Blöcke durch
 - Einfügen in *free* Liste
 - Wenn möglich verschmelzen

free-List Ablaufsszenarium



Beispiel 2: Buddy-System

- Folge von Blockgrößen mit der Eigenschaft, dass zwei alignierte, aufeinanderfolgende Blöcke (buddies*) einer Grösse zu einem Block der nächsthöheren Grösse verschmolzen werden können, z. B.
 - Zweierpotenzen $x_k = 2^k$
 - Fibonacci Zahlen $x_{k+2} = x_{k+1} + x_k$; $x_0 = 0$; $x_1 = 1$
- Unterhalte separate free-Liste für jede Blockgröße



- Interne Fragmentierung: $50\% < \text{Nutzgrad} \leq 100\%$

*Buddy= Kamerad, Kollege

Buddy-System Operationen

- $\text{alloc}(\text{size})$ mit $2^{k-1} < \text{size} \leq 2^k$

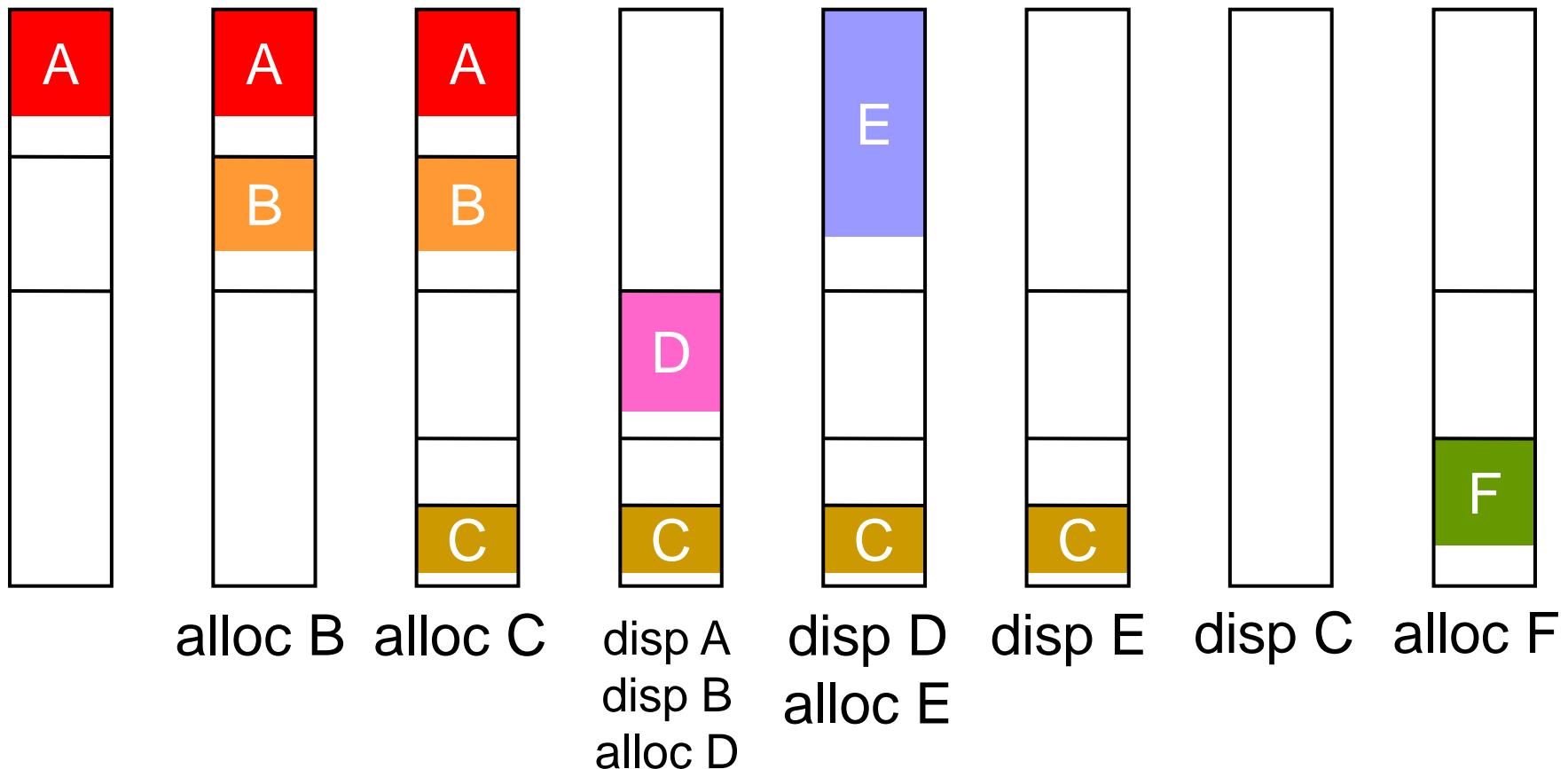
```
if freeList[k] nonempty then „quick fit“  
else { probe  
with upper sizes:  $2^{k+1} = 2^k + 2^k$  (split)  
with lower sizes:  $2^{k-1} + 2^{k-1} = 2^k$  (merge)  
}
```

nur falls dispose
„ohne Verschmelzen“

- **dispose**
 - Ohne Verschmelzen: Einfügen am Ende der free-Liste
 - Mit Verschmelzen: Benutze bitmap[k] statt free-Liste

10010001

Buddy-System Beispiel



Baumstruktur des Buddy Systems

1 Mbyte block	1 M				
Request 100 K	A = 128 K	128 K	256 K		512 K
Request 240 K	A = 128 K	128 K	B = 256 K		512 K
Request 64 K	A = 128 K	c = 64 K	64 K	B = 256 K	512 K
Request 256 K	A = 128 K	c = 64 K	64 K	B = 256 K	D = 256 K
Release B	A = 128 K	c = 64 K	64 K	256 K	
Release A	128 K	c = 64 K	64 K	256 K	
Request 75 K	E = 128 K	c = 64 K	64 K	256 K	
Release C	E = 128 K	128 K	256 K		512 K
Release E		512 K			
Release D		1 M			

Figure 7.6 Example of Buddy Sys

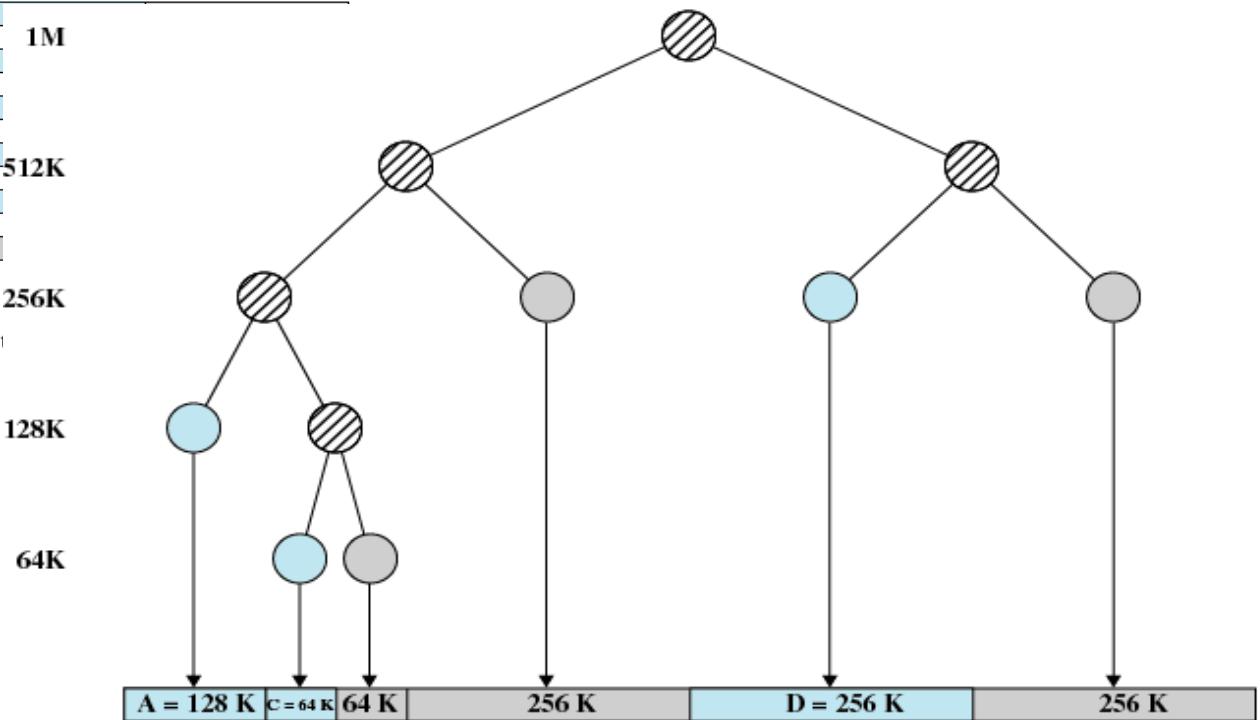
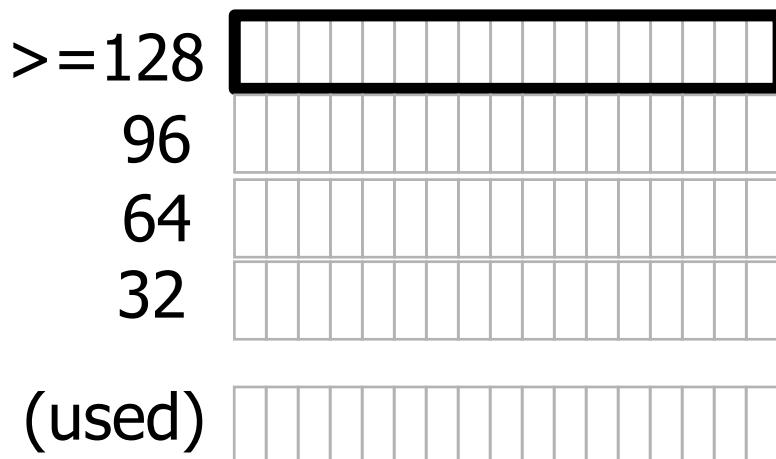


Figure 7.7 Tree Representation of Buddy System

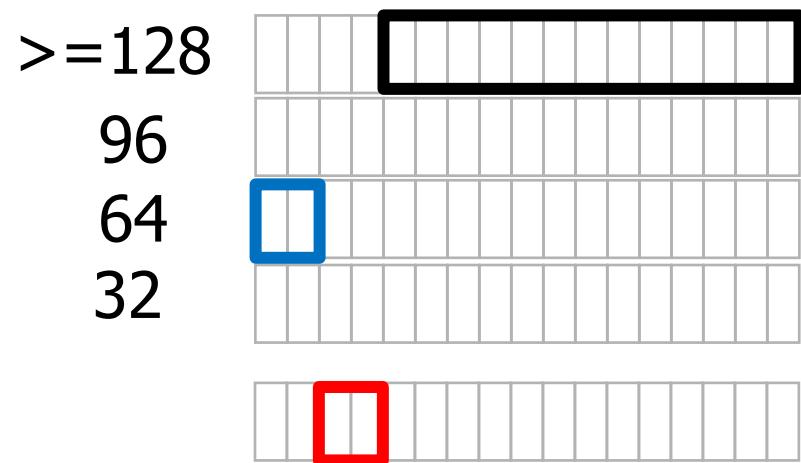
Eine Hybride Allokationsstruktur

- Mindest-Block-Grösse, Granularität B = 32 (Bytes)
 - Listen für Blöcke der Größen $x = k \cdot B$ ($k = 1, 2, \dots, n$) und für Blöcke mit $x > n \cdot B$
 - Allocate via free-Listen, ev. nach Splitting
 - Rückgabe unbenutzter Blöcke an die *free*-Listen durch sporadisches „Einsammeln“ (Garbage Collection)

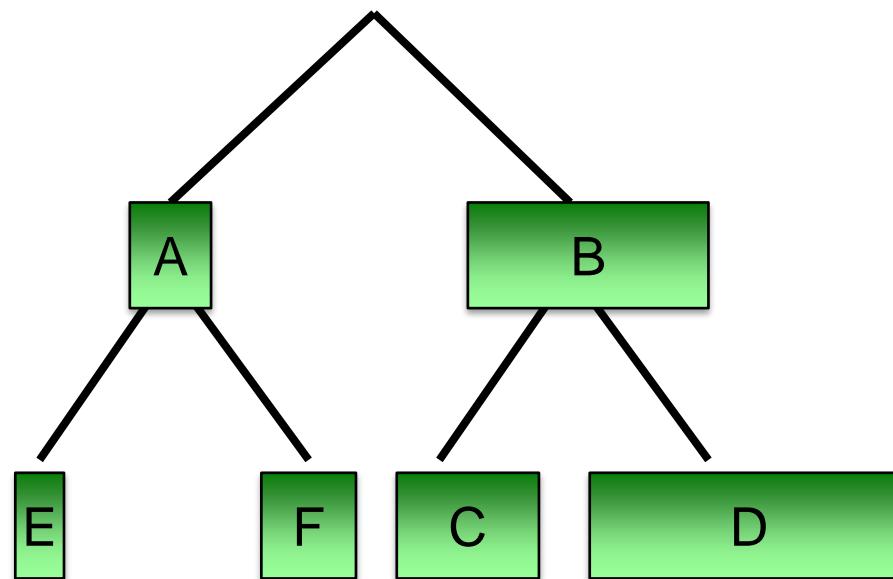
Anfangszustand ($n=4, B=32$)



`alloc(50), alloc(50), dispose`



Block Pool Darstellung: Balancierter Baum (Sortierung nach Größe)



Optimierungen bei der Speicher-Allokation

- Verwendung von Slab Cache
 - Vorallozierte Blöcke häufig genutzer Grösse (z.B. im Linux-Kern)
- Einteilung in Global Heap und Per-Prozessor Heap
 - für Multi-Prozessor Systeme mit Schwergewichtsprozessen (z.B. in Hoard Allocator)
 - zur Vermeidung gemeinsamer Nutzung der gleichen Cache-Lines auf verschiedenen Prozessoren
- Allokationen in Chunks
 - nachträgliche Unterteilung der Chunks, um System-Call bei jeder Allokation zu vermeiden (Artefakt der Schwergewichtsprozesse, z.B. genutzt in malloc)

4.1.2.2. Garbage Collectors

- Reference Counting
- Mark & Sweep
- Kopierend
- Mark & Compact
- Mit Objekt Generationen *
- Inkrementell & Concurrent
- Cache-bewusste Garbage Kollektion
- Verteilte Garbage Kollektion

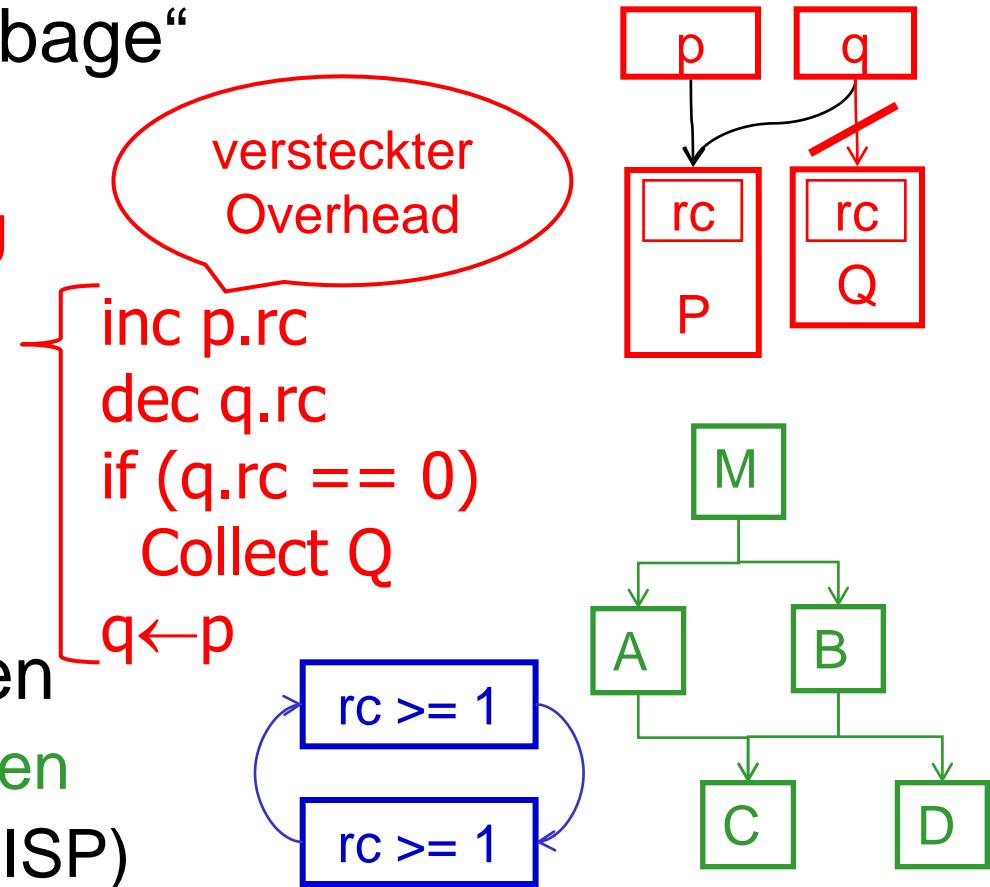
Es folgen nun Beispiele für Garbage Kollektoren

* generational GC, ephemeral GC

Beispiel 1: Referenzzählung

- Jedes Objekt enthält Referenzzähler rc
- $rc = 0 \Rightarrow$ Objekt ist „Garbage“
- Probleme
 - Ineffiziente Buchhaltung

p, q Objektzeiger
 $q = p$
 - Zirkuläre Strukturen (!)
- Geeignete Anwendungen
 - Softwaremodulhierarchien
 - DAG-Strukturen (z. B. LISP)



Reference Counting Pseudocode

Allokation

allocate()

```
newcell = free_list  
free_list = next(free_list)  
return newcell
```

New()

```
if free_list == nil  
    abort „Memory exhausted“  
newcell = allocate()  
RC(newcell) = 1  
return newcell
```

Update

free(N)

```
next(N) = free_list  
free_list = N
```

delete (T)

```
RC(T) = RC(T)-1  
if RC(T) == 0  
    for U in Children (T)  
        delete(*U)  
    free(T)
```

Update(R,S)

```
RC(S) = RC(S) +1  
delete (*R)  
*R = S
```

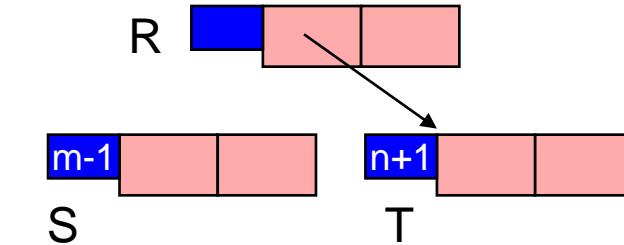
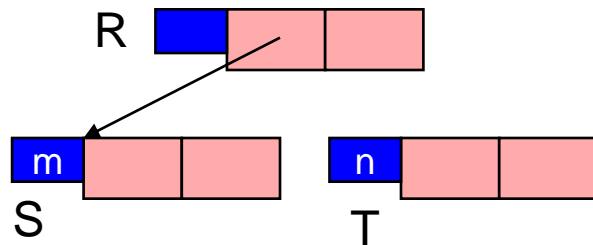
a = b translates to Update(a,b)

a = New() translates to Update(a,New())

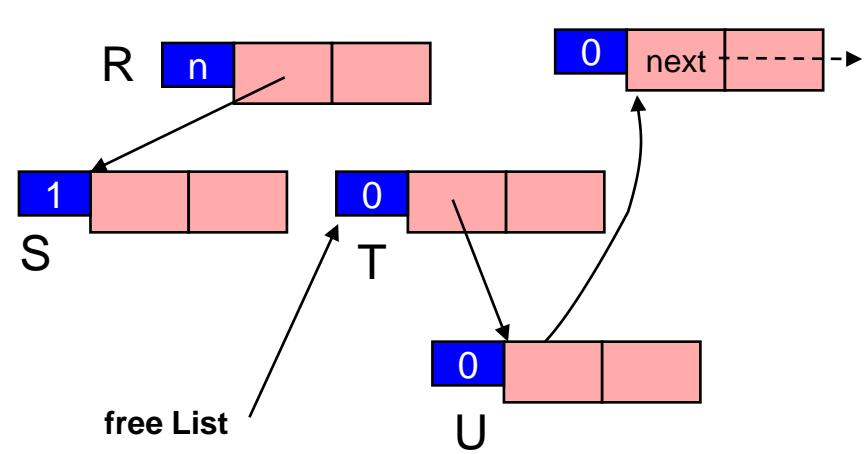
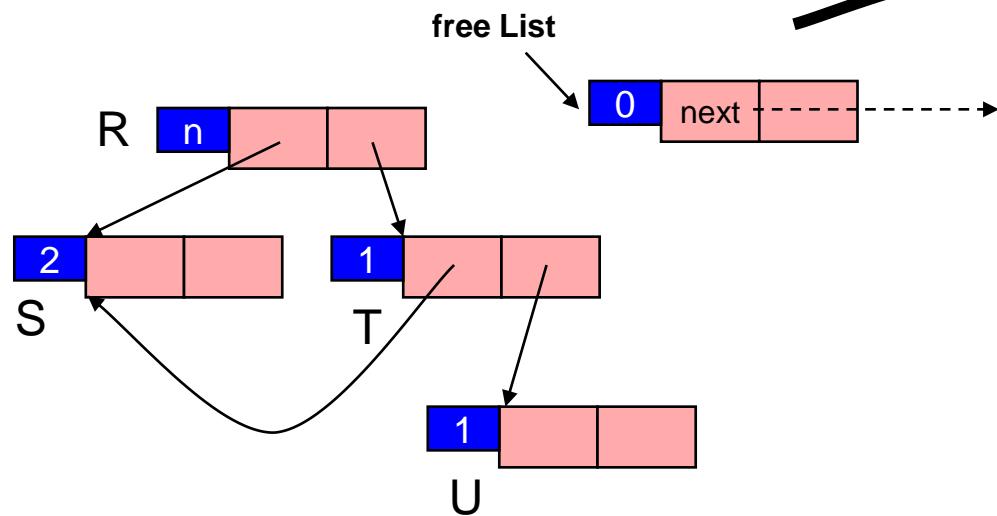
a = NIL translates to Update(a,NIL)

Beispiele

Update(left(R),T)



Update(right(R),NIL)



Vor und Nachteile des Reference Counting

- 😊 Memory Management Overhead verteilt über die Ausführungszeit
- 😊 Lokalität der Referenz zur Kollektionszeit
- 😊 Erlaubt eine Stack-ähnliche Allokation von temporären Objekten.
(Beobachtung: wenige Objekte verteilt, viele kurzlebig)
- 😊 „in-place“ Kopie nicht mehr referenzierter Objekte möglich
- 😊 sofortige Finalisierung möglich (vgl. Mark&Compact in .NET)

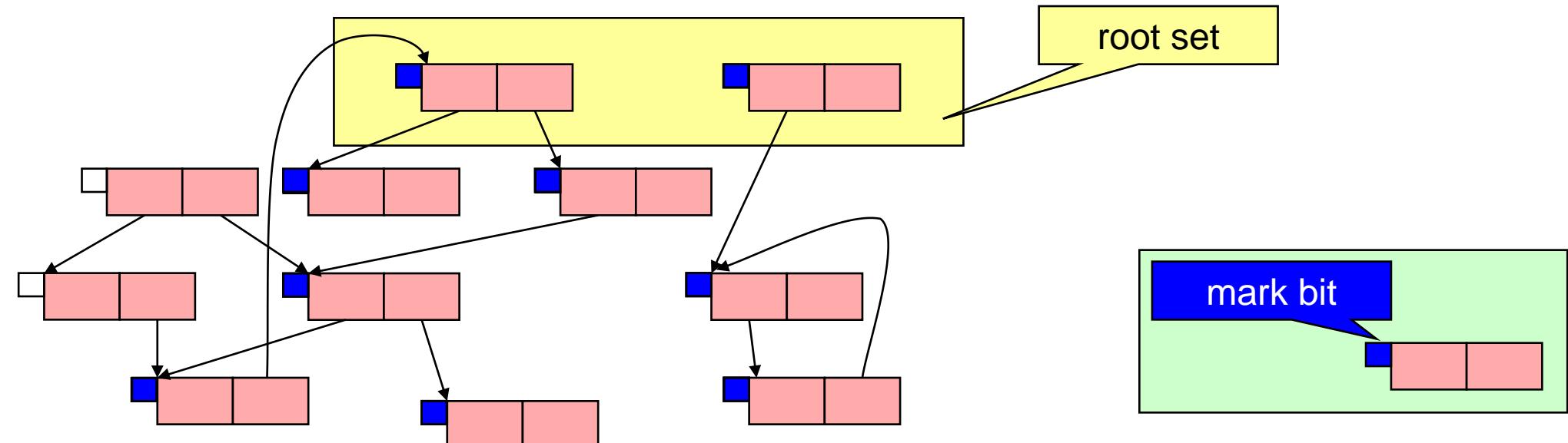
- 😢 hoher (Zeit-)Aufwand für Referenznachführung auf Standard-Hardware
- 😢 Compiler schwieriger zu warten, da viele Abhängigkeiten vom Referenz-Zähler
- 😢 Speicherplatz: in naiven Ansätzen eine Pointer-Größe pro Referenz
- 😢 **Zyklische Datenstrukturen (z.B. doppelt verkettete Listen) können nicht aufgelöst werden**
- 😢 Fragmentierung

Beispiel 2: Mark & Sweep Collector

- „Stop-and-Go“ Verfahren
 - Wird bei Bedarf oder prophylaktisch gestartet
 - Unterbricht den produktiven Prozess
- Läuft in zwei Phasen ab
 - Garbage Detection (Mark)
 - Garbage Collection (Sweep)
- Basiert auf dem Objektgraph
 - Typisiert gemäss höherer Programmiersprache
 - Beschreibung durch Metadaten
 - Offsets der Zeiger im Block

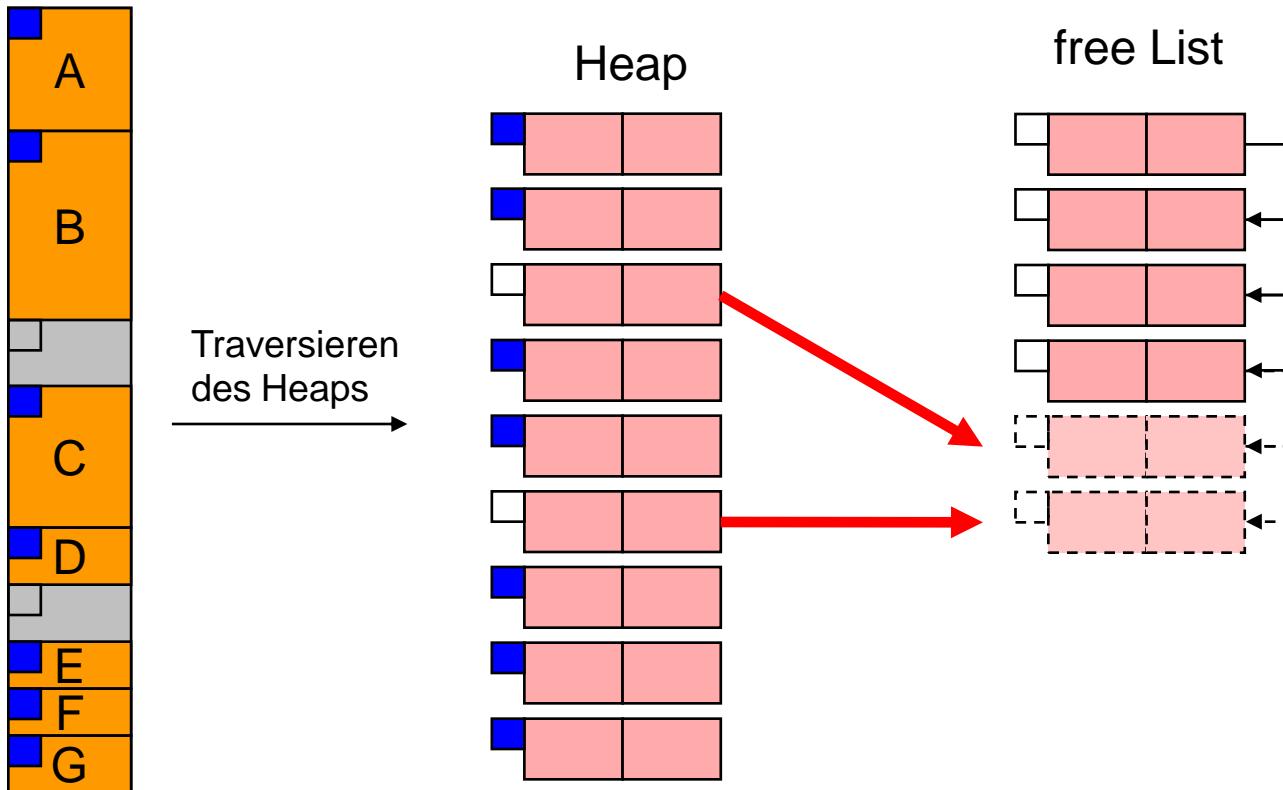
Mark Phase (Mark reachable blocks)

- Bestimme das Root-Set (Ankermenge der Zeiger)
 - globale Zeiger im Codebereich
 - lokale Zeiger auf dem Stack
 - temporäre Zeiger in Registern
- Traversiere den Objektgraph vom Root Set aus und markiere alle traversierten Objekte.



Sweep Phase (Return unmarked blocks)

- Traversiere den Heap linear. Gebe Bereiche zwischen markierten Blöcken an die free-Listen zurück.
- Kann mit alloc kombiniert werden („Lazy Sweep“)



Mark&Sweep Pseudocode

Allocation

allocate()

```
get element from free_pool  
return element
```

New()

```
if free_pool is empty  
    mark_sweep()  
newcell = allocate()  
return newcell
```

Mark&Sweep

mark_sweep()

```
for R in Roots  
    mark(R)  
sweep()  
if free_pool is empty  
    abort „Memory exhausted“
```

mark(N) =

```
if mark_bit(N) == unmarked  
    mark_bit(N) = marked  
    for M in Children(N)  
        mark(*M)
```

free(N) =

```
integrate N in free_pool
```

sweep ()

```
N = Heap_bottom  
while N < Heap_top  
    if mark_bit(N) == unmarked  
        free(N)  
    else mark_bit(N) = unmarked  
    N = N + size(N)
```

Root-Set Bestimmung

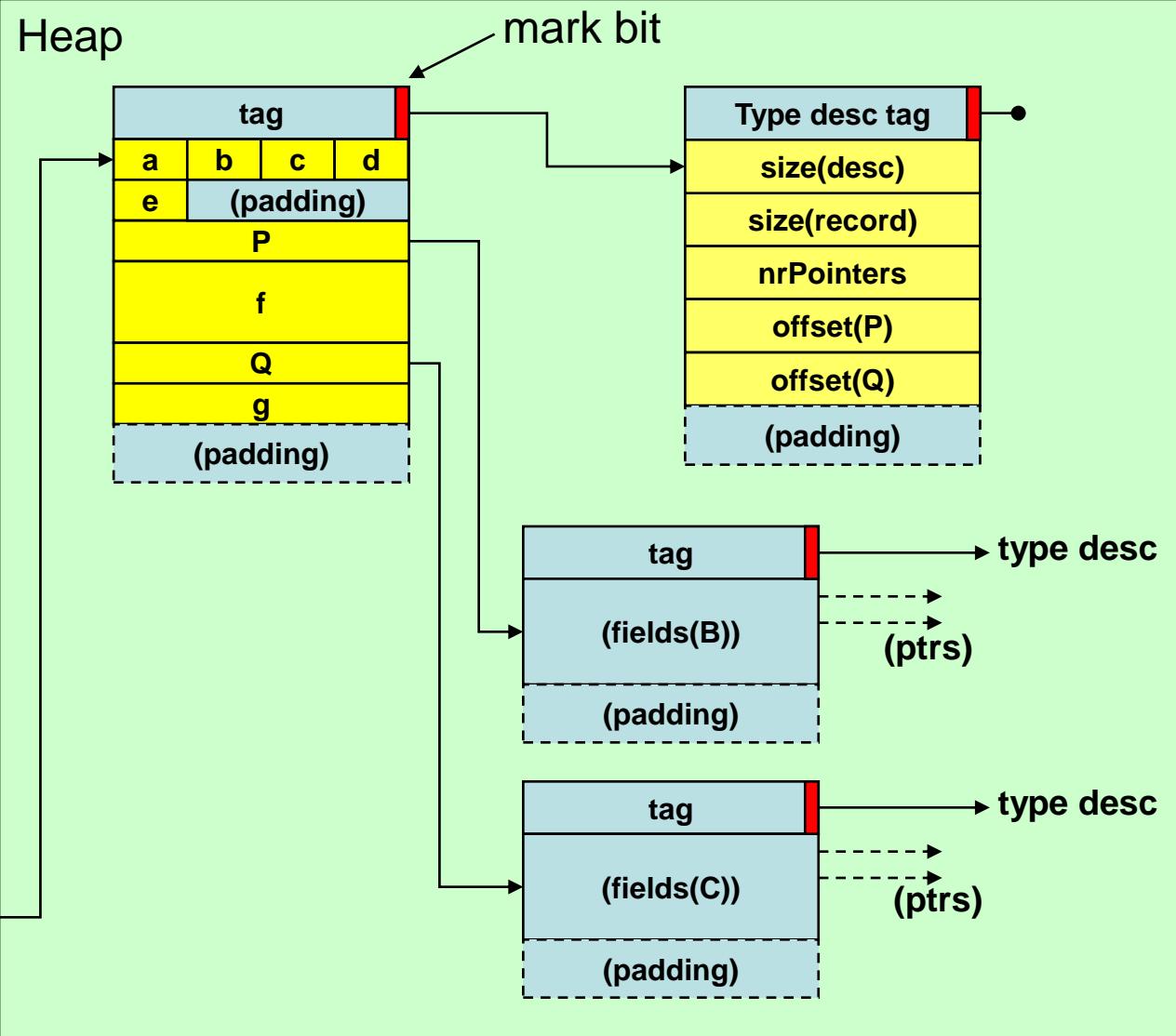
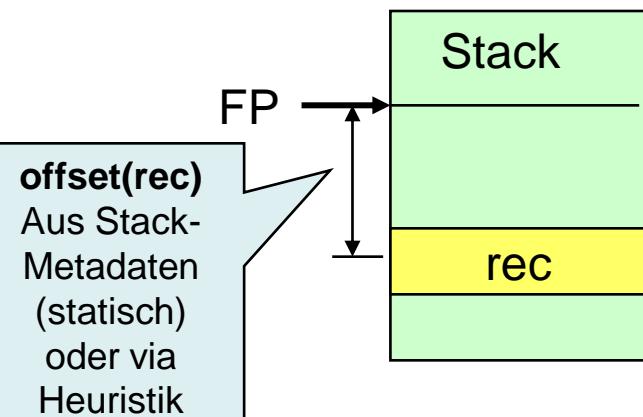
- Codebereich und Heapbereich
 - Compilererzeugte Metadaten
 - Zeiger Offsets in Modulen und Typen
 - Laufzeit Typdeskriptor
 - Verweis vom Objekt auf Typdeskriptor
- Stackbereich und Register
 - Compilererzeugte Metadaten
 - Zeiger Offsets in Prozeduren
 - oder
 - Spekulative Methode
 - Basiert auf Plausibilitätstests (Wert im Adressbereich des Heap?)
 - Entscheidet „konservativ“

Root-Set Bestimmung

Beispiel: Heap Meta Daten für ein record

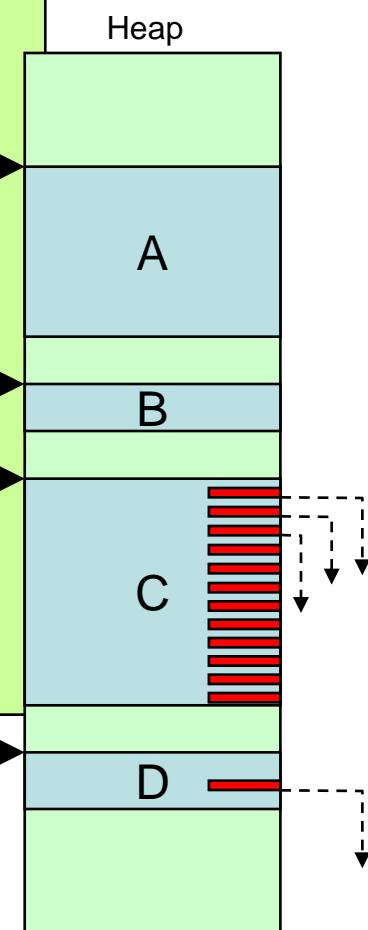
Code

```
A = pointer to record  
a,b,c,d,e:char;  
P: pointer to B;  
f: longreal;  
Q: pointer to C;  
g: longint;  
end;  
...  
var rec: A;  
new(rec);
```



Typisierte Programmiersprachen als ein nötiger Baustein der Garbage Collection

```
type
  Vector = pointer to array of real;
  Objekt= object x: real end Objekt;
  ObjektArray= pointer to array of Objekt;
  Composite = object
    c: ObjektArray;
  end Composite;
...
var
  A: Vector;
  B: Objekt;
  C: ObjektArray;
  D: Composite;
...
  new(A,10); new(B);
  new(C,10); new(D);
...
  B.x := 5; A[5] := 3; C[5] := B; D.c := C;
```



Untypisierte Sprache

1. Zeiger
Arithmetik

```
float *A;
A=new float[10];
A+= 8;
A[0] = 10;
```

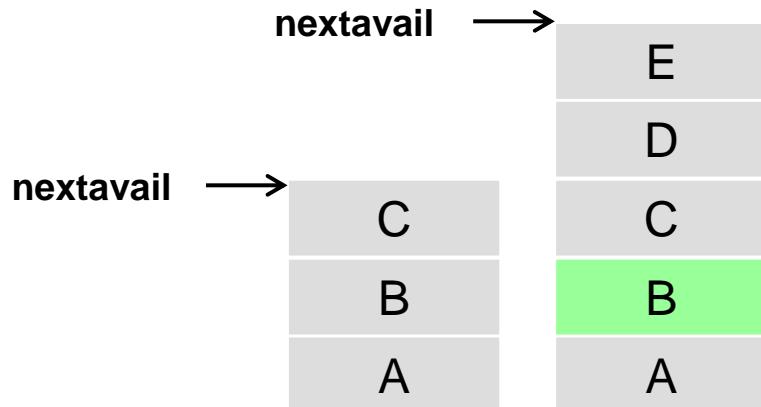
2. keine Typ-Deskriptoren
in Objekt-Files
3. keine Laufzeit-
Unterstützung

Vor und Nachteile des Mark & Sweep

- 😊 Zyklische Strukturen können aufgelöst werden (vgl. Reference Counting)
- 😊 i.A. bessere Performance als R.C.
- 😢 Stop-and-go Verfahren: Prozesse stehen still während Kollektion: Einsetzbarkeit in interaktiven oder gar real-time Systemen?
- 😢 Asymptotische Komplexität proportional zur Heap-Größe (im naiven Ansatz)
- 😢 Naiver Algorithmus erzeugt Fragmentierung

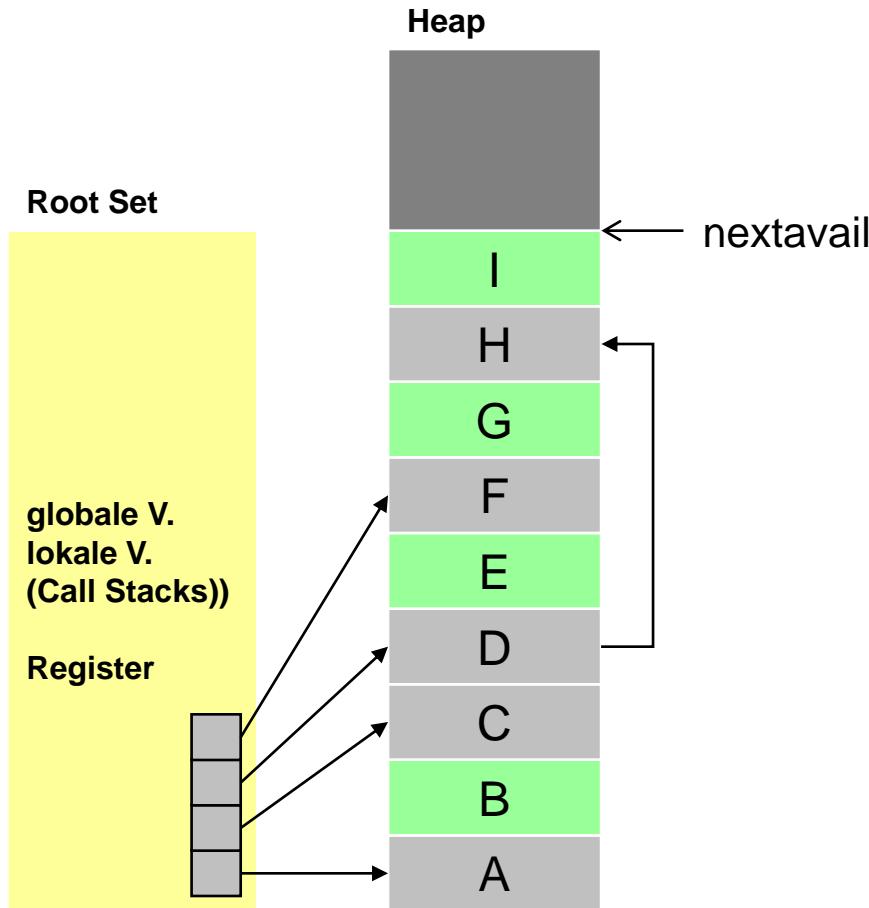
Beispiel 3: Mark & Compact in .NET

- Heap organisiert als lineare verkettete Liste von Objekten
 - Freier Platz durch „nextavail“ Pointer gekennzeichnet
 - Allokation durch Anhängen neuer Objekte

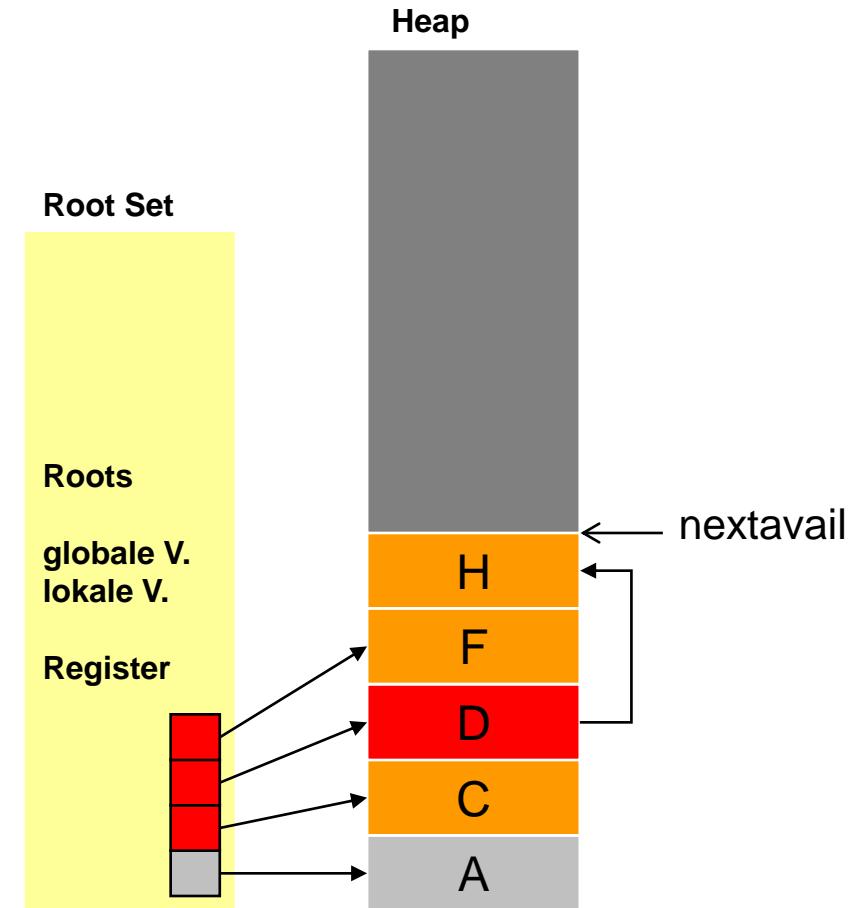


- Mark & Compact Garbage Collection
 - Markiere die noch benötigten Objekte
 - Root Set Bestimmung vollständig via Metadaten
 - wie beim Mark & Sweep GC
 - Kompaktifiziere Heap durch Verschieben

Illustration: Mark & Compact in .NET



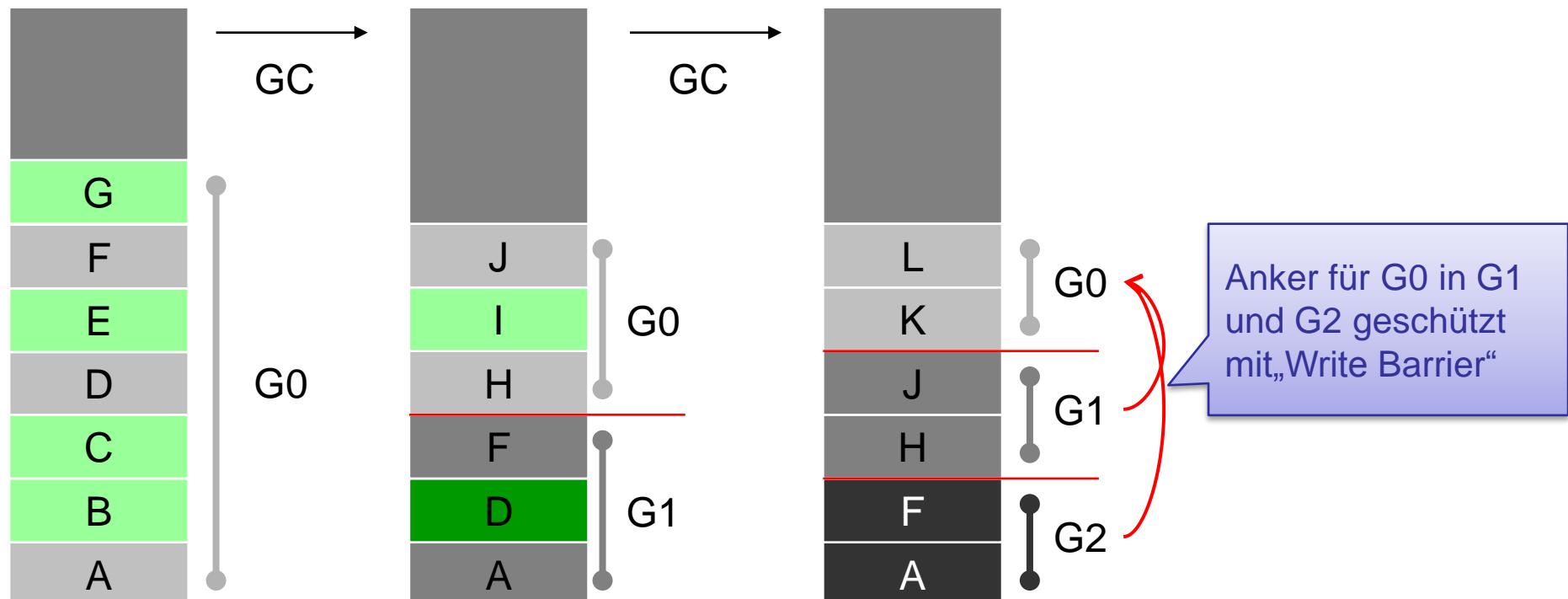
Allozierte Objekte im Heap



Managed Heap nach GC

.NET GC: Objekt Generationen

	Alter	Lebenserwartung	Kollektion
G0	jung	kurz	häufig
G1	mittel	mittel	gelegentlich
G2	alt	lang	selten



.NET GC: Finalisierung

- Finalisierer sind spezielle Methoden.

```
class MyObject {  
    ~MyObject() {  
        ...  
    }  
}
```

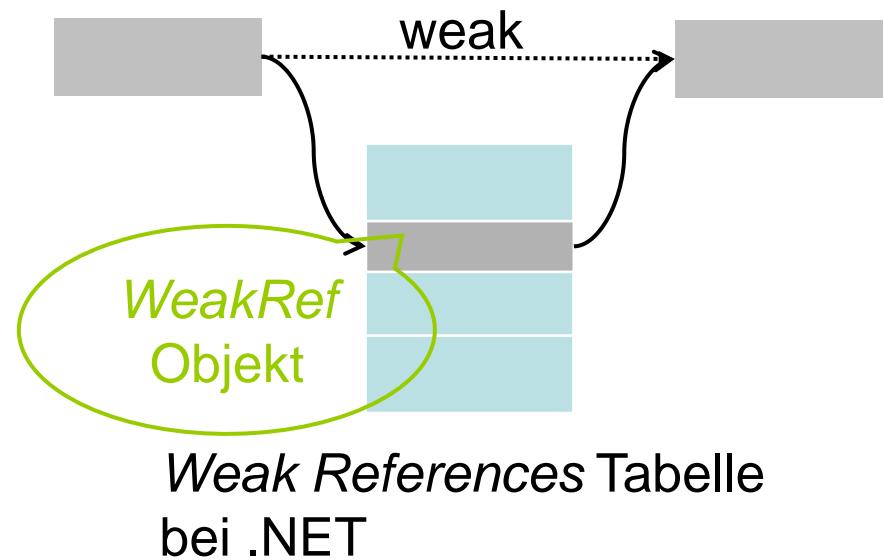
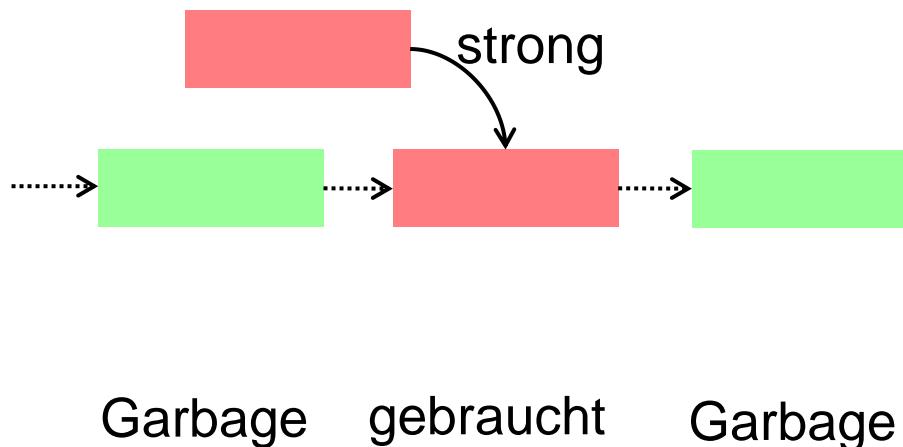
Finalisierer in C#

Finalisierer in C# sind **formal** den Desktruktoren in C++ ähnlich.
Semantisch sind sie etwas anderes.

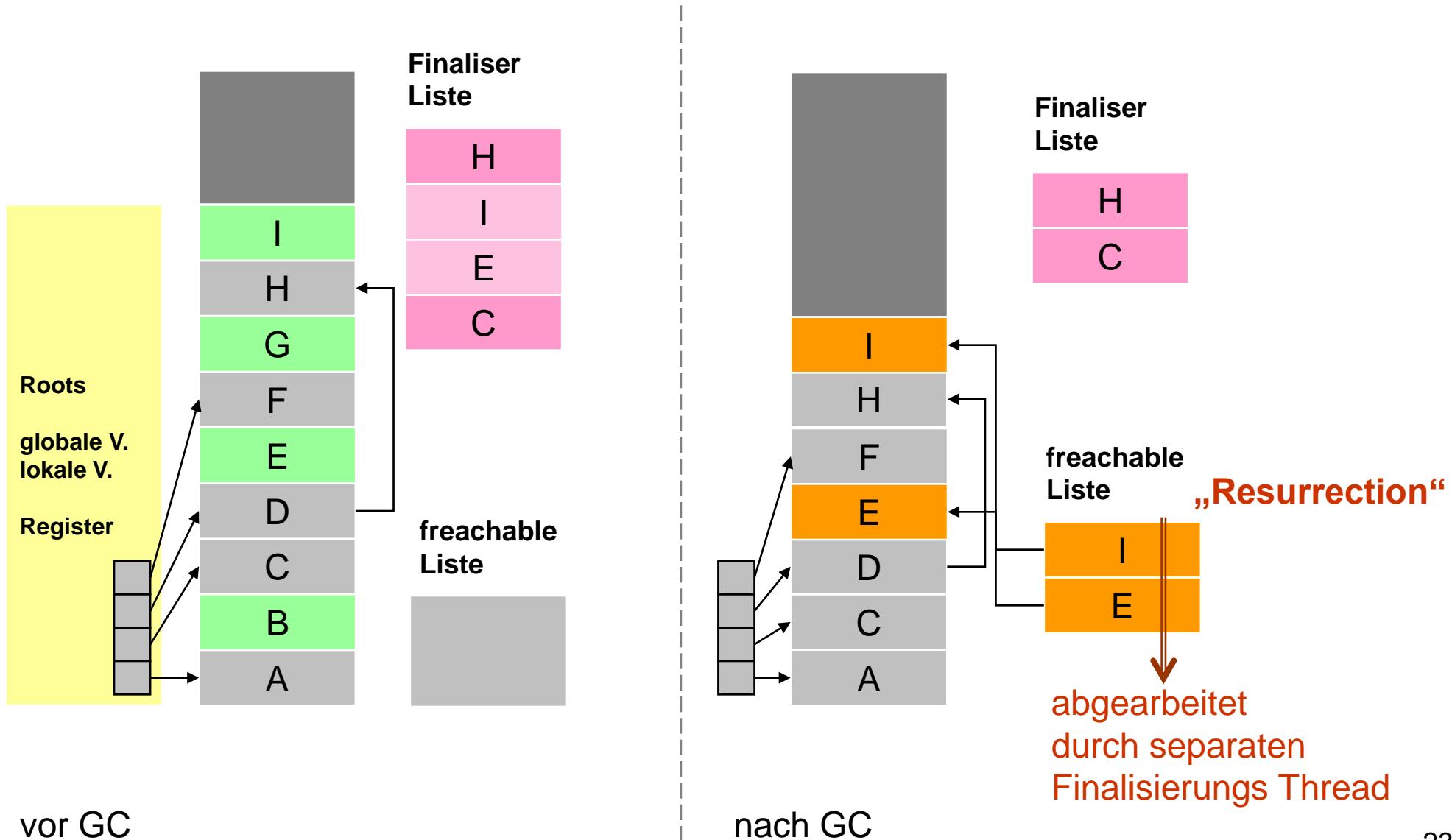
- Sie werden aufgerufen, wenn Objekte freigegeben werden. Sie können u.a. dienen zur
 - Etablierung der Konsistenz,
 - Sicherung von Pufferinhalten und Caches,
 - Rückgabe von Ressourcen,
 - Freigabe von Verbindungen.

„Weak“ Pointers

- Nur vollwertig in Verbindung mit „strong“ Pointers
- Ist für den Garbage Collector unsichtbar (trägt also nichts zur Mark-Phase des GC bei).
- Verwendet für Cache-Objekte, wie z.B. in Finalizer-Listen oder für Files

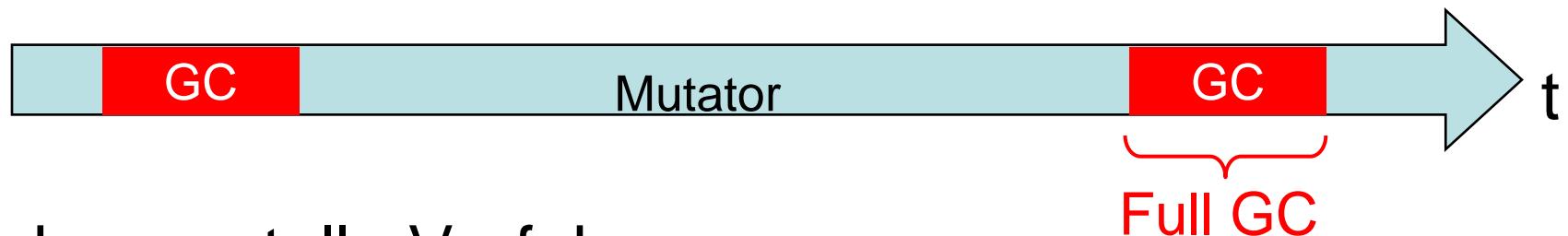


.NET GC: Finalisierung (2)

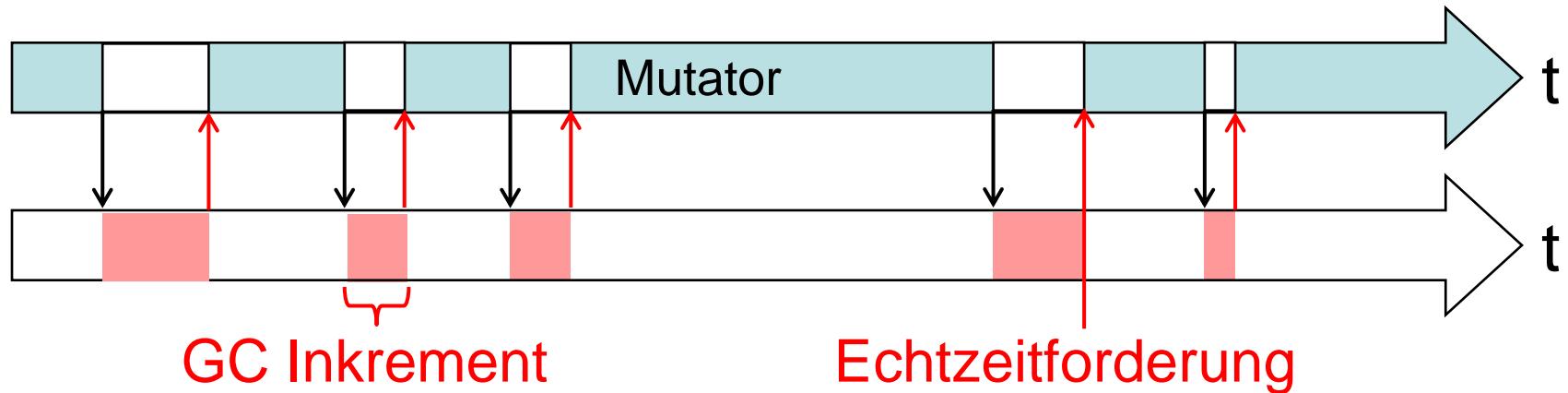


Beispiel 4: Inkrementelle Kollektion

- Stop-and-Go Verfahren



- Inkrementelle Verfahren



Synchronisationsproblem

time

P1

(2) **right(B)** \leftarrow **right(A)**;

(5) **right(A) ← right(B);**

(3) **right(A)** \leftarrow NIL;

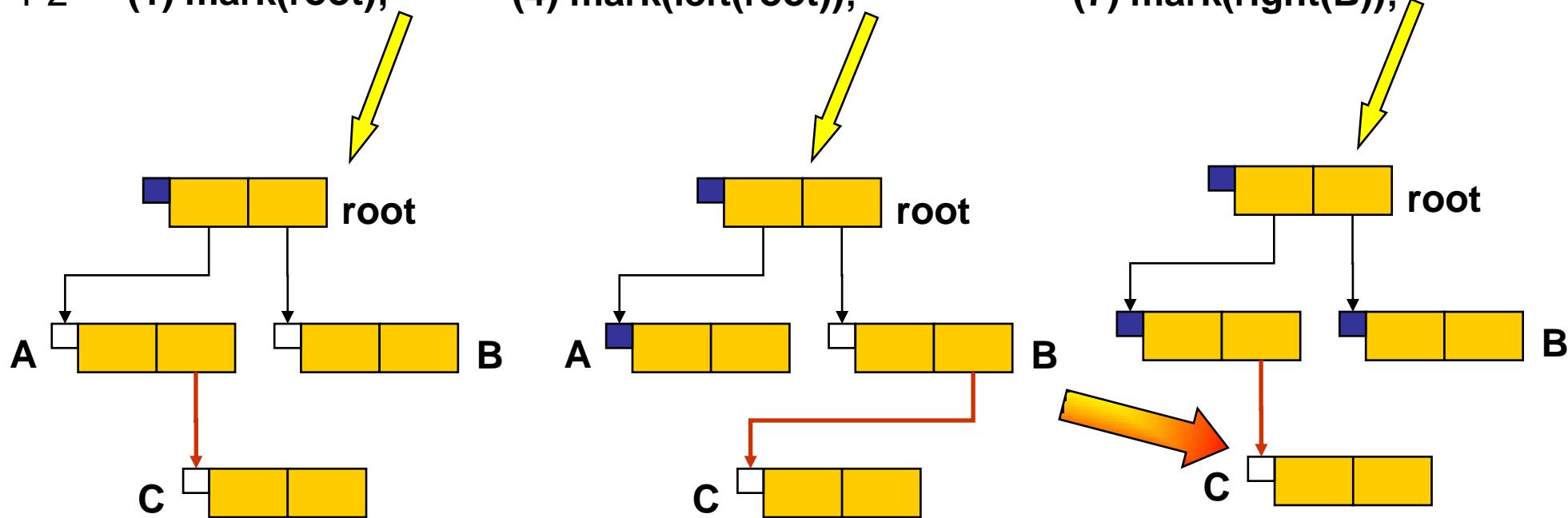
(6) **right(B) ← NIL;**

P2

(1) mark(root);

(4) `mark(left(root))`;

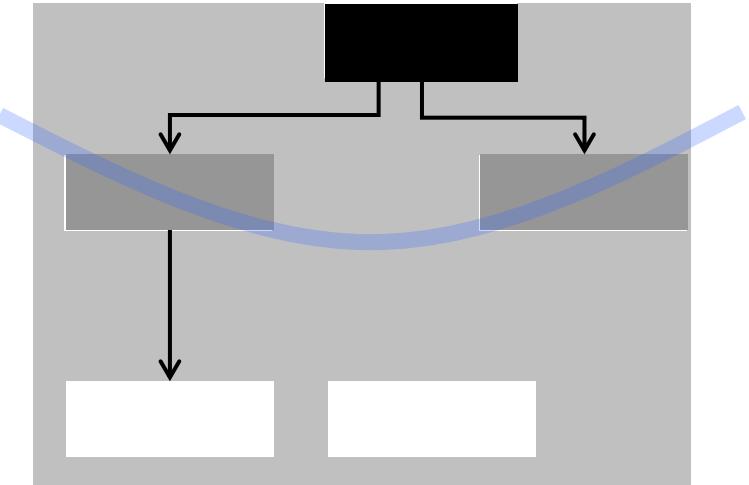
(7) mark(right(B));



Dreifarben Abstraktion

Wellenfront Modell

Zustand	Farbe
wurde traversiert, <i>hinter</i> Wellenfront	schwarz
wird jetzt traversiert / muss noch traversiert werden (ist bereits zur Bearbeitung eingetragen) <i>auf</i> Wellenfront	grau
Noch unberührt, <i>vor</i> Wellenfront	weiss



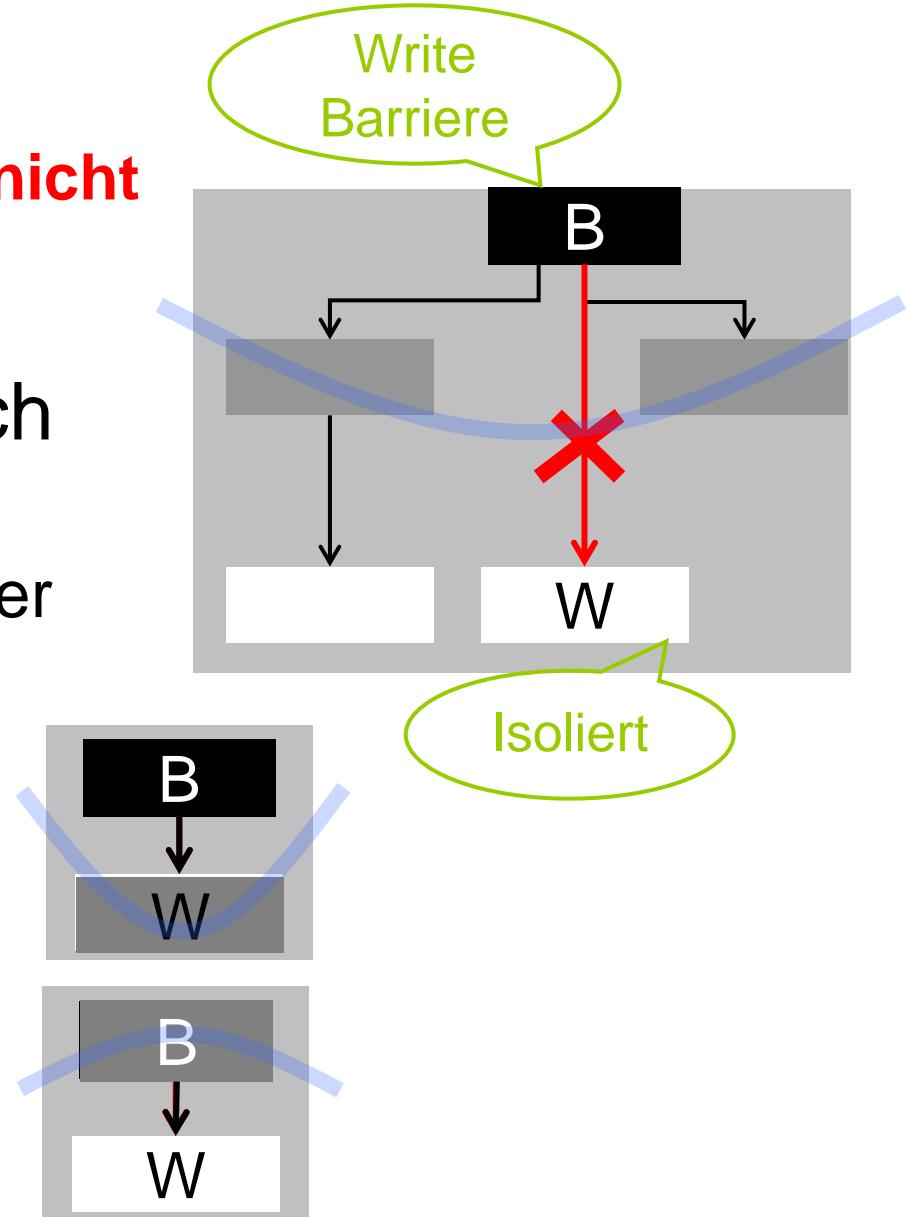
Verboten (nach der „Update“ Operation):

Auf einen weißen Knoten wird nur von schwarzen Knoten verwiesen.

Das Isolationsproblem

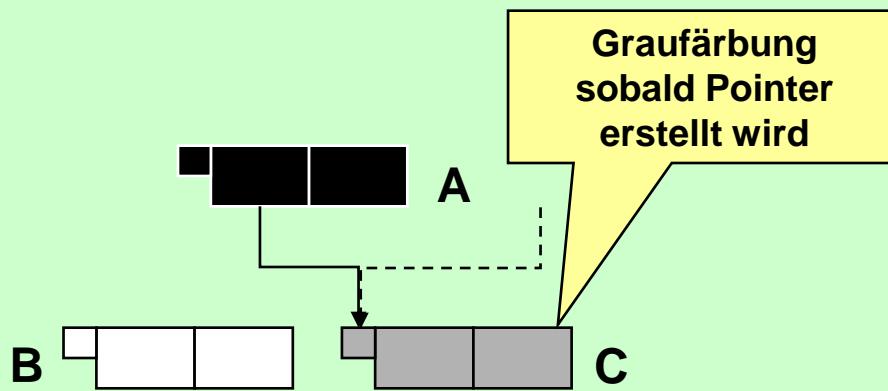
Auf einen weißen Knoten darf nicht nur von schwarzen Knoten verwiesen werden.

- Zeigerzuweisungen durch Mutator
 - Kritischer Fall: Neuer Zeiger *schwarz → weiss*
- Heilungsmöglichkeit
 - Read Barriere Dämon Graufärbung von W
 - Write Barriere Dämon Graufärbung von B

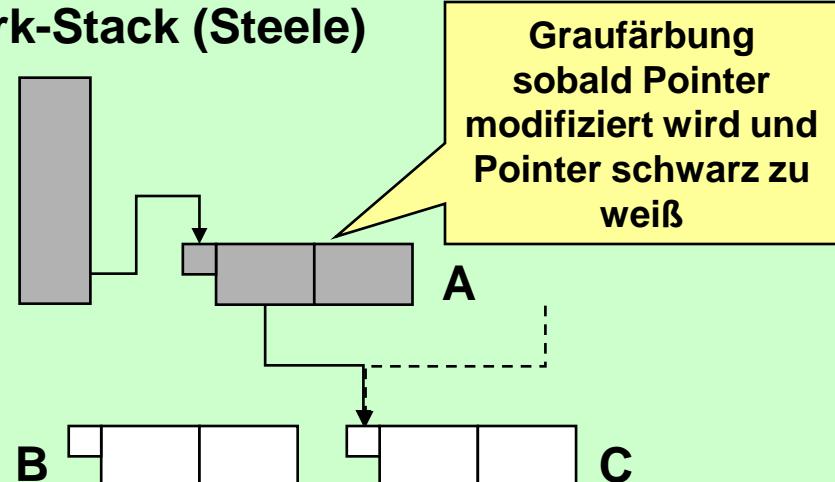


Beispiel: Inkrementelle Mark & Sweep Kollektoren

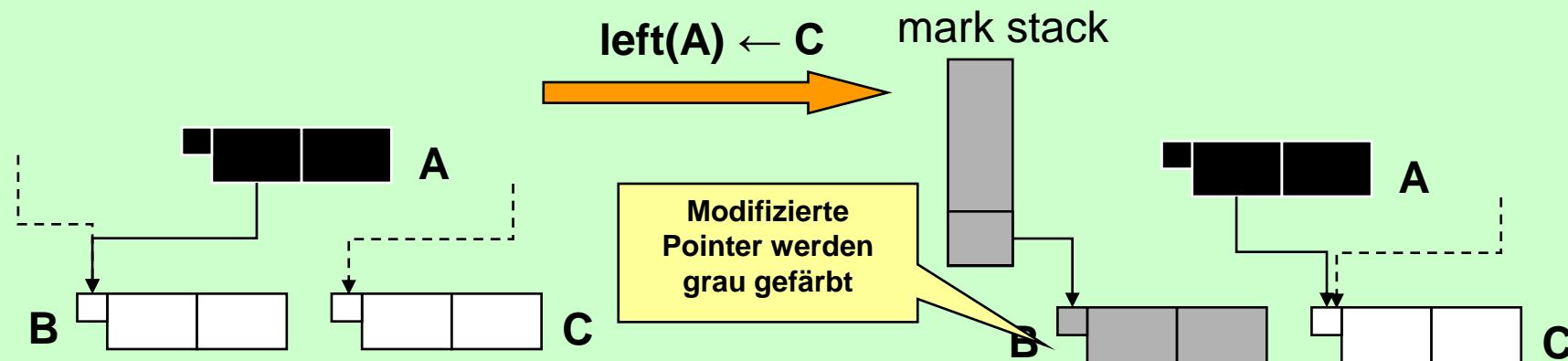
Explizite Farb-Bits (Dijkstra)



Mark-Stack (Steele)

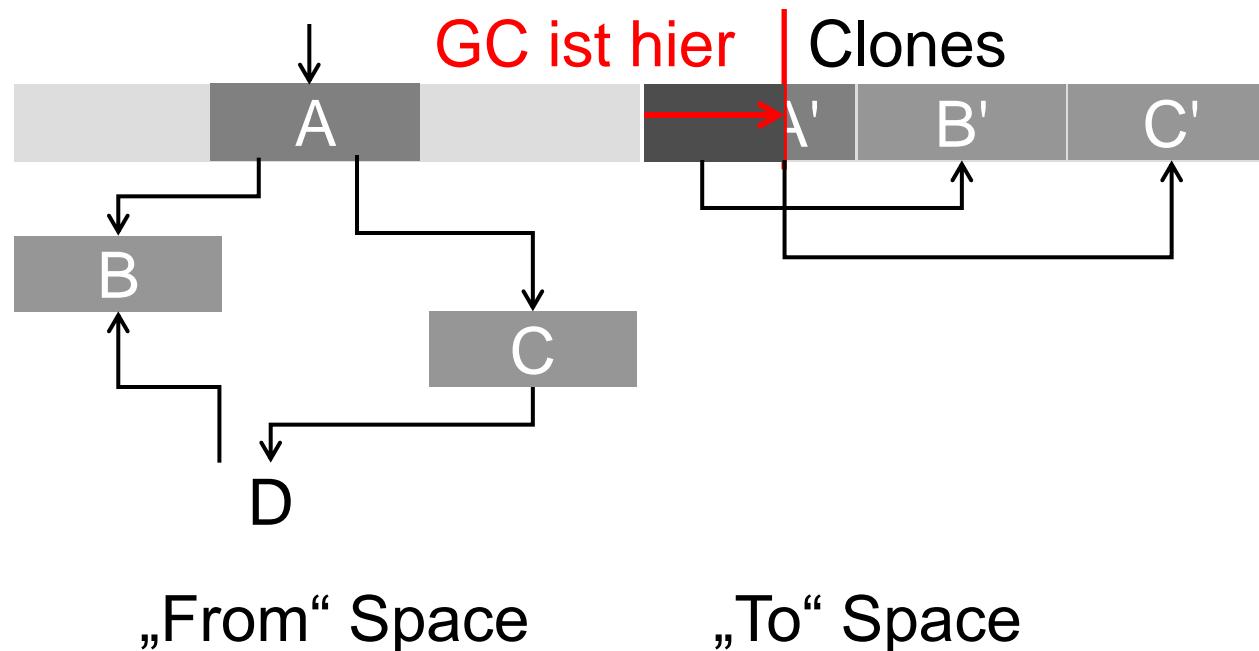


Mark-Stack (Yuasa) Snapshot at the beginning

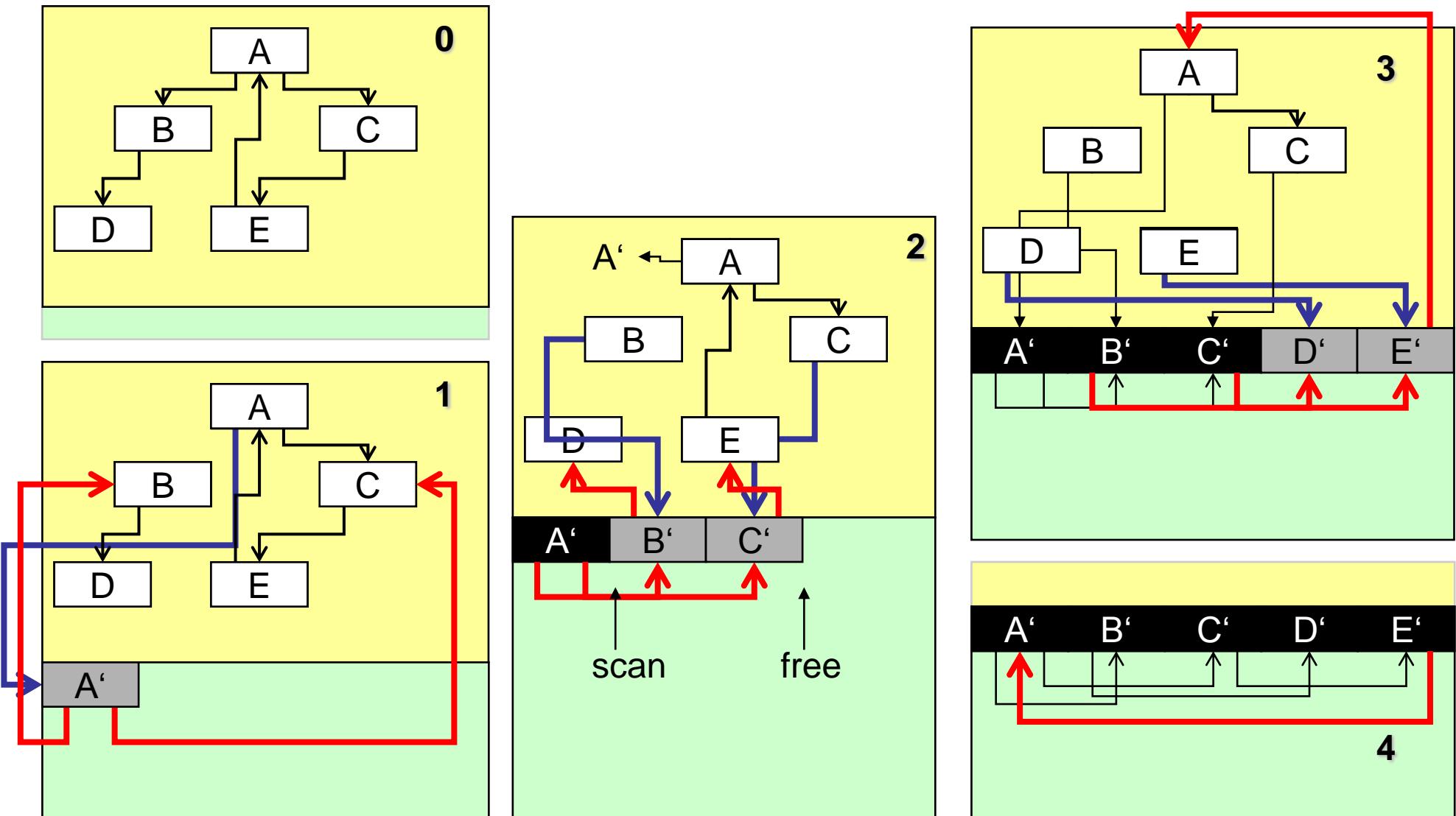


Beispiel: Inkrementeller kopierender Kollektor

- Einteilung des Heap in zwei Hälften „From-Space“ und „To-Space“
- Breadth-first Traversierung und Kopieren des Objektgraphen in den „To-Space“

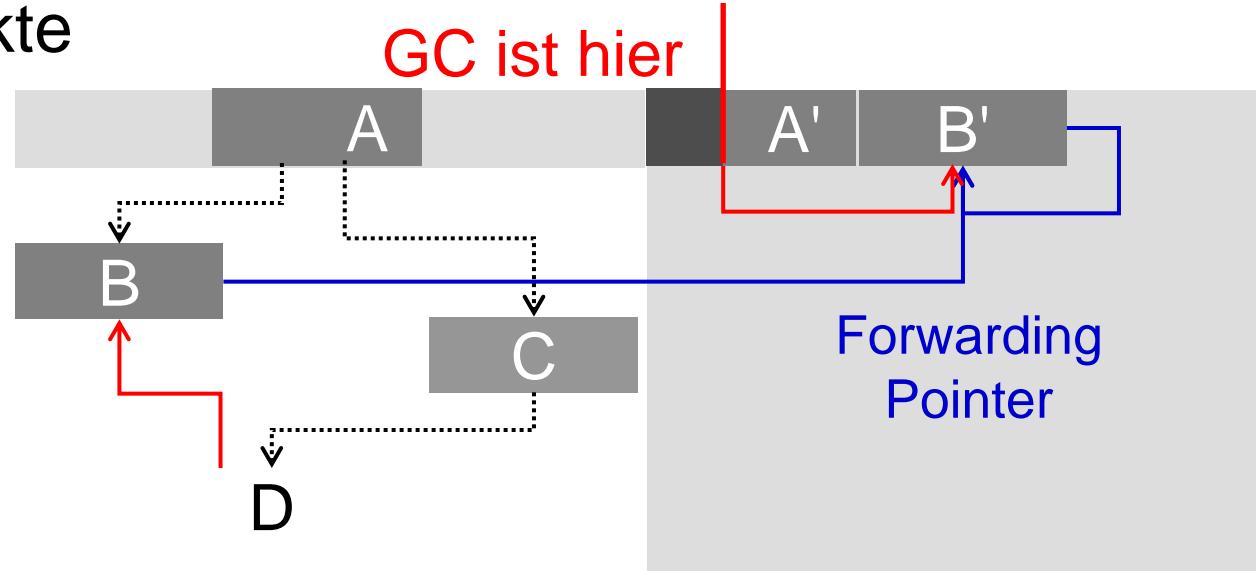


Kopierender Garbage Collector



Das Inkonsistenzproblem

- Zeiger sowohl zu B als auch zu B'
- Heilungsmöglichkeiten
 - Doppelte Indirektion via „Forwarding-Pointer“
 $D \rightarrow B \rightarrow B'$ bzw. $A' \rightarrow B' \rightarrow B'$
 - „Read Barriere“-Dämon bei Zugriffen auf weisse Objekte



Vor und Nachteile kopierender Kollektoren

- 😊 Geringe Allokationskosten
- 😊 Fragmentierung wird eliminiert
- 😊 Asymptotische Komplexität proportional zur Größe der aktiven Daten (vgl. Mark&Sweep)

- 😢 Verwendung zweier Hälften des Speichers
- 😢 Auch unter Verwendung von virtuellem Speicher ein Problem (Paging Kosten)

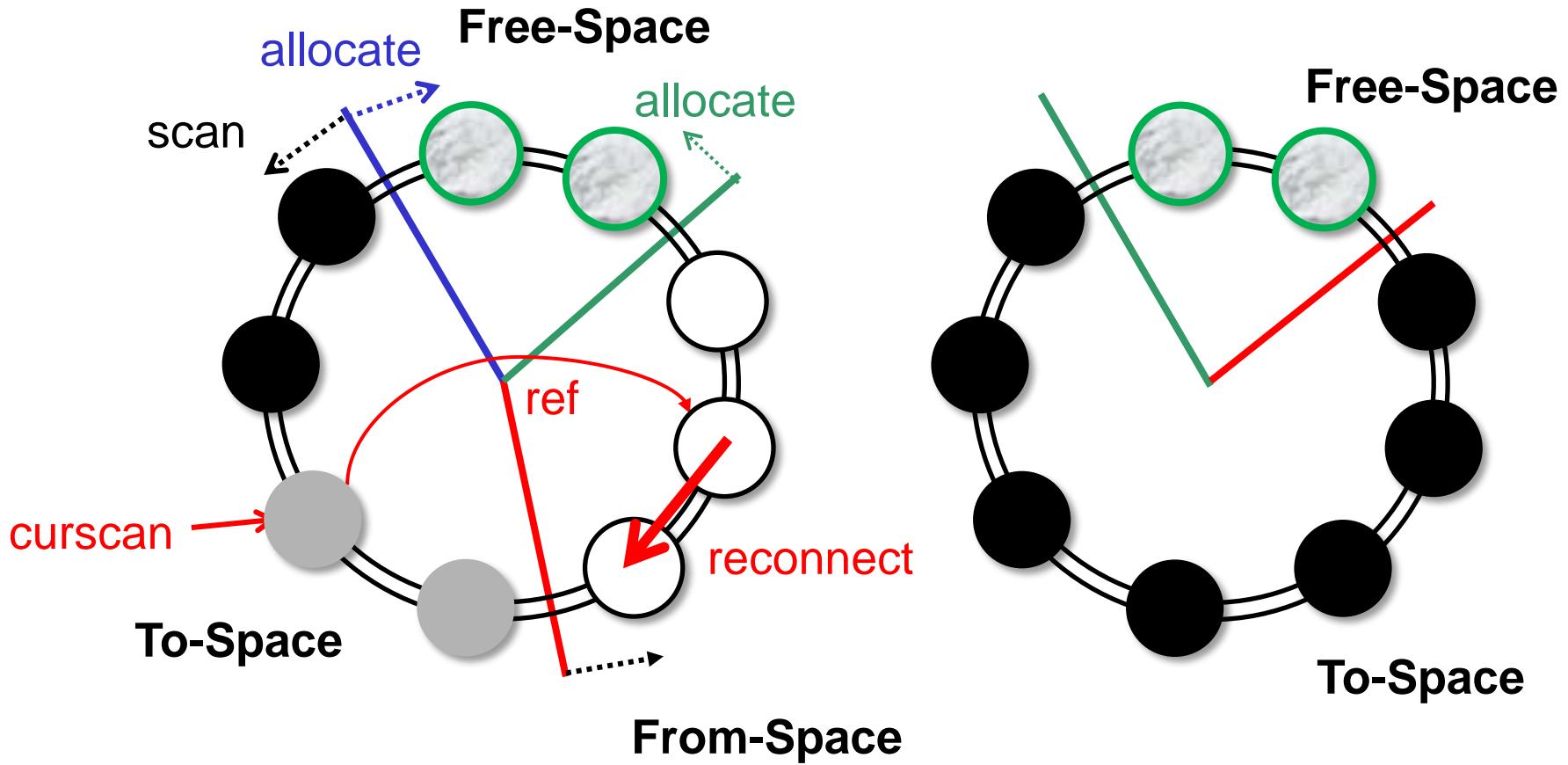
Beispiel: Baker'sche Treadmill* (1)

- Beobachtung: Der Heap zerfällt in 4 Mengen
 - besuchte, markierte Objekte
 - besuchte, unmarkierte Objekte
 - noch nicht besuchte Objekte
 - ▣ freier Platz
- B.T. organisiert diese Mengen in einem nicht verschiebendem Kollektor in eine zyklische, doppelt verkettete Liste
- Offensichtlicher Vorteil: kein Schutz des Mutators: keine Read-Barrieren oder Forwarding Pointer nötig

*Tretmühle

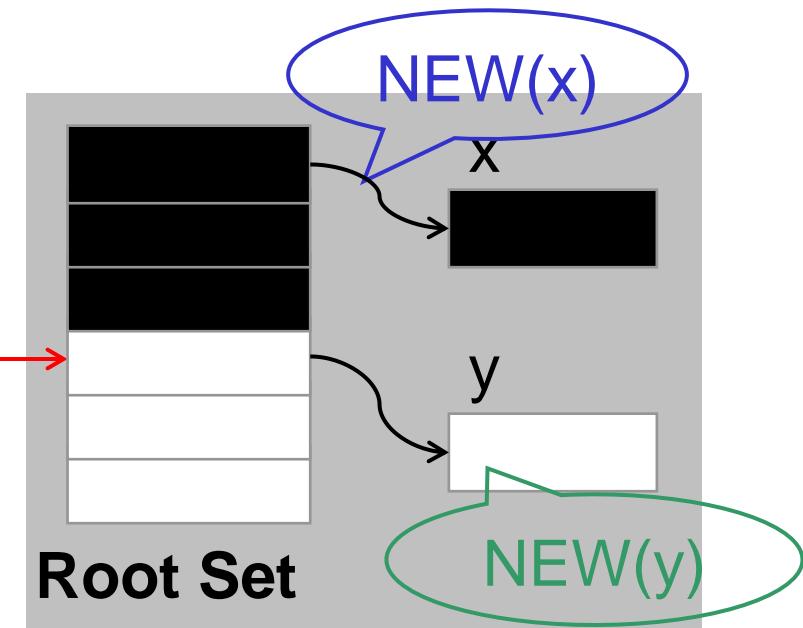
Beispiel: Baker'sche Treadmill (2)

- Simuliertes Kopieren
- Heap: doppelt verketteter Ring von Blöcken



Beispiel: Baker'sche Treadmill (3)

- Übergänge nach GC Zyklus
 - From-Space + Free-Space → Free-Space
 - ToSpace → FromSpace
- Fragmentierung
 - Extern: Nicht geheilt
 - Intern: Abhängig von den unterstützten Blockgrößen
- Neuallozierungen
 - Konservativ: Schwarzfärbung
 - Progressiv: Weissfärbung



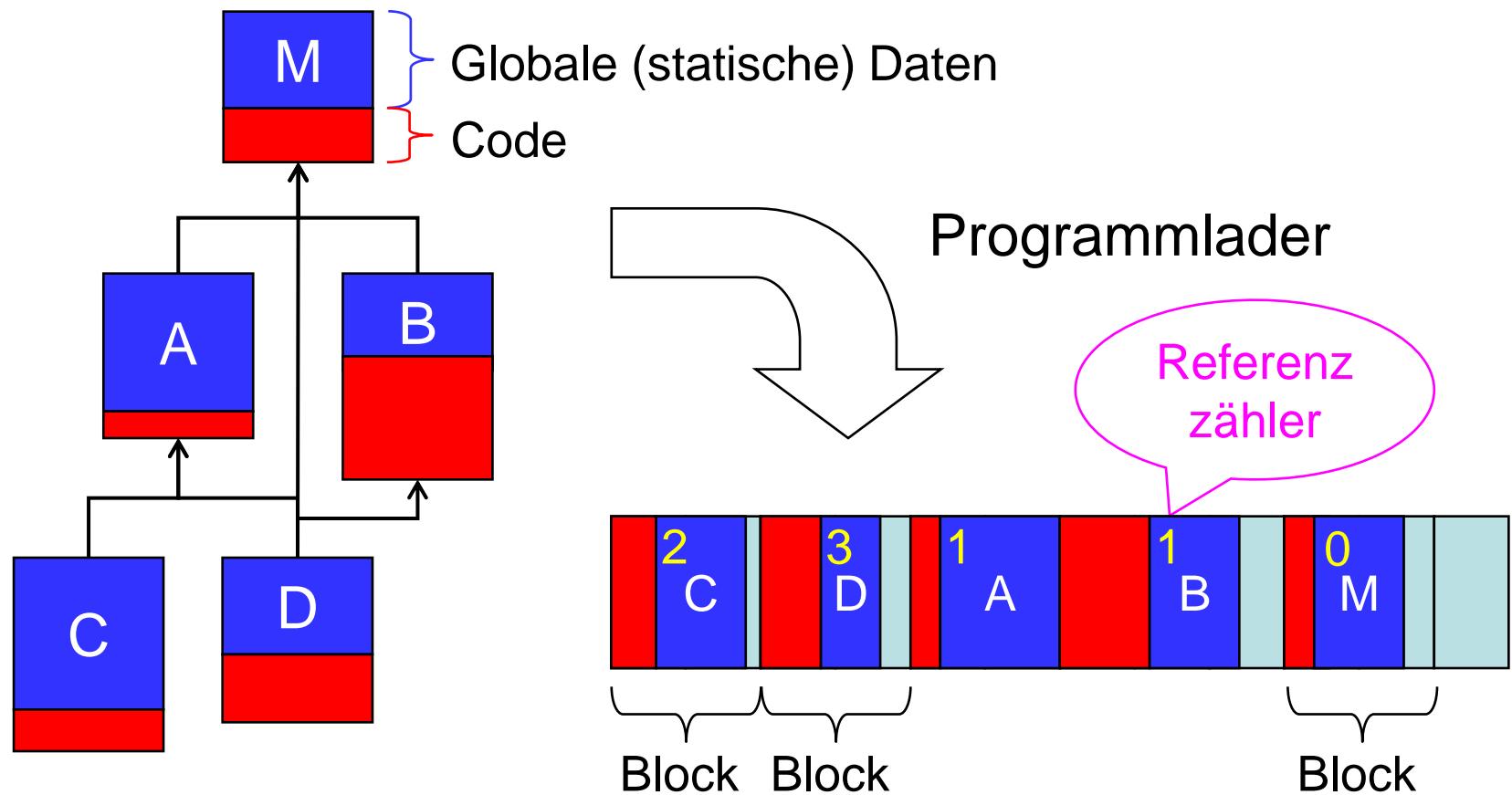
Beispiel 5: Selbstverwaltete Pools

- Programm verwendet transiente Instanzen einiger weniger Objekttypen
 - Beispiel: Simulation diskreter Ereignisse
- Programminterne Menge von Objektpools mit individuellen New-/Dispose-Methoden

```
T NewT () {  
    T t;  
    if (freeT == null) t = new T  
    else { t = freeT; freeT = freeT.next }  
    return t;  
};  
  
void DisposeT (T t) {  
    t.next = freeT; freeT = t  
}
```

4.1.3. Codebereichsverwaltung

- Linearisierte Modulhierarchie



4.1.4. Laufzeitunterstützung für OOP

- 4.1.4.1. Adressierungsschema
- 4.1.4.2. Methodentabellen
- 4.1.4.3. Interfacetabellen

Referenz- und Wertesemantik

- Wertesemantik: Dinge werden Werte-orientiert repräsentiert.

Java

nur Basistypen
(int, float, char,
bool)

C#

Basistypen
„struct“

C/C++

Basistypen
„struct“
„class“
Arrays (statisch)

Active Oberon

Basistypen
„RECORD“
„ARRAY“

„Inlined“ im Stack
oder Objekt

- Referenzsemantik: Dinge werden über Referenzen (Pointer) zugegriffen.

Immer auf dem
Heap (außer bei
Optimierungen)

Java

alles außer
Basistypen
(Arrays, Objekte)

C#

„class“
Arrays

C/C++

T* p
(Referenzsemantik
möglich für alles)

Active Oberon

„POINTER TO RECORD“
„OBJECT“
„POINTER TO ARRAY“

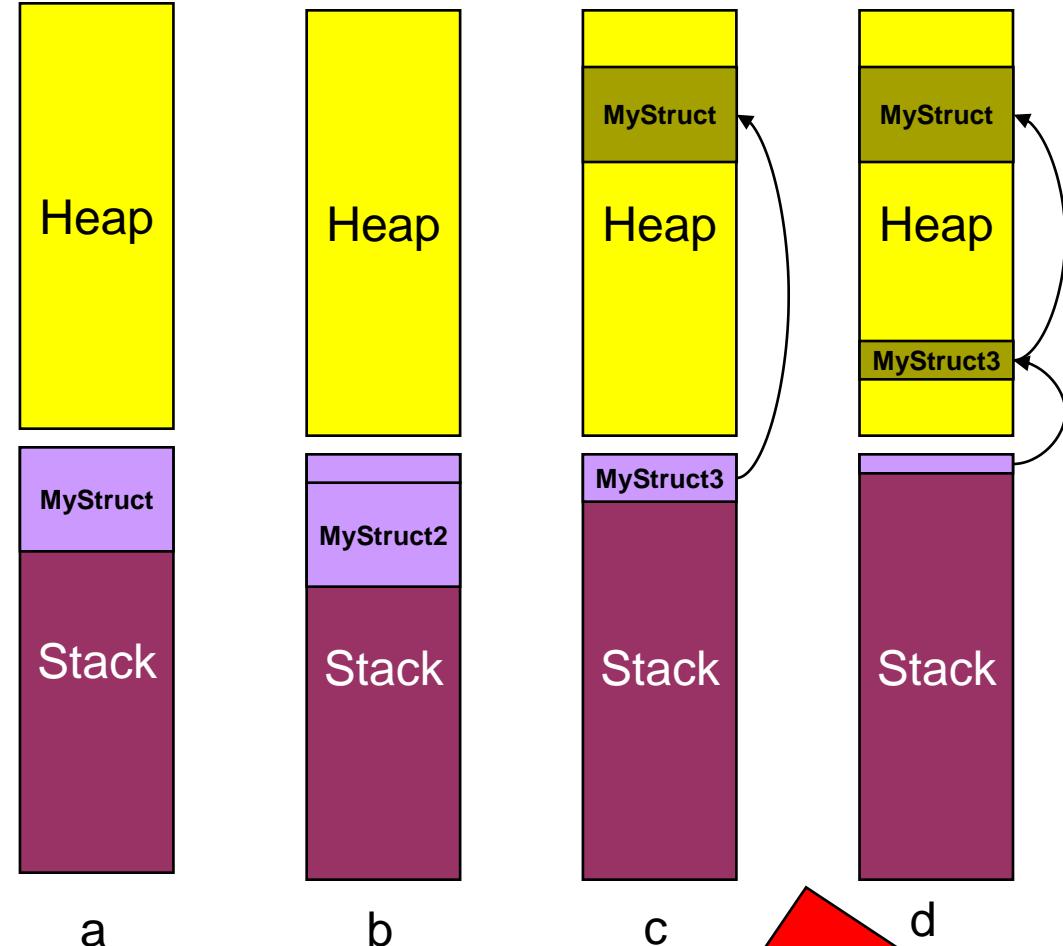
Allokation: Referenz- versus Wertesemantik

```
MyStruct= record
  first: real; a,b: integer;
end;

MyStruct2= record
  a: MyStruct;
  b: real;
end;

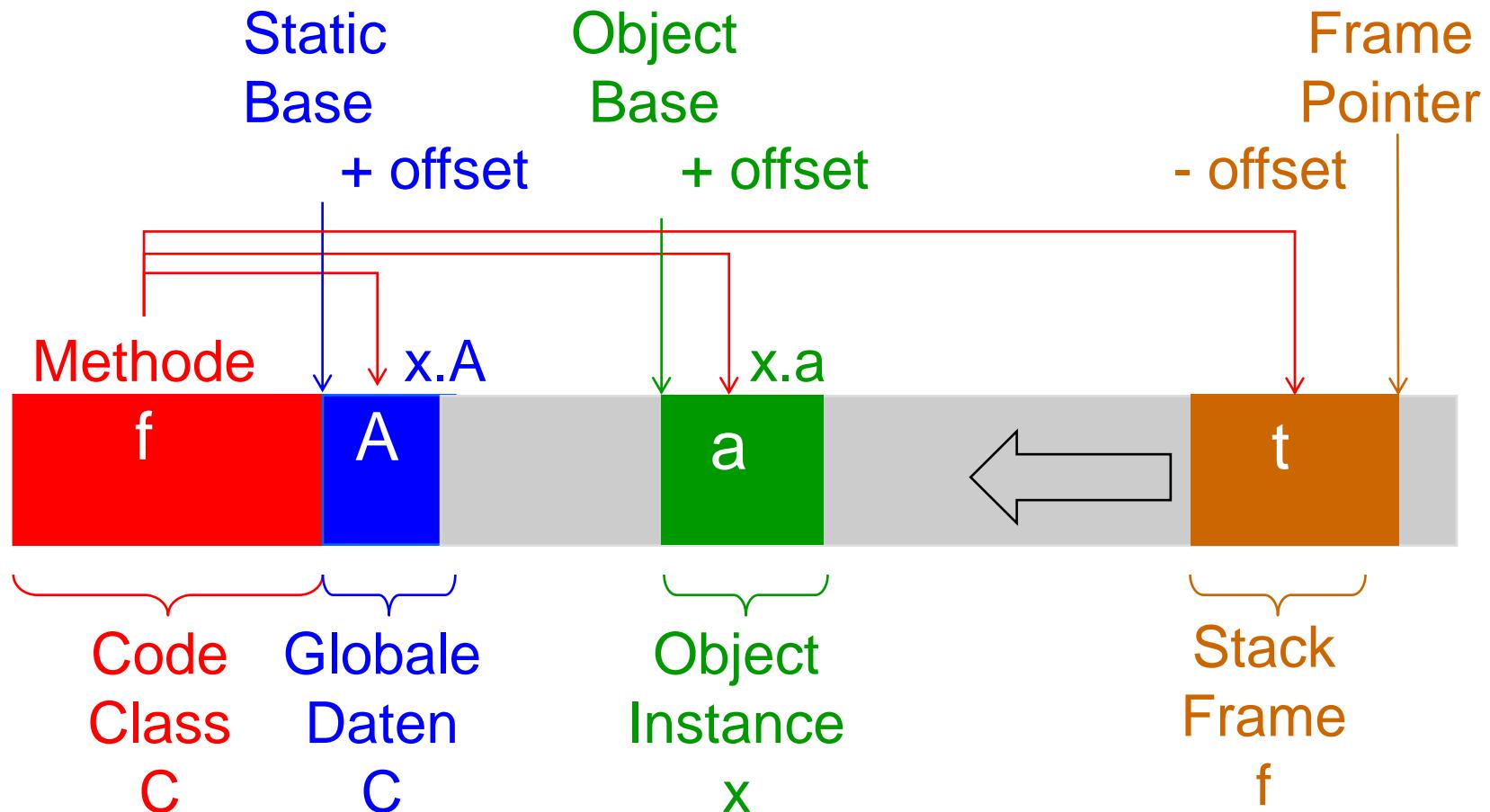
MyStruct3= record
  a: pointer to MyStruct;
  b: real;
end;

var
  a: MyStruct;
  b: MyStruct2;
  c: MyStruct3;
  d: pointer to MyStruct3;
```



Für c und d ist zusätzlich auch noch ein Typdeskriptor notwendig ! (Hier weggelassen).

4.1.4.1. Adressierungsschema



4.1.4.2. Methodentabellen

■ Virtuelle Methoden (in C#)

```
class C0
{
    void e (...) { ... }
    virtual void f (...) { ... }
    virtual void g (...) { ... }
}
```

statische Methode,
Adresse zur Compilezeit
bekannt

```
class C1: C0
{
    override void f (...) { ... }
    void h (...) { ... }
}
```

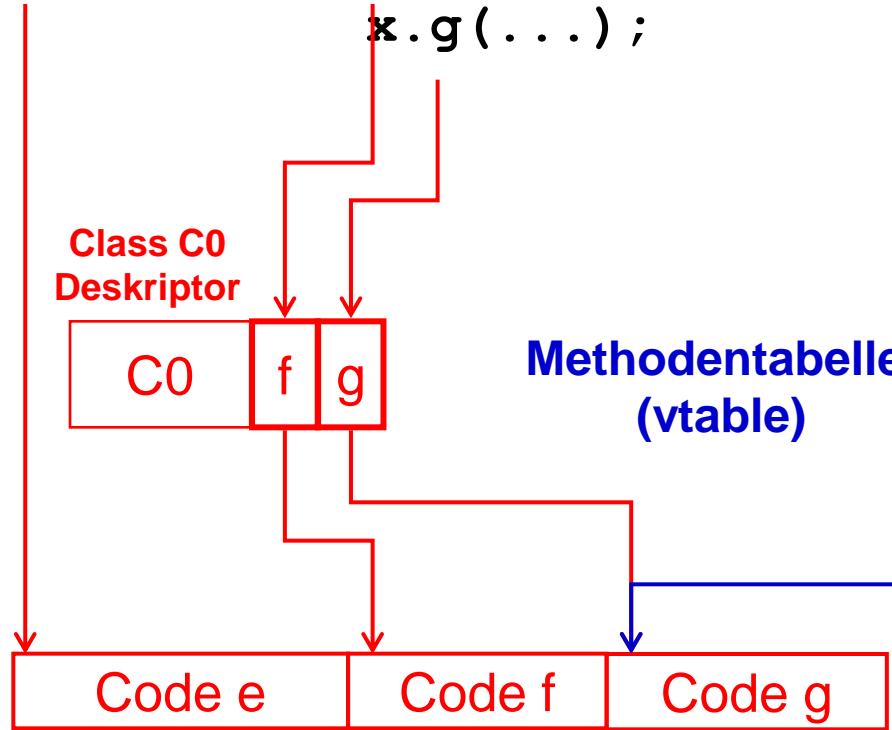
überladbare Methode
Adresse erst zur Laufzeit
bekannt

Überladen der Methode
C0.f

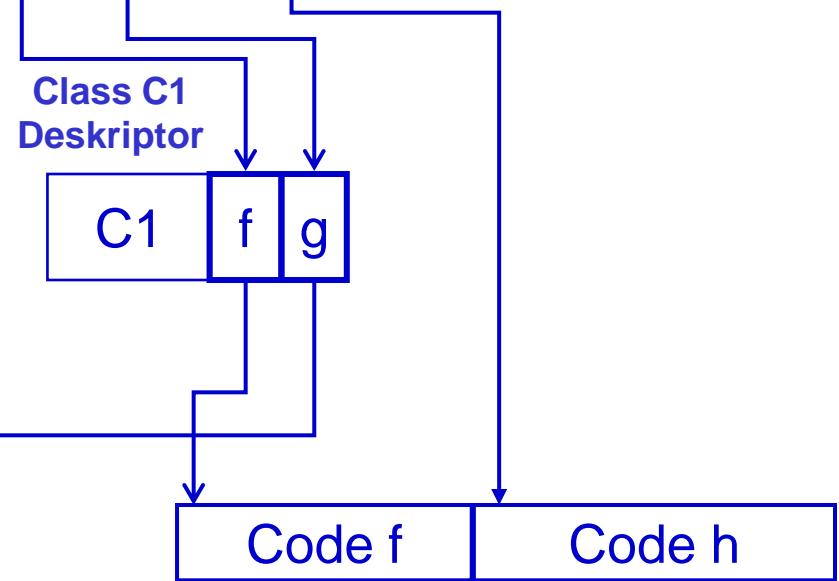
statische Methode

Laufzeitstruktur des CTS* in .NET

```
C0 x; C1 y;  
x = new C0 (...);  
x.e(...); x.f(...);  
x.g(...);
```



```
y = new C1 (...);  
x = y;  
x.f(...);  
x.g(...);  
(C1)x.h(...);
```



*Common Type System

4.1.4.3. Interfacetabellen

- Interfaces (in C#)

interface X { ... void f(); ... }

...

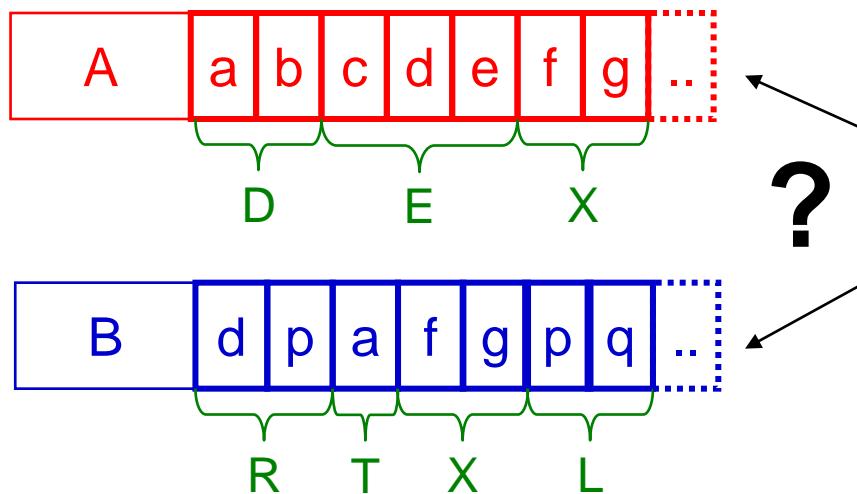
class A: D, E, X { ... } // interfaces

class B: R, T, X, L { ... }

A implementiert
D,E und X

B implementiert
R,T, X und L

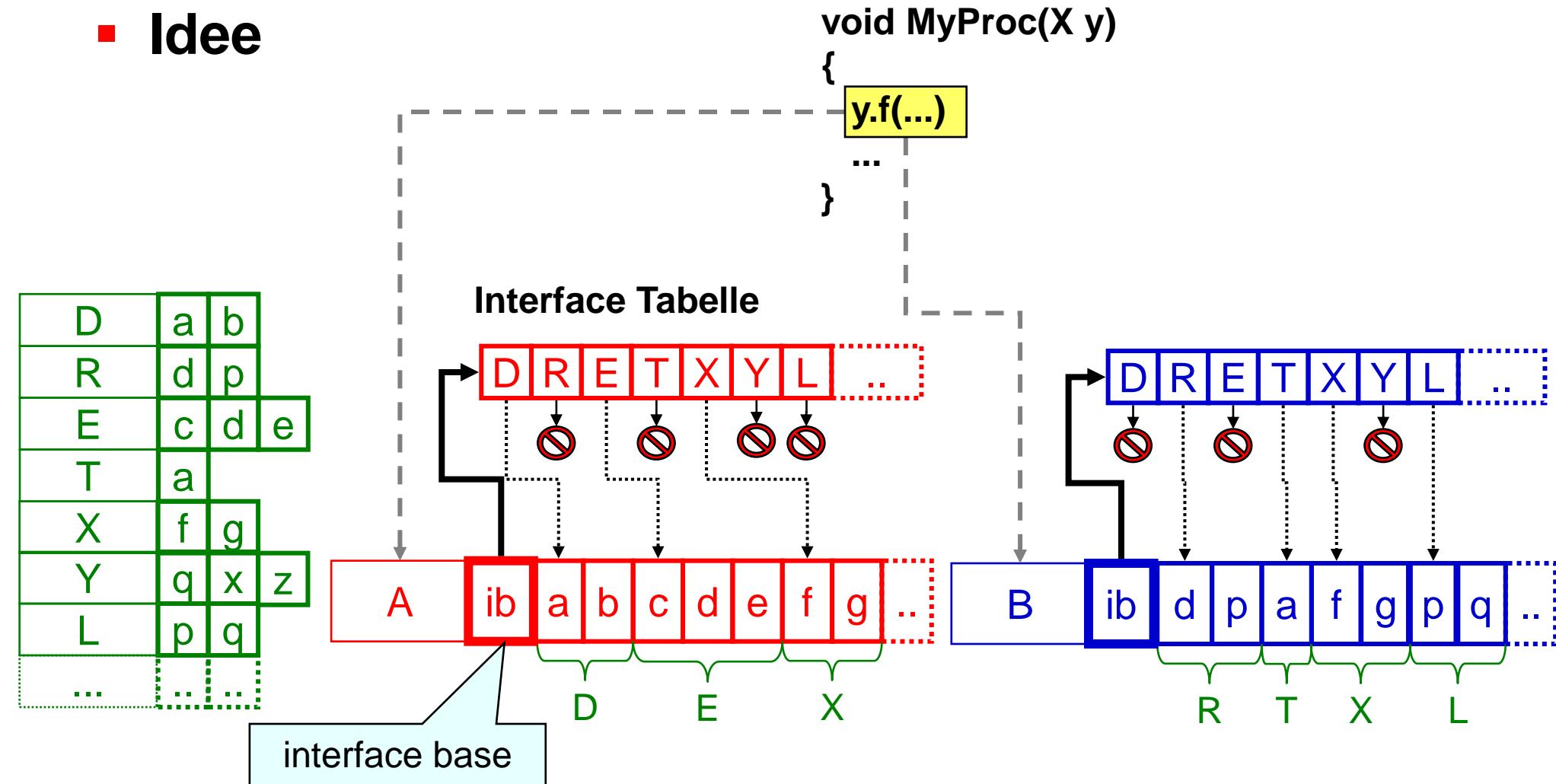
D	a	b	
R	d	p	
E	c	d	e
T	a		
X	f	g	
Y	q	x	z
L	p	q	
...			



void MyProc(X y)
{
 y.f(...)
...
}

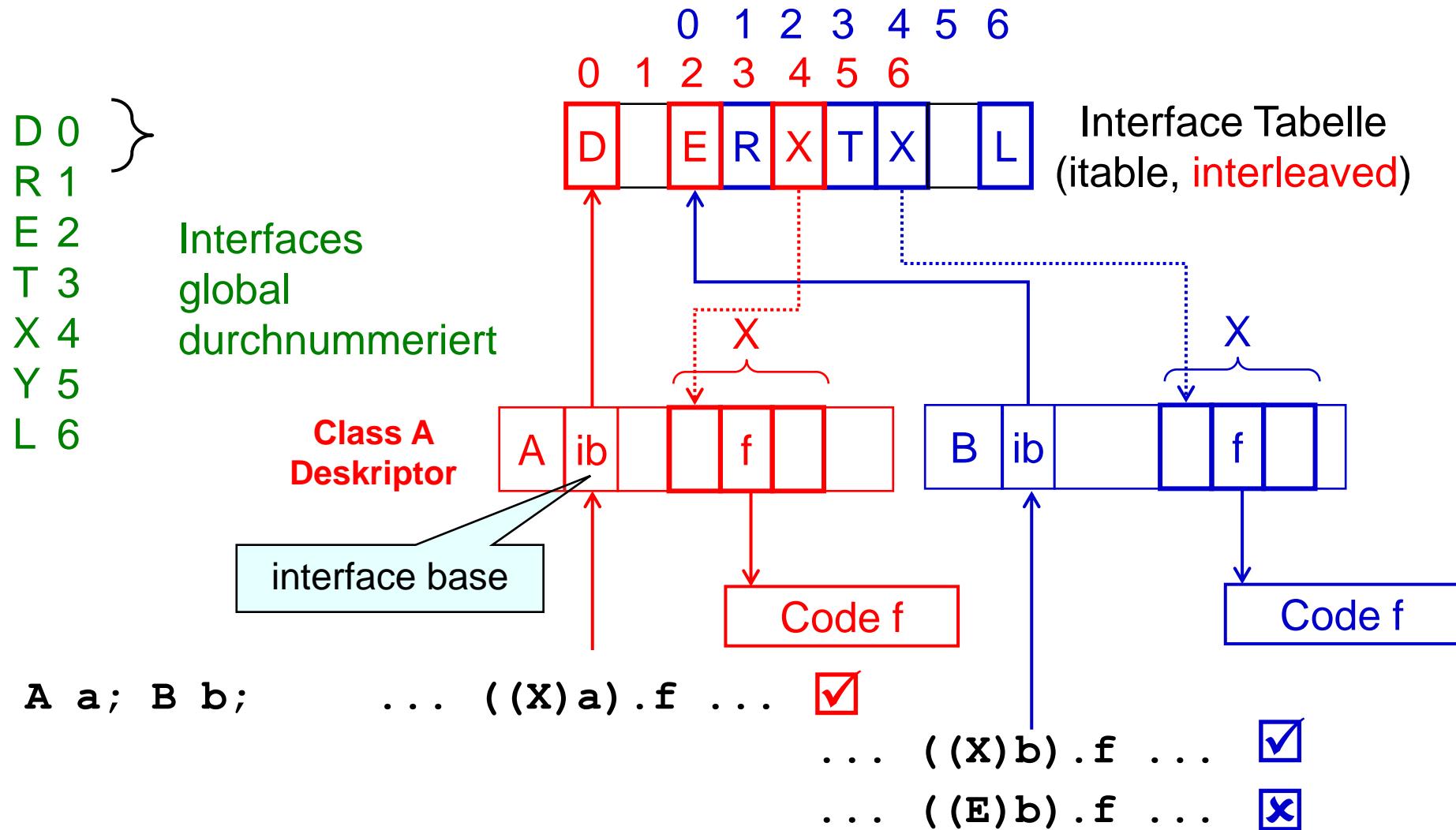
Interfacetabellen (II)

■ Idee

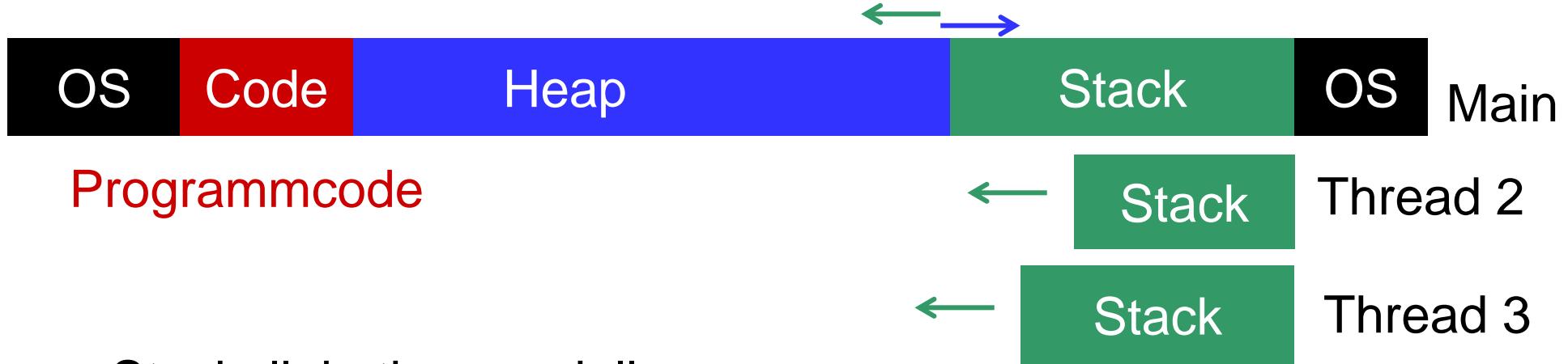


- zu viele große Interface-Tabellen, so nicht praktikabel

Laufzeitstruktur des CTS in .NET



4.1.5. Multi Thread* Speicherlayout

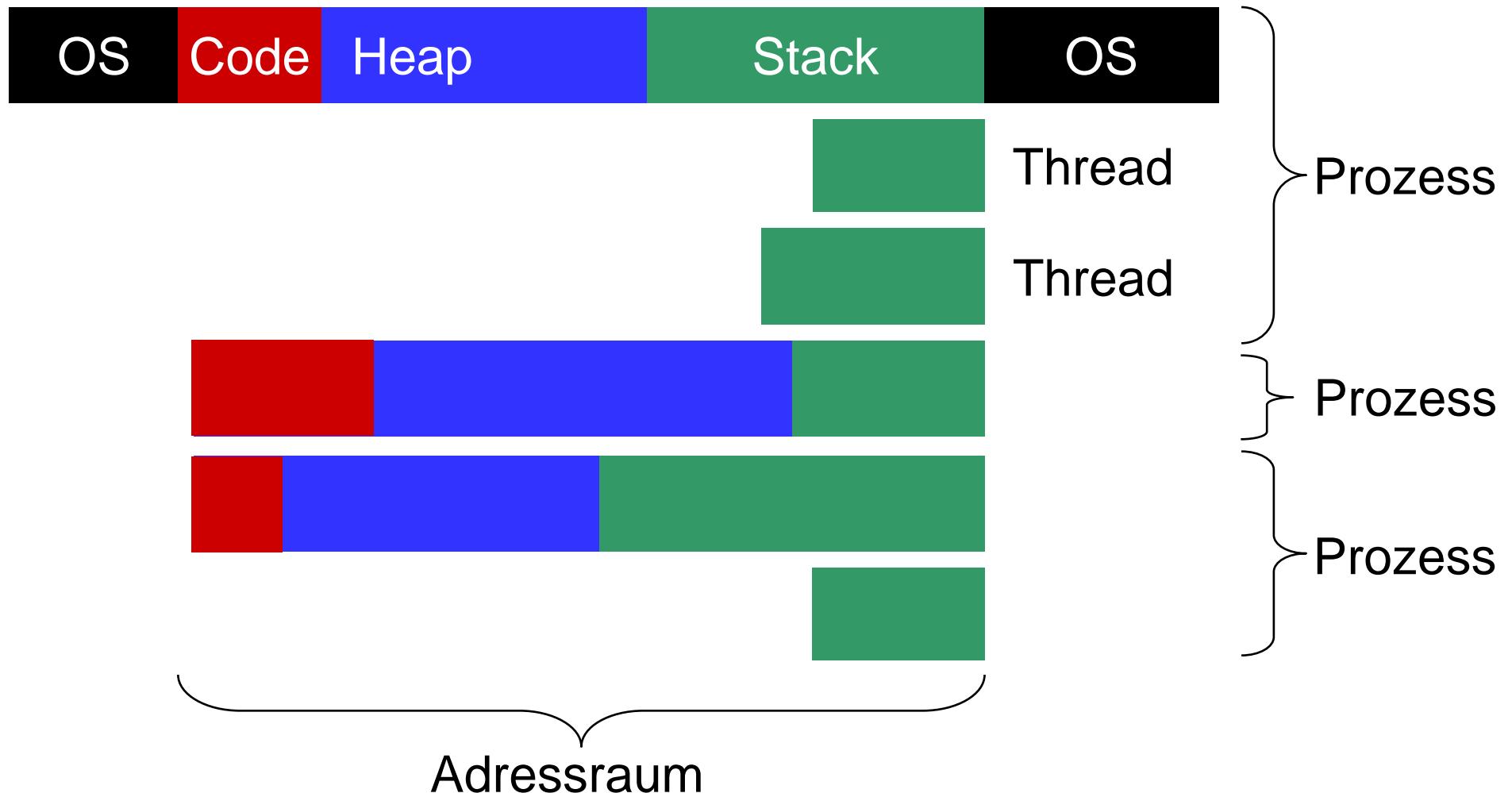


- Stackallokationsmodelle
 - Stackframes im Heap
 - Microstacks (unzusammenhängend im Heap)
 - Virtuelle Adressierung
 - Separater Adressraum pro Stack

*thread=Faden, hier: Leichtgewichtsprozess

4.2. Multi Prozess Speicherlayout

Tanenbaum, Kapitel 4 / Silberschatz Teil 3



Anforderungen an die Speicherverwaltung

- Relokation
 - bei Allokation und
 - bei Aus- und Einlagerung (Swapping / Demand Paging)
- Schutz
 - vor Zugriff fremder Prozesse auf private Datenebereiche eines Prozesses.
 - muss vom Prozessor unterstützt sein
- Gemeinsame Nutzung
 - Zugriff auf gemeinsame Daten zwischen den Prozessen
- Logische Organisation
 - Unterstützung für die Übersetzung von Modulen / Programmen
- Physikalische Organisation
 - Transfer zwischen Primär- und Sekundärspeicher

Partitionierung

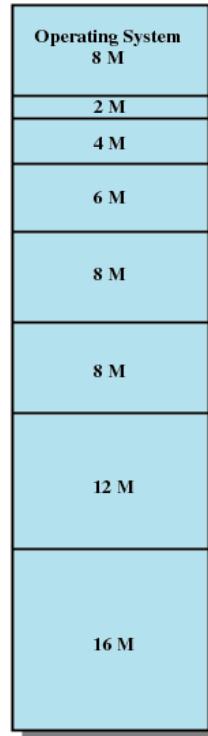
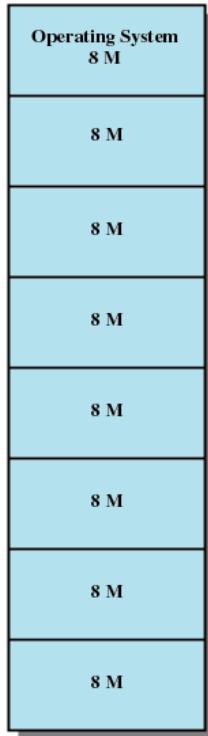
- Statische Partitionierung
- Dynamische Partitionierung
- Segmentierung
 - Prozesse auf mehrere Segmente aufgeteilt
 - Mit und ohne virtuellen Speicher
- Paging
 - Prozesse auf mehrere Seiten aufgeteilt
 - Mit und ohne virtuellen Speicher (demand paging)

4.2.1. Partitionierung

Statische Partitionierung

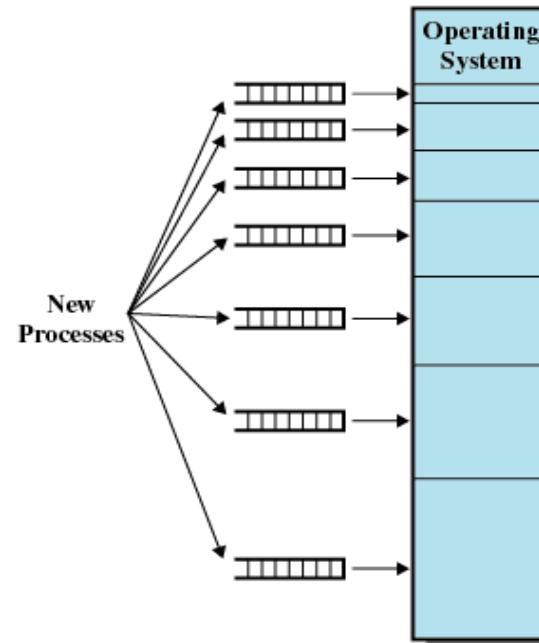


- Einteilung des Speichers in fixierte Partitionen gleicher oder ungleicher Größe.

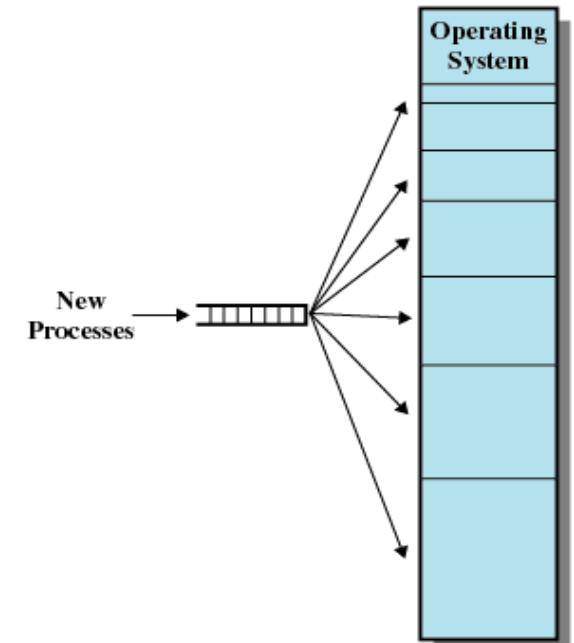


(a) Equal-size partitions

(b) Unequal-size partitions



(a) One process queue per partition



(b) Single queue

Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

Figure 7.3 Memory Assignment for Fixed Partitioning

Partitionierung

Dynamische Partitionierung



- Allokationsverwaltung im Zusammenhang mit Einprozess-Speicherlayout im wesentlichen behandelt.
- Stichworte: Free-Liste, Best-Fit, FirstFit, Buddy System etc.

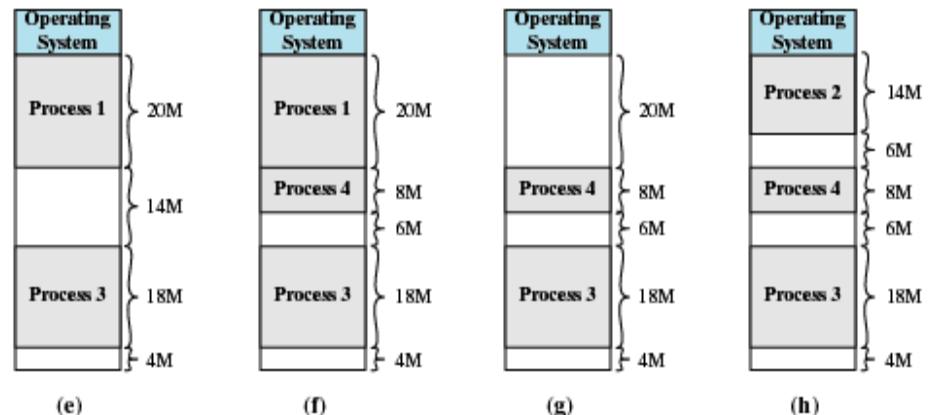
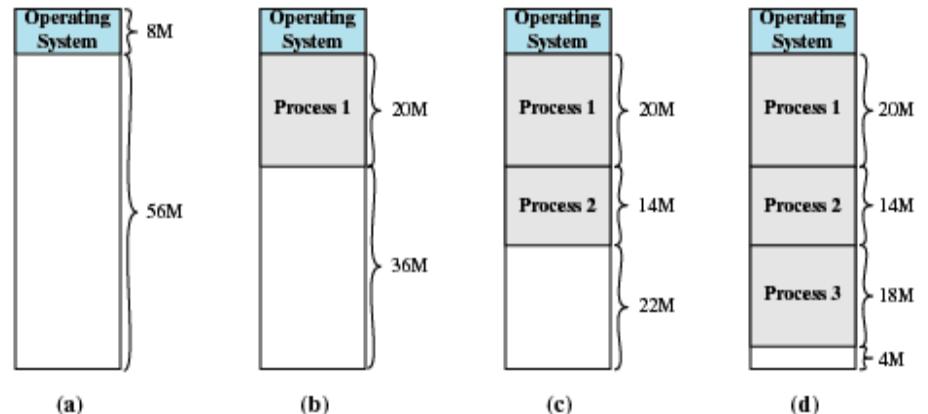
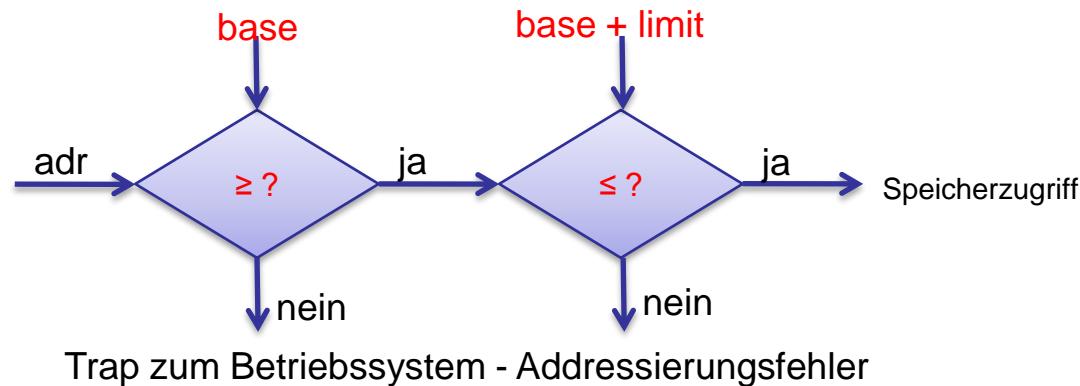


Figure 7.4 The Effect of Dynamic Partitioning

(Einfache) Hardware Unterstützung

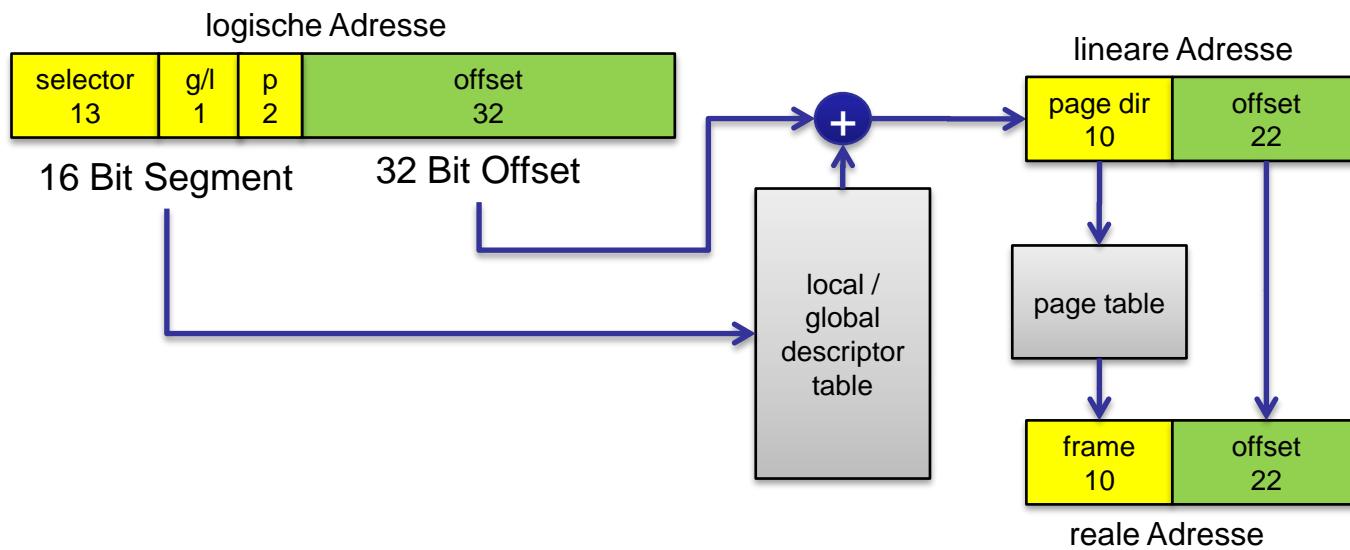
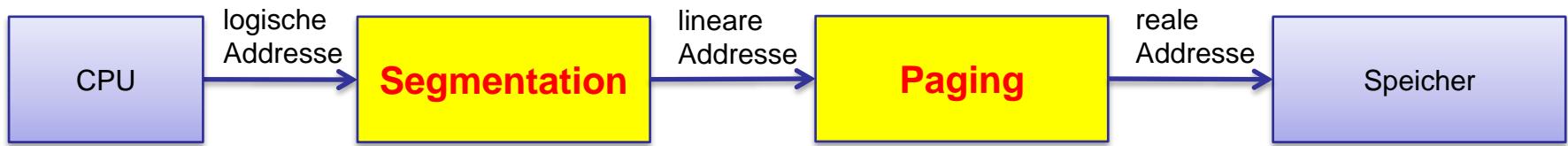
- Basis- und Grenzregister in der CPU



- Relocationsregister in der MMU

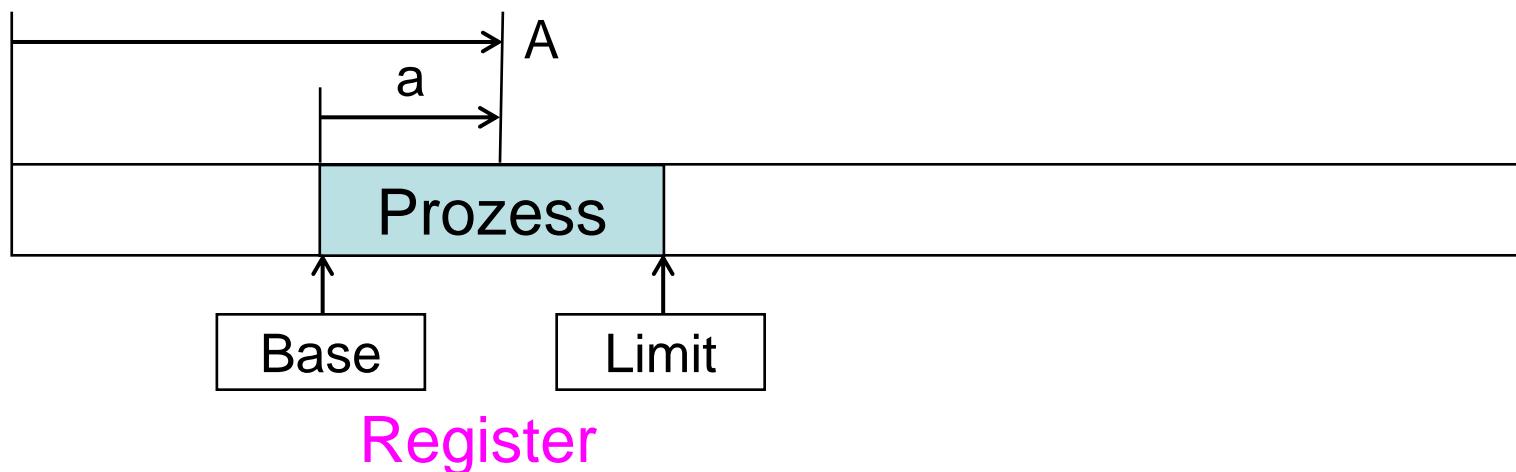


Intel Pentium Hardware (32bit)



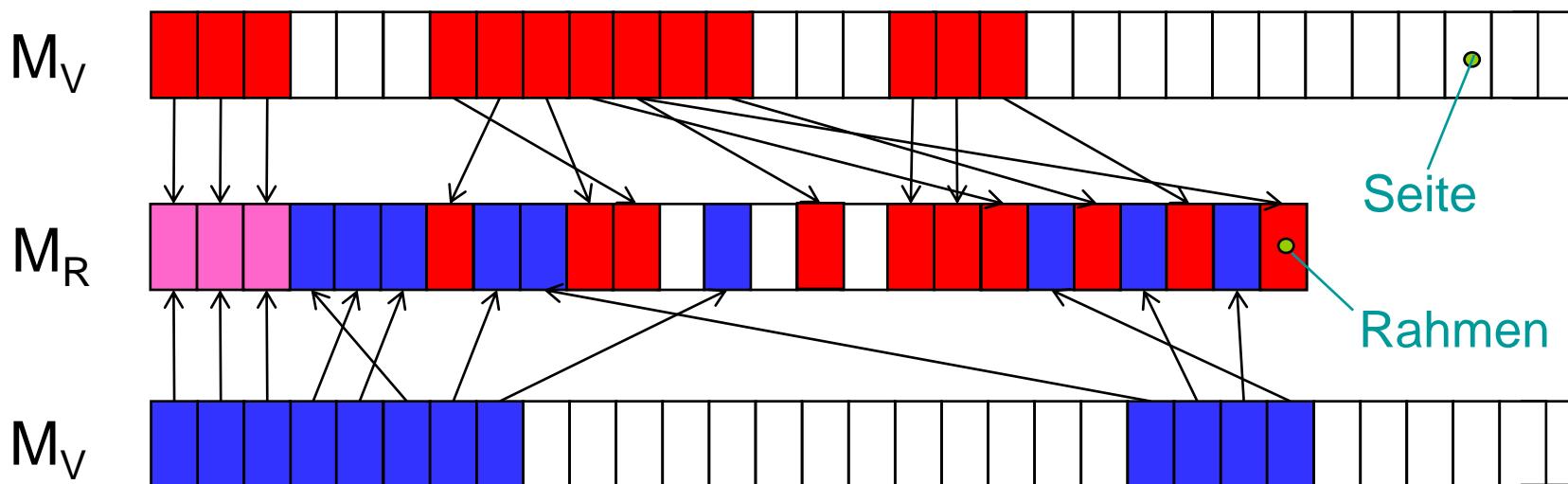
4.2.2. Segmentierung

- Bietet Reloziertbarkeit durch relative Adressierung
- Implementiert Schutzmechanismus
- Erfordert Laufzeitadresstranslation

$$a \rightarrow \{ A = \text{Base} + a; \text{ if } (A > \text{Limit}) \text{ error; } \}$$


4.2.3. Paging

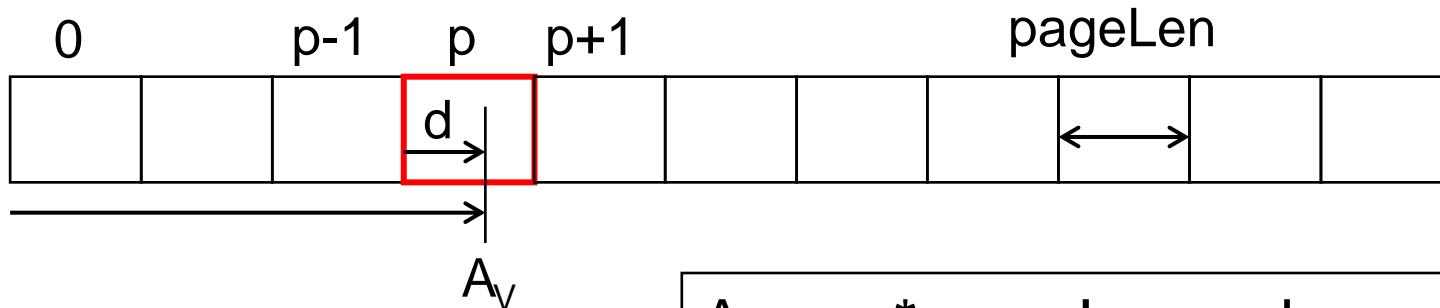
- Realer Speicher M_R als array max of **Frame**
- Virtuelle Speicher M_V als array of **Page**
- Abbildung $M_V \rightarrow M_R$



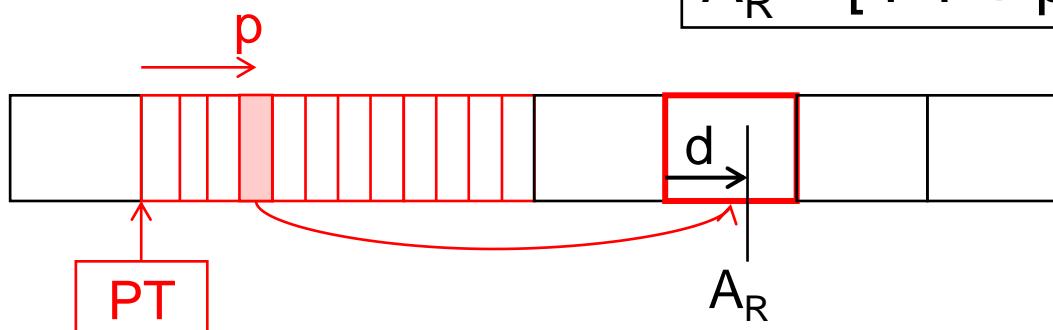
Paging

Adresstranslation (1)

Virtueller Adressraum



Realer Adressraum



Page Table

$$A_V = p * \text{pageLen} + d$$

$$A_R = [PT + p^*4] * \text{pageLen} + d$$

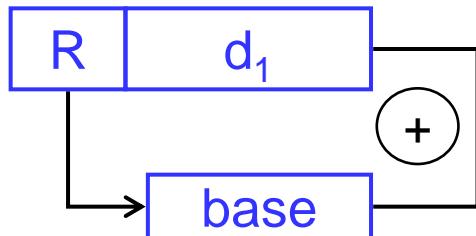
[$PT+x$]: Inhalt
der Pagetable
an Stelle x

kann sehr groß
werden

Paging

Adresstranslation (2)

Instruktionsoperand



Basisregister

Virtuelle Adresse



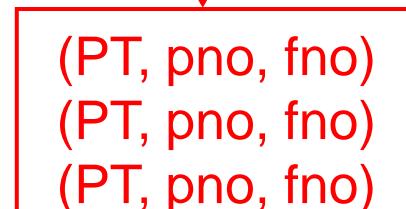
Reale Adresse



A_V

A_R

Translation
Lookaside
Buffer
(Cache)



Memory
Management
Unit (MMU)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R Y	45

typisch:
Größe: 8-4096 Einträge
Zugriffszeit (hit): 0.5 - 1 Taktzyklus
Penalty (miss): 10 - 30 Taktzyklen
Miss rate: 0.01% - 1%

Seitentabelle

PT

page-no

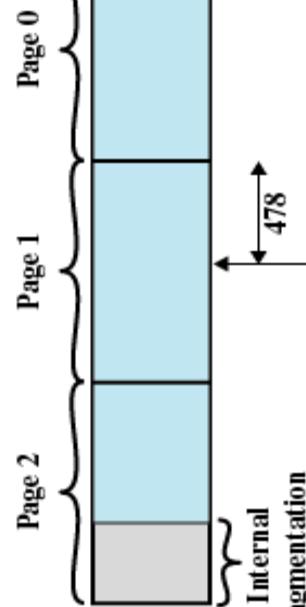
Logische Adressen

Relative address = 1502
0000010111011110

User process
(2700 bytes)

(a) Partitioning

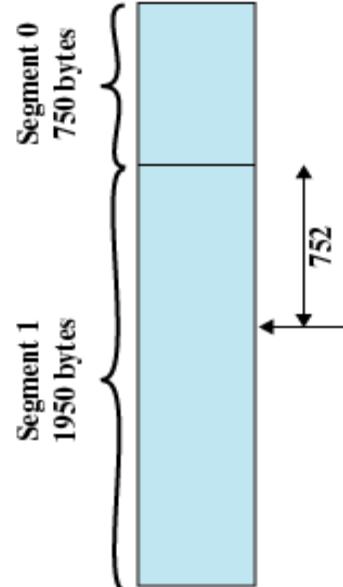
Logical address =
Page# = 1, Offset = 478
0000010111011110



(b) Paging
(page size = 1K)

Partitionierung

Logical address =
Segment# = 1, Offset = 752
0001001011110000

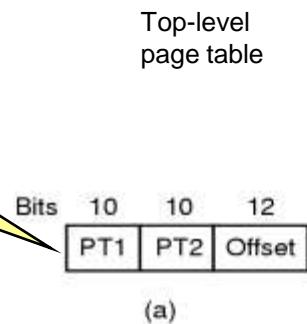


(c) Segmentation

Segmentierung

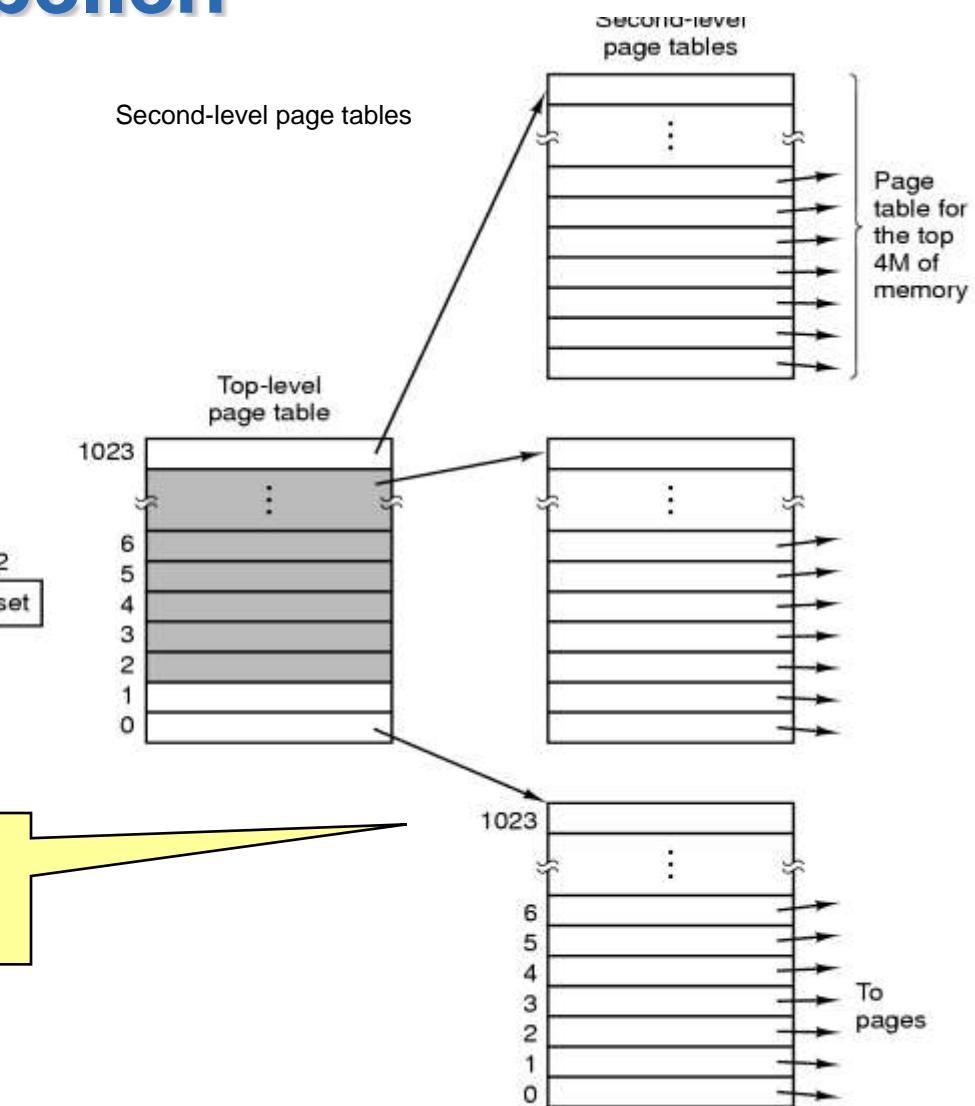
Mehrstufige Seitentabellen

32 bit Adresse mit
zwei 10-Bit Feldern
für Seitennummern



Zweistufige
Seitentabelle

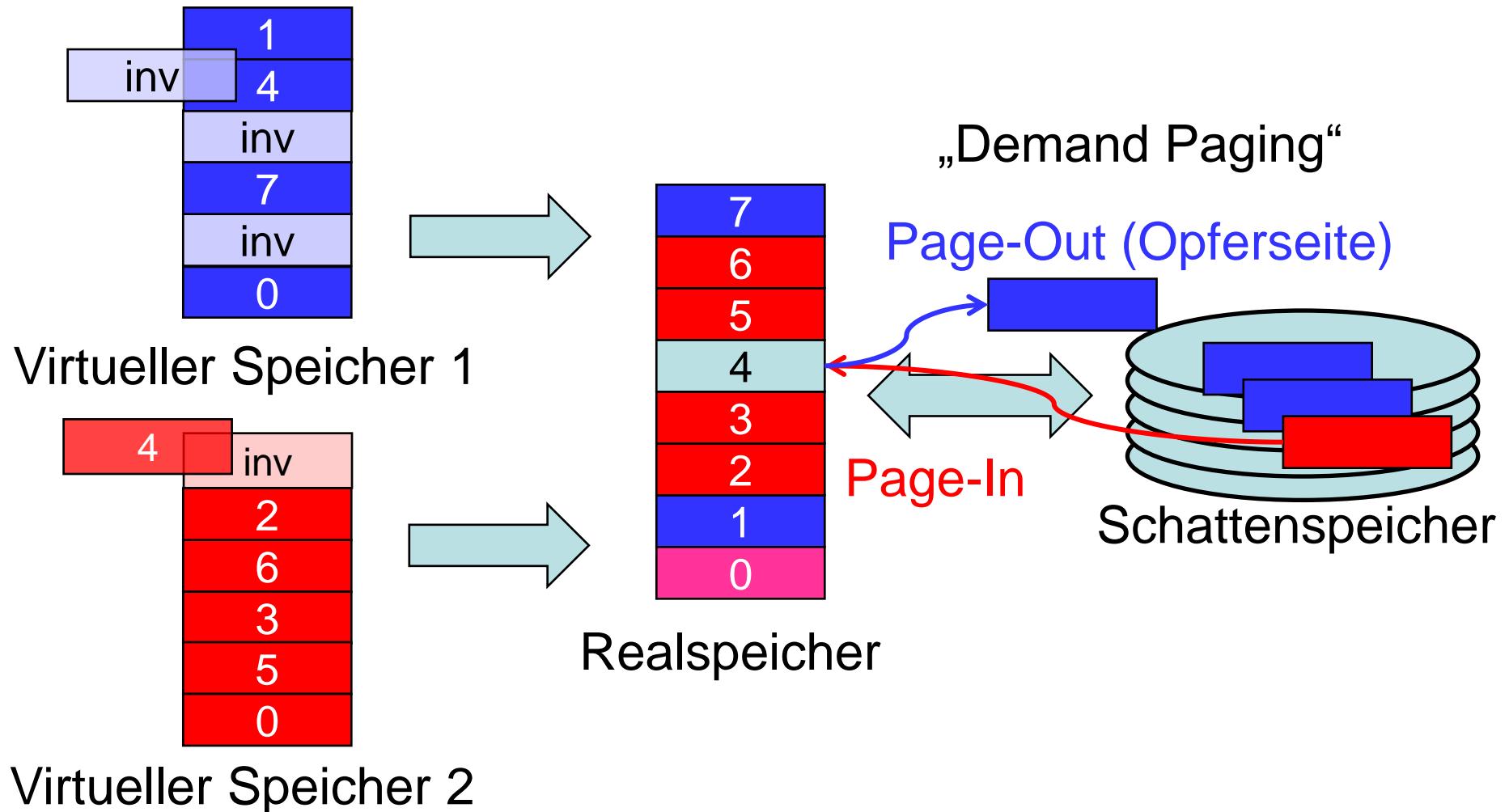
Weitere Lösung:
Inverse Tabellen (hashing)



4.2.4. Virtueller Speicher (Demand Paging)

- Illusion des unendlich grossen Speichers
- Verwendung von „**Schattenspeicher**“ (Disk) zur Speicherung derzeit nicht gebrauchter Seiten
- Unterstützung durch „Flags“ in den Seitentabellen
 - ***invalid*** \Leftrightarrow Seite derzeit ausgelagert
 - ***referenced*** \Leftrightarrow Seite seit Einlagerung referenziert
 - ***modified*** \Leftrightarrow Seite seit Einlagerung modifiziert
- Zugriff auf ungültige Seite führt zu ***Page-Fault*** Interrupt und Betriebssystemaufruf
 - Suchen einer „Opferseite“ bei Realspeicherknappheit
 - Verwaltung der ausgelagerten Seiten

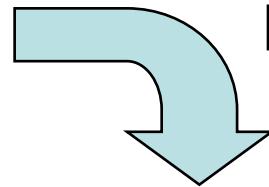
Aus- und Einlagerung



Ablauf Adressübersetzung $A_V \rightarrow A_R$

```
if ( $A_V$  in TLB) return  $A_R$ 
else {
    // MMU involved
    access page table;
    if (page invalid) page-fault
    else return  $A_R$ 
}
```

Hardware



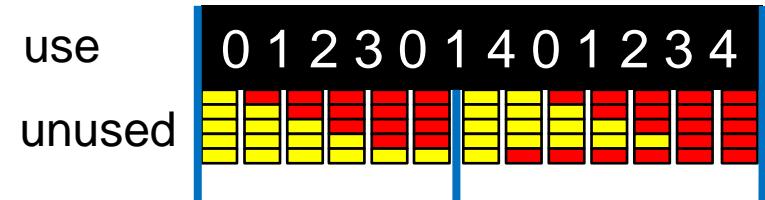
Page-Fault Interrupt

Software

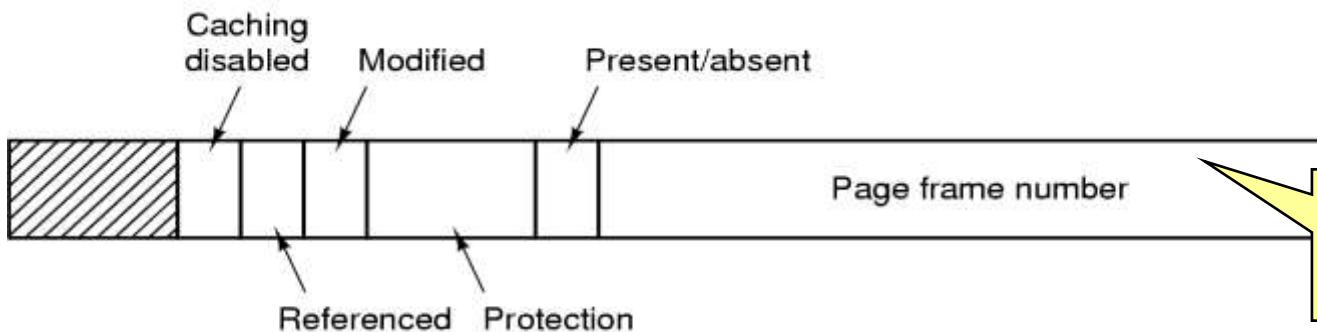
```
if (free page frame exists) assign it to  $A_V$ 
else {
    search victim page;
    if (victim page modified) {
        page-out to secondary storage;
        invalidate victim page;
        assign corresponding frame to  $A_V$ 
    }
}
page-in from secondary storage;
reset invalid flag
```

Seitenersetzungsalgorithmen (1)

- Ideale Strategie (Longest Unused)
 - Verdränge die **zukünftig** am längsten nicht (oder nie) mehr benutzte Seite
 - Benötigt "Orakel"
- "Not Recently Used" NRU
 - Setze bei jedem Takschlag das Referenzflag aller Seiten zurück
 - Bilde Präferenzkategorien (z.B. {ref,mod} < {ref} < {mod} < {})
 - Wähle eine der Seiten höchster Präferenz
 - Voraussetzung:
 - Hardware setzt die Flags oder
 - Software (durch Erzwingen von Page-Faults).



	pref	ref	mod
3	0	0	0
2	0	1	0
1	1	0	0



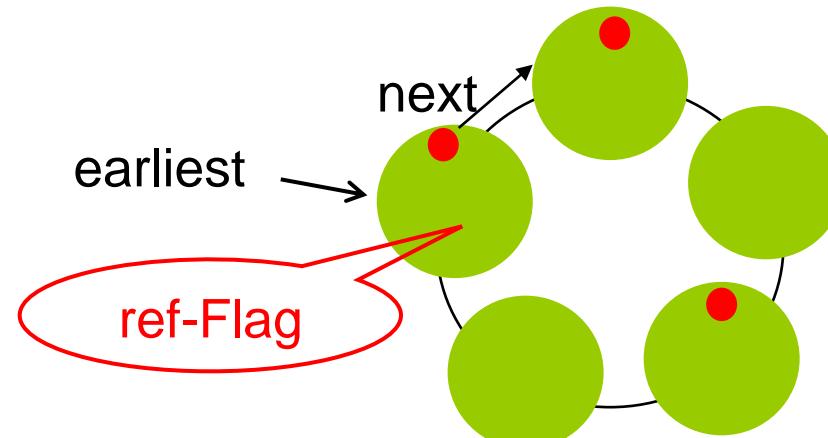
typischer
Seitentabelleneintrag

Seitenersetzung (2)

- “Least Recently Created” LRC (FIFO)
 - Lebensdauer als Kriterium
 - Verkettung nach “Geburtszeit”
 - Schlechte Behandlung permanent benutzter Seiten
 - Modifikation: “Second Chance”, falls ref-Flag seit letztem Taktschlag gesetzt wurde
 - => “Clock Algorithmus”

use	0 1 2 3 0 1 4 0 1 2 3 4
fifo Q	0 1 2 3 0 1 4 4 4 2 3 3
	0 1 2 3 0 1 1 1 4 2 2
	0 1 2 3 0 0 0 1 4 4
	0 1 2 3 0 1

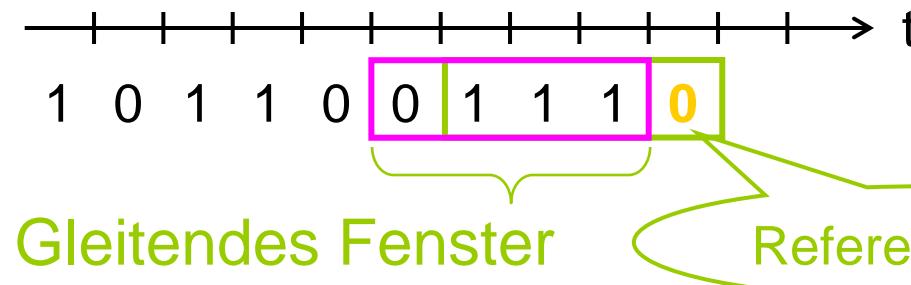
```
cur = earliest;  
while (cur.ref) {  
    cur.ref = false;  
    cur = cur.next  
}
```



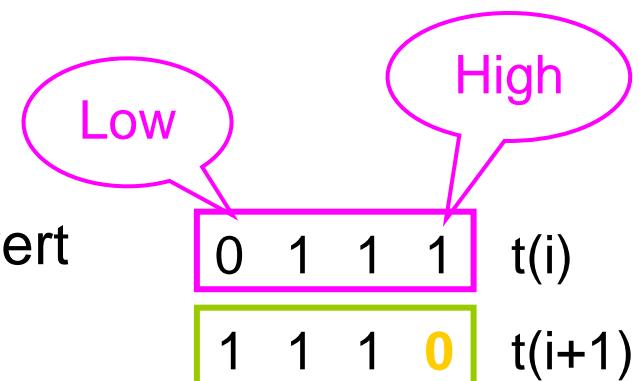
Seitenersetzung (3)

- “Least Recently Used” LRU

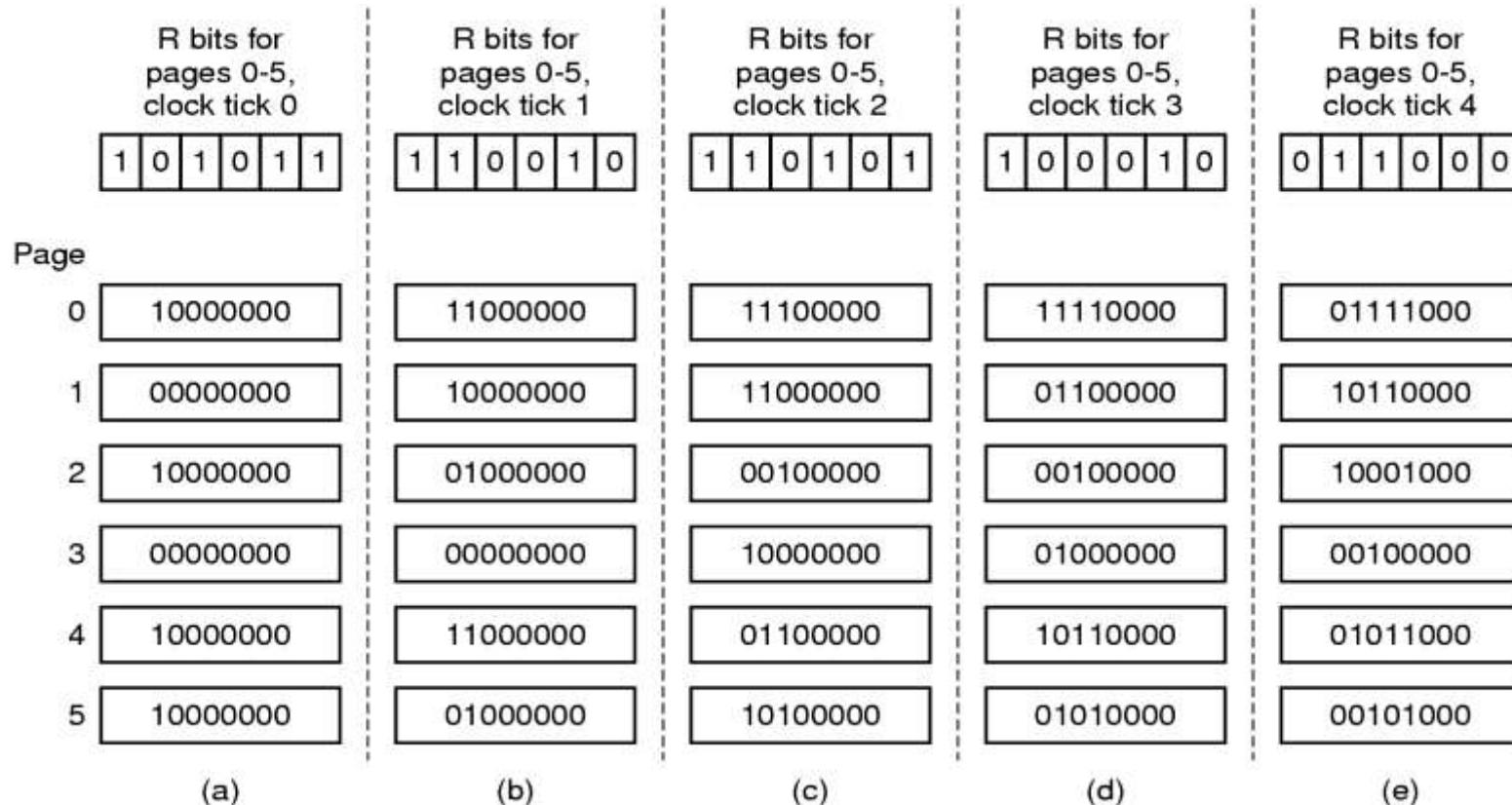
- Entferne Seite, die am längsten nicht genutzt wurde
- Hardwareimplementation
 - 64-Bit Zeitmarke für jede Seite
- Softwareimplementation
 - “Aging”-Algorithmus
 - Wähle Seite mit kleinstem Zählerwert



use	0 1 2 3 0 1 4 0 1 2 3 4
fifo Q	0 1 2 3 0 1 4 4 4 2 3 4
	0 1 2 3 0 1 1 1 0 2 3
	0 1 2 3 0 0 0 1 1 2
	0 1 2 3 4 0 1



Aging Algorithmus



- Zustände nach fünf Intervallen (a)-(b) für sechs Seiten

Belady'sche Anomalie

- LRC Strategie

- 3 Seitenrahmen
 - 9 Page Faults

0 1 2 3 0 1 4 0 1 2 3 4

Seitenzugriffs-szenarium

0 1 2 3 0 1 4 4 4 2 3 3
0 1 2 3 0 1 1 1 4 2 2
0 1 2 3 0 0 0 1 4 4

0 1 2 3 0 1

Opfer

0 1 2 3 3 3 4 0 1 2 3 4
0 1 2 2 2 3 4 0 1 2 3
0 1 1 1 2 3 4 0 1 2
0 0 0 1 2 3 4 0 1

0 1 2 3 4 0

Opfer

- 4 Seitenrahmen
 - 10 Page Faults

Keine Anomalie für optimale Strategie

- Optimale Strategie

- 3 Seitenrahmen
 - 6** Page Faults
 - 4 Swap-In

0	1	2	3	0	1	4	0	1	2	3	4
---	---	---	---	---	---	---	---	---	---	---	---

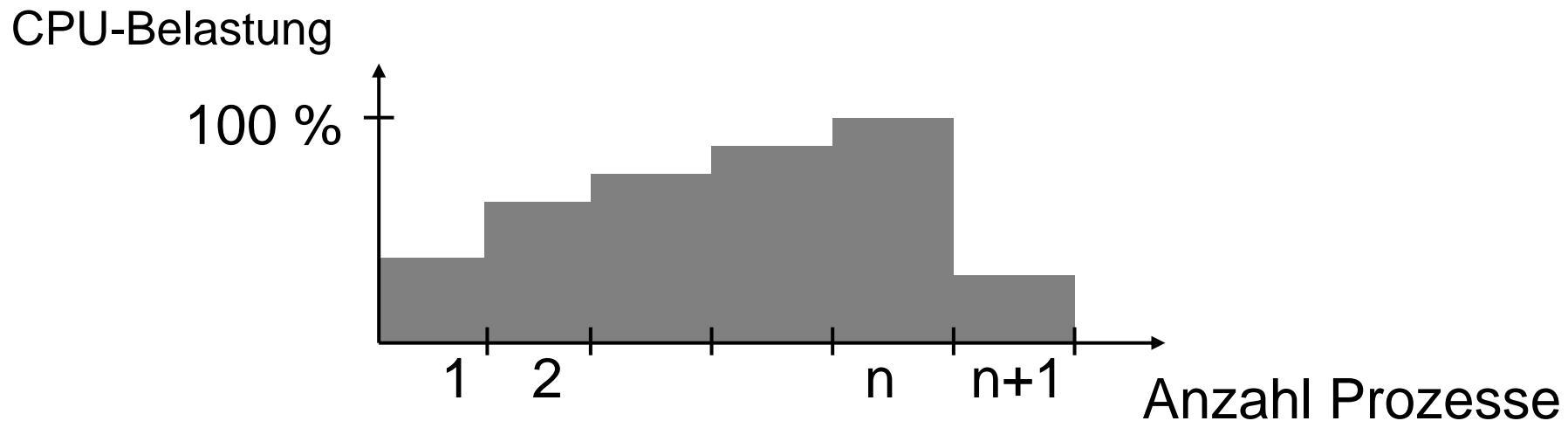
0	1	2	3	0	0	4	4	4	2	3	3
0	1	1	1	1	0	0	0	4	2	2	
0	0	3	3	1	1	1	1	4	4		
		2		3		0	1				

0	1	2	3	3	3	4	4	4	4	3	3
0	1	2	2	2	0	0	0	0	4	4	
0	1	1	1	1	1	1	1	1			
0	0	0	2	2	2	2	2	2			
		3		0							

- LRU und optimale Strategie weisen die Belady-Anomalie nicht auf

Zuteilungsstrategien (1)

- Gleichmässige Verteilung
 - Jeder Prozess erhält gleich viel Realspeicher
 - Phänomen: Thrashing (Schlagen)
 - Übergrosser Paging-Bedarf wegen Realspeichermangels im aktuellen *Working-Set** einiger Prozesse



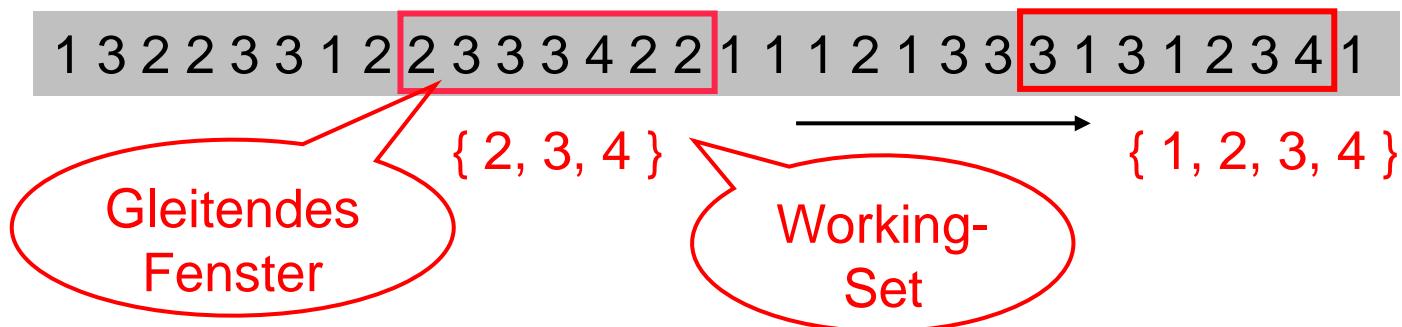
* Working Set $w(P, t_0, t_1)$:

Menge der im Intervall $[t_0, t_1]$ vom Prozess P benötigten Seiten

Zuteilungsstrategien (2)

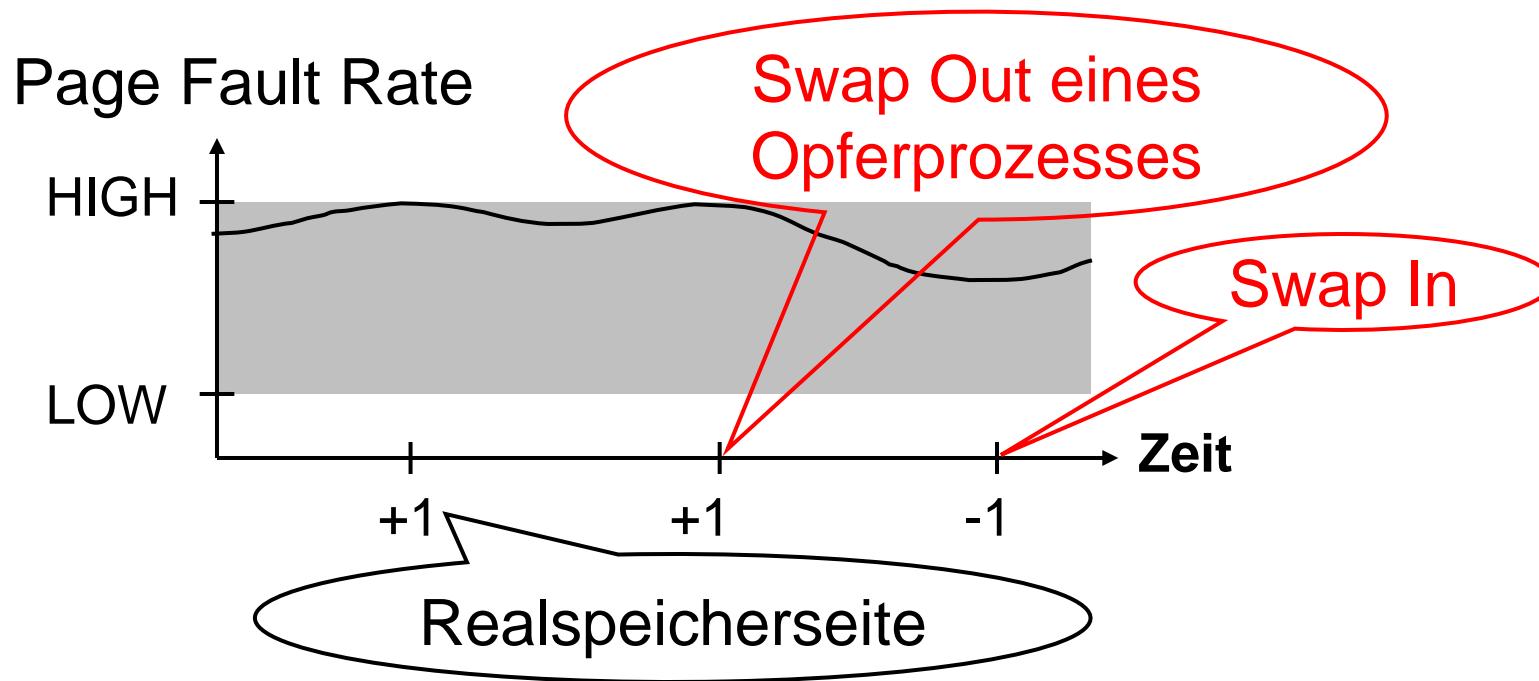
- Bedarfsorientierte Verteilung (1)
 - Steuerung direkt über das **Working Set**
 - Realspeicherzuteilung nach Grösse des Working Sets. Wenn nötig **Swap Out** des “Opferprozesses”

Speicherzugriffsprotokoll

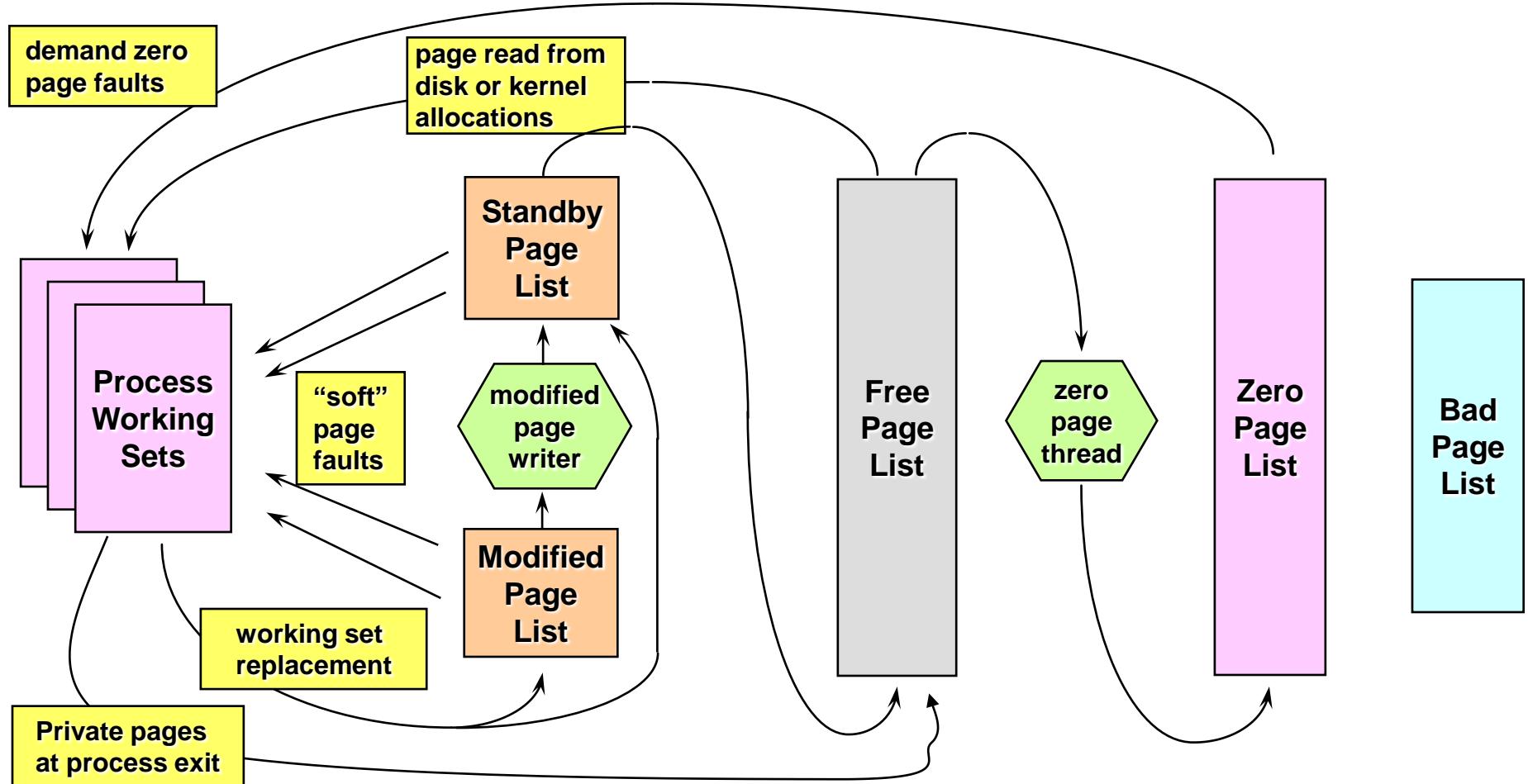


Zuteilungsstrategien (3)

- Bedarfsorientierte Verteilung (2)
 - Steuerung über Page-Fault Rate



Paging Dynamik (Windows)



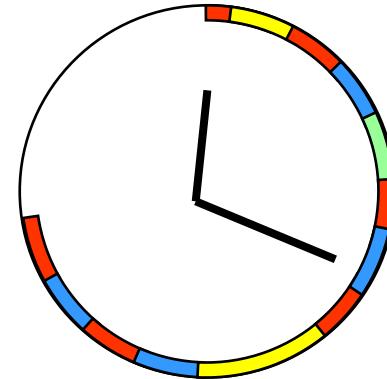
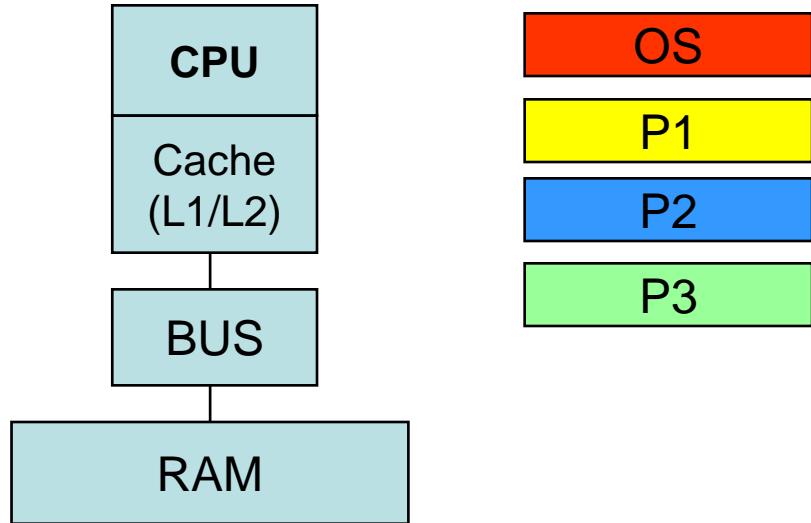
aus: *Windows Operating System Internals Curriculum Development Kit*,
David A. Solomon und Mark E. Russinovich

Kapitel 5. Prozessorenverwaltung

- 5.1 Parallele Architekturen
 - Multiprogramming, SMP, NUMA, Multicore
- 5.2 Konzeptuelles zu Prozessen
 - gemeinsame Ressourcen
 - kritische Abschnitte, Semaphoren
 - Locks und Deadlocks
 - Monitore und Signale
- 5.3. Prozesse
- 5.4. Scheduling

5.1. Parallelle Architekturen

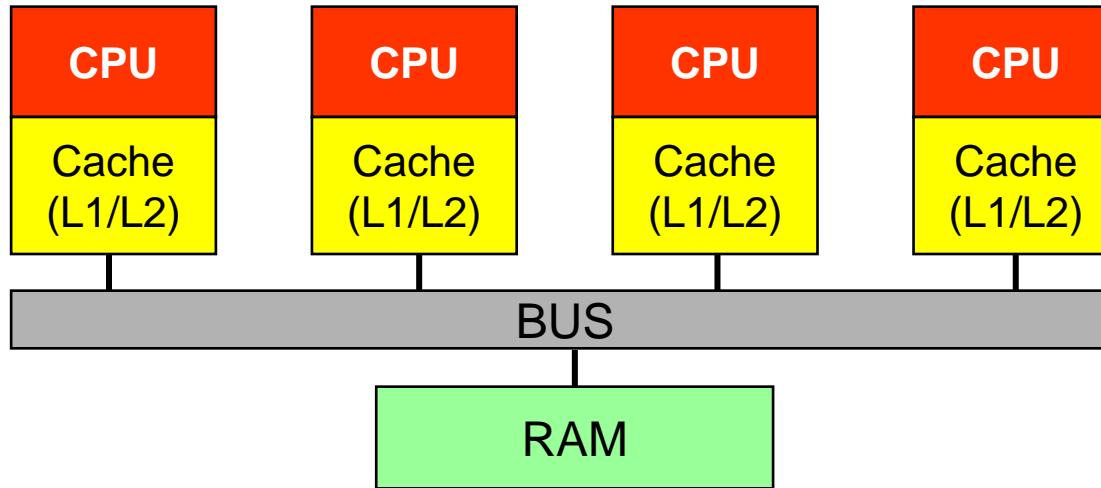
Multiprogramming auf einem Prozessor



Prozesswechsel durch

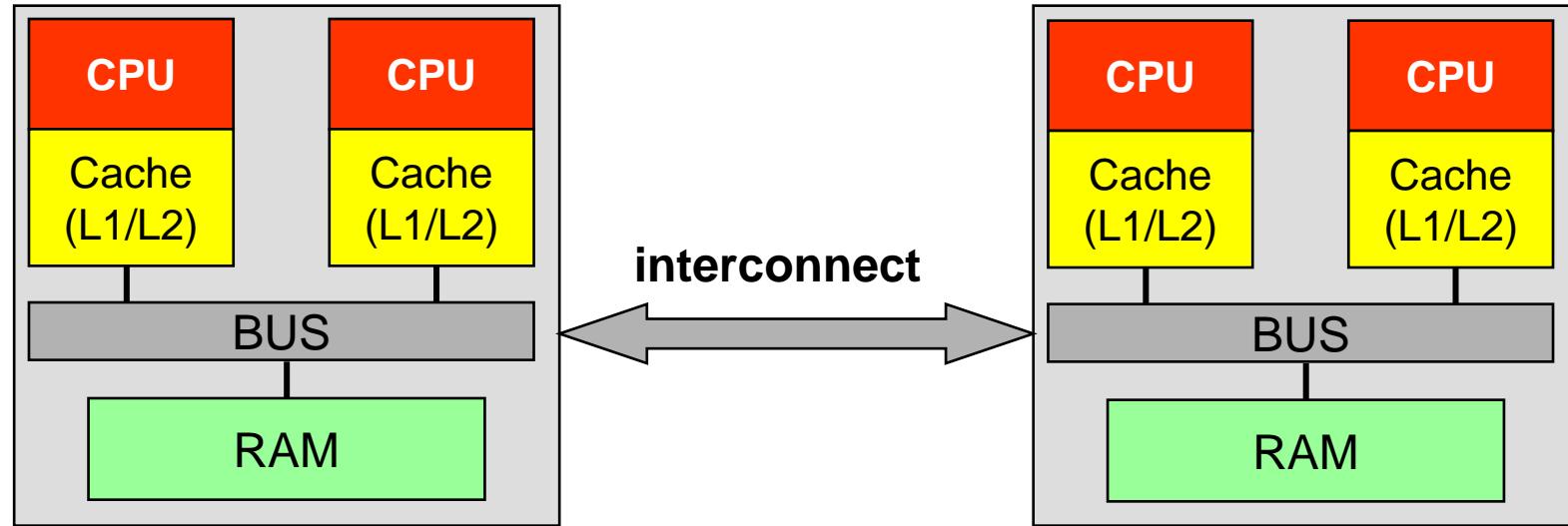
- Interrupts:
 - asynchron: Timer / IO / Trap (implizit)
 - synchron: Trap (explizit)
- Explizite Kontrollübergabe („Yield“)

SMP – Symmetric Multiprocessing



- Alle Prozessoren haben gleichberechtigten Zugriff zum gleichen RAM
- Hardwareunterstützung für die Cache-Kohärenz
 - Bus Snooping
 - MESI Protokoll
(Modified, Exclusive, Shared & Invalid Bits im Cache)

NUMA – Non Uniform Memory Access



- Zugriff zum lokalen RAM schnell
- Zugriff zum entfernten RAM langsam
- Beispiele
 - Cell (IBM, Sony & Toshiba): PPE, 8 SPEs
 - NVIDIA GPU
 - Project „Supercomputer In The Pocket“
- Verschiedene Speicher-Konsistenzmodelle in Hardware implementiert

Multicore

- Mehrere Prozessorkerne auf einem Die
 - Separater und gemeinsamer Cache (L1-L3)
 - Unterschied aus Sicht des Betriebssystems:
 - Aufteilung der Speicherzugriffe für verschiedene Prozesse kann zu Cache-Trashing im Prozessorkern führen.
- Einsatz auf Architekturen
 - NUMA
 - z.B. AMD Opteron
 - Separater Zugriff zu RAM und zu Geräten
 - SMP
 - z.B. Intel Xeon
 - Gemeinsamer Zugriff zu RAM und Geräten

5.2. Prozesskonzept

- Prozess = Ablauf
- Gegenstand der Betrachtung
 - Prozess bestehend aus einer Menge separater, sequenzieller Teilprozesse
- Synchronisationsbedürfnisse
 - Verwendung gemeinsamer Ressourcen
 - Kausale Interoperabilität

Motivation: Produzenten-Konsumenten Problem

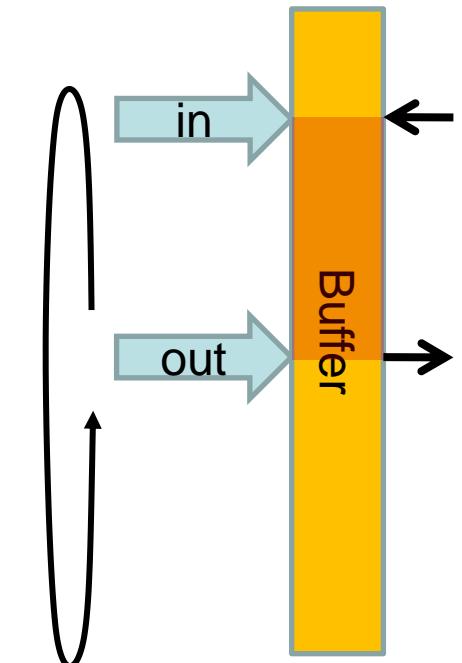
Naive Implementation eines Bounded Buffer

- Produzent

```
while (counter == bufferSize)
    ; /* warten bis Puffer nicht voll */
buffer[in] = produced;
in = (in+1) % bufferSize;
counter++;
```

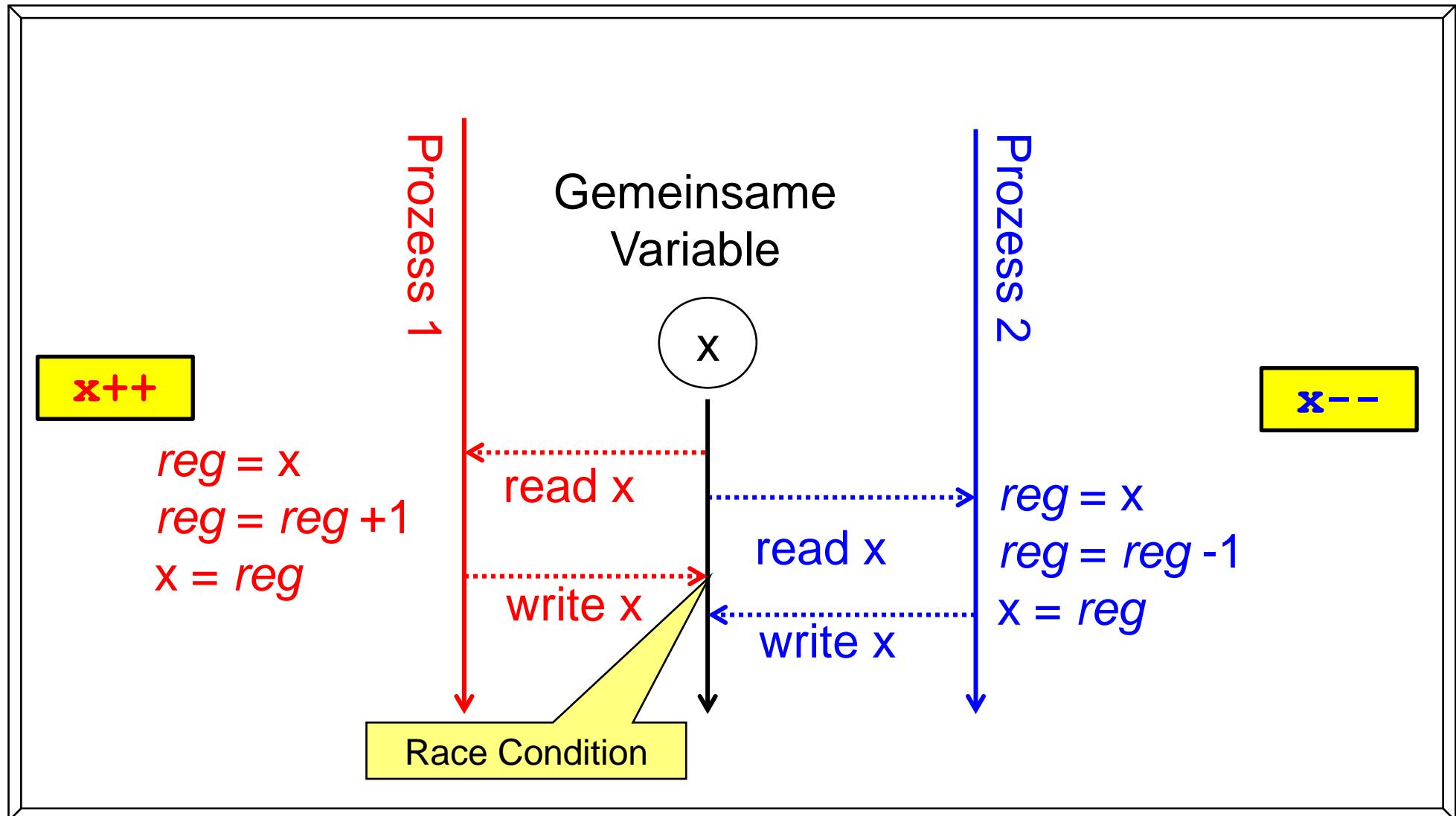
- Konsument

```
while (counter == 0)
    ; /* warten bis Puffer nicht leer */
consumed = buffer[out];
out = (out+1) % bufferSize;
counter--;
```

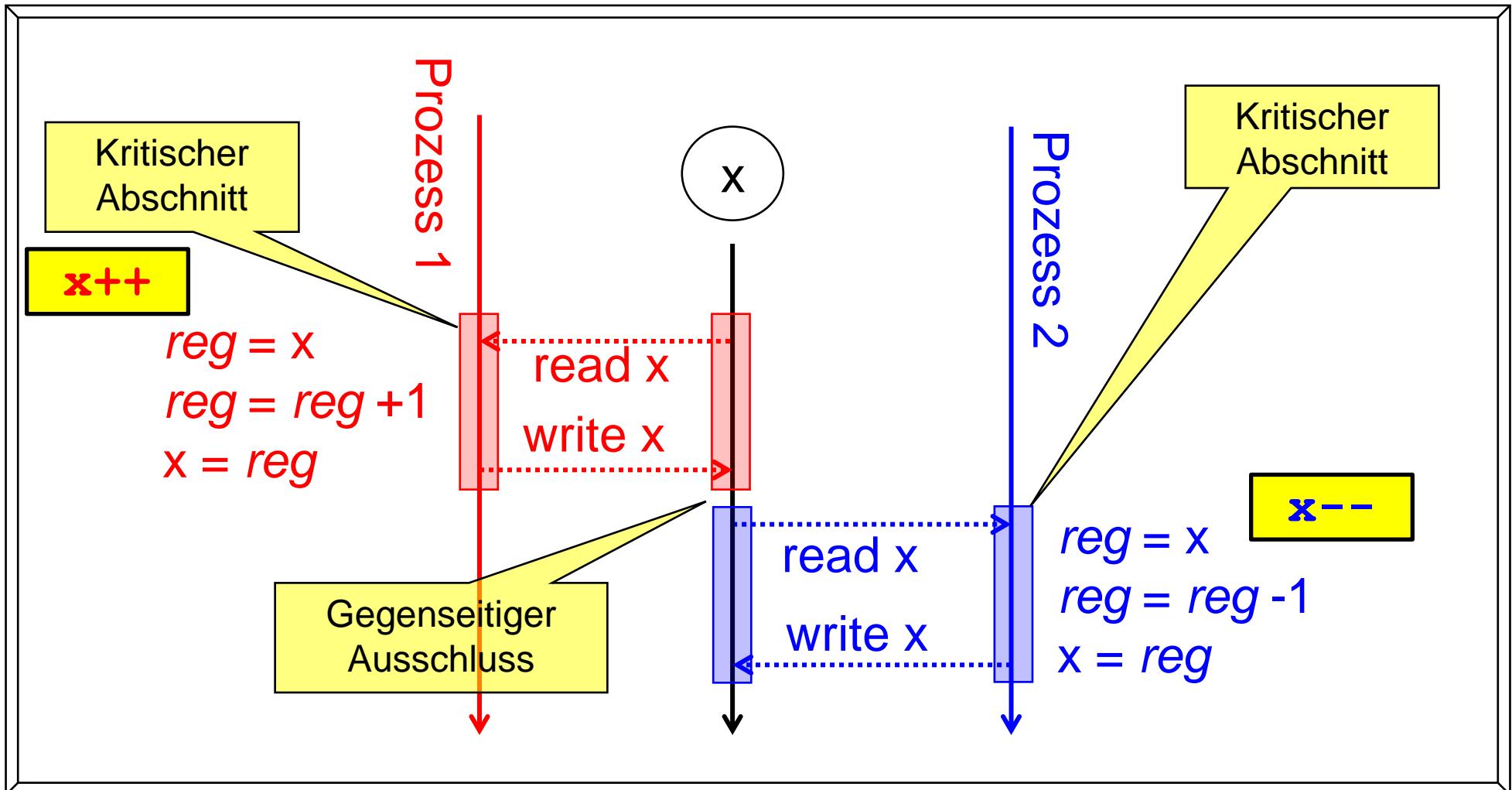


- Beispiel: Produzent = Gerät (über IRQ), Konsument = System / Applikation

5.2.1. Gemeinsame Ressourcen



5.2.2 Kritische Abschnitte*



*critical sections

Bedingungen für kritische Abschnitte

1. Keine zwei Prozesse dürfen gleichzeitig in eine kritische Region eintreten (**Wechselseitiger Ausschluss***).
2. Es darf keine Annahme über Geschwindigkeit oder Anzahl der CPUs gemacht werden.
3. Nur dadurch, dass ein Prozess die kritische Region betreten hat, dürfen andere Prozesse am Eintreten gehindert werden.
4. Kein Prozess sollte ewig auf Zutritt warten müssen. Kein Prozess sollte ewig im k.A. verweilen. Kein **Verhungern****, keine **Verklemmung*****.

*mutual exclusion **starvation ***deadlock

Softwareansatz: Vorschläge (1)

- ~~Variablen durch Hilfsvariablen sperren~~
→ Problem nur verschoben, weiterhin „Race Conditions“

- ~~Strikter Wechsel:~~

turn = 0

P0:

```
while(turn != 0)
    /* do nothing */;
/* critical section */
turn=1;
```

[1]

[2]

[3]

P1:

```
while(turn!=1)
    /* do nothing */;
/* critical section */
turn=0;
```

busy waiting:
spinlock

verletzt Bedingung 3: P0: [1],[2],[3],[1] (locked)

Weiterer Versuch (2)

flag = {false,false}

P0:

```
while(flag[1])
    /* do nothing */;
flag[0]=true;
/* critical section */
flag[0]=false;
```

[1]

[2]

[3]

[4]

P1:

```
while(flag[0])
    /* do nothing */;
flag[1]=true;
/* critical section */
flag[1]=false;
```

noch schlechter: **kein gegenseitiger Ausschluss**

P0,P1: [1]; P0,P1:[2];

Weiterer Versuch (3)

P0:

```
flag[0] = true; [1]
while(flag[1])
    /* do nothing */
/* critical section */
flag[0]=false;
```

[2]

[3]

[4]

P1:

```
flag[1] = true; [1]
while(flag[0])
    /* do nothing */
/* critical section */
flag[1]=false;
```

Verklemmung (deadlock)

P0,P1: [1] ; P0, P1: [2];

Weiterer Versuch (4)

P0:

```
flag[0] = true;  
while(flag[1])  
{  
    flag[0]= false;  
    /* delay */  
    flag[0]=true;  
}  
/* critical section */  
flag[0]=false;
```

[1]

[2]

[3]

[4]

[5]

[6]

P1:

```
flag[1] = true;  
while(flag[0])  
{  
    flag[1]= false;  
    /* delay */  
    flag[1]=true;  
}  
/* critical section */  
flag[1]=false;
```

dynamische Verklemmung (livelock)

P0,P1: [1]; P0,P1: [2]; P0,P1: [3]; P0,P1: [4]; ...



Konsequenz: Dekker Algorithmus (vor 1965)

```

flag = {false,false}; turn=1;

flag[0] = true; [1] flag[1] = true;
while(flag[1]) [2] while(flag[0])
if (turn == 1) [3] if (turn ==0)
{ [4] {
flag[0]= false; [5] flag[1]= false;
while(turn==1); [6] while(turn==0);
flag[0]=true; [7] flag[1]=true;
}
/* critical section */ [8] /* critical section */
turn=1; [9] turn=0;
flag[0]=false; [10] flag[1]=false;

```

Besser, einfacher: Peterson Algorithmus (1981)

flag = {false,false};

flag[0] = true;

[1]

turn = 1;

[2]

while(flag[1] && turn == 1)

[3]

/* wait */;

/* critical section */

[4]

flag[0]=false;

[5]

flag[1] = true;

turn = 0;

while(flag[0] && turn == 0)

/* wait */;

/* critical section */

flag[1]=false;

Implementation von Ausschluss

Im Falle eines Prozessors

- Potentielle Konkurrenten sind Interruptroutinen und DMA Transfers
 - Strategie: Mask Interrupts; CS; Unmask Interrupts
 - ⌚ funktioniert nur bei Einprozessorsystemen
 - ⌚ Kontrollentzug des Systems
- Manchmal günstig und nötig im Kern des OS.

```
/* Interrupts sperren */  
/* critical section */  
/* Interrupts freischalten */
```

Implementation von Ausschluss mit Hardware Unterstützung

- Atomare Instruktionen zum Prüfen / Setzen einer Speicherzelle
 - Test-And-Set (TAS)
 - Compare-And-Swap (CAS, Intel: LOCK CMPXCHG)
 - Exchange (XCHG, Intel: LOCK XCHG)

TAS s

```
if (s == 0)
  {s = 1;cc = true}
else
  cc=false
end
```

CAS R1,R2,A

```
if (R1 == M[A])
  { M[A] = R2; CC = true; }
else
  { R1 = M[A]; CC = false; }
end
```

XCHG s,t

```
tmp := s;
s := t;
t := tmp;
```

Spinlock mit XCHG

PROCEDURE Acquire (VAR lock: BOOLEAN);

CODE {SYSTEM.i386}

MOV EBX, lock[EBP] ; EBX := ADR(lock)

MOV AL, 1 ; AL := 1

test:

XCHG [EBX], AL ; set and read the lock atomically.

CMP AL, 1 ; was locked?

JNE exit ; no, we have it now

NOP ; spin hint

JMP test

exit:

END Acquire;

PROCEDURE Release(VAR lock: BOOLEAN);

BEGIN

lock := FALSE; (* equivalently [ADR(lock)] := 0 *)

END Release;

Implementation von Ausschluss

Im Falle mehrerer Prozessoren

Spinlocks: Petersons Algorithmus oder Hardware
Unterstützung

- 😊 beliebige Anzahl Prozesse (nur für Hardware Locks)
 - 😊 beliebige Anzahl Prozessoren
 - 😊 Unterstützung mehrerer kritischer Abschnitte
 - ☹ Aktives Warten: Prozesse verbrauchen Prozessorzeit
 - ☹ Verhungern (Starvation) möglich: Auswahl des wartenden Prozesses zufällig, es kann passieren, dass ein Prozess unendlich lange wartet
 - ☹ Verklemmungen (Deadlocks) möglich, z.B. durch Prioritätsprobleme.
- Verwendung von Scheduling-basierten Locks auf Applikationsebene: Semaphoren, Mutexe, Monitore ...

Lock Vermeidung

- Erhöhung der OS-Parallelität durch Entfernen globaler Locks
 - z.B. Linux Kern 2.4 -> 2.6: globale Locks weitgehend ersetzt durch lokale Locks
- Vermeidung von Locks
 - Einsatz von Strukturen, die kein Lock benötigen
 - z.B. Ringbuffer bei Zugriff durch nur zwei Prozesse
 - Read-Copy-Update (RCU) für Readers-Writers Lock
 - Readers greifen ohne Lock zu
 - Writers (im gegenseitigen Ausschluss)
 - Kopiere Daten
 - Modifiziere Daten
 - Aktualisiere Referenz (Laufzeitunterstützung / Metadaten!)
 - Warte auf Austritt aller Readers (Laufzeitunterstützung!)

5.2.3 Semaphore (E. W. Dijkstra, 1965)



σεμα = zeichen, φέρω = tragen

Semaphore (1)

- Ganzzahliger, abstrakter Datentyp mit Operationen

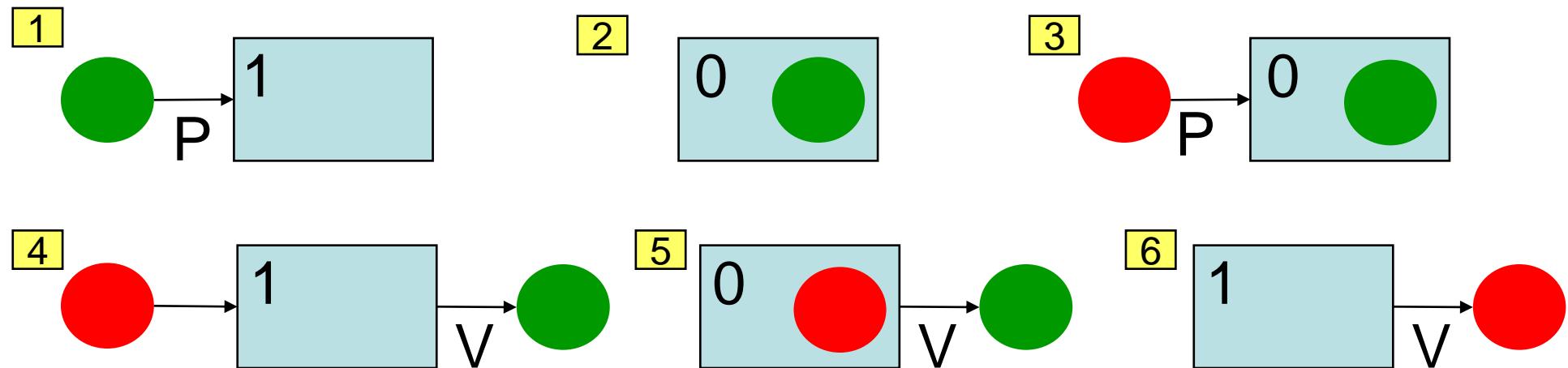
P ("proberen"), **V** ("vrijgeven")

- Semantik

P(S) ≡ { S > 0 } dec(S);

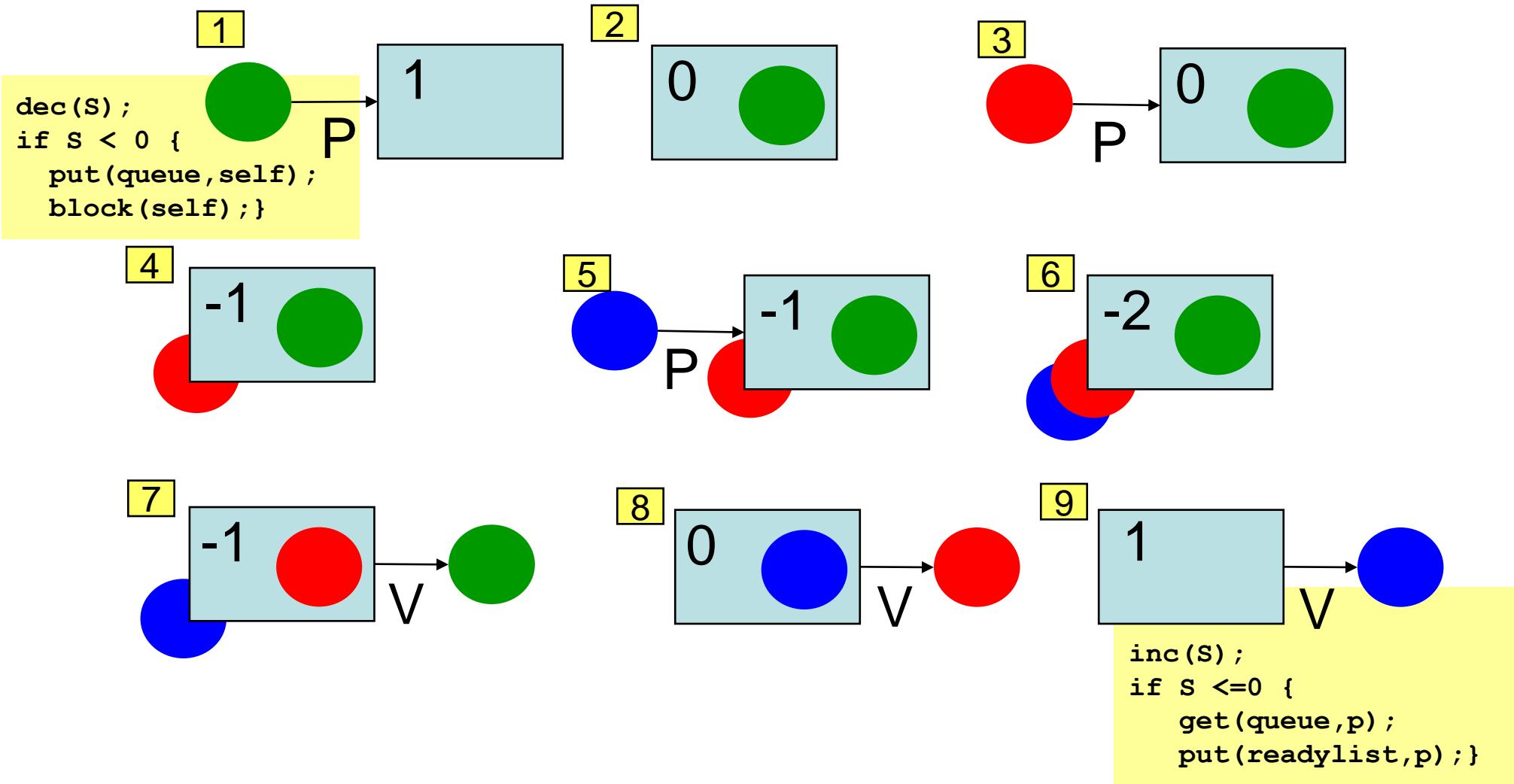
V(S) ≡ inc(S);

warte, bis $S > 0$,
atomic: sobald $S > 0$: $\text{dec}(S)$



Semaphore (2)

mit (FIFO-) Warteschlange



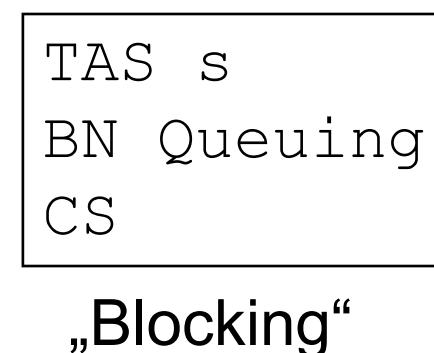
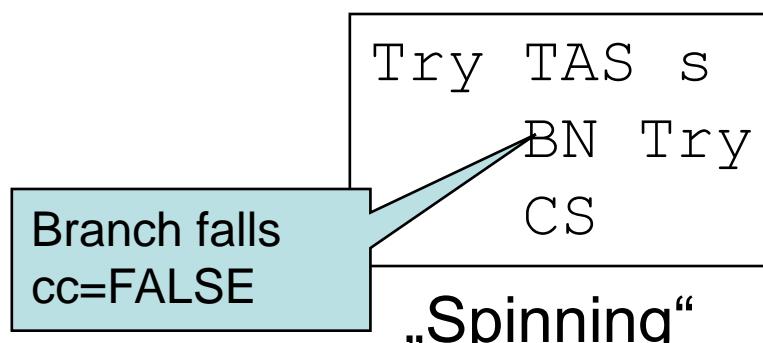
Kritische Abschnitte mit Semaphoren

- Semaphore s = 1;
. . . P(s); . . . ; V(s); . . . ; P(s); . . . ; V(s); . . .

Kritischer
Abschnitt

Kritischer
Abschnitt

- Implementation von Semaphoren mit "Spinning", "Blocking" oder Kombination



Branch falls
cc=FALSE

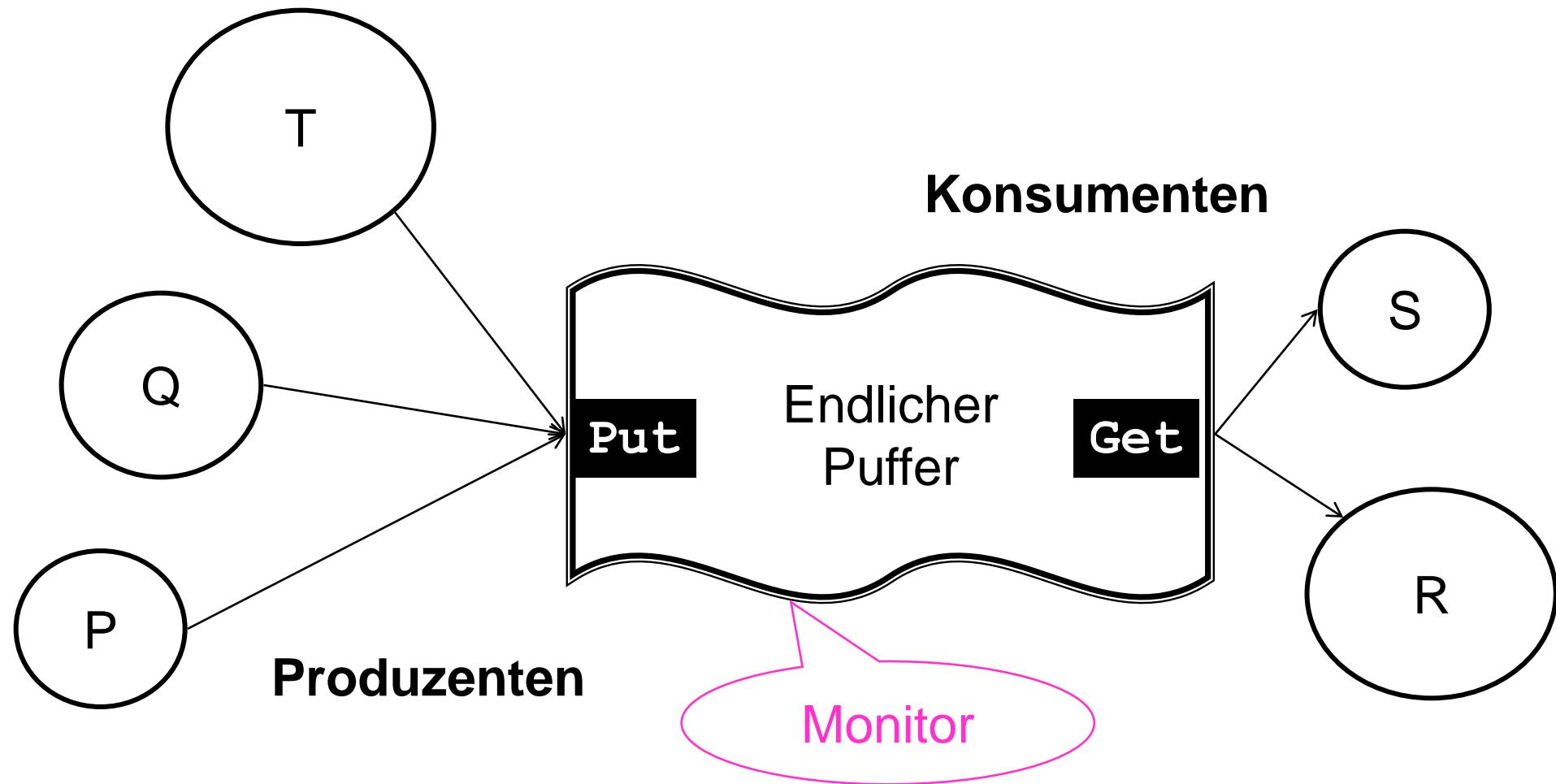
„Spinning“

„Blocking“

5.2.4. Monitore & Signale

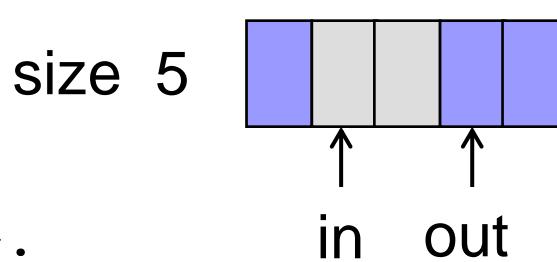
- Monitor (Brinch-Hansen, Hoare 1972)
 - Kapselung einer Datenstruktur samt aller Zugriffsmethoden zu einem Objekt
 - Methodenaufruf im gegenseitigen Ausschluss
 - Signalisationskonzept zur Ablaufsteuerung innerhalb des Monitorobjektes
- Signale
 - Abstrakter Datentyp zur Repräsentierung von *Bedingungen* innerhalb des Monitorobjektes
 - Operationen *Wait* und *Signal* zum Warten auf eine Bedingung bzw. Signalisieren einer Bedingung

Beispiel: Konsumenten & Produzenten



Endlicher Puffer als Monitor

```
monitor Buffer {  
    private int size;  
    private object [] buf;  
    private int out; // pos of next item to get  
    private int in; // pos of next item to put  
    private int m; // number of empty slots  
    private int n; // number of occupied slots  
    public Buffer (int size) {  
        m = size; n = 0; in = 0; out = 0;  
        this.size = size;  
        buf = new object[size];  
    }  
    public void Put (object x) { ... }  
    public object Get () { ... }  
}
```



im
gegenseitigen
Ausschluss

Endlicher Puffer als Monitor

```
private signal nonfull, nonempty;
public void Put (object x)
{
    if (m == 0) wait(nonfull);
    m--;
    buf[in] = x;
    in = (in + 1) % size;
    n++;
    signal(nonempty);
}
public object Get ()
{
    if (n == 0) wait(nonempty);
    n--;
    object x = buf[out];
    out = (out + 1) % size;
    m++;
    signal(nonfull); return x;
}
```

Implementation in Active Oberon

```
Buffer = OBJECT
  VAR size,out,in,m,n: INTEGER;
  buf: POINTER TO ARRAY OF PTR;

  PROCEDURE &Init (size: INTEGER);
  BEGIN
    m := size; n := 0; in := 0; out := 0; SELF.size := size;
    NEW(buf,size);
  END Init;

  PROCEDURE Put(x: PTR);
  BEGIN{EXCLUSIVE}
    AWAIT(m>0);
    DEC(m); buf[in]:= x; in:=(in+1) MOD size; INC(n);
  END Put;

  PROCEDURE Get(): PTR;
  VAR x: PTR;
  BEGIN{EXCLUSIVE}
    AWAIT(n>0);
    DEC(n); x:= buf[out]; out:=(out+1) MOD size; INC(m);
    RETURN x;
  END Get;

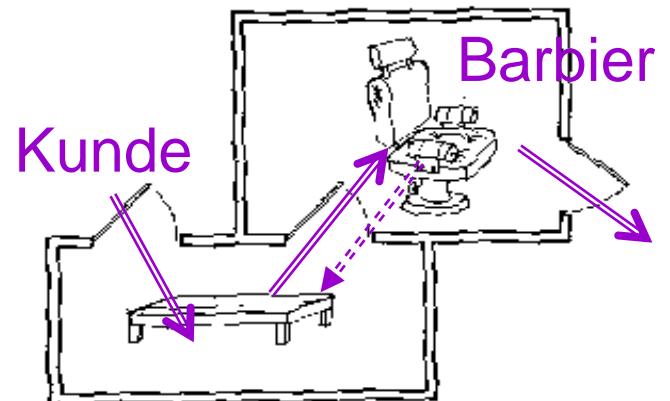
END Buffer;
```

Exclusive Block: beim Verlassen werden alle offenen AWAITs geprüft.
Kein Signaling notwendig.

"Sleeping Barber" Variante

- Nachteil der präsentierten Lösung
 - Signale werden prophylaktisch versandt, z. B. `signal (nonempty)` wenn kein Konsument wartet
- "Sleeping Barber" Metapher (Dijkstra)
 - Erweiterte Definitionsbereiche

$m \leq 0 \Leftrightarrow$ buffer full
 & -m Produzenten warten
 $n \leq 0 \Leftrightarrow$ buffer empty
 & -n Konsumenten warten



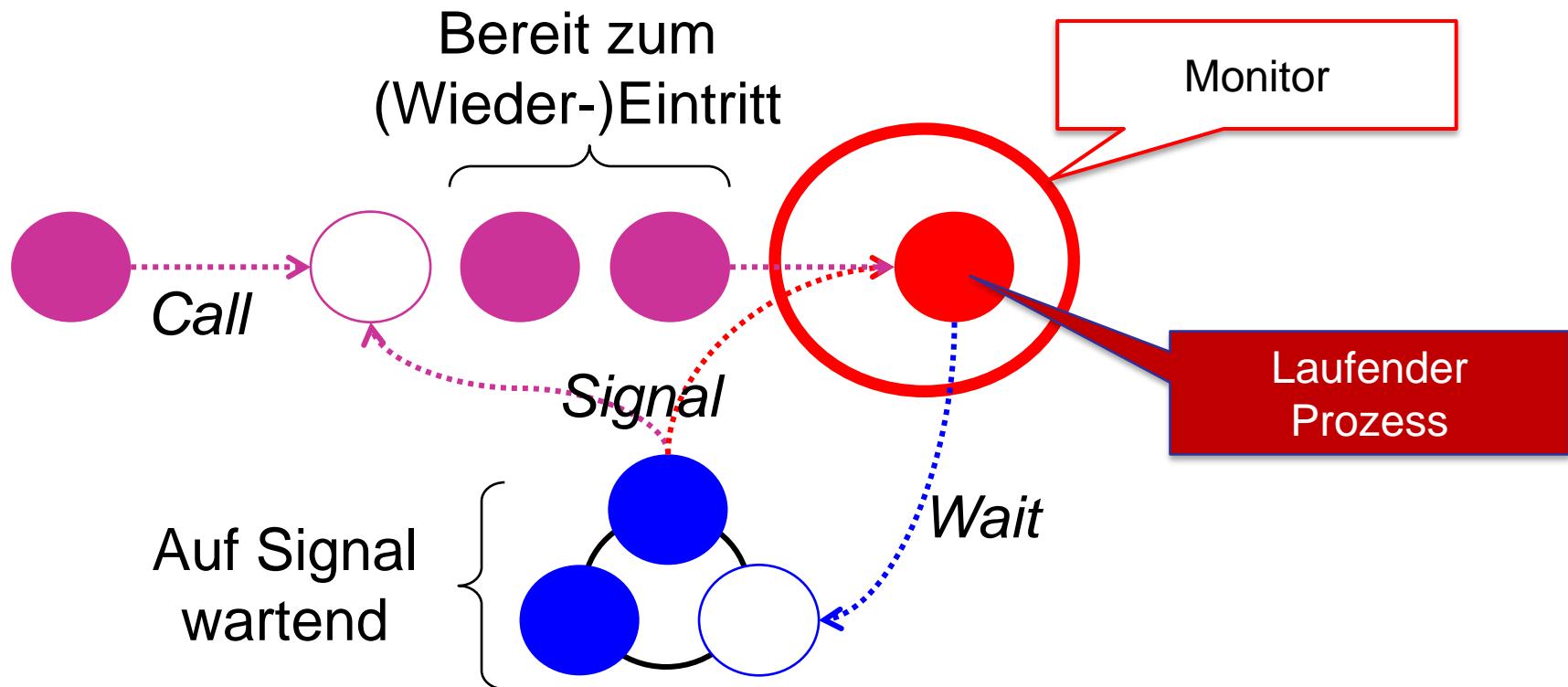
- Z. B. `signal (nonempty)` nur wenn $n < 0$,
d. h. mindestens ein Konsument wartet (Barbier schläft)

"Sleeping Barber" Implementation

```
monitor Buffer
{
    ...
private signal nonfull, nonempty;
public void Put (object x) {
    m--;
    if (m < 0) wait (nonfull);
    buf[in] = x;
    in = (in + 1) % size;
    n++;
    if (n <= 0) signal (nonempty);
}
public object Get () {
    n--;
    if (n < 0) wait (nonempty);
    object x = buf[out];
    out = (out + 1) % size;
    m++;
    if (m <= 0) signal (nonfull); return x;
}
}
```

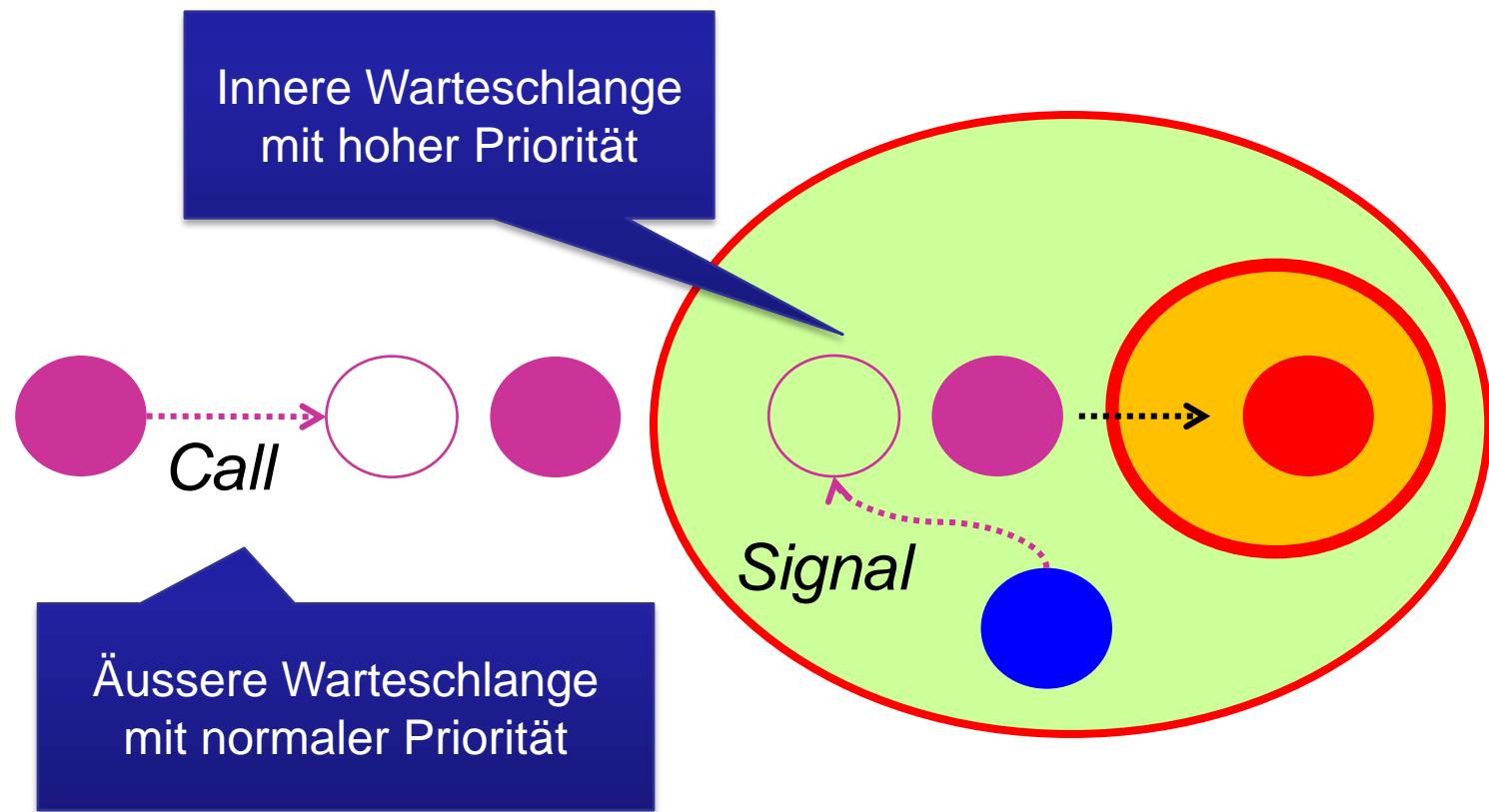
Monitor Laufzeitimplementation

- Zum Monitor gehörende Warteschlangen
 - alle Einlass begehrenden Prozesse
 - alle auf Signal wartenden Prozesse



Das "Eggshell" Modell

- Prozesse im *Wait*-Zustand haben Priorität gegenüber neueintretenden Prozessen



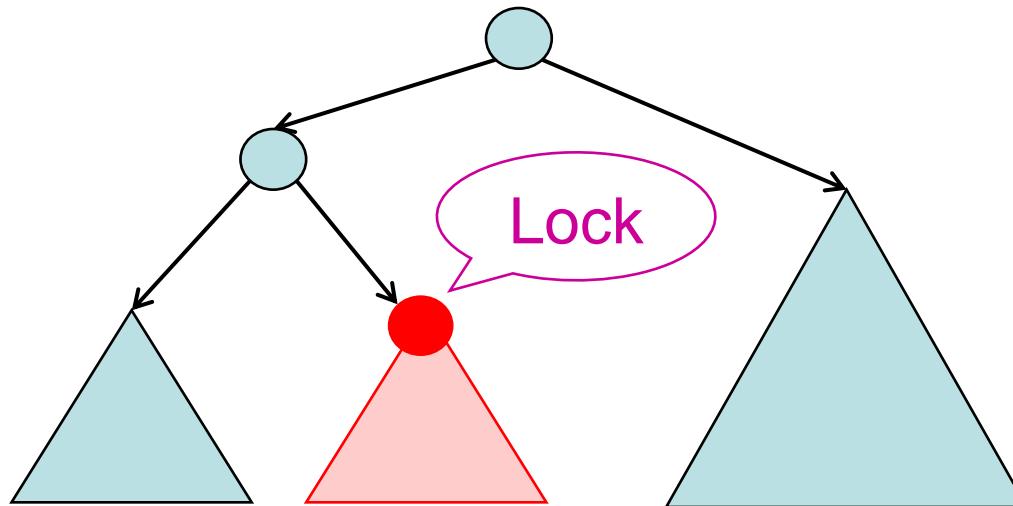
Varianten Wait-/Signal Strategien

Strategie	Sender	Empfänger
Signal and Continue	Behält den Monitor	Geht in die äussere Warteschlange
Signal & Exit	Tritt aus Monitor aus	Erhält den Monitor
Signal and Wait	Geht in die äussere Warteschlange	Erhält den Monitor
Signal and Urgent Wait (Favorit von Hoare & Dahl)	Geht in die innere Warteschlange	Erhält den Monitor

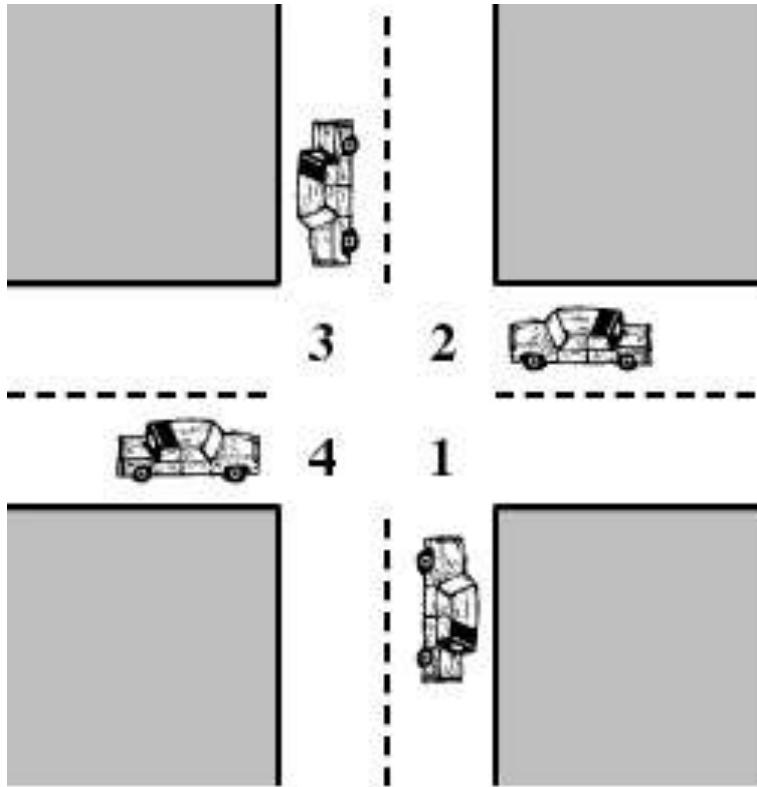
(nach G. R. Andrews, Concurrent Programming, Principles and Practice,
The Benjamin/Cummings Publishing Company, 1991)

5.2.5. Locks und Deadlocks

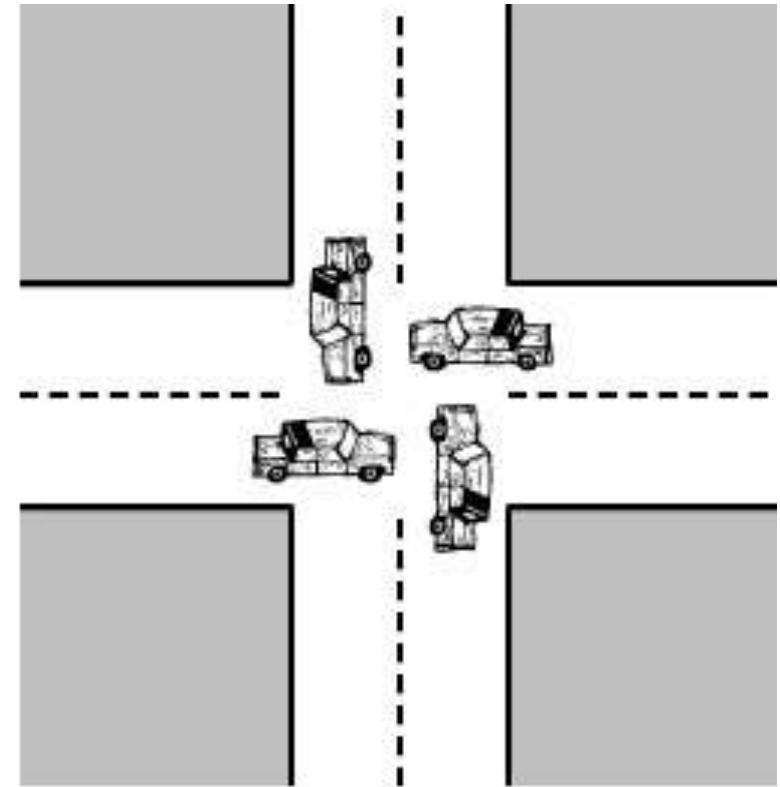
- Shared Resource R → Operationen Lock(R), Unlock(R)
- Besitz des Locks als Vorbedingung für den Zugriff auf die Ressource
- Implementation von Locks als Semaphore L_R mit $\text{Lock}(R) \Leftrightarrow P(L_R)$ und $\text{Unlock}(R) \Leftrightarrow V(L_R)$
- Beispiel: Manipulation in Baumstruktur



Deadlock



(a) Deadlock possible



(b) Deadlock

„Eine Menge von Prozessen befindet sich in einem Deadlock, wenn jeder enthaltene Prozess auf ein Ereignis wartet, das nur ein anderer enthaltener Prozess auslösen könnte.“
(nach Tanenbaum, S.181)

Deadlock Beispiel 1

Process P		Process Q	
Step	Action	Step	Action
p ₀	Request (D)	q ₀	Request (T)
p ₁	Lock (D)	q ₁	Lock (T)
p ₂	Request (T)	q ₂	Request (D)
p ₃	Lock (T)	q ₃	Lock (D)
p ₄	Perform function	q ₄	Perform function
p ₅	Unlock (D)	q ₅	Unlock (T)
p ₆	Unlock (T)	q ₆	Unlock (D)

The diagram illustrates the state of resources D and T for both processes. Resource D is at the top, and Resource T is below it. Red arrows show the sequence of actions: P0 requests D, P1 locks D, P2 requests T, P3 locks T, Q0 requests T, Q1 locks T, Q2 requests D, and Q3 locks D. A diagonal red line connects P3 (Locking T) to Q2 (Requesting D), indicating a deadlock cycle where both processes are waiting for each other's resources.

Deadlock: verschiedene Prozessspfade

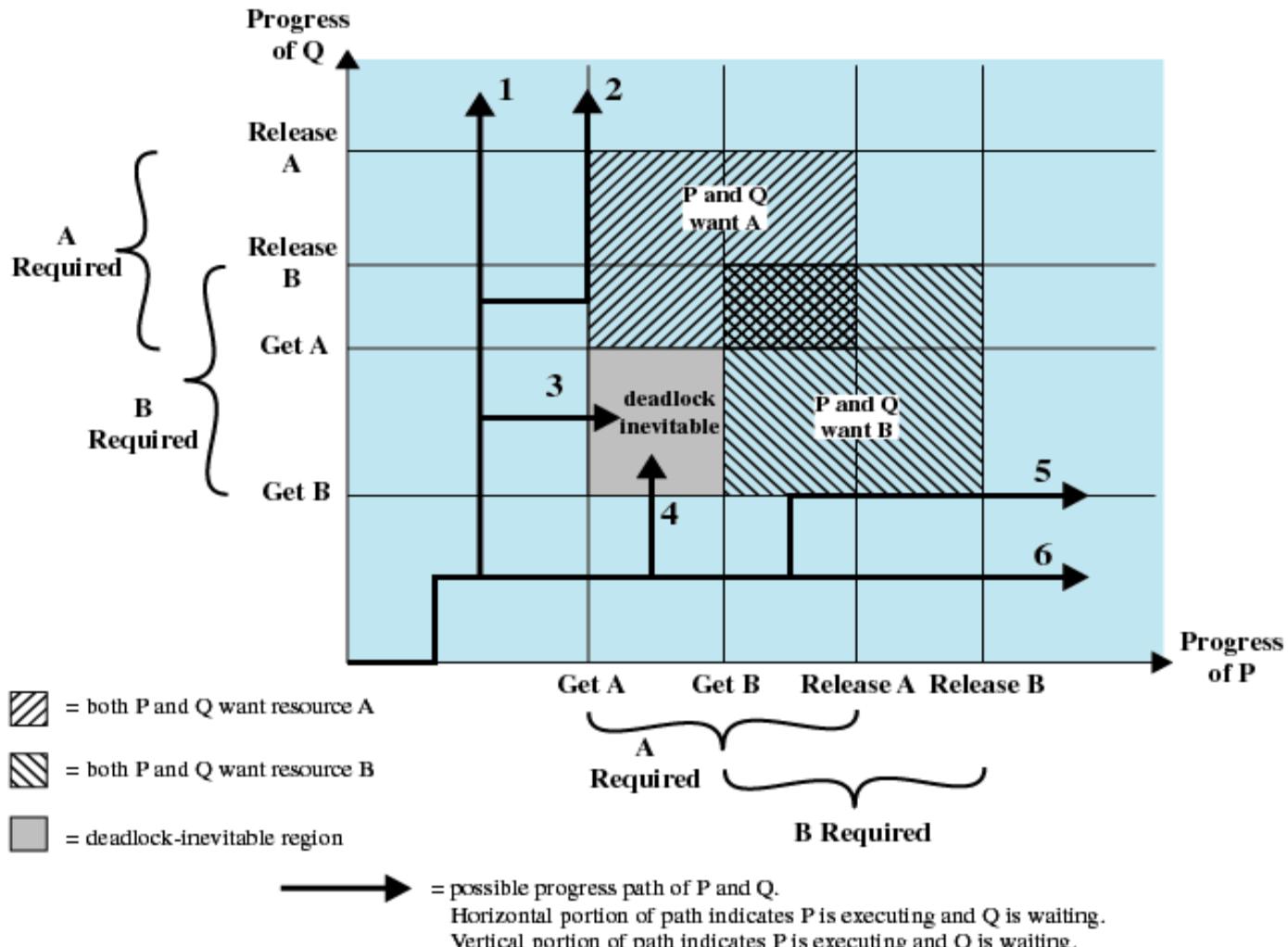


Figure 6.2 Example of Deadlock

Deadlock Beispiel 2

- Angenommen 200 KB seien verfügbar
- Deadlock passiert wenn beide Prozesse zur zweiten Anforderung vorwärtschreiten

Prozess P

Request 80 KB
...
Request 60 KB

Prozess Q

Request 70 KB
...
Request 80 KB

Voraussetzungen für Locks und das Auftreten von Deadlocks

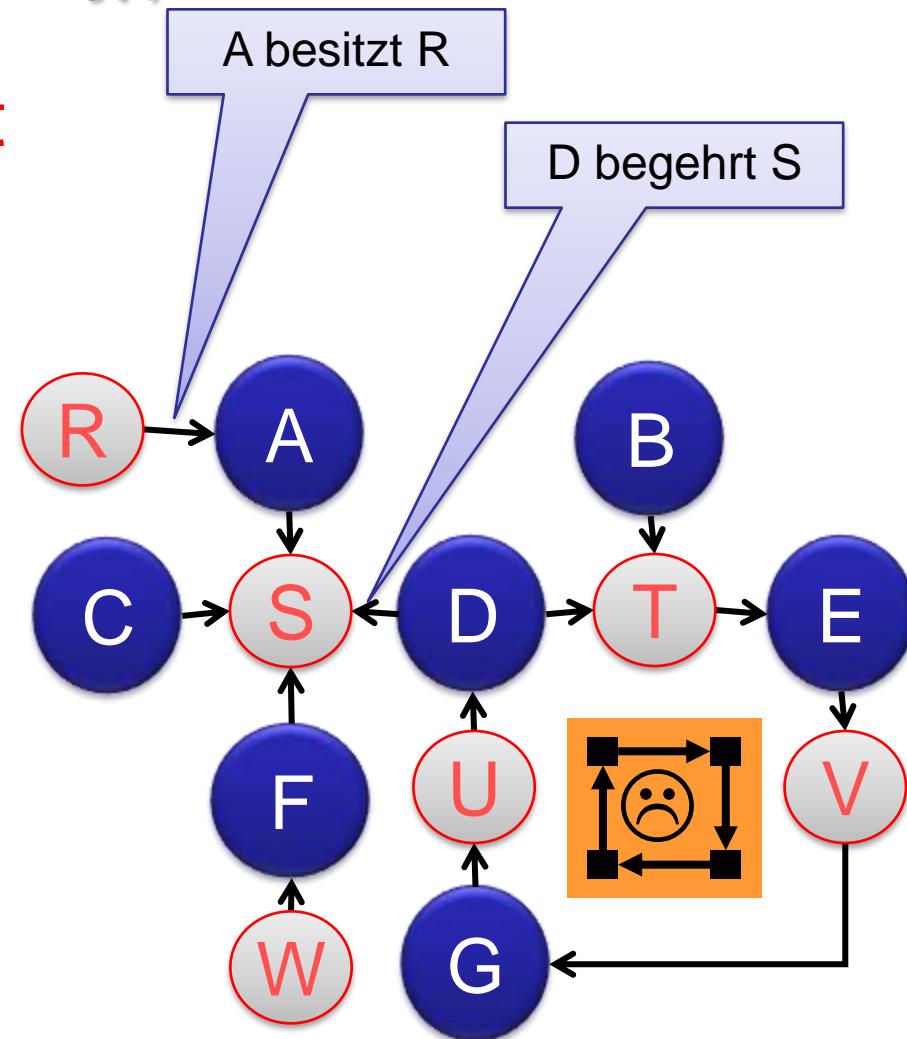
- (1) Wechselseitiger Ausschluss:** jede Resource ist gleichzeitig höchstens einem Prozess zugeordnet.
- (2) Hold and Wait:** wenn einem Prozess bereits Ressourcen zugeordnet sind, kann er weitere Ressourcen anfordern.
- (3) Kein Resourcenentzug:** Ressourcen können einem Prozess nicht „unfreiwillig“ entzogen werden.
- D. Zyklisches Warten:** Es gibt eine zyklische Kette von Prozessen für die gilt: Jeder Prozess wartet auf eine Ressource, die dem nächsten Prozess in der Kette gehört.

Unter (1) - (3) ist D eine hinreichende und notwendige Bedingung für das Vorliegen von Deadlock.

Deadlockerkennung

(bei einer Resource pro Resourcentyp)

Prozess	hält	begehrt
A	R	S
B	-	T
C	-	S, T
D	U	S, T
E	T	V
F	W	S
G	V	U



Deadlockerkennung

(bei mehreren Ressourcen pro Typ)

Tape drives
Plotters
Scanners
CD Roms

$$E = (4 \quad 2 \quad 3 \quad 1)$$

Ressourcenvektor

Tape drives
Plotters
Scanners
CD Roms

$$A = (2 \quad 1 \quad 0 \quad 0)$$

Ressourcenrestvektor

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Belegungsmatrix

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Prozess 1
Prozess 2
Prozess 3

Anforderungsmatrix

Hier kein unvermeidbarer Deadlock, denn möglich:
P3: danach A= (2 2 2 0)
P2: danach A= (4 2 2 1)
P1: danach A= (2 2 3 0)

- **Deadlockfreiheit:** Ausgehend von der Belegungsmatrix und dem Ressourcenvektor gibt es eine Zuteilung, bei der alle Prozesse mit ihren Anforderungen zu Ende laufen können.
- Mit der Bedingung lässt sich nur nachprüfen ob **gegenwärtig** schon unvermeidbar ein Deadlock vorliegt, nicht ob Deadlock in Zukunft noch vorliegen wird!

Deadlockvermeidung

- Globale Ordnung aller Ressourcen
 - Locking nach aufsteigender Ordnungsnummer
 - Zyklen unmöglich

$$R_i \rightarrow A \rightarrow R_j \rightarrow B \rightarrow R_k \rightarrow C \rightarrow R_i$$

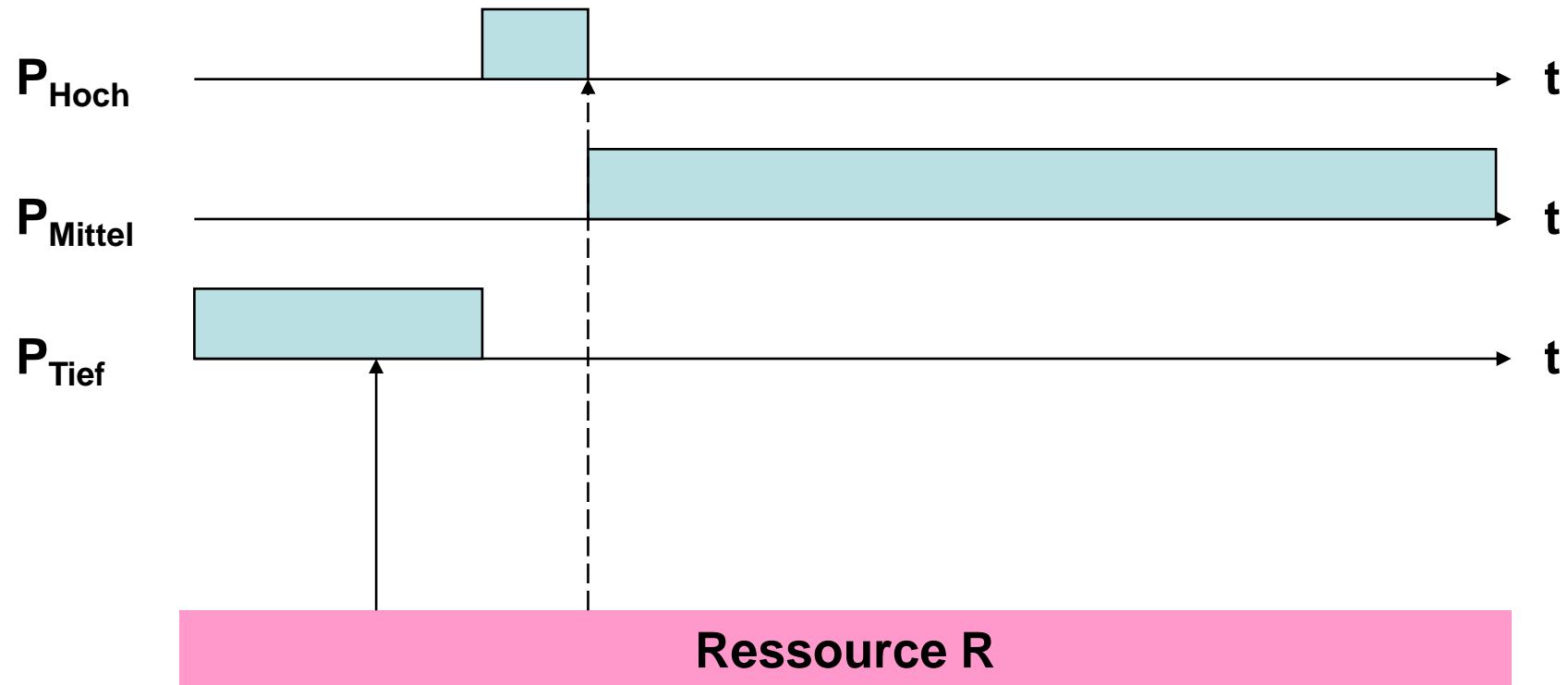
$i < j < k < i$

- Two-Phase-Locking
 - Versuche alle benötigten Ressourcen zu akquirieren
 - Wenn erfolgreich, locke sie, sonst gib sie wieder frei
 - Gefahr des Lockout (Starvation)

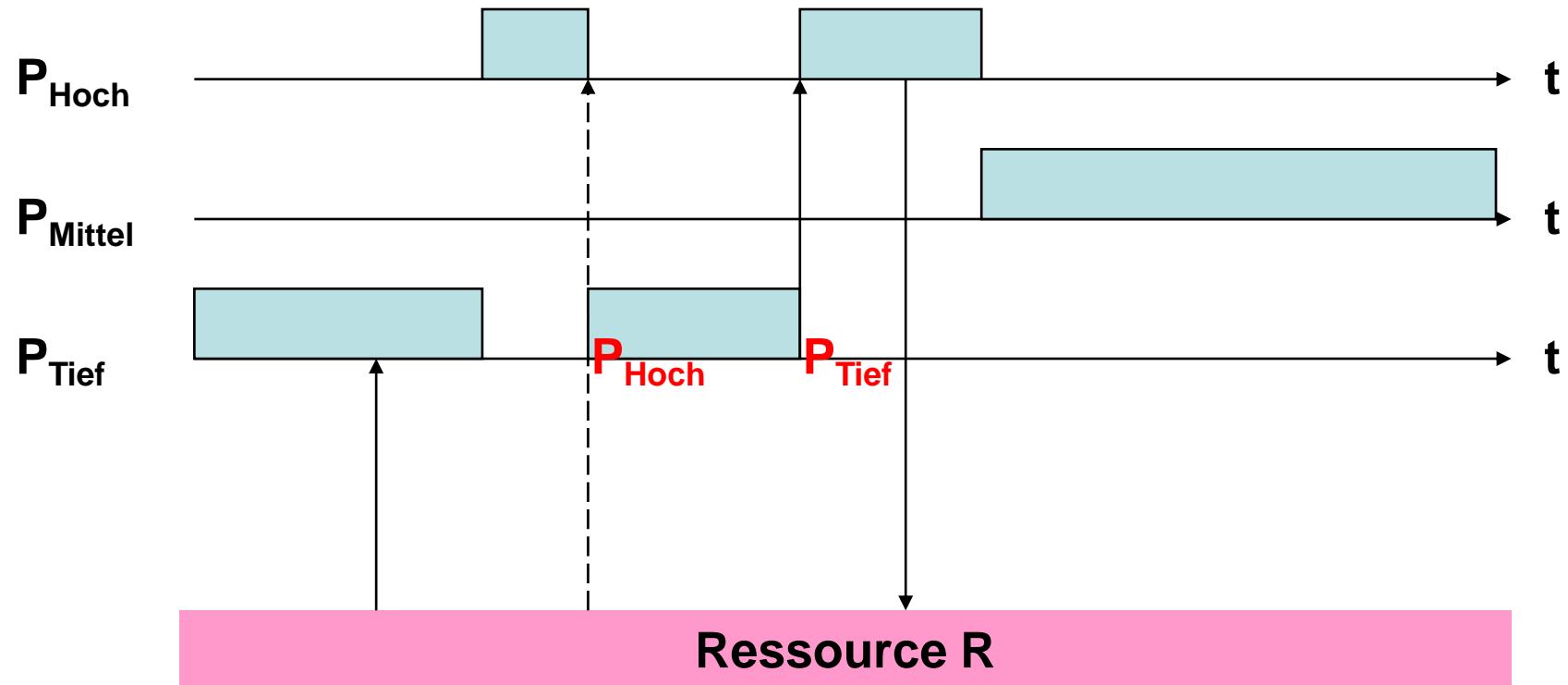
5.2.6. Priority Inversion

- Jedem Prozess wird eine Priorität zugeordnet.
- Priority based preemptive scheduler (PBPS): Der PBPS lässt immer den Prozess mit der höchsten Priorität laufen.
- Die Nutzung gemeinsamer Ressourcen kann die Bemühungen des PBPS torpedieren: Prioritäts-Inversion!

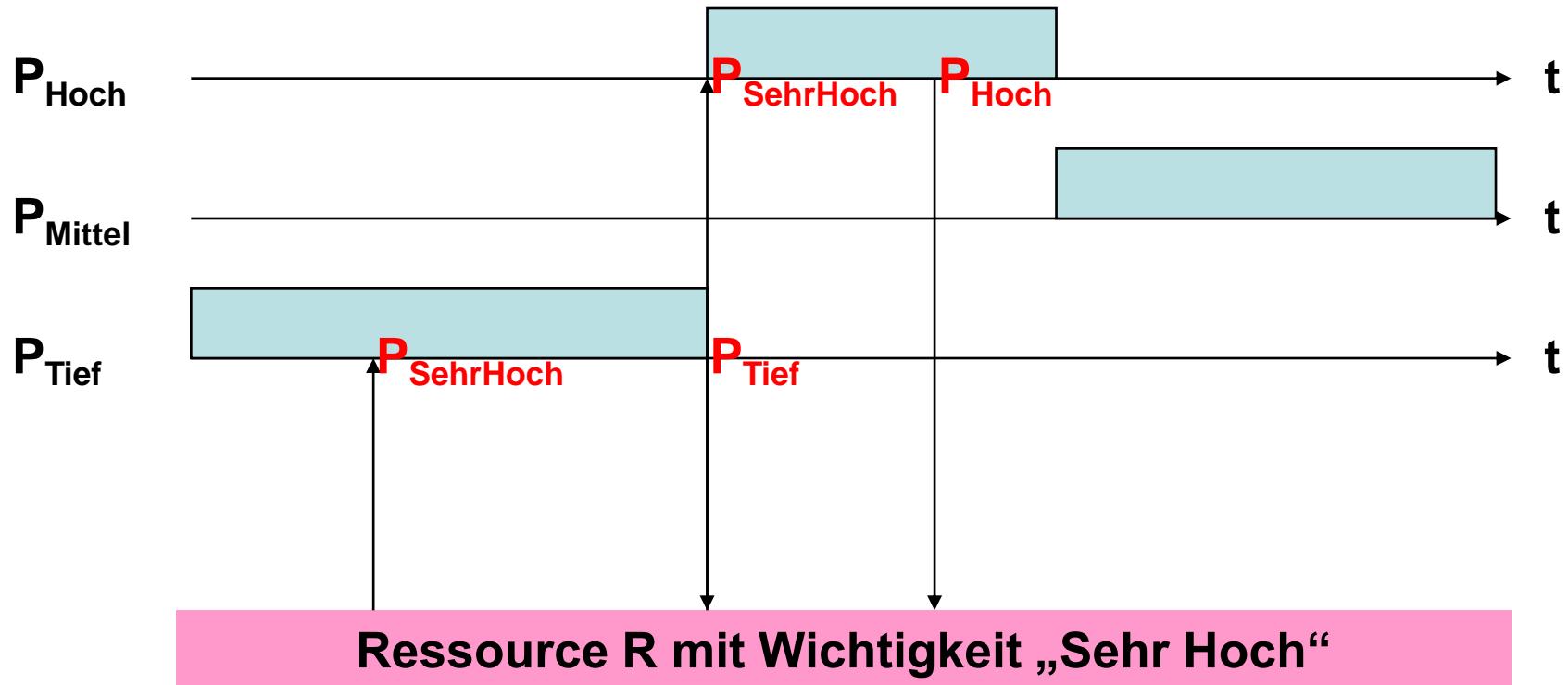
Priority Inversion: Beispiel



Priority Inheritance



Priority Ceiling



Priority Inversion in der Praxis

- Der Mars Rover musste vom Boden aus gepached werden, weil die Programmierer sich der Problematik nicht bewusst waren.
- *“Build exceptional reliability and scalability into your embedded device. A true microkernel operating system, ...offers industry-leading realtime performance.”*

Priority Inversion in der Praxis

Fehlerhafte Implementation der Prioritätsvererbung

P5:18

P4 fordert b

Priorität von P5

P4:16

P2 erhält b

P3:14

P2 fordert b

P2:12

P1:10

10 10 10 10 10

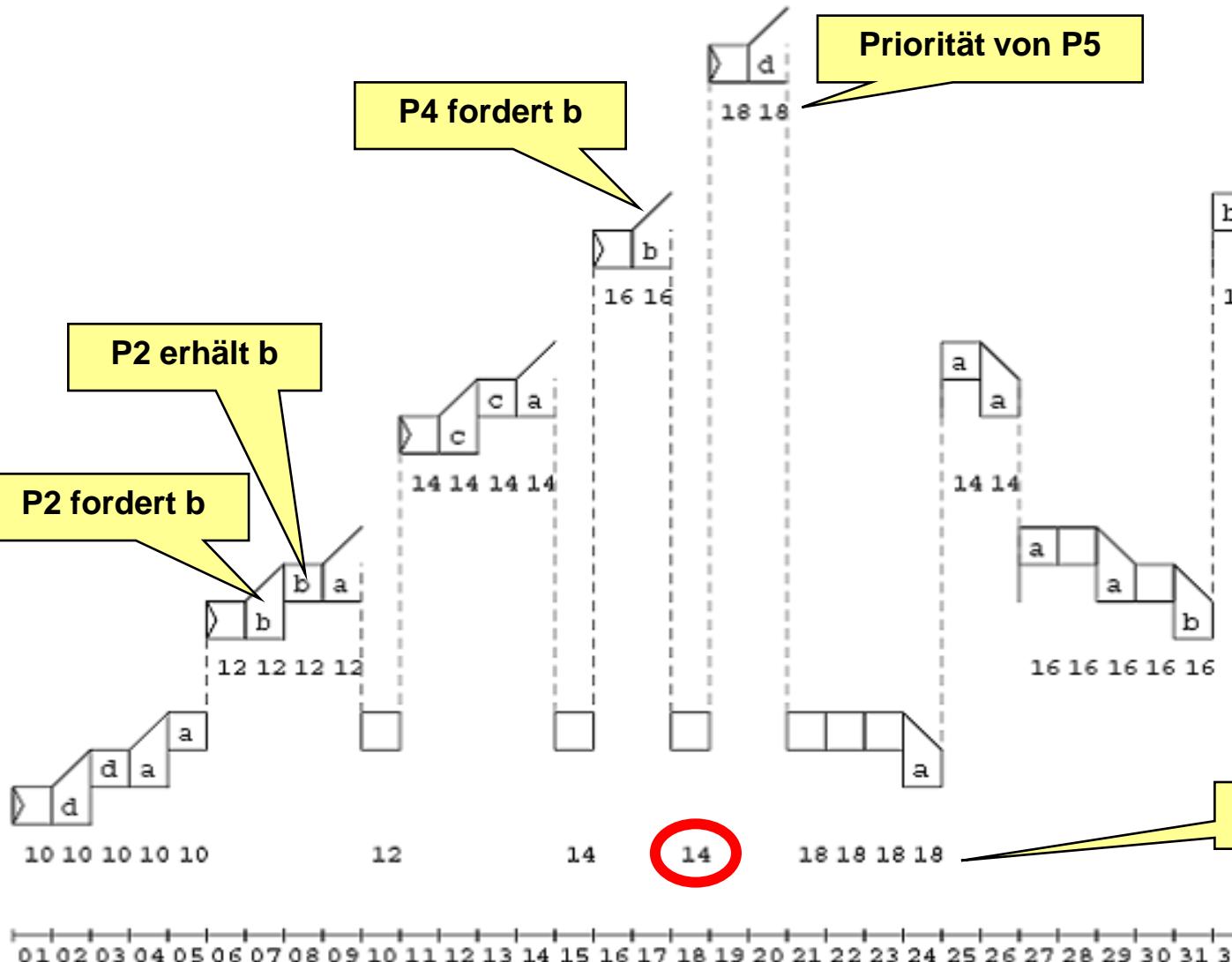
12

14

14

18 18 18 18

Priorität von P1

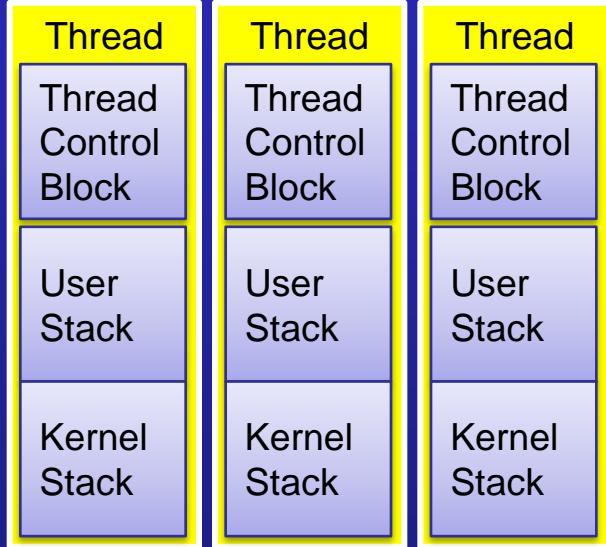


5.3. Prozesse

- Prozesse repräsentieren die Instanzen von laufenden Programmen
 - Start eines Programms bedeutet Start eines Prozesses
- Virtueller Adressraum pro Prozess (User Space)
- Menge von Threads pro Prozess
 - Kernel Threads
 - Bilden einen „Thread Pool“
 - Dienen als „Träger“ von User-Threads
 - User Threads
 - Unsichtbar für das Betriebssystem
 - Werden auf Kernel-Threads abgebildet
- Aufgaben des Betriebssystems
 - Ausführung der Prozesse (und/oder Threads) im Wechsel
 - Ressourcenzuteilung
 - Interprozesskommunikation
- Ursprünglich waren Prozesse die Ausführungseinheit. In vielen modernen Systemen werden Threads scheduliert.
(Linux: Threads ↔ Leichtgewichtsprozesse mit einem Thread)

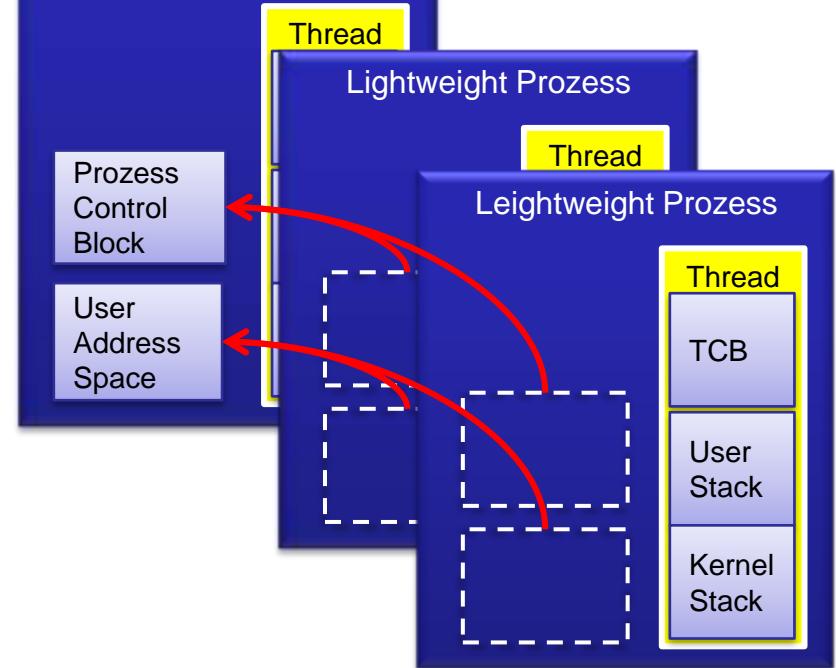
Prozessmodelle

Multi-threaded Prozess



Mehrere Threads in einem Prozess
(z.B. Windows)

Single-Thread Prozess



Mehrere Threads verteilt auf
„Leichtgewichtsprozesse“
(z.B. Linux)

Steuerungsstrukturen des Betriebssystems

- Speichertabellen (vgl. Kap. 4)
 - Zuteilung von Hauptspeicher zu Prozessen
 - Zuteilung Sekundärspeicher zu Prozessen
 - Schutzverwaltung gemeinsamer Speicherregionen
 - Verwaltung des virtuellen Speichers
- I/O Tabellen (vgl. Kap. 3)
 - Verfügbarkeit Gerät
 - Status der I/O Operationen
 - Ports / Memory Mapped IO Bereich
- Dateitabellen / File Management System (vgl. Kap. 2)
 - Verfügbarkeit Files
 - Zugriff Sekundärspeicher
 - Status / Attribute
- **Prozesstabellen**

5.3.1. Threads

- Konzeptuelle Aufspaltung eines Gesamtprozesses in separate Abläufe: Threads
- Besitzt ein System nur Threads, so spricht man auch von Leichtgewichtsprozessen
- Jeder Thread besitzt seinen eigenen Prozeduraktivierungsstack
- Threads interoperieren im gemeinsamen Adressraum („shared memory“).
- In der (OO-)Programmierung kommunizieren Threads auf gemeinsam verwendeten Objekten („shared objects“) durch
 - Referenzen (Zeiger)
 - Methodenaufrufe
 - Nachrichten

Beispiel: Parallel Matrixmultiplikation

MatMulHObjR = OBJECT

VAR done: BOOLEAN;

A,B,C, Stride, IncC, StrideC, RowsA, RowsB, Cols: LONGINT;

PROCEDURE & InitR(A,B,C,Stride,IncC,StrideC,RowsA,RowsB,Cols: LONGINT);

BEGIN

done := FALSE;

SELF.A := A; ... copy all values ...

Konstruktor, wird vor Prozessbeginn aufgerufen

END InitR;

PROCEDURE Compute;

BEGIN {EXCLUSIVE}

MatMulHBlockR(A,B,C,Stride,IncC,StrideC,RowsA,RowsB,Cols);

done := TRUE;

END Compute;

Wechselseitiger Ausschluss (Monitor)

PROCEDURE Wait*;

BEGIN {EXCLUSIVE}

AWAIT(done);

END Wait;



Condition (wird sofort geprüft und sobald ein Prozess den Monitor verlässt).

BEGIN {ACTIVE} Compute;

END MatMulHObjR;

Aktives Objekt: eigener Thread

Beispiel (2)

```
VAR proc: ARRAY nrProcesses OF MatMulHObjR;
```

```
...
```

```
FOR i := 0 TO nrProcesses - 1 DO
```

```
    to := ColsB * (i + 1) DIV nrProcesses;
```

```
    adrB := B + from * stride; adrC := C + from * IncC;
```

```
    NEW( proc[i], A, adrB, adrC, stride, IncC, StrideC, RowsA, to - from, RowsB );
```

```
    from := to;
```

```
END;
```

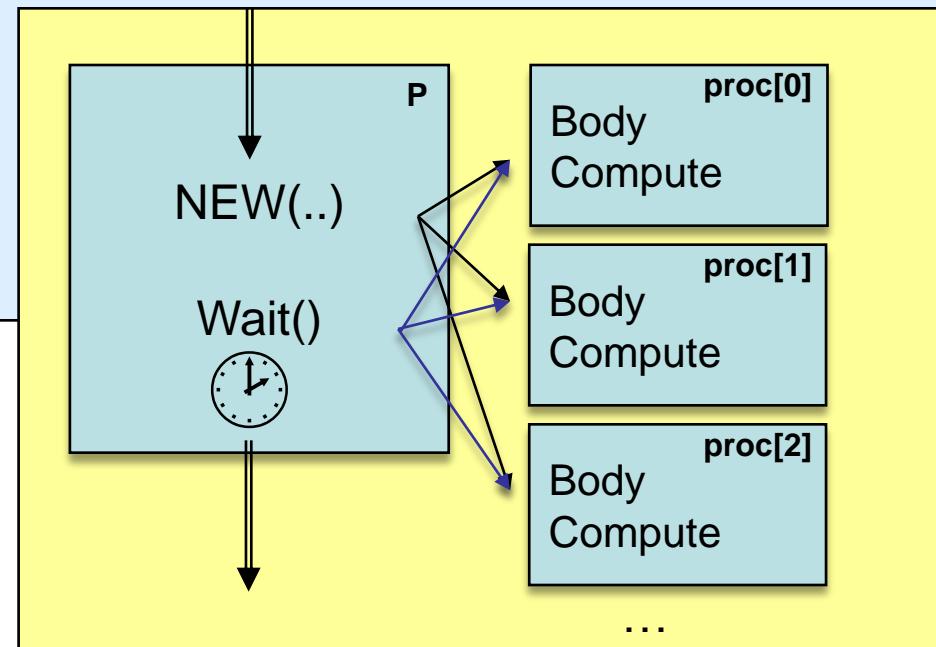
```
FOR i := 0 TO nrProcesses - 1 DO
```

```
    proc[i].Wait();
```

```
END;
```

Mögliche Prozess-Zustände:

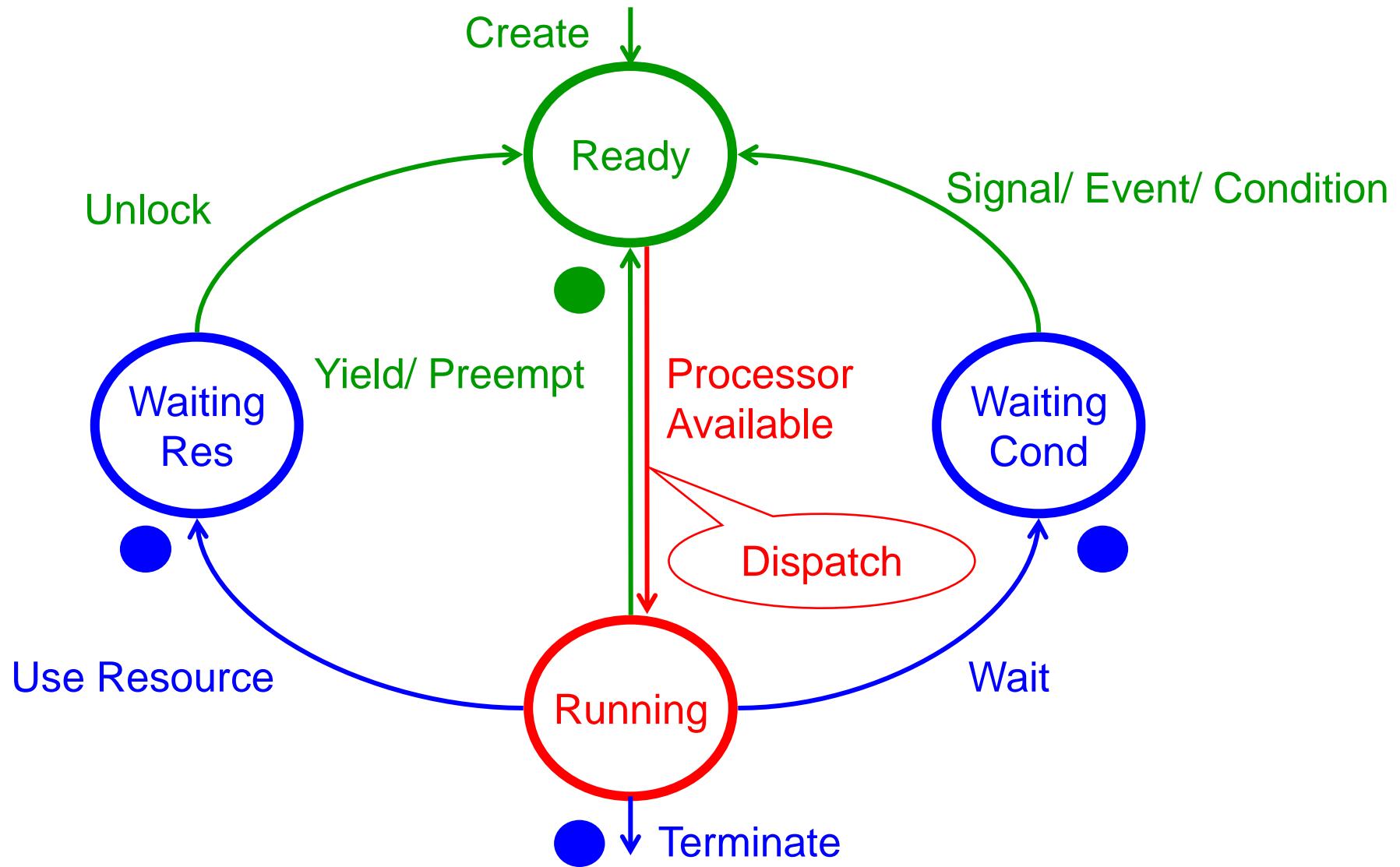
- Läuft
- Wartet auf Bedingung
- Wartet auf Zutritt
- Wartet auf Prozessorzeit
- Ist Beendet



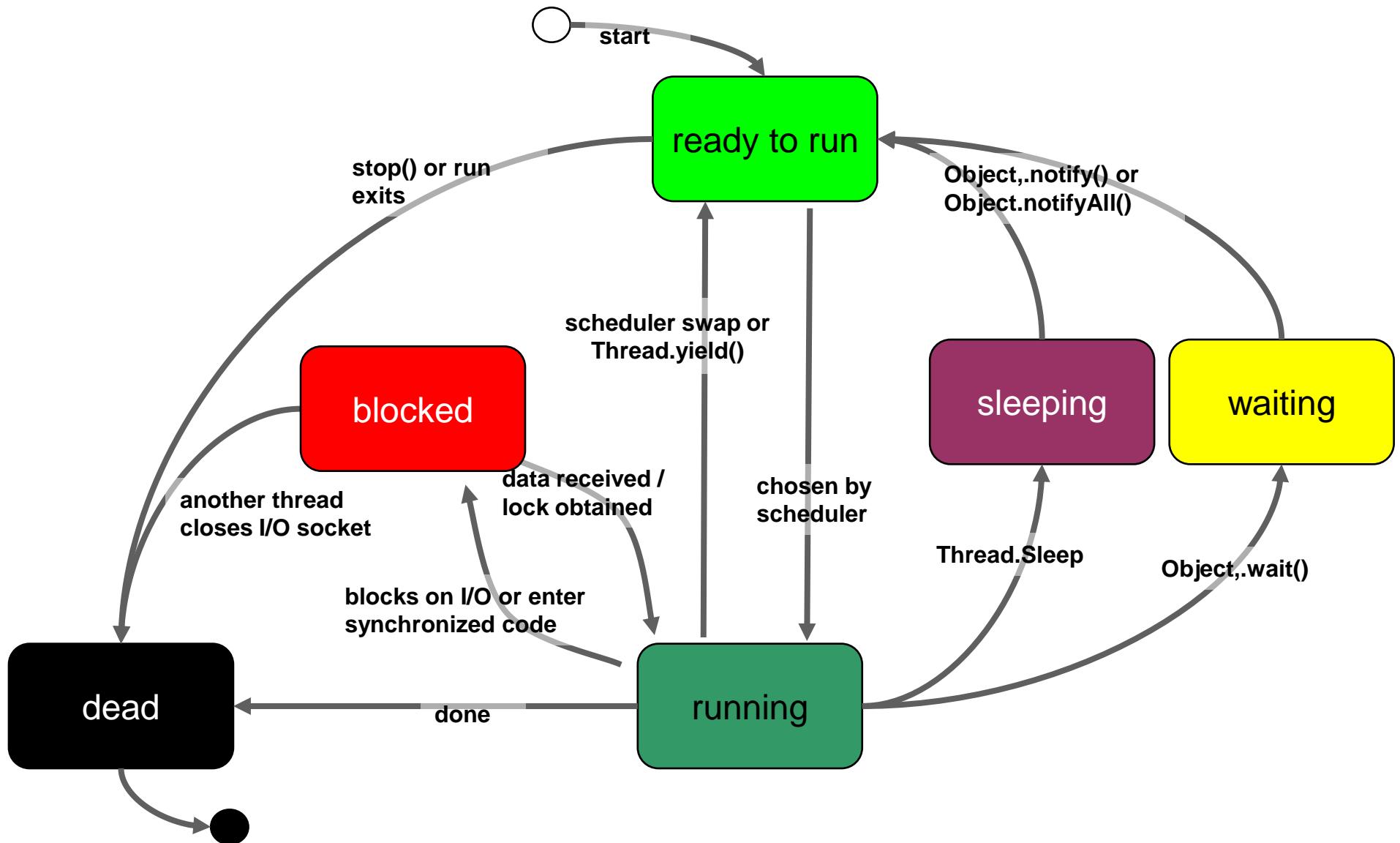
Operationen

- Allgemeine Operationen
 - Create und Exit
 - Prozessinkarnation und reguläres Ende
 - Suspend und Resume
 - Unterbrechung der Ausführung, möglicherweise zeitbeschränkt (Timer)
 - Wait/Await und Signal/Pulse
 - Mechanismus zum Warten auf eine Bedingung
- Spezielle Operationen
 - Cancel
 - Threadbeendigung erwünscht aber nicht garantiert (Cancellation signal kann maskiert werden).
 - Join
 - Thread wartet beim Beenden auf andere Threads (blockiert bis zum Beenden).

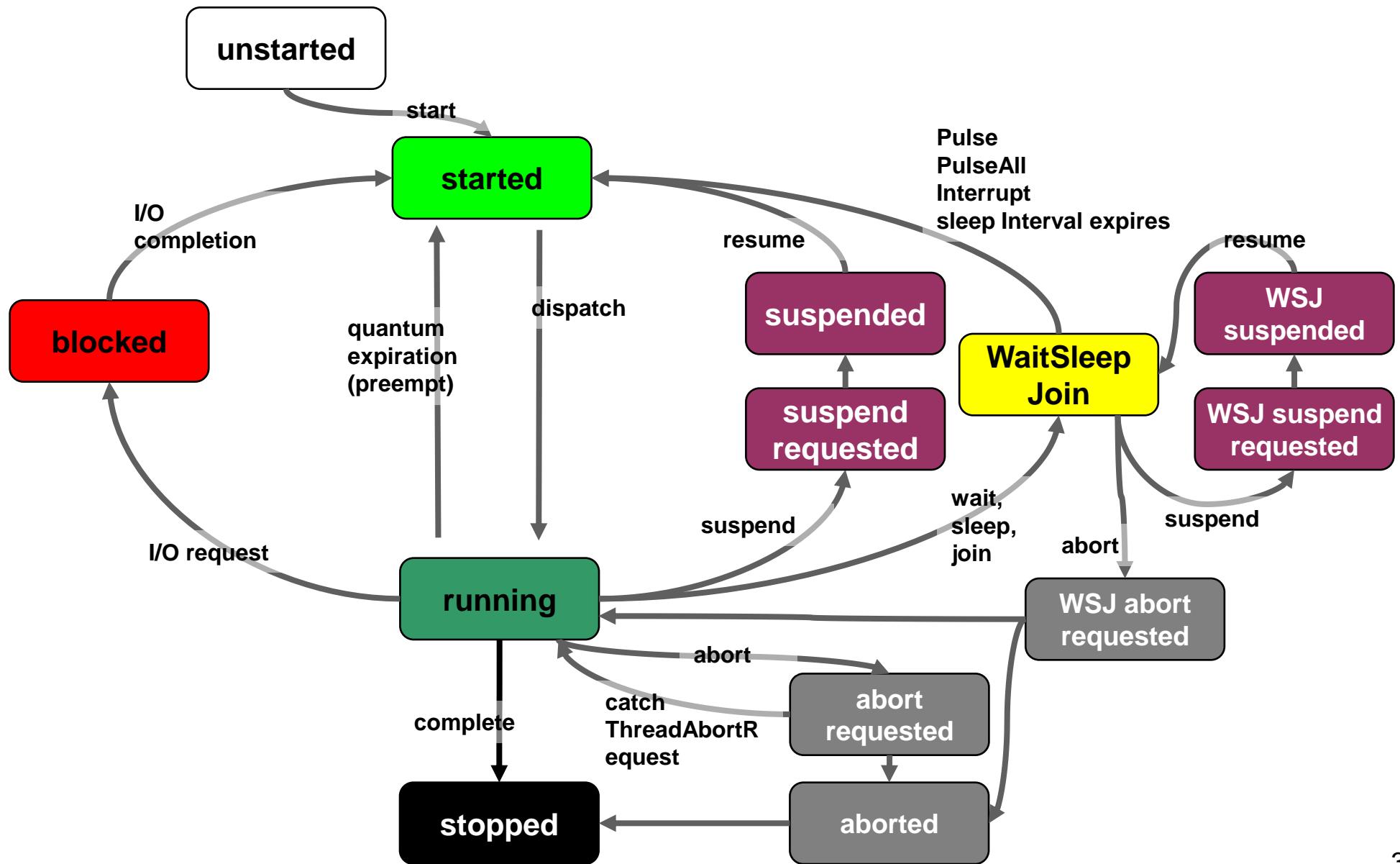
5.3.2. Lebenszyklen von Threads



Beispiel: Lebenszyklen von Java Threads



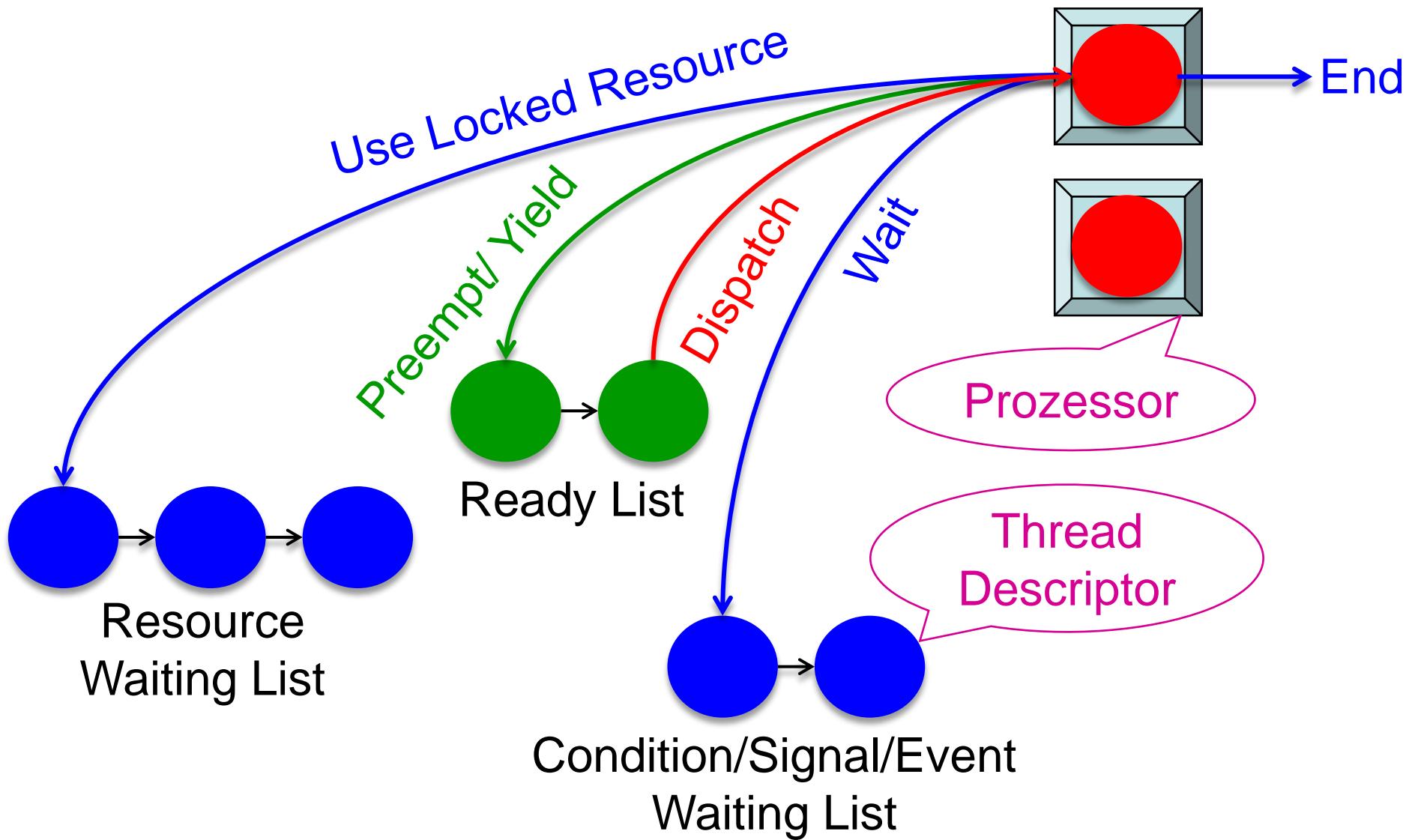
Beispiel: Lebenszyklen von .NET Threads



5.3.3. Thread Management

- Erzeugen
- Einfügen in Wartelisten
 - Resourceneingangsliste
 - Signalwarteliste
 - Readyliste
- Preemption
 - Unterbruch bzw. vorzeitiger Abbruch
- Dispatching/ Scheduling
 - Zuordnung zu Prozessoren (CPU)

Laufzeit Datenstruktur



Thread Kontextwechsel

- Basiert auf Thread Control Block TCB zur Speicherung des aktuellen Threadzustandes
 - Stack Pointer
 - Program Counter
 - Prozessor Zustandsregister
 - Adressregister, Datenregister
 - Gleitkommaregister
- Kontextwechsel =
{
 Abspeichern aktueller Threadzustand in TCB_{old};
 Laden des neuen Threadzustandes von TCB_{new}
}

Beispiel: Runtime Daten im Prozess-Objekt in A2

Process* = OBJECT(..)

VAR

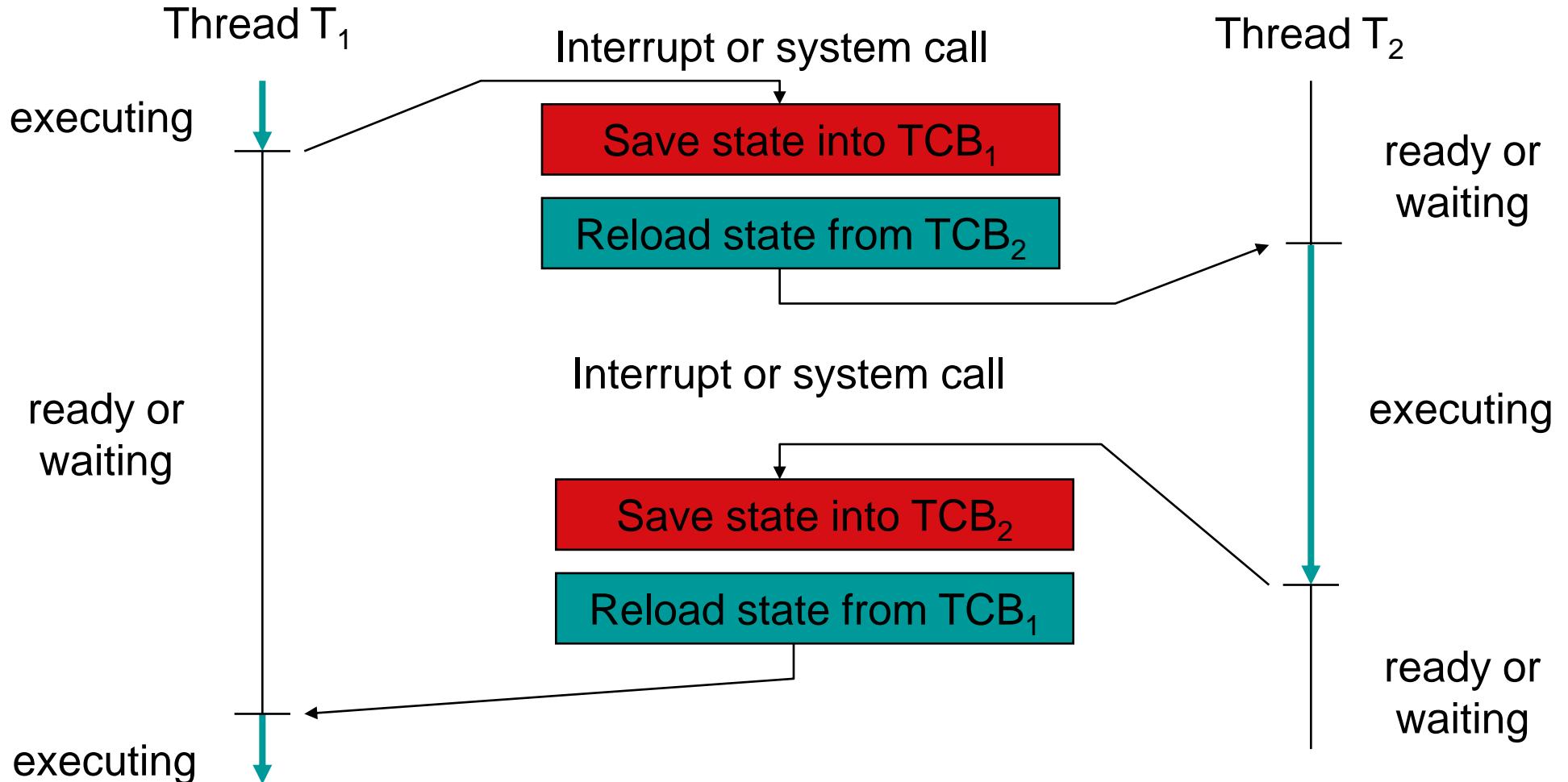
next,prev,link: Process;
obj*: ProtectedObject;
state*: Machine.State;
condition*: Condition;
condFP*: LONGINT;
mode*: LONGINT;
procID*: LONGINT;
waitingOn*:ProtectedObject;(*
flags*: SET;
priority*: LONGINT;
stack*: Machine.Stack;
exp*: Machine.ExceptionState;

```
State* = RECORD  
    EDI*, ESI*, EBX*, EDX*, ECX*, EAX*: LONGINT;  
    INT*, EBP*, EIP*, CS*: LONGINT;  
    ....  
END;
```

(* in active processQueue *)
(* associated active object *)
(** processor state of suspended process *)
(** awaited process' condition *)
(** awaited process' condition's context *)
(** process state *)
(** processor ID where running *)
(** obj this process is waiting on (lock or
condition) *)
(** process flags *)
(** process priority *)
(** user-level stack of process *)

...

CPU Switch von Thread zu Thread



*aus: Windows Operating System Internals Curriculum Development Kit,
David A. Solomon und Mark E. Russinovich*

Preemption

- Thread Unterbrechung durch das System
 - Bei Processor Sharing via „Time Slicing“: Ablauf des zugestandenen „Zeitquants“



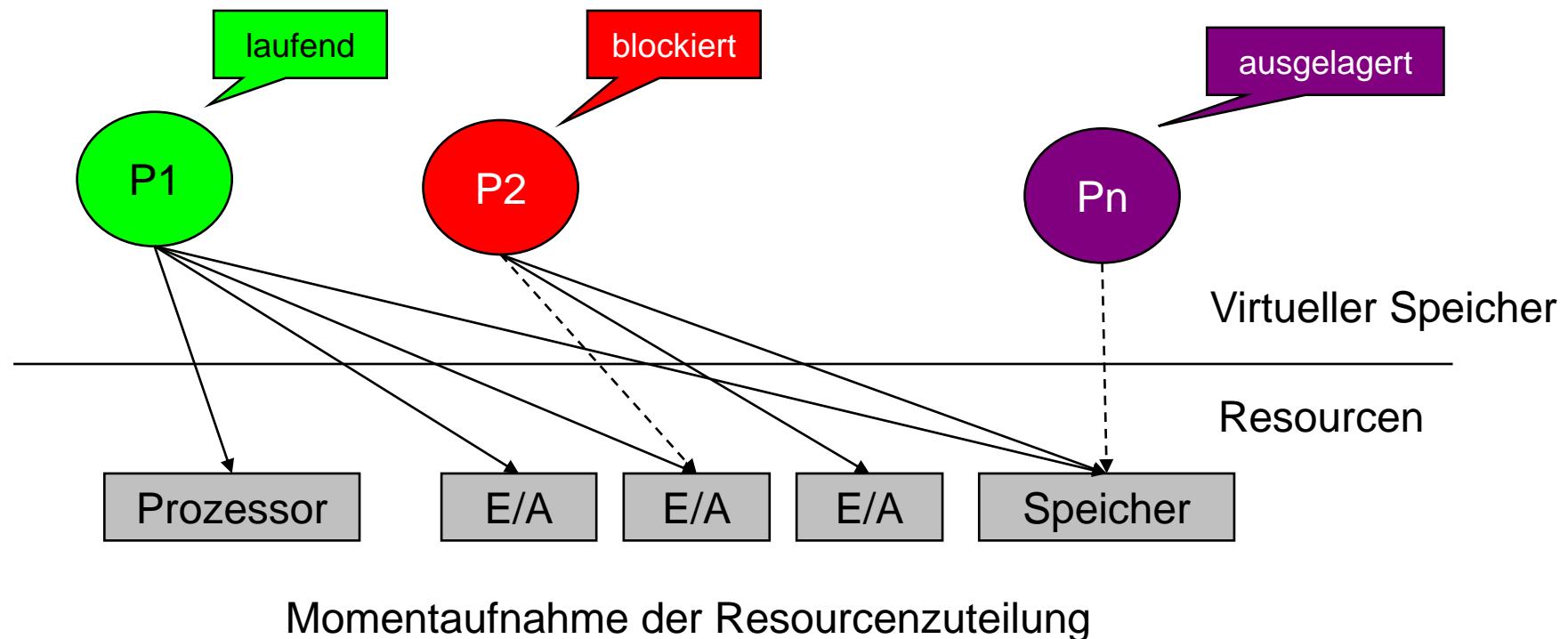
- Falls Threads Prioritäten zugeordnet sind: Bedarf durch Thread höherer Priorität
 - Invariante

$$\text{Max(Priorität in Ready List)} \leq \text{Min(Priorität in Ausführung begriffener Threads)}$$

5.3.4 Prozessbeschreibung

Betriebssystem

- plant Ablauf von Prozessen
- übernimmt Zuteilung der Prozesse für Ausführung auf dem Prozessor
- weist Prozessen Ressourcen zu.

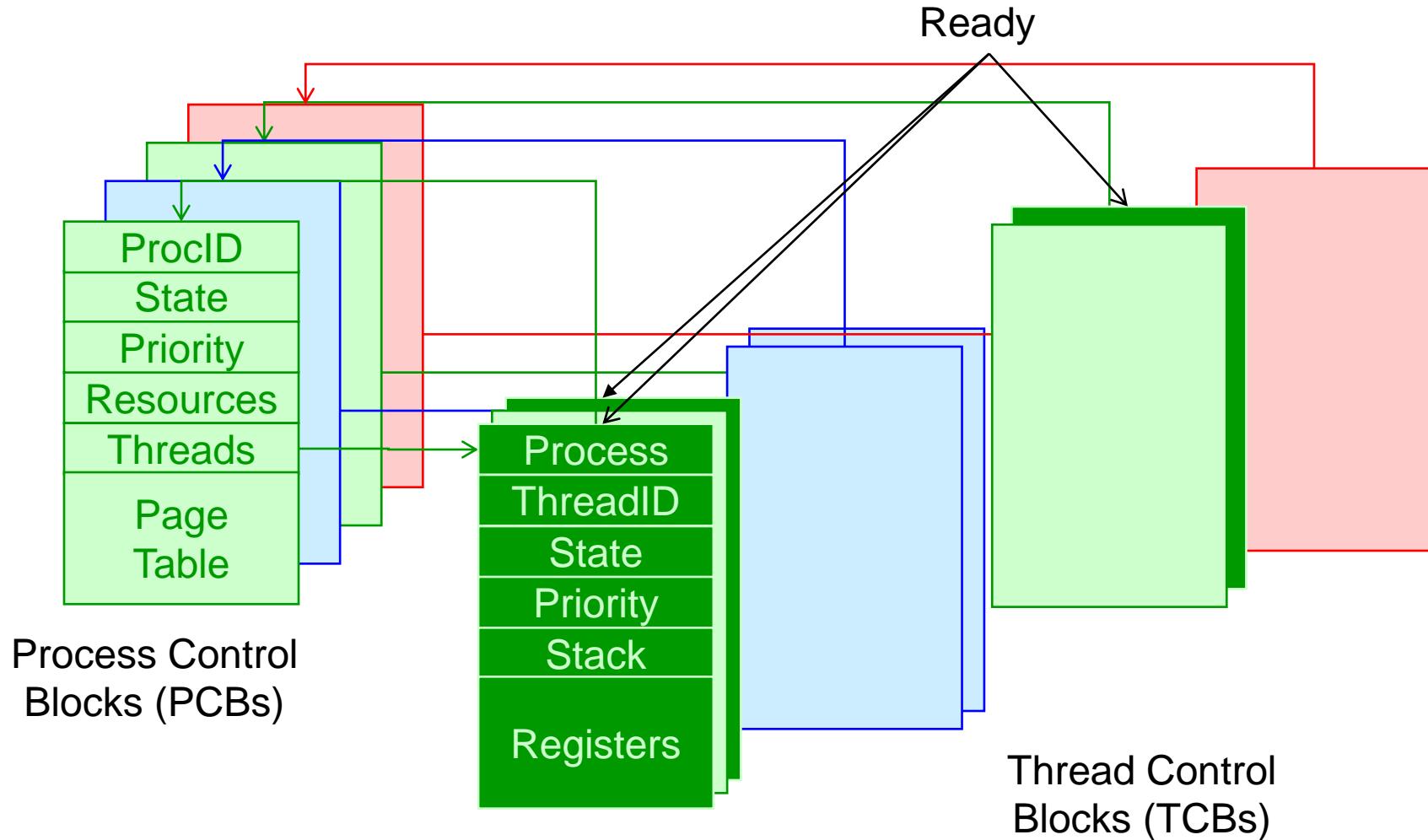


Prozesskontrollblock (PCB)

- Prozessidentifikation
 - Kennung des Prozesses, Benutzerkennung, Elternprozess
- Prozessorstatusinformationen
 - entweder
 - Allgemeine Register, Steuer- und Statusregister (Programmzähler, Zustandsregister, Statusregister)
 - Stack-Pointer
 - oder mittelbar durch Liste der assoziierten Threads
- Prozesskontrollinformationen
 - Scheduling und Zustand
 - Prozesszustand: aktiv/bereit/wartend o.ä.
 - Priorität & Informationen für das Scheduling: Zeitdauer o.ä.
 - Event: Events (Ereignisse), auf die der Prozess wartet
 - Datenstrukturierung (Prozessverkettung in Queues)
 - Interprozesskommunikation (Flags, Signale, Nachrichten)
 - Prozessprivilegien (zugreifbarer Speicherplatz, erlaubte Befehlsarten)
 - Speicherverwaltung (Zeiger auf Segment- oder Seitentabellen)
 - Resourcenbesitz- und Nutzung (geöffnete Dateien etc.)

sofern Schedulig
nicht via Kernel-
Threads

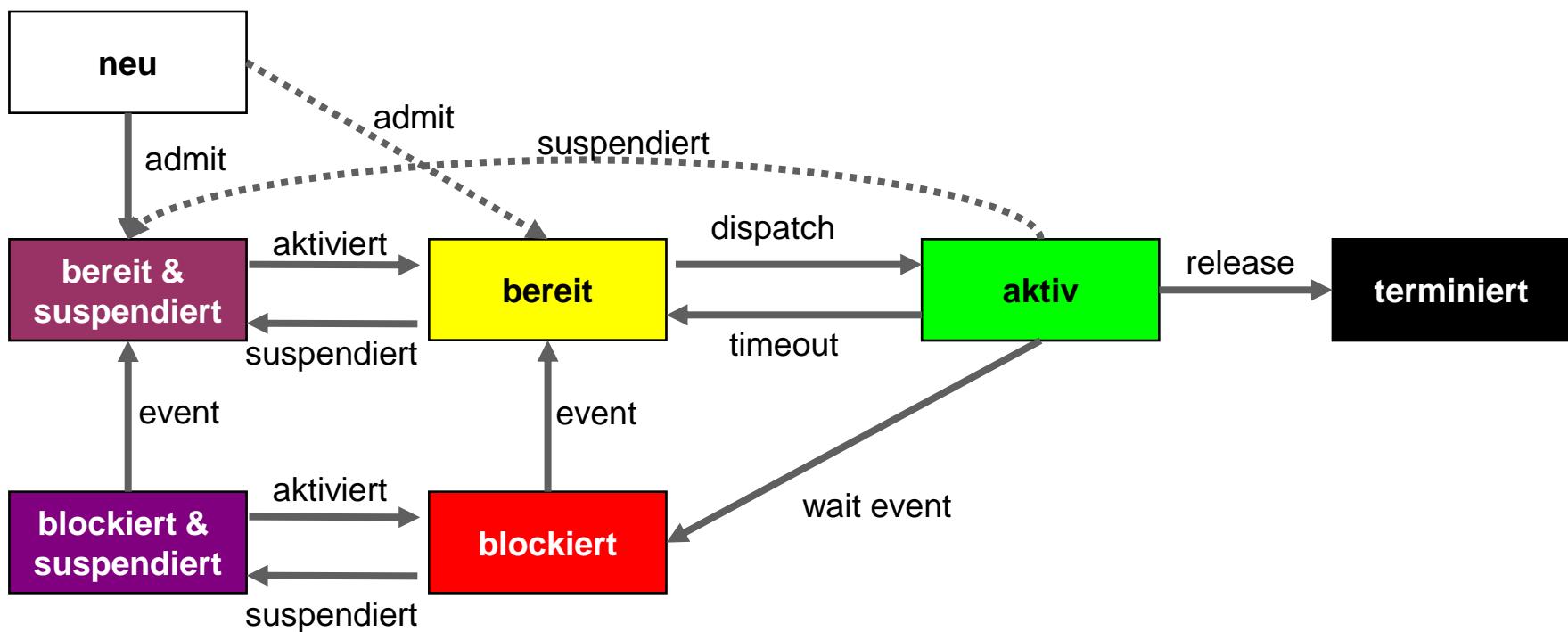
Laufzeit Datenstruktur bei Multithreading und Multiprocessing



Suspendierte/Ausgelagerte Prozesse

■ Gründe für Suspendierung

- Auslagerung/Swapping
- Benutzeranforderung
- Timing
- Anforderung durch Elternprozess
- Andere Gründe des Betriebssystems



5.3.5. Prozesssteuerung

Prozesserzeugung

- Zuweisung einer eindeutigen Kennung
- Speicherplatzzuteilung
- Initialisierung des Prozesskontrollblocks
- Integration in dynamische Datenstrukturen
(Warteschlangen etc.)
- Erzeugung weiterer Datenstrukturen (sofern nötig)

Erzeugung

- aus dem Nichts („Create“) oder
- durch Cloning (Vergabelung / „Fork“) [Unix].

Forking

- Abspaltung eines mit dem laufenden “Parent” Prozess identischen “Child” Prozesses



Ablauf:

- Cloning des Prozess Control Blocks
- Cloning der Seitentabelle statt Speicherinhalt dank **Copy-on-Write** (“Lazy-Copying”)
 - Alle Seiten mit “read-only” Flag markieren
 - Bei Schreibzugriff durch parent oder child
 - Page-Fault
 - Seitenkopie erzeugen

Forking Codebeispiel

```
int pid, j, i;
pid = fork();
if (pid == 0)
{   for (j=0; j < 5; j++)
    { printf ("Kind: %d\n", j); sleep (1); }
    exit (0);
}
else if (pid > 0)
{
    for (i=0; i < 5; i++)
    { printf ("Vater: %d\n", i); sleep (1); }
    exit (0);
}
else
{
    fprintf (stderr, "Error");
    exit (1);
}
```

= 0 ⇔ Child
= pid ⇔ Parent
< 0 ⇔ failed

```
Vater: 0
Kind: 0
Vater: 1
Vater: 2
Kind: 1
Kind: 2
Kind: 3
Vater: 3
Vater: 4
Kind: 4
>_
```

Scheduler (Ablaufplanung)

- Long-term scheduler (job scheduler)
 - wählt Prozesse mit ihren Threads zum Eintrag in die Ready Queue aus.
 - berücksichtigt Speicher Management (ausgelagerte Prozesse)
 - kontrolliert das Mass des Multiprogramming
 - langsam, nicht oft ausgeführt
- Short-term scheduler (CPU scheduler)
 - wählt den nächsten ausführbaren Thread aus und alloziert die CPU
 - oft ausgeführt, muss schnell sein
 - trifft Entscheidungen, auch bei nicht-preemptivem Wechsel:
 1. Threadwechsel “running” → “waiting”
 2. Threadwechsel “running” → “ready”
 3. Threadwechsel “waiting” → “ready”
 4. Thread wird beendet

Prozesswechsel

Prozesswechsel findet statt via

- Interrupt
 - Timer Interrupt
 - Zeitquantum abgelaufen: Preemption
 - I/O Interrupt
 - Kann Event darstellen, auf den Prozesse warten
 - Page Fault
 - beim Einsatz von Virtual Memory
- Trap (Fehler / Exception)
- Supervisor Call

außerhalb der aktuellen Befehlsausführung

implizit verursacht durch aktuelle Befehlsausführung

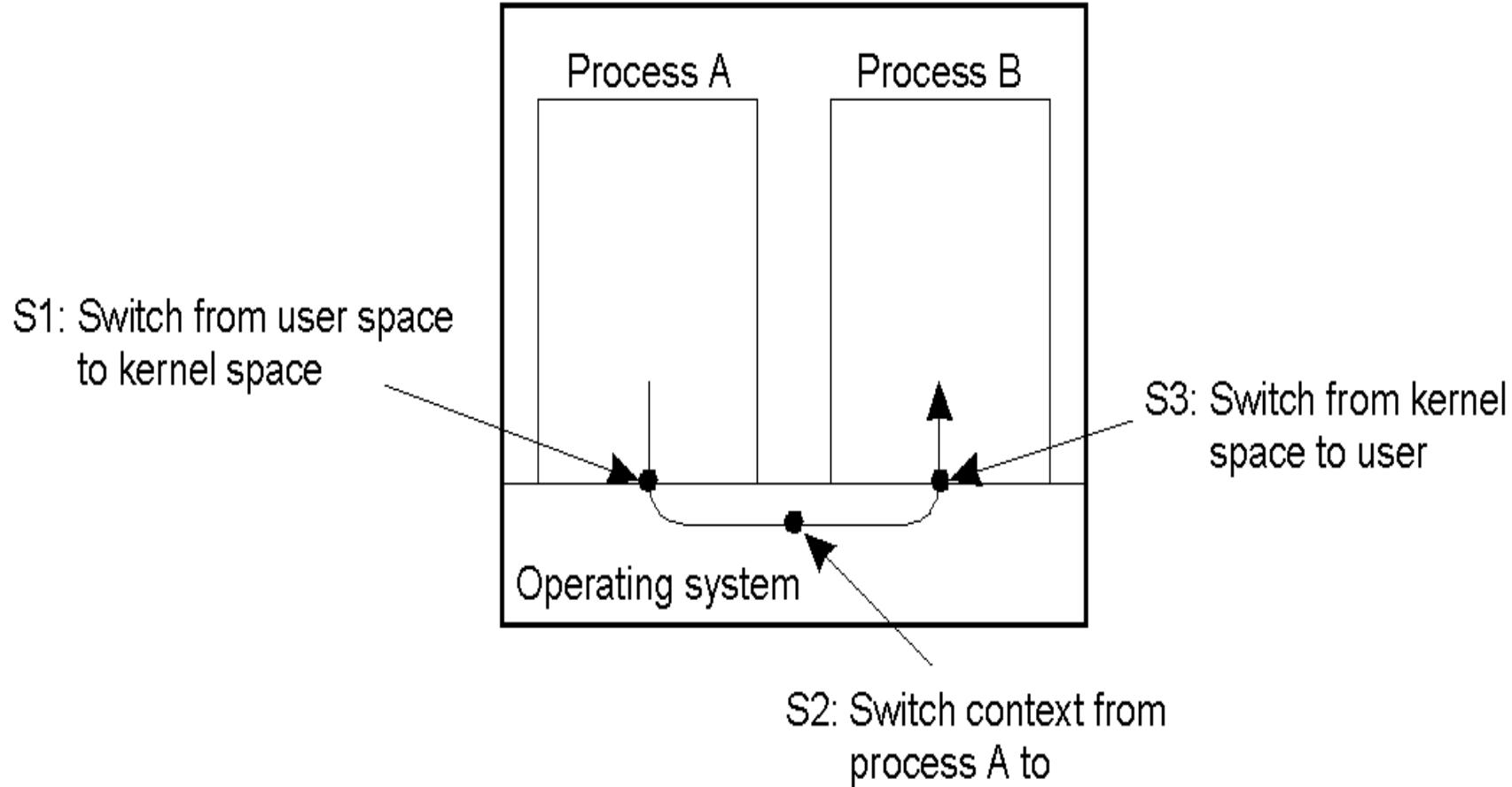
explizit verursacht durch aktuelle Befehlsausführung

Moduswechsel vs. Änderung des Prozesszustands (Prozesswechsel)

- Moduswechsel:
 - nötig beim Wechsel zum Kernel Mode
 - sofern Prozess/Kernelthread dabei nicht gewechselt wird ist nur Speicherung des Programmkontexts nötig.
- Threadwechsel ohne Prozesswechsel
Kontextwechsel = { Abspeichern des alten TCBs; Laden des neuen TCBs }
- Änderung des Prozesszustands beinhaltet hingegen
Kontextwechsel = {
 - Speicherung der alten TCBs;
 - Speicherung des alten PCB;
 - // Umschalten der Seitentabelle
 - Laden des neuen PCB;
 - Laden der neuen TCBs;**}**



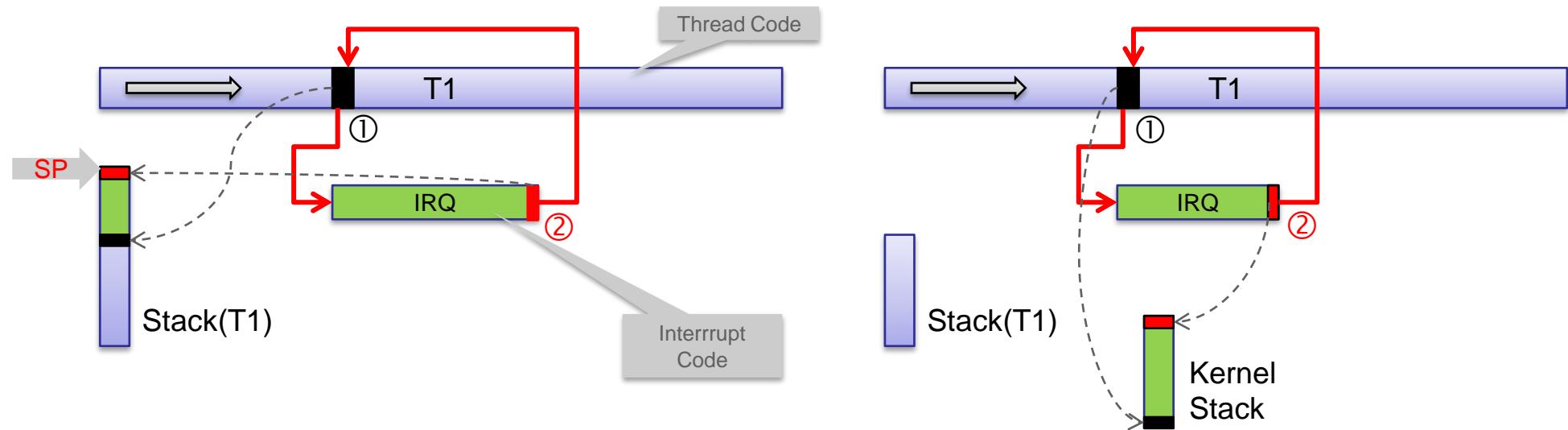
Prozess Kontextwechsel



Prozessterminierung

- Letzter Thread eines Prozesses übergibt Kontrolle an das Betriebssystem (exit)
 - Elternprozess erhält Rückgabe-Code (via wait)
 - Prozessressourcen werden durch das BS freigegeben
- Elternprozess terminiert einen Kindsprozess (kill)
 - Kind hat Resourcenzuteilung nicht beachtet
 - Kindsprozess hat keine Aufgabe mehr zu erfüllen
 - Elternprozess beendet
 - Das BS erlaubt das Weiterlaufen eines Kindprozesses ohne Elternprozess nicht (i.A. konfigurierbar).
 - Rekursives Verhalten

Interrupts mit und ohne Kernel Stacks



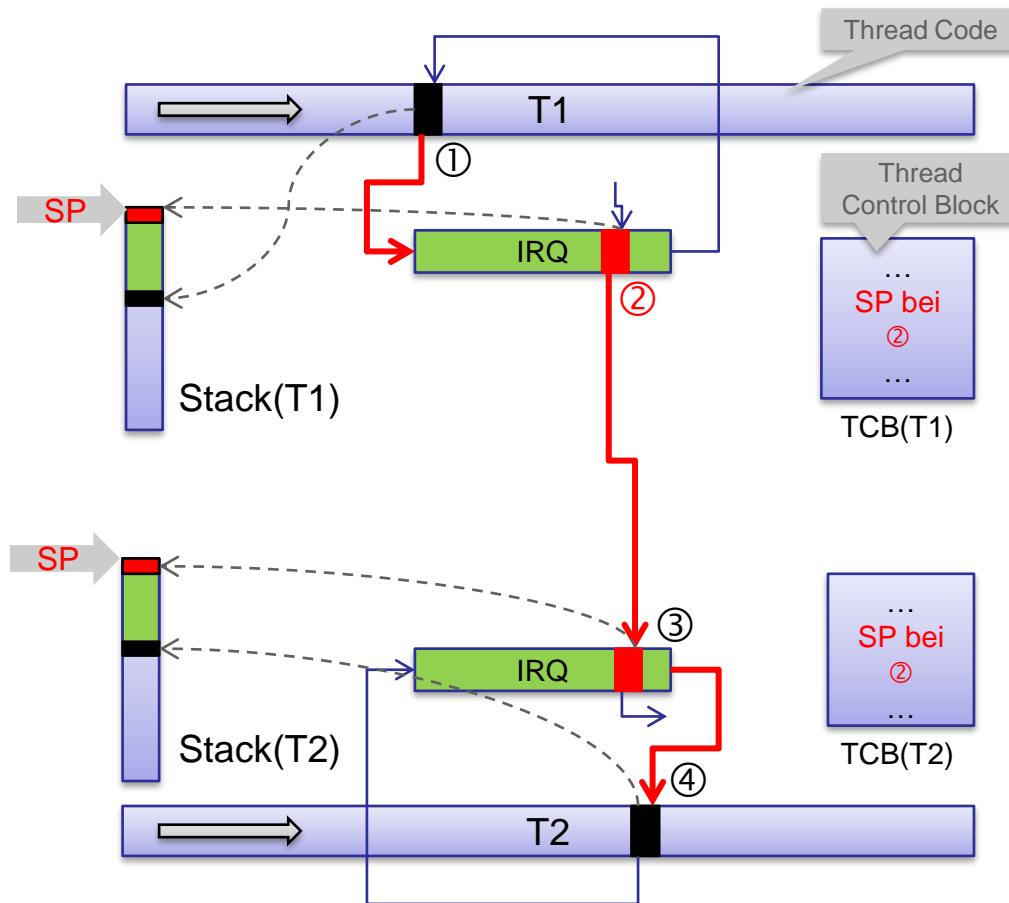
Interrupt ohne Kernel Stacks

- Code läuft auf User Stack weiter
- Einfach zu implementieren
- Betriebssystem hat keine separate Kontrolle über den Stack
- Keine Stackgrößengarantie beim Eintritt in den IRQ

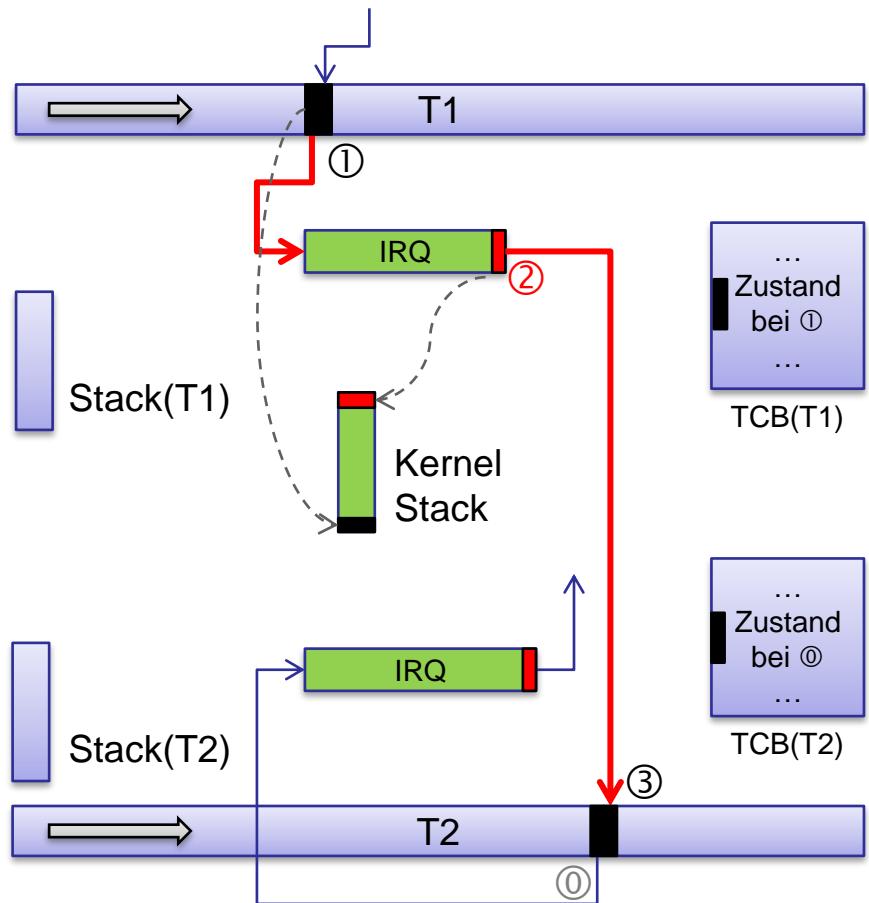
Interrupt mit Kernel Stacks

- Bei Intel/AMD unterstützt durch automatisches Umschalten des Stack-Pointers bei Privilegienwechsel (spezifiziert im Task State Segment, welches via Task Register lokalisiert wird)
- Schutz des Kernel Stacks durch Segmentierung und/oder Paging

Threadwechsel mit und ohne Kernel Stacks



Threadwechsel ohne Kernel Stacks:
Stack Pointer zum Zeitpunkt ② wird im TCB gespeichert.
Prozessorzustand zum Zeitpunkt ① im User Stack



Threadwechsel mit Kernel Stacks:
Prozessorzustand zum Zeitpunkt ① wird im TCB gespeichert

5.3.6 User- und Kernel-Level Threads

- In schwergewichtigen Systemen gibt es eine Unterscheidung zwischen Prozessen und Threads.
- Darüberhinaus sind Zugriffe auf das Betriebssystem in solchen Systemen nur über System-Calls via Interrupt möglich
- Daraus resultiert die Unterscheidung und Notwendigkeit von User- und Kernel-Level Threads.
- In schlanken Systemen verschwimmen die Grenzen und die Vorteile beider Konzepte kommen beim direkten Einsatz der Leichtgewichtsprozesse zum Tragen.

User-Level Threads

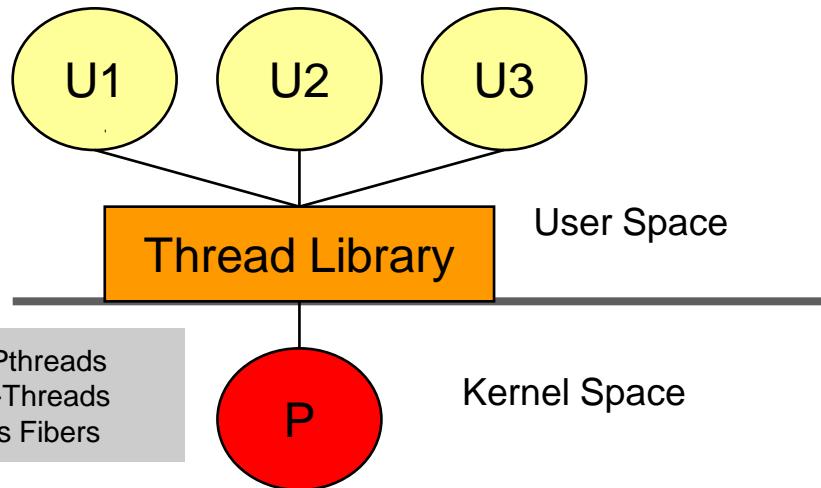
- Benutzer Threads werden mit einer Thread Library im User-Level (außerhalb des Kerns) implementiert
 - Prozess oder Kernel-Level Thread ist die Einheit, die vom BS der CPU zugeteilt wird wird.
- Beispiele
 - POSIX *Pthreads* (*in manchen Implementationen*), C-threads (bei Mach), Solaris threads, Fibers bei Windows
- ☺ Schneller Thread-Wechsel, schnelle Synchronisation (keine Kernel-Modus Privilegien benötigt)
- ☺ Scheduling kann anwendungsspezifisch ausgelegt sein
- ☺ Kann auf jedem Betriebssystem auf Library-Ebene implementiert werden
- ☹ Keine Unterstützung mehrerer Prozessoren (Zuteilung zu genau einem Prozess).
- ☹ Probleme beim Blockieren durch verweigerte Resourcenzuteilung

Kernel-Level Threads

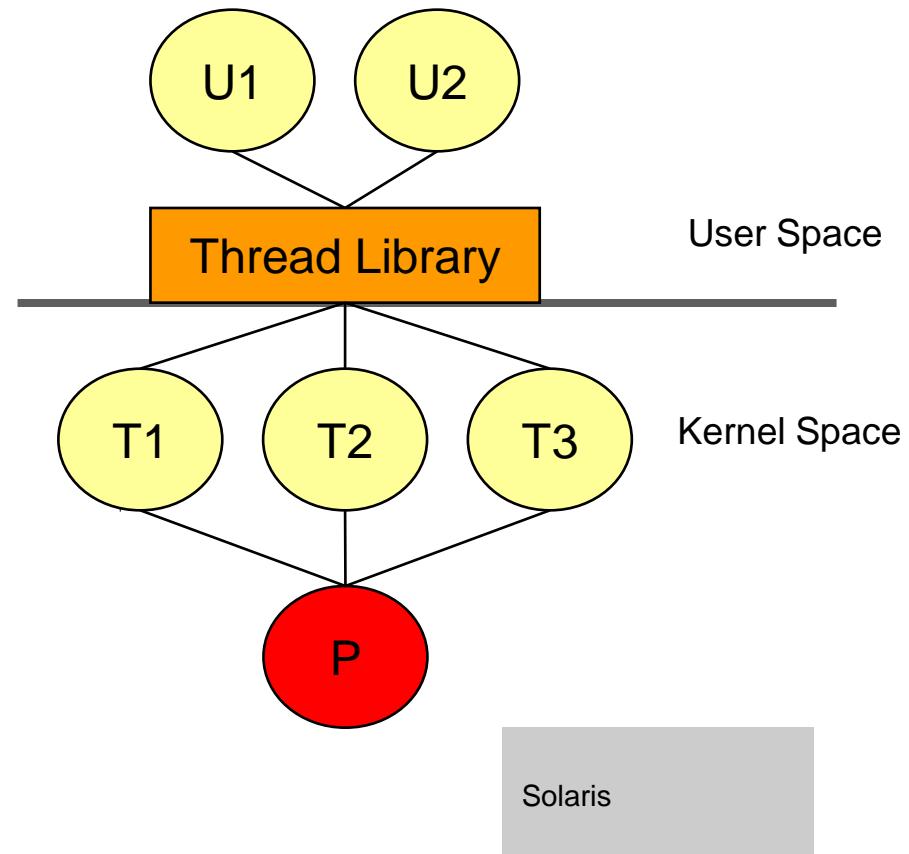
- Kernel Threads werden vom Betriebssystem bereitgestellt.
Ablaufplanung durch den Kern.
 - Der Thread ist die Einheit, die vom System scheduliert wird.
- Beispiele: Windows Threads, Linux Tasks
- ⌚ Threadwechsel erfordert Moduswechsel zum Kern.
(Synchronisationskonzepte langsamer).
- 😊 Kern kann die Threads verschiedenen Prozessoren zuteilen
- 😊 Durch Ressourcen blockierte Threads blockieren keinen Prozess (Kern weist neuen Thread zu).
- 😊 Kern kann selbst mit Threads implementiert werden.

Verschiedene Ansätze

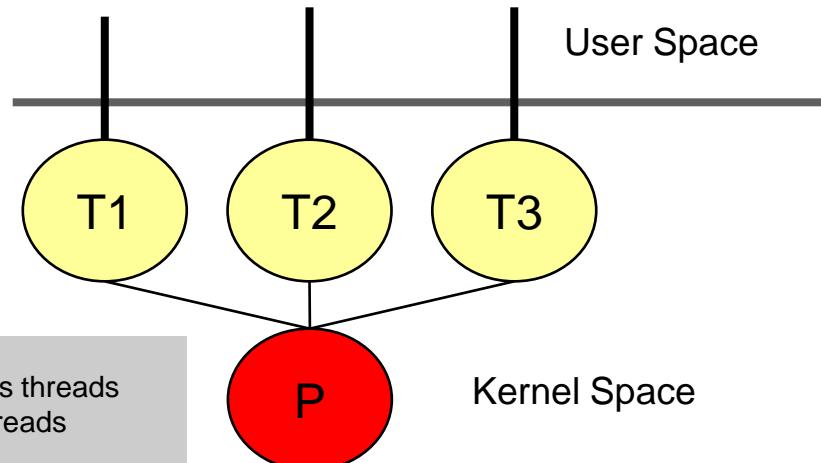
Benutzer Threads („one to many“)



hybrider Ansatz („many to many“)



Kernel Threads („one to one“)

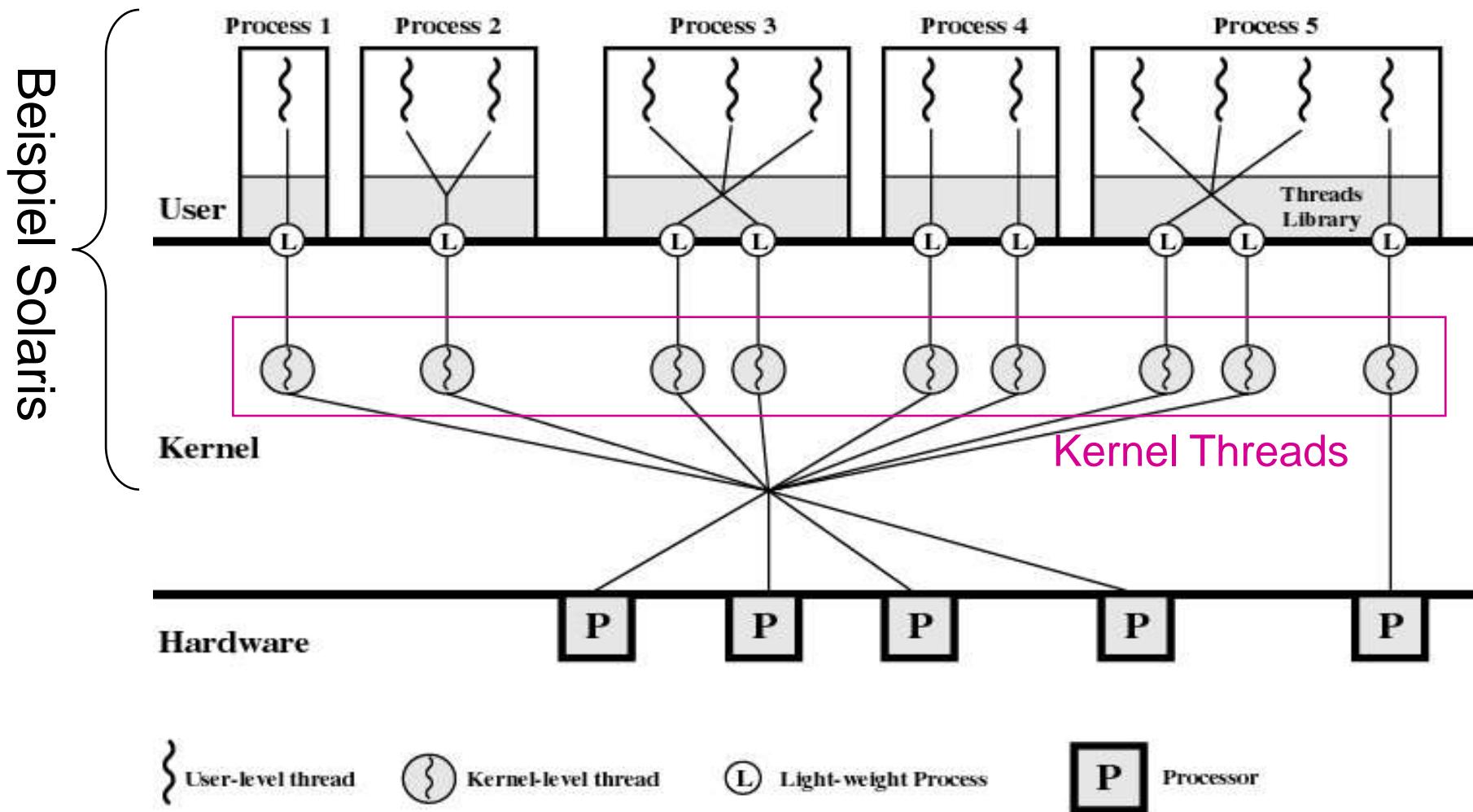


PThreads

- POSIX Standard (IEEE 1003.1c) API für die Threaderzeugung und –Synchronisation
- API spezifiziert das Verhalten der Bibliotheken (nicht die Implementation)
- auf viele UNIX Betriebssystemen implementiert
- Services for Unix (SFU) implementiert PThreads auf Windows

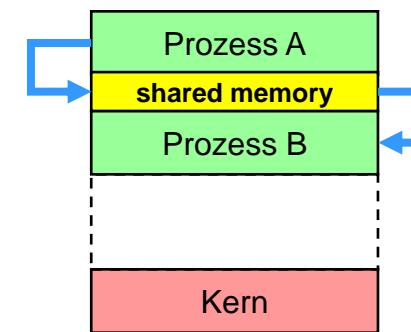
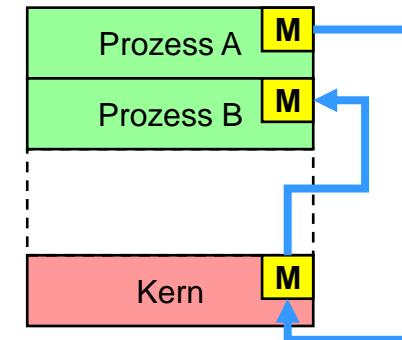
Mehrprozess Systemstruktur

Komplexität verdeutlicht am Beispiel Solaris



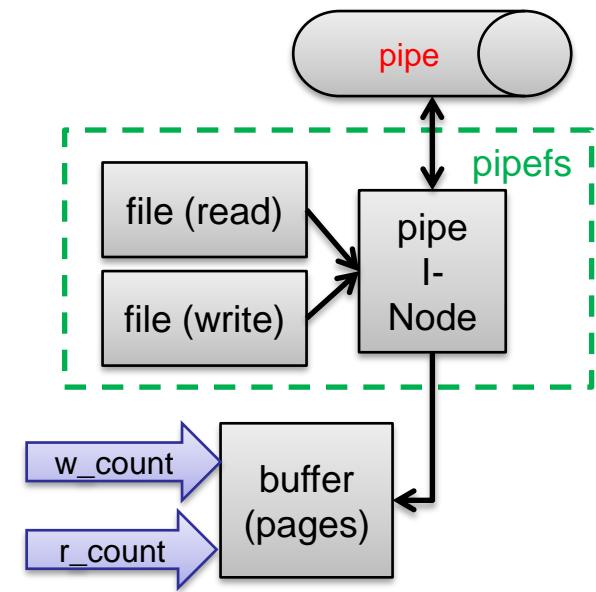
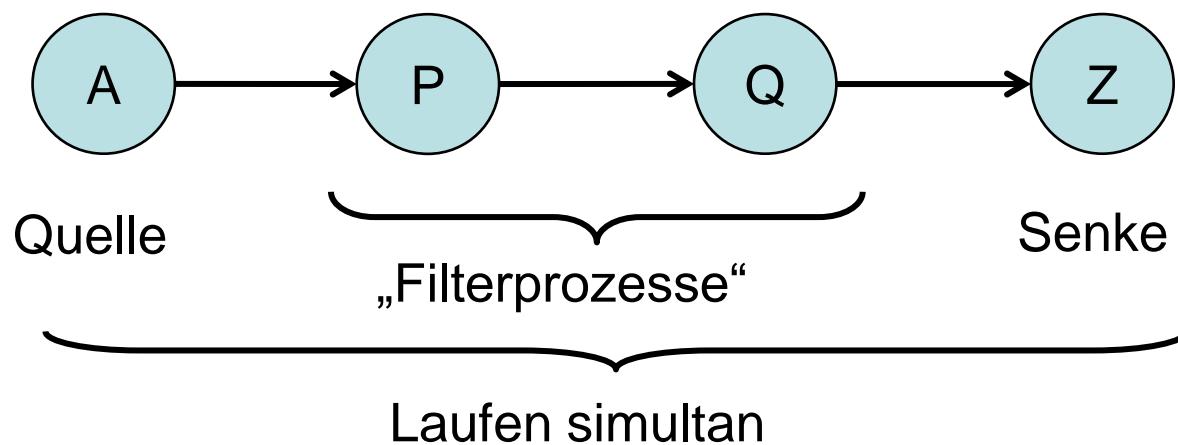
5.3.7 Interprozesskommunikation IPC

- **Message Passing**
 - Synchron und Asynchron
 - via Channels, Connections oder Thread zu Thread
- **Shared Memory**
 - kann als Träger für „bulk messages“ verwendet werden
- Signale
- POSIX Message Queues / Mailboxes
- Sockets
- Pipes, FIFOs, Files
- Semaphoren
- RPCs



Pipes

- “Pipe” ermöglicht Einwegzeichenstrom von einem Prozess zu einem anderen.
- Schreiben und Lesen via read/write System Call
- Üblicherweise realisiert als Consumer-Producer Modell mit einem Ringbuffer
- Verbindet Prozesse zu “Pipeline” A | P | Q | Z
- Pipe privat zwischen zwei Prozessen
- Öffentliche Pipes: Named Pipes = FIFOs



„Message Passing“ Codebeispiel

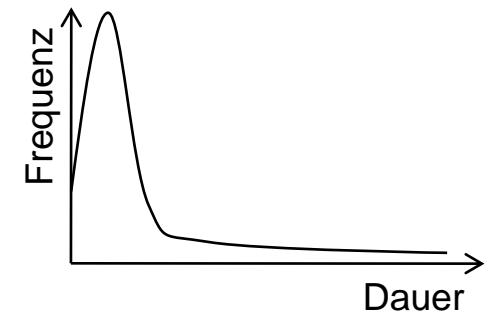
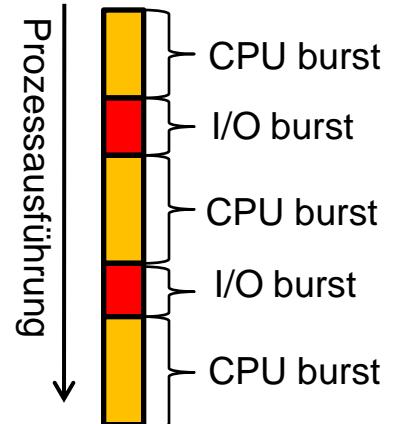
```
■ main() {  
    ...  
    if (!fork()) producer(); else consumer();  
}  
producer() {  
    while (1) { ...  
        produce item nextp;  
        ...  
        msgsnd (consPID,nextp,length);  
    }  
}  
consumer() {  
    while (1) {  
        msgrcv (prodPID,nextc,size,0);  
        ...  
        consume item nextc;  
        ...  
    }  
}
```

Signale

- Signal [Unix] = Benachrichtigung, dass ein bestimmtes Ereignis eingetreten ist
- Asynchrone und Synchrone Signale
 - synchrone Signale innerhalb eines Prozesses versendet
 - Thread sendet Signal immer an sich selbst
 - asynchrone Signale von ausserhalb
 - Empfangsverhalten konfigurierbar
 - Signale an alle Threads oder
 - Signale an bestimmte Threads
 - vom Betriebssystem verwaltet (z.B. während Timer-Interrupt)
 - werden mit einem Signal Handler behandelt
- können unter Windows durch Asynchrone Prozeduraufrufe (APCs) emuliert werden

5.4. Scheduling

- Scheduling: Zuteilung der Prozessoren zu lauffähigen Prozessen
- Grundmuster für Prozesse
 - CPU-lastig
 - Prozess nutzt viel Rechenzeit, wartet selten auf I/O
 - I/O-lastig
 - Prozess nutzt Peripherie und wartet oft auf I/O.
- Prozessklassen
 - Batch Jobs
 - Prozesse ohne Benutzerdialog
 - Interaktive Prozesse
 - Ablauf gesteuert durch Benutzerdialog
 - Real-Time Prozesse
 - Einhalten eines vorgegebenen Zeitlimits obligatorisch



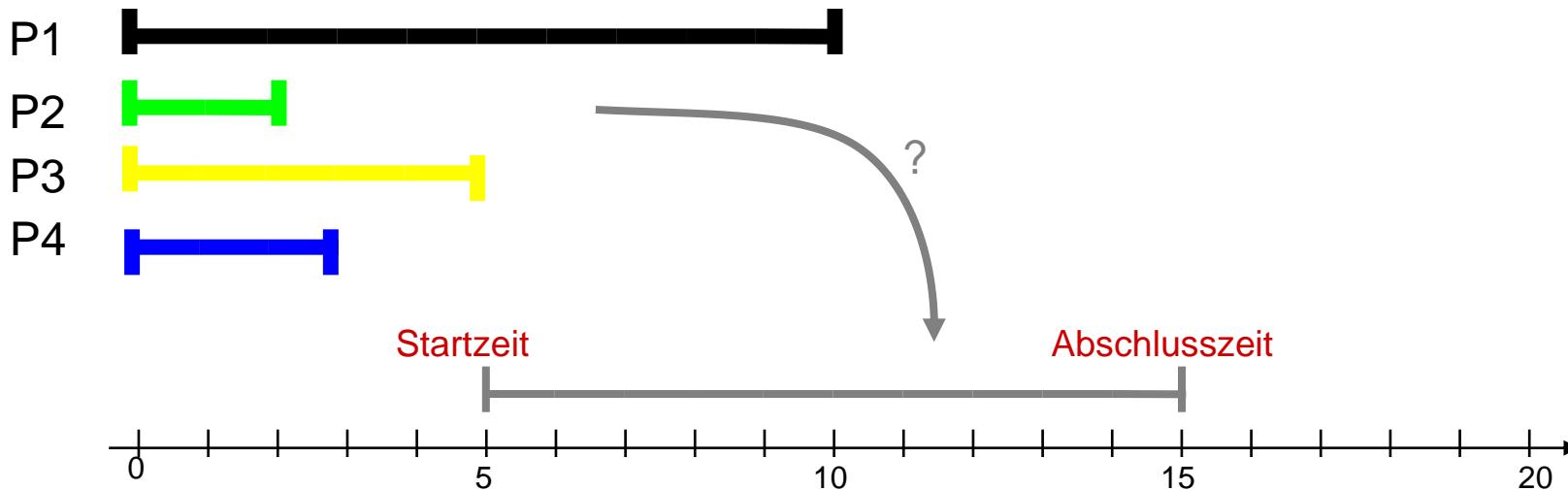
Histogramm der CPU-Burst Dauer (typisch)

Scheduling: Einteilung

- **Optimierungsziele**
 - Durchlaufzeit (turnaround time)
 - Gesamtzeit von Prozessstart bis Beendigung
 - Antwortzeit (response time)
 - Zeit zwischen Eingabe und Reaktion
 - Endtermin (deadline)
 - Zeitpunkt, zu dem Aktion erfolgt sein muss
 - Vorhersagbarkeit
 - Ausführungszeit eines bestimmten Prozesses vorhersagbar
 - Prozessorauslastung
 - Prozentualer Anteil der Zeit, zu der der Prozessor beschäftigt ist
 - Durchsatz
 - Maximierung der Anzahl Prozesse pro Zeiteinheit
- Prozess-basiertes Scheduling (ohne Threads) vs. Thread-basiertes Scheduling
- Kurzfristiges Scheduling unterteilt in
 - verdrängende Strategien (mit Preemption)
 - nicht verdrängende Strategien (ohne Preemption)

Beispielszenario

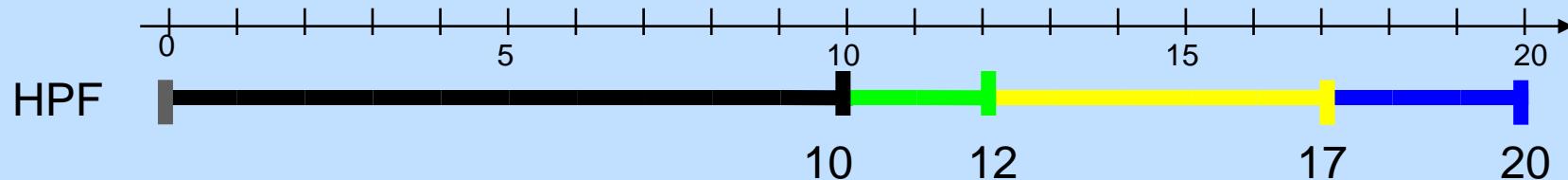
- Prozesse P1, P2, P3, P4
 - Gleichzeitige Ankunft (**Ankunftszeiten alle 0**)
 - Absteigende **Prioritäten**
 - Verarbeitungszeiten 10, 2, 5, 3
- Einprozessorsystem
- Von Interesse: Mittlere **Verweilzeiten**



Globale Dispatching Strategien (1)

- First Come First Served (FIFO)
 - Fair, unterschiedliche erwartete Wartezeit
- Highest Priority First (HPF)
 - Ohne oder mit Preemption

Im Beispiel: mittlere Verweilzeit (HPF/FIFO) = 14.75



Quotienten Verweilzeit / Bearbeitungszeit:

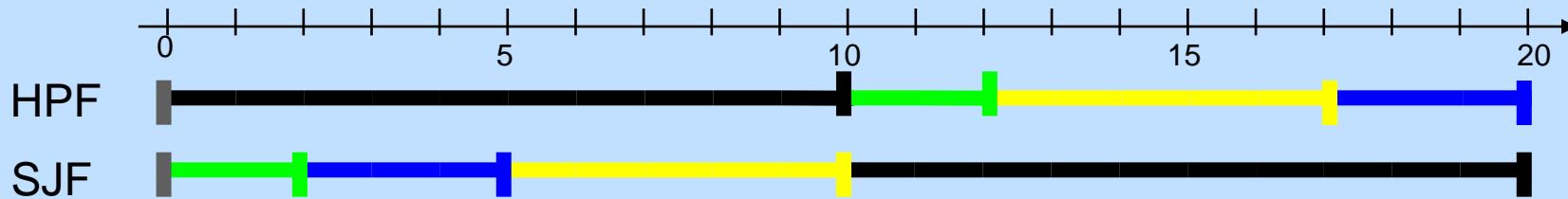
- TV/TB (P1) = 1
- TV/TB (P2) = 6
- TV/TB (P3) = 3.4
- TV/TB (P4) = 6.7

Bevorzugung langer Prozesse
Bevorzugung CPU-lastiger Prozesse

Globale Dispatching Strategien (2)

- Shortest Job First
 - Bedingt Kenntnis/Schätzung der Verarbeitungszeit T
 - Spezialfall von HPF mit $P=T^{-1}$

Im Beispiel: mittlere Verweilzeit (SJF) = 9.25



Quotienten Verweilzeit / Bearbeitungszeit:

- TV/TB (P1) = 2
- TV/TB (P2) = 1
- TV/TB (P3) = 2
- TV/TB (P4) = 1.7

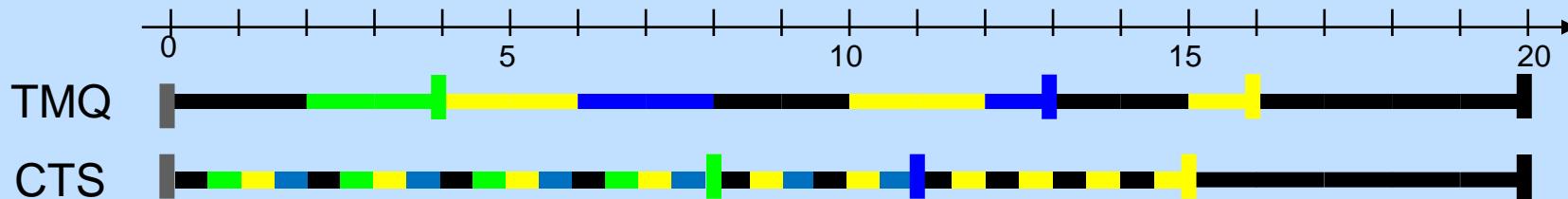
Globale Dispatching Strategien (3)

- Round Robin (Timesharing)
 - Mit Preemption durch Zeitquantum

Im Beispiel:

mittlere Verweilzeit (RR mit Quantum 2) = 13.25

mittlere Verweilzeit (RR kontinuierlich) = 13.5



Quotienten Verweilzeit / Bearbeitungszeit

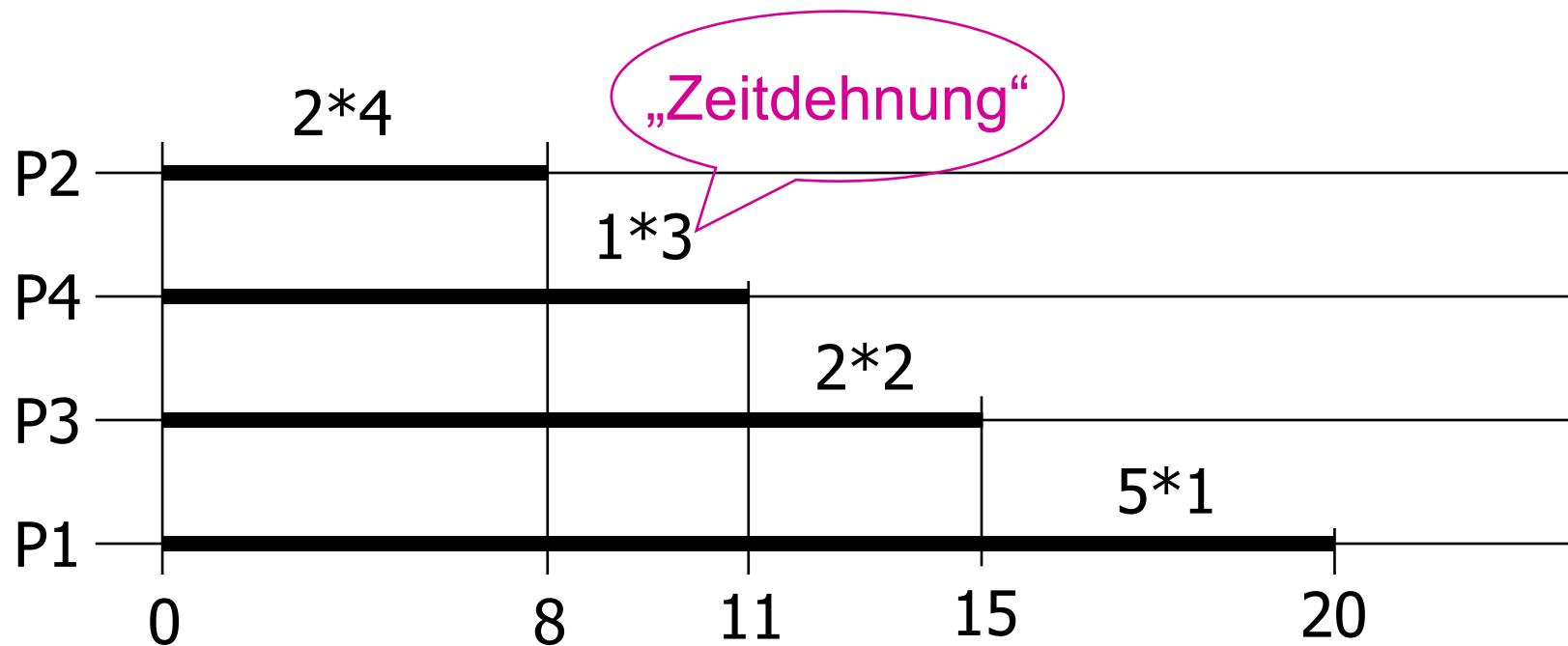
mit Quantum 2
■ TV/TB (P1) = 2
■ TV/TB (P2) = 2
■ TV/TB (P3) = 3.2
■ TV/TB (P4) = 4.3

kontinuierlich
■ TV/TB (P1) = 2
■ TV/TB (P2) = 4
■ TV/TB (P3) = 3
■ TV/TB (P4) = 3.7

Globale Dispatching Strategien (3)

zum kontinuierlichen Timesharing

- Kontinuierliches Timesharing (Quantum $\rightarrow 0$), ohne Berücksichtigung der Umschaltzeiten

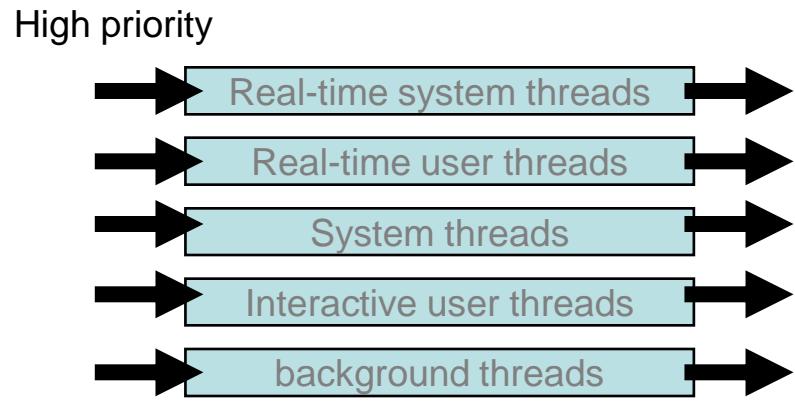


Globale Dispatching Strategien (4)

- Shortest Remaining Time
 - Unterbrechende Version von SJF
- Longest (Highest) Response Ratio Next
 - Response Ratio = Verweilzeit / Bearbeitungszeit
 - Abhängig von der bisherigen Wartezeit
- Feedback (Multi-Level Feedback)
 - Preemptiv mit Zeitquantum
 - Warteschlangen für Prioritäten
 - Dynamisches Vergeben von Prioritäten während der Ausführung.

Multilevel-Warteschlangen

- Ready-Queue wird in mehrere Warteschlangen aufgeteilt
 - Real-time (System, Multimedia)
 - Interactive, etc.
- Verschiedene Scheduling-Algorithmen
 - z.B.
 - Real-Time – RR / HPF
 - Interactive – RR + priority-elevation + quantum stretching
- Scheduling auch zwischen den Warteschlangen
 - Fixed Priority Scheduling (zuerst alle Echtzeit-Threads, dann die interaktiven)
 - Verhungern möglich
 - Time-slices – jede Warteschlange bekommt einen Anteil an der CPU-Zeit.
- Prozesswechsel zwischen Warteschlangen möglich



Starvation

- **Starvation** ist ein Problem bei Prioritäts-basiertem Scheduling: Threads mit niedriger Priorität verhungern
- **Lösungen**
 - **Aging** (Unix)
 - Priorität von CPU-lastigen Prozessen wird verringert / Priorität lange wartender Prozesse erhöht
 - Exponentielles Mitteln der CPU-Nutzung, um die Priorität blockierter Threads zu erhöhen
 - τ = geschätzte Zeit, t = real benötigte Zeit
 - $$\tau(n+1) = \alpha * t(n) + (1-\alpha) \tau(n) \quad (0 \leq \alpha \leq 1)$$
 - **Priority Elevation** (Windows/VMS)
 - Erhöhung der Priorität von Threads nach Erledigung von I/O
 - System gibt verhungernden Threads einen extra Schub (burst)
 - **Quantum Stretching**:
 - Bevorzugung der GUI-aktiven Threads oder
 - Bevorzugung I/O-lastiger Threads

Multiprozessor-Scheduling

- Symmetrie
 - **Asymmetrisches** Multi-Processing
 - Scheduling und I/O Processing auf einem designierten Prozessor
 - **Symmetrisches** Multi-Processing
 - Jeder Prozessor ist selbstplanend
- Affinität
 - **Soft Affinity**: Thread wird möglichst auf einem Prozessor gehalten
 - **Hard Affinity**: Feste Zuordnung eines Threads zu einem Prozessor
- Load Balancing
 - Möglichst ebenmäßige Lastverteilung auf die Prozessoren
 - Nötig bei Warteschlangen pro Prozessor
 - Wirkt der Affinität entgegen

Zusammengefasst

Scheduling in heutigen Systemen: Windows & Linux

- Prioritäts-getriebene, preemptive Scheduler
 - Es läuft immer der höchst-priorisierte Prozess, sofern mit der Prozessor-Affinität vereinbar, die Prozessor-Affinität kann sich ändern
- Der Scheduling Code ist über den Kern verstreut und wird bei Bedarf aufgerufen, es gibt keinen eigenen Scheduler Prozess
 - Aufruf bei: neuer Prozess, Ablauf des Zeitquants (Timer IRQ), Prozess beendet oder gibt Kontrolle ab (yield oder Wartezustand), die Priorität eines Threads wird geändert (System Call oder Dynamik), Prozessor-Affinität wird gewechselt
- Die Systeme unterhalten dynamische Warteschlangen nach Priorität
 - Warteschlange pro Prozessor
 - Ein neuer Thread kann in O(1) ermittelt werden

Scheduling in Windows & Linux

Prioritäten

- Die Systeme unterhalten verschiedene Prioritätsgruppen
 - Echtzeitprioritäten, Variable Prioritäten, Background/Batch Prioritäten
 - Prioritäten werden bei der Prozesserzeugung eingestellt (statische Priorität) und können während der Ausführung variieren (dynamische Priorität)
 - Dynamische Prioritäten können durch explizite Aufforderung variieren (Linux: nice) und/oder
 - aufgrund von Messwerten im System geändert werden (Windows: Priority Boosting nach I/O Beendigung, nach Wartebedingungen, nach CPU Starvation; Linux: bonus Werte nach durchschnittlicher Wartezeit und statischer Priorität: interactive Delta)

Scheduling in Windows & Linux

Quantum

- Wird ein Thread zur Ausführung ausgewählt, läuft er für eine gewisse Zeit: *Quantum*
 - *Quantum* Werte können von System zu System und von Prozess zu Prozess variieren (Systemeinstellung / Vorder-/Hintergrundstatus eines Prozesses / Dynamische Veränderung durch das System)
 - Die Ausführung des Threads kann vor Ablauf des Quantums unterbrochen werden (z.B. Thread höherer Priorität betritt das System)
 - Die Länge des Quantums richtet sich nach der statischen Priorität eines Prozesses (Linux), sie kann dynamisch verändert werden (Quantum Boosting, Windows)

5.5. Echtzeit-Systeme

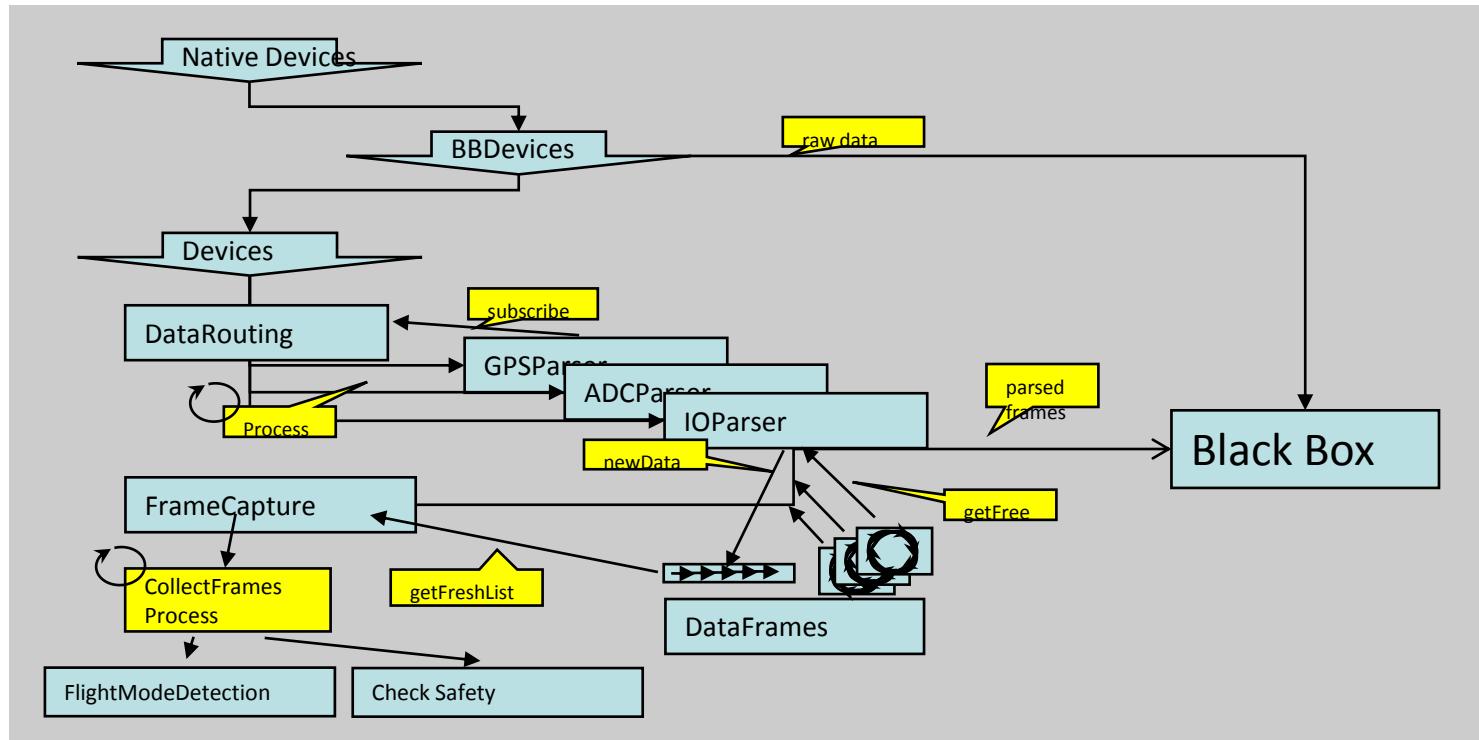
- In einem Echtzeitsystem werden nicht nur *korrekte* Resultate erwartet, sondern es müssen auch Garantien für die Ausführungszeit gegeben werden.
- Echtzeitsysteme sind oft sicherheitskritische Systeme, in denen die Verletzung dieser Garantien katastrophale Folgen haben kann.
- Man unterscheidet in „Hard Real-Time Systems“ und „Soft Real-Time Systems“.
 - Die meisten Echtzeitsysteme können mit harten Grenzen nicht umgehen, statt dessen wird „so schnell wie möglich“ verfahren.

Typische Features von Echtzeitsystemen

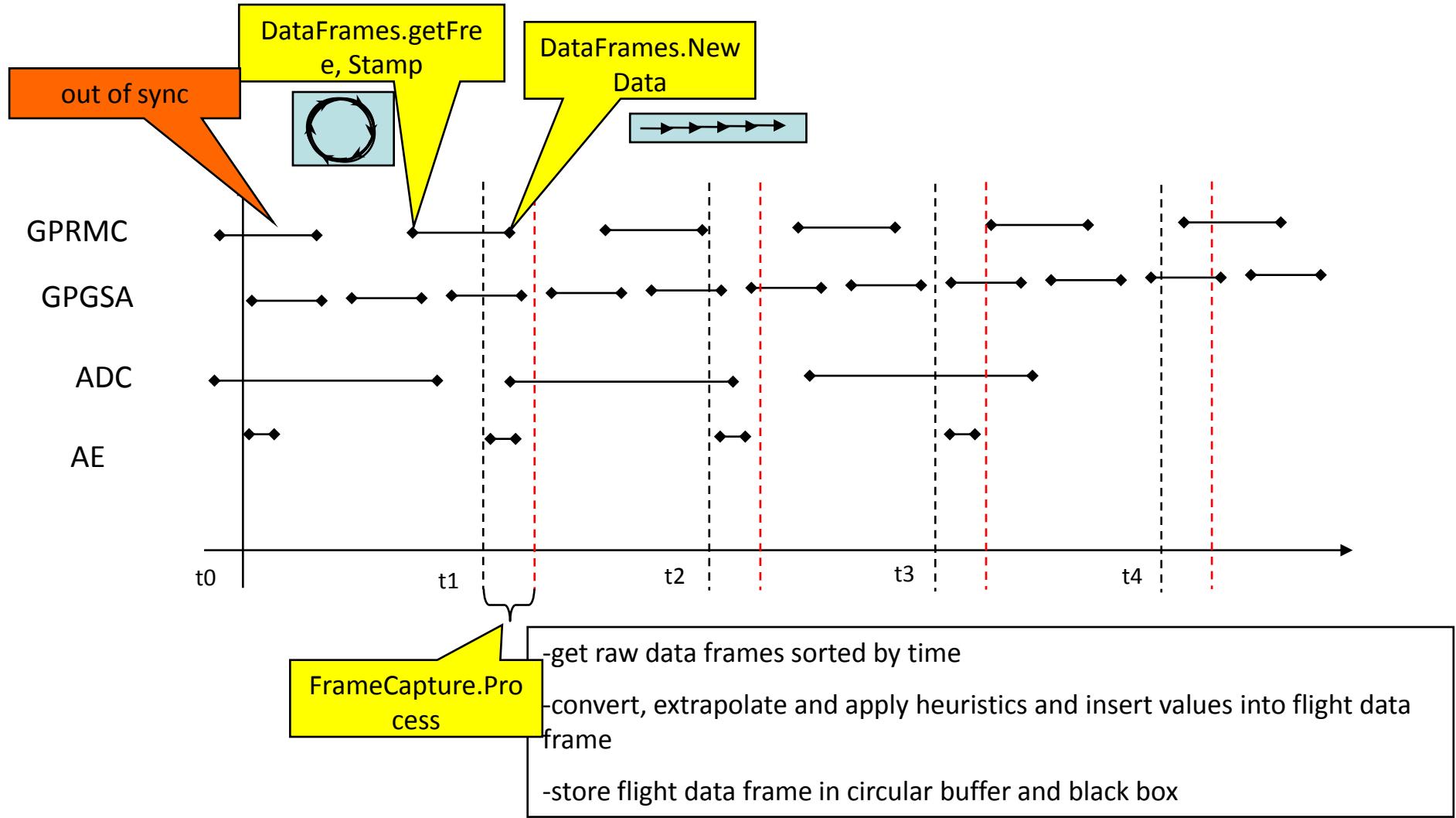
- Schneller Prozess- / Thread-Wechsel
- Geringe Größe
- Schnelle Behandlung externer Unterbrechungen
- Schnelle Dateiverwaltung
- Lock-Vermeidung
- Scheduling mit Vorrangunterbrechung anhand von Prioritäten
- Minimierung von Zeitintervallen, in denen Interrupts gesperrt sind
- Verzögerung von Tasks für eine fixierte Zeitdauer.
- Nicht oder sehr selten
 - Variation über die angeschlossenen Geräte
 - Zugriffsschutz und Sicherheitsmechanismen zwischen Prozessen
 - Mehrere Benutzer

Beispiel: On Board Active Safety System (ONBASS) Device

- Flugdateneingabe alle 1/8 Sekunden
- Datenkonversion, -Filterung, Flugphasenerkennung und weitere Algorithmen in dieser Zeit garantiert
- Webserver und Benutzerinteraktion während „Slacktime“



Beispiel: ONBASS Flugdateneingabe (Geräte)

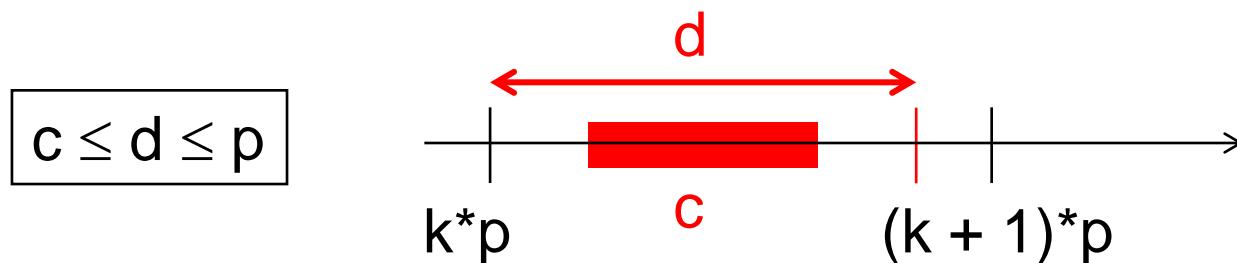


Implementation von Echtzeitsystemen

- Preemptives, prioritätsbasiertes Scheduling
 - System muss auf Echtzeit-Prozesse sofort antworten.
 - Meist in Soft-Realtime Systemen (Hard-Realtime Systems.u.)
- Preemptiver (unterbrechbarer) Kern
 - Echtzeit-Task darf nicht auf den Kern warten müssen
 - Erreicht durch Einfügen sog. Preemption Points im Kern bzw. durch entsprechenden Schutz beim Zugriff auf gemeinsam verwendeten Datenstrukturen. Nicht-Präemptiver Code muss extrem kurz gehalten bleiben
- Minimierte Latenz
 - Interrupt-Latenz: bestimmt durch Hardware, durch Zeit des Kontextwechsels und durch Kernel-Code mit Ausmaskierung der Interrupts
 - Dispatch Latenz: Konfliktphase + Dispatching
 - Konfliktphase: Preemption eines im Kern laufenden Prozesses, Resourcenblockierung durch Prozesse niedrigerer Priorität (-> Priority Inversion)

5.5.1. Echtzeit Scheduling

- Prozessarten
 - Periodisch
 - Sporadisch / Aperiodisch
- Prozesscharakteristik
 - Computation (Execution) Time c (Worst Case)
 - Periode p (bzw. Minimum zwischen Events, ev. 0)
 - Deadline d



Scheduling-Arten

- Statisch, tabellengesteuert
 - Statische Analyse der Durchführbarkeit
- Statisch, prioritätengesteuert
 - Statische Analyse
 - Prioritätengesteuerter Scheduler
- Dynamisch, planungsgesteuert
 - Dynamische Festlegung der Zuteilung
 - Dynamisches Scheduling
 - Zuteilung von Tasks nur bei Einhaltung der Deadlines
- Dynamisch, best effort
 - Versuch der Einhaltung von Fristen ohne Garantie

Beispiele

1. Periodische Prozesse

- A (10, 20, 20) (Computation Time, Periode, Deadline)
- B (25, 50, 50)

2. Sporadische Prozesse

- A (10, 110) (Ankunftszeit, Starting Deadline)
- B (20, 20)
- C (40, 50)
- D (50, 90)
- E (60, 70)

zu 1.: Scheduling periodischer Echtzeit-Tasks

Ankunftszeiten,
Ausführungszeiten und
Deadlines

Prioritätenscheduling:
A hat Priorität

Prioritätenscheduling:
B hat Priorität

Earliest Deadline First
(EDF) mit
Berücksichtigung der
Abschlusszeiten

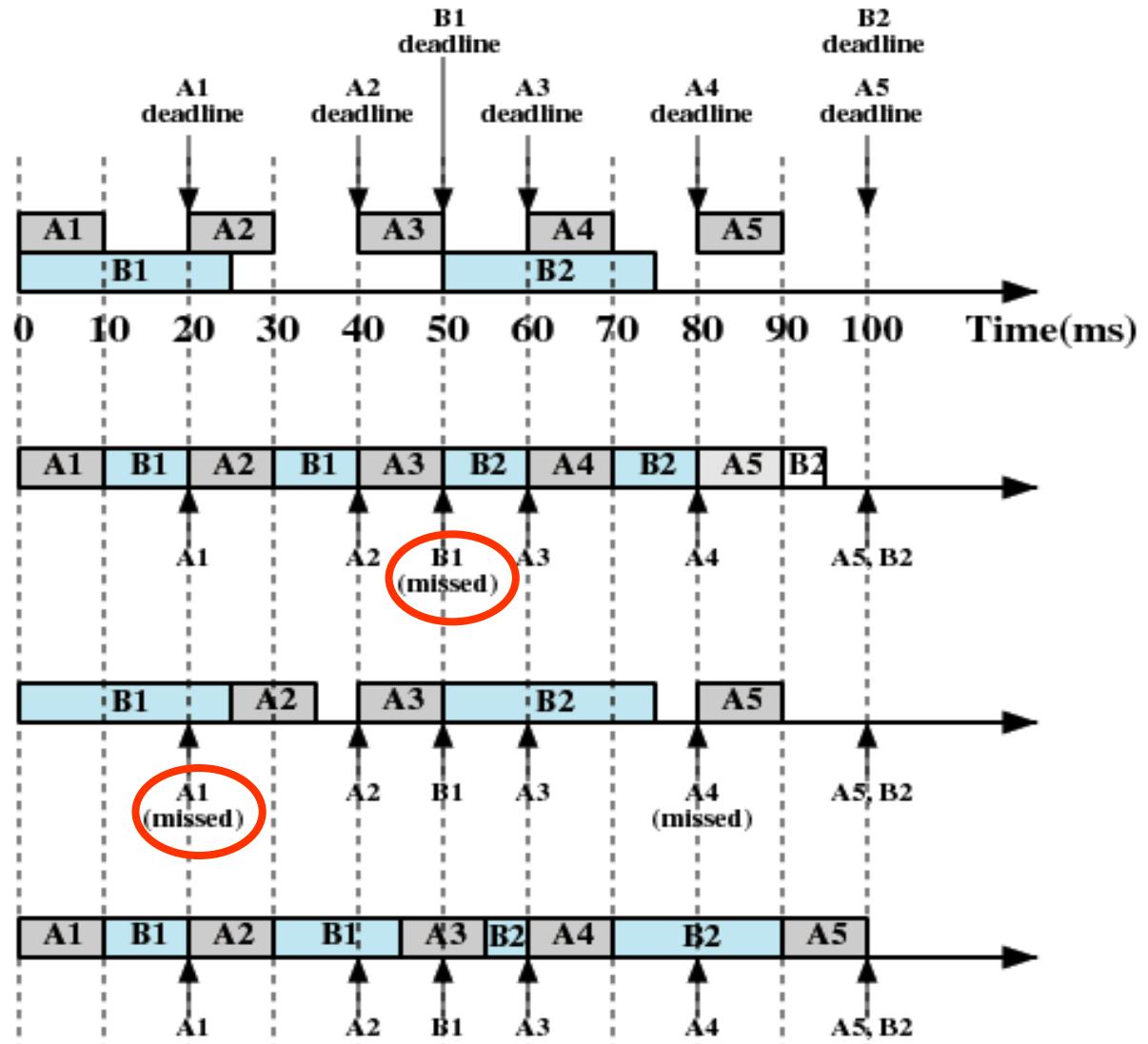


Figure 10.5 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.2)

zu 2.: Scheduling aperiodischer Echtzeit-Tasks mit Starting deadlines

einsetzbar, wenn starting deadlines und arrival times von vornherein bekannt:
Akzeptiere, dass Prozessor für gewisse Zeit unbeschäftigt ist.

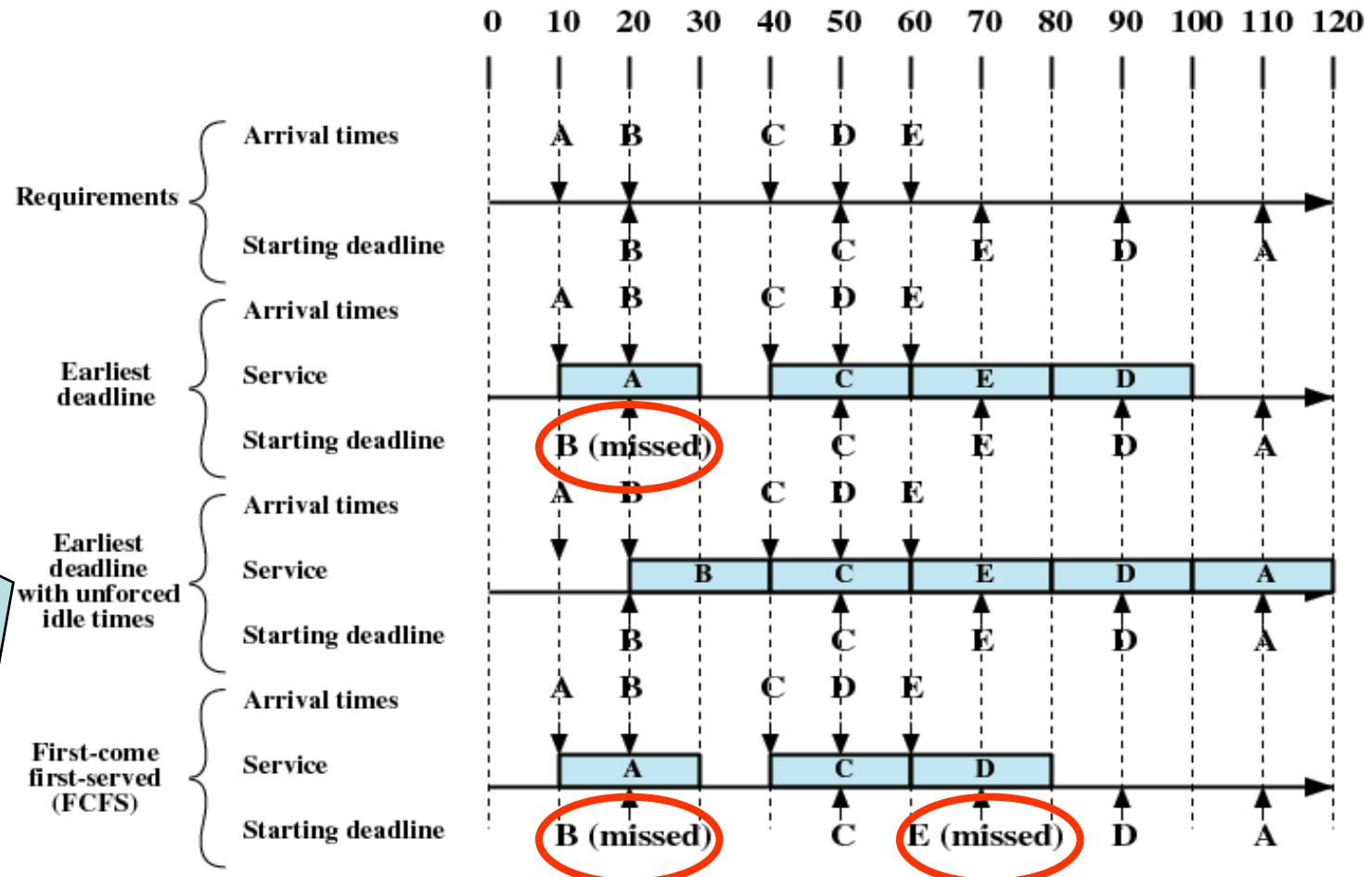


Figure 10.6 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines

Alternative: Preemptives Scheduling.

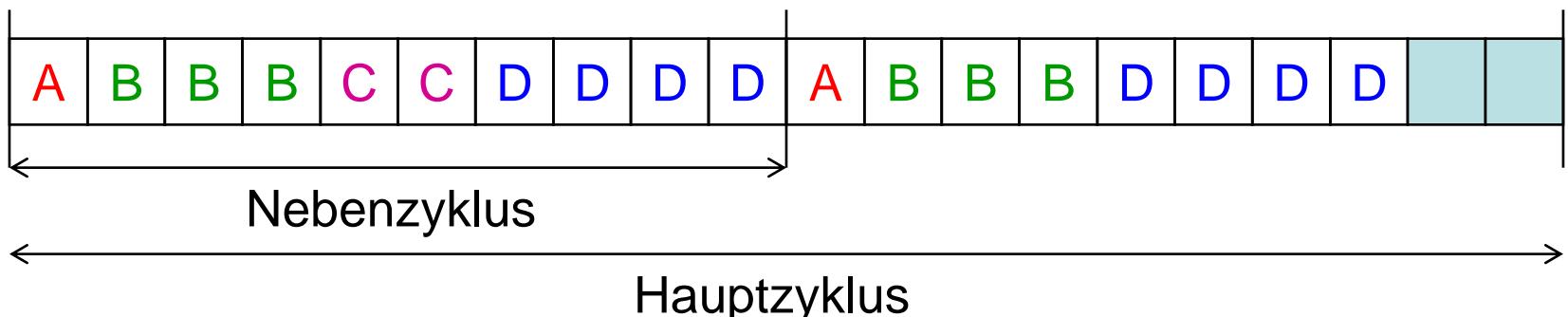
Zyk lenscheduling

(Clock-Driven Scheduling – Cyclic Schedules*)

- Rasterung der Zeitachse und Gliederung in Zyklen
- Statische Berechnung der Zyklenbelegung
- Beispiel

computation time, period, deadline

- Prozesse (c, p, d)
 - A (1, 10, 10), B (3, 10, 10), C (2, 20, 20), D (2 * 4, 20, 20)
- Hauptzyklus (kgv), Nebenzyklus (ggt)
 - 20, 10
- Plan



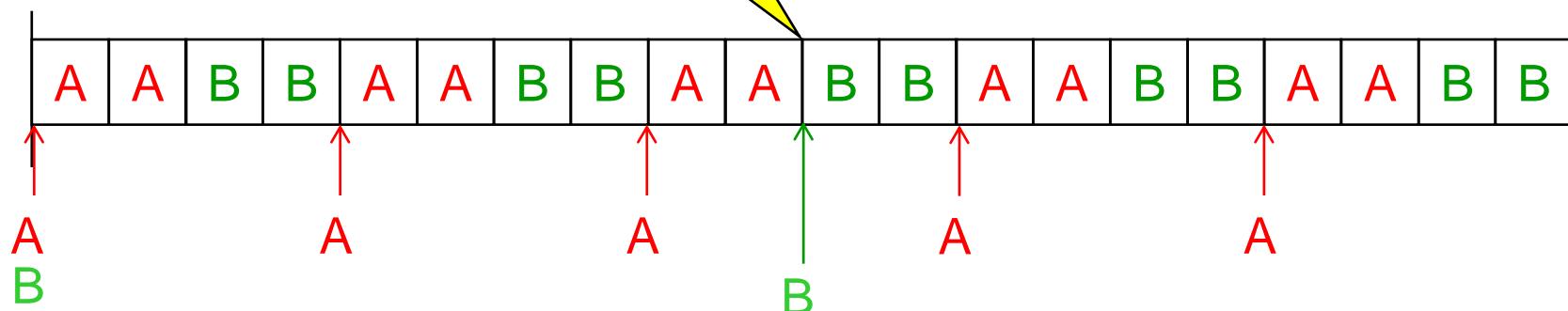
Prioritätenbasiertes Scheduling

Rate Monotonic (RM), Highest Rate First, **Statisch**

- preemptives Verfahren für periodische, unterbrechbare, voneinander unabhängige Tasks
- Idee: statische Priorität proportional zur Repetitionsrate
- Hier kann eine Grenze für erfolgreiches Scheduling angegeben werden:
 - Prozessorauslastung durch Prozess $P(j)$:
 $u(j) = c(j)/p(j)$
 - Es muss offensichtlich gelten [notwendig]
 $u(1) + \dots + u(n) \leq 1$ (optimistisch, perfektes Scheduling)
 - Korrektes Scheduling existiert (für $d(j)=p(j)$) für [hinreichend]
 $u(1) + \dots + u(n) \leq n * (2^{1/n} - 1)$
- Beispiel

A (2, 4, 4)
B (5, 10, 10)

nicht korrekt und
 $u(1) + u(2) = 1 > 0.83$

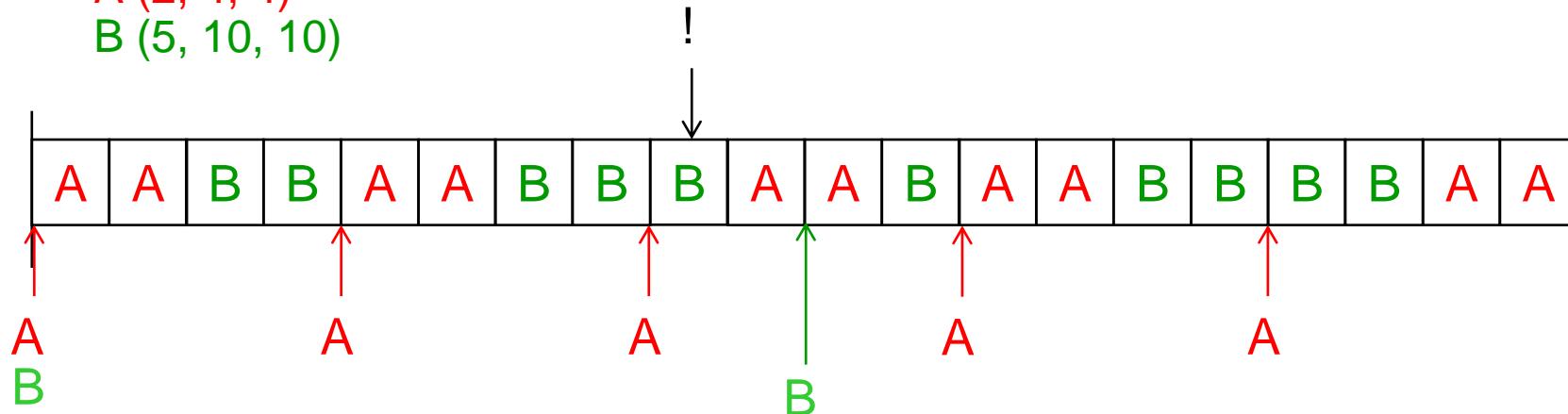


Prioritätenbasiertes Scheduling (2)

Earliest Deadline First (EDF), Dynamisch

- Bestimmung der nächsten Deadlines und dynamische Anpassung der Prioritäten jeweils zum Nebenzyklus.
- Theorem: Unter Standardannahmen* ist EDF mindestens genauso korrekt wie jeder andere Scheduler. **
- Für EDF gilt sogar, dass $u(1) + \dots + u(n) \leq 1$ hinreichende Bedingung für Korrektheit darstellt! **
- Beispiel EDF

- A (2, 4, 4)
B (5, 10, 10)



*Tasks unterbrechbar, Tasks unabhängig, Tasks periodisch, Deadline=Periodenlänge

** Quelle: Jane W.S. Liu: Real Time Systems (Prentice Hall 2000).

Prioritätenbasiertes Scheduling (3)

Latest Release Time (LRT), Dynamisch

- EDF umgedreht: Ausführung zum spätest-möglichen Zeitpunkt.

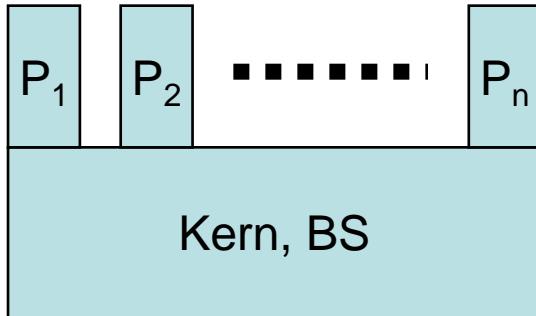
Least Slacktime (LST), Dynamisch

- Slacktime = Deadline – Time to Complete
- Bestimmung der nächsten Deadline und Restausführungszeit: Slacktime. Kürzeste Slacktime gewinnt.
- Theorem: Unter Standardannahmen sind LST und LRT auch mindestens genauso korrekt wie jeder andere Scheduler.

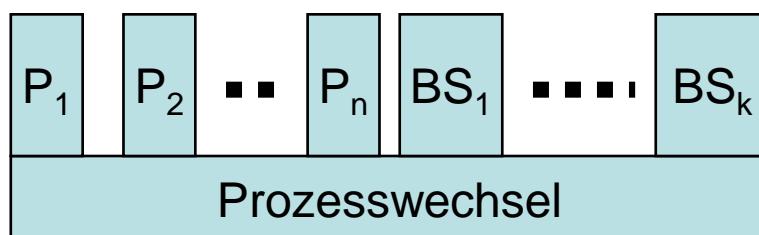
Kapitel 6. Betriebssystemarchitekturen

- 6.1. Kernel Architekturen
 - 6.1.1. Monolithische Systeme,
 - 6.1.2. Geschichtete Systeme
 - 6.1.3. Microkernel (Client-Server Modell)
 - 6.1.4. Exokernel
 - 6.1.5. Objektbasierte Systeme
- 6.2. Virtuelle Maschinen
 - 6.2.1. Grundideen und Beispiele
 - 6.2.2. Systematik: System- und Prozess-VM
 - 6.2.3. Emulation
 - 6.2.4. High Level Language VMs

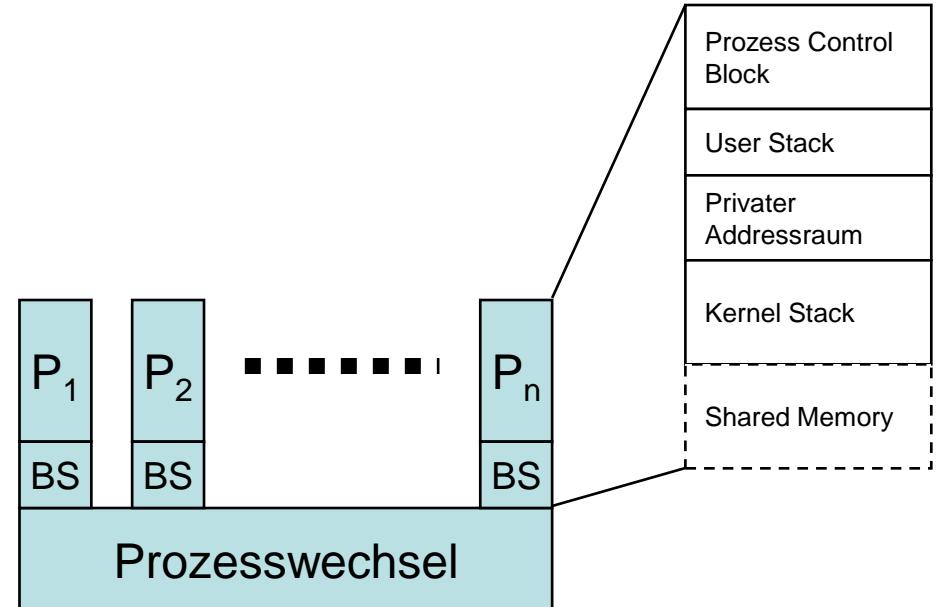
Betriebssystem und Benutzerprozesse



- Ausführung des Kerns außerhalb der Prozesse.
- Eigener Speicherbereich und Stack



- Prozessbasiertes Betriebssystem
- Kernel-Funktionen in Form von separaten Prozessen organisiert.



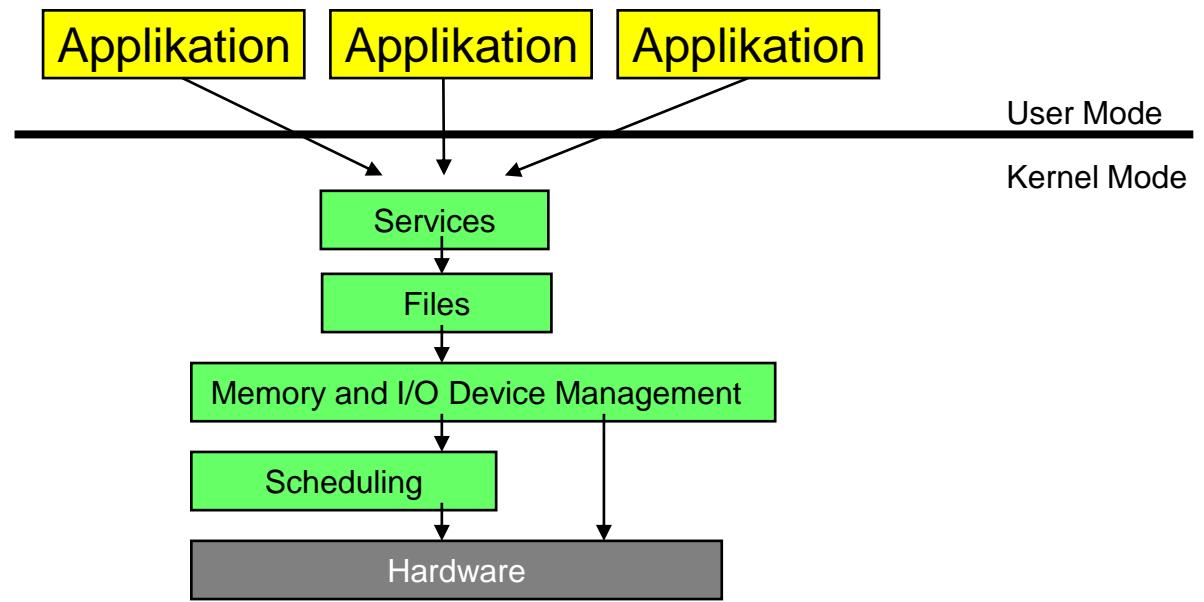
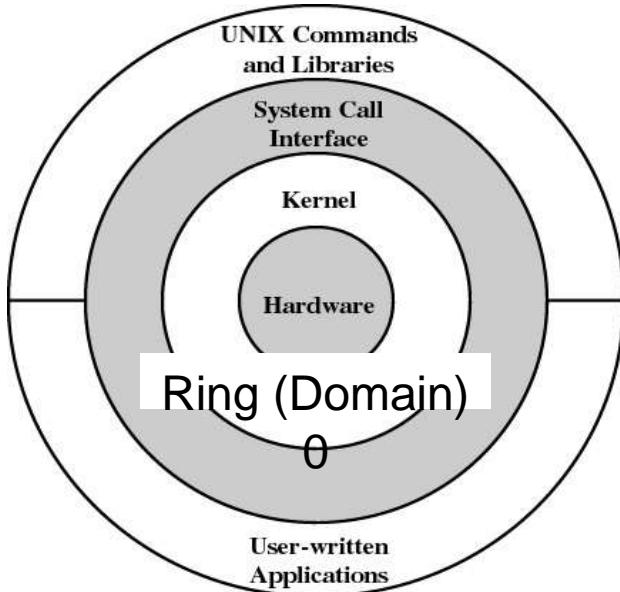
- Ausführung des BS im Kontext der Benutzerprozesse
- Zum Aufruf von BS Funktionen nur Modus-Wechsel nötig

6.1.1. Monolithische Systeme

- Keine oder unklare Struktur
- Evolutionär gewachsene Systeme
- Beispiele
 - MS-DOS,
 - ältere Unix-Varianten

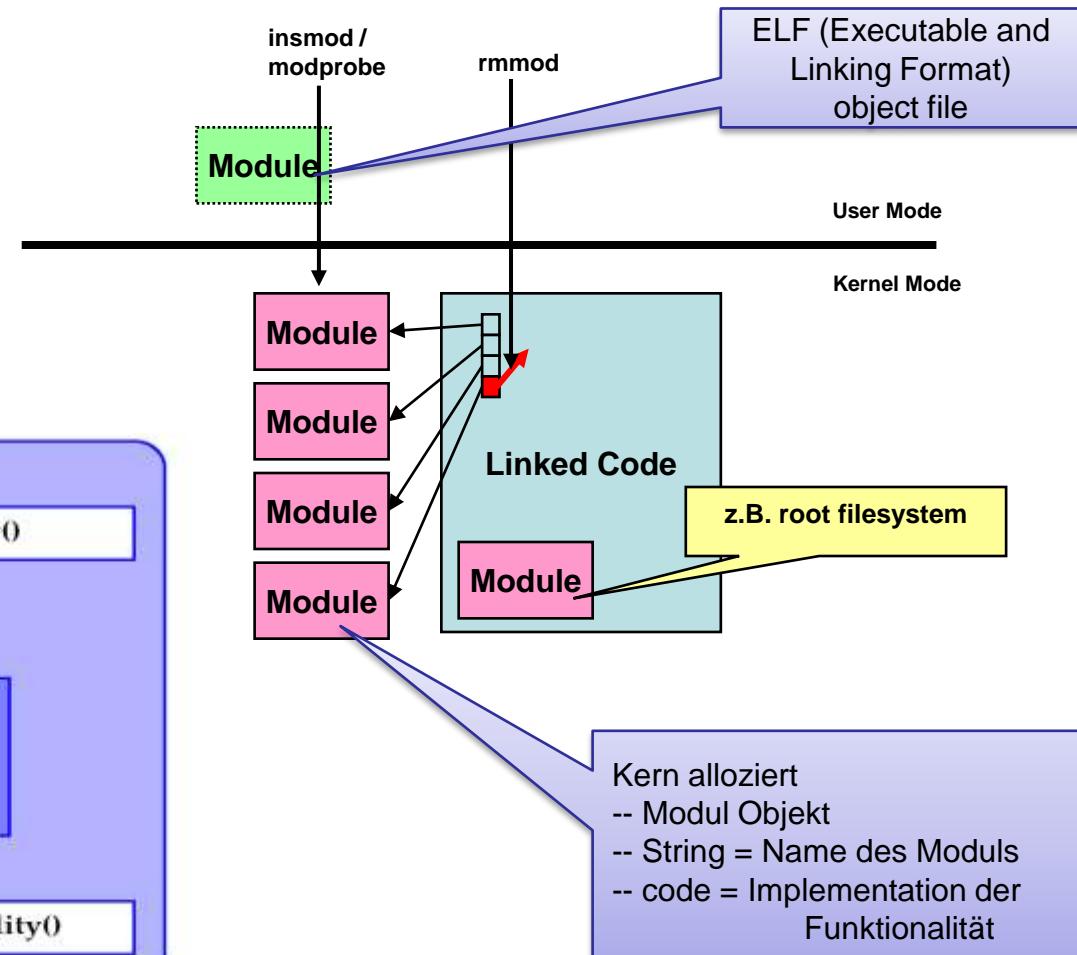
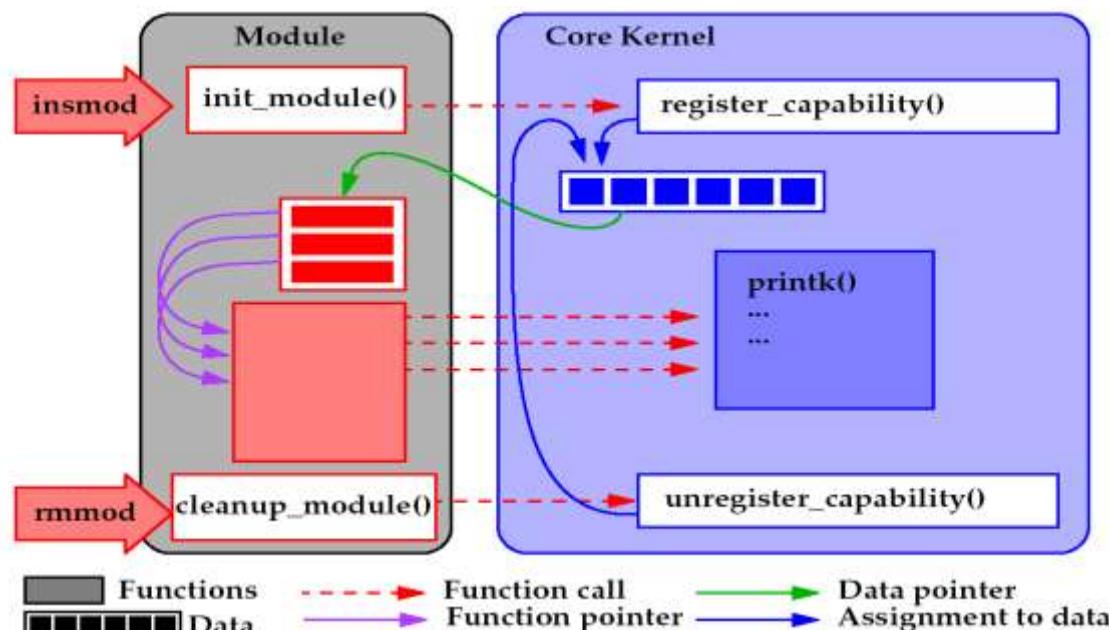
6.1.2. Geschichtete Systeme

- Jedem Layer wird nur Zugriff auf darunterliegende Layer zugelassen
- Beispiele
 - Viele Varianten von Linux, Windows



Linux: Module

- Linux Module als Plugins
(Kompromisslösung zwischen Monolith und Mikrokernen oder modularen Systemen)

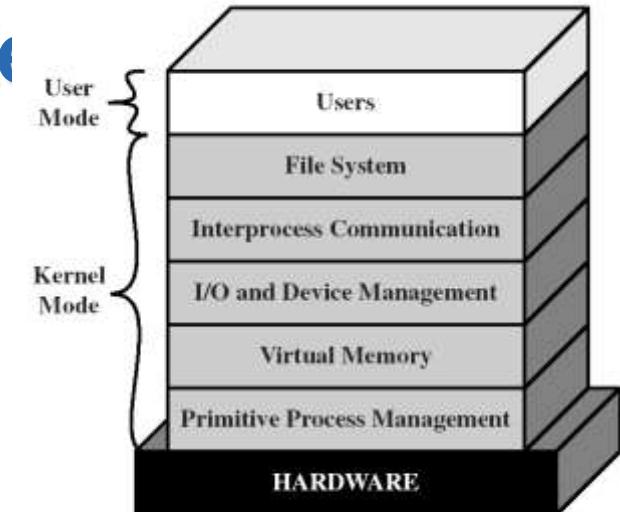
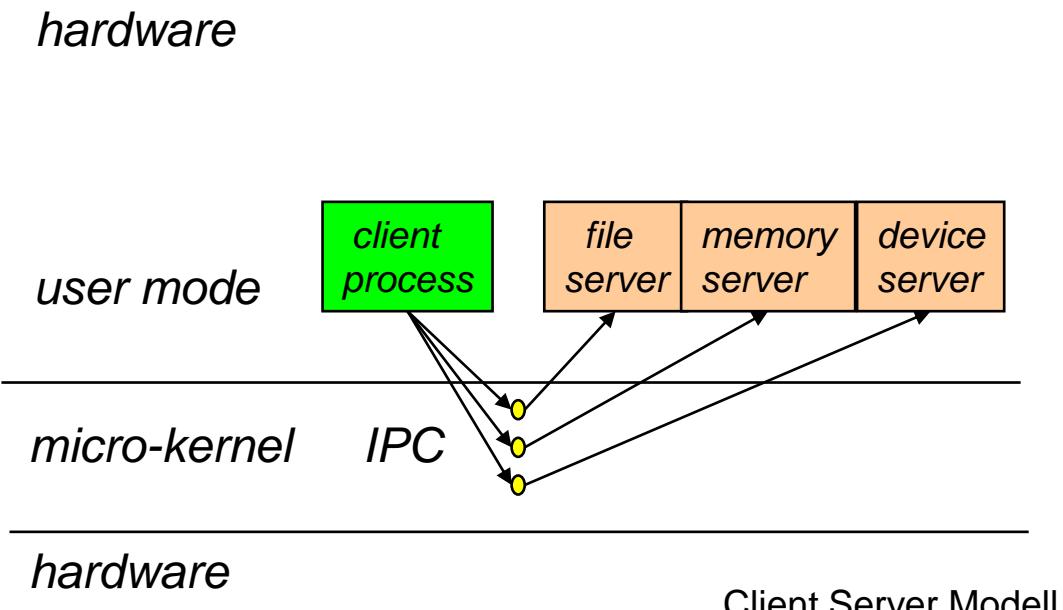
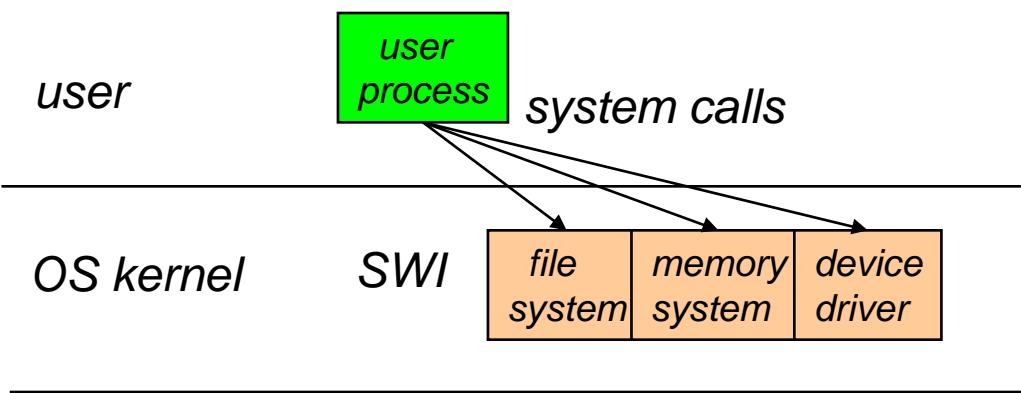


6.1.3. Microkern (Client-Server Modell)

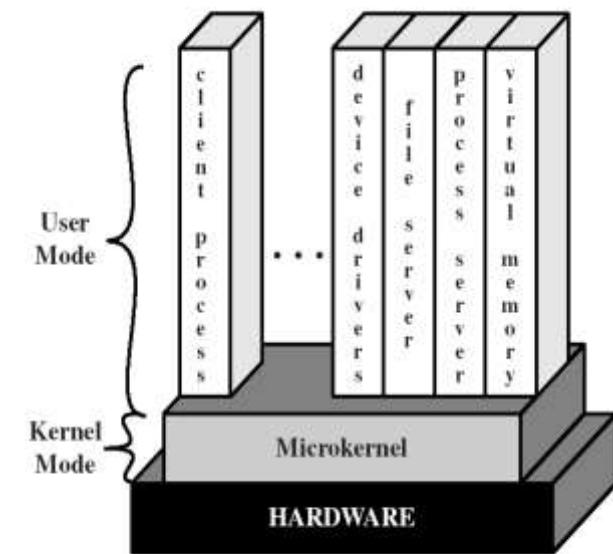
- Ziel: Reduktion der „trusted computing base“
 - Minimierung der im privilegierten Modus implementierten Funktionalität
 - Isolation der user-level Komponenten voneinander
- Reduzierte Funktionalität im Mikrokern
 - Geringere Codegröße (10.000 Zeilen C++ in L4 vs. 1 Million Zeilen C in Linux excl. Gerätetreiber)
- Microkernel
 - Prozess-/ Threadmanagement und IPC
 - Low-Level Memory Management
 - Internal Services
 - Device Drivers
- External Services
 - Memory Management
 - File Management
 - Device Drivers
 - Connectivity Management (TCP)
 - ...
- Aufruf von Systemfunktionen und Kommunikation zwischen Prozessen via Nachrichtenaustausch (IPC)

Beispiele: QNX, L4Linux (unter L4 Kern), AmigaOS, BeOS, GNU HURD, MkLinux, Mac OS X

Microkern vs. geschichteten Kernel



(a) Layered kernel



(b) Microkernel

Probleme der Microkerne

1. Generation: 80er Jahre

- Probleme von Mach
 - hoher Overhead für IPC-Operationen: Systemaufrufe Faktor 10 langsamer gegenüber monolithischem Kern
 - große Code-Basis
- Praktischer Einsatz von Mach nur in hybriden Systemen
 - Separat entwickelte Komponenten für Mikrokern und Server
 - Zusammenführung mehrerer Komponenten in einem Adressraum, Ersetzen von inkernel-IPC durch Funktionsaufrufe
 - Apple OS X: Mach 3 Mikrokern + FreeBSD

Microkerne

2. Generation: >= 1996

- Beschleunigung von IPC-Operationen
 - L4 (Jochen Liedtke, 1996)
 - „.... a concept is tolerated inside the μ kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.” *
- Mikrokern unterstützt
 - Ein Modell für Threads
 - Synchrone Kommunikation zwischen Threads
 - Scheduling
 - Abstraktion des Adressraums
- Nur 10 Funktionen im Kern
- Insbesondere die Überprüfung von Rechten bei IPC wurde aus dem Kern entfernt.
- 2. Generation der Mikrokerne wurden auch *Nano-Kerne* genannt

*On μ -Kernel Construction, Jochen Liedtke, Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP), Copper Mountain Resort, CO, December 1995

6.1.4. Exokerne

MIT 1995*

- Reduktion der Implementation im Betriebssystem auf absolutes Minimum
 - Leistungsverbesserung durch Entfernen von Hardware-Abstraktionsebenen
 - Klass. Kerne verbergen die Hardware vor den Applikationen
 - Exokerne erlauben Programmen den direkten Zugriff
 - Minimalste Größe des Kerns. Implementation von
 - Schutz
 - Resourcen-Multiplexing
 - Keine Implementation von IPC Mechanismen (Mikrokerne) oder weiteren Betriebssystemabstraktionen (Monolithen)
- Implementation der Abstraktion auf Anwendungsebene
 - Aufgabe des OS: Lediglich Multiplexing der Hardware
- Beispiele
 - ExOS (MIT)
 - Nemesis (Univ. of Cambridge et al)
 - Vorgänger/Inspiration für den Virtual Machine Monitor Xen (s.u.)

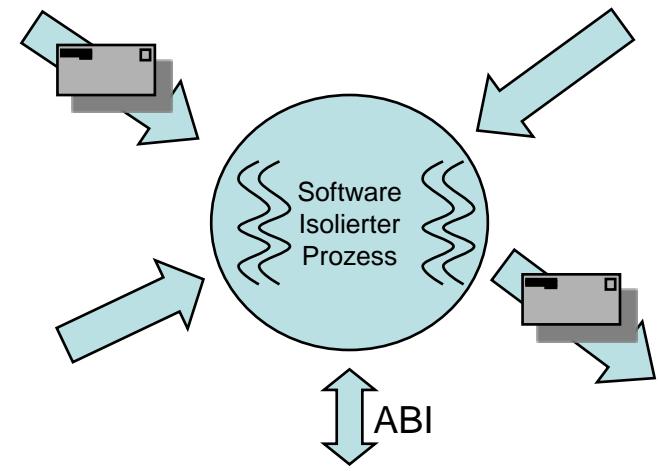
*Exokernel: An Operating System Architecture for Application-Level Resource Management,
D.R.Engler, M.F. Kaashoek, J.O`Toole, SOSP (1995)

6.1.5. Objektbasierte Systeme

- Sicherung der Modul- und Objekt-Grenzen durch Programmiermodell und Software
 - Byte-Code wird vom JIT-Compiler oder dynamischem Linker überprüft
 - Compiler sichert die Kapselung der Objekte
 - Gemeinsamer Adressraum für alle Objekte
- Beispiele:
 - Oberon/A2, Java VM, .NET CLI
 - Singularity (MS Research Redmond)

>> Contemporary operating systems—Windows, Linux, Mac OS X, and BSD—share a large number of design decisions (....)

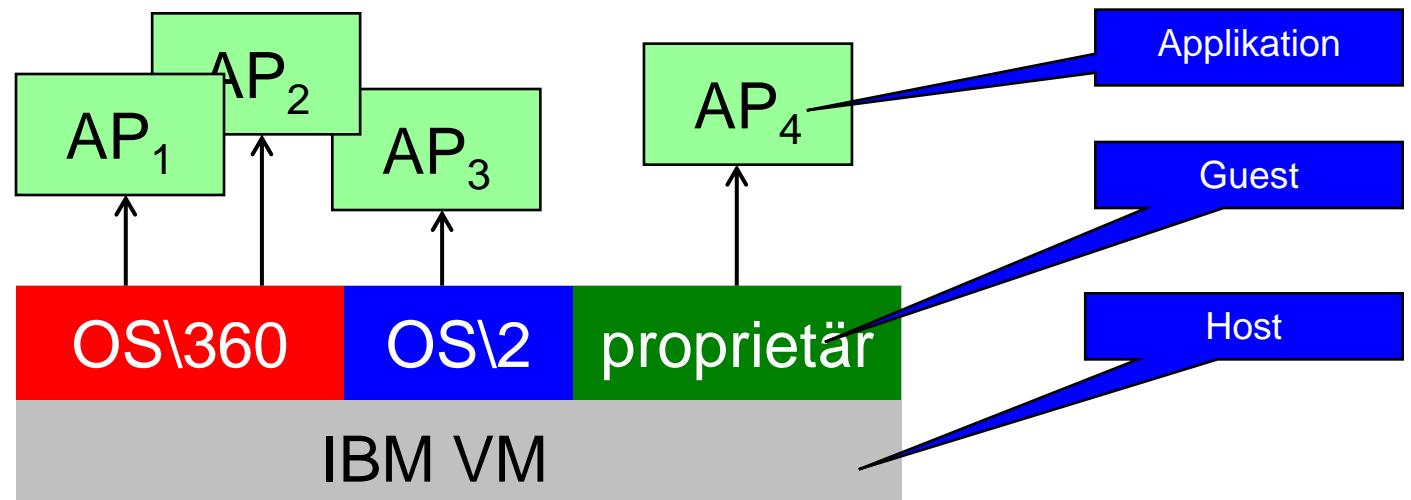
We believe that many of these problems are attributable to systems that have not evolved far beyond the computer architectures and programming languages of the 1960's and 1970's << *



6.2. Virtuelle Maschinen

6.2.1. Grundidee und Beispiele

- Emulation von Basishardware zum Betrieb eines gegebenen Betriebs- oder Laufzeitsystems
- Multiplexing emulierter Basishardware zum (ev. gleichzeitigen) Betrieb mehrerer Betriebssysteme
- Historische Wurzel: IBM VM CP/CMS* 1967



*Control Program / Cambridge Monitor System

Fragen

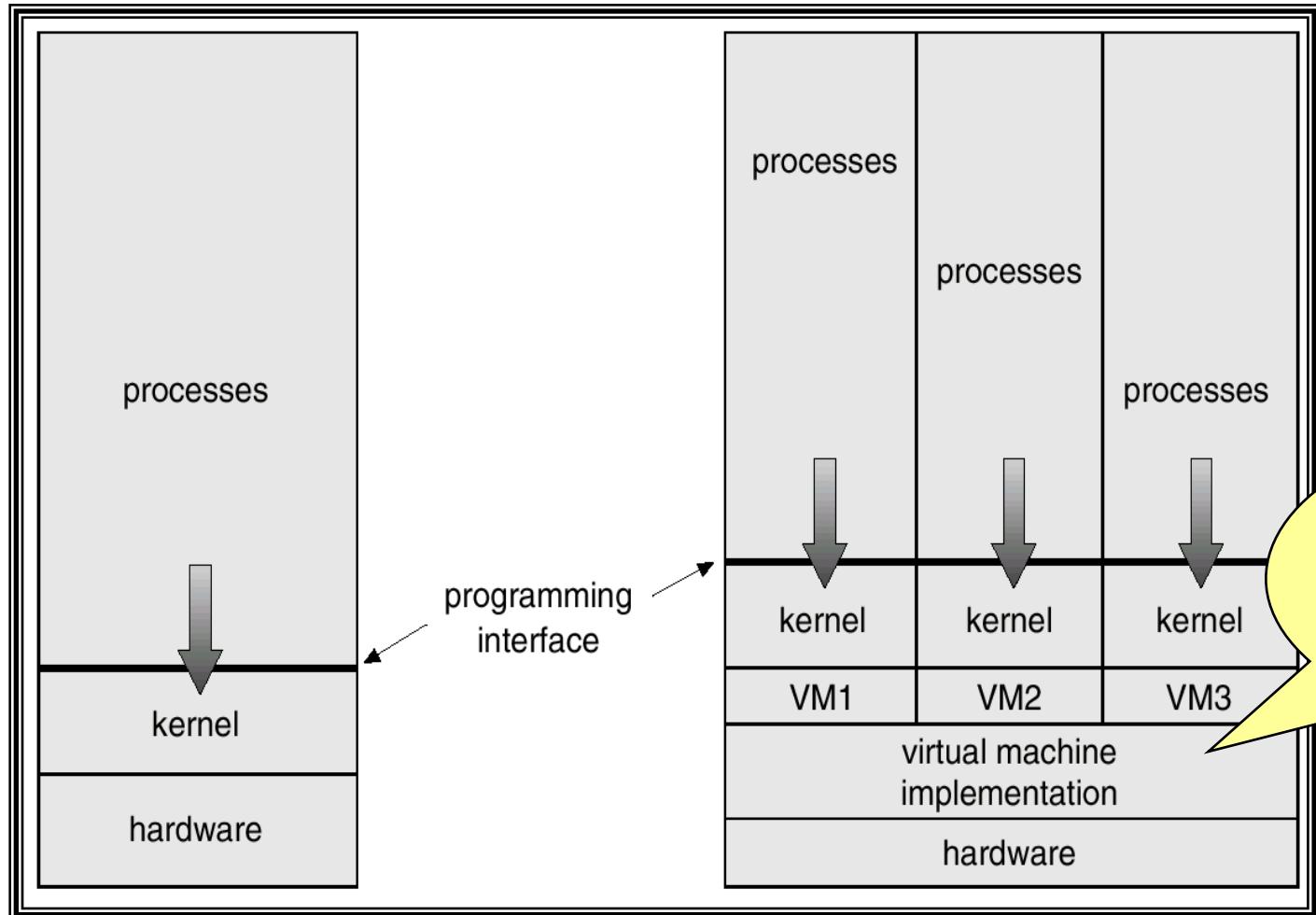
(Antworten im Laufe des Kapitels)

- Wozu virtuelle Maschinen?
- Kritische Ingredienzen eines Computersystems (bzgl. VM)?
- Wo könnten die Probleme liegen?
- Was darf eine virtuelle Maschine nicht auf dem Host System ausführen?
- Was muss die Virtuelle Maschine leisten?
- Wie lassen sich VMs beschleunigen?

Zweck

- Senkung des Kosten- und Administrationsaufwands
 - Multiplexing auf genügend schneller Hardware
 - Bsp.: Gemietete Remote-Hardware, Netzwerk-Server
- Sicherheit
 - Isolation und Überwachbarkeit
- Cross-Platform Kompatibilität
 - Bsp: Linux unter Windows, Windows unter Linux etc.
- Implementationsunterstützung
 - Debugging von Betriebssystemen
- Portabilität
 - Bsp.: Java VM, .NET VM, Pascal P-Code
- Performance Isolation

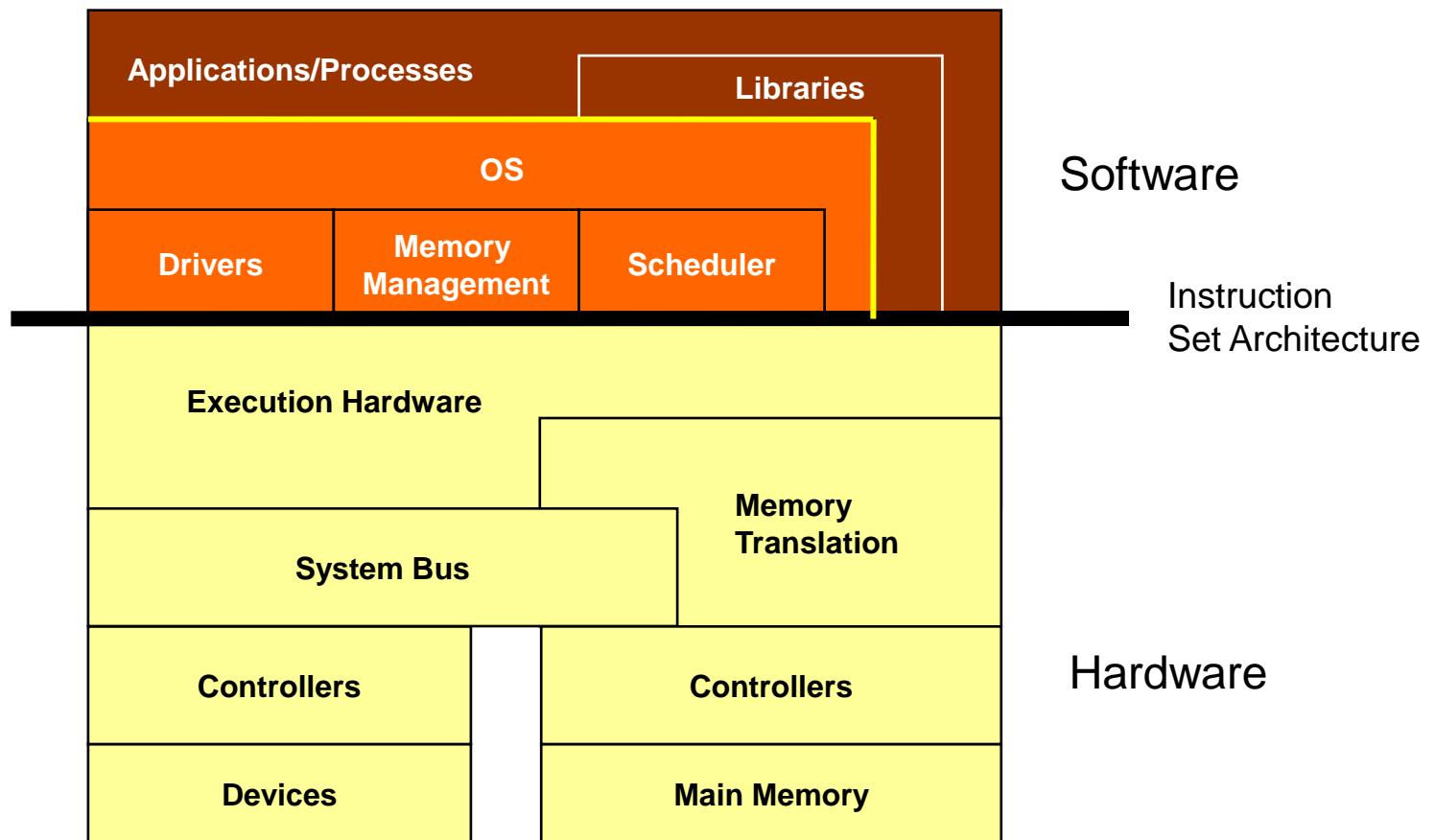
Prinzip



„Hypervisor“ /
„Virtual Machine
Monitor“ (VMM)

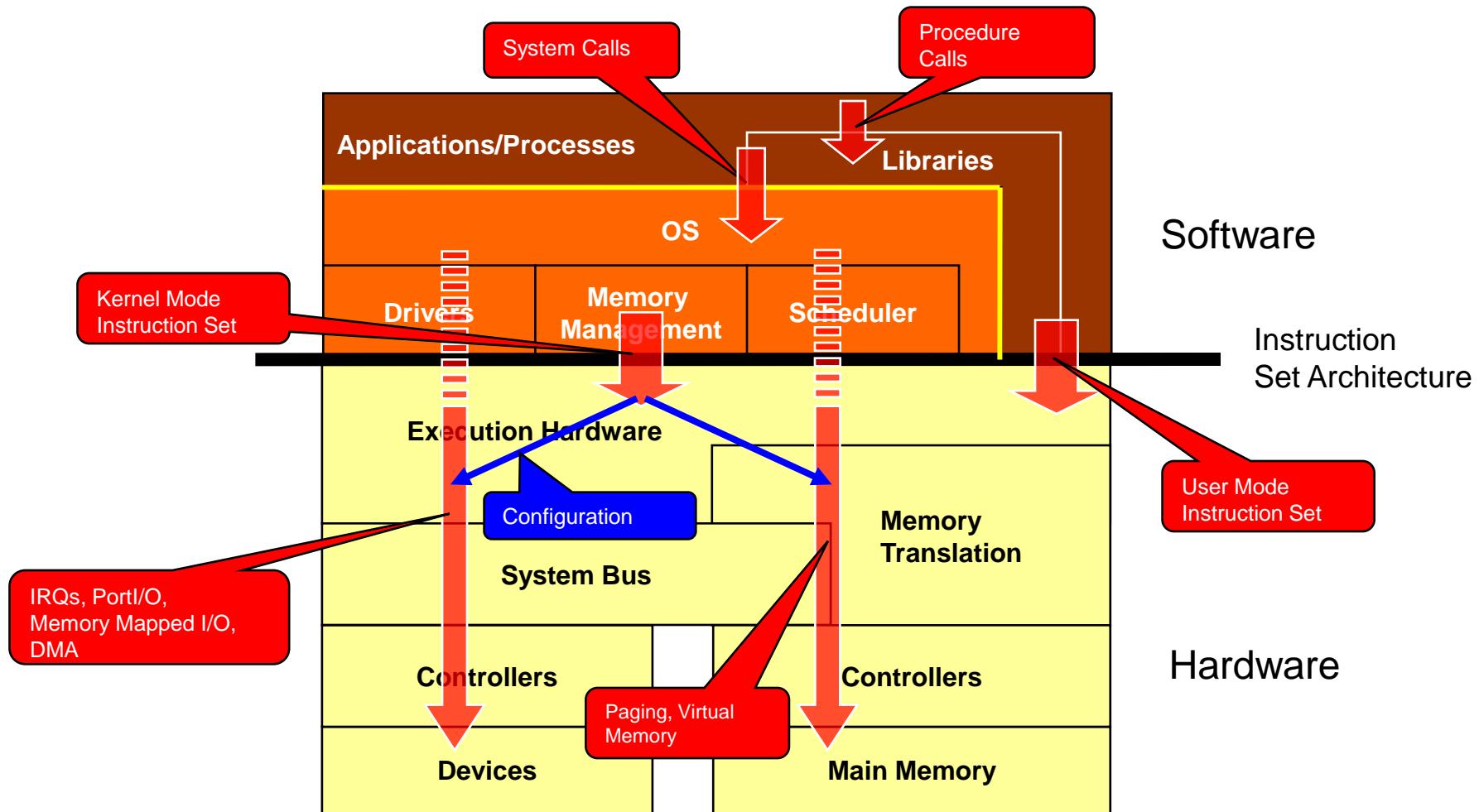
Computer System Architektur

zur Erinnerung, schematisch



Computer System Architektur

Zugriffswege



Einfaches Beispiel

Emulation in der ARM7 VM in A2

```
PROCEDURE DecodeARM(addr, code : LONGINT);
VAR cc, instr, q0, q8, q12, q16 : LONGINT;
BEGIN
IF BigEndian THEN SwapBytes(code) END;
cc := SYSTEM.LSH(code, -28); (* extract condition code *)

IF (cc = 14) OR (CheckCond(cc) = TRUE) THEN
instr := SYSTEM.VAL(LONGINT, SYSTEM.VAL(SET, code) - {28..31});
```

CASE (instr DIV Bit26) MOD Bit2 OF

```
0 : q12 := (instr DIV Bit12) MOD Bit4; q16 := (instr DIV Bit16) MOD Bit4;
IF ODD(instr DIV Bit25) THEN
CASE (instr DIV Bit21) MOD Bit4 OF
  0 : And(q12, q16, AddrMode1(instr), ODD(instr DIV Bit20)); RETURN (* AND *)
  | 1 : Eor(q12, q16, AddrMode1(instr), ODD(instr DIV Bit20)); RETURN (* EOR *)
  | 2 : Sub(q12, q16, AddrMode1(instr), ODD(instr DIV Bit20)); RETURN (* SUB *)
...
END DecodeArm
```

```
WHILE running DO
...
reg[PC] := instrAddr + 8;
DecodeARM(instrAddr,
          ReadMemory(instrAddr, 4));
...
END;
```

Beispiel

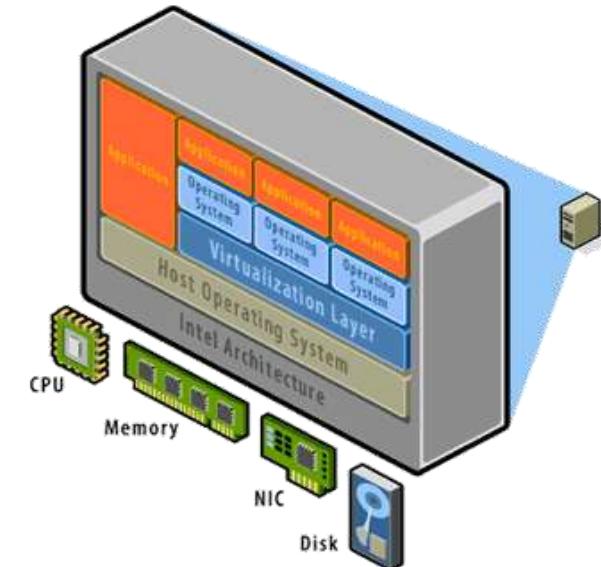
QEMU: Emulation mit virtueller Codegenerierung

- Virtualisiert Prozessoren (x86, PowerPC, ARM ...) auf verschiedenen Hosts (x86, PowerPC, ARM..) in verschiedenen Betriebssystemen (Linux, Windows, FreeBSD...).
- Bestandteile
 - CPU Emulator
 - mit „dynamischer Codegenerierung“: dyngen
 - Emulierte Geräte
 - Generische Geräte (block / character / network), die zur Host-Hardware (über emulierte Geräte) verbunden werden können
 - Maschinenbeschreibungen, Debugger, User Interface

Beispiel

VMware Workstation

- Die VMWare Workstation
 - Erzeugt vollständig isolierte, sichere virtuelle Maschinen (**fully virtualized**)
 - Installiert sich auf dem Host Betriebssystem und bietet entsprechend breite Hardwareunterstützung an
- Jede virtuelle Maschine
 - Kapselt ein Betriebssystem plus Applikationen
 - Besitzt ihre eigene CPU, Hauptspeicher, Disks, I/O Geräte
 - Ist vollständig äquivalent mit einer Standard x86 Maschine
- Der Virtualisierungslayer
 - Bildet die physischen Ressourcen auf die Ressourcen der virtuellen Maschinen ab
- Implementation
 - „Dynamic Code Rewriting“ (für privilegierte Instruktionen)
 - Schattenversionen von Systemstrukturen (z.B. Pagetables)



Beispiel

Xen

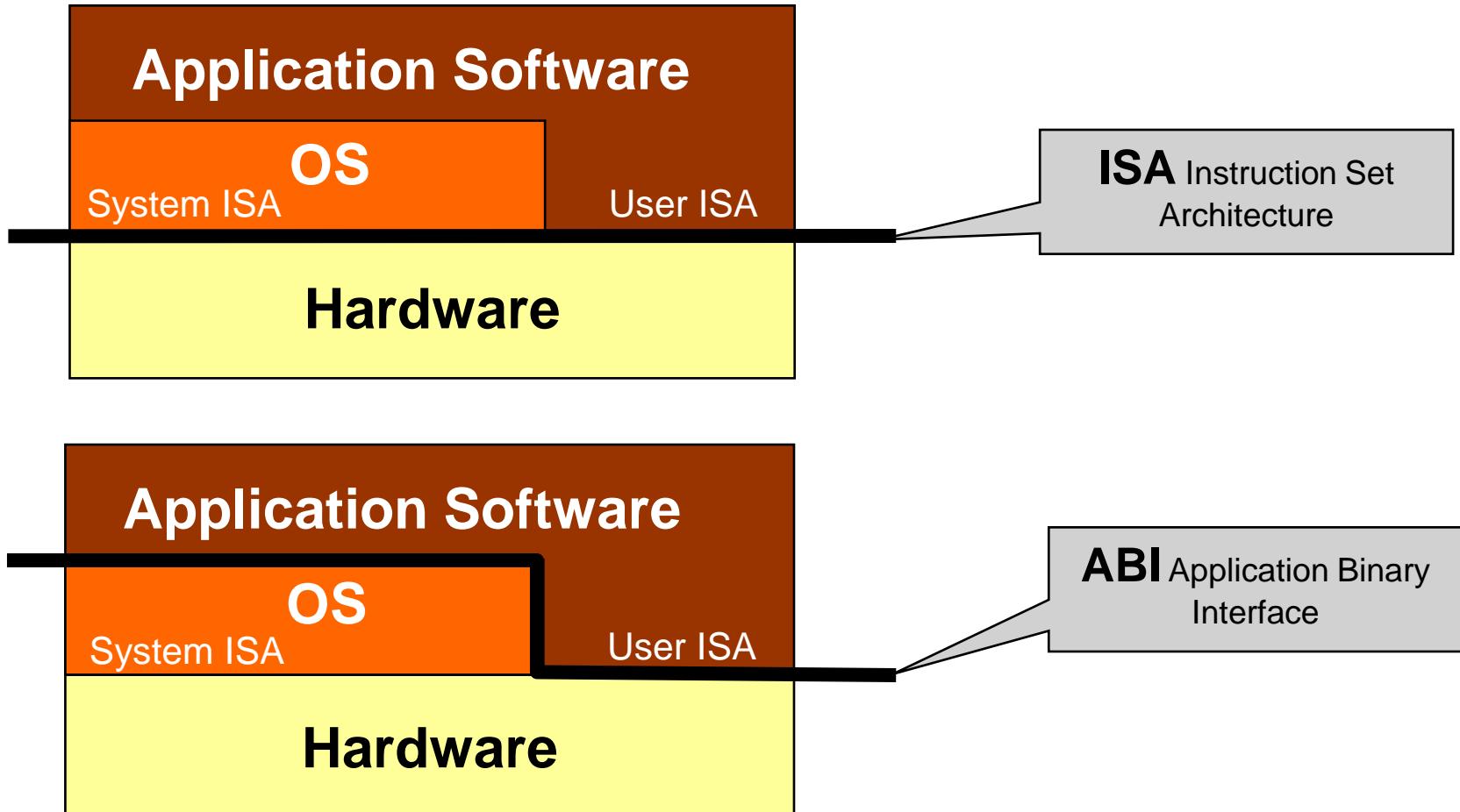
- Multiplexing physikalischer Ressourcen auf der Granularität von Betriebssystemen mit Performance-Isolation
- **Paravirtualisierung**
 - nicht voll virtualisiert. Gründe:
 - Volle Virtualisierung (bisher) nicht unterstützt in IA32/x86 Architektur
 - Vermittlung von realen *und* virtuellen Ressourcen an das Gast-Betriebssystem (z.B. reale / virtuelle Zeit für handling von Timeouts o.ä.)
 - Unterstützung von unmodifizierten Application Binary Interfaces aber:
 - Virtuelle Maschine Abstraktion ähnlich aber nicht identisch mit der Hardware
 - benötigt spezielle Übersetzung der Gast-Betriebssysteme
 - maßgeschneidert auf existierende, große Betriebssysteme (Windows, Linux etc.)
- Overhead im Vergleich zur unmittelbaren Ausführung auf Hardware im Bereich weniger Prozent.

Beispiel

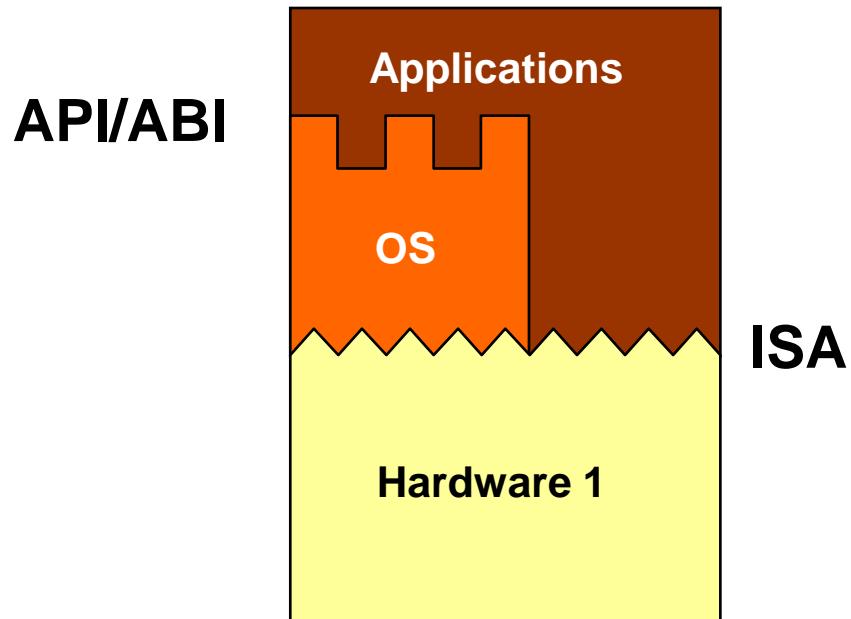
Xen: Implementation

- **Memory Management**
 - direkt auf Hardware Page Tables (Performance!)
 - Paging schwierig, denn x86 unterstützt keinen Software-Managed TLB
 - (1) Gastbetriebssysteme verantwortlich für Hardware Page Tables
 - (2) Xen im oberen 64 MB jedes Addressraumes
 - (3) PageTable Updates / Writes über Xen
 - Segmentation Descriptor Tables
 - (1) mit kleineren Privilegien als Xen
 - (2) Kein Zugriff auf Addressraum von Xen
- **CPU**
 - Host OS muss größeres Privileg haben als jedes Guest OS
 - x86: Xen in Ring 0, User in Ring 3, OS in Ring 1
 - Privilegierte Instruktionen paravirtualisiert
 - Exceptions (PageFaults, Traps) natürlicherweise virtualisiert
 - Fast Direct Exception Handling für System Calls
 - werden von Xen bei Registrierung validiert
- **Device I/O**
 - I/O Datentransport von und zu jedem Domain via Xen (asynchrone Buffers)
 - Lightweight Event Delivery Mechanism (bitmap updates + Aufrufen von vorgängig installierten Event Handlern)

Instruction Set Architecture vs. Application Binary Interface

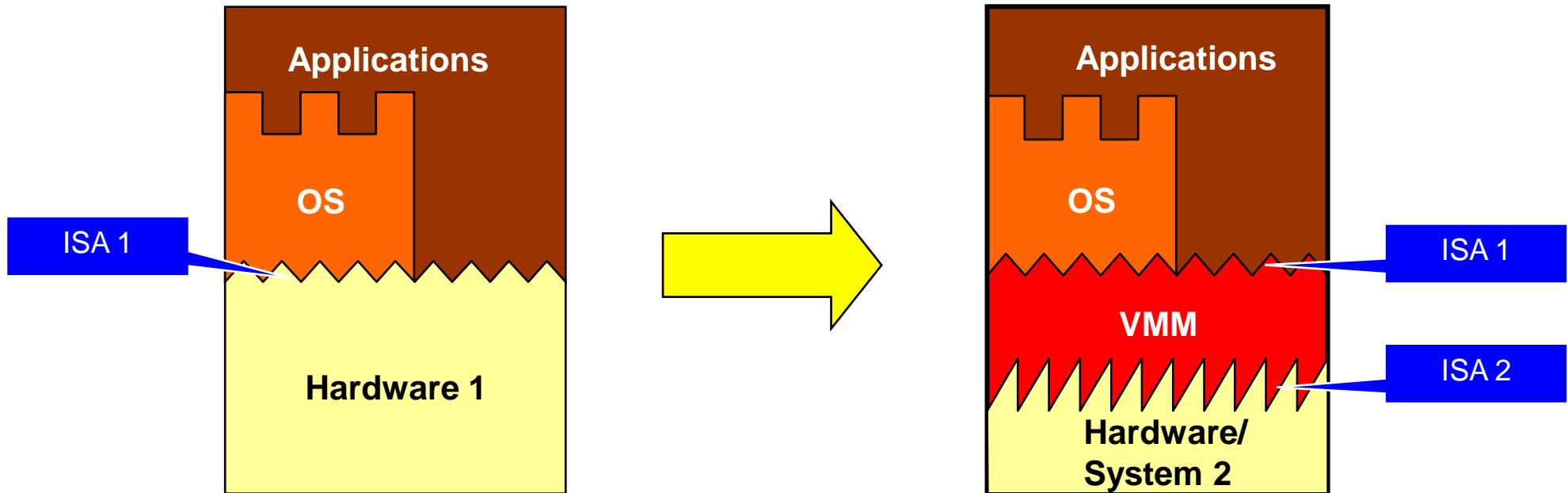


6.2.2. Systematik

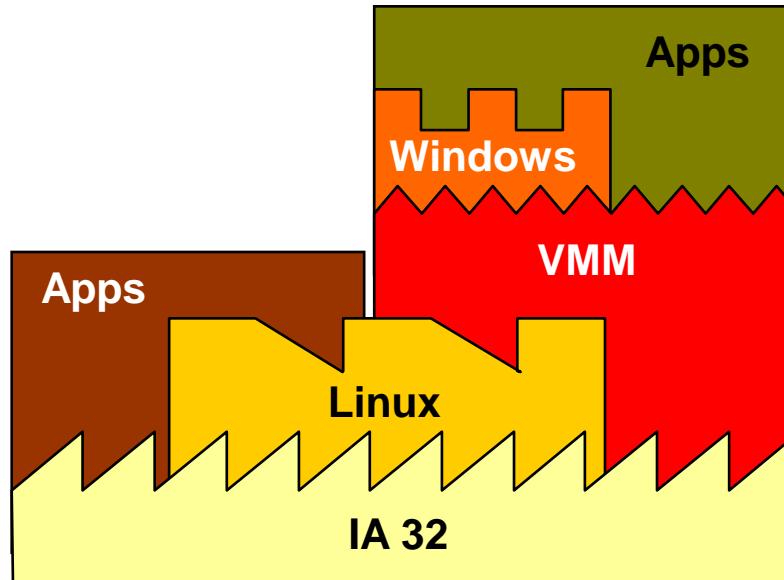


System Virtual Machines

- Virtualisiert ISA auf fremder Hardware Plattform
- Beispiele: VMWare Workstation, QEMU, Bochs

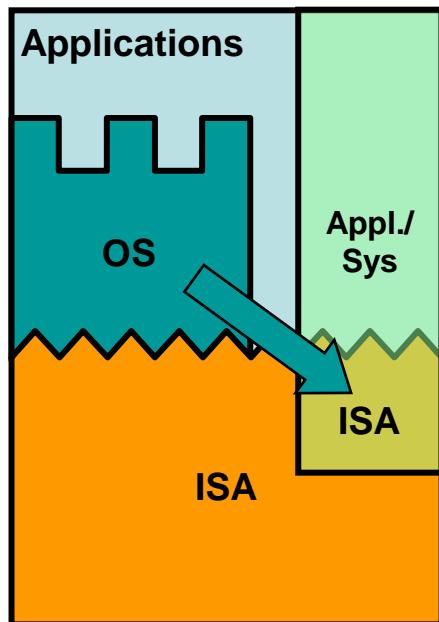


Whole System VM



Hardware Unterstützung

- Wenn gleiche ISA virtualisiert wird: Optimierung möglich durch Verwendung bestehender ISA
- Beispiel: VMWare (z.B. Linux / Windows auf x86)
- Bedingung: Gewisse Hardware-Unterstützung von Virtualisierung bzw. Schutz gewisser Einheiten durch die VM.

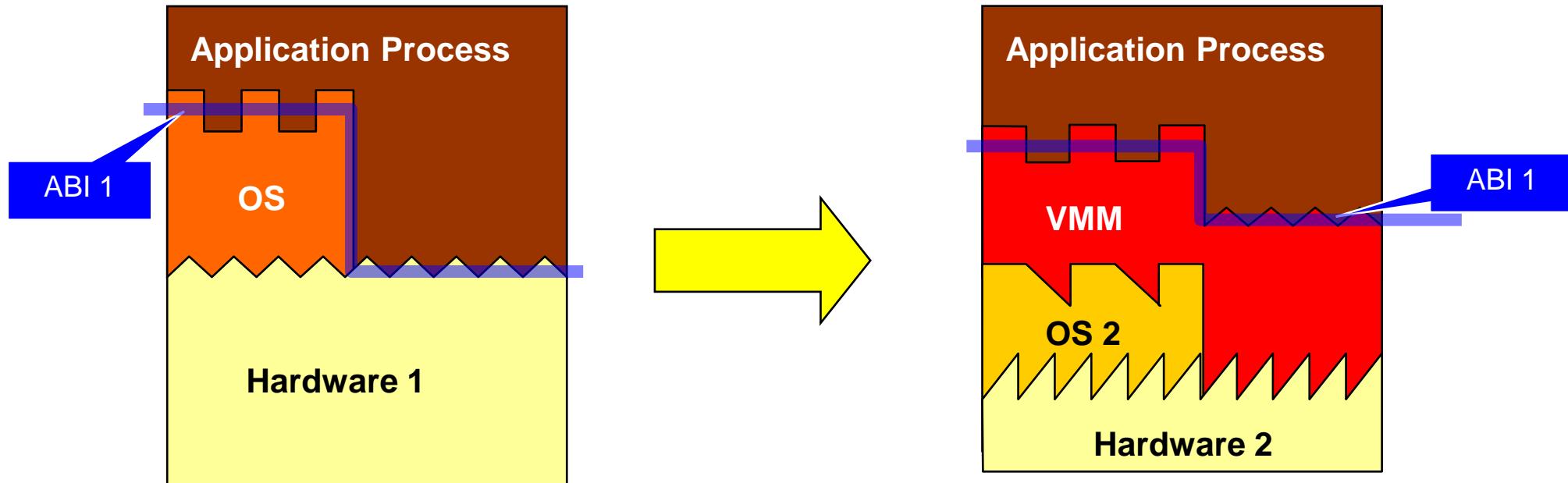


- Intel VMM/VMX Unterstützung
 - VMM: Monitor, VMX: Guest
 - VMX Instruktionen zum Generieren und Kontrollieren einer virtuellen Umgebung
 - VMM kontrolliert Interrupts / Exceptions
 - Guest Interrupt Tabelle
 - Memory Virtualisierung
 - Guest Page Table

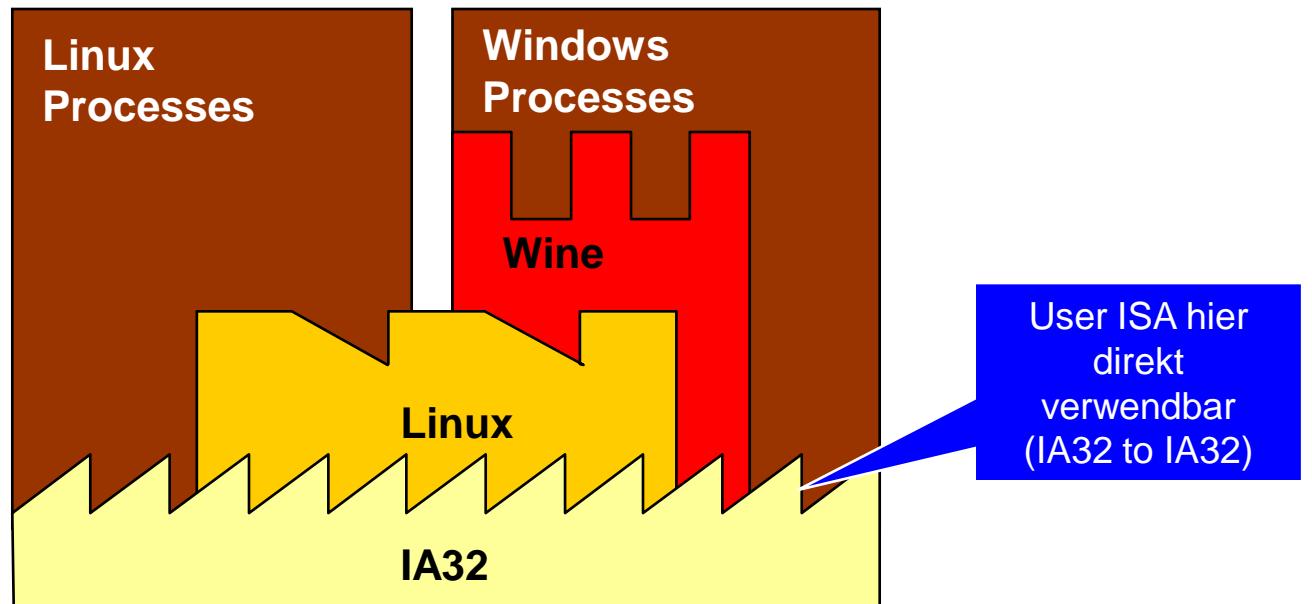
(Intel Virtualization Technology Specification
for the IA-32 Intel Architecture)

Process Virtual Machines

- Stellt unifizierte Laufzeitumgebung unter verschiedenen Betriebssystemen zur Verfügung.
- Virtualisiert ABI
- Beispiele: Wine, High Level Language Virtual Machines



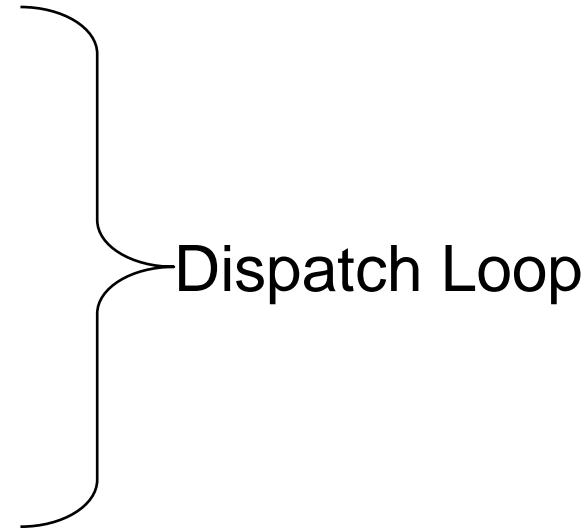
Beispiel: Wine



6.2.3. Implementation

Emulation: Einfache Interpretation der Instruktionen

```
while(!halt && !interrupt)
{
    inst=code[PC];
    opcode = extract(inst,31,6);
    switch(opcode)
    {
        case LoadWordAndZero: LoadWordAndZero(inst);
        case ALU: ALU(inst);
        ...
    }
}
```



LoadWordAndZero(inst){

```
...
    PC=PC+4;
}
```

ALU(inst){

```
...
    PC = PC+4;
}
```

 einfach
 langsam

Optimierungsmöglichkeiten (1)

- (Indirect) Threaded Interpretation

- ohne globalen Dispatch Loop
 - verwendet Dispatch Table

```
ALU(inst){  
    ...  
    routine = dispatch[opcode];  
    call routine;  
}
```

- Predecoding

- separater Decoding-Durchlauf:
Übersetzung der Instruktionen
in Meta-Daten

```
struct instruction= {  
    unsigned long op;  
    unsigned char dest;  
  
    unsigned char src1;  
    unsigbned int src2;  
} code [CODE_SIZE]
```

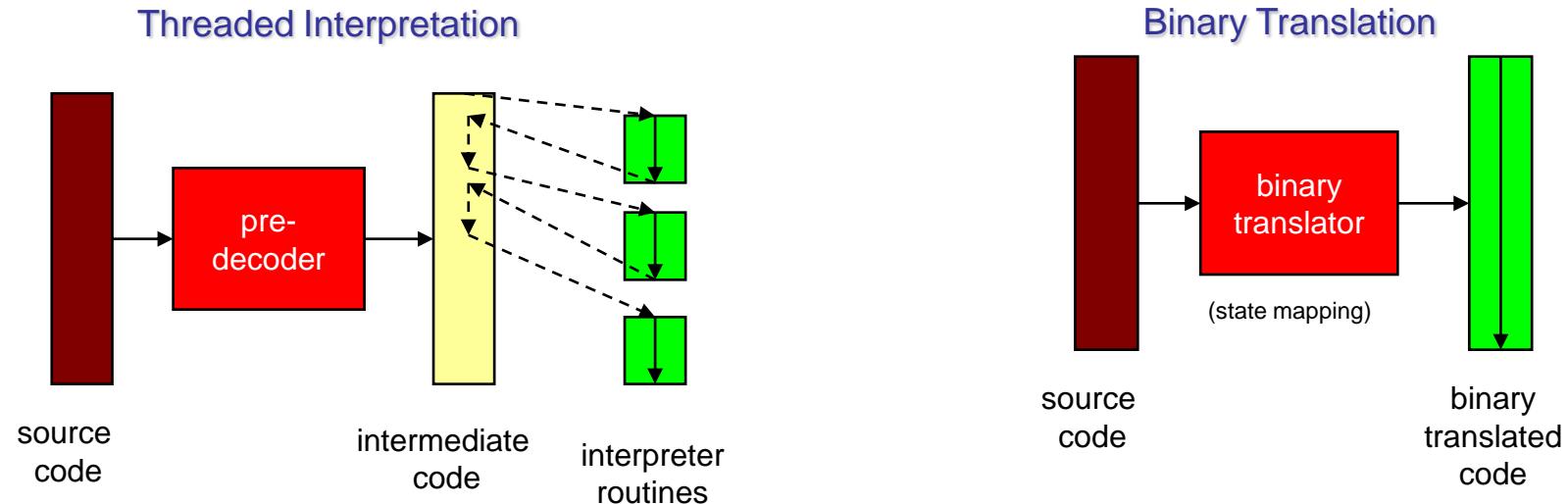
- Direct Threaded Interpretation

- ohne Dispatch Table

```
LoadWordAndZero(inst){  
    ...  
    TPC = TPC + 1;  
    routine = code[TPC].op;  
    call routine;  
}
```

Optimierungsmöglichkeiten (2)

■ Binary Translation



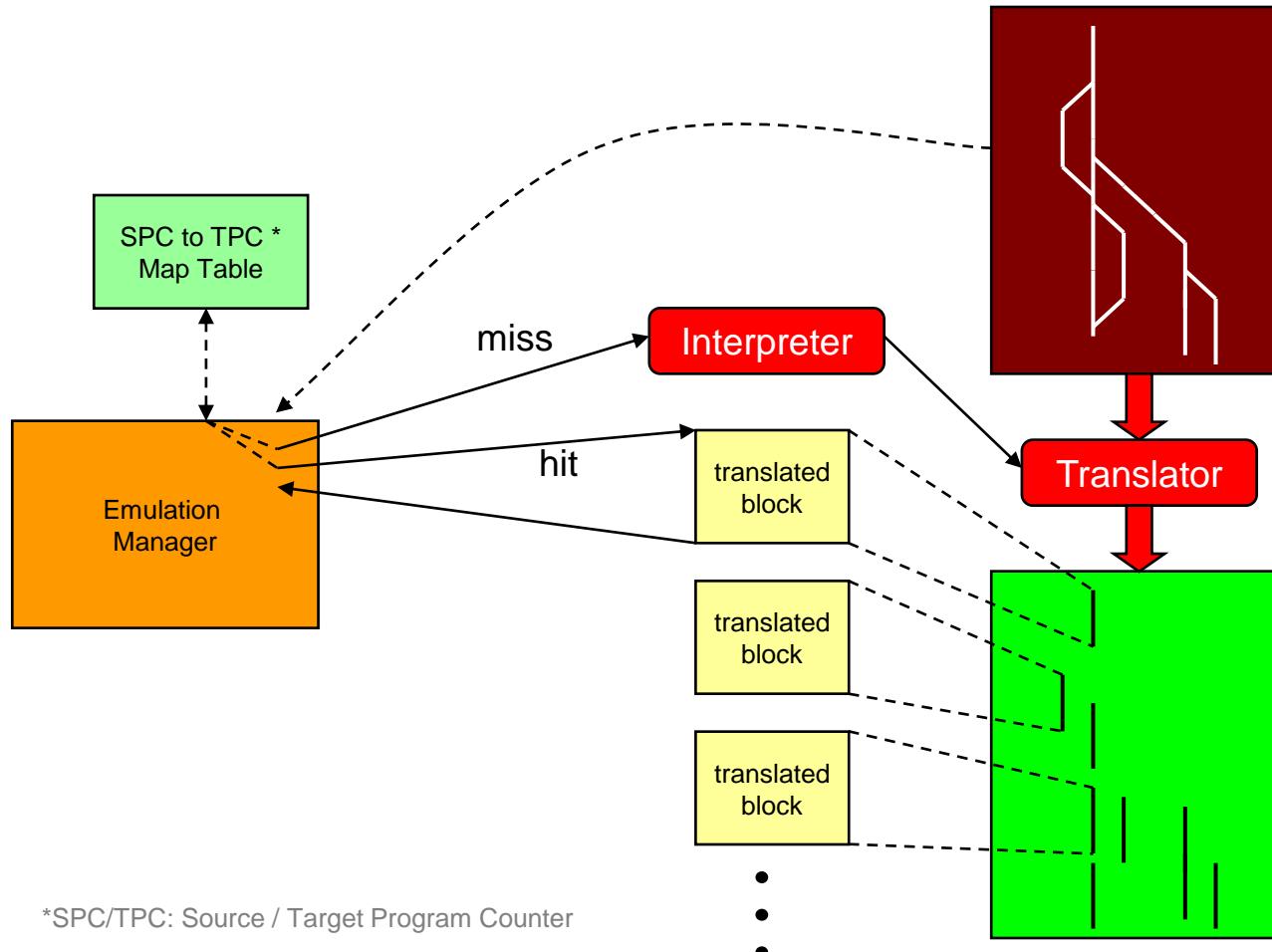
■ Probleme

- statische Übersetzung vs. dynamische Programminformation
 - indirekter Sprung: Ziel nicht vorhersagbar
 - Self Referencing Code: Daten im Code (z.B. Konstanten)
 - Self Modifying Code
- Code Location Problem
 - indirekter Sprung: Zieladresse

■ Lösung: dynamic Translation

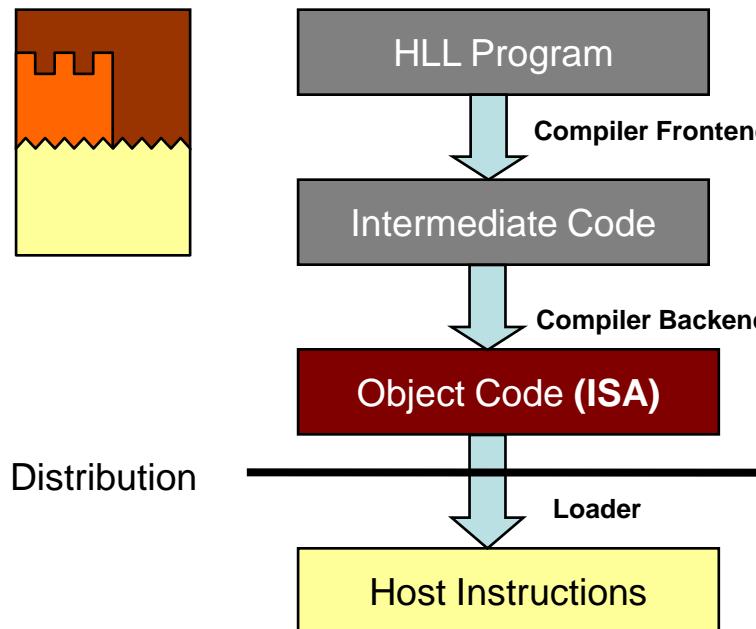
Optimierungsmöglichkeiten (3)

- Dynamic Translation
 - Inkrementelles Predecoding und Übersetzung

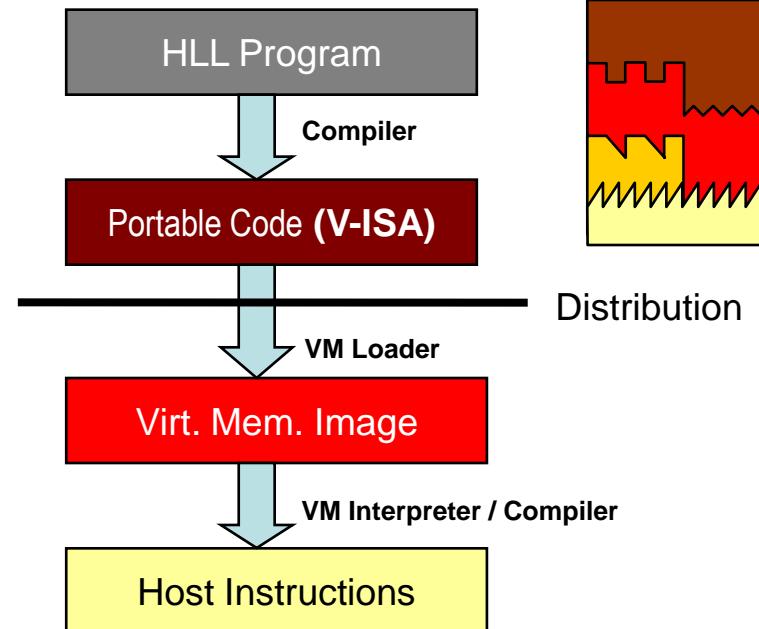


6.2.3. High Level Language VMs

- Spezielle Prozess Virtual Machines
 - virtualisiert keine reale Maschine (Ausnahmen: Sun/ARM Java ISA)
 - Entwurf gemeinsam mit der Applikationsumgebung für die HLL
- Vorteile: Portabilität, zugeschnitten auf HLL
- Beispiele: Java VM, P-Code Unterstützung, .NET CLI



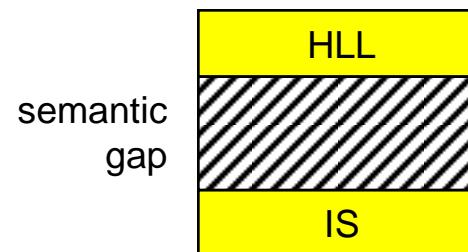
Herkömmliches System



HLL VM Umgebung

High Level Language VMs (2)

- „implementation factorization“
 - m Maschinen, n Sprachen: $m * n$ Compiler
 - VM: m Laufzeitumgebungen + n Compiler
- „semantic factorization“
 - VM schließt die semantische Lücke zwischen High Level Language und Machine Instruction Set.



Pascal P-Code Virtual Machine

(Nori et al. 1973, Wirth 1975)

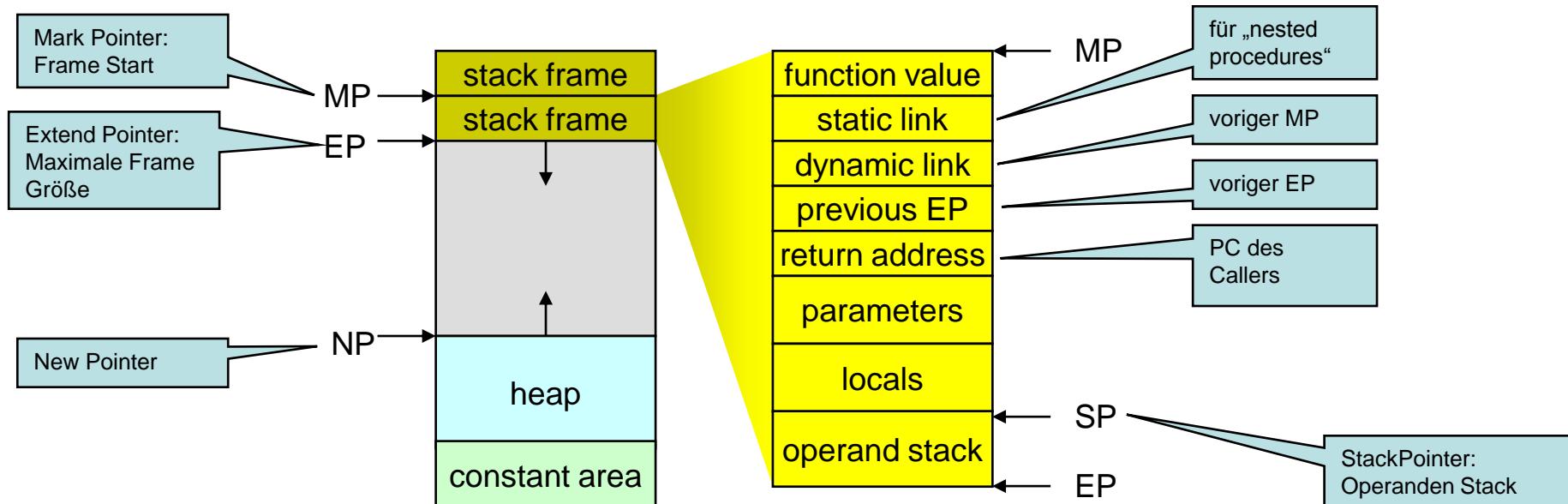
- einfach und historisch bedeutsam
- vereinfachte die Portierung von Pascal Compilern:
 - Parsing des Pascal Programms durch Compiler Front-End, Generierung von P-Code
 - Pascal Compiler liegt in P-Code vor
 - Implementation der VM => Portierung des Pascal Compilers
- P-Code: Stack-orientiertes Instruction Set
- Hauptbestandteile der P VM
 - P-code Emulator
 - Standard Library Routinen

P-Code VM

Speicherarchitektur

Bestandteile

- PC (Program Counter)
 - Momentane Position im Code
- Stack, Constant Area, Heap
 - Datenbereich in „Cells“ eingeteilt



P-Code VM

Instruction Set

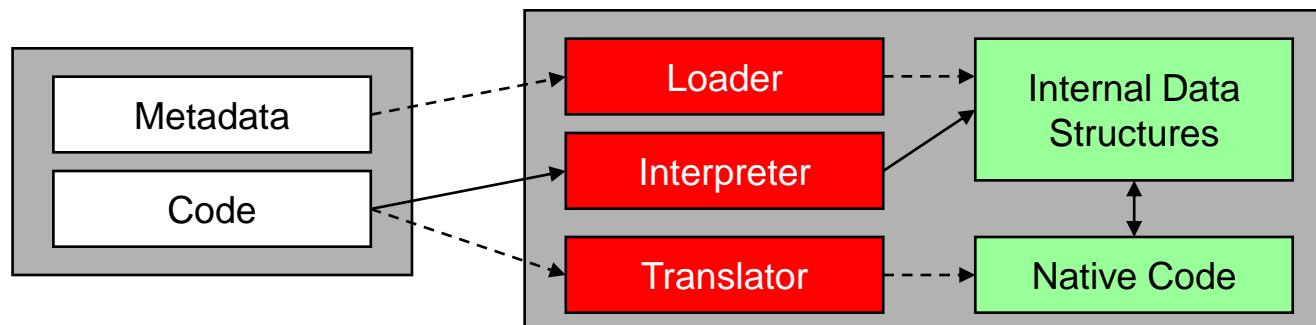
Stack Instruction Set, das sehr wenige Register benötigt

- Stack Operationen (push, pop)
- Logical / Shift / Arithemtik
- Instruktionen sind typisiert
 - addi: add integer
 - addr: add real
- Beispiel: Code, der 1 zu einer lokalen Variable addiert (lokale Variable drei Zellen oberhalb des MP)

```
lodi 0 3 // load variable from current frame (nest 0 depth),  
          // offset 3 from top of mark stack  
ldci 1    // push constant 1  
addi      // add  
stri 0 3 // store variable back to location 3 of current frame
```

Objekt Orientierte HLL VMs

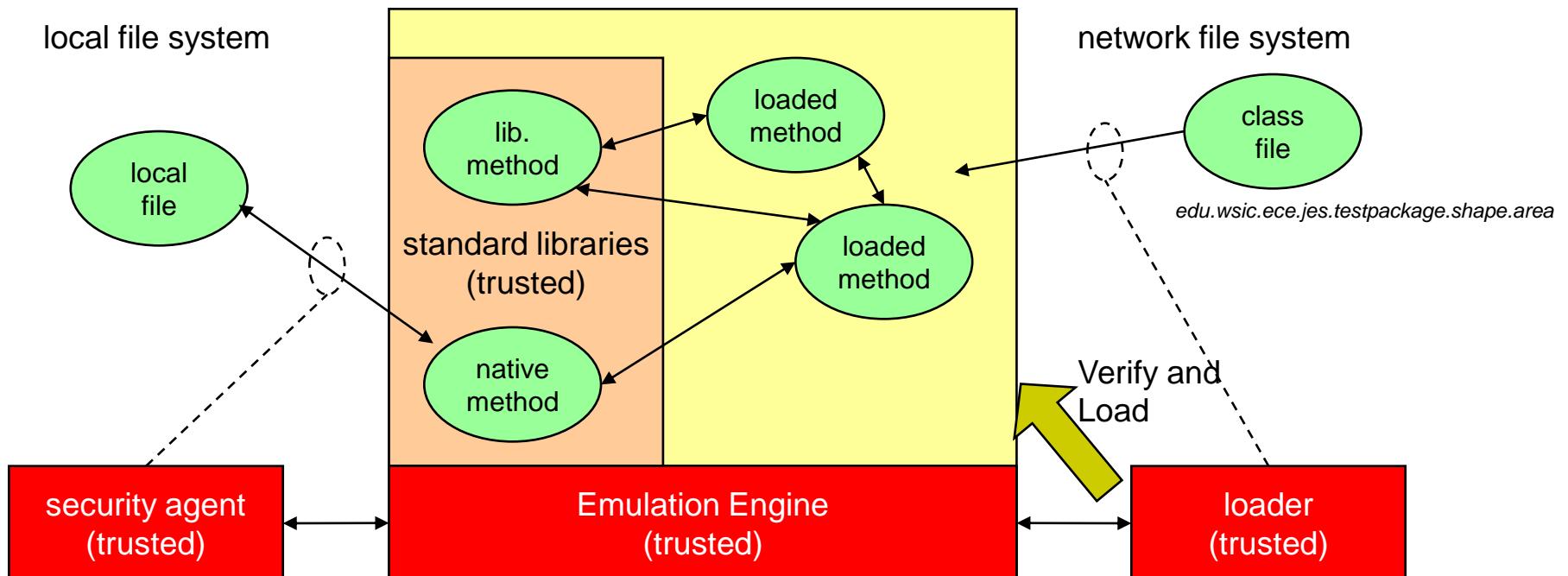
- Wichtige Eigenschaften
 - Portabilität
 - Sicherheit und Schutz („Sandbox“: „managed“ code)
 - Robustheit
 - Performance
- Beispiele
 - Java VM
 - Sun Microsystems Java virtual machine: Java-Programme werden zu Java *binary classes* compiliert (Java bytecode) und (später) von der Java VM geladen und ausgeführt
 - .NET CLI
 - Microsofts **Common Language Infrastructure**: Architektur entworfen für eine große Klasse an Hochsprachen. Code wird zu *Common Intermediate Language* (CLI/MSIL) übersetzt und auf der Common Language Runtime (CLR) ausgeführt.



Sicherheit / Schutz: Sandbox

- VMM und Software läuft im gleichen Prozess, kein Schutz durch Hardware. Daher Schutz durch die VM selbst.
- Schutz nach innen: stark typisierte Sprachen
 - möglichst viel statischer Schutz (Compile time)
 - darüber hinaus dynamischer Schutz (Runtime: z.B. Array Bounds, Typecasts, NULL-Pointer Check etc.)
- Schutz nach außen: Sandbox

Komponenten der Java Protection Sandbox



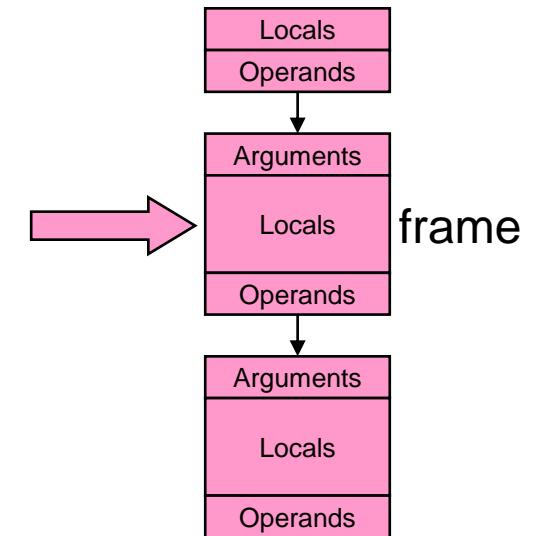
6.2.3.1. Java VM (1)

(Lindholm and Yellin 1999)

- Java Platform: Kombination JVM und standard Libraries (APIs)
 - Java VM: virtueller Instruktionssatz (Sprache) und Typsystem plus Laufzeitumgebung.
 - Die Java VM „weiß nichts“ von der Sprache Java, sie kennt nur das binary *class* file format
- Definierte Datentypen der JVM
 - einfache Datentypen
 - int, char, byte, short, long, float, double
[boolean in der JVM als int oder char implementiert]
 - returnAddress (benutzt von jsr und ret)
 - Referenzen
 - Objekte (class)
 - Arrays (array),
 - Interfaces (interface)

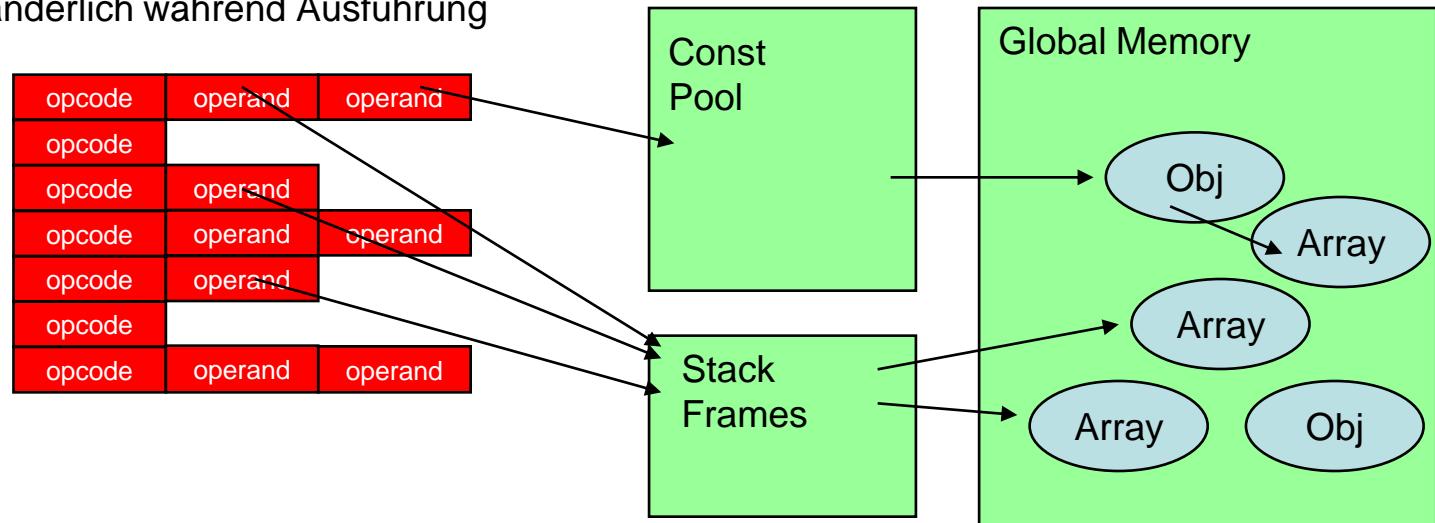
Java VM (2)

- Weitere definierte Inhalte
 - Das pc Register (für jeden Thread eines)
 - Stack
 - Heap-Allozierbar
 - statische oder dynamische Größe
 - nicht notwendigerweise zusammenhängend
 - wird zusammen mit Threads erzeugt
 - Heap
 - managed (Garbage Collection)
 - wird beim Start der VM instanziert
 - Methodenbereich
 - wird beim Start der VM instanziert
 - Constant Pool
 - alloziert mit dem Methodenbereich
 - Native method stacks
 - für Laufzeitunterstützung nativer Programme (C-Code etc.)



Java VM (3)

- Speicherarten
 - Stack
 - **lokale Variablen** und **Operanden**, Prozedurparameter
 - keine Objekte, kein Code
 - Globaler Speicher
 - Code
 - Arrays, Objekte (**global**)
 - managed (GCed)
 - Constant Pool
 - Pool der im Code verwendeten Konstanten
 - unveränderlich während Ausführung



Java VM (4)

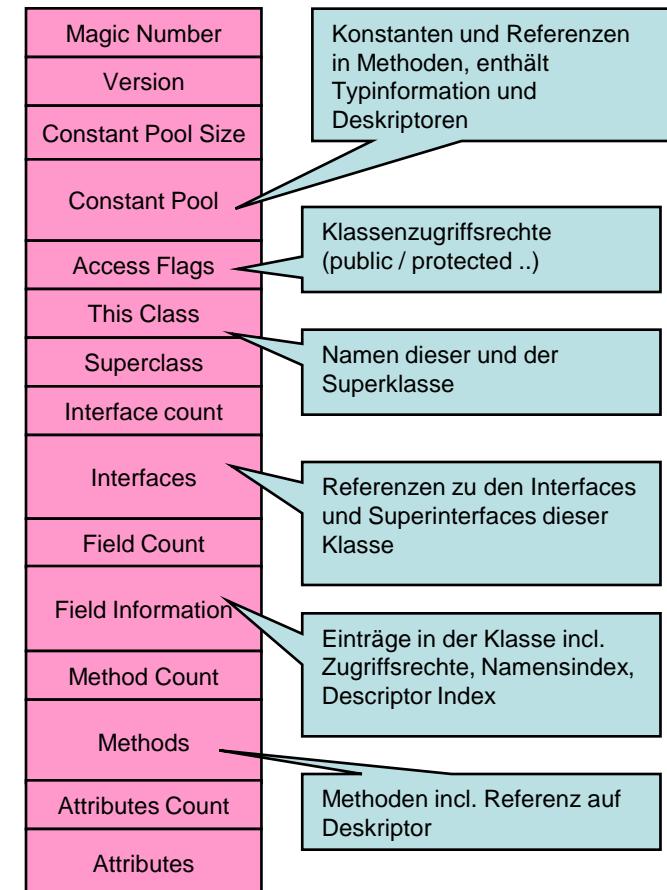
1 Byte (=> „Bytecode“)

■ Java Instruction Set

- Format: opcode [operand [operand]]
 - Arithmetik
 - Typkonversion
 - Objekterzeugung und Manipulation
 - Stack Operationen (Push, Pop)
 - Control Transfer (Branching)
 - Methodenaufruf und Rücksprung
 - Exceptions
 - Monitorbasierte Concurrency

■ Binäre Klassen

- Code
- Metadaten
- Interface



Operand Stack Tracing

- Java VM verlässt sich auf genaue Stack-verfolgung (Stack Tracing)
- Legaler Beispielcode (geprüft durch den Byte Code Verifier)

```
        iload A          // push int A from local mem
        iload B          // push int B from local mem
        IF_cmpne 0 else1 // branch if B ne 0
        iload C          // push int C from local mem
        goto endelse1
else1      iload F          // push F
endelse1   add             // add from stack; result to stack
            istore D       // pop sum to D
```

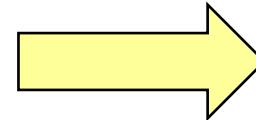
Operand Stack Tracing

- Illegaler Beispielcode

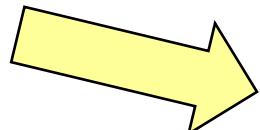
```
        iload B          // push int B from local mem
        If_cmpne 0 skip1 // branch if B ne 0
skip1      iload C          // push int C from local mem
           iload D          // push int D
           iload E          // push int E
           If_cmpne 0 skip2 // branch if E ne 0
skip2      add             // add from stack; result to stack
           istore F         // pop sum to F
```

Beispiel: Euklidischer Algorithmus

```
class gcd {  
    static int gcdof ( int a, int b ) {  
        int t;  
        while (b > 0)  
        {  
            t = a % b; a = b; b = t;  
        }  
        return a;  
    }  
    static void main(String[] args)  
    {  
        System.out.println(gcdof(777,555));  
    }  
}
```



```
0 0:iload_1  
1 1:ifle      15  
2 4:iload_0  
3 5:iload_1  
4 6:irem  
5 7:istore_2  
6 8:iload_1  
7 9:istore_0  
8 10:iload_2  
9 11:istore_1  
10 12:goto     0  
11 15:iload_0  
12 16:ireturn
```



```
0 0:getstatic #2 <Field PrintStream System.out>  
1 3:sipush    777  
2 6:sipush    555  
3 9:invokestatic #3 <Method int gcdof(int, int)>  
4 12:invokevirtual #4 <Method void PrintStream.println(int)>  
5 15:return
```

Java VM (5)

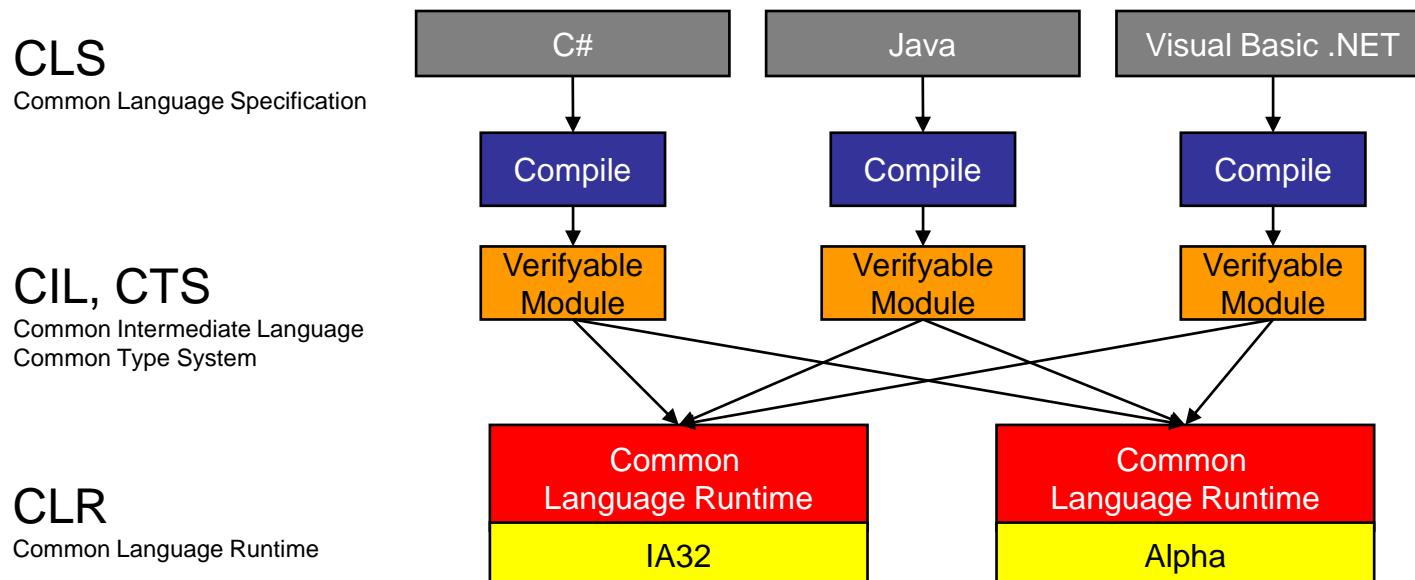
- Java Native Interface
 - ermöglicht Zugriff auf native compilierten Code
- APIs
 - Java Platformen (J2SE,J2EE,J2ME)
 - Standard Spezifikationen
 - Java APIs
 - java.lang, java.util, ...
- Java Threads
 - class Threads in java.lang definiert
 - run, stop, suspend, resume
 - Monitor Konzept als Instruktionen in der Java VM (verbunden mit Methodenaufruf und return)
 - monitorenter, monitorexit
 - zusätzliche Synchronisation in class *Object*
 - wait(), notify, notifyAll

6.2.3.2 Die .NET CLI*

(Box 2002)

■ Vergleich mit Java VM

- Java VM designed für Plattform-Unabhängigkeit mit Java
- CLI strebt Plattform- und HLL-Unabhängigkeit an



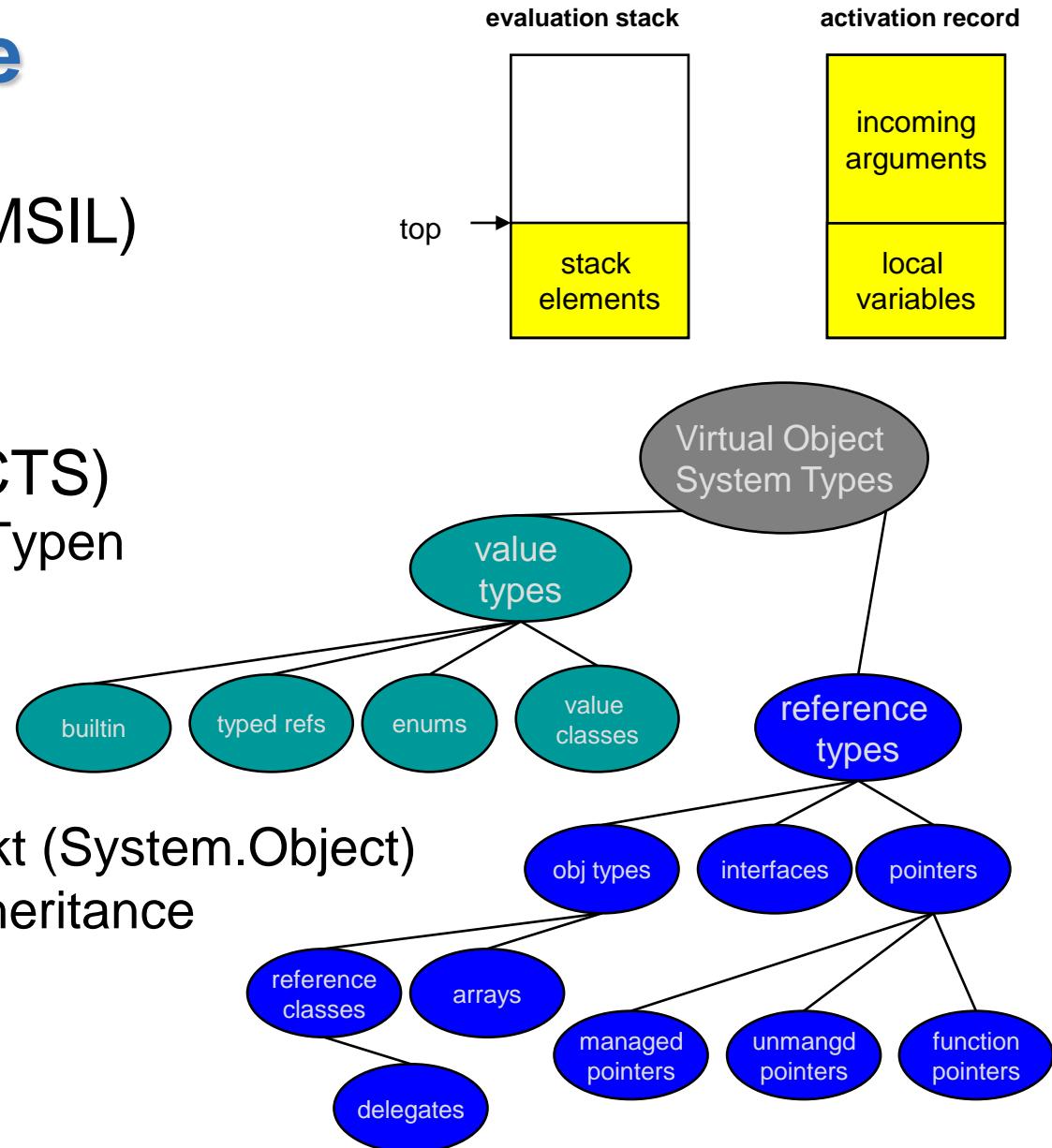
*Common Language Infrastructure

Verifiable vs. unverifiable Code

- Für eine garantiierte Laufzeitsicherheit muss der Code gewisse Bedingungen erfüllen (**verifiable code**), z.B.
 - „managed data“
 - Allokation der Objekte nur im managed heap (Garbage collected)
 - Self describing objects
 - Typsicherheit des Codes
 - Signaturprüfung bei Methodenaufruf
- Stellt auch Bedingung an die Programmiersprache
 - z.B. **unverifiable**:
 - union types
 - pointer Arithmetikkönnen nur innerhalb einer Sprache verwendet werden, dürfen nicht exportiert werden
- Um möglichst viele Sprachen zu unterstützen, kann ein Compiler in .NET entweder verifiable Code (Java, C#) oder unverifiable Code (ANSI C) generieren.

.NET Virtual Machine

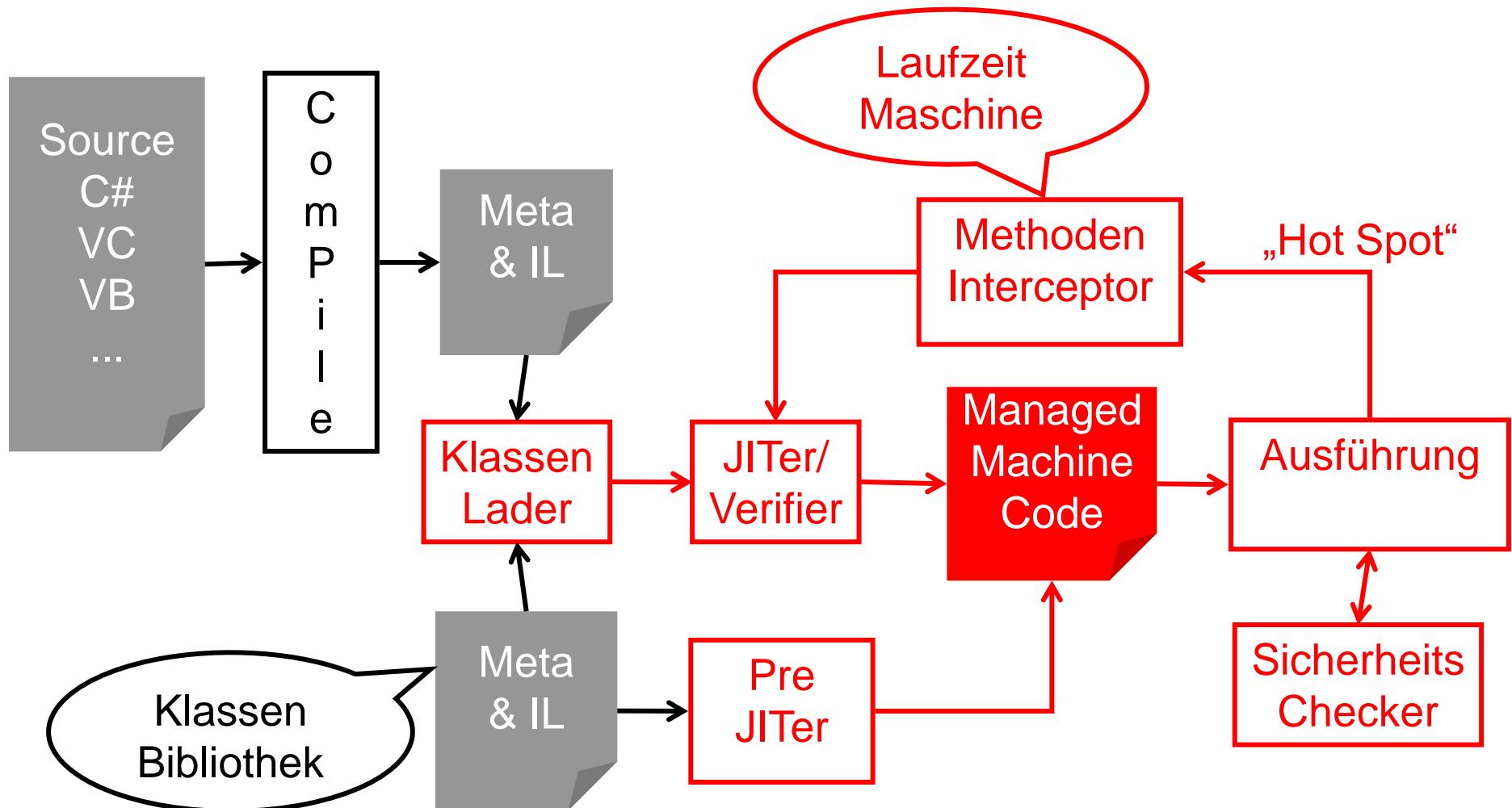
- Intermediate Language (MSIL)
 - Abstrakte Stack Machine
 - Evaluation Stack
 - Activation Record
- Common Type System (CTS)
 - Grundunterscheidung der Typen
 - Wertetypen (Stack)
 - Referenztypen(Heap)
 - Boxing / Unboxing
 - Metadaten
 - Gemeinsames Wurzelobjekt (System.Object)
 - Unterklassen mit Single Inheritance
 - Schnittstellen
 - Exception Handler



.NET Virtual Machine (2)

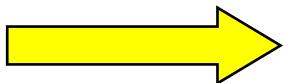
- Metadaten in proprietärem Format
 - normierte Datenbeschreibung
 - Unterklassen- und Implementationsbeschreibung
 - Signaturen von Feldern, Methoden, Properties, Delegates
 - System Reflection
 - im Compiler Frontend
 - zur Laufzeit
 - Debugging
- Common Language Runtime (CLR)
 - Just-In-Time-Compiler (JIT)
 - Class Loader & Verifier (keine Interpretation, vgl. Java)
 - CTS Runtime
 - Reflection
 - Memory Manager (Garbage Collector)
 - Exception Handler (try ... catch ... finally)

Die Execution Engine



Beispiel: Euklidischer Algorithmus

```
using System;
namespace Euclid {
    class gcd {
        static int gcd_of( int a, int b ) {
            int t;
            while (b > 0) {
                t = a % b; a = b; b = t;
            }
            return a;
        }
        static void Main()
        {
            System.Console.WriteLine(gcd_of(777,555));
        }
    }
}
```



JIT optimiert

```
.method private hidebysig static int32
gcd_of(int32 a, int32 b) cil managed {
// Code size          22 (0x16)
.maxstack 2
.locals init ([0] int32 t, [1] int32 cs$00000003$00000000)
IL_0000: br.s      IL_000c
IL_0002: ldarg.0           // 000014: t = a % b;
IL_0003: ldarg.1           // 000015: a = b;
IL_0004: rem
IL_0005: stloc.0
IL_0006: ldarg.1           // 000016: b = t;
IL_0007: starg.s a
IL_0009: ldloc.0           // 000017: 
IL_000a: starg.s b
IL_000c: ldarg.1           // 000018: while (b > 0)
IL_000d: ldc.i4.0
IL_000e: bgt.s      IL_0002
IL_0010: ldarg.0           // 000019: return a;
IL_0011: stloc.1
IL_0012: br.s      IL_0014
IL_0014: ldloc.1           // 000019: }
IL_0015: ret
} // end of method Mcd::gcd_of
```

Zusammengefasst: Vergleich .NET / Java

- .NET für mehrere Sprachen konzipiert (offener)
Java für und mit einer Sprache konzipiert (stringenter)
- Java Programme werden anfänglich interpretiert (schneller startbereit)
.NET wird JIT (just in time) compiliert (keine Interpretation nötig, sofort performant)
- kleinere Unterschiede
 - JVM unterstützt nur Call by Value Parameter
.NET auch Call by Reference Parameter
 - Java unterstützt nur Objekte auf dem Heap (ausgenommen Optimierungen)
.NET unterstützt Objekte am Heap und Stack („boxing“)
 - Java unterstützt nur 1-d Matrizen (einfacher)
.NET unterstützt Blockmatrizen (performanter)

Kapitel 7. Fallstudien

Windows & Linux

- Geschichte und andere Gemeinsamkeiten
- Windows Kern
- Linux Kern

- kein erschöpfender Vergleich der Systeme
 - Auswahl gewisser Themen
 - keine Qualitätsbeurteilung

- Quellen:
 - *Windows Operating System Internals Curriculum Development Kit*, developed by David A. Solomon and Mark E. Russinovich with Andreas Polze (© Microsoft)
 - Mark E. Russinovich, David Solomon, Microsoft Windows Internals, Fourth Edition, Microsoft Press 2005
 - Understanding the Linux Kernel, 3rd Edition, by Daniel P. Bovet, Marco Cesati, O'Reilly 2005

Geschichte

Unix

- 1969: Ken Thompson entwickelt erste UNIX-Version am Bell Labs
 - Dennis Ritchie (Designer der Sprache C) beteiligt sich am Projekt: 1974 Veröffentlichung des Papers „The Unix Time Sharing System“
 - 1976: erste kommerzielle Version von UNIX (Unix V6)
- UNIX wird an Universitäten eingesetzt und 1978 wird das Unix Time-Sharing System (Portabilität!) von Bell Labs herausgegeben.
- Bell Labs gaben UNIX mit Quell-Code heraus. Es entwickelten sich drei wichtige Seitenzweige:
 - UNIX System III von der Bell Lab's UNIX Support Group (USG)
 - UNIX Berkeley Source Distribution (BSD) der University of California at Berkeley
 - Microsoft's XENIX
- Trotz Standardisierungsbemühungen (IEEE POSIX, X/Open Group Portability Guide) ist Unix in den 1980er Jahren weiter zersplittert.

Linux

- 1991 besuchte Linus Torvalds einen Informatikvorlesung, die das Minix OS einsetzte
 - Minix ist ein minimales UNIX-artiges OS von Andrew Tanenbaum, primär zu Lehrzwecken erstellt.
 - Minix 3: System mit Microkern, noch heute in Entwicklung.
 - Linus Torvalds wollte Minix für realen Einsatz tauglich machen aber Andrew Tanenbaum wollte es super-einfach halten.
- Linus Torvalds ging danach eigene Wege und arbeitete an Linux
 - Okt. 1991: Linux v0.02
 - März 1994: Linux v1.0
- Heute
 - sehr weit verbreitet.
 - aktuelle Version: Linux Kernel vs. 2.6.

Windows

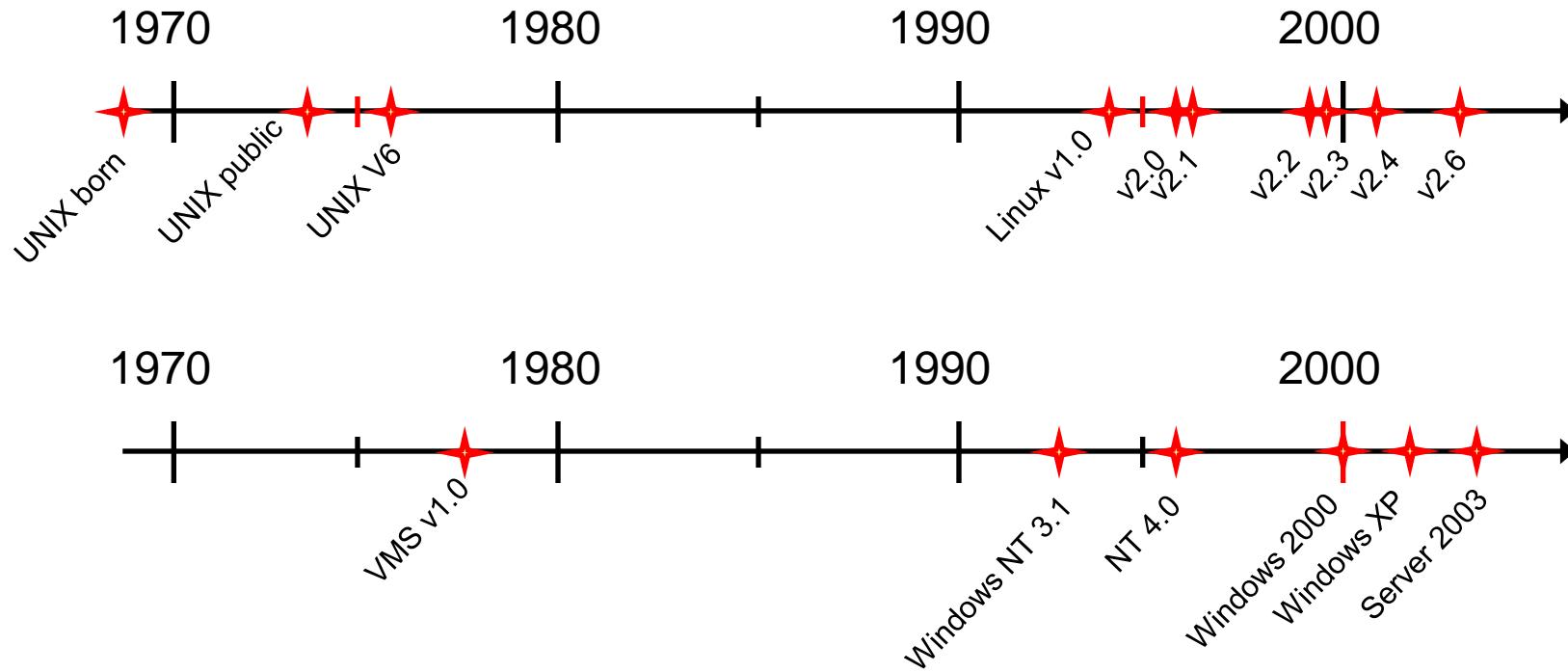
- Die Geschichte von Windows beginnt in den 1970er Jahren mit dem Design des Betriebssystems VMS für einen 32-bit VAX Prozessor (Dick Hustvedt, Peter Lipman, David Cutler bei Digital Equipment Corporation DEC)
 - 1978: VMS v1.0 wird herausgegeben
- Cutler ging nach Seattle um DECWest zu eröffnen und arbeitete am Digital Mica OS für eine neue CPU (mit dem Codenamen Prism).
 - Bis zu 200 Ingenieure arbeiteten mit
 - 1988 wurde das Projekt gestoppt.
- Bill Gates wollte einen UNIX Rivalen aufbauen
 - er stellte Cutler und 20 Ingenieure ein (1989)
 - Das neue Projekt hieß NT OS/2
- Mit dem Erfolg von Windows 3.0 (1990) wurde die Bestrebung in Richtung OS/2 aufgegeben
 - Das Projekt wurde Windows NT genannt
 - NT kommt im August 1993 auf den Markt.
- Heute
 - sehr weit verbreitet.
 - aktuelle Version: Windows Vista (NT 6)

Gemeinsamkeiten

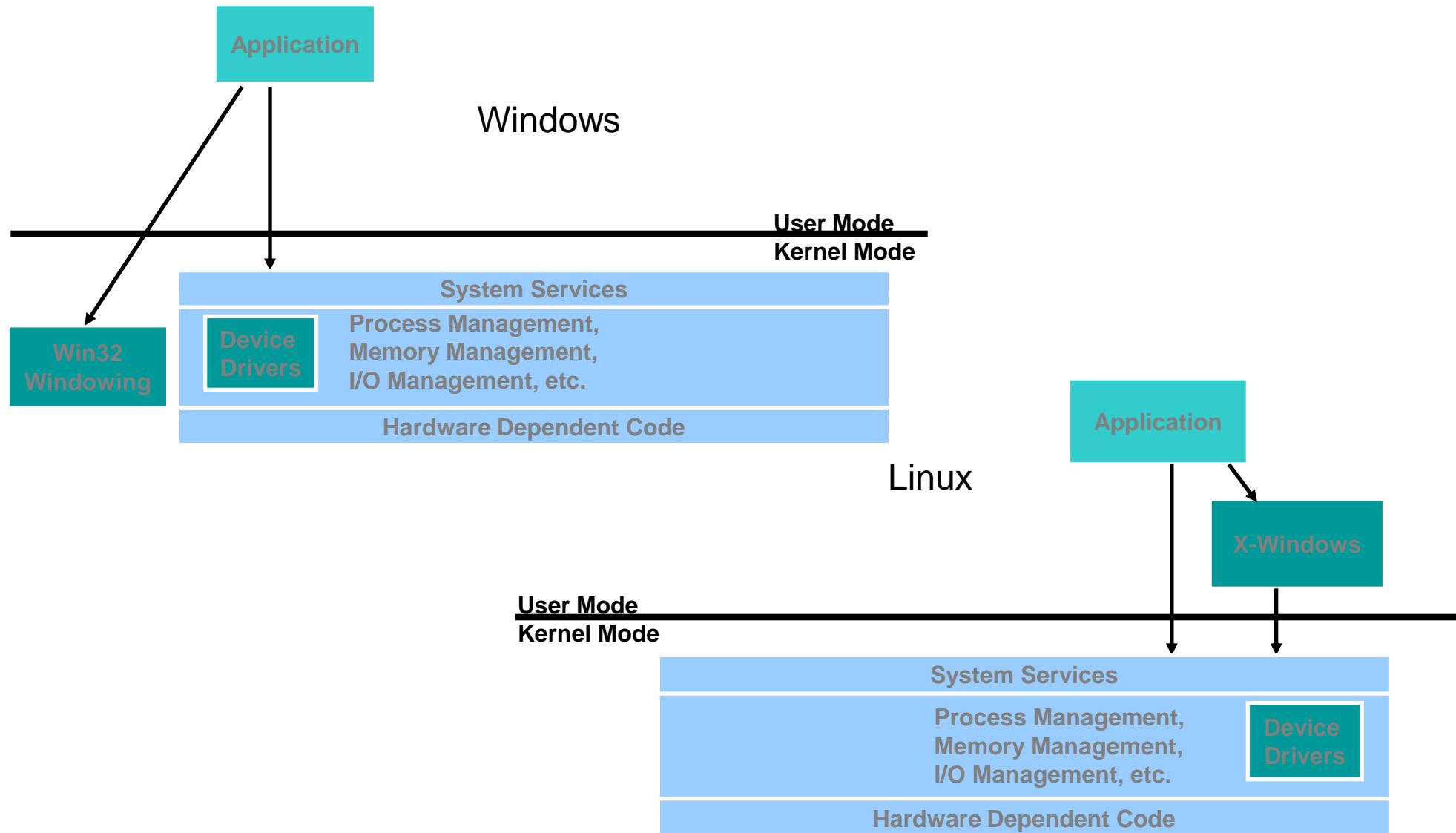
- Sowohl Windows als auch Linux basieren auf Ihrer Gründung in den 1970er Jahren.
- Sowohl Windows als auch Linux haben ursprünglich einen monolithischen Kern (heute: geschichteter Kern)
 - Basale Betriebssystem-Funktionen laufen in einem gemeinsamen Adressraum im Kernel-Mode
 - Die System-Services sind im wesentlichen Teil eines einzelnen Pakets
 - Linux: vmlinuz
 - Windows: ntoskrnl.exe
- Die GUI ist verschieden gestaltet
 - Windows: Kernel-Mode Windowing Subsystem
 - Linux: User-Mode X-Windowing system

Windows und Linux

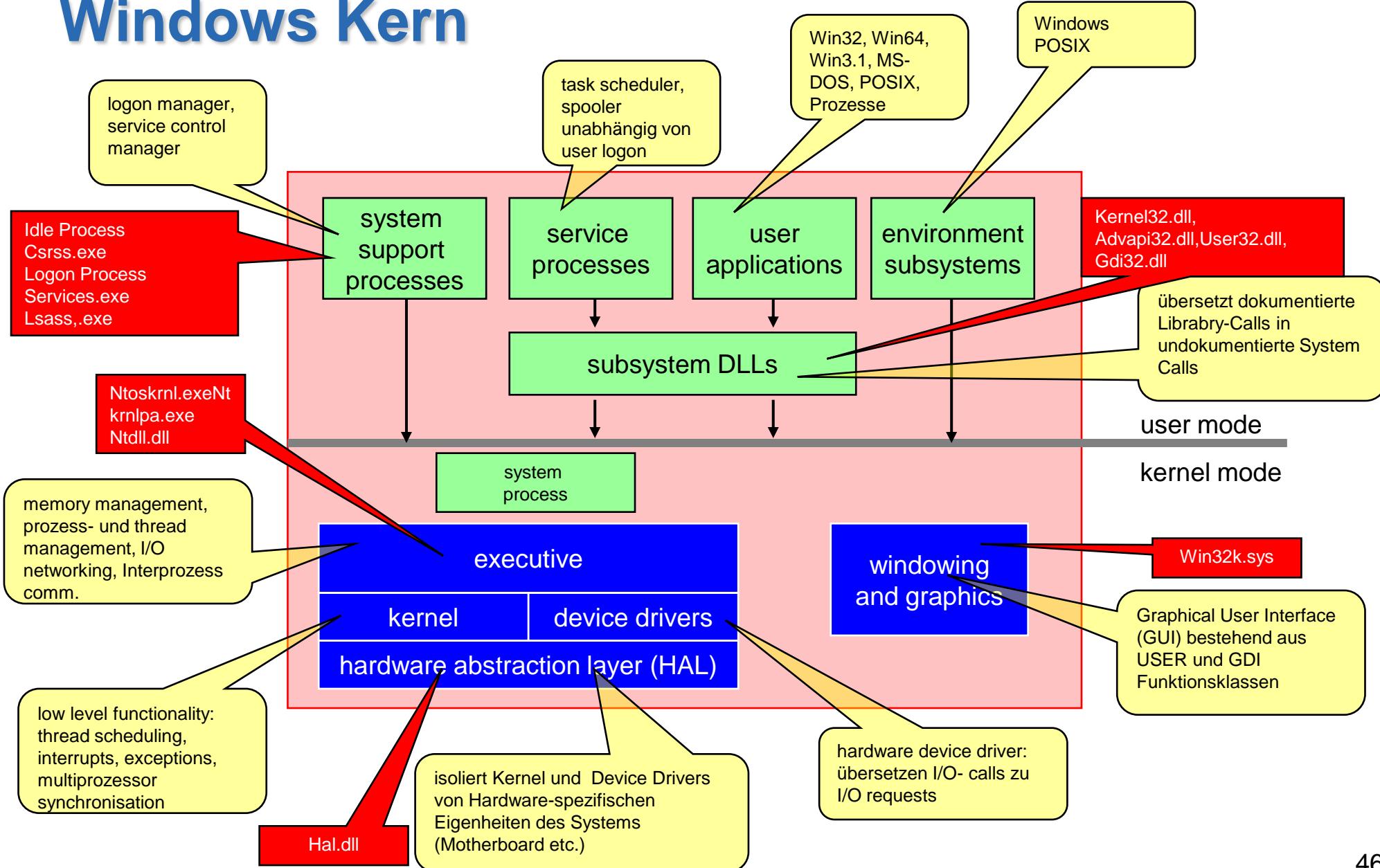
- Linux und Windows basieren beide auf Betriebssystemen aus den 70er Jahren



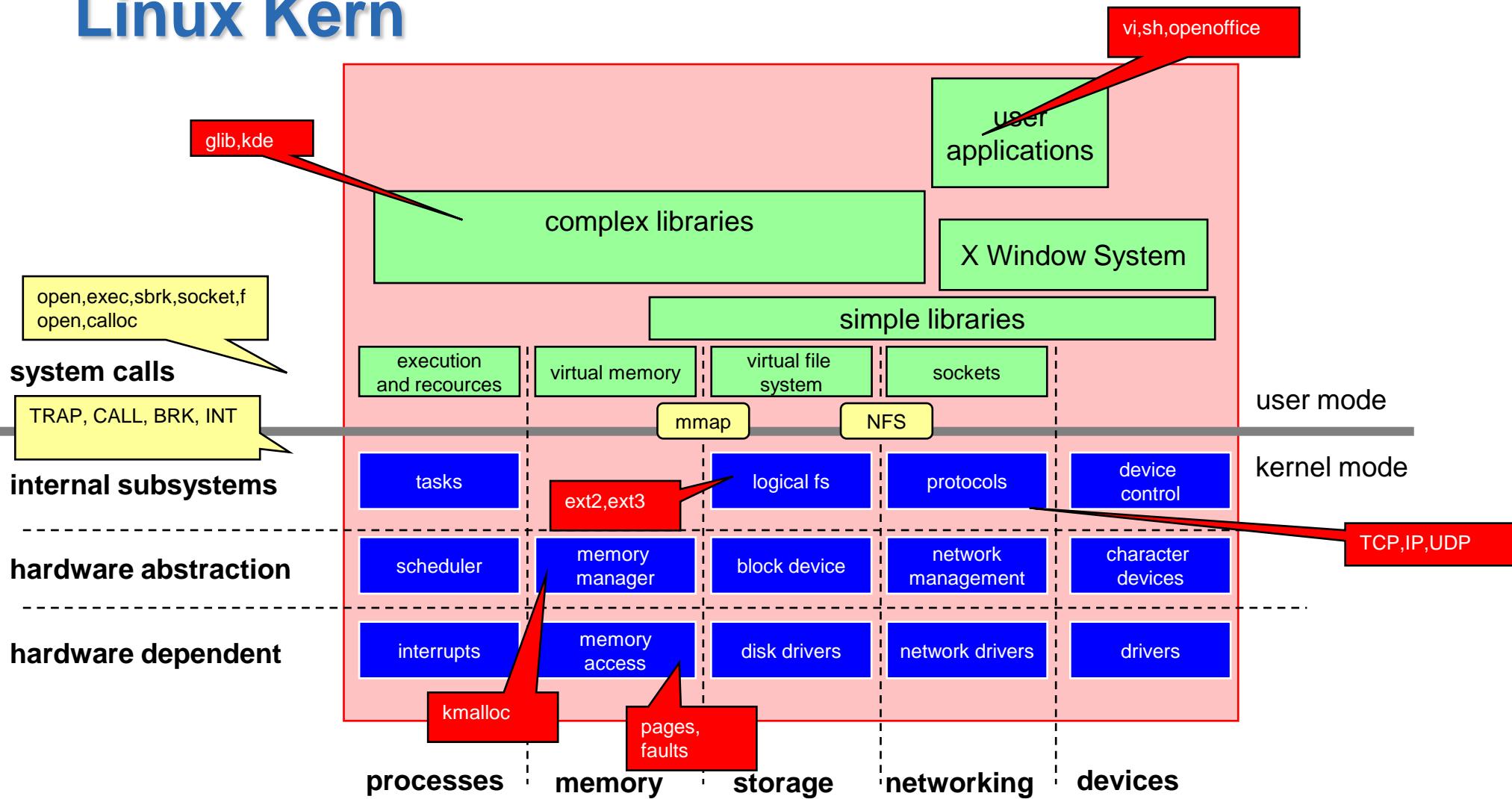
Kern Architekturen



Windows Kern



Linux Kern



Module (Kern)

- Linux Unterstützung für Modularität
 - Compiler Optionen
 - Viele Komponenten des Kerns können als dynamisch ladbare Kern-Module (DLKM) implementiert werden
- Linux DLKMs
 - Separat vom Kern erstellt
 - Zur Laufzeit in den Kern eingehängt (automatisch oder „on demand“)
 - Module können inkrementell erweitert werden
 - Unterstützung eines minimalen Kerns, der benötigte Komponenten automatisch maschinenangepasst nachlädt.
- Windows Unterstützung für Modularität schwächer:
 - Windows Drivers erlauben eine dynamische Erweiterung der Kern-Funktionalität
- Windows Drivers sind dynamisch ladbare Kern-Module
 - Großer Teil des Code läuft als Driver (z.B. network stacks, TCP/IP und viele Services)
 - Unabhängig vom Kern erstellt
 - Können auf Verlangen geladen werden
 - Abhängigkeiten zwischen Drivers kann spezifiziert werden

Portabilität

- Linux und Windows Kerne sind weitgehend portabel
 - geschrieben in C
 - Portierung zu einer großen Menge Architekturen
 - Windows
 - x86, MIPS, PowerPC, Alpha, IA-64, x86-64
 - Minimum 64MB Hauptspeicher
 - Linux
 - Alpha, ARM, ARM26, CRIS, H8300, x86, IA-64, M68000, MIPS, PA-RISC, PowerPC, S/390, SuperH, SPARC, VAX, v850, x86-64
 - Sehr kleine Kerne durch dynamisch ladbare Kernelmodule
 - Minimum 4MB Hauptspeicher
- Windows
 - Kernel exportiert etwa 250 System Calls (ntdll.dll)
 - Layered Windows/POSIX subsystems
 - Große Windows API (17 500 functions on top of native APIs)
- Linux
 - Kernel exportiert etwa 200 System Calls
 - Layered BSD, Unix Sys V, POSIX shared system libraries
 - Kompakte APIs (1742 functions in Single Unix Specification Version 3; not including X Window APIs)

Volumen, Qualität

(Quellen: Wikipedia u.ä.)

- Windows Betriebssysteme
 - WinNT: 3-8 M Lines Of Code (LOC)
 - Win2000: >29 MLOC
 - WinXP: 40 MLOC
 - Vista: 50 MLOC
- Linux/Unix OS
 - Minix (1.1-1.2) [kernel]: 30-60 [15-30] KLOC
 - RedHat Linux (6.2-7.1): 17-30 MLOC
 - Debian (2.2-4.0): 55-283 MLOC
 - OpenSolaris: 9.7 MLOC
 - MacOS X 10.4: 86 MLOC
 - Linux Kernel 2.6: 5.2 MLOC

Zur Bewertung der **Qualität**:

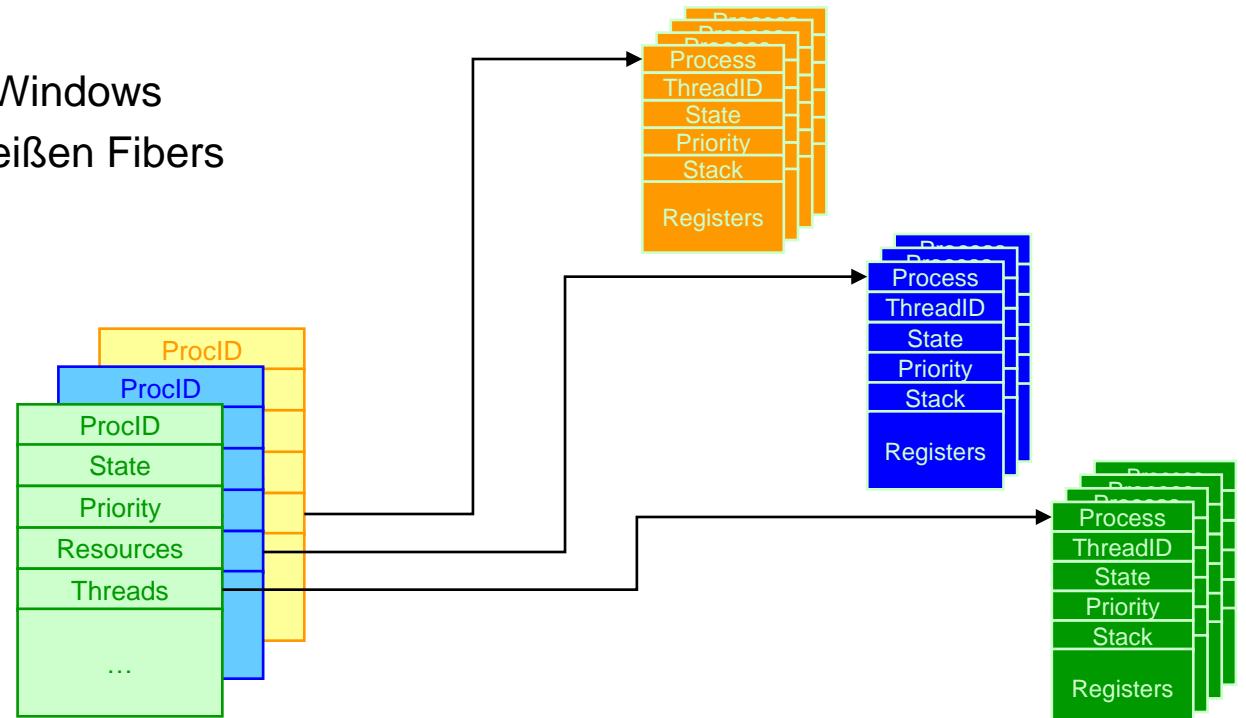
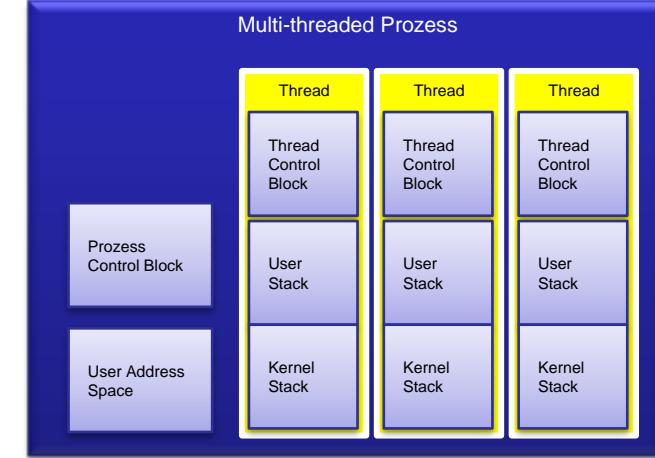
„Arguments regarding the efficiency of open source products and development processes often employ external quality attributes, anecdotal evidence or even plain hand-waving ... The main contribution of this research is the finding that there are no significant across-the board code quality differences between the four large working systems...“

(aus „A Tale of Four Kernels, Diomidis Spinellis, Proceedings of the 30th international Conference on Software Engineering ICSE '08. ACM 2008)

Windows

Prozesse und Threads

- Prozesse beinhalten
 - Adressraum,
 - Tabelle der handles (files etc.) und
 - Liste von Threads, die mindestens ein Element beinhaltet
- Threads
 - Scheduling-Einheit in Windows
 - User-Level Threads heißen Fibers

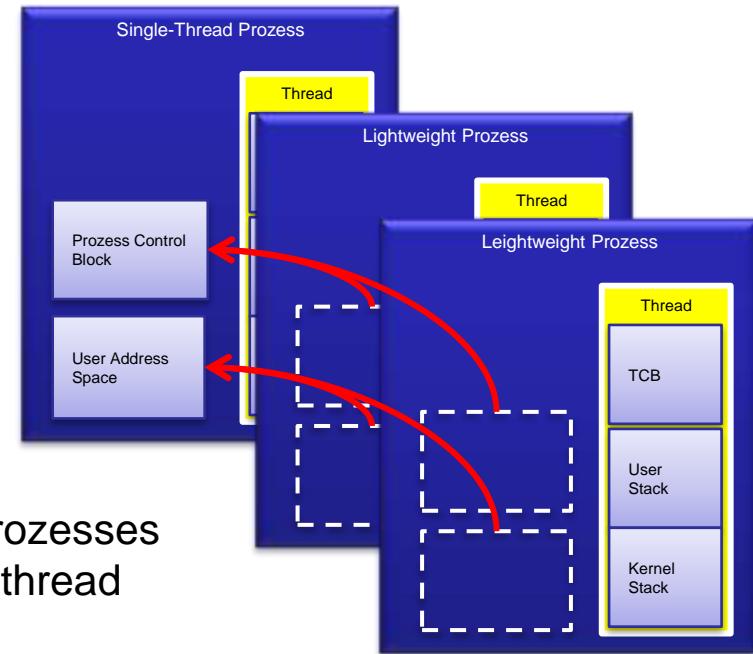


Quelle: Mark E. Russinovich, David Solomon, Microsoft Windows Internals, Fourth Edition, Microsoft Press 2005

Linux

Prozesse (Tasks) und Threads

- Prozessgenerierung durch Aufspaltung „fork“
 - Logische Kopie des erzeugenden Prozesses
 - Separate Kopien von Stack und Heap für erzeugenden und abgeleiteten Prozess (*Parent Child Relationship*)
- Frühere Unix-Kerne:
 - Elternprozess hat keinen Zugriff auf Daten des Kindprozesses
 - Unterstützung von multithreaded Applications durch pthread (POSIX thread) library (User-level Prozesse)
- Threads gibt es eigentlich nicht
- Heute, in Linux: Lightweight-Prozesse teilen sich Ressourcen wie Speicher etc. Mit jedem Thread wird in Linux ein Lightweight-Process assoziiert.
- POSIX Threads in Linux werden unterstützt durch einen Kern, der Thread-Gruppen unterstützt
 - Thread-Gruppe: Menge von Leichtgewichtsprozessen, die eine multithreaded application implementieren.
 - „Verstehen“ system calls wie getpid(), kill() und _exit().
- Scheduling Einheit: Prozess (=Task)



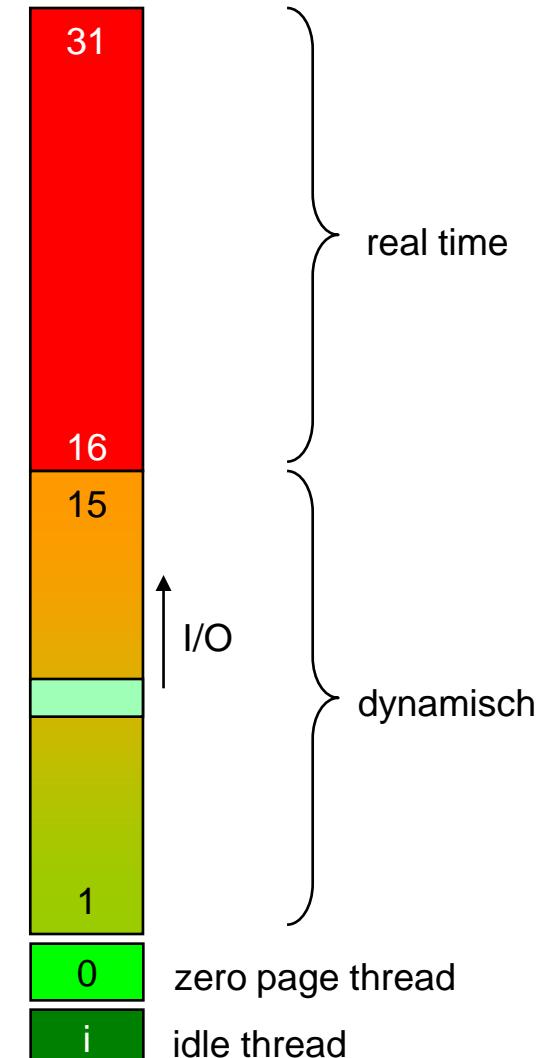
Windows Scheduling

Prioritäts-basiertes, preemptives Scheduling System

- Höchst-priorisierter Thread läuft
- Jeder Thread läuft für ein gewisses Zeitquantum
- Threads mit der gleichen Priorität werden mit einem Round-Robin Verfahren eingesetzt
- Es gibt keinen separaten Scheduler. Scheduling Code ist über den Kern verteilt
- Dispatcher Routinen werden z.B. durch folgende Ereignisse angestoßen
 - Ein Thread wird neu in die Ready-Liste eingetragen
 - Ein Thread verlässt den Zustand „laufend“ (quantum läuft ab/ Warten auf Ereignis)
 - Die Priorität eines Threads wird verändert (z.B. system call)
- Es gibt 32 Prioritätsstufen
 - 16-31: Realtime
 - 1-15: dynamisch
 - 0: Idle
 - eingestellt durch Windows API Funktion SetThreadPriority() und SetProcessPriority()
 - hängt von Prozess- und Thread Priorität ab (s.u.)

Windows Scheduling

- Aus der Sicht der Windows API
 - Prozesse bekommen eine Prioritätsklasse bei der Erstellung
 - Idle, Normal, High, Realtime
 - Threads haben relative Priorität innerhalb der Klasse
 - Idle, Lowest, Below_Normal, Normal, Above_Normal, Highest, and Time_Critical
- Aus der Sicht des Kerns
 - Threads werden scheduliert, nicht Prozesse
 - Die Prozessprioritätsklasse wird nicht zu Scheduling-Entscheidungen herangezogen
 - Zwei Klassen
 - Real-time, statisch, mit Prioritäten 16-31
 - Dynamisch, mit Prioritäten 1-15
 - Höhere Prioritäten werden bevorzugt
 - Prioritäten dynamischer Threads werden bei I/O Unterbrechung erhöht („boost“)
 - Quantum stretching für optimiertes Antwortverhalten
 - Prioritäten fallen nie unter die Basispriorität

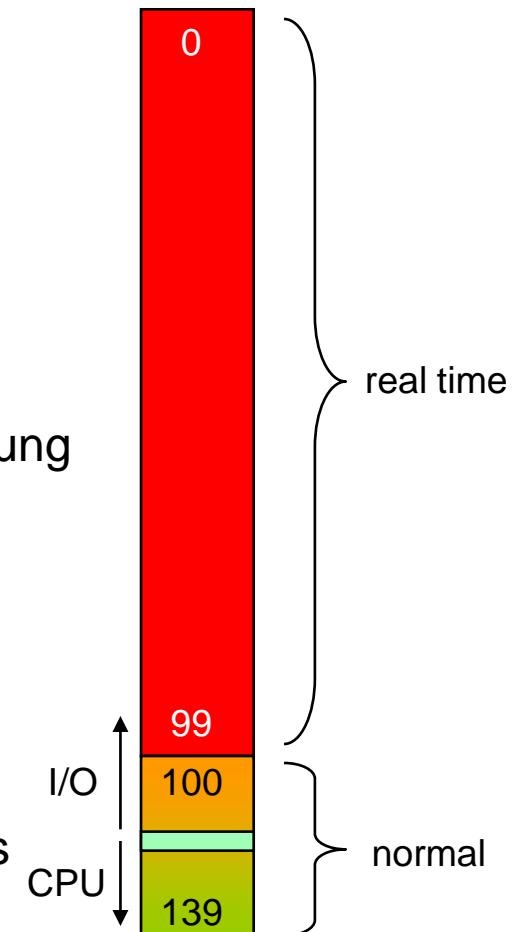


Windows Real-Time Scheduling

- Windows unterstützt statische round-robin Strategie für Threads der Prioritäten 16-31
 - Threads laufen maximal ein Quantum
 - Quantum wird zurückgesetzt falls Thread preempted wird
 - Real-Time Prioritäten werden nicht erhöht
- RT Threads können wichtige System-Services verlangsamen
 - z.B. CSRSS.EXE
- System Calls können Priority Inversion auslösen

Linux Scheduling

- 3 Prioritätsklassen
 - Normal: Priorität 100-139
 - Fixed Round Robin: Prioritäten 0-99
 - Fixed Fifo: Prioritäten 0-99
- Kleinere Prioritäts-Werte werden bevorzugt
 - Werte normaler Threads werden erhöht während der Nutzung von Prozessorzeit
 - Werte interaktiver Threads werden verkleinert („boost“)
- Normale Prozesse
 - Neue Prozesse erben immer die Priorität ihrer Erzeuger
 - nice() und setpriority() Funktionalität
 - Priorität bestimmt das *base time quantum* eines Prozesses
 - Unterscheidung in *active* und *expired processes*

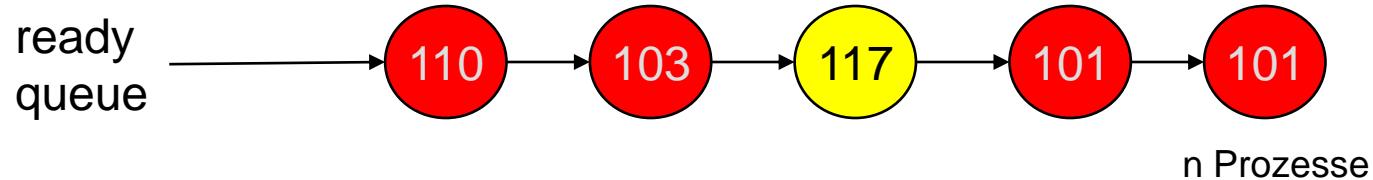


Linux Real-Time Scheduling

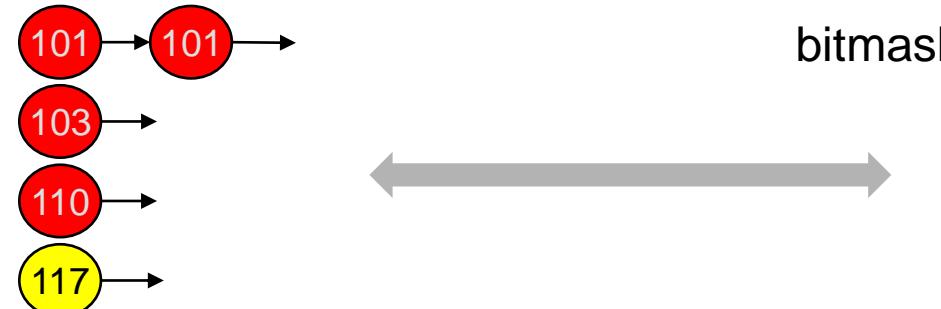
- Unterstützt sind zwei Klassen Round Robin und FIFO
 - Klasse wird mit sched_setscheduler() Systemaufruf ausgewählt
 - Statische Prioritäten 1 bis 99
 - Strikte Ausführung nach Priorität
 - RoundRobin
 - Thread läuft bis zum Ablauf eines Zeitquants
 - wechselt dann zum nächsten Thread der gleichen Priorität
 - FIFO
 - Thread muss zur Beendigung yield() aufrufen
- Real Time Prozesse
 - Prozesse werden durch andere Prozesse ersetzt, wenn
 - sie durch Prozess höherer Priorität preemptet werden
 - Prozess blockiert, endet oder *gekilled wird*
 - Prozess Kontrolle abgibt (yield)
 - Prozess RR scheduliert ist und Zeitquantum abläuft
- Lange laufende System Calls können Priority Inversion auslösen
 - wie in Windows, nicht in rtLinux

Linux Scheduling

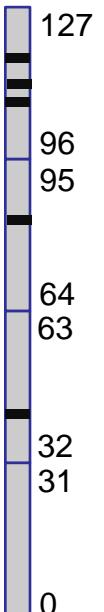
- Scheduler von Linux 2.4
 - $O(n)$ Algorithmus, da alle Prozesse in einer Queue



- Linux 2.6
 - $O(1)$ Algorithmus mit mehreren Queues (pro CPU)



- Windows: $O(1)$ Scheduler basierend auf vorsortierten Queues



Multi-Prozessor Unterstützung

Windows

- SMP (symmetric multiprocessing) unterstützt
 - bis zu 32 Prozessoren in 32-bit Version
 - bis zu 64 Prozessoren in der 64-bit Version
 - *Lizenzlimitierung*
 - Interrupts auf allen CPUs
- Unterstützung von NUMA (Non-Uniform Memory Access) Systemen
 - Affinität der Prozesse zu einem Prozessor
- Unterstützung von Hyperthreading
 - Scheduler favorisiert unbeschäftigte Prozessor
 - Logische CPUs tragen nicht zur Lizenzlimitierung bei

Linux

- SMP (symmetric multiprocessing) unterstützt
 - kein Limit für Anzahl Prozessoren (Konstante im Kern)
 - Interrupts auf alle CPUs
- Unterstützung von NUMA (Non-Uniform Memory Access) Systemen
 - Affinität der Prozesse zu einem Prozessor
- Unterstützung von Hyperthreading
 - Scheduler favorisiert unbeschäftigte Prozessor

Virtual Memory Management

Windows

- Speicheraufteilung in 2GB User Memory und 2GB für den Kernel-Mode (variabel bis 3GB/1GB)



Linux

- Speicheraufteilung in 1GB/3GB bis zu 3GB/1GB



- Demand-paged Virtual Memory, Unterstützung für
 - Copy-on-write
 - Shared memory
 - Memory mapped files

- In Version 2.6 gibt es eine Option, in der der Kern einen eigenen Adressraum zugewiesen bekommt.

- Demand-paged virtual memory, Unterstützung für
 - Copy-on-write
 - Shared memory
 - Memory mapped files

In den 64bit Varianten ist die Aufteilung anders, außerdem werden dort 3-4 stufige Page-Tables verwendet

Virtual Memory Management: Paging

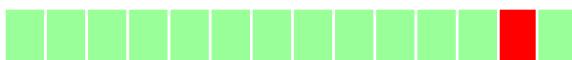
Windows

- *Lokales Working Set (pro Prozess)*
 - LRU Working Set Bestimmung mit dem Aging Verfahren
- Kein Swapping (nur Paging)

Process Memory



LRU



LRU



LRU

Linux

- *Globales Management des LRU Working Set Bestimmung mit Aging Verfahren*
- Kein Swapping
 - Dienst, der die Working Set Bestimmung leistet heißt trotzdem “swap daemon”

Global Memory



LRU

Windows Synchronisation

- *Interrupt-Masken* zum Schutz globaler Ressourcen gegen gleichzeitigen Zugriff auf Uniprozessor Systemen
- Auf Multiprozessor Systemen werden *Spinlocks* eingesetzt
- Stellt *Dispatcher Objekte* für Mutexe und Semaphoren zur Verfügung
- Dispatcher Objekte können auch *Events* generieren. Ein Event entspricht in etwa einer Bedingungsvariable.

Linux Synchronisation

- Ausschalten aller *Interrupts* für den synchronisierten Zugriff auf globale Daten auf Uniprozessor Systemen.
- Auf Multiprozessor Systemen werden *Spinlocks* eingesetzt
- *Semaphoren* und *Readers-Writers Locks* für Synchronisation in längeren Codesequenzen
- POSIX Synchronisation für die Unterstützung von Multitasking, Multithreading (incl. Echtzeit Threads) und Multiprocessing.

Leichtgewichtige Synchronisierung

- Linux 2.6: Futexes (Fast Mutual Exclusion)
 - Sprung zum Kernel-Mode nur notwendig, wenn mehrere Prozesse warten
- Windows: CriticalSections (für Threads)
 - gleiches Verhalten
- aber: Futexe gehen einen Schritt weiter:
 - funktionieren auch zwischen Prozessen
 - behandeln Priority Inversion via *Priority Inheritance*