



Sommer 2018

Betriebssysteme und Rechnerarchitektur

Prof. Dr. Wolfgang Weitz





► Prüfungsmodalitäten

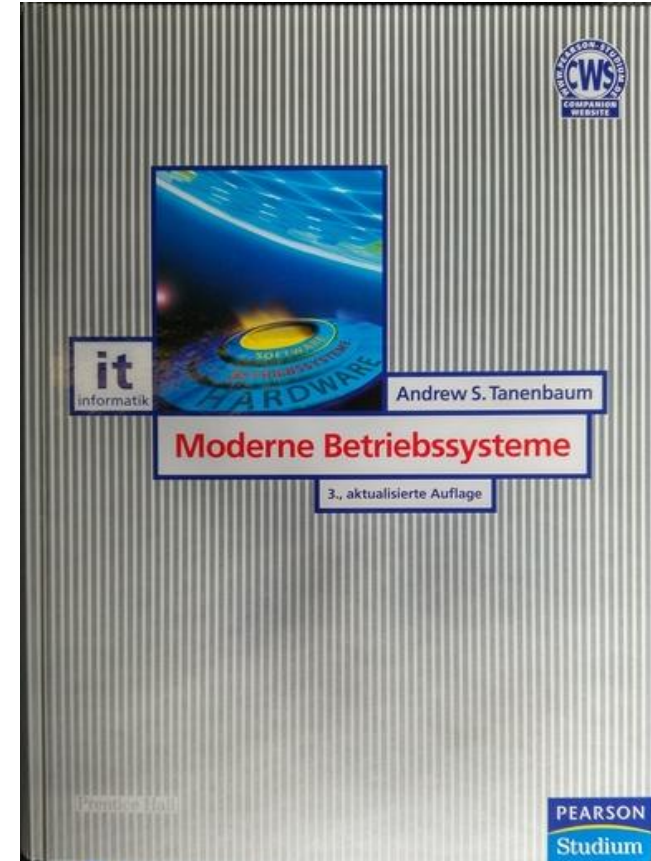
- Praktikum
(unbenotete SL, Projekt mit Zwischenabgaben)
- Klausur am Semesterende (PL, 100% Modulnote)

► Praktikum:

- Linux-Rechner (PC-Pool)
- Systemnahe Programmierung (in ANSI-C)

Andrew S. Tanenbaum,
„Moderne Betriebssysteme“,
Pearson Studium

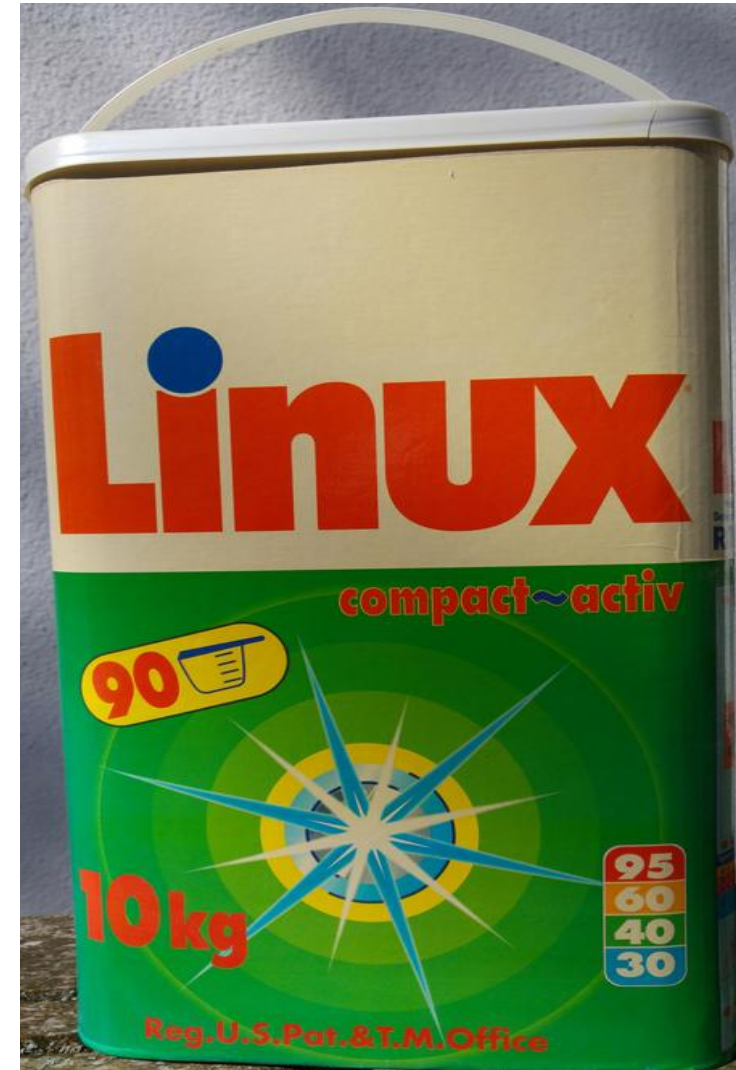
Grundlage der Veranstaltung,
3. Auflage ok,
reichlich in Bibliothek vorhanden



► Betriebssysteme

- Aufgaben eines Betriebssystems
- Wichtige Konzepte und Verfahren
- Schwerpunkt: UNIX-Familie
- systemnahe Programmierung in C

► Rechnerarchitektur





Rechnersystem aus Hardwaresicht

5

Hardwarekomponenten eines Rechnersystems

CPU(s)

Anschlüsse: PCI, USB, ...

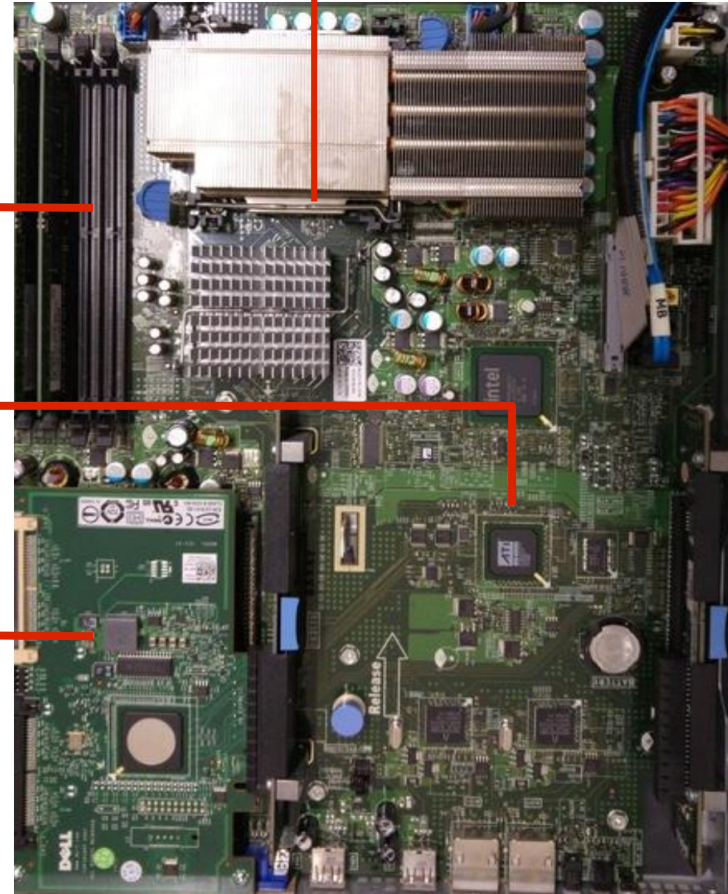
Hauptspeicher (RAM)

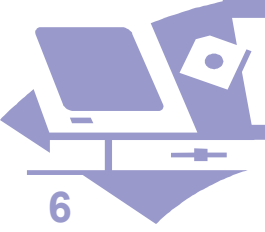
Netzwerkcontroller

Display/Grafik-Controller

SSD / Festplatte und
zugehörige Controller

...





Rechnersystem aus Nutzersicht

► Bereitstellung von Diensten wie z.B.

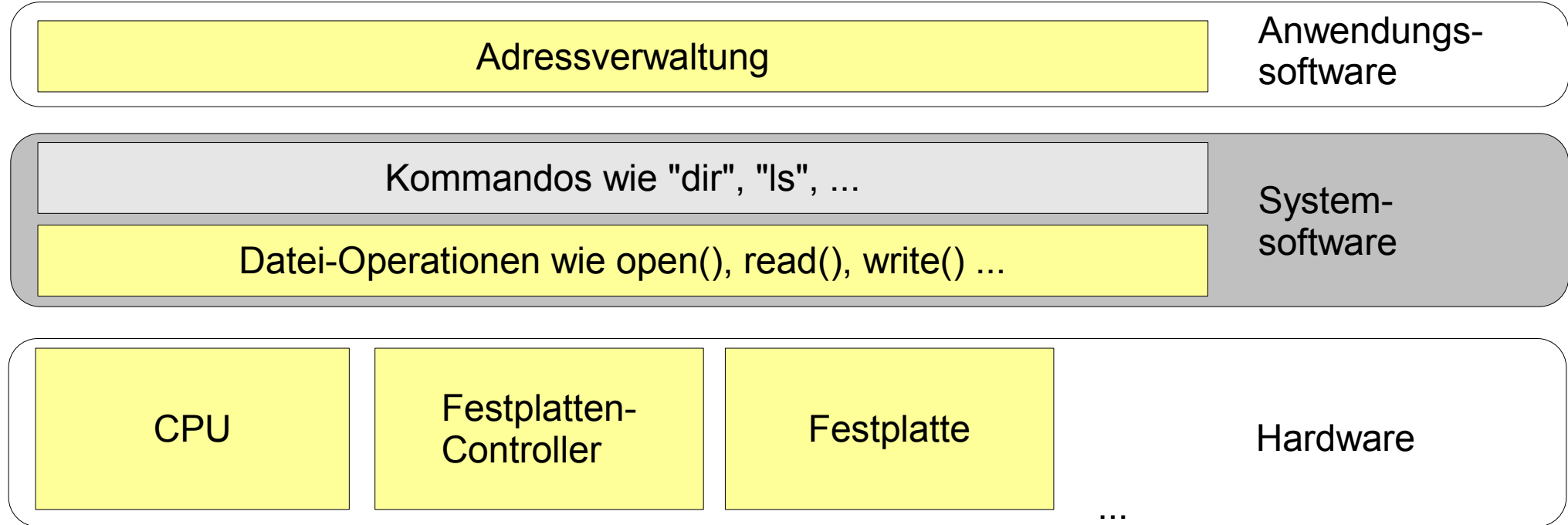
- **Dateiverwaltung**
(Ordner, Suchmöglichkeiten, Zugriffsschutz, ...)
- **Ein- und Ausgabemöglichkeiten**
- Speicherverwaltung
- Threads / Prozesse / Inter-Prozess-Kommunikation
- evtl. **Mehrbenutzerfähigkeit**
- **Netzwerkzugang** (Verbindungsaufbau, ...)

► **Programmierungsumgebung**
(Compiler, Debugger, Bibliotheken,...)

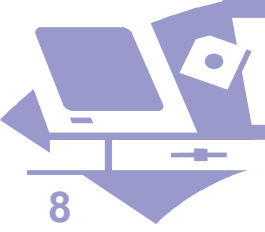
► ...



Beispiel: Festplattenzugriff



Das Betriebssystem stellt oberen Schichten eine "virtuelle Maschine" mit **höherem Abstraktionsniveau** bereit (z.B. *verallgemeinerte* Funktionen zur Dateiverarbeitung statt Ansteuerung der Hardware-Register eines *konkreten* Festplatten-Controllers und Lesen/Schreiben einzelner Sektoren).



Schichtenmodell

Allgemeines Schichtenmodell eines Rechnersystems:

Web-Browser, Buchungssysteme, Malprogramme, ...

Anwendungs-
software

Compiler, Kommandointerpreter (shell), ...

Betriebssystem

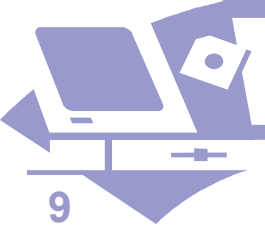
System-
software

Maschinensprache

Mikroarchitektur

Physische Geräte

Hardware



„Nah an der Maschine“

9

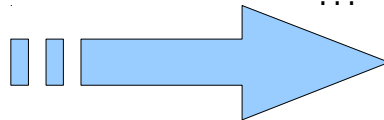
```
int main(void) {
    const char *msg = "Hello, world!\n";
    const char *p = msg;
    int z, laenge;

    /* Stringlaenge finden */
    while (*p++);
    laenge = p - msg;

    /* String 17x ausgeben */
    for (z=0; z < 17; z++) {
        /* UNIX-Systemfunktion */
        write(0, msg, laenge);
    }
    return 0;
}
```

```
...
0000000000400506 <main>:
400506: 55                push    rbp
400507: 48 89 e5          mov     rbp, rsp
40050a: 48 83 ec 20       sub     rsp, 0x20
40050e: 48 c7 45 e8 04 06 40 mov     QWORD PTR [rbp-0x18], 0x400604
400515: 00
400516: 48 8b 45 e8       mov     rax, QWORD PTR [rbp-0x18]
40051a: 48 89 45 f8       mov     QWORD PTR [rbp-0x8], rax
40051e: 90                nop
40051f: 48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
400523: 48 8d 50 01       lea     rdx, [rax+0x1]
400527: 48 89 55 f8       mov     QWORD PTR [rbp-0x8], rdx
40052b: 0f b6 00          movzx   eax, BYTE PTR [rax]
40052e: 84 c0             test    al, al
400530: 75 ed             jne     40051f <main+0x19>
400532: 48 8b 55 f8       mov     rdx, QWORD PTR [rbp-0x8]
400536: 48 8b 45 e8       mov     rax, QWORD PTR [rbp-0x18]
40053a: 48 29 c2          sub     rdx, rax
40053d: 48 89 d0          mov     rax, rdx
400540: 89 45 e4          mov     DWORD PTR [rbp-0x1c], eax
400543: c7 45 f4 00 00 00 00 mov     DWORD PTR [rbp-0xc], 0x0
40054a: eb 1b             jmp     400567 <main+0x61>
40054c: 8b 45 e4          mov     eax, DWORD PTR [rbp-0x1c]
40054f: 48 63 d0          movsxd  rdx, eax
400552: 48 8b 45 e8       mov     rax, QWORD PTR [rbp-0x18]
400556: 48 89 c6          mov     rsi, rax
400559: bf 00 00 00 00    mov     edi, 0x0
40055e: e8 7d fe ff ff    call    4003e0 <write@plt>
400563: 83 45 f4 01       add     DWORD PTR [rbp-0xc], 0x1
400567: 83 7d f4 10       cmp     DWORD PTR [rbp-0xc], 0x10
40056b: 7e df             jle     40054c <main+0x46>
40056d: b8 00 00 00 00    mov     eax, 0x0
400572: c9                leave
400573: c3                ret
...
```

C - prozessorunabhängig



Intel x86 Maschinencode (PCs, ...)



„Nah an der Maschine“ (andere Maschine)

10

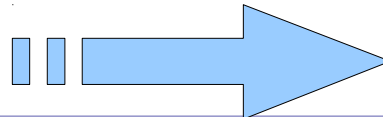
```
int main(void) {
    const char *msg = "Hello, world!\n";
    const char *p = msg;
    int z, laenge;

    /* Stringlaenge finden */
    while (*p++);
    laenge = p - msg;

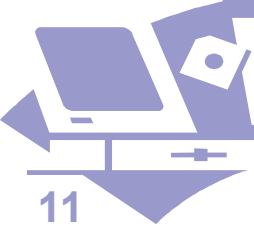
    /* String 17x ausgeben */
    for (z=0; z < 17; z++) {
        /* UNIX-Systemfunktion */
        write(0, msg, laenge);
    }
    return 0;
}
```

```
0001041c <main>:
1041c: e92d4800    push    {fp, lr}
10420: e28db004    add     fp, sp, #4
10424: e24dd010    sub     sp, sp, #16
10428: e59f307c    ldr     r3, [pc, #124] ; 104ac <main+0x90>
1042c: e50b3010    str     r3, [fp, #-16]
10430: e51b3010    ldr     r3, [fp, #-16]
10434: e50b3008    str     r3, [fp, #-8]
10438: e1a00000    nop
1043c: e51b3008    ldr     r3, [fp, #-8] ; (mov r0, r0)
10440: e2832001    add     r2, r3, #1
10444: e50b2008    str     r2, [fp, #-8]
10448: e5d33000    ldrb    r3, [r3]
1044c: e3530000    cmp     r3, #0
10450: 1afffff9    bne     1043c <main+0x20>
10454: e51b2008    ldr     r2, [fp, #-8]
10458: e51b3010    ldr     r3, [fp, #-16]
1045c: e0633002    rsb     r3, r3, r2
10460: e50b3014    str     r3, [fp, #-20] ; 0xffffffffec
10464: e3a03000    mov     r3, #0
10468: e50b300c    str     r3, [fp, #-12]
1046c: ea000007    b       10490 <main+0x74>
10470: e51b3014    ldr     r3, [fp, #-20] ; 0xffffffffec
10474: e3a00000    mov     r0, #0
10478: e51b1010    ldr     r1, [fp, #-16]
1047c: e1a02003    mov     r2, r3
10480: ebffff95    bl      102dc <write@plt>
10484: e51b300c    ldr     r3, [fp, #-12]
10488: e2833001    add     r3, r3, #1
1048c: e50b300c    str     r3, [fp, #-12]
10490: e51b300c    ldr     r3, [fp, #-12]
10494: e3530010    cmp     r3, #16
10498: dafffff4    ble     10470 <main+0x54>
1049c: e3a03000    mov     r3, #0
104a0: e1a00003    mov     r0, r3
104a4: e24bd004    sub     sp, fp, #4
104a8: e8bd8800    pop     {fp, pc}
```

C - prozessorunabhängig



ARM Maschinencode (Raspberry Pi, Handys ...)



Erkenntnis: C ist komfortabel :-)

```
int main(void) {
    const char *msg = "Hello, world!\n";
    const char *p = msg;
    int z, laenge;

    /* Stringlaenge finden */
    while (*p++);
    laenge = p - msg;

    /* String 17x ausgeben */
    for (z=0; z < 17; z++) {
        /* UNIX-Systemfunktion */
        write(0, msg, laenge);
    }
    return 0;
}
```

C

```
...
0000000000400506 <main>:
400506: 55                push    rbp
400507: 48 89 e5          mov     rbp, rsp
40050a: 48 83 ec 20       sub     rsp, 0x20
40050e: 48 c7 45 e8 04 06 40 mov     QWORD PTR [rbp-0x18], 0x400604
400515: 00
400516: 48 8b 45 e8       mov     rax, QWORD PTR [rbp-0x18]
40051a: 48 89 45 f8       mov     QWORD PTR [rbp-0x8], rax
40051e: 90               nop
40051f: 48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
400523: 48 8d 50 01       lea     rdx, [rax+0x1]
400527: 48 89 55 f8       mov     QWORD PTR [rbp-0x8], rdx
40052b: 0f b6 00         movzx   eax, BYTE PTR [rax]
40052e: 84 c0            test    al, al
400530: 75 ed            jne     40051f <main+0x19>
400532: 48 8b 55 f8       mov     rdx, QWORD PTR [rbp-0x8]
400536: 48 8b 45 e8       mov     rax, QWORD PTR [rbp-0x18]
40053a: 48 29 c2         sub     rdx, rax
40053d: 48 89 d0         mov     rax, rdx
400540: 89 45 e4         mov     DWORD PTR [rbp-0x1c], eax
400543: c7 45 f4 00 00 00 00 mov     DWORD PTR [rbp-0xc], 0x0
40054a: eb 1b            jmp     400567 <main+0x61>
40054c: 8b 45 e4         mov     eax, DWORD PTR [rbp-0x1c]
40054f: 48 63 d0         movsxd  rdx, eax
400552: 48 8b 45 e8       mov     rax, QWORD PTR [rbp-0x18]
400556: 48 89 c6         mov     rsi, rax
400559: bf 00 00 00 00   mov     edi, 0x0
40055e: e8 7d fe ff ff   call    4003e0 <write@plt>
400563: 83 45 f4 01      add     DWORD PTR [rbp-0xc], 0x1
400567: 83 7d f4 10      cmp     DWORD PTR [rbp-0xc], 0x10
40056b: 7e df            jle     40054c <main+0x46>
40056d: b8 00 00 00 00   mov     eax, 0x0
400572: c9              leave   rax
400573: c3              ret
...
```

Intel x86

```
00010410 <main>:
10410: e92d4800         push    {fp, lr}
10420: e28db004         add     fp, sp, #4
10424: e24dd010         sub     sp, sp, #16
10428: e59f307c         ldr     r3, [pc, #124] ; 104ac <main+0x90>
1042c: e50b3010         str     r3, [fp, #-16]
10430: e51b3010         ldr     r3, [fp, #-16]
10434: e50b3008         str     r3, [fp, #-8]
10438: e1a00000         nop
1043c: e51b3008         ldr     r3, [fp, #-8]
10440: e2832001         add     r2, r3, #1
10444: e50b2008         str     r2, [fp, #-8]
10448: e5d33000         ldrb    r3, [r3]
1044c: e3530000         cmp     r3, #0
10450: 1afffff9         bne     1043c <main+0x20>
10454: e51b2008         ldr     r2, [fp, #-8]
10458: e51b3010         ldr     r3, [fp, #-16]
1045c: e0633002         rsb     r3, r3, r2
10460: e50b3014         str     r3, [fp, #-20] ; 0xffffffff
10464: e3a03000         mov     r3, #0
10468: e50b300c         str     r3, [fp, #-12]
1046c: ea000007         b       10490 <main+0x74>
10470: e51b3014         ldr     r3, [fp, #-20] ; 0xffffffff
10474: e3a00000         mov     r0, #0
10478: e51b1010         ldr     r1, [fp, #-16]
1047c: e1a02003         mov     r2, r3
10480: ebffff95         bl      102dc <write@plt>
10484: e51b300c         ldr     r3, [fp, #-12]
10488: e2833001         add     r3, r3, #1
1048c: e50b300c         str     r3, [fp, #-12]
10490: e51b300c         ldr     r3, [fp, #-12]
10494: e3530010         cmp     r3, #16
10498: dafffff4         ble     10470 <main+0x54>
1049c: e3a03000         mov     r3, #0
104a0: e1a00003         mov     r0, r3
104a4: e24bd004         sub     sp, fp, #4
104a8: e8bd8800         pop     {fp, pc}
```

ARM

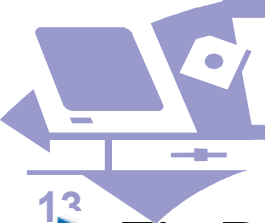


Experiment: Dasselbe mit C-Standardbib

```
int main(void) {  
    const char *msg = "Hello, world!\n";  
  
    /* Bibliotheksfunktion */  
    printf(msg);  
  
    return 0;  
}
```

► Kürzer. Kürzer?

- Voriges Beispiel, direkte Nutzung des Systemaufrufs: < 5 kBytes
- Dieses Beispiel, Stdbibliothek *statisch* dazugelinkt, also „stand-alone“ ohne Abhängigkeiten lauffähig: 747 kBytes



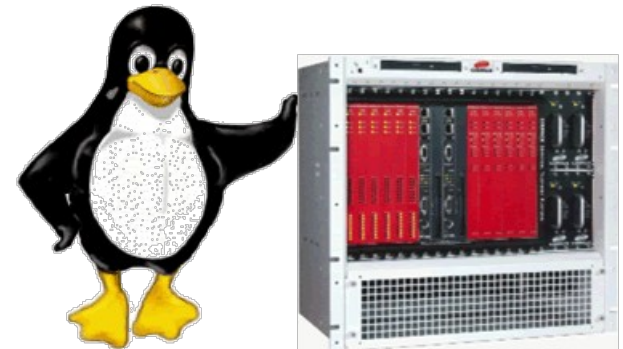
Betriebssystem

Ein Betriebssystem

- verwaltet die **Betriebsmittel** eines Rechnersystems (Effizienz, Koordination, Schutz, Abrechnung, ...)
- stellt eine **Abstraktionsschicht** oberhalb der Hardware bereit, die Hardware-Details verbirgt und
- stellt Anwendern und Programmierern dadurch eine **höhere**, leichter zu handhabende **Schnittstelle** zu den **Diensten** des Rechners bereit.

► Betriebsmittel:

- Softwarebetriebsmittel wie Dateien, Programme, ...
- Hardwarebetriebsmittel wie CPU, Speicher, ... (s.o.)





Wichtige BS-Konzepte

14

► Prozesse

- "Programme in Ausführung" mit eigenem Adressraum
- Mechanismen zur Inter-Prozess-Kommunikation

► Speicher(-hierarchie), Speichermanagement

- Verwaltung von Hauptspeicher, Plattenspeicher
- "virtueller Speicher" (Hauptspeichereinhalte auslagern)

► Dateisysteme

- Dateien, Verzeichnisse, Zusatzinformationen

► Zugriffskontrolle und Sicherheit

► Echtzeitbetrieb

- Einhalten definierter Reaktionszeiten

► Fehlertoleranz

► ...



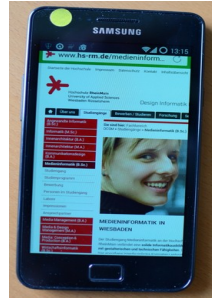
Spektrum von Betriebssystemen

- ▶ **Mainframe**(Großrechner)-Betriebssysteme
 - große Datenmengen, hohe I/O-Bandbreiten
 - Siemens BS2000, IBM OS/390, z/OS, (Linux)
- ▶ **Server**-Betriebssysteme
 - Gemeinsame Nutzung, Networking
 - UNIX (z.B. IBM AIX, HP-UX, ...),
Windows 200x Server, Linux, Mac OS X, ...
- ▶ **Arbeitsplatz**-Betriebssysteme
 - Windows 10, Mac OS X, Linux
- ▶ **Echtzeit**-Betriebssysteme
 - QNX, RTLinux, erweiterte UNIXe
- ▶ BSe für **mobile / eingebettete Systeme**
Android, iOS, ...

Beispiele (Linux inside)



Fernsehempfänger



Android (Linux)
Handy



Raspberry Pi 2



Internet-Radio

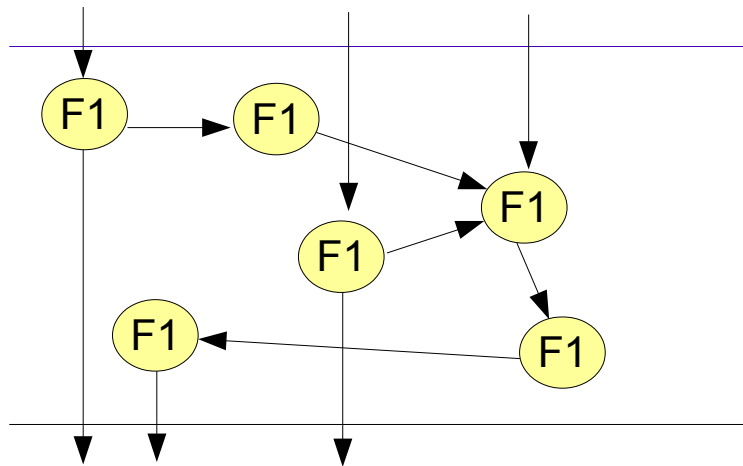


WLAN-Router



Monolithische BS-Struktur

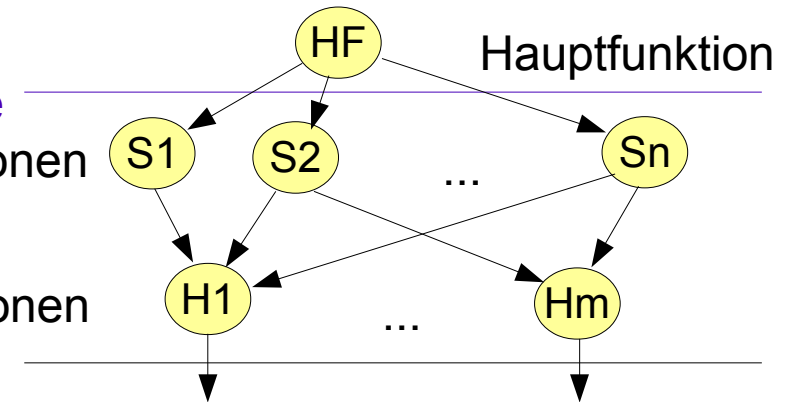
- ▶ Keine (links) oder wenig geordnete (rechts) innere Struktur; BS besteht aus sich gegenseitig aufrufenden Programmstücken
- ▶ Dienstfunktionen (Systemaufrufe S_i) stehen auf einer Stufe, nutzen Hilfsfunktionen



Systemschnittstelle

Systemfunktionen

Hilfsfunktionen



- ▶ Beispiele: UNIX, MS-DOS

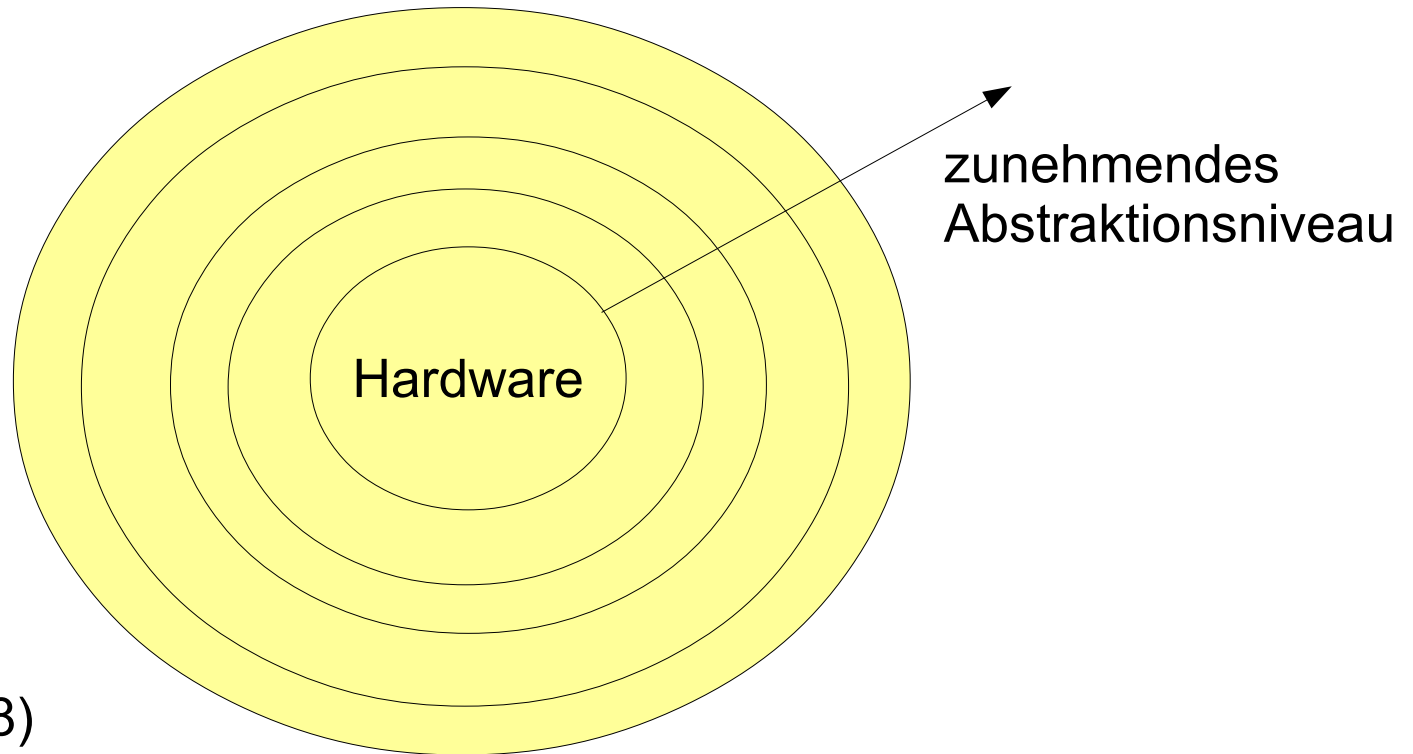


18

Hierarchische BS-Struktur

Strenges Ordnungsprinzip

Schichten / Schalenmodell (steigende Abstraktionsstufen)

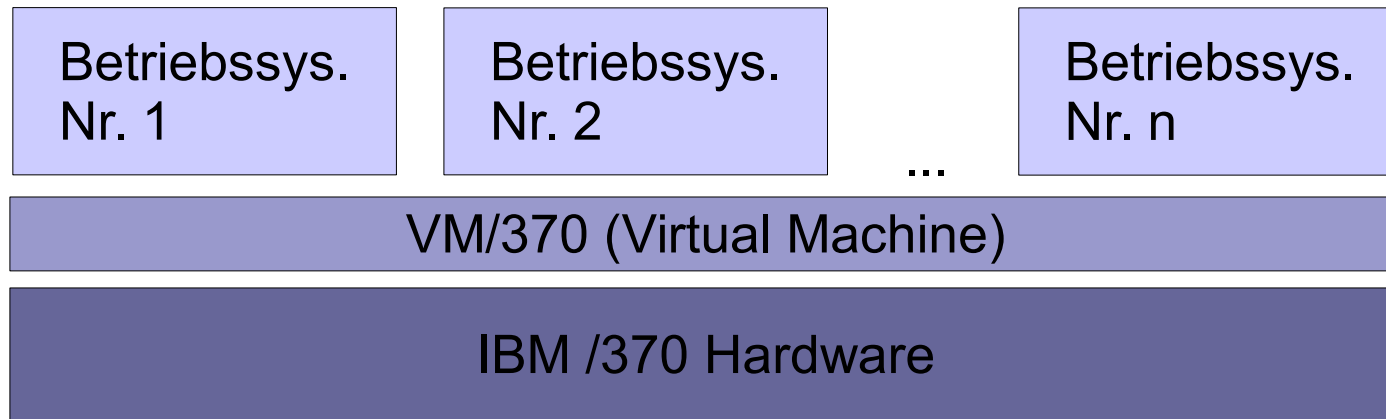


Beispiel: THE (1968)

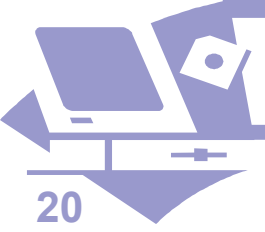


Virtuelle Maschinen

- ▶ Idee: Eine weitere Schicht stellt "virtuelle Kopie" der Hardware mehrfach den oberen Schichten zur Verfügung
- ▶ Betrieb verschiedener Betriebssysteme nebeneinander auf **derselben Hardware** möglich



Vergleiche dagegen: JVM (Java Virtual Machine):
unterschiedliche darunterliegende Hardware
einheitliche "Java-Hardware" für obere Schichten

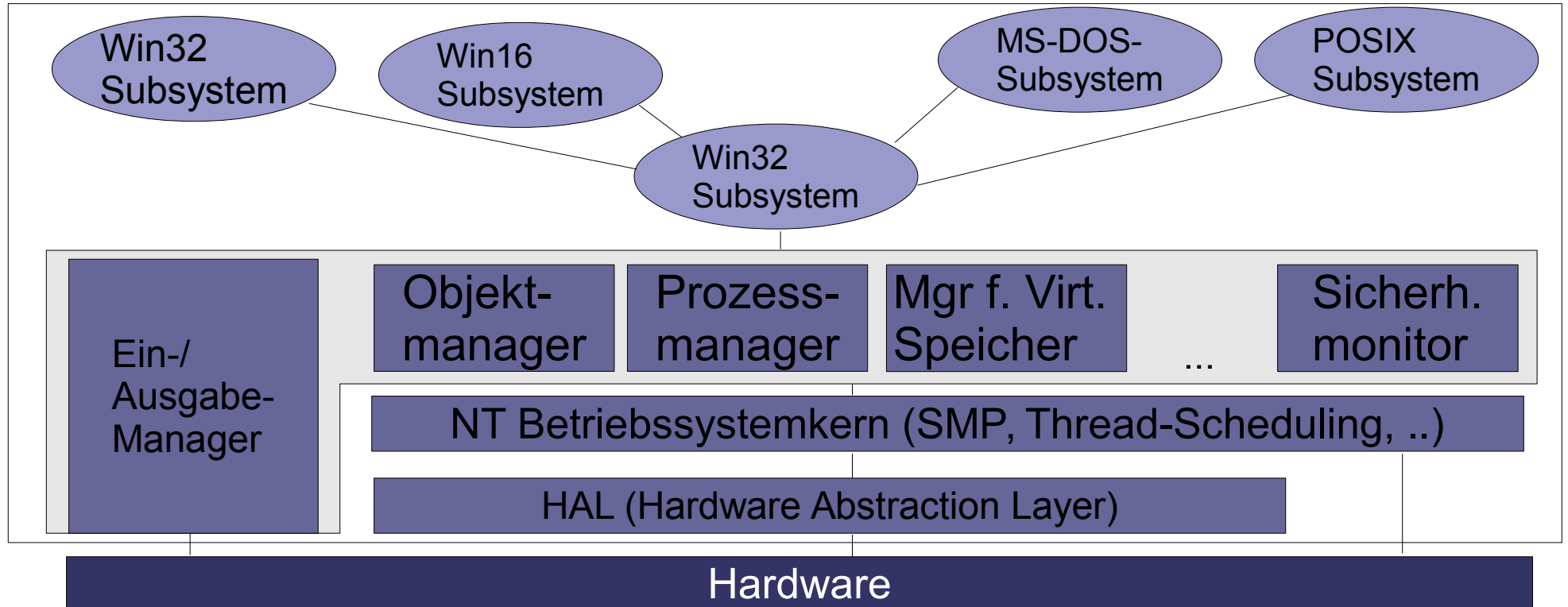


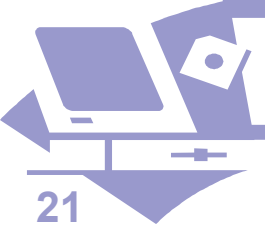
Microkernel

Bisher: Alle Funktionen im Kern → unübersichtlich

Idee: minimaler Kern, darauf aufbauende Module

Beispiele: Mach-Kernel, teilweise auch Windows NT:





Ausführungsmodi von Programmen

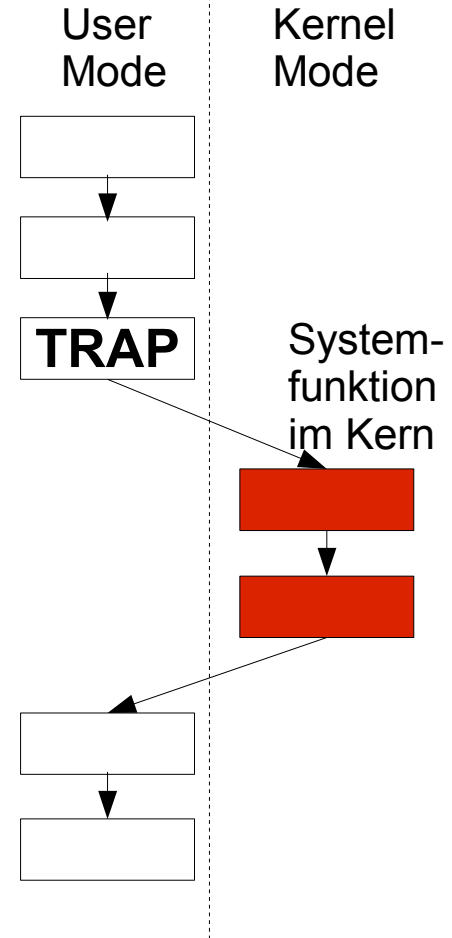
21

► User Mode:

- Programmcode des Benutzers,
- beschränkter Zugriff auf Betriebssystem-Daten,
- kann bestimmte Maschinenbefehle nicht ausführen

► Kernel Mode:

- Funktionen des Betriebssystemkerns
- uneingeschränkte Privilegien
- Aufruf aus User Mode nur über bestimmte Maschineninstruktionen (trap), in der Regel über eine Bibliotheksfunktion der Systemsoftware



Kleine UNIX Historie

22

- ▶ **1965** - AT&T Bell Labs, General Electric und MIT beginnen mit Entwicklung des Betriebssystems **MULTICS** (Multiplexed Information and Computing System); Ziel: einige hundert gleichzeitiger Nutzer bedienen; gute Ideen, wenig Erfolg
- ▶ **1969** - Ken Thompson entwickelt UNICS (AT&T Bell Labs)
- ▶ Thompson/Ritchie entwickeln UNIX für Minirechner PDP-11 (Einfachheit; Flexibilität; durchgängige, elegante Konzepte)
- ▶ **1973** - UNIX von "B" nach "C" übertragen (portiert)
- ▶ **1974** - "das" Papier zu UNIX in den C.ACM veröffentlicht
- ▶ Zunächst stark im akademischen Bereich (Quelltexte verfügbar)
- ▶ Abspaltung des "Berkeley UNIX" ("BSD-UNIX")
- ▶ **1991** - Linus Torvalds beginnt Arbeit an Linux
- ▶ ab **1998** Versuch der Vereinheitlichung von BSD- und AT&T System V Systemschnittstelle: POSIX-Spezifikation des IEEE

By Unknown - <http://www.catb.org/~esr/jargon/html/U/Unix.html>, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=31308>



Thompson

Ritchie

1972



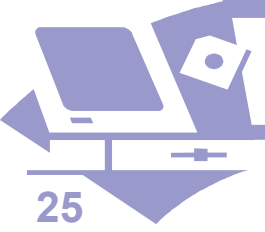
Dennis Ritchie, Ken Thompson und eine PDP-11



Eigenschaften von UNIX

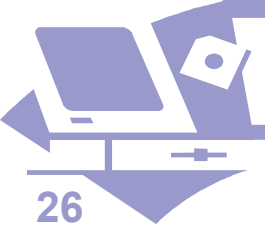
24

- ▶ Mehrbenutzer- und Mehrprogrammbetrieb (*multi-user / multi-tasking*)
- ▶ Hierarchisches Dateisystem
 - eine Wurzel ("/")
 - Dateibaum kann *mehrere physische* Geräte umfassen
- ▶ Hohe Übertragbarkeit, dadurch in Varianten
- ▶ verfügbar vom „Handy bis zum Großrechner“
- ▶ größtenteils in C geschrieben
- ▶ Mächtige Kommandosprache (der "Shell"), einfache Bausteine, aber flexible Verknüpfungsmöglichkeiten

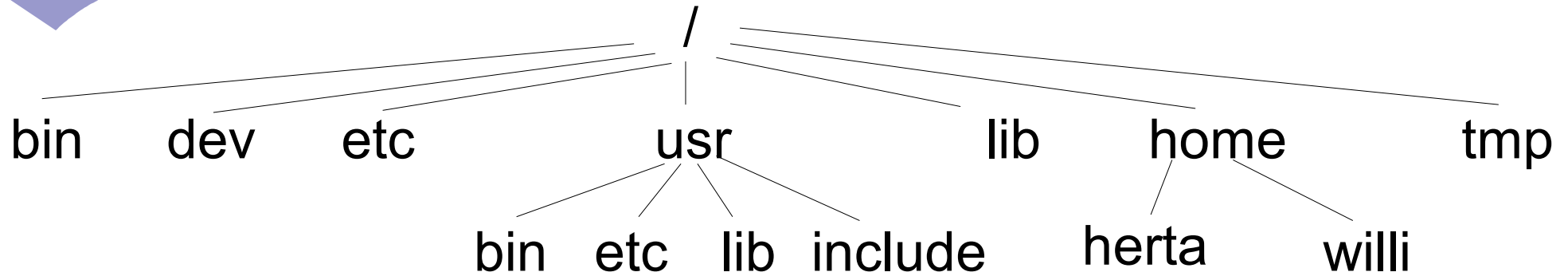


Beispiel: Kommandosprache

- ▶ Gegeben: Datei mit Städtenamen (1 je Zeile)
- ▶ Frage: Wie oft kommt "Visselhoevede" vor?
- ▶ Folgende einfachen Kommandos gibt es schon:
 - „grep“ sucht in seiner Eingabe nach Zeilen, die Suchbegriff enthalten
 - "wc" (word counter) zählt Zeilen/Wörter/Buchstaben seine Eingabe
 - mit "|" kann man Aus- und Eingabe zweier Kommandos verbinden ("Pipeline")
- ▶ Lösung:
 - `grep "Visselhoevede" datei | wc`
- ▶ "Baukastenprinzip"



Dateibaum



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



Einige wichtige Kommandos

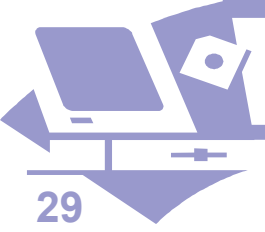
<code>ls</code>	Dateien auflisten (" <code>ls -l</code> " für mehr Infos)
<code>cd</code>	aktuelles Verzeichnis wechseln
<code>rm</code>	Datei löschen (<i>remove</i>)
<code>mkdir</code>	Verzeichnis anlegen
<code>rmdir</code>	Verzeichnis löschen (<i>remove directory</i>)
<code>ps</code>	Prozeßliste ausgeben
<code>mv</code>	Datei verschieben / umbenennen (<i>move</i>)
<code>cat</code>	Datei(en) ausgeben
<code>man</code>	Online-Manual abrufen (z.B. " <code>man ls</code> ")



Heute...

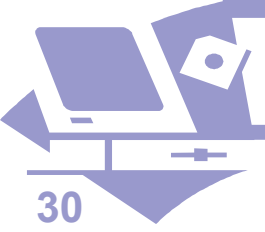
Speichermanagement





Ausgangspunkt

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



Die Speicherhierarchie

↑
schneller / teurer

Primärspeicher

CPU Register

CPU Cache

Hauptspeicher
(RAM)

direkter, wahlfreier Zugriff
durch den Prozessor,
sehr schnell



Sekundärspeicher
(z.B. Festplatte)



extern; wahlfreier
Zugriff auf Inhalt

Tertiärspeicher
(z.B. Backup-Bänder)



extern; langsam,
oft nur sequenzieller
Zugriff, hohe Kapazität



Anforderungen an Speichermanagement

31

► Schutz

- Prozesse sollten nicht unerlaubt auf fremde Speicherbereiche zugreifen können

► Gemeinsame Nutzung

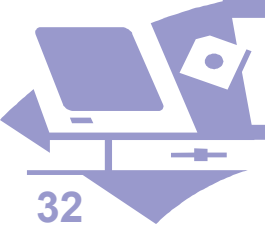
- Andererseits soll eine kontrollierte gemeinsame Nutzung von Speicherbereichen möglich sein
(z.B. 50 gleichzeitige emacs-Nutzer → Code *nicht* 50x laden)

► Relokation

- Absolute Adressbezüge im Programmcode z.B. beim Laden in einen (anderen) konkreten Speicherbereich anpassen

► Speicherorganisation

- Unterstützung von Programm-Modularisierung durch *Segmentierung*, abgestufter Schutz z.B. für Daten / Code
- Ein-/Auslagern von Speicherbereichen zwischen Hauptspeicher und Sekundärspeicher ("Festplatte")



Einprogrammbetrieb

Beispiele für Speichernutzung:

0xFF...

Benutzer-
programm

Betriebssystem
im RAM

Betriebssystem
im ROM

Benutzer-
programm

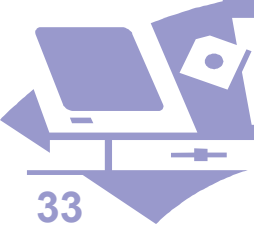
Gerätetreiber
im ROM

Benutzer-
programm

Betriebssystem
im RAM

Adresse 0

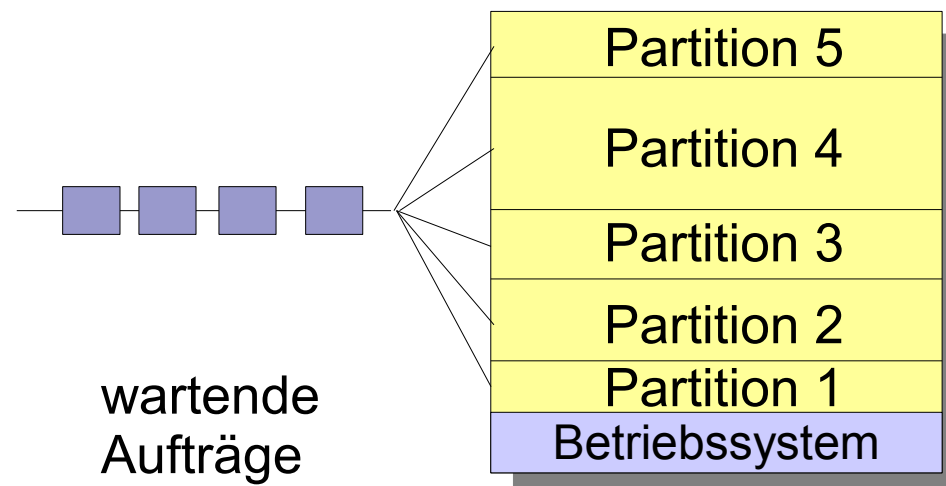
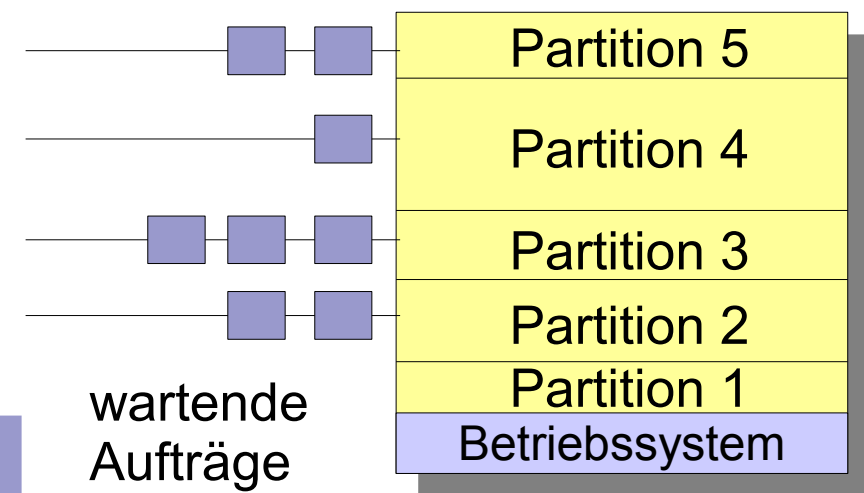
- ▶ "einfachster Fall"; Beispiel: MS-DOS
- ▶ Genügt nicht für Mehrprogrammbetrieb



Mehrprogrammbetrieb, feste Speicherpartitionen

33

- ▶ Speicher in (beim Systemstart) **fest eingerichtete Abschnitte** (Partitionen) gleicher oder verschiedener Größe aufteilen
- ▶ Anstehende **Aufträge** werden auf Partitionen verteilt (i.d.R. "Batch Betrieb": Aufträge nacheinander abarbeiten)
- ▶ Ein Programm kann
 - für eine bestimmte Partition **gebunden** sein (läuft *nur* dort)
 - oder ihm wird eine geeignete **freie** Partition zugewiesen werden (ggf. ist dabei Adressanpassung nötig, *relocation*)

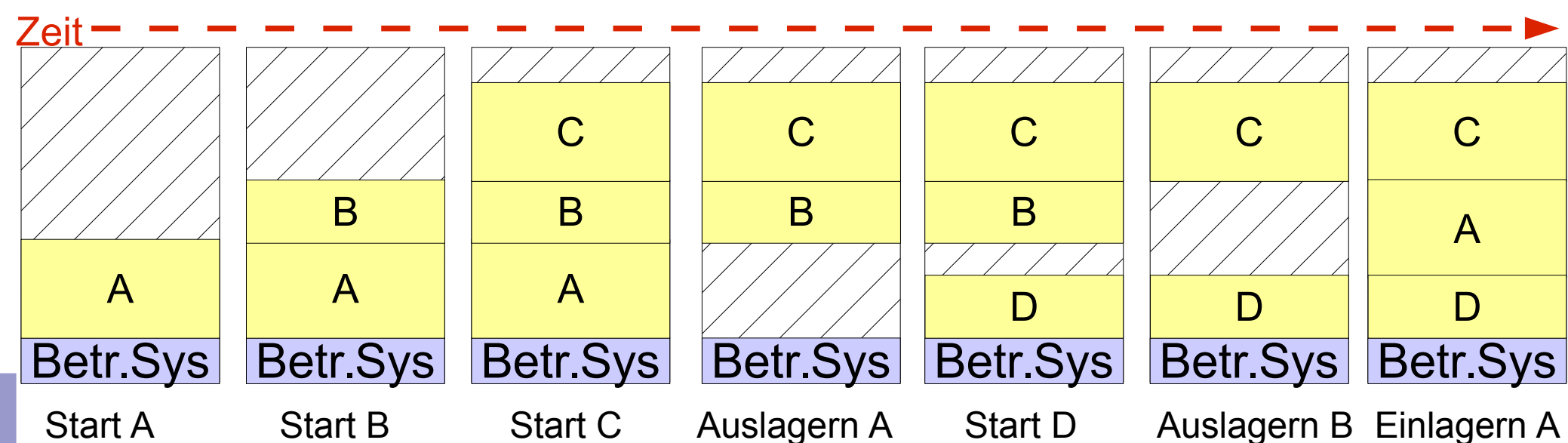


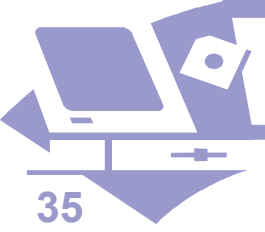


Swapping

34

- ▶ **Timesharing**-Betrieb (Rechenzeit wird auf viele "gleichzeitige" Benutzer verteilt):
Zu wenig Speicher für viele Aufträge
- ▶ **Swapping**: Gesamten Prozess auf Platte aus- bzw. einlagern
- ▶ **Idee**: Jeden Auftrag eine Zeit lang rechnen lassen, dann durch Swapping Platz für den nächsten schaffen usw.
- ▶ Zerstückelung d. freien Speichers ("externe Fragmentierung")
- ▶ "Variable Partitionen" (Anzahl, Größe, Adresse... dynamisch)

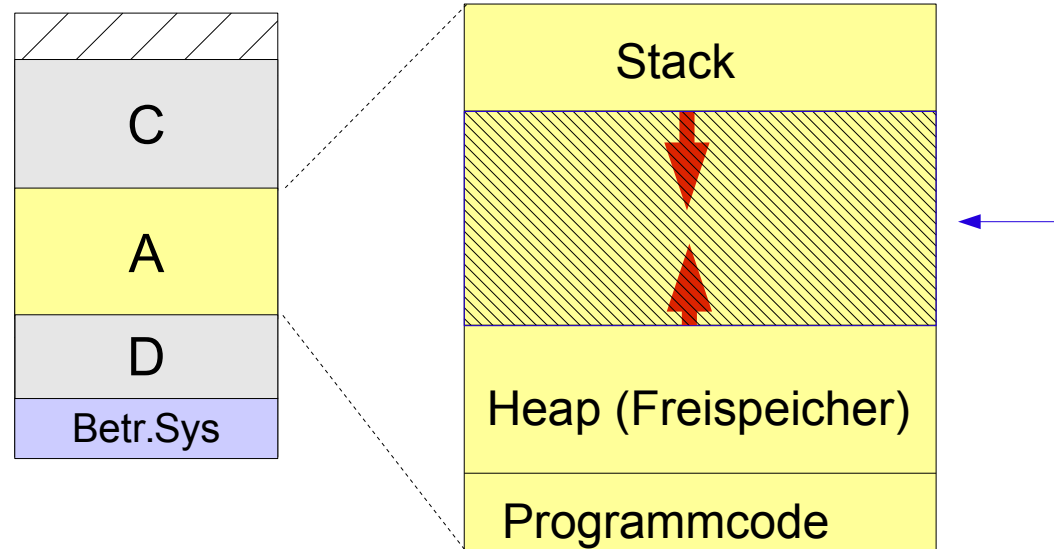


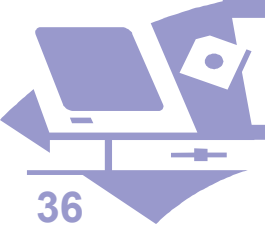


Dynamische Speicheranforderung

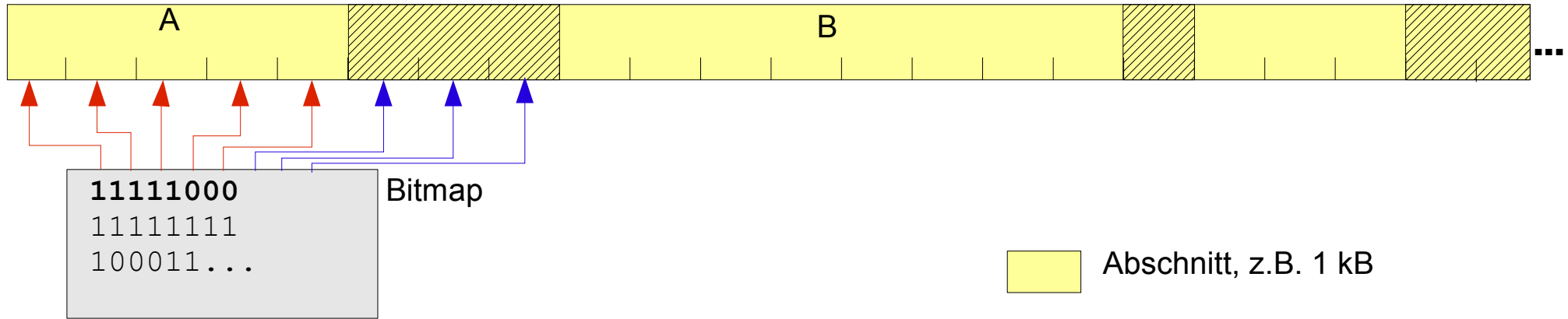
35

- Speicherbedarf **eines Prozesses variiert während Laufzeit**
 - Belegen angrenzender "Löcher" (ungenutzter Speicher)
 - ggf. andere Prozesse verschieben oder auslagern
 - vorab ausreichend **Platz zum Wachsen** mitreservieren

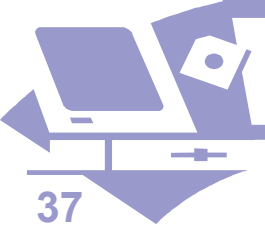




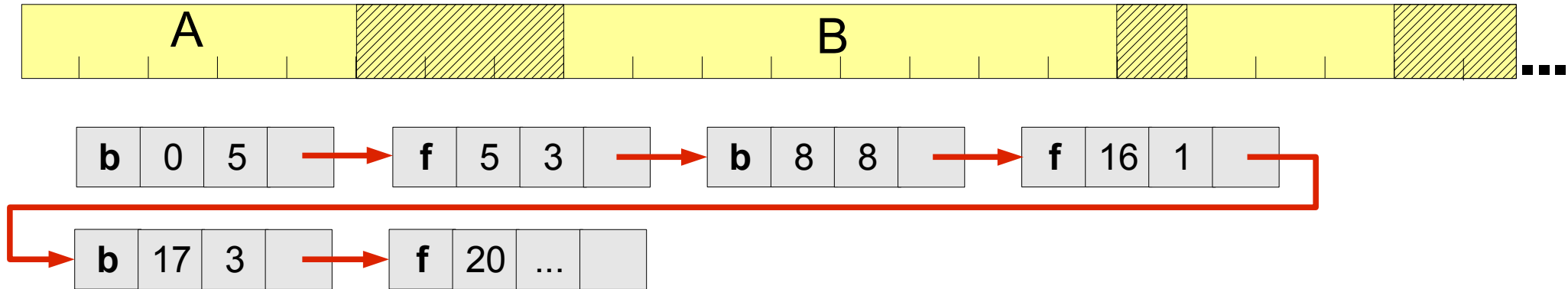
Speicherverwaltung: Bitmaps



- ▶ Speicher in **Abschnitte fester Größe** einteilen
- ▶ Jedem Abschnitt entspricht ein Bit in der **Bitmap**
 - größere Abschnitte: mehr "Verschnitt"
 - kleinere Abschnitte: umfangreichere Bitmap
- ▶ Bit = 0: Abschnitt **frei**; Bit = 1: Abschnitt **belegt**
- ▶ Speicherblock der Größe k Einheiten angefordert
→ Bitmap nach k aufeinanderfolgenden Nullen durchsuchen (aufwendig!)



Speicherverwaltung: Listen

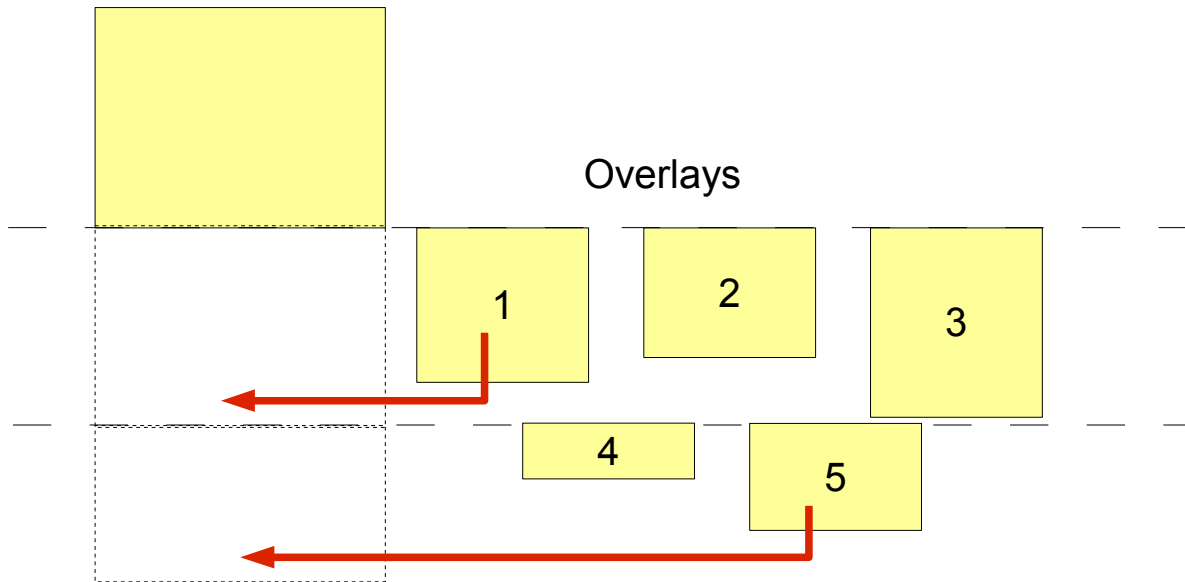


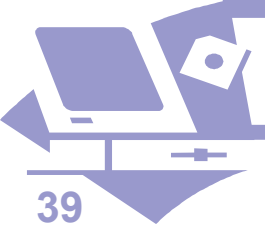
- ▶ Verkettete Liste zur Speicherverwaltung:
 - **b** = belegt, **f** = frei
 - Anfang des Speicherblocks, Länge des Speicherblocks
 - Verweis auf nächsten Eintrag der Speicherliste
- ▶ Gefundener f-Block wird (falls zu groß) ggf. gesplittet.
- ▶ Entstehen irgendwann benachbarte F-Listenelemente
→ zugehörige Speicherblöcke zusammenfassen
- ▶ Varianten: Getrennte "b"- und "f"-Listen,
Sortierung nach Größe, ...

Overlays

38

- ▶ Swapping funktioniert nur, wenn kein Programm größer als verfügbarer Speicher ist - was aber, wenn doch?
- ▶ Früher: Overlays
 - Programmierer **teilt Programm** in einzelne *Overlays* **auf** (= Aufwand für entsprechende Planung)
 - Overlays werden zur Laufzeit bei Bedarf nachgeladen





Virtueller Speicher

39

► Swapping

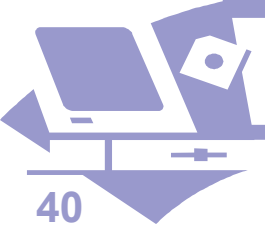
- **ganze Prozesse** werden ein- und ausgelagert
- Prozess kann nicht größer werden als Hauptspeicher

► Overlays

- **Zerteilen des Programms**, nur Teile gleichzeitig geladen
- Gesamtgröße kann Hauptspeichergröße übersteigen
- Extraaufwand bei Programmierung

► Virtueller Speicher (aktueller Ansatz)

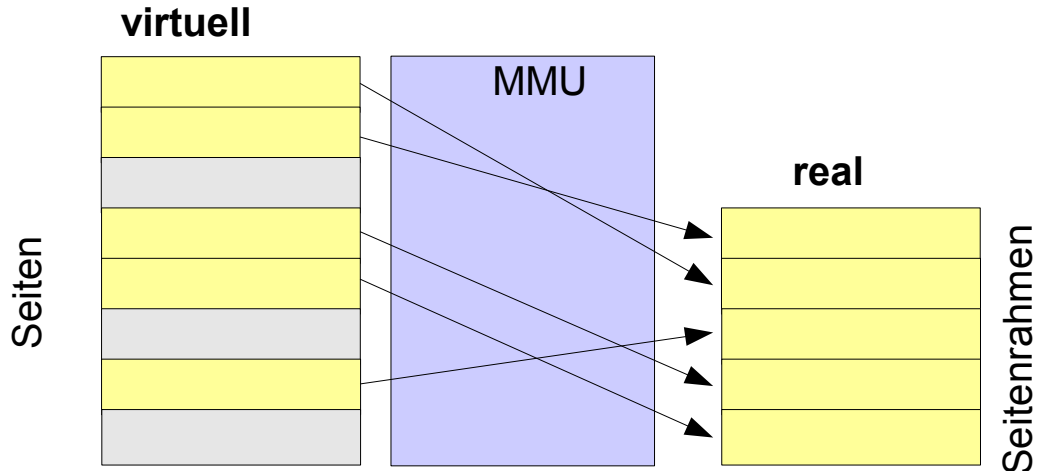
- Programme bekommen Existenz eines großen Speicherraums vorgespiegelt
- Dieser virtuelle Speicherraum wird vom Betriebssystem auf Basis von realem Hauptspeicher und Plattenplatz realisiert.
- **Für Programm ist der Unterschied nicht sichtbar**, daher keine besondere Vorkehrungen bei Programmierung nötig
- Zwei Ansätze: Paging und Segmentierung



MMU - Memory Management Unit

40

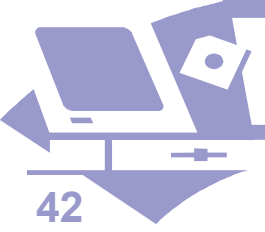
- ▶ Das Programm "sieht" nur einen *virtuellen* Adressraum
- ▶ Virtuelle Adressen werden von einer Speicherverwaltungseinheit (MMU, *memory management unit*) auf reale Adressen (im "echten" Hauptspeicher) abgebildet
- ▶ Die MMU ist heute in der Regel in den Prozessorchip integriert (also Hardware)
- ▶ Aufteilung beider Adressräume in feste Abschnitte (z.B. Seitengröße 4 kB)
- ▶ **virtueller** Adressraum: **Seiten** (*pages*)
- ▶ **realer** Adressraum: **Seitenrahmen** (*page frames*)





Seitenfehler

- ▶ MMU führt eine **Seitentabelle** mit Seiten/Rahmen-Zuordnung und Statusinformationen
- ▶ **Zugriff** auf **virtuelle** Adresse, deren zugehörige Seite momentan nicht im realen Hauptspeicher liegt
→ MMU meldet **Seitenfehler** (*page fault*) an CPU
- ▶ Betriebssystem **lädt** daraufhin zugehörige Seite in einen freien Seitenrahmen nach
- ▶ Unter Umständen muß dazu zuerst ein Seitenrahmen **freigemacht** werden (und die enthaltene Seite - falls sie geändert wurde (*dirty page*) - zuvor auf Festplatte zurückgeschrieben werden)



Seitentabelle

Beispiel: MMU, 16 Seiten zu 4 KB

Rahmen
belegt?

15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

Seitenrahmennr.

1 1 0 0 0 0 0 0 0 0 0 0 1 0 0

reale Adresse 24580

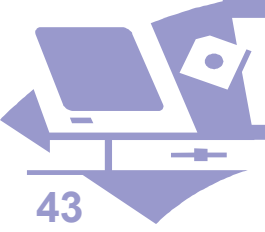
12-Bit-Offset übernehmen

Seitennr.

0 0 1 0 0 0 0 0 0 0 0 0 1 0 0

virtuelle Adresse 8196

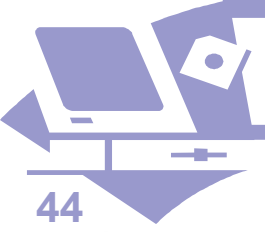
= 2



Seitentabelleneintrag

R	M	Z	B	Seitenrahmennummer
---	---	---	---	--------------------

- Neben Seitenrahmennummer und "**B**elegt"-Flag enthält Seitenrahmentabelleneintrag oft weitere Infos:
- **R**eferenziert-Flag
(auf die Seite wurde lesend und/oder schreibend zugegriffen)
 - **M**odifikations-Flag (Seite wurde verändert; "*dirty bit*")
 - **Z**ugriffsschutz (z.B. "Inhalt ist lesbar / schreibbar / ausführbar")
 - ...



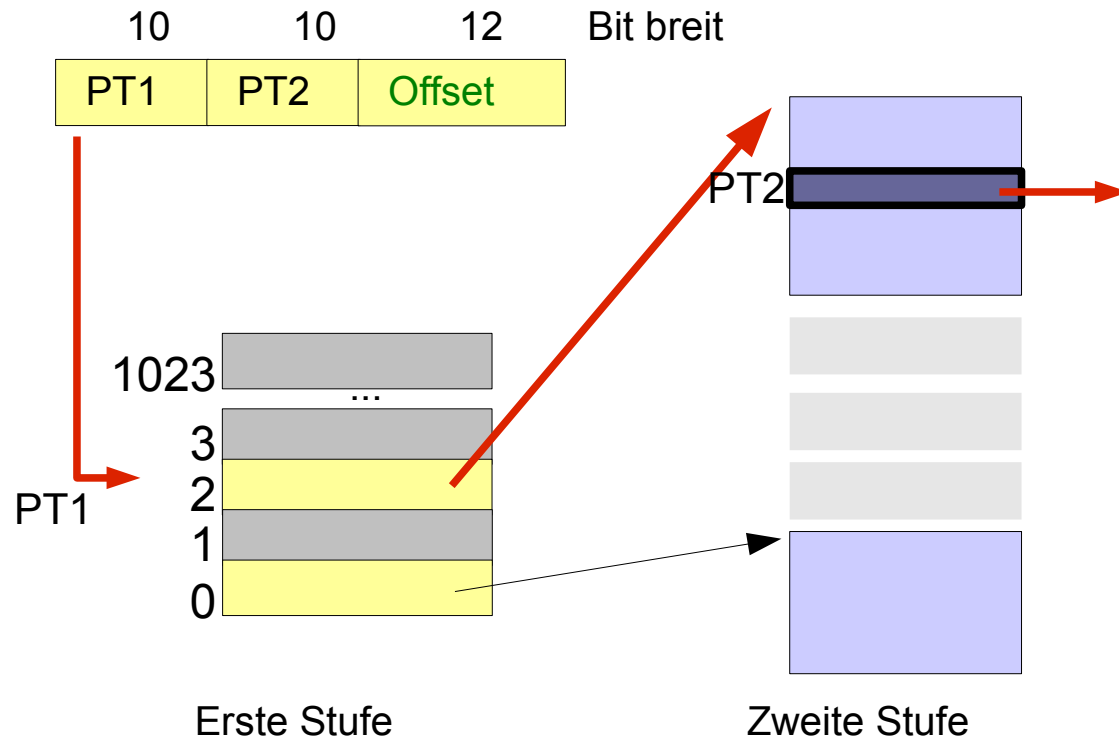
Probleme bei Seitentabellen

- ▶ Seitentabelle kann sehr groß werden. Beispiel:
virtuelle Adressen: 32 Bit lang
Seitengröße: 4 KB (also mit 12 Bit adressierbar)
=> **Seitentabellenindex** $32 - 12 = 20$ Bit lang
=> $2^{20} = 1.048.576$ **Seitentabelleneinträge!**
und: **jeder Prozess** hat heute seinen eigenen Adressraum,
damit auch seine eigene Seitentabelle!
- ▶ Problem: Seitentabelle...
in schnellen Hardwareregistern => teuer
im Hauptspeicher => langsam

Mehrstufige Seitentabellen

45

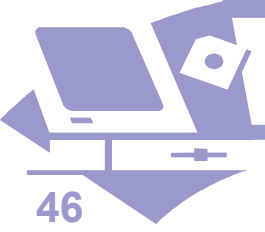
- ▶ Prozesse belegen in der Regel nicht den gesamten Adressraum
 - viele Seitentabelleneinträge bleiben also leer
 - verschwenden damit selbst Speicherplatz
- ▶ Lösung: Mehrstufige Seitentabellen (*wieso spart das Speicher?*)



PT1: Index f. Eintrag in 1. Tabellenstufe, führt zu 2. Stufe

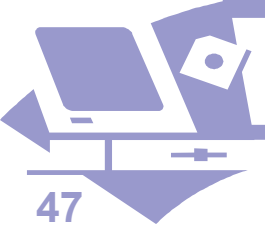
PT2: Index in 2. Stufe

Reale Adresse wie zuvor:
Inhalt 2. Stufe
(→ Seitenrahmennr) mit
angehängtem Offset



Translation Lookaside Buffer (TLB)

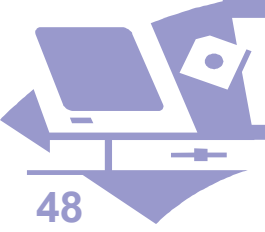
- ▶ schneller Assoziativspeicher
- ▶ direkt auf der MMU angesiedelt (Hardware)
- ▶ speichert kleinere Anzahl der zuletzt genutzten Seitentabelleneinträge zwischen
- ▶ ggf. wird dazu ältester Eintrag im TLB verdrängt
- ▶ Speicherzugriffe konzentrieren zeitlich oft auf eine kleine Seitenzahl:
 - viele Adress-Lookups werden direkt aus dem TLB beantwortet
 - wenige Hauptspeicherzugriffe auf Seitentabelle
 - im Regelfall große Beschleunigung



Seitenersetzungsverfahren

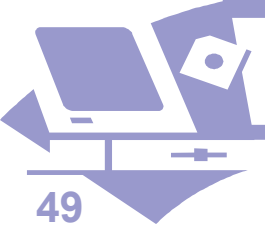
- ▶ **Auswahlproblem:** Welche Seite soll bei Auftreten eines Seitenfehlers ggf. **ausgelagert** werden?
- ▶ Ziel: Hohe Systemperformance insgesamt
- ▶ **Häufig** gebrauchte Seiten sollten also nach Möglichkeit **nicht** ausgelagert werden.

Wie kann man "*günstige*" Seiten *schnell* identifizieren?



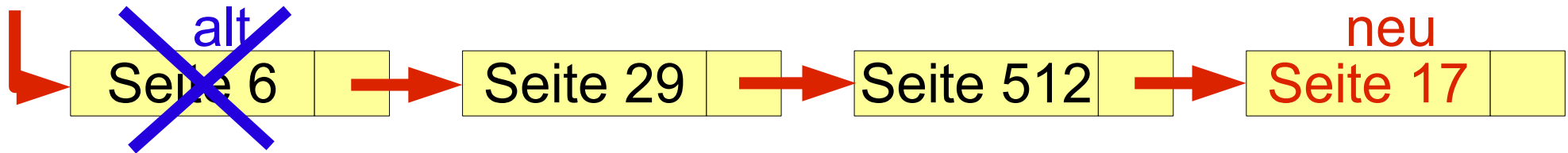
NRU - not recently used

- ▶ Zu Beginn alle **M**odifikations- und **R**eferenziert-Flags aus Seitentabelle zurücksetzen
- ▶ In regelmäßigen Zeitabständen **R**eferenziert-Flags zurücksetzen (*nicht* das Modifikations-Flag)
- ▶ Bei Seitenfehler: klassifizieren eingelagerte Seiten nach **M-/R**-Bit-Status
 - Klasse 0: nicht **referenziert**, nicht **modifiziert**
 - Klasse 1: nicht **referenziert**, aber **modifiziert**
 - Klasse 2: **referenziert**, aber nicht **modifiziert**
 - Klasse 3: **referenziert** und **modifiziert**
- ▶ Wähle eine Seite aus der nichtleeren Klasse mit der **kleinsten Nummer** zum Auslagern aus.

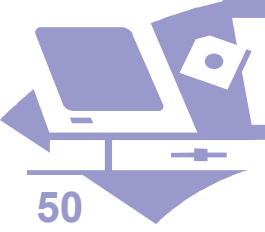


FIFO - first in, first out

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



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



Second Chance

Idee: Ähnlich FIFO, aber mit Beachtung des **R-Flags**

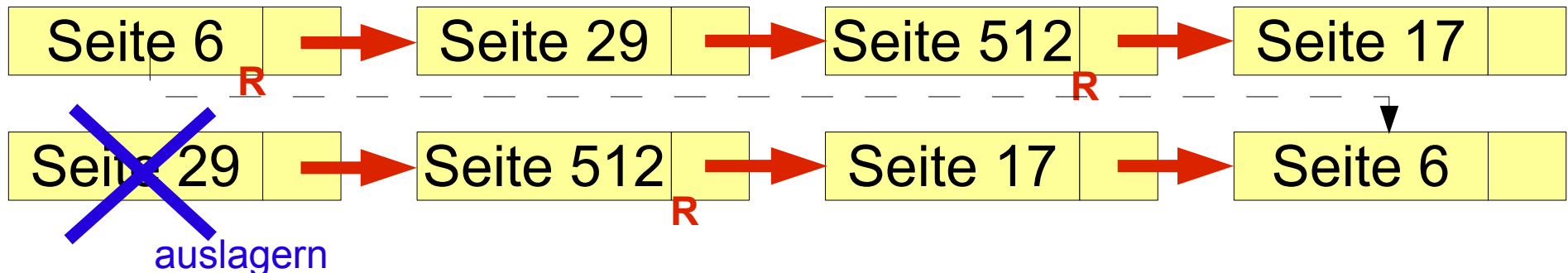
Bei **Seitenfehler**:

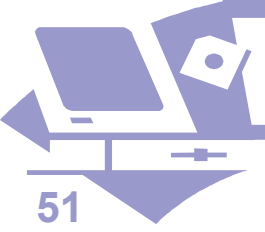
Vom Listenkopf ausgehend Seite Knoten durchlaufen:

Wenn **R-Flag gelöscht**: Seite auslagern, fertig

Wenn **R-Flag gesetzt**: löschen und Seite hinten anhängen
(also wie neu geladene Seite behandeln, „zweite Chance“)

Sollte überall **R-Flag** gesetzt sein, degeneriert Verfahren zu FIFO (erster Eintrag ist mit gelöschtem **R-Flag** wieder vorne)

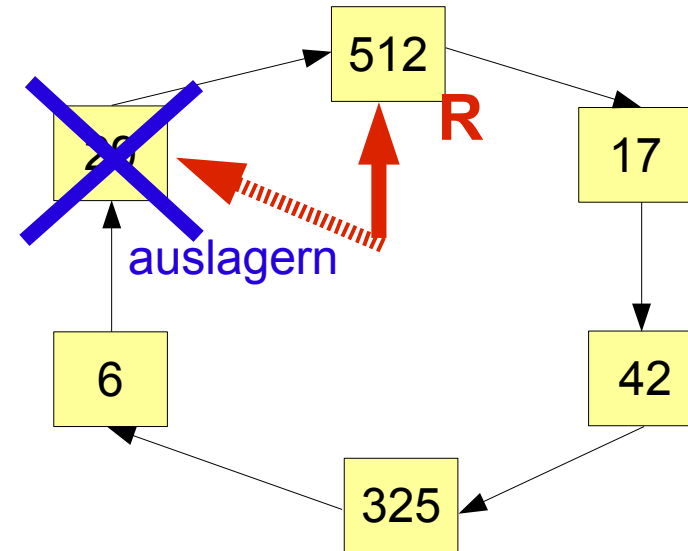
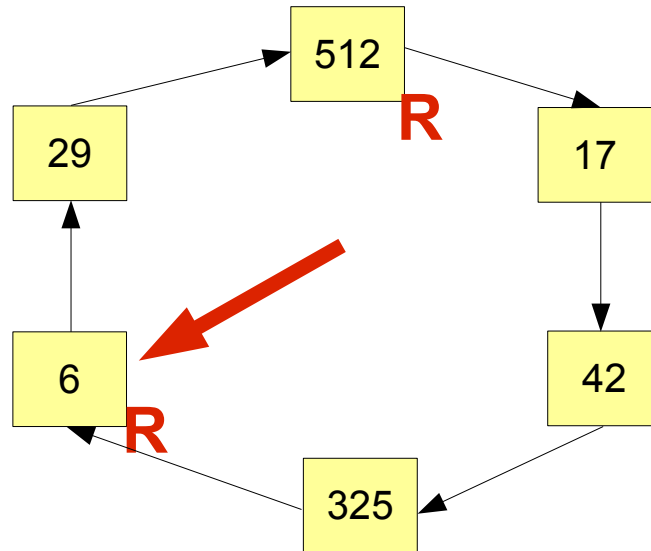




Clock-Algorithmus

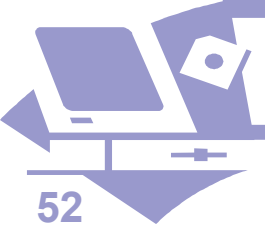
Effizientere Umsetzung von "Second Chance"

Statt Liste: Ringpuffer, "**Uhrzeiger**" zeigt auf älteste Seite



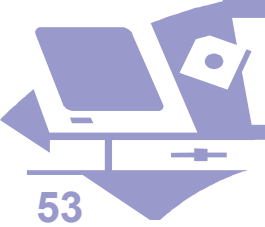
► Bei Seitenfehler:

- "R" an Zeigerposition nicht gesetzt:
Seite auslagern, Zeiger auf Nachfolger, fertig
- ansonsten: "R" löschen, Zeigeriterrücken
(bis Eintrag mit gelöschtem "R" gefunden wird)



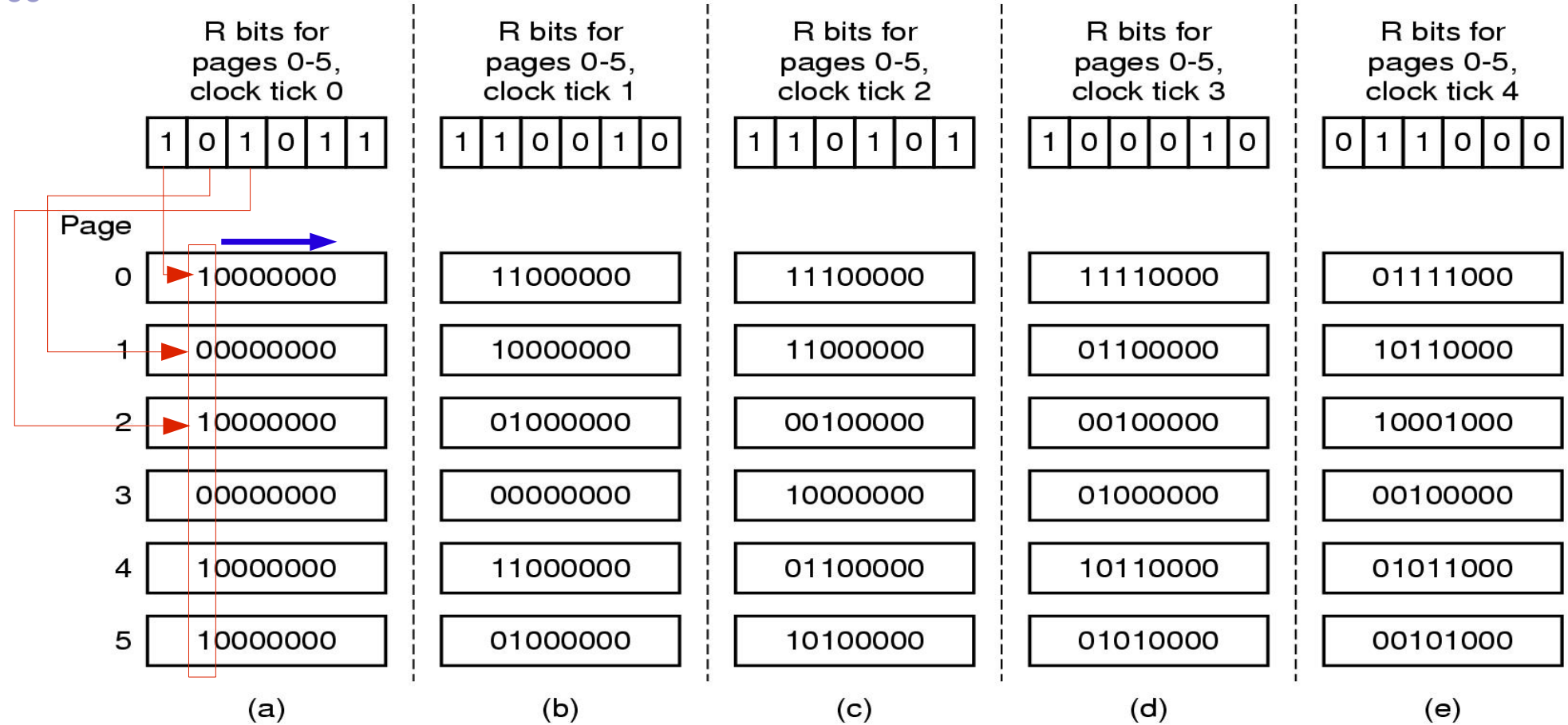
LRU - least recently used

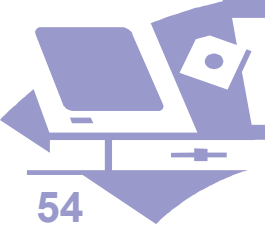
- ▶ Idee: Entferne bei Seitenfehler Seite, die **am längsten nicht mehr benutzt** wurde.
- ▶ Aufwendig zu realisieren (Seiten-Liste müsste bei jedem Speicherzugriff aktualisiert werden)
- ▶ **Näherung: Aging-Verfahren**
 - Für jede Seite gibt es einen "**Zähler**" mit n Bit Breite (z.B. 8)
 - In regelmäßigen Intervallen (z.B. 20 ms) wird für jede Seite der Zähler um **eine Bitposition nach rechts** geschoben, **links** rückt das zugehörige "**R**"-Flag für diese Seite nach; das "R"-Flag wird danach gelöscht.
 - Bei Seitenfehler wird die Seite mit dem **kleinstem Zählerstand** entfernt.



Beispiel: Aging-Verfahren

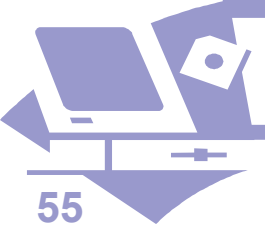
53





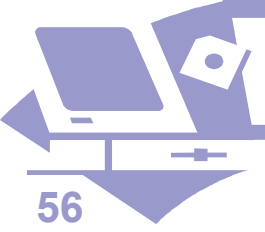
Working Set

- ▶ Prozesse zeigen in der Regel ein **Lokalitätsverhalten**
- ▶ **"Working Set"** bezeichnet eine Menge von Seiten, die ein Prozess zu einem bestimmten Zeitpunkt nutzt.
 - Gesamtes Working Set im Hauptspeicher: wenige Seitenfehler zu erwarten
 - Wenn Speicher nicht für Working Set ausreicht: Viele Seitenfehler, **"Thrashing"** (ständiges Ein-/Auslagern) mit u.U. deutlichem Performanceeinbruch.
- ▶ Ist das Working Set eines (ausgelagerten) Prozesses bekannt, so kann es genutzt werden, damit der Prozess nach dem Einlagern nicht erst durch (viele) Seitenfehler seine Arbeitsumgebung wieder aufbauen muss.
→ Performancegewinn zu erwarten
- ▶ Z.B. Aging-Algorithmus liefert Hinweise auf Working Set



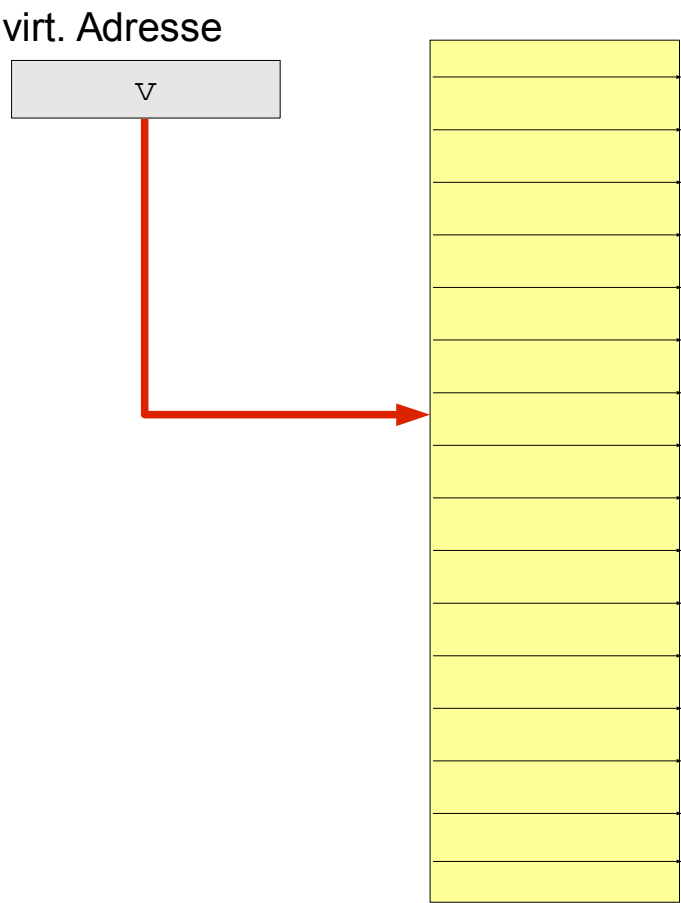
Segmentierung

- ▶ **Paging: eindimensionaler** virtueller Adressraum
- ▶ Wünschenswert: **Viele** große, unabhängige **Adressräume**
→ Segmente
 - **jedes** Segment hat lineare Folge von Adressen 0...max
 - verschiedene Segmente können **verschieden groß** sein (im Gegensatz zu Seiten)
 - Jeweilige Segmentgröße kann sich **zur Laufzeit ändern** (z.B. einzelne Segmente für Stacks, Bäume etc.)
 - Programmierer setzt Segmentierung **bewußt** ein (Verteilung von Datenstrukturen bzw. Code auf Segmente)
 - Vergabe von **Schutzattributen** pro Segment
 - Vereinfachte Nutzung von "shared libraries" (von mehreren Prozessen gemeinsam genutzten Code-Bibliotheken)

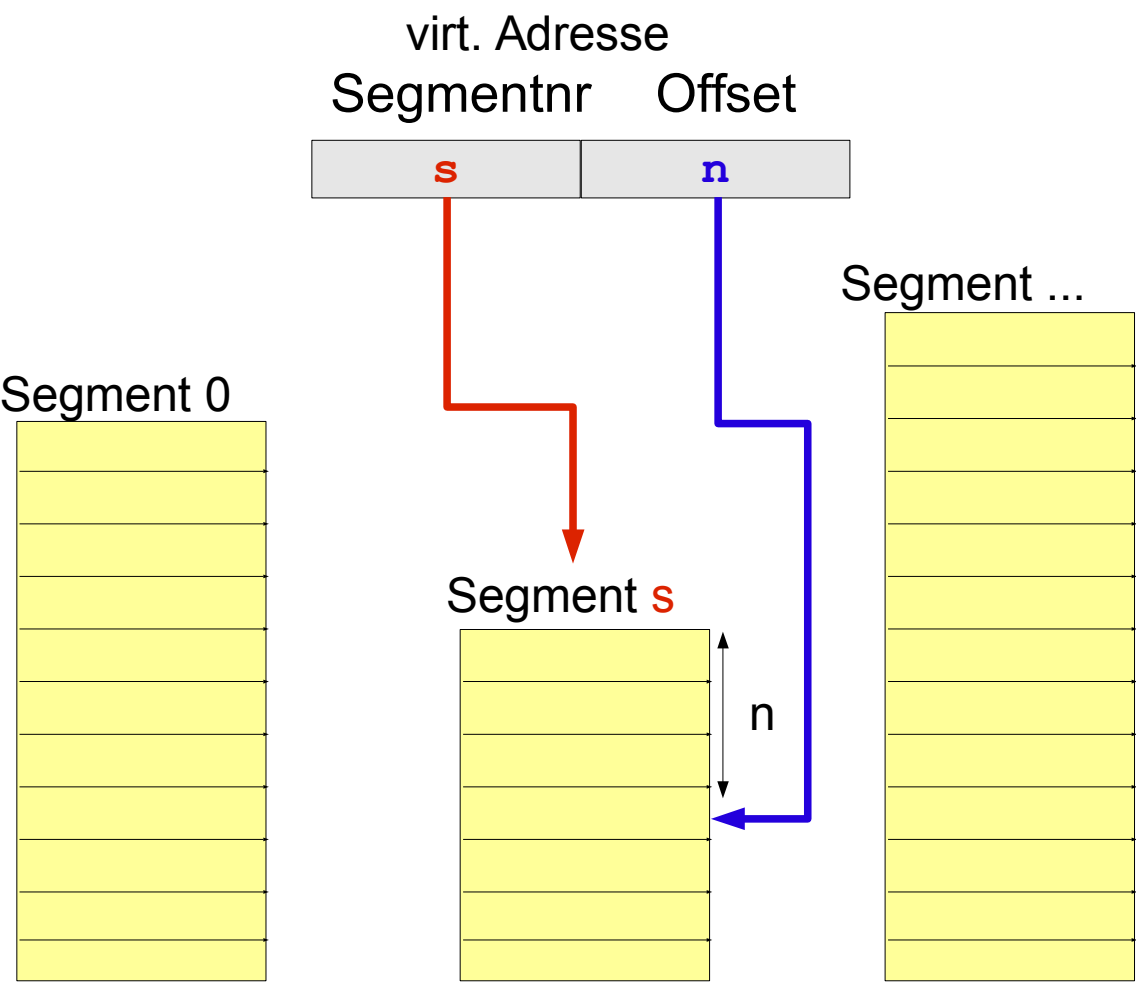


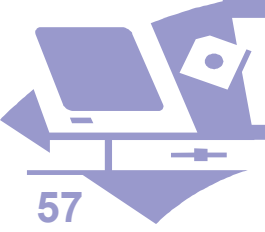
Ein-/zweidimensionale Adressräume

Eindimensional (Paging)



Zweidimensional (Segmentierung)





Segmentierung mit Paging

► Reine Segmentierung

- zu jedem Zeitpunkt einige Segmente im Hauptspeicher, andere ausgelagert
- ein **Segment** ist entweder **ganz** im Hauptspeicher oder ganz ausgelagert
- Problem: Externe Fragmentierung (vgl. "Swapping")

► Idee: **Kombination** Segmentierung / Paging

- Jedes Segment ist Folge von Seiten
- Dadurch können **Segment-Teile** ausgelagert werden

► Beispiele:

- MULTICS (UNIX-Vorgänger)

Dateisysteme



Die Speicherhierarchie

↑
schneller / teurer

Primärspeicher

CPU Register

CPU Cache

**Hauptspeicher
(RAM)**

direkter, wahlfreier Zugriff
durch den Prozessor,
sehr schnell



Sekundärspeicher
(z.B. Festplatte)

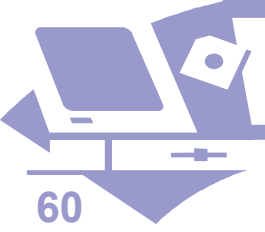


extern; wahlfreier
Zugriff auf Inhalt

Tertiärspeicher
(z.B. Backup-Bänder)

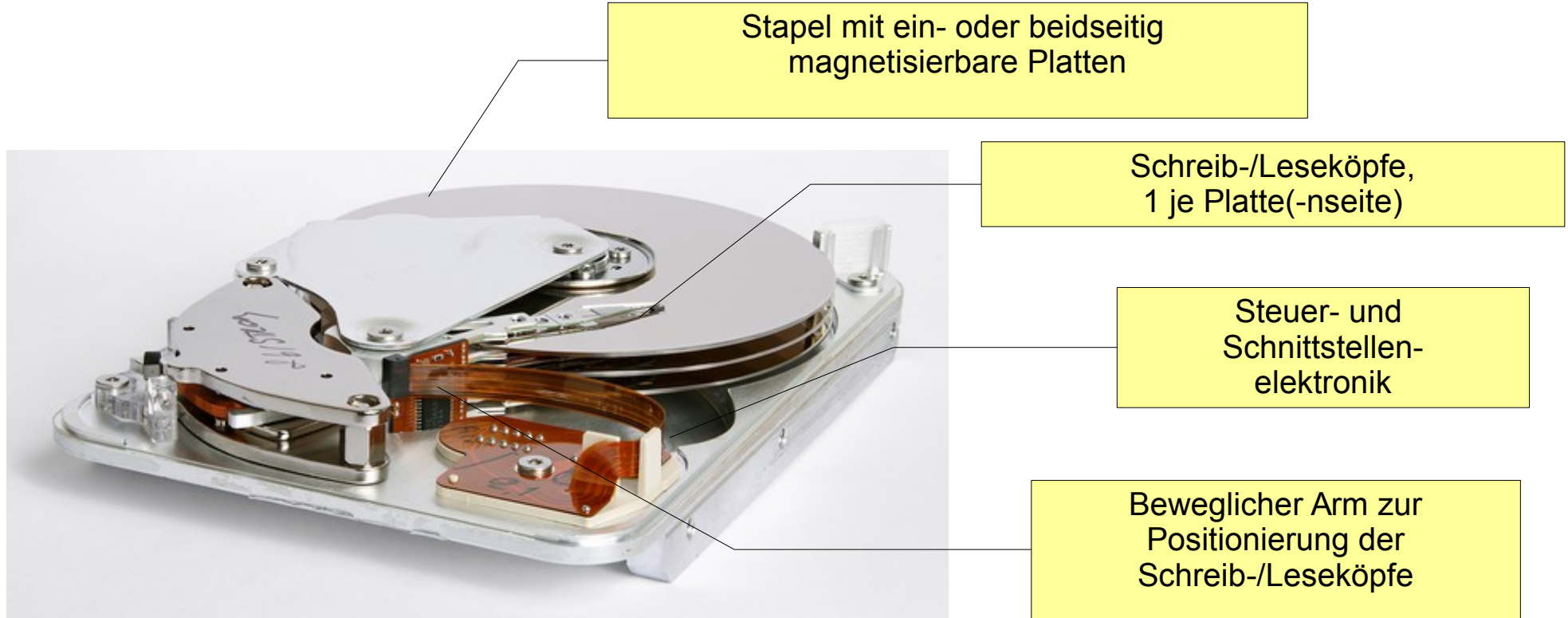


extern; langsam,
oft nur sequenzieller
Zugriff, hohe Kapazität

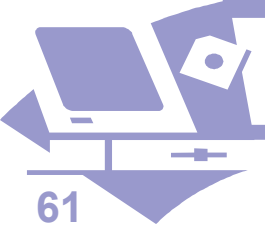


Speichermedium: Magnetplatte

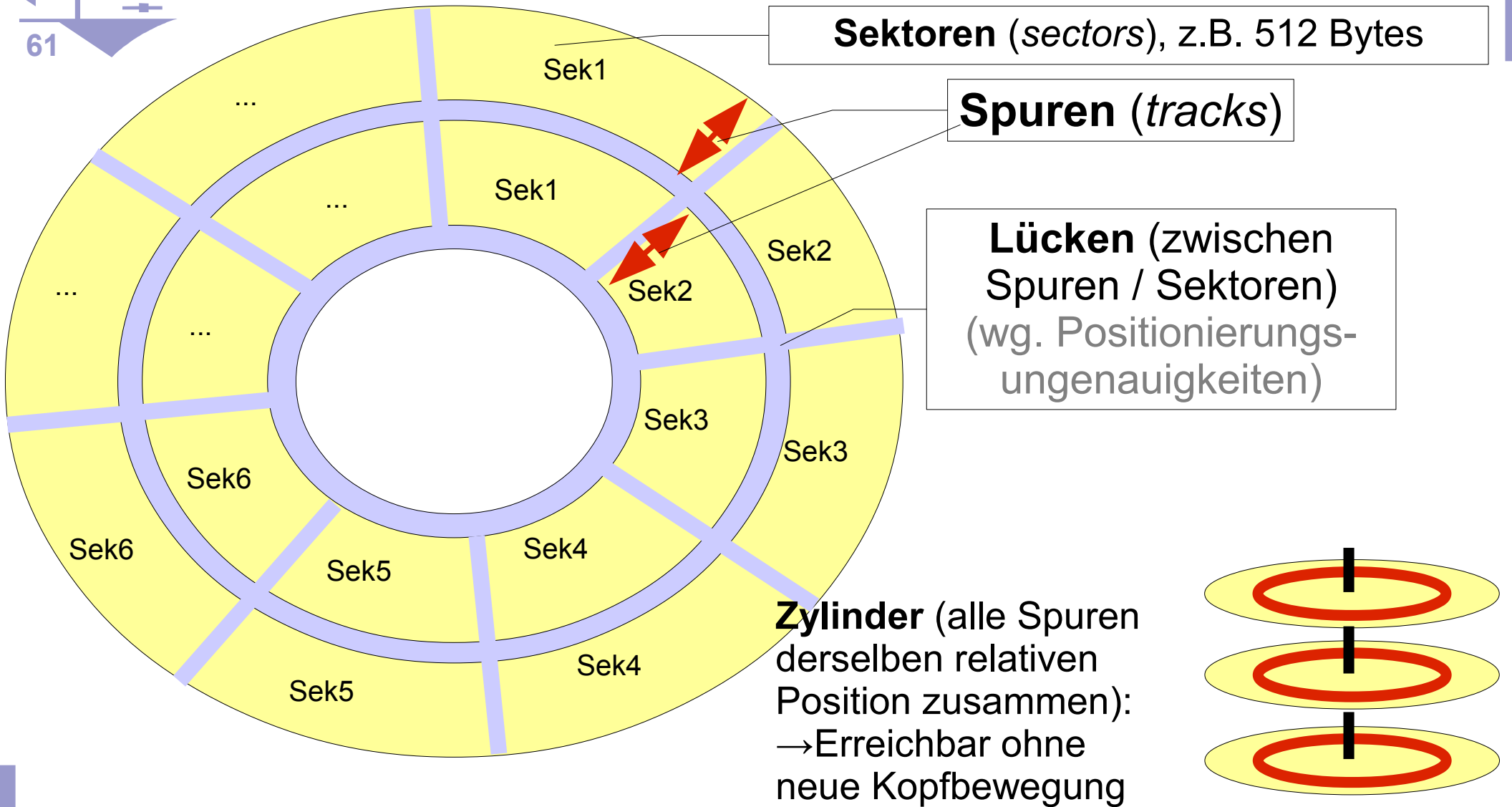
60



5400 bis über 15000 Umdrehungen/Minute
Datenübertragungsraten: bis ca 125 MBytes/s
Mittlere Positionierungszeit: ca. 5 ms und weniger



Magnetplatte: Datenorganisation



Beispiel: Festplattendaten

62

Technische Daten

Modellnummer

600 GB¹

450 GB¹

300 GB¹

ST3600057SS
ST3600957SS²
ST3600857SS³
ST3600057FC
ST3600957FC^{2,4}
ST3600857FC^{3,4}

ST3450857SS
ST3450757SS²
ST3450657SS³
ST3450857FC
ST3450757FC^{2,4}
ST3450657FC^{3,4}

ST3300657SS
ST3300557SS²
ST3300457SS³
ST3300657FC
ST3300557FC^{2,4}
ST3300457FC^{3,4}

Kapazität

Formatiert mit 512 KB/Sektor (GB)

600

450

300

Externe Übertragungsrate (MB/s)

Fibre Channel mit 4 Gbit/s

400

400

400

Serial Attached SCSI mit 6 Gbit/s

600

600

600

Leistung

Spindelgeschwindigkeit (U/min)

15.000

15.000

15.000

Durchschnittliche Latenz (ms)

2,0

2,0

2,0

Suchzeit, Lesen/Schreiben (Durchschnitt, ms)

3,4/3,9

3,4/3,9

3,4/3,9

Übertragungsrate

Intern (Mbit/s, OD-ID)

1.450 bis 2.370

1.450 bis 2.370

1.450 bis 2.370

Anhaltend (MB/s, 1.000 x 1.000)

122 bis 204

122 bis 204

122 bis 204

Cache, multisegmentiert (MB)

16

16

16

Konfiguration/Organisation

Scheiben/ Köpfe

4/8

3/6

2/4

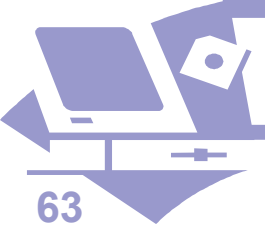
Nicht korrigierbare Lesefehler pro gelesenem Bit

1 Sektor pro 10¹⁶

1 Sektor pro 10¹⁶

1 Sektor pro 10¹⁶

Datenblatt-Auszug
Hersteller: Seagate
Modell: Cheetah 15k.7



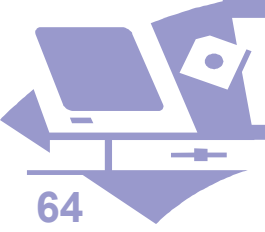
Aufgaben des Dateisystems

63

Will ein Anwendungsentwickler Sektoren, Spuren etc. selbst ansteuern, seine Daten in 512-Byte-Blöcke aufteilen müssen usw? Vermutlich nein.

Er will beispielsweise...

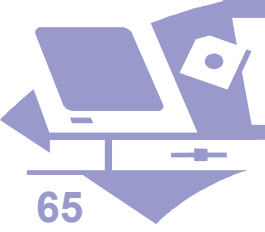
- ▶ Daten (unter einem **Namen**) abspeichern und den Bestand sinnvoll strukturiert zugänglich machen
- ▶ Operationen ausführen wie
 - sequentielles Lesen/Schreiben
 - wahlfreies Lesen/Schreiben
 - Löschen, (Um-)Benennen, Kopieren
 - ...
- ▶ optimale **Hardwareausnutzung** (ohne eigene HW-Kenntnisse)
- ▶ einheitlichen Zugang zu *vielen* (**verschiedenen**) Speichergerätearten (standardisierte Schnittstelle)
- ▶ Vergeben und Überwachung von **Zugriffsrechten**
- ▶ **Zugriffskoordination** bei Mehrbenutzerbetrieb



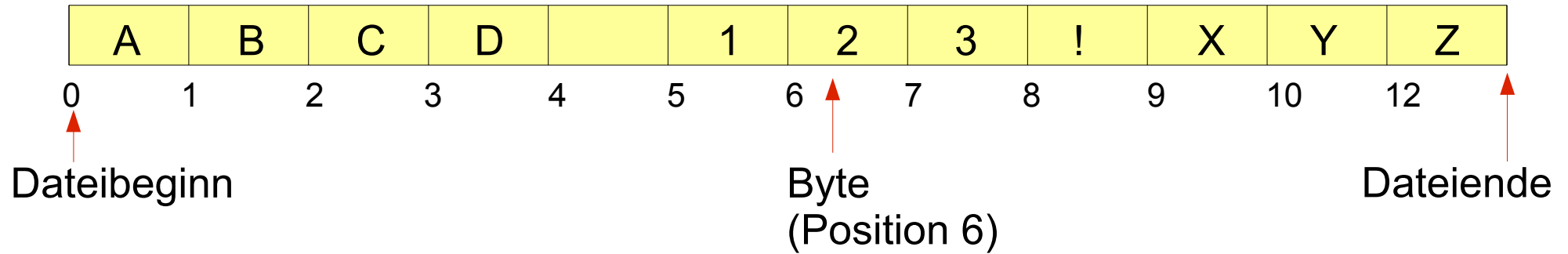
Datei, Dateisystem

64

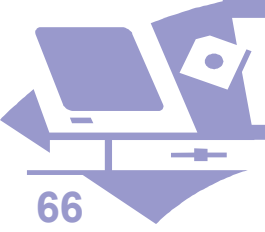
- ▶ Das **Dateisystem** (*file system*) ist der Teil des Betriebssystems, der für die Verwaltung von Dateien zuständig ist
- ▶ Eine **Datei** (*file*) ist
 - eine logische Einheit zur Speicherung von Informationen
 - auf (externen) Speichermedien,
 - dauerhaft (persistent): "Inhalt überlebt Programmende",
 - durch mehrere Prozesse (gleichzeitig) nutzbar
 - mit einem Dateinamen versehen
- ▶ Zulässigkeit von **Dateinamen** hängt vom Dateisystem ab
 - Längenbeschränkung (z.B. früher MS-DOS: 8+3 Zeichen)
 - Unterscheidung von Groß-/Kleinschreibung (Windows: nein, UNIX: ja)
 - Beschränkung zulässiger Zeichen (z.B. "keine Umlaute")



Byte-orientierte Dateistruktur

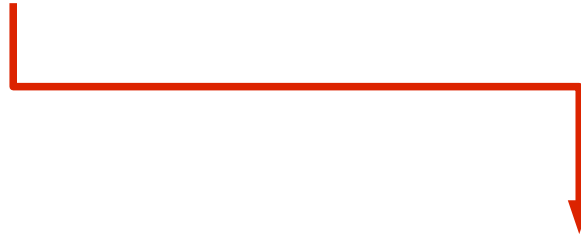


- ▶ Datei wird als **Folge von Bytes** aufgefasst
 - Für Dateisystem unstrukturiert
(Interpretation der Bytefolge durch die zugreifenden Anwendungen)
 - Einfach und flexibel
- ▶ Sequenzielle Verarbeitung und/oder wahlfreier Zugriff durch Ansteuern einer Byte-Position
- ▶ z.B. MS-DOS, UNIX, ...

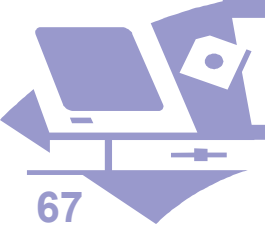


Satz-Dateistruktur

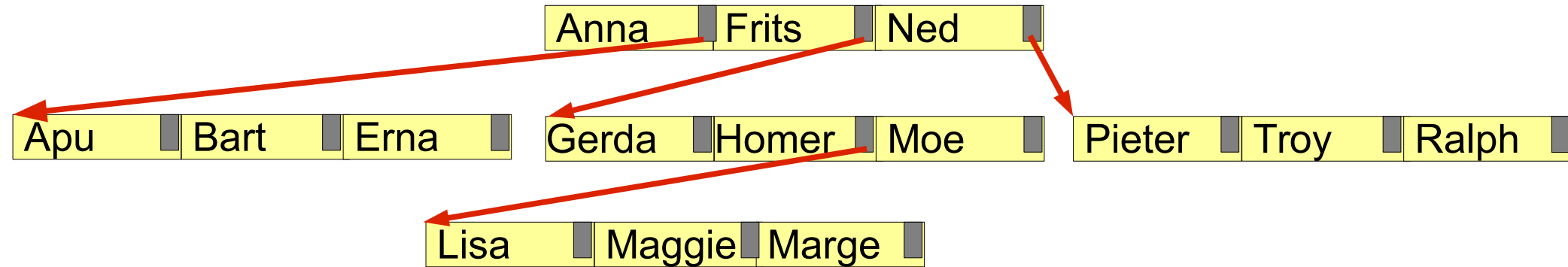
- ▶ Datei als Folge von Sätzen (Records) oft **fester Länge** (System kennt Satzlänge, aber nicht inneren Aufbau)
- ▶ Lese / Schreib / Änderungsoperationen für ganze Sätze
- ▶ Sätze sind über ihre Satznummer ("durch Abzählen") direkt ansteuerbar



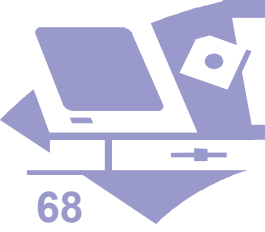
Satz 0	Satz 1	Satz 2	Satz 3	Satz 4	Satz 5	Satz 6
Meier 25.1	Huber 19.5	Wegner 11.0	Nöggi 16.2	Berger 17.3	Huber 18.0	Jokel 19.2



Baum-Dateistruktur



- ▶ **Datei besteht aus** Sätzen (Records) möglicherweise unterschiedlicher Länge, die
 - nach einem Schlüssel (identifizierender Satz-Bestandteil) sortiert sind; Dadurch ist neben sequentiell auch ein
 - direkter Zugriff über gewünschten Schlüsselwert möglich.
- ▶ Im Wesentlichen im Großrechner-Bereich verbreitet



Dateitypen

68

► Reguläre Dateien

- enthalten Benutzerdaten, Programme usw.
- **Text**dateien: Textzeilen variabler Länge, durch i.a. *betriebssystemabhängiges* Kontrollzeichen getrennt (UNIX: "`\n`", MacOS: "`\r`", Windows: "`\r\n`")
- **Binär**dateien: Rest; ausführbare Programme, Word-Daten, ...

► Verzeichnisse

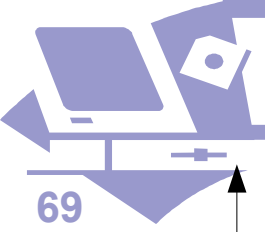
► Systemdateien zur Verwaltung des Dateisystems

► Zeichenorientierte Spezialdateien

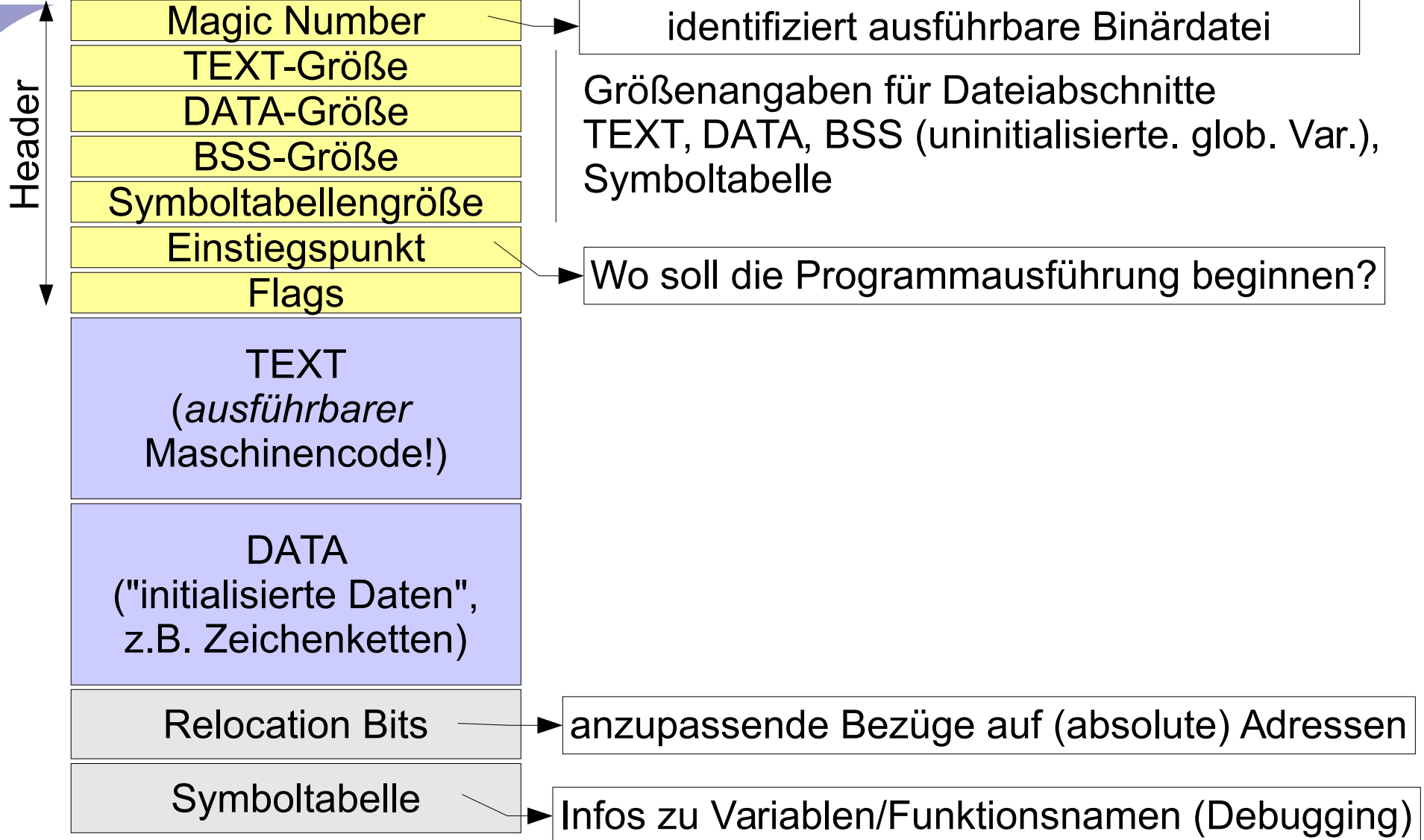
- Schnittstelle zu zeichenorientierten Ein-/Ausgabegeräten wie Druckern, Terminals, Modems
- z.B. Linux: `/dev/ttyS0` für erste serielle Schnittstelle

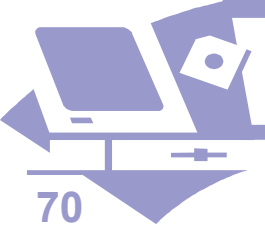
► Blockorientierte Spezialdateien

- Schnittstelle zu blockorientierten Geräten wie Festplatten
- z.B. Linux: `/dev/sda2` (Partition 2, der 1. (→'a') Platte)



Beispiel: Ausführbare UNIX-Datei





Datei-Zugriffsarten

Neben der Dateioorganisation unterscheidet man verschiedene Zugriffsarten. Eine Organisationsform kann dabei eine oder mehrere Zugriffsarten erlauben:

► **Sequentieller** Zugriff

- Verarbeitung "von vorne nach hinten", ggf. "zurückspulen";
- entspricht Zugriff auf Magnetband.

► Direkter, **wahlfreier** Zugriff (*random access*)

- Positionierung auf beliebige Byte-Position als Startpunkt für Dateioperationen (z.B. Lesen) jederzeit möglich
- mit Hilfe einer speziellen Funktion (seek) oder durch Parameter zu gewünschter Dateioperation.

► **Indexsequentieller** Zugriff (ISAM)

- Datensätze haben Schlüsselfelder, für die das
- Dateisystem eine Index-Struktur verwaltet.
- Zugriff direkt über Schlüsselwert möglich.

Dateiattribute

71

► Systemabhängige Zusatzinformationen zu Datei, z.B.

- Datum: Erstellung, letzte Änderung, letzter Zugriff
- Ersteller / Besitzer der Datei
- Systemdatei-Flag
- „archiviert“-Flag
- Dateityp
- Schlüsselposition / -länge
- Dateigröße
- Zugriffsschutz-Informationen

► Beispiel: `ls -l /usr/bin/vi`

Zugriffsrechte	Besitzer	Gruppe	Größe	Änderungszeitpunkt	Name
-rwxr-xr-x	1 root	editors	456076	Dez 23 23:02	vi

↑ ↑ ↑ ↑
...für "alle"
...für Gruppe
...für Besitzer
Dateityp: "d"=Verzeichnis, "-"=reguläre Datei, ...

Read, Write, eXecute (lesen, schreiben, ausführen)



Fehlerbehandlung, perror()

72

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

Beispiel:

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

```
$ a.out gibts_nicht irgendwas
```

```
Fehler beim Umbenennen: No such file or directory
```

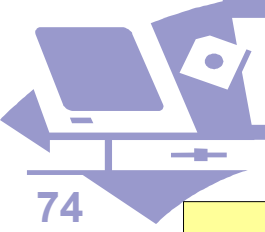



open()

73

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

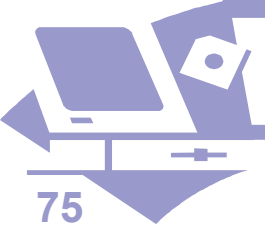
- ▶ `pathname` - Name oder Pfad der zu öffnenden Datei
- ▶ `flag` - wie soll die Datei geöffnet werden? (→ `fcntl.h`)
 - `O_RDONLY` nur lesen, `O_WRONLY` nur schreiben
 - `O_RDWR` lesen und schreiben
- ▶ Per Bit-ODER können verschiedene Flags hinzugefügt werden
 - `O_APPEND` Schreibzugriffe: am Dateiende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



Beispiel: open()

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

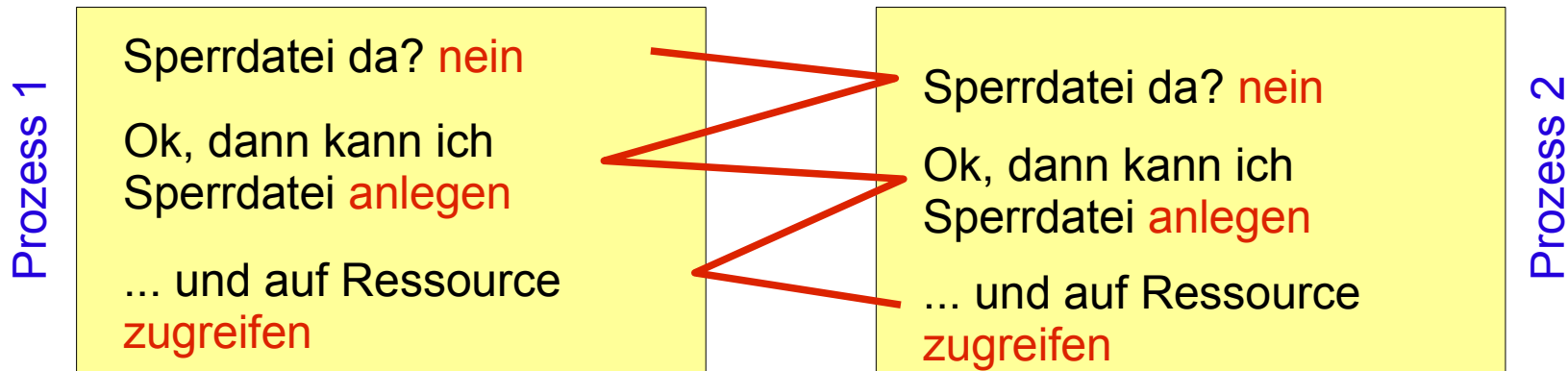
- ▶ Datei `test-1` kann über Dateideskriptor `fd1` gelesen werden
- ▶ Datei `test-2` kann über Dateideskriptor `fd2` beschrieben werden. Dabei wird am Ende angehängt.
- ▶ Datei `test-3` kann über Dateideskriptor `fd3` geschrieben und gelesen werden.
 - Falls die Datei noch nicht existiert, wird sie angelegt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

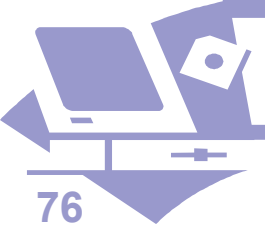


Beispiel: Sperrdateien

75

- ▶ Ziel: Exklusiver Zugriff auf eine Ressource (z.B. Modem)
- ▶ Problem: Wie bekommen konkurrierende Prozesse den Status der Ressource (belegt/frei) heraus?
- ▶ Idee: **Sperrdatei** (*lock file*) in einem gemeinsamen Verzeichnis anlegen
 - Datei existiert: Ressource gesperrt
 - Datei fehlt: Ressource frei
- ▶ Achtung: Existenztest & ggf. Anlegen **muß in einem Schritt** geschehen, sonst Gefahr von Überschneidungen:





Realisierung: sperren()

```
int sperren(char *pfad) {
    int fd;
    fd = open(pfad, O_WRONLY | O_CREAT | O_EXCL, 0644);
    if (fd >= 0) close(fd);
    return fd >= 0;
}

int freigeben(char *pfad) {
    return (unlink(pfad) == 0);
}
```

```
...
while ( !sperren("/tmp/mein_modem") ) {
    /* warten ...*/
}
/* Nutzung der gesperrten Ressource */
freigeben("/tmp/mein_modem");
...
```



Lesen, Schreiben, Schließen

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

▶ read()

- liest bis zu `anzahl` Bytes vom Dateideskriptor `fd` in den Hauptspeicher ab Adresse `daten` ein
- Rückgabewert: Anzahl tatsächlich gelesener Bytes oder -1 für Fehler

▶ write()

- schreibt (bis zu) `anzahl` Bytes ab Adresse `daten` auf `fd`
- Rückgabewert: Anzahl tatsächlich geschriebener Bytes oder -1 für Fehler

▶ close()

- schließt Datei mit Deskriptor `fd`

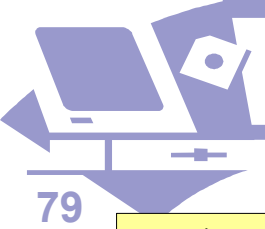
Beispiel: Kopierprogramm (1)

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

int main(int argc, char *argv[]) {
    enum { BUFSIZE=1000 };
    char buffer[BUFSIZE];
    int lese_fd, schreib_fd, gelesen, geschrieben;
    if (argc != 3) { printf("Falscher Aufruf\n"); exit(1); }

    lese_fd = open(argv[1], O_RDONLY);
    if (lese_fd < 0) {
        perror("Bei Oeffnen der Eingabedatei");
        exit(2);
    }
    schreib_fd = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT, 0644);
    if (schreib_fd < 0) {
        perror("Bei Oeffnen der Ausgabedatei");
        exit(3);
    }
}
```

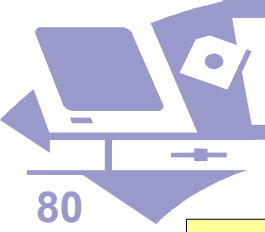
*Programm sofort mit
Rückgabewert 1 beenden*



Beispiel: Kopierprogramm (2)

```
/* Fortsetzung ... */
```

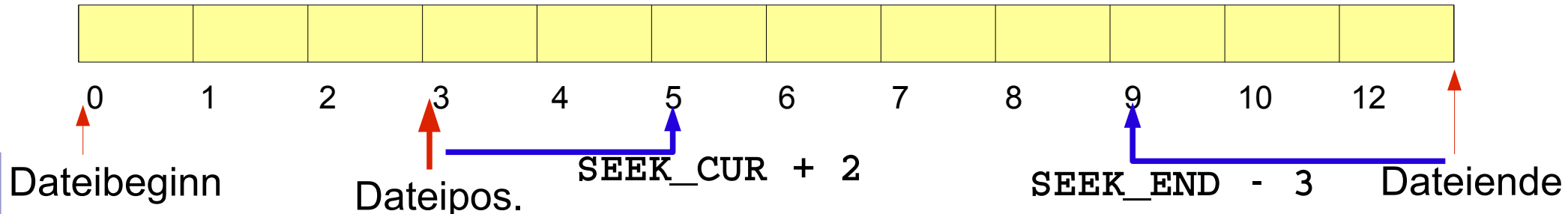
```
while (1) {  
    gelesen = read(lese_fd, buffer, BUFSIZE);  
    if (gelesen == 0) {  
        break;  
    } else if (gelesen < 0) {  
        perror("Lesefehler");  
        break;  
    }  
    geschrieben = write(schreib_fd, buffer, gelesen);  
    if (geschrieben <= 0) {  
        perror("Schreibfehler");  
        exit(4);  
    }  
}  
  
close(lese_fd);  
close(schreib_fd);  
return gelesen == 0? 0 : 5;  
}
```



Direktpositionierung: lseek()

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

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





Beispiel: Direktzugriff

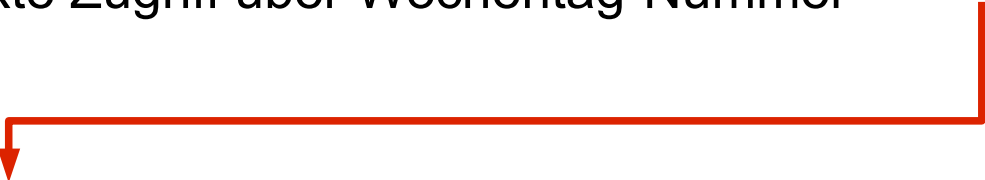
81

struct-Typ beschreibt Meßwerte-Datensatz

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

► Ziel:

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



Satz 0	Satz 1	Satz 2	Satz 3	Satz 4	Satz 5	Satz 6
Meier	Huber	Wegner	Nöggi	Berger	Huber	Jokel
25.1	19.5	11.0	16.2	17.3	18.0	19.2



speichern() mit Direktzugriff


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

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

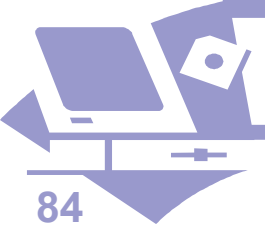


stat(): Dateiattribute abfragen

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



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



Dateiattribute setzen (Auswahl)

Zugriffsrechte ändern

```
int chmod(char *Pfad, mode_t Rechte);
```

Ergebnis: 0 für ok, -1 für Fehler

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

Dateibesitzer / -gruppe ändern

```
int chown(char *path, uid_t owner, gid_t group);
```

Ergebnis: 0 für ok, -1 für Fehler

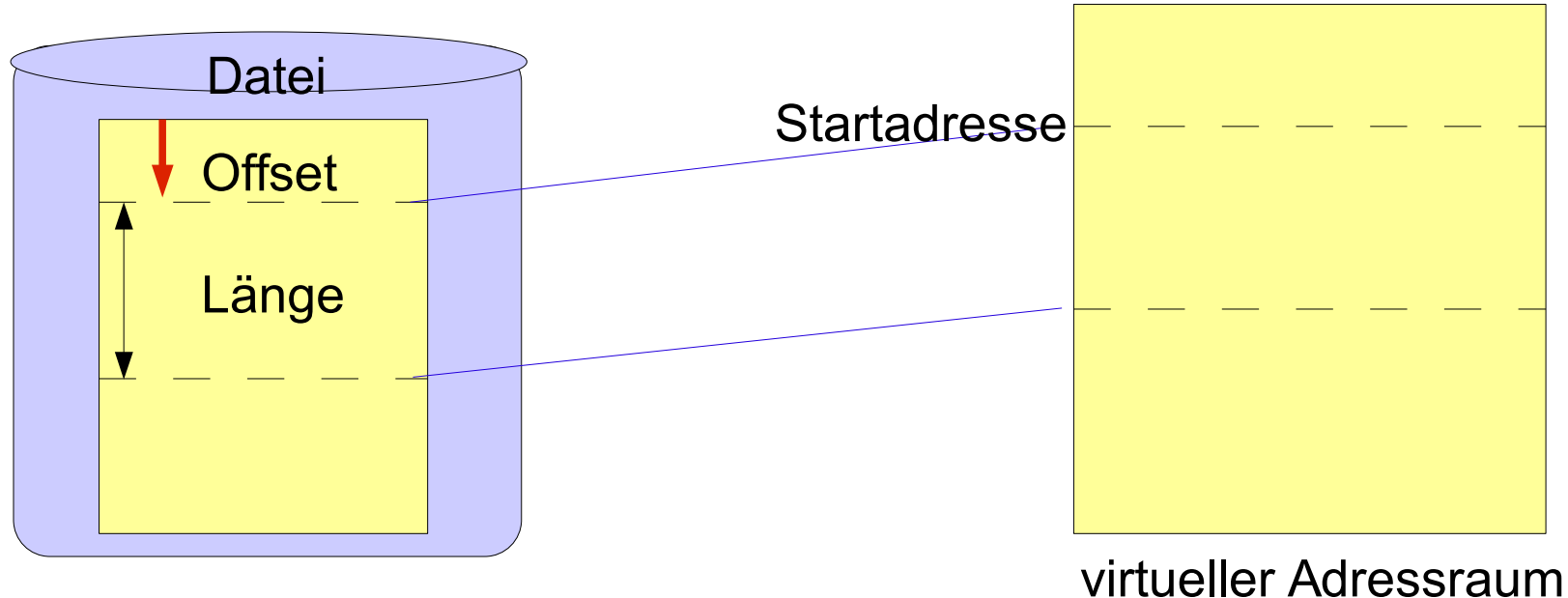
```
if ( chown("meineDatei.txt", 7, 27) == 0 ) {  
    /* ok! */  
}
```

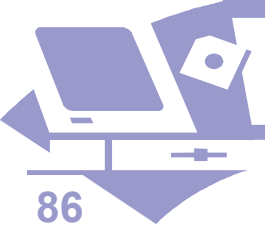
Hinweis: ID-Nummern für Eigentümer (uid) und Gruppe (gid) stehen z.B. in der Datei `/etc/passwd`; Angabe von -1: keine Änderung

Memory-mapped Files

85

- ▶ (Teile von) Dateien können in den Adressraum des verarbeitenden Prozesses eingeblendet werden.
- ▶ Zugriff auf Dateiinhalte dann wie normale Speicherzugriffe (statt mit `read()` / `write()`)
- ▶ Oft basierend auf Managementfunktionen für virtuellen Speicher realisiert





mmap()

```
#include <unistd.h>
```

```
#include <sys/mman.h>
```

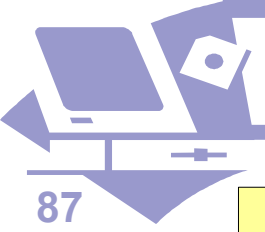
```
void *mmap(    void *start, size_t length, int prot,  
              int flags, int fd, off_t offset);
```

- length Bytes von Dateideskriptor fd ab Position offset
- sollte ab Adresse start eingeblendet werden (start == 0: System wählt Adresse selbst)
- prot gibt Zugriffsart an (lesen, schreiben, ausführ.)
- flags: z.B. MAP_SHARED: Änderungen für andere sichtbar
- Ergebnis: Anfangsadresse oder -1 bei Fehler

```
int munmap(void *start, size_t length);
```

- Aufheben des Mappings

Die ganze Wahrheit: man mmap

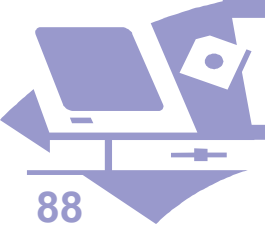


Beispiel: mmap()

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

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

    return 0;
}
```



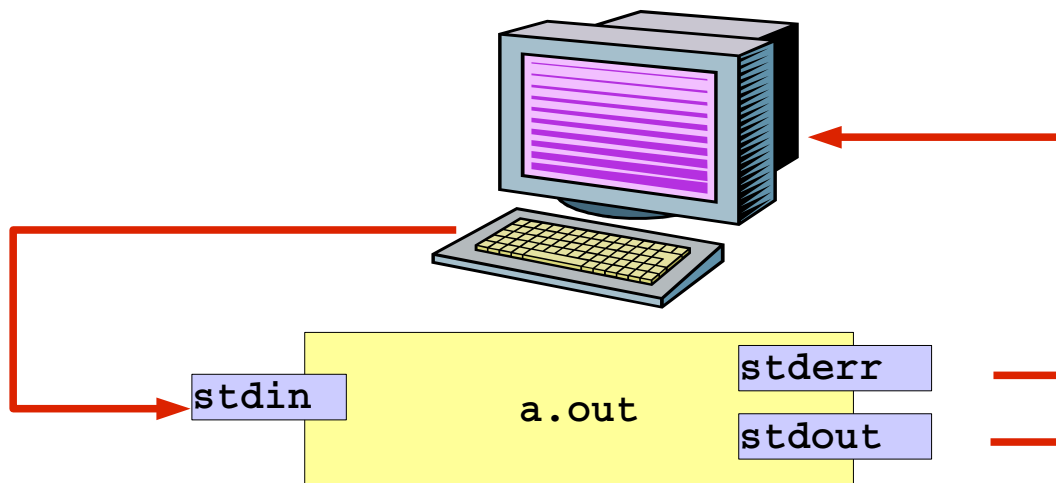
Dateisysteme (2)



Standardein-/ausgabe

90

- ▶ Unter UNIX gibt es drei spezielle Dateideskriptoren:
 - 0 = Standardeingabe (`stdin`)
 - 1 = Standardausgabe (`stdout`)
 - 2 = Standardfehlerausgabe (`stderr`)
- ▶ Bei interaktiver Verwendung aus einer Shell ist üblicherweise
 - Standardeingabe = Tastatur,
 - Standard(fehler)ausgabe = Terminalfenster



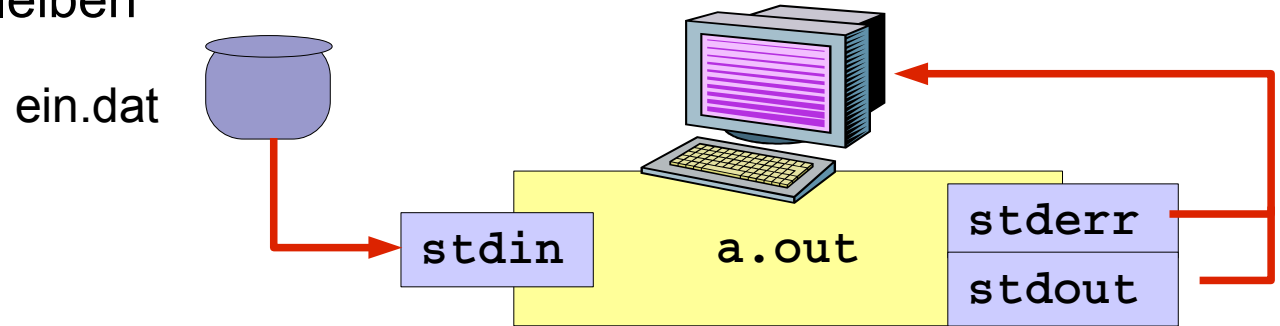


Ein-/Ausgabeumleitung

91

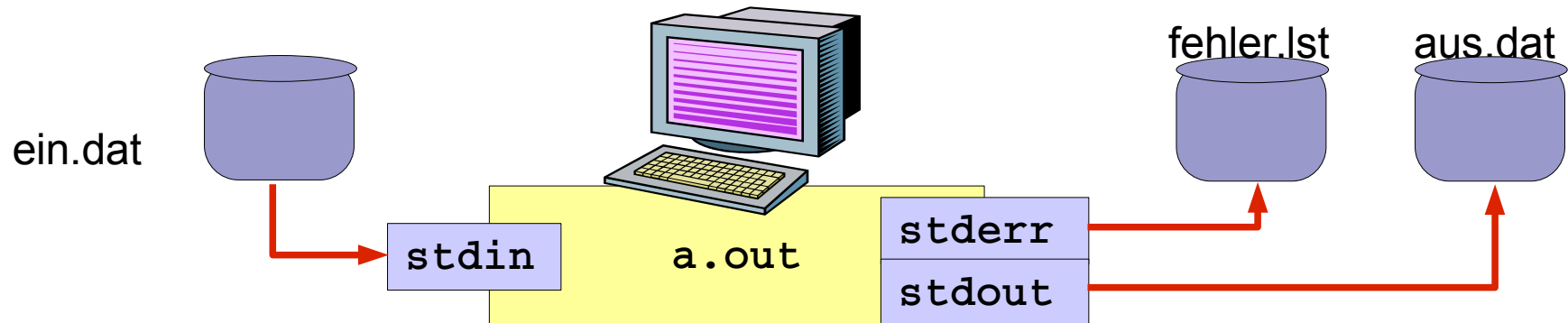
`a.out <ein.dat`

Setzt Standardeingabe (stdin) auf Datei "ein.dat",
Standard(fehler)ausgabe bleiben



`a.out <ein.dat >>aus.dat 2>fehler.lst`

Setzt Standard**e**ingabe auf Datei "eingabe.dat",
std**o**ut hängt ggf. an Datei aus.dat an, **std****e**rr legt fehler.lst (neu) an



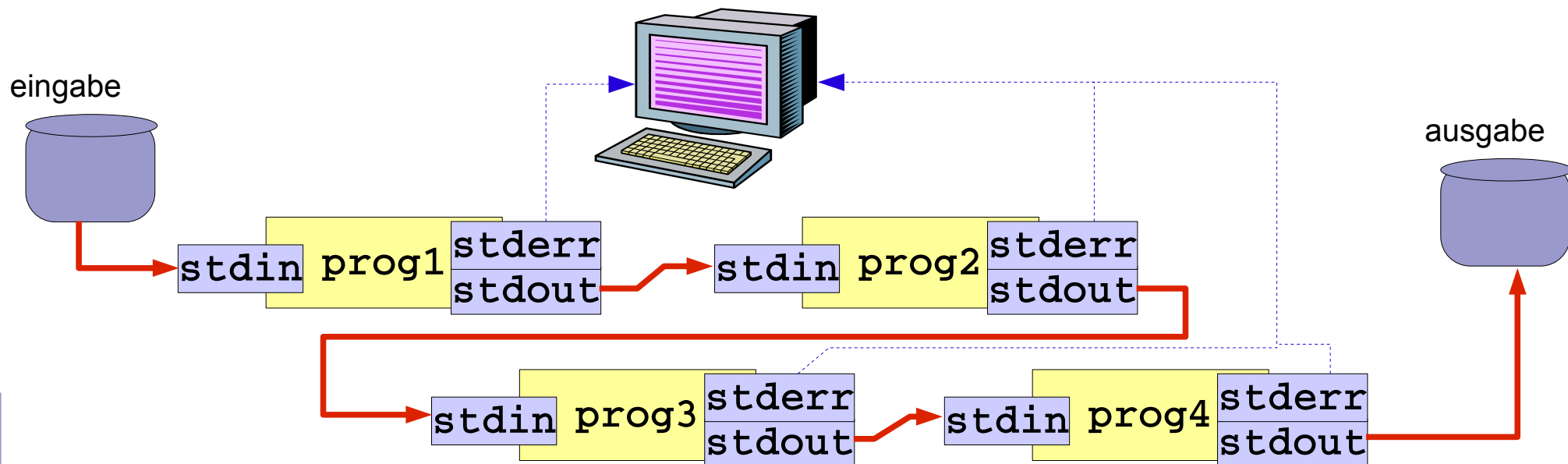
Pipelines

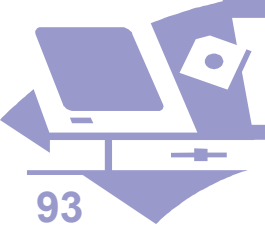
92

(Vgl. „Einführung in die Medieninformatik“)

- ▶ Standardausgabe eines Programms wird mit der Standardeingabe eines anderen Programms verbunden
- ▶ Alle beteiligten Programme laufen in parallelen Prozessen, Teilausgaben können direkt weiterverarbeitet werden
- ▶ Es fallen keine (u.U. umfangreichen) Zwischendateien an

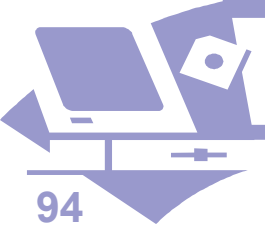
```
prog1 <eingabe | prog2 | prog3 | prog4 >ausgabe
```



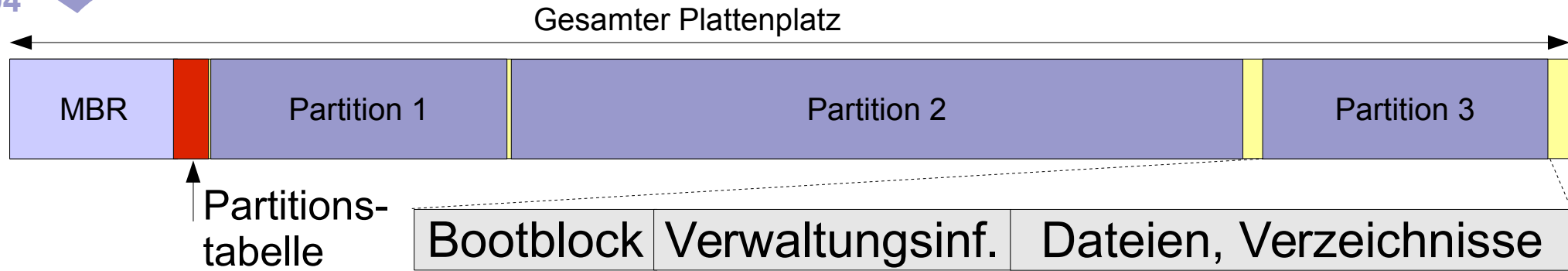


Implementierung von Dateisystemen

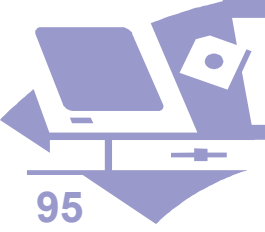
- ▶ Anfängliche Aufteilung der Festplatte
- ▶ Verwaltung des Plattenplatzes im laufenden Betrieb
- ▶ Darstellung von Verwaltungsinformationen
- ▶ Umsetzung von Dateien / Verzeichnissen



Datenträger-Aufteilung (Beispiel: MBR)



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



Beispiel: Linux fdisk

95 Partitionierungstabelle der Festplatte /dev/sda (=erste Festplatte)

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

```
...
```

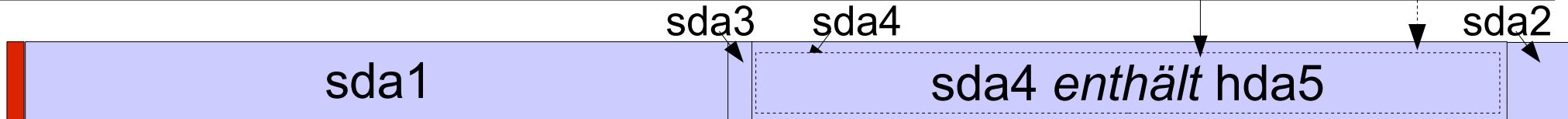
```
Befehl (m für Hilfe): print
```

```
Festplatte /dev/sda: 240 Köpfe, 63 Sektoren, 2584 Zylinder
```

```
Einheiten: Zylinder mit 15120 * 512 Bytes
```

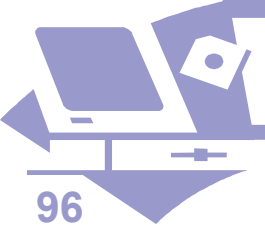
Gerät	boot.	Anfang	Ende	Blöcke	Id	Dateisystemtyp
/dev/sda1		1	1163	8792248+	c	Win95 FAT32 (LBA)
/dev/sda2	*	2386	2584	1504440	c	Win95 FAT32 (LBA)
/dev/sda3		1164	1173	75600	83	Linux
/dev/sda4		1174	2385	9162720	f	Win95 Erw. (LBA)
/dev/sda5		1174	2385	9162688+	8e	Linux LVM

Partition table entries are not in disk order



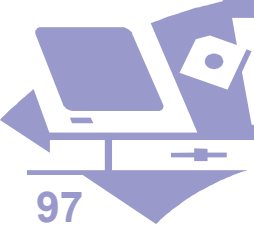
MBR

sda4 ist sogenannte "erweiterte Partition",
die Unterpartitionen (hier: sda5) enthalten kann



Realisierung von Dateien

- ▶ Zuordnung von Speicherblöcken auf der Platte zu Dateien
- ▶ Lösungsmöglichkeiten z.B.
 - zusammenhängende Belegung
 - verkettete Liste / Allokationstabelle
 - inodes



Zusammenhängende Belegung

97

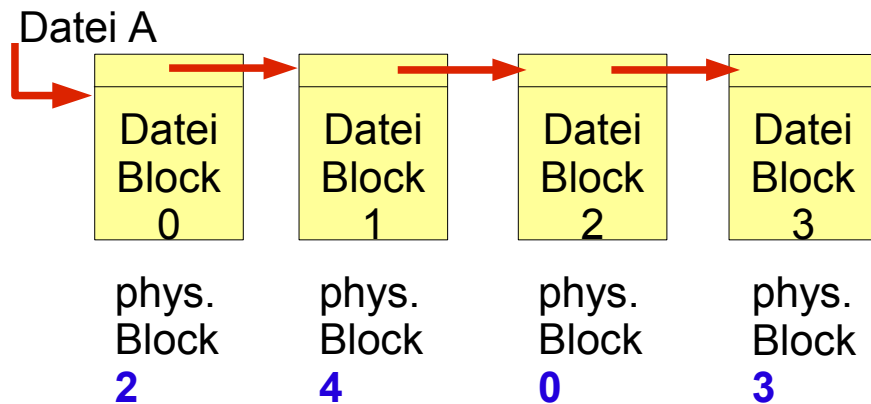
Datei 1	Datei 2	D.3	Datei 4	D.5	D.6
---------	---------	-----	---------	-----	-----

- ▶ Datei wird auf eine zusammenhängende Folge von Plattenblöcken abgespeichert
- ▶ Reservierung einer festen Zahl von Blöcken beim Datei-Anlegen.
- ▶ **Vorteile**
 - einfach zu implementieren: Speicherplatz für Datei durch Anfangsblock und Länge beschrieben,
 - sehr schnell (Minimum an Kopfpositionierungen)
- ▶ **Nachteile**
 - max. Größe zu Beginn festzulegen
 - externe Fragmentierung ("Verschnitt") bei wiederholtem Löschen/Anlegen von Dateien unterschiedlicher Größe
 - De-Fragmentierung aufwendig
- ▶ **Anwendung: z.B.**
 - Echtzeit-Anwendungen (schnell), CD-ROMs (Größe fest)

Verkettete Liste, Allokationstab.

98

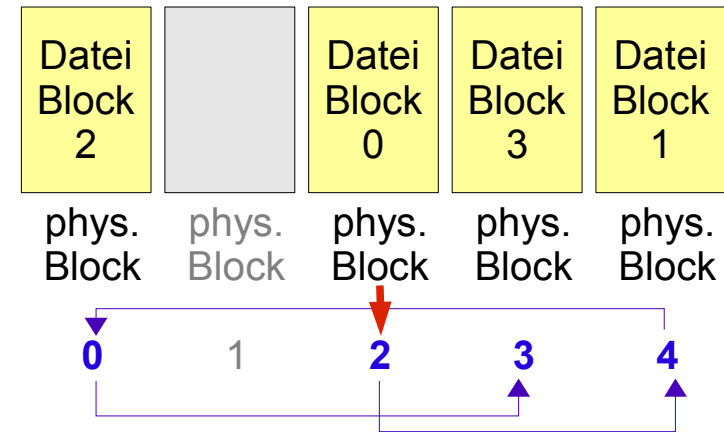
- ▶ Datei: Speicherblöcken durch **Verweise** miteinander verkettet
- ▶ Jeder Speicherblock hat Verweis auf **Nachfolger**-Block.
 - (a) Verweis direkt am Beginn jedes Speicherblocks oder
 - (b) Verweise in Allokationstabelle (in Hauptspeicher geladen)



phys. Block / Nachfolgerblock

Datei A

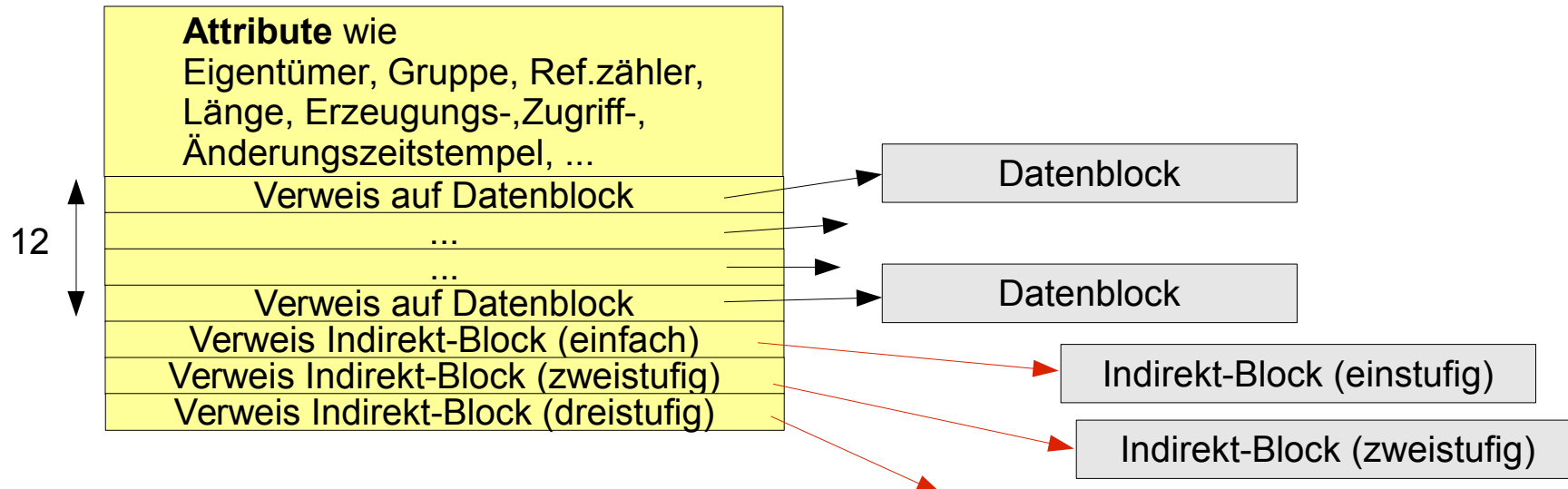
0	→	3
1		
2	→	4
3		
4	→	0
5		
Allokationstabelle		

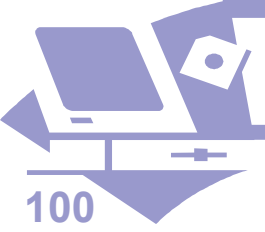


- ▶ keine externe Fragmentierung
- ▶ Verzeichniseintrag verweist auf ersten Block der Datei
 - Bei (a): wahlfreier Zugriff seeehr langsam, bei (b) akzeptabel
 - Bei (b): Allokationstabelle braucht bei großen Platten viel Hauptspeicher
- ▶ Beispiel (b): MS-DOS FAT (File Allocation Table) Dateisystem

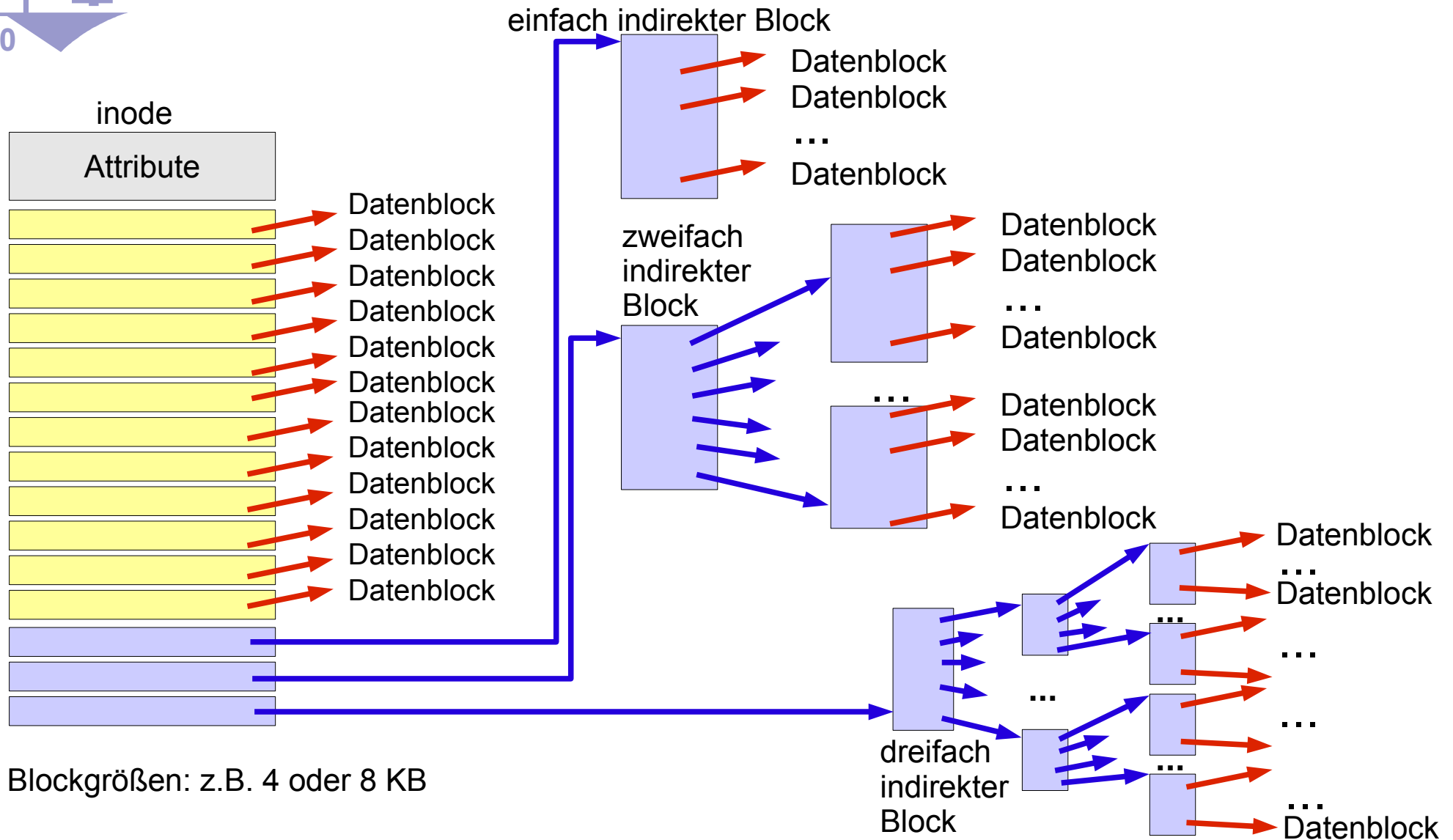
I-Nodes (z.B. UNIX)

- ▶ I-Node (*index node*, *inode*): Dateikontrollblock je Datei mit
 - Dateiattributen
 - Adressen der Plattenblöcke *dieser Datei*
- ▶ I-Nodes werden *nur für geöffnete Dateien* in Hauptspeicher geladen (vgl. dagegen: FAT)
- ▶ Datei-Wachstums-Problem: Verweis auf weitere Verweisblöcke (über ein- bis dreistufige Indirekt-Blocks)





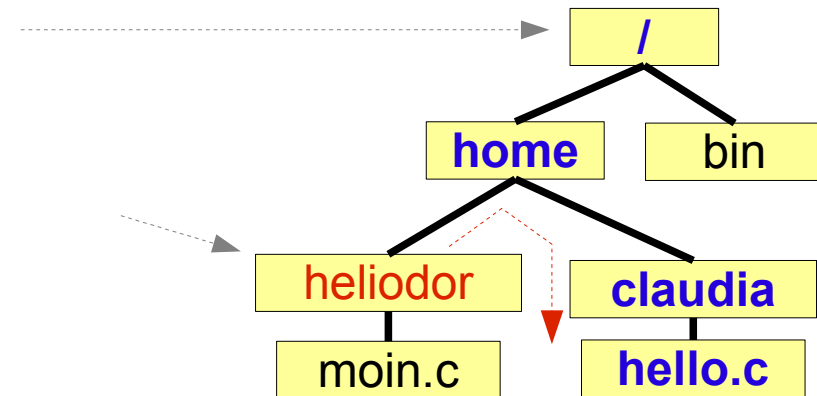
inodes: Indirekt-Blöcke





Dateiverzeichnisse

- ▶ Das Dateisystem führt üblicherweise ein Verzeichnis über die von ihm verwalteten Dateien.
- ▶ Je nach Dateisystem können Verzeichnis ein- oder mehrstufig sein (d.h. selbst Unterverzeichnisse enthalten; hierarchische Verzeichnissysteme)
- ▶ Dies ist nützlich zur Umsetzung einer
 - inhaltlichen Strukturierung des Dateibestands und zur
 - Zugriffskontrolle auf die enthaltenen Dateien
- ▶ Bezeichnung einer Datei geschieht durch
 - **absolute** Pfadnamen (ab Wurzel)
z.B. (UNIX) `/home/claudia/hello.c`
 - **relative** Pfadnamen, bezogen auf das gerade *aktuelle Verzeichnis* (".")
z.B. `moin.c`, `./moin.c`, `../claudia/hello.c`
(".." ist das jeweilige Vater-Verzeichnis)
- ▶ Das Pfad-Trennzeichen ist systemabhängig!





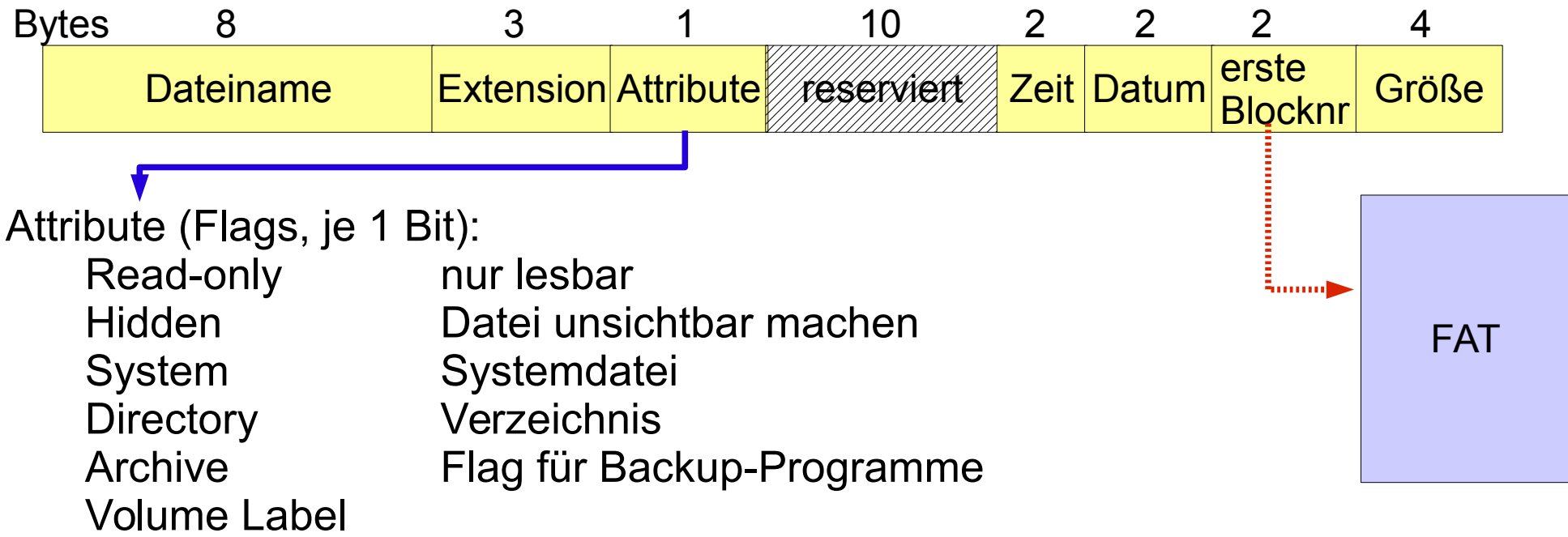
Realisierung von Verzeichnissen

- ▶ Lokalisierung der zugeordneten Plattenblöcke auf Basis des Dateinamens / der Pfadangabe (Zeichenketten!)
- ▶ Verzeichniseintrag identifiziert je nach verwendetem Konzept der Realisierung von Dateien:
 - bei **zusammenhängender Belegung**:
 - Anfangsblock + Länge
 - oder Unterverzeichnis
 - bei verketteter Liste / **Allokationstabelle**
 - Plattenadresse des ersten Blocks
 - oder Unterverzeichnis
 - bei Verwendung von **inodes**:
 - Inode-Nummer
 - oder Unterverzeichnis



Verzeichniseintrag FAT-Filesys

- ▶ Hierarchisches Verzeichnissystem
- ▶ FAT (file allocation table) mit Verkettungsinformationen in Allokationstabelle

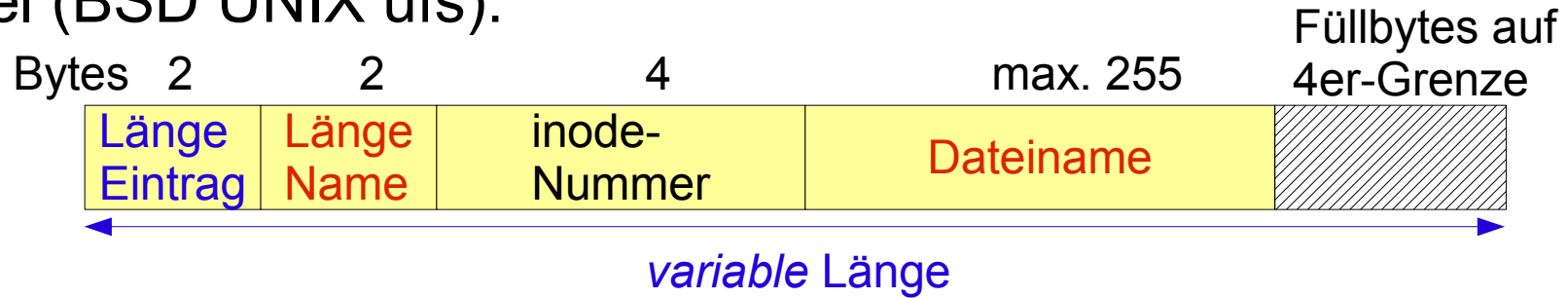




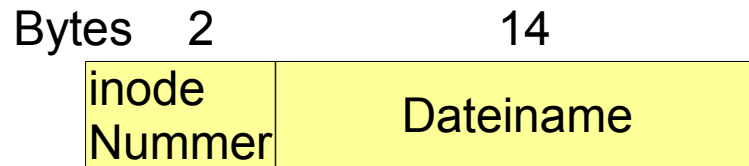
Verzeichniseintrag UNIX

- ▶ Hierarchisches Verzeichnissystem
- ▶ Allokation von Plattenblöcken mit inodes

Beispiel (BSD UNIX ufs):



Beispiel (klassisches UNIX System V)





Operationen auf Verzeichnissen

► Operationen auf Verzeichnissen

- **Links** ("harte" und symbolische Links)
- **Verzeichnisse** anlegen und löschen
- Verzeichnis**inhalt** ermitteln
- Speichergeräte **in Dateibaum einhängen** (mount/umount)



link, unlink

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

- ▶ `link()` legt einen neuen Verzeichniseintrag `newpath` an, der auf dieselbe Datei (inode) wie der bestehende `oldpath` verweist, und erhöht den Referenzzähler im inode. (Keine Datei-Kopie!)

- ▶ Entsprechendes Shell-Kommando:

`ln alter_eintrag neuer_eintrag`

- ▶ `unlink()` erniedrigt Referenzzähler im zugehörigen inode und löscht Verzeichniseintrag (sowie Datei, sobald Referenzzähler Null ist)
- ▶ Verwendet z.B. im Shell-Kommando "`rm`"



ls -li

```
$ ls -li /usr/bin
```

-rwxr-xr-x	2	root	root	2003 Jun 23 2017	zdiff
-rwxr-xr-x	3	root	root	3029 Jun 23 2017	zegrep
-rwxr-xr-x	3	root	root	3029 Jun 23 2017	zfgrep
-rwxr-xr-x	1	root	root	1016 Jun 23 2017	zforce
-rwxr-xr-x	3	root	root	3029 Jun 23 2017	zgrep
-rwxr-xr-x	1	root	root	8408 Aug 4 2017	zic2xpm
-rwxr-xr-x	1	root	root	64344 Jun 24 2017	zip

- ▶ `ls -li` zeigt Anzahl der Verweise (Links) auf den inode einer Datei
- ▶ Option `-li` zeigt zusätzlich inode-Nummer; so werden verschiedene Verweise auf gleichen inode erkennbar:

```
$ cd /usr/bin; ls -li -i z*grep
```

376494	-rwxr-xr-x	3	root	root	3029 Jun 23 2017	zegrep
376494	-rwxr-xr-x	3	root	root	3029 Jun 23 2017	zfgrep
376494	-rwxr-xr-x	3	root	root	3029 Jun 23 2017	zgrep
377039	-rwxr-xr-x	1	root	root	1180 Jun 24 2017	zipgrep

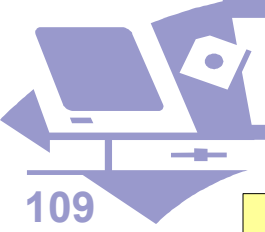


„Selbstaufräumende Zwischendateien“

```
int tmpopen(char *path) {  
    int fd;  
    fd = open(path, O_RDWR | O_TRUNC | O_CREAT, 0600);  
    if (fd) {  
        unlink(path); /* alter Trick! */  
        return fd;  
    } else {  
        return -1;  
    }  
}
```

```
tmpfd = tmpopen("/tmp/zwischendatei");  
...  
write(tmpfd, ...);  
...  
close(tmpfd);
```

- ▶ Datei wird geöffnet und Verzeichniseintrag sofort gelöscht.
- ▶ Einzige Referenz auf Datei ist die durch `open()` erzeugte
- ▶ Sobald Datei geschlossen wird (`close()` / Programmende), wird der Plattenplatz "automatisch" freigegeben



symlink, readlink

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

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

Sym.Links: ls -l

110

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

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

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

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

- ▶ Verweisziel wird bei sym.Links von `ls` angezeigt ("`->`")
- ▶ Typkennzeichen in `ls`-Ausgabe: "`l`" (symLink)
- ▶ Hier: Zugriff auf `/usr/lib/sendmail` führt *letztlich* auf die Datei `/usr/sbin/sendmail.sendmail`
- ▶ Die Längenangabe bei Symlinks gibt offenbar *nicht* die Größe der Ziel-Datei an... (sondern was?)



mkdir, rmdir, chdir

```
#include <unistd.h>
int mkdir(char *pathname, mode_t mode);
int rmdir(char *pathname);
int chdir(char *pathname);
```

- ▶ `mkdir()` legt Verzeichnis `pathname` mit Zugriffsrechten `mode` an und erzeugt Verzeichnis-Einträge für "." und ".."
- ▶ `rmdir()` löscht das (bis auf die Einträge "." und ".." leere!) Verzeichnis `pathname`
- ▶ `chdir()` setzt das aktuelle Verzeichnis für den ausführenden Prozess auf `pathname`
- ▶ Wieso ist der Verweis-Zähler für ein Verzeichnis mindestens 2?

```
$ mkdir beispiel
$ ls -l
drwx----- 2 jockel studis 4096 Apr 20 15:00 beispiel
```



opendir, readdir, closedir

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

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



Beispiel: mini-"ls" (Ausgabe)

Ziel: So etwas...

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




Beispiel: mini-"ls" (1)

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

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

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

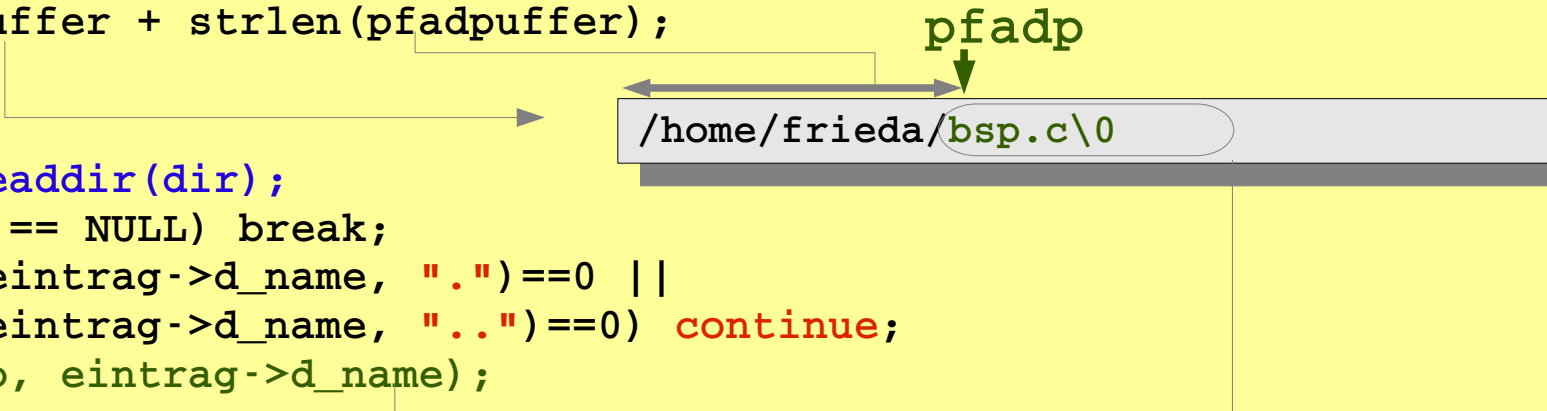
    dir = opendir(argv[1]);
    if (dir == NULL) {
        perror(argv[1]);
        exit(2);
    }
}
```

Beispiel: mini-"ls" (2)

```
strcpy(pfadpuffer, argv[1]);
strcat(pfadpuffer, "/");
pfadp = pfadpuffer + strlen(pfadpuffer);

while (1) {
    eintrag = readdir(dir);
    if (eintrag == NULL) break;
    if (strcmp(eintrag->d_name, ".")==0 ||
        strcmp(eintrag->d_name, "..")==0) continue;
    strcpy(pfadp, eintrag->d_name);

    if (stat(pfadpuffer, &statbuf) == -1) {
        perror(pfadpuffer);
    } else if (S_ISDIR(statbuf.st_mode)) {
        printf("[%s]\n", pfadpuffer);
    } else {
        printf("%s (%ld Bytes)\n", pfadpuffer, statbuf.st_size);
    }
}
closedir(dir);
return 0;
}
```

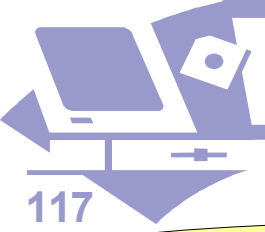


`S_ISDIR(m)` ist "wahr", wenn `m` der `st_mode`-Wert eines Verzeichnisses ist (siehe: `man stat`)

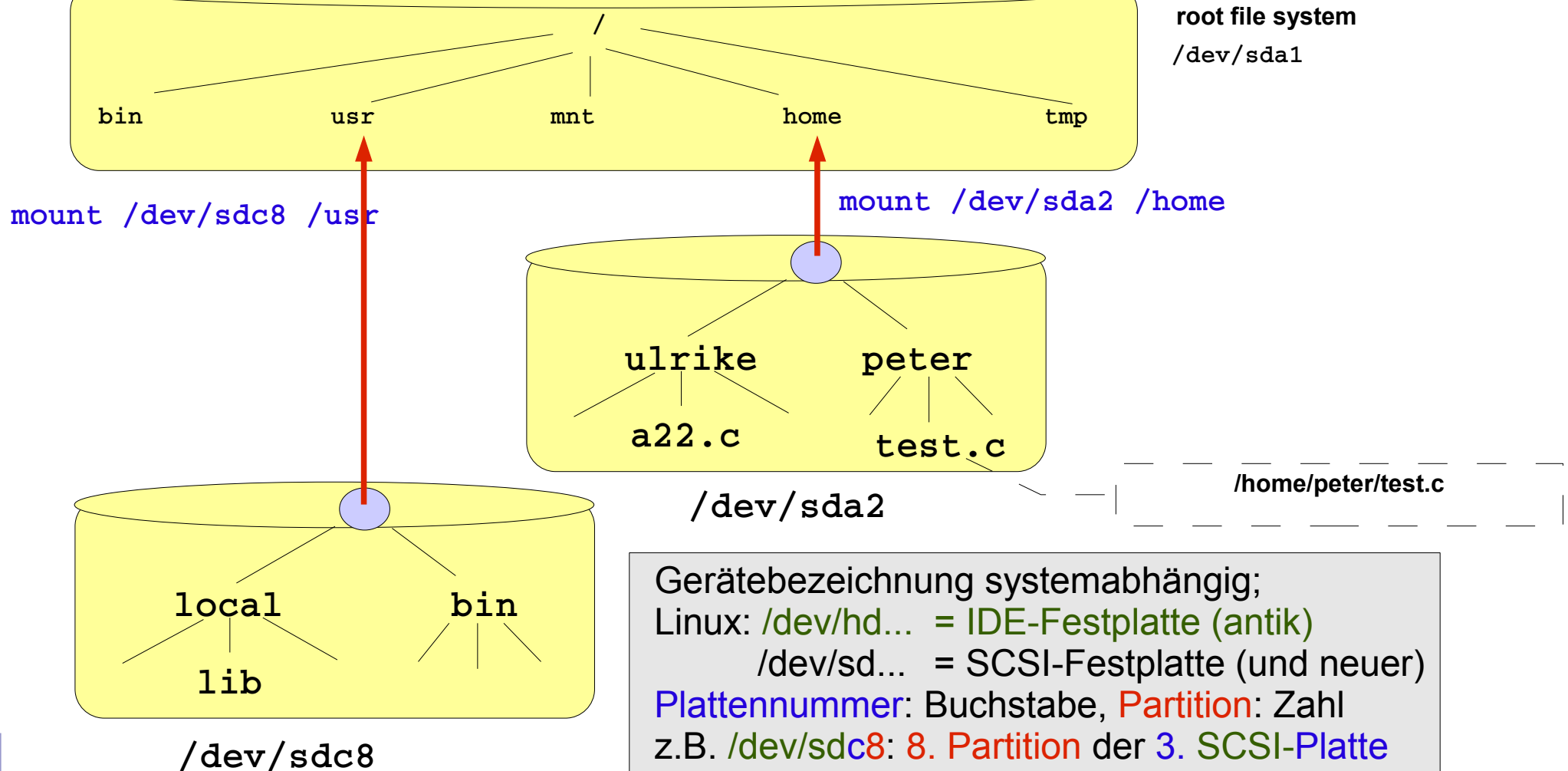


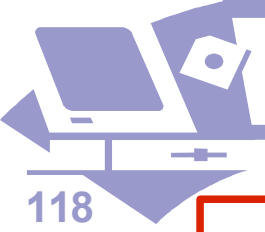
mount / umount

- ▶ UNIX: **Ein** Dateibaum, der sich evtl. über mehrere Speichersysteme (incl. vernetzter Speicher) erstreckt
 - **transparent** und **erweiterbar**: späteres Hinzufügen von Speichergeräten ohne Auswirkung auf Pfadnamen möglich
 - Kommandos (und gleichnamige UNIX-Systemfunktionen)
 - `mount` *gerät mountpoint*
z.B. `mount /dev/cdrom /mnt/cd`
`mount server17:/bin /usr/local/progs`
 - `umount` *mountpoint*
 - "*mountpoint*": Einhängepunkt (ein Verzeichnis), wo das Dateisystem von *gerät* eingehängt werden soll
 - Liste der beim Systemstart einzuhängender Geräte:
Datei `/etc/fstab` (file system table)
- ▶ Alternative: Speichergerätbezeichnung in Pfadnamen sichtbar (z.B. Windows Laufwerksbuchstaben "`C:\temp\test.c`")



Beispiel: mount





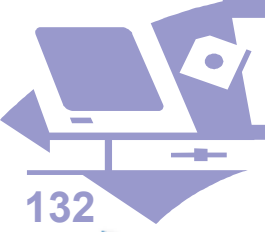
Beispiel: /etc/fstab

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

/dev/hda6	/	ext4	defaults	1	1
/dev/hda3	/boot	ext4	defaults	1	2
/dev/hda2	/home	ext4	defaults	1	2
none	/proc	proc	defaults	0	0
none	/dev/shm	tmpfs	defaults	0	0
/dev/hda4	swap	swap	defaults	0	0
/dev/hda1	/mnt/dos	vfat	defaults	0	0
/dev/sda1	/mnt/usb1	auto	noauto,owner,user	0	0
/dev/cdrom	/mnt/cdrom	iso9660	noauto,owner,ro	0	0

Dateisysteme (3)





Wahl der Blockgröße

- ▶ Fast alle Dateisysteme bilden Dateien aus gleich großen Plattenblöcken (z.B. bei Verwendung FAT, inodes, ...)
- ▶ **Plattenblock**: (zusammenhängende) Folge von Sektoren
 - z.B. ein Sektor (z.B. 512 Bytes), mehrere Sektoren, eine Spur, ein Zylinder, ...
- ▶ **Verwaltung** freier Blöcke z.B. mit
 - Bitmap (z.B. BSD UNIX FFS)
 - Verkettete Liste (z.B. MS-DOS, Unix System V)
- ▶ Was ist eine **gute Blockgröße**?
 - zu klein gewählt
 - hoher Verwaltungsaufwand
 - schlechte Performance (viele Kopfbewegungen)
 - zu gross gewählt
 - Platzverschwendung
- ▶ Gängige Größen:
 - verschiedene UNIXe: 1-4 kB (z.B. Linux: 4kB nicht unüblich)
 - MS-DOS: 512 Bytes - 32 kB, abhängig von Plattengröße

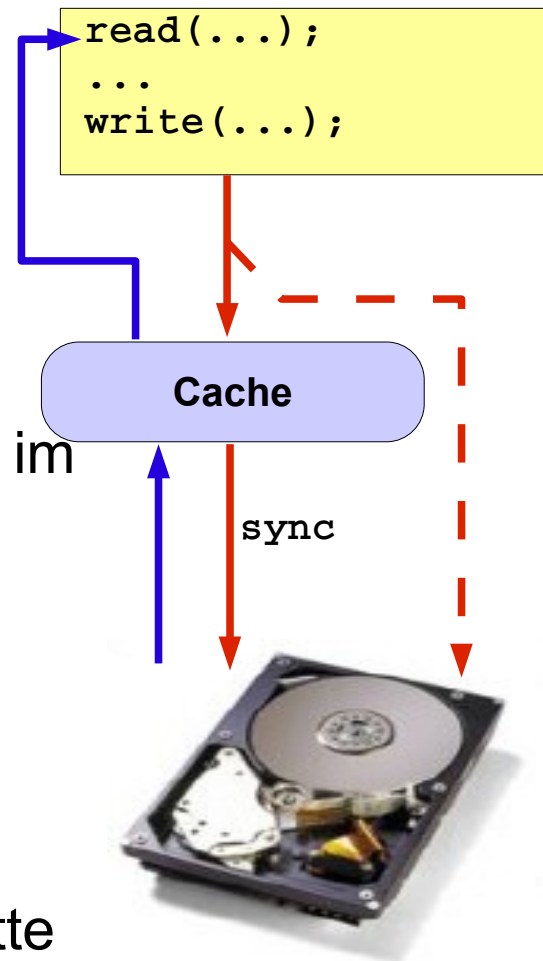
Performancesteigerung durch Caching

133

► Ziel: Plattenzugriffe (= langsam) vermeiden

► Block-Cache

- gewisse Anzahl von **Plattenblöcken** im Hauptspeicher **zwischenspeichern**
- Block-Zugriffe können dann (zum Teil) aus dem **schnellen** Hauptspeicher bedient werden
- Regelmäßiges Rückschreiben veränderter Blöcke im Cache auf die Platte (z.B. UNIX: Kommando `sync`)
- **Gefahr**: Datenverluste / Inkonsistenzen, wenn Cache-Inhalt nicht mehr mit Platte synchronisiert werden kann (z.B. durch Systemausfall)
- **Alternative**: Modifizierte Blöcke sofort auf die Platte schreiben ("*write through cache*")



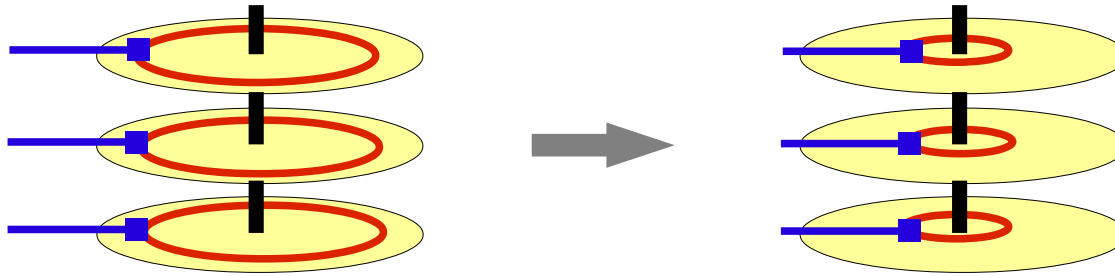


Vorauslesen von Blöcken

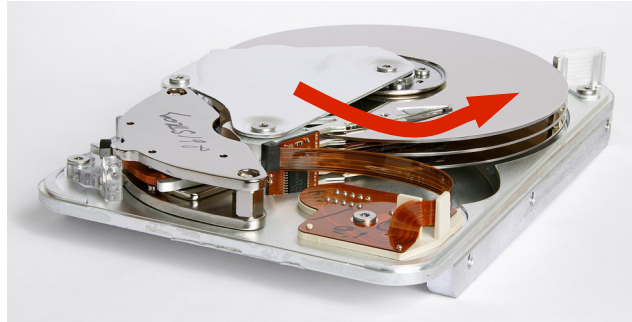
- ▶ Dateien werden **oft sequentiell** gelesen. Idee:
 - Wenn ein Block k einer Datei angefordert wird, wird geprüft, ob Block $k+1$ schon im Cache ist
 - Falls nein: Lesezugriff für Block $k+1$ erzeugen, um ihn schon einmal **im Voraus** in den Cache zu laden.
- ▶ **Nachteil:** Verschlechtert Performance bei *wahlfreiem* Zugriff auf die Datei
- ▶ **Lösungsansätze:**
 - **Statisch:** Zugriffsmodus (sequentiell / wahlfrei) beim Öffnen einer Datei festlegen lassen
 - **Dynamisch:** Betriebssystem hält "Zugriffs-Flag"
 - Zu Beginn und nach sequentielltem Lesen Flag setzen
 - bei seek-Operationen Flag löschen
 - abhängig vom Flag das Vorauslesen ein- bzw. ausschalten

Zeitaufwand für Blockzugriff

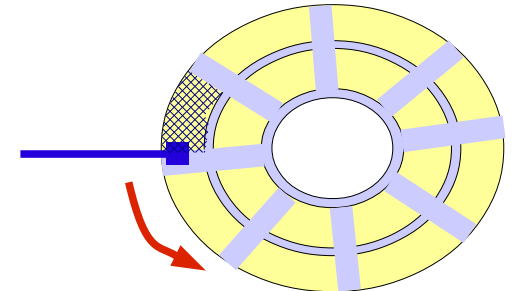
- **Positionierungszeit:** Arm über Ziel-Zylinder bewegen

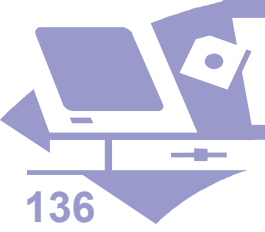


- **Rotationsverzögerung,** bis gesuchter Sektor unter Kopf ist



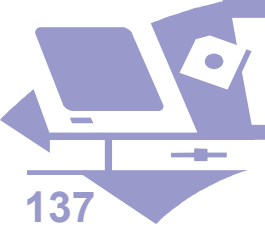
- Zeit zur **Datenübertragung** beim Auslesen





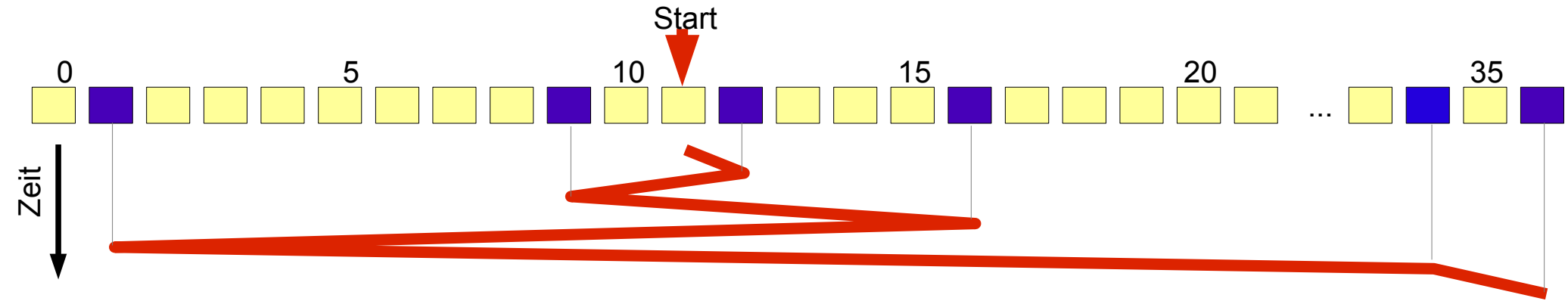
Planung der Armbewegung

- ▶ Einfache Strategie: Anforderungen für Plattenblöcke der Reihe nach abarbeiten (**FCFS** - *first come, first served*)
- ▶ Hoher zeitlicher Anteil für Kopfbewegungen, wenn z.B. mehrere Prozesse parallel auf verschiedene Dateien zugreifen (möglicherweise Positionierungs-Operation bei jedem Prozesswechsel)
→ oft **schlechte Performance**
- ▶ Idee: Während des Abarbeitens einer Anfrage **sammelt** der Treiber schon **Folgeanfragen** und **wählt** danach eine günstige aus ("Plattenarm-Scheduling")
- ▶ Positionierungszeit besonders "teuer"
- ▶ Ziel: Möglichst kleine durchschnittliche Positionierungszeit

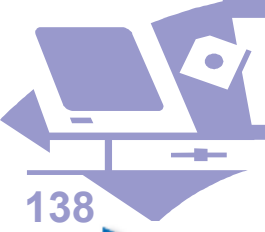


Shortest Seek First

- ▶ Abarbeiten desjenigen Folgeauftrags mit dem **geringsten Positionierungsweg** von der aktuellen Position aus.
- ▶ Beispiel: Anforderungen 1, 36, 16, 34, 9, 12



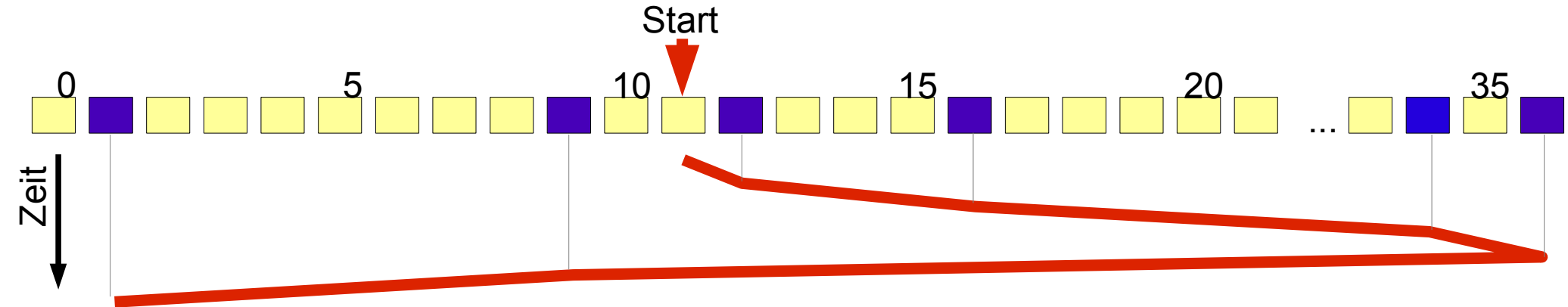
- ▶ Insgesamt $1 + 3 + 7 + 15 + 33 + 2 = \mathbf{61}$ **Zylinderwechsel**
- ▶ Zum Vergleich: FCFS benötigt bei gleicher Anforderungsfolge $10 + 35 + 20 + 18 + 25 + 3 = \mathbf{111}$
- ▶ **Problem: Mangelnde Fairness**; Verfahren tendiert zu "Mitte", Rand-Zylinder müssen oft länger warten



Aufzug-Verfahren

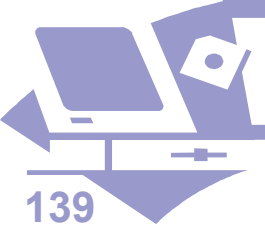
► "Aufzug-Verfahren":

- Kopfbewegung in eine Richtung, solange dort noch Aufträge sind;
- danach Richtungswechsel usw.



$$1 + 4 + 18 + 2 + 27 + 8 = 60 \text{ Zylinderwechsel}$$

Für *beliebige* Auftragsmenge maximal $2 \cdot \text{Zylinderzahl}$ Bewegungen.



Dateisystem-Konsistenz

- ▶ Wenn das System abstürzt, bevor alle modifizierten Blöcke zurückgeschrieben sind, kann es zu **Inkonsistenzen** im Dateisystem kommen (*deswegen auch PCs nicht im laufenden Betrieb ausschalten!*)
- ▶ Gilt insbesondere, wenn **Verwaltungsdaten** betroffen sind (z.B. inodes, Bitmap mit freien Blöcken, ...)
- ▶ Hilfsprogramme helfen, Inkonsistenzen zu entdecken und nach Möglichkeit zu beheben. Beispiele:
- ▶ Windows `scandisk`
- ▶ Unix `fsck` (file system check)
- ▶ Hilfsprogramme werden in der Regel automatisch beim Systemstart aufgerufen, wenn eine Inkonsistenz vermutet wird (oder eine bestimmte Zeit nicht geprüft wurde).



UNIX fsck: Blockprüfung

fsck führt verschiedene Konsistenzüberprüfungen durch:

Blocküberprüfung

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

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

erste Tabelle: wie oft tritt jeder **Block in (irgend)einer Datei** auf?

- alle inodes lesen
- für jeden verwendeten Block Zähler in Tab 1 aktualisieren

zweite Tabelle: **freie Blöcke**

- Für Blöcke in der Liste / Bitmap der freien Blöcke Zähler in Tab. 2 aktualisieren

Konsistenz: Für jeden Block ist Zählerstand aus Tab 1 und Tab 2 zusammen "1"



Blockprüfung: Fehler

► **Fehlender Block:** Block 4 ist weder belegt noch frei?

- Maßnahme: Block zu freien Blöcken hinzunehmen

0					5					10					15
1	1	0	1	0	1	0	0	0	0	0	1	1	0	1	0
0	0	1	0	0	0	1	1	1	1	1	0	0	1	0	1

belegte Blöcke
freie Blöcke

► Doppelter Block in Freiliste (Block 9)

- Maßnahme: Freiliste neu aufbauen

0					5					10					15
1	1	0	1	0	1	0	0	0	0	0	1	1	0	1	0
0	0	1	0	1	0	1	1	1	2	1	0	0	1	0	1

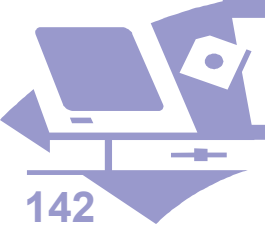
belegte Blöcke
freie Blöcke

► **Doppelter belegter Block (Block 12)**

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

0					5					10				15	
1	1	0	1	0	1	0	0	0	0	0	1	2	0	1	0
0	0	1	0	1	0	1	1	1	1	1	0	0	1	0	1

belegte Blöcke
freie Blöcke



fsck Verzeichnisprüfung

- ▶ Darüber hinaus Überprüfung der Verzeichniseinträge:
 - Vergleiche **Anzahl aller Verzeichniseinträge** mit Verweis auf einen inode mit dem darin gespeicherten Referenzzähler
 - ggf. **inode-Referenzzähler** der durch Zählung festgestellten Zahl von Referenzen anpassen
- ▶ Hinweis: Es kann "beliebig viele" (>0) Referenzen auf einen inode geben (\rightarrow harte Links!)
- ▶ Problem: Für große Platten kann ein `fsck`-Lauf sehr lange (Stunden!) dauern. In dieser Zeit ist das System möglicherweise nicht verfügbar (Kosten!)



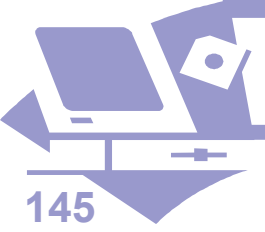
Journaling File Systems

- ▶ Neuere Dateisysteme können Änderung **atomar** durchführen von ("alles oder nichts")
- ▶ **Transaktionskonzept** (vgl: Datenbanken)
- ▶ Häufig beschränkt auf Metadaten (Verzeichnisse, Bitmaps/Freilisten, inodes; nicht: Benutzer-Datei*inhalte*)
- ▶ **Änderung von Metadaten** erfolgt in zwei Schritten:
 - geplanten Änderungen in eine **Journal**-Datei schreiben
 - erst dann Änderungen an Metadaten tatsächlich ausführen
- ▶ Falls ein Systemausfall zwischen den Schritten auftritt, wird nach Neustart ein **journal replay** ausgeführt: Die im Journal verzeichneten Änderungen werden ausgeführt.
- ▶ Beispiele (Linux): ext3/4, btrfs, XFS; Windows NTFS



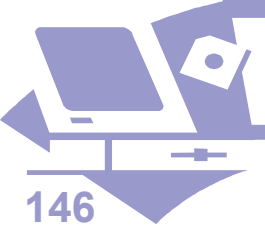
Plattenfehler

- ▶ Herstellungsfehler nicht auszuschließen, fast jede Festplatte hat **defekte Sektoren**
- ▶ Defekte Sektoren werden beim Formatieren der Platte entdeckt und in eine **Defektliste** eingetragen
- ▶ Behandlung defekter Sektoren:
 - **durch Plattencontroller** (Hardware):
 - Platte führt **Defektliste**
 - Automatische Verwendung eines Ersatz-Sektors bei Zugriff auf defekten Sektor
 - **durch Betriebssystem**
 - Bei Entdecken eines defekten Sektors bei einem Blockzugriff
 - „**Datei aus defekten Blöcken**“ konstruieren.
Die Datei wird nie verwendet (gelesen oder geschrieben).



Sicherungskopien: Wozu?

- ▶ **"Desaster Recovery":** Platteninhalten wiederherstellen nach Ereignissen wie...
 - Plattencrash
 - Feuer, Überschwemmung
 - Sabotage
 - Computerviren
 - ...
- ▶ **Benutzerfehler:** Dateiverluste...
 - nach irrtümlichem Löschen von Dateien
 - durch Fehler während der Programmentwicklung
 - ...
- ▶ Sicherungskopien sollten in geeigneter Entfernung sicher verwahrt werden (...damit sie z.B. nicht mit dem gesicherten Plattensystem verbrennen) und
- ▶ mehrere Stände (z.B. die letzten 8 Wochen) umfassen



Sicherungskopien: wie?

- ▶ Sicherung dauert ggf. lange (insb. bei Magnetband, CD, ...)
- ▶ Daher: planen, welche Dateien sicherungswürdig sind
- ▶ **Vollsicherung:** alle (geplanten) Dateien sichern
- ▶ **Inkrementelle Sicherung:** Nur seit letzter Sicherung geänderte Dateien sichern
 - Beispiel:
 - monatlich Vollsicherung
 - wöchentlich inkrementelle Sicherung
 - schneller, (da) in der Regel weniger Volumen pro Lauf
 - Wiederherstellung: neueste Vollsicherung und der Reihe nach folgende inkrementelle Sicherungen einspielen
- ▶ Problem: aktive, in Verwendung befindliche Dateisysteme
 - Sicherung nachts / an Wochenenden
 - Moderneres Volume-Management (z.B. LVM, s.u.)



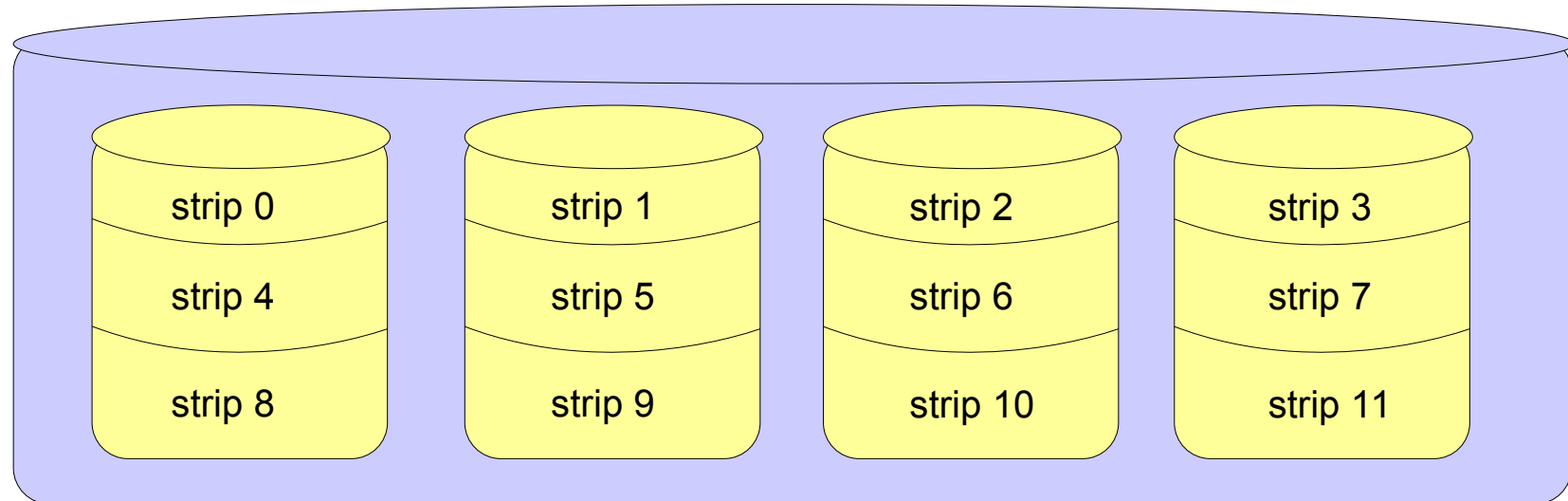
RAID

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



RAID 0 - "striping"

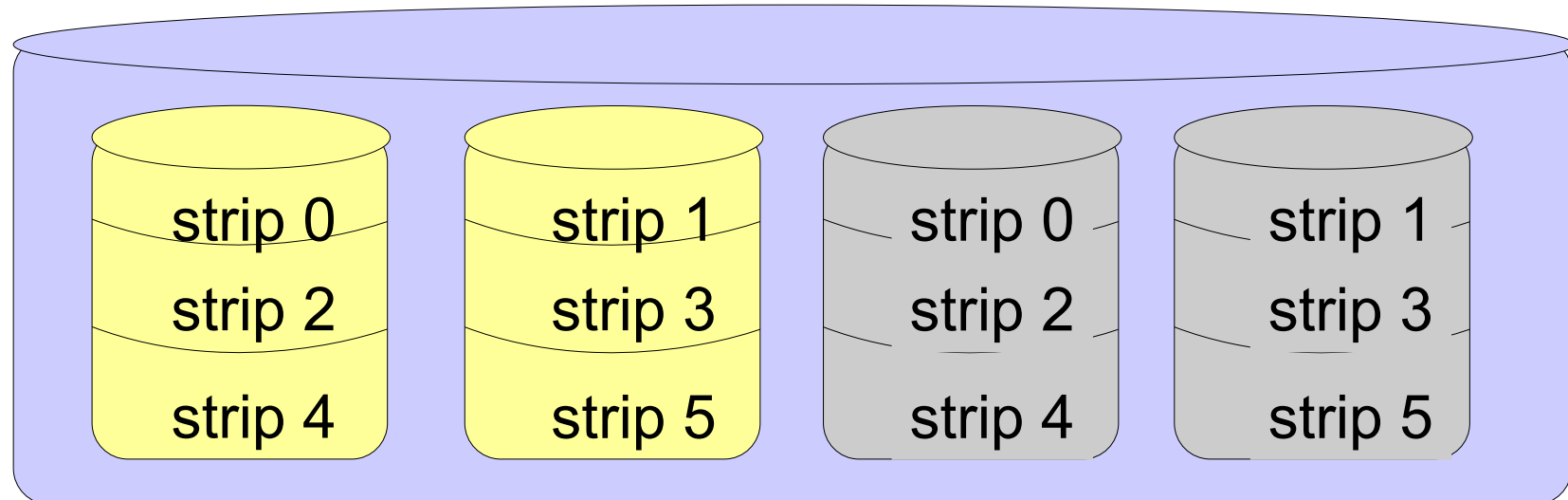
- ▶ RAID-Platte wird in Streifen mit k Sektoren eingeteilt
- ▶ Streifen werden reihum auf den angeschlossenen Platten abgelegt.
- ▶ keine Redundanz, damit keine höhere Fehlertoleranz
- ▶ Schneller Zugriff besonders bei großen Dateien, da Platten parallel arbeiten können
- ▶ RAID-Kapazität: Summe der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





RAID 1 - "mirroring"

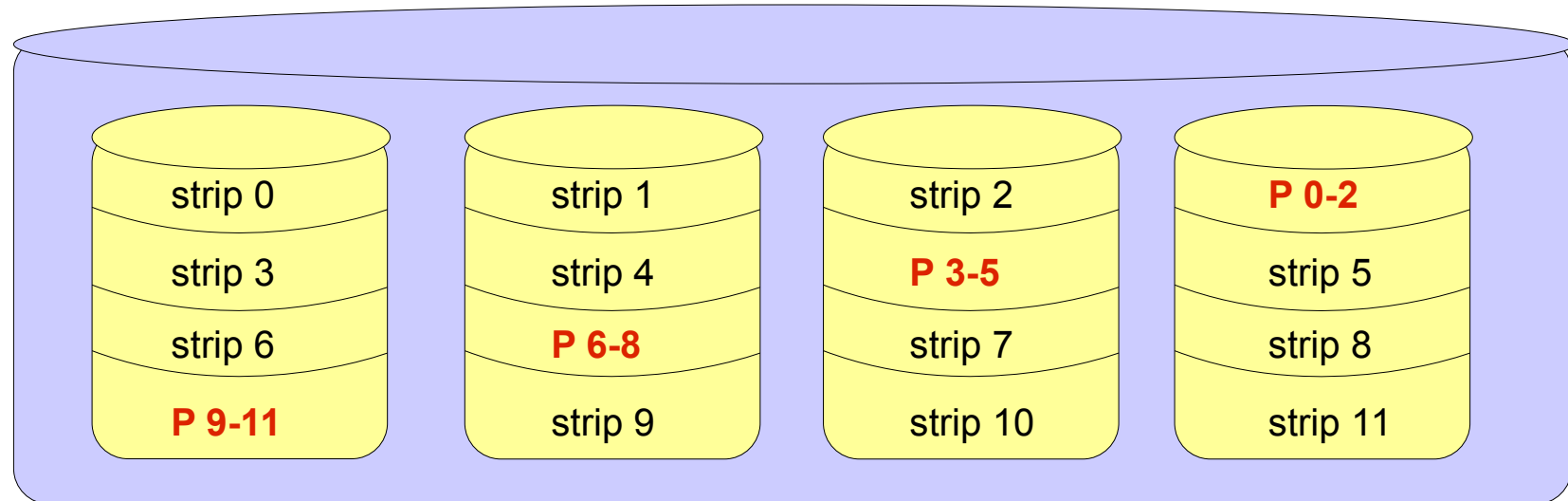
- ▶ Zu jeder Platte gibt es eine Spiegelplatte gleichen Inhalts
- ▶ Fehlertoleranz: Wenn eine Platte ausfällt, kann andere sofort einspringen (übernimmt Controller automatisch)
- ▶ Schreiben: etwas langsamer; Lesen: schneller durch Parallelzugriff auf beide zuständigen Platten
- ▶ Kapazität: Hälfte der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





RAID 5

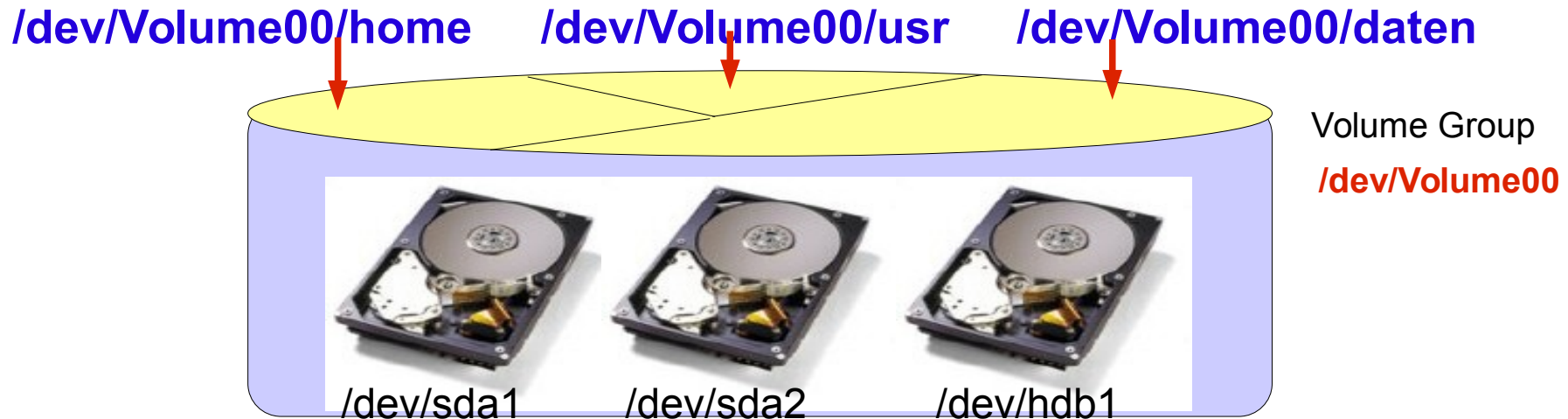
- ▶ Paritätsinformation auf alle Platten verteilt.
- ▶ Beispiel: **P 0-2** enthält XOR-Verknüpfung über die Streifen 0, 1, 2
- ▶ Inhalt jeder beliebigen Platte kann mit Hilfe der Inhalte aller übrigen (im laufenden Betrieb) rekonstruiert werden (wieder per XOR)
- ▶ Fehlertoleranz und gute Kapazitätsnutzung;
Leseoperationen schnell; Schreiben aufwendiger

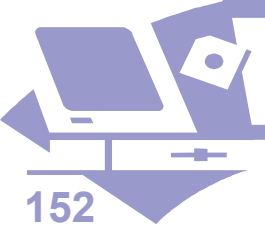


Logical Volume Manager

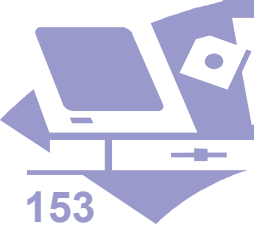
151

- ▶ Bestandteil vieler UNIX-Systeme;
hier betrachtet: Linux-Version LVM
- ▶ Physische Speichergeräte (*physical volumes*) werden zu **Laufwerksgruppen** (*volume groups*) zusammengefaßt
- ▶ Auf einer Laufwerksgruppe können **logische Laufwerke** eingerichtet (entspricht Partitionierung) und mit einem Dateisystem versehen werden.



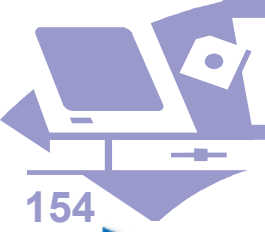


- ▶ Im laufenden Betrieb (!) ...
 - kann die Kapazität der Laufwerksgruppe durch **Hinzufügen weiterer physische Volumes** vergrößert werden
 - können **Daten** von alten Platten auf neue **verlagert** und die alten Platten außer Betrieb genommen werden
 - kann logischen Laufwerken mehr **Speicherplatz zugeordnet** werden oder Speicherplatz **entzogen** werden.



► LVM unterstützt "Filesystem Snapshots"

- Beim Anlegen eines Snapshots wird ein neues logisches Laufwerk angelegt, das den **momentanen Zustand** seines zugehörigen Ursprungs-Laufwerks enthält (eingefrorene Sicht, keine Kopie)
- Ermöglicht **konsistente Backups** über Snapshot-Laufwerk **trotz weiterlaufenden Betriebs** auf dem ursprünglichen Laufwerk

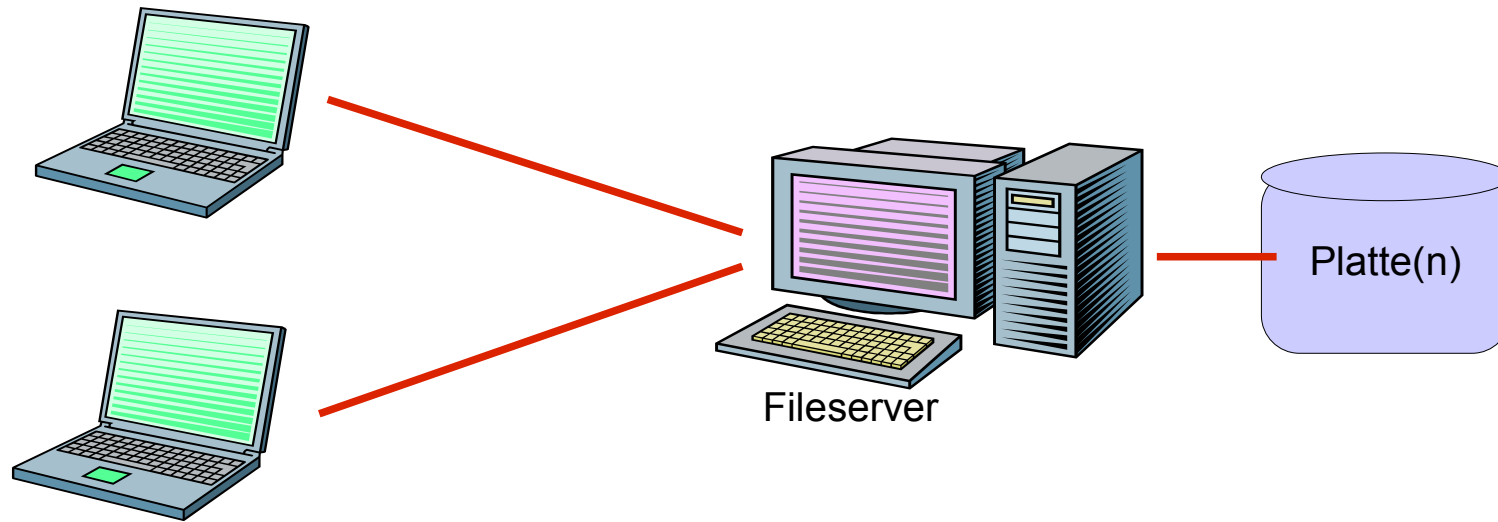


Fileserver-zentriertes Speichermgmt

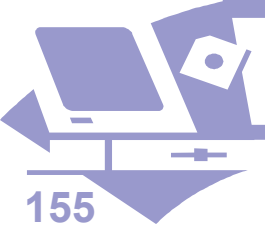
154

► Traditionell: Server-zentrierte Architektur:

- Server mit direkt angeschlossenen Platten
- Zugriff ausschließlich über diesen Server via Netzwerk



- Verfügbarkeit bei Serverausfall? Backup?
- Performance bei massivem Zugriff auf gemeinsame Daten durch (viele) andere IT-Systeme?
- Management?

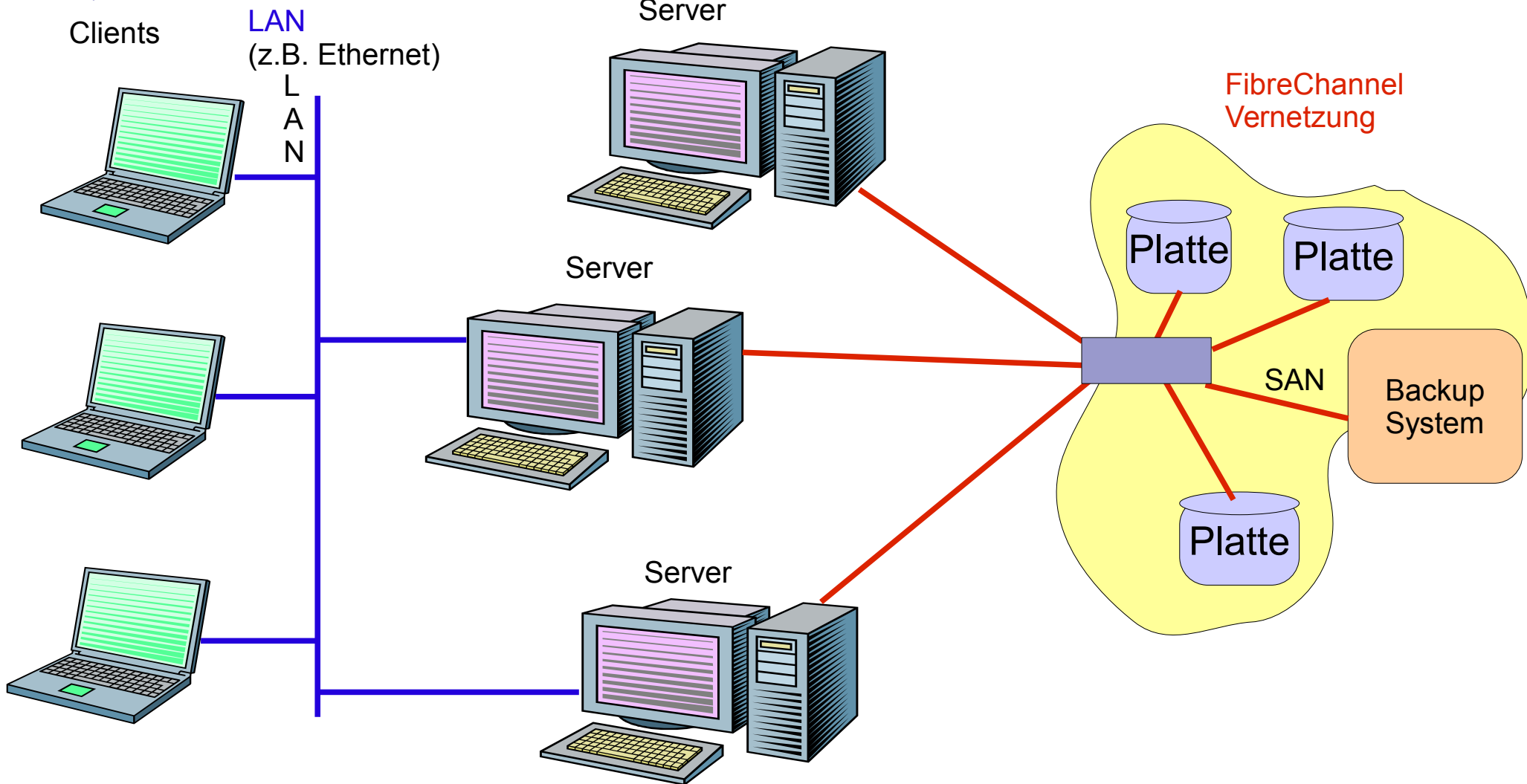


SAN: Storage Area Network

- ▶ Eigenes **Netzwerk für Speicherressourcen**
- ▶ Verbindung der Geräte über schnelles Netz
 - "Fibre Channel" (FC) auf Glasfaser oder Kupferkabel, ~400 MBytes/s bidirektional, bis zu 10 km ohne weitere Geräte überbrückbar; Tunneln über IP-Verbindung möglich
 - Es gibt auch IP-basierte Protokoll-Alternativen (iSCSI, FCIP)
- ▶ **Speicher-zentrierte** Architektur
- ▶ Gleichzeitiger Zugriff von **mehreren Rechnern** aus
- ▶ Rechner sieht Speicher wie "gewöhnlichen" Plattenspeicher (**Block-Device**, kein Fileserver).
- ▶ Snapshot-Möglichkeit ("instant copy")
- ▶ "LAN-free / Server-free **Backup**"
- ▶ zentrales **Management: z.B.** Namensdienst, Verwendung von XML, HTTP für Konfiguration

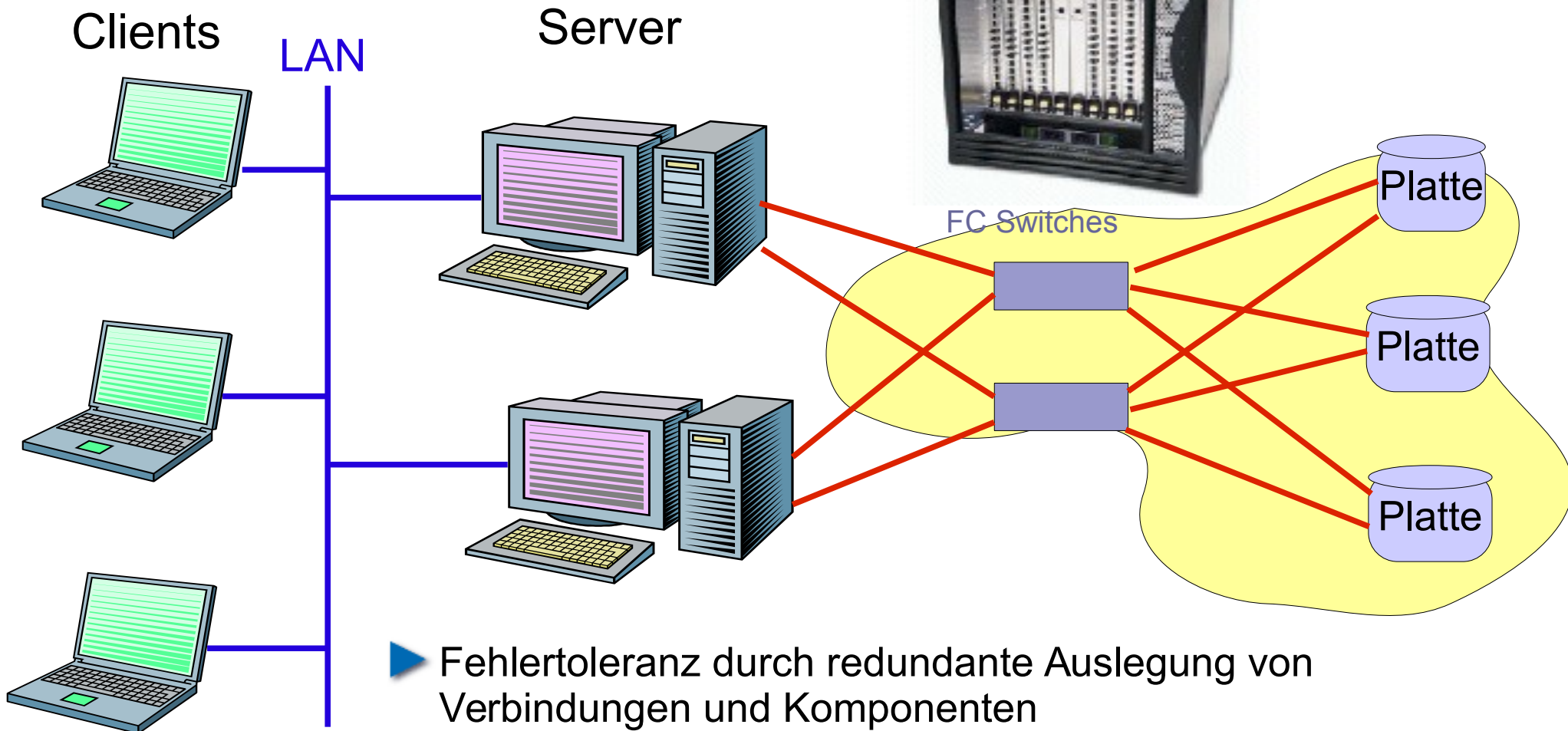
SAN: Beispiel

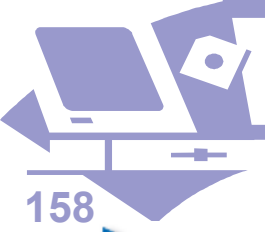
156



SAN: Hochverfügbarkeit

157

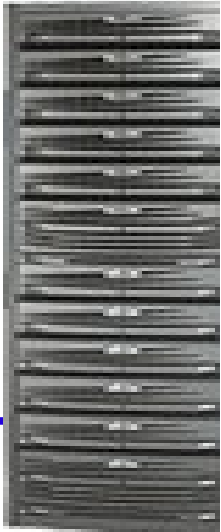
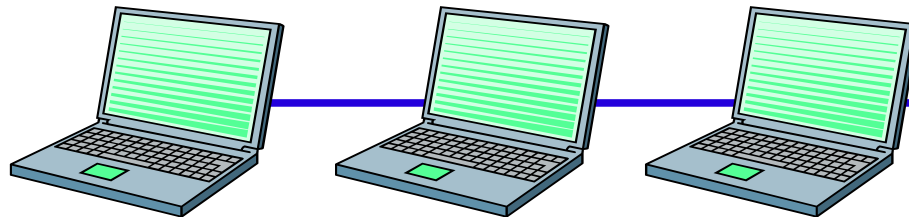




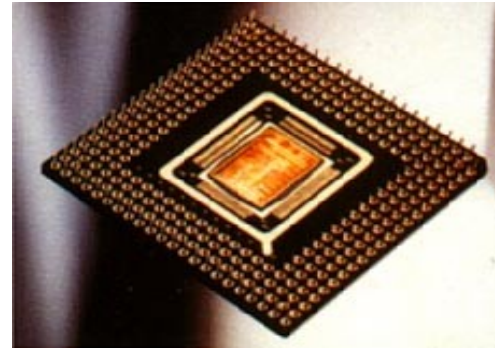
158

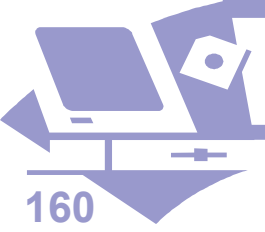
NAS: Network Attached Storage

- ▶ "Vorkonfigurierter, optimierter File-Server"
- ▶ Spezialsoftware für Snapshots
- ▶ Replikationsmöglichkeit mit zweitem NAS-Server (Ausfallsicherheit)
- ▶ Kein Gegensatz: NAS kann auf SAN aufsetzen
- ▶ leichter zu verwenden als SAN
 - SAN sieht für Server wie Blockgerät aus ("Festplatte mit SCSI-Schnittstelle")
 - NAS bietet gleich höhere Protokolle wie HTTP, CIFS, NFS, ... an



Prozessverwaltung (1)





Prozessmodell

► Prozess:

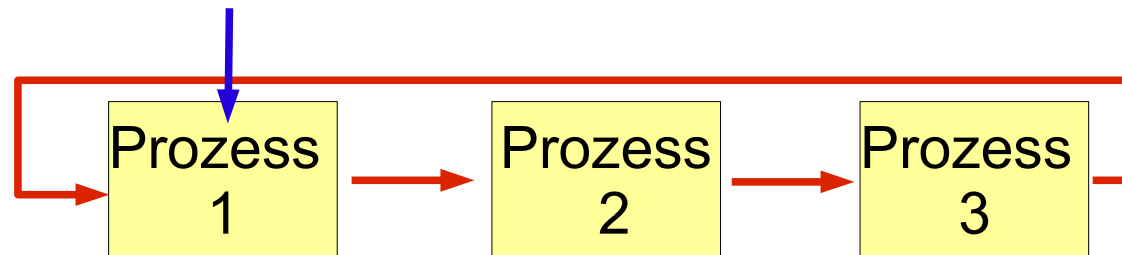
"Programm in Ausführung",
schließt *Kontextinformationen* ein, wie etwa

- aktueller Wert des CPU-Befehlszählers,
- CPU-Registerinhalte, Speicher (→ Variablenbelegungen etc)

► Jeder Prozess hat konzeptionell eigene "virtuelle CPU":

- echte CPU schaltet zwischen Prozessen hin- und her ("Kontextwechsel")
- "Multiprogrammierung", "Multitasking"

► Echte Parallelverarbeitung setzt mehrere CPUs voraus (Multiprozessorsysteme)





Programm und Prozess

► Programm:

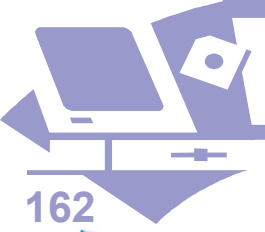
- feststehende Beschreibung eines Algorithmus

► Prozess:

- "Aktivität"
- Programm plus Ausführungskontext
- Mehrere Prozesse können sich eine CPU teilen

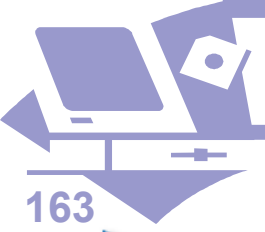
► Daher:

- ...kann ein Programm in mehreren Prozessen quasi gleichzeitig ausgeführt werden
- ...ist die Ausführungszeit bei Programmierung kaum reproduzierbar (hängt u.a. von der Anzahl der in laufenden Prozesse und deren Verhalten ab)



Prozesserzeugung

- ▶ Einfachster Fall: **Feste Menge** von Prozessen wird beim Systemstart erzeugt (z.B. Videorekorder-Steuerung)
- ▶ Bei komplexeren Systemen werden neue Prozesse im Laufe der Zeit dynamisch erzeugt, z.B.
 - beim **Systemstart**
 - z.B. UNIX-Daemons: Hintergrundprozesse zum Annehmen von E-Mail, Druckjobs, Web-Anfragen, ...
 - durch **andere Prozesse** per Systemfunktion (z.B. "Hilfsprozess" erzeugen)
 - durch den **Benutzer** veranlasst
 - z.B. Programmstart: "Prozesserzeugung per Doppelklick"
 - zur Abarbeitung von **Batch-Jobs** (auf Großrechnern)



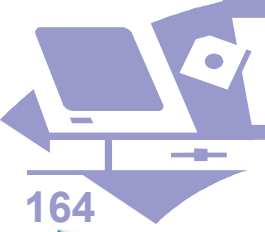
Prozessende

► **Freiwilliges** Prozessende (Prozess beendet *sich selbst*)

- Normale Beendigung
 - Prozess ist "normal" durchgelaufen
- Beendigung aufgrund eines Fehlers
 - z.B. angegebene Datei kann nicht geöffnet werden, Programm sieht Ausgabe einer Fehlermeldung und geordnetes Prozessende vor

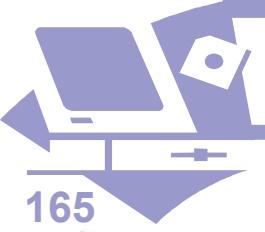
► **Unfreiwilliges** Prozessende (Prozess *wird* beendet)

- Beendigung aufgrund eines schweren Fehlers, z.B.
 - Zugriff auf unzulässige Speicheradresse
 - Division durch Null
- Beendigung durch anderen Prozess
 - ein anderer Prozess hat mit Hilfe einer Systemfunktion das Betriebssystem überzeugt, den Prozess abzurechnen.



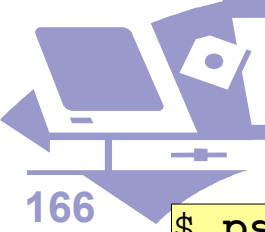
Prozesshierarchie

- ▶ Manche Systeme merken sich Zusammenhang zwischen erzeugendem Prozess (Vaterprozess) und von diesem erzeugtem Prozess (Sohn-/Kindprozess)
- ▶ **Prozessfamilie**: Prozess und alle seine Nachkommen
- ▶ **Prozesshierarchie**: Baum-strukturierte Prozess-Menge (z.B. UNIX)
- ▶ Gegenbeispiel Windows:
 - keine Hierarchie,
 - alle Prozesse sind gleichwertig,
 - erzeugender Prozess erhält Verweis ("Handle") auf erzeugten Prozess,
 - dieses Handle kann er jedoch beliebig weitergeben (→ nicht notwendig Baumstruktur)



Beispiel: UNIX Systemstart (klassisch)

- ▶ Beim UNIX-Systemstart wird der Prozess `init` (Prozess-Nr. 1) erzeugt (= Vater aller nachfolg. Prozesse)
- ▶ Neuere Alternative z.B. bei aktuellen Linuxen: `systemd`
- ▶ `init` ...
 - liest die Bezeichnungen der angeschlossenen Terminals und die Pfade zu den zu startenden Anmelde-Programmen aus der Datei `/etc/inittab` und
 - startet jeweils einen Prozess zur Benutzeranmeldung
- ▶ Meldet sich ein Benutzer an, wird für ihn ein Shell-Prozess erzeugt, der seinerseits bei Kommandoeingaben entsprechende Unterprozesse erzeugt usw.
- ▶ UNIX-Kommandos zur Ausgabe der Prozessliste:
 - `ps` Standard-Kommando
 - `pstree` baum-formatierte Ausgabe (nicht überall verfügbar)



ps - Prozessliste ausgeben

```
$ ps -ef
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
weitz	30297	30296	0	16:23	pts/1	00:00:00	-bash
weitz	30376	30297	0	16:35	pts/1	00:00:00	ps -ef
postgres	19678	1	0	Apr23	?	00:00:01	/opt/pgsql/bin/postmaster
postgres	19680	19678	0	Apr23	?	00:00:00	postgres: stats buffer pr
postgres	19681	19680	0	Apr23	?	00:00:00	postgres: stats collector
root	22077	1	0	Apr25	?	00:00:00	/usr/sbin/inetd
wwwrun	5043	5042	0	Apr10	?	00:00:00	/usr/sbin/fcgi -f /etc/h
wwwrun	5044	5042	0	Apr10	?	00:00:09	/usr/sbin/httpd -f /etc/h
jlude001	20472	1	0	Apr03	?	00:09:19	kdeinit: kded
fherm001	2645	1	0	Apr09	?	00:00:09	kdeinit: dcopserver --nos
root	5998	1	0	2017	tty6	00:00:00	/sbin/mingetty tty6
wstad001	11166	1	0	2017	?	00:00:00	ftpd: p5081251E.dip0.t-ip
mgraf001	14027	1	0	2017	?	00:00:00	ftpd: p5081346A.dip.t-dia

- ▶ UID : UserID (Benutzername)
- ▶ PID: Process ID; PPID: Parent Process ID
- ▶ Beispiel: `ps -ef` (Prozess 30376) ist Sohn von 30297 (bash-Shell)
- ▶ BSD-UNIXe: andere Optionen, z.B. `ps aux` (mehr dazu: `man ps`)

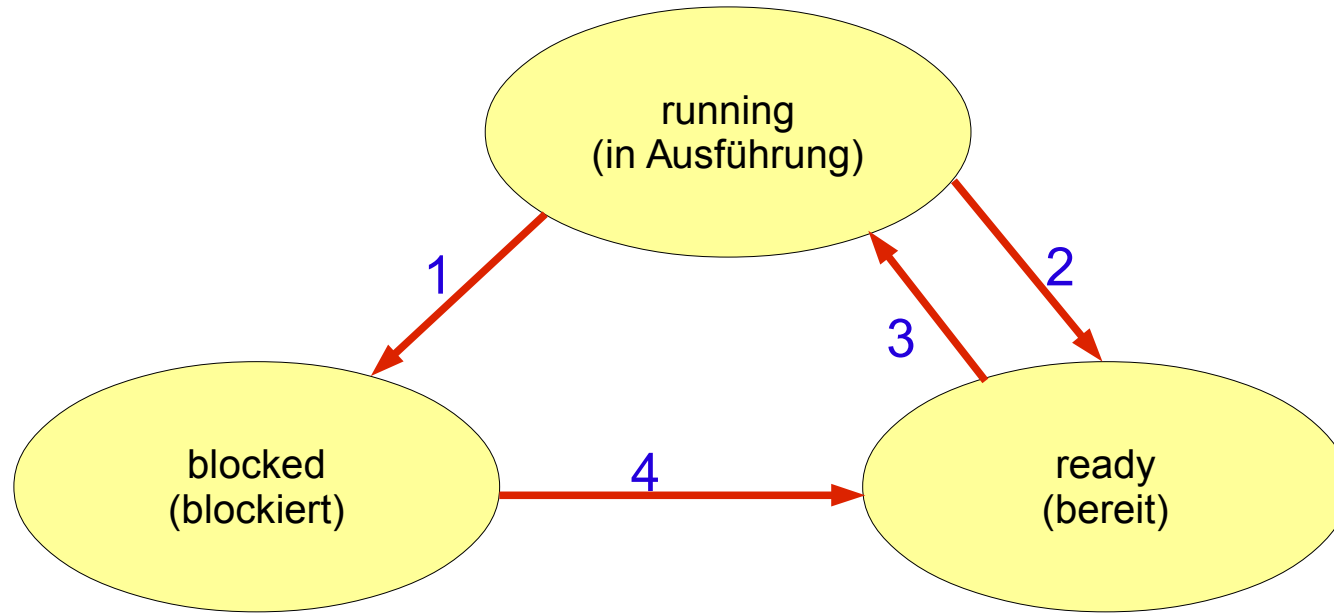


pstree - Prozessbaum ausgeben

```
$ pstree -aup
init(1)
|-atd(372)
|-(bdflush,7)
|-cron(515)
|-httpd(5042) -f /etc/httpd/httpd.conf
|   |-httpd(5043) -f /etc/httpd/httpd.conf
|   |-httpd(5044) -f /etc/httpd/httpd.conf
|-postmaster(19678,postgres) -i -D /opt/pgsql/data
|   `--postmaster(19680)
|       `--postmaster(19681)
|-sshd(328)
|   `--sshd(30536)
|       `--bash(30537,weitz)
|           `--pstree(30559) -aup
|-syslogd(341) -a /chroot/dev/log
...
```

- ▶ Option -a alle Prozesse zeigen
- ▶ Option -u Benutzer (*user*) ausgeben (falls nicht root)
- ▶ Option -p Prozessnummer ausgeben

Prozesszustände (vereinfacht)

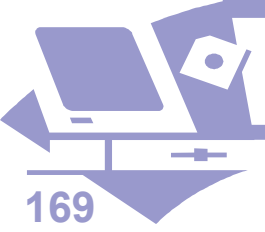


1 (running→blocked): Prozess muss warten, z.B. auf Eingabe

2 (running→ready): Prozess bekommt CPU entzogen

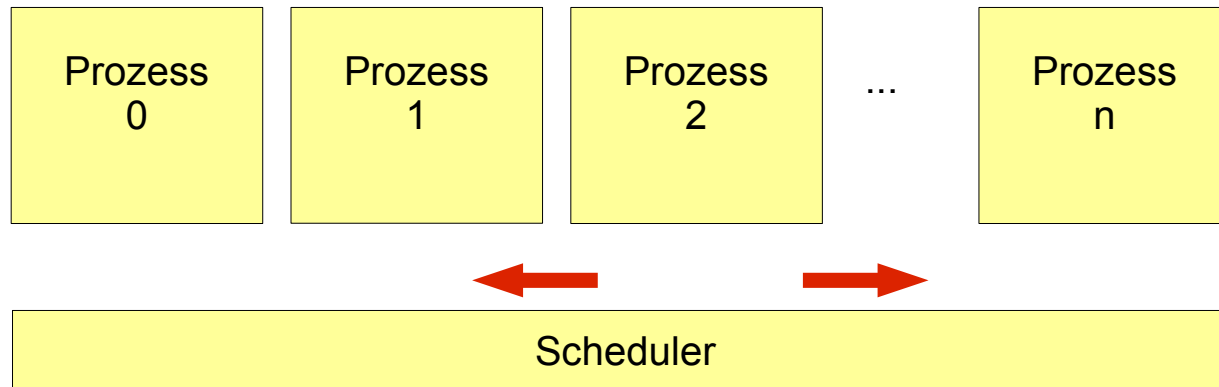
3 (ready→running): Prozess erhält CPU zugeteilt

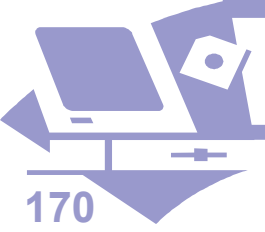
4 (blocked→ready): z.B. erwartete Eingabe liegt an



Scheduler

- ▶ Der **Scheduler** ist der Teil des Betriebssystems, der für das Umschalten zwischen den Prozessen und damit die CPU-Zuteilung zuständig ist.
- ▶ Dazu implementiert er einen **Scheduling-Algorithmus**, der bestimmt welcher der bereiten Prozesse wie lange die CPU erhält.
- ▶ Er gehört damit zu den untersten Schichten eines Betriebssystems.





Wer aktiviert den Scheduler?

- ▶ Wenn der Scheduler die Kontrolle an einen ausgewählten Prozess abgibt
- wie bekommt er sie dann wieder zurück?
- ▶ Ein Ansatz: Jedes Programm führt "oft genug" einen Systemaufruf aus, um Abgabe der Kontrolle anzubieten.
 - **"kooperatives Multitasking"**
 - z.B. in früheren Windows- und MacOS-Versionen
 - Nachteil: Ein Programm kann nicht gezwungen werden, Kontrolle abzugeben; Problem bei "bösen" Programmen.
- ▶ Alternative: **Preemptives Multitasking**
 - benötigt Hardware-Unterstützung
 - Bei Eintreten bestimmter Ereignisse (Ein-/Ausgabe, Ablauf eines Timers, ...) wird gerade laufender Prozess "von außen" unterbrochen und Code zur Unterbrechungs-Verarbeitung aufgerufen
 - hierbei kann Aufruf des Schedulers vorgesehen werden



Prozesstabelle

Die Prozesstabelle (Prozesskontrollblock, PCB) enthält Verwaltungs- und Kontextinformationen zu jedem Prozess (ein Eintrag je Prozess), z.B.

Prozessmanagement

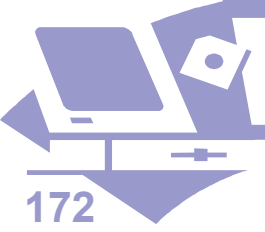
Prozessorregister
Statusregister
Stack-Zeiger
Befehlszähler
Priorität des Prozesses
Prozess ID
Vaterprozess
eingegangene Signale
Startzeit,
verbrauchte CPU-Zeit
...

Speichermanagement

Zeiger auf Segmente f.
- Text
- Daten
- Stack
...

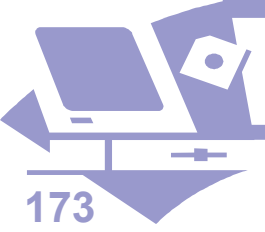
Dateiverwaltung

Wurzelverzeichnis
Arbeitsverzeichnis
Dateideskriptoren
 offener Dateien
Benutzer-ID
Gruppen-ID
...

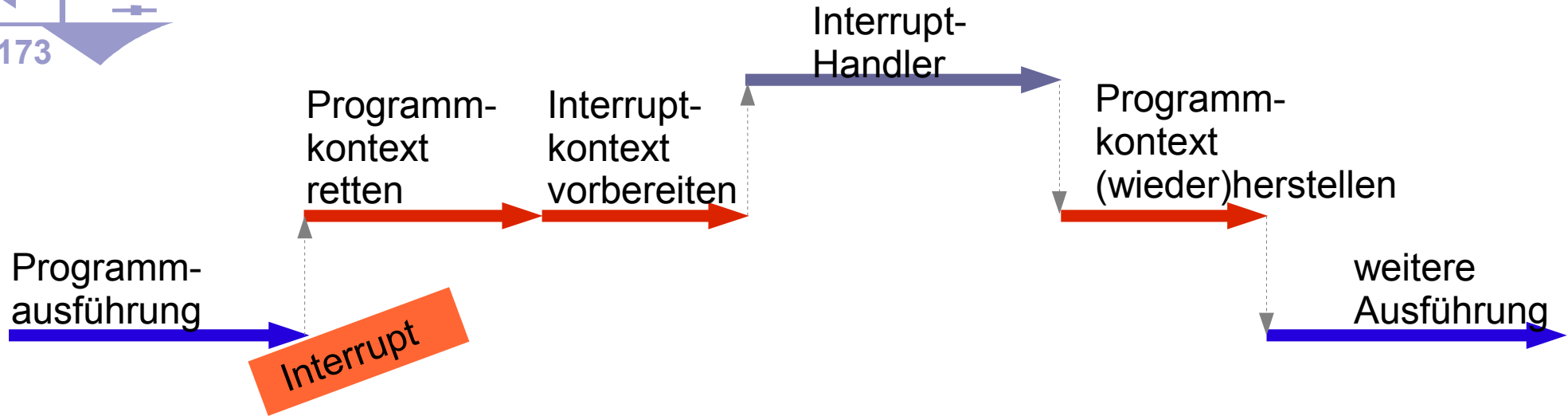


Unterbrechungen

- ▶ Ein-/Ausgabegeräte können Unterbrechung der normalen CPU-Arbeit auslösen (→ **interrupt**), z.B. Festplattencontroller, Hardware-Timer, Terminals, ...
- ▶ Je Klasse von E/A-Geräten gibt es einen Zeiger (**Interrupt-Vektor**), der auf Programmcode zur Handhabung des Interrupts verweist
- ▶ Unterbrechungen können auch durch ein Programm ausgelöst werden (→ **traps**),
 - bei Fehlern (etwa Division durch Null)
 - absichtlich durch spezielle Maschineninstruktion
- ▶ **Maskierbare Unterbrechung** (*maskable interrupt*): Reaktion auf eine solche Unterbrechung kann per Software (Prozessor-Flag setzen) unterbunden werden
- ▶ Gegenstück: **Nicht-maskierbare Unterbrechung** (non-maskable interr.)



Ablauf Interrupt-Behandlung

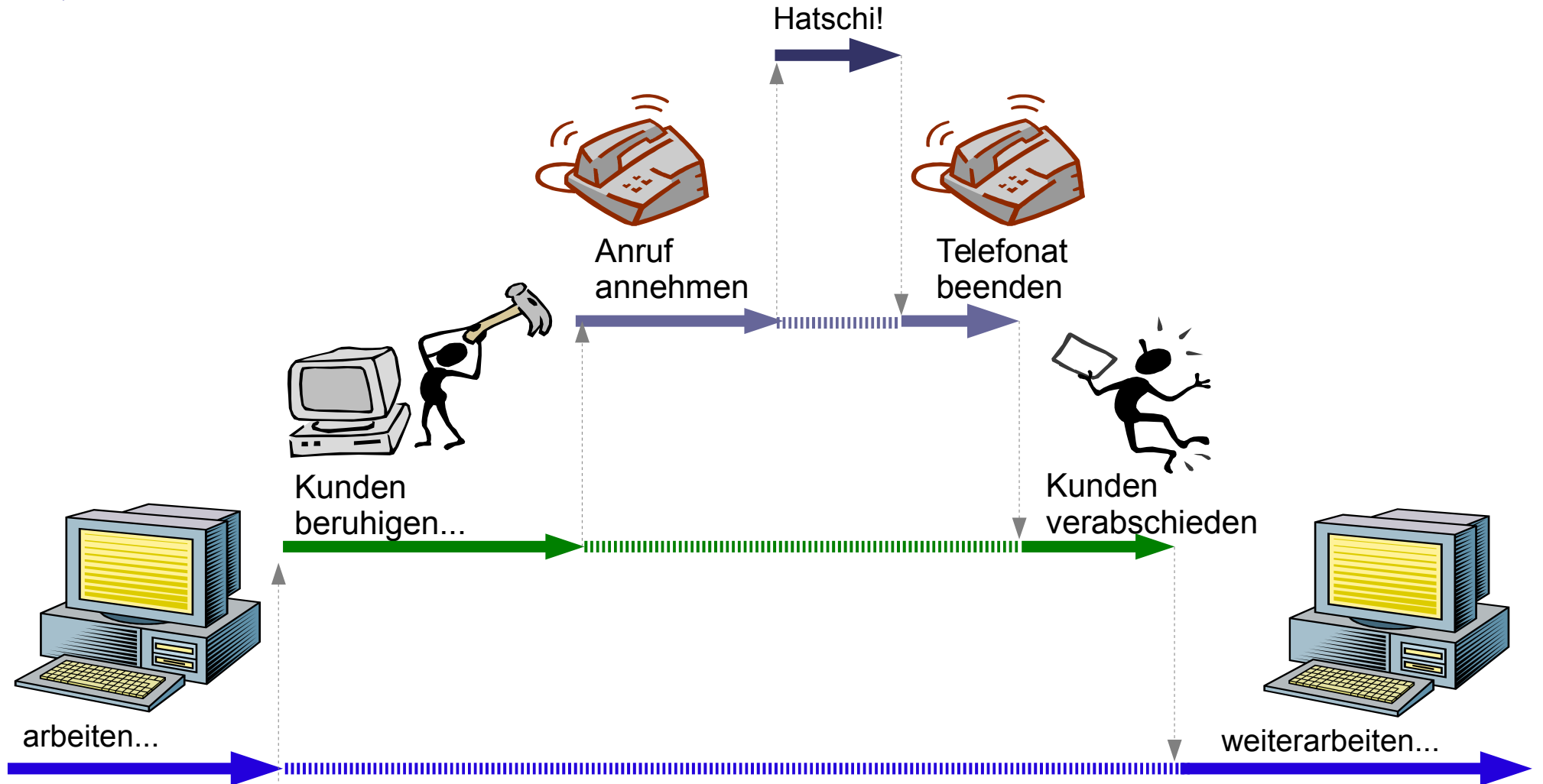


► Typische Reaktion beim Auftreten eines Interrupts:

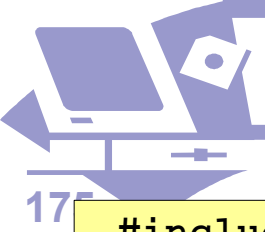
- Befehlszähler und andere Register werden gesichert
- Befehlszähler wird auf Wert des zugehörigen Interrupt-Vektors gesetzt (Interrupt-Handler, oft eine Assemblerfunktion)
- Interrupt-Handler sichert weitere Prozessor-Register,
- ruft ggf. weitere Funktionen auf, die "inhaltlich" auf Interrupt reagiert (z.B. Terminal-Eingabe auslesen)
- Scheduler sucht nächsten Prozess
- nächster Prozess wird gestartet

Geschachtelte Interrupts

174



(vgl. Vogt 2001)



UNIX-Prozesserzeugung: fork()

```
#include <unistd.h>
pid_t fork(void);
pid_t getpid(void);
pid_t getppid(void);
```

► Die Systemfunktion `fork()`

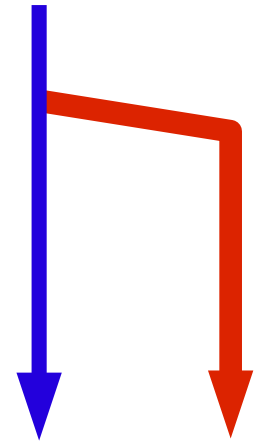
- erzeugt eine **Kopie** (Sohn) des ausführenden Prozesses (Vater)
- insb. gleicher Programmcode und Programmzähler-Stand nach `fork()`, **aber** getrennte Speicherbereiche (Kopie)

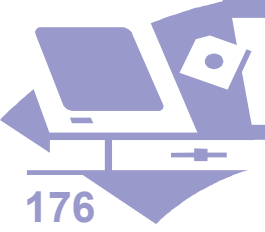
► Ergebnis im *Vaterprozess*:

ProzessID (PID) des Sohnes (oder -1 bei Fehler),

► Ergebnis im *neuen Sohnprozess*: immer 0

► `getpid()` und `getppid()` liefern die Prozess-ID des ausführenden Prozesses bzw. die Prozess-ID des Vaterprozesses („parent“)





Beispiel: fork()

176

```
#include <unistd.h>
#include <stdio.h>
```

```
int main(void) {
    int pid, n = 0;
```

```
    printf("Ich habe pid %d\n", getpid());
```

```
    pid = fork();
```

```
    if (pid == -1) {
```

```
        perror("Fehler bei fork()");
```

```
    } else if (pid == 0) {
```

```
        printf("Ich bin der Sohn!\n");
```

```
    } else {
```

```
        printf("Ich bin Vater von pid %d\n", pid);
```

```
    }
```

```
    n = n + 1;
```

```
    printf("%d Tschuess von %d\n", n, getpid());
```

```
    return 0;
```

```
}
```

```
$ ./a.out
```

```
Ich habe pid 1440
```

```
Ich bin Vater von pid 1441
```

```
1 Tschuess von 1440
```

```
Ich bin der Sohn!
```

```
1 Tschuess von 1441
```

```
(andere Ausgabe-Reihenfolge möglich!)
```



Warten auf Prozessende: wait()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

► **wait()** wartet auf das Ende (irgend)eines Sohn-Prozesses

- Ergebnis: PID des beendeten Sohnes
- in `*status` wird der Rückgabewert (*exit code*) des Prozesses abgelegt

► **waitpid()** wartet auf Ende des Sohn-Prozesses `pid`

- Ergebnis und `*status` wie oben
- `options`: Bitmaske mit Optionen, z.B. `WNOHANG`:
blockiere *nicht*, wenn (noch) kein Sohn endete

► **Prozesse, die zwar schon beendet sind, für die aber noch kein `wait` ausgeführt wurde, heißen Zombie-Prozesse.** System-Ressourcen werden erst nach `wait` vollständig freigegeben!

Teilzeit-Zombie

178

```
#include <unistd.h>
#include <stdio.h>
int main(void) {
    int pid;
    pid = fork();
    if (!pid) {
        printf("%d: Sohn wartet, ppid=%d\n", getpid(), getppid());
        sleep(5);
        printf("%d: Sohn fertig, ppid=%d\n", getpid(), getppid());
    } else {
        printf("%d: Vater wartet 25 Sekunden\n", getpid());
        sleep(25);
        printf("%d: Vater fertig\n", getpid());
    }
    return 0;
}
```

\$./a.out

2737: Vater wartet 25 Sekunden

2738: Sohn wartet, ppid=2737

2738: Sohn fertig, ppid=2737

jetzt ps abrufen

2737: Vater fertig

\$ ps -elf

0 S berta 2737 1366 ... 12:44 pts/1 00:00:00 ./a.out

1 Z berta 2738 2737 ... 12:44 pts/1 00:00:00 [a.out <defunct>]

Zombie (bis wait() ausgeführt wird)

init sammelt Waisen ein

```
#include <unistd.h>
#include <stdio.h>
```

```
int main(void) {
    int pid;
```

```
    pid = fork();
    if (!pid) {
```

```
        printf("%d: Sohn wartet, ppid=%d\n", getpid(), getppid());
```

```
        sleep(10);
```

```
        printf("%d: Sohn fertig, ppid=%d\n", getpid(), getppid());
```

```
    } else {
```

```
        printf("%d: Vater wartet 2 Sekunden\n", getpid());
```

```
        sleep(2);
```

```
        printf("%d: Vater fertig\n", getpid());
```

```
    }
```

```
    return 0;
```

```
}
```

```
$ ./a.out
```

```
2860: Vater wartet 2 Sekunden
```

```
2861: Sohn wartet, ppid=2860
```

```
2860: Vater fertig
```

```
2861: Sohn fertig, ppid=1
```

- ▶ Der `init`-Prozess (pid 1) nimmt sich aller verwaisten Prozesse an und führt jeweils `wait()` für sie aus.



Programmausführung: `exec()`

```
#include <unistd.h>
int execve(char *filename, char *argv [], char *envp[])
```

- ▶ `execve()` startet das Programm `filename`
 - mit den Parametern `argv` und
 - den Umgebungsvariablen `envp` (Vektor von Zeichenketten der Form "*variablenname=wert*")
 - Letztes Element von `argv` und `envp` muss `NULL`-Zeiger sein!
- ▶ Bei Erfolg wird der aufrufende Programm **ersetzt** durch das neu gestartete Programm
 - Prozessnummer und offenen Dateien bleiben erhalten
 - Ergebnis: -1 bei Fehler, kein Ergebnis sonst (wieso?)
- ▶ Varianten von `execve()`: `execl()`, `execv()`, `execle()`, ...

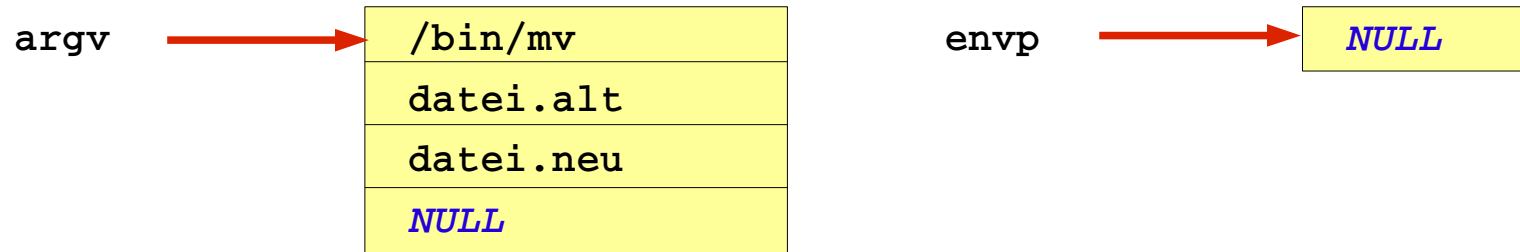


Shell-Funktionsweise

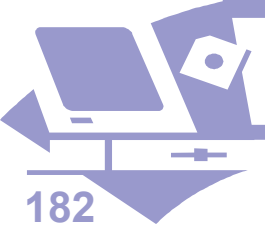
Was passiert bei der Kommandoeingabe in einer Shell?

```
$ /bin/mv datei.alt datei.neu
```

- ▶ Shell zerlegt Eingabezeile in "Wörter" und konstruiert Argument-Vektor `argv` und Vektor der Umgebungsvar. `envp`

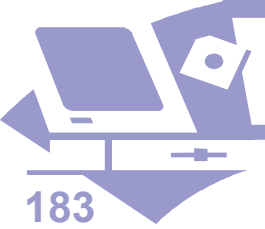


- ▶ Shell spaltet sich mit `fork()`
 - Sohn führt `execve("/bin/mv", argv, envp)` aus
 - Vater (Shell) führt `wait(&status)` aus und wartet
 - Danach fragt Vater nach nächstem Kommando usw.
- ▶ Bemerkung: Das Anhängen von "&" an die Kommandozeile sorgt für Weglassen des "`wait()`" → Sohn läuft als Hintergrundprozess



Signale

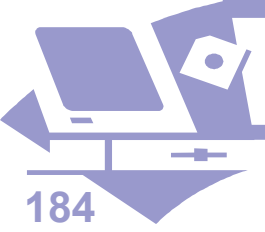
- ▶ Ein **Signal** ist eine spezielle (vordefinierte) Nachricht, von
 - einem Prozess an einen anderen (vorbehaltlich Berechtigung)
 - vom Betriebssystem-Kern an einen Prozess
- ▶ Signale teilen das Auftreten eines (unerwarteten?) Ereignisses mit, z.B.
 - Abbruch-Wunsch durch Benutzer (z.B. "[strg] [c]" gedrückt)
 - Verbindung abgebrochen (z.B. Modem-Verbindung)
 - Gleitkommafehler
- ▶ UNIX-Signale haben vordefinierte Nummern
- ▶ Den meisten Signalen kann eine eigene (C-)Funktion als **Signal-Handler** zugewiesen werden
- ▶ Analogie zu Interrupts / Interrupt-Handlern



Beispiele UNIX-Signale

Aus dem Linux-Online-Manual (man 7 signal)

Signalname	Wert	Bemerkung
-----+-----+-----		
SIGHUP	1	Verbindung beendet (Aufgehängt)
SIGINT	2	Interrupt-Signal von der Tastatur
SIGQUIT	3	Quit-Signal von der Tastatur
SIGILL	4	Falsche Instruktion
SIGTRAP	5	Überwachung/Stop Punkt
SIGABRT	6	Abbruch
SIGFPE	8	Fliesskomma Überschreitung
SIGKILL	9	Beendigungssignal <i>(nicht unterdrückbar)</i>
SIGUSR1	10	Benutzer-definiertes Signal 1
SIGSEGV	11	Ungültige Speicherreferenz
SIGUSR2	12	Benutzer-definiertes Signal 2
SIGPIPE	13	Schreiben in eine Pipeline ohne Lesen
SIGALRM	14	Zeitsignal von alarm(1).
SIGTERM	15	Beendigungssignal
SIGSTKFLT	16	Stack-Fehler im Koprozessor
SIGCHLD	17	Kind-Prozess beendet

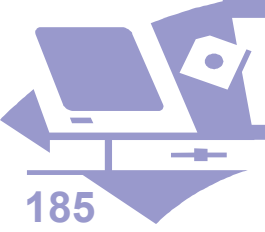


Signale verschicken: kill()

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- ▶ `kill()` schickt das Signal `sig` an den Prozess mit der ProzessID `pid`
- ▶ Rückgabewert: 0 für ok, -1 für Fehler
- ▶ Spezialfall: `sig == 0`
 - Signal wird nicht wirklich verschickt
 - Fehlerprüfung wird trotzdem durchgeführt
 - Anwendung: Überprüfung, ob Prozess `pid` existiert:

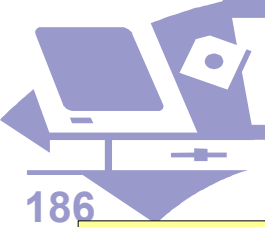
```
if (kill(pid, 0) == 0) { /* Prozess pid existiert */ }
```



Warten auf Signal: pause()

```
#include <unistd.h>
int pause(void);
```

- ▶ `pause()` wartet auf das Eintreffen eines Signals (Prozessausführung wird so lange blockiert)
- ▶ Rückgabewert ist immer -1

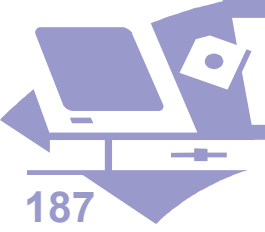


Programmstart mit Timeout-Abbruch

```
#include <...>
int main(int argc, char *argv[]) {
    char *dummyenv = NULL;
    int pid, status;

    if (pid=fork()) {
        fprintf(stderr,"Sohn pid=%d gestartet\n",pid);
        sleep(TIMEOUT);
        if (waitpid(pid,&status,WNOHANG)==0) {
            fprintf(stderr,"Timeout, Abbruch!\n");
            kill(pid, SIGKILL);
            wait(&status);
        }
        fprintf(stderr,"Sohn endet, Status=%d\n",status);
    } else {
        execve(argv[1], argv+1, &dummyenv);
        fprintf(stderr,"Fehler beim Starten von %s",argv[1]);
        exit(-1);
    }
    return 0;
}
```

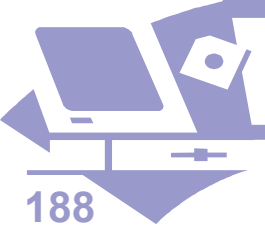
```
$ ./a.out /bin/sleep 100
Sohn pid=2337 gestartet
Timeout, Abbruch!
Sohn endet, Status=9
```



Eigener Signal-Handler

```
#include <signal.h>
sighandler_t signal(int signum, sighandler_t handler);
```

- ▶ `handler` ist (ein Zeiger auf) eine Funktion, die einen `int`-Parameter (Signalnummer) erwartet
- ▶ Besondere Werte für "`handler`":
 - `SIG_IGN`: ignoriere dieses Signal
 - `SIG_DFL`: setze Default-Aktion für dieses Signal



Beispiel: Abfangen von "ctrl-c"

188

```
#include <stdio.h>
#include <signal.h>

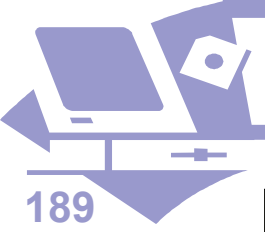
void myIntHandler(int sig) {
    fprintf(stderr, "Autsch!\n");
}

int main(void) {
    int i;

    signal(SIGINT, myIntHandler);
    for (i=0; i < 17; i++) {
        printf("Runde %d\n", i);
        sleep(1);
    }
    signal(SIGINT, SIG_DFL);
    return 0;
}
```

ctrl-c
gedrückt

```
$ ./a.out
Runde 0
Runde 1
Autsch!
Runde 2
Autsch!
Runde 3
Autsch!
Runde 4
Runde 5
...
```



alarm()

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

- ▶ `alarm()` sorgt dafür, daß dem Prozess nach `seconds` Sekunden das Signal `SIGALRM` geschickt wird
- ▶ Es gibt nur einen Alarm-Timer pro Prozess
- ▶ blockiert den Prozess *nicht* (vgl. dagegen: `sleep()`)
- ▶ Timer löschen mit `alarm(0)`
- ▶ Rückgabewert: Verbleibende Sekunden bis zum Auslösen des Signals (oder 0, falls kein Alarm aktiv)
- ▶ Abfangen eines Alarms: z.B. wie gesehen mit `signal()` Handler für Signal `SIGALRM` installieren

Prog.start mit Timeout-Abbruch (2)

```
19 #define TIMEOUT 17
   int pid = 0;

   void killer(int sig) {
       fprintf(stderr, "Timeout, kill %d!\n", pid);
       kill(pid, SIGINT);
   }

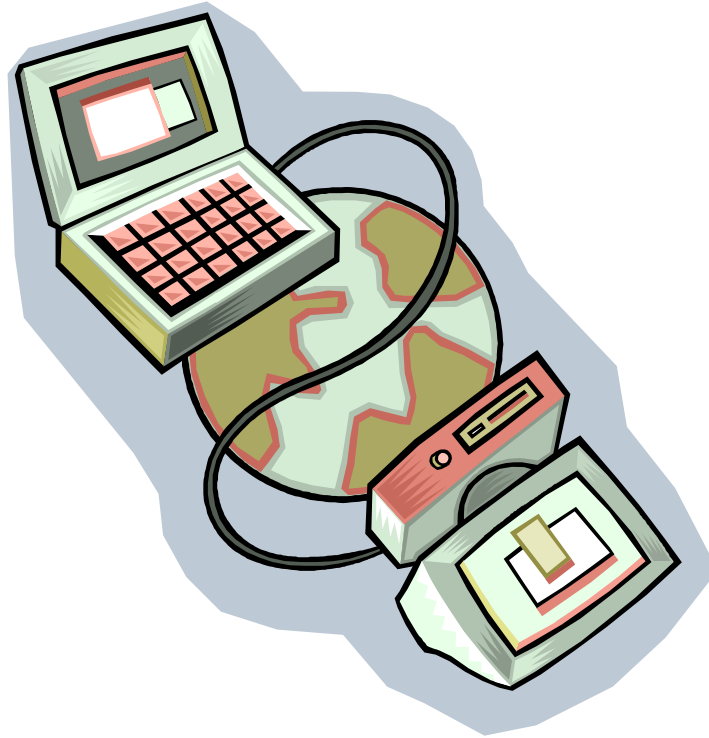
   int main(int argc, char *argv[]) {
       int status;
       if (pid=fork()) {
           fprintf(stderr, "Sohn pid=%d gestartet\n", pid);
           signal(SIGALRM, killer);
           alarm(TIMEOUT);
           wait(&status);
           fprintf(stderr, "Sohn endet mit %d\n", status);
       } else {
           execv(argv[1], argv+1);
           fprintf(stderr, "Fehler beim Starten von %s", argv[1]);
           exit(-1);
       }
       return 0;
   }
```

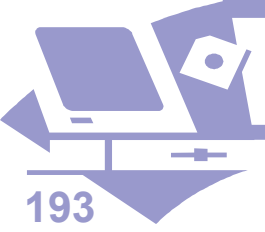
```
$ ./a.out /bin/sleep 100
Sohn pid=2061 gestartet
Timeout, kill 2061!
Sohn endet mit Status 2
```



Heute...

Interprozesskommunikation





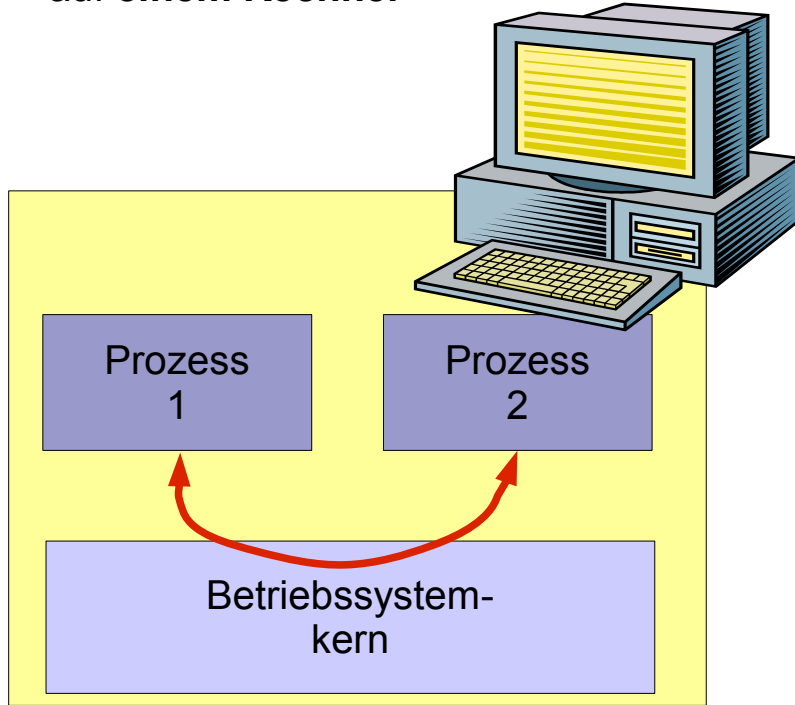
Inter-Prozess-Kommunikation

- ▶ Eine Aufgabe des Betriebssystems ist die Ermöglichung eines geregelten "Zusammenlebens" verschiedener Prozesse. Dazu gehört die Bereitstellung von Mitteln zur
- ▶ **Synchronisation**: zeitliche Koordination von Prozessen
 - Durchsetzen von Abhängigkeiten/Bedingungen zwischen Prozessen, z.B.
 - (zeitweise) alleinigen Zugriff eines Prozesses auf einen Drucker
 - Reihenfolgebedingungen (z.B. abwechselnde Aktivitäten zwischen mehreren Prozessen)
- ▶ **Kommunikation**: (umfangreicherer) Datenaustausch, z.B.
 - gemeinsam genutzter Speicher (*shared memory*)
 - Verschicken von Nachrichten (z.B. *pipes*, *message queues*)
 - Sockets (z.B. Internet, "UNIX domain sockets", ...)

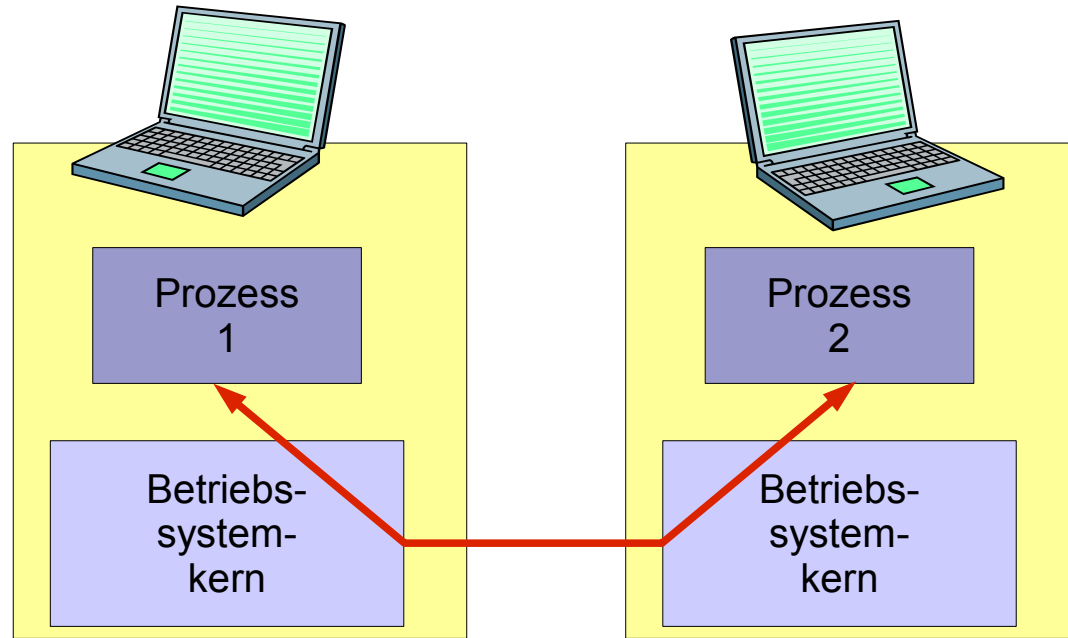


IPC - ein oder mehrere Rechner

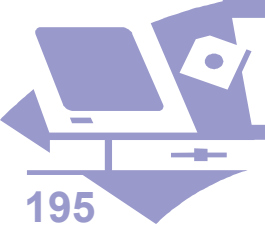
IPC zwischen Prozessen
auf **einem Rechner**



IPC zwischen Prozessen
auf **verschiedenen Rechnern**



- ▶ Einige IPC-Mechanismen funktionieren nur zwischen Prozessen auf dem selben Rechner (z.B. shared memory)
- ▶ Kriterium bei Auswahl eines IPC-Mechanismus bei der Entwicklung einer Anwendung



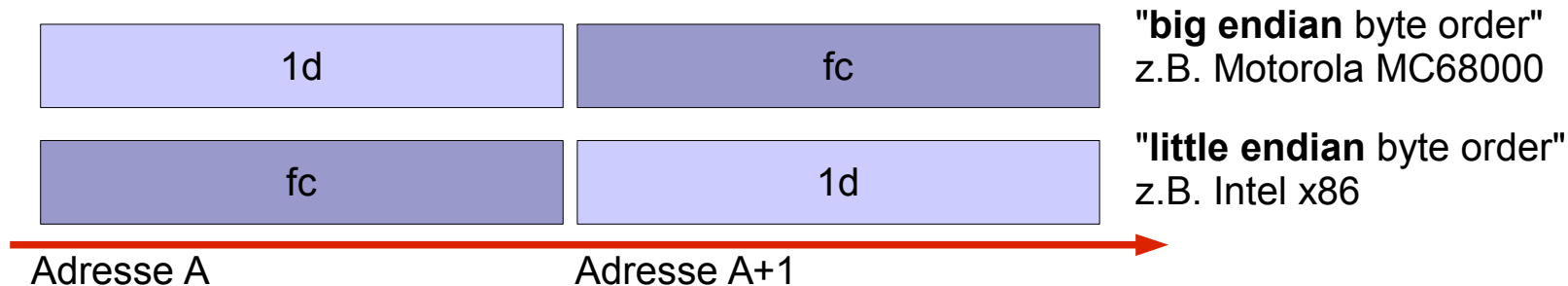
Sockets

- ▶ **Sockets:** Verbindungsendpunkte
 - **"stream sockets":** verbindungsorientiert
 - **"datagram sockets":** verbindungslos
- ▶ Programmierschnittstelle (API), eingeführt im 4.1BSD UNIX (ca 1982)
- ▶ Einheitlicher Zugang zu verschiedenen darunterliegenden Kommunikationsprotokollen, z.B.
 - "UNIX-Domain" - Rechner-lokaler Komm.-Verfahren
 - "Internet-Domain" - TCP/IP-Netzwerk-Kommunikation
 - "XNS domain" - Kommunikationsprotokoll von Xerox
 -
- ▶ Typischerweise Client/Server-Rollenteilung
(vgl. Rechnernetze-Vorlesung)



Bytes, Oktette, Network Order

- ▶ **Byte**: kleinste adressierbare Speichergröße, heute in der Regel 8 Bit
- ▶ **Oktett** (*octet*): Größe von **genau** 8 Bit
- ▶ Darstellung "größerer" Zahlen CPU-abhängig
- ▶ Beispiel: 16-Bit-Wert `0x1dfc`

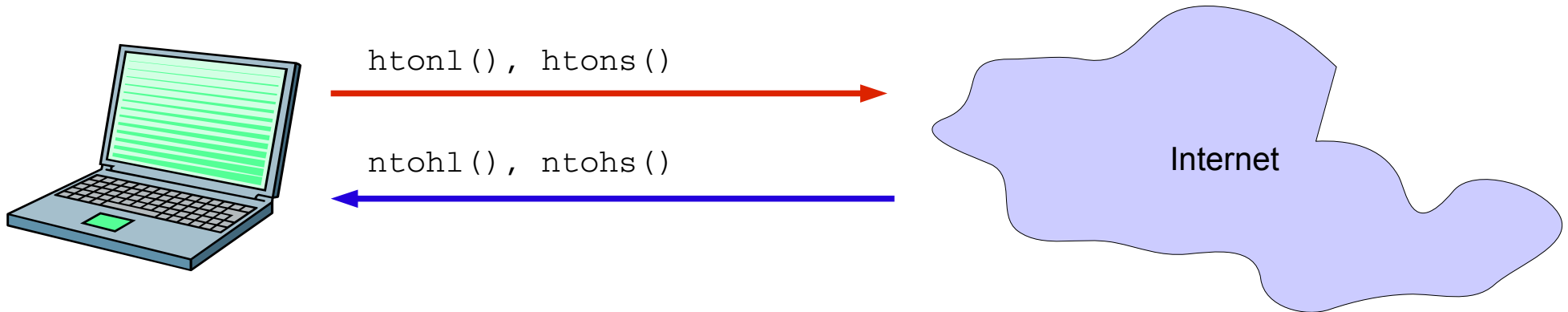


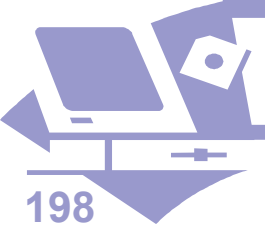
- ▶ Mögliches Problem beim Datenaustausch zwischen verschiedenen Rechnern
- ▶ Notwendigkeit einer **"Netzwerk-Byte-Ordnung"**
- ▶ TCP/IP: big endian Ordnung (für 16/32-Bit-Werte in Headern)

Konvertierungsfunktionen

```
#include <netinet/in.h>
unsigned long  int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long  int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

- Konvertierungsfunktionen zur Umwandlung zwischen Ganzzahldarstellung des Host-Rechners und der (TCP/IP) Netzwerk-Ordnung (host to net / net to host)



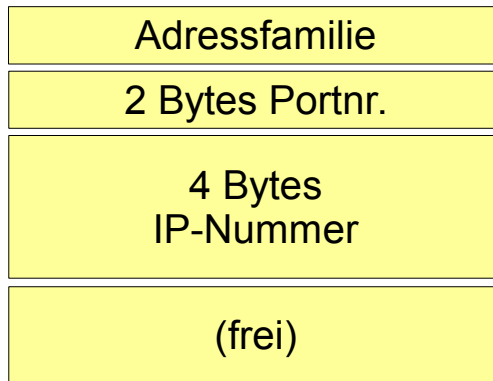


Adressierung

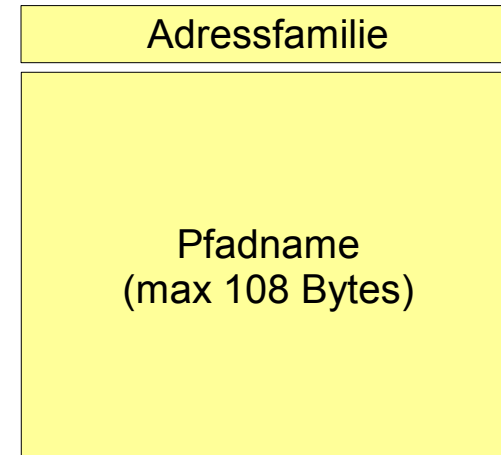
- ▶ Sockets wurden als *allgemeines* API zur Netzwerk-Programmierung entworfen
- ▶ Verschiedene Netzwerke nutzen unterschiedliche Adress-Formate, um
 - das gewünschte **Netzwerk**,
 - einen **Host** auf diesem Netzwerk und
 - einen **Prozess** auf diesem Host zu bezeichnen.
- ▶ Beispiele:
 - "UNIX Domain": Pfadname,
z.B. `/tmp/uml.ct1`
 - "Internet Domain": IP-**Adresse** und **Port**-Nummer
z.B. `192.168.177.42:80`

AF_XXX

- ▶ Socket-API-Funktionen kapseln die Adressangabe in einer entsprechenden "sockaddr"-Struct
- ▶ Die Adressfamilie (= Protokollfamilie) gibt dabei die Art des verwendeten Protokolls an.
 - AF_INET (= PF_INET)
 - AF_UNIX (= PF_UNIX)
 - ...
- ▶ Je Adressfamilie gibt es eine sockaddr-Variante, z.B.



struct sockaddr_in



struct sockaddr_un



Konkret: structs für's Internet

Internet-Adresse (32 Bit) in Netzwerk-Byteordnung

```
struct in_addr { unsigned long int  s_addr; };
```

Internet-Adress-Struktur (IP-Adresse *und* Port)

```
struct sockaddr_in {  
    short                sin_family;           /* = AF_INET */  
    unsigned short       sin_port;            /* Portnummer */  
    struct in_addr       sin_addr;            /* 32 Bit IP-Adr */  
    char                 sin_zero[8];         /* nicht benutzt */  
};
```

► Die Hilfsfunktion

`void *memset(void *s, int c, size_t n)`
setzt n Bytes, mit Adresse s beginnend, auf den Wert c.

► Beispiel: `memset(&mystruct, 0, sizeof(mystruct))`



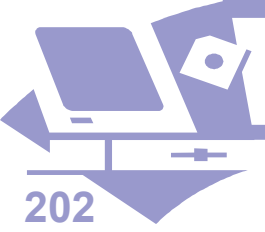
Adressumwandlungsfunktionen

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
int inet_aton(char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);
```

```
struct in_addr {
    unsigned long int s_addr;
}
```

- ▶ **inet_aton()** konvertiert Zeichenkette **cp** mit IP-Adresse in Punkt-Notation ("a.b.c.d") in Adress-Struktur **struct in_addr *inp**; liefert "wahr", falls Adresse ok (!), sonst "falsch" (0)
- ▶ **inet_ntoa()** gibt Zeichenkette mit IP-Adresse zu übergebener Adress-Struktur **in** zurück (Vorsicht, wird möglicherweise bei nächstem Aufruf überschrieben; erhaltenes Ergebnis ggf. gleich *kopieren*!);



Wer liefert was?

- ▶ Eine Verbindung wird beschrieben durch eine Assoziation
 - Protokoll
 - lokale Adresse, lokaler Prozess
 - entfernte Adresse, entfernter Prozess
- ▶ Welche Socket-Funktion trägt diese Angaben bei?

	Protokoll	lokale Adr/Proz	entfernter Adr/Proz.
verb.orient. Server	<code>socket()</code>	<code>bind()</code>	<code>listen()</code> , <code>accept()</code>
verb.orient. Client	<code>socket()</code>	<code>connect()</code>	
verb.loser Server	<code>socket()</code>	<code>bind()</code>	<code>recvfrom()</code>
verb.loser Client	<code>socket()</code>	<code>bind()</code>	<code>sendto()</code>



Sockets (verbindungsorientiert)

203 Server

`socket()` - Socket anlegen

`bind()`
Socket an Adresse/Port binden

`listen()`
Max. n wartende Verbindungen

`accept()`
Warten auf eingehende Verbindung

`read()` / `recv()`
max. n Bytes vom Client empfangen

`write()` / `send()`
Daten an Client senden

`close()`
Verbindung schließen

Client

`socket()` - Socket anlegen

`connect()`
Verbindung zu Server aufbauen

`write()` / `send()`
Daten an Server senden

`read()` / `recv()`
Daten vom Server empfangen

`close()`
Verbindung schließen



Sockets (verbindungslos)

Empfänger

`socket()` - Socket anlegen

`bind()`
Socket an Adresse/Port binden

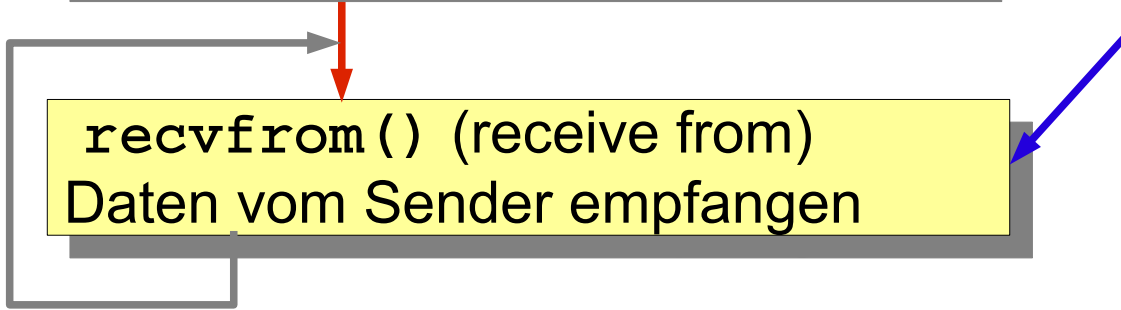
`recvfrom()` (receive from)
Daten vom Sender empfangen

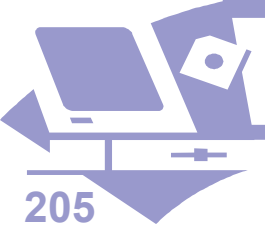
Sender

`socket()` - Socket anlegen

`bind()` / `connect()` möglich

`sendto()`, `write()`, `send()`
Daten an Empfänger senden





socket() - Socket erzeugen

```
#include <sys/socket.h>
#include <sys/types.h>
int socket(int domain, int type, int protocol);
```

domain: AF_UNIX, AF_INET, AF_...

type: Socket-Typ

SOCK_STREAM	Vollduplex Bytestrom, verb.orientiert
SOCK_DGRAM	Datagramme, verbindungslos
SOCK_RAW	direkter Zugriff auf unterliegendes Protokoll

protocol für explizite Protokollwahl; normalerweise 0

Ergebnis: socket-Deskriptor für andere Socket-Funktionen

Kombinationen und resultierende Protokollwahl z.B.

	AF_UNIX	AF_INET
SOCK_STREAM	(ja)	TCP
SOCK_DGRAM	(ja)	UDP
SOCK_RAW		IP



bind()

```
#include <sys/socket.h>
#include <sys/types.h>
int bind(int sockfd, struct sockaddr *my_addr, int len);
```

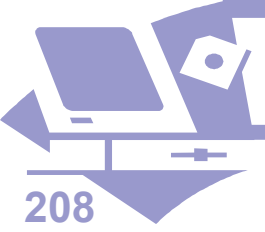
- ▶ **bind()** weist einem Socket "einen Namen zu", abhängig vom verwendeten Protokoll, z.B.
 - einen **Pfadname** für AF_UNIX
 - eine **Internet-Adresse/Port** für AF_INET
- ▶ **sockfd**: Socket-Deskriptor aus socket()
- ▶ **my_addr**: Zeiger auf zuvor belegte Adress-Struktur
- ▶ **len**: Länge der Adress-Struktur in Bytes
- ▶ Ergebnis: 0 für ok, -1 für Fehler



connect()

```
#include <sys/socket.h>
#include <sys/types.h>
int connect(int sockfd, struct sockaddr *server_addr, int len);
```

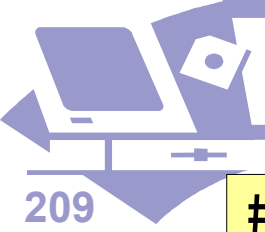
- ▶ **connect()** baut eine Verbindung zu einem Server auf
 - **sockfd**: Socket-Deskriptor
 - **server_addr**: Adress-Struktur mit Adresse des Servers
 - **len**: Länge der Adress-Struktur
- ▶ Client muß nicht **bind()** aufrufen; **connect()** füllt dann neben den entfernten auch die lokalen Angaben zu Adr./Prozess
- ▶ Verwendung mit verbindungslosen Clients:
- ▶ Festlegung einer Zielangabe, nachfolgend kann **write()/send()** (ohne Adressangabe) benutzt werden und es werden nur Datagramme von diesem Ziel empfangen
- ▶ Überprüfung unzulässiger Adressangaben, falls möglich (liefert dann Fehler zurück)



listen()

```
#include <sys/socket.h>
int listen(int sockfd, int anzahl);
```

- ▶ `listen()` sorgt dafür, dass ein Stream-Socket (`SOCK_STREAM`) Verbindungen annehmen kann
 - `sockfd`: Socket-Deskriptor
 - `anzahl`: Länge der Warteschlange für Verbindungswünsche
 - Stehen mehr als `anzahl`-viele Verbindungswünsche an, werden die überzähligen abgewiesen ("connection refused")
- ▶ Die Entgegennahme einer Verbindung erfolgt mit `accept()`
- ▶ Ergebnis: 0 für ok, -1 für Fehler



accept()

209

```
#include <sys/socket.h>
#include <sys/types.h>
int accept(int sockfd, struct sockaddr *addr, int *len);
```

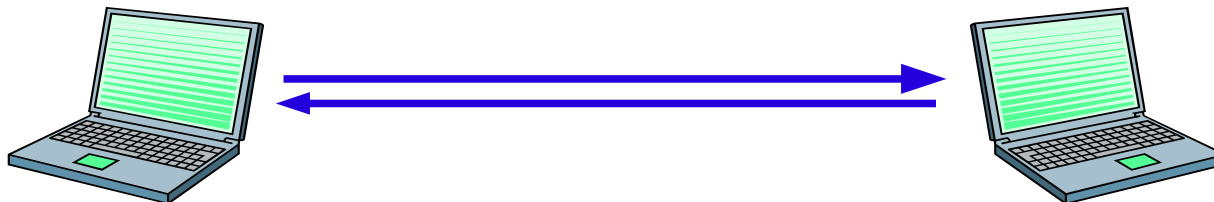
- ▶ **accept()** wartet auf eine eingehende Verbindung und nimmt sie entgegen
 - **sockfd**: Socket-Deskriptor
 - **addr**: Adressangabe der eingehenden Verbindung
 - **len**: *Zeiger* auf `int` mit Inhalt
 - ➔ vor Aufruf von `accept()`: Länge der `sockaddr`-Struktur
 - ➔ im Aufruf schreibt `accept()` *tatsächliche* Länge hinein
- ▶ Wird in verbindungsorientierten Servern verwendet (Voraussetzung: vorheriges `listen()`)
- ▶ Ergebnis: -1 für Fehler bzw. neuer Socket-Deskriptor (dann wurden auch `*addr` und `*len` entsprechend aktualisiert)

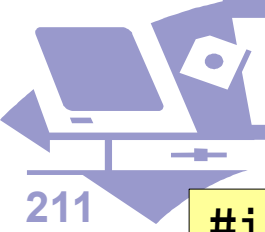


close(), shutdown()

```
#include <sys/socket.h>
#include <sys/types.h>
int close(int sockfd);
int shutdown(int sockfd, int modus);
```

- ▶ **close()** schließt eine Socket-Verbindung **sockfd**
- ▶ **shutdown()** erlaubt "partiell" Schließen der Duplex-Verbindung
 - **modus=0**: es können keine Daten mehr über **sockfd** empfangen werden
 - **modus=1**: es können keine Daten mehr über **sockfd** geschrieben werden
 - **modus=2**: kein Schreiben und Lesen mehr über **sockfd**





Lesen und Schreiben

```
#include <sys/socket.h>
#include <sys/types.h>
int send(int sockfd, void *msg, size_t len, int flags);
int sendto(int sockfd, void *msg, size_t len,
           int flags, struct sockaddr *to, socklen_t tolen);
int recv(int sockfd, void *buf, size_t len, int flags);
int recvfrom(int sockfd, void *buf, size_t len,
             int flags, struct sockaddr *from, socklen_t *fromlen);
```

- ▶ `send()` verschickt die `len` lange Nachricht `msg` über Socket `sockfd`, `sendto()` zusätzlich explizite Ziel-Adresse `to`
- ▶ `recv()` empfängt eine max. `len` lange Nachricht über Socket `sockfd` und schreibt sie in Buffer `buf` (`flags = 0`), `recvfrom()` speichert Absender-Adresse in `from`
- ▶ `sendto()` und `recvfrom()` bei verbindungslosen Diensten
- ▶ `flags`: Standardwert ist 0 (mehr im Online-Manual)
- ▶ Die Verwendung von `read()` und `write()` ist ebenfalls möglich
- ▶ Ergebnisse: Anzahl der geschickten/gelesenen Bytes oder -1 (für Fehler)



Beispiel: Zählserver (1)

Zählserver ist ein TCP-Server, der einen Verbindungsaufbau mit dem Senden der jeweils nächsten natürlichen Zahl beantwortet.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

#define PORTNUMMER 1234

int main(void) {
    char nachricht[80];
    int zaehl=0, sockfd, newsockfd, clientlen;
    struct sockaddr_in servaddr, clientaddr;

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORTNUMMER);
    ...
}
```

steht für
„beliebige Adresse / alle Interfaces“



Beispiel: Zählserver (2)

```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket"); exit(-1);
}
if (bind(sockfd, (struct sockaddr*)&servaddr,
          sizeof(struct sockaddr_in)) < 0) {
    perror("bind"); exit(-1);
}
if (listen(sockfd, 5) < 0) { perror("listen");exit(-1);}

for(;;) {
    clientlen = sizeof(struct sockaddr);
    newsockfd = accept(sockfd,
                       (struct sockaddr *)&clientaddr, &clientlen);
    if (newsockfd < 0) { perror("accept"); exit(-1); }
    sprintf(nachricht,"%d\r\n",++zaehl);
    write(newsockfd, nachricht, strlen(nachricht));
    close(newsockfd);
}
return 0;
}
```



Zähl-Client (1)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>

#define PORTNUMMER 1234

int main(int argc, char *argv[]) {
    char buffer[80];
    int wert, sockfd, n;
    struct sockaddr_in servaddr;
    char *host = argv[1];

    if (inet_aton(host, &servaddr.sin_addr) == 0) {
        perror("inet_aton"); exit(1);
    };
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORTNUMMER);
```




Zähl-Client (2)

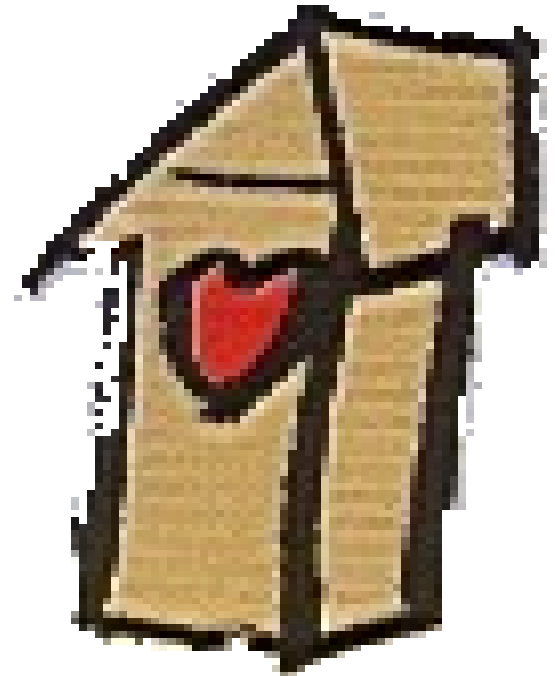
```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {  
    perror("socket"); exit(-1);  
}  
if (connect(sockfd, (struct sockaddr *)&servaddr,  
           sizeof(servaddr)) < 0) {  
    perror("connect"); exit(-1);  
}  
n = read(sockfd, buffer, sizeof(buffer));  
sscanf(buffer, "%d", &wert);  
printf("Empfangene Zahl: %d\n", wert);  
close(sockfd);  
return 0;  
}
```

- ▶ `sscanf()` und `sprintf()` funktionieren analog zu `fscanf()` und `fprintf()`, aber
- ▶ anstelle einer Datei den als ersten Parameter übergebenen `char`-Vektor zum Lesen/Schreiben



Heute...

Interprozess- kommunikation (2)





Beispiel: UDP-Server

Der UDP-Server empfängt UDP-Pakete und schickt sie mit einer Seriennummer versehen an den Absender zurück

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

#define PORTNUMMER 1234

int main(void) {
    char nachricht[80], buffer[100];
    int zaehl=0, sockfd, clientaddrsz, n;
    struct sockaddr_in servaddr, clientaddr;

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORTNUMMER);
    ...
}
```



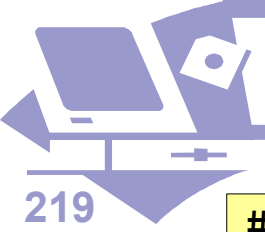
Beispiel: UDP-Server (2)

```
if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("socket"); exit(-1);
}

if (bind(sockfd, (struct sockaddr*)&servaddr,
        sizeof(struct sockaddr_in)) < 0) {
    perror("bind"); exit(-1);
}

for(;;) {
    clientaddrsz = sizeof(clientaddr);
    n = recvfrom(sockfd, buffer, sizeof(buffer), 0,
        (struct sockaddr *)&clientaddr, &clientaddrsz);
    fprintf(stderr, "-> Nachricht %d (%s)\n", n, buffer);

    sprintf(nachricht, "%s (%d)\r\n", buffer, ++zaehl);
    if (sendto(sockfd, nachricht, strlen(nachricht)+1, 0,
        (struct sockaddr*)&clientaddr, clientaddrsz) == -1)
        { perror("sendto"); exit(-1); };
}
return 0;
}
```



Beispiel: UDP-Client (1)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
```

```
#define PORTNUMMER 1234
```

```
int main(int argc, char *argv[]) {
    char buffer[80];
    int wert, sockfd, servlen;
    struct sockaddr_in servaddr;
    char *host = argv[1];
```

```
    if (inet_aton(host, &servaddr.sin_addr) == 0) {
        perror("inet_aton"); exit(1);
    }
```

```
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORTNUMMER);
    ...
```

Aufruf:

a.out ipaddr nachricht

z.B.

a.out 192.15.33.2 hallo

Ergebnis: hallo (17)



Beispiel: UDP-Client (2)

```
if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {  
    perror("socket"); exit(-1);  
}  
  
sendto(sockfd, argv[2], strlen(argv[2])+1, 0,  
        (struct sockaddr *)&servaddr, sizeof(struct sockaddr));  
  
recvfrom(sockfd, buffer, sizeof(buffer), 0,  
          (struct sockaddr *)&servaddr, &servlen);  
  
fprintf(stderr, "Ergebnis: %s\n", buffer);  
close(sockfd);  
return 0;  
}
```

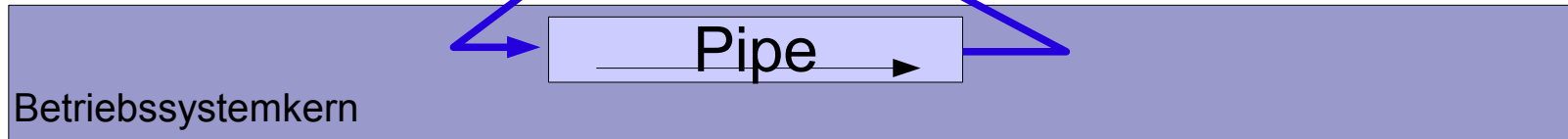
Anonyme Pipes

Vaterprozess

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

Sohnprozess

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



- ▶ Einfacher IPC-Mechanismus zwischen Vater-/Sohn-Prozessen
- ▶ Pipe ("Rohrleitung") überträgt einen Einweg-Byte-Strom von Prozess A zu Prozess B (in first-in-first-out Reihenfolge)
- ▶ Feste Puffergröße (z.B. 4096 Bytes)
- ▶ Systemfunktion `int pipe(int fds[2]);`
erzeugt zwei File-Deskriptoren im übergebenen Vektor `fds`:
`fds[0]` ist zum Lesen geöffnet
`fds[1]` zum Schreiben
- ▶ Rückgabewert: 0 für ok, -1 für Fehler

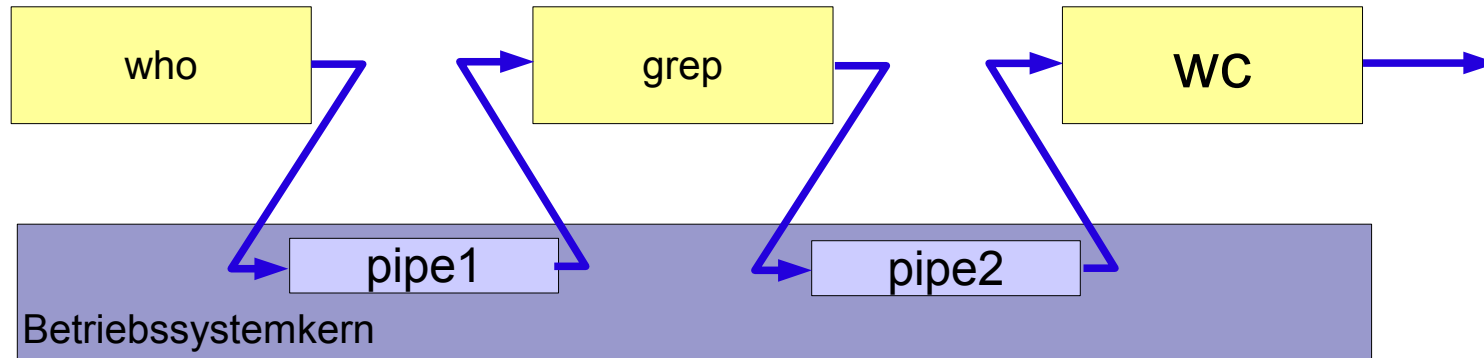


Beispiel: Pipes in der Shell

Shell-Kommandozeile (wie oft ist User "trude" auf dem Rechner angemeldet?)

```
who | grep "trude" | wc
```

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



popen()

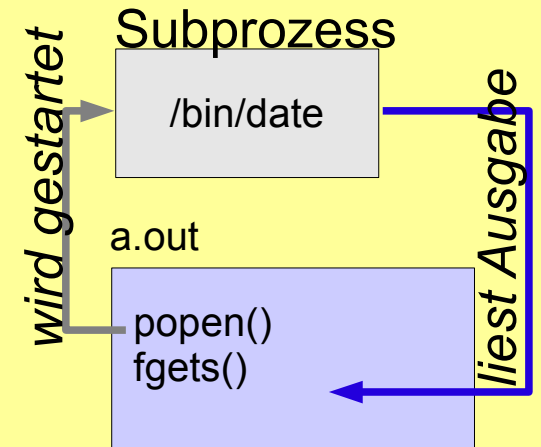
229

```
#include <stdio.h>
#include <stdlib.h>
#define MAXZEILE 80
```

```
int main(void) {
    char zeile[MAXZEILE];
    FILE *fp;
    if ( (fp = popen("/bin/date", "r")) == NULL) {
        perror("Fehler bei popen"); exit(1);
    }
    if (fgets(zeile, MAXZEILE, fp) == NULL) {
        perror("Fehler bei fgets"); exit(2);
    }
    fclose(fp);
    printf("Ausgabe von 'date' ist: %s\n", zeile);
    return 0;
}
```

```
$ ./a.out
```

Ausgabe von 'date' ist: Tue May 20 11:29:06 CEST 2003



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

Benannte Pipes (FIFOs)

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

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

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

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

(Fehlerbehandlung weggelassen)

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



select()

```
struct timeval {  
    long tv_sec; /* Sekunden */  
    long tv_usec; /* Mikrosek.*/  
};
```

```
#include <sys/time.h>  
#include <sys/types.h>  
#include <unistd.h>  
int select(int n, fd_set *readfds, fd_set *writefds,  
           fd_set *exceptfds, struct timeval *timeout);
```

```
FD_CLR(int fd, fd_set *set); FD_ISSET(int fd, fd_set *set);  
FD_SET(int fd, fd_set *set); FD_ZERO(fd_set *set);
```

- ▶ `fd_set` ist eine Menge von Deskriptoren; Ein bestimmter Deskriptor `fd` kann mit `FD_SET` zu einem `fd_set` hinzugefügt, mit `FD_CLR` herausgenommen und mit `FD_ISSET` auf Enthaltensein getestet werden. `FD_ZERO` löscht ein `fd_set`.
- ▶ `select` wartet, bis einer der Deskriptoren die entsprechende Bedingung eintritt: `readfds` (Eingabe), `writefds` (Ausgabe), `exceptfds` (Ausnahmebedingung), oder bis die `timeout`-Frist abgelaufen ist (`timeout==NULL`: Timeout "unendlich")
- ▶ `n` ist höchster benutzter Deskriptor + 1
- ▶ Ergebnis: -1 für Fehler, 0 für Timeout, >0 für Anzahl bereiter Deskriptoren (die `fds` werden von `select()` neu belegt)



Beispiel: select()

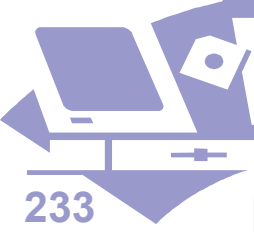
```
...
fd_set readfds;
fd_set writefds;

FD_ZERO(&readfds);
FD_ZERO(&writefds);

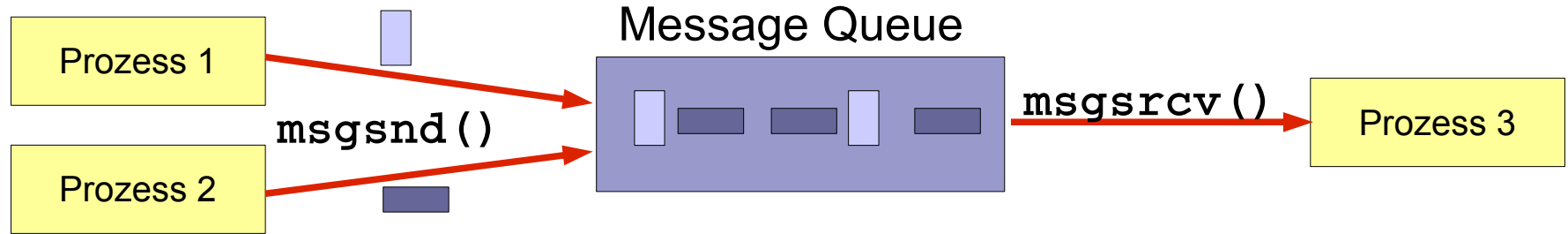
FD_SET(0, &readfds);          /* Filedeskriptor 0 = StdEingabe */
FD_SET(sockfd, &readfds);
FD_SET(pipefd, &writefds);

/* NULL fuer "unbenutzte Bedingung" zulaessig */
if (select(5, &readfds, &writefds, NULL, NULL) < 0) {
    perror("Fehler bei select()"); exit(1);
}

/* Hier wissen wir: Select hat ausgelöst, daher liegt
 * Input auf Stdin oder sockfd bzw. Outputmögl auf pipefd
 * entsprechende Lese-/Schreibop wird also nicht blocken
 */
if (FD_ISSET(sockfd, &readfds)) {
    /* Input von Deskriptor 'sockfd' verfügbar */
    ...
}
...
```



Message Queues



- ▶ Eine Message Queue ist eine verkettete (Nachrichten-) Liste, die vom Kernel verwaltet wird
- ▶ und die durch einen (vom Programmierer vergebenen) Schlüsselwert (key) identifiziert wird.
- ▶ Mehrere Sender- und Empfänger können **typisierte Nachrichten variabler Länge austauschen**, sofern sie die nötigen Berechtigungen haben.
- ▶ Die Interpretation eines Nachrichtentyps ist Sache der Prozesse (nicht vom System vordefiniert).
- ▶ Empfangsreihenfolge normalerweise "first-in-first-out", eine Priorisierung der Nachrichten ist aber auch möglich



Übersicht: IPC-Verfahren

IPC-Typ	verbindungslos?	verlässlich?	Flußkontrolle?	Datensätze?	Nachrichtentypen/ Prioritäten?
Message Queues	nein	ja	ja	ja	ja
UNIX Stream Sockets	nein	ja	ja	nein	nein
UNIX Datagr. Sockets	ja	(ja)	nein	ja	nein
Pipes, FIFOs	nein	ja	ja	nein	nein



Shared Memory

- ▶ "Speicherbasierte Kommunikation": Gemeinsame Nutzung von Speicherbereichen (*shared memory segments*) durch verschiedene Prozesse; sehr schnell
- ▶ Zugriffs-Koordination obliegt Sender/Empfänger
- ▶ Funktionen:
 - `int shmget(long key, int size, int flag)`
erzeugt bzw. gibt Zugriff auf das Shared-Memory-Segment `key` der Größe `size` und liefert eine ID zurück
 - `char *shmat(int id, char *addr, int flag)`
blendet Shared-Mem-Segment `id` in den Adressraum des Prozesses ein (möglichst bei Wunschadresse `addr`, 0=egal)
 - `int shmdt(char *addr)`
entfernt Shared-Mem-Seg. aus Adressraum des Prozesses
 - `int shmctl(int id, int cmd, struct shmid_ds *param)`
Kontrolloperationen ausführen (insb. shm-Segm. entfernen)



Eigenschaften der Komm.Mechanismen

- ▶ Mögliche **Anzahl** der Kommunikationsteilnehmer
 - genau zwei (z.B. Pipes)
 - mehr als zwei (z.B. Message Queues)
- ▶ **Richtung** des Nachrichtenflusses
 - gerichtet (unidirektional): Sender-/Empfängerrolle ist zwischen den Prozessen fest verteilt (z.B. Pipe)
 - ungerichtet (bidirektional): Prozesse können beide Rollen haben (z.B. sockets)
- ▶ Entwurfsaspekte
- ▶ **Adressierung**
 - direkt (Ziel-Prozess ist Sender bekannt, z.B. sockets) oder
 - indirekt (Ziel-Prozess ist Sender unbekannt, z.B. MsgQueue)
 - Format der Adresse: IP-Adressen, Pfade, Keys
- ▶ Nachrichten-**Pufferung**
- ▶ **Art der Nachricht** (getypt? Bytestrom/Datagramm?, ...)



Prozess-Synchronisation

- ▶ Prozesse werden unabhängig voneinander ausgeführt
- ▶ Notwendig daher:
 - unerwünschte gegenseitige Beeinflussung vermeiden
(z.B. zeitweise exklusiver Zugriff auf gemeinsam genutzte Ressource, etwa einen Drucker)
 - erwünschte Kooperation ermöglichen



Konflikt / race condition

- ▶ Zwei Prozesse stehen **im Konflikt** zueinander, wenn es ein Betriebsmittel gibt, das sie gemeinsam nutzen (ansonsten heißen sie **unabhängig**).
- ▶ Situationen, in denen
 - **mehrer** Prozesse auf ein **gemeinsames** Betriebsmittel zugreifen und
 - das **Ergebnis** davon abhängt, welcher Prozess wann läuft (wie die Anweisungen der Prozesse in ihrer **Ausführungsreihenfolge** verzahnt sind),
 - heißen **race conditions**.



Beispiel: race condition

Prozess 1

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

Prozess 2

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

Mögliche Ausführungsreihenfolge der Anweisungen in Prozess 1,2

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

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

- ▶ Pech, Gehaltsüberweisung ist "verloren gegangen"
- ▶ Bei anderen Reihenfolgen werden die beiden Berechnungen "richtig" ausgeführt, oder es geht der Dauerauftrag verloren → Problem.



Kritischer Abschnitt / w. Ausschluss

- ▶ Ein Abschnitt eines Programms mit Zugriffen auf gemeinsame Betriebsmittel heißt **kritischer Abschnitt** (*critical section*).
- ▶ Ein Verfahren, das den
 - gleichzeitigen lesenden oder schreibenden Zugriff
 - von mehr als einem Prozess auf ein Betriebsmittel verhindert,
- ▶ heißt Verfahren zum **wechselseitigen Ausschluss** (*mutual exclusion*).

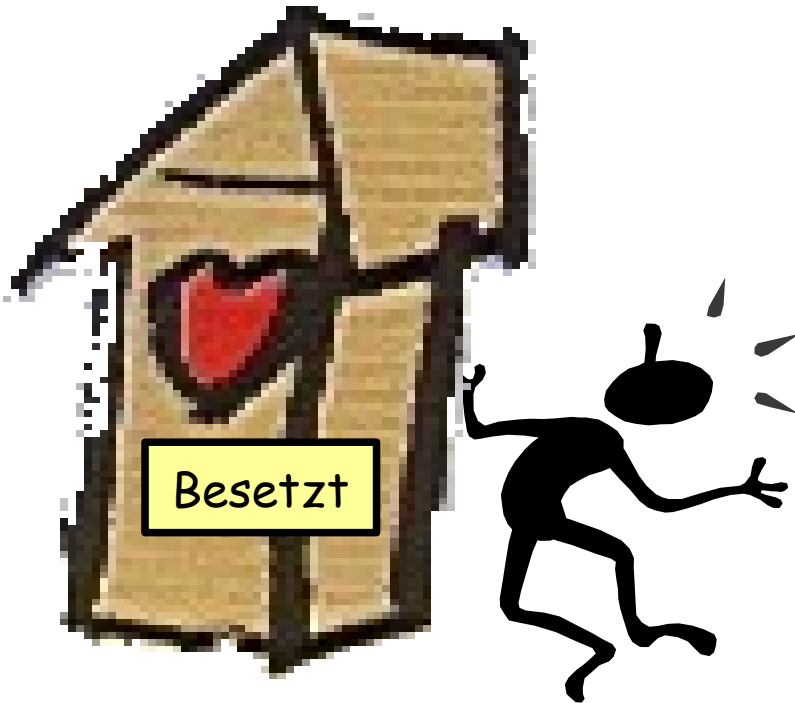


Anforderungen

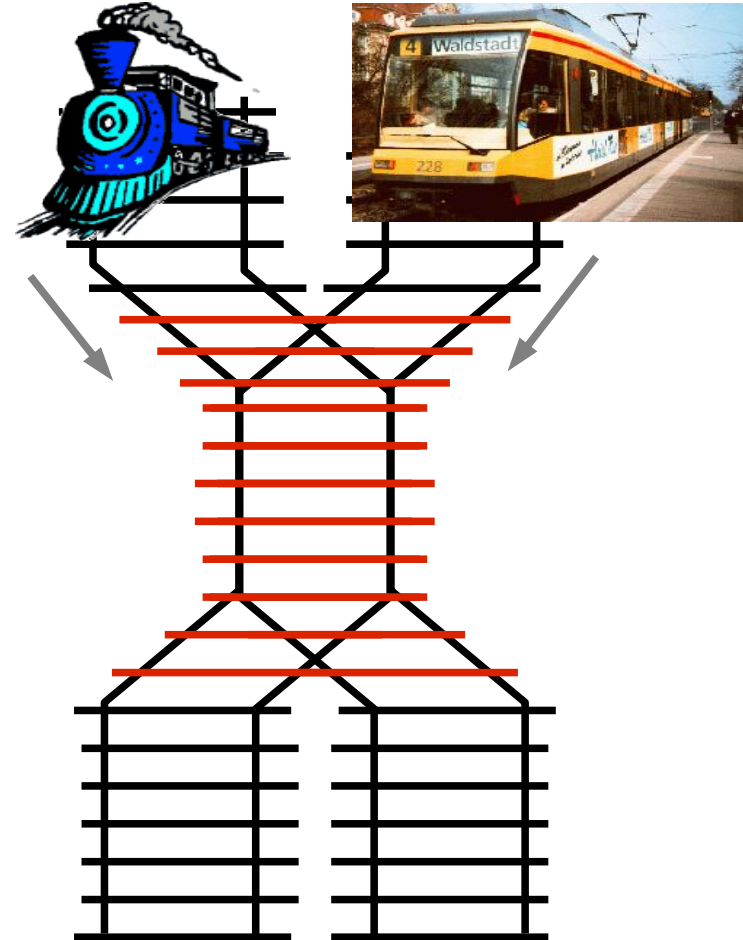
- ▶ Anforderungen an ein gutes Verfahren für gegenseitigen Ausschluss sind:
 - **nur ein** Prozess gleichzeitig im kritischen Abschnitt
 - keine Annahmen über Ausführungskontext (z.B. CPUs)
 - außerhalb des kritischen Abschnitts darf ein Prozess keinen anderen blockieren
 - **Fairness**: alle Prozesse werden gleich behandelt
 - kein Prozess darf unendlich lang auf Eintritt in den kritischen Abschnitt warten müssen ("verhungern")
- ▶ Damit können *race conditions* verhindert werden.

Praxisbeispiele

Wechselseitiger Ausschluss



Kritischer Abschnitt



Wie kann man wechselseitigen Ausschluss realisieren?



Interrupts sperren?

► Skrupellose Lösung:

- Bei Betreten eines kritischen Bereichs **sperrt** der Prozess einfach **alle Unterbrechungen**
- damit wird u.a. der Aufruf des Prozess-Schedulers verhindert,
- es kann also insbesondere kein Kontextwechsel stattfinden.
- Am Ende des kritischen Bereichs schaltet der Prozess die Unterbrechungen wieder ein.



Interrupts sperren - Nachteile

► Nachteile:

- Normale Benutzer dürfen i.d.R. nicht alle Interrupts sperren
- Bei Mehrprozessor-Maschinen wäre ohnehin nur eine CPU betroffen, die anderen könnten noch auf die gemeinsame Ressource zugreifen.
- Gefahr, daß bei Programmfehler die Interrupts abgeschaltet bleiben
→ System wird lahmgelegt

Sperrvariablen

246

- ▶ **Annahme: es gibt eine gemeinsame Variable, die**
 - beim Betreten des kritischen Bereichs auf 1 und
 - beim Verlassen auf 0 gesetzt wird.
- ▶ **Initialisierung der Variablen mit 0.**

Prozess 1

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

Prozess 2

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

- ▶ **Genügt das?**
 - **Nein!** Ähnliches Problem wie Konto-Beispiel (s.o.)



Modifikation: Spinlock

247

Prozess 1

```
while (1) {  
  while (dran != 1) { }  
  /* kritischer Bereich */  
  dran = 2;  
  /* unkritischer Ber. */  
}
```

Prozess 2

```
while (1) {  
  while (dran != 2) { }  
  /* kritischer Bereich */  
  dran = 1;  
  /* unkritischer Ber. */  
}
```

- ▶ Gemeinsame Variable "dran" gibt an, welcher Prozess den kritischen Bereich betreten darf (Anfangswert z.B. 1).
- ▶ Im Gegensatz zu oben räumt jeder Prozess *einem anderen* das Recht zum Betreten des kritischen Bereichs ein, damit keine Überschneidung.
- ▶ Dieses Verfahren (*spinlock*) vermeidet race conditions, aber unschön:
 - verschwenderisches "aktives" Warten (*busy wait*)
 - strenges Abwechseln der Prozesse erforderlich
 - ...



Lösung: Hardware-Unterstützung

- ▶ Es wurden verschiedene reine Software-Lösungen vorgeschlagen, die aber alle zu aufwendig sind.
- ▶ Lösung: Prozessor hat einen **Maschinenbefehl** zum
 - Testen einer Speicherstelle mit
 - anschließendem Schreiben in diese Speicherstelle
- ▶ **"Test-and-Set", z.B.**
 - Testergebnis "falsch": Speicherstelle war bereits belegt
 - Testergebnis "wahr": Speicherstelle war nicht belegt
 - in *beiden* Fällen ist die Speicherstelle nachher belegt.
- ▶ Keine race condition, weil Testen und Setzen **ununterbrechbar in *einer* Maschineninstruktion** erfolgt.



Passives Warten

- ▶ Bisher: "aktives Warten" (z.B. *spinlock*) vor Betreten des kritischen Bereichs; **Verschwendung** von CPU-Zeit
- ▶ **Daher: Betriebssystem-Unterstützung**
 - CPU-Zeit soll sinnvoll genutzt werden
 - Prozesse, die auf Eintritt in einen kritischen Bereich warten, werden daher blockiert
 - und beim Austritt eines anderen Prozesses aus diesem kritischen Bereich wieder de-blockiert (passives Warten).
- ▶ **Das Betriebssystem muss dem Programmierer also Mittel zur Verfügung stellen, um**
 - kritische **Bereiche** **kenntlich** zu **machen** und
 - den **Zugang** zu **kontrollieren**



Heute...

Synchronisierung (Forts.)

Threads





Praktikum:

- Projektende in letzter Vorlesungswoche
 - Abgabe per Upload in's read.MI
(Sourcen, Makefile)
- Abnahmetermin nach Upload vereinbaren
(sobald fertig - gerne früher als Endtermin)
- Test der spezifizierten Funktionalität mit Mail-Client
Thunderbird bzw. Python Standardbibliothek
- Termine siehe Übungsblatt (2. Seite, unten)

Semaphoren



By Hamilton Richards - manuscripts of Edsger W. Dijkstra, University Texas at Austin, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=4204157>

- ▶ 1965 von Edsger W. Dijkstra eingeführt
- ▶ Supermarkt-Analogie:
 - Kunde darf den Laden nur mit Einkaufswagen betreten
 - es steht nur eine begrenzte Anzahl von Einkaufswagen bereit
 - sind alle Wagen vergeben, müssen neue Kunden warten, bis ein Wagen zurückgegeben wird.



3dman_eu, <https://pixabay.com/de/einkaufswagen-shopping-chromstahl-1019925/>

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

Operationen auf Semaphoren

253

Atomare Operationen!

► Initialisierung

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

► Operation $P()$: Passier(-Wunsch)

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

► Operation $V()$: Freigeben

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





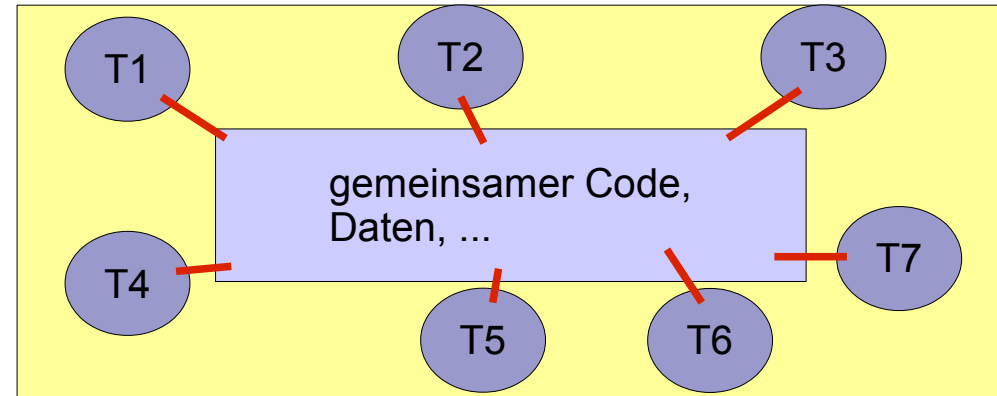
Prozesse und Threads

- ▶ **Prozesse** haben eigenen umfangreichen Kontext: Speicherbereiche, offene Dateien, ...
 - Kontextwechsel "teuer" (aufwendig)
 - Vorteil: Sicherheit
 - Nachteil: kein (sinnvoller) Zugriff auf fremde Kontexte, Behinderung der Kooperation zwischen Prozessen

- ▶ **Threads** sind "leichtgewichtige Prozesse" mit sehr geringem Kontext
 - **schneller** zu erzeugen
 - globale **Variablen etc.** des Prozesses sind **für alle** in ihm ablaufenden **Threads sichtbar** und manipulierbar
 - **Änderungen** durch einen Thread sind damit sofort für alle anderen sichtbar (nicht: lokale Kopie wie bei Prozessen)
 - Vorsicht bei nebenläufiger Verwendung des gemeinsamen Speichers!



Threads



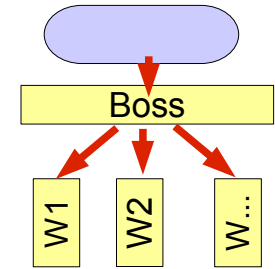
- ▶ Ein **Thread** hat (nur) einen...
 - eigenen **CPU-Kontext**
(Programmzähler und Statusregister und andere CPU-Register)
 - eigenen **Stack**
 - kleinen **privaten Speicherbereich**
- ▶ Folge: **schnelleres Umschalten**, mehr Kooperationsmöglichkeiten, weniger Schutz.
- ▶ Ein **Prozess** kann **mehrere Threads** umfassen.
- ▶ Threading ermöglicht mehrere nebenläufige Programmausführungen **im gleichen (Prozess-)Kontext**.

Nebenläufige Verarbeitungsmodelle (Beispiele)

265

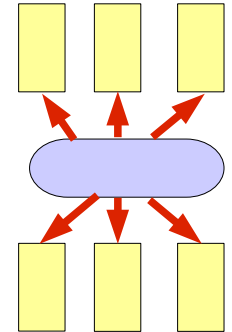
► Boss / Worker

- Boss-Thread verteilt Arbeit auf Worker-Threads
- jeder Worker-Thread arbeitet sein Arbeitspaket ab



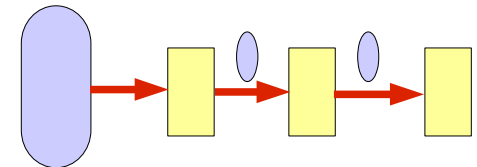
► Aufgaben-Pool

- Aufgaben liegen für alle Threads sichtbar in einem Pool
- Threads holen sich jeweils verschiedene Aufgaben heraus und arbeiten sie ab



► Pipeline

- Threads sind hintereinandergeschaltet
- nehmen Input von ihrem Vorgänger an
- und geben Output an ihren Nachfolger weiter





Thread-Implementierungen

- ▶ Implementierung von Threads systemabhängig:
- ▶ Implementierung durch eine **Benutzerbibliothek**
 - z.B. 4BSD UNIX
 - Threadverwaltung ist komplett im "user space" realisiert
- ▶ Implementierung mit **Unterstützung im Betriebssystem-Kernel**
 - native threads
 - z.B: UNIX System V
 - unabhängig von Benutzerprozessen
- ▶ Verbreiteter Standard: "***pthread***s"
 - pthreads = POSIX Threads
 - Standard-Schnittstelle, die z.B. auch Synchronisationsfunktionen umfasst (Mutexe)
 - verfügbar auf vielen Systemen, z.B. als C-Bibliothek,



Beispiel

```
#include <stdio.h>
#include <pthread.h>
```

```
#define MAX 5
int zaehler = 0;
```

```
void *gruss(void *args) {
    zaehler++;
    printf("%d: Hallole! Zaehler = %d\n", pthread_self(), zaehler);
    return NULL;
}
```

```
int main(void) {
    pthread_t tid[MAX];
    int i;
    for (i=0; i < MAX; i++) {
        pthread_create(&tid[i], NULL, gruss, NULL);
    }
    for (i=0; i < MAX; i++) pthread_join(tid[i], NULL);
    exit(0);
}
```

```
$ gcc pthread.c -lpthread
```

```
$ ./a.out
```

```
16386: Hallole! Zaehler = 1
```

```
32771: Hallole! Zaehler = 2
```

```
49156: Hallole! Zaehler = 3
```

```
65541: Hallole! Zaehler = 4
```

```
81926: Hallole! Zaehler = 5
```

*Thread erzeugen,
Thread-ID in tid[i] ablegen,
Funktion gruss() starten*

ähnlich "wait()" bei Prozessen

pthread Erzeugung

```
#include <pthread.h>
```

```
void *func(void *func_arg) {  
    void ergptr = malloc(sizeof(struct ergtyp));  
    ...  
    if (...) { ...; pthread_exit(ergptr); }  
    ...  
    return ergptr;  
}  
int main(void) {  
    pthread_t tid  
    struct ergtyp *erg;  
    ...  
    pthread_create(&tid, NULL, func, func_arg);  
    ...  
    pthread_join(tid, &erg);  
}
```

Immer Rückgabewerte prüfen!

- ▶ `pthread_create()` führt die Funktion `func` als Thread aus (Funktion muß einen `void*`-Parameter nehmen und `void*` liefern)
- ▶ `func` erhält beim Start `func_arg` (ein `void*`) als Argument
- ▶ `pthread_exit()` beendet Thread (vergleichbar mit `exit()`)
- ▶ `pthread_join()` ähnelt `wait()`, Thread-Ergebnis (`void*`) wird in `&erg` gespeichert (bzw.: `NULL` = Ergebnis egal)



Thread-Synchronisation: Mutexe

269

```
#include <pthread.h>
```

```
pthread_mutex_t mutex;
```

```
void *func(void *func_arg) {
```

```
...
```

```
pthread_mutex_lock(&mutex);
```

```
... kritischer Bereich ...
```

```
pthread_mutex_unlock(&mutex);
```

```
return NULL;
```

```
}
```

```
int main(void) {
```

```
pthread_t tid
```

```
pthread_create(&tid, NULL, func, NULL);
```

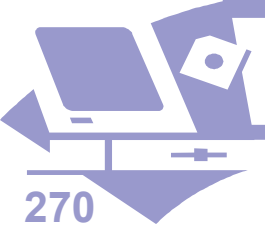
```
...
```

auf mutex-Freigabe warten,
danach selbst sperren

mutex freigeben

Immer Rückgabewerte prüfen!

- ▶ Mutexe sorgen für gegenseitigen Ausschuß,
- ▶ einfach anzuwenden (siehe Beispiel)
- ▶ (Implementierbar als *binäre* Semaphore)



pthread_detach()

```
#include <pthread.h>

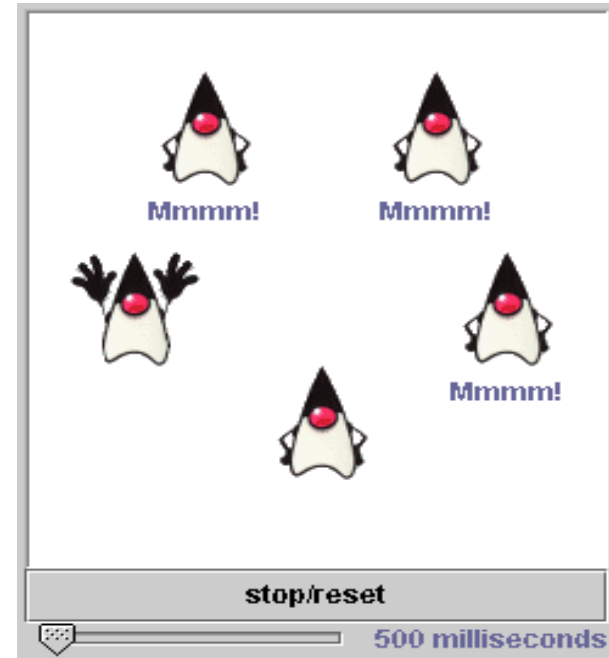
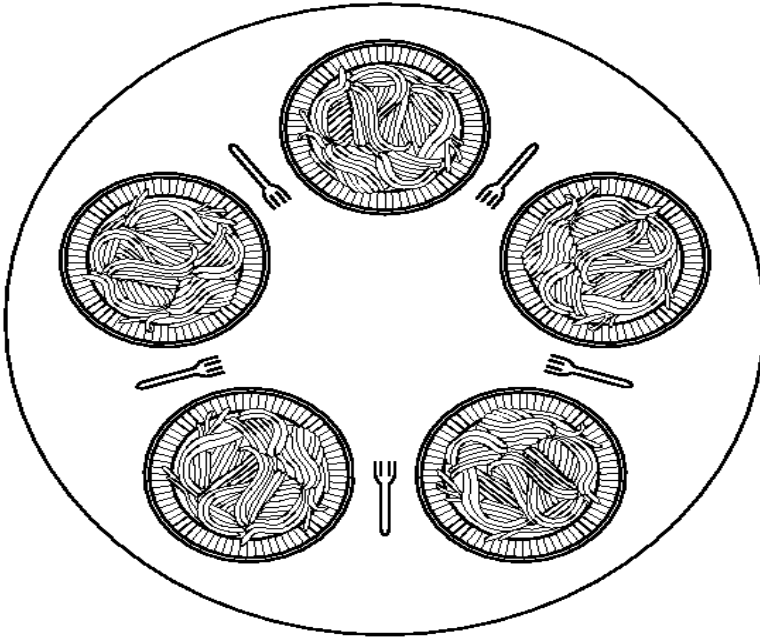
void *func(void *func_arg) {
    ...
}

int main(void) {
    pthread_t tid
    ...
    pthread_create(&tid, NULL, func, parameter);
    pthread_detach(tid);
    ...
}
```

Immer Rückgabewerte prüfen!

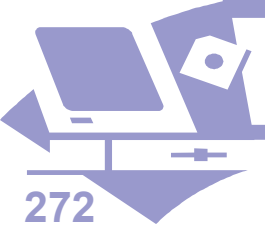
- ▶ `pthread_detach()` versetzt den angegebenen Thread in einen unabhängigen, losgelösten Zustand (*detached state*)
 - bei Thread-Ende werden die Speicher-Ressourcen des Threads sofort freigegeben, es ist dazu
 - kein `pthread_join()` vom Hauptthread aus nötig (bzw. möglich), und damit auch
 - kein Abholen eines Rückgabe-Wertes mit `pthread_join()` möglich

Anwendung: Philosophen-Problem



<http://java.sun.com/docs/books/tutorial/essential/threads/deadlock.html>

- Ursprung: Dijkstras "dining philosophers"-Problem (1965)
 - 5 fernöstliche Philosophen sitzen an einem runden Tisch
 - **Zwischen je zwei** Tellern liegt **jeweils ein** Eßstäbchen
 - Jeder Philosoph isst und denkt abwechselnd
 - Zum Essen werden **zwei Stäbchen benötigt**,
 - nach dem Essen beide Stäbchen wieder zurückgelegt.



Lösungsansatz 1

```
#define N 5

void philosoph(int i) {
    while (1) {
        denken();
        staebchen_nehmen(i);
        staebchen_nehmen( (i+1) % N );
        essen();
        staebchen_zuruecklegen(i);
        staebchen_zuruecklegen( (i+1) % N );
    }
}
```

- Falls alle Philosophen gleichzeitig ihr `staebchen_nehmen(i)` ausführen, blockieren **alle** bei `staebchen_nehmen((i+1)%N)`
- **"Deadlock"** (Verklemmung): alle warten aufeinander („nichts geht mehr“)



Lösungsansatz 2

- ▶ Idee: nach Aufnehmen des ersten Stäbchens prüfen, ob zweites verfügbar ist; falls nein: erstes zurücklegen
- ▶ Vermeidet Deadlock,
- ▶ aber wenn **alle** Philosophen **gleichzeitig** das erste Stäbchen aufnehmen (und wieder ablegen usw.), kommt auch hier keiner weiter.
- ▶ Solche eine Situation heißt **Starvation** (Verhungern) ("endlose" Ausführung, aber ohne Fortschritt)

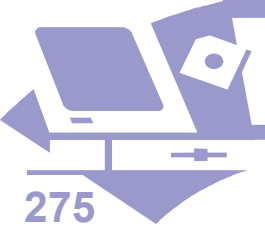


Lösungsansatz 3

```
#define N 5

void philosoph(int i) {
    semaphore mutex;
    while (1) {
        denken();
        P(&mutex);
        staebchen_nehmen(i);
        staebchen_nehmen( (i+1) % N );
        essen();
        staebchen_zuruecklegen(i);
        staebchen_zuruecklegen( (i+1) % N );
        V(&mutex);
    }
}
```

- ▶ Semaphore schützt gesamten "Essens-Abschnitts"
- ▶ Keine Deadlocks, aber: Es kann immer **nur ein Philosoph gleichzeitig** essen, unnötige **Einschränkung von Nebenläufigkeit**



Lösungsansatz 4 (Teil 1)

```
#define N 5

enum { DENKT, HUNGRIG, ISST };
int zustand[N];
semaphore mutex = 1, sema[N];

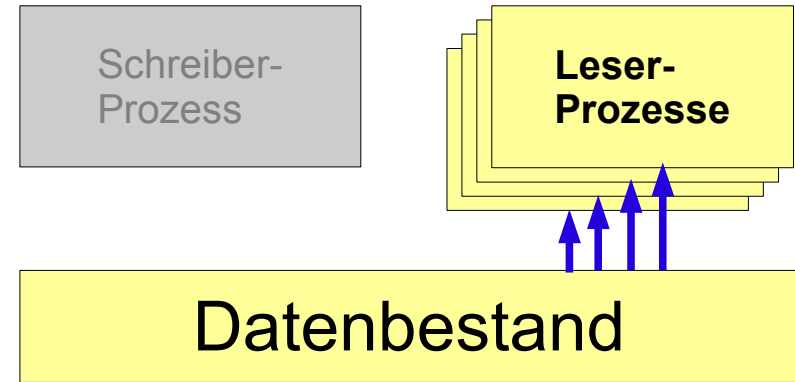
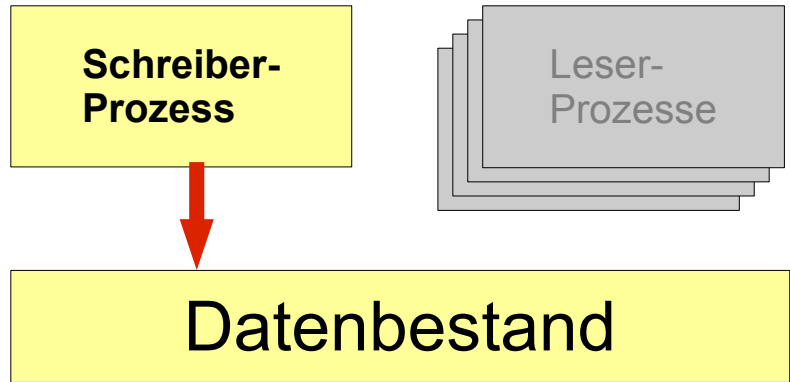
void philosoph(int i) {
    while (1) {
        denken();
        beide_staebchen_nehmen(i);
        essen();
        beide_staebchen_zuruecklegen(i);
    }
}
...
```

Lösungsansatz 4 (Teil 2)

```
27 void beide_staebchen_nehmen(int philo) {
    P(&mutex);
    zustand[philo] = HUNGRIG;
    teste(philo);
    V(&mutex);
    P(&sema[philo]);
}
/* Hunger zeigen */
/* 2 Staebchen verfügbar? */
/* blockieren, falls noch keine
   Staebchen verfügbar */
void beide_staebchen_zuruecklegen(int philo) {
    P(&mutex);
    zustand[philo] = DENKT;
    teste( (philo-1) % N );
    teste( (philo+1) % N );
    V(&mutex);
}
/* fertig mit Essen */
/* kann linker Nachbar essen? */
/* kann rechter Nachbar essen? */
void teste(int philo) {
    if (zustand[philo] == HUNGRIG
        && zustand[(philo-1)%N] != ISST
        && zustand[(philo+1)%N] != ISST) {
        zustand[philo] = ISST;
        V(&sema[philo]);
    }
}
```



Leser-Schreiber-Problem



- ▶ Zu jedem Zeitpunkt dürfen **entweder** (möglicherweise **mehrere**) **Leser** **oder genau ein Schreiber** auf einen Datenbestand zu.
- ▶ Verboten: gleichzeitiges Schreiben und Lesen.
- ▶ Wie stellt man diese Zugriffsbedingung sicher?



leser()

```
semaphore mutex = 1, db = 1;  
int nLeser = 0;
```

```
void leser(void) {
```

```
    while (1) {
```

```
        P(&mutex);
```

```
        nLeser++;
```

```
        if (nLeser == 1) P(&db);
```

```
        V(&mutex);
```

```
        datenbestand_lesen();
```

```
        P(&mutex);
```

```
        nLeser--;
```

```
        if (nLeser == 0) V(&db);
```

```
        V(&mutex);
```

```
        gelesene_daten_verarbeiten();
```

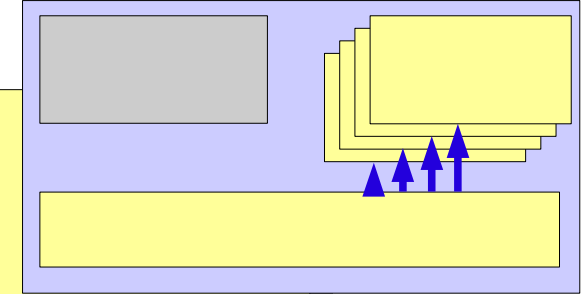
```
    }
```

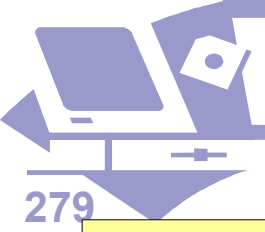
```
}
```

```
...
```

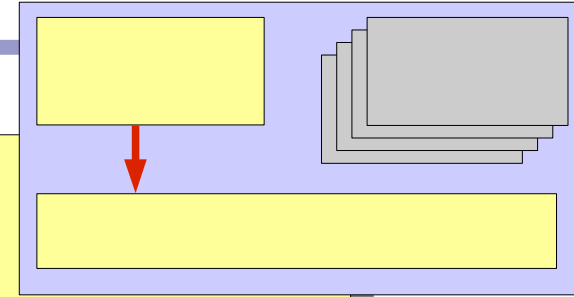
```
/* erster Leser  
   reserviert DB */
```

```
/* letzter Leser gibt  
   DB wieder frei */
```





schreiber()



```
void schreiber(void) {
    while (1) {
        daten_bereitstellen();

        P(&db);          /* exklusiven Zugriff auf
                           Datenbank anfordern */
        daten_schreiben();
        V(&db);          /* freigeben */
    }
}
```

- ▶ mutex sichert Zugriff auf Leser-Zähler,
- db sichert den Zugriff auf den Datenbestand für
 - genau **einen Schreiber** bzw.
 - beliebig **viele Leser** (der erste sperrt, der letzte gibt frei)
- ▶ Leser werden bevorzugt; im "Lese-Modus" erhält jeder neu hinzukommende Leser sofort Zugriff;
- ▶ Problem: wartender Schreiber kommt ("beliebig lang") nicht zum Zuge, solange noch mindestens ein Leser aktiv ist.



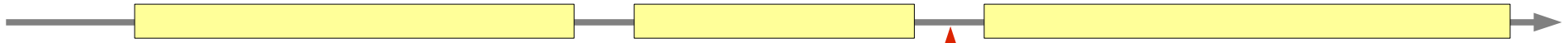
Scheduling

- ▶ Der **Scheduler** ist die Betriebssystemkomponente, die das Umschalten der realen CPU zwischen den Prozessen plant.
- ▶ Technisch durchgeführt wird das Umschalten (Kontextwechsel) vom sogenannten **Dispatcher**.
- ▶ Dazu implementiert er einen **Scheduling-Algorithmus**.
 - **preemptiv** - laufender Prozess kann suspendiert (verdrängt) werden
 - **non-preemptiv** - ein einmal gestarteter Prozess läuft bis er endet oder sich selbst blockiert.
- ▶ Scheduling-Verfahren mit Prozess-**Prioritäten**:
 - **statisch** - Prioritäten ändern sich bei Bearbeitung nicht
 - **dynamisch** - Prioritäten können sich verändern



Prozessverhalten

Rechenintensiver Prozess



*Warten auf
Ein-/Ausgabe-
Operationen*

Ein-/Ausgabe-intensiver Prozess





Ziele für Scheduling-Algo.

28.

► Alle Systeme

- **Fairness** - "faire" CPU-Zuteilung für alle Prozesse
- **Policy Enforcement** - Vorgaben werden eingehalten
- **Balance** - alle Systemkomponenten sind ausgelastet

► speziell für **Stapelverarbeitungssysteme** (batch processing)

- **Durchsatz** - maximiere Jobs/Stunde
- **Turnaround-Zeit** - Zeit Jobstart/-ende minimieren

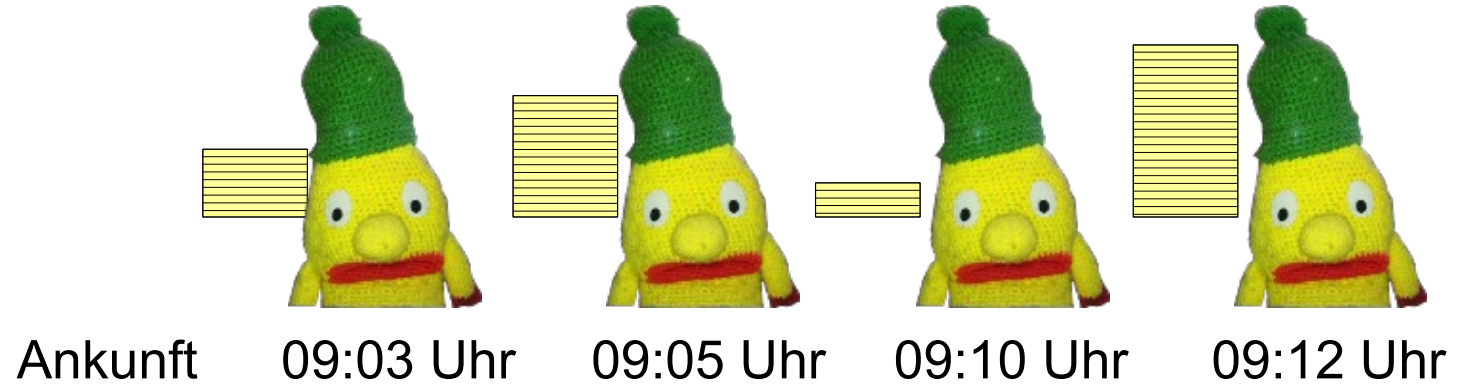
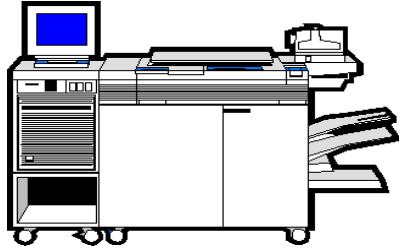
► CPU-Belegung - CPU soll konstant mit Jobs belegt sein

- speziell für **Interaktive** Systeme
- **Antwortzeiten** - schnellstmögliche Reaktion auf Anfragen
- **Proportionalität** - Eingehen auf Nutzerbedürfnisse

► speziell für **Echtzeitsysteme**

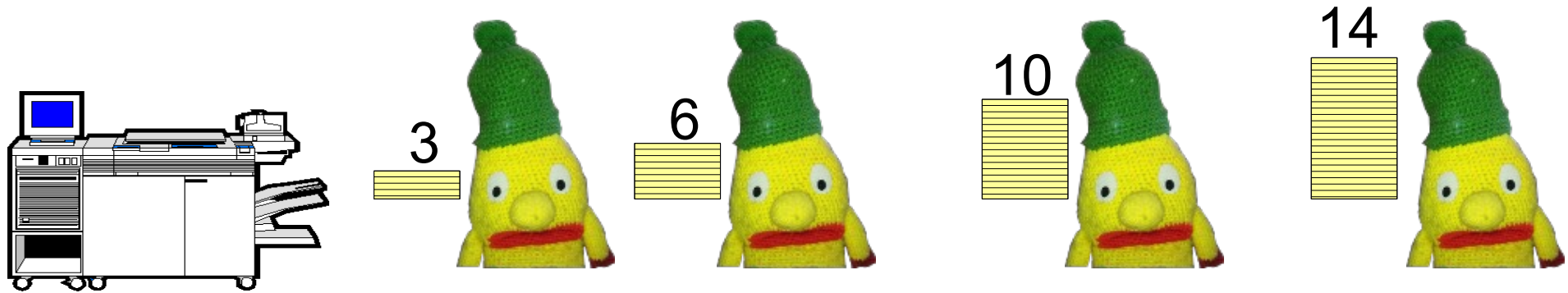
- **Termintreue** - keine Daten verlieren (durch "Verpassen")
- **Vorhersagbarkeit** – Planbares Verhalten

Strategien: First Come - First Served



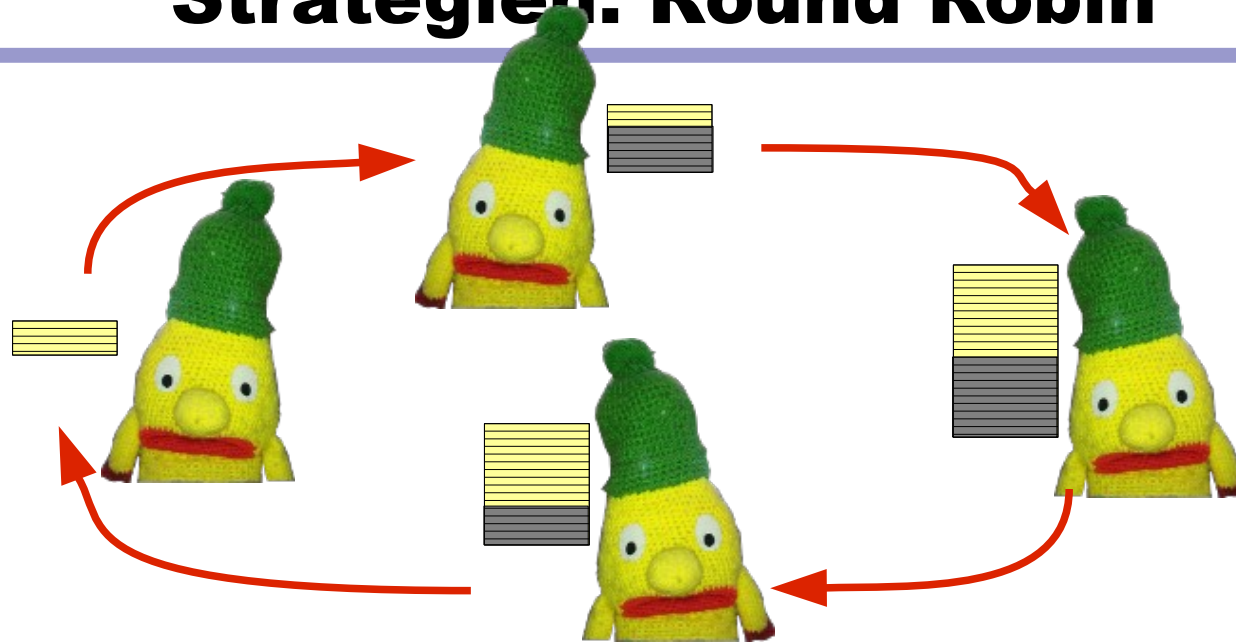
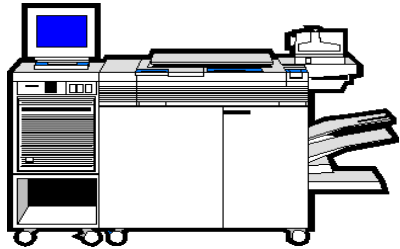
- ▶ Jobs werden in der **Reihenfolge des Eintreffens** abgearbeitet
- ▶ FCFS ist **non-preemptive**
- ▶ "Pech", wenn Langläufer vor kurzem Prozess in der Schlange steht

Strategien: Shortest Job First



- ▶ Von allen rechenbereiten Prozessen wird der mit der **kleinsten Bedienzeitanforderung** ausgeführt (bei Gleichheit: FCFS)
- ▶ Sichert kürzeste mittlere Wartezeit für alle Aufträge
- ▶ Bedienzeit muss vorab bekannt sein (unrealistisch?)
- ▶ Kann unterbrechend (preemptive) und nicht-unterbrechend (non-preemptive) implementiert werden (falls preemptive → Unterbrechung, wenn kürzerer Prozess eintrifft)
- ▶ Bevorzugt kurze Prozesse

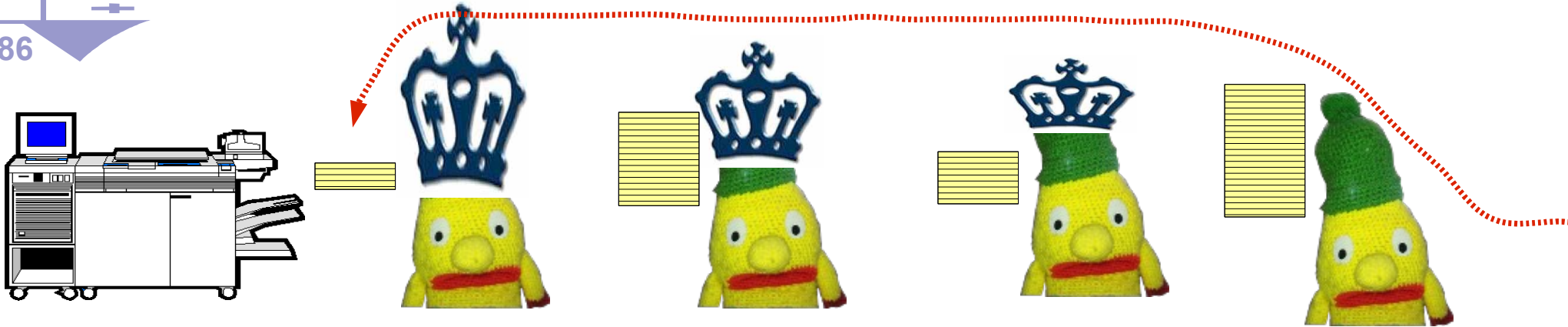
Strategien: Round Robin



- ▶ preemptives Verfahren; Rechenbereite Prozesse werden reihum bedient, wobei jeder maximal eine **festgelegte Zeitspanne** ohne Unterbrechung rechnen darf
("Zeitscheibe", in der Regel im zweistelligen Millisekundenbereich)
- ▶ Wenn ein Prozess blockiert oder endet, erfolgt der Prozesswechsel sofort.
- ▶ Ein langlaufender Prozess benötigt ggf. mehrere "Runden".
- ▶ Bevorzugt Kurzläufer (ohne Bedienzeit vorab zu kennen)

Strategien: prioritätsgesteuert

286



- ▶ Jeder Prozess hat eine **Priorität** (in der Regel kleine Zahl = hohe Priorität), und werden gemäß Priorität abgearbeitet (z.B. „interaktive Prozesse bevorzugen“)
- ▶ neue, höher priorisierte Prozesse verdrängen ggf. niedriger priorisierte
- ▶ Bei einer **dynamischen Variante** des Verfahrens kann sich die Priorisierung im Zeitverlauf ändern, z.B.
 - *Aging*: Priorität steigt mit zunehmendem Alter
 - *Multilevel Queueing*: Eine Warteschlange je Prioritätsklasse, innerhalb einer Klasse z.B. Round Robin; bedient wird stets die höchste nichtleere Klasse



UNIX Scheduling (System V)

- ▶ **Prioritätsgesteuertes** Scheduling mit dynamischen Prio.
- ▶ Eine Warteschlange je Prioritätsstufe für bereite Prozesse, Round Robin innerhalb jeder Prioritätsstufe
- ▶ Jeder Benutzerprozess
 - hat bestimmte (nichtnegative) **Basispriorität**,
 - die vom Benutzer *herabgesetzt* werden kann ("**nice**-Wert")
 - Bei Ausführung wird in regelmäßigen Zeitabständen wird der **CPU-Nutzungszähler** des Prozesses erhöht
 - **Tatsächliche Priorität** (Neuberechnung jede Sekunde)
= **Basispriorität + nice-Wert + CPU-Nutzung**

Kommando "top"

288

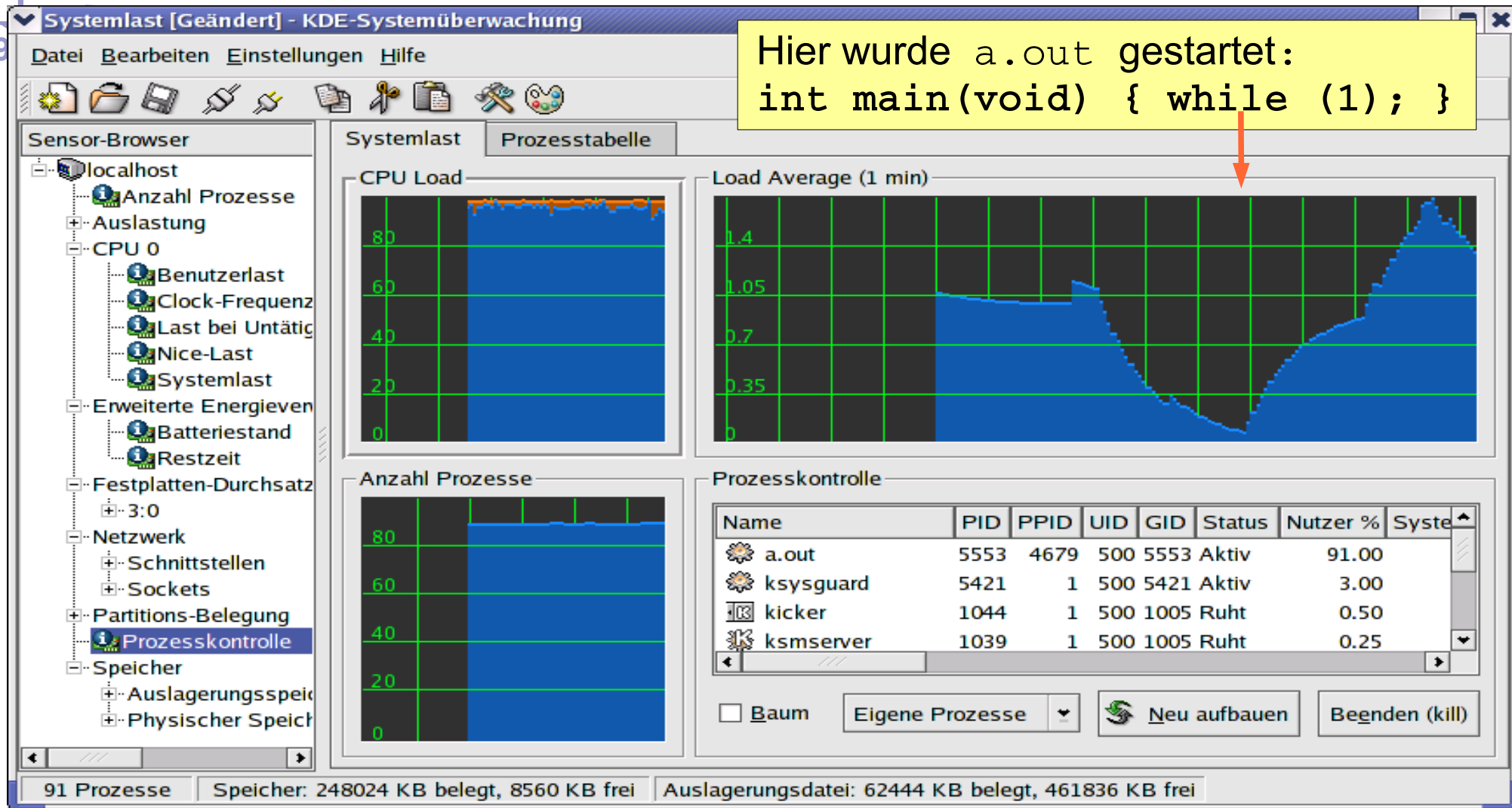
```
Befehlsfenster 2 - Konsole
2:25am up 229 days, 9:52, 1 user, load average: 0.63, 0.18, 0.06
66 processes: 64 sleeping, 2 running, 0 zombie, 0 stopped
CPU0 states: 55.1% user, 0.2% system, 0.0% nice, 44.1% idle
CPU1 states: 44.3% user, 0.0% system, 0.0% nice, 55.1% idle
Mem: 512520K av, 509880K used, 2640K free, 0K shrd, 2440K buff
Swap: 1052216K av, 273356K used, 778860K free 405972K cached

  PID USER      PRI  NI  SIZE  RSS SHARE STAT  %CPU  %MEM  TIME COMMAND
 28957 weitz     14   0   300   300   248 R    99.9   0.0   0:58 a.out
 28958 weitz     10   0  1136  1136   904 R     0.3   0.2   0:00 top
      1 root        9   0    80    64    64 S     0.0   0.0   1:30 init
```

► top zeigt (regelmäßig aktualisiert) unter anderem...

- die top *n* Prozesse mit der höchsten CPU-Belastung,
- Informationen über Anzahl der Prozesse im System,
- CPU- und Speicher-Auslastung sowie
- gleitende Durchschnitte der Rechnerlast über die letzten 1, 5 und 15 Minuten (*load* = Anzahl ausführbarer Prozesse).

Beispiel: KDE KSysGuard



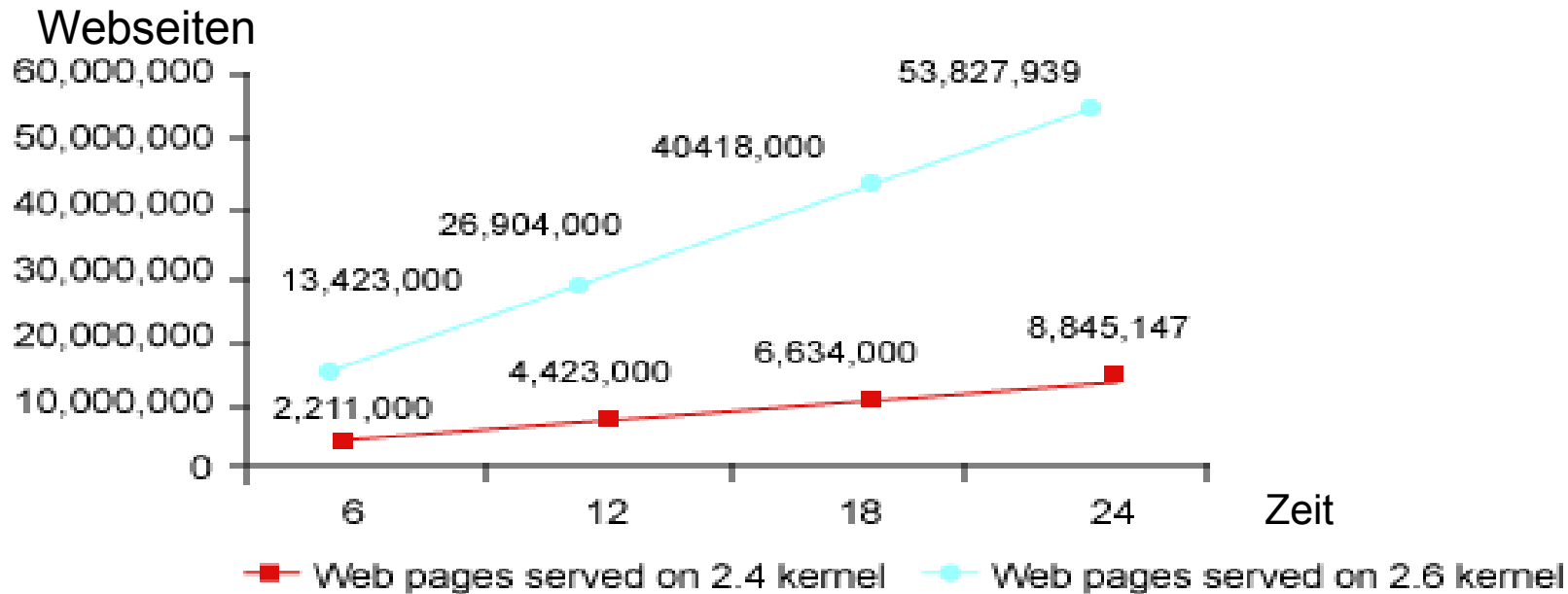
Konfigurierbares, graphisches (Fern-)Systemüberwachungswerkzeug



Scheduler in Linux 2.6

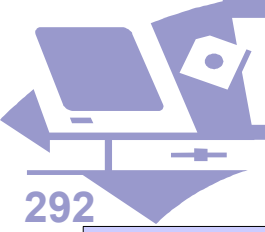
Kernel	Speicher Auslastung	Servierte WebSeiten	Seiten/s	Verarb.zeit/Seite
2.4.18	6.41%	8845.15	102.37	294.44
2.6.0t5	35.96%	53827.94	623.00	57.71

(Durchschnittswerte)

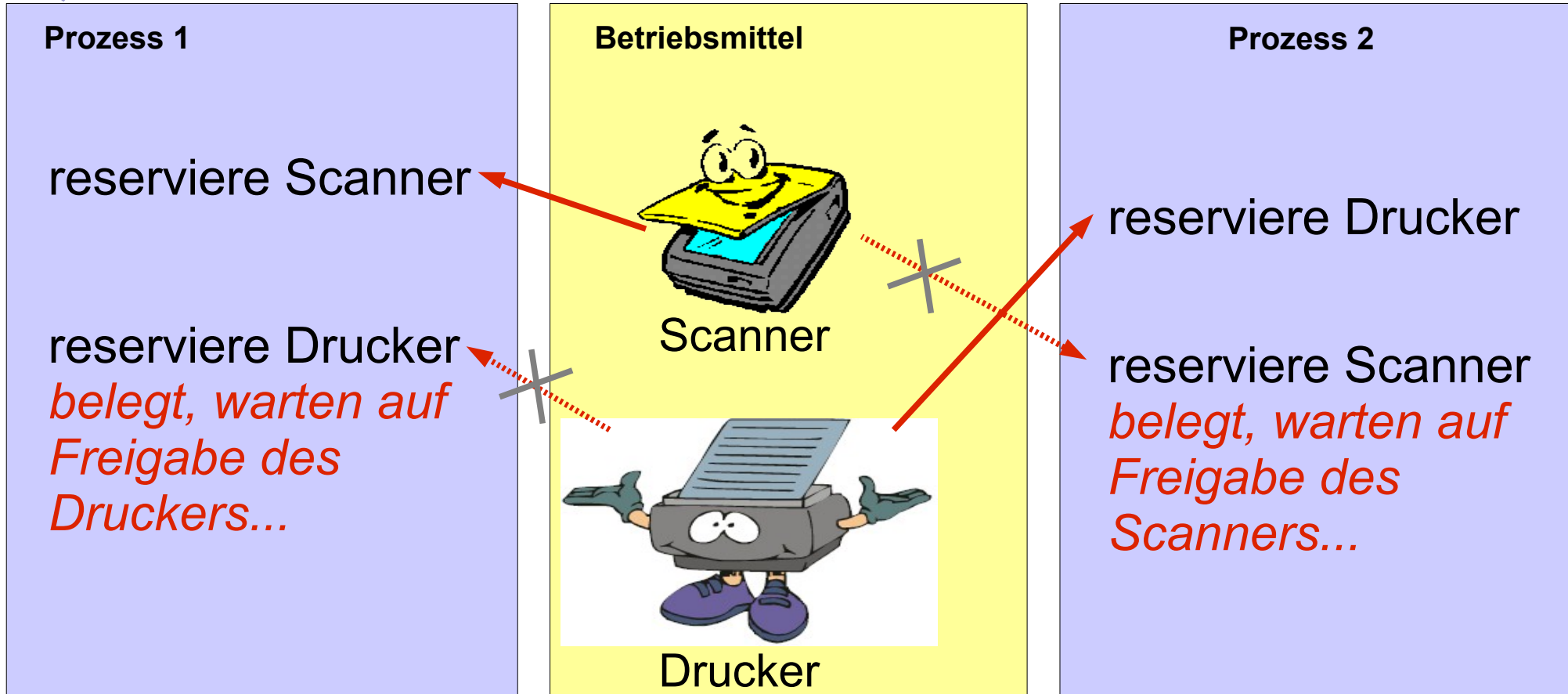




Über das Wesen der Deadlocks



Deadlock-Situation



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



Betriebsmittel

- ▶ Reservierbare Objekte (Objekte, auf die Zugriff erteilt werden kann) heißen **Betriebsmittel**.
- ▶ Diese können **Hard-** oder **Softwarekomponenten** sein:
 - DVD/CD-Brenner
 - Prozessor
 - ein Datensatz
 - eine Verwaltungsstruktur des Betriebssystems
 - ...
- ▶ Ein Betriebsmittel ist **unterbrechbar**, wenn es einem Prozess ohne negative Auswirkungen entzogen werden kann, sonst heißt es "nicht unterbrechbar"
 - **unterbrechbar**: realer Speicher
(Prozeß aus- und später wieder einlagern)
 - **ununterbrechbar**: DVD-Brenner, Drucker



Benutzung eines Betriebsmittels

► Anforderung

- z.B. bei Dateien mit `open()`, Speicher mit `malloc()`, ...
- falls Anforderung gerade nicht erfüllbar: warten
 - "busy waiting": wiederholter Neuversuch oder (besser)
 - Prozess blockieren und bei Verfügbarkeit wecken
- Prozess wird durch **Zuteilung** des Betriebsmittel (vorübergehend?) dessen **Eigentümer**

► Nutzung

- z.B. Datei lesen / schreiben

► Freigabe

- z.B. mit `close()`, `free()`, ...



Voraussetzungen für Deadlocks

Coffman (1971) hat folgende Voraussetzungen gefunden:

- ▶ **Wechselseitiger Ausschluß:**

Jedes Betriebsmittel ist entweder **frei** oder **genau einem** Prozess zugeteilt

- ▶ **"Hold-and-wait"**-Bedingung:

Prozesse können zu bereits reservierten Betriebsmitteln noch **weitere** anfordern

- ▶ **Ununterbrechbarkeit:**

Einmal einem Prozess zugeteilte Betriebsmittel können **nicht** wieder ohne dessen Zustimmung (Freigabe) **entzogen** werden.

- ▶ **Zyklisches Warten:**

Es muss eine **zyklische** Kette von Prozessen geben, in der jeder Prozess auf ein Betriebsmittel **wartet**, das dem nächsten Prozess in der Kette gehört.

Belegungs-Anforderungs-Graphen

296

► Graphische Darstellung der Beziehung von Prozessen zu Betriebsmitteln (Holt, 1972)

► Es gibt zwei Knotentypen:

- **Prozesse**, repräsentiert durch Kreise
- **Betriebsmittel**, repräsentiert durch Quadrate

► Pfeile:

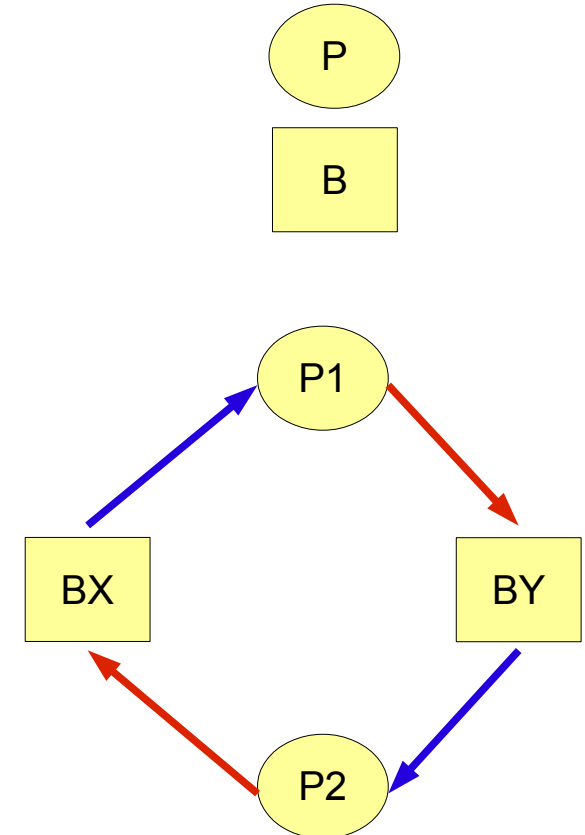
- B ist **vergeben** an P



- P **wartet** auf B



► Zyklus im Graphen: Deadlock





Beispiel

Prozess A

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

Prozess B

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

Prozess C

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

► Gegeben:

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

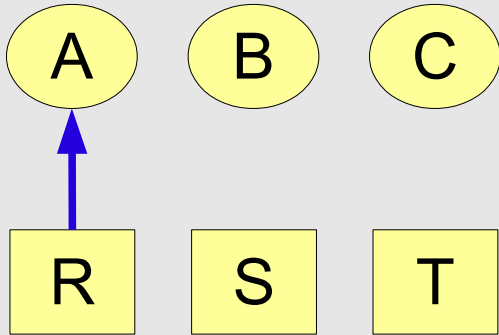
► Das Betriebssystem kann jeden (nicht blockierten) Prozess jederzeit ausführen

► Sequentielle Ausführung von A, B, C wäre unproblematisch

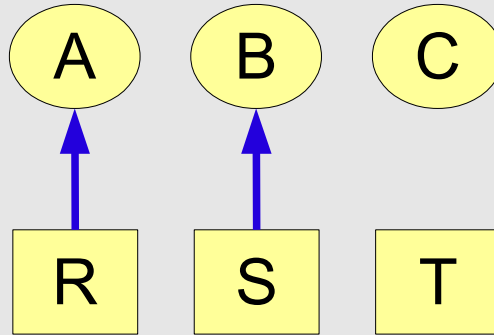
► Wie sieht es bei nebenläufiger Ausführung aus?

Ausführung I

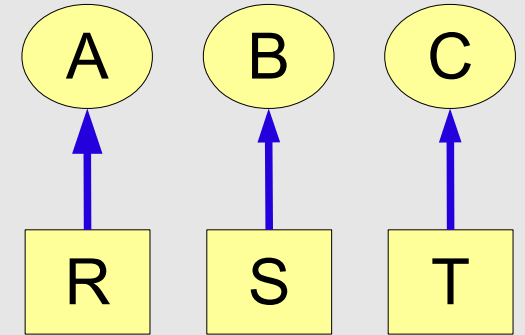
1. A bekommt R



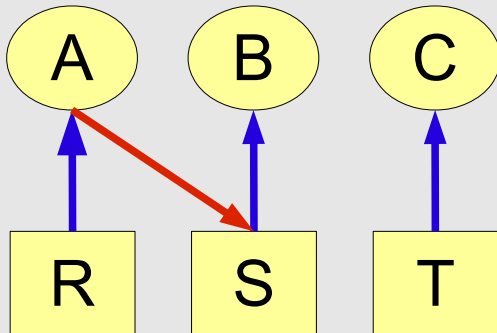
2. B bekommt S



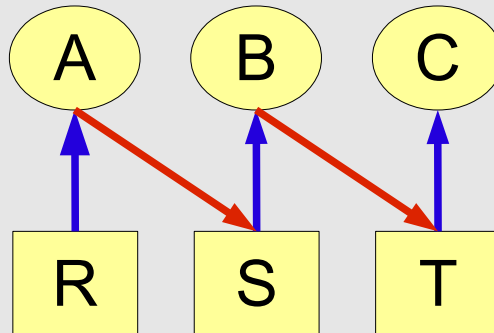
3. C bekommt T



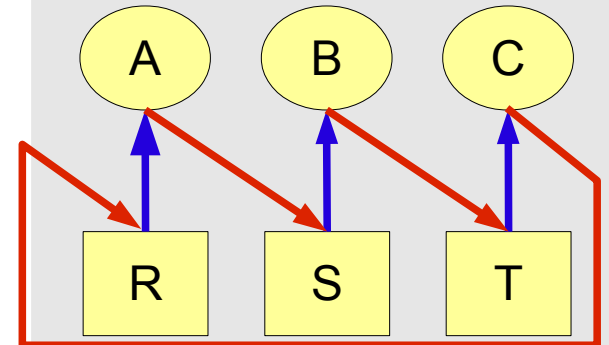
4. A wartet auf S



5. B wartet auf T



6. C wartet auf R
=> **Deadlock**

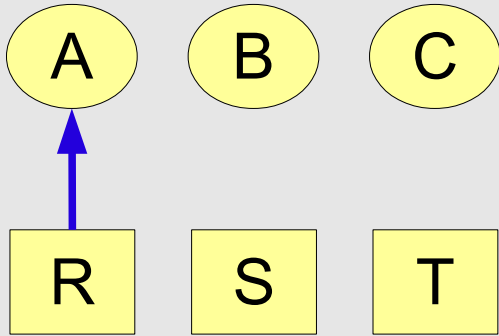




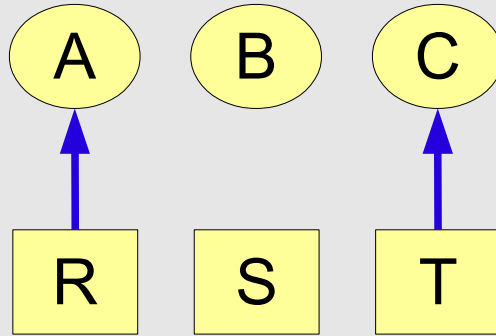
Ausführung II

(B wird zunächst suspendiert)

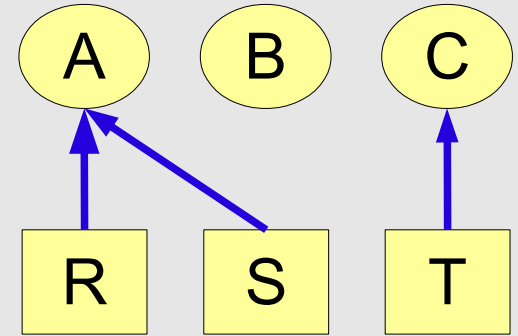
1. A bekommt R



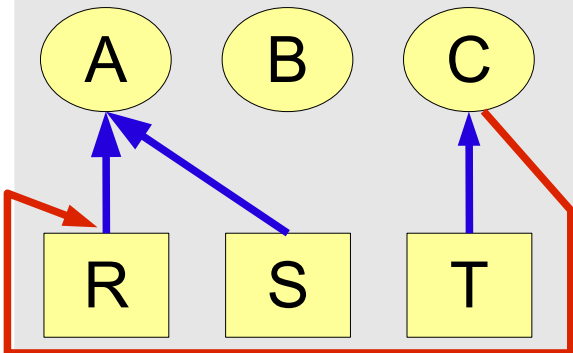
2. C bekommt T



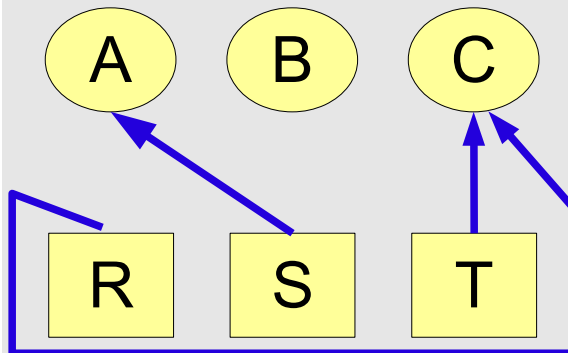
3. A bekommt S



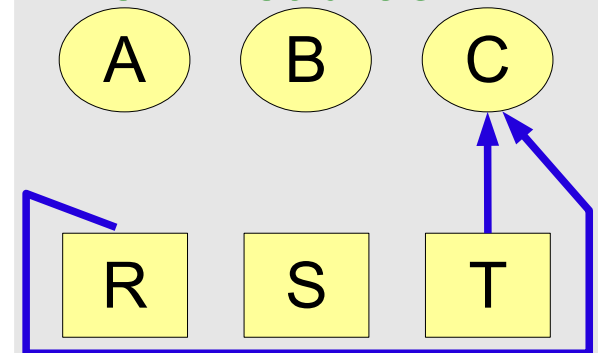
4. C wartet auf R



5. A gibt R frei,
C bekommt R



6. A gibt S frei
kein Deadlock





Verfahren zur DL-Behandlung

- ▶ Mit Belegungs/Anforderungs-Graphen lassen sich Deadlocks erkennen (→ Zyklus im Graph)
- ▶ **Wie weiter verfahren?**
 - **Ignorieren** ("Vogel-Strauß-Verfahren")
 - Deadlocks **erkennen** und **beheben**
 - **Verhinderung** durch Planung der Betriebsmittelzuordnung
 - **Vermeidung** durch Nichterfüllung (mindestens) einer der vier Voraussetzungen für Deadlocks



Vogel-Strauß-“Algorithmus“



- 301
- ▶ Ausdruck optimistischer Lebenshaltung:

"Deadlocks kommen in der Praxis sowieso nie vor" (wirklich?)

- ▶ ...warum also dann Aufwand in ihre Vermeidung stecken?

- ▶ Beispiel:

- UNIX-System mit z.B. 100 Einträge großer Prozesstabelle
- 10 Programme versuchen gleichzeitig, je 12 Kindprozesse zu erzeugen
- Deadlock nach 90 erfolgreichen `fork()`-Aufrufen (wenn keiner der Prozesse aufgibt)

- ▶ Ähnliche Beispiele sind mit anderen begrenzt großen Systemtabellen möglich (z.B. inode-Tabelle)



Deadlocks erkennen

▶ **Einfacher Fall: Ein Betriebsmittel je Betriebsmitteltyp**

▶ **Vorgehen:**

- erzeuge Belegungs-/Anforderungs-Graph
- suche nach Zyklen
- falls ein **Zyklus** gefunden wurde: Deadlock beheben (s.u.)

▶ **Wann** wird die Untersuchung durchgeführt?

- bei **jeder** Betriebsmittelanforderung?
- in **regelmäßigen** Zeitabständen?
- wenn "**Verdacht**" auf Deadlock besteht
(z.B. Abfall der CPU-Auslastung unter eine Grenze)



Beheben von Deadlocks

Wie kann man auf erkannte Deadlocks reagieren?

► Prozessunterbrechung

- Betriebsmittel zeitweise entziehen, anderem Prozess bereitstellen und dann zurückgeben
- Kann je nach Betriebsmittel schwer oder nicht möglich sein

► Teilweise Wiederholung (Rollback)

- System sichert regelmäßig Prozesszustände (Checkpoints)
- Dadurch ist Abbruch und späteres Wiederaufsetzen möglich
- Arbeit seit letztem Checkpoint geht beim Rücksetzen verloren und wird beim Neuaufsetzen wiederholt
(ungünstig z.B. bei seit Checkpoint ausgedruckten Seiten)

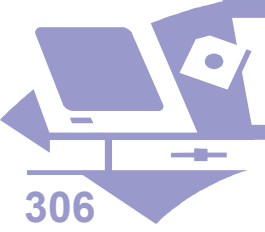
► Prozessabbruch

- Härteste Maßnahme
- Nach Möglichkeit Prozesse auswählen, die relativ problemlos neu gestartet werden können (z.B. Compilierung)



Anderer Ansatz: Verhindern von Deadlocks

- ▶ Bisher: **Erkennung** von Deadlocks, gegebenenfalls "drastische" Maßnahmen zur Auflösung
- ▶ Kann man Deadlocks durch "geschicktes" Vorgehen bei der Betriebsmittelzuteilung **von vornherein verhindern**?
- ▶ Welche **Informationen** müssen dazu **vorab** zur Verfügung stehen?



Betriebsmittelspur

Ablauf von
Prozess B

mögliche *Betriebsmittelspuren* (Beispiele)

Ziel

beide belegen
den Drucker
(unmöglich)

unerreichbar

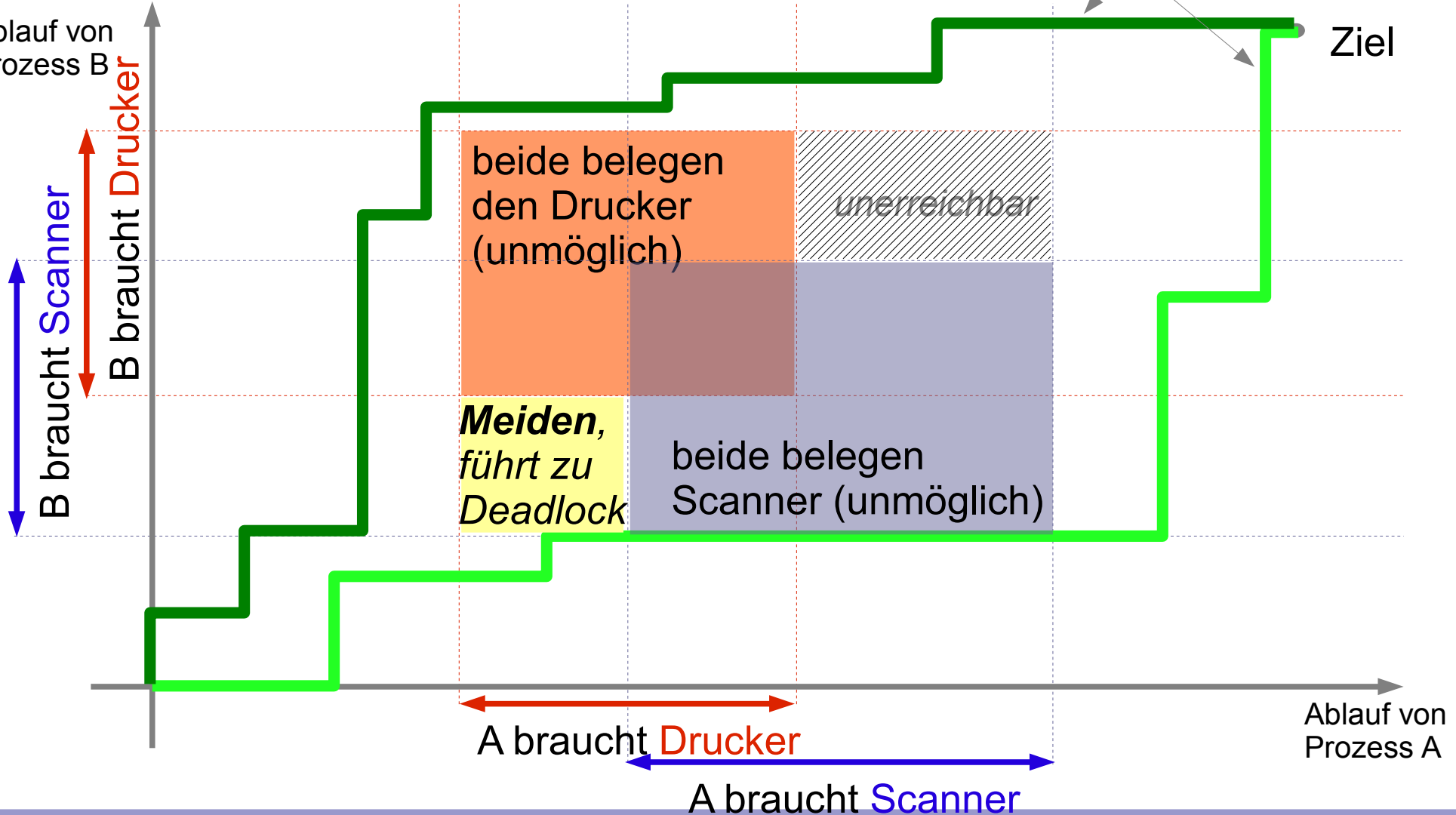
Meiden,
führt zu
Deadlock

beide belegen
Scanner (unmöglich)

A braucht Drucker

A braucht Scanner

Ablauf von
Prozess A





(Un-)Sichere Zustände

- ▶ Ein Systemzustand ist **sicher**, wenn er
 - **keinen Deadlock** repräsentiert und
 - es *mindestens eine* geeignete Prozessausführungsreihenfolge gibt, bei der **alle Anforderungen erfüllt** werden
(die also *auch dann* nicht in einen Deadlock führt, wenn alle Prozesse gleich ihre max. Ressourcenanzahl anfordern)
- ▶ Sonst heißt der Zustand **unsicher**.



Was sagt uns das?

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



Bankier-Algorithmus



► Dijkstra (wer sonst? 1965):

- Ein Bankier kennt die Kreditrahmen seiner Kunden.
- Er geht davon aus, dass nicht alle Kunden gleichzeitig ihre Rahmen voll ausschöpfen werden.
- Daher hält er weniger Bargeld bereit als die Summe der Kreditrahmen.
(er könnte also *nicht alle gleichzeitig im maximalen Umfang bedienen*)
- Gegebenenfalls **verzögert** er die Zuteilung eines Kredits,
bis ein anderer Kunde zurückgezahlt hat.
- Zuteilung erfolgt nur, wenn sie "sicher" ist
(also letztlich alle Kunden bis zu ihrem Kreditrahmen bedient werden können).

- Bankier = Betriebssystem, Bargeld = Betriebsmitteltyp,
Kunden = Prozesse, Kredit = BM-Anforderung,



Bankier-Algorithmus

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



Beispiel

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

noch verfügbare
BM-Einheiten

(a)

3	Prozess	besitzt	max.
A	3	9	
B	2	4	
C	2	7	

Verfügbar: $10 - 3 - 2 - 2 = 3$

(b1)

1	Prozess	besitzt	max.
A	3	9	
B	4	4	
C	2	7	

Verfügbar: $10 - 9 = 1$

(c1)

5	Prozess	besitzt	max.
A	3	9	
B	0	-	
C	2	7	

Verfügbar: $10 - 5 = 5$

(d1)

0	Prozess	besitzt	max.
A	3	9	
B	0	-	
C	7	7	

Verfügbar: $10 - 5 = 0$

(e1)

7	Prozess	besitzt	max.
A	3	9	
B	0	-	
C	0	-	

Verfügbar: $10 - 3 = 7$

(b2)

2	Prozess	besitzt	max.
A	4	9	
B	2	4	
C	2	7	

Verfügbar: $10 - 8 = 2$

(c2)

0	Prozess	besitzt	max.
A	4	9	
B	4	4	
C	2	7	

Verfügbar: $10 - 10 = 0$

(d2)

4	Prozess	besitzt	max.
A	4	9	
B	-	-	
C	2	7	

Verfügbar: $10 - 6 = 4 < 5 (= 9 - 4 \text{ bzw } 7 - 2)$



- ▶ Zustand (a) ist **sicher** (es gibt eine DL-freie Lösung, auch wenn die Prozesse ihren maximalen Bedarf auf einmal abrufen und erst am Ende freigeben)
- ▶ (b2) ist **nicht sicher** (in Schritt (d2) A und C brauchen je 5, frei sind nur 4=>Deadl. möglich)



Beispiele: Sicher?

4 Prozesse, ein Betriebsmitteltyp (10 Stück vorhanden)

verfügbar: 10

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

sicher!

z.B. sequentielle
Ausführung von
A, B, C, D sogar in
beliebiger Reihen-
folge ist möglich

verfügbar: 2

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

sicher!

C ist ausführbar,
(\rightarrow dann $2+2=4$ verfügbar)
dann D, B, A möglich.

verfügbar: 1

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

unsicher!

Differenz „*max - hat*“
immer $>$ verfügbar.

Deadlock, sobald
irgendein Prozess
auf sein Maximum
zugeht (also das eine
verbleibene BM nimmt)



Erweiterter Bankier-Algorithmus

- ▶ Erweiterung: Mehrere Betriebsmitteltypen i , davon E_i -viele vorhanden
- ▶ Prozesse P_1, \dots, P_n

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

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

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

$$\begin{pmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \dots & \dots & \dots & \dots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{pmatrix}$$

Anforderungsmatrix R: Zeile j gibt
maximalen weiteren BM-Bedarf für Prozess j an
("Prozess j fordert R_{jk} Einheiten von BM k ")

$$\begin{pmatrix} R_{11} & R_{12} & \dots & R_{1m} \\ R_{21} & R_{22} & \dots & R_{2m} \\ \dots & \dots & \dots & \dots \\ R_{n1} & R_{n2} & \dots & R_{nm} \end{pmatrix}$$



Erkennungsalgorithmus

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



Beispiel

315

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

Ausführbar ist zunächst nur **P3**

Freigabe von $C_3 = (0 \ 1 \ 2 \ 0)$

$\Rightarrow A = (2 \ 1 \ 0 \ 0) + (0 \ 1 \ 2 \ 0)$

$\Rightarrow A = (2 \ 2 \ 2 \ 0)$

Nun ausführbar: **P2**

(benötigt $R_2 = (1 \ 0 \ 1 \ 0)$)

Freigabe von $C_2 = (2 \ 0 \ 0 \ 1)$

Danach: $A = (4 \ 2 \ 2 \ 1)$

Schließlich auch **P1** ausführbar

Danach: $A = (4 \ 2 \ 3 \ 1)$

Alle Prozesse markiert,
Ausführung ohne Deadlock möglich,
Ausgangszustand „sicher“,



Bankier-Algo praktikabel?

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



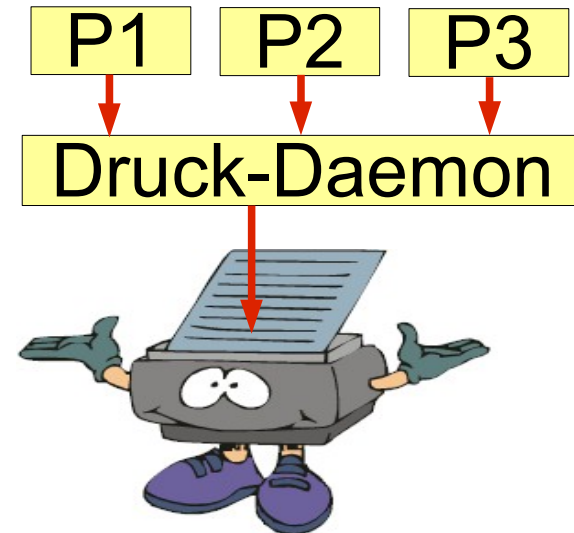
Deadlock-Vermeidung

- ▶ **Deadlock-Verhinderung wenig praktikabel :-)**
- ▶ Alternative: Vermeidung **mindestens einer** der vier Deadlock-Voraussetzungen
 - Wechselseitiger Ausschluss
 - Hold-and-Wait (zu reservierten BM weitere anforderbar)
 - Ununterbrechbarkeit (kein erzwungener BM-Entzug)
 - zyklisches Warten



Wechselseitiger Ausschluß?

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





Hold-and-Wait?

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



Ununterbrechbarkeit?

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



Zyklische Wartebedingung?

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



DL-Vermeidung im Überblick

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

- | | |
|-----------------------------|------------------------------|
| ▶ Wechselseitiger Ausschluß | → z.B. Spooling |
| ▶ Hold-and-wait | → z.B. Preclaiming |
| ▶ Ununterbrechbarkeit | (... <i>besser nicht</i>) |
| ▶ Zyklisches Warten | → z.B. Betriebsmittel ordnen |

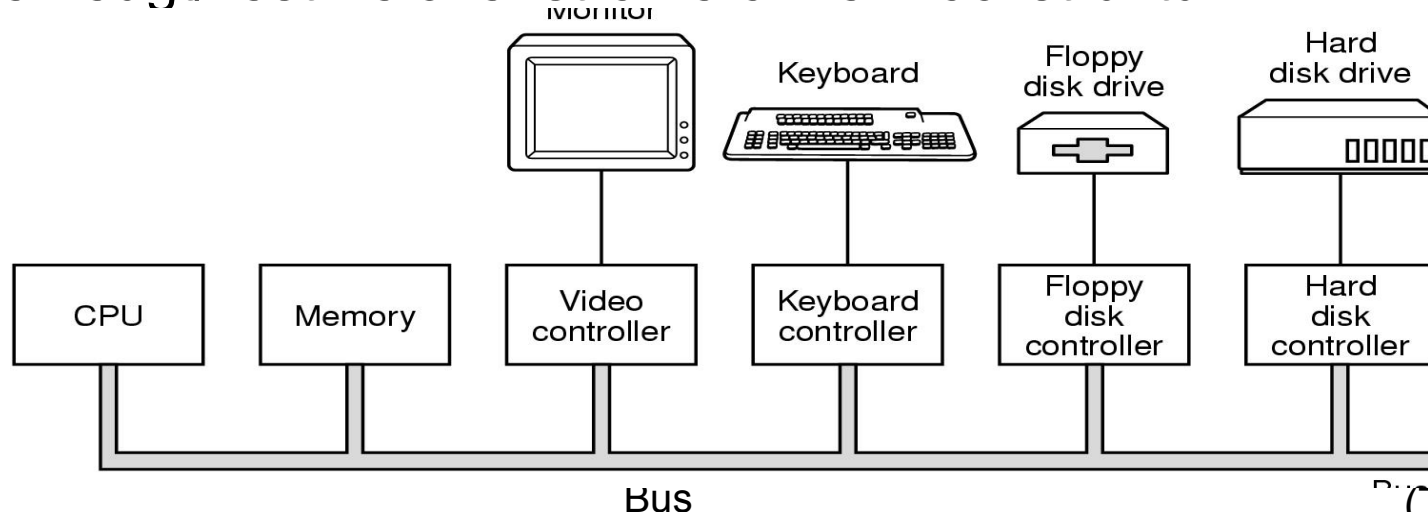


Heute...

Das Ein-/Ausgabesystem

Ein-/Ausgabegeräte

- ▶ Ein-/Ausgabegeräte bestehen oft
 - aus einem **Controller**-Baustein
 - und dem zu steuernden **Gerät**
- ▶ Einteilung im wesentlichen in
 - **blockorientierte** Geräte (z.B. Festplatte)
 - Datenspeicherung in **adressierbaren** Blöcken **fester** Größe
 - **zeichenorientierte** Geräte (z.B. Tastatur, Netzwerkkarte)
 - erzeugt/liest Zeichenströme ohne Blockstruktur



Controller

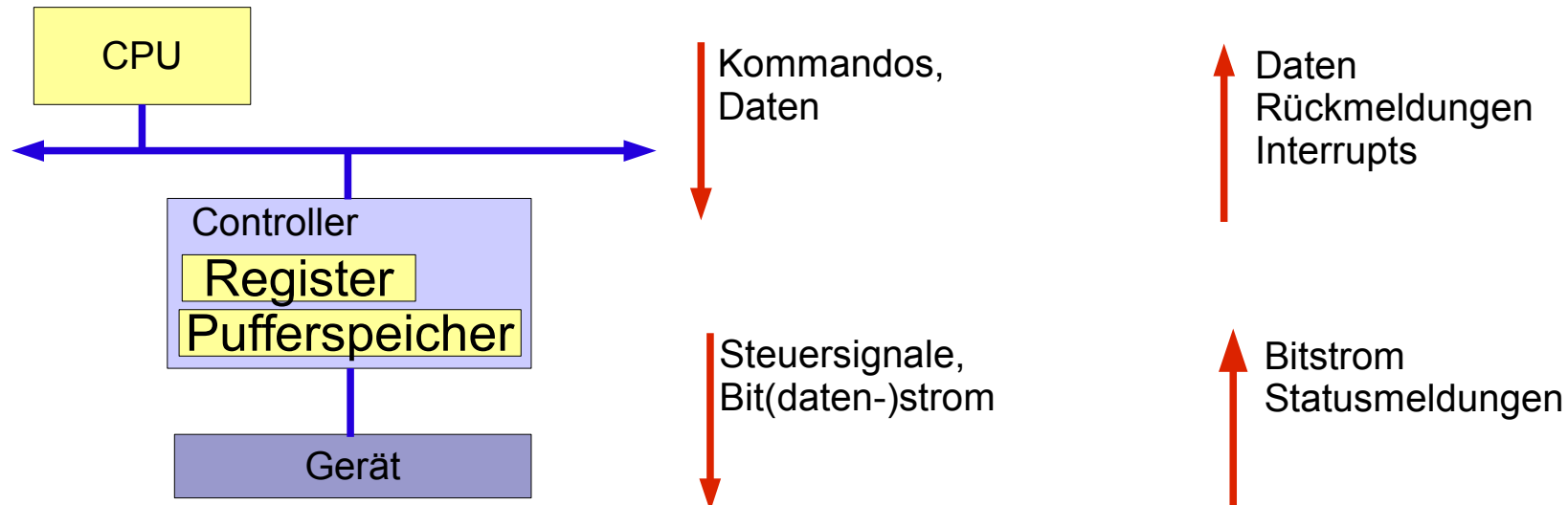
▶ Ein **Controller**

- steuert das zugehörige Gerät direkt an und
- stellt "nach oben" einfachere Schnittstelle zum Gerät bereit

▶ Kommunikation über **Controller-Register**:

- mit **speziellen CPU-Anweisungen** oder
- wie Hauptspeicher-Zugriff ("**memory-mapped I/O**")

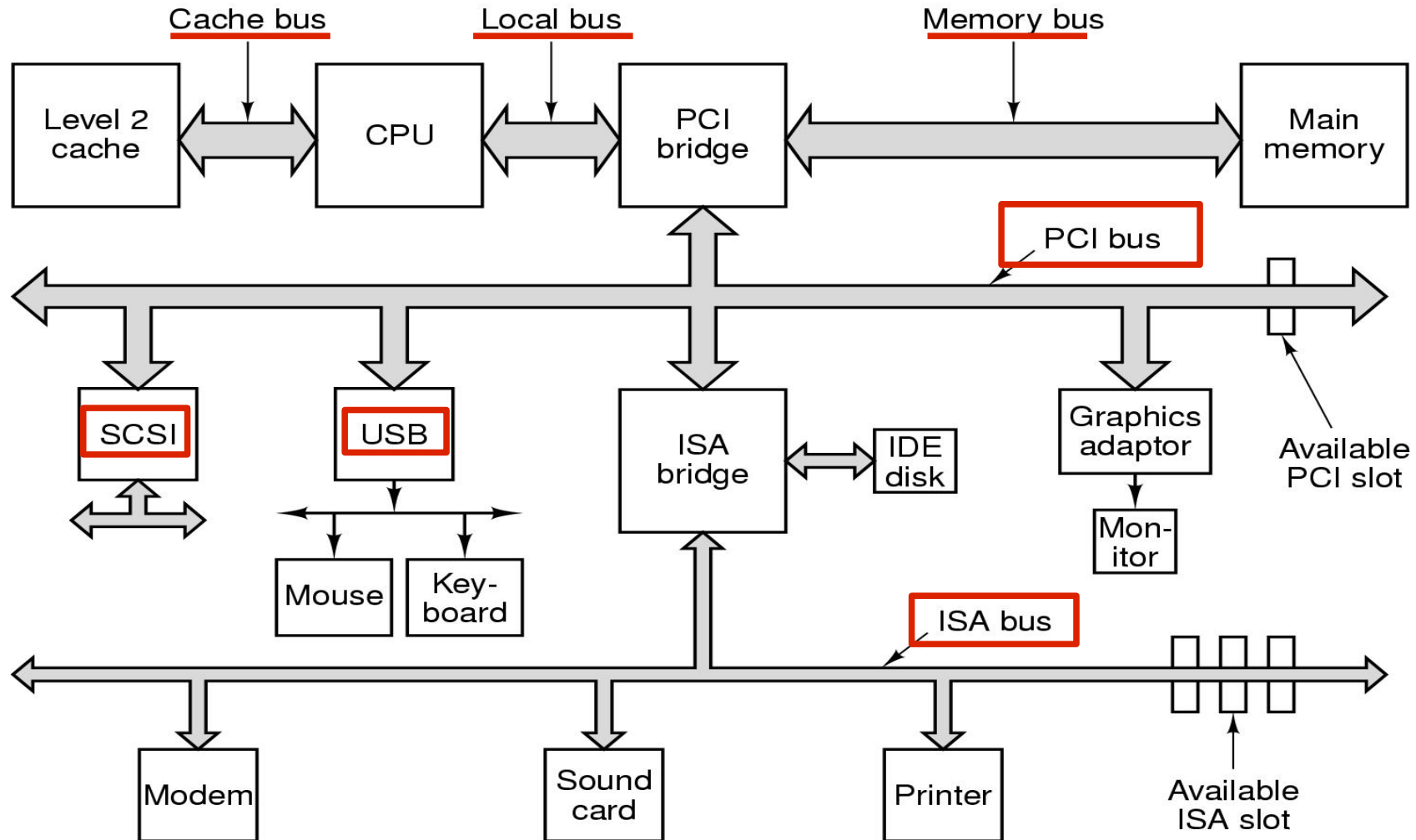
▶ Signalisierung (z.B. "Auftrag erledigt") an CPU: Interrupts





Beispiel: PC-System (etwas antik)

Bus-Systeme: rot





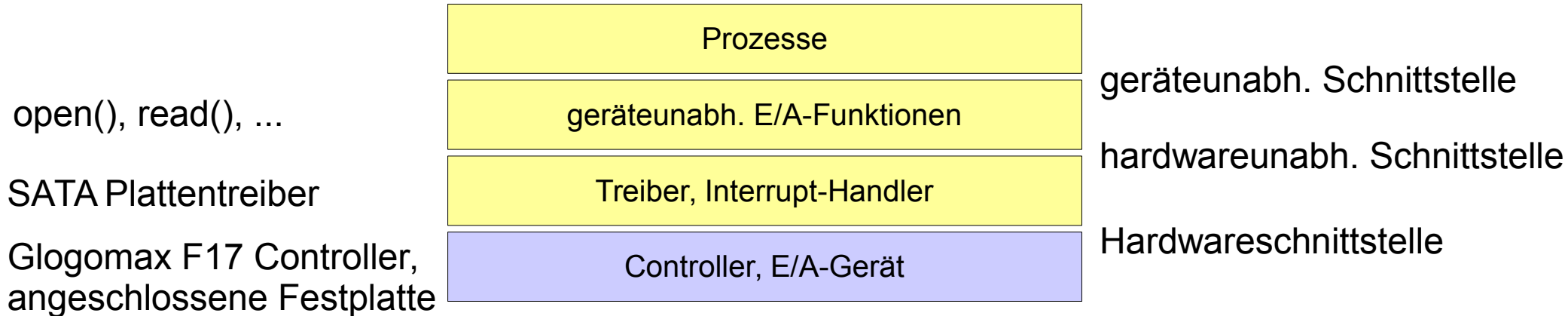
Bus-Systeme im PC-Bereich

- ▶ Interne Erweiterungs-**Steckplätze**
(Festplatten-Controller, Graphikkarten, ...)
 - PCI / PCIe
 - ISA (veraltet)
- ▶ Schnelle (**externe**) **serielle Bus-Systeme**
(Tastaturen, Webcams, externe Platten, ...)
 - USB - Universal Serial Bus (USB 3.2 - bis 20 GBit/s spezifiziert)
 - Thunderbolt, ggf. noch IEEE 1394 ("FireWire")
- ▶ **Anschlüsse für Festplatten, CD-Laufwerke** etc
 - SATA - (serial ATA) neuere, serielle Variante
 - ATA, IDE - früher populäre parallele Schnittstelle im PC-Bereich
 - SAS (Serial Attaches SCSI) - schnelle Platten/SSDs, bis zu 12 Gbit/s (SCSI - frühere parallele Schnittstelle)



(Geräte-)Treiber

- ▶ Gerätetreiber (*device driver*) stellen die unterste Software-Schicht dar und
- ▶ dienen zur Ansteuerung von E/A-Komponenten
(sind also i.a. hardwareabhängig)
- ▶ Ein Treiber verwaltet oft mehrere Geräte (eines Typs)





Typische Treiber-Komponenten

▶ **Autokonfigurations-** und Initialisierungsroutinen

- Überprüfung des Vorhandenseins von Geräten beim Systemstart und ggf. Initialisierung

▶ **E/A-Auftragsbearbeitung**

- Auftrag kann durch Anwendungsprogramm (Systemcall)
- oder durch das virtuelle Speichersystem ausgelöst werden.
- Oft synchron zum Auftraggeber hin (blockiert im Treiber)

▶ **Interrupt-Handling**

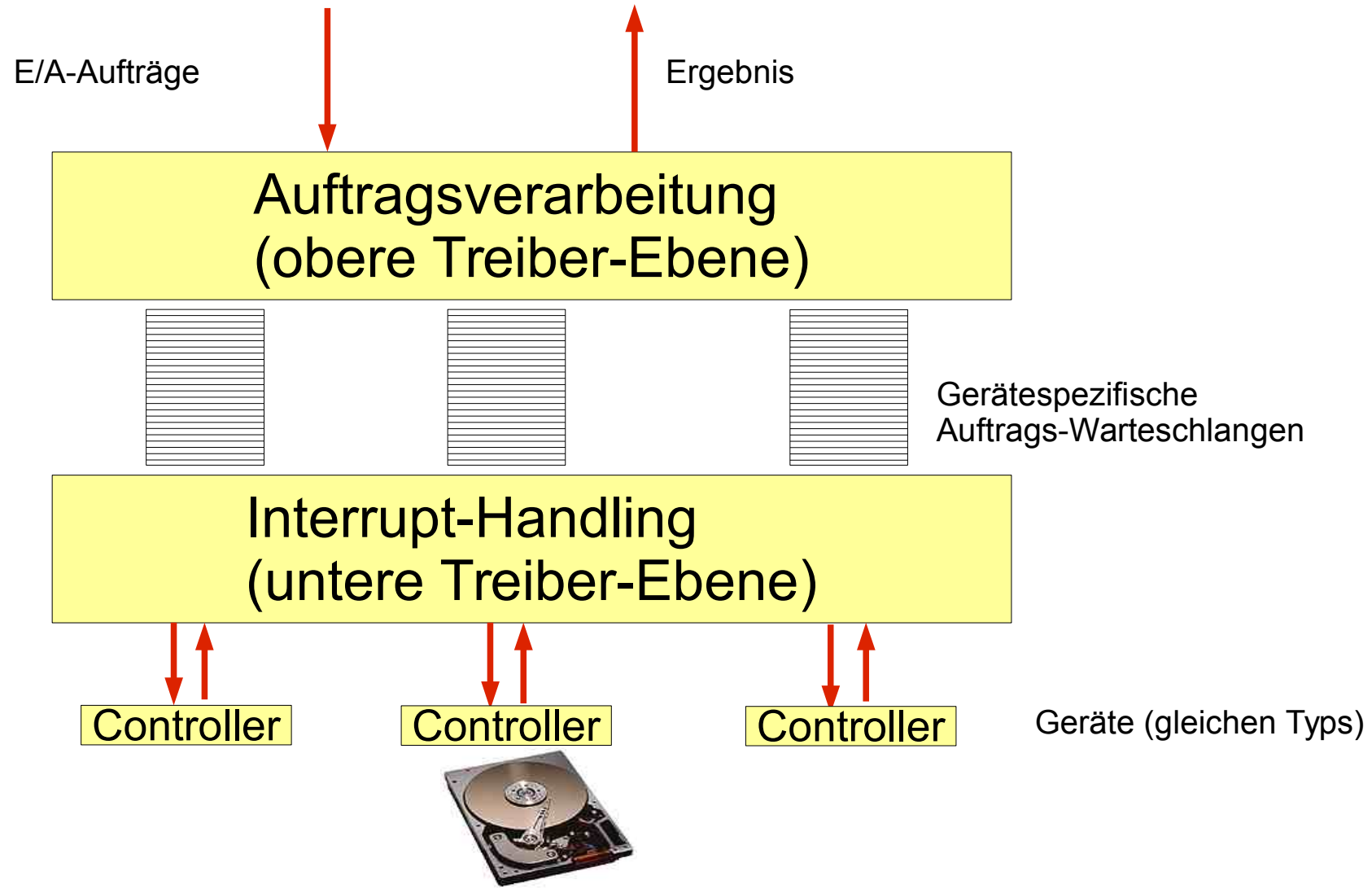
- Treiber ist i.a. asynchron zur Geräteseite hin

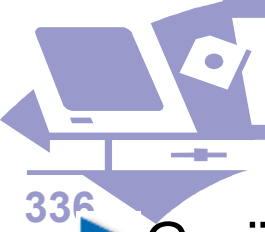
▶ **Geräteabhängige Warteschlangen**

- Aufträge für die vom Treiber verwalteten Geräte



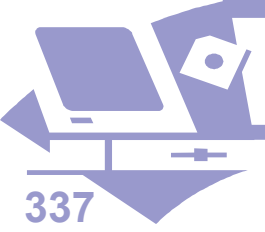
Treiber-Struktur





Hot-Plugging

- ▶ Geräte sollten zur Laufzeit ohne Neustart angeschlossen und abgezogen werden können (hot plugging)
 - USB-Memory-Stick
 - Webcam
 - externe Festplatten (z.B. USB), ...
- ▶ **Hinzufügen** von Geräten:
 - **Identifizieren** des hinzugekommenen Geräts
 - **Nachladen** von Treibern, falls erforderlich
 - Oberhalb der Treiber-Ebene: z.B. Starten einer passenden Applikation oder Auslösen einer "mount"-Operation
- ▶ **Entfernen** von Geräten:
 - Abbruch eventuell laufender E/A-Operationen
 - Evtl. Fehlermeldung an wartende Aufrufer
 - Freigabe reservierter Ressourcen, Konsistenzwahrung



Umsetzung von E/A-Operationen

► Drei gängige Varianten:

- Programmierte Ein-/Ausgabe (PIO)
- Interruptgesteuerte Ein-/Ausgabe
- Direkter Speicherzugriff

► Beispiel: Ausgabe einer Zeichenkette auf einem (Zeilen-)Drucker



Programmierte Ein-/Ausgabe

- ▶ Bei programmierter Ein/Ausgabe (*programmed I/O, PIO*)
 - schreibt die CPU die zu übertragenden Daten schrittweise in das entsprechende Controller-Register
 - und fragt nach jeder Übergabe wiederholt ein Statusregister des Controllers ab, um herauszufinden, wann er wieder empfangsbereit ist (*polling*).
 - Danach wird mit Schritt 1 fortgefahren usw, bis alle Daten übertragen sind.
- ▶ **Vorteil:**
 - einfach
- ▶ **Nachteil:**
 - "Aktives Warten" verschwendet CPU-Zeit



Interrupt-gesteuerte E/A

▶ Nachteil von PIO ist um so größer, je länger das E/A-Gerät für einen (Teil-)Auftrag braucht.

▶ **Beispiel:**

- Drucker mit 100 Zeichen/s
- benötigt 10ms / Zeichen
- ...in der Zeit könnte die CPU viel Gutes tun.

▶ **Daher interrupt-gesteuerter Lösungsansatz:**

- Treiber gibt (Teil-)Auftrag an Controller und blockiert
- Scheduler kann in der Wartezeit einen anderen Prozess rechnen lassen
- Controller erzeugt nach Erledigung einen Interrupt
- Interrupt-Handler gibt nächsten (Teil-)Auftrag an Controller usw.

▶ **Nachteil:**

- schnelle Geräte → viele Interrupts (kosten auch Zeit)

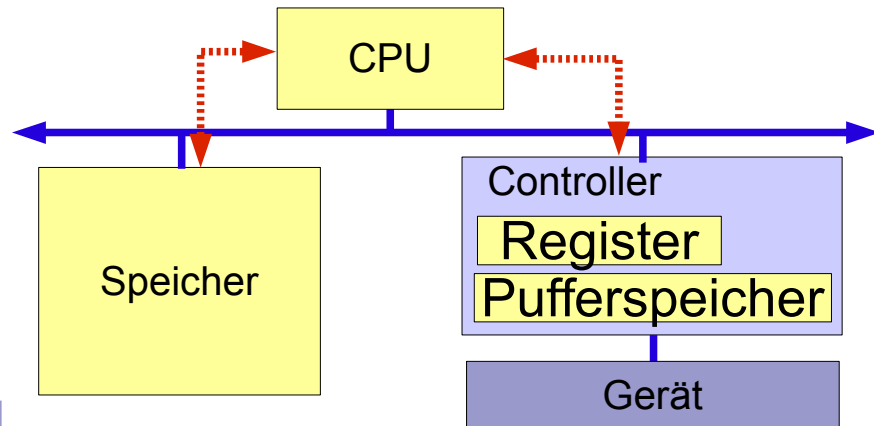
Direct Memory Access (DMA)

► Direkter Speicherzugriff durch Controller:

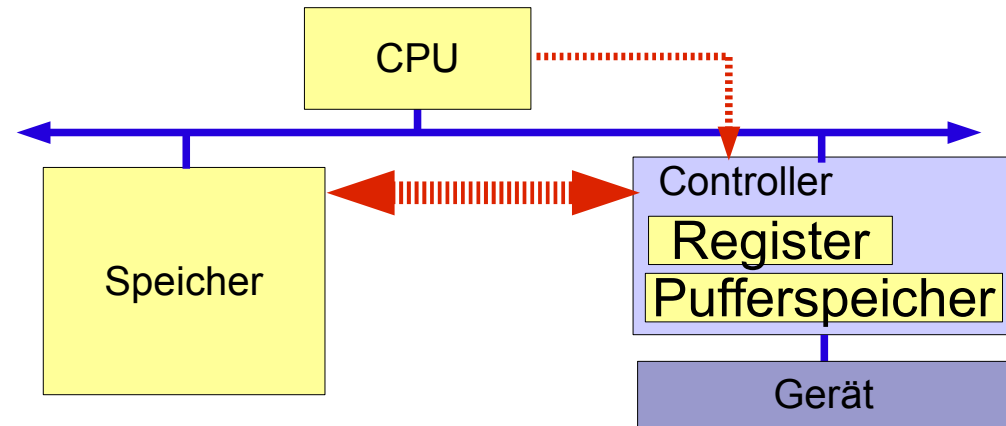
- zu übertragende Daten liegen im Hauptspeicher bereit
- Treiber übermittelt dem Controller lediglich den Anfang und die Länge des Speicherbereichs
- Controller greift dann (ohne CPU) direkt auf diesen Speicherbereich zu (lesend und/oder schreibend)

► Interrupt erst nach Abarbeitung des Auftrags

► Zugriff von CPU / Controller(n) auf Bus ist zu koordinieren.



ohne DMA



mit DMA



Festplattentreiber

- ▶ Ansteuerung von Festplatten- und Diskettenlaufwerken
- ▶ Bekannt aus Abschnitt "Dateisysteme"
 - Armbewegungs-Planung
 - Shortest Seek First
 - Aufzug-Verfahren
- ▶ RAID-Controller



Text-Terminals

▶ Zeichenorientierte (Text-)Terminals

- angeschlossen über serielle Schnittstelle

▶ "raw / cooked mode" für Eingabe

- raw: Zeichen werden wie empfangen durchgereicht
- cooked:
 - Zeilenpufferung: kein Weiterreichen der Eingabe, bis ein Zeilenendezeichen (z.B. <return>) erkannt wurde
 - dadurch: Korrekturmöglichkeiten (Backspace, Zeile löschen)

▶ Ausgabe: Interpretation von speziellen Kommandosequenzen für

- Cursor-Positionierung
- Einstellung von Darstellungsattributen (Farbe, Blinken, ...)
- ...



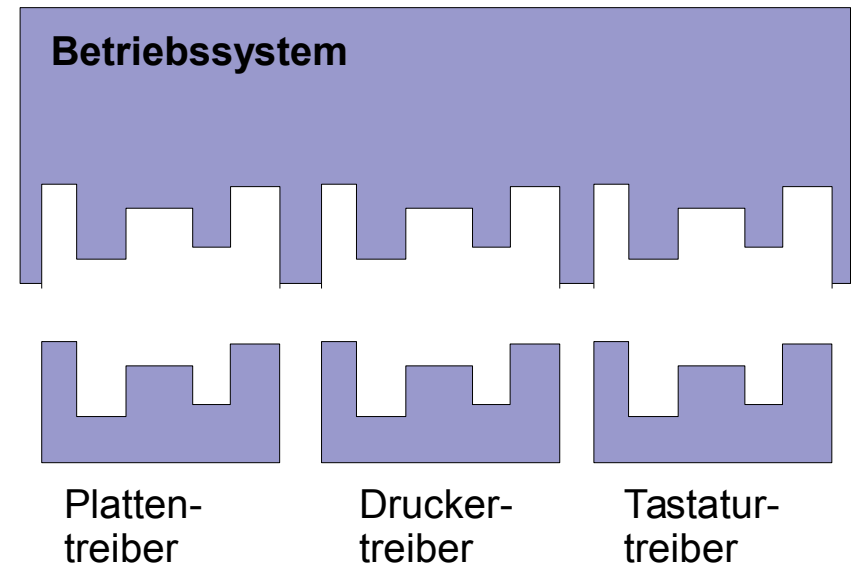
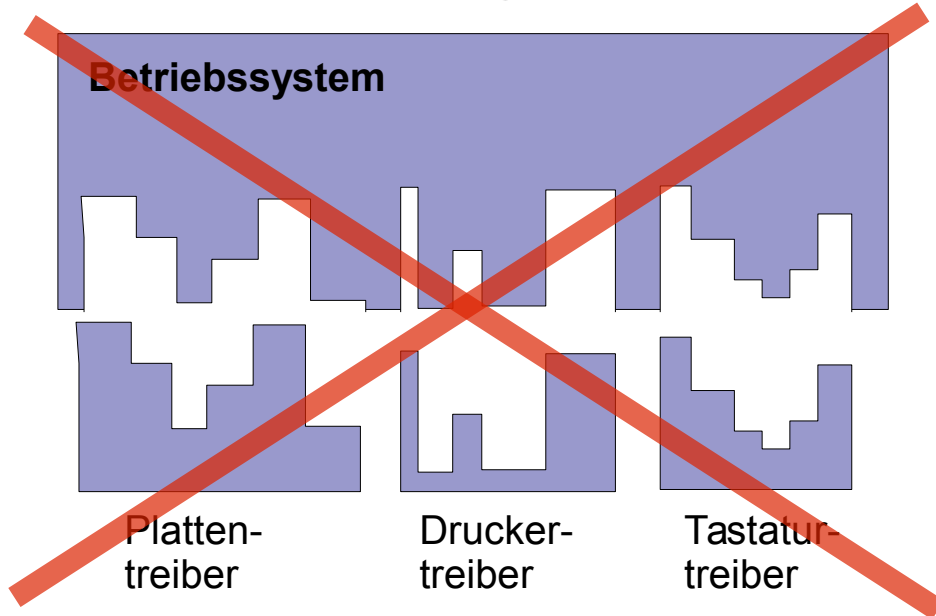
Treiber-Schnittstelle

34

► Wir haben gesehen: Ein Betriebssystem hat es mit vielen, sehr unterschiedliche Geräten zu tun.

► Ziel:

- **Standardisierte** Treiber-Schnittstelle, um Aufwand für die Unterstützung neuer Geräte in Grenzen zu halten
- **Einheitliche Benennung/Handhabung** auf der geräteunabhängigen Ebene ermöglichen



UNIX: Major/Minor Numbers

345

- ▶ E/A-Geräte sind in das Dateisystem eingebunden (/dev/...)
- ▶ Eigentümer / Rechteverwaltung somit wie bei "normalen" Dateien
- ▶ inode-Typ: Gerätedatei
 - statt Verweis auf Datenblöcke: zwei Gerätenummern
 - "major device number": Gerätetyp (z.B. Platte, serielle Schnittstelle, Uhr, ...)
 - "minor device number": Teileinheit (z.B. Partition)
- ▶ Bereits bekannt: Zwei Arten von Gerätedateien:
 - Blockorientierte Geräte (**b**lock devices)
 - Zeichenorientierte Geräte (**c**haracter devices)

major number: z.B. IDE-Festplatte, Terminal
minor number: z.B. Partition, Terminalnr

brw-rw----	1	root	disk	3,	0	30. Jan 11:24	/dev/hda
brw-rw----	1	root	disk	3,	1	30. Jan 11:24	/dev/hda1
brw-rw----	1	root	disk	3,	2	30. Jan 11:24	/dev/hda2
crw-----	1	wwe	uucp	4,	64	10. Jun 10:18	/dev/ttyS0
crw-rw----	1	root	uucp	4,	65	30. Jan 11:24	/dev/ttyS1

UNIX Device Switch Tabellen

346

- ▶ Wenn es eine "einheitliche Treiber-Schnittstelle" gibt:
- ▶ Wie findet das System z.B. die "richtige" `close()`-Implementation für ein konkretes Gerät?
- ▶ **Der Betriebssystem-Kern hält zwei Tabellen**
 - `cdevsw` (*character device switch*) für Zeichengeräte
 - `bdevsw` (*block device switch*) für Blockgeräte
- ▶ **(Zeilen-)Adressierung per *major device number***

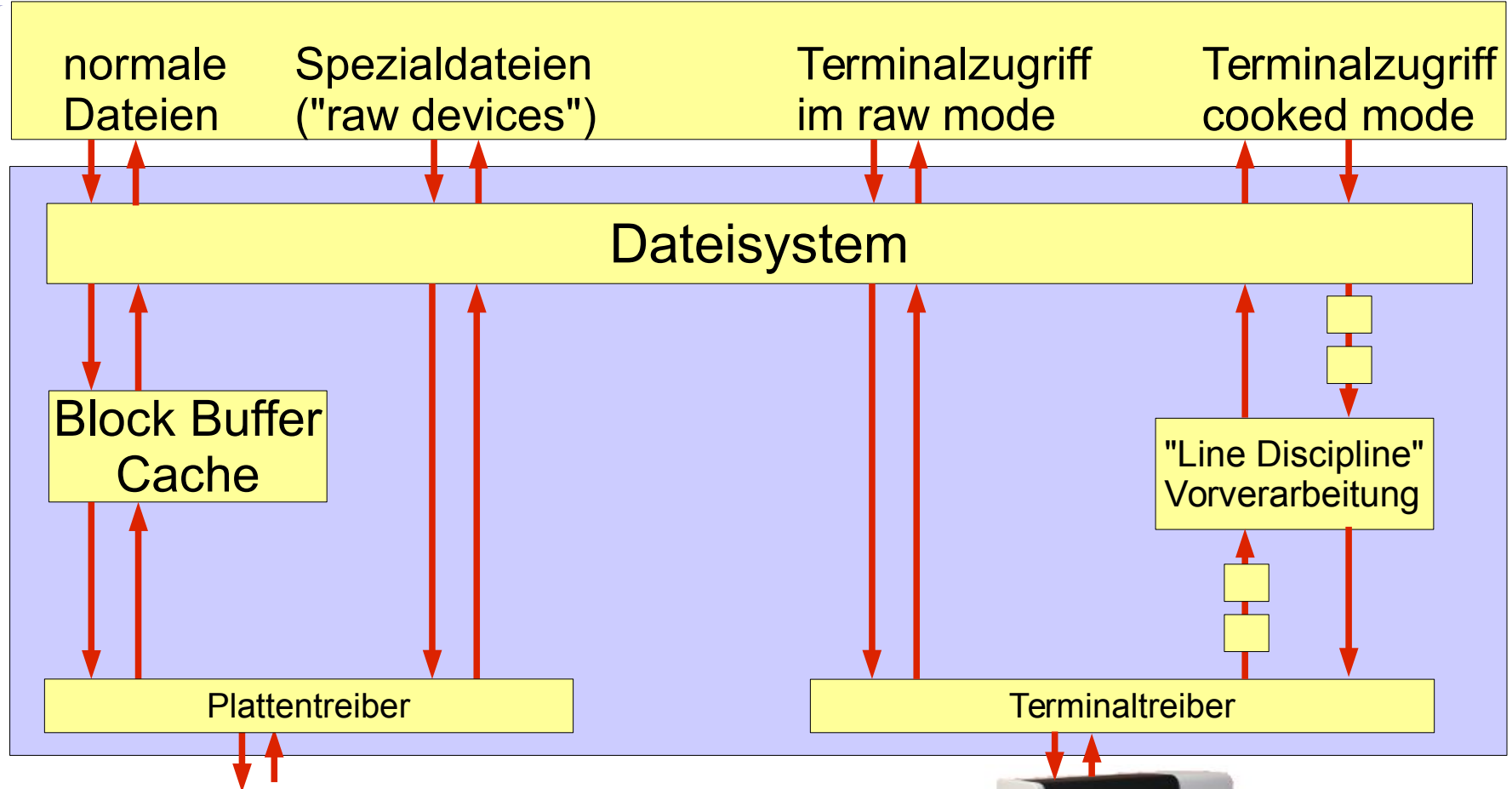
`cdevsw`

	dev		open	close	read	write	ioctl	andere
	1	Speicher	<code>null</code>	<code>null</code>	<code>mem_read</code>	<code>mem_write</code>	<code>null</code>	...
	5	Konsole	<code>tty_open</code>	<code>tty_close</code>	<code>tty_read</code>	<code>tty_write</code>	<code>tty_ioctl</code>	...
▶	6	Drucker	<code>lp_open</code>	<code>lp_close</code>	<code>error</code>	<code>lp_write</code>	<code>lp_ioctl</code>	...

Funktionszeiger auf
`close()`-Implementierung für Drucker
(major device number 6)



UNIX E/A-System





► Früher: Alle notwendigen Treiber mussten im Betriebssystemkern fest incompiliert werden

- Systemausfall-Zeiten, wenn ein Treiber vergessen wurde
- Auch nicht benötigte Treiber waren immer geladen

► Heute: Kernel-Module

- Treiber, aber auch höhere Betriebssystem-Komponenten (Filesysteme etc) im laufenden Betrieb nachladbar und
- entladbar, wenn sie nicht mehr benötigt werden.
- So ist sogar nachträgliches Compilieren und Nachladen ohne Betriebsunterbrechung möglich



Linux: Ladbare Module

- ▶ **Konfiguration / Parametrisierung von Modulen:**
`/etc/modprobe.conf`
- ▶ Herausfinden von **Abhängigkeiten** zwischen Modulen
("wenn A geladen wird, werden auch B,C,D benötigt")
Hilfsprogramm `depmod`
- ▶ Manuelles Laden mit Hilfsprogrammen
`insmod / modprobe`
- ▶ Automatisches Nachladen von Modulen bei Bedarf



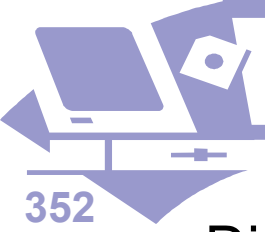
Fehlermeldungen des Kernels

- ▶ Wie gibt eine Betriebssystemkomponente **Fehlermeldungen** aus?
 - Direktes Schreiben in Logfiles
 - Ereignis-Log-Dienst (Windows)
 - `syslog`-Daemon (Unix)
- ▶ Was ist, wenn die dazu benötigten Dienste (Networking, Dateisysteme etc) nicht oder noch nicht verfügbar sind?
 - Systemstart
 - Fehlermeldungen aus Treibern
- ▶ Linux:
 - Kernel hält einen **Ringpuffer** für Fehlermeldungen
 - spezielle Funktion `printk()` zum Schreiben
 - Auslesbar z.B. mit Hilfsprogramm `dmesg`
 - hilfreich auch zum Zurückholen von Boot-Meldungen



Beispielausgabe dmesg (gekürzt)

```
Linux version 2.4.20-mh6 (root@wilhelmus)
Detected 1798.522 MHz processor.
Console: colour VGA+ 80x25
Memory: 256120k/261504k available (1087k kernel code, 4996k reserved,...
ttyS00 at 0x03f8 (irq = 4) is a 16550A
eth0: Intel Corp. 82801CAM (ICH3) PRO/100 VE (LOM) Ethernet Controller,
      00:09:6B:7A:7F:6B, IRQ 11, Physical connectors present: RJ45
IP Protocols: ICMP, UDP, TCP, IGMP
VFS: Mounted root (ext2 filesystem).
ide: Assuming 33MHz system bus speed for PIO modes
ICH3M: IDE controller on PCI bus 00 dev f9
ICH3M: not 100% native mode: will probe irqs later
      ide0: BM-DMA at 0x1860-0x1867, BIOS settings: hda:DMA, hdb:pio
hda: IC25N020ATCS04-0, ATA DISK drive
ide0 at 0x1f0-0x1f7,0x3f6 on irq 14
Real Time Clock Driver v1.10e
usb.c: new USB bus registered, assigned bus number 3
hub.c: 2 ports detected
SCSI subsystem driver Revision: 1.00
scsi0 : SCSI host adapter emulation for IDE ATAPI devices
      Vendor: MATSHITA   Model: UJDA730 DVD/CDRW   Rev: 1.04
      Type:   CD-ROM                                ANSI SCSI revision: 02
apm: BIOS version 1.2 Flags 0x03 (Driver version 1.16)
parport0: PC-style at 0x3bc, irq 7 [PCSP,TRISTATE]
lp0: using parport0 (interrupt-driven).
```



"Systemaufruf-Autopsie"

Dieses C-Programm liest den ersten Sektor der ersten IDE-Festplatte (`/dev/hda`)

```
#include <unistd.h>
#include <fcntl.h>
int main() {
    int fd;
    char buf[512];

    fd = open("/dev/hda", O_RDONLY);
    if (fd >= 0)
        read(fd, buf, sizeof(buf));
    return 0;
}
```

Was passiert hier (Linux Kernel 2.4.0)?



read()

```
ssize_t sys_read(unsigned int fd, char *buf,  
                 size_t count) {  
    struct file *file = fget(fd);  
    return file->f_op->read(file, buf, count, &file->f_pos);  
}
```

- ▶ **Der read() -Systemaufruf beauftragt das Filesystem,**
 - aus Datei "file"
 - ab Position, die in &file->f_pos abgelegt ist
 - count-viele Bytes
 - in Speicher ab buf einzulesen
- ▶ **file->f_op->read zeigt in unserem Fall (/dev/hda geöffnet) auf die Funktion block_read()**

block_read()

354 ssize_t

```
block_read(struct file *filp, char *buf, size_t count, loff_t *ppos) {  
    struct inode *inode = filp->f_dentry->d_inode;  
    kdev_t dev = inode->i_rdev;  
    ssize_t blocksize = blksize_size[MAJOR(dev)][MINOR(dev)];  
    loff_t offset = *ppos;  
    ssize_t read = 0;  
    size_t left, block, blocks;  
    struct buffer_head *bhreq[NBUF];  
    struct buffer_head *buflist[NBUF];  
    struct buffer_head **bh;  
  
    left = count;                /* bytes to read */  
    block = offset / blocksize;    /* first block */  
    offset &= (blocksize-1);       /* starting offset in block */  
    blocks = (left + offset + blocksize - 1) / blocksize;  
  
    ...
```



block_read()

355

```
...
bh = buflist;
do {
    while (blocks) {
        --blocks;
        *bh = getblk(dev, block++, blocksize);
        if (*bh && !buffer_uptodate(*bh))
            bhreq[bhrequest++] = *bh;
    }
    if (bhrequest)
        ll_rw_block(READ, bhrequest, bhreq);
    /* wait for I/O to complete,
       copy result to user space,
       increment read and *ppos, decrement left */
} while (left > 0);
return read;
}
```

ll_rw_block()

```
ll_rw_block(int rw, int nr, struct buffer_head * bhs[]) {
    int i;

    for (i = 0; i < nr; i++) {
        struct buffer_head *bh = bhs[i];
        bh->b_end_io = end_buffer_io_sync;
        submit_bh(rw, bh);
    }
}
```

Nach Abschluss der
I/O-Operation diese
Funktion aufrufen

```
end_buffer_io_sync(struct buffer_head *bh, int uptodate) {
    mark_buffer_uptodate(bh, uptodate);
    unlock_buffer(bh);
}
```

```
submit_bh(int rw, struct buffer_head *bh) {
    bh->b_rdev = bh->b_dev;
    bh->b_rsector = bh->b_blocknr * (bh->b_size >> 9);
    generic_make_request(rw, bh);
}
```

führt hier letztlich zu `do ide_request()`



do_ide_request()

```
do_ide_request(request_queue_t *q) {
    ide_do_request(q->queuedata, 0);
}

ide_do_request(ide_hwgroup_t *hwgroup, int masked_irq) {
    ide_startstop_t startstop;
    while (!hwgroup->busy) {
        hwgroup->busy = 1;
        drive = choose_drive(hwgroup);
        startstop = start_request(drive);
        if (startstop == ide_stopped) hwgroup->busy = 0;
    }
}

ide_startstop_t start_request (ide_drive_t *drive) {
    unsigned long block, blockend;
    struct request *rq;

    rq = blkdev_entry_next_request(&drive->queue.queue_head);
    block = rq->sector;
    block += drive->part[minor & PARTN_MASK].start_sect;
    SELECT_DRIVE(hwif, drive);
    return (DRIVER(drive)->do_request(drive, rq, block));
}
```

└─ führt zu do_rw_disk()



do_rw_disk()

358

```
ide_startstop_t
do_rw_disk (ide_drive_t *drive, struct request *rq, unsigned long block)
{
    if (IDE_CONTROL_REG) OUT_BYTE(drive->ctl, IDE_CONTROL_REG);
    OUT_BYTE(rq->nr_sectors, IDE_NSECTOR_REG);
    if (drive->select.b.lba) {
        OUT_BYTE(block, IDE_SECTOR_REG);
        OUT_BYTE(block>>=8, IDE_LCYL_REG);
        OUT_BYTE(block>>=8, IDE_HCYL_REG);
        OUT_BYTE(((block>>8)&0x0f) | drive->select.all, IDE_SELECT_REG);
    } else {
        ...
    }
    if (rq->cmd == READ) {
        ide_set_handler(drive, &read_intr, WAIT_CMD, NULL);
        OUT_BYTE(WIN_READ, IDE_COMMAND_REG);
        return ide_started;
    }
    ...
}
```

Controller-Register setzen

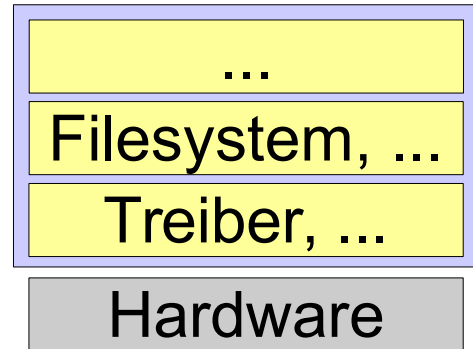
Interrupt-Handler setzen,
löst u.a. b_end_io aus (s.o.)

OUT_BYTE() schreibt Byte in ein Controller-Register



Virtualisierung - Motivation

- ▶ Normalerweise stellt eine Betriebssystemschicht nach „oben“ eine abstraktere (höhere) Schnittstelle bereit
- ▶ Beispiel:

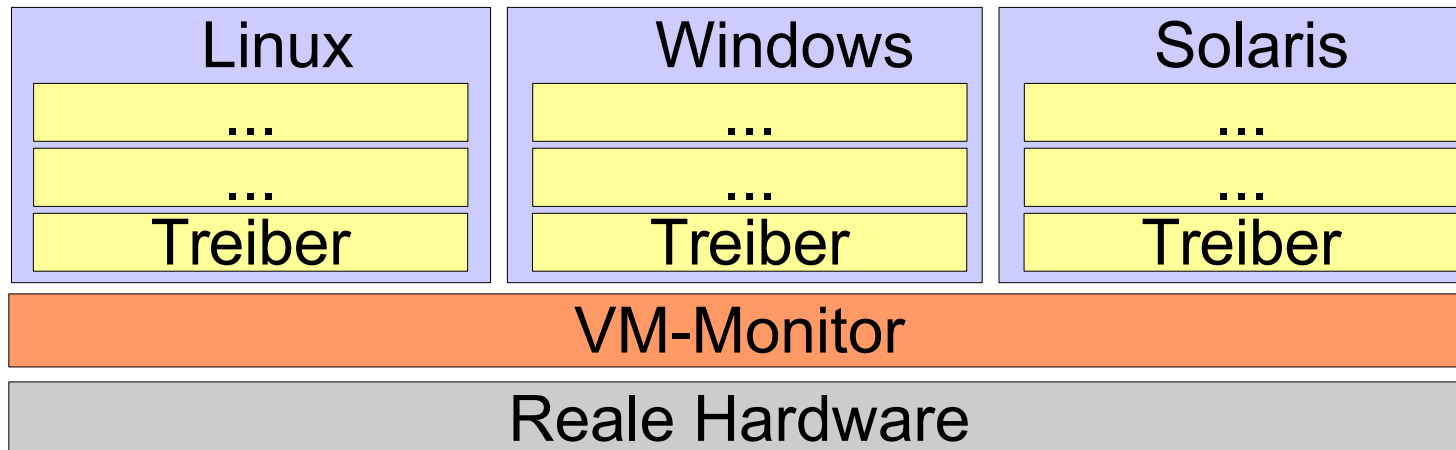


- ▶ **„Serverkonsolidierung“: Dienste mehrerer Server auf einem leistungsfähigen Rechner konzentrieren**
 - kostengünstiger, effektiveres Ressourcen-Sharing
 - leichtere Administration, ...
- ▶ **Probleme, z.B.**
 - Dienste benötigen vielleicht verschiedene Betriebssysteme
 - Sicherheit: gegenseitige Abschottung nötig
 - ...



Virtualisierung (HW-Ebene)

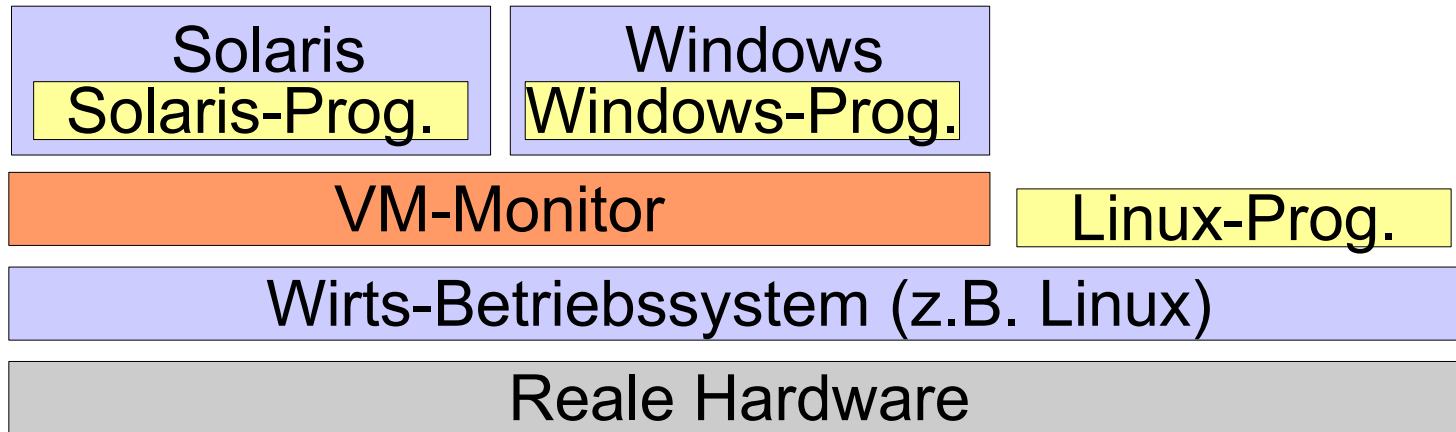
- ▶ Ein *virtual machine monitor* (VMM) bietet eine Hardwareschicht nach „oben“ mehrfach identisch an.
- ▶ Darauf können mehrere (auch verschiedene) Gast-Betriebssysteme unmodifiziert nebeneinander ablaufen.
- ▶ Der VMM fängt I/O-Operationen der Gast-Betriebssysteme ab und koordiniert den Zugriff auf die gemeinsame (reale) Hardware.





Virtualisierung mit Wirts-BS

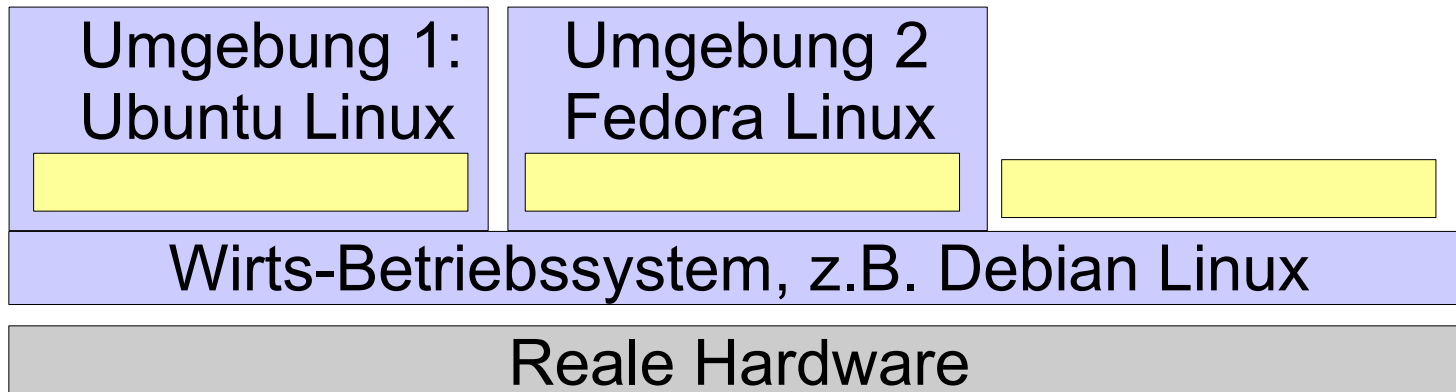
- ▶ Statt direkt auf der Hardwaresebene (s.o.) kann sich der VMM für I/O-Operationen auch auf ein „Wirts-Betriebssystem“ abstützen
- ▶ Ansteuerung der realen HW ist damit kein Problem des VMM mehr (Treiber des Wirts-BS kümmern sich darum),
- ▶ dafür ggf. Effizienzverluste durch Zusatzschicht





Virtualisierung auf BS-Ebene

- ▶ Nur **ein** Betriebssystem**kern** läuft, dieser
- ▶ stellt mehrere, voneinander **getrennte, einschränkbare Ausführungsumgebungen** zur Verfügung durch
 - separate Prozesstabellen, Speicherzuteilung,
 - UserIDs, GruppenIDs, eigene virtuelle Netzwerkgeräte etc.
- ▶ In Umgebungen können verschiedene Betriebssystem-Varianten laufen, solange sie mit dem **gemeinsamen BS-Kern** kompatibel sind.
- ▶ „Container“-Konzept gerade populär durch Docker & Co
- ▶ (Fast) kein Performanceverlust





Wie alles begann...

► Ein Betriebssystem

- verwaltet die **Betriebsmittel** eines Rechnersystems (Effizienz, Koordination, Schutz, Abrechnung, ...)
- stellt eine **abstraktere Schicht** oberhalb der Hardware bereit, die Hardware-Details verbirgt und
- stellt Anwendern und Programmierern dadurch eine höhere, leichter zu handhabende **Schnittstelle zu den Diensten** des Rechners bereit.

► Betriebsmittel:

- **Softwarebetriebsmittel** wie Dateien, Programme, ...
- **Hardwarebetriebsmittel** wie CPU, Speicher, ... (s.o.)

