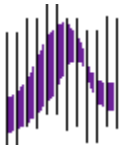
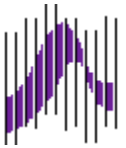
Vorlesung Betriebssysteme



Martin Zeller
Hochschule Ravensburg-Weingarten

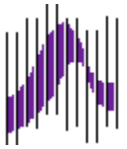
Der vorliegende Foliensatz basiert auf Folien
von Frau Prof. Dr. Keller

Ausweich-Termin



Mittwoch 2. Block

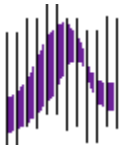
Ziel der Vorlesung



Grundlegendes Verständnis für die Aufgaben und die Arbeitsweise von Betriebssystemen.
Kenntnisse zu Betriebssystem-Schnittstellen.

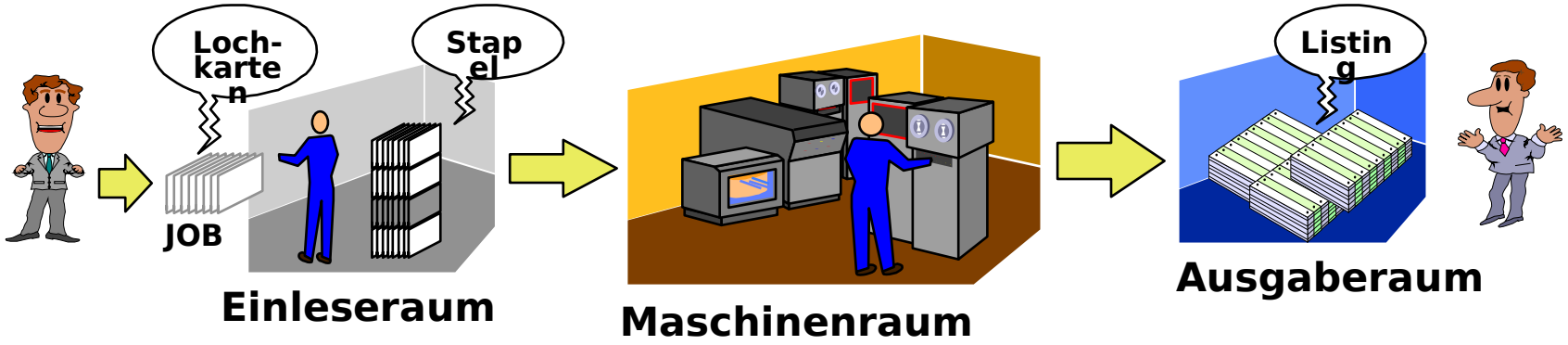
- Prozesse, Threads
- Interprozesskommunikation, Synchronisation
- Scheduling
- Speicherverwaltung
- Ein- Ausgabe
- Dateisysteme

Inhalt der Vorlesung

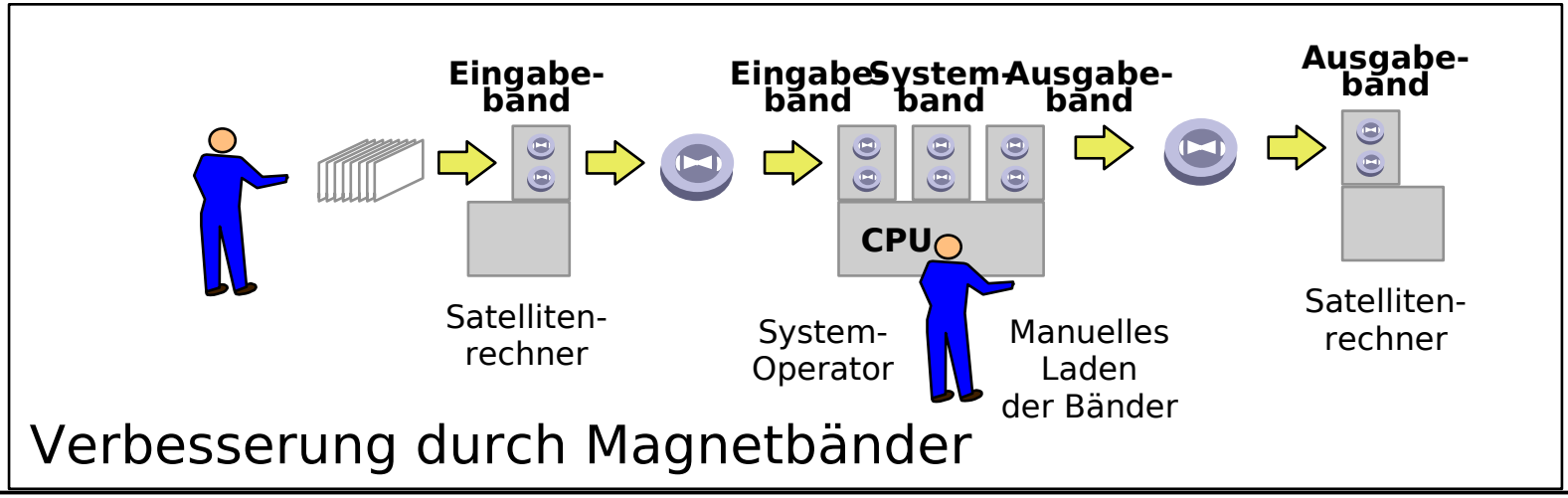


- Grundlegende Mechanismen
  - Prozesse, Threads
  - User-Level, Kernel-Level
  - Systemaufrufe, Interrupts
- Speicherverwaltung
- Synchronisation
- Interprozesskommunikation
- Ein- und Ausgabe
- Dateisysteme

Stapelbetrieb (Batch Systeme)



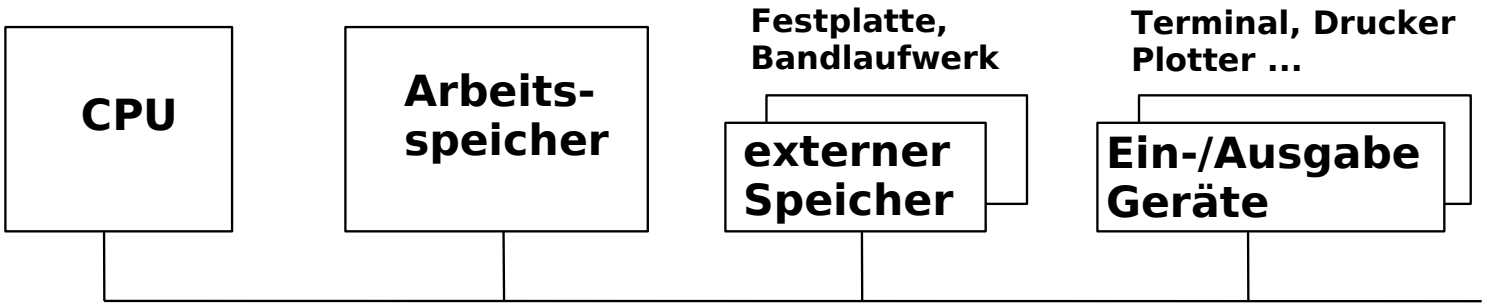
- Erste Programmiersprachen und Betriebssysteme (Monitore)
- Assembler, FORTRAN (Formular Translator)
- FMS (Fortran Monitoring System)



Verbesserung durch Magnetbänder

Systemressourcen

von Neumann Architektur



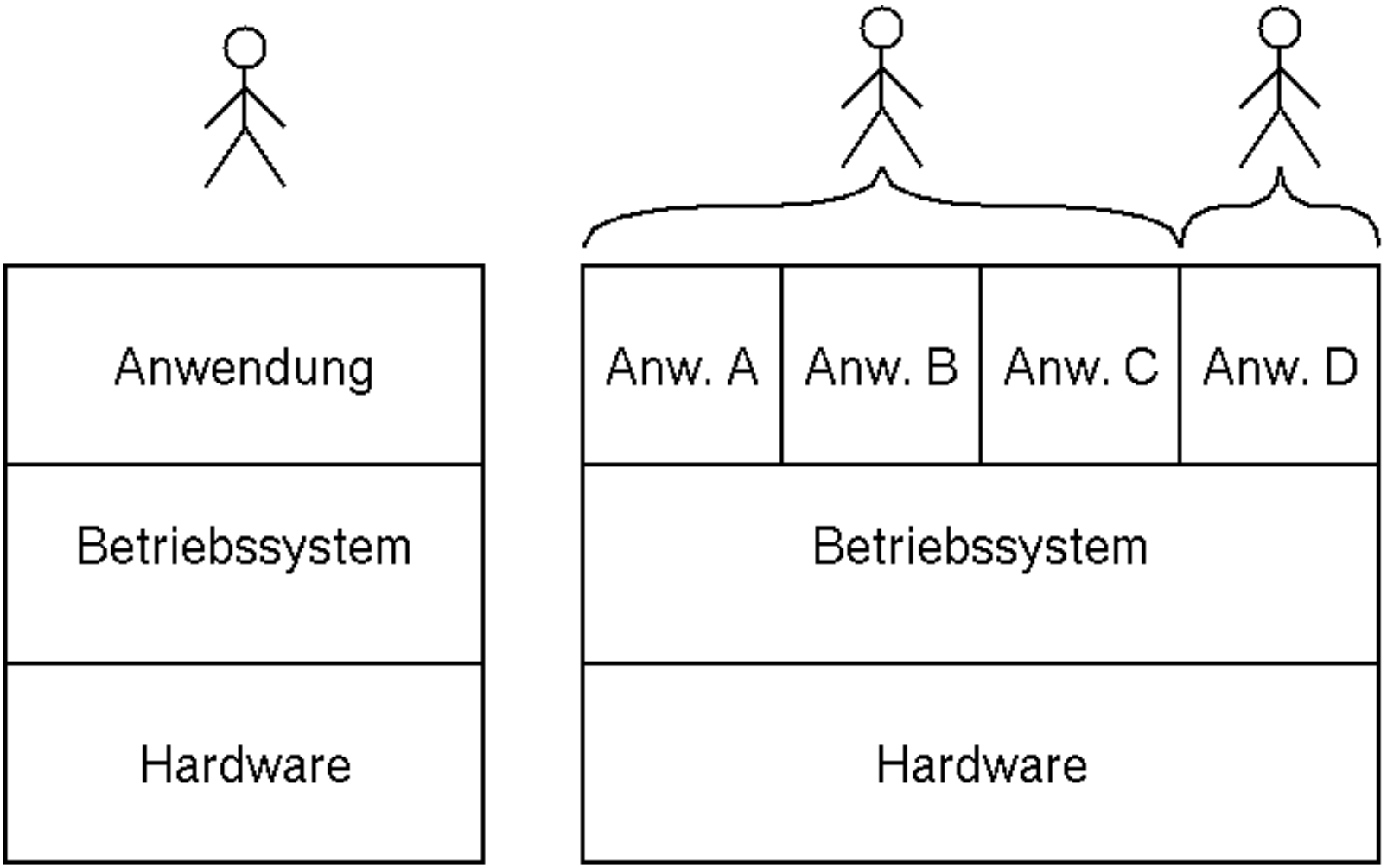
Ausführung mehrerer nebenläufiger Programme (Multiprogramming)

Betriebssystem
Anwendungs-Programm
Daten

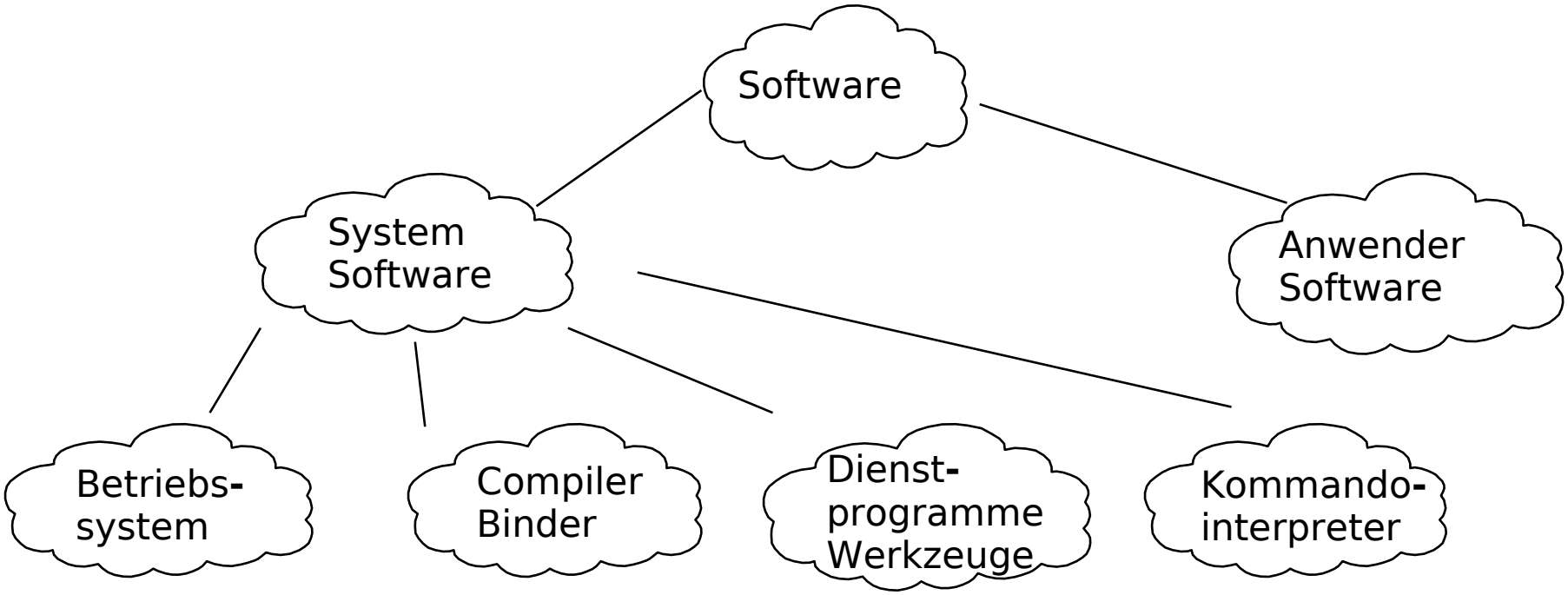
Verwaltung von Daten

Zuteilung an Programme

Multi Program Multi User Systeme



Systemsoftware



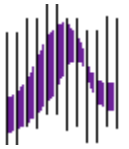
- Hardwareunabhängige Ein-/Ausgabe
- Effiziente Nutzung der Ressourcen
- Laden von Programmen

- Erstellen von Anwender-programmen

- Editoren
- Filter

- Mensch/Maschine Schnittstelle
- Textorientiert z.B. UNIX-Shell
- Grafische Oberfläche z.B. MS-Windows, X-Windows, MacOS

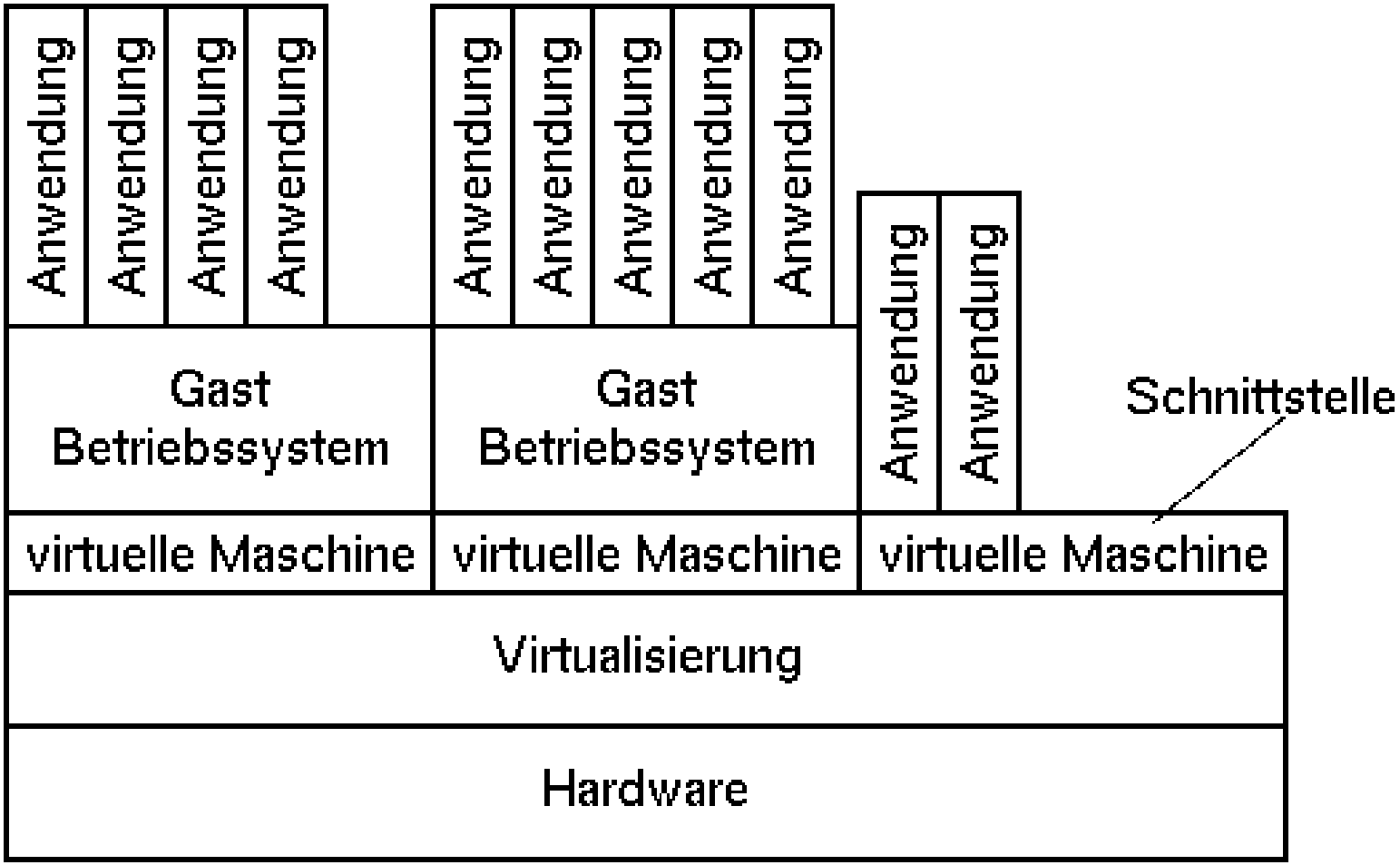
Virtualisierungssysteme



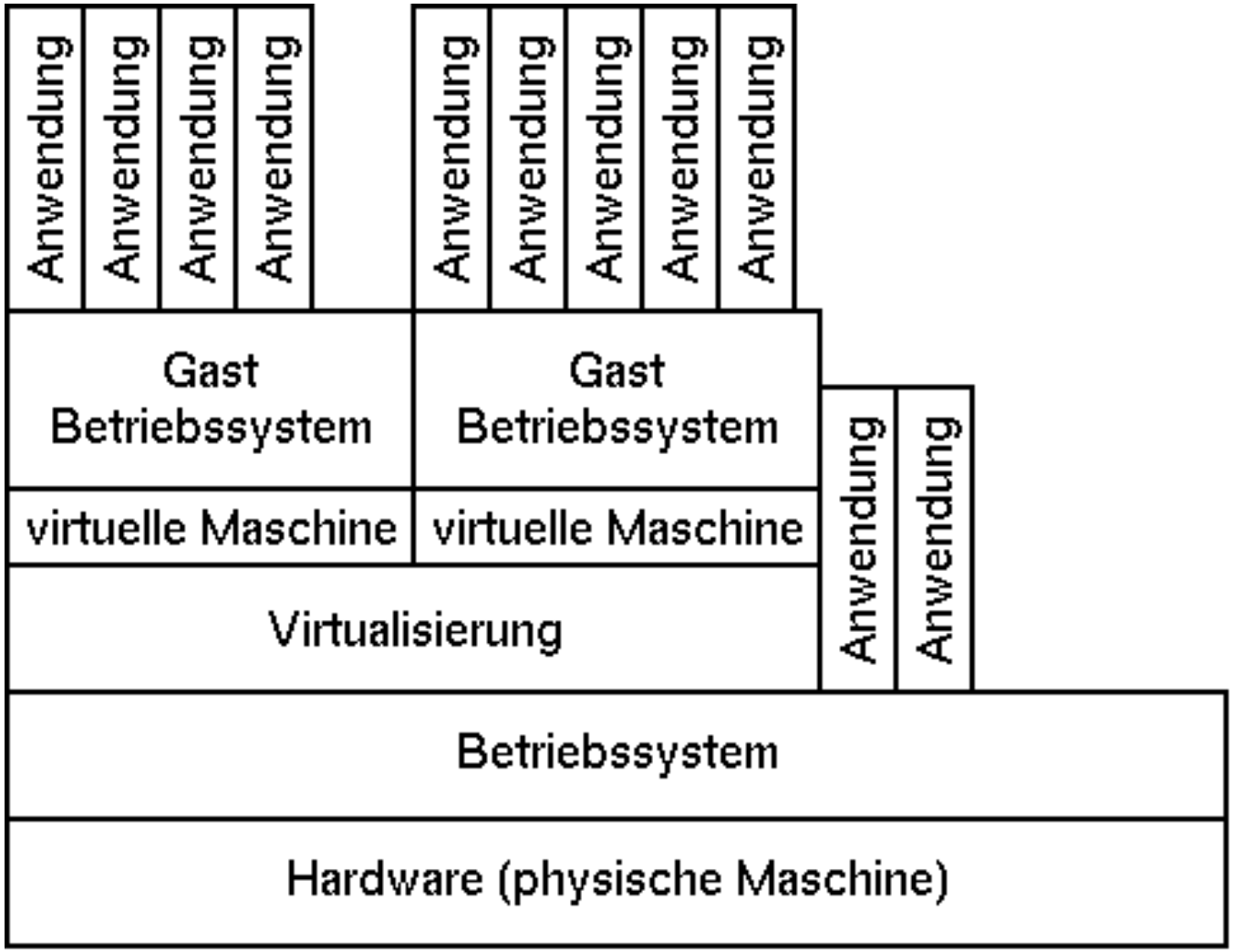
Virtualisierungssysteme realisieren mehrere virtuelle Maschinen auf einem Rechner

- VmWare, KVM, VirtualBox, Xen ...
- Java Virtual Machine (SmallTalk, . . .)
- OS/370, OS/390
- Partitionierende RTOS (LynxOS-178, . . .)
- Solaris (Container)

Virtualisierungssysteme (2)

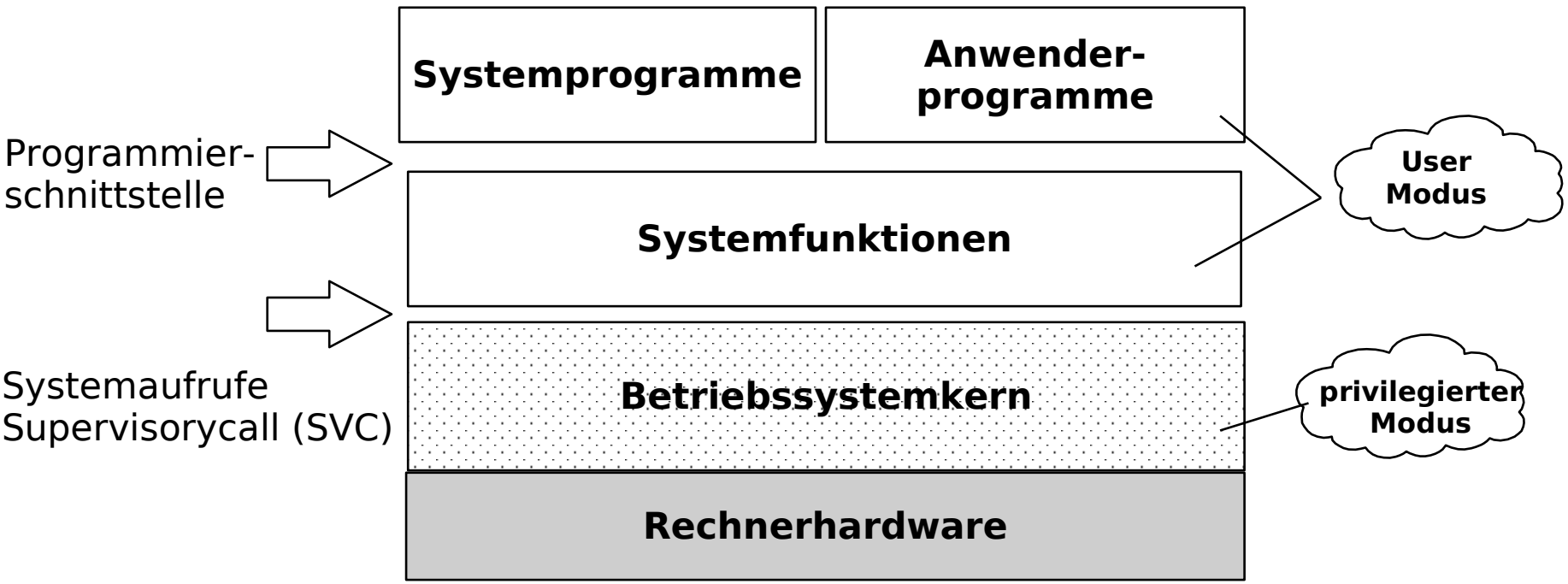


Virtualisierungssysteme (3)



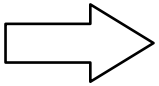
Betriebssystemstrukturen

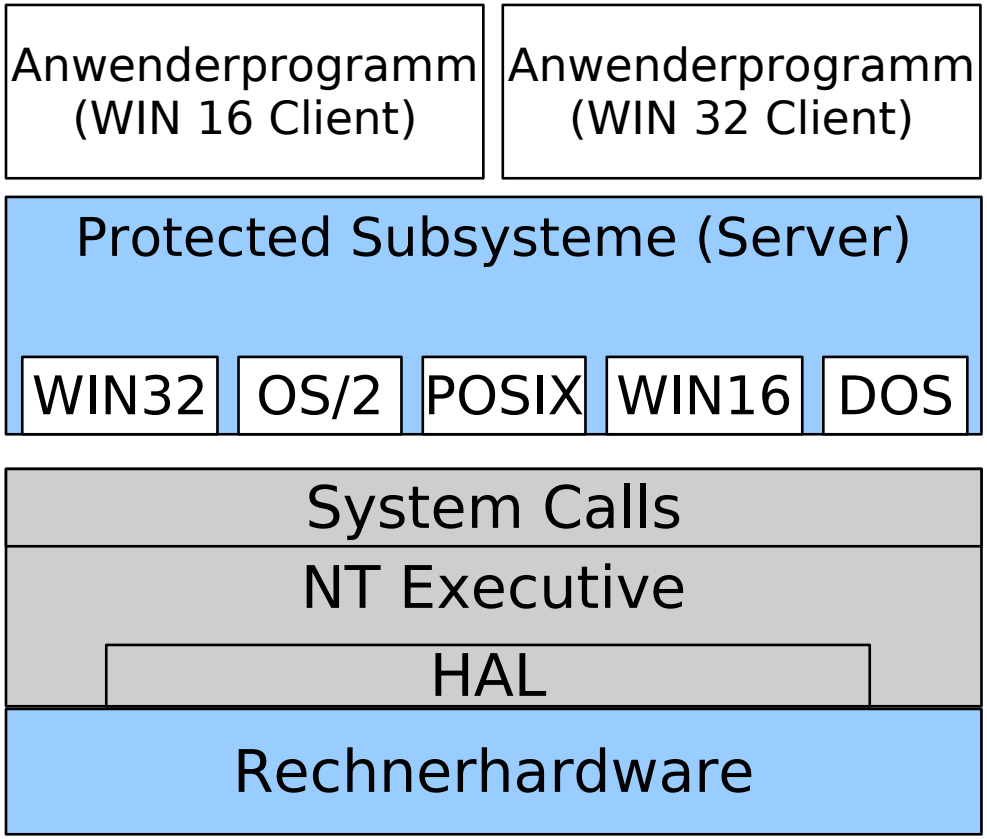
Struktur aus der Sicht des Systemprogrammierers



Windows NT Modell

zu jedem Subsystem existiert eine API



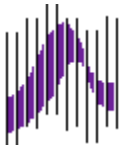


User Modus

privilegierter Modus

Windows NT = Windows New Technologie
HAL = Hardeware Abstraction Layer

Prozess



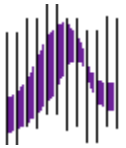
Aus Sicht des Anwenders:

- laufendes Programm

Aus Sicht des Betriebssystems:

- Programm mit Ausführungszustand
- Einheit der Ressourcen-Vergabe / Verwaltungseinheit
  - Rechenzeit
  - Speicher
  - Zugriff auf I/O-Geräte
- ggf. Einheit der Abrechnung

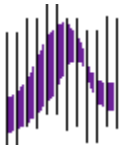
Prozess-Attribute



Ausführungszustand

- PID (process identification)
- PC (program counter)
- SP (stack pointer)
- PSW (program status word)
  - Kennung User-Level/Kernel-Level
  - Priorität
  - Interrupt-Steuerung
  - Overflow-Anzeige
  - Ergebnis des letzten Vergleichs

Prozess-Attribute



Hauptspeicher-Organisation

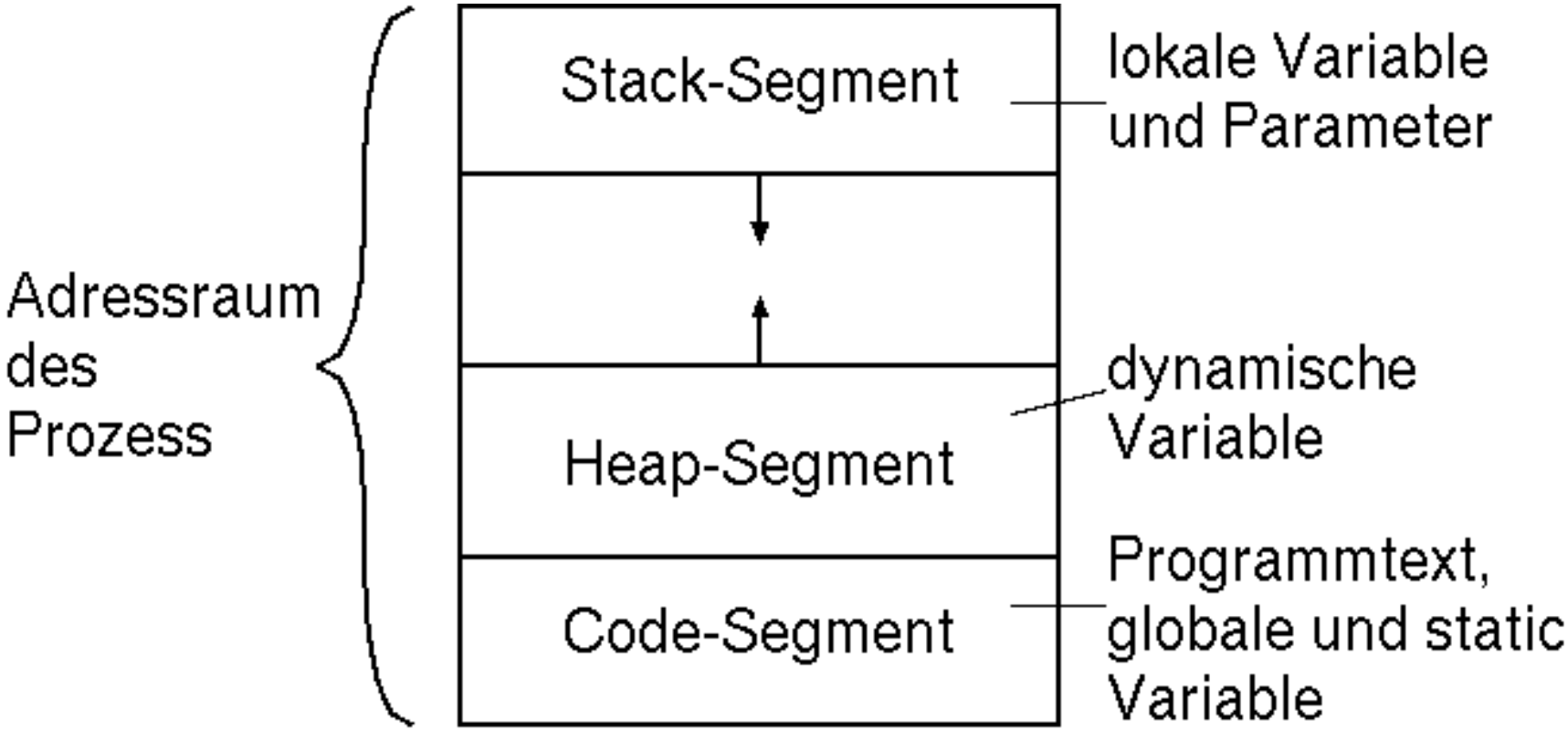
- Pointer auf Code-Segement
  Programm, globale und static Variable
- Pointer auf Stack-Segement
- Pointer auf Heap-Segment

Externer Speicher/Geräte

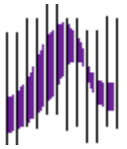
- Wurzel-Verzeichnis
- Arbeits-Verzeichnis
- Datei-Deskriptoren

Benutzer-ID, Gruppen-ID

Speicherbelegung eines Prozess



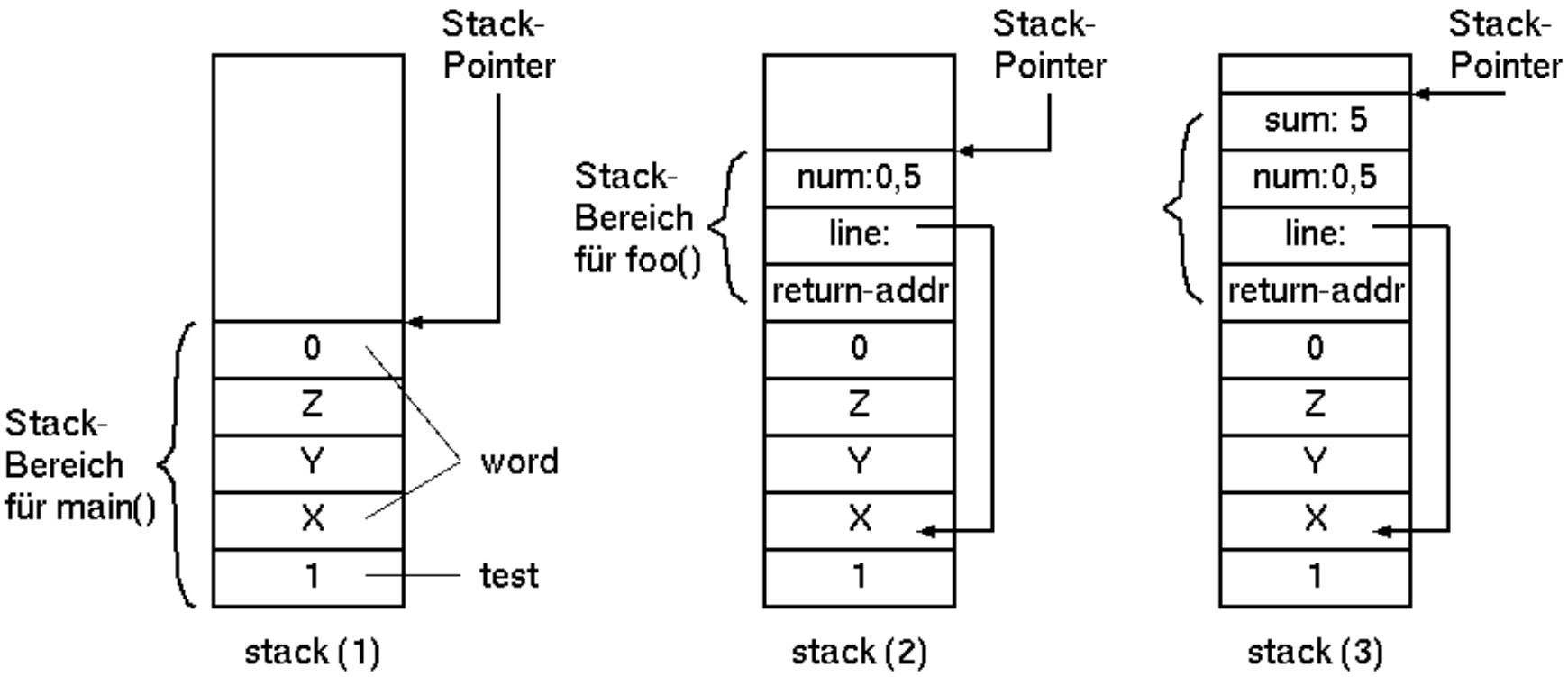
Aufruf einer Funktion im Prozess



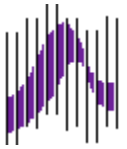
Funktion foo() ruft bar() auf

- foo() speichert auf Stack und verschiebt SP
  - Rücksprungadresse
  - Parameterwerte
- foo() verzweigt zu bar() (PC zeigt auf bar())
- bar() legt lokale Variable auf dem Stack an
- bar() führt Anweisungen aus
- bar() speichert Return-Wert an vereinbarter Stelle
- bar() setzt SP zurück auf Frame von foo()
- bar() springt zurück zu foo() (PC zeigt auf foo())

Aufruf einer Funktion im Prozess



Aufruf einer Betr.Sys-Funktion (1)

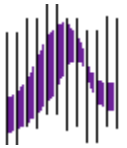


Funktion foo() ruft write() auf

- foo() speichert Parameterwerte in Registern
- foo() speichert Index von write() in Register
- foo() führt Anweisung „SVC“ aus (Kernel-Mode)
- SVC holt Funktionszeiger auf write() aus Tabelle
- write() sichert (Rest des) Prozess-Zustands auf System-Stack
- write() führt Anweisungen aus

⋮

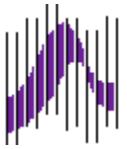
Aufruf einer Betr.Sys-Funktion (2)



Funktion foo() ruft write() auf (2)

- write() führt Anweisungen aus
- ggf. startet das BS einen anderen Prozess
- write() speichert Return-Wert an vereinbarter Stelle
- write() stellt den Zustand von foo() wieder her
- write() springt zurück zu foo() (PC zeigt auf foo())

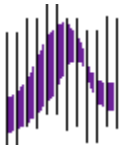
Interrupt einer Funktion



foo() läuft, ein I/O-Gerät erzeugt einen Interrupt

- CPU geht in den Kernel-Mode
- CPU speichert u.a. Befehlszähler von foo()-Prozess
- CPU verzweigt zu Code gemäß Interrupt-Vektor
  z.B. Assembler-Code + BS-Funktion bar()
- bar() sichert (Rest-)Zustand von foo()-Prozess auf Stack
- bar() führt Anweisungen aus
- bar() stellt den Zustand von foo() wieder her
- bar() springt zurück zu foo() (PC zeigt auf foo())

SVC/Trap und Interrupt



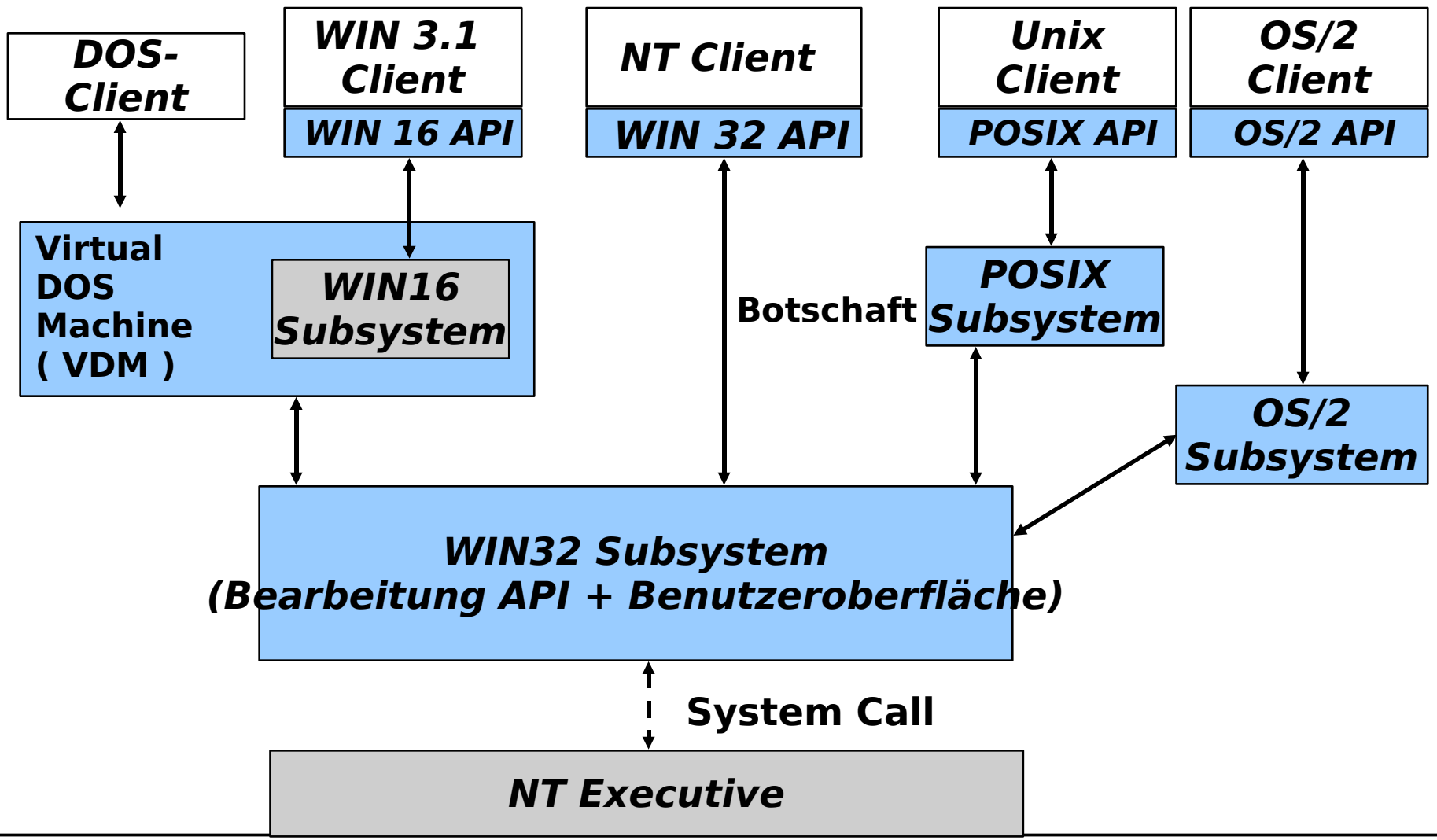
Hard- und Soft-Interrupt: Übergang User/Kernel

- Anweisungen svc/trap/int: Soft-Interrupt
- Geräte erzeugen Hard-Interrupt

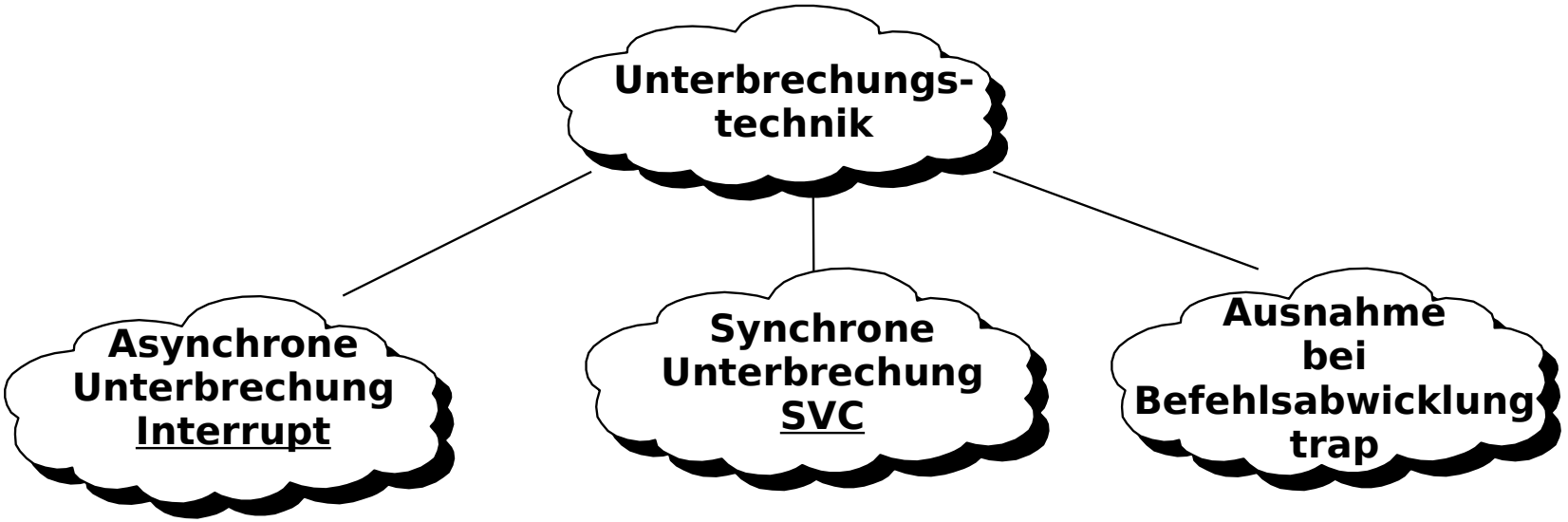
Ablauf

- CPU sichert einen Teil des Prozess-Zustands auf einem zentralen Stack.
- PC, PSW, SP, Code-/Heap-Pointer . . .
- CPU verzweigt über Tabelle zur Interrupt-Routine
- Anweisung „Return-from-Interrupt“ stellt vorherigen Prozessorzustand wieder her.

Windows NT - Protected Subsysteme



Unterbrechungstechnik

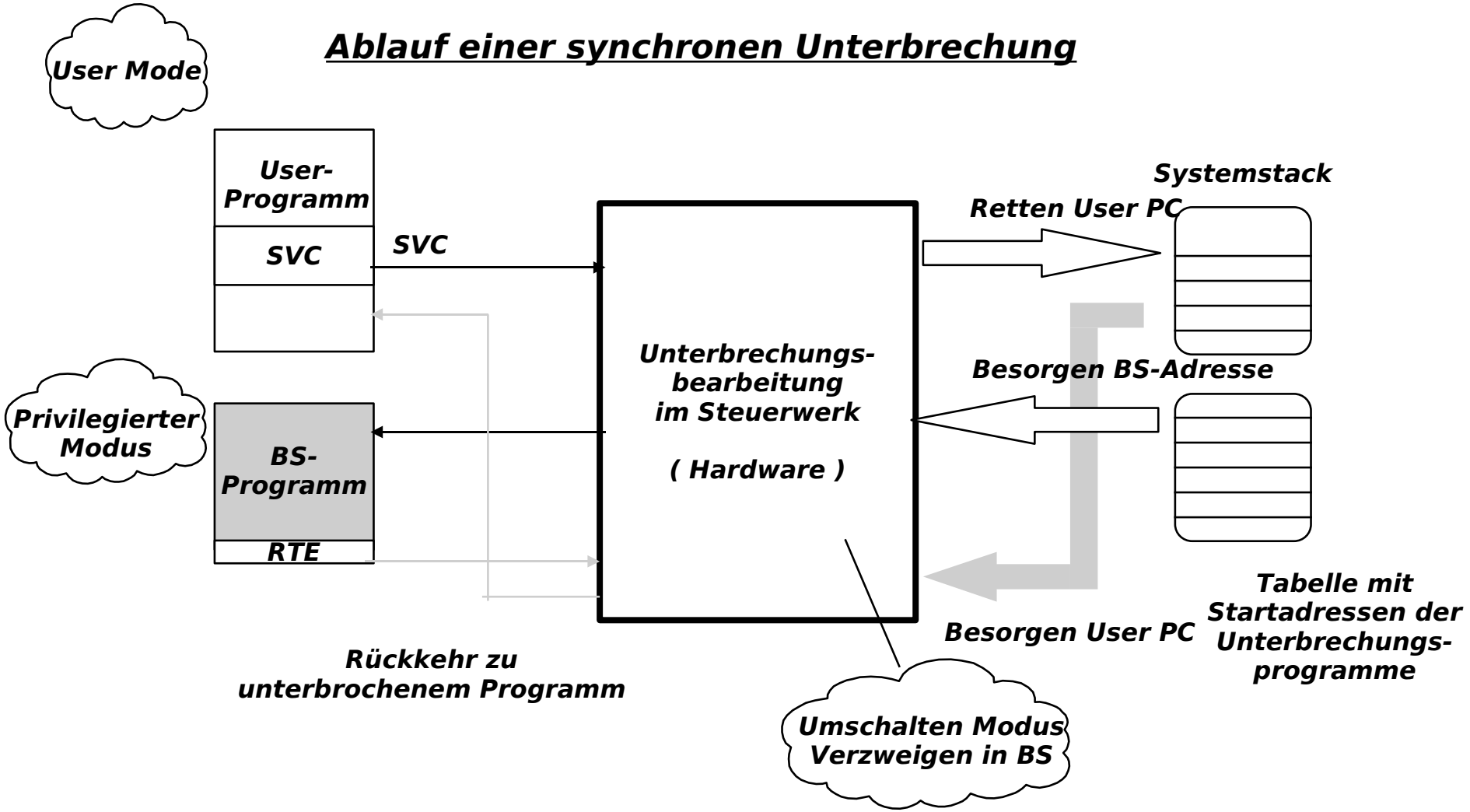


Gerätetreiber
löst Aktion aus

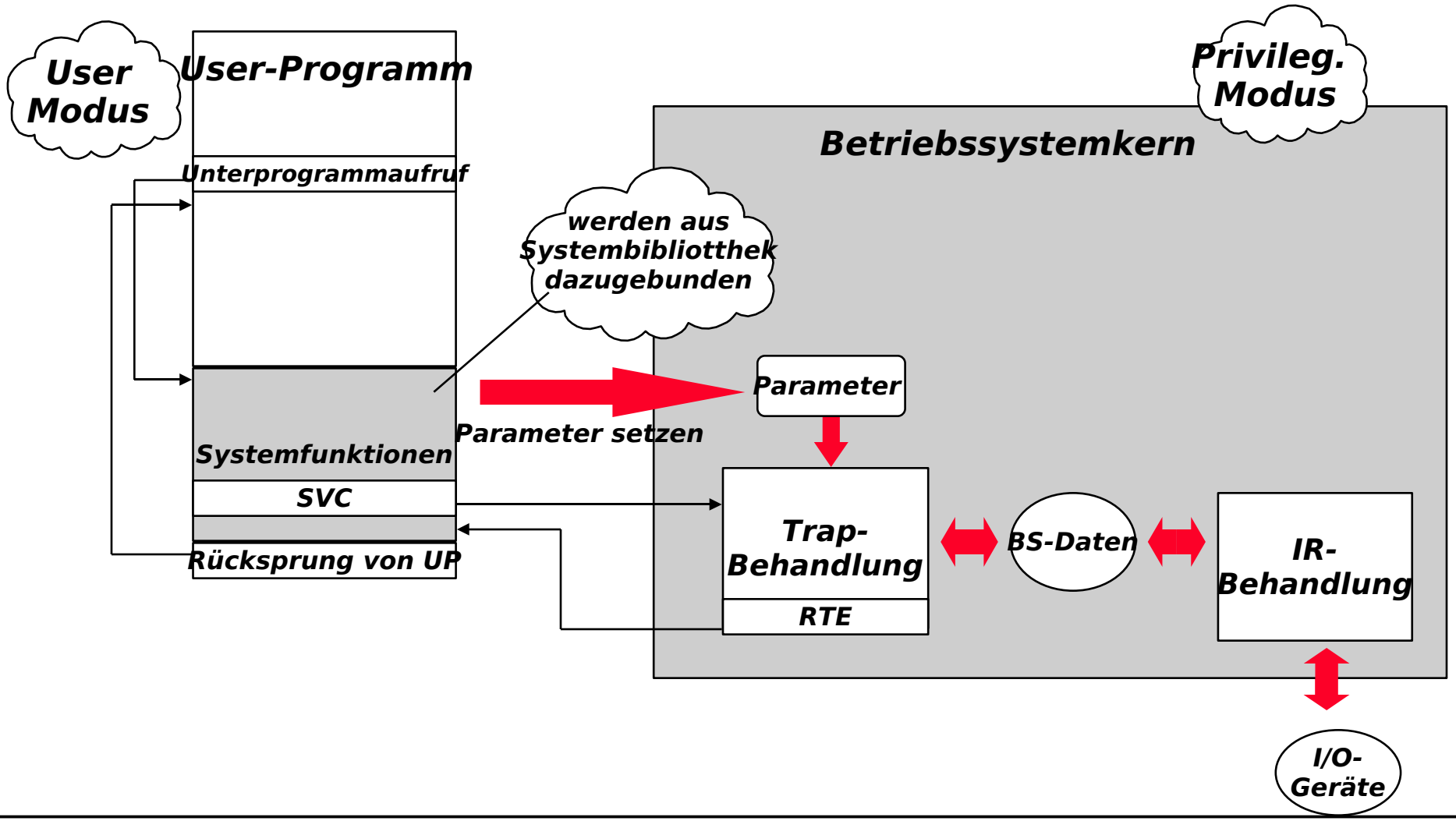
Betriebssystemkern
Anwendungsprogramm
löst Aktion aus

Fehlerbehandlung
CPU löst Aktion aus

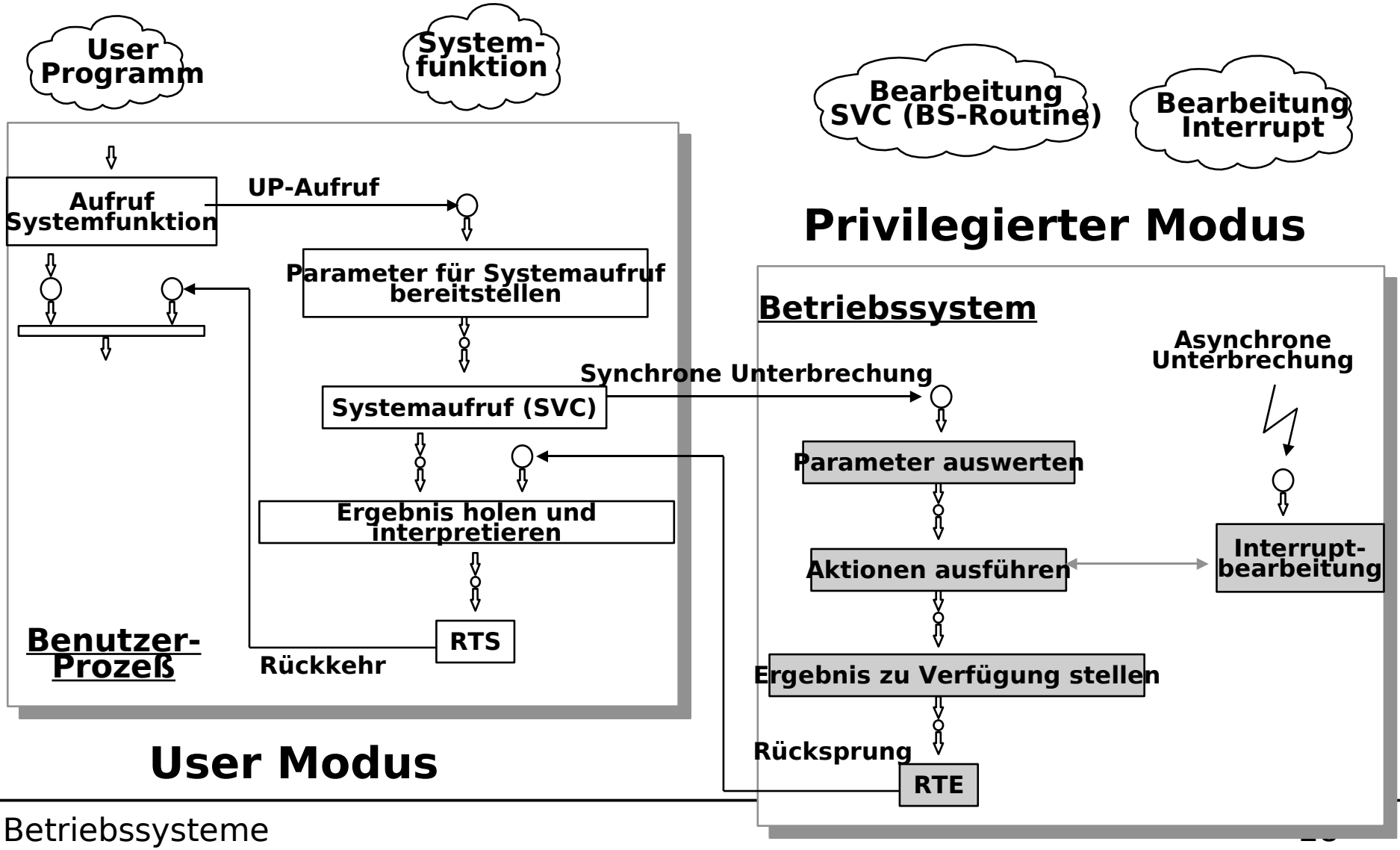
Systemaufrufe (SVC)



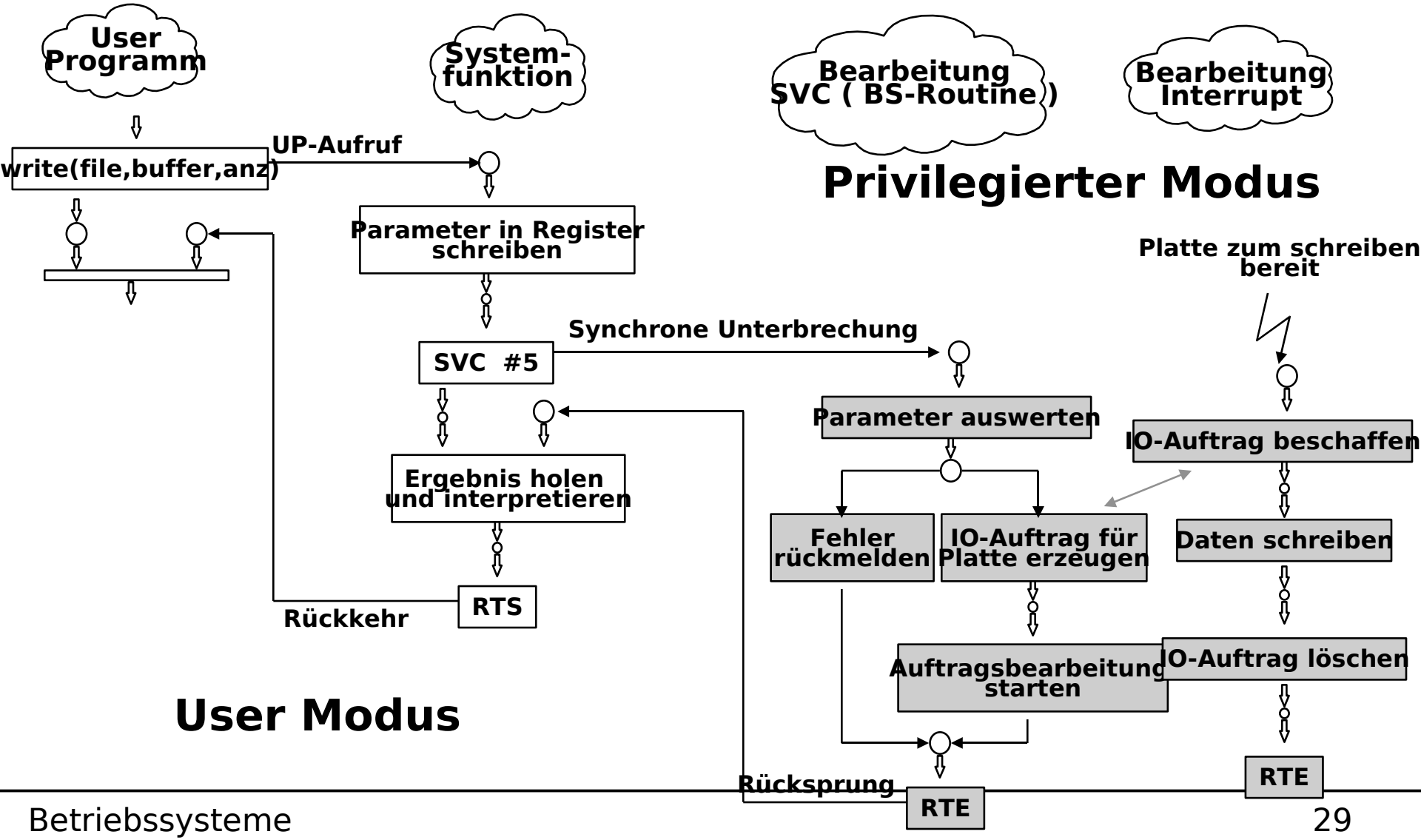
Systemfunktionen als Programmierschnittstelle



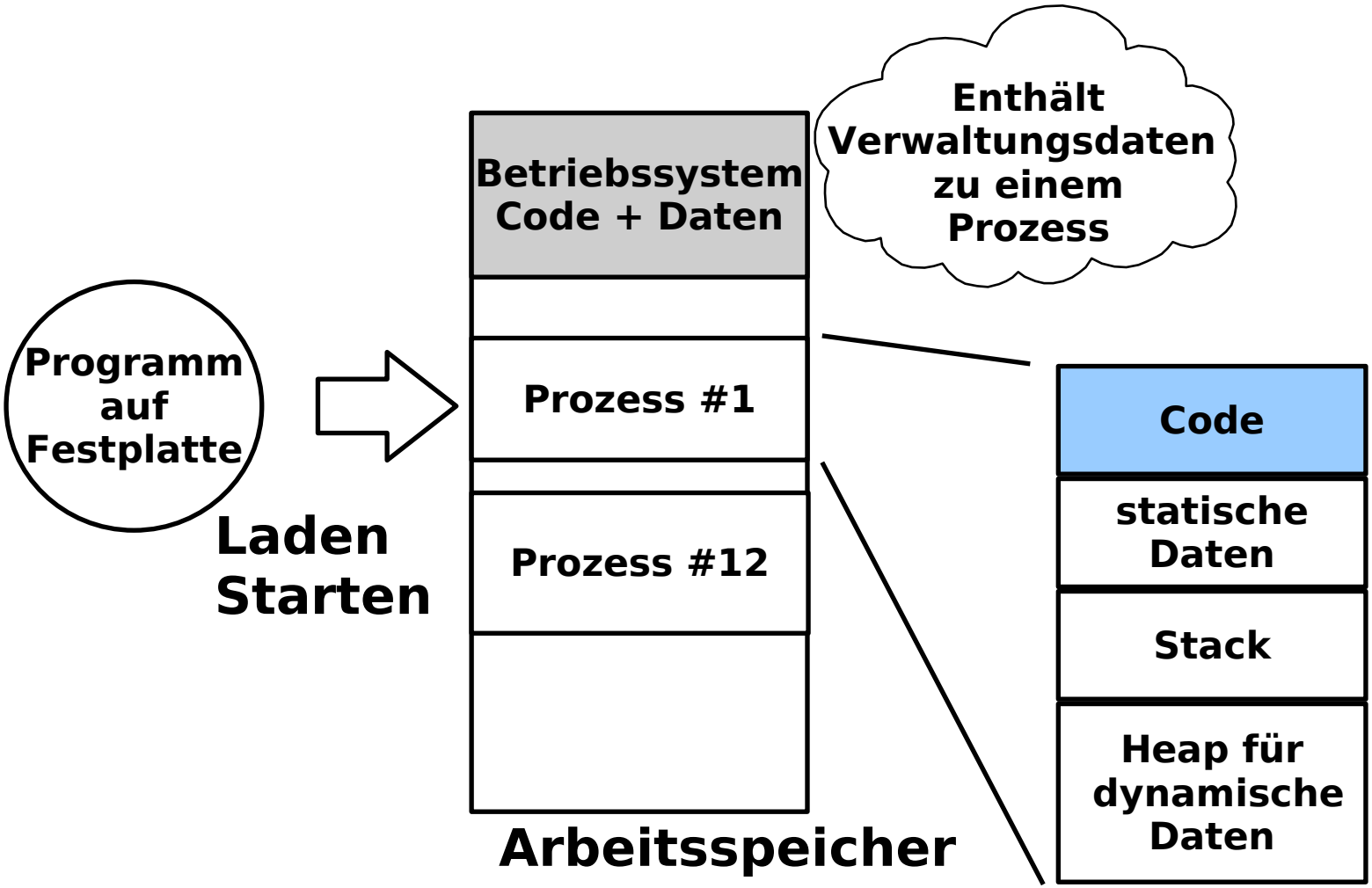
Systemaufrufe



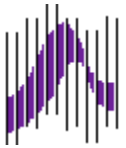
Beispiel : Daten in Datei schreiben



Multitasking - Prozess, Task

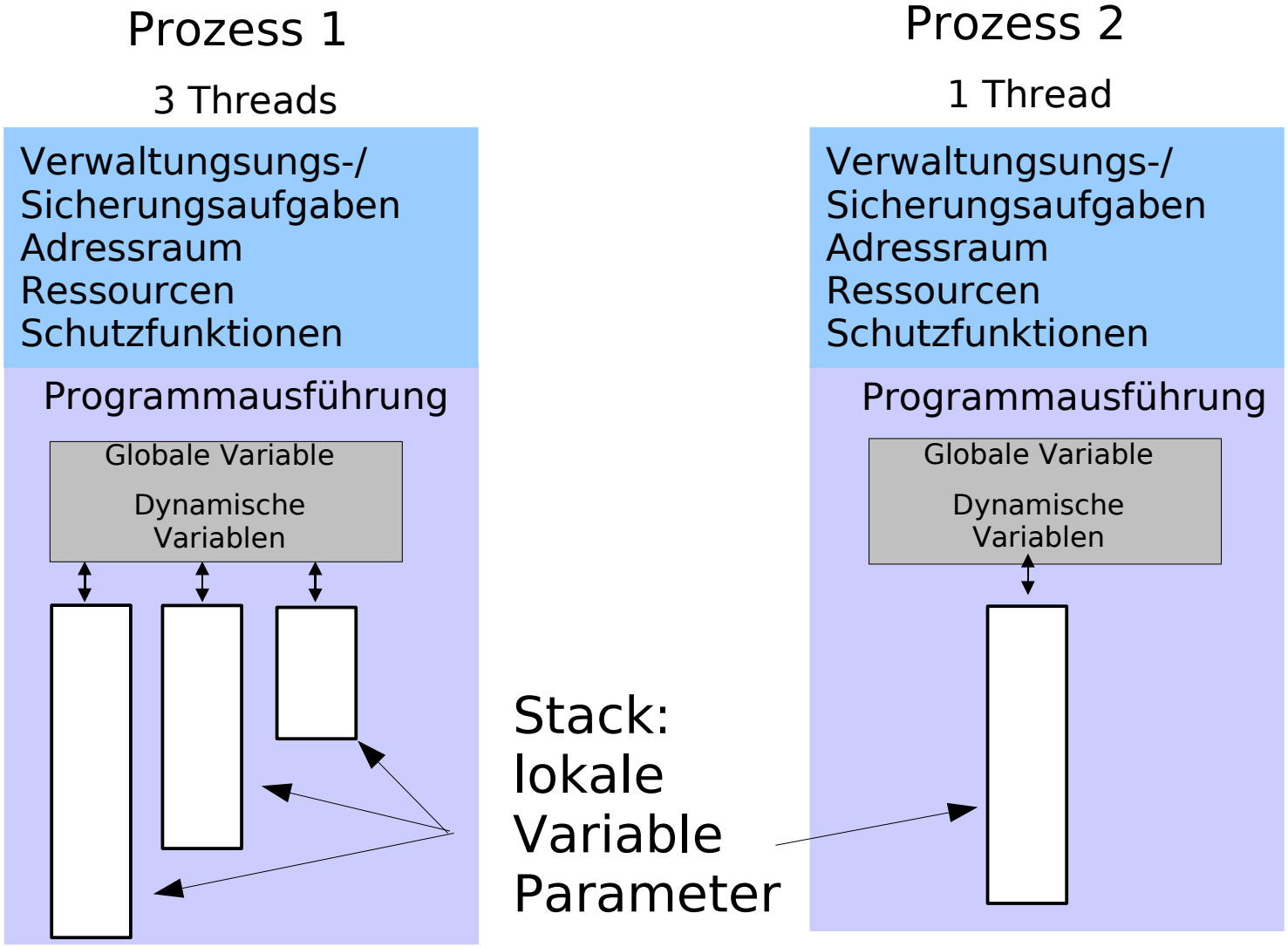


Threads

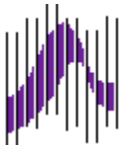


- Sequentielle Programm- / Funktionsausführung innerhalb eines Prozesses **(thread = Ausführungsfaden)**
  - Ein Thread bearbeitet eine sequentielle Teilaufgabe innerhalb eines Prozesses
  - Mehrere nebenläufige Teilaufgaben können in mehreren Threads innerhalb eines Prozesses abgearbeitet werden (**Multithreading**)
  - Thread verfügt über einen eigenen **Hardwarekontext** (PC, SP, Prozessorregister), einen Ausführungszustand und einen eigenen Stack
  - Ein Thread läuft jedoch im Softwarekontext des Prozesses und teilt sich damit den virtuellen Adressraum und alle Ressourcen mit allen anderen Threads des Prozesses

Threads
Nebenläufige Ausführungsfunktionen

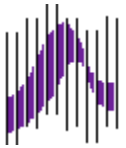


Threads



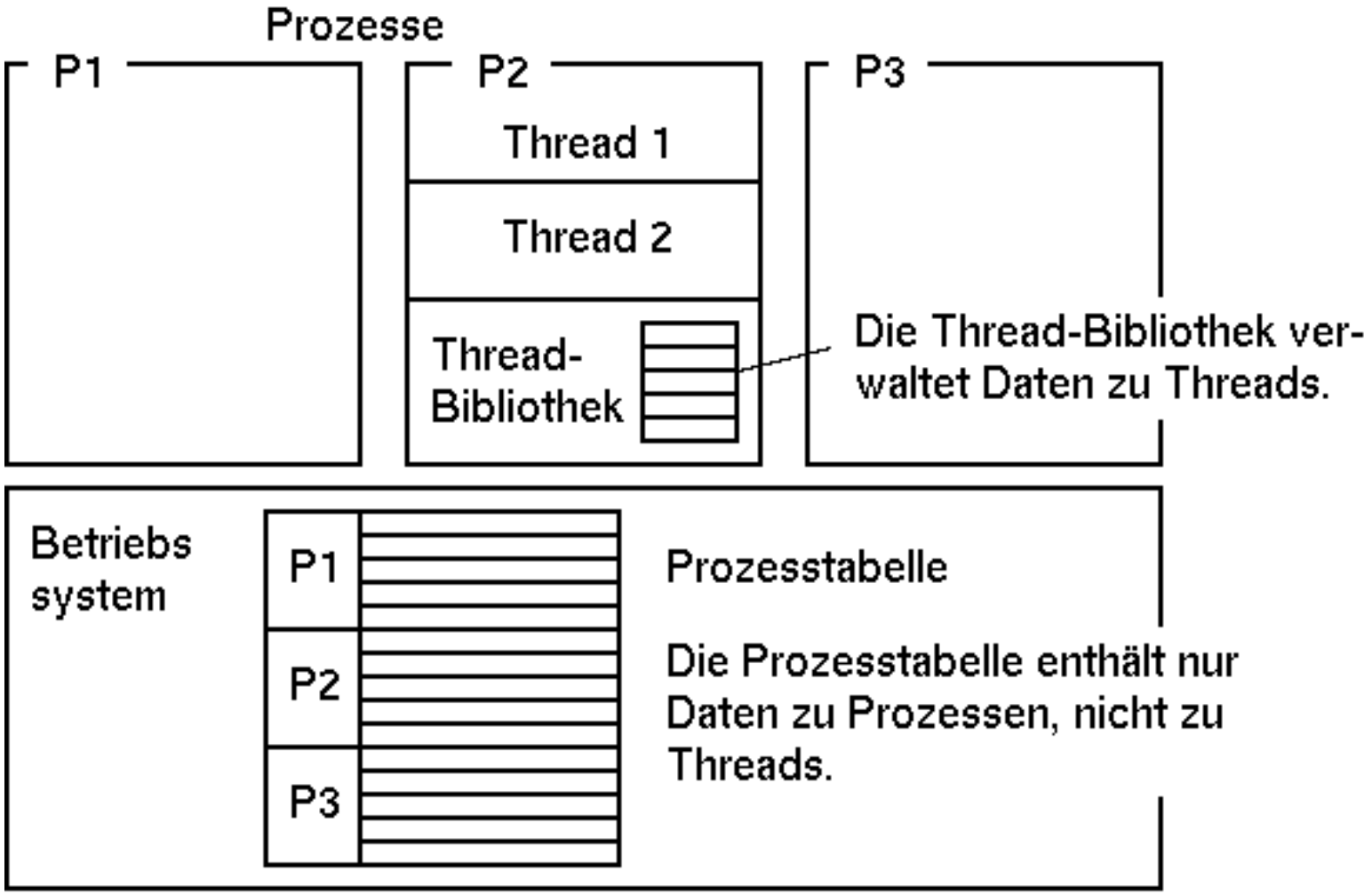
- Vorteile:
  - Threadwechsel ist schneller als Prozesswechsel, da nur der Hardwarekontext ausgetauscht werden muss
  - einfacher Austausch von Informationen zwischen nebenläufigen Threads durch einen gemeinsamen Adressraum (globale Daten des Prozesses), s.u.
- Nachteile:
  - Durch den gemeinsamen Adressraum ist kein Schutz vor fehlerhaftem Zugriff auf globale Daten möglich
    - Wenn mehrere Threads auf gemeinsame Daten arbeiten, müssen diese Threads synchronisiert werden. Fehlende oder fehlerhafte Synchronisation verursacht Datenverluste oder Dateninkonsistenzen
    - Fehler in der Synchronisation von Threads sind meist schwer zu lokalisieren

Multithreading
Thread-Implementierung

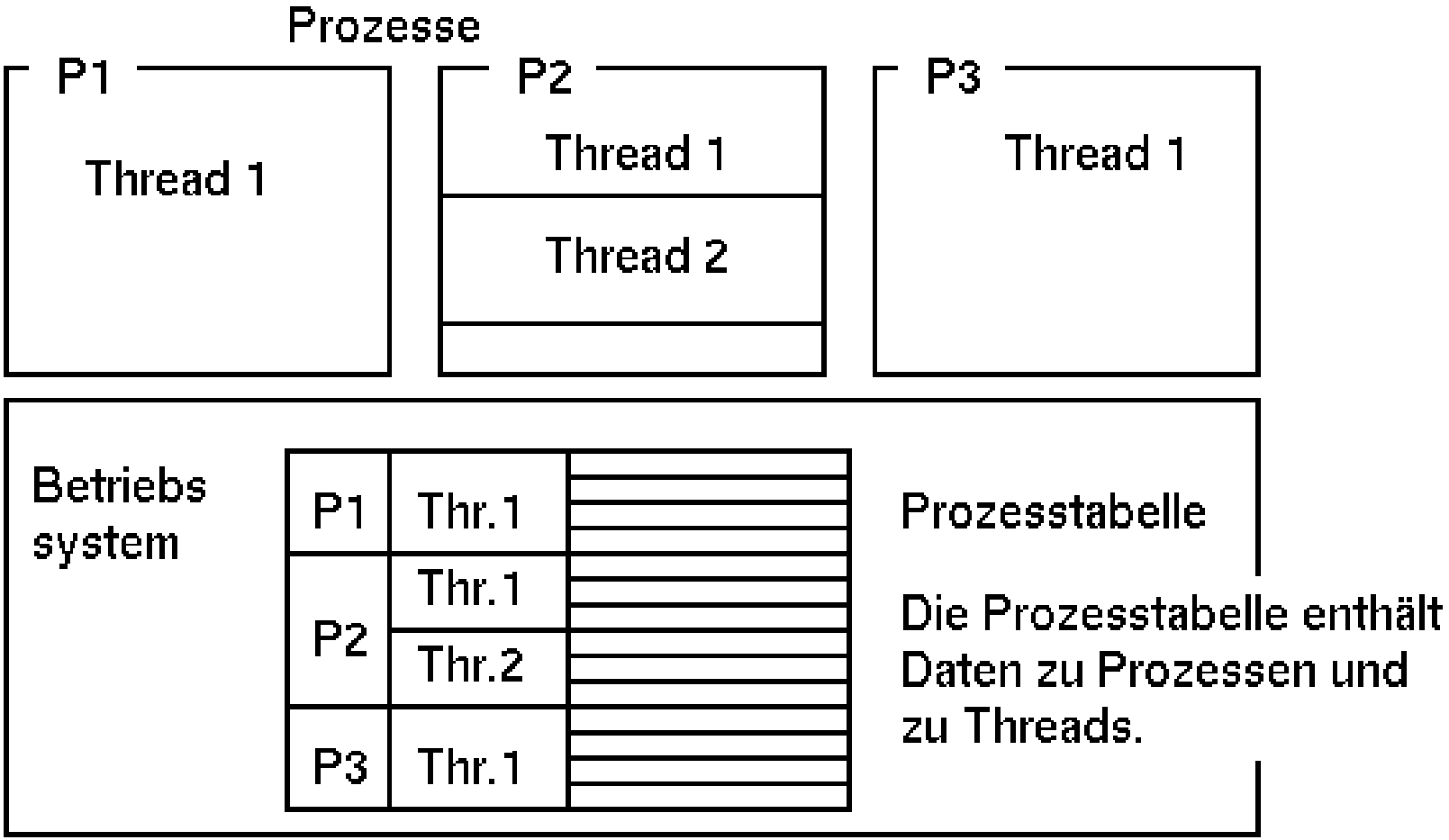


- Benutzer-Threads (ULT = user level thread)
  - unabhängig vom Betriebssystem realisierbar
- Kernel-Threads (KLT = kernel level thread)
  - Windows NT/2000/XP, Linux (ab Kernel 2.6), Mach
- Kombination (Solaris)

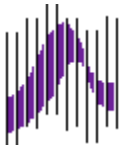
User Level Threads



Kernel Level Threads

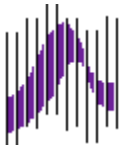


Multithreading
Benutzer-Threads



- Laufen im User Modus. Der BS-Kern (Kernel) kennt und verwaltet selbst keine Threads sondern nur Prozesse.
- Der Anwendungsprozess verwaltet seine Threads selbst
- Die Verwaltung erfolgt über eine Thread-Bibliothek, in der alle Funktionen zur Verwaltung, Erzeugung, Terminierung und Synchronisation realisiert werden.
  - In der Bibliothek sind für jeden Thread entsprechende Datenstrukturen implementiert.
  - Weiterhin übernimmt die Bibliothek auch das Umschalten und Scheduling zwischen den Threads innerhalb des Prozesses.
  - Scheduling von Prozessen ist damit unabhängig vom Scheduling der Threads

Multithreading
Benutzer-Threads



Beispiel

- BS kennt Prozess A und Prozess B
- Prozess A besitzt Thread A.1 und Thread A.2.
- Prozess B besitzt nur Thread B.1

Fall 1:

Prozess A ist rechnend. Innerhalb A ist Thread A.2 rechnend. Thread A.2 ruft eine blockiernde Systemfunktion z.B. I/O-Wunsch auf.

SVC d.h. Sprung in den Kernel zur Abarbeitung der Systemfunktion.

Prozess A wird blockiert. Prozess B wird rechnend.

Im Prozess A wird Thread A.2 immer noch als rechnend eingestuft.

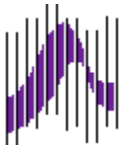
I/O.Ereignis ist eingetroffen.

Sprung in den Kernel

Prozess A wird vom Kernel wieder deblockiert.

Wenn Prozess A vom Kernel wieder in den Zustand rechnend überführt wird kann Thread A.2 weiter ausgeführt werden.

Multithreading
Benutzer-Threads



Beispiel

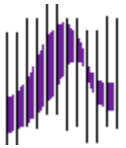
- BS kennt Prozess A und Prozess B
- Prozess A besitzt Thread A.1 und Thread A.2.
- Prozess B besitzt nur Thread B.1

Fall 2:

Prozess A ist rechnend. Innerhalb A ist Thread A.2 rechnend. Thread A.2 ruft blockiernde Funktion der Thread-Bibliothek auf z.B. warten auf Ereignis von Thread A.1

- Prozess A bleibt rechnend (falls Zeitscheibe noch nicht abgelaufen ist),
- Thread A.2 wird blockiert und Thread A.1 wird rechnend.

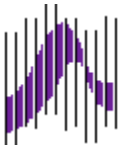
Multithreading
Benutzer-Threads



- Vorteile:
  - Threadwechsel wird im User-Modus abgearbeitet. Kein SVC und daher wesentlich schneller.
  - Prozessspezifisches Scheduling der Threads unabhängig vom Scheduling des BS-Kerns.
  - Die Bibliothek kann auf jedes beliebige Betriebssystem aufgebaut werden. Es sind keinerlei Änderungen im Kernel notwendig.

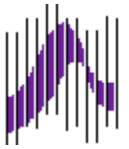
- Nachteile:
  - Blockiert ein Thread durch einen SVC blockiert auch der Prozess und damit auch alle anderen Threads im Prozess.
  - Mehrprozessorbetrieb ist nicht nutzbar. Alle Threads eines Prozesses laufen auf einem Prozessor ab.

Multithreading
Kernel-Threads



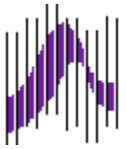
- Kernel übernimmt die Verwaltung von Threads und Prozessen
- Das Scheduling erfolgt im Betriebssystem auf Basis der Threads.
- Vorteile:
  - Unterstützt Mehrprozessoren-Archtiketur
  - Ein blockierter Thread blockiert nicht andere Threads im gleichen Prozess.
- Nachteil:
  - Ein Threadwechsel bedeutet immer auch aufrufen eines SVC und damit Übergang in den privilegierten Modus.

Prozesse unter UNIX - Prozessarten



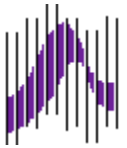
- Interaktive Prozesse
  - Shell-Prozesse arbeiten mit stdin (Tastatur) und stdout (Bildschirm)
- Dämon-Prozesse
  (daemon = Disc And Execution MONitor)
  - arbeiten im Hintergrund ohne stdin und stdout z.B. Druckspooler, Swapper, Paging Dämon, cron, ... (unter Windows: services)
  - jedem run-level (Systemzustand) sind deamons zugeordnet: /etc/rcX.d/*

Prozesse unter UNIX



- Prozesse werden in einer Prozesstabelle und PCB‘s verwaltet. Ein Prozess wird unter Unix auch als Task bezeichnet.
  - Jeder Eintrag in der Prozesstabelle verweist auf den PCB des Prozesses.
- Jeder Prozess wird durch eine eindeutige PID verwaltet und verfügt über eine Gruppen-ID und Benutzer-ID.
  - Über Gruppe und Benutzer werden Zugriffsrechte auf Ressourcen geregelt.

Prozesse unter UNIX - Erzeugen und Terminieren



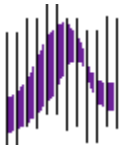
- Prozess erzeugen
  - Systemfunktion pid = fork()
    Es wird eine Kopie des aktuellen Prozesses erzeugt.
    Ergebnis der Funktion (gespeichert in pid):
    - < 0 wenn Prozess nicht erzeugt werden konnte
      (z.B. Prozesstabelle ist voll)
    - = 0 im Kind-Prozess
    - >0 im Eltern-Prozess; Variable pid enthält die Prozessidentifikation der erzeugten Kopie d.h. des Kind-Prozesses

- freiwilliges Ende
  - exit (...) (wird vom Kind-Prozess aufgerufen)

- Ausführen eines neuen Programms
  - exec(....) (wird vom Kind-Prozess aufgerufen)

- warten auf das Ende eines erzeugten Kind-Prozesses
  - wait(....), waitpid() (wird vom Eltern-Prozess aufgerufen)
    waitpid() kann auch nicht-blockierend verwendet werden.

Prozesse unter UNIX



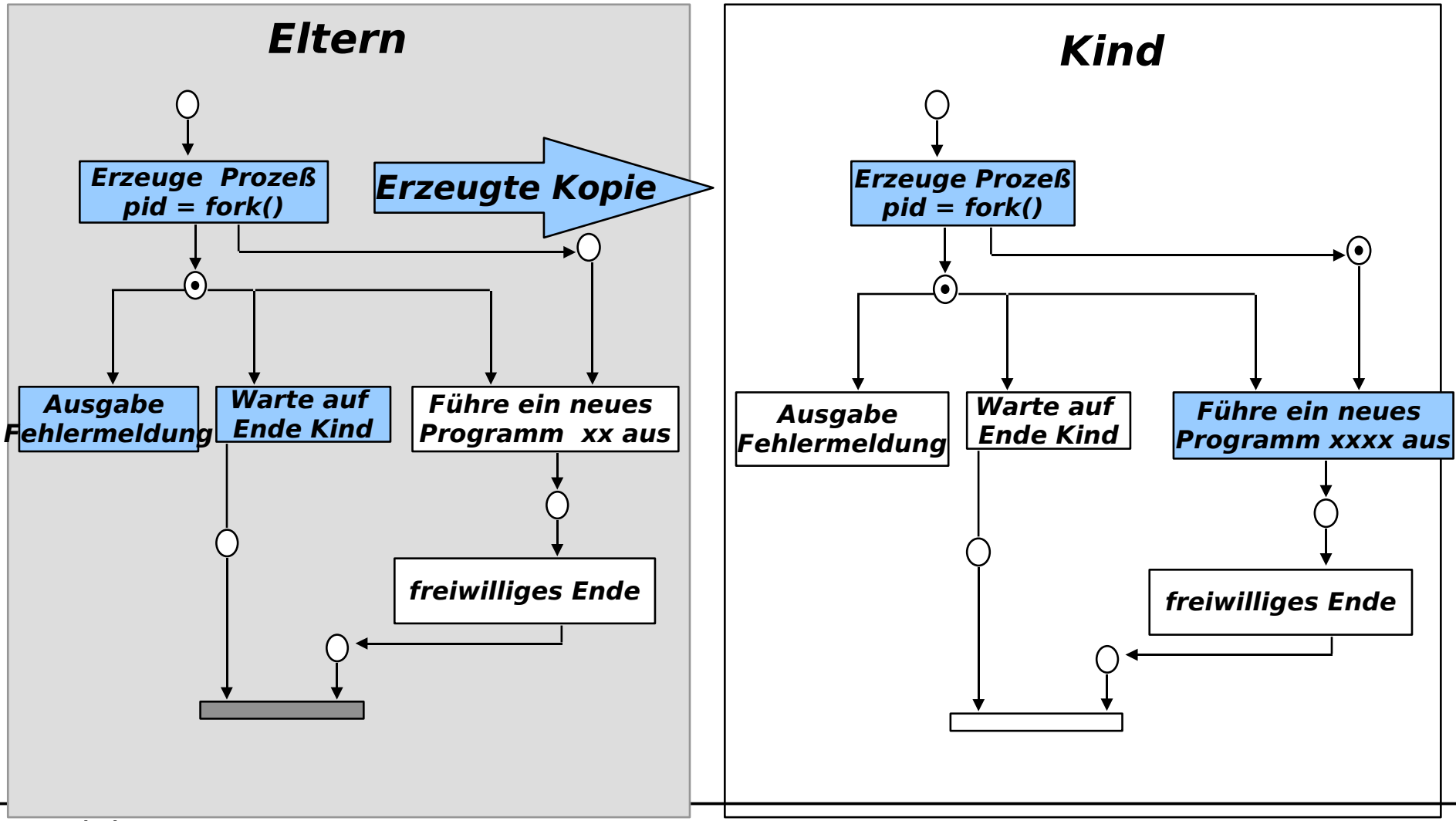
- Jeder Prozess kennt seinen Eltern-Prozess, seine Geschwister und alle seine Kinder
- Terminiert ein Kind-Prozess, so wird er in den Zustand **Zombie** versetzt bis der Eltern-Prozess den Terminierungsstatus vom BS abgeholt hat.
- Terminiert der Eltern-Prozess, so werden alle existierenden Kind-Prozesse einem neuen Eltern-Prozess zugeordnet.
  - Dies ist immer der Prozess INIT (Pid = 1).
- Möchte der Eltern-Prozess nicht auf das Ende eines Kind-Prozesses warten, sondern ohne zu warten nur den Terminierungsstatus abholen, um sein Kind-Prozess aus dem Zustand Zombie zu erlösen, kann er die Systemfunktion waitpid() verwenden.

Prozesse unter UNIX - Erzeugen

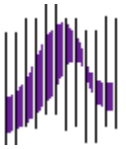

```
.
.
.
.

pid = fork();
if ( pid <0 )  printf("Prozeß wurde nicht erzeugt\n");
else
    if ( pid == 0 )
        /* Starten eines neuen Programms im erzeugten Kindprozess */
       execl(......)
     else
         /* Elternprozeß wartet auf Ende des Kindprozesses */
         wait(...)
```

Prozesse unter UNIX erzeugen

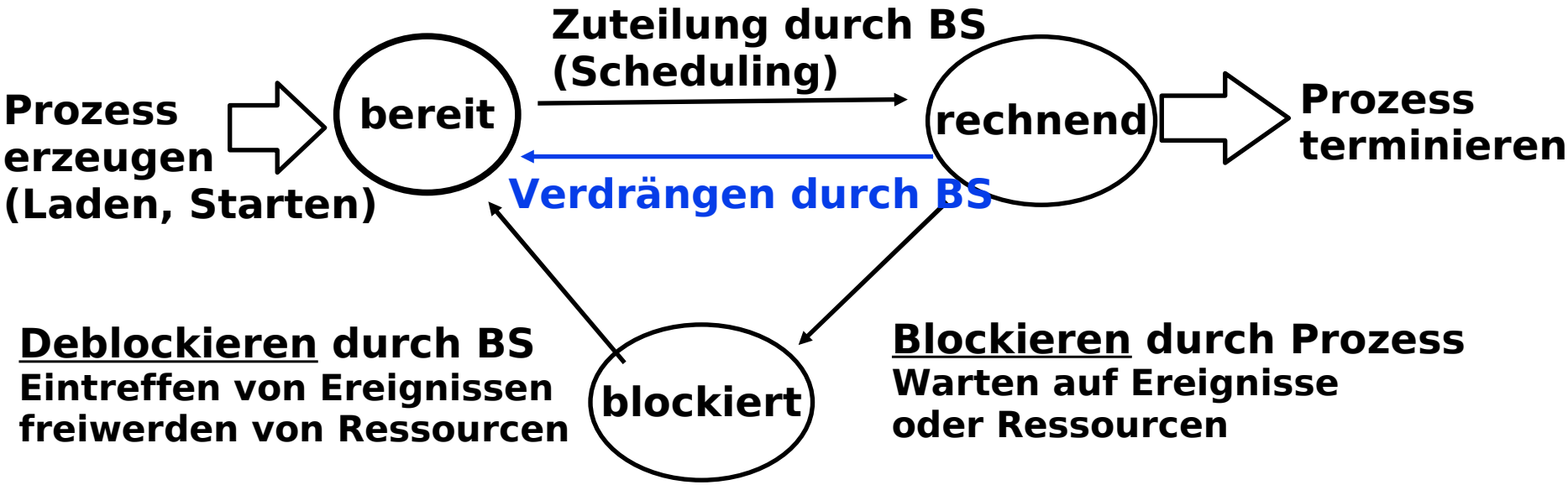


Prozesse unter UNIX – weitere Systemfunktionen



- Eigene PID erfragen pid()
- PID des Elternprozesses erfragen ppid()

Multitasking - Prozesszustände



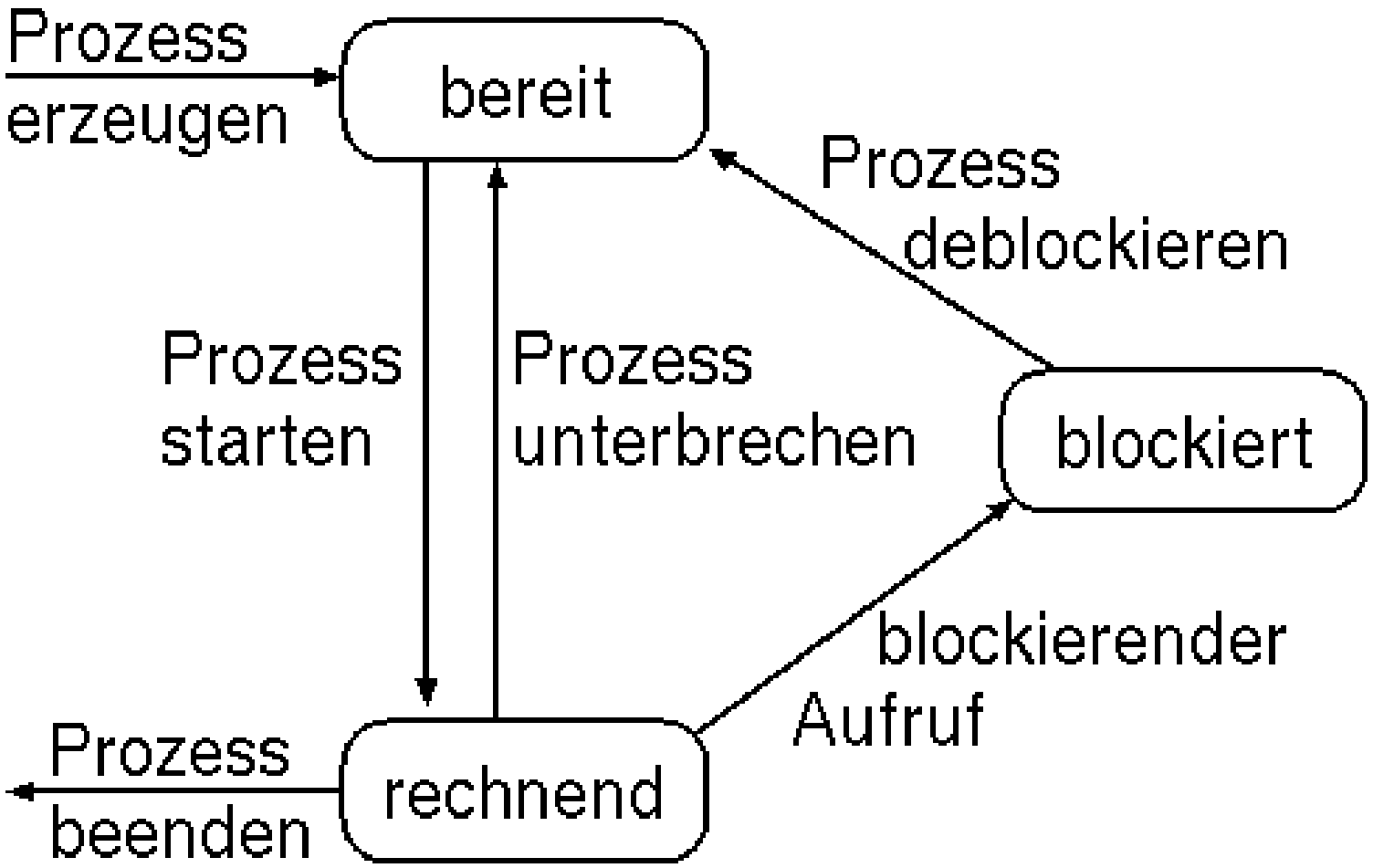
preemptive Scheduling:

Prozesswechsel wird vom Betriebssystem erzwungen (Zeitscheibenverfahren)

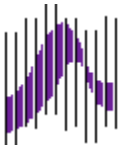
non preemptive Scheduling:

Prozesswechsel erfolgt nur durch freiwillige Abgabe der CPU durch einen Prozess (Systemaufrufe im Programm)

Prozess- bzw. Thread-Zustände

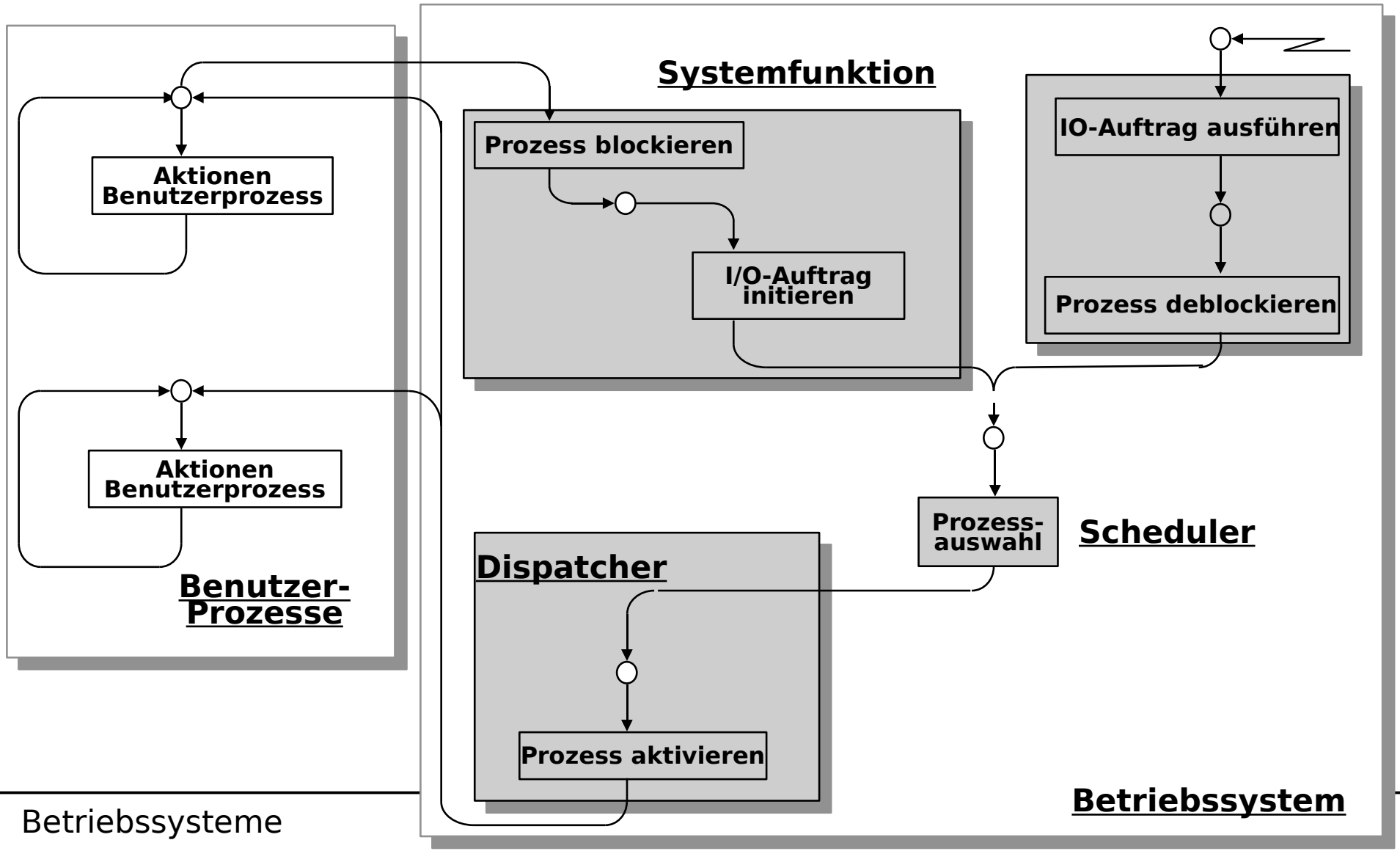


Scheduling/Dispatching

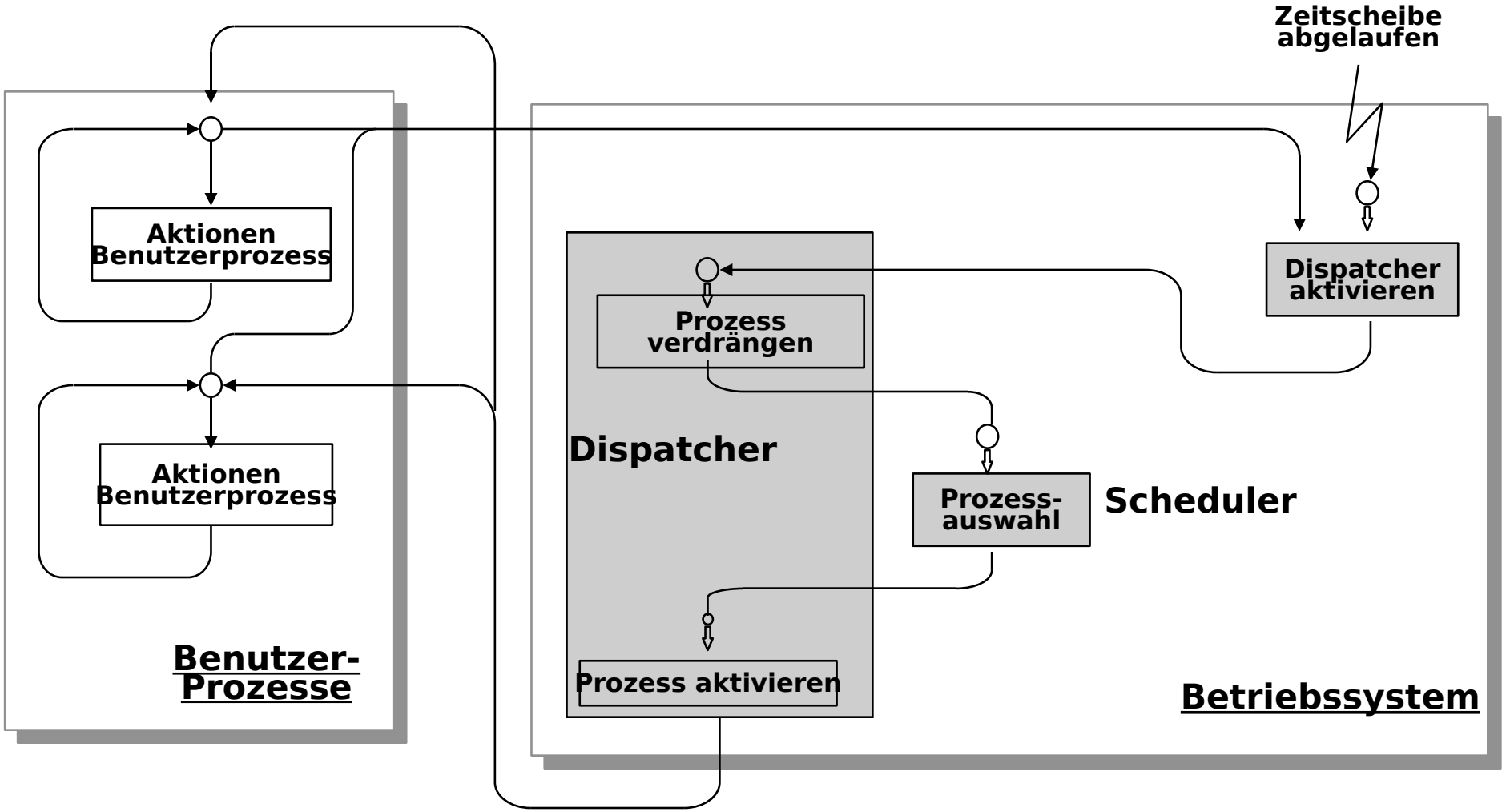


- Scheduling
  - Auswahl des Prozess, der als nächstes rechnen darf
  - Der Scheduling-Algorithmus betrachtet nur Prozesse im Zustand „bereit“.
- Dispatching
  - Auslagern des laufenden Prozesses: Prozess-Zustand sichern
  - Einlagern des ausgewählten Prozesses: Prozess-Zustand rekonstruieren

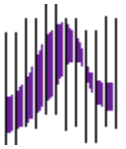
Multitasking
Prozesswechsel durch Systemaufruf



Multitasking
Prozesswechsel durch Zeitscheibe

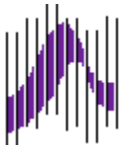


Ziele des Schedulings (Prozesse/Threads)



- Alle Systeme:
  - Fairness (jeder Prozess erhält Rechenzeit)
  - Policy Enforcement (Prioritäten einhalten)
  - Auslastung (alle Komponenten sind ausgelastet)
- Batch-Systeme
  - Durchsatz
  - Bearbeitungszeit/Verweilzeit
- Dialog-Systeme
  - Antwortzeit
- Echtzeitsysteme
  - Fristen einhalten (meeting deadlines)
  - Gleichförmiges, vorhersehbares Verhalten (predictable)

Scheduling für Batch-Systeme



First-Come First-Served (keine Unterbrechung)

- Vorteil:
  - einfach zu implementieren
- Nachteil:
  - evtl. schlechte Auslastung
  - evtl. schlechter Durchsatz

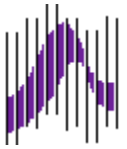
Shortest Job First (keine Unterbrechung)

- Laufzeit muss vorab bekannt sein

Shortest Remainig Time Next

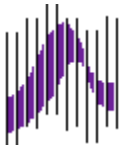
- verbleibende Laufzeit muss bekannt sein

Scheduling für Echtzeitsysteme



- Prioritätsbasiertes Scheduling
- Raten Monotones Scheduling
  - Mehrere periodisch aktive Prozesse
  - Priorität ist umgekehrt Proportional zur Laufzeit
- Earliest Deadline First
  - Jeder Prozess gibt seine nächste Deadline an
  - Der Scheduler sortiert die Prozesse nach Deadline und führt sie entsprechend aus

Scheduling für Dialog-Systeme



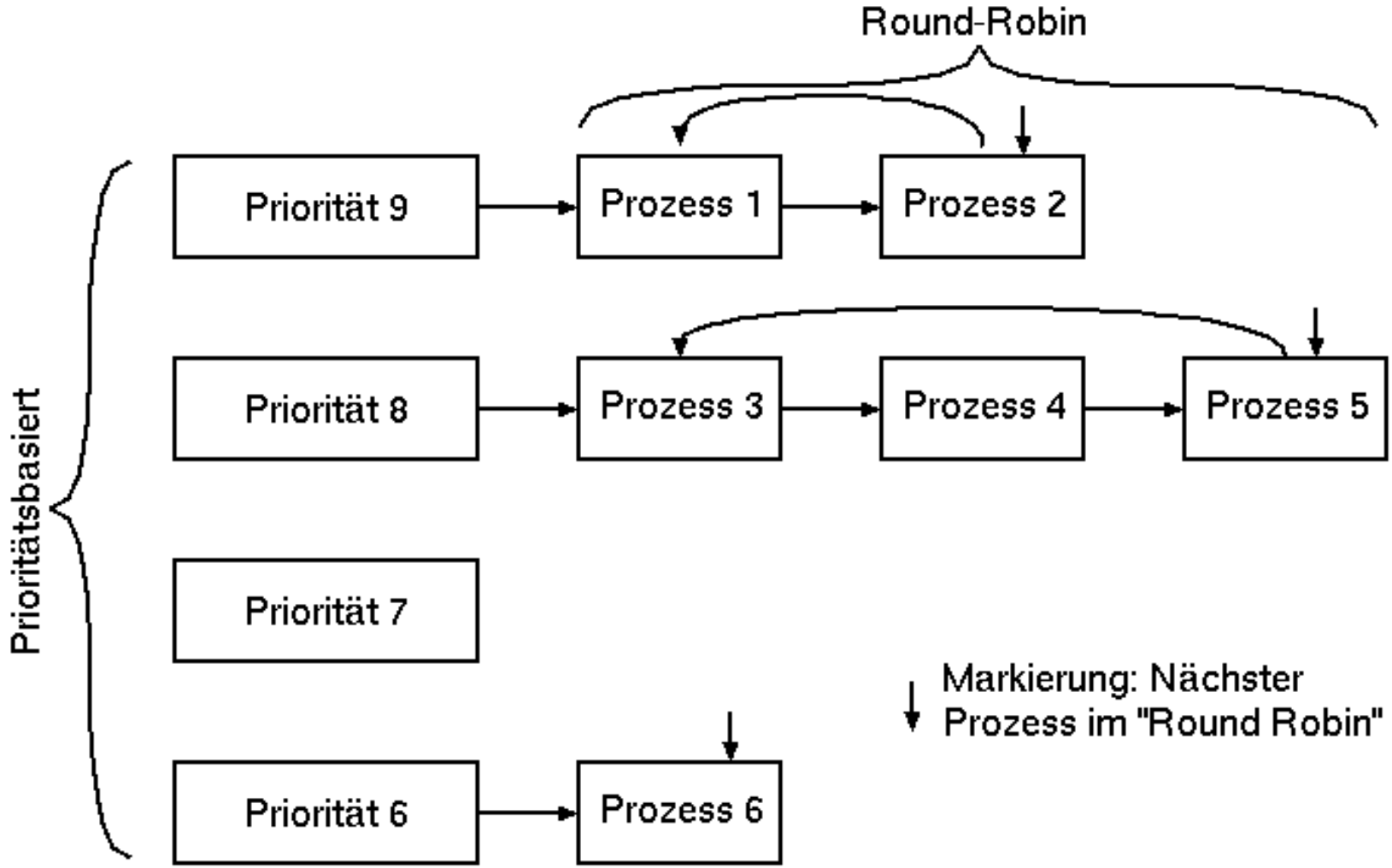
Round-Robin

- Alle Prozesse bekommen der Reihe nach eine Zeitscheibe
- Kritische Größe: Länge der Zeitscheibe
  z.B. 4 - 100ms

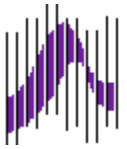
Prioritätsbasiertes Scheduling

- der Prozess mit höchster Priorität läuft
  - evtl. nicht fair
- Laufen verringert Priorität (und oder)
- Warten erhöht Priorität

Scheduling für Dialog-Systeme (2)



Scheduling für Dialog-Systeme (3)

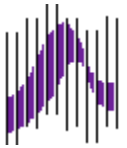


Lottery-Scheduling

- Jeder Prozess bekommt „Lose“
- Scheduler wählt zufällig eine Losnummer
- Der Gewinner-Prozesse darf laufen

Die CPU-Zeit wird – im Mittel - gemäß der Anzahl der Lose verteilt.

Scheduling für Threads



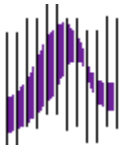
User Level Threads

- Anwendung / Thread-Blibliothek übernimmt das Scheduling
- Algorithmus kann an die Anwendung angepasst werden

Kernel Threads

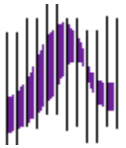
- Kernel übernimmt Scheduling
- Threads im bereits laufenden Prozess werden bevorzugt (kein Prozesswechsel)

Multithreading in Linux



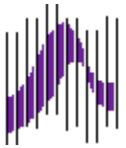
- Unter Linux werden Threads durch einen Systemaufruf „clone()“ erzeugt
- User-API: PThread-Bibliothek
- Linux verwaltet Threads in verschiedenen Varianten
- Ein neuer Thread kann diverse Ressourcen des „Eltern-Threads“ verwenden.
  - Verzeichnisse
  - Datei-Deskriptoren
  - Signalhandler
  - PID
- Ein Thread kann auch ein neuer Prozess sein.

Scheduling Algorithmen



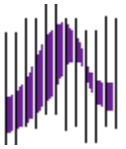
- Kategorien
  - Stapelverarbeitung
    - Minimierung der Prozesswechsel um einzelne Jobs mit schneller Geschwindigkeit abzuarbeiten
    - Maximaler Durchsatz (Jobs pro Zeiteinheit)
    - Non preemtive, preemptive mit langen Zuteilungszeiten
  - Interaktion
    - Möglichst schnelle Antwort auf Interaktionen des Benutzers
    - Fairness – jeder Benutzerprozess bekommt anteilig Rechenzeiten auf der CPU zugeteilt
    - Preemptive ist ein Muss – ein Prozess darf die CPU nicht für sich alleine belegen
  - Echtzeit
    - Prioritäten werden verlangt, Unterbrechungen sind nur für die Behandlung der Handware dringend notwendig

Scheduling Feedback



- Kriterium zur Auswahl = momentan vergangene Rechenzeit eines Prozesses
- Preemptive: CPU wird dem Prozess nach einer Zeitscheibe durch das BS entzogen
- Jedem Prozess werden Prioritäten dynamisch vergeben. Es gibt n Prioritäten. Beim Start gelangt der Prozess in die Priorität 0. Nach Ablauf einer Zeitscheibe wird er auf die Priorität 1 gesetzt. Nach Ablauf von i Zeiteinheiten wird er auf die Priorität i gesetzt. Für jede Priorität existiert eine Warteschlange die nach dem Prinzip FCFS abgearbeitet wird.
- Die Warteschlange mit höchster Nummer hat die niedrigste Priorität. Diese wird nach dem Verfahren RR abgearbeitet.
- Nachteil: Prozesse mit hoher Rechenzeit können verhungern (starvation)

Scheduling Feedback



Beispiel

- Prozesse:
  A (RZ=20)
  B (RZ=3)
  C (RZ=5)
  D (RZ=10)
- Alle Prozesse starten alle zur gleichen zeit t0.
- ABCDABCDABCDACDACDADADADADADAAAAA
  A

Scheduler in UNIX

- **Timesharing nach Round Robin**

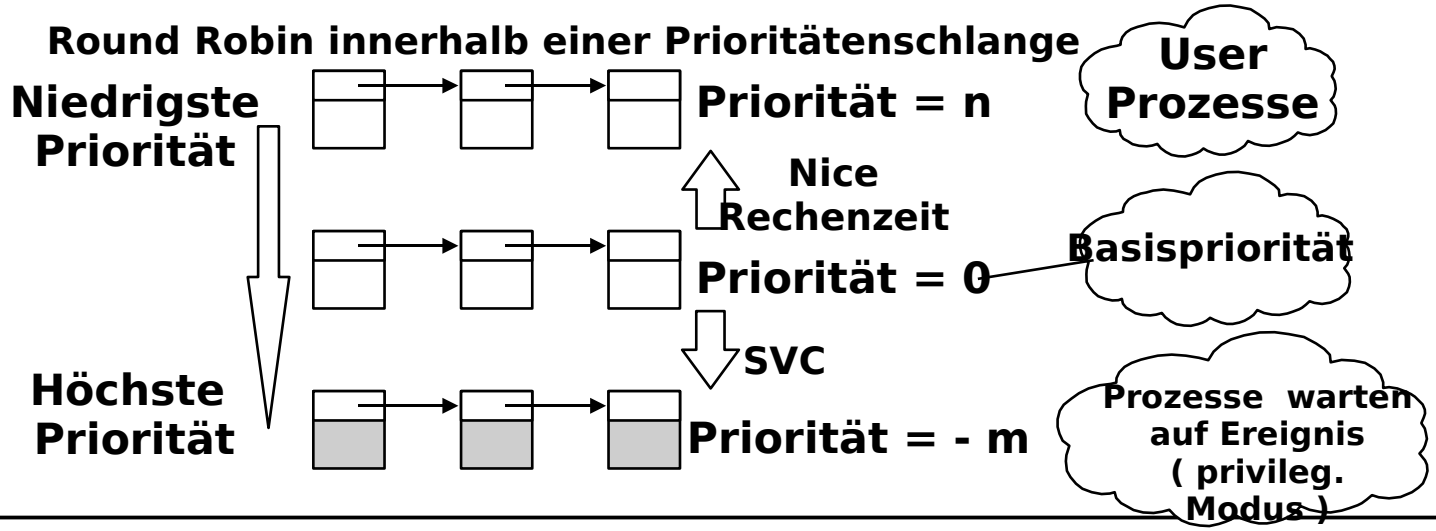
  Ziel : Optimieren der Antwortzeiten von interaktiven Prozessen
  Zeitscheibe: ca. 4ms - 100ms

- **Zuteilung der CPU nach Prioritäten**

Die Priorität eines Prozesses wird dynamisch entsprechend seiner Aktiviäten verändert. Rechneintensive Prozesse werden bestraft. Interaktive Prozesse werden belohnt.

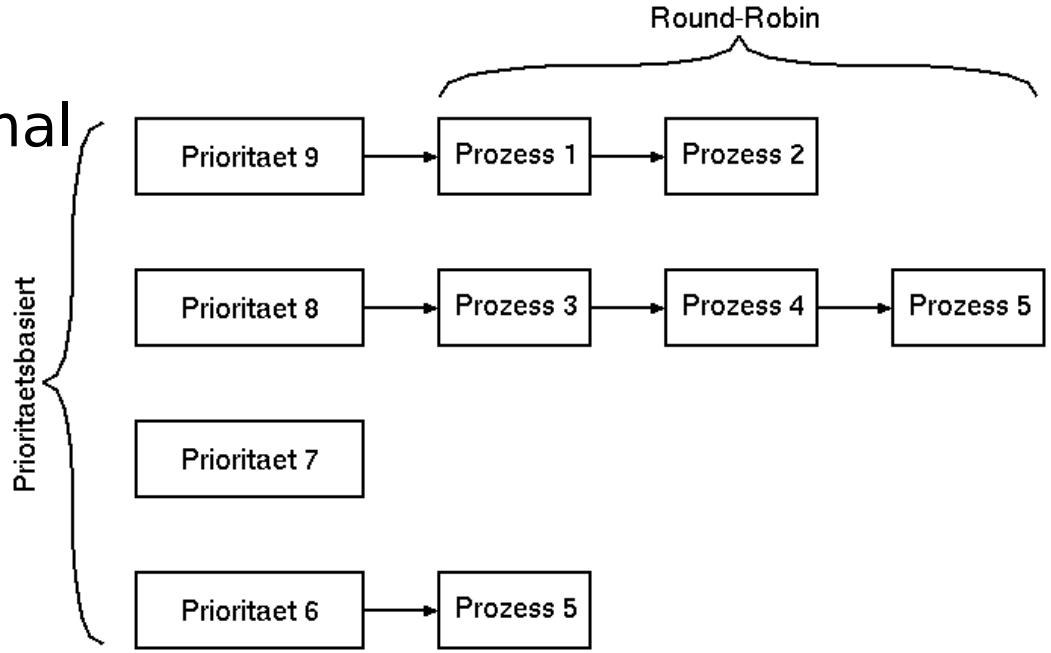
Auslastung der CPU durch den Prozeß: a
Der Wert von a wird mit jedem Zeittakt um 1 erhöht. Nach jeder Sekunde werden die Prioritäten neu berechnet

a:= a/2
Priorität:= Basispriorität +a/Konstante +Nice-Wert



Scheduler in Linux

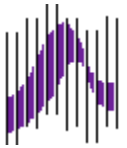
Die Priorität eines Prozesses wird einmal pro Sekunde neu berechnet.



Priorität = f(BasisPriorität, nice-Wert, CPU-Nutzung)

Prozesse, die im Kernel-Mode warten, erhalten eine höhere Priorität. Diese hängt davon ab, auf welches Ereignis sie warten.

Prozesse in Windows NT / 2000



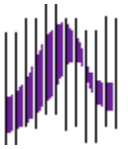
- Zwischen Prozessen besteht keine Vater/Sohn-Beziehung.

- Objektorientierte Implementierung d.h. für jeden Prozess wird ein Systemobjekt angelegt, in dem die Eigenschaften des aktuellen Prozesses gespeichert sind.

- Das Scheduling erfolgt nur über Threads. Damit muß ein Prozess mindestens einen Thread besitzen, damit er lauffähig ist.

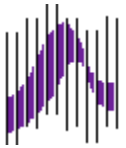
- Ein Prozess kann über die Systemfunktion CreateProcess() dynamische zur Laufzeit mehrere Prozesse erzeugen.

Objektklasse Prozess in Windows NT



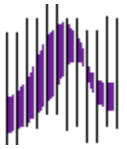
Objektklasse PROZESS	
Process ID **Access token** **Base priority** **Default processor affinity** **Quota limits** **Execution time** **I/O counters** **VM operation counters** **Execution/debugging ports** **Exit status**	**Objektattribute**
Create process **Open process** **Query process information** **Set process information** **Current process** **Terminate process** **Allocate/free virtual memory** **Read/write virtual memory** **Protect virtual memory** **Lock/unlock virtual memory** **Query virtual memory** **Flush virtual memory**	**Objektmethoden**

Prozesse in Windows NT / 2000



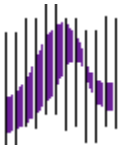
- Eigenschaften eines Prozesses
  - Process ID :
    - Eine eindeutige Nummer, die NT für jeden erzeugten Prozess vergibt, um damit den Prozess eindeutig identifizieren zu können
  - Access token:
    - Eine Referenz auf ein NT-Objekt, in dem Sicherheits- und Zugriffsinformationen über den eingeloggten Benutzer, zu dem der Prozess gehört, abgelegt sind.
  - Base priority:
    - Eine Basispriorität für den Prozess. Beim starten des Prozesses wird die Priorität auf diese Basispriorität gesetzt.
  - Default processor affinity:
    - Eine vorgegebene Menge von Prozessoren, auf denen die Threads eines Prozesses laufen können.
  - Quota limits:
    - Maximalwerte für die Anzahl von physikalischen Seiten, Größe der Paging-Datei, und Rechenzeit
  - Execution time:
    - Die aktuell von allen Threads verbrauchte Rechenzeit
  - I/O counters:
    - Zähler für die Anzahl der I/O Operationen
  - VM operation counters:
    - Zähler für die ausgeführten Speicheroperationen
  - Exit status:
    - Ein Wert, der den Grund für ein Prozessende beschreibt.

Prozesse in Windows NT / 2000



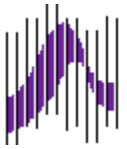
- Methoden
  - Prozesserzeugung CreateProcess()
  - Auf das Ende eines Prozesses warten WaitForSingleObject()
  - Abfrage von Prozessinformationen
  - Einstellen von Prozessinformationen
  - Prozessterminierung (freiwillig) ExitProcess()
  - Prozessterminierung (unfreiwillig) TerminateProcess()

Scheduler in Windows NT



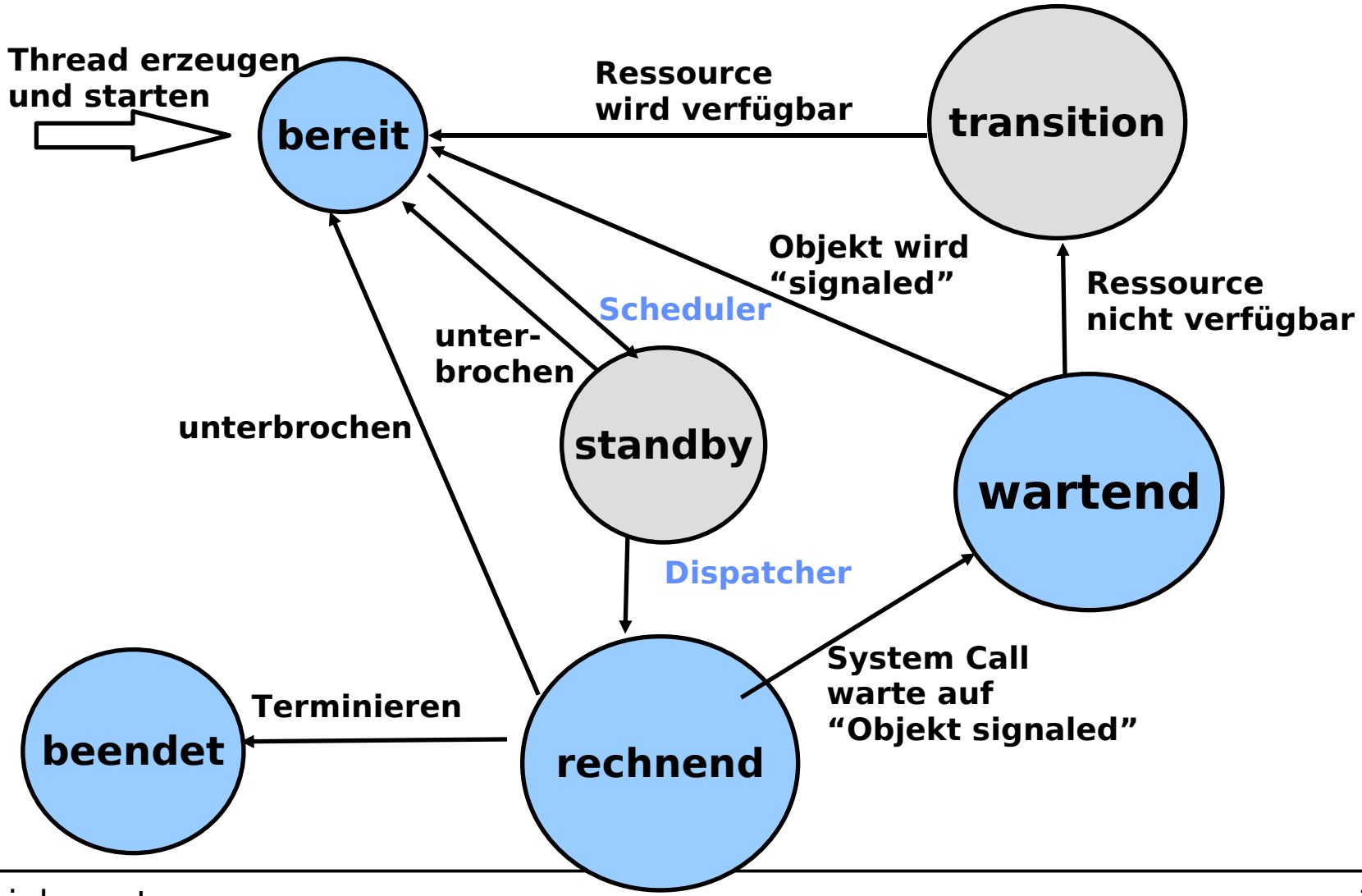
- **Variable Prioritäten**
  - Preemptives Zeit- und Prioritätengesteuertes Scheduling
    - Die Prioritäten 1..15 werden zur Laufzeit vom Kernel verändert mit dem Ziel interaktive Prozesse mit hoher Priorität, I/O-intensive Threads mit mittlerer Priorität und rechenintensive Prozesse mit niedriger Priorität zu behandeln.
    - Die Priorität wird erhöht, wenn ein Prozess vom Zustand „wartend“ in den Zustand „bereit“ gelangt
    - Die Priorität wird erniedrigt, wenn ein Thread nach Ablauf einer Zeitscheibe unterbrochen wird.
    - Die Priorität wird minimal zur Basispriorität und maximal auf 15 verändert.
    - Realzeitprioritäten (16 .. 31) werden nicht vom System geändert.
- **Ein Thread erbt die Basispriorität vom Prozess**
  -
  -
  - Ein Thread kann die vererbte Basispriorität um maximal 2 nach oben oder nach unten verändern
  -
  -

Multithreading Windows NT/2000

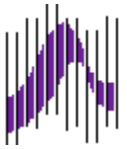


- Threads werden als Kernel-Threads realisiert
- Ein Prozess kann mehrere Threads besitzen
- Scheduling in NT/2000 erfolgt ausschließlich über Threads, d.h. in Prozess besteht aus mindestens einem Thread.
- Ein Thread kann selbst wieder Threads erzeugen.
- Eigenschaften eines Thread sind:
  - eine eindeutige Kennung (Thread-ID)
  - Thread-Kontext (Prozessorregister, PC, Stack, Stackzeiger)
  - Basispriorität und dynamische Priorität
  - Prozessoraffinität (Prozessoren auf der dieser Thread ausgeführt werden kann)
  - Ausführungzeit (Rechenzeit)
  - Terminierungsstatus

Threadzustände Windows NT/2000



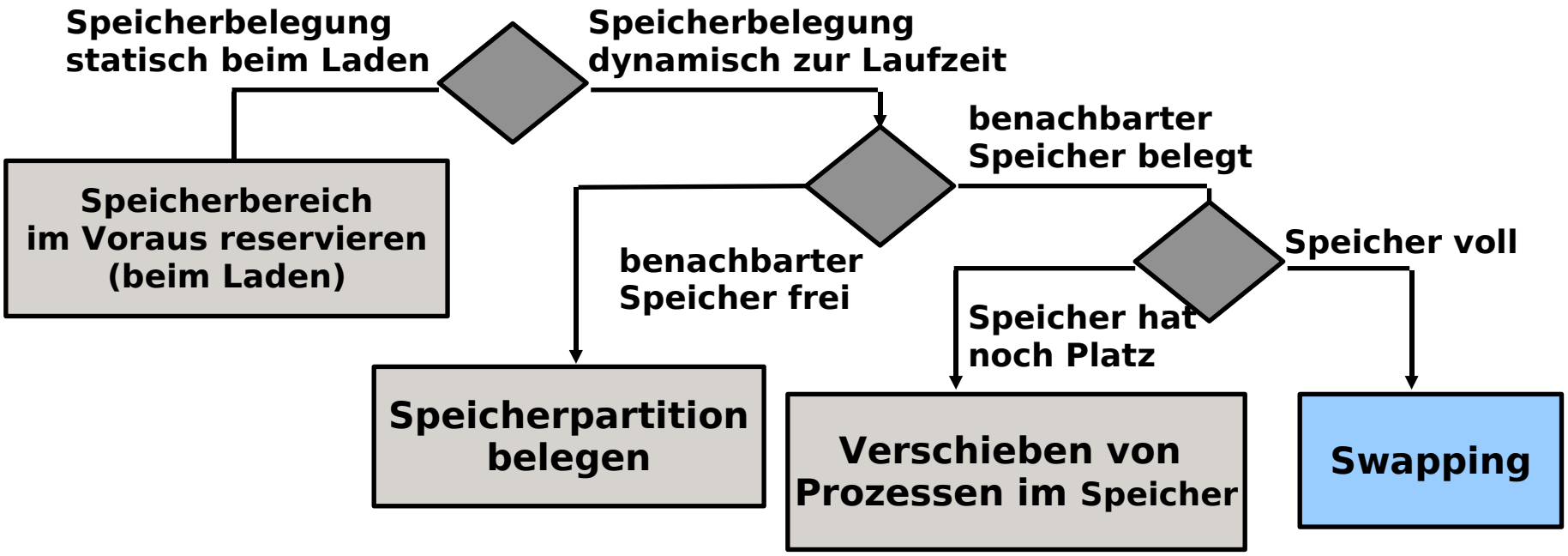
Threadzustände Windows 2000



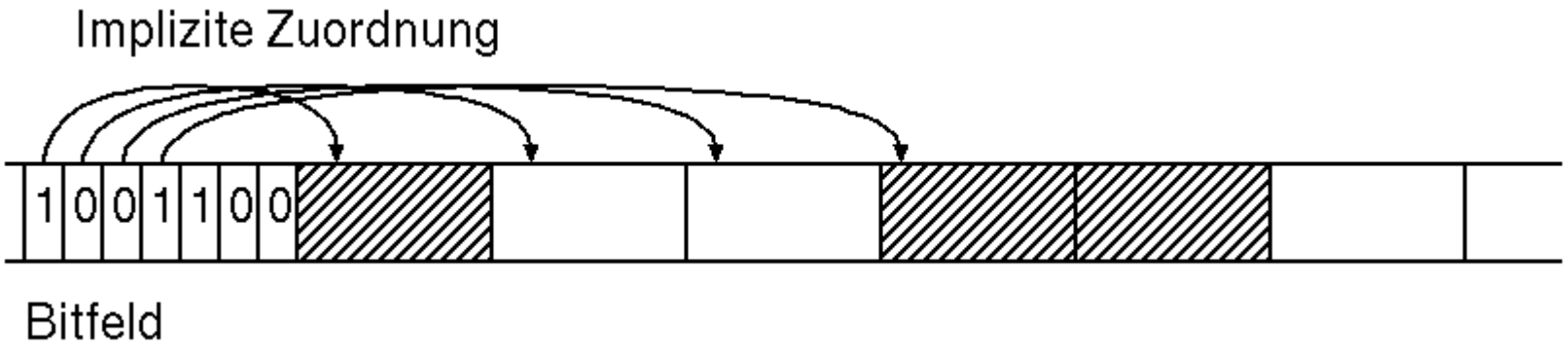
- Bereit: Kann vom Scheduler eingeplant werden
- Rechnend (aktiv): Ist gerade auf einem Prozessor rechnend
- Standby: Wurde vom Scheduler ausgewählt muss aber noch auf einen bestimmten Prozessor warten. Die Abarbeitung erfolgt sobald der Prozessor frei wird.
- Wartend/blockiert: Wartet auf Ereignis, auf I/O, auf Systemobjekte oder freiwillig ohne Grund
- transition: Grund der Blockierung ist aufgehoben kann jedoch nicht abgearbeitet werden da noch Ressourcen z.B. Speicher fehlen.
- Beendet (Terminiert): Hat sich selbst beendet oder wurde von einem anderen Thread bzw. Prozess beendet

Dynamische Speicheranforderung

- Prozesse können zur Laufzeit dynamisch Speicherblöcke anfordern bzw. freigeben
  - Beispiel: Aufbau einer verketteten Liste, Bäume, Funktionen : malloc(), free() in C, new in C++, Java
- Realisierung durch Speicherverwaltung

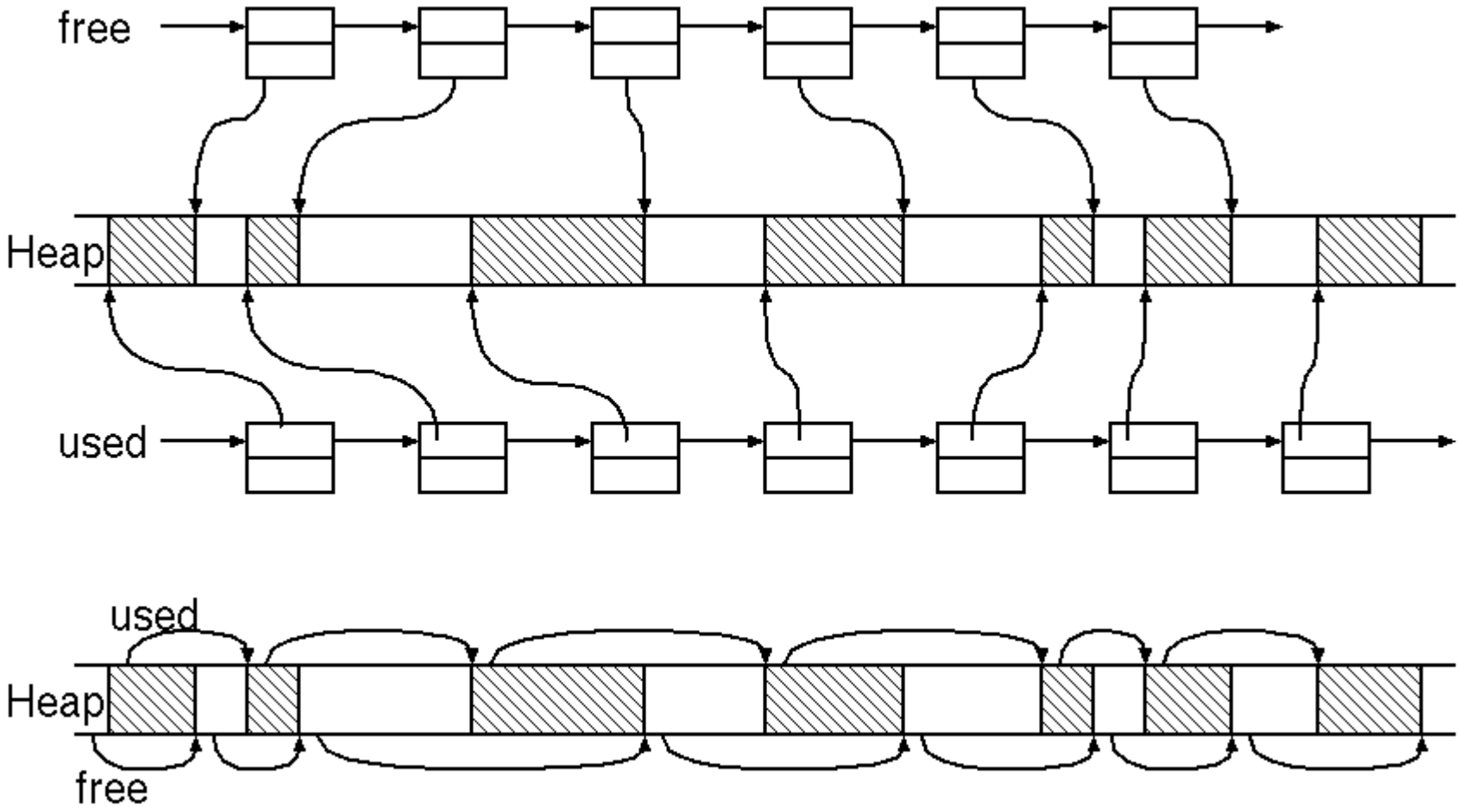


Verwaltung mit Bitliste

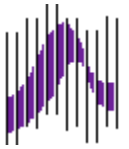


- Der Speicher wird in Blöcke fester Länge eingeteilt z.B. 4 Byte, 8 Byte bis wenige KB.
- Jedem Speicherblock ist ein Bit in einer Bitliste zugeordnet.
- Das Bit gibt an, ob der Block gerade frei oder belegt.
- Speicherblöcke können also nur als Ganzes belegt oder freigegeben werden.

Verwaltung mit Listen

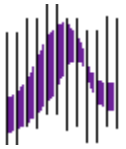


Heap-Speicher Strategien



- First-Fit
  Der Speicher wird in die erste Lücke, die groß genug ist, gelegt.
- Next-Fit
  Der Speicher wird in die nächste Lücke, die groß genug ist, gelegt.
- Best-Fit
  Der Speicher wird in die kleinste Lücke, die die groß genug ist, gelegt.
- Worst-Fit
  Der Speicher wird in die größte Lücke gelegt.

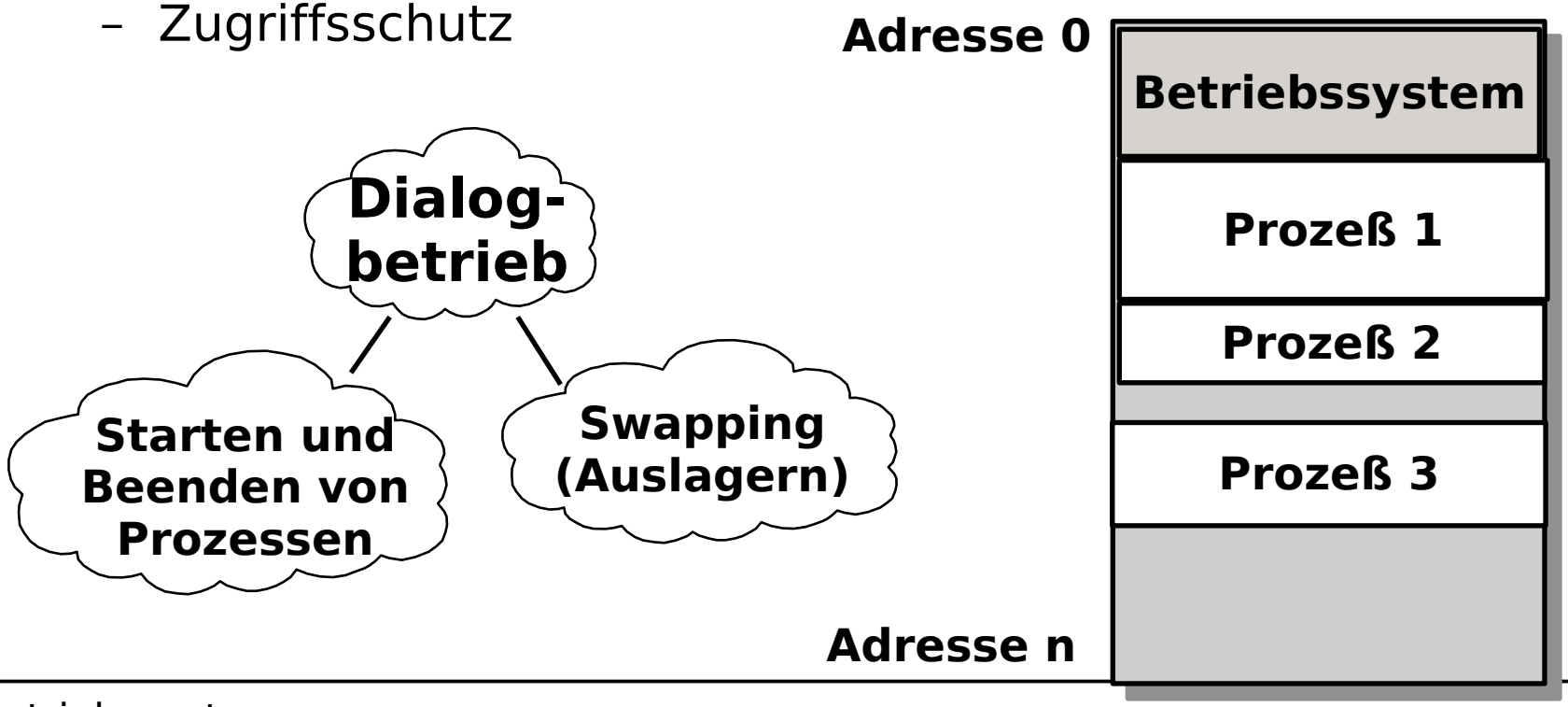
Speicher-Hierarchie



- Cache-Speicher, Größe: n
  - schnell
  - teuer
  - flüchtig
- CMOS RAM Hauptspeicher, Größe: ca. 1000*n
  - mittelschnell
  - mittelteuer
  - flüchtig (noch)
- Festplatten, SSD Größe: ca. 1 000 000*n
  - langsam
  - günstig
  - persistent

Hauptspeicherverwaltung

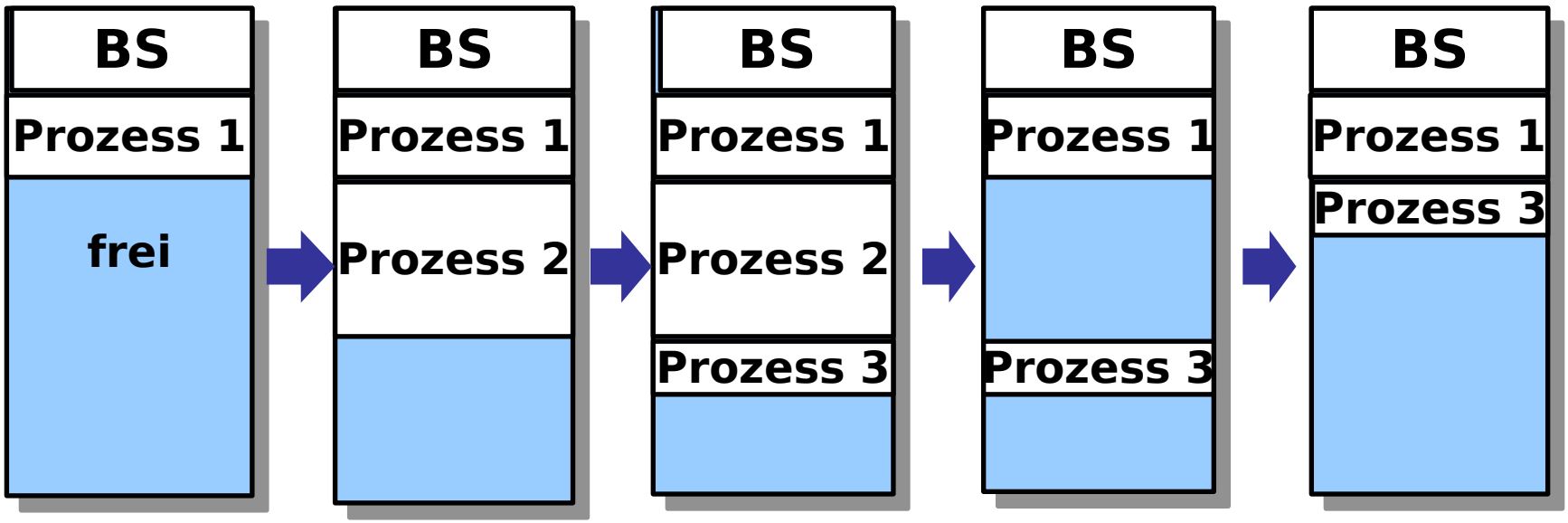
- Arbeitsspeicher muss auf mehrere Prozesse aufgeteilt werden
  - Dynamisches Einlagern und Auslagern von Prozessen
  - Verschieben von Prozessen zur Laufzeit (**relocation**)
  - Zugriffsschutz



Speicherpartitionierung

Partition

- **Feste Größe**
- **Variable Größe**



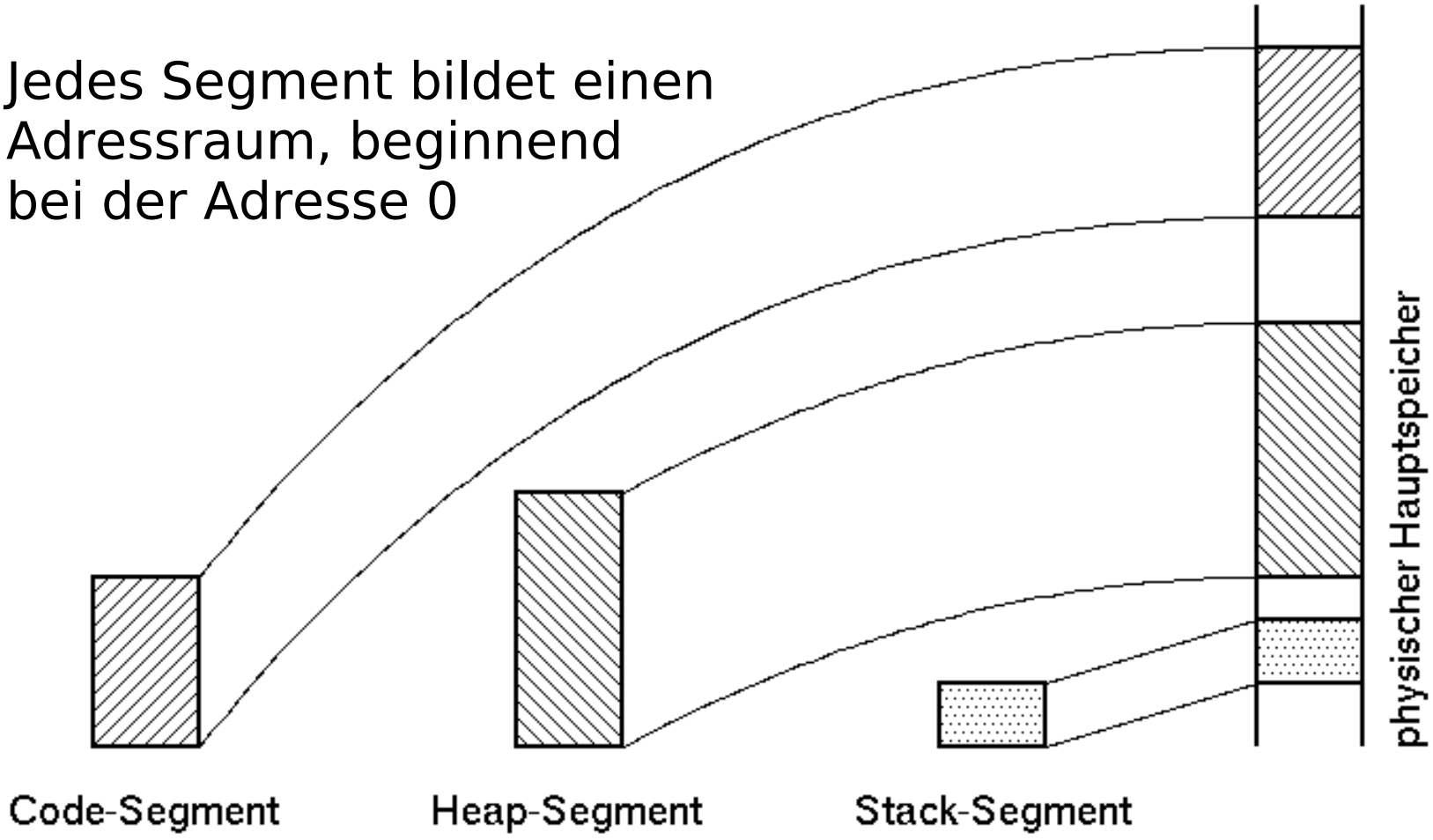
Fragmentierung

Garbage Collection

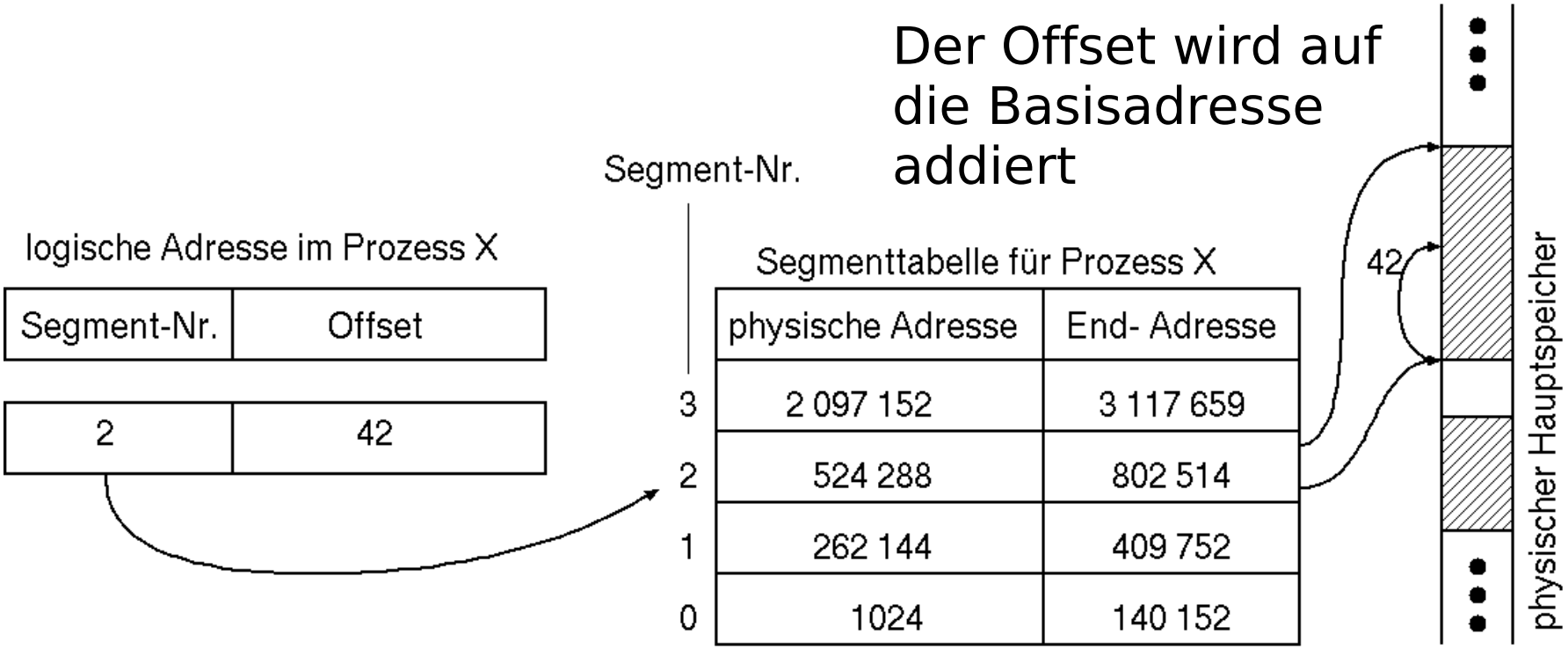
Abbildung von Segmenten

Jeder Prozess erhält ein oder mehrere Segment(e).

Jedes Segment bildet einen Adressraum, beginnend bei der Adresse 0

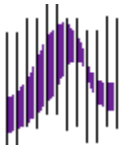


Adressierung bei Segmentierung



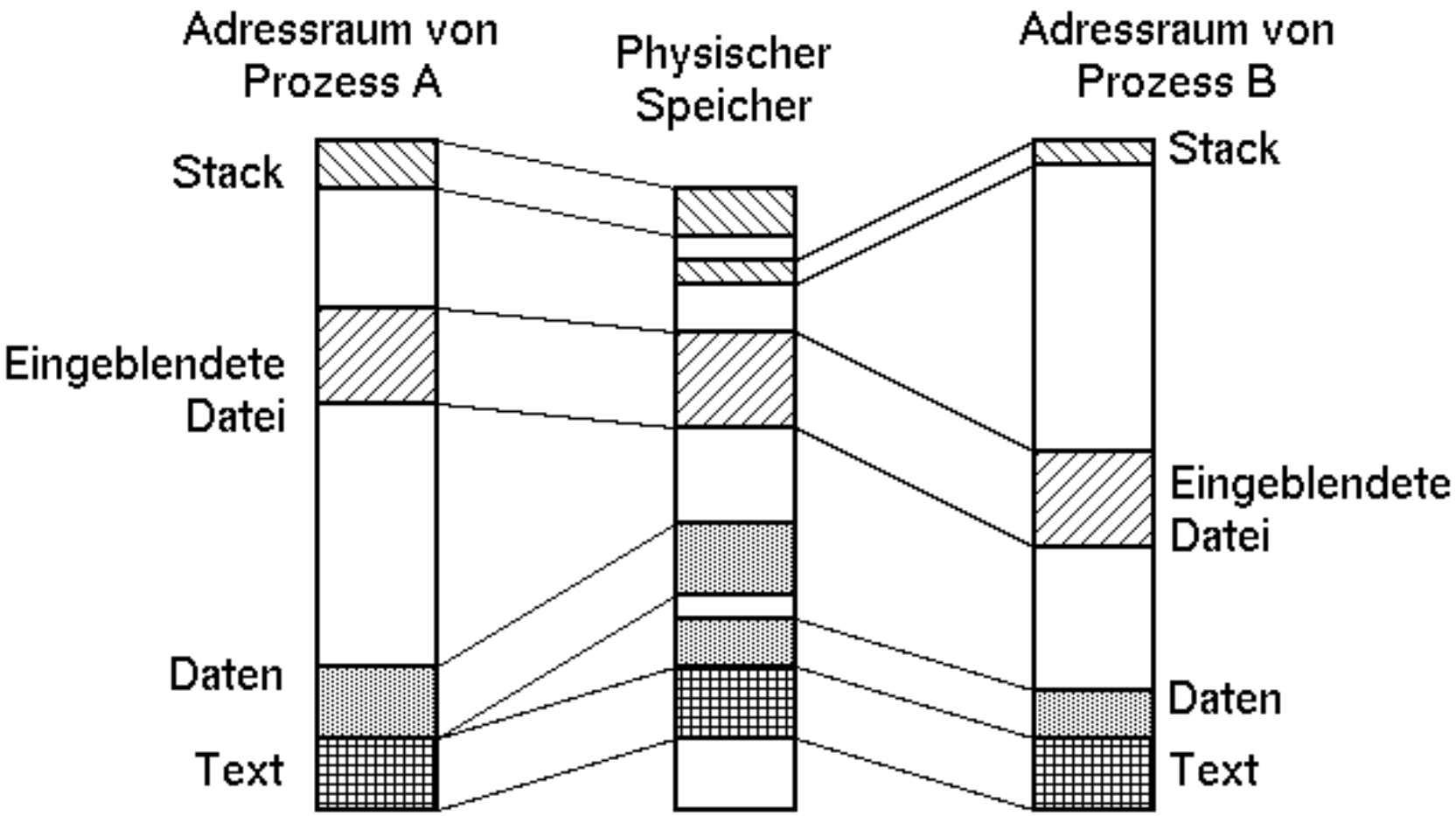
In der Segmenttabelle steht i.A. auch ein Zeiger auf das Ende des Segments. Damit können unzulässige Speicherzugriffe entdeckt und verhindert werden.

Verwaltung der Segmentinformation

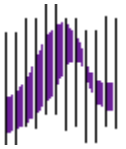


- Die Segmentnummer und der der Offset werden vom Compiler festgelegt.
- Die Segment-Basisadresse, und die Segment-Endadresse werden vom Betriebssystem festgelegt. Beide Größen können zur Laufzeit geändert werden.
- Weitere Attribute eines Segments:
  - Zugriffsrechte (lesen, schreiben, ausführen)
  - verschiebbar
  - auslagerbar
  - sharable

Shared Segments



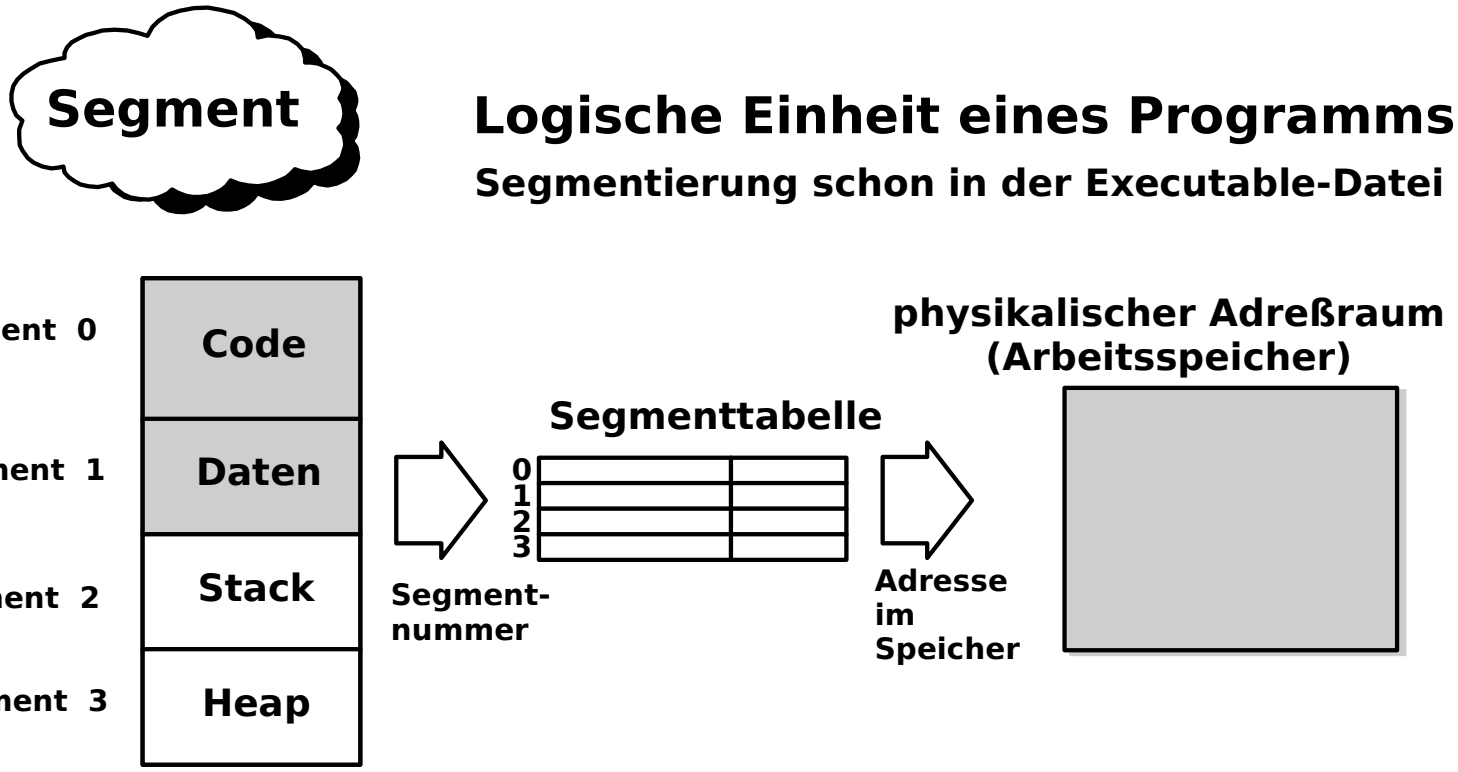
Shared Segments (2)



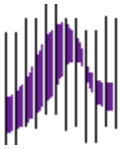
Gemeinsame Nutzung von Segmenten

- Text:
  Wird ein Programm mehr als einmal gestartet, so genügt ein Text-Segment für alle Prozesse. (Globale und static Variable getrennt halten.)
- Bibliotheken:
  Benutzen verschiedene Programme dieselbe Bibliothek, so kann diese in einem gemeinsamen Segment gehalten werden.
- Eingeblendete Dateien
- Shared Memory

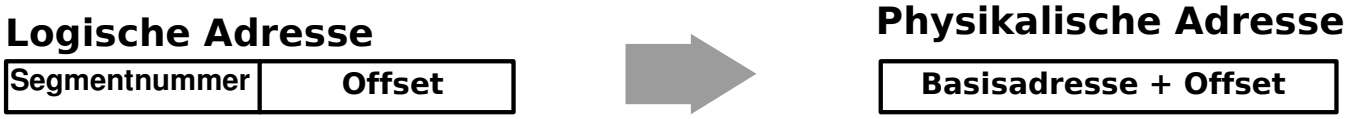
Segmentierung



Segmentierung



- **Adressumrechnung**



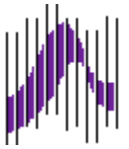
- **Eintrag einer Segmenttabelle**

Anfangsadresse	Größe des Segment	Attribute

- **Attribute eines Segments**
  - **Zugriffsrechte (lesen, schreiben, ausführen)**
  - **verschiebbar**
  - **auslagerbar**
  - **sharable**

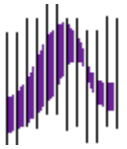
- **Compiler und/oder die CPU muss Segmentierung unterstützen**

Paging



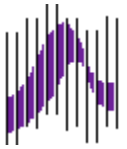
- Der physische Hauptspeicher wird in Kacheln/Rahmen gleicher Größe aufgeteilt.
- Der virtuelle Speicher der Prozesse ist in Seiten dieser Größe aufgeteilt.
- Eine virtuelle Adresse besteht aus Seitennummer und Offset.
- Bei jedem Speicherzugriff wird die Seitennummer der virtuellen Adresse auf eine Kachelnummer abgebildet.
- Seiten können auf die Festplatte ausgelagert werden.

Paging Größen und Umrechnung



- Eine virtuelle Adresse besteht aus n Bit Seiten-Nr. und k Bit Offset.
- Größe einer Seite, Größe einer Kachel: 2^k Speicherworte bzw. Byte.
- Jeder Prozess besitzt eine Seitentabelle, in der die physischen Seiten-Adressen abgelegt sind.
- Die Seiten-Nr. wird als Index in die Seitentabelle verwendet.
- Die letzten k Bit der Seiten-Adresse sind immer 0.
- Der Offset wird in die letzten k Bit eingeblendet.

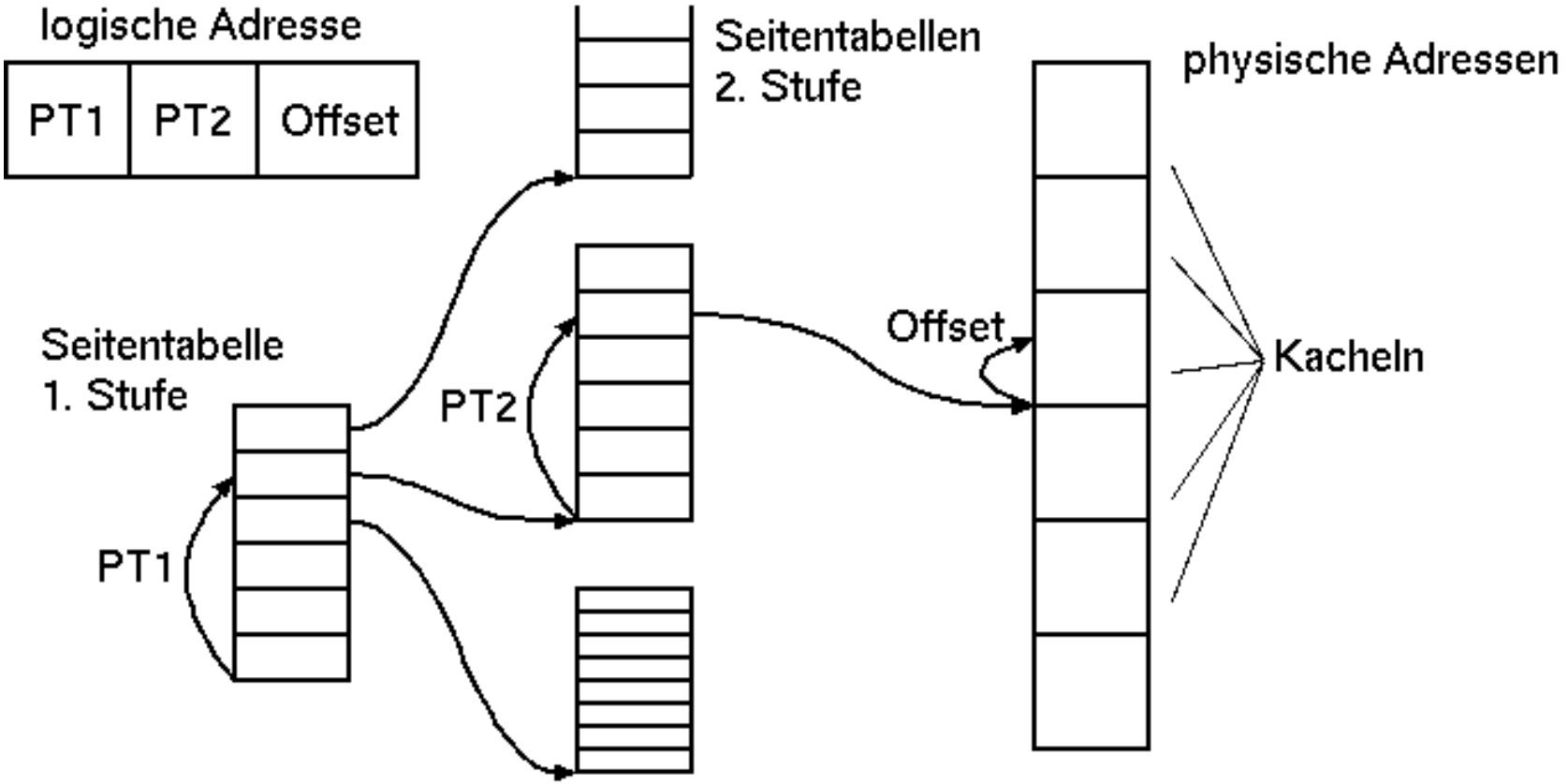
Die Seitentabelle



Die Seitentabelle muss jeder Seitennummer eine Adresse zuordnen

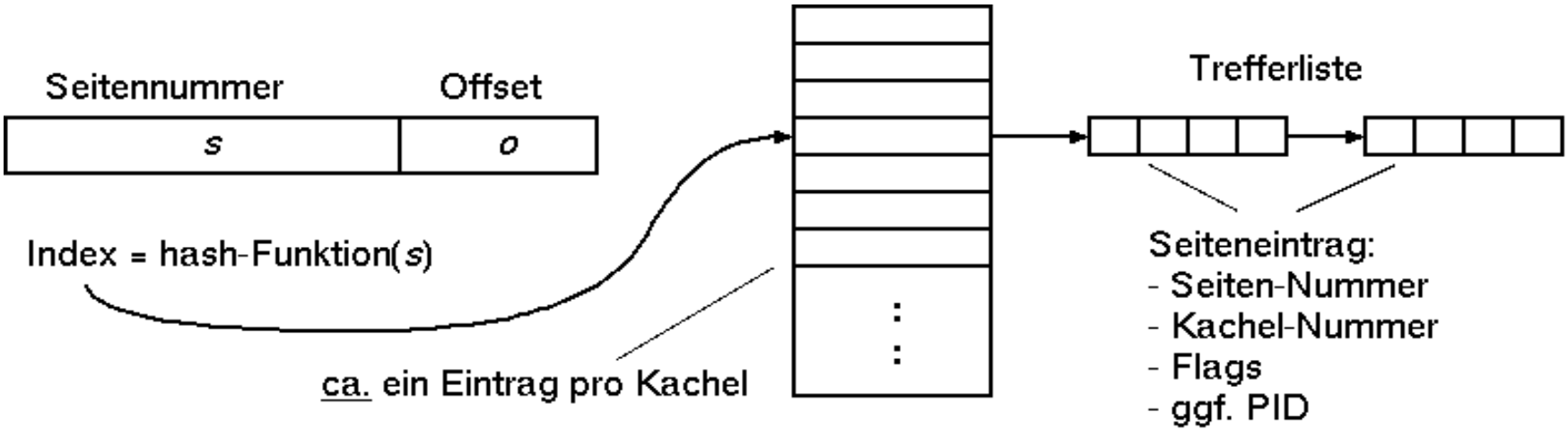
- Eine Tabelle mit 2^n Einträgen
- Ein Baum aus mehreren Tabellen (Digitalbaum)
- Eine kürzere Tabelle, z.B. mit einem Eintrag je Kachel (invertierte Liste)

Zweistufiges Paging

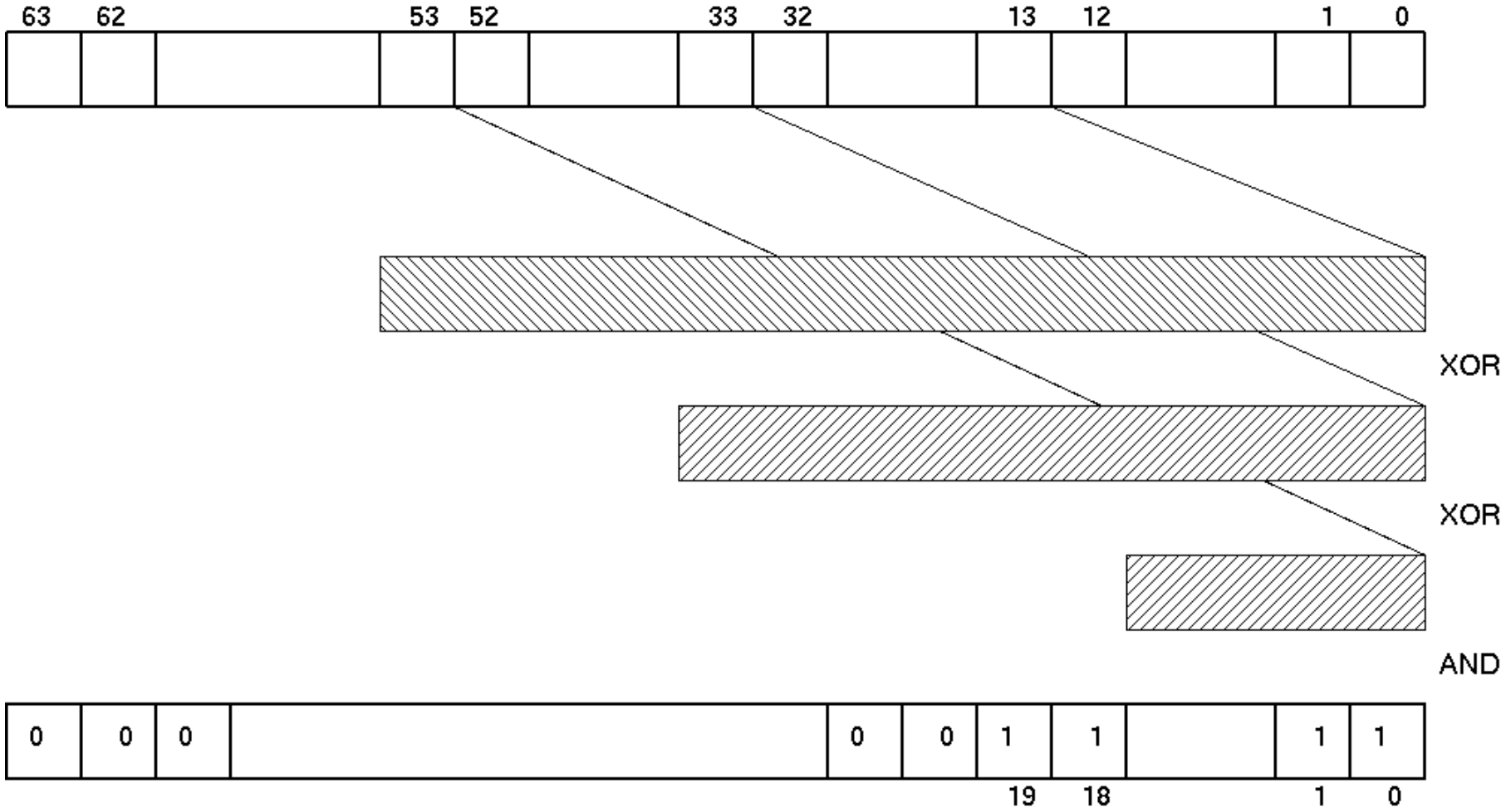


Invertierte Liste

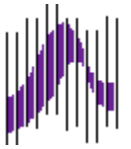
Aus der Seitennummer wird ein Hash-Wert errechnet. Dieser Hash-Wert ist dann der Index in die invertierte Liste.



Beispiel für eine Hash-Funktion



Beispiel-Implementierung in Java



Eine logische 64-Bit Adresse wird auf einen 20-Bit Hash-Wert abgebildet.

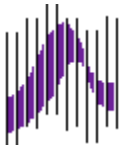
64-Bit Adresse -> 52 Bit Seiten-Adresse:

```
adr = adr >>> 12;
```

52-Bit Seiten-Adresse -> Hash-Wert:

```
index =   adr ^ (adr >>> 32);
index =  (index ^ (index >>> 20)) & 1048575;
```

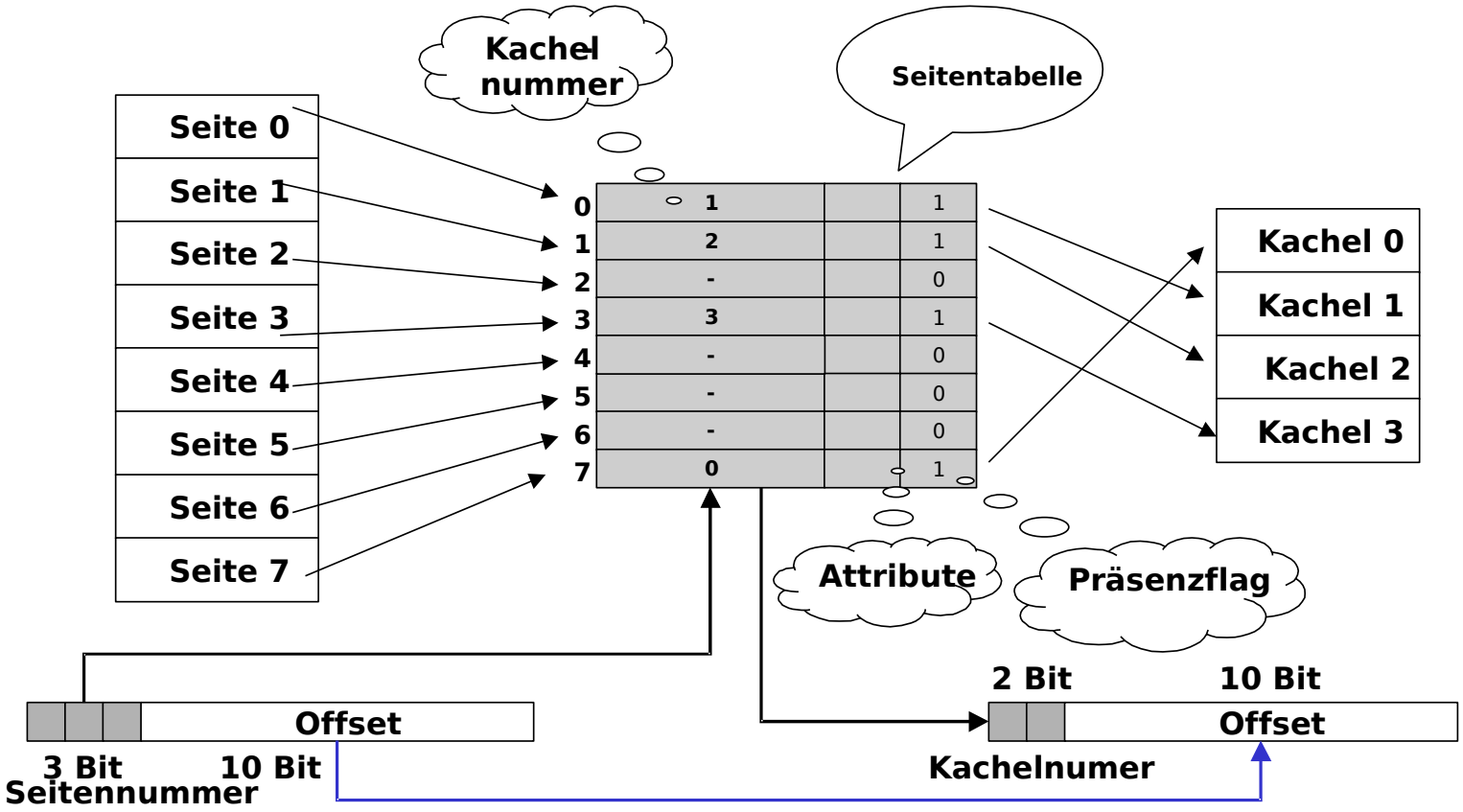
Eintrag in der Seitentabelle



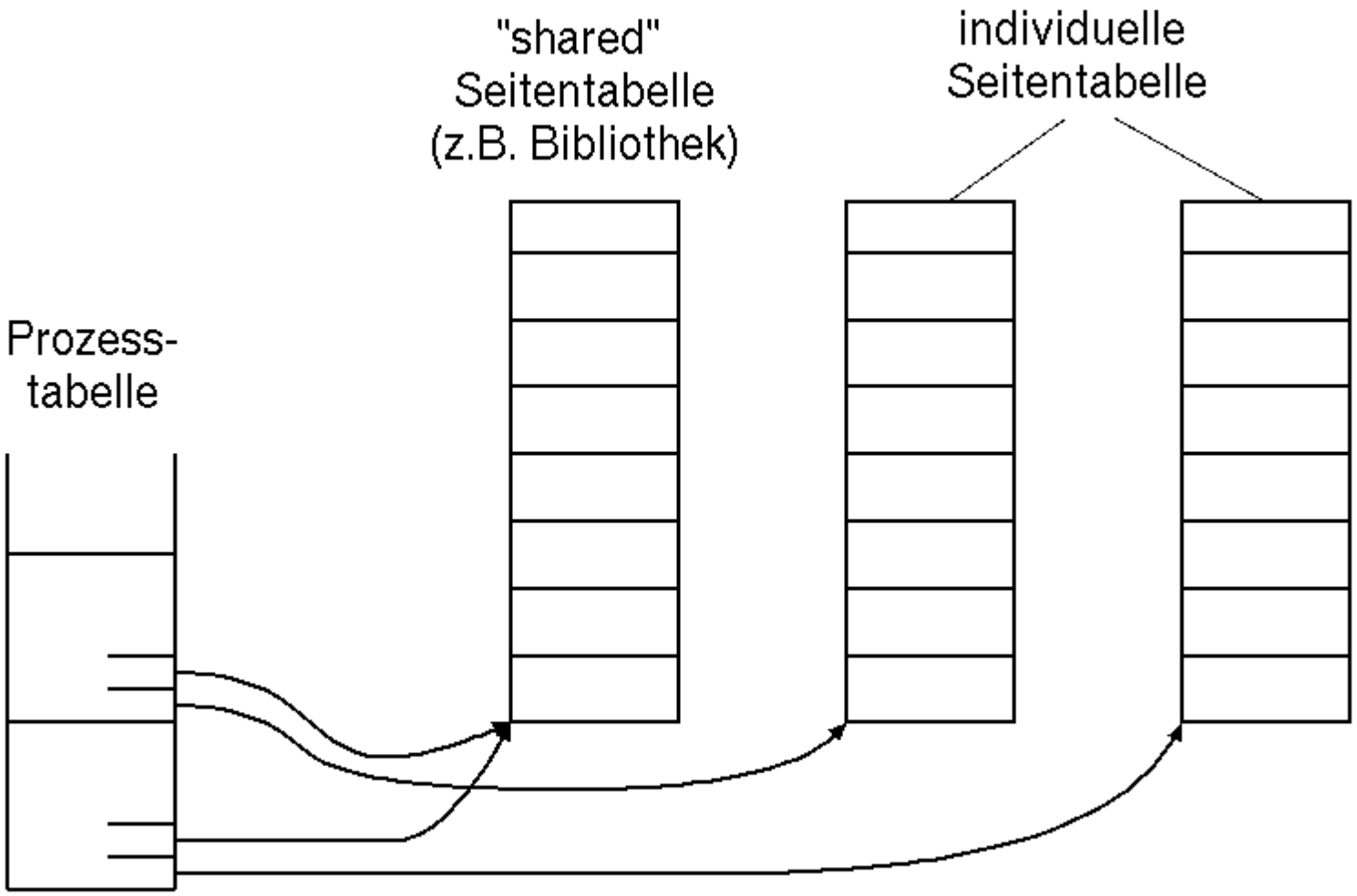
- gültig (present/valid): Seite ist (nicht) im RAM
- Kachelnummer: Physische Anfangsadresse
- dirty (M-Bit): Modifikation seit dem letzen Laden
- reference (R-Bit): Zugriff seit dem letzen Laden
- Schutz: read, write, execute
- auslagerbar: Seite darf (nicht) ausgelagert werden.
- ggf. PID

Paging - ein Beispiel

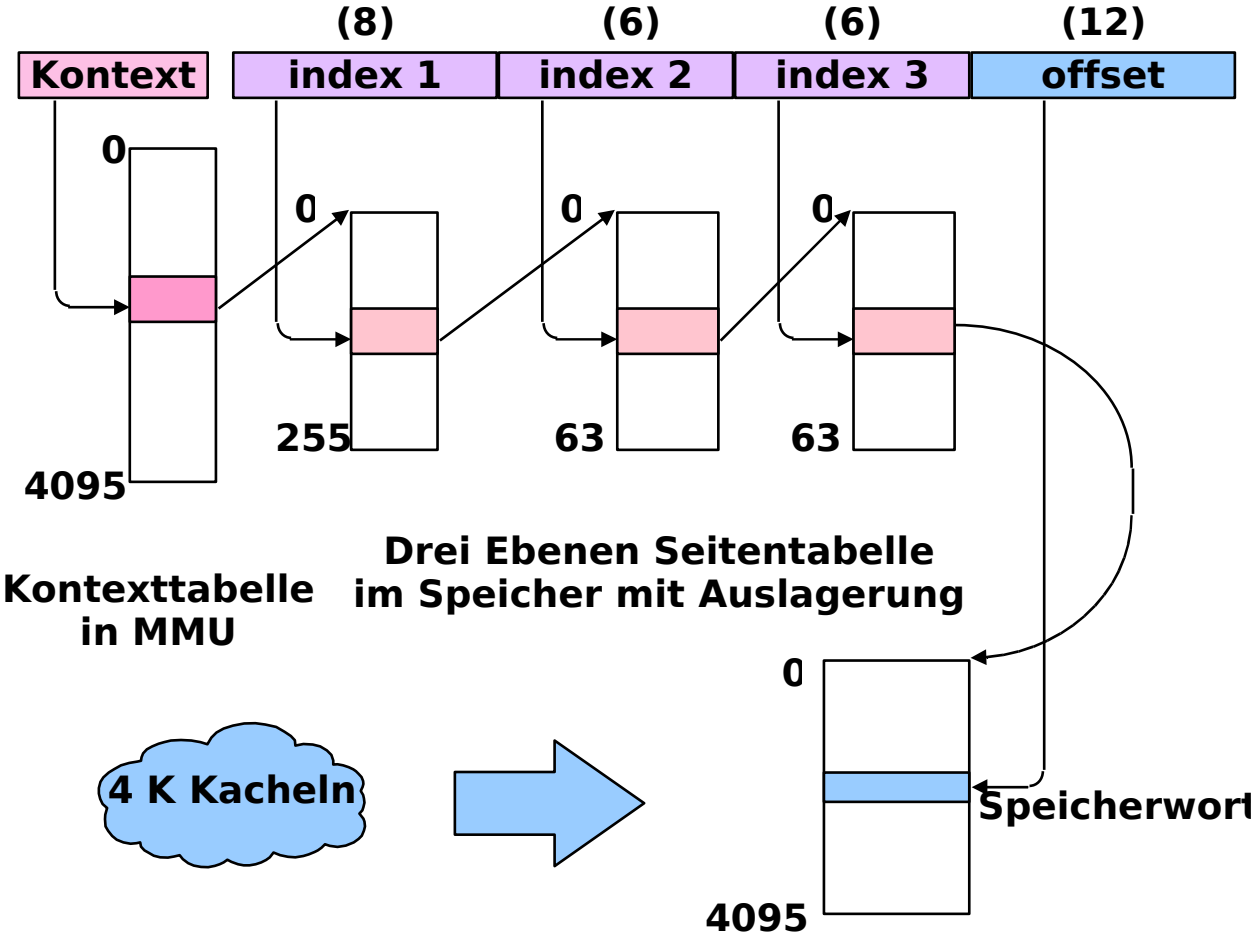
Physikalischer Adreßraum:	4K	➡ 4 Kacheln
Vorhanden:	4K	
Virtueller Adreßraum	8K	➡ 8 Seiten
Seitengröße:	1K	



Gemeinsam genutzte Adressen

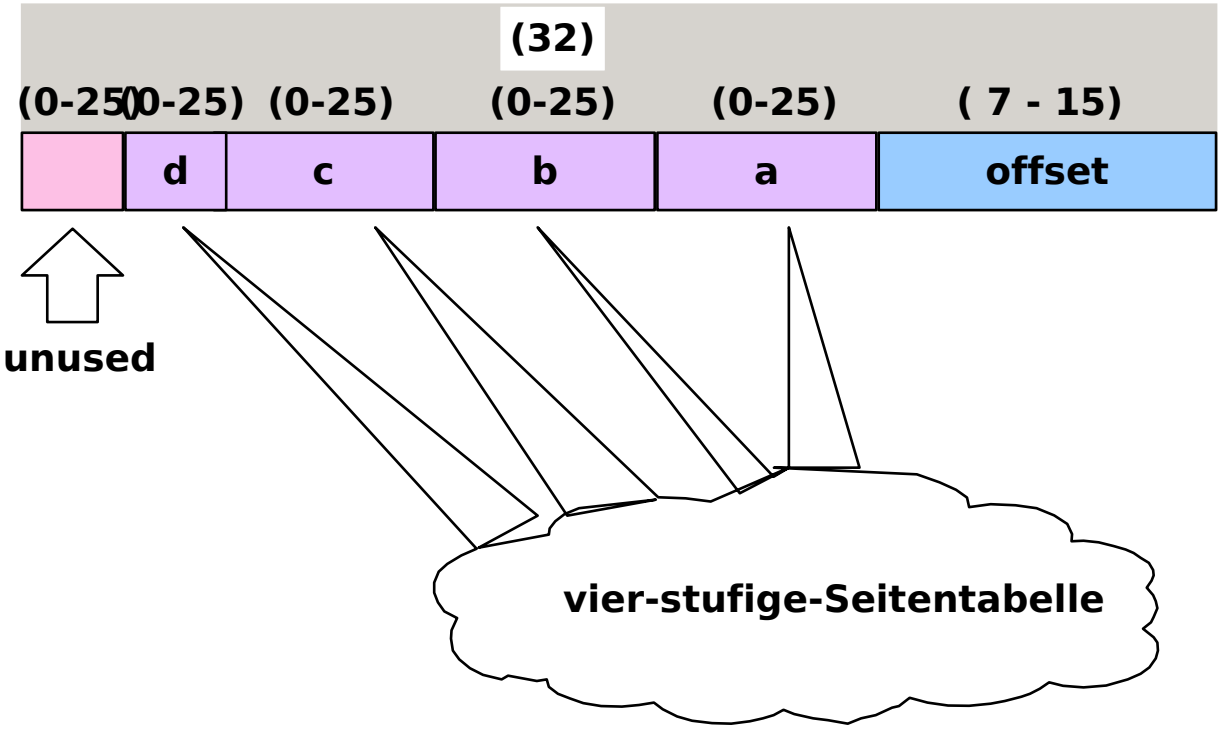


Paging: Drei-Stufen-Paging, die SPARC-Referenz



- Je Prozeß ein Prozeßkontextzeiger Kontextumschaltung durch neuen Kontextzeiger
- Kontextzeiger über die Lebensdauer in Kontext-Tabelle (MMU- Hardware) konstant
- Beschränkung auf 4K Prozesse
- Kontextzeiger und virtuelle Adresse physikalische Adresse
- 4 G virtueller Adressraum je Prozess
- 1 M Seiten je Prozess á 4 K
- 384 Einträge je process page table

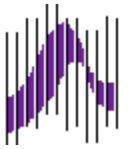
Vier-Stufen-Paging, M68030



- Paging kann durch das Betriebssystem mitbestimmt werden (parametrierbare Hardware- MMU
- 0 - 4 Stufen und Anzahl der Adressbits, die eine Stufe ausmachen sind SW-seitig einstellbar (Index - Breite) translation control register werden durch BS definiert
- BS legt die Interpretation der virtuellen Adresse fest
- virtueller Adressraum kann eingestellt werden ($< 2^{32}$)
- Seitengröße kann eingestellt werden (265 bytes bis 32 K)
- Einstellung wieviel Bit der Adressraum wirklich hat

- Die Eigenschaften des Paging werden ausschließlich durch das Betriebssystem bestimmt
- Kein bekanntes Betriebssystem nutzt die Flexibilität ernsthaft aus
- Prozess-spezifische Definitionen des virtuellen Adressraums denkbar und eventuell sinnvoll

Translation Lookaside Buffer (TLB)

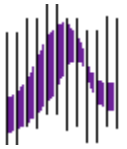


Seitenersetzungstabelle (page table)

- bei mehrstufigen Verfahren liegen die Seitentabellen im Arbeitsspeicher
  - die Adressumrechnung erfordert mehrere Speicherzugriffe
  - Verlangsamung gegenüber der physikalischen Adressierung

- die meisten Programme haben viele Speicherzugriffe in einer kleinen Zahl von Seiten
  - Caching der Seitentabelle in einem **Assoziativspeicher:** Translation Lookaside Buffer (**TLB**)

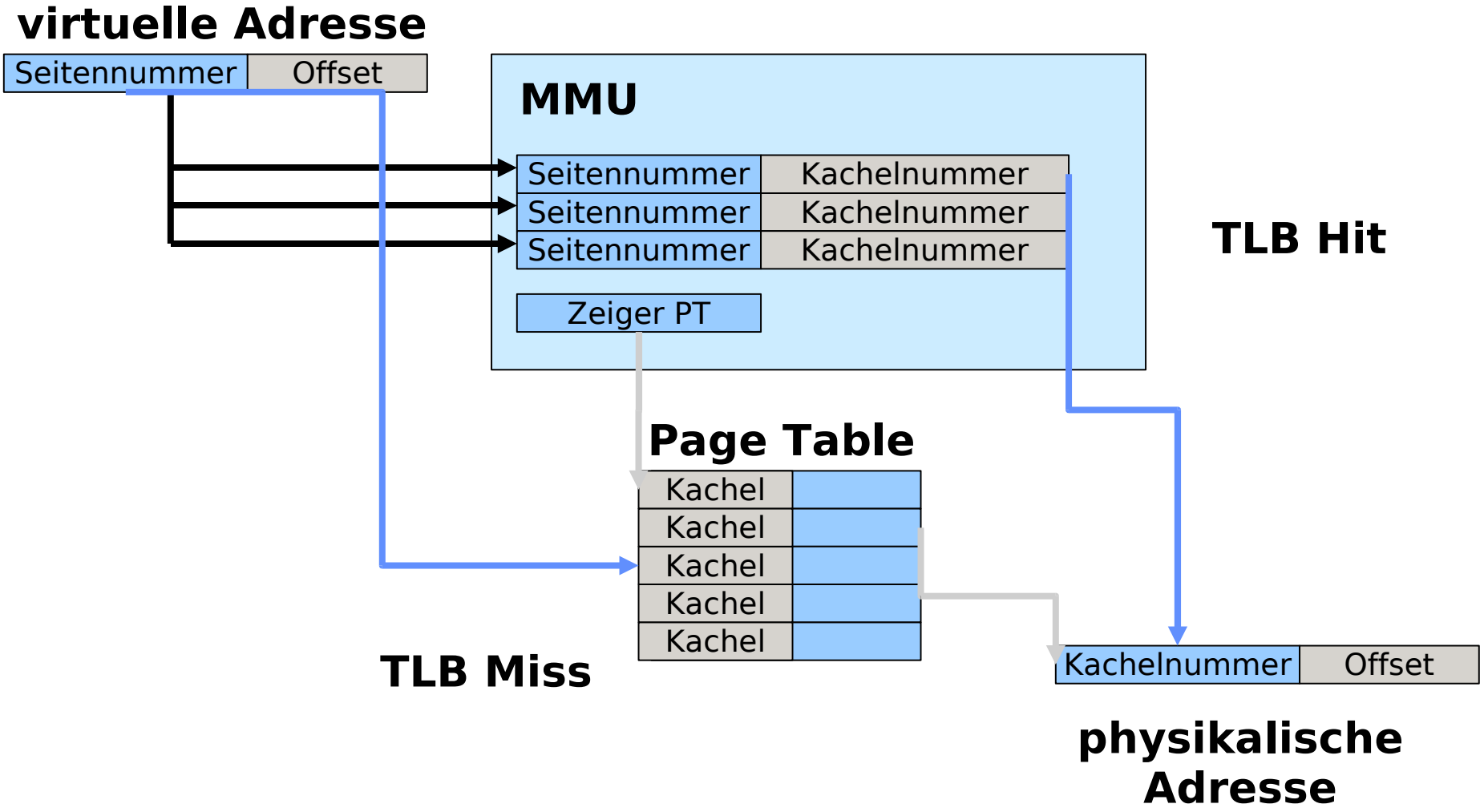
Translation Lookaside Buffer (TLB)



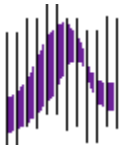
Assoziativspeicher

- Assoziativspeicher ist Teil der MMU (minimal 32 Speicherzellen, bis ca. 128)
- Im Assoziativspeicher werden die im Moment gebräuchlichsten Seiten mit ihrem Seitentabelleneintrag vorgehalten. Neben der Seitennummer enthält ein Speichereintrag auch die Kachelnummer
- Assoziativspeicher realisieren eine parallele Suche d.h. die virtuelle Adresse wird gleichzeitig mit allen Tabellen-einträgen verglichen. Die Suchzeit ist konstant.
- Wird eine virtuelle Seite im TLB gefunden (**TLB Hit**), steht dort die Kachelnummer und die Adressumrechnung kann direkt erfolgen
- War die Suche im TLB ohne Erfolg (**TLB Miss**), muss auf die Seitentabellen im Arbeitsspeicher zurückgegriffen werden

Translation Lookaside Buffer (TLB)



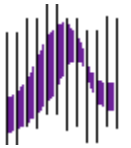
Überblick Speicherzugriff



Zugriff auf eine virtuelle Adresse

- Die MMU sucht Adresse im TLB
  - Erfolg (Gültigkeits-Flag): Zugriff im Hauptspeicher
  - Misserfolg ...
- ... Das Betriebssystem oder die MMU sucht die Adresse in der Seitentabelle
  - Erfolg und Seite im Hauptspeicher: Zugriff, TLB aktualisieren
  - Erfolg und Seite ausgelagert: Seite einlagern Seitentabelle und TLB aktualisieren, Zugriff
  - Misserfolg: „memory fault – core dumped“

mehrstufiges Speichern der Adressen

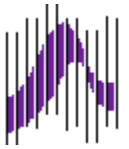


(1) TLB innerhalb der MMU (Assoziativspeicher), parallele Suche durch Hardware.

(2) invertierte Liste (Hash-Tabelle) ohne Kollisionsbehandlung;
enthält mehr Adressen als der TLB aber nicht alle.

(3) Seiten-Tabellen ggf. mehrstufig (Digitalbaum), enthält alle vom Prozess verwendeten Seiten-Adressen.

Größe der Speicher-Seiten



Größere Speicher-Seiten (z.B. 4MB)

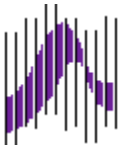
Vorteile:

- Größere Trefferwahrscheinlichkeit im TLB
- Seitentabelle benötigt weniger Einträge

Nachteile:

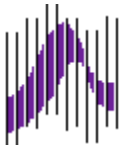
- Höherer Verschnitt bei teilweise genutzte Seiten
- Größerer Aufwand beim ersten Laden einer Seite
- Größerer Aufwand beim Schreiben einer Seite

gängige Seitengrößen



- Pentium, Athlon: 4K oder 4M
- UltraSPARC: 8K, 64K, 512K, 4M
- PowerPC: 4K . . . 4M?

Verdrängungsstrategien (1)

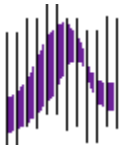


Eine Seite muss aus dem Hauptspeicher entfernt werden: welche?

- Die Seite, die am längsten ausgelagert bleiben kann (optimale Strategie).
- Die Seite, von der man hofft, dass sie am längsten ausgelagert bleiben kann.

Die optimale Strategie kann i. Allg. nur nachträglich ermittelt werden. Sie dient als Vergleichsmaßstab für andere Strategien.

Verdrängungsstrategien (2)

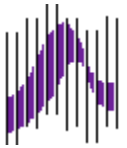


Annahme: Eine Seite, die in letzter Zeit häufig angesprochen wurde, wird voraussichtlich bald wieder angesprochen.

- FIFO
- LRU (Least Recently Used)
- NRU (Not Recently Used)
- Aging
- Second Chance/Clock, WS-Second Chance

Alle Strategien außer FIFO versuchen, die obige Annahme zu nutzen.

Least Recently Used

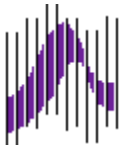


Die Seite, die am längsten nicht mehr referenziert wurde, wird ausgelagert.

Mögliche Implementierung:

- Prozess hält Liste aller Seiten.
- Eine referenzierte Seite kommt jeweils an den Anfang der Liste.
- Die letzte Seite der Liste wird ausgelagert.

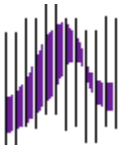
Not Recently Used



Eine Seite, die eine gewisse Zeitspanne nicht benutzt wurde, wird ausgelagert.

- Jede Seite erhält ein Markierungs-Bit (R-Bit).
- Bei jedem Zugriff wird das R-Bit der Seite gesetzt.
- Regelmäßig werden alle R-Bits zurückgesetzt.
- Eine Seite mit nicht gesetztem R-Bit wird ausgelagert.

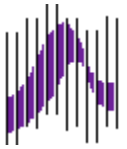
Aging



Eine „gealterte“ Seite wird ausgelagert.

- Jede Seite erhält einen Zähler (Alter).
- Bei jedem Zugriff wird das höchstwertigste Bit des Zählers gesetzt.
- Regelmäßig werden alle Zählerstände halbiert.
- Die Seite mit dem niedrigstem Zählerstand wird ausgelagert.

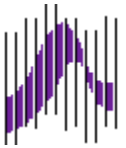
Das „Working Set“



Die meisten Prozesse verbringen einen großen Teil ihrer Zeit in einem kleinen Teil ihres Speichers: dem sog. „working set“.

- Das „working set“ kann sich von Zeit zu Zeit ändern.
- Das „working set“ eines Prozess kann nur mit hohem Aufwand ständig erfasst werden.
- Die „working sets“ aller aktiven Prozesse sollten in den Hauptspeicher passen. Ansonsten müssen zu viele Seiten ein- und ausgelagert werden.

Speicherverwaltung in Linux



Segmentierung

Der virtuelle Adreßraum wird in 4 Segmente eingeteilt (i.a. nicht zur Adressrechnung):

- Text-Segment (read-only, shared)
- Statische Daten (initialisiert, nicht initialisiert (BSS))
- Dynamische Daten, Heap von unten nach oben
- Dynamische Daten, Stack von oben nach unten

Swapping

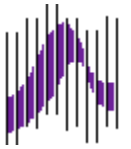
Hintergrundprozeß Swapper
Wird aktiviert zum Auslagern von Prozessen, wenn die Page Fault Rate zu hoch ist, oder kein Speicherplatz mehr vorhanden ist. Wird Periodisch aktiviert um nachzusehen, ob Prozesse wieder eingelagert werden können

Paging

- **Demand Paging WS-Clock Algorithmus**

Paging Daemon, Flush Daemon
Hintergrundprozess wird periodisch (ca. 1 – 4 mal pro Sekunde) aktiviert. Falls weniger als ca. 1/4 des Speichers frei ist, werden im Vorgriff Seiten ausgelagert. Der Flush Daemon schreibt veränderte Seiten auf die Festplatte.

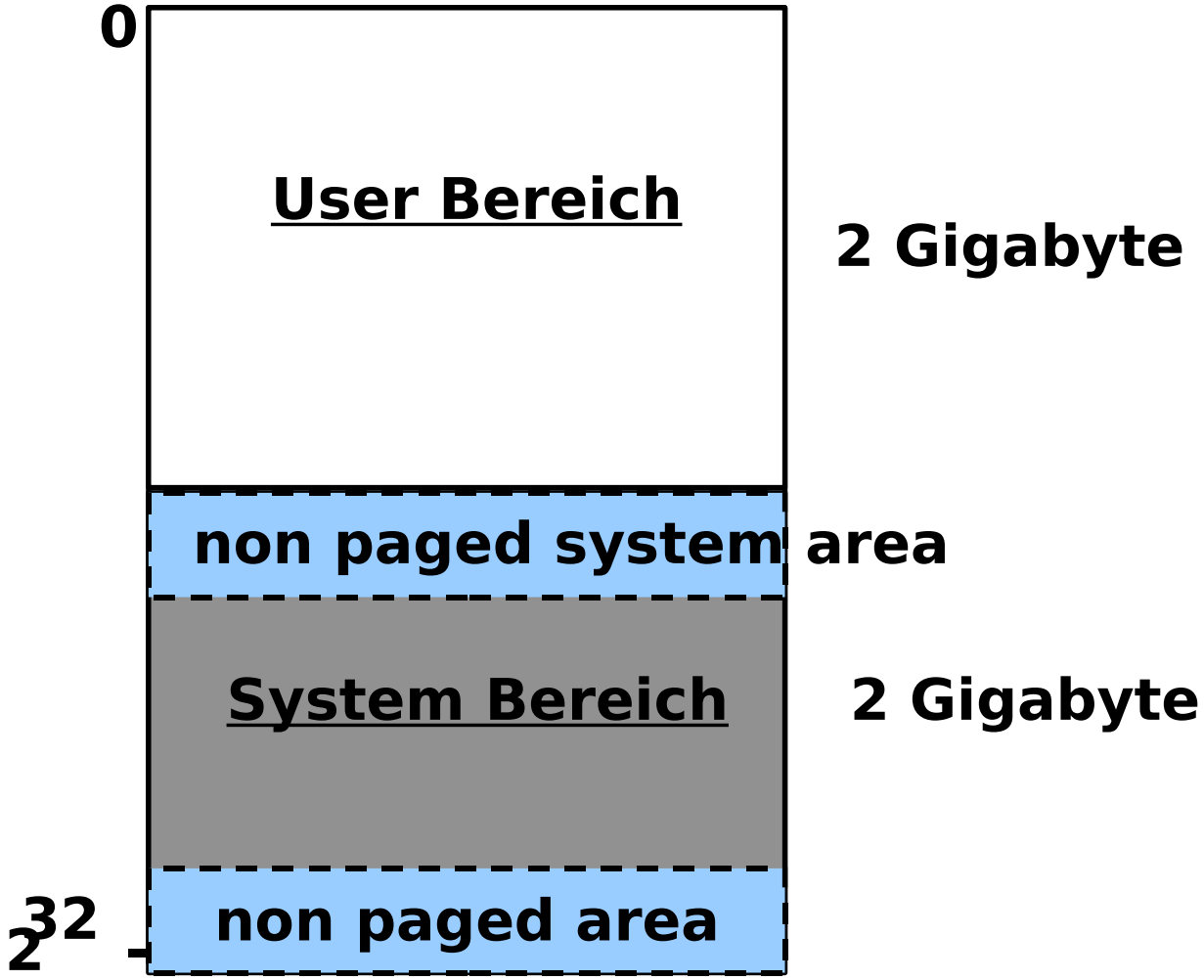
Speicherverwaltung bei Unix fork()



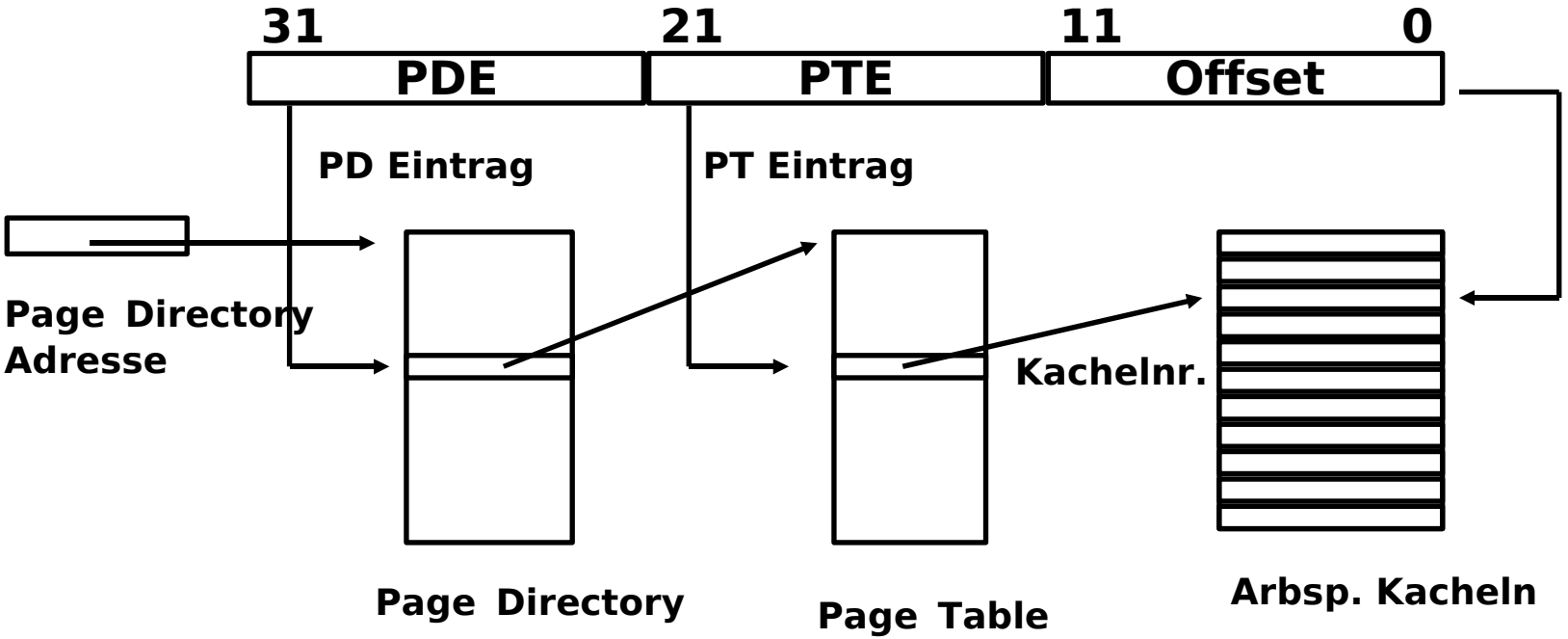
Mechanismus: copy-on-write (lazy copy)

- Kind-Prozess erhält Kopie der Seitentabelle.
- Alle Seiten werde als „read only“ markiert.
- Ein Schreibzugriff führt zu einem Interrupt.
- Die Seite wird kopiert und in beiden Prozessen als „read write“ markiert.

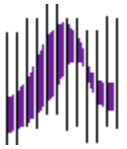
Virtueller Adressraum eines NT Prozesses



Paging in Windows NT (IA32)



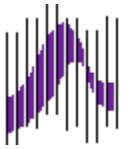
Synchronisation (1)



- Zugriff auf gemeinsame Ressourcen muss synchronisiert werden.
- Bsp: Mehrere Prozesse schreiben Dateien in eine Drucker-Warteschlange.
- So lange ein Prozess schreibt, darf kein anderer Prozess schreiben.

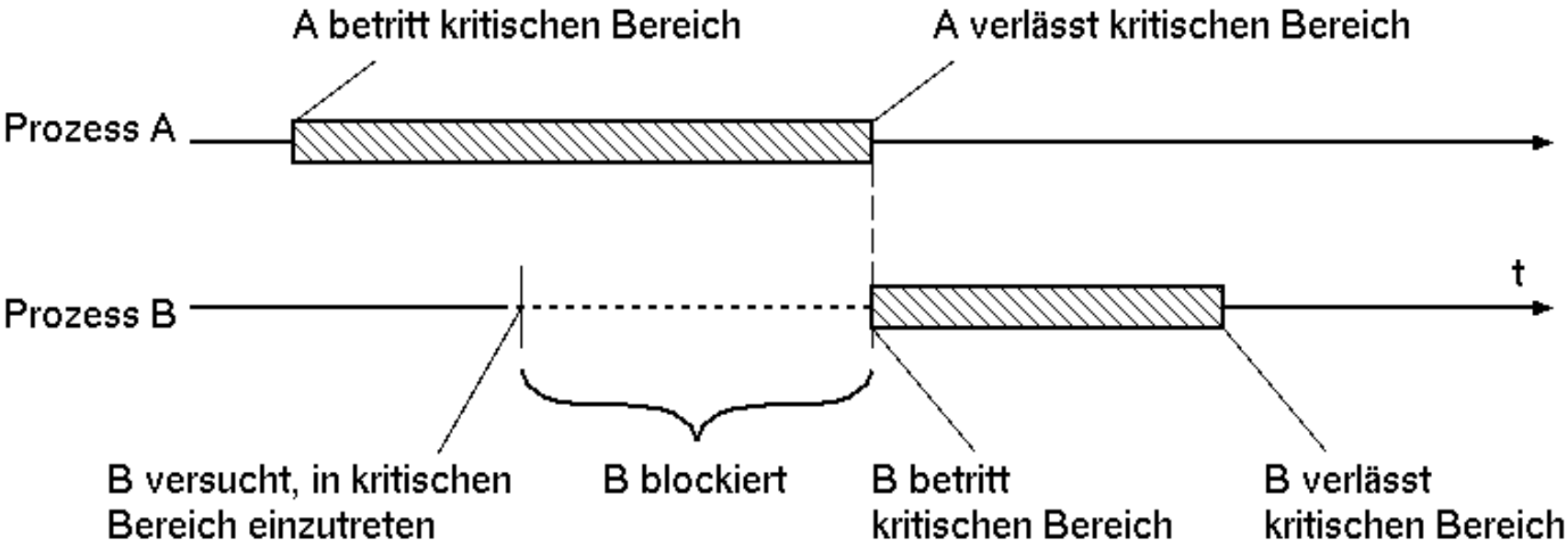
„Kritischer Bereich“: Der Teil eines Programms, der auf gemeinsam genutzte Ressourcen zugreift.

Synchronisation (2)

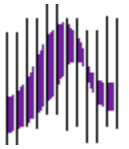


- Keine zwei Prozesse dürfen gleichzeitig in ihrem kritischen Bereichen sein.
- Es darf keine Annahme über die Geschwindigkeit und die Anzahl der CPUs gemacht werden.
- Kein Prozess, der außerhalb seines kritischen Bereichs läuft, darf einen anderen Prozess blockieren.
- Kein Prozess darf dauerhaft blockiert werden.

Zwei Prozesse mit kritischem Bereich



Deadlock

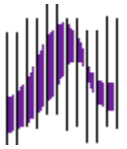


Deadlock durch Wettbewerb um Ressourcen

Notwendige Bedingungen:

- exklusive Nutzung von Ressourcen
- Nachforderung von Ressourcen
- zyklisches Warten möglich

Petrinetze

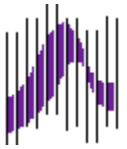


Ablaufmodell zur Beschreibung nebenläufiger Systeme

- bipartiter, gerichteter Graph
- Knoten:
  - Transition, aktive Komponente
  - Stelle, passive Komponente
- Kante: verbindet Transitionen und Stellen

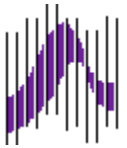
Durch Markierungen und Schaltregeln entsteht ein dynamischer, abwickelbarer Graph, mit dem Systemzustände sichtbar gemacht werden können.

Petrinetze Marken



- Bedingungs-Ereignisnetz:
  Jede Stelle kann genau eine Marke aufnehmen
- Allg. Petri-Netz:
  Jede Stelle kann eine bestimmte Anzahl von Marken aufnehmen
- Die Belegung der Stellen mit Marken definiert den Systemzustand.

Petrinetze Schaltregeln



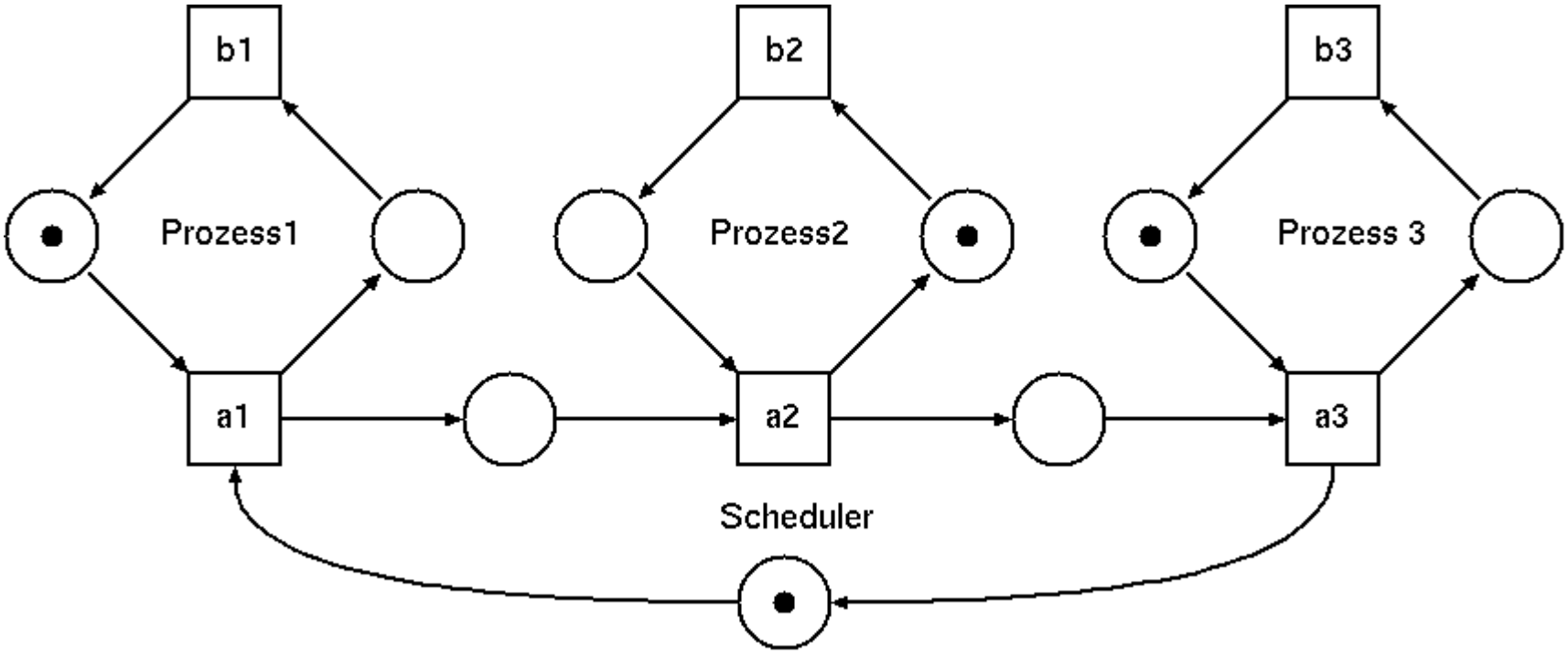
Schaltbereitschaft

- Eine Transition ist schaltbereit, wenn alle Eingangsstellen markiert sind und jede Ausgangsstelle [, die nicht auch Eingangsstelle ist,] eine Marke aufnehmen kann.

- Schalten
  - Eine Transition kann schalten, wenn sie schaltbereit ist.
  - Durch das Schalten wird von jeder Eingangstelle eine Marke entfernt. Zu jeder Ausgangsstelle wird eine Marke hinzugefügt.
  - Das Schalten einer Transaktion ist ein atomarer Vorgang.

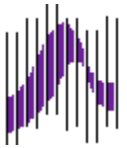
Schalten führt zu einer Zustandsänderung (Marken verschwinden und/oder entstehen).

Beispiel: Milner Scheduler



Die drei Prozesse laufen im gleichen Takt

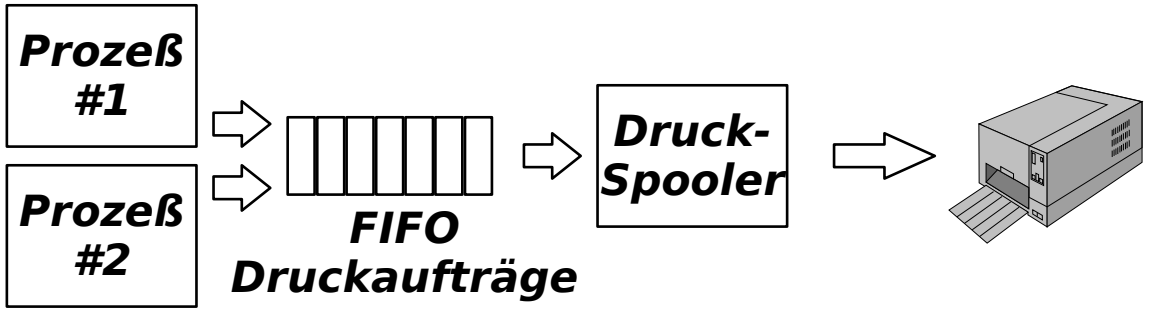
Petrinetz



- Unsicher markiertes Netz
- Synchronisationsgraph
- Zustandsgraph

Kooperation über gemeinsame Daten

Beispiel



Problem

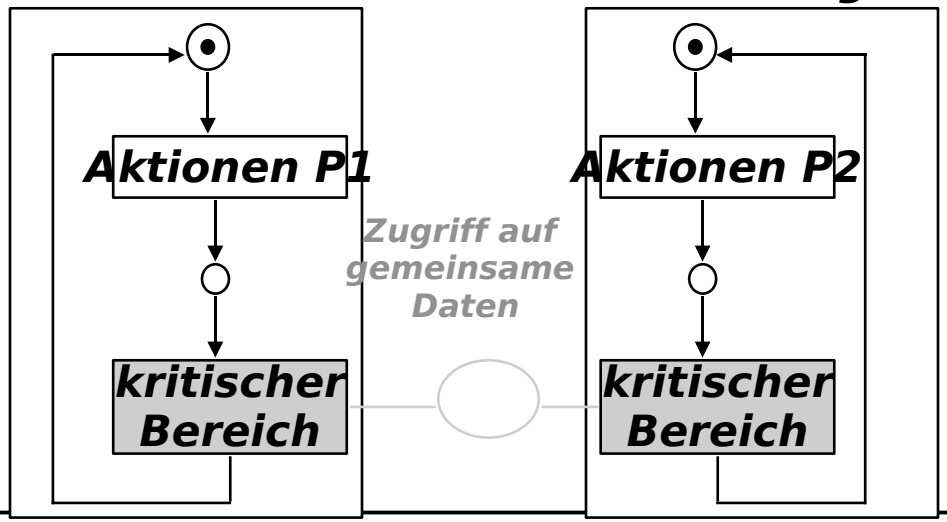
Zeitlicher Wettlauf zwischen Prozeß 1 und Prozeß 2

Kritischer Bereich

Teil des Prozsesses, in dem ein Zugriff auf gemeinsame Datenbereiche erfolgt

Ursache

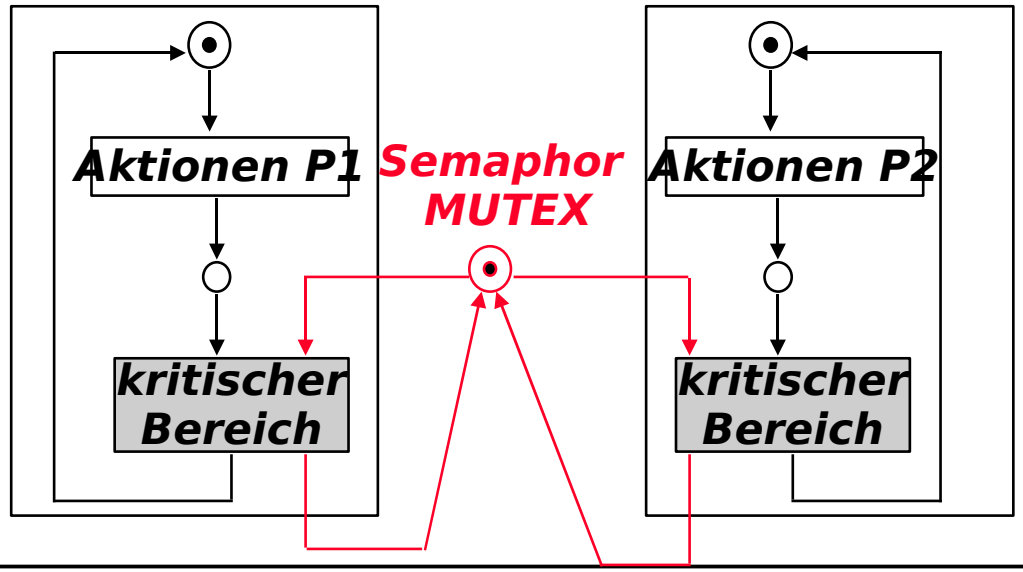
Prozeßwechsel innerhalb kritischem Bereich.



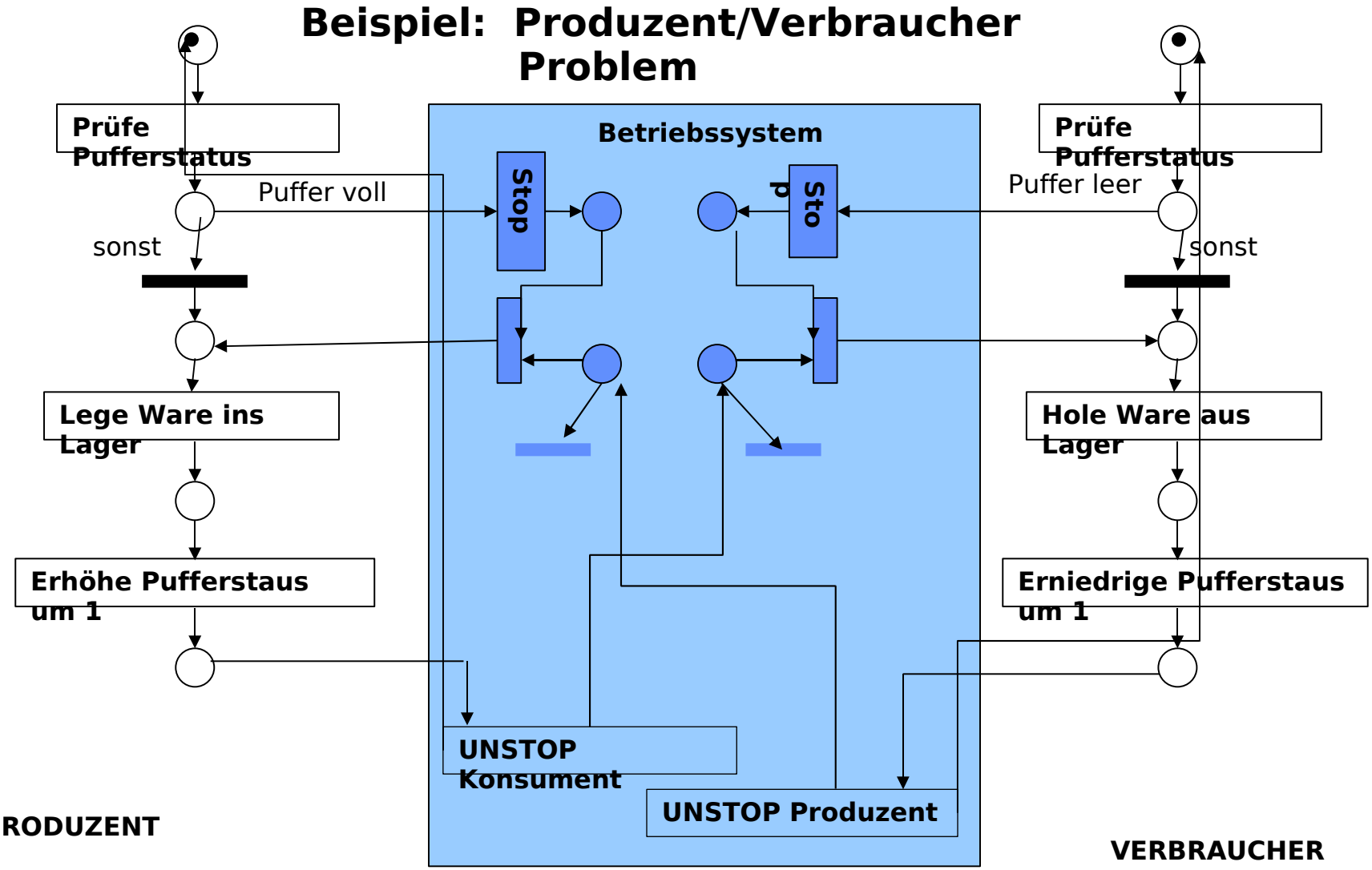
Kooperation über gemeinsame Daten

Wechselseitiger Ausschluß (mutual exclusion)

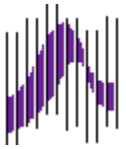
Binäre Semaphore



Synchronisation durch passives Warten



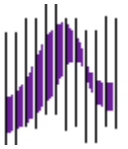
Semaphore



- Abstrakter Datentyp S
- Initialisierung mit ganzer Zahl n >= 0
  - n Element von {0, 1} binäres Semaphor
  - n Element von {0, ... k} allgemeines Semaphor
- Operationen
  - down(S) auch P(S), kann blockieren s.u.
  - up(S) auch V(S)

Ein Prozess ruft beim Betreten eines kritischen Bereichs down(S) auf, beim Verlassen up(S).
Zu jedem Zeitpunkt gilt:
Anzahl (down(S)) <= Anzahl(up(S)) + n

Implementierung - unsicher



```
void down(S){
    while (S == 0){
       wait();
    }
    S = S - 1;
}
```

```
void up(S){
    S = S + 1;
    notify();
}
```

Ein wartender Prozess wird geweckt

Wird ein Prozess in der Funktion down(S) zwischen dem Vergleich und dem Dekrementieren von S unterbrochen, so kann ein zweiter Prozess fälschlicherweise den kritischen Bereich betreten.

Implementierung - sicher

TSL: nichtunterbrechbare Anweisung,
holt *und* setzt einen Wert.

MUTEX: 0 ≡ frei, 1 ≡ belegt

```
mutex_lock:
      TSL REGISTER, MUTEX
      CMP REGISTER, #0
      JZE ok
      CALL thread_yield
      JMP mutex_lock
ok:   RET

mutex_unlock:
      MOVE MUTEX, #0
      RET
```

kopiere MUTEX in Register und setze MUTEX = 1

Vergleich auf 0

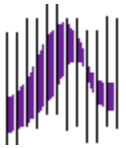
jump on zero

MUTEX belegt - warten

nächster Versuch

MUTEX frei - weiterlaufen

Producer - Consumer

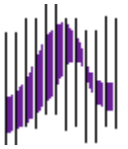


```
typedef int semaphore;
semaphore mutex = 1, empty = 5, full = 0;
```

```
void producer (void){
   int item;
   while(true){
      item = produce();
      down(&empty);
      down(&mutex);
      insertItem(item);
      up(&mutex);
      up(&full);
   }
}
```

```
void consumer (void){
   int item;
   while(true){
      down(&full);
      down(&mutex);
      item = getItem();
      up(&mutex);
      up(&empty);
      consume(item);
   }
}
```

Producer – Consumer - Deadlock



```
void producer (void){
   int item;
   while(true){
      item = produce();
      down(&mutex);
      down(&empty);
      insertItem(item);
      up(&mutex);
      up(&full);
   }
}
```

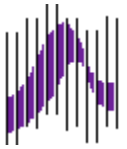
Anweisungen sind vertauscht

Wenn ein Producer den Puffer füllt, blockiert er in down(&empty).

Ein Consumer blockiert aber in down(&mutex).

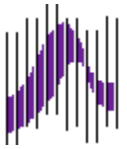
Kein Prozess kann weiterlaufen.

Semaphore



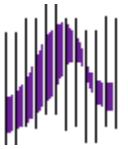
- Semaphore können grundsätzlich alle Synchronisationsaufgaben lösen.
- Semaphore sind schwierig in der Handhabung und Fehleranfällig.

Semaphore in POSIX/UNIX/Linux



- Semaphore werden zu Gruppen zusammengefasst.
- Eine atomaren Aktion manipuliert alle Semaphore der Gruppe: up() bzw. down() ggf. gemischt.
- Undo-Operationen können vom System mitgeführt werden. Beendet sich ein Prozess, so wird die Undo-Operation ausgeführt.

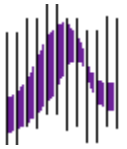
Monitore



Synchronisationsmethode nach Hoare (1974) und Hansen (1975), realisiert mutual Exclusion

- Ein Monitor besteht aus einer Menge von Prozeduren, Variablen und Datenstrukturen, die in einem Modul bzw. in einer Klasse zusammengefasst sind
- Ein Programm/Prozess kann nur über die Monitorprozeduren auf die Daten im Monitor zugreifen
- Nur genau ein Prozess kann in einem Monitor aktiv sein

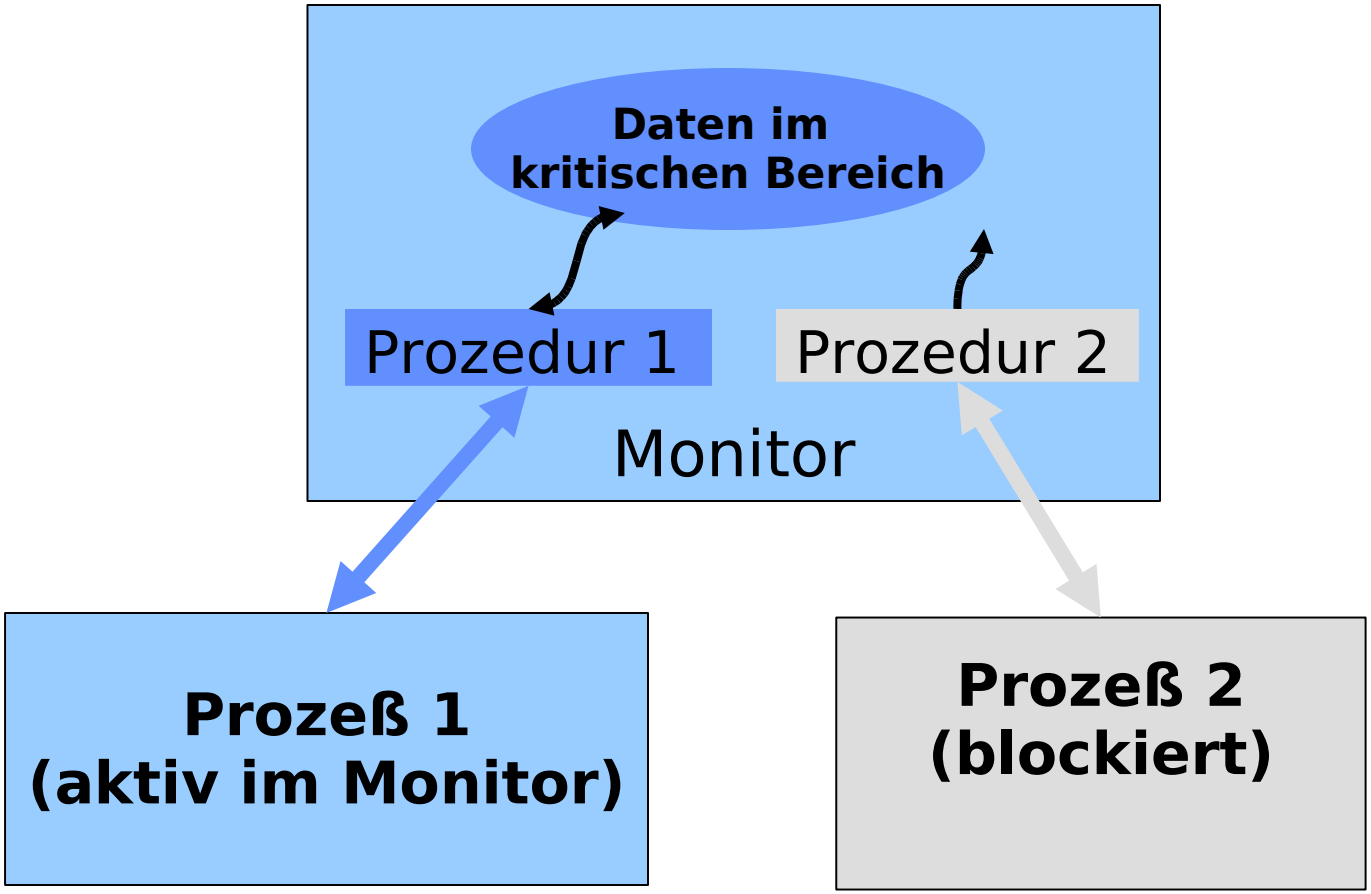
Monitore



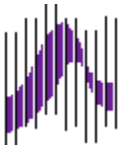
Monitore sind Konstrukte der Programmiersprache

- Der Compiler realisiert Monitore:
  - Ruft ein Prozeß P2 eine Monitorprozedur auf, während gerade ein anderer Prozess P1 den Monitor (aktiv) benutzt, so wird P2 blockiert, bis der Prozess P1 den Monitor wieder verlässt.
  - Ist kein anderer Prozess im Monitor aktiv, so erhält P2 das Recht die Monitorprozedur zu benutzen und sperrt damit für andere Prozesse den Zugriff auf die Monitordaten.
- Der Compiler muss für einen Monitor den wechselseitigen Ausschluss umsetzen.
- Falls vorhanden verwendet der Compiler dazu binäre Semaphore

Monitore

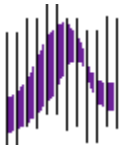

Daten im
kritischen Bereich
Prozedur 1
Prozedur 2
Monitor
Prozeß 1
(aktiv im Monitor)
Prozeß 2
(blockiert)

Monitor
Konstrukt der Programmiersprache



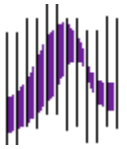
- Vorteil:
  - Der Programmierer muss sich um mutual exclusion nicht selbst kümmern.
  - Dies regelt der Compiler. Daher können auch keine Fehler durch zeitliche Wettläufe erfolgen. Der Programmierer muss „nur“ alle kritischen Bereiche in Monitore legen.
- Nachteil:
  - Monitore sind Elemente einer Programmiersprache.
    - C und C++ ermöglichen keine Monitore.
    - JAVA unterstützt das Monitorkonzept

Monitore - Synchronisation



- In den meisten Fällen muss neben mutual exclusion zusätzlich ein Synchronisationsprotokoll zwischen den Prozessen implementiert werden, das einen Prozess blockiert wenn die globale Variable bestimmte Bedingungen nicht erfüllt, und den Prozess wieder deblockiert wenn die Bedingungen wieder hergestellt sind.
- Beispiel Druckspooler:
  - Der Zugriff auf die Druckwarteschlange (Einschreiben von Druckaufträgen, herausnehmen von Druckaufträgen) muss mittels mutual exclusion sicher implementiert werden.
  - Ist kein Auftrag in der Schlange muss der Druckspooler blockieren.
  - Ist die Druckerwarteschlange voll muss der druckende Prozess blockiert werden.

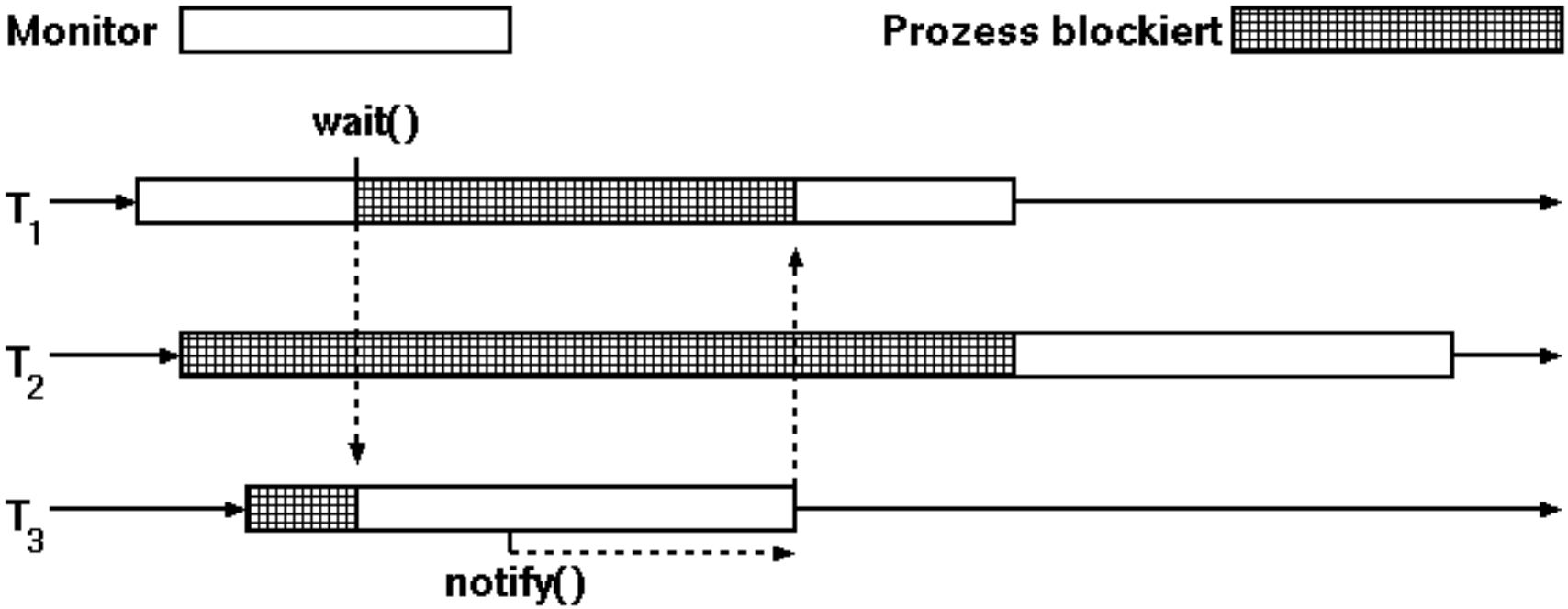
Synchronisation durch Monitore



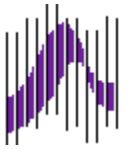
Realisierung in Java: Ein Thread, der sich in einem Monitor befindet, kann die Funktionen wait() und notify() aufrufen.

- Die Funktion wait() bewirkt, dass der aufrufende Thread blockiert wird.
- Ein zweiter Thread kann nun den Monitor betreten.
- Die Funktion notify() bewirkt, dass (irgend) ein anderer Thread, der im gleichen Monitor blockiert ist, deblockiert wird, sobald der aufrufende Thread den Monitor verlässt.

Synchronisation durch Monitore



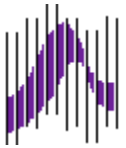
Synchronisation durch Monitore



Blockieren und Deblockieren wird innerhalb der Prozeduren des Monitors entschieden und realisiert.

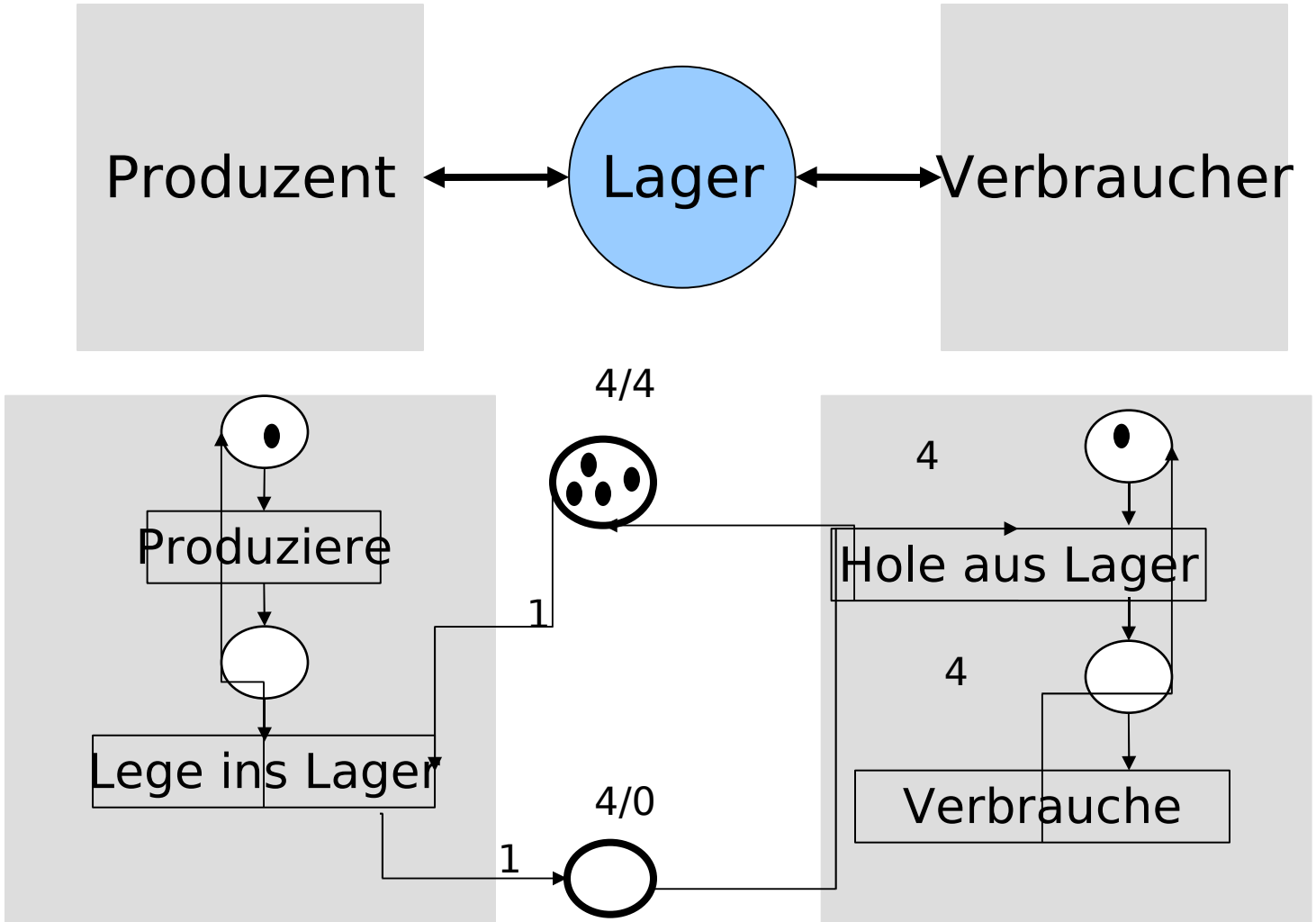
- Warten auf Bedingungsvariablen bzw. Ereignissen
- Setzen von Bedingungsvariablen bzw. Ereignissen

Monitore - Synchronisation

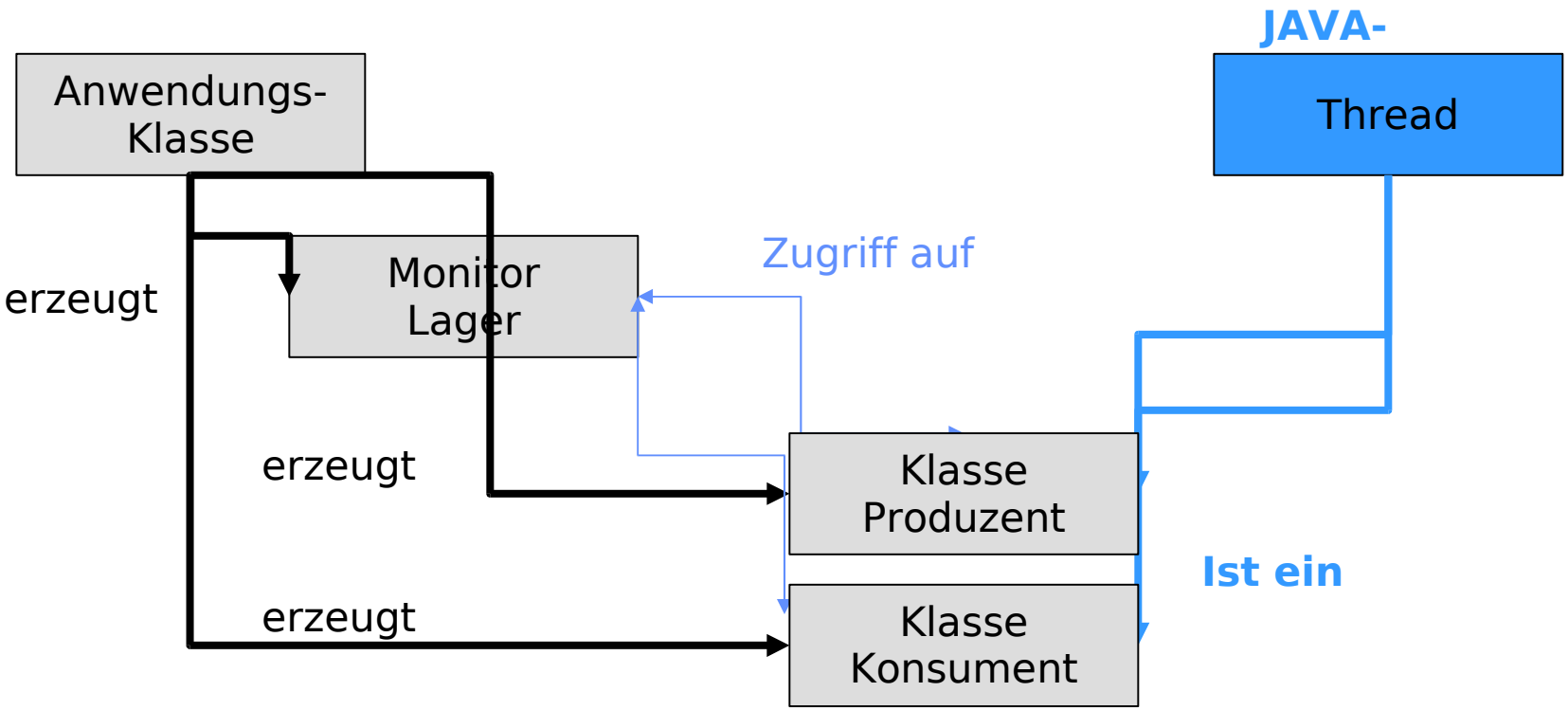


- Wird in einer Monitorprozedur erkannt, dass eine Weiterverarbeitung nicht möglich ist, wird auf das Eintreten einer Bedingung gewartet: **wait(event)**
  - Dies bewirkt dass der **Prozess**, der sich in der Monitorprozedur befindet **blockiert** wird.
  - Darauf wird der Monitor für **einen** auf Zutritt in den Monitor wartenden Prozess freigegeben, der dann eine Monitorprozedur ausführen kann, die die Bedingung für die Weiterverarbeitung eines blockierten Prozesses herstellen kann (**notify bzw. signal(event)**).
- Damit nicht zwei Prozesse gleichzeitig den Monitor benutzen,
  - darf der jetzt deblockierte Prozess erst dann weiterlaufen, wenn der die Deblockierung auslösende Prozess den Monitor wieder verlassen hat.
  - Das Deblockieren eines Prozesses sollte daher die letzte Anweisung in einer Monitorprozedur sein. (In Java wird der Aufruf von notify() erst wirksam, wenn der Thread den Monitor verlässt).

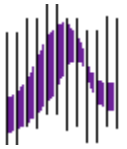
Monitore - JAVA-Beispiel



Monitore - JAVA-Beispiel



POSIX Threads



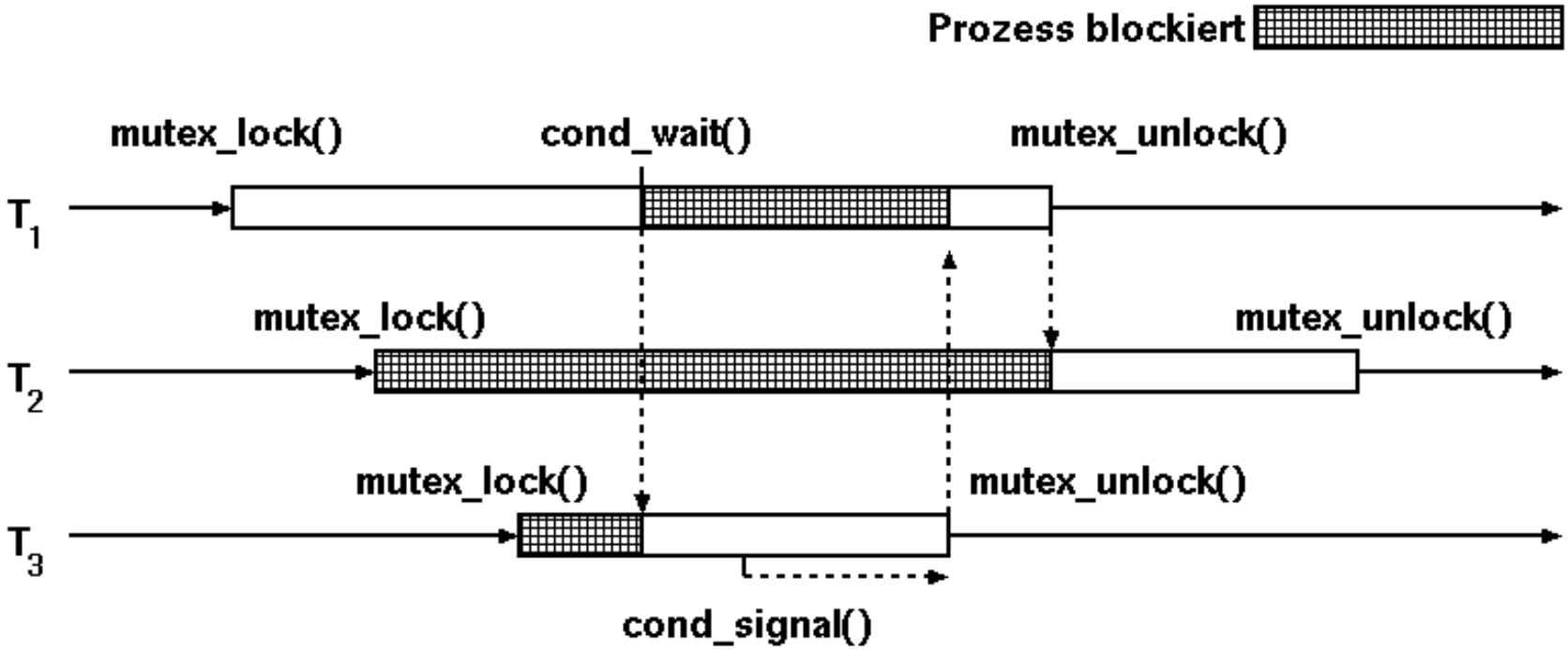
Thread-Blibliothek gemäß POSIX-Standard

Thread-Management
- create
- yield
- exit

Synchronisierung
- join
- Mutex
- Condition Variable

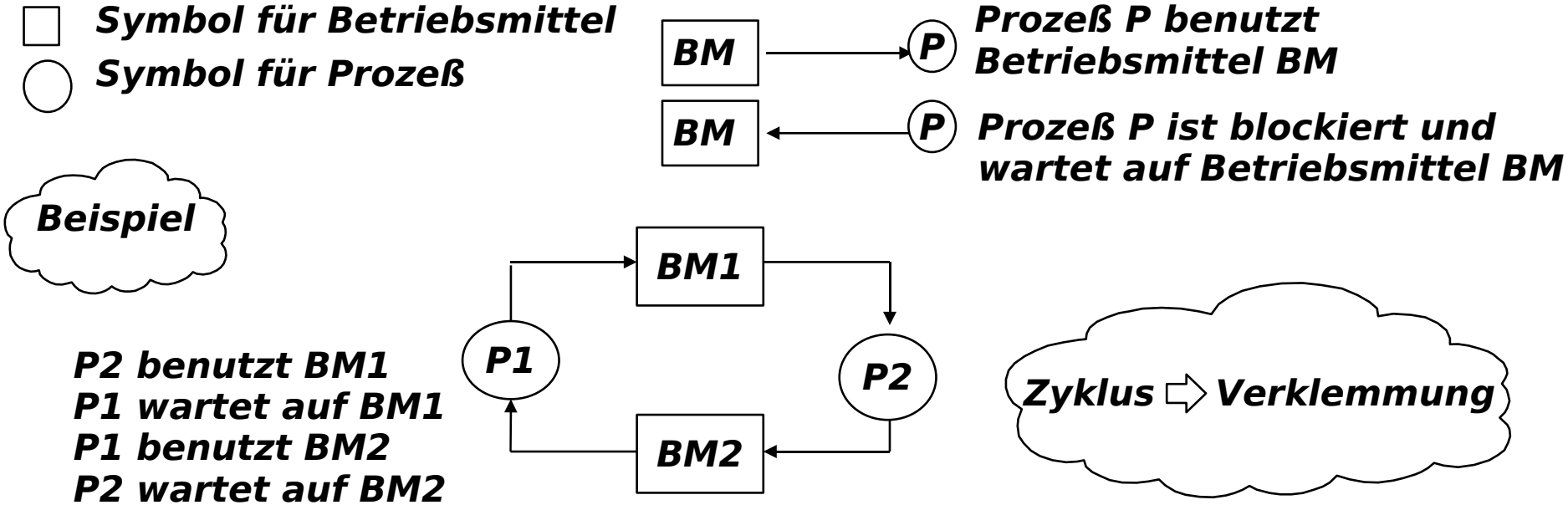
NB.: Diese Funktionen steuern nur Threads eines Prozesses.

POSIX Threads

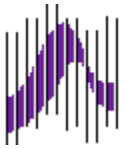


Verklemmungen

- Entdecken und Beheben
  - Protokollieren von Betriebsmittelanforderungen und Betriebsmittelbenutzung durch Aktualisieren des Betriebsmittelgraphen.
  - Prüfen des Graphen auf Zyklen und entfernen eines Prozesses aus dem Zyklus, so daß Zyklus aufgelöst wird.
- Was ist ein Betriebsmittelgraph?

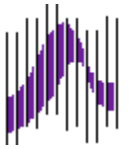


Deadlock - Strategien



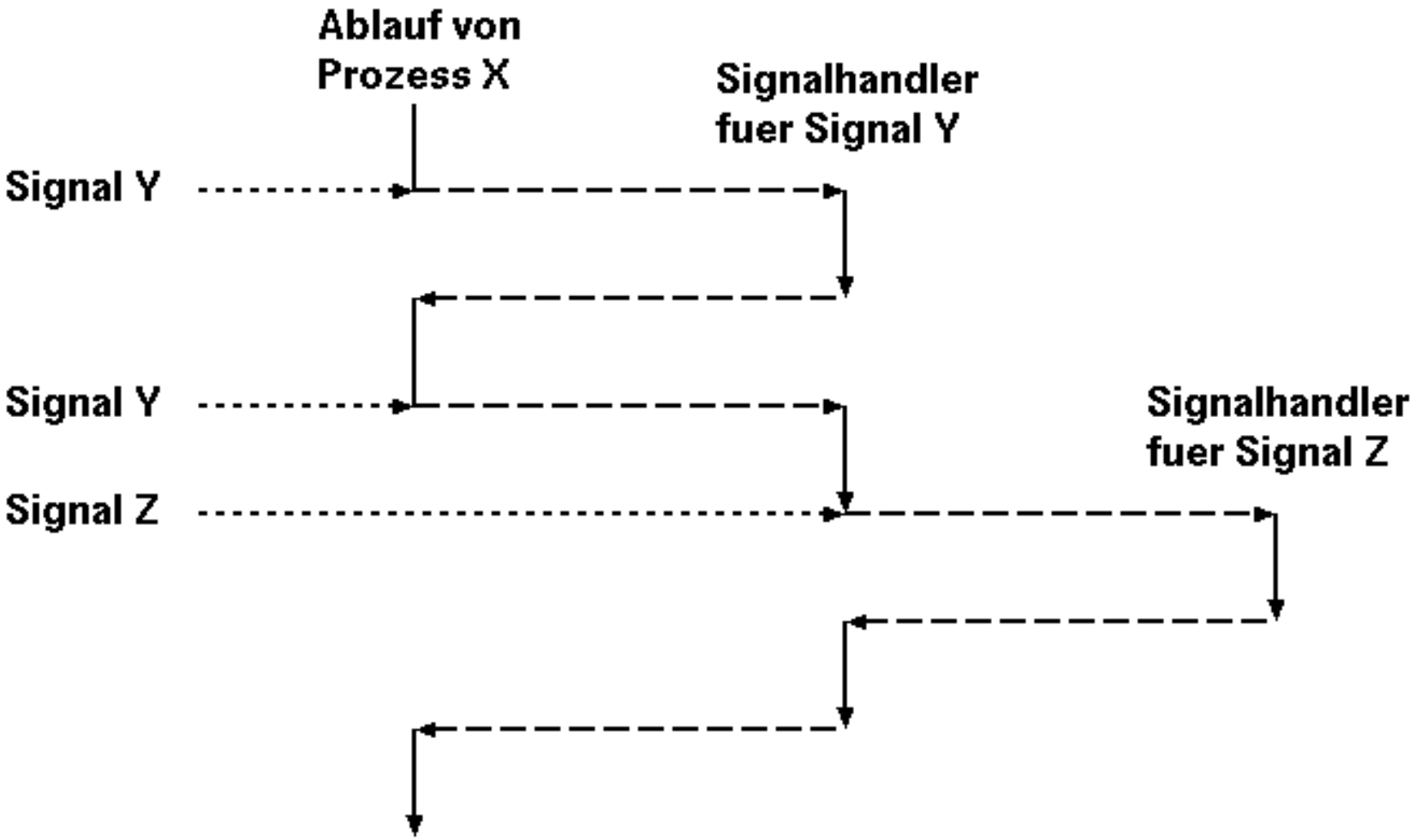
- Vermeidung
  - Vorbeisteuern: Anforderungsketten von vorneherein vermeiden
  - **Banker Algorithmus** (Dijkstra, 1965)
    (Quelle: Tanenbaum, Modern Operating Systems)
    - Jeder Prozess gibt zu Beginn an, welche Betriebsmittel er zur Laufzeit benötigt. In der Praxis nicht anwendbar
    - Variante: Prozess muss alle Ressourcen freigeben bevor er wieder Ressourcen anfordern kann
- Automatisch erkennen und durchbrechen
- Problem ignorieren
  - Wenn eine Verklemmung auftritt, muss ein Administrator (Nutzer) eingreifen

Interprozesskommunikation unter POSIX

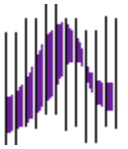


- Signale
  - asynchrone Benachrichtigung
- Pipes
  - FIFO-Puffer-Speicher mit Stream-Interface
- Message-Queues
  - FIFO-Puffer-Speicher mit Zugriffsfunktionen
- Shared Memory
  - gemeinsamer Speicher, Zugriff über Pointer
- Sockets
  - Stream-Interface zu Kommunikationsprotokoll

Signale

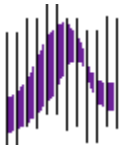

Ablauf von
Prozess X
Signalhandler
fuer Signal Y
Signal Y
Signal Y
Signalhandler
fuer Signal Z
Signal Z

Signale



- Der sendende Prozess muss die PID des Empfängers kennen.
- Der Empfänger muss Signalhandler-Funktionen einrichten
- Die Signalhandler-Funktionen müssen reentrant sein.
- Signale können keine Daten übertragen.
- Ein Prozess kann auf ein Signal warten. Prozesse können sich daher über Signale synchronisieren.

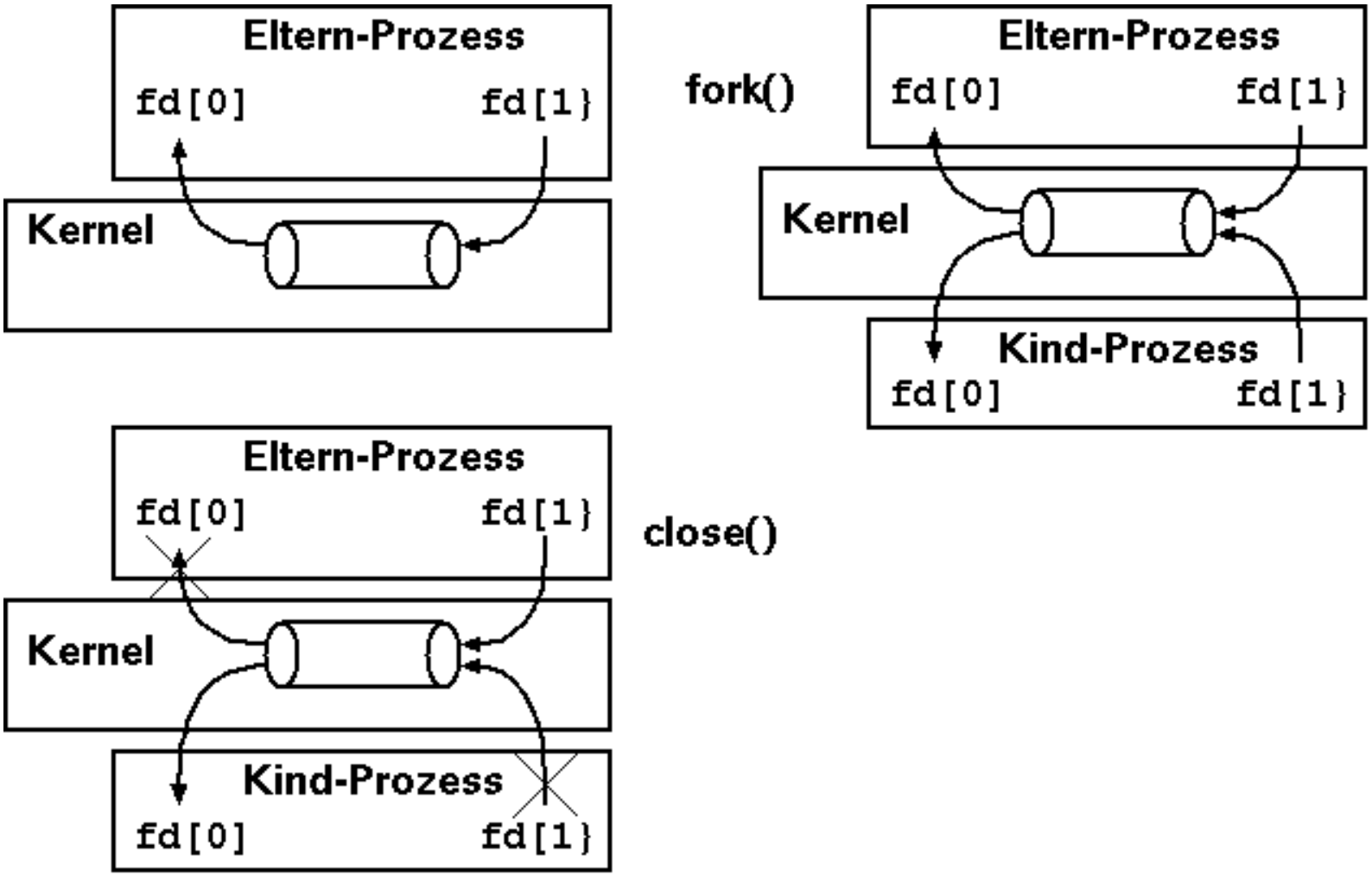
Signale



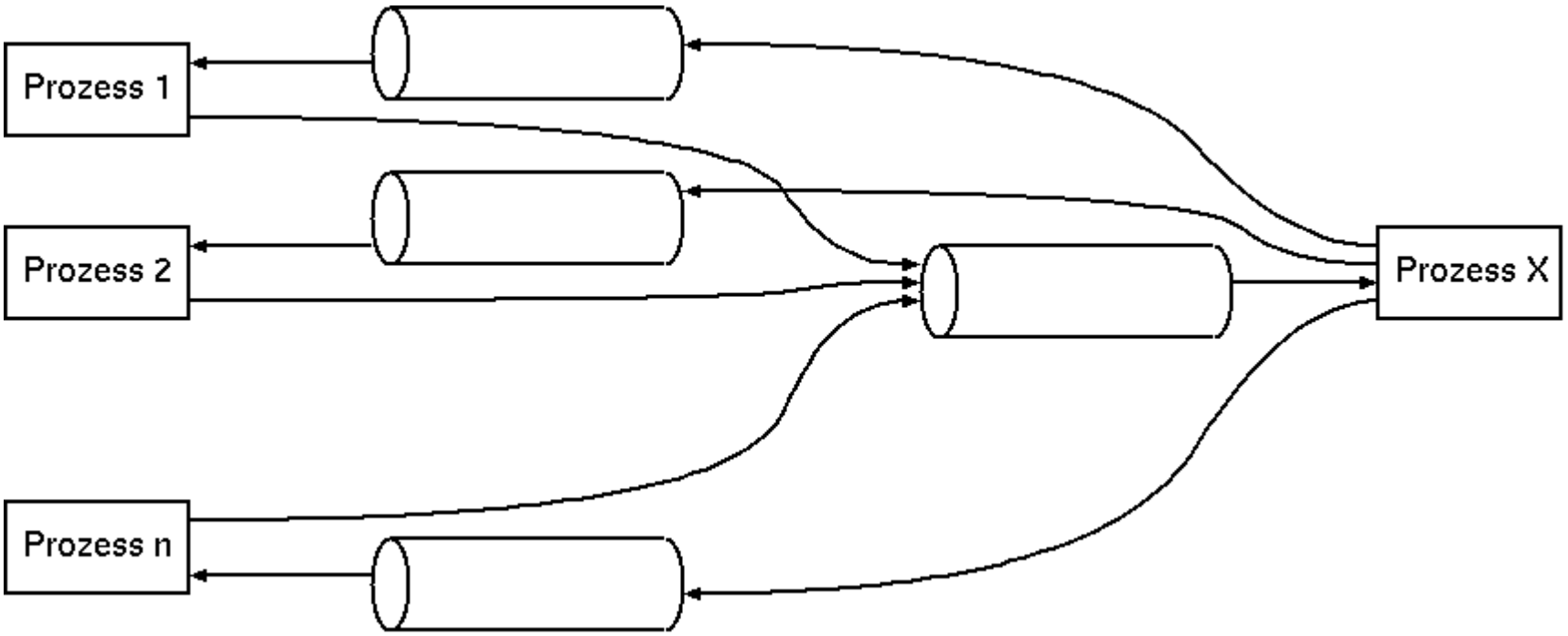
- Der Signalhandler kann eine globale Variable ändern, um „seinen“ Prozess zu informieren.
- Die globale Variable sollte (muss) atomar sein.
- Atomar les- und schreibbare Typen sind:
  - sig_atomic_t (ganzzahliger Datentyp)
  - i.Allg. int und Pointer

Funktionen: signal() aus dem POSIX-Standard, sigaction() unter Unix/Linux. Die Funktion signal() besitzt Schwächen.

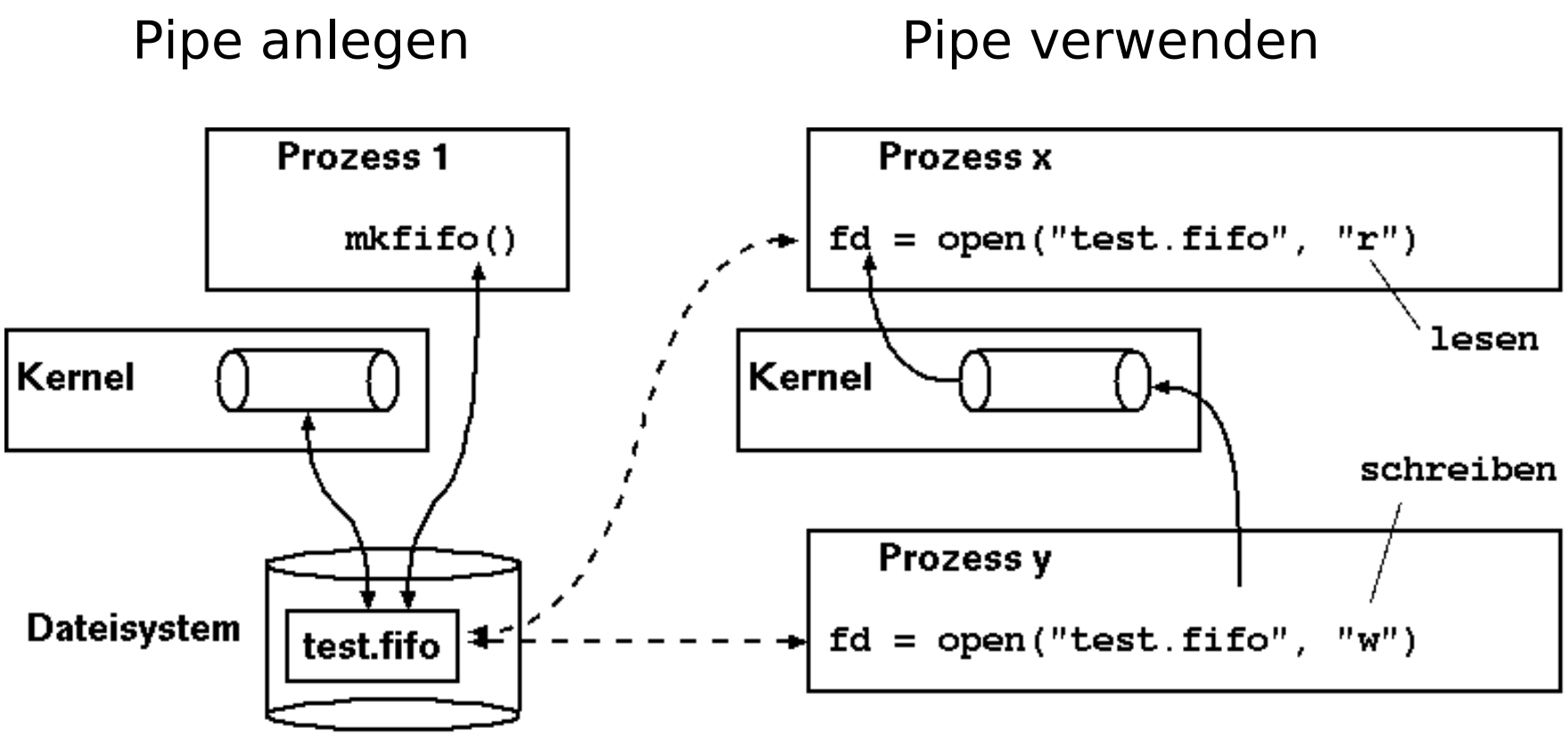
Pipe zwischen Eltern und Kind



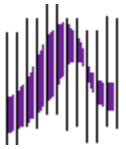
1 zu n Beziehung



Named Pipe / FIFO



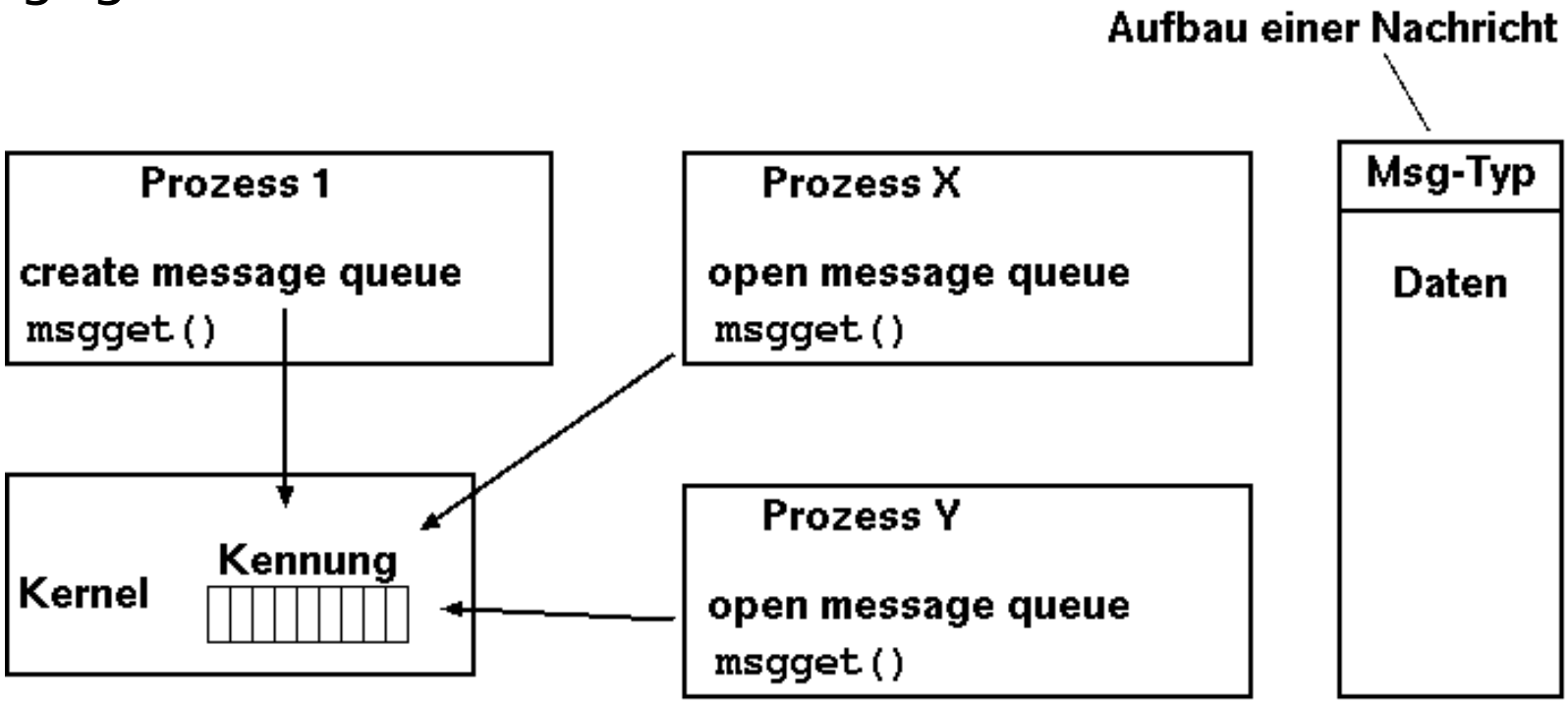
Zugriff auf Pipes



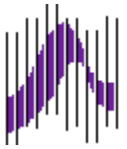
- Kernel synchronisiert „kurze“ Lese-und Schreiboperationen
- Maximale Länge: PIPE_BUF
- Längere Operationen: Synchronisation durch Anwendung
- Lesen entfernt Daten aus der Pipe
- Schreiben blockiert bei voller Pipe
- Lesen blockiert bei leerer Pipe

Message Queue

Nachrichten werden als Ganzes gesendet und empfangen.
Eine Nachricht besitzt ein gewisse Struktur:
Typ und Daten unmittelbar, die Länge wird beim Senden angegeben



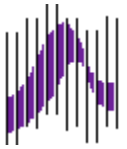
Message Queue



Funktionen zu Message Queues

- Erzeugen: system-global, Namenskonflikte können auftreten
- Löschen
- Öffnen: Kennung muss bekannt sein
- Senden
- Empfangen
  - FIFO über alle Nachrichten
  - FIFO nach Msg-Typ getrennt

Shared Memory



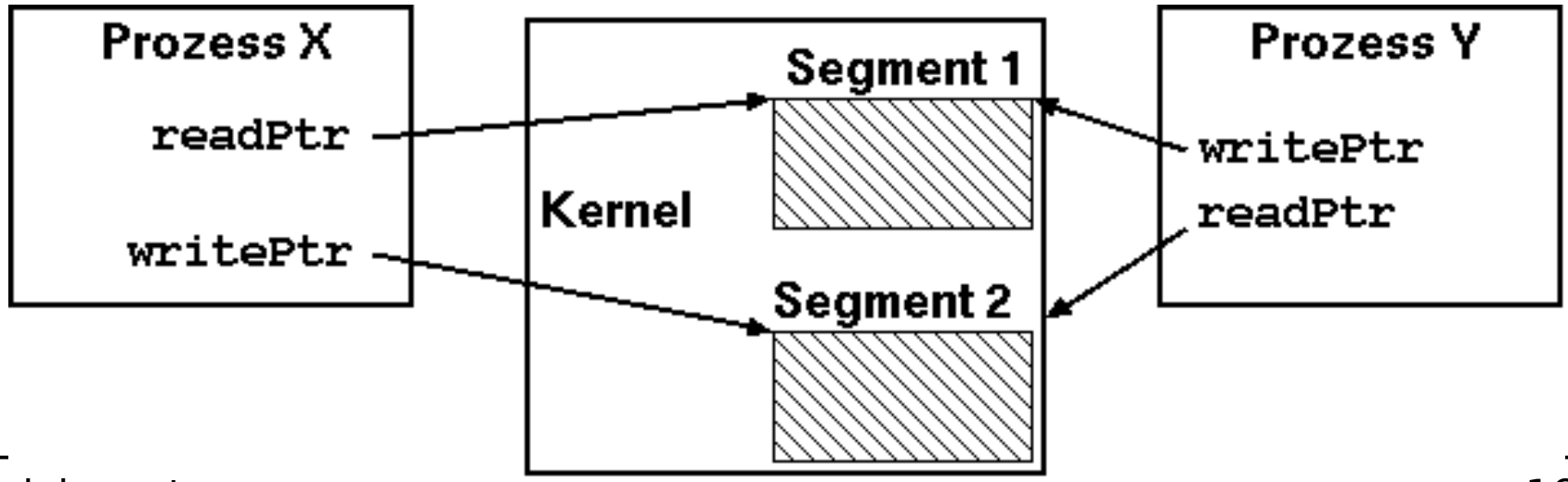
- Gemeinsamer Speicherbereich für zwei oder mehrere Prozesse
- Zugriff muss synchronisiert werden
  - Semaphore
  - evtl. andere Mechanismen
- Prozesse besitzen einen Pointer auf den Anfang des Speicherbereichs.
- Synchronisation und Strukturierung bzw. Interpretation des Speicherbereichs ist Sache der Anwendung
- Schnellste Form der Interprozesskommunikation

Shared Memory

Funktionen:

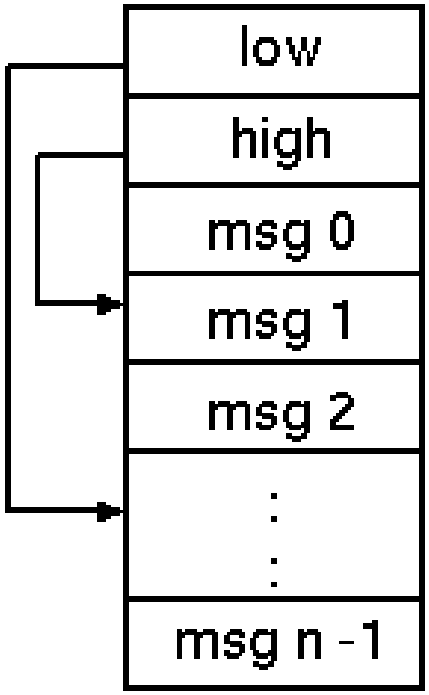
- Einrichten Shared Memory Segment anlegen
- Anbinden (Pointer setzen)
- Löschen

Ggf. werden zwei Shared Memory Segmente angelegt, um die Synchronisation zu vereinfachen.



Shared Memory – Producer Consumer

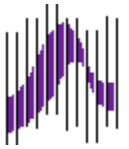
Shared Memory Segment als Puffer eines Producer-Consumer-Systems



- Semaphore: Mutex, Full, Empty
- Zeiger auf erstes und letztes Element sind Teil des Shared Memory

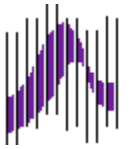
```
struct shm_buffer{
    int high;
    int low;
    IPC_MSG messages[n];
};
```

Sockets



- Schnittstelle zwischen einem Prozess und einem Transportprotokoll
- bidirektional
- Kommunikation innerhalb eines Rechners und über Netzwerk
- Transportprotokoll z.B. TCP oder UDP
- Adressierung über Host-Adresse und Port

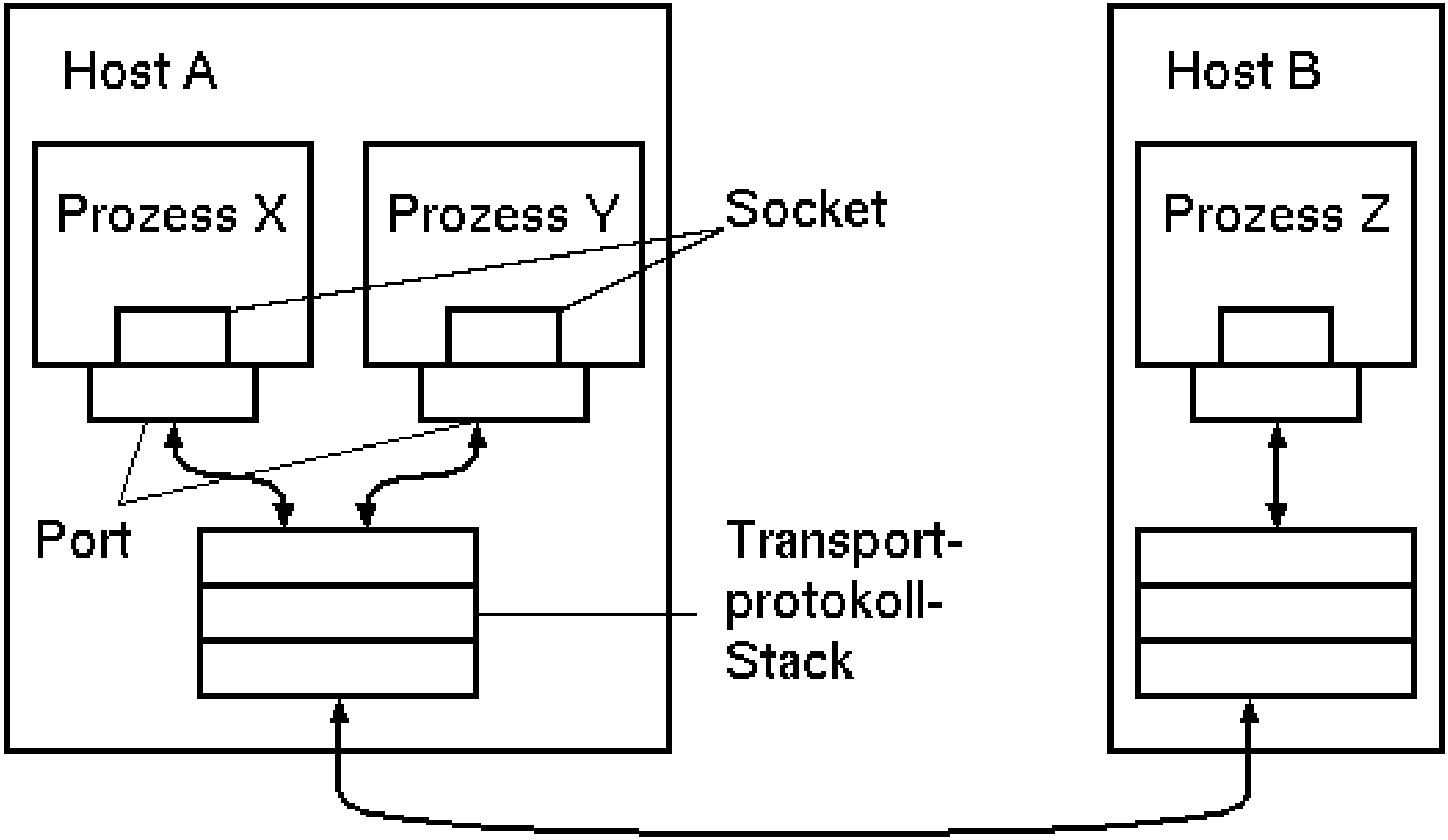
Sockets



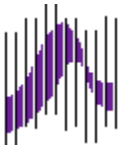
- ServerSocket
  wartet auf Verbindungsanfragen
- ClientSocket
  versucht eine Verbindung herzustellen

Wenn der ServerSocket eine Verbindungsanfrage erhält, erzeugt er einen ClientSocket, der die Verbindung repräsentiert.

Sockets 2

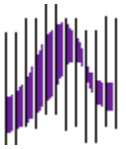

Host A
Prozess X
Prozess Y
Socket
Port
Transport-
protokoll-
Stack
Host B
Prozess Z

Sockets: Network Byte Order



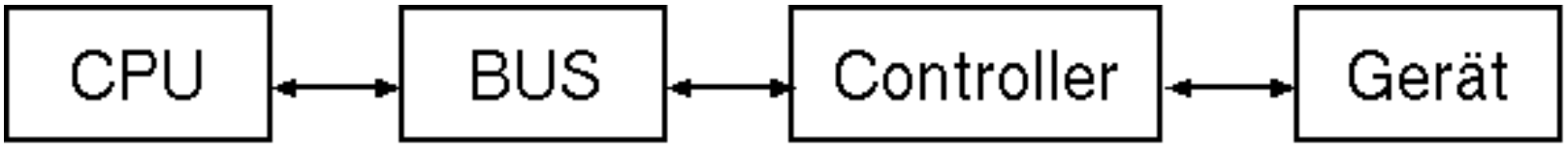
- Austausch von Daten zwischen unterschiedlichen Systemen (Little Endian, Big Endian)
- Daten werden zur Übertragung einheitlich kodiert. Kodierung ist Sache der Anwendung.
- Umwandlungsfunktionen:
  hton (HostToNetwork) ntoh (NetworkToHost)
  für 32- und 16-Bit-Zahlen

Ein- und Ausgabe

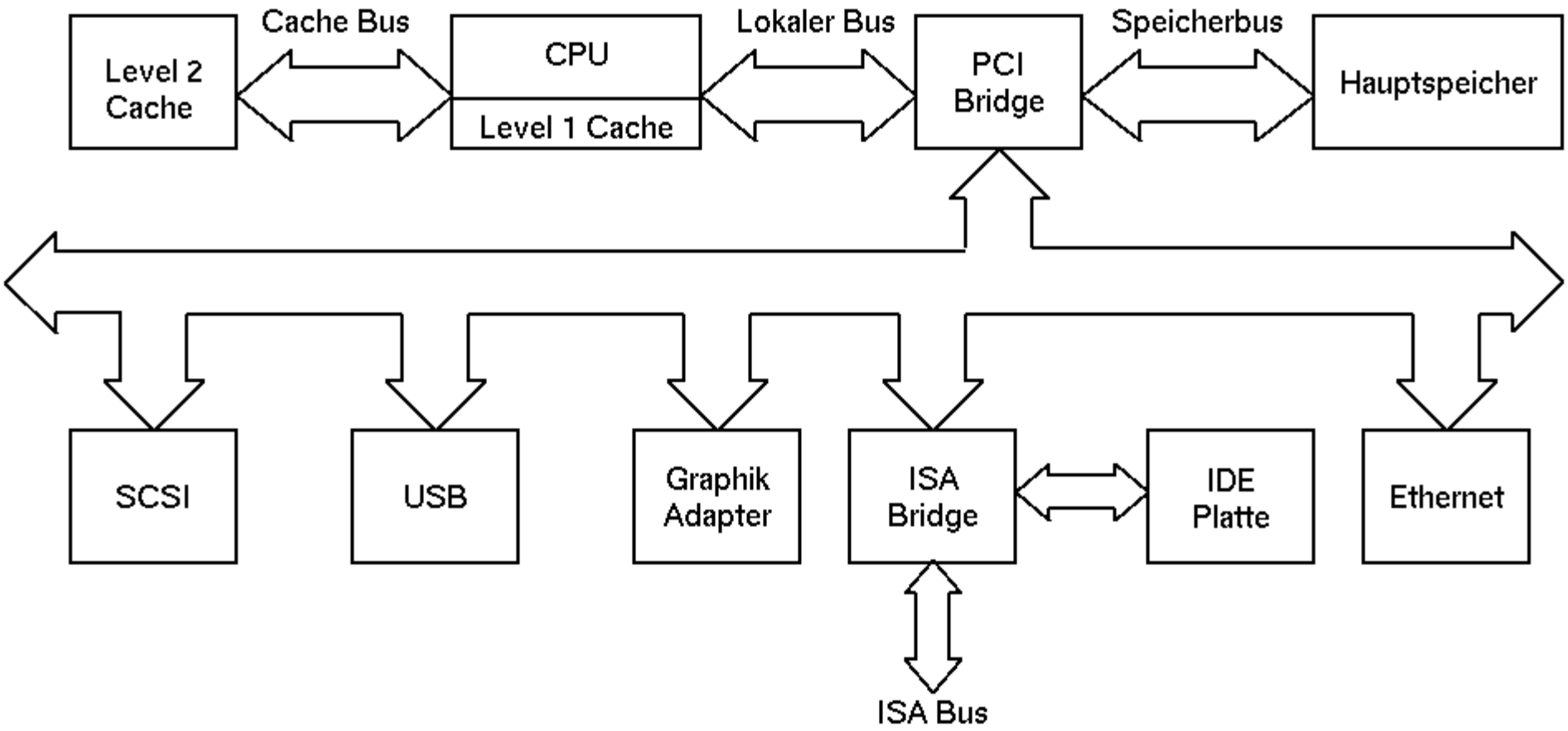


Aufgaben des Betriebssystems

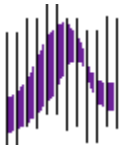
- Überwachung und Steuerung von Geräten
  - Befehle und Daten an Geräte übermitteln
  - Daten und Statusmeldungen entgegennehmen
- Schnittstellen für Anwendungsprogramme
  - möglichst alle Geräte einheitlich ansprechbar
  - Innerer Aufbau soll verborgen bleiben



Aufbau eines Rechners

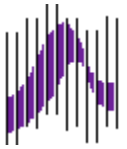


Ein- und Ausgabe Geräte



- Blockorientierte Geräte
  - Daten-Blöcke können adressiert werden
  - Festplatte, SSD
  - Memory-Stick, etc.
- Zeichenorientierte Geräte
  - Fortlaufende Ein- oder Ausgabe
  - Netzwerkkarte (unter Linux eine eigene Kategorie)
  - Drucker
  - Maus, Tastatur, etc.
- Sonstige Geräte
  - Uhr, Bildschirm

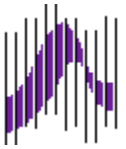
Ein- und Ausgabe Geräte



Geräte werden über Geräte-Register gesteuert

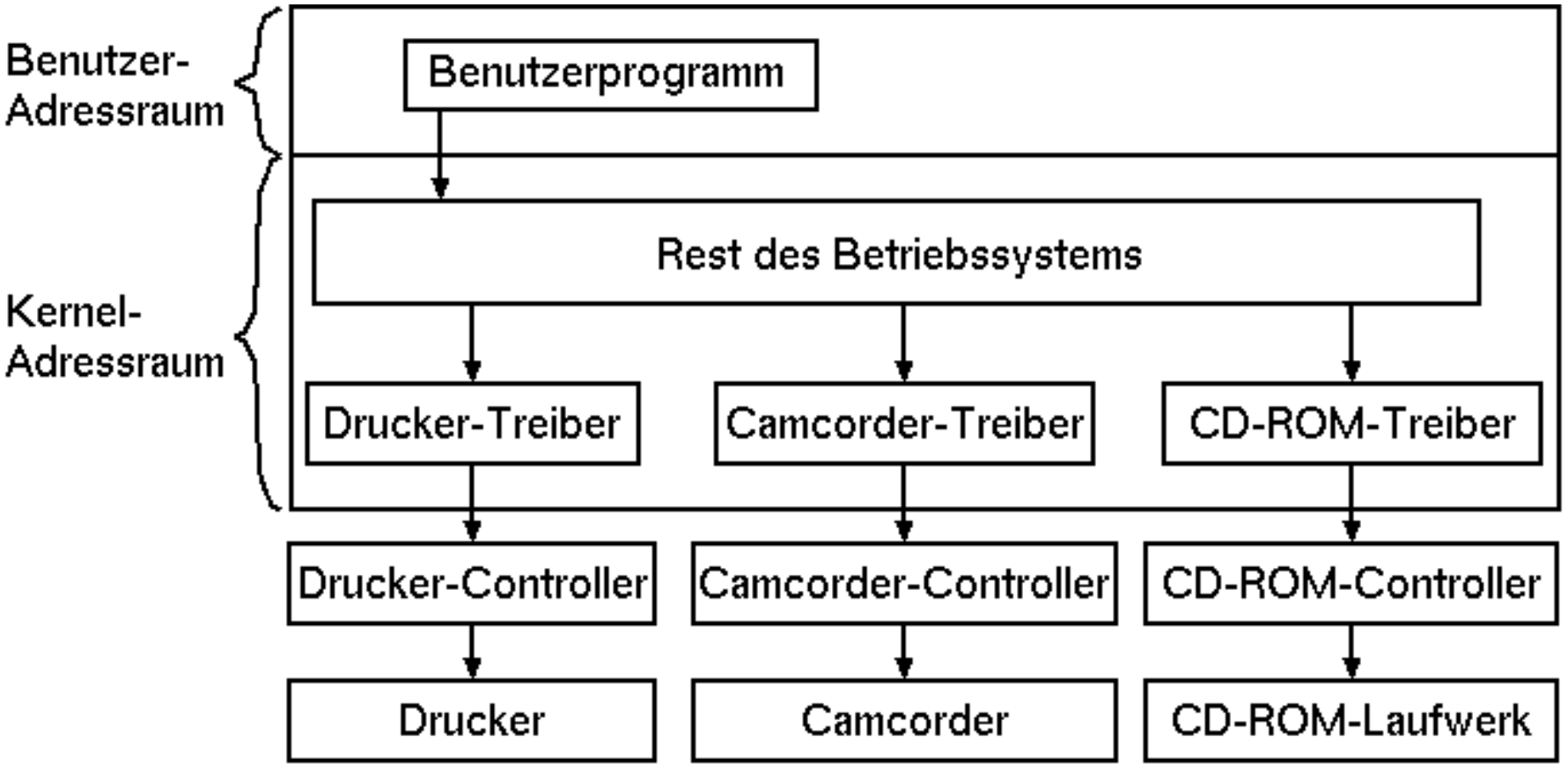
- I/O-Ports
  - Eigener Adressraum
  - Eigene Assembler-Befehle
  - Programmierung mit Assembler
- Memory mapped
  - Teil des Adressraums wird auf Geräte-Register abgebildet
  - in C programmierbar
  - Caching muss für I/O-Adressen ausgeschaltet werden
  - Adressen müssen erkannt werden
    (z.B. in der PCI-Bridge: Adressen beim Booten laden)

Schnittstelle zum Anwendungsprogramm

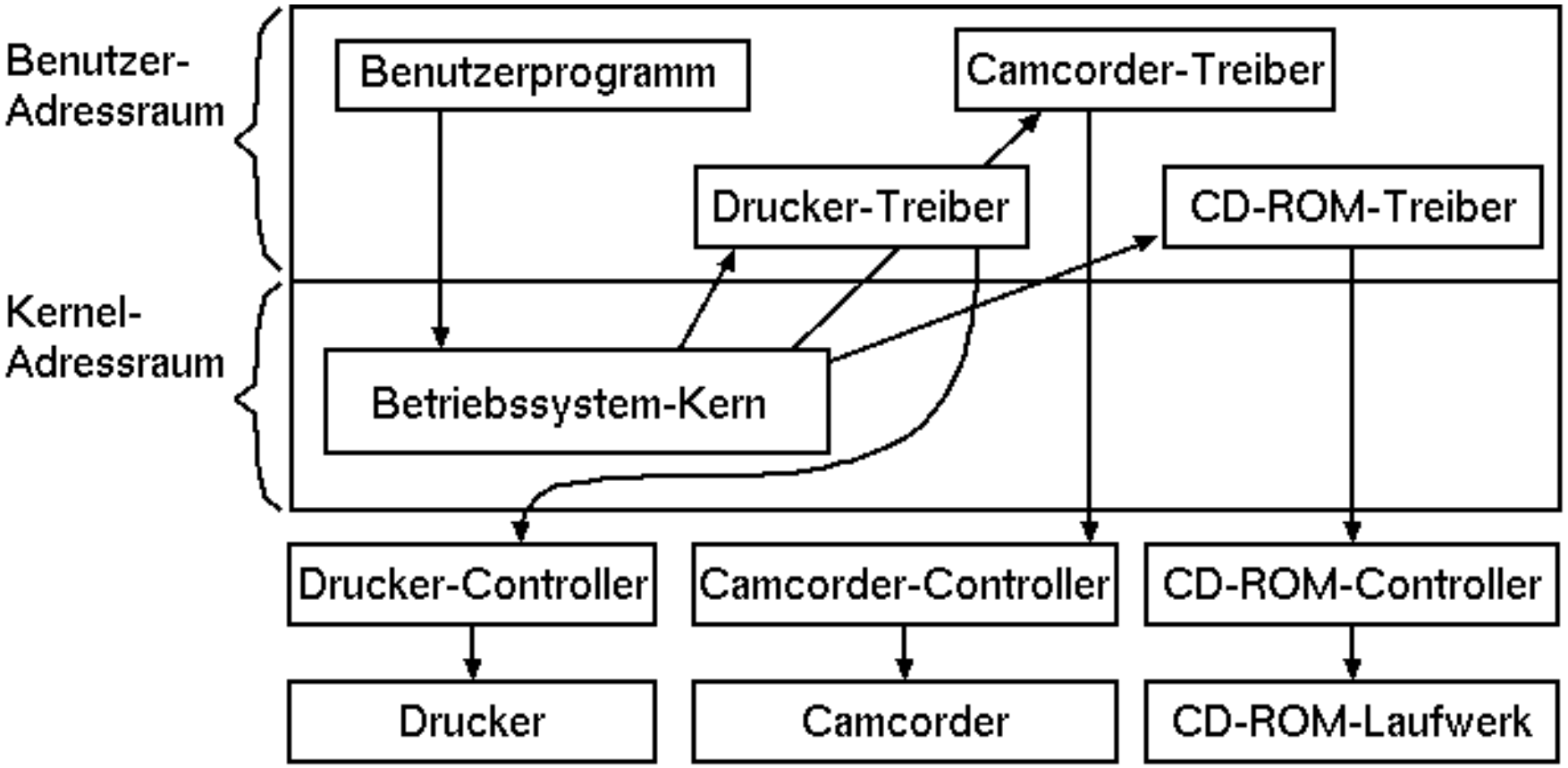


- Einheitlicher Namensraum
  - „mounten“ von Geräten (blockorientierte Geräte)
- Fehlerbehandlung
  - nur nicht behebbare Fehler melden
- Pufferung
- Synchrone Schnittstelle für asynchrone I/O-Operationen
  - Lesefunktion blockiert, bis Daten verfügbar sind
- Gemeinsamer oder exklusiver Zugriff auf Geräte
  - z.B. Festplatte vs. Bandlaufwerk

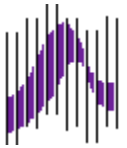
Aufbau des I/O-Systems



Alternativer Aufbau (Micro Kernel)

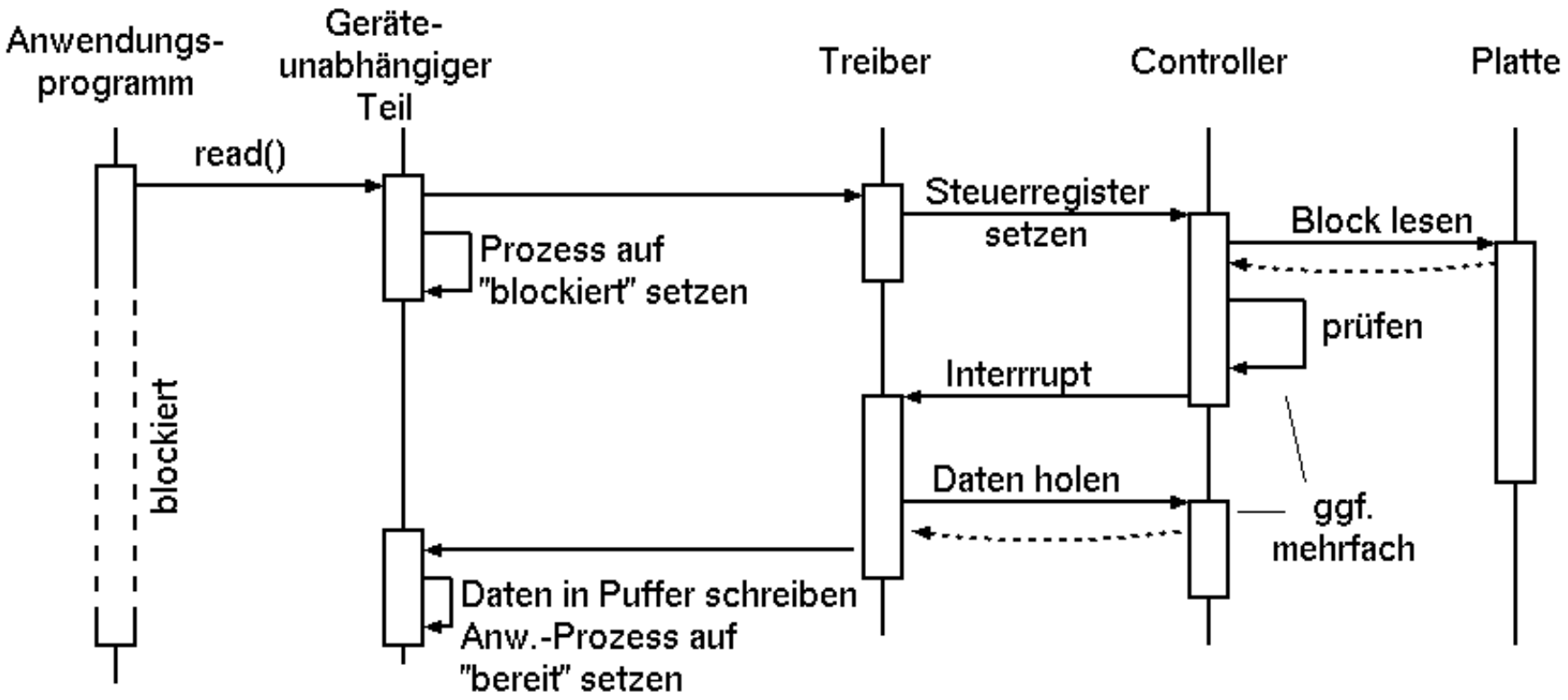


Ein-/Ausgabe-Konzepte

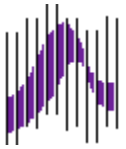


- Programmierte Ein-/Ausgabe
  (1) Der Prozess überträgt Daten-Wort zu Gerät
  (2) Der Prozess fragt Gerät ab und geht zu (1) oder (2)
- Interruptgesteuerte Ein-/Ausgabe
  Für jedes Gerät wird eine Interrupt-Routine eingerichtet
  (1) Prozess überträgt Daten in den Kernel
  (2) Der Prozess überträgt 1. Daten-Wort zu Gerät
  (3) Der Prozess lässt sich blockieren
  (4) Das Gerät sendet einen Interrupt
  (5) Eine Unterbrechungsroutine überträgt das nächste Wort zum Gerät (weiter mit (4)) oder fertig.
- Direct Memory Access
  Wie Programmierte Ein-Ausgabe mit eigenem Prozessor

Interrrupt gesteuerte Ein-/Ausg.



Direct Memory Access



Viele Rechner nutzen einen DMA-Controller, um den Prozessor von der Ein- und Ausgabe zu entlasten.
Funktionsweise:

(1) CPU beauftragt DMA-Controller
(meist mit physischen Adressen)

(2) DMA-Controller beauftragt Geräte-Controller

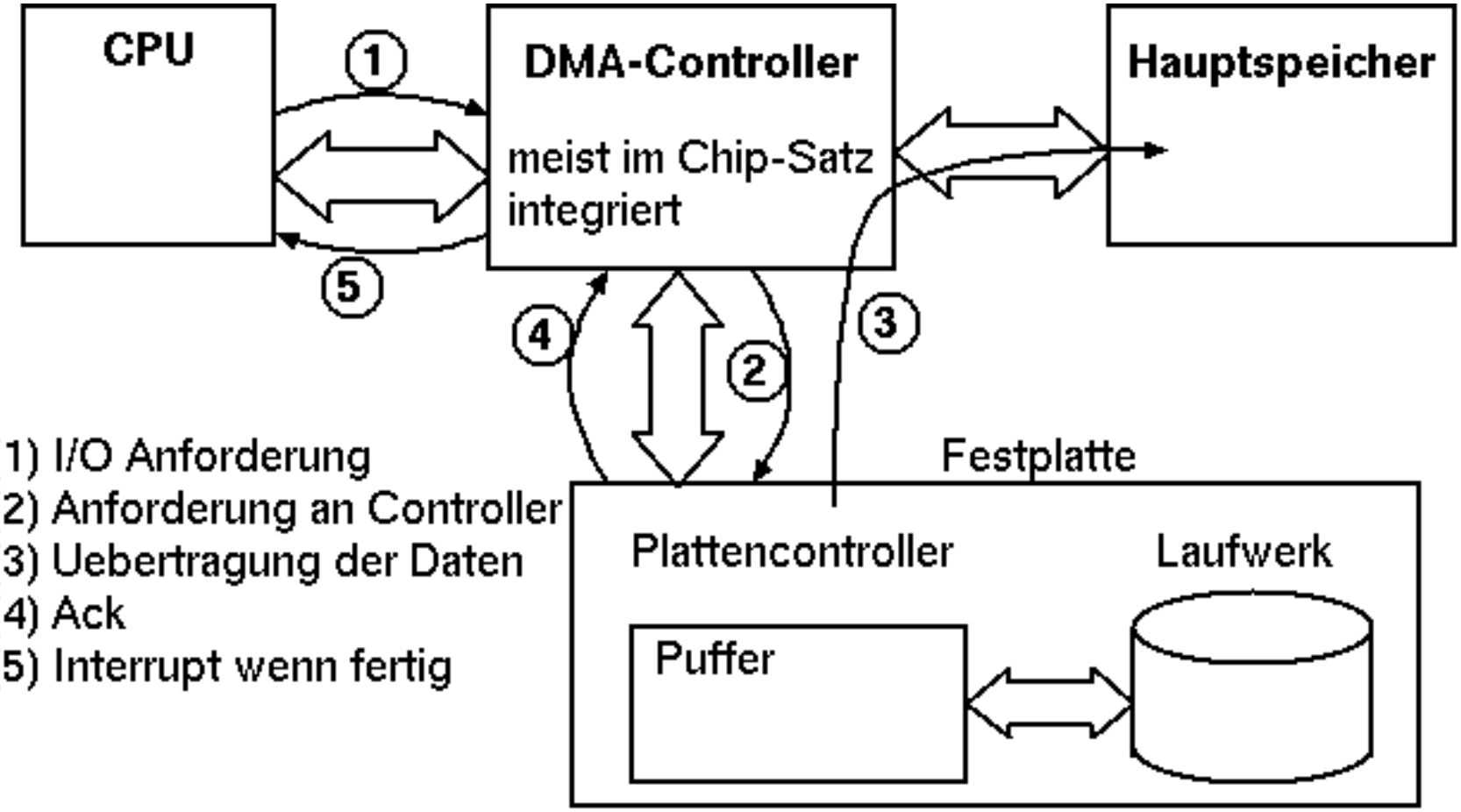
(3) Gerät überträgt Daten und meldet Ende an DMA-Controller

(4) Zurück zu Schritt (2) oder zu (5)

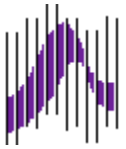
(5) DMA-Controller benachrichtigt CPU

Zu (3): Übertragung von einzelnen Worten oder von Seiten, direkt in den Speicher oder über den DMA-Controller

Direct Memory Access

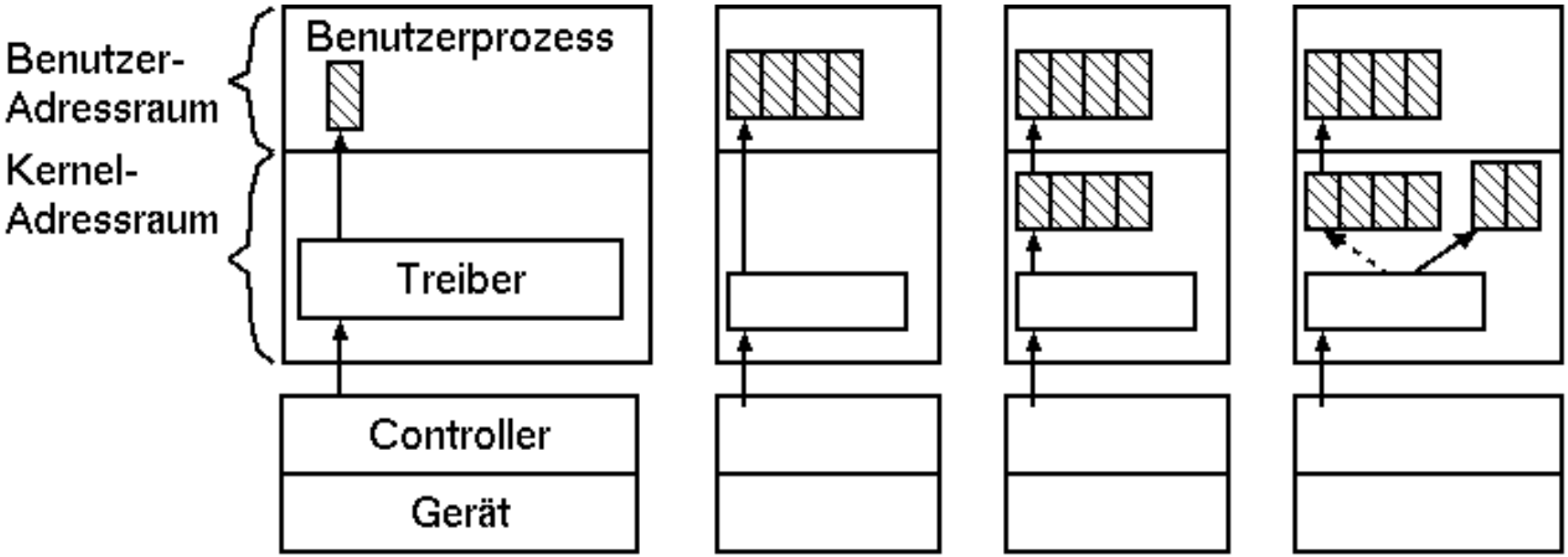


Eingabe-Pufferung

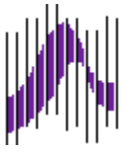


- Keine Pufferung:
  Der Benutzerprozess empfängt jedes Wort einzeln.
- Pufferung im Benutzeradressraum:
  Die I/O-Routine kopiert Worte in den Puffer ohne, dass der Benutzerprozess läuft.
- Pufferung im Kernel:
  Die I/O-Routine legt die Daten in einen Kernel-Puffer. Wenn dieser voll ist, wird der Inhalt in den Benutzeradressraum kopiert
- Doppelte Pufferung im Kernel: Wenn der eine Puffer voll ist, schreibt die I/O-Routine in den zweiten. Währenddessen wird der 1. Puffer kopiert.

Pufferung der Eingabe



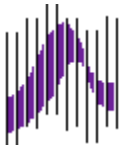
Datei-Systeme



Persistente Speicherung von Daten in Dateien

- Nutzersicht
  - Datei-Operationen
  - Benennung von Dateien (i.Allg. hierarchisch)
  - Typen von Dateien (strikte vs. nichtstrikte Typisier.)
  - Zugriffssteuerung (Rechte-Vergabe)
- Systemsicht
  - Abbildung der Geräte-Schnittstellen auf die Datei-System-Schnittstelle

Festplatten



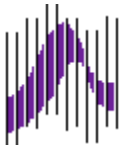
Aufbau

- Mehrere Platten, magnetisch beschichtet
- Konzentrische Spuren auf jeder Plattenseite
- Jede Spur ist in Sektoren unterteilt
- Ein Zylinder ist die Zusammenfassung aller k-ten Spuren der n Plattenseiten

Typische Größen (2008)

- 2- 6 Platten d.h. 4 - 12 Plattenseiten
- 10000 Zylinder
- 100 – 200 Sektoren pro Spur
- 8 – 32 MB Pufferspeicher

Festplatten

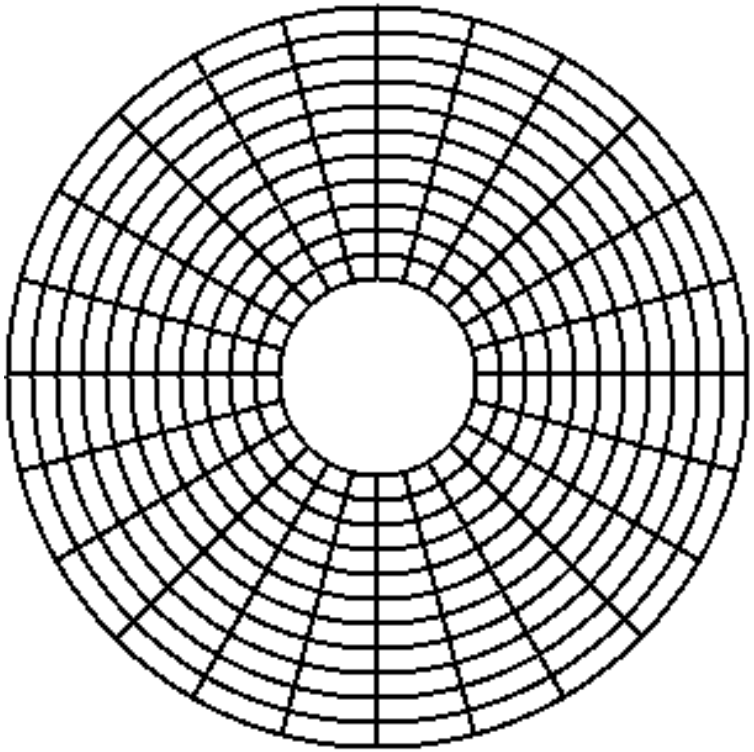


Weitere typische Größen (2008)

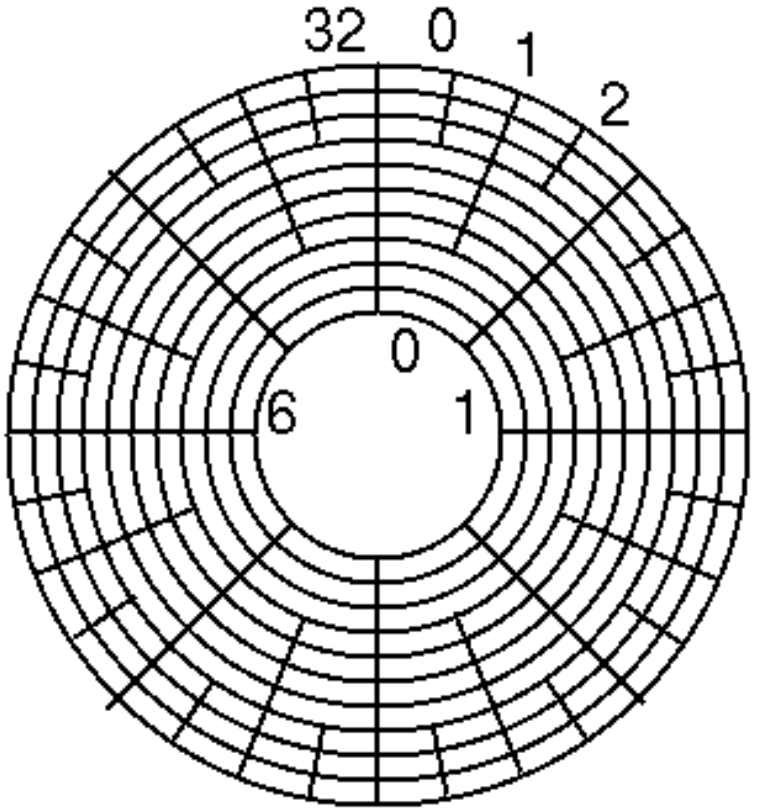
- mittlere Zugriffszeit: 5-10ms
  (CPU: 10 - 20 Millionen Instruktionen)
- Transferrate: 50 - 100MB/s
- Caching für Schreibfunktionen abschaltbar
- Sektorgröße: 512Byte

Formatierung

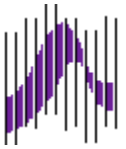
logisches Format



physikalisches Format



CD (ROM, RW)



Eine Spur als Spirale ca. 5,6 km

Einfache Geschwindigkeit ca. 4,3 km/h

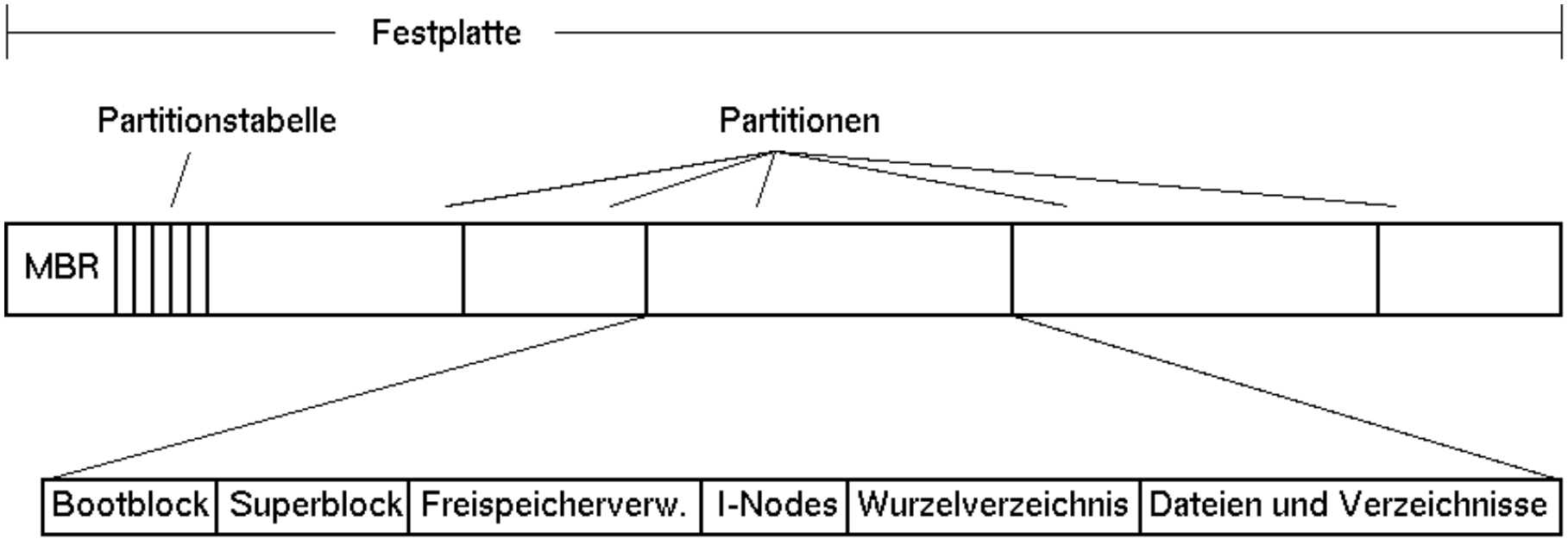
Fehlererkennung und -korrektur in drei Stufen

1) Ein Byte wird in 14 Bit codiert
2) Ein Rahmen enthält 42 Byte (je 14 Bit) und 252 zusätzliche Korrektur-Bits
   Cross Interleaved Reed-Solomon-Code
3) Ein Sektor enthält 98 Rahmen und 288 zusätzliche Korrektur-Bits (ein von zwei Layouts)

Stufe 1) und 2) gemäß Audio-CD, Stufe 3) zusätzlich für Daten. Insgesamt belegt die Fehlerkorrektur ca. 72% der Kapazität.

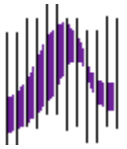
Aufteilung einer Festplatte

Ein weit verbreitetes Layout für Festplatten:



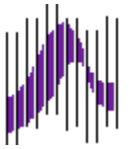
MBR: Master Boot Record
Der Superblock identifiziert den Dateisystemtyp

Partitionen



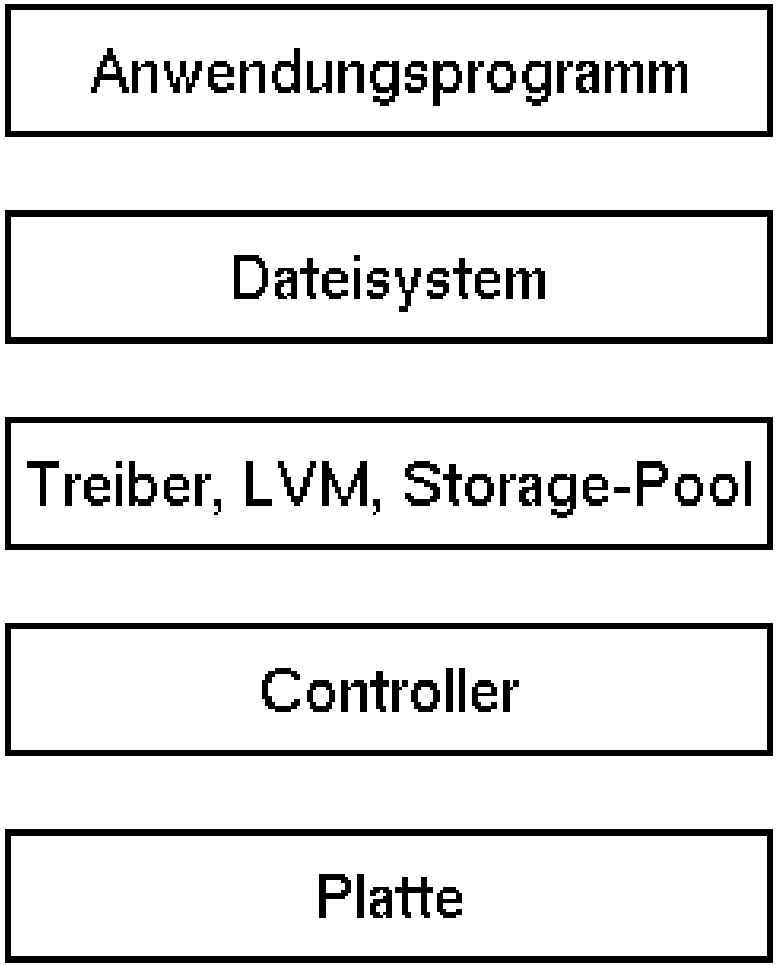
- Windows, Unix, MacOS, ...
  - 4 primäre Partitionen
  - 3 primäre Partitionen und mehrere logische Partitionen (i. Allg. bis zu 64)
- Jede Partition kann (maximal) ein eigenes Dateisystem enthalten.
- logical volume / storage pool
  - Zusammenfassung mehrerer Partitionen
  - Verwaltung durch *ein* Dateisystem

Booten von der Festplatte



- BIOS lädt MBR und führt ihn aus
- MBR-Prog. lokalisiert aktive Partition und lädt Boot-Block
- Der Boot-Block lädt das Betriebssystem oder einen Boot-Loader

Einordnung des Dateisystems



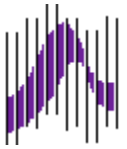
open(), close(), fread() etc.

blockorientierter Speicher,
z.B. Gerät, Partition, Pool
(read, write für Blöcke)

logisches Layout

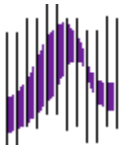
physisches Layout

Datei-Operationen



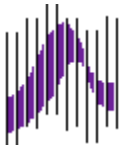
- Erzeugen (auch Verzeichnisse)
- Löschen (auch Verzeichnisse)
- Öffnen
- Schließen
- Lesen
- Schreiben
- Anfügen
- Positionieren
- Attribute abfragen/setzen (auch Verzeichnisse)
- Umbenennen (auch Verzeichnisse)

Datei-Typen



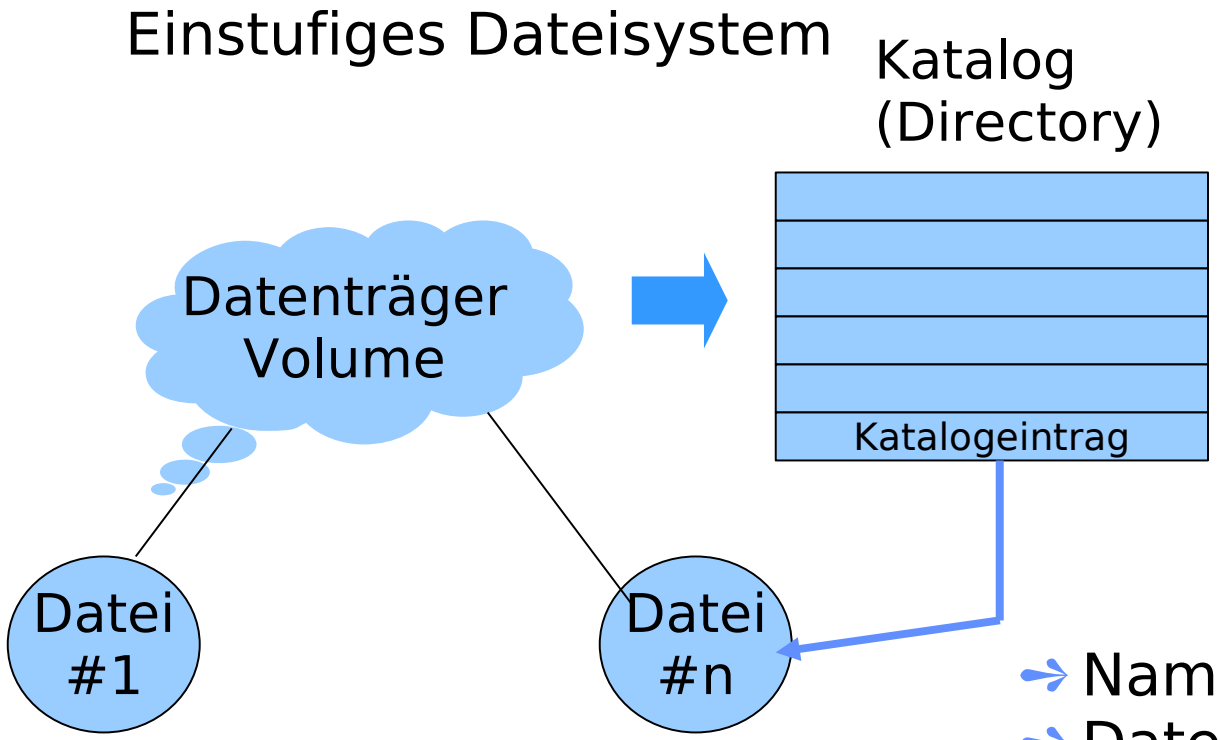
- reguläre Dateien
  - Daten des Benutzers
- Verzeichnisse
  - verwalten bzw. strukturieren das Dateisystem
- Links (hard- bzw. soft-)
  - Verweise auf andere Dateien
- spezielle Dateien
  - Schnittstelle zu zeichenorientiertem Gerät
  - Schnittstelle zu blockorientiertem Gerät
  - Pipes, Shared Files, . . .

Beispiele für Datei-Typen



- reguläre Datei
  -rw-r--r-- 1 martin martin   3,6K 2008-12-23 10:41 print.ps
- Verzeichnis
  drwxr-xr-x   5 martin martin 4.0K 2008-12-09 12:27 tmp
- Soft-Link
  lrwxrwxrwx . . . 11 2008-12-23 10:44 www -> public_html
- spezielle Dateien
  - Schnittstelle zu zeichenorientiertem Gerät
    crw-rw-rw- 1 root tty 5, 0 2008-12-22 10:35 /dev/tty
  - Schnittstelle zu blockorientiertem Gerät
    brw-rw---- 1 root disk 3, 2 2008-12-22 10:35 /dev/hda2
  - Named Pipe
    prw-r----- 1 martin martin   0 2008-12-09 12:29 test.fifo

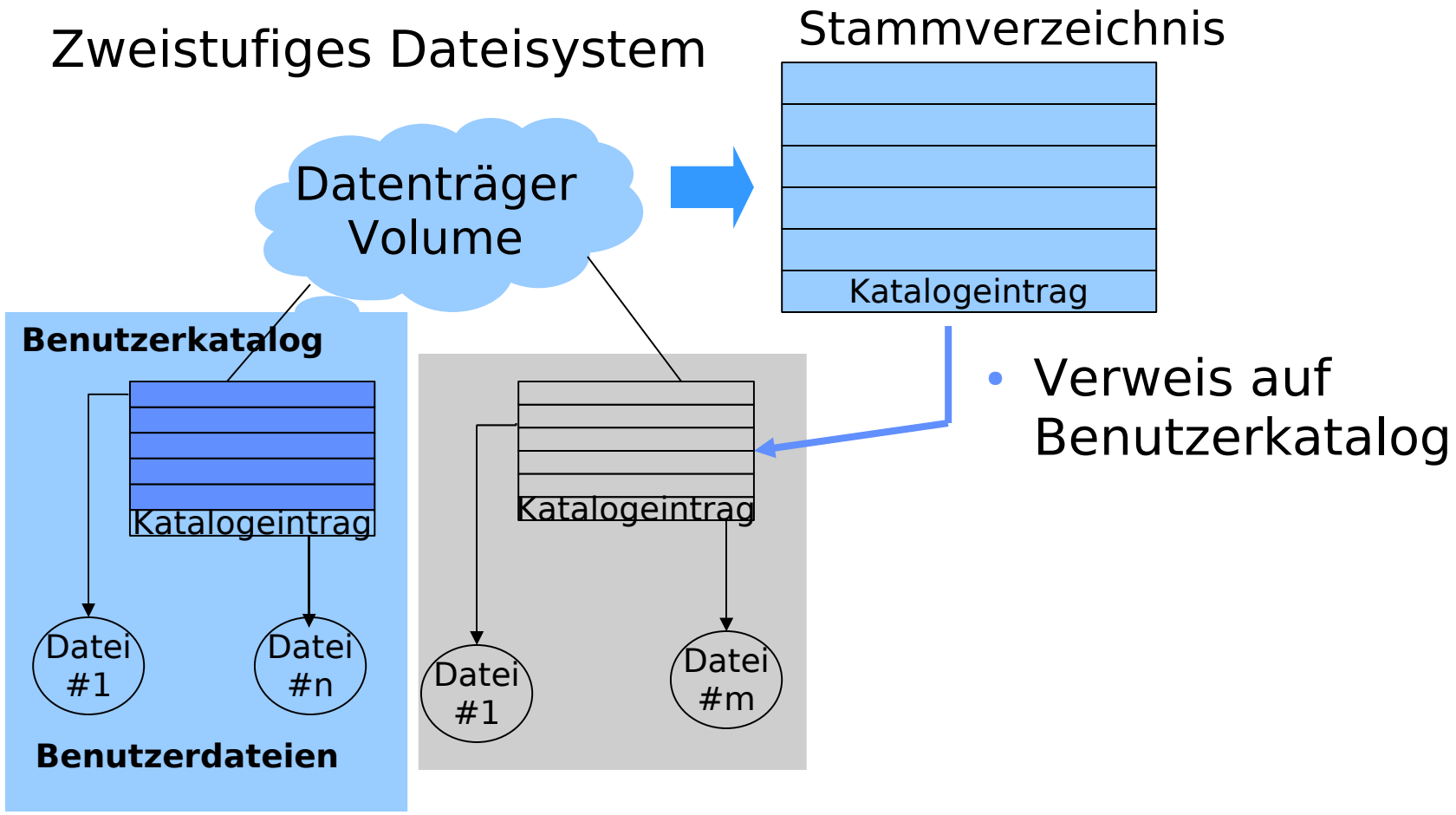
Aufbau des Dateisystems - Kataloge



- Inhaltsverzeichnis des Volume
- Verwaltung der Dateien
- Pro Datei genau 1 Eintrag

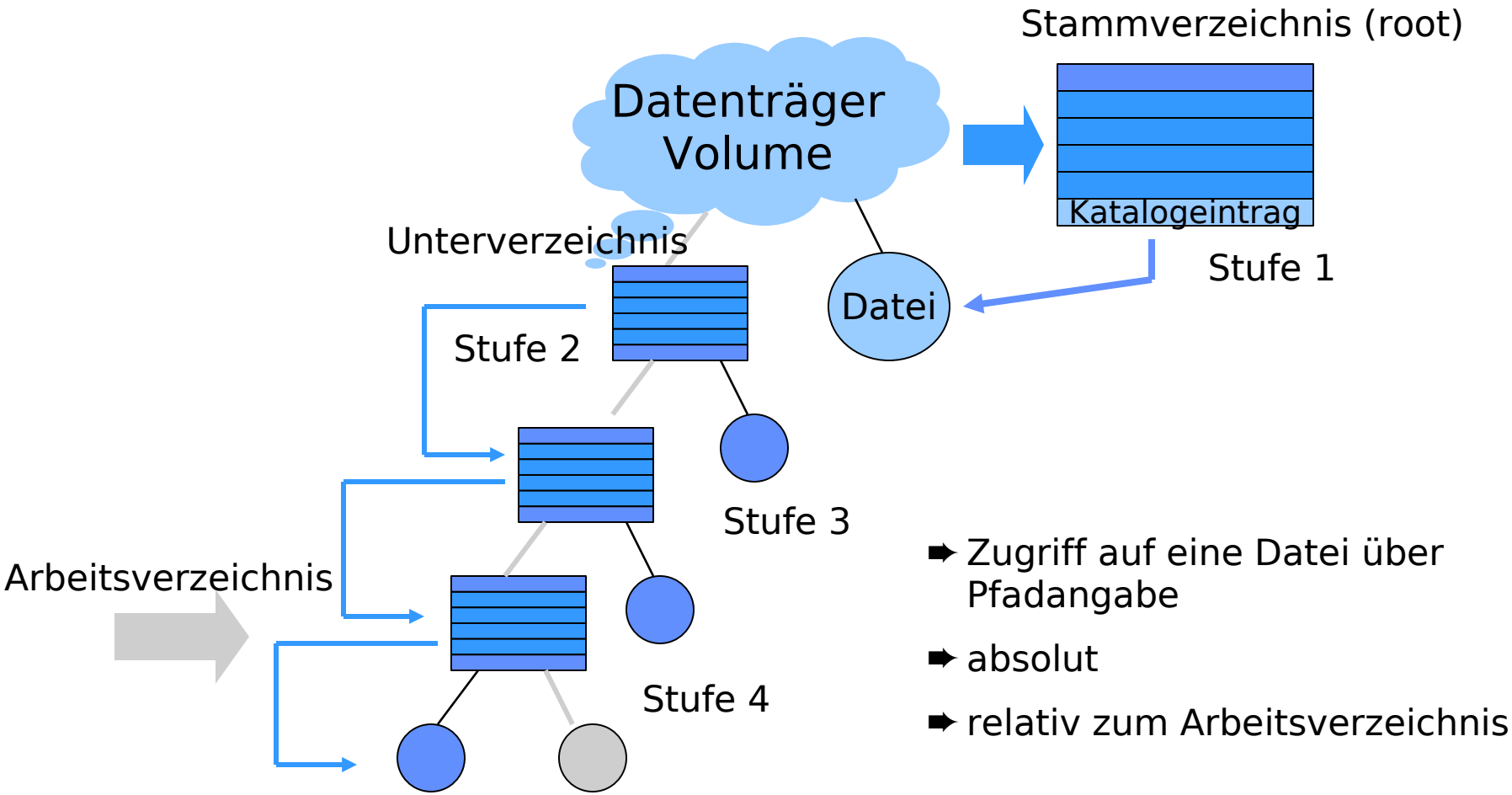
- Name der Datei
- Dateiattribute
- Ort der Daten auf dem Volume

Aufbau des Dateisystems - Kataloge



Aufbau des Dateisystems - Kataloge

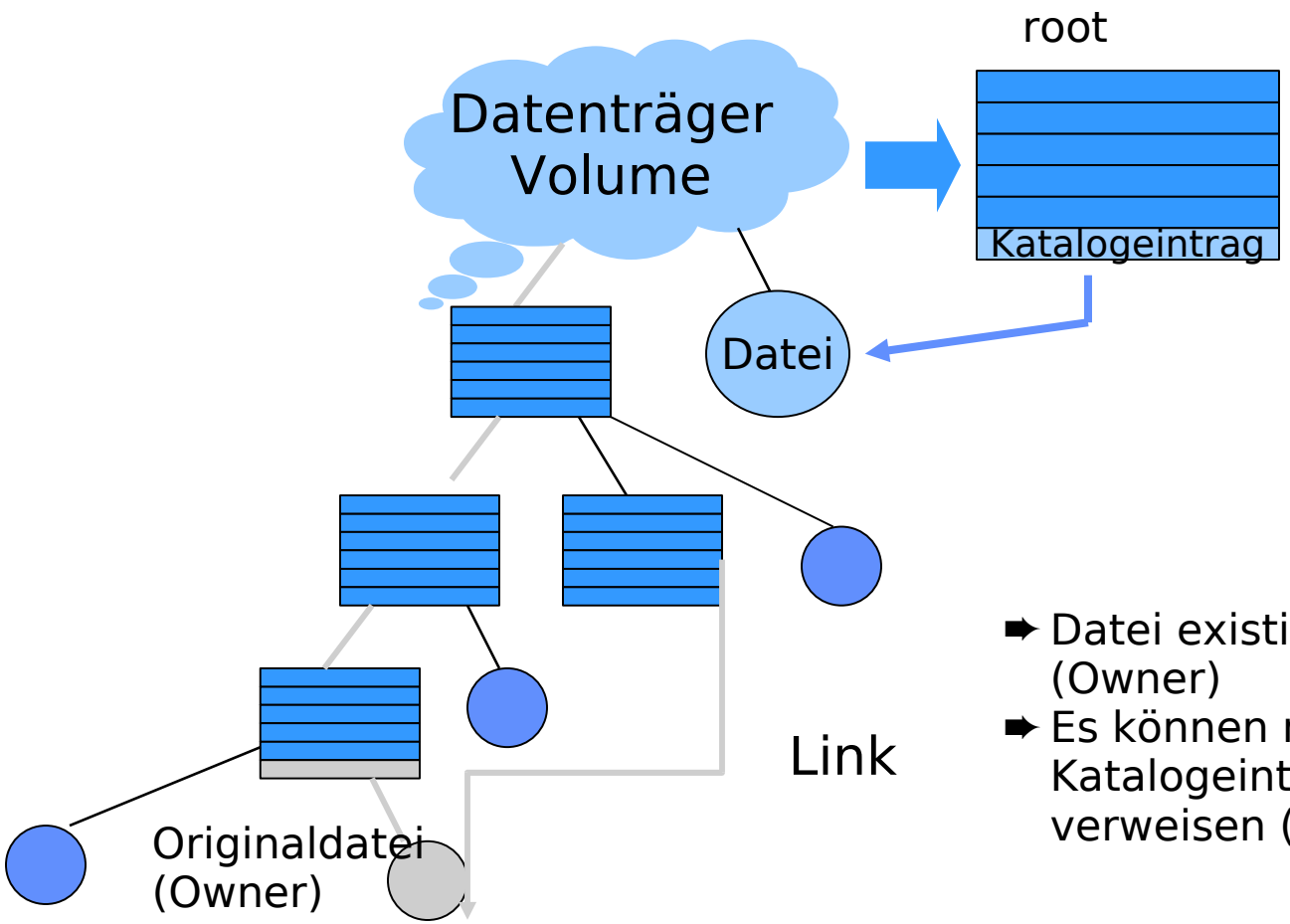
Hierarchisches Dateisystem



- Zugriff auf eine Datei über Pfadangabe
- absolut
- relativ zum Arbeitsverzeichnis

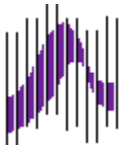
Aufbau des Dateisystems - Kataloge

Hierarchisches Dateisystem mit Links



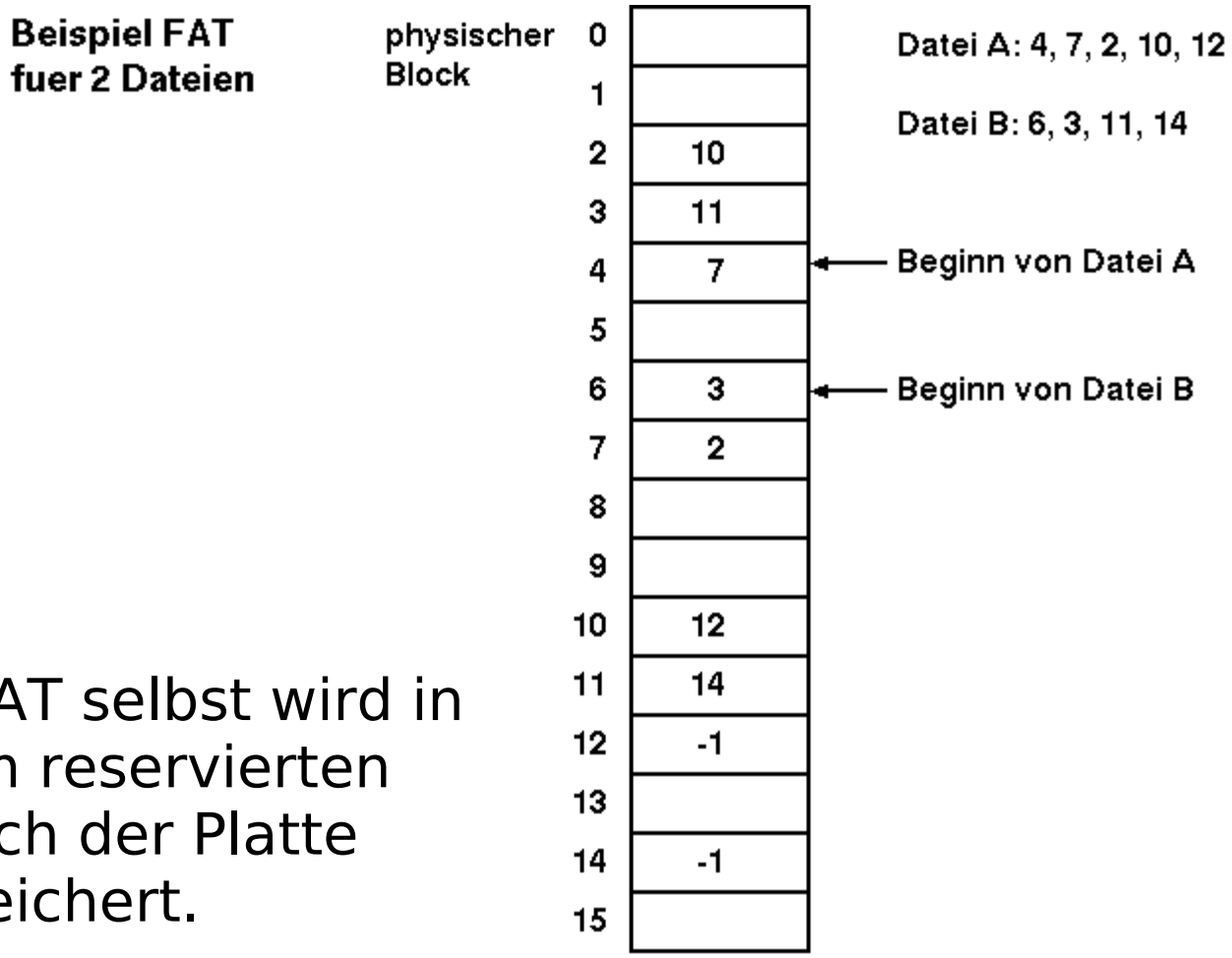
- Datei existiert genau einmal (Owner)
- Es können mehrere Katalogeintrage auf die Datei verweisen (Links)

Belegung von Blöcken



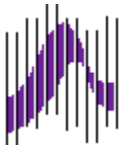
- Zusammenhängend
  - alle Blöcke einer Datei liegen unmittelbar hintereinander
  - einfach und performant
  - unflexibel und fragmentierend
  - wird für CD-ROMs genutzt
- Verkettete Liste auf der Platte
  - keine Fragmentierung
  - wahlfreier Zugriff ineffizient
- Verkettete Liste mit Tabelle im Hauptspeicher FAT
  - effizient
  - hoher Verbrauch an Hauptspeicher;
    ein Eintrag für jeden (logischen) Block der Festplatte
- Baum-Struktur, Wurzel: I-Node, Uberblock, ...

Beispiel für eine FAT



Die FAT selbst wird in einem reservierten Bereich der Platte gespeichert.

FAT Verzeichniseintrag



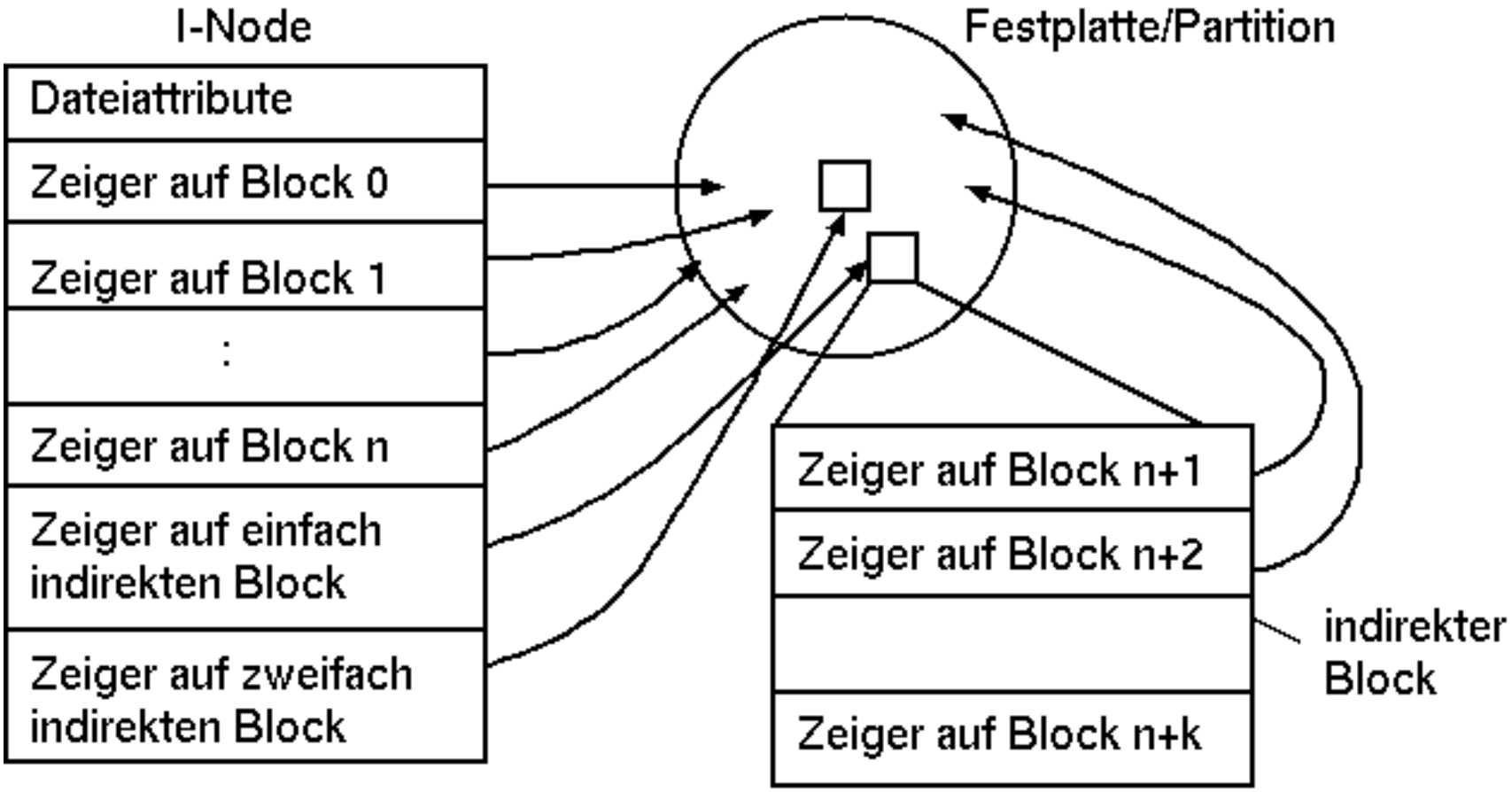
Ein Directory/Verzeichnis ist eine Datei.

Ein Eintrag enthält:

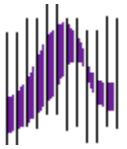
- Name der Datei
- Attribute
- Zeiger auf ersten Block
- Länge

Die Datei-Attribute stehen also im Verzeichnis.

I-Node (Prinzip)



I-Node



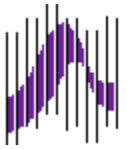
Für jede geöffnete Datei wird der I-Node im Hauptspeicher gehalten.

Speicherbedarf ist nur abhängig von der Größe eines I-Nodes und der Anzahl der geöffneten Dateien.

- effizient und flexibel
- akzeptabler Speicherverbrauch

Ein I-Node belegt i.Allg. 64 oder 128 Byte

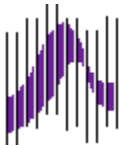
I-Node



Die Datei-Attribute werden im I-Node gespeichert (nicht aber der Namen und der Typ der Datei)

- Besitzer
- Gruppe
- Zugriffsrechte
- Zeitstempel (Änderung, Zugriff, Löschen)
- Größe der Datei in Bytes
- Anzahl der (hard-)Links
- . . .

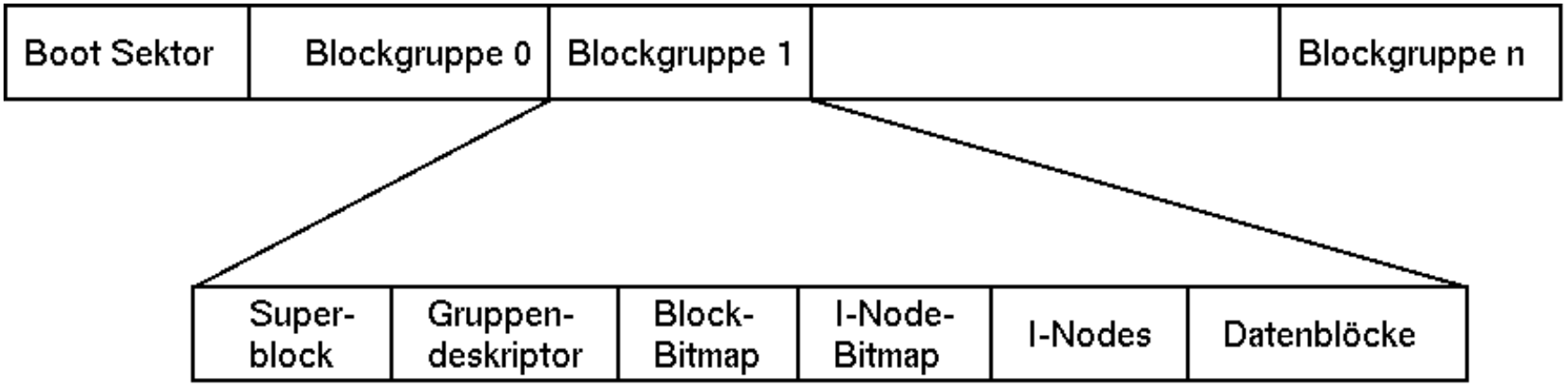
Verwaltung freier/defekter Blöcke



- Verkettete Liste
  - Ein freier Block enthält Verweise auf andere freie Blöcke und einen Verweis auf den nächsten Verwaltungsblock
- Bitmap
  - Für jeden Block des Dateisystems wird ein Bit belegt.
  - Benachbarte Bits bezeichnen benachbarte Blöcke; eine Datei sollte benachbarte Blöcke belegen.

Einige Dateisysteme reservieren für eine Datei mehrere sequentielle Blöcke, um Dateien möglichst kompakt zu halten.

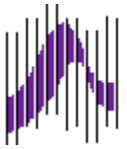
Struktur einer Partition in ext2



Gruppendeskriptor enthält:

- Position der Bitmaps (I-Nodes und Blöcke)
- Anzahl der freien Blöcke, der freien I-Nodes (Blockgruppe)
- Anzahl der Verzeichnisse
- . . .

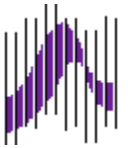
Superblock in ext2



Superblock für das Dateisystem / die Partition:

- Magic Number
- Blockgröße
- Größe der Blockgruppen
- Anzahl der freien Blöcke, der freien I-Nodes (gesamte Partition)
- Versions-Nummer
- Mount Count, Zeitpunkt der letzten Überprüfung
- . . .

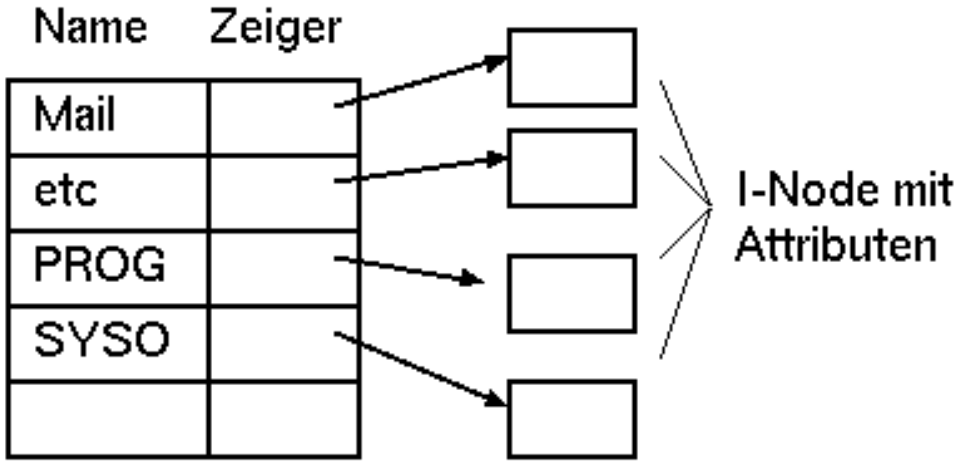
Verzeichnisse: Einträge fester Länge



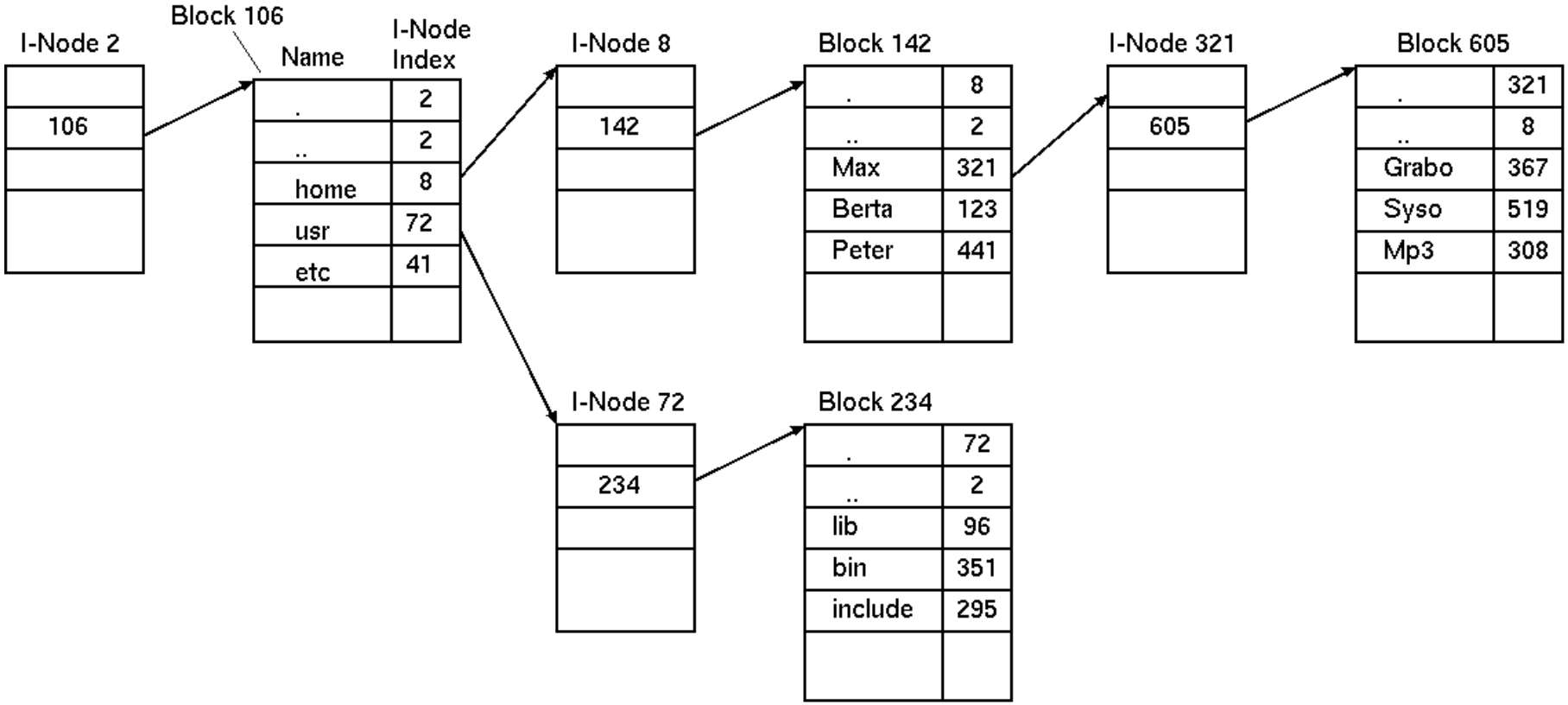
Attribute im Verzeichnis (Redundanz bei hard-Links)

Name	Attribute
Mail	
etc	
PROG	
SYSO	

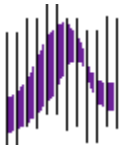
Verweise auf I-Nodes mit Attributen. Das Verzeichnis selbst belegt weniger Blöcke; keine redundante Attribute.



Verzeichnisse und I-Nodes



Hierarchisches Dateisystem mit Links



Logischer Link

- Der Link wird über eine zusätzliche (Link-)datei realisiert
- Die (Link-)datei enthält nur den Pfadnamen der verlinkten Datei
- Der Katalogeintrag verweist auf diese angelegte Link-datei.

Nachteil: Wird die Originaldatei gelöscht oder verschoben so existiert die (Link-)datei noch und enthält den alten unveränderten Pfadnamen. Der Link ist ist aber ungültig

Vorteil: Netzwerkfähig, da der Pfadname eine Netzwerkadresse beinhalten kann.

Beispiel: Unix, Windows NT

Hard-Link

Der Katalogeintrag verweist auf eine physikalische Adresse, in der eine Beschreibung der Originaldatei liegt. Diese Dateibeschreibung ist fest mit der Originaldatei verbunden und enthält alle Datei-attribute, auch die physikalische Lage auf der Festplatte und einen Link-Zähler.

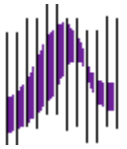
Vorteil: Beim verschieben oder umbenennen der Datei bleibt der Link auf die Dateibeschreibung gültig. Beim Löschen wird der Linkzähler geprüft. Die Datei wird nur dann gelöscht wenn der Linkzähler auf 1 steht, sonst wird nur der Link-Zähler erniedrigt.

Nachteil: Löscht der Erzeuger A die Datei während noch ein weitere Link von B existiert, wird zwar die Datei nicht gelöscht. Der Katalogeintrag von A wird gelöscht, der Link von B existiert weiter. Jedoch wird nicht der Besitzereintrag in der Dateibeschreibung geändert. Besitzer bleibt also A.

Beispiel: UNIX

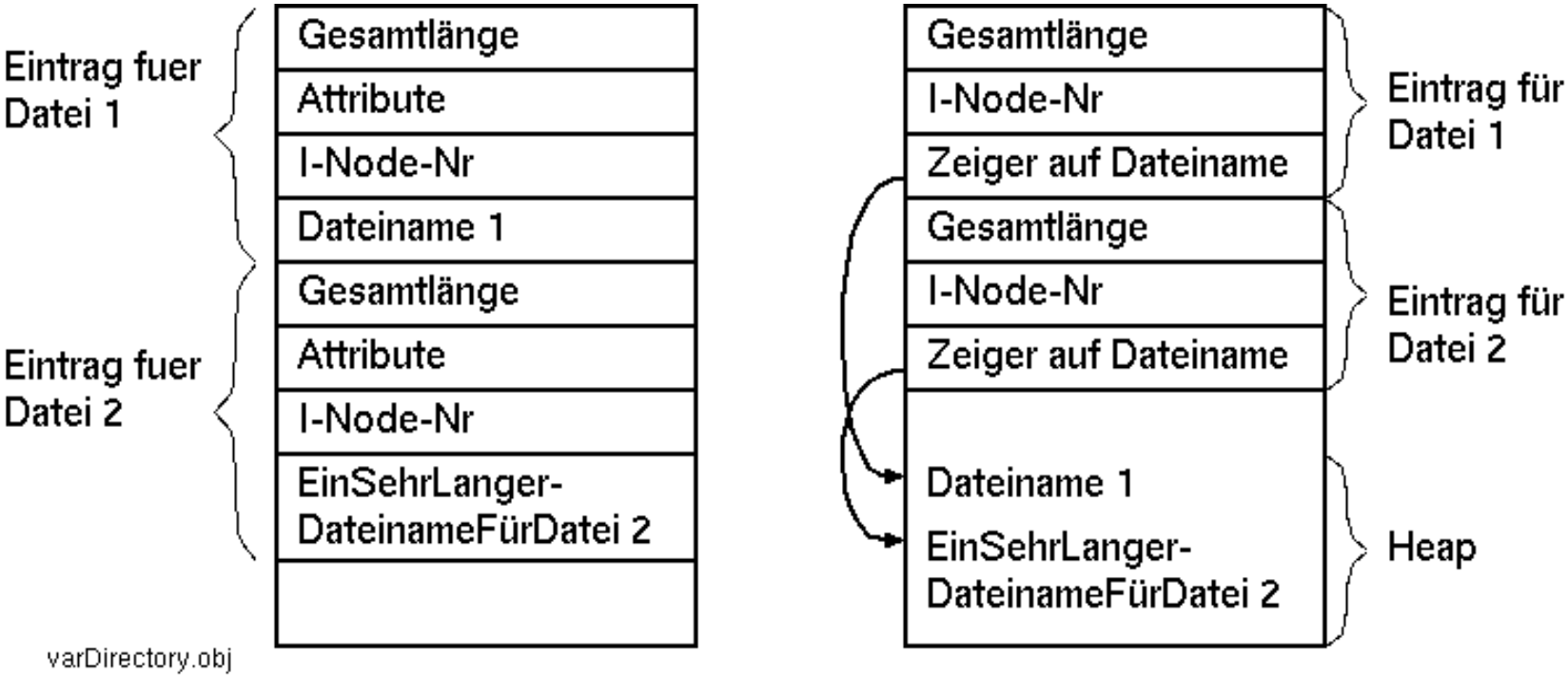
Problem: Links können dazu führen, dass Dateien und/oder Verzeichnisse mehrfach kopiert werden, z.B. bei einem Backup.

Verzeichnisse mit langen Namen



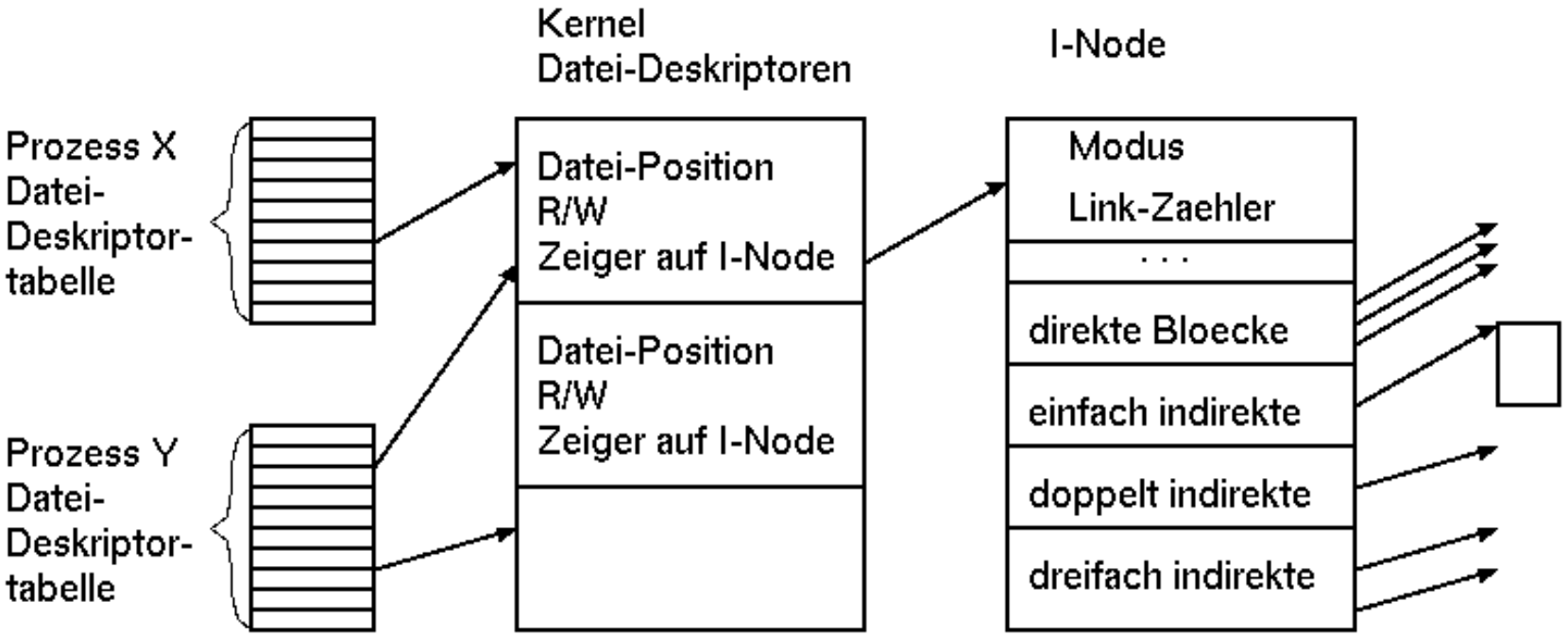
- Problem: Speicherplatz und Zugriffszeit
- Verkettete Liste
  Die Einträge sind unterschiedlich lang, die Länge steht einer fest vereinbarten Stelle.
- Tabelle mit Pointern in Heap-Speicher
  Die Dateinamen werden in einem separaten Speicherbereich abgelegt.
- Suchstrukturen, z.B. Extendible Hashing
  Die Namen werden in einer Suchstruktur abgelegt.

Verzeichnisse: Einträge variabler Länge



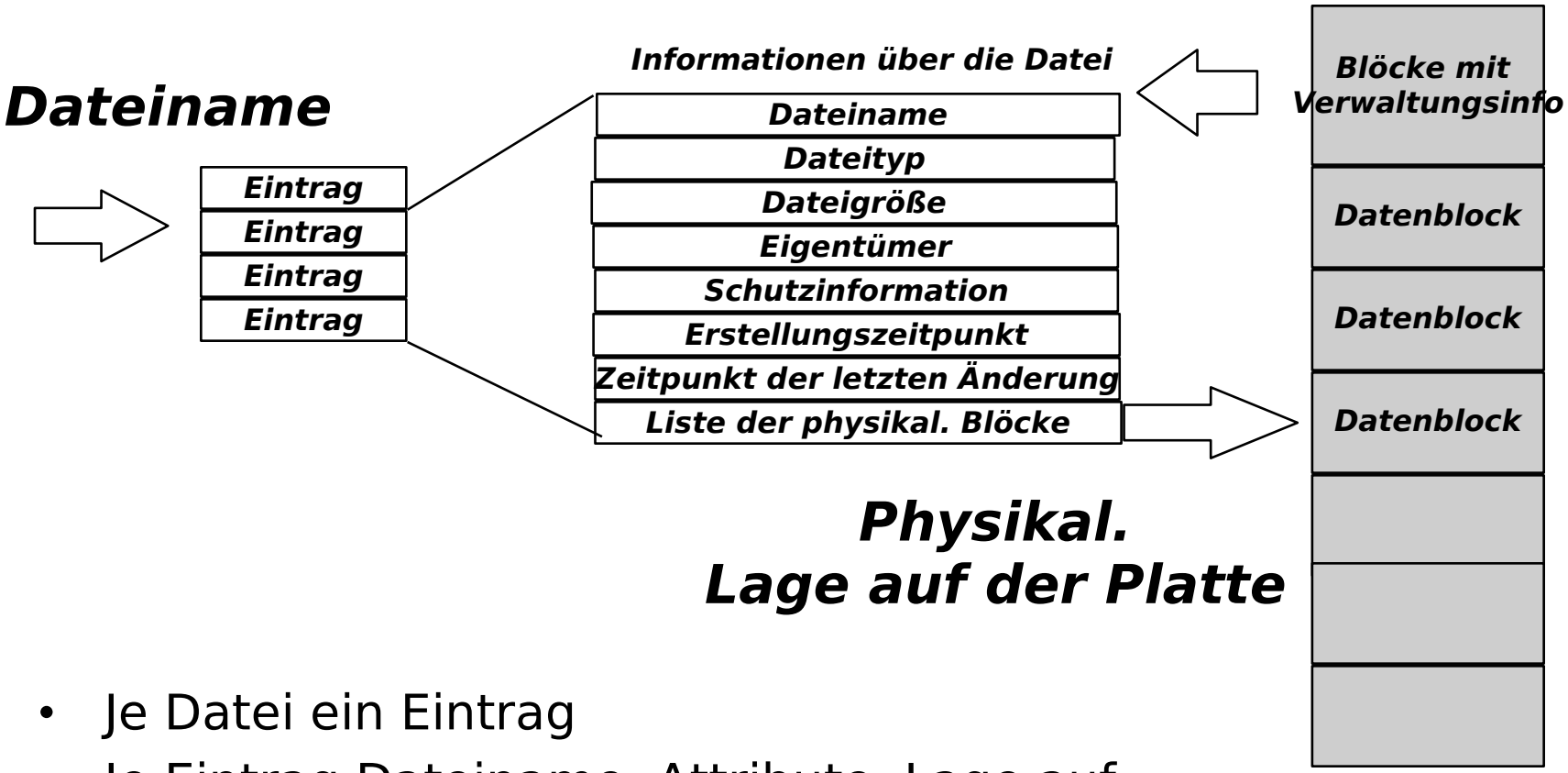
- Zugriff auf Dateinamen durch lineare Suche
- Beschleunigung durch Caching
- Alternativ: Hashing oder Suchbäume für Verzeichnisse mit vielen Einträgen

Unix/Linux



Verwandte Prozesse können eine Datei gemeinsam benutzen.

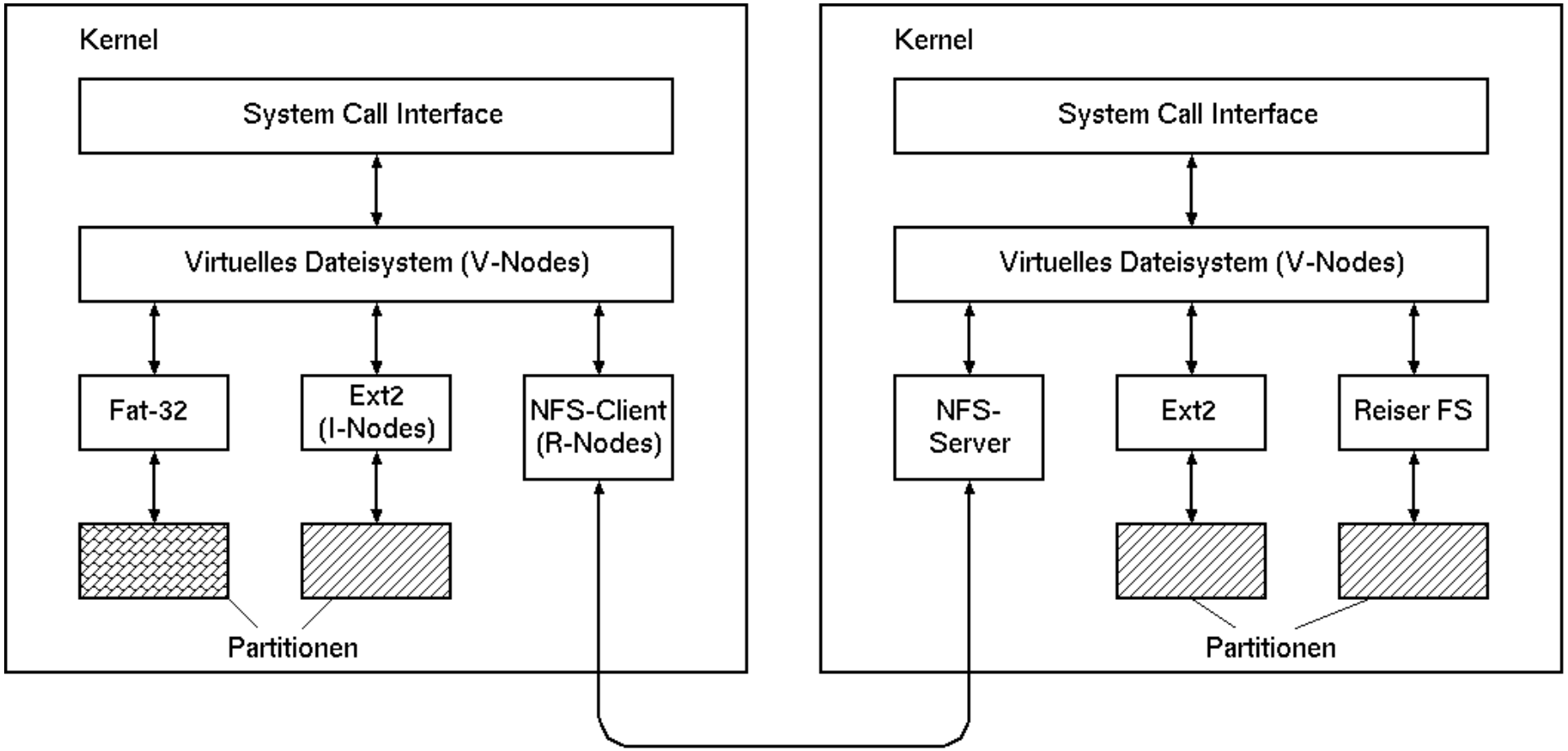
Dateisystem
Aufbau von Verzeichnissen



- Je Datei ein Eintrag
- Je Eintrag Dateiname, Attribute, Lage auf Platte

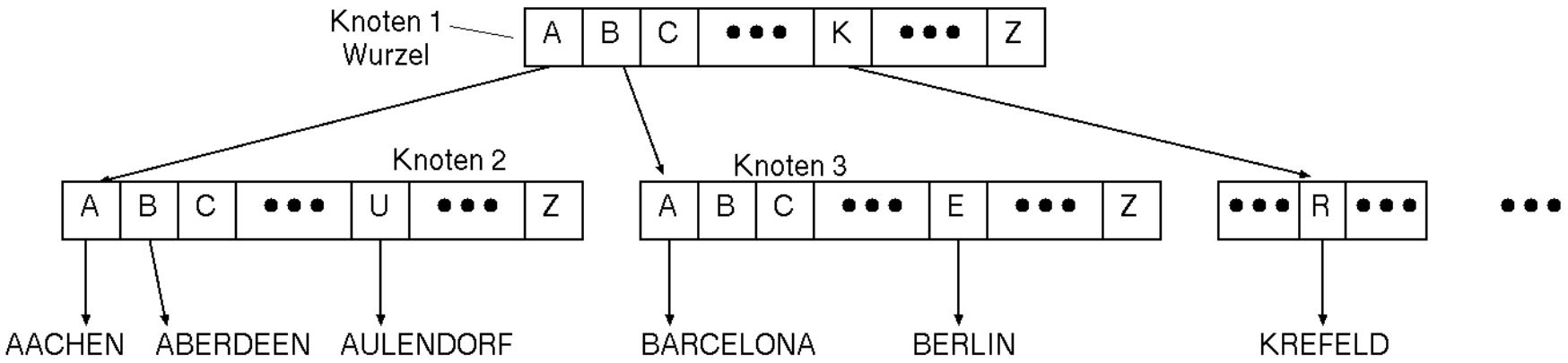
Verteilte Dateisysteme

Integration von Dateisystemen in Linux



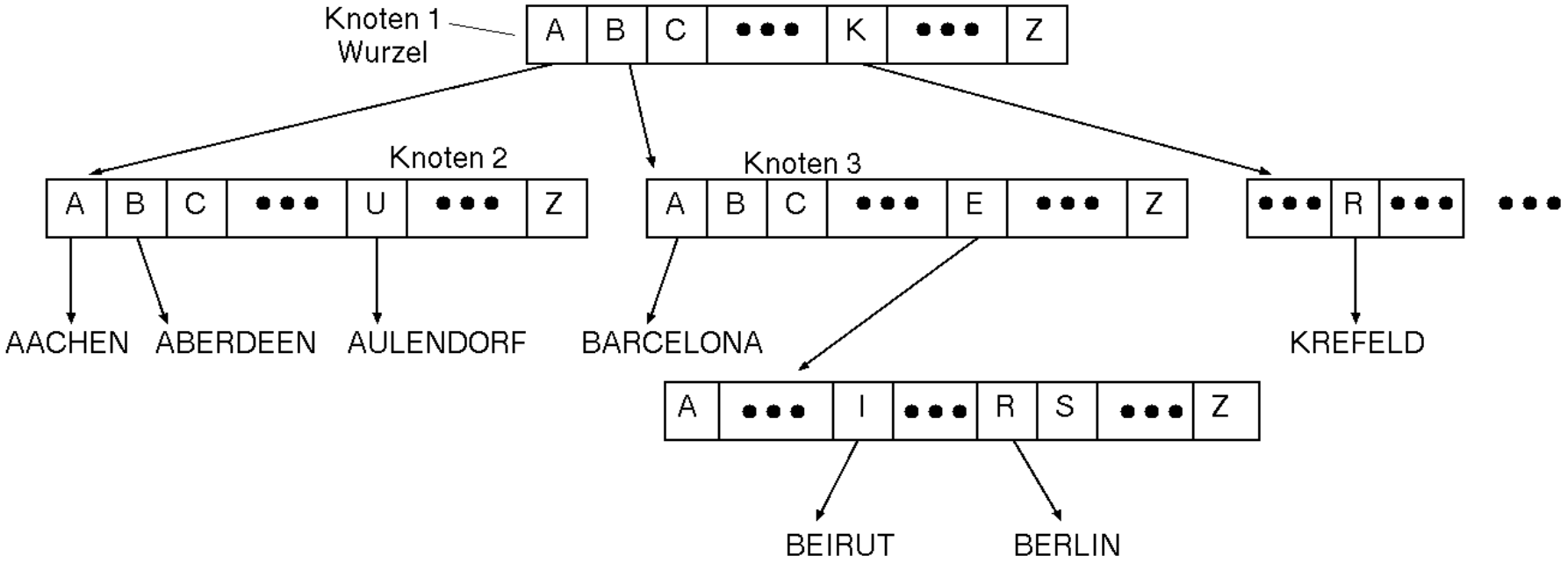
Digitalbaum

- Der Schlüssel wird in Komponenten zerlegt
- Ein Knoten enthält ein Array mit Zeigern auf Kind-Knoten bzw. Objekte
- Eine Komponenten des Schlüssel dient als Index in diese Array (s. z.B. mehrstufige Seitetabelle)

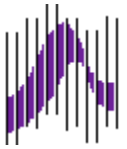


Digitalbaum - Wachstum

- Tritt eine Kollision auf, kann der Baum nach unten wachsen. Auf unterster Ebene: Liste

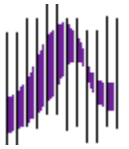


Extendible Hashing



- Objekte werden in einem Digital-Baum gespeichert
- Der Suchschlüssel wird aus dem Objekt-Schlüssel durch eine Hashfunktion errechnet
- Der Digital-Baum ist i.Allg. gut balanciert
- Auf der Ebene der Blätter können i.Allg. Kollisionen auftreten. Diese müssen aufgelöst werden, z.B. duch verkettete Listen.

Zuverlässigkeit von Dateisystemen

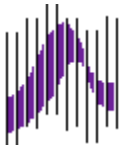


Dateioperationen benötigen oft mehrere Schritte von einem konsistenten Zustand zum nächsten.

Unterbrechungen oder defekte Blöcke können zu Inkonsistenzen führen.

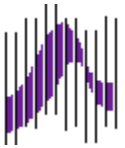
- Regelmäßge Konsistenzprüfung;
  für große Partitionen/Volumes zeitaufwändig
- Journaling zur Beschleunigung der Tests
- Prüfsummen für die Konsistenz von Daten
- Transaktionsmechanismus beim Schreiben
  Copy on Write

Journaling File Systeme

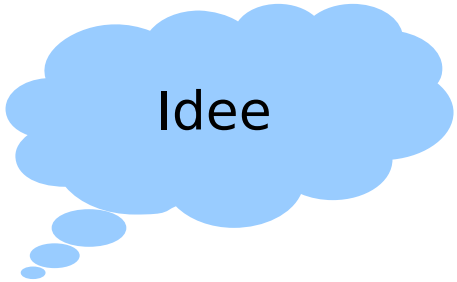


- Um Inkonsistenzen zu entdecken muss das System die gesamte Platte/Partition prüfen.
- Journaling File Systeme
  - geplante Änderungen in das Journal schreiben
  - Änderungen ausführen, dann aus Journal löschen
- Nach einer Unterbrechung muss das System nur das Journal mit dem Ist-Zustand vergleichen.
- Problem: Block des Journals kann defekt sein.

RAID Disks



Redundant Arrays of Inexpensive/Independend Disks



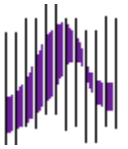
Zusammenfassung mehrerer Festplatten zu einem "Platten-Array"

Für den Anwender erscheint das Array bestehend aus mehreren kleinen Platten als wäre es nur eine grosse Platte

Höhere Geschwindigkeiten durch zeitlich verschränktes Schreiben auf mehrere Platten

Fehlertoleranz

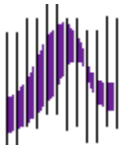
RAID
(Redunant Array of Independend Disks)



- **Hardwarelösungen**
  - Externe RAID-Boxen mit eigenem Controller, Platten, Lüftung und Stromversorgung
  - RAID-Controller im Rechner. Es sind mehrere Platten anschließbar. Die RAID-Levels werden im Contoller realisiert

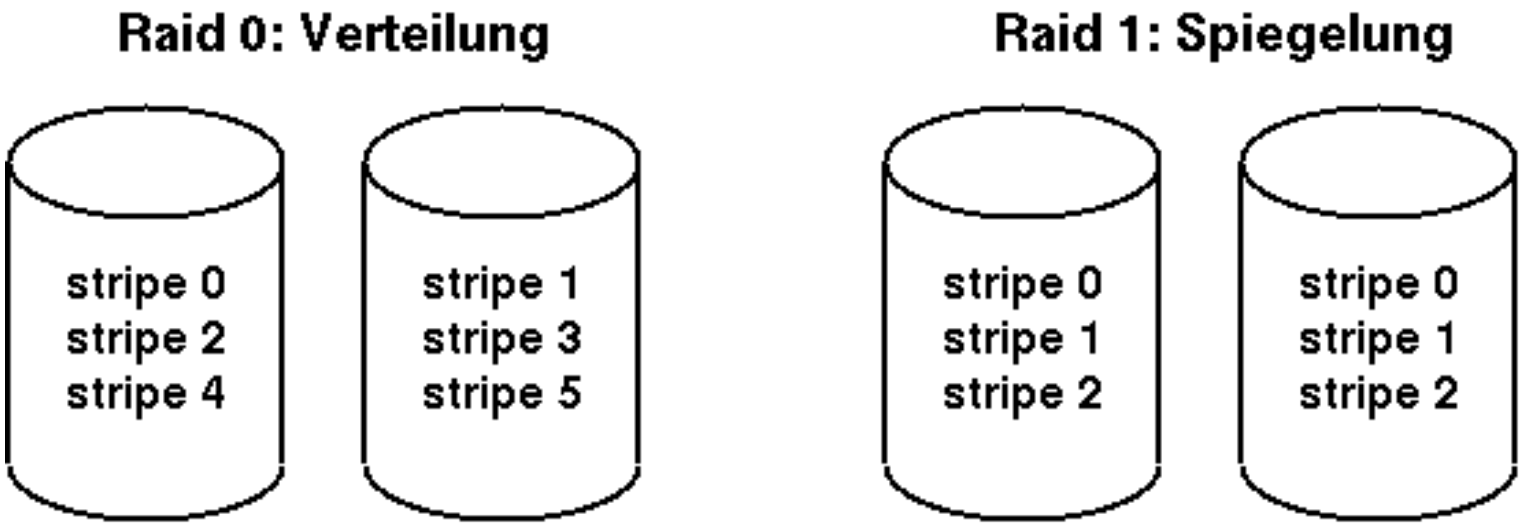
- **Softwarelösungen**
  - Raid-Levels werden im Betriebssystemkern implementiert
    - Windows NT Raid 0,1,4,5
    - LINUX RAID 0,1,4,5

RAID-Level



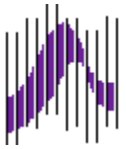
- Raid 0: Dateien werden in „Stripes“ auf mehrere Platten verteilt.
- Raid 1: Wie Raid 0, jeder „Stripe“ wird auf zwei verschiedene Platten geschrieben.
- Raid 2 und 3: Bitweise paralleles Schreiben (kaum verwendet)
- Raid 4: Wie Raid 0 mit einem zusätzlichen „Parity-Stripe“ auf einer zusätzlichen Platte.
- Raid 5: Wie Raid 4, Parity-Stripes werden auf alle Platten verteilt.
- Weitere (Misch-)Formen existieren

RAID Level 0 und 1



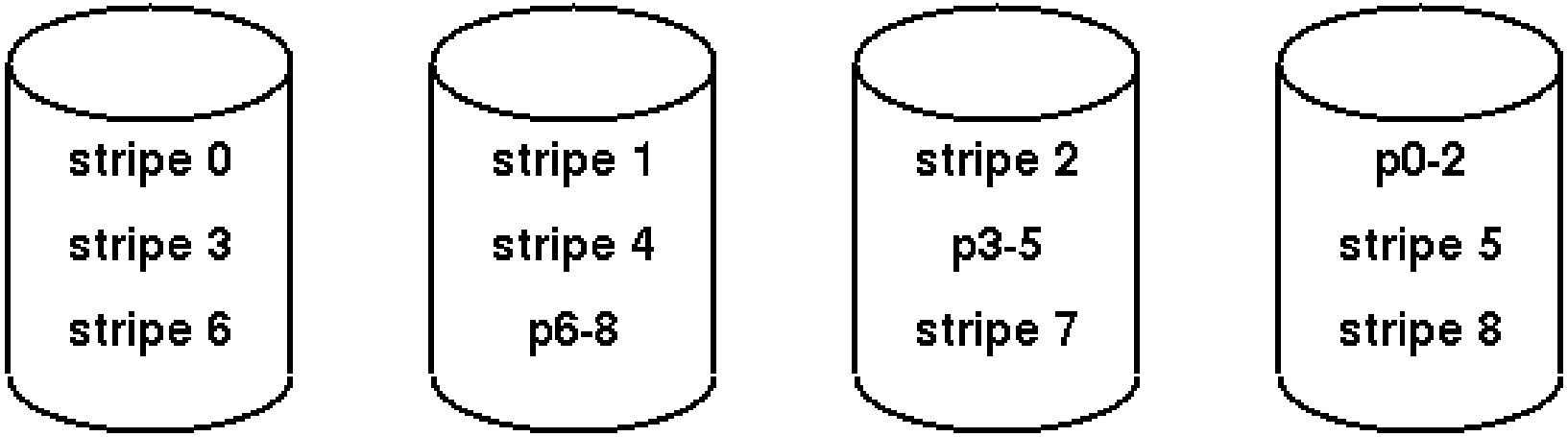
- Raid 0
  - Höherer Durchsatz durch parallelen Zugriff
  - Höhere Fehlerrate: Bei Ausfall einer Platte sind alle Dateien beschädigt
- Raid 1
  - Zuverlässiger durch Spiegelung, benötigt doppelte Speicherkapazität

Raid 10



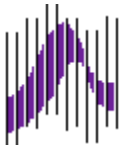
- Kombination von RAID 0 und Raid 1
- Spiegelung von RAID 0 System

Raid Level 5



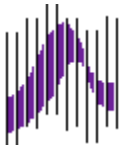
- Bei n Platten wird für n-1 Stripes ein Parity-Stripe auf eine Platte geschrieben
  (XOR Verknüpfung der n-1 Stripes).
- Fällt eine Platte aus, kann der Inhalt aus den verbliebenen Daten rekonstruiert werden.
- Raid 4: Parity-Platte ist Flaschenhals.

RAID 4/5 vs. RAID-Z



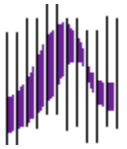
- RAID 4/5: Sicherheitslücke
  - Alle Daten-Stripes und der Parity-Stripe bilden eine Einheit.
  - Stromausfall nach dem ersten und vor dem letzen Stripe führt zu Inkonsistenz. (Pufferung: teuer)
- RAID 4/5: Kurz Schreibzugriffe langsam (partial stripe writes)
  - Bisherige oder restliche Daten lesen, Parity berechnen, neue Stripes und Parity-Stripe schreiben
- RAID-Z (ZFS): RAID an Block-Größe anpassen
  - Jeder Schreibzugriff wird auf Daten- und Parity-Stripe aufgeteilt (full stripe write).
  - Kleine Blöcke werden gespiegelt.

ZFS



- Copy on write
- Aktuelle Daten werden nie überschrieben:
  Kopie schreiben, dann Zeiger umsetzen
- Prüfsumme für Daten und Meta-Daten (Zeiger etc.)
- Zusätzlich Platte „lazy“ in RAID-System integriert

Memory Mapped Files



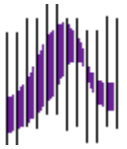
- **Vorteil**: schneller Zugriff auf Daten

- **Nachteil**:
  - schreibende Zugriffe erfolgen im Arbeitspeicher und werden nicht sofort auf Platte übertragen. Bei Programmabsturz sind daher die Daten verloren.

    Abhilfe: Bei kritischen Daten kann ein Schreiben auf Platte erzwungen werden.

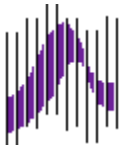
  - Besteht gleichzeitig ein lesender Zugriff auf die Datei können inkonsistente Zustände entstehen, da schreibende Zugriffe nicht direkt auf Platte erfolgen. Damit kann eine Änderung schon erfolgt sein, der lesende Prozess bekommt diese jedoch nicht mit.

Memory Mapped Files



- Benötigte BS-Funktionen:
  - mmap (Datei, Ausschnitt, Schutz, . . .)
    Liefert die Anfangsdresse des eingeblendeten Dateiausschnitts zurück
  - munmap(virtuelle Adresse)
- Entspricht Shared Memory mit persistenter Speicherung

Memory Mapped Files



- Dateien, die entweder vollständig oder in Ausschnitten in den virtuellen Adressraum des Prozesses eingeblendet werden.
  - Damit kann auf die Daten der Datei wahlfrei, wie auf Daten im Arbeitspeicher, zugegriffen werden.

- Bei einem Zugriff auf die virtuelle Seite wird der Inhalt der Datei von Festplatte in den Arbeitsspeicher gelagert. Lesen und Schreiben kann dann über Pointer im Programm ohne Dateifunktionen read() und write() erfolgen. Erst beim Auslagern der Seite aber spätestens beim Terminieren des Prozesses werden alle Daten im Speicher wieder in die Datei auf die Festplatte geschrieben.