Betriebssysteme und nebenläufige Programmierung

SoSe 2025

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. April 2025

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

0 Organisation

Zu meiner Person

- Studium der Informatik an der Techn. Univ. München
  - dort 1994 promoviert, 2001 habilitiert
- Seit 2004 Prof. für Betriebssysteme und verteilte Systeme
- **Forschung:** Sichere komponentenbasierte Systeme; parallele und verteilte Systeme
- Vorsitzender des Prüfungsausschusses Informatik

- **e-mail:** roland.wismueller@uni-siegen.de
- **Tel.:** 0271/740-4050
- **Büro:** H-B 8404

- **Sprechstunde:** Mo., 14:15-15:15 Uhr

Zur Fachgruppe „Betriebssysteme / verteilte Systeme“

Andreas Hoffmann

andreas.hoffmann@uni-...

0271/740-4047

H-B 8405

- El. Prüfungs- und Übungssysteme
- IT-Sicherheit
- Web-Technologien
- Mobile Anwendungen

Felix Breitweiser

felix.breitweiser@uni-...

0271/740-4719

H-B 8406

- Betriebssysteme
- Programmiersprachen
- Virtuelle Maschinen

Sven Jacobs

sven.jacobs@uni-...

0271/740-2533

H-B 8407

- El. Prüfungs- und Übungssysteme
- Generative KI
- Web-Technologien

Lehrangebot

Vorlesungen/Praktika

- Rechnernetze I, 5/6 LP (Bachelor, SoSe)
- Rechnernetze Praktikum, 6 LP (Bachelor, WiSe)
- Rechnernetze II, 6 LP (Master, SoSe)
- Betriebssysteme I, 5/6 LP (Bachelor, SoSe)
- Parallelverarbeitung, 6 LP (Master, WiSe)
- Verteilte Systeme, 6 LP (Bachelor, WiSe)

Lehrangebot ...

Projektgruppen

- z.B. Sichere Kooperation von Software-Komponenten
- z.B. Konzepte zum sicheren Management von Linux-basierten Thin-Clients

Abschlussarbeiten (Bachelor, Master)

- Themengebiete: sichere virtuelle Maschine, Parallelverarbeitung, Mustererkennung in Sensordaten, eAssessment, ...

Seminare

- Themengebiete: IT-Sicherheit, Programmiersprachen, Mustererkennung in Sensordaten, ...
- Ablauf: Blockseminare (30 min. Vortrag, 5000 Worte Paper)
- Master: vorher Vorlesung „Wissenschaftliches Arbeiten“!

Termine

Vorlesung (3 SWS)

- Do., 12:15 - 13:45 Uhr, H-C 3305
- Fr., 10:15 - 11:45 Uhr, H-C 3305, 14-tägig ab 02.05.

Übung (2 SWS)

- Zwei alternative Termine:
  - Mo., 10:15 - 11:45, H-C 6321, ab 14.04.
  - Fr., 12:15 - 13:45, H-D 3206, ab 25.04.
- Für Studierende nach einer FPO 2019 oder neuer:
  - Abgabe einiger Übungsaufgaben ist Pflicht (☞ **Folie 10**)

Materialien

Information, Folien und Ankündigungen

- **Auf der Vorlesungs-Webseite:**

  `http://www.bs.informatik.uni-siegen.de/lehre/bs1`

- Vollständiger Foliensatz ist verfügbar
- Ggf. Aktualisierungen/Ergänzungen kurz vor der Vorlesung
  - auf das Datum achten!

- Es gibt zusätzlich einen Moodle Kurs
  - Vorlesungs-Aufzeichnungen aus dem WiSe 2019/20(!)
  - Aufzeichnungen einiger der neuen Themen
  - Abgabe der Lösungen zu den Übungen

Zur Übung

- Teilweise praktische Übungen mit Linux
  - Bearbeitung zu Hause!
  - Installieren Sie sich dazu eine Linux-Variante oder nutzen Sie ein Linux Live-System auf USB-Stick, z.B. Ubuntu
    - siehe `https://ubuntu.com/download/desktop`
  - Sollte dies nicht möglich sein, können wir Ihnen auch einen Zugang auf ein Linux-System einrichten
    - bitte melden Sie sich dazu per E-Mail (von Ihrer studentischen Uni-Adresse aus!) bei Prof. Wismüller
- In den Übungsstunden sollen vor allem Ihre Fragen beantwortet bzw. Ihre Lösungsideen vorgestellt werden!
  - Musterlösungen werden nach der Übung als PDF auf die Webseite gestellt

Studienleistung

- Vorgeschrieben in den Studiengängen:
  - Bachelor Informatik nach FPO-B 2021 (inkl. dual + Lehramt)
  - Master Wirtschaftsinformatik nach FPO-M 2019
  - Bachelor/Master Mathematik nach FPO-B/FPO-M 2021
  - Bachelor DBHS
  - ggf. weitere (bitte selbst informieren!)
- Bearbeitung von $\geq$ 4 von 6 praktischen Programmieraufgaben
  - Abgabe und Feedback über Moodle
  - voraussichtlich ab Übung 3
- **Nicht vergessen: Anmeldung in unisono**
  - 4INFBA011-S - Studienleistung Betriebssysteme und nebenläufige Programmierung
  - **ohne Anmeldung** ist **keine Abgabe** der Aufgaben möglich!

Prüfung

- 60-minütige schriftliche Klausur
  - ohne Hilfsmittel
- Anmeldung zur Klausur über unisono
  - Bachelor Informatik: **bis 14 Tage vor der Prüfung**
    - siehe Webseite des Prüfungsamts!
  - Informatik, PO 2012: Mentorengenehmigung erforderlich
    - **Frist: 07. Juli**
  - andere Studiengänge: bitte selbst informieren!
- Voraussichtliche(!) Klausurermine:
  - SoSe: Do., 28.08.2025 (ohne Gewähr!)
  - WiSe: Do., 12.03.2025 (ohne Gewähr!)
- Zeit / Ort wird noch verbindlich bekanntgegeben (unisono!)

Literatur

- Andrew S. Tanenbaum, Herbert Bos. *Moderne Betriebssysteme, 4. Auflage*. Pearson Studium, 2016.
- William Stallings. *Betriebssysteme, 4. Auflage*. Pearson Studium, 2003.
- William Stallings. *Operating Systems – Internals and Design Principles, 8. Auflage*. Pearson Education, 2015.
- Jürgen Nehmer, Peter Sturm. *Systemsoftware – Grundlagen moderner Betriebssysteme, 2. Auflage*. dpunkt.verlag, 2001.
- E. Ehses, L. Köhler, P. Riemer, H. Stenzel, F. Victor. *Betriebssysteme – Ein Lehrbuch mit Übungen zur Systemprogrammierung in UNIX/Linux*. Pearson Studium, 2005.

Inhalt der Vorlesung

- Einführung
  - was ist ein Betriebssystem (BS), wozu braucht man es?
- Prozesse und Threads
  - was sind Prozesse/Threads, wie werden sie verwaltet?
- Interprozeßkommunikation (IPC)
  - wie können Prozesse/Threads kooperieren?
  - was kann dabei schiefgehen (Verklemmungen)?
- Nebenläufige und asynchrone Programmierung
  - wie programmiert man nebenläufige Aktivitäten?
  - wie reagiert man auf asynchrone Ereignisse?
- Scheduling
  - wer darf wann wie lang rechnen?

Inhalt der Vorlesung ...

- Speicherverwaltung
  - wie teilt das Betriebssystem den Speicher an Prozesse zu?
- Ein-/Ausgabe
  - wie kommuniziert der Rechner mit externen Geräten
- Dateisysteme
  - wie wird z.B. die Festplatte verwaltet?
- Schutzmechanismen
  - wie werden Benutzer gegeneinander geschützt?
- Virtualisierung
  - wie funktionieren virtuelle Maschinen?

Ziel der Vorlesung

- Grundwissen jedes Informatikers in den Bereichen
  - Betriebssysteme
  - nebenläufige Programmierung
- Verständnis der Probleme und ihrer Lösungen
- Grundverständnis gängiger BS-Konzepte und -Mechanismen
  - wichtig für (effiziente) Programmierung!
  - Konzepte oft auch für Anwendungsprogramme nutzbar
- Grundlage für andere Vorlesungen
  - Verteilte Systeme (WiSe)
  - Parallelverarbeitung (WiSe)
  - ...

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

1 Einführung

1 Einführung ...

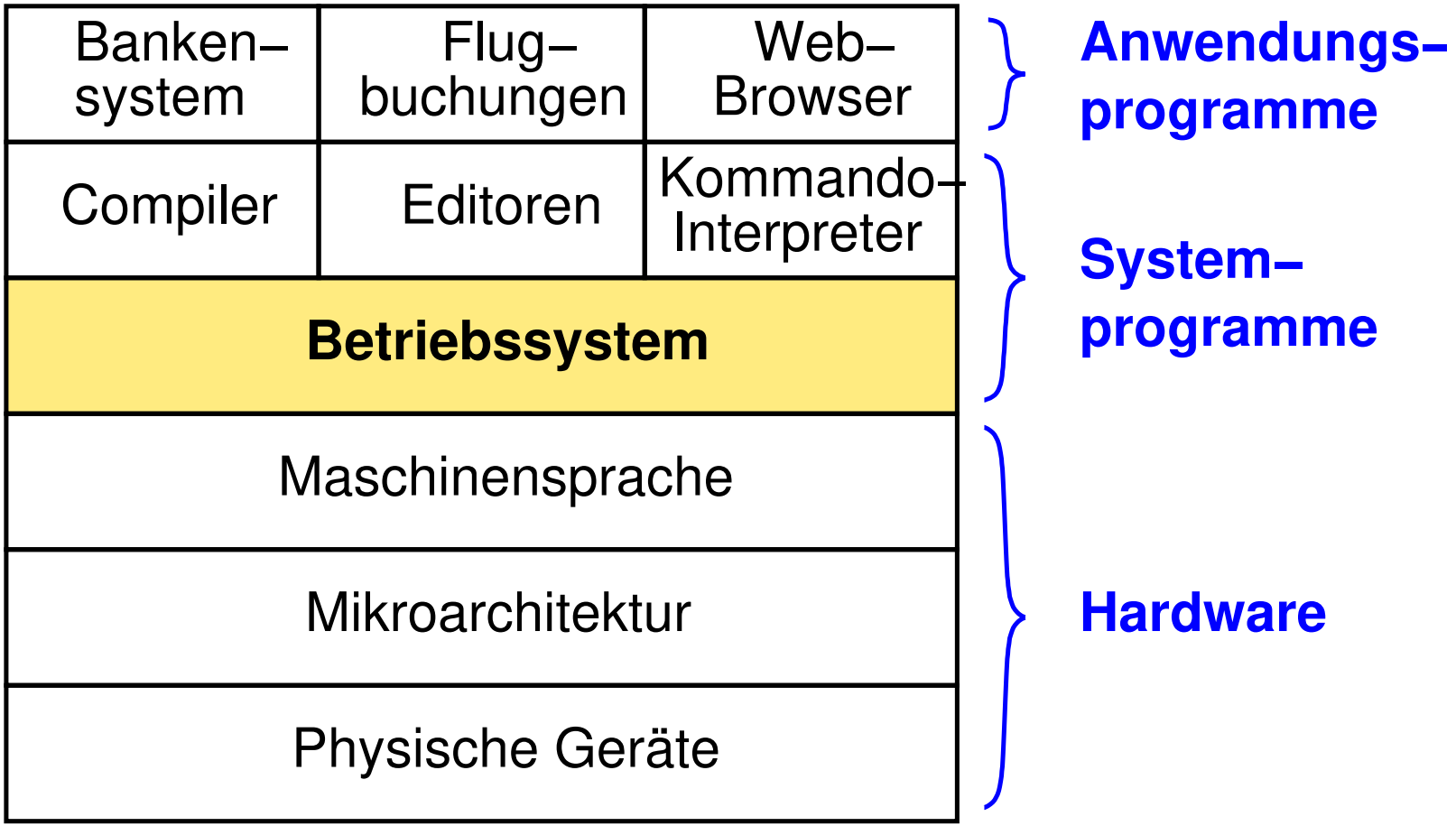
- Aufgaben eines Betriebssystems
- Historische Entwicklung der Betriebssysteme
- Arten von Betriebssystemen
- Überblick Computer-Hardware
- Grundlegende Betriebssystemkonzepte
- Systemaufrufe, Dienste eines Betriebssystems

- Tanenbaum 1.1 - 1.4.3, 1.5, 1.6

Ein Rechensystem besteht aus:

- Anwendungssoftware: zur Lösung bestimmter Probleme, z.B.:
  - Textverarbeitung, Adreßbuch, Datenbank, ...
  - WWW-Browser, Spiele, ...
  - Wettervorhersage, CAD, ...
  - Steuerung eines Kraftwerks, ...
- Systemsoftware: unterstützt Anwendungen, z.B.:
  - Übersetzer: erstellt Maschinenprogramme
  - Betriebssystem: unterstützt **laufende** Anwendungen
  - Dateimanager (z.B. Windows-Explorer), ...
- Hardware

Einordnung des Betriebssystems



Was soll ein Betriebssystem leisten?

- **Erweiterung (Veredelung) der Hardware**
- **Abstraktion der Hardware**
- **Verwaltung von Betriebsmitteln**

Abkürzung: **BS** = Betriebssystem

Erweiterung / Veredelung der Hardware

- Hardware muß billig, schnell und zuverlässig sein
  - Beschränkung auf das absolut notwendige
  - Folge: schwierige Programmierung
- BS stellt komplexe Funktionen bereit, die die Anwendungsprogramme verwenden können
  - Beispiel: Schreiben auf Festplatte
    - BS findet automatisch freie Blöcke auf der Platte, legt Verwaltungsinformation an
    - interne Struktur der Platte (Anzahl Köpfe, Zylinder, Sektoren, etc.) für Anwendung nicht mehr wichtig
  - Folge: erhebliche Vereinfachung der Programmierung

Abstraktion der Hardware

- Rechner sind trotz ähnlicher Architektur im Detail sehr unterschiedlich, z.B.:
  - Einteilung des Adreßraums (Speicher, E/A-Controller)
  - verschiedenste E/A-Controller und Geräte
- Fallunterscheidung wird vom BS vorgenommen
- BS realisiert einheitliche Sicht für Anwendungen
- Beispiel: Dateien abstrahieren Externspeicher
  - für Anwendungen kein Unterschied zwischen Festplatte, Diskette, CD-ROM, USB-Stick, Netzlaufwerk, ...
  - UNIX: selbst Drucker etc. wie Dateien behandelt
- BS realisiert eine **virtuelle Maschine**

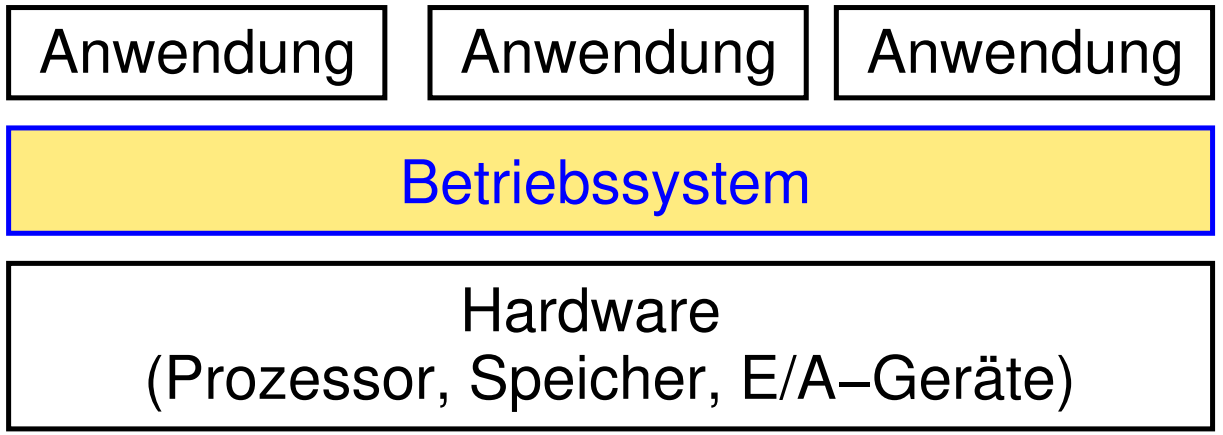
Verwaltung von Betriebsmitteln

- **Betriebsmittel** (**Ressourcen**): alles was eine Anwendung zur Ausführung braucht
  - Prozessor, Speicher, Geräte (Festplatte, Netzwerk, ...)
- Früher: auf einem Rechner lief zu jedem Zeitpunkt nur eine Anwendung eines Benutzers
- Heute: Rechner im **Mehrprozeß-** und **Mehrbenutzerbetrieb**
  - mehrere Anwendungen verschiedener Benutzer werden „gleichzeitig“ ausgeführt
- Notwendig:
  - Fairness: „gerechte“ Verteilung der Betriebsmittel
  - Sicherheit: Schutz der Anwendungen und Benutzer voreinander

Verwaltung von Betriebsmitteln ...

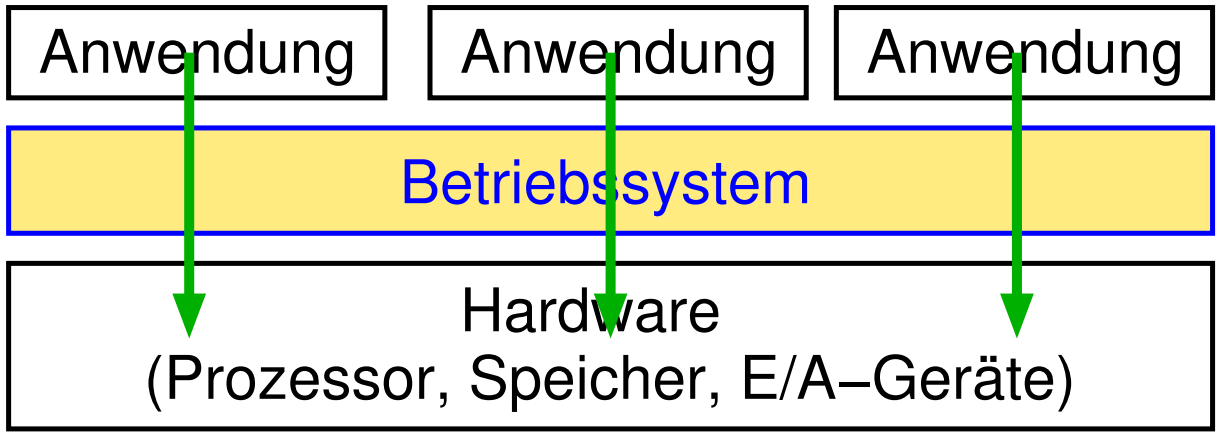
- Beispiel: Dateien
  - jeder Datei werden **Rechte** zugeordnet
    - legen z.B. fest, wer die Datei lesen darf
  - BS stellt die Einhaltung dieser Rechte sicher
    - unbefugter Zugriff wird verweigert
- Beispiel: Drucker
  - während Max druckt, will auch Moritz drucken
    - aber nicht auf dasselbe Blatt Papier ...
  - BS regelt den Zugriff auf den Drucker
    - der Auftrag von Moritz wird zurückgestellt, bis der von Max beendet ist

BS als Mittler zwischen Anwendungen und Hardware



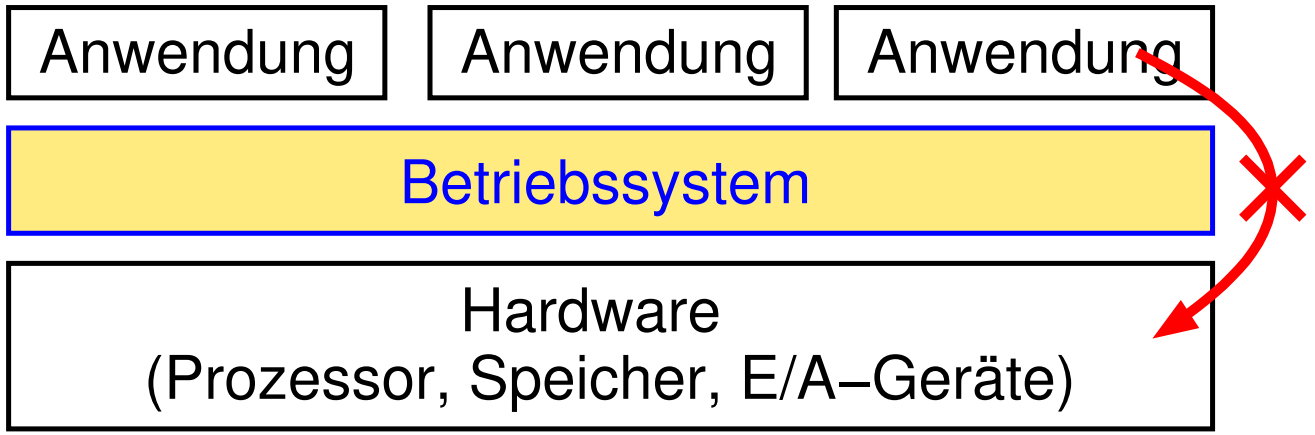
- Essentiell: Anwendungen können nicht direkt (d.h. **unkontrolliert**) auf die Hardware zugreifen
- Unterstützende Hardware-Mechanismen:
  - Ausführungsmodi des Prozessors (System- und Benutzermodus, ☞ **1.4.1**)
  - Adreßumsetzung (virtueller Speicher, ☞ **8.3**)

BS als Mittler zwischen Anwendungen und Hardware



- Essentiell: Anwendungen können nicht direkt (d.h. **unkontrolliert**) auf die Hardware zugreifen
- Unterstützende Hardware-Mechanismen:
  - Ausführungsmodi des Prozessors (System- und Benutzermodus, ☞ **1.4.1**)
  - Adreßumsetzung (virtueller Speicher, ☞ **8.3**)

BS als Mittler zwischen Anwendungen und Hardware



- Essentiell: Anwendungen können nicht direkt (d.h. **unkontrolliert**) auf die Hardware zugreifen
- Unterstützende Hardware-Mechanismen:
  - Ausführungsmodi des Prozessors (System- und Benutzermodus, ☞ **1.4.1**)
  - Adreßumsetzung (virtueller Speicher, ☞ **8.3**)

1. Generation (-1955): kein Betriebssystem

- Programm (jedesmal) manuell in Speicher eingeben

2. Generation (-1965): Stapelverarbeitung

- Lochkarten mit Programmcode (z.B. Assembler, Fortran)
- BS startet Übersetzer und Programm
- BS nimmt Ergebnis entgegen, gibt es auf Drucker aus
- später: auch mehrere Programme (**Jobs**) nacheinander (auf Magnetband): **Stapelbetrieb** (*batch*)
- Stapelbetrieb auch heute noch teilweise sinnvoll
  - lange, nicht-interaktive Jobs (z.B. Jahresabrechnungen)

3. Generation (-1980):

- Rechnerfamilien mit gleichem Befehlssatz (z.B. IBM 360)
  - BS abstrahiert Unterschiede der Rechner / Geräte
- Einführung des Mehrprogrammbetriebs
  - CPU wartet oft (bis zu 90% der Zeit) auf Geräte: Verschwendung!
  - besser: statt zu warten wird ein anderer Job bearbeitet
    - Problem: Verwaltung / Zuteilung der Betriebsmittel
- Gleichzeitig: interaktive Nutzung der Rechner
  - Terminals statt Lochkarten und Drucker
  - mehrere Benutzer gleichzeitig aktiv
    - gegenseitiger Schutz erforderlich

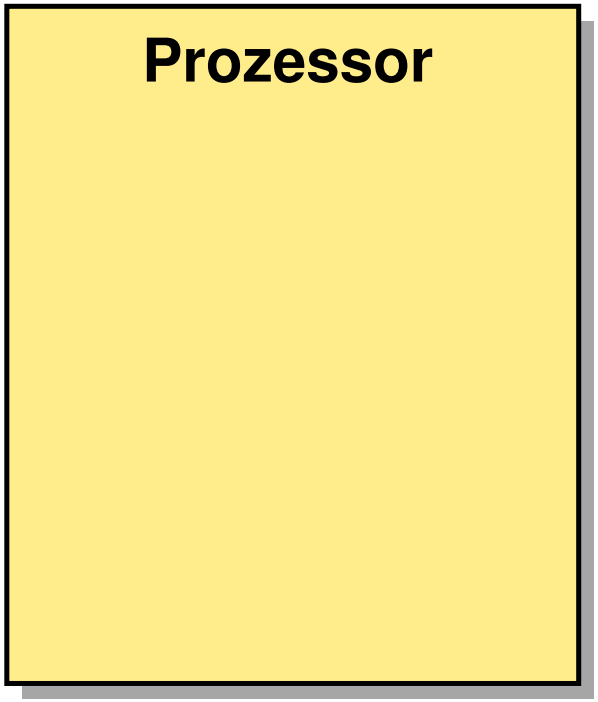
4. Generation (1980 - heute):

- Einführung von Mikroprozessoren
  - kleine, billige Rechner: Arbeitsplatzrechner
  - zurück zu Einbenutzersystemen (DOS, Windows 95, ...)
- Zunehmende Vernetzung der Rechner
  - Client/Server-Systeme: wieder mehrere Benutzer
  - Unix, Linux, Windows (ab NT), ...
- Trend / Zukunft: verteilte Betriebssysteme
  - mehrere Rechner erscheinen wie ein einziger
  - Ziele: höhere Leistungsfähigkeit und Zuverlässigkeit

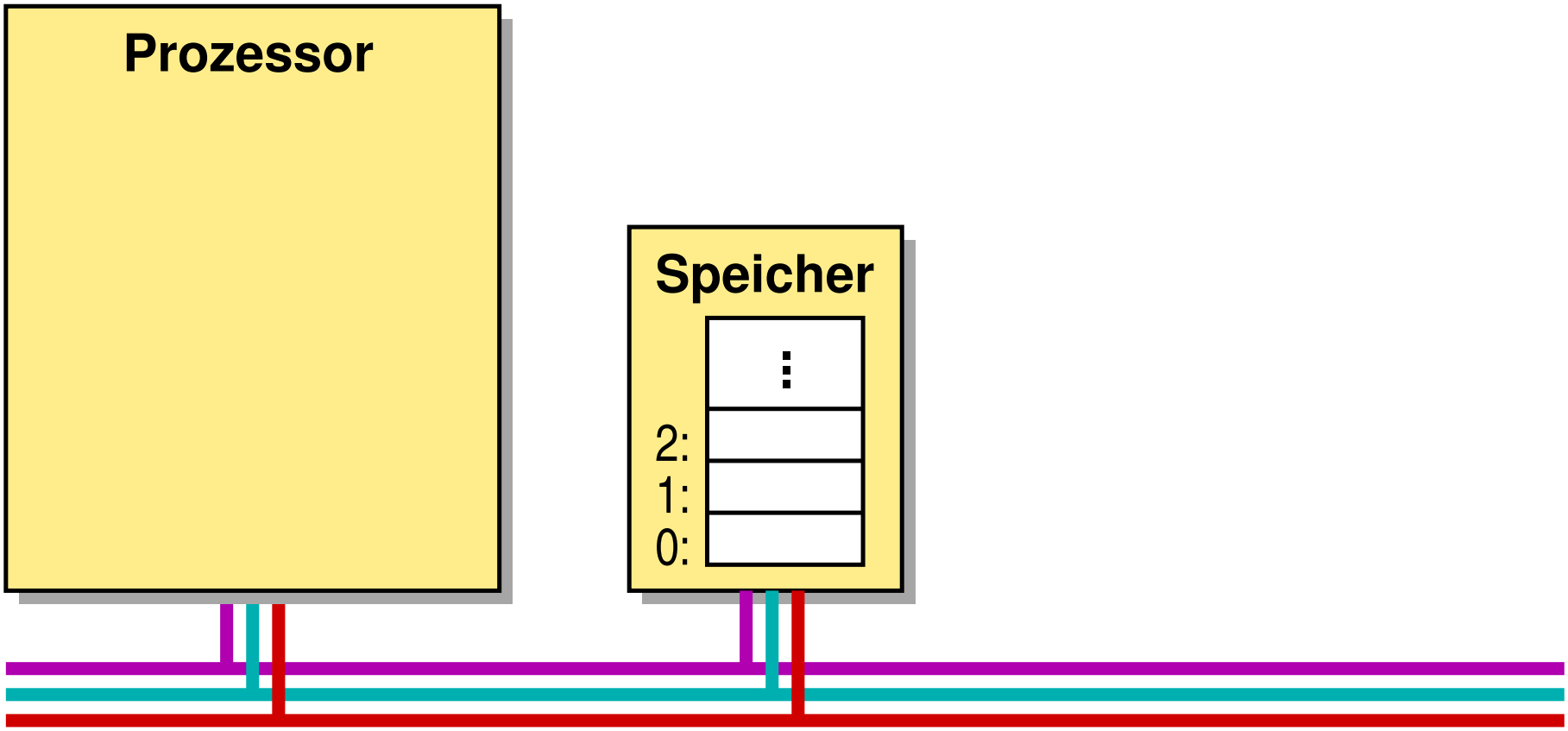
1.3 Arten von Betriebssystemen

- Mainframe-BSe
  - schnelle E/A, viele Prozesse, Transaktionen
- **Server-BSe**
  - viele Benutzer gleichzeitig, Netzwerkanbindung
- Multiprozessor-BSe
  - für Parallelrechner
- **PC-BSe**
- Echtzeit-BSe
- BSe für eingebettete Systeme
- BSe für Chipkarten

Aufbau eines typischen PCs (stark vereinfacht)

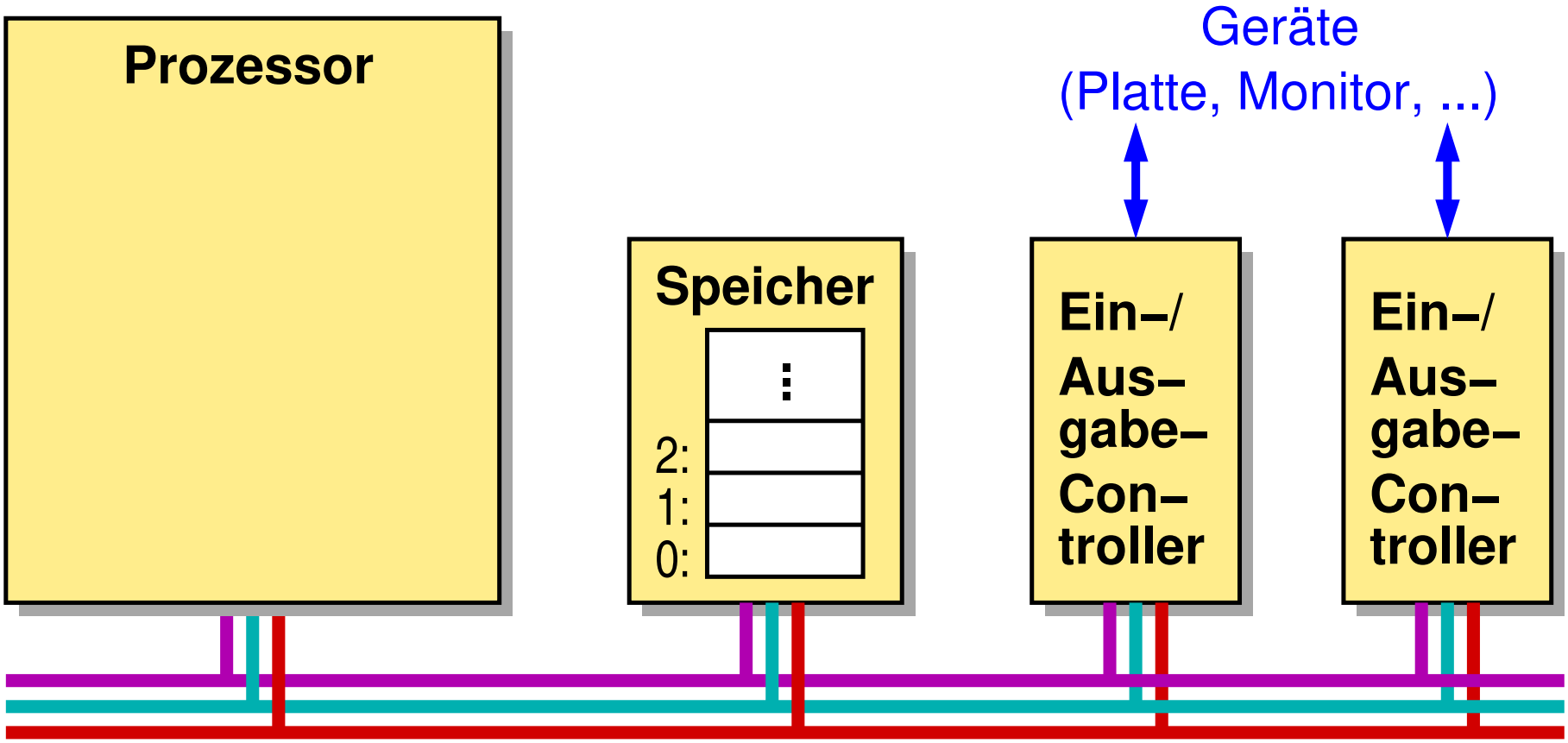


Aufbau eines typischen PCs (stark vereinfacht)



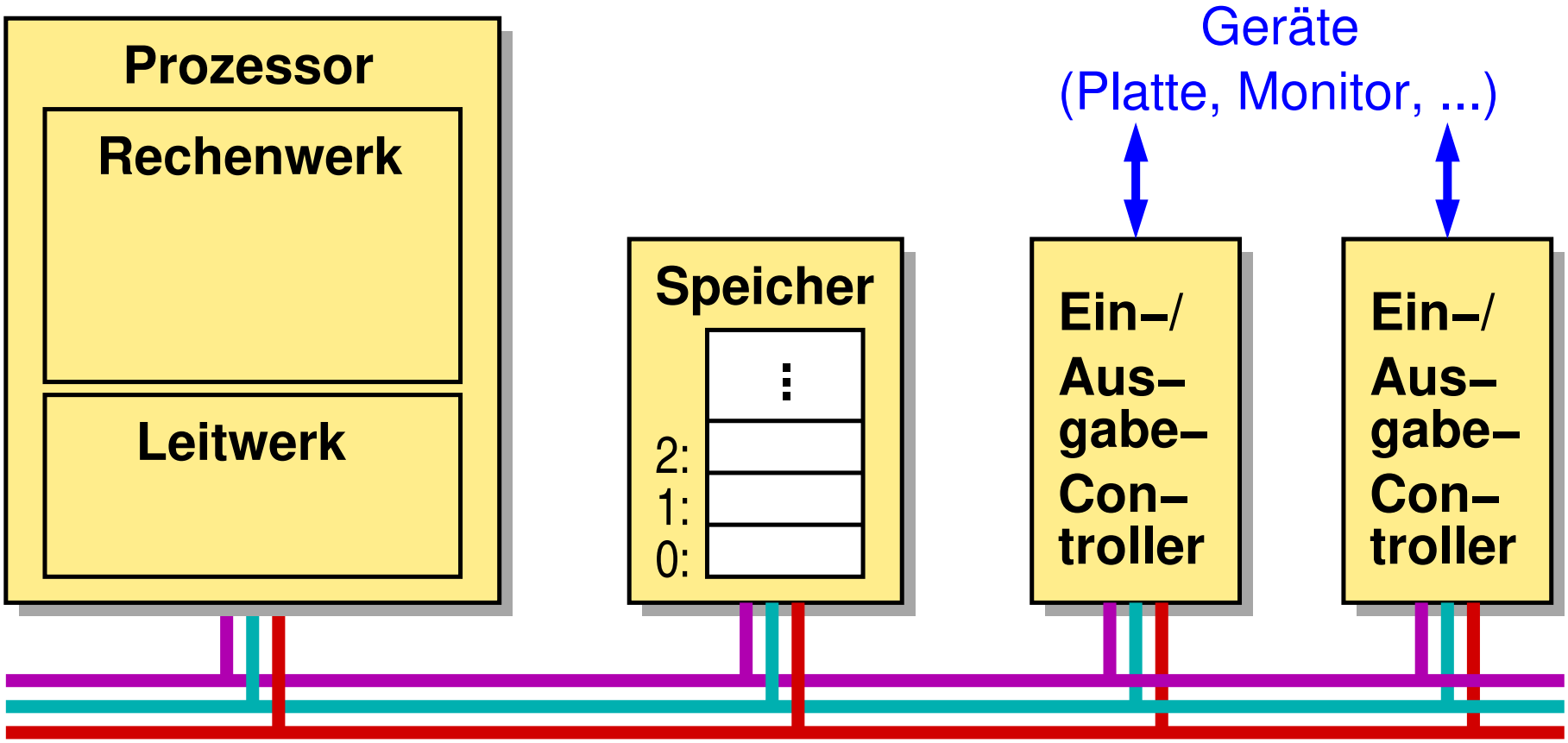
Systembus: Adressen, Daten, Steuersignale

Aufbau eines typischen PCs (stark vereinfacht)



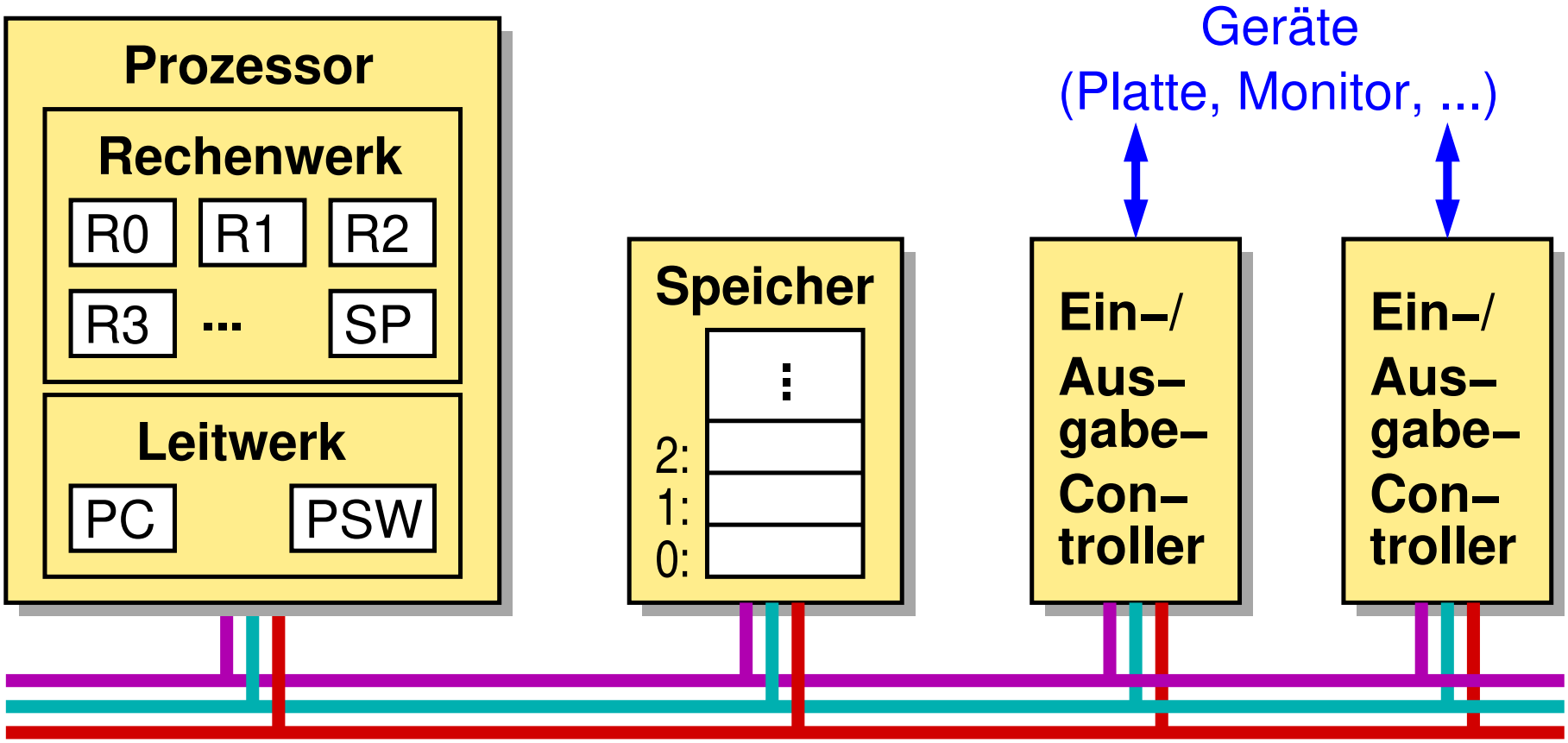
Systembus: Adressen, Daten, Steuersignale

Aufbau eines typischen PCs (stark vereinfacht)



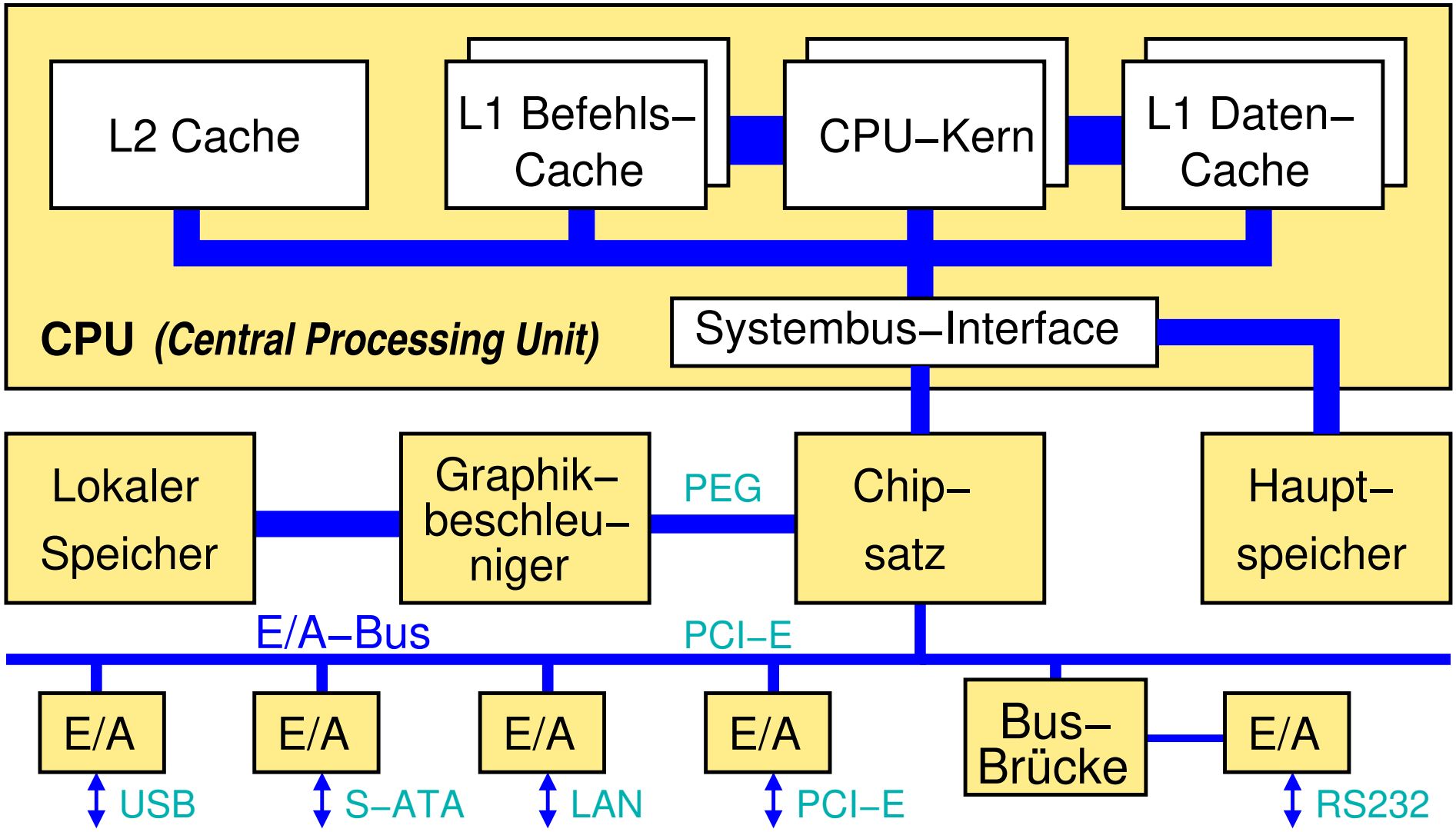
Systembus: Adressen, Daten, Steuersignale

Aufbau eines typischen PCs (stark vereinfacht)



Systembus: Adressen, Daten, Steuersignale

Aufbau eines typischen PCs (realistischer)



Multiprozessor-Systeme

- Heute i.a. Rechner mit mehreren CPUs (bzw. CPU-Kernen)
  - Multicore-Prozessoren
  - Server mit mehreren Prozessoren
- Im Folgenden einheitlich als Multiprozessor-Systeme bezeichnet
  - Begriff „CPU“ bzw. „Prozessor“ bezeichnet ggf. nur einen CPU-Kern
- Typische BS-Architektur für Multiprozessorsysteme:
  - im Speicher: eine Instanz des BSs für alls Prozessoren
  - jeder Prozessor kann Code des BSs ausführen
  - **symmetrisches Multiprozessor-System**

Elemente einer CPU

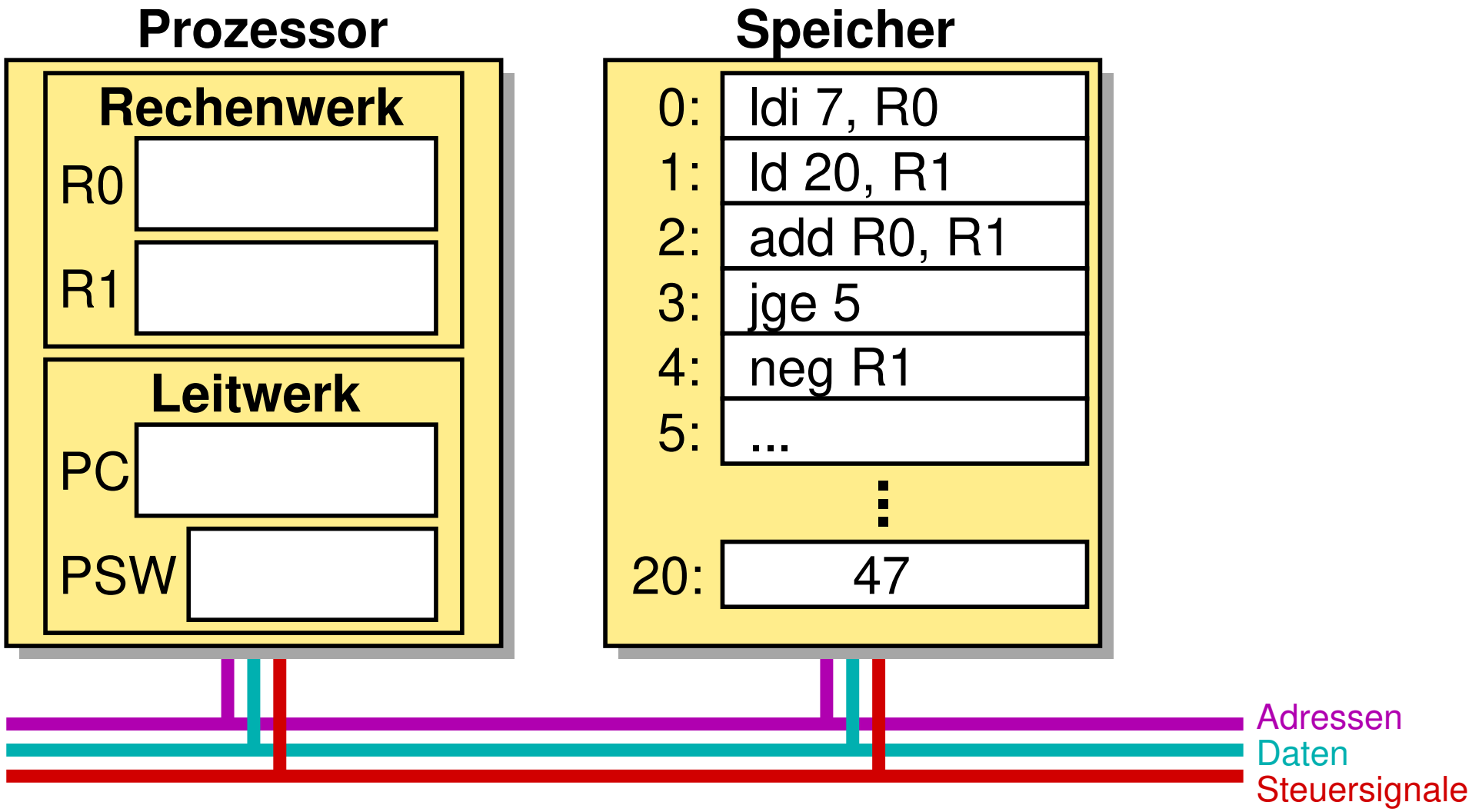
- Register
  - Ganzzahl-, Gleitkomma-Register
  - Befehlszähler (PC: *Program Counter*)
  - Kellerzeiger (SP: *Stack Pointer*)
  - Statusregister (PSW: *Program Status Word*)
- Arithmetisch-Logische Einheit (Rechenwerk, ALU)
  - führt die eigentlichen Berechnungen durch
- Steuerwerk
  - holt und interpretiert Maschinenbefehle

1.4.1 Prozessor (CPU) ...

Operationen:

- Lade- und Speicheroperationen
- Arithmetische und logische Operationen
- Sprünge (Änderung des Befehlszählers)
  - bedingt und unbedingt
  - Unterprogramm-Aufrufe und -Rücksprünge

Beispiel zur Befehlsausführung



Ausführungsmodi

- Eine der Maßnahmen, um den direkten Zugriff auf Systemressourcen durch Anwendungsprogramme zu unterbinden
- **Systemmodus**: ausgeführtes Programm hat vollen Zugriff auf alle Rechnerkomponenten
  - für das Betriebssystem
- **Benutzermodus**: eingeschränkter Zugriff
  - keine **privilegierten Befehle**
    - z.B. Ein-/Ausgabe, Zugriff auf Konfigurationsregister
  - Speicher nur über Adreßabbildung zugreifbar (☞ **1.5.2, 8.3**)

Ausführungsmodi ...

- Ausführung eines privilegierten Befehls im Benutzermodus führt zu **Ausnahme** (*Exception, Software Interrupt*)
  - Prozessor bricht gerade ausgeführten Befehl ab
  - Prozessor sichert PC (Adresse des Befehls) im Keller (Rückkehradresse)
  - Programmausführung wird an **vordefinierter** Adresse im Systemmodus fortgesetzt
    - d.h. Verzweigung an eine feste Stelle im BS

  Hardware

  - BS behandelt die Ausnahme
    - z.B. Abbruch des laufenden Prozesses
    - ggf. auch andere Behandlung und Rückkehr in die Anwendung
  - Rückkehrbefehl schaltet wieder in Benutzermodus

  BS

Ausführungsmodi ...

- Kontrollierter Moduswechsel
  - spezieller Befehl (**Systemaufruf**, *Trap, System Call*)
  - bei Ausführung des Befehls:
    - Prozessor sichert PC im Keller (Rückkehradresse)
    - Umschalten in Systemmodus
    - Verzweigung an eine **vordefinierte** Adresse (im BS)
  - BS analysiert die Art des Systemaufrufs und führt den Aufruf aus
  - Rückkehrbefehl schaltet wieder in Benutzermodus

Hardware

BS

Interrupts

- Prozessor kann auf externe, asynchrone Ereignisse reagieren
  - spezielles Eingangssignal: Unterbrechungssignal
  - Signal wird jeweils nach der Abarbeitung eines Befehls abgefragt
  - falls Signal gesetzt:
    - Prozessor sichert PC im Keller (Rückkehradresse)
    - Umschalten in Systemmodus
    - Verzweigung an eine **vordefinierte** Adresse (im BS) (**Unterbrechungsbehandlungsroutine**, *interrupt handler*)
    - im BS: Behandlung der Unterbrechung
    - Rückkehrbefehl schaltet wieder in Benutzermodus
- Hauptanwendung: Ein-/Ausgabe

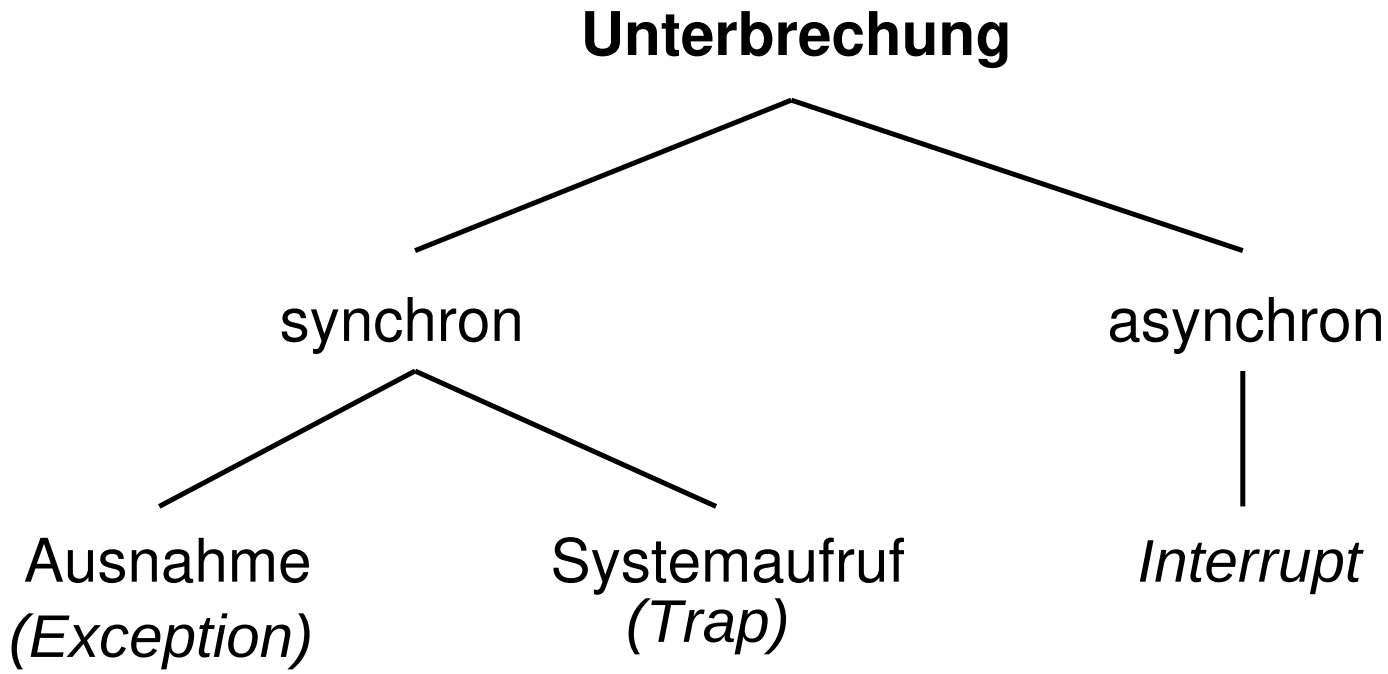
Hardware

BS

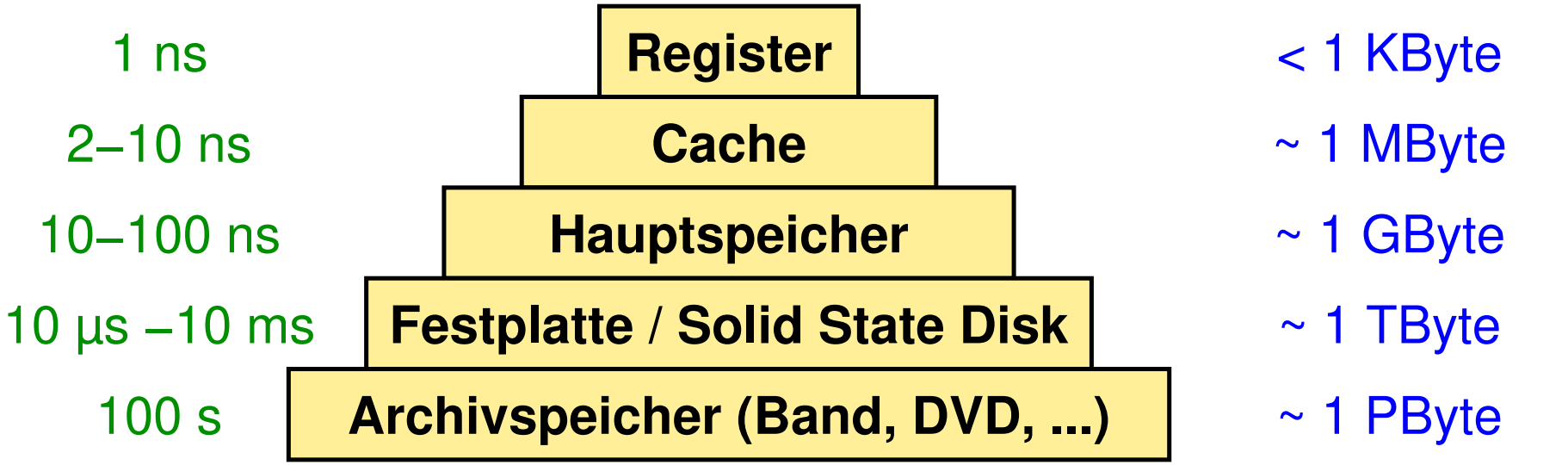
Interrupts ...

- I.d.R. mehrere Interrupts mit verschiedenen Prioritäten
  - jedem Interrupt kann eine eigene Behandlungsroutine zugewiesen werden
    - Tabelle von Adressen: **Unterbrechungsvektor**
  - ggf. Unterbrechung aktiver Behandlungsroutinen
- Interrupts können **maskiert** („ausgeschaltet“) werden
  - privilegierter Befehl!

Unterbrechungen des Programmablaufs



Speicherhierarchie



➥ Verwaltung von Hauptspeicher und Festplatte durch BS

Caches in Multiprozessorsystemen (incl. Multicore)

- **Cache**: schneller, prozessornaher Zwischenspeicher
  - speichert Kopien der zuletzt am häufigsten benutzten Daten aus dem Hauptspeicher
    - i.a. Blöcke (Cachezeilen) mit 32-64 Byte
  - falls Daten im Cache: kein Hauptspeicherzugriff nötig
  - durch Hardware verwaltet, für Programme transparent
- Caches sind in Multiprozessorsystemen essentiell
  - Cachezugriff 10-1000 mal schneller als Hauptspeicherzugriff
  - Entlastung von Hauptspeicher und Bus
- Konsistenz der Inhalte der einzelnen Caches durch Hardware sichergestellt (Cache-Kohärenz-Protokolle)

Ansteuerung der Geräte durch *Controller*

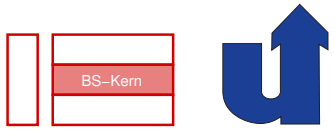
- Spezielle Hardware, oft mit eigenen Mikroprozessoren
- Steuert Gerät weitgehend autonom
- Register für Kommandos, Daten, Status
- Kann Interrupt an CPU senden, falls
  - Eingabedaten vorhanden
  - Ausgabeoperation abgeschlossen

Anbindung Controller - CPU

- Speicherbasierte E/A
  - Register des Controllers sind in den Speicheradreßraum eingeblendet
  - Zugriff über normale Schreib-/Lesebefehle
  - Zugriffsschutz über Adreßabbildung (☞ **1.5.2**, **8.3**)
- Separater E/A-Adreßraum (z.B. bei x86 üblich)
  - Zugriff auf Controller-Register nur über spezielle (privilegierte) E/A-Befehle
- Beide Varianten in Gebrauch

Klassen von E/A-Geräten

- **Zeichen-orientierte Geräte**
  - Ein-/Ausgabe einzelner Zeichen
  - z.B. Tastatur, Maus, Modem
- **Block-orientierte Geräte**
  - Ein-/Ausgabe größerer Datenblöcke
  - z.B. Festplatte, Netzwerk
  - Vorteil: Reduzierung der Interrupt-Rate
    - wichtig für Programm-Geschwindigkeit



Grundlegende Konzepte von BSen:

- Prozesse und Threads
- Speicherverwaltung
- Ein/Ausgabe
- Dateiverwaltung

Dazu orthogonale Aufgaben:

- Sicherheit
- Ablaufplanung und Resourcenverwaltung

Definitionen

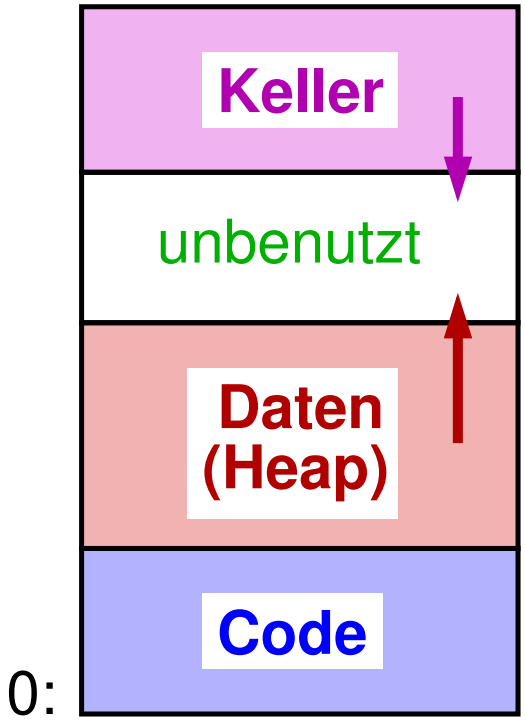
- Anschaulich: ein **Prozeß** ist ein Programm in Ausführung
- Formaler:
  - Aktivitätseinheit, gekennzeichnet durch
    - eine Ausführungsumgebung
      - Adreßraum (Programmcode und Daten)
      - Zustandsinformation benutzter Ressourcen (z.B. offene Dateien, Position der Lesezeiger, ...)
    - ein oder mehrere Aktivitätsträger (**Threads**, „Ablauffäden“)
- Anmerkungen:
  - implementierungsnahe Definition
  - klassischer Prozess enthält genau einen Thread
  - heute in den meisten BSen mehrfädige Prozesse möglich

Abarbeitung von Threads: *Timesharing*

- Threads werden abwechselnd vom Prozessor (bzw. den Prozessoren) bearbeitet
  - BS entscheidet, wer wann wie lange (auf welchem Prozessor) rechnen darf
- Gründe für Timesharing:
  - Bedürfnisse des Nutzers (mehrere Anwendungen)
  - bessere Auslastung des Rechners
- **Prozeß**wechsel bedingt Wechsel der Ausführungsumgebung
- Threads eines Prozesses teilen sich die Ausführungsumgebung
  - Wechsel zwischen Threads desselben Prozesses ist effizienter

(Virtueller) Adreßraum

- Beginnt bei Adresse 0, durchnumeriert bis Obergrenze
  - linearer Adreßraum
- Enthält:
  - Programmcode
  - Programmdaten (incl. Heap)
  - Keller (Rückkehradressen)



- Abstraktion des physischen Speichers
  - unabhängig von Größe und Technologie des physischen Speichers

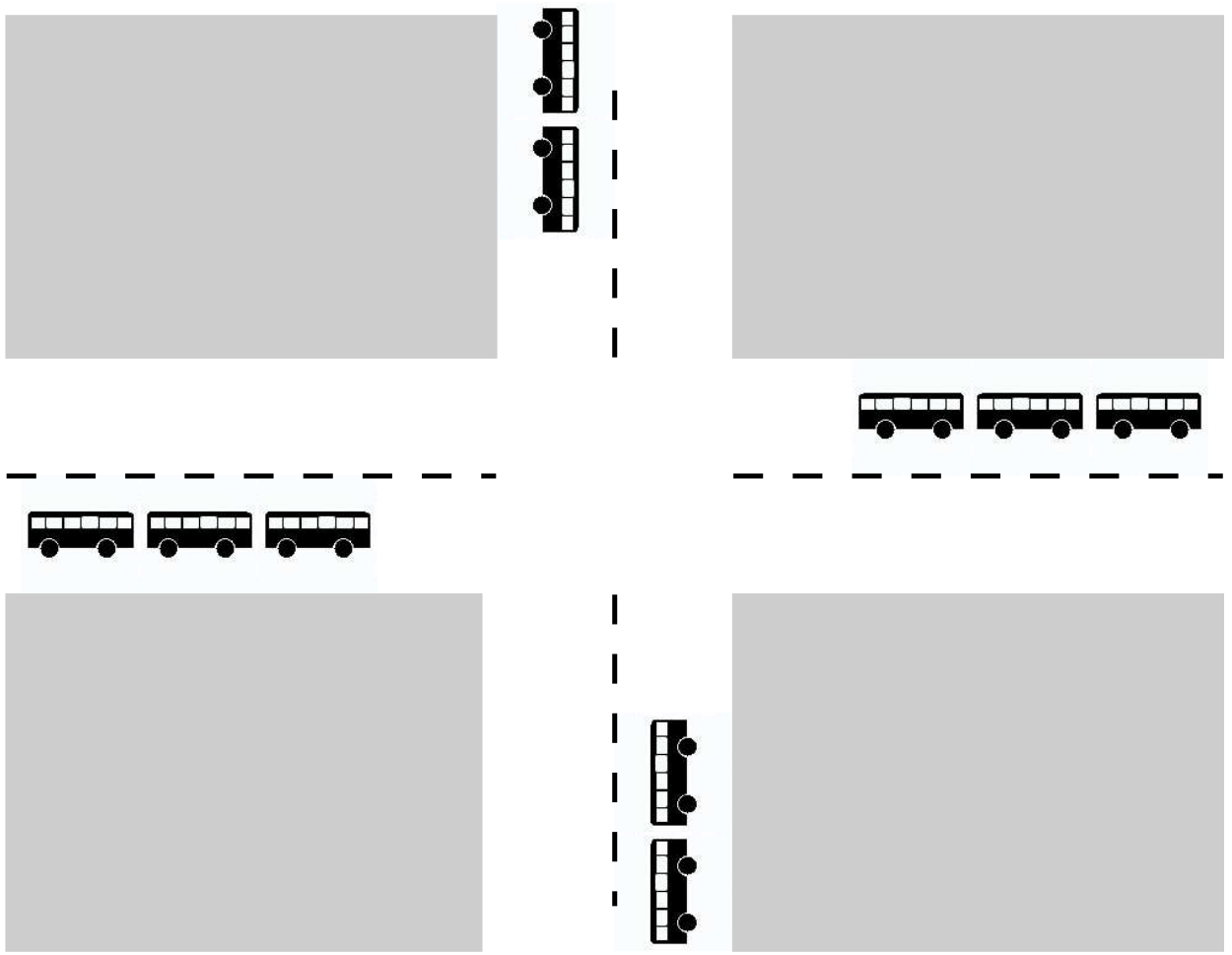
Ausführungskontext

- Alle sonstigen Daten, die zur Ausführung gebraucht werden
  - Prozessorstatus (Datenregister, PC, PSW, ...)
  - BS-relevante Daten (Eigentümer, Priorität, Eltern-Prozeß, genutzte Betriebsmittel, ...)
- Aufgeteilt in Prozeß- und Threadkontext
- Speicherung des Ausführungskontexts:
  - i.d.R. im Adreßraum des BSs: **Prozeß-** bzw. **Threadtabelle**
  - alternativ: (geschützter) Teil des Prozeßadreßraums

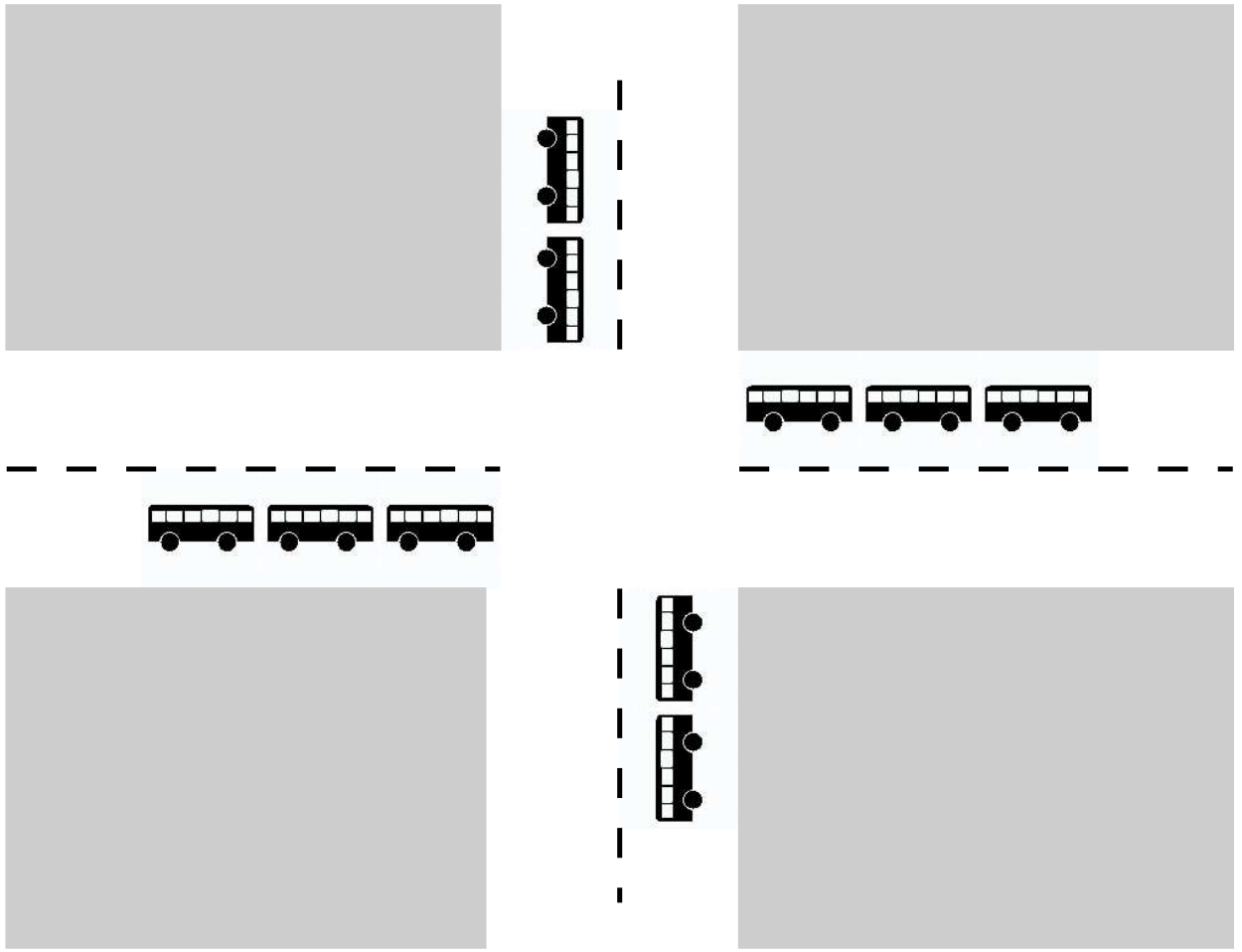
Prozeß-/Threadinteraktion (Interprozeßkommunikation, IPC)

- **Kommunikation**: Austausch von Informationen
  - durch Verwendung eines gemeinsamen Adreßraums (**Speicherkopplung**)
    - i.d.R. zwischen Threads eines Prozesses
  - durch explizites Senden/Empfangen von Nachrichten (**Nachrichtenkopplung**)
    - i.d.R. zwischen verschiedenen Prozessen
- **Synchronisation**
  - Steuerung der zeitlichen Reihenfolge von Aktivitäten (meist: Zugriffe auf gemeinsame Resourcen)
  - Problem: **Verklemmungen** (***Deadlocks***)
    - zyklische Wartebeziehungen

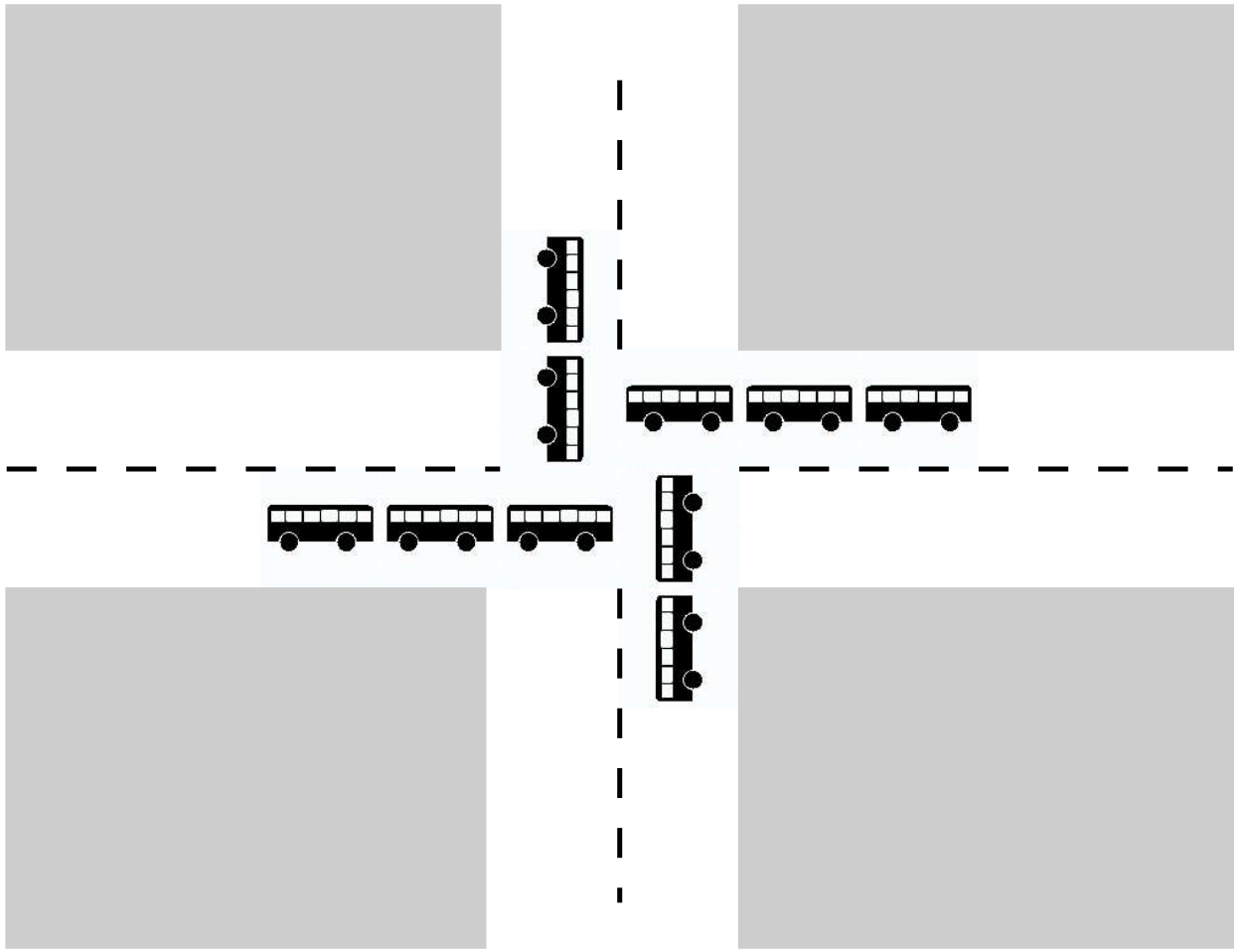
Eine anschauliche Verklemmung



Eine anschauliche Verklemmung

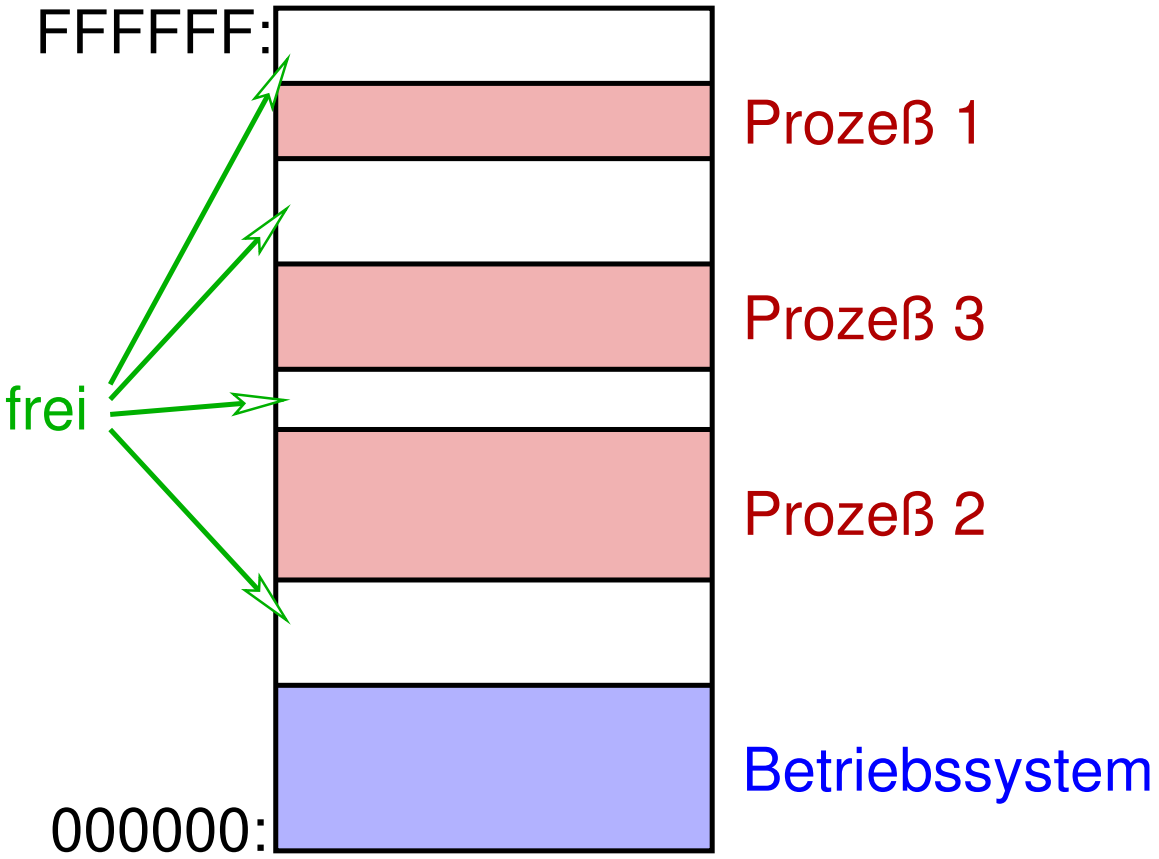


Eine anschauliche Verklemmung



Situation in Mehrprozeß-Systemen:

- Prozesse und BS teilen sich den Hauptspeicher, z.B.:



Fragen / Probleme für das BS:

- Wie werden die Speicherbereiche an Prozesse zugeteilt?
  - BS muß genügend großes Stück Speicher finden
  - was ist, wenn laufender Prozeß mehr Speicher anfordert?
- Programme enthalten Adressen
  - kann man Programme an beliebige Adressen schieben?
- Schutz der Prozesse gegeneinander
  - wie kann man verhindern, daß ein Prozeß auf den Speicher eines anderen Prozesses (oder des BSs) zugreift?
- Begrenzte Größe des Hauptspeichers
  - was ist, wenn nicht alle Prozesse in den Hauptspeicher passen?

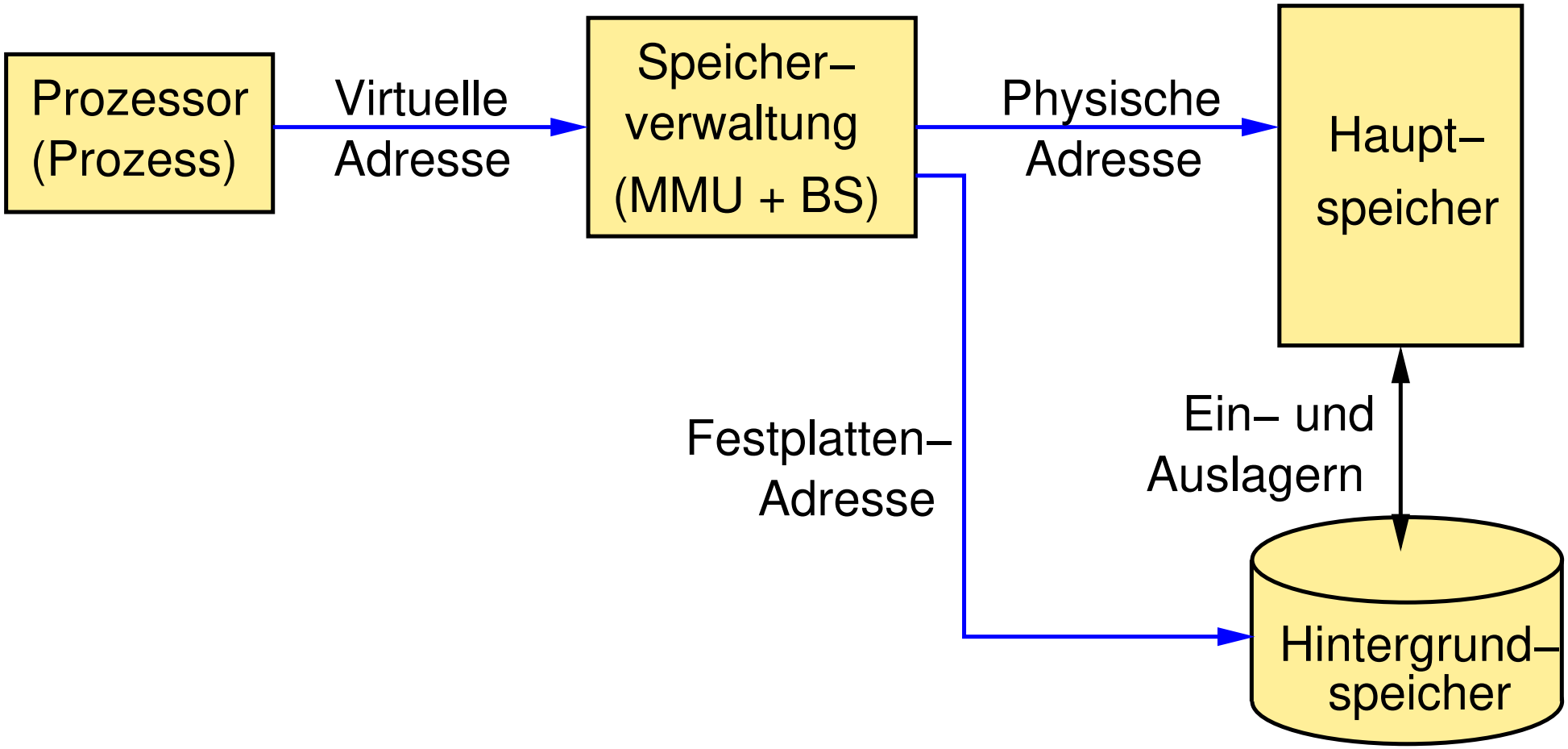
Fragen / Probleme für das BS: ...

- Transparenz für die Anwendungen
  - im Idealfall sollte die Anwendung annehmen dürfen, sie hätte den Rechner für sich alleine
  - Programmieraufwand sollte sich durch Speicherverwaltung nicht erhöhen

Lösung: Adressabbildung

- Adressen des Prozesses (**virtuelle Adressen**) werden durch Hardware (***Memory Management Unit***, **MMU**) auf Speicheradressen (**physische Adressen**) umgesetzt
  - einfachster Fall: Addition einer Basis-Adresse und Limit
- Weiterer Vorteil: Adreßraum kann auf Hauptspeicher und Hintergrundspeicher aufgeteilt werden

Adressierung beim virtuellen Speicher



1.5.3 Dateiverwaltung

- **Datei**: Einheit zur Speicherung von Daten
  - **persistente Speicherung**: über das Ende von Prozessen (Anwendungen) hinaus
- Struktur einer Datei:
  - bei heutigen BSen für PCs und Server: unstrukturierte Folge von Bytes (d.h.: BS kennt Struktur nicht)
  - in Mainframe-BSen u.a.:
    - Sequenz von Datensätzen fester Struktur
    - Index zum schnellen Zugriff auf Datensätze
- Typen von Dateien (u.a.):
  - reguläre Dateien (Dokumente, Programme, ...)
  - Verzeichnisse: Information zur Struktur des Dateisystems
  - Gerätedateien: falls Geräte ins Dateisystem abgebildet sind

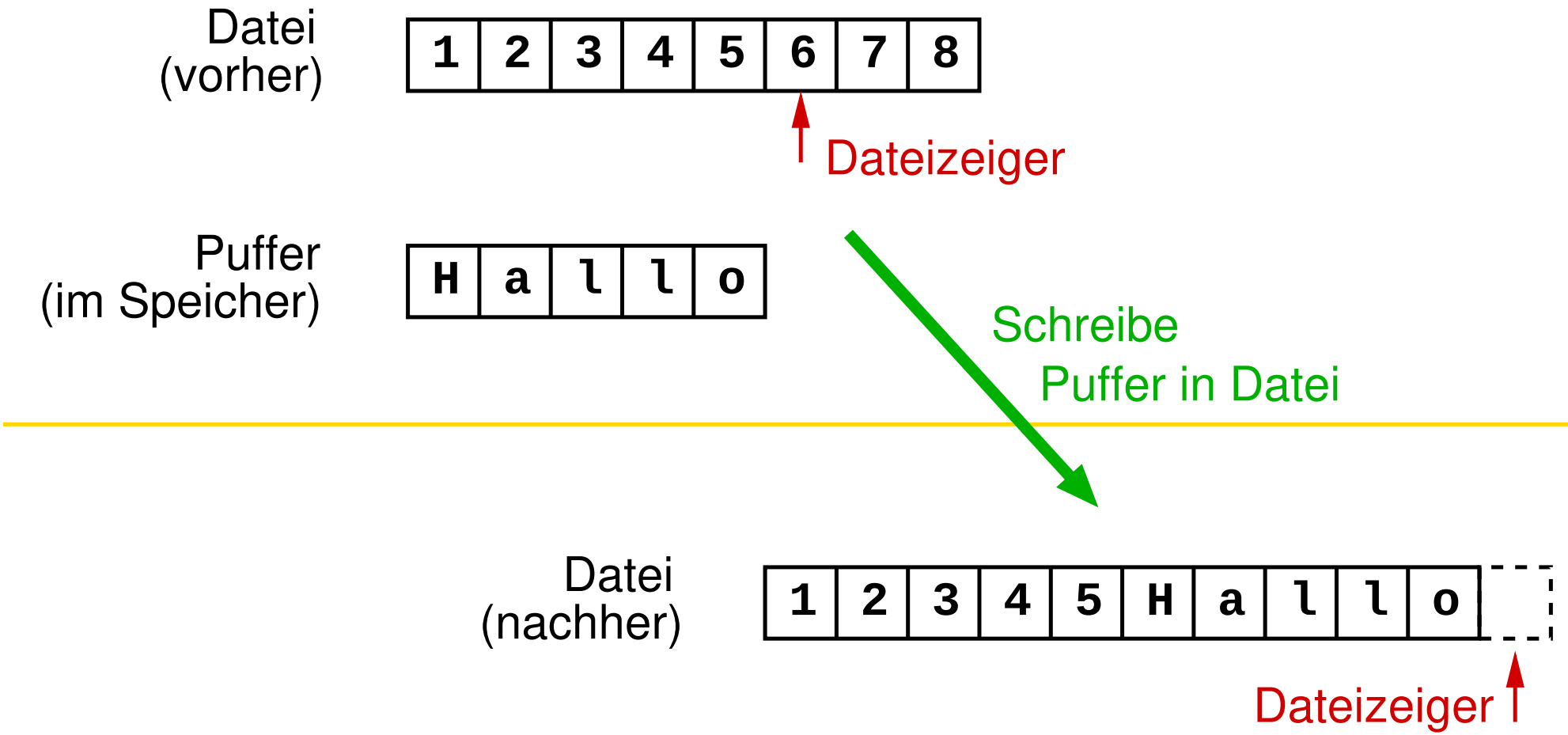
Typische Dateioperationen

- Öffnen einer Datei; Argumente u.a.:
  - Lesen/Schreiben/Anfügen, Erzeugen, Sperren, ...
- Schließen der Datei
- Lesen / Schreiben von Daten
  - Zugriff meist sequentiell (historisch bedingt)
  - daneben auch wahlfreier Zugriff möglich
    - Offset als Parameter beim Lesen / Schreiben oder explizites Positionieren (*seek*)
  - Alternative: Einblenden in den Prozeß-Adreßraum
- Erzeugen, Löschen, Umbenennen
- Lesen und Schreiben von Dateiattributen (z.B. Zugriffsrechte)

Das sequentielle Dateimodell

- Datei wird als unstrukturierte Byte-Folge aufgefaßt
- Nach dem Öffnen einer Datei besitzt ein Prozeß einen privaten **Dateizeiger**
  - nächstes zu lesendes Byte bzw. Schreibposition
- Lese- und Schreib-Operationen kopieren einen Datenblock zwischen Hauptspeicher und Datei
  - Dateizeiger wird entsprechend weitergeschoben
- Lesen über das Dateiende hinaus (***End-of-file***) ist nicht möglich
- Schreiben über das Dateiende führt zum Anfügen an Datei
- Dateizeiger kann auch explizit positioniert werden

Beispiel: Schreiben in eine sequentielle Datei

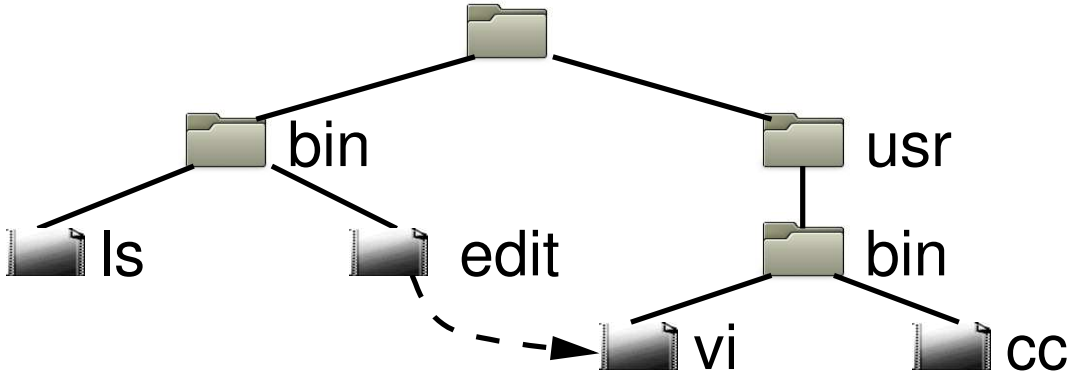


Verzeichnisse

- Zur hierarchischen Organisation von Dateien
  - Benennung von Dateien über Pfadnamen
  - absolute Pfadnamen (beginnend bei Wurzelverzeichnis)
    - z.B. `/home/wismueller/Vorlesungen/BS1/v02.pdf`
  - relative Pfadnamen
    - beginnen beim aktuellen Arbeitsverzeichnis
    - z.B. `BS1/v02.pdf`: selbe Datei wie oben, wenn Arbeitsverzeichnis `/home/wismueller/Vorlesungen` ist

Verzeichnisse ...

- Oft: Einführung von Verweisen (*Links*)
  - Datei (bzw. Verzeichnis) kann in mehr als einem Verzeichnis auftreten, ist aber physisch nur einmal vorhanden



/bin/edit führt zur Datei /usr/bin/vi

  - Links sind in der Regel transparent für die Anwendungen

1.5.3 Dateiverwaltung ...

Typische Aufgaben / Konzepte von BSen (Beispiel: Linux)

- Realisierung von Dateien und Verzeichnishierarchien
  - Abbildung von Dateien / Verzeichnissen auf das Speichermedium (z.B. Festplatte)
- Realisierung der Dateioperationen: (Erzeugen, Lesen, ...)
  - dabei Überprüfung der Zugriffsrechte
- Einhängen von Dateisystemen in die Verzeichnishierarchie
  - z.B. Zugriff auf Dateien auf Diskette über den Pfad `/media/floppy`
  - einheitliche Sicht auf alle Dateisysteme
- Spezialdateien (Gerätedateien, Pipes, Prozeßdateisystem, ...)

Aufgabenbereiche:

- Zugriffskontrolle
  - Benutzerzugriff auf Gesamtsystem, Subsysteme, Daten
  - Prozeßzugriff auf Ressourcen und Objekte des Systems
- Kontrolle des Informationsflusses
  - Regulierung des Datenflusses im System / an Benutzer (keine Weitergabe vertraulicher Daten an Unautorisierte)
- Zertifizierung
  - Nachweis, daß gewünschte Sicherheitseigenschaften vom System durchgesetzt werden

- In dieser Vorlesung wird nur Zugriffskontrolle betrachtet!

Beispiel: Zugriffskontrolle unter Linux

- Einführung von Benutzern und Benutzergruppen
  - spezieller Benutzer `root`: Administrator
- Prozesse und Dateien haben Eigentümer und Eigentümergruppe
- Zugriff auf Prozesse nur für Eigentümer (und `root`)
- Zugriff auf Dateien über 9 Rechtebits festgelegt: `rwx r-x ---` (user group others)
  - `r` = Lesen, `w` = Schreiben, `x` = Ausführen
  - im Bsp.: Eigentümer darf alles, Benutzer der Eigentümergruppe darf nicht schreiben, für alle anderen kein Zugriff
- Rechte werden bei jedem (relevanten) Zugriff vom BS geprüft

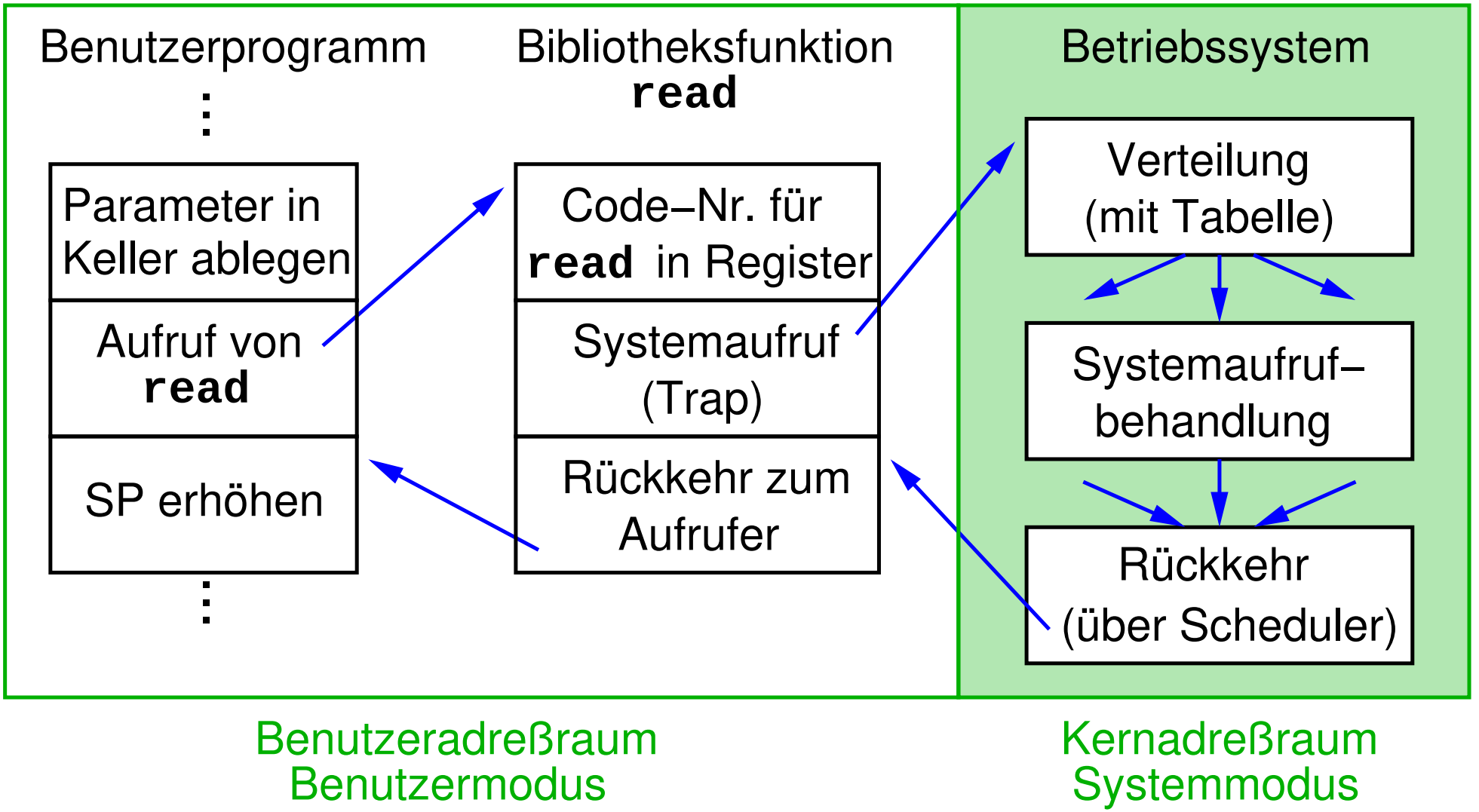
1.5.5 Ablaufplanung u. Resourcenverwaltung

- Verwaltung / Planung der Ressourcen-Verwendung, z.B.
  - Hauptspeicher: Welcher Prozeß erhält wann wo wieviel Speicher? Wann wird Speicher ausgelagert?
  - Platten-E/A: Reihenfolge der Bedienung? (Optimierung der Armbewegung)
- Ziele:
  - Fairness: jeder Prozess sollte denselben Anteil der Ressourcen erhalten
  - Differenzierung: Berücksichtigung der unterschiedlichen Anforderungen verschiedener Job-Klassen
  - Effizienz: Durchsatz, Antwortzeit, Anzahl der Benutzer
    - Ziele teilweise widerspüchlich

1.6 Systemaufrufe

- Schnittstelle zwischen Benutzerprogrammen und BS
- Systemaufrufe meist in Bibliotheksfunktionen gekapselt
  - Details des Systemaufrufs sind hardwareabhängig
  - In der Regel:
    - Systemaufrufe sind durchnummeriert
    - Nummer wird vor Ausführung des *Trap*-Befehls in festes Register geschrieben
  - Im BS dann Verzweigung über Funktionstabelle

Ablauf eines Systemaufrufs



Anmerkungen zum Ablauf

- BS sichert zunächst den vollständigen Prozessorstatus in der Threadtabelle
- Aufrufender Thread kann blockiert werden
  - z.B. Warten auf Eingabe
- Rückkehr aus dem BS erfolgt über den Scheduler
  - er bestimmt den Thread, der weitergeführt wird
  - Rückkehr also nicht unbedingt (sofort) zum aufrufenden Thread
- Bei Rückkehr: Restaurieren des Prozessorstatus des weitergeführten Threads

Typische Systemaufrufe (Beispiel: Linux)

- Prozeßmanagement
  - `fork` – Erzeugen eines Kindprozesses
  - `waitpid` – Warten auf Ende eines Kindprozesses
  - `execve` – Ausführen eines anderen Programms
  - `exit` – Prozeß beenden und Statuswert zurückgeben
- Dateimanagement
  - `open` – Datei öffnen (Lesen/Schreiben)
  - `close` – Datei schließen
  - `read` – Daten aus Datei in Puffer lesen
  - `write` – Daten aus Puffer in Datei schreiben
  - `lseek` – Dateizeiger verschieben

Typische Systemaufrufe (Beispiel: Linux) ...

- Verzeichnismanagement
  - `mkdir` – Verzeichnis erzeugen
  - `rmdir` – Leeres Verzeichnis löschen
  - `unlink` – Datei löschen
  - `mount` – Datenträger in Dateisystem einhängen
- Verschiedenes
  - `chdir` – Arbeitsverzeichnis wechseln
  - `chmod` – Datei-Zugriffsrechte ändern
  - `kill` – Signal an Prozeß senden
  - `time` – Uhrzeit

1.7 Zusammenfassung / Wiederholung

- Aufgaben eines BSs
  - Erweiterung / Abstraktion der Hardware
  - Verwaltung der Betriebsmittel
    - Prozessor, Speicher, Platte, E/A, ...
  - BS liegt zwischen Anwendungen und Hardware
  - Zugriff auf Hardware nur über BS
- Entwicklung der BSe
- Arten von BSen

1.7 Zusammenfassung / Wiederholung ...

- Computer-Hardware
  - Ausführungsmodi der CPU
    - Systemmodus für BS
    - Benutzermodus für Anwendungen
      - Speicherabbildung, keine privilegierten Befehle
  - Unterbrechungen:
    - Systemaufruf (*Trap*)
    - Ausnahme (*Exception*)
    - Interrupt

1.7 Zusammenfassung / Wiederholung ...

- Grundkonzepte von BSen
  - Prozesse
  - Speicherverwaltung
  - Ein-/Ausgabe
  - Dateiverwaltung
- Konzeptübergreifende Aufgaben
  - Sicherheit
  - Ablaufplanung und Resourcenverwaltung
- Systemaufrufe

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

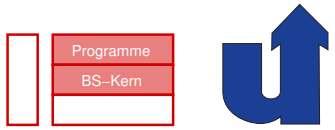
2 Prozesse und Threads

2 Prozesse und Threads ...

- Begriffsklärung
- Nebenläufige Programmierung
- Thread-/Prozeßmodell und -zustände
- Implementierung von Prozessen und Threads
- Realisierungsvarianten für Threads
- Schnittstellen zur Nutzung von Threads

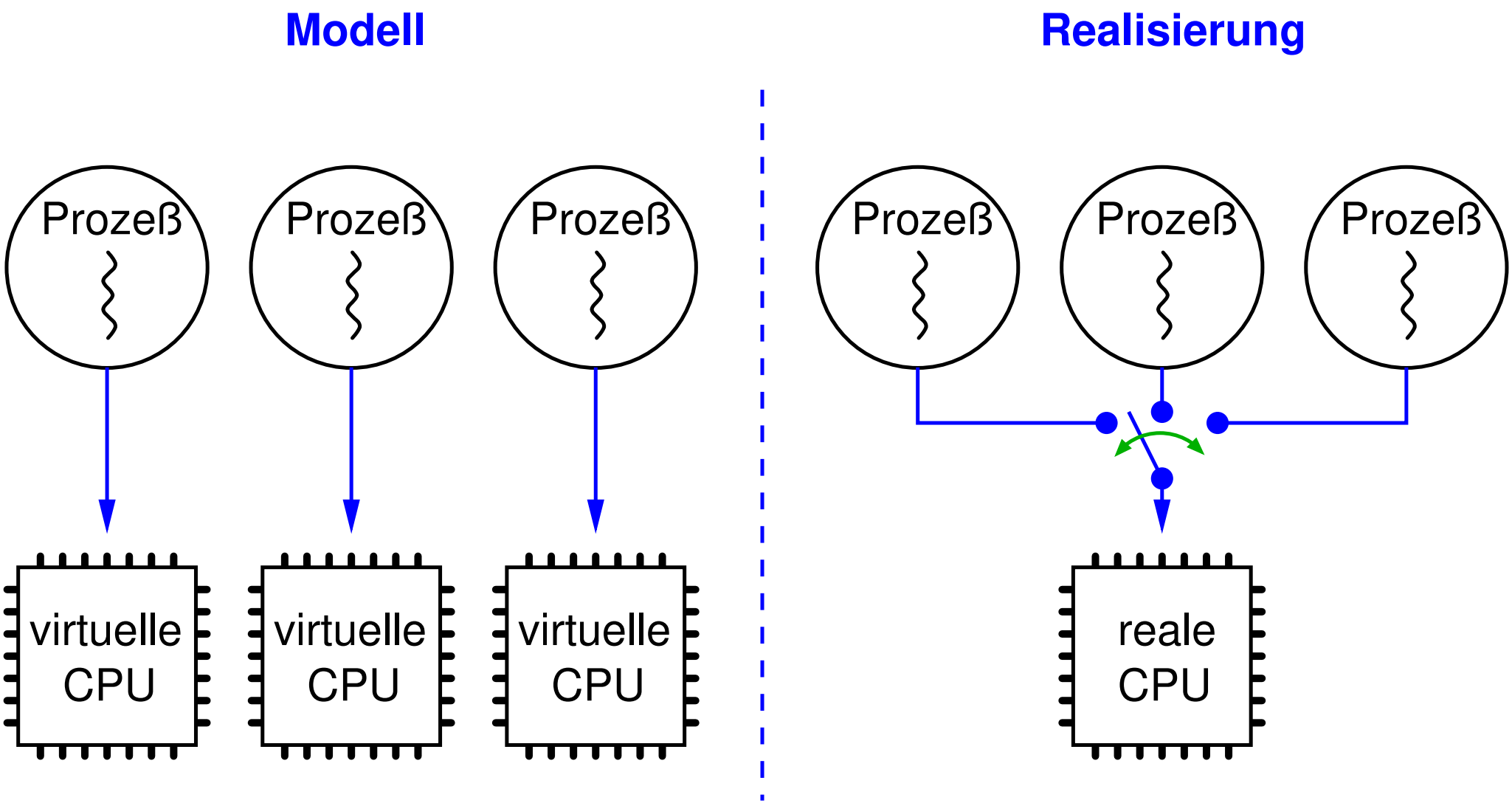
- Tanenbaum 2.1 - 2.2
- Stallings 3.1 - 3.2

2.1 Begriffsklärung



- Ein Prozeß ist ein Programm in Ausführung
- Wunsch: ein Rechner soll mehrere Programme „gleichzeitig“ ausführen können
- Konzeptuell: jeder Prozeß
  - wird durch eine eigene, virtuelle CPU ausgeführt
    - **nebenläufige** (quasi-parallele) Abarbeitung der Prozesse
  - hat seinen eigenen (virtuellen) Adreßraum („Speicher“, ☞ **8**)
- Real: (jede) CPU schaltet zwischen den Prozessen hin und her
  - **Multiprogrammierung**, **Mehrprogrammbetrieb**
  - Umschalten durch Umladen der CPU-Register (incl. PC)
  - Beachte: Annahmen über die Geschwindigkeit der Ausführung sind nicht zulässig

2.1 Begriffsklärung ...

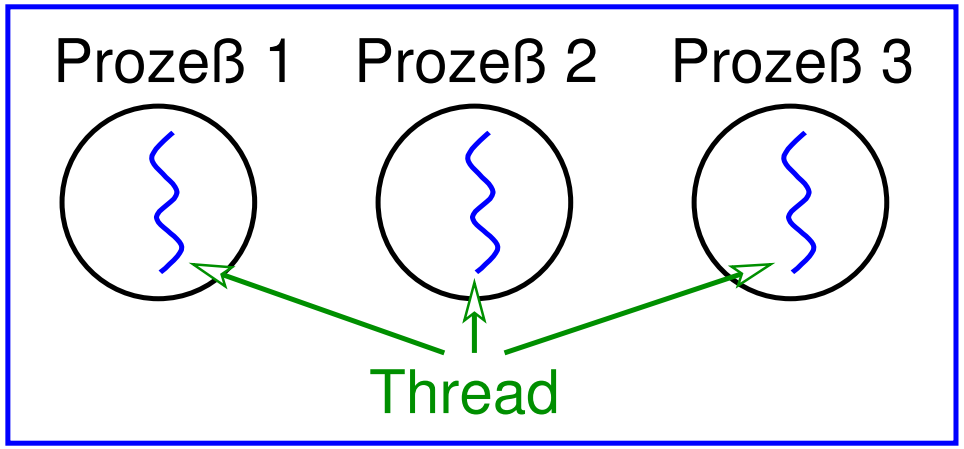


Ein (klassischer) Prozeß besitzt zwei Aspekte:

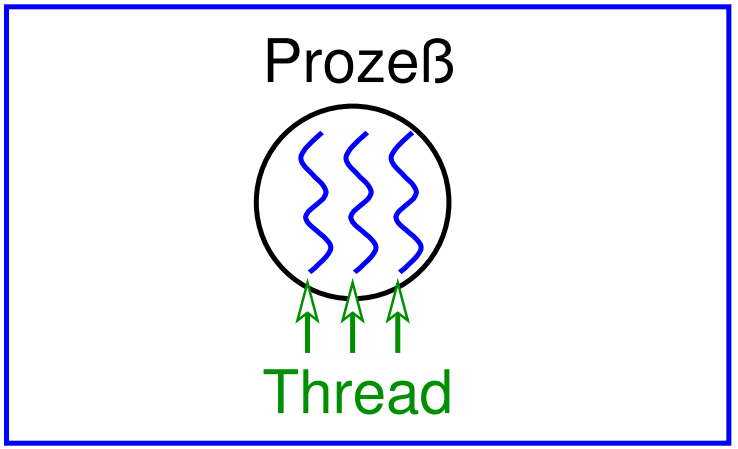
- Einheit des Ressourcenbesitzes
  - eigener (virtueller) Adreßraum
  - allgemein: Besitz / Kontrolle von Ressourcen (Hauptspeicher, Dateien, E/A-Geräte, ...)
  - BS übt Schutzfunktion aus
- Einheit der Ablaufplanung / Ausführung
  - Ausführung folgt einem Weg (*Trace*) durch ein Programm
  - verzahnt mit der Ausführung anderer Prozesse
  - BS entscheidet über Zuteilung des Prozessors
- vgl. Definition aus 1.5.1

In heutigen BSen: Trennung der Aspekte

- **Prozeß**: Einheit des Ressourcenbesitzes und Schutzes
- **Thread**: Einheit der Ausführung (Prozessorzuteilung)
  - Ausführungsfaden, leichtgewichtiger Prozeß
- Damit: innerhalb eines Prozesses auch mehrere Threads möglich
  - d.h., Anwendungen können mehrere nebenläufige Aktivitäten besitzen



3 (klassische) Prozesse



3 Threads in einem Prozeß

Vorteile bei der Nutzung mehrerer Threads in einer Anwendung

- Nebenläufige Programmierung möglich: mehrere Kontrollflüsse
- Falls ein Thread auf Ein-/Ausgabe wartet: die anderen können weiterarbeiten
- Kürzere Reaktionszeit auf Benutzereingaben
- Bei Multiprozessor-Systemen (bzw. mit Hyperthreading): echt parallele Abarbeitung der Threads möglich

Beispiel: GUI-Programmierung

- Sequentielles Programm (1 Thread):

```
while (true) {
    ComputeStep();              // z.B. Animationsschritt
    if (QueryEvent()) {         // Ereignis angekommen?
        e = ReceiveEvent(); // Ereignis abholen
        ProcessEvent(e);    // und bearbeiten
    }
}
```

  - Verzahnung von Berechnung und Ereignisbehandlung
  - **Polling** von Ereignissen: wann / wie oft?

Beispiel: GUI-Programmierung ...

- Nebenläufiges Programm mit 2 Threads:

Thread 1:

```
while (true) {
    ComputeStep();
}
```

Thread 2:

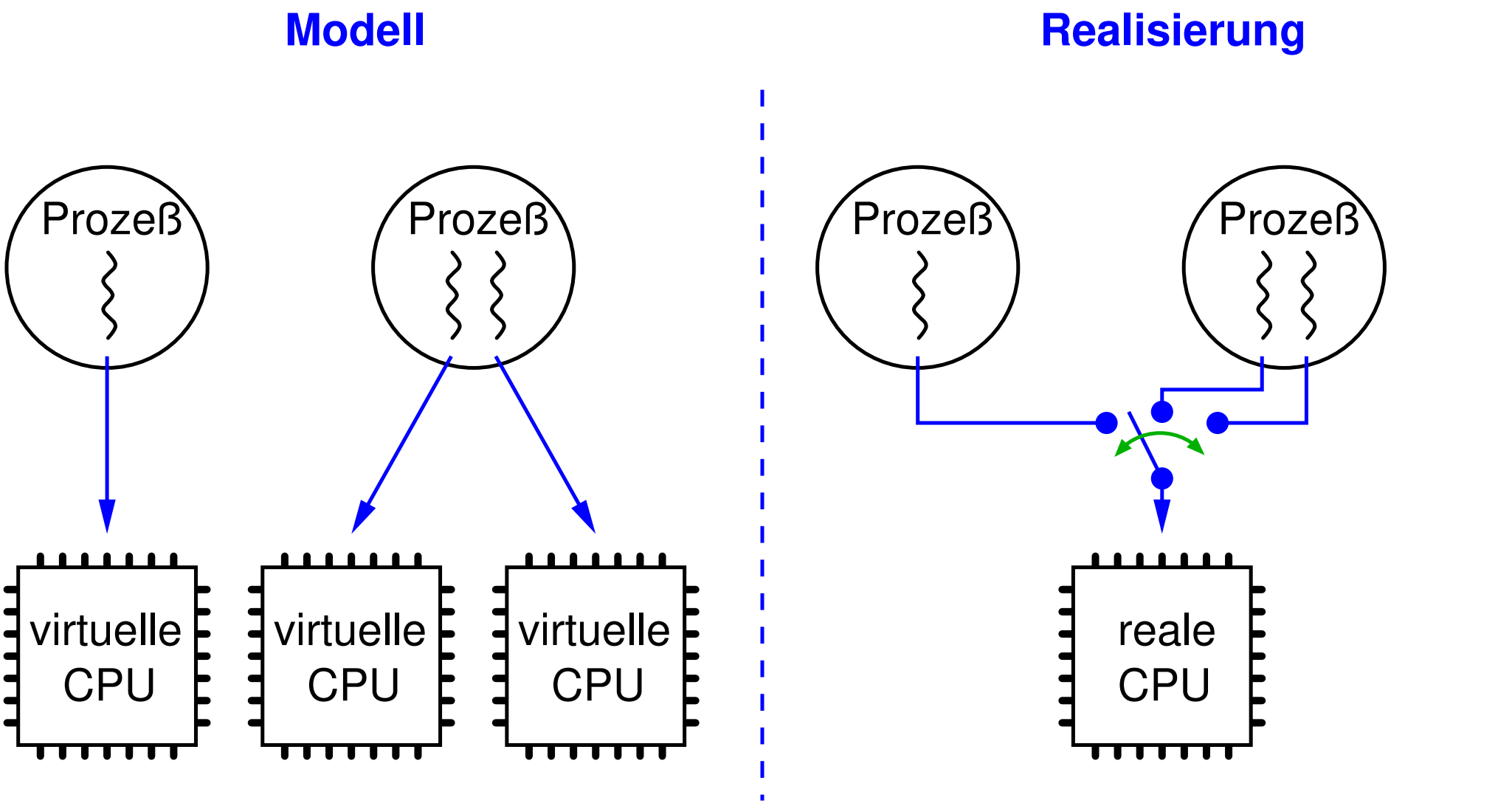
```
while (true) {
    e = ReceiveEvent();
    ProcessEvent(e);
}
```

- einfachere Programmstruktur
- aber: Zugriff auf gemeinsame Variable erfordert Synchronisation (☞ **3**)

Realisierung von Threads

- Heute meist direkt durch das BS
  - andere Alternativen: siehe Ergänzungsfolien
- Konzeptuell: jeder Thread
  - wird durch eine eigene, virtuelle CPU ausgeführt
    - nebenläufige (quasi-parallele) Abarbeitung der Threads
  - nutzt alle anderen Ressourcen seines Prozesses (u.a. den virtuellen Adreßraum) gemeinsam mit dessen anderen Threads
- Real: (jede) CPU schaltet zwischen den Threads hin und her
  - **Multithreading**
  - Umschalten durch Umladen der CPU-Register (incl. PC)

2.1 Begriffsklärung ...

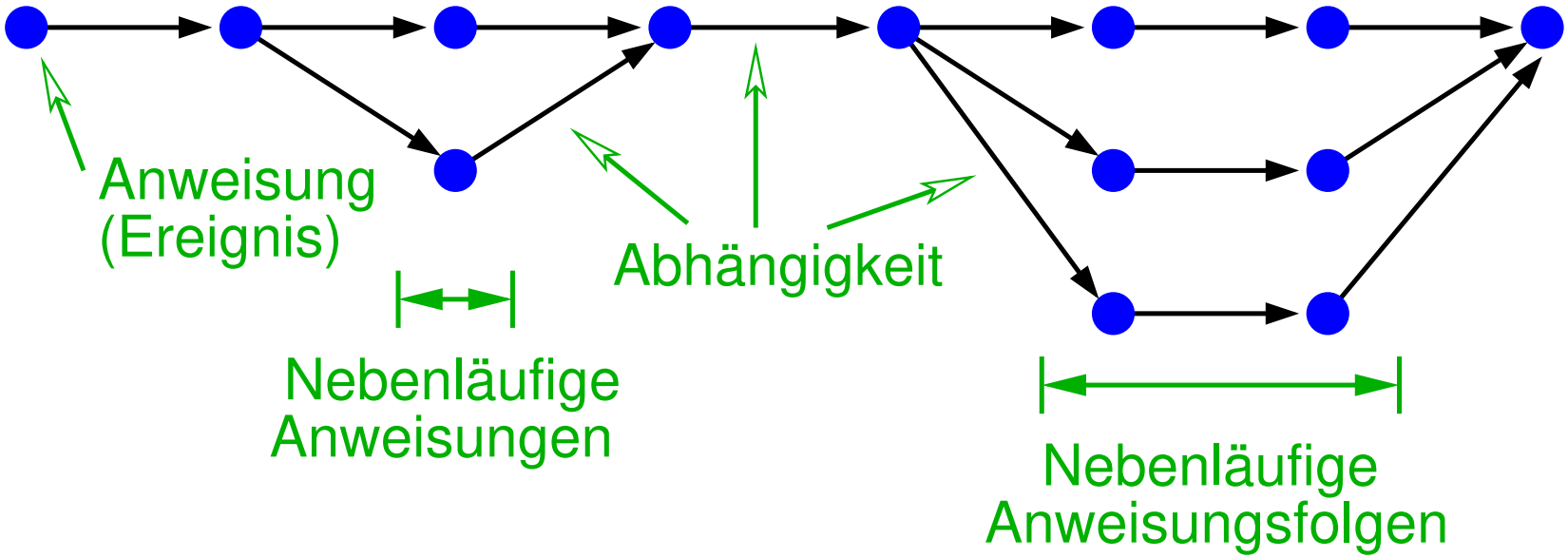


2.2 Nebenläufige Programmierung

- Einfache Programme arbeiten rein **sequentiell**
  - die Anweisungen werden streng nacheinander abgearbeitet
- Oft will man Aktivitäten definieren, die „gleichzeitig“ ausgeführt werden können
  - z.B. Abspielen von Bild und Ton eines Videos
  - z.B. Drucken im Hintergrund
- Wichtige Unterscheidung:
  - zwei Aktivitäten sind **parallel**, wenn sie gleichzeitig ausgeführt werden
  - zwei Aktivitäten sind **nebenläufig**, wenn sie parallel ausgeführt werden **können**
    - d.h., wenn es keine kausalen Abhängigkeiten zwischen ihnen gibt

Graphische Darstellung

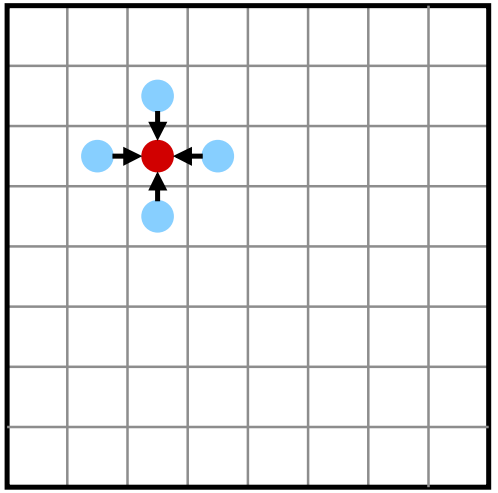
- Aktivitäten (Ereignisse, Anweisungen, ...) werden als Knoten gezeichnet
- Abhängigkeiten als gerichtete Kanten
- Beispiel:



- Mathematisch entspricht das einer Halbordnung

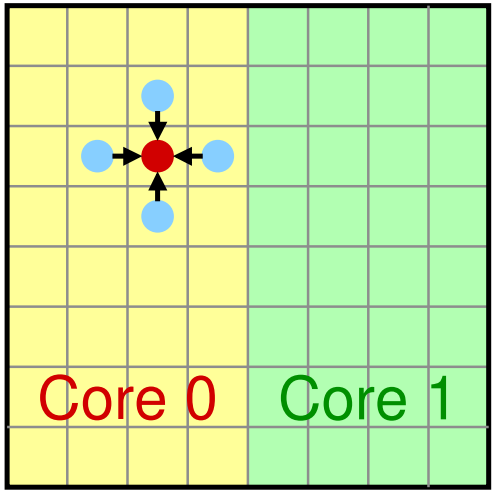
Motivationen für nebenläufige Abarbeitung

- Gleichzeitiges Rechnen
  - Beschleunigung von Programmen durch gleichzeitige Nutzung mehrerer CPU-Cores
  - Parallelverarbeitung
  - bei rechenintensiven Programmen
  - Beispiel: Simulation des Wärmeausgleichs (Jacobi-Verfahren)
    - iteratives Verfahren
    - ersetze Temperatur eines Punktes durch Mittelwert der Nachbarpunkte



Motivationen für nebenläufige Abarbeitung

- Gleichzeitiges Rechnen
  - Beschleunigung von Programmen durch gleichzeitige Nutzung mehrerer CPU-Cores
  - Parallelverarbeitung
  - bei rechenintensiven Programmen
  - Beispiel: Simulation des Wärmeausgleichs (Jacobi-Verfahren)
    - iteratives Verfahren
    - ersetze Temperatur eines Punktes durch Mittelwert der Nachbarpunkte
    - einfache Aufteilung der Arbeit auf mehrere Bearbeiter möglich



Motivationen für nebenläufige Abarbeitung ...

- Gleichzeitiges Warten
  - optimale Nutzung der einzelnen CPU-Cores
  - Vermeidung von Leerlauf der Cores
  - bei E/A-lastigen Programmen
  - Beispiel: Drucken im Texteditor
    - während auf den Drucker gewartet wird, sollten weiterhin Benutzereingaben möglich sein

Abhängigkeiten in Programmen

- Frage: wann können zwei Aktivitäten (z.B. Anweisungen) eines Programms nebenläufig ausgeführt werden?
- Antwort: wenn es keine **Abhängigkeit** zwischen ihnen gibt
- Eine Anweisung S' ist abhängig von einer Anweisung S, wenn das Programm nur dann korrekt arbeitet, wenn S' **nach** S ausgeführt wird
- Beispiel:

S_1: `var1 = 8;`
S_2: `var2 = 5;`
S_3: `sum = var1 + var2;`
S_4: `System.out.println(sum);`

Abhängigkeiten in Programmen

- Frage: wann können zwei Aktivitäten (z.B. Anweisungen) eines Programms nebenläufig ausgeführt werden?
- Antwort: wenn es keine **Abhängigkeit** zwischen ihnen gibt
- Eine Anweisung S' ist abhängig von einer Anweisung S, wenn das Programm nur dann korrekt arbeitet, wenn S' **nach** S ausgeführt wird
- Beispiel:

S_2: `var2 = 5;` ⇄ Vertauschen ändert nichts am Ergebnis
S_1: `var1 = 8;`
S_3: `sum = var1 + var2;`
S_4: `System.out.println(sum);`

  - S_1 und S_2 sind unabhängig

Abhängigkeiten in Programmen

- Frage: wann können zwei Aktivitäten (z.B. Anweisungen) eines Programms nebenläufig ausgeführt werden?
- Antwort: wenn es keine **Abhängigkeit** zwischen ihnen gibt
- Eine Anweisung S' ist abhängig von einer Anweisung S, wenn das Programm nur dann korrekt arbeitet, wenn S' **nach** S ausgeführt wird
- Beispiel:

S_1: `var1 = 8;`
S_3: `sum = var1 + var2;`
S_2: `var2 = 5;`
S_4: `System.out.println(sum);`

nicht vertauschbar!

  - S_1 und S_2 sind unabhängig, S_3 ist von S_1 und S_2 abhängig

Abhängigkeiten in Programmen

- Frage: wann können zwei Aktivitäten (z.B. Anweisungen) eines Programms nebenläufig ausgeführt werden?
- Antwort: wenn es keine **Abhängigkeit** zwischen ihnen gibt
- Eine Anweisung S' ist abhängig von einer Anweisung S, wenn das Programm nur dann korrekt arbeitet, wenn S' **nach** S ausgeführt wird
- Beispiel:

```
S1: var1 = 8;
S2: var2 = 5;
S4: System.out.println(sum);
S3: sum = var1 + var2;
```

nicht vertauschbar!

  - S_1 und S_2 sind unabhängig, S_3 ist von S_1 und S_2 abhängig, S_4 ist von S_3 abhängig

Arten von Abhängigkeiten

- Wir betrachten zwei Anweisungen S, S' in einem imperativen Programm
- Festlegung: S ist die gemäß Programm zuerst auszuführende Anweisung
- **Echte Abhängigkeit** von S nach S':
  - S' nutzt das Rechenergebnis von S
  - d.h. S schreibt in eine Variable, die S' liest
- Beispiel:

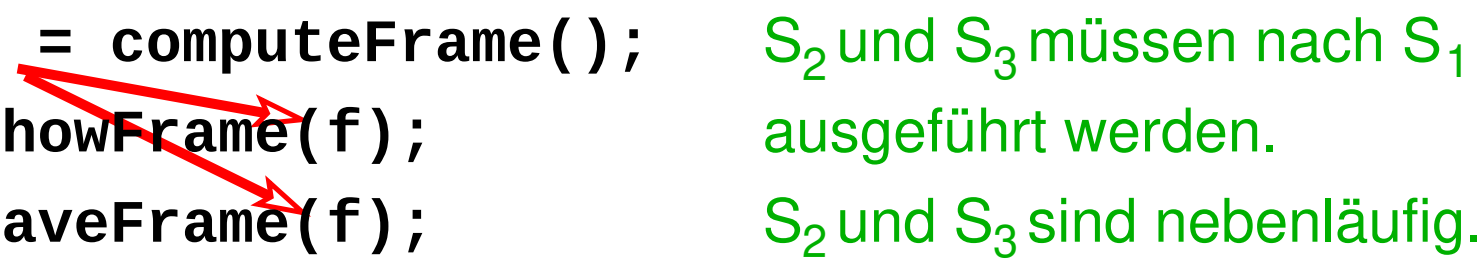
S_1: `f = computeFrame();`

S_2: `showFrame(f);`

S_3: `saveFrame(f);`

Arten von Abhängigkeiten

- Wir betrachten zwei Anweisungen S, S' in einem imperativen Programm
- Festlegung: S ist die gemäß Programm zuerst auszuführende Anweisung
- **Echte Abhängigkeit** von S nach S':
  - S' nutzt das Rechenergebnis von S
  - d.h. S schreibt in eine Variable, die S' liest
- Beispiel:



Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:

S_1: `f = computeFrame();`
S_2: `showFrame(f);`
S_3: `f = compress(f);`
S_4: `saveFrame(f);`

Arten von Abhängigkeiten ...

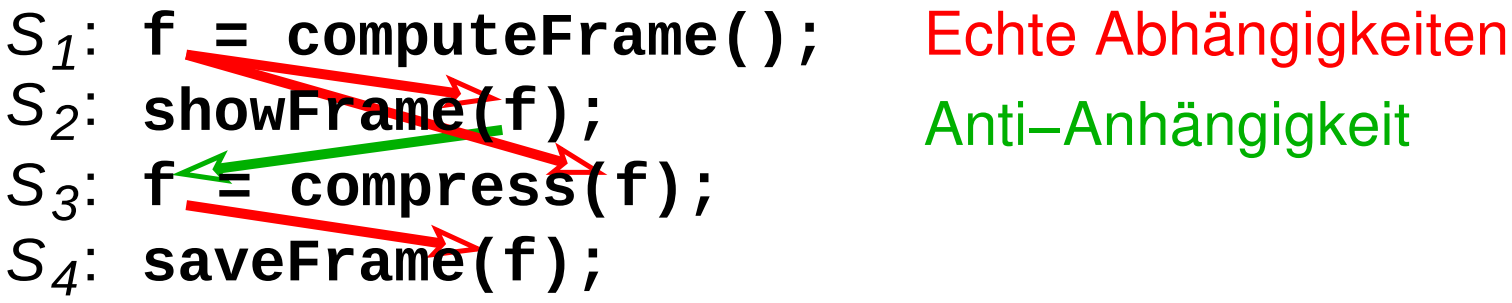
- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:

S_1: `f = computeFrame();`
S_2: `showFrame(f);`
S_3: `f = compress(f);`
S_4: `saveFrame(f);`

Echte Abhängigkeiten

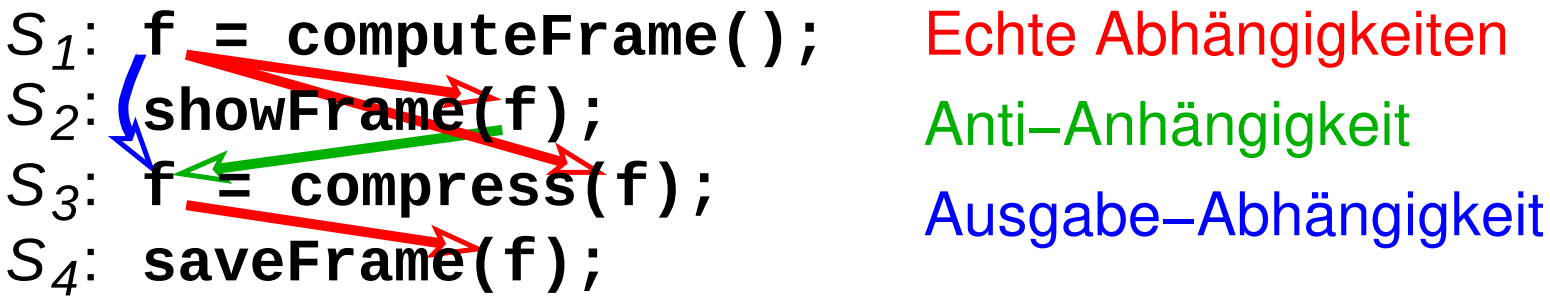
Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:



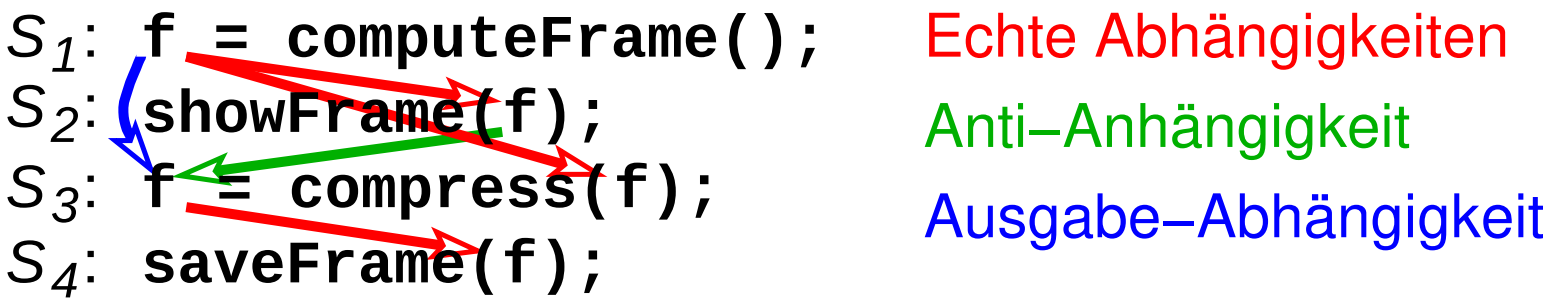
Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:



Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:



- Anti- und Ausgabe-Abh. gibt es nur in imperativen Sprachen
  - werden durch Überschreiben von Variablen verursacht
  - sind immer durch Umbenennung von Variablen zu entfernen

Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:

```
S1: f = computeFrame();
S2: showFrame(f);
S3: c = compress(f);
S4: saveFrame(c);
```

- Anti- und Ausgabe-Abh. gibt es nur in imperativen Sprachen
  - werden durch Überschreiben von Variablen verursacht
  - sind immer durch Umbenennung von Variablen zu entfernen

Arten von Abhängigkeiten ...

- **Anti-Abhängigkeit** von S nach S':
  - S liest eine Variable, die S' überschreibt
- **Ausgabe-Abhängigkeit** von S nach S':
  - S schreibt eine Variable, die S' überschreibt
- Beispiel:

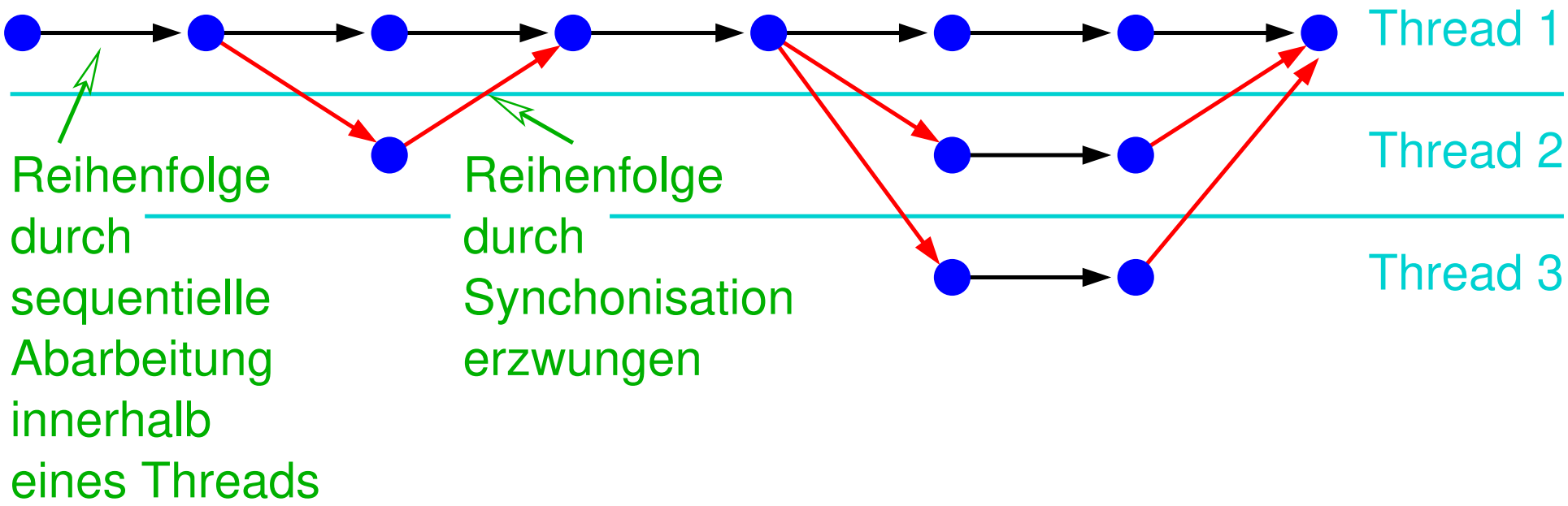
```
S1: f = computeFrame();
S2: showFrame(f);
S3: c = compress(f);
S4: saveFrame(c);
```

Echte Abhängigkeiten

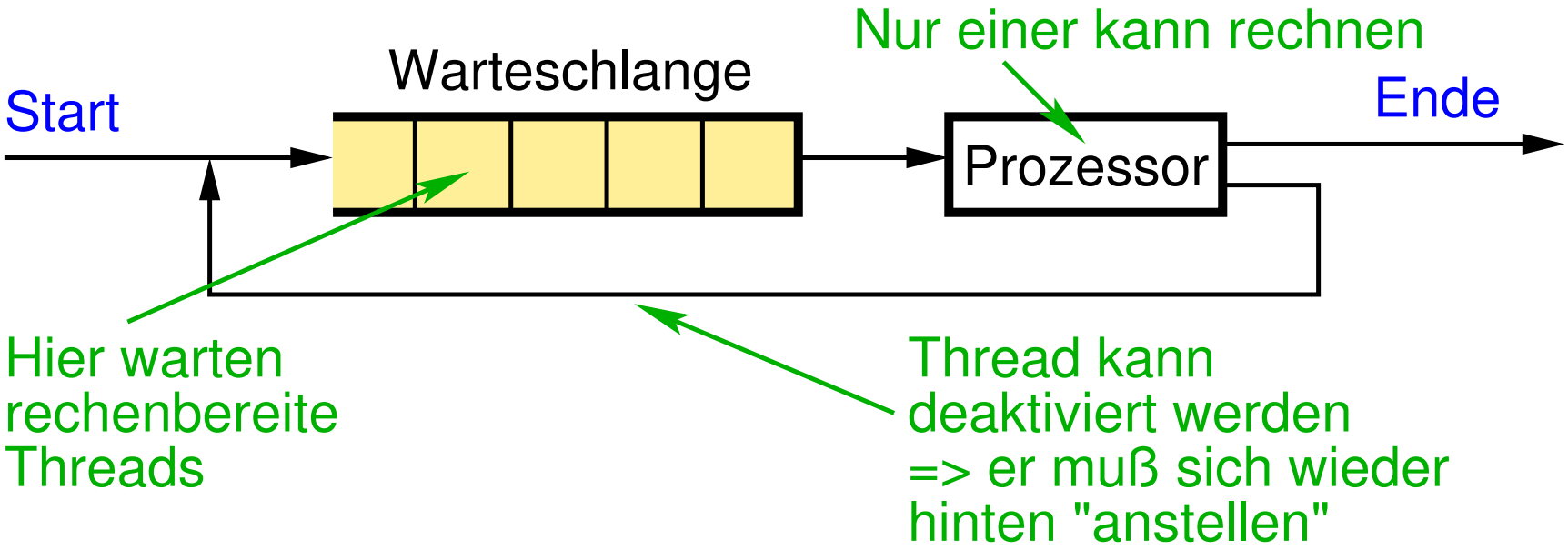
- Anti- und Ausgabe-Abh. gibt es nur in imperativen Sprachen
  - werden durch Überschreiben von Variablen verursacht
  - sind immer durch Umbenennung von Variablen zu entfernen

Abhängigkeiten und Synchronisation

- Abhängigkeit zwischen S und S' bedeutet nicht, dass S und S' im selben Thread ausgeführt werden müssen
- Aber: Synchronisation nötig, um Reihenfolge zu erzwingen
- Im Eingangsbeispiel:

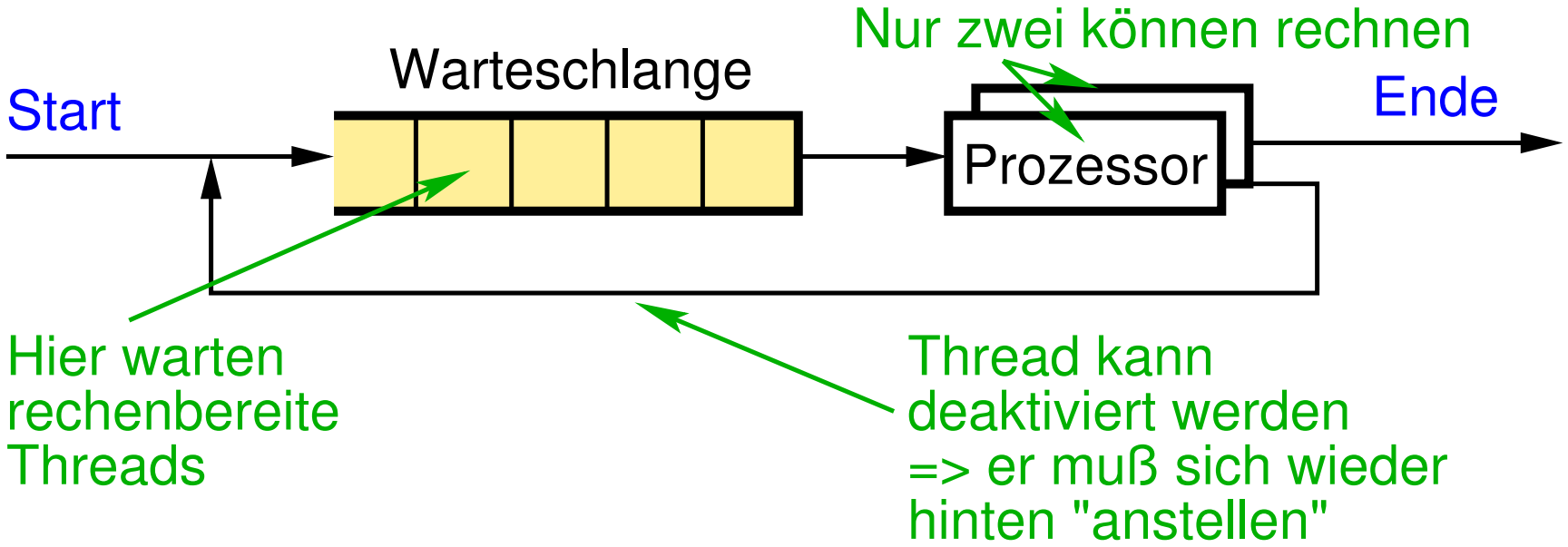


Ein einfaches Thread-Modell



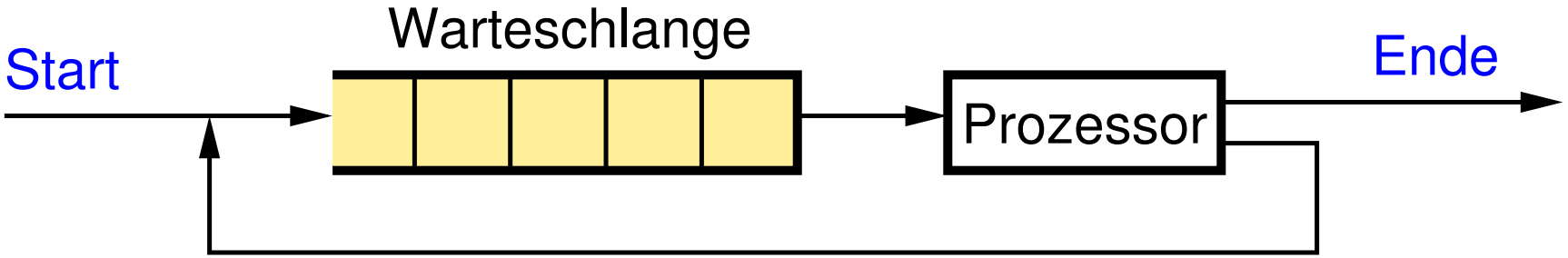
- Anmerkung: alle Modelle in diesem Abschnitt gelten in der selben Form auch für klassische Prozesse (mit genau einem Thread)

Ein einfaches Thread-Modell

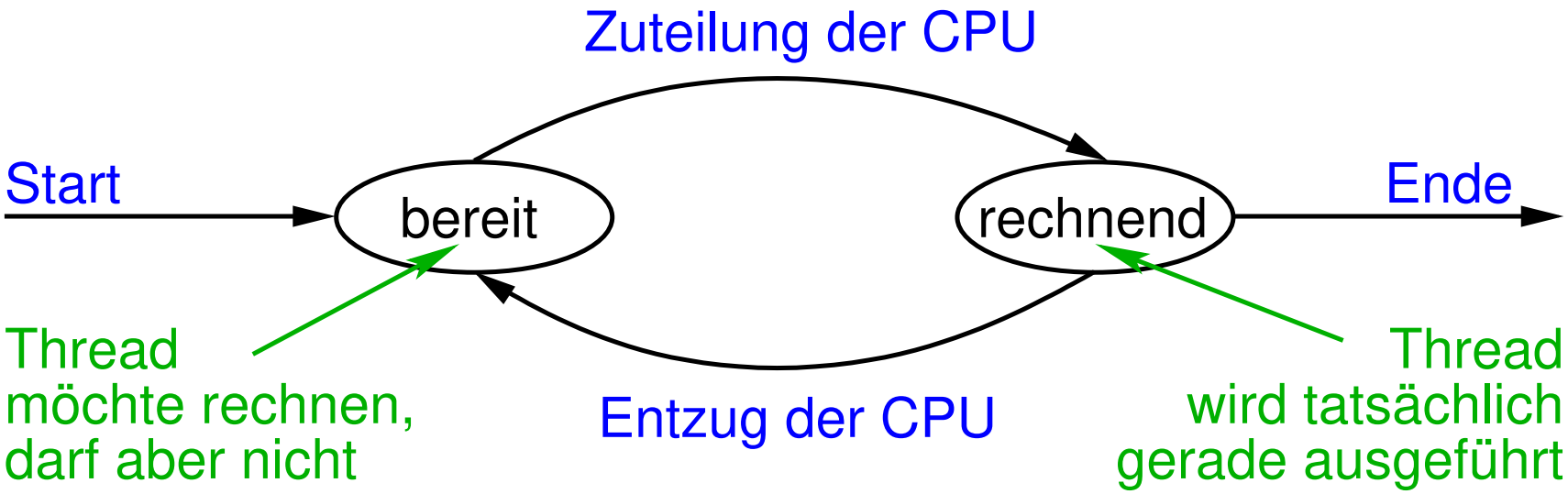


- Anmerkung: alle Modelle in diesem Abschnitt gelten in der selben Form auch für klassische Prozesse (mit genau einem Thread)

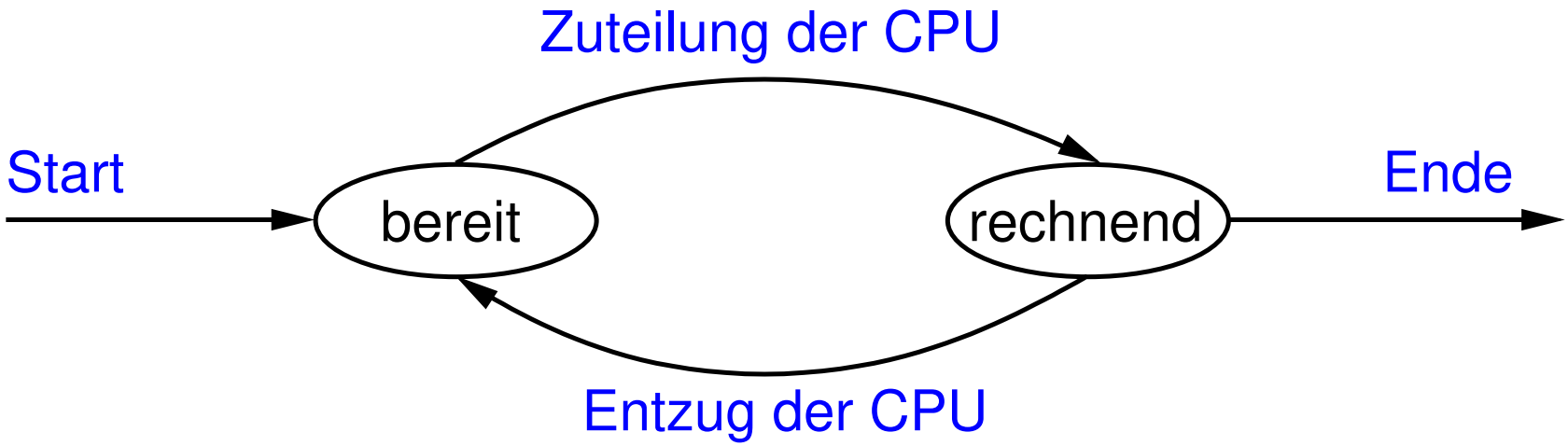
Ein einfaches Thread-Modell ...



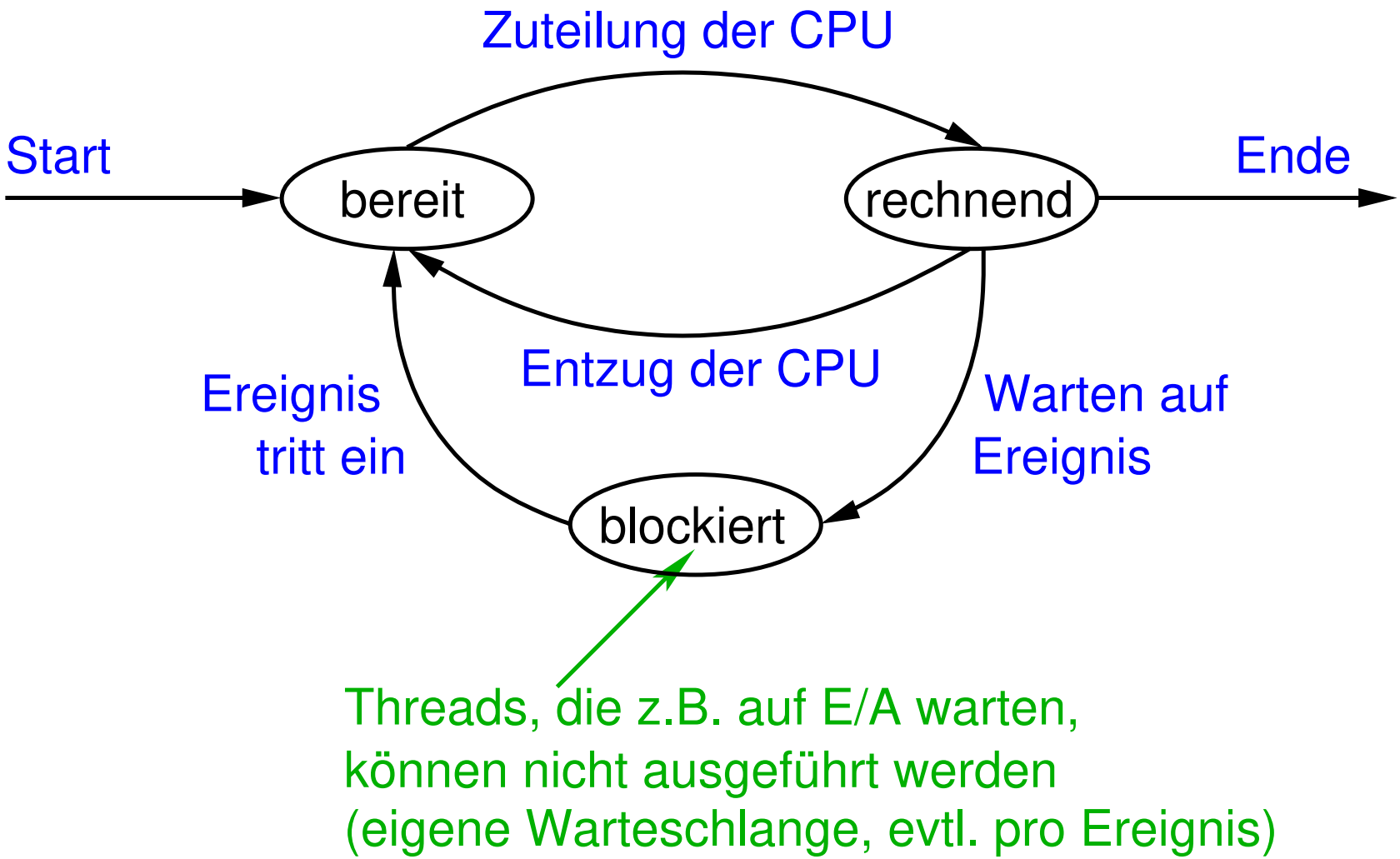
Zustandsgraph eines Threads



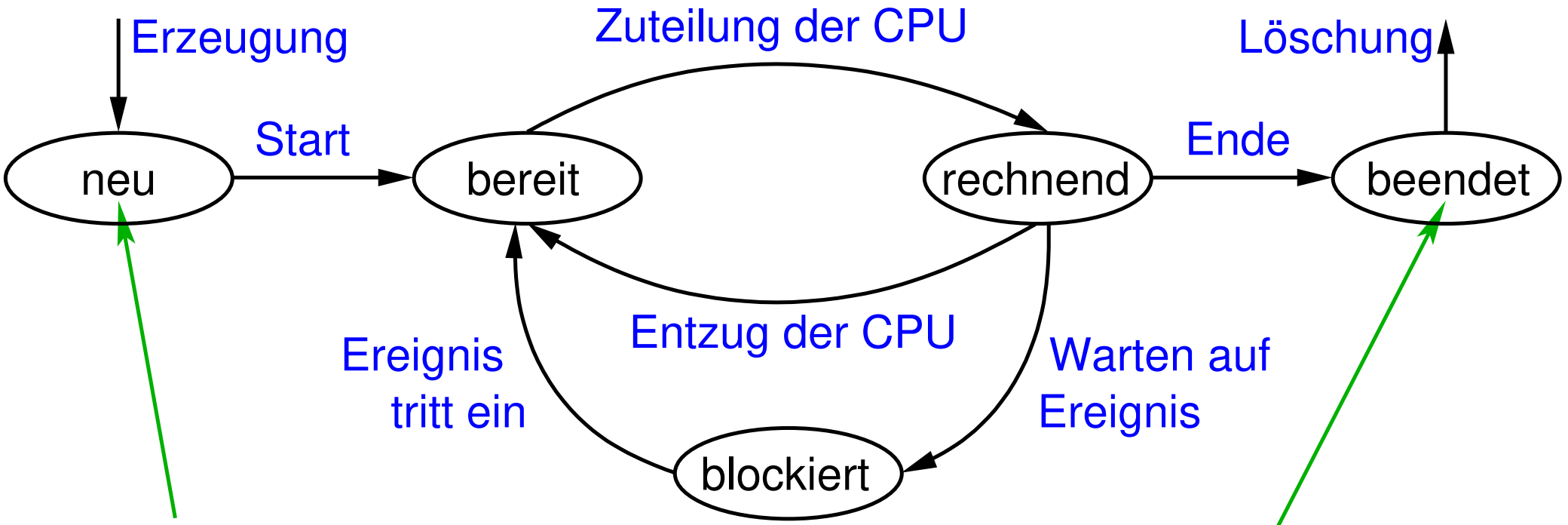
Zustandsgraph für ein erweitertes Thread-Modell



Zustandsgraph für ein erweitertes Thread-Modell



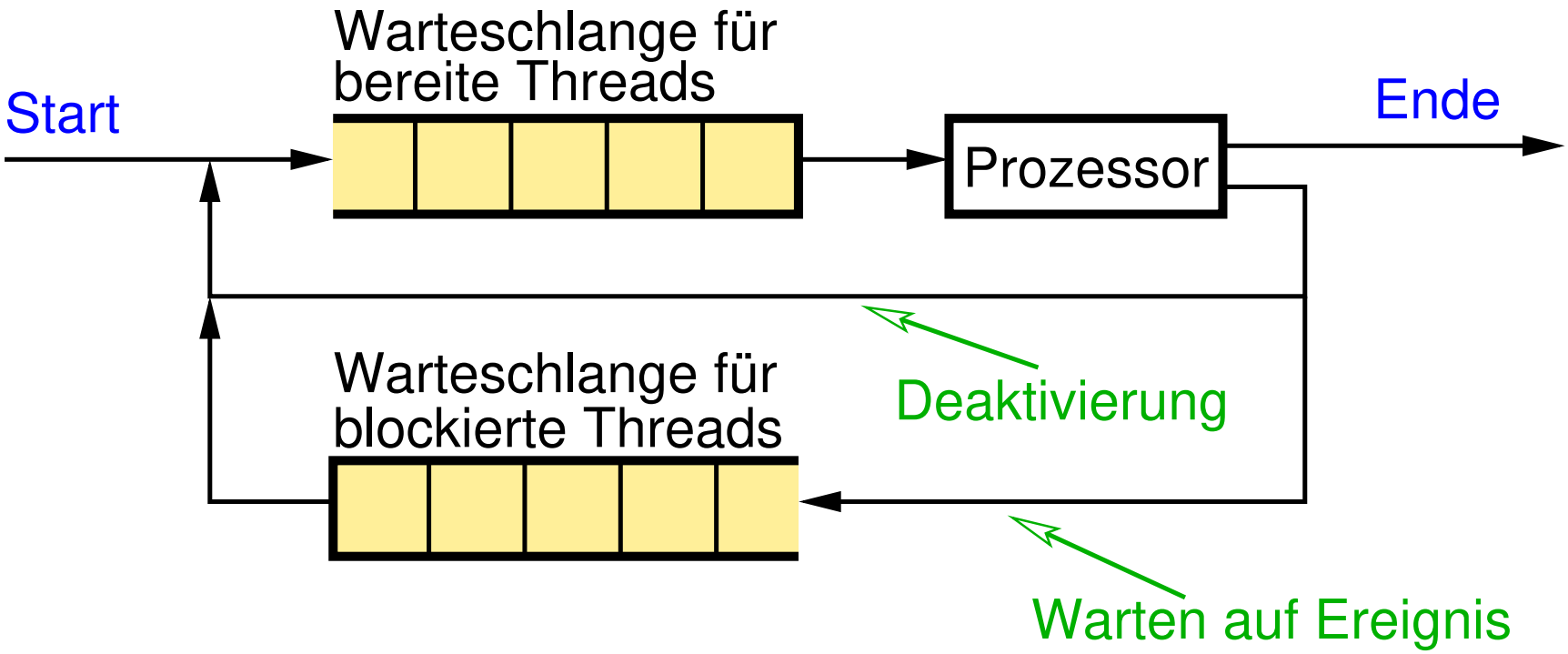
Zustandsgraph für ein erweitertes Thread-Modell ...



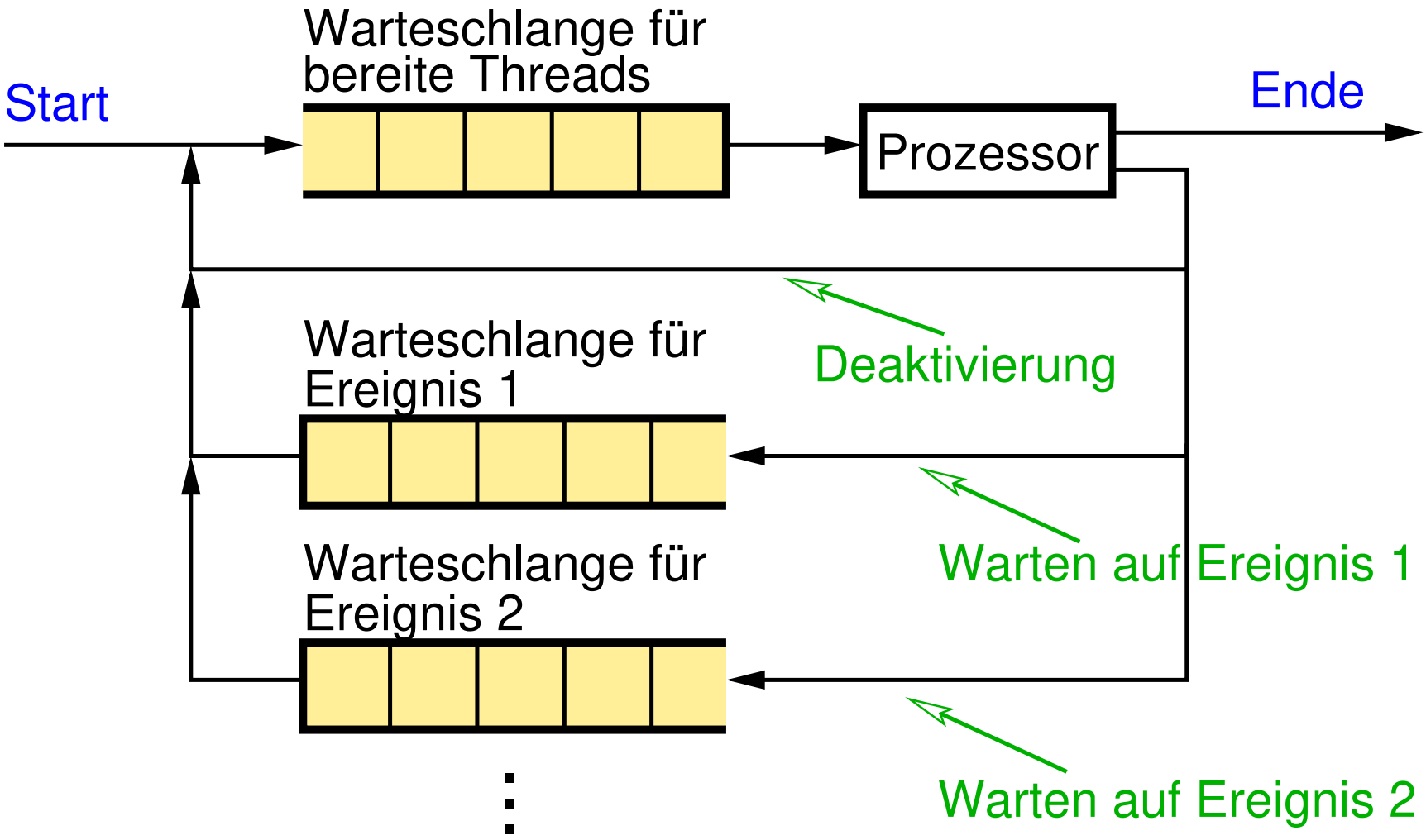
Die Verwaltungsdatenstruktur für den Thread ist bereits angelegt, der Thread selbst existiert aber noch nicht

Thread ist terminiert, Verwaltungsdaten sind noch vorhanden (z.B. zum Auslesen des Exit–Status)

Eine Warteschlange für alle blockierten Threads



Eine Warteschlange pro Ereignis

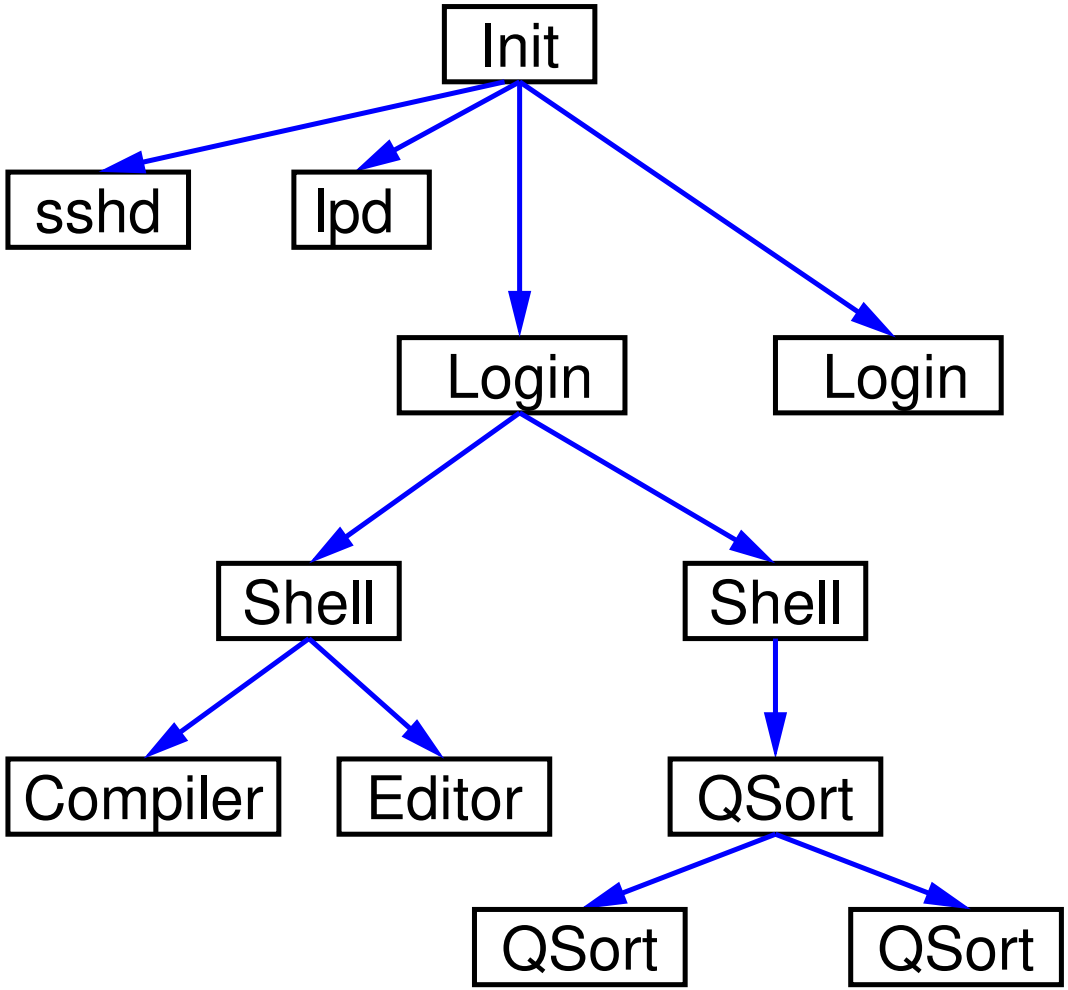


2.4 Erzeugung u. Terminierung von Prozessen

Gründe für Prozeßerzeugung:

- Initialisierung des Systems
  - Hintergrundprozesse (*Daemons*) für BS-Dienste
- Benutzeranfrage
  - Interaktive Anmeldung, Start eines Programms
- Erzeugung durch Systemaufruf eines bestehenden Prozesses
- Initiierung eines *Batch-Jobs*
  - in Mainframe-BSen
- Technisch wird ein neuer Prozeß (fast) immer durch einen Systemaufruf (z.B. `fork` bzw. `CreateProcess`) erzeugt
  - führt zu **Prozeßhierarchie**

Beispiel: Prozeßhierarchie unter UNIX



Initialisierungsprozeß

Daemons

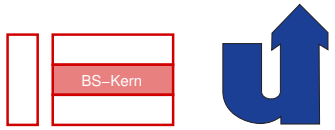
Für jeden Benutzer ein Login–Prozeß

Mehrere Kommando–interpreter (Shells) pro Benutzer

Anwendungen

Gründe für Prozeßterminierung:

- Freiwillig
  - durch Aufruf von z.B. `exit` bzw. `ExitProcess`
  - normal oder wegen Fehler
- Unfreiwillig (Abbruch durch BS)
  - wegen schwerwiegender Fehler, z.B. Speicherüberschreitung, Ausnahme, E/A-Fehler, Schutzverletzung
  - durch andere (berechtigte) Prozesse, über Systemaufruf (z.B. `kill` bzw. `TerminateProcess`)
  - Teilweise ist noch eine Reaktion des Prozesses auf das Ereignis möglich (☞ **4.4**: Signale)



- BS pflegt **Prozeßtabelle** mit Informationen über alle Prozesse
  - der Eintrag für einen Prozeß heißt **Prozeßkontrollblock**
- Analog: **Threadtabelle** für alle Threads
  - Eintrag: **Threadkontrollblock**
- Prozeßadressraum, Prozeßkontrollblock und Threadkontroll-blöcke beschreiben einen Prozeß vollständig
- Typische Elemente des Prozeßkontrollblocks:
  - Prozeßidentifikation, Zustands- und Ressourceninformation
- Typische Elemente des Threadkontrollblocks:
  - Threadidentifikation, Zustandsinformation
  - Scheduling- und Prozessorstatus-Information

Inhalt des Prozeßkontrollblocks

- Prozeßidentifikation
  - Kennung des Prozesses und des Elternprozesses
  - Benutzerkennung
  - Liste der Kennungen aller Threads
- Zustandsinformation
  - Priorität, verbrauchte CPU-Zeit, ...
- Verwaltungsinformation
  - Daten für Interprozeßkommunikation (☞ **3, 4**)
  - Prozeßprivilegien
  - Tabellen für Speicherabbildung (Speicherverwaltung, ☞ **8**)
  - Ressourcenbesitz und -nutzung
    - offene Dateien, Arbeitsverzeichnis, ...

Inhalt des Threadkontollblocks

- Threadidentifikation
  - Kennung des Threads
  - Kennung des zugehörigen Prozesses
- Scheduling- und Zustandsinformation
  - Threadzustand (bereit, rechnend, blockiert, ...)
  - ggf. Ereignis, auf das der Thread wartet
  - Priorität, verbrauchte CPU-Zeit, ...
- Prozessorstatus-Information
  - Datenregister
  - Steuer- und Statusregister: PC, PSW, ...
  - Kellerzeiger (SP)

Elemente von Prozessen und Threads

Elemente pro Prozeß	Elemente pro Thread
Adreßraum	Befehlszähler
geöffnete Dateien	Register
Kindprozesse	Keller*
Signale	Zustand (bereit, ...)
Privilegien	
...	
...	* genauer: Kellerzeiger

➥ (Bei Verwendung höherer Programmiersprachen:
lokale Variable sind pro Thread, globale pro Prozeß)

Ablauf einer Prozeßerzeugung

- Eintrag mit eindeutiger Kennung in Prozeßtabelle erzeugen
- Zuteilung von physischem Adreßraum an den Prozeß
  - für Programmcode, Daten, und Keller
  - (siehe später: **8**. Speicherverwaltung)
- Initialisierung des Prozeßkontrollblocks
  - Ressourcen evtl. von Elternprozeß geerbt
- Erzeugung und Initialisierung eines Threadkontrollblocks
  - PC und SP (und alle anderen Register)
  - Threadzustand: bereit
  - Prozeß startet mit genau einem Thread
- Einhängen des Threads in die Bereit-Warteschlange

Ablauf eines Threadwechsels

1. Prozessorstatus im Threadkontrollblock sichern
   - ➥ PC, PSW, SP und alle anderen Register
2. Thread- und Prozeßkontrollblock aktualisieren
   - ➥ Threadzustand, Grund der Deaktivierung, Buchhaltung, ...
3. Thread in entsprechende Warteschlange einreihen
4. Nächsten bereiten Thread auswählen (☞ **7**. Scheduling)
5. Threadkontrollblock des neuen Threads aktualisieren
   - ➥ Zustand auf „rechnend“ setzen
6. Falls neuer Thread in anderem Prozess liegt:
   - ➥ Aktualisierung der Speicherabbildung in der MMU (☞ **8**)
7. Prozessorstatus aus neuem Threadkontrollblock laden

Anmerkungen

- Beim Threadwechsel **innerhalb desselben Prozesses** entfällt die Aktualisierung der Speicherabbildung in der MMU
  - Threads im selben Prozeß haben gemeinsamen Adressraum
- BS (Scheduler) entscheidet direkt, welcher **Thread** als nächstes rechnen soll
  - falls nötig, wird dann auch der Prozeß mit umgeschaltet
  - Scheduler kann/sollte die Zuordnung von Threads zu Prozessen bei der Entscheidung berücksichtigen
    - z.B. wegen unterschiedlicher Kosten

Wann erfolgt ein Threadwechsel?

- Threadwechsel kann immer dann erfolgen, wenn BS die Kontrolle erhält:
  - bei Systemaufruf (z.B. E/A)
    - Thread gibt Kontrolle (d.h. Prozessor) freiwillig ab
  - bei Ausnahme (z.B. unzulässigem Befehl)
    - Prozeß wird ggf. beendet
    - evtl. auch Behandlung der Ausnahme durch BS
  - bei Interrupt (z.B. E/A-Gerät, Timer)
    - Behandlung des Interrupts erfolgt im BS
    - **periodischer Timer-Interrupt** stellt sicher, daß kein Thread die CPU monopolisieren kann

Ablauf beim Systemaufruf

(vgl. Folie 38)

- Durch Hardware: Einsprung ins BS (Systemmodus)
- Ablauf im BS:
  - Sichern des gesamten Prozessorstatus (1)
  - Ausführung bzw. Initiierung des Auftrags
    - dabei je nach Auftrag Thread in Zustand „bereit“ oder in Zustand „blockiert“ setzen (2,3)
  - Sprung zum Scheduler (4-7)
    - Aktivieren eines (anderen) Threads
      - dieser Thread wird nach Rückkehr in den Benutzer-modus fortgesetzt

(Die Zahlen in Klammern beziehen sich auf Folie 109)

Ablauf bei Ausnahme

(vgl. Folie 37)

- Durch Hardware: Einsprung ins BS (Systemmodus)
- Ablauf im BS:
  - Sichern des gesamten Prozessorstatus (1)
  - je nach Art der Ausnahme:
    - Beenden des Prozesses (2)
    - Blockieren des Threads (2,3)
      (z.B. bei Seitenfehler, ☞ **8.3.2**: dyn. Seitenersetzung)
    - Behebung der Ursache der Ausnahme
  - Sprung zum Scheduler (4-7)

(Die Zahlen in Klammern beziehen sich auf Folie 109)

Ablauf bei Interrupt

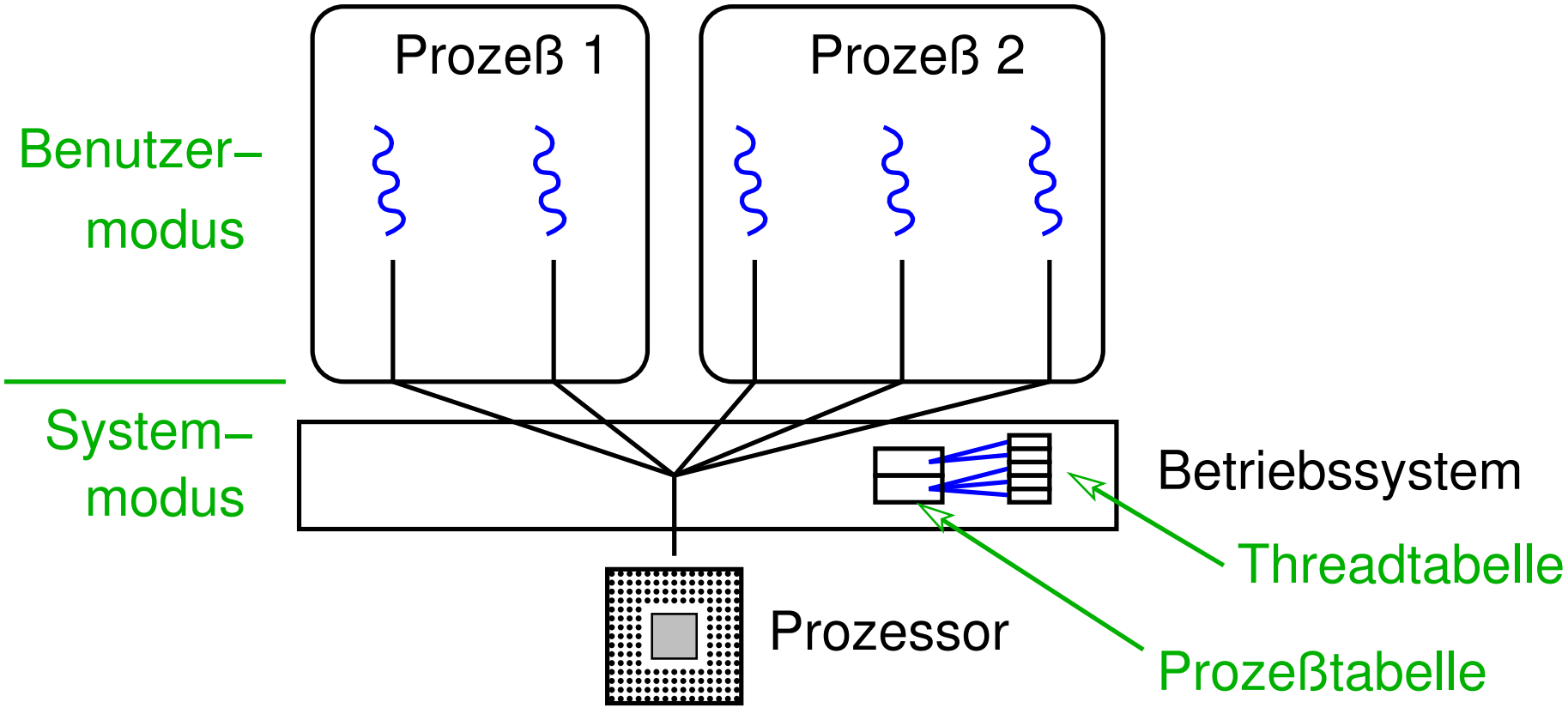
(vgl. Folie 39)

- Durch Hardware: Einsprung ins BS (Systemmodus)
- Ablauf im BS:
  - Sichern des gesamten Prozessorstatus (1)
  - aktuellen Thread auf „bereit“ setzen (2,3)
  - Ursache der Unterbrechung ermitteln
  - Ereignis (z.B. Ende der E/A) entsprechend behandeln
  - evtl. blockierte Threads wieder auf „bereit“ setzen
  - Sprung zum Scheduler (4-7)

(Die Zahlen in Klammern beziehen sich auf Folie 109)

2.6 Realisierungsvarianten für Threads

Realisierung durch BS-Kern

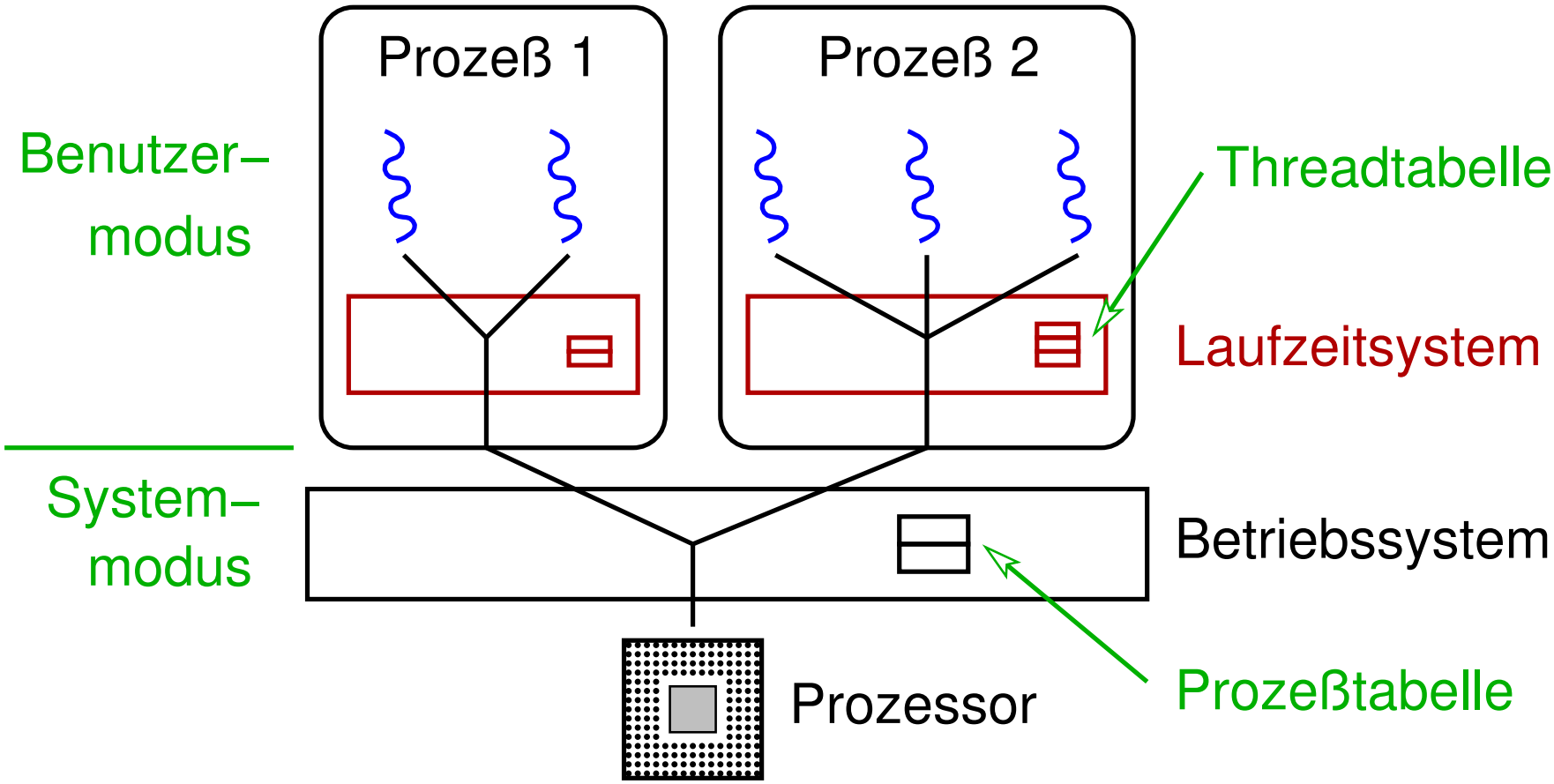


➥ Heute gängigste Realsierungsvariante

Realisierung durch BS-Kern: Diskussion

- Vorteile:
  - bei Blockierung eines Threads kann BS einen anderen Thread desselben Prozesses auswählen
  - bei Mehrprozessorsystemen: echte Parallelität innerhalb eines Prozesses möglich
- Nachteil: hoher Overhead
  - Threadwechsel benötigt Moduswechsel zum BS-Kern
  - Erzeugen, Beenden, etc. benötigt Systemaufruf

Realisierung im Benutzeradreßraum



➥ Genutzt in frühen Thread-Implementierungen

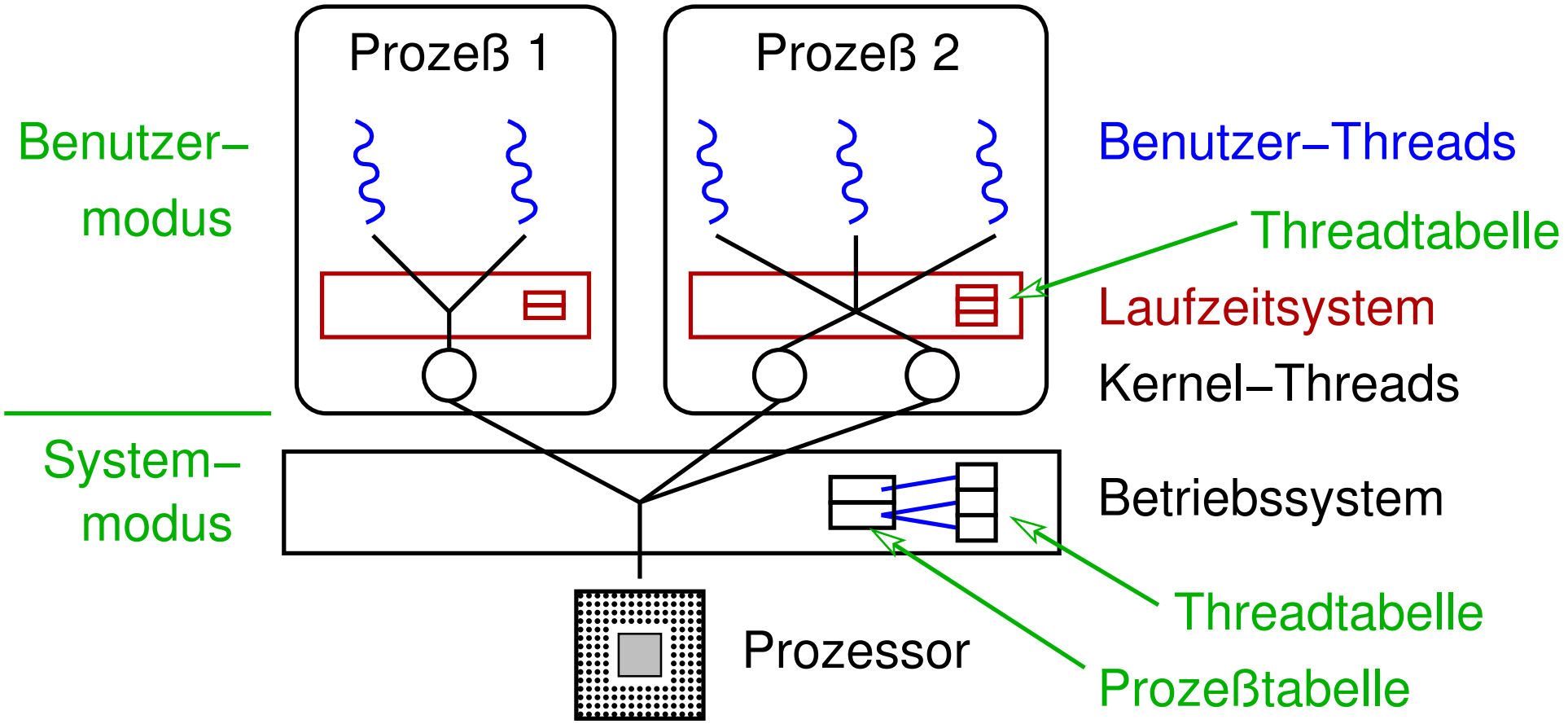
Realisierung im Benutzeradreßraum: Diskussion

- Vorteile:
  - keine Unterstützung durch BS notwendig
  - schnelle Threaderzeugung und Threadwechsel
    - z.B. Zeit für Erzeugung (Linux 4.19, Intel i7, 3.4 GHz)

Benutzer-Thread	Kernel-Thread	Prozess
0,03 μs	20 μs	200 μs

  - individuelle Scheduling-Algorithmen möglich
- Nachteile:
  - blockierender Systemaufruf blockiert alle Threads
    - macht eine Hauptmotivation für Threads zunichte
  - Threads müssen Prozessor i.d.R. freiwillig abgeben
    - Threadwechsel erfolgt durch Bibliotheksfunktion

Hybride Realisierung



➥ Sinnvoll für Programme mit sehr vielen nebenläufigen Aktivitäten

2.7 Schnittstellen zur Nutzung von Threads

- Alle heute gängigen BSe unterstützen Threads
- Anwendungen nutzen jedoch i.d.R. nicht die Systemaufrufe, sondern höhere Programmierschnittstellen
- Beispiele:
  - POSIX Threads (zusätzliche Programmierbibliothek für C)
  - C++-Threads (Teil der C++-Standardbibliothek)
  - Java Threads (in Sprache und Klassenbibliothek integriert)

2.7.1 Threads in Java

- Java unterstützt in der Sprache die Programmierung mit Threads
  - unterstützt durch die Java Klassen-Bibliothek
  - unabhängig vom Betriebssystem
- Programmiermodell:
  - bei Start des Programms: genau ein (Master-)Thread
  - Master-Thread erzeugt bei Bedarf weitere Threads
  - Prozess terminiert erst, wenn alle Thread beendet sind

Die Klasse `Thread`

- Die Methode `void run()` dieser Klasse kann nebenläufig in einem neuen Thread ausgeführt werden
- Typisch: eigene Klasse erbt von `Thread` und überschreibt die Methode `run()`:

```
public class MyThread extends Thread {
  public void run() {
    ... // nebenläufig auszuführender Code
  }
}
```

- Achtung: ein Objekt dieser Klasse ist (noch) **kein** BS-Thread
- Erzeugung eines BS-Threads:
  - Aufruf der Methode `void start()`
  - nur einmal pro Java-Objekt erlaubt

Weitere Methoden der Klasse `Thread` (unvollständig)

- `void join()`
  - wartet bis der BS-Thread seine Ausführung beendet hat
- `void setDaemon()`
  - markiert den Thread als Hintergrund-Thread
  - am Programmende wird der BS-Thread terminiert, ohne zu warten

Beispiel: Hello World!

```
public class MyThread extends Thread
{
  public void run() // Methode wird durch den neuen Thread abgearbeitet
  {
    System.out.println("Hello World!");
  }
  public static void main(String[] args)
         throws InterruptedException
  {
    MyThread t = new MyThread();   // Erzeuge Java–Objekt
    t.start();   // erzeuge neuen BS–Thread
    t.join();    // warte auf Beendigung des BS–Threads
  }
}
```

Besonderheiten

- Methode `run()` hat weder Argumente noch Rückgabewert
  - Datenaustausch mit dem Thread muß über Attribute erfolgen:
    - erzeuge Java-Objekt
      - initialisere Eingabe-Attribute im Konstruktor
    - starte BS-Thread (`start()`)
    - warte auf Beendigung des BS-Threads (`join()`)
    - lies Ergebnisse aus Attributen des Java-Objekts
- Erben von `Thread` nicht immer möglich (keine Mehrfachvererbung in Java)
  - daher auch Delegation möglich:
    - Konstruktor akzeptiert Objekt mit Schnittstelle `Runnable`
    - `run()` wird an dieses Objekt delegiert

Beispiel: Hello World 2!

```
public class Greeter implements Runnable
{
  private String whom;
  private int result;
  public Greeter(String whom) {
    this.whom = whom;
  }
  public int getResult() {
    return result;
  }
  public void run() // Methode wird durch den neuen Thread abgearbeitet
  {
    System.out.println("Hello " + whom + "!");
    result = 1;
  }
```

Beispiel: Hello World 2! ...

```
  public static void main(String[] args)
  {
    Greeter greet = new Greeter("World");
    Thread t = new Thread(greet); // Erzeuge Java–Objekt
    t.start();                    // erzeuge neuen BS–Thread
    try {
      t.join();                   // warte auf Beendigung des BS–Threads
    }
    catch (InterruptedException e) {
    }
    System.out.println("Result: " + t.getResult());
  }
}
```

2.7.2 Threads in C++

- Der C++ Sprachstandard enthält seit 2011 (C++-11) eine Threadschnittstelle
  - implementiert durch C++ Standard-Bibliothek
  - unabhängig vom Betriebssystem
- Programmiermodell:
  - bei Start des Programms: genau ein (Master-)Thread
  - Master-Thread erzeugt bei Bedarf weitere Threads und sollte auf deren Beendigung warten
  - Prozess terminiert, wenn Master-Thread terminiert
    - falls noch andere Threads laufen, wird eine Exception geworfen

Erzeugung von Threads

- Klasse `std::thread`
  - repräsentiert einen laufenden Thread
- Erzeugung eines neuen Threads:

  `std::thread myThread(`*`function, args ...`*`);`
  - erzeugt ein neues C++-Objekt und einen neuen BS-Thread
  - BS-Thread wird terminiert (mit Exception), wenn C++-Objekt deallokiert wird
    - bei obiger Deklaration am Ende des Gültigkeitsbereichs von `myThread`
  - *`function`*: Funktion, die der BS-Thread ausführen soll
    - kein Rückgabewert, aber Ergebnisparameter möglich
  - `args ...`: Argumente, die an *`function`* übergeben werden

Methoden der Klasse `thread` (unvollständig)

- `void join()`
  - wartet bis der Thread seine Ausführung beendet hat
  - nach Rückkehr dieser Methode kann das C++-Objekt gelöscht werden
- `void detach()`
  - koppelt den BS-Thread von dem C++-Objekt ab
  - d.h. der BS-Thread läuft auch dann noch weiter, wenn das C++-Objekt gelöscht wird
  - die Methode `void join()` kann nicht mehr aufgerufen werden
- Die Klasse überschreibt u.a. den Zuweisungsoperator
  - stellt sicher, daß beim Kopieren des C++-Objekts *kein* neuer BS-Tread erzeugt wird

Beispiel: Hello World!

```
#include <iostream>
#include <thread>

void greet(std::string whom)
{
    std::cout << "Hello " << whom << "!\n";
}

int main(int argc, char **argv)
{
    std::thread t(greet, "World"); // Erzeuge einen neuen Thread
    t.join();                      // Warte auf Beendigung
    return 0;
}
```

2.7.3 Threads in C

- PThreads (IEEE 1003.1c): Standard-Schnittstelle zur Programmierung mit Threads
  - implementiert als System-Bibliothek
  - (weitgehend) unabhängig vom Betriebssystem
- Programmiermodell
  - bei Start des Programms: genau ein (Master-)Thread
  - Master-Thread erzeugt bei Bedarf weitere Threads und sollte auf deren Beendigung warten
  - Prozess terminiert, wenn Master-Thread terminiert

Erzeugung von Threads

- `int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*function)(void *), void *arg)`
  - Eingabeparameter:
    - `attr`: Thread-Attribute
      - z.B. für Scheduling (i.a. BS-abhängig)
    - `function`: Funktion, die der Thread ausführen soll
    - `arg`: Argument, das an `function` übergeben wird
  - Ergebnisse:
    - Rückgabewert: Status (erfolgreich bzw. Fehler)
    - `*thread`: Zeiger auf (opake) Thread-Struktur

Thread Management (unvollständig)

- `void pthread_exit(void *retval)`
  - aufrufender Thread wird beendet (mit Rückgabewert `retval`)
- `int pthread_join(pthread_t thread, void **retval)`
  - wartet bis der gegebene Thread terminiert
  - gibt den Rückgabewert in `*retval` zurück
- `int pthread_cancel(pthread_t thread)`
  - sendet Terminierungsanfrage an den Thread
  - vor Terminierung: Aufruf eines *Cleanup-Handlers*
  - Thread kann Terminierungsanfragen ignorieren

Beispiel: Hello World!

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

void *greet(void *arg) {
  std::cout << "Hello " << (char*)arg << "!\n";
  return NULL;
}
int main(int argc, char **argv) {
  pthread_t t;
  // Erzeuge einen neuen Thread
  if (pthread_create(&t, NULL, greet, (void*)"World")!=0) {
    /* Fehlerbehandlung! */
  }
  pthread_join(t, NULL);   // Warte auf Beendigung
  return 0;
}
```

2.7.4 Ablauf einer Threaderzeugung (Linux)

1. Speicher für den (privaten) Keller des Threads anfordern
   - für Rückkehradressen, Argumente, lokale Variable, ...
   - typische Größe: 8 MiB*
   - Systemaufruf: `mmap` (legt Bereich im virtuellen Adreßraum an)
2. Kellerende durch schreib-/lesegeschützten Bereich sichern
   - bei Kellerüberlauf: Ausnahme
   - Systemaufruf: `mprotect` (Zugriffsrechte für Speicher setzen)
3. Neuen Thread erzeugen
   - Systemaufruf: `clone` (Prozess klonen)
   - hier: neuer „Prozess“ teilt sich alle Ressourcen mit dem alten, d.h. ist ein Thread
   - Parameter: Keller, aufzurufende Funktion und Argument

* Nach NIST: 1 KiB = 1024 Byte, 1 MiB = 1024 KiB, ...

2.8 Zusammenfassung / Wiederholung

- Zwei Aspekte:
  - Prozeß: Einheit der Ressourcenverwaltung, Schutzeinheit
  - Thread: Einheit der Prozessorzuteilung
    - pro Prozeß mehrere Threads möglich
- Thread-Schnittstellen:
  - Java Threads
- Threadmodell
  - Zustände „rechnend“, „bereit“, „blockiert“ + andere
  - Warteschlangen
- Zum Prozeß gehören u.a.:
  - Adreßraum, geöffnete Dateien, Signale, Privilegien, ...

2.8 Zusammenfassung / Wiederholung ...

- Zum Thread gehören u.a.:
  - Befehlszähler, CPU-Register, Keller(zeiger), Scheduling-Zustand, ...
- Threadwechsel:
  - Umladen des Prozessorkontexts
  - bei Prozeßwechsel auch Wechsel der Speicherabbildung
  - kann bei Systemaufruf, Ausnahme und Interrupt erfolgen

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

3 Synchronisation

Vorbemerkung

Klassen der Interaktion zwischen Threads (nach Stallings)

- Threads kennen sich gegenseitig nicht
  - nutzen aber gemeinsame Ressourcen (Geräte, Dateien, ...)
    - unbewußt (**Wettstreit**)
    - bewußt (**Kooperation durch Teilen**)
  - wichtig: **Synchronisation** (☞ **3**)
- Threads kennen sich (d.h. die Prozeß-/Threadkennungen)
  - **Kooperation durch Kommunikation** (☞ **4**)
- **Anmerkungen**:
  - Threads können ggf. in unterschiedlichen Prozessen liegen
  - in der Literatur hier i.a. keine klare Unterscheidung zwischen Threads und Prozessen

Inhalt (1):

- Einführung und Motivation
- Wechselseitiger Ausschluß
- Wechselseitiger Ausschluß mit aktivem Warten
  - Lösungsversuche, korrekte Lösungen
- Synchronisation in Mehrprozessorsystemen
- Semaphore

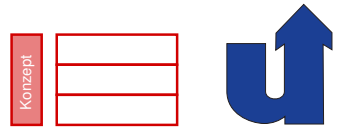
- Tanenbaum 2.3.1-2.3.6
- Stallings 5.1-5.4.1

Inhalt (2):

- Klassische Synchronisationsprobleme
  - Erzeuger/Verbraucher-Problem
  - Leser/Schreiber-Problem
- Monitore
- Schnittstellen zur Thread-Synchronisation
- Speicherkonsistenz
- *Lock-Free* Datenstrukturen
- *Transactional Memory*

- Tanenbaum 2.4.2, 2.4.3, 2.3.7
- Stallings 5.4.4, 5.5

3.1 Einführung und Motivation



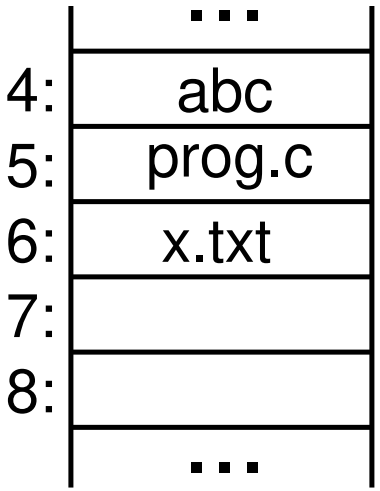
- Mehrprogrammbetrieb führt zu Nebenläufigkeit
  - Abarbeitung im Wechsel (praktisch) äquivalent zu echt paralleler Abarbeitung
- Mehrere Threads können gleichzeitig versuchen, auf gemeinsame Ressourcen zuzugreifen
  - Beispiel: Drucker
- Für korrekte Funktion in der Regel notwendig:
  - zu einem Zeitpunkt darf nur jeweils einem Thread der Zugriff erlaubt werden

Beispiel: Drucker-Spooler

- Threads tragen zu druckende Dateien in Spool-Bereich ein:
  - Spooler-Verzeichnis mit Einträgen 0, 1, 2, ... für Dateinamen
  - zwei gemeinsame Variable:
    - `out`: nächste zu druckende Datei
    - `in`: nächster freier Eintrag
  - in gemeinsamem Speicherbereich oder im Dateisystem
- Druck-Thread überprüft, ob Aufträge vorhanden sind und druckt die Dateien

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich



in:	7
out:	4

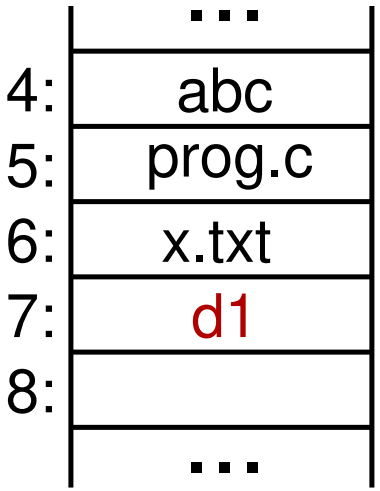
Thread A

```
...
s[in]="d1";
```

Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich



in:	7
out:	4

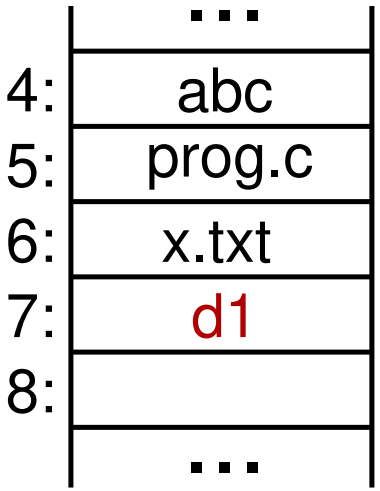
Thread A

```
...
s[in]="d1";
```

Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich



in:	7
out:	4

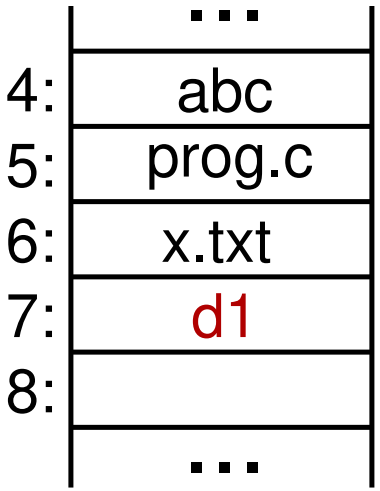
Thread A

```
...
s[in]="d1";
in=in+1;
```

Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich



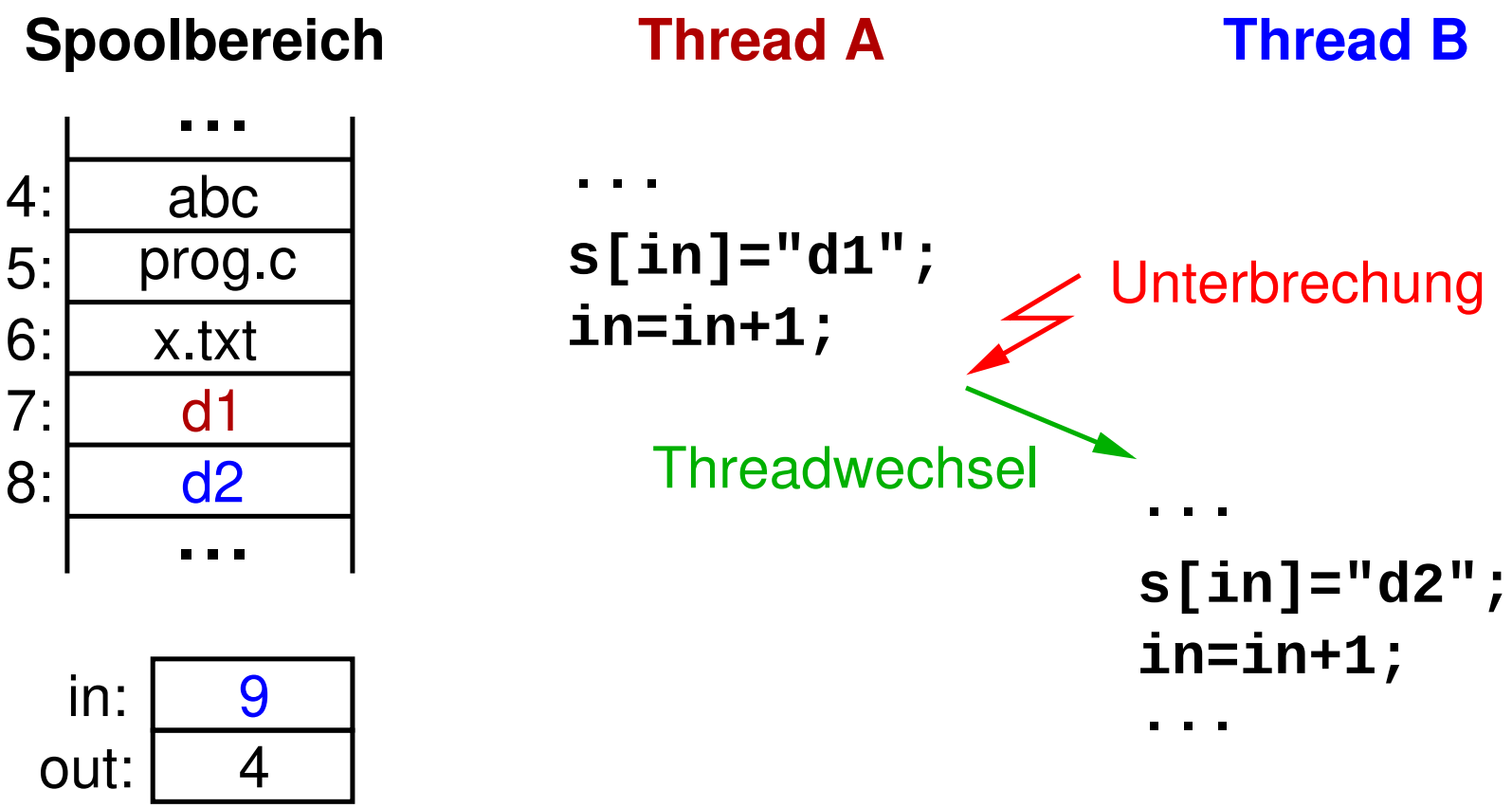
in:	8
out:	4

Thread A

```
...
s[in]="d1";
in=in+1;
```

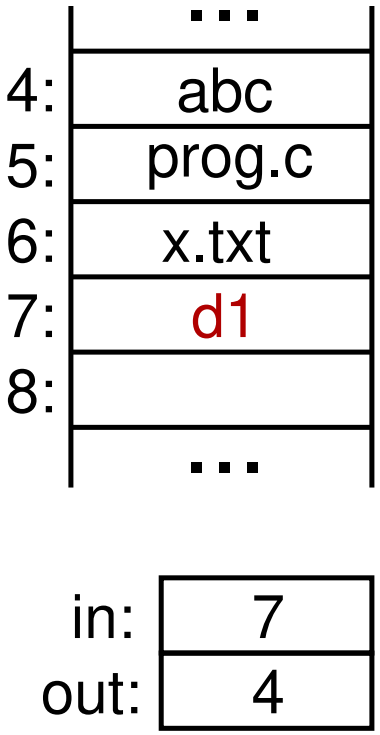
Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf



Beispiel: Drucker-Spooler, fehlerhafter Ablauf

Spoolbereich

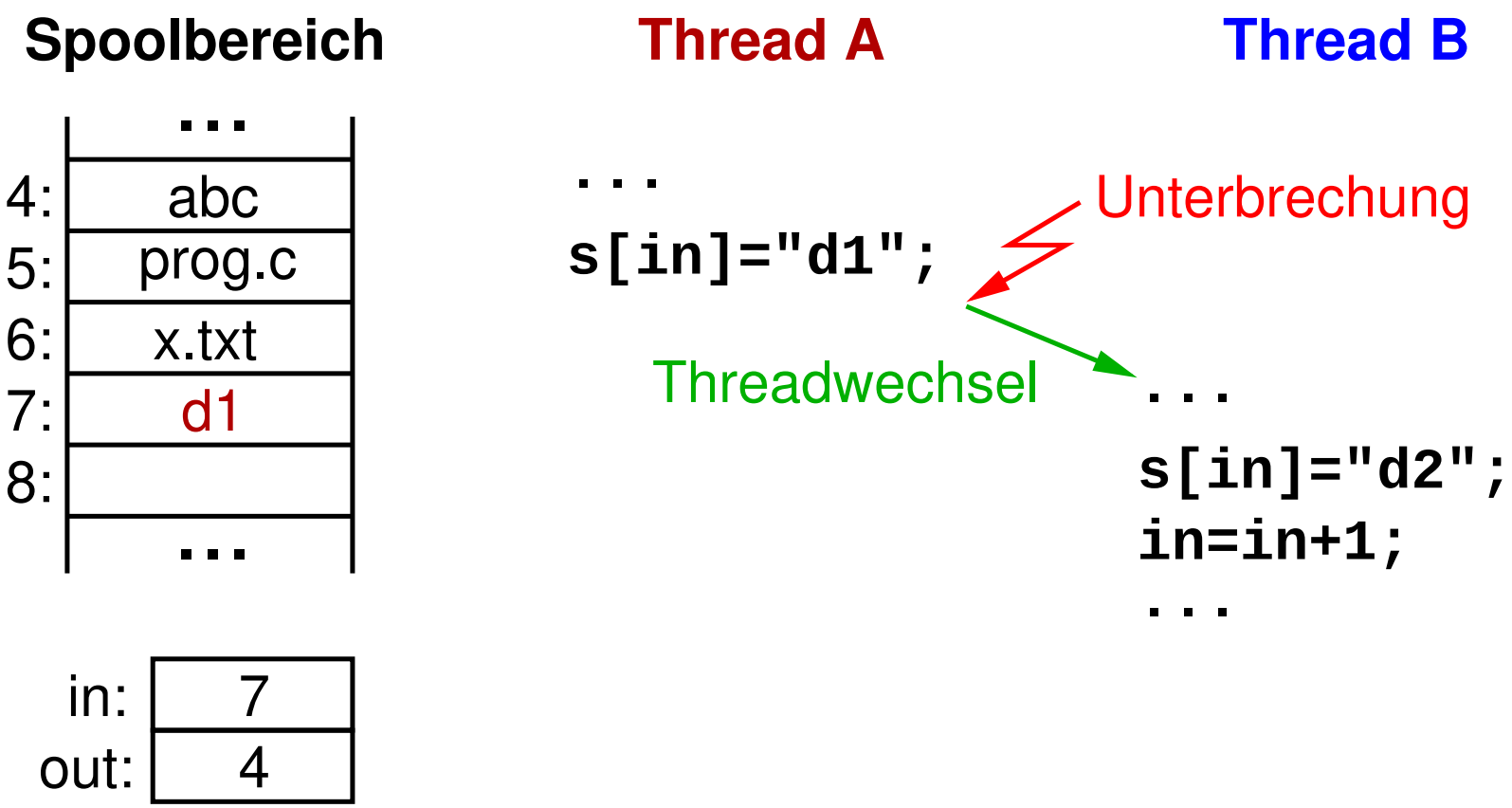


Thread A

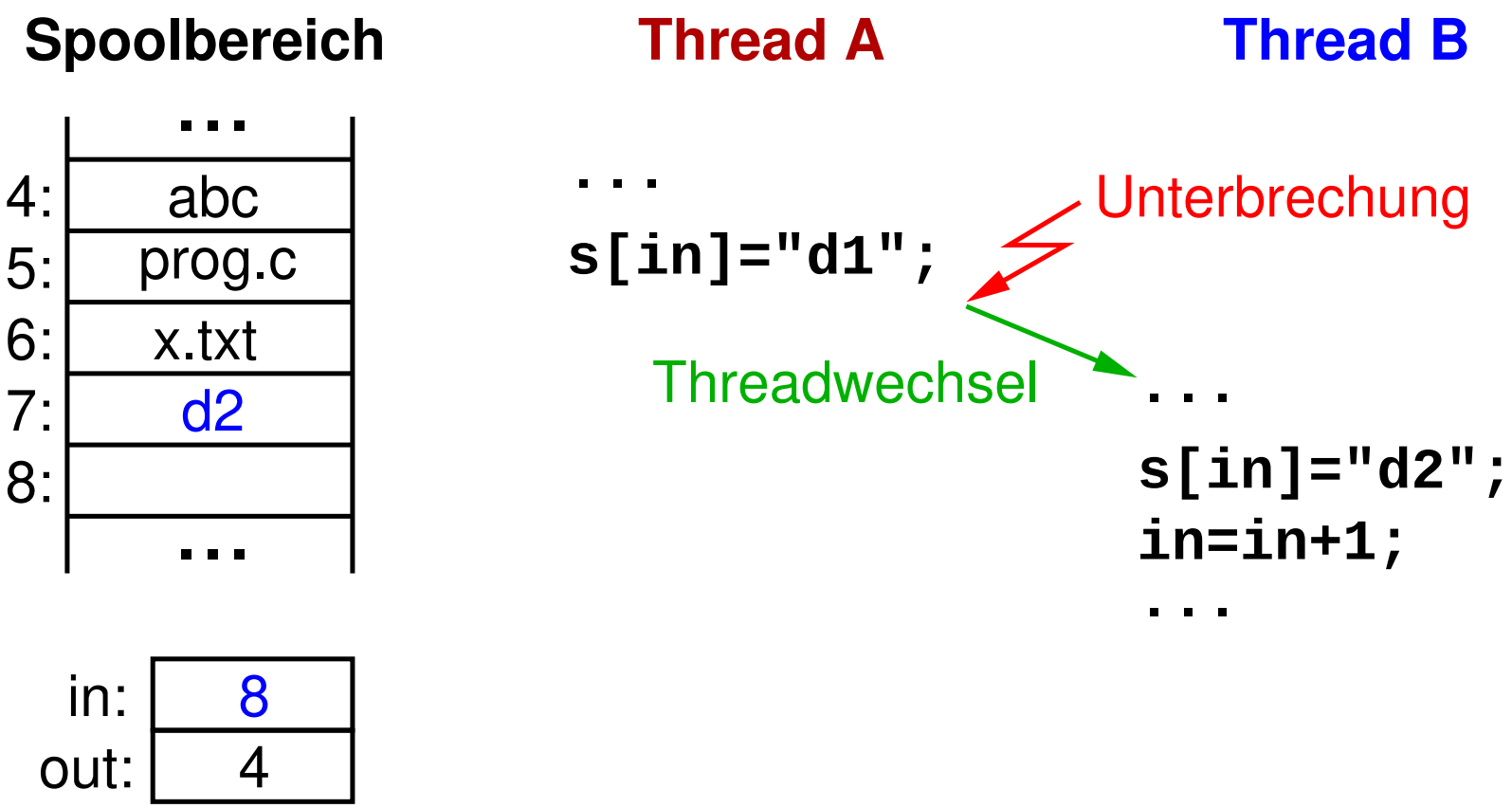
```
...
s[in]="d1";
```

Thread B

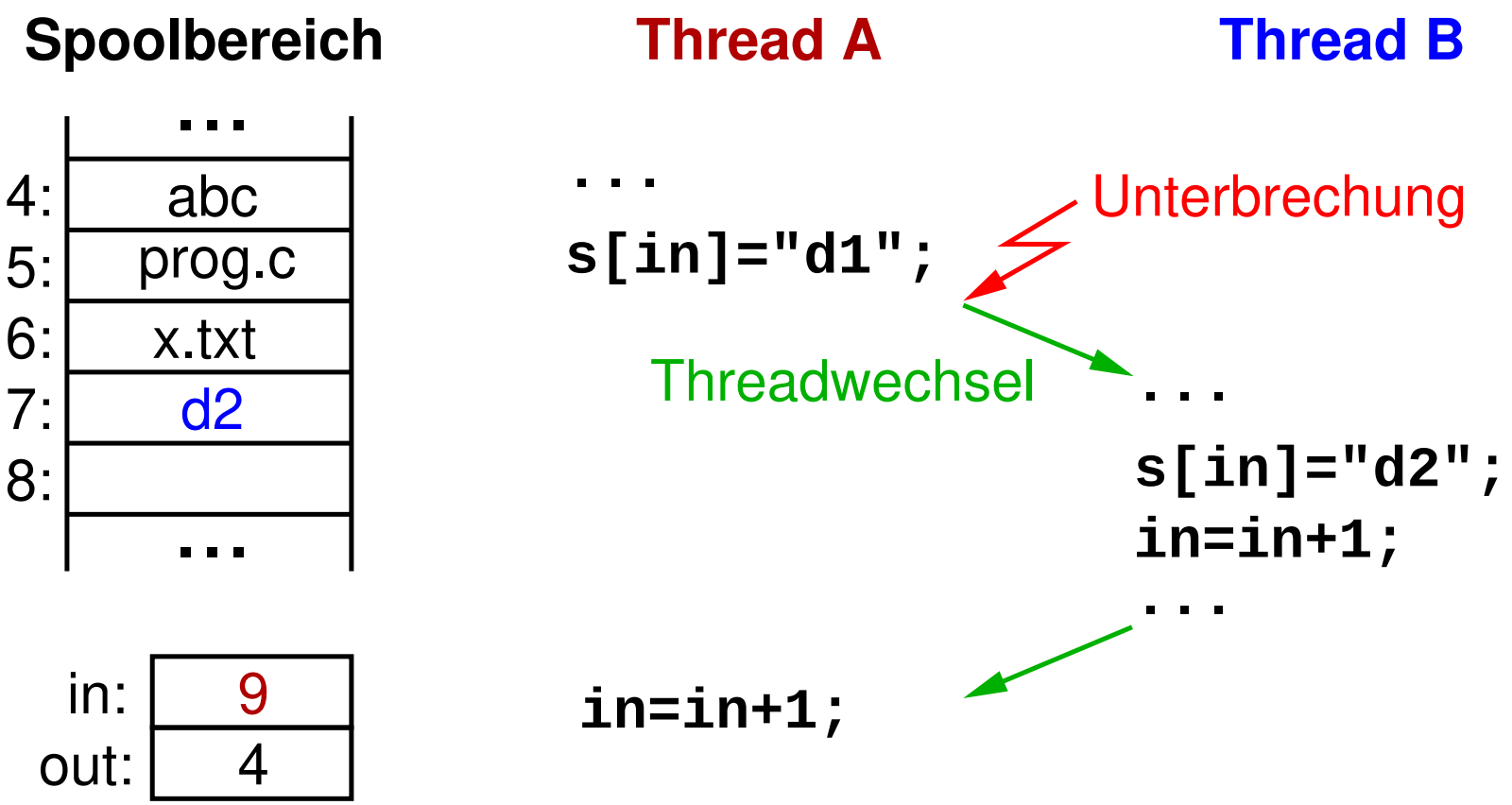
Beispiel: Drucker-Spooler, fehlerhafter Ablauf



Beispiel: Drucker-Spooler, fehlerhafter Ablauf



Beispiel: Drucker-Spooler, fehlerhafter Ablauf



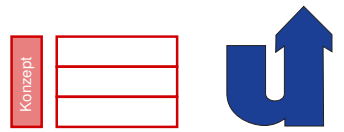
➥ ***Race Condition***

Arten der Synchronisation

- **Sperrsynchronisation** (**wechselseitiger Ausschluß**)
  - stellt sicher, daß Aktivitäten in verschiedenen Threads **nicht gleichzeitig** ausgeführt werden
  - d.h., die Aktivitäten werden nacheinander (in beliebiger Reihenfolge) ausgeführt
  - z.B. kein gleichzeitiges Drucken

- **Reihenfolgesynchronisation**
  - stellt sicher, daß Aktivitäten in verschiedenen Threads in einer **bestimmten Reihenfolge** ausgeführt werden
  - z.B. erst Datei erzeugen, dann lesen

3.2 Reihenfolgesynchron. mit aktivem Warten



- Beispiel: Thread 1 darf eine Datei erst öffnen, nachdem Thread 0 sie erzeugt hat
- Idee: Nutzung einer Boole'schen Variable
  - zeigt an, ob Wartebedingung erfüllt ist
  - Warten erfolgt durch eine (leere) Schleife, die die Bedingung laufend testet (**aktives Warten**)
- Im Beispiel:

  Initialisierung:

```
ready = false; // zeigt an, ob Datei erzeugt wurde
```

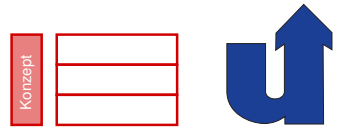
  Thread 0

```
// Datei erzeugen
ready = true;
```

  Thread 1

```
while (!ready); // aktives Warten
Datei öffnen
```

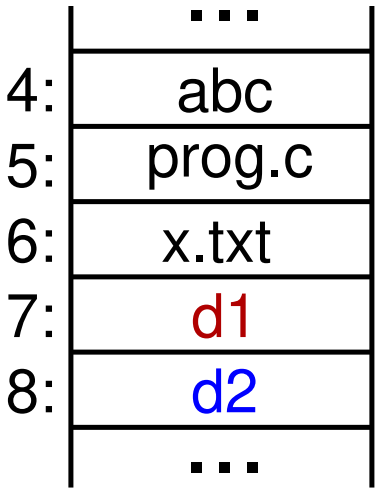
3.3 Wechselseitiger Ausschluß



- **Kritischer Abschnitt**
  - Abschnitt eines Programms, der Zugriffe auf ein gemeinsam genutztes Objekt (**kritische Ressource**) enthält
- **Wechselseitiger Ausschluß** von Aktivitäten
  - zu jeder Zeit darf nur ein Thread die Aktivität ausführen
  - Sperrsynchronisation
- Gesucht: Methode zum wechselseitigen Ausschluß kritischer Abschnitte

Beispiel: Drucker-Spooler mit wechselseitigem Ausschluß

Spoolbereich



Thread A

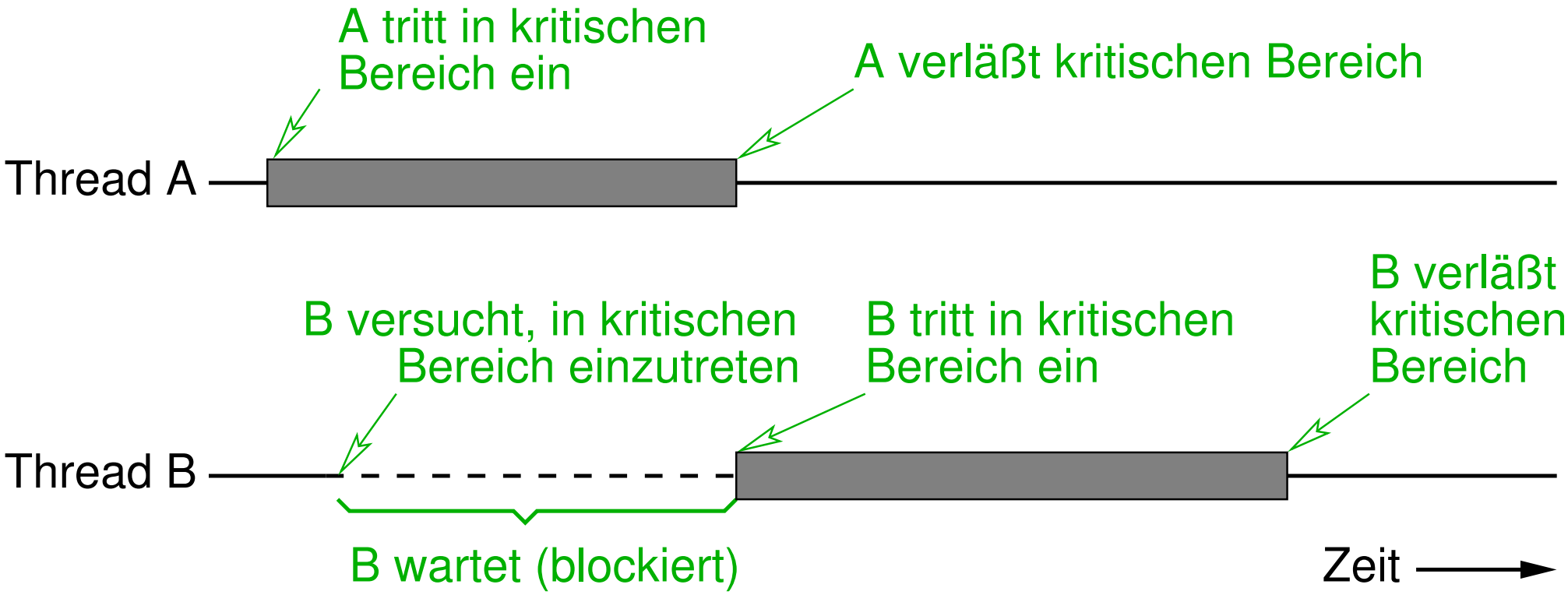
```
begin_region();
s[in]="d1";
in=in+1;
end_region();
```

Thread B

```
begin_region();
s[in]="d2";
in=in+1;
end_region();
```

➥ Frage: Implementierung von `begin_region()` / `end_region()`?

Idee des wechselseitigen Ausschlusses



Anforderungen an Lösung zum wechselseitigen Ausschluß:

1. Höchstens ein Thread darf im kritischen Abschnitt (k.A.) sein
2. Keine Annahmen über Geschwindigkeit / Anzahl der CPUs
3. Thread außerhalb des k.A. darf andere nicht behindern
4. Kein Thread sollte ewig warten müssen, bis er in k.A. eintreten darf
   - ➥ Voraussetzung: kein Thread bleibt ewig im k.A.
5. Sofortiger Zugang zum k.A., wenn kein anderer Thread im k.A. ist

Lösungsversuch 1: Sperren der Interrupts

- Abgesehen von freiwilliger Abgabe der CPU: Threadwechsel nur durch Interrupt
- Sperren der Interrupts in `begin_region()`, Freigabe in `end_region()`
- Probleme:
  - Ein-/Ausgabe ist blockiert
  - BS verliert Kontrolle über den Thread
  - Funktioniert nur bei Einprozessor-Rechnern
- Anwendung aber im BS selbst

Lösungsversuch 2: Sperrvariable

- Variable `belegt` zeigt an ob kritischer Abschnitt belegt
- Thread 0

```
while(belegt);          } begin_region()
belegt = true;          }
// kritischer Abschnitt
belegt = false          } end_region()
```

- Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:

Lösungsversuch 2: Sperrvariable

- Variable `belegt` zeigt an ob kritischer Abschnitt belegt
- Thread 0

```
while(belegt);          } begin_region()
belegt = true;          }
// kritischer Abschnitt
belegt = false          } end_region()
```

- Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:
  - Threads führen gleichzeitig `begin_region()` aus
  - lesen gleichzeitig `belegt`
  - finden `belegt` auf `false`

Lösungsversuch 2: Sperrvariable

- Variable `belegt` zeigt an ob kritischer Abschnitt belegt
- Thread 0

```
while(belegt);          } begin_region()
belegt = true;          }
// kritischer Abschnitt
belegt = false          } end_region()
```

- Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:
  - Threads führen gleichzeitig `begin_region()` aus
  - lesen gleichzeitig `belegt`
  - finden `belegt` auf `false`
  - setzen `belegt=true`

Lösungsversuch 2: Sperrvariable

- Variable `belegt` zeigt an ob kritischer Abschnitt belegt
- Thread 0

```
while(belegt);          } begin_region()
belegt = true;          }
// kritischer Abschnitt
belegt = false          } end_region()
```

- Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:
  - Threads führen gleichzeitig `begin_region()` aus
  - lesen gleichzeitig `belegt`
  - finden `belegt` auf `false`
  - setzen `belegt=true` und betreten kritischen Abschnitt!!!

Lösungsversuch 3: Strikter Wechsel

- Variable `turn` gibt an, wer an der Reihe ist
- Thread 0

```
while (turn != 0);
// kritischer Abschnitt
turn = 1;
```

Thread 1

```
while (turn != 1);
// kritischer Abschnitt
turn = 0
```

- Problem:
  - Threads **müssen** abwechselnd in kritischen Abschnitt
  - verletzt Anforderungen 3, 4, 5

Lösungsversuch 4: Erst belegen, dann prüfen

- Variable `interested[i]` zeigt an, ob Thread `i` in den kritischen Abschnitt will
- Thread 0

```
interested[0] = true;
while (interested[1]);
// kritischer Abschnitt
interested[0] = false;
```

Thread 1

```
interested[1] = true
while (interested[0]);
// kritischer Abschnitt
interested[1] = false
```

- Problem:
  - Verklemmung, falls Threads `interested` gleichzeitig auf `true` setzen

Eine richtige Lösung: Peterson-Algorithmus

- Thread 0

```
interested[0] = true;
turn = 1;
while ((turn != 0) &&
       interested[1]);
// kritischer Abschnitt
interested[0] = false;
```

Thread 1

```
interested[1] = true;
turn = 0;
while ((turn != 1) &&
       interested[0]);
// kritischer Abschnitt
interested[1] = false
```

- Verklemmung wird durch `turn` verhindert
- Jeder Thread bekommt die Chance, den kritischen Bereich zu betreten
  - keine **Verhungerung**

Zur Korrektheit des Peterson-Algorithmus

- **Wechselseitiger Ausschluß**:
  - Widerspruchsannahme: T0 und T1 beide im k.A.
  - damit: `interested[0]=true` und `interested[1]=true`
  - falls `turn=1`:
    - da T0 im k.A.: in der `while`-Schleife muß `turn==0` oder `interested[1]==false` gewesen sein
    - falls `turn==0` war: Widerspruch! (wer hat `turn=1` gesetzt?)
    - falls `interested[1]==false` war:
      T1 hat `interested[1]=true` noch nicht ausgeführt, hätte also später `turn==0` gesetzt und blockiert, Widerspruch!
  - falls `turn=0`:
    - symmetrisch!

Zur Korrektheit des Peterson-Algorithmus

- **Verklemmungs- und Verhungerungsfreiheit**:
  - Annahme: T0 dauernd in `while`-Schleife blockiert
  - Damit: immer `turn=1` und `interested[1]=true`
  - Mögliche Fälle für T1:
    - T1 will nicht in k.A.: `interested[1]` wäre `false`!
    - T1 wartet in Schleife: geht nicht wegen `turn==1` !
    - T1 ist immer im k.A.: nicht erlaubt!
    - T1 kommt immer wieder in k.A.: geht nicht, da T1 `turn=0` setzt, damit kann aber T0 in k.A.!
  - In allen Fällen ergibt sich ein Widerspruch

Lösungen mit Hardware-Unterstützung

- Problem bei den Lösungsversuchen:
  - Abfragen und Ändern einer Variable sind zwei Schritte
- Lösung: **atomare** *Read-Modify-Write* Operation der CPU
  - z.B. Maschinenbefehl *Test-and-Set*

```
boolean testAndSet(boolean &var) { // var: Referenzparameter
    boolean tmp = var; var = true; return tmp;
}
```

  - ununterbrechbar, auch in Multiprozessorsystemen unteilbar
- Lösung mit *Test-and-Set*:

```
while (testAndSet(belegt));
// kritischer Abschnitt
belegt = false;
```

Aktives Warten (*Busy Waiting*)

- In bisherigen Lösungen: Warteschleife (***Spinlock***)
- Probleme:
  - Thread belegt CPU während des Wartens
  - Bei Einprozessorsystem und Threads mit Prioritäten sind Verklemmungen möglich:
    - Thread H hat höhere Priorität wie L, ist aber blockiert
    - L rechnet, wird in kritischem Abschnitt unterbrochen; H wird rechenbereit
    - H will in kritischen Abschnitt, wartet auf L; L kommt nicht zum Zug, solange H rechenbereit ist ...
- Notwendig bei Multiprozessorsystemen
  - für kurze kritische Abschnitte im BS-Kern

Read-Modify-Write Befehle

- Sind auf einer CPU atomar, da Unterbrechungen nur zwischen Befehlen auftreten
- In Mehrprozessorsystemen:
  - Befehl benötigt zwei Speicherzugriffe (Lesen, Schreiben)
  - andere CPU kann dazwischen auf den Speicher zugreifen
- Daher: Unterstützung durch Speichersystem nötig
  - Bus bzw. Speicher kann über mehrere Zugriffe hinweg gesperrt werden (wechselseitiger Ausschluß!)
- Wechselseitiger Ausschluß in Mehrprozessorsystemen damit im Endeffekt immer durch Bus- bzw. Speicherarbiter realisiert

Spin-Locks und Caches

- Verbleibendes Problem bei *Spin-Locks*:
  - extreme Belastung des Busses / Speichers
  - trotz Caches!
- *Test-and-Set* ist eine **schreibende** Operation
  - Cache-Kohärenz-Protokolle führen zur Invalidierung aller anderen Kopien bei jedem *Test-and-Set*
- Während des Wartens wird der betroffene Cache-Block somit laufend invalidiert
  - hohe Bus-Belastung durch Invalidierung und Neu-Laden

Test and Test-and-Set

- Idee: einfache Abfrage des Locks vor dem Sperrversuch mit *Test-and-Set*

```
while (belegt || (testAndSet(belegt));
// kritischer Abschnitt
belegt = false;
```

- Während des Wartens wird `belegt` nur gelesen (aus dem Cache)
  - Verbesserung gegenüber einfacher Sperre
- Bei Freigabe: Invalidierung der Caches
  - mehrere Threads können `belegt == false` sehen und versuchen, mit `testAndSet()` die Sperre zu bekommen
  - dadurch viele weitere (überflüssige) Invalidierungen

Exponential Backoff

- Weitere Möglichkeit: zeitliche Entzerrung der Sperrversuche
  - Warteschleife zwischen zwei Abfragen der Sperre
  - falls Sperre belegt: Wartezeit verdoppeln (bis zu einem Maximum)
- Ggf. kombiniert mit *Test and Test-and-Set*
- Busbelastung wird (auch bei Freigabe) geringer
- Aber: erhöhte Reaktionszeit bei Freigabe der Sperre

Load Linked / Store Conditional

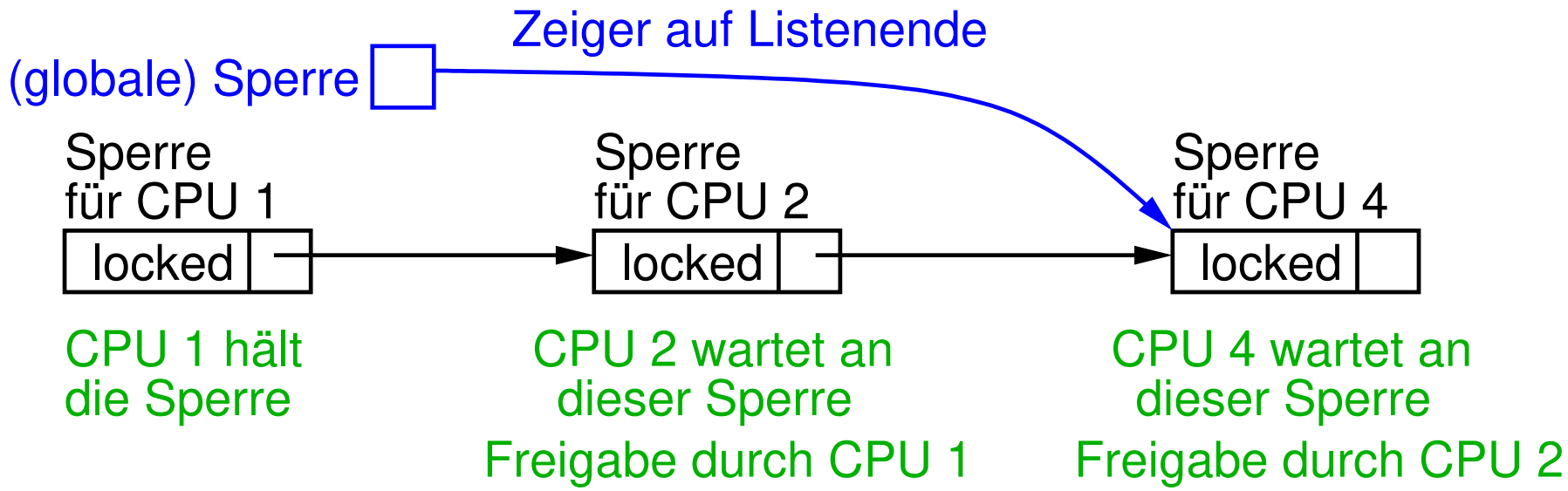
- Spezielle Maschinenbefehle zur Realisierung von *Spin Locks*
- *Load Linked*
  - Laden aus dem Speicher (bzw. Cache)
  - Adresse wird in *Link Register* vermerkt
  - *Link Register* wird bei Invalidierung der Cachezeile gelöscht
- *Store Conditional*
  - speichert einen Wert, falls die Adresse mit der im *Link Register* übereinstimmt
- Maximal eine Invalidierung bei freiwerdender Sperre

Code:

```
      li r2,#1
wt:   ll r1,locked
      bnz wt
      sc locked,r2
      bz wt
      ... ;k.A.
      st locked,#0
```

Listenbasierte Sperren (Mellor-Crummey / Scott)

- Idee: Sperre als Liste mit lokalen Sperr-Variablen realisiert



- Falls gesperrt: CPU fügt Listenelement mit lokaler Sperre an
  - jede CPU wartet nur an ihrer lokalen Sperre
- Belegen / Freigeben des Locks mit $\mathcal{O}(1)$ Speichertransaktionen
- Garantiert FIFO-Reihenfolge

3.5 Synchronisation in Mehrprozessorsystemen ...

```
Node l;                                          // Globale Variable: Listenende
void lock(Node n) {                              // n = eigener Knoten
  n.next = null;
  Node pred = fetchAndStore(l,n);                // pred = l; l = n, Einfügen (Teil 1)
  if (pred != null) {                            // falls Sperre belegt (Liste nicht leer)
    n.locked = true;
    pred.next = n;                               // Einfügen in Liste (Teil 2)
    while (n.locked);                            // Warte auf Freigabe
  }
}
void unlock(Node n) {                            // n = eigener Knoten
  if (n.next == null) {                          // kein Nachfolger?
    if (compareAndSet(l, n, null))               // falls noch l == n ist: l = null
      return;                                    //    und fertig
    while (n.next == null);                      // sonst: warten bis Nachfolger
  }                                              //    eingetragen wurde
  n.next.locked = false;                         // Freigabe an Nachfolger
}
```

Aktives Warten oder Blockierung des Threads?

- Aktives Warten in Multiprozessorsystemen möglich / sinnvoll
  - in Einprozessorsystemen extrem schlechte Leistung bzw. Verklemmung möglich
- In einigen Fällen ist aktives Warten unvermeidlich
  - innerhalb des Betriebssystems
- Threadwechsel bedeutet immer zusätzlichen Overhead
  - Systemaufruf, Umladen der CPU-Register, Caches, ...
- Optimale Entscheidung daher abhängig von mittlerer Wartezeit
- Praxis: warte einige Zeit aktiv, dann Blockierung des Threads

Reihenfolgesynchronisation über eine gemeinsame Variable

- Typisches Beispiel:

  Initialisierung:

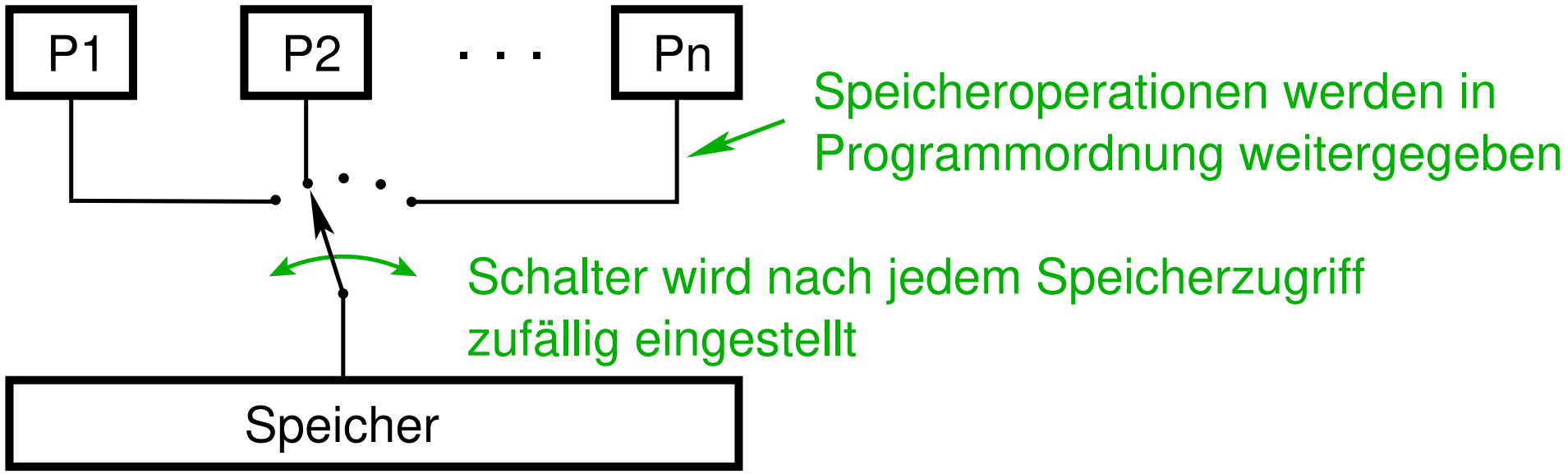
```
double value = 0;
boolean ready = false; // ist 'value' gültig?
```

Thread 0	Thread 1
`value = 0.5*2;`	`while (!ready); // aktives Warten`
`ready = true;`	`use(value);`

- Annahme dabei: Thread 1 sieht die Ergebnisse der Schreiboperationen in der Reihenfolge, in der Thread 0 sie ausführt
- Diese Annahme wird durch heutige Prozessoren verletzt!
  - z.B. durch *out-of-order execution* und Schreibpuffer
- Problem der **Speicherkonsistenz**

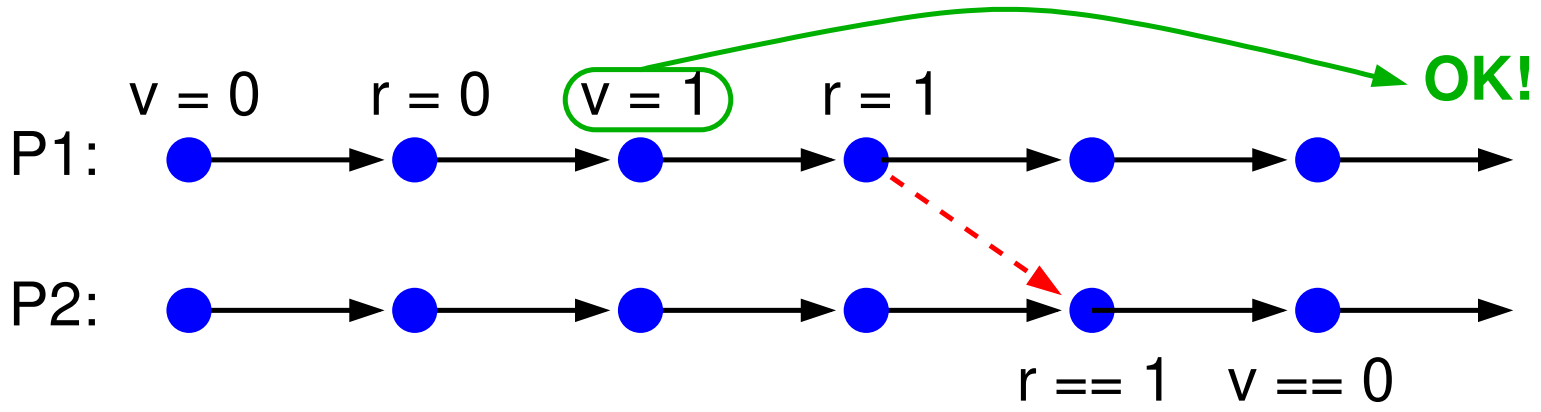
Konsistenzmodelle

- Legen fest, in welcher Reihenfolge Schreibzugriffe auf den Speicher „gesehen“ werden können
  - d.h., welche Werte eine Leseoperation zurückgeben darf
- **Sequentielle Konsistenz**: Ergebnis jeder Programmausführung ist durch folgendes Modell erklärbar:



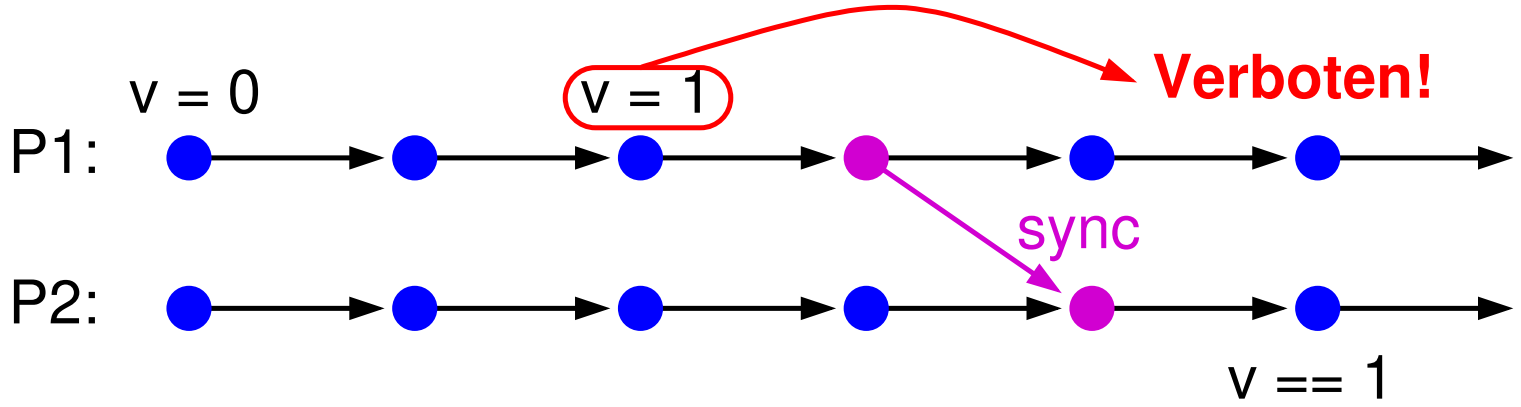
Konsistenzmodelle ...

- Sequentielle Konsistenz bedeutet: alle Prozessoren sehen alle Speicheroperationen in derselben Reihenfolge
  - Operationen eines Prozessors dürfen nicht vertauscht werden
- **Abgeschwächte Konsistenzmodelle** erlauben Vertauschung von Speicheroperationen, außer in speziellen Fällen
  - typisch: Vertauschung über explizite Synchronisations-operationen hinweg ist nicht erlaubt



Konsistenzmodelle ...

- Sequentielle Konsistenz bedeutet: alle Prozessoren sehen alle Speicheroperationen in derselben Reihenfolge
  - Operationen eines Prozessors dürfen nicht vertauscht werden
- **Abgeschwächte Konsistenzmodelle** erlauben Vertauschung von Speicheroperationen, außer in speziellen Fällen
  - typisch: Vertauschung über explizite Synchronisations-operationen hinweg ist nicht erlaubt



Das Java Speichermodell

- Herausforderung: Modell muss prozessorunabhängig sein!
- Bei Betrachtung nur eines Threads: *as-if-serial* Semantik
  - die eigenen Zugriffe erscheinen wie in Programmordnung ausgeführt
  - entspricht dem Verhalten heutiger CPUs
- Bei Betrachtung mehrerer Threads:
  - wenn Operation a aufgrund von **Programmordnung** und/oder expliziter **Synchronisation** vor b aufgeführt werden **muß**, dann sieht b die Effekte von a
  - explizite Synchronisation: `start()` / `join()` sowie von Java bereitgestellte Konstrukte (siehe später)

Das Java Speichermodell: Schlüsselwort `volatile`

- Attribute können durch das Schlüsselwort `volatile` als Synchronisationsvariable deklariert werden
- Schreiben und Lesen einer Synchronisationsvariable wird dann wie explizite Synchronisation behandelt
- Damit Beispiel korrekt implementierbar:

Initialisierung:

```
double value = 0;
volatile boolean ready = false; // ist 'value' gültig?
```

Thread 0	Thread 1
`value = 0.5*2;`	`while (!ready); // aktives Warten`
`ready = true;`	`use(value);`

Das C++ Speichermodell

- Ähnlich zum Java-Speichermodell, aber mit mehr Möglichkeiten
- Basis: Datentyp `std::atomic<`*T*`>`, z.B. `std::atomic<int>`
  - Zugriff nur über explizite `load()` und `store()` Methoden
    - werden garantiert atomar ausgeführt
- Konsistenzanforderungen können über Argument von `load()` bzw. `store()` festgelegt werden
  - Standardeinstellung: sequentielle Konsistenz
- Nachbildung des Java-Speichermodells:
  - Synchronisationsvariablen als `std::atomic<`*T*`>` deklarieren
  - Lesen mit `load(std::memory_order_acquire)`
  - Schreiben mit `store(std::memory_order_release)`

Das C++ Speichermodell ...

- Beispiel als C++ Code:

Initialisierung:

```
double value = 0;
std::atomic<bool> ready = false; // ist 'value' gültig?
```

Thread 0

```
value = 0.5*2;
ready.store(true, std::memory_order_release);
```

Thread 1

```
// aktives Warten
while (!ready.load(std::memory_order_acquire));
use(value);
```

3.7 Semaphore

- Eingeführt durch Edsger Wybe Dijkstra (1965)
- Allgemeines Synchronisationskonstrukt
  - nicht nur wechselseitiger Ausschluß, auch Reihenfolge-synchonisation
- Semaphor ist i.W. eine ganzzahlige Variable
  - Wert kann auf nichtnegative Zahl initialisiert werden
  - zwei **atomare** Operationen:
    - `P()` (auch `wait`, `down` oder `acquire`)
      - verringert Wert um 1
      - falls Wert < 0: Thread blockieren
    - `V()` (auch `signal`, `up` oder `release`)
      - erhöht Wert um 1
      - falls Wert ≤ 0: einen blockierten Thread wecken

Semaphor-Operationen

```
struct Semaphor {
  int count;              // Semaphor–Zähler
  ThreadQueue queue;      // Warteschlange für blockierte Threads
}
```

```
void P(Semaphor &s) {
  s.count--;
  if (s.count < 0) {
    Thread in s.queue
      ablegen;
    Thread blockieren;
  }
}
```

```
void V(Semaphor &s) {
  s.count++;
  if (s.count <= 0) {
    Einen Thread T aus s.queue
      entfernen;
    T auf bereit setzen;
  }
}
```

➥ Hinweis: Tanenbaum definiert Semaphore etwas anders (Zähler zählt höchstens bis 0 herunter)

Interpretation des Semaphor-Zählers

- Zähler ≥ 0: Anzahl freier Ressourcen
- Zähler < 0: Anzahl wartender Threads

Wechselseitiger Ausschluß mit Semaphoren

- Thread 0

```
P(mutex);
// kritischer Abschnitt
V(mutex);
```

Thread 1

```
P(mutex);
// kritischer Abschnitt
V(mutex);
```

- Semaphor `mutex` wird mit 1 vorbelegt
- Semaphor, das an positiven Werten nur 0 oder 1 haben kann, heißt **binäres Semaphor**

Reihenfolgesynchronisation mit Semaphoren

- Beispiel: Thread 1 darf Datei erst öffnen, nachdem Thread 0 sie erzeugt hat
- Thread 0

```
// Datei erzeugen
V(sema);
```

Thread 1

```
P(sema);
// Datei öffnen
```

- Semaphor `sema` wird mit 0 vorbelegt
  - damit: Thread 1 wird blockiert, bis Thread 0 die `V()`-Operation ausgeführt hat
- Merkregel:
  - `P()`-Operation an der Stelle, wo gewartet werden muß
  - `V()`-Operation signalisiert, daß Wartebedingung erfüllt ist
  - vgl. die alternativen Namen `wait()` / `signal()` für `P()` / `V()`

Realisierung von Semaphoren

- Eng verbunden mit Thread-Implementierung
- Bei Kernel-Threads:
  - Implementierung im BS-Kern
  - Operationen sind Systemaufrufe
  - atomare Ausführung durch Interrupt-Sperre und Spinlocks gesichert

Das Erzeuger/Verbraucher-Problem

- Situation:
  - Zwei Thread-Typen: Erzeuger, Verbraucher
  - Kommunikation über einen gemeinsamen, beschränkten Puffer der Länge `N`
    - Operationen `insertItem()`, `removeItem()`
  - Erzeuger legen Elemente in Puffer, Verbraucher entfernen sie
- Synchronisation:
  - Sperrsynchronisation: wechselseitiger Ausschluß
  - Reihenfolgesynchronisation:
    - kein Entfernen aus leerem Puffer: Verbraucher muß warten
    - kein Einfügen in vollen Puffer: Erzeuger muß warten

Lösung des Erzeuger/Verbraucher-Problems

Erzeuger

```
while (true) {
   item = produce();

   insertItem(item);

}
```

Verbraucher

```
while (true) {

   item = removeItem();

   consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
```

für wechselseitigen Ausschluß

Erzeuger

```
while (true) {
   item = produce();

   P(mutex);
   insertItem(item);
   V(mutex);

}
```

Verbraucher

```
while (true) {

   P(mutex);
   item = removeItem();
   V(mutex);

   consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
Semaphor full = 0;
```

für wechselseitigen Ausschluß

verhindert Entfernen aus leerem Puffer

Erzeuger

```
while (true) {
   item = produce();

   P(mutex);
   insertItem(item);
   V(mutex);
   V(full);
}
```

Verbraucher

```
while (true) {
   P(full);
   P(mutex);
   item = removeItem();
   V(mutex);

   consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
Semaphor full = 0;
```

für wechselseitigen Ausschluß

verhindert Entfernen aus leerem Puffer

Erzeuger

```
while (true) {
   item = produce();

   P(mutex);
   insertItem(item);
   V(mutex);
   V(full);
}
```

Verbraucher

```
while (true) {
   P(full);
   P(mutex);
   item = removeItem();
   V(mutex);

   consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
Semaphor full = 0;
Semaphor empty = N;
```

für wechselseitigen Ausschluß
verhindert Entfernen aus leerem Puffer
verhindert Einfügen in vollen Puffer

Erzeuger

```
while (true) {
   item = produce();
   P(empty);
   P(mutex);
   insertItem(item);
   V(mutex);
   V(full);
}
```

Verbraucher

```
while (true) {
   P(full);
   P(mutex);
   item = removeItem();
   V(mutex);
   V(empty);
   consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
Semaphor full = 0;
Semaphor empty = N;
```

für wechselseitigen Ausschluß

verhindert Entfernen aus leerem Puffer

verhindert Einfügen in vollen Puffer

Erzeuger

```
while (true) {
   item = produce();
   P(empty);
   P(mutex);
   insertItem(item);
   V(mutex);
   V(full);
}
```

Verbraucher

```
while (true) {
   P(full);
   P(mutex);
   item = removeItem();
   V(mutex);
   V(empty);
   consume(item);
}
```

Das Leser/Schreiber-Problem

- Gemeinsamer Datenbereich mehrerer Threads
- Zwei Klassen von Threads (bzw. Zugriffen)
  - Leser (*Reader*)
    - dürfen gleichzeitig mit anderen Lesern zugreifen
  - Schreiber (*Writer*)
    - stehen unter wechselseitigem Ausschluß, auch mit Lesern
      - verhindert Lesen von inkonsistenten Daten
- Typisches Problem in Datenbank-Systemen

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   readDataBase();

   useData();
}
```

Semaphore und gemeinsame Variable

Schreiber

```
while(true) {
   createData();

   writeDataBase();

}
```

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   readDataBase();

   useData();
}
```

Semaphore und gemeinsame Variable

```
Semaphor db=1; // Schützt Datenbank
```

Schreiber

```
while(true) {
   createData();
   P(db);
   writeDataBase();
   V(db);
}
```

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   P(db);

   readDataBase();

   V(db);

   useData();
}
```

Semaphore und gemeinsame Variable

```
Semaphor db=1; // Schützt Datenbank
```

Schreiber

```
while(true) {
   createData();
   P(db);
   writeDataBase();
   V(db);
}
```

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   rc++;
   if (rc == 1)
     P(db);

   readDataBase();

   rc--;
   if (rc == 0)
     V(db);

   useData();
}
```

Semaphore und gemeinsame Variable

```
int rc=0;       // Anzahl Leser
Semaphor db=1;  // Schützt Datenbank
```

Schreiber

```
while(true) {
   createData();
   P(db);
   writeDataBase();
   V(db);
}
```

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {
   P(mutex);
   rc++;
   if (rc == 1)
     P(db);
   V(mutex);
   readDataBase();
   P(mutex);
   rc--;
   if (rc == 0)
     V(db);
   V(mutex);
   useData();
}
```

Semaphore und gemeinsame Variable

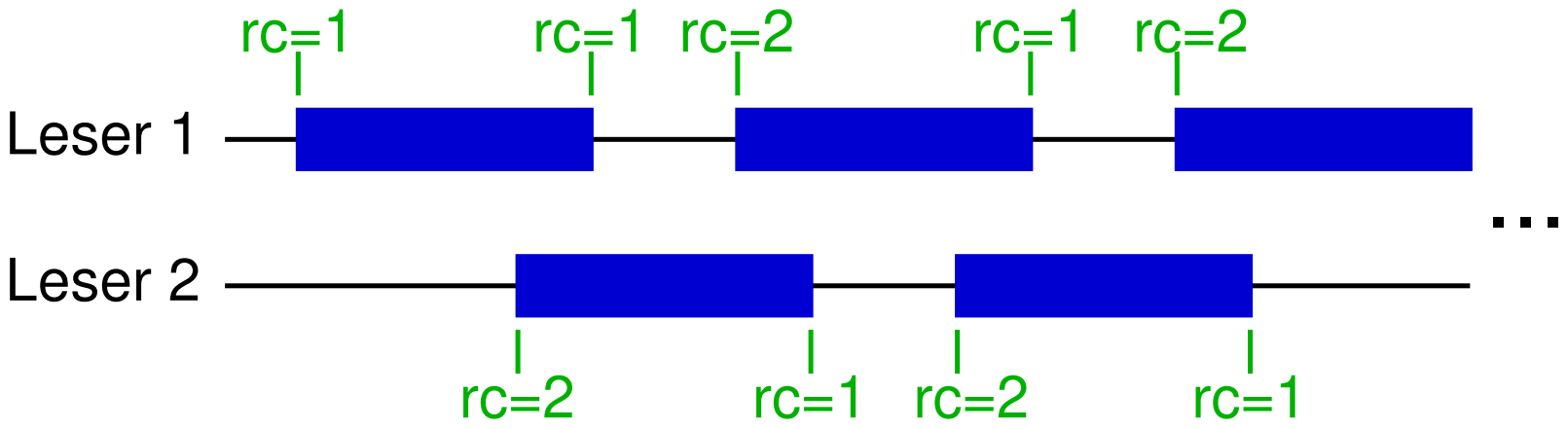
```
int rc=0;        // Anzahl Leser
Semaphor db=1;   // Schützt Datenbank
Semaphor mutex=1;
```

Schreiber

```
while(true) {
   createData();
   P(db);
   writeDataBase();
   V(db);
}
```

Eigenschaft der skizzierten Lösung

- Die Synchronisation ist unfair: Leser haben Priorität vor Schreibern
  - Schreiber kann verhungern



- Mögliche Lösung:
  - neue Leser blockieren, wenn ein Schreiber wartet

Motivation

- Programmierung mit Semaphoren ist schwierig
  - Reihenfolge der P/V-Operationen: Verklemmungsgefahr
  - Synchronisation über gesamtes Programm verteilt

Monitor (Hoare, 1974; Brinch Hansen 1975)

- Modul mit Daten, Prozeduren und Initialisierungscode
  - Zugriff auf die Daten nur über Monitor-Prozeduren
  - (entspricht in etwa einer Klasse)
- Alle Prozeduren stehen unter wechselseitigem Ausschluß
  - nur jeweils ein Thread kann Monitor benutzen
- Programmiersprachkonstrukt: Realisierung durch Übersetzer

Bedingungsvariable (Zustandsvariable, *condition variables*)

- Zur Reihenfolgesynchronisation zwischen Monitor-Prozeduren
- Darstellung anwendungsspezifischer Bedingungen
  - z.B. voller Puffer im Erzeuger/Verbraucher-Problem
- Zwei Operationen:
  - `wait()`: Blockieren des aufrufenden Threads
  - `signal()`: Aufwecken blockierter Threads
- Bedingungsvariable verwaltet Warteschlange blockierter Threads
- Bedingungsvariable hat kein „Gedächtnis“:
  - `signal()` weckt nur einen Thread, der `wait()` bereits aufgerufen hat

Funktion von `wait()`:

- Aufrufender Thread wird blockiert
  - nach Ende der Blockierung kehrt `wait()` zurück
- Aufrufender Thread wird in die Warteschlange der Bedingungsvariable eingetragen
- Monitor steht bis zum Ende der Blockierung anderen Threads zur Verfügung

Funktion von `signal()`:

- Falls Warteschlange der Bedingungsvariable nicht leer:
  - mindestens einen Thread wecken:
    aus Warteschlange entfernen und Blockierung aufheben

Varianten für `signal()`:

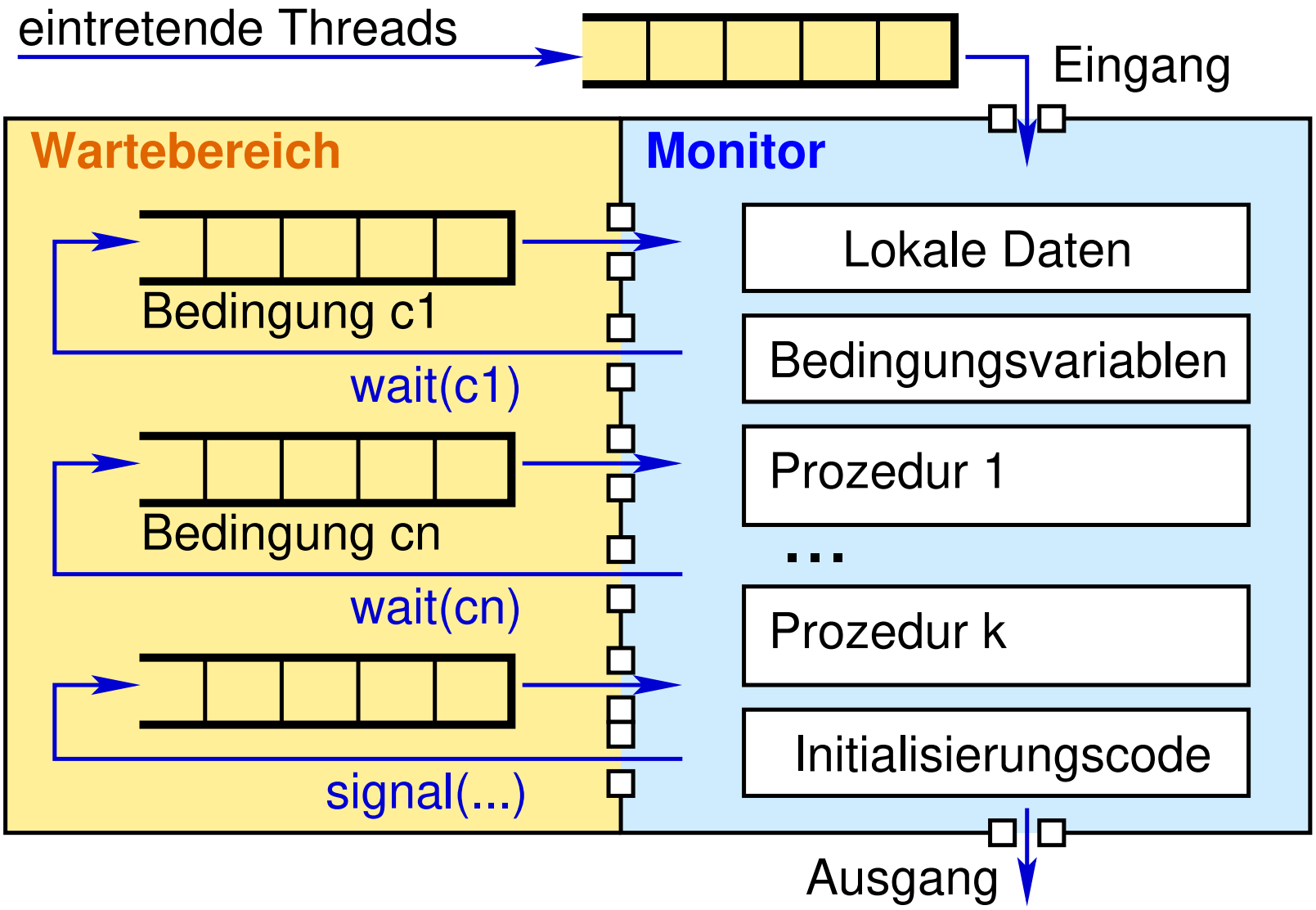
1. Ein Thread wird geweckt (meist der am längsten wartende)
   - **a)** signalisierender Thread bleibt im Besitz des Monitors
   - **b)** geweckter Thread erhält den Monitor sofort
     - **i.** signalisierender Thread muß sich erneut bewerben (Hoare)
     - **ii.** `signal()` muß letzte Anweisung in Monitorprozedur sein (Brinch Hansen)
2. Alle Threads werden geweckt
   - ➥ signalisierender Thread bleibt im Besitz des Monitors

➥ Bei 1a) und 2) ist nach Rückkehr aus `wait()` **nicht** sicher, daß die Bedingung (noch) erfüllt ist!

Typische Verwendung von `wait()` und `signal()`

- Testen einer Bedingung
  - bei Variante 1b):
    - `if (!Bedingung) wait(condVar);`
  - bei Varianten 1a) und 2):
    - `while (!Bedingung) wait(condVar);`
- Signalisieren der Bedingung
  - [`if (Bedingung)`] `signal(condVar);`

Aufbau eines Monitors nach Hoare



Semaphor-Realisierung m. Monitor (Pascal-artig, Brinch Hansen)

```
monitor Semaphor
  condition nonbusy;
  integer count;

  procedure P
  begin
    count := count - 1;
    if count < 0 then
      wait(nonbusy);
  end;

  procedure V
  begin
    count := count + 1;
    if count <= 0 then
      signal(nonbusy);
  end;

  count = 1;
end monitor;
```

➥ Umgekehrt können auch Monitore (insbes. Bedingungsvariable) mit Semaphoren nachgebildet werden

Erzeuger/Verbraucher m. Monitor (Pascal-artig, Brinch Hansen)

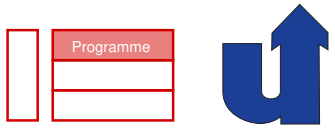
```
monitor ErzeugerVerbraucher
  condition nonfull, nonempty;
  integer count;

  procedure insert(item: integer)
  begin
    if count = N then
      wait(nonfull);
    insertItem(item);
    count := count + 1;
    signal(nonempty);
  end;

  function remove: integer
  begin
    if count = 0 then
      wait(nonempty);
    remove := removeItem();
    count := count - 1;
    signal(nonfull);
  end;

  count = 0;
end monitor;
```

Motivation für Broadcast-Signalisierung (Variante 2)

- Aufwecken aller Threads sinnvoll, wenn unterschiedliche Wartebedingungen vorliegen
- Beispiel: Erzeuger/Verbraucher-Problem mit variablem Bedarf an Puffereinträgen

```
procedure insert(item: ..., size: integer)
begin
  while count + size > N do
    wait(nonfull);
  ...
```

- Nachteil: viele Threads konkurrieren um Wiedereintritt in den Monitor



3.10.1 Standard-Synchronisations-Mechanismen in Java

- Sind bereits in der Programmiersprache definiert
- Monitor-ähnlich, aber Klassen statt Module
- Synchronisierte Methoden
  - müssen explizit als `synchronized` deklariert werden
  - stehen (pro Objekt!) unter wechselseitigem Ausschluß
- Keine expliziten Bedingungsvariablen, stattdessen genau eine implizite Bedingungsvariable pro Objekt
  - Basisklasse `Object` definiert Methoden `wait()`, `notify()` und `notifyAll()`
    - diese Methoden werden von allen Klassen geerbt
    - `notify()`: Signalisierungsvariante 1a)
    - `notifyAll()`: Signalisierungsvariante 2)

Beispiel: Erzeuger/Verbraucher-Problem

```
public class ErzVerb {
  ...
  public synchronized void insert(int item) {
    while (count == buffer.getSize()) { // Puffer voll?
      try {
        wait();                         //   ja: warten ...
      }
      catch (InterruptedException e) {}
    }
    buffer.insertItem(item);            // Item eintragen
    count++;
    notifyAll();                        // alle wecken
  }
```

Beispiel: Erzeuger/Verbraucher-Problem ...

```
  public synchronized int remove() {
    int result;
    while (count == 0) {                    // Puffer leer?
      try {
        wait();                             //   ja: warten ...
      }
      catch (InterruptedException e) {}
    }
    result = buffer.removeItem();           // Item entfernen
    count--;
    notifyAll();                            // alle wecken
    return result;
  }
}
```

Anmerkungen zum Beispiel

- Vollständiger Code ist im WWW (Vorlesungsseite) verfügbar
- Bedingungen müssen immer in `while`-Schleife geprüft werden
- `notify()` statt `notifyAll()` ist **nicht** korrekt!
  - funktioniert nur bei genau einem Erzeuger und genau einem Verbraucher
    - da nur eine Bedingungsvariable für zwei verschiedene Bedingungen benutzt wird, kann es sein, daß der falsche Thread aufgeweckt wird
  - Übungsaufgabe:
    - mit Programm aus WWW ausprobieren!
    - mit Simulator (siehe Webseite) nachvollziehen!

`synchronized` Blöcke

- Java unterstützt auch wechselseitigen Ausschluß beliebiger Code-Blöcke über das eingebaute Mutex eines Objekts
  - Syntax: `synchronized( Objekt ) { Block }`
- Beispiel: gegeben ist die Klasse

```
public class Queue {
  public synchronized void enqueue(Object data) { ... }
  ...
```

  - atomares Einfügen von zwei Elementen:

```
synchronized(queue) {     // Sperrt das Objekt 'queue'
  queue.enqueue(elem1);
  queue.enqueue(elem2);
}
```

3.10.2 Java Klassenbibliothek: Synchronisationsklassen

- Java-Paket `java.util.concurrent.lock`
- Klasse `Semaphore`
- Schnittstellen zur Implementierung des Monitor-Konzepts:
  - *Mutual Exclusion Locks* (Mutex): Schnittstelle `Lock`
    - Verhalten wie binäres Semaphor
    - Zustände: gesperrt, frei
  - Bedingungsvariable: Schnittstelle `Condition`
    - fest an ein `Lock` gebunden
    - `Lock` wird für wechselseitigen Ausschluß der Monitor-Prozeduren genutzt
- Schnittstelle `ReadWriteLock` (Leser-Schreiber-Sperren)

Klasse `Semaphore`

- Konstruktor: `Semaphore(int wert)`
  - erzeugt Semaphor mit angegebenem Initialwert
- Wichtigste Methoden:
  - `void acquire()`
    - entspricht P-Operation
  - `void release()`
    - entspricht V-Operation

Schnittstelle `Lock`

- Wichtigste Methoden:
  - `void lock()`
    - sperren (entspricht P bei binärem Semaphor)
  - `void unlock()`
    - freigeben (entspricht V bei binärem Semaphor)
  - `Condition newCondition()`
    - erzeugt neue Bedingungsvariable, die an dieses `Lock`-Objekt gebunden ist
      - beim Warten an der Bedingungsvariable wird dieses `Lock` freigegeben
- Implementierungsklasse: `ReentrantLock`
  - neu erzeugte Sperre ist zunächst frei

Schnittstelle `Condition`

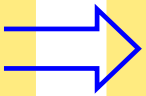
- Wichtigste Methoden:
  - `void await()`
    - *wait*-Operation: warten auf Signalisierung
    - Thread wird blockiert, zur `Condition` gehöriges `Lock` wird freigegeben
    - nach Signalisierung: `await` kehrt erst zurück, wenn `Lock` wieder erfolgreich gesperrt ist
  - `void signal()`
    - Signalisierung eines wartenden Threads (Variante 1a)
  - `void signalAll()`
    - Signalisierung aller wartenden Threads (Variante 2)

Anmerkungen

- Mit diesen Objekten lassen sich Monitore nachbilden:

Monitor

```
monitor Bsp

   condition cond;
   procedure foo
   begin
      if ... then
         wait (cond);
      ...
      signal (cond);

   end;
end monitor;
```



Java–Code

```
public class Bsp {
   Lock mutex;      // = new ReentrantLock();
   Condition cond; // = mutex.newCondition();
   public void foo() {
      mutex.lock();
      while(...)
         cond.await();
      ...
      cond.signal();
      mutex.unlock();
   }
}
```

Im Konstruktor

- Ähnliche Konzepte wie `Lock` und `Condition` auch in anderen Thread-Schnittstellen

Anmerkungen ...

- Herstellen der signalisierten Bedingung und Signalisierung muß bei gesperrtem Mutex erfolgen!
- Sonst: Gefahr des *lost wakeup*

Thread 1

```
condition = true;
cond.signal();
```

Thread 2

```
mutex.lock();
...
while (!condition)
  cond.await();
...
mutex.unlock();
```

Anmerkungen ...

- Herstellen der signalisierten Bedingung und Signalisierung muß bei gesperrtem Mutex erfolgen!
- Sonst: Gefahr des *lost wakeup*

Thread 1

```
condition = true;
cond.signal();
```

Thread 2

```
mutex.lock();
...
while (!condition)
  cond.await();
...
mutex.unlock();
```

Anmerkungen ...

- Herstellen der signalisierten Bedingung und Signalisierung muß bei gesperrtem Mutex erfolgen!
- Sonst: Gefahr des *lost wakeup*

Thread 1

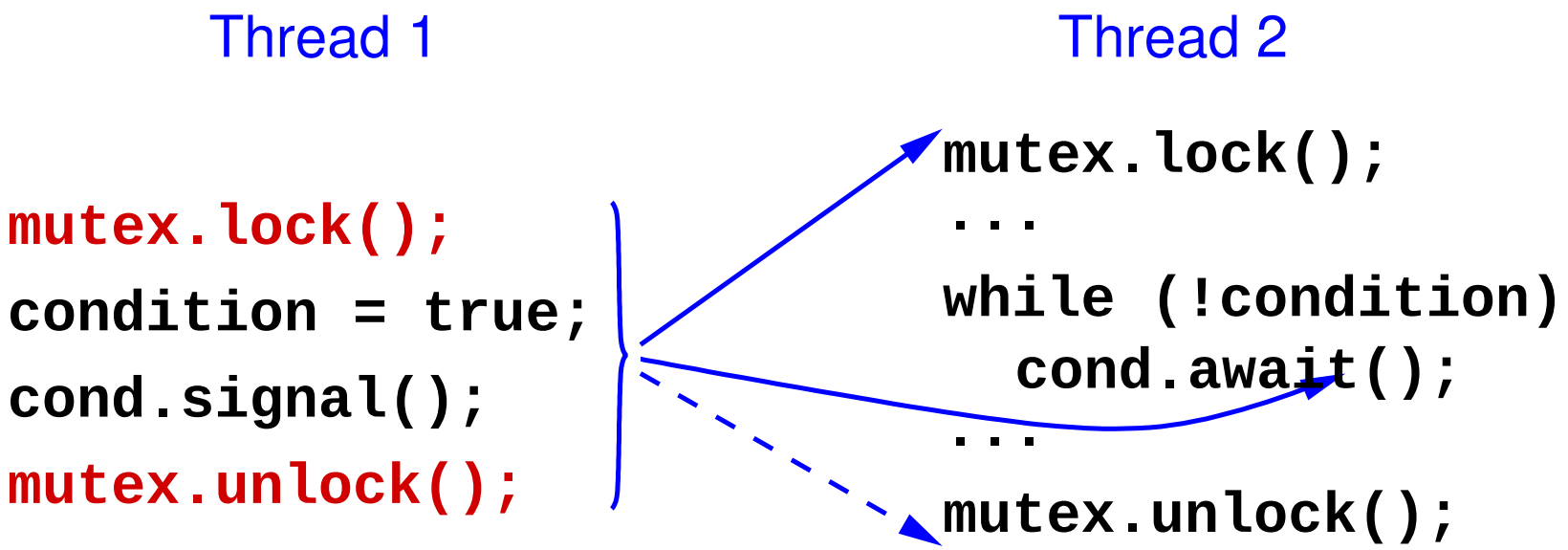
```
mutex.lock();
condition = true;
cond.signal();
mutex.unlock();
```

Thread 2

```
mutex.lock();
...
while (!condition)
  cond.await();
...
mutex.unlock();
```

Anmerkungen ...

- Herstellen der signalisierten Bedingung und Signalisierung muß bei gesperrtem Mutex erfolgen!
- Sonst: Gefahr des *lost wakeup*



Anmerkungen ...

- Freigabe der Sperre sollte über `try` - `finally` erfolgen
  - sonst bleibt `mutex` ggf. gesperrt, wenn die Methode `return` ausführt oder eine Exception wirft
- Beispiel:

```
public void foo() {
  mutex.lock();
  try {
    ... // Code der Methode
  }
  finally { mutex.unlock(); }
}
```

  - `finally`-Block wird **immer** ausgeführt, nachdem `try`-Block verlassen wird
- Alternative ab Java 7: *try-with-resources*

Synchronisationspaket `BSsync` für die Übungen

- Für die Übungen verwenden wir eine vereinfachte Version der `java.util.concurrent.lock`-Klassen
  - weniger Methoden (nur die wichtigsten, siehe vorherige Folien)
  - keine `InterruptedException` bei `await()`
  - Optionen zur besseren Fehlersuche
  - `Lock` direkt als Klasse implementiert
    - d.h. `new Lock()` statt `new ReentrantLock()`
- JAR-Archiv `BSsync.jar` und API-Dokumentation im WWW verfügbar
  - über die Vorlesungsseite

Unterstützung der Fehlersuche in `BSsync`

- Konstruktor `Semaphore(int wert, String name)`
  Konstruktor `Lock(String name)`
  Methode `newCondition(String name)` von `Lock`
  - Erzeugung eines Objekts mit gegebenem Namen
- Attribut `public static boolean verbose`
  in den Klassen `Semaphore` und `Lock`
  - schaltet Protokoll aller Operationen auf Semaphoren bzw. Locks und Bedingungsvariablen ein
  - Protokoll benutzt obige Namen
- Erlaubt Verfolgung des Programmablaufs
  - z.B. bei Verklemmungen

Beispiel: Erzeuger/Verbraucher-Problem

```
public class ErzVerb {
  private Lock mutex;              // Wechsels. Ausschluß
  private Condition nonfull;       // Warten bei vollem Puffer
  private Condition nonempty;      // Warten bei leerem Puffer
  private int count;               // Zählt belegte Pufferplätze

  Buffer buffer;

  public ErzVerb(int size) {
    buffer = new Buffer(size);
    mutex = new Lock("mutex");                          // Lock erzeugen
    nonfull = mutex.newCondition("nonfull");    // Bedingungsvar.
    nonempty = mutex.newCondition("nonempty");// erzeugen
    count = 0;
  }
```

Beispiel: Erzeuger/Verbraucher-Problem ...

```
public void insert(int item) {
  mutex.lock();                          // Mutex sperren
  while (count == buffer.getSize())      // Puffer voll?
    nonfull.await();                     //   ja: warten...
  buffer.insertItem(item);               // Item eintragen
  count++;
  nonempty.signal();                     // ggf. Thread wecken
  mutex.unlock();                        // Mutex freigeben
}
```

Beispiel: Erzeuger/Verbraucher-Problem ...

```
public int remove() {
  int result;
  mutex.lock();                          // Mutex sperren
  while (count == 0)                     // Puffer leer?
    nonempty.await();                    //   ja: warten...
  result = buffer.removeItem();          // Item entfernen
  count--;
  nonfull.signal();                      // ggf. Thread wecken
  mutex.unlock();                        // Mutex freigeben
  return result;
}
```

Beispiel: Erzeuger/Verbraucher-Problem ...

```
  public static void main(String argv[]) {
    ErzVerb ev = new ErzVerb(5);
    Lock.verbose = true;                    // Protokoll anschalten
    Producer prod = new Producer(ev);
    Consumer cons = new Consumer(ev);
    prod.start();
    cons.start();
  }
}
```

➥ Vollständiger Code im WWW (Vorlesungsseite)!

Reader/Writer-Locks

- Schnittstelle `ReadWriteLock`
  - zwei Methoden `readLock()` und `writeLock()`
    - geben ein Objekt der Schnittstelle `Lock` zurück
- Implementierungsklasse `ReentrantReadWriteLock`
  - Lese- und Schreibsperren können rekursiv belegt werden
  - Schreiber kann auch noch Lesesperre belegen, aber nicht umgekehrt
  - für Schreibsperre kann auch `Condition` erzeugt werden
  - im Konstruktor kann Fairness eingeschaltet werden

Mutex Variablen

- Verhälten ähnlich zu Lava `Lock` (binäres Semaphor)
  - Zustände: frei, gesperrt; bei Erzeugung: frei
- Deklaration (und Initialisierung):
  `std::mutex mutex;`
- Zum Sperren des Mutex: Erzeugung eines Objeks der Klasse `std::unique_lock`:
  - `std::unique_lock<std::mutex> lock(mutex);`
  - Mutex wird automatisch freigegeben, wenn `lock` deallokiert wird, d.h., wenn aktueller Code-Block verlassen wird
- Klasse `mutex` erlaubt kein rekursives Sperren
  - d.h. selber Thread kann Mutex nicht zweimal sperren
  - Alternative: Klasse `recursive_mutex`

Bedingungsvariablen

- Deklaration (und Initialisierung):
  `std::condition_variable cond;`
- Wichtige Methoden:
  - wait: `void wait(unique_lock<mutex>& lock)`
    - Thread wird blockiert, das von `lock` verwaltete Mutex wird temporär freigegeben
    - signalisierender Thread behält das Mutex (Signalisierungsvariante 1a)
    - daher typisch:
      ```
      while (!condition_met)
          cond.wait(lock);
      ```
  - Signalisierung eines Threads: `void notify_one()`
  - Signalisierung aller Threads: `void notify_all()`

Beispiel: Nachbildung eines Monitors

```
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mutex;
std::condition_variable cond;
volatile bool ready = false;
volatile int result;

void StoreResult(int arg) {
    std::unique_lock<std::mutex> lock(mutex);
    result = arg;   /* Ergebnis speichern */
    ready = true;
    cond.notify_all();
    // 'lock' wird bei Verlassen der Methode deallokiert, damit wird 'mutex' freigegeben!
 }
```

Beispiel: Nachbildung eines Monitors ...

```
int ReadResult()
{
    std::unique_lock<std::mutex> lock(mutex);
    while (!ready)
        cond.wait(lock);
    return result;
  // mutex wird automatisch freigegeben
}
```

Mutex Variablen

- Analog zu Java `Lock` und C++ `std::mutex`
  - Zustände: gesperrt, frei; Initialzustand: frei
- Deklaration und Initialisierung (globale/statische Variable):
  `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Operationen:
  - Sperren: `pthread_mutex_lock(&mutex)`
    - Verhalten bei rekursiver Belegung nicht festgelegt
  - Freigeben: `pthread_mutex_unlock(&mutex)`
    - bei Terminierung eines Threads werden Sperren *nicht* automatisch freigegeben!
  - Sperrversuch: `pthread_mutex_trylock(&mutex)`
    - blockiert nicht, liefert ggf. Fehlercode

Bedingungsvariablen

- Deklaration und Initialisierung (globale/statische Variable):
  `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- Operationen:
  - Warten: `pthread_cond_wait(&cond, &mutex)`
    - Thread wird blockiert, `mutex` wird temporär freigegeben
    - signalisierender Thread behält `mutex`
    - typische Verwendung:

```
while (!condition_met)
  pthread_cond_wait(&cond, &mutex);
```

  - Signalisieren:
    - eines Threads: `pthread_cond_signal(&cond)`
    - aller Threads: `pthread_cond_broadcast(&cond)`

Beispiel: Nachbildung eines Monitors

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;

volatile bool ready = false;
volatile int result;


void StoreResult(int arg)
{
    pthread_mutex_lock(&mutex);
    ergebnis = arg;   /* speichere Ergebnis */
    ready = true;
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Beispiel: Nachbildung eines Monitors ...

```
int ReadResult()
{
    int tmp;
    pthread_mutex_lock(&mutex);
    while (!ready)
        pthread_cond_wait(&cond, &mutex);
    tmp = ergebnis;   /* lies Ergebnis */
    pthread_mutex_unlock(&mutex);
    return tmp;
}
```

3.10.5 Synchronisationsunterstützung in Linux

- Ziel: BS-Aufrufe vermeiden, wann immer das möglich ist
- Basis: Systemaufruf `futex` (*Fast User space muTEX*)
- Argumente u.a.:
  - `addr1`: Adresse einer `int`-Variable im Benutzeradreßraum
  - `op`: auszuführende Futex-Operation
  - `val1`: abhängig von Operation
- Operationen u.a.:
  - `FUTEX_WAIT`: falls `*addr1 == val1` blockiere den Thread
    - Abfrage und Blockierung erfolgen atomar, da im BS-Kern
  - `FUTEX_WAKE`: wecke `val1` Threads auf, die wegen eines `FUTEX_WAIT` mit Adresse `addr1` blockiert sind

Realisierung eines Mutex

```
class Mutex
{
  int nThreads = 0;          // Zahl der Threads vor/im kritischen Abschnitt
  bool signaled = false;     // Zur Signalisierung

  void lock()
  {
    if (fetchAndAdd(nThreads, 1) == 0) // Erster Thread
      return;                          // darf in kritischen Abschnitt

    while (!fetchAndSet(signaled, false))   // Warte auf
      futex(&signaled, FUTEX_WAIT, false); // Signalisierung
  }
```

Realisierung eines Mutex ...

```
  void unlock()
  {
    if (fetchAndAdd(nThreads, -1) == 1) // Einziger Thread
      return;                           // muss niemanden wecken

    signaled = true;
    futex(&signaled, FUTEX_WAKE, 1); // Einen Thread wecken
  }
};
```

➥ Systemaufruf nur dann, wenn ein Thread blockiert bzw. geweckt werden muß

Realisierung von `signalAll()` für Bedingungsvariable

- Problem: alle Threads werden geweckt und versuchen gleichzeitig, das Mutex zu sperren
  - alle bis auf einen werden wieder blockiert
- Lösung: weitere Operation `FUTEX_CMP_REQUEUE`
  - wie bei `FUTEX_WAIT` wird eine Anzahl Threads aufgeweckt
  - alle anderen Threads, die am *Futex* (der Bedingungsvariable) warten, werden direkt in die Warteschlange eines zweiten *Futex* (des Mutex) verschoben

3.11 *Lock-free* Datenstrukturen

- Ziel: Datenstrukturen (typ. *Collections*) ohne wechselseitigem Ausschluss
  - performanter, keine Gefahr von Verklemmungen
- Typische Vorgehensweise:
  - Verwendung atomarer *Read-Modify-Write*-Befehle anstelle von Sperren
  - im Konfliktfall, d.h. bei gleichzeitiger Änderung durch einen anderen Thread, wird die betroffene Operation wiederholt
- ***Lock-free***: unter allen Umständen macht **mindestens ein** Thread nach endlich vielen Schritten Fortschritt
- ***Wait-free***: unter allen Umständen macht **jeder** Thread nach endlich vielen Schritten Fortschritt

Beispiel: Anfügen am Ende eines Arrays

```
Data buffer[N];     // Puffer–Array
int wrPos = 0;      // Position für nächstes einzutragendes Element

void append(Data data) {
   int wrPosOld = fetchAndAdd(wrPos, 1);
   buffer[wrPosOld] = data;
}
```

➥ Nutzt atomare *fetch-and-add* Operation:

```
int fetchAndAdd(int &addr, int val) {
   int tmp = addr;
   addr += val;
   return tmp;
}
```

Atomar!

➥ erhöht Wert bei `addr` um `val`, gibt alten Wert zurück

Beispiel: Treiber-Stack

- *Lock-free* Implementierung eines Stacks auf Basis einer verketteten Liste (von R. Kent Treiber)
- Basis: atomare *compare-and-set* Operation:

```
bool compareAndSet(T &addr, T expect, T newval) {
  if (addr == expect) {
    addr = newval;
    return true;
  }
  return false;
}
```

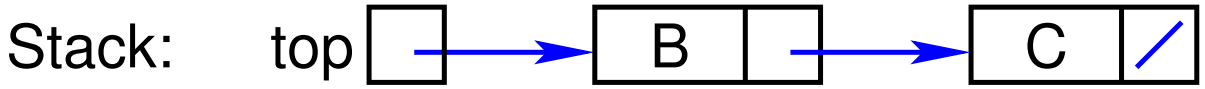
Atomar!

  - falls der Wert bei `addr` noch dem erwarteten Wert `expect` übereinstimmt, überschreibe ihn mit `newval`

3.11 *Lock-free* Datenstrukturen ...

```
Node top = null; // zeigt auf oberstes Kellerelement

void push(Data data) { // Neues Element auf den Keller schreiben
  Node tmp;
  Node node = new Node(data); // Neues Listenelement erzeugen
  do {
    tmp = top;          // top merken
    node.next = tmp; // Verkettung
  } while (!compareAndSet(top, tmp, node)); // top aktualisieren
}

Data pop() {        // Oberstes Element vom Keller entfernen
  Node tmp, next; //  Hier ohne Fehlerbehandlung (leerer Keller) gezeigt!
  do {
    tmp = top;          // top merken
    next = tmp.next; // next = zweitoberstes Element
  } while (!compareAndSet(top, tmp, next)); // top aktualisieren
  return tmp.data;
}
```

Beispiel



Thread 1: tmp

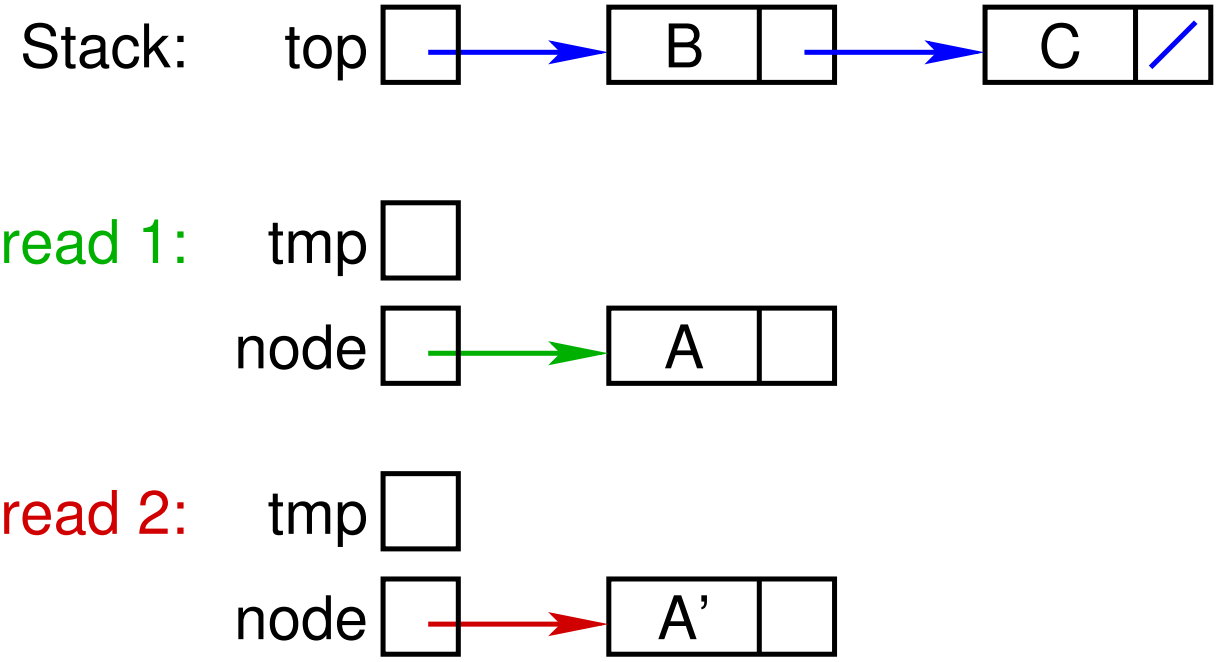
node

Thread 2: tmp

node

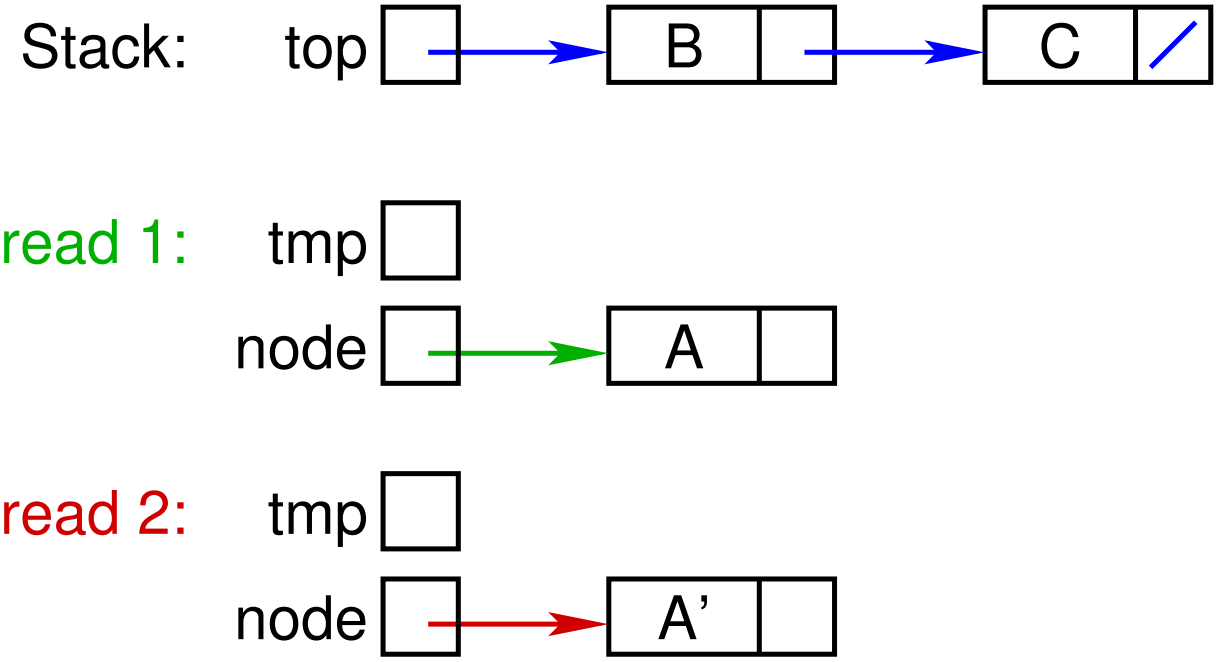
```
Node node = new Node(data);
do{
  tmp = top;
  node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



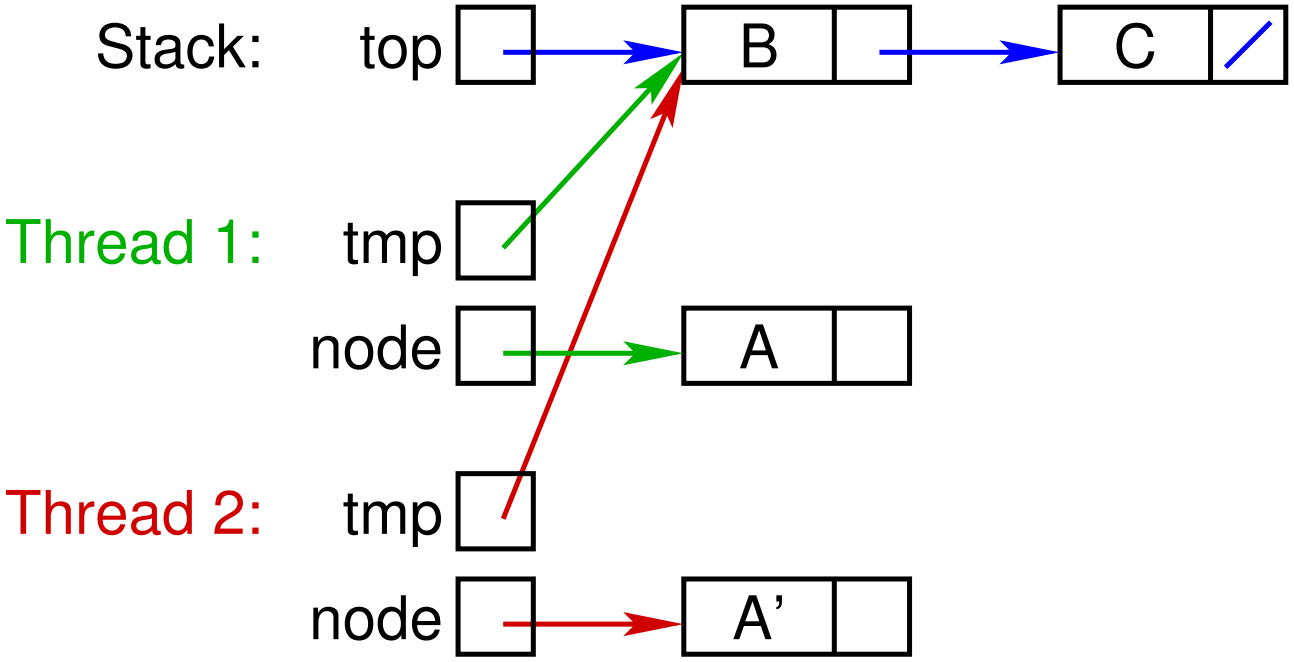

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



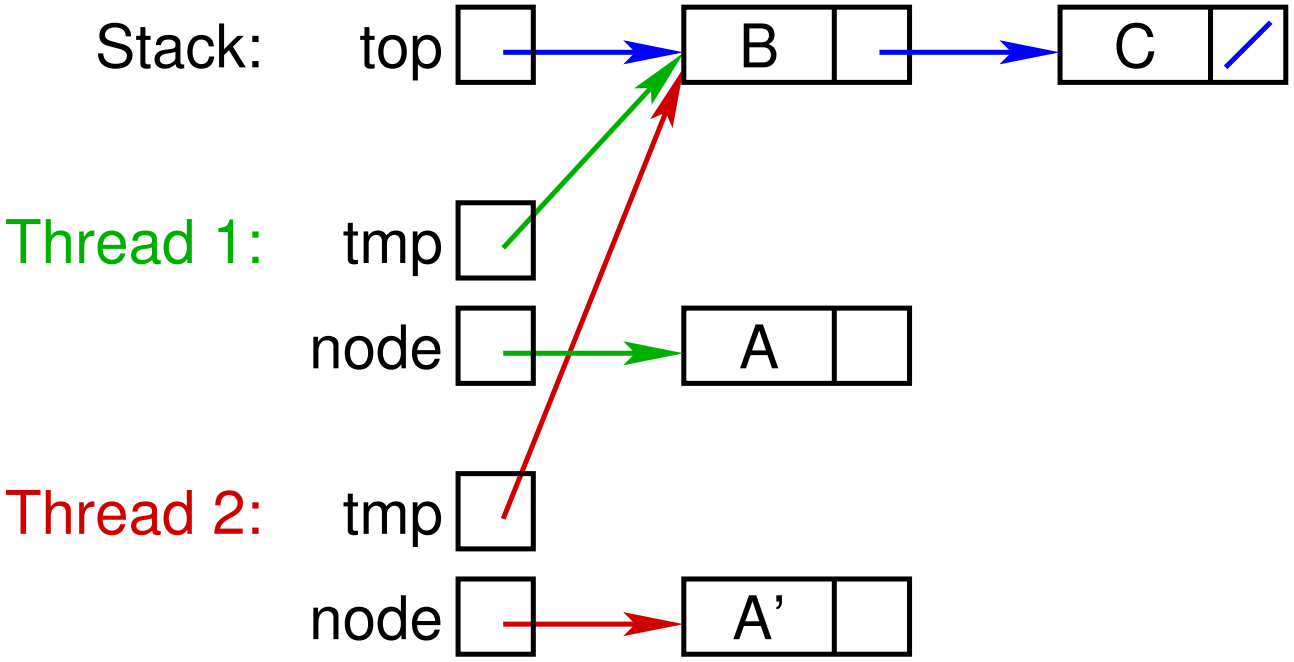

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



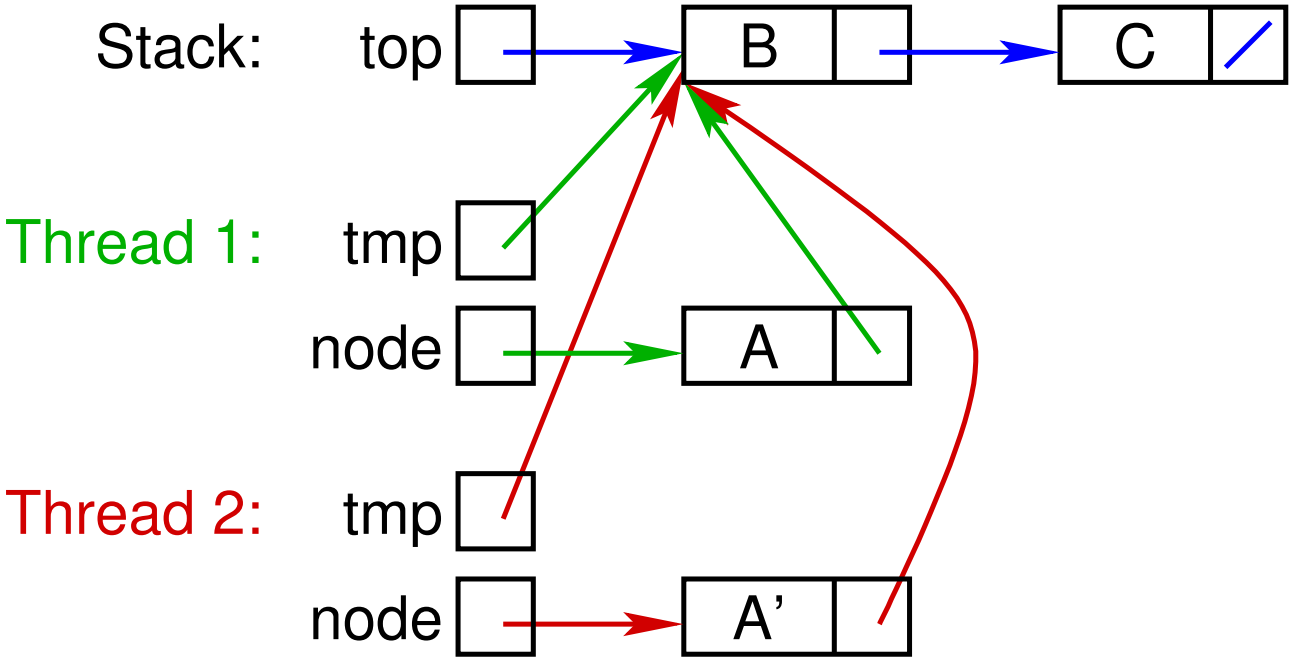

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



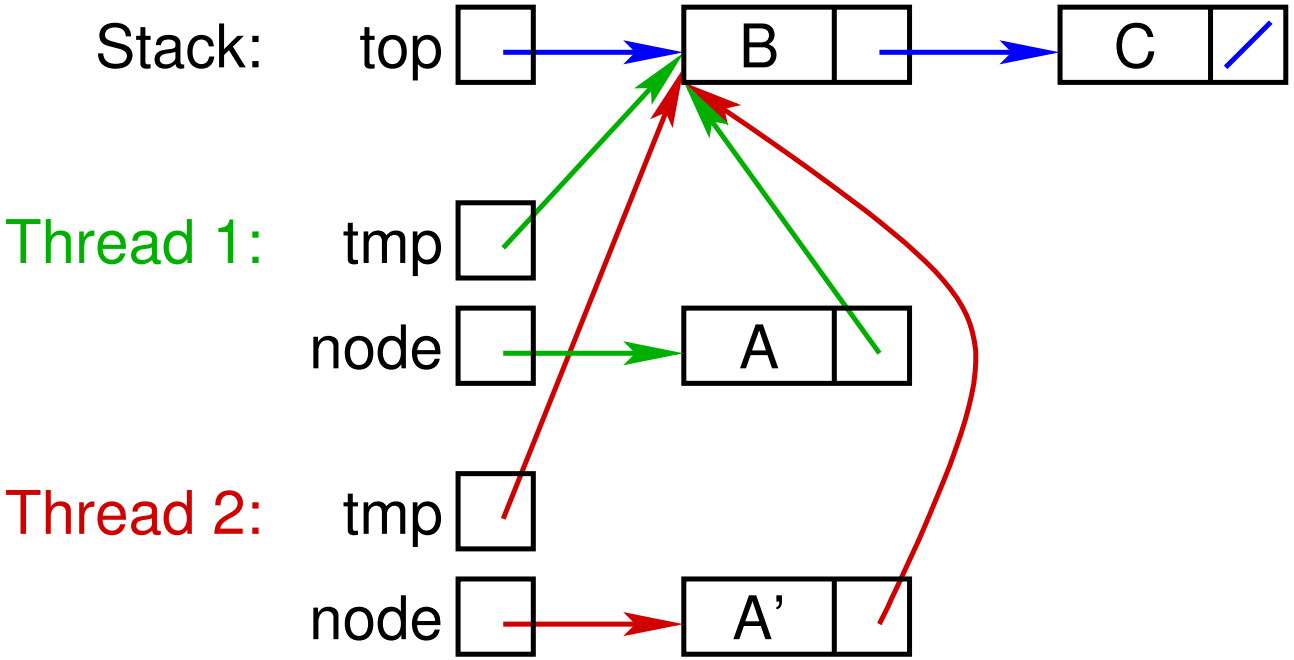

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



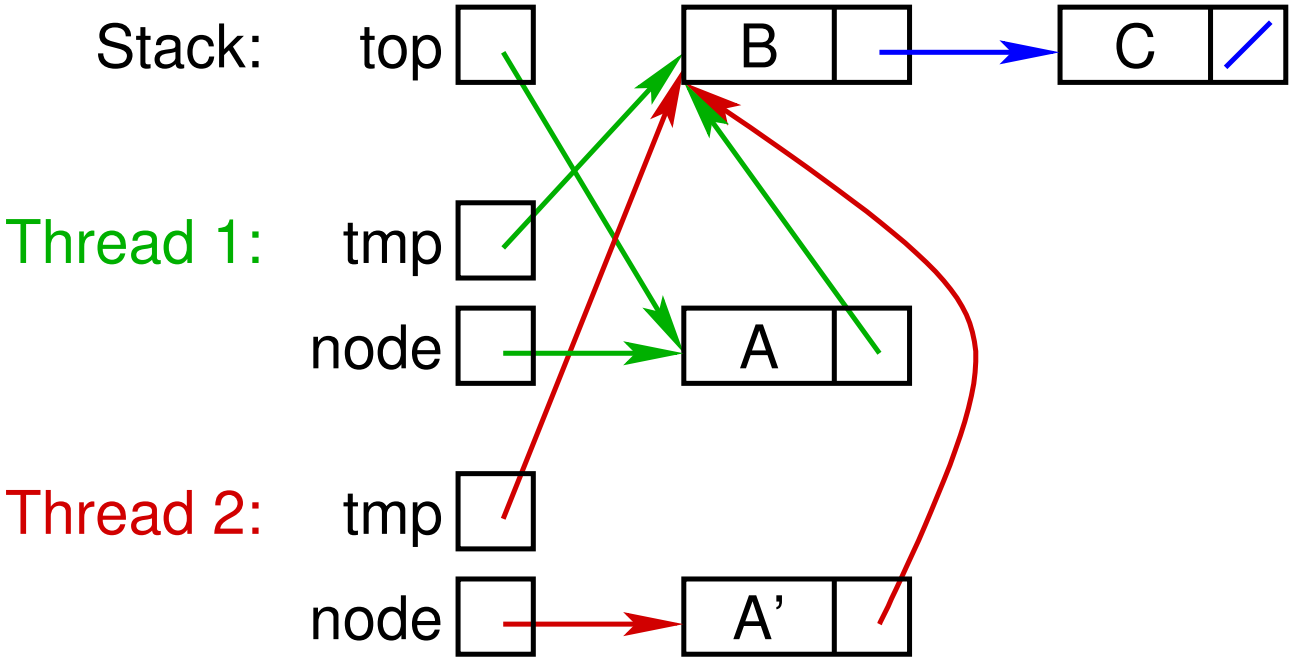

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



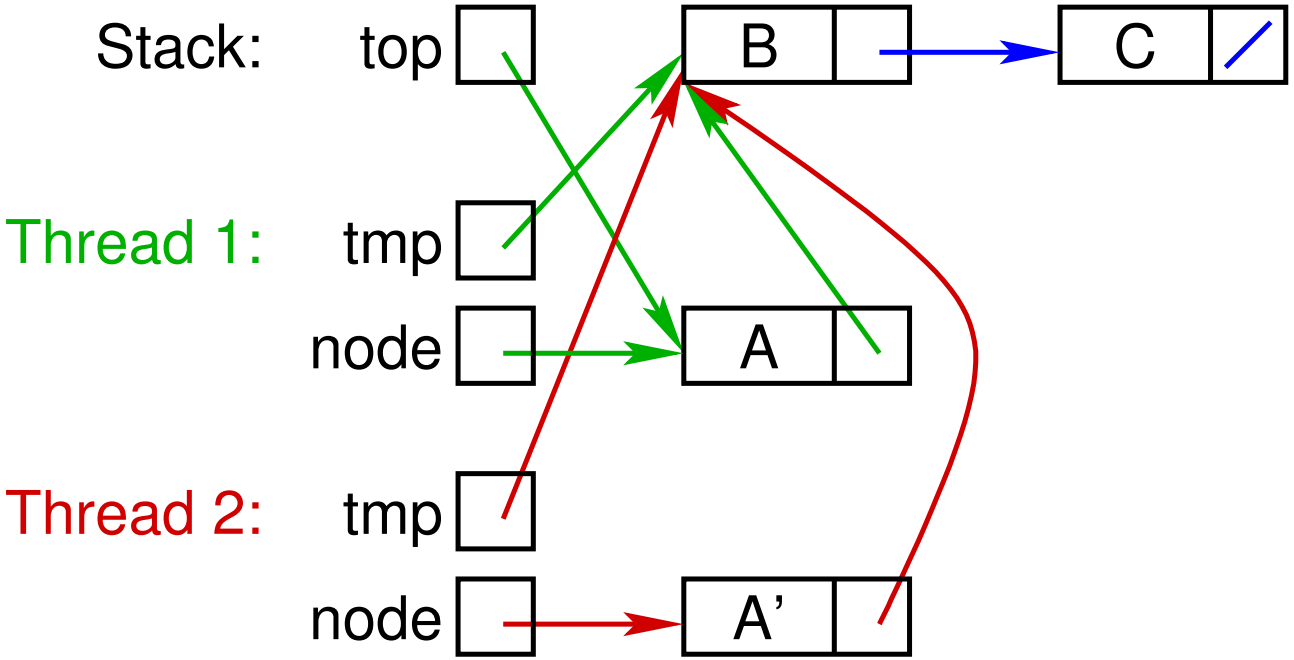

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



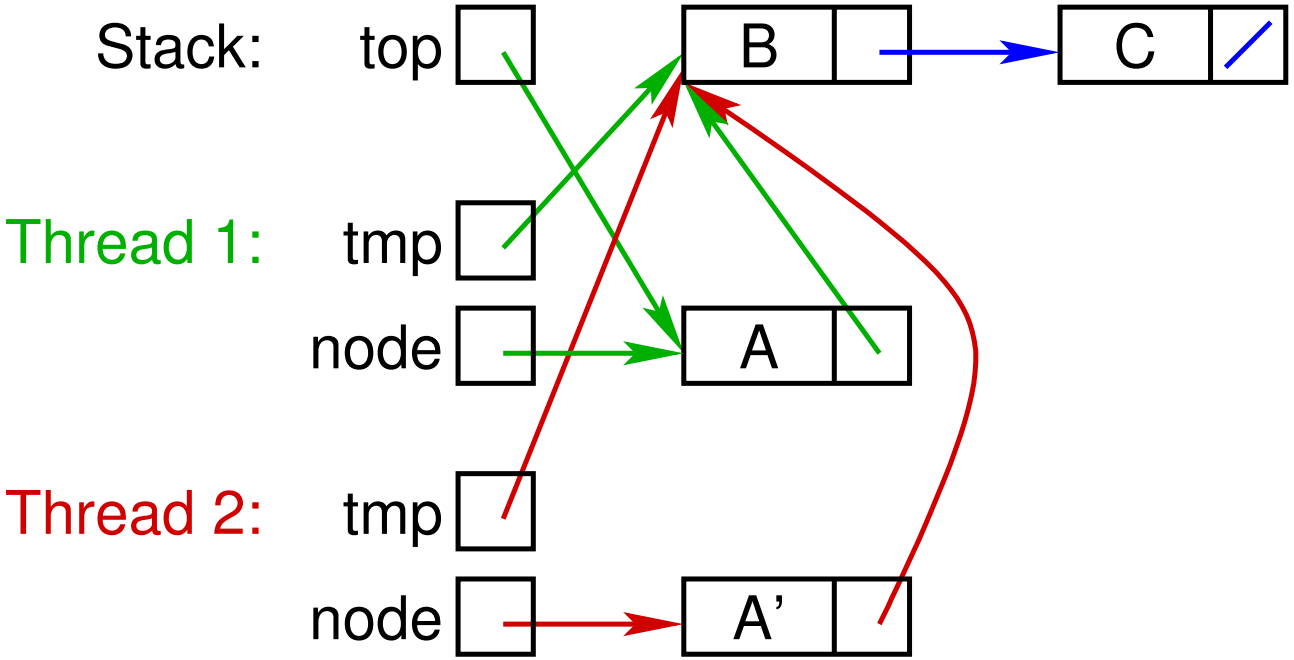

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



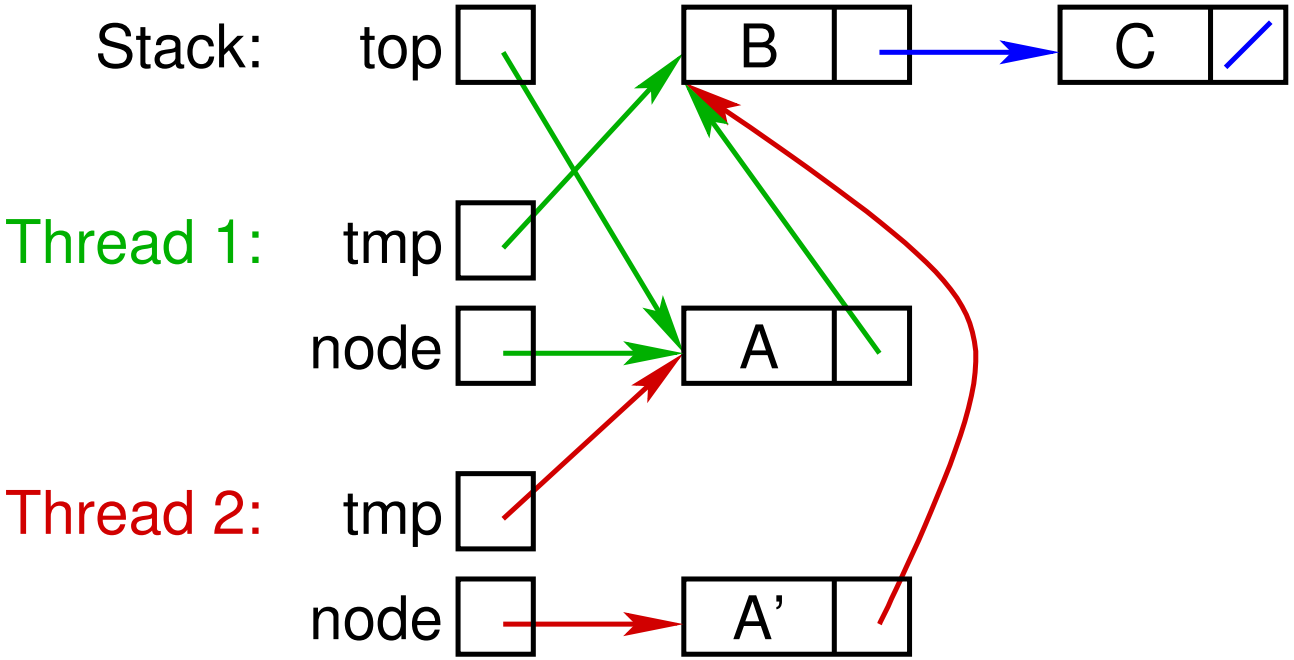

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



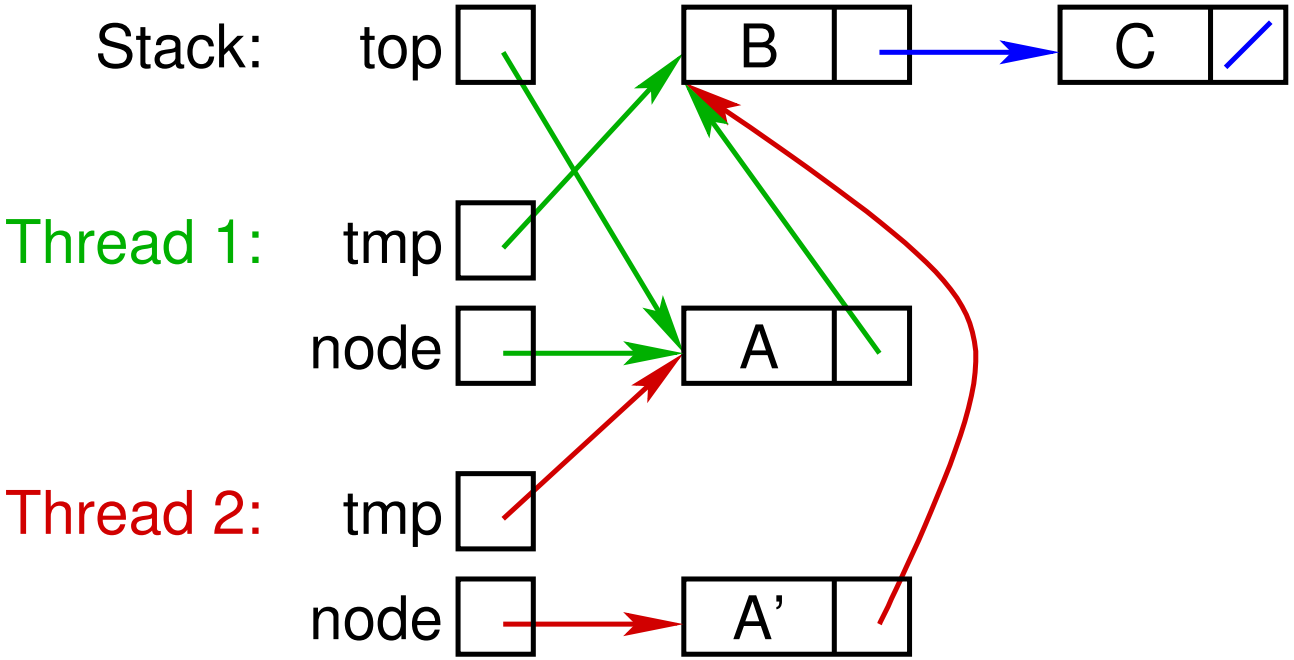

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



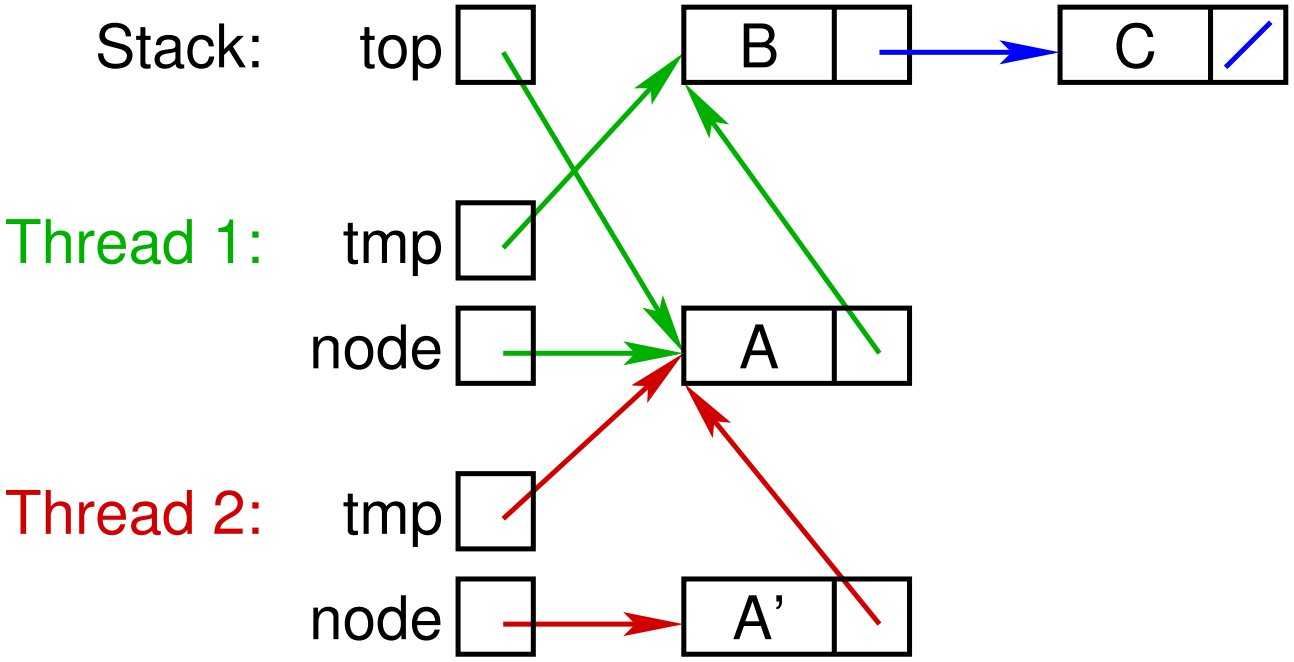

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



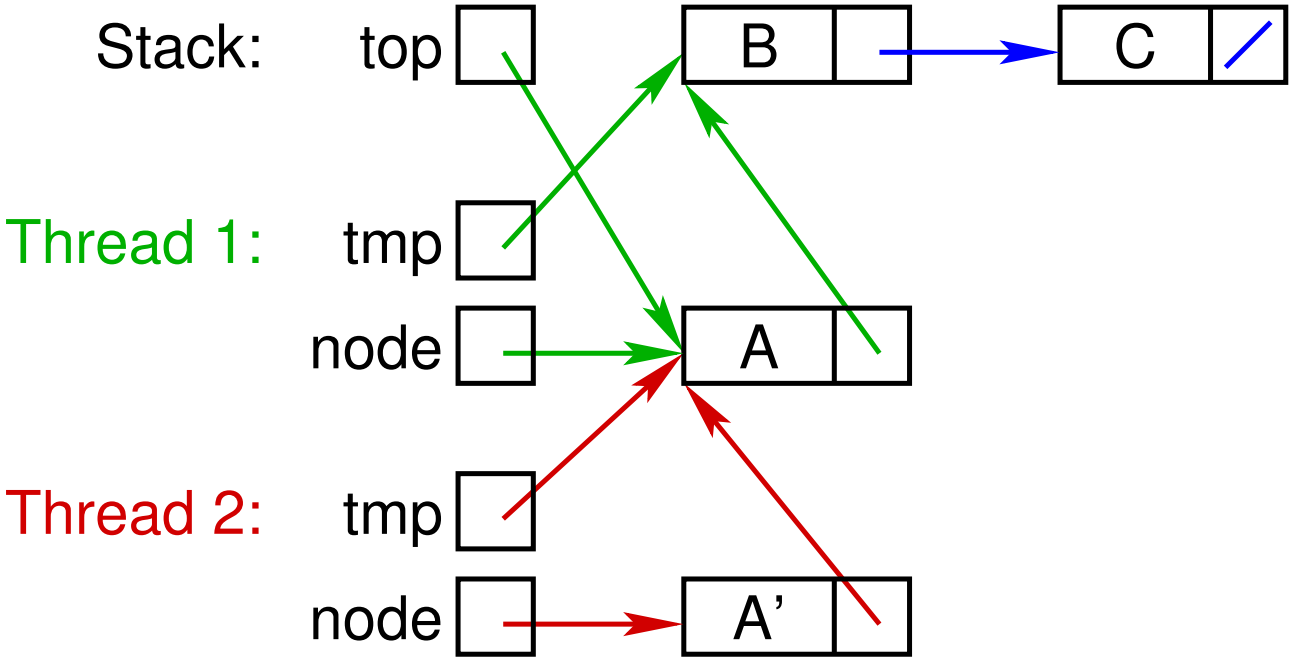

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



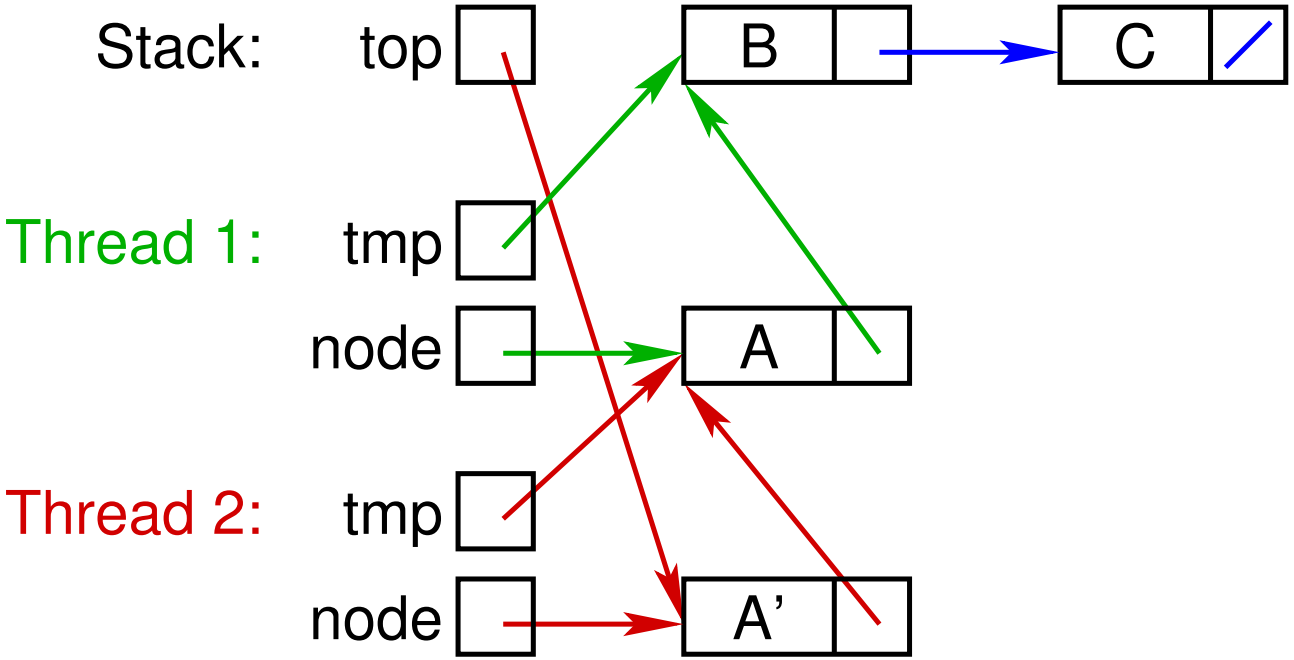

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel




```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel

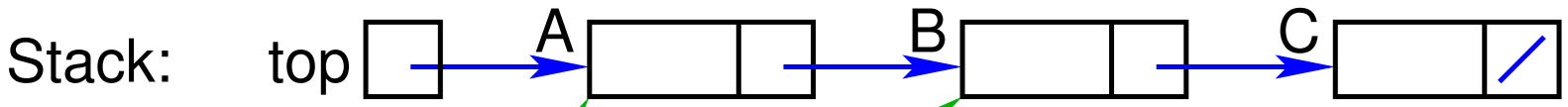



```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Das ABA-Problem

- *Lock-free* Implementierungen müssen Konflikte feststellen können
  - d.h.: wurde die Datenstruktur zwischenzeitlich von einem anderen Thread geändert?
  - falls ja: Operation wiederholen
- Dazu häufig: Test, ob eine bestimmte Referenz verändert wurde
  - beim Treiber Stack: Referenz auf oberstes Kellerelement
- Bei Sprachen ohne *Garbage-Collector* nicht ausreichend
  - z.B. beim Treiber Stack:
    - Entfernen / Freigeben der obersten beiden Elemente (A, B)
    - Schreiben eines neues Element auf den Stack
      - kann an selber Adresse allokiert sein, wie das alte A!
- Lösungen sind verfügbar, jedoch komplex

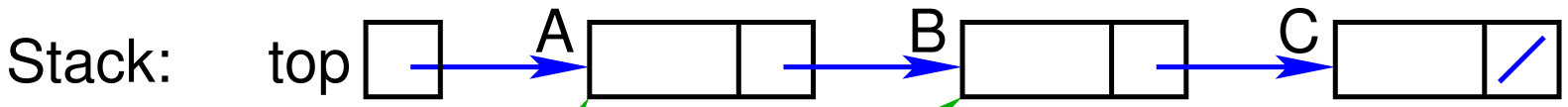
Das ABA Problem beim Treiber-Stack



Thread 1:

```
// in pop():
tmp = top;
next = tmp.next;
```

Das ABA Problem beim Treiber-Stack



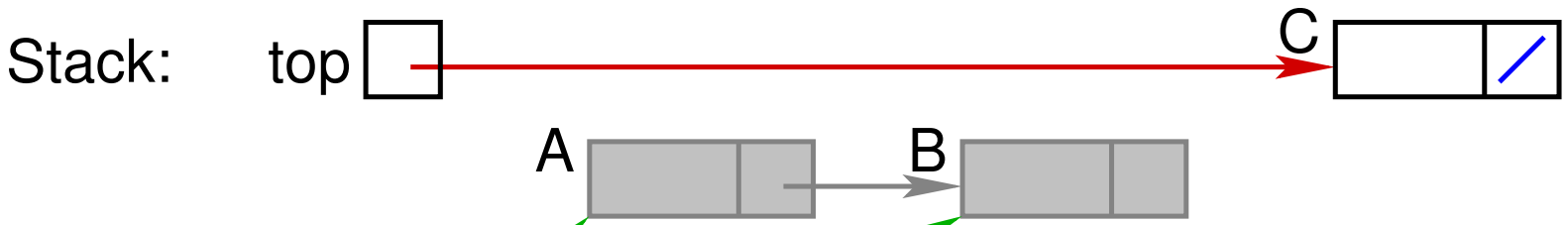
Thread 1:

```
// in pop():
tmp = top;
next = tmp.next;
```

Thread 2:

```
a = pop(); // gibt Speicherblock A frei
b = pop(); // gibt Speicherblock B frei
```

Das ABA Problem beim Treiber-Stack



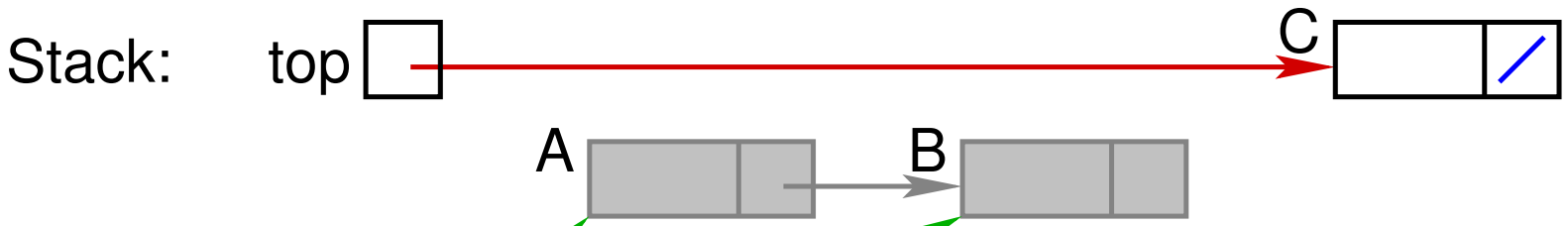
Thread 1:

```
// in pop():
tmp = top;
next = tmp.next;
```

Thread 2:

```
a = pop(); // gibt Speicherblock A frei
b = pop(); // gibt Speicherblock B frei
```

Das ABA Problem beim Treiber-Stack



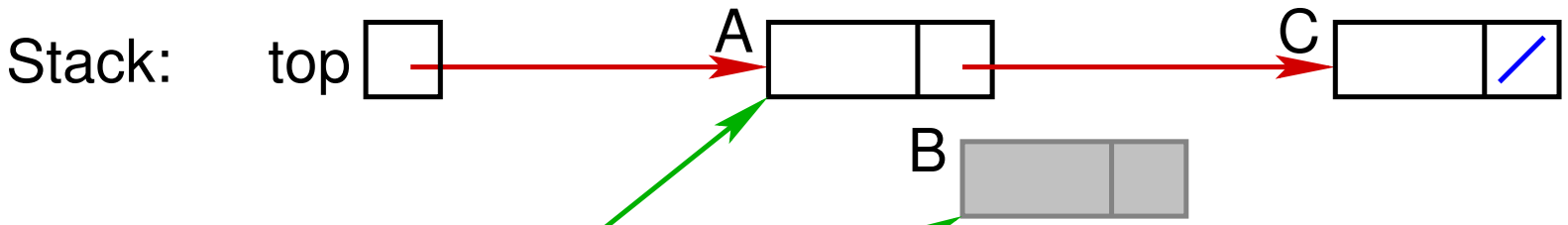
Thread 1:

```
// in pop():
tmp = top;
next = tmp.next;
```

Thread 2:

```
a = pop();  // gibt Speicherblock A frei
b = pop();  // gibt Speicherblock B frei
push(x);    // Allokiert Speicherblock
            // an Adresse A
```

Das ABA Problem beim Treiber-Stack



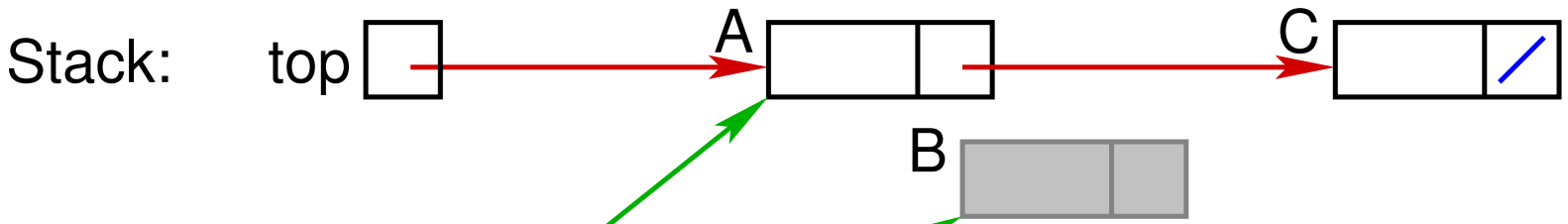
Thread 1:

```
// in pop():
tmp = top;
next = tmp.next;
```

Thread 2:

```
a = pop();  // gibt Speicherblock A frei
b = pop();  // gibt Speicherblock B frei
push(x);    // Allokiert Speicherblock
            // an Adresse A
```

Das ABA Problem beim Treiber-Stack

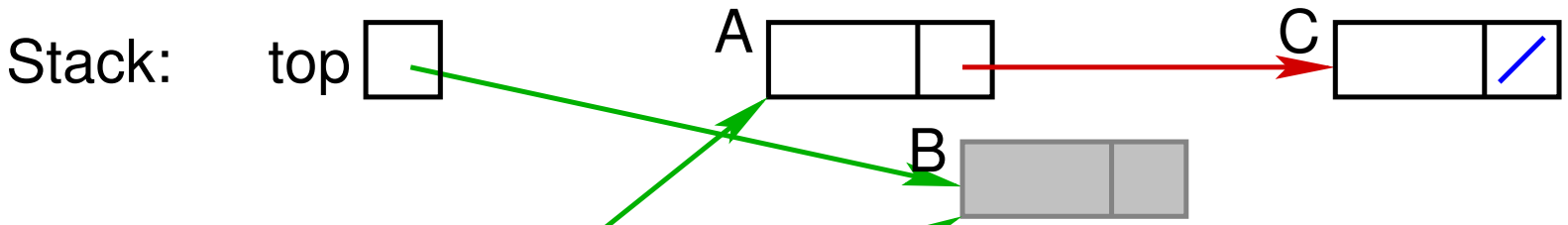


Thread 1:
```
// in pop():
tmp = top;
next = tmp.next;
```

Thread 2:
```
a = pop(); // gibt Speicherblock A frei
b = pop(); // gibt Speicherblock B frei
push(x);   // Allokiert Speicherblock
           // an Adresse A
```

Thread 1:
```
while (!compareAndSet(top, tmp, next));
```

Das ABA Problem beim Treiber-Stack



Thread 1:
```
// in pop():
tmp = top;
next = tmp.next;
```

Thread 2:
```
a = pop(); // gibt Speicherblock A frei
b = pop(); // gibt Speicherblock B frei
push(x);   // Allokiert Speicherblock
           // an Adresse A
```

Thread 1:
```
while (!compareAndSet(top, tmp, next));
```

3.11 *Lock-free* Datenstrukturen ...

- Es gibt etliche Bibliotheken mit *lock-free* bzw. *wait-free* Datenstrukturen für Java und C++
  - Java: z.B. Amino Concurrent Building Blocks, Highly Scalable Java
  - C++: z.B. boost.lockfree, libcds, Concurrency Kit, liblfds
- Programmiersprachen und/oder Compiler stellen *Read-Modify-Write*-Operationen bereit, z.B.:
  - Java: Paket `java.util.concurrent.atomic`
  - C++: Klasse `std::atomic<T>`
  - `gcc/g++`: eingebaute Funktionen `__sync_...()` bzw. `__atomic_...()`

3.12 Transactional Memory

- Probleme der bisherigen Ansätze:
  - Sperren: Performanz, fehleranfällig (z.B. Verklemmungen)
  - *Lock-free*: Einschränkung durch Wortbreite
  - Kompositionalität, z.B. atomares Verschieben eines Elements von einem Stack in einen anderen
- Lösungsidee: Nutzung von Transaktionen auf dem Speicher
- Dafür relevante Eigenschaften von Transaktionen:
  - *Atomicity*: entweder werden alle Operationen der Transaktion durchgeführt, oder keine
  - *Consistency*: Transaktionen erhalten die Konsistenz der Daten
  - *Isolation*: Ergebnis nebenläufig durchgeführten Transaktionen entspricht immer einer (beliebigen) sequentiellen Ausführung

Basisprimitive für *Transactional Memory*

- `atomic`-Blöcke
  - Anweisungen innerhalb des Blocks werden atomar und serialisierbar ausgeführt
    - in Bezug auf alle(!) anderen `atomic`-Blöcke
- `abort`-Befehl (innerhalb eines `atomic`-Blocks)
  - führt zum Abbruch der Transaktion
  - Verlassen des Blocks; Wiederherstellen des alten Zustands
- `retry`-Befehl (innerhalb eines `atomic`-Blocks)
  - Abbruch der Transaktion (mit Wiederherstellung)
  - neuer Versuch
- `orElse`-Befehl (optionale Erweiterung)
  - alternative Transaktion, falls erste mit `retry` abbricht

Beipiel: Warteschlangen

- Element aus Warteschlange entfernen:

```
Data dequeue() {
  atomic {                          // Atomare Ausführung
    if (first == null)              // Falls Liste leer:
      retry;                        //   Transaktion neu starten
    Data res = first.data;
    first = first.next;
    return res;
  }
}
```

- Atomares Verschieben zwischen zwei Listen:

```
atomic {
  q1.enqueue(q2.dequeue());
}
```

Beipiel: Warteschlangen ...

- Alternatives Lesen aus zwei Warteschlangen:

```
atomic {
  x = q1.dequeue();
}
orElse {
  x = q2.dequeue();
}
```

  - falls Transaktion in `q1.dequeue()` mit `retry` abbricht: versuche `q2.dequeue()`
  - falls diese auch mit `retry` abbricht: wieder von vorne

Transactional Memory Systeme

- Realisierungen in Software (Bibliotheken, Compiler) oder Hardware verfügbar
- Unterscheidung *weak / strong isolation*
  - werden in `atomic`-Blöcken genutzte Variablen auch ausserhalb der `atomic`-Blöcke geschützt?
- Unterschiedliche Behandlung geschachtelter Transaktionen:
  - flache Tr.: gesamte Tr. bricht ab, falls innere Tr. abbricht
  - offene Tr.: Ergebnisse erfolgreicher innerer Tr. sofort sichtbar (selbst wenn äußere Tr. abbricht)
  - geschlossene Tr.: Ergebnisse erfolgreicher innerer Tr. erst sichtbar, wenn äußere Tr. erfolgreich beendet ist

3.13 Zusammenfassung / Wiederholung

- Synchronisation
  - wechselseitiger Ausschluß
    - nur jeweils ein Thread darf im kritischen Abschnitt sein
    - kritischer Abschnitt: Zugriff auf gemeinsame Ressourcen
  - Lösungsansätze:
    - Sperren der Interrupts (nur im BS, Einprozessorsysteme)
    - Sperrvariable: Peterson-Algorithmus
    - mit Hardware-Unterstützung: *Read-Modify-Write*
    - Nachteil: Aktives Warten (Effizienz, Verklemmungsgefahr)

3.13 Zusammenfassung / Wiederholung ...

- Semaphor
  - besteht aus Zähler und Threadwarteschlange
    - `P()`: herunterzählen, ggf. blockieren
    - `V()`: hochzählen, ggf. blockierten Thread wecken
    - Atomare Operationen (im BS realisiert)
  - wechselseitiger Ausschluß:

Thread 0

```
P(Mutex);
// kritischer Abschnitt
V(Mutex);
```

Thread 1

```
P(Mutex);
// kritischer Abschnitt
V(Mutex);
```

  - auch für Reihenfolgesynchronisation nutzbar
    - Beispiel: Erzeuger/Verbraucher-Problem

3.13 Zusammenfassung / Wiederholung ...

- Monitor
  - Modul mit Daten, Prozeduren, Initialisierung
  - Datenkapselung
  - Prozeduren stehen unter wechselseitigem Ausschluß
  - Bedingungsvariable zur Synchronisation
    - `wait()` und `signal()`
  - Varianten bei der Signalisierung:
    - einen / alle wartenden Threads wecken?
    - erhält geweckter Thread sofot den Monitor?
      - falls nicht: Bedingung nach Rückkehr aus `wait()` erneut prüfen!

3.13 Zusammenfassung / Wiederholung ...

- Synchronisation in Java:
  - Klassen mit `synchronized` Methoden
    - wechselseitiger Ausschluß der Methoden (pro Objekt)
  - `wait()`, `notify()`, `notifyAll()`
    - genau eine (implizite) Bedingungsvariable pro Objekt
  - JDK 1.5: Semaphore, *Locks* und Bedingungsvariablen
    - *Locks* und Bedingungsvariable erlauben die genaue Nachbildung des Monitorkonzepts
      - *Locks* für wechselseitigen Ausschluß der Methoden
      - Bedingungsvariablen sind fest an *Lock* gebunden
      - mehrere Bedingungsvariablen pro Objekt möglich

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

4 Kommunikation

Inhalt:

- Einführung
- Elementare Kommunikationsmodelle
- Adressierung
- Varianten und Erweiterungen

- Tanenbaum 2.3.8, 8.2.3, 8.2.4
- Stallings 5.6, 6.7, 13.3
- Nehmer/Sturm 7

Methoden zur Kommunikation

- Speicherbasierte Kommunikation
  - über gemeinsamen Speicher (s. Erzeuger/Verbraucher-Problem)
  - i.d.R. zwischen Threads desselben Prozesses
  - gemeinsamer Speicher auch zwischen Prozessen möglich
  - Synchronisation muß explizit programmiert werden
- Nachrichtenbasierte Kommunikation
  - Senden / Empfangen von Nachrichten (über das BS)
  - i.d.R. zwischen Threads verschiedener Prozesse
  - auch über Rechnergrenzen hinweg möglich
  - implizite Synchronisation

Nachrichtenbasierte Kommunikation

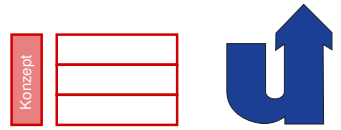
- Nachrichtenaustausch durch zwei Primitive:
  - `send(Ziel,Nachricht)` – Versenden einer Nachricht
  - `receive(Quelle,Nachricht)` – Empfang einer Nachricht
    - oft: spezieller Parameterwert für beliebige Quelle
      evtl. Quelle auch als Rückgabewert
- Implizite Synchronisation:
  - Empfang einer Nachricht erst **nach** dem Senden möglich
  - `receive` blockiert, bis Nachricht vorhanden ist
    - manchmal zusätzlich auch nichtblockierende `receive`-Operationen; ermöglicht *Polling*

Beispiel: Erzeuger/Verbraucher-Kommunikation mit Nachrichten

- Typisch: BS puffert Nachrichten bis zum Empfang
  - Puffergröße wird vom BS bestimmt (meist konfigurierbar)
  - falls Puffer voll ist: Sender wird in `send()`-Operation blockiert (Flußkontrolle, ☞ Rechnernetze I)

```
void producer() {
  int item;
  Message m;
  while (true) {
    item = produceItem() ;
    build_message(&m, item);
    send(consumer, m);
  }
}
```

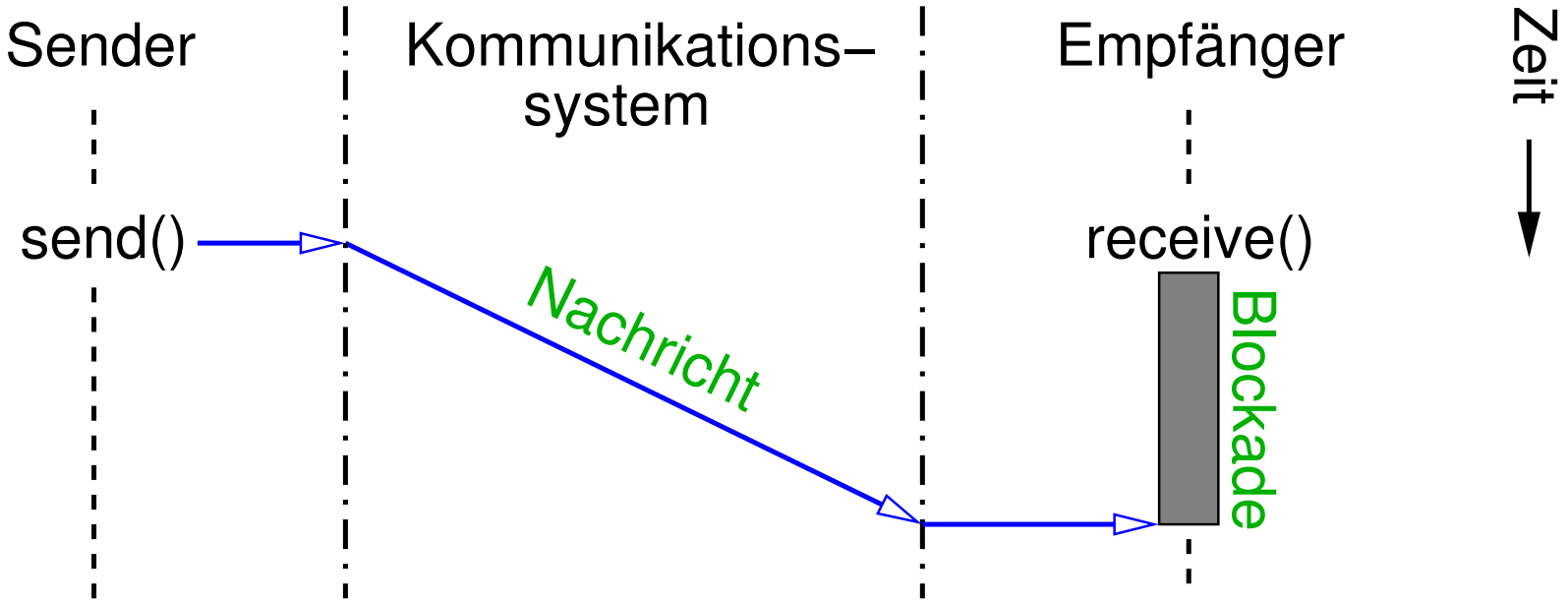
```
void consumer() {
  int item;
  Message m;
  while(true) {
    receive(producer, &m);
    item = extractItem(m);
    consumeItem(item);
  }
}
```



Klassifikation (nach Nehmer/Sturm)

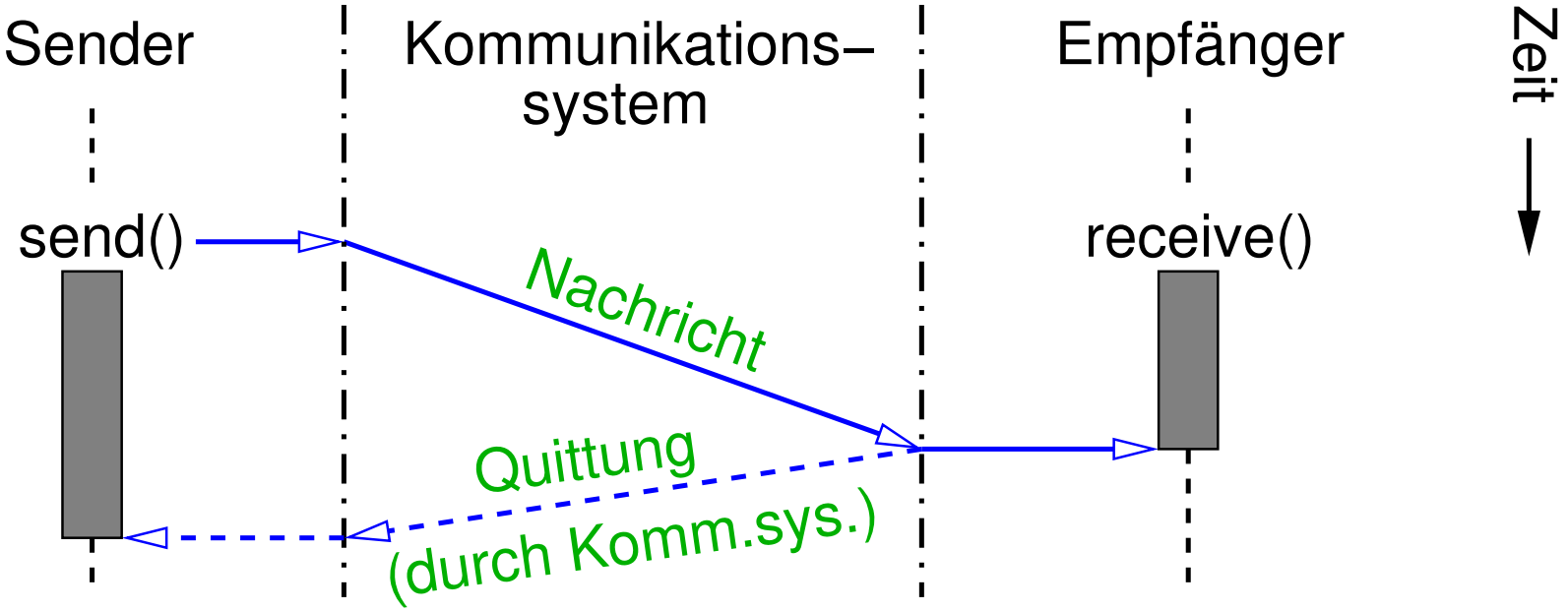
- Zeitliche Kopplung der Kommunikationspartner:
  - synchrone vs. asynchrone Kommunikation
  - auch: blockierende vs. nicht-blockierende Kommunikation
  - wird der Sender blockiert, bis der Empfänger die Nachricht empfangen hat?
- Muster des Informationsflusses:
  - Meldung vs. Auftrag
  - Einweg-Nachricht oder Auftragserteilung mit Ergebnis?

Asynchrone Meldung



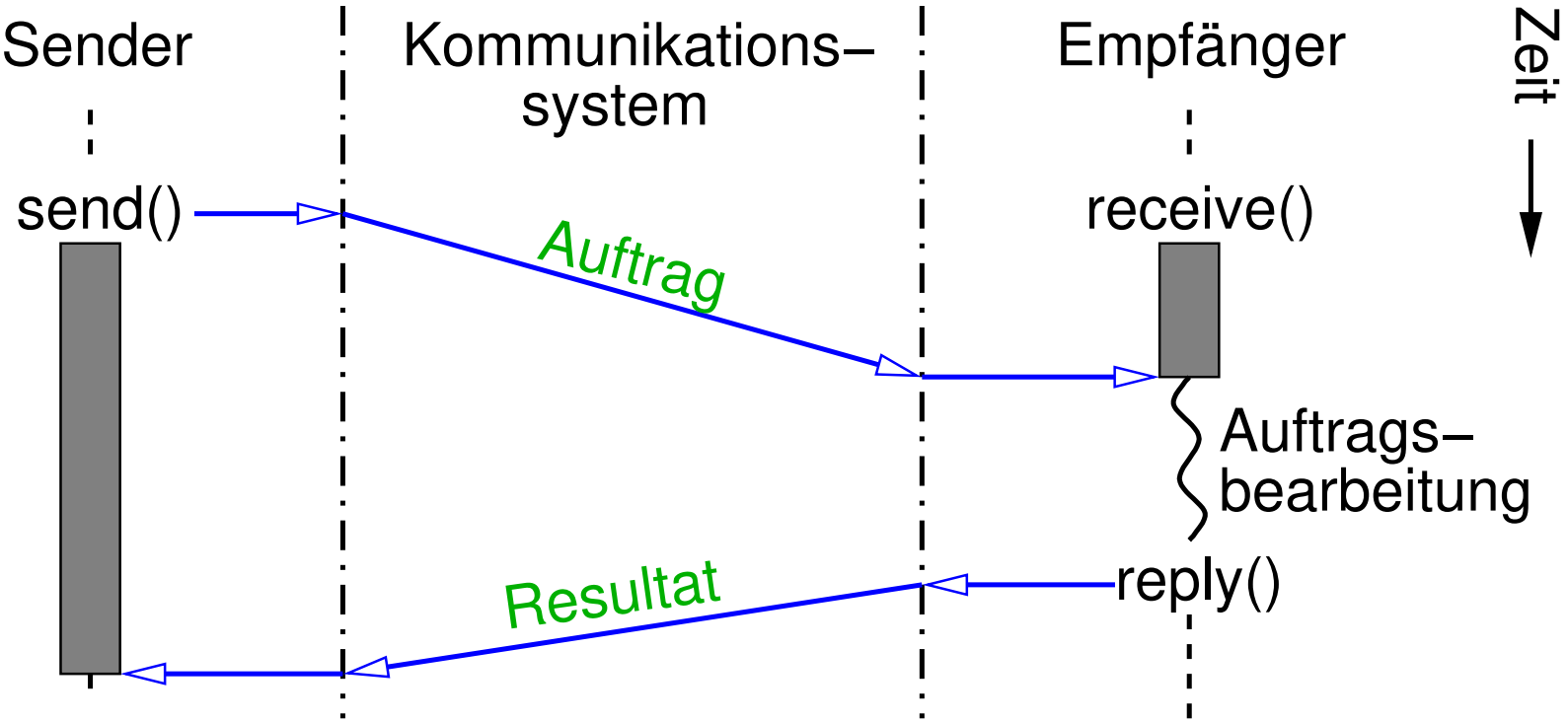
- Sender wird nicht mit Empfänger synchronisiert
- Kommunikationssystem (BS) muß Nachricht ggf. puffern, falls Empfänger (noch) nicht auf Nachricht wartet

Synchrone Meldung



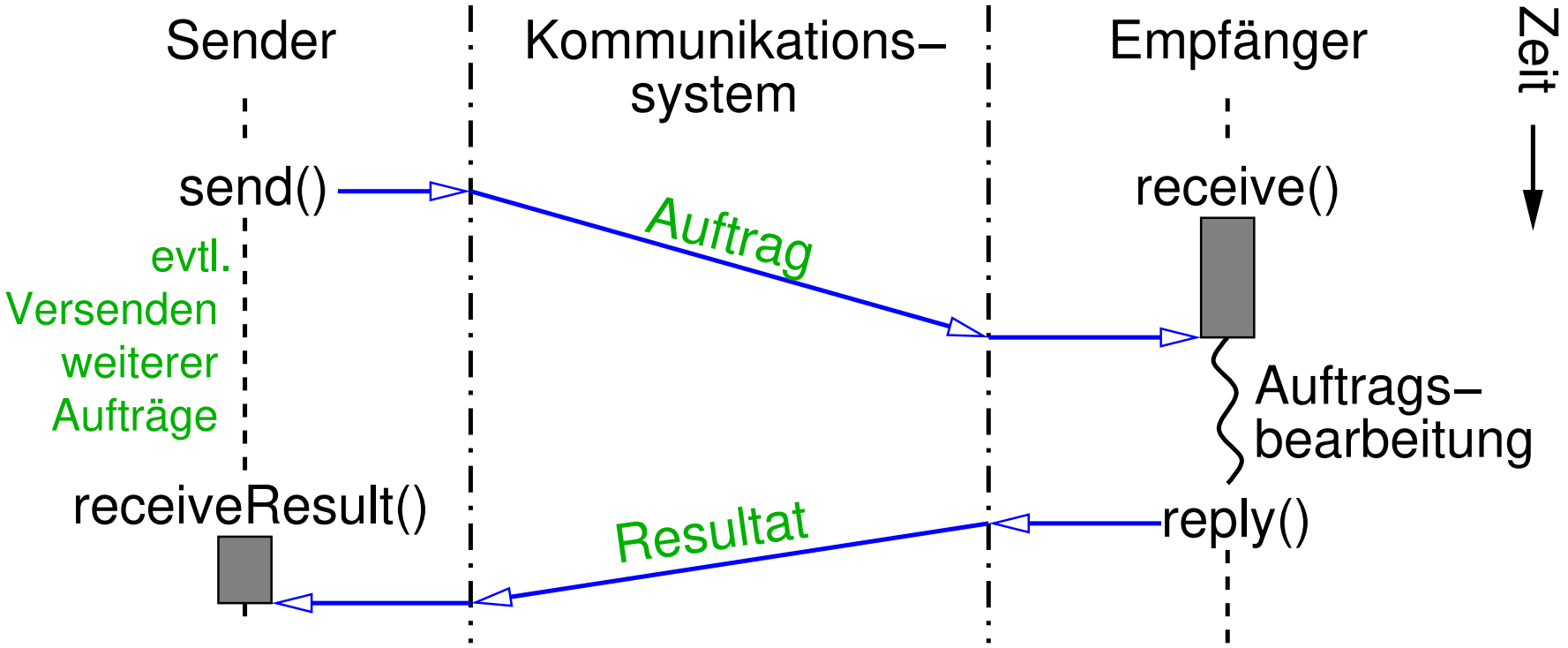
- Sender wird blockiert, bis Nachricht empfangen ist
- **Rendezvous**-Technik
- Keine Pufferung erforderlich

Synchroner Auftrag



- Empfänger sendet Resultat zurück
- Sender wird blockiert, bis Resultat vorliegt

Asynchroner Auftrag



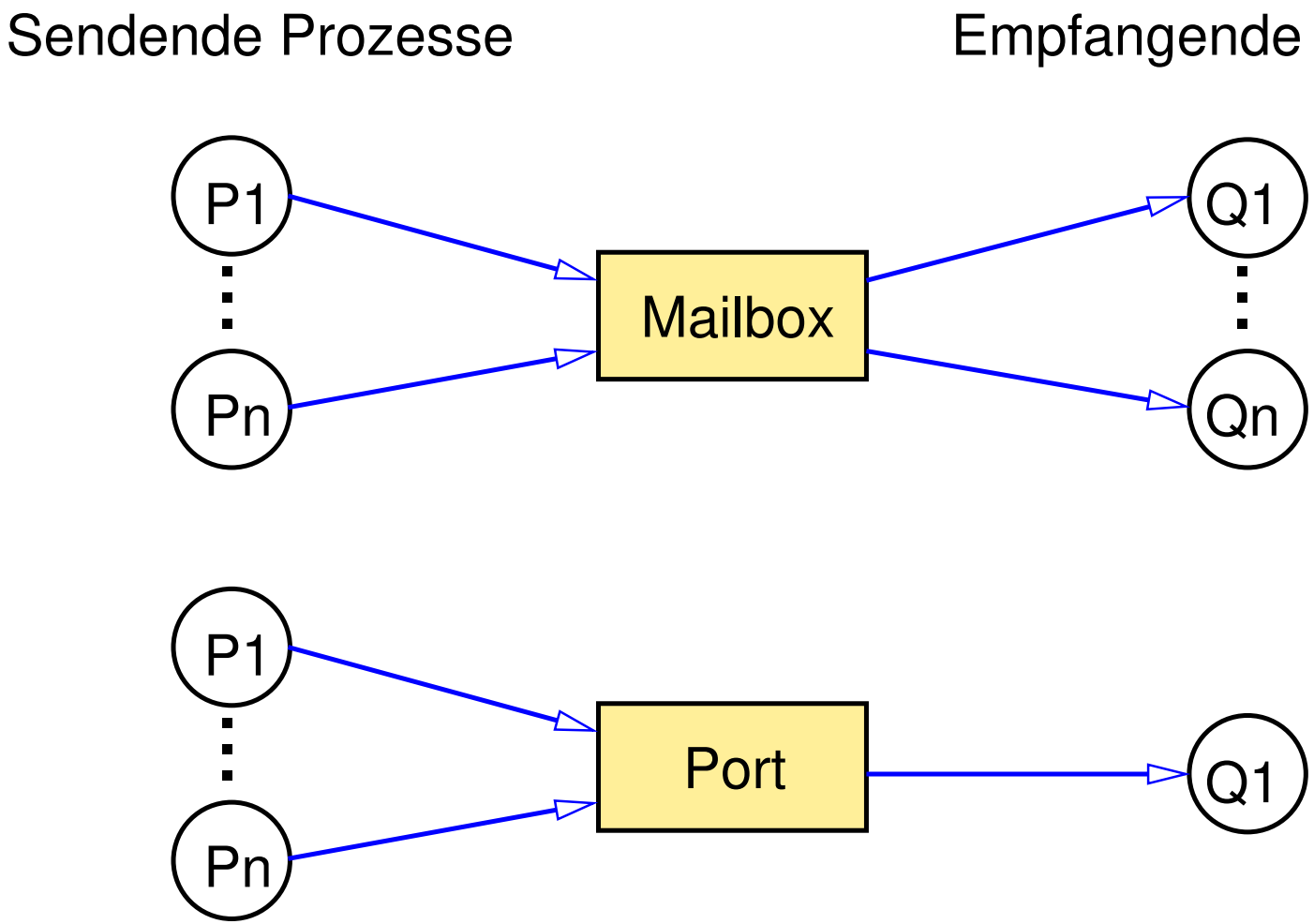
- Sender kann mehrere Aufträge gleichzeitig erteilen
  - Parallelverarbeitung möglich

Wie werden Sender bzw. Empfänger festgelegt?

- Direkte Adressierung
  - Kennung des jeweiligen Prozesses
- Indirekte Adressierung
  - Nachrichten werden an Warteschlangen-Objekt (**Mailbox**) gesendet und von dort gelesen
  - Vorteil: höhere Flexibilität
    - Mailbox kann von mehreren Prozessen gelesen werden
      - **Port**: Mailbox mit nur einem möglichen Empfänger-Prozess
    - Empfänger kann mehrere Mailboxen / Ports besitzen

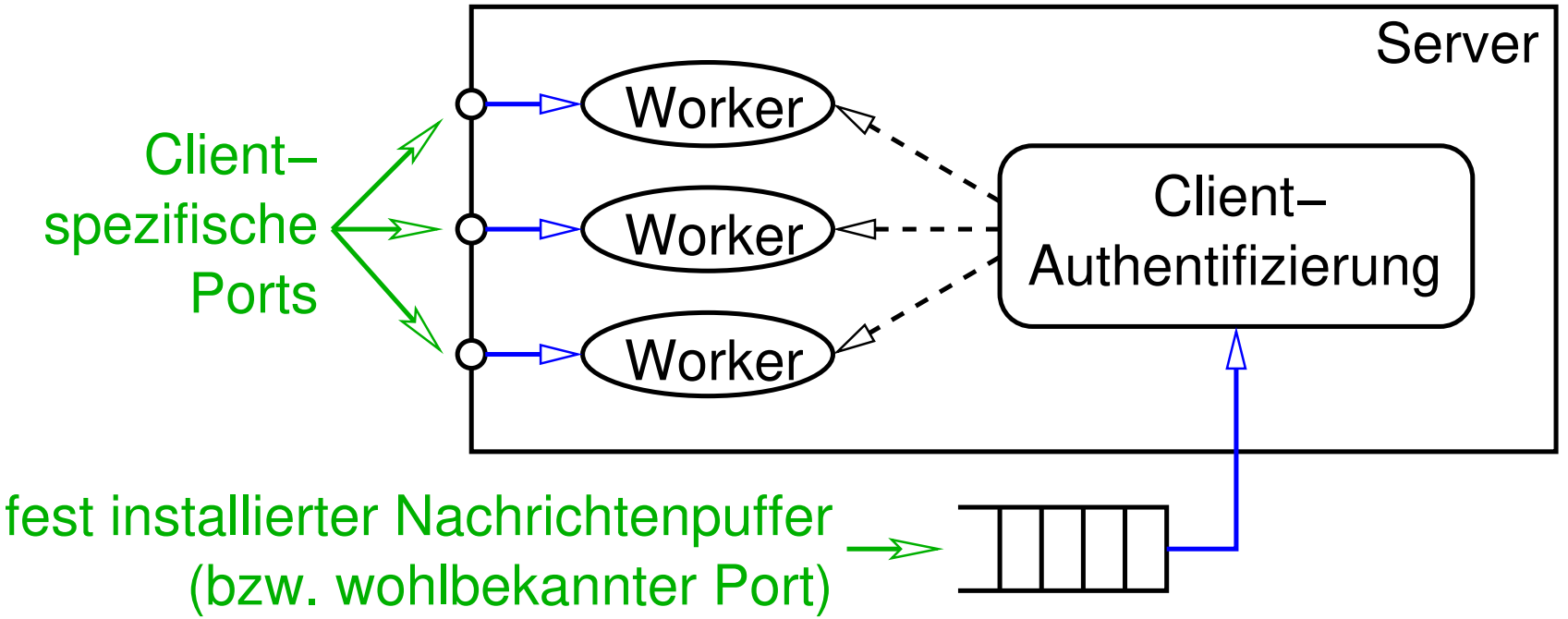
(Anm.: Für Sender und Empfänger werden nur Prozesse betrachtet)

Mailbox und Port



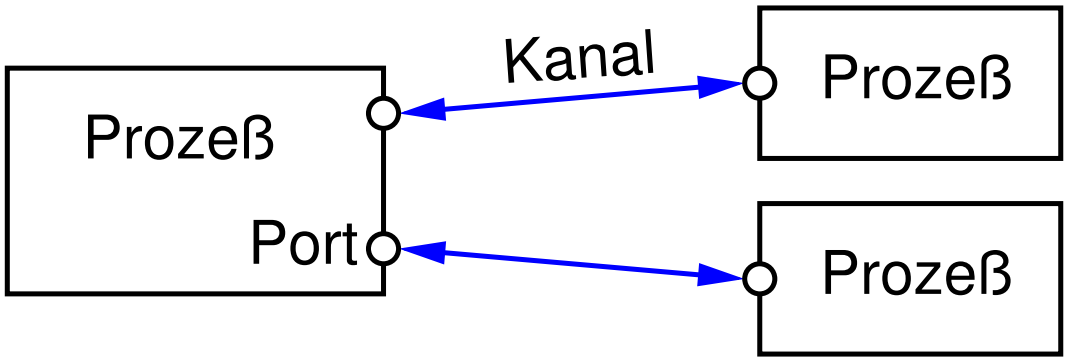
Anwendung von Ports: z.B. selektiver Nachrichtenempfang

- Server kann nach Anmeldung eines Clients für jeden Client einen eigenen Port erzeugen
  - jeder Port kann durch einen eigenen Worker-Prozeß (oder Thread) bedient werden



Kanäle

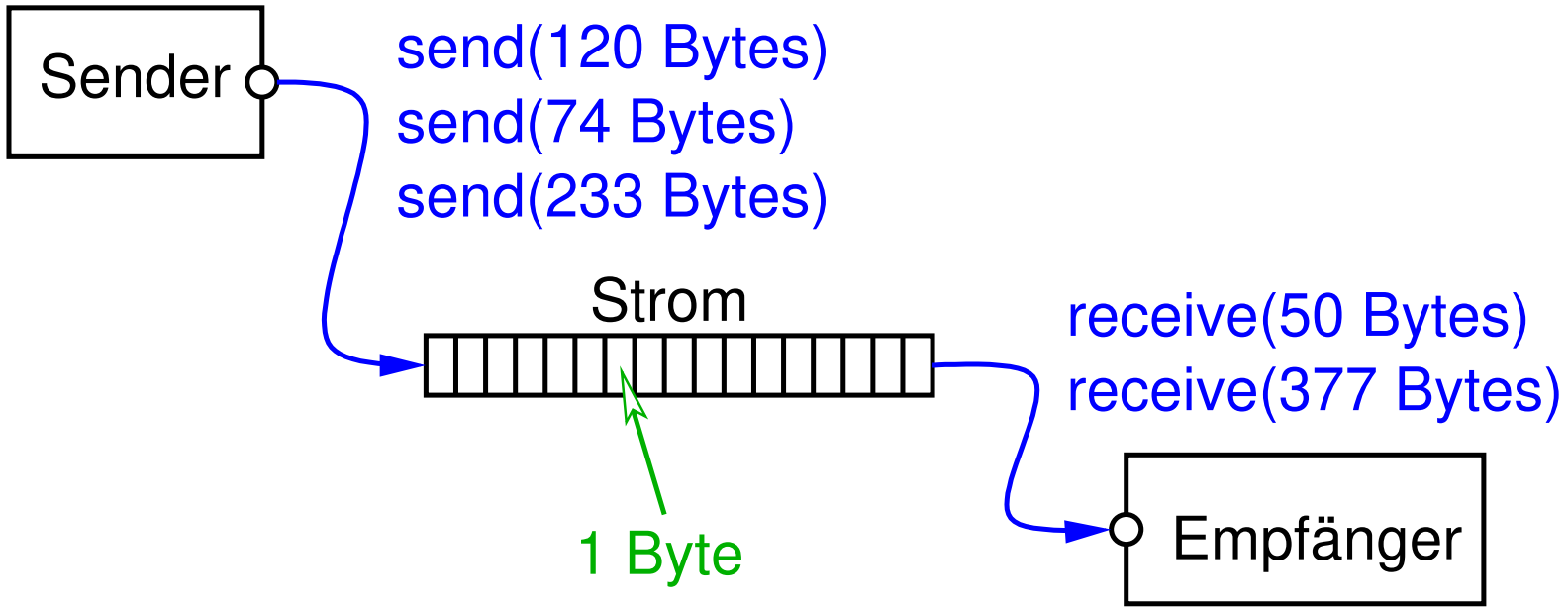
- Bisher: verbindungslose Kommunikation
  - wer Adresse eines Ports kennt, kann senden
- Kanal: logische Verbindung zw. zwei Kommunikationspartnern



  - expliziter Auf- und Abbau einer Verbindung (zw. zwei Ports)
  - meist bidirektional: Ports für Senden und Empfangen
  - garantierte Nachrichtenreihenfolge (FIFO)
  - Beispiel: TCP/IP-Verbindung

Ströme (*Streams*)

- Kanal zur Übertragung von Sequenzen von Zeichen (Bytes)
- Keine Nachrichtengrenzen oder Längenbeschränkungen
- Beispiele: TCP/IP-Verbindung, UNIX *Pipes*, Java *Streams*



Ströme in POSIX: *Pipes*

- *Pipe*: Unidirektionaler Strom
- Schreiben von Daten in die Pipe:
  - `write(int pipe_desc, char *msg, int msg_len)`
    - `pipe_desc`: Dateideskriptor
  - bei vollem Puffer wird Schreiber blockiert
- Lesen aus der Pipe:
  - `int read(int pipe_desc, char *buff, int max_len)`
    - `max_len`: Größe des Empfangspuffers `buff`
    - Rückgabewert: Länge der tatsächlich gelesenen Daten
  - bei leerem Puffer wird Leser blockiert

Ströme in POSIX: Erzeugen einer *Pipe*

- Unbenannte (*unnamed*) *Pipe*:
  - ist zunächst nur im erzeugenden Prozeß bekannt
  - Dateideskriptoren werden an Kindprozesse vererbt
- Beispielcode:

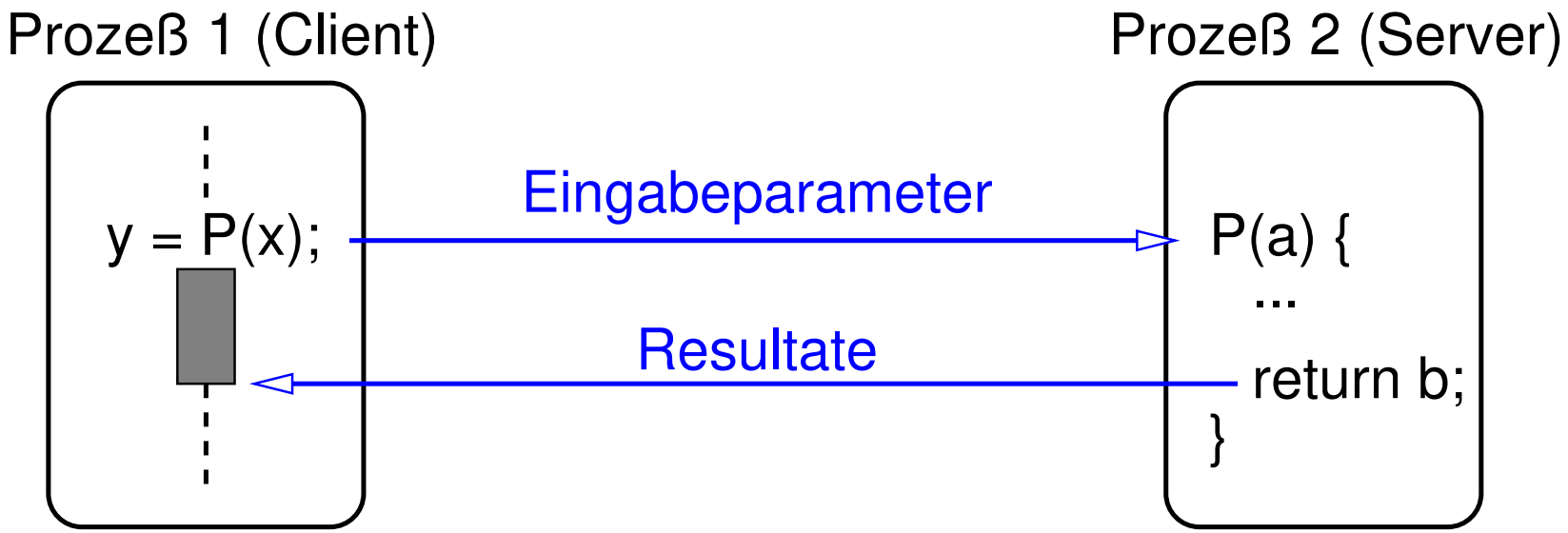
```
int pipe_ends[2];    // Dateideskriptoren der Pipe–Enden
pipe(pipe_ends);     // Erzeugen der Pipe
if (fork() != 0) {
   // Vaterprozeß
   write(pipe_ends[1], data, ...);
} else {
   // Kindprozeß (erbt Dateideskriptoren)
   read(pipe_ends[0], data, ...);
}
```

Ströme in POSIX: Erzeugen einer *Pipe* ...

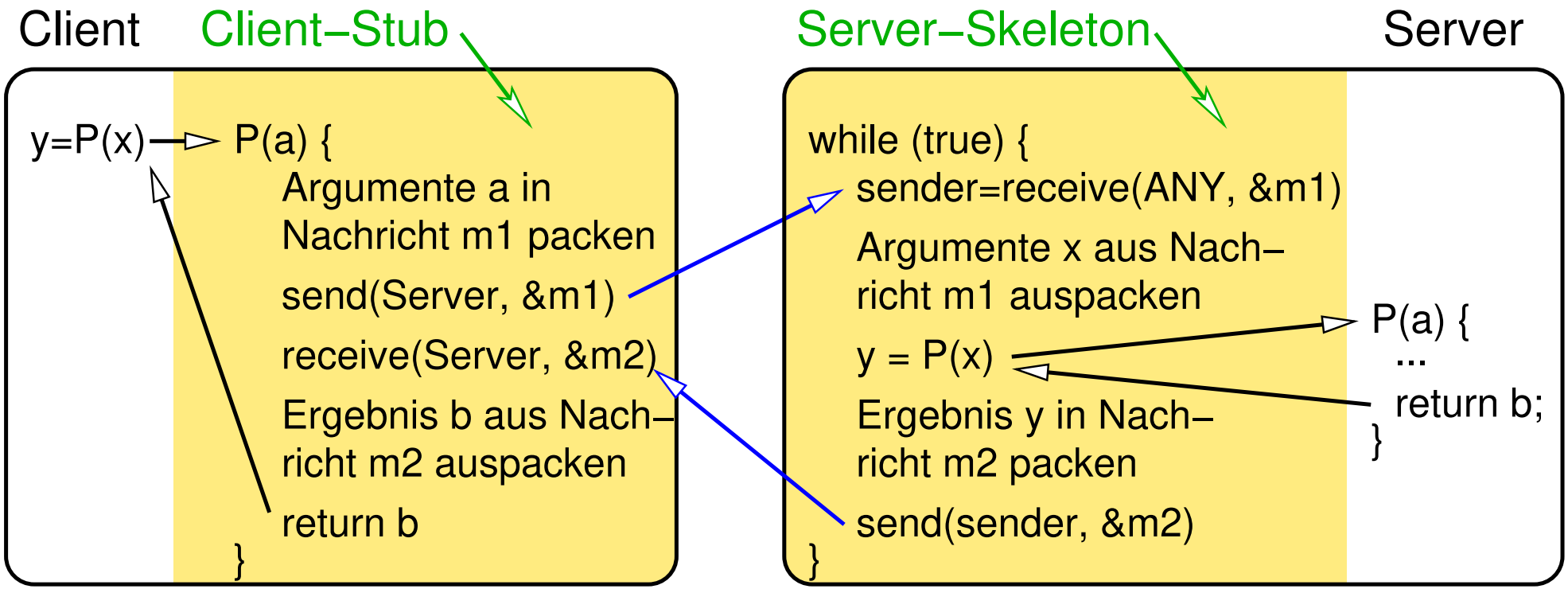
- Benannte (*named*) *Pipe*:
  - ist als spezielle „Datei“ im Dateisystem sichtbar
  - Zugriff erfolgt exakt wie auf normale Datei:
    - Systemaufrufe `open`, `close` zum Öffnen und Schließen
    - Zugriff über `read` und `write`
- Erzeugung:
  - Systemaufruf `mkfifo(char *name, mode_t mode)`
  - in Linux auch Shell-Kommando: `mkfifo <name>`
- Löschen durch normale Dateisystem-Operationen

Remote Procedure Call (RPC)

- Idee: Vereinfachung der Realisierung von synchronen Aufträgen
  - RPC: Aufruf einer Prozedur (Methode) in einem anderen Prozeß



Realisierung eines RPC



➥ Client-Stub und Server-Skeleton werden i.d.R. aus Schnittstellen-beschreibung generiert: **RPC-Compiler**

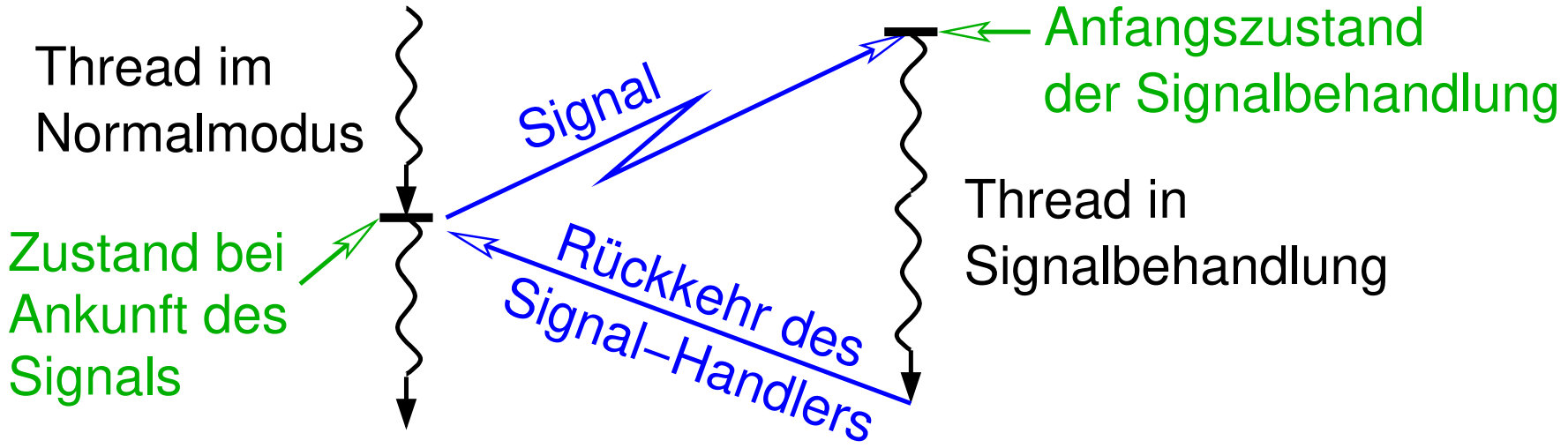
RPC: Diskussion

- Client muß sich zunächst an den richtigen Server binden
  - Namensdienste verwalten Adressen der Server
- Danach: RPC syntaktisch exakt wie lokaler Prozeduraufruf
- Semantische Probleme:
  - *Call-by-Reference* Parameterübergabe sehr problematisch
  - Kein Zugriff auf globale Variablen möglich
  - Nur streng getypte Schnittstellen verwendbar
    - Datentyp muß zum Ein- und Auspacken (***Marshaling***) genau bekannt sein
  - Behandlung von Fehlern in der Kommunikation?
- Beispiel: Java RMI

Signale

- Erlauben sofortige Reaktion des Empfängers auf eine Nachricht (asynchrones Empfangen)
- Asynchrone Unterbrechung des Empfänger-Prozesses
  - „Software-Interrupt“: BS-Abstraktion für Interrupts
- Operationen (abstrakt):
  - `Receive(Signalnummer, HandlerAdresse)`
  - `Signal(Empfänger, Signalnummer, Parameter)`
- Ideal: Erzeugung eines neuen Threads beim Eintreffen des Signals, der Signal-Handler abarbeitet
- Historisch: Signal wird durch unterbrochenen Thread behandelt

Behandlung eines Signals im unterbrochenen Thread

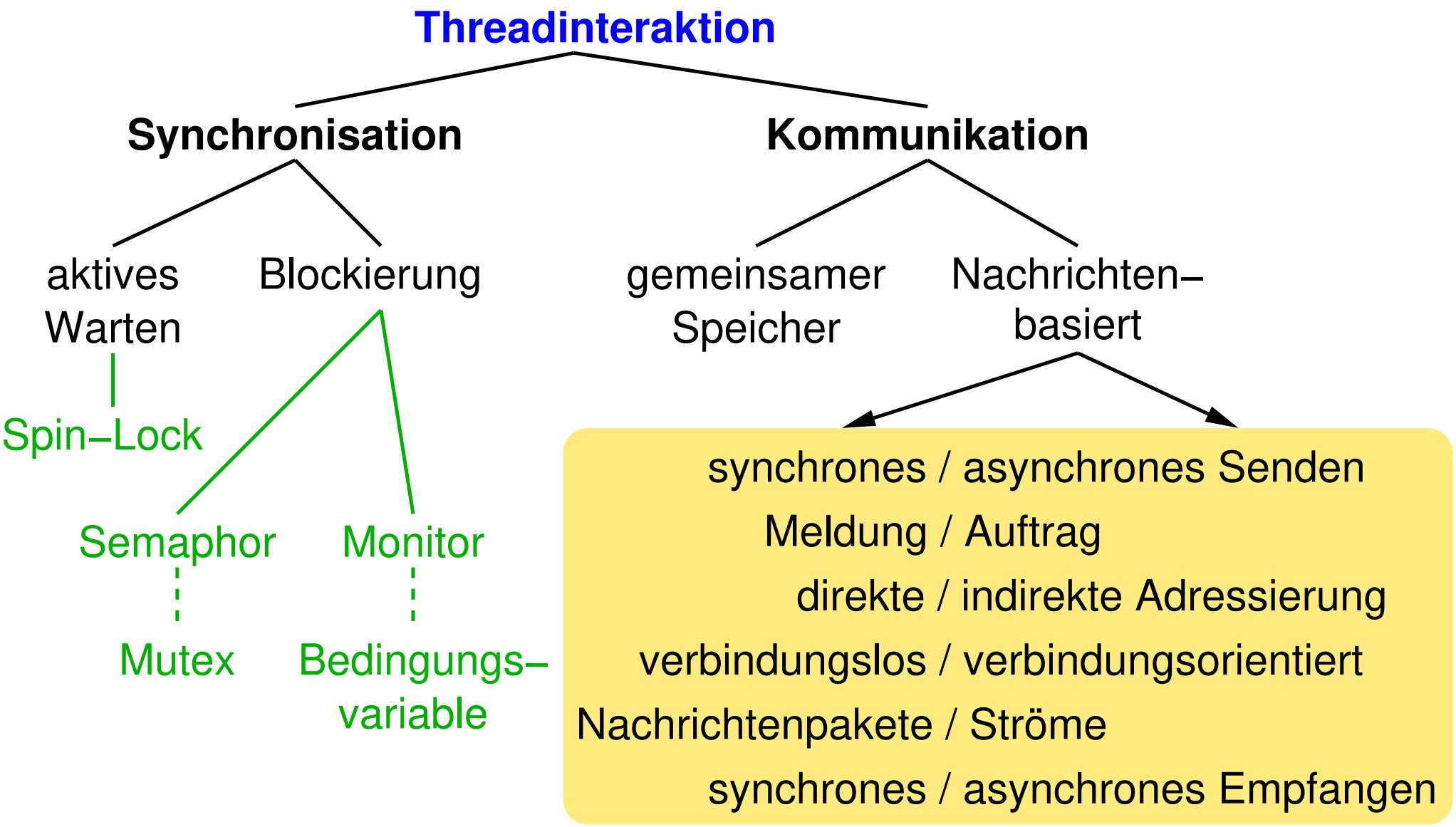


- BS sichert Threadzustand im Thread-Kontrollblock und stellt Zustand für Signalbehandlung her
- Kein wechselseitiger Ausschluß (z.B. Semaphore) möglich
  - ggf. müssen Signale gesperrt werden

Signale in POSIX

- Senden eines Signals an einen Prozeß (nicht Thread!):
  - `kill(pid_t pid, int signo)`
- Für jeden Signal-Typ ist eine Default-Aktion definiert, z.B.:
  - SIGINT: Terminieren des Prozesses (`^C`)
  - SIGKILL: Terminieren des Prozesses (nicht änderbar!)
  - SIGCHLD: Ignorieren (Kind-Prozeß wurde beendet)
  - SIGSTOP: Stoppen (Suspendieren) des Prozesses (`^Z`)
  - SIGCONT: Weiterführen des Prozesses (falls gestoppt)
- Prozeß kann Default-Aktionen ändern und eigene Signal-Handler definieren
- Handler wird von beliebigem Thread des Prozesses ausgeführt

4.5 Zusammenfassung / Wiederholung



4.5 Zusammenfassung / Wiederholung ...

- Nachrichtenbasierte Kommunikation
  - Primitive `send` und `receive`
  - synchrones / asynchrones Senden, Meldung / Auftrag
  - Mailboxen und Ports: indirekte Adressierung
  - Kanäle: logische Verbindung zwischen zwei Ports
    - Spezialfall: Ströme zur Übertragung von Sequenzen von Zeichen (z.B. POSIX Pipes)
  - RPC:
    - synchroner Auftrag, syntaktisch wie Prozeduraufruf
    - generierte Client- und Server-Stubs für *Marshaling*
  - Signale
    - asynchrone Benachrichtigung eines Prozesses
    - Unterbrechung eines Threads $\Rightarrow$ führt Signal-Hander aus

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

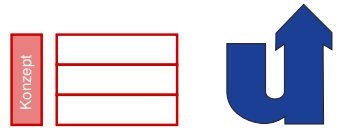
5 Verklemmungen (*Deadlocks*)

Inhalt:

- Einführung
- Formale Definition, Voraussetzungen
- Behandlung von Deadlocks:
  - Erkennung und Behebung
  - Vermeidung (*Avoidance*)
  - Verhinderung (*Prevention*)

- Tanenbaum 3
- Stallings 6.1-6.6
- Nehmer/Sturm 8

5.1 Einführung



- Wichtige Aufgabe des BSs: Verwaltung von Ressourcen
  - Ressourcen werden an Prozesse zugeteilt
  - Ressource kann nur jeweils von einem Prozeß genutzt werden
    - wechselseitiger Ausschluß
  - aber: mehrere identische Instanzen einer Ressource möglich
- Mögliches Problem: **Deadlock**, z.B.
  - Prozeß A hat Scanner belegt, benötigt CD-Brenner
  - Prozeß B hat CD-Brenner belegt, benötigt Scanner
  - A wartet auf B, B wartet auf A, ...

Unterbrechbare und ununterbrechbare Ressourcen

- **Unterbrechbare Ressource**
  - kann einem Prozeß ohne Schaden wieder entzogen werden
  - z.B. Prozessor (Sichern und Wiederherstellen des Prozessorzustands beim Threadwechsel)
  - z.B. Hauptspeicher (Auslagern auf Platte)
  - Deadlocks können verhindert werden
- **Ununterbrechbare Ressource**
  - kann einem Prozeß nicht entzogen werden, ohne daß seine Ausführung fehlschlägt
  - z.B. CD-Brenner
- Im Folgenden: ununterbrechbare Ressourcen

Anforderung von Ressourcen

- Systemabhängig, z.B.
  - explizite Anforderung über speziellen Systemauruf
  - implizit beim Öffnen eines Geräts (Systemaufruf `open`)
  - falls Ressource nicht verfügbar: Systemaufruf blockiert*
- Häufig wird „Zuteilung“ auch explizit programmiert
  - z.B. durch Nutzung von Semaphoren
- Umgekehrt ist auch Belegung eines Semaphors als Ressourcen-zuteilung interpretierbar
  - dann i.d.R. Threads statt Prozesse

* Vereinfachung in diesem Kapitel: Prozesse mit nur einem Thread, d.h. gesamter Prozess blockiert!

Beispiel: Nutzung von zwei Ressourcen

- Schutz durch zwei Semaphore

```
Semaphore resource1 = 1;
Semaphore resource2 = 1;
```

Thread A

```
P(resource1);
P(resource2);
UseBothResources();
V(resource2);
V(resource1);
```

Beispiel: Nutzung von zwei Ressourcen

- Schutz durch zwei Semaphore

```
Semaphore resource1 = 1;
Semaphore resource2 = 1;
```

Thread B

```
P(resource2);
P(resource1);
UseBothResources();
V(resource1);
V(resource2);
```

Beispiel: Nutzung von zwei Ressourcen

- Verklemmung möglich

```
Semaphore resource1 = 1;
Semaphore resource2 = 1;
```

Thread A

```
P(resource1);
P(resource2);
UseBothResources();
V(resource2);
V(resource1);
```

Thread B

```
P(resource2);
P(resource1);
UseBothResources();
V(resource1);
V(resource2);
```

Beispiel: Nutzung von zwei Ressourcen

- Verklemmung möglich

```
Semaphore resource1 = 1;
Semaphore resource2 = 1;
```

Thread A

```
P(resource1);
P(resource2);
UseBothResources();
V(resource2);
V(resource1);
```

Thread B

```
P(resource2);
P(resource1);
UseBothResources();
V(resource1);
V(resource2);
```

Beispiel: Nutzung von zwei Ressourcen

- Verklemmungsfreie Lösung

```
Semaphore resource1 = 1;
Semaphore resource2 = 1;
```

Thread A

```
P(resource1);
P(resource2);
UseBothResources();
V(resource2);
V(resource1);
```

Thread B

```
P(resource1);
P(resource2);
UseBothResources();
V(resource2);
V(resource1);
```

Definition

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

- Alle Prozesse* warten
- Die Ereignisse werden daher niemals ausgelöst
- Keiner der Prozesse* wird jemals wieder aufwachen

- Im BS-Kontext oft: Ereignis = Freigabe einer Ressource

* Bzw. Thread(s)

Bedingungen für einen Ressourcen-Deadlock

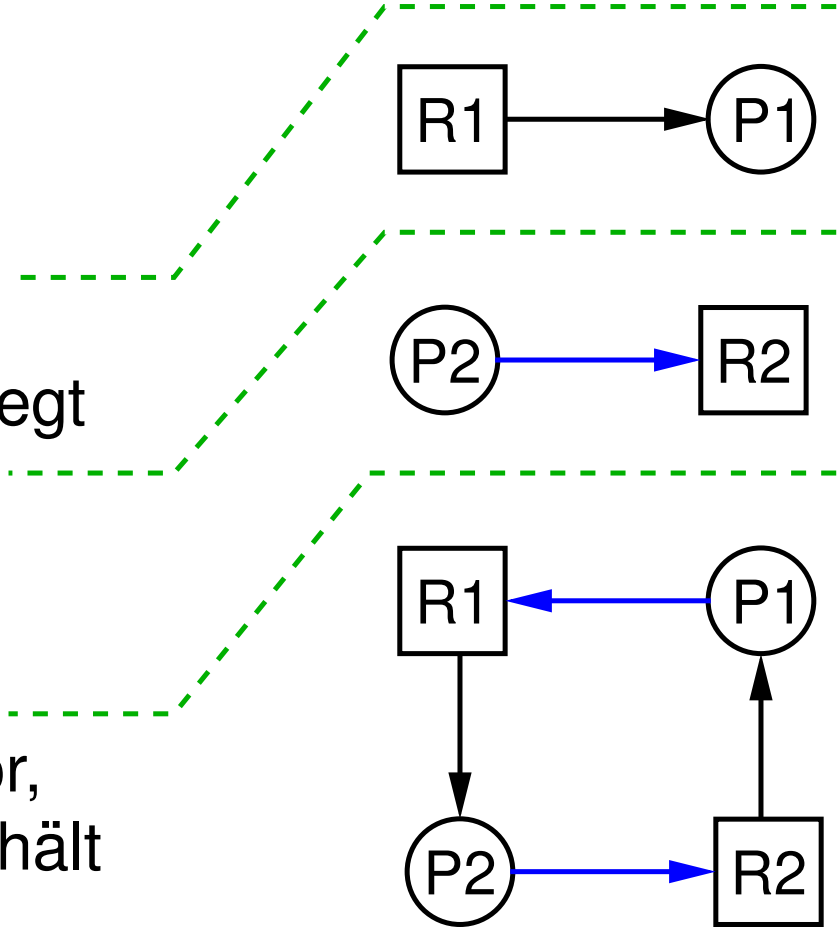
1. **Wechselseitiger Ausschluß**:
   Jede Ressource kann zu einem Zeitpunkt von höchstens einem Prozeß genutzt werden.
2. **Hold-and-Wait (Besitzen und Warten)**:
   Ein Prozeß, der bereits Ressourcen besitzt, kann noch weitere Ressourcen anfordern.
3. **Ununterbrechbarkeit (kein Ressourcenentzug)**:
   Einem Prozeß, der im Besitz einer Ressource ist, kann diese nicht gewaltsam entzogen werden.
4. **Zyklisches Warten**:
   Es gibt eine zyklische Kette von Prozessen, bei der jeder Prozeß auf eine Ressource wartet, die vom nächsten Prozeß in der Kette belegt ist.

Bedingungen für einen Deadlock ...

- Anmerkungen:
  - Bedingungen 1-3 sind **notwendige** Bedingungen, aber nicht hinreichend
  - Bedingung 4 ist eine potentielle Konsequenz aus 1-3
    - die Unauflösbarkeit des zyklischen Wartens ist eine Folge aus 1-3
  - Alle vier Bedingungen zusammen sind **notwendig** und **hinreichend** für eine Verklemmung
- Konsequenz:
  - wenn eine der Bedingungen unerfüllbar ist, können keine Deadlocks auftreten

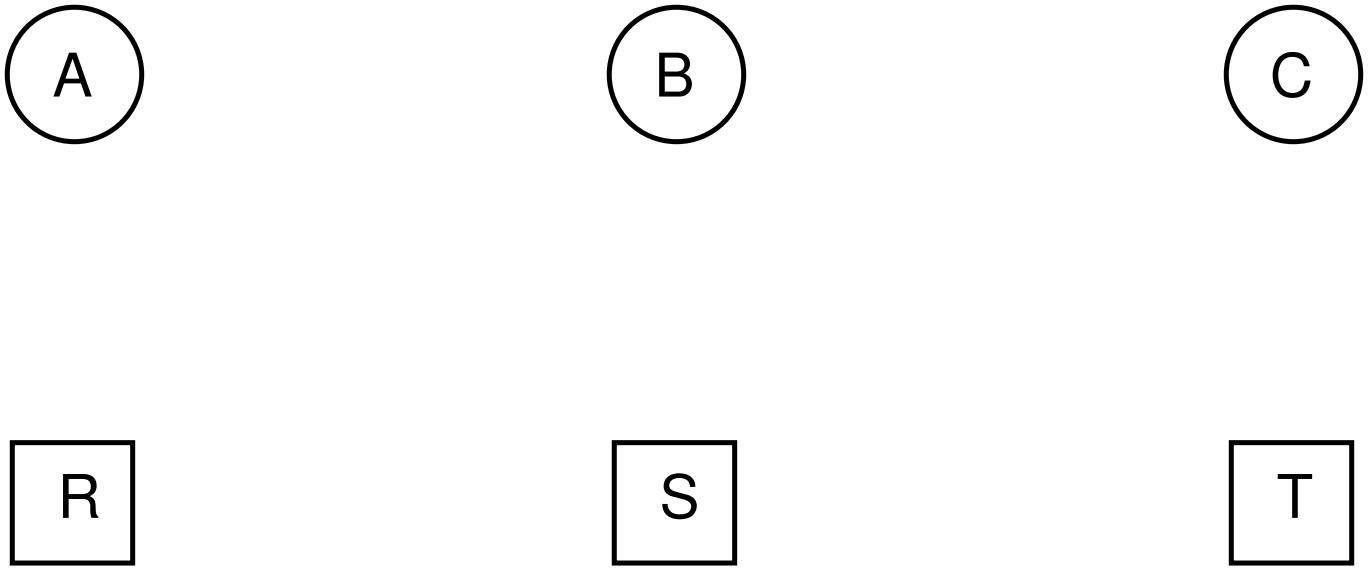
Modellierung von Deadlocks

- **Belegungs-Anforderungs-Graph**:
  - gerichteter Graph
  - zwei Arten von Knoten:
    - Prozesse (◯)
    - Ressourcen (□)
  - Kante Ressource → Prozeß:
    Ressource ist vom Prozeß belegt
  - Kante Prozeß → Ressource:
    Prozeß wartet auf Zuteilung
    der Ressource
- Ein Deadlock liegt genau dann vor,
  wenn der Graph einen Zyklus enthält



Beispiel: Ablauf mit Verklemmung

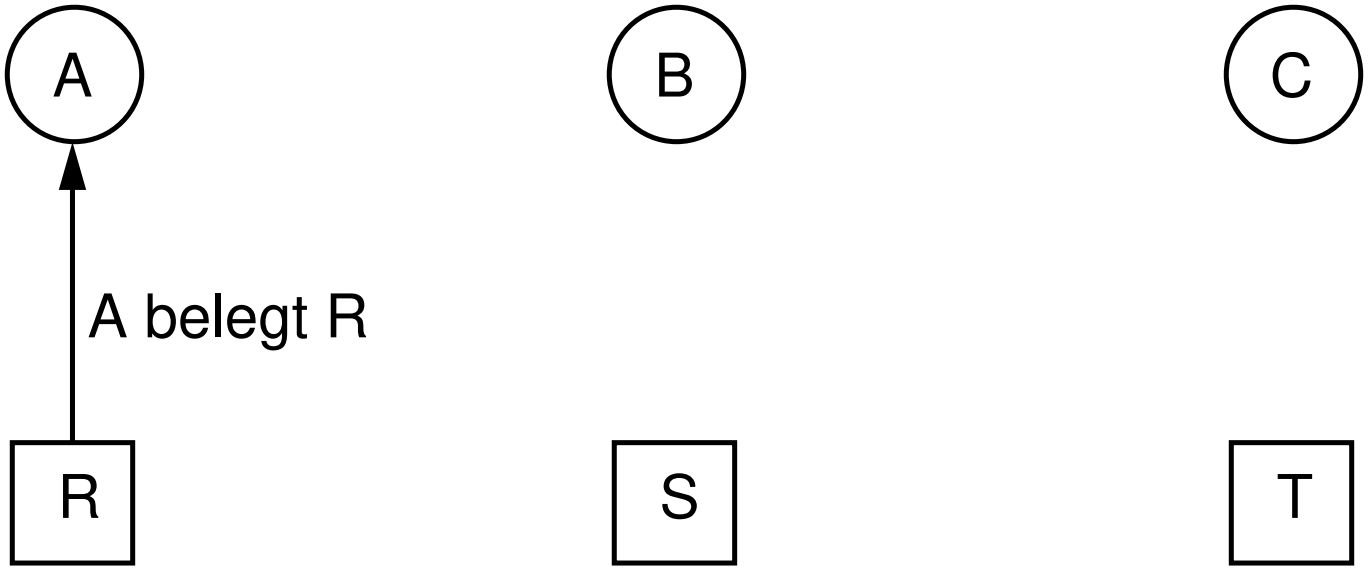
A	B	C
Anfrage R	Anfrage S	Anfrage T
Anfrage S	Anfrage T	Anfrage R
Freigabe R	Freigabe S	Freigabe T
Freigabe S	Freigabe T	Freigabe R



Reihenfolge: A–B–C–A–B–C...

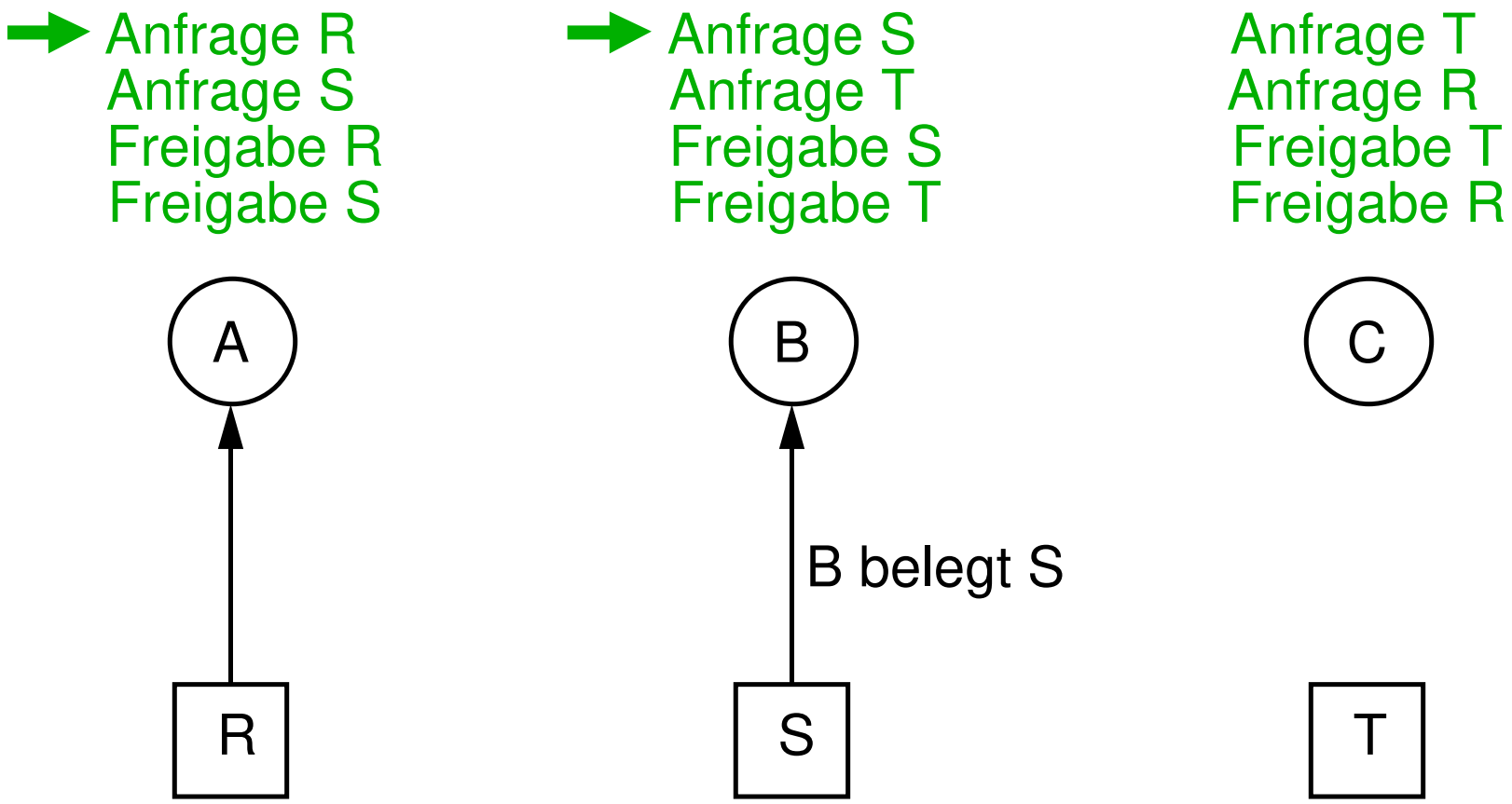
Beispiel: Ablauf mit Verklemmung

Anfrage R	Anfrage S	Anfrage T
Anfrage S	Anfrage T	Anfrage R
Freigabe R	Freigabe S	Freigabe T
Freigabe S	Freigabe T	Freigabe R



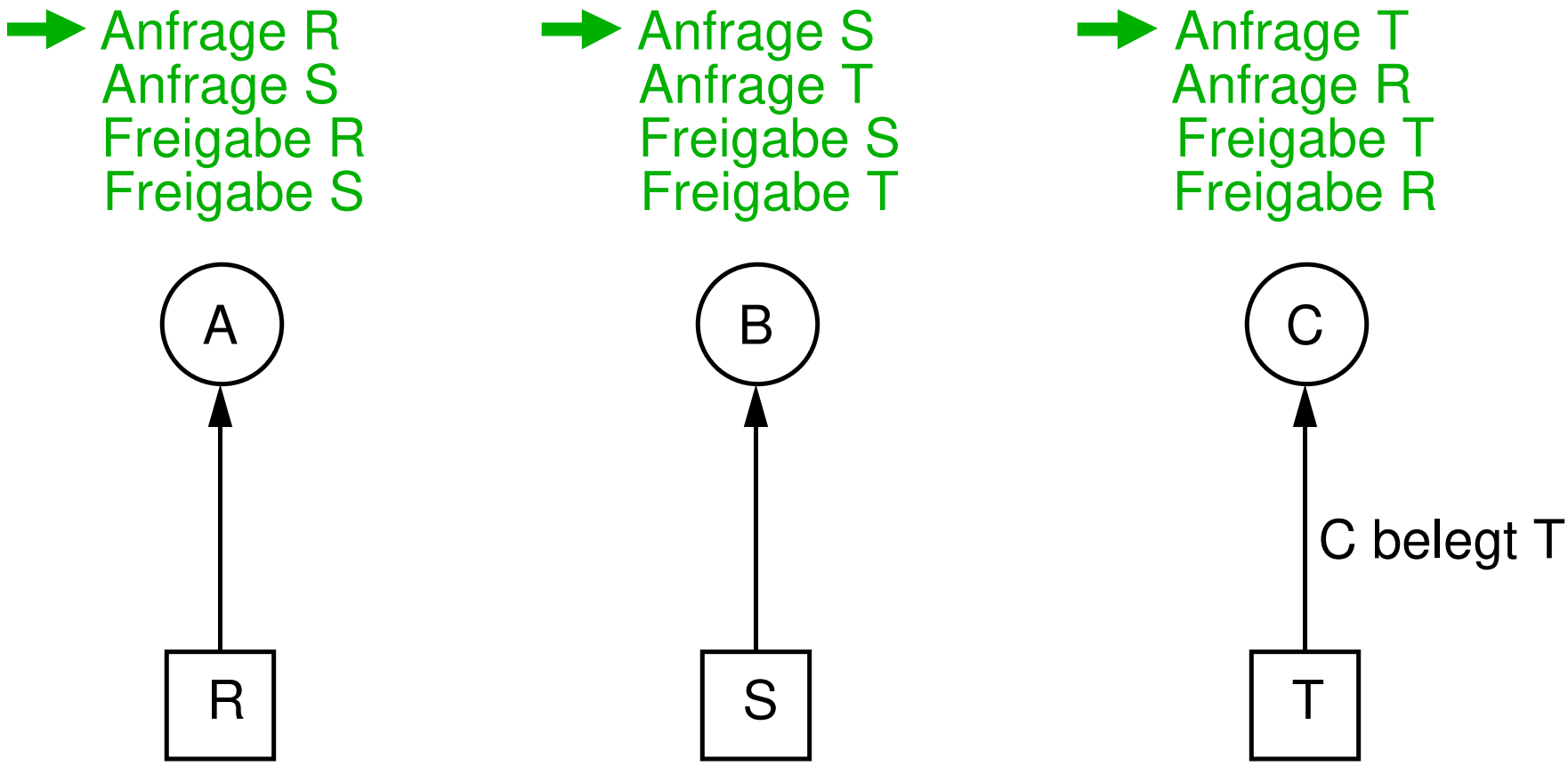
Reihenfolge: A–B–C–A–B–C...

Beispiel: Ablauf mit Verklemmung



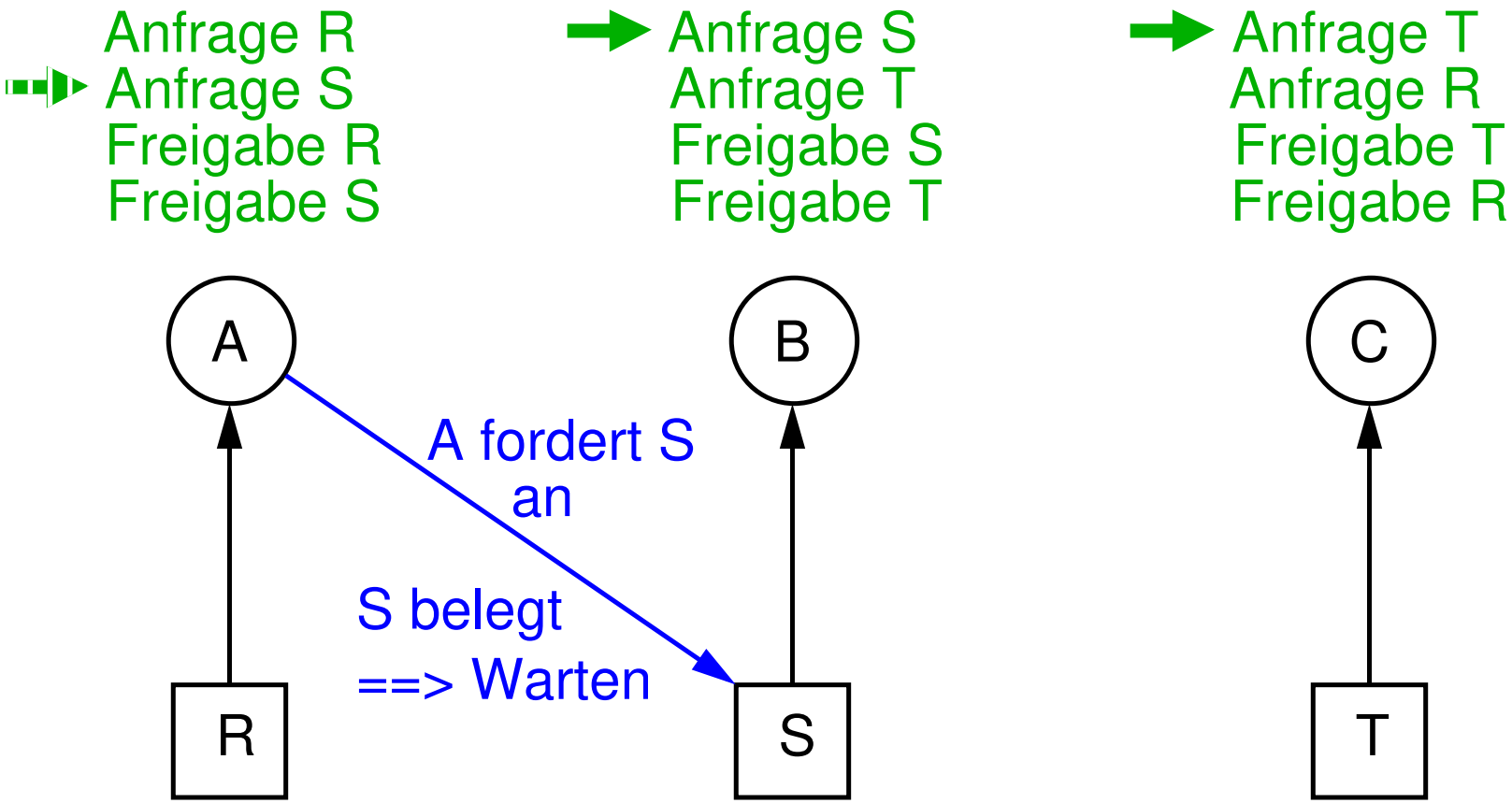
Reihenfolge: A–B–C–A–B–C...

Beispiel: Ablauf mit Verklemmung



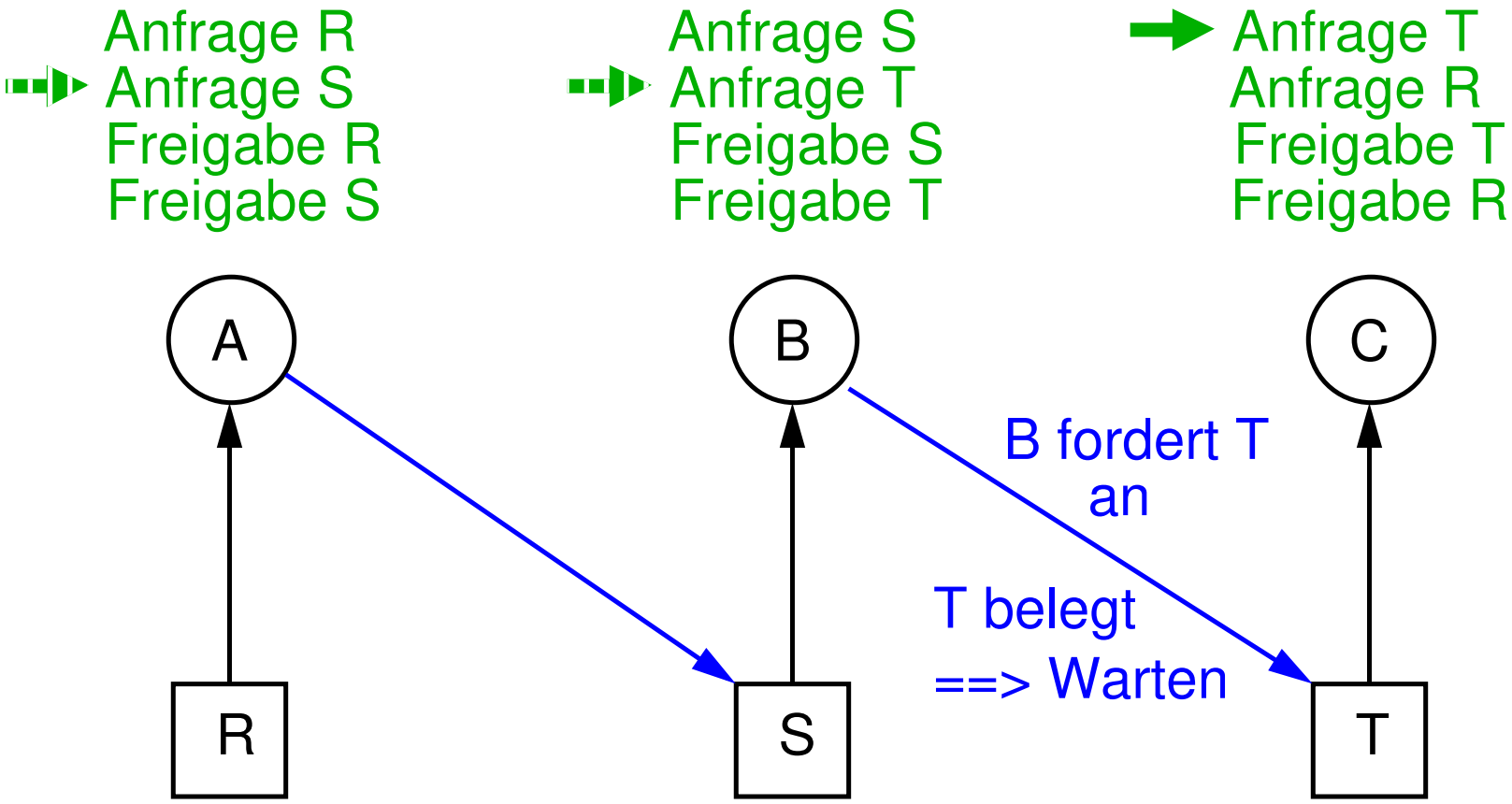
Reihenfolge: A–B–C–A–B–C...

Beispiel: Ablauf mit Verklemmung



Reihenfolge: A–B–C–A–B–C...

Beispiel: Ablauf mit Verklemmung



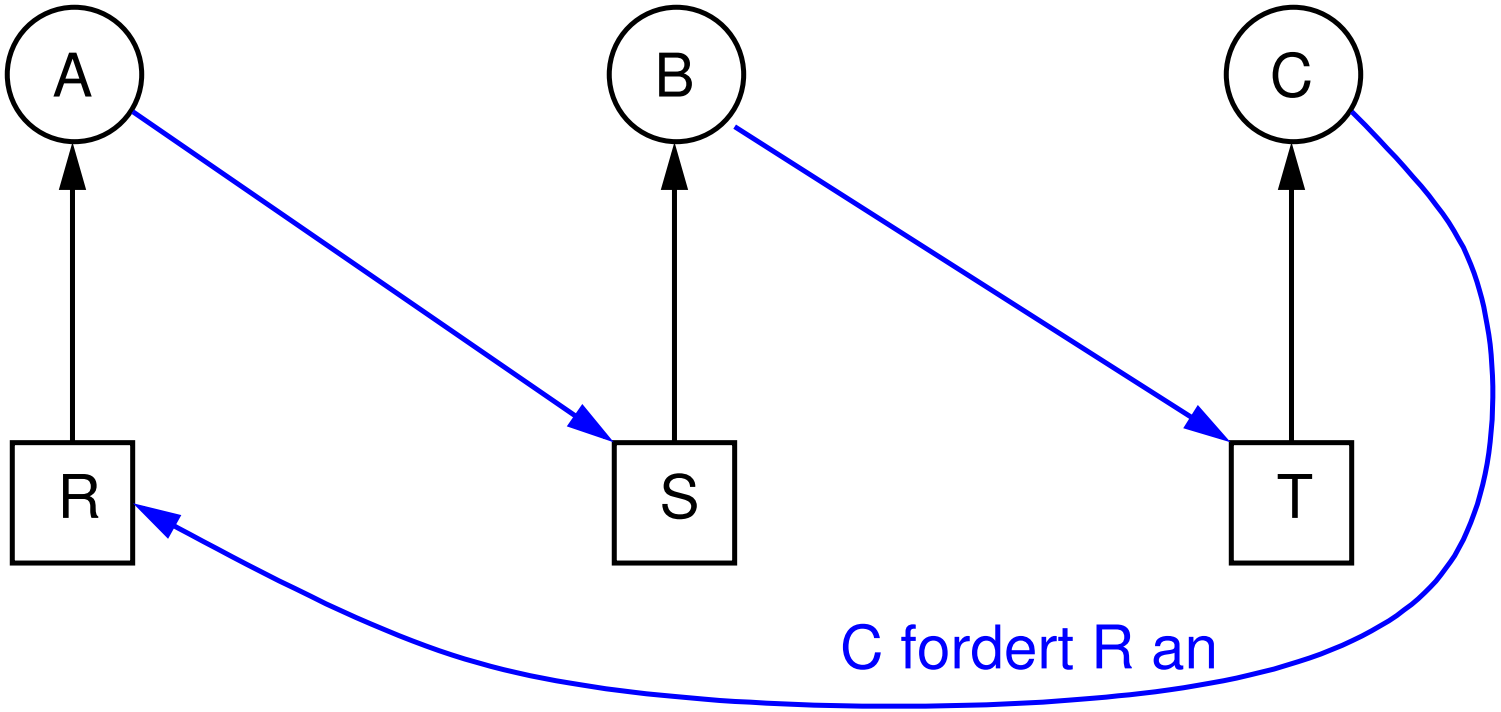
Reihenfolge: A–B–C–A–B–C...

Beispiel: Ablauf mit Verklemmung

Anfrage R
Anfrage S
Freigabe R
Freigabe S

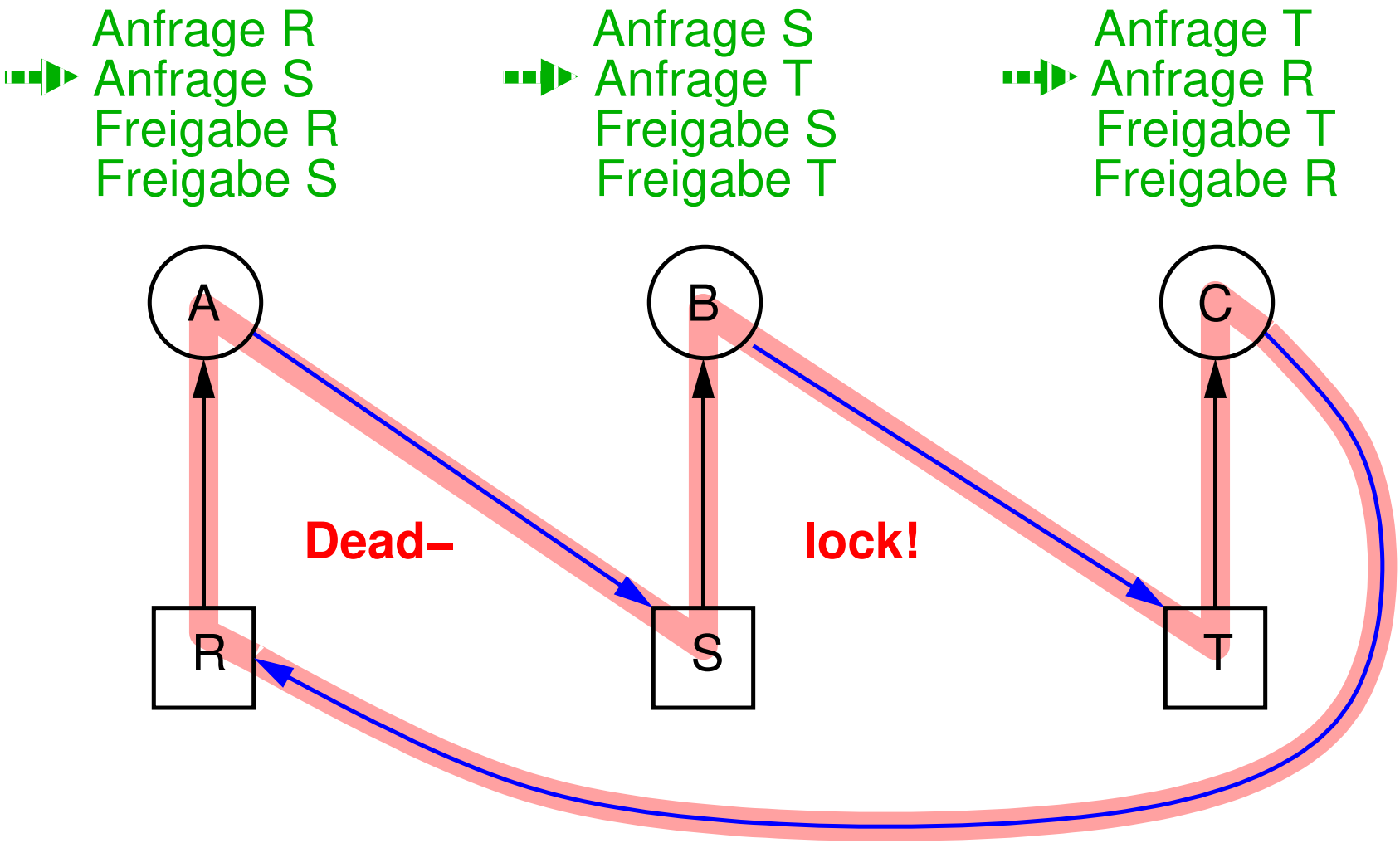
Anfrage S
Anfrage T
Freigabe S
Freigabe T

Anfrage T
Anfrage R
Freigabe T
Freigabe R



Reihenfolge: A–B–C–A–B–C...

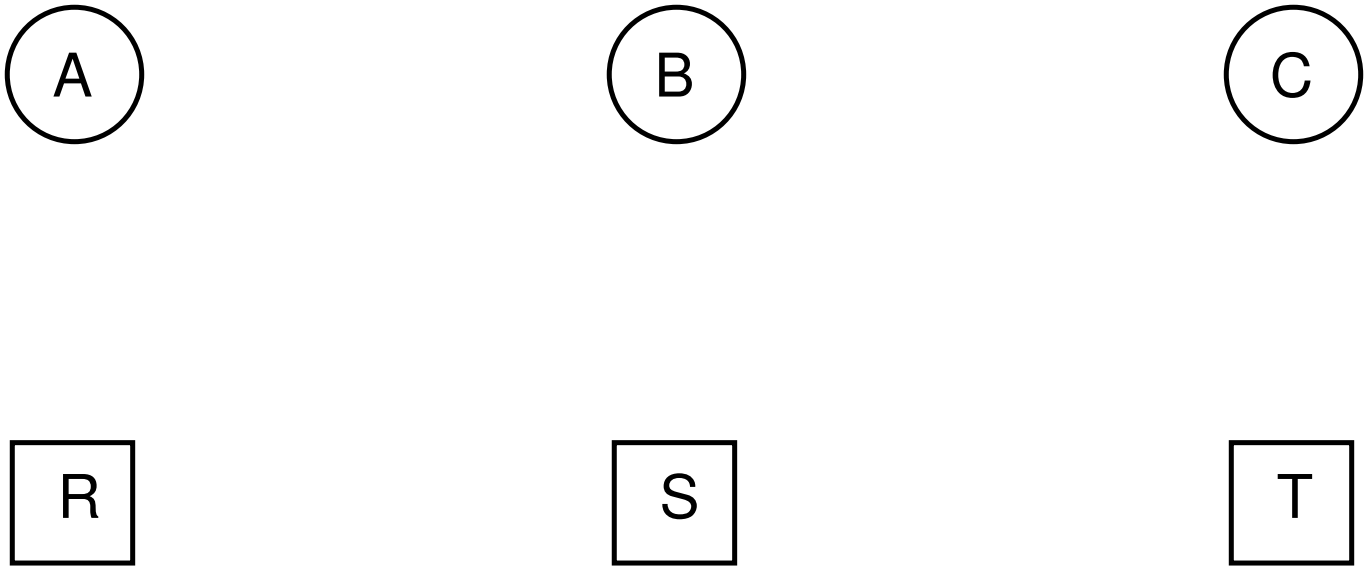
Beispiel: Ablauf mit Verklemmung



Reihenfolge: A–B–C–A–B–C...

Beispiel: Ablauf ohne Verklemmung

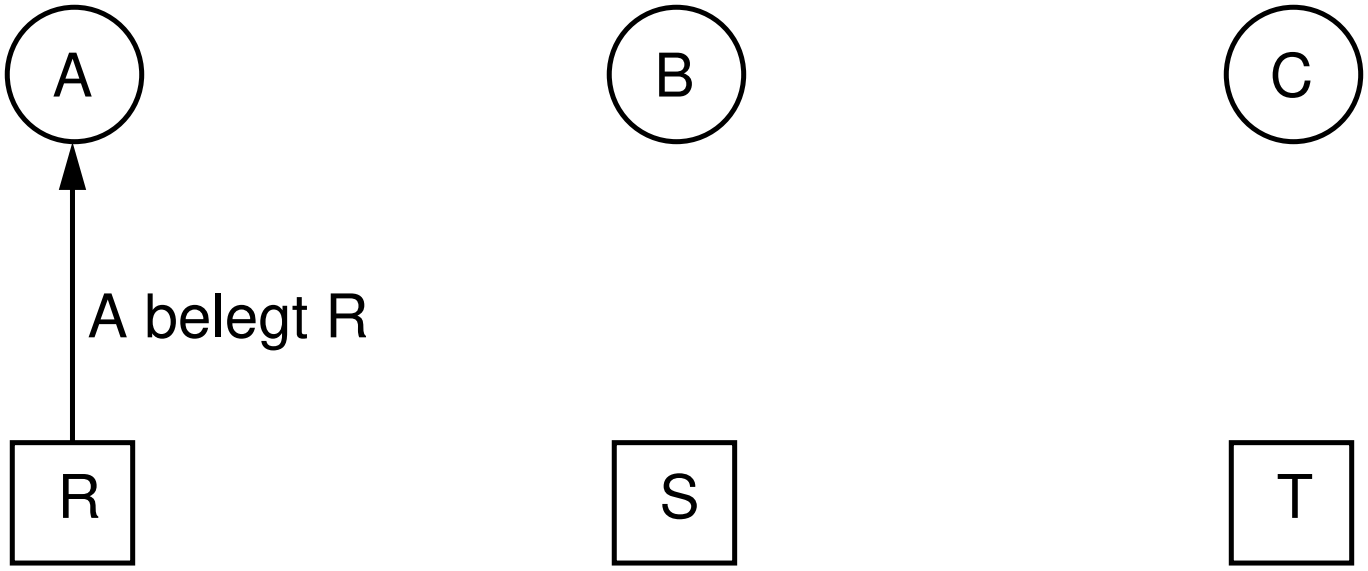
A	B	C
Anfrage R	Anfrage S	Anfrage T
Anfrage S	Anfrage T	Anfrage R
Freigabe R	Freigabe S	Freigabe T
Freigabe S	Freigabe T	Freigabe R



Reihenfolge: A–B–C–A–B–C...

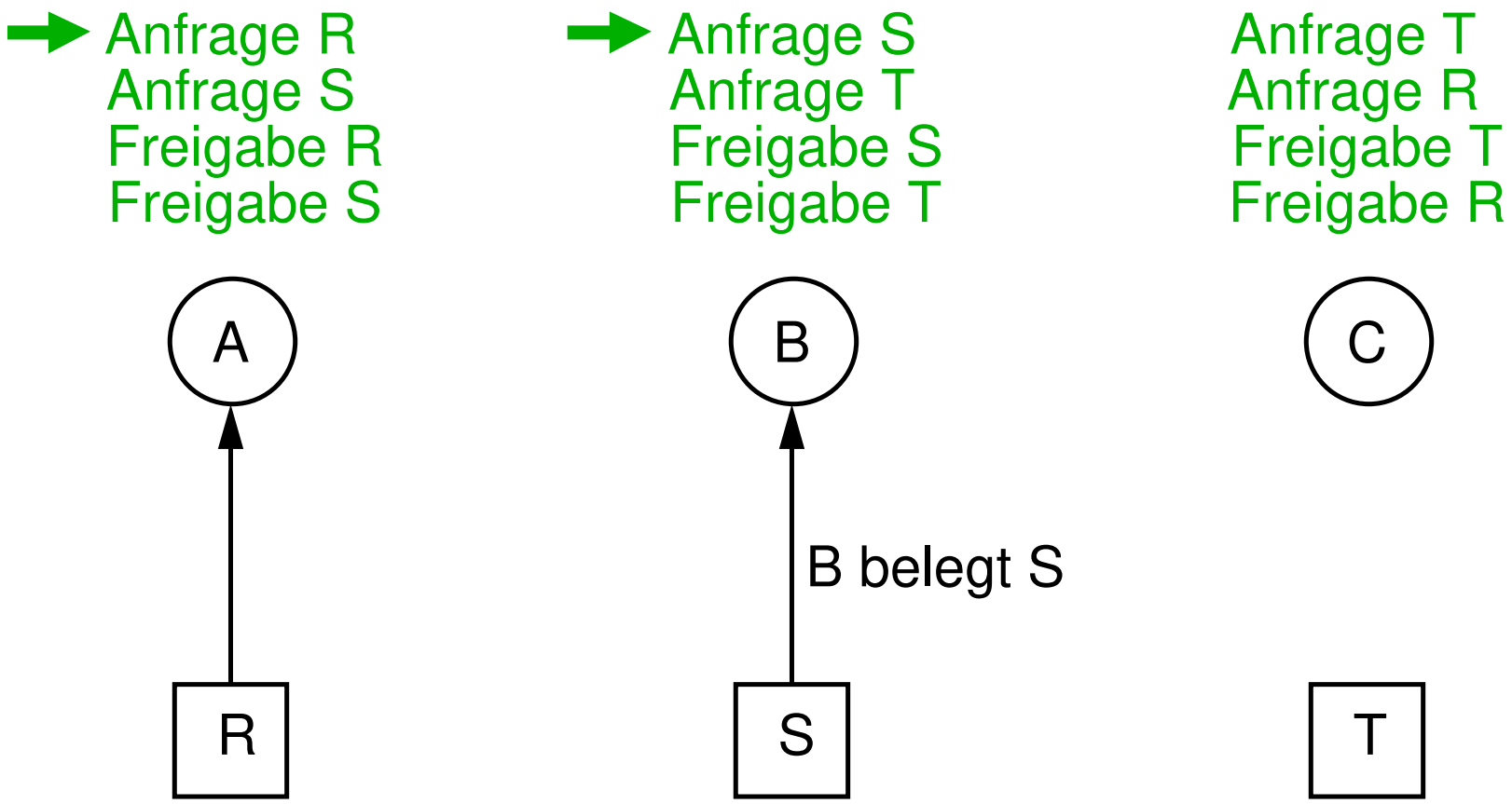
Beispiel: Ablauf ohne Verklemmung

A	B	C
Anfrage R	Anfrage S	Anfrage T
Anfrage S	Anfrage T	Anfrage R
Freigabe R	Freigabe S	Freigabe T
Freigabe S	Freigabe T	Freigabe R



Reihenfolge: A–B–C–A–B–C...

Beispiel: Ablauf ohne Verklemmung



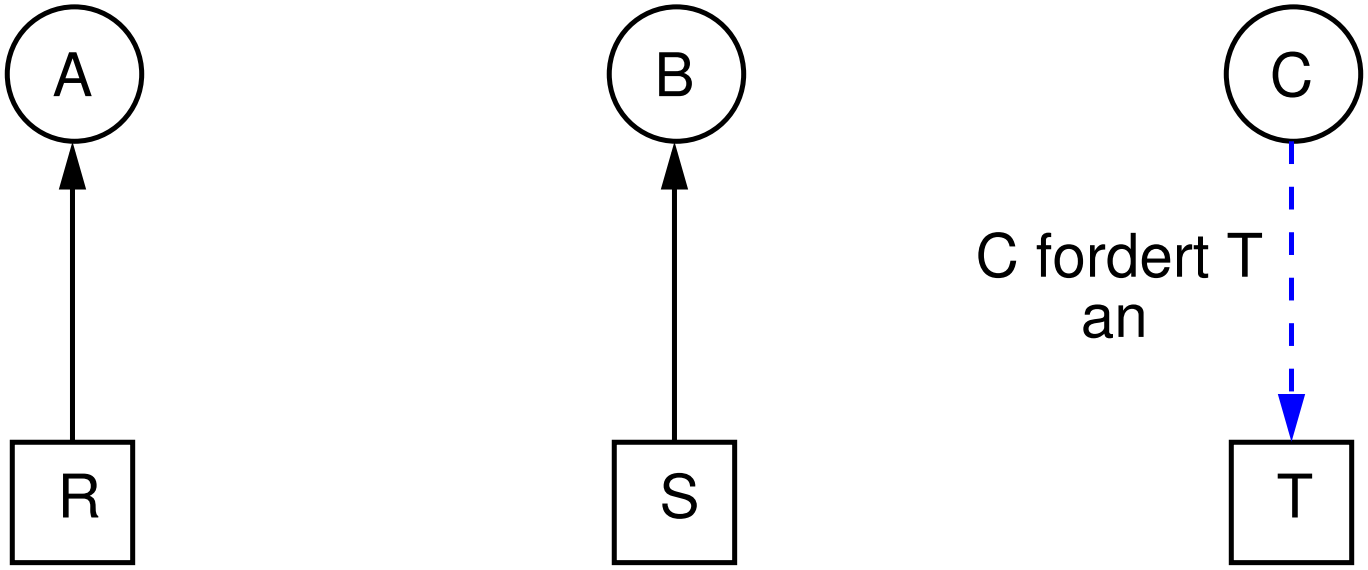
Reihenfolge: A–B–C–A–B–C...

Beispiel: Ablauf ohne Verklemmung

Anfrage R
Anfrage S
Freigabe R
Freigabe S

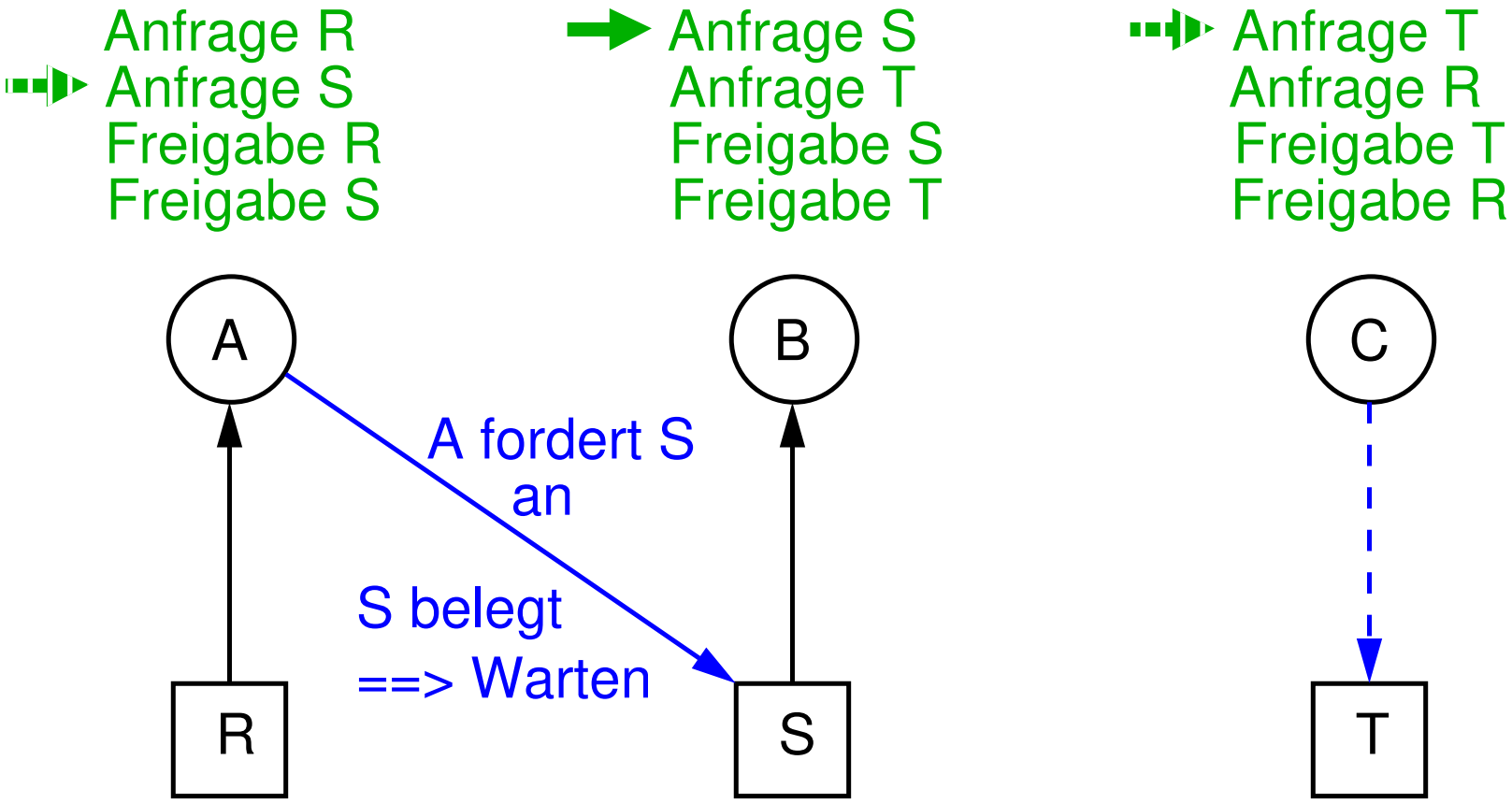
Anfrage S
Anfrage T
Freigabe S
Freigabe T

Anfrage T
Anfrage R
Freigabe T
Freigabe R

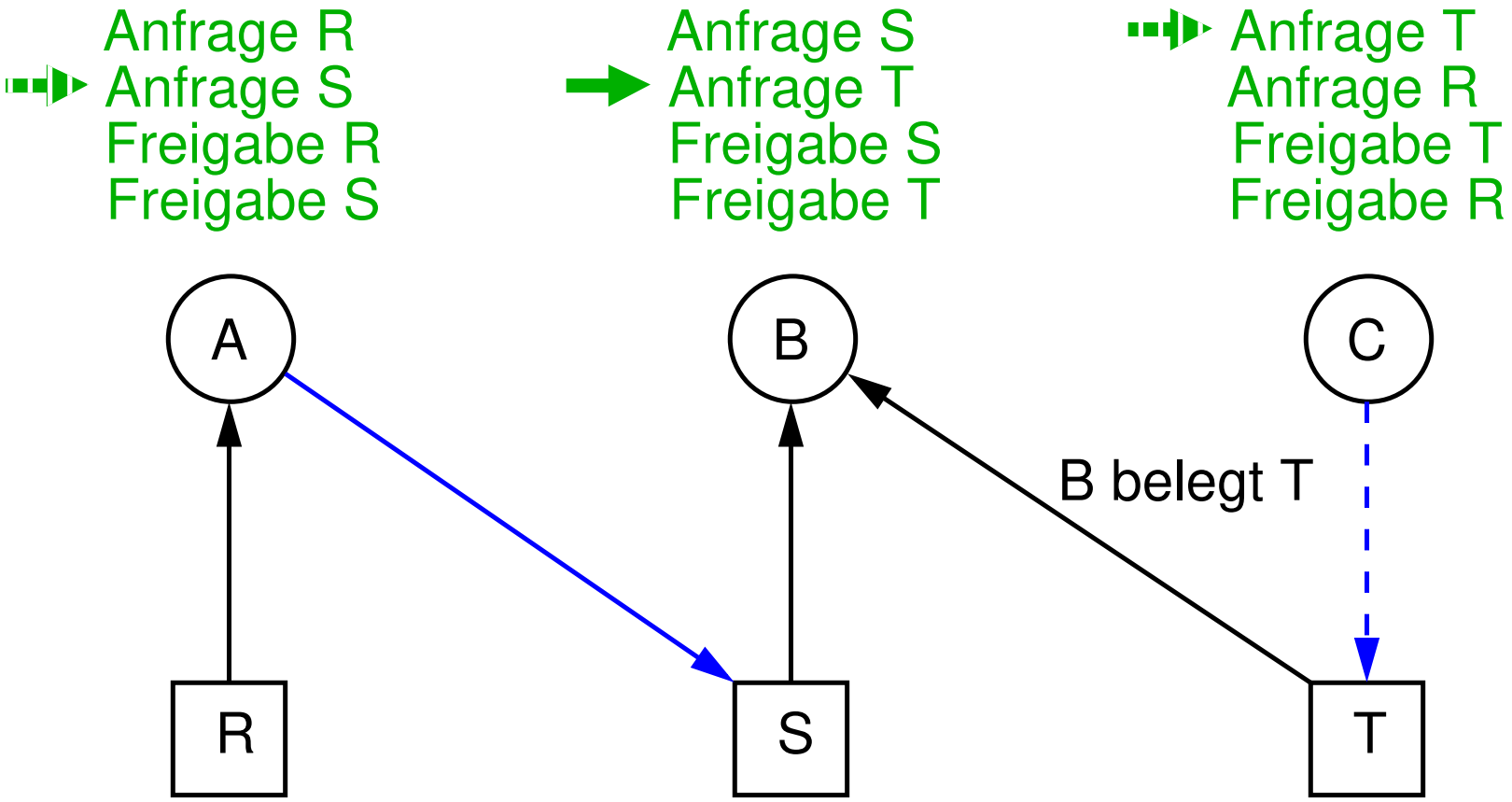


Betriebssystem erkennt, daß Belegung von T durch C zu Deadlock führen könnte ==> C wird blockiert

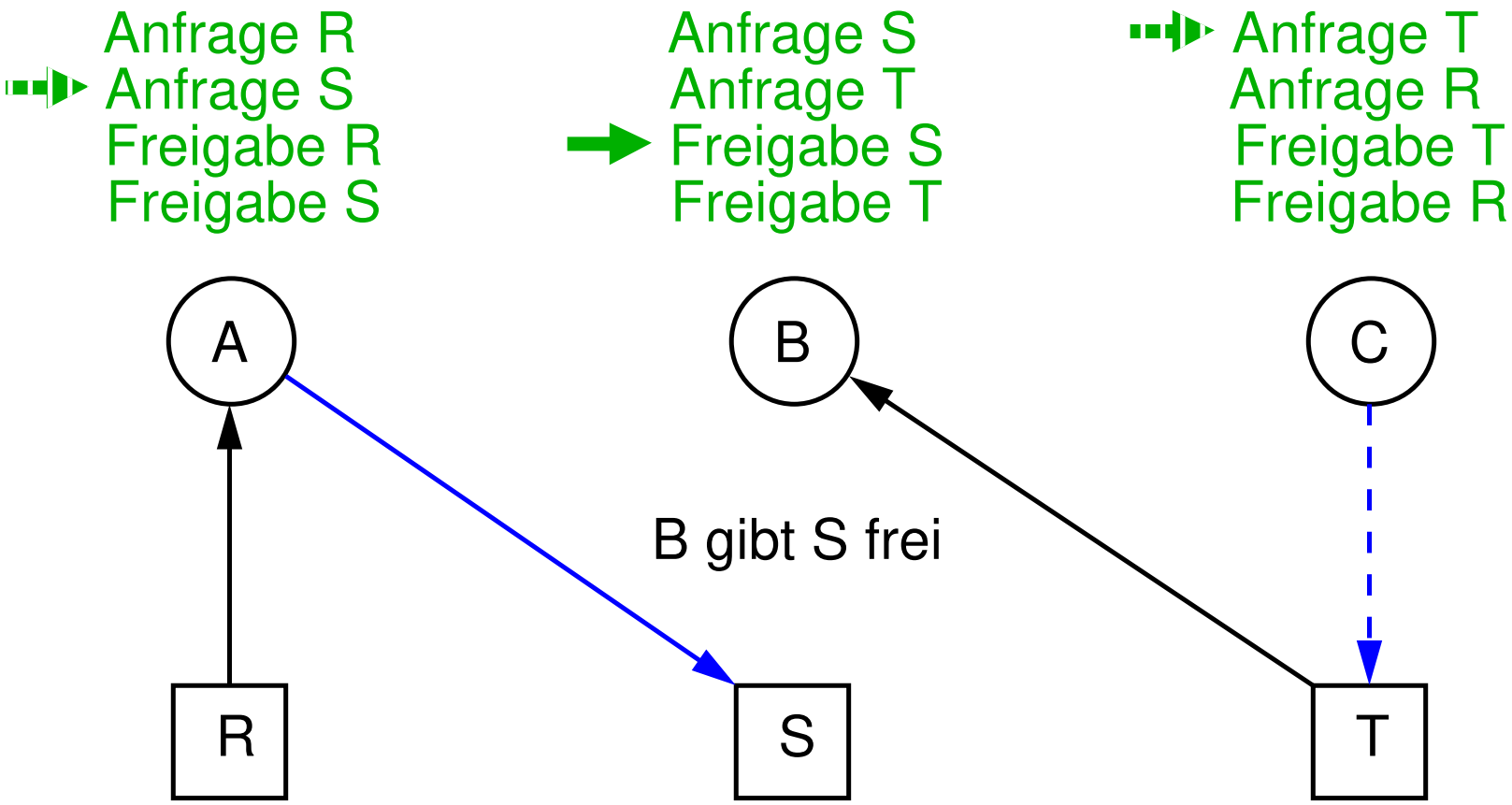
Beispiel: Ablauf ohne Verklemmung



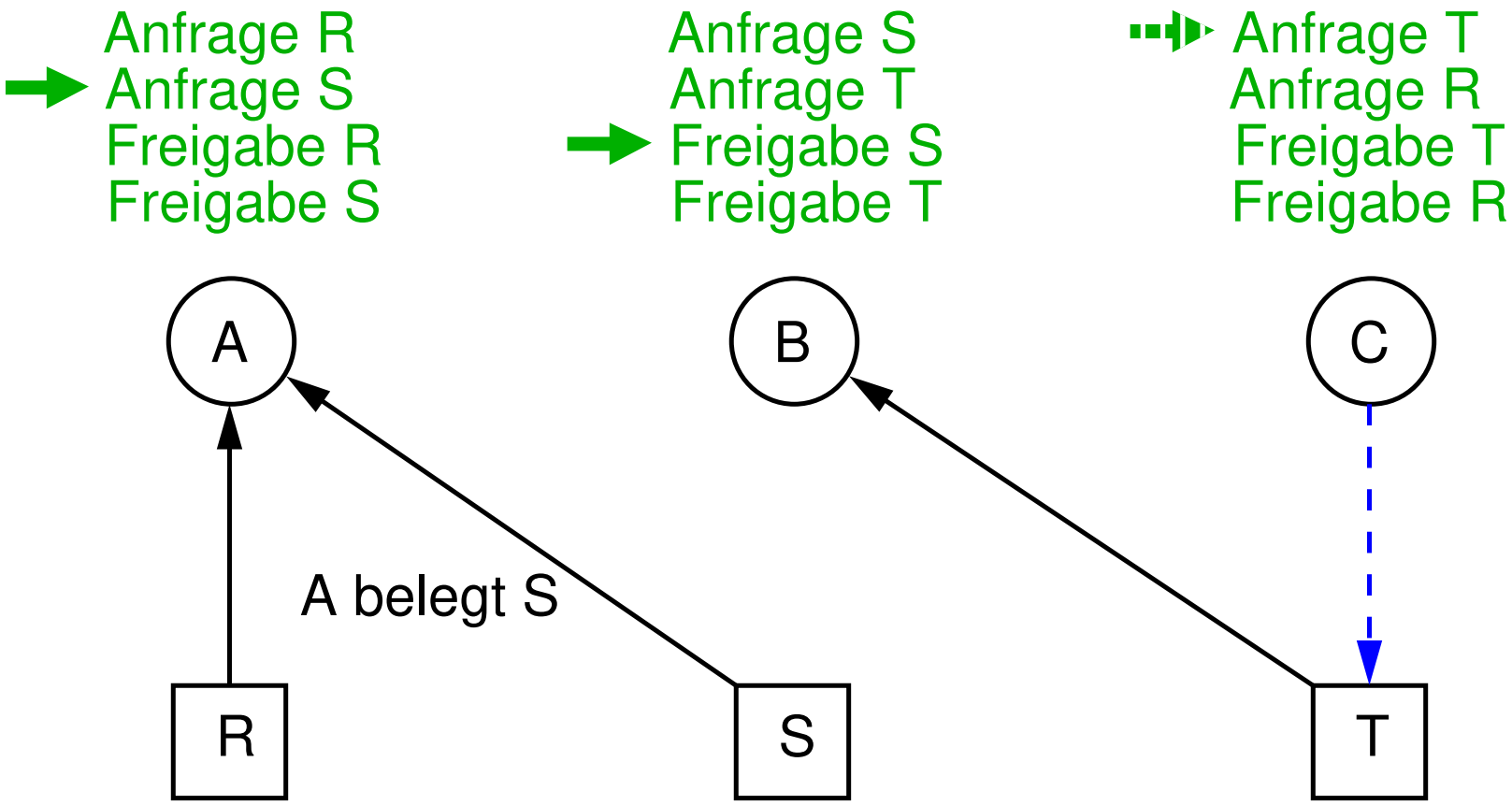
Beispiel: Ablauf ohne Verklemmung



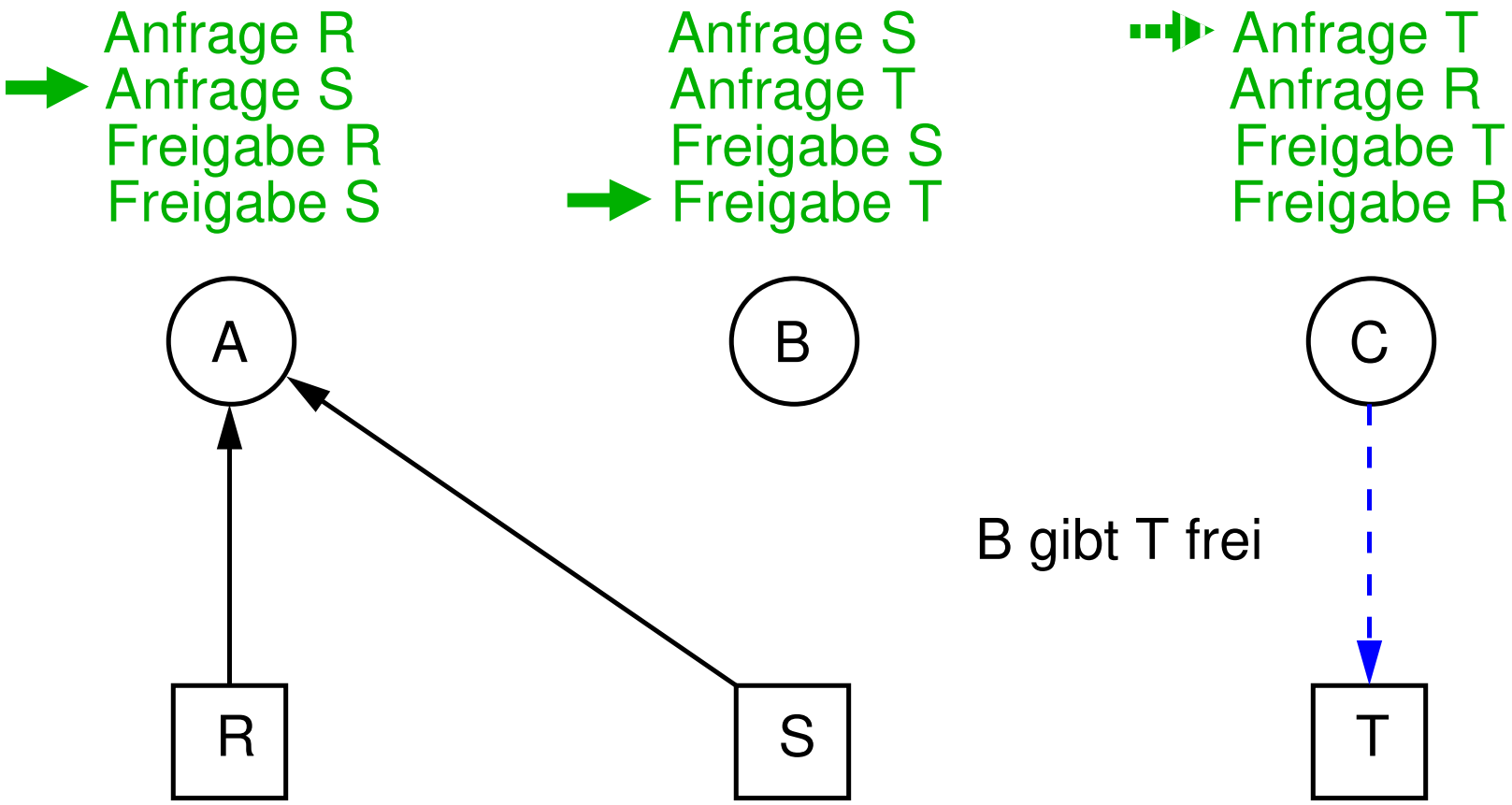
Beispiel: Ablauf ohne Verklemmung



Beispiel: Ablauf ohne Verklemmung



Beispiel: Ablauf ohne Verklemmung



Jetzt kann C weiterlaufen und T belegen

5.3 Behandlung von Deadlocks

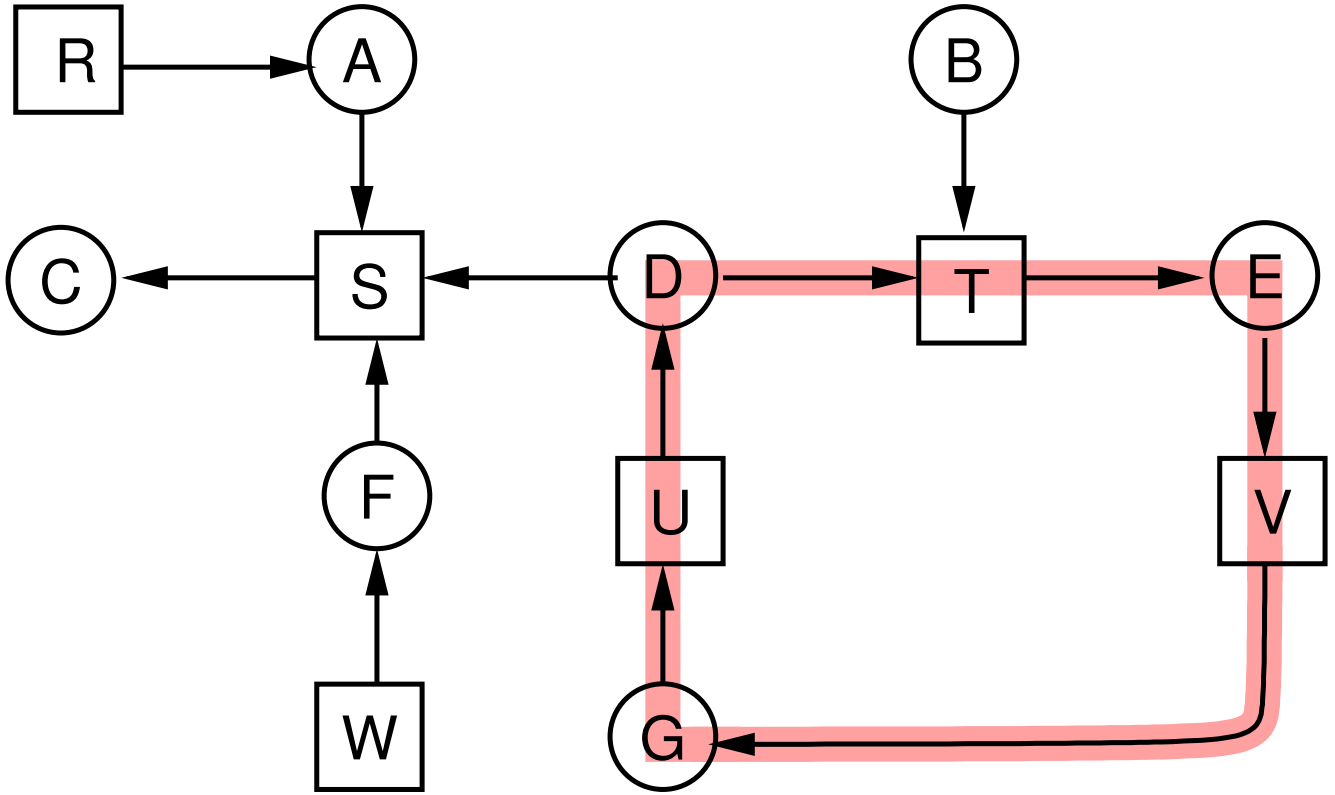
Prinzipiell vier Möglichkeiten

- Vogel-Strauß-Algorithmus
  - nichts tun und hoffen, daß alles gut geht
- **Deadlock-Erkennung und -Behebung**
  - lässt zunächst alle Anforderungen zu, löst ggf. Deadlock auf
- ***Deadlock-Avoidance*** (Deadlock-Vermeidung)
  - BS lässt Resourcenanforderung nicht zu, wenn dadurch ein Deadlock entstehen könnte
- ***Deadlock-Prevention*** (Deadlock-Verhinderung)
  - macht eine der 4 Deadlock-Bedingungen unerfüllbar

Achtung: Begriffsverwirrung bei Verhinderung vs. Vermeidung!

Deadlock-Erkennung bei einer Ressource pro Typ

- Deadlock ⇔ Belegungs-Anforderungs-Graph enthält Zyklus



- Benötigt Algorithmus, der Zyklen in Graphen findet

Deadlock-Erkennung bei mehreren Ressourcen pro Typ

- Belegungs-Anforderungs-Graph nicht mehr adäquat
  - Prozeß wartet nicht auf **bestimmte** Ressource, sondern auf irgendeine Ressource des passenden Typs
- Im Folgenden: **Matrix-basierter Algorithmus**
- Das System sei wie folgt modelliert:
  - n Prozesse $P_1, ..., P_n$
  - m Klassen von Ressourcen
  - Klasse i $(1 \leq i \leq m)$ enthält E_i Ressource-Instanzen

Datenstrukturen des Matrix-Algorithmus

- **Ressourcenvektor E**
  - Anzahl verfügbarer Ressourcen für jede Klasse
- **Ressourcenrestvektor A**
  - gibt für jede Klasse an, wieviele Ressourcen noch frei sind
- **Belegungsmatrix C**
  - $C_{ij} =$ Anzahl der Ressourcen der Klasse j, die Prozeß P_i belegt
  - Invariante: $\forall j = 1...m : \sum_{i=1}^{n} C_{ij} + A_j = E_j$
- **Anforderungsmatrix R**
  - $R_{ij} =$ Anzahl der Ressourcen der Klasse j, die Prozeß P_i im Moment anfordert bzw. auf deren Zuteilung er wartet

Beispiel:

Bandgeräte, Plotter, Scanner, CD–ROM

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

Ressourcenvektor

Bandgeräte, Plotter, Scanner, CD–ROM

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

Ressourcenrestvektor

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

Belegungsmatrix

Zeile 2: Belegung durch Prozeß 2

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

Anforderungsmatrix

Zeile 2: Forderungen von Prozeß 2

Matrix-Algorithmus

1. Suche unmarkierten Prozeß P_i, so daß i-te Zeile von R kleiner oder gleich A ist, d.h. $R_{ij} \leq A_j$ für alle $j = 1...m$
2. Falls ein solcher Prozeß existiert:
   addiere i-te Zeile von C zu A, markiere Prozeß P_i, gehe zu 1
3. Andernfalls: Ende

➥ Alle am Ende nicht markierten Prozesse sind an einem Deadlock beteiligt

➥ Schritt 1 sucht Prozeß, der zu Ende laufen könnte

➥ Schritt 2 simuliert Freigabe der Ressourcen am Prozeßende

Beispiel:

Bandgeräte Plotter Scanner CD-ROM

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

Ressourcenvektor

Bandgeräte Plotter Scanner CD-ROM

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

Ressourcenrestvektor

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

Belegungsmatrix

Zeile 2: Belegung durch Prozeß 2

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

Anforderungsmatrix

Zeile 2: Forderungen von Prozeß 2

Beispiel:

Bandgeräte Plotter Scanner CD-ROM

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

Ressourcenvektor

Bandgeräte Plotter Scanner CD-ROM

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

Ressourcenrestvektor

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

Belegungsmatrix

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

Anforderungsmatrix

1. $R_{3,j} <= A_j$ für $j = 1 \ldots 4$

   Anforderungen von Prozeß 3 erfüllbar

Beispiel:

	Bandgeräte	Plotter	Scanner	CD-ROM
E =	4	2	3	1

Ressourcenvektor

	Bandgeräte	Plotter	Scanner	CD-ROM
A =	2	2	2	0

Ressourcenrestvektor

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

Belegungsmatrix

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

Anforderungsmatrix

2. $A_j = A_j + C_{3,j}$; markiere Prozeß 3

Prozeß 3 kann zu Ende laufen und Ressourcen freigeben

Beispiel:

Bandgeräte Plotter Scanner CD-ROM

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

Bandgeräte Plotter Scanner CD-ROM

$A = (\ 2 \quad 2 \quad 2 \quad 0\ )$ **Ressourcenrestvektor**

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

$$R = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 0 & 3 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$ **Anforderungsmatrix**

1. $R_{2,3} > A_3$ Prozeß 2 kann nicht weiterlaufen

Beispiel:

Bandgeräte Plotter Scanner CD-ROM

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

Bandgeräte Plotter Scanner CD-ROM

$A = (2 \quad 2 \quad 2 \quad 0)$ **Ressourcenrestvektor**

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

Belegungsmatrix

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

Anforderungsmatrix

1. $R_{1,4} > A_4$ Prozeß 1 kann auch nicht weiterlaufen

Deadlock zwischen Prozeß 1 und 2!

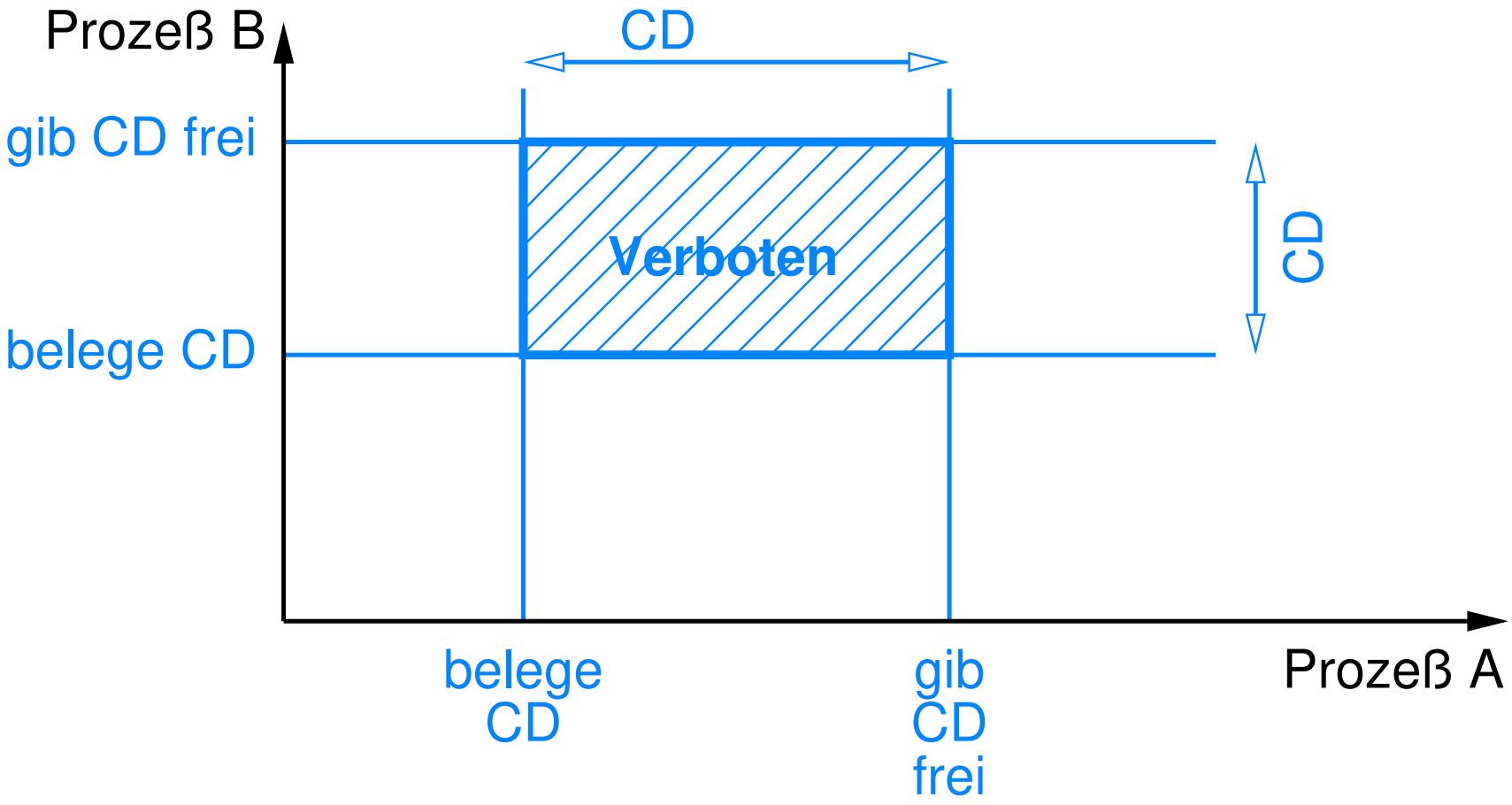
Wann wird Deadlock-Erkennung durchgeführt?

- Bei jeder Ressourcen-Anforderung:
  - Deadlocks werden schnellstmöglich erkannt
  - ggf. zu hoher Rechenaufwand
- In regelmäßigen Abständen
- Wenn Prozessorauslastung niedrig ist
  - Rechenkapazität ist verfügbar
  - niedrige Auslastung kann Hinweis auf Deadlock sein

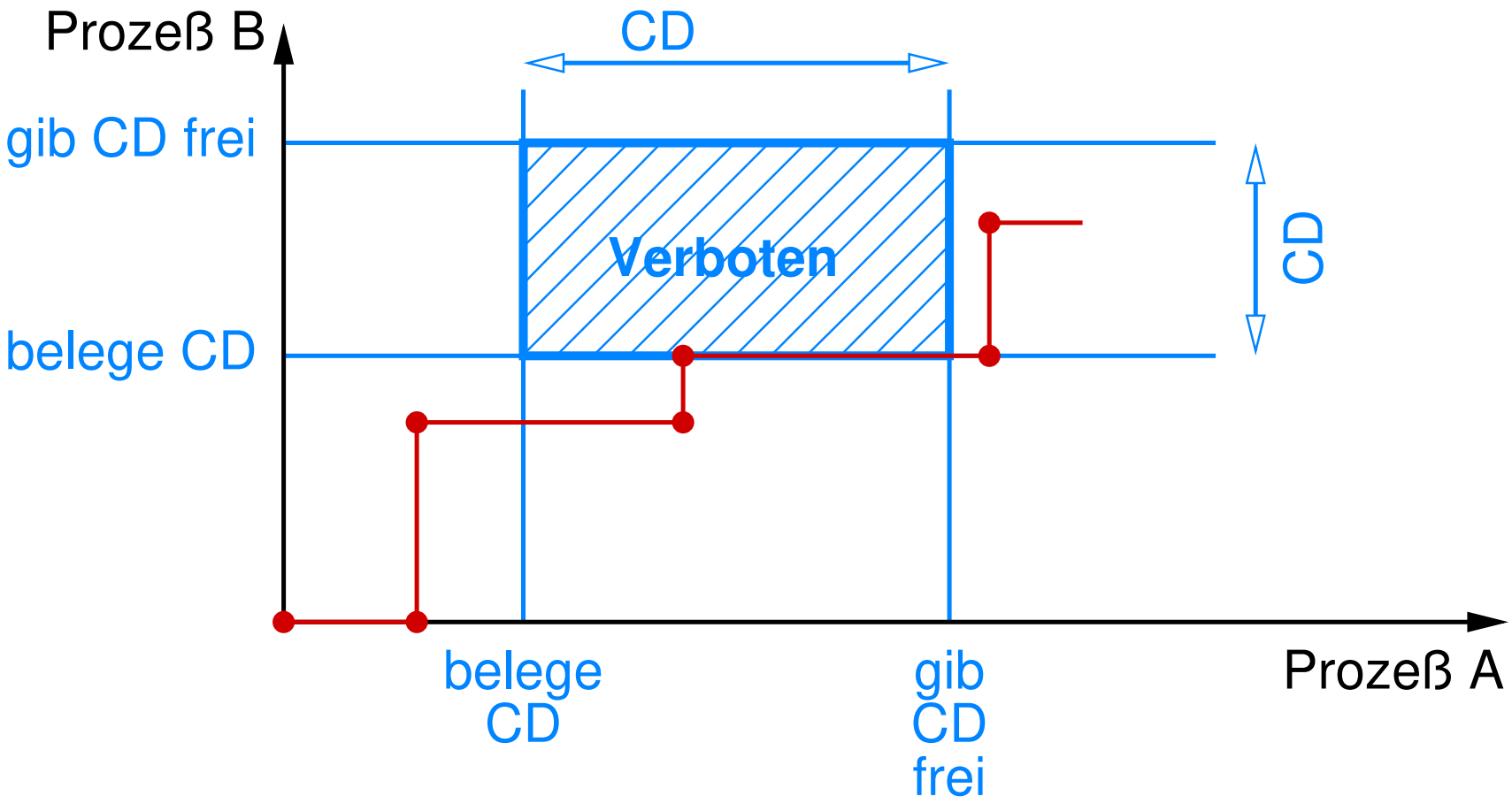
Möglichkeiten zur Behebung von Deadlocks

- Temporärer Entzug einer Ressource
  - schwierig bis unmöglich
  - (wir betrachten hier ununterbrechbare Ressourcen ...)
- Rücksetzen eines Prozesses
  - Prozeßzustand wird regelmäßig gesichert (*Checkpoint*)
  - bei Verklemmung: Rücksetzen auf Checkpoint, bei dem Prozeß Ressource noch nicht belegt hatte
  - auch schwierig
- Abbruch eines Prozesses
  - wähle Prozeß, der problemlos neu gestartet werden kann
  - z.B. in Datenbanksystemen (Transaktionen)

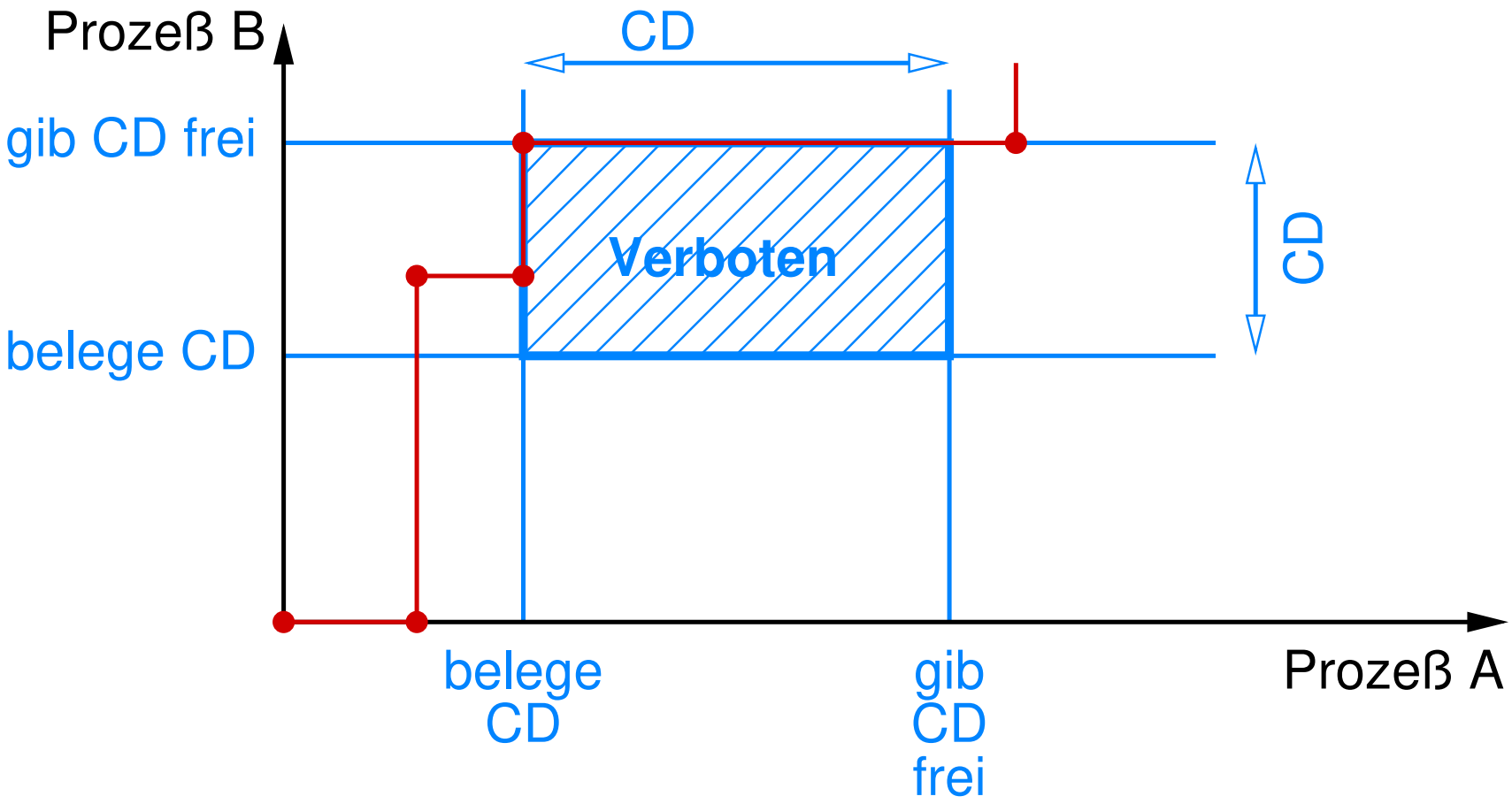
Vorbemerkung: Ressourcenspur



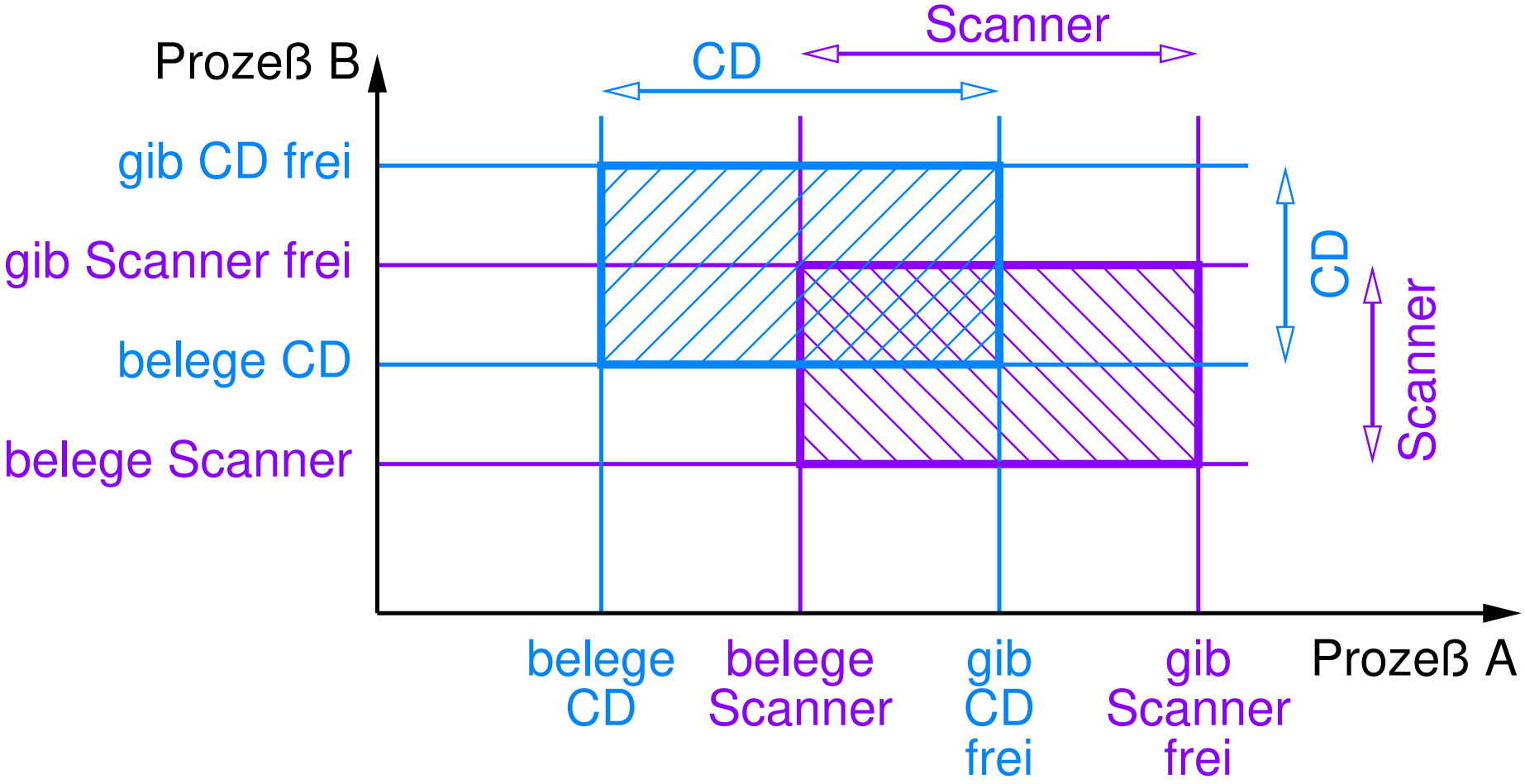
Vorbemerkung: Ressourcenspur



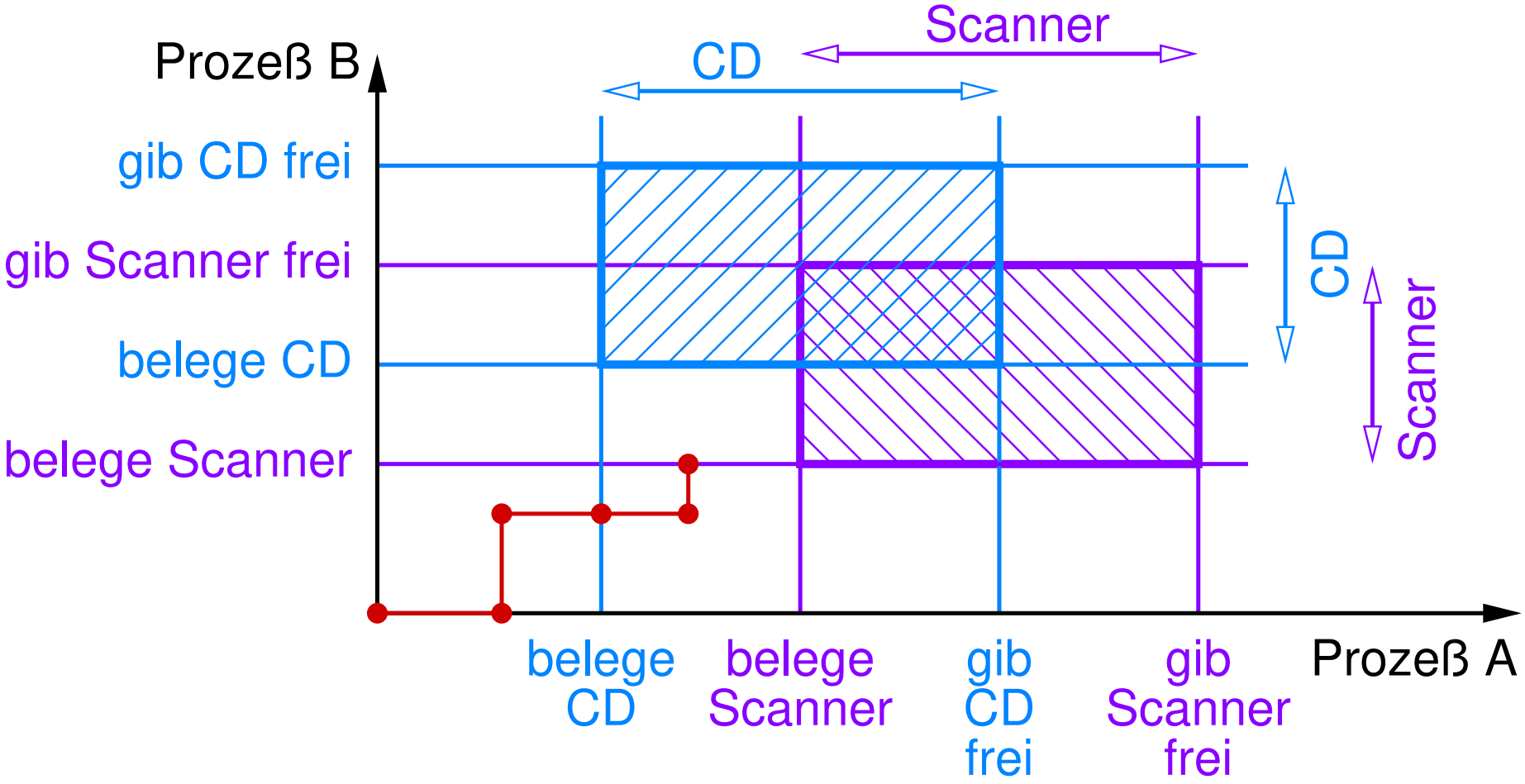
Vorbemerkung: Ressourcenspur



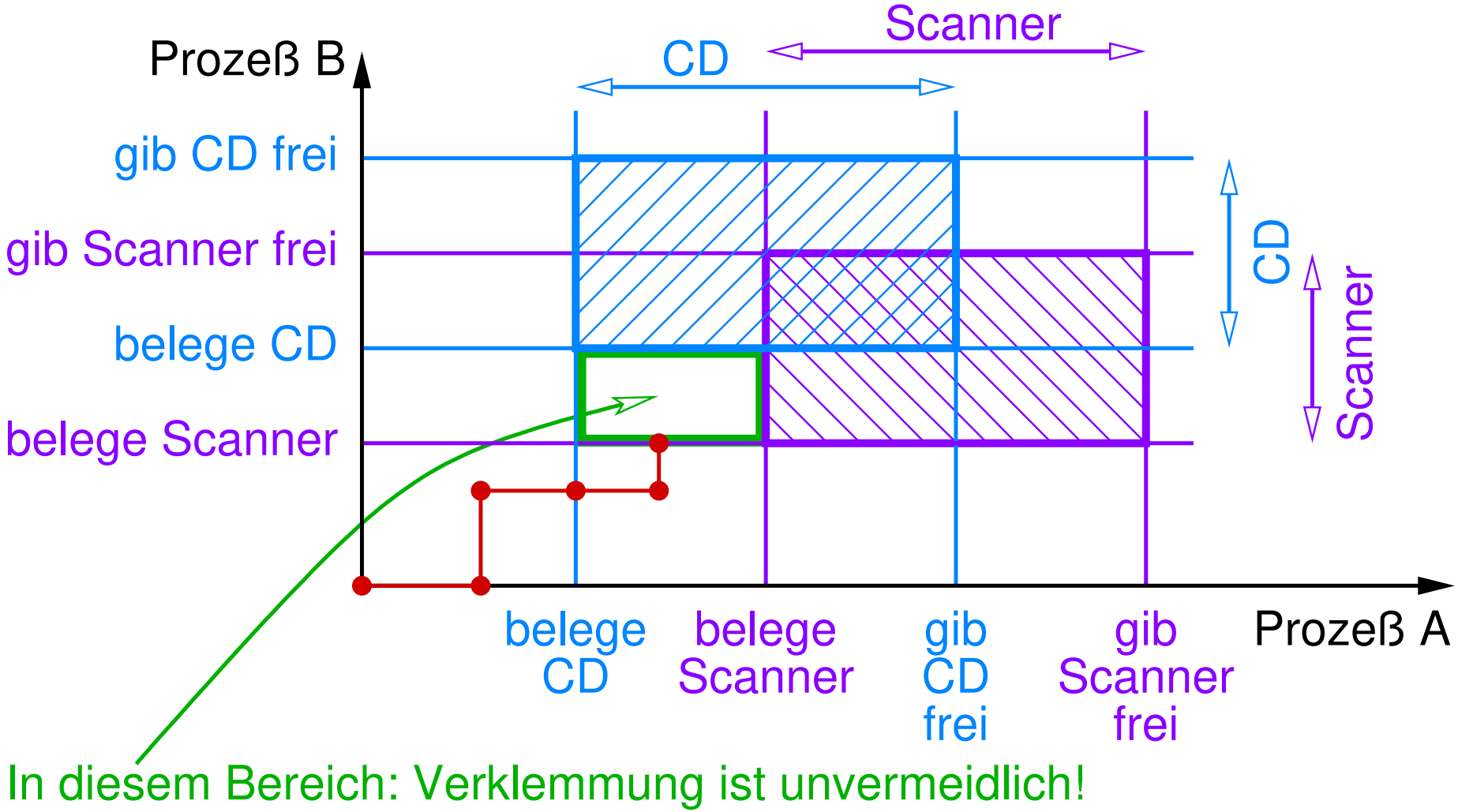
Vorbemerkung: Ressourcenspur



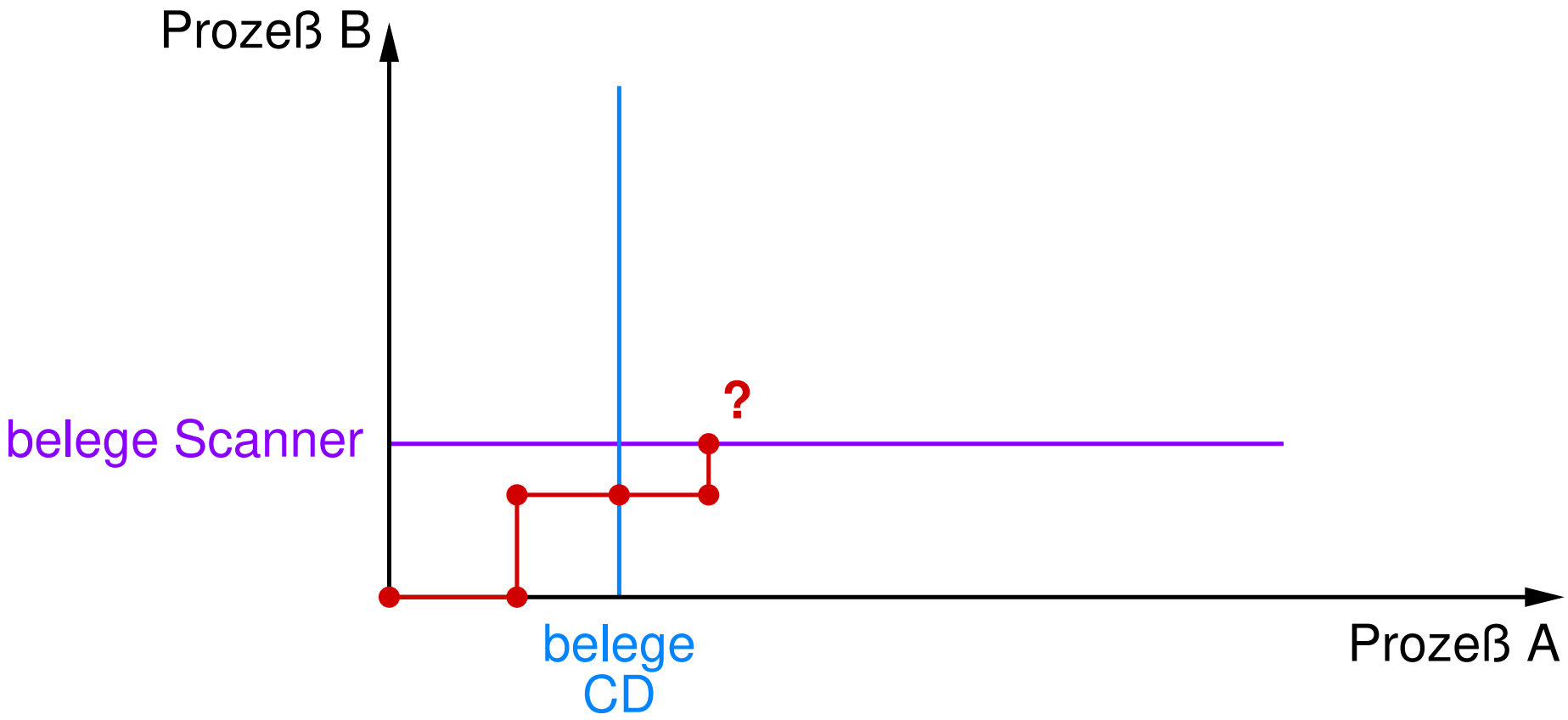
Vorbemerkung: Ressourcenspur



Vorbemerkung: Ressourcenspur

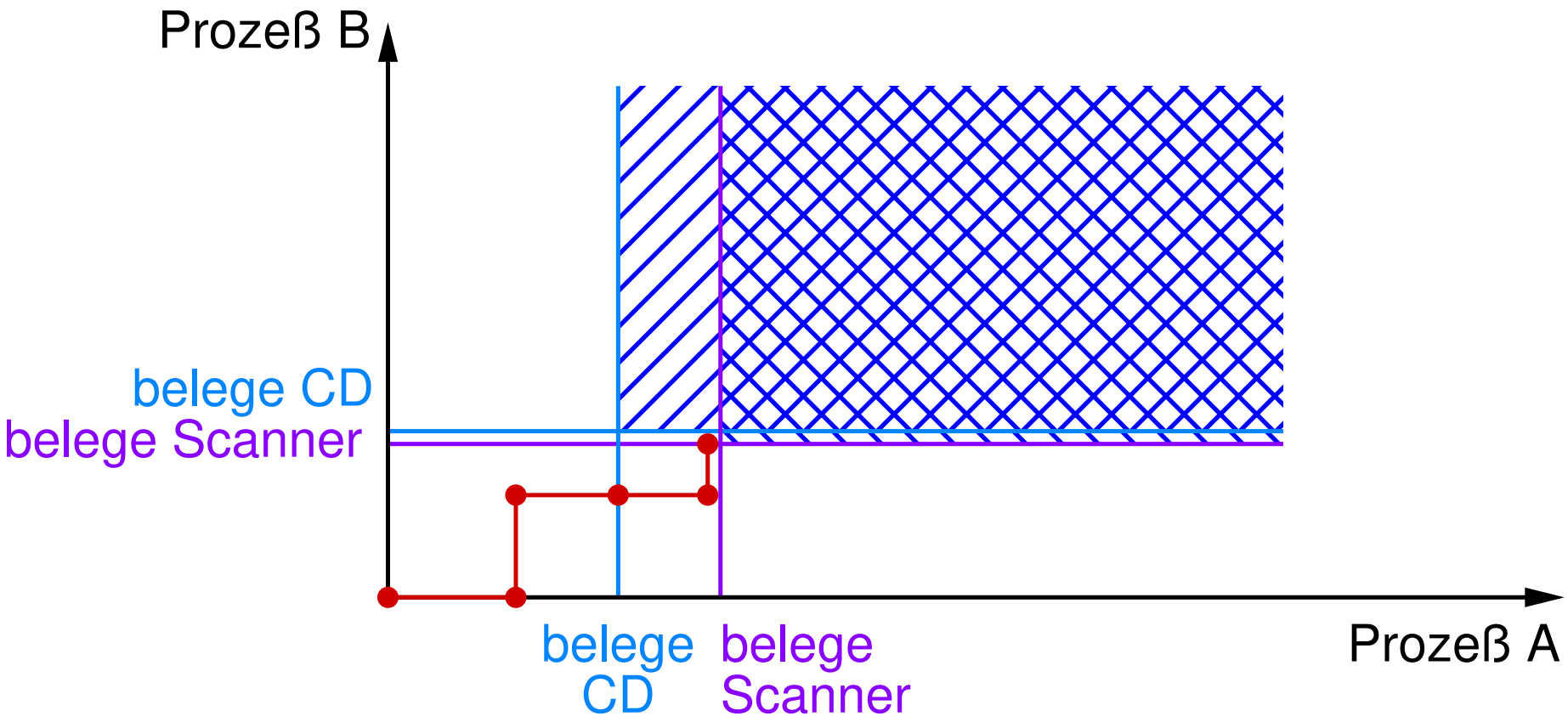


Vorbemerkung: Ressourcenspur



Problem: BS kennt die Zukunft nicht!

Vorbemerkung: Ressourcenspur



Was wäre, wenn unmittelbar nach Zuteilung des Scanners alle Pro–zesse ihre maximalen Restforderungen stellen würden? Verklemmung!

Sichere und unsichere Zustände:

- Ein Zustand heißt **sicher**,
  wenn es eine Ausführungsreihenfolge der Prozesse gibt,
  in der alle Prozesse (ohne Deadlock) zu Ende laufen,
  selbst wenn alle Prozesse sofort die maximalen Ressourcen-anforderungen stellen
- Sonst heißt der Zustand **unsicher**
- Anmerkung:
  - ein unsicherer Zustand muß nicht zwangsläufig in eine Verklemmung führen
  - es müßen ja nicht alle Prozesse (sofort) ihre Maximalforderungen stellen!

Bankiers-Algorithmus

- BS teilt Ressource nur dann zu, wenn dadurch auf keinen Fall ein Deadlock entstehen kann
  - dazu: BS prüft bei jeder Ressourcenanforderung, ob der Zustand **nach** der Anforderung noch **sicher** ist
    - falls ja, wird die Ressource zugeteilt
    - falls nein, wird der anfordernde Prozeß blockiert
  - sehr vorsichtige (restriktive) Ressourcenzuteilung!

- Das BS muß dazu die (maximalen) zukünftigen Forderungen aller Prozesse kennen!
  - in der Praxis i.a. nicht erfüllbar
  - Verfahren nur für spezielle Anwendungsfälle tauglich

Bestimmung, ob ein Zustand sicher ist

- Durch angepaßten Matrix-Algorithmus zur Deadlock-Erkennung (☞ **5.3.1**, Folie 284)
- Der aktuelle Zustand des Systems ist durch E, A, und C beschrieben
  - vorhandene, freie und belegte Ressourcen
- R beschreibt nun die **maximalen zukünftigen Forderungen**
- Der Zustand ist genau dann sicher, wenn der Algorithmus für diese Werte von E, A, C und R keine Verklemmung erkennt
  - d.h., auch wenn im aktuellen Zustand (E, A, C) alle Prozesse sofort ihre maximalen Forderungen (R) stellen, gibt es einen Ablauf, der alle Prozesse zu Ende führt

5.3.3 Deadlock-Prevention

Ziel: Vorbeugendes Verhindern von Deadlocks

- Eine der vier Deadlock-Bedingungen wird unerfüllbar gemacht
  - damit sind Deadlocks unmöglich
  - häufigste Lösung in der Praxis!

Wechselseitiger Ausschluß

- Ressourcen nur dann direkt an einzelne Prozesse zuteilen, wenn dies unvermeidlich ist
- Beispiel: Zugriff auf Drucker nur über Drucker-Spooler
  - es gibt keine Konkurrenz mehr um den Drucker
  - aber: evtl. Deadlocks an anderer Stelle möglich (Konkurrenz um Platz im Spooler-Puffer)

Hold-and-Wait-Bedingung

- Forderung: jeder Prozeß muß alle seine Ressourcen bereits beim Start anfordern
  - Nachteil: i.d.R. sind Forderungen nicht bekannt
  - Weiterer Nachteil: mögliche Ressourcen-Verschwendung
  - Anwendung bei Transaktionen in Datenbank-Systemen: *Two Phase Locking* (Sperrphase, Zugriffsphase)
- Alternative: vor Anforderung einer Ressource alle belegten Ressourcen kurzzeitig freigeben
  - i.a. auch nicht praktikabel (wechselseitiger Ausschluß!)

Ununterbrechbarkeit (kein Ressourcenentzug)

- i.a. unpraktikabel, siehe Deadlock-Behebung

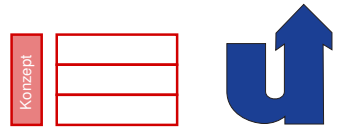
Zyklisches Warten

- Durchnumerieren aller Ressourcen
- Anforderungen von Ressourcen zu beliebigem Zeitpunkt, aber nur in aufsteigender Reihenfolge!
  - d.h. Prozeß darf nur Ressource mit größerer Nummer als bereits belegte Ressourcen anfordern
- Deadlocks werden damit unmöglich:
  - zu jedem Zeitpunkt: betrachte reservierte Ressource mit der höchsten Nummer
  - Prozeß, der diese Ressource belegt, kann zu Ende laufen
    - er wird nur Ressourcen mit höherer Nummer anfordern
  - Induktion: alle Prozesse können zu Ende laufen

Zyklisches Warten ...

- In der Praxis erfolgversprechendster Ansatz zur Verhinderung von Deadlocks
- Probleme:
  - Festlegen einer für alle Prozesse tauglichen Ordnung nicht immer praktikabel
  - kann zu Resourcenverschwendung führen
    - Ressourcen müssen evtl. bereits reserviert werden, bevor sicher ist, daß sie benötigt werden

5.4 Verhungerung



- Situation: ein Prozeß P bekommt eine Ressource nie zugeteilt, obwohl kein Deadlock vorliegt
  - wegen ständiger Ressourcenanforderungen anderer Prozesse, die vor der von P erfüllt werden
- Beispiel: Lösung des Leser/Schreiber-Problems aus **3.8**
- Vermeidung des Verhungerns: z.B. durch **FIFO-Strategie**
  - FIFO: *First In First Out*
  - teile Ressource an den Prozeß zu, der am längsten wartet
- Anmerkung: manchmal ist die Möglichkeit des Verhungerns „gewollt“
  - wenn bestimmten Prozessen Priorität gegeben werden soll

5.5 Zusammenfassung / Wiederholung

- Deadlock: eine Menge von Prozessen wartet auf Ereignisse, die nur ein anderer Prozeß der Menge auslösen kann
- Bedingungen für Ressourcen-Deadlocks:
  - Wechselseitiger Ausschluß
  - *Hold-and-Wait* (Besitzen und Warten)
  - Ununterbrechbarkeit (kein Ressourcenentzug)
  - Zyklisches Warten
- Behandlung von Deadlocks:
  - Erkennung und Behebung
    - Erkennung: Zyklus im Belegungs-Anforderungs-Graph, Matrix-basierter Algorithmus
    - Behandlung: meist Abbruch eines Prozesses
      - z.B. bei Datenbank-Transaktionen

5.5 Zusammenfassung / Wiederholung ...

- Behandlung von Deadlocks ...:
  - *Avoidance* (Vermeidung)
    - Ressource nur dann vergeben, wenn der entstehende Zustand **sicher** ist
    - sicherer Zustand: kein Deadlock, selbst wenn alle Prozesse sofort ihre Maximalforderungen stellen
  - *Prevention* (Verhinderung)
    - eine der vier Bedingungen unerfüllbar machen
    - z.B. *Spooling*: vermeidet wechselseitigen Ausschluß
    - oft auch: Festlegung einer Reihenfolge der Ressourcen
      - wenn alle Prozesse die Ressourcen nur in dieser Reihenfolge belegen, können keine Deadlocks entstehen

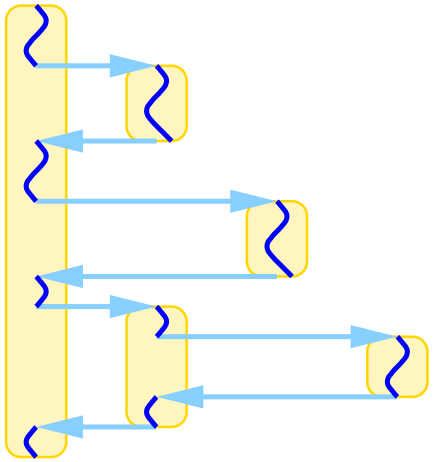
Betriebssysteme und nebenläufige Programmierung

SoSe 2025

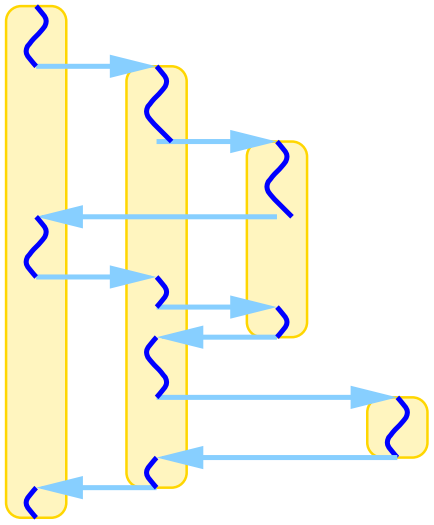
6 Koroutinen und asynchrone Programmierung

6.1 Koroutinen

- Normaler Prozeduraufruf: asymmetrisch
  - Aufrufer sichert Rückkehradresse
  - aufgerufene Prozedur läuft von Anfang bis Ende
  - Aufrufer macht nach Aufrufstelle weiter



- **Koroutinen** (Convay 1963): symmetrisch
  - kein Aufruf, sondern Wechsel in andere Koroutine
  - diese Koroutine macht ab letzter „Unterbrechungsstelle“ weiter
  - statt Rückkehr wieder Wechsel in beliebige andere Koroutine



Koroutinen vs. Threads

- Gemeinsamkeiten:
  - Ausführung wechselt zwischen Koroutinen bzw. Threads hin und her
  - Fortsetzung erfolgt immer an der letzten „Unterbrechungs-stelle“
  - während der Unterbrechung bleibt der Zustand (lokale Variable) gespeichert
- Wesentlicher Unterschied:
  - bei Koroutinen erfolgt der Wechsel kooperativ, an genau festgelegten Stellen
    - der zu speichernde Zustand kann daher minimiert werden
  - Koroutinen sind verzahnt, aber nicht wirklich nebenläufig

Realisierung von Koroutinen

- Über Fortsetzungen (*continuations*)
- Eine Fortsetzung repräsentiert den Rest der Ausführung eines unterbrochenen Kontrollflusses
  - Fortsetzungsadresse (Befehlszähler)
  - lokale Variablen (inkl. Register)
    - der Koroutine selbst, sowie ggf. von aufgerufenen normalen Prozeduren
- Jede Koroutine benötigt daher ihren eigenen Keller
- Koroutinen-Wechsel entspricht Wechsel des Kellers und „Rücksprung“
  - Register werden dabei durch den Compiler im Keller gesichert

6.2 Die Protothread-Bibliothek

- Extrem leichtgewichtige „Thread“-Implementierung im Benutzermodus für die Programmiersprache C
  - in Form von Makros
- Realisiert Koroutinen ohne Keller
  - d.h. lokale Variable werden beim Koroutinen-Wechsel nicht gesichert
  - im Bedarfsfall `static` Variablen verwenden
    - möglich, solange Koroutinen nicht mehrfach instantiiert werden
  - der Zustand einer Koroutine besteht damit nur aus der Fortsetzungsadresse (Programmzähler)
- Einsatz in ressourcesbeschränkten Systemen (z.B. eingebetteten Systemen, Sensornetzen, etc.)

Beispiel

```
PT_THREAD(thr(struct pt *pt, int no)) {
  PT_BEGIN(pt);
  printf("Thr %d: A\n", no);
  PT_YIELD(pt);
  printf("Thr %d: B\n", no);
  PT_YIELD(pt);
  printf("Thr %d: C\n", no);
  PT_END(pt);
}
int main() {
  struct pt pt1, pt2;
  PT_INIT(&pt1);
  PT_INIT(&pt2);
  while(PT_SCHEDULE(thr(&pt1, 1) & thr(&pt2, 2)));
}
```

Beispiel

```
PT_THREAD(thr(struct pt *pt, int no)) {
  PT_BEGIN(pt);
  printf("Thr %d: A\n", no);
  PT_YIELD(pt);
  printf("Thr %d: B\n", no);
  PT_YIELD(pt);
  printf("Thr %d: C\n", no);
  PT_END(pt);
}
int main() {
  struct pt pt1, pt2;
  PT_INIT(&pt1);
  PT_INIT(&pt2);
  while(PT_SCHEDULE(thr(&pt1, 1) & thr(&pt2, 2)));
}
```

Deklaration der Koroutine

Beispiel

```
PT_THREAD(thr(struct pt *pt, int no)) {
  PT_BEGIN(pt);
  printf("Thr %d: A\n", no);
  PT_YIELD(pt);
  printf("Thr %d: B\n", no);
  PT_YIELD(pt);
  printf("Thr %d: C\n", no);
  PT_END(pt);
}
int main() {
  struct pt pt1, pt2;
  PT_INIT(&pt1);
  PT_INIT(&pt2);
  while(PT_SCHEDULE(thr(&pt1, 1) & thr(&pt2, 2)));
}
```

In dieser Struktur wird der Zustand der Koroutine gespeichert

Beispiel

```
PT_THREAD(thr(struct pt *pt, int no)) {
  PT_BEGIN(pt);
  printf("Thr %d: A\n", no);
  PT_YIELD(pt);
  printf("Thr %d: B\n", no);
  PT_YIELD(pt);
  printf("Thr %d: C\n", no);
  PT_END(pt);
}
int main() {
  struct pt pt1, pt2;
  PT_INIT(&pt1);
  PT_INIT(&pt2);
  while(PT_SCHEDULE(thr(&pt1, 1) & thr(&pt2, 2)));
}
```

Koroutinen–Wechsel
(wechselt immer zum Aufrufer
der Koroutine, hier **`main`**)

Beispiel

```
PT_THREAD(thr(struct pt *pt, int no)) {
  PT_BEGIN(pt);
  printf("Thr %d: A\n", no);
  PT_YIELD(pt);
  printf("Thr %d: B\n", no);
  PT_YIELD(pt);
  printf("Thr %d: C\n", no);
  PT_END(pt);
}
int main() {
  struct pt pt1, pt2;
  PT_INIT(&pt1);
  PT_INIT(&pt2);
  while(PT_SCHEDULE(thr(&pt1, 1) & thr(&pt2, 2)));
}
```

Initialisierung von zwei Koroutinen

Beispiel

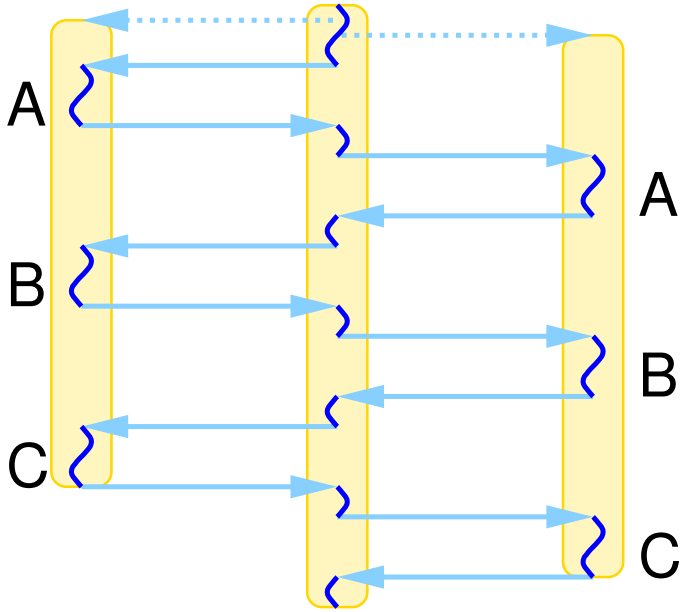
```
PT_THREAD(thr(struct pt *pt, int no)) {
  PT_BEGIN(pt);
  printf("Thr %d: A\n", no);
  PT_YIELD(pt);
  printf("Thr %d: B\n", no);
  PT_YIELD(pt);
  printf("Thr %d: C\n", no);
  PT_END(pt);
}
int main() {
  struct pt pt1, pt2;
  PT_INIT(&pt1);
  PT_INIT(&pt2);
  while(PT_SCHEDULE(thr(&pt1, 1) & thr(&pt2, 2)));
}
```

Beide Koroutinen abwechselnd vorantreiben

Beispiel

```
PT_THREAD(thr(struct pt *pt, int no)) {
  PT_BEGIN(pt);
  printf("Thr %d: A\n", no);
  PT_YIELD(pt);
  printf("Thr %d: B\n", no);
  PT_YIELD(pt);
  printf("Thr %d: C\n", no);
  PT_END(pt);
}
int main() {
  struct pt pt1, pt2;
  PT_INIT(&pt1);
  PT_INIT(&pt2);
  while(PT_SCHEDULE(thr(&pt1, 1) & thr(&pt2, 2)));
}
```

Ablauf:



Beispiel nach Expansion der Makros (vereinfacht)

```
bool thr(struct pt *pt, int no) {
  switch(pt->lc) {
  case 0:
    printf("Thr %d: A\n", no);
    pt->lc = 9;
    return false;
  case 9:
    printf("Thr %d: B\n", no);
    pt->lc = 11;
    return false;
  case 11:
    printf("Thr %d: C\n", no);
  }
  return true;
}
```

Beispiel nach Expansion der Makros (vereinfacht)

```
bool thr(struct pt *pt, int no) {
  switch(pt->lc) {
  case 0:
    printf("Thr %d: A\n", no);
    pt->lc = 9;
    return false;
  case 9:
    printf("Thr %d: B\n", no);
    pt->lc = 11;
    return false;
  case 11:
    printf("Thr %d: C\n", no);
  }
  return true;
}
```

PT_BEGIN(pt)

Switch–Anweisung setzt an der in pt gespeicherten Stelle fort

Beispiel nach Expansion der Makros (vereinfacht)

```
bool thr(struct pt *pt, int no) {
  switch(pt->lc) {
  case 0:
    printf("Thr %d: A\n", no);
    pt->lc = 9;
    return false;
  case 9:
    printf("Thr %d: B\n", no);
    pt->lc = 11;
    return false;
  case 11:
    printf("Thr %d: C\n", no);
  }
  return true;
}
```

PT_YIELD(pt)

Sichert Fortsetzungsadresse und kehrt zum Aufrufer zurück

Beispiel nach Expansion der Makros (vereinfacht)

```
bool thr(struct pt *pt, int no) {
  switch(pt->lc) {
  case 0:
    printf("Thr %d: A\n", no);
    pt->lc = 9;
    return false;
  case 9:
    printf("Thr %d: B\n", no);
    pt->lc = 11;
    return false;
  case 11:
    printf("Thr %d: C\n", no);
  }
  return true;
}
```

PT_END(pt)

Kehrt (endgültig) zurück

Weiteres Beispiel: Erzeuger/Verbraucher-Problem

```
PT_THREAD(producer(struct pt *pt)) {
  static int i;
  PT_BEGIN(pt);
  for (i=0; i<N; i++) {
    PT_SEM_WAIT(pt, &empty);
    insertItem(produce());
    PT_SEM_SIGNAL(pt, &full);
  }
  PT_END(pt);
}
```

- Schleifenvariable muß als `static` deklariert werden
  - die Werte lokaler Variablen werden beim Koroutinenwechsel nicht gesichert
- Kein wechselseitiger Ausschluß nötig

Weiteres Beispiel: Erzeuger/Verbraucher-Problem ...

```
PT_THREAD(consumer(struct pt *pt)) {
  static int i;
  PT_BEGIN(pt);
  for (i=0; i<N; i++) {
    PT_SEM_WAIT(pt, &full);
    consume(removeItem());
    PT_SEM_SIGNAL(pt, &empty);
  }
  PT_END(pt);
}
```

- `PT_SEM_WAIT` entspricht P() Operation auf einem Semaphor
- `PT_SEM_SIGNAL` entspricht V() Operation auf einem Semaphor
- `full` und `empty` sind aber lediglich `int`-Variable

Beispiel nach Makro-Expansion

```
bool consumer(struct pt *pt) {
  static int i;
  switch (pt->lc) {
  case 0:
      for (i=0; i<N; i++) {
          pt->lc = 94;
case 94:
          if (full <= 0)
              return false;
          full--;
          consume(removeItem());
          empty++;
      }
  }
  ...
```

PT_SEM_WAIT(pt, &full)

Semaphor P–Operation
kehrt zum Aufrufer zurück
statt zu blockieren => Polling

PT_SEM_SIGNAL(pt, &empty)

Semaphor V–Operation
zählt lediglich Zähler hoch

6.3 Asynchrone Programmierung

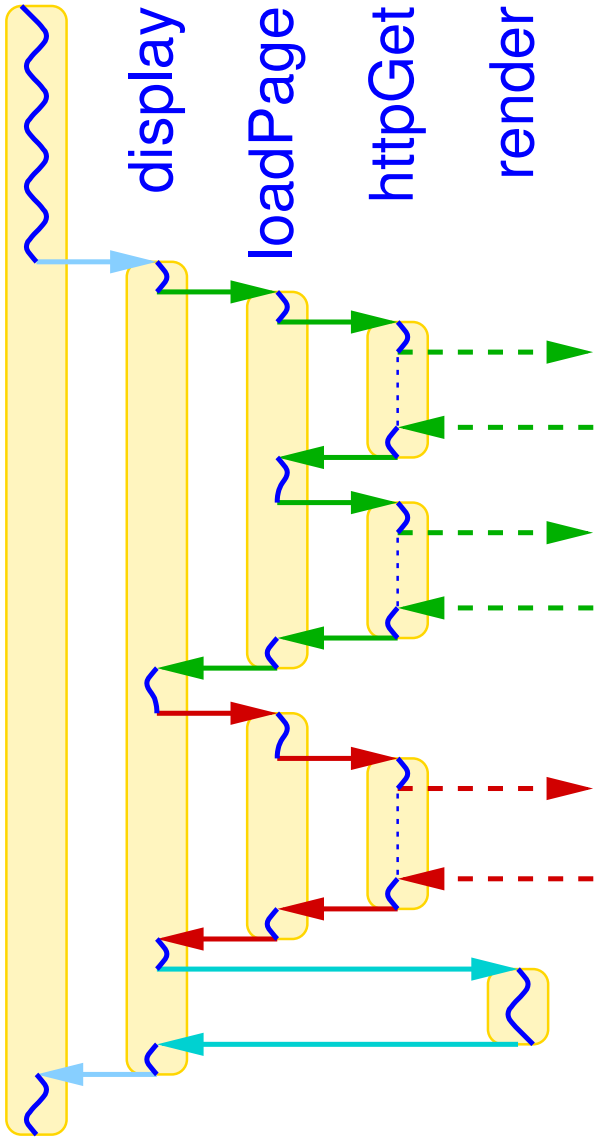
- Viele Programme (insbes. netzwerkbasierte Clients und Server) sind stark E/A-lastig
  - Programm blockiert die meiste Zeit, um auf E/A zu warten
- Wunsch: Warten auf E/A soll nebenläufig erfolgen können
  - z.B. unabhängige E/A Aufträge gleichzeitig ausführen
  - beschleunigt Programmausführung auch auf einer CPU
- Beispiel im Folgenden:
  - Web-Browser muss zwei verschiedene HTML-Dokumente laden, um eine Webseite darzustellen
  - HTML-Anfrage kann '*Redirect*' zurückliefern
    - dann ist eine neue Anfrage notwendig
    - d.h., Anfragen können auch voneinander abhängen

6.3.1 Sequentielle Realisierung

- Lade beide Seiten nacheinander
- Im Beispiel soll bei der ersten Seite zunächst ein '*Redirect*' kommen

```
void display() {
  String p1 = loadPage("...1");
  String p2 = loadPage("...2");
  render(p1, p2);
}
String loadPage(String url) {
  String p = httpGet(url);
  String target = checkRedirect(p);
  if (target != null)
    return httpGet(target);
  return p;
}
```

Zeitlicher Ablauf:

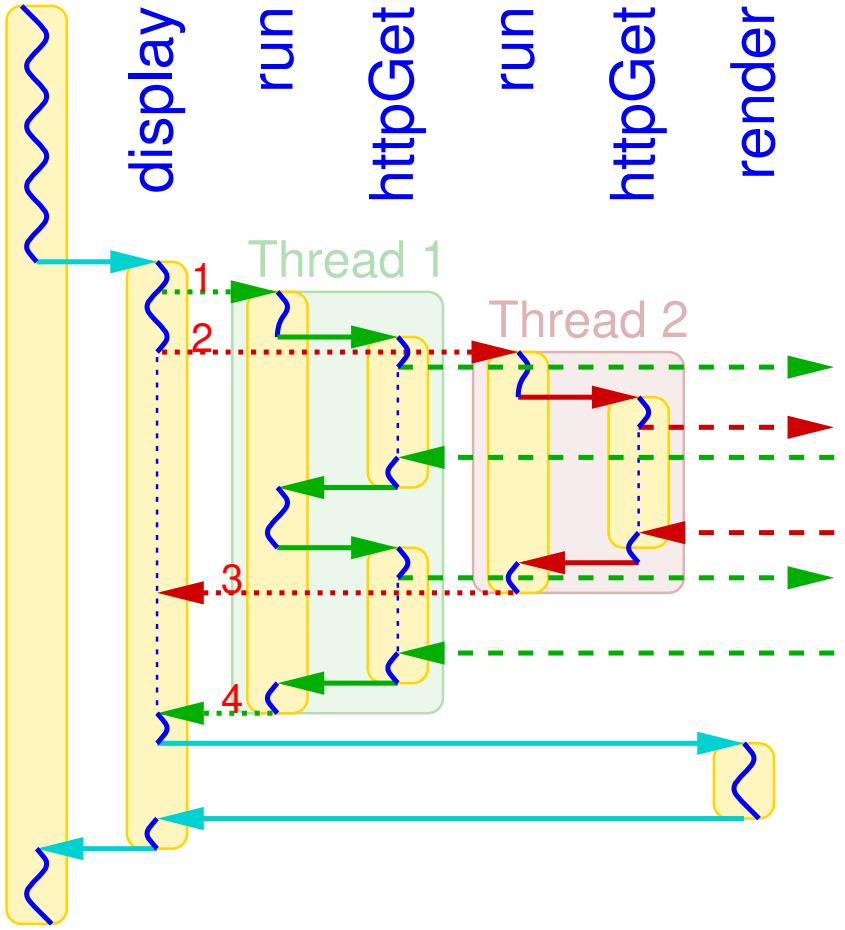


6.3.2 Realisierung mit Threads


```
void display() {
  HttpThread t1 = new HttpThread("...1"); t1.start();
  HttpThread t2 = new HttpThread("...2"); t2.start();
  t1.join(); t2.join();
  render(t1.res, t2.res);
}
class HttpThread extends Thread {
  String res, url;
  HttpThread(String url) { this.url = url; }
  void run() {
    res = httpGet(url);
    String target = checkRedirect(res);
    if (target != null)
      res = httpGet(target);
  }
}
```

6.3.2 Realisierung mit Threads

```
void display() {
  HttpThread t1 = new HttpThread("...1"); t1.start();
  HttpThread t2 = new HttpThread("...2"); t2.start();
  t1.join(); t2.join();
  render(t1.res, t2.res);
}
class HttpThread extends Thread
  String res, url;
  HttpThread(String url) { this
  void run() {
    res = httpGet(url);
    String target = checkRedire
    if (target != null)
      res = httpGet(target);
  }
}
```



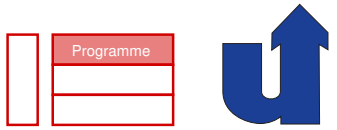
Diskussion

- Laden der beiden Seiten erfolgt nebenläufig
  - während des Wartens auf die erste Seite kann weitergearbeitet werden
- Relativ hoher Overhead zum Erzeugen der Threads
  - Systemaufrufe!
- Ressourcenverbrauch der Threads (insbes. Kellerspeicher) bei Servern problematisch
  - ggf. Tausende von nebenläufigen Anfragen!
- Ggf. Synchronisation der Threads erforderlich
- Komplexe Änderung der Programmstruktur notwendig

6.3.3 Asynchrone Programmierung

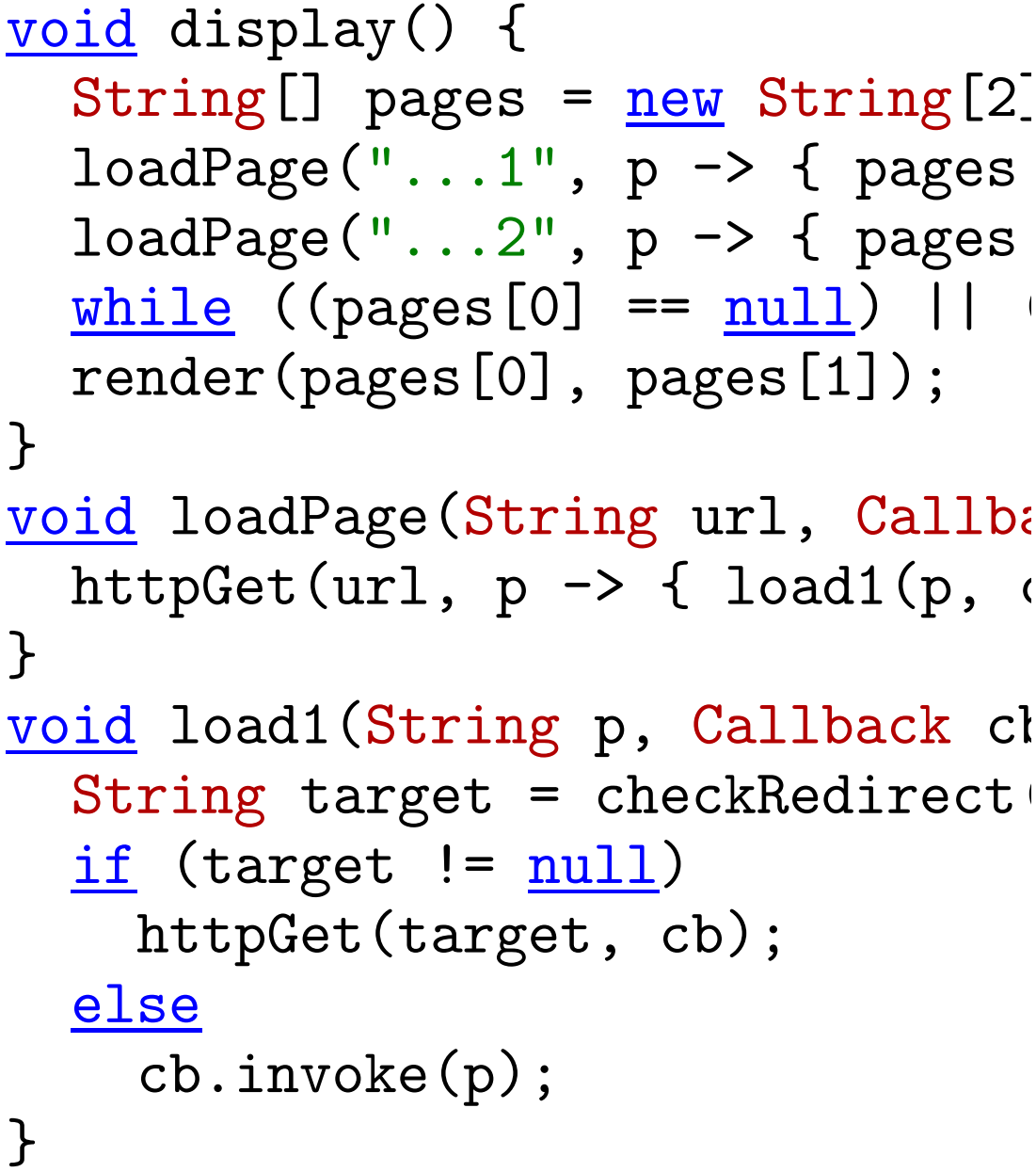
- Basis: Nutzung **a**synchroner Methodenaufrufe
- Idee: aufgerufene Methode kann zum Aufrufer zurückkehren, **bevor** ihre Aufgabe beendet ist
  - Aufgabe wird „im Hintergrund“ weiter bearbeitet
- Aufrufer kann jetzt nebenläufig weiterarbeiten
  - z.B. weitere asynchrone Methoden aufrufen
- Nach Abschluss der Aufgabe müssen aber i.a. weitere Aktionen durchgeführt werden
- Dazu zwei Alternativen:
  - Aufrufer kann bei Bedarf explizit auf Abschluss warten
  - bei Abschluss der Aufgabe wird automatisch eine vorgegebene Aktion (Methode) gestartet
    - z.B. in Form eines Callbacks

6.3.4 Asynch. Programmierung mit Callbacks

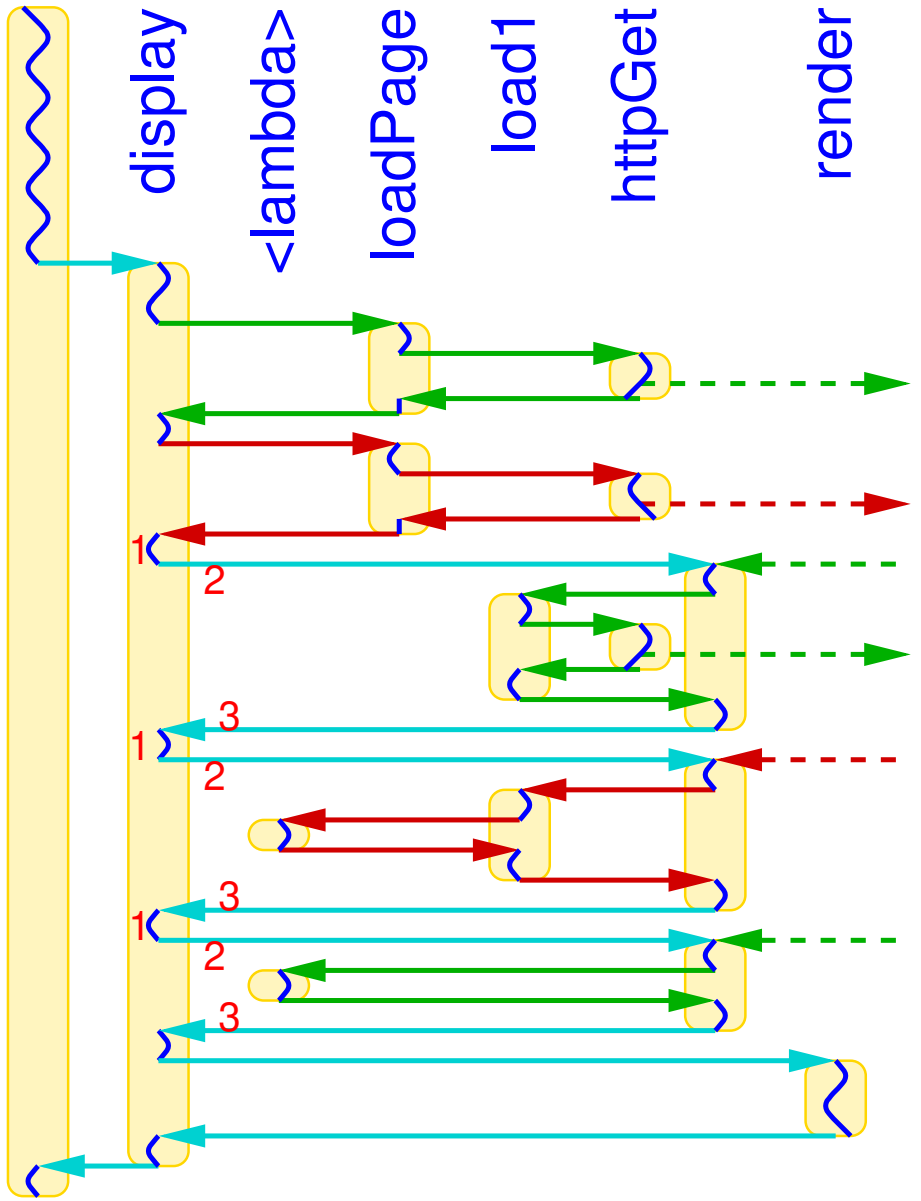



```
void display() {
  String[] pages = new String[2];
  loadPage("...1", p -> { pages[0] = p; });
  loadPage("...2", p -> { pages[1] = p; });
  while ((pages[0] == null) || (pages[1] == null));
  render(pages[0], pages[1]);
}
void loadPage(String url, Callback cb) {
  httpGet(url, p -> { load1(p, cb); });
}
void load1(String p, Callback cb) {
  String target = checkRedirect(p);
  if (target != null)
    httpGet(target, cb);
  else
    cb.invoke(p);
}
```

6.3.4 Asynch. Programmierung mit Callbacks



```
void display() {
  String[] pages = new String[2]
  loadPage("...1", p -> { pages
  loadPage("...2", p -> { pages
  while ((pages[0] == null) ||
  render(pages[0], pages[1]);
}
void loadPage(String url, Callba
  httpGet(url, p -> { load1(p, c
}
void load1(String p, Callback cb
  String target = checkRedirect
  if (target != null)
    httpGet(target, cb);
  else
    cb.invoke(p);
}
```



Diskussion

- Keine Verwendung von Threads nötig
  - z.B., wenn keine Threadunterstützung vorhanden ist
- Verwendung von Callbacks führt zu sehr unübersichtlicher Programmstruktur
- Bearbeitung im Hintergrund muß geeignet realisiert werden
  - z.B. mit Interrupt- oder Signalhandlern, ggf. auch mit Threads
- Übergang in die „synchrone Welt“ schwierig
  - benötigt eine Möglichkeit, auf den Aufruf eines Callbacks zu warten
  - im Beispiel mit aktivem Warten gelöst (unschön)

6.3.5 *Futures* und *Promises*

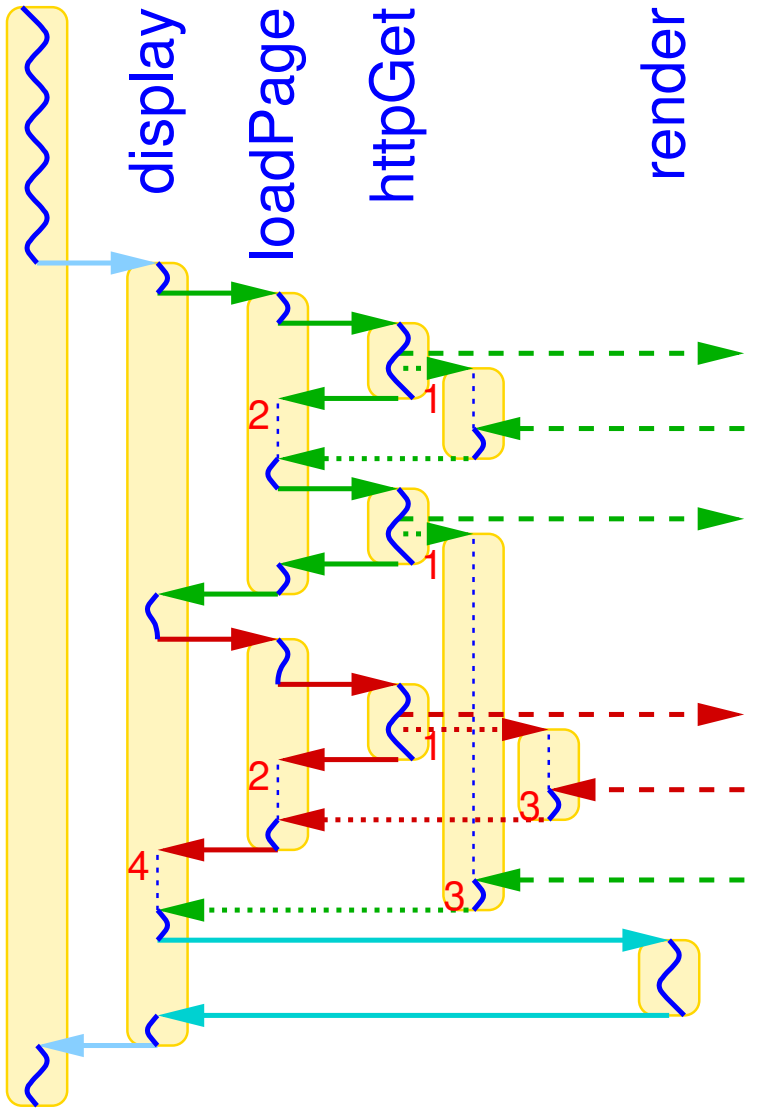
- ***Future***: Variable, deren Wert erst in der Zukunft verfügbar wird
  - sobald die zugehörige Berechnung beendet ist
- ***Promise***: Funktion/Berechung, die den Wert des *Future*s setzt
  - oft mit *Future* gleichgesetzt
- Unterscheidung explizites / implizites *Future*
  - explizit: vor der Verwendung des Werts muß explizit gewartet werden, bis dieser berechnet wurde
  - implizit: Compiler erzeugt Synchronisation, wo nötig
- *Future* kann als Argument etc. weitergegeben werden, ohne auf den Wert zu warten
- Ggf. auch Aufruf einer Berechnung mit dem noch nicht vorhandenen Wert möglich (***Promise Chaining***):
  - Ergebnis ist dann wieder ein *Future*

Explizite *Futures* mit blockierendem Warten

```
import java.util.concurrent.*;

void display() {
  Future<String> p1 = loadPage("...1");
  Future<String> p2 = loadPage("...2");
  render(p1.get(), p2.get());
}
Future<String> loadPage(String url) {
  String p = httpGet(url).get();
  String target = checkRedirect(p);
  if (target != null)
    return httpGet(target);
  return CompletableFuture.completedFuture(p);
}
```

Explizite *Futures* mit blockierendem Warten

```
import java.util.concurrent.*;

void display() {
  Future<String> p1 = loadPage("...
  Future<String> p2 = loadPage("...
  render(p1.get(), p2.get());
}
Future<String> loadPage(String url)
  String p = httpGet(url).get();
  String target = checkRedirect(p);
  if (target != null)
    return httpGet(target);
  return CompletableFuture.complete
}
```

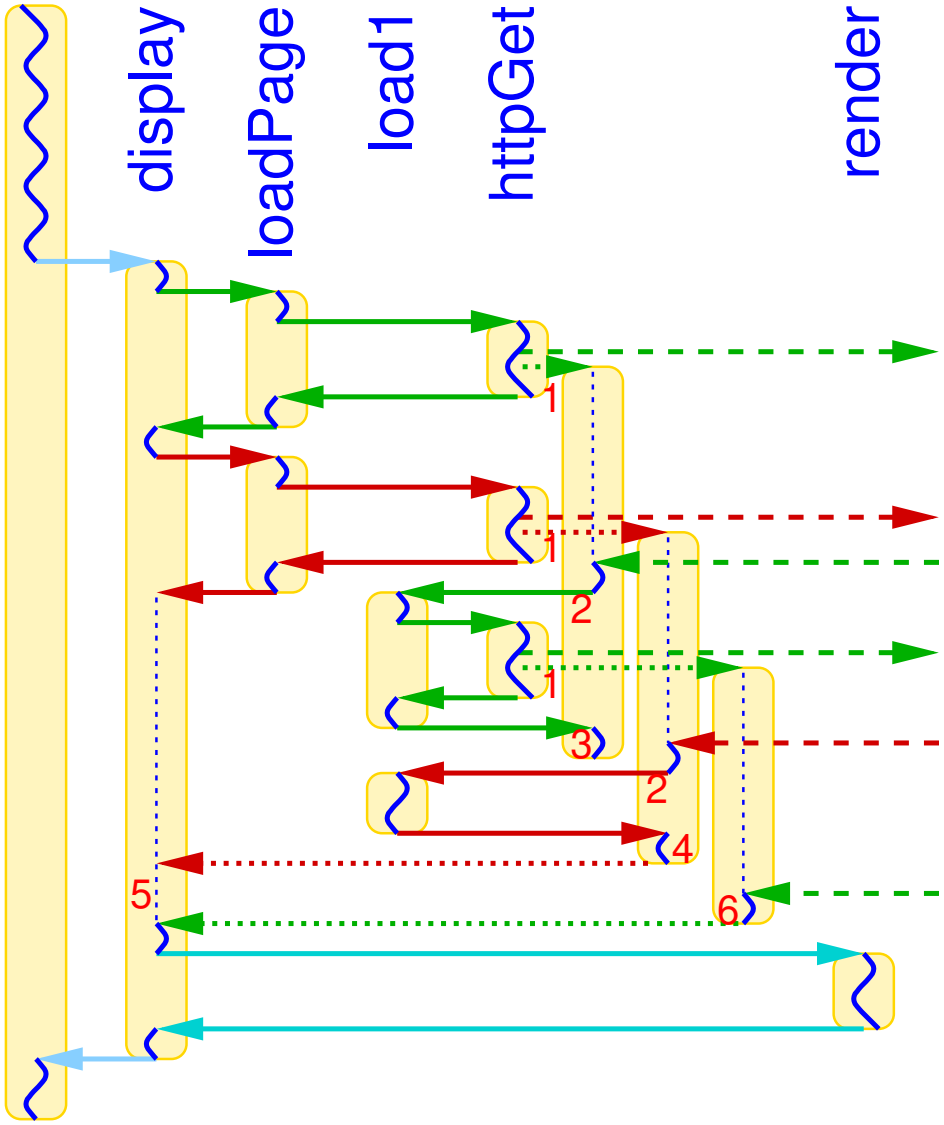


Futures mit Promise Chaining

```
void display() {
  Future<String> p1 = loadPage("...1");
  Future<String> p2 = loadPage("...2");
  render(p1.get(), p2.get());
}
Future<String> loadPage(String url) {
  // Anmerkung: so in Java NICHT möglich!!
  return httpGet(url).then(p -> { load1(p); });
}
Future<String> load1(String p) {
  String target = checkRedirect(p);
  if (target != null)
    return httpGet(target);
  return CompletableFuture.completedFuture(p);
}
```

Futures mit Promise Chaining

```
void display() {
   Future<String> p1 = loadPage
   Future<String> p2 = loadPage
   render(p1.get(), p2.get());
}
Future<String> loadPage(String
   // Anmerkung: so in Java NICHT möglich
   return httpGet(url).then(p -
}
Future<String> load1(String p)
   String target = checkRedirec
   if (target != null)
      return httpGet(target);
   return CompletableFuture.com
}
```



6.3.6 Async / Await (Koroutinen)

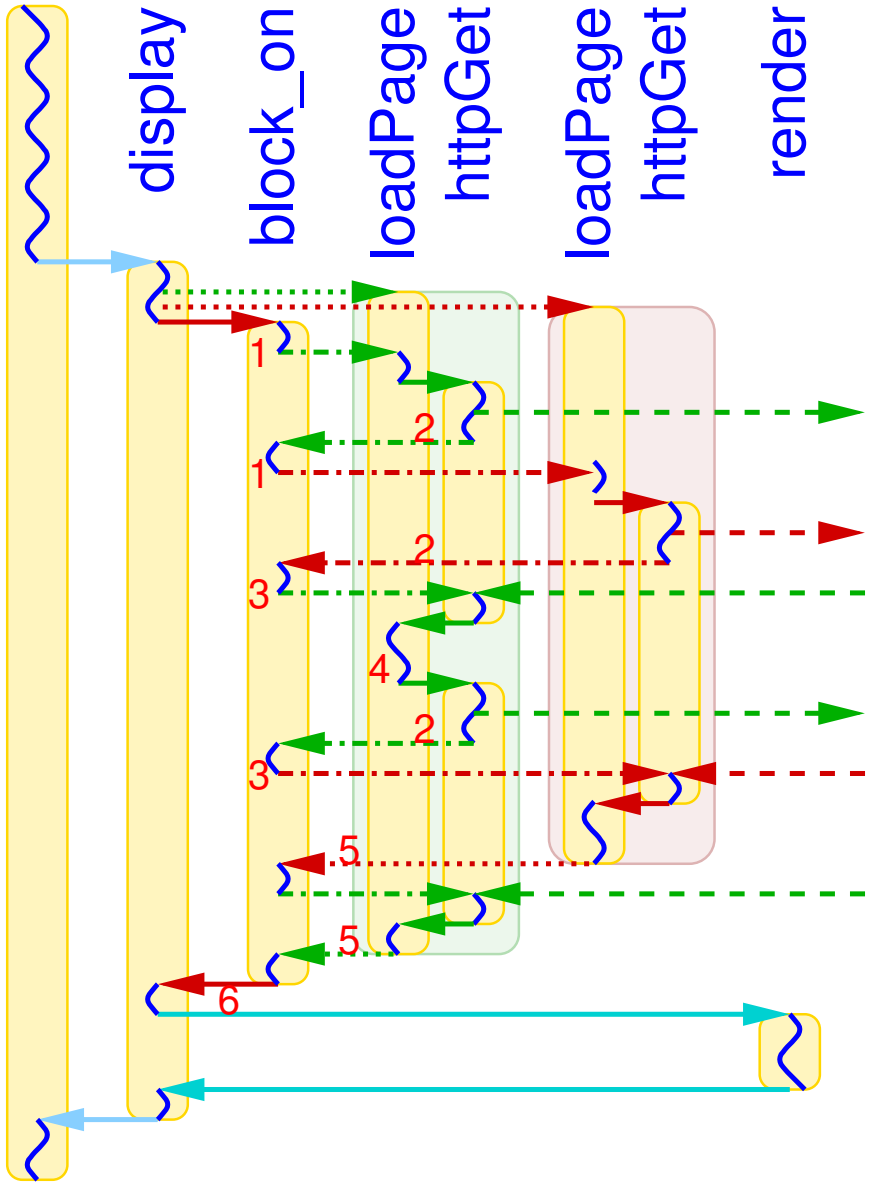
- Unterstützung asynchroner Funktionsaufrufe durch Sprache / Compiler
- Schlüsselwort `async` kennzeichet asynchrone Funktion
  - liefert automatisch ein *Future* zurück
- Schlüsselwort `await` „wartet“, bis der Wert verfügbar ist
  - bedeutet hier: Funktion (Koroutine!) kehrt erst einmal zurück, neuer Aufruf, wenn Wert (voraussichtlich) zur Verfügung steht
  - benötigt einen *Executor*, der regelmäßiges Polling der Koroutinen durchführt
- Programmstruktur sehr ähnlich zu sequentiellem Programm
  - aber: zeitlicher Ablauf trotzdem sehr komplex
- Realsierung ohne Threads möglich (z.B. in Rust)

6.3.6 Async / Await (Koroutinen) ...

```
// Anmerkung: so in Java NICHT möglich!!
void display() {
  Future<String> p1 = loadPage("...1");
  Future<String> p2 = loadPage("...2");
  Future<String[]> pp = join(p1, p2);
  String[] p = block_on(pp);
  render(p[0], p[1]);
}
async String loadPage(String url) {
  String p = await httpGet(url);
  String target = checkRedirect(p);
  if (target != null)
    return await httpGet(target);
  return p;
}
```

6.3.6 Async / Await (Koroutinen) ...

```
// Anmerkung: so in Java NICHT möglich!!
void display() {
   Future<String> p1 = loadPage("..
   Future<String> p2 = loadPage("..
   Future<String[]> pp = join(p1, p
   String[] p = block_on(pp);
   render(p[0], p[1]);
}
async String loadPage(String url)
   String p = await httpGet(url);
   String target = checkRedirect(p)
   if (target != null)
      return await httpGet(target);
   return p;
}
```



6.4 Asynchrone Ein-/Ausgabe in POSIX

- Asynchrone Alternativen für die blockierenden Systemaufrufe `read()` und `write()`
  - Zugriff auf Dateien und Geräte, Netzwerkkommunikation
- `aio_read()` und `aio_write()` starten Lese- bzw. Schreib-Aufträge, ohne auf Beendigung zu warten
- Auftrag enthält u.a.:
  - Dateideskriptor
  - Offset in der Datei
    - kein globaler Dateizeiger wegen nebenläufiger Operationen
  - Quell- bzw. Zielpuffer mit Länge
  - Benachrichtigungs-Methode: keine Benachrichtigung, Signal, oder Aufruf einer Funktion in einem separaten Thread

6.4 Asynchrone Ein-/Ausgabe in POSIX ...

- `aio_error()` erlaubt Abfrage des Ausführungszustands
  - inkl. Fehlerstatus
- `aio_suspend()` blockiert den Aufrufer, bis mindestens einer der spezifizierten Aufträge beendet ist
- Mit `lio_listio()` kann eine ganze Liste nebenläufiger Aufträge gestartet werden
  - zur Einsparung von Systemaufrufen

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

7 Scheduling

7 Scheduling ...

Inhalt:

- Einführung
- Kriterien für das Scheduling
- Scheduling-Verfahren: allgemeine Aspekte
- Scheduling-Algorithmen

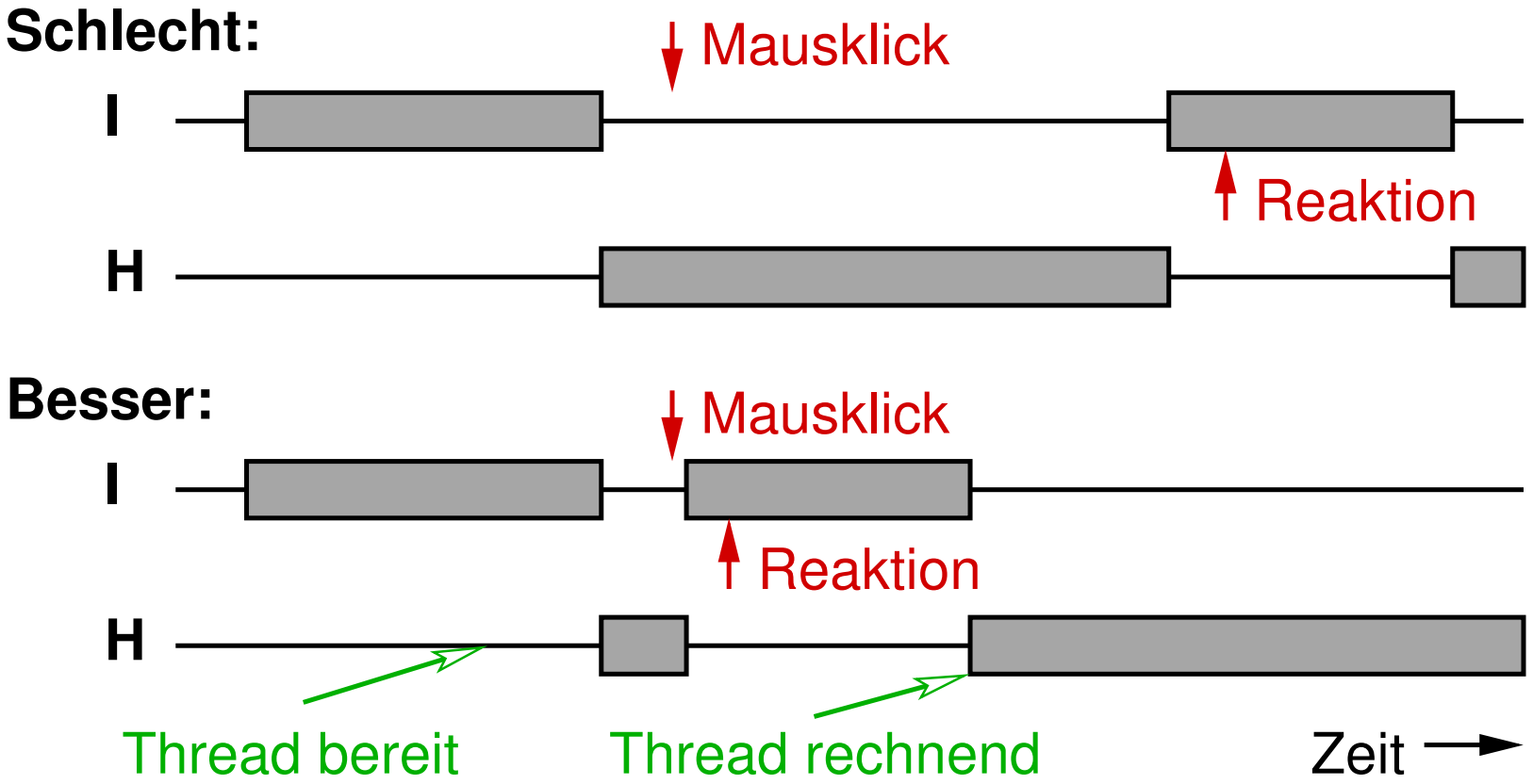
- Tanenbaum 2.5
- Stallings 9
- Nehmer/Sturm 5.3

Teilaufgabe des BS: Mehrprogammbetrieb

- „Gleichzeitige“ Abarbeitung mehrerer Threads (und damit auch mehrerer Programme) im schnellen Wechsel
- Bisher behandelt:
  - Mechanismus des Threadswechsels
- Jetzt: **Thread-Scheduling**
  - Auswahlstrategie: welcher Thread darf als nächstes wie lange (und ggf. auf welcher CPU) rechnen?
  - Verwaltung des Betriebsmittels Prozessor
- (Daneben: weitere Scheduling-Aufgaben, ☞ **8.3.3, 9.3**)

Scheduling beeinflußt Nutzbarkeit des Systems

- Beispiel: interaktiver Thread (I) und Hintergrundthread (H)
  - Darstellung als **Gantt-Diagramm**

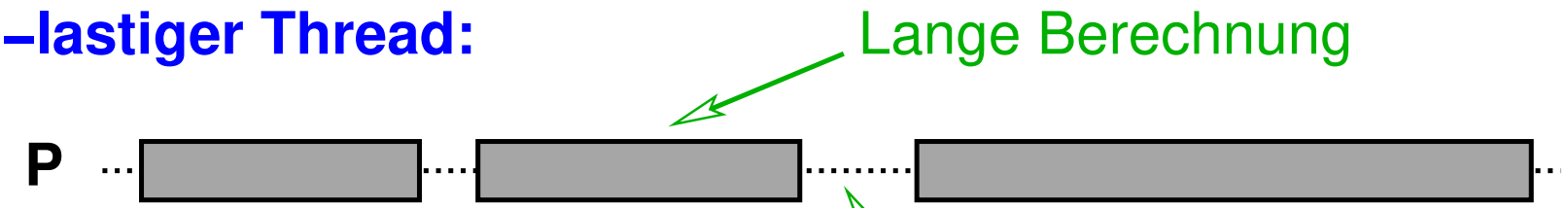


Vorbemerkung: Betriebsarten eines Systems

- Stapelverarbeitungs-Betrieb
  - System arbeitet nicht-interaktive Aufträge ab
  - häufig bei Großrechnern
- Interaktiver Betrieb
  - typisch für Arbeitsplatzrechner, Server
- Echtzeitbetrieb
  - Steueraufgaben, Multimedia-Anwendungen

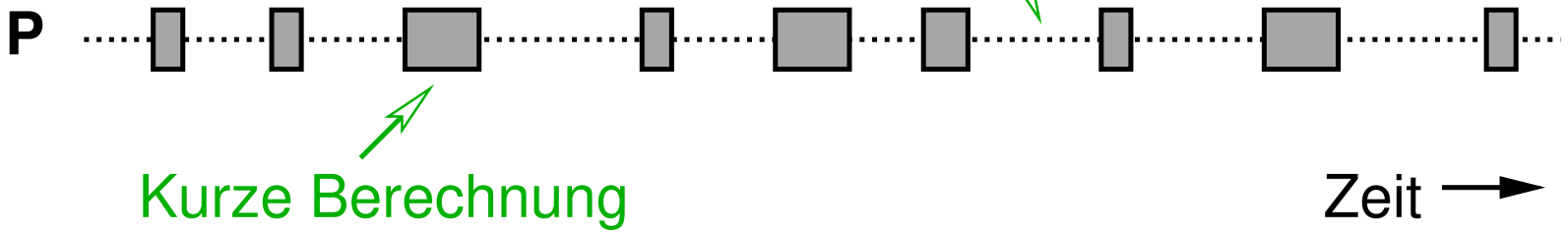
Vorbemerkung: Threadcharakteristiken

CPU–lastiger Thread:



Warten auf E/A

E/A–lastiger Thread:



Kriterien für das Scheduling (Benutzersicht)

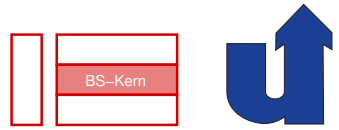
- Minimierung der Durchlaufzeit (Stapelbetrieb)
  - Zeit zwischen Eingang und Abschluß eines Jobs (inkl. aller Wartezeiten)
- Minimierung der Antwortzeit (Interaktiver Betrieb)
  - Zeit zwischen Eingabe einer Anfrage und Beginn der Ausgabe
- Einhaltung von Terminen (Realzeitbetrieb)
  - Ausgabe muß nach bestimmter Zeit erfolgt sein
- Vorhersehbarkeit
  - Durchlaufzeit / Antwortzeit i.W. unabhängig von Auslastung des Systems

Kriterien für das Scheduling (Systemsicht)

- Maximierung des Durchsatzes (Stapelbetrieb)
  - Anzahl der fertiggestellten Jobs pro Zeiteinheit
- Optimierung der Prozessorauslastung (Stapelbetrieb)
  - prozentualer Anteil der Zeit, in der Prozessor beschäftigt ist
- Balance
  - gleichmäßige Auslastung aller Ressourcen
- Fairness
  - vergleichbare Threads sollten gleich behandelt werden
- Durchsetzung von Prioritäten
  - Threads mit höherer Priorität bevorzugt behandeln

Scheduling-Kriterien nicht gleichzeitig erfüllbar

- z.B. Prozessorauslastung $\leftrightarrow$ kurze Antwortzeit
- Scheduling-Ziele und -Verfahren von Betriebsart abhängig
  - Stapelverarbeitung: hoher Durchsatz, gute Auslastung
    - bevorzuge Aufträge, die freie Ressourcen nutzen
  - Interaktiver Betrieb: kurze Antwortzeiten
    - bevorzuge Threads, die auf E/A (Benutzereingabe) gewartet haben und nun rechenbereit sind
  - Echtzeitbetrieb: Einhaltung von Zeitvorgaben
    - bevorzuge Threads, deren Ausführungsfristen ablaufen



Ebenen des Schedulings

- Langfristiges Scheduling (Eingangs-Scheduler)
  - bei Systemen mit Stapelverarbeitung von Jobs:
    - welcher Job wird als nächstes zugelassen?
      D.h. wann werden Prozesse für den Job erzeugt?
  - Ziel z.B. Mischung aus CPU- und E/A-lastigen Prozessen
- Mittelfristiges Scheduling (Speicher-Scheduler)
  - Auslagern und Suspendieren von Prozessen (z.B. bei Speicherengpässen)
  - legt Multiprogramming-Grad fest
- Kurzfristiges Scheduling (CPU-Scheduler)
  - Zuteilung der CPU(s) an bereite Threads

Nicht-präemptives und präemptives Scheduling

- **Nicht-präemptives Scheduling**
  - Thread darf so lange rechnen, bis er freiwillig die CPU aufgibt oder blockiert
  - sinnvoll bei Stapelverarbeitung und teilw. Echtzeitsystemen
- **Präemptives Scheduling**
  - Scheduler kann einem Thread die CPU zwangsweise entziehen
    - wenn ein anderer Thread (mit höherer Priorität) rechenbereit geworden ist
    - nach Ablauf einer bestimmten Zeit
  - unterstützt interaktive Systeme und Echtzeitsysteme

Scheduling-Zeitpunkte (CPU-Scheduler)

- Bei Beendigung eines Threads
- Bei freiwilliger CPU-Aufgabe eines Threads
- Bei Blockierung eines Threads
  - Grund der Blockierung kann/sollte berücksichtigt werden
  - z.B. bei P()-Operation: weise CPU dem Thread zu, der Semaphor belegt $\Rightarrow$ kürzere Blockierung
- Bei Erzeugung eines Threads
- Bei E/A-Interrupt
  - evtl. werden blockierte Threads rechenbereit
- Regelmäßig bei Timer-Interrupt

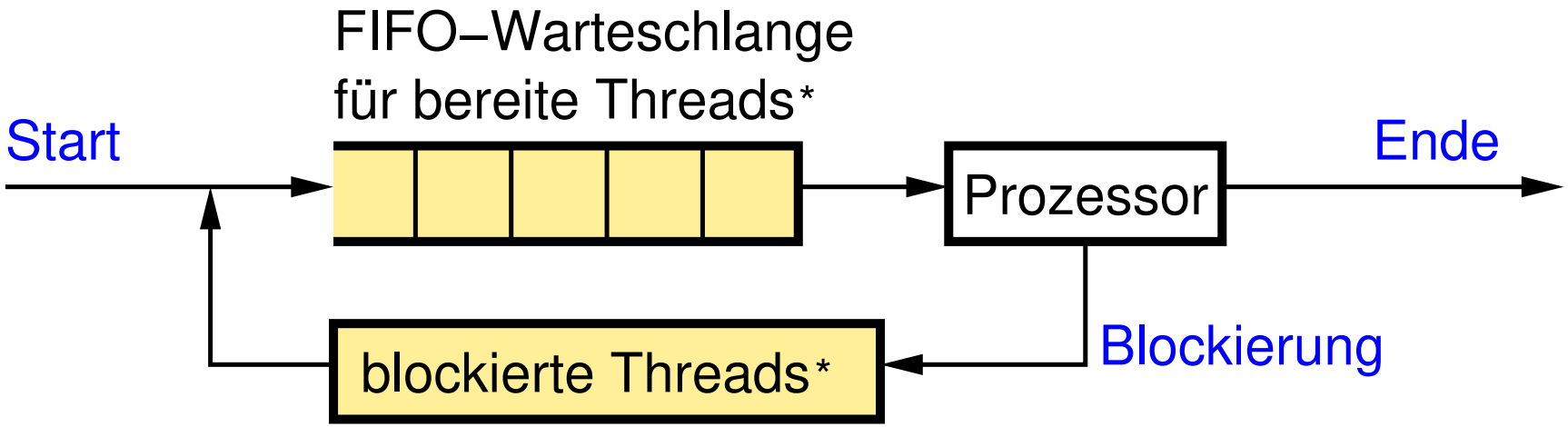
} nur bei prä–emptivem Scheduling

7.4 Scheduling-Algorithmen

- Legen fest, welcher Thread auf einer CPU als nächstes ausgeführt wird und wie lange er rechnen darf
- Bei Multiprozessor-Systemen prinzipiell:
  - jede CPU führt Algorithmus unabhängig von den anderen aus
  - Zugriffe auf Bereit-Warteschlange unter wechselseitigem Ausschluß
- Detailaspekt: ***Cache-Affinity*** / ***Affinity Scheduling***
  - ein einmal auf einer CPU ausgeführter Thread sollte nach Möglichkeit auf dieser CPU bleiben
    - Daten des Threads sind im Cache dieser CPU!
  - jeder Thread erhält eine (oder mehrere) bevorzugte CPU(s)
  - einfache Realisierung: getrennte Warteschlangen für jede CPU
    - falls leer: Warteschlangen anderer CPUs inspizieren

7.4.1 *First Come First Served* (FCFS)

- Nicht-präemptives Verfahren

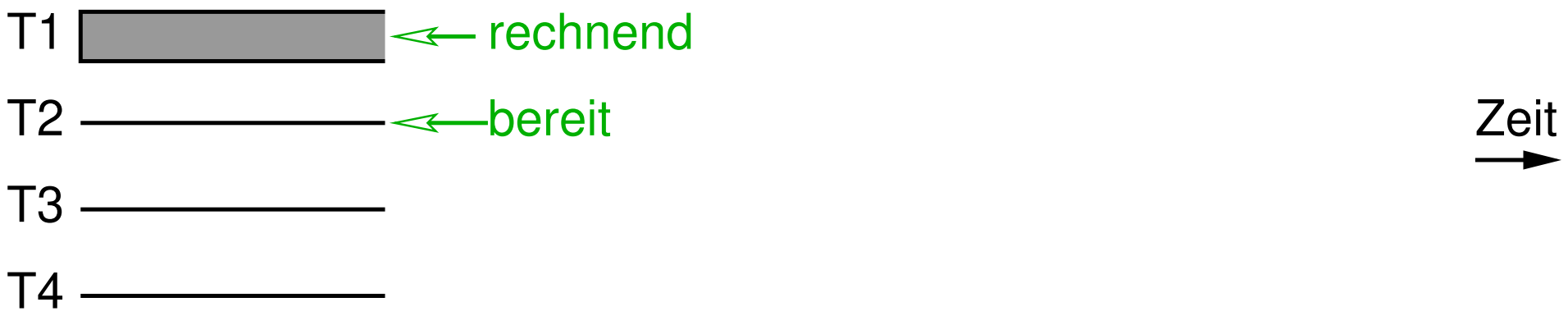


- Der am längsten wartende bereite Thread* darf als nächstes rechnen
  - nach Aufhebung einer Blockierung reiht sich ein Thread* wieder hinten in die Warteschlange ein

* bzw. Job(s) bei Stapelbetrieb

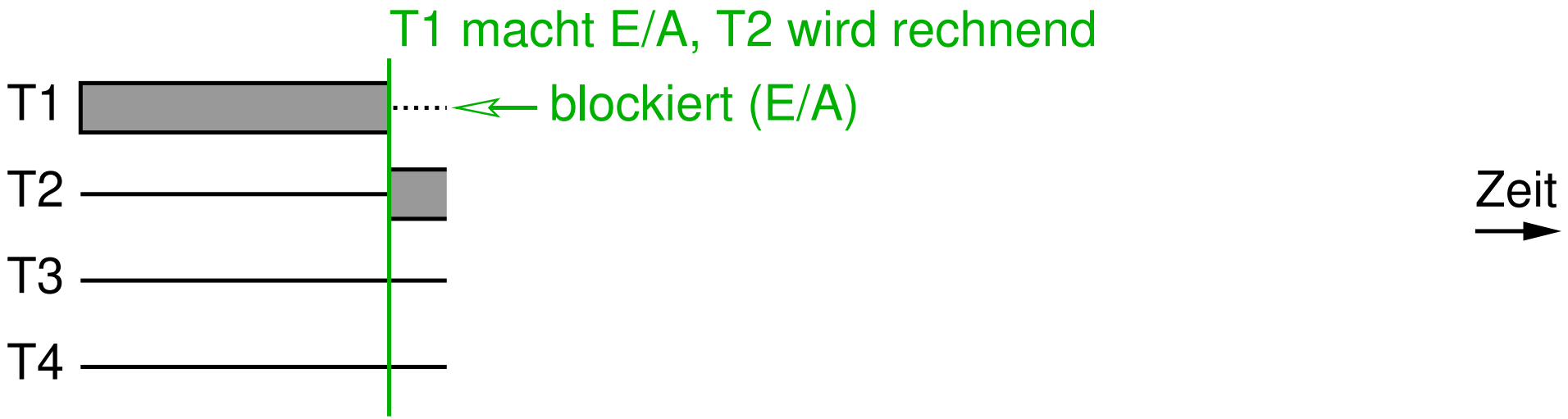
Diskussion

- Sehr einfacher Algorithmus
- Stellt gute CPU-Auslastung sicher
- Vorwiegend für Stapelverarbeitung
- Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- Balance evtl. schlecht durch Convoy-Effekt



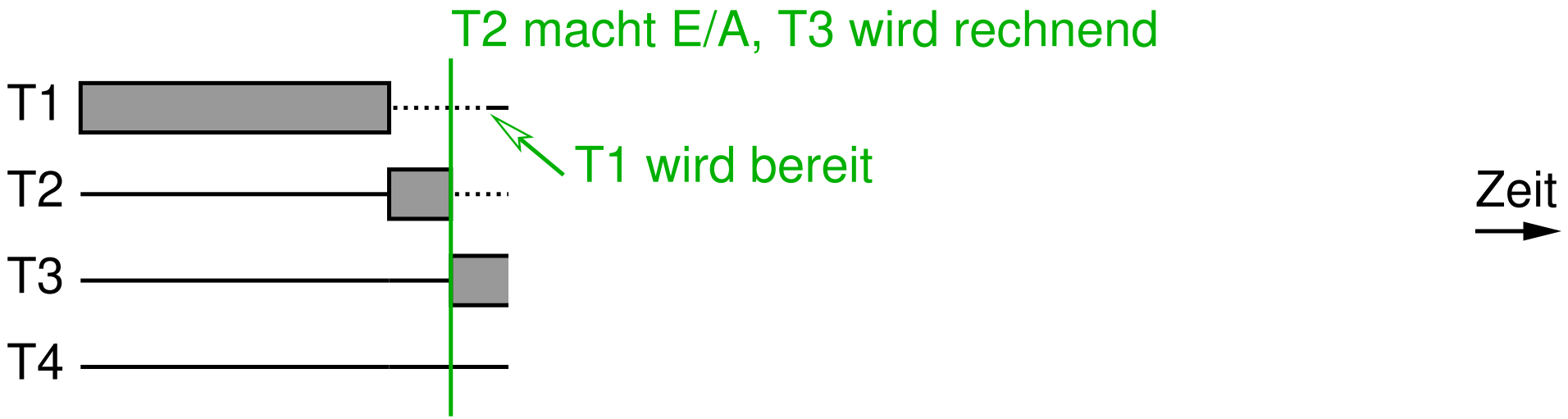
Diskussion

- Sehr einfacher Algorithmus
- Stellt gute CPU-Auslastung sicher
- Vorwiegend für Stapelverarbeitung
- Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- Balance evtl. schlecht durch Convoy-Effekt



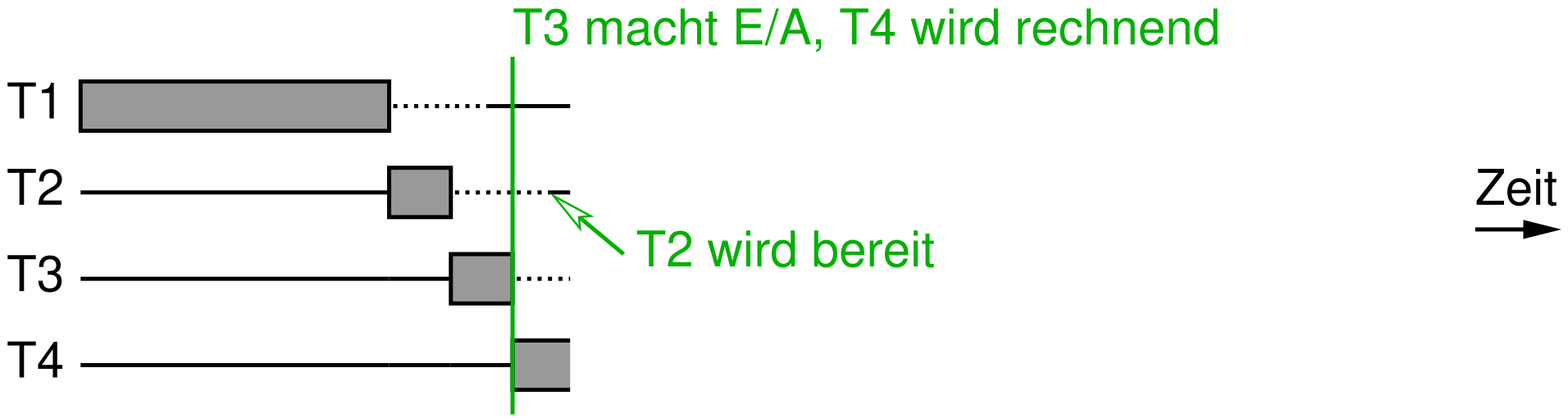
Diskussion

- Sehr einfacher Algorithmus
- Stellt gute CPU-Auslastung sicher
- Vorwiegend für Stapelverarbeitung
- Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- Balance evtl. schlecht durch Convoy-Effekt



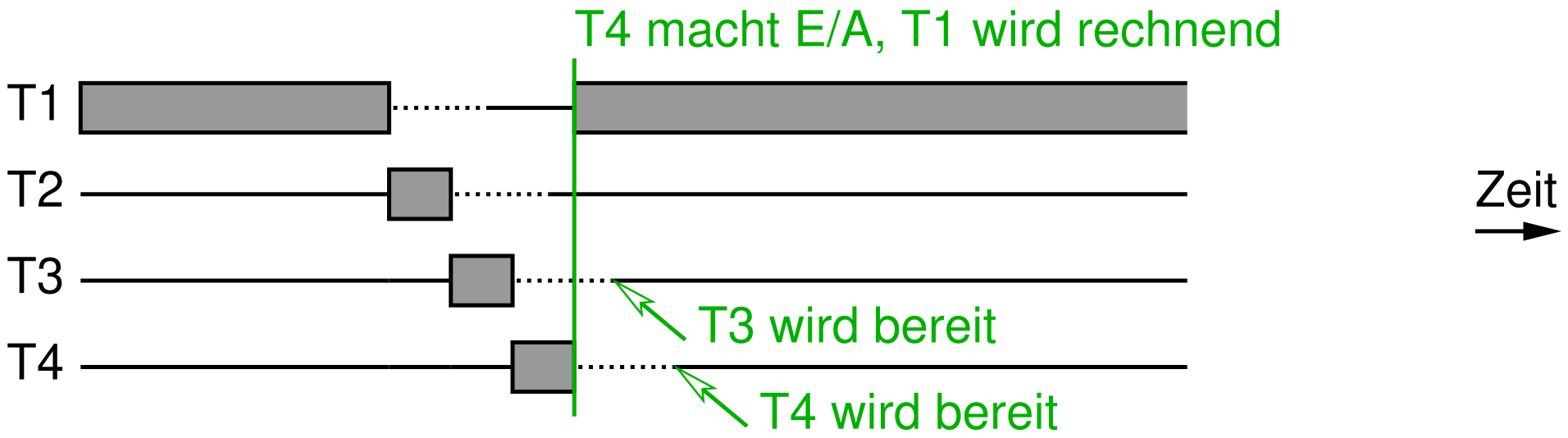
Diskussion

- Sehr einfacher Algorithmus
- Stellt gute CPU-Auslastung sicher
- Vorwiegend für Stapelverarbeitung
- Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- Balance evtl. schlecht durch Convoy-Effekt



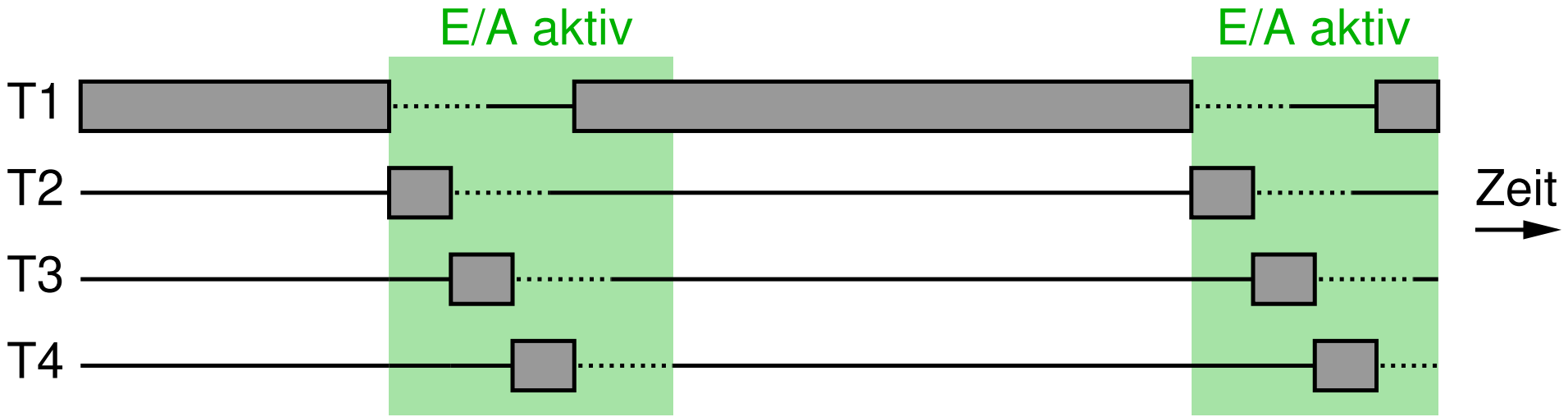
Diskussion

- Sehr einfacher Algorithmus
- Stellt gute CPU-Auslastung sicher
- Vorwiegend für Stapelverarbeitung
- Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- Balance evtl. schlecht durch Convoy-Effekt



Diskussion

- Sehr einfacher Algorithmus
- Stellt gute CPU-Auslastung sicher
- Vorwiegend für Stapelverarbeitung
- Durchlaufzeit wird nicht optimiert (siehe gleich: SJF)
- Balance evtl. schlecht durch Convoy-Effekt

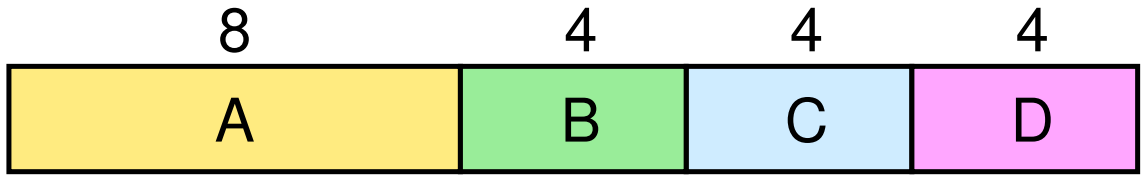


7.4.2 *Shortest Job First* (SJF)

- Ziel: Optimierung der Durchlaufzeit
- Strategie: kürzester Job wird zuerst gerechnet
  - Dauer muß vorab bekannt sein (bei Stapelbetrieb i.a. erfüllt)
- SJF ist beweisbar optimal, falls alle Jobs gleichzeitig vorliegen
  - in der Regel: Jobs treffen nacheinander ein, Scheduler kann nur die Jobs berücksichtigen, die bereits bekannt sind
- Einfachster Fall: nicht-präemptiv, keine Blockierungen durch E/A
- In präemptiver Variante und bei Blockierungen durch E/A:
  - betrachte **Rest**laufzeiten
- Variante für interaktive Systeme:
  - betrachte CPU-Burst als Job, mit Schätzung der Dauer

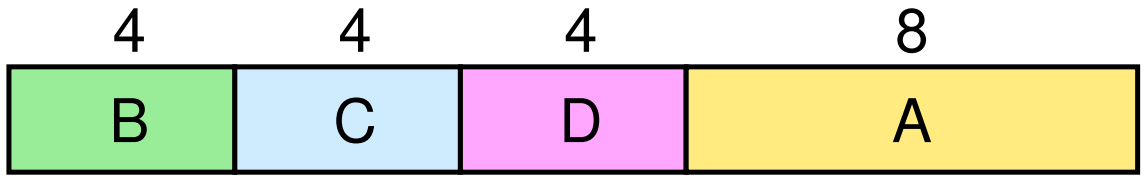
Beispiel (alle Jobs liegen gleichzeitig vor)

Vier Jobs in FCFS–Reihenfolge



Mittlere Durchlaufzeit:
(8+12+16+20)/4 = 14

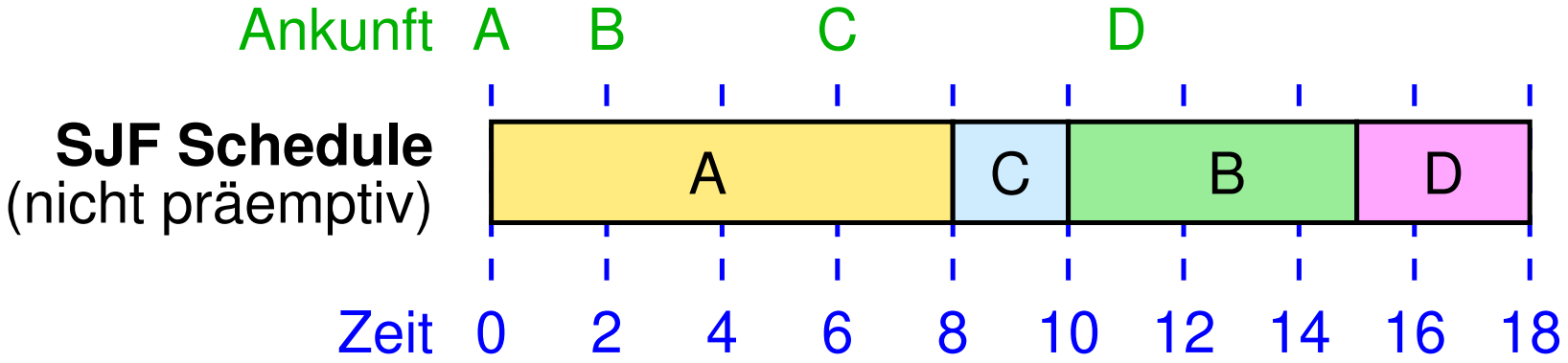
Dieselben Jobs in SJF–Reihenfolge



Mittlere Durchlaufzeit:
(4+8+12+20)/4 = 11

Beispiel (die Jobs treffen nacheinander ein)

Job	Ankunftszeit	Bedienzeit
A	0	8
B	2	5
C	6	2
D	11	3

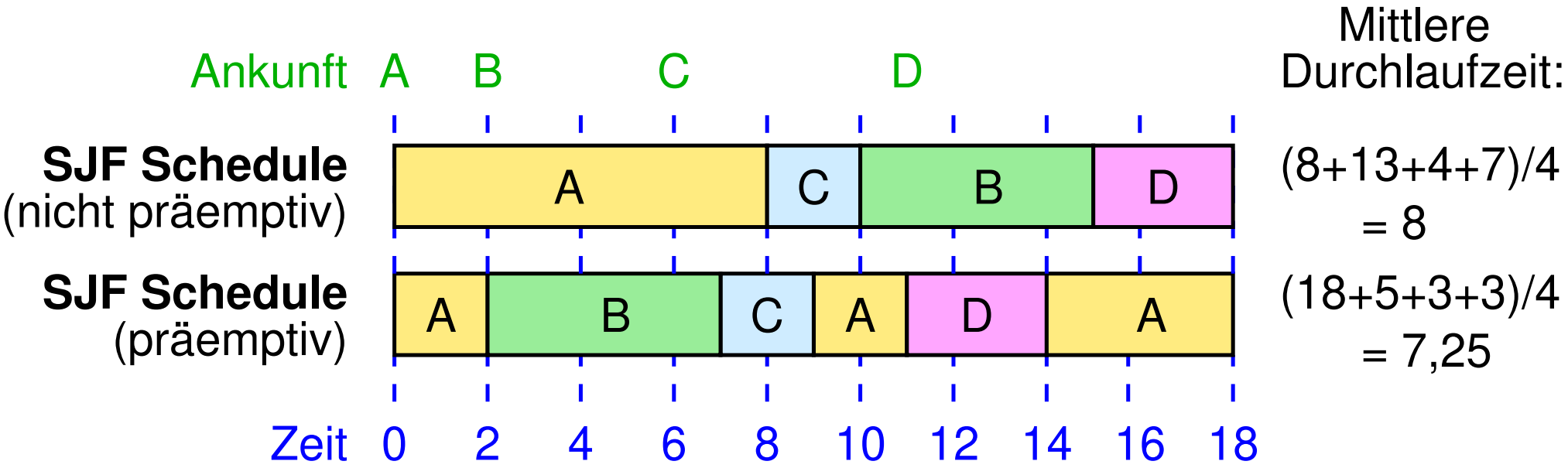


Mittlere Durchlaufzeit:

(8+13+4+7)/4
= 8

Beispiel (die Jobs treffen nacheinander ein)

Job	Ankunftszeit	Bedienzeit
A	0	8
B	2	5
C	6	2
D	11	3



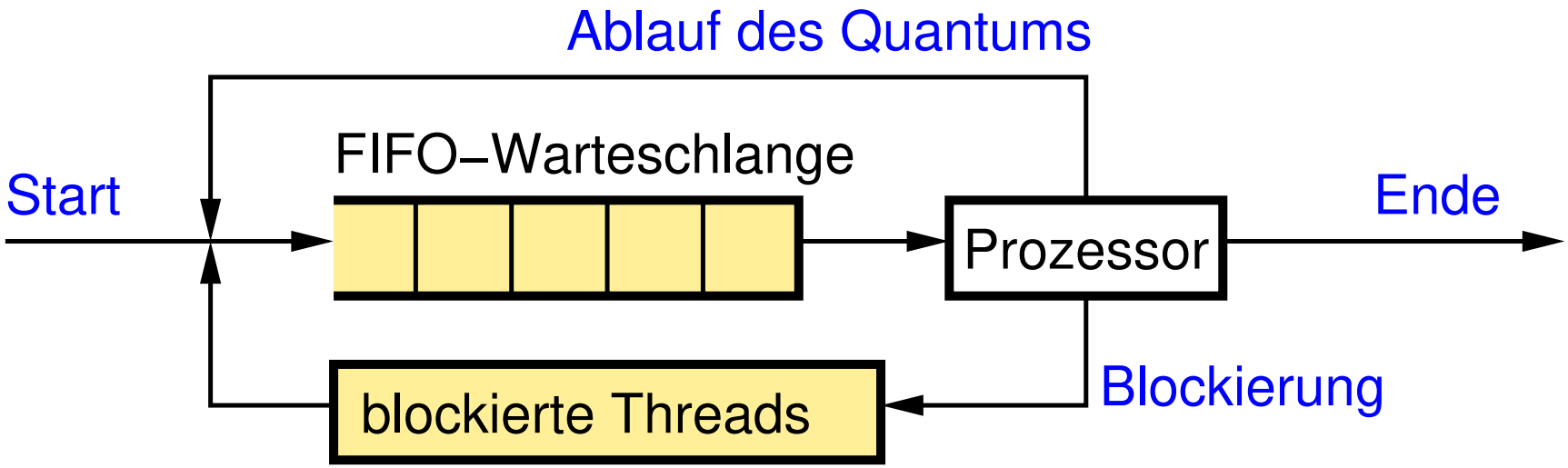
Mittlere Durchlaufzeit:

(8+13+4+7)/4 = 8

(18+5+3+3)/4 = 7,25

7.4.3 *Round Robin* (RR), Zeitscheibenverfahren

- Präemtive Variante von FCFS
- Jeder Thread darf höchstens eine bestimmte Zeit (**Quantum**, **Zeitscheibe**, ***Time Slice***) rechnen

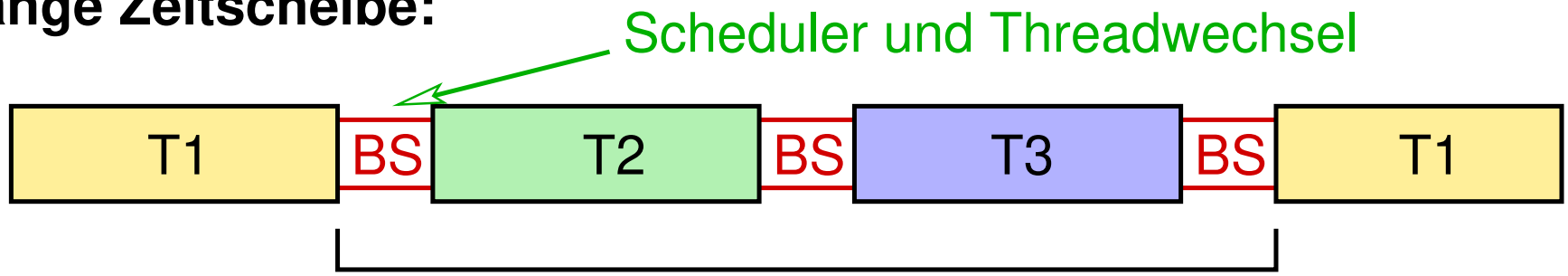


Diskussion

- Einfach zu realisieren
  - FIFO-Warteschlange aller rechenbereiten Threads und Timer-Interrupt
- Frage: Länge des Quantums
  - kurzes Quantum:
    - kurze Antwortzeiten
    - schlechte CPU-Nutzung durch häufige Threadwechsel
  - langes Quantum: umgekehrt
  - Praxis: 10 - 100 ms
- RR ist nicht fair:
  - E/A-lastige Threads werden benachteiligt

Zeitscheibenlänge

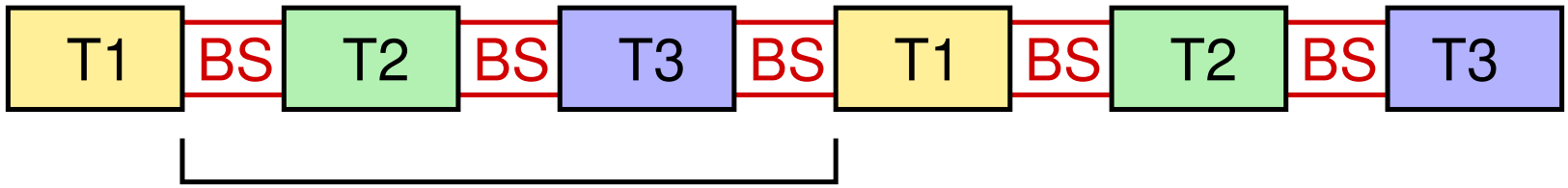
Lange Zeitscheibe:



gute Prozessornutzung, hohe Antwortzeit

Zeit

Kurze Zeitscheibe:

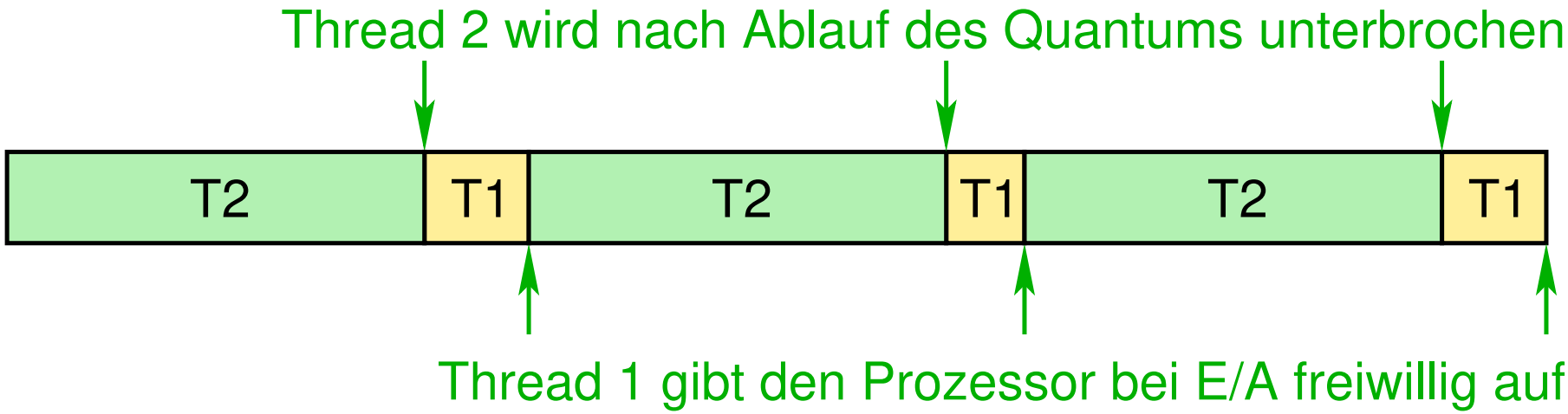


kurze Antwortzeit, schlechte Prozessornutzung

Fairness

T1 E/A–lastiger Thread

T2 CPU–lastiger Thread



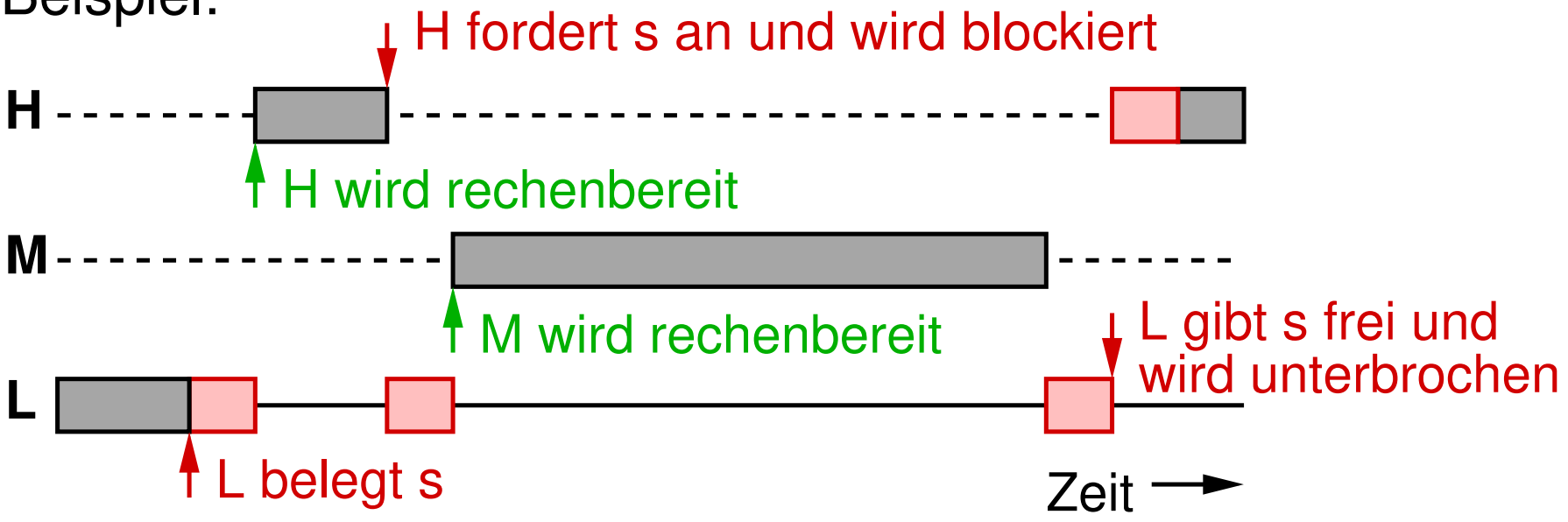
➥ Thread 1 ist gegenüber Thread 2 benachteiligt

7.4.4 Prioritätenbasiertes Scheduling

- Jeder Thread erhält eine (unterschiedliche) Priorität
  - CPU wird an den Thread mit der höchsten Priorität zugeteilt
- I.d.R. präemptiv
- Statische Prioritäten
  - Priorität eines Threads bei Erzeugung festgelegt
  - häufig bei Realzeit-Systemen
- Dynamische Prioritäten
  - Priorität kann laufend geändert werden, abhängig vom Verhalten des Threads (**Feedback Scheduling**)
  - z.B.: Priorität bestimmt durch Länge des letzten CPU-Bursts des Threads ($\sim$ SJF)

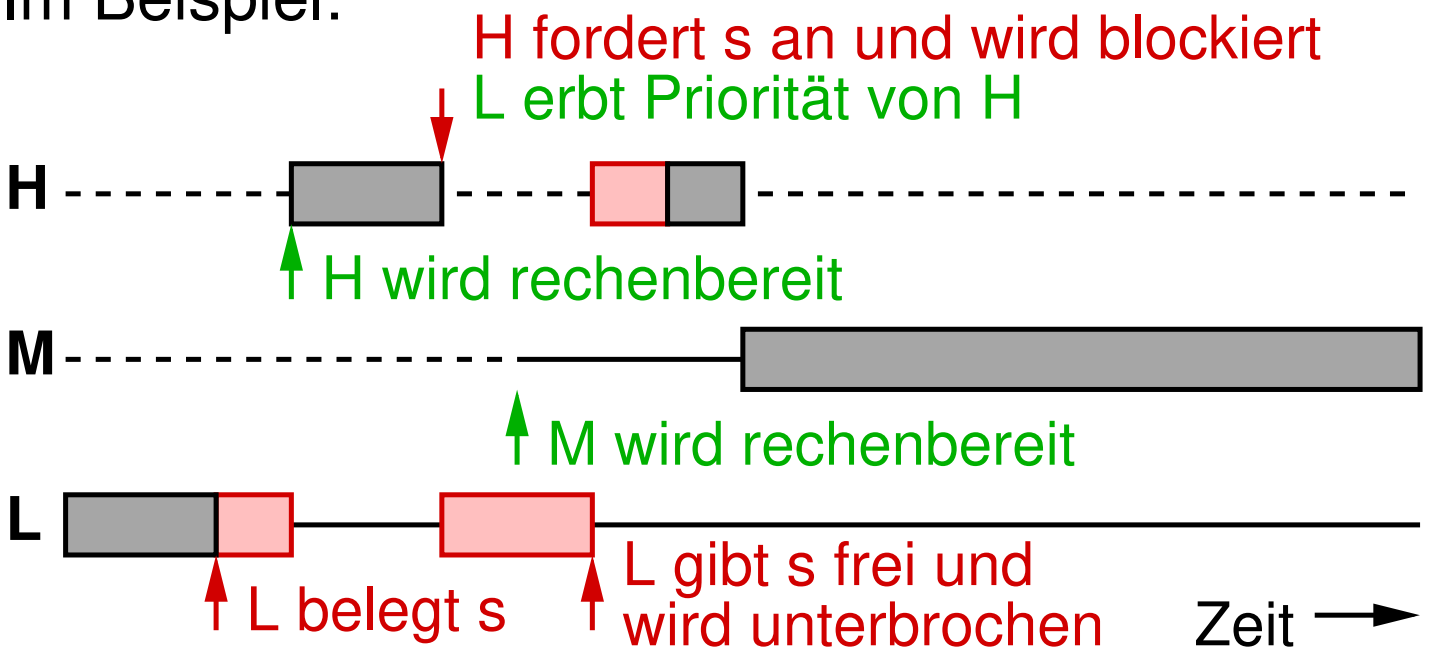
Prioritätsinversion

- Problem bei wechselseitigem Ausschluß:
  - hoch priorisierter Thread **H** muss ggf. warten, bis ein Thread **L** mit niedriger Priorität ein Semaphor **s** freigibt
  - Thread **M** mit mittlerer Priorität kann **L** (und damit auch **H**) beliebig lange aufhalten
- Beispiel:



Prioritätsinversion ...

- Lösung: Prioritätsvererbung
  - Wenn ein hoch priorisierter Thread durch einen niedriger priorisierten Thread blockiert ist, erbt dieser für die Dauer der Blockierung die Priorität des blockierten Threads
- Im Beispiel:



7.4.5 *Multilevel* Scheduling

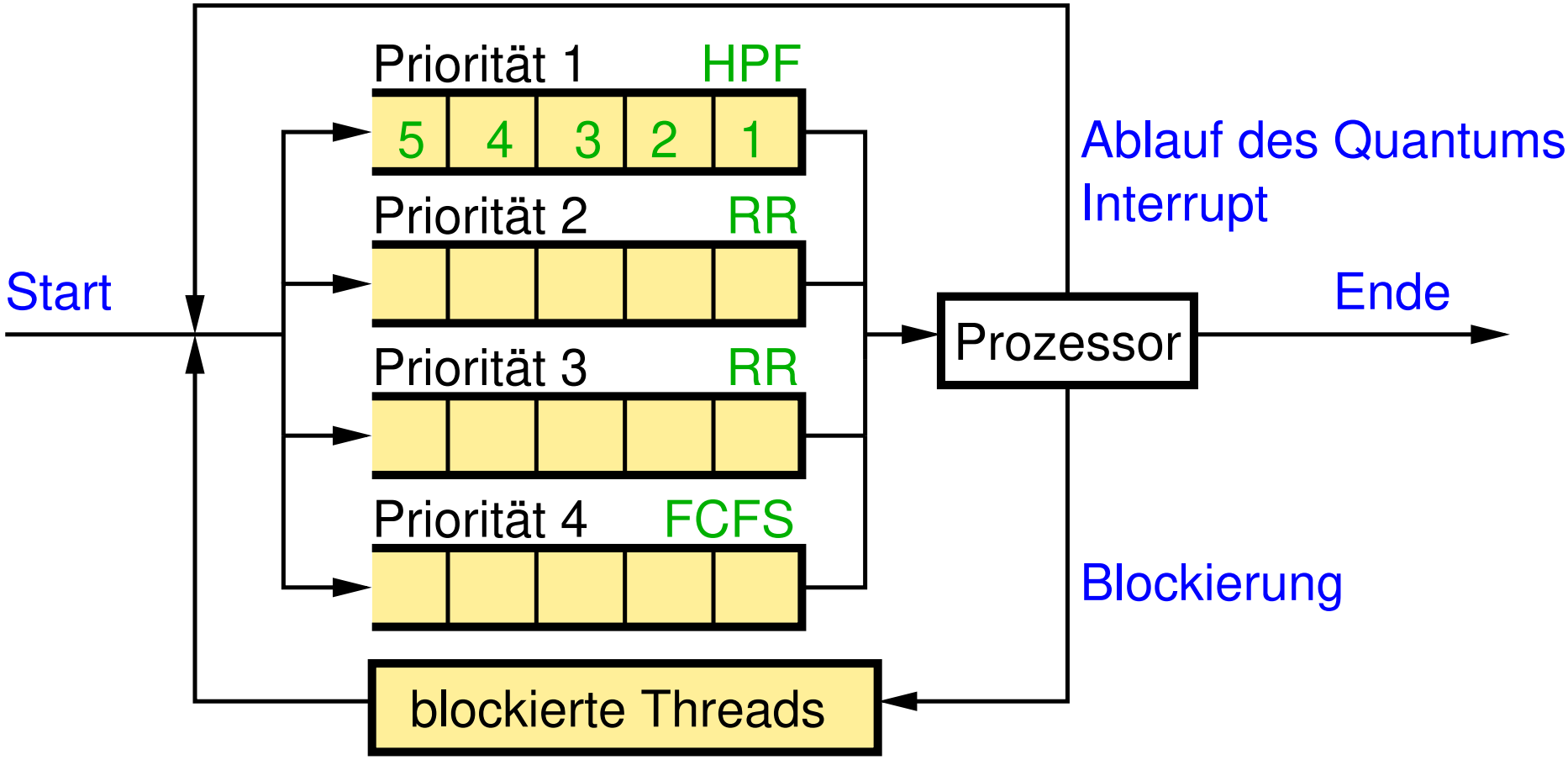
- Mehrere Warteschlangen für bereite Threads
- Jede Warteschlange kann eigene Auswahlstrategie besitzen
- Zusätzlich: Strategie zur Auswahl der **aktuellen** Warteschlange
  - z.B. Prioritäten oder Round-Robin
- Statische Verfahren:
  - Thread wird fest einer Warteschlange zugeordnet
- Dynamische Verfahren:
  - Thread kann in Abhängigkeit seines Verhaltens zwischen Warteschlangen wechseln

Beispiel: Statisches *Multilevel* Scheduling

- Unterstützung verschiedener Betriebsarten in einem System
- Threads werden in Prioritäts**klassen** eingeteilt
  - pro Prioritätsklasse eine Warteschlange
  - unterschiedliche Scheduling-Strategien innerhalb der Warteschlangen

Priorität	Threadklasse	Strategie
1	Echtzeitthreads (Multimedia)	Prioritäten
2	Interaktive Threads	RR
3	E/A-intensive Threads	RR
4	Rechenintensive Stapel-Jobs	FCFS

Beispiel: Statisches *Multilevel* Scheduling ...

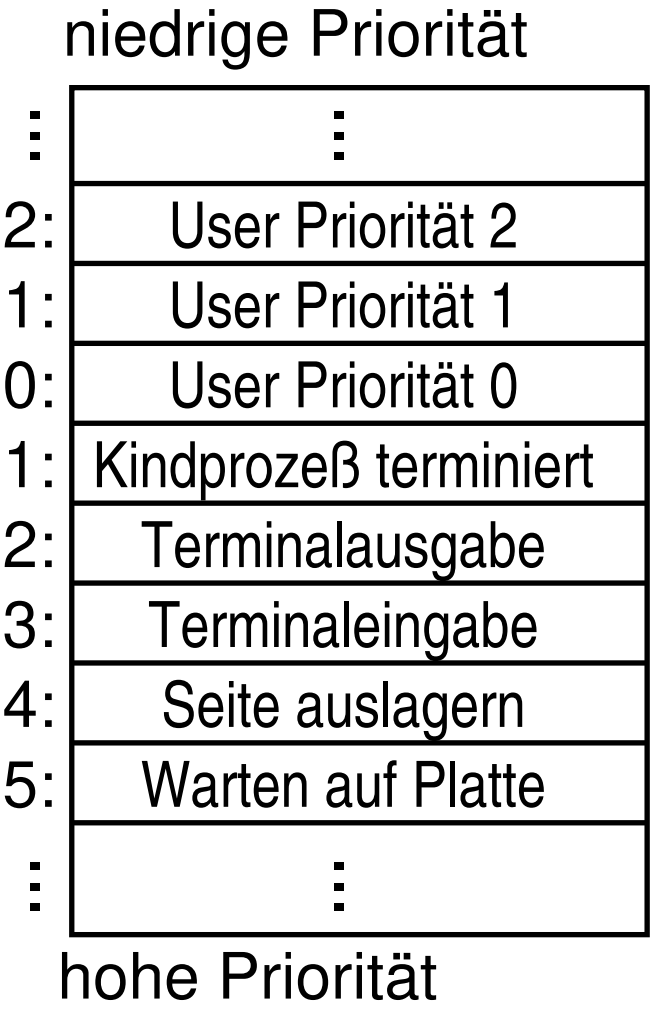


Beispiel: *Multilevel Feedback* Scheduling

- Dynamische Prioritäten und variable Zeitscheibenlänge
- Mehrere Warteschlangen mit unterschiedlicher Priorität
  - innerhalb einer Warteschlange: *Round Robin*
  - bei niedrigerer Priorität: längeres Quantum
- Falls Thread Quantum aufbraucht: erniedrigen der Priorität
  - CPU-lastiger Thread erhält längeres Quantum, wird seltener unterbrochen
- Sonst: Priorität nicht verändern bzw. wieder erhöhen
  - E/A-lastiger (bzw. interaktiver) Thread erhält bevorzugt die CPU, aber nur für kurze Zeit

7.5.1 Scheduling in BSD-Unix

- Priorisierte Warteschlangen mit *Round Robin*
- Threads, die durch BS-Aufruf im BS-Kern blockiert sind, erhalten hohe (negative) Priorität
  - nach Ende der Blockierung: Thread erhält bevorzugt die CPU, Priorität wird dann wieder auf alten Wert gesetzt
- Anpassung der normalen User-Priorität je nach CPU-Nutzung (Aging-Verfahren)



7.5.2 Scheduling in Windows NT / Vista

- 32 Prioritätsklassen
  - Priorität 16-31 (höchste)
    - Echtzeitklasse, statische Priorität
  - Priorität 1-15
    - normale Threads, dynamische Priorität
    - starke Prioritätserhöhung beim Warten auf Benutzereingabe, moderatere Erhöhung beim Warten auf E/A,
      - danach schrittweise Reduktion zum Ausgangswert
  - Priorität 0 (niedrigste)
    - *Idle*-Thread
- Innerhalb einer Klasse: *Round-Robin*

7.5.3 Scheduling in Linux (O(1) Scheduler)

- Jeder Thread wird in Linux einer Scheduling-Klasse zugeordnet:
  - `SCHED_FIFO`: „Echtzeit“-Threads mit FIFO Scheduling
  - `SCHED_RR`: „Echtzeit“-Threads mit Round Robin Scheduling
  - `SCHED_OTHER`: normale Threads
- „Echtzeit“-Threads besitzen lediglich höhere Priorität (0-99)
  - Priorität normaler Threads: 100-139
- Für jede Priorität: je zwei Warteschlangen pro CPU
  - *active*: Threads, die Zeitscheibe noch nicht aufgebraucht haben
  - *expired*: Threads, deren Quantum abgelaufen ist
- Zusätzliches Bit-Set zeigt an, welche Warteschlangen nicht leer sind

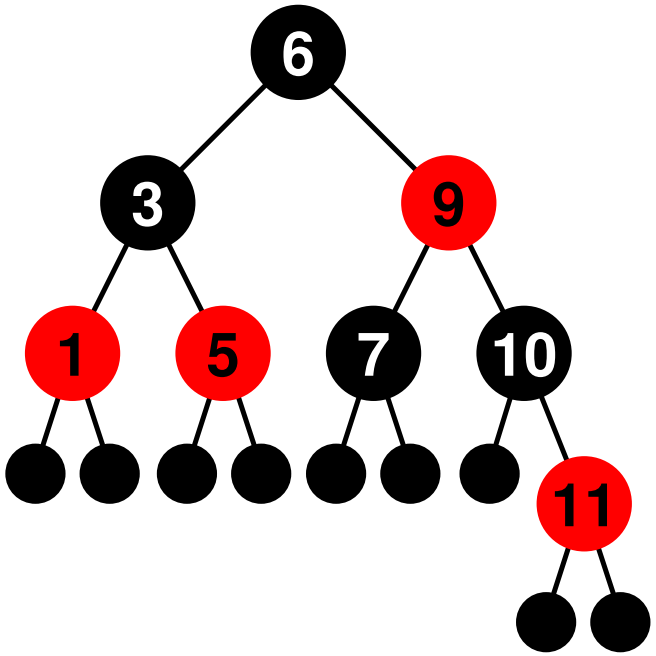
Vorgehensweise

- Scheduler wählt Thread mit höchster Priorität in *active* Warteschlange
  - falls Quantum aufgebraucht wird: Einreihen in *expired* Warteschlange
  - falls Thread blockiert: nach Blockierung wieder in *active* Warteschlange (mit entsprechend reduziertem Quantum)
- Falls *active* Warteschlange leer: Tausch der Warteschlangen
- Prioritäten der normalen Threads werden durch komplexe Heuristiken angepasst
  - interaktive (E/A-lastige) Threads sollen höhere Priorität erhalten

7.5.4 Linux: *Completely Fair Scheduler* (CFS)

- Ziel: jeder Thread bekommt einen fairen Anteil der CPU-Leistung
  - ohne aufwendige und problematische Heuristiken
- CFS simuliert eine ideale Multitasking-CPU
  - bei n Threads erhält jeder den Anteil $1/n$ der CPU-Leistung
- Dazu: die bisher erhaltene CPU-Zeit eines Threads wird ns-genau erfasst (*virtual runtime*)
  - neu erzeugte Threads erhalten als *virtual runtime* den Mittelwert aller anderen Threads
- Rechenbereite Threads werden nach *virtual runtime* sortiert in einen balancierten Baum (*Red-Black-Tree*) eingetragen
- Der Thread mit der kleinsten *virtual runtime* erhält die CPU

Red-Black-Tree



- Balancierter, binärer Suchbaum
  - Knoten sind entweder rot oder schwarz eingefärbt
  - jeder interne Knoten hat zwei Kinder
  - interne Knoten tragen Informationen
- Forderungen:
  - Wurzel und Blätter sind schwarz
  - rote Knoten haben zwei schwarze Kinder
  - jeder Pfad von einem Knoten zu einem Blatt hat dieselbe Zahl schwarzer Knoten
- Damit Baumhöhe bei n inneren Knoten: $\mathcal{O}(\log n)$
  - Einfügen / Suchen / Löschen in $\mathcal{O}(\log n)$

Details

- Länge der Zeitscheibe:
  - Systemparameter *latency*: Zeit, bis jeder Thread einmal „drankommt“
  - Quantum = *latency* / N, wobei N = Zahl der rechenbereiten Threads
    - ggf. Sonderbehandlung, falls N zu groß
- Behandlung der Prioritäten (*nice*-Werte):
  - Quantum wird entsprechend der Priorität gewichtet
- Gruppen-Scheduling
  - erlaubt Definition von Gruppen von Threads
  - jede Gruppe erhält dann denselben Anteil der CPU-Leistung

Multi-Core Scheduling

- Jeder Core hat eigene Thread-Warteschlangen (bzw. RB-Tree)
  - damit: Skalierbarkeit, keine Synchronisation notwendig
- Warteschlangen müssen zwischen Cores balanciert sein
  - z.B.: nicht alle hochprioren Threads bei einem Core
- Lastbalancierung ist aber teuer (Synchronisation, Caches)
  - seltene Durchführung; gute Verfahren notwendig
- Last berücksichtigt Prioritäten und durchschnittliche CPU-Nutzung
- Lastverteilung erfolgt hierarchisch
  - wegen hierarchischer Cache-/Speicherstruktur

7.6 Zusammenfassung / Wiederholung

- Scheduling
  - Entscheidung welcher Thread wann wie lange (und ggf. auch welcher CPU) rechnen darf
  - Unterschiedliche Anforderungen, je nach Sichtweise und Betriebsmodus
  - Nicht-präemptives und präemptives Scheduling
    - präemptiv: BS kann einem Thread die CPU zwangsweise entziehen
- Scheduling-Algorithmen
  - FCFS: FIFO-Warteschlange rechenbereiter Threads, nicht-präemptiv
  - SJF: *Shortest Job First*
    - optimiert Durchlaufzeit von Jobs

7.6 Zusammenfassung / Wiederholung ...

- Scheduling-Algorithmen ...
  - *Round Robin* (RR): präemptive Version von FCFS
    - Threads dürfen nur bestimmte Zeit rechnen
  - Prioritätenbasiertes Scheduling:
    - nur der Thread mit höchster Priorität bekommt CPU (bzw. die n höchstprioren Threads bei n CPUs)
  - *Multilevel* Scheduling
    - mehrere Warteschlangen mit unterschiedlicher Auswahlstrategie
    - statisches *Multilevel* Scheduling
      - feste Zuordnung Thread $\rightarrow$ Warteschlange
    - *Multilevel Feedback* Scheduling
      - dynamische Zuordnung Thread $\rightarrow$ Warteschlange

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

8 Speicherverwaltung

Inhalt:

- Einführung, Grundlagen
- *Swapping* und dynamische Speicherverwaltung
- Seitenbasierte Speicherverwaltung (*Paging*)
  - Adreßumsetzung
  - Dynamische Seitenersetzung
  - Seitenersetzungsalgorithmen

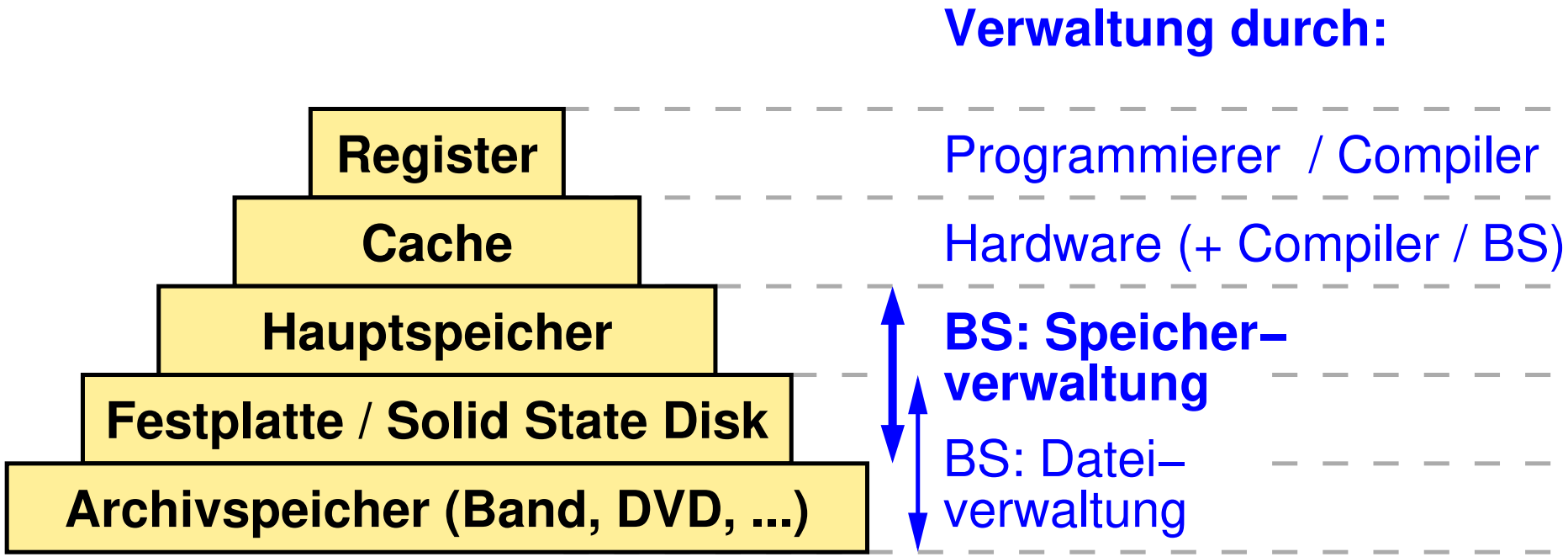
- Tanenbaum 4
- Stallings 7, 8
- Nehmer/Sturm 4

Wichtige Unterscheidung / Begriffe

- **Adreßraum**: Menge aller verwendbaren Adressen
  - **logischer (virtueller) Adreßraum**: aus Sicht eines Programms bzw. Prozesses gesehen
    - Adressen, die der Prozess* verwenden kann
  - **physischer Adreßraum**: aus Sicht der Hardware
    - Adressen, die die Speicher-Hardware verwendet / verwenden kann
  - nicht immer identisch! (siehe später: *Paging*)
- **Speicher**: das Stück Hardware, das Daten speichert ...
  - (oft: Speicher als Synonym für physischer Adreßraum)

* genauer: die Threads des Prozesses

Erinnerung: Speicherhierarchie

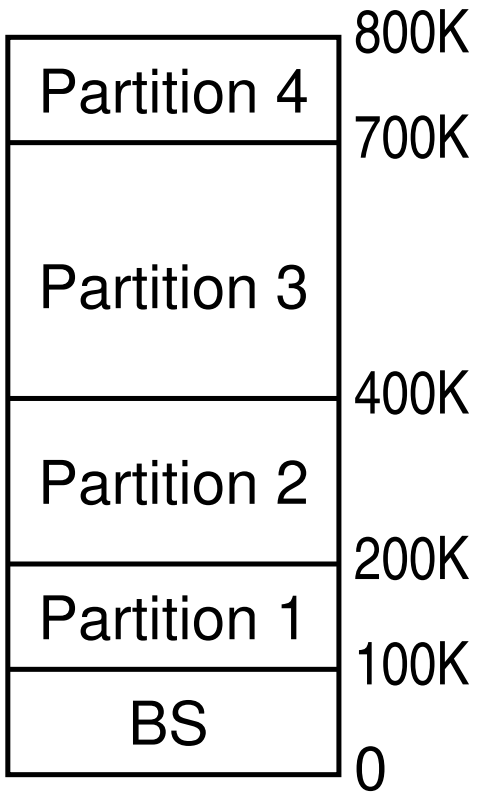


Aufgaben der Speicherverwaltung im BS

- Finden und Zuteilung freier Speicherbereiche
  - z.B. bei Erzeugung eines neuen Prozesses
  - oder bei Speicheranforderung durch existierenden Prozess
- Effiziente Nutzung des Speichers: verschiedene Aspekte
  - gesamter freier Speicher sollte für Prozesse nutzbar sein
  - Prozeß benötigt nicht immer alle Teile seines Adreßraums
  - Adreßräume aller Prozesse evtl. größer als verfügbarer Hauptspeicher
- **Speicherschutz**
  - Threads eines Prozesses sollten nur auf Daten dieses Prozesses zugreifen können

Mehrprogrammbetrieb mit festen Partitionen

- Einfachste Speicherverwaltung
- Hauptspeicher in Partitionen fester Größe eingeteilt
  - Festlegung bei Systemstart
  - Anzahl legt Multiprogramming-Grad fest
    - damit auch erreichbare CPU-Auslastung
  - unterschiedlich grosse Partitionen möglich
  - eine Partition für BS



- Bei Ankunft eines Auftrags (d.h. Start eines Programms)
  - Einfügen in Warteschlange für Speicherzuteilung
    - pro Partition oder gemeinsam

Relokation und Speicherschutz

- Hintergrund: Programmcode enthält Adressen
  - z.B. in Unterprogrammaufrufen oder Ladebefehlen
- Ohne Hardware-Unterstützung:
  - beim Laden des Programms in eine Partition:
    - Adressen im Code müssen durch das BS angepaßt werden (**Relokation**)
    - möglich durch Liste in Programmdatei, die angibt, wo sich Adressen im Code befinden
  - durch Relokation beim Laden kann kein Speicherschutz erreicht werden
    - Threads des Prozesses können auf Partitionen anderer Prozesse zugreifen

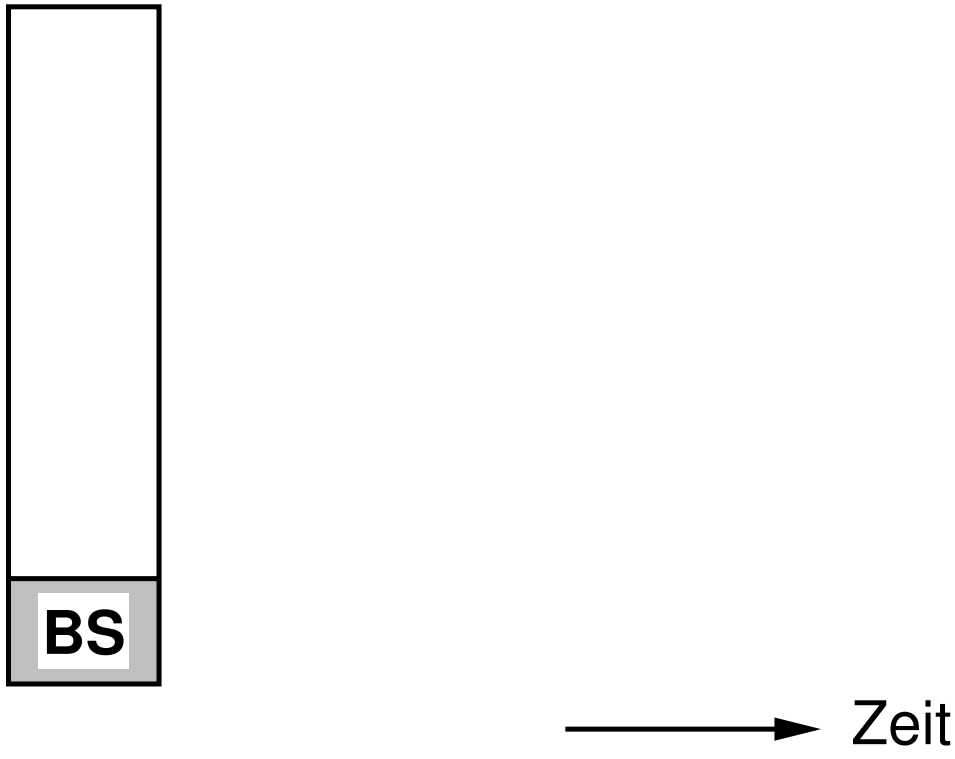
Relokation und Speicherschutz ...

- Einfachste Unterstützung durch Hardware:
  - zwei spezielle, privilegierte Prozessorregister:
    - **Basisregister**: Anfangsadresse der Partition
    - **Grenzregister**: Länge der Partition
    - werden vom BS beim Prozeßwechsel mit Werten des aktuellen Prozesses geladen
  - bei jedem Speicherzugriff (in Hardware):
    - vergleiche (logische) Adresse mit Grenzregister
      - falls größer/gleich: Ausnahme auslösen
    - addiere Basisregister zur (logischen) Adresse
      - Ergebnis: (physische) Speicheradresse

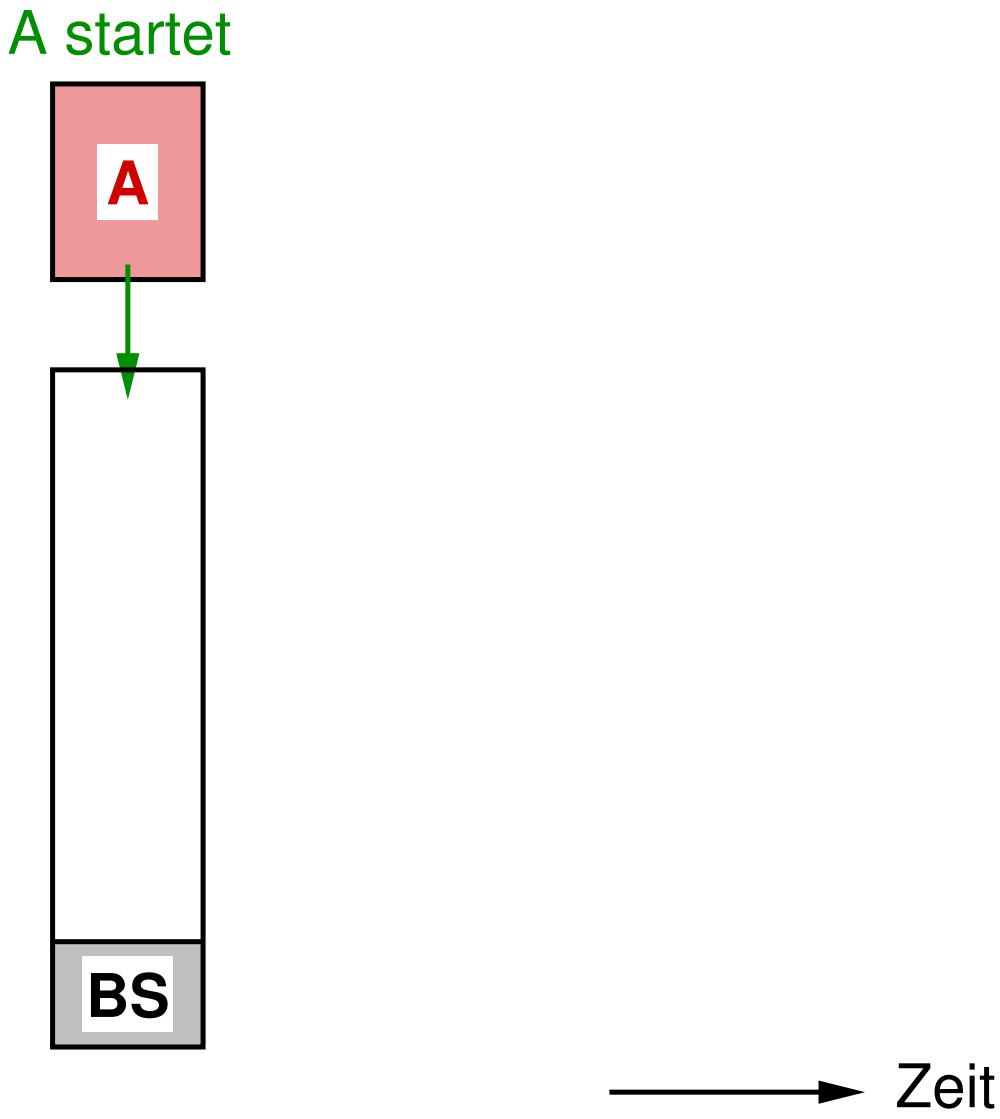
8.2 *Swapping* und dyn. Speicherverwaltung

- Feste Partitionen nur für Stapelverarbeitung geeignet
- Bei interaktiven Systemen:
  - alle aktiven Programme zusammen benötigen evtl. mehr Partitionen bzw. Speicher als vorhanden
- Lösung:
  - dynamische Partitionen
  - Partitionen ggf. temporär auf Festplatte auslagern
- Einfachste Variante: ***Swapping***
  - kompletter Prozeßadreßraum wird ausgelagert und Prozeß suspendiert
  - Wieder-Einlagern evtl. an anderer Stelle im Speicher
    - Relokation i.d.R. über Basisregister
- (Bessere Variante: ***Paging***, ☞ **8.3**)

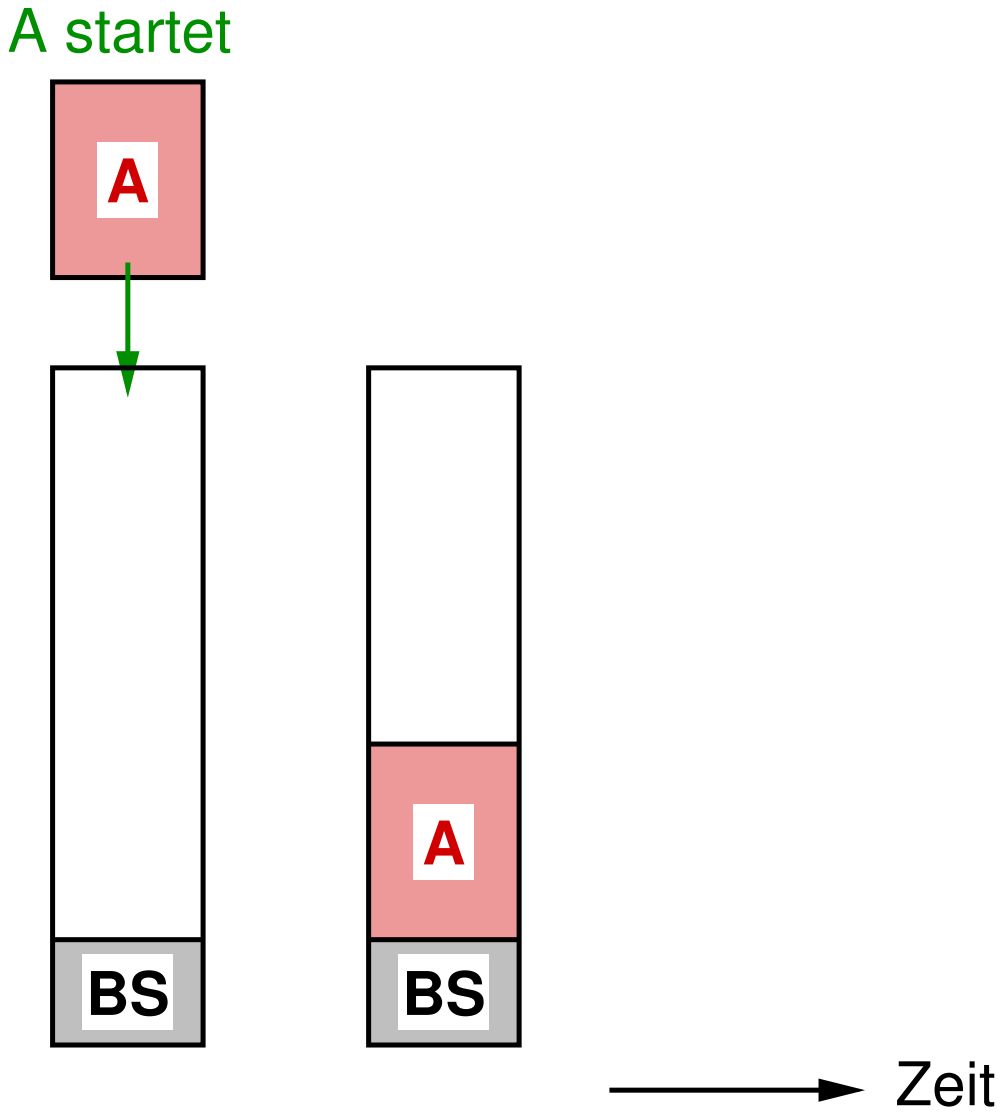
Beispiel



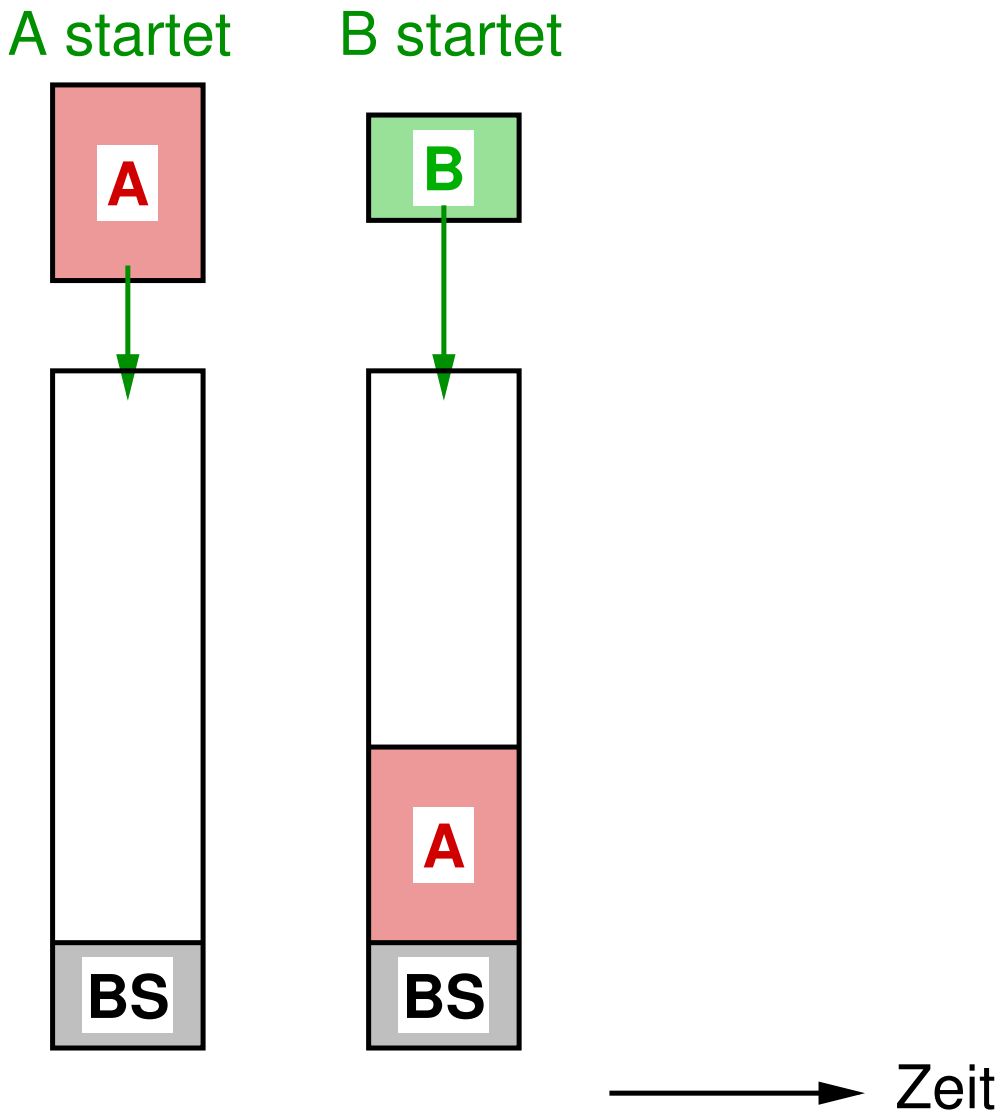
Beispiel



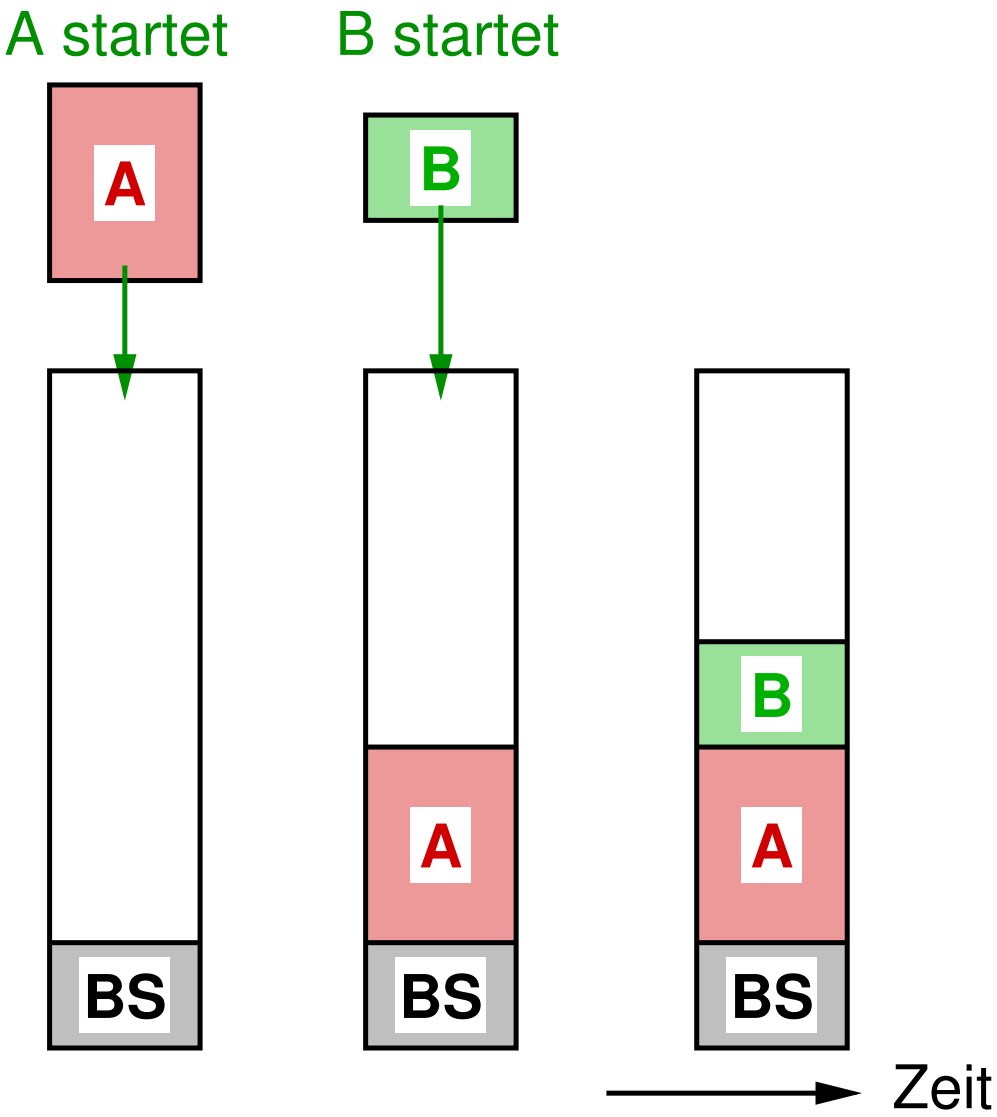
Beispiel



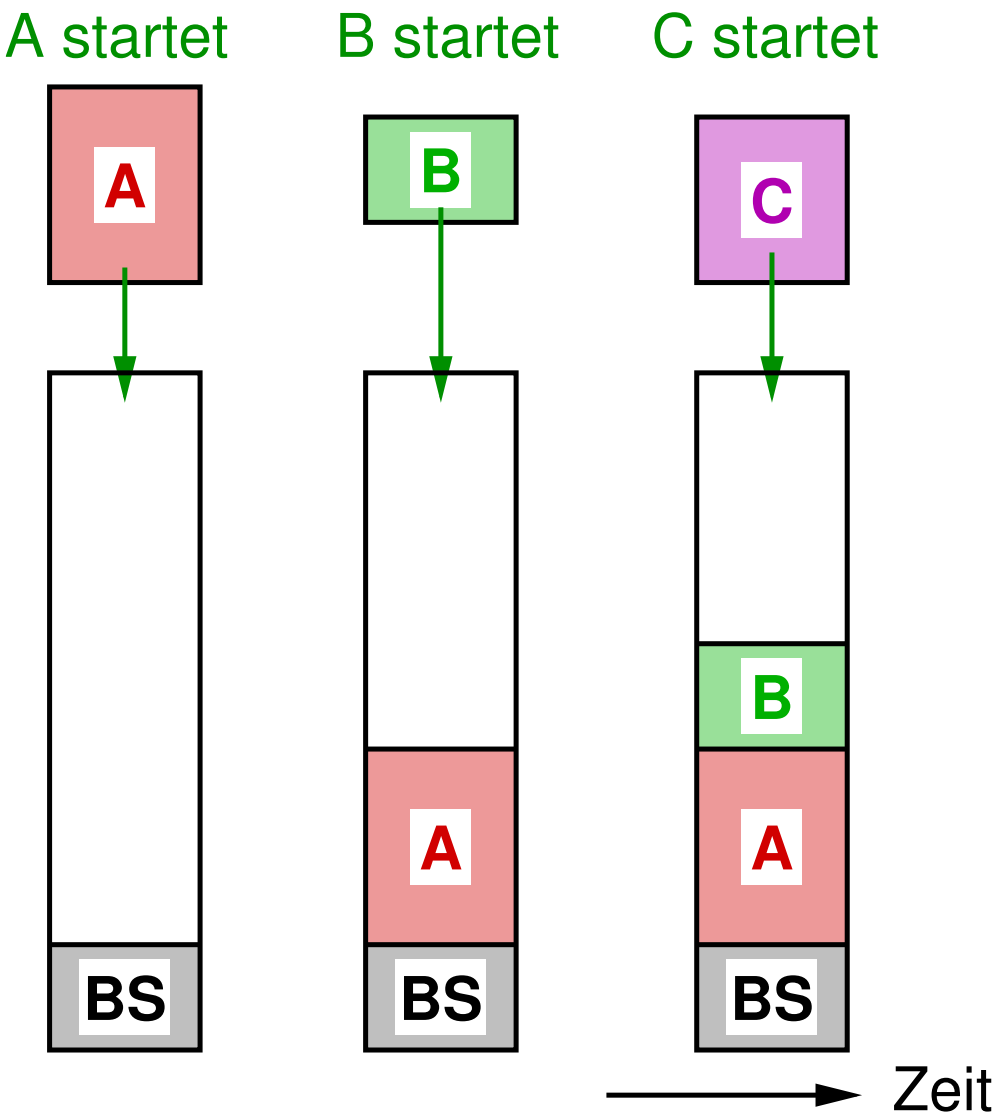
Beispiel



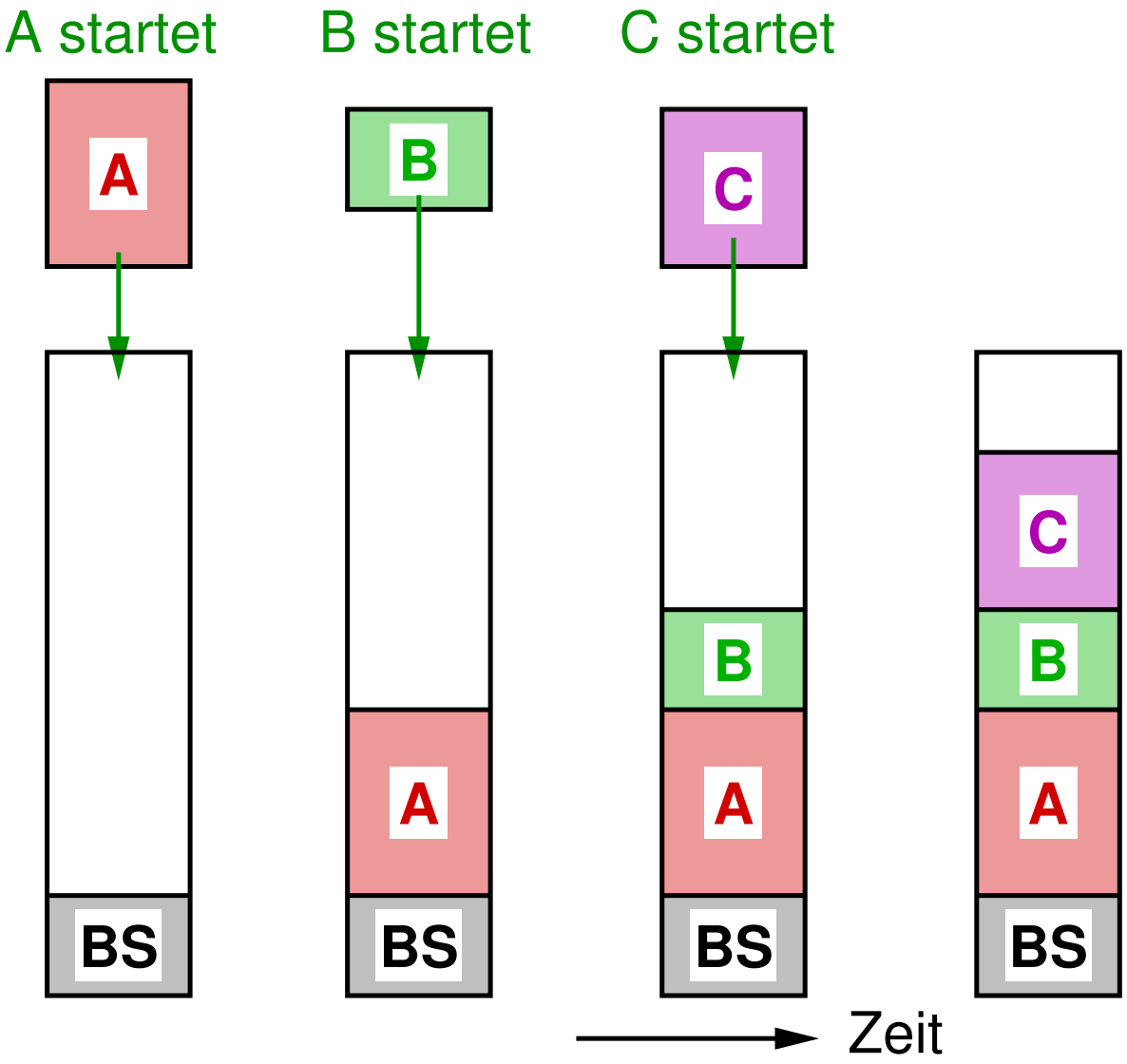
Beispiel



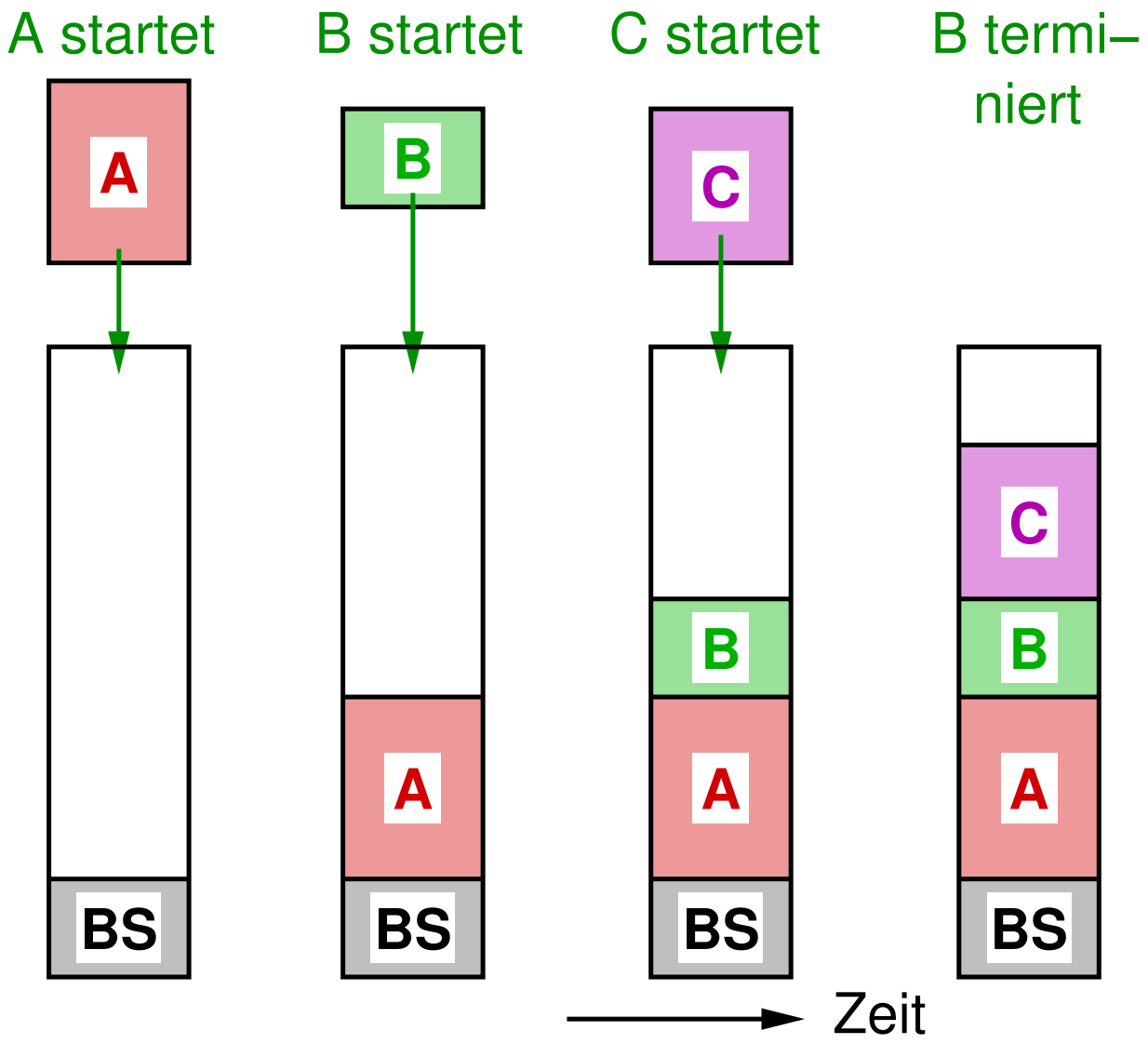
Beispiel



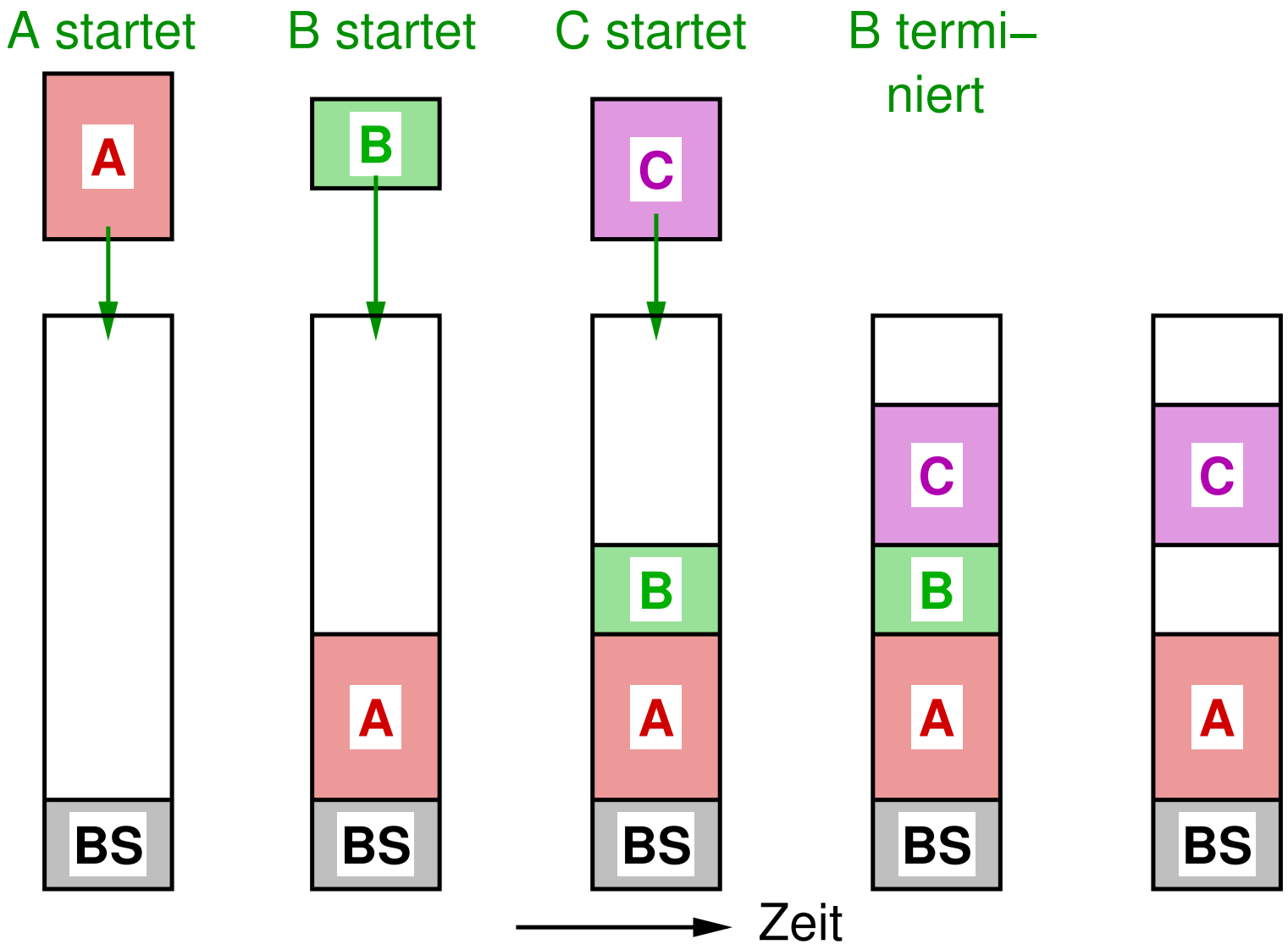
Beispiel



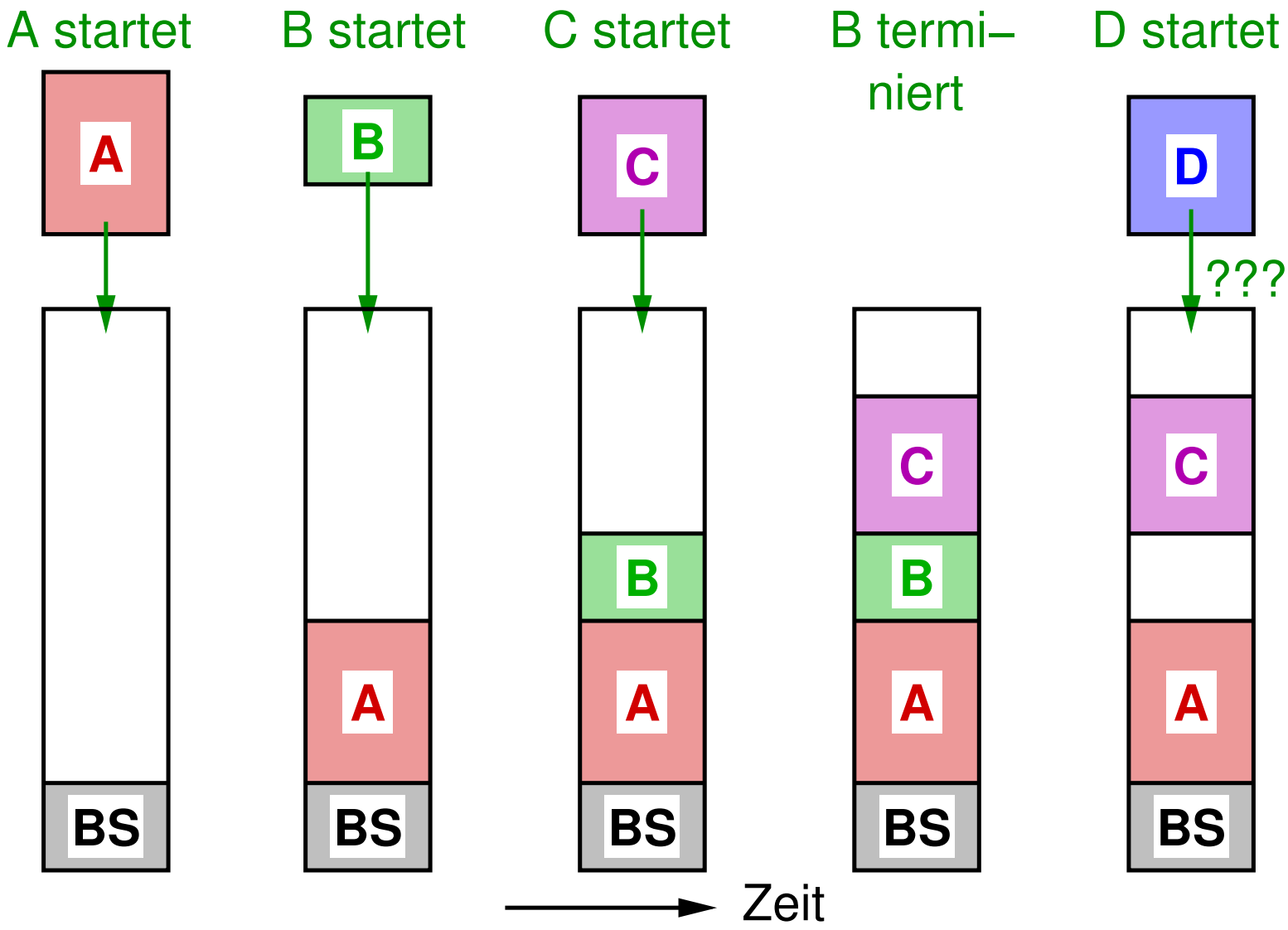
Beispiel


A startet
B startet
C startet
B termi–
niert
A
B
C
BS
A
B
BS
A
B
C
BS
A
B
C
BS
Zeit

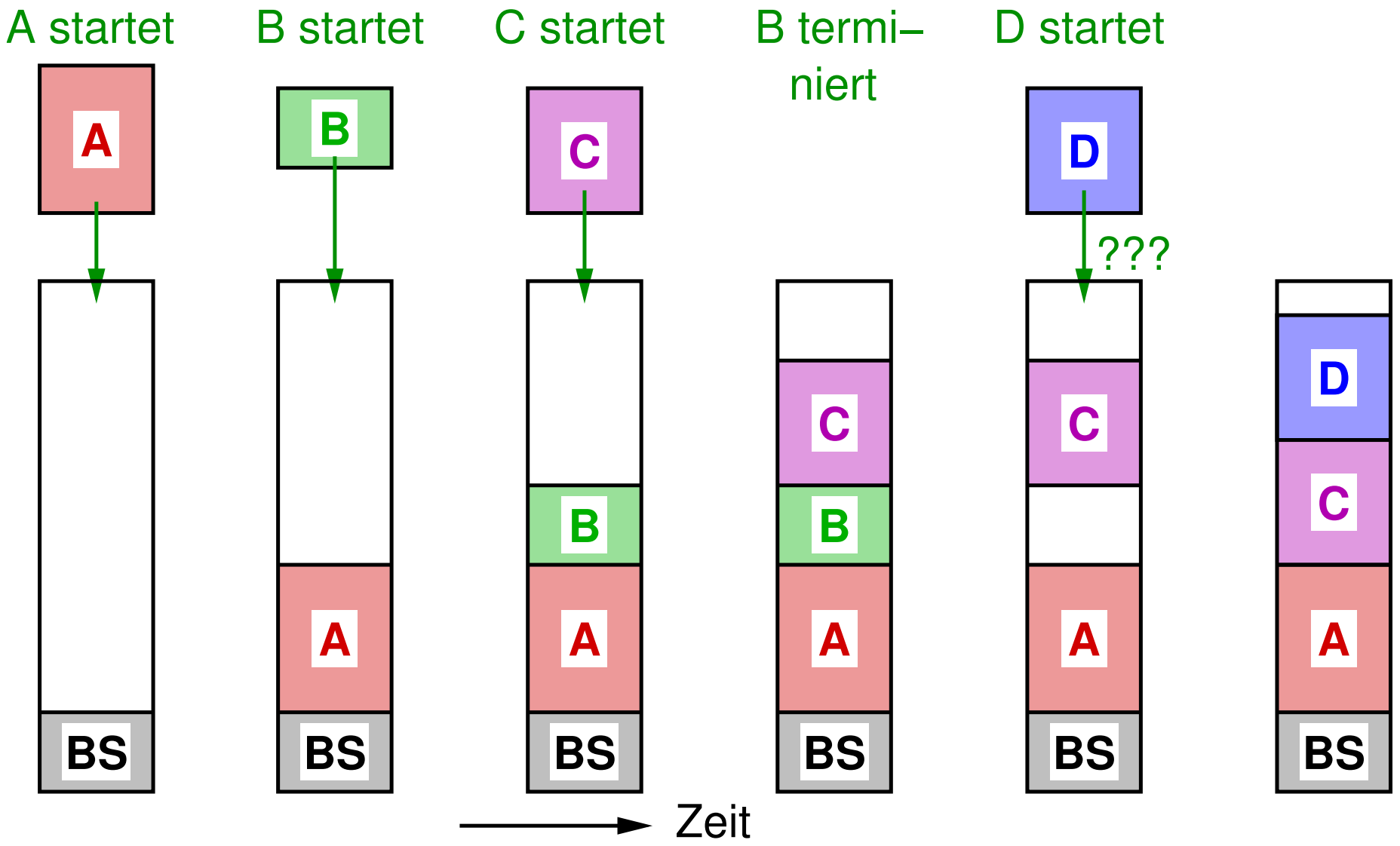
Beispiel



Beispiel


A startet
B startet
C startet
B termi–
niert
D startet
A
B
C
D
???
BS
Zeit

Beispiel

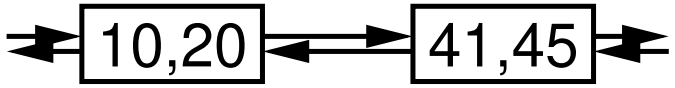

A startet
B startet
C startet
B termi–
niert
D startet
???
A
B
C
D
BS
Zeit

Diskussion

- Unterschied zu festen Partitionen:
  - Zahl, Größe und Ort der Partitionen variabel
- Löcher im Speicher können im Prinzip durch Verschieben zusammengefasst werden
- Probleme:
  - Ein- und Auslagern ist sehr zeitaufwendig
  - Prozeß kann zur Laufzeit mehr Speicher anfordern
    - benötigt evtl. Verschiebung oder Auslagerung von Prozessen
- Schlechte Speichernutzung: Prozeß benötigt i.d.R. nicht immer seinen ganzen Adreßraum

8.2.2 Dynamische Speicherverwaltung

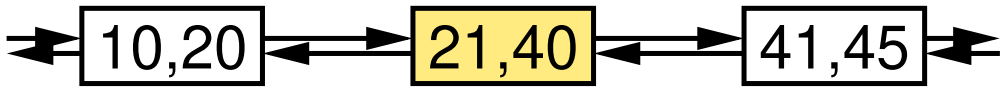
- BS muß bei Prozeßerzeugung oder Einlagerung einen passenden Speicherbereich finden
- Dazu: BS benötigt Information über freie Speicherbereiche
- Typisch: Liste aller freien Speicherbereiche
  - Listenelemente jeweils im freien Bereich gespeichert
- Wichtig bei Speicherfreigabe: Verschmelzen der Einträge für aneinander grenzende freie Speicherbereiche
  - dazu: doppelt verkettete Liste, sortiert nach Adressen



- Anmerkung: Problem/Lösungen identisch zur Verwaltung des Heaps durch das Laufzeitsystem

8.2.2 Dynamische Speicherverwaltung

- BS muß bei Prozeßerzeugung oder Einlagerung einen passenden Speicherbereich finden
- Dazu: BS benötigt Information über freie Speicherbereiche
- Typisch: Liste aller freien Speicherbereiche
  - Listenelemente jeweils im freien Bereich gespeichert
- Wichtig bei Speicherfreigabe: Verschmelzen der Einträge für aneinander grenzende freie Speicherbereiche
  - dazu: doppelt verkettete Liste, sortiert nach Adressen



- Anmerkung: Problem/Lösungen identisch zur Verwaltung des Heaps durch das Laufzeitsystem

8.2.2 Dynamische Speicherverwaltung

- BS muß bei Prozeßerzeugung oder Einlagerung einen passenden Speicherbereich finden
- Dazu: BS benötigt Information über freie Speicherbereiche
- Typisch: Liste aller freien Speicherbereiche
  - Listenelemente jeweils im freien Bereich gespeichert
- Wichtig bei Speicherfreigabe: Verschmelzen der Einträge für aneinander grenzende freie Speicherbereiche
  - dazu: doppelt verkettete Liste, sortiert nach Adressen

- Anmerkung: Problem/Lösungen identisch zur Verwaltung des Heaps durch das Laufzeitsystem

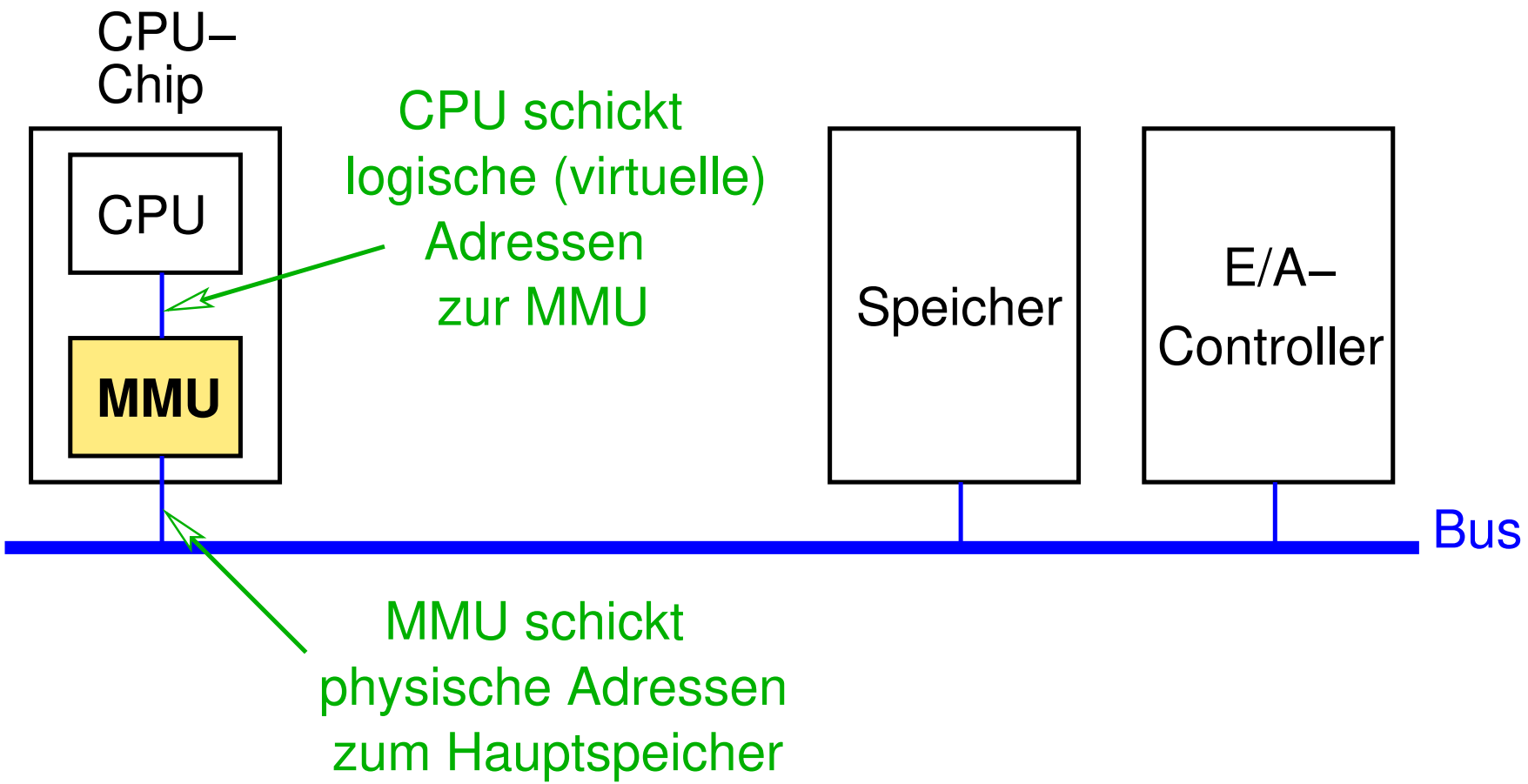
Suchverfahren

- Ziel: finde einen passenden Eintrag in der Freispeicherliste
  - benötigter Anteil wird an Prozeß zugewiesen
  - Rest wird wieder in Freispeicherliste eingetragen
    - führt zu (externer) Fragmentierung des Speichers
- ***First Fit***: verwende ersten passenden Speicherbereich
  - einfach, relativ gut
- ***Quick Fit***: verschiedene Listen für Freibereiche mit gebräuchlichen Größen
  - schnelle Suche
  - Problem: Verschmelzen freier Speicherbereiche
  - Variante **Buddy-System**: Blockgrößen sind Zweierpotenzen
    - einfaches Verschmelzen, aber interne Fragmentierung

8.3 Seitenbasierte Speicherverwaltung (*Paging*)

- Grundlage: strikte Trennung zwischen
  - **logischen Adressen**, die der Prozeß sieht / benutzt
    - auch **virtuelle Adresse** genannt
  - **physischen Adressen**, die der Hauptspeicher sieht
- Idee: **bei jedem Speicherzugriff** wird die vom Prozeß erzeugte logische Adresse auf eine physische Adresse abgebildet
  - durch Hardware: **MMU** (***Memory Management Unit***)
- Vorteile:
  - kein Verschieben beim Laden eines Prozesses erforderlich
  - auch kleine freie Speicherbereiche sind nutzbar
  - keine aufwendige Suche nach passenden Speicherbereichen
  - Speicherschutz ergibt sich (fast) automatisch
  - ermöglicht, auch Teile des Prozeßadreßraums auszulagern

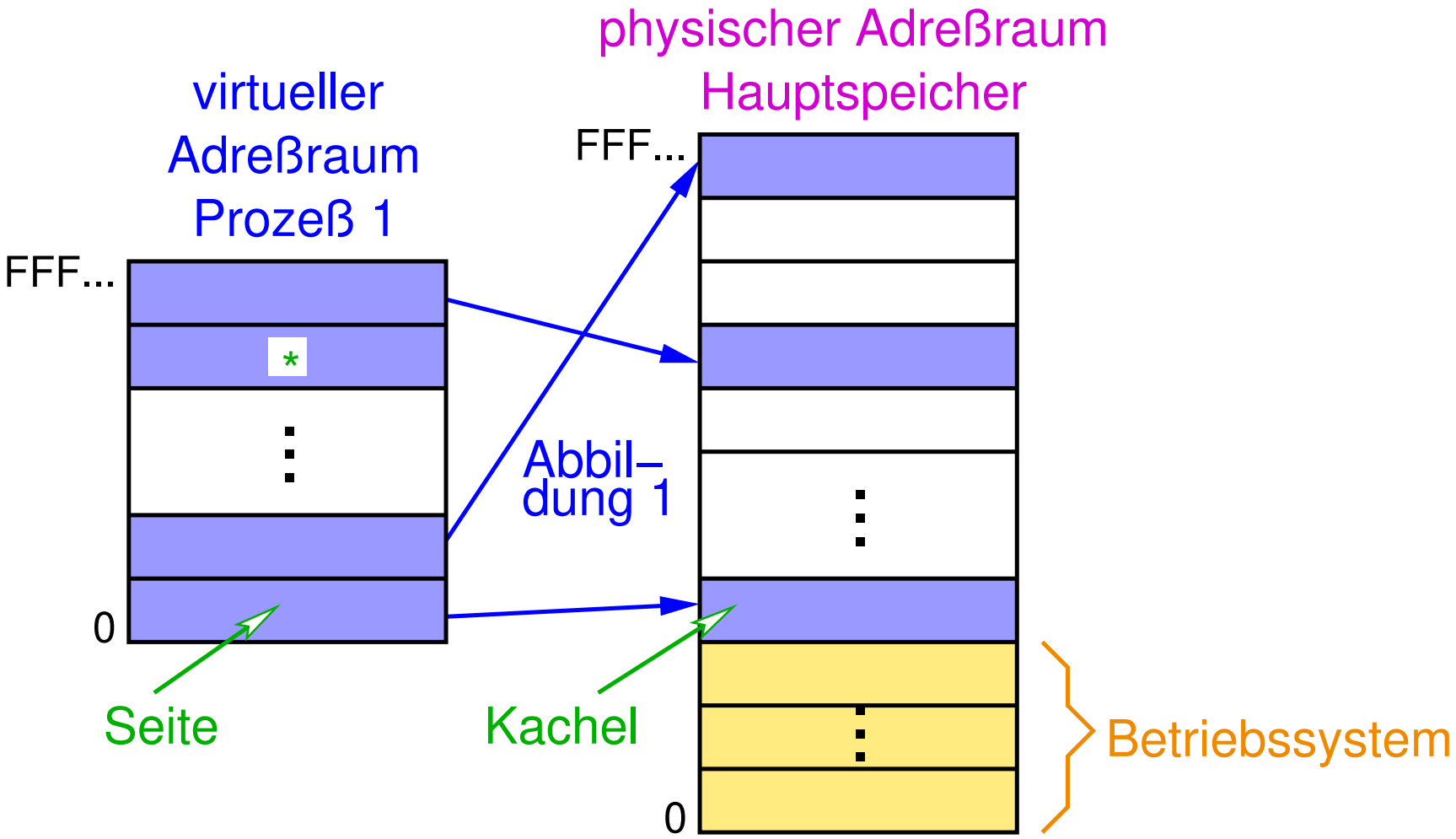
Ort und Funktion der MMU im Rechner



Seitenbasierte Speicherabbildung

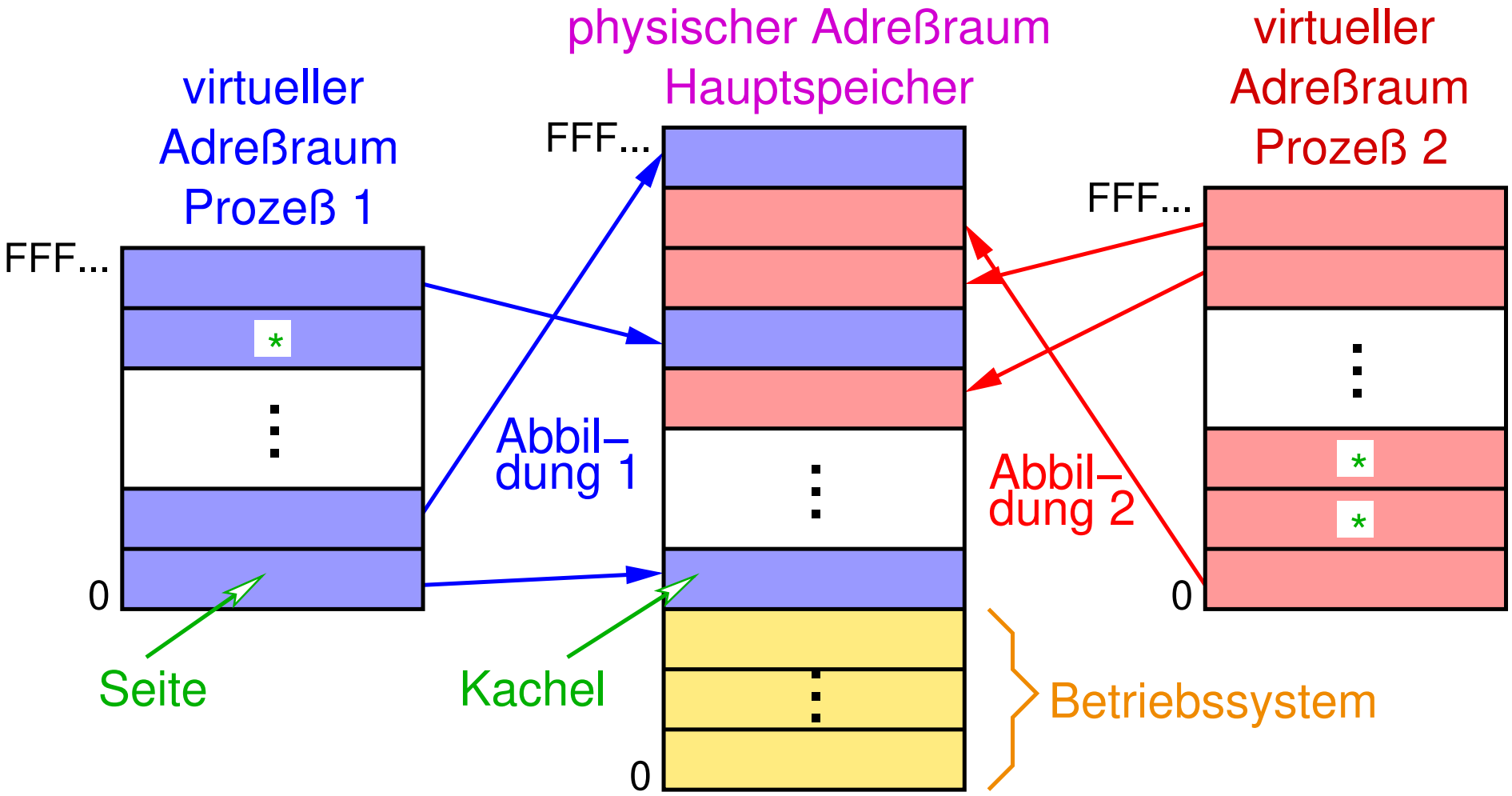
- Abbildung von virtuellen auf physische Adressen über Tabelle
  - Tabelle wird vom BS erstellt und aktualisiert
- Dazu: Aufteilung der Adreßräume in Blöcke fester Größe
  - **Seite**: Block im virtuellen Adreßraum
  - **Kachel** (**Seitenrahmen**): Block im physischen Adreßraum
  - Typische Seitengröße: 4 KiB
- Umsetzungstabelle (**Seitentabelle**) definiert für jede Seite:
  - physische Adresse der zugehörigen Kachel (falls vorhanden)
  - Zugriffsrechte

Grundidee der seitenbasierten Speicherabbildung



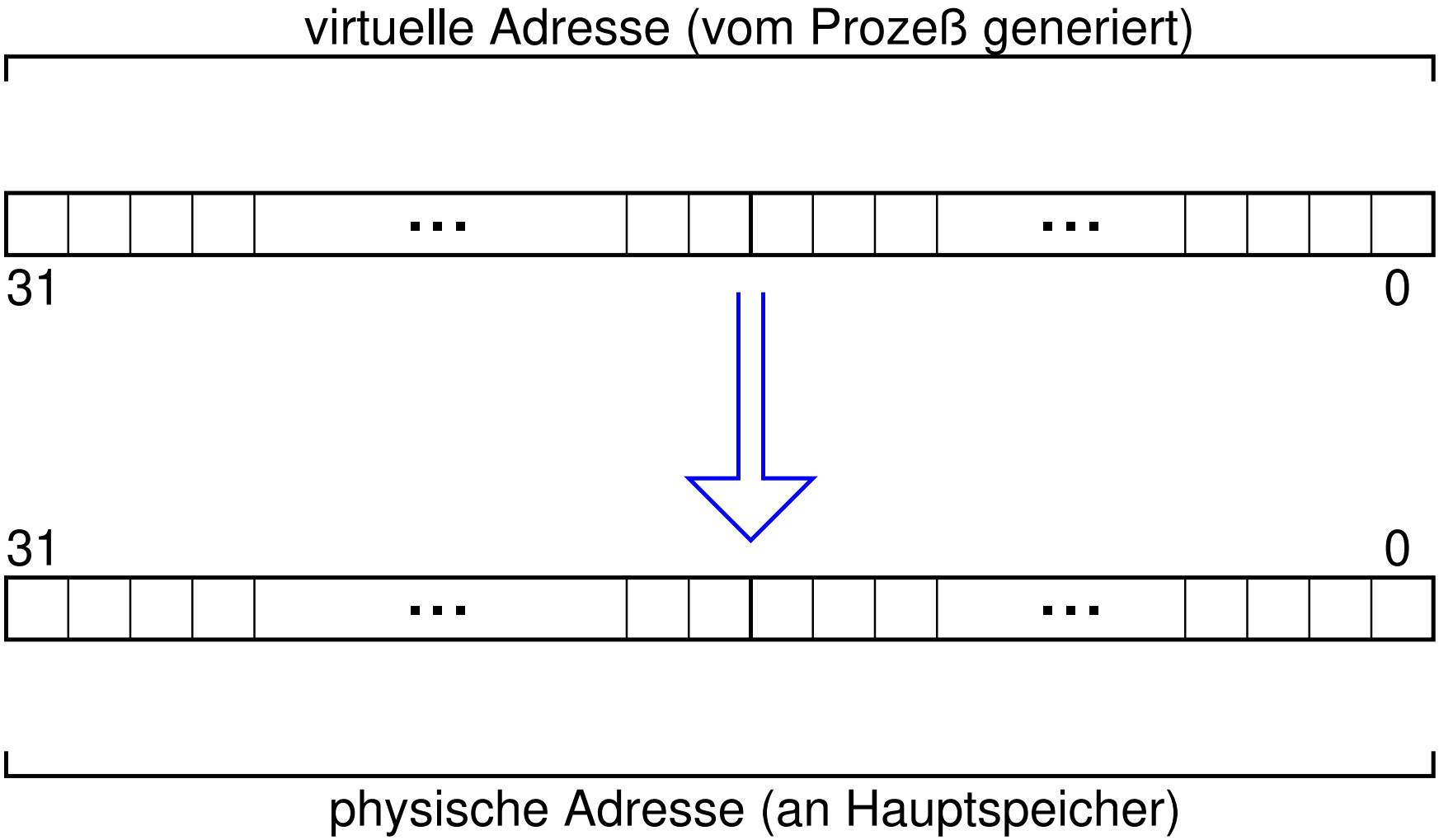
* Diese Seiten sind nicht in den Hauptspeicher abgebildet
Sie könnten z.B. auf Hintergrundspeicher ausgelagert sein.

Grundidee der seitenbasierten Speicherabbildung

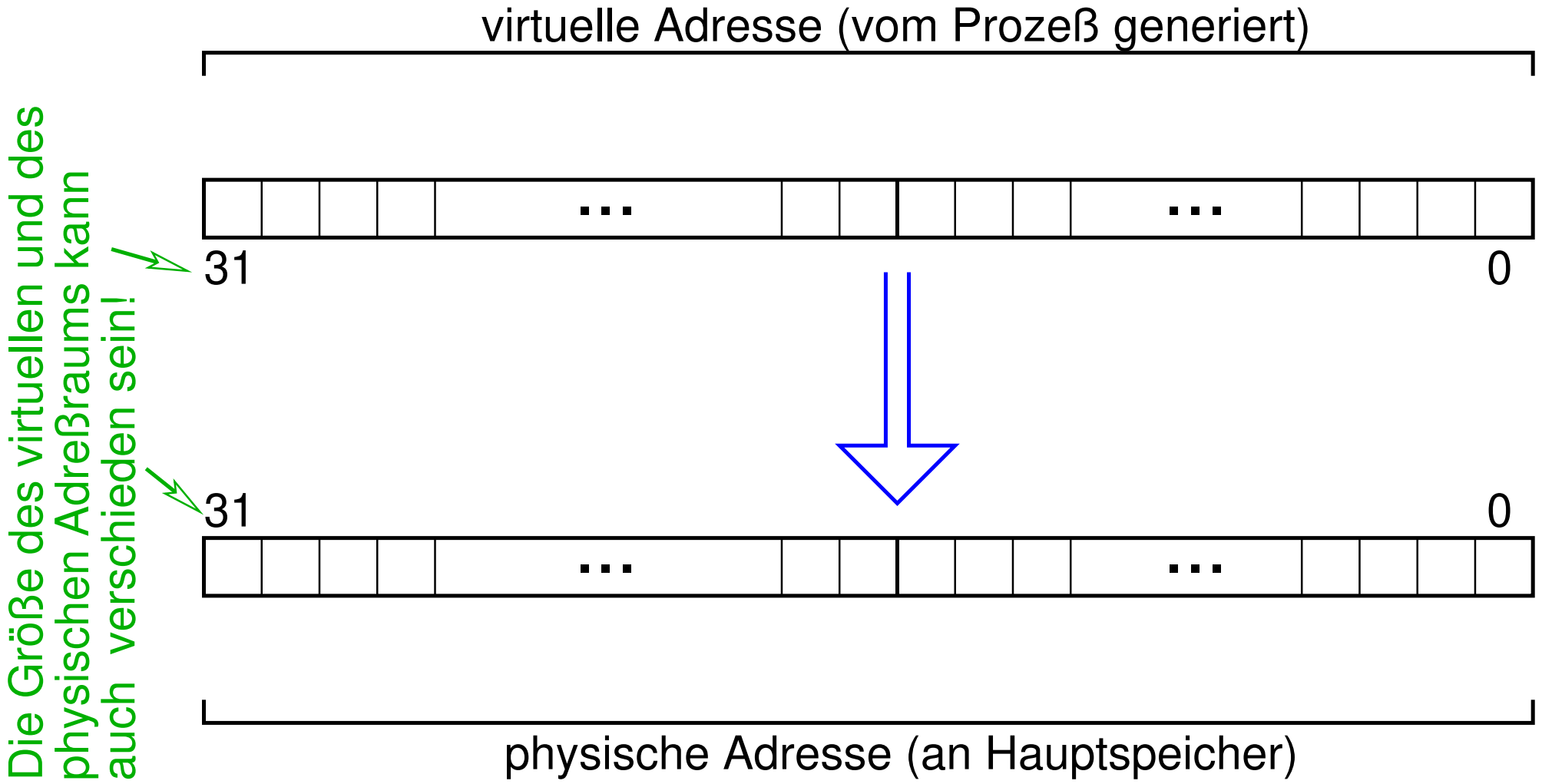


* Diese Seiten sind nicht in den Hauptspeicher abgebildet
Sie könnten z.B. auf Hintergrundspeicher ausgelagert sein.

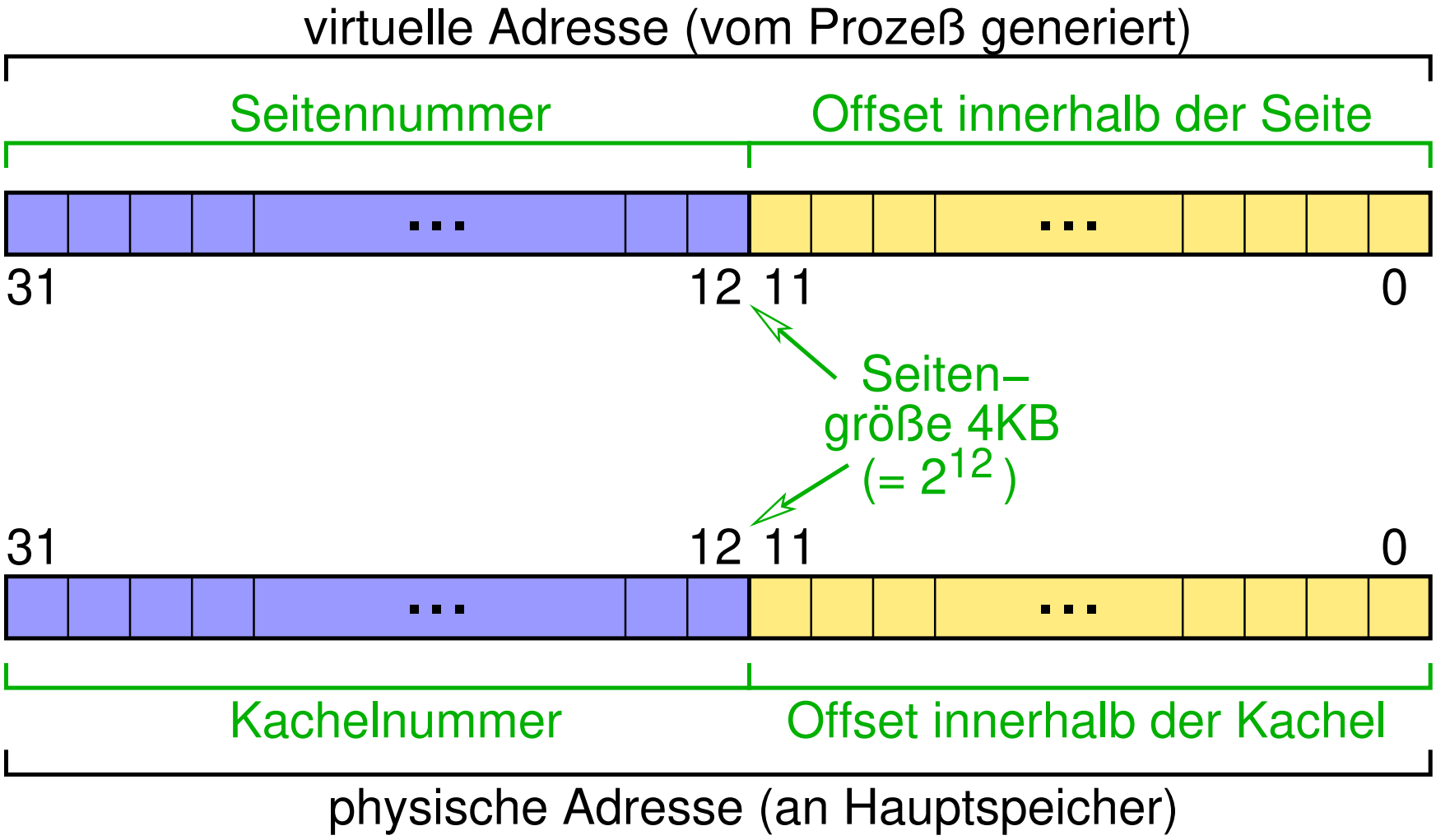
Prinzip der Adreßumsetzung



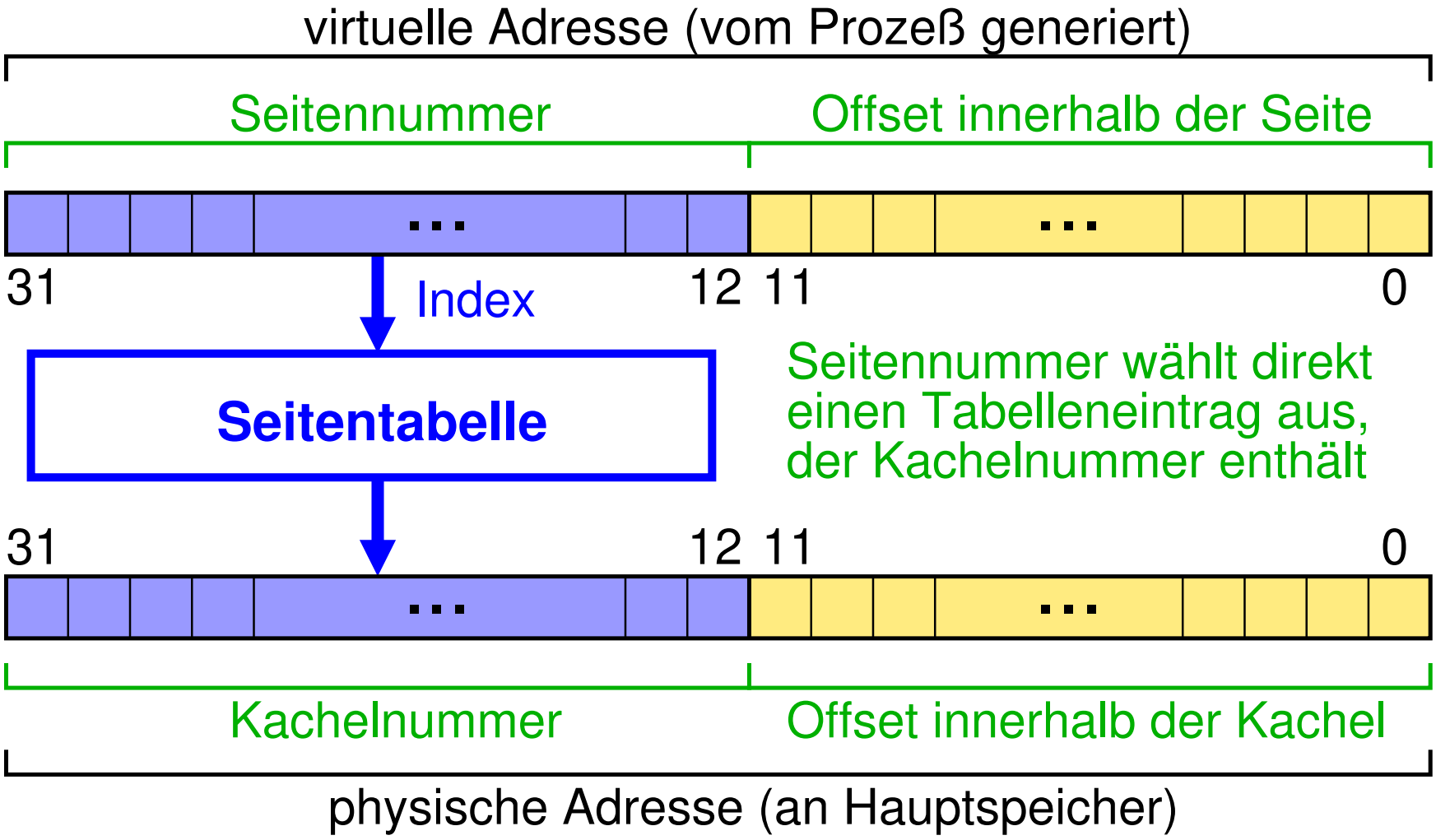
Prinzip der Adreßumsetzung



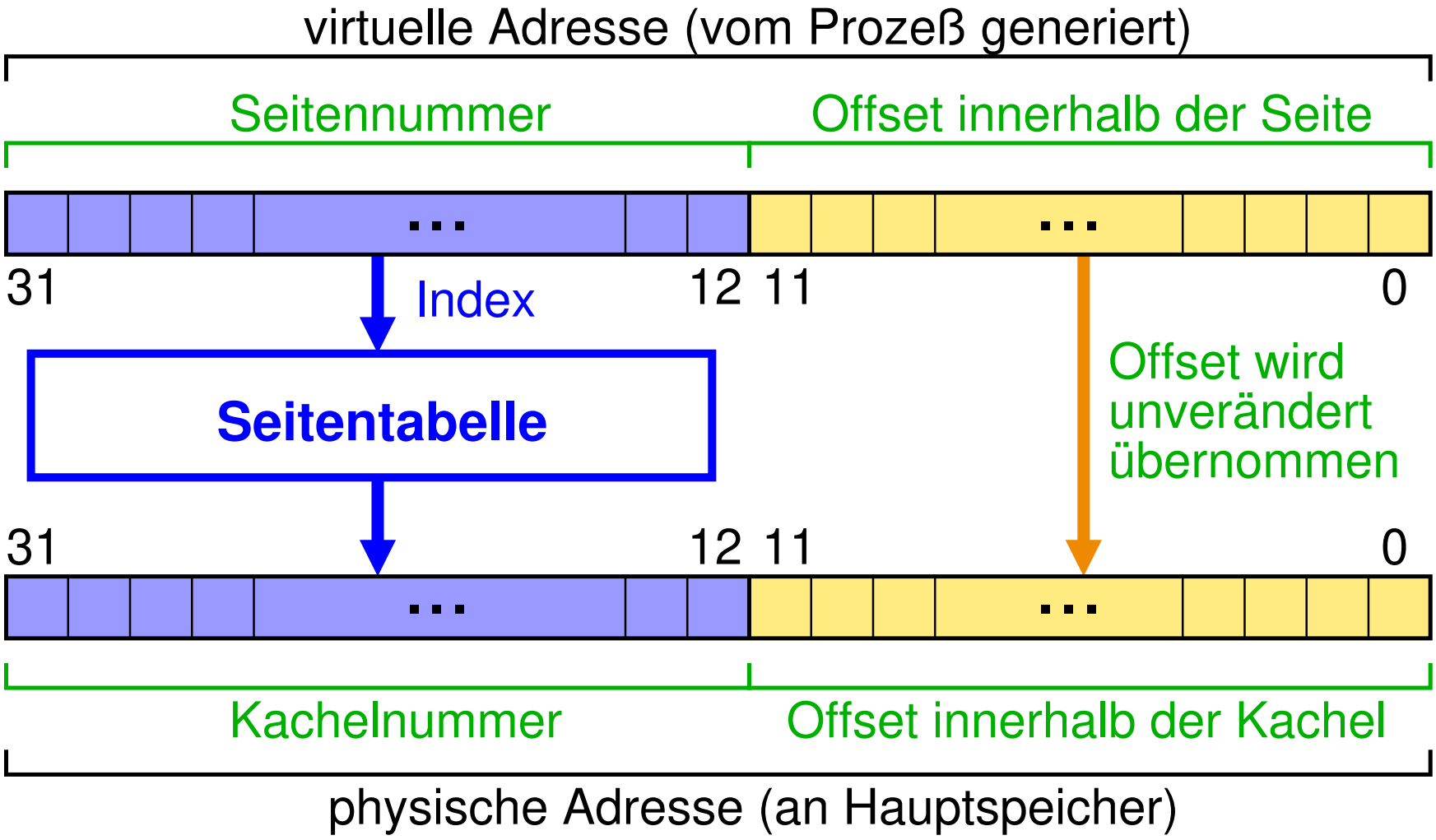
Prinzip der Adreßumsetzung



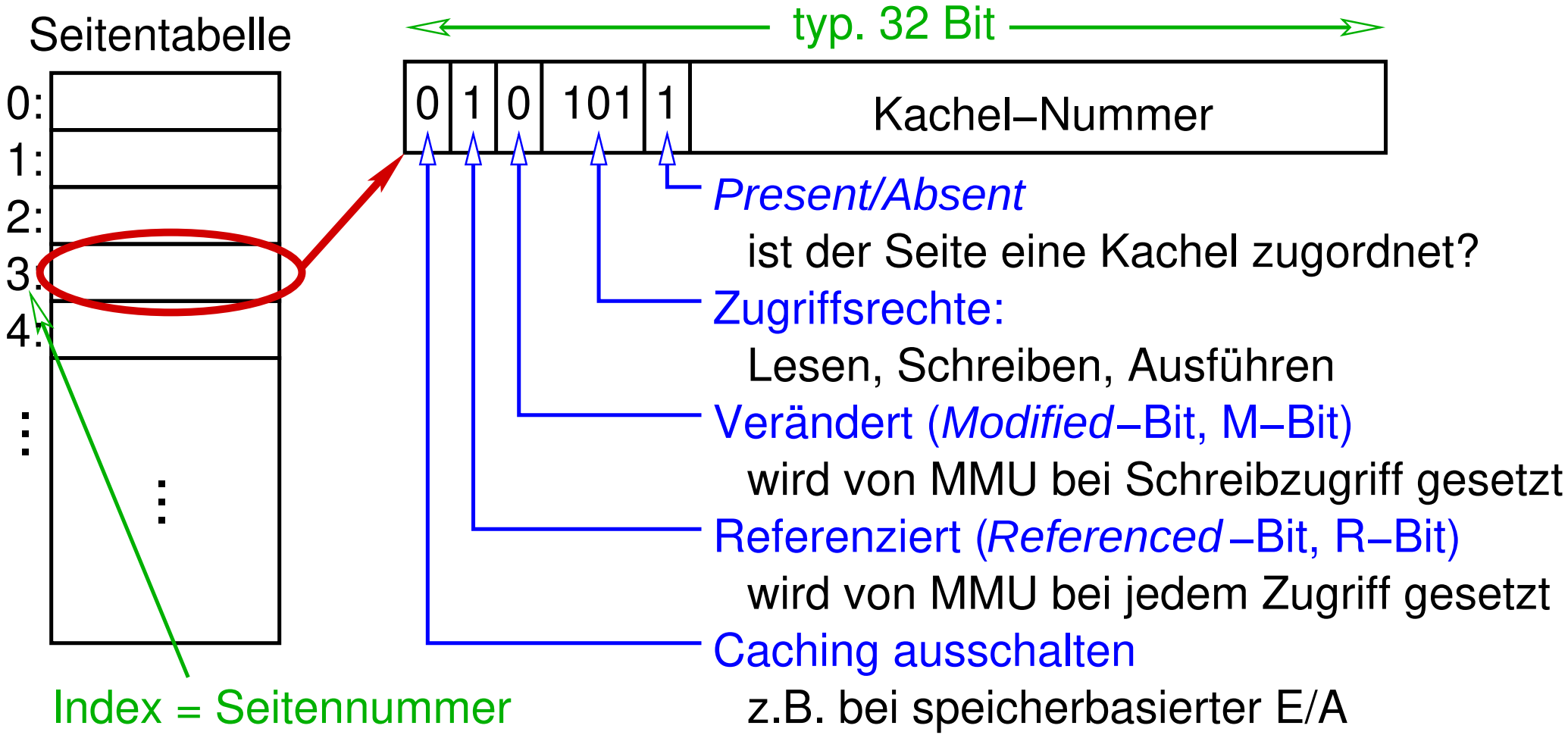
Prinzip der Adreßumsetzung



Prinzip der Adreßumsetzung



Typischer Aufbau der Seitentabelleneinträge



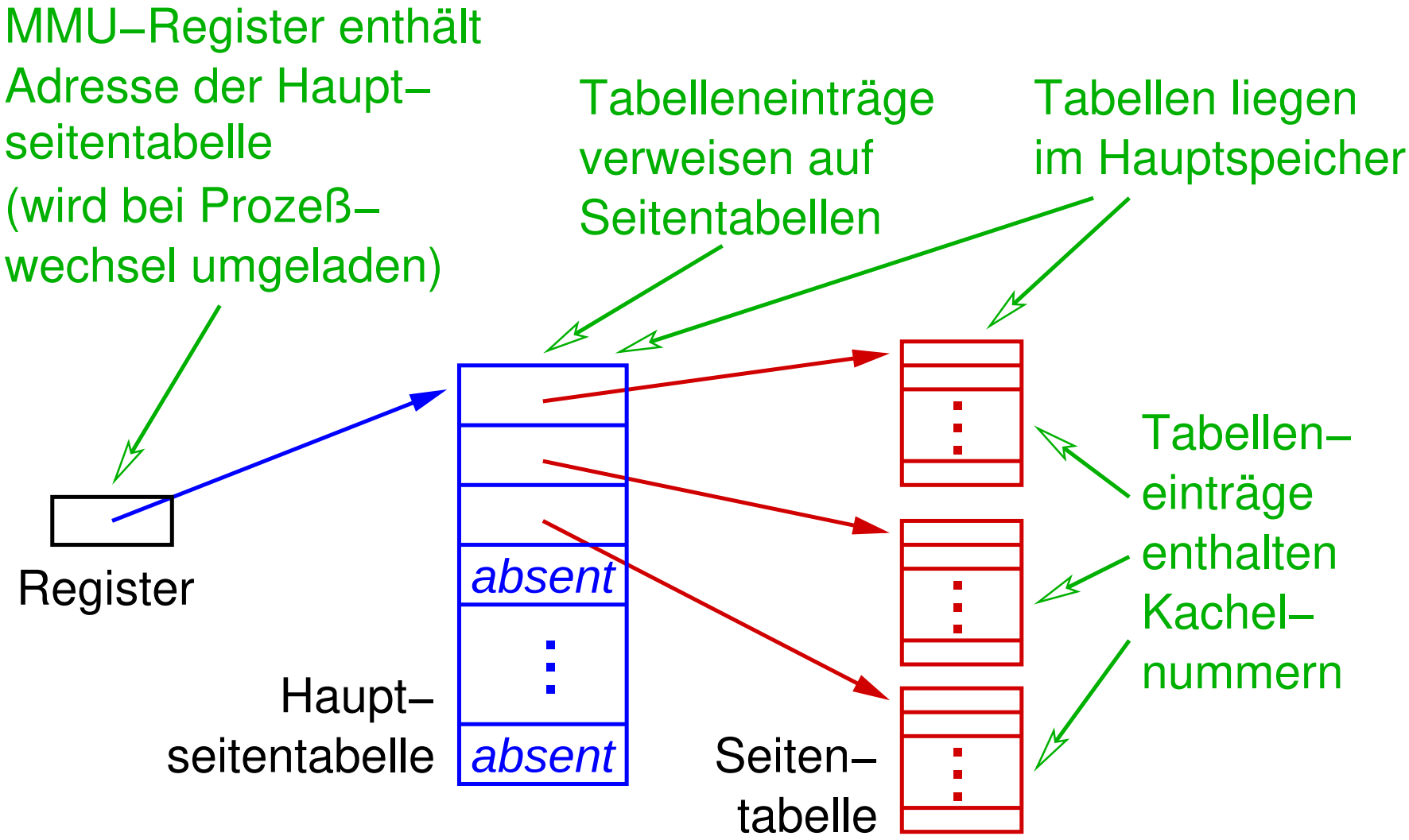
Adreßabbildung ist Zusammenspiel von BS und MMU

- BS setzt für jeden Prozeß eine Seitentabelle auf
  - Tabelle wird beim Prozeßwechsel ausgetauscht
- MMU realisiert die Adreßumsetzung
  - falls einer Seite keine Kachel zugeordnet ist (*Present*-Bit ist gelöscht):
    - MMU erzeugt Ausnahme (**Seitenfehler**)
  - Speicherschutz ist automatisch gegeben, da Seitentabelle nur auf Kacheln verweist, die dem Prozeß zugeteilt sind
  - zusätzlich: Zugriffsrechte für einzelne Seiten
    - z.B. Schreibschutz für Programmcode
    - bei Verletzung: Erzeugen einer Ausnahme

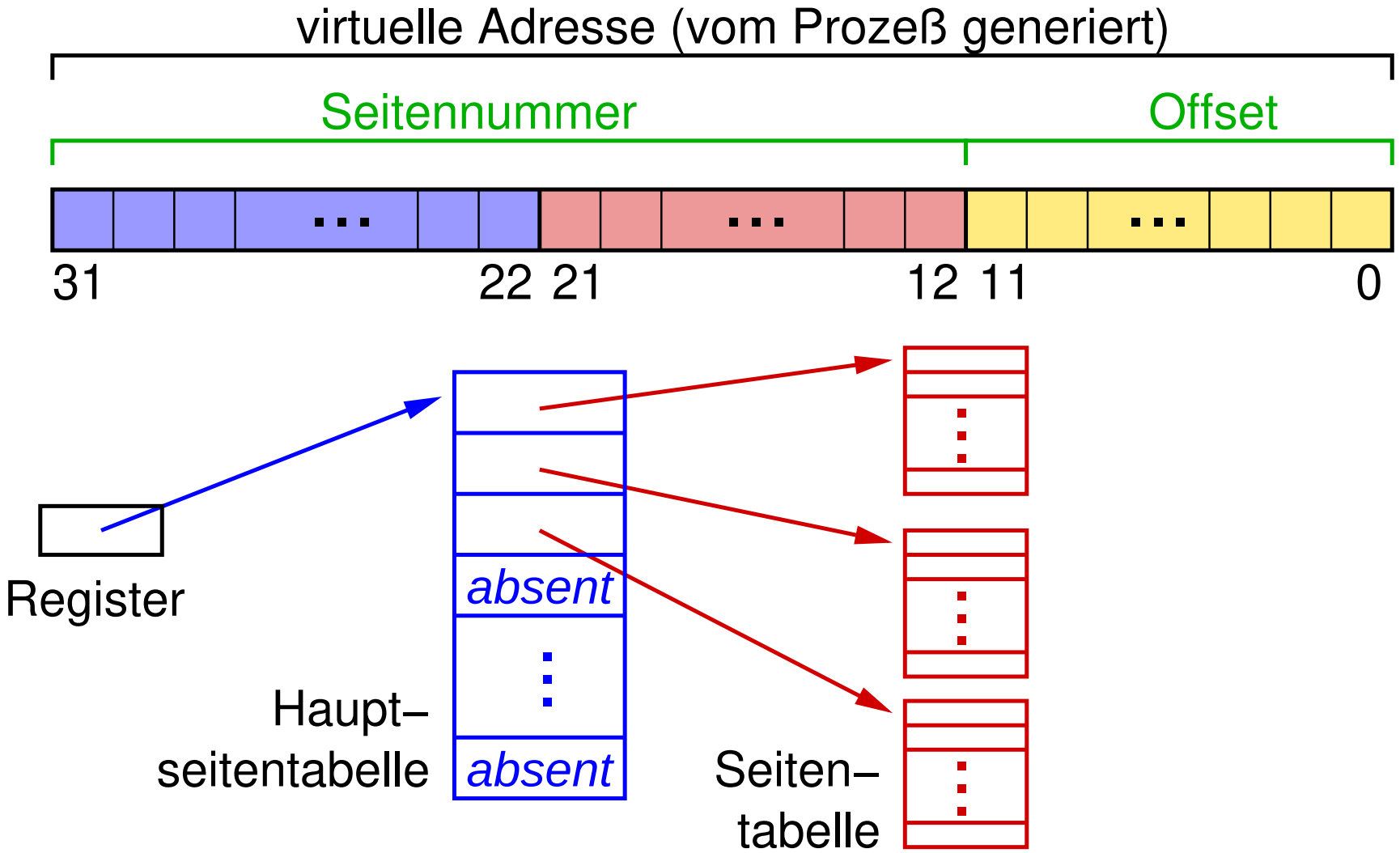
Mehrstufige Seitentabellen

- Jeder Seite muß eine Kachel zugeordnet werden können
- Problem: Speicherplatzbedarf der Tabelle
  - z.B. bei virtuellen Adressen mit 32 Bit Länge: 2^{20} Seiten
    - für jeden Prozeß wäre eine 4 MiB große Tabelle nötig
  - bei 64-Bit Prozessoren sogar 2^{52} Seiten, d.h. 32 PiB!
- Aber: Prozeß nutzt virtuellen Adreßraum i.a. nicht vollständig
  - daher mehrstufige Tabellen sinnvoll, z.B. für 2^{20} Seiten:
    - 1. Stufe: Hauptseitentabelle mit 1024 Einträgen
    - 2. Stufe: $\leq$ 1024 Seitentabellen mit je 1024 Einträgen (Tabellen nur vorhanden, wenn notwendig)
- (Alternative: **Invertierte Seitentabellen**)

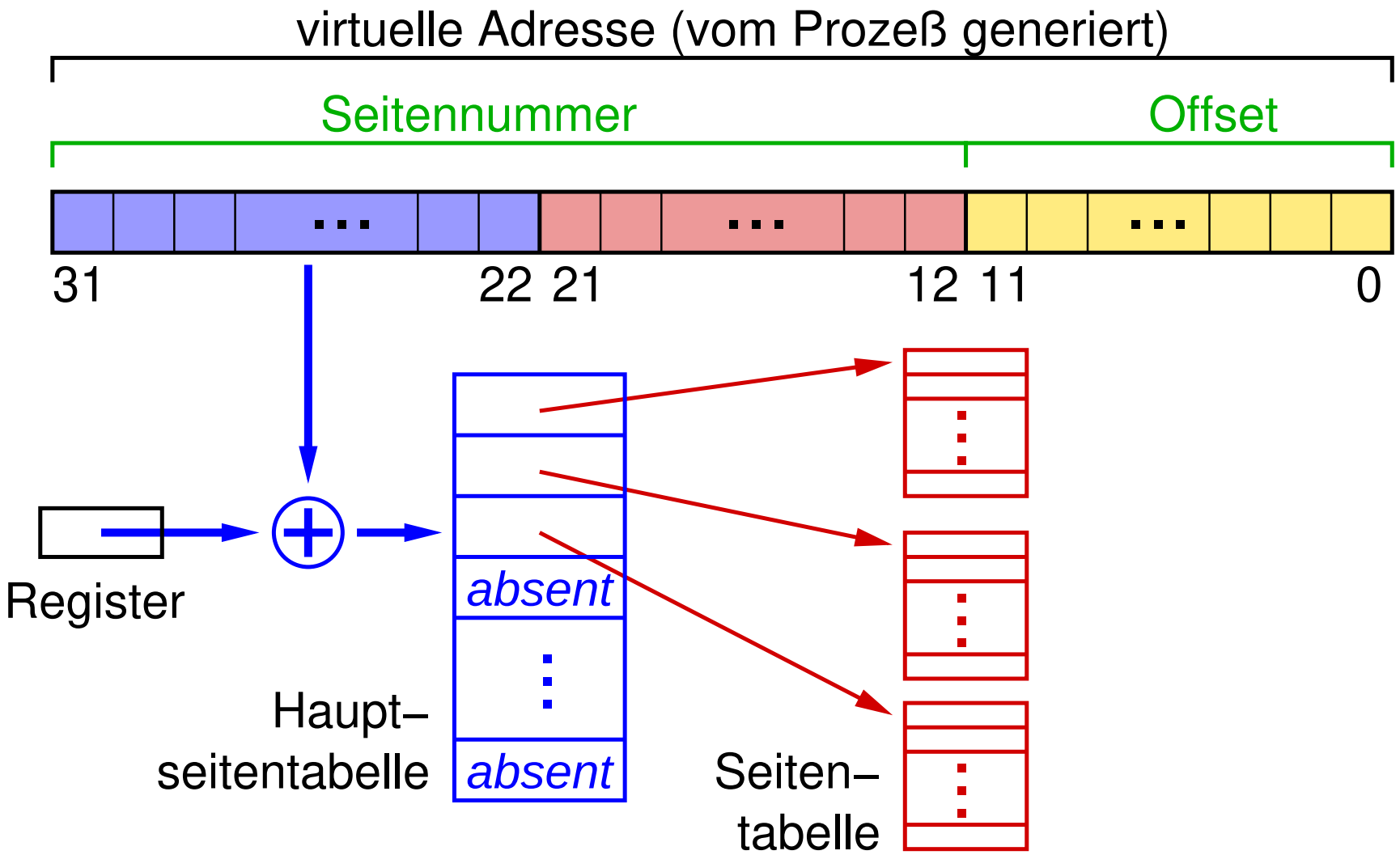
Adreßumsetzung mit zweistufiger Seitentabelle



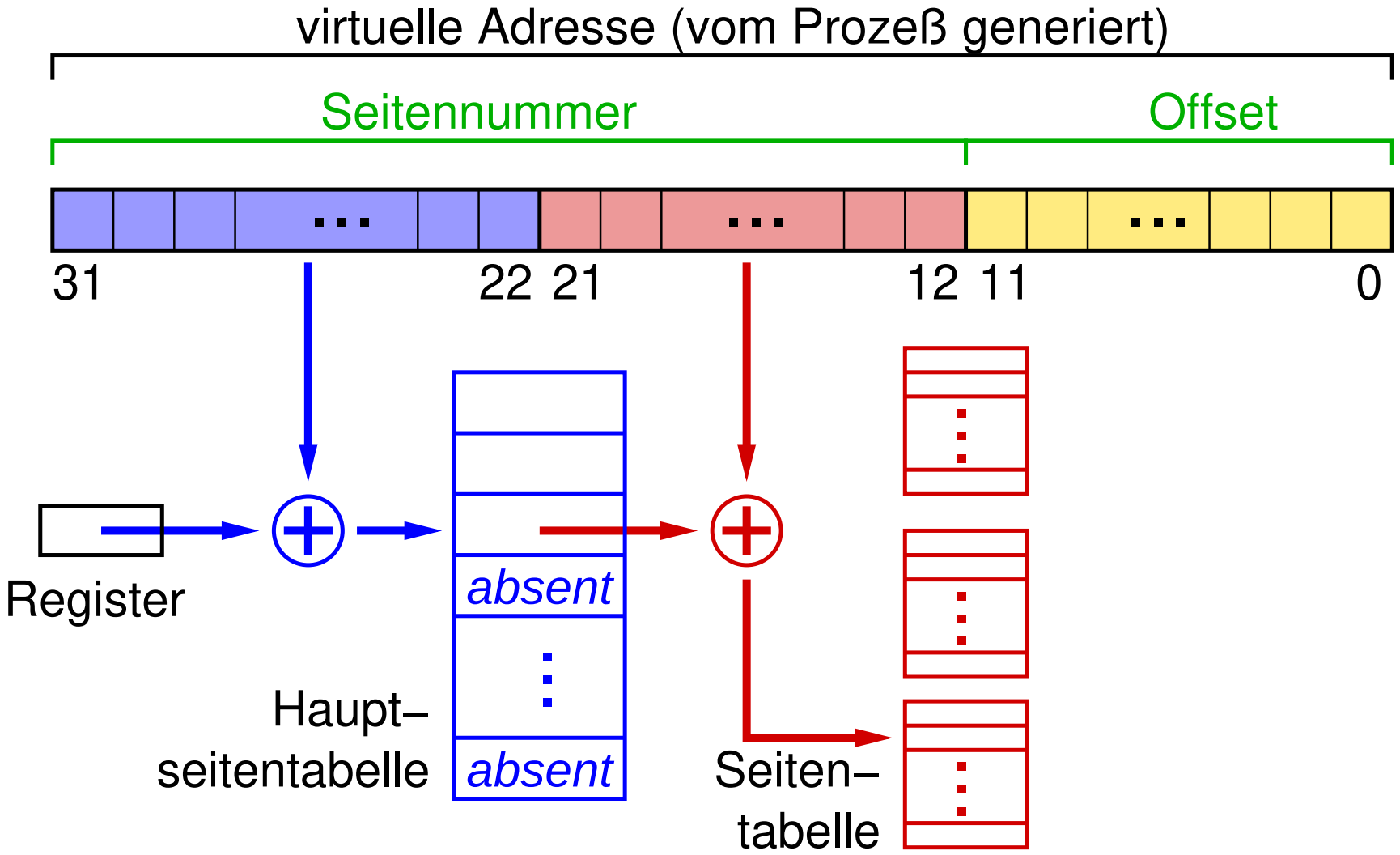
Adreßumsetzung mit zweistufiger Seitentabelle



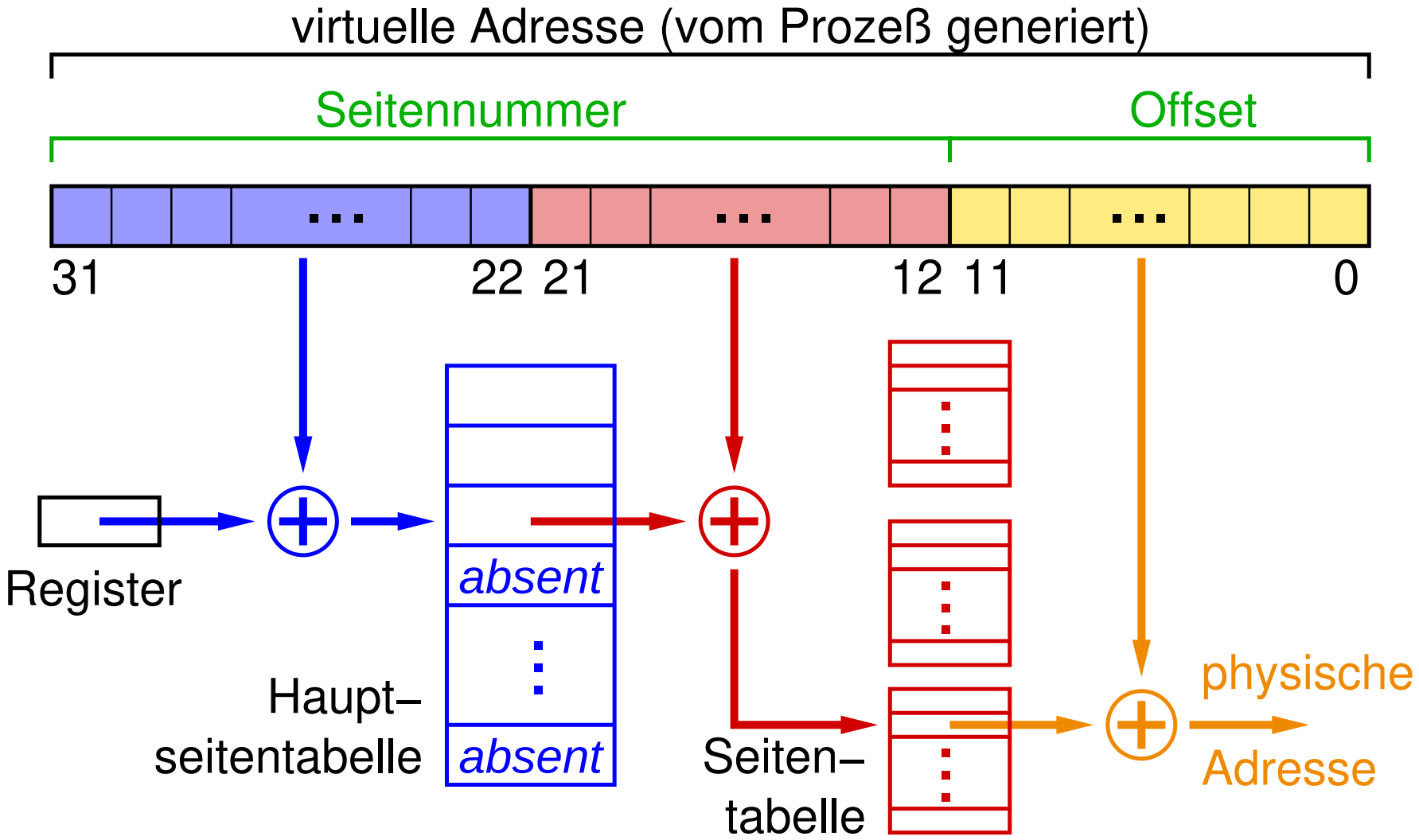
Adreßumsetzung mit zweistufiger Seitentabelle



Adreßumsetzung mit zweistufiger Seitentabelle



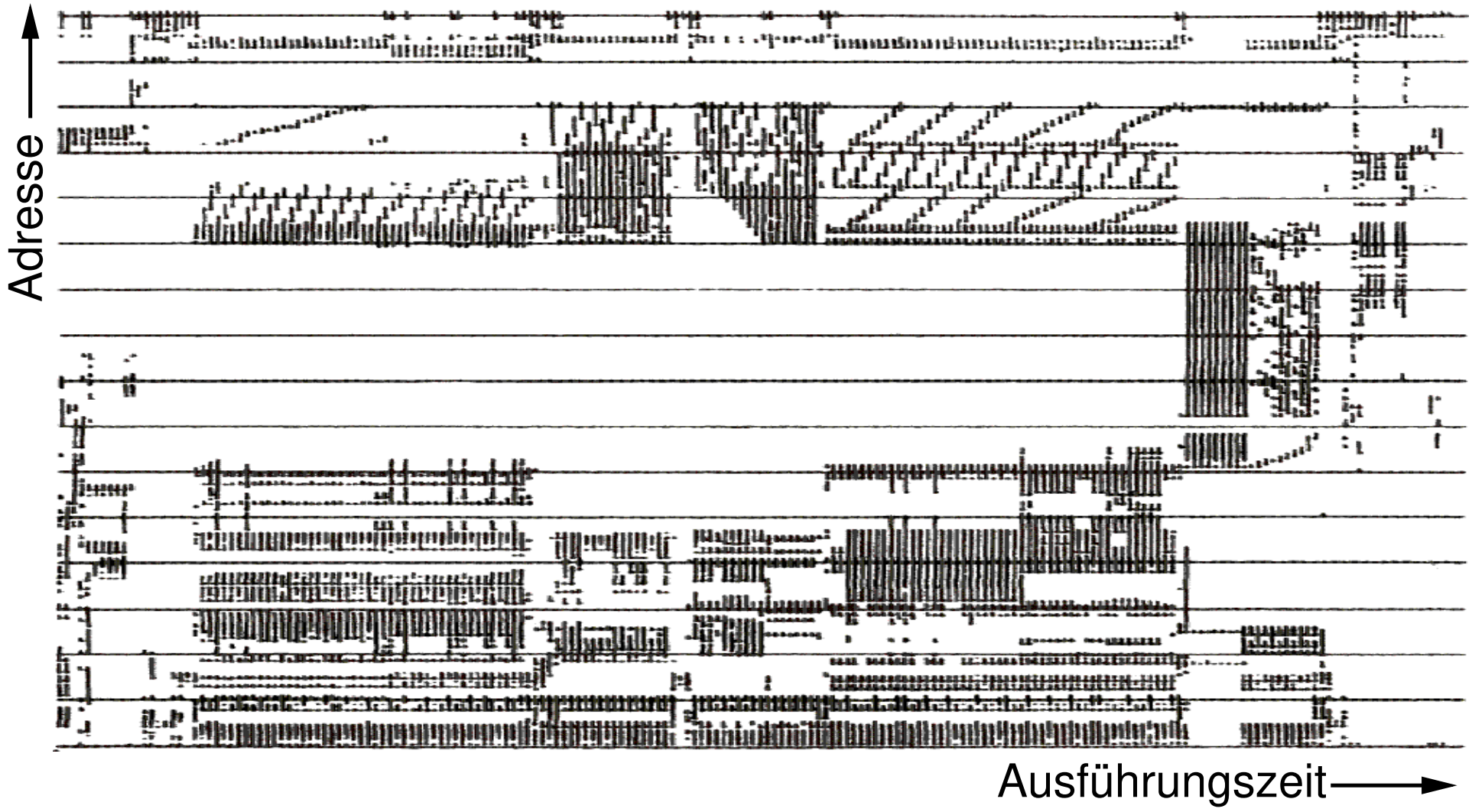
Adreßumsetzung mit zweistufiger Seitentabelle



TLB: *Translation Lookaside Buffer*

- Seitentabellen liegen im Speicher
  - für jeden Speicherzugriff mehrere zusätzliche Zugriffe für Adreßumsetzung nötig!
- Optimierung: MMU hält kleinen Teil der Adreßabbildung in einem internen Cache-Speicher: **TLB**
- Verwaltung des TLB
  - durch Hardware (MMU): bei CISC-Prozessoren (z.B. x86)
  - durch Software (BS): bei einigen RISC-Prozessoren
    - MMU erzeugt Ausnahme, falls für eine Seite kein TLB-Eintrag vorliegt
    - BS behandelt Ausnahme: Durchsuchen der Seitentabellen und Ersetzen eines TLB-Eintrags

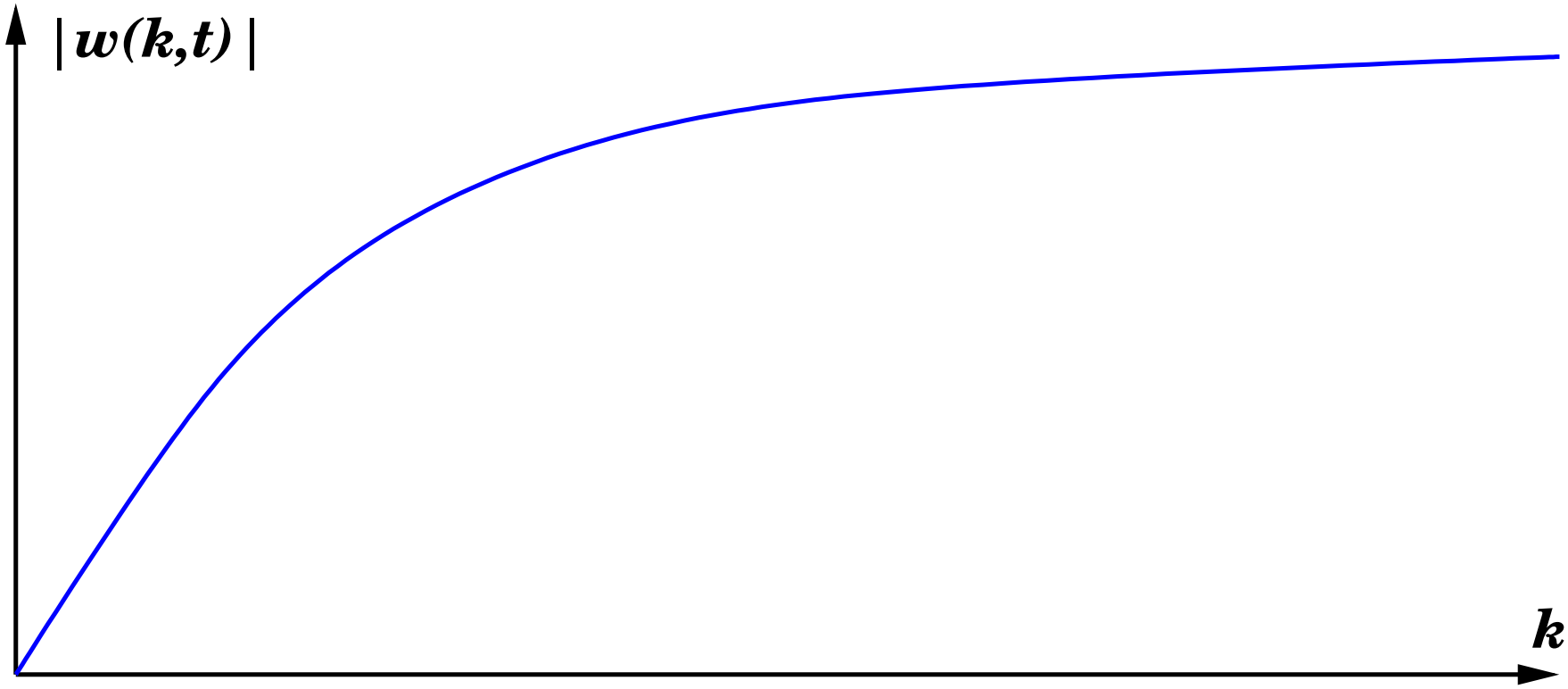
Motivation: Lokalitätsprinzip



Virtueller Speicher

- Idee: nur eine Teilmenge der Seiten eines Prozesses wird im Hauptspeicher gehalten: ***Resident Set***
  - alle anderen Seiten sind auf Festplatte ausgelagert
- Auswirkungen:
  - höherer Multiprogramming-Grad möglich
  - Prozeßadreßraum (logischer Adreßraum) kann größer als Hauptspeicher (physischer Adreßraum) sein
- ***Working Set*** eines Prozesses P: $w(k,t)$
  - zur Zeit t: Menge der Seiten, die P bei den letzten k Speicherzugriffen benutzt hat

Abhängigkeit des *Working Set* von k



- Für genügend großes k: $|w(k,t)|$ ist beinahe konstant
  - aber meist noch deutlich kleiner als Prozeßadreßraum

Working Set und Speicherzuteilung

- BS muß genügend Speicher bereitstellen, um für jeden Prozeß das *Working Set* $w(k,t)$ für genügend großes k im Hauptspeicher zu halten
- Falls nicht: ***Thrashing***, **Seitenflattern**
  - Prozesse benötigen ständig ausgelagerte Seiten
  - zum Einlagern muß aber andere Seite verdrängt werden
  - führt zu ständigem Ein- und Auslagern
  - Systemleistung sinkt dramatisch
- Mögliche Abhilfe: *Swapping*
  - verdränge einen Prozeß komplett aus dem Speicher

Strategieentscheidungen

- **Abrufstrategie**: wann werden Seiten eingelagert?
  - erst bei Bedarf, also bei Seitenfehler: ***Demand Paging***
  - im Voraus: ***Prepaging***
    - z.B. bei Programmstart
    - oder: lade Folgeseiten auf Platte gleich mit
  - (bei *Swapping* werden immer alle zuvor residenten Seiten wieder eingelagert)
- **Zuteilungsstrategie**: welche Kacheln werden einem Prozeß zugeteilt?
  - nur in Spezialfällen (Parallelrechner) relevant

Strategieentscheidungen ...

- **Austauschstrategie**
  - welche Seite wird verdrängt, wenn keine freie Kachel mehr vorhanden ist?
    - Seitenersetzungsalgorithmen (☞ **8.3.3**)
  - wo wird nach Verdrängungskandidaten gesucht?
    - **lokale Strategie**: verdränge nur Seiten des Prozesses, der neue Seite anfordert
      - Adreßraumgröße fest oder variabel
      - Größe sollte *Working Set* entsprechen
      - Einstellung z.B. aufgrund der Seitenfehlerfrequenz
    - **globale Strategie**: suche Verdrängungskandidaten unter den Seiten aller Prozesse

Ablauf eines Seitenwechsels (vereinfacht)

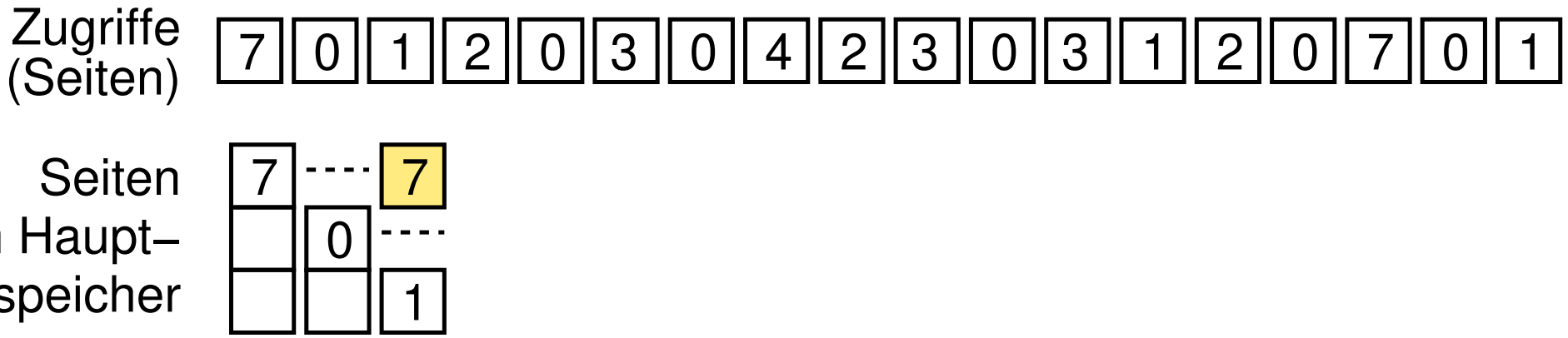
0. MMU hat Seitenfehler (Ausnahme) ausgelöst
1. BS ermittelt virtuelle Adresse (Seite S) u. Grund der Ausnahme
2. Falls Schutzverletzung vorlag: Prozeß abbrechen, Fertig.
3. Falls keine freie Kachel verfügbar:
   a) bestimme die zu verdrängende Seite S'
   b) falls S' modifiziert (*Modified*-Bit = 1): S' auf Platte schreiben
   c) Seitentabelleneintrag für S' aktualisieren (u.a. *Present*-Bit = 0)
4. Seite S von Platte in freie Kachel (ggf. die von S') laden
5. Seitentabelleneintrag für S aktualisieren (u.a. *Present*-Bit = 1)
6. unterbrochenen Thread fortsetzen
   ➥ abgebrochener Befehl wird weitergeführt oder wiederholt

Zu den Kosten von Seitenwechseln

- Mittlere Speicherzugriffszeit bei Wahrscheinlichkeit p für Seitenfehler:
  - $t_Z = t_{HS} + p \cdot t_{SF}$
  - t_{HS}: Zugriffszeit des Hauptspeichers (ca. 10-100 ns)
  - t_{SF}: Zeit für Behandlung eines Seitenfehlers
    - dominiert durch Plattenzugriff (ca. 10 ms)
- Um Leistungsverlust im Rahmen zu halten:
  - p muß sehr klein sein
  - max. ein Seitenfehler bei mehreren Millionen Zugriffen

Optimale Strategie (nach Belady)

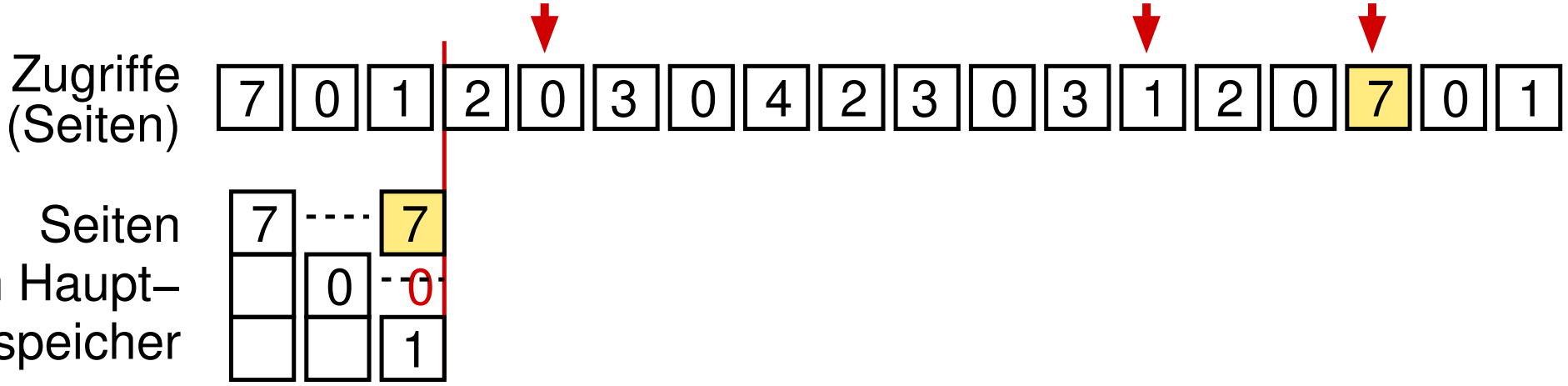
- Verdränge die Seite, die in Zukunft am längsten nicht mehr benötigt wird



- In der Praxis nicht realisierbar
- Als Referenzmodell zur Bewertung anderer Verfahren

Optimale Strategie (nach Belady)

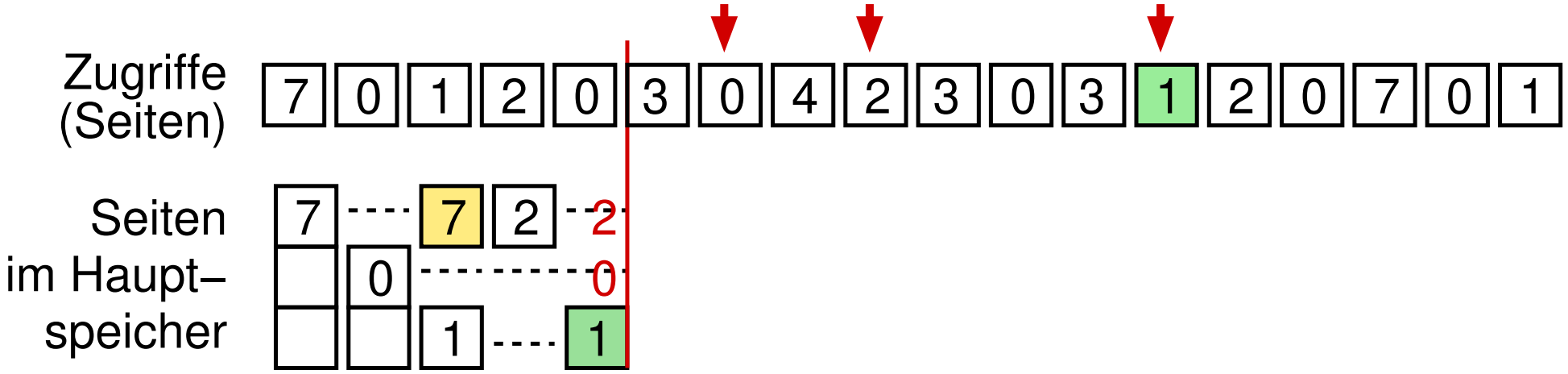
- Verdränge die Seite, die in Zukunft am längsten nicht mehr benötigt wird



- In der Praxis nicht realisierbar
- Als Referenzmodell zur Bewertung anderer Verfahren

Optimale Strategie (nach Belady)

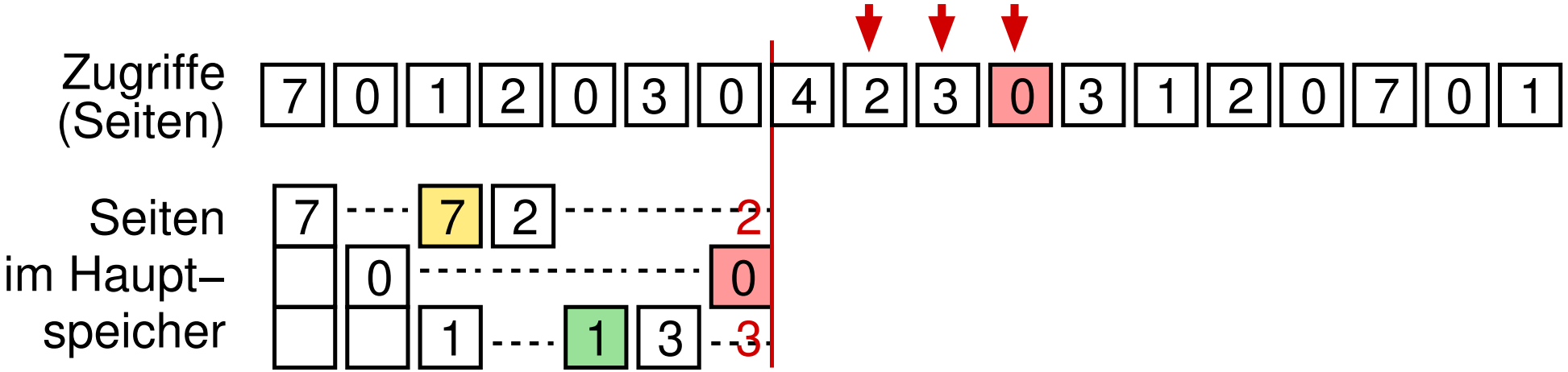
- Verdränge die Seite, die in Zukunft am längsten nicht mehr benötigt wird



- In der Praxis nicht realisierbar
- Als Referenzmodell zur Bewertung anderer Verfahren

Optimale Strategie (nach Belady)

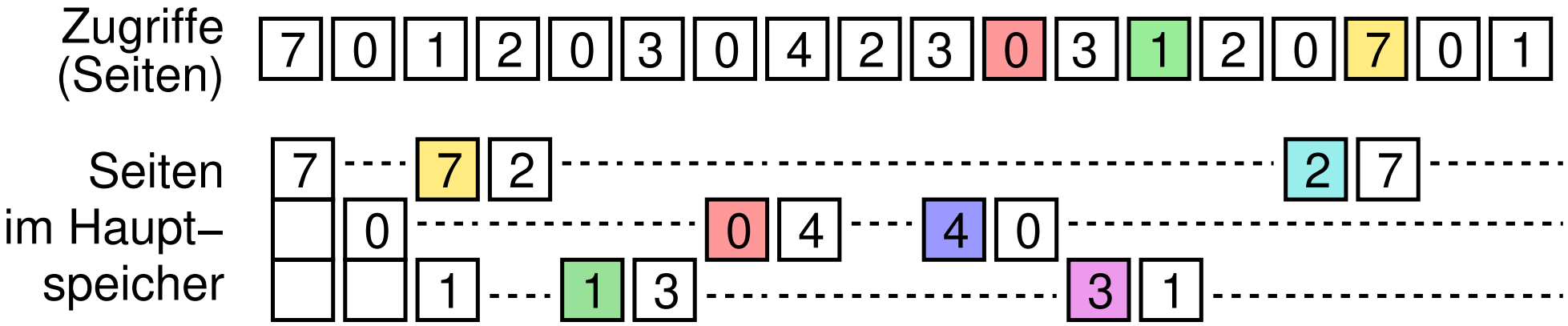
- Verdränge die Seite, die in Zukunft am längsten nicht mehr benötigt wird



- In der Praxis nicht realisierbar
- Als Referenzmodell zur Bewertung anderer Verfahren

Optimale Strategie (nach Belady)

- Verdränge die Seite, die in Zukunft am längsten nicht mehr benötigt wird



- In der Praxis nicht realisierbar
- Als Referenzmodell zur Bewertung anderer Verfahren

Not Recently Used (NRU)

- Basis: von der MMU gesetzte Statusbits in Seitentabelle:
  - **R**-Bit: Seite wurde referenziert
  - **M**-Bit: Seite wurde modifiziert
- **R**-Bit wird vom BS in regelmäßigem Abstand gelöscht
- Bei Verdrängung: vier Prioritätsklassen

Klasse 0:	nicht referenziert,	nicht modifiziert
Klasse 1:	nicht referenziert,	modifiziert
Klasse 2:	referenziert,	nicht modifiziert
Klasse 3:	referenziert,	modifiziert

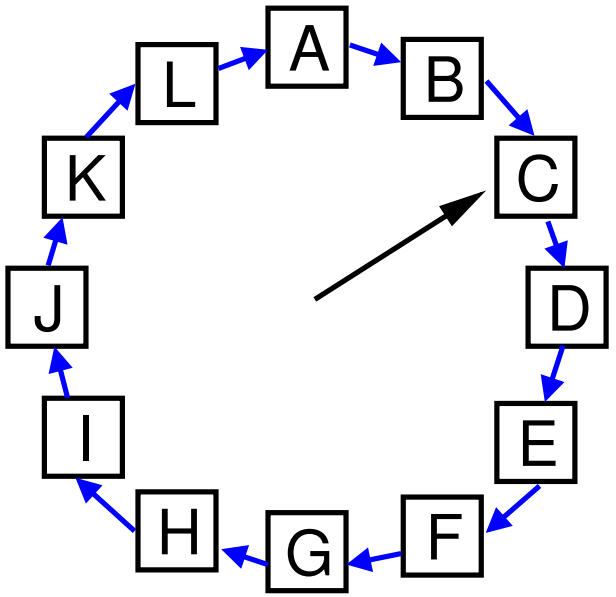
- Auswahl innerhalb der Klasse zufällig
- Nicht besonders gut, aber einfach

First In First Out (FIFO)

- Verdränge die Seite, die am längsten im Hauptspeicher ist
- Einfache, aber schlechte Strategie
  - Zugriffsverhalten (*Working Set*) wird ignoriert

Second Chance bzw. *Clock*-Algorithmus

- Erweiterung von FIFO
- Idee: verdränge älteste Seite, auf die seit dem letzten Seitenwechsel nicht zugegriffen wurde
- Seiten im Hauptspeicher werden nach Alter sortiert in Ringliste angeordnet, Zeiger zeigt auf älteste Seite



Bei Seitenfehler:

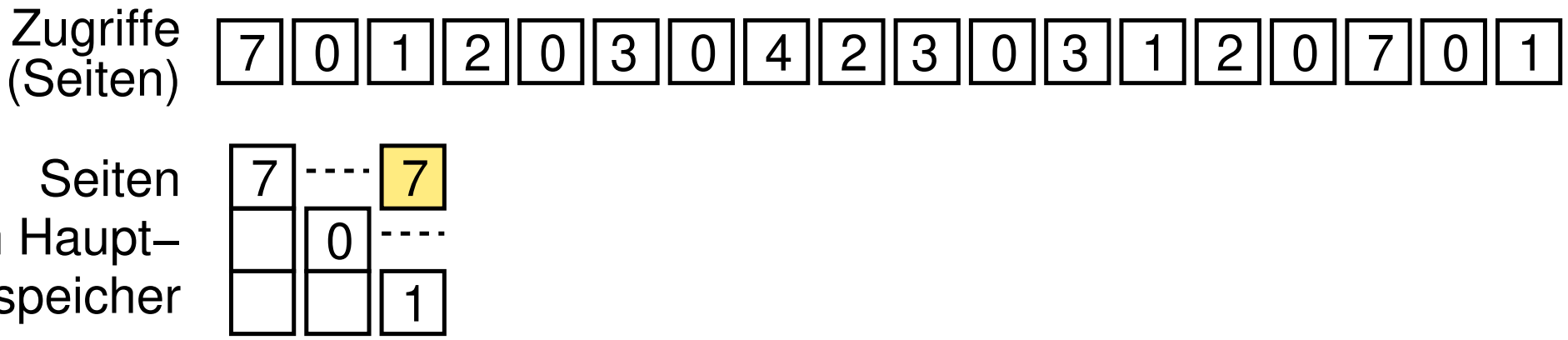
betrachte Seite, auf die der Zeiger zeigt:

R = 0: verdränge Seite, fertig

R = 1: lösche **R**-Bit,
setze Zeiger eins weiter,
wiederhole von vorne

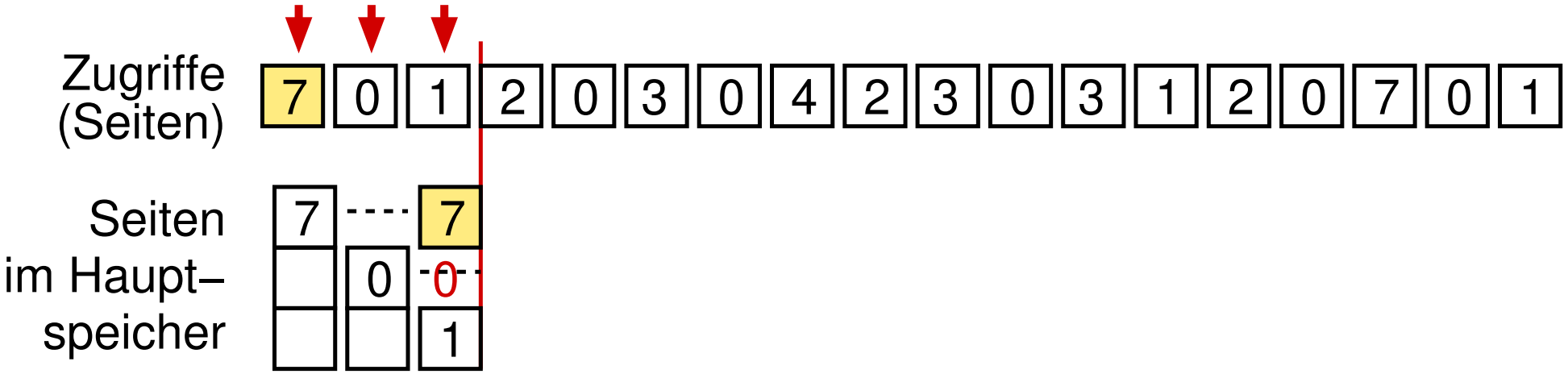
Least Recently Used (LRU)

- Verdränge die Seite, die am längsten nicht benutzt wurde
  - Vermutung: wird auch in Zukunft nicht mehr benutzt
- Nahezu optimal, wenn Lokalität gegeben ist
- Problem: (Software-)Implementierung
  - bei jedem Zugriff muß ein Zeitstempel aktualisiert werden
  - daher i.a. nur Näherungen (z.B. *Aging*)



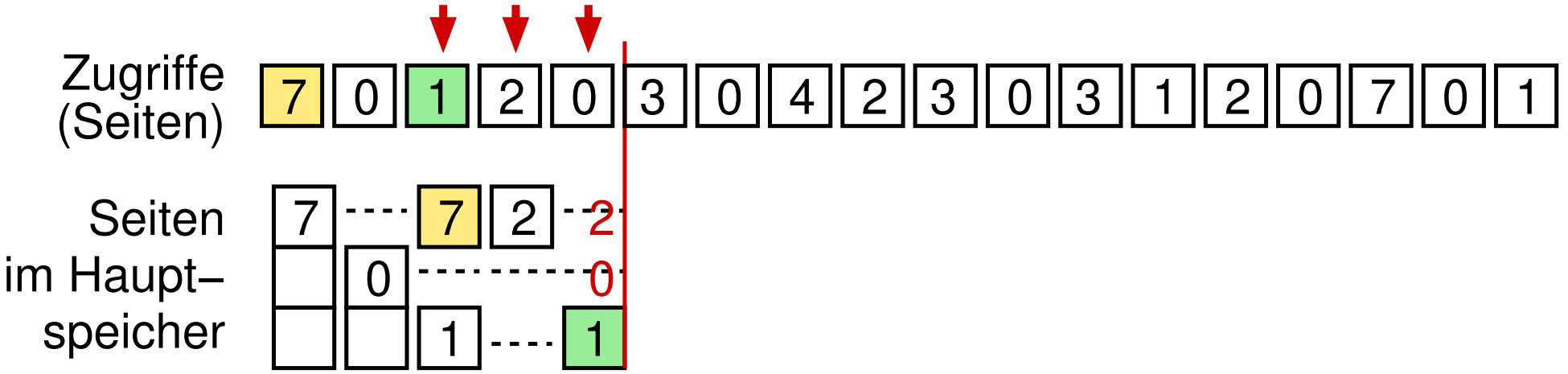
Least Recently Used (LRU)

- Verdränge die Seite, die am längsten nicht benutzt wurde
  - Vermutung: wird auch in Zukunft nicht mehr benutzt
- Nahezu optimal, wenn Lokalität gegeben ist
- Problem: (Software-)Implementierung
  - bei jedem Zugriff muß ein Zeitstempel aktualisiert werden
  - daher i.a. nur Näherungen (z.B. *Aging*)



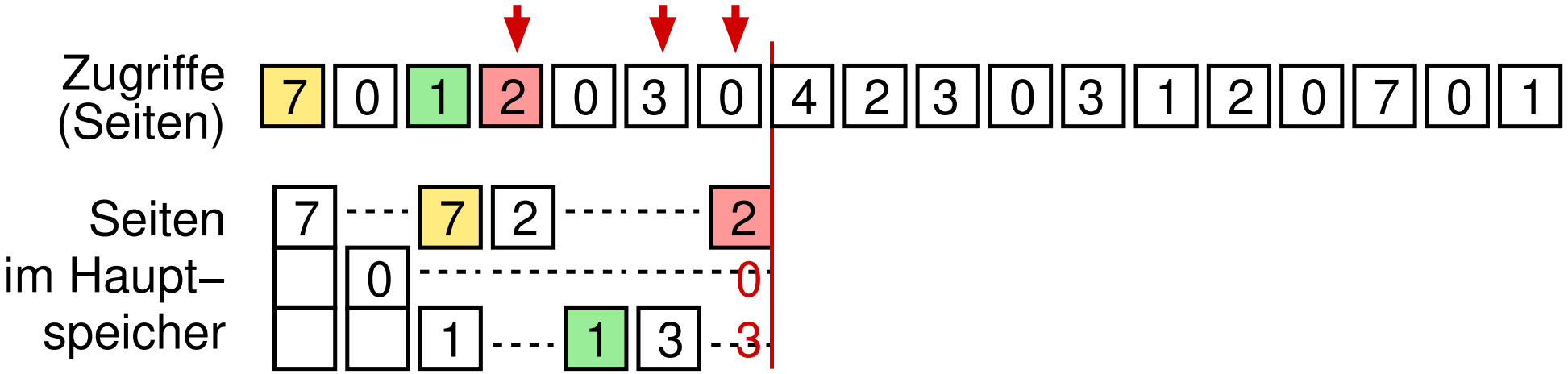
Least Recently Used (LRU)

- Verdränge die Seite, die am längsten nicht benutzt wurde
  - Vermutung: wird auch in Zukunft nicht mehr benutzt
- Nahezu optimal, wenn Lokalität gegeben ist
- Problem: (Software-)Implementierung
  - bei jedem Zugriff muß ein Zeitstempel aktualisiert werden
  - daher i.a. nur Näherungen (z.B. *Aging*)



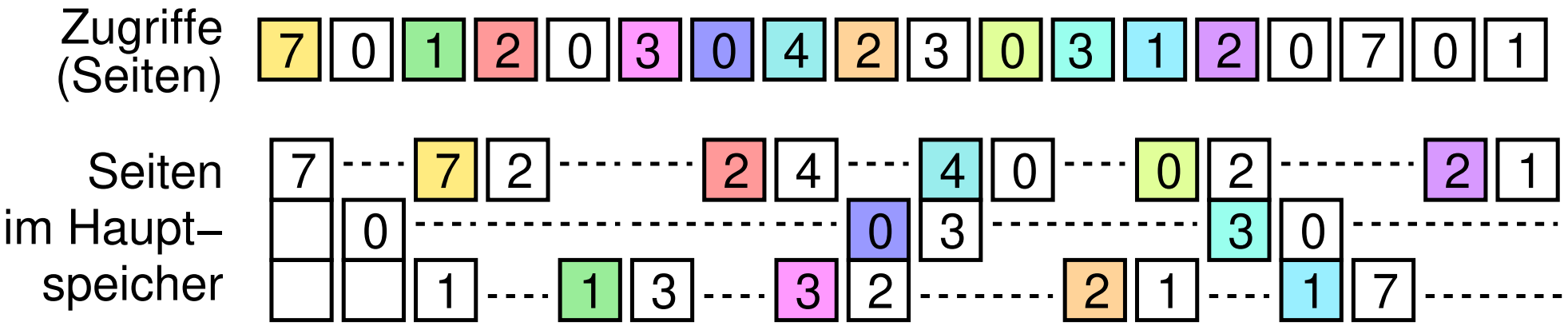
Least Recently Used (LRU)

- Verdränge die Seite, die am längsten nicht benutzt wurde
  - Vermutung: wird auch in Zukunft nicht mehr benutzt
- Nahezu optimal, wenn Lokalität gegeben ist
- Problem: (Software-)Implementierung
  - bei jedem Zugriff muß ein Zeitstempel aktualisiert werden
  - daher i.a. nur Näherungen (z.B. *Aging*)



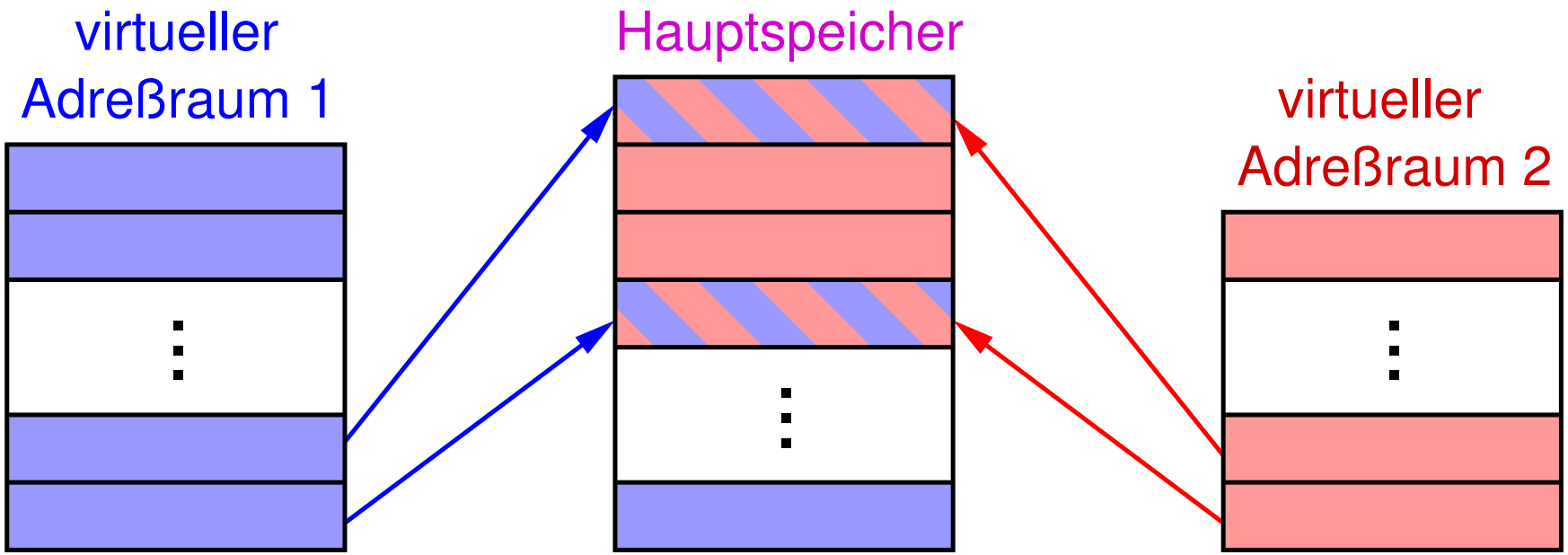
Least Recently Used (LRU)

- Verdränge die Seite, die am längsten nicht benutzt wurde
  - Vermutung: wird auch in Zukunft nicht mehr benutzt
- Nahezu optimal, wenn Lokalität gegeben ist
- Problem: (Software-)Implementierung
  - bei jedem Zugriff muß ein Zeitstempel aktualisiert werden
  - daher i.a. nur Näherungen (z.B. *Aging*)



8.3.4 Gemeinsamer Speicher zw. Prozessen

- Adreßräume von Prozessen können teilweise überlappen



- Anwendung:
  - gemeinsame Speichersegmente zur Kommunikation
  - gemeinsam genutzte Bibliotheken (*Shared Library*, DLL)

8.4 Zusammenfassung / Wiederholung

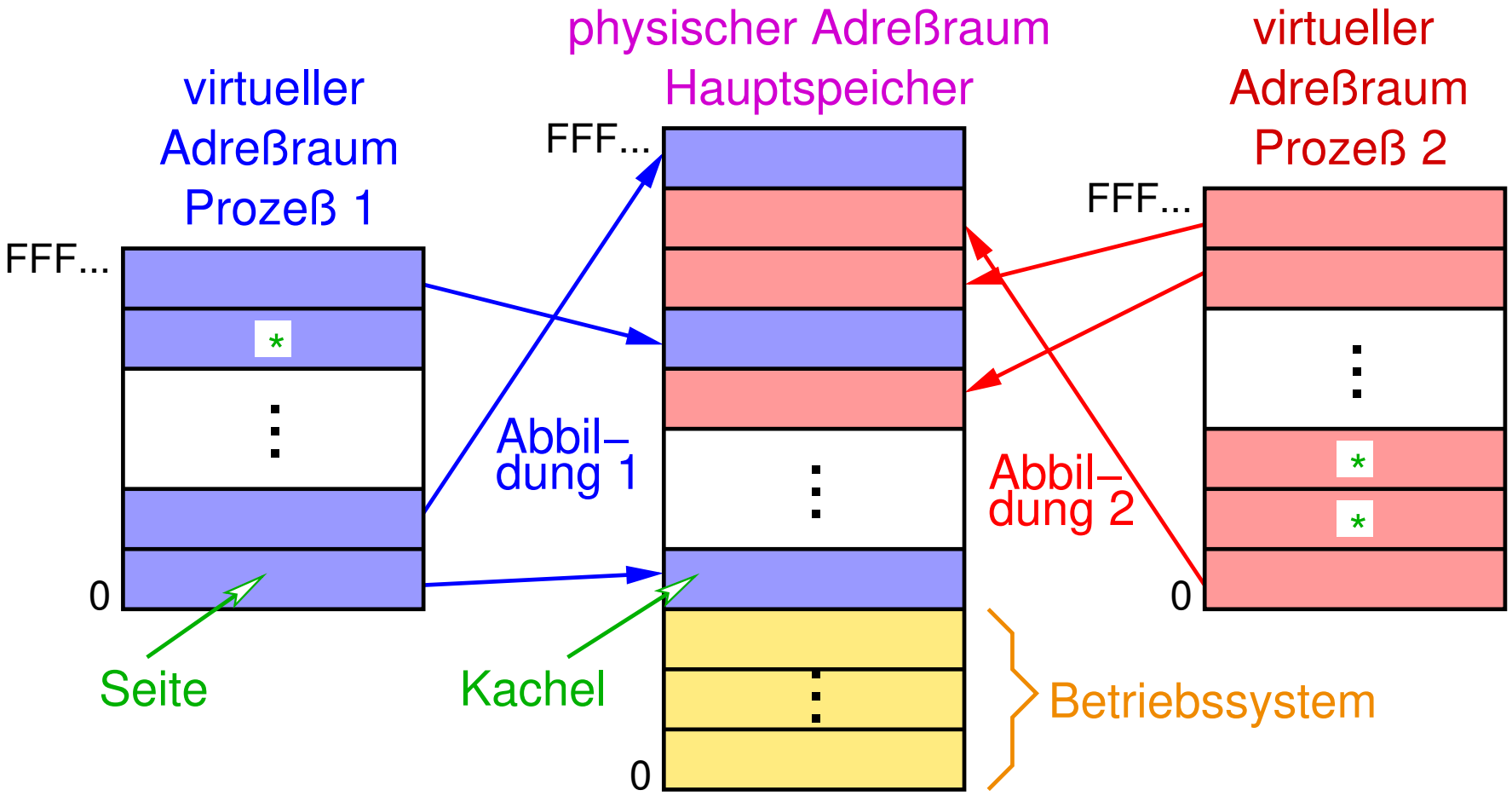
- Ziele der Speicherverwaltung:
  - effiziente Speicherzuweisung, Speicherschutz
- Wichtig: Unterscheidung logischer / physischer Adreßraum
- *Swapping*: Ein-/Auslagern kompletter Adreßräume
  - Suche nach freiem Speicher: *First Fit*, *Quick Fit*

8.4 Zusammenfassung / Wiederholung ...

- Virtuelle Speicherverwaltung: *Paging*
  - Einteilung des logischen Adreßraums in Seiten
  - Einteilung des physichen Adreßraums in Kacheln
  - jede Seite kann
    - auf beliebige Kachel im Speicher abgebildet werden
    - auf Festplatte ausgelagert werden
  - Hardware (MMU) bildet bei jedem Speicherzugriff logische auf physische Adresse ab
  - Beschreibung der Abbildung in Seitentabelle
    - pro Seite ein Eintrag, enthält u.a.:
      - Kachel-Nummer
      - *Present*-Bit: ist der Seite eine Kachel zugewiesen?

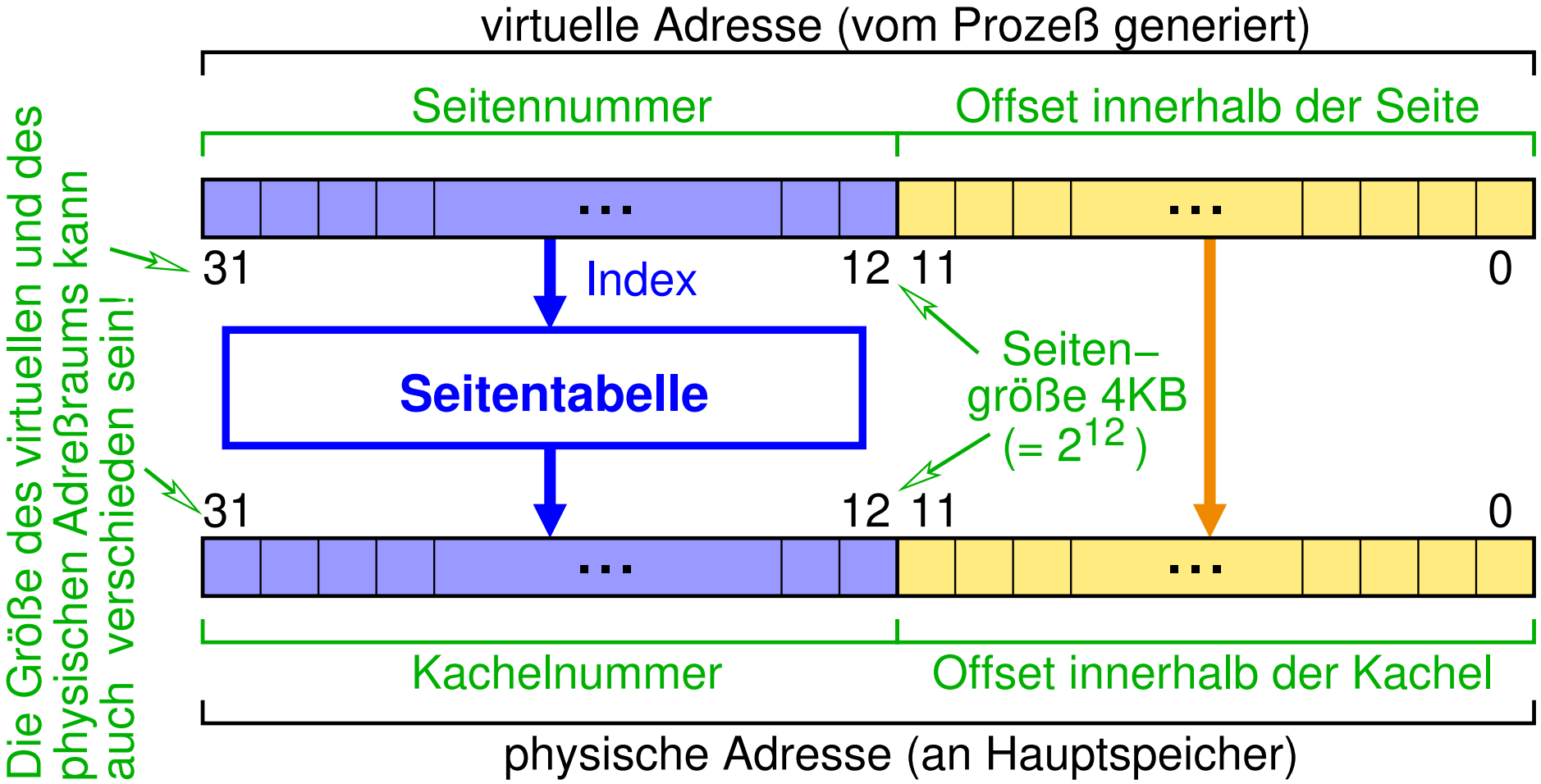
8.4 Zusammenfassung / Wiederholung ...

- Grundidee des *Paging*



* Diese Seiten sind nicht in den Hauptspeicher abgebildet
Sie könnten z.B. auf Hintergrundspeicher ausgelagert sein.

- Prinzip der Adreßumsetzung



8.4 Zusammenfassung / Wiederholung ...

- Mehrstufige Seitentabellen
  - Tabellen tieferer Stufen nur vorhanden, falls nötig
- TLB: Cache in der MMU
  - speichert die zuletzt verwendeten Tabelleneinträge
- Dynamische Seitenersetzung
  - nur die aktuell benötigten Seiten (*Working Set*) werden im Hauptspeicher gehalten
  - Rest auf Plattenspeicher verdrängt
  - bei Zugriff auf ausgelagerte Seite: Seitenfehler
    - BS lädt Seite in Hauptspeicher, muß ggf. andere Seite verdrängen (Seitenersetzung)

8.4 Zusammenfassung / Wiederholung ...

- Seitenersetzungsalgorithmen
  - bestimmen, welche Seite verdrängt wird
  - Optimale Strategie: die Seite, die in Zukunft am längsten nicht benötigt wird
  - NRU: vier Klassen gemäß **R** und **M**-Bit
  - FIFO: die Seite, die am längsten im Hauptspeicher ist
  - *Second Chance*: die älteste Seite, die seit letztem Seitenwechsel nicht benutzt wurde
    - *Clock*-Algorithmus: effiziente Implementierung
  - LRU: die Seite, die am längsten nicht benutzt wurde

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

9 Ein-/Ausgabe und Dateisysteme

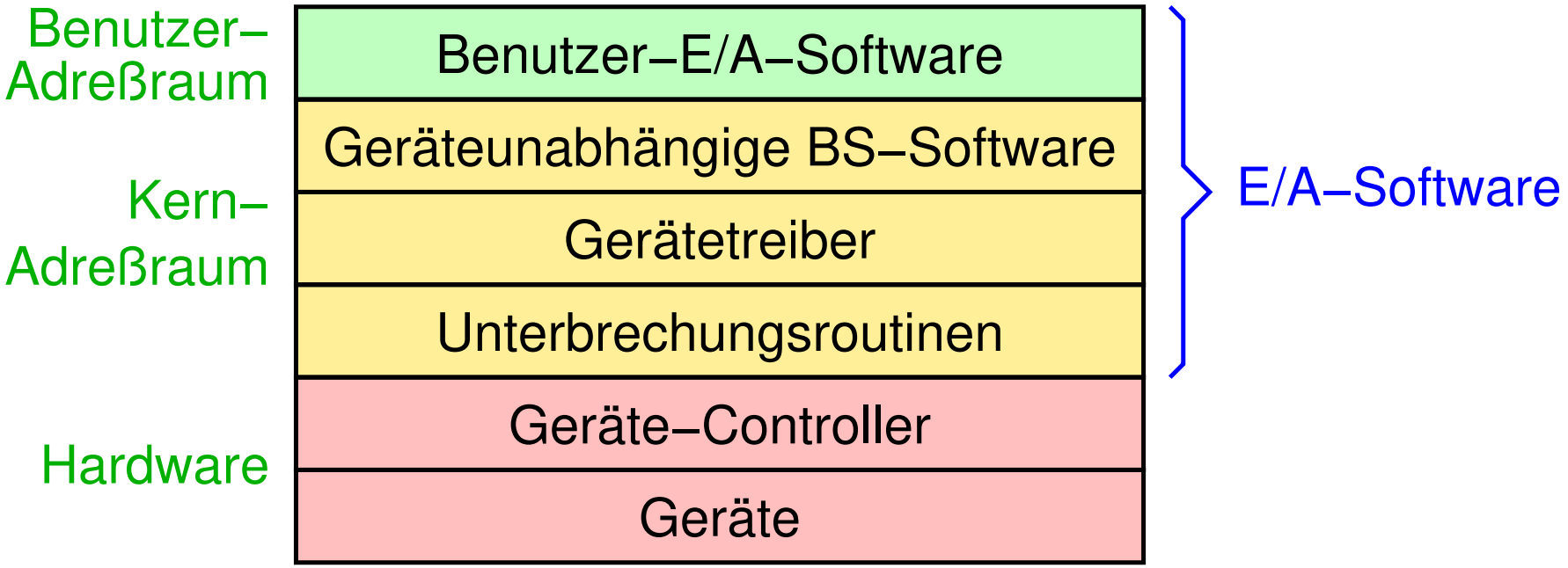
Inhalt:

- Schichten der E/A-Software
- Ansätze zur Durchführung der E/A
- Festplatten
- (Dateiverwaltung, ☞ **1.5.3**)
- Realisierung von Dateisystemen

- Tanenbaum 5.2 - 5.4, 6.1-6.3, 6.4.5
- Stallings 11.2 - 11.6, 12.1, 12.3, 12.5-12.7
- Nehmer/Sturm 10.1, 9

9.1 Schichten der E/A-Software

- Schichten bei der Ein-/Ausgabe:



- In der Regel für jeden Gerätetyp eigene Controller und Gerätetreiber

Gerätetreiber

- Geräteabhängiger Teil der E/A-Software
  - kommuniziert direkt mit dem jeweiligen Geräte-Controller
- Heute i.d.R. in Kern-Adreßraum eingebunden
  - beim Hochfahren des BSs oder zur Laufzeit
  - Problem: fehlerhafte Treiber können zu BS-Abstürzen führen
  - Lösungsmöglichkeiten:
    - zertifizierte Treiber
    - Treiber als Systemprozesse im Benutzeradreßraum
    - Universalgeräte mit universellen Treibern
- Einheitliche Schnittstelle zwischen Treibern und BS
  - nur Unterscheidung block-/zeichenorientierte Geräte

Geräteunabhängige E/A-Software

- Aufgaben:
  - Einheitliche Schnittstelle für Gerätetreiber
    - Namensgebung für Geräte, Zuordnung zu Treibern
    - Zugriffsschutz
  - Pufferung von Daten
  - Fehlerbehandlung
  - Anforderung und Freigabe von Geräten
    - für exklusive Nutzung, z.B. CD-Brenner
  - Verdeckung von Unterschieden der Geräte
    - z.B. unterschiedliche Blockgrößen

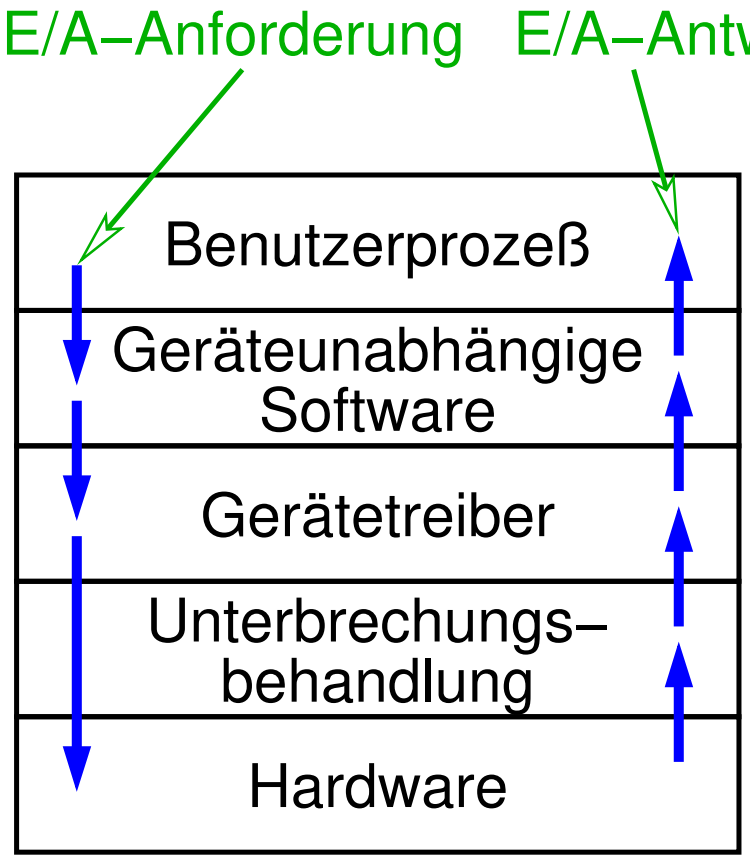
Geräteunabhängige E/A-Software ...

- Bereitgestellte Grundfunktionen:
  - Öffnen eines Geräts
    - Argumente: Gerätename, Betriebsparameter
    - Ergebnis: *Handle* zum Zugriff auf das Gerät
  - Schließen des Geräts
  - Lesen / Schreiben von Daten(blöcken)
- In UNIX:
  - Geräte sind in das Dateisystem abgebildet
    - z.B. `/dev/mouse`, `/dev/hda`
  - Zugriff auf Geräte über normale Dateioperationen

Benutzer-E/A-Software

- Bibliotheksfunktionen
  - z.B. zur formatierten Ein-/Ausgabe von Zeichenketten
- Spooling-System
  - Hintergrundprozesse nehmen Auftrag entgegen und führen eigentliche E/A durch
  - E/A-Geräte müssen nicht mehr exklusiv zugeteilt werden
  - Beispiele: Drucker, Mail-System

Zusammenfassung: Kontrollfluß im E/A-System



E/A–Funktionen:

E/A–Aufruf, Formatierung, Spooling

Benennung, Schutz, Puffern, Belegen

Geräteregister schreiben/lesen, Status prüfen

Treiber aktivieren, wenn E/A beendet

E/A–Operation durchführen

Programmierte E/A

- Gerätetreiber wartet nach einem Auftrag an den Controller aktiv (in einer Warteschleife) auf das Ergebnis
  - **aktives Warten** (***busy waiting***)
- Beispiel: BS-Code zur Ausgabe auf einen Drucker

Puffer aus Benutzer–Adreßraum in Kern–Adreßraum kopieren;

// buf = Puffer im Kern, count = Länge

Treiber

```
for (i=0; i<count; i++) {
    while (printer.status != READY);  // aktives Warten!
    printer.data = buf[i];            // ein Zeichen ausgeben
}
scheduler(); // Rückkehr in Benutzermodus
```

- Nachteil: ineffizient
  - CPU könnte der Zwischenzeit andere Aufgaben erfüllen

Interrupt-gesteuerte E/A

- Treiber beauftragt den Controller und kehrt sofort zurück
  - auftraggebender Thread wird ggf. blockiert
- Controller sendet Interrupt an CPU, wenn der Auftrag erledigt ist
  - d.h. Gerät ist wieder bereit
  - i.d.R.: Interrupt-Nummer identifiziert das Gerät
- Treiber behandelt die Unterbrechung
  - auftraggebender Thread wird ggf. wieder rechenbereit gesetzt
- Sinnvoll bei langsamen E/A-Geräten

Interrupt-gesteuerte E/A ...

- Beispiel: Ausgabe auf Drucker

BS–Code

Benutzer–Puffer kopieren;
// ==> buf, count Treiber

```
while (printer.status
       != READY);
printer.data = buf[0];
i = 1;
```

aktuellen Thread T blockieren;

```
scheduler();
```

Interrupt–Handler (im Treiber)

```
if (i == count) {
   Thread T bereit setzen;
} else {
   printer.data = buf[i];
   i = i+1;
}
```

Interrupt bestätigen;
Rückkehr aus Interrupt–Routine;

Direkter Speicherzugriff (*Direct Memory Access*, DMA)

- Transport der Daten zwischen Controller und Speicher erfolgt nicht durch die CPU, sondern durch separate Hardware (**DMA-Controller**)
  - oft auch in Geräte-Controller integriert
- Treiber sendet Geräte-Adresse, Startadresse und Länge der Daten, sowie Transferrichtung an DMA-Controller
- DMA-Controller erzeugt Interrupt als Fertigmeldung
- Vorteile: Entlastung der CPU, Einsparung von Interrupts
- Sinnvoll (nur) bei Übertragung größerer Datenblöcke
  - DMA-Controller arbeitet nebenläufig mit CPU
  - DMA-Controller aber oft langsamer als CPU

Direkter Speicherzugriff (*Direct Memory Access*, DMA) ...

- Beispiel: Ausgabe auf Drucker

Treiber–Code

evtl.: Benutzer–Puffer kopieren;
// ==> buf, count

Treiber

DMA–Controller aufsetzen
// Parameter: buf, count, printer

aktuellen Thread T blockieren;

`scheduler();`

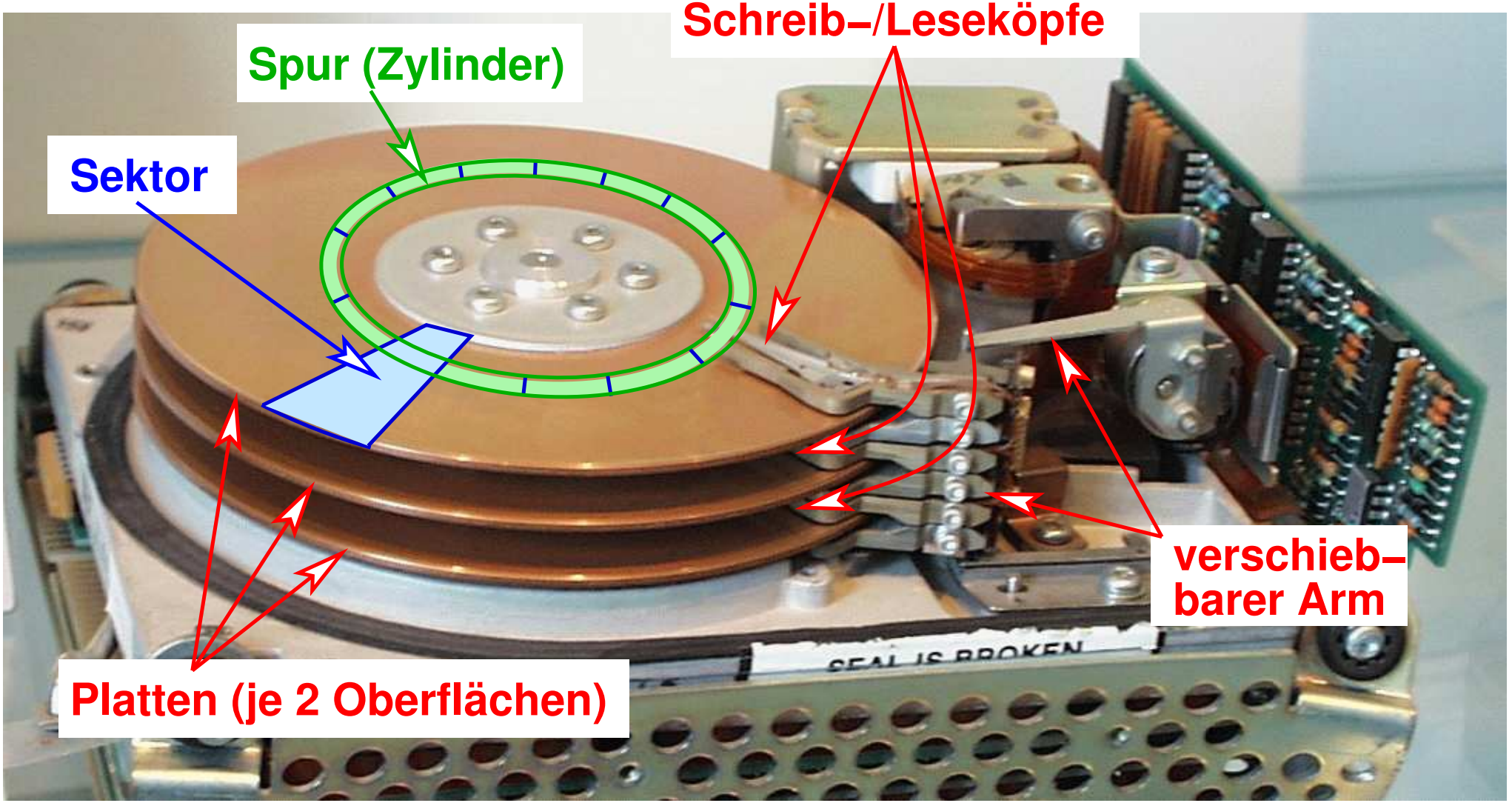
Interrupt–Handler (im Treiber)

Thread T bereit setzen;

Interrupt bestätigen;

Rückkehr aus Interrupt–Routine;

Aufbau einer Festplatte



Aufbau einer Festplatte ...

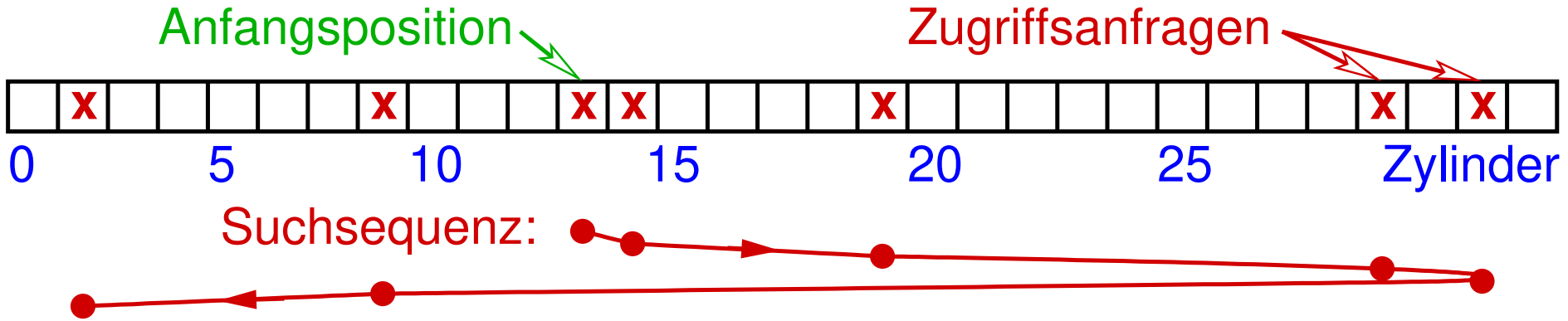
- Spurwechesel durch Verschieben des Arms (d.h. aller Köpfe)
  - Menge der jeweils übereinanderliegenden Spuren: Zylinder
- Einteilung der Festplatte (Adressierung):
  - Oberfläche (Kopf), Zylinder, Sektor
    - Sektor einer Spur nimmt einen Datenblock auf (meist 512 Byte)
  - Beispiel: physische Geometrie einer 18.3 GByte Festplatte
    - 12 Oberflächen, 10601 Zylinder, ca. 281 Sektoren
- Seit längerem: äußere Spuren besitzen mehr Sektoren als innere
  - früher: Controller zeigte dem BS eine virtuelle Geometrie an
  - heute: lineare Adressierung der Blöcke (LBA)

Zugriffszeit einer Festplatte

- Drei bestimmende Faktoren:
  - Suchzeit (Anfahren der gewünschten Spur)
    - im Durchschnitt ca. 5 - 10 ms
  - Rotationsverzögerung (bis Sektor unter dem Kopf ist)
    - im Durschnitt ca. 2 - 6 ms (5400 - 15000 U/min)
  - Dauer der Datenübertragung
    - ca. 5 - 100 μs pro Block
- Zugriffszeit dominiert durch Suchzeit, daher:
  - Dateien möglichst in aufeinanderfolgenden Sektoren
    - meist auch: *Prefetching* und *Caching*
  - geeignetes Scheduling der Plattenzugriffe

Scheduling von Plattenzugriffen

- FCFS: viele unnötige Bewegungen des Plattenarms
- SSF (*Shortest Seek First*)
  - Zugriffe in der Nähe der aktuellen Position bevorzugen
  - Problem: Unfairness, Verhungerung möglich
- Aufzug-Algorithmus
  - erst in eine Richtung, bis es in dieser Richtung keine Anfragen mehr gibt; dann Richtung wechseln

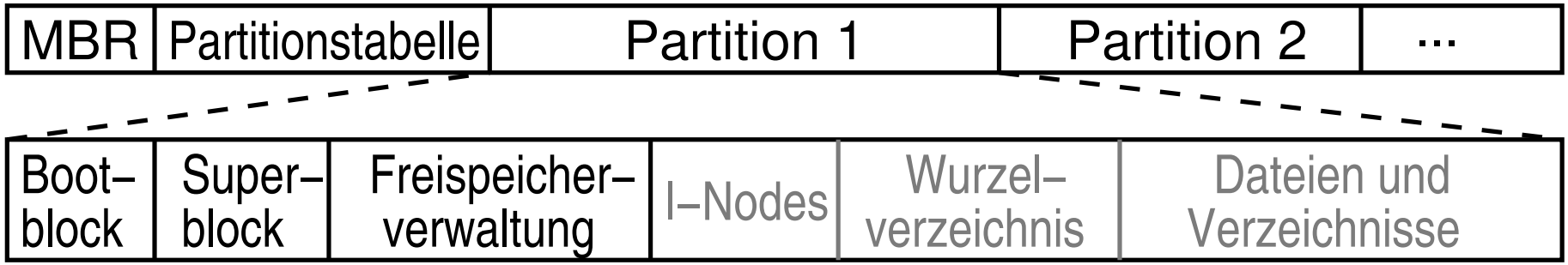


Schichtenmodell

- Datenträgerorganisation
  - einheitliche Schnittstelle zu allen Datenträgern
  - Datenträger als Folge von Blöcken betrachtet
- Blockorientiertes Dateisystem
  - Realisierung von Dateien
- Dateiverwaltung
  - Dateinamen und Verzeichnisse

Datenträgerorganisation

- Evtl. Einteilung des Datenträgers in Partitionen
- Partition wird als Folge von (logischen) Blöcken betrachtet
  - Blöcke fortlaufend numeriert
- Typisches Layout einer Festplatte (UNIX):



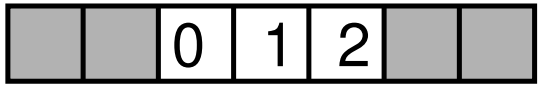
  - MBR: *Master Boot Record*
  - Superblock: Verwaltungsinformation der Partition
    - Größe, Blockgröße, ...

Datenträgerorganisation: Aufgaben

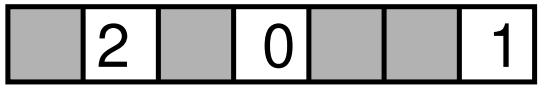
- Formatieren des Datenträgers
- Lesen / Schreiben von Blöcken
- Verwaltung freier Blöcke
  - vgl. dynamische Speicherverwaltung!
  - meist: Bitvektoren statt Freispeicherliste
    - Bit i gesetzt $\Leftrightarrow$ Block i belegt
    - Bitvektor wird auf Datenträger gespeichert
- Verwaltung defekter Blöcke
  - ebenfalls über Bitvektoren

Blockorientiertes Dateisystem

- Datei als Folge von Blöcken realisiert
- Zuteilung von Blöcken an Dateien:
  - zusammenhängende Belegung



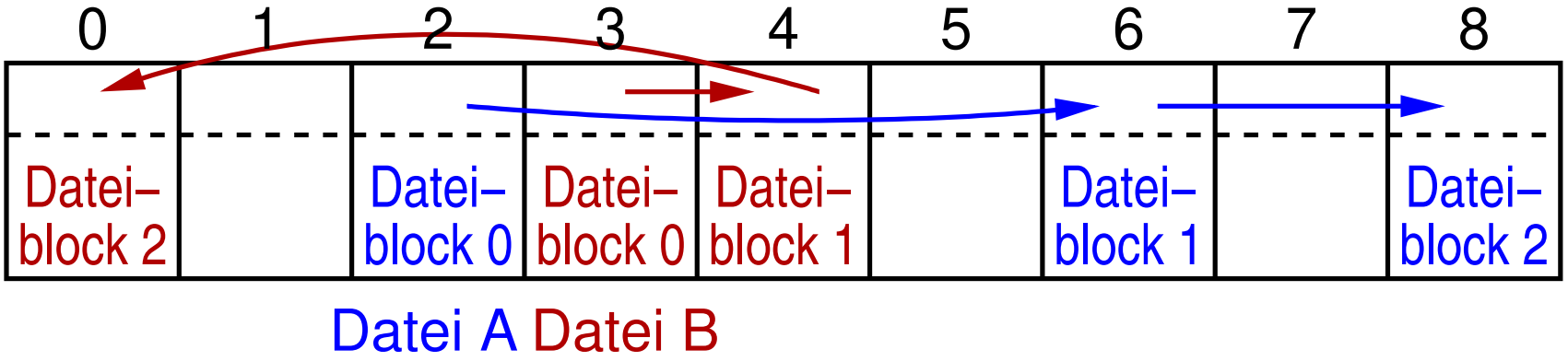
    - Datei durch Anfangsblock und Blockanzahl beschrieben
    - sehr gute Performance beim Lesen
    - Problem: Speicherverwaltung (vgl. **6.2**)
      - Anfügen an Dateien, Fragmentierung, ...
  - verteilte Belegung



    - einfache Speicherverwaltung, schlechtere Performance
    - Praxis: Belegung möglichst zusammenhängend

Blockorientiertes Dateisystem: verteilte Belegung

- Realisierung durch verkettete Listen
  - Verkettung innerhalb der Blöcke

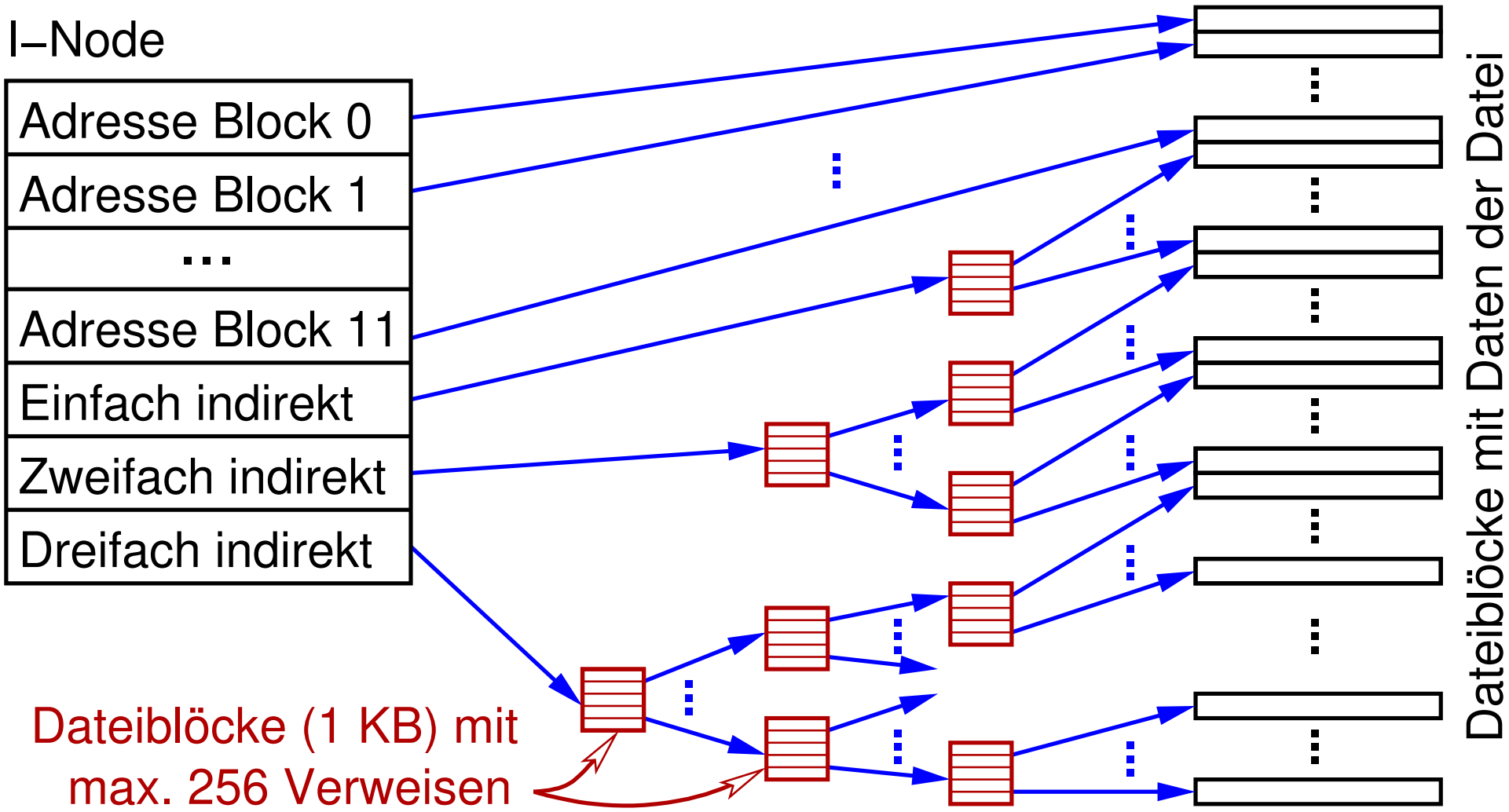


  - Nachteile: wahlfreier Zugriff ineffizient, Dateiblock-Länge keine Zweierpotenz mehr
- Realisierung durch externe Tabelle (*File Allocation Table*, FAT)
  - Verkettung ausserhalb der Blöcke
  - Tabelle kann (teilweise) im Hauptspeicher gehalten werden

Blockorientiertes Dateisystem: verteilte Belegung ...

- Realisierung durch Index-Knoten (*I-Nodes*)
  - jeder Datei ist eine Datenstruktur (*I-Node*) zugeordnet
  - *I-Node* enthält Tabelle mit Verweisen auf Dateiblöcke
    - wegen Speicherplatzbedarf: Tabelle ggf. mehrstufig
  - Tabelle kann im Hauptspeicher gehalten werden
  - schneller wahlfreier Zugriff auf Datei möglich
  - Beispiel: *I-Nodes* in UNIX
  - ähnliches Konzept auch in NTFS

Blockorientiertes Dateisystem: *I-Nodes* in UNIX



Dateiverwaltung

- Aufgabe: Realisierung von Verzeichnissen
- Verzeichnis enthält für jede Datei:
  - Dateiname, Plattenadresse (erster Dateiblock, *I-Node*), Dateiattribute
- Anmerkung: Bei Verwendung von *I-Nodes* werden die Dateiattribute im *I-Node* gespeichert
- Dateiattribute (u.a.):
  - Eigentümer, Schutzinformation (Zugriffsrechte), ...
  - Dateityp, Sperre, archiviert, ...
  - Erstellungszeit, Zeit der letzten Änderung, ...

Konzepte moderner Dateisysteme

- Vermeidung von Datenverlusten
  - bei Systemabsturz oder Stromausfall
- *Logical Volumes*
  - dynamisch veränderbare Partitionen
- *Snapshots*
  - z.B. für Backup vor einem System-Update
- *Subvolumes*
  - mehrere Sub-Dateisysteme in einer Partition
- Deduplizierung
  - Vermeidung von Redundanz

Vermeidung von Datenverlusten

- Aus Performanzgründen: Datenblöcke werden zunächst nur im Hauptspeicher verändert
  - periodisches Rückschreiben auf Festplatte ca. alle 30s
- Bei Absturz oder Stromausfall: Änderungen gehen verloren
  - dadurch evtl. Struktur des Dateisystems inkonsistent
  - ggf. Reparatur durch Dateisystem-Check
- Bessere Lösung: *Journaling*-Dateisystem
  - Journal beschreibt alle Änderungen im Dateisystem
  - sequentiell auf Platte gespeichert (keine Kopfbewegung)
  - von Zeit zu Zeit: Ausführen der Operationen und Löschung im Journal

Vermeidung von Datenverlusten ...

- Bei Absturz oder Stromausfall:
  - Operationen im Journal werden beim Neustart ausgeführt
- Anwendung z.B. in Linux `ext4` und Windows NTFS
- Details zu `ext4`
  - Journal ist zirkulärer Puffer
  - Schreib-/Leseoperationen über eigenes Gerät (JBD)
  - JBD stellt sicher, daß zusammenhängende Journal-Einträge atomar gespeichert werden
  - Verwendung des Journals i.a. nur für Metadaten
    - aber auch für alle Daten möglich

Logical Volumes

- Abstraktion der physischen Datenträger (Partitionen)
- *Logical Volume* kann sich über mehrere physische Partitionen erstrecken
  - auch dynamisch erweiterbar
  - realisiert durch Schicht oberhalb Datenträgerorganisation
    - Abbildung von logischem Block auf Partition und Block
- Beispiele: LVM (Linux), Dynamische Datenträger (Windows)
- Oft: zusätzliche (Software-)RAID-Funktion
  - RAID 1 (*Mirroring*): Daten werden zweifach gespeichert
  - RAID 5: Verteilung + Redundanz durch Paritätsbildung
    - Ausfall einer von N Platten kann toleriert werden

Snapshots, Subvolumes, Deduplizierung

- Zugrundeliegende Technik: *Copy-on-Write*
- Z.B. bei Erstellung eines *Snapshots*
  - Daten werden nicht kopiert, nur die Verweise auf die Plattenblöcke (*reflink*)
  - bei Änderung eines Blocks: Block wird kopiert, Verweis wird aktualisiert
  - damit: Kopie sehr schnell und platzsparend
- Snapshots i.d.R. erstellt bei Software-Updates
- Alter Snapshot kann als *Subvolume* direkt gemounted werden
- Deduplizierung: ersetze Block durch *reflink*, falls möglich
  - dazu: Prüfsummen aller Blöcke vergleichen

9.6 Zusammenfassung / Wiederholung

- Schichten der E/A-Software
  - Benutzer-E/A-Software, geräteunabhängige BS-Software, Gerätetreiber, Unterbrechungsroutinen
  - Treiber: geräteabhängig, zur Laufzeit in BS-Kern geladen
- Durchführung der E/A: programmierte E/A, Interrupts, DMA
- Festplatten: eingeteilt in Oberfläche, Zylinder, Sektor
- Aufbau von Dateisystemen: Schichtenmodell
  - Datenträgerorganisation
    - Freispeicherverwaltung
  - Blockorientiertes Dateisystem (Datei = Menge von Blöcken)
    - zusammenhängende / verteilte Belegung von Blöcken
  - Dateiverwaltung
    - Verzeichnis: Name, Adresse, Attribute für jede Datei

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

10 Schutz

Inhalt:

- Einführung
- Schutzmatrix
- Zugriffskontrolllisten und *Capabilities*

- Tanenbaum 9.6
- Stallings 15.2.3
- Nehmer/Sturm 11

Sicherheitsdienste: AAA („*triple A*“)

- **Authentifizierung**
  - Feststellung der Identität eines Benutzers
  - typisch: Paßwort-Abfrage bei der Anmeldung
- **Autorisierung**
  - Vergabe von Zugriffsrechten an Benutzer
  - Wer darf was im System tun?
- **Accounting**
  - Protokollierung von Aktivitäten, Abrechnung

10.2 Schutzmatrix

- Legt fest, wer welche Operationen auf welchen Objekten ausführen darf

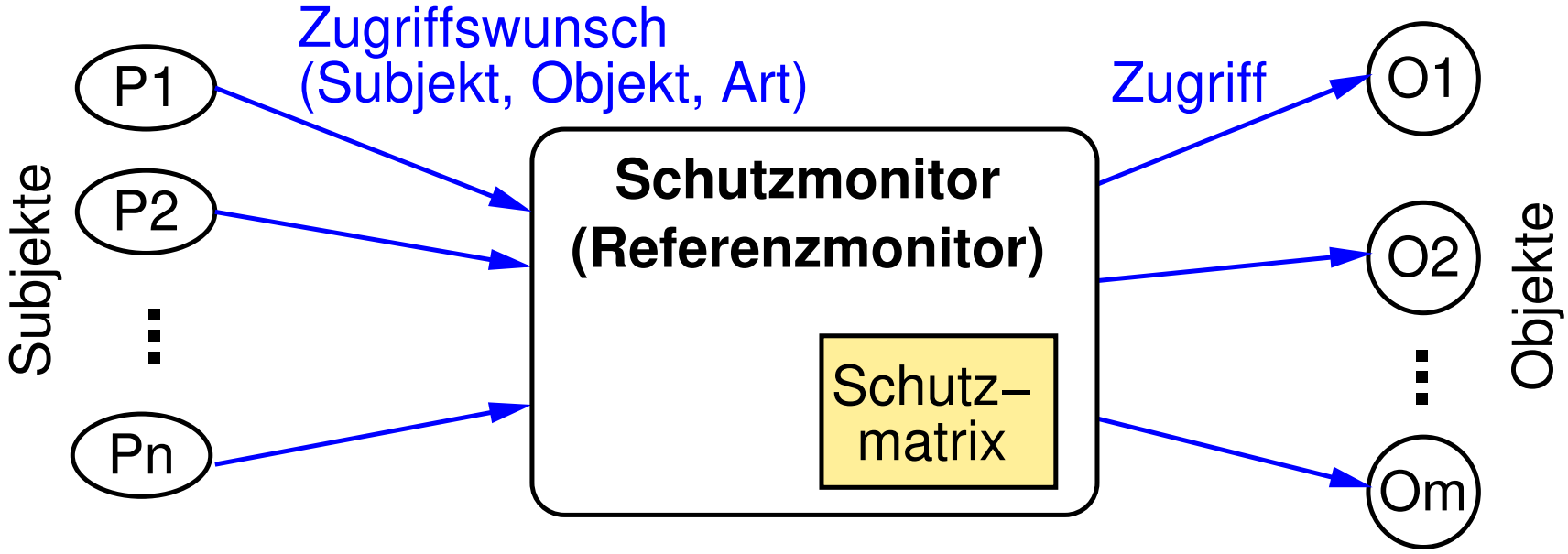
Objekte →	Datei 1	Datei 2	Datei 3	Datei 4	Datei 5	Drucker	Plotter
Benutzer 1	Lesen	Lesen Schreiben					
Benutzer 2		Lesen	Lesen	Lesen Schreiben Ausführen		Schreiben	
Benutzer 3		Lesen		Lesen Ausführen	Lesen Schreiben Ausführen	Schreiben	Schreiben

Subjekte

Rechte

- **Subjekt**: z.B. Benutzer(gruppe), Prozeß eines Benutzers
- **Objekt**: z.B. Datei, Gerät, (anderer) Prozeß
- **Recht**: Erlaubnis zur Durchführung einer Operation

Realisierung des Zugriffsschutzes: Schutzmonitor



- Subjekte dürfen Indentität nicht wechseln können
- Zugriff auf Objekte darf nur über Schutzmonitor möglich sein
- Schutzmonitor muß privilegiert sein, um Zugriffe durchzuführen
- Schutzmonitor muß vertrauenswürdig sein

Speicherung der Schutzmatrix

- Schutzmatrix sehr groß und sehr dünn besetzt
- Daher: zeilen- oder spaltenweise Speicherung in Listen

	Datei 1	Datei 2	Datei 3	Datei 4	Datei 5	Drucker	Plotter
Benutzer 1	Lesen	Lesen Schreiben					
Benutzer 2		Lesen	Lesen	Lesen Schreiben Ausführen		Schreiben	
Benutzer 3		Lesen		Lesen Ausführen	Lesen Schreiben Ausführen	Schreiben	Schreiben

Capability

Zugriffskontrollliste

***Access Control List* (ACL)**

Zugriffskontrollliste (*Access Control List*, ACL)

- Spalte der Schutzmatrix
- Gibt für ein Objekt an, welche Subjekte welche Rechte an dem Objekt haben
- Wird zusammen mit dem betroffenen Objekt gespeichert
  - z.B. bei Datei im zugehörigen *I-Node*
- Listenelemente: Paare (Subjekt, Rechte)
  - Subjekt: Benutzer und/oder Benutzergruppe
  - für Subjekt oft auch Platzhalter (*Wildcard*) erlaubt
    - erster passender Eintrag wird verwendet

Capability

- Zeile der Schutzmatrix
- Wird vom BS-Kern an Subjekte (Prozesse) übergeben, berechtigt zur Ausführung von Operationen auf Objekten
- *Capability* muß vor Manipulation geschützt werden!
  - Speicherung im BS-Kern, Prozeß erhält nur Verweis (*Handle*)
  - kryptographischer Schutz (analog zu digitaler Signatur)
    - geeignet für verteilte Systeme: *Capability* kann als Nachricht weitergegeben werden
- Problem: (selektiver) Widerruf von Rechten schwierig

Beispiel: Zugriffsschutz in UNIX und Windows (2000/NT)

- Mischform aus ACLs und *Capabilities*
  - Rechte an Objekten werden über ACL spezifiziert
  - Prüfung der ACL aber nur beim Öffnen
    - Datei-/Geräte-*Handle* hat die Funktion einer *Capability*
    - bei den eigentlichen Zugriffen ist keine Prüfung der ACL mehr notwendig
- UNIX: ACL unterscheidet bei Subjekten nur zwischen Eigentümer, Mitgliedern derselben Gruppe und allen anderen
  - Speicherung in 9 Bits im *I-Node*: `rwx r-x ---` (user group others)
- Windows: ACL mit Einträgen für beliebige Benutzer / Gruppen

10.4 Zusammenfassung / Wiederholung

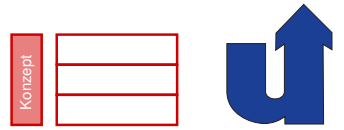
- Sicherheitsdienste: Authentifizierung, Autorisierung, Accounting
- Schutzmatrix:
  - wer darf welche Operationen auf welchen Objekten ausführen?
- Zugriffskontrollliste (*Access Control List*, ACL)
  - Spalte der Schutzmatrix, beim Objekt gespeichert
- *Capability*
  - Zeile der Schutzmatrix, an Subjekt übergeben

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

11 Virtualisierung

11.1 Virtualisierungstypen



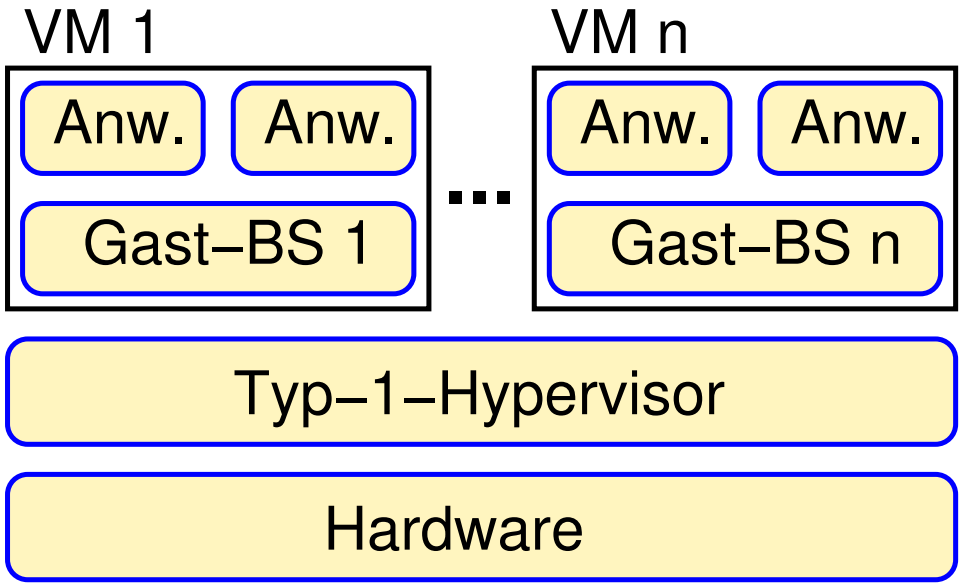
- Virtuelle Prozessoren: Emulator, z.B. QEMU, JVM
- Virtuelle Prozessumgebungen: normales BS
- Virtuelles BS: ABI für BS X auf BS Y, z.B. WINE
- Virtueller Desktop: z.B. RDP, VNC
- Virtuelle Ressourcen
  - z.B. *Storage Area Network* (SAN), virtuelle Netzwerke
- Virtuelles Laufzeitsystem (Sandboxing)
  - *Container*
- Virtuelle Computer (virtuelle Maschine)
  - wirkt (auch für BS) wie realer Computer

Motivation

- Server (WWW, Mail, ...) laufen i.a. auf eigenen Rechnern
  - sicherer und zuverlässiger
  - unterschiedliche Anforderungen an Systemsoftware / BS
- Statt einzelner Rechner: virtuelle Maschinen auf einem Rechner
  - kostengünstiger / energiesparender
  - Migration virtueller Maschinen (VMs) zwischen Rechnern möglich
  - Erneuerung der Hardware problemlos(er)
- Realisierung durch *Hypervisor* (*Virtual Machine Monitor*, VMM)
  - spielt BS auf VM vor, dass es direkt auf der Hardware läuft
  - VMs sind logisch getrennt; ggf. auf jeder VM anderes BS

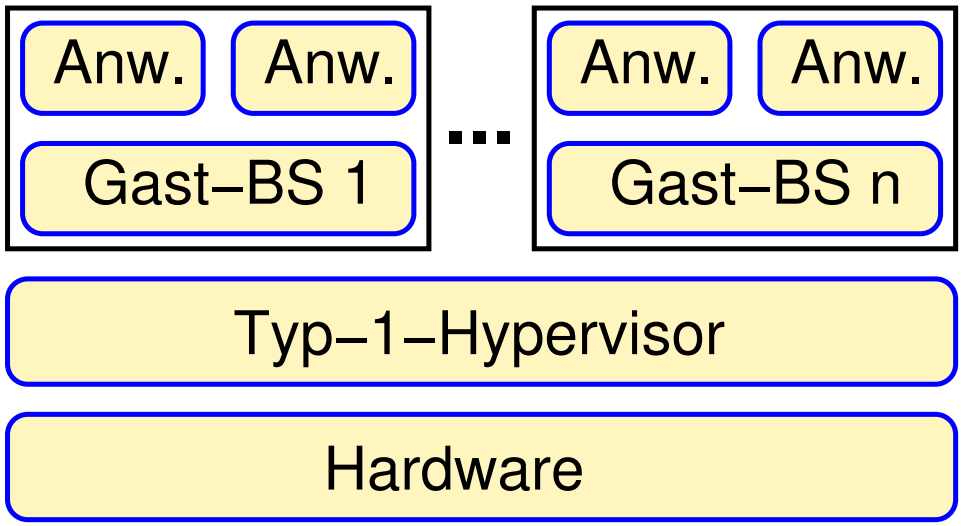
Schichtenmodell

Typ–1–Hypervisor

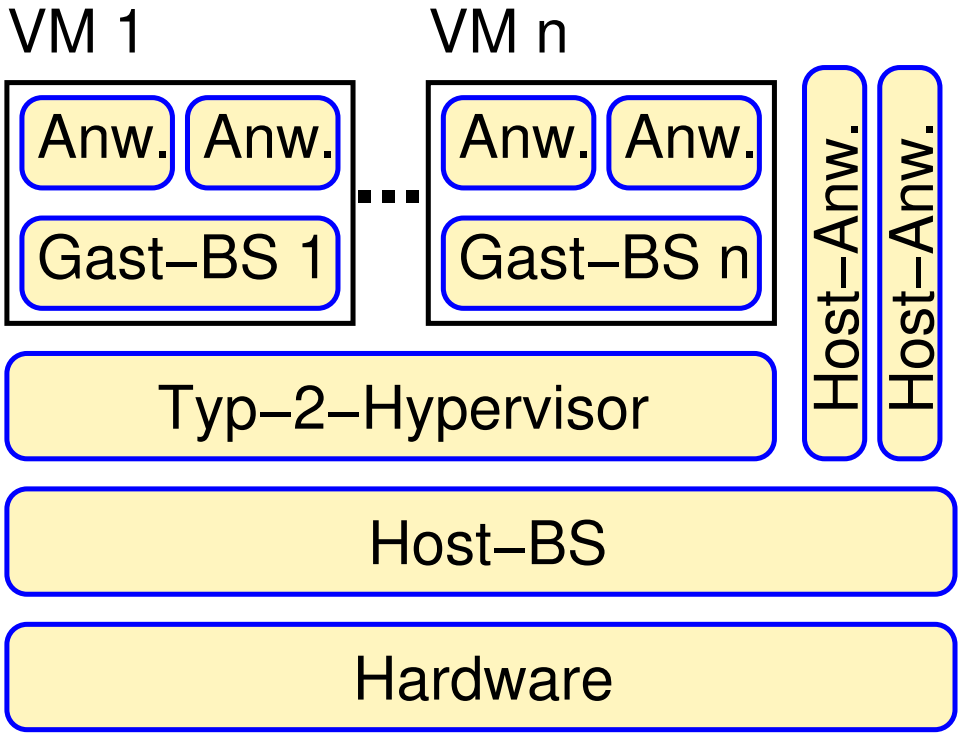


Schichtenmodell

Typ–1–Hypervisor



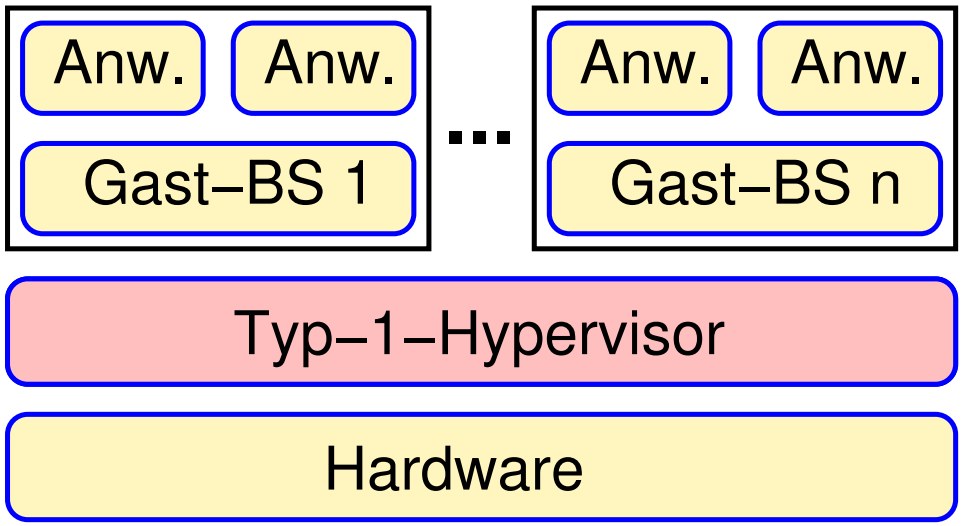
Typ–2–Hypervisor



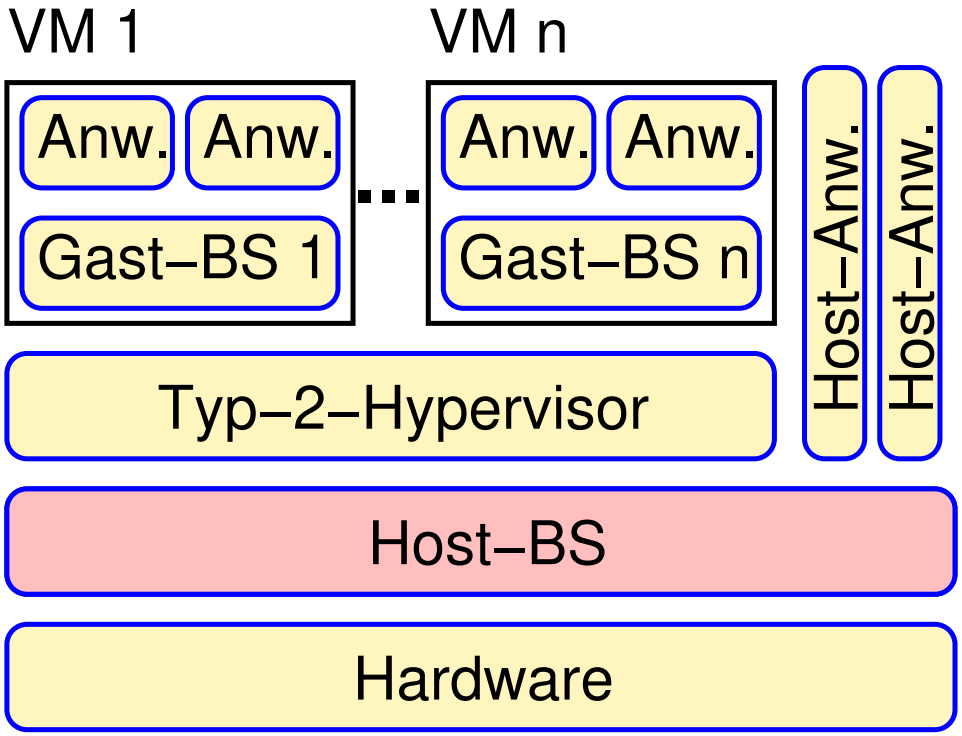
Mischform: Hypervisor im Host–BS integriert

Schichtenmodell

Typ–1–Hypervisor



Typ–2–Hypervisor

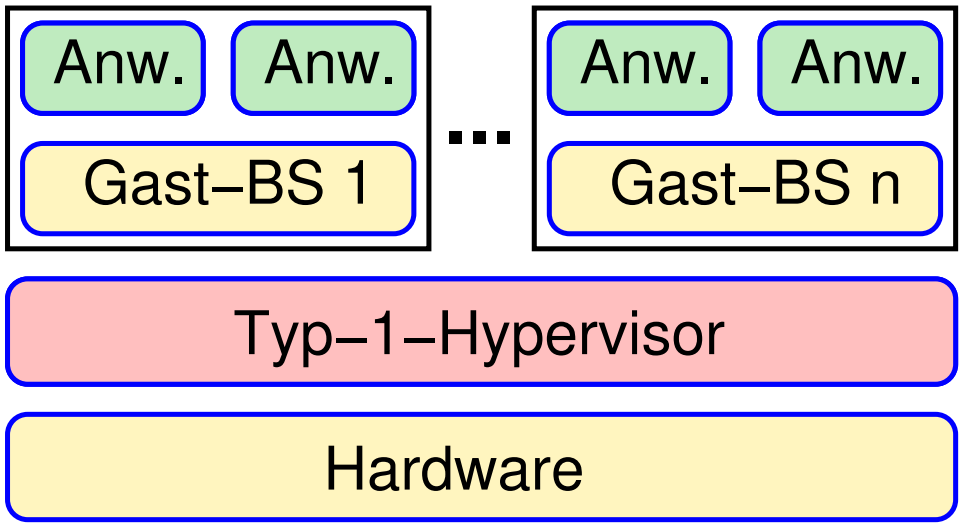


Systemmodus

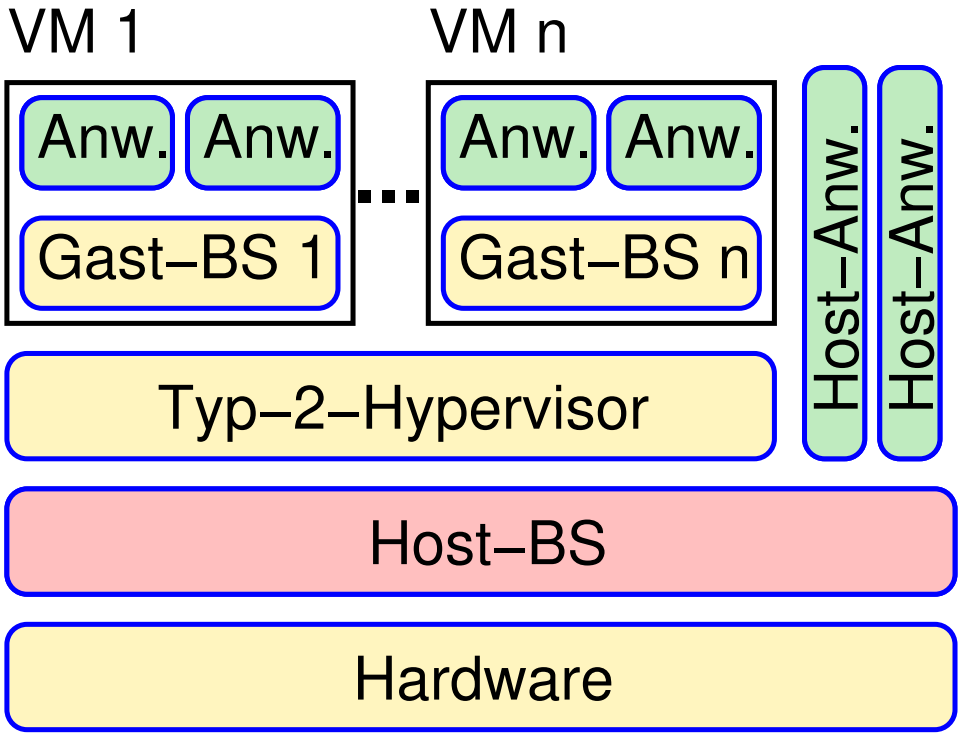
Mischform: Hypervisor im Host–BS integriert

Schichtenmodell

Typ–1–Hypervisor



Typ–2–Hypervisor



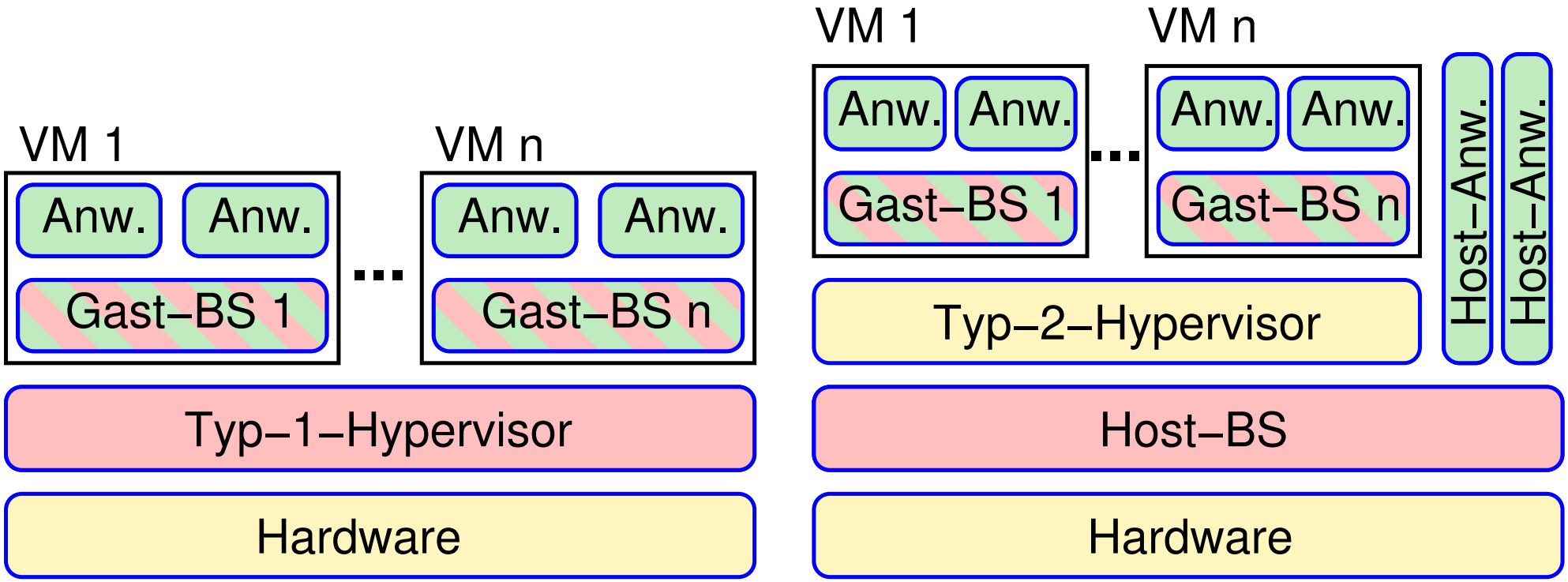
Systemmodus Benutzermodus

Mischform: Hypervisor im Host–BS integriert

Schichtenmodell

Typ–1–Hypervisor

Typ–2–Hypervisor

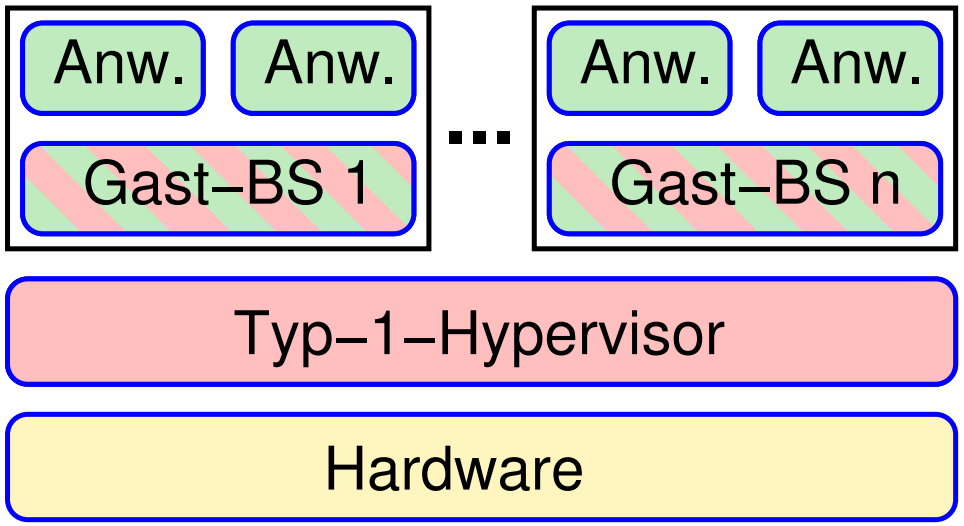


Systemmodus Benutzermodus Virtueller Systemmodus

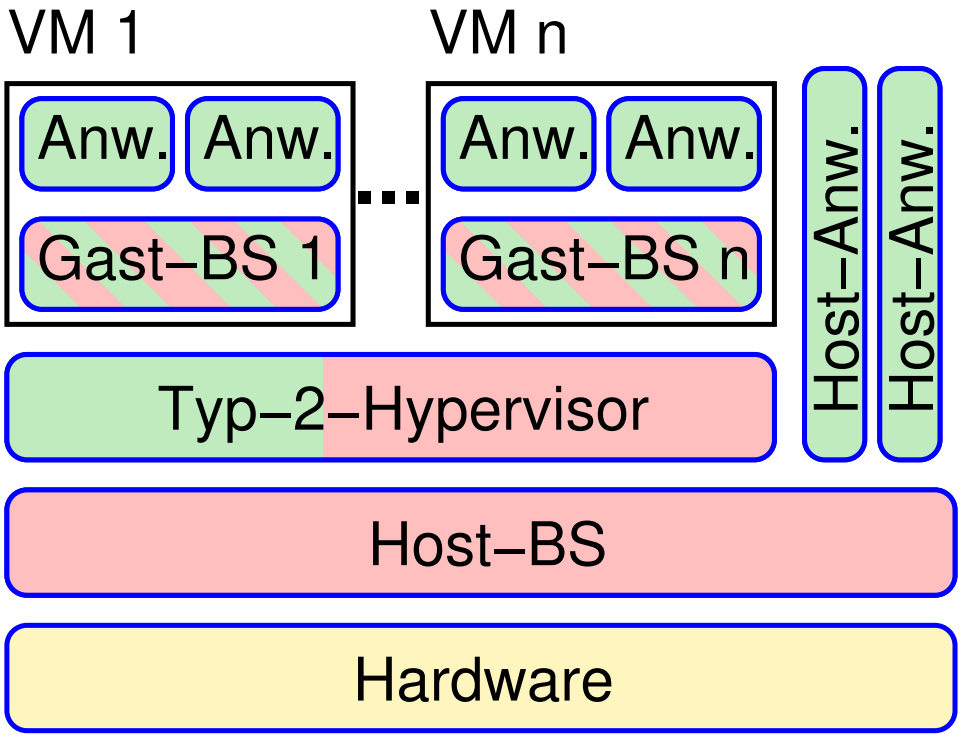
Mischform: Hypervisor im Host–BS integriert

Schichtenmodell

Typ–1–Hypervisor



Typ–2–Hypervisor



Systemmodus Benutzermodus Virtueller Systemmodus

Mischform: Hypervisor im Host–BS integriert

Beispiele

- Typ-I-Hypervisor
  - VMware ESX
- Typ-II-Hypervisor
  - VMware Workstation, VirtualBox, Xen
  - Xen kann durch eigenes Linux auch direkt auf HW aufsetzen
- In BS integriert:
  - KVM (Linux), Hyper-V (Windows)

Realisierung

- Gast-BS muss im Benutzermodus ausgeführt werden
  - Problem: Ausführung **sensitiver Befehle**
    - Verhalten im Benutzer- und Systemmodus unterschiedlich
- Sensitive Befehle müssen vom Hypervisor ausgeführt (emuliert) werden
- Einfach, falls alle sensitiven Befehle auch privilegiert sind
  - dann: Trap zum Hypervisor, dort Emulation
- Bei vielen (älteren) Prozessoren nicht der Fall (z.B. Pentium)
  - Lösung: Binärübersetzung des Codes
  - Alternative: **Paravirtualisierung**
    - Gast-BS wird angepasst

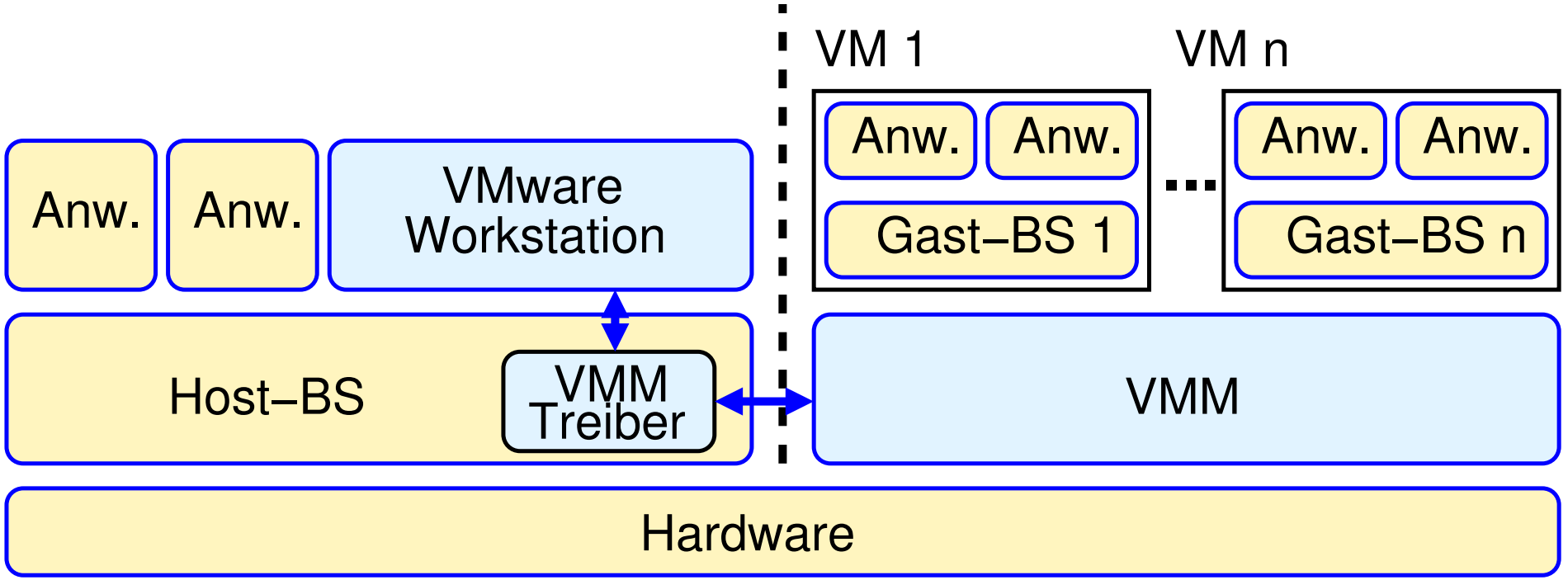
Binärübersetzung

- Idee: Code, der im Gastsystem im Systemmodus laufen muß, wird ersetzt
  - der neue Code beinhaltet keine sensitiven Befehle mehr
- Ersetzung erfolgt bei Bedarf auf Ebene von Basisblöcken
  - Basisblock: lineare Befehlsfolge, höchstens ein Sprungziel am Anfang, höchstens ein (i.a. bedingter) Sprung am Ende
  - Hypervisor hält einen Cache mit bereits übersetzten Basisblöcken
  - am Ende eines Blocks: Rückkehr zum Hypervisor
- Wird aus *Performance*-Gründen z.T. auch bei virtualisierbaren Prozessoren eingesetzt

Typ-2-Hypervisor

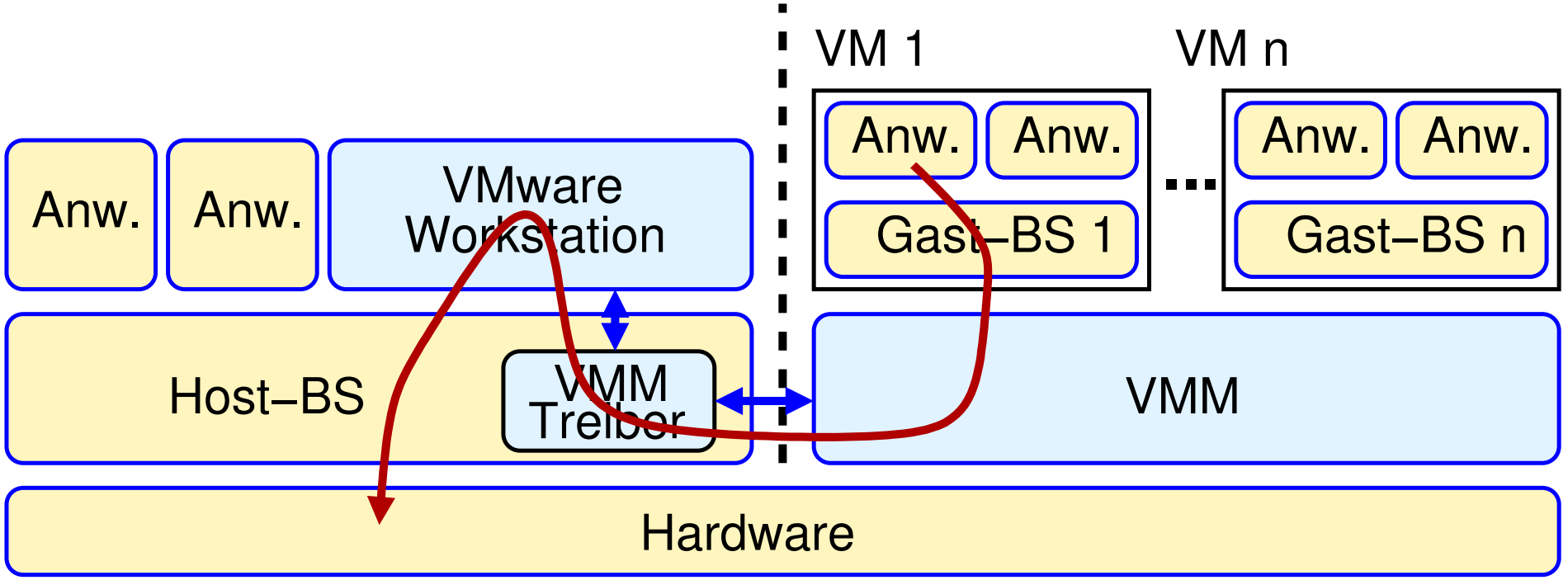
- Typ-2-Hypervisor ist „normale“ Anwendung
  - kann so aber CPU und Speicher nicht multiplexen
  - Hypervisor muss teilweise im Systemmodus laufen
- Daher: Aufspaltung in Anwendung, Gerätetreiber und VMM
- Anwendung: Bedienoberfläche, Realisierung von E/A über Host-BS
- Gerätetreiber: ermöglicht Ausführung des VMM im Systemmodus, emuliert Interrupts
  - *World-Switch*: Umschaltung zwischen Host-BS und VMM
- VMM: Multiplexing von CPU und Speicher, Binärübersetzung, Unterbrechungsbehandlung

Typ-2-Hypervisor: Beispiel VMware



- Ausgeführt wird entweder Host-BS oder VMM
  - *World-Switch* durch VMM-Treiber sichert Zustand des Host-BS

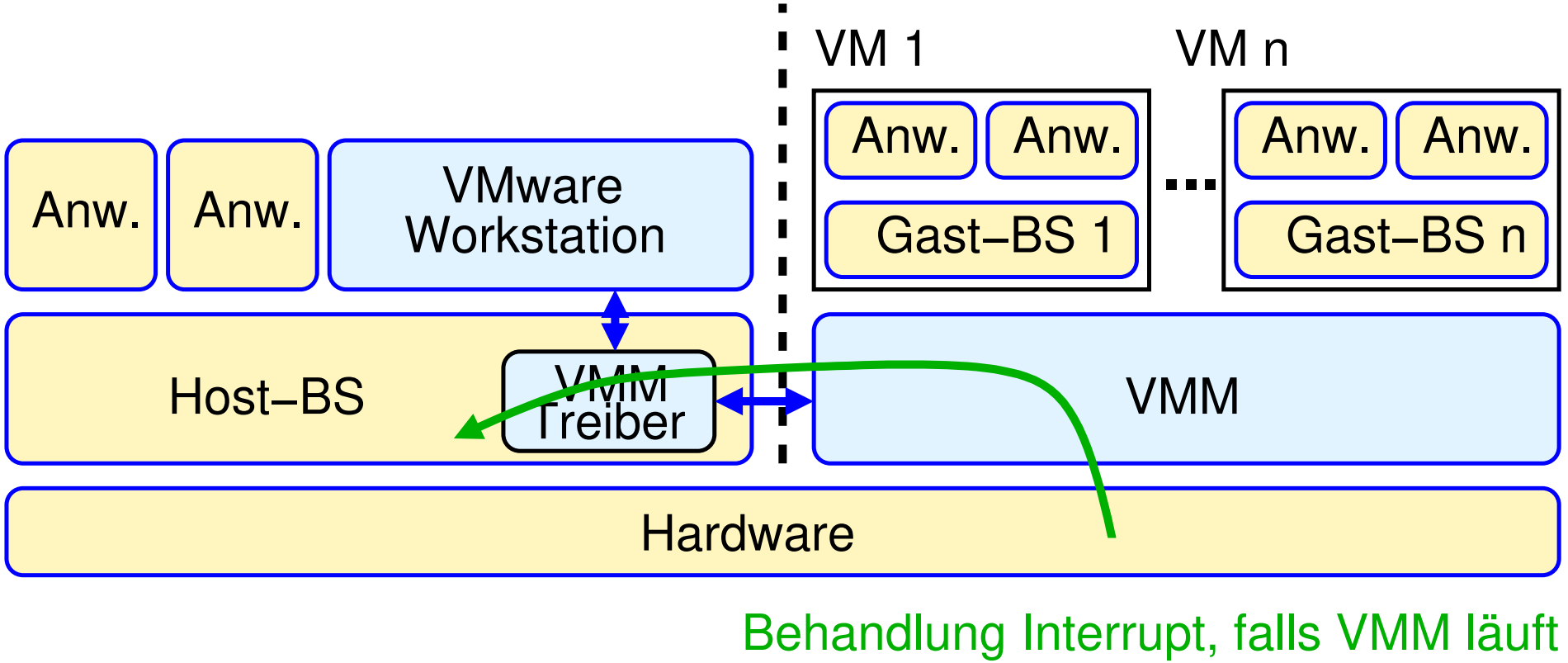
Typ-2-Hypervisor: Beispiel VMware



Routing einer E/A–Anfrage

- Ausgeführt wird entweder Host-BS oder VMM
  - *World-Switch* durch VMM-Treiber sichert Zustand des Host-BS

Typ-2-Hypervisor: Beispiel VMware



- Ausgeführt wird entweder Host-BS oder VMM
  - *World-Switch* durch VMM-Treiber sichert Zustand des Host-BS

Speichervirtualisierung

- Problem: jedes Gast-BS erstellt Seitentabellen unabhängig
  - dabei ggf. Abbildung auf dieselben Kacheln
- Hypervisor muß die Kacheln ggf. ändern
  - dazu: Schattentabellen im Hypervisor
- Problem: Hypervisor bekommt nur das Laden der Seitentabelle in die MMU mit, nicht aber spätere Veränderungen
- Mögliche Lösungen:
  - Seiten der Seitentabelle schreibschützen
  - Hinzufügen neuer Kacheln erst bei Seitenfehler
    - Löschen von Seiten mit privilegiertem Befehl (wegen TLB)
  - Hardwareunterstützung für verschachtelte Seitentabellen

Speichervirtualisierung ...

- Häufig: *Overcommitment*
  - alle VMs zusammen haben mehr Speicher als der Host
- Problem: Auslagerung durch Hypervisor nicht effizient / sinnvoll
  - unklar, welche Seiten für Gast-BS wichtig sind
  - Bei Auslagerung durch Gast-BS muss Hypervisor Seite ggf. erst wieder einlagern
- Lösung: *Balloon*-Treiber
  - im Gast-BS, kann ggf. fixierte Seiten allokieren
    - (fixierte Seiten können nicht ausgelagert werden)
  - Gast-BS muss dafür „normale“ Seiten auslagern
- Z.T. auch Reduktion des Speicherverbrauchs durch Deduplikation

E/A-Virtualisierung

- Gast-BS bekommt virtuelle (Standard-)Geräte
  - z.B. virtuelle SATA-Platte, realisiert als normale Datei
  - für Netzwerk realisiert Hypervisor ggf. auch einen virtuellen Switch
- Heute auch Geräte mit Hardware-Unterstützung für Virtualisierung
  - Single-Root-I/O-Virtualisierung (SR-IOV)
  - Gerätecontroller bieten jeder VM eine eigene Schnittstelle (virtuelle Funktionen)
    - keine Mitwirkung des Hypervisors nötig

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

12 Zusammenfassung, wichtige Themen

12.1 „Heiße“ Themen für die Klausur

- **Prozesse und Threads**
  - **Zustandsgraph**
  - Elemente des Prozeß- bzw. Thread-Kontrollblocks
  - Ablauf von Interrupt, Ausnahme, Systemaufruf
  - **Threadwechsel**
- **Synchronisation**
  - Kritischer Abschnitt, wechselseitiger Ausschluß
  - (Lösungen mit Schreib-/Leseoperationen, *Spin locks*)
  - **Semaphore**
  - **Monitore**
  - **Java *Locks* und Bedingungsvariable** (nur 6 LP)
  - [Code angeben bzw. gegebener Code mit Fragen]

12.1 „Heiße“ Themen für die Klausur ...

- (Kommunikation)
- **Verklemmungen**
  - Definition und Bedingungen
  - *Deadlock*-Erkennung, v.a.: **Algorithmen**
  - *Deadlock-Avoidance*, v.a.: **sichere Zustände, Bankiers-Alg.**
  - *Deadlock-Prevention*
- **Scheduling**
  - präemptiv, nicht-präemptiv
  - **FCFS, SJF, RR, Prioritäten, Multilevel-Scheduling**

12.1 „Heiße“ Themen für die Klausur ...

- **Speicherverwaltung**
  - logischer / physischer Adreßraum
  - Speicherschutz
  - Zuteilung zusammenhängender Speicherbereiche
    - Swapping, Relokation, (dynamische Speicherverwaltung)
  - **Paging**
    - **Prinzip**; Seiten, Kacheln, Seitentabelle
    - **Ablauf der Umsetzung**
      - ein- und zweistufige Seitentabelle
    - virtueller Speicher
      - Grundlagen: Lokalität, *Working Set, Resident Set*
      - **Ablauf eines Seitenwechsels**
      - **Seitenersetzungsalgorithmen**: Belady, **NRU, FIFO, *Second Chance* / *Clock*, LRU**

12.1 „Heiße“ Themen für die Klausur ...

- Ein-/Ausgabe und Dateisysteme
  - Programmierte E/A, Interrupt-gesteuerte E/A, DMA
  - (Schichten der E/A-Software)
  - (Festplatten: Aufbau, Zugriffszeit)
  - (Schichten des Dateisystems)
  - Zuteilung von Blöcken an Dateien: verteilte Belegung
- Schutz
  - Schutzmatrix, ACL, Capability

Nicht in der Klausur

- 1.6 Systemaufrufe
- 2.2 Nebenläufige Programmierung
- 3.5 Synchronisation in Mehrprozessorsystemen
- 3.6 Speicherkonsistenz
- 3.11 Lock-free Datenstrukturen
- 3.12 Transactional Memory
- 6 Koroutinen und asynchrone Programmierung
- 7.5 Scheduling: Beispiele
- 9.5 Moderne Dateisysteme
- 11 Virtualisierung

Nicht in der Klausur ...

- Programmierung im Detail
  - Shell-Programmierung
  - Thread-Programmierung mit C, C++, Linux
    - Abschnitte 2.7.2 - 2.7.4 und 3.10.3 - 3.10.5
  - konkrete Java-Programmierung
    - **der Umgang mit *Locks* und Bedingungsvariablen ist aber Klausurstoff (für 6 LP)!**