Zusammenfassung der Vorlesung

„Systemprogrammierung“ im WS 2009/10

zusammengestellt von Michael Fiedler <simified@stud.informatik.uni-erlangen.de>

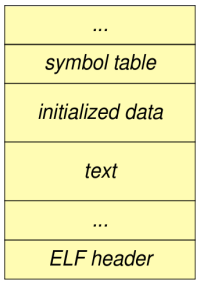
Inhaltsverzeichnis

1. Organisation

2. Einführung in C

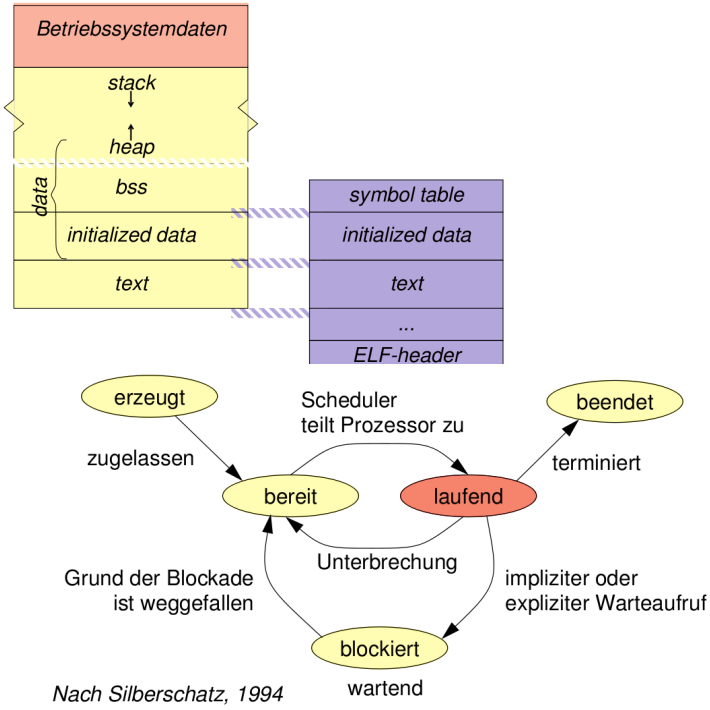
3. Programm ↦ Prozess

- Speicherorganisation eines Prozesses, definiert durch ELF-Format – wichtigste Elemente



  - ELF header: Identifikator und Verwaltungsinformationen
  - text: Programmcode
  - initialized Data: initialisierte globale und *static* Variablen
  - symbol table: Zuordnung der im Programm verwendeten symbolischen Namen von Funktionen und globalen Variablen zu Adressen (→ u. a. Debugger)

- Speicherorganisation eines Prozesses



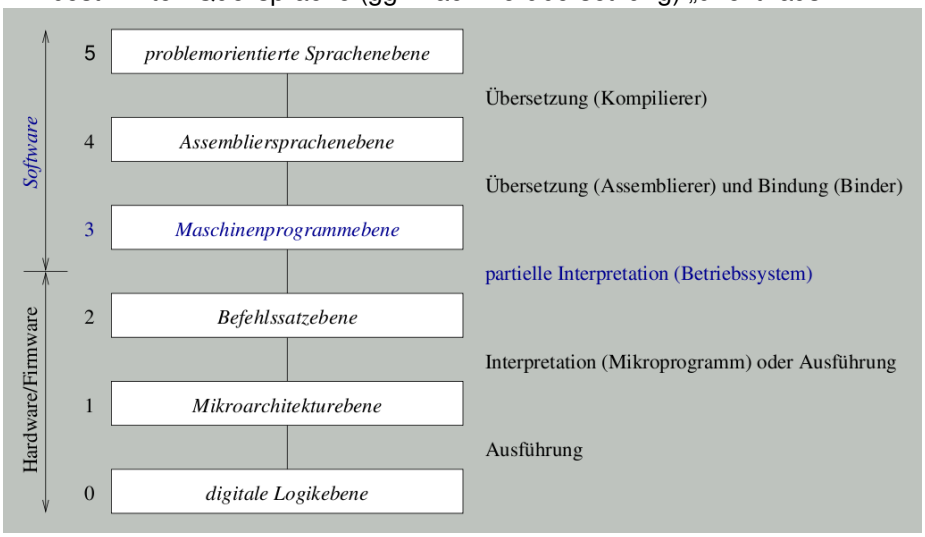
Nach Silberschatz, 1994

4. Einleitung

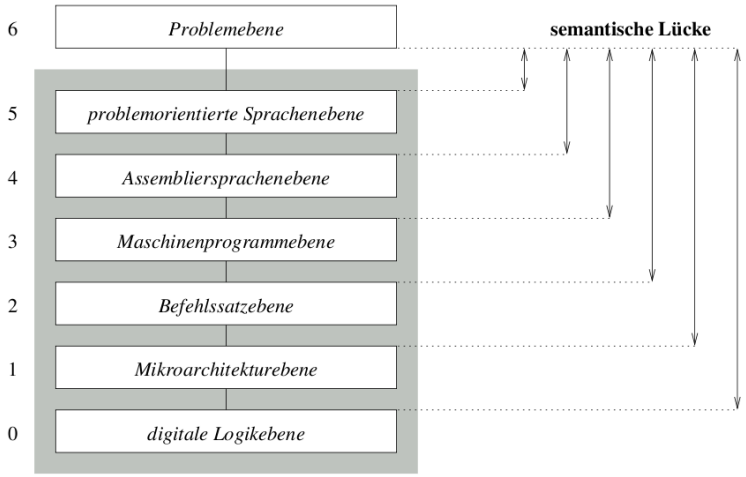
- Fallstudie Arraydurchlauf byRow/byColumn
  - funktionale Eigenschaft: „was“ etwas tut
    → byRow, byColumn tun das Gleiche
  - nicht-funktionale Eigenschaft: „wie“ sich etwas ausprägt
    → byRow: Aufzählung Feldelemente entsprechend Feldabspeicherung; für gegebene Hardware günstigere Zugriffsmuster; kann schneller als byColumn ablaufen

5. Rechnerorganisation

- semantische Lücke: Unterschied zwischen den komplexen Operationen, die von Hochsprachenkonstrukten durchgeführt werden, und den einfachen, die von den Befehlssatzarchitekturen von Computern angeboten werden. Ein Versuch, diese Lücke zu schließen, war es, Computer mit zunehmend komplexeren Befehlssatzarchitekturen zu entwerfen.
  → Kluft zwischen gedanklich Gemeintem und sprachlich Geäußertem
  - Abstraktion → Konzentration auf Wesentliches (virtuelle Maschinen/abstrakte Prozessoren), schrittweise Abbildung der Problemlösung auf reale Maschine
  - Abbildung durch Programme
    - Kompilator: Softwareprozessor, Quellsprache → Zielsprache
    - Interpreten: in Hard-/Firm-/Software realisierter Prozessor führt Programme einer bestimmten Quellsprache (ggf. nach Vorübersetzung) „direkt“ aus

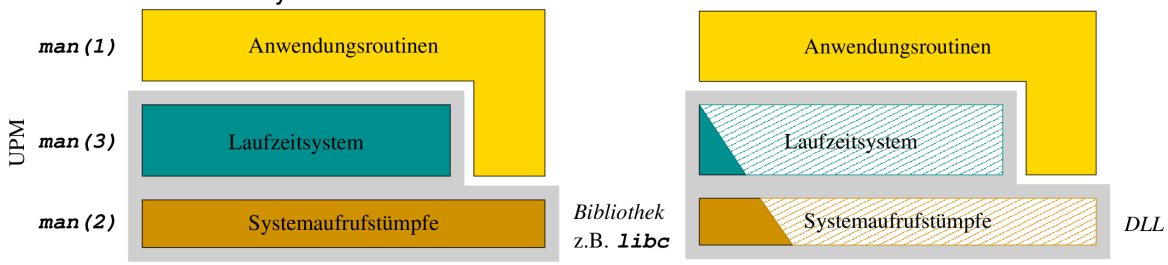


  - zur Maschinenprogrammebene:
    - Programme bestehen aus Systemaufrufen (Betriebssystem) und Maschinenbefehlen (ISA)
    - Betriebssysteme „ko-implementieren“ Maschinenprogrammebenen
  - zur Befehlssatzebene: implementiert das Programmiermodell der CPU → Mikroanweisungen, Konstrukte einer Hardwarebeschreibungssprache
  - zur Mikroarchitekturebene: beschreibt Aufbau der Operations- und Steuerwerke, der Zwischenspeicher und die Befehlsverarbeitung → Programme: Konstrukte einer Hardwarebeschreibungssprache
  - zur digitalen Logikebene: wirkliche Hardware des Rechners auf Basis von Transistoren, Gattern, Schaltnetzen/-werken → Programme: Elemente der Schaltalgebra
  - Abstraktionsniveau versus semantische Lücke



  - Abbildung implementiert durch Prozessoren:

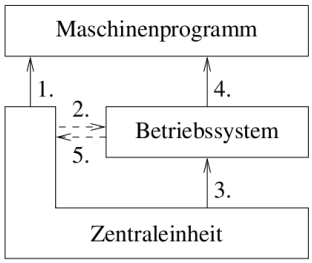
    - Ebene 5 → Kompilator
    - Ebene 4 → Assemblierer, Binder
    - Ebene 3 → Betriebssystem
    - Ebene 2 → Zentraleinheit (CPU)
- Systemaufrufschnittstelle/system call interface
    - Aufrufstümpfe verbergen technische Auslegung der Interaktion zwischen Anwendungsprogramm und Betriebssystem
        - „nach außen“: Systsemaufruf als normaler Prozeduraufruf
        - „nach innen“: Systemaufruf setzt (synchrone) Programmunterbrechung ab
    - Systemaufrufe als spezielle „Prozedurfernaufrufe“, die ggf. bestehende Schutzdomänen in kontrollierter Weise überwinden müssen
        - getrennte Adressräume für Anwendungsprogramm und Betriebssystem
        - Ein-/Ausgabeparameter in Registern übergeben, „Trap“ auslösen
- Laufzeitumgebung/runtime environment: Programmbausteine in Form eines zur Laufzeit zur Verfügung gestellten universellen Satzes von Funktionen und Variablen
- Organisation von Maschinenprogrammen
    - Anwendungsprogramm gebildet aus (zusammengebunden)
        - Anwendungsroutinen (des Rechners)
          main() und andere Selbstgebautes → Absetzen von Betriebssystem- oder Laufzeitsystemaufrufen
        - Laufzeitsystem (des Kompilierers/Betriebssystems)
          z. B. Bibliotheksfunktionen printf(3), malloc(3) etc. → Absetzen von Betriebssystem- oder Laufzeitsystemaufrufen
        - Systemaufrufstümpfe (des Betriebssystems)
          z. B. Bibliotheksfunktionen write(2), sbreak(2) → Absetzen von synchronen Programmunterbrechungen (→ Traps)
    - statische versus dynamische Bibliothek



        - dynamisch:
            - Einbinden von Bibliotheksfunktionen bei Bedarf, z. B. beim erstmaligen Aufruf („trap on use“)
            - Speicherplatzersparnis (Hintergrundspeicher)
            - bindender Lader Teil des Betriebssystems
- Zusammenspiel von Ebene 2 und Ebene 3
    - zwei Sorten von Befehlen eines Maschinenprogramms
        - Aufrufe an Betriebssystem (Ebene 3)
            - explitzit als Systemaufruf (system call) kodiert
            - implizit als Programmunterbrechung (trap, interrupt) ausgelöst
        - Anweisungen an die CPU (Ebene 2)

      CPU immer als ausführende Instanz (Ebene-2-Befehle)
      Ebene-3-Befehle
        - werden „wahrgenommen“, nicht ausgeführt
        - signalisieren eine Ausnahme (exception)

      → Betriebssystem fängt Ebene-3-Befehle ab, behandelt Ausnahmen
    - partielle Interpretation eines Maschinenprogramms durch das Betriebssystem



1. Die Zentraleinheit interpretiert das Maschinenprogramm befehlsweise,
2. setzt dessen Ausführung aus,
   - Ausnahmesituation
   - Programmunterbrechung

   startet das Betriebssystem und
3. interpretiert die Programme des Betriebssystems befehlsweise.

Folge von 3., der Ausführung von Betriebssystemprogrammen. . .

4. Das Betriebssystem interpretiert das soeben oder zu einem früheren Zeitpunkt unterbrochene Maschinenprogramm befehlsweise und
5. instruiert die Zentraleinheit, die Ausführung des/eines zuvor unterbrochenen Maschinenprogramms wieder aufzunehmen.

Eine bei Ausführung eines Maschinenbefehls vom realen Prozessor Ebene 2, Zentraleinheit) gestartete Behandlung einer Ausnahmesituation bewirkt – bildlich gesprochen – das „Einschieben“ von Befehlsfolgen jener Programme in den Befehlsstrom, die den abstrakten Prozessor „Betriebssystem“ (Ebene 3) implementieren

- Ausnahmesituationen/Programmunterbrechungen (der Ebene 2)
  - Kategorien
    - „Falle“ (trap)
    - „Unterbrechung“ (interrupt)
  - Unterschiede hinsichtlich
    - Quelle
    - Synchronität
    - Vorhersagbarkeit
    - Reproduzierbarkeit

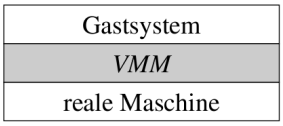
  Behandlung zwingend und grundsätzlich prozessorabhängig
  → vom jeweiligen realen *und* abstrakten Prozessor → Zentraleinheit und Betriebssystem
  - Trap: synchron, vorhersagbar, reproduzierbar
    - Ein in die Falle gelaufenes („getrapptes“) Programm, das unverändert wiederholt und jedesmal mit denselben Eingabedaten versorgt auf ein und demselben Prozessor zur Ausführung gebracht wird, wird auch immer wieder an derselben Stelle in dieselbe Falle tappen
    - Trapvermeidung ohne Behebung der Ausnahmebedingung unmöglich
    - Unterbrechungsverzögerung (interrupt latency)
      - d_{BS} (Ebene 3): begrenzt Echtzeitfähigkeit; unbegrenzt sonst
  - Interrupt: asynchron, unvorhersagbar, nicht reproduzierbar
    - Ein „externer Prozess“ (z. B. ein Gerät) signalisiert einen Interrupt unabhängig vom Arbeitszustand des gegenwärtig sich in Ausführung befindlichen Programms. Ob und ggf. an welcher Stelle das betreffende Programm unterbrochen wird, ist nicht vorhersagbar
    - Ausnahmesituationsbehandlung muss nebeneffektfrei verlaufen
    - Unterbrechungsverzögerung
      - d_{HW} (Ebene 2): konstant bei RISC, variabel (begrenzt) bei CISC
      - d_{BS} (Ebene 3): Trap → Betriebssystem kann d_{HW} beeinträchtigen
  - Behandlungsmodelle
    - Wiederaufnahmemodell/resumption model:
      - Behandlung der Ausnahmesituation führt zur Fortsetzung der Ausführung des unterbrochenen Programms
      - Trap kann, Interrupt muss so behandelt werden
    - Beendigungsmodell/termination model:
      - Behandlung der Ausnahmesituation führt zum Abbruch der Ausführung des unterbrochenen Programms
      - Trap kann, Interrupt darf niemalsso behandelt werden
  - Auslösung (raising)/Behandlung einer Ausnahme impliziert Kontexwechsel/nicht-lokale Sprünge (unterbrochenes ↔ behandelndes Programm)

        - Maßnahmen zur Zustandssicherung/-wiederherstellung erforderlich
        - Mechanismen liefert behandelndes Programm/die tiefere Ebene
    - Prozessorstatus unterbrochener Programme muss invariant sein
        - Ebene 2 (CPU) sichert bei Ausnahme Zustand minimaler Größe
            - Statusregister (SR), Befehlszeiger (program counter, PC)
            - evtl. kompletter Registersatz
            - evtl. große Menge Daten bewegt (CPU-abhängig)
        - Ebene 3/5 (Betriebssystem/Kompilierer) sichert restlichen Zustand
            - alle dann noch ugesicherten, im weiteren Verlauf verwendeten CPU-Register

    → zu ergreifende Maßnahmen höchst prozessorabhängig (≠ CPU-abhängig)
- Echtzeitbedingungen
    - unvorhersagbare Laufzeitvarianzen (bedingt durch Unterbrechungen) als Problem für determinierte Programme, da asynchrone Unterbrechungen sie nicht-deterministisch machen (nicht zu jedem Zeitpunkt ist bekannt, wie weitergefahren wird)
    - Nichtdeterminismus kritisch für echtzeitabhängige Programme → Laufzeitumgebung dieser Programme muss echtzeitfähig sein
    - verschiedene Echtzeitbedingungen
        - weich/soft/schwach:
            - Ergebnis einer zu einem vorgegebenen Termin nicht geleisteten Arbeit ist weiterhin von Nutzen
            - Terminverletzung tolerierbar
        - fest/firm/stark: ↝ plangemäß weiterarbeiten
            - Ergebnis einer zu einem vorgegebenen Termin nicht geleisteten Arbeit ist wertlos und wird verworfen
            - Terminverletzung tolerierbar, führt zum Arbeitsabbruch
            - Details
                - Betreibssystem bricht Arbeitsauftrag ab
                - nächster Arbeitsauftrag wird gestartet
                - transparent für Anwendung
        - hart/hard/strikt: ↝ sicheren Zustand finden
            - Versäumnis eines fest vorgegebenen Termins kann eine „Katastrophe“ auslösen
            - Terminverletzung keinesfalls tolerierbar
            - Details
                - Betriebssystem löst Ausnahmesituation aus
                - Ausnahmebehandlung führt zu sicherem Zustand
                - intransparent für Anwendung
- Asynchronität von Programmunterbrechungen
    - Begriffe
        - kritischer Abschnitt
        - Wettlaufsituation (*race condition*)
    - Überlappungsschutz für i++-Wettlaufsituationen – Lösungsansätze (Monoprozessoren)
        - Schutz vor möglicher überlappender Programmausführung
            - temporäres Abschalten asynchroner Programmunterbrechungen
            - „Synchronisationsklammern“ um den kritischen Abschnitt setzen
        - auf Elementaroperation eines tieferen Prozessors (→ CPU) abbilden
            - *bessere* Lösung, sofern CPU eine passende Elementaroperation anbietet
            - ggf. praktikabel bei CISC, nicht jedoch bei RISC
    - *Abstraktion* als grundsätzliche Vorgehensweise: eine Elementaroperation schaffen
        - einen kritischen Abschnitt als *Modul* abkapseln
        - den modularisierten Programmtext passend synchronisieren
- Virtualisierung
    - allgemein
        - Ebene-i-Befehl kann durch Ebene-i-Programm emuliert werden
        - Ebene-i-Programm kann durch Ebene-i-Befehl implementiert werden
    - Emulator interpretiert die von einer realen Maschine nicht ausführbaren Befehle
        - betreffende Befehle sind realer Maschine bekannt

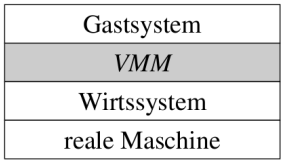
    - müssen jedoch nicht zwingend auch in ihr implementiert sein
- Selbstvirtualisierung: Rechensystem emuliert sich selbst
    - Voraussetzung: bei Ausführung eines Maschinenprogramms trappt die CPU *sensitive Befehle* (Befehle, deren direkte Ausführung für die virtuelle Maschine nicht tolerierbar ist) im „nicht-privilegierten“ Arbeitsmodus
    - Arbeitsmodi (einer CPU – jedoch nicht jeder)
        - privilegierter Modus → Betriebssystem, ggf. auch nur ein Minimalkern (virtual machine monitor, VMM) davon
        - nicht-privilegierter Modus: alle anderen Programme
    - Ausgangspunkt: Anzeige des Eintritts einer Ausnahmesituation, Umschaltung in privilegierten Arbeitsmodus des Prozessors
        1. Betriebssystem/VMM analysiert die Unterbrechung → Systemaufruf/Trap/Interrupt
        2. Emulator für unterbrochenes Programm starten
            - Ausführung des die Unterbrechung verursacht habenden Befehls
            - Abbildung von Geräteaktionen auf E/A-Funktionen (des Betriebssystems)
            - Umsetzung von Adressraumzugriffen und privilegierten Befehlen
        3. unterbrochenes Programm wird weiter fortgeführt
            - Beendigung der Emulation des Befehls bzw. Zugriffs
            - Reaktivierung des nicht-privilegierten Arbeitsmodus

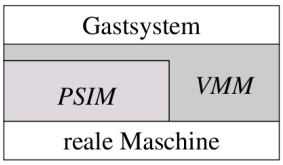
        → Abruf- und Ausführungszyklus eines „fiktiven“ Prozessors durchlaufen
- Paravirtualisierung: Verzahnung von Betriebssystem und VMM ↦ *alternative* Ebene 3
    - nicht gefordert: auch Betriebssysteme unverändert auf virtueller Maschine ablaufen lassen
    - stattdessen: im Betriebssystem spezielle Virtualisierungsunterstützung vorsehen
        - Problemlösung für nicht-privilegierte sensitive Befehle
        - „Beschleunigung“ von Virtualisierung
    - Präparation des Betreibssystems zum Zwecke der Ausführung auf einer durch einen *Hypervisor* implementierten virtuellen Maschine
        - Betreibssystem läuft dem VMM untergeodrnet auf der CPU ab
            - privilegierte Befehle werden weiterhin vom VMM abgefangen/emuliert
        - VMM liegt *intransparent* zwischen Betriebssystem und CPU
            - VMM kapselt sensitive Befehle in „komplexen“ Elementaroperationen
            - Betriebssystem nutzt diese Operationen, nicht die „nackten“ CPU-Befehle
- Ausprägungen von Virtualisierungssystemen



Typ 1 VMM privilegierte Befehle werden vom VMM abgefangen

Typ 2 VMM privilegierte Befehle werden vom Wirtssystem abgefangen und an den VMM hochgereicht





hybrider VMM (HVM) privilegierte Befehle werden vom VMM abgefangen, laufen jedoch auf einer partiell in Software implementierten Maschine (PSIM) ab

- Grenzen der Emulation
  - funktionale Eigenschaften: Nachahmung „leicht“ möglich
  - nicht-funktionale Eigenschaften: emulierter Befehl durch „Unterprogramm“ ausgeführt → läuft langsamer aber, verbraucht mehr Energie

- Zusammenfassung: Hierarchie virtueller Maschinen (bzw. abstrakter Prozessoren)
  - schrittweises Schließen der semantischen Lücke
  - Mehrebenenmaschinen – Betriebssysteme implementieren Ebene 3
  - Ebene i ↦ Ebene i-1 durch Programme, für i > 1
  - Teilinterpretation (von Systemaufrufen) durch das Betriebssystem
  - synchrone und asynchrone Programmunterbrechungen
  - nebenläufige (bzw. überlappende) Ausführung von Programmen
  - Nachahmung der Eigenschaften von (abstrakten) Prozessoren

6. Betriebsarten

6.1 Präludium

6.2 Stapelbetrieb

6.3 Echtzeitbetrieb

- Echtzeitbetrieb: Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorbestimmten Zeitpunkten anfallen
- Echtzeitfähigkeit ≙ Rechtzeitigkeit: zuverlässige Reaktion des Rechensystems auf Umgebungsereignisse
  - Determiniertheit: bei ein und derselben Eingabe verschiedene Abläufe zulässig, alle Abläufe liefern jedoch stets das gleiche Resultat
  - Determinismus: zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird
  - Terminvorgaben sind weich, fest, hart

6.4 Mehrprogrammbetrieb

- Maßnahmen zur Leistungssteigerung: Trennung von Belangen (*separation of concerns*)
  - Multiprogrammbetrieb (*multiprogramming*) und Simultanbetrieb (*multiprocessing*), Maßnahmen zur Strukturierung

        - Multiplexen der CPU zwischen mehreren Programmen → Nebenläufigkeit (*concurrency*)
        - Abschottung der sich in Ausführung befindlichen Programme → Adressraumschutz (address space protection)
        - Überlagerung unabhängiger Programmteile
- Multiplexen des Prozessors
    - Wartezeit von Programm x als Laufzeit von Programm y nutzen
        - aktives Warten (*busy waiting*) auf Ereigniseintritte vermeiden
        - stattdessen passives Warten betreiben, CPU einem anderen lauffähigen Programm zuteilen
    - Abstraktion von wartenden/laufenden Programmen → generisches Konzept zur Programmverarbeitung und -verwaltung
        - Prozess: beliebiges „Programm in Abarbeitung“
        - für jedes zu ladende Programm wird ein Prozess erzeugt
    - Einplanung (*scheduling*) und Einlastung (*dispatching*) von Prozessen regeln die Verarbeitung lauffähiger Programme
- Adressraumschutz
    - *Bindungszeitpunkt* von Programm- an Arbeitsspeicheradressen beeinflusst Modell zur Abschottung der Programme – und umgekehrt
    - *vor Laufzeit:* Schutz durch Eingrenzung
        - Programme laufen im physikalischen Adressraum
        - verschiebender Lader besorgt Bindung zum *Ladezeitpunkt*
        - CPU überprüft generierte Adressen zur Ausführungszeit
    - *zur Laufzeit*: Schutz durch Segmentierung
        - jedes Programm läuft in eigenem logischen Adressraum
        - Lader bestimmt Bindungsparameter (physikalische Basisadresse)
        - CPU (bzw. MMU) besorgt Bindung zur Ausführungszeit

  → in beiden Fällen: Binder richtet Programme relativ zu einer logischen Basis aus (meist 0)
- Adressraumschutz durch Eingrenzung
    - Festlegen der Unter-/Obergrenze eines Programms im physikalischen Adressraum durch Begrenzungsregister
        - für jedes Programm ein Registerpaar
          → Softwareprototypen der Adressüberprüfungshardware
        - Abbildung auf Hardwareregister bei Prozesseinlastung
    - Speicherverwaltung statisch, geschieht vor Programmausführung
        - verschiebender Lader legt Lage im Arbeitsspeicher fest
        - zur Programmlaufzeit zusätzlicher Speicher nur bedingt zuteilbar → spätestens zum Ladezeitpunkt muss weiterer Bedarf bekannt sein

  ⚠ Fragmentierung, Verdichtung
- Adressraumschutz durch Segmentierung
    - Basis-/Längenregister (*base*/*limit register*) geben Programmen vom physikalischen Adressraum abstrahierende logische Adressräume
        - für jedes Programm ein Registerpaar
          → Softwareprototypen der Adressumsetzungshardware
        - Abbildung auf Hardwareregister bei Prozesseinlastung
    - Speicherverwaltung dynamisch, geschieht zur Programmausführung
        - Lader legt initiale Lage im Arbeitsspeicher fest
        - zur Programmlaufzeit zusätzlicher Speicher „beliebig“ zuteilbar (limitierend: physikalisch noch freier Arbeitsspeicher)

  ⚠ Überbelegung des Arbeitsspeichers (zu große/viele Programme)

  → Grundlage für MMU des heutigen Stands der Technik
- dynamisches Laden: Überlagerung (overlay) von Programmteilen im Arbeitsspeicher
    - Technik der Überlagerung: Ein Programm, das einschließlich Daten die Kapazität des Hauptspeichers übersteigt, wird in hinreichend kleine Teile zergliedert, die nicht ständig im Hauptspeicher vorohanden sein müssen, sondern stattdessen im Hintergrundspeicher (Trommel, Platte) vorgehalten werden
    - Nachladen von Überlagerungen nur bei Bedarf (on demand)
        - Nachladen programmiert/im Programm festgelegt

        - Entscheidung zum Nachladen fällt zur Programmlaufzeit
    - ⚠ Finden der (zur Laufzeit) „optimalen“ Überlagerungsstruktur
    - Überlagerungsstruktur eines Programms
        - grobe Einteilung:
            1. arbeitsspeicherresidenter Programmteil (*root program*)
            2. mehrere in Überlagerungen zusammengefasste Unterprogramme
            3. ein gemeinsamer Datenbereich (common section)
        - Überlagerungen als Ergebnis einer Programmzerlegung zur Programmier-, Übersetzungs- und/oder Bindezeit
            - manuelle/automatisierte Abhängigkeitsanalyse des Programms
            - Anzahl und Größe von Überlagerungen statisch festgelegt
            - dynamisch: aktivieren/laden von Überlagerungen
- Simultanverarbeitung mehrerer Auftragsströme (multiprocessing, multi-stream batch monitor)
    - Verarbeitungsströme sequentiell auszuführender Programme/Aufträge
        - pseudo-parallele Abarbeitung mehrerer Stapel im Multiplexverfahren
            - sequentielle Ausführung der Programme desselben Auftragsstapels
            - überlappende Ausführung der Programme verschiedener Auftragsstapel
        - „wer [innerhalb des Stapels] zuerst kommt, mahlt zuerst“
    - Stapelwechsel
        - kooperativ (*cooperative*): bei (programmierter) Ein-/Ausgabe und Programmende (Beendigung des CPU-Stoßes eines Prozesses)
        - verdrängend (*preemptive*): bei Ablauf einer Frist (*time limit*)
    - ⚠ interaktionsloser Betrieb, Mensch/Maschine-Schnittstelle

6.5 Mehrzugangsbetrieb

- Dialogbetrieb (*conversational mode*): gleichzeitiger, interaktiver Zugang für mehrere Benutzer
    - Abwechslung von Benutzereingaben und deren Verarbeitung (E/A-intensive Anwendungsprogramme)
    - interaktive (E/A-intensive) Prozesse bei Einplanung bevorzugt
    - ⚠ Zusatz zum Stapelbetrieb, Monopolisierung der CPU, Sicherheit
- Hintergrundbetrieb: Programme „im Vordergrund“ starten, „im Hintergrund“ verarbeiten
    - Mischbetrieb
        - E/A-intensive Programme im Vordergrund
        - CPU-intensive Programme im Hintergrund

Vordergrund	Hintergrund
Echtzeitbetrieb Dialogbetrieb	Dialogbetrieb Stapelbetrieb

    - ⚠ Arbeitsspeicher
- Teilnehmerbetrieb: Dialogstationen können eigene Dialogprozesse absetzen
    - *Zeitscheibe* (*time slice*) als „Betriebsmittel“
        - time sharing: Prozesse berechtigen, die CPU für einige Zeit zur Abarbeitung der ihnen zugeordneten Programme zu nutzen
        - Zeitspanne geendet → erneute Einplanung (ggf. Verdrängung (*preemption*))
    - *Zeitgeber* (timer) → zyklische Programmunterbrechungen
        - Intervall durch Zeitscheibenlänge vorgegeben
        - CPU-Schutz: Monopolisierung nicht mehr möglich
    - ⚠ Arbeitsspeicher, Einplanung, Einlastung, Ein-/Ausgabe, Sicherheit
- Teilhaberbetrieb: Dialogstationen teilen sich gemeinsamen Dialogprozess
    - Bedienung mehrerer Benutzer durch ein Programm/einen Prozess
        - gleichartige, bekannte Vorgänge an vielen Daten-/Dialogstationen
        - Wiederverwendung derselben residenten Dialogprozessinstanz
    - Endbenutzerdienst mit festem, definiertem Funktionsangebot
        - Kassen, Bankschalter, Auskunft-/Buchungssysteme, …

        - Transaktionssysteme
    - Klient-/Anbieter-System (*client/server system*)
        - sehr *Dienstnehmer* (*service user*)
        - einen/wenige *Dienstgeber* (*service provider*)

    ⚠ Antwortverhalten (weiche/feste Echtzeit), Durchsatz
- Multiprozessorbetrieb
    - homogenes System
        - SMP (*symmetric multiprocessing*): Zwei oder mehr gleiche/identische Prozessoren, die über ein gemeinsames Verbindungssystem gekoppelt sind
            - gleichberechtigter Zugriff auf Hauptspeicher (*shared-memory access*), E/A-Geräte
            - Zugriff auf alle Speicherzellen für alle Prozessoren gleichförmig (uniform memory access, UMA)
            - Prozessoren bilden homogenes System, werden von demselben Betriebssystem verwaltet

        ⚠ Koordination, Skalierbarkeit
    - homogenes/heterogenes System
        - SMP (*shared-memory processor*): Parallelrechnersystem, bei dem sich alle Prozessoren denselben Arbeitsspeicher teilen, nicht jedoch gleichberechtigen Zugriff darauf haben müssen – sie können zusammen ein heterogenes System bilden
            - asymmetrischer SMP:
                - impliziert hardwarebedingt asymmetrischen Multiprozessorbetrieb (*asymmetric multiprocessing*) durch Betriebssystem
                - Programme ggf. prozessorgebunden (incl. Betriebssystem)
            - symmetrischer SMP: lässt Betriebssystem in Abhängigkeit von Anwendungsprogrammen die Wahl der Multiprozessorbetriebsart (symmetrisch/asymmetrisch)

        ⚠ Koordination, Skalierbarkeit, Anpassbarkeit
    - Parallelverarbeitung (*parallel processing*) eines Programms/mehrerer Programme
- Sicherheit (*security*): Schutz (*protection*) vor nicht autorisierten Zugriffen
    - Adressraumisolation: Schutz ↝ Eingrenzung/Segmentierung
      → i. A. keine selektive Zugriffskontrolle möglich, sehr grobkörnig
    - Prozessen Befähigung (*capability*) zum Zugriff erteilen: verschiedenen Subjekten unterschiedliche Zugriffsrechte auf dasselbe Objekt einräumen
      alternativ: Zugriffskontrollliste (*access control list*, ACL)

    ⚠ Widerruf (*revocation*), verdeckter Kanal (*covered channel*)
- Arbeitsspeicher und Grad an Mehrprogrammbetrieb: Überlagerung durch programmiertes dynamisches Laden hat Grenze
    - Anzahl × Größe arbeitsspeicherresidenter Text-/Datenbereiche begrenzt Anzahl der gleichzeitig zur Ausführung vorgehaltenen Programme
    - Umlagerung von solchen Bereichen gegenwärtig nicht ausführbereiter Programme (*swapping*) verschiebt Grenze nach hinten
    - Auslagerung von Bereichen sich in Ausführung befindlicher Programme (*paging*, *segmentation*) liefert weiteren Spielraum
    - Umlagerung von Programmen: Funktion der mittelfristigen Einplanung (*medium-term scheduling*) von Prozessen

      ⚠ Fragmentierung, Verdichtung, Körnigkeit
- virtueller Speicher
    - Überbelegung des Arbeitsspeichers zulassen, Programme trotz Arbeitsspeicherknappheit ausführbar
        - nicht benötigte Programmteile → Hintergrundspeicher
        - Zugriff auf ausgelagerte Teile ↝ partielle Interpretation
        - Abwechslung von Aus-/Einlagerung

      ⚠ Lade- und Ersetzungsstrategien
    - Körnigkeit/Granularität: fest oder variabel
    - Seiten (*pages*)/Segmente (*segments*): Programmteile, die ein-, aus- und/oder überlagert werden können

        - Seitenüberlagerung (*paging*): Abbildung von Adressraumteilen fester Größe (Seiten) auf entsprechend große Teile (Seitenrahmen) des Haupt-/Hintergrundspeichers
        - Segmentierung (*segmentation*): Abbildung von Adressraumteilen variabler Größe (Segmente) auf entsprechend große Teile des Haupt-/Hintergrundspeichers
        - seitennummerierte Segmentierung (*paged segmentation*): Segmente in Seiten untergliedern → Kombination beider Verfahren

      ⚠ interne Fragmentierung (Seiten), externe Fragmentierung (Segmente)
    - Ringschutz (*ring-protected paged segmentation*):
      - Multics als Maßstab bzgl. Adressraum-/Speicherverwaltung:
        - *jede* im System gespeicherte abrufbare Information direkt von einem Prozessor adressierbar, von jeder Berechnung referenzierbar
        - *jede* Referenzierung unterliegt durch Hardwareschutzringe implementierten mehrstufigen Zugriffskontrolle
      - trap on use: ausnahmebedingtes dynamisches Binden/Laden
        - automatisches Nachladen der Textsegmente bei Bedarf
        - mögliche synchrone Programmunterbrechung beim Prozeduraufruf

      ⚠ hochkomplexes System (Hardware/Software)
    - automatische Überlagerung durch partielle Interpretation → ähnlich Überlagerungstechnik, jedoch…
      - Seiten-/Segmentanforderungen nicht in Anwendungen programmiert, sondern in Betriebssystem, Anforderungen stellvertretend durch Systemprogramm gestellt → Ladeanweisungen vor Anwendungsprogramm verborgen
      - Abfangen von Zugriffen auf ausgelagerte Seiten/Segmente auf Befehlssatzebene, Weiterleitung an Betriebssystem
        - Unterbrechung des Anwendungsprogramms durch CPU/MMU
        - partielle Interpretation des Zugriffs vom Betriebssystem
      - Wiederholungen des unterbrochenen Maschinenbefehls in Betracht zu ziehen, vom Betriebssystem zu veranlassen

      ⚠ Komplexität, Determiniertheit
- Allgemeinzweckbetriebssysteme (*general purpose operating systems*): verschiedensten Anforderungen gerecht werden wollen ⇒ evtl. für keine ideal
    - Verbesserungen der Benutzerfreundlichkeit
      - Selbstvirtualisierung
      - Konzentration auf das Wesentliche

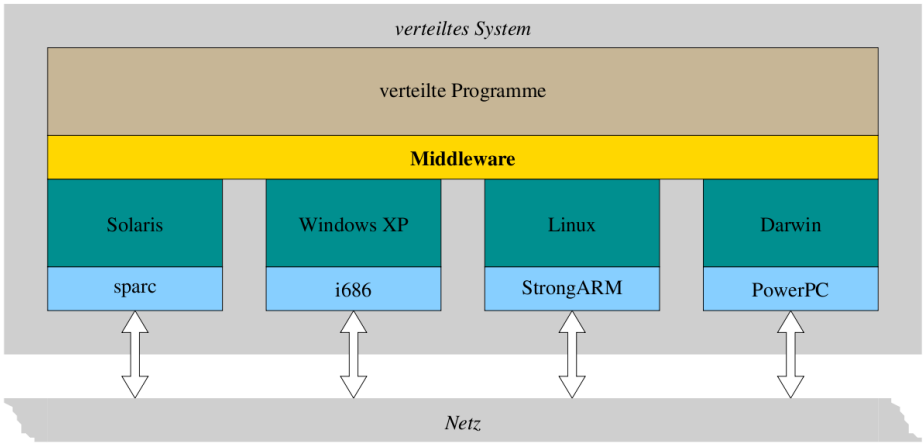
6.6 Netzbetrieb

- offenes System
    - Programme haben Zugriff auf Betriebsmittel eines Rechnerverbunds
      - Transparenz des Netzwerks in verschiedenster Weise
        - Netzwerktransparenz → Zugriffs-, Ortstransparenz
        - Replikations-, Migrations-, Fehler-, Ausfall-, Leistungs-, Skalierungs-, Nebenläufigkeits-, …, X-Transparenz
      - Virtualisierung ferner Betriebsmittel durch lokale Repräsentanten
    - verteilte Programmverarbeitung (ein Stück weit)

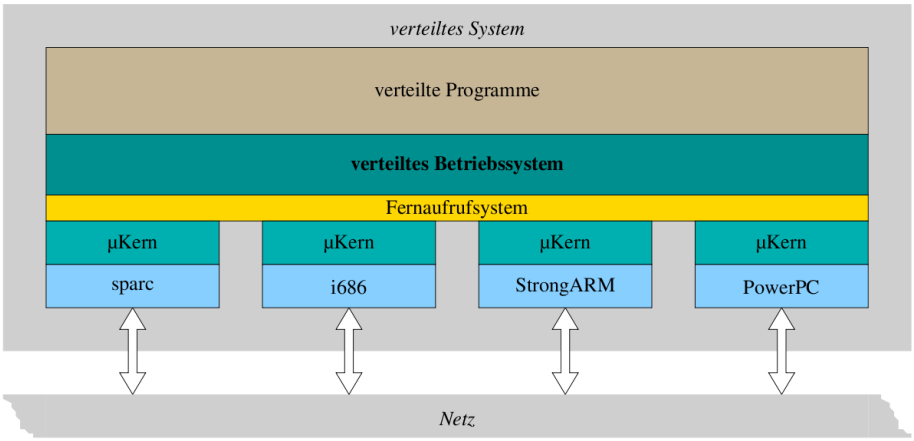
  ⚠ Heterogenität, netzwerkzentrische Betriebsmittelverwaltung
- Prozessgrenzen überwindende Unterprogrammaufrufe → Dienstschicht (*middleware*)
    - Prozedurfernaufruf (*remote procedure call*), RPC) → Illusion des lokalen Zugriffs auf entfernte Rechner
    - Virtualisierung von Aufrufer/Aufgerufenem durch Stümpfe (*stubs*) (Art bezeichnet Lage)
      - Klientenstumpf (*client stub*): Repräsentation des Aufgerufenen (Anbieters) beim Aufrufer (Klienten)
      - Anbieterstumpf (*server stub*): Repräsentation des Aufrufers (Klienten) beim Aufgerufenen (Anbieter)

  ⚠ Parameterübergabe, Serialisierung von Datenstrukturen, Fehlermodell

- verteiltes System
  - als einzelnes System präsentierter Zusammenschluss vernetzter Rechner



  - Virtualisierung aller im Rechnerverbund verfügbaren Betriebsmittel



- Hauptarten von Rechnernetzen (Klassifikation nach Abdeckungsbereichen)
  - LAN (*local area network*)
    - typisch für Vernetzung von Arbeitsplatzrechnern
    - meist homogenes System: dieselben Rechner, Betriebssysteme
  - MAN (*metropolitan area network*)
    - typisch für Vernetzung von Großrechnern, LANs
    - heterogenes System: verschiedene Rechner, Betriebssysteme
  - WAN (*wide area network*)
    - typisch für Vernetzung von LANs, WANs: Internet
    - heterogenes System: verschiedene Rechner, Betriebssysteme

  ⚠ Skalierbarkeit, Echtzeitfähigkeit
  - CAN (*control area network*)
    - typisch für Vernetzung elektronisch gesteuerter Maschinen
      → Netzknoten als Steuergeräte (electronic control units, ECU)
    - heterogenes System (je nach Anwendungsfall) → z. B. KFZ

6.7 Integrationsbetrieb

- Spezialzweckbetriebssystem (special purpose operating system)
  - → Betriebssystem, Anwendungsprogramm(e) mit-/ineinander verwoben, bilden (in sich geschlossene) Einheit
    - *im Sinne der Funktionalität* (obligatorisch): Betriebssystem maßgeschneidert, anwendungsgewahr
    - *im Sinne der Repräsentation* (optional): Betriebssystem liegt in Form einer (Quelltext-)Bibliothek vor

    - eingebettetes System: in einem Produkt verstecktes Rechensystem, wobei das Produkt selbst *kein* Rechner ist
- drahtlose Sensornetzwerke
- einbettbare Betriebssysteme
  ⚠ Skalierbarkeit, Betriebsmittelbedarf (insbesondere RAM, Energie) bei „abgewandelten Desktopbetriebssystemen“(?)
  ⚠ Softwaretechnik ↝ Wiederverwendbarkeit, Variantenverwaltung bei anderen („Rad neu erfinden“) (?)

7. Funktionale Abstraktionen

7.1 Adressraum

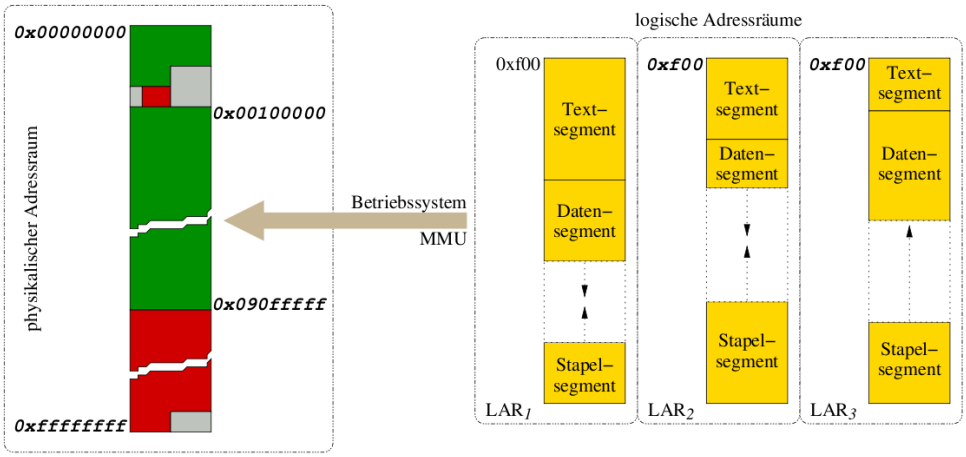
- Adressraumkonzepte und virtuelle Maschinen
  - physikalischer Adressraum (Hardware): Ebene 2
    - durch jeweils gegebene Hardwarekonfiguration definiert
    - nicht jede Adresse gültig, zur Programmspeicherung verwendbar
  - logischer Adressraum (Kompilierer, Binder, Betriebssystem): Ebene 5/4/3
    - abstrahiert von Aufbau/Struktur des Hauptspeichers
    - alle Adressen gültig, zur Programmspeicherung verwendbar
  - virtueller Adressraum (Betriebssystem): Ebene 3
    - auf Vorder- und Hintergrundspeicher abgebildeter logischer Adressraum
    - erlaubt Ausführung unvollständig im RAM liegender Programme
- logischer Adressraum
  - Illusion von einem eigenen (nicht zwingend linearen) Adressraum für jedes im Hauptspeicher vollständig vorliegende Programm
    - Anfangsadressen meist gleich (→ Systemkonstante)
    - Endadressen variabel, aber nach oben begrenzt (→ Programmlängen, Hardwarefähigkeiten)
  - mehrstufige Adressabbildung (address mapping):
    - Programm ↦ logischer Adressraum
    - logischer Adressraum ↦ physikalischer Adressraum

    → logische Adressen mehrdeutig, physikalische Adressen eindeutig
  - Abbildungszeitpunkte: Adress(raum)abbildung auf verschiedenen Ebenen möglich
    → Zielkonflikt (trade-off) bzgl. Flexibilität, Effizienz
    je später die Abbildung, desto
    - höher das Abstraktionsniveau und geringer die Hardwareabhängigkeit
    - höher der Systemaufwand und geringer der Spezialisierungsgrad
- Verantwortlichkeiten bei Adressraumabbildung: Zusammenspiel Betriebssystem und Hardware/MMU
  - Betriebssystem (Ebene 3): Adressraumabbildung zur Ladezeit
    - Lader fordert Betriebsmittel zur Programmausführung an
    - Verwaltungsinformationen für MMU werden aufgesetzt
    - neuer Prozess wird der Einplanung (*scheduling*) zugeführt
  - Hardware/MMU (Ebene 2): Adressumsetzung zur Laufzeit → Verwendung der in den Deskriptoren gespeicherten Informationen

  → Verantwortung trägt allein Betriebssystem, MMU führt nur aus
- Segmentierung eines logischen Adressraums: logische Unterteilung zur effektiveren Programmverwaltung
  - Textsegment/text segment
    - Maschinenbefehle (Ebene 2/3), anderen Programmkonstanten
    - statische oder dynamische Größe (abhängig von Betriebssystem)
    - shared text: ggf. gemeinsam angelegt für mehrere Prozesse
  - Datensegment/data segment
    - initialisierte Daten, globale Variablen und ggf. die Halde (heap)
    - statische oder dynamische Größe (abhängig von Betriebssystem)

- Stapelsegment/stack segment
  - lokale Variablen, Hilfsvariablen, aktuelle Parameter
  - dynamische Größe

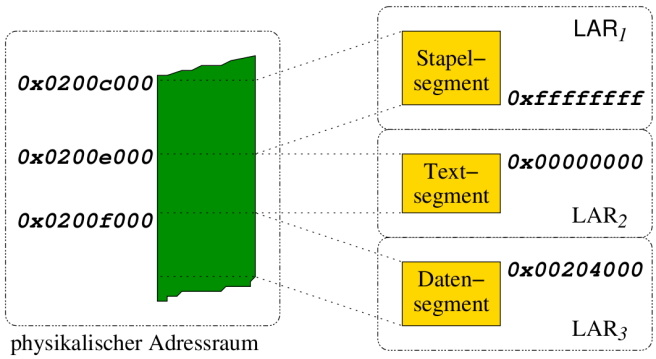
- Adressraumabbildung auf Ebene 3: Betriebssystem und MMU implementieren logische Adressräume



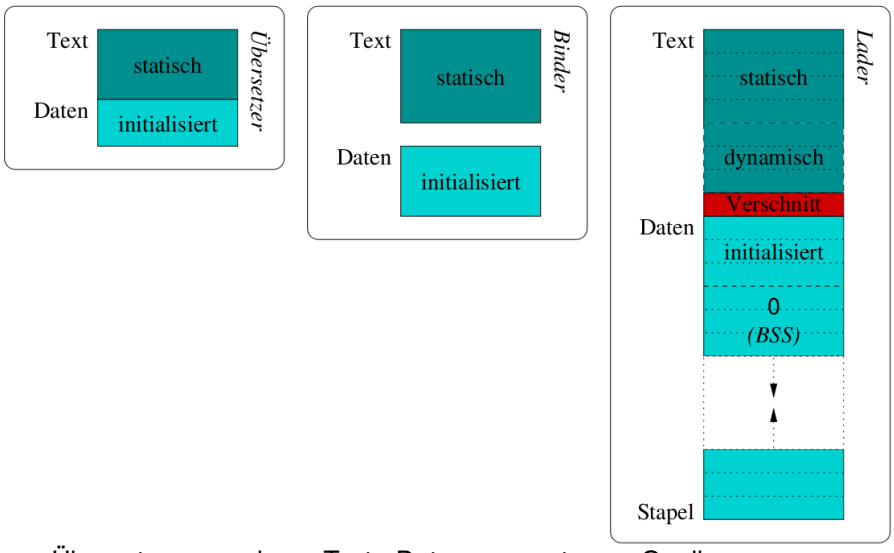
→ Segmente brauchen nicht angrenzend im logischen Adressraum zu liegen

- disjunktive Abbildung zur Ladezeit: Betriebssystem ordnet Segmente überschneidungsfrei im physikalischen Adressraum an
  → Mitbenutzung (*sharing*) von Segmenten unter Kontrolle des Betriebssystems möglich: absichtliche Überschneidung im Hauptspeicher
  - Verletzung der Segmentierung (*segmentation violation*) durch MMU verhindert, bewirkt Programmabbruch:
    $0 \leq \text{Adresse}_{log} - \text{Adresse}_{min} < \text{Länge(Segment)}$, sonst Trap
  - konstante Adresse_{min} bestimmt den Anfang eines logischen Adressraums
- Adressrelokation zur Laufzeit: MMU wandelt jede logische Adresse im Abrufzyklus (*fetch cycle*) der CPU um
  - Veränderung einer logischen Adresse um eine Relokationskonstante (Prinzip):
    $\text{Adresse}_{phy} = \text{Adresse}_{log} - \text{Adresse}_{min} + \text{Basis(Segment)}$
  - Basis(Segment): Ladeadresse im physikalischen Adressraum
    - $\text{Adresse}_{log} - \text{Adresse}_{min}$ relativiert zu Null → relative Adresse bzgl. Basis(Segment)
    - „Verschiebung“ des relativierten Werts durch anschließende Addition
  - Ladeadresse eines Segments ist gleichfalls Relokationskonstante
    → für alle relativ(iert)en Adressen innerhalb von *Segment*

  ☞ Relokation erfolgt nur bei unverletzter Segmentierung



- logischer Adressraum als *Schutzdomäne* ↝ Verbesserung der Robustheit von Softwaresystemen in Rechensystemen im Mehrprogrammbetrieb
  - Adressraumisolation → Erhöhung der Sicherheit
    - safety: Schutz von Menschen und Sachwerten vor dem Versagen technischer Systeme
    - security: Schutz von Informationen und Informationsverarbeitung vor „intelligenten“ Angreifern
- UNIX-Segmentierung: Dienstprogramm-basierter (*utility [based]*) seitennummerierter Ansatz

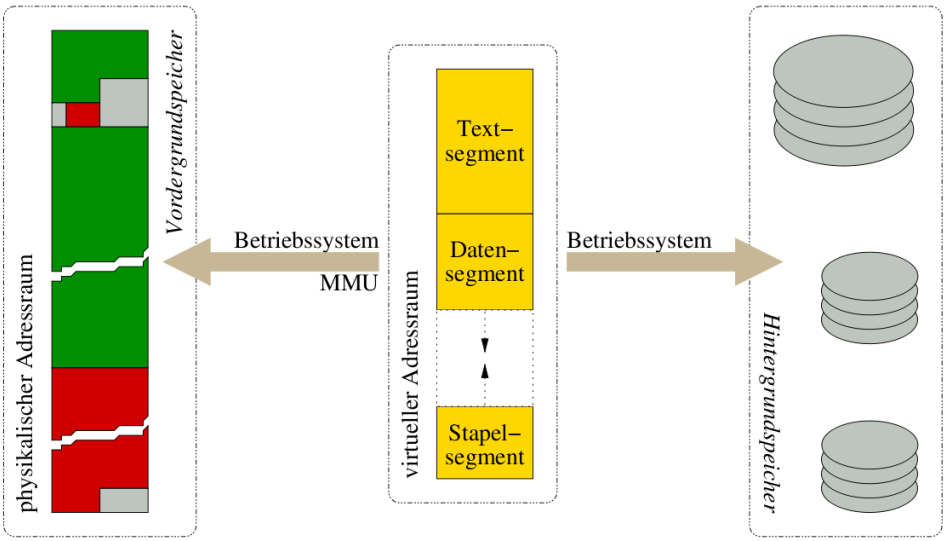


- Übersetzer generieren Text-, Datensegmente aus Quellprogramm
- Binder packen Text/Daten aus Bibliotheken dazu, hinterlassen ggf. seitenbündige Segmente
- Lader bringen statische Segmente in RAM, fügen dynamische Text-/Stapelsegmente hinzu, setzen BSS (*block started by symbol*) auf 0

- virtueller Adressraum
  - Erhöhung des Grads des Mehrprogrammbetriebs (*degree of multiprogramming*)
  - Illusion eines eigenen (nicht zwingend linearen) Adressraums für jedes im Hauptspeicher ggf. unvollständig vorliegende Programm
    - Erweiterung/Spezialisierung des logischen Adressraums
    - meist verbreitet: Seitenüberlagerung
    - Adressraumzugriffe können E/A (Hintergrundspeicher) implizieren
  - mehrstufige Adressabbildung (*address mapping*)
    Programm ↦ logischer Adressraum
    logischer Adressraum ↦ virtueller Adressraum
    virtueller Adressraum ↦ physikalischer Adressraum

☞ virtuelle Adressen ebenso mehrdeutig wie logische Adressen

  - Adressraumabbildung auf Ebene 3: Implementierung virtueller Adressräume durch Betriebssystem, MMU



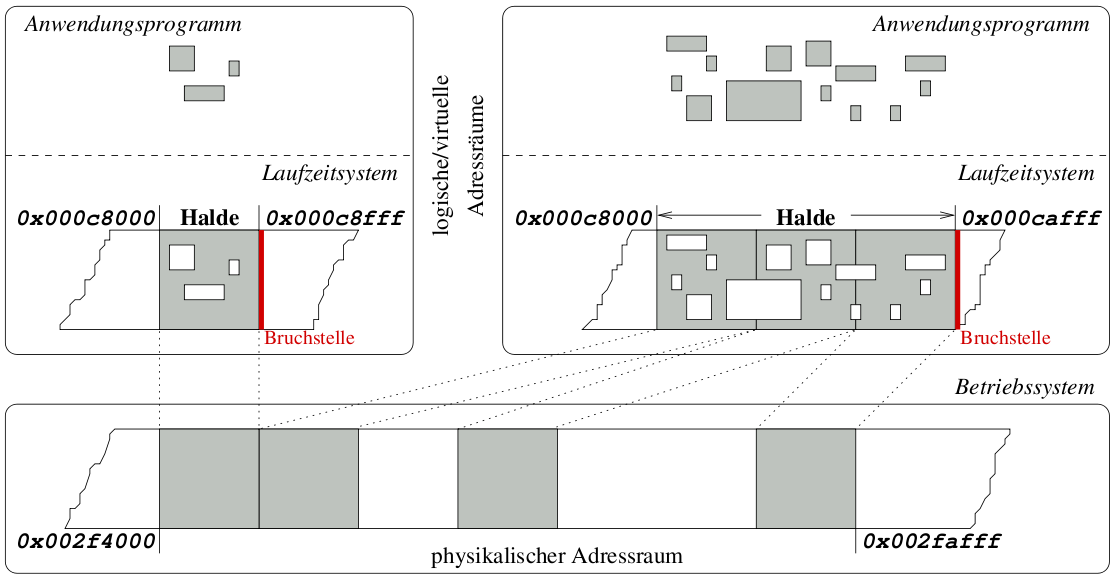
  - Umfang eines virtuellen Adressraums: Hauptspeicher ⊂ Arbeitsspeicher (Rechner i. d. R. nur mit Bruchteil des von einer CPU adressierbaren Arbeitsspeichers wirklich bestückt)

7.2 Speicher

- Speicherkonzepte, -medium: kurz-, mittel-, langfristige Informationsspeicherung
  - Vordergrundspeicher: Hauptspeicher (RAM)
    - entsprechend bestückter Bereich im physikalischen Adressraum
    - Zentralspeicher zur Programmausführung (→ Von-Neumann-Rechner)
    - kann physikalischen Adressraum überschreiten: Speicherbankumschaltung
    - kurzfristige Speicherung, Zugriffszeiten im ns-Bereich
  - Hintergrundspeicher: Massenspeicher (Band, Platte, CD, DVD)
    - über Rechnerperipherie (E/A-Geräte) angeschlossene Bereiche
    - Datenablage und Implementierung virtueller Adressräume
    - größer als physikalischer Adressraum
    - mittel- bis langfristige Speicherung, Zugriffszeiten im ms-Bereich

  ☝ Virtualisierung kann Zugriffstransparenz mit sich bringen
- Speicherverwaltung (memory management) → Symbiose von Laufzeit- und Betriebssystem
  - Laufzeitsystem (bzw. Bibliotheksebene): Verwaltng des lokal vorrätigen Speichers eines logischen/virtuellen Adressraums
    - Speicherblöcke: sehr feinkörnige Struktur/Größe möglich (z. B. einzelne Bytes, Verbundobjekte)
    - Orientierung der Verfahrensweisen (mehr) an Programmiersprachen
  - Betriebssystem: Verwaltung des global vorrätigen Speichers (→ verfügbarer RAM) des physikalischen Adressraums
    - Speicherblöcke: grobkörnige Struktur/Größe (z. B. Vielfaches von Seiten)
    - Fokussierung der Verfahrensweisen auf Benutzer-/Systemkriterien

  → Trennung von Belangen (separation of concerns)
  - Synergie bei Speicherverwaltung: Betriebssystemaufruf als „Ausnahme“



  - Unix-Systemfunktionen – free(): Freigabe von Speicher nur mit lokaler Relevanz
    - keine freiwillige Rückgabe an Betriebssystem
    - Wiedergewinnung freigegebener Bereiche nur bei Beendigung des Progamms und/oder auf Basis virtuellen Speichers

7.3 Datei

- langfristige Datenspeicherung ↝ Abstraktion von Informationen tragenden Betriebsmitteln
  - Datei (*file*): Sammlung von Daten
    - zusammenhängende, abgeschlossene Einheit von Daten
    - „beliebige“ Anzahl eindimensional adressierter Bytes
  - Dauerhaftigkeit → Frage des Speichermediums (Datei selbst durchaus unbeständig)

        - nicht-flüchtige Datenträger
        - flüchtige Datenträger
    - Kommunikationsmittel für kooperierende Prozesse ↝ Mechanismus zur Weiterleitung (*pipe*) von Informationen
- Arten von Dateien
    - ausführbar: Binär-/Skriptprogramme
        - von Prozessor ausführbarer Programmtext
        - Prozessor in Hard-, Firm- und/oder Software
    - nicht ausführbar: Text-, Bild-, Tondateien
        - von Prozessor verarbeitbare Programmdaten
        - Prozessor in Form von Programmtext
- Bezeichnung von Dateien: symbolische und numerische Dateiadressen
    - von außen: Dateien über symbolische Adressen erreichbar
        - benutzerdefinierter Name → Dateiname (*file name*), ggf. von Betriebssystem (teilweise) interpretiert
    - nach innen: numerische Adresse für jede Datei
        - systemdefinierte Kennung einer Datenstruktur der Dateiverwaltung
        - damit Identifikation des Dateikopfes (*file head*)

  → symbolische, numerische Dateiadresse als (festes) Paar
- Erweiterung des Dateinamens → semantische Information
  → Dateinamensuffix (*file extension*): symbolische Erweiterung des Dateinamens, meist durch Punkt vom Dateinamen abgegrenzt
    - Hinweis auf Dateiformat/Dateityp
    - Dienstprogrammen (UNIX) und/oder Betriebssystem (Windows) bekannt
- Dateikopf, Dateikopfnummer → systeminterne Verwaltung einer UNIX-Datei
    - inode/i-node → Dateiattribute
    - inode number: Index in Tabelle von Dateiköpfen (*inode table*) → numerische Adresse der Datei (innerhalb des Dateisystems)
- Dateityp ↝ Dateien zur Abstraktion von Daten, Geräten, Kommunikationsmitteln
    - reguläre Datei (*regular file*, *ordinary file*)
      → problemorientiertes, eindimensionales Bytefeld
    - spezielle Datei
        - Verzeichnis (*directory*): Katalog von regulären und/oder speziellen Dateien
        - Gerätedatei (*device file*): Zugang zu zeichen-/blockorientierten Geräte(treiber)n
        - symbolische Verknüpfung (*symbolic link*): Abbildung eines Dateinamens auf einen Pfadnamen
        - benannte Leitung (*named pipe*): Kommunikationskanal zwischen unverwandten lokalen Prozessen
        - Buchse (*socket*): Endpunkt zur bidirektionalen Kommunikation zwischen Prozessen
- Dateiverzeichnis: Konzept zur Gruppierung von Dateinamen
    - Katalog (*catalogue, directory*) von symbolischen Namen
        - Definition eines gemeinsamen Kontext
          → symbolische Adressen nur innerhalb ihrer Kontexte eindeutig
        - Implementierung einer Umsetzungstabelle

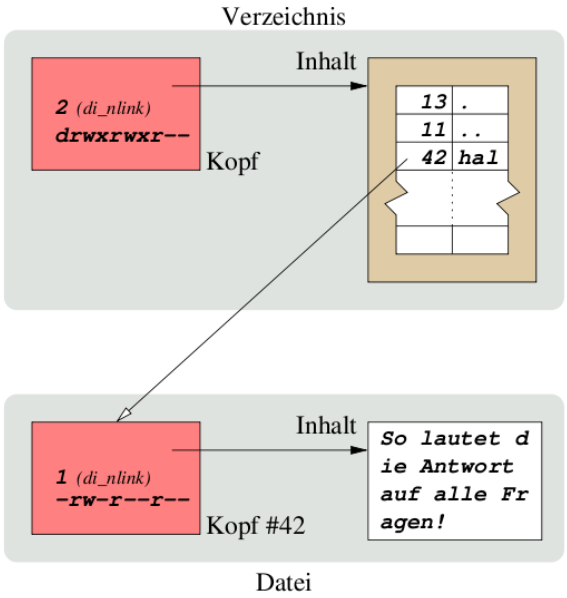
          $$\left\{\begin{array}{c}\text{symbolische Adresse}\\ \text{Dateiname}\\ \texttt{foo}\end{array}\right\} \rightsquigarrow \left\{\begin{array}{c}\text{numerische Adresse}\\ \text{Dateikopf}\\ 4711\end{array}\right\}$$

        - Speicherung der Abbildung Dateiname ↦ Dateikopfnummer
- Verknüpfung Dateiname – Dateikopf (UNIX *hard link*/*link)*
    - Eintrag in Dateiverzeichnis: Dateiname ↦ Dateikopfnummer

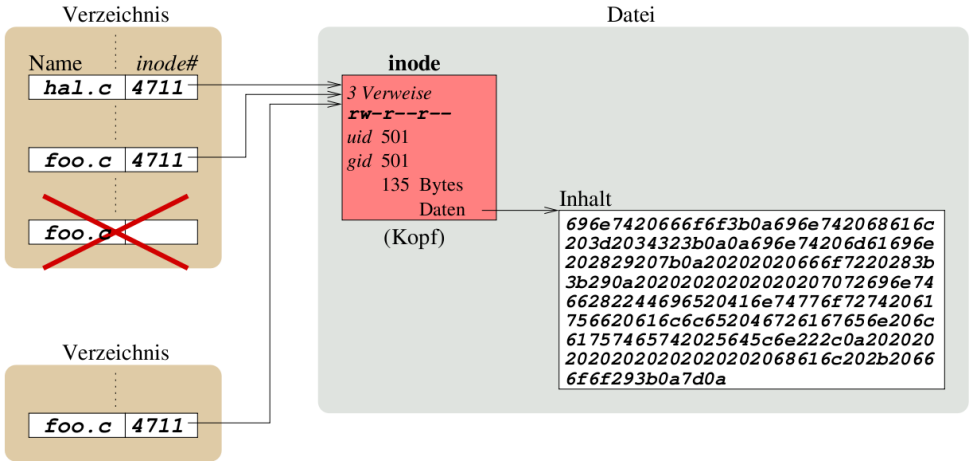
UNIX V7, `dir.h` [4]

```
typedef unsigned short ino_t;

#define DIRSIZ 14

struct direct {
    ino_t d_ino;
    char  d_name[DIRSIZ];
};
```

        - Abbildung als Wertepaar gespeichert
        - mehrere Einträge können auf denselben Dateikopf verweisen
            - Dateikopfnummern identisch
            - Dateinamen verschieden
        - Referenzzähler im Dateikopf für Anzahl Verweise
    - Anlegen/Löschen: nur Schreibzugriffsrecht auf *Verzeichnis* nötig (unabhängig von Zugriffsrechten auf referenzierte Datei)
      Verzeichnis als Spezialdatei:
        - Name, Dateikopf
        - erreichbar über Verknüpfung eines anderen Verzeichnisses



        - Speicherung von Namen getrennt von Dateien
- Referenzzähler (reference count) ↝ Unterstützung der Müllsammlung (garbage collection)
    - Verknüpfungszähler (*link count*; di_nlink) ↝ Buchführung über Anzahl Verknüpfungen zu einem Dateikopf
        - di_nlink != 0: Datei/Verzeichnis wird referenziert → Dateikopf (aus Sicht des Systems) in Benutzung
        - di_nlink == 0: Datei/Verzeichnis nicht referenziert → Dateikopf samt anhängender Daten kann freigegeben werden
    - Veränderung des Verknüpfungszählerwertes bei Eintragen (++)/Löschen (--) von Verknüpfungen im jeweiligen Verzeichnis (Verzeichnisverknüpfung/Dateiverknüpfung)
- UNIX-Dateiverzeichnis und -Datei: Verknüpfung, Dateikopf, Dateiinhalt

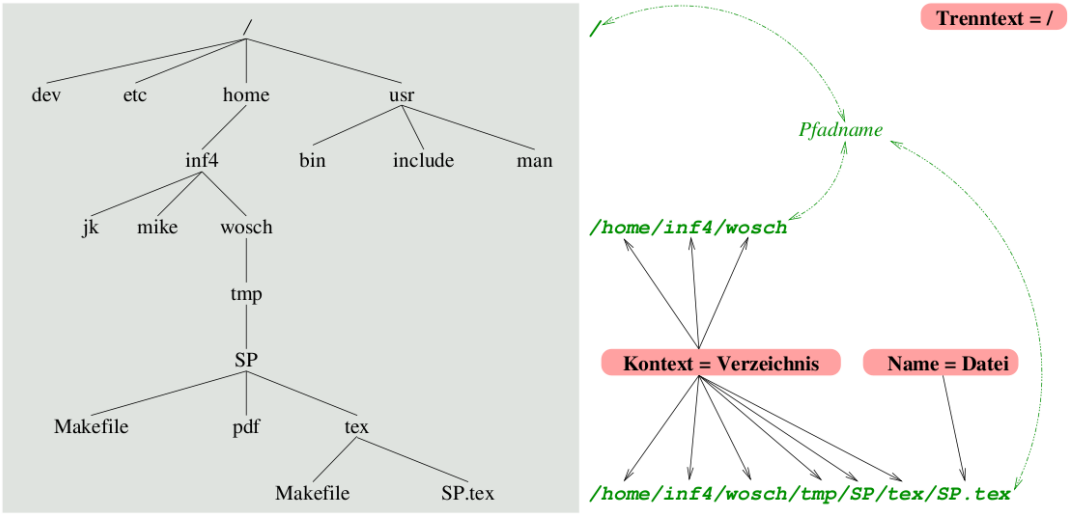


  → Eindeutigkeit von Namenseinträgen in Verzeichnis notwendig
- Dateideskriptor (*file descriptor*): von Dateiverwaltung implementierter eindeutiger Bezeichner (*identifier*) einer geöffneten Datei (meist als Ganzzahl (*integer*) repräsentiert)

7.4 Namensraum

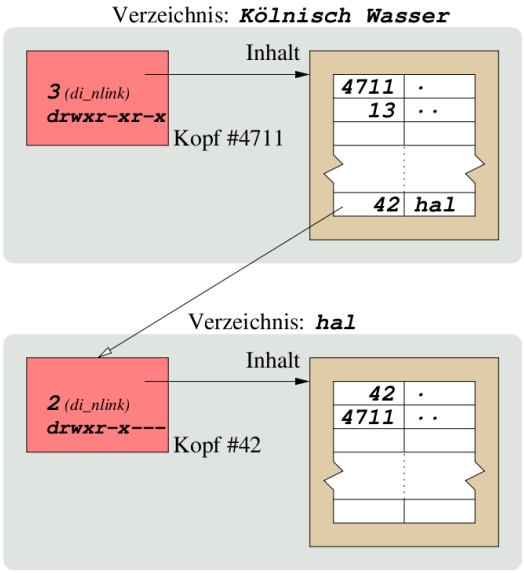
- Kontextfreie Namen bedeutungslos → Zuordnung einer Bedeutung zu einem Namen durch Namensräume (*name spaces*)
- Aufbau von Namensräumen
  - flache Struktur: Definition eines einzigen Kontexts → Gewährleistung der Eindeutigkeit durch Namenswahl selbst
  - hierarchische Struktur: Definition mehrerer Kontexte
    - Eindeutigkeit durch Kontextnamen als Präfix
      - Kontexte enthalten Namen von Dateien und/oder (anderer) Kontexte
      - Name einer Datei entspricht einem „Blatt“ des Namensbaums
    - Sonderzeichen („Trenntext“) meist für Separatoren stehend („/“, „\“)

    Beispiel für Dateibaum (*file tree*)

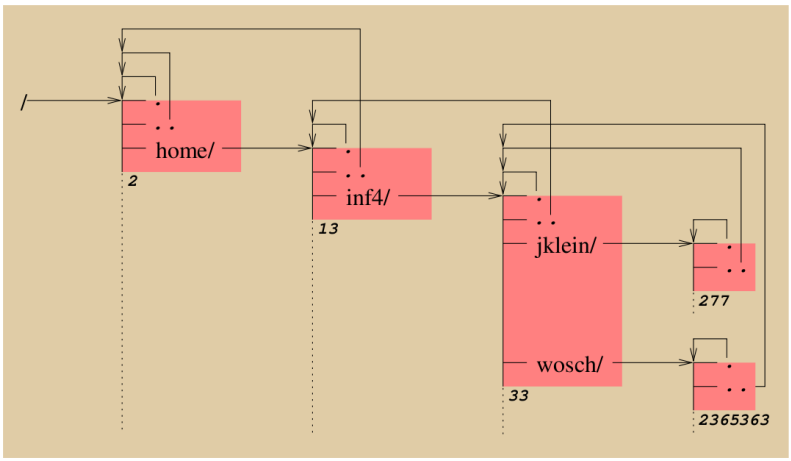


- Navigation im Namensraum: Eindeutigkeit der symbolischen Adresse (einer Datei) durch einen Pfad (*path*) im Namensraum gegeben → Pfadname (*path name*) als vollständiger Dateiname
- spezielle Kontexte: Sonderverzeichnisse
  - Wurzelverzeichnis (*root directory*): Wurzel des Dateibaums, vom Administrator gesetzt (chroot(2), *privilegierte Operation*)
  - Arbeitsverzeichnis (*working directory*): gegenwärtige Position eines Programms/Prozesses im Dateibaum, Änderung beim „Durchklettern“ des Dateibaums (chdir(2))
  - Heimatverzeichnis (*home directory*): initiales Arbeitsverzeichnis eines Benutzers/Prozesses, vom System bei Sitzungsbeginn gesetzt (login(1))
- relative Adressierung von Kontexten: systemdefinierte Verzeichnisnamen
  - .: aktuelles Arbeitsverzeichnis (*current working directory*) ↝ erster Eintrag in jedem Verzeichnis
  - ..: aktuelles Elternverzeichnis ↝ zweiter Eintrag in jedem Verzeichnis (≙ ., falls kein Elternverzeichnis vorhanden (Wurzelverzeichnis))
  - di_nlink: Anzahl Verknüpfungen, die .
    - referenzieren (SunOS, Linux)

- speichert (MacOS)

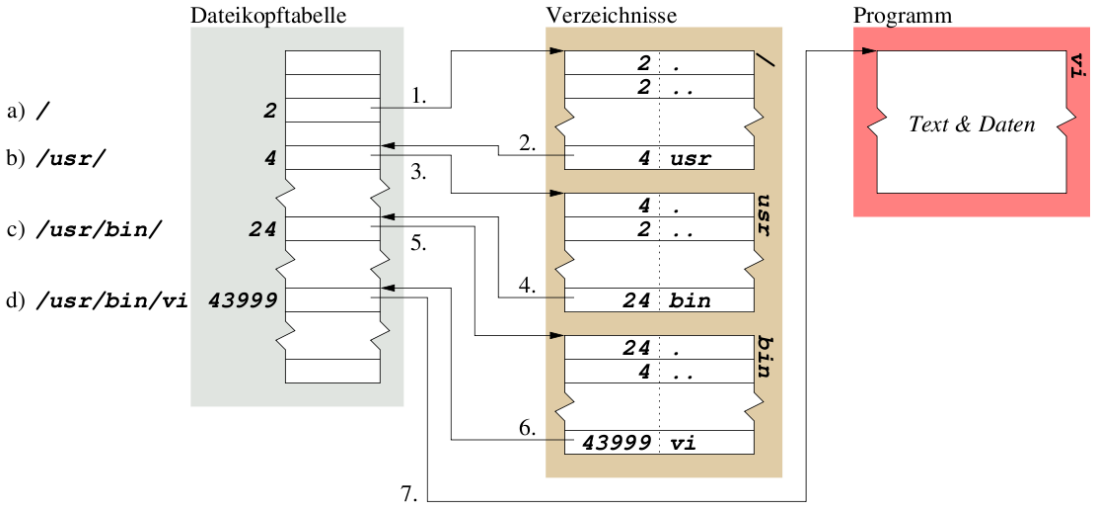


- dynamische Datenstruktur „Dateibaum“
  - Verzeichnisnamen ≙ Vorwärtsverkettung
  - . ≙ Selbstreferenz
  - .. ≙ Rückwärtsverkettung



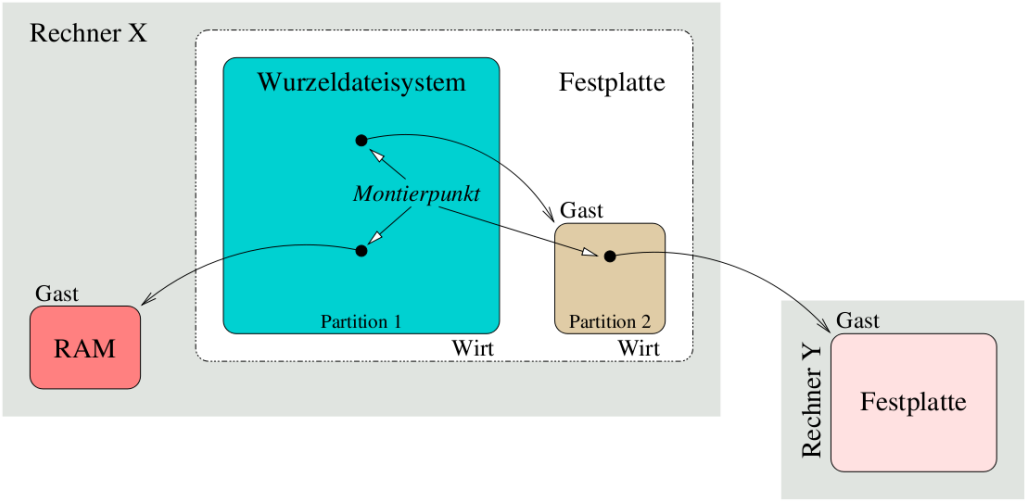
- Arten von Pfadnamen
  - relativer Pfadname: vom aktuellen Arbeitsverzeichnis ausgehend
  - absoluter Pfadname: vom Wurzelverzeichnis ausgehend
- Bindung und Auflösung von Namen/Pfadnamen: Abbildung/Umsetzung: symbolische Adresse ↦ numerische Adresse
  - Namensbindung (*name binding*): Abbildung der symbolischen Adresse in eine numerische Adresse
    → zum Erzeugungszeitpunkt einen Datei-/Verzeichnisnamen…
    - mit einem freien/belegten Dateikopf verknüpfen
    - in ein Dateiverzeichnis eintragen
  - Namensauflösung (*name resolution*): Umsetzung der symbolischen Adresse in eine numerische Adresse
    → zum Benutzungszeitpunkt Dateiverzeichnisse durchsuchen…
    - schrittweise für jeden einzelnen Verzeichnisnamen im Pfad

- schließlich für den Dateinamen

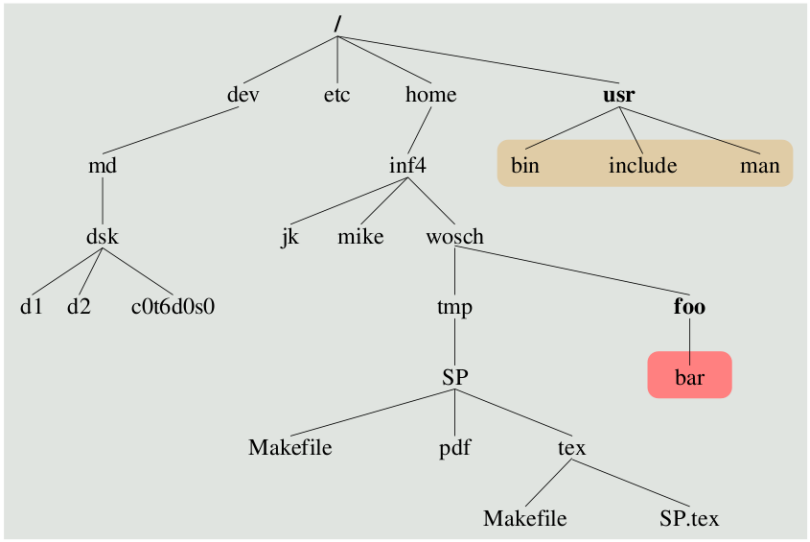


- Dateisystem (*file system*): Verwaltung von Dateien und Dateibäumen
  - Datenstrukturkomplex zur Verwaltung von Hintergrundspeicher
    - Dateisystemkopf (*UNIX super block*)
      - Speichern von Verwaltungsinformationen und Systemparametern
      - Festlegung der Grenzwerte des Dateisystems
    - Dateikopftabelle (*UNIX inode table*)
      - Beschreibung von Dateien und/oder Verzeichnissen
    - Datenblöcke (*data blocks*)
      - Speicherung der Inhalte der Dateien/Verzeichnisse
  - Beschreibung einer Partition (*partition*) im Hintergrundspeicher
    → logische Unterteilung in einen Satz zusammenhängender Sektoren
- Montieren von Dateisystemen: Auf-/Abbau einer Dateisystemhierarchie
  - Montierpunkt (*mount point*): Stelle im Wirtsdateisystem, an der ein Gastdatesystem einebunden werden kann
    → Überlagerung eines Verzeichnises im Wirtsdateisystem mit Wurzel des Gastdateisystems
  - Wirts- und Gastdateisystem bilden jeweils eigene Partitionen
  - Ausgangspunkt: Wurzeldateisystem (root file system, Dateisystem, von dem Betriebssystem urgeladen wird)

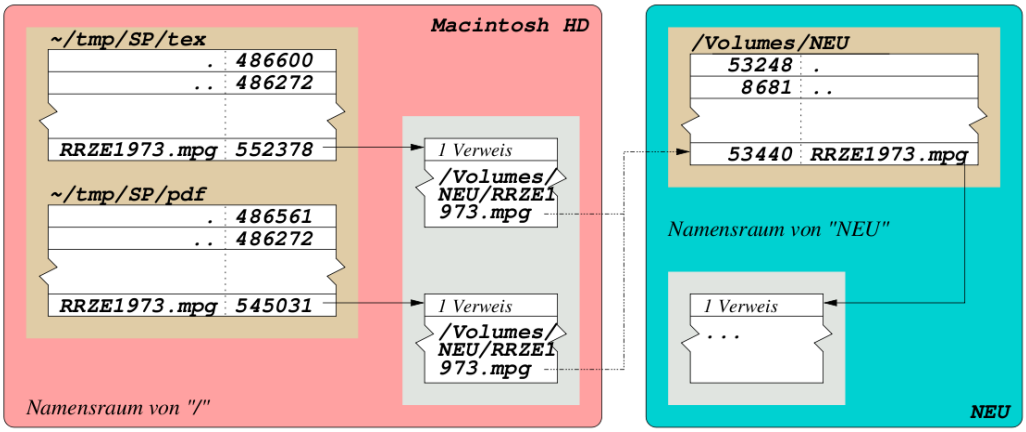
Hierarchiebildung von Dateisystemen



Einbindung von Namensräumen → Integration von Dateibäumen montierbarer Dateisysteme



- Symbolische Verknüpfung (*symbolic link*): Verweis hinein in einen anderen Namensraum



7.5 Prozess

- Prozess ≠ Programm
  - Prozess: kann Ausführung mehrerer Programme bedeuten
    - Anwendungsprogramm ruft Betriebssystemprogramm auf
    - Prozess als Aktivitätsträger mehrerer Programme: Adressraumüberlagerung mit anderem Programm (exec(2))
  - Programm: kann von mehreren Prozessen ausgeführt werden
    - Uniprozessorsystem: nicht-sequentielles Programm
    - Multiprozessorsystem: paralleles Programm
  - Programm statisch, Prozess dynamisch: Wissen über gegenwärtig ausgeführtes Programm wenig aussagekräftig bzgl. gerade im System stattfindender Aktivität
    → Betriebssystemkontext: Konzept „Prozess“ nützlicher als Konzept „Programm“ (Beschreibung/Verwaltung von Abläufen)
- Prozess ≠ Prozessinstanz
  - Prozess: abstraktes Gebilde
    - „Programm in Ausführung“, sequentieller Kontrollfluss
    - „Ablauf“, der eine Verwaltungseinheit ist
  - Prozessinstanz/Prozessinkarnation: konkretes Gebilde
    - die „physische Instanz“ des abstrakten Gebildes „Prozess“
      - an Betriebsmittel (*resource*) gebunden
      - Identität (*identity*) einer Programmausführung
    - die einen Prozess repräsentierende Verwaltungseinheit
      - „dynamische Datenstruktur“ verschiedenartiger Strukturelemente

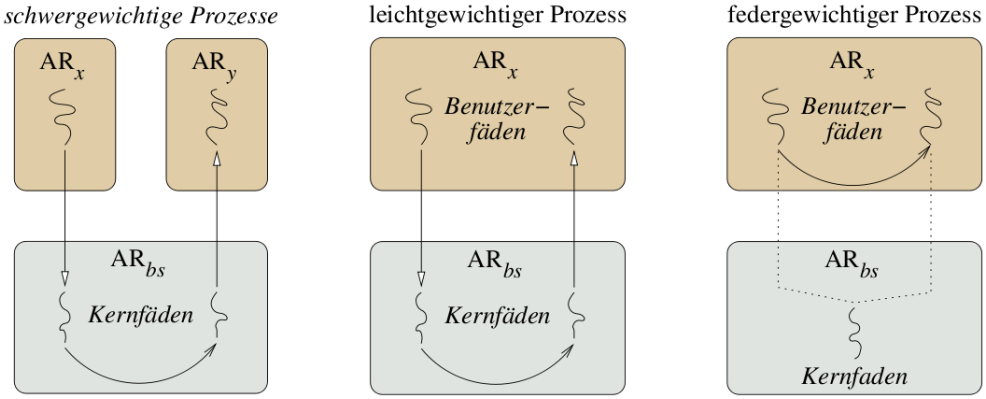
→ Missverständnisse durch synonyme Verwendung?!

- Prozessmodelle/Gewichtsklassen
  - schwergewichtiger Prozess (*heavyweight process*)
    - Prozessinstanz und Benutzeradressraum bilden eine Einheit
    - Prozesswechsel ⇝ zwei Adressraumwechsel: $AR_x \Rightarrow BS \Rightarrow AR_y$

    → „klassischer“ UNIX-Prozess
  - leichtgewichtiger Prozess (*lightweight process*)
    - Prozessinstanz und Adressraum voneinander entkoppelt
    - Prozesswechsel ⇝ ein Adressraumwechsel: $AR_x \Rightarrow BS \Rightarrow Ar_x$

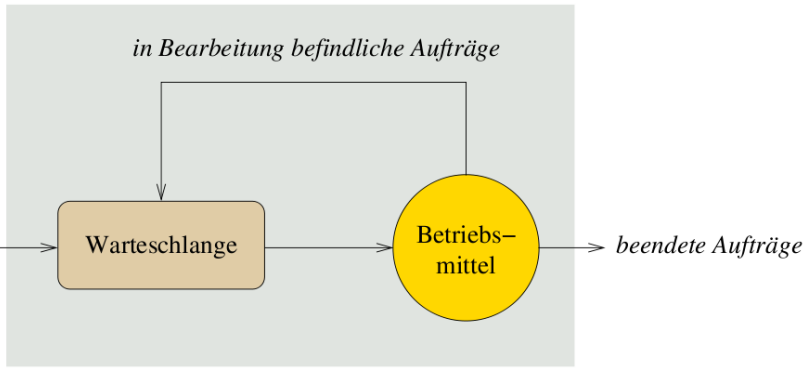
    → Kernfaden (kernel thread)
  - federgewichtiger Prozess (*featherweight process*)
    - Prozessinstanzen und Adressraum bilden eine Einheit
    - Prozesswechsel ⇝ kein Adressraumwechsel: $Ar_x \Rightarrow Ar_x$

    → Benutzerfaden (*user thread*)

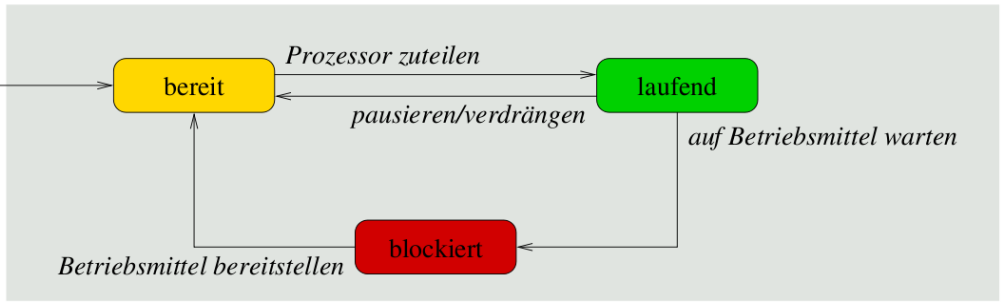


  - Kosten für Adressraumwechsel je nach MMU mehr oder weniger hoch
    - mit jedem Wechselvorgang: Verdrängung der zur Adressrumsetzung benötigten Deskriptoren aus dem Zwischenspeicher (*cache*)
    - erneute Adressraumaktivierung ⇒ erneutes Zwischenspeichern der Adressraumdeskriptoren durch MMU nötig
- Implementierung von Prozessen
  - Basis: federgewichtiger Prozess
    - eigentlicher Kontrollfluss
    - Steuerbefehle ≙ Prozeduren des laufenden Programms
  - Erweiterungen zum Mehrprogrammbetrieb → „Gewichtszunahme“
    - vertikale Isolation vom Betriebssystem: leichtgewichtiger Prozess
      - Steuerbefehle ≙ Systemaufrufe an Betriebssystem
    - horizontale Isolation von anderen Fäden: schwergewichtiger Prozess
      - eigener (logischer/virtueller) Adressraum für jeden Faden
  - Implementierungskonzept von Prozess(instanz)en: Coroutine
- Prozesseinplanung (scheduling)
  - grundsätzliche Fragestellungen
    1. Zeitpunkt der Einspeisung eines Prozesses ins System?
    2. Reihenfolge des Ablaufs der Prozesse?
  - Einplanungsalgorithmus: Strategie, nach der ein von einem Rechensystem zu leistender Ablaufplan zur Erfüllung der jeweiligen Anwendungsanforderungen aufzustellen und zu aktualisieren ist
  - Erstellung eines Ablaufplans (*schedule*) zur Betriebsmittelzuteilung
    - Ordnung nach Ankunft, Zeit, Termin, Dringlichkeit, Gewicht, …
    - der jeweiligen Einplanungsstrategie entsprechend
    - Unterstützung einer bestimmten Rechnerbetriebsart

    → koordinierte Zuteilung der Betriebsmittel zu Prozessen
  - prinzipielle Funktionsweise: Verwaltung von (betriebsmittelgebundenen) Warteschlangen



- Verarbeitungszustände von Prozessen (Zustandsübergänge von Planer (*scheduler*) implementiert)



  Prozessverarbeitung ⇒ Verwaltung mehrerer Warteschlangen
  - häufig: eigene Warteschlangen für ein Betriebsmittel
  - in Warteschlangen stehende Prozesse i. d. R. blockiert
    Ausnahme: Bereitliste (ready list) der CPU: Prozesse laufbereit
- Einplanungsverfahren steht und fällt mit Zieldomäne, Kompromisslösungen geläufig (jedoch nicht immer tragfähig)
  → Scheduling als Querschnittsbelang (*cross-cutting concern*)
- UNIX Scheduling – charakteristische Eigenschaften (Linux, MacOS, SunOS)
  - verdrängende (*preemptive*) Verfahren
    - keine Monopolisierung der CPU durch laufenden Prozess möglich
    - Entziehung der CPU für den laufenden Prozess möglich (CPU-Schutz)
  - fortgeschriebener Ablaufplan nicht-deterministisch
    - nicht zu jedem Zeitpunkt bestimmt, wie weitergefahren wird
    - keine exakte Vorhersage der Prozessorauslastung möglich
  - Prozessausführung und -einplanung gekoppelt (*online*)
    - dynamische Prozesseinplanung während Programmausführung
    - Planungsziel: Minimierung von Antwortzeiten, Förderung von Interaktivität
  - Zeitmultiplexbetrieb (*time sharing*) des Systems

7.6 Koordinationsmittel

- Koordination durch Kommunikation: Interprozesskommunikation (*inter-process communication*, IPC)
  - Erreichen von Fortschritten in Programmverarbeitung: Interaktion von Prozessen zwingend erforderlich
    - implizit: innerhalb des Betriebssystem
      - nebenläufige Ausführung mehrfädiger Systemprogramme
        - asynchrone Programmunterbrechungen
        - verdrängende Prozesseinplanung
        - ggf. auch SMP (*shared multiprocessing*)
      - Konkurrenz der Prozesse um Betriebsmittelzuteilung
    - explizit: innerhalb des Anwendungssystems
      - arbeitsteilige Ausführung eines Programms durch mehrere Fäden → paralleles/verteiltes Programm

- Kooperation der Prozesse zur gemeinsamen Programmausführung
- Nebenläufigkeit/kritischer Abschnitt (*critical section*)
  - nebenläufiges Zählen: i++ nicht immer unteilbare Operation → unterbrechungsbedingte Überlappungseffekte möglich
  - Warteschlangen als Beispiel für potentielle Brennpunkte
    - oft beliebige Permutation der Zugriffsoperationen möglich
    - Auslegung der Datenstruktur „Schlange“ von Bedeutung
  - Einreihung in einfach verkettete Liste
    - Elementaroperation zum Anhängen eines Kettenglieds

```
typedef struct chainlink {
    struct chainlink *link;
} chainlink;

void chain (chainlink **next, chainlink *item) {
    *next = (*next)->link = item;
}
```

Die Funktion hängt ein neues Kettenglied hinter `*next` ein: `(*next)->link = item`. Der Zuweisungswert ist gleichfalls die Adresse des Kettenglieds, an dem das nächste Kettenglied angehängt werden soll. Diese Adresse wird vermerkt: `*next = Wert`. Damit ist `next` Einfügezeiger in eine einfach verkettete Liste.

`gcc -O6 -fomit-frame-pointer -S chain.c`. Auf der Assembliersprachenebene ist erkennbar, dass die Einfügeoperation als Folge von Einzelschritten auf der CPU zur Ausführung kommt. Im Falle der nicht-sequentiellen Ausführung ist nach jedem Einzelschritt mit einer asynchronen Programmunterbrechung zu rechnen, die eventuell die Verdrängung des laufenden Prozesses bewirkt und einen anderen Prozess startet, der dieselbe Funktion zeitlich überlappend (und damit nebenläufig) ausführen könnte.

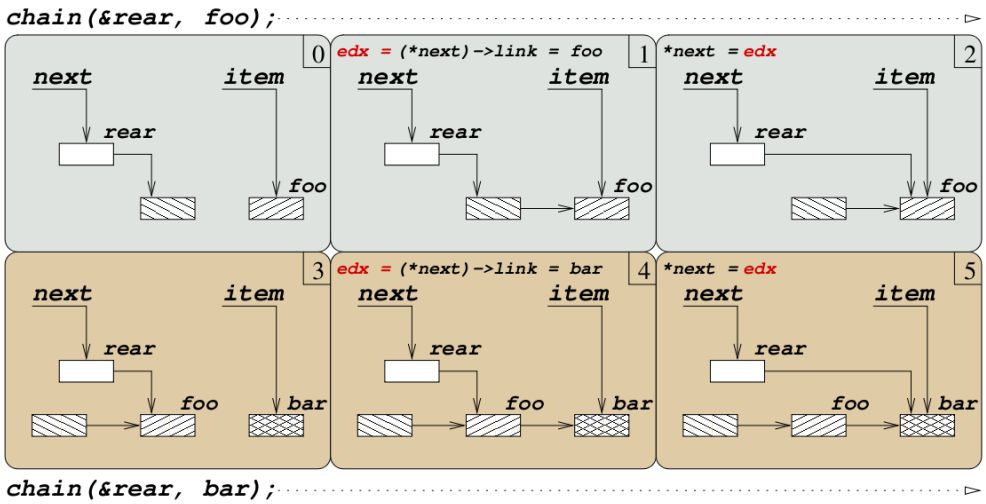
x86

```
chain:
  movl 4(%esp), %ecx
  movl 8(%esp), %edx
  movl (%ecx), %eax
  movl %edx, (%eax)
  movl %edx, (%ecx)
  ret
```

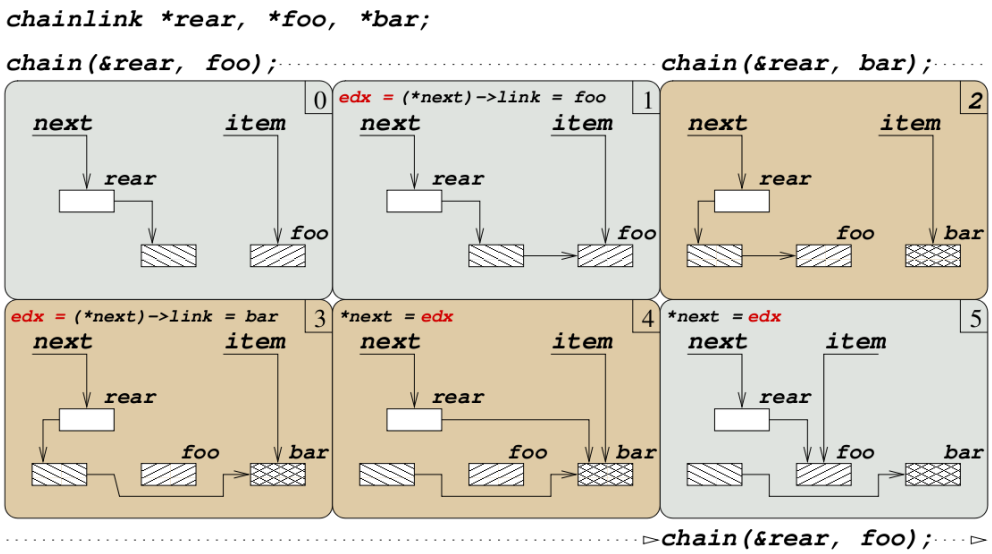
PPC

```
_chain:
  lwz r5,0(r3)
  stw r4,0(r5)
  stw r4,0(r3)
  blr
```

  - sequentielle Ausführung



  - nicht-sequentielle Ausführung



- Koordinationsvariable Semaphor (*semaphore*): nicht-negative ganze Zahl, für die zwei unteilbare Operationen definiert sind:
  - P (*prolaag*, erniedrige; *down*, *wait*)
    - Semaphor == 0: laufender Prozess wird blockiert
    - sonst: Semaphor--
  - V (*verhoog*, erhöhe; *up*, *signal*)
    - Semaphor++
    - auf den Semaphor ggf. blockierte Prozesse werden deblockiert

  → abstrakter Datentyp zur Signalisierung von Ereignissen zwischen gleichzeitigen Prozessen (deren Ausführung sich zeitlich überschneidet)
- Synchronisation (*synchronization*): Sequentialisierung nicht-sequentieller Programme
  → Koordination von Kooperation und Konkurrenz
  - prozessübergreifend das erreichen, wofür innerhalb eines Prozesses Sequentialität von Aktivitäten sorgt
  - gezielte Unterbindung von Nebenläufigkeit/Parallelität

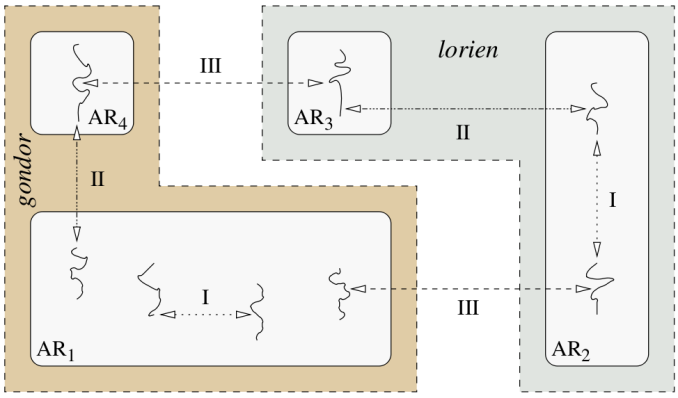
  Gegenseitiger Ausschluss der Listenmanipulation[a]

  [a] `P()` und `V()` klammern den kritischen Abschnitt. Durch `P()` wird erreicht, dass die Anweisungen bis zum `V()` nicht zugleich von mehreren Prozessen ausführbar sind.

```
void chain (chainlink **next, chainlink *item) {
    P();
    *next = (*next)->link = item;
    V();
}
```

  - UNIX: Sequenzen von Semaphoroperationen unteilbar ausführbar (semop(2))
- Kommunikation durch Nachrichten – Motivation zum Botschaftenaustausch (*message passing*)
  - Konsequenzen physikalischer Adressraumtrennung durch MMU:
    - Abschottung der in Ausführung befindlichen Programme
    - Kooperation muss Adressraumgrenzen überwinden können
  - Konsequenzen mehrerer Ausführungskontexte innerhalb eines Adressraums
    - ggf. mehrfädiger (*multi-threaded*) Ablauf von Programmen (→ mehrere Kontrollflüsse)
    - Kooperation muss Kontrollflussgrenzen überwinden können
  - Semaphor
    - ✓ Anzeige: Daten haben einen Prozess verlassen, anderen erreicht
    - ✗ Datenaustausch selbst
- Problemdomänen der Kommunikation
  I. innerhalb desselben Adressraums
  II. zwischen verschiedenen Adressräumen desselben Rechensystems
  III. zwischen verschiedenen Rechensystemen

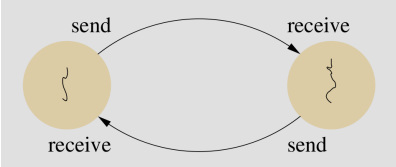
  ⇒ Notwendigkeit domänenspezifischer Kommunikationsmechanismen



- Datenaustausch zwischen Prozessen – prinzipielle Aktionen
  - Datentransfer vom Sende- zum Empfangsadressraum
    → den Prozessen gemeinsamer Kommunikationskanal
  - Synchronisation von Sende- und Empfangsprozess
    - Fortschritt des Empfangsprozesses abhängig von Sendeprozess
      - Nachricht als konsumierbares Betriebsmittel
      - Empfangsprozess: Konsument; Sendeprozess: Produzent
      - Konsum nur nach vorheriges Produktion!
    - Fortschritt des Sendeprozesses abhängig von Empfangsprozess
      - Nachrichtenpuffer als wiederverwendbares Betriebsmittel
      - Füllen des Puffers durch Sendeprozess, Leeren durch Empfangsprozess
      - Füllen erfordert freien Platz ↝ Leeren
    - Koordination implizit durch angewandte Primitive
- Kommunikationssemantiken – Primitiven zum Botschaftenaustausch
  - Wirkung von **Sendeprimitiven** auf ausführenden Prozess je nach Grad der Synchronisation unterschiedlich:
    - no-wait send: Sendeprozess wartet, bis Nachricht im Transportsystem zum Absenden bereitgestellt worden ist
      → Interprozesskommunikation im Vorübergehen (→ Pufferung)
    - synchronization send: Sendeprozess wartet, bis Nachricht vom Empfangsprozess angenommen wurde
      → Rendezvous Sender ↔ Empfänger (→ keine Pufferung)
    - remote-invocation send: Sendeprozess wartet, bis Nachricht vom Empfangsprozess verarbeitet und beantwortet ist
      → Fernaufruf einer vom Empfangsprozess auszuführenden Funktion
  - Wirkung von **Empfangsprimitiven** auf ausführenden Prozess gleich: Warten auf Eintreffen einer Nachricht von einem Sendeprozess
- Kommunikationsmodelle
  - gleichberechtigte Kommunikation: dieselbe Rolle für miteinander kommunizierende Prozesse

$$P_1 \left\{ \begin{array}{ccc} send & \longrightarrow & receive \\ receive & \longleftarrow & send \end{array} \right\} P_2$$

    P_1/P_2: Sender *und* Empfänger
    - no-wait send oder synchronized send
    - beteiligte Prozesse Produzent *und* Konsument



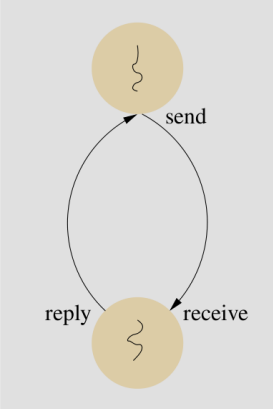
      - send: Bereitstellung eines konsumierbaren Betriebsmittels
        in Identifikation des Empfängers (Konsument)
        in Basis/Länge der Nachricht
      - receive: Anforderung eines konsumierbaren Betriebsmittels

in Basis/Länge eines Empfangspuffers
out Identifikation des Senders
- Ermittlung der Empfängeradresse zur Nachrichtenzustellung nötig → Namensdienst, lookup (Senderadresse implizit)

- ungleichberechtigte Kommunikation (remote-invocation send): verschiedene Rollen für miteinander kommunizierende Prozesse P_1/P_2: P_2 Dienstgeber/Anbieter (*server*), P_1 Dienstnehmer/Klient (*client*)

$$\text{(Klient) } P_1 \left\{ \begin{array}{ccc} send & \longrightarrow & receive \\ & \longleftarrow & reply \end{array} \right\} P_2 \text{ (Anbieter)}$$

  - send: Anforderungsnachricht
    in Identifikation des Anbieters
    in Basis/Länge der Nachricht
    in Basis/Länge des Empfangspuffers
    out Identifikation eines Anbieters
  - receive: Anforderungsnachricht
    in Basis/Länge des Empfangspuffers
    out Identifikation des Klienten
  - reply: Antwortnachricht
    in Identifikation des Klienten
    in Basis/Länge der Nachricht



- Senke der Interprozesskommunikation: Adressierung des Kommunikationspartners – direkt ↔ indirekt
  - Faden (*thread*): Konsument der Nachricht
    - direkte Adresse der die Nachricht verarbeitenden Instanz → Prozessidentifikation (PID)
  - Tor (*port*): Anschluss zur Weiterleitung/Zustellung von Nachrichten, der einem bestimmten Prozess zugeordnet ist
    - mehrere Anschlüsse pro Prozess möglich → Ein-/Ausgangstore für Nachrichten
    - statische oder dynamische Zuordnung
  - Briefkasten (*mailbox*): Zwischenspeicher für Nachrichten, der durch Senden gefüllt und Empfangen geleert wird
    - Pufferbereich keinem Prozess zugeordnet
    - *N* Prozesse können dahin senden und daraus empfangen
- Kommunikation und Betriebsmittel: synchrone vs. asynchrone Interprozesskommunikation
  - Synchronisation von Prozessen zur Kommunikation durch Anforderung/Bereitstellung von Betriebsmitteln
    - Sender → wiederverwendbares Betriebsmittel „Puffer“ benötigt
      - synchrone IPC ⇒ Zielpuffer (des Empfängers)
      - asynchrone IPC ⇒ Zwischenpuffer
    - Empfänger → konsumierbares Betriebsmittel „Nachricht“ benötigt
      - asynchrone IPC ⇒ Zwischenpuffer
      - synchrone IPC ⇒ Quellpuffer (des Senders)
  - Betriebsmittelmangel als Ursache für ggf. Blockade von Prozessen bei Kommunikation
    - Empfänger erwartet Nachricht, Sender erwartet freien Puffer

- „asynchron“ ≠ „nicht-blockieren“ oder „wartefrei“

- Verbindungen zwischen kommunizierenden Prozessen: Garantie von Gütemerkmalen (quality of service)
  - IPC: hier Torverbindungen nutzen, Verlauf in 3 Phasen
    - Aufbauphase: Einplanung der zur Durchsetzung der jeweils angeforderten Gütemerkmale notwendigen Betriebsmittel (→ Puffer, Fäden, Bandbreite, …, Protokoll)
    - Nutzungsphase: Botschaftenaustausch gemäß Gütemerkmale
    - Abbauphase: Freigabe der reservierten (eingeplanten) Betriebsmittel, Auflösen der Verbindung

Richtung/Betriebsart verbindungsorientierter Kommunikation

Richtung				Betriebsart
unidirektional	Tor_s	$\longrightarrow$	Tor_r	halbduplex
bidirektional	Tor_{sr}	$\longleftrightarrow$	Tor_{rs}	vollduplex

- Botschaftenaustausch mit UNIX-Systemfunktionen
  - Sendeprimitive: jede Aktivierung fordert Socket und Puffer an → Betriebsmittel nicht garantiert
  - Empfangsprimitive: nur für Dauer von recvfrom(2) (lokale)PID des Empfängers an (globalen) Namen gebunden → Sendeversuche können scheitern
  - Protokolldateneinheit (*protocol data unit*, PDU) und Namensfunktion
    - Nachricht als Dienstdateneinheit (service data unit, SDU), Transport in einer PDU vom

```
typedef struct ipc_message {
    pid_t ipc_from;
    char  ipc_data[0];
} message;
```

Schablone mit der PID des Senders und einem generischen Feld zur Aufnahme der SDU.

Sender zum Empfänger

    - Kommunikationsendpunkt: Prozessinstanz, aus deren PID eine symbolische Adresse (*receiver ID*, RID) generiert wird

```
#define RIDSZ (int)((sizeof(pid_t) * 2.5) + 3)

inline char* pid2rid (pid_t pid, char rid[]) {
    sprintf(rid, "rid%u", pid);
    return rid;
}
```

Die RID wird im UNIX Namensraum abgelegt (`bind(2)`).

  - UNIX-Kommunikationsfunktionen „considered harmful“: typisch „Allgemeinzweck, dennoch problematisch für IPC

Pros

- Datenstromkonzept
  - *no-wait send* Semantik, kanalorientiert
- in E/A-Schnittstelle integriert
  - `read(2)` empfängt, `write(2)` sendet
  - `close(2)` gibt *Socket* frei, `unlink(2)` entfernt *Socket*-Verknüpfung
- verschiedenste Protokolle und Betriebsarten

Cons

- rein asynchrones Modell
  - *synchronization/remote-invocation send* fehlt
- sehr komplexe Schnittstelle
  - knifflige, problemorientierte Lösungen auf Bibliotheksebene
  - synchrone IPC nur sehr aufwändig nachbildbar
- man muss mit Kanonen auf Spatzen schießen…

→ gut für schwergewichtige Klient/Anbieter-Systeme

7.7 Zusammenfassung

- Betriebssysteme bieten einige nützliche Abstraktionen
  - Adressräume, Speicher, Dateien, Namensräume

    - Prozesse, Koordinationsmittel
- Abbildung von Namen auf Adressen von zentraler Bedeutung
    - symbolische ↦ numerische ↦ logische ↦ virtuelle ↦ physikalische
    - Fortführung des Konzepts im Kontext von Rechnernetzen
- Einplanung von Prozessen
    - Planung des zeitlichen Ablaufs der Prozessorzuteilung
    - allgemein: Planung der Zuteilung von Betriebsmitteln an Prozesse
- Gewichtsklassen von Prozessen
    - schwer-, leicht-, federgewichtig
    - u. a. Frage der Verzahnung von Prozessinstanz und Adressraum
- Koordination von Prozessen für Mehrprozessbetrieb erforderlich

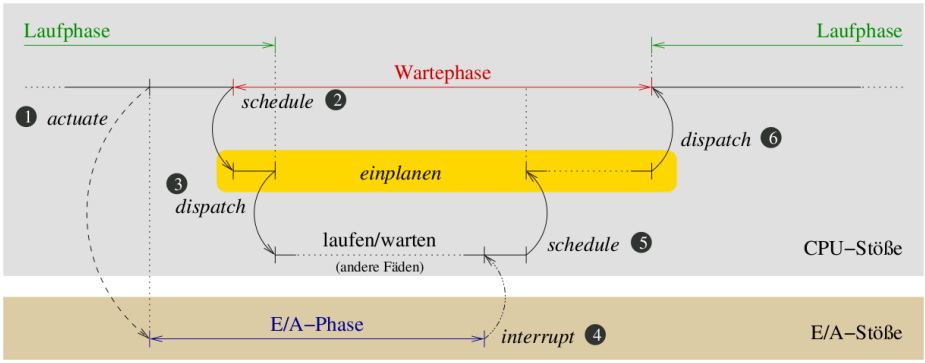
8. Zwischenbilanz

9. Prozesseinplanung

9.1 Prozessorzuteilungseinheit

- Programmfaden: Einplanungseinheit (*unit of scheduling*) für CPU-Vergabe
    - Ablaufplanung von Fäden
        - betriebsmittelorientiert
        - ereignisgesteuert oder zeitgesteuert
    - Einplanung ≠ Einlastung
        - Einplanung: Vorgang der Reihenfolgebildung von Aufträgen
        - Einlastung: Moment der Zuteilung von Betriebsmitteln

      → Vorgänge ent- oder gekoppelt (zeitversetzt/zeitgleich)
    - Fadenverläufe: Stoßbetrieb (*burst mode*)
        - Laufphase: CPU-Stoß (*CPU burst*), aktive Phase/Faden eingelastet → alle erforderlichen Betriebsmittel verfügbar
        - Wartephase: E/A-Stoß (*I/O burst*) im weitesten Sinn, inaktive Phase → nicht alle erforderlichen Betriebsmitteln verfügbar
            - Warten auf konsumierbare/nicht-konsumierbare Betriebsmittel
            - Bereitstellung der Betriebsmittel durch andere Fäden
    - Durchlaufen eines Kontrollflusses zur Einplanung und Einlastung anderer Fäden



1. laufender Faden stößt E/A-Vorgang an (*actuate*)
2. passives Warten auf Beendigung der E/A (*schedule*) → Anforderung eines Betriebsmittels
3. Einlastung eines eingeplanten, laufbereiten Fadens (*dispatch*)
4. Signalisierung der Beendigung der E/A (*interrupt*) → Bereitstellung des konsumierbaren Betriebsmittel „Signal“
5. Einplanung des auf dieses Ereignis wartenden Fadens (*schedule*)
6. Einlastung des Fadens, sobald an der Reihe (*dispatch*)

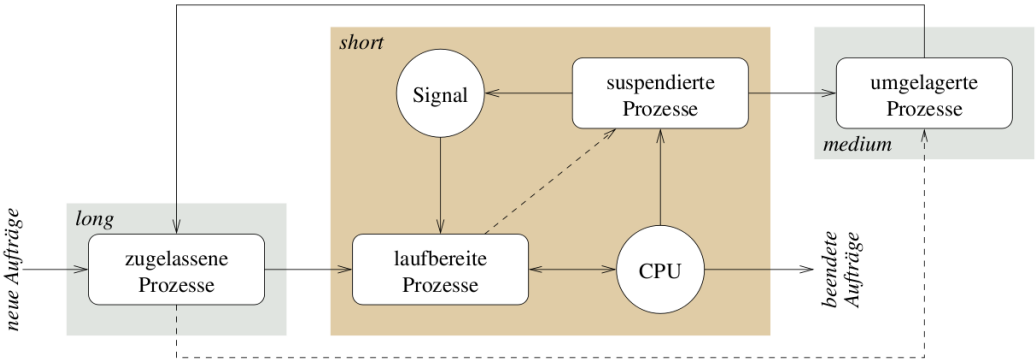
- Sonderfall: „CPU im Leerlauf“ (*idle state*) (keine anderen laufbereiten Fäden) → Selbsteinplanung, -lastung eines Fadens

- Arbeitsteilung in nebenläufigen/parallelen Systemen → Fäden als Mittel zur Leistungsoptimierung
  - Überlappung von Lauf-/Wartephasen ⇒ Erhöhung der Rechnerauslastung
  - Auslastsung von CPU *und* Peripherie (E/A-Geräte) gesteigert
    1 CPU-Stoß pro CPU parallel zu vielen E/A-Stößen

  #Prozessoren < #Fäden ⇒ Serialisierung der Fäden
- Zwangsserialisierung von Programmfäden bzgl. einer Instanz des Betriebsmittel CPU
  - Verlängerung der absoluten Ausführungsdauer später „eintreffender“ laufbereiter Fäden
  - Vergrößerung der mittleren Verzögerung proportional zur Fadenanzahl
- subjektive Empfindung der Fadenverzögerung: Überlast durch zuviel eingeplante Fäden vermeiden
  - Wartephase bei E/A-Operationen dominieren Fadenverzögerung

9.2 Ebenen der Prozessorzuteilung

- Dauerhaftigkeit von Zuteilungsentscheidungen – logische Ebenen der Prozesseinplanung
  - langfristige Einplanung (*long-term scheduling*) [s – min]
    - Lastkontrolle, Grad an Mehrprogrammbetrieb einschränken
    - Programme laden und/oder zur Ausführung zulassen
    - Prozesse der mittel-/kurzfristigen Einplanung zuführen
  - mittelfristige Einplanung (*medium-term scheduling*) [ms – s]
    - Teil der Umlagerungsfunktion (*swapping*)
    - Programme vom Hinter- in Vordergrundspeicher bringen
    - Prozesse der langfristigen Einplanung zuführen
  - kurzfristige Einplanung (*short-term scheduling*)
    - Einlastungsreihenfolge der Prozesse festlegen – obligatorisch



  - kurzfristige Einplanung als Voraussetzung für Mehrprozessbetrieb
    - laufbereite Prozesse erwarten Zuteilung des wiederverwendbaren Betriebsmittel „CPU“ (Start der Laufphase)
    - suspendierte Prozesse erwarten Zuteilung eines konsumierbaren Betriebsmittel „Signal“ (Ende der Wartephase)

9.3 Zustandsübergänge

- logischer Zustand eines Prozesses in Abhängigkeit von Einplanungsebene
  - kurzfristig (*short-term*): bereit, laufend, blockiert
  - mittelfristig (*medium-term*, *mid-term*): schwebend bereit, schwebend blockiert
  - langfristig (*long-term*): erzeugt, gestoppt, beendet

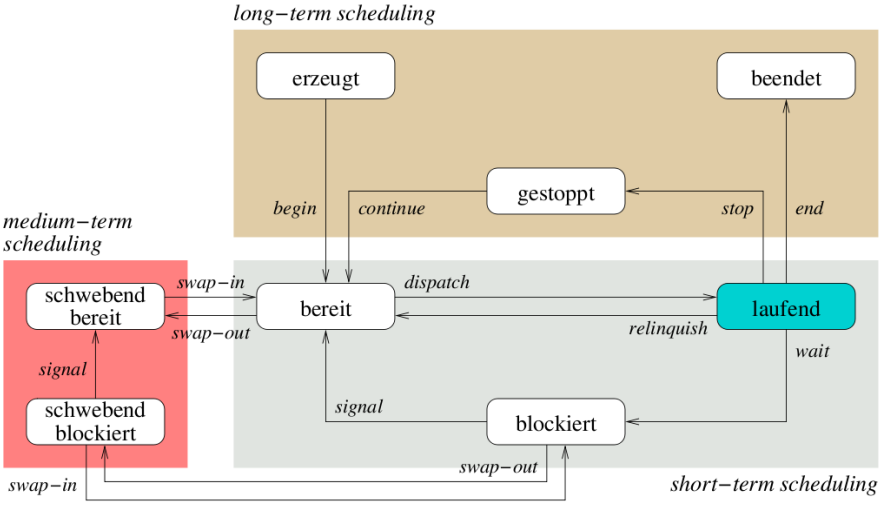
  Anwendungsfall ⇒ durch Betriebssystem zur Verfügung gestellte Einplanungsebenen (nicht ⇐)
- kurzfristige Einplanung → Festlegung der Prozessorzuteilungsreihenfolge
  → Betriebssystem bietet Mehrprozessbetrieb auf Basis der Serialisierung von Programmfäden
  - bereit (*ready*) zur Ausführung durch Prozessor (CPU)
    - Prozess auf Bereitliste (ready list)

    - Einplanungsverfahren bestimmt Listenposition
  - laufend (*running*), erfolgte Zuteilung des Betriebsmittel „CPU“
    - Prozess vollzieht CPU-Stoß
    - ein laufender Prozess zu einem Zeitpunkt pro CPU
  - blockiert (*blocked*) auf ein bestimmtes Ereignis
    - Prozess erwartet Zuteilung eines Betriebsmittels (außer „CPU“)
    - ggf. Vollziehen des E/A-Stoßes

  Speziallfall: „blockiert“ ↦ „laufend“ bei voll verdrängender Einlastung
- mittelfristige Einplanung → Festlegung der Umlagerungsreihenfolge
  → Betriebssystem implementiert Umlagerung (*swapping*) von kompletten Programmen/logischen Adressräumen
  - schwebend bereit (*ready suspend*)
    - Adressraum des Prozesses ausgelagert
      - in Hintergrundspeicher verschoben
      - „*swap-out*“ erfolgt
      - „*swap-in*“ erwartet
    - Einlastung des Prozesses (→ aller Fäden des Adressraums) außer Kraft
  - schwebend blockiert (*blocked suspend*)
    - ausgelagerter ereigniserwartender Prozess
    - Ereigniseintritt ↦ „schwebend bereit“

  „schwebend bereit“ ↦ „bereit“ ↦ langfristige Einplanung
- langfristige Einplanung → Festlegung der Zulassungsreihenfolge
  → Betriebssystem mit Funktion zur Lastkontrolle, steuert Grad an Mehrprogrammbetrieb
  - erzeugt (*created*) und fertig zur Programmverarbeitung
    - Prozess instantiiert, Programm zugeordnet
    - ggf. Speicherzuteilung noch ausstehend
  - gestoppt (*stopped*) und erwartet seine Fortsetzung
    - Prozess angehalten → Überlast, Verklemmungsvermeidung, …
  - beendet (*ended*) und erwartet seine Entsorgung
    - Prozess terminiert, Betriebsmittelfreigabe erfolgt
    - Vollendung des „Kehraus“ ggf. durch anderen Prozess

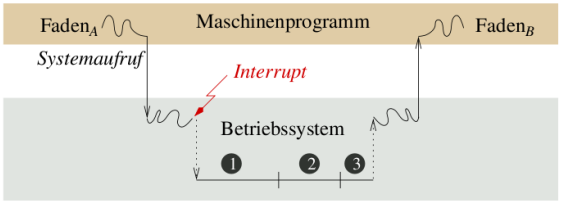
  ⚠ auch bereite/blockierte Prozesse stoppbar
- Abfertigungszustände im Zusammenhang → weitere Zwischenzustände in Abhängigkeit vom Betriebssystem



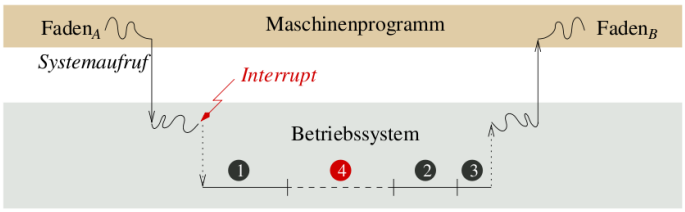
- Einplanungs-/Auswahlzeitpunkt
  - Zustandsübergang nach „bereit“ ⇒ Aktualisierung der Bereitliste
    - Entscheidung über Einlastungsreihenfolge
    - Ausführung einer Funktion der Einplanungsstrategie
  - Einplanung (*scheduling*)/Umplanung (*rescheduling*)
    - Prozess erzeugt → *begin*
    - Prozess gibt freiwillig CPU ab → *relinquish*

        - von Prozess erwartetes Ereignis tritt ein → *signal*
        - Prozess kann wieder aufgenommen werden → *continue*
- verdrängende (*preemptive*) Prozesseinplanung ⇒ Prozesse zur CPU-Abgabe drängbar
    - Verdrängung:
        1. Eintritt eines Ereignisses, dessen Behandlungsverlauf zum Planer führt
           → Einplanung des das Ereignis ggf. erwartenden Prozesses
        2. Einplanung des (vom Ereignis unterbrochenen laufenden Prozesses
        3. Auswahl und Einlastung eines eingeplanten Prozesses
    - Latenzzeit zwischen Ereigniseintritt und Einplanung, Einlastung abhängig von Betriebssystem(kern)architektur (ggf. Verdrängung eines Prozesses verzögert!)
      ⇒ ggf. Beeinträchtigung von Anwendungen
- Latenzzeiten und Determinismus – Verdrängung als Querschnittsbelang von Betriebssystemen
    - Einplanungslatenz (*scheduling latency*): an sich unvermeidbar, die Unbestimmtheit jedoch nicht → deterministische Einplanung:
        - zu jedem Zeitpunkt: nachfolgender Schritt festgelegt (unabhängig von aktueller/-n Systemlast/-aktivitäten
        - Latenzzeit konstant oder mit fester oberer Schranke variabel
    - Einlastungslatenz (*dispatching latency*): Zeitspanne zwischen Einplanung und Verdrängung
        - verdrängend (*preemptive*): „programmierte Verdrängung“
            - Freigabe der Einlastung nur an bestimmten Stellen
            - Verdrängungspunkte (*latency points*)
        - voll verdrängend (*full preemptive*): Einlastung jederzeit möglich
    - Unterbrechungslatenz (*interrupt latency*)

voll verdrängend



1. Behandlung
   - Interrupt
2. Einplanung
3. Einlastung
4. Synchronisation



verdrängend

    - Latenzzeiten in Bezug zu Betriebsmodus → asynchrone Programmunterbrechungen als Quelle der Ungewissheit

9.4 Gütemerkmale

- Kriterien zur Aufstellung der Einlastungsreihenfolge von Prozessen
    - benutzerorientiert → Benutzerdienlichkeit
      → vom Benutzer wahrgenommen, Akzeptanz des Systems
        - Antwortzeit
        - Durchlaufzeit
        - Termineinhaltung
        - Vorhersagbarkeit
    - systemorientiert → Systemperformanz
      → effektive/effiziente Auslastung der Betriebsmittel, Rentabilität des Systems
        - Durchsatz
        - Prozessorauslastung
        - Gerechtigkeit

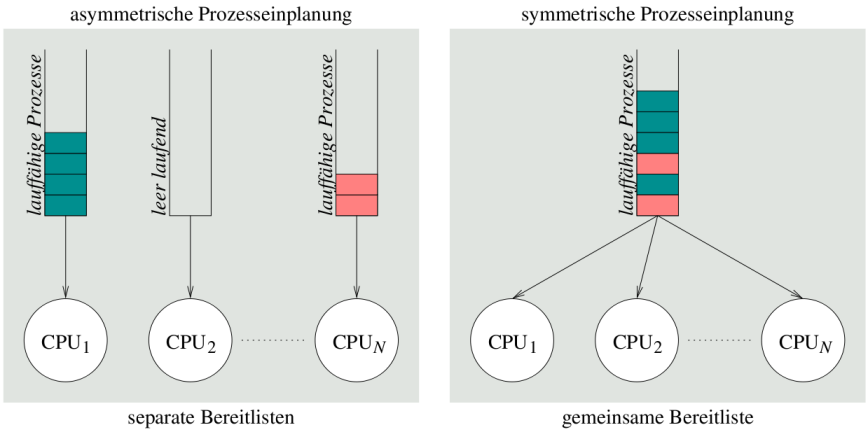
        - Dringlichkeiten
        - Lastausgleich

    → Schwerpunktsetzung, kein gegenseitiger Ausschluss!
- Betriebsart vs. Einplanungskriterien: Prozesseinplanung impliziert eine Betriebsart und umgekehrt
    - allgemein: Gerechtigkeit, Lastausgleich
      → Durchsetzung der jeweiligen Strategie
    - Stapelbetrieb: Durchsatz, Durchlaufzeit, Prozessorauslastung
    - interaktiver Betrieb: Antwortzeit, Proportionalität
      → der (oft auch falschen) inhärenten Vorstellung der Benutzer über die Dauer bestimmter Aktionen sollte das System möglichst entsprechen (→ Benutzerakzeptanz)
    - Echtzeitbetrieb: Dringlichkeit, Termineinhaltung, Vorhersagbarkeit → ⚡ Gerechtigkeit/Lastausgleich

9.5 Verfahrensweisen

- kooperativ ↔ präemptiv: Souveränität bei Anwendung oder Betriebssystem
    - *cooperative scheduling* → Prozesse voneinander abhängig
        - Monopolisierung der CPU durch „unkooperative“ Prozesse möglich
        - Systemaufrufe während Programmausführung → ☠ Endlosschleifen ohne Systemaufrufe (kein anderer Prozess bekommt CPU)
        - alle Systemaufrufe müssen Scheduler durchlaufen
    - *preemptive scheduling* → Prozesse voneinander unabhängig
        - CPU-Entzug zugunsten anderer Prozesse
        - ereignisbedingte Verdrängung des laufenden Prozesses von CPU
        - direkte/indirekte Aktivierung des Schedulers durch Ereignisbehandlung
        - CPU-Schutz: keine CPU-Monopolisierung möglich
- deterministisch ↔ probabilistisch: mit oder ohne à-priori-Wissen
    - *deterministic scheduling* → Prozesse bekannt, exakt vorberechnet
        - CPU-Stoßlängen, ggf. auch Termine bekannt (strikte Echtzeitsysteme: ≥ *worst case execution time*, WCET)
        - keine genaue Vorhersage der CPU-Auslastung möglich
        - Sicherstellung des Einhaltens von Zeitgarantien (unabhängig von Systemlast)
    - *probabilistic scheduling* → Prozesse unbekannt
        - exakte CPU-Stoßlängen, ggf. auch Termine unbekannt (lediglich Abschätzung der CPU-Auslastung möglich)
        - keine Zeitgarantien möglich (durch Anwendung sicherzustellen)
- statisch ↔ dynamisch: (Ent-)Kopplung von/mit Programmausführung
    - *offline scheduling* → statisch, *vor* Programmausführung
        - Komplexität → ☠ Ablaufplanung im laufenden Betrieb
        - vollständiger Ablaufplan als Ergebnis der Vorberechnung (→ Quelltextanalyse/Übersetzer, oft zeitgesteuert abgearbeitet als Teil der Prozesseinlastung)
        - zumeist Beschränkung auf strikte Echtzeitsysteme
    - *online scheduling* → dynamisch, *während* Programmausführung
        - Stapelsysteme, interaktive Systeme, verteilte Systeme
        - schwache, feste Echtzeitsysteme
- asymmetrisch ↔ symmetrisch → Programmausführung gebunden/ungebunden an eine CPU
    - *asymmetric scheduling* → Abhängigkeit von Eigenschaften der Ebene 2/3
        - oblilgatorisch in asymmetrischem Multiprozessorsystem (→ Rechnerarchitektur mit programmierbaren Spezialprozessoren)
        - optional in symmetrischem Multiprozessorsystem
        - (Zwang) zur ungleichen Verteilung (in funktionaler Hinsicht)
    - *symmetric scheduling* → Abhängigkeit von Eigenschaften der Ebene 2
        - identische Prozessoren (alle gleich geeignet zur Programmausführung)
        - Lastausgleich: gleiche Verteilung der Prozesse auf Prozessoren

    → jedem Prozessor eigene Bereitliste ↔ allen Prozessoren gemeinsame Bereitliste



- ungleichmäßige Auslastung
- Unterbrechungssynchronisation
- gleichmäßige Auslastung
- Multiprozessorsynchronisation

9.6 Grundlegende Strategien

- Überblick über klassische Einplanungs-/Auswahlverfahren

kooperativ	*First Come, First Served*	gerecht
verdrängend	• *Round Robin* • *Virtual Round Robin* → wer zuerst kommt, mahlt zuerst	reihum
probabilistisch	• *Shortest Process Next* (*Shortest Job Next*) • *Shortest Remaining Time First* • *Highest Response Ratio Next* → die Kleinen nach vorne	priorisierend
mehrstufig	• *MultiLevel Queue* • *FeedBack* • *MultiLevel Feedback Queue* → Rasterfahndung	

- *First Come First Served*: fair, einfach zu implementieren (FIFO-Queue), …, dennoch problematisch
  - Einplanung/Verarbeitung der Prozesse nach Ankunftszeit (*arrival time*)
    → nicht-verdrängend, Kooperation als Voraussetzung
  - Gerechtigkeit zu Lasten hoher Antwortzeit, niedrigem E/A-Durchsatz
    → ☹ Mix von kurzen und langen CPU-Stößen

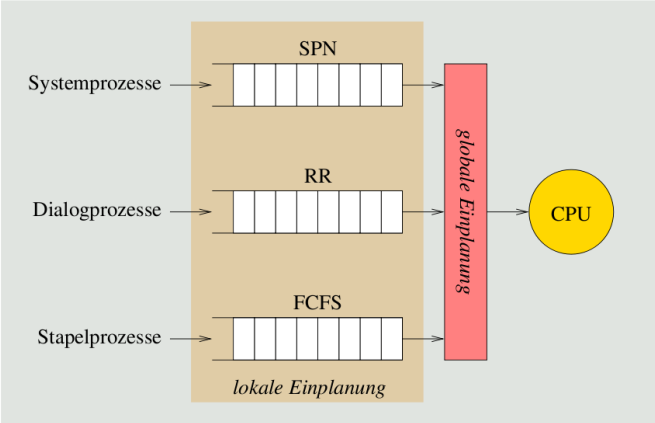
    Prozesse mit { langen / kurzen } CPU-Stößen werden { begünstigt / benachteiligt }
  - Konvoi(d)ef(f)ekt: ☠ mehrere kurze Aufträge folgen einem langen…
    oft lange normalisierte Durchlaufzeit: Bedienzeit ÷ Durchlaufzeit
- *Round Robin*: verdrängendes FCFS, Zeitscheiben, CPU-Schutz
  - Einplanung nach Ankunftszeit, regelmäßige (periodische) Umplanung
    - verdrängend, periodische Unterbrechungen (asynchrone Programmunterbrechungen durch Zeitgeber (*timer*))
    - Zuteilung einer Zeitscheibe (*time slice*) pro Prozess → obere Schranke für CPU-Stoßlänge eines laufenden Prozesses
  - Verringerung der Benachteiligung für Prozesse mit kurzen CPU-Stößen (i. Vgl. zu FCFS)
    - Effektivität durch Zeitscheibenlänge bestimmt

zu lang/Degenerierung zu FCFS → *etwas länger als typischer CPU-Stoß* ← zu kurz/hoher Mehraufwand
    - Leistungsprobleme bei Mix von Prozessen
        - E/A-intensive Prozesse: Bedarf < Zeitscheibe ⇒ freiwilliges Beenden des CPU-Stoßes vor Ablauf der Zeitscheibe
        - CPU-intensive Prozesse: Bedarf ≥ Zeitscheibe ⇒ unfreiwilliges Beenden des CPU-Stoßes durch Verdrängung

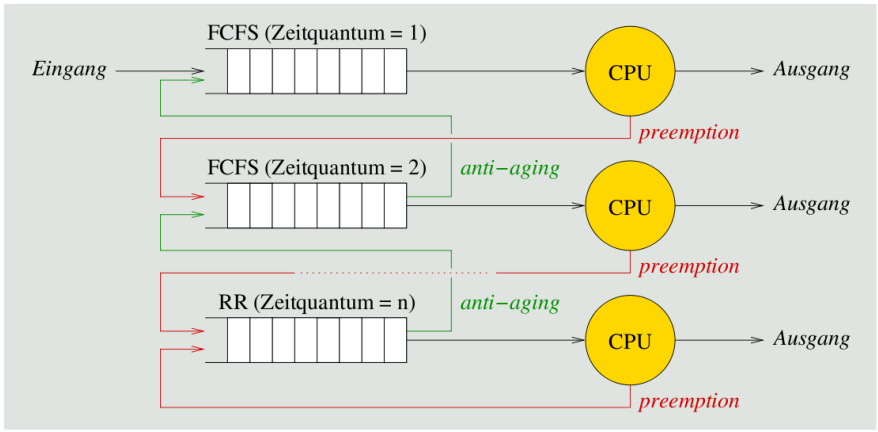
      → Konvoi(d)ef(f)ekt: mehrere kurze CPU-Stöße folgen einem langen
        - ungleiche Verteilung der CPU-Zeit zu Gunsten CPU-intensiver Prozesse → E/A-Geräte schlechter ausgelastet

      → Varianz der Antwortzeit E/A-intensiver Prozesse groß
- *Virtual Round Robin*: RR mit Vorzugswarteschlange und variablen Zeitscheiben
    - bevorzugte Einplanung nach Beendigung des E/A-Stoßes (jedoch: nicht zwingend bevorzugte Einlastung)
        - Einreihung in eine der Bereitliste vorgeschaltete Vorzugsliste
          → FIFO ↝ evtl. Benachteiligung hoch-interaktiver Prozesse ⇒ aufsteigende Sortierung nach Zeitscheibenrest eines Prozesses
        - Umplanung bei Beendigung des jeweils laufenden CPU-Stoßes
          → zuerst Einlastung der Prozesse auf Vorzugsliste → CPU-Zuteilung für Rest ihrer Zeitscheibe → Einreihung in Bereitliste
    - Vermeidung von ungleicher Verteilung der CPU-Zeiten (bei RR möglich)
      → durch strukturelle Maßnahmen: Bevorzugung interaktiver Prozesse mit kurzen CPU-Stößen
- *Shortest Process Next:* Zeitreihen bilden, analysieren und verwerten
    - Einplanung nach erwarteter Bedienzeit
        - Grundlage: à-priori-Wissen über Prozesslaufzeiten
            - Stapelbetrieb: Programmierer setzt Frist (*time limit*)
            - Produktionsbetrieb: Erstellung einer Statistik durch Probeläufe
            - Dialogbetrieb: Abschätzung von CPU-Stoßlängen zur Laufzeit
        - Abarbeitung einer aufsteigend nach Prozesslaufzeiten sortierten Bereitliste
          → Abschätzung vor (statisch) oder zur (dynamisch) Laufzeit
    - Verkürzung von Antwortzeiten und Steigerung der Gesamtleistung des Systems auf Kosten länger laufender Prozesse
      → Verhungern (*starvation*) dieser Prozesse möglich
- *Shortest Remaining Time First:* verdrängendes SPN, Verhungerungsgefahr, Effektivität von VRR
    - Einplanung nach erwarteter Bedienzeit, sporadische Umplanung in unregelmäßigen Zeitabständen
      → Verdrängung, wenn erwartete CPU-Stoßlänge eines eintreffenden Prozesses < verbleibender CPU-Stoßlänge des laufenden Prozesses
    - Umplanung ereignisbedingt und (ggf. voll) verdrängend
      → bei Aufhebung der Wartebedingung für einen Prozess
    - Verdrängung ⇒ bessere Antwort- und Durchlaufzeiten
      → Overhead zur CPU-Stoßlängenabschätzung i. Vgl. zu VRR
- *Highest Response Ratio Next:* SRTF ohne Verhungern der Prozesse
    - Einplanung nach erwarteter Bedienzeit, periodische Umplanung unter Berücksichtigung der Wartezeit
      → periodische Aktualisierung aller Einträge in Bereitliste
    - Alterung (*aging*) von Prozessen: Anstieg der Wartezeit
      → Alterung entgegenwirken (*anti-aging*) beugt Verhungern vor
- *MultiLevel Queue*: unterstützt Mischbetrieb: Verder- und Hintergrundbetrieb
    - Einplanung nach Typ (≙ für zutreffend geglaubte Eigenschaften)
        - Aufteilung der Bereitliste in separate („getypte“) Listen (z. B. System-/Dialog-/Stapelprozesse)
        - lokale Einplanungsstrategie mit jeder Liste verbinden (z. B. SPN, RR, FCFS)
        - globale Einplanungsstrategie zwischen den Listen
            - statisch: feste Zuordnung einer Liste zu bestimmter Prioritätsebene (→ Verhungerungsgefahr für Prozesse tiefer liegender Listen!)

        - dynamisch: Wechseln der Listen im Zeitmultiplexverfahren (→ 40% System-, 40% Dialog-, 20% Stapelprozesse)
    - ⚠ Typzuordnung ≙ statische Entscheidung ↝ Zeitpunkt der Prozesserzeugung



- *FeedBack*: Begünstigung kurzer/interaktiver Prozesse, ohne relative Stoßlängen kennen zu müssen
    - Einplanung nach Ankunftszeit, periodische Umplanung
        - Hierarchie von Bereitlisten, je nach Prioritätsebenen
          → erstmalig eintreffender Prozesse steigt oben ein, Zeitscheibenablauf drückt laufenden Prozess weiter nach unten
        - je nach Ebene verschiedene Einreihungsstrategien und -parameter
          → unterste Ebene RR, andere nach FCFS; Zunahme der Zeitscheibengröße von unten nach oben
    - Bestrafung (*penalization*) von Prozessen mit langen CPU-Stößen
        - Prozess mit kurzen CPU-Stößen → relativ schneller Durchlauf
        - Prozess mit langen CPU-Stößen → relativ langsamer Durchlauf
          (ggf. Alterung entgegenwirken: Anheben von Prozessen)



- Prioritäten setzende Verfahren: statische Prioritäten (MLQ) ↔ dynamische Prioritäten (VRR, SPN, SRTF, HRRN, FB)
    - Prozessvorrang = bevorzugte Einlastung von Prozessen mit höherer Priorität
      Bestimmung erfolgt
        - statisch: zum Zeitpunkt der Prozesserzeugung ↝ Laufzeitkonstante
            - keine Veränderung im weiteren Verlauf
            - Erzwingung einer deterministischen Ordnung zwischen Prozessen
        - dynamisch: zum Zeitpunkt der Prozessausführung ↝ Laufzeitvariable
            - Berechnung durch Betriebssystem (ggf. in Kooperation mit Anwendungsprogrammen)
            - kein Erzwingen einer deterministischen Ordnung zwischen Prozessen
    - Echtzeitverarbeitung ⇒ Prioritäten setzende Verfahren
        - aber: nicht jedes solcher Verfahren zum Echtzeitbetrieb geeignet
        - Einplanung muss deterministisches Laufzeitverhalten liefern (entsprechend jeweiliger Anforderungen der Anwendungsdomäne)
- Gegenüberstellung von Strategien und Verfahrensweisen: kooperativ/verdrängend ↔

probabilistisch/deterministisch

	FCFS	RR	VRR	SPN	SRTF	HRRN	FB
kooperativ	✓			✓			
verdrängend		✓	✓		✓	✓	✓
probabilistisch				✓	✓	✓	
deterministisch	keine bzw. nicht von sich aus allein ⇝ EZS (S. 1-3)						

- MLQ umfasst Eigenschaften der in dem Verfahren vereinigten Strategien
  - Priorisierung von Strategien liefert Nuancen im laufzeitverhalten
  - speziellen Anwendungsanforderungen (teilweise) entgegenkommen: z. B.: FCFS priorisieren ⇝ „*number crunching*“ fördern

9.7 Fallstudien

9.8 Zusammenfassung

- Einplanung ⇝ Einlastungsreihenfolge von Prozessen
  → Zuteilung von Betriebsmitteln an konkurrierenden Prozesse
  - Betriebssysteme treffen Zuteilungsentscheidungen auf drei Ebenen:
    - *long-term scheduling*: Lastkontrolle des Systems
    - *medium-term scheduling*: Umlagerung von Programmen
    - *short-term scheduling*: Einlastungsreihenfolge von Prozessen
  - die Entscheidungskriterien haben verschiedene Dimensionen:
    - Benutzer: Antwort-/Durchlaufzeit, Termine, Vorhersagbarkeit
    - System: Durchsatz, Auslastung, Gerechtigkeit, Dringlichkeit, Lastausgleich
  - Prozesseinplanung kennt z. T. sehr unterschiedliche Verfahrensweisen
    - kooperativ/verdrängend, deterministisch/probabilistisch
    - entkoppelt/gekoppelt, asymmetrisch/symmetrisch
  - Dimension, Kriterium und Verfahrensweise ⇝ Einplanungsstrategien
    → FCFS, RR, VRR, SPN, SRTF, HRRN, MLQ, FB (MLFQ)

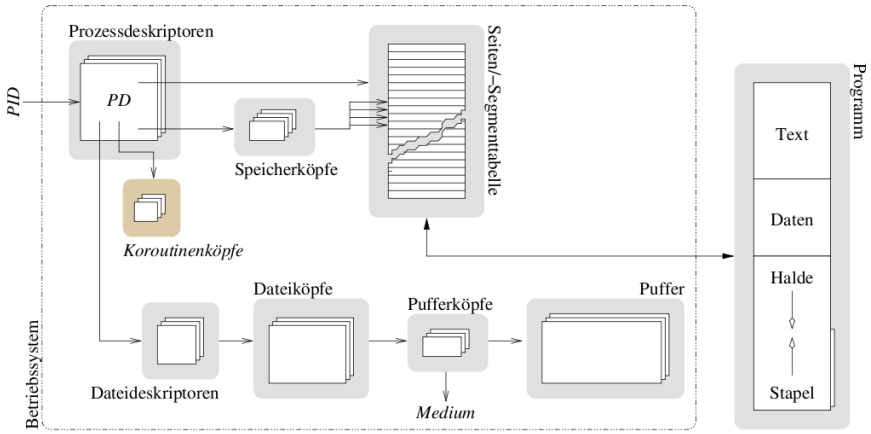
10. Prozesseinlastung

10.1 Coroutine

10.2 Eigenständige Coroutine

10.3 Programmfaden

10.4 Prozessdeskriptor



einfädiger Prozess $\mapsto$ 1 Koroutinenkopf
mehrfädiger Prozess $\mapsto$ $N > 1$ Koroutinenköpfe

10.5 Zusammenfassung

11. Synchronisation

11.1 Konkurrenz und Koordination

11.2 Verfahrensweisen

11.3 Schlossvariable

11.4 Bedingungsvariable

11.5 Semaphor

11.6 Monitor

11.7 Zusammenfassung

12. Verklemmungen

12.1 Grundlagen

12.2 Fallstudie

12.3 Vorbeugung

12.4 Vermeidung

12.5 Erkennung und Erholung

11.6 Monitor

12.7 Zusammenfassung

13. Adressräume

13.1 Ausgangspunkt

- Adressraumkonzepte – Betriebssystemsicht

- physikalischer Adressraum ⊩ nicht-linear adressierbarer E/A- und Speicherbereich, dessen Größe der Adressbreite der CPU entspricht
  - 2^N Bytes bei N Bits Adressbreite
  - von Lücken durchzogen ↝ ungültige Adressen
- logischer Adressraum ⊩ linear adressierbarer Speicherbereich von 2^M Bytes bei N Bits Adressbreite
  - M = N z. B. bei Harvard-Architektur (getrennter Programm-, Daten-, E/A-Adressraum)
  - M < N sonst
- virtueller Adressraum ⊩ logischer Adressraum von 2^K Bytes bei N Bits Adressbreite
  - K > N bei Speicherbankumschaltung, Überlagerungstechnik
  - K ≤ N sonst

13.2 Physikalischer Adressraum

Adressbereich	Größe (KB)	Verwendung
00000000–0009ffff	640	RAM (System)
000a0000–000bffff	128	Video RAM
000c0000–000c7fff	32	BIOS Video RAM
000c8000–000dffff	96	keine
000e0000–000effff	64	BIOS Video RAM (*shadow*)
000f0000–000fffff	64	BIOS RAM (*shadow*)
00100000–090fffff	147456	RAM (Erweiterung)
09100000–fffdffff	4045696	keine
fffe0000–fffeffff	64	SM-RAM (*system management*)
ffff0000–ffffffff	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

- Adressraumorganisation – Adressraumbelegung (*address assignments*)
- ungültige Adressen: Zugriff ⇒ Busfehler (*bus error*)
  - reservierte Adressen: Zugriff ⇒ Schutzfehler (*protection fault*)
  - freie Adressen: Hauptspeicher (*main memory*)

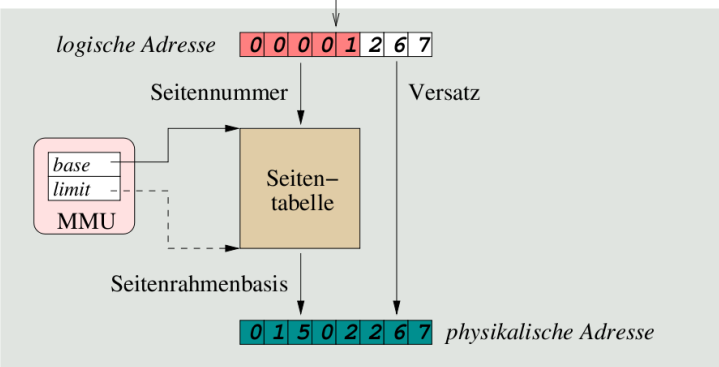
13.3 Logischer Adressraum

- Segmentierung: logische Unterteilung von Programmadressräumen
  - logische Aufteillung von Ebene-4-Programmen in mindestens zwei Segmente:
    - Text → Maschinenanweisungen, Programmkonstanten
    - Daten → initialisierte Daten, globale Variablen, Halde
  - Ebene-3-Programme: mindestens ein weiteres Segment:
    - Stapel → lokale Variablen, Hilfsvariablen, aktuelle Parameter
  - Betriebssysteme → Verwaltung der Segmente im physikalischen Adressraum
    - ggf. mit Hilfe einer MMU (*memory management unit*) (Abbildung logische ↦ physikalische Adresse)
      → Verwaltung des Speichers durch Betriebssystem
    - MMU legt Organisationsstruktur auf physikalischen Adressraum: Unterteilung in Seiten fester/Segmente variabler Länge
- Ausprägungen von Programmadressräumen
  - eindimensional in Seiten aufgeteilt (*paged*)
    - Programmadresse A_P bildet Tupel *(p, o)*:
      - $p = A_P$ div 2^N ↝ Seitennummer (*page number*)
      - $o = A_P \bmod 2^N$ ↝ Versatz (*byte offset*)

      mit $2^N \triangleq$ Seitengröße (*page size*) in Bytes
    - Seite ↦ Seitenrahmen (auch: Kachel) des physikalischen Adressraums
  - zweidimensional in Segmente aufgeteilt (*segmented*)
    - Programmadresse A_S bildet Paar (S, A)
      - Adresse A relativ zu Segment(name/nummer) S
      - bei seitennummerierten Segmenten: Interpretation von A als A_P

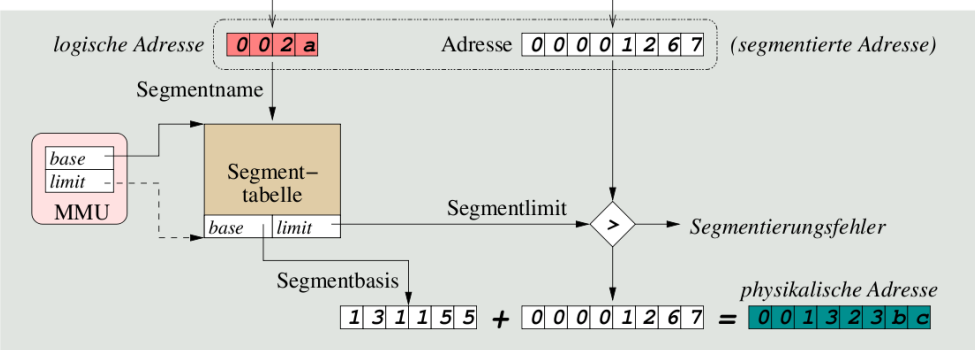
    - Segment ↦ Folge von Bytes/Seitenrahmen des physikalischen Adressraums
- Adressumsetzung
  - seitennummerierter Adressraum (*paged address space*)
    - Seitennummer als Index in Seitentabelle
      - pro Prozess
      - dimensioniert durch MMU
    - ungültiger Index ⇒ Trap
    - indizierte Adressierung (der MMU) liefert Seitendeskriptor
      - enthält Seitenrahmennummer, die die Seitennummer ersetzt
      - entspricht Basisadresse des Seitenrahmens im physikalischen Adressraum
    - setzen der Basis-/Längenregister der MMU ↝ Adressraumwechsel

```
char *p = 4711;     /* 0x1267 */
```

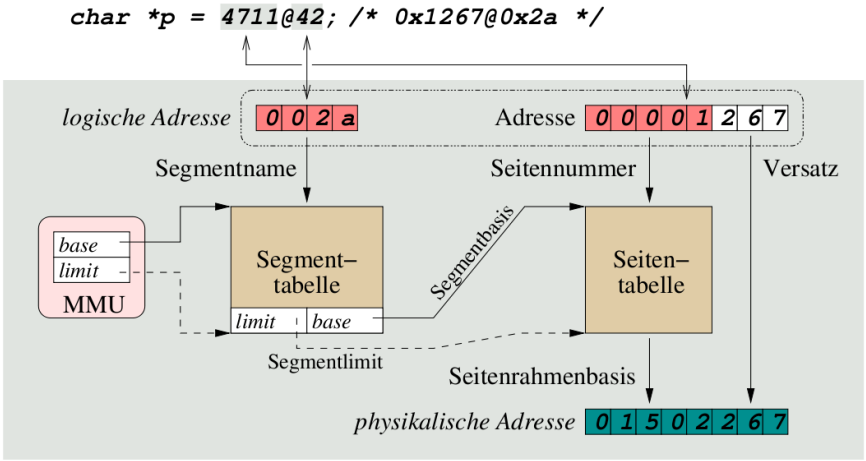


  - segmentierter Adressraum (*segmented address space*)
    - Segmentname ≙ Index in Segmenttabelle eines Prozesses
      → dimensioniert durch MMU, ungültiger Index ⇒ Trap
    - indizierte Adressierung (der MMU) ergibt Segmentdeskriptor
      - enthält Basisadresse und Länge des Segments (*base*/*limit*)
      - $address_{phys}$ = address > limit ? Trap : base + address

```
char *p = 4711@42; /* 0x1267@0x2a */
```



  - segmentierter seitennummerierter Adressraum (*page-segmented address space*)
    - Reihenschaltung von zwei Adressumsetzungseinheiten der MMU:
      - Segmenteinheit löst eine segmentierte Adresse auf (→ adressiert und begrenzt Seitentabelle)
      - Seiteneinheit generiert die physikalische Adresse
    - jeder Prozess hat (mind.) eine Segment- und eine Seitentabelle



- Segmentregister bzw. Segmentselektor (*segment selector*)
  - je nach Art des Speicherzugriffs selektiert MMU implizit passendes Segment
    - Befehlsabruf (*instruction fetch*) ↦ *code*
    - Operandenabruf (*operand fetch*)
      - Direktwerte ↦ *code*
      - globale/lokale Daten ↦ *data*/*stack*

  ☞ Programme können weiterhin 1-dimensionale logische Adressen verwenden
- Seiten- bzw. Segmentdeskriptor: Abbildung steuernder Verbund
  - Adressumsetzung basierend auf Deskriptoren der MMU, die für jede Seite/Segment eines Prozesses Relokations- und Zugriffsdaten verwalten
    - Basisadresse des Seitenrahmens/Segments im physikalischen Adressraum
    - Zugriffsrechte des Prozesses (rwx)
  - Segmente von variabler, dynamischer Größe (≠ Seiten) ⇒ zusätzliche Verwaltungsdaten nötig ⇝ Segmentdeskriptor
    - Segmentlänge, um Segmentverletzungen abfangen zu können (Basis-/Längenregister ⊂ Segmentdeskriptor)
    - Expansionsrichtung: Halde ↑, Stapel ↓
  - Deskriptorprogrammierung zur Programmlade- und -laufzeit
    - bei Erzeugung/Zerstörung schwer- und leichtgewichtiger Prozesse
    - bei Anforderung/Freigabe von Arbeitsspeicher
- Seiten- bzw. Segmenttabelle: Adressraum beschreibende Datenstruktur
  - Deskriptoren des Adressraums eines Prozesses sind in Tabelle im Arbeitsspeicher zusammengefasst
    - Arbeitsmenge (*working set*) von Deskriptoren eines Prozesses: im Zwischenspeicher (*cache*) gehalten
      → TLB (*translation lookaside buffer*) der MMU
    - Adressraumwechsel als Folge eines Prozesswechsels:
      1. Zerstören der Arbeitsmenge (TLB *flush*; teuer, schwergewichtig)
      2. Tabellenwechsel (Zeiger umsetzen; billig, federgewichtig)
  - Basis-/Längenregister (*base*/*limit register*)
    - Beschreibung einer Tabelle → exakt 1 Prozessadressraum
    - Indexprüfung bei Adressumsetzung: *descriptor* = *index* ≤ *limit* ? &*base*[*index*] : *Trap*
      (*index* ≙ Seitennummer/Segmentname)

13.4 Virtueller Adressraum

- Adressraumabbildung einhergehend mit Ein-/Ausgabe: Integration von Vorder- und Hintergrundspeicher
  - Abstraktion von Größe und Örtlichkeit des verfügbaren Hauptspeichers
    - vom Prozess nicht benötigte Programmteile auslagerbar (→ Hintergrundspeicher)
    - Prozessadressraum könnte über Rechnernetz verteilt sein (Programmteile über

Hauptspeicher anderer Rechner verstreut)
    - Zugriffe auf ausgelagerte Programmteile durch Prozessor abgefangen: *Trap*
        - partielle Interpretation durch Betriebssystem
        - unterbrochener Prozess zu E/A-Stoß gezwungen
            - Erwartung der erfolgreichen Einlagerung eines Programmteils
            - ggf. sind anderen Programmteile aus Hauptspeicher zu verdrängen
        - Wiederaufnahme des CPU-Stoßes ↝ Wiederholung des Zugriffs
    - zusätzliche Attribute bei Seiten-/Segmentdeskriptor
      → Adressumsetzung unterliegt Steuerung, die transparent für zugreifenden Prozess die Einlagerung auslöst
        - Gegenwart eines Segments/einer Seite → *present bit*
            - 0 ↦ ausgelagert; Trap, partielle Interpretation, Einlagerung
                - Basisadresse = Adresse im Hintergrund
                - nach Einlagerung: *present bit* ≔ 1
            - 1 ↦ eingelagert; Befehl abrufen, Operanden lesen/schreiben
                - Basisadresse = Adresse im Vordergrund
                - nach Auslagerung: *present bit* ≔ *0*
            - gelegentlich noch andere Zwecke für Gegenwartsbit
                - z. B. Zugriffszähler, Zeitstempel für Zugriffe
                - Betriebssystemmaßnahmen zur Optimierung der Ein-/Auslagerung
- Zugriffsfehler: Seitenfehler (*page fault*)/Segmentfehler (*segmentation fault*) → *present bit* = 0
  ⇒ Behandlungsaufwand im Betriebssystem ⇒ Leistungsverlust (je nach Befehlssatz, Adressierungsarten beträchtlich!; Beispiel für schlimmsten Fall siehe Vorlesungsfolien)
    - Aufwandsabschätzung eines Seitenfehlers siehe Vorlesungsfolien
        - effektive Zugriffszeit (*effective access time*, *eat*) abhängig von Seitenfehlerwahrscheinlichkeit, direkt proportional zu Seitenfehlerrate
        - mittlere Zugriffszeit (*mean access time*, *mat*) abhängig von effektiver Seitenzugriffszeit, Seitengröße
    - Seitenfehler ≙ nicht-funktionale Programmeigenschaft, nicht wirklich transparent!
- Seitenüberlagerung „*considered harmful*“ → Pro und Contra
    - virtuelle Adressräume sind …
        - vorteilhaft wenn übergroße/gleichzeitig mehrere Programme in Anbetracht zu knappen Hauptspeichers auszuführen sind
        - ernüchternd wenn durch Virtualisierung bedingter Mehraufwand zu berücksichtigen ist und sich für ein gegebenes Anwendungsszenario als problematisch bis unakzeptabel erweisen sollte
    - Seitenfehler sind …
        - nicht wirklich transparent, wenn zeitliche Aspekte relevant (→ Echtzeitverarbeitung, Hochleistungsrechnen etc.)
        - erst zur Laufzeit ggf. entstehende nicht-funktionale Eigenschaften

13.5 Zusammenfassung

- Adressräume – Ebenen der Abstraktion
    - physikalischer Adressraum → gültige und ungültige Adressen
        - ungültige Adressen: Zugriff ⇒ Busfehler
        - gültige Adressen: Zugriff gelingt, jedoch: reservierte Adressbereiche ↝ Schutz
    - logischer Adressraum: → gültige Adressen
        - Zugriffsrechte der Prozesse stecken Gültigkeitsbereiche ab (Zugriff auf reservierte Adresse ⇒ Schutzfehler)
        - Prozesse in ihrem Programmadressraum abgeschottet/isoliert (Zugriff auf fremde freie Adressen ⇒ Schutzfehler)
    - virtueller Adressraum → flüchtige Adressen
        - Bindung der Adressen zu Speicherzellen nicht fest
        - Bindung variiert phasenweise zwischen Vorder- und Hintergrundspeicher

- Adressraumdeskriptoren – seitennummerierte und segmentierte Adressräume
  - Abbildung steuernde Verbunde zur Erfassung einzelner Adressraumteile
    - speicher Attribute von Seiten oder Segmente (→ Relokations-, Zugriffsdaten, Zugriffsrechte)
    - bilden Seiten fester Größe auf gleichgroße Seitenrahmen ab (→ seitennummerierter Adressraum)
    - bilden Segmente variabler Größe auf Byte- oder Seitenfolgen ab (→ segmentierter und ggf. seitennummerierter Adressraum)
  - Abbildungstabellen fassen Deskriptoren (eines Adressraums) zusammen
    - Tabellen liegen im Arbeitsspeicher des Betriebssystems (→ erforderlicher Speicherbedarf evtl. beträchtlich)
    - Zwischenspeicherung der Arbeitsmengen von Deskriptoren im TLB (→ ein *Cache* der MMU, ohne den Adressumsetzung ineffizient ist)
  - Zugriffsfehler als intransparente, nicht-funktionale Eigenschaften

14. Arbeitsspeicher

14.1 Speicherzuteilung

- Verwaltung des Arbeitsspeichers: Fragmente = fest abgesteckte oder ausdehn-/zusammenziehbare Gebiete für jedes Programm
  - statisch: allen Programmen inkl. des Betriebssystems sind Arbeitsspeichergebiete maximaler, fester Größe zugewiesen
    - innerhalb der Gebiete Speicher dynamisch zuteilbar
    - Gefahr von Leistungsbegrenzung/-verlust, z. B.
      - Brache eines Gebiets in anderen Gebieten nicht nutzbar
      - kleine E/A-Bandbreite mangels Puffer im Gebiet des Betriebssystems (???)
      - erhöhte Wartezeit von Prozessen wegen zu kleiner Puffer (???)
  - dynamisch: Betriebssystem ermittelt freie Arbeitsspeicherfragmente angeforderter Größe, teilt sie den Programmen zu
    → Zusammenspiel Laufzeit- und Betriebssystem (→ malloc(3), break(2))

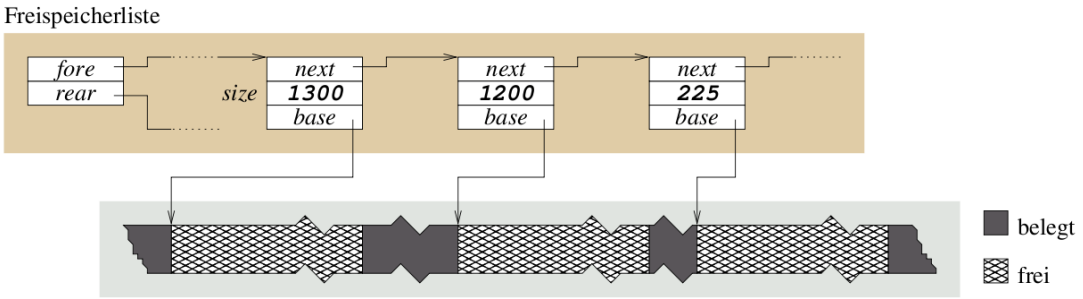
  ☞ Zuteilungseinheiten können in beiden Fällen gleich ausgelegt sein
- Zuteilungseinheiten und Verschnitt
  - Aufbau, Struktur der jeweils zugeteilten Arbeitsspeicherfragmente abhängig von Adressraumausprägung
    - Seitennummerierung: Fragmen ↦ Vielfaches von Seitenrahmen
      → ggf. mehr Speicher als benötigt zugeteil ⇒ interne Fragmentierung des Seitenrahmens
    - Segmentierung: Fragment ↦ Vielfaches von Bytes (Segment)
      → ggf. passendes Stück nicht verfügbar ⇒ externe Fragmentierung des Arbeitsspeichers
  - Optimierung des Verschnitts (als Folge in-/externer Fragmentierung) als eine der zentralen Aufgaben der Speicherverwaltung
    - anfallender Rest bei Speicherzuteilung allgemein → „Abfall“ bei interner, „Hohlräume“ bei externer Fragmentierung
- Politiken bei Speicherverwaltung („wohin, wann, welches Opfer …)
  - Platzierungsstrategie (*placement policy*): obligatorisch
    → wohin Fragment ablegen?
    - wo Verschnitt minimal/maximal
    - egal, da Verschnitt zweitrangig
  - Ladestrategie (*fetch policy*): optional: dynamisches Binden, virtuelle Maschine
    → wann Fragment laden?
    - auf Anforderung
    - im Voraus
  - Ersetzungsstrategie (*replacement strategy*): optional: virtuelle Maschine
    → welches Fragment ggf. verdrängen?

- ältestes, am seltensten genutztes
- am längsten genutztes

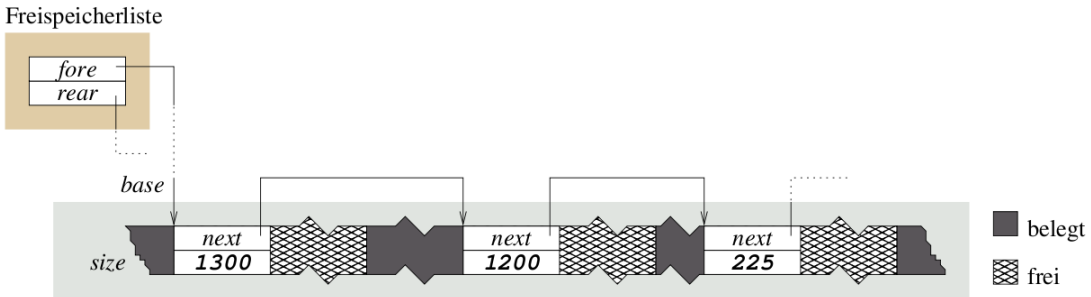
14.2 Platzierungsstrategie

- Freispeicherorganisation: Verwaltung der „Hohlräume“ im Arbeitsspeicher
  - Bitkarte (*bit map*) von Fragmenten fester Größe
    - Eignung für seitennummerierte Adressräume
    - grobkörnige Vergabe auf Seitenrahmenbasis
    - alle freien Fragmente gleich gut
  - verkettete Liste (*free list*) von Fragmenten variabler Größe
    - typisch für segmentierte Adressräume
    - feinkörnige Vergabe auf Segmentbasis
    - nicht alle freien Fragmente gleich gut

  Freispeicher als „Hohlräume“/„Löcher“ im RAM, die mit Programmtext/-daten auffüllbar sind
  → als Bitkarte/verkettete Liste implementierte Löcherliste (*hole list*)
- Freispeicher(bit)karte: Erfassung freien Speichers fester Größe
  - Zuordnung von (≥ 1) Zuständen (Bits) zu Arbeitsspeicherfragmenten: 0 ↦ belegt, 1 ↦ frei
    ⇒ Suche, Belegung, Freigabe als Operationen zur Bitverarbeitung
  - Speicherbedarf für Erfassung freier Fragmente ~ Feinkörnigkeit der Speicherzuteilung
- Freispeicherliste: Erfassung freien Speichers variabler Größe
  - Löcherliste führt Buch über freie Speicherbereiche
    - Listenelement ↝ Anfangsadresse, Länge eines Bereichs (⇒ 1 freies Fragment)
    - zwei grundsätzliche Repräsentationen
      1. Liste und Löcher voneinander getrennt
         - Listenelemente als Löcherdeskriptoren, belegen Betriebsmittel
         - Löcher mit beliebiger Größe
         - Implementierung



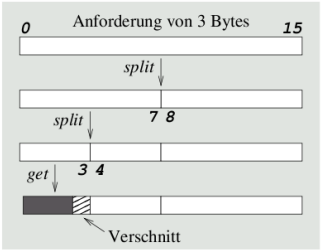
         - Liste, Listenkopf im Betriebssystemadressraum
           - *fore*, *rear*, *next*: logische/virtuelle Adressen
           - *size*: Lochgröße, Vielfaches von *sizeof*(*unit*)
           - *base*: physikalische Adresse des freien Fragments
         - Listenmanipulationen innerhalb eines logischen/virtuellen Adressraums
      2. Liste und Löcher miteinander vereint
         - Listenelemente ≙ Löcher, keine Betriebsmittelbelegung
         - Mindestgröße für Löcher: *sizeof*(*list element*)
         - Implementierung



        - nur Listenkopf im Betriebssystemadressraum
            - *fore*, *rear*, *next*, *base*: physikalische Adressen
            - *size*: Lochgröße, Vielfaches von *sizeof*(*unit*)
        - ggf. Adressraumüberschreitung durch Listenmanipulationen
            - Listenoperationen im Betriebssystemadressraum
            - Betriebssystemadressraum ≙ logischer/virtueller Adressraum
            - Liste im physikalischen Adressraum ↝ Adressraumumschaltung
    - Art der Listenverwaltung durch strategische Überlegungen bestimmt
- Zuteilungsverfahren: Löcher der Größe nach sortieren
    - *best-fit* verwaltet Löcher nach aufsteigenden Größen
        - Minimierung des Verschnitts → Suche des kleinsten passenden Lochs
        - Erzeugung kleiner Löcher am Anfang, Erhalt großer Löcher am Ende
          → Hinterlassen kleiner Löcher bei steigendem Suchaufwand
    - *worst-fit* verwaltet Löcher nach absteigenden Größen
        - Minimierung des Suchaufwands: Loch zu klein ⇒ alle anderen auch zu klein
        - Zerstörung kleiner Löcher am Anfang, Erzeugung kleiner Löcher am Ende
          → Hinterlassen großer Löcher bei konstantem Suchaufwand

    ggf. mehrmaliger Durchlauf durch Freispeicherliste: angeforderte Größe < gefundenes Loch ⇒ Verschnitt ≙ verbleibendes Loch ⇒ erneute Einsortierung
    - buddy verwaltet Löcher nach aufsteigender Größe
        - kleinstes passendes Loch $buddy_i$ der Größe 2^i suchen
          (i: Index in Tabelle von Adressen auf Löcher der Größe 2^i)
        - $buddy_i$ entsteht durch sukzessive Splittung von $buddy_{j,\, j>i}$:
            - $2^n = 2 \times 2^{n-1}$
            - zwei gleichgroße Blöcke, die *Buddy* des jeweils anderen sind
        - ggf. anfallender Verschnitt kann beträchtlich sein
          → $2^i - 1$ bei $2^i + 1$ angeforderten Einheiten
        - Kompromiss zwischen *best-fit*, *worst-fit*
            - vergleichsweise geringer Such- und Aufsplittungsaufwand
            - gut zum Laufzeitsystem passend, das in 2er-Potenzen anfordert

        ☞ jeder Rest als Summe freier *Buddies* darstellbar, wie auch jede Dezimalzahl als Summe von 2er-Potenzen



1. Block 2^4 teilen: $3 < 2^4/2$
2. Block 2^3 teilen: $3 < 2^3/2$
3. Block 2^2 vergeben: $3 \geq 2^2/2$
    - Verschnitt von $2^2 - 3 = 1$ Byte

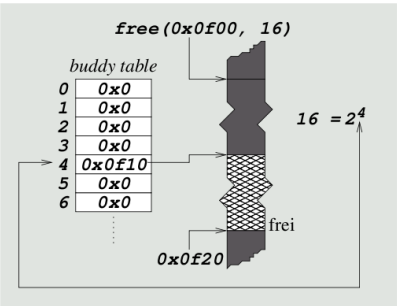
Ob der Verschnitt als interne Fragmentierung zu verbuchen ist, hängt von der MMU ab.

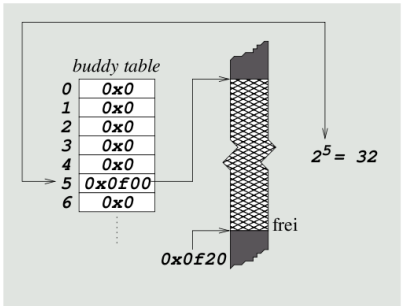
        - Verschmelzung einfach
            - Verschmelzung zweier Blöcke mögliche ⇔ Blöcke sind Buddies
            - zwei Blöcke der Größe 2^i sind Buddies ⇔ Adressunterschied nur in Bitposition i
    - *first-fit* verwaltet Löcher nach aufsteigender Adresse
        - Minimierung des Verwaltungsaufwands
          Sortierkriterium: invariante Adressen ⇒ keine Umsortierung bei anfallendem Rest
        - Erzeugung kleiner Löcher vorne, Erhalt großer Löcher am Ende
          → Hinterlassen kleiner Löcher bei steigendem Suchaufwand
    - *next-fit*: Reihum-Variante (*round-robin*) von *first-fit*
        - Minimierung des Suchaufwands: Suchbeginn bei zuletzt zugeteiltem Loch
        - Annäherung an Verteilung „gleichgroßer Löcher“ ⇒ Abnahme des Suchaufwands

    first-fit, *next-fit*: wenn angeforderte Größe < gefundenes Loch: Verschnitt, jedoch kein Einsortieren als Restloch nötig ⇒ Freispeicherliste nur einmal zu durchlaufen
- Verschmelzung
    - Vereinigung eines Lochs mit angrenzenden Löchern
        - beschleunigte Speicherzuteilung, Verringerung der externen Fragmentierung

- erfolgt bei Speicherfreigabe, scheiternder Speichervergabe
- 4 Situationen bei Löchervereinigung in Abhängigkeit von relativer Lage eines Lochs im Arbeitsspeicher
  1. zwischen zwei zugeteilten Bereichen ⇒ keine Vereinigung möglich
  2. direkt nach einem Loch ⇒ Vereinigung mit Vorgänger
  3. direkt vor einem Loch ⇒ Vereinigung mit Nachfolger
  4. zwischen zwei Löchern ⇒ Kombination von 2., 3.

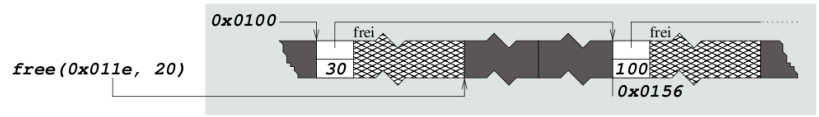
  ☞ z. T. starke Variation des Aufwands in Abhängigkeit von Zuteilungsverfahren
- Verschmelzungsaufwand i. Abh. v. Zuteilungsverfahren
  - *buddy* – klein: anhand eines Bits der Adresse des zu verschmelzenden Lochs lässt sich leicht feststellen, ob sein *Buddy* bereits als Loch in der Tabelle verzeichnet ist
  - *first/next-fit* – mittel: beim Durchlaufen der Freispeicherliste (bei Freigabe): Überprüfung jedes Eintrags hinsichtlich …
    - Adresse + Größe eines Eintrags = Adresse des zu verschmelzenden Lochs ↝ 2.
    - Adresse + Größe eines zu verschmelzenden Lochs = Adresse eines Eintrags ↝ 3.
  - *best/worst-fit* – groß: ähnlich *first/next-fit*, jedoch: Vorgänger-/Nachfolgerelement in Liste ist nicht unbedingt angrenzendes Loch → Weitersuchen
- Verschmelzung – *buddy*: zwei Blöcke 2^i sind *buddies* ⇔ Adressen unterscheiden sich in Bitposition i



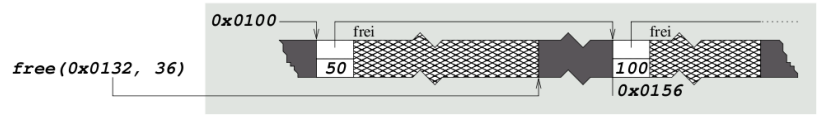


$0f00_{16} = 0000\ 1111\ 0000\ 0000_2$ $0f00_{16} = 0000\ 1111\ 0000\ 0000_2$
$0f10_{16} = 0000\ 1111\ 0001\ 0000_2$ $0f20_{16} = 0000\ 1111\ 0010\ 0000_2$
$16_{10} = 0000\ 0000\ 0001\ 0000_2$ $32_{10} = 0000\ 0000\ 0010\ 0000_2$

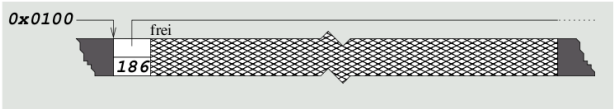
- Verschmelzung – *first/next-fit*: freizugebender Block = Nachfolger und/oder Vorgänger



► das alte 30 Byte große Loch kann um 20 Bytes vergößert werden



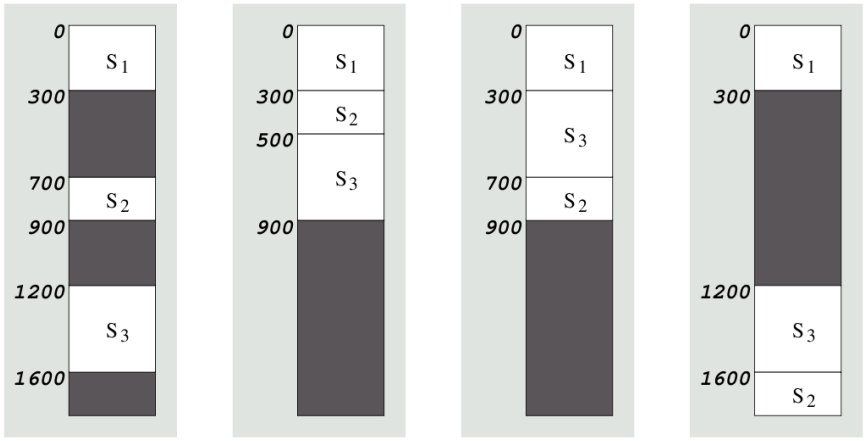
1. das alte 50 Byte große Loch kann um 36 Bytes vergößert werden
2. das neue 86 Byte große Loch kann um 100 Bytes vergößert werden



- Fragmentierung = Bruchstückbildung → Zerstückelung des Speichers in immer kleinere, verstreut vorliegende Bruchstücke
  - intern bei seitennummerierten Adressräumen ↝ Verschwendung
    - Speichervergabe in Einheiten gleicher, fester Größe
    - „lokaler Verschnitt“ nutzbar, dürfte aber nicht sein
  - extern bei segmentierten Adressräumen ↝ Verlust
    - Speicher in Einheiten variabler Größe vergeben (linear zusammenhängende Bytefolge passender Länge → Entstehen vieler kleiner Bruchstücke im Betrieb)
    - „globaler Verschnitt“ ggf. nicht mehr zuteilbar → ggf. Abhilfe durch Kompaktifizierung

- Kompaktifizierung: Auflösung externer Fragmentierung durch Vereinigung des globalen Verschnitts
  - Verschieben von Segmenten (Bytes oder Seitenrahmen), sodass am Ende großes Loch vorhanden ist
    - sukzessives Verschmelzen aller in Freispeicherliste erfassten Löcher → 1 Loch bleibt übrig
    - „Erleichterung“ des Kopiervorgangs durch Umlagerung (*swapping*) kompletter Segmente bzw. Adressräume
  - Relokation der verschobenen Segmente/Seiten(rahmen) erforderlich → Möglichkeiten:
    - Betriebssystem implementiert logische/virtuelle Adressräume
    - Übersetzer generiert positionsunabhängigen Programmtext

  → komplexes Optimierungsproblem, je nach Fragmentierungsgrad
  - Beispiel: Loslegen und Aufwand riskieren oder vorher nachdenken und Aufwand einsparen …



14.3 Ladestrategie

14.4 Ersetzungsstrategie

14.5 Zusammenfassung

15. Dateisysteme

16. Zusammenfassung