Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – IX.1 Prozessverwaltung: Einplanungsgrundlagen

Wolfgang Schröder-Preikschat

26. Oktober 2021

Agenda

Einführung

Programmfaden
- Grundsätzliches
- Fadenverläufe
- Leistungsoptimierung

Arbeitsweisen
- Ebenen
- Ebenenübergänge
- Verdrängung

Gütemerkmale
- Benutzerdienlichkeit
- Systemperformanz
- Betriebsart

Zusammenfassung

Gliederung

Lehrstoff

- **Programmfäden** als ein gängiges Mittel zum Zweck der Einplanung von Prozessen kennenlernen
  - Lauf- und Wartephasen von Fäden im Zusammenhang behandeln
  - phasenverschränkte Fadenausführung zur Leistungssteigerung nutzen
  - Uneindeutigkeit logischer Prozesszustände als Normalität sehen
- grundsätzliche **Arbeitsweisen** der Prozessplaner (*process scheduler*) verstehen und zu differenzieren
  - lang-, mittel- und kurzfristige Prozesseinplanung betrachten
  - Facetten der verdrängenden Prozesseinplanung beleuchten
  - Latenz und Determinismus in Zusammenhang bringen
- typische **Kriterien** zur und charakteristische (Güte-) **Merkmale** der Einplanung von Prozessen ansprechen
  - benutzer- und systemorientierte Kriterien unterscheiden
  - den Zusammenhang mit bestimmten Rechnerbetriebsarten erkennen

Gliederung

Prozessorzuteilungseinheit

- Einplanungseinheit für die Prozessorvergabe ist der **Faden**[1] (*thread*)
  - geplant wird, wann ein Faden wie lange ablaufen darf
- die Ablaufplanung der Fäden geschieht **betriebsmittelorientiert**, sie geschieht **ereignisbedingt** oder ist **zeitgesteuert** ausgelegt
  - die Laufbereitschaft eines Fadens hängt von der Verfügbarkeit all jener Betriebsmittel ab, die für seinen Ablauf erforderlich sind
  - die Bereitstellung von Betriebsmitteln (ggf. durch andere Fäden) kann die sofortige Einplanung von Fäden bewirken oder
  - die Einplanung erfolgt in fest vorgegebenen Zeitintervallen
- dabei handelt es sich um Vorgänge im System, die voneinander ent- oder miteinander gekoppelt sein können
  - d.h., zeitversetzt oder zeitgleich zum Ablauf eingeplanter Fäden
- **Einplanung** eines Fadens ist nicht gleichzusetzen mit **Einlastung**:
  - Einplanung ist der Vorgang der Reihenfolgenbildung von Aufträgen
  - Einlastung ist der Moment der Zuteilung von Betriebsmitteln

[1] Verbreitete Variante einer Prozessinkarnation [1, S. 5].

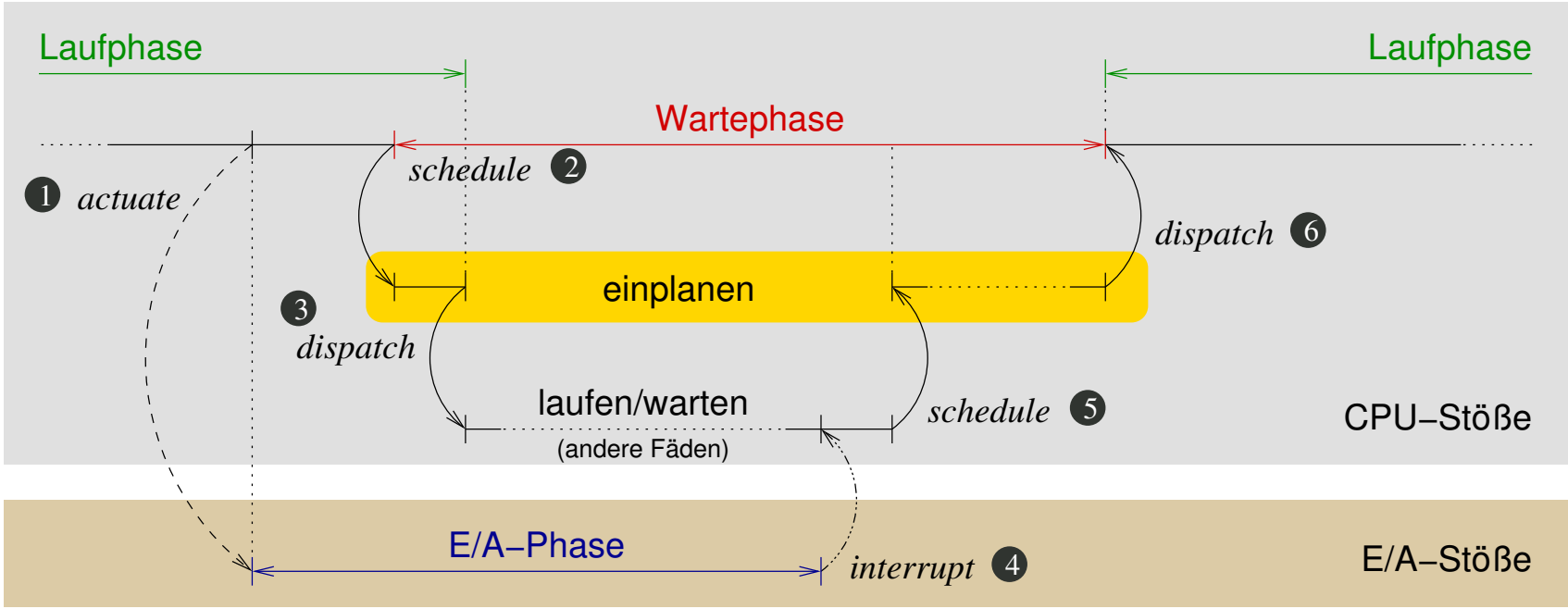
Stoßbetrieb

burst mode

- einerseits die als **Rechenstoß** (*CPU burst*) bezeichnete Laufphase
  - aktive Phase eines Fadens (auch: Rechenphase)
    - alle zur Ausführung erforderlichen Betriebsmittel sind verfügbar
  - der Faden ist eingelastet, ihm wurde der Prozessor zugeteilt
- andererseits der **Ein-/Ausgabestoß** (*I/O burst*) als evtl. Wartephase
  - ggf. die inaktive Phase eines Fadens (auch: E/A-Phase)
    - nicht alle zur Ausführung erforderlichen Betriebsmittel sind verfügbar
  - Ein-/Ausgabe abwarten meint, **Betriebsmittel** [1, S. 9–10] zu erwarten
    - konsumierbare Betriebsmittel: Eingabedaten, Ausgabequittung (Signal)
    - wiederverwendbare Betriebsmittel: Puffer, Geräte, . . . , Prozessor
  - die Betriebsmittel werden letztlich durch andere Fäden bereitgestellt
    - ein E/A-Gerät kann dabei als „externer Faden“ betrachtet werden[2]
- kontinuierlich einander abwechselnde Phasen eines jeden Prozesses
  - je nach Programm oder Progammmix, mit unterschiedlichen Stoßlängen

[2] Ein „externer Prozess“, der (a) durch Anweisung eines „internen Prozesses“ entsteht oder (b) entsprechend physikalischer Vorgaben autonom voranschreitet.

Lauf-, E/A- und Wartephasen von Fäden (1)



- in Praxis ist der **Prozesszustand** [1, S. 18] „phasenuneindeutig“
  - laufend
    - Laufphase eines Fadens, direkt vor (logisch) oder nach (real) dem Prozesswechsel bei der Einlastung (3 bzw. 6)
  - blockiert
    - noch zur Laufphase des Fadens, vor der Prozessorabgabe (2–3)
    - physische Wartephase des Fadens bis zur Bereitstellung (3–5)
  - bereit
    - noch zur Laufphase, vor der Prozessorabgabe (2–3) und bei entsprechend kurzer E/A-Phase (1–5)
    - physische Wartephase des Fadens bis zur Einlastung (3–6)

Lauf-, E/A- und Wartephasen von Fäden (2)

- **Betriebssystemkontrollfluss** zur Fadeneinplanung und -einlastung
  1. der laufende Faden stößt einen E/A-Vorgang an (*actuate*)
  2. er wartet passiv auf die Beendigung der Ein-/Ausgabe (*schedule*)
     - Anforderung eines wiederverwend-/konsumierbaren Betriebsmittels
  3. und lastet einen eingeplanten, laufbereiten Faden ein (*dispatch*)
  4. die Beendigung der Ein-/Ausgabe wird signalisiert (*interrupt*)
     - Bereitsstellung des konsumierbaren Betriebsmittels „Signal“
  5. der auf dieses Ereignis wartende Faden wird eingeplant (*schedule*)
  6. der Faden wird eingelastet, sobald er an der Reihe ist (*dispatch*)

- Kreislauf bis zum **Leerlauf** (*idle state*) mangels laufbereiter Fäden

Hinweis (Energiesparmodus)

Normalerweise wird der Prozessor dann in Bereitschaft (standby) versetzt. Untypisch ist aktives Warten auf die Bereitstellung eines Fadens durch (a) den einzigen „blockiert laufenden“ Prozess oder (b) einen sonst untätigen Prozess (idle process). Im Fall von (a) kann der so wartende Prozess sich selbst bereitstellen — oder einlasten, wenn dies die Einplanungsstrategie (full preemption) zulässt. Unabhängig davon, muss ein „blockierend laufender“ Prozess sich selbst „laufend bereit“ stellen können.

Fäden als Mittel zum Kaschieren von Totzeiten

- die **Überlappung** von Lauf- und Wartephasen verschiedener Fäden lässt eine Erhöhung der Rechnerauslastung erwarten
  - die Wartephase eines Fadens als Laufphase anderer Fäden nutzen
  - die Stöße anderer Fäden zum „Auffüllen“ von Wartephasen nutzen
- nicht nur die **Auslastung** der CPU kann sich steigern, sondern auch die der angeschlossenen Peripherie (E/A-Geräte)
  - eine CPU kann zu einem Zeitpunkt nur einen Rechenstoß verarbeiten
  - parallel dazu können jedoch mehrere Ein-/Ausgabestöße laufen
    - ausgelöst während eines Rechenstoßes: in der Laufphase eines Fadens wurden mehrere E/A-Vorgänge gestartet
    - ausgelöst von mehreren Rechenstößen: die Wartephase eines Fadens wurde mit Laufphasen anderer Fäden gefüllt
  - Folge: CPU und E/A-Geräte sind andauernd mit Arbeit beschäftigt
- beachte: Fäden sind **Ausdrucksmittel** von (a) Mehrprogrammbetrieb oder (b) nicht-sequentiellen Programmen
  - bei weniger Prozessoren als Fäden, sind Fäden zu serialisieren

Zwangsserialisierung von Programmfäden

In Bezug auf ein Exemplar des wiederverwendbaren Betriebsmittels „CPU“ beziehungsweise „Core“ (d.h., Rechenkern).

- die **absolute Ausführungsdauer** nach Ankunftszeit eingeordneter laufbereiter Fäden verlängert sich:
  - Ausgangspunkt seien n Fäden mit gleichlanger Bearbeitungsdauer k
  - der erste Faden wird um die Zeitdauer 0 verzögert
  - der zweite Faden um die Zeitdauer k, der i-te Faden um $(i-1)\cdot k$
  - der letzte von n Fäden wird verzögert um $(n-1)\cdot k$

Mittlere Fadenverzögerung

$$\frac{1}{n}\cdot\sum_{i=1}^{n}(i-1)\cdot k = \frac{n-1}{2}\cdot k$$

- die Vergrößerung der **mittleren Verzögerung** ist proportional zur Fadenanzahl

Subjektive Empfindung der Fadenverzögerung

Nur bis zu einer bestimmten Last, die sich unter anderem durch die Anzahl eingeplanter Fäden bestimmt.

- Startzeiten von Fäden verzögern sich im Mittel um: $\frac{n-1}{2} \cdot t_{cpu}$, mit t_{cpu} gleich der mittleren Dauer eines Rechenstoßes
  - sofern $t_{cpu} \geq t_{ea}$, der mittleren Dauer eines Ein-/Ausgabestoßes
  - die Praxis liefert als Regelfall jedoch ein anderes Bild: $t_{cpu} \ll t_{ea}$
- Wartephasen bei E/A-Operationen dominieren die Fadenverzögerung
  - zwischen Rechen- und Ein-/Ausgabestößen besteht eine Zeitdiskrepanz
  - der proportionale Verzögerungsfaktor bleibt weitestgehend verborgen
  - er greift erst ab einer bestimmten Anzahl von Programmfäden
    - nämlich wenn zu einem Zeitpunkt gilt: $\sum_{i=1}^{m} t_{cpu}^{i} > \sum_{j=1}^{n} t_{ea}^{j}$
  - sehr häufig ist die Fadenverzögerung daher nicht wahrnehmbar
- beachte: **Überlast** durch zuviel eingeplante Fäden ist zu **vermeiden**
  - akkumulierte Länge der Rechenstöße der jew. E/A-Auslastung anpassen

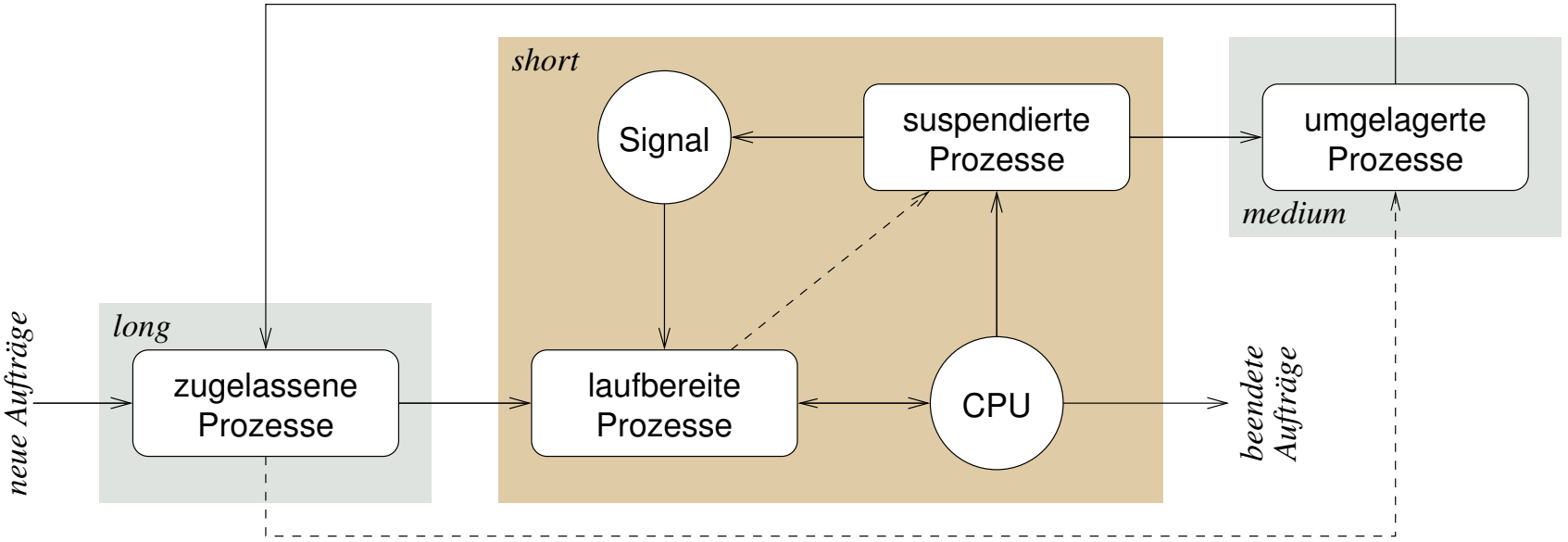
Gliederung

Dauerhaftigkeit von Zuteilungsentscheidungen

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

Logische Ebenen der Prozesseinplanung, die, mit Ausnahme der untersten Ebene, der kurzfristigen Planung, nicht in jedem Betriebssystem ein physisches Äquivalent haben.

Phasen der Prozesseinplanung



- kurzfristige Planung dient der Mitbenutzung (*sharing*) der CPU
  - laufbereite Prozesse erwarten die Zuteilung des wiederverwendbaren Betriebsmittels „CPU“, d.h. den Start ihrer Laufphase
  - suspendierte Prozesse erwarten die Zuteilung eines konsumierbaren Betriebsmittels „Signal“, d.h. das Ende ihrer Wartephase
- lang- und mittelfristige Planung regeln den Mehrprogrammbetrieb
  - Umlagerung macht Hauptspeicher frei für weitere Prozessexemplare
  - im Ergebnis könnten neue Prozesse (z.B. *login*) zugelassen werden

Prozesszustand vs. Einplanungsebene

- Prozesse haben in Abhängigkeit von der Einplanungsebene (S. 14) zu einem Zeitpunkt einen **logischen Zustand**:
  - kurzfristig (*short-term*)
    - bereit, laufend, blockiert
  - mittelfristig (*medium-term*, *mid-term*)
    - schwebend bereit, schwebend blockiert
  - langfristig (*long-term*)
    - erzeugt, gestoppt, beendet

  *Jedoch legen die **Anwendungsfälle** fest, welche dieser Ebenen von einem Betriebssystem wirklich zur Verfügung zu stellen sind, nicht umgekehrt.*

- ein **Universalbetriebssystem** implementiert eher alle Ebenen, ein **Spezialbetriebssystem** dagegen eher nur die unterste Ebene
  - kurzfristige Prozesseinplanung, obligatorisch bei Mehrprozessbetrieb

Kurzfristige Planung

- das Betriebssystem bietet **Mehrprozessbetrieb** (*multi-processing*) auf Basis der Serialisierung von Programmfäden:
  - bereit (*ready*) zur Ausführung durch den Prozessor (die CPU)
    - der Prozess ist auf der Bereitliste (*ready list*)
    - das Einplanungsverfahren bestimmt die Listenposition
  - laufend (*running*), erfolgte Zuteilung des Betriebsmittels „CPU“
    - der Prozess vollzieht seinen CPU-Stoß
    - zu einem Zeitpunkt pro CPU nur ein laufender Prozess
  - blockiert (*blocked*) auf ein bestimmtes Ereignis
    - der Prozess erwartet die Zuteilung eines Betriebsmittels
      - zusätzlich zum Betriebsmittel „CPU“
    - die Zuteilung löst ein anderer (ggf. externer) Prozess aus
- gleichzeitige Vorgänge innerhalb eines Betriebssystems bewirken ggf. **uneindeutige logische Prozesszutände** (S. 8)
  - ein Prozess, der blockieren wird, ist zeitweilig „laufend blockiert“
  - ein solcher Prozess kann dann auch „laufend bereit“ gestellt werden
    - insb. ein Prozess, der den Prozessor im Leerlauf betreibt $\mapsto$ *idle process*

Mittelfristige Planung

- das Betriebssystem implementiert die **Umlagerung** (*swapping*) von kompletten Programmen bzw. logischen Adressräumen:

  schwebend bereit (*ready suspend*)
  - das **Exemplar** eines Prozesses ist ausgelagert
    - verschoben in den Hintergrundspeicher
    - „*swap-out*“ ist erfolgt
    - „*swap-in*“ wird erwartet
  - die Einlastung des Prozesses ist außer Kraft
    - genauer: aller Fäden seines Programms

  schwebend blockiert (*blocked suspend*)
  - ausgelagerter ereigniserwartender Prozess
  - Ereigniseintritt $\mapsto$ „schwebend bereit“

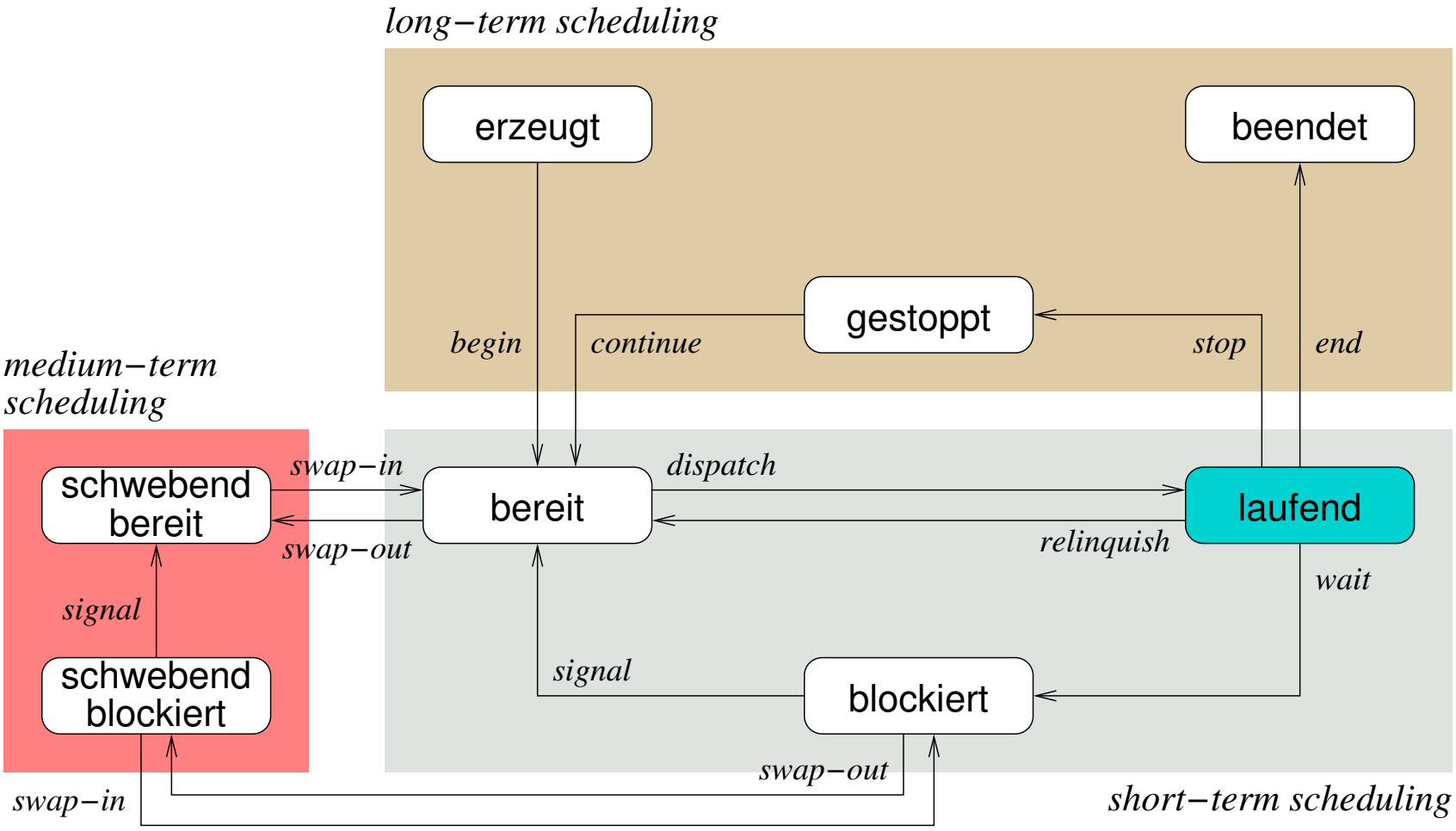
  - Prozesse im Zustand „laufend“ werden normalerweise nicht umgelagert

- im Falle langfristiger Einplanung konkurrieren „schwebend bereite“ Prozesse mit neu zuzulassenden Prozessen
  - gleichwohl genießen in dem Fall erstere (zumeist) Vorrang vor letzteren

Langfristige Planung

- das Betriebssystem verfügt über Funktionen zur **Lastkontrolle** und steuert den Grad an Mehrprogrammbetrieb:
  - erzeugt (*created*) und fertig zur Programmverarbeitung
    - das **Prozessexemplar** eines Programms wurde geschaffen
    - ggf. steht die Speicherzuteilung jedoch noch aus
  - gestoppt (*stopped*) und erwartet seine Fortsetzung/Beendigung
    - der Prozess wurde angehalten (z.B. `^Z` bzw. `kill(2)`)
    - Gründe: Überlast, Verklemmungsvermeidung, ...
  - beendet (*ended*) und erwartet seine Entsorgung
    - der Prozess ist terminiert, Betriebsmittelfreigabe erfolgt
    - ggf. muss ein anderer Prozess den „Kehraus“ vollenden
- „gestoppt“ werden können auch bereite, laufende, blockierte Prozesse
  - durch entsprechende Aktionen anderer Prozesse, über Systemaufrufe
  - sofern das Betriebssystem die dazu notwendigen Mechanismen bietet

Abfertigungszustände im Zusammenhang



- je nach Betriebssystem/-art gibt es weitere „Zwischenzustände“

Einplanungs-/Auswahlzeitpunkt

- **Einplanung** (*scheduling*) bzw. **Umplanung** (*rescheduling*):
  - nachdem ein Prozess erzeugt worden ist: *begin*
  - wenn ein Prozess freiwillig die CPU abgibt: *relinquish*
  - falls das von einem Prozess erwartete Ereignis eingetreten ist: *signal*
  - sobald ein Prozess wieder aufgenommen werden kann: *continue*
- Übergänge in den Zustand „bereit“ aktualisieren die **Bereitliste**
  - eine Entscheidung über die Einlastungsreihenfolge wird getroffen
  - eine Funktion der Einplanungsstrategie wird ausgeführt
  - Operationen auf einer „gemeinsamen“ Datenstruktur finden statt
    - je nach Betriebsart auf einer Tabelle, Warteschlange, Vorrangwarteschlange
    - ein Exemplar pro Prozessor oder für mehrere Prozessoren
- Prozesse können dazu gedrängt werden, die CPU freiwillig abzugeben
  - sofern **verdrängende** (*preemptive*) **Prozesseinplanung** erfolgt

Verdrängende Prozesseinplanung

Ereignis → *Signalisierung* → *Einplanung* → *Einlastung* → *Reaktion*

- **Verdrängung** (*preemption*) des laufenden Prozesses bedeutet:
  1. ein Ereignis tritt ein, dessen Behandlungsverlauf zum Planer führt
     - ggf. wird ein Prozess von „blockiert“ in den Zustand „bereit“ überführt[3]
  2. der (vom Ereignis unterbrochene) laufende Prozess wird eingeplant
     - d.h., vom Zustand „laufend“ in den Zustand „bereit“ überführt
  3. der einzulastende Prozess wird ausgewählt & (wieder) aufgenommen
     - ggf. handelt es sich dabei um den unter 1. eingeplanten Prozess
- Einplanung und Einlastung von Prozessen erfolgt nicht immer zeitnah zum Ereigniseintritt bzw. Moment der Verdrängungsaufforderung
  - ereignisbasierte Betriebssystem(kern)architektur
    - keine *kernel threads* (vgl. [1, S. 27]) ⇝ einen Stapel pro Betriebssystemkern
  - Unterbrechungs-/Verdrängungssperre zum Schutz kritischer Abschnitte
- ggf. entstehende Latenzzeiten können Anwendungen beeinträchtigen

[3] Nur bei Zuteilung eines konsumierbaren Betriebsmittels, nicht jedoch bei Ablauf der Zeitscheibe eines Prozesses.

Latenzzeiten und Determinismus

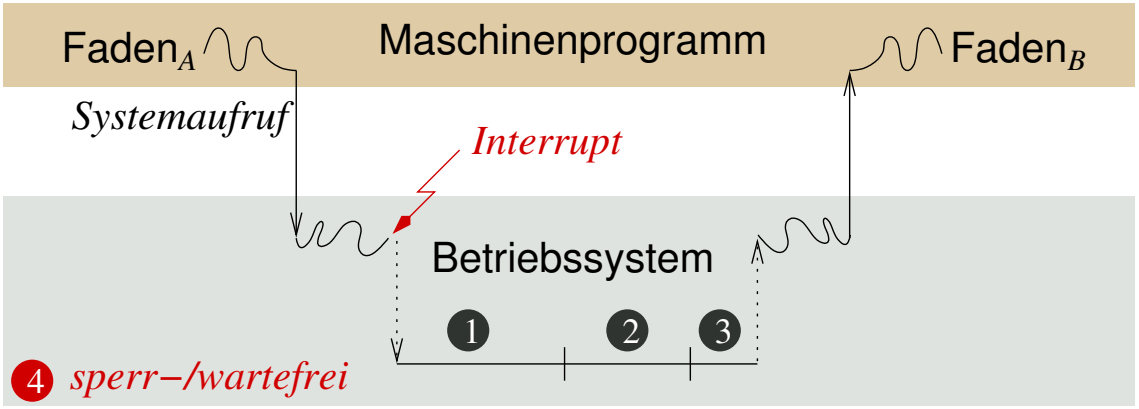
Verdrängung als Querschnittsbelang von Betriebssystemen.

- **Einplanungslatenz** (*scheduling latency*), die Zeitdauer der Ein- oder Umplanung eines Prozesses — bis zu seiner Bereitstellung
  - ist ggf. vorhersehbar (*predictable*) und **deterministisch**
    - zu jedem Zeitpunkt ist der nachfolgende Schritt eindeutig festgelegt, unabhängig von Systemlast/-aktivitäten in dem Moment
    - die Latenzzeit ist konstant oder mit fester oberer Schranke variabel
  - sollte kurz sein, um „Hintergrundrauschen“ klein zu halten
- **Einlastungslatenz** (*dispatching latency*), die Zeitspanne zwischen erfolgter Einplanung und Prozessorzuteilung eines Prozesses
  - ereignisbasierte Betriebssysteme lassen Einlastung nur an bestimmten Stellen zu („programmierte Verdrängung“) $\rightsquigarrow$ größere Latenz
    - an einem ausgewählten **Verdrängungspunkt** (*preemption point*)
  - prozessbasierte Betriebssysteme lassen Einlastung jederzeit zu, sie können voll verdrängend (*full preemptive*) arbeiten $\rightsquigarrow$ kleinere Latenz
    - sofern sie frei von Unterbrechungs- oder Verdrängungssperren sind
  - die Zeitdauer der Einlastung ist i.A. vorhersehbar und deterministisch

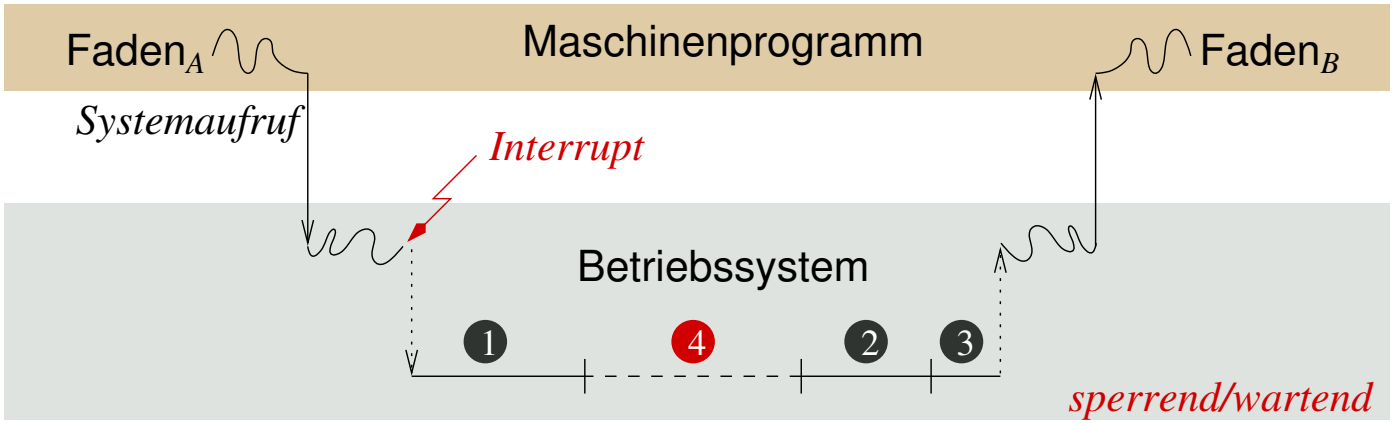
Latenzzeiten in Bezug zum Betriebsmodus

Asynchrone Programmunterbrechungen als Quelle der Ungewissheit.

voll verdrängend



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



verdrängend

Gliederung

Dimensionen der Prozesseinplanung

- Kriterien zur Aufstellung einer Einlastungsreihenfolge von Prozessen

  benutzerorientierte Kriterien

  - fokussieren auf **Benutzerdienlichkeit**
  - d.h. das vom jeweiligen Benutzer wahrgenommene Systemverhalten
  - bestimmen im großen Maße die Akzeptanz des Systems
    - bedeutsam für die Anwendungsdomäne in technischer Hinsicht
    - z.B. Einhaltung und Durchsetzung von Gütemerkmalen

  systemorientierte Kriterien

  - fokussieren auf **Systemperformanz**
  - d.h. die effektive und effiziente Auslastung der Betriebsmittel
  - bestimmen im großen Maße die „Rentabilität“ des Systems
    - bedeutsam für die Anwendungsdomäne in kommerzieller Hinsicht
    - z.B. Amortisierung hoher Anschaffungskosten von Großrechnern

- Ausschlusskriterien sind dies nicht, vielmehr **Schwerpunktsetzung**:
  - gute Systemperformanz ist auch der Benutzerdienlichkeit förderlich

Benutzerorientierte Kriterien

- charakteristische **Anforderungsmerkmale**:

  Antwortzeit Minimierung der Zeitdauer von der Auslösung eines Systemaufrufs bis zur Entgegennahme der Rückantwort, bei gleichzeitiger Maximierung der Anzahl interaktiver Prozesse.

  Durchlaufzeit Minimierung der Zeitdauer vom Starten eines Prozesses bis zu seiner Beendigung, d.h., der effektiven Prozesslaufzeit und aller anfallenden Prozesswartezeiten.

  Termineinhaltung Starten und/oder Beendigung eines Prozesses (bis) zu einem fest vorgegebenen Zeitpunkt.

  Vorhersagbarkeit Deterministische Ausführung des Prozesses unabhängig von der jeweils vorliegenden Systemlast.

- je nach **Anwendungsdomäne** mit unterschiedlicher Wichtung

Systemorientierte Kriterien

- wünschenswerte Anforderungserkmale:

Durchsatz	Maximierung der Anzahl vollendeter Prozesse pro vorgegebener Zeiteinheit, d.h., der (im System) geleisteten Arbeit.
Prozessorauslastung	Maximierung des Prozentanteils der Zeit, in der die CPU Programme ausführt, d.h., „sinnvolle“ Arbeit leistet.
Gerechtigkeit	Gleichbehandlung der Prozesse; Zusicherung, ihnen innerhalb gewisser Zeiträume die CPU zuzuteilen.
Dringlichkeiten	Vorzugbehandlung des Prozesses mit der höchsten (statischen/dynamischen) Priorität.
Lastausgleich	Gleichmäßige Betriebsmittelauslastung; ggf. auch Vorzugbehandlung der Prozesse, die stark belastete Betriebsmittel eher selten belegen.

- sollten verträglich zu der jeweiligen Anwendungsdomäne ausgelegt sein

Betriebsart vs. Einplanungskriterien

- Prozesseinplanung impliziert die Rechnerbetriebsart und umgekehrt:

  allgemein Gerechtigkeit, Lastausgleich
  - **Durchsetzung der jeweiligen Strategie**

  Stapelbetrieb Durchsatz, Durchlaufzeit, Prozessorauslastung

  Dialogbetrieb Antwortzeit

  Echtzeitbetrieb Dringlichkeit, Termineinhaltung, Vorhersagbarkeit
  - oft im Konflikt mit Gerechtigkeit/Lastausgleich

Proportionalität

Für bestimmte Prozesse ein Laufzeitverhalten „simulieren“, das nicht unbedingt dem technischen Leistungsvermögen des Rechensystems entspricht:

- es kommt gelegentlich vor, dass Nutzer eine inhärente Vorstellung über die Dauer bestimmter Aktionen des Betriebssystems haben
- aus Gründen der **Nutzerakzeptanz** sollte diesen entsprochen werden

Gliederung

Resümee

- **Einplanungseinheit** für die Prozessorvergabe ist der Faden
  - seine Lauf- und Wartephasen betreiben einen Rechner stoßartig
  - Fäden sind Mittel zum Kaschieren von Totzeiten anderer Fäden
- 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

- Durchsetzung der Kriterien impliziert eine bestimmte **Betriebsart**
  - umgekehrt: Betriebsarten erwarten die Durchsetzung gewisser Kriterien

Literaturverzeichnis

[1] Kleinöder, J. ; Schröder-Preikschat, W. :
Prozesse.
In: Lehrstuhl Informatik 4 (Hrsg.): *Systemprogrammierung*.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 6.1

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – IX.2 Prozessverwaltung: Einplanungsverfahren

Wolfgang Schröder-Preikschat

2. November 2021

Agenda

Einführung

Einordnung
- Klassifikation

Verfahrensweisen
- Kooperativ
- Verdrängend
- Probabilistisch
- Mehrstufig

Zusammenfassung

Gliederung

Lehrstoff

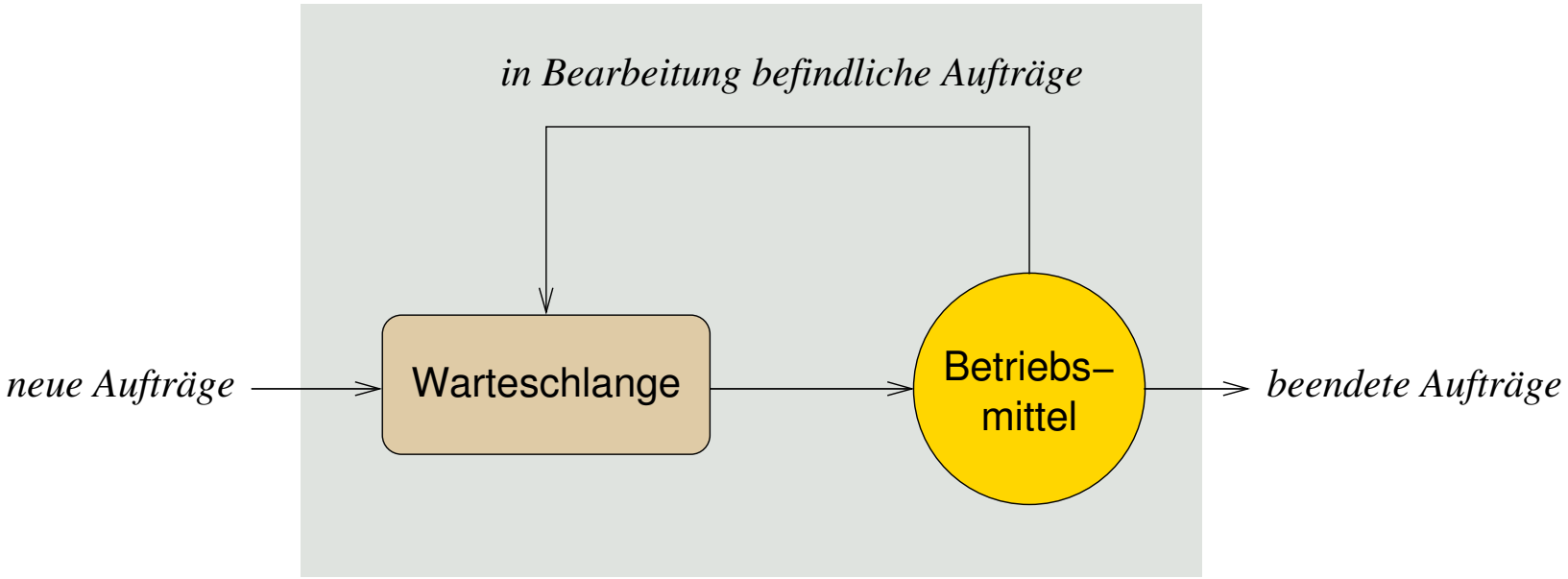
- gängige Klassen der **Ein-/Umplanung** von Prozessen kennenlernen und in ihrer Bedeutung einschätzen können
  - jedes Verfahren einer Klasse hat bestimmte **Gütemerkmale** im Fokus
  - bei mehreren Merkmalen müsste ein **Kompromiss** gefunden werden
  - scheidet Konfliktlösung aus, ist eine geeignete **Priorisierung** vorzunehmen
- die Verfahren auf **nicht-funktionale Eigenschaften** untersuchen, so Gemeinsamkeiten und Unterschiede erfassen
  - Gerechtigkeit, minimale Antwort- oder Durchlaufzeit
  - maximaler Durchsatz, maximale Auslastung
  - Termineinhaltung, Dringlichkeiten genügend, Vorhersagbarkeit
- erkennen, dass es die „eierlegende Wollmilchsau“ auch in einer virtuellen Welt nicht geben kann...

> *Kein einziges Verfahren zur Ein-/Umplanung von Prozessen hat nur Vorteile, befriedigt alle Bedürfnisse, genügt allen Ansprüchen.*

Einplanungsalgorithmen

vgl. [8, S. 40]

- Verwaltung von (betriebsmittelgebundenen) **Warteschlangen**



Ein einzelner Einplanungsalgorithmus ist charakterisiert durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse in die Warteschlange eingereiht werden. [12]

Warteschlangentheorie

vgl. [8, S. 41]

- die Charakterisierung von **Einplanungsalgorithmen** macht glauben, Betriebssysteme fokussiert „mathematisch“ studieren zu müssen
  - R. W. Conway, L. W. Maxwell, L. W. Millner. *Theory of Scheduling*.
  - E. G. Coffman, P. J. Denning. *Operating System Theory*.
  - L. Kleinrock. *Queuing Systems*, *Volume I: Theory*.
- praktische Umsetzung offenbart jedoch einen **Querschnittsbelang** (*cross-cutting concern*), der sich kaum modularisieren lässt
  - spezifische Betriebsmittelmerkmale stehen ggf. Bedürfnissen der Prozesse, die Aufträge zur Betriebsmittelnutzung abgesetzt haben, gegenüber
  - dabei ist die Prozessreihenfolge in Warteschlangen (bereit, blockiert) ein Aspekt, die Auftragsreihenfolge dagegen ein anderer Aspekt
  - **Interferenz**[1] bei der Durchsetzung der Strategien kann die Folge sein
- Einplanungsverfahren stehen und fallen mit den Vorgaben, die für die jeweilige **Zieldomäne** zu treffen sind
  - die „Eier-legende Wollmilchsau“ kann es nicht geben
  - Kompromisslösungen sind geläufig — aber nicht in allen Fällen tragfähig

[1] lat. *inter* zwischen und *ferire* von altfrz. *s'entreferir* sich gegenseitig schlagen.

Gliederung

Kooperativ vs. Präemptiv

- **Souverän** ist die Anwendung oder das Betriebssystem verhält sich Entwicklungen gegenüber zuvorkommend, vorsorglich, vorbeugend

  kooperative Planung (*cooperative scheduling*)
  - Ein-/Umplanung voneinander abhängiger Prozesse
  - Prozessen wird die CPU nicht zugunsten anderer Prozesse entzogen
  - der laufende Prozess gibt die CPU nur mittels Systemaufruf ab
    - die Systemaufrufbehandlung aktiviert (direkt/indirekt) den Scheduler
    - systemaufruffreie Endlosschleifen beeinträchtigen andere Prozesse
  - **CPU-Monopolisierung** ist möglich: *run to completion*

  präemptive Planung (*preemptive scheduling*)
  - Ein-/Umplanung voneinander unabhängiger Prozesse
  - Prozessen kann die CPU entzogen werden, zugunsten anderer Prozesse
  - der laufende Prozess wird ereignisbedingt von der CPU verdrängt
    - die Ereignisbehandlung aktiviert (direkt/indirekt) den Scheduler
    - Endlosschleifen beeinträchtigen andere Prozesse nicht (bzw. kaum)
  - Monopolisierung der CPU ist nicht möglich: **CPU-Schutz**
- **Synergie**: auf Maschinenprogrammebene kooperative *user threads*, auf Betriebssystemebene präemptive *kernel threads* [8, vgl. S. 27]. . .

Deterministisch vs. Probabilistisch

- alle Abläufe durch ***à priori* Wissen** eindeutig festlegen könnend oder die **Wahrscheinlichkeit** berücksichtigend

  deterministische Planung (*deterministic scheduling*)
  - alle Prozesse (Rechenstoßlängen)[2] und ggf. auch **Termine** sind bekannt
  - die genaue Vorhersage der CPU-Auslastung ist möglich
  - das System stellt die Einhaltung von **Zeitgarantien** sicher
  - die Zeitgarantien gelten unabhängig von der jeweiligen Systemlast

  probabilistische Planung (*probabilistic scheduling*)
  - Prozesse (exakte Rechenstoßlängen) sind unbekannt, ggf. auch Termine
  - die CPU-Auslastung kann lediglich abgeschätzt werden
  - das System kann Zeitgarantien weder geben noch einhalten
  - Zeitgarantien sind durch die Anwendung sicherzustellen

- dabei fällt die Abgrenzung nicht immer so scharf aus: wahrscheinliche Abläufe vorherzusagen, ist sehr nützlich. . .

[2] Bei (strikten) Echtzeitsystemen mindestens die Stoßlänge des „schlimmsten Falls“ (*worst-case execution time*, WCET).

Statisch vs. Dynamisch

- Abläufe entkoppelt von oder gekoppelt mit der Programmausführung bestimmen und entsprechend entwickeln

  statische Planung (*off-line scheduling*), vorlaufend
  - vor Betrieb des Prozess- oder Rechensystems
  - die Berechnungskomplexität verbietet Planung im laufenden Betrieb
    - z.B. die Berechnung, dass alle Zeitvorgaben garantiert eingehalten werden
    - unter Berücksichtigung jeder abfangbaren katastrophalen Situation
  - Ergebnis der Vorberechnung ist ein **vollständiger Ablaufplan**
    - u.a. erstellt per Quelltextanalyse spezieller „Übersetzer“
    - oft zeitgesteuert abgearbeitet als Teil der Prozesseinlastung
  - die Verfahren sind zumeist beschränkt auf **strikte Echtzeitsysteme**

  dynamische Planung (*on-line scheduling*), mitlaufend
  - während Betrieb des Prozess- oder Rechensystems
  - Stapelsysteme, interaktive Systeme, verteilte Systeme
  - schwache und feste Echtzeitsysteme

- auch hier ist die Abgrenzung nicht immer so scharf: von vorläufigen Ablaufplänen ausgehen zu können, ist sehr hilfreich. . .

Asymmetrisch vs. Symmetrisch (1)

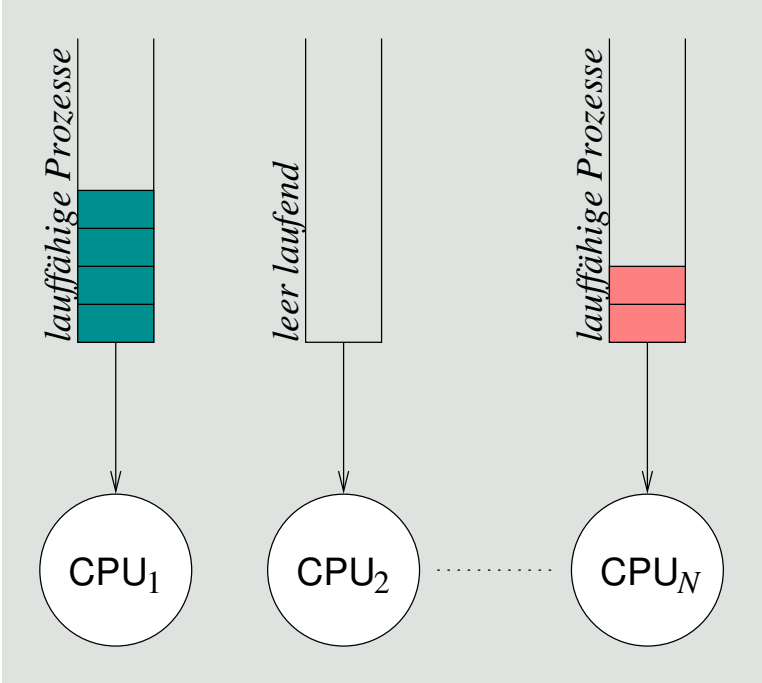
- für **mehrere Prozessoren** Abläufe nach verschiedenen Kriterien oder ihren wechselseitigen Entsprechungen festlegen

  asymmetrische Planung (*asymmetric scheduling*)
  - je nach den Prozessoreigenschaften der **Maschinenprogrammebene**
  - obligatorisch in einem asymmetrischen Multiprozessorsystem
    - Rechnerarchitektur mit programmierbare Spezialprozessoren
    - z.B. Grafik- und/oder Kommunikationsprozessoren einerseits
    - ein Feld konventioneller (gleichartiger) Prozessoren andererseits
  - optional in einem symmetrischen Multiprozessorsystem (s.u.)
    - das Betriebssystem hat freie Hand über die Prozessorvergabe
  - Prozesse in funktionaler Hinsicht ungleich verteilen (müssen)

  symmetrische Planung (*symmetric scheduling*)
  - je nach den Prozessoreigenschaften der **Befehlssatzebene**
  - identische Prozessoren, alle geeignet zur Programmausführung
  - Prozesse möglichst gleich auf die Prozessoren verteilen: Lastausgleich
- dabei kann jedem Prozessor eine eigene Bereitliste zugeordnet sein oder (Gruppen von) Prozessoren teilen sich eine Bereitliste

Asymmetrisch vs. Symmetrisch (2)

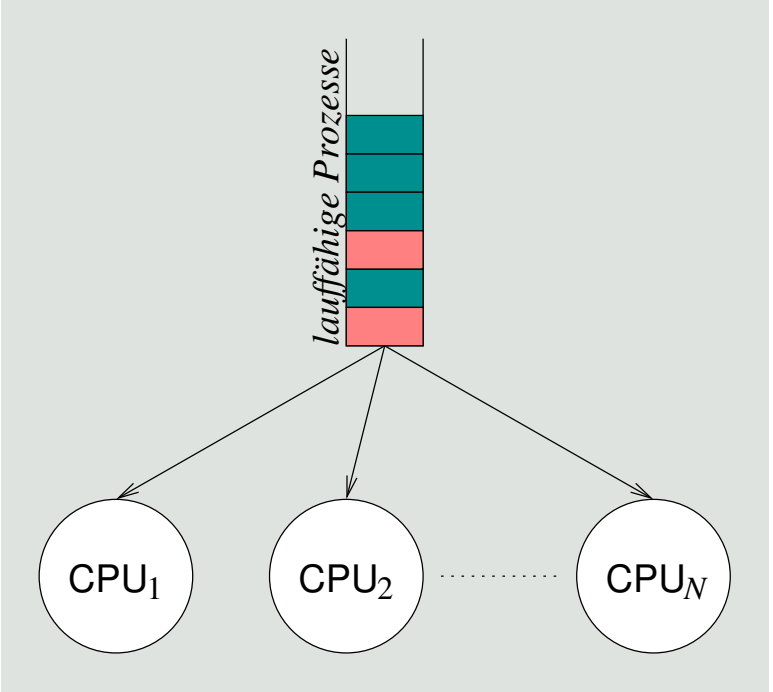
asymmetrische Prozesseinplanung



separate Bereitlisten

- lokale Bereitliste
- ggf. ungleichmäßige Auslastung
- ohne gegenseitige Beeinflussung
  - keine Multiprozessorsynchronisation

symmetrische Prozesseinplanung



gemeinsame Bereitliste

- globale Bereitliste
- ggf. gleichmäßige Auslastung
- gegenseitige Beeinflussung
  - Multiprozessorsynchronisation

Gliederung

Klassische Planungs- bzw. Auswahlverfahren

- betrachtet werden grundlegende Ansätze für **Uniprozessorsysteme**, je nach Klassifikationsmerkmal bzw. nichtfunktionaler Eigenschaft:

kooperativ	FCFS	gerecht
	▪ wer zuerst kommt, mahlt zuerst...	
verdrängend	RR, VRR	reihum
	▪ jeder gegen jeden...	
probabilistisch	SPN (SJF), SRTF, HRRN	priorisierend
	▪ die Kleinen nach vorne...	
mehrstufig	MLQ, MLFQ (FB)	
	▪ Multikulti...	

- dabei steht die Fähigkeit zur **Interaktion** mit „externen Prozessen" (insb. dem Menschen) als Gütemerkmal im Vordergrund
  - d.h., Auswirkungen auf die **Antwortzeit** von Prozessen

FCFS

first-come, first-served

Fair, einfach zu implementieren (FIFO), . . . , dennoch problematisch.

- Prozesse werden nach ihrer **Ankunftszeit** (*arrival time*) eingeplant und in der sich daraus ergebenden Reihenfolge auch verarbeitet
  - nicht-verdrängendes Verfahren, setzt kooperative Prozesse voraus
- gerechtes Verfahren auf Kosten einer im Mittel höheren Antwortzeit und niedrigerem E/A-Durchsatz
  - suboptimal bei einem Mix von kurzen und langen Rechenstößen

Prozesse mit $\left\{ \begin{array}{l} \text{langen} \\ \text{kurzen} \end{array} \right\}$ Rechenstößen werden $\left\{ \begin{array}{l} \text{begünstigt} \\ \text{benachteiligt} \end{array} \right\}$

- Problem: **Konvoieffekt**
  - kurze Prozesse bzw. Rechenstöße folgen einem langen. . .

FCFS: Konvoieffekt

- Durchlaufzeit kurzer Prozesse im Mix mit langen Prozessen:

Prozess	Zeiten					T_q/T_s
	Ankunft	T_s	Start	Ende	T_q	
A	0	1	0	1	1	1.00
B	1	100	1	101	100	1.00
C	2	1	101	102	100	100.00
D	3	100	102	202	199	1.99
Ø					100	26.00

T_s = Bedienzeit, T_q = Duchlaufzeit

normalisierte Duchlaufzeit (T_q/T_s)

- ideal für A und B, unproblematisch für D
- schlecht für C
  - sie steht in einem extrem schlechten Verhältnis zur Bedienzeit T_s
  - typischer Effekt im Falle von kurzen Prozessen, die langen folgen

RR

round robin

Verdrängendes FCFS, Zeitscheiben, CPU-Schutz.

- Prozesse werden nach ihrer Ankunftszeit ein- und in regelmäßigen Zeitabständen (periodisch) umgeplant
  - verdrängendes Verfahren, nutzt **periodische Unterbrechungen**
    - Zeitgeber (*timer*) liefert asynchrone Programmunterbrechungen
  - jeder Prozess erhält eine **Zeitscheibe** (*time slice*) zugeteilt
    - obere Schranke für die Rechenstoßlänge eines laufenden Prozesses
- Verringerung der bei FCFS auftretenden Benachteiligung von Prozessen mit kurzen Rechenstößen
  - die **Zeitscheibenlänge** bestimmt die Effektivität des Verfahrens
    - zu lang, Degenierung zu FCFS; zu kurz, sehr hoher Mehraufwand
  - Faustregel: etwas länger als die Dauer eines „typischen Rechenstoßes“
- Problem: **Konvoieffekt**
  - Prozesse kürzer als die Zeitscheibe folgen einem, der verdrängt wird...

RR: Konvoieffekt

- da die Zuteilung der Zeitscheiben an Prozesse nach FCFS geschieht, werden kurze Prozesse nach wie vor benachteiligt:

  E/A-intensive Prozesse schöpfen ihre Zeitscheibe selten voll aus
  - sie beenden ihren Rechenstoß freiwillig
    - vor Ablauf der Zeitscheibe

  CPU-intensive Prozesse schöpfen ihre Zeitscheibe meist voll aus
  - sie beenden ihren Rechenstoß unfreiwillig
    - durch Verdrängung
  - unabhängig davon werden jedoch alle Prozesse immer reihum bedient

- wird eine Zeitscheibe durch einen Prozess nicht ausgeschöpft, verteilt sich die CPU-Zeit zu Ungunsten E/A-intensiver Prozesse
  - E/A-intensive Prozesse werden schlechter bedient
  - E/A-Geräte sind schlecht ausgelastet
  - Varianz der Antwortzeit E/A-intensiver Prozesse kann beträchtlich sein
    - in Abhängigkeit vom jeweiligen Mix von Prozessen

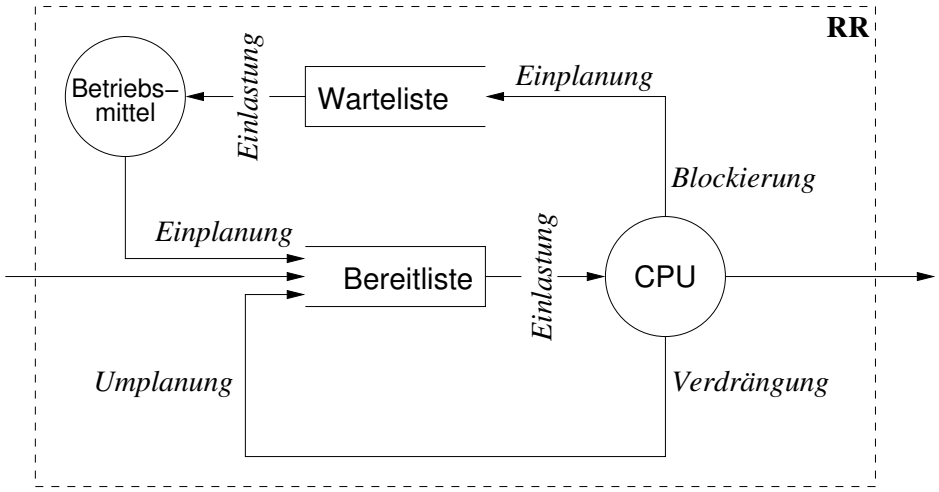
VRR

virtual round robin

RR mit Vorzugswarteschlange und variablen Zeitscheiben, um interaktive (d.h., E/A-intensive) Prozesse nicht zu benachteiligen.

- auf E/A wartende Prozesse werden mit Beendigung ihres jeweiligen Ein-/Ausgabestoßes bevorzugt eingeplant (d.h., bereit gestellt)
  - **Einplanung** mittels einer der Bereitliste vorgeschalteten **Vorzugsliste**
    - FIFO $\rightsquigarrow$ evtl. Benachteiligung hoch-interaktiver Prozesse; daher...
    - aufsteigend sortiert nach dem **Zeitscheibenrest** eines Prozesses
  - **Umplanung** bei Ablauf der aktuellen Zeitscheibe
    - die Prozesse auf der Vorzugsliste werden zuerst eingelastet
    - sie bekommen die CPU für die Restdauer ihrer Zeitscheibe zugeteilt
    - bei Ablauf dieser Zeitscheibe werden sie in die Bereitsliste eingereiht
  - erreicht durch strukturelle Maßnahmen — nicht durch analytische
- kein voll-verdrängendes Verfahren
  - die Einlastung auch des kürzesten bereitgestellten Prozesses erfolgt nicht zum Zeitpunkt seiner Bereitstellung
  - sondern frühestens nach Ablauf der aktuellen Zeitscheibe

RR vs. VRR



Bereitliste
- lauffähiger Fäden[a]
- dreiseitig bestückt
  2 × Einplanung
  1 × Umplanung
- unbedingt bedient

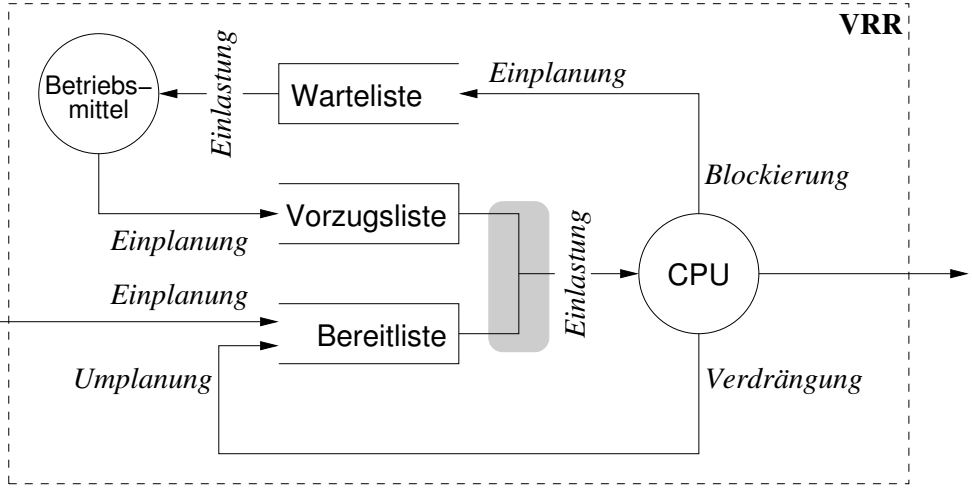
Warteliste
- blockierter Fäden

[a] CPU „Warteliste“

Bereitliste
- wie bei RR
- 2-seitig bestückt
  1 × Einplanung
  1 × Umplanung
- bedingt bedient

Warteliste
- wie bei RR

Vorzugsliste
- unbedingt bedient



SPN

shortest process next

Zeitreihen bilden, analysieren und verwerten: nicht verdrängend.

- jeder Prozess wird entsprechend der für ihn im Durchschnitt oder maximal **erwarteten Bedienzeit** eingeplant
  - Grundlage dafür ist *à priori* Wissen über die **Prozesslaufzeiten**:
    - Stapelbetrieb Programmierer setzen **Frist** (*time limit*)
    - Produktionsbetrieb Erstellung einer **Statistik** durch Probeläufe
    - Dialogbetrieb **Abschätzung** von Rechenstoßlängen zur Laufzeit
  - Abarbeitung einer aufsteigend nach Laufzeiten sortierten Bereitsliste
    - Abschätzung erfolgt vor (statisch) oder zur (dynamisch) Laufzeit
- Verkürzung von Antwortzeiten und Steigerung der Gesamtleistung des Systems bei Benachteiligung längerer Prozesse
  - ein **Verhungern** (*starvation*) dieser Prozesse ist möglich
- ohne Konvoi-Effekt — jedoch ist als praktikable Implementierung nur die **näherungsweise Lösung** möglich
  - da die Rechenstoßlängen nicht exakt im Voraus bestimmbar
  - die obere Grenze einer Stoßlänge nicht selten auch unvorhersagbar ist

SPN: Mittlung

- ein **heuristisches Verfahren**, das für jeden Prozess den Mittelwert über seine jeweiligen Rechenstoßlängen bildet
  - damit ist die erwarte Länge des nächsten Rechenstoßes eines Prozesses:

$$S_{n+1} = \frac{1}{n} \cdot \sum_{i=1}^{n} T_i$$

    - arithmetisches Mittel aller gemessenen Rechenstoßlängen des Prozesses
  - Problem dieser Berechnung ist die **gleiche Wichtung** aller Rechenstöße
  - jüngere Rechenstöße machen jedoch die **Lokalität** eines Prozesse aus
    - diesen Stößen sollte eine größere Wichtung gegeben werden (vgl. S. 23)
- die **Messung** der Dauer eines Rechenstoßes geschieht im Moment der Prozesseinlastung (d.h., der Prozessumschaltung)
  - Stoppzeit T_2 von P_j entspricht (in etwa) der Startzeit T_1 von P_{j+1}
    - gemessen in Uhrzeit (*clock time*) oder Uhrtick (*clock tick*)
  - dann ergibt $T_2 - T_1$ die gemessene Rechenstoßlänge für jeden Prozess P_i
  - der Differenzwert wird im jeweiligen Prozesskontrollblock akkumuliert

SPN: Wichtung

- mittels **Dämpfungsfiler** (*decay filter*), d.h., der Dämpfung (*decay*) der am weitesten zurückliegenden Rechenstöße:

$$\begin{aligned} S_{n+1} &= \frac{1}{n} \cdot T_n + \frac{n-1}{n} \cdot S_n \\ &= \alpha \cdot T_n + (1-\alpha) \cdot S_n \end{aligned}$$

  - mit zuletzt gemessener (T_n) und geschätzter (S_n) Rechenstoßlänge
  - für den konstanten **Wichtungsfaktor** α gilt dabei: $0 < \alpha < 1$
    - drückt die relative Wichtung einzelner Rechenstöße der Zeitreihe aus
- um die Wirkung des Wichtungsfaktors zu verdeutlichen, die teilweise Expansion der Gleichung wie folgt:
  - $S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + \ldots + (1-\alpha)^i \alpha T_{n-1} + \ldots + (1-\alpha)^n S_1$
- Beispiel der Entwicklung für $\alpha = 0.8$:
  - $S_{n+1} = 0.8\,T_n + 0.16\,T_{n-1} + 0.032\,T_{n-2} + 0.0064\,T_{n-3} + \ldots$
  - zurückliegende Rechenstöße des Prozesses verlieren schnell an Gewicht

HRRN

highest response ratio next

Hungerfreies SPN.

- Prozesse werden nach ihrer **erwarteten Bedienzeit** eingeplant und periodisch unter Berücksichtigung ihrer **Wartezeit** umgeplant
  - in regelmäßigen Zeitabständen wird ein Verhältniswert R berechnet:

$$R = \frac{w + s}{s}$$

    w aktuell abgelaufene Wartezeit eines Prozesses
    s erwartete (d.h., abgeschätzte) Bedienzeit eines Prozesses
  - ausgewählt wird der Prozess mit dem größten Verhältniswert R
- die periodische Aktualisierung betrifft alle Einträge in der Bereitliste und findet im Hintergrund des aktuellen Prozesses statt
  - ausgelöst durch einen Uhrtick (*clock tick*)
- Anmerkung: ein Anstieg der Wartezeit eines Prozesses bedeutet seine **Alterung** (*aging*)
  - der Alterung entgegenwirken beugt Verhungern (*starvation*) vor

SRTF

shortest remaining time first

Verdrängendes SPN, Hungergefahr, Effektivität von VRR.

- Prozesse werden nach ihrer **erwarteten Bedienzeit** eingeplant und in unregelmäßigen Zeitabständen **spontan** umgeplant
  - sei T_{et} die erwartete Rechenstoßlänge eines eintreffenden Prozesses
  - sei T_{rt} die verbleibende Rechenstoßlänge des laufenden Prozesses
  - der laufende Prozess wird verdrängt, wenn gilt: $T_{et} < T_{rt}$
- die **Umplanung** erfolgt ereignisbedingt und (ggf. voll) verdrängend im Moment der Ankunftszeit eines Prozesses
  - z.B. bei Beendigung des Ein-/Ausgabestoßes eines wartenden Prozesses
  - allgemein: bei Aufhebung der Wartebedingung für einen Prozess
- bei **Verdrängung** kommt der betreffende Prozess entsprechend der Restdauer seiner erwarteten Rechenstoßlänge auf die Bereitliste
  - führt allgemein zu besseren Antwort- und Durchlaufzeiten
  - gegenüber VRR steht der Aufwand zur Rechenstoßlängenabschätzung

MLQ

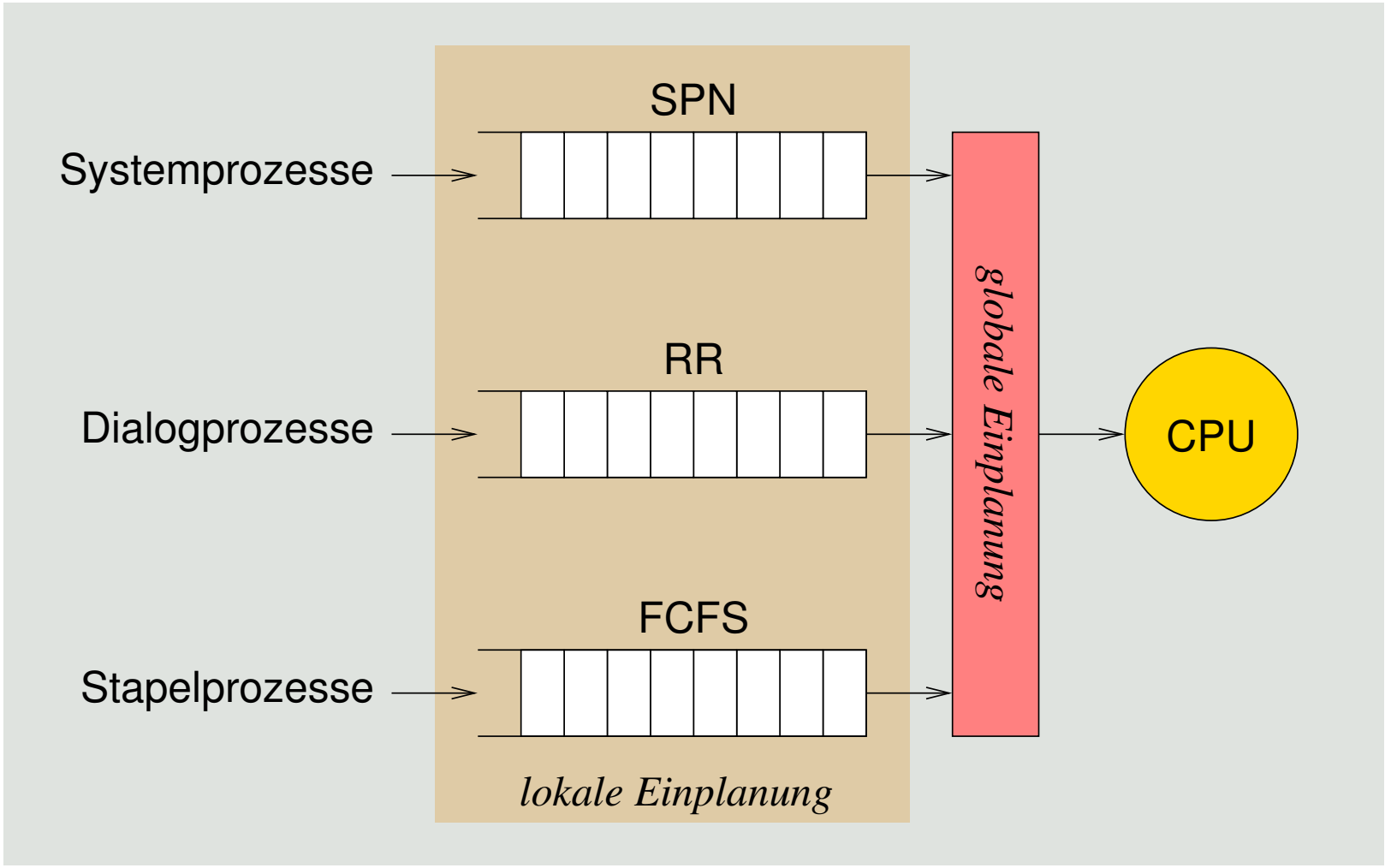
multilevel queue

Unterstützt Mischbetrieb: Vorder- und Hintergrundbetrieb.

- Prozesse werden nach ihrem **Typ** (d.h., nach den für sie zutreffend geglaubten Eigenschaften) eingeplant
  - Aufteilung der Bereitliste in separate („getypte“) Listen
    - z.B. für System-, Dialog- und Stapelprozesse
  - mit jeder Liste eine **lokale Einplanungsstrategie** verbinden
    - z.B. SPN, RR und FCFS
  - zwischen den Listen eine **globale Einplanungsstrategie** definieren
    - statisch
      - Liste einer bestimmten Prioritätsebene fest zuordnen
      - Hungergefahr für Prozesse tiefer liegender Listen
    - dynamisch
      - die Listen im Zeitmultiplexverfahren wechseln
      - z.B. 40 % System-, 40 % Dialog-, 20 % Stapelprozesse
- dem Prozess einen Typen zuordnen ist eine statische Entscheidung
  - sie wird zum Zeitpunkt der Prozesserzeugung getroffen

MLQ: Mischbetrieb

System-, Dialog- und Stapelprozesse im Mix



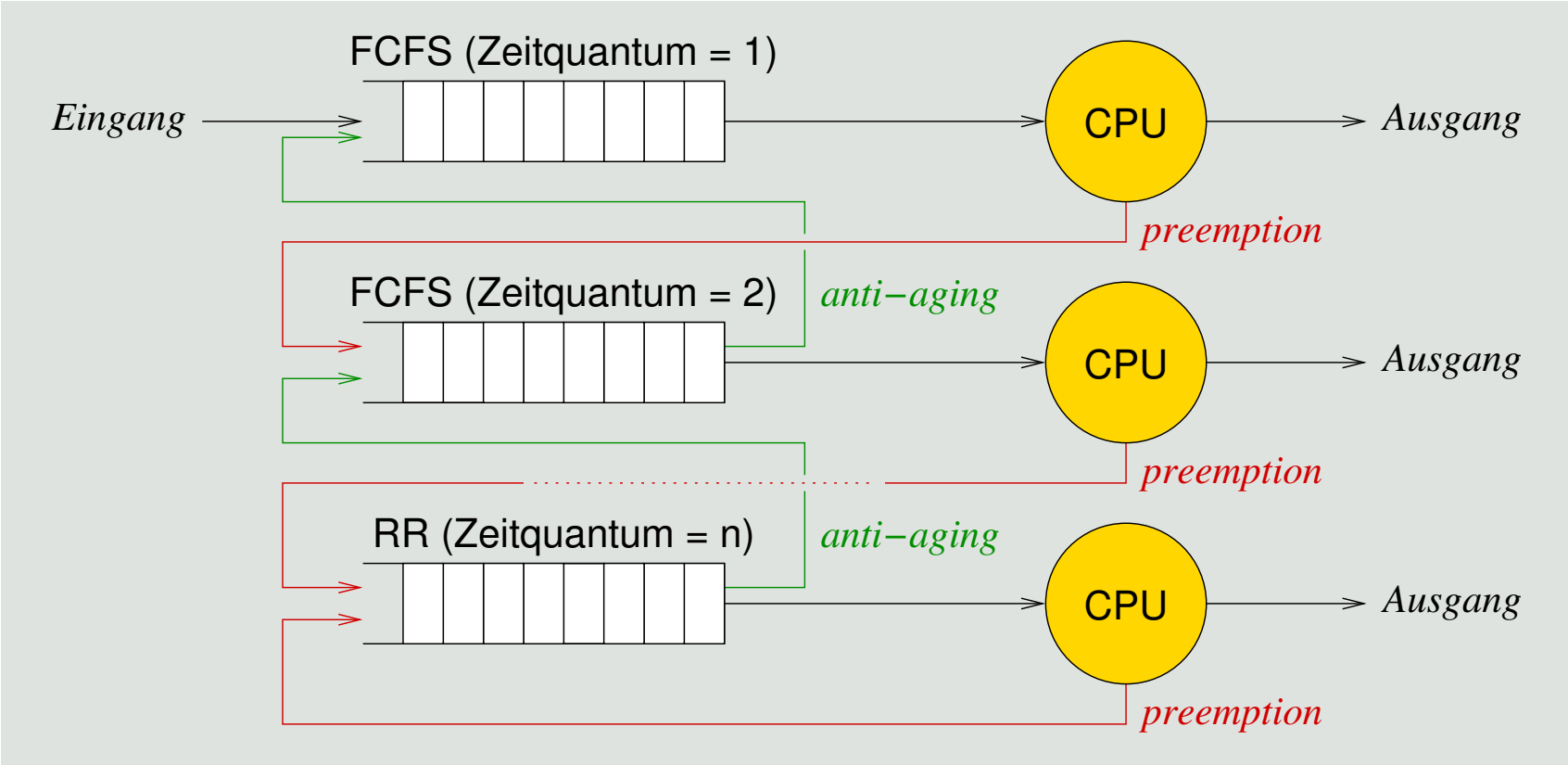
MLFQ

multilevel feedback queue

Begünstigt kurze/interaktive Prozesse, ohne die relativen Stoßlängen kennen zu müssen.

- Prozesse werden nach ihrer **Ankunftszeit** ein- und in regelmäßigen Zeitabständen (periodisch) umgeplant
  - Hierarchie von Bereitlisten, je nach Anzahl der **Prioritätsebenen**
    - erstmalig eintreffende Prozesse steigen oben ein
    - Zeitscheibenablauf drückt den laufenden Prozess weiter nach unten
  - je nach Ebene verschiedene Einreihungsstrategien und -parameter
    - unterste Ebene arbeitet nach RR, alle anderen (höheren) nach FCFS
    - die Zeitscheibengrößen nehmen von oben nach unten zu
- **Bestrafung** (*penalisation*)
  - Prozesse mit langen Rechenstößen fallen nach unten durch
  - Prozesse mit kurzen Rechenstößen laufen relativ schnell durch
- **Alterung** (*ageing*)
  - nach unten durchfallende Prozesse finden seltener statt
  - durchgefallene Prozesse nach einer **Bewährungsfrist** wieder anheben

MLFQ: Bestrafung und Bewährung



feedback (FB)

Gliederung

Gegenüberstellung

	FCFS	RR	VRR	SPN	HRRN	SRTF
kooperativ	✓			(✓)	(✓)	
verdrängend		✓	✓			✓
probabilistisch				✓	✓	✓
deterministisch	keine bzw. nicht von sich aus allein $\rightsquigarrow$ EZS [13]					

- MLQ und MLFQ erlauben eine Kombination dieser Verfahren, jedoch abgestuft und nicht alle zusammen auf derselben Ebene
  - dadurch wird letztlich eine **Priorisierung** der Strategien vorgenommen
    - entsprechend der globalen Strategie, die den Ebenenwechsel steuert
  - teilweise wird so speziellen Anwendungsbedürfnissen entgegengekommen
    - z.B. FCFS priorisieren $\rightsquigarrow$ „*number crunching*“ fördern
- jedes dieser Verfahren stellt bestimmte **Gütemerkmale** [7] in den Vordergrund und vergibt damit indirekt Prioritäten an Prozesse

Prioritäten setzende Verfahren

Statische Prioritäten (MLQ) vs. dynamische Prioritäten (VRR, SPN, SRTF, HRRN, MLFQ).

- **Prozessvorrang** bedeutet die bevorzugte Einlastung von Prozessen mit höherer Priorität und wird auf zwei Arten bestimmt:
  - statisch
    - Zeitpunkt der **Prozesserzeugung** $\rightsquigarrow$ Laufzeitkonstante
    - wird im weiteren Verlauf nicht mehr verändert
    - erzwingt die deterministische Ordnung zw. Prozessen
  - dynamisch
    - „jederzeit“ im **Prozessintervall** $\rightsquigarrow$ Laufzeitvariable
    - die Berechnung erfolgt durch das Betriebssystem
      - ggf. in Kooperation mit den Anwendungsprogrammen
    - erzwingt keine deterministische Ordnung zw. Prozessen
- damit ist allerdings noch nicht **Echtzeitverarbeitung** garantiert, bei der Prozessvorrang eine maßgebliche Rolle spielt
  - die **Striktheit von Terminvorgaben** ist einzuhalten: weich, fest, hart
    - entsprechend der jeweiligen Anforderungen der Anwendungsdomäne
  - keines der behandelten Verfahren sichert dies dem Anwendungssystem zu

Resümee

. . . Fokus lag auf Durschnittsleistung (*average performance*)

- Prozesseinplanung unterliegt einer breit gefächerten **Einordnung**
  - kooperativ/verdrängend
  - deterministisch/probabilistisch
  - statisch/dynamisch
  - asymmetrisch/symmetrisch
- die entsprechenden **Verfahrensweisen** sind z.T. sehr unterschiedlich
  - FCFS: kooperativ
  - RR, VRR: verdrängend
  - SPN, HRRN, SRTF: probabilistisch
  - MLQ, MLFQ (FB): mehrstufig
- Prioritäten setzende Verfahren legen einen **Prozessvorrang** fest
  - FCFS: Ankunftszeit
  - RR: Ankunftszeit, VRR: Ankunftszeit nach Ein-/Ausgabestoßende
  - SPN: Rechenstoß, HRRN: Verhältniswert, SRTF: Rechenstoßrest
- eine weitere Dimension ist die **Striktheit von Terminvorgaben**
  - die jedoch keins der behandelten Verfahren an sich berücksichtigt. . .

Literaturverzeichnis I

[1] Bayer, R. :
Symmetric binary B-Trees: Data structure and maintenance algorithms.
In: *Acta Informatica* 1 (1972), Dezember, S. 290–306

[2] Coffman, E. G. ; Denning, P. J.:
Operating System Theory.
Prentice Hall, Inc., 1973

[3] Conway, R. W. ; Maxwell, L. W. ; Millner, L. W.:
Theory of Scheduling.
Addison-Wesley, 1967

[4] Guibas, L. J. ; Sedgewick, R. :
A dichromatic framework for balanced trees.
In: *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS 1978)*, IEEE, 1978, S. 8–21

[5] Hönig, T. :
Der O(1)-Scheduler im Kernel 2.6.
In: *Linux Magazin* (2004), Februar, Nr. 2

Literaturverzeichnis II

[6] Kleinöder, J. ; Schröder-Preikschat, W. :
Dialog- und Echtzeitverarbeitung.
In: [9], Kapitel 7.2

[7] Kleinöder, J. ; Schröder-Preikschat, W. :
Einplanungsgrundlagen.
In: [9], Kapitel 9.1

[8] Kleinöder, J. ; Schröder-Preikschat, W. :
Prozesse.
In: [9], Kapitel 6.1

[9] Kleinöder, J. ; Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)

[10] Kleinrock, L. :
Queuing Systems. Bd. I: Theory.
John Wiley & Sons, 1975

[11] Kornai, J. :
Economics of Shortage.
North-Holland Publishing Company, 1980

Literaturverzeichnis III

[12] Lister, A. M. ; Eager, R. D.:
Fundamentals of Operating Systems.
The Macmillan Press Ltd., 1993. –
ISBN 0–333–59848–2

[13] Liu, J. W. S.:
Real-Time Systems.
Prentice-Hall, Inc., 2000. –
ISBN 0–13–099651–3

UNIX klasssich

- zweistufig, Anwortzeiten minimierend, Interaktivität fördernd:

  low-level kurzfristig; präemptiv, MLFQ, **dynamische Prioritäten**
  - einmal pro Sekunde: $prio = cpu\_usage + p\_nice + base$
  - CPU-Nutzungsrecht mit jedem „Tick“ $(1/10\,s)$ verringert
    - Prioritätswert kontinuierlich um „Tickstand“ erhöhen
    - je höher der Wert, desto niedriger die Priorität
  - über die Zeit gedämpftes CPU-Nutzungsmaß: *cpu_usage*
    - der Dämpfungsfilter variiert von UNIX zu UNIX

  high-level mittelfristig; mit **Umlagerung** (*swapping*) arbeitend

- Prozesse können relativ zügig den Betriebssystemkern verlassen
  - gesteuert über die beim Schlafenlegen einstellbare **Aufweckpriorität**

UNIX 4.3 BSD

- MLFQ (32 Warteschlangen, RR), dynamische Prioritäten (0–127):

Berechnung der **Benutzerpriorität** bei jedem vierten Tick (40 ms)

- $p\_usrpri = PUSER + \left[\frac{p\_cpu}{4}\right] + 2 \cdot p\_nice$
  - mit $p\_cpu = p\_cpu + 1$ bei jedem Tick (10 ms)
  - **Gewichtungsfaktor** $-20 \leq p\_nice \leq 20$ (vgl. `nice(2)`)
- Prozess mit Priorität P kommt in Warteschlange $P/4$

Glättung des Wertes der **Prozessornutzung** (p_cpu), sekündlich

- $p\_cpu = \frac{2 \cdot load}{2 \cdot load + 1} \cdot p\_cpu + p\_nice$
- Sonderfall: Prozesse schliefen länger als eine Sekunde
  - $p\_cpu = \left[\frac{2 \cdot load}{2 \cdot load + 1}\right]^{p\_slptime} \cdot p\_cpu$

- Annahme 1:
  - mittlere Auslastung (*load*) sei 1

$$\frac{2 \cdot load}{2 \cdot load + 1} = \frac{2}{3} = 0.66 \rightsquigarrow p\_cpu = 0.66 \cdot p\_cpu + p\_nice$$

- Annahme 2:
  - Prozess sammelt T_i Ticks im Zeitinterval i an, $p\_nice = 0$:

$$\begin{aligned} p\_cpu &= 0.66 \cdot T_0 \\ &= 0.66 \cdot (T_1 + 0.66 \cdot T_0) = 0.66 \cdot T_1 + 0.44 \cdot T_0 \\ &= 0.66 \cdot T_2 + 0.44 \cdot T_1 + 0.30 \cdot T_0 \\ &= 0.66 \cdot T_3 + \cdots + 0.20 \cdot T_0 \\ &= 0.66 \cdot T_4 + \cdots + 0.13 \cdot T_0 \end{aligned}$$

  - nach fünf Sekunden gehen nur noch etwa 13 % der „Altlast“ ein

UNIX Solaris

- MLQ (4 Klassen) und MLFQ (60 Ebenen, Tabellensteuerung)

quantum	*tqexp*	*slpret*	*maxwait*	*lwait*	Ebene
200	0	50	0	50	0
200	0	50	0	50	1
		...			
40	34	55	0	55	44
40	35	56	0	56	45
40	36	57	0	57	46
40	37	58	0	58	47
40	38	58	0	58	48
40	39	58	0	59	49
40	40	58	0	59	50
40	41	58	0	59	51
40	42	58	0	59	52
40	43	58	0	59	53
40	44	58	0	59	54
40	45	58	0	59	55
40	46	58	0	59	56
40	47	58	0	59	57
40	48	58	0	59	58
20	49	59	32000	59	59

`/usr/sbin/dispadmin -c TS -g`

MLQ (Klasse)		Priorität
time-sharing	TS	0–59
interactive	IA	0–59
system	SYS	60–99
real time	RT	100–109

MLFQ in Klasse TS bzw. IA:

quantum	Zeitscheibe (ms)
tqexp	Ebene bei Bestrafung
slprt	Ebene nach Deblockierung
maxwait	ohne Bedienung (s)
lwait	Ebene bei Bewährung

- Besonderheit: *dispatch table* (TS, IA) kapselt alle Entscheidungen
  - kunden-/problemspezifische Lösungen durch verschiedene Tabellen

UNIX Solaris

Beispiel:

- 1 × CPU-Stoß à 1000 ms
- 5 × E/A-Stoß → CPU-Stoß à 1 ms

#	Ebene	CPU-Stoß	Ereignis
1	59	20	Zeitscheibe
2	49	40	Zeitscheibe
3	39	80	Zeitscheibe
4	29	120	Zeitscheibe
5	19	160	Zeitscheibe
6	9	200	Zeitscheibe
7	0	200	Zeitscheibe
8	0	180	E/A-Stoß
9	50	1	E/A-Stoß
10	58	1	E/A-Stoß
11	58	1	E/A-Stoß
12	58	1	E/A-Stoß

Variante: nach 640 ms. . .

- Prozess wird verdrängt, muss auf die erneute Einlastung warten
- Alterung des wartenden Prozesses wird durch Prioritätsanhebung entgegengewirkt (*anti-aging*)
- die höhere Ebene erreicht, sinkt der Prozess danach wieder ab

	. . .		
7	0	20	*anti-aging*
8	50	40	Zeitscheibe
9	40	40	Zeitscheibe
10	30	80	Zeitscheibe
11	20	120	Zeitscheibe
12	10	80	E/A-Stoß
13	50	1	E/A-Stoß
	. . .		

Linux 2.4

- Prozessen zugewiesene Prozessorzeit ist in **Epochen** unterteilt:

  beginnen alle lauffähige Prozess haben ihr Zeitquantum erhalten

  enden alle lauffähigen Prozesse haben ihr Zeitquantum verbraucht

- **Zeitquanten** (Zeitscheiben) variieren mit Prozessen und Epochen:
  - jeder Prozess besitzt eine einstellbare Zeitquantumbasis (`nice(2)`)
    - 20 Ticks $\approx$ 210 ms
    - das Zeitquantum eines Prozesses nimmt periodisch (Tick) ab
  - beide Werte addiert liefert die dynamische Priorität eines Prozesses
    - dynamische Anpassung: $quantum = quantum/2 + (20 - nice)/4 + 1$

- **Echtzeitprozessse** besitzen statische Prioritäten: 0–99
  - je kleiner der Wert, desto höher die Priorität
  - schwache Echtzeit (vgl. [6, S. 16])

Linux 2.4

- Prozesseinplanung unterscheidet zwischen drei ***Scheduling*-Klassen**:

FIFO	verdrängbare, kooperative Echtzeitprozesse	} eine Bereitliste
RR	Echtzeitprozesse derselben Priorität	
other	konventionelle („*time-shared*“) Prozesse	

- Prozessauswahl greift auf eine **Gütefunktion** zurück: $O(n)$

$v = -1000$	der Prozess ist *Init*
$v = 0$	der Prozess hat sein Zeitquantum verbraucht
$0 < v < 1000$	der Prozess hat sein Zeitquantum nicht verbraucht
$v \geq 1000$	der Prozess ist ein Echtzeitprozess

- Prozesse können bei der Auswahl einen **Bonus** („*boost*“) erhalten
  - sofern sie sich mit dem Vorgänger den Adressraum teilen

- Prozessplanung hat **konstante Berechnungskomplexität** [5]:
  - Prioritätsfelder zwei Tabellen pro CPU: *active*, *expired*
    - jedes Feld eine Bitkarte (*bitmap*) von n Einträgen
    - mit n gleich der Anzahl von Prioritätsebenen
  - Prioritätsebenen 140 Ebenen = Einträge pro Tabelle
    - 0–99 für Echtzeit-, 100–139 für sonstige Prozesse
    - pro Ebene eine (doppelt verkettete) Bereitliste
  - ist Bitkartenposition i gesetzt (1, *true*), dann ist wenigstens ein Prozess auf der Bereitliste von Ebene i verzeichnet
  - zur Listenauswahl wird die Bitkarte von Anfang ($i = 0$) an abgesucht
    - ggf. unter Zuhilfenahme spezieller Bitoperationen des Prozessors (x86: BSF)
- Prioritäten gemeiner Prozesse skalieren je nach Interaktivitätsgrad
  - **Bonus** (-5) für interaktive Prozesse, **Strafe** ($+5$) für rechenintensive
  - berechnet am Zeitscheibenende: $prio = MAX\_RT\_PRIO + nice + 20$
- Ablauf des Zeitquantums befördert aktiven Prozess ins „*expired*“-Feld
  - zum Epochenwechsel werden die Tabellen ausgetauscht: Zeigerwechsel

Linux 2.6.23

- der Planer verzichtet auf Prozessheuristik und feste Zeitscheiben
  - vielmehr garantiert er jedem Prozess einen bestimmten **Prozessoranteil**
    - $1/N$ Zeiteinheiten, mit N gleich der Anzahl der laufbereiten Prozesse
    - der Gesamtanteil wächst und schrumpft mit der Gesamtzahl dieser Prozesse
    - der relative Anteil variiert mit der Priorität (`nice(2)`) eines Prozesses
  - in Bezug auf diesen Anteil gilt dabei für einen laufenden Prozess:
    (a) er kann den Prozessor früher abgeben (blockiert: E/A-intensiver Prozess)
    (b) er darf den Prozessor länger behalten (rechenintensiver Prozess)
    (c) er wird verdrängt, sobald er sein $1/N$-tel Zeitanteil konsumiert hat und ein anderer Prozess wartet, er länger als der kürzeste laufbereite Prozess läuft
  - Fall (c) wird regelmäßig überprüft, durch einen Zeitgeber (*clock tick*)
- Rechenstoßlängen der Prozesse haben eine **minimale Granularität**

Definition (*minimum granularity, MG*)

Die Zeitspanne, die einem Prozess auf dem Prozessor zuzubilligen ist, bevor dem Prozess der Prozessor wieder entzogen werden kann.

- 1 – 4 ms, die **Periodenlänge** des Zeitgebers; Vorgabe 4 ms

- die **Ziellatenz** des Planers bestimmt den effektiven Prozessoranteil

Definition (*target latency*, *TL*)

Die Mindestzeit, die für jeden laufbereiten Prozess zur Verfügung zu stellen ist, um wenigstens eine Runde des Prozessors zu erhalten.

  - auch die untere Grenze der **Periodenlänge** des Planers; Vorgabe 20 ms
  - $P_{sched} = \begin{cases} TL & \text{wenn } N \leq TL/MG \\ N \times MG & \text{sonst} \end{cases}$

- wartende Prozesse geben den Beginn ihres nächsten Zeitschlitzes vor

Definition (*virtual runtime*, *VR*)

Die aus den jeweiligen Rechenstoßlängen akkumulierte Laufzeit eines Prozesses in *ns*, gewichtet mit seiner „Nettigkeit“ (*niceness*).

  - wird in jeder Zeitgeberperiode oder bei der Prozessorabgabe aktualisiert
  - bestimmt den Moment des Prozessorentzugs und den Bereitlistenplatz
    - der laufende Prozess wird verdrängt, wenn $VR_{running} > VR_{ready}$

- Prozesse vom Prozessor zu verdrängen ist **Mangelwirtschaft** [11] ☹
  - es besteht ein Mangel an „Waren“ (Prozessoren)
  - während genug „Geld“ (Prozesse) zum Kauf dieser Waren vorhanden ist

Definition (idealer Prozessor)

Ein Prozessor, der jeden Prozess mit genau gleicher Geschwindigkeit parallel ausführen kann, jeweils mit $1/N$-tel Zeiteinheiten (*ns*).

- alle Prozesse erhalten den Prozessor für die gleiche relative Laufzeit

Definition (*maximum execution time, MET*)

Die Zeit, die ein Prozess auf einen idealen Prozessor erwarten würde.

  - für den nächsten Prozess auf der Bereitliste ist dies die Wartezeit auf den Prozessor, normalisiert auf die Gesamtzahl der lauffähigen Prozesse
  - nach $MET = P_{sched}/N$ Zeiteinheiten wird der Prozess mit der kürzesten VR den Prozessor zugeteilt erhalten

- die Bereitliste ist ein **balancierter binärer Suchbaum** [1]:[3] $O(\log n)$
  - n gleicht der Anzahl der Knoten im Baum, d.h., der laufbereiten Prozesse
  - sortiert nach der „**Vorkaufszeit**“ eines Prozesses für den Prozessor in *ns*

Definition (*preemption time*)

Der Zeitpunkt eines (auf der Bereitliste) wartenden Prozesses zur Ausübung des Vorkaufsrechts, den Prozessor als erster angeboten zu bekommen.

  - die **gewichtete virtuelle Laufzeit** eines vormals gelaufenen Prozesses
  - diese variiert von Prozess zu Prozess auf der Bereitliste
  - ein zunehmender Wert von links nach rechts innerhalb des RS-Baums
- Prozesse sind nach dem Beginn ihres nächsten Rechenstoßes gelistet
  - frühestens am Ende der laufenden Zeitgeberperiode wird verdrängt
    - erhält der ganz links im RS-Baum verzeichnete Prozess den Prozessor und
    - der verdrängte Prozess wird gemäß seiner *VR* im RS-Baum eingetragen
  - ein erzeugter Prozess erhält die **minimale aktuelle virtuelle Laufzeit**
    - die kleinste vom System festgestellte virtuelle Laufzeit eines Prozesses

[3] Rot-Schwarz-Baum (RS-Baum: *red-black tree* [4])

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – X.1 Prozesssynchronisation: Nichtsequentialität

Wolfgang Schröder-Preikschat

9. November 2021

Agenda

Einführung

Kausalitätsprinzip
- Parallelisierbarkeit
- Kausalordnung
- Aktionsfolgen

Sequentialisierung
- Koordinierung
- Konkurrenz

Verfahrensweisen
- Einordnung
- Fallstudie
- Lebendigkeit

Zusammenfassung

Gliederung

Lehrstoff

- **Nebenläufigkeit** von Prozessen als Eigenschaft begreifen, die ein Betriebssystem fördern und schon gar nicht behindern sollte
  - um das Leistungspotential mehr- oder vielkerniger Prozessoren zu nutzen
  - **Parallelrechner** sind gang und gäbe, brauchen aber parallele Abläufe
- erkennen, dass Nebenläufigkeit jedoch nur für **kausal unabhängige Prozesse** gilt, die nicht durchgängig gegeben sind
  - problembedingte Rollenspiele von Prozessen (Konsument vs. Produzent)
  - Konkurrenzsituationen bei Zugriffen auf gemeinsame Betriebsmittel
- Prinzipen kennenlernen/vertiefen, um **kausal zusammenhängende Aktionen** nacheinander stattfinden zu lassen
  - **Sequentialisierung** von gleichzeitigen (gekoppelten) Prozessen erzwingen
  - Konkurrenz dieser Prozesse durch wechselseitigen Ausschluss koordinieren
  - verstehen, dass Aktionen auf vertikaler Ebene nicht unteilbar sein müssen
  - **Unteilbarkeit** in Bezug auf Betriebsmittel und Aktionen kennenlernen
- Verfahrensweisen zur Synchronisation erklären, mit einer **Fallstudie** Probleme und deren Lösungen aufzeigen
  - ein- und mehrseitige Synchronisation am Beispiel „*bounded buffer*“

Gliederung

Nebenläufige Aktionen

Sequentielles $\mapsto$ *Nichtsequentielles Programm*

- **Nebenläufigkeit** (*concurrency*) bezeichnet das Verhältnis von nicht kausal abhängigen Ereignissen, die sich also nicht beeinflussen

```
1     foo = 4711;
2     bar = 42;
3  foobar = foo + bar;
4  barfoo = bar + foo;
5     hal = foobar + barfoo;
```

  - Aktion „=“ in Zeile 1 ist nebenläufig zu der in Zeile 2
  - die Aktionen „=“ und „+“ in Zeile 3 sind nebenläufig zu denen in Zeile 4

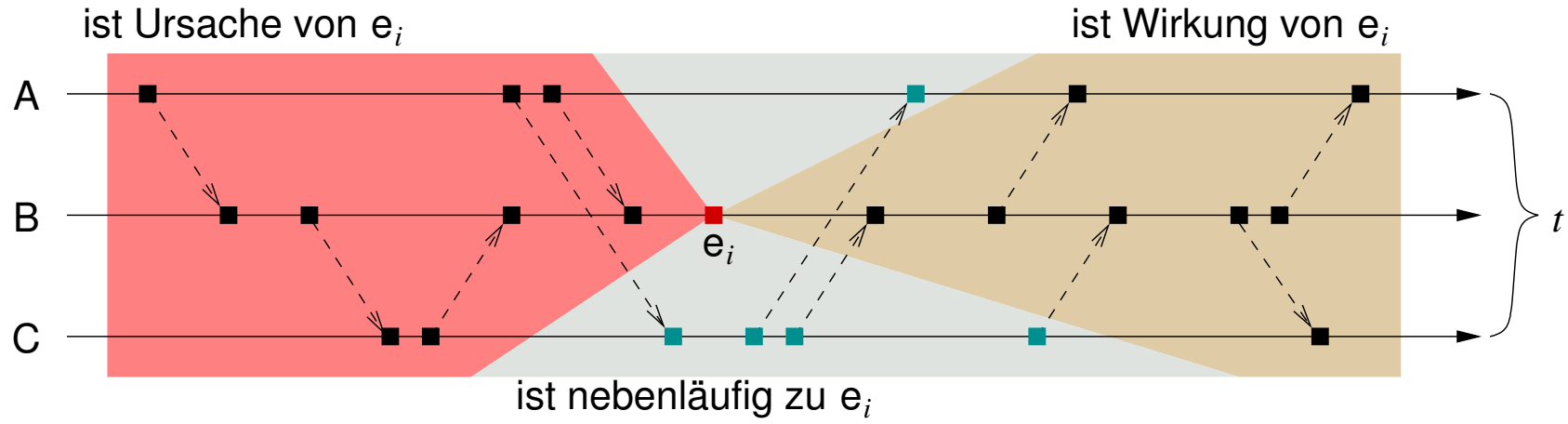
  - in logischer Hinsicht sind Aktionen potentiell nebenläufig, wenn keine das Resultat der anderen benötigt
  - in physischer (d.h., körperlicher) Hinsicht ist für jede dieser Aktionen ein Aktivitätsträger erforderlich, der autonom agieren kann

- **Kausalität** (lat. *causa*: Ursache) ist die Beziehung zwischen Ursache und Wirkung, d.h., die ursächliche Verbindung zweier Ereignisse
  - Ereignisse sind nebenläufig, wenn keines Ursache des anderen ist

[1] Aktion: Anweisungsausführung einer (virtuellen/realen) Maschine. [8, S. 12]

Ursache und Wirkung

Nebenläufigkeit als relativistischer Begriff von Gleichzeitigkeit:



- ein Ereignis ist nebenläufig zu einem anderen (e_i), wenn es im **Anderswo** des anderen Ereignisses (e_i) liegt
  - d.h., weder in der Zukunft noch in der Vergangenheit des anderen
- das Ereignis ist nicht Ursache/Wirkung des anderen Ereignisses (e_i)
  - ggf. aber Ursache/Wirkung anderer (von e_i verschiedener) Ereignisse

Daten- und Zeit(un)abhängigkeit

- ein „im Anderswo anderer Ereignisse liegendes“ Ereignis steht für eine **nebenläufige Aktion**, sofern eben:
  - allgemein
    - keine das Resultat der anderen benötigt (S. 6)
    - **Datenabhängigkeiten** gleichzeitiger Prozesse beachten
  - speziell
    - keine die **Zeitbedingungen** der anderen verletzt
    - zusätzliches, zwingendes Merkmal nur für Echtzeitbetrieb
      - Zeitpunkte dürfen nicht/nur selten verpasst werden
      - Zeitintervalle dürfen nicht/nur begrenzt gedehnt werden
- je nach Art der Beziehung zwischen den Ereignissen bzw. Aktionen, ist die **Konsequenz für gleichzeitige Prozesse** verschieden

„ist Ursache von“
„ist Wirkung von“ } $\rightsquigarrow$ **Koordinierung** (vor/zur Laufzeit)

„ist nebenläufig zu“ $\rightsquigarrow$ **Parallelität** (implizit)

- Koordinierung durch **Sequentialisierung**: Schaffen einer Ordnung für eine Menge von Aktionen entlang der Kausalordnung

Gleichzeitige Prozesse

vgl. auch [3]

Definition (*concurrent/simultaneous processes*)

Mehrere (ggf. **nichtsequentielle**) **Prozesse**, durch die sich mehr als eine Aktionsfolge in Raum und Zeit überlappen.

- notwendige Bedingung dazu ist die Fähigkeit des Betriebssystems zur **Simultanverarbeitung** (*multiprocessing*) von Programmen
  - **vertikal** ausgelegt, durch Multiplexen ein und desselben Prozessors
    - Mehrbenutzer-, Teilnehmer- oder Zeitmultiplexbetrieb: *time sharing* [1]
    - pseudo Parallelität durch asynchrone Programmunterbrechungen (*interrupts*)
  - **horizontal** ausgelegt, durch Vervielfachung des Prozessors
    - symmetrischer oder asymmetrischer Multiprozessorbetrieb: *multiprocessing*
    - echte Parallelität durch mehrere physische Ausführungseinheiten
- hinreichende Bedingung ist die Verfügbarkeit von Programmen, durch die zugleich mehrere Ausführungsstränge möglich werden
  - ein nichtsequentielles Programm oder mehrere sequentielle Programme
  - eine beliebige Kombination derartiger Programme

Gekoppelte Prozesse

vgl. auch [3]

Definition (*interacting processes*)

Gleichzeitige Prozesse, die durch direkte order indirekte Nutzung einer oder mehrerer gemeinsamer Variablen bzw. Ressourcen interagieren.

- dabei interagieren die Prozesse schon im Moment des Zugriffs, da sie dadurch **Interferenz**[2] in zeitlicher Hinsicht erzeugen
  - durch logisch gleichzeitige Zugriffe auf höherer Ebene, wenn diese jedoch auf tieferer Ebene nur sequentiell durchgeführt werden können/dürfen
  - z.B. Sequentialisierung durch den Bus oder einen kritischen Abschnitt
- entscheidend ist jedoch die **logische Bedeutung** der Variablen bzw. Ressource für die beteiligten gleichzeitigen Prozesse
  - Medium zur Kommunikation mit dem jeweils anderen internen Prozess
  - Instrument zur Interaktion mit einem externen Prozess (Peripherie)
- diese Bedeutung schließt **Datenabhängigkeiten** ein und bezieht sich gerade auch auf das **Rollenspiel** der Prozesse
  - Produzent/Konsument (Datum), Sender/Empfänger (Signal, Nachricht)

[2] Abgeleitet von (altfrz.) *s'entreferir* „sich gegenseitig schlagen".

Gliederung

Koordination von Konkurrenz

- **gleichzeitige Aktionen** überlappen einander in Raum und Zeit
  - i der **Moment** ihres Zusammentreffens ist i. A. nicht vorherbestimmt
  - ii Aktionen können komplex sein (d.h., **mehrere Einzelschritte** umfassen)
  - iii ihre besondere Eigenschaft ist die **Teilbarkeit in zeitlicher Hinsicht**
- kausal zusammenhängende Aktionen müssen nacheinander stattfinden
  - *off-line*
    - statische Einplanung, Daten- und Kontrollflussabhängigkeiten
    - der Ablaufplan sorgt für die **implizite Synchronisation**
    - analytischer Ansatz, der Vorabwissen erfordert (s. aber i, oben)
  - *on-line*
    - dynamische Einplanung, ausgelöst durch externe Ereignisse
    - **explizite Synchronisation** durch Programmanweisungen
    - konstruktiver Ansatz, der ohne Vorabwissen auskommen muss
- explizite Prozesssynchronisation kann **Wettstreit** hervorbringen
  - bei Mitbenutzung (*sharing*) desselben wiederverwendbaren Betriebsmittels
  - bei Übergabe (*handover*) eines konsumierbaren Betriebsmittels

Hinweis

*Die gewählte Methode sollte minimal invasiv auf die Prozesse wirken, bei expliziter Synchronisation ist **Interferenz** unvermeidbar. . .*

Atomare Aktionen

Definition (atomare Aktion)

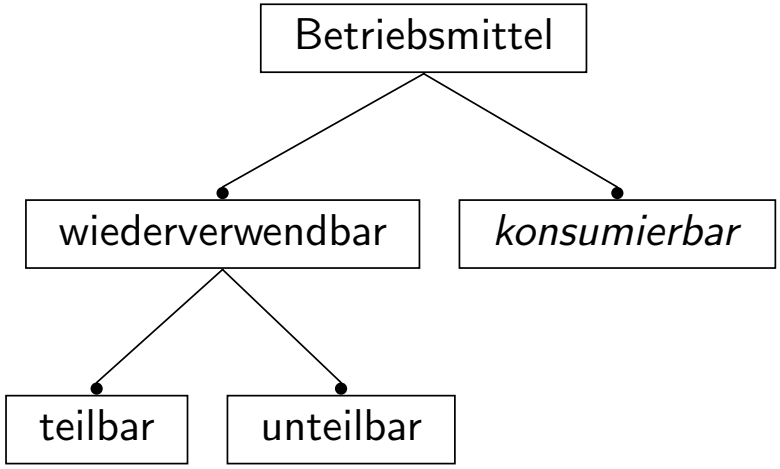
Eine Aktion, ***deren Einzelschritte*** nach außen sichtbar im Verbund ***scheinbar gleichzeitig stattfinden***.

- dabei wird das Herstellen von Gleichzeitigkeit (Simultanität) durch **Synchronisation**[3] der gekoppelten Aktionen/Prozesse erreicht
  - **Koordination** der Kooperation und Konkurrenz zwischen Prozessen [6]
  - Sequentialisierung von Ereignissen entlang einer Kausalordnung
  - Aktionen gleichzeitig/in einer bestimmten Reihenfolge stattfinden lassen
- zentrales Konzept, um **gleichzeitige Aktionen** zu **koordinieren**, ist der **wechselseitige Ausschluss** (*mutual exclusion*)
  - i ein **kritischer Abschnitt** [2, S. 11] der Maschinenprogrammebene
  - ii eine **Elementaroperation** (*read-modify-write*) der Befehlssatzebene
- dabei ist die Auswirkung auf die beteiligten Prozesse je nach Ebene der Abstraktion bzw. Paradigma sehr unterschiedlich
  - d.h., die Synchronisation wirkt blockierend (i) oder nichtblockierend (ii)

[3] (gr). *sýn*: zusammen, *chrónos*: Zeit

Betriebsmittel und Aktionen

- je nach **Betriebsmittelart** (vgl. [8, S. 9–10]) ist die Nutzung durch gleichzeitige Prozesse eingeschränkt:



Hardware
- CPU, Speicher
- Geräte (Peripherie)
- *Signale*

Software
- Dateien, E/A-Puffer
- Seitenrahmen
- Deskriptoren, ...
- *Signale, Nachrichten*

- bereits Aktionen zum **Zugriff** auf ein unteilbares wiederverwendbares Betriebsmittel unterliegen dem wechselseitigen Ausschluss
  - **mehrseitige/multilaterale Synchronisation** gekoppelter Prozesse
- wohingegen die Aktion der **Entgegennahme** eines konsumierbaren Betriebsmittels nur auf einen Prozess verzögernd wirkt
  - **einseitige/unilaterale Synchronisation** gekoppelter Prozesse

Unteilbarkeit

Definition (in Anlehnung an den Duden)

(Betriebssystem) Umstand, der die Verteilung der Betriebsmittel auf mehrere Prozessoren *oder* Prozesse verhindert.

- unteilbar ist ein Betriebsmittel, wenn es zu einem Zeitpunkt von nur genau einem Prozessor/Prozess genutzt werden darf
  - Zugriffsoperationen darauf können/dürfen nicht zeitlich zerteilt werden
  - sie müssen **atomar**, d.h., als **Elementaroperation** ausgeführt werden

  ↪ Aktion/Aktionsfolge mehrerer kausal abhängender Einzelschritte
- teilbar ist ein Betriebsmittel, wenn mehrere Prozessoren/Prozesse es gleichzeitig benutzen dürfen
  - es dem einem entzogen und einem anderen gegeben werden darf
  - Zugriffe auf das Betriebsmittel können/dürfen zeitlich zerteilt werden

  ↪ Aktion/Aktionsfolge mehrerer kausal unabhängiger Einzelschritte
- **beachte**: ein Betriebsmittel besonderer Art ist der Prozessor im Falle eines kritischen Abschnitts in einem nichtsequentiellen Programm
  - das *Unteilbarsein* auf Maschinenprogramm- oder Befehlssatzebene (S. 13)

Wettstreit

Definition (Duden)

Bemühen, einander in etwas zu übertreffen, einander den Vorrang streitig zu machen.

- ein unter gleichzeitigen Prozessen auftretender **Konflikt**, der diese implizit koppelt und damit zur **Interaktion** zwingt, wenn:
  1. Zugriffe auf wenigstens ein **gemeinsames Betriebsmittel** erfolgen,
  2. nur eine **begrenzte Anzahl** dieses Betriebsmittels vorrätig ist und
  3. die betreffenden Betriebsmittel **unteilbar** und von derselben Art sind
- es entsteht eine **Konkurrenzsituation** (*contention*), wenn einer dieser Prozesse ein Betriebsmittel anfordert, das ein anderer bereits besitzt
  - der anfordernde Prozess blockiert und wartet auf die Freigabe des Betriebsmittels durch den Prozess, der das Betriebsmittel belegt
  - der das Betriebsmittel belegende Prozess löst den auf die Freigabe des Betriebsmittels wartenden Prozess aus, deblockiert ihn wieder

Wechselseitiger Ausschluss

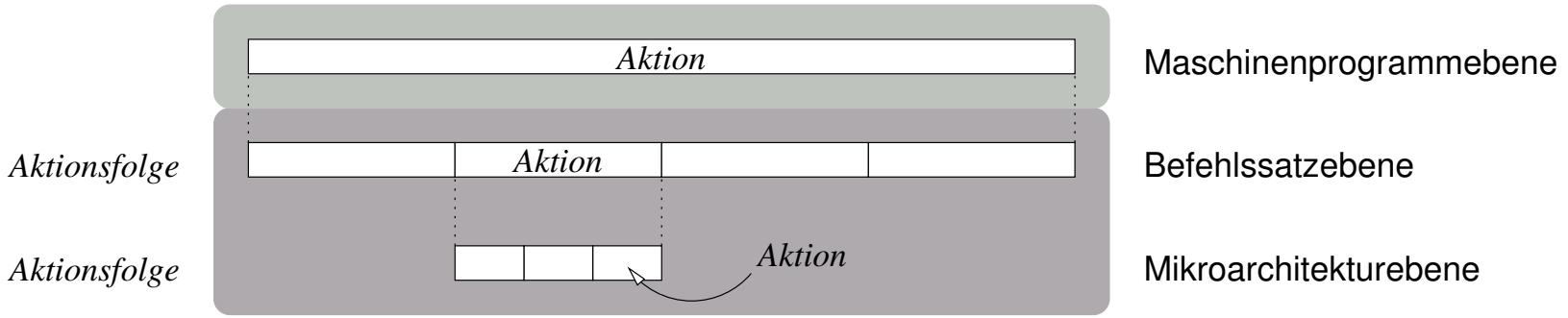
- Protokoll zur **Sequentialisierung** gleichzeitiger Aktionen bei Zugriff auf ein gemeinsames wiederverwendbares/unteilbares Betriebsmittel:

  Vergabe $\mapsto$ vor der Aktion das Betriebsmittel sperren
  - im Moment der Anforderung eines gesperrten Betriebsmittels wird die betreffende Aktion blockiert
  - die blockierte Aktion erwartet (mit/ohne Prozessorabgabe) das Ereignis zur Freigabe des gesperrten Betriebsmittels

  Freigabe $\mapsto$ nach der Aktion das Betriebsmittel entsperren
  - sollten Aktionen die Freigabe dieses Betriebsmittels erwarten, wird es zur **Wiedervergabe** bereitgestellt; das bedeutet:
    - alle Aktionen deblockieren, erneut das Vergabeprotokoll durchlaufen oder
    - eine Aktion deblockieren, für sie das Betriebsmittel (weiterhin) sperren
  - normalerweise nur durch den das Betriebsmittel „besitzenden“ Prozess

- dabei beziehen sich die Aktionen auf ein und dieselbe **Phase** in einem Soft- oder Hardwareprozess, je nach Betrachtungsebene
  - d.h., der Maschinenprogramm- oder Befehlssatzebene (S. 13)

Teilbarkeit in vertikaler Hinsicht



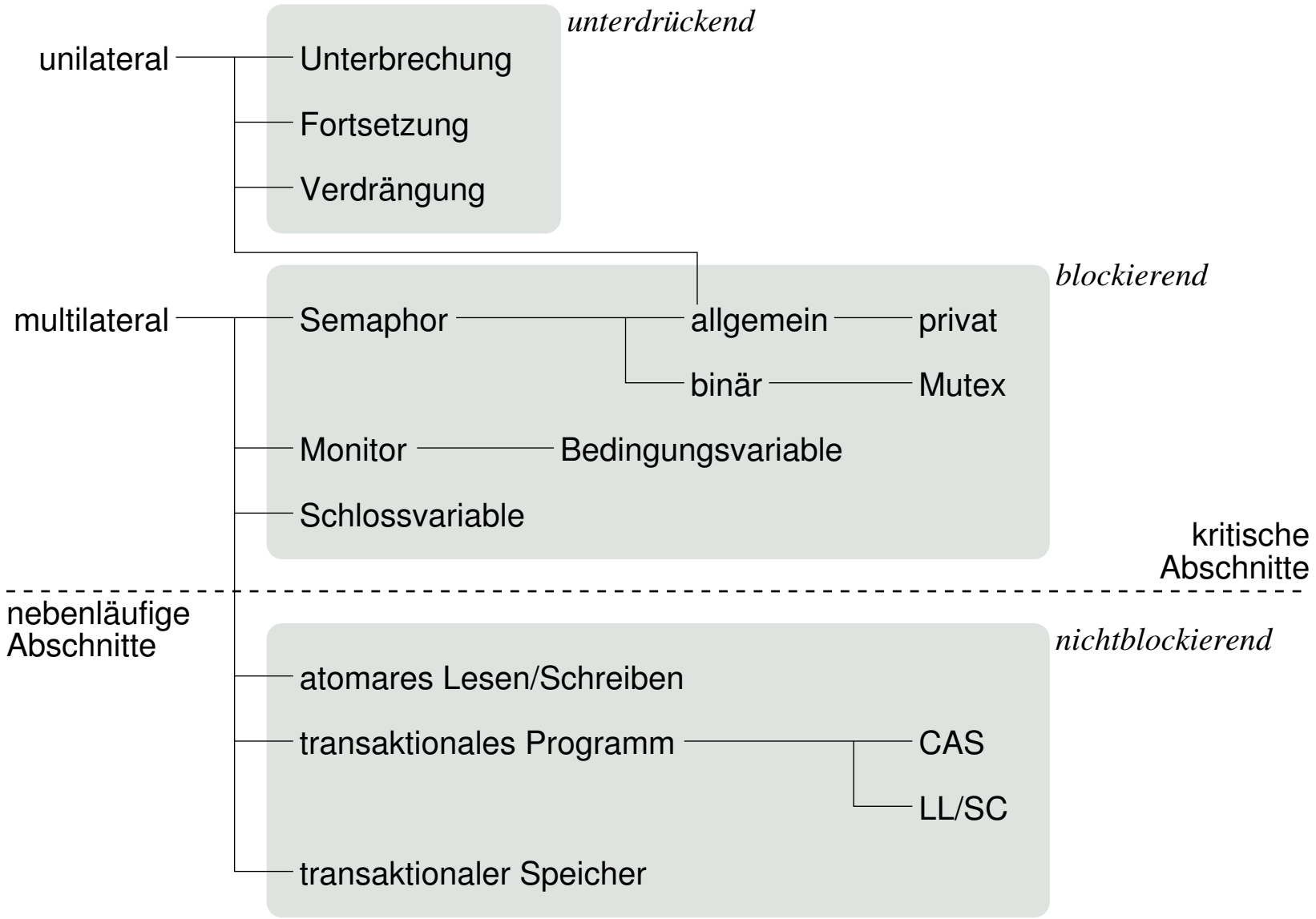
Beachte: Aktion $\sim$ Programmablauf (vgl. [8, S. 11–12])

Ein und derselbe Programmablauf kann auf einer Abstraktionsebene sequentiell, auf einer anderen parallel sein. [9]

- wechselseitiger Ausschluss ist eine Methode der Maschinenprogramm- oder Befehlssatzebene, um **atomare Aktionen** zu schaffen
  - kritischer Abschnitt auf höherer Ebene, Elementaroperation auf tieferer
  - letztere entspricht einem kritischen Abschnitt in der Hardware...
- je nach **Bezugssystem** wirken sich diese Methoden blockierend oder nichblockierend auf zu synchronisierende gleichzeitige Prozesse aus

Gliederung

Synchronisationarten-/techniken



Wirkung

- je nach Art und Technik ist der Effekt von Synchronisation auf die gleichzeitigen Prozesse sehr unterschiedlich:

  unterdrückend
  - verhindert die **Prozessauslösung** anderer Prozesse
    - unabhängig des eventuellen gleichzeitigen Geschehens
    - betrifft konsumierbare Betriebsmittel
  - der aktuelle Prozess wird dabei nicht verzögert

  blockierend
  - sperrt die **Betriebsmittelvergabe** an Prozesse
    - ist nur bei gleichzeitigem Geschehen wirksam
    - betrifft wiederverwendbare/konsumierbare Betriebsmittel
  - der aktuelle Prozess wird möglicherweise suspendiert

  nichtblockierend
  - unterbindet **Zustandsverstetigung** durch Prozesse
    - ist nur bei gleichzeitigem Geschehen wirksam
    - betrifft wiederverwendbare Betriebsmittel: **Speicher**
  - der aktuelle Prozess wird möglicherweise zurückgerollt

- es gibt keine einzige Methode, die nur Vorteile hat, allen Ansprüchen genügt und jeder Anforderung gerecht wird...

Einseitige Synchronisation

- die Verfahren wirken lediglich auf einen der beteiligten Prozesse
  - die anderen beteiligten Prozesse schreiten ungehindert fort[4]
- dabei gibt ein **logischer Programmablauf** die **Bedingungen** vor

  Bedingungssynchronisation
  - der Fortschritt des einen Prozesses ist abhängig von einer Bedingung, die in einem nichtsequentiellen Programm formuliert ist
  - der andere Prozess, der diese Bedingung aufhebt, erfährt dabei keine Verzögerung in seinem Ablauf

  logische Synchronisation
  - die Maßnahme resultiert aus der logischen Abfolge der Aktivitäten
  - vorgegeben durch das „Rollenspiel“ der beteiligten Prozesse
- beachte: andere Prozesse sind jedoch nicht gänzlich unbeteiligt
  - die Aufhebung der Bedingung, die zum Warten eines Prozesses führte, ist von einem anderen Prozess zu leisten
  - **gekoppelte Prozesse** müssen ihrer jeweiligen Rolle gerecht werden...

[4] Ungeachtet der Gemeinkosten (*overhead*) der Verfahren.

Mehrseitige Synchronisation

- die Verfahren wirken auf alle in dem Moment beteiligten Prozesse
  - welcher dieser Prozesse ungehindert fortschreitet, ist unbestimmt
- den **Fortgang** der beteiligten Prozesse explizit kontrollieren

blockierend ∼ pessimistisch: wahrscheinliche, häufige Konkurrenz

- der **wechselseitige Ausschluss** gleichzeitiger Prozesse
  - Warten mit (passiv) oder ohne (aktiv) Prozessorabgabe
- die Verfahren profitieren von der **Maschinenprogrammebene**
  - Systemfunktionen zur Einplanung und Einlastung von Prozessen
- im Regelfall zeitlich begrenzte, **exklusive Betriebsmittelvergabe**

nichtblockierend ∼ optimistisch: unwahrscheinliche, seltene Konkurrenz

- auf Basis einer **Transaktion** zwischen gleichzeitigen Prozessen
  - eine Folge von Aktionen, die nur komplett oder gar nicht stattfinden
- den Verfahren genügen Merkmale der **Befehlssatzebene**
  - **Spezialbefehle** mit atomaren Aktionen der Mikroarchitekturebene
- ungeeignet für wiederverwendbare unteilbare Betriebsmittel

Zusammenführendes Beispiel

vgl. [8, S. 23]

- angenommen, die folgenden Unterprogramme (put und get) werden in beliebiger Reihenfolge und gleichzeitig ausgeführt:

```
1  char buffer[64];
2  unsigned in = 0, out = 0;
3
4  void put(char item) {
5    buffer[in++ % 64] = item;
6  }
7
8  char get() {
9    return buffer[out++ % 64];
10 }
```

↪ mit buffer als wiederverwendbares und item als konsumierbares Betriebsmittel

  - logische Probleme:
    - gepufferte Daten werden ggf. überschrieben: Überlauf
    - ein leerer Puffer gibt ggf. Daten zurück: Unterlauf
  - andere Probleme:
    - überlappendes Schreiben an dieselbe Speicherstelle
    - überlappendes Lesen von derselben Speicherstelle
    - überlappendes Addieren gibt ggf. falsche Zählerwerte

- put und get unterliegen der uni- und multilateralen Synchronisation
  - eine uneingeschränkt gleichzeitige Ausführung darf nicht geschehen

Nichtsequentieller Programmablauf ... Lösung mit „Untiefe“

```
1  char buffer[64];
2  unsigned in = 0, out = 0;
3
4  void put(char item) {
5    if (((in + 1) % 64) == out) await(get);
6
7    buffer[FAA(&in, 1) % 64] = item;
8    cause(put);
9  }
10
11 char get() {
12   if (out == in) await(put);
13
14   char item = buffer[FAA(&out,1)%64];
15   cause(get);
16
17   return item;
18 }
```

„Verlorenes Aufwachen“

Überlappt die Aktion zur Ereignisanzeige mit der Überprüfung der Wartebedingung, kann das Ereignis unbemerkt bleiben.

`await(e)`
- erwarte Ereignis *e*

`cause(e)`
- zeige Ereignis *e* an

`FAA(c,n)`
- verändere Zähler *c* um Wert *n*
- liefere vorherigen Zählerstand
- tue dies unteilbar

Vorbeugung des Ereignisverlusts

- `if (`*`condition`*`) await(`*`event`*`)`: **wettlaufkritische Anweisung**

lost wake-up

Zwischen Feststellung der Wartebedingung eines Prozesses und seiner daraufhin logisch korrekten **Blockierung**, wird diese Bedingung durch einen gleichzeitigen Prozess aufgehoben.

  - die Aktionsfolge 1. ***Prüfen und*** ggf. 2. ***Warten*** findet unwiderruflich statt
  - sie eröffnet eine **Konkurrenzsituation** zwischen gleichzeitigen Prozessen
- die Anweisung ist als **bedingter kritischer Abschnitt** auszuführen
  - dabei definiert die Wartebedingung ein **Prädikat** über die im kritischen Abschnitt von den Prozessen gemeinsam verwendeten Daten
  - Auswertung und Folgerung erfolgen im kritischen Abschnitt, der während der Wartezeit des Prozesses für andere Prozesse aber frei sein muss [7]
- alternative und für das Pufferbeispiel besser geeignete Lösung:
  - **allgemeiner Semaphor**, der die Anzahl freier/belegter Einträge mitzählt

Fortschrittsgarantien

vgl. [4, 5]

- Aussagen zur **Lebendigkeit** (*liveliness*) nichtsequentieller Programme

  behinderungsfrei (*obstruction-free*)
  - ein einzelner, in Isolation stattfindender Prozess wird seine Aktion in begrenzter Anzahl von Schritten beenden
  - der Prozess findet isoliert statt, sofern alle anderen Prozesse, die ihn behindern könnten, zurückgestellt sind

  sperrfrei (*lock-free*), umfasst Behinderungsfreiheit
  - jeder Schritt eines Prozesses trägt dazu bei, dass die Ausführung des nichtsequentiellen Programms insgesamt voranschreitet
  - systemweiter Fortschritt ist garantiert, jedoch können einzelne Prozesse der **Aushungerung** (*starvation*) unterliegen

  wartefrei (*wait-free*), umfasst Sperrfreiheit
  - die Anzahl der zur Beendigung einer Aktion auszuführenden Schritte ist konstant oder zumindest nach oben begrenzt
  - garantiert systemweiten Fortschritt und ist frei von Aushungerung
- Merkmale von Verfahren für die **nichtblockierende Synchronisation**
  - Eigenschaften der Algorithmen, unabhängig von Umgebungswissen

Gliederung

Resümee

- **Nebenläufigkeit** setzt voneinander unabhängige Prozesse voraus
  - bezeichnet das Verhältnis von nicht kausal abhängigen Ereignissen
  - schränkt sich ein aus Gründen von Daten- oder Zeitabhängigkeit
- gleichzeitige abhängige Prozesse implizieren **Koordinierung**
  - nämlich der Kooperation und Konkurrenz zwischen Prozessen
  - durch analytische (implizite) oder konstruktive (explizite) Techniken
- **Synchronisation** zeigt einen großen Facettenreichtum
  - klassifiziert nach der jeweiligen Auswirkung auf beteiligte Prozesse:
    - einseitig oder mehrseitig
    - unterdrückend, blockierend oder nichtblockierend
    - behinderungs-, sperr- oder wartefrei
  - klassifiziert nach der Ebene im Rechensystem $\rightsquigarrow$ nächsten Vorlesungen:

Hochsprachenebene	Bedingungsvariable, Monitor
Maschinenprogrammebene	Verdrängungssteuerung, Semaphor
Befehlssatzebene	Schlossvariable, Spezialbefehle (CPU)

- Aussagen zur „Lebendigkeit“ nichtsequentieller Programme leiten sich aus den **Fortschrittsgarantien** der Synchronisationsverfahren ab

Literaturverzeichnis I

[1] Corbató, F. J. ; Merwin-Daggett, M. ; Dalex, R. C.:
An Experimental Time-Sharing System.
In: *Proceedings of the AIEE-IRE '62 Spring Joint Computer Conference*, ACM, 1962, S. 335–344

[2] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

[3] Hansen, P. B.:
Concurrent Processes.
In: *Operating System Principles*.
Englewood Cliffs, N.J., USA : Prentice-Hall, Inc., 1973. –
ISBN 0–13–637843–9, Kapitel 3, S. 55–131

[4] Herlihy, M. :
Wait-Free Synchronization.
In: *ACM Transactions on Programming Languages and Systems* 11 (1991), Jan., Nr. 1, S. 124–149

Literaturverzeichnis II

[5] Herlihy, M. ; Luchangco, V. ; Moir, M. :
Obstruction-Free Synchronization: Double-Ended Queues as an Example.
In: *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003), May 19–22, 2003, Providence, Rhode Island, USA*, IEEE Computer Society, 2003, S. 522–529

[6] Herrtwich, R. G. ; Hommel, G. :
Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme.
Springer-Verlag, 1989. –
ISBN 3–540–51701–4

[7] Hoare, C. A. R.:
Towards a Theory of Parallel Programming.
In: Hoare, C. A. R. (Hrsg.) ; Perrot, R. H. (Hrsg.): *Operating System Techniques*. New York, NY : Academic Press, Inc., Aug. – Sept. 1971 (Proceedings of a Seminar at Queen's University, Belfast, Northern Ireland), S. 61–71

[8] Kleinöder, J. ; Schröder-Preikschat, W. :
Prozesse.
In: Lehrstuhl Informatik 4 (Hrsg.): *Systemprogrammierung*.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 6.1

Literaturverzeichnis III

[9] Löhr, K.-P. :
Nichtsequentielle Programmierung.
In: Institut für Informatik (Hrsg.): *Algorithmen und Programmierung IV*.
Freie Universität Berlin, 2006 (Vorlesungsfolien)

Logische Synchronisation zählender Semaphor (*counting semaphore*)

```
1  semaphore free = 64, data = 0;
2
3  char buffer[64];
4  unsigned in = 0, out = 0;
5
6  void put(char item) {
7    P(&free);    /* block iff buffer is full: free = 0 */
8    buffer[FAA(&in, 1) % 64] = item;
9    V(&data);    /* signal data availability */
10 }
11
12 char get() {
13   P(&data);    /* block iff buffer is empty: data = 0 */
14   char item = buffer[FAA(&out, 1) % 64];
15   V(&free);    /* signal buffer-place availability */
16
17   return item;
18 }
```

- Prinzip „**begrenzter Puffer**“ (*bounded buffer*), siehe auch [8, S. 30–33]

Wechselseitiger Ausschluss beim Zählen

fetch and add

- **binärer Semaphor**, Lösung auf Maschinenprogrammebene: ☹

```
1  int FAA(int_t *ref, int val) {
2    P(&ref->mutex);
3    int aux = ref->value;
4    ref->value += val;
5    V(&ref->mutex);
6
7    return aux;
8  }
```

```
9  typedef struct {
10   int value;
11   semaphore_t mutex;
12 } int_t;

13 int_t in = {0,1}, out = {0,1};
```

- **atomarer Spezialbefehl**, Lösung auf Befelssatzebene: ☺

```
14 inline int FAA(int *ref, int val) {
15   int aux = val;
16
17   asm volatile ("xaddl %0, %1"
18     : "=g" (aux), "=m" (*ref) : "0" (aux), "m" (*ref)
19     : "memory", "cc");
20
21   return aux;
22 }
```

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – X.2 Prozesssynchronisation: Monitore

Wolfgang Schröder-Preikschat

16. November 2021

Agenda

Einführung

Monitor
- Eigenschaften
- Architektur

Bedingungsvariable
- Definition
- Operationen
- Signalisierung

Beispiel
- Daten(ring)puffer

Zusammenfassung

Gliederung

Lehrstoff

- Auseinandersetzung mit Begrifflichkeiten bezüglich “a shared variable and the set of meaningful operations on it” [5, p. 121]:
  - monitor
    - ursprünglich auch **kritischer Bereich** (*critical region*, [4, 5])
    - assoziiert Prozeduren mit einer gemeinsamen Variablen
      - versetzt einen Kompilierer in die Lage:
        (a) die für die Variable definierten Operationen zu prüfen
        (b) den wechselseitigen Ausschluss der Operationen zu erzwingen
  - condition
    - eine **Variable** für die gilt: “it does not have any stored value accessible to the program” [8, p. 550]
    - dient der Anzeige und Steuerung eines Wartezustands
      - für den jeweiligen Prozess innerhalb des Monitors
- die Funktionsweise des Monitors als ein **Mittel zur Synchronisation** verstehen, unabhängig linguistischer Merkmale
  - Erklärung verschiedener Stile: Hansen, Hoare, Concurrent Pascal, Mesa
  - diesbezügliche schematische Darstellung von Implementierungsvarianten
- jedoch schon die **problemorientierte Programmiersprachenebene** als Verortung dieser Konzepte im Rechensystem identifizieren (s. [11])

Instrument zur Überwachung

Hinweis (Monitor [5, S. 121])

The purpose of a monitor is to control the scheduling of resources among individual processes according to a certain policy.

- ein Konzept kennenlernen, das als **programmiersprachlicher Ansatz** einzustufen ist, aber gleichsam darüber hinaus geht
  - ein klassenähnlicher synchronisierter Datentyp [5, 8, 12]
  - inspiriert durch SIMULA 67 [3, 2]
  - zuerst implementiert in Concurrent Pascal [6]
  - danach realisiert in unterschiedlichen Ausführungen [1, 7]
- die Technik ist grundlegend für die Systemprogrammierung und den systemnahen Betrieb von gekoppelten Prozessen
  - mit dem Monitorkonzept ist auch eine **Programmierkonvention** gemeint und nicht immer bloß ein **Programmiersprachenkonstrukt**
  - diese Konvention ist in jeder Programmiersprache nutzbar, jedoch nicht in jeder integriert und nicht von jedem Kompilierer umgesetzt

Gliederung

Synchronisierter abstrakter Datentyp: Monitor

- in den grundlegenden Eigenschaften ein **abstrakter Datentyp** [13], dessen Zugriffsoperationen implizit synchronisiert sind [5, 8]

  mehrseitige Synchronisation an der Monitorschnittstelle
  - **wechselseitiger Ausschluss** der Ausführung exportierter Prozeduren
    - realisiert mittels **Schlossvariablen** oder vorzugsweise **Semaphore**

  einseitige Synchronisation innerhalb des Monitors
  - logische Synchronisation mittels **Bedingungsvariable**

    wait blockiert einen Prozess auf das Eintreten eines Ereignisses und gibt den Monitor implizit wieder frei

    signal zeigt das Eintreten eines Ereignisses an und deblockiert, je nach Typ des Monitors, einen oder alle darauf blockierte Prozesse
  - bei **Ereigniseintritt** löst der betreffende Prozess ein Signal aus und zeigt damit die **Aufhebung einer Wartebedingung** an
- ein sprachgestützer Ansatz, bei dem der Übersetzer automatisch die Synchronisationsbefehle generiert
  - Concurrent Pascal, PL/I, Mesa, . . . , Java

Monitor ≡ (eine auf ein Modul bezogene) Klasse

- ein Monitor ist einer **Klasse** [2] ähnlich und er besitzt alle Merkmale, die auch ein **Modul** [14] mitbringt

  Kapselung (*encapsulation*)
  - von mehreren Prozessen gemeinsam bearbeitete Daten müssen, analog zu Modulen, in Monitoren organisiert vorliegen
  - die Programmstruktur macht kritische Abschnitte explizit sichtbar
    - inkl. zulässige (an zentraler Stelle definierte) Zugriffsfunktionen

  Datenabstraktion (*information hiding*)
  - wie ein Modul, so kapselt auch ein Monitor für mehrere Prozeduren Wissen über gemeinsame Daten
  - Auswirkungen lokaler Programmänderungen bleiben begrenzt

  Bauplan (*blueprint*)
  - wie eine Klasse, so beschreibt ein Monitor für mehrere Exemplare seines Typs den **Zustand** und das **Verhalten**
  - er ist eine gemeinsam benutzte Klasse (*shared class*, [5])

Klassenkonzept mit Synchronisationssemantik

Monitor $\equiv$ *implizit synchronisierte Klasse*

- **Monitorprozeduren** (*monitor procedures*)
  - schließen sich bei konkurrierenden Zugriffen durch mehrere Prozesse in ihrer Ausführung gegenseitig aus
    - der erfolgreiche Prozeduraufruf sperrt den Monitor
    - bei Prozedurrückkehr wird der Monitor wieder entsperrt
  - repräsentieren per Definition kritische Abschnitte, deren Integrität vom Kompilierer garantiert wird
    - die „Klammerung“ kritischer Abschnitte erfolgt automatisch
    - der Kompilierer erzeugt die dafür notwendigen Steueranweisungen

- **Synchronisationsanweisungen** $\leadsto$ **Bedingungsvariable**
  - sind Querschnittsbelang eines Monitors und nicht des gesamten nichtsequentiellen Programms
  - sie liegen nicht quer über die ganze Software verstreut vor

Wiederverwendbare unteilbare Ressource

- ein Monitor ist Bauplan für ein **Softwarebetriebsmittel**, mit dem verschiedene Sorten von Warteschlangen verbunden sind
  - in der **Monitorwarteschlange** befinden sich Prozessexemplare, die den Eintritt in den Monitor erwarten
    - sie warten nur, wenn der Monitor im Moment des Eintrittsversuchs bereits von einem anderen Prozess belegt war
    - erst bei Monitorfreigabe wird einer dieser Prozesse zur Auswahl bereitgestellt
  - die **Ereigniswarteschlange** enthält Prozessexemplare, die im Monitor die Aufhebung einer Wartebedingung erwarten
    - sie warten nur, wenn ein mit einer Bedingungsvariable verknüpftes Ereignis noch nicht eingetreten ist
    - beachte: dieses Ereignis bildet ein **konsumierbares Betriebsmittel** [9, S. 14]
- ein Prozess wartet jedoch stets außerhalb des Monitors, das heißt, er belegt den Monitor während seiner Wartezeit nicht
  - ansonsten könnte kein anderer Prozess den Monitor betreten und somit die Wartebedingung für einen Prozess aufheben
  - mit Aufhebung der Wartebedingung eines Prozesses, wird diesem der **Wiedereintritt** in den Monitor ermöglicht

Monitor mit blockierenden Bedingungsvariablen

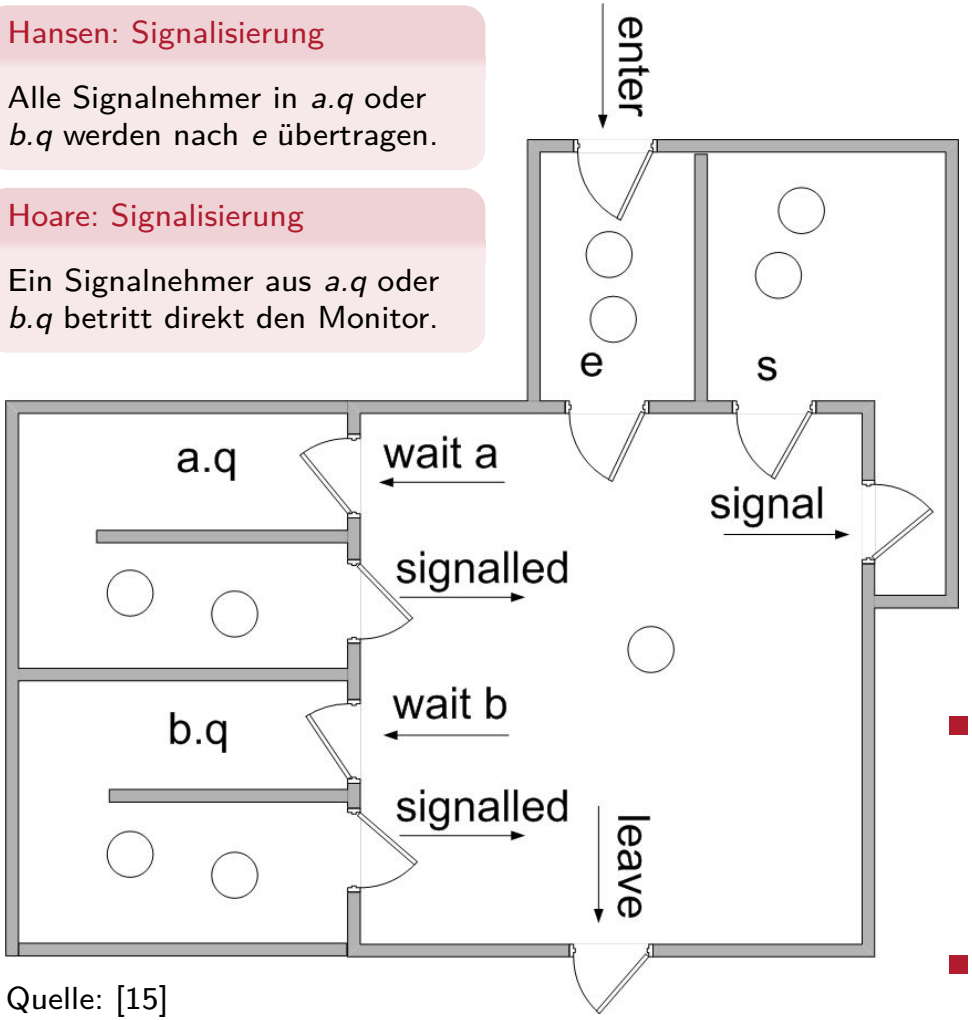
- nach Hansen [5] und Hoare [8], letzterer hier im Bild skizziert:

Hansen: Signalisierung

Alle Signalnehmer in *a.q* oder *b.q* werden nach *e* übertragen.

Hoare: Signalisierung

Ein Signalnehmer aus *a.q* oder *b.q* betritt direkt den Monitor.



Quelle: [15]

Monitorwarteschlangen

- e der Zutrittsanforderer
- s der Signalgeber: optional
  - Vorzugswarteliste oder vereint mit *e*

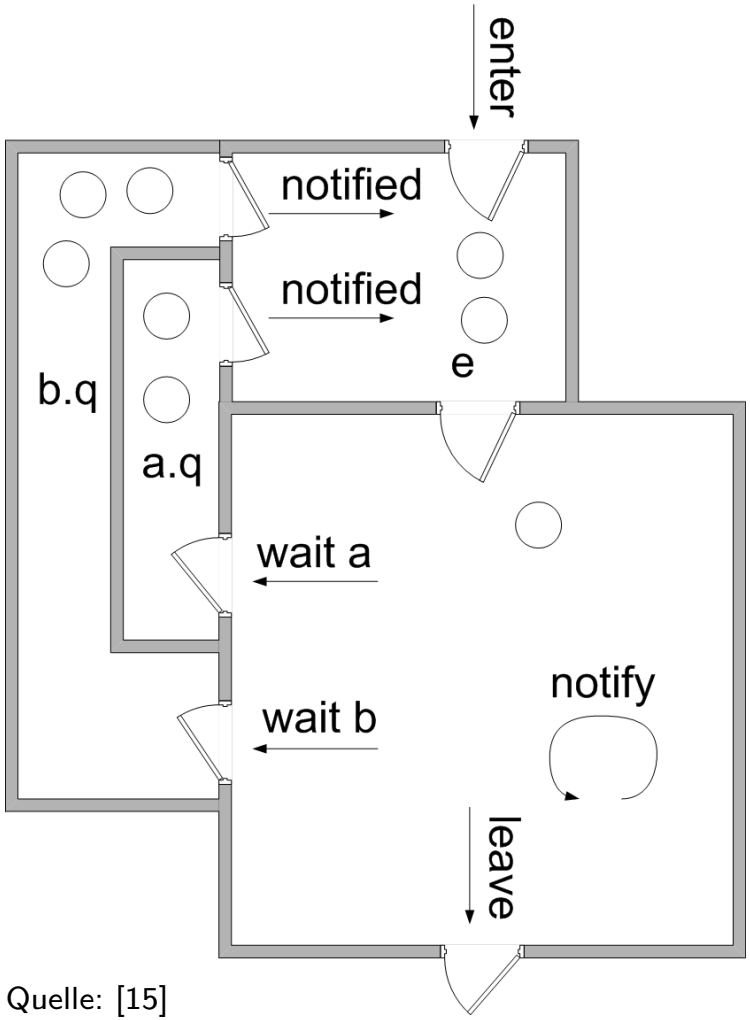
Ereigniswarteschlangen

- a.q für Bedingungsvariable *a*
- b.q für Bedingungsvariable *b*

- **Signalgeber blockieren**
  - warten außerhalb
  - verlassen den Monitor
- **Wiedereintritt** falls *signal* nicht letzte Operation

Monitor mit nichtblockierenden Bedingungsvariablen

- in Mesa [12]:



Quelle: [15]

Monitorwarteschlange

e der Zutrittsanforderer und der signalisierten Prozesse

Ereigniswarteschlangen

a.q für Bedingungsvariable *a*

b.q für Bedingungsvariable *b*

- Signalgeber fahren fort
  - „Sammelaufruf“ möglich
  - $n > 1$ Ereignisse signalisierbar
- Signalnehmer starten erst nach Monitorfreigabe (*leave*)

Monitorvergleich: Hansen, Hoare, Mesa

- Ausgangspunkt für die Verschiedenheit der Monitorkonzepte ist die **Semantik der Bedingungsvariablen**:

  blockierend
  - gibt dem Signalnehmer Vorrang

  nichtblockierend
  - gibt dem Signalgeber Vorrang

- Folge davon ist eine unterschiedliche **Semantik der Signalisierung** für die betrachteten Monitorarten:

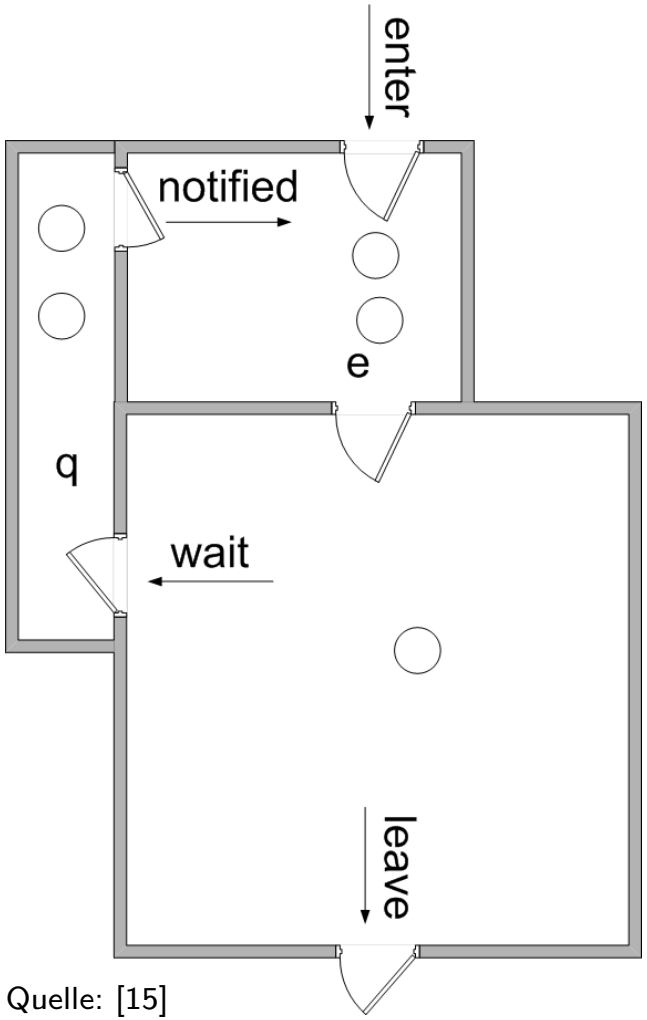
  Hansen
  - verwendet blockierende Bedingungsvariablen
  - Signalisierung lässt den Signalgeber den Monitor verlassen, nachdem er alle Signalnehmer „bereit“ gesetzt hat

  Hoare
  - verwendet blockierende Bedingungsvariablen
  - Signalisierung lässt den Signalgeber den Monitor verlassen und genau einen Signalnehmer fortfahren $\rightsquigarrow$ *atomare Aktion*

  Mesa
  - verwendet nichtblockierende Bedingungsvariablen
  - Signalisierung lässt den Signalgeber im Monitor fortfahren, nachdem er einen oder alle Signalnehmer „bereit“ gesetzt hat

Monitor mit impliziten Bedingungsvariablen

- in Java, C#:



Quelle: [15]

Monitorwarteschlange

e der Zutrittsanforderer und der signalisierten Prozesse

Ereigniswarteschlange

q für den gesamten Monitor

- Objekte sind zwar keine Monitore, können aber als solche verwendet werden
- `synchronized`-Anweisung an Methoden oder Basisblöcken
- Signalisierung wie bei Mesa (S. 12)

Gliederung

Variable ohne Inhalt. . .

Hinweis (Bedingungsvariable (*condition variable* [8, S. 550]))

Note that a condition "variable" is neither true nor false; indeed, it does not have any stored value accessible to the program.

- fundamentale Primitive [8] zur **Bedingungssynchronisation** wobei die Operationen folgende intrinsische Eigenschaften haben:
  - `signal`
    - zeigt ein Ereignis an
    - ist wirkungslos, sollte kein Prozess auf das Ereignis warten
    - nimmt genau einen wartenden Prozess sofort wieder auf
  - `wait`
    - setzt den Prozess bis zur Anzeige eines Ereignisses aus
    - gibt den Monitor bis zur Wiederaufnahme implizit frei
- auch **Ereignisvariable** (*event variable* [4]), mit den beiden zu oben korrespondierenden Operationen `cause` und `await`
  - alle dasselbe Ereignis erwartende Prozesse werden durch `cause` befreit[1]
  - wobei vorrangige Prozesse Vorrang beim Eintritt in den Monitor erhalten

[1] Wobei [4] Aussetzung/Fortsetzung des signalisierenden Prozesses offen lässt.

Aufhebung einer Wartebedingung erwarten

wait

- *when* `(`*condition*`) wait(`*event*`)` mit *when* gleich:

  `if`
  - Ereignisauslösung bis Prozesseinlastung ist **unteilbare Aktion**
  - dazwischen ist die Wartebedingung nicht erneut erfüllbar
  - d.h., kein Prozess kann zwischenzeitlich in den Monitor eintreten

  `while`
  - sonst

- die Aktion, innerhalb eines Monitors zu warten, muss zwingend die **Monitorfreigabe** zur Folge haben
  - andere Prozesse wären sonst am Monitoreintritt gehindert
  - als Folge könnte die Wartebedingung nie aufgehoben werden
  - schlafende Prozesse würden nie mehr erwachen $\rightsquigarrow$ **Verklemmung**

- da in der Wartezeit ein anderer Prozess den Monitor betreten muss, ist explizit für **Konsistenz** der Monitordaten zu sorgen
  - ein oder mehrere andere Prozesse heben die Wartebedingung auf
  - als Folge werden sich Daten- bzw. Zustandsänderungen ergeben
  - vor Eintritt in die Wartephase muss der Datenzustand konsistent sein

Aufhebung einer Wartebedingung anzeigen

signal

- *cancel*(*condition*) ... signal(*event*) wobei *cancel* die Aktion zur Aufhebung der Wartebedingung (*condition*) repräsentiert
  - diese Bedingung ist ein **Prädikat** über den internen Monitorzustand
- Zweck der Signalisierung ist es, eine bestehende **Prozessblockade** in Bezug auf eine Wartebedingung zu beenden
  - warten Prozesse, muss die Operation für **Prozessfortschritt** sorgen
    - mindestens einer der das Ereignis erwartenden Prozesse wird deblockiert
    - höchstens ein Prozess rechnet nach der Operation im Monitor weiter
  - erwartet kein Prozess das Ereignis, ist die Operation wirkungslos
    - d.h., Signale dürfen in Bedingungsvariablen nicht gespeichert werden
- die Verfahren dazu sind teils von sehr unterschiedlicher Semantik und wirken sich auf die Programmierung auf
  - das betrifft etwa die Anzahl der deblockierten Prozesse
    - alle auf dasselbe Ereignis wartenden oder nur einer: `while` vs. `if` (S. 17)
    - falsche Signalisierungen werden toleriert (`while`) oder nicht (`if`)
  - bzw. ob **Besitzwechsel** oder **Besitzwahrung** des Monitors stattfindet

Besitzwechsel

signal and [urgent] wait

- `signal` befreit einen oder mehrere Prozesse und sorgt dafür, dass der aktuelle Prozess den Monitor abgibt
  - alle das Ereignis erwartenden Prozesse befreien $\mapsto$ Hansen [4, S. 576]
    - alle Prozesse aus der Ereignis- in die Monitorwarteschlange bewegen
    - bei Freigabe alle n Prozesse (Monitorwarteschlange) „bereit“ setzen
    - $n-1$ Prozesse reihen sich erneut in die Monitorwarteschlange ein

    $\hookrightarrow$ Neuauswertung der Wartebedingung erforderlich (S. 17, `while`)

    $\hookrightarrow$ falsche Signalisierungen (S. 29) werden toleriert
  - höchstens einen das Ereignis erwartenden Prozess befreien $\mapsto$ Hoare [8]
    - einen einzigen Prozess der Ereigniswarteschlange entnehmen und fortsetzen
    - den signalisierenden Prozess in die Monitorwarteschlange eintragen
    - direkt vom signalisierenden zum signalisierten Prozess wechseln

    $\hookrightarrow$ Neuauswertung der Wartebedingung entfällt (S. 17, `if`)

    $\hookrightarrow$ falsche Signalisierungen (S. 29) werden nicht toleriert
- der signalisierende Prozess bewirbt sich erneut um den Monitor oder wird fortgesetzt, wenn der signalisierte Prozess den Monitor verlässt
  - letzteres (*urgent*, Hoare) greift auf eine **Vorzugswarteschlange** zurück

Besitzwahrung

signal and continue

- `signal` befreit die auf das Ereignis wartenden Prozesse, setzt jedoch den aktuellen Prozess im Monitor fort
  - einen oder alle das Ereignis erwartenden Prozesse befreien $\mapsto$ Mesa [12]
    - Prozess(e) aus der Ereignis- in die Monitorwarteschlange bewegen
    - bei Freigabe $n \geq 1$ Prozesse (Monitorwarteschlange) „bereit“ setzen

    $\hookrightarrow$ Neuauswertung der Wartebedingung erforderlich (S. 17, `while`)

    $\hookrightarrow$ falsche Signalisierungen (S. 29) werden toleriert
- genau einen Prozess auszuwählen (Mesa, Hoare) birgt die Gefahr von **Prioritätsverletzung** [12]
  - betrifft die Entnahme eines Prozesses aus der Ereigniswarteschlange
  - **Interferenz** mit der Prozesseinplanung ist vorzubeugen/zu vermeiden
- mehrere oder gar alle Prozesse auszuwählen (Mesa, Hansen) birgt das Risiko der erneuten Erfüllung der Wartebedingung
  - nach Fortsetzung des ersten befreiten Prozesses: erfüllt durch ihn selbst oder durch andere Prozesse, die zwischenzeitlich im Monitor waren
  - da ein Prozess nicht weiß, ob er als erster befreit wurde, muss jeder die Wartebedingung erneut auswerten

Gliederung

Wiederverwendbares Softwarebetriebsmittel vgl. [9, S. 24ff]

```
1  template<typename T, unsigned N=64, monitor>
2  class Buffer {
3      T buffer[N];          // N should be power of two
4      unsigned in, out;
5      condition data, free;
6
7  atomic:                   // public, mutual exclusive methods
8      Buffer() { in = out = 0; }
9
10     void put(T item) {
11         when (((in + 1) % N) == out) free.wait();
12         buffer[in++ % N] = item;
13         data.signal();
14     }
15
16     T get() {
17         when (out == in) data.wait();
18         T item = buffer[out++ % N];
19         free.signal();
20         return item;
21     }
22 };
```

when (*condition*) wait

Diese bedingte Anweisung wird innerhalb kritischer Abschnitte ausgeführt, sie geschieht damit „**atomar**“. So wird einem möglichen *lost wake-up* vorgebeugt.

- gedachtes Concurrent C++ (s. S. 28)

Gliederung

Resümee

- ein Monitor ist ein **ADT** mit impliziten Synchronisationseigenschaften
  - mehrseitige Synchronisation von Monitorprozeduren
  - einseitige Synchronisation durch Bedingungsvariablen
- seine **Architektur** lässt verschiedene Ausführungsarten zu
  - Monitor mit beid- oder einseitig blockierenden Bedingungsvariablen
- Unterschiede liegen vor allem in der **Semantik der Signalisierung**:
  - wirkt blockierend (Hansen, Hoare) oder nichtblockierend (Mesa, Java) für den ein Ereignis signalisierenden Prozess
  - deblockiert einen (Hoare, Mesa, Java) oder alle (Hansen, Mesa, Java) auf ein Ereignis wartende Prozesse
  - die Wartebedingung für den jeweils signalisierten Prozess ist garantiert (Hoare) oder nicht garantiert (Hansen, Mesa, Java) aufgehoben
  - erfordert (Hansen, Mesa, Java) oder erfordert nicht (Hoare) die erneute Auswertung der Wartebedingung nach Wiederaufnahme
  - ist falschen Signalisierungen gegenüber tolerant (Hansen, Mesa, Java) oder intolerant (Hoare)
- Java-Objekte sind keine Monitore, wohl aber als solche verwendbar
  - so bietet Java keine Monitorobjekte, entgegen der Informatikfolklore. . .

Literaturverzeichnis I

[1] Buhr, P. A. ; Fortier, M. :
Monitor Classification.
In: *ACM Computing Surveys* 27 (1995), März, Nr. 1, S. 63–107

[2] Dahl, O.-J. ; Myhrhaug, B. ; Nygaard, K. :
SIMULA Information: Common Base Language / Norwegian Computing Center.
1970 (S-22). –
Forschungsbericht

[3] Dahl, O.-J. ; Nygaard, K. :
SIMULA—An ALGOL-Based Simulation Language.
In: *Communications of the ACM* 9 (1966), Sept., Nr. 9, S. 671–678

[4] Hansen, P. B.:
Structured Multiprogramming.
In: *Communications of the ACM* 15 (1972), Jul., Nr. 7, S. 574–578

[5] Hansen, P. B.:
Operating System Principles.
Englewood Cliffs, N.J., USA : Prentice-Hall, Inc., 1973. –
ISBN 0–13–637843–9

Literaturverzeichnis II

[6] Hansen, P. B.:
The Programming Language Concurrent Pascal.
In: *IEEE Transactions on Software Engineering* SE-I (1975), Jun., Nr. 2, S. 199–207

[7] Hansen, P. B.:
Monitors and Concurrent Pascal: A Personal History.
In: Bergin, Jr., T. (Hrsg.) ; Gibson, Jr., R. G. (Hrsg.): *History of Programming Languages—II*.
New York, NY, USA : ACM, 1996. –
ISBN 0–201–89502–1, S. 121–172

[8] Hoare, C. A. R.:
Monitors: An Operating System Structuring Concept.
In: *Communications of the ACM* 17 (1974), Okt., Nr. 10, S. 549–557

[9] Kleinöder, J. ; Schröder-Preikschat, W. :
Nichtsequentialität.
In: [10], Kapitel 10.1

[10] Kleinöder, J. ; Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)

Literaturverzeichnis III

[11] Kleinöder, J. ; Schröder-Preikschat, W. :
Virtuelle Maschinen.
In: [10], Kapitel 5.1

[12] Lampson, B. W. ; Redell, D. D.:
Experiences with Processes and Monitors in Mesa.
In: *Communications of the ACM* 23 (1980), Febr., Nr. 2, S. 105–117

[13] Liskov, B. J. H. ; Zilles, S. N.:
Programming with Abstract Data Types.
In: Leavenworth, B. (Hrsg.): *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages* Bd. 9.
New York, NY, USA : ACM, Apr. 1974 (ACM SIGPLAN Notices 4), S. 50–59

[14] Parnas, D. L.:
On the Criteria to be used in Decomposing Systems into Modules.
In: *Communications of the ACM* 15 (1972), Dez., Nr. 12, S. 1053–1058

[15] Wikipedia:
Monitor (synchronization).
`http://en.wikipedia.org/wiki/Monitor_(synchronization)`, Dez. 2010

Fiktion: *Concurrent C++*

- hier als **Klassenvorlage** (*class template*) ausgelegt, um in einfacher Weise eine generische Implementierung des Datenpuffers zu zeigen:

  `monitor`
  - Vorlagenparameter zur Deklaration der Klasse als Monitor
  - Verfeinerung durch `monitor=`*name*, wobei *name* sein kann:
    - i *signal* bzw. *signal continue* (vorgegeben) oder
    - ii *signal wait*, *signal urgent wait*, *signal return*
  - Exemplare dieser Klasse sind **Monitorobjekte**

  `condition`
  - Deklaration von einer oder mehreren Bedingungsvariablen
  - Voraussetzung für die Operatoren `wait` und `signal`

  `when`
  - Kontrollstruktur zum bedingten Warten (`wait`) im Monitor
  - je nach Verfeinerung des Monitors entspricht diese implizit:
    - `while` für *signal [continue]* und *signal return*
    - `if` sonst, d.h., für *signal wait* und *signal urgent wait*
  - jedoch kann jede Kontrollstruktur explizit gemacht werden

- aber auch ohne „Spracherweiterung“ lässt sich das Monitorkonzept integrieren, wenngleich auch mit Abstrichen ⇝ vgl. S. 30
  - Spezialisierung durch Verberung `class Buffer : private Monitor`
  - `Condition` als Klasse mit `wait()`/`signal()`-Methoden, `when` umseitig

Wiederaufnahme nach Signalisierung

vgl. S. 17

```
1 #ifdef __MONITOR_SIGNAL_AND_RESUME__
2 #define when if      /* waiting condition nullified */
3 #else
4 #define when while   /* re-evaluate waiting condition */
5 #endif
```

- die **Neuauswertung** der Wartebedingung kann entfallen, wenn:
  - i `signal` höchstens einen Prozess auswählt und
  - ii der ausgewählte Prozess unteilbar wieder aufgenommen wird
- in dem Fall, wird eine **falsche Signalisierung** jedoch nicht toleriert
  - dabei handelt es sich um einen **Programmierfehler**, d.h., `signal` kommt zur Ausführung, obwohl die Wartebedingung nicht aufgehoben ist
  - `signal` wurde entweder auf die richtige, dann aber zu unrecht, oder die falsche Bedingungsvariable appliziert
- Neuauswertung der Wartebedingung toleriert falsche Signalisierungen und ist notwendig, wenn mehrere Prozesse signalisiert werden
  - generelle Lösung für alle Signalisierungsvarianten...

Monitorkonzept durch Vererbung

vgl. S. 28

```
1  template<typename T, unsigned N=64>
2  class Buffer : private Monitor {
3      T buffer[N];          // N should be power of two
4      unsigned in, out;
5      Condition data, free;
6  public:
7      Buffer() {
8          enter();          // critical section
9          in = out = 0;
10         leave();
11     }
12
13     void put(T item) {
14         enter();          // critical section
15         when (((in + 1) % N) == out) free.wait(*this);
16         buffer[in++ % N] = item;
17         data.signal();
18         leave();
19     }
20
21     T get() {
22         enter();          // critical section
23         when (out == in) data.wait(*this);
24         T item = buffer[out++ % N];
25         free.signal();
26         leave();
27
28         return item;
29     }
30 };
```

```
31  class Monitor {
32      /* ... */
33  public:
34      void enter();
35      void leave();
36  };
```

```
37  class Condition {
38      /* ... */
39  public:
40      void signal();
41      void wait(Monitor&);
42  };
```

- `wait` muss den Monitor kennen, der zu entsperren und anzufordern ist

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – X.3 Prozesssynchronisation: Semaphore und Sperren

Wolfgang Schröder-Preikschat

23. November 2021

Agenda

Einführung

Semaphor
- Definition
- Anwendung
- Implementierung
- Ablaufunterbrechung

Mutex
- Abgrenzung
- Implementierung

Sperre
- Grundsätzliches
- Varianten

Zusammenfassung

Gliederung

Lehrstoff

- *das* Konzept der **Maschinenprogrammebene** (s. [11]) kennenlernen, mit dem die Synchronisation gleichzeitiger Prozesse erreicht wird
  - binärer, allgemeiner bzw. ausschließender, zählender, privater Semaphor
  - zwei Varianten für zwei **Synchronisationsmuster**: ein- vs. mehrseitig
- die Implementierung eines Semaphors durchleuchten und sich damit auseinandersetzen, **wettlaufkritische Aktionen** zu bewältigen
  - ablaufinvariante bzw. unteilbare/atomare Semaphorprimitiven
  - beispielhaft diese als kritischen Abschnitt begreifen: Standardsicht
  - Ereignisvariable zur Bedingungssynchronisation in diesem Abschnitt
- den **Mutex** erklären als minimale (funktionale) Erweiterung eines binären Semaphors zum *autorisierten* wechselseitigen Ausschluss
  - genauer: ausschließender Semaphor mit Kontrolle der Eigentümerschaft
  - Eigentumslosigkeit für Semaphore als Merkmal nicht als Makel verstehen
- schließlich **Sperren** behandeln, um Atomarität der Primitiven eines Semaphors physisch gewährleisten zu können
  - Unterbrechungs-, Fortsetzungs- und Verdrängungssperre
  - d.h., Lösungen für (einkernige) Uniprozessorsysteme: **Pseudoparallelität**

Gliederung

Semaphor

vgl. [8, S. 22]

- spezielle **ganzzahlige Variable** [4, p. 345] mit zwei Operationen [2]:

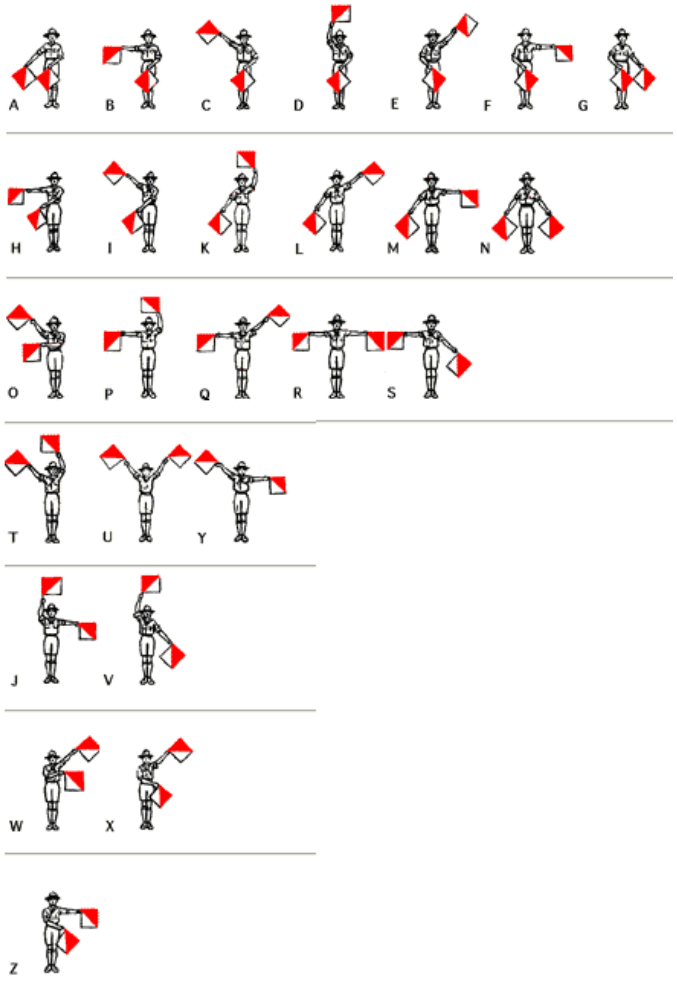
  P Abk. für (Hol.) ***prolaag***; alias *down*, *wait* oder *acquire*
  - verringert[1] den Wert des Semaphors s um 1:
    - i genau dann, wenn der resultierende Wert nichtnegativ wäre [3, p. 29]
    - ii logisch uneingeschränkt [4, p. 345]
  - ist oder war der Wert vor dieser Aktion 0, blockiert der Prozess
    - er kommt auf eine mit dem Semaphor assoziierte Warteliste

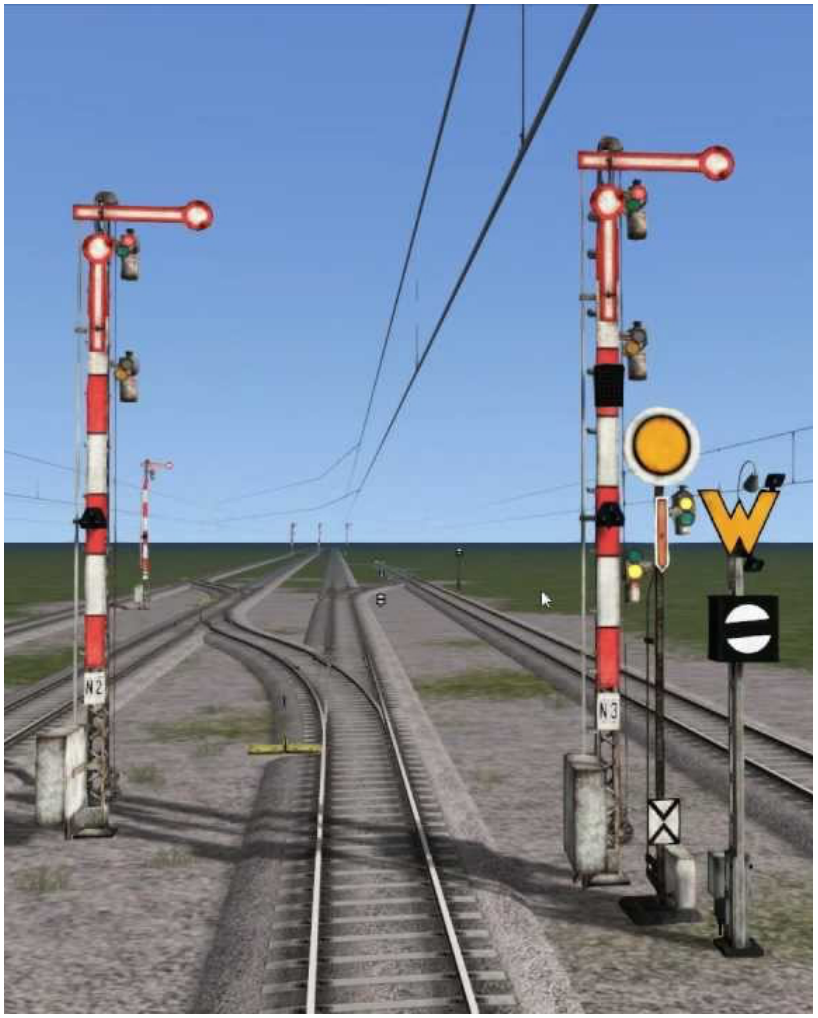
  V Abk. für (Hol.) ***verhoog***; alias *up*, *signal* oder *release*
  - erhöht[1] den Wert des Semaphors s um 1
  - ein ggf. am Semaphor blockierter Prozess wird wieder bereitgestellt
    - welcher Prozess von der Warteliste genommen wird, ist nicht spezifiziert
  - beide Primitiven sind logisch oder physisch **unteilbare Operationen**

- ursprünglich definiert als **binärer Semaphor** ($s = [0, 1]$), generalisiert als **allgemeiner Semaphor** ($s = [n, m], m > 0$ und $n \leq m$)

[1] Nicht zwingend durch Subtraktion oder Addition im arithmetischen Sinn.

Instrument zur Kommunikation und Koordination





Signalisierender Semaphor

- **einseitige Synchronisation** (Beispielvorlage, vgl. auch [7, S. 33]):

```
1  typedef struct buffer {
2    char ring[64];              /* buffer memory: ring buffer */
3    int_t in, out;              /* initial: {0,1}, {0,1} */
4    semaphore_t free, data;     /* initial: 64, 0 */
5  } buffer_t;
6  void put(buffer_t *pool, char item) {
7    P(&pool->free);    /* block iff buffer is full: free = 0 */
8    pool->ring[FAA(&pool->in, 1) % 64] = item;
9    V(&pool->data);    /* signal data availability */
10 }
11 char get(buffer_t *pool) {
12   P(&pool->data);    /* block iff buffer is empty: data = 0 */
13   char item = pool->ring[FAA(&pool->out, 1) % 64];
14   V(&pool->free);    /* signal buffer-place availability */
15   return item;
16 }
```

- ist der Puffer voll, wartet der Produzent in Z. 7 auf den Konsumenten — der in Z. 12 auf den Produzenten wartet, wenn der Puffer leer ist
- diesbezügliche Signalisierungen (Z. 9 und 14) setzen die Prozesse fort

Ausschließender Semaphor

- **mehrseitige Synchronisation** (Beispielvorlage, vgl. auch [7, S. 34]):
  - im Anwendungszenario (S. 8) können Produzenten und Konsumenten gleichzeitig auf die Indexvariablen (`in`, `out`) zugreifen
    - und zwar wann immer der Puffer nicht voll bzw. nicht leer ist
  - die gleichzeitigen Zugriffe müssen koordiniert erfolgen, um die konsistente Werteveränderung der Indexvariablen zuzusichern
    - dazu kommt **wechselseitiger Ausschluss** (*mutual exclusion*) zum Einsatz

```
1  int FAA(int_t *ref, int val) {
2    P(&ref->mutex);
3    int aux = ref->value;
4    ref->value += val;
5    V(&ref->mutex);
6
7    return aux;
8  }
```

```
9  typedef struct {
10   int value;              /* data */
11   semaphore_t mutex; /* lock */
12 } int_t;
```

  - da die Zugriffszeitpunkte der beteiligten Prozesse unbekannt ist, ist jeder dieser Prozesse in Z. 2 (d.h., im *P*) ggf. zum Warten verurteilt

EWD[2] beim Wort genommen

- Programme für P und V bilden **kritische Abschnitte**

```
1  void ewd_prolaag(int *sema) {
2      atomic {
3          *sema -= 1;
4          if (*sema < 0)
5              await(sema);
6      }
7  }
```

```
8  void ewd_verhoog(int *sema) {
9      atomic {
10         *sema += 1;
11         if (*sema <= 0)
12             cause(sema);
13     }
14 }
```

i gleichzeitiges Ausführen von P kann mehr Prozesse passieren lassen, als es der Semaphorwert (`sema`) erlaubt — oder überhaupt keinen Prozess

ii gleichzeitiges Zählen kann Werte hinterlassen, die nicht der wirklichen Anzahl der ausgeführten Operationen (P, V) entsprechen

iii gleichzeitiges Auswerten der Wartebedingung (P) und Hochzählen (V) kann das Schlafenlegen (`await`) von Prozessen bewirken, obwohl die Wartebedingung für sie schon nicht mehr gilt („*lost wake-up*“)

[2] Edsgar Wybe Dijkstra

Optionen für die Absicherung von P und V

- **pessimistischer Ansatz**, der annimmt, dass gleichzeitige Aktionen mit demselben Semaphor wahrscheinlich sind
  - i **wechselseitiger Ausschluss** wird die Aktionen nicht überlappen lassen, weder sich selbst noch gegenseitig
    - P und V sind durch eine gemeinsame „Sperre“ pro Semaphor zu schützen
  - ii Schlafenlegen eines Prozesses in P muss implizit die **Entsperrung** des kritischen Abschnitts zur Folge haben
    - sonst wird kein V die Ausführung vollenden können
    - als Folge werden in P schlafende Prozesse niemals aufgeweckt
  - iii Aufwecken von Prozessen in V sollte bedingt erfolgen, und zwar falls wenigstens ein Prozess in P schlafengelegt wurde
  - legt die Implementierung als **Monitor** nahe — dann scheidet mehrseitige Sychronisation des Monitors durch Semaphore [6, S. 7] aber aus ☹
    - gegebenenfalls sind andere Sperrtechniken erforderlich ⇝ S. 24
- **optimistischer Ansatz**, der obige Annahme eher nicht trifft und sich **nichtblockierende Synchronisation** zu eigen macht
  - knifflig, ein Thema für das fortgeschrittene Studium [12]...

Monitor als Programmierkonvention[3]

```
1  void mps_prolaag(semaphore_t *sema) {
2      enter(&sema->lock.bolt);      /* lock critical section */
3      sema->load -= 1;              /* decrease semaphore value */
4      if (sema->load < 0)           /* resource(s) exhausted? */
5          await(&sema->lock);       /* fulfilled, wait outside */
6      leave(&sema->lock.bolt);      /* unlock critical section */
7  }
8
9  void mps_verhoog(semaphore_t *sema) {
10     enter(&sema->lock.bolt);      /* lock critical section */
11     sema->load += 1;              /* increase semaphore value */
12     if (sema->load <= 0)          /* any waiting process? */
13         cause(&sema->lock);       /* notify exactly one process */
14     leave(&sema->lock.bolt);      /* unlock critical section */
15 }
```

- wechselseitiger Ausschluss der Ausführung von „Monitorprozeduren“ sicherzustellen, ist eine **manuelle Maßnahme** geworden
  - Synchronisationsklammern (Z. 2/6 und 10/14) explizit machen
- für die erforderliche mehr- und einseitige Synchronisation ist eine dem Semaphor eigene Datenstruktur hinzuzufügen $\rightsquigarrow$ `lock`-Attribut

[3] *monitor programming style*, MPS

Semaphordatentyp als Verbund

```
1  typedef volatile struct semaphore {
2      int load;          /* # of allowed/waiting processes */
3      guard_t lock;      /* synchronisation variables */
4  } semaphore_t;
```

- der als ganze Zahl ($\mathbb{Z}$, `int`) repräsentierte Semaphorwert (s, `load`) gibt verschiedene Interpretationen:

$\mathbb{N}^*$	Anzahl der Prozesse, für die P keine Blockierung bewirkt
0	der nächste P aufrufende Prozess wird blockieren
$\mathbb{Z}^*_-$	als Betrag $\lvert s\rvert$ genommen die Anzahl der blockierten Prozesse

- zum Schutz (*guard*) von P/V sowie um über Prozesse zu wachen, die auf das Ereignis der Ausführung von V warten, dient:

```
5  typedef volatile struct guard {
6      detent_t bolt;   /* device to arrest concurrent processes */
7      event_t wait;    /* per-event waitlist of processes */
8  } guard_t;
```

- für die **Sperre** (*detent*) gibt es sehr unterschiedliche Implementierungen:
  - Unterdrückungstechniken (S. 24) oder Schlösser bzw. Schlossvariablen [9]

Plausibilitätsprüfung

- seien P_p, P_v **gleichzeitige Prozesse**, die P bzw. V ausführen, mit:

```
1 #define P(s) mps_prolaag(s)      /* acquire resource */
2 #define V(s) mps_verhoog(s)      /* release resource */
```

- sei s Zeiger auf Exemplar x von `semaphore_t` mit der **Vorbelegung**:

```
1 semaphore_t x = { 1 };           /* one resource, unlocked */
```

- dann ist festzustellen:
  - P_p kann sich weder mit sich selbst noch mit P_v überlappen
  - P_v kann sich weder mit sich selbst noch mit P_p überlappen
- darüber hinaus gilt als **Randbedingung**:
  - P_p legt sich außerhalb des kritischen Abschnitts schlafen, wenn die durch s kontrollierte Anzahl von Ressourcen erschöpft ist; es gilt $s \in \mathbb{Z}_-^*$
    - andere Exemplare von P_p oder P_v können den kritischen Abschnitt betreten
    - jeder weitere P_p erniedrigt s, auch jeder dieser P_p legt sich schlafen
  - P_v weckt höchstens ein Exemplar von P_p auf, der mit anderen Prozessen um Eintritt in den kritischen Abschnitt konkurriert; es gilt $s \in \mathbb{Z}_-$
    - aber jeder **Ersteintritt** von P_p erniedrigt s und blockiert P_p; es gilt $s \in \mathbb{Z}_-^*$
    - nur der P_p, der als einziger aufgeweckt wurde, begeht den **Wiedereintritt**

  $\hookrightarrow$ nur für diesen P_p ist die Wartebedingung aufgehoben, er verlässt $P(s)$...

Prozessblockade im kritischen Abschnitt

- ein Prozess, für den eine Wartebedingung erfüllt ist, während er einen kritischen Abschnitt belegt, muss sich wie folgt verhalten:
  - i den kritischen Abschnitt freigeben, ihn faktisch verlassen
  - ii blockieren, bis ein anderer Prozess die Wartebedingung aufheben konnte
  - iii sich um den Wiedereintritt in den kritischen Abschnitt bewerben
- auf den ersten Blick ist die damit verbundene **Ablaufunterbrechung und -fortsetzung** eines Prozesses einfach zu bewerkstelligen

```
1  void await(guard_t *lock) {
2      leave(&lock->bolt); /* unlock critical section */
3      sleep(&lock->wait); /* delay process, reschedule CPU */
4      enter(&lock->bolt); /* lock critical section */
5  }
6
7  void cause(guard_t *lock) {
8      process_t *next = elect(&lock->wait);
9      if (next)              /* one process unblocked */
10         ready(next);       /* schedule process */
11 }
```

- auf den zweiten Blick zeigt sich eine **wettlaufkritische Aktionsfolge**
- das **Aufwecksignal** für den Prozess kann verlorengehen (*lost wake-up*)

Wettlaufkritische Aktionsfolge beim Warten *lost wake-up*

- Ausgangssituation:
  - P hat die Wartebedingung für den Prozess festgestellt
  - V wird die Aufhebung eben dieser Bedingung signalisieren

```
1  void await(guard_t *lock) {
2       leave(&lock->bolt);
3       sleep(&lock->wait);
4       enter(&lock->bolt);
5  }
```

- seien P_p und P_v **gleichzeitige Prozesse**, die P bzw. V ausführen:

2–3
- P_p hat den kritischen Abschnitt freigegeben, ist noch nicht blockiert
- P_v betritt den kritischen Abschnitt, ruft `cause` auf (S. 12, Z. 13)
- `elect` findet P_p nicht auf der Warteliste (S. 15, Z. 8–9)

↪ das Signal zur Aufhebung der Wartebindung erreicht P_p nicht

3
- P_p legt sich schlafen, wird der Warteliste hinzugefügt und blockiert

↪ betritt niemand mehr den kritischen Abschnitt, blockiert P_p ewig

- die Lösung des Problems findet sich in den Aktionen, um P_p vom Zustand „laufend“ in den Zustand „blockiert“ zu überführen
  - für diese Überführung inkl. Wartelistenvermerk sorgt `sleep`...

Wartelistenvermerk und Zustandsübergang

- **Schlafenlegen** eines Prozesses umfasst zwei wichtige Hauptschritte:
  i. den aktuellen Prozess als „blockiert“ und für die Warteliste vermerken
  ii. einen anderen Prozess auswählen und diesem den Prozessor zuteilen

```
1  void sleep(event_t *wait) {
2      allot(wait);          /* register that process will block */
3      block();              /* delay process, reschedule CPU */
4  }
```

- `sleep` aufbrechen und den mit `allot` gemeinten Wartelistenvermerk in den kritischen Abschnitt „hochziehen“:

```
1  void await(guard_t *lock) {
2      allot(&lock->wait); /* register that process will block */
3      leave(&lock->bolt); /* unlock critical section */
4      block();            /* delay process, reschedule CPU */
5      enter(&lock->bolt); /* lock critical section */
6  }
```

  - damit kann `elect` P_p auf der Warteliste finden (S. 15, Z. 8–9)
  - P_p wird in den Zustand „bereit“ überführt (S. 15, Z. 10)
  - woraufhin `block` erkennt, dass P_p nicht (mehr) zu blockieren ist

Gliederung

Semaphor v. Mutex I

Hinweis (Informatikfolklore)

Ein Semaphor kann von jedem Prozess freigegeben werden.

- diese Feststellung wird oft als Nachteil vorgebracht, jedoch sind dabei die Semaphorarten (allgemein, binär) zu unterscheiden
  - strenggenommen ist sie eine **Anforderung** für den allgemeinen Semaphor und lediglich eine **Option** für den binären Semaphor
  - ein binärer Semaphor schützt einen kritischen Abschnitt, wobei eben zu differenzieren ist, ob darin ein **Prozesswechsel** geschieht oder nicht

$$\Rightarrow \quad \left.\begin{array}{r}\text{ohne}\\ \text{mit}\end{array}\right\} \text{muss} \left\{\begin{array}{c}\text{derselbe}\\ \text{ein anderer}\end{array}\right\} \text{Prozess den Semaphor freigeben}$$

Hinweis (Informatikfolklore)

Ein Mutex kann nur von dem Besitzerprozess freigegeben werden.

- diese Feststellung wird oft als Vorteil vorgebracht, ist jedoch nur auf den binären Semaphor ausgerichtet
  - nämlich zum Schutz eines kritischen Abschnitts ohne Prozesswechsel!

Semaphor v. Mutex II

Hinweis

*Prüfung der **Berechtigung** zur Freigabe eines kritischen Abschnitts (KA) ist ungeeignet für einen allgemeinen Semaphor, optional für einen binären Semaphor und notwendig für einen Mutex.*

notwendig
- ein **Mutex** sichert zu, dass die Freigabe von KA nur für den Prozess gelingen kann, der KA zuvor erworben hatte
- durch Verwendung eines binären Semaphors, Erfassung und Überprüfung des Besitzrechts für KA (vgl. S. 21)

ungeeignet
- P und V mit demselben **allgemeinen Semaphor** muss für verschiedene Prozesse möglich sein
- einseitige Synchronisation: Konsumenten und Produzenten

optional
- grundsätzlich lässt sich ein **binärer Semaphor** durch einen allgemeinen Semaphor S repräsentieren, wenn $S \leq 1$
- ungeeignet zum Schutz eines KA mit Prozesswechsel

- bei **unberechtigter Freigabe** sollte der Prozess abgebrochen werden — im privilegierten Modus ist das Rechensystem anzuhalten...

Spezialisierung eines binären Semaphors

- ein Mutex benutzt einen binären Semaphor, ersetzt ihn jedoch nicht
  - die Mutex-Datenstruktur setzt sich aus zwei Komponenten zusammen:
    i einem binären Semaphor zum Schutz eines kritischen Abschnitts *und*
    ii einer Handhabe zur eindeutigen Identifizierung eines Prozesses[4]
  - ausgehend davon seien die beiden folgenden Operationen definiert:

    `acquire` – vollzieht P und registriert den aktuellen Prozess als Eigentümer
    `release` – zeigt eine Ausnahme an, wenn der Prozess nicht Eigentümer ist
    – löscht ansonsten den Eigentümereintrag und vollzieht V
- ein dazu korrespondierender **Datentyp** kann wie folgt ausgelegt sein:

```
1 typedef volatile struct mutex {
2     semaphore_t sema;    /* binary semaphore */
3     process_t *link;     /* owning process or 0 */
4 } mutex_t;
```

[4] Auf Kernebene ist diese Handhabe der Zeiger zu einem Prozesskontrollblock, auf Benutzerebene ist sie die Prozessidentifikation.

Erwerben und freigeben eines Mutex

```
1  extern void panic(char*) __attribute__ ((noreturn));
2
3  void acquire(mutex_t *mutex) {
4      P(&mutex->sema);            /* lockout */
5      mutex->link = being(ONESELF);   /* register owner */
6  }
7
8  void release(mutex_t *mutex) {
9      if (mutex->link != being(ONESELF)) /* it's not me! */
10         panic("unauthorised release of mutex");
11
12     mutex->link = 0;            /* deregister owner */
13     V(&mutex->sema);            /* unblock */
14 }
```

- die **unberechtigte Freigabe** eines Mutex ist eine sehr **ernste Sache**
  - das nichtsequentielle Programm enthält einen **Softwarefehler** (*bug*)
  - Fehlerkode liefern ist keine Option, da seine Behandlung nicht sicher ist
  - anderes als eine **nichtmaskierbare Ausnahme** anzuzeigen, ist fraglich...

Gliederung

Prozessauslösung verhindern

- gleichzeitigen Prozessen vorbeugen dadurch, dass der **Mechanismus** für ihre Auslösung zeitweilig außer Kraft gesetzt ist

  Unterbrechung
  - Ursprung unvorhersehbarer gleichzeitiger Abläufe
  - asynchron zum aktuellen Prozess und Betriebssystem
  - *first-level interrupt handler* (FLIH)

  Fortsetzung
  - anschließender Teil des FLIH, synchron zum Systemkern
  - *second-level interrupt handler* (SLIH)

  Verdrängung
  - anschließender Teil eines SLIH, synchron zum Planer
  - Aktionsfolge für die präemptive Prozessumschaltung

- damit werden immer auch Prozesse ausgesperrt, die überhaupt nicht in Konflikt mit dem aktuellen Prozess geraten werden ☹
  - kausal unabhängige gleichzeitige Abläufe werden unnötig unterbunden
  - unkritische Parallelität wird eingeschränkt, Leistungsfähigkeit beschnitten
- darüber hinaus: diese Techniken greifen nur prozessor(kern)lokal, sind **ungeeignet für** ein-, mehr- oder vielkernige **Multiprozessorsysteme**
  - für letztere ist auf Schlösser bzw. Schlossvariablen zurückzugreifen [9]

FLIH zeitweilig maskieren


```
1  inline void enter(detent_t *nest) {
2      if (nest) {       /* save contents of PSW */
3          ...           /* vgl. S.36 */
4      }
5      asm volatile ("cli" : : : "cc");     /* disable interrupts */
6  }
7
8  inline void leave(detent_t *nest) {
9      if (!nest)        /* enable interrupts */
10         asm volatile ("sti" : : : "cc");
11     else {            /* restore contents of PSW */
12         ...           /* vgl. S.36 */
13     }
14 }
```

- **Verschachtelungen** erfordern bei Entsperrung die Wiederherstellung des Sperrzustands, der im Moment der Sperrung galt

```
15 typedef volatile struct detent {
16     psw_t flags;      /* process status word */
17 } detent_t;
```

- dazu muss der Inhalt des Prozessorstatuswortes invariant gehalten werden

SLIH zeitweilig maskieren

s. auch S. 37 und 38

```
1  typedef volatile struct detent {
2      int flag;                          /* saved SLIH lock status */
3  } detent_t;
```

- der Sperre zur **Unterbrechungsanforderung** (*interrupt request*) sehr ähnlich, nur wird ein Software- und nicht Hardwaresignal geblockt

```
4  void enter(detent_t *gate) {
5      gate->flag = avert(&slih);   /* disable SLIH */
6  }
7
8  void leave(detent_t *gate) {
9      if (gate->flag == 0) {       /* nested? */
10         grant(&slih);            /* no, enable SLIH */
11         if (order(&slih))        /* SLIH pending? */
12             flush(&slih);        /* yes, catch up... */
13     }
14 }
```

- während die Ausführung eines SLIH unterbunden ist, kann sein FLIH allerdings zur Ausführung kommen
  - der vom FLIH ausgelöste SLIH kommt ggf. in eine Warteschlange
  - beim Verlassen des gesperrten Abschnitts wird diese abgebaut (Z. 11–12)

Planer zeitweilig maskieren

scheduler lock

- im Grunde genommen die Spezialisierung des eben (S. 26) skizzierten Ansatzes, den SLIH zeitweilig zu maskieren
  - nicht jeder SLIH wird verzögert, sondern nur die zum Planer führenden
  - also jeder SLIH, der die **Umplanung** (*rescheduling*) von Prozessen auslöst
- bei aktivierter Sperre wird der aktuelle Prozess zwar unterbrochen, ihm wird jedoch nicht der Prozessor entzogen
  - der gesperrte Abschnitt wird auch als **nichtunterbrechender kritischer Abschnitt** (*non-preemptive critical section*, NPCS) bezeichnet
  - Entzug des Prozessors ist erst nach Verlassen dieses Abschnitts möglich
- dabei muss es nicht wirklich zur Verzögerung der Prozesseinplanung kommen, wohl aber zu der der **Prozesseinlastung**
  - der Planer reiht den bereitgestellten Prozess strategiegemäß ein, wird den **Abfertiger** (*dispatcher*) ggf. zum Prozesswechsel auffordern
    - wenn die Umplanung feststellt, den aktuellen Prozess wegschalten zu müssen
  - ist die Sperre im Moment der Aufforderung aktiv, wird der Aufruf an den Abfertiger jedoch zurückgestellt (*deferred procedure call*, DPC [1])
  - zurückgestellte Aufrufe werden beim Verlassen des gesperrten Abschnitts von dem dann aktuellen Prozess wieder aufgenommen und durchgeführt

Gliederung

Resümee

- Synchronisation in der Maschinenprogrammebene kann auf Konzepte von Betriebssystemen zurückgreifen
  - die den Zeitpunkt von Einplanung oder Einlastung gezielt beeinflussen
  - die Prozesse kontrolliert schlafen legen und wieder aufwecken
- typische ELOP dieser Ebene ist der **Semaphor**, ein Verbundatentyp bestehend aus Zähl- und Ereignisvariable
  - unterschieden wird zwischen binärem und allgemeinem Semaphor
  - seine Primitiven (P, V) bilden logisch bzw. physisch atomare Aktionen
- **Atomarität** der Semaphorprimitiven ist durch Techniken zu erreichen, die hierarchisch tiefer (d.h., auf Befehlssatzebene) angesiedelt sind
  - **Sperren** (physisch) von Unterbrechungen, Fortsetzungen, Verdrängungen
  - **Schlösser** (physisch) oder **nichtblockierende Synchronisation** (logisch)
- nicht zu vergessen der **Mutex**: eine Semaphorspezialisierung, die die Eigentümerschaft bei Freigabe prüft und letztere bedingt zulässt
  - der Mutex benutzt einen binären Semaphor, ersetzt ihn jedoch nicht
  - denn uneingeschränkte Semaphorfreigabe ist ein Merkmal, kein Makel

Literaturverzeichnis I

[1] Baker, A. ; Lozano, J. :
Deferred Procedure Calls.
In: *Windows 2000 Device Driver Book: A Guide for Programmers*.
Prentice Hall, 2000

[2] Dijkstra, E. W.:
Over seinpalen / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1964 ca. (EWD-74). –
Manuskript. –
(dt.) Über Signalmasten

[3] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

[4] Dijkstra, E. W.:
The Structure of the "THE"-Multiprogramming System.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 341–346

Literaturverzeichnis II

[5] Hoare, C. A. R.:
Communicating Sequential Processes.
In: *Communications of the ACM* 21 (1978), Nr. 8, S. 666–677

[6] Kleinöder, J. ; Schröder-Preikschat, W. :
Monitor.
In: [10], Kapitel 10.2

[7] Kleinöder, J. ; Schröder-Preikschat, W. :
Nichtsequentialität.
In: [10], Kapitel 10.1

[8] Kleinöder, J. ; Schröder-Preikschat, W. :
Prozesse.
In: [10], Kapitel 6.1

[9] Kleinöder, J. ; Schröder-Preikschat, W. :
Schlösser und Spezialbefehle.
In: [10], Kapitel 10.4

[10] Kleinöder, J. ; Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)

Literaturverzeichnis III

[11] Kleinöder, J. ; Schröder-Preikschat, W. :
Virtuelle Maschinen.
In: [10], Kapitel 5.1

[12] Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Concurrent Systems — Nebenläufige Systeme.
FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien)

Semaphormonitor


```
1  template<monitor = signal urgent wait>
2  class Semaphore {
3      int load;              // # of allowed/waiting processes
4      condition free;        // to block/unblock processes
5
6  atomic:
7      Semaphore(int seed = 0) { load = seed; }
8
9      void prolaag() {
10         load -= 1;
11         if (load < 0)
12             free.wait();
13     }
14
15     void verhoog() {
16         load += 1;
17         if (load <= 0)
18             free.signal();
19     }
20 };
```

```
21 void P(Semaphore& sema) {
22     sema.prolaag();
23 }
24
25 void V(Semaphore& sema) {
26     sema.verhoog();
27 }
```

- nur der Hoare'sche Monitor (*signal and urgent wait*) lässt hier die Formulierung von `prolaag()` zu, wie vorher (S. 10) skizziert
  - Annahme ist die sofortige Wiederaufnahme des signalisierten Prozesses

Es ist nicht alles Gold, was glänzt. . .

- so naheliegend die Implementierung eines Semaphors als Monitor ist, sie muss sich einigen Herausforderungen stellen:
  - keine der heute gebräuchlichen Systemprogrammiersprachen hat den Begriff „Monitor“ als Sprachkonstrukt integriert, auch Java nicht:
    `synchronized` – wechselseitiger Ausschluss
    – Methoden- oder Grundblockausführung
    `wait` – blockiert den Prozess, hebt umfassendes `synchronized` auf
    – bewirbt umfassendes `synchronized` bei Wiederaufnahme
    `notify` – deblockiert genau einen wartenden Prozess
    `notifyAll` – deblockiert alle wartende Prozesse
  - darüberhinaus ist (Standard-) Java keine Systemprogrammiersprache, die zur Implementierung von hardwarenahen Programmen geeignet ist
- der Hoare'sche Monitor hat einen recht hohen Laufzeitaufwand und, bis auf CSP [5], keine praktische Umsetzung erfahren
  - vergleichsweise hohe Anzahl von Prozess-/Kontextwechsel
  - Atomarität der Aktionsfolgen `signal` → `wait` und wieder zurück
- ein Monitor abstrahiert vom Semaphor, um Fehler beim Umgang mit Semaphore zu vermeiden ⇝ **Bruch im Abstraktionsprinzip. . .**

Trennung der Zuständigkeiten

separation of concerns

- der Semaphor entscheidet nur noch, wann ein Prozess zu deblockieren ist, jedoch überlässt es dem **Planer**, welcher dies sein wird

```
1  void mps_prolaag(semaphore_t *sema) {
2      enter(&sema->lock.bolt);      /* lock critical section */
3      while (--sema->load < 0)      /* resource(s) exhausted? */
4          await(&sema->lock);       /* fulfilled, wait outside */
5      leave(&sema->lock.bolt);      /* unlock critical section */
6  }
7
8  void mps_verhoog(semaphore_t *sema) {
9      enter(&sema->lock.bolt);      /* lock critical section */
10     if (sema->load >= 0)          /* any waiting process? */
11         sema->load += 1;          /* no, increase sema. value */
12     else {
13         sema->load = 1;           /* yes, enable at most one */
14         rouse(&sema->lock);       /* but notify all processes */
15     }
16     leave(&sema->lock.bolt);      /* unlock critical section */
17 }
```

- *V* weckt alle wartende Prozesse auf, lässt aber nur einen davon aus *P*
- *P* zwingt erwachte Prozess zur Neuauswertung der Wartebedingung

FLIH verschachtelungsfähig maskieren


```
1  inline void enter(detent_t *nest) {
2      if (nest) {                          /* save contents of PSW */
3          asm volatile (
4              "pushf\n\t"                  /* read from flags register */
5              "popl %0"                    /* save to prototype */
6              : "=m" (nest->flags) :
7              : "memory", "cc");
8      }
9      asm volatile ("cli" : : : "cc");     /* disable interrupts */
10 }
11
12 inline void leave(detent_t *nest) {
13     if (!nest)                           /* enable interrupts */
14         asm volatile ("sti" : : : "cc");
15     else {                               /* restore contents of PSW */
16         asm volatile (
17             "pushl %0\n\t"               /* read from prototype */
18             "popf"                       /* write to flags register */
19             : : "m" (nest->flags)
20             : "memory", "cc");
21     }
22 }
```

Zurückgestellte Prozeduraufrufe

```
1  typedef struct sentry {
2      int lock;                        /* activiy state */
3      queue_t wait;                    /* deferred procedure calls */
4  } sentry_t;
5
6  inline int avert(sentry_t *gate) {
7      return FAS(&gate->lock, 1);      /* try to activate section */
8  }
9
10 inline void grant(sentry_t *gate) {
11     gate->lock = 0;                  /* deactivate section */
12 }
13
14 inline chain_t *order(sentry_t *gate) {
15     return gate->wait.head.link; /* next DPC to be processed */
16 }
17
18 extern void flush(sentry_t *);     /* process all pending DPCs */
19 extern sentry_t slih;              /* kernel-global guardian */
```

Atomare Lese-/Schreibaktion

fetch and store, x86

```
1  inline int FAS(int *ref, int val) {
2      int aux;
3      asm volatile(
4          "xchgl %0, %1"        /* atomic read-write action */
5          : "=q" (aux), "=m" (*ref)
6          : "r" (val), "m" (*ref)
7          : "memory", "cc");
8      return aux;
9  }
```

- darauf und auf die Implementierung eines DPC (S. 37) basierende Kompilierung[5] von `enter` (S. 26) liefert:

```
10 _enter:
11     movl    4(%esp), %eax    # get pointer to detent flag
12     movl    $1, %ecx         # get target activity state
13     ## InlineAsm Start
14     xchgl   %ecx, _slih      # exchange with sentry lock
15     ## InlineAsm End
16     movl    %ecx, (%eax)     # save former activity state
17     retl
```

[5] `gcc -O3 -m32 -static -fomit-frame-pointer -S`

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – X.4 Prozesssynchronisation: Kreiseln und Spezialbefehle

Wolfgang Schröder-Preikschat

30. November 2021

Agenda

Einführung

Umlaufsperre
- Definition
- Funktionsweise
- Schlossalgorithmen
- Diskussion

Transaktion
- Motivation
- Prinzip
- Beispiele
- Diskussion

Zusammenfassung

Gliederung

Lehrstoff

- Konzepte der **Befehlssatzebene** (s. [10]) kennenlernen, womit die Synchronisation gleichzeitiger Prozesse erreicht wird
  - **Umlaufsperren**, d.h., Sperren für mehr-/vielkernige Multiprozessoren
  - sperrfreie Synchronisation mittels (Mikro-)**Transaktionen**
- für Umlaufsperren typische **Schlossalgorithmen** behandeln und ihre Auswirkungen auf andere Prozesse untersuchen
  - grundsätzliche wie auch spezielle Schwachstellen thematisieren
  - schrittweise Techniken für Verbesserungen entwickeln und erklären
- als Antwort zu Unzulänglichkeiten von Schlossalgorithmen, im Ansatz die **nichtblockierende Synchronisation** vorstellen
  - dazu einfache, musterhaftige **transaktionale Programme** diskutieren
  - sie als **nebenläufige Abschnitte** mit kritischen Abschnitten vergleichen
- die Bedeutung der **Spezialbefehle** und der diesbezüglichen Rolle von Schleifenkonstruktionen für beide Konzepte erfassen
  - sehen, wie TAS, CAS und FAA genutzt wird und implementiert ist
  - Gemeinsamkeiten und Unterschiede bei Anwendung der Befehle erkennen

Gliederung

Sperren durch schnelle Drehung

Kein „Drängelgitter“, das Passierende, durch Blickwendung im Umlauf weiter induziert, zur Vorsicht aufruft.

- diesem aber nicht ganz unähnlich:
  - Passierende ≡ Prozesse
  - Blickwendung ≡ Zustandsabfrage
  - Umlauf ≡ Schleife
- durch **Kreiseln** (*spin*) das **Sperren** (*lock*) von Prozessen steuern:

`acquire`
- verzögert den aktuellen Prozess, solange die Sperre gesetzt ist
  - **aktives Warten** (*busy waiting*) lässt den Prozess kreiseln
- verhängt die Sperre, sobald sie aufgehoben wurde

`release`
- hebt die Sperre auf, ohne den aktuellen Prozess zu verzögern

- alias *lock* (sperren) und *unlock* (entsperren)

- die Prozesssteuerung manifestiert sich in Verfahren, die in Form von sogenannten **Schlossalgorithmen** (*lock algorithms*) umgesetzt sind

[1] vgl. auch `https://de.wikipedia.org/wiki/Umlaufgitter` (28/10/15).

Grundsatz

*Die **Umlaufsperre** dient der Synchronisation gleichzeitiger (gekoppelter) Prozesse **verschiedener Prozessoren** oder Prozessorkerne eines Rechensystems mit gemeinsamem Speicher.*

*Synchronisation solcher Prozesse **desselben Prozessors** erfordert gegebenenfalls zusätzlich eine **unilaterale Sperre** zur Vorbeugung unvorhersehbarer Latenz,[2] beispielsweise eine Unterbrechungssperre — die allerdings den privilegierten Modus voraussetzt.*

[2] Wegen der sonst für gewöhnlich möglichen Unterbrechung und Verzögerung des die Sperre haltenden Prozesses: *lock-holder preemption*.

Schlossalgorithmus: naive Fassung

- in einfachster Form bildet eine **binäre Schlossvariable** die Grundlage z.B. entsprechend folgendem Datentyp:

```
1  #include <stdbool.h>
2
3  typedef volatile struct lock {
4      bool busy;            /* initial: false */
5  } lock_t;
```

- auf einem Exemplar dieses Datentyps operieren die Primitiven einer Umlaufsperre dem **Prinzip** nach wie folgt:

```
6  void acquire(lock_t *lock){
7      while (lock->busy); /* spin until lock release */
8      lock->busy = true;  /* claim lock-out */
9  }
10
11 void release(lock_t *lock){
12     lock->busy = false; /* abandon lock-out */
13 }
```

- das wäre zu schön, um wahr zu sein: **wettlaufkritische Aktionsfolge** ☹

Problemanalyse

- Ausgangssituation:
  - **gleichzeitige Prozesse**
  - die in `acquire` geschehen

```
1  void acquire(lock_t *lock){
2       while (lock->busy);
3       lock->busy = true;
4  }
```

- **wettlaufkritische Aktionsfolge**:
  1. $n > 1$ Prozesse rufen gleichzeitig `acquire` auf, betreten den Rumpf
  2. sie werten gleichzeitig den Zustand der Schlossvariablen (`busy`) aus
     - alle stellen gleichzeitig die Aufhebung der Wartebedingung/Sperre fest
     - als Folge verlassen alle gleichzeitig die kopfgesteuerte Schleife
  3. alle Prozesse verhängen gleichzeitig die (soeben aufgehobene) Sperre
  4. sie verlassen gleichzeitig `acquire`, betreten den kritischen Abschnitt

  ↪ wechselseitiger Ausschluss scheitert: $n > 1$ Prozesse fahren fort ☹

- Lösungsansatz:
  - die Aktionsfolge „Sperre prüfen und ggf. verhängen“ atomar auslegen
  - Atomarität ist in dem Fall nur mit Mitteln der Befehlssatzebene erreichbar
    - unilaterale Sperren [15, S. 24–27] scheiden aus: sie wirken nur lokal auf dem Prozessor, wofür Umlaufsperren eben nicht vorgesehen sind (S. 7)
    - stattdessen sind in Hardware implementierte **Spezialbefehle** erforderlich

Sperre prüfen und verhängen — atomar

- testen, das Ergebnis vermerken, und setzen: ***test and set*** [7, p. 144]

```
1 bool TAS(bool *ref) {
2     atomic { bool aux = *ref; *ref = true; }
3     return aux;
4 }
```

- diese Operation wirkt immer schreibend auf den Arbeitsspeicher, aber die Werteveränderung der adressierten Variable erfolgt nur bedingt
  - nämlich nur, wenn die Variable den Wert 0 bzw. `false` enthielt
  - jedoch wird unbedingt der Wert 1 bzw. `true` in die Variable geschrieben
  - das Operationsergebnis ist der Variablenwert vor dem Überschreiben
- bei Operationsausführung durch die Hardware erfolgt **wechselseitiger Ausschluss** gleichzeitiger Speicherbuszugriffe der Prozessoren
  - bewirkt wird ein **atomarer Lese-/Schreibzyklus**
    - unilaterale Sperre asynchroner Programmunterbrechungen[3] und
    - multilaterale Sperre gleichzeitiger Buszugriffe „von außen kommend“

[3] Normal: heute lässt ein Prozessor Programmunterbrechungen bei sich erst am Ende des im Moment der Unterbrechungsanforderung interpretierten Befehls zu.

Implementierung

- Reduktion auf eine atomare, in GCC eingebaute intrinsische Funktion:

```
1  #define TAS(ref) __sync_lock_test_and_set(ref, 1)
```

- Anwendung dieser Funktion für den Schlossalgorithmus:

```
2  void acquire(lock_t *lock) {
3      while (TAS(&lock->busy));    /* spin until claimed */
4  }
```

- Kompilierung in Assembliersprache (ASM86, AT&T Syntax):

```
5  _acquire:
6      movl   4(%esp), %eax # get pointer to lock variable
7  LBB0_1:                 # come here to retry (while)...
8      movb   $1, %cl       # want to change lock to "true"
9      xchgb  %cl, (%eax)   # atomically swap operand values
10     testb  $1, %cl       # check former lock value
11     je     LBB0_1        # if it equals "true", retry
12     ret                  # was "false" and is now "true"
```

- die relevante atomare Operation findet sich in Zeile 9: `xchgb` [8]

Kreiseln mit TAS

```
1  void acquire(lock_t *lock) {
2      while (TAS(&lock->busy));    /* spin until claimed */
3  }
```

- naive Lösung mit schädlicher Wirkung auf **Pufferspeicher** (*cache*)
  - unbedingtes Schreiben bei **Wettstreit** löst massiven Datentransfer aus:
    - $n-1$ Kopien invalidieren und 1 Original zum auslösenden Prozessor bewegen
      oder
    - 1 Original schreiben und $n-1$ Kopien bei anderen Prozessoren aktualisieren
  - die Pufferspeicherzeile (*cache line*) mit der Schlossvariablen „flattert“
- hinzu kommt schädliche Wirkung auf kausal unabhängige Prozesse
  - nahezu anhaltender wechselseitiger Ausschluss von Speicherbuszugriffen
    - jede Ausführung von TAS sperrt den Bus für andere Prozessoren
    - dazwischen liegen nur wenige (z.B. drei, vgl. S. 11) normale Befehle
  - blockt Prozessoren, wenn Prozessdaten nicht im Pufferspeicher vorliegen
  - erzeugt **Störung** (*interference*) in Prozessen anderer Prozessoren
- in nichtfunktionaler Hinsicht skaliert die Lösung ziemlich schlecht
  - Kreiseln mit bedingtem Schreiben, mit Ablesen oder Zurückhaltung...

Kreiseln mit CAS

```
1 #define CAS __sync_bool_compare_and_swap

2 void acquire(lock_t *lock) {
3     while (!CAS(&lock->busy, false, true));
4 }
```

- wobei Funktionssignatur ***CAS(Variable, Prüfwert, Neuwert)*** einen atomaren Spezialbefehl wie folgt definiert beschreibt:

$$CAS = \begin{cases} true \rightarrow \text{Neuwert zugewiesen}, & \text{falls Variable} = \text{Prüfwert} \\ false, & \text{sonst} \end{cases}$$

  - der Befehl schreibt nur, wenn die **Gleichheitsbedingung** erfüllt ist
- die schädliche Wirkung auf den Pufferspeicher bleibt aus, nicht aber auf kausal unabhängige Prozesse
  - nahezu anhaltender wechselseitiger Ausschluss von Speicherbuszugriffen
  - ungünstiges Verhältnis zur Anzahl normaler Befehle (1:3, vgl. S. 34)
- in nichtfunktionaler Hinsicht skaliert die Lösung schlecht
  - ***bus-lock burst*** $\rightsquigarrow$ Kreiseln mit Ablesen oder mit Zurückhaltung...

Kreiseln mit Ablesen

spin on read

```
1  void acquire(lock_t *lock) {
2      do {
3          while (lock->busy);
4      } while (!CAS(&lock->busy, false, true));
5  }
```

- schwächt Wettstreit beim Buszugriff und damit Interferenz ab
  - 3
    - die eigentliche Warteschleife, fragt nur den Pufferspeicher ab
    - keine Datenbuszugriffe, kausal unabhängige Prozesse bleiben ungestört
  - 4
    - die Sperre wird verhängt, wenn sie immer noch aufgehoben ist[4]
    - betrifft gekoppelte (gleichzeitige) Prozesse stark, andere jedoch kaum
- steht und fällt allerdings mit der Länge des kritischen Abschnitts
  - ist er zu kurz, degeneriert die Lösung zum Kreiseln mit CAS

  ↪ dabei wird eine Umlaufsperre aber gerade oft für diesen Fall favorisiert ☹
- bei großem Wettstreit stauen sich nach wie vor viele Prozesse (Z. 4)
  - sich wiederholende **Häufung der Bussperre** (*bus-lock burst*)
  - Prozessen anderer Prozessoren wird stoßartig Buszugriffe verwehrt
- in nichtfunktionaler Hinsicht skaliert die Lösung mehr oder weniger
  - Kreiseln mit Zurückhaltung: **Stauauflösung**, Lücken schaffen...

[4] Beachte, dass der kreiselnde Prozess überholt worden sein kann.

Kreiseln mit Zurückhaltung

spin with backoff

Definition (*backoff*)

Statische oder dynamische **Verweilzeit**, prozessorweise abgestuft, bis zur Wiederaufnahme der vormals wettstreitigen Aktion.

```
1  void acquire(lock_t *lock) {
2      do {
3          while (lock->busy); /* spin on read */
4          if (CAS(&lock->busy, false, true))
5              return;         /* lock acquired, done */
6          backoff(lock->time, earmark());
7      } while (true);         /* contention faced, retry */
8  }
```

- angenommen sei eine *sperrenspezifische Verweilzeit* (`time`)

`earmark` ▪ liefert die Nummer des ausführenden Prozessor(kern)s

- der Telekommunikation entlehnter Ansatz zur **Blockierungskontrolle** (*congestion control*) bei **Kanalüberzeichnung**:
  - statische (ALOHA [1]) oder dynamische (Ethernet [14]) Verzögerungen
  - **Stauauflösung** (*contention resolution*), ausgeübt zum Sendezeitpunkt

Verhungern

starvation

- alle bisher diskutierten Verfahren können Prozessen eine **nach oben unbegrenzte Wartezeit** bescheren
  - jedoch wird einem System gekoppelter Prozesse Fortschritt zugesichert — wenn Verklemmungen einmal außer Acht gelassen werden
  - jedoch können einzelne Prozesse ewig im Eintrittsprotokoll kreiseln
- grenzenlose Verzögerung einzelner Prozesse ist vorzubeugen
- die Wartezeit muss für jeden Prozess limitiert sein
  - eine obere Schranke ist notwendig

- jedoch darf für jeden Prozess die **effektive Wartezeit** bis zur oberen Schranke variabel sein
- das meint Verfahren, die (a) fair für die Prozesse und (b) auch noch frei von Interferenz mit dem Planer sind
  - (a) ist durchaus einfach (s. umseitig), verträgt sich aber selten mit (b)

Kreiseln mit proportionaler Zurückhaltung

s. auch S. 35

- das **Maß der angestauten Prozesse** bestimmt den Wettstreitgrad, der im Moment der Sperrverhängung gilt $\rightsquigarrow$ Wettstreiter zählen
- Ideengeber ist der sog. Bäckereialgorithmus [12]: ***ticket spin lock***

```
1  typedef volatile struct lock {
2      long next;  /* number being served next */
3      long this;  /* number being currently served */
4      long time;  /* duration of critical section */
5  } lock_t;
```

```
#define FAA __sync_fetch_and_add                    /* atomic */
```

```
6  void acquire(lock_t *lock) {
7      long self = FAA(&lock->next, 1);     /* my number served */
8      if (self != lock->this) {            /* wait one's turn */
9          backoff(lock->time, self - lock->this);
10         while (self < lock->this);
11     }
12 }
13
14 void release(lock_t *lock) { lock->this += 1; } /* next one */
```

Der Wert *self* − *this* gibt die Anzahl der Prozesse, die den kritischen Abschnitt zuerst durchlaufen werden.

- ein der mittels **Wartemarkenspender** sowie **Personenaufrufanlage** realisierten Kundenverkehrssteuerung entlehnter Ansatz ☺

Zwischenzusammenfassung

- **Sperren** wirken einseitig (unilateral: Unterbrechungs-, Fortsetzungs-, Verdrängungssperre) oder **mehrseitig** (multilateral: Umlaufsperre)
  - nur die Umlaufsperre wirkt auf die tatsächlich gekoppelten Prozesse
- Umlaufsperren und **verdrängende Prozesseinplanung** integriert im selben Bezugssystem vertragen sich nicht ohne weiteres
  - Prozessorentzug des Schlosshalters (*lock-holder preemption*) ist möglich
  - führt auch bei kleinsten kritischen Abschnitten zu hohem Leistungsverlust
    - Stau gekoppelter Prozesse verlängert sich, unbestimmte Verweilzeit
  - jeder Art von Verzögerung des Schlosshalters muss vorgebeugt werden
  - Konsequenz ist, zusätzlich eine **Unterbrechungssperre** zu verhängen
- **wechselseitiger Ausschluss** ist kein Allheilmittel, um Aktionsfolgen mit wettlaufkritischen Eigenschaften abzusichern
  - **arbeitsloses Kreiseln** für den wartenden Prozess und gleichzeitig damit **Störung** anderer Prozesse, die mit ihm denselben Prozessor teilen
  - **Verklemmungsgefahr** (*deadly embrace* [5, S. 73]) gekoppelter Prozesse
- Defizite, die blockierende Synchronisation grundsätzlich betreffen, jedoch mit Umlaufsperren besonders zum Vorschein kommen
  - Alternative ist die **nichtblockierende Synchronisation**: **Transaktion**

Gliederung

Nachteile blockierender Synchronisation

- Probleme des in **Software** erzwungenen wechselseitigen Ausschlusses gekoppelter Prozesse durch Monitore, Semaphore oder Sperren
  - Leistung (*performance*) paralleler Systeme bricht ein/nimmt ab
    - Kreiseln vor Sperren reduziert Busbandbreite [3]
    - höherer Anteil sequentieller Programmbereiche [2]
  - Robustheit (*robustness*) „*single point of failure*“
    - im kritischen Abschnitt „abstürzen“ lässt diesen gesperrt
    - schlimmstenfalls wird das ganze System lahmgelegt
  - Interferenz (*interference*) mit dem Planer
    - Planungsentscheidungen werden nicht durchgesetzt
    - Prioritätsverletzung, Prioritätsumkehr [13]
      - Mars Pathfinder [16, 9]
  - Lebendigkeit (*liveness*) einiger oder sogar aller Prozesse
    - Gefahr von Verhungern (*starvation*)
    - inherent anfällig für Verklemmung (*deadlock*)
- etwas anderes ist wechselseitiger Ausschluss in der **Hardware**, insb. bei der Ausführung von Spezialbefehlen (TAS, CAS, FAA)

Pessimistischer vs. optimistischer Ansatz

- **softwaregesteuerter wechselseitiger Ausschluss** trifft eine negative Erwartung in Bezug auf gleichzeitige Prozesse
  - es wird die wettlaufkritische Aktionsfolge geben: **pessimistischer Ansatz**
  - um den Konflikten vorzubeugen, wird frühzeitig die Reißleine gezogen
  - da die Prozesse wahrscheinlich nur für kurze Zeit blockieren werden ☹
- demgegenüber stehen Paradigmen, die eine positive Erwartung treffen und gleichzeitige Prozesse in Software nicht ausschließen
  - die wettlaufkritische Aktionsfolge gibt es nicht: **optimistischer Ansatz**
  - falls doch, sind Konflikte nachträglich erkenn- und behandelbar ☺
  - dazu wird **hardwaregesteuerter wechselseitiger Ausschluss** benutzt
- hierzu greift letzteres auf Konzepte zurück, die ihren Ursprung in der Programmierung von Datenbanksystemen finden

Definition (Optimistic Concurrency Control [11])

Method of coordination for the purpose of updating shared data by mainly relying on **transaction backup** as control mechanisms.

Kreiseln mit Bedingung

Definition (Transaktion, nach [6, S. 624])

Eine **Konsistenzeinheit**, die Aktionsfolgen eines Prozesses gruppiert.

- die Aktionsfolge ist nicht atomar, jedoch wird das berechnete Datum einer gemeinsamen Variablen nur bei **Isolation** übernommen
  - d.h., wenn diese Folge **zeitlich** isoliert von sich selbst stattfindet
  - wozu sie **ablaufinvariant** für gekoppelte Prozesse formuliert sein muss
- eine solche Aktionsfolge wird i.A. durch eine **fußgesteuerte Schleife** umfasst, in der gleichzeitige Prozesse stattfinden können

```
1  ERLEDIGE Transaktion:
2    WIEDERHOLE
3      erstelle die lokale Kopie des Datums an einer globalen Adresse;
4      verwende diese Kopie, um ein neues Datum zu berechnen;
5      versuche, das neue Datum an der globalen Adresse zu bestätigen;
6    SOLANGE die Bestätigung gescheitert ist;
7  BASTA.
```

- zur Bestätigung (Z. 5) kommt ein **Spezialbefehl** der CPU zum Einsatz
- nur für den gilt **hardwaregesteuerter wechselseitiger Ausschluss**

Bedingte Wertzuweisung

vgl. auch S. 13

- **atomare Bestätigungsaktion** einer Transaktion (S. 22, Z. 5):

```
1 atomic bool CAS(type *ref, type old, type new) {
2     return (*ref == old) ? (*ref = new, true) : false;
3 }
```

  - `true` ▪ Bestätigung gelang, neues Datum geschrieben
  - `false` ▪ Bestätigung scheiterte, referenzierte Variable ist unverändert

- Reduktion auf eine atomare, in GCC eingebaute intrinsische Funktion:

```
1 #define CAS __sync_bool_compare_and_swap
```

  - zur Kompilierung o.g. Funktion nach Assembliersprache, siehe S. 37

- typisches Muster (*pattern*) einer fußgesteuerten (CAS) Transaktion:

```
1 do /* transaction */ {
2     any_t old = *ref;              /* make local copy */
3     any_t new = handle(old);       /* compute some value */
4 } while (!CAS(ref, old, new));     /* try to commit */
```

  - alle Aktionen im **Schleifenrunpf** können durch gleichzeitige Prozesse geschehen, sie unterliegen nicht dem wechselseitigen Ausschluss
  - nur die **Fußsteuerung** der Schleife mit dem CAS läuft synchronisiert ab

Atomare multiplikative Variablenänderung

- Fassung als klassischer **kritischer Abschnitt** zum Vergleich:

```
1  long mult(long_t *ref, long val) {
2      long new;
3
4      enter(&ref->bolt);              /* lock critical section */
5      new = (ref->data *= val);       /* perform computation */
6      leave(&ref->bolt);              /* unlock critical section */
7
8      return new;
9  }
```

```
10 typedef struct longlock {
11     long data;
12     detent_t bolt;
13 } long_t;
```

- semantisch äquivalente Fassung als **nebenläufiger Abschnitt**:

```
14 long mult(long *ref, long val) {
15     long new, old;
16
17     do old = *ref;                  /* make copy, compute & commit */
18     while (!CAS(ref, old, new = old * val));
19
20     return new;
21 }
```

- funktional ist die Multiplikation zu leisten, die ungesperrt stattfindet
- nur die Bestätigung des Ergebnisses unterliegt wechelseitigem Ausschluss

Atomare Listenmanipulation I

- einfach verkettete Liste, Verarbeitung nach LIFO (*last in, first out*):

```
1  typedef struct chain {
2      struct chain *link;
3  } chain_t;
```

```
4  typedef struct chainlock {
5      chain_t item;
6      detent_t bolt;
7  } chainlock_t;
```

- Einfügeoperation als klassischer **kritischer Abschnitt** zum Vergleich:

```
8  void push(chainlock_t *head, chain_t *item) {
9      enter(&head->bolt);              /* lock critical section */
10     item->link = head->item.link;    /* prepend item */
11     head->item.link = item;          /* adjust head pointer */
12     leave(&head->bolt);              /* unlock critical section */
13 }
```

- semantisch äquivalente Fassung als **nebenläufiger Abschnitt**:

```
14 void push(chain_t *head, chain_t *item) {
15     do item->link = head->link;      /* prepend item & commit */
16     while (!CAS(&head->link, item->link, item));
17 }
```

  - funktional ist das Voranstellen und die Kopfzeigeraktualisierung zu leisten
  - nur letztere Aktion unterliegt dem wechelseitigen Ausschluss

Atomare Listenmanipulation II

- Entnahmeoperation als **kritischer Abschnitt** zum Vergleich:

```
1  chain_t *pull(chainlock_t *head) {
2      chain_t *item;
3
4      enter(&head->bolt);      /* lock critical section */
5      if ((item = head->item.link) != 0)
6          head->item.link = item->link;
7      leave(&head->bolt);      /* unlock critical section */
8
9      return item;
10 }
```

- semantisch äquivalente Fassung als **nebenläufiger Abschnitt**:

```
11 chain_t *pull(chain_t *head) {
12     chain_t *item;
13
14     do if ((item = head->link) == 0) break;
15     while (!CAS(&head->link, item, item->link));
16
17     return item;
18 }
```

- funktional ist das Entfernen und die Kopfzeigeraktualisierung zu leisten
- nur letztere Aktion unterliegt dem wechelseitigen Ausschluss

Umlaufsperre vs. Wiederholung

. . . *rollback* einer Transaktion

- in beiden Fällen können gekoppelte Prozesse ins Kreiseln geraten

  Umlaufsperre
  - gleichzeitige Prozesse kreiseln ohne Nutzen für sich selbst
  - sie kommen in der Schleife nicht mit Berechnungen voran
  - Schleifendauer bedeutet **Wartezeit**
  - **Nutzarbeit** startet mit einer atomaren Aktion (z.B. CAS)

  Transaktion
  - gleichzeitige Prozesse kreiseln mit Nutzen für sich selbst
  - sie kommen in der Schleife mit Berechnungen voran
  - Schleifendauer bedeutet **Nutzarbeitszeit**
  - **Nutzarbeit** endet mit einer atomaren Aktion (z.B. CAS)

- Unkosten des kritischen Abschnitts und der Transaktion abwägen
  - seien t_{ka} die Zeitdauer und t_{lock} die Unkosten des kritischen Abschnitts
  - ferner seien t_{na} die Zeitdauer und $o_{na} = t_{na} - t_{ka}$ die Unkosten des nebenläufigen Abschnitts, $t_{na} \geq t_{ka}$ angenommen
  - sei N die Zahl gekoppelter Prozesse

$$\sum_{n=1}^{N} t_{lock}^{n} < \sum_{n=1}^{N} o_{na}^{n}$$

↪ Umlaufsperren „rechnen“ sich, falls:

↪ Entwicklungsaufwand und Blockierungsnachteile unberücksichtigt. . .

Gliederung

Resümee

- mit dem Konzept der **Umlaufsperre** wird wechselseitiger Ausschluss softwaregesteuert umgesetzt
  - pessimistischer Ansatz zum Schutz kritischer Abschnitte: *leicht*
    - negative Erwartung, dass sich Prozesse gleichzeitig an einer Stelle treffen
    - gekoppelte Prozesse blockieren wahrscheinlich nur für kurze Zeit
  - kritischer Aspekt ist die starke **Störanfälligkeit** bei hohem Wettstreit
    - Häufigkeit von „*read-modify-write*“-Zyklen pro Durchlauf minimieren
    - prozessspezifische Zurückhaltung vom wiederholten Sperrversuch
    - variable Verweilzeiten, um Konflikte bei Wiederholungen zu vermeiden
  - als blockierende Synchronisation besteht hohe **Verklemmungsgefahr**
- im Gegensatz dazu die **nichtblockierende Synchronisation**, bei der wechselseitiger Ausschluss ein Merkmal der Hardware ist
  - optimistischer Ansatz zum Schutz kritischer Abschnitte: *schwer*
    - positive Erwartung, dass Prozesse nicht gleichzeitig zusammentreffen
  - verklemmungsfrei, robust, nichtsequentiell, störunanfällig
- obwohl grundweg verschieden, sind **Spezialbefehle** der Hardware die beiden Konzepten gemeinsame Grundlage: TAS, CAS, FAA

Literaturverzeichnis I

[1] Abramson, N. :
The ALOHA System: Another Alternative for Computer Communication.
In: *Proceedings of the Fall Joint Computer Conference (AFIPS '70)*.
New York, NY, USA : ACM, 1970, S. 281–285

[2] Amdahl, G. M.:
Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities.
In: *Proceedings of the AFIPS Spring Joint Computer Conference (AFIPS 1967)*, AFIPS Press, 1967, S. 483–485

[3] Bryant, R. ; Chang, H.-Y. ; Rosenburg, B. S.:
Experience Developing the RP3 Operating System.
In: *Computing Systems* 4 (1991), Nr. 3, S. 183–216

[4] Dechev, D. ; Pirkelbauer, P. ; Stroustrup, B. :
Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs.
In: *Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2010)*, IEEE Computer Society, 2010. – ISBN 978–1–4244–7083–9, S. 185–192

Literaturverzeichnis II

[5] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

[6] Eswaran, K. P. ; Gray, J. N. ; Lorie, R. A. ; Traiger, I. L.:
The notions of consistency and predicate locks in a database system.
In: *Communications of the ACM* 19 (1976), Nr. 11, S. 624–633

[7] IBM Corporation (Hrsg.):
IBM System/370 Principles of Operation.
Fourth.
Poughkeepsie, New York, USA: IBM Corporation, Sept. 1 1974.
(GA22-7000-4, File No. S/370-01)

[8] Intel Corporation:
XCHG.
In: *x86 Instruction Set Reference*.
Rene Jeschke, Nov. 2015. –
http://x86.renejeschke.de/html/file_module_x86_id_328.html

Literaturverzeichnis III

[9] Jones, M. B.:
What really happened on Mars?
http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html, 1997

[10] Kleinöder, J. ; Schröder-Preikschat, W. :
Virtuelle Maschinen.
In: Lehrstuhl Informatik 4 (Hrsg.): *Systemprogrammierung*.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 5.1

[11] Kung, H.-T. ; Robinson, J. T.:
On Optimistic Methods for Concurrency Control.
In: *ACM Transactions on Database Systems* 6 (1981), Jun., Nr. 2, S. 213–226

[12] Lamport, L. :
A New Solution of Dijkstra's Concurrent Programming Problem.
In: *Communications of the ACM* 17 (1974), Aug., Nr. 8, S. 453–455

[13] Lampson, B. W. ; Redell, D. D.:
Experiences with Processes and Monitors in Mesa.
In: *Communications of the ACM* 23 (1980), Febr., Nr. 2, S. 105–117

Literaturverzeichnis IV

[14] Metcalfe, R. M. ; Boogs, D. R.:
Ethernet: Distributed Packet Switching for Local Computer Networks.
In: *Communications of the ACM* 19 (1976), Jul., Nr. 5, S. 395–404

[15] Schröder-Preikschat, W. :
Semaphore.
In: Lehrstuhl Informatik 4 (Hrsg.): *Concurrent Systems — Nebenläufige Systeme*.
FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien), Kapitel 7

[16] Wilner, D. :
Vx-Files: What really happened on Mars?
Keynote at the 18th IEEE Real-Time Systems Symposium (RTSS '97), Dez. 1997

Kreiseln mit CAS

- CAS als **intrinsische Funktion** des Kompilierers:

```
1  _acquire:
2      movl      4(%esp), %ecx   # get pointer to lock variable
3      movb      $1, %dl         # want to set lock "true"
4  LBB0_1:                      # come here for retry (while)
5      xorl      %eax, %eax      # test value is "false"
6      lock                      # next instruction is atomic
7      cmpxchgb  %dl, (%ecx)     # compare and swap values
8      testb     %al, %al        # check if "true" was read
9      jne       LBB0_1          # if so, retry
10     ret                       # was "false" and is now "true"
```

5–7
- die eigentliche Umsetzung von CAS, abgebildet auf `cmpxchg` (x86)
- wobei `lock` lediglich die Atomarität dieses Befehls erzwingt

- erkennbar sind auch die recht wenigen zusätzlichen Operationen der Umlaufsperre in der Wartephase (Z. 5–9)

Skalierungsproblem (vgl. S. 13: *bus-lock burst*)

Je mehr Prozesse gleichzeitig in die Schleife eintreten, desto länger die nahtlose Sequenz der busatomaren Befehle `cmpxchg`.

Atomar „abrufen und Φ tun“

- sei Φ = „addieren“ $\rightsquigarrow$ FAA (*fetch and add*, vgl. S. 17):

```
1  type FAA(type *ref, type val) {
2      atomic { type aux = *ref; *ref = aux + val; }
3      return aux;
4  }
```

  - Reduktion auf eine atomare, in GCC eingebaute intrinsische Funktion:

```
5  #define FAA __sync_fetch_and_add
```

  - Kompilierung in Assembliersprache (ASM86, AT&T Syntax):

```
6  _acquire: ...
7      movl    16(%esp), %eax  # get pointer to global variable
8      movl    $1, %ecx        # constant term to be added
9      lock                    # make next instruction atomic
10     xaddl   %ecx, (%eax)    # exchange and add operands
11     ...                     # ecx now holds the value fetched
```

- sei Φ = „einspeichern“ $\rightsquigarrow$ FAS (*fetch and store*)
  - ein weiterer, überaus universell verwendbarer Spezialbefehl
  - Reduktion auf eine atomare, in GCC eingebaute intrinsische Funktion:

```
12 #define FAS __sync_lock_test_and_set
```

  - letztlich die Basisoperation, um TAS verfügbar zu machen (vgl. S. 11)

Verweilzeit bestimmen und absitzen

- ein Parameter, der von der **Restlaufzeit** des im kritischen Abschnitt operierenden Prozesses und vom **Wettstreitgrad** abhängt
  - bestenfalls kann dies nur ein gut abgeschätzter **Näherungswert** sein
- minimale **Datentyperweiterung** (vgl. S. 8) für Sperrexemplare:

```
1 typedef volatile struct lock {
2     bool busy;  /* initial: false */
3     long time;  /* duration of critical section */
4 } lock_t;
```

  - `time` ist Zeitwert des günstigsten, mittleren oder schlechtesten Falls
    - *best-*, *mean-* oder *worst-case execution time* (BCET, MCET bzw. WCET)
  - durch **statische Programmanalyse** des betreffenden kritischen Bereichs
- im Vergleich zur Zeitanalyse ist der Zeitverbrauch nahezu einfach:

```
1 void backoff(long time, int rate) {
2     volatile long term = time * rate;
3     while (term--);     /* just spend processor cycles */
4 }
```

  - eine **Schlafsperre** (*sleeping lock*) gäbe hier den Prozessor frei
  - die Unkosten dafür sollten aber in die **effektive Verweilzeit** einfließen ☹

Bedingte Wertezuweisung

- **prozedurale Abstraktion** von CAS:

```
1  _CAS:
2      pushl   %esi              # save non-volatile data
3      movl    12(%esp), %ecx    # get test value
4      movl    16(%esp), %edx    # get target value
5      movl    8(%esp), %esi     # get pointer to shared variable
6      movl    %ecx, %eax        # set up test value for CAS
7      lock                      # next instruction is atomic
8      cmpxchgl %edx, (%esi)     # compare and swap (CAS) values
9      cmpl    %ecx, %eax        # check if CAS succeeded (ZF=1)
10     sete    %al               # expand ZF bit into operand
11     movzbl  %al, %eax         # extend operand value to word size
12     popl    %esi              # restore non-volatile data
13     ret                       # "true" if succeeded, "false" else
```

- die Unkosten (*overhead*) im Vergleich zur intrinsischen Funktion des Kompilieres sind beträchtlich
  - drei Befehle (Z. 6–8 bzw. S. 34, Z. 5–7) gegenüber 12 Befehlen
  - Parameterbe- und -entsorgung sowie Prozeduraufruf kämen hinzu...

Untiefe

Definition (ABA, auch A-B-A)

*The ABA problem is a **false positive** execution of a CAS-based speculation on a shared location L_i. [4, p. 186]*

- CAS wurde erfolgreich ausgeführt, die Transaktion scheint gelungen:
  - i die beiden verglichenen Operanden waren identisch, womit die Gültigkeit einer bestimmten Bedingung behauptet wird (*positive*),
  - ii aber diese Behauptung ist faktisch nicht korrekt (*false*)
- angenommen Prozesse P_1 and P_2 verwenden gleichzeitig Adresse L_i
  - Wert A gelesen von P_1 aus L_i meint einen bestimmten Zustand S_1, aber P_1 wird verzögert, bevor der neue Wert an L_i bestätigt werden kann
  - zwischenzeitlich ändert P_2 den Wert an L_i in B und dann zurück in A, meint damit aber einen neuen globalen Zustand $S_2 \neq S_1$
  - P_1 fährt fort, erkennt, dass in L_i der Wert A steht und agiert aber unter der Annahme, dass der globale Zustand S_1 gilt — was jetzt falsch ist
- die **Kritikalität** solcher falsch positiven Ergebnisse steht und fällt mit dem Problem: `mult` ist unkritisch, nicht aber `push` und `pull`

Stapel mit Wettlaufsituation

- Ausgangszustand der Liste: *head* ⇨ *A* ⇨ *B* ⇨ *C*, *head* ist $\texttt{ref}_{CAS}$:

	$\mathfrak{m}$	Op.	*ref	old	new	Liste	
1.	P_1	pull	*A*	*A*	*B*	unverändert	
2.	P_2	pull	*A*	*A*	*B*	ref ⇨ *B* ⇨ *C*	
3.	P_2	pull	*B*	*B*	*C*	ref ⇨ *C*	
4.	P_2	push	*C*	*C*	*A*	ref ⇨ *A* ⇨ *C*	
5.	P_1	pull	*A*	*A*	*B*	ref ⇨ *B* ⇨ ☹	*A* ⇨ *C* verloren

1. P_1 wird im pull vor CAS verzögert, behält lokalen Zustand bei
2.–4. P_2 führt die drei Transaktionen durch, aktualisiert die Liste
5. P_1 beendet pull mit dem zu 1. gültigen lokalen Zustand

- beachte: das eigentliche Problem ist die Wiederverwendung derselben Adresse, wofür ein **beschränkter Adressvorrat** die Ursache ist
  - in 64-Bit-Systemen ist der Adressvorrat logisch nahezu unerschöpflich...

Kritische Variable mittels „Zeitstempel“ absichern

- **Abhilfe** besteht darin, den umstrittenen Zeiger (nämlich `item`) um einen problemspezifischen **Generationszähler** zu erweitern

  Etikettieren
  - Zeiger mit einem Anhänger (*tag*) versehen
  - Ausrichtung (*alignment*) ausnutzen, z.B.:

$$\begin{aligned} \mathit{sizeof}(\mathit{chain\_t}) &\rightsquigarrow 4 = 2^2 \Rightarrow n = 2 \\ &\Rightarrow \mathit{chain\_t}* \text{ ist Vielfaches von 4} \\ &\Rightarrow \mathit{chain\_t}*_{\mathit{Bits}[0:1]} \text{ immer 0} \end{aligned}$$

  - Platzhalter für n-Bit Marke/Zähler in jedem Zeiger

  DCAS
  - Abk. für (engl.) *double compare and swap*
  - Marke/Zähler als elementaren Datentyp auslegen
    - *unsigned int* hat Wertebereich von z.B. $[0, 2^{32} - 1]$
  - zwei Maschinenworte (Zeiger, Marke/Zähler) ändern

- `push` bzw. `pull` verändern dann den Anhänger bzw. die Marke des Zählers (`item`) mit jedem Durchlauf um eine Generation

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – XI. Stillstand

Wolfgang Schröder-Preikschat

7. Dezember 2021

Agenda

Einführung

Betriebsmittel
- Systematik
- Verwaltung

Systemblockade
- Grundlagen
- Fallbeispiel
- Gegenmaßnahmen

Zusammenfassung

Gliederung

Lehrstoff

- über den **Stillstand** (*stalemate*) gekoppelter Prozesse, hervorgerufen durch fehlkonstruierte oder -geleitete Betriebsmittelzuteilung
  - überkreuzte Anforderungen von Betriebsmitteln
  - „verlorene Abgabe“ produzierter oder im Voraus erworbener Ressourcen
- eine **tödliche Umarmung** (*deadly embrace* [2, S. 73]) solcher direkt oder indirekt voneinander abhängigen Prozesse
  - bedingt durch **Entwurfsfehler**, zu beheben durch Entwurfsänderungen
  - der Schwerpunkt (der Vorlesung) liegt auf **konstruktive Maßnahmen**
- verschiedene Facetten von Verklemmungen, je nach Ausprägung des Wartezustands der gekoppelten Prozesse
  - **Totsperre** (*deadlock*) als kleineres Übel, da erkennbar
  - **Lebendsperre** (*livelock*) als größeres Übel, da nicht erkennbar
- **Gegenmaßnahmen** sind Vorbeugung, Vermeidung oder Erkennung und Erholung von Systemblockaden
  - wobei Vorbeugung als konstruktive Maßnahme verbreitet ist, die anderen (analytische Maßnahmen) dagegen nur bedingt umzusetzen sind

Verklemmung, Blockierung, Stillstand

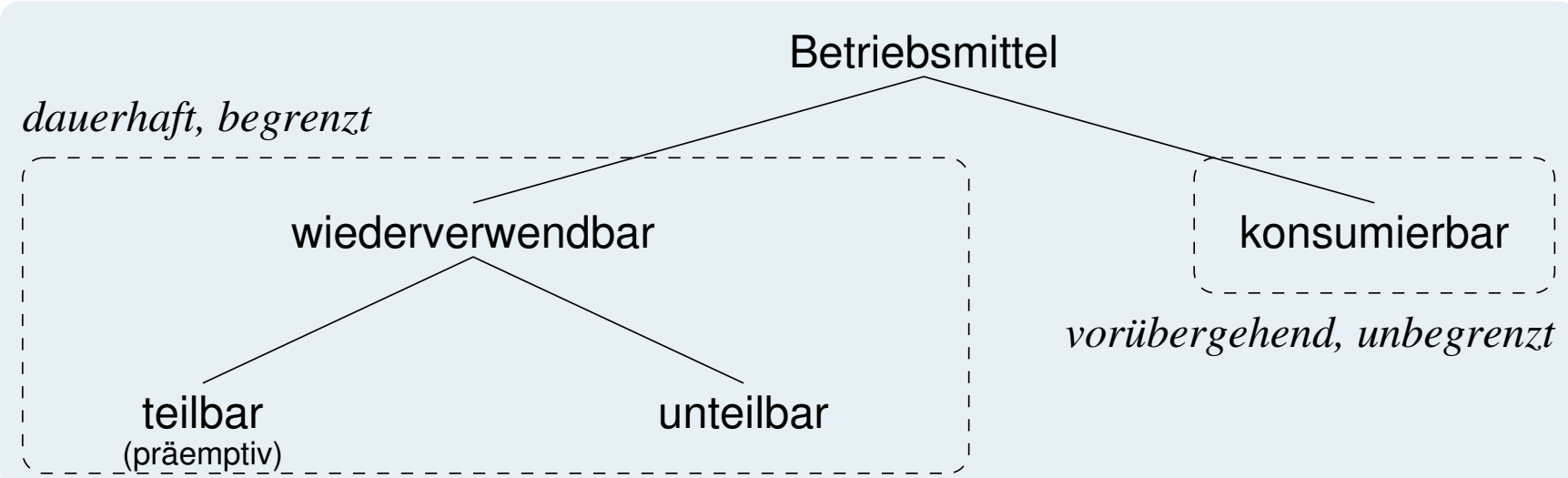
deadlock

source: National Geographic

Gliederung

Betriebsmittelklassifikation

nach [4, 5]



- alle Betriebsmittel werden angefordert, zugeteilt, belegt, benutzt und freigegeben, jedoch wird dabei wie folgt unterschieden:

  wiederverwendbar
  - sie sind (ggf. zeitweilig) **persistent**, nicht flüchtig
  - Anforderung durch mehrseitige Synchronisation
  - Freigabe ermöglicht ihre Wiederverwendung

  konsumierbar
  - sie sind **transient**, flüchtig
  - Anforderung durch einseitige Synchronisation
  - Freigabe führt zu ihrer Entsorgung (Zerstörung)

- entsprechend gestaltet sich der Wettbewerb um sie (*resource contention*)

Ziele

- konfliktfreie Abwicklung der anstehenden Aufträge
- korrekte Bearbeitung der Aufträge in endlicher Zeit
- gleichmäßige, maximierte Auslastung der Betriebsmittel
- hoher Durchsatz, kurze Durchlaufzeit, hohe Ausfallsicherheit

⋮

- **Betriebsmittelanforderung** frei von Verhungern/Verklemmung

  Verhungern
  - andauernde (zeitweilige) Benachteiligung von Prozessen
  - das Prozesssystem macht Fortschritt, steht nicht still

  Verklemmung
  - irreversible gegenseitige Blockierung von Prozessen
  - das Prozesssystem macht keinen Fortschritt, steht still

  - gemeinsames Merkmal ist der **Wettbewerb** um Betriebsmittelzuteilung

- allgemein:
  - Durchsetzung der vorgegebenen Betriebsstrategie
  - eine optimale Realisierung in Bezug auf relevante Kriterien

Aufgaben

- **Buchführung** über die im Rechensystem vorhandenen Betriebsmittel
  - Art, Klasse
  - Zugriffsrechte, Prozesszuordnung, Nutzungszustand und -dauer
- **Steuerung** der Verarbeitung von Betriebsmittelanforderungen
  - Entgegennahme, Überprüfung (z.B. der Zugriffsrechte)
  - Einplanung der Nutzung angeforderter Betriebsmittel durch Prozesse
  - Einlastung (Zuteilung) von Betriebsmittel
  - Entzug oder Freigabe von Prozessen benutzter Betriebsmittel

- **Betriebsmittelentzug**
  - Zurücknahme (*revocation*) der Betriebsmittel, die von einem „aus dem Ruder geratenen“ Prozess belegt werden
  - bei **Virtualisierung** zusätzlich:
    - Rückforderung und Neuzuteilung eines realen Betriebsmittels
    - wobei das zugehörige virtuelle Betriebsmittel dem Prozess zugeteilt bleibt

Verfahrensweisen

- **statisch**
  - vor Laufzeit oder vor einem Laufzeitabschnitt
  - Anforderung aller (im Abschnitt) benötigten Betriebsmittel
  - Zuteilung der Betriebsmittel erfolgt ggf. lange vor ihrer eigentlichen Benutzung
  - Freigabe aller belegten Betriebsmittel mit Laufzeit(abschnitt)ende

↪ Risiko einer nur suboptimalen Auslastung der Betriebsmittel

- **dynamisch**
  - zur Laufzeit, in beliebigen Laufzeitabschnitten
  - Anforderung des jeweils benötigten Betriebsmittels bei Bedarf
  - Zuteilung des jeweiligen Betriebsmittels erfolgt „im Moment“ seiner Benutzung
  - Freigabe eines belegten Betriebsmittels, sobald kein Bedarf mehr besteht

↪ Risiko der **Verklemmung** von abhängigen Prozessen

Gliederung

Stillstand von Prozessen

Definition (deadly embrace)

Eine Situation, in der gekoppelte Prozesse gegenseitig die Aufhebung einer Wartebedingung entgegensehen, diese aber durch Prozesse eben dieses Systems selbst aufgehoben werden müsste.

- die Bedingung sagt etwas zur Verfügbarkeit eines Betriebsmittels aus
  - unabhängig von der Art des Betriebsmittels erwarten gekoppelte Prozesse die Versorgung durch entsprechende Aktionen gleichgestellter Prozesse
  - da jedoch alle Prozesse so handeln, wird kein Betriebsmittel verfügbar
- nach [1] kann die „tödliche Umarmung“ von Prozessen entstehen:
  - i obwohl kein einziger Prozess mehr als die insgesamt verfügbare Menge von Betriebsmitteln benötigt und
  - ii unabhängig davon, ob Betriebsmittelzuteilung in der Verantwortlichkeit des Betriebssystems oder des Anwendungsprogramms selbst liegt
- das Warten kann **inaktiv** (*deadlock*) oder **aktiv** (*livelock*) geschehen
  - d.h., mit oder ohne Abgabe des Betriebsmittels „Prozessor“

Wartezustand „inaktiv“

Definition ([7, S. 235] ↦ zeitabhängiges Fehlverhalten)

Mit **Verklemmung** (*deadlock*) bezeichnet man einen Zustand, in dem die beteiligten Prozesse wechselseitig auf den Eintritt von Bedingungen warten, die nur durch andere Prozesse in dieser Gruppe selbst hergestellt werden können.

- die „tödliche Umarmung“ gekoppelter Prozesse im **Schlafzustand**
  - die Befehlszähler der verklemmten Prozesse bleiben (für die meiste Zeit) konstant und warten meint für den einzelnen Prozess:

tief
  - er ist im Zustand „blockiert“, das erwartete Ereignis ist definiert
  - er gibt den Prozessor zu Gunsten anderer Prozesse ab
    - mit Ausnahme eines „blockiert“ fortschreitenden Leerlaufprozesses
    - der Prozessor ist in Wartestellung bis ein Prozess „bereit“ wird
- **gutartig**, das kleinere von zwei (inaktiv, aktiv) Übeln
  - wenn nicht vorgebeugt oder vermieden, so kann das erkannt werden
  - sofern das Ereignis, worauf ein Prozess wartet, bekannt/definiert ist
  - Abgrenzung von sich in Bewegung befindlichen Prozessen ist machbar

Wartezustand „aktiv“

livelock

Definition (Lebendsperre)

Eine der Verklemmung sehr ähnliche Situation, in der gekoppelte Prozess zwar nicht im Zustand „blockiert“ sind, sie jedoch dennoch keinen Fortschritt bei der Programmausführung erzielen.

- die „tödliche Umarmung“ gekoppelter Prozesse im **Wachzustand**
  - die Befehlszähler der verklemmten Prozesse verändern sich fortwährend und warten meint für den einzelnen Prozess:

beschäftigt
- er bleibt im Zustand „laufend“, hält den Prozessor

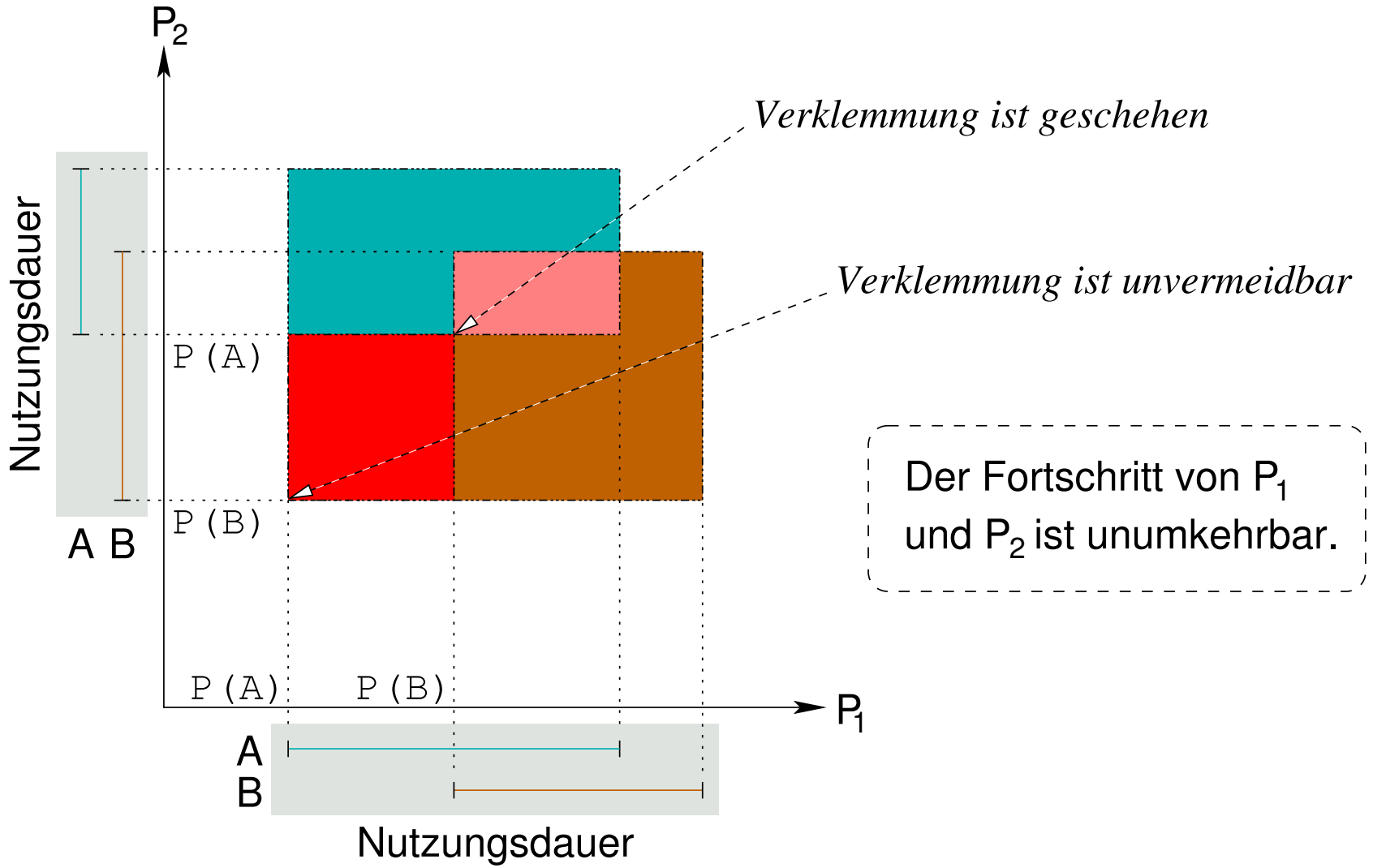
träge
- er alterniert zwischen den Zuständen „laufend“ und „bereit“
- er gibt den Prozessor zu Gunsten anderer Prozesse ab

- **bösartig**, das größere von zwei (inaktiv, aktiv) Übeln
  - wenn nicht vorgebeugt oder vermieden, so kann das nicht erkannt werden
  - keine Abgrenzung von normalen, voranschreitenden Prozessen[1]

[1] Wie häufig und lange sollte etwa geprüft werden, dass sich Befehlszähler von Prozessen in welchen Wertebereichen bewegen?

Zustandsraum

nach [1, p. 70/71]



Der Fortschritt von P_1 und P_2 ist unumkehrbar.

Kontoführung

- **Banküberweisung** (*bank transfer*) eines gewissen Betrags von einem Konto auf ein anderes Konto

```
1  void transfer(account_t *from, account_t *to, double amount) {
2      claim(from, to);          /* acquire account data records */
3      from->value -= amount;    /* withdraw from one account */
4      to->value += amount;      /* credit the other account */
5      clear(from, to);          /* release account data records */
6  }
```

  - dabei seien die Konten **wiederverwendbare Betriebsmittel** (Software)
  - die vom Überweisungsprozess zeitweilig **unteilbar** benutzt werden müssen

- für die Überweisungsoperation sei folgender **Datensatz** (*data record*) eines Kontos angenommen:

```
7  typedef struct account {
8      double value;          /* actual balance */
9      semaphore_t count;     /* safeguard for account management */
10 } account_t;
```

  - **wechelseitiger Ausschluss** sichert den Überweisungsprozess ab
  - hierzu wird ein **zählender Semaphor** mit Initialwert 1 verwendet

Betriebsmittelverwaltung

- **Inanspruchnahme** der für den Transfer benötigten Datensatzobjekte:

```
1  void claim(account_t *from, account_t *to) {
2      P(&from->count);      /* acquire source account */
3      P(&to->count);        /* acquire target account */
4  }
```

  - jedes dieser Objekte ist nur in einfacher Ausfertigung verfügbar
  - zu einem Zeitpunkt wird nur ein Prozess ein Exemplar belegen können

- **Bereitstelllung** der beiden Objekte zur Wiederverwendung:

```
5  void clear(account_t *from, account_t *to) {
6      V(&to->count);        /* release target account */
7      V(&from->count);      /* release source account */
8  }
```

  - Programmierkonvention sei paarweise Verwendung von `claim` und `clear`
  - d.h., es werden nie mehr Objekte bereitgestellt als angefordert wurden

- in dem Szenario verbirgt sch eine **wettlaufkritische Aktionsfolge**
  - die zur Verklemmung von „überweisungswilligen“ Prozessen führen kann

Wettlaufkritische Aktionsfolge

- Ausgangssituation:
  - sei P_i ein Überweisungsprozess, wobei $1 \leq i \leq 3$ verschiedene von diesen Prozessen gleichzeitig geschehen
  - sei K_j, $j > 1$, ein Konto, jeweils repräsentiert durch ein Datensatzobjekt
- Szenario (mit α, β und γ beliebige Geldbeträge):

$P_1 : \mathit{transfer}(K_1, K_2, \alpha)$	durchläuft `claim`, belegt K_1, wird verdrängt
$P_2 : \mathit{transfer}(K_2, K_3, \beta)$	durchläuft `claim`, belegt K_2, wird verdrängt
$P_3 : \mathit{transfer}(K_3, K_1, \gamma)$	durchläuft `claim`, belegt K_3, wird verdrängt
$P_1 : \mathit{transfer}(K_1, K_2, \alpha)$	fährt fort, fordert $K_2 \mapsto P_2$ an, blockiert
$P_2 : \mathit{transfer}(K_2, K_3, \beta)$	fährt fort, fordert $K_3 \mapsto P_3$ an, blockiert
$P_3 : \mathit{transfer}(K_3, K_1, \gamma)$	fährt fort, fordert $K_1 \mapsto P_1$ an, blockiert

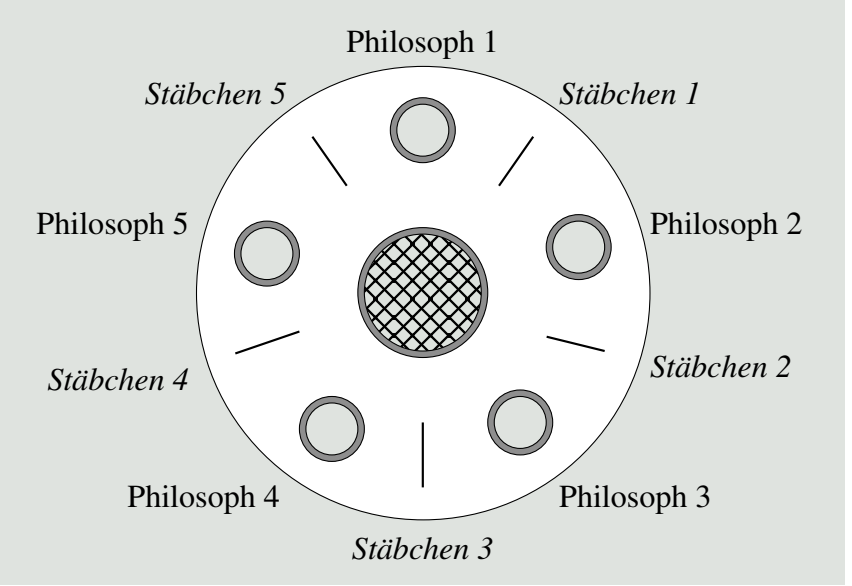
- diese Aktionsfolge bedeutet also:
  - P_1 wartet auf die Zuteilung von K_2, das von P_2 belegt wird
  - P_2 wartet auf die Zuteilung von K_3, das von P_3 belegt wird
  - P_3 wartet auf die Zuteilung von K_1, das von P_1 belegt wird
- d.h., P_1, P_2 und P_3 sind untereinander verklemmt ☹
  - keiner der Prozesse kann sich von selbst aus dieser Situation befreien...

Speisende Philosophen I

Hinweis

Fünf Philosophen, die nichts anderes zu tun haben, als zu denken und zu essen, sitzen an einem runden Tisch. Denken macht hungrig — also wird jeder Philosoph auch essen. Dazu benötigt ein Philosoph jedoch stets beide neben seinem Teller liegenden Stäbchen.

- Philosoph
  - Prozess
  - ↪ Überweisung
- Stäbchen
  - Betriebsmittel
  - ↪ Konto



- **Stillstand**:
  - alle Philosophen nehmen zugleich das eine (linke) Stäbchen auf
  - anschließend greifen sie auf das andere (rechte) zu

↪ die Überweisungsprozesse fordern zugleich das jeweilige Quellkonto an, belegen es und fordern anschließend das Zielkonto an

Speisende Philosophen II

- Philosophendasein:

```
1  void phil(int who) {
2    int any = check(who);
3    while (any > 0) {
4      think();
5      claim(any);
6      munch();
7      clear(any);
8    }
9  }

10 void think() { ... }
11 void munch() { ... }
```

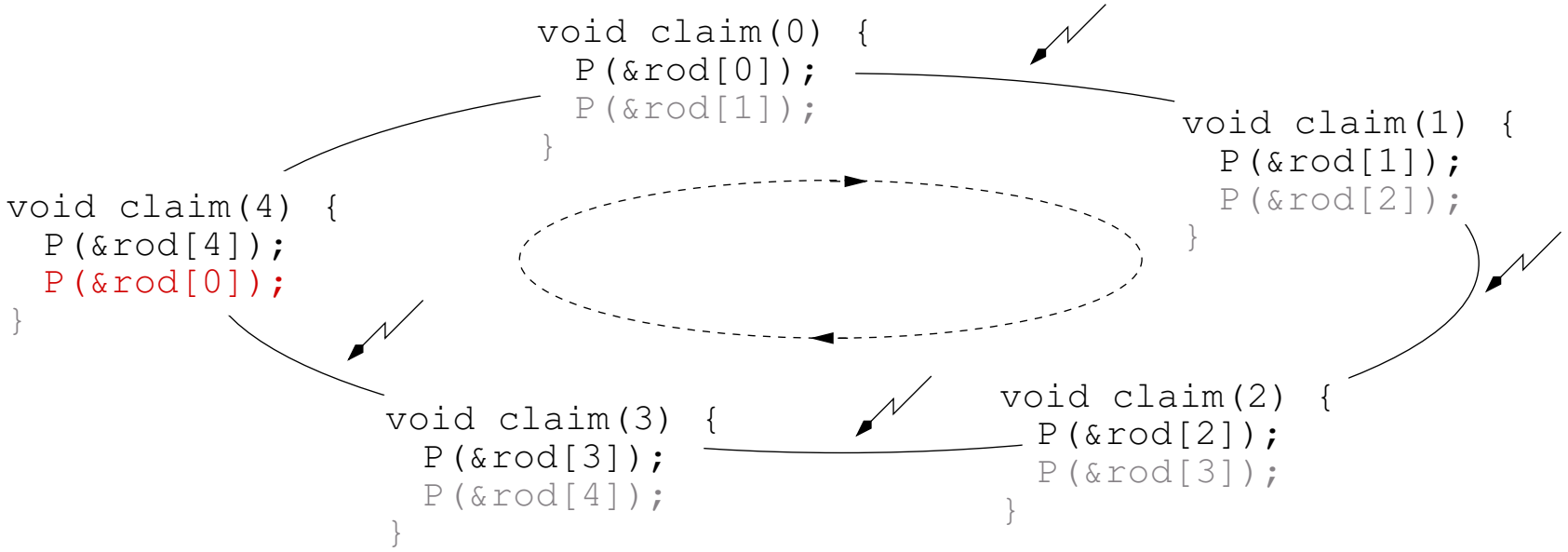
- altbekanntes Muster der Betriebsmittelzuteilung
  - vgl. S. 17
  - *P* vergibt zu einem Zeitpunkt nur ein Stäbchen (*rod*), umgekehrt gibt *V* entsprechend nur eins frei

- Stäbchenverwaltung:

```
12 semaphore_t rod[NPHIL] = {
13     { 1 }, { 1 }, { 1 }, { 1 }, { 1 }
14 };
15
16 int check(int who) {
17     ...          /* validate who */
18     return who - 1;
19 }
20 void claim(unsigned slot) {
21     P(&rod[slot]);
22     P(&rod[(slot + 1) % NPHIL]);
23 }
24
25 void clear(unsigned slot) {
26     V(&rod[(slot + 1) % NPHIL]);
27     V(&rod[slot]);
28 }
```

Speisende Philosophen III

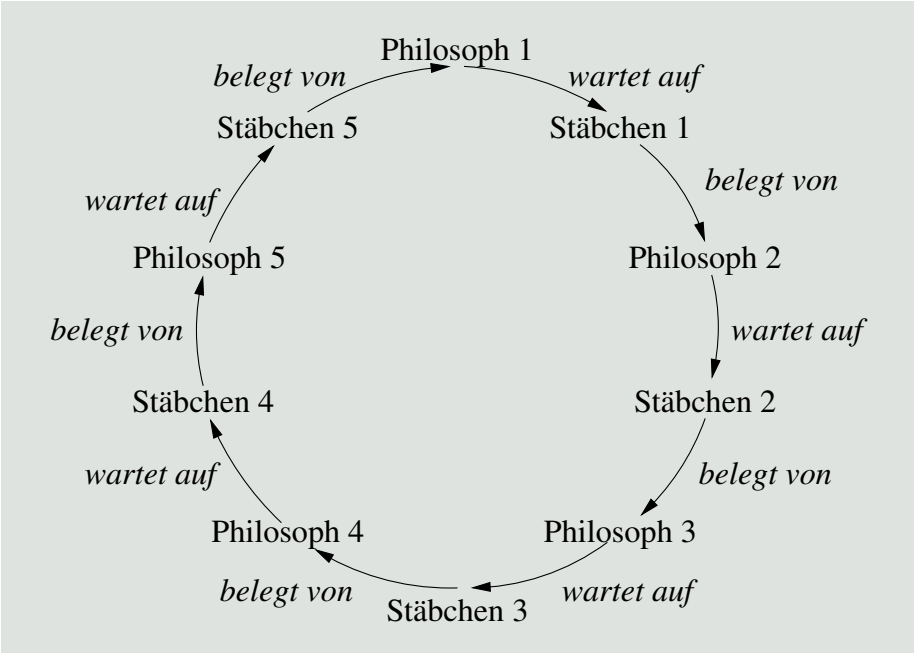
- sei P_i, $1 \leq i \leq 5$, Philosoph i, der Stäbchen S_i und S_{i+1} benötigt
  - mit $r = i - 1$ als Index im Wertebereich $[0, 4]$ für das Stäbchenfeld *rod*



1. P_1 nimmt $S_1 \mapsto r_0$ auf, wird vor Aufnahme von $S_2 \mapsto r_1$ gestört
2. P_2 nimmt $S_2 \mapsto r_1$ auf, wird vor Aufnahme von $S_3 \mapsto r_2$ gestört
3. P_3 nimmt $S_3 \mapsto r_2$ auf, wird vor Aufnahme von $S_4 \mapsto r_3$ gestört
4. P_4 nimmt $S_4 \mapsto r_3$ auf, wird vor Aufnahme von $S_5 \mapsto r_4$ gestört
5. P_5 nimmt $S_5 \mapsto r_4$ auf, fordert $S_1 \mapsto r_0$ an und muss warten
   - in Folge fordern alle anderen P_i, $1 \leq i < 4$, ihr zweites Stäbchen an... ☹

Speisende Philosophen IV

- für jedes belegte Betriebsmittel ist der **Eigentümer** bekannt
- für jeden wartenden Prozess ist der **Blockadegrund** bekannt
- der **Wartegraph** (S. 31) zeigt die Verklemmungssituation
  - ist das Ergebnis der Analyse der Betriebsmittelbelegung
  - wird im Bedarfsfall aufgebaut



↪ ein geschlossener Kreis (im Wartegraphen) erfasst all jene Prozesse, die sich zusammen im ***Deadlock*** befinden.

↪ es muss sichergestellt sein, dass ein solcher Kreis entweder nicht entstehen oder dass er erkannt und „durchbrochen“ werden kann

Voraussetzungen für Verklemmungen

nach [1, p. 70]

Hinweis (Lebendsperre)

Nachfolgendes gilt auch für Stillstand im aktiven Wartezustand.

- **notwendige Bedingungen** für die gekoppelten Prozesse:
  1. **Ausschließlichkeit** in der Betriebsmittelnutzung (*mutual exclusion*)
  2. **Nachforderung** eines oder mehrerer Betriebsmittel (*hold and wait*)
  3. **Unentziehbarkeit** (*no preemption*) der zugeteilten Betriebsmittel
- **notwendige und hinreichende Bedingung**:
  4. **zirkulares Warten** muss eingetreten sein
     - jeder der Prozesse hält eins oder mehrere Betriebsmittel, die einer oder mehrere andere Prozesse in der Kette angefordert haben

Hinweis (Vorbeugung/Vemeidung)

*Jede dieser Bedingungen muss zu einem Zeitpunkt erfüllt sein, damit Prozesse verklemmen. Die Aufhebung nur einer dieser Bedingungen resultiert in ein **verklemmungsfreies System** von Prozessen.*

Verklemmungsvorbeugung

deadlock prevention

Hinweis (Prävention)

Vorsorgemaßnahmen, damit gekoppelte Prozesse gar nicht erst eine Verklemmung entwickeln können.

- **indirekte Methoden**, eine der notwendigen Bedingungen aufheben
  1. nichtblockierende Synchronisation, Atomarität auf tieferer Ebene nutzen
  2. alle benötigten Betriebsmittel unteilbar anfordern
  3. Betriebsmittel virtualisieren, damit den Entzug der realen Betriebsmittel ermöglichen, nicht jedoch ihrer virtuellen Gegenstücke
- **direkte Methoden**, notwendige & hinreichende Bedingung aufheben
  4. eine lineare Ordnung von Betriebsmittelklassen definieren, die zusichert, dass Betriebsmittel R_i vor R_j zuteilbar ist, nur wenn $i < j$

Hinweis (Prophylaxe)

*Jede dieser Methoden steht für eine **konstruktive Maßnahme**, die Aufbau und Struktur nichtsequentieller Programme beeinflusst.*

Betriebsmittel unteilbar anfordern

- Beispiel zum Bankwesen (vgl. S. 17):

```
1  void claim(account_t *from, account_t *to) {
2      static semaphore_t mutex = {1};
3      P(&mutex);                    /* enter critical section */
4      P(&from->count);              /* demand */
5      P(&to->count);                /* additional demand */
6      V(&mutex);                    /* leave critical section */
7  }
```

- Beispiel zum Philosophenproblem (vgl. S. 20):

```
1  void claim(unsigned slot) {
2      static semaphore_t mutex = {1};
3      P(&mutex);                    /* enter critical section */
4      P(&rod[slot]);                /* demand */
5      P(&rod[(slot + 1) % NPHIL]);/* additional demand */
6      V(&mutex);                    /* leave critical section */
7  }
```

- die **Nachforderungsbedingung** wurde insofern aufgehoben, als dass An- und Nachforderung (Z. 4–5) nunmehr unteilbar geschehen

Atomarität auf tieferer Ebene nutzen

- Beispiel zum Bankwesen (vgl. S. 16):
  - Umstrukturierung, **prozedurale Abstraktion** der kritischen Aktionsfolge:

```
1 void transfer(account_t *from, account_t *to, double amount) {
2     issue(from, -amount);    /* withdraw from one account */
3     issue(to, amount);       /* credit the other account */
4 }
```

  - Abbildung auf eine problemspezifische **Elementaroperation**:

```
5 inline void issue(account_t *this, double amount) {
6     FAA(&this->balance, amount);
7 }
```

  - Reduktion auf eine atomare, in GCC eingebaute intrinsische Funktion:

```
8 #define FAA __sync_fetch_and_add
```

- die **Nachforderungsbedingung** wurde abgebildet auf nur noch eine Anforderung des Betriebsmittels „kritischer Abschnitt“: **FAA**
  - die Verwendung eines binären Semaphors anstelle von FAA ginge ebenso, jedoch wäre dies bei weitem nicht so effizient, wie mit FAA

↪ vollständig semantisch äquivalent im Vergleich zur Vorlage (S. 16) ist die Lösung jedoch nicht, da Datensatzobjekte ungeschützt sind

Verklemmungsvermeidung

deadlock avoidance

Hinweis

***Vorabwissen** zu Prozessen und ihren Betriebsmittelanforderungen.*

- Vereitelung „tödlicher Umarmung“ durch **strategische Methoden**
  - kein Versuch wird unternommen, eine notwendige Bedingung aufzuheben
  - vielmehr verhindert **laufende Anforderungsanalyse** zirkulares Warten
- Prozesse und ihre Betriebsmittelanforderungen werden überwacht
  - jede Anforderung prüft auf einen möglichen **unsicheren Zustand**
  - sollte ein solcher möglich sein, wird die **Zuteilung abgelehnt**
  - anfordernden Prozess suspendieren $\rightsquigarrow$ langfristige Planung [6, S. 19]
- Betriebsmittelzuteilung erfolgt nur im **sicheren Zustand**
  - im Falle einer **Prozessfolge**, die alle zukünftigen Anforderungen erfüllt
  - in Anbetracht aller aktuellen Belegungen und anstehenden Freigaben

Hinweis (Vermeidung)

*Jede Methode, die Verklemmungen „vermeidet“, ist eine **analytische Maßnahme** zur Laufzeit nichtsequentieller Programme.*

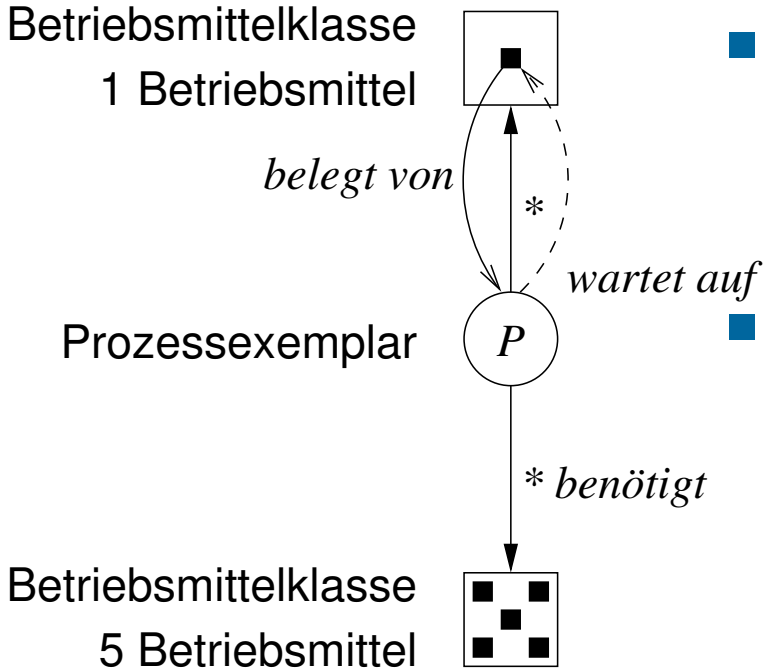
Bestimmung des unsicheren Zustands

- Ansatz **Betriebsmittelgraph** (S. 29)
  - definiert einen **Mengenkontrakt** für Prozessexemplare bezüglich Bedarf und aktueller Vergabe von Betriebsmitteln bestimmter Klassen
  - mit Vorabwissen angelegt und initialisiert bei der Prozesserzeugung und fortgeschrieben mit jeder Betriebsmittelanforderung
  - anhaltende Analyse hinsichtlich möglicher **Zyklenbildung** im Graphen
- Ansatz **Bankiersalgorithmus** (*banker's algorithm* [2])
  - Annahme ist, dass das System die Menge von Betriebsmitteln kennt, die:
    - i jeder Prozess möglicherweise anfordert (*maximum claim*: Bedarf),
    - ii jeder Prozess gegenwärtig hält (*allocated*: Belegung) und
    - iii noch nicht allen Prozessen zugewiesen wurde (*available*: Verfügungsrahmen)
  - **sicherer Zustand** ist, wenn eine Anforderung folgendes nicht übersteigt:
    - (a) den Bedarf des anfragenden Prozesses und
    - (b) den Verfügungsrahmen von Betriebsmitteln der angefragten Klasse
  - scheitert (a), wird die Anforderung zurückgewiesen, scheitert (b), wird der anfordernde Prozess suspendiert $\rightsquigarrow$ langfristige Planung [6, S. 19]
- nicht nur die Erfordernis von **Vorabwissen** erweist sich als ein großes Problem, auch die Skalierbarkeit von Methode und Implementierung

Betriebsmittelgraph

resource allocation graph, RAG

- ein **gerichteter Graph**, der Prozessexemplare und Betriebsmittel oder Betriebsmittelklassen zusammenhängend darstellt
  - eine vom Betriebssystem zu verwaltende **dynamische Datenstruktur**



  - optionales (Vorab-) Wissen, um die *benötigt*-Beziehung zu bilden:
    - Betriebsmittelklassen und den jeweiligen Betriebsmittelbedarf
  - obligatorisches Wissen bezüglich aller Prozesse/Betriebsmittel:
    - für jeden Prozess gibt es eine Liste zugeteilter Betriebsmittel (*benötigt*)
    - jedes Betriebsmittel verbucht den Besitzerprozess (*belegt von*)
- schließlich noch (obligatorisches) Wissen, um aus dem RAG einen **Wartegraphen** abzuleiten zu können:
  - für jeden Prozess ist vermerkt, worauf er wartet (*wartet auf*)

Verklemmungsnachweis und Erholung *deadlock detection*

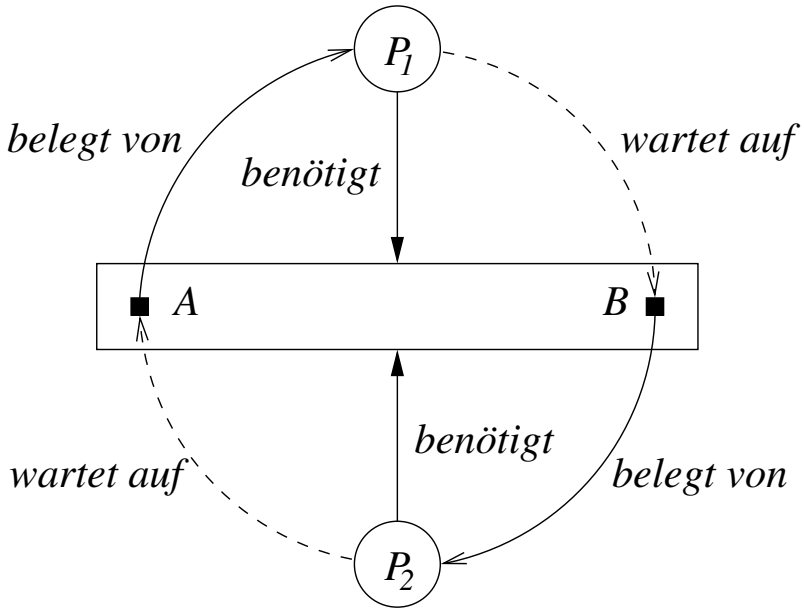
- Prozessverklemmungen werden stillschweigend in Kauf genommen
  - kein Versuch wird unternommen, eine der vier Bedingungen aufzuheben
  - stattdessen läuft die **sporadische Suche** nach blockierten Prozessen
    - ein **Wartegraph** (S. 31) wird aufgebaut und nach Zyklen abgesucht
    - Grundlage dafür bildet der **Betriebsmittelgraph** (S. 29)
- erkannte Zyklen werden im nachgeschalteten Schritt durchbrochen
  - eine Option ist die **Prozesszerstörung** eines einzelnen, ausgewählten Exemplars oder aller Exemplare im Zyklus
  - die andere Option ist **Betriebsmittelentzug**, durch Auswahl eines Opfers und anschließendem **Zurückrollen** des besitzenden Prozesses
- gegebenenfalls wiederholt sich der ganze Vorgang, solange nicht alle Zyklen aufgelöst werden konnten

Gratwanderung zwischen Schaden und Aufwand

Schaden macht klug, aber zu spät. (Sprichwort)

Wartegraph

wait-for graph



Hinweis (Erzeugung)

Wenn das Betriebssystem den Verklemmungsfall für wahrscheinlich hält:

- *Antwortzeitzunahme*
- *Durchsatzabnahme*
- *Leerlaufzeitanstieg*

- seien A und B Betriebsmittel derselben Klasse:
  1. P_1 vollzieht $P(A)$, A ist frei und wird P_1 zugeteilt
  2. P_2 vollzieht $P(B)$, B ist frei und wird P_2 zugeteilt
  3. P_1 vollzieht $P(B)$, B ist belegt $\rightsquigarrow$ P_1 muss auf $V(B)$ durch P_2 warten
  4. P_2 performs $P(A)$, A ist belegt $\rightsquigarrow$ P_2 muss auf $V(A)$ durch P_1 warten
- ein **Zyklus** von P_1 nach P_2 über A und B, hin und zurück
  - P_1 und P_2 befinden sich im ***Deadlock***...

Gliederung

Resümee

- **Betriebsmittel** zeigen sich als Entitäten von Hardware und Software
  - wiederverwendbar
    - begrenzt verfügbar: teilbar, unteilbar
  - konsumierbar
    - unbegrenzt verfügbar
- Ziele, Aufgaben und Verfahrensweise der **Betriebsmittelverwaltung**
  - Betriebsmittelanforderung frei von Verhungern/Verklemmung
  - Buchführung der Betriebsmittel, Steuerung der Anforderungen
  - statische/dynamische Zuteilung von Betriebsmitteln
- für eine Verklemmung müssen **vier Bedingungen** gleichzeitig gelten
  - exklusive Belegung, Nachforderung, kein Entzug von Betriebsmitteln
  - zirkulares Warten der die Betriebsmittel beanspruchenden Prozesse
  - nicht zu vergessen: Verklemmung bedeutet „*deadlock*“ oder „*livelock*“
- die **Gegenmaßnahmen** sind:
  - Vorbeugen, Vermeiden, Erkennen & Erholen
  - die Verfahren können im Mix zum Einsatz kommen
- Verfahren zum **Vermeiden/Erkennen** sind eher praxisirrelevant
  - sie sind kaum umzusetzen, zu aufwendig und damit schlecht einsetzbar
  - Vorherrschaft sequentieller Programmierung macht sie verzichtbar. . .

Literaturverzeichnis I

[1] Coffman, Jr., E. G. ; Elphick, M. J. ; Shoshani, A. :
System Deadlocks.
In: *Computing Surveys* 3 (1971), Jun., Nr. 2, S. 67–78

[2] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

[3] Habermann, A. N.:
Prevention of System Deadlocks.
In: *Communications of the ACM* 12 (1969), Jul., Nr. 7, S. 373–377/385

[4] Holt, R. C.:
On Deadlock in Computer Systems.
Ithaca, NY, USA, Cornell University, Diss., 1971

[5] Holt, R. C.:
Some Deadlock Properties of Computer Systems.
In: *ACM Computing Surveys* 4 (1972), Sept., Nr. 3, S. 179–196

Literaturverzeichnis II

[6] Kleinöder, J. ; Schröder-Preikschat, W. :
Einplanungsgrundlagen.
In: Lehrstuhl Informatik 4 (Hrsg.): *Systemprogrammierung*.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 9.1

[7] Nehmer, J. ; Sturm, P. :
Systemsoftware: Grundlagen moderner Betriebssysteme.
dpunkt.Verlag GmbH, 2001. –
ISBN 3–898–64115–5

Sicherer Zustand

nach [3]

- sei P_k ein sequentieller Prozess
- sei S eine geordnete Menge solcher Prozesse
- sei b_k der Betriebsmittelanspruch eines Prozesses, P_k
- sei $s(k)$ die Ordnungszahl von $P_k \in S$
- sei $r(t)$ die Anzahl verfügbarer Betriebsmittel zum Zeitpunkt t
- sei $c_k(t)$ die Anzahl der P_k zur Zeit t zugeteilten Betriebsmittel
- dann gilt ein Zustand als sicher, wenn es eine vollständige Folge in S gibt, so dass:

$$\forall_{P_k \in S} b_k \leq r(t) + \sum_{s(l) \leq s(k)} c_l(t) \tag{1}$$

Condition (1) says that the claim by process P_k must not exceed the sum of the free resources and those resources which will become free "in due time," when the processes preceding in S have released theirs. [3, p. 375]

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – XII.1 Speicherverwaltung: Adressräume

Wolfgang Schröder-Preikschat

14. Dezember 2021

Agenda

Einführung
- Rekapitulation

Adressräume
- Real
- Logisch
- Virtuell

Mehradressraumsysteme
- Virtualität
- Exklusion
- Inklusion

Zusammenfassung

Gliederung

Lehrstoff

- **Adressräume** detailiert behandeln, um Bedeutungen der jeweiligen Ausprägungen erfassen zu können
  - real: manifestiert im **Adressraumbelegungsplan** des Herstellers
  - logisch: abstrahiert von diesem Plan, aber nicht vom Speicher
  - virtuell: abstrahiert von der Speicherlokalität (Vorder-/Hintergrund)
- gängige[1] **Adressumsetzungstechniken** vorstellen und vertiefen, um die logische/virtuelle Adresse auf eine reale abbilden zu können
  - seitennummeriert/gekachelt, Seitentabelle und -deskriptor
  - explizit oder implizit segmentiert, Segmenttabelle und -deskriptor
  - segmentiert und seitennummeriert, als Kombination beider Techniken
  - inkl. **Übersetzungspuffer** zur Latenzverbergung bei der Abbildung
- Modelle von **Mehradressraumsysteme** kennenlernen und sie in ihren nichtfunktionalen Eigenschaften differenzieren können
  - Exklusion: total private Adressräume für alle Programme
  - Inklusion: partiell private Adressräume für Maschinenprogramme

[1] Weitere siehe [4, S. 37–43].

[6, S. 9]

Referenzfolge

reference string

- ein **„laufender“ Prozess** [5, S. 19] generiert Folgen von Adressen auf den Haupt-/Arbeitsspeicher, und zwar:
  - i nach Vorschrift des Programms, das diesen Prozess spezifiziert, wie auch
  - ii in Abhängigkeit von den Eingabedaten für den Programmablauf
- der **Wertevorrat** dieser Adressen ist aber stets gemäß Programm in der Größe nach oben begrenzt
  - er ist initial statisch und gibt die zur Programmausführung mindestens erforderliche Menge an Haupt-/Arbeitsspeicher vor
  - jedoch gestaltet er sich zur Laufzeit dynamisch, nimmt zu und kann dabei aber den „einem Prozess zugebilligten“ Wertevorrat nicht überschreiten
  - letzteres sichert der Kompilierer (typsichere Programmiersprache) oder das Betriebssystem (in Zusammenspiel mit der MMU) zu
- der einem Prozess zugebilligte Wertevorrat gibt den **Adressraum** vor, in dem dieser Prozess (logisch/physisch) eingeschlossen ist
  - der Prozess kann aus seinem Adressraum normalerweise nicht ausbrechen und folglich nicht in fremde Adressräume eindringen
  - der **Prozessadressraum** hat eine maximale, hardwarebeschränkte Größe

Adressraumlehre I

- Befehlssatzebene (Ebene$_2$)

Definition (realer Adressraum)

Der durch einen Prozessor definierte Wertevorrat $A_r = [0, 2^n - 1]$ von Adressen, mit $e \leq n \leq 64$ und (norm.) $e \geq 16$. Nicht jede Adresse in A_r ist jedoch gültig, d.h., A_r kann Lücken aufweisen.

- der **Hauptspeicher** ist adressierbar durch einen oder mehrere Bereiche in A_r, je nach Hardwarekonfiguration

- Maschinenprogrammebene (Ebene$_3$)

Definition (logischer Adressraum)

Der in Programm P definierte Wertevorrat $A_l = [n, m]$ von Adressen, mit $A_l \subset A_r$, der einem Prozess von P zugebilligt wird. Jede Adresse in A_l ist gültig, d.h., A_l enthält *konzeptionell* keine Lücken.

- führt **Arbeitsspeicher** ein, der linear adressierbar ausgelegt ist und durch das Betriebssystem auf den Hauptspeicher abgebildet wird

Adressraumlehre II

- Maschinenprogrammebene (Ebene$_3$)

Definition (virtueller Adressraum)

$A_v = A_l$: A_v übernimmt alle Eigenschaften von A_l. Jedoch nicht jede Adresse in A_v bildet ab auf ein im Hauptspeicher liegendes Datum.

  - Benutzung einer solchen nicht abgebildeten Adresse in A_v verursacht in dem betreffenden Prozess einen **Zugriffsfehler**
  - der Prozess erfährt eine **synchrone Programmunterbrechung** (*trap*), die vom Betriebssystem behandelt wird
  - das Betriebssystem sorgt für die **Einlagerung** des adressierten Datums in den Hauptspeicher und
  - der Prozess wird zur **Wiederholung** der gescheiterten Aktion gebracht
- der durch A_v für den jeweiligen Prozess benötigte Hauptspeicher ist „nicht in Wirklichkeit vorhanden, aber echt erscheinend“
  - jedoch steht jederzeit genügend Arbeitsspeicher für A_v zur Verfügung
    - einesteils im Hauptspeicher, anderenteils im Ablagespeicher (*swap area*)
  - der Arbeitsspeicher ist eine virtuelle, der Hauptspeicher eine reale Größe

Gliederung

Adressraumbelegung

address-space assignment

- ein **Adressraumbelegungsplan** bestimmt, welche Hardwareeinheiten über welche Adressen oder Adressbereiche zugreifbar sind
  - nicht nur **Speicher** (RAM, ROM), sondern auch **Peripheriegeräte**

Adressbereich	Größe (KiB)	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

- die konkrete Auslegung gibt der **Hersteller des Rechensystems** vor, nicht der Hersteller (Intel) des Prozessors

Ungültige Adressen

- im **Adressraumbelegungsplan** finden sich auch Adressbereiche, mit denen keine Hardwareeinheiten assoziiert sind
  - der Zugriff darauf ist undefiniert oder liefert einen **Busfehler** (*bus error*)

Adressbereich	Größe (KiB)	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

- **zeitgenössische Hardware** lässt den Zugriff mit ungültigen Adressen nicht undefiniert, sondern unterbricht den zugreifenden Prozess (*trap*)

Reservierte Adressen

- aber auch Adressbereiche mit speziellem Verwendungszweck sind im **Adressraumbelegungsplan** festgeschrieben
  - der Zugriff darauf bedeutet eine **Zugriffsverletzung** (*access violation*)

Adressbereich	Größe (KiB)	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

- typischerweise sind dies Adressbereiche, die der **residente Monitor** (BIOS, *Open Firmware* (OFW), EFI) sein Eigen nennt

Freie Adressen

- schließlich benennt der **Adressraumbelegungsplan** Adressbereiche, die dem allgemeinen Verwendungszweck unterliegen
  - der Zugriff darauf kann einen **Schutzfehler** (*protection fault*) liefern

Adressbereich	Größe (KiB)	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

- der **Hauptspeicher** (*main memory*), in dem Betriebssystem und die Maschinenprogramme (in Gänze/Teilen) zeitweilig liegen

Abstraktion von der realen Adressraumbelegung

- ein **logischer Adressraum** beschreibt die geradlinige Beschaffenheit des Hauptspeichers eines (schwergewichtigen) Prozesses
  - Hauptspeicher getrennter realer Adressbereiche wird **linear adressierbar**
  - zuzüglich speicherabgebildeter (*memory mapped*) Entitäten der Hardware
    - hier insb. Geräteregister zur Interaktion mit Peripheriegeräten
    - d.h., zur speicherabgebildeten Ein-/Ausgabe (*memory-mapped I/O*)
- er umfasst alle für einen Prozess gültigen Text- und Datenadressen, entsprechend des durch ihn ablaufenden Programms
  - auf Programmiersprachenebene bezugnehmend auf mind. zwei Segmente
    - Text – Maschinenanweisungen, Programmkonstanten
    - Daten – initialisierte Daten, globale Variablen, Halde
  - auf Maschinenprogrammebene mindestens ein weiteres Segment
    - Stapel – lokale Variablen, Hilfsvariablen, aktuelle Parameter
    - andere – Gemeinschaftsbibliotheken (*shared libraries*) oder -einrichtungen
- festgelegt durch das **Adressraummodell** (S. 26) des Betriebssystems und von letzteres abgebildet auf den realen Adressraum
  - mittels MMU (*memory management unit*), die dazu eine gekachelte bzw. seitennummerierte oder segmentierte Organisationsstruktur definiert

Eindimensionaler Adressraum

Seitennummerierung *steht für eine Unterteilung des Adressraums in gleichgroße Einheiten und deren* ***lineare Aufzählung****.*

- je nach Adressraumtyp werden diese Einheiten verschieden benannt

  Seite (*page*) im logischen/virtuellen Adressraum

  Seitenrahmen (*page frame*), auch Kachel, im realen Adressraum
- die vom Prozess generierte lineare Adresse *la* ist ein Tupel (p, o):
  - p ist eine **Seitennummer** (*page number*) im Adressraum $[0, 2^N - 1]$
    - Wertebereich für $p = [0, (2^N \texttt{ div } 2^O) - 1]$
  - o ist der **Versatz** (*offset*, *displacement*) innerhalb von Seite p
    - Wertebereich für $o = [0, 2^O - 1]$

  ↪ mit $O << N$ und 2^O auch Seitengröße (in Bytes): typisch ist $2^{12} = 4096$
- tabellengesteuerte Abbildung von *la* mit p als **Seitenindex**
  - **Seitentabelle** (*page table*) von sogenannten **Seitendeskriptoren**
    - auch **Seiten-Kachel-Tabelle**, ein „dynamisches Feld“
    - wobei ggf. mehrere solcher Felder pro Prozessexemplar angelegt sind

Seitendeskriptor

page descriptor

- ein von der Hardware (MMU) vorgegebener **Verbund** von Daten, der statische/dynamische **Seiteneigenschaften** beschreibt:

  Kachel-/Seitenrahmennummer
  - seitenausgerichtete reale Adresse

  Attribute
  - Schreibschutzbit
  - Präsenzbit (an-/abwesend)
  - Referenzbit[2]
  - Modifikationsbit[2]

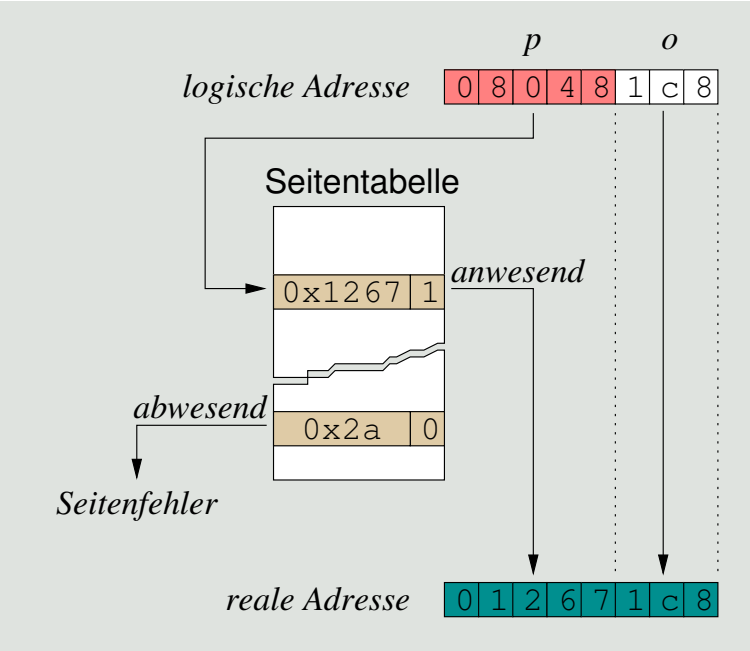
  - je nach Hardware und Adressraummodell gibt es weitere Attribute
    - Privilegstufe, Seiten(rahmen)größe, Spülungssteuerung (TLB), ...

- Betriebssysteme definieren pro Seitendeskriptor oft Attribute, die im **Schatten** der Seitentabelle gehalten werden müssen
  - Seitendeskriptor des Betriebssystems in der „*shadow page table*“
  - für allgemeine Verwaltungsaufgaben, aber auch speziellen Funktionen
    - Seitenersetzung (bei virtuellem Speicher), dynamisches Binden
    - *copy on write* (COW), *copy on reference* (COR)

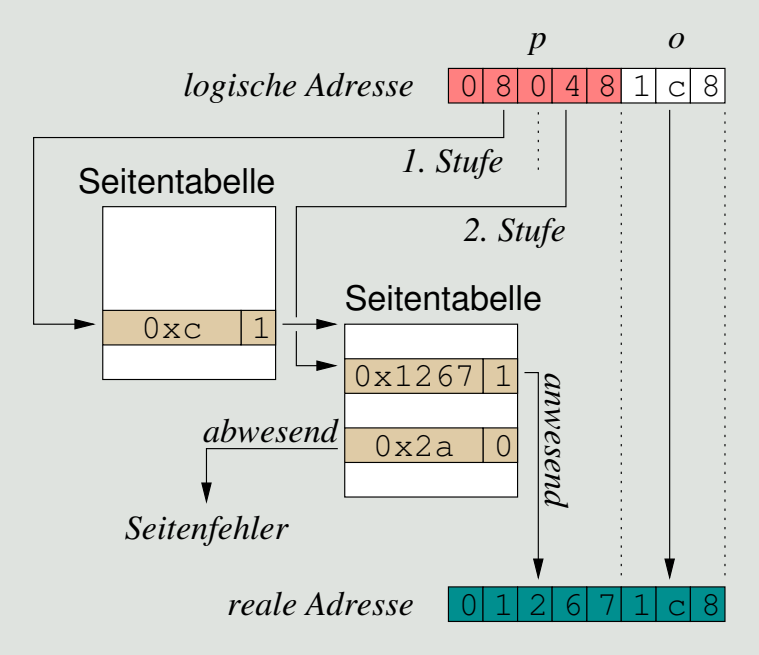
[2] „klebriges“ (*sticky*) Bit: wird von Hardware gesetzt aber nicht gelöscht.

Adressabbildung I

- angenommen, die CPU dereferenziert die Adresse 0x080481c8:
  - einstufige Abbildung



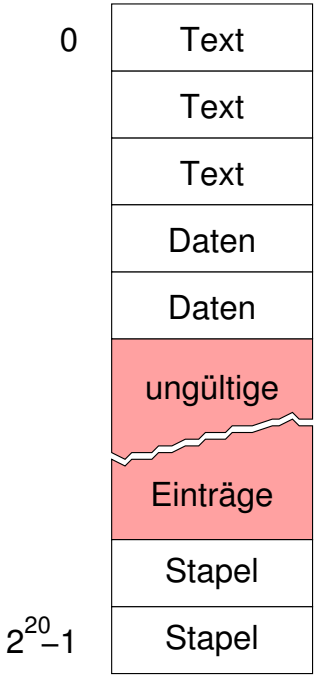
  - zweistufige Abbildung (x86)



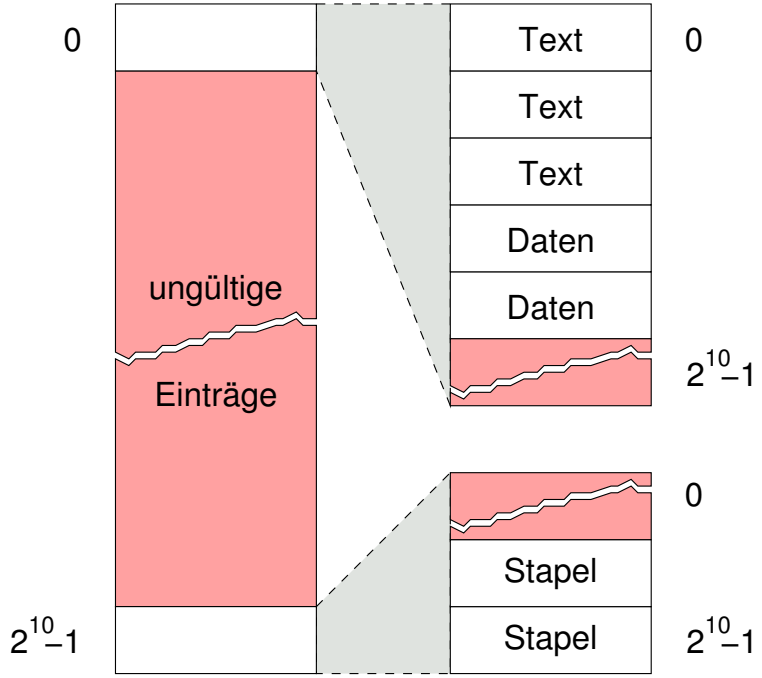
  - *base/limit* Registerpaar (MMU) grenzt die Seitentabelle ggf. ein
  - je nach Prozessexemplar ggf. verschieden große Seitentabellen
  - *base* Register (MMU) lokalisiert die Seitentabelle der 1. Stufe
  - gleich große Seitentabellen, für alle Prozessexemplare
  - ***Trap***, falls $p \geq$ *limit* (li.) oder ungültiger/leerer Seitendeskriptor (beide)

Adressabbildung ohne Tabelleneingrenzung

- ein Prozess belege 12 KiB Text, 8 KiB Daten und 8 KiB Stapel
  - 4 KiB Seiten, 32-Bit Seitendeskriptor, 32-Bit Adresse/Prozessor
- einstufige Tabelle



  - $2^{20} - 7$ ungültige Einträge
  - 1 Tabelle pro Prozessexemplar
  - 4 MiB Speicherplatzbedarf ☹
- zweistufige Tabelle



  - $3 * 2^{10} - 9 = 3063$ ungültige Ein.
  - 3 Tabellen pro Prozessexemplar
  - 12 KiB Speicherplatzbedarf ☺

Zweidimensionaler Adressraum

Segmentierung *meint die Unterteilung des Adressraums in Einheiten von möglicherweise verschiedener Größe, sogenannter* ***Segmente****, die ihrerseits gleichgroße Einheiten linear aufgezählt enthalten.*

- jedes Segment S bildet eine lineare Folge fester Speichereinheiten:

  Byte
  - S ist unbedingt zusammenhängend, auch im realen Adressraum

  Seite
  - S ist unbedingt zusammenhängend im logischen Adressraum, aber
  - bedingt zusammenhängend im realen Adressraum

  $\hookrightarrow$ **seitennummerierte Segmentierung** (*paged segmentation*)

- die vom Prozess generierte Adresse *la* bildet ein Paar (S, D):
  - S ist **Segmentname** (auch Segmentnummer) $\rightsquigarrow$ 1. Dimension
    - Wertebereich für $S = [0, 2^M - 1]$; bei IA-32: $M = 13$
  - D ist **Verschiebung** (*displacement*) im Segment S $\rightsquigarrow$ 2. Dimension
    - Wertebereich für $D = [0, N - 1]$

- tabellengesteuerte Abbildung von *la* mit S als **Segmentindex**
  - selektiert den für S gültigen **Segmentdeskriptor** in der **Segmenttabelle**

Segmentdeskriptor

segment descriptor

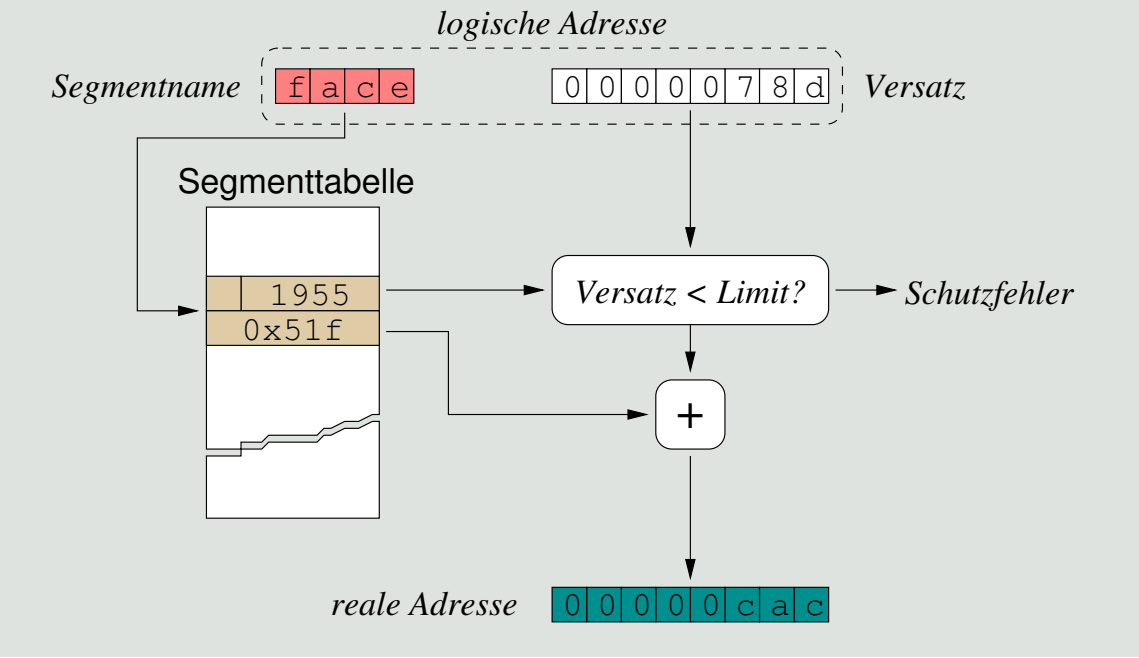
- ein von der Hardware (MMU) vorgegebener **Verbund** von Daten, der statische/dynamische **Segmenteigenschaften** beschreibt:
  - Basis
    - Segmentanfangsadresse im Haupt- bzw. Arbeitsspeicher
    - Ausrichtung (*alignment*) entsprechend der Granulatgröße
  - Limit
    - Segmentlänge als Anzahl der Granulate
    - Zahl der gültigen, linear aufgezählten Granulatadressen
  - Attribute
    - Typ (Text, Daten, Stapel)
    - Zugriffsrechte (lesen, schreiben, ausführen)
    - Expansionsrichtung (auf-/abwärts)
    - Präsenzbit
- je nach Hardware und Adressraummodell gibt es weitere Attribute
  - Privilegstufe, Klasse (*interrupt*, *trap*, *task*), Granulatgröße, ...
  - Seiten-Kachel-Tabelle (seitennummerierte Segmentierung)

Hinweis

Ursprünglich war Segmentierung eine Technik, um mehr Hauptspeicher adressieren zu können, als es durch die Adressbreite allein möglich war. Ein prominentes Beispiel dafür war/ist der i8086: 16-Bit breite Adresse, jedoch $A_r = [0, 2^{20} - 1]$.

Adressabbildung II

- angenommen, die CPU dereferenziert die Adresse `0x78d` im Segment namens `0xface` ⇝ **zweikomponentige Adresse**:

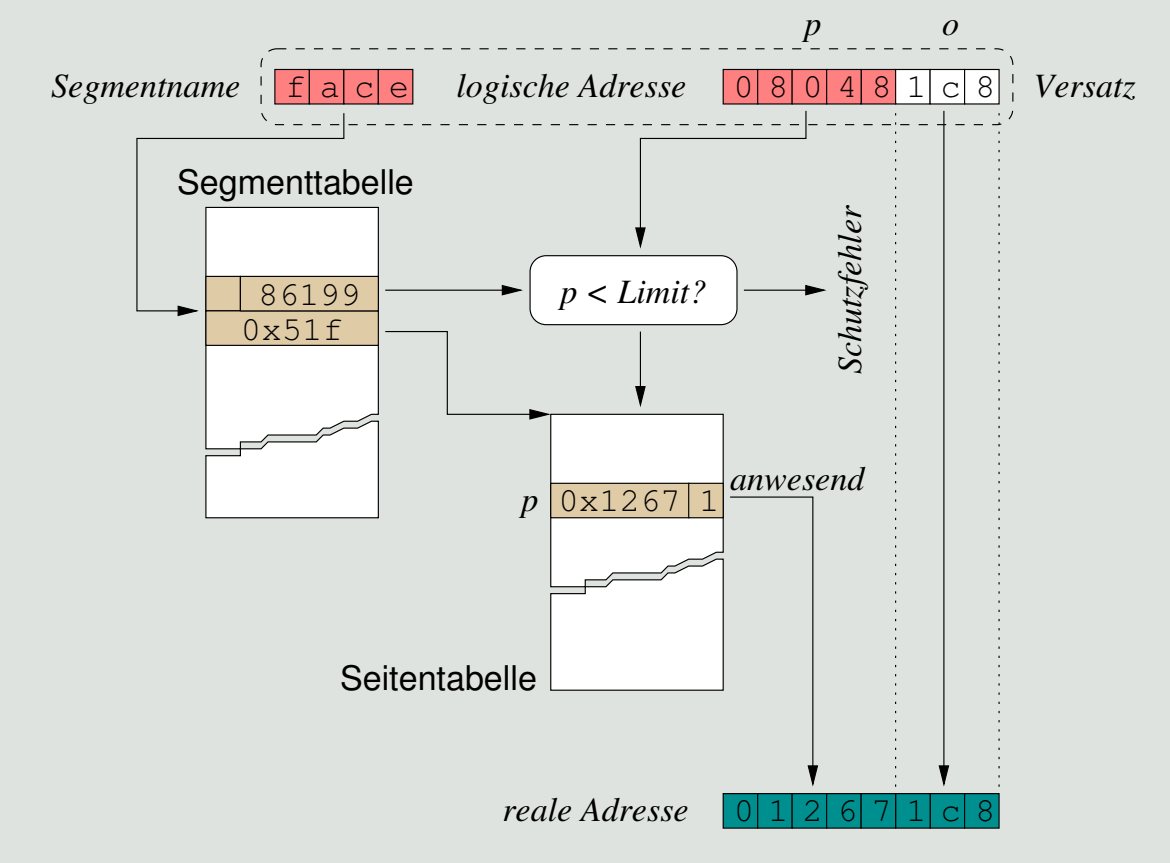


- evtl. Ausnahmen beim Abbildungsvorgang:
  - Schutzfehler (s.u.)
  - Segmentfehler wegen Abwesenheit: *swap-out*
  - Zugriffsverletzung (falsche Rechte)
- bewirken einen ***Trap***, den die MMU erzeugt

- der Tabelleneintrag $face_{16} = 64206_{10}$ liefert den Segmentdeskriptor
- die Adresse ist ein Versatz zur Segmentanfangsadresse im Hauptspeicher
  - sie ist gültig, wenn ihr Wert kleiner als die Segmentlänge ist und
  - wird dann zur Segmentanfangsadresse addiert ⇝ **Verlagerung** (*relocation*)
  - ansonsten ist sie ungültig ⇝ **Schutzfehler** (*segmentation fault*)

Adressabbildung III

- **seitennummerierte Segmentierung** (*paged segmentation*):

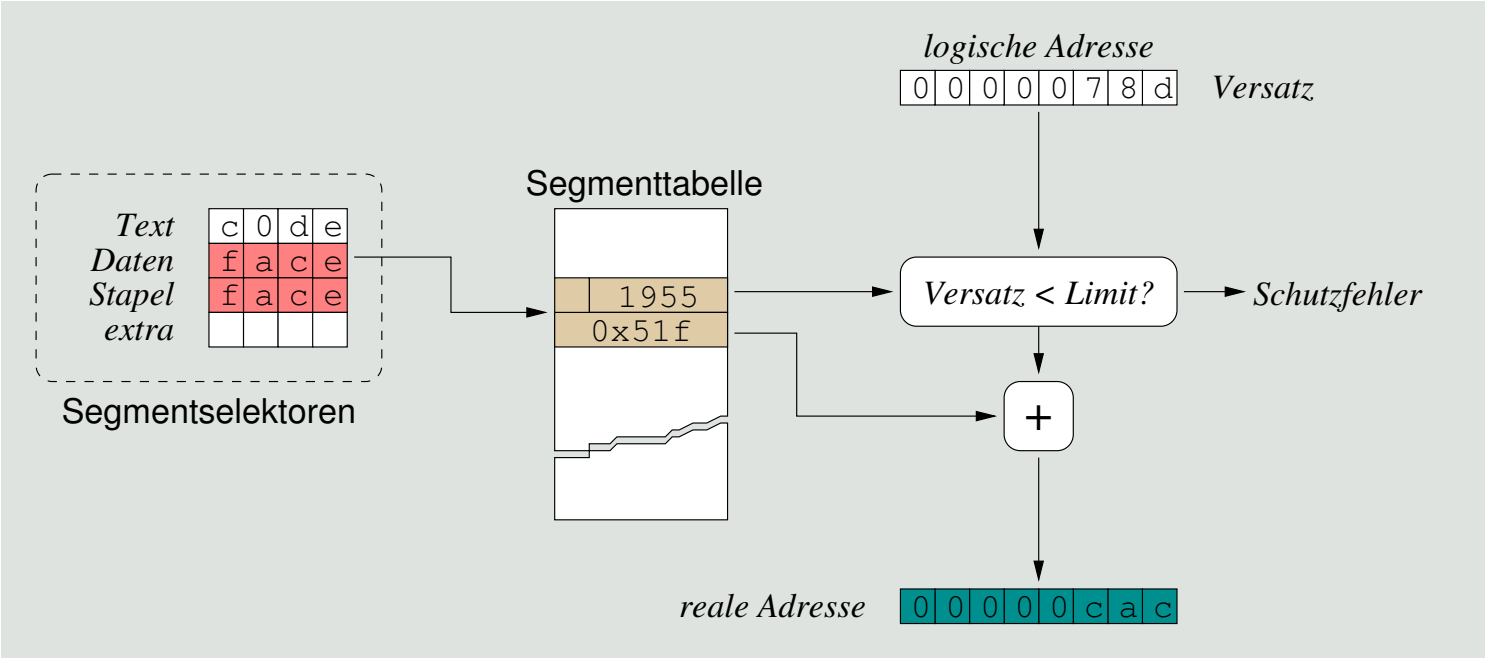


- gekachelte Segmente, grobgranular
- keine mehrstufigen Seitentabellen
- Seitentabellen verschiedener Größen
- globale oder lokale Segmenttabellen
  - pro System oder
  - pro Prozessexemplar
- nicht unkompliziert. . .

- ein Segment erfasst letztlich eine Seitentabelle bestimmter Größe
  - die Segmentanfangsadresse lokalisiert die Seitentabelle im Hauptspeicher
  - die Segmentlänge definiert die Größe der Seitentabelle

Adressabbildung IV

- je nach **Zugriffsart** des in Ausführung befindlichen Befehls selektiert die MMU implizit das passende Segment:



- Befehlsabruf (*instruction fetch*) aus Textsegment
  - Operantionskode ↦ Segmentname „Text“
- Operandenabruf (*operand fetch*) aus Text-, Daten-, Stapelsegment
  - Direktwerte ↦ Segmentname „Text“
  - globale/lokale Daten ↦ Segmentname „Daten“ ≡ „Stapel“

↪ Programme können weiterhin **einkomponentige Adressen** verwenden

Übersetzungspuffer

translation lookaside buffer (TLB)

Jede Dereferenzierung bedeutet den indirekten Zugriff über eine oder mehrere Tabellen im Hauptspeicher, der impraktikabel ist, wenn keine Vorkehrung zur ***Latenzverbergung*** *getroffen wird.*

- **Zwischenspeicher** (*cache*) für das Übersetzungsergebnis, d.h., einer Unter- oder Obermengen von Deskriptoren[3]
  - Assoziativspeicher für eine kleine Anzahl (8 – 128) von Puffereinträgen
  - Segment- bzw. Seitenindex (der virtuellen Adresse) als Suchschlüssel
- ein **Umsetzungsfehler** (*lookup miss*) führt zur **Tabellenwanderung** (*table walk*), die hard- oder softwaregeführt geschieht

  hardwaregeführt
  - die CPU läuft die Tabellen ab
  - mittelbarer Trap, bei erfolgloser Tabellenwanderung

  ↪ in Hardware implementierte MMU (x86, PPC)

  softwaregeführt
  - das Betriebssystem läuft die Tabellen ab
  - unmittelbarer Trap der CPU, beim Umsetzungsfehler

  ↪ in Software implementierte MMU (MIPS, Alpha)
- letztere hat eine höhere **Auffüllzeit**, aber auch höhere **Flexibilität** [12]

[3] Erstmalig umgesetzt in IBM System/370 [3, 1].

Abstraktion von der Speicherlokalität

Eine virtuelle Adresse erbt alle Eigenschaften einer logischen Adresse und erlaubt darüber hinaus ***ortstransparente Zugriffe*** *auf externen Speicher — desselben oder eines anderen Rechensystems.*

- **lose Bindung** zwischen Adresse und durch sie adressierten Entität:

  logische Adresse
  - entkoppelt von der Lokalität im **Hauptspeicher**
  - ermöglicht **dynamisches Binden** aktiver Prozesse
  - erlaubt **Tauschen** (*swapping*) inaktiver Prozesse

  virtuelle Adresse
  - ist eine logische Adresse und geht darüber hinaus, sie:
  - entkoppelt von der Lokalität im **Arbeitsspeicher**
  - erlaubt **Seitenumlagerung** (*paging*) aktiver Prozesse

- die Adressabbildung impliziert **partielle Interpretation** der Zugriffe
  - steuerndes Mittel ist das **Präsenzbit** eines Segments/einer Seite
    - 0 – abwesend, verursacht einen **Zugriffsfehler** (*access fault*) $\rightsquigarrow$ ***Trap***
    - 1 – anwesend, unterbricht die Dereferenzierung nicht
- **Ausnahmebehandlung** und Wiederaufnahme des Prozesses
  - das Betriebssystem sorgt für die Anwesenheit des Segments/der Seite und
  - die CPU wird instruiert, den unterbrochenen Befehl zu wiederholen

Gliederung

Adressvirtualisierung

Definition (Virtualität [13])

Die Eigenschaft einer Sache, nicht in der Form zu existieren, in der sie zu existieren scheint, aber in ihrem Wesen oder ihrer Wirkung einer in dieser Form existierenden Sache zu gleichen.

- **Virtualisierung des realen Adressbereichs** – nicht Hauptspeichers!
  - i Vervielfachung von $A = [0, 2^N - 1]$
    - komplett A für Betriebssystem und allen Maschinenprogrammen, jeweils
  - ii Einrichtung von $A_t = [0, 2^N - 1]$, Vervielfachung von $A_p \subset A_t$
    - komplett A_t (total) für das Betriebssystem
    - komplett A_p (partiell) für alle Maschinenprogramme, jeweils
- Adressen dieser Bereiche sind nicht wirklich (physisch), wohl aber in ihrer Funktionalität vorhanden
  - hinter jeder dieser Adresse steht eine speicherabbildbare Entität
  - sie referenzieren Entitäten der Programmtexte (d.h., Befehle) oder -daten

Private Adressräume

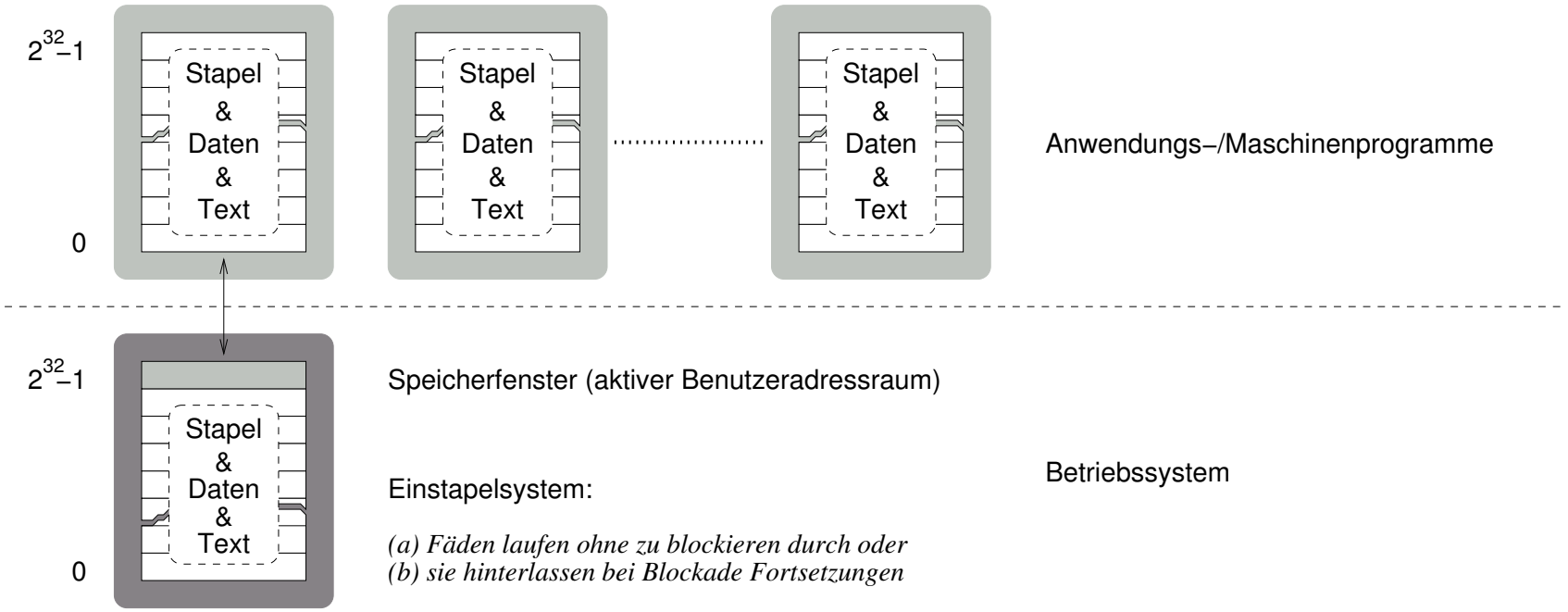
Illusion *von einem eigenen physischen Adressraum für Betriebssystem und Maschinenprogramme* $\rightsquigarrow$ ***Exklusion***

- **Vervielfachung** des Adressbereichs $A = [0, 2^N - 1]$
  - wobei N bestimmt ist durch die reale Adressbreite des Prozessors
  - evolutionär betrachtet galt/gilt z.B. für $N = 16, 20, 24, 31, 32, 48, 64$
- **Spezialhardware**[4] verhindert ein Ausbrechen von Prozessen aus A
  - dies gilt für alle durch das Betriebssystem verwalteten Prozesse, *also*
  - sowohl für Maschinenprogramme als auch für das Betriebssystem selbst
- evtl. Datenaustausch zwischen Programmen erfordert **Spezialbefehle**
  - des Betriebssystems für die Maschinenprogramme *und* $\mapsto$ Ebene 3
    - Systemaufrufe zur Interprozesskommunikation oder Adressbereichsabbildung
  - der CPU für die Betriebssystemprogramme $\mapsto$ Ebene 2
    - privilegierte Befehle zum Lese-/Schreibzugriff auf den Benutzeradressraum
- im Vordergrund steht die **strikte Isolation** von ganzen Adressräumen

[4] MMU, aber ebenso eine MPU (*memory protection unit*).

Programmierte Mitbenutzung

partiell



- zeitgenössisch **fensterbasierter Ansatz** zum Datenaustausch
  - horizontal
    - Interprozesskommunikation (Nachrichtenversenden)
    - seitenbasierte Mitbenutzung
  - vertikal
    - Zugriffe auf den Benutzeradressraum durch **Speicherfenster**
- prominentes Beispiel eines Betriebssystems der Art:
  - OS X
    - Hybridkernansatz, ereignis-/prozedurbasiert, 512 MiB Fenster

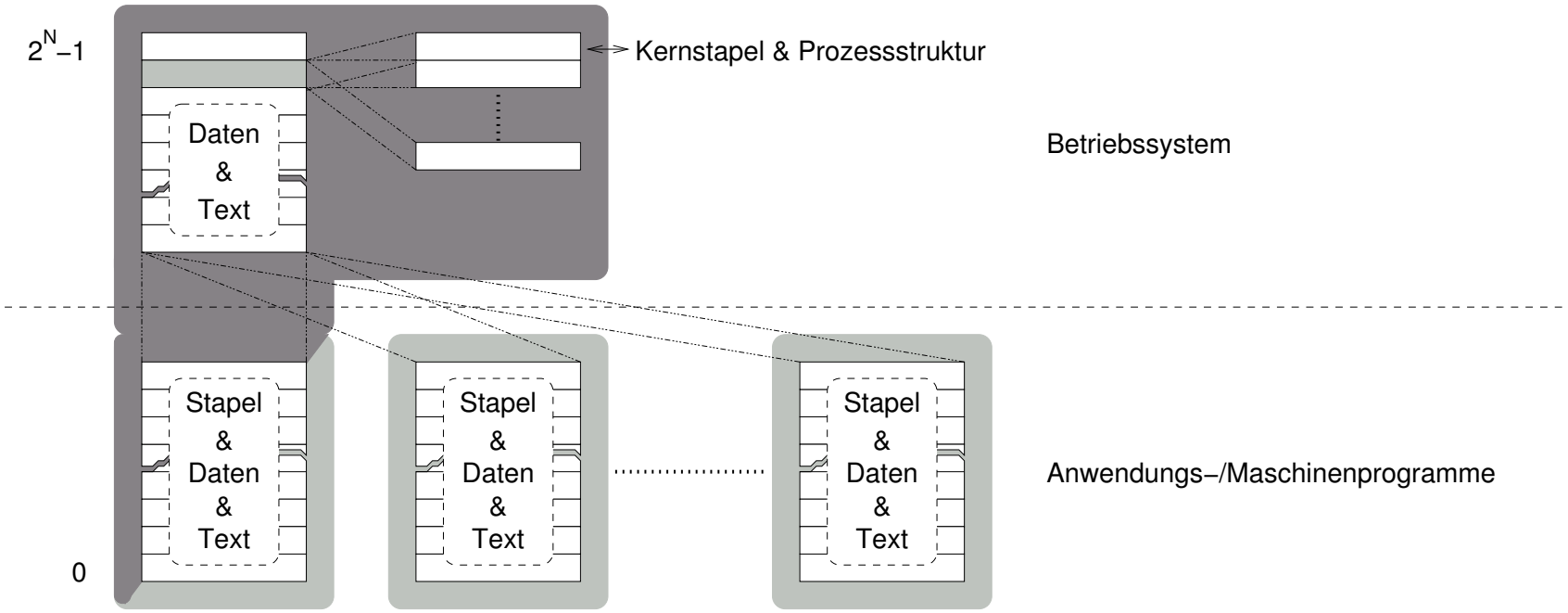
Partiell private Adressräume

Illusion *von einem eigenen physischen Adressraum bzw. -bereich für die Maschinenprogramme* $\rightsquigarrow$ ***Inklusion*** *des Betriebssystem(kern)s*

- **Vervielfachung** des Adressbereichs $A_p \subset A_t$
  - A_t ist der dem Betriebssystem *total* zugeordnete Adressbereich
    - existiert einfach, aber mit A_p als integrierten variablen (mehrfachen) Anteil
  - A_p ist der einem Maschinenprogramm in A_t *partiell* zugeordnete Bereich
    - existiert mehrfach, einmal für jedes Anwendungs- bzw. Maschinenprogramm
- der Benutzeradressraum ist ein Teil (genauer: eine echte Teilmenge) des Betriebssystemadressraums
  - die MMU verhindert ein Ausbrechen von Prozessen aus A_p und A_t, nicht jedoch deren Eindringen heraus aus $A_t - A_p$ und hinein in A_p
    - bedingter Schreibschutz von A_p für A_t dämmt Betriebssystemfehler ein
  - dabei erstreckt sich A_p über den oberen oder unteren Bereich von A_t
  - ein Prozesswechsel zwischen A_p bedingt das Umschalten der MMU
- im Vordergrund steht, **weniger Adressraumwechsel** hervorzurufen

Implizite Mitbenutzung

gekachelt



- zeitgenössisch **partitionsbasierter Ansatz** zum Datenaustausch
  - horizontal: Interprozesskommunikation, Segment-/Seitenmitbenutzung
  - vertikal: **speicherabgebildeter Zugriff** auf den Benutzeradressraum
- prominente Beispiele von Betriebssystemen der Art:
  - Linux: monolithisch, prozedurbasiert
  - OS X, NT: Hybridkernansatz, prozess-/prozedurbasiert

Partitionierung des Adressbereichs

Inklusion des Benutzeradressraums in den Betriebssystemadressraum ist nur bei hinreichend großem N ein sinnvoller Ansatz

- das Modell wurde attraktiv mit Adressbreiten von $N \geq 30$ Bits
  - also für reale Adressbereiche ab 1 GiB Speicherumfang
- typische Aufteilung von $A = [0, 2^{32} - 1] = 4$ GiB:
  - gleich
    - 2 GiB jeweils für Benutzer- und Betriebssystemadressraum
    - NT
  - ungleich
    - 3 GiB Benutzer- und 1 GiB Betriebssystemadressraum
    - Linux, NT (Enterprise Edition)
  - steht und fällt mit der Größe von Benutzerprogrammen/-prozessen

*Inklusion bedeutet aber eben auch, dass die Benutzerprozesse dem Betriebssystem ein **stärkeres Vertrauen** schenken müssen*

- Schreibschutz auf A_p legen und nur bei Bedarf zurücknehmen/lockern
  - sonst sind Zeigerfehler in $A_t - A_p$ verheerend für Programme in A_p
  - aber auch implizit erlaubte Lesezugriffe verletzten die **Privatsphäre**...

Gliederung

Resümee

. . . ohne TLB geht nichts

- **Prozessadressräume** sind (a) real, (b) logisch oder (c) virtuell
  - (a) lückenhafter, wirklicher Hauptspeicher
  - (b) lückenloser, wirklicher Hauptspeicher
  - (c) lückenloser, scheinbarer Hauptspeicher
  - Arbeitsspeicher liegt im Vordergrund (a, b) bzw. Hintergrund (c)
- logische/virtuelle Adressräume sind **seiten- oder segmentorientiert**
  - d.h., sie sind eine Aufzählung ein- oder zweidimensionaler Adressen

    eindimensional – Tupel (Seitennummer, Versatz)

    zweidimensional – Paar (Segmentnummer, Adresse) bzw.

    – Paar (Segmentnummer, (Seitennummer, Versatz))
  - letztere Paarung gilt für die **seitennummerierte Segmentierung**
- **Mehradressraumsysteme** vervielfachen den realen Adressbereich
  - implementieren (total/partiell) private Adressräume
  - Informationsaustausch zwischen Betriebssystem- & Benutzeradressraum:
    - fensterbasiert, bedarfsorientierte Einblendung von Adressraumabschnitten
    - spezialbefehlbasiert, selektives Kopieren von Maschinenwörtern
    - adressraumgeteilt, direkter Zugriff auf kompletten Benutzeradressraum

Literaturverzeichnis I

[1] Case, R. P. ; Padegs, A. :
Architecture of the IBM System/370.
In: *Communications of the ACM* 21 (1978), Jan., Nr. 1, S. 73–96

[2] Hildebrand, D. :
An Architectural Overview of QNX.
In: *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures (USENIX Microkernels)* USENIX Association, 1992. – ISBN 1–880446–42–1, S. 113–126

[3] IBM Corporation (Hrsg.):
IBM System/370 Principles of Operation.
Fourth.
Poughkeepsie, New York, USA: IBM Corporation, Sept. 1 1974.
(GA22-7000-4, File No. S/370-01)

[4] Kleinöder, J. ; Schröder-Preikschat, W. :
Adressbindung.
In: [7], Kapitel 6.3

[5] Kleinöder, J. ; Schröder-Preikschat, W. :
Prozesse.
In: [7], Kapitel 6.1

Literaturverzeichnis II

[6] Kleinöder, J. ; Schröder-Preikschat, W. :
Speicher.
In: [7], Kapitel 6.2

[7] Kleinöder, J. ; Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)

[8] Lions, J. :
A Commentary on the Sixth Edition UNIX Operating System.
The University of New South Wales, Department of Computer Science, Australia :
`http://www.lemis.com/grog/Documentation/Lions`, 1977

[9] Lions, J. :
UNIX Operating System Source Code, Level Six.
The University of New South Wales, Department of Computer Science, Australia :
`http://v6.cuzuco.com`, Jun. 1977

[10] Quantum Software Systems Ltd. (Hrsg.):
QNX Operating System User's Manual.
Version 2.0.
Toronto, Canada: Quantum Software Systems Ltd., 1984

Literaturverzeichnis III

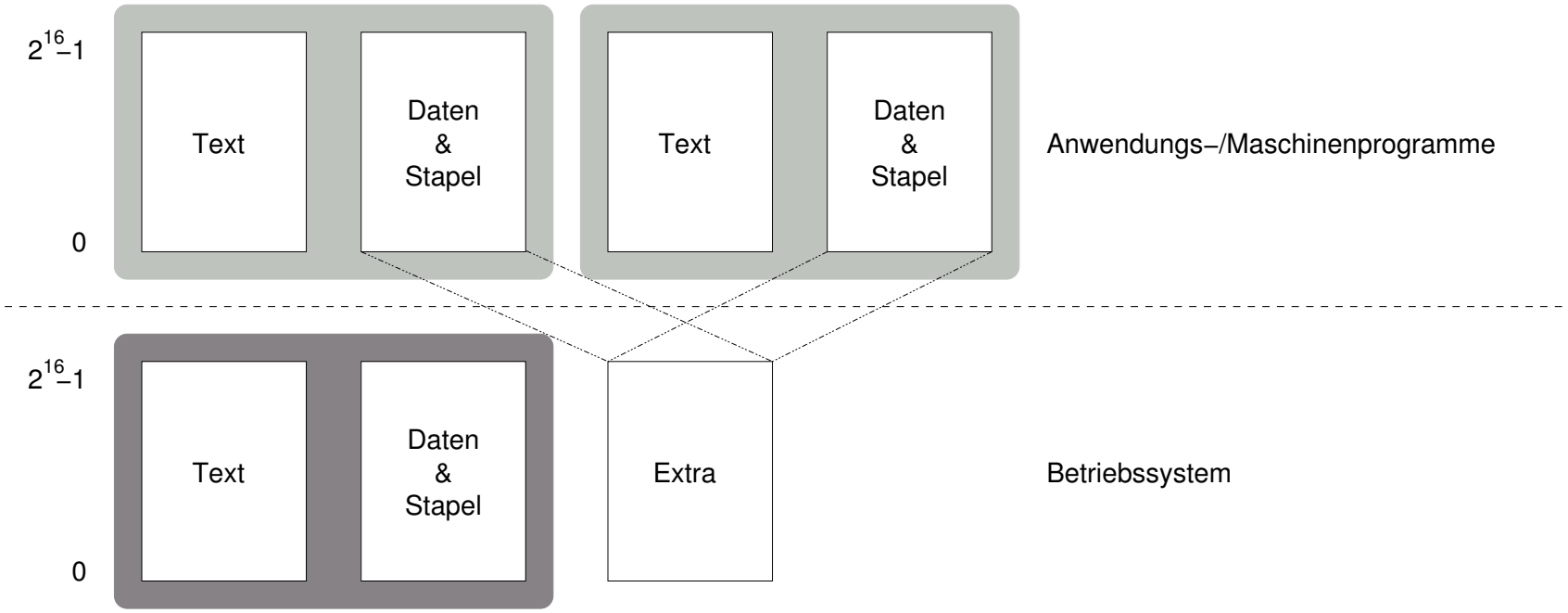
[11] Schröder, W. :
Eine Familie von UNIX-ähnlichen Betriebssystemen – Anwendung von Prozessen und des Nachrichtenübermittlungskonzeptes bem strukturierten Betriebssystementwurf, Technische Universität Berlin, Diss., Dez. 1986

[12] Uhlig, R. ; Nagle, D. ; Stanley, T. ; Mudge, T. ; Sechrest, S. ; Brown, R. :
Design Tradeoffs for Software-Managed TLBs.
In: *ACM Transactions on Computer Systems* 12 (1994), Aug., Nr. 3, S. 175–205

[13] Wikipedia:
Virtualität.
`https://de.wikipedia.org/wiki/Virtualit{ä}t`, Aug. 2015

Programmierte Mitbenutzung

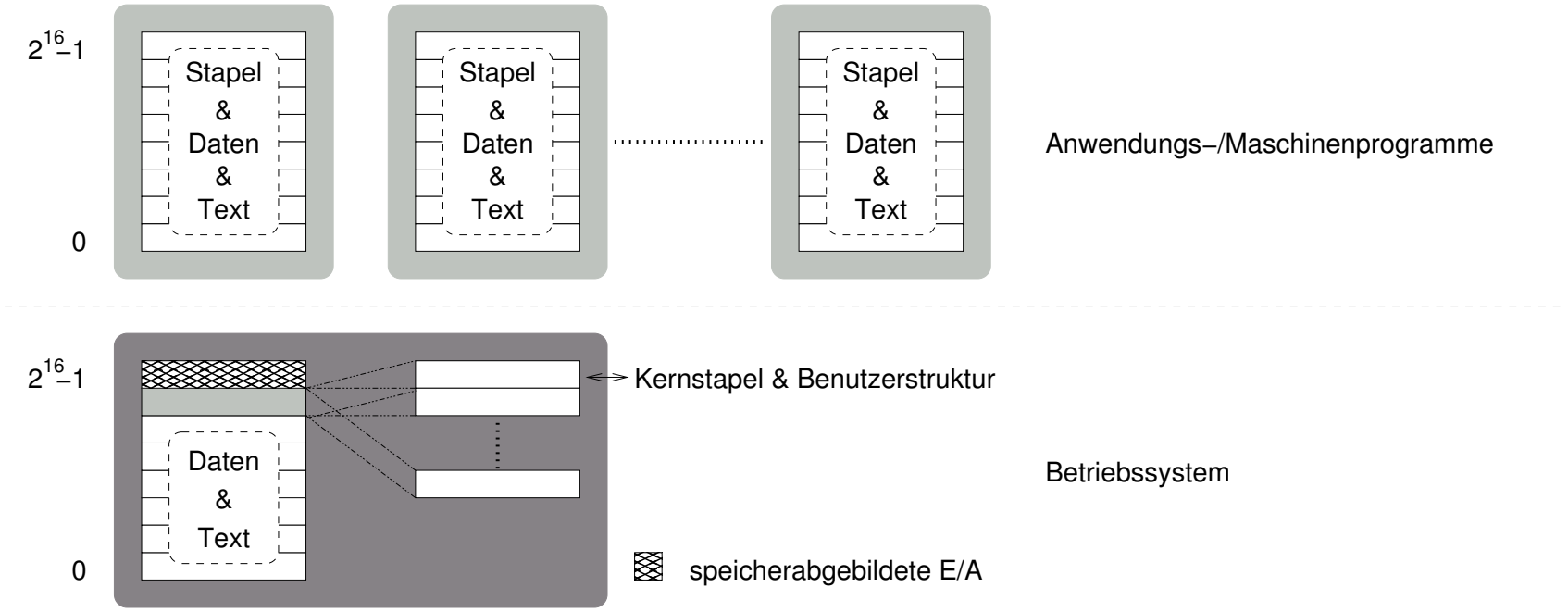
total



- antiquiert (i8086) **fensterbasierter Ansatz** zum Datenaustausch
  - horizontal: Interprozesskommunikation (Nachrichtenversenden)
  - vertikal: Zugriffe auf den Benutzeradressraum mittels **Extrasegment**
- Beispiele von (mikrokernbasierten) Betriebssystemen der Art:
  - QNX [10]: ereignisbasiert, vgl. auch [2]
  - AX [11]: ereignis-/prozedurbasiert, QNX-kompatibel

Programmierte Fernzugriffe

selektiv



- antiquiert (PDP 11/40) **kopiebasierter Ansatz** zum Datenaustausch
  - horizontal
    - Interprozesskommunikation (Nachrichtenversenden, *pipe*)
    - seitenbasierte Mitbenutzung in Inkrementen von 64 Bytes
  - vertikal
    - Zugriffe auf den Benutzeradressraum mittels **Spezialbefehle**
    - *move from/to previous instruction space* (`mfpi`/`mtpi`)
- prominentes Beispiel eines (monolithischen) Betriebssystems der Art:
  - UNIX
    - Version 6 [9, 8], prozedurbasiert

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – XII.2 Speicherverwaltung: Zuteilungsverfahren

Wolfgang Schröder-Preikschat

21. Dezember 2021

Agenda

Einführung
- Rekapitulation

Platzierungsstrategie
- Freispeicherorganisation
- Verfahrensweisen

Speicherverschnitt
- Fragmentierung
- Verschmelzung
- Kompaktifizierung

Zusammenfassung

Gliederung

Lehrstoff

- Grundlagen der **Speicherzuteilungsstrategie** eines Betriebssystems für Mehrprogrammbetrieb thematisieren und punktuell vertiefen
  - verschiedene Formen der Organisation freien Speichers darstellen
  - Abspeicherung von Verwaltungsstrukturen beleuchten
  - Freispeicher, sog. **Löcher**, als speziell gefüllte **Hohlräume** auffassen
- klassische **Verfahrensweisen** besprechen und dadurch verschiedene Aspekte einer Zuteilungsstrategie herausarbeiten
  - Löcher nach Größe verwalten: *best-fit*, *worst-fit*, *buddy*
  - Löcher nach Adresse verwalten: *first-fit*, *next-fit*
- auf **Speicherverschnitt** eingehen, ein grundsätzliches Problem jeder Zuteilungsvariante, das ihre Effizienz bestimmt

  intern
  - der, wenn er auftritt, unvermeidbar ist

  extern
  - der aufwendig auflösbar ist
- **Verschmelzung** und **Kompaktifizierung** erklären, zwei Maßnahmen, um Speicherverschnitt zu minimieren oder aufzulösen

Aufgaben der Speicherverwaltung

- zentrale Aufgabe ist es, über die **Speicherzuteilung** an einen Prozess Buch zu führen und seine Adressraumgröße dazu passend auszulegen

  Platzierungsstrategie (*placement policy*)
  - wo im Hauptspeicher ist noch Platz?
- zusätzliche Aufgabe kann die **Speichervirtualisierung** sein, um trotz knappem Hauptspeicher Mehrprogrammbetrieb zu maximieren

  Ladestrategie (*fetch policy*)
  - wann muss ein Datum im Hauptspeicher liegen?

  Ersetzungsstrategie (*replacement policy*)
  - welches Datum im Hauptspeicher ist ersetzbar?
- die zur Durchführung dieser Aufgaben typischerweise zu verfolgenden Strategien profitieren voneinander — oder bedingen einander
  - ein Datum kann ggf. erst platziert werden, wenn Platz freigemacht wurde
  - etwa indem das Datum den Inhalt eines belegten Speicherplatzes ersetzt
  - ggf. aber ist das so ersetzte Datum später erneut zu laden
  - bevor ein Datum geladen werden kann, ist Platz dafür bereitzustellen

Gliederung

Verwaltung der freien Speicherbereiche

*Ein freier Bereich erscheint als **Hohlraum** im Innern des Haupt- oder Arbeitsspeichers eines Rechensystems.*

- ein solcher Hohlraum wird als **Loch** (*hole*) bezeichnet, wobei mehrere davon und getrennt voneinander im realen Adressraum liegen
  - die Struktur dieser Hohlräume ist von fester oder variabler Größe
  - entsprechend motiviert sie verschiedene Darstellungen des Freispeichers

  Bitkarte
  - für Hohlräume fester Größe $\rightsquigarrow$ *bit map*
    - eignet sich für **seitennummerierte Adressräume**
    - grobkörnige Speichervergabe auf Seitenrahmenbasis

    $\hookrightarrow$ alle Hohlräume sind gleich gut bei der Löchersuche ☺

  Lochliste
  - für Hohlräume variabler Größe $\rightsquigarrow$ *hole list*
    - ist typisch für **segmentierte Adressräume**
    - feinkörnige Speichervergabe auf Segmentbasis

    $\hookrightarrow$ nicht alle Hohlräume sind gleich gut bei der Löchersuche ☹

- Anforderung an Verfahren zur Hohlraumzuteilung ist **Effizienz**, d.h., Sparsamkeit bezüglich Rechenzeit und Speicherplatz
  - in Hinsicht auf **Vergeudung** und **Zerstückelung** freien Speichers

Bitkarte

- der Speicher ist aufgeteilt in **gleichgroße Stücken**, die jeweils Platz für n Bytes bieten, mit n typischerweise eine **Zweierpotenz**
  - d.h., n ist Vielfaches der **Seitengröße** eines logischen Adressraums
- jedes solcher Stücke hat einen zweiwertigen logischen Zustand, der eine Aussage zur freien **Verfügbarkeit** macht

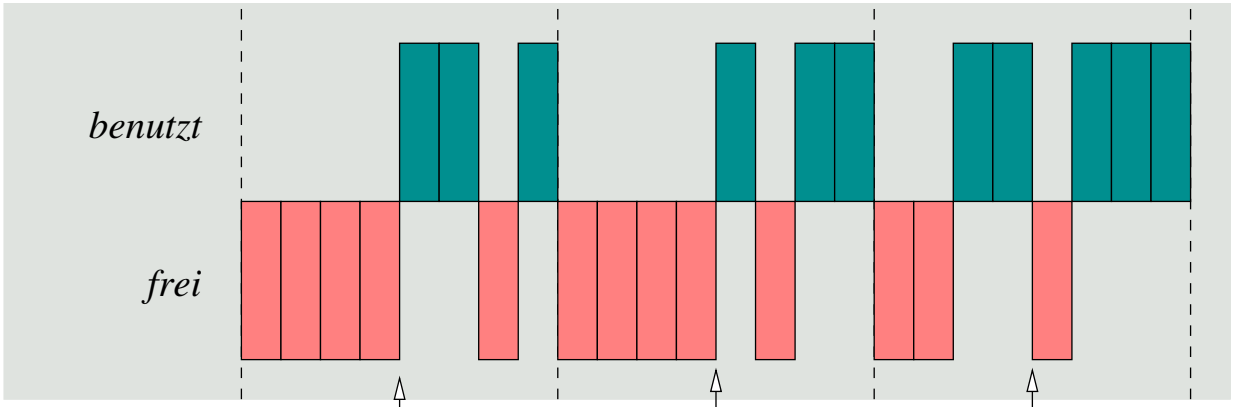
  frei
  - das Stück ist ein Hohlraum, keinem Prozess zugeordnet

  benutzt
  - das Stück ist kein Hohlraum, einem Prozess zugeordnet
  - je nach Konvention mit den Werten 1 und 0 kodiert, oder umgekehrt
- der Speicherbedarf der Karte für den gesamten Hauptspeicher eines Rechners hängt damit maßgeblich von der Stückgröße ab
  - angenommen 8 GiB Hauptspeicher und 4 KiB Stück (Seitengröße):

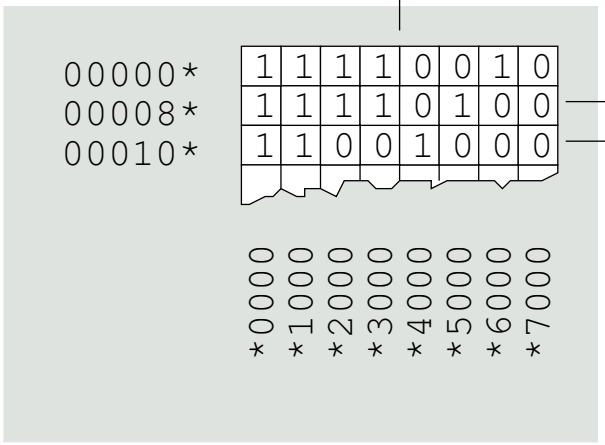
$$\begin{aligned} 8\,\text{GiB} &= 2097152 \text{ Seiten à } 4096 \text{ Bytes} = 2097152 \text{ Bits} \\ &= 262144 \text{ Bytes} = 256\,\text{KiB} \end{aligned}$$

  - d.h., die Unkosten zur Abspeicherung der Bitkarte betragen 0.003 %
- Aktionen zur Suche, zum Erwerben und zur Abgabe eines Hohlraums operieren auf ein byteweise gespeichertes zweidimensionales Bitfeld
  - manche Prozessoren (x86) bieten hierfür spezielle Maschinenbefehle

Freispeicherverwaltung mit Bitkarte



Seitenrahmen/Kacheln



Bitkarte

- Beispiel mit 4 KiB Seitengröße:
  - als Bytefeld repräsentierte Bitkarte
  - jede Zeile attributiert 8 Seiten
- Adressschritte direkt ablesbar
  - pro Zeile 0x8000 ($8 \times 4KiB$)
  - pro Spalte 0x1000 ($4KiB$)
- in Kombination die Kacheladresse

Lochliste

- der Speicher ist aufgeteilt in eventuell **verschiedengroße Stücke**, die jeweils Platz für mindestens n Bytes bieten
  - wobei n typischerweise Vielfaches der Größe von einem **Listenelement** ist
    - d.h., 4, 8 oder 16 Bytes bei einer 16-, 32- bzw. 64-Bit Maschine

```
1 typedef struct piece {
2     chain_t *next;  /* single-linked list assumed */
3     size_t size;    /* # of bytes claimed by this piece */
4 } piece_t;
```

- der Speicherbedarf einer Liste für den gesamten Hauptspeicher eines Rechners hängt damit von Anzahl und Größe der Hohlräume ab
  - gleiche Annahme wie zuvor, jedoch Seite gleich Segment und 64-Bit:

$$\begin{aligned} 8\,\text{GiB} &< \text{2097152 Listenelemente (}\texttt{piece\_t}\text{) à 16 Bytes} \\ &< 33554432\ \text{Bytes} \equiv 32\,\text{MiB} \end{aligned}$$

  - d.h., die Unkosten zur Abspeicherung der Lochliste liegen unter 0.390 %

↪ sie fallen an, falls Hohlräume selbst unbrauchbar zur Abspeicherung sind

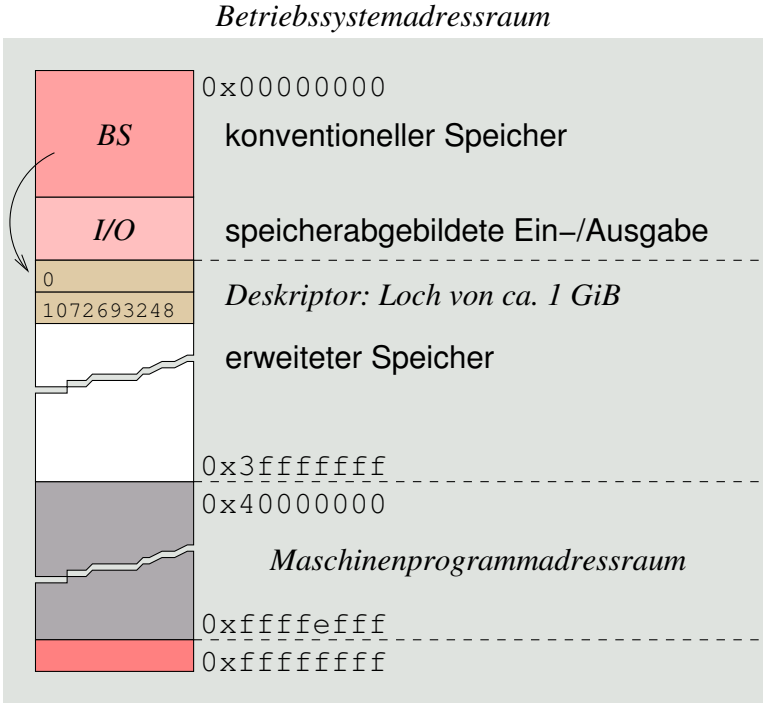
- Aktionen zur Suche, zum Erwerben und zur Abgabe eines Hohlraums beziehen sich auf eine **dynamische Datenstruktur**

Abspeicherung der Lochliste

- jedes Listenelement beschreibt ein Stück freien Speicher, d.h., einen leeren oder mit etwas angefüllten Hohlraum im Speicherinnern
  - z.B. angefüllt mit eben dem Listenelement, das den Hohlraum beschreibt
- somit ergeben sich zwei grundlegende **Speicherausprägungen** für die Lochliste, mit Konsequenzen in verschiedener Hinsicht
  - i die Hohlräume sind wirklich leer, adressräumlich von der Liste getrennt
    - jeder Hohlraum ist freier Speicherplatz im realen Adressraum, das durch ihn repräsentierte Loch kann beliebig klein sein: $sizeof(hole) > 0$
    - jedes Listenelement belegt Speicher im Adressraum des Betriebssystems und die Listenoperationen wirken in derselben Schutzdomäne
  - ii die Hohlräume sind scheinbar leer, adressräumlich mit der Liste vereint
    - jeder Hohlraum ist freier Speicher und zugleich ein Listenelement im realen Adressraum, er hat eine Mindestgröße: $sizeof(hole) \geq sizeof(piece\_t)$
    - kein Listenelement belegt Speicher im Adressraum des Betriebssystems, aber die Listenoperationen wirken in einer anderen Schutzdomäne
- bei spezieller Auslegung des Betriebssystemadressraums kann von den positiven Eigenschaften beider Ausprägungen profitiert werden

Adressraumbelegungsplan Betriebssystem

- angenommen, der Hauptspeicher von $\approx 1\,\mathrm{GiB}$ liegt partitioniert im realen Adressraum wie folgt:
  - 640 KiB **konventioneller Speicher** ab Adresse 0x00000000
  - 1 GiB − 640 KiB **erweiterter Speicher** ab Adresse 0x00100000
- weiter sei angenommen, dass für das Betriebssystem eine **identische Abbildung** (*identity mapping*) von logischen zu realen Adressen gilt
  - die **Adressraumpartition** für das Betriebssystem macht die unteren 1 GiB aus (vgl. [2, S. 29–31]):
    - der konventionelle Speicher ist für das Betriebssystem bestimmt
    - der erweiterte Speicher ist für die Maschinenprogramme bestimmt
    - die **Lochliste** liegt dann ebenfalls im erweiterten Speicher
  - initial besteht die Lochliste aus nur einem Listenelement



Identische Abbildung

identity mapping

*Die **Identität** von realem und logischem/virtuellem Adressraum des Betriebssystems, in dem eine logische/virtuelle Adresse identisch zu einer realen Adresse ist.*

- im gegebenen Beispiel bedeutet dies, dass die logische Adresse eines Elements der Lochliste der realen Adresse des Lochs gleicht
  - d.h., die Elemente der Lochliste liegen im Betriebssystemadressraum und
  - jedes Element füllt dabei jeweils auch einen Hohlraum im Hauptspeicher

  ↪ die Listenoperationen wirken in der Domäne des Betriebssystems

  ↪ zur Verwaltung freien Speichers fällt kein zusätzlicher Speicherbedarf an
- bei dieser **Hilfskonstruktion** (*workaround*) sind nicht nur Hohlräume, sondern alle Stücke dem Betriebssystem direkt zugänglich
  - Adressierungsfehler im Betriebssystem können daher leicht Stücke treffen, die Maschinenprogramme oder einige ihrer Bestandteile speichern
  - diese Stücke sind daher im Betriebssystemadressraum auszublenden
- sowohl segmentierter als auch seitennummerierter Adressraum helfen, die **Gebrauchsstücke** vor direkten Zugriffen zu schützen

Gebrauchsstück

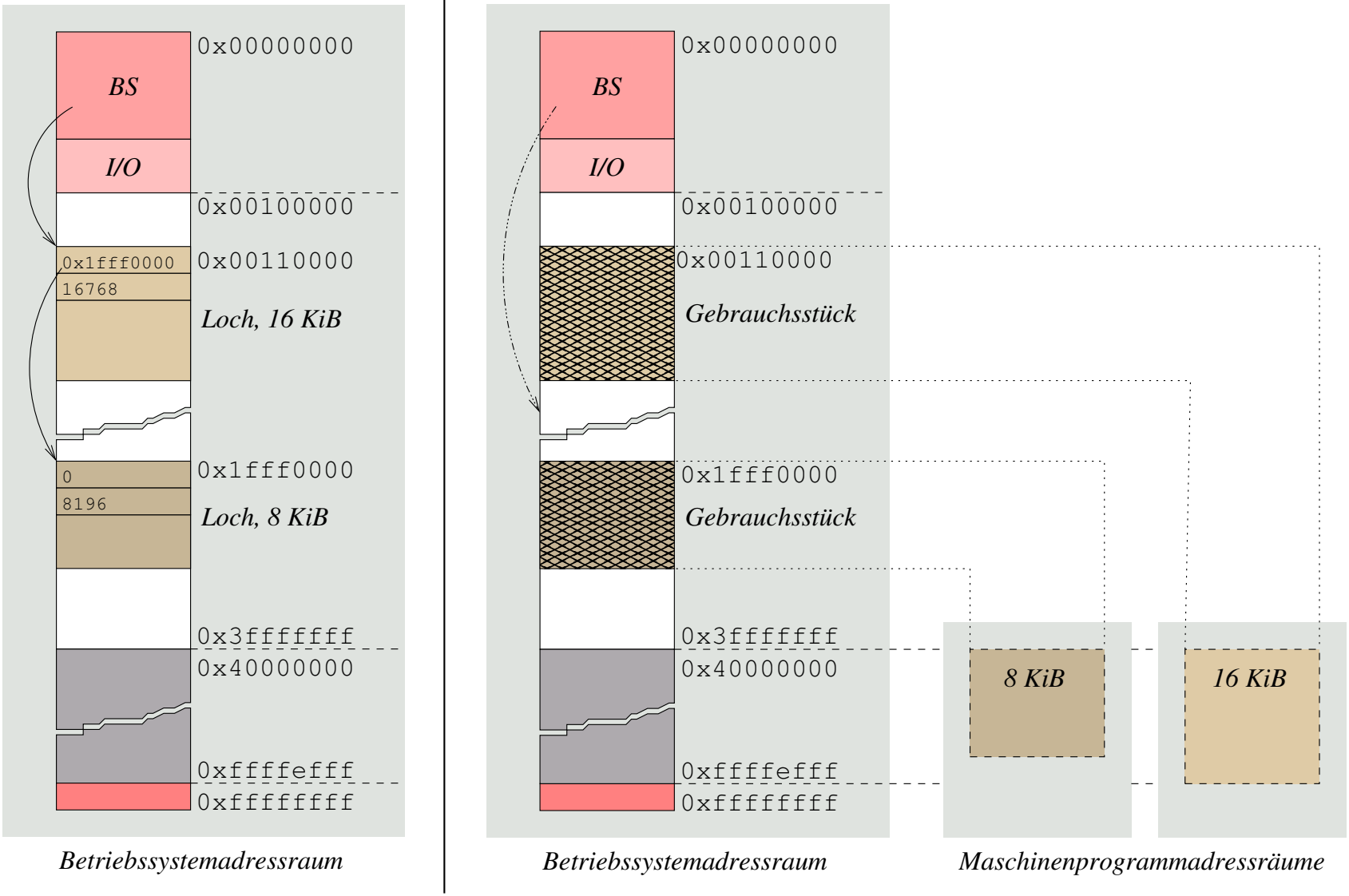
functional piece

Neben den Stücken, die Hohlräume darstellen, solche, die allgemein zur Ablage von Programmtext, -daten und Stapeln im Hauptspeicher von Prozessexemplaren in Gebrauch sind.

- ein solches Stück bildet entweder ein **Segment** oder ein **Vielfaches von Seiten**, je nach Adressraumkonzept (vgl. [2])
  - geschützt durch einen Segmentdeskriptor bzw. $n \geq 1$ Seitendeskriptoren
  - zugeteilt dem Adressraum des Prozesses, der das Stück gebraucht
- im Moment der **Zuteilung** zum Prozessadressraum, wird es aus dem Betriebssystemadressraum ausgeblendet
  - der mit dem Stück darin abgedeckte Adressbereich bleibt jedoch gültig
  - allerdings ist dieser Bereich nicht mehr durch **Adresszugriffe** zugänglich
- bei **Zurücknahme** der Stücke bzw. **Zerstörung** des Prozessexemplars werden sie wieder in den Betriebssystemadressraum eingeblendet
  - die Stücke werden wieder zu Hohlräumen, kommen auf die Lochliste
  - sie erscheinen wieder an ihren alten Stellen im Betriebssystemadressraum

Die Lösung ist immer einfach, man muss sie nur finden. (Alexander Solschenizyn)

Ein- und Ausblendung von Speicherstücken



Lineare Lochliste I

Hinweis (Verschnitt vs. Suchaufwand)

Ist die angeforderte Größe kleiner als die Größe des gefundenen Loch, die Differenz jedoch größer als ein Listenelement ist, fällt Verschnitt an, der als verbleibendes Loch in die Liste neu einsortiert werden muss.

- die Lochliste ist der Größe nach auf- oder absteigend sortiert:
  - *best-fit*
    - aufsteigende Lochgrößen, das kleinste passende Loch suchen
    - beste Zuteilung, minimaler Verschnitt, aber eher langsam
    - erzeugt kleine Löcher von vorn, erhält große Löcher hinten
    - ↪ hinterlässt eher kleine Löcher, bei steigendem Suchaufwand
  - *worst-fit*
    - absteigende Lochgrößen, das größte passende Loch suchen
    - sehr schnelle Zuteilungsentscheidung, begünstigt Zerstückelung
    - zerstört große Löcher von vorn, macht kleine Löcher hinten
    - ↪ hinterlässt eher große Löcher, bei konstantem Suchaufwand
- fällt ein Restloch an, muss dieses in die Liste einsortiert werden, aber nur, wenn es eine bestimmte **Mindestgröße** nicht unterschreitet
  - typischerweise die Größe (in Bytes) eines Listenelements

Lineare Lochliste II

Hinweis (Suchaufwand vs. Zuteilung)

Ist die angeforderte Größe kleiner als die Größe des gefundenen Loch, die Differenz jedoch größer als ein Listenelement ist, fällt Verschnitt an, der jedoch nicht in die Liste einsortiert werden muss.

- die Lochliste ist der Größe des Adresswerts nach aufsteigend sortiert:
  - *first-fit*
    - schnelle Zuteilung, begünstigt aber Verschwendung
    - erzeugt kleine Löcher von vorn, erhält große Löcher hinten
    - ↪ hinterlässt eher kleine Löcher, bei steigendem Suchaufwand
  - *next-fit*
    - reihum (*round-robin*) Variante von *first-fit*
      - die Suche beginnt immer beim zuletzt zugeteiltem Loch
    - ↪ hinterlässt eher gleichgroße Löcher (Gleichverteilung)
    - ↪ Konsequenz ist ein im Mittel eher abnehmender Suchaufwand
- keine dieser Verfahren erzeugt ein Restloch, das im nachgeschalteten zweiten Listendurchlauf einsortiert werden müsste
  - sie machen eine effiziente Hohlraumverwaltung (vgl. S. 14) möglich

Halbierungsverfahren

binary buddy

Hinweis (Verschnitt vs. Suchaufwand)

Das zur Speicheranfrage gegebener Größe am besten passende Stück durch fortgesetzte Halbierung eines großen Stücks gewinnen.

- die Lochliste ist der **Zweierpotenzgröße** nach aufsteigend sortiert:

  buddy
  - sucht das kleinste passende Loch $buddy_i$ der Größe 2^i
    - i ist Index in eine Tabelle von Adressen auf Löcher der Größe 2^i
    - wobei i so zu bestimmen ist, dass gilt $2^{i-1} < size \leq 2^i$, $i > 1$
    - mit *size* als Größe (in Bytes) des angeforderten Speicherstücks
  - $buddy_i$ entsteht durch sukzessive Splittung von $buddy_j$, $j > i$:
    - $2^n = 2 \times 2^{n-1}$
    - zwei gleichgroße Stücke, die „Kumpel“ des jeweils anderen sind
  - i wird fortgesetzt dekrementiert, solange $2^{i-1} > size$, $i > 1$

- mögl. Verschnitt durch eine **Auswahl von Stückgrößen** begegnen
  - vergleichsweise geringer Such- und Aufsplittungsaufwand, jedoch kann der anfallende Verschnitt dennoch beträchtlich sein
    - im Mittel sind die zugeteilten Stücke um 1/3 größer als angefordert und die belegten Stücke nur zu 3/4 genutzt [1, S. 32]

Gliederung

Bruchstückbildung

Verschnitt durch zuviel zugeteilte oder nicht nutzbare Bereiche, der als Abfall in Erscheinung tritt und Verschwendung bedeutet.

- je nach **Adressraumkonzept** und **Zuteilungsverfahren** zeigen sich verschiedene Ausprägungen der **Fragmentierung**

  intern
  - seitennummerierte Adressräume, Halbierungsverfahren (*buddy*)
  - die angeforderte Größe ist kleiner als das zugeteilte Stück
    - falls Seitennummerierung, ist die Größe auch kein Seitenvielfaches
  - der „lokale Verschnitt“ ist nutzbar, dürfte es aber nicht sein

  ↪ **Verschwendung**, ist (durch das Betriebssystem) unvermeidbar

  extern
  - segmentierte Adressräume, Halbierungsverfahren (*buddy*)
  - die angeforderte Größe ist zu groß für jedes einzelne Loch
    - in Summe ihrer Größen genügen die Löcher der angeforderten Größe
    - allerdings sind sie im Hauptspeicher nicht linear angeordnet
  - der „globale Verschnitt“ ist ggf. nicht mehr zuteilbar

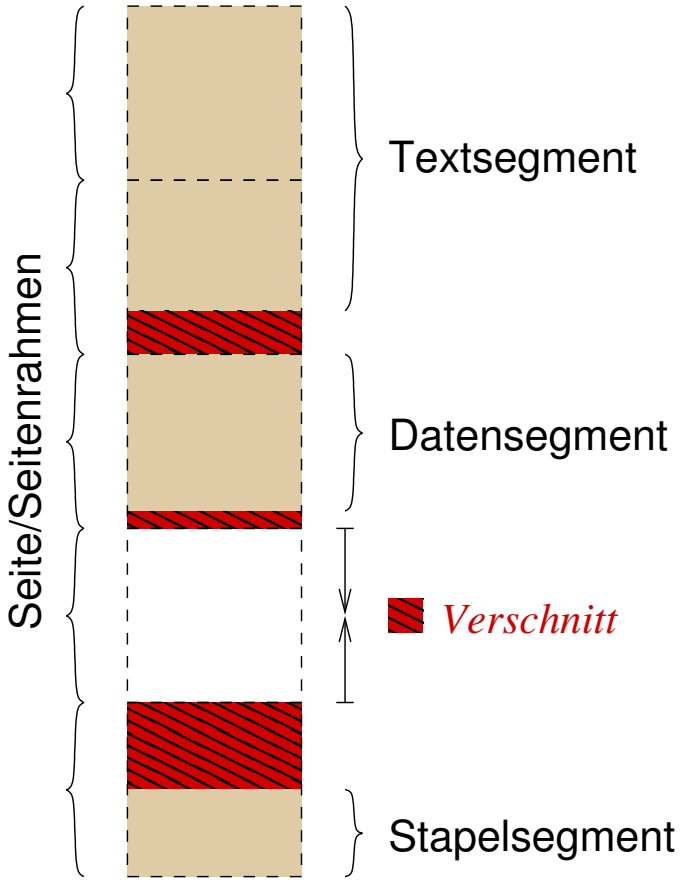
  ↪ **Verlust**, ist (durch das Betriebssystem) aufwendig vermeidbar

- externe Fragmentierung kann durch **Verschmelzung** verringert und **Kompaktifizierung** aufgelöst werden

Interne Fragmentierung

internal fragmentation

- **seitennummerierter Adressraum**
  - abzubildende Programmsegmente sind Vielfaches von Bytes
  - der (log./virt.) Prozessadressraum ist aber ein Vielfaches von Seiten
  - die jew. letzte Seite der Segmente ist ggf. nicht komplett belegt
- **seitenlokaler Verschnitt**
  - wird vom Programm *logisch* nicht beansprucht
  - ist vom Prozess *physisch* jedoch adressierbar
  - da eine seitennummerierte MMU Seiten schützt, keine Segmente

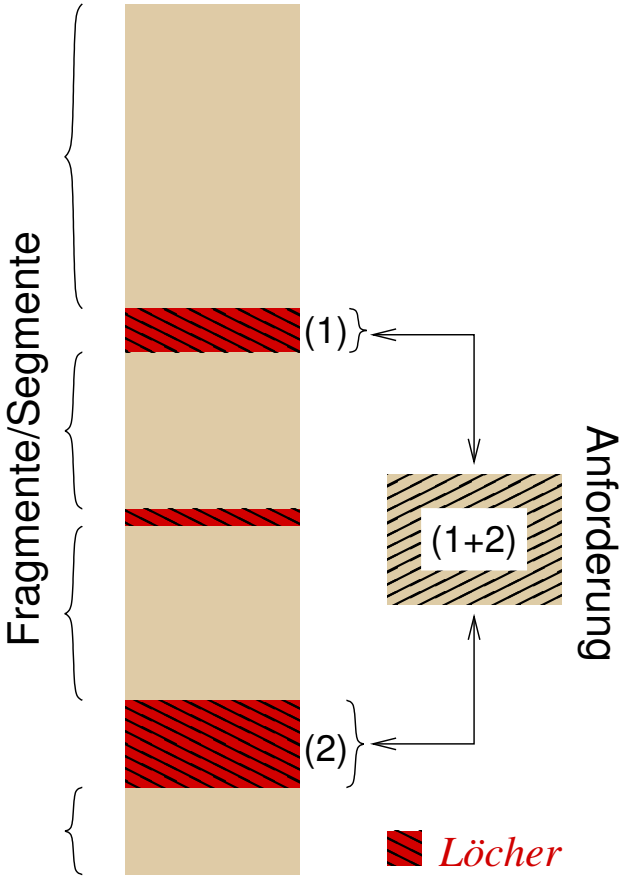


- das **Halbierungsverfahren** (*buddy*) liefert ein ähnliches Bild
  - immer dann, wenn die Anforderungsgröße keine Zweierpotenz ist
  - ein Verschnitt von $2^i - size$ (in Bytes) ergibt sich zum Stückende hin

Externe Fragmentierung

external fragmentation

- **segmentierter Adressraum**
  - die zu platzierenden Fragmente sind Vielfaches von Bytes
  - sie werden 1:1 auf Segmente einer MMU abgebildet
  - die jew. eine lineare Bytefolge im realen Adressraum bedingen
- **globaler Verschnitt**
  - die Summe von Löchern ist groß genug für die Speicheranforderung
  - die Löcher liegen aber verstreut im realen Adressraum vor und
  - jedes einzelne Loch ist zu klein für die Speicheranforderung



- das **Halbierungsverfahren** (*buddy*) liefert ein ähnliches Bild
  - immer dann, wenn zwischen zu kleinen Löchern ein Gebrauchsstück liegt
  - jede Stückgröße ist eine Zweierpotenz, das größte Loch ist aber zu klein

Vereinigung eines Lochs mit angrenzenden Löchern

*Eine wichtige Maßnahme, die bei der **Zurücknahme** eines Gebrauchsstücks oder **Zerstörung** eines Prozessexemplars greift.*

- **Verschmelzung** von Löchern erzeugt größere Hohlräume und bringt damit folgende positive (nichtfunktionale) Eigenschaften
  - weniger Löcher, dadurch geringere externe Fragmentierung
  - weniger Lochdeskriptoren, dadurch kürzere Listen und Suchzeiten
  - beides beschleunigt die Speicherzuteilung, gibt kürzere Antwortzeiten

- **Löchervereinigung** sieht sich mit vier Situationen konfrontiert, je nach dem, welche relative Lage ein Loch im Adressraum hat:

1. zw. zwei Gebrauchsstücken	▪ 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. und 3.

Bezug zum Zuteilungsverfahren

- die **Verschmelzungsaufwände** variieren teils sehr stark mit der Art und Weise, wie die Lochliste vom Betriebssystem geführt ist:

buddy

- das Stück wird mit seinem *Buddy*-Stück verschmolzen
- Adressen zweier *Buddies* gleichen sich bis auf einem Bit
- ein Stück ist *Buddy* eines anderen Stücks, wenn gilt:

```
1  bool buddy(void *this, unsigned size, void *that) {
2      return (size && !(size & (size - 1))) /* power of two!? */
3          && (((unsigned)this ^ (unsigned)that) == size);
4  }
```

- ggf. mit jeweils nächst größerem *Buddy* verschmelzen

first/next-fit

- beim Einsortieren in die Lochliste Nachbarschaft prüfen
- ein Stück ist Nachbar eines anderen Stücks, wenn gilt:

```
5  bool neighbor(void *this, unsigned size, void *that) {
6      return ((unsigned)this + size) == (unsigned)that;
7  }
```

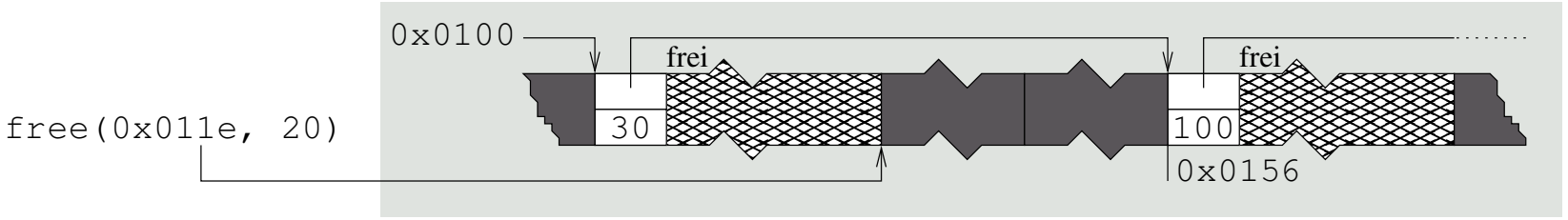
- mit jeweils aktuellem und nächsten Listenelement prüfen

best/worst-fit

- wie *first/next-fit*, beim Einsortieren prüfen
- aber Listennachfolger müssen keine Nachbarn sein ☹
- die ganze Lochliste durchlaufen: zwei Nachbarn finden ☹
- erst dann ggf. verschmelzen und neu einsortieren ☹

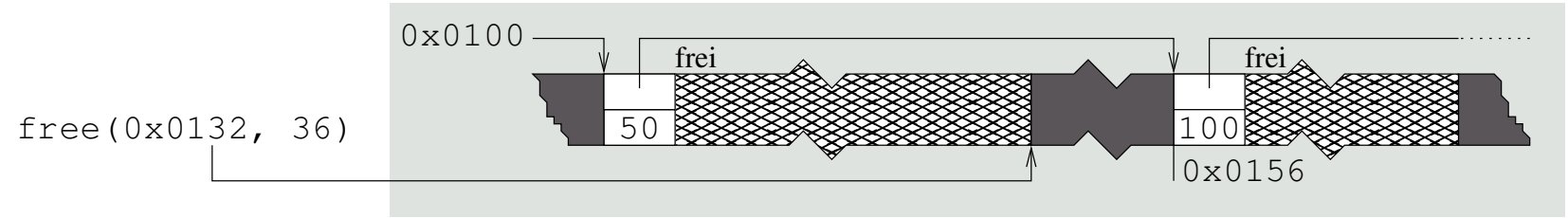
Vereinigung beliebiger Löcher

- das Stück ist „rechter Nachbar“ (Nachfolger) des Lochs:



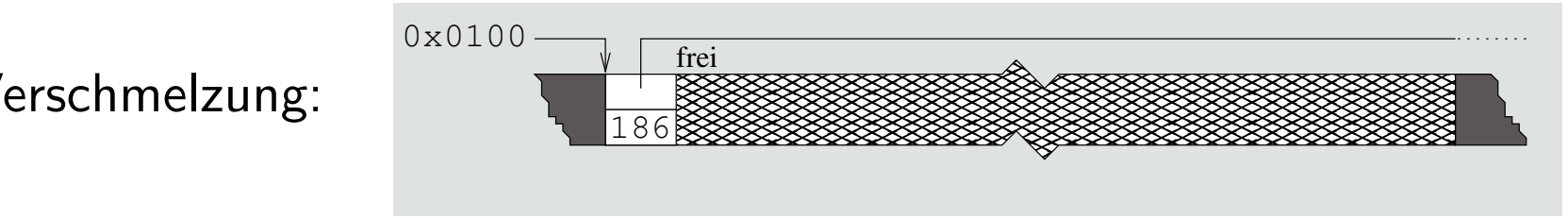
1. das alte 30 Bytes große Loch kann um 20 Bytes vergrößert werden

- Verschmelzung; das nächste Stück ist „rechter Nachbar“ (Nachfolger) des Lochs und „linker Nachbar“ (Vorgänger) des Lochnachfolgers



2. das alte 50 Bytes große Loch kann um 36 Bytes vergrößert werden
3. das neue 86 Bytes große Loch kann um 100 Bytes vergrößert werden

- Verschmelzung:

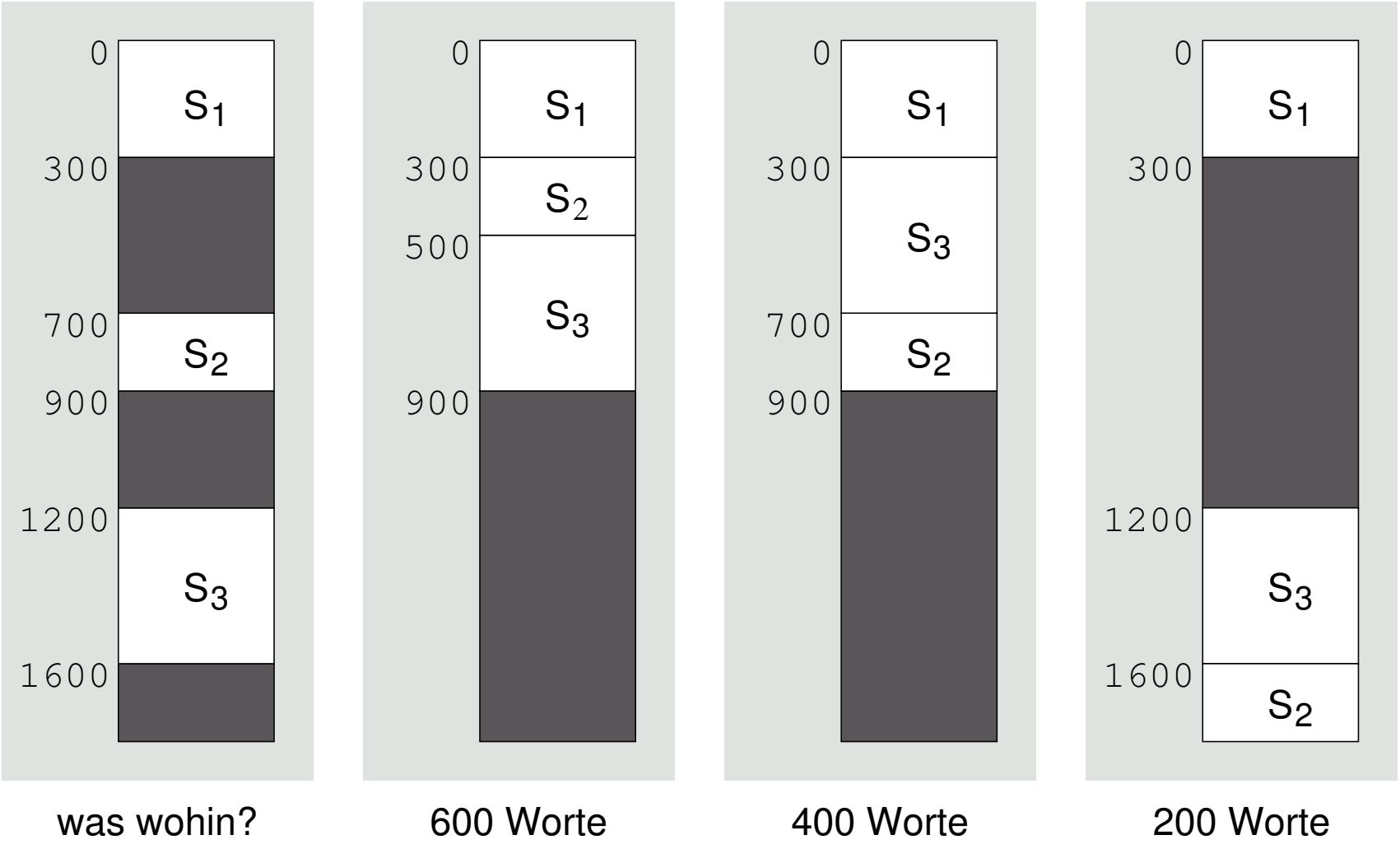


Vereinigung des globalen Verschnitts

Die Gebrauchsstücke im Hauptspeicher werden so verschoben, dass am Ende ein einziges großes Loch vorhanden ist.

- um **externe Fragmentierung** aufzulösen, sind Gebrauchsstücke im Hauptspeicher durch **Kopiervorgänge** umzulagern
  - i direkt im Hauptspeicher oder
  - ii indirekt über den Ablagespeicher $\rightsquigarrow$ *swapping*
  - so wird zunächst ein weiteres Loch geschaffen, das dann aber gleich wieder mit Nachbarlöchern verschmilzt
    - schrittweise wird die Lochliste verkürzt, bis nur noch ein Loch übrigbleibt
- Umlagerung zieht **Verlagerung** der betroffenen Segmente oder Seiten nach sich, wenn sie ihre neue Position im realen Adressraum haben
  - deren Lage ändert sich nur im realen Adressraum, nicht im logischen
    - nur die Basisadresse im Segment-/Seitendeskriptor ist zu aktualisieren
  - im logischen Adressraum behält jedes Segment/jede Seite seine Adresse
- zentraler Aspekt dabei ist, die Anzahl der Umlagerungsvorgänge zu minimieren, was ein komplexes **Optimierungsproblem** darstellt

Auflösung externer Fragmentierung: Optionen



Gliederung

Resümee

- Zuteilung von Arbeitsspeicher ist Aufgabe der **Platzierungsstrategie**
  - die Erfassung freier Speicherstücke hängt u.a. ab vom Adressraummodell
    - i Seiten bzw. Seitenrahmen $\rightsquigarrow$ Bitkarte oder Lochliste
    - ii Segmente $\rightsquigarrow$ Lochliste
  - weitere Folge davon ist **interne** (i) oder **externe** (ii) **Fragmentierung**
    - Speicherverschnitt durch zuviel zugeteilte bzw. nicht nutzbare Bereiche
- die **Zuteilungsverfahren** verwalten Löcher nach Größe oder Adresse
  - nach abnehmender Größe *worst-fit*
  - nach ansteigender Größe *best-fit*, *buddy*
  - nach ansteigender Adresse *first-fit*, *next-fit*
- angefallener **Speicherverschnitt** ist zu reduzieren oder aufzulösen
  - i Verschmelzung von Löchern verringert externe Fragmentierung
    - beschleunigt die Speicherzuteilungsverfahren und
    - lässt die Speicherzuteilung im Mittel häufiger gelingen
  - ii Kompaktifizierung der Löcher löst externe Fragmentierung auf
    - hinterlässt (im Idealfall) ein großes Loch
    - erfordert aber positionsunabhängige Programme d.h. logische Adressräume

Literaturverzeichnis I

[1] Heiss, H.-U. :
Speicherverwaltung.
In: AG Betriebssysteme und Verteilte Systeme (Hrsg.): *Konzepte und Methoden der Systemsoftware*.
Universität-GH Paderborn, 2000 (Vorlesungsfolien), Kapitel 5

[2] Kleinöder, J. ; Schröder-Preikschat, W. :
Adressräume.
In: [4], Kapitel 12.1

[3] Kleinöder, J. ; Schröder-Preikschat, W. :
Speicher.
In: [4], Kapitel 6.2

[4] Kleinöder, J. ; Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)

Halbierungsverfahren: Beispiel

binary buddy

- angenommen sei die Anforderung eines Speicherstücks von 42 Bytes:
  - $2^6 = 64$ ist kleinste Zweierpotenz ≥ 42, d.h., zuzuteilen sind 64 Bytes
  - ein entsprechend großes Loch fehlt, es ist durch Splittung zu erzeugen
  - nächstes Loch in der Liste ist ein Stück von 1024 KiB
    1. $1024 = 2^{10} = 2 \times 2^9 = 512 + 512$, zu groß, eins davon wird halbiert
    2. $512 = 2^9 = 2 \times 2^8 = 256 + 256$, zu groß, eins davon wird halbiert
    3. $256 = 2^8 = 2 \times 2^7 = 128 + 128$, zu groß, eins davon wird halbiert
    4. $128 = 2^7 = 2 \times 2^6 = 64 + 64$, passt, eins davon wird zugeteilt

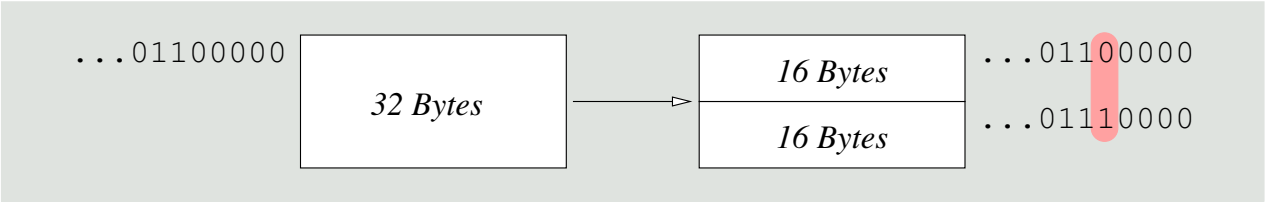
	1024 KiB				
1	512				512 KiB
2	256		256		512 KiB
3	128		128	256	512 KiB
4	64	64	128	256	512 KiB

- geeignet ist eine **zweidimensionale Repräsentation** der Lochliste:
  - i eine **Tabelle** von *Buddy*-Klassen, aufsteigend sortiert (Zweierpotenzen), lokal gespeichert im Betriebssystem
  - ii eine **lineare Liste** gleicher *Buddies*, gespeichert in den Hohlräumen

Halbierungsverfahren: Randbedingungen

binary buddy

- bei dieser Technik bilden die Größen der Adressbereiche aller Löcher und Gebrauchsstücke eine Zweierpotenz
  - wird ein beliebiges dieser Stücke in zwei gleich große Hälften gesplittet, entstehen zwei *Buddies*, deren Größe wieder einer Zweierpotenz bildet
  - umgekehrt: werden diese beiden *Buddies* wieder kombiniert, entsteht ein einzelner *Buddy* doppelter Größe
- *Buddy*-Stücke der Größe 2^n werden im Adressraum so platziert, dass ihre jeweilige Anfangsadresse ein Vielfaches von 2^n ist
  - d.h., für $n > 1$ sind die niederwertigen n Bits dieser Adressen gleich 0
- aus dieser Nebenbedingung ergibt sich folgende Konsequenz, die in zweierlei Hinsicht von Bedeutung ist:
  - i bei Splittung eines *Buddies* der Größe 2^{n+1} unterscheiden sich die beiden Adressen der *Buddy*-Hälften nur in Bit 2^n
  - ii gegeben sei ein Stück der Größe 2^n an Adresse a, dann errechnet sich die Adresse b seines *Buddies* wie folgt: $b = a + 2^n$, mit $a \bmod 2^n = 0$
- Beispiel: *Buddy* der Größe 2^5
  - gibt 2×2^4



Halbierungsverfahren: Allokation

binary buddy

- die **Lochliste**, repräsentiert als Tabelle verketteter *Buddies*:

```
1  static chain_t *holelist[NSLOT];
```

- ein Loch als *Buddy* hervorbringen (*breed*):

```
2  void *breed(unsigned slot) {
3      chain_t *hole = 0;
4      if (slot < NSLOT) {
5          if (holelist[slot] != 0) {
6              hole = holelist[slot];
7              holelist[slot] = hole->link;
8          } else {
9              if ((hole = breed(slot + 1)) != 0) {
10                 chain_t *next = (chain_t *)((unsigned)hole ^ (1 << slot));
11                 next->link = 0;
12                 holelist[slot] = next;
13             }
14         }
15     }
16     return hole;
17 }
```

- einen Bedarf (*need*) an freien Speicher geltend machen:

```
18 void *need(unsigned size) {
19     unsigned slot;
20     for (slot = 0; (1 << slot) < size; slot++);
21     return breed(slot);
22 }
```

Vereinigung hälftiger Löcher

- vgl. auch die Belegung von S. 31 (v. li.): fünf Gebrauchsstücke `A`–`E`

1. `free(D, 256)`, kein freier *Buddy*, verbleibt als Loch `D` von 256 Bytes
2. `free(B,  64)`, kein freier *Buddy*, verbleibt als Loch `B` von 64 Bytes
3. `free(A,  64)`, freier *Buddy* `B`, verschmilzt zu Loch `AB` von 128 Bytes
4. `free(C, 128)`, freier *Buddy* `AB`, verschmilzt zu Loch `ABC` von 256 Bytes
   - freier *Buddy* `D`, verschmilzt zu Loch `ABCD` von 512 Bytes
5. `free(E, 512)`, freier *Buddy* `ABCD`, verschmilzt zum Loch von 1024 KiB

<table>
<tr><td></td><td>64</td><td>64</td><td>128</td><td>256</td><td>512 KiB</td></tr>
<tr><td>1</td><td>64</td><td>64</td><td>128</td><td>256</td><td>512 KiB</td></tr>
<tr><td>2</td><td>64</td><td>64</td><td>128</td><td>256</td><td>512 KiB</td></tr>
<tr><td>3</td><td colspan="2">128</td><td>128</td><td>256</td><td>512 KiB</td></tr>
<tr><td>4</td><td colspan="3">256</td><td>256</td><td>512 KiB</td></tr>
<tr><td>4</td><td colspan="4">512</td><td>512 KiB</td></tr>
<tr><td>5</td><td colspan="5">1024 KiB</td></tr>
</table>

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – XII.3 Speicherverwaltung: Virtualisierung

Wolfgang Schröder-Preikschat

11. Januar 2022

Agenda

Gliederung

Lehrstoff

- **Speichervirtualisierung** im Detail behandeln und in Bezug auf ihre beiden zentralen Aufgaben untersuchen:

  Ladestrategie (*fetch policy*)
  - wann muss ein Datum im Hauptspeicher liegen?

  Ersetzungsstrategie (*replacement policy*)
  - welches Datum im Hauptspeicher ist ersetzbar?

- **Vor- und Nachteile** erkennen, das heißt, als optionales Merkmal der Speicherverwaltung eines Betriebssystems verstehen
  - als benutzerorientiertes oder systemorientiertes Kriterium begreifen
  - entweder einen Prozess oder mehrere Prozesse weitestgehend unabhängig von der Größe des Hauptspeichers ermöglichen
  - bei vielen Prozessen, den Grad an **Mehrprogrammbetrieb** maximieren

- dynamische Bindung zwischen virtueller Adresse eines Prozesses und realer Adresse im **Arbeitsspeicher** verinnerlichen
  - d.h., die **Synergie** von Haupt- und Ablagespeicher
  - desselben Rechensystems oder verschiedener (vernetzter) Rechensysteme

Gliederung

Einlagerung der Gebrauchsstücke

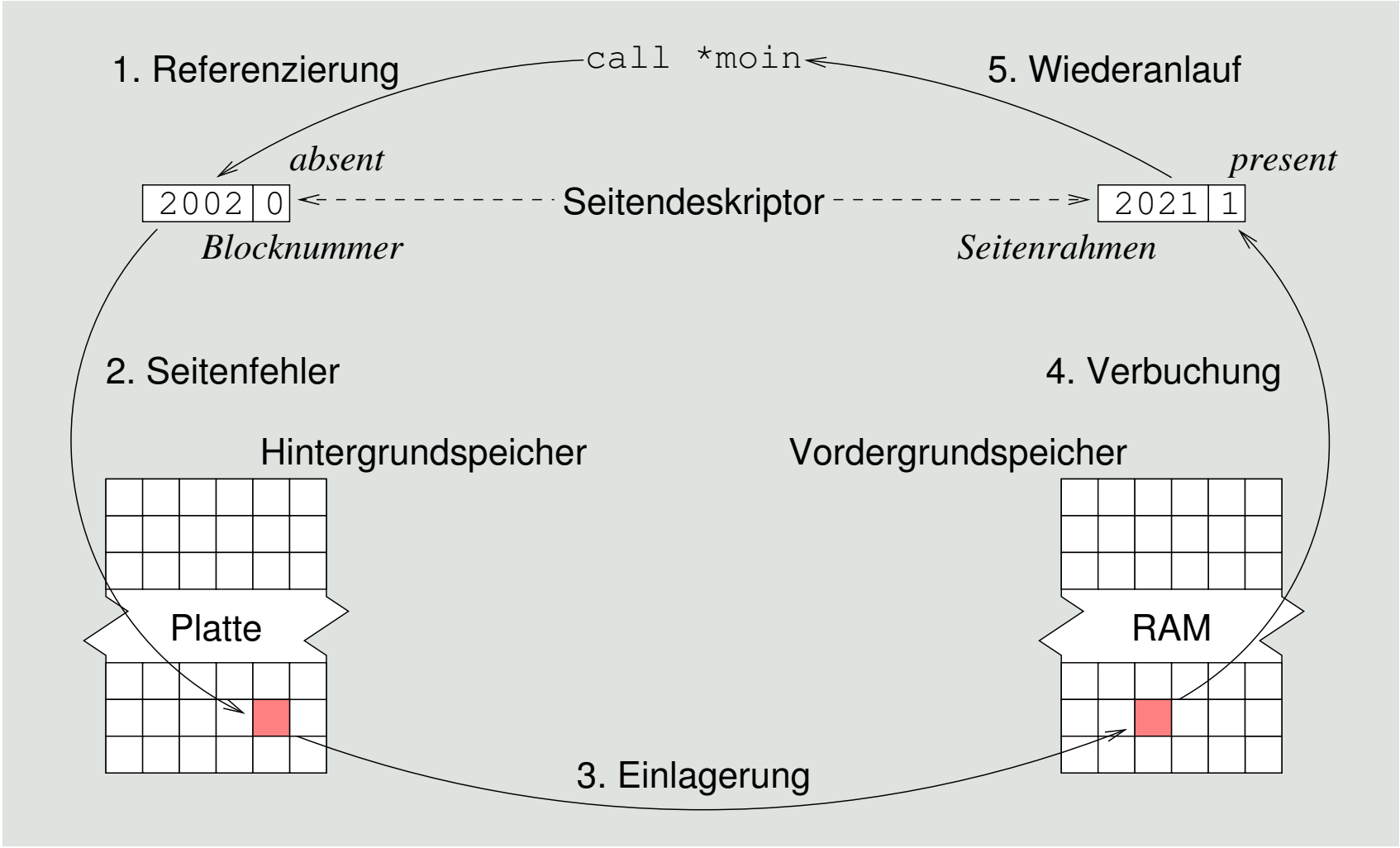
Hinweis (Gebrauchsstück (vgl. [11, S. 14]))

*Ein **umlagerungsfähiger Bestandteil** eines Prozessadressadraums, in Format (Seite, Segment) und Größe (fest, veränderlich) bestimmt durch die Adressumsetzungseinheit (MMU).*

- **Einzelanforderung**: *on demand*
  - gesteuert durch das **Präsenzbit** eines jedem Stücks
    anwesend – Zugriff möglich
    abwesend – *page/segment fault*
  - bei Abwesenheit erfolgt die **partielle Interpretation** des Zugriffs
    - Ausnahmebehandlung
    - durch das Betriebssystem
  - veranlasst durch MMU (*trap*)
- **Vorausladen**: *anticipatory*
  - der Einzelanforderung höchstmöglich zuvorkommen
    - **Vorabruf** (*prefetch*)
    - Vermeidung von Folgefehlern
  - **Heuristiken** liefern Hinweise über zukünftige Zugriffe
    - Prozesslokalität
    - Arbeitsmenge (*working set*)
  - veranlasst durch Betriebssystem
- ggf. fällt die **Verdrängung** (Ersetzung) von anwesenden Stücken an
  - wenn die Platzierungsstrategie keinen Platz im Hauptspeicher findet ☹

Einzelanforderung

- die **Seitenumlagerungsfunktion** (*pager*) des Betriebssystems



Einzelanforderung mit Vorausladen

vgl. S. 7

Hinweis (`call *moin` — und mehr dieser Art)

Bei der Ausführung ein und desselben Maschinenbefehls durch die CPU kann es mehr als einen Zugriffsfehler geben.

- Vorbeugung ggf. nachfolgender Zugriffsfehler desselben Befehls
  1. den gescheiterten Befehl dekodieren, Adressierungsart feststellen
  2. da der Operand die Adresse einer Zeigervariablen (`moin`) ist, den Adresswert auf Überschreitung einer Seitengrenze prüfen
  3. da der Befehl die Rücksprungadresse stapeln wird, die gleiche Überprüfung mit dem Stapelzeiger durchführen
  4. in der Seitentabelle die entsprechenden Deskriptoren lokalisieren und prüfen, ob die Seiten anwesend sind
     - jede abwesende Seite (*present bit* $= 0$) ist einzulagern
  5. da jetzt die Zeigervariable (`moin`) vorliegt, sie dereferenzieren und ihren Wert auf Überschreitung einer Seitengrenze prüfen
     - hierzu wie bei 4. verfahren
  6. den unterbrochenen Prozess den Befehl wiederholen lassen

  ↪ **Teilemulation** fast aller Maschinenbefehle durch das Betriebssystem ☹

Zugriffsfehler

- **Seitenfehler** (*page fault*) oder **Segmentfehler** (*segment fault*)

  present bit = 0

  - je nach Befehlssatz und Adressierungsarten der CPU kann der **Behandlungsaufwand** im Betriebssystem und somit der **Leistungsverlust** beträchtlich sein
  - erkannt durch die MMU, definiert im Betriebssystem

- Fallstudie: Aufruf einer Prozedur indirekt über einen Funktionszeiger

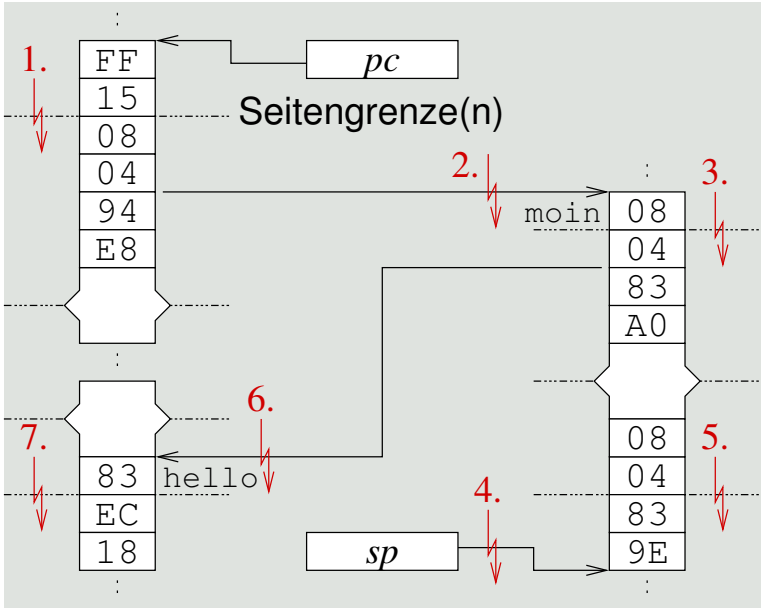
```
1  void hello () {
2      printf("Hi!\n");
3  }
4
5  void (*moin)() = &hello;
6
7  main () {
8      (*moin)();
9  }
```

```
10 main:
11         pushl %ebp
12         movl  %esp,%ebp
13         pushl %eax
14         pushl %eax
15         andl  $-16,%esp
16         call  *moin          FF15080494E8
17         leave
18         ret
```

- wie viele Seitenfehler sind bei Ausführung dieses einen Befehls möglich?

Zugriffsfehler: Schlimm(st)er Fall eines Seitenfehlers

- Aufruf einer indirekt adressierten Prozedur: `call *moin`
  - Operationskode (`FF15`) sei bereits gelesen, Seitengrenzen sind gezogen



  - Interpretation des Befehls (`call`):
    1. Operandenadresse holen (`080494E8`)
    2. Funktionszeiger lesen (`08`)
    3. Funktionszeiger weiterlesen (`0483A0`)
    4. Rücksprungadresse stapeln (`0804`)
    5. Rücksprungadresse weiterstapeln (`839E`)
  - Aufnahme des Folgebefehls (`sub`):
    6. Operationskode holen (`83`)
    7. Operanden holen (`EC18`)
  - Seitenfehler 6. und 7. sind bereits der Ausführung des ersten Befehls der aufgerufenen Prozedur (`hello`) zuzurechnen
- **Ausrichtung** (*alignment*) von Programmtext und -daten im logischen Adressraum hilft, die Anzahl von Seitenfehlern zu reduzieren
  - eine Aufgabe des Kompilierers, Assemblierers oder Binders

Aufwandsabschätzung von Einzelzugriffen

- Speichervirtualisierung bringt **nichtfunktionale Eigenschaften** bei der Ausführung von Programmen mit sich
  - **Interferenz** in Bezug auf den Prozess, der einen Zugriffsfehler produziert
- ein Seitenfehler bewirkt eine Prozessverzögerung, dehnt die **effektive Zugriffszeit** (*effective access time*, *eat*) auf den Hauptspeicher
  - hängt stark ab von der **Seitenfehlerwahrscheinlichkeit** (p) und verhält sich direkt proportional zur **Seitenfehlerrate**:

$$eat = (1 - p) \cdot pat + p \cdot pft, 0 \leq p \leq 1$$

  - angenommen, folgende Systemparameter sind gegeben:
    - 50 ns Zugriffszeit auf den RAM (*physical access time*, *pat*)
    - 10 ms mittlere Zugriffszeit auf eine Festplatte (*page fault time*, *pft*)
    - 1 % Wahrscheinlichkeit eines Seitenfehlers ($p = 0,01$)
  - dann ergibt sich:
    $eat = 0,99 \cdot 50\,\text{ns} + 0,01 \cdot 10\,\text{ms} = 49,5\,\text{ns} + 10^5\,\text{ns} \approx 0,1\,\text{ms}$

$\Rightarrow$ Einzelzugriffe sind im Ausnahmefall um den Faktor 2000 langsamer !

Aufwandsabschätzung bei Folgezugriffen

- anzunehmen ist eine **mittlere Zugriffszeit** (*mean access time*, *mat*) auf den Hauptspeicher
  - hängt stark ab von der effektiven **Seitenzugriffszeit** und der Anzahl der **Seitenreferenzierungen**:
    - je nach Seitengröße (in Bytes pro Seite) und Verweildauer

$$mat = (eat + (sizeof(page) - 1) \cdot pat)/sizeof(page)$$

  - angenommen, folgende Systemparameter sind gegeben:
    - Seitengröße von 4 096 Bytes (4 KB)
    - 50 ns Zugriffszeit (*pat*) auf ein Byte im RAM
    - effektive Zugriffszeit (*eat*) wie eben berechnet bzw. abgeschätzt
  - dann ergibt sich: $mat = (eat + 4\,095 \cdot 50\,\text{ns})/4\,096 \approx 74,41\,\text{ns}$
    - aber nur, wenn jede Speicherstelle der Seite einmal referenziert wird

$\Rightarrow$ daraus resultiert bestenfalls eine Verlangsamung um den Faktor 1,5

- ***Seitenfehler sind nicht wirklich transparent***...
  - ihre Auswirkungen stehen und fallen mit der **Ersetzungsstrategie**
  - ihre Häufigkeiten stehen und fallen zusätzlich mit der **Prozesslokalität**

Gliederung

Zeitmultiplex von Seitenrahmen

Hinweis (Speichervirtualisierung)

Abbildung der logisch abzählbar unendlichen Menge von Seiten eines oder mehrerer virtueller Adressräume auf die abzählbar beschränkte Menge von Seitenrahmen des realen Adressraums.

- logisch, da angenommen wird, dass die Anzahl virtueller Adressräume im Allgemeinen unbestimmbar ist
  - jeder virtuelle Adressraum ist einem Prozessexemplar zugeordnet
  - die Anzahl dieser Exemplare ist in Mehrbenutzersystemen i.A. unbekannt
- aber physisch ist diese Anzahl auf Grund der **endlichen Darstellung** solcher Adressräume in Rechensystemen nach oben beschränkt
  - jeder virtuelle Adressraum ist durch mehrere Datenstrukturen beschrieben
  - Datenstrukturen belegen Speicherplatz, der nur begrenzt vorhanden ist
- gleichwohl ist die Seitenmenge (virtueller Adressraum) größer als die Seitenrahmenmenge (realer Adressraum)
  - mehrere Seiten müssen sich ein und denselben Seitenrahmen teilen
  - Seitenrahmeninhalte sind durch (logisch) verschiedene Seiten zu ersetzen

Menge residenter Seiten

Definition (*resident set*)

Die abzählbar endliche Menge der Seiten des virtuellen Adressraums eines Prozessexemplars, die gegenwärtig auf Seitenrahmen abgebildet ist und damit im Hauptspeicher platziert vorliegt.

- das Betriebssystem entscheidet, wie viel Seiten in den Hauptspeicher gebracht, d.h., wie viel Seitenrahmen einem Prozess zugeteilt werden
  - nicht vorliegende Seiten werden automatisch nachgezogen (Ladestrategie)
- **verschiedene Faktoren** bestimmen die Anzahl residenter Seiten, die einem jeweiligen Prozess zugestanden werden können/sollten:
  i. je kleiner die Menge, desto höher der **Grad an Mehrprogrammbetrieb**
  ii. bei zu kleiner Menge steigt jedoch die **Seitenfehlerwahrscheinlichkeit**
  iii. und bei zu großer Menge sinkt die **Seitenfehlerrate** nur unwesentlich
- vor diesem Hintergrund wird die Menge residenter Seiten einem jeden Prozess in **fester oder variabler Größe** zugeteilt
  - wobei die Befehlssatzebene eine **Mindestmenge** vorgibt (vgl. S. 10)
  - entsprechend folgt Ersetzung einem statischen oder dynamischen Ansatz

Einzugsbereich der Seitenersetzung

- die Ladestrategie fordert, dass ein Seitenrahmen zur Aufnahme der jeweils einzulagernden Seite verfügbar sein muss
  - solange nicht alle einem Prozess zugeteilten Seitenrahmen belegt sind, ist die Entscheidung, wohin die Seite einzulagern ist, einfach
    - der erstbeste zugeteilte unbelegte (d.h., freie) Seitenrahmen wird genommen
  - anderenfalls ist ein Seitenrahmen freizumachen, d.h., die dort eingelagerte Seite ist durch die einzulagernde Seit zu ersetzen
    - es ist zu prüfen, ob die eingelagerte Seite zuvor ausgelagert werden muss !
- dazu geschieht eine **lokale oder globale Suche** nach Seitenrahmen, die zur Einlagerung der Seite freigemacht werden können

  lokal
  - Suchraum ist nur die Menge der residenten Seiten des Prozesses, der den Seitenfehler verursacht hat
  - ein Seitenfehler ist vorhersag-/reproduzierbar [10, S. 12]

  global
  - Suchraum ist die Menge aller residenten Seiten aller Prozesse im System, unabhängig vom Verursacher des Seitenfehlers
  - ein Seitenfehler ist unvorhersag-/unreproduzierbar [10, S. 14]
- so besteht ein Zusammenhang zwischen der Anzahl residenter Seiten eines Prozesses und dem Wirkungskreis der Seitenersetzung

Gegenüberstellung

vgl. [14, S. 364]

Zuteilung	***Ersetzung***	
	lokal	*global*
fest	Die Anzahl der Seitenrahmen des Prozesses ist fest. Die zu ersetzende Seite wird unter den dem Prozess zugeteilten Seitenrahmen ausgewählt.	Nicht möglich!
variabel	Die Anzahl der Seitenrahmen des Prozesses ist nicht fest, sie variiert mit der **Arbeitsmenge** (S. 27) über die Zeit. Die zu ersetzende Seite wird unter den dem Prozess zugeteilten Seitenrahmen ausgewählt.	Die Anzahl der Seitenrahmen des Prozesses ist nicht fest, sie kann sich zu seiner **Lebenszeit** verändern. Die zu ersetzende Seite wird unter allen verfügbaren Seitenrahmen ausgewählt.

- feste Zuteilung von Seitenrahmen erfolgt spätestens zur **Ladezeit** des Maschinenprogramms, d.h., zum Zeitpunkt der Prozesserzeugung
  - basierend auf Anwendungswissen oder Vorgabe des Betriebssystems

Lokalitätsprinzip

Hinweis

Ein Prozess, der zu einem Zeitpunkt eine Stelle in seinem Adressraum referenziert, wird mit gewisser Wahrscheinlichkeit dieselbe Stelle oder eine andere Stelle in direkter Umgebung referenzieren.

- festgelegt in der **Programmvorschrift** — aber auch bestimmt durch Struktur und Übersetzung der jew. Software (vgl. [3, S. 250]):
  - i Programmausführung ist meist sequentiell, d.h., der nächste abzurufende Maschinenbefehl folgt dem jetzigen an nächster Stelle
  - ii Programmschleifen umfassen vergleichsweise wenig Maschinenbefehle, d.h., Berechnungen finden in kleinen begrenzten Bereichen statt
  - iii Programmberechnungen verwenden oft große Datenstrukturen wie Felder, Verbünde, Dateien, d.h., sie neigen zu benachbarten Datenzugriffen
- daher kann davon ausgegangen werden, dass die **Referenzfolge** eines Prozesses abschnitt- bzw. phasenweise nahezu stetig ist
  - sie gut abschätzen zu können, ist für jedes Ersetzungsverfahren wichtig
  - hier helfen „klebrige“ **Statusbits** im Seitendeskriptor (vgl. [9, S. 15])

Seitenersetzung ist eine Zukunftsfrage. . .

Auswahl jenes Seitenrahmens, dessen Seite (am längsten) nicht mehr referenziert werden wird.

- diese Strategie ist nachweislich optimal, da sie die Anzahl möglicher Seitenfehler in Bezug auf eine beliebige Referenzfolge minimiert [1]

  MIN
  - wähle eine Seite, die zukünftig nicht erneut referenziert wird
  - falls es keine solche Seite gibt, wähle eine, auf die relativ zur Zeit des Seitenfehlers am weitesten entfernt zugegriffen werden wird
  - auch als OPT bezeichnet

- gleichsam ist sie unrealistisch, nicht implementierbar, weil praktisch keine Referenzfolge eines Prozesses im Voraus bekannt ist
  - Ausnahmen (Echtzeitsysteme: WCET-Analyse) bestätigen die Regel
    - der Ablaufpfad in seinem Programm ist absehbar ?
    - die diesen Pfad beeinflussende Eingabewerte sind vorherbestimmt ?
    - und Verzweigungen durch Programmunterbrechungen sind vorhersagbar ?
  - bestenfalls ist es möglich, eine gute **Approximation** anzugeben

Approximation der optimalen Strategie

- Grundlage bildet Wissen über die **Vergangenheit** und **Gegenwart** von Seitenzugriffen, um Annahmen über die **Zukunft** zu treffen:

  FIFO (*first-in, first-out*) ✓
  - ersetzt wird die zuerst eingelagerte Seite $\rightsquigarrow$ verketten
  - Belady's Anomalie [2]: mehr Seitenrahmen, ggf. mehr Seitenfehler

  LFU (*least frequently used*)
  - ersetzt wird die am seltensten referenzierte Seite $\rightsquigarrow$ zählen
  - Alternative: MFU (*most frequently used*)

  LRU (*least recently used*) ✓
  - ersetzt wird die am längsten nicht mehr referenzierte Seite
    - pro Seitendeskriptor **Zeitstempel** setzen oder **Kondensator** aufladen
    - eine **Stapeltechnik** einsetzen: die referenzierte Seite „oben ablegen"
    - die **Alterungsstruktur** einer Seite in einem Schieberegister erfassen ☺
    - bzw. weniger aufwendig durch einzelne **Statusbits** abschätzen ☺
  - die Seite mit dem **größten Rückwärtsabstand** wird ausgewählt
    - größter Zeitabstand, kleinste Ladung, unten liegend, wenigsten Einsen

- die Effektivität der Verfahren hängt ab von der **Prozesslokalität**, der Stetigkeit seiner gegenwärtigen Referenzfolge

LRU: Alterungstruktur

aging

- dazu ist jedem Seitendeskriptor ein **Schieberegister** (*aging register*) zugeordnet, meist in Software implementiert (Schattendeskriptor)
- zusätzlich wird ein **Zeitgeber** (*timer*) benötigt, der eine **periodische Unterbrechung** des Prozesses verursacht $\rightsquigarrow$ **Hintergrundrauschen**
- bei jedem Ablauf des Zeitintervalls wird das **Referenzbit** einer jeden eingelagerten Seite des unterbrochenen Prozesses eingeprüft:
  - i ist es gesetzt, wurde die Seite referenziert: das Bit wird zurückgesetzt
  - ii ist es nicht gesetzt, wurde die Seite nicht referenziert

tick	Ref.	*aging register*
		00000000
n	1	10000000
$n+1$	1	11000000
$n+2$	0	01100000
$n+3$	1	10110000
$\vdots$	$\vdots$	$\vdots$

- Registerinhalt (*age*) als Ganzzahl interpretiert liefert ein Maß für die Aktivität einer Seite
- mit abnehmendem Betrag, d.h. einer sinkenden Prozessaktivität, steigt die Ersetzungspriorität
- bei Seitenfehler/-ersetzung wird die global älteste Seite gewählt

LRU: zweite Chance I

second-chance replacement

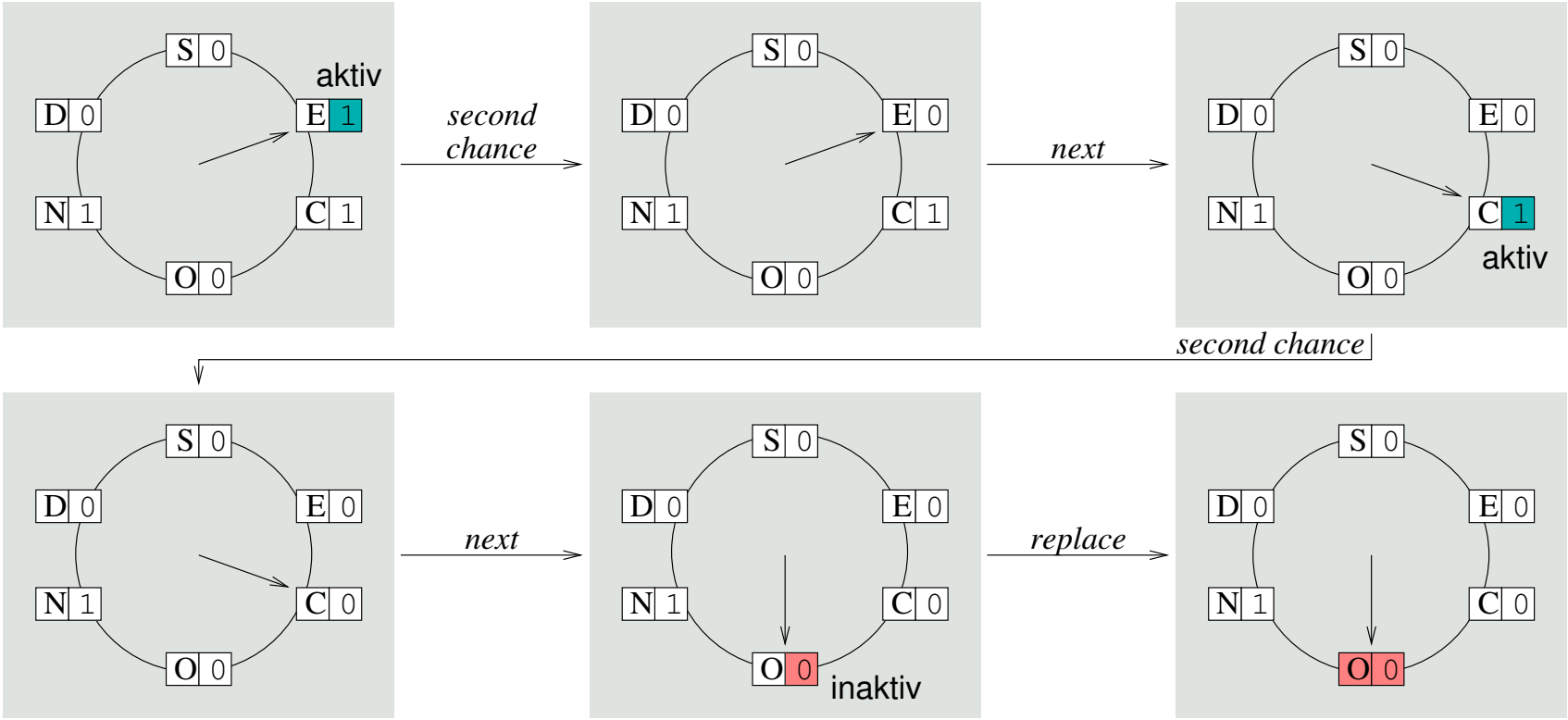
- arbeitet im Grunde nach **FIFO**, berücksichtigt jedoch zusätzlich noch **Statusbits** der jeweils in Betracht zu ziehenden Seiten
- nutzt einen **Zeitgeber** (*timer*), der eine **periodische Unterbrechung** des Prozesses verursacht $\rightsquigarrow$ **Hintergrundrauschen**
- bei jedem Ablauf des Zeitintervalls wird das **Referenzbit** (*use*) einer eingelagerten Seite des unterbrochenen Prozesses geprüft:

use	Aktion	Bedeutung
1	Referenzbit zurücksetzen	Seite erhält eine zweite Chance
0	—	Seite kann ersetzt werden

  - bei einem Seitenfehler, der eine Seitenersetzung nach sich zieht, wird die global zuerst eingelagerte und unreferenzierte Seite gewählt
- schlimmstenfalls erfolgt ein Rundumschlag über alle Seiten, wenn die Referenzbits aller betrachteten Seiten (auf 1) gesetzt waren
  - die Strategie „entartet“ dann zu FIFO

LRU: zweite Chance II

clock replacement



- Annahme ist, referenzierte Seiten sind vermeintlich aktive Seiten:
  - E: aktiv, Referenzbit zurücksetzen, Seite im Hauptspeicher behalten
  - C: aktiv, Referenzbit zurücksetzen, Seite im Hauptspeicher behalten
  - O: inaktiv, Seite ist ersetzbar, Seitenrahmen für andere Seite nutzen
- allgemein konzipiert für $n \geq 0$ *use*-Bits pro Seite [4]: FIFO bei $n = 0$, bestes Kosten-Nutzen-Verhältnis bei $n = 1$, Alterung bei $n > 1$

LRU: dritte Chance

third-chance replacement

- übernimmt alle Merkmale von *second chance*
  - FIFO als Basis, periodische Unterbrechung, Referenzbit (*use bit*)
  - Hintergrundrauschen
- zusätzlich zum Referenz- wird das **Modifikationsbit** (*dirty bit*) jeder eingelagerten Seite des unterbrochenen Prozesses geprüft
  - dieses Bit wird bei Schreibzugriffen gesetzt, bleibt sonst unverändert
  - zusammen mit dem Referenzbit zeigen sich vier Paarungen (*use*, *dirty*):

	Bedeutung	Entscheidung
(0, 0)	ungenutzt	beste Wahl
(0, 1)	beschrieben	keine schlechte Wahl
(1, 0)	kürzlich gelesen	keine gute Wahl
(1, 1)	kürzlich beschrieben	schlechteste Wahl

  - ausgewählt wird die global zuerst eingelagerte, unreferenzierte und wenn möglich unveränderte Seite
- kann für jede eingelagerte Seite zwei Umläufe erwirken, gibt aktiven, d.h., referenzierten Seiten damit eine dritte Chance [8]
  - auch als ***enhanced second chance*** bezeichnet

Seitenflattern

thrashing

Kritisches Systemverhalten, wenn die durch Seitenein-/auslagerungen verursachte E/A die gesamten Systemaktivitäten dominiert [5]

- eben erst ausgelagerte Seiten werden sofort wieder eingelagert
  - es wurde die falsche Seite ausgewählt, die Vorhersage stimmte nicht
  - Folge: Prozesse verbringen mehr Zeit beim Umlagern als beim Rechnen
- ein mögliches **Phänomen der globalen Seitenersetzung**
  - Prozesse bewegen sich zu nahe am Seitenrahmenminimum
    - die (fest) oder ihre jeweilige (variabel) Menge residenter Seite ist zu klein
    - d.h., sie ist kaum größer als die durch die Hardware definierte Mindestmenge
  - zum Lastprofil eher ungünstige Ersetzungsstrategie
  - zu hoher Grad an Mehrprogrammbetrieb
- verschwindet ggf. so plötzlich von allein, wie es aufgetreten ist...

Hinweis

Ein ernstes Problem, das allerdings auch immer in Relation zu der Zeit, die Prozesse mit sinnvoller Arbeit verbringen, zu setzen ist.

Freiseitenpuffer

page buffering

- arbeitet im Grunde nach **FIFO**, verwaltet aber zusätzlich noch einen **Zwischenspeicher** (*cache*) potentiell zu ersetzender Seiten [13]
  - getrennt in zwei Listen für modifizierte und unmodifizierte Seiten
  - modifizierte Seiten sind auszulagern, bevor sie ersetzt werden können
- Seiten im Zwischenspeicher sind logisch abwesend, ihr **Präsenzbit** ist gelöscht, physisch aber noch anwesend, bis sie ersetzt wurden
  - sie kommen jeweils als **Fußseite** in die ihrem Zustand entsprechende Liste
- bei einem Zugriffsfehler auf eine im Zwischenspeicher liegende Seite, erfolgt ihre **Reklamierung** und ihr Präsenzbit wird wieder gesetzt
  - anderenfalls wird die **Kopfseite** (aus einer der beiden Listen) entfernt
  - ausgewählt wird die lokal zuerst eingelagerte und längst ungenutze Seite
- der Zwischenspeicher ist die **Reserve** „ungebundener“ Seitenrahmen, *pager*-gesteuert durch **Schwellwerte** $\rightsquigarrow$ **Hintergrundrauschen**:
  - *low*: Seitenrahmen als frei markieren, Seiten zwischenspeichern
  - *high*: Seitenrahmen liegen brach, Seiten einlagern $\rightsquigarrow$ **Vorausladen**
  - damit entspricht die Ersetzungszeit einer Seite ihrer Ladezeit

Arbeitsmenge

working set (vgl. [6, S. 326] und [7])

Definition (Standpunkt eines Prozesses)

Die kleinste Sammlung von Programmtext und -daten, die in einem Hauptspeicher vorliegen muss, damit effiziente Programmausführung zugesichert werden kann.

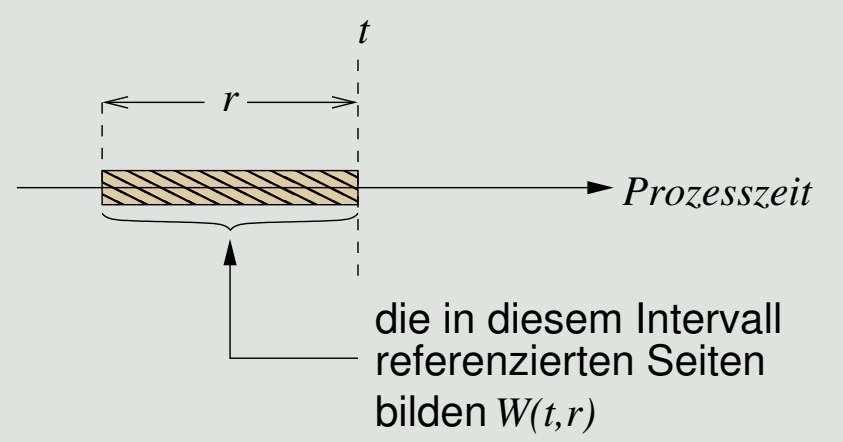
- eine Forderung, die ohne **exakte Voranzeigen** zum Platzbedarf über die Zeit der Programmausführung nicht umsetzbar ist
  - damit einhergehendes **Vorabwissen** ist nicht verfügbar, es lässt sich nie vollständig durch statische Analyse der Programme herleiten
  - wie viel Hauptspeicher faktisch belegt sein wird, ist nur **zur Laufzeit** durch Aufzählung der wirklich benutzten Seiten feststellbar

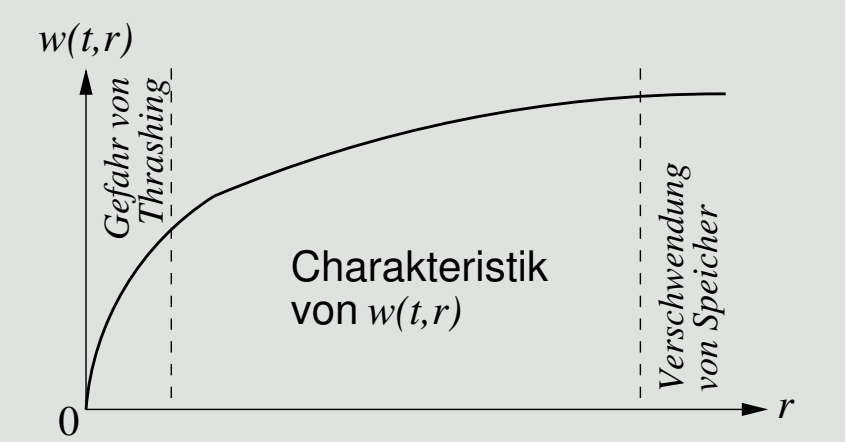
Definition (Standpunkt eines Betriebssystems)

Die **Teilmenge** WS der zuletzt (*most recently*) referenzierten Seiten eines Prozesses aus der Menge seiner residenten Seiten RS, $WS \subseteq RS$.

Aktive Seiten eines Prozesses

- die Menge von Seiten ***W(t, r)***, die ein Prozess zum Zeitpunkt t und im vorangegangenen relativen Zeitfenster r in Benutzung hatte:
  - t Beobachtungszeitpunkt
  - r Arbeitsmengenparameter (*working set parameter*), Konstante
- die Anzahl aktiver Seiten in $W(t, r)$, die also referenziert und damit genutzt wurden, legt die **Arbeitsmengengröße** $\omega(t, r)$ fest



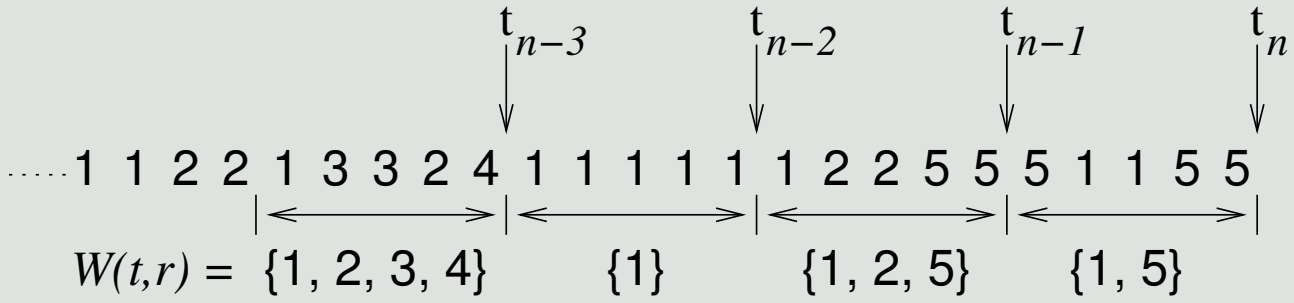


- die im **Zeitinterval** $(t - r, t)$ jüngst von einem Prozess referenzierten Seiten werden von ihm wahrscheinlich weiter benutzt
  - nur einzelne davon auszulagern, erhöht das Risiko zum Seitenflattern...

Arbeitsmengendynamik

moving working set window

- die Arbeitsmengengröße hängt von der **Prozesslokalität** ab und kann nur näherungsweise bestimmt werden
  - gegeben sei die **Seitenreferenzfolge** der jüngeren Vergangenheit
  - darauf wird ein **Arbeitsmengenfenster** (*working set window*) geöffnet
    - dieses bewegt sich schrittweise vorwärts mit jeder weiteren Referenz
- die **Fensterbreite** r gibt eine „feste Anzahl von Maschinenbefehlen“
  - realisiert als **Zeitfenster** (*timer*), das eine **periodische Unterbrechung** des Prozesses verursacht $\rightsquigarrow$ **Hintergrundrauschen**
  - die **Periodenlänge** macht in etwa eine feste Anzahl von Befehlen aus



- zu kleine Fenster halten benutzte Seiten draußen (Seitenfehlerrate steigt), zu große halten unbenutzte Seiten drinnen (Speicherverschwendung)
- WS ähnelt lokalem LRU, begrenzt aber durch das Fenster den Suchraum

Operative Umsetzung

- eine Ersetzungsstrategie auf Basis des Arbeitsmengenmodells verfährt sodann nach den folgenden beiden **Regeln**:
  - i bei jeder Seitenreferenz wird die Arbeitsmenge aktualisiert, woraufhin nur die Seiten eben dieser Menge im Hauptspeicher gehalten werden
  - ii ein Prozess kann voranschreiten genau dann, wenn seine Arbeitsmenge im Hauptspeicher vollständig vorliegt
- obgleich konzeptionell attraktiv, so ist die Umsetzung des Konzepts schwierig und sie zieht auch **hohe Unkosten** nach sich
  - die Bestimmung der Fenstergröße muss empirisch geschehen, indem r solange variiert wird, bis die beste Leistung verzeichnet wird
  - die aktuelle Arbeitsmenge kann sich mit jeder Seitenreferenz verändern und damit die Zuteilung von Seitenrahmen an den Prozess ☹
- nur **Approximation** lässt eine praxistaugliche Lösung erhoffen
  - die **Alterung** von Seiten erfassen (vgl. S. 21) mit $r = \delta * n$, wobei δ das Zeitinterval festlegt und n die Bitanzahl im Alterungsregister
  - ungenutzte Seiten verlieren graduell an Bedeutung, ab einem festgelegten Schwellwert $l \geq 0$ wird die Seite aus der Arbeitsmenge entfernt
    - die Seite fiel mit Erreichen von l aus dem Arbeitsmengenfenster heraus

Gliederung

Resümee

- die **Ladestrategie** bestimmt, wann ein Datum im Hauptspeicher liegt und wie es dort hingebracht wird
  - **Einzelanforderung** (*demand paging*) oder **Vorausladen** (*anticipatory*)
  - ersteres setzt Zugriffsfehler voraus, letzteres versucht diesem vorzubeugen
- die **Ersetzungsstrategie** bestimmt, welches Datum seinen Platz im Hauptspeicher für ein anderes Datum freimachen muss
  - **globale Verfahren** untersuchen die Mengen aller residenten Seiten aller Prozesse im System, unabhängig vom Verursacher des Zugriffsfehlers
    - FIFO, LRU (*aging*, *second chance*/*clock*, *advanced second chance*)
    - als unerwünschter Effekt ist **Seitenflattern** (*thrashing*) möglich
  - wohingegen **lokale Verfahren** nur die Menge der residenten Seiten des Prozesses, der den Zugriffsfehler verursacht hat, in Betracht ziehen
    - Freiseitenpuffer, Arbeitsmenge (insb. in Kombinbation mit lokalem LRU)
    - mit letzterem Konzept sind hohe **Unkosten** (*overhead*) verbunden
- die **residente Menge von Seiten** (*resident set*) eines Prozesses ist nicht mit seiner **Arbeitsmenge** (*working set*) zu verwechseln
  - letztere ist eine Teilmenge ersterer und hoch dynamisch (s. auch S. 36)

Literaturverzeichnis I

[1] Bélády, L. A.:
A Study of Replacement Algorithms for a Virtual Storage Computer.
In: *IBM Systems Journal* 5 (1966), Nr. 2, S. 78–101

[2] Bélády, L. A. ; Nelson, R. A. ; Shelder, G. S.:
An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine.
In: *Communications of the ACM* 12 (1969), Jun., Nr. 6, S. 349–353

[3] Bic, L. F. ; Shaw, A. C.:
Operating System Principles.
Pearson Education, Inc., 2003

[4] Corbató, F. J.:
A Paging Experiment with the Multics System / Project MAC, Defense Technical Information Center.
1968. –
Forschungsbericht

Literaturverzeichnis II

[5] Denning, P. J.:
Thrashing: Its Causes and Prevention.
In: *AFIPS Conference Proceedings of the 1968 Fall Joint Computer Conference (AFIPS '68), December 9–11, 1968, San Francisco, CA, USA* Bd. 33, ACM, 1968 (Part I), S. 915–922

[6] Denning, P. J.:
The Working Set Model for Program Behavior.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 323–333

[7] Denning, P. J.:
Working Sets Past and Present.
In: *IEEE Transactions on Software Engineering* SE-6 (1980), Jan., Nr. 1, S. 64–84

[8] Goldman, P. :
Mac VM Revealed.
In: *BYTE* 14 (1989), Nov., Nr. 12, S. 350–360

[9] Kleinöder, J. ; Schröder-Preikschat, W. :
Adressräume.
In: [12], Kapitel 12.1

Literaturverzeichnis III

[10] Kleinöder, J. ; Schröder-Preikschat, W. :
Betriebssystemmaschine.
In: [12], Kapitel 5.3

[11] Kleinöder, J. ; Schröder-Preikschat, W. :
Speicherzuteilung.
In: [12], Kapitel 12.2

[12] Kleinöder, J. ; Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)

[13] Levy, H. M. ; Lipman, P. H.:
Virtual Memory Management in the VAX/VMS Operating System.
In: *IEEE Computer* 15 (1982), März, Nr. 3, S. 35–41

[14] Stallings, W. :
Operating Systems: Internals and Design Principles.
Prentice Hall, 2001

Größe der Menge residenter Seiten

Linux 3.2.0-4-amd64

```
wosch@faui40 102$ ps
  PID TTY          TIME CMD
26125 pts/11   00:00:00 csh
28439 pts/11   00:00:00 ps
wosch@faui40 103$ pidstat -r -p 26125 1
09:02:14          PID  minflt/s  majflt/s     VSZ    RSS   %MEM  Command
09:02:15        26125      0.00      0.00   10216   2028   0.00  csh
09:02:16        26125      0.00      0.00   10216   2028   0.00  csh
09:02:17        26125      0.00      0.00   10216   2028   0.00  csh
^C
wosch@faui40 104$ pidstat -r -p SELF 1
09:15:45          PID  minflt/s  majflt/s     VSZ    RSS   %MEM  Command
09:15:46        24996     14.00      0.00    4128    756   0.00  pidstat
09:15:47        24996     12.00      0.00    4128    788   0.00  pidstat
09:15:48        24996      4.00      0.00    4128    788   0.00  pidstat
^C
wosch@faui40 105$ getconf PAGESIZE
4096
```

- `csh`: $RSS = 2028\,KiB = 507$ Seiten á 4096 Bytes
- `pidstat`: $RSS = 756\,KiB = 189$ Seiten initial, dann 197 Seiten á 4 KiB

- die Menge residenter Seiten der Prozesse ist variabel (s. `pistat`) — Linux verwaltet aber keine Arbeitsmengen, die viel variabler wären

Systemprogrammierung

Grundlage von Betriebssystemen

Teil C – XIII. Dateisysteme

Jürgen Kleinöder

18. Januar 2022
25. Januar 2022

Agenda

Medien

Speicherung von Dateien

Freispeicherverwaltung

Beispiele: Dateisysteme unter UNIX und Windows

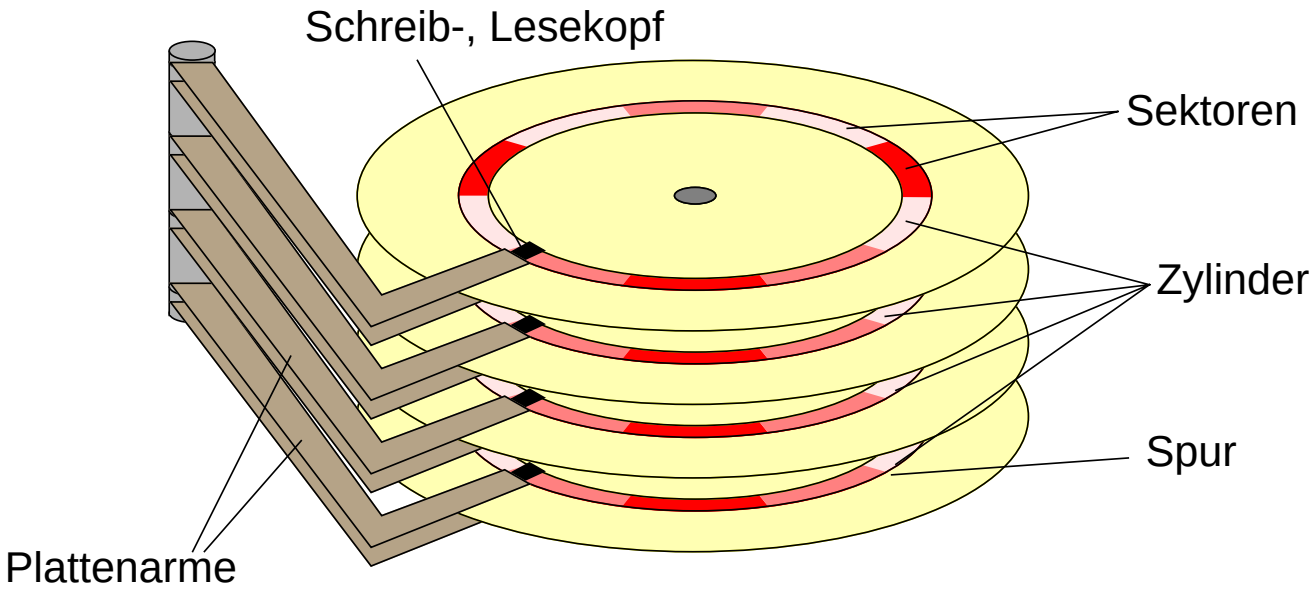
Dateisysteme mit Fehlererholung

Datensicherung

Medien

Festplatten (hard disk drive - HDD)

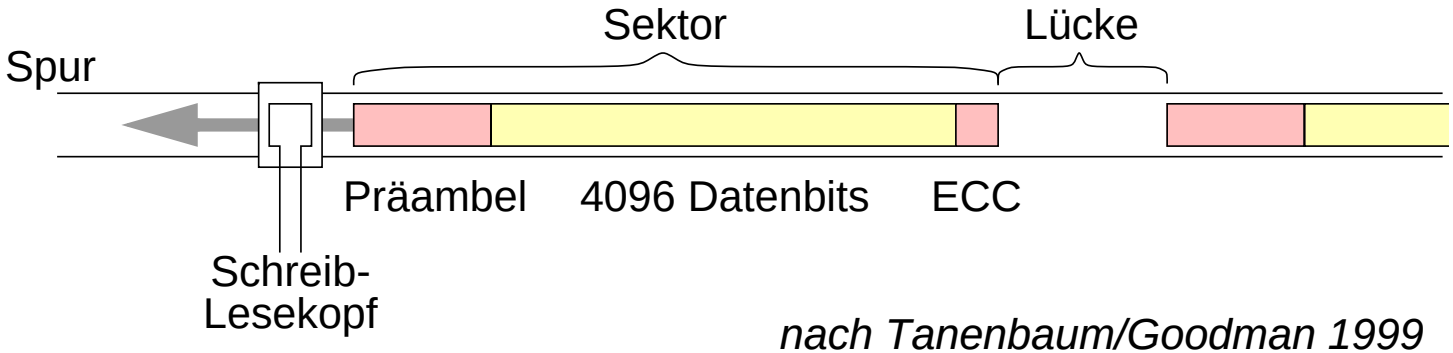
- Lange Zeit häufigstes Medium zum Speichern von Dateien
  - Aufbau einer Festplatte



  - Kopf schwebt auf Luftpolster

Festplatten (2)

- Sektoraufbau



nach Tanenbaum/Goodman 1999

  - Breite der Spur: 0,2 µm ,
    bei "Shingled Magnetic Recording" überlappende Spuren
  - Spuren pro Zentimeter: ca. 10.000
  - Bitdichte: 34 Bit/µm

- Zonen
  - Mehrere Zylinder (10–30) bilden eine Zone mit gleicher Sektorenanzahl (bessere Plattenausnutzung)

HDD / SSD

- SSD (solid-state disk)
  - Nicht-flüchtiger Halbleiter-Speicher (Flash-Mem.), meist NAND-Chips
  - Schnittstelle zum Rechner kompatibel mit HDD
  - inzwischen erheblich höhere Kapazitäten als bei HDD möglich
  - erheblich schneller und robuster (keine Mechanik)
  - noch erheblich teurer (Faktor 2 - 4)
- Zugriffsmerkmale
  - blockorientierter und wahlfreier Zugriff
  - Blockgröße zwischen 32 und 4096 Bytes (typisch 512 Bytes)
  - HDD: Zugriff erfordert Positionierung des Schwenkarms auf den richtigen Zylinder und Warten auf den entsprechenden Sektor
  - SSD: Lesen direkt und schnell, Schreiben nur blockweise und langsamer (Faktor 10), Blöcke müssen vorher explizit gelöscht worden sein
  - Interner Controller verbirgt die Hardware-Details, enthält Cache-Speicher, optimiert Zugriffe, Blöcke sind durchnummeriert

HDD / SDD (2)

- Datenblätter von drei (alten) Beispielplatten

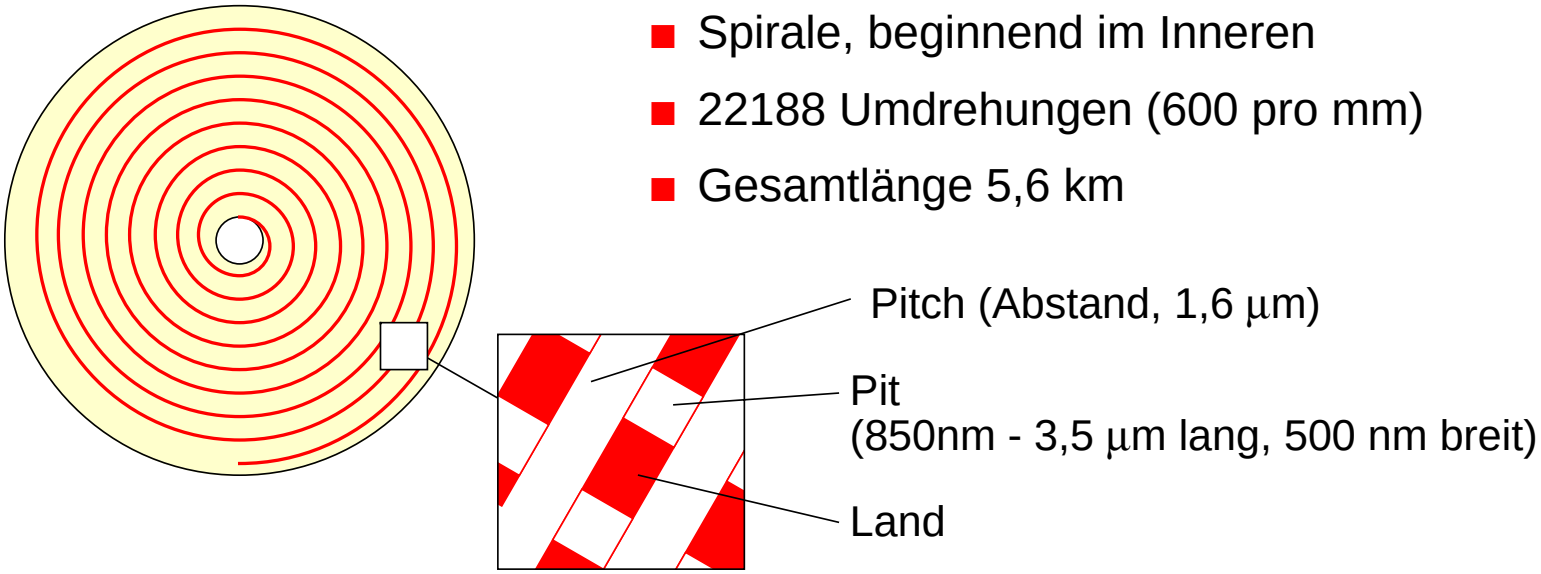
Plattentyp		Fujitsu M2344 (1987)	Seagate Cheetah	Seagate Barracuda
Kapazität		690 MB	300 GB	400 GB
Platten/Köpfe		8 / 28	4 / 8	781.422.768 Sektoren
Zylinderzahl		624	90.774	
Cache		-	4 MB	8 MB
Posititionier-zeiten	Spur zu Spur	4 ms	0,5 ms	-
	mittlere	16 ms	5,3 ms	8 ms
	maximale	33 ms	10,3 ms	-
Transferrate		2,4 MB/s	320 MB/s	-150 MB/s
Rotationsgeschw.		3.600 U/min	10.000 U/min	7.200 U/min
eine Plattenumdrehung		16 ms	6 ms	8 ms
Stromaufnahme		?	16-18 W	12,8 W

12.2020: Kapazität bis 20TB bei 7.200 U/min oder 0,6 - 2 TB bei 15.000 U/min, Zugriffszeit ab 2 ms, Transferrate bis 315 MB/s

SSD: Kapazität bis 100 TB, Zugriffszeit ab. 0,03/0,3 ms, Transferrate bis 4 GB/s

CD-ROM / DVD

- Aufbau einer CD



  - Spirale, beginnend im Inneren
  - 22188 Umdrehungen (600 pro mm)
  - Gesamtlänge 5,6 km
  - **Pit**: Vertiefung, wird von Laser (780 nm Wellenlänge) abgetastet
- DVD
  - gleiches Grundkonzept, Wellenlänge des Lasers 650 nm
  - Pits und Spurabstand weniger als halb so groß

CD-ROM / DVD (2)

- Kodierung einer CD
  - **Symbol**: ein Byte wird mit 14 Bits kodiert
    (kann bereits bis zu zwei Bitfehler korrigieren)
  - **Frame**: 42 Symbole (192 Datenbits, 396 Fehlerkorrekturbits)
  - **Sektor** : 98 Frames werden zusammengefasst
    (16 Bytes Präambel, 2048 Datenbytes, 288 Bytes Fehlerkorrektur)
  - *Effizienz* : 7203 Bytes transportieren 2048 Nutzbytes (28,4 %)
- Kodierung einer DVD
  - Codierung mit Reed-Solomon-Product-Code, 8/16-Bit-Modulation,
    43,2 % Nutzdaten
- Transferrate
  - CD-Single-Speed-Laufwerk: 75 Sektoren/Sek. (153.600 Bytes/Sek.)
  - CD-72-fach-Laufwerk: 11,06 MB/Sek.
  - DVD 1-fach: 1.3 MB/sec, 24-fach: 33.2 MB/sec

CD-ROM / DVD (3)

- Kapazität
  - CD: ca. 650 MB
  - DVD single layer: 4.7 GB
  - DVD dual layer: 8.5 GB, beidseitig: 17 GB
- Varianten
  - **DVD/CD-R** (Recordable): einmal beschreibbar
  - **DVD/CD-RW** (Rewritable): mehrfach beschreibbar

Speicherung von Dateien

- Dateien benötigen oft mehr als einen Block auf der Festplatte
  - Welche Blöcke werden für die Speicherung einer Datei verwendet?

Kontinuierliche Speicherung

- Datei wird in Blöcken mit aufsteigenden Blocknummern gespeichert
  - Nummer des ersten Blocks und Anzahl der Folgeblöcke muss gespeichert werden

★ Vorteile
  - Zugriff auf alle Blöcke mit minimaler Positionierzeit des Schwenkarms
  - Schneller direkter Zugriff auf bestimmter Dateiposition
  - Einsatz z. B. bei Systemen mit Echtzeitanforderungen

Kontinuierliche Speicherung (2)

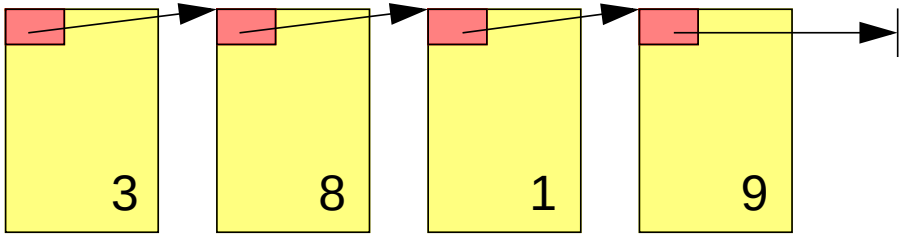
- Probleme
  - Finden des freien Platzes auf der Festplatte (Menge aufeinanderfolgender und freier Plattenblöcke)
  - Fragmentierungsproblem (Verschnitt: nicht nutzbare Plattenblöcke; siehe auch externer Verschnitt bei Speicherverwaltung)
  - Größe bei neuen Dateien oft nicht im Voraus bekannt
  - Erweitern ist problematisch
    - ➤ Umkopieren, falls kein freier angrenzender Block mehr verfügbar

Kontinuierliche Speicherung (3)

- Variation
  - Unterteilen einer Datei in Folgen von Blöcken *(Chunks, Extents)*
  - Blockfolgen werden kontinuierlich gespeichert
  - Pro Datei muss erster Block und Länge jedes einzelnen Chunks gespeichert werden
- Problem
  - Verschnitt innerhalb einer Folge (siehe auch Speicherverwaltung: interner Verschnitt bei Seitenadressierung)

Verkettete Speicherung

- Blöcke einer Datei sind verkettet



  - z. B. Commodore Systeme (CBM 64 etc.)
    - Blockgröße 256 Bytes
    - die ersten zwei Bytes bezeichnen Spur- und Sektornummer des nächsten Blocks
    - wenn Spurnummer gleich Null: letzter Block
    - 254 Bytes Nutzdaten

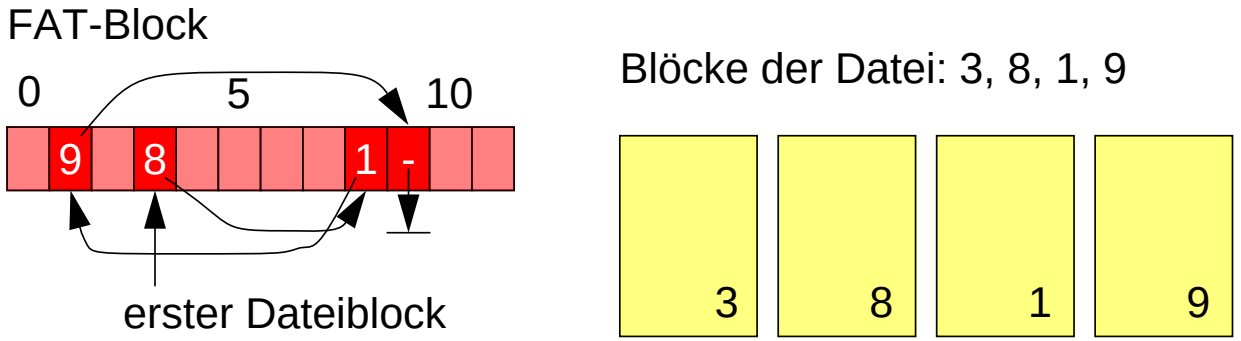
- ★ Datei kann wachsen und verlängert werden

Verkettete Speicherung (2)

- Probleme
  - Speicher für Verzeigerung geht von den Nutzdaten im Block ab (ungünstig im Zusammenhang mit Paging: Seite würde immer aus Teilen von zwei Plattenblöcken bestehen)
  - Fehleranfälligkeit: Datei ist nicht restaurierbar wenn ein Fehler in der Verzeigerung entsteht
  - schlechter direkter Zugriff auf bestimmte Dateiposition
  - häufiges Positionieren des Schreib-Lesekopfs bei verstreuten Datenblöcken

Verkettete Speicherung (3)

- Verkettung wird in speziellem Plattenblocks gespeichert
  - FAT-Ansatz *(FAT: File Allocation Table)*, z. B. MS-DOS, Windows 95



  - Dateiverzeichnis enthält zu jeder Datei die Nummer des ersten Blocks

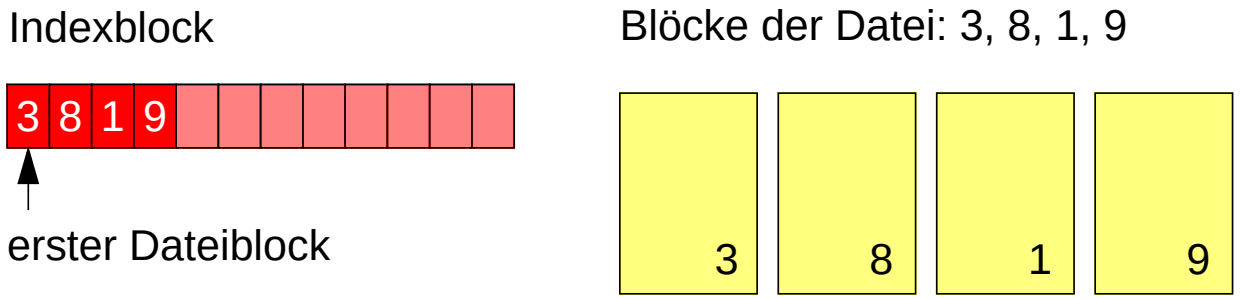
- ★ Vorteile
  - kompletter Inhalt des Datenblocks ist nutzbar (günstig bei Paging)
  - mehrfache Speicherung der FAT möglich: Einschränkung der Fehleranfälligkeit

Verkettete Speicherung (4)

- Probleme
  - mindestens ein zusätzlicher Block muss geladen werden (Caching der FAT zur Effizienzsteigerung nötig)
  - FAT enthält Verkettungen für alle Dateien: das Laden der FAT-Blöcke lädt auch nicht benötigte Informationen
  - aufwändige Suche nach dem zugehörigen Datenblock bei bekannter Position in der Datei
  - häufiges Positionieren des Schreib-, Lesekopfs bei verstreuten Datenblöcken

Indiziertes Speichern

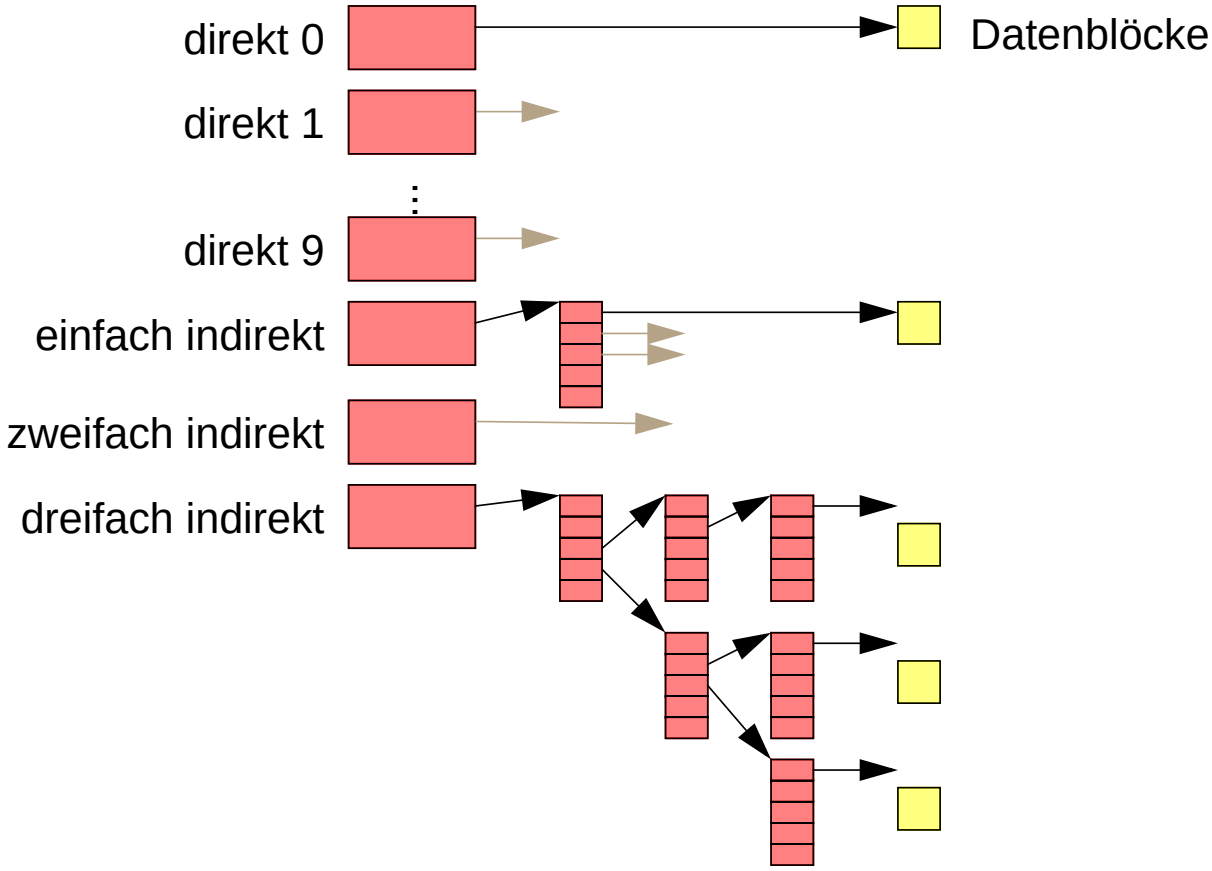
- Spezieller Plattenblock enthält Blocknummern der Datenblocks einer Datei
  - entspricht eigener FAT pro Datei



- Problem
  - feste Anzahl von Blöcken im Indexblock
    - Verschnitt bei kleinen Dateien
    - Erweiterung nötig für große Dateien

Indiziertes Speichern (2)

- Beispiel UNIX Inode



Indiziertes Speichern (3)

★ Einsatz von mehreren Stufen der Indizierung

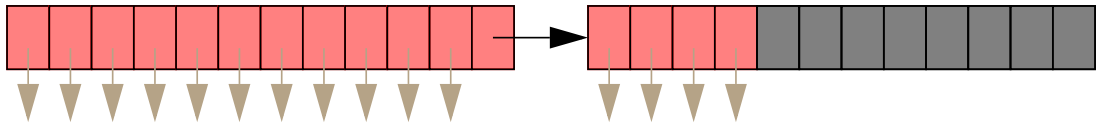
- Inode benötigt sowieso einen Block auf der Platte (Verschnitt unproblematisch bei kleinen Dateien)
- durch mehrere Stufen der Indizierung auch große Dateien adressierbar

▲ Nachteil

- mehrere Blöcke müssen geladen werden (nur bei langen Dateien)

Freispeicherverwaltung

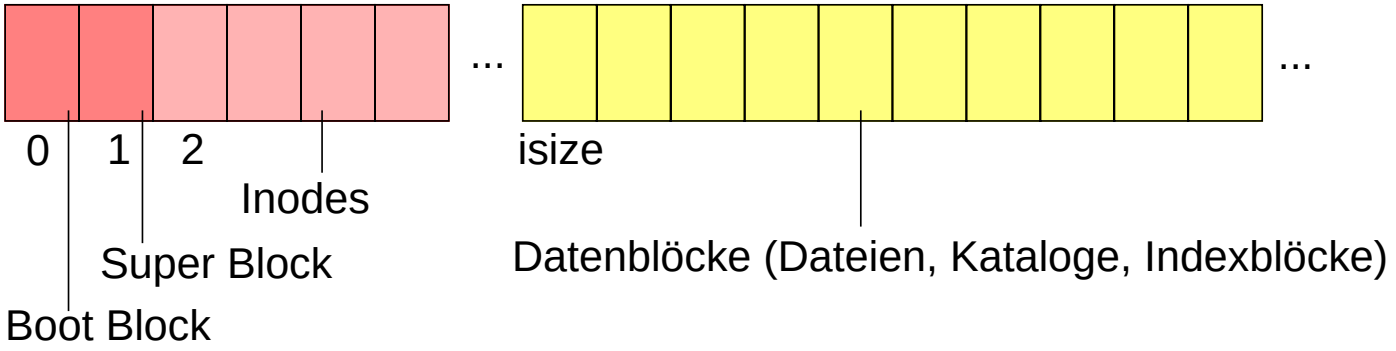
- Prinzipiell ähnlich wie Verwaltung von freiem Hauptspeicher
  - Bitvektoren zeigen für jeden Block Belegung an
  - verkettete Listen repräsentieren freie Blöcke
    - Verkettung kann in den freien Blöcken vorgenommen werden
    - Optimierung: aufeinanderfolgende Blöcke werden nicht einzeln aufgenommen, sondern als Stück verwaltet
    - Optimierung: ein freier Block enthält viele Blocknummern weiterer freier Blöcke und evtl. die Blocknummer eines weiteren Blocks mit den Nummern freier Blöcke



Beispiel: UNIX Dateisysteme

System V File System

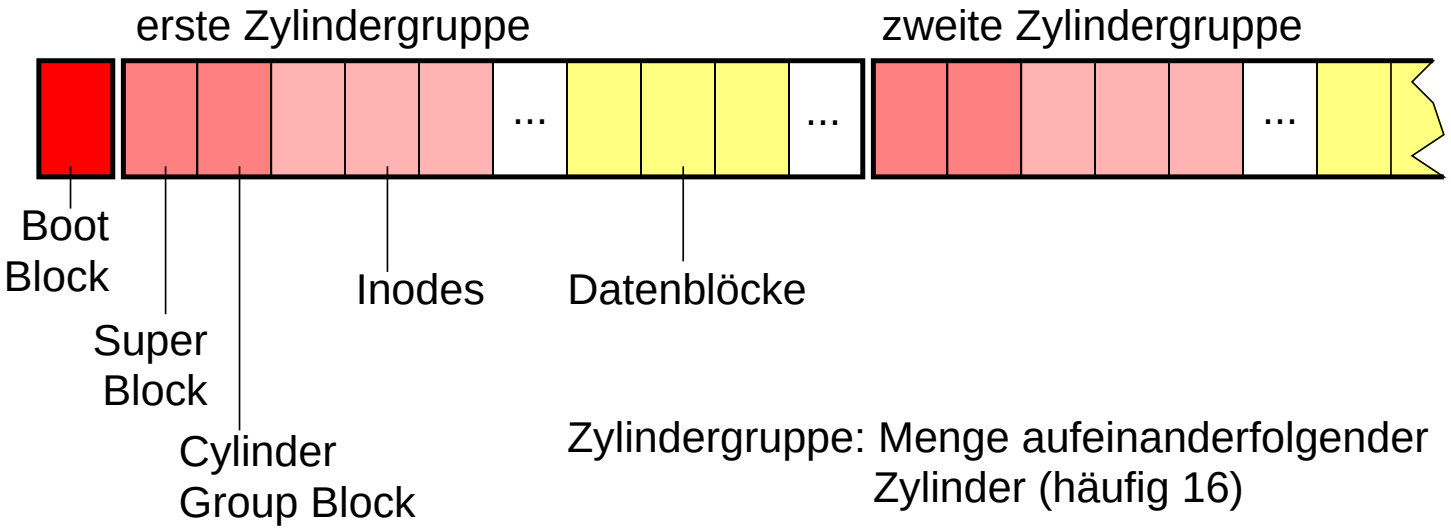
- Blockorganisation



- Boot Block enthält Informationen zum Laden eines initialen Programms
- Super Block enthält Verwaltungsinformation für ein Dateisystem
  - Anzahl der Blöcke, Anzahl der Inodes
  - Anzahl und Liste freier Blöcke und freier Inodes
  - Attribute (z.B. *Modified flag*)
- seit den 1970er Jahren in den ersten UNIX-Systemen eingesetzt

BSD 4.2 (Berkeley Fast File System)

- Blockorganisation

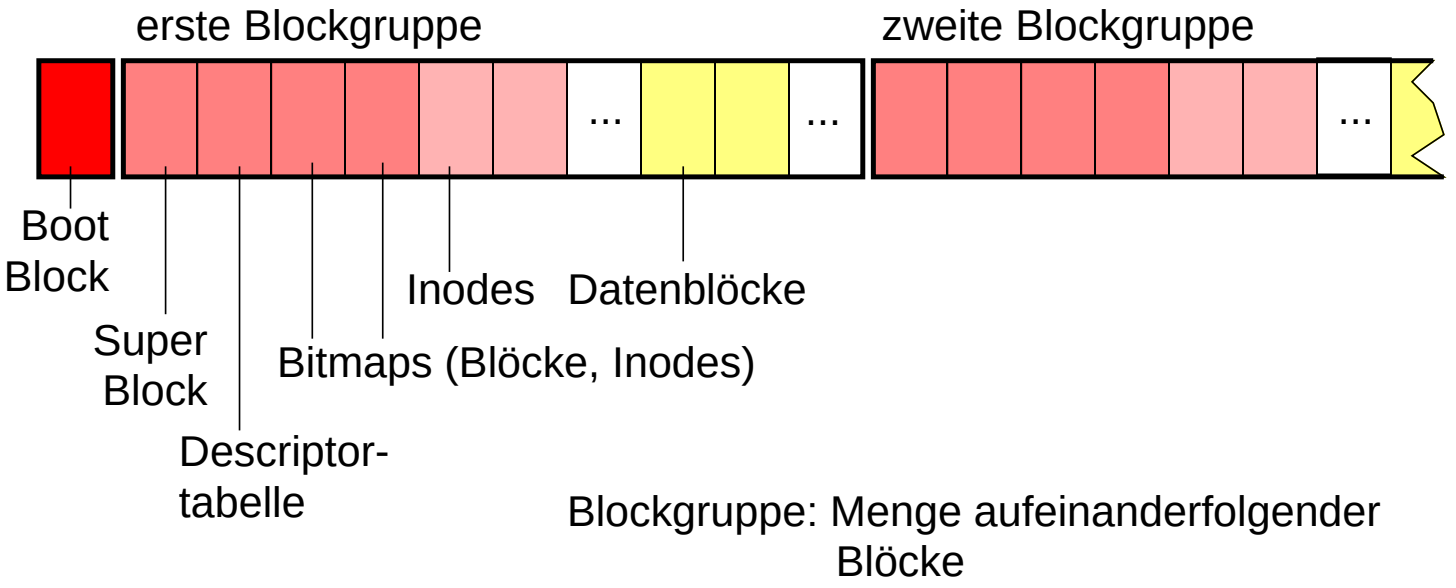


  - Kopie des Super Blocks in jeder Zylindergruppe
  - freie Inodes u. freie Datenblöcke werden im *Cylinder Group Block* gehalten
  - eine Datei wird möglichst innerhalb einer Zylindergruppe gespeichert

- ★ Vorteil: kürzere Positionierungszeiten
  - Optimierung der Zugriffszeiten. in den 1980er Jahren entwickelt

Linux EXT2/3/4

- EXT2: Blockorganisation



  - Ähnliches Layout wie BSD FFS
  - Blockgruppen unabhängig von Zylindern
- EXT3: Erweiterung als Journaling-File-System (siehe Abschnitt 7.2)
- EXT4: Einführung von Extents (siehe NTFS) und viele neue Details

Beispiel: Windows NTFS

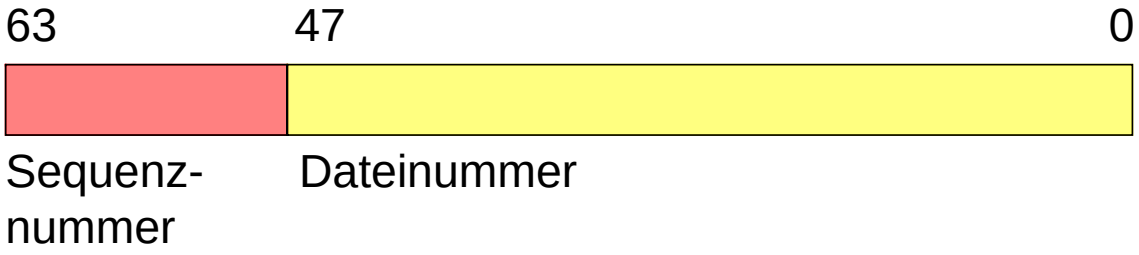
- Dateisystem für Windows-Systeme (seit Windows NT 3.1, 1993)
- Datei
  - beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - Rechte verknüpft mit NT-Benutzern und -Gruppen
  - Datei kann automatisch komprimiert oder verschlüsselt gespeichert werden
  - große Dateien bis zu 2^{64} Bytes lang
  - Hard links: mehrere Einträge derselben Datei in verschiedenen Katalogen möglich
- Dateiinhalt: Sammlung von *Streams*
  - *Stream*: einfache, unstrukturierte Folge von Bytes
  - "normaler Inhalt" = unbenannter Stream (default stream)
  - dynamisch erweiterbar
  - Syntax: dateiname:streamname

Dateiverwaltung

- Basiseinheit „Cluster“
  - 512 Bytes bis 4 Kilobytes (beim Formatieren festgelegt)
  - wird auf eine Menge von hintereinanderfolgenden Blöcken abgebildet
  - logische Cluster-Nummer als Adresse (LCN)
- Basiseinheit „Strom“
  - jede Datei kann mehrere (Daten-)Ströme speichern
  - einer der Ströme wird für die eigentlichen Daten verwendet
  - Dateiname, MS-DOS Dateiname, Zugriffsrechte, Attribute und Zeitstempel werden jeweils in eigenen Datenströmen gespeichert
    (leichte Erweiterbarkeit des Systems)

Dateiverwaltung (2)

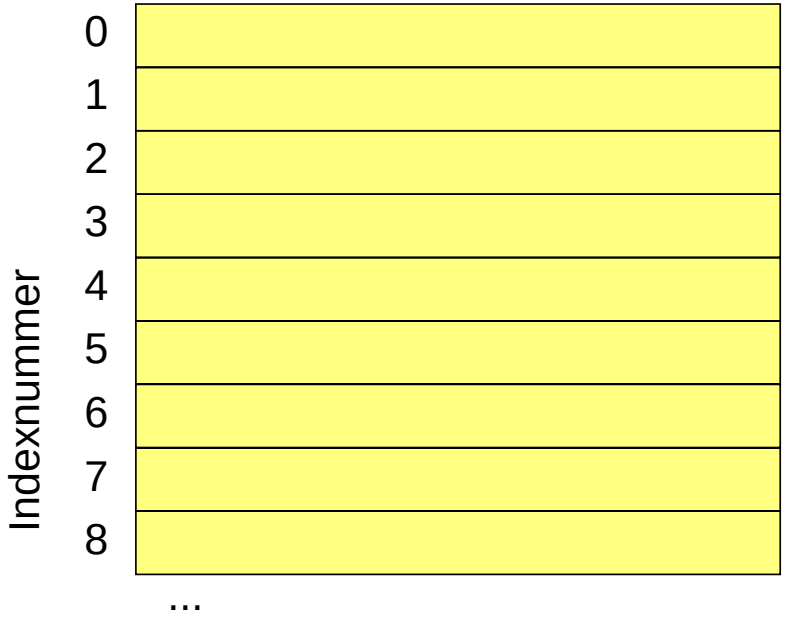
- *File-Reference*
  - Bezeichnet eindeutig eine Datei oder einen Katalog



  - Dateinummer ist Index in eine globale Tabelle *(MFT: Master File Table)*
  - Sequenznummer wird hochgezählt, für jede neue Datei mit gleicher Dateinummer

Master-File-Table

- Rückgrat des gesamten Systems
  - große Tabelle mit gleich langen Elementen
    (1KB, 2KB oder 4KB groß, je nach Clustergröße)
  - kann dynamisch erweitert werden

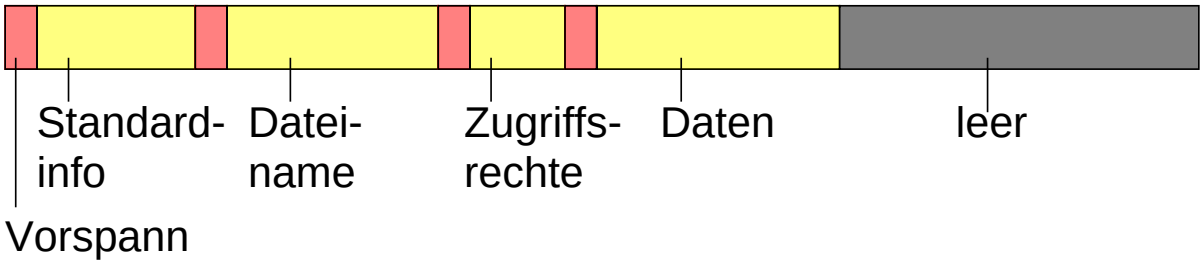


entsprechender Eintrag für eine *File-Reference*
enthält Informationen über bzw.
die *Streams* der Datei

  - Index in die Tabelle ist Teil der *File-Reference*

Master-File-Table (2)

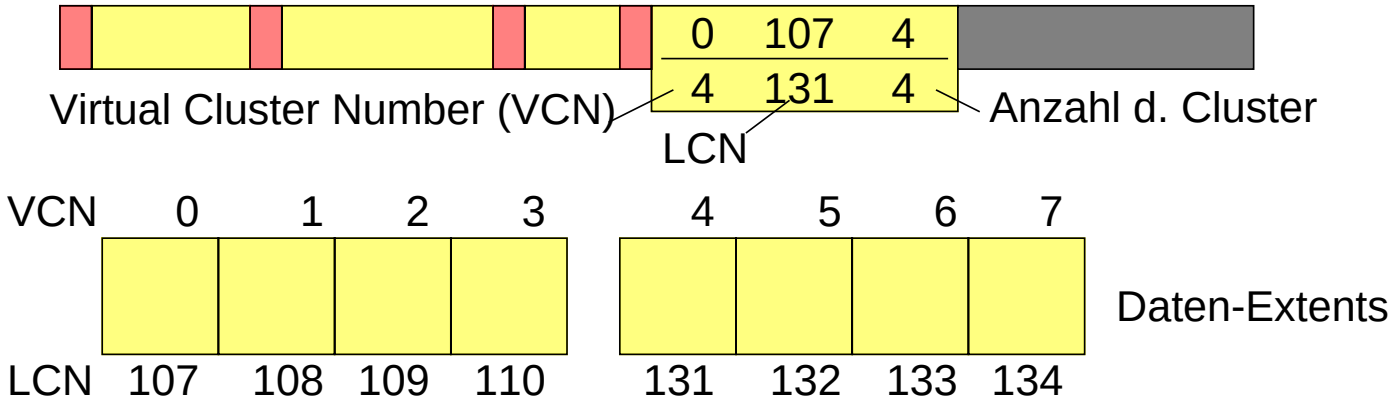
- Eintrag für eine kurze Datei



- Streams
  - Standard-Information (immer in der MFT)
    - enthält Länge, Standard-Attribute, Zeitstempel, Anzahl der Hard links, Sequenznummer der gültigen File-Reference
  - Dateiname (immer in der MFT)
    - kann mehrfach vorkommen (Hard links)
  - Zugriffsrechte *(Security Descriptor)*
  - Eigentliche Daten

Master-File-Table (3)

- Eintrag für eine längere Datei



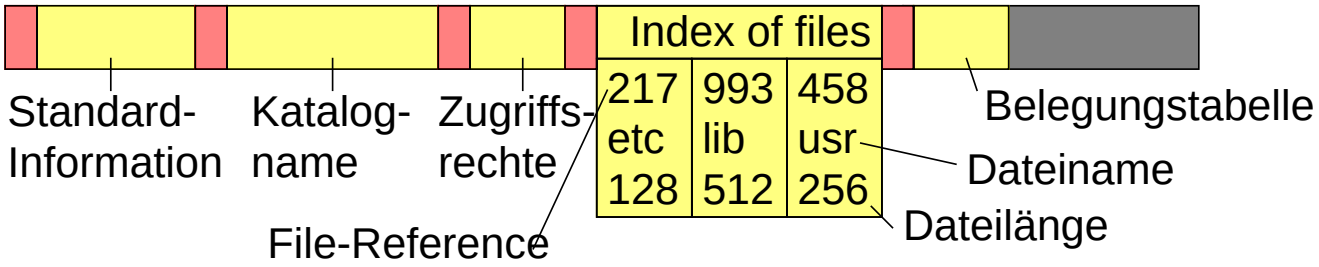
- ***Extents*** werden außerhalb der MFT in aufeinanderfolgenden Clustern gespeichert
- Lokalisierungsinformationen werden in einem eigenen Stream gespeichert

Master-File-Table (4)

- Mögliche weitere Streams *(Attributes)*
  - Index
    - Index über einen Attributschlüssel (z.B. Dateinamen) implementiert Katalog
  - Indexbelegungstabelle
    - Belegung der Struktur eines Index
  - Attributliste (immer in der MFT)
    - wird benötigt, falls nicht alle Streams in einen MFT Eintrag passen
    - referenzieren weitere MFT Einträge und deren Inhalt
  - Streams mit beliebigen Daten
    - wird gerne zum Verstecken von Viren genutzt, da viele Standard-Werkzeuge von Windows nicht auf die Bearbeitung mehrerer Streams eingestellt sind (arbeiten nur mit dem unbenannten Stream)

Master File Table (5)

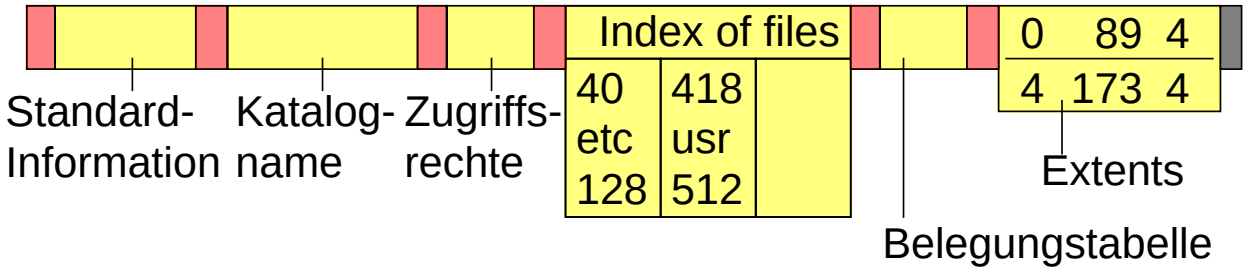
- Eintrag für einen kurzen Katalog



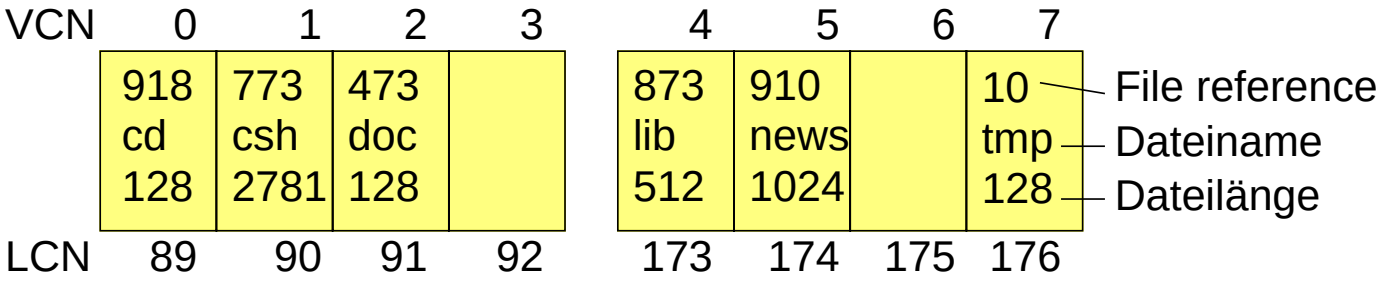
- Dateien des Katalogs werden mit File-References benannt
- Name und Standard-Attribute (z.B. Länge) der im Katalog enthaltenen Dateien und Kataloge werden auch im Index gespeichert
  (doppelter Aufwand beim Update; schnellerer Zugriff beim Kataloglisten)

Master File Table (6)

- Eintrag für einen längeren Katalog



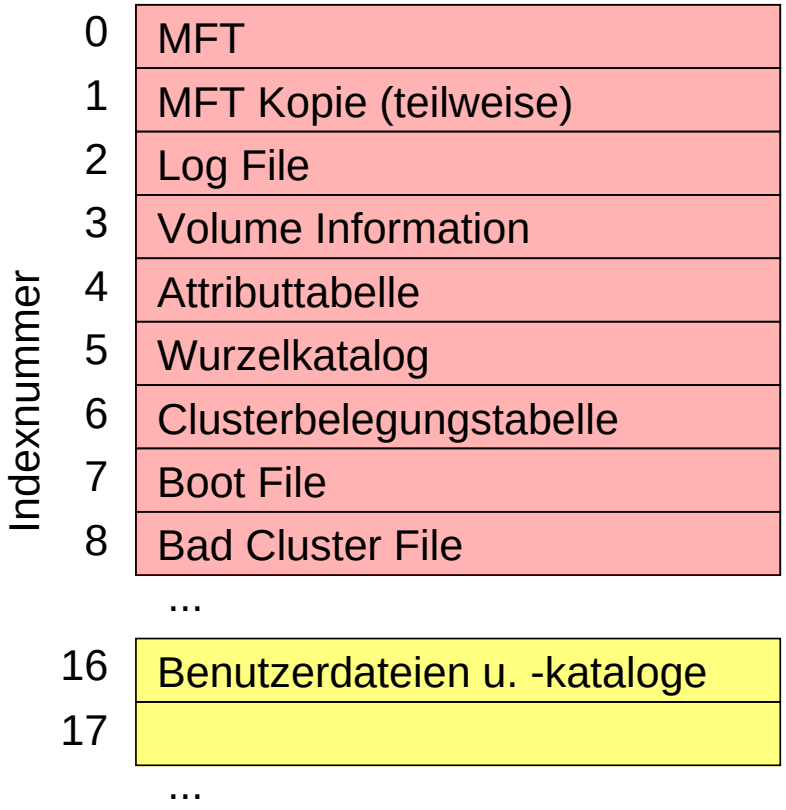
Daten-Extents



- Speicherung als B^+-Baum (sortiert, schneller Zugriff)
- in einen Cluster passen zwischen 3 und 15 Dateien (im Bild nur eine)

Metadaten

- Alle Metadaten werden in Dateien gehalten



Feste Dateien in der MFT

Metadaten (2)

- Bedeutung der Metadateien
  - MFT und MFT Kopie: MFT wird selbst als Datei gehalten
    (d.h. Cluster der MFT stehen im Eintrag 0)
    MFT Kopie enthält die ersten 16 Einträge der MFT (Fehlertoleranz)
  - Log File: enthält protokollierte Änderungen am Dateisystem
  - Volume Information: Name, Größe und ähnliche Attribute des Volumes
  - Attributtabelle: definiert mögliche Ströme in den Einträgen
  - Wurzelkatalog
  - Clusterbelegungstabelle: Bitmap für jeden Cluster des Volumes
  - Boot File: enthält initiales Programm zum Laden, sowie ersten Cluster der MFT
  - Bad Cluster File: enthält alle nicht lesbaren Cluster der Platte
    NTFS markiert automatisch alle schlechten Cluster und versucht die Daten in einen anderen Cluster zu retten

Fehlererholung

- NTFS ist ein Journaling-File-System
  - Änderungen an der MFT und an Dateien werden protokolliert.
  - Konsistenz der Daten und Metadaten kann nach einem Systemausfall durch Abgleich des Protokolls mit den Daten wieder hergestellt werden.
- Nachteile
  - etwas ineffizienter
  - nur für Volumes >400 MB geeignet

Dateisysteme mit Fehlererholung

- Metadaten und aktuell genutzte Datenblöcke geöffneter Dateien werden im Hauptspeicher gehalten (Dateisystem-Cache)
  - effizienter Zugriff
  - Konsistenz zwischen Cache und Platte muss regelmäßig hergestellt werden
    - ➤ synchrone Änderungen: Operation kehrt erst zurück, wenn Änderungen auf der Platte gespeichert wurden
    - ➤ asynchrone Änderungen: Änderungen erfolgen nur im Cache, Operation kehrt danach sofort zurück,
      Synchronisation mit der Platte erfolgt später
- Mögliche Fehlerursachen
  - Stromausfall (oder dummer Benutzer schaltet einfach Rechner aus)
  - Systemabsturz

Konsistenzprobleme

- Fehlerursachen & Auswirkungen auf das Dateisystem
  - Cache-Inhalte und aktuelle E/A-Operationen gehen verloren
  - inkonsistente Metadaten
    z. B. Katalogeintrag fehlt zur Datei oder umgekehrt
    z. B. Block ist benutzt aber nicht als belegt markiert
- ★ Reparaturprogramme
  - Programme wie **`chkdsk`**, **`scandisk`** oder **`fsck`** können inkonsistente Metadaten reparieren
- ▲ Datenverluste bei Reparatur möglich
- ▲ Große Platten bedeuten lange Laufzeiten der Reparaturprogramme

Journaling-File-Systems

- Zusätzlich zum Schreiben der Daten und Meta-Daten (z. B. Inodes) wird ein Protokoll der Änderungen geführt
  - Grundidee: Log-based Recovery bei Datenbanken
  - alle Änderungen treten als Teil von Transaktionen auf.
  - Beispiele für Transaktionen:
    - Erzeugen, Löschen, Erweitern, Verkürzen von Dateien
    - Dateiattribute verändern
    - Datei umbenennen
  - Protokollieren aller Änderungen am Dateisystem zusätzlich in einer Protokolldatei *(Log File)*
  - beim Bootvorgang wird Protokolldatei mit den aktuellen Änderungen abgeglichen und damit werden Inkonsistenzen vermieden.

Journaling-File-Systems (2)

- Protokollierung
  - für jeden Einzelvorgang einer Transaktion wird zunächst ein Logeintrag erzeugt und
  - danach die Änderung am Dateisystem vorgenommen
  - dabei gilt:
    - ➤ der Logeintrag wird immer vor der eigentlichen Änderung auf Platte geschrieben
    - ➤ wurde etwas auf Platte geändert, steht damit sicher auch der Protokolleintrag dazu auf der Platte
- Fehlererholung
  - Beim Bootvorgang wird überprüft, ob die protokollierten Änderungen vorhanden sind:
    - ➤ Transaktion kann wiederholt bzw. abgeschlossen werden *(Redo)* falls alle Logeinträge vorhanden
    - ➤ angefangene, aber nicht beendete Transaktionen werden rückgängig gemacht *(Undo).*

Journaling-File-Systems (3)

- Beispiel: Löschen einer Datei im NTFS
  - Vorgänge der Transaktion
    - ➤ Beginn der Transaktion
    - ➤ Freigeben der Extents durch Löschen der entsprechenden Bits in der Belegungstabelle (gesetzte Bits kennzeichnen belegten Cluster)
    - ➤ Freigeben des MFT-Eintrags der Datei
    - ➤ Löschen des Katalogeintrags der Datei (evtl. Freigeben eines Extents aus dem Index)
    - ➤ Ende der Transaktion
  - Alle Vorgänge werden unter der File-Reference im Log-File protokolliert, danach jeweils durchgeführt.
    - ➤ Protokolleinträge enthalten Informationen zum *Redo* und zum *Undo*

Journaling-File-Systems (4)

- Log vollständig (Ende der Transaktion wurde protokolliert und steht auf Platte):
  - ➤ *Redo* der Transaktion:
    alle Operationen werden wiederholt, falls nötig
- Log unvollständig (Ende der Transaktion steht nicht auf Platte):
  - ➤ *Undo* der Transaktion:
    in umgekehrter Reihenfolge werden alle Operation rückgängig gemacht

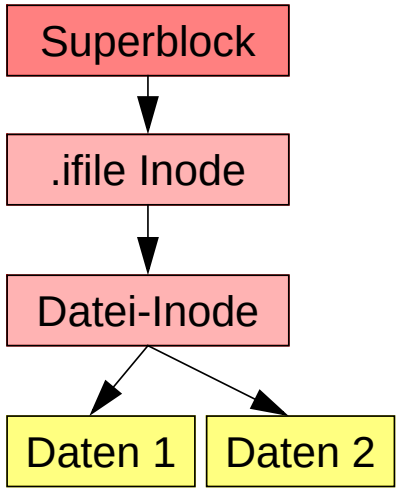
- Checkpoints
  - Log-File kann nicht beliebig groß werden
  - gelegentlich wird für einen konsistenten Zustand auf Platte gesorgt *(Checkpoint)* und dieser Zustand protokolliert (alle Protokolleinträge von vorher können gelöscht werden)

Journaling-File-Systems (5)

- ★ Ergebnis
  - eine Transaktion ist entweder vollständig durchgeführt oder gar nicht
  - Nutzer können ebenfalls Transaktionen über mehrere Dateizugriffe definieren, wenn diese ebenfalls im Log erfasst werden
  - keine inkonsistenten Metadaten möglich
  - Hochfahren eines abgestürzten Systems benötigt nur den relativ kurzen Durchgang durch das Log-File.
    - ➤ Alternative **`chkdsk`** benötigt viel Zeit bei großen Platten
- ▲ Nachteile
  - etwas langsamer, da zusätzlich Log-File-Einträge geschrieben werden müssen
- ■ Beispiele: NTFS, EXT3, EXT4, ReiserFS

Copy-on-Write- / Log-Structured-File-Systems

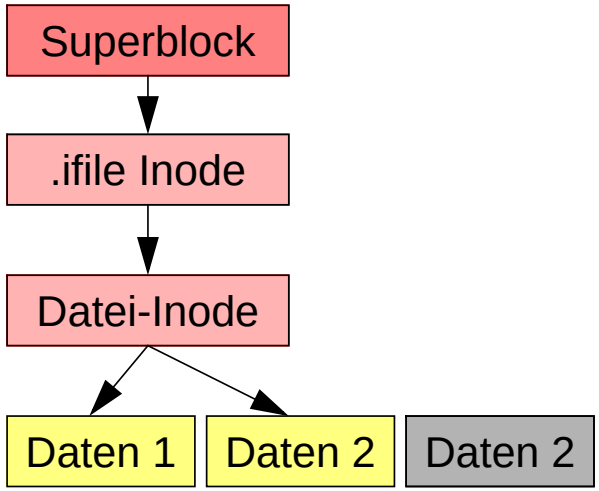
- Alternatives Konzept zur Realisierung von atomaren Änderungen
- Alle Änderungen im Dateisystem erfolgen auf Kopien
  - Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



  - Beispiel LinLogFS: Superblock einziger nicht ersetzter Block

Copy-on-Write- / Log-Structured-File-Systems

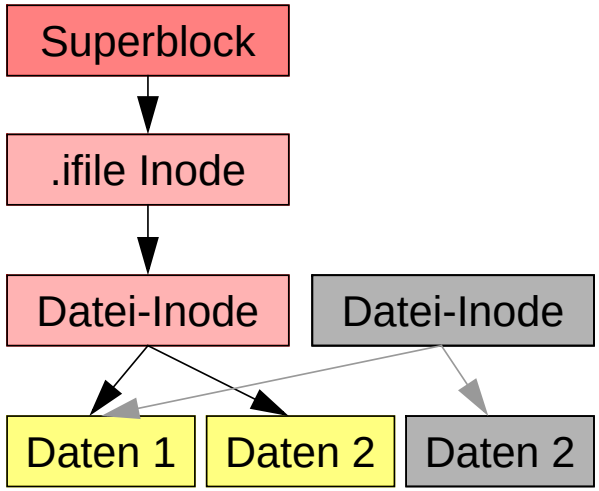
- Alternatives Konzept zur Realisierung von atomaren Änderungen
- Alle Änderungen im Dateisystem erfolgen auf Kopien
  - Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



  - Beispiel LinLogFS: Superblock einziger nicht ersetzter Block

Copy-on-Write- / Log-Structured-File-Systems

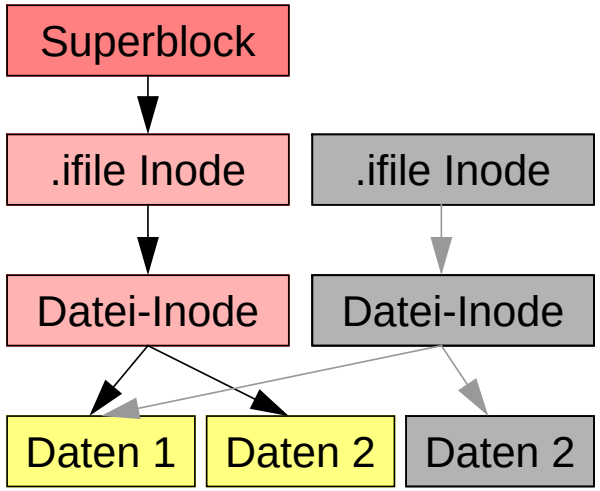
- Alternatives Konzept zur Realisierung von atomaren Änderungen
- Alle Änderungen im Dateisystem erfolgen auf Kopien
  - Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



  - Beispiel LinLogFS: Superblock einziger nicht ersetzter Block

Copy-on-Write- / Log-Structured-File-Systems

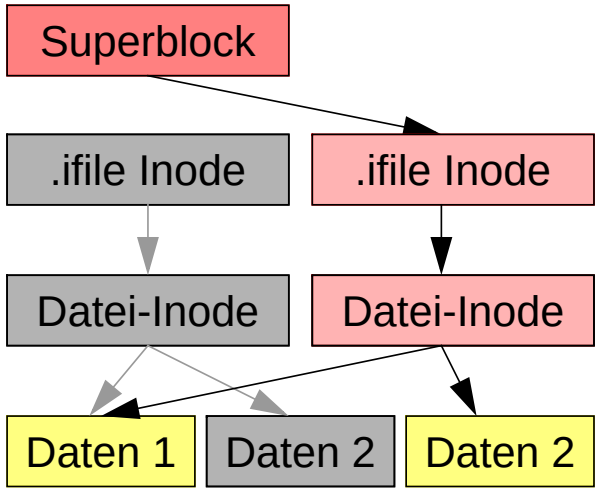
- Alternatives Konzept zur Realisierung von atomaren Änderungen
- Alle Änderungen im Dateisystem erfolgen auf Kopien
  - Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



  - Beispiel LinLogFS: Superblock einziger nicht ersetzter Block

Copy-on-Write- / Log-Structured-File-Systems

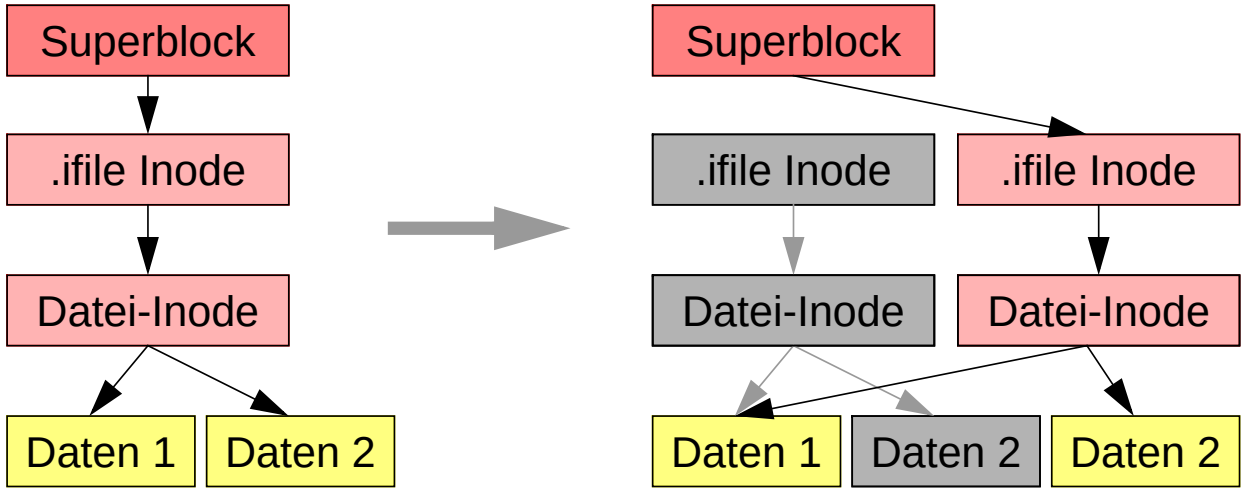
- Alternatives Konzept zur Realisierung von atomaren Änderungen
- Alle Änderungen im Dateisystem erfolgen auf Kopien
  - Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



  - Beispiel LinLogFS: Superblock einziger nicht ersetzter Block

Copy-on-Write- / Log-Structured-File-Systems

- Alternatives Konzept zur Realisierung von atomaren Änderungen
- Alle Änderungen im Dateisystem erfolgen auf Kopien
  - Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



  - Beispiel LinLogFS: Superblock einziger statischer Block (Anker im System)

Copy-on-Write- / Log-Structured-File-Systems (2)

★ Vorteile

- Gute Schreibeffizienz - vor allem bei Log-Structured-File-Systems
- Datenkonsistenz bei Systemausfällen
  - ein atomare Änderung macht alle zusammengehörigen Änderungen sichtbar
- Schnappschüsse / Checkpoints einfach realisierbar

▲ Nachteile

- Erzeugt starke Fragmentierung, die sich beim Lesen auswirken kann
  ➥ Performanz nur akzeptabel, wenn Lesen primär aus Cache erfolgen kann oder Positionierzeiten keine Rolle spielen (SSD)

■ Unterschied zwischen Copy-on-Write- und Log-Structured-File-Systems

- Log-Structured-File-Systems schreiben kontinuierlich an das Ende des belegten Plattenbereichs und geben vorne die Blöcke wieder frei (kontinuierlicher Log)
- Beispiele: Log-Structured: LinLogFS, BSD LFS
  Copy-on-Write: ZFS, Btrfs (Oracle)

Fehlerhafte Plattenblöcke

- Blöcke, die beim Lesen Fehlermeldungen erzeugen
  - z.B. Prüfsummenfehler
- Hardwarelösung
  - Platte und Plattencontroller bemerken selbst fehlerhafte Blöcke und maskieren diese aus
  - Zugriff auf den Block wird vom Controller automatisch auf einen „gesunden“ Block umgeleitet
- Softwarelösung
  - File-System bemerkt fehlerhafte Blöcke und markiert diese auch als belegt

Datensicherung

- Schutz vor dem Totalausfall von Platten
  - z. B. durch Head-Crash oder andere Fehler

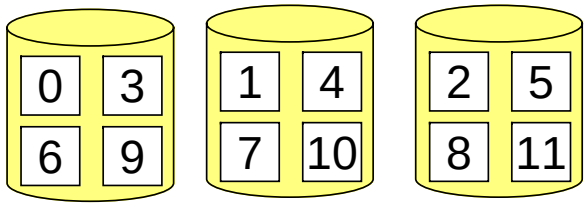
Sichern der Daten auf Tertiärspeicher

- Bänder, Bandroboter mit vorgelagertem Platten-Cache
- WORM-Speicherplatten *(Write Once Read Many)*

- Sichern großer Datenbestände
  - Total-Backups benötigen lange Zeit
  - Inkrementelle Backups sichern nur Änderungen ab einem bestimmten Zeitpunkt
  - Mischen von Total-Backups mit inkrementellen Backups

Einsatz mehrerer (redundanter) Platten

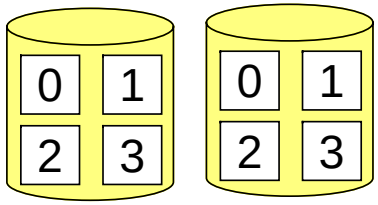
- Gestreifte Platten (*Striping*; RAID 0)
  - Daten werden über mehrere Platten gespeichert



  - Datentransfers sind nun schneller, da mehrere Platten gleichzeitig angesprochen werden können
- Nachteil
  - keinerlei Datensicherung: Ausfall einer Platte lässt Gesamtsystem ausfallen

Einsatz mehrerer redundanter Platten (2)

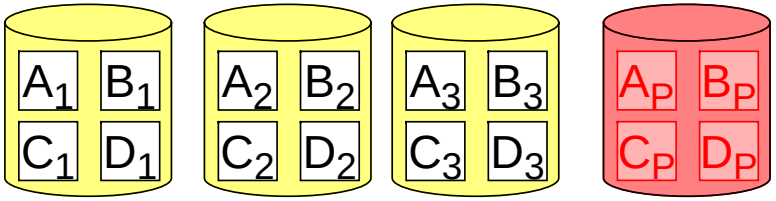
- Gespiegelte Platten (*Mirroring*; RAID 1)
  - Daten werden auf zwei Platten gleichzeitig gespeichert



  - Implementierung durch Software (File-System, Plattentreiber) oder Hardware (spez. Controller)
  - eine Platte kann ausfallen
  - schnelleres Lesen (da zwei Platten unabhängig voneinander beauftragt werden können)
- Nachteil
  - doppelter Speicherbedarf
- wenig langsameres Schreiben durch Warten auf zwei Plattentransfers
- Verknüpfung von RAID 0 und 1 möglich (RAID 0+1)

Einsatz mehrerer redundanter Platten (3)

- Paritätsplatte (RAID 4)
  - Daten werden über mehrere Platten gespeichert, eine Platte enthält Parität



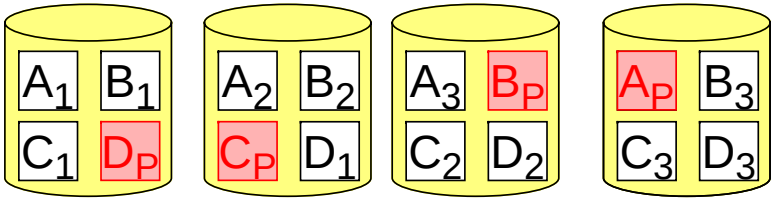
  - Paritätsblock enthält byteweise XOR-Verknüpfungen von den zugehörigen Blöcken aus den anderen Streifen
  - eine Platte kann ausfallen
  - schnelles Lesen
  - prinzipiell beliebige Plattenanzahl (ab drei)

Einsatz mehrerer redundanter Platten (4)

- Nachteil von RAID 4
  - jeder Schreibvorgang erfordert auch das Schreiben des Paritätsblocks
  - Erzeugung des Paritätsblocks durch Speichern des vorherigen Blockinhalts möglich: $P_{neu} = P_{alt} \oplus B_{alt} \oplus B_{neu}$ (P=Parity, B=Block)
  - Schreiben eines kompletten Streifens benötigt nur einmaliges Schreiben des Paritätsblocks
  - Paritätsplatte ist hoch belastet

Einsatz mehrerer redundanter Platten (5)

- Verstreuter Paritätsblock (RAID 5)
  - Paritätsblock wird über alle Platten verstreut



  - zusätzliche Belastung durch Schreiben des Paritätsblocks wird auf alle Platten verteilt
  - heute gängigstes Verfahren redundanter Platten
  - Vor- und Nachteile sonst wie RAID 4
- Doppelte Paritätsblöcke (RAID 6)
  - ähnlich zu RAID 5, aber zwei Paritätsblöcke (verkraftet damit den Ausfall von bis zu zwei Festplatten)
  - wichtig bei sehr großen, intensiv genutzten RAID-Systemen, wenn die Wiederherstellung der Paritätsinformation nach einem Plattenausfall lange dauern kann