1) **Betriebsarten**

Stapelverarbeitung (Lochkarten)

- Rechnerfamilien für wissenschaftliche und kommerzielle Berechnungen
- günstig (ICs statt Röhren)
- Software sollte auf diversen Rechnern laufen

Mehrprogrammbetrieb (Mutliprogramming, Multitasking)

- Gleichzeitiges Bereithalten mehrerer Jobs im Hauptspeicher (Partitionierung)
- Auf anderen Job umschalten statt auf E/A zu warten
- Timesharing
    - Jeder Benutzer hat Zugang zum System über sein Terminal

2) **Betriebssystemstrukturen**

Kernaufruf

- Anwendungsprogramm springt über TRAP in den Kern und führt den Code selbst aus.
- BS Code bestimmt die Nummer des angeforderten Dienstes.
- BS Code lokalisiert Prozedur-Code für Systemaufruf und ruft sie auf.
- Kontrolle wird an das Anwendungsprogramm zurückgegeben.
- Wichtig: Kern selbst ist passiv (Menge von Datenstrukturen und Prozeduren)

Betriebsmodi

- meisten 2 Modi (privilegiert, nicht-privilegiert), bei x86 4 Modi.
- Hardwaresicht (Privilegierter Modus)
    - Sperren von Unterbrechungen, Zugriff auf Speicherverwaltung-Hardware
    - Exceptions (Interrupts,TRAPs und Faults (z.B. division by 0)) schalten in den privilegierten Modus
- Betriebssystemsicht (Benutzungsmodus = nicht-privilegiert)
    - Beschränkter Zugriff auf Betriebsmittel
    - Unberechtigter Zugriff auf Betriebsmittel lösen Faults aus
    - Unerlaubte Operationen lösen Faults aus
    - Systemcall = expliziter TRAP Befehl
- Betriebssystemsicht (privilegiert)
    - Uneingeschränkter Zugriff auf alle Betriebsmittel
    - Faults und Exceptions führen zum Absturz

Monolithische Systeme (Windows, Unix, ...)

- prozedurorientiert
    - Kern ist passiv und der Code besteht aus einer Menge von Prozeduren
    - Struktur: Hauptprogramme → Dienstprozeduren → Hilfsprozeduren
- Nachteil: Viel Code → viele Fehler, nicht alle Anwedungen benötigen alle Dienste , Art und Anzahl der Dienste vom Kern vorgegeben

Client/Server-Strukturen (Mikrokerne)

- Ansatz: Nur Dienste die im Kernmodus laufen müssen, dürfen in diesem laufen
- Dateisystem, Netzwerkprotokolle, Speicherverwaltung müssen nicht im Kern sein , „Server“-Prozesse (ohne besondere Privilegien) bieten diese Dienste an
- Kern bietet nur Dienste zur Kommunikation zwischen Klienten (Anwendungen) und Servern untereinander an
- Dienste werden durch Nachtrichten per IPC: Interprozesskommunikation von Servern angefordert (send & receive)
- Server liefern Dienste auch mit IPC-Nachrichten (reply & wait)

-Vorteile: Isolation der Systemteile, Erweiterbarkeit, Nachrichtenbasiert
- Policy & Mechanism
    - Beispiel Speicherverwaltung
        - Strategie (policy): Zuteilung von Speicher an Prozesse
        - Mechanismus (mechanism): Konfiguration der Hardware
- µKern SOLLTE klein und wenig komplex sein

-Single Server: Monolithisches BS in Server umwandeln
- Mehrere BS in einem Rechner, große Trusted Code Base, schlechte Performance

Virtualisierung
- Virtuelle Maschinen (Beispiel VM/370)
    - Trennen der Funktionen „Mehrprogrammbetrieb“ und „erweiterte Maschine“
    - Virtualisierung durch Hypervisor
    - virtuelle Maschinene als identische Kopien der Hardware
    -in jeder virtuellen Maschine: übliches Betriebssystem
- Virtualisierbarkeit (Anforderung: Identisches Verhalten der VM)
    - Emulation: Nachbild der HW ins SW (ineffizient!) [Bochs, JWVM]
    - Virtualisierung: die meisten Befehle werden von der realen Hardware ausgeführt,der Rest emuliert(schnell, Architekturabhängig)[QEMU,VMWare]
    - Paravirtualisierung (Falls nicht virtualisierbar): Priviligierte Befehle des Gast-BS durch „Hypercalls“ (= Aufrufe in den Hypervisor) ersetzen. Schnell oder schneller als Virtualisierung, aber Gast-BS muss angepasst werden. [Xen, KVM, Hyper-V]

3) **Prozesse und Threads**

Prozessmodell
- Prozess: ein in sich in Ausführung befindliches Programm inkl.Stack, Register,Pc
    - Menge von (virtuellen) Adressen, von Prozess zugreifbar
    - Programm und Daten in Addressraum sichtbar
- Verhältnis Prozessor - Prozessor
    - Prozess besitzt konzeptionel eigenen virtuellen Prozessor
    - Reale(r) Prozessor(en) werden zwischen virtuellen Prozessoren umgeschaltet (Mehrprogrammbetrieb)
    - Umschaltungseinheit heißt Scheduler oder Dispatcher
    - Umschaltvorgang heißt Prozesswechsel oder Kontextwechsel
- Prozesserzeugung
    1) feste Menge von Prozessen werden beim Systemstart erzeugt
    einfache, meist eingebettete System [Motorsteuerung, Videorekorder]
    einfache Verwaltung, deterministisches Zeitverhalten, unflexibel

    2) dynamisch (es können im Laufe der Zeit neue Prozesse erzeugt werden)
    impliziert die Bereitstellung geigneter Systemaufrufen durch BS
- Prozessende
    1) freiwillig: Prozess ist fertig (egal ob erfolgreich oder nicht)
    2) unfreiwillig: Prozess WIRD beendet (Bsp: Division 0, Segmentation Fault)
- Prozesshierarchie (Unix ja, Windows nein [Prozesse gleichwertig])

-Prozesszustände (aktiv, bereit, schlafend/blockiert) selten auch initiiert, terminiert

Implementierung
- PCB (Process Control Block)
  - Prozessverwaltung: Register, Id, Pc, StackPtr, Flags, Signal, Parent, Zustand
  - Speicherverwaltung: zeiger auf .text .data .bss, real und effektiv UID & GID
  - Dateisystem: effektive UID & GID, Flags, Wurzel- & aktuelles Verzeichnis
  - Zeiger zur Verkettung des PCB in (verschiedenen) Warteschlangen
- Scheduler-Aktivierung
  - kooperatives Multitasking: Problem MUSS Kontrolle an BS abgeben
  - preemptiv: Code wird unterbrochen bei z.B. Ablauf eines Timers, Scheduler wird aufgerufen
- Unterbrechungsbehandlung
  - Interrupt-Handler: Interrupt-Vektor-Tabelle (IVT) enthält Interrupts mit IDs
  - Ablauf: Pc (u.a.) wird durch HW auf dem Stack abgelegt
    HW lädt Pc-Inhalt aus Unterbrechungsvektor
    Assembly-Routine rettet Registerinhalte
    Assembly-Routine bereitet den neuen Stack vor
    C-Prozedur markiert den unterbrochenen Prozess als bereit
    Scheduler bestimmt den nächsten auszuführenden Prozess
    C-Prozedur gibt Kontrolle an die Assembly-Routine zurück
    Assembly-Routine startet den ausgewählten Prozess
- Interrupts aus Sicht des Prozesses
  - IRT: Interrupt Response Time
  - PDLT Process Dispatch Latency Time
  - SWT Process Switch Time

Threads (Leichtgewichtsprozesse für billige Nebenläufigkeit im Prozessadressraum)
- Idee einer „parallel ausgeführten Programmfunktion“
- eigener Prozessor-Context (Registerinhalte usw.)
- eigener Stack (i.d.R. 2, getrennt für unser und kernel mode)
- eigener kleiner privater Datenbereich (Thread Local Storage)
- Threads nutzen alles Betriebsmittel, Programm- & Adressraum des Prozesses
- WICHTIG: Bein 1-Prozessorsystemen kein Performancegewinn
- Kooperationsformen: Verteiler-/Arbeitermodell, Teammodell, Fließbandmodell
- Implementierung
  - Thread-Bibliothek (User level threads)
    - Threadfunktionen/Kontextwechsel auf Applikationsebene
    - einfache Implementierung, keine Nutzung von MehrprozessorArch
  - Im BS-Kern (Kernel level threads)
    - Threads als Einheiten denen Prozessoren zugeordnet sind
    - Nutzung von Mehrprozessor Architekturen,Kernelunterstüzung nötig

4) **Scheduling (Priorität- oder Zeitscheiben-basiert)**

Begriffe
-Bedienzeit: Zeitdauer für reine Bearbeitung des Auftrags
-Antwortzeit: Zeitdauer vom Eintreffen bis zur Fertigstellung des Auftrags
Bei Dialogaufträgen Zeitdauer von Benutzereingabe bis Ausgabe
Bei Stapelaufträgen auch Verweilzeit genannt
-Wartezeit: Antwortzeit - Bedienzeit
-Durchsatz: Anzahl erledigter Aufträge pro Zeiteinheit
-Auslastung: Anteil der Zeit im Zustand „belegt“

-Fairness: „Gerechte“ Behandlung aller Aufträge

Moderne Anforderungen

-Scheduling wird von Applikationsebene gesteuert

Non-Preemptive Scheduling (Annahme: Bekannte Bedienzeiten)

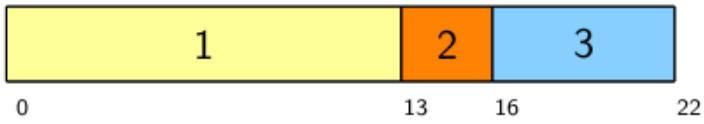
-FCFS (first come first served): Ready-Queue als FIFO Liste

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit
1	13
2	3
3	6

Alle Aufträge seien zur Zeit Null bekannt

Resultierender Schedule:



Prozess	Wartezeit	Antwortzeit
1	0	13
2	13	16
3	13+3=16	22

Durchschnittliche Wartezeit:
$(13 + 16)/3 = 29/3$

Im Falle der Ausführungsfolge 3, 2, 1 hätte sich ergeben:
Durchschnittliche Wartezeit: $(6 + 9)/3 = 5$

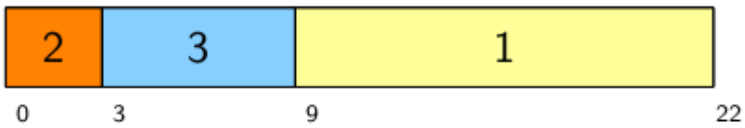
-SJF (Shortest Job First)

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit
1	13
2	3
3	6

Alle Aufträge seien zur Zeit Null bekannt

Resultierender Schedule:



Prozess	Wartezeit	Antwortzeit
1	3+6=9	22
2	0	3
3	3	9

Durchschnittliche Wartezeit:
$(9 + 3)/3 = 4$

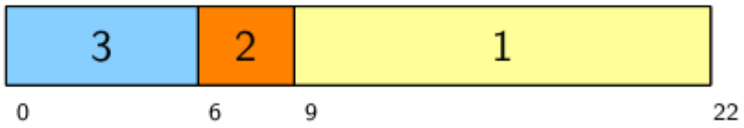
-Prioritäts-Scheduling: Jeder Auftrag hat statische Priorität
höchste Priorität hat Vorrang
Bei gleicher Priorität FCFS

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit	Priorität
1	13	2
2	3	3
3	6	4

Alle Aufträge seien zur Zeit Null bekannt

Resultierender Schedule:



Prozess	Wartezeit	Antwortzeit
1	6+3=9	22
2	6	9
3	0	6

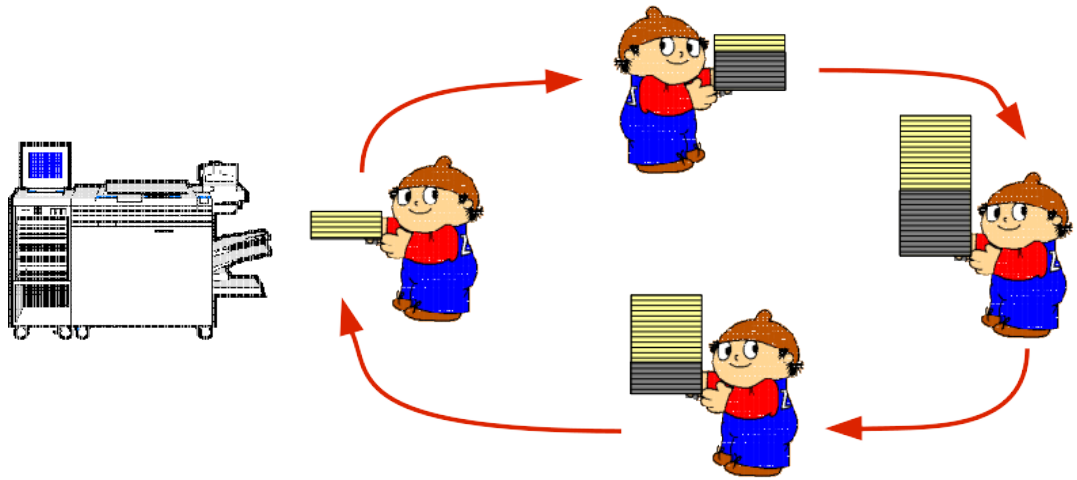
Durchschnittliche Wartezeit[4]:
$(9+6)/3 = 5$

[4] In diesem Beispiel – abhängig von Prioritätsvergabe sind auch alle anderen Ergebnisse möglich

Preemptive Scheduling

Round-Robin-Scheduling (RR)

Hochschule RheinMain



Algorithmus:

- Menge der rechenwilligen Prozesse linear geordnet.
- Jeder rechenwillige Prozess erhält den Prozessor für eine feste Zeitdauer q, die **Zeitscheibe** (*time slice*) oder **Quantum** genannt wird.
- Nach Ablauf des Quantums wird der Prozessor entzogen und dem nächsten zugeordnet (preemptive-resume).
- Tritt vor Ende des Quantums Blockierung oder Prozessende ein, erfolgt der Prozesswechsel sofort.
- Dynamisch eintreffende Aufträge werden z.B. am Ende der Warteschlange eingefügt.

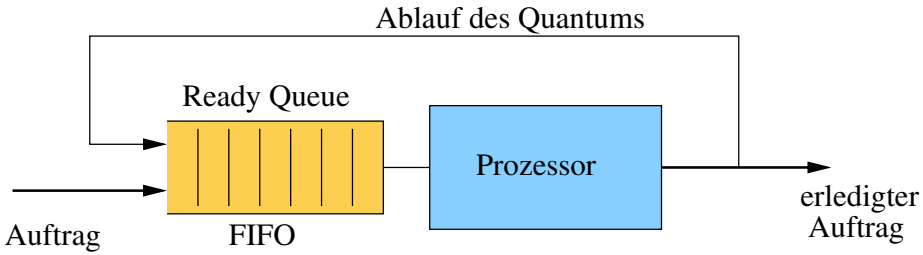
Implementierung:

- Die Zeitscheibe wird durch einen Uhr-Interrupt realisiert.
- Die Ready Queue wird als lineare Liste verwaltet, bei Ende eines Quantums wird der Prozess am Ende der Ready Queue eingefügt.

Round-Robin-Scheduling (RR)

Hochschule RheinMain

Bedienmodell:



Bewertung:

- Round-Robin ist einfach und weit verbreitet.
- Alle Prozesse werden als gleich wichtig angenommen und fair bedient.
- Langläufer benötigen ggf. mehrere „Runden“
- Keine Benachteiligung von Kurzläufern
  (ohne Bedienzeit vorab zu kennen)
- Einziger kritischer Punkt: Wahl der Dauer des Quantums.
  - Quantum zu klein → häufige Prozesswechsel, sinnvolle Prozessornutzung sinkt
  - Quantum zu groß → schlechte Antwortzeiten bei kurzen interaktiven Aufträgen.

Rechenbeispiel

Hochschule RheinMain

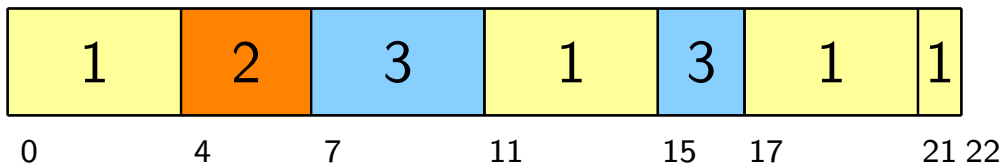
Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit
1	13
2	3
3	6

Alle Aufträge seien zur Zeit Null bekannt

Quantum sei $q = 4$

- Resultierender Schedule:



Prozess	Wartezeit	Antwortzeit
1	3+4+2=9	22
2	4	7
3	4+3+4=11	17

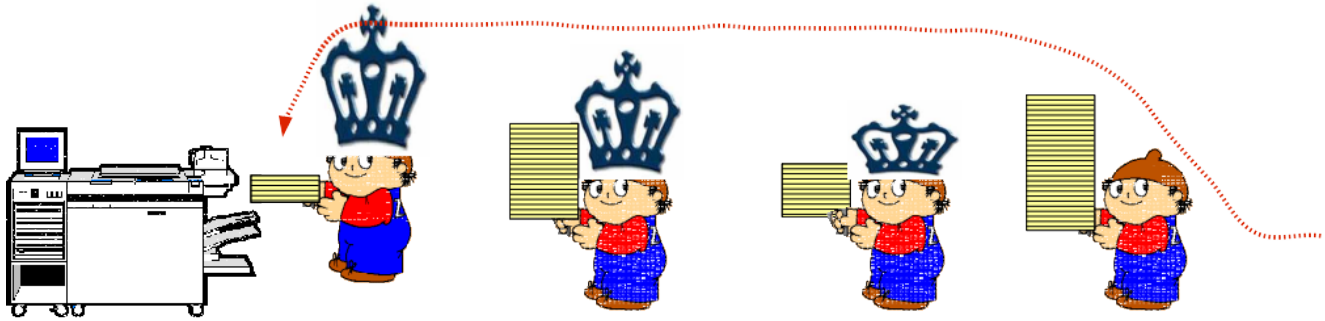
Durchschnittliche Wartezeit:
$(9 + 4 + 11)/3 = 8$

Grenzwertbetrachtung

Grenzwertbetrachtung für Quantum q:

- $q \to \infty$:
  Round-Robin verhält sich wie FCFS.
- $q \to 0$:
  Round-Robin führt zu sogenanntem **processor sharing**:
  jeder der n rechenwilligen Prozesse erfährt $\frac{1}{n}$ der Prozessorleistung.
  (Kontextwechselzeiten als Null angenommen).

Unterbrechendes Prioritäts-Scheduling



Algorithmus:

- Jeder Auftrag besitze eine statische Priorität.
- Prozesse werden gemäß ihrer Priorität in eine Warteschlange eingereiht.
- Von allen rechenwilligen Prozessen wird derjenige mit der höchsten Priorität ausgewählt und bedient.
- Wird ein Prozess höherer Priorität rechenwillig (z.B. nach Beendigung einer Blockierung), so wird der laufende Prozess unterbrochen (*preemption*) und in die Ready Queue eingefügt.

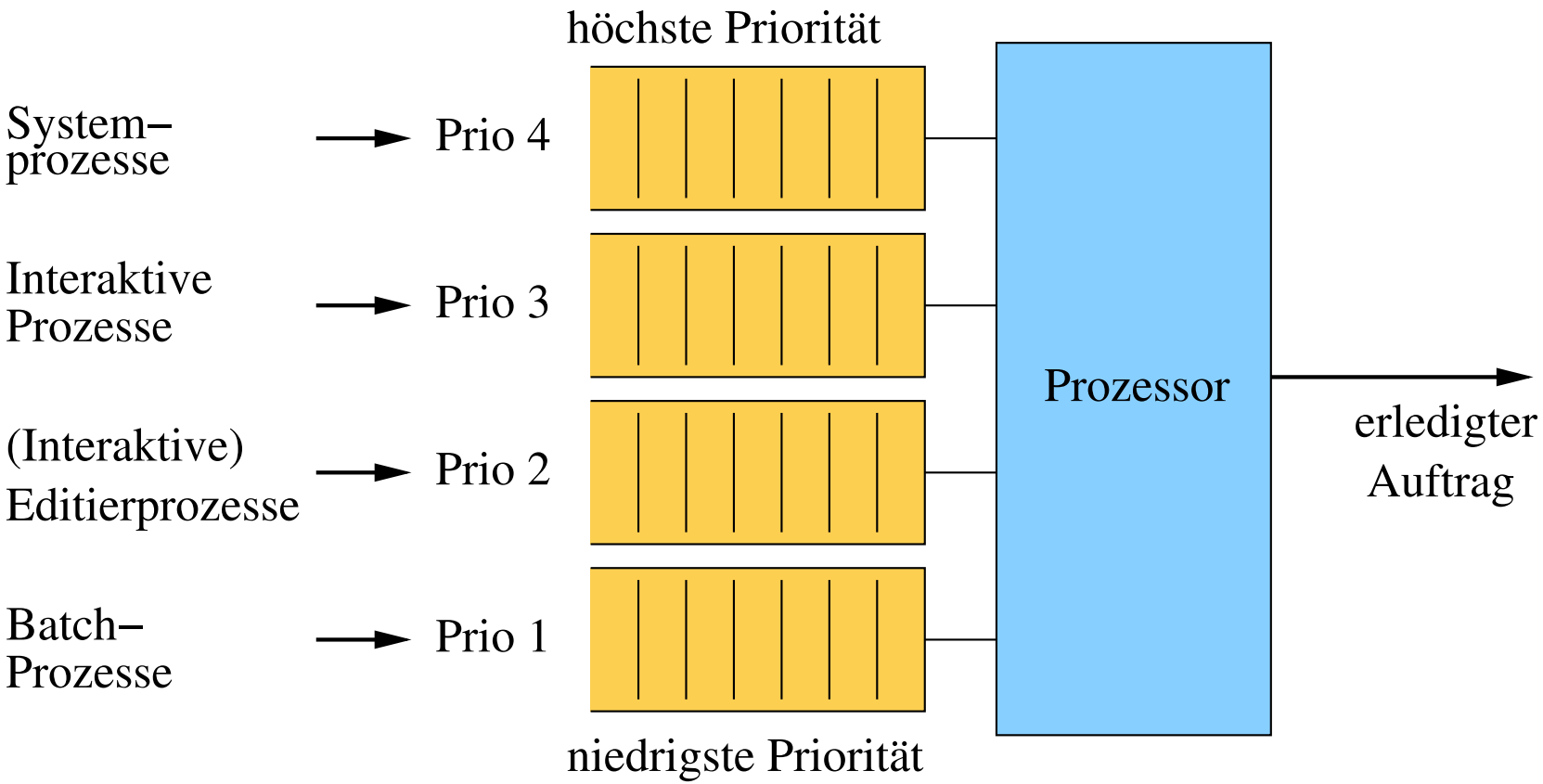
Mehrschlangen-Scheduling

Algorithmus:

- Prozesse werden statisch klassifiziert als einer bestimmten Gruppe zugehörig (z.B. interaktiv, batch).
- Alle rechenwilligen Prozesse einer bestimmten Klasse werden in einer eigenen Ready Queue verwaltet.
- Jede Ready Queue kann ihr eigenes Scheduling-Verfahren haben (z.B. Round-Robin für interaktive Prozesse, FCFS für batch-Prozesse).
- Zwischen den Ready Queues wird i.d.R. unterbrechendes Prioritäts-Scheduling angewendet, d.h.: jede Ready Queue besitzt eine feste Priorität im Verhältnis zu den anderen; wird ein Prozess höherer Priorität rechenwillig, wird der laufende Prozess unterbrochen (preemption).

Mehrschlangen-Scheduling (2)

Bedienmodell (Beispiel):



Mehrschlangen-Feedback-Scheduling

Prinzip:

- Erweiterung des Mehrschlangen-Scheduling.
- Rechenwillige Prozesse können im Verlauf in verschiedene Warteschlangen eingeordnet werden (dynamische Prioritäten).
- Algorithmen zur **Neubestimmung der Priorität** wesentlich

Bsp. 1: Wenn ein Prozess blockiert, wird die Priorität nach Ende der Blockierung um so größer, je weniger er von seinem Quantum verbraucht hat (Bevorzugung von I/O-intensiven Prozessen).

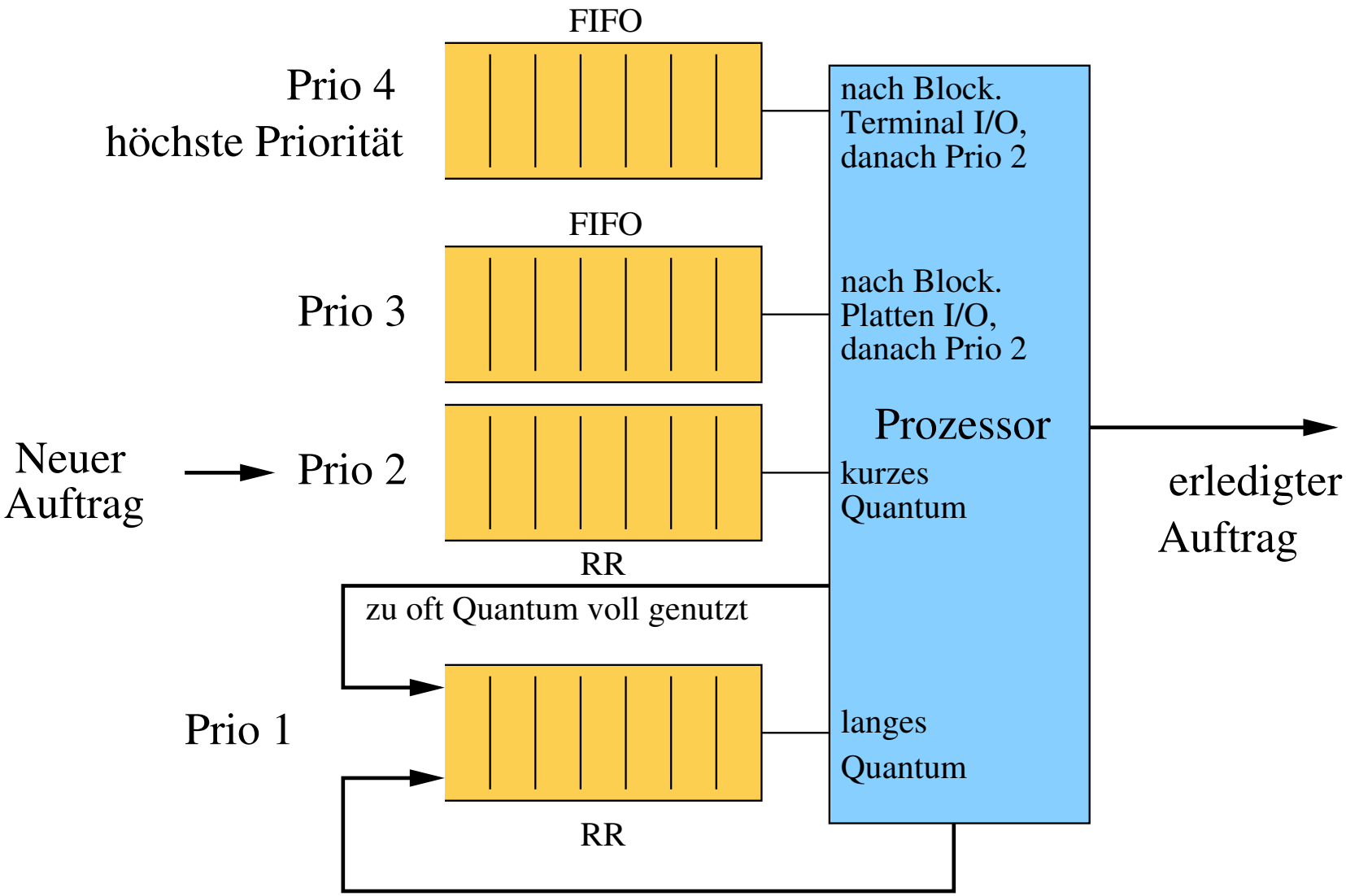
Bsp. 2: Wenn ein Prozess in einer bestimmten Priorität viel Rechenzeit zugeordnet bekommen hat, wird seine Priorität verschlechtert (Bestrafung von Langläufern).

Bsp. 3: Wenn ein Prozess lange nicht bedient worden ist, wird seine Priorität verbessert (Altern, Vermeidung einer „ewigen“ Bestrafung).

Mehrschlangen-Feedback-Scheduling (2)

Hochschule RheinMain

Bedienmodell:



Bewertung

- Mit wachsender Bedienzeit sinkt die Priorität, d.h. Kurzläufer werden bevorzugt, Langläufer werden zurückgesetzt.
- Wachsende Länge des Quantums mit fallender Priorität verringert die Anzahl der notwendigen Prozesswechsel (Einsparen von Overhead).
- Verbesserung der Priorität nach Beendigung einer Blockierung berücksichtigt I/O-Verhalten (Bevorzugung von I/O-intensiven Prozessen). Durch Unterscheidung von Terminal I/O und sonstigem I/O können interaktive Prozesse weiter bevorzugt werden.
- sehr flexibel.
- Die Scheduler in Windows und Linux arbeiten nach diesem Prinzip

Scheduling in Linux (1)

- Linux 1.2
  - Zyklische Liste, Round-Robin
- Linux 2.2
  - Scheduling-Klassen (Echtzeit, Non-Preemptive, Nicht-Echtzeit)
  - Unterstützung für Multiprozessoren
- Linux 2.4
  - O(n)-Komplexität (jeder Task-Kontrollblock muss angefasst werden)
  - Round-Robin
  - Teilweiser Ausgleich bei nicht verbrauchter Zeitscheibe
  - Insgesamt relativ schwacher Algorithmus

Scheduling in Linux (2)

- Linux 2.6
  - O(1)-Komplexität (konstanter Aufwand für Auswahl unabhängig von Anzahl Tasks)
  - Run Queue je Priorität
  - Zahlreiche Heuristiken für Entscheidung I/O-intensiv oder rechenintensiv
  - Sehr viel Code
- ab Linux Kernel 2.6.23: „Completely Fair Scheduler“ (CFS)
  - Sehr gute Approximation von Processor Sharing
  - Task mit geringster *Virtual Runtime* (größter Rückstand) bekommt Prozessor
  - Zeit-geordnete spezielle Baumstruktur für Taskverwaltung ($\rightarrow$ O(log n)-Komplexität)
  - Kein periodischer Timer-Interrupt sondern One-Shot-Timer („tickless Kernel“)

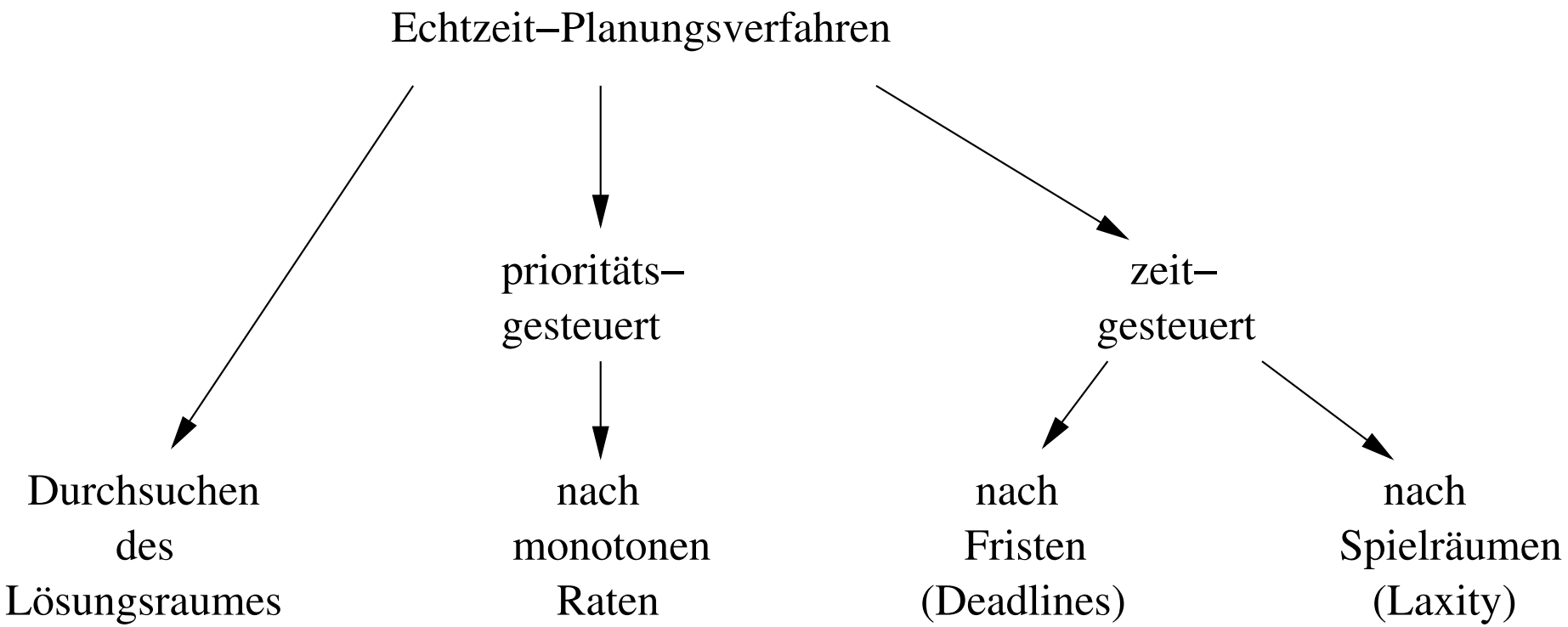
Echtzeit-Scheduling

- Scheduling in Realzeit-Systemen beinhaltet zahlreiche neue Aspekte. Hier nur erster kleiner Einblick.[5]
- Varianten in der Vorgehensweise
  - *Statisches Scheduling:*
    Alle Daten für die Planung sind vorab bekannt, die Planung erfolgt durch eine Offline-Analyse.
  - *Dynamisches Scheduling:*
    Daten für die Planung fallen zur Laufzeit an und müssen zur Laufzeit verarbeitet werden.
  - *Explizite Planung:*
    Dem Rechensystem wird ein vollständiger Ausführungsplan (Schedule) übergeben und zur Laufzeit befolgt (Umfang kann extrem groß werden).
  - *Implizite Planung:*
    Dem Rechensystem werden nur die Planungsregeln übergeben.

[5] Mehr dazu im Listenfach „Echtzeitverarbeitung“ im nächsten SoSe

Klassifizierung

Hochschule RheinMain



Periodische Prozesse

- Gewisse Prozesse müssen häufig zyklisch, bzw perodisch ausgeführt werden.
- **Hard-Realtime**-Prozesse müssen unter allen Umständen ausgeführt werden (ansonsten sind z.B. Menschenleben bedroht).
- Zeitliche **Fristen (*Deadlines*)** vorgegeben, zu denen der Auftrag erledigt sein muss.
- Scheduler muss die Erledigung aller Hard-Realtime-Prozesse innerhalb der Fristen **garantieren**.
- Scheduling geschieht in manchen Anwendungssystemen statisch vor Beginn der Laufzeit (z.B. Automotive). Dazu muss die Bedienzeit-Anforderung (z.B. worst case) bekannt sein.
- Im Falle von dynamischem Scheduling sind das **Rate-Monotonic** (RMS) und das **Earliest-Deadline-First** (EDF) Scheduling-Verfahren verbreitet.

Rate Monotonic Scheduling (RMS) (1)

- Ausgangspunkt: Periodisches Prozessmodell
  - Planungsproblem gegeben als Menge unterbrechbarer, periodischer Prozesse P_i mit Periodendauern Δp_i und Bedienzeiten Δe_i.
  - Perioden zugleich Fristen.
- RMS ordnet Prozessen **feste Prioritäten** proportional zur **Rate**[6] zu:

$$prio(i) < prio(j) \iff \frac{1}{\Delta p_i} < \frac{1}{\Delta p_j}$$

- Daher auch *fixed priority scheduling*
- Die meisten Echtzeit-Betriebssysteme unterstützen prioritätsbasiertes, unterbrechendes Scheduling

→ Voraussetzungen für die Anwendung sind unmittelbar gegeben

- Zur Festlegung der Prioritäten genügt allein die Kenntnis der Periodendauern Δp_i

[6] = Kehrwert der Periodendauer

Rate Monotonic Scheduling (RMS) (2)

- RMS Zulassungskriterium (*admission test*):
  Wenn für n periodische Prozesse gilt ...:

$$\sum_{i=0}^{n} \frac{\Delta e_i}{\Delta p_i} \leq n \cdot \left(2^{\frac{1}{n}} - 1\right)$$

- ...dann ist bei Prioritätsvergabe nach RMS **garantiert**, dass alle Fristen eingehalten werden.
- Hinreichendes (nicht: notwendiges) Kriterium
- Einfach zu überprüfen, mathematisch beweisbare Garantie
- Erfordert Kenntnis der *worst case* Bedienzeiten (WCET)

Beispiel

Hochschule RheinMain

Gegeben: Prozessmenge mit 2 Prozessen

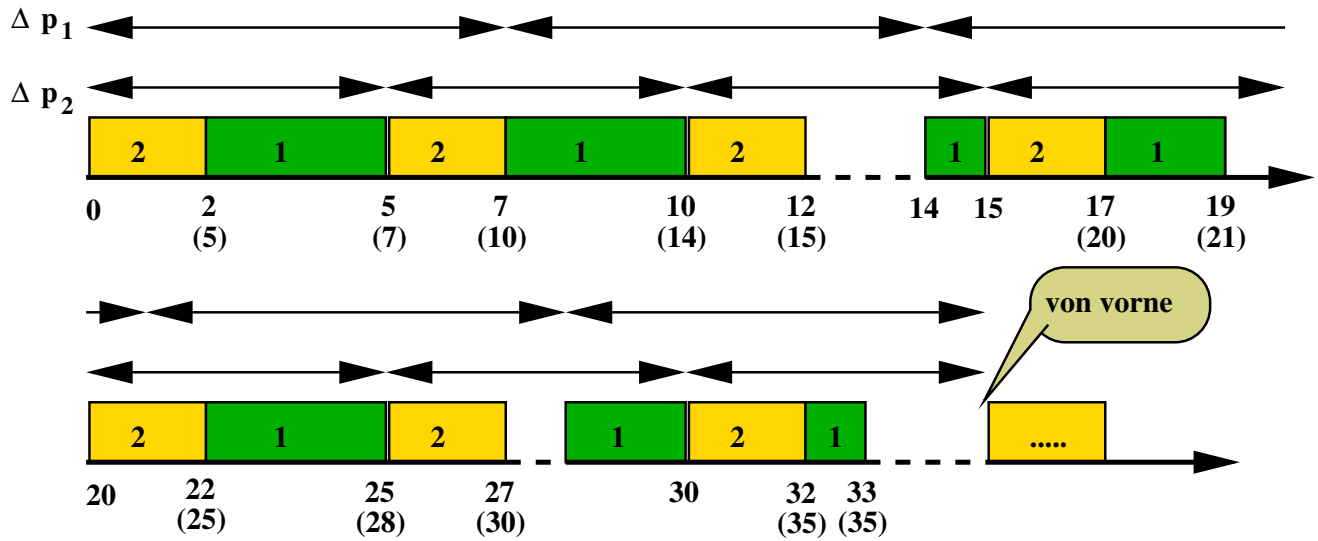
Prozess i	Bedienzeit Δe_i	Periode Δp_i
1	3	7
2	2	5

$$\frac{3}{7} + \frac{2}{5} \approx 0,8286$$

$$2 \cdot \left(2^{\frac{1}{2}} - 1\right) \approx 0,8284$$

→ Kriterium knapp nicht erfüllt, trotzdem wurde ein Plan gefunden

- Aus RMS resultierender Schedule [7]:



[7] (wg. $\frac{1}{7} < \frac{1}{5}$ bekommt P_2 höhere Priorität)

Earliest Deadline First Scheduling (EDF) (1)

- Strategie: *Earliest Deadline First* (EDF)
  - Der Prozessor wird demjenigen Prozess P_i zugeteilt, dessen Frist d_i den kleinsten Wert hat (am nächsten ist)
  - Wenn es keinen rechenbereiten Prozess gibt, bleibt der Prozessor untätig (d.h. „idle“)
- Falls EDF keinen brauchbaren Plan liefert, gibt es keinen (!)
- Zur Planung nach EDF genügt allein die Kenntnis der Fristen, bzw. der Periodendauern Δp_i
- Die Umsetzung eines EDF-Planes mithilfe des prioritätsbasierten, unterbrechenden Scheduling erfordert die dynamische Änderung von Prozessprioritäten zur Laufzeit.
- Daher auch *dynamic priority scheduling*
- Nicht alle Echtzeitbetriebssysteme unterstützen dynamische Prioritäten.

Earliest Deadline First Scheduling (EDF) (2)

- EDF Zulassungskriterium (*admission test*):
  Wenn für n periodische Prozesse gilt ...:

$$\sum_{i=0}^{n} \frac{\Delta e_i}{\Delta p_i} \leq 1$$

- ...dann ist bei Planung nach EDF **garantiert**, dass alle Fristen eingehalten werden.
- Notwendiges und hinreichendes Kriterium
- Einfach zu überprüfen, mathematisch beweisbare Garantie
- Erfordert Kenntnis der *worst case* Bedienzeiten (WCET)

Beispiel

Hochschule RheinMain

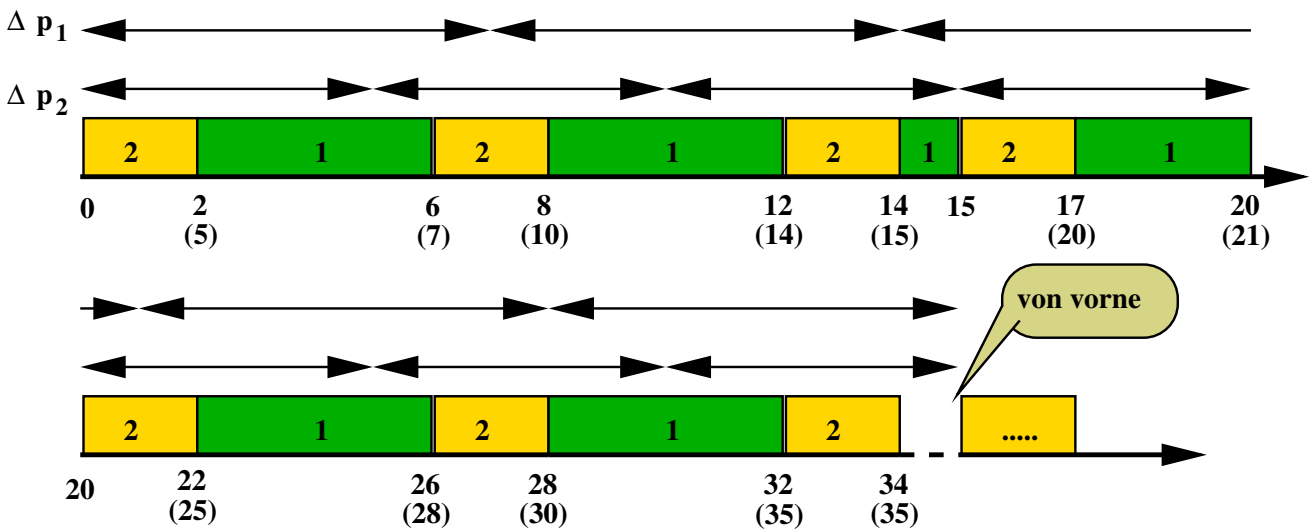
Gegeben: Prozessmenge mit 2 Prozessen

Prozess i	Bedienzeit Δe_i	Periode Δp_i
1	4	7
2	2	5

$$\frac{4}{7} + \frac{2}{5} \approx 0,97143 < 1$$

$\rightarrow$ Kriterium erfüllt, Plan existiert

- Aus EDF resultierender Schedule



Gegenüberstellung RMS ↔ EDF

- RMS
  - \- Keine 100% Auslastung möglich
  - \+ Auf gängigen Echtzeit-BS direkt einsetzbar
  - \+ Bei Überlastsituationen werden zunächst niedrig priorisierte Prozesse nicht mehr bedient
- EDF
  - \+ 100% Auslastung möglich
  - \- Erfordert dynamische Prioritäten - nicht auf allen Echtzeit-BS möglich
  - \- Bei Überlastsituationen erratisches Verhalten

7) Deadlocks (Systemverklemmungszustand)

Betriebsmittel: Können sowohl HW- als auch SW-Komponenten sein. (CD Brenner, CPU…)

Benutzung von BM: Anfordern, Benutzen, Freigeben.

Deadlock: Prozessmenge ist im Deadlock-Zustand falls ein Prozess auf ein Ereginis eines anderen Prozesses dieser Menge wartet.

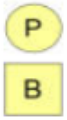
Vorraussetzungen:
-Wechselseitiger Ausschluß: (BM frei oder einem Prozess zugeteilt)
-Belegungs-Anforderungsbedingung (Hold-and-wait): Prozesse können zu bereits reservierten BM noch weitere anfordern
-Ununterbrechbarkeit: zugeteilt BM müssen freigegeben werden um für einen anderen Prozess verfügbar zu sein.
-Zyklisches Warten: Es muss eine zyklische Kette von Prozessen geben, in der jeder Prozess auf ein Betriebsmittel wartet, das dem nächsten Prozess in der Kette gehört.
-**ALLE 4** Bedingungen **gleichzeitig erfüllt** sind → Deadlock **möglich**

Belegungs-Anforderungs-Graphen

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

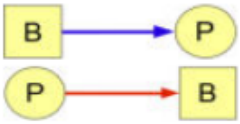
Es gibt zwei Knotentypen:

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

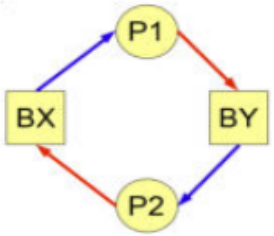


Pfeile:

- P belegt B
- P wartet auf B



Zyklus im Graphen → Deadlock



Verfahren zur Deadlock-Behandlung

- Mit Betriebsmittelzuteilungsgraphen lassen sich Deadlocks erkennen.

1) Ignorieren (Vogel-Strauß-Algorithmus)
2) Erkennen & Beheben:
   - Belegungs-/Anforderungs-Graph erstellen und nach Zyklen absuchen
   - falls Zyklus gefunden wurde: Deadlock beheben
   -Untersuchung kann bei BM Anforderungen, in regelmäßigen Zeitabständen oder bei Verdacht (CPU-Auslastung niedrig) stattfinden

Belegungs-Anforderungs-Graphen

Hochschule RheinMain

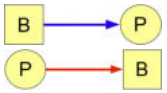
Graphische Darstellung der Beziehung von Prozessen zu Betriebsmitteln (Holt, 1972)

Es gibt zwei Knotentypen:

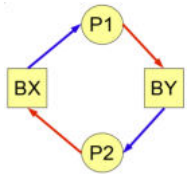
- Prozesse, repräsentiert durch Kreise:
- Betriebsmittel, repräsentiert durch Quadrate:

Pfeile:

- P belegt B
- P wartet auf B



Zyklus im Graphen → Deadlock



Notizen

Beispiel

Hochschule RheinMain

Gegeben:

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

Prozess A	**Prozess B**	**Prozess C**
- Anforderung R	- Anforderung S	- Anforderung T
- Anforderung S	- Anforderung T	- Anforderung R
- Freigabe R	- Freigabe S	- Freigabe T
- Freigabe S	- Freigabe T	- Freigabe R

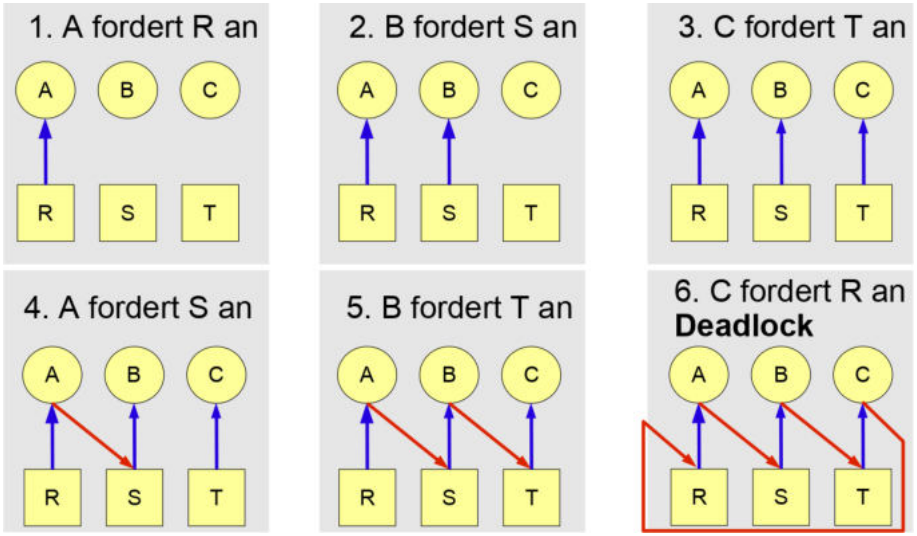
Das Betriebssystem kann jeden (nicht blockierten) Prozess **jederzeit** ausführen

Sequentielle Ausführung von A, B, C wäre unproblematisch (dann aber auch keine Nebenläufigkeit)

Wie sieht es bei nebenläufiger Ausführung aus?

Notizen

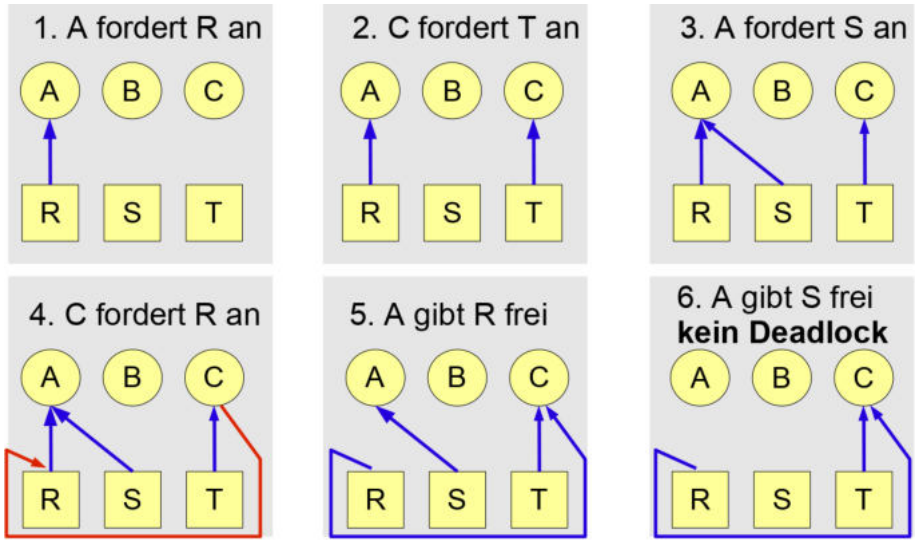
Ausführung I



Notizen

Ausführung II

(B zunächst suspendiert)



Notizen

Verfahren zur Deadlock-Behandlung

Hochschule RheinMain

Mit Betriebsmittelzuteilungsgraphen (*„Belegungs/Anforderungs-Graphen“*) lassen sich Deadlocks erkennen (→Zyklus im Graph)
Wie weiter verfahren?

- **Ignorieren** („Vogel-Strauß-Verfahren“)
- Deadlocks **erkennen** und **beheben**
- **Verhinderung** durch Planung der Betriebsmittelzuordnung (*deadlock avoidance*)
- **Vermeidung** durch Nichterfüllung (mindestens) einer der vier Voraussetzungen für Deadlocks (*deadlock prevention*)

Diese Strategien werden im folgenden untersucht.

Notizen

Ignorieren des Problems

Hochschule RheinMain

„Vogel-Strauß-Algorithmus“

Ausdruck optimistischer Lebenshaltung:

„Deadlocks kommen in der Praxis sowieso nie vor“

http://clipart.coolclips.com/480/vectors/tf05038/CoolClips_anim0613.png

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

Beispiel:

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

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

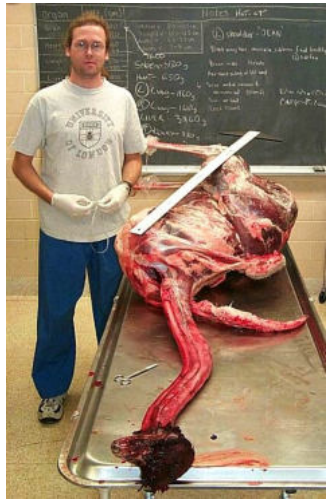
Notizen

...manchmal nicht so gut

Hochschule RheinMain

http://clipart.coolclips.com/480/vectors/tf05038/CoolClips_anim0613.png

→



http://www.instore.si/newsarticle/newsarticle/Septembra-v-Aldiju-nojevo-meso

Notizen

Deadlock-Erkennung und Behebung

Hochschule RheinMain

Engl.: ***deadlock detection and resolution / recovery***

Vorgehensweise: Das Auftreten von Deadlocks wird vom Betriebssystem nicht verhindert. Es wird versucht, Deadlocks zu erkennen und anschließend zu beheben.

Betrachtet werden im folgenden:

- Deadlock-Erkennung mit einem Betriebsmittel je Klasse (Einfacher Fall)
- Deadlock-Erkennung mit mehreren Betriebsmitteln je Klasse (Allgemeiner Fall)
- Verfahren zur Deadlock-Behebung

Notizen

Deadlocks erkennen (Einfacher Fall)

Hochschule RheinMain

Vereinfachende Annahme: **Ein Betriebsmittel** je Betriebsmitteltyp

Vorgehen:

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

Wann wird die Untersuchung durchgeführt?

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

Notizen

Beispiele: Sicher?

Hochschule RheinMain

4 Prozesse, ein Betriebsmitteltyp (10 Stück vorhanden)

verfügbar: 10

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

sicher!

z.B. sequenzielle Ausführung von A, B, C, D in beliebiger Reihenfolge ist möglich.

verfügbar: 2

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

sicher!

C ist ausführbar,(→ dann 4 verfügbar) dann D, B, A möglich.

verfügbar: 1

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

unsicher!

Differenz $max - hat$ immer $> verfgbar$. Deadlock, sobald irgend ein Prozess auf sein Maximum zugeht

Notizen

Beispiel

Hochschule RheinMain

A belegt R und fordert S an.

B fordert T an.

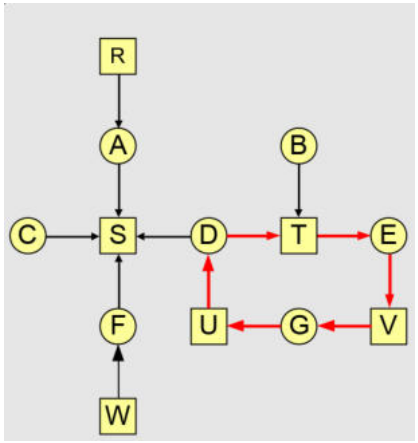
C fordert S an.

D belegt U und fordert S und T an.

E belegt T und fordert V an.

F belegt W und fordert S an.

G belegt V und fordert U an.



Notizen

Deadlocks erkennen

Hochschule RheinMain

Erweiterung: Mehrere (E_i-viele) Betriebsmittel je Betriebsmitteltyp i (z.B. mehrere Drucker)

Prozesse $P_1, \ldots, P_n$

$$E = (E_1, E_2, \ldots, E_m)$$

Betriebsmittelvektor E: Gesamtzahl der BM je Typ i

$$A = (A_1, A_2, \ldots, A_m)$$

Verfügbarkeitsvektor A: Gesamtzahl der BM je Typ i

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

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

Anforderungsmatrix R: Zeile j gibt BM-Belegung durch Prozess j an („Prozess j belegt R_{jk} Einheiten von BM k")

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

Notizen

Erkennungsalgorithmus

Hochschule RheinMain

Zu Beginn sind alle Prozesse aus P unmarkiert
(Markierung heißt, dass der Prozess in keinem DL steckt)

Suche einen Prozess, der ungehindert durchlaufen kann, also einen unmarkierten Prozess P_i, dessen Zeile in der Anforderungsmatrix-Zeile R_i (komponentenweise) kleiner oder gleich dem Verfügbarkeitsvektor A ist

Kein passendes P_i gefunden? Dann → **Ende**

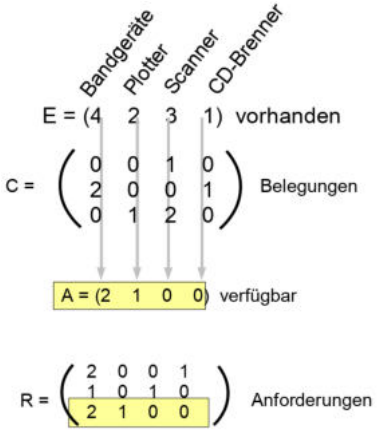
Gefunden? Dann kann P_i durchlaufen und gibt danach seine belegten Betriebsmittel zurück: $A = A + C_i$, wird markiert und es geht beim nächsten unmarkierten Prozess weiter

Beim Ende des Verfahrens sind **alle unmarkierten** Prozesse an einem **Deadlock beteiligt**.

Notizen

Beispiel

Hochschule RheinMain



Ausführbar ist zunächst nur P_3
Freigabe $C_3 = (0120)$
⇒ $A = (2100) + (0120)$
⇒ $A = (2220)$
Nun ausführbar: P_2
(benötigt $R_2 = (1010)$)
Freigabe $C_2 = (2001)$
⇒ $A = (4221)$
Schließlich auch P_1 ausführbar
⇒ $A = (4231)$
⇒ Alle Prozesse markiert,
kein Deadlock aufgetreten.

Notizen

Beheben von Deadlocks

Hochschule RheinMain

Wie kann man auf erkannte Deadlocks reagieren?

Prozessunterbrechung

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

Teilweise Wiederholung (*rollback*)

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

Prozessabbruch

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

Notizen

Verhindern von Deadlocks

Hochschule RheinMain

Bisher: Erkennung von Deadlocks, gegebenenfalls „drastische“ Maßnahmen zur Auflösung

Annahme bisher: Prozesse fordern alle Betriebsmittel „auf ein Mal“ an (vgl. 7.4.2).

In den meisten praktischen Fällen werden BM jedoch nacheinander angefordert

Das Betriebssystem muss dann dynamisch über die Zuteilung entscheiden

Notizen

Verhindern von Deadlocks

Hochschule RheinMain

Kann man **Deadlocks** durch „geschicktes“ Vorgehen bei der Betriebsmittelzuteilung **von vornherein verhindern**?

Welche Informationen müssen dazu vorab zur Verfügung stehen?

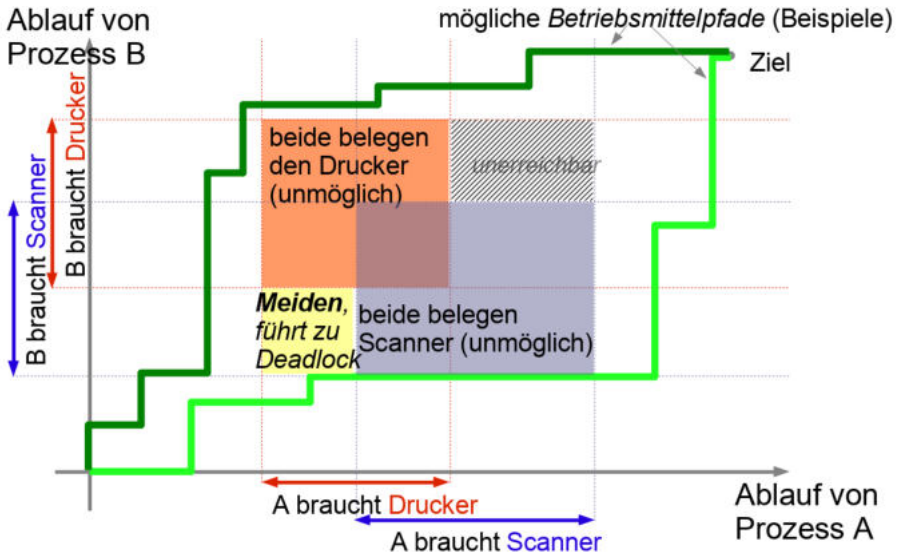
Im folgenden betrachtet

- Betriebsmittelpfade (Grafische Veranschaulichung)
- Sichere und unsichere Zustände
- Der vereinfachte Bankiersalgorithmus für eine BM-Klasse
- Der Bankiersalgorithmus für mehrere BM-Klassen

Notizen

Betriebsmittelpfade

Hochschule RheinMain



Notizen

(Un-)Sichere Zustände

Hochschule RheinMain

Definition

Ein Systemzustand ist **sicher**, wenn er

keinen Deadlock repräsentiert und
es eine geeignete Prozessausführungsreihenfolge gibt, bei der alle Anforderungen erfüllt werden
(die also **auch dann** nicht in einen Deadlock führt, wenn alle Prozesse gleich ihre max. Ressourcenanzahl anfordern)

Sonst heißt der Zustand **unsicher**.

Bei einem sicherem Zustand kann das System **garantieren**, dass alle Prozesse bis zum Ende durchlaufen können.
Bei unsicherem Zustand ist das nicht garantierbar (aber auch nicht ausgeschlossen!).

Beispiel: Ein Prozess gibt ein BM zu einem „glücklichen Zeitpunkt“ kurzzeitig frei, wodurch eine Deadlock-Situation „zufällig“ vermieden wird. (→ „Glück“ nicht vorhersehbar)

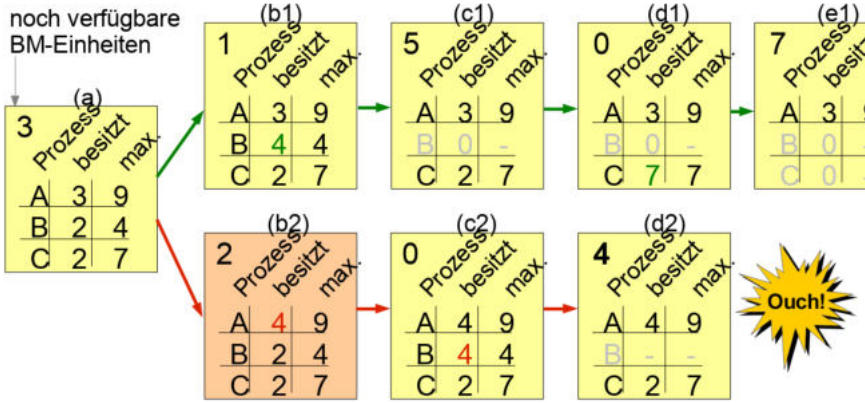
„Unsicher“ bedeutet also nicht „Deadlock unvermeidlich“.

Notizen

Beispiel

Hochschule RheinMain

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



Zustand (a) ist sicher (es gibt eine DL-freie Lösung)
(b2) ist **nicht** sicher (A und C brauchen je 5, frei sind nur 4)

Notizen

Bankier-Algorithmus (1 BM-Klasse)

Hochschule RheinMain

- Dijkstra (wer sonst? 1965):
- Ein **Bankier kennt** die **Kreditrahmen** seiner Kunden.
- Er geht davon aus, dass **nicht alle** Kunden **gleichzeitig** ihre Rahmen **voll** ausschöpfen werden.
- Daher hält er **weniger Bargeld** bereit als die **Summe** der Kreditrahmen.
- Gegebenenfalls **verzögert** er die **Zuteilung** eines Kredits, bis ein anderer Kunde zurückgezahlt hat.
- **Zuteilung** erfolgt **nur**, wenn sie **"sicher"** ist (also letztlich alle Kunden bis zu ihrem Kreditrahmen bedient werden können).

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

Notizen

Bankier-Algorithmus (2)

Hochschule RheinMain

Prüfe bei jeder Anfrage, ob die Bewilligung in einen sicheren Zustand führt:

Prüfe dazu, ob ausreichend Betriebsmittel bereitstehen, um **mindestens einen** Prozess **vollständig** zufrieden zu stellen.

Davon ausgehend, dass dieser Prozess nach Durchlauf seine Betriebsmittel freigibt: führe **Test** mit dem Prozess aus, der dann am nächsten am Kreditrahmen ist

usw., **bis alle** Prozesse positiv getestet sind;

Falls **ja**, kann die aktuelle Anfrage **bewilligt** werden.

Sonst: Anforderung **verschieben** (warten)

Notizen

Verallgemeinerter Bankier-Algorithmus

Hochschule RheinMain

Mehrere Betriebsmittelklassen

Datenstrukturen wie bei „Deadlockerkennung“ (7.4.2)

Matrizen mit belegten / angeforderten Betriebsmitteln

Vektoren mit BM-Bestand, verfügbaren BM und belegten BM je Betriebsmitteltyp

- E Betriebsmittelvektor
- A Verfügbarkeitsvektor
- C Belegungsmatrix
- R Anforderungsmatrix

Notizen

Beispiel

Hochschule RheinMain

$$E = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix} \text{ vorhanden}$$

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ zugewiesen}$$

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 \end{pmatrix} \text{ verfügbar}$$

$$P = \begin{pmatrix} 5 & 3 & 2 & 2 \end{pmatrix} \text{ belegt}$$

$$R = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix} \text{ angefordert}$$

Sicher? Ja, Ausführungsfolge P_4, P_1, P_5, ...ist möglich:

$$P_4 \rightarrow A = \begin{pmatrix} 2 & 1 & 2 & 1 \end{pmatrix}$$

$$P_1 \rightarrow A = \begin{pmatrix} 5 & 1 & 3 & 2 \end{pmatrix}$$

$$P_5 \rightarrow A = \begin{pmatrix} 5 & 1 & 3 & 2 \end{pmatrix}$$

$$P_2 \rightarrow A = \begin{pmatrix} 5 & 2 & 3 & 2 \end{pmatrix}$$

$$P_3 \rightarrow A = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix}$$

Notizen

Beispiel

Hochschule RheinMain

$$E = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix} \text{ vorhanden}$$

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ zugewiesen}$$

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix} \text{ verfügbar}$$

$$P = \begin{pmatrix} 5 & 3 & 3 & 2 \end{pmatrix} \text{ belegt}$$

$$R = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix} \text{ angefordert}$$

P2 fordere ein BM 3 an (rot)

Sicher? Ja, Ausführungsfolge P_4, P_1, P_5, P_2,P_3 möglich:

$P_4 \rightarrow A = \begin{pmatrix} 2 & 1 & 1 & 1 \end{pmatrix}$

$P_1 \rightarrow A = \begin{pmatrix} 5 & 1 & 2 & 2 \end{pmatrix}$

$P_5 \rightarrow A = \begin{pmatrix} 5 & 1 & 2 & 2 \end{pmatrix}$

$P_2 \rightarrow A = \begin{pmatrix} 5 & 2 & 3 & 2 \end{pmatrix}$

$P_3 \rightarrow A = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix}$

also erhält P_2 ein BM3

Notizen

Beispiel

Hochschule RheinMain

$$E = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix} \text{ vorhanden}$$

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \text{ zugewiesen}$$

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \text{ verfügbar}$$

$$P = \begin{pmatrix} 5 & 3 & 4 & 2 \end{pmatrix} \text{ belegt}$$

$$R = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix} \text{ angefordert}$$

Nun fordere auch P_5 ein BM 3 an

→ dann würde $A = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}$

Sicher? *Nein!*

→ daher Anfrage von P_5 blockieren

Notizen

Ist der Bankier-Algorithmus praktikabel?

Hochschule RheinMain

In der Praxis gibt es mehrere Probleme beim Einsatz:

- Prozesse können „maximale Ressourcenanforderung“ selten im Voraus angeben
- Anzahl der Prozesse ändert sich ständig
- Ressourcen können verschwinden (z.B. durch Ausfall)

Notizen

Deadlock-Vermeidung

Hochschule RheinMain

Deadlock-Verhinderung ist wenig praktikabel ☹

Ansatz: **Vermeidung** mindestens einer der vier Deadlock-**Voraussetzungen** (vgl 7.2)

- Wechselseitiger Ausschluss
- Belegungs-/Anforderungsbedingung („Hold-and-Wait“, d.h. zu reservierten BM weitere anforderbar)
- Ununterbrechbarkeit (kein erzwungener BM-Entzug)
- zyklisches Warten

Notizen

1. Wechselseitiger Ausschluß?

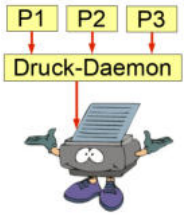
Hochschule RheinMain

Falls es keine exklusive Zuteilung eines Betriebsmittels an einen Prozess gibt, gibt es auch keine Deadlocks.

Beispiel: Zugriff auf Drucker

Einführung eines **Spool-Systems**, das

- Druckaufträge von Prozessen (schnell) entgegennimmt
- ggf. zwischenspeichert
- und der Reihe nach auf dem Drucker ausgibt



Entkopplung zwischen (konkurrierenden) Prozessen und dem (langsamen) Betriebsmittel

Vermeidung einer exklusiven Zuteilung des Betriebsmittels „Drucker“

Notizen

2. Belegungs-/Anforderungsbedingung?

Hochschule RheinMain

Vermeiden, dass neue Betriebsmittel-Anforderungen zu bereits bestehenden hinzukommen.

„Preclaiming“: Alle Anforderungen zu Beginn der Ausführung stellen („alles oder nichts“)

Vorteil: Wenn Anforderungen erfüllt werden, kann der Prozess sicher bis zum Ende durchlaufen (er hat ja dann alles, was er braucht)

Nachteil:

- Anforderungen müssen **zu Beginn bekannt** sein
- Betriebsmittel werden unter Umständen **lange blockiert**
- und können zwischenzeitlich nicht (sinnvoll) anders genutzt werden.

Beispiel: Batch-Jobs bei Großrechnern.

Notizen

3. Ununterbrechbarkeit?

Hochschule RheinMain

Hängt vom Betriebsmittel ab, aber

„gewaltsamer“ Entzug ist in der Regel nicht akzeptabel

- Drucker?
- CD-Brenner?

Notizen

4. Zyklische Wartebedingung?

Hochschule RheinMain

Wenn es kein zyklisches Auf-einander-warten gibt, entstehen auch keine Deadlocks

Idee:

- Betriebsmitteltypen **linear ordnen** und
- nur in aufsteigender Ordnung Anforderungen annehmen (wenn mehrere Exemplare eines Typs gebraucht werden: alle Exemplare auf einmal anfordern)
- z.B. „Drucker *vor* Scanner *vor* CD-Brenner *vor* …“

Dadurch entsteht **automatisch** ein **zyklenfreier** Belegungs-Anforderungs-Graph,

wodurch Deadlocks ausgeschlossen sind.

Tatsächlich praktikables Verfahren.

Notizen

Deadlock-Vermeidung im Überblick

Hochschule RheinMain

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

Wechselseitiger Ausschluß	→ Spooling
Belegungs-/Anforderungsbed.	→ Preclaiming
Ununterbrechbarkeit	(BM-Entzug...besser nicht)
Zyklisches Warten	→ Betriebsmittel ordnen

Notizen

Verwandte Fragestellungen

Hochschule RheinMain

Deadlocks bei der Benutzung von Semaphoren (vgl. Kap. 3)

Zwei-Phasen-Locking in Datenbanken

Verhungern (Starvation), kein Deadlock, aber auch kein Fortschritt für einen Prozess (vgl. Philosophen-Problem)

Notizen

8) Cache

-Prozessor immer noch schneller als Speicher (ca. 10 mal)
-Maschinenbefehl besteht aus Opcode und ggf Operanden
-Cache gehört zur Mikroarchitektur und nicht zur Instruction Set Architecture (ISA)
-Räumliche Lokalität:Zugriffe häufig auf Adresse in Nähe bereits zuvor benutzer Adressen
-Zeitliche Lokalität:Zugriffe auf dieselbe/benachbarte Adressen zeitlich nahe beieinander
-beide Lokalitätsprinzipien treffen auf Befehlzugriffe (meisten, außer bei jmp), Daten (programmabhängig)
-Working Set: Gesamtheit der Speicherobjekte auf die ein Prozess zugreift. (Stack, evt. Shared Libs, Heap, bss, .data, .text). Working Set besteht aus > 5-6 „Regionen“
- Cache hält Kopien von im Speicher liegenden Objekten (Gefahr: Inkosistenz)
- Konsistent: Alle Cache-Kopien und Originial im Hauptspeicher haben gleichen Wert
- Kohärent: Cache und Hauptspeicher für Objekt liefern gleichen Wert
- Vorübergehende INKOSISTENZ ist tolerabel
- Bei L1: separater Cache für Daten und Befehle (jeweils ca. 64 kB, L2: 4 MB)

-Kohärenzprotokoll
  -Lesezugriff
    - Hit: Daten aus Cache liefern
    - Miss: Daten liefern und in Cache kopieren
  -Schreibzugriff
    -Hit
      -Write Through: Daten in Speicher und Cache schreiben
      -Copy-Back: Daten nur in Cache speicher. Cache Zeile ist „dirty“
    -Miss
      -No Write Allocate: Daten nur in Speicher schreiben
      -Write Allocate:Daten mit umliegender Zeile in Speicher & Cache

-Effektive Wartezeit: $T_{eff} = H * T_{hit} + (1 - H) * T_{miss}$
mehrstufig: $T_{eff} = H * T_{hit1} + (1 - H1) * (H2 * T_{hit2} + (1 - H2) * T_{miss})$

-Assoziativspeicher (auch „inhaltsadressierter Speicher“, Wertepaare: Adresse, Daten)
  - enthält Kopien kleiner (max 100 B) Hauptspeicher-Ausschnitte
  - Cache-Eintrag: Tag (Etikett), Daten (Cache-Zeile), Valid Bit, Dirty Bit
  - Bei Speicherzugriff: gleichzeitiger Vergleich der Adresse mit Cache-Einträgen

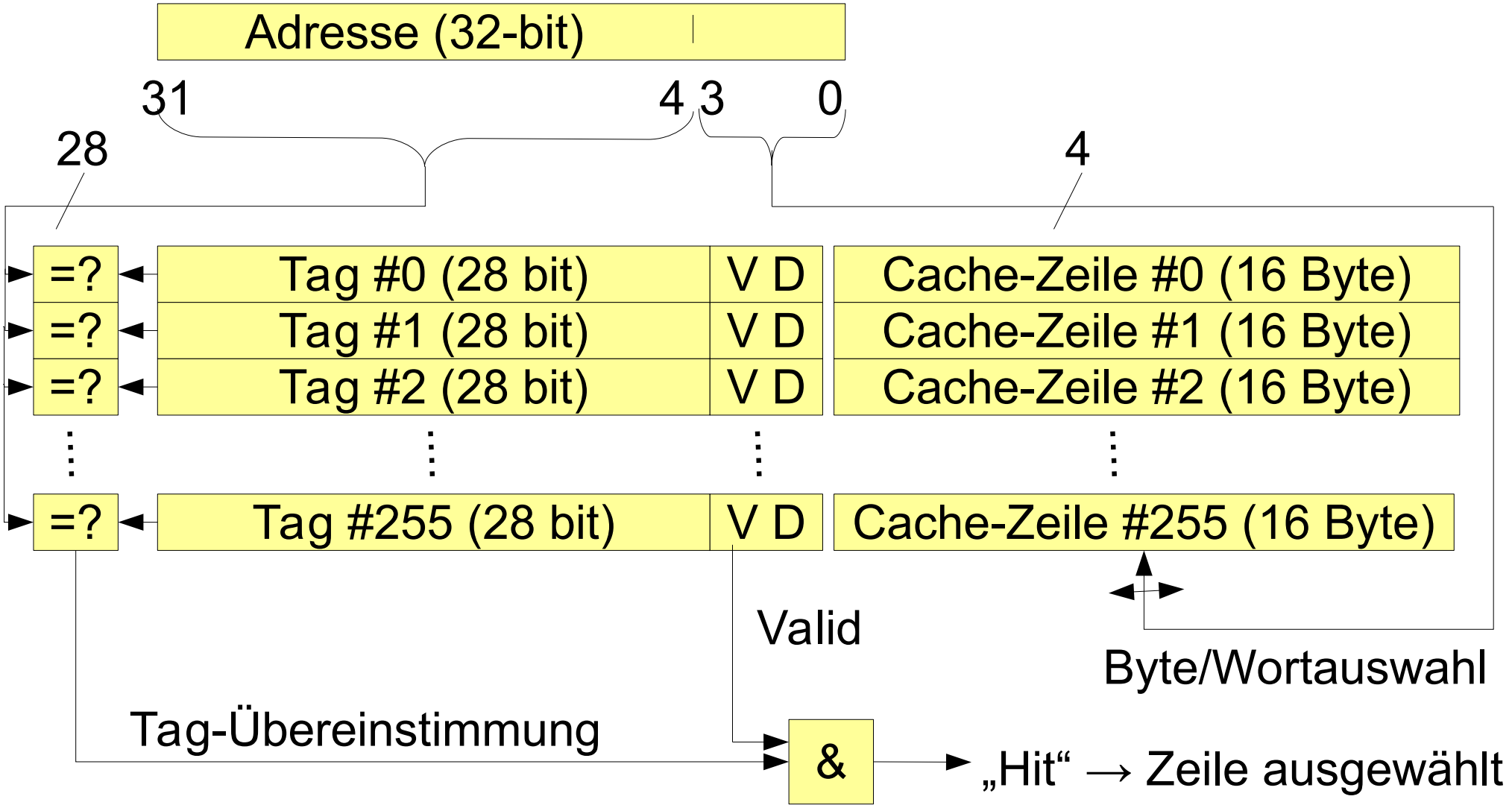
-Verdrängungsstrategien (wenn alle Cache Zeilen belegt sind → „Platz schaffen“)
  - Random (einfach, überraschend gut)
  - FIFO (die im längsten im Cache gespeicherte Adresse wird ersetzt → schlecht)
  - LRU (least recently used): die im längsten nicht verwendete Zeile ersetzen
  - LFU (least frequently used): die am wenigsten verwendete Zeile ersetzen
-Organisationformen
  - vollassoziativ: fully associative
  -direkt abbildend: direct-mapped
  -mehrfach assoziativ: N-way set associative

Cache-Organisationsformen

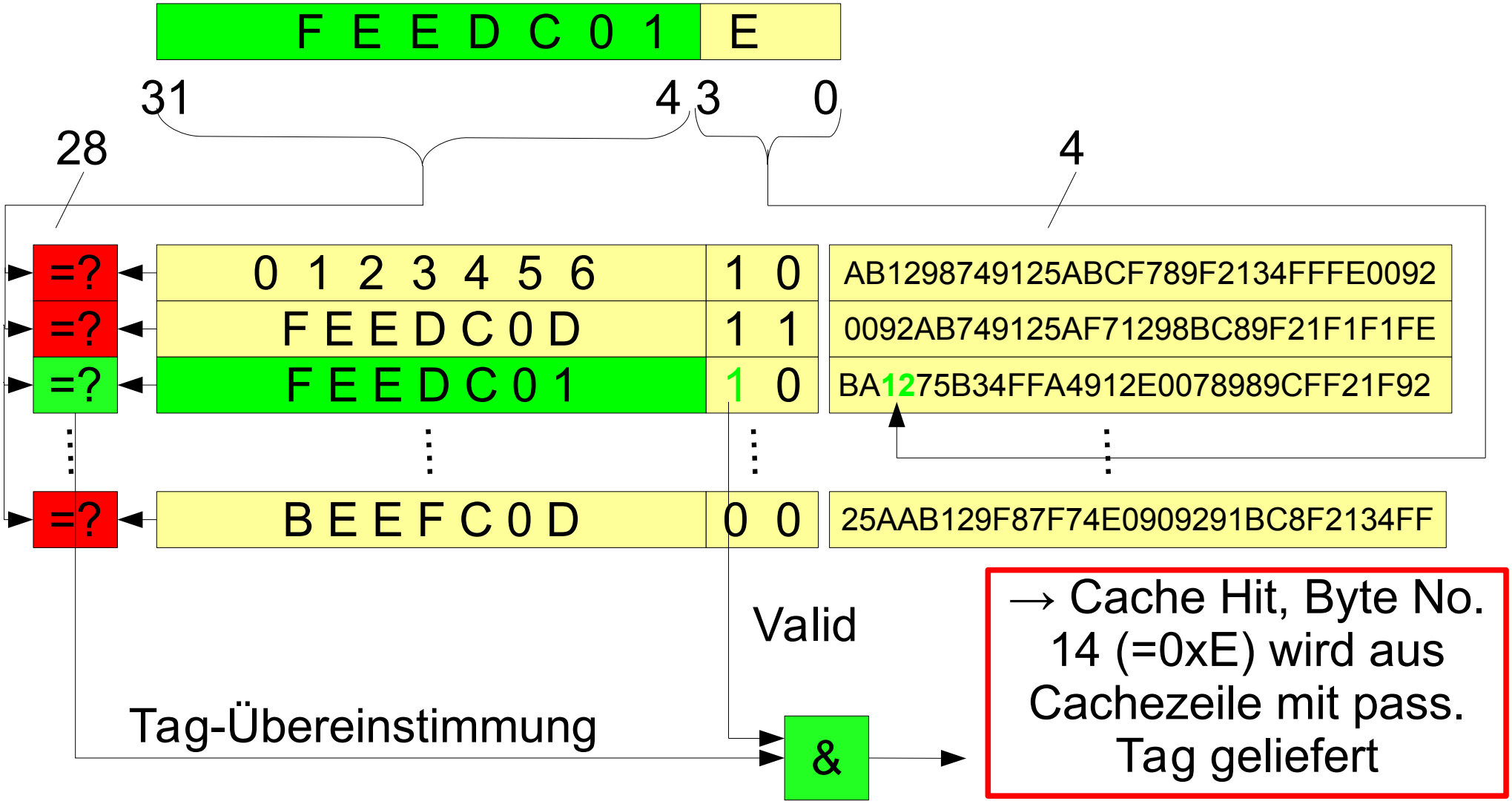
- Cache-Organisationsformen:
  1. Vollassoziativ: *fully associative*
  2. Direkt abbildend: *direct-mapped*
  3. Mehrfach assoziativ: *N-way set associative*
- Annahmen bei den folgenden Beispielen:
  - 32-bit Adressierung
  - 4K (4096) Byte Cache
  - Cache-Zeilengröße: 16 Byte
    (→ 4096 : 16 = 256 Cache-Einträge)

Vollassoziativer Cache: Aufbau

Hochschule RheinMain



Vollassoziativer Cache: Beispiel

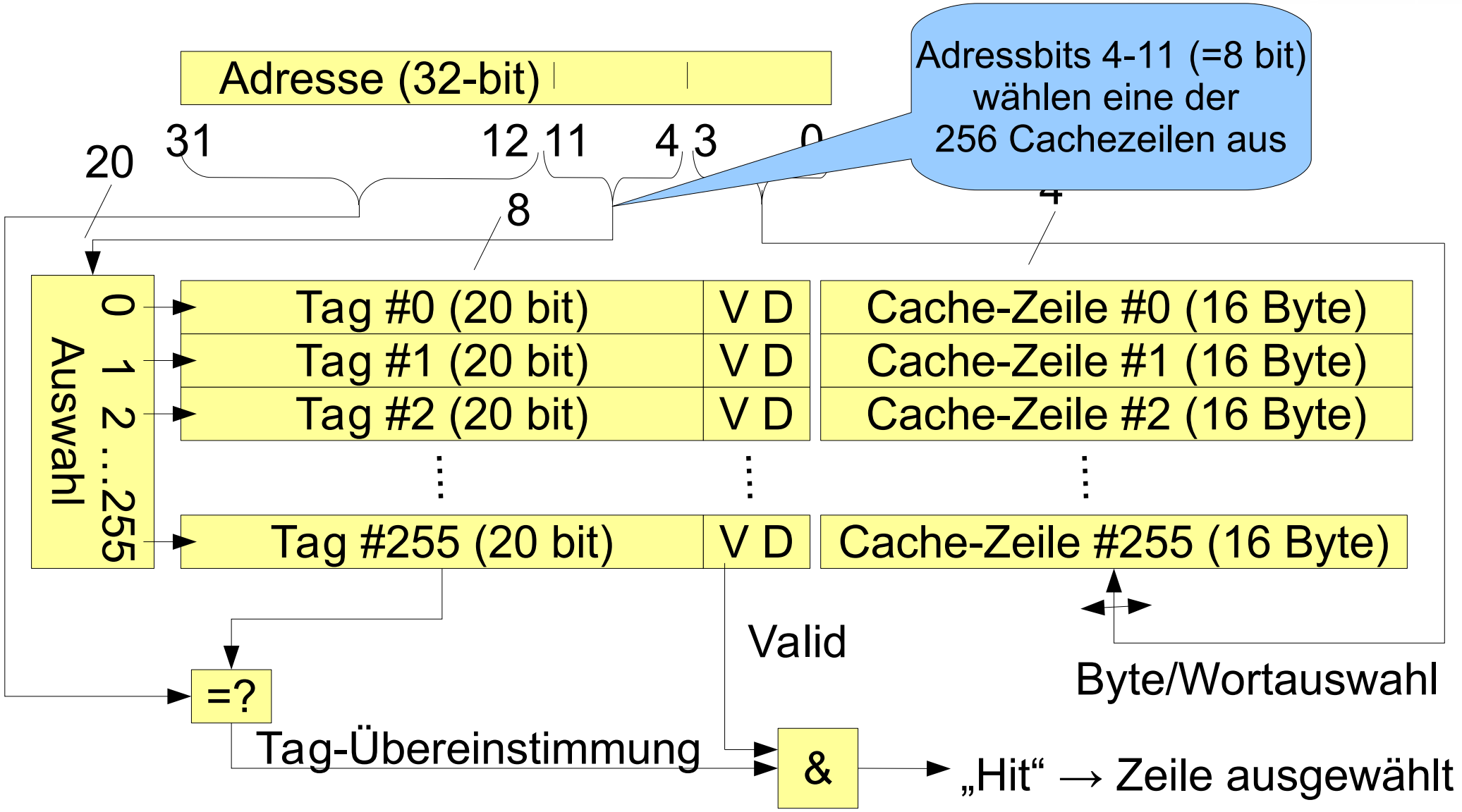


Vollassoziativer Cache

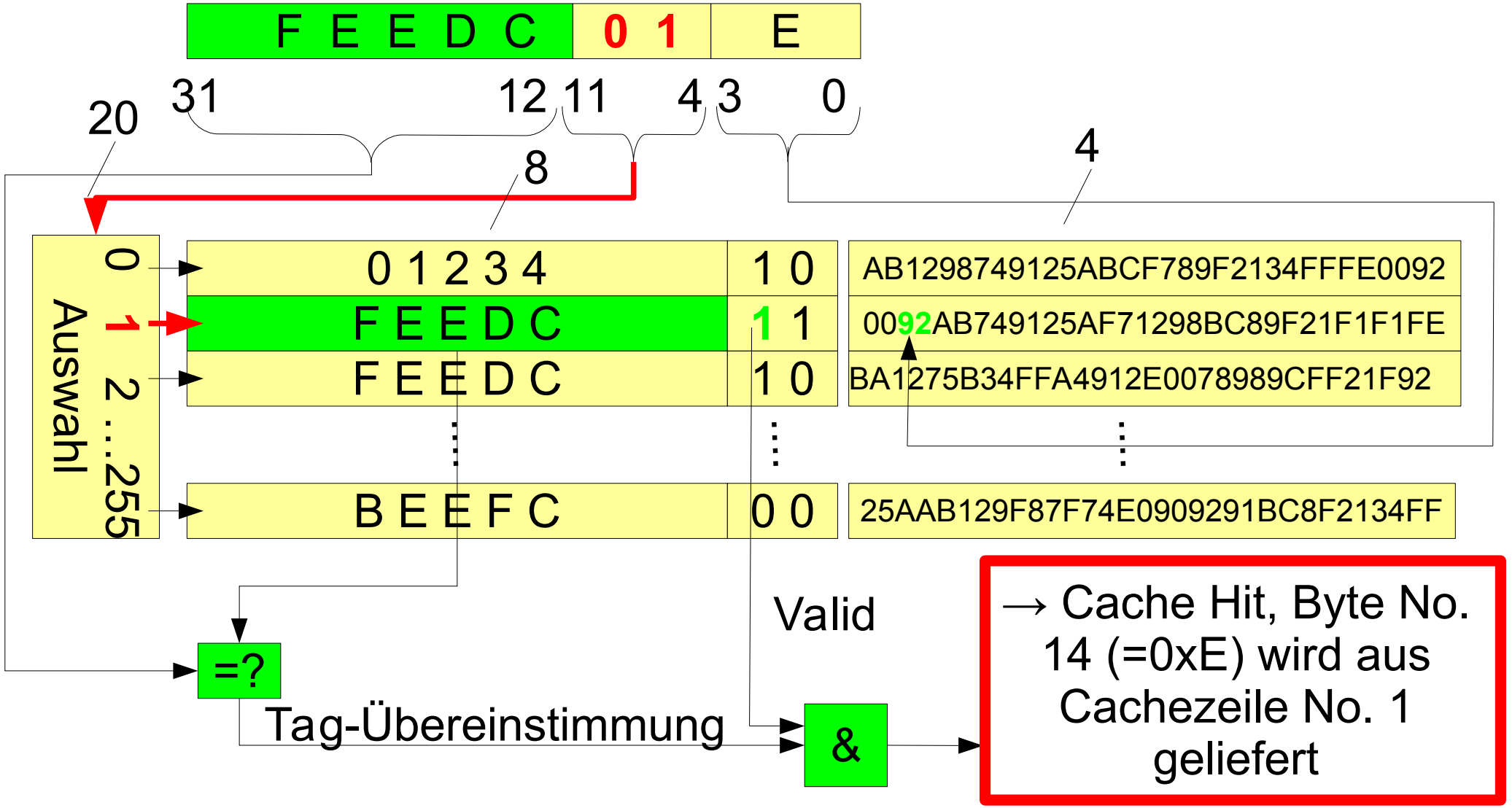
- Ein Objekt kann in eine beliebige Cache-Zeile kopiert werden
  → Freie Auswahl eines freien / zu verdrängenden Eintrags
- Identifikation der Zeile ausschließlich anhand des Tags
  → Konsequenzen:
  - Tags **aller** Zeilen müssen mit Adresse verglichen werden
  - Vergleich muss **gleichzeitig** auf allen Zeilen erfolgen
  - Für jede Zeile wird ein eigener Vergleicher benötigt
  - Jedes Tag darf maximal ein Mal vorkommen
- Erreicht höchste Trefferquote (wg. Eintrags-Wahlfreiheit)
- Große Anzahl an Vergleichern (Im Beispiel: 256 für einen 4K Cache) → sehr hoher Hardwareaufwand
- Beispiel:
  - TLB-Cache des MIPS R3000 / R4000: Vollassoziativer Cache mit 64 / 128 Einträgen

Direkt abbildender Cache: Aufbau

Hochschule RheinMain



Direkt abbildender Cache: Beispiel



Direkt abbildender Cache

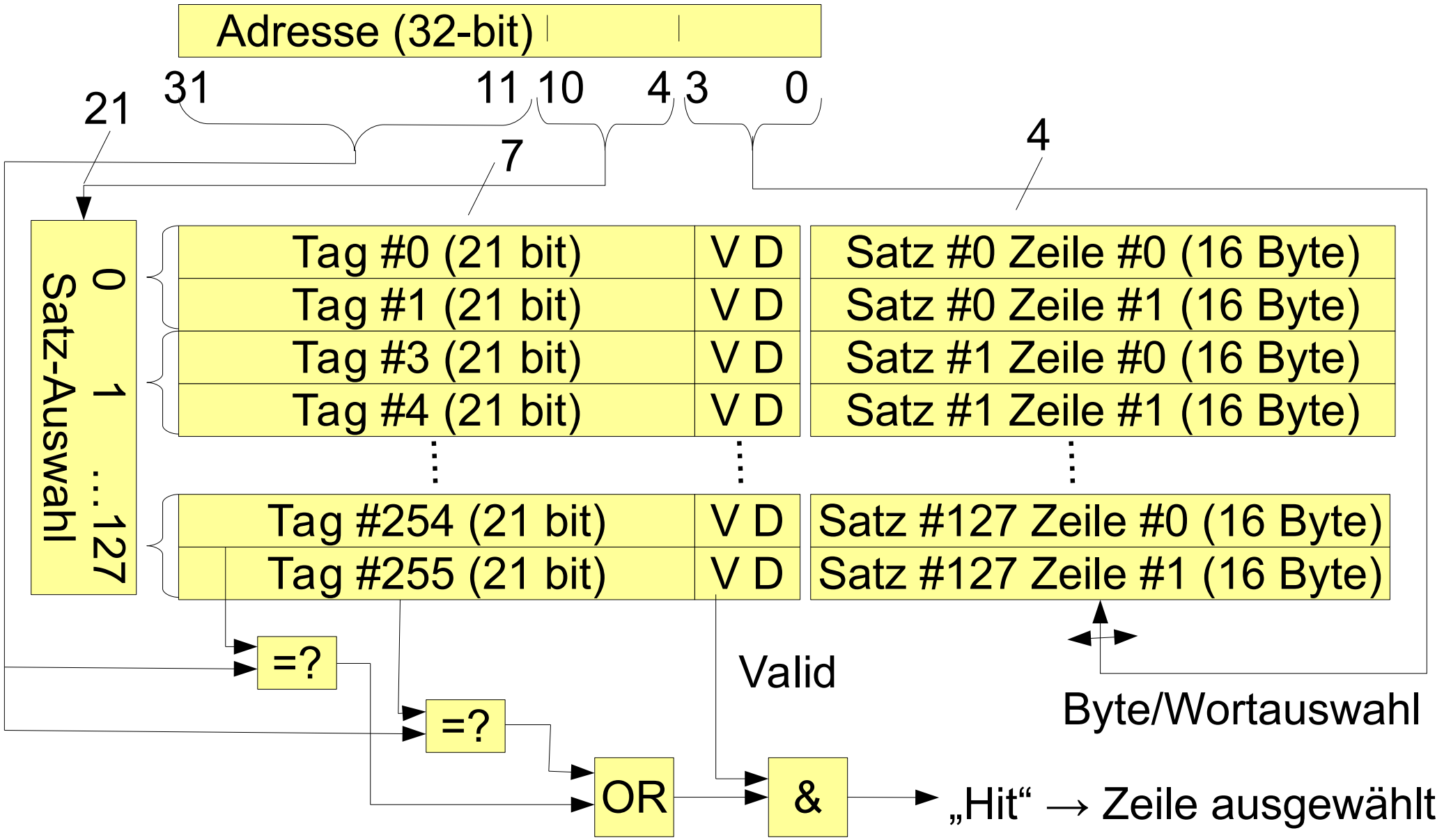
- Ein Teil der Adresse (im Beispiel: Bits 4 bis 11) wählt die Cachezeile aus
- Eindeutige Zuordnung ohne Wahlfreiheit, keine alternativen Verdrängungsstrategien möglich (aber auch keine nötig)
- Tag dient allein zur Hit/Miss Entscheidung
- **Konsequenzen**:
  - (+) Einfacher Aufbau (nur ein Vergleicher erforderlich)
  - (-) Schlechte Trefferquote
- **Beispiel**:
  - Ein Programm arbeitet in einer Schleife mit zwei Objekten, die in verschiedenen Cache-Zeilen liegen, deren Adressen sich aber in Bits 4 bis 11 nicht unterscheiden → Objekte verdrängen sich permanent gegenseitig (sog. „*Cache Trashing*“)

Mehrfach assoziativer Cache

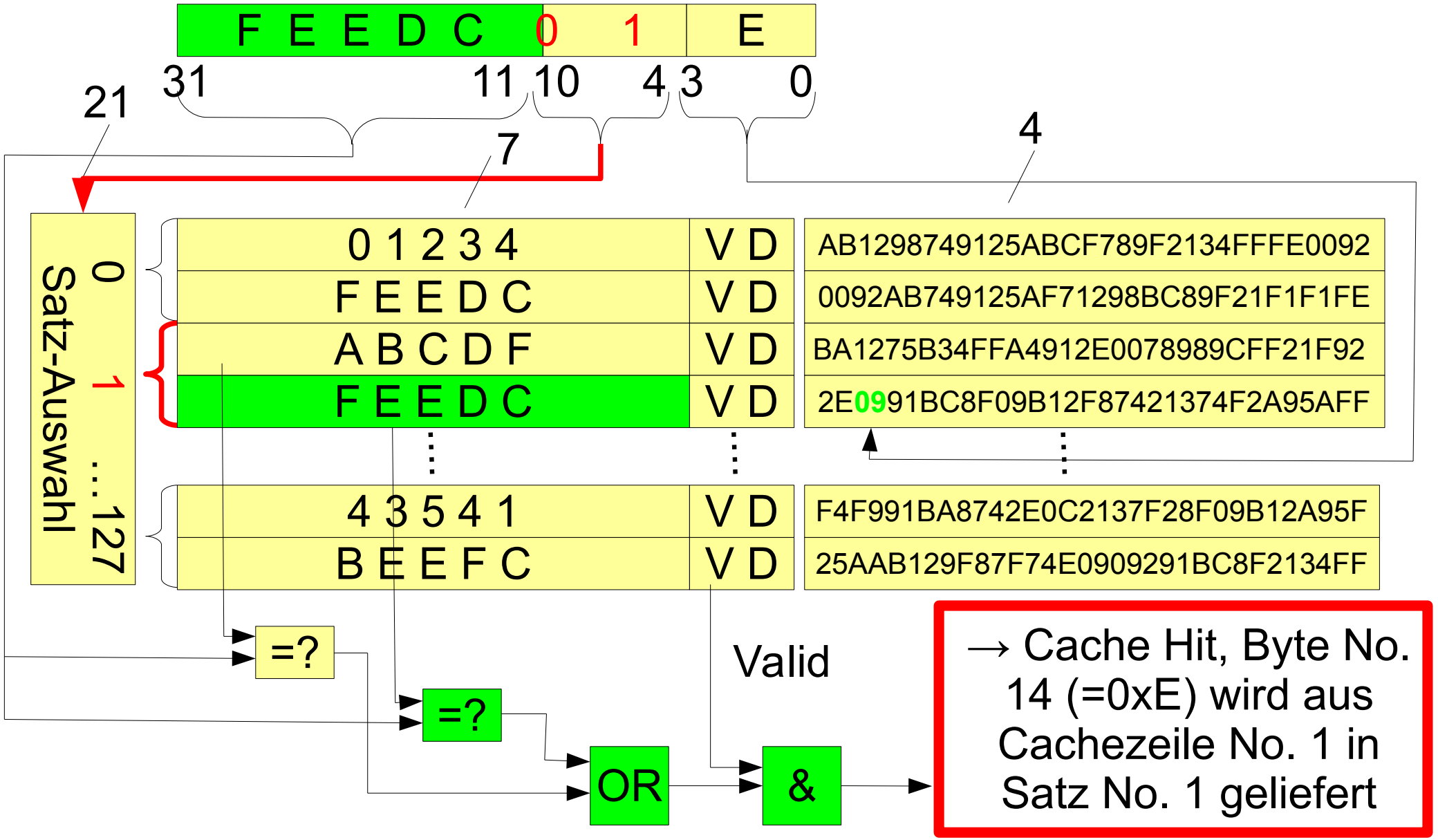
- Vollassoziativer Cache: Hohe Trefferquote, aber aufwändig
- Direkt abbildender Cache: Geringer Aufwand, aber schlechte Trefferquote (neigt zu „thrashing")
- Kompromiss: Mehrfach assoziativer Cache(*)
  - Zusammenfassen von je N (N = 2, 4, 8, ...) Cache-Zeilen zu einem „Satz" (engl. „s*et*")
  - Ein Teil der Adresse dient als Satznummer
  - Innerhalb eines Satzes gibt es N mögliche Cache-Zeilen („Wege", engl. „*ways*"), die anhand ihres Tags unterschieden werden
- ➔ Für die Auswahl eines freien bzw. zu verdrängenden Cache-Eintrags stehen N Alternativen zur Verfügung
- Verdrängungsstrategien können –wenn auch eingeschränkt– umgesetzt werden

(*) engl. *N-way set associative cache*

z.B. zweifach assoziativer Cache: Aufbau



zweifach assoziativer Cache: Beispiel



Mehrfach assoziativer Cache: Fazit

- Deutliche Verbesserung der Trefferquote gegenüber direkt abbildendem Cache
- N Wege → N Vergleicher werden benötigt
- Für N = 1 „degeneriert" er zum direkt abbildenden Cache
- Für N = <Anzahl der Cache-Zeilen> „degeneriert" er zum vollassoziativen Cache
- Für Zwischenwerte von N: guter Kompromiss zwischen Aufwand und Trefferquote
- Heute der am meisten verwendete Cache
- (s.o.) Working Set üblicher Programme besteht aus > 5-6 Regionen
- N sollte >= Anzahl der Regionen sein (sonst → Thrashing)

Mehrfach assoziativer Cache: Beispiel

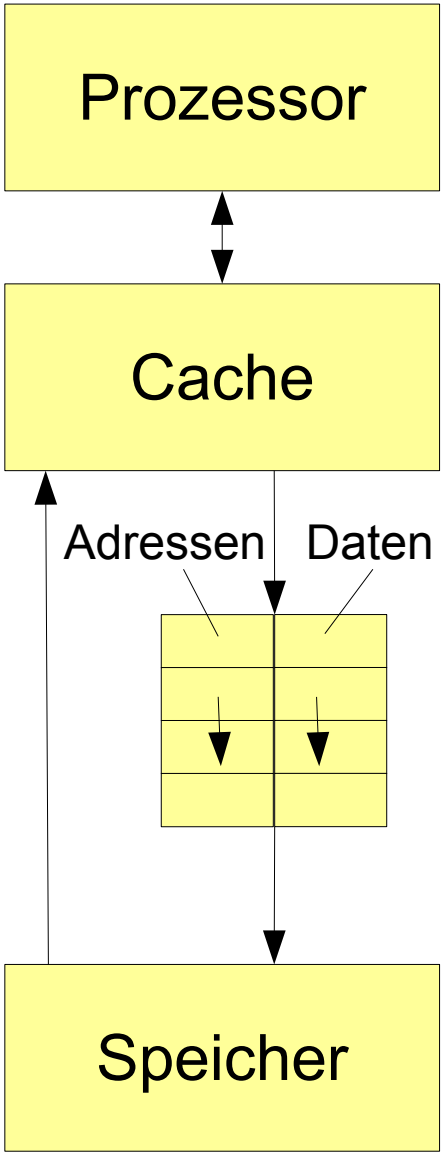
- Pentium 4:
  - L1 Datencache: 4-fach assoziativ (64 Byte Zeilengröße)
  - L1 Befehlscache: 8-fach assoziativ
  - L2 Cache: 8-fach assoziativ (64 Byte Zeilengröße)

Fallstricke

- Potenzielle Probleme im Zusammenhang mit Caches
  - „zerklüfteter“ Working Set oder ungünstige Adresslage von Variablen kann zu *Thrashing* führen → drastischer Performance-Einbruch
  - **Multitasking**: Prozesswechsel bedeutet i.d.R. auch kompletten Wechsel des Working Set
  - ➔ Nach Prozesswechsel ist der Prozessor langsamer (u.U. bis Faktor 30!)
  - DMA und Schreibzugriffe auf Codespeicher: evtl. explizites *flush* & *invalidate* erforderlich (s.o.)
  - **Multicore:** Vielfach gemeinsamer L2/L3 Cache: → gegenseitiges „ausbremsen“ der Cores, wenn auf verschiedenen Working sets gearbeitet wird.

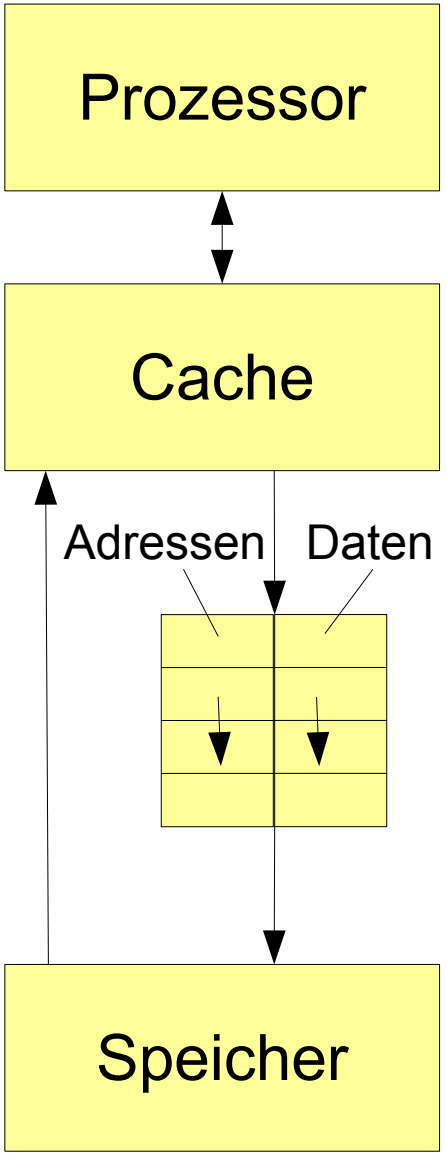
Schreib-Pufferspeicher (*Write Buffer*)

- Charakteristisches Verhalten von –z.B.– C-Programmen:
  - Etwa 10% „store”-Befehle, d.h. speichern von Daten
  - Solche Schreibzugriffe kommen häufig in schneller Folge („Bursts”) vor (z.B. wenn zu Beginn eines Unterprogramms Register gerettet werden)
- Insbesondere bei einem *write through* Cache muss der Prozessor hier auf den langsamen Hauptspeicher warten
- Abhilfe durch *Write Buffer*:
  - Ausstehende Schreibzugriffe (Adressen **und** zu schreibende Daten) werden in einen FIFO-Puffer zwischengespeichert
  - ➔ Prozessor kann sofort weiterarbeiten
  - ➔ Zwischgespeicherte Speicherzugriffe werden parallel dazu abgearbeitet



Schreib-Pufferspeicher (*Write Buffer*)

- Write Buffer finden sich z.B. bei ARM, PowerPC und MIPS-Prozessoren
- Potenzielles Problem: Lesezugriffe können Schreibzugriffe „überholen“
- z.B. bei Ein-/Ausgabe:
  - Gerät löst Interrupt aus, obwohl der bereits (per Schreibzugriff) abgeschaltet wurde
- Lösungswege:
  - **Software**: Puffer explizit „flushen“ (spezieller Maschinenbefehl)
  - **Hardware**: Jeder Lesezugriff wartet, bis der Puffer leer ist



10) Dateisysteme

-Dateistrukturen:

1) Bytefolge: Datei als unstrukturierte Folge von Bytes. BS kennt den Inhalt nicht
2) Satz-Dateistruktur: Datei als Folge von Sätzen fester Länge. BS kennt Satzlänge
Lese/Schreib/Änderungsoperationen nur auf ganze Sätze anwendbar. Satznummern
3) Baum von Sätzen: Baum von Sätzen mit Schlüsselfeld. BS sieht Satzlänge
Operationen: - Satz bei gegebenem Schlüsselfeld suchen
- **Ordungserhaltendes** Hinzufügen eines Satzes

-Dateizugriff: Sequenziell: - Verarbeitung von Anfang bis Ende
- Zurückspulen auf Dateianfang (Abstraktion Magnetband)
Direkt : - Bytes/Sätze können beliebig gelesen/geschrieben werden
-alternativ kann die Position (z.B. seek()) gesetzt werden

-Dateiattribute: Typ, Zugriffsrechte, # Referenzen, Größe, Datum (Zugriff, Erstellung).
-möglich Attribute: Schutz, Passwort, Eigentümer, Urheber, Hidden,Readonly

-memory-mapped files: -Memory Mapping von Dateien bezeichnet das Ein- & Ausblenden von Dateien in den Adressraum eines Prozesses
- Vorteil: Dateizugriff mit normalen Befehlen (kein read/write)

- Verzeichnisse (Strukturierung von Dateien. Verzeichniss selbt besteht aus Dateien)
- Verzeichnissysteme mit einer Ebene (1 Verzeichnis pro Laufwerk oder Benutzer)
- hierarchisch: bestehend aus Verzeichnis, Datei, Symbolic Link (azyklischer Graph)

- Montierbare Verzeichnisbäume: UNIX mount / umount, Windows Laufwerksbuchstaben

- Festplatte: -5 bis 10 k RPM, n * 100 Mbit/s, Mittlere Positionierungszeit: < ca. 6 ms
- Sektoren, Spuren, Lücken (auch gaps oder Servo tracks genannt)
- LBA = (C * Nheads + H) * Nblocks_per_track + S (heute wegabstrahiert)

-Halbleiterspeicher (NAND-Flash, Speicherung mithilfe von Transistor-Gates)

-Platzverwaltung:

-Partitionierung: i.d.R wird der gesamt Platz auf mehrere Partitionen verteilt
-MBR (master boot record) enthält ausführbaren Code, wird beim Booten gestartet
-logische Laufwerke: dynamisch veränderbare Partitionierung, die sich auch über mehrere physische Datenträger hinweg erstrecken kann.
-Linux LVM: physical volumes werden zu volume groups zusammengefaßt
Auf einer Laufwerksgruppe können logische Laufwerke eingerichtet (entspricht Partitionierung) und mit einem Dateisystem versehen werden. Im laufenden Betrieb kann die Kapazität der Laufwerksgruppe durch Hinzufügen weiterer physische Volumes vergrößert werden. können Daten von alten Laufwerken auf neue verlagert und die alten Laufwerke außer Betrieb genommen werden. kann logischen Laufwerken mehr Speicherplatz zugeordnet werden oder Speicherplatz entzogen werden. LVM unterstützt „Filesystem Snapshots“ Beim Anlegen eines Snapshots wird ein neues logisches Laufwerk angelegt, das den momentanen Zustand seines zugehörigen Ursprungs-Laufwerks enthält (eingefrorene Sicht, keine Kopie)
→ Ermöglicht konsistente Backups über Snapshot-Laufwerk trotz weiterlaufenden Betriebs auf dem ursprünglichen Laufwerk

RAID (Redundant Array of Independent [Inexpensive] Disks)

-Realisierungen: -Hardware-RAID (spezieller Festplatten Controller)
-Software-RAID (BS verwaltet Platten als RAID)

-Ziele: -Erhöhung der Datensicherheit
-Austausch defekter Platten im laufenden Betrieb ohne Unterbrechung („hot-standby“-Platte)
-Verteilung der Daten durch RAID level (0-6) definiert

-RAID0 – „stripping“

RAID0 - „striping“

Hochschule Rh

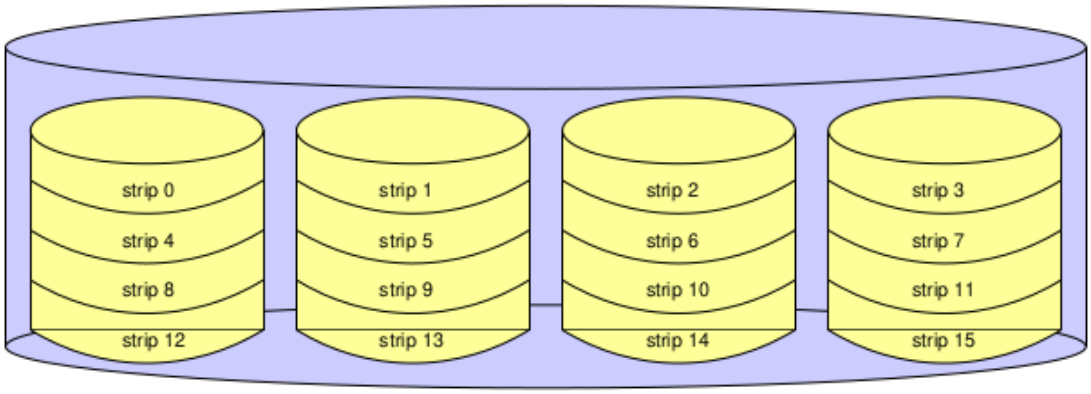
RAID-Platte wird in „Streifen“ mit *k* Blöcken eingeteilt

Streifen werden reihum auf den angeschlossenen Platten abgelegt.

→ **keine Redundanz**, damit keine höhere Fehlertoleranz

→ **schneller Zugriff** besonders bei großen Dateien, da Platten parallel arbeiten können

RAID-Kapazität: Summe der Plattenkapazitäten



RAID1 - „mirrorring“

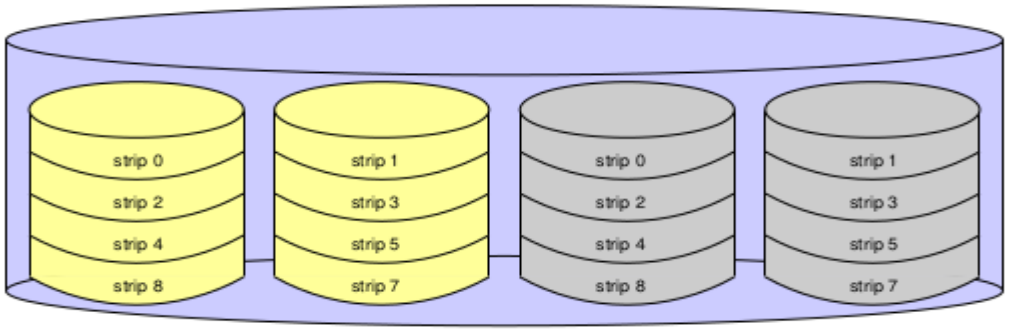
Hochschule RheinM

Zu jeder Platte gibt es eine Spiegelplatte gleichen Inhalts

Fehlertoleranz: Wenn eine Platte ausfällt, kann andere sofort einspringen (übernimmt Controller automatisch)

Schreiben: etwas langsamer; **Lesen**: schneller durch Parallelzugriff auf beide zuständigen Platten

RAID-Kapazität: Hälfte der Summe der Plattenkapazitäten



RAID5 - parity

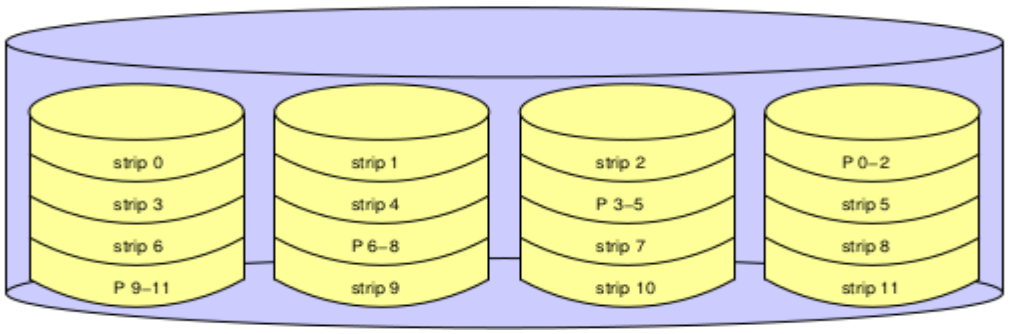
Hochschule RheinM

Paritätsinformation (XOR) auf alle Platten verteilt
Beispiel: P 0-2 enthält XOR-Verknüpfung über die Streifen 0, 1, 2
XOR Verknüpfung ist „selbstinvers“:

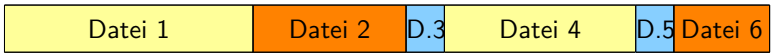
Wenn gilt:	$P = A \oplus B \oplus C$
dann ist:	$A = B \oplus C \oplus P$
und:	$B = A \oplus C \oplus P$
und:	$C = A \oplus B \oplus P$

+ Fehlertolerant bei guter Kapazitätsnutzung
+ Leseoperationen schnell
- Schreiben aufwändiger

→ Solange maximal eine (beliebige) Platte ausfällt, kann ihr Inhalt aus den Übrigen (im lfd. Betrieb) rekonstruiert werden (wieder per XOR)



Kontinuierliche Allokation



Jeder Datei wird eine Menge zusammenhängender Blöcke zugeordnet (beim **Anlegen reserviert**).

Vorteile:

- Einfach zu implementieren (nur die Adresse des ersten Blocks ist zu speichern).
- Sehr gute Performance beim Lesen und Schreiben der Datei (minimale Kopfbewegungen).

Nachteile:

- Maximalgröße der Datei muss zum Erzeugungszeitpunkt bekannt sein, Wachsen (Append) ist nicht möglich
- Externe Fragmentierung durch Löschen / Überschreiben
- Verdichtung extrem aufwändig / langwierig

Einsatzgebiete:

- Echtzeit-Anwendungen (zusammenhängende Dateien (*contiguous files*) für kalkulierbare Zugriffszeiten)
- Write-Once Dateisysteme (CD/DVD, Logs, Backups, Versionierung).

Notizen

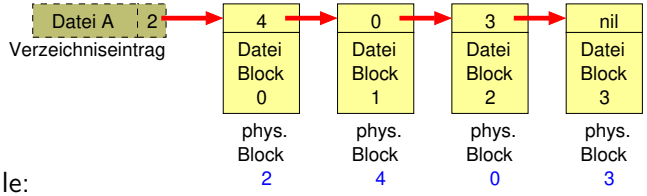
Allokation mittels verketteter Liste

Idee: Speicherblöcke einer Datei werden durch Verweise miteinander verkettet.

Jeder Block hat einen **Verweis auf Nachfolger-Block**

Verweis z.B. direkt am Beginn jedes Speicherblocks

Verzeichniseintrag verweist auf ersten Block der Datei



Vorteile:

- Keine Externe Fragmentierung

Nachteile:

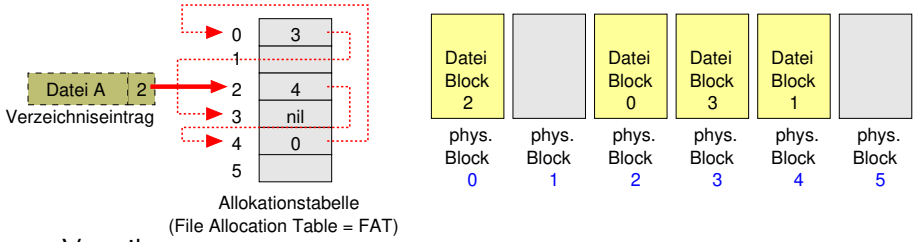
- Wahlfreier Zugriff ist seeehr langsam.
- Es steht nicht der gesamte Datenblock für Daten zur Verfügung.

Notizen

Allokation mittels verketteter Liste und Index

Hochschule RheinMain

Speicherung der Verkettungsinformation in einer separaten Tabelle (Index oder **F**ile **A**llocation **T**able = FAT).



Vorteile:

- Gesamter Datenblock steht für Daten zur Verfügung.
- Akzeptable Performance bei direktem Zugriff, da der Index im Arbeitsspeicher gehalten werden kann.

Nachteile:

- Die gesamte Tabelle muss im Arbeitsspeicher gehalten werden. Kann bei großer Platte sehr speicherplatzaufwändig sein.

Beispiel: MS-DOS FAT File System

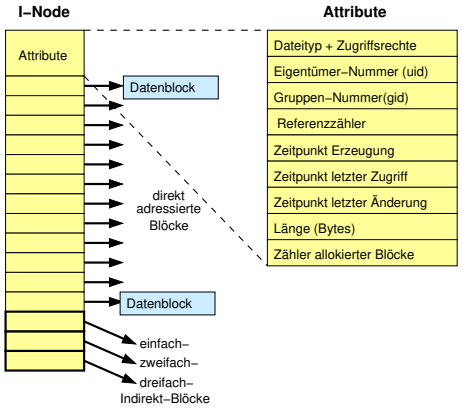
Notizen

Allokation mittels Index Nodes (1)

Hochschule RheinMain

Definition:I-Node

Ein **I-Node** (UNIX: *inode*) oder Index Node ist ein Dateikontrollblock



enthält neben Attributen der Datei eine Tabelle mit Adressen von zugeordneten Plattenblöcken.

Ursprung Beispiel: BSD UNIX Fast File System (ufs)

Notizen

Allokation mittels Index Nodes (2)

Hochschule RheinMain

Logische Blocknummern einer Datei ...

...werden fortlaufend vergeben,
beginnend bei den direkt adressierten Blöcken.

Einige wenige (~12) Blockadressen sind im Inode selbst gespeichet

→ Nach Öffnen einer Datei und damit verbundenem Einlagern des Inodes in den Hauptspeicher stehen diese Adressen sofort zur Verfügung.
→ Schneller Zugriff bei kleinen Dateien.

Für größere bis sehr große Dateien werden nach und nach einfach, zweifach und dreifach indirekte Blöcke zur Speicherung verwendet.

→ Zugriffsgeschwindigkeit sinkt bei größeren Dateien.

Beispiel: Linux ext2, ext3

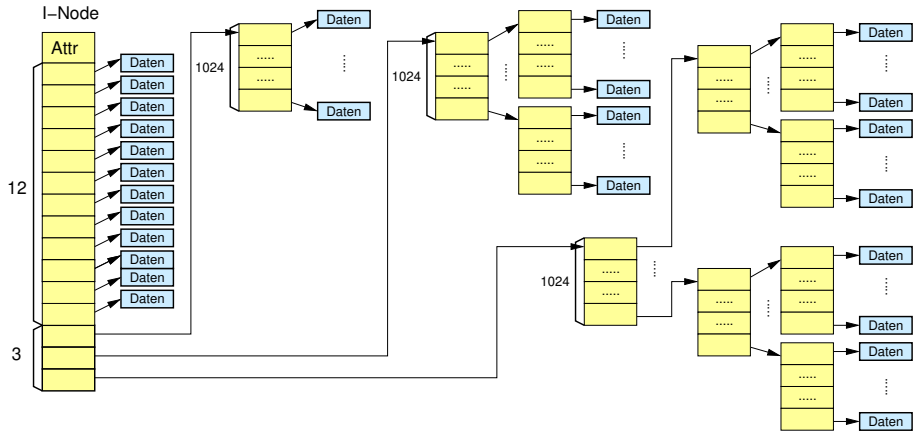
Blockgröße: 4KB
12 direkt adressierte Blöcke → 48KB
Es sind 1-fach, 2-fach und 3-fach indirekte Blöcke vorgesehen
Indirekte Blöcke (4KB) speichern bis zu 1024 ($=2^{10}$) weitere Verweise
⇒ Maximale Dateigröße:
$(12 + 2^{10} + 2^{30} + 2^{30}) \cdot 4KB = 48KB + 4MB + 4GB + 4TB \approx 4TB$

Notizen

Allokation mittels Index Nodes (3)

Hochschule RheinMain

Nutzung von Indirekt-Blöcken



Notizen

Implementierung von Verzeichnissen

Hochschule RheinMain

Hauptaufgabe des Verzeichnissystems:
Abbildung der Zeichenketten-Namen von Dateien in Informationen zur Lokalisierung der zugeordneten Plattenblöcke.

Bei Pfadnamen werden die Teilnamen zwischen Separatoren schrittweise über eine Folge von Verzeichnissen umgewandelt.

Verzeichniseintrag liefert bei gegebenem Namen (Teilnamen) die Information zum Auffinden der Plattenblöcke:

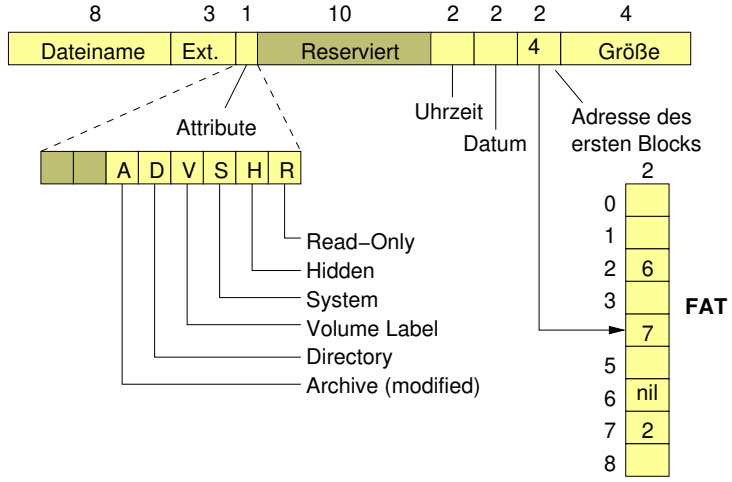
- bei kontinuierlicher Allokation: die Plattenadresse der gesamten Datei oder des Unterverzeichnisses.
- bei Allokation mit verketteter Liste mit und ohne Index: die Plattenadresse des ersten Blocks der Datei oder des Unterverzeichnisses.
- bei Allokation mit Index Nodes: die Nummer des Inodes der Datei oder des Unterverzeichnisses.

Notizen

Beispiel: MS-DOS

Hochschule RheinMain

(s.o.) Hierarchisches Verzeichnissystem, Allokation von Plattenblöcken mittels verketteter Liste und Index.
Verzeichniseintrag:



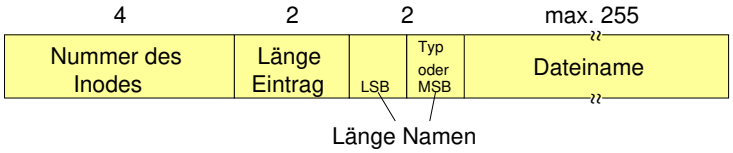
Notizen

Beispiel: UNIX (1)

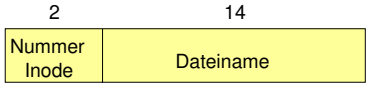
Hochschule RheinMain

(s.o.) Hierarchisches Verzeichnissystem, Allokation von Plattenblöcken mittels Index Nodes (inodes).

Verzeichniseintrag Linux (ext2:)



Verzeichniseintrag klassisches UNIX System V (s5):

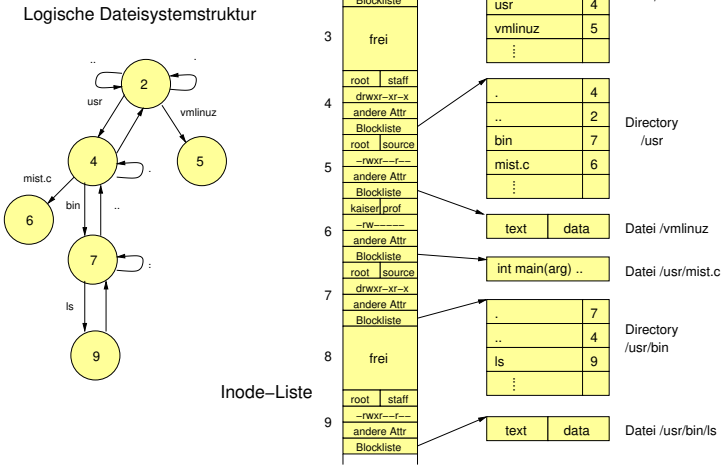


Notizen

Beispiel: UNIX (2)

Hochschule RheinMain

Prinzip der Umsetzung eines Pfadnamens



Notizen

Dateisystemstruktur

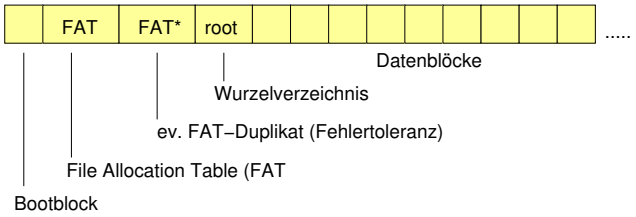
Hochschule RheinMain

Die Struktur eines Dateisystems wird beim Erzeugen auf die Blockmenge eines logischen Laufwerks (Partition) aufgeprägt.

Dienstprogramme zum Erzeugen:

- MS-DIS: `format`[5]
- UNIX: `mkfs`, (`newfs`)

Beispiel: MS-DOS



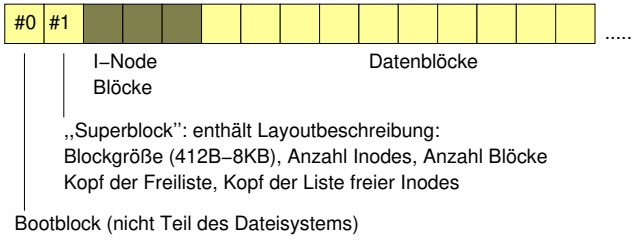
[5] nicht zu verwechseln mit dem Formatieren eines Mediums, in MS-DOS low-level Formatierung genannt

Notizen

Dateisystemstruktur (2)

Hochschule RheinMain

Beispiel: Klassisches UNIX System V (s5)



Notizen

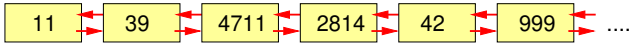
Verwaltung freier Blöcke

Hochschule RheinMain

Angewendete Methoden:

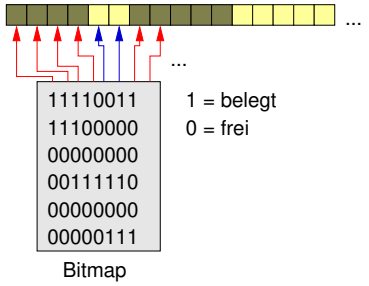
Verkettete Liste.

- Vorteil: Größere freie Bereiche einfacher erkennbar
- Beispiele: MS-DOS FAT, UNIX System V



Bitmap

- Vorteil: Größere freie Bereiche einfacher erkennbar.
- Beispiel: Linux ext2



Notizen

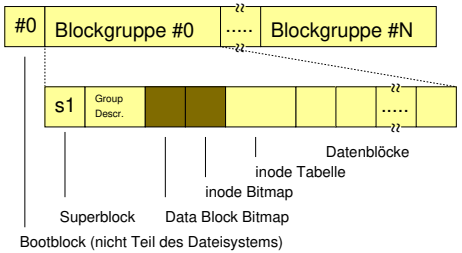
Dateisystemstruktur (3)

Hochschule RheinMain

Beispiel: Linux ext2: Besonderheiten

Einführung sogenannter **Bockgruppen**, d.h. Mengen von aufeinander folgenden Blöcken innerhalb eines Dateisystems mit jeweils eigenen Verwaltungsstrukturen.

I-Node und zugehörige Datenblöcke sollen möglichst dicht beisammen bleiben (Performance).



Redundante Kopien s1, .. des Superblocks in jeder Blockgruppe an verschiedenen Stellen (Kopfpositionen) zur Verbesserung der Verfügbarkeit der Layoutinformation auch bei Plattenfehlern.

Notizen

Dateisystemstruktur (4)

Hochschule RheinMain

Weitere Ansätze (hier nicht im Detail besprochen)

Log-basierte Dateisysteme:

- Für Halbleiter-Speichermedien konzipiert (keine seek-Zeiten)
- Schreibaufträge puffern, in regelmäßigen Zeitabständen als ganzes Segment schreiben.
- Position der inodes über in-Memory Map verwalten.
- Platte wird als „Ringpuffer“ betrieben.
- „Cleaner“ Thread gibt regelmäßig unbenutzte Blöcke frei.

Journaling-Dateisysteme:

- Erst: Geplante Operationen (idempotent) in Log schreiben
- Dann: Operationen ausführen
- Bei Absturz: Geplante, aber nicht mehr zur Ausführung gekommene Operationen nachholen.

Notizen

Quotas (Plattenplatz-Kontingentierung)

Hochschule RheinMain

Ziel:
Vermeidung der Monopolisierung von Plattenplatz durch einzelne Benutzer in Mehrbenutzersystemen.

Systemadministrator kann jedem Benutzer **Schranken** (Quotas) zuordnen für

- maximale Anzahl von eigenen **Dateien** und
- Anzahl der von diesen benutzten **Plattenblöcke**

Betriebssystem stellt sicher, dass diese Schranken nicht überschritten werden.

Beispiel: UNIX Quotas

Je Benutzer und Dateisystem werden verwaltet:

- Weiche Schranke (Soft Limit) für die Anzahl der benutzten Blöcke (kurzfristige Überschreitung möglich).
- Harte Schranke (Hard Limit) für die Anzahl der benutzten Blöcke (kann nicht überschritten werden).
- Anzahl der aktuell insgesamt zugeordneten Blöcke.
- Restanzahl von Warnungen. Diese werden bei Überschreitung des Soft Limits beim Login beschränkt oft wiederholt, danach ist kein Login mehr möglich.
- Gleiche Information für die Anzahl der benutzten Dateien (Inodes).

Notizen

Wahl der Blockgröße

Hochschule RheinMain

Fast alle Dateisysteme bilden Dateien aus Blöcken fester Länge
(Block umfasst zusammenhängende Folge von Sektoren.)
Problem: Welches ist die optimale Blockgröße?

Kandidaten aufgrund der Plattenorganisation sind:

- Sektor (512 B - 4KB)
- Spur (z.B. 256 KB)

Untersuchungen an Dateisystemen in UNIX-Systemen zeigen:

- Die meisten Dateien sind klein ($<$ 10 KB) aber mit wachsender Tendenz.
- In Hochschul-Umgebung im Mittel 1 KB [Tanenbaum]

Eine große Allokationseinheit (z.B. Spur) verschwendet daher zuviel Platz.

Beispielrechnung: Verschwendeter Platz

(Daten basieren auf realen Dateien, Quelle [Leffler et al].)

Gesamt [MB]	Overhead [%]	Organisation
775.2	0.0	nur Daten, byte-variabel lange Segmente
807.8	4.2	nur Daten, Blockgröße 512 B, int. Fragmentierung
828.7	6.9	Daten und Inodes, UNIX System V, Blockgröße 512 B
866.5	11.8	Daten und Inodes, UNIX System V, Blockgröße 1 KB
948.5	22.4	Daten und Inodes, UNIX System V, Blockgröße 2 KB
1128.3	45.6	Daten und Inodes, UNIX System V, Blockgröße 4 KB

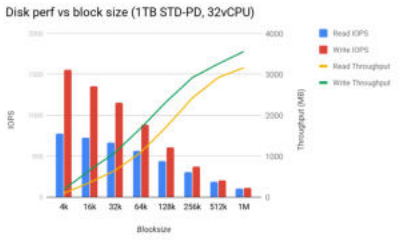
Notizen

Wahl der Blockgröße (2)

Hochschule RheinMain

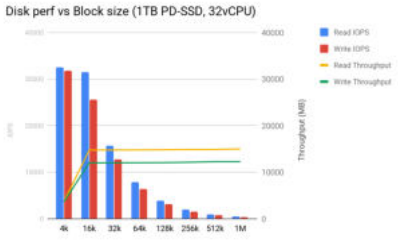
Eine kleine Allokationseinheit (z.B. Sektor) führt zu schlechter zeitlicher Performance (viele Blöcke = viele Kopfbewegungen).

Mechanische Festplatte



https://medium.com/@duhroach/the-impact-of-blocksize-on-persistent-disk-performance-7e50a85b2647

zum Vergleich: SSD
N.B.: andere Skalierung



https://medium.com/@duhroach/the-impact-of-blocksize-on-persistent-disk-performance-7e50a85b2647

Notizen

Wahl der Blockgröße (3)

Hochschule RheinMain

Kompromiss:

Wahl einer mittleren Blockgröße, z.B. 4 KB oder 8 KB.
Bei Sektorgröße 512 B entspricht ein Block von 4 KB Größe dann 8 aufeinanderfolgenden Sektoren.
Für Lesen oder Schreiben eines Blockes wird die entsprechende Folge von Sektoren als Einheit gelesen oder geschrieben.
Ab ca. 2010 für PC-Systeme und Notebooks vermehrt Festplatten mit 4 KB-Sektorgröße (*Advanced Format*)

Ca. 9% Kapazitätsgewinn (da weniger Gaps)
Kompatibilitätsprobleme (Lösung durch Emulation) können Performance-Nachteile haben

Notizen

Wahl der Blockgröße (3)

Hochschule RheinMain

Tricks ...

Beispiel: BSD UNIX Fast File System (heute ähnlich auch bei ext2)

Einführung **zweier** Blockgrößen, genannt **Block** und **Fragment** als Teil eines Blocks (typ. heute 8 KB / 1 KB).

Eine Datei besteht aus ganzen Blöcken (falls nötig) sowie ein oder mehreren Fragmenten am Ende der Datei.

→ Transfer großer Dateien wird effizient.

→ Speicherplatz für kleine Dateien wird gut genutzt.
Empirisch wurde ein ähnlicher Overhead für ein 4KB/1KB BSD File System beobachtet wie für das 1 KB System V File System.

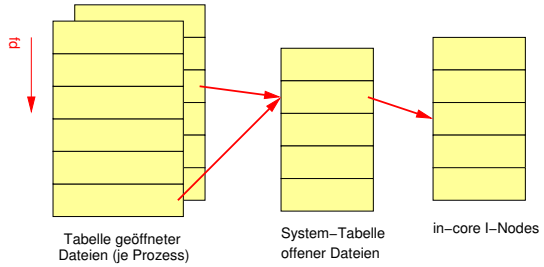
Notizen

Repräsentierung im Hauptspeicher

Hochschule RheinMain

Bisher wurde die Repräsentierung von Dateien und Verzeichnissen auf dem Hintergrundspeicher betrachtet.

Geöffnete Dateien besitzen eine Repräsentierung im Arbeitsspeicher



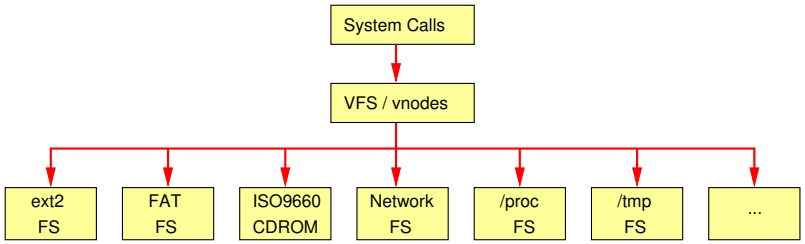
N.B.: Die System-Tabelle enthält insbesondere die Datei-Position jeder geöffneten Datei. Da geöffnete Dateien bei fork() vererbt werden, ist die einmalige Verarbeitung des Dateiinhalts durch Eltern- und Kindprozess oder durch mehrere Kindprozesse möglich. Erneutes Öffnen derselben Datei resultiert dagegen in einem neuen Eintrag mit unabhängigem Positionszeiger.

Notizen

Virtuelles Dateisystem

Hochschule RheinMain

Innerhalb des BS-Kerns wurde neue Schnittstelle des Dateisystems eingezogen, das sogenannte vnode-Interface (virtual inode) des VFS (Virtual File System).
Das vnode-Interface umfasst generische Operationen zum Umgang mit Dateien und Verzeichnissen bzw. Dateisystemen als ganzes.
Die virtuelle Schnittstelle wird für jeden Dateisystemtyp implementiert.
Das Interface ist auch auf Pseudo-Dateisysteme anwendbar, wie etwa das `/proc`-Prozessdateisystem (jedem aktiven Adressraum entspricht eine Datei) oder RAM-Disk.



Notizen

Zuverlässigkeit des Dateisystems

Hochschule RheinMain

Hauptgesichtspunkte:

Behandlung fehlerhafter Blöcke.

Dateisystemkonsistenz.

Erzeugen und Verwalten von Backups.

Notizen

Behandlung fehlerhafter Blöcke

Hochschule RheinMain

Sowohl Festplatten als auch Flash-Speicher haben i.d.R. von Anfang an **fehlerhafte Blöcke** (z.B. aufgrund ungleichmäßiger Magnetisierung der Oberflächen, Toleranzen in der Chip-Fertigung, etc).
Bei Festplatten werden fehlerhafte Sektoren beim Formatieren des Mediums (Aufbau der Sektoren) festgestellt. (Im PC-Umfeld wird das Formatieren auch low-level-Formatierung genannt).
Bei Flash-Speichern kommen aufgrund des Verschleißes im lfd. Betrieb weitere defekte Blöcke hinzu.
Verzeichnis der fehlerhaften Sektoren wird **Media Defect List** genannt.
Hardware-Lösung (heute üblich):

- Media Defect List wird vom Gerätecontroller selbst geführt.
- Jedem defekten Block wird ein Ersatzblock (i.d.R. aus einem dafür reservierten Bereich) zugeordnet, der statt des defekten Blocks benutzt wird.
- Bei Festplatten Problem: evtl. unvorhersehbare Kopfbewegungen.
- Bei Flash-Speichern kann der reservierte Bereich irgendwann erschöpft sein.

Software-Lösung (heute eher unüblich):

- Es wird eine Datei konstruiert, die nie gelesen oder geschrieben wird und der alle defekten Blöcke zugeordnet werden.

Notizen

Konsistenz des Dateisystems

Hochschule RheinMain

Definition: Dateisystem-Konsistenz

Konsistenz eines Dateisystems meint Korrektheit der inneren Struktur des Dateisystems.

d.h. aller mit der Aufprägung der Dateisystemstruktur auf die Blockmenge verbundenen Informationen (Meta-Information des Dateisystems, UNIX: z.B. Superblock, Freiliste oder Bitmap).

Beispiel einer Konsistenzregel:

- Jeder Block ist entweder Bestandteil genau einer Datei oder eines Verzeichnisses, oder er ist genau einmal als freier Block bekannt.

Verletzung der Konsistenz:

- I.d.R. durch Systemzusammenbruch (z.B. aufgrund eines Stromausfalls) vor Abspeicherung aller modifizierten Blöcke eines Dateisystems.

Überprüfung der Konsistenz:

- Betriebssysteme besitzen Hilfsprogramme zur Überprüfung und evtl. Wiederherstellung der Konsistenz bei eventuell auftretendem Datenverlust.

Notizen

Beispiel: UNIX

Hochschule RheinMain

UNIX war traditionell schwach in Bezug auf Sicherstellung der Konsistenz von Dateisystemen:

Dateisystem (z.B. ufs) wird **nicht in atomaren Schritten** von einem konsistenten Zustand in einen neuen konsistenten Zustand überführt. Eine solche Veränderung verlangt i.d.R. mehrere Schreibzugriffe.

Modifizierte Datenblöcke bleiben im Pufferspeicher (Block Buffer Cache) und werden durch einen Dämonprozess spätestens nach 30 sec zurückgeschrieben.

Modifizierte Blöcke mit Meta-Informationen werden zur Verringerung der Gefahr der Inkonsistenz sofort zurückgeschrieben.

System Call `sync` existiert zur Einleitung eines sofortigen Zurückschreibens aller veränderten Blöcke (*forced write*).

Notizen

Beispiel: UNIX (2)

Hochschule RheinMain

Durch Einführung von Journaling-Dateisystemen[6] hat sich die Situation deutlich gebessert.

- Meta-Informationen des Dateisystems werden vorab per Write-Ahead-Logging (analog Datenbanken) gespeichert.
- Konsistenz ist gewährleistet, z.B. bei Systemzusammenbruch.
- Dennoch können plötzliche Stromausfälle während laufender physischer Schreibvorgänge Schäden verursachen.
- Gilt für Festplatten wie für Flash-Speicher.
- Deshalb: Medien vor dem Entfernen immer erst „unmounten“.

Zur Verbesserung der Verfügbarkeit von Daten im Falle von Plattenfehlern werden z.B. eingesetzt:

- Spiegelplattenbetrieb (RAID1 - Disk mirroring, vgl. 10.3.2)
- Paritätsinformation speichern (RAID5, vgl. 10.3.2)

[6] `ext3`, `ext4`, ReiserFS, Btrfs, XFS, ..., vgl 10.3.5

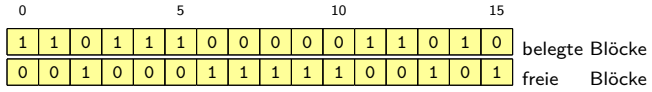
Notizen

Überprüfung und Reparatur

Hochschule RheinMain

Dienstprogramme zur Überprüfung / Reparatur der Konsistenz eines **nicht in Benutzung** befindlichen Dateisystems **bei möglichem Datenverlust**: `fsck`, z.T. auch `ncheck`
`fsck` (*file system check*) führt Konsistenzüberprüfungen durch:
Blocküberprüfung:

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



- erste Tabelle: wie oft tritt jeder **Block in einer Datei** auf?
  - ★ alle inodes lesen.
  - ★ für jeden verwendeten Block Zähler in erster Tabelle inkrementieren.
- zweite Tabelle: **freie Blöcke**
  - ★ Für Blöcke in der Liste / Bitmap der freien Blöcke Zähler in zweiter Tabelle inkrementieren.

Konsistenz: Für jeden Block muss der Zählerstand aus Tab 1 und Tab 2 zusammen „1“ sein.

Notizen

Überprüfung und Reparatur (2)

Hochschule RheinMain

Fehlender Block: Block 4 ist weder belegt noch frei?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	0	0	0	0	0	1	1	0	1	0	belegte Blöcke
0	0	1	0	0	0	1	1	1	1	1	0	0	1	0	1	freie Blöcke

→ Maßnahme: Block zu freien Blöcken hinzunehmen

Doppelter Block in Freiliste (Block 8)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	1	1	0	0	0	0	0	1	1	0	1	0	belegte Blöcke
0	0	1	0	0	0	1	1	2	1	1	0	0	1	0	1	freie Blöcke

→ Maßnahme: Freiliste neu aufbauen

Doppelter belegter Block (Block 11)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	1	1	0	0	0	0	0	2	1	0	1	0	belegte Blöcke
0	0	1	0	0	0	1	1	1	1	1	0	0	1	0	1	freie Blöcke

→ Maßnahme: Block kopieren, Kopie-Block in eine der beiden betroffenen Dateien statt Block 11 einbauen

Notizen

Überprüfung und Reparatur (3)

Hochschule RheinMain

Darüber hinaus **Überprüfung der Verzeichniseinträge**:

- Vergleiche Anzahl aller Verzeichniseinträge mit Verweis auf einen inode mit dem darin gespeicherten Referenzzähler.

→ Maßnahme: ggf. inode-Referenzzähler der durch Zählung festgestellten Zahl von Referenzen anpassen.
(N.B.: Es kann „beliebig viele“ (> 0) Referenzen auf einen inode geben (aufgrund harter Links!))

Problem: Für große Platten kann ein `fsck`-Lauf sehr lange (Stunden!) dauern. In dieser Zeit ist das System möglicherweise nicht verfügbar (Kosten!)

Notizen

Performance des Dateisystems

Hochschule RheinMain

Maßnahmen zur Performance-Steigerung:

Blockgruppen:

- Ziel: Vermeiden von weiten Kopfbewegungen.
- Beispiel: Linux ext2 File System (vgl. 10.3.5).
- Kein Effekt (auch kein negativer) bei Flash-Speicher

Block Buffer Cache.

- Ziel: Reduzierung der Anzahl der Plattenzugriffe.
- Ein Teil des Arbeitsspeichers, **(Block) Buffer Cache** genannt, wird als Cache für Dateisystemblöcke organisiert.
- Typische Hitrate: 85%
- Modifizierter LRU-Algorithmus zur Auswahl zu verdrängender Blöcke unter Berücksichtigung der Forderungen zur Verringerung der Gefahr von Inkonsistenz
- Blöcke verbleiben im Cache, auch wenn die entsprechenden Dateien geschlossen sind.

Notizen

Performance des Dateisystems (2)

Hochschule RheinMain

Dateinamens-Cache:

- Ziel: Verringerung der Anzahl der Schritte bei der Abbildung von Datei-Pfadnamen auf Blockadressen.
- Beispiel: BSD UNIX namei-Cache:
  - `namei()`-Routine
    zur Umsetzung von Pfadnamen auf inode-Nummern.
    benötigte vor Einführung von Caching ca. 25% der Zeit eines Prozesses im BS-Kern, auf <10% gesenkt.
  - Cache der *n* letzten übersetzten Teilnamen:
    Typische Hitrate: 70-80%.
    Höchste Bedeutung für Gesamtperformance.
  - Cache des letzten benutzten Directory-Offsets:
    Wenn ein Name im selben Verzeichnis gesucht wird, beginnt die Suche am gespeicherten offset und nicht am Anfang des Verzeichnisses (sequentielles Lesen eines Verzeichnisses ist häufig, z.B. durch das ls-Kommando).
    Typische Hitrate: 5-15 %.
  - Gesamthitrate von ca. 85% wird erreicht.

Notizen

5) Prozesssynchronisation

Definitionen

-Prozesskonflikt: zwei nebenläufige (Concurrent) Prozesse heißen im Konflikt zueinander stehend oder überlappend, wenn es eine BM gibt, das sie gemeinsam (lesend und schreibend) benutzen, ansonsten heißen sie unabhängig oder disjunkt

- Zeitkritische Abläufe (race conditions): Folgen von Lese/Schreib-Operationen der verschiedenen Prozesse heißen zeitkritische Abläufe (engl. race conditions), wenn die Endzustände der Betriebsmittel (Endergebnisse der Datenbereiche) abhängig von der zeitlichen Reihenfolge der Lese/Schreib-Operationen sind.

- Wechselseitiger Ausschluss (mutual exclusion): Ein Verfahren, das verhindert, dass zu einem Zeitpunkt mehr als ein Prozess auf ein gemeinsames Datum zugreift, heisst Verfahren zum wechselseitigen Ausschluss.
Bemerkung: ein solches Verfahren vermeidet zeitkritische Abläufe und löst somit ein Basisproblem des Concurrent Programming

- Kritischer Abschnitt (critical section): Der Teil eines Programms, in dem auf gemeinsam benutzte Datenbereiche zugegriffen wird
Bemerkung: Ein Verfahren, das sicherstellt, dass sich zu keinem Zeitpunkt zwei Prozesse in ihrem kritischen Abschnitt befinden, vermeidet zeitkritische Abläufe. Kritische Abschnitte realisieren sog. komplexe unteilbare oder atomare Operationen.

Anforderungen an einen guten Algorithmus

- Immer nur ein Prozess in seinem kritischen Abschnitt (Korrektheit, Basisforderung)
-Kein Prozess,der nicht in seinem kritischen Bereich ist,darf andere Prozesse blockieren (Fortschritt)
- Alle Prozesse werden gleich behandelt (Fairness)
- Kein Prozess darf unendlich lange warte müssen, bis er in seinen kritischen Bereich eintreten kann (starvation)

Synchronisationsprimitive

Wechselseitiger Ausschluss mit aktivem Warten

- Funktionen: enter_critical_section und leave_critical_section
- Lösungen:
  1 Sperren aller Unterbrechungen: Interruptkonfiguration i.d.R. nur im Kernmodus möglich, unbrauchbar bei Multiprozessor-Systeme
  2 Sperrvariablen: Zwischen Abfrage der Sperrvariablen und folgendem Setzen kann der Prozess unterbrochen werden
  3 Striktes Alternieren: erfüllt im Vergleich zu 2 die Korrektheits-bedingung. Wenn ein Prozess viel langsamer als der andere ist kann die Fortschrittsbedingung verletzt werden
  4 Peterson: enter_critical Funktion zeigt eigenes Interesse und setzt Marke. leave_critical verlässt kritischen Bereich (kein Interesse mehr)
  5 Atomare read-modify-write Instruktionen: Algorithmen sind komplex, fehleranfällig und starvation-anfällig. Lösung durch HW-Unterstützung. Atomare Maschinenbefehle (TAS = Test And Set)
- -Lock Holder Preemption Problem (kann lange dauern [Quantum])
  -ein (virtueller) Prozessor hat einen durch Spinlock geschützten Bereich betreten und wird dort unterbrochen
  - anderer (virtueller) Prozessor wartet auf Freigabe des Spinlocks

Wechselseitiger Ausschluss mit passivem Warten

- Einfachste Primitive heißen meistens SLEEP() und WAKEUP(process)
- Mutex-Locks (lock() als Prolog und unlock als Epilog)
- Problem der Prioritätsinversion: beim prioritätsbasierten Scheduling muss ein Prozess mit hoher Priorität auf einen Prozess mit niedriger Priorität warten weil dieser den kritschen Abschnitt noch nicht freigegeben hat
- Semaphore: Supermarkt-Einkaufswagen Analogie besteht aus Zählvariable, die begrenzt, wieviele Prozesse momentan ohne Blockierung passieren dürfen. Und einer Warteschlange für (passiv) wartende Prozesse.

Operationen:

- Zähler auf initialen Wert (# Freie Einkaufswagen) setzen
- P(): Passierwunsch (auch DOWN() genannt)
- V(): Freigeben (auch UP() genannt)
- P() und V() sind atomar
- kein Prozess wird bei der Ausführung von V() blockiert
- i.d.R. als Systemaufrufe implementiert
- Einprozessorsysteme sperren Interrupts bei P() und V()
- Multiprozessorsysteme beschützen Semaphore (unkritisch) durch Spinlocks. Es kann immer nur ein Prozessor den Semaphor manipulieren

-Binär- und Zählsemaphore [Programmierung ist fehleranfällig)

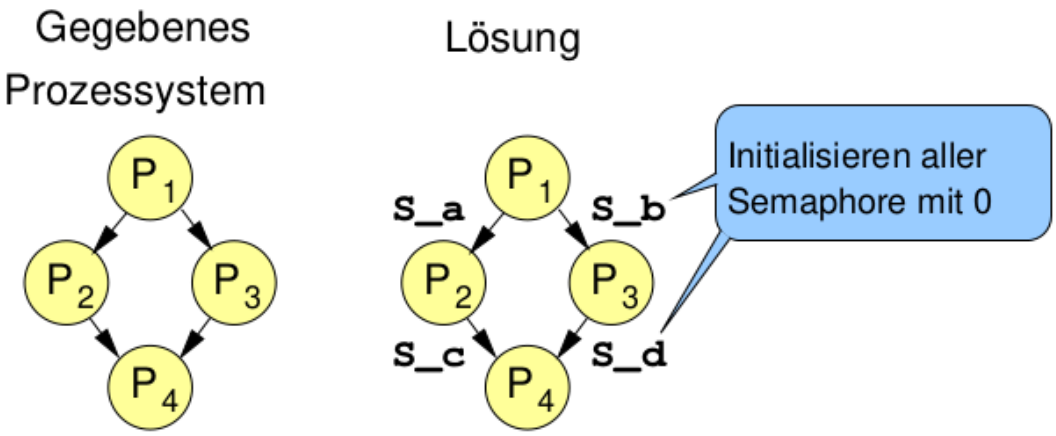
```
/* Semaphore initialisieren:
 * empty = Puffergroesse,
 * full=0, mutex=1
 */

void insert(int item) {
    P(&empty);
    P(&mutex);
/* hier: item in Puffer
   stellen                */
    V(&mutex);
    V(&full);
}

int remove(void) {
    P(&full);
    P(&mutex);
/* hier: vorderstes Item
   aus Puffer holen    */
    V(&mutex);
    V(&empty);
}
```

full: zählt belegte Einträge im Puffer, verhindert Entnahme aus leerem Puffer

empty: verwaltet freie Plätze im Puffer, verhindert Einfügen in vollen Puffer

mutex: schützt den kritischen Bereich vor gleichzeitigem Betreten (binär-Semaphor: nimmt nur Werte 1/0 an)

Beispiel: Vorrangrelation




```
P1() {
.. work ..
 V(S_a);
 V(S_b);
 exit();
}
```

```
P2() {
 P(S_a);
.. work ..
 V(S_c);
 exit();
}
```

```
P3() {
 P(S_b);
.. work ..
 V(S_d);
 exit();
}
```

```
P4() {
 P(S_c);
 P(S_d);
.. work ..
 exit();
}
```

Weitere Ansätze

-Condition Variable

-Monitore

Klassische Synchronisationprobleme

-Erzeuger-Verbraucher Problem

Erzeuger: Will Einfügen, aber Puffer ist voll.
Lösung: Lege dich schlafen, lass dich vom Verbraucher wecken, wenn er ein Datum entnommen hat.
Verbraucher: Will Entnehmen, aber Puffer ist leer.
Lösung: Lege dich schlafen, lass dich vom Erzeuger wecken, wenn er ein Datum eingefügt hat.

Lösung mit Semaphoren (1)

Hochschule RheinMain

Wurde bereits besprochen (vgl. 5.2.3)

```
#define N 100                   /* Kapazitaet des Puffers        */

/* gemeinsame Variablen
*/
semaphore mutex = 1;            /* kontrolliert krit. Bereich    */
semaphore empty = N;            /* zaehlt leere Eintraege        */
semaphore full  = 0;            /* zaehlt belegte Eintraege      */

void erzeuger(void)             /* Erzeuger                      */
{
    int item;
    while (TRUE) {
        produce_item(&item);    /* erzeuge Eintrag               */
        P(&empty);              /* besorge freien Platz          */
        P(&mutex);              /* tritt in krit. Abschnitt ein  */
        enter_item(item);       /* fuege Eintrag in Puffer ein   */
        V(&mutex);              /* verlasse krit. Bereich        */
        V(&full);               /* erhoehe Anz. belegter Eintr.  */
    }
}
```


Lösung mit Semaphoren (2)

Hochschule RheinMain

```
void verbraucher(void)/* Verbraucher                    */
{
    int item;
    while(TRUE) {
        P(&full);           /* belegter Eintrag vorhanden?   */
        P(&mutex);          /* tritt in krit. Abschnitt ein  */
        remove_item(&item); /* entnimm Eintrag aus Puffer    */
        V(&mutex);          /* verlasse krit. Bereich        */
        V(&empty);          /* erhoehe Anz. freier Eintraege */
        consume_item(item); /* verarbeite Eintrag            */
    }
}
```

Lösung mit Nachrichtenaustausch (1)

Hochschule RheinMain

Grundlage: Funktionen zum Nachrichtenaustausch

`send(process, int* message)` - Nachricht an Prozess senden

`receive(process, int* message)` - Nachricht von Prozess empfangen

```
#define N      100              /* Kapazitaet des Puffers        */
#define MSIZE    4              /* Nachrichtengroesse            */

typedef int message[MSIZE];     /* Nachrichtentyp                */

void producer(void)             /* Erzeuger                      */
{
   int item;
   message m;

   while (TRUE) {
      produce_item(&item);      /* erzeuge Eintrag               */
      receive(consumer, &m);    /* warte auf leere Nachricht     */
      build_message(&m, item); /* erzeuge zu sendende Nachricht*/
      send(consumer, &m);       /* sende Nachricht z Verbraucher*/
   }
}
```


Lösung mit Nachrichtenaustausch (2)

Hochschule RheinMain

```
void consumer(void)             /* Verbraucher                   */
{
   int item, i;
   message m;

   for (i = 0; i < N; i++)      /* sende N leere Nachrichten     */
      send (producer, &m);

   while (TRUE) {
      receive(producer, &m);    /* empfange Nachricht v Erzeuger*/
      extract_item(&m, &item); /* entnimm Eintrag               */
      send(producer, &m);       /* sende leere Nachricht zurueck*/
      consume_item(item);       /* verarbeite Eintrag            */
   }
}
```

- Dining Philosophers Problem

Lösung mit Semaphoren (1)

Hochschule RheinMain

```
#define N           5       /* Anzahl der Philosophen              */
#define LEFT  (i-1+N)%N     /* Nummer des linken Nachbarn von i    */
#define RIGHT (i+1)%N       /* Nummer des rechten Nachbarn von i   */
#define THINKING    0       /* Zustand: Denkend                    */
#define HUNGRY      1       /* Zust: Versucht, Gabeln zu bekommen  */
#define EATING      2       /* Zustand: Essend                     */

/* gemeinsame Variablen                                           */
int state[N];               /* Zustaende aller PhilosophInnen      */
semaphore mutex = 1;        /* fuer wechselseitigen Aussschluss    */
semaphore s[n];             /* Semaphor fuer jeden Philosoph       */

void philosopher(int i) {  /* i:0..N-1, welcher Philosoph      */
    while (TRUE) {
        think();            /* Denken                              */
        take_forks(i);      /* Greife beide Gabeln oder blockiere  */
        eat();              /* Essen                               */
        put_forks(i);       /* Ablegen beider Gabeln               */
    }
}
```


Lösung mit Semaphoren (1)

Hochschule RheinMain

```
void take_forks(int i) /* i:0..N-1, welche(r) PhilosophIn?  */
{
    P(&mutex);          /* tritt in krit. Bereich ein          */
    state[i] = HUNGRY;  /* zeige, dass du hungrig bist         */
    test(i);            /* versuche, beide Gabeln zu bekommen  */
    V(&mutex);          /* verlasse krit. Bereich              */
    P(&s[i]);           /* bockiere, falls Gabeln nicht frei   */
}

void put_forks(int i) /* i:0..N-1, welche(r) PhilosophIn?   */
{
    P(&mutex);          /* tritt in krit. Bereich ein          */
    state[i] = THINKING;/* zeige, dass du fertig bist          */
    test(LEFT);         /* kann linker Nachbar jetzt essen ?   */
    test(RIGHT);        /* kann rechter Nachbar jetzt essen ?  */
    V(&mutex);          /* verlasse krit. Bereich              */
}

void test(int i) /* i:0..N-1, welche(r) PhilosophIn? */
{
    if (state[i]==HUNGRY &&
        state[LEFT]!=EATING && state[RIGHT]!=EATING) {
        state[i]=EATING;/* jetzt kann Phil i essen !           */
        V(&s[i]);       /* "sage es ihm"                       */
    }
}
```

- Leser Schreiber Problem
Zu jedem Zeitpunkt dürfen entweder mehrere Leser oder ein Schreiber zugreifen.
Verboten: gleichzeitiges Lesen und Schreiben
Wie sollten Leser- und Schreiber-Programme aussehen?

Lösung mit Semaphoren (1)

Hochschule RheinMain

Lesezugriff:

```
/* gemeinsame Variablen: */
semaphore mutex = 1;      /* wechsels. Aussschluss fuer rc     */
semaphore db = 1;         /* Semaphor fuer Datenbestand        */
int rc = 0;               /* readcount: Anzahl Leser           */

void reader(void)         /* Leser                             */
{
   while (TRUE) {
      P(&mutex);          /* erhalten exkl. Zugriff auf rc     */
      rc = rc + 1;        /* ein zusaetzlicher Leser           */
      if (rc==1)          /* Erster Leser?                     */
          P(&db);         /* ja -> reserviere Daten            */
      V(&mutex);          /* freigeben exkl. Zugriff auf rc    */

      read_data_base();   /* lies Datenbestand                 */

      P(&mutex);          /* erhalten exkl. Zugriff auf rc     */
      rc = rc - 1;        /* ein Leser weniger                 */
      if (rc==0)          /* letzter Leser ?                   */
          V(&db);         /* ja -> Daten freigeb.              */
      V(&mutex);          /* freigeben exkl. Zugriff auf rc    */
      use_data_read();    /* unkrit. Bereich                   */
   }
}
```


Lösung mit Semaphoren (2)

Hochschule RheinMain

Schreibzugriff:

```
void writer(void)         /* Schreiber                         */
{
   while (TRUE) {
      create_data();      /* unkrit. Bereich                   */
      P(&db);             /* erhalten exkl. Zugriff auf Daten  */
      write_data_base();  /* schreibe Datenbestand             */
      V(&db);             /* freigeben exkl. Zugriff auf Daten */
   }
}
```

`mutex` sichert krit. Abschnitt bezüglich des Read-Counters `rc`.
`db` sichert Zugriff auf den Datenbestand, so dass **entweder** mehrere Leser **oder** ein Schreiber zugreifen können.
Der erste Leser führt eine P-Operation auf `db` aus, alle weiteren inkrementieren nur `rc`.
Der letzte Leser führt eine V-Operation auf `db` aus, so dass ein wartender Schreiber Zugriff erhält.
Die **Lösung bevorzugt Leser**: Neu eintreffende Leser erhalten Zugriff vor einem schon wartenden Schreiber, wenn noch mindestens ein Leser Zugriff hat.