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)

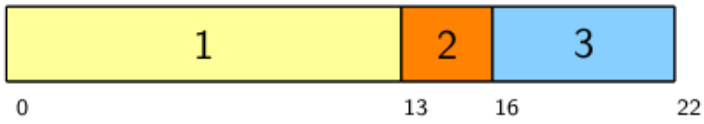
-FCFS (first come first served): Ready-Queue als FIFO Liste

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: $(6 + 9)/3 = 5$

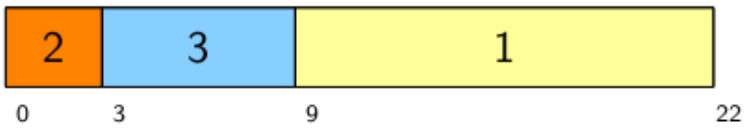
-SJF (Shortest Job First)

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

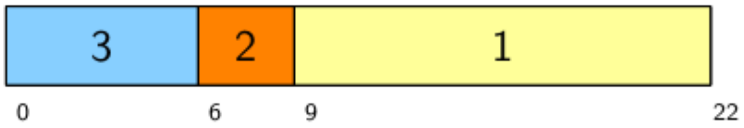
-Prioritäts-Scheduling: Jeder Auftrag hat statische Priorität
höchste Priorität hat Vorrang
Bei gleicher Priorität FCFS

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

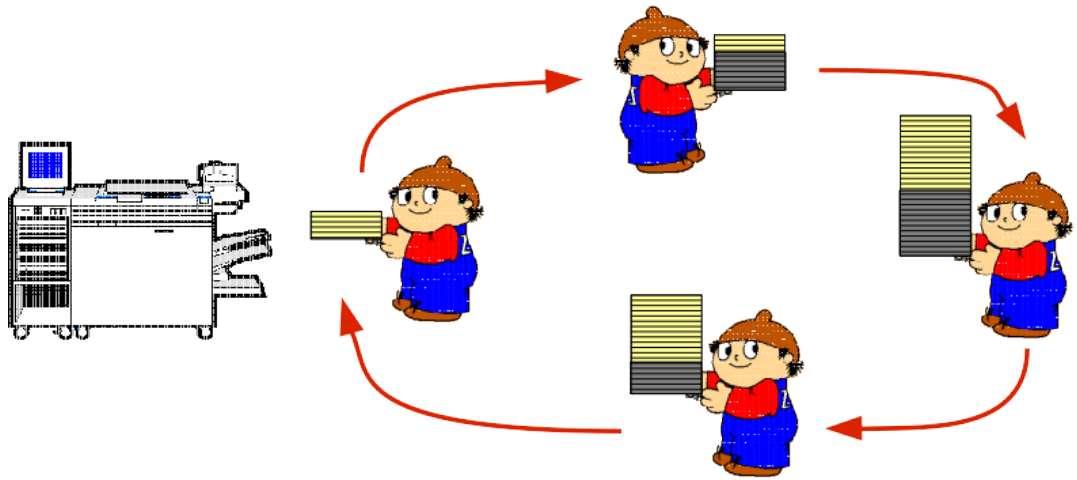
Durchschnittliche Wartezeit[4]:
$(9+6)/3 = 5$

[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.

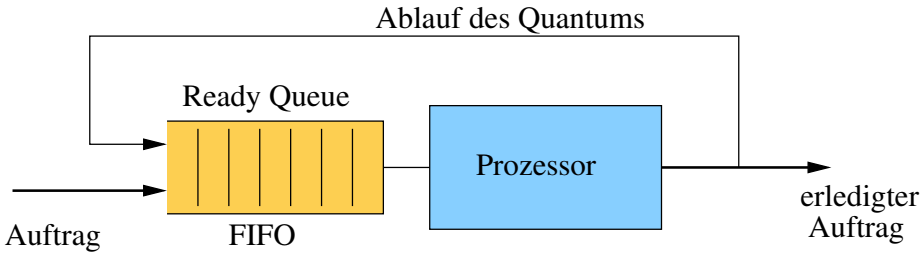
Implementierung:

- 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

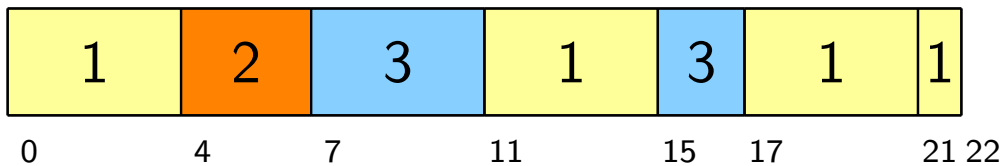
Gegeben: Prozessmenge mit 3 Prozessen

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

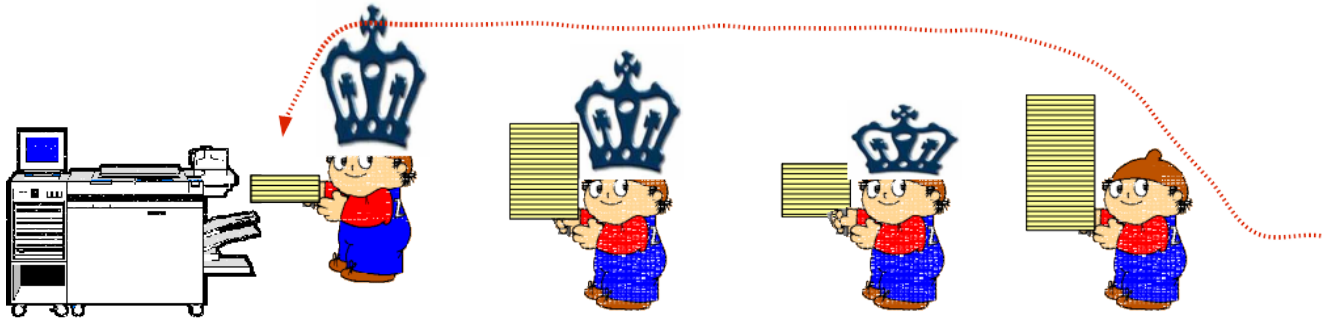
Durchschnittliche Wartezeit:
$(9 + 4 + 11)/3 = 8$

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.

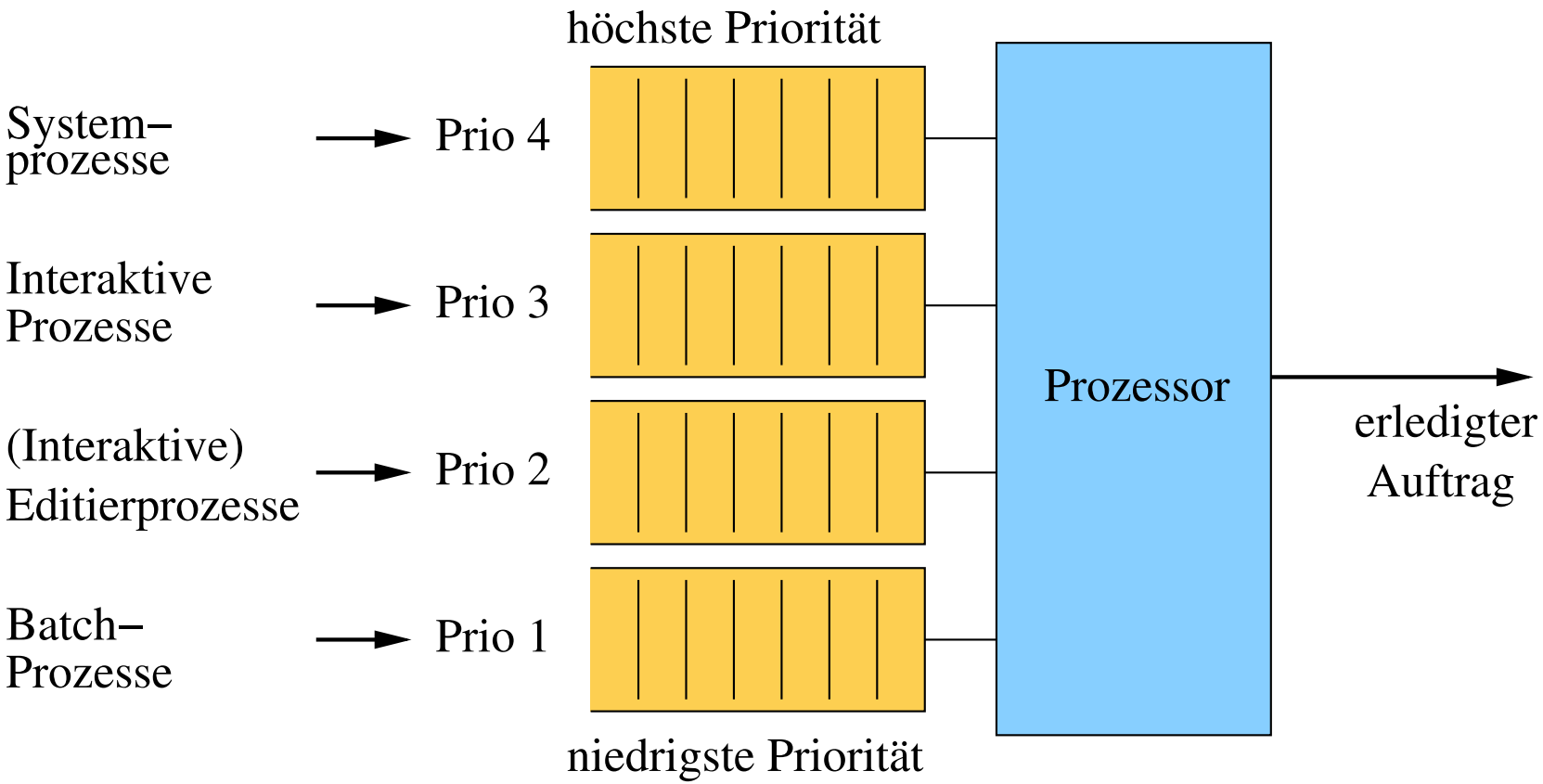
Mehrschlangen-Scheduling

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).

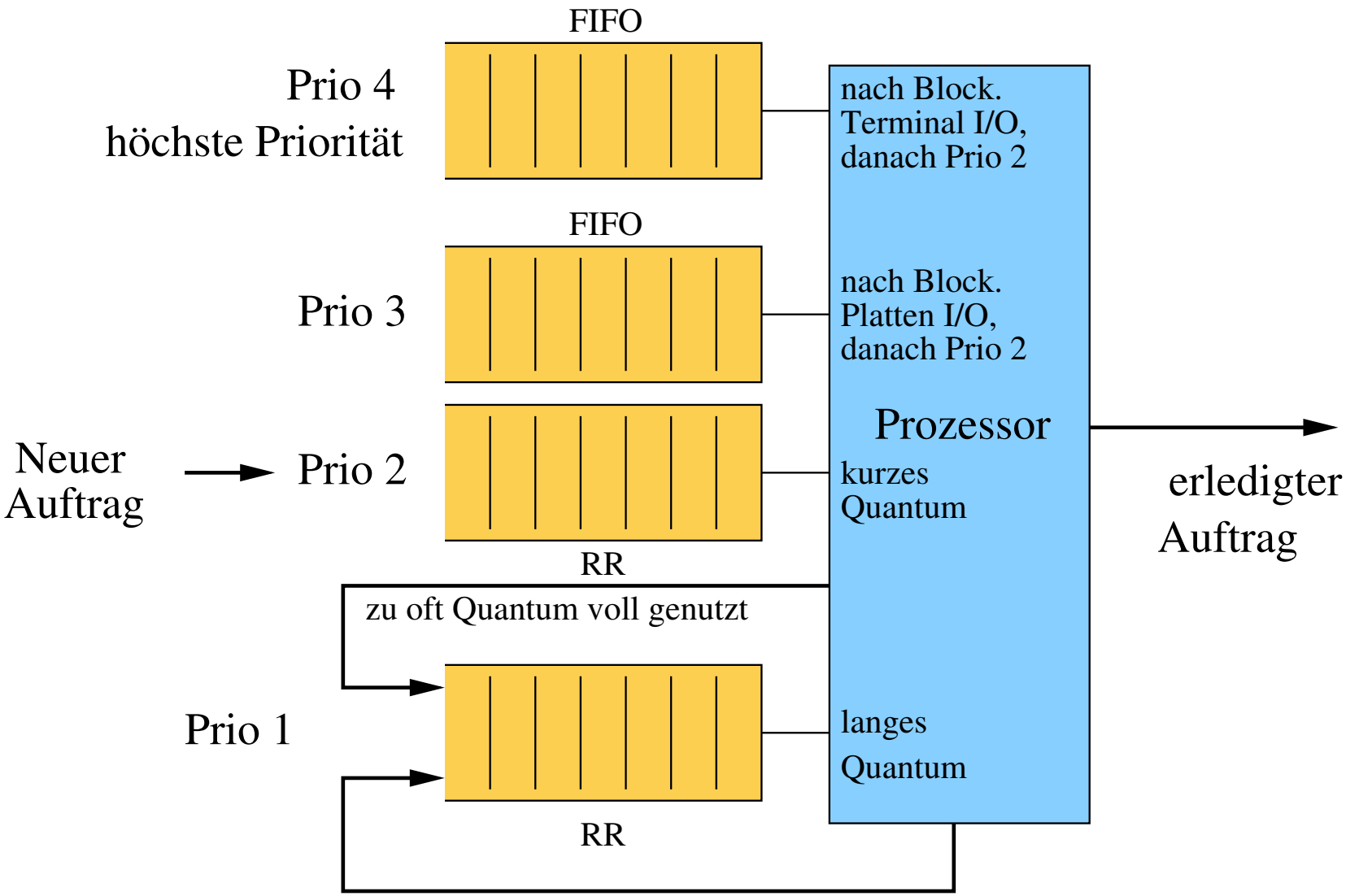
Bsp. 2: Wenn ein Prozess in einer bestimmten Priorität viel Rechenzeit zugeordnet bekommen hat, wird seine Priorität verschlechtert (Bestrafung von Langläufern).

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“)

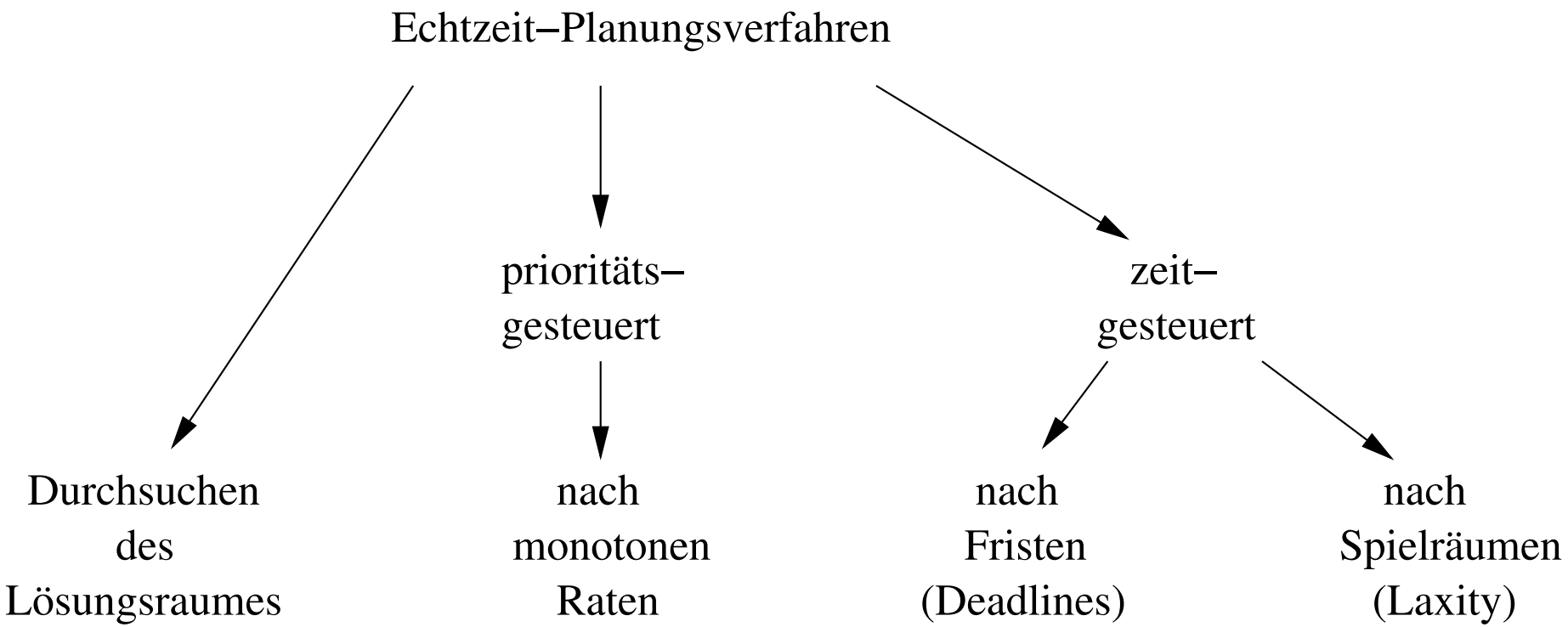
Echtzeit-Scheduling

- 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

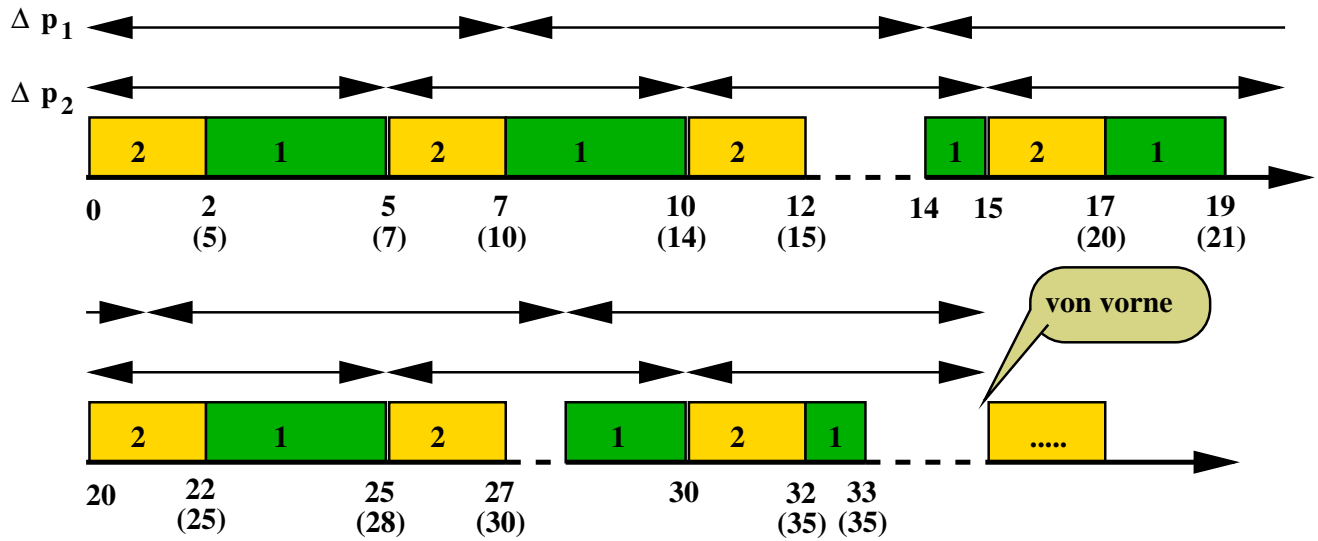
Prozess i	Bedienzeit Δe_i	Periode Δp_i
1	3	7
2	2	5

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

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

→ 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

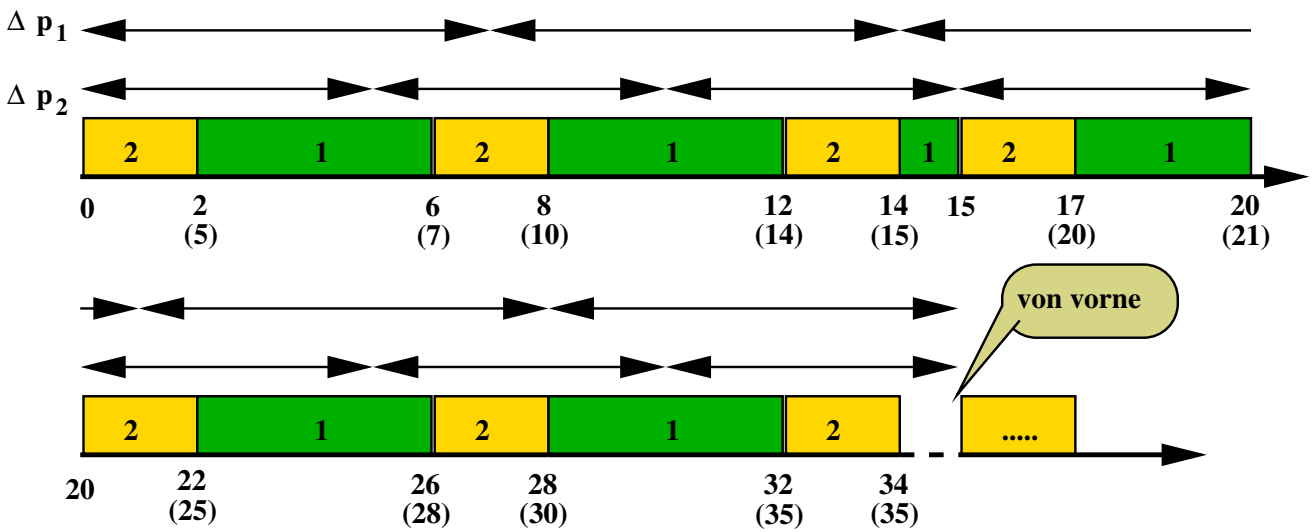
Gegeben: Prozessmenge mit 2 Prozessen

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.

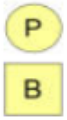
Vorraussetzungen:
-Wechselseitiger Ausschluß: (BM frei oder einem Prozess zugeteilt)
-Belegungs-Anforderungsbedingung (Hold-and-wait): Prozesse können zu bereits reservierten BM noch weitere anfordern
-Ununterbrechbarkeit: zugeteilt BM müssen freigegeben werden um für einen anderen Prozess verfügbar 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)

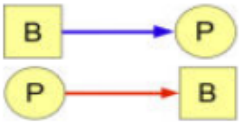
Es gibt zwei Knotentypen:

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

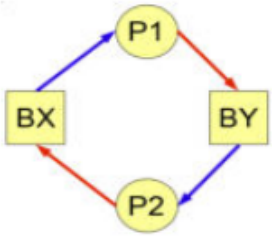


Pfeile:

- 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

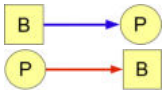
Graphische Darstellung der Beziehung von Prozessen zu Betriebsmitteln (Holt, 1972)

Es gibt zwei Knotentypen:

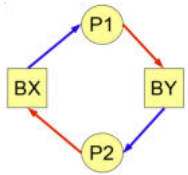
- Prozesse, repräsentiert durch Kreise:
- Betriebsmittel, repräsentiert durch Quadrate:

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

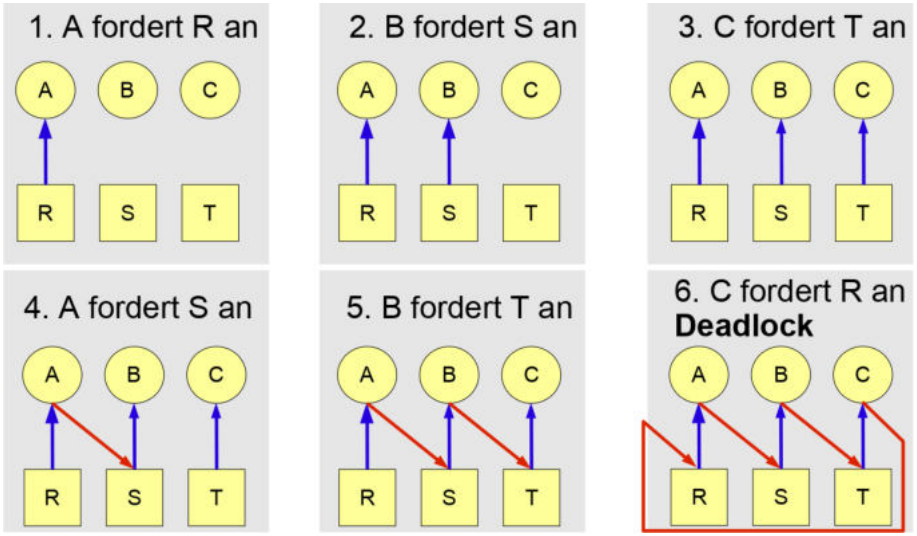
Das Betriebssystem kann jeden (nicht blockierten) Prozess **jederzeit** ausführen

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

Wie sieht es bei nebenläufiger Ausführung aus?

Notizen

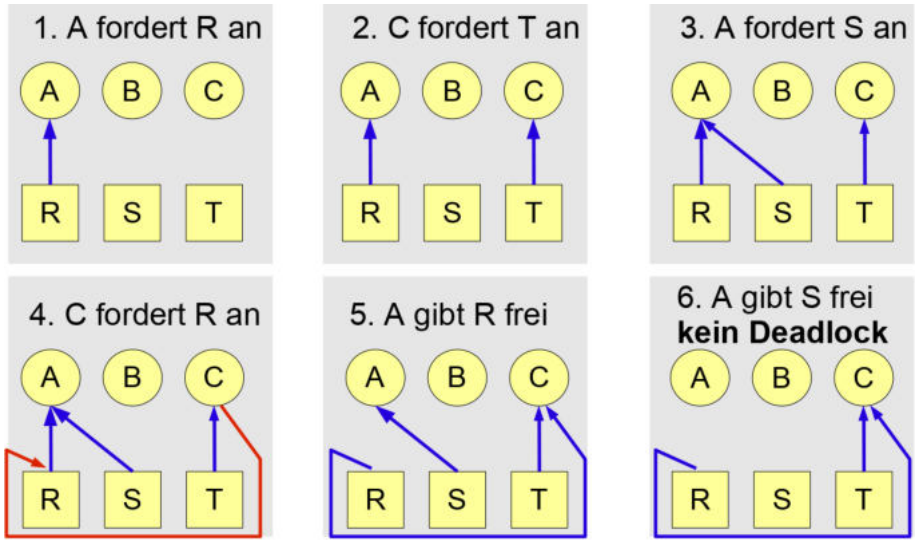
Ausführung I



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)

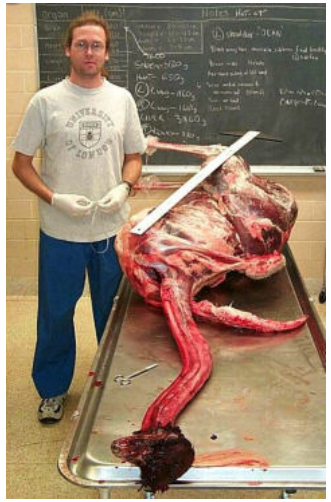
Notizen

...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.

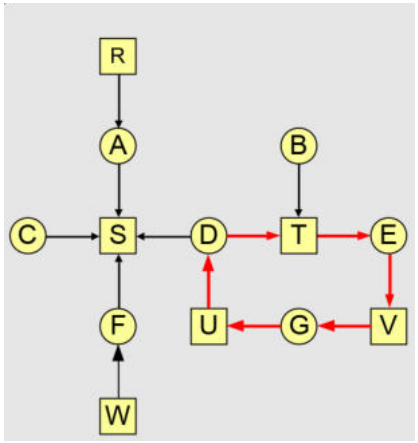
C fordert S an.

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**

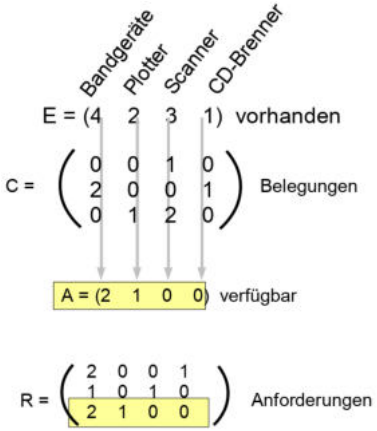
Gefunden? Dann kann P_i durchlaufen und gibt danach seine belegten Betriebsmittel zurück: $A = A + C_i$, wird markiert und es geht beim nächsten unmarkierten Prozess weiter

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?

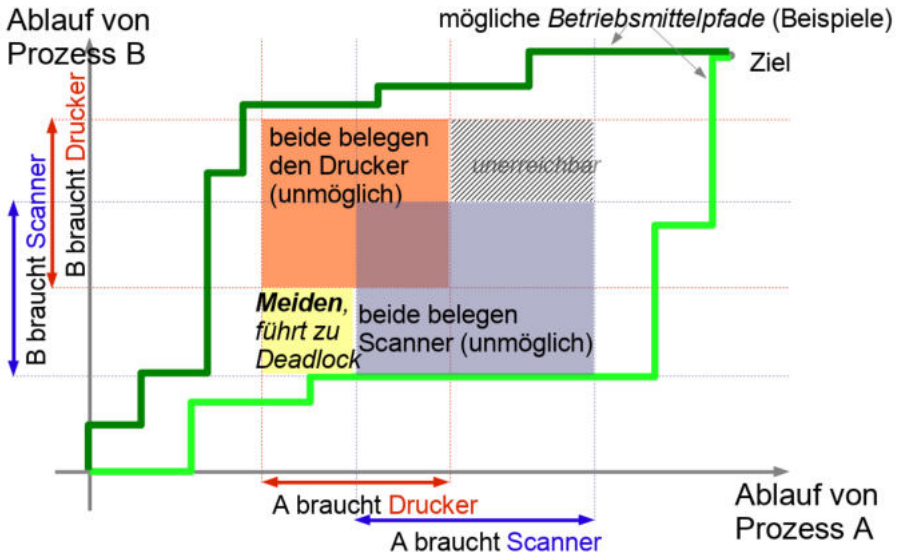
Im folgenden betrachtet

- 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)

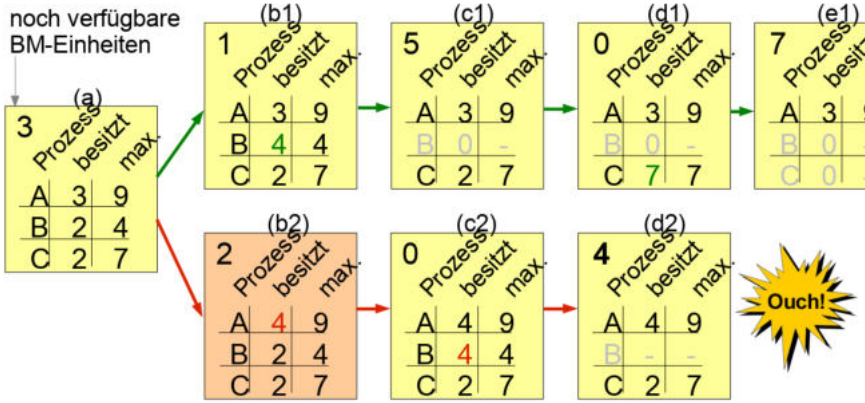
„Unsicher“ bedeutet also nicht „Deadlock unvermeidlich“.

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?

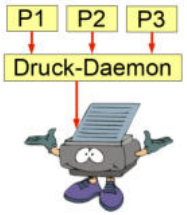
Hochschule RheinMain

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

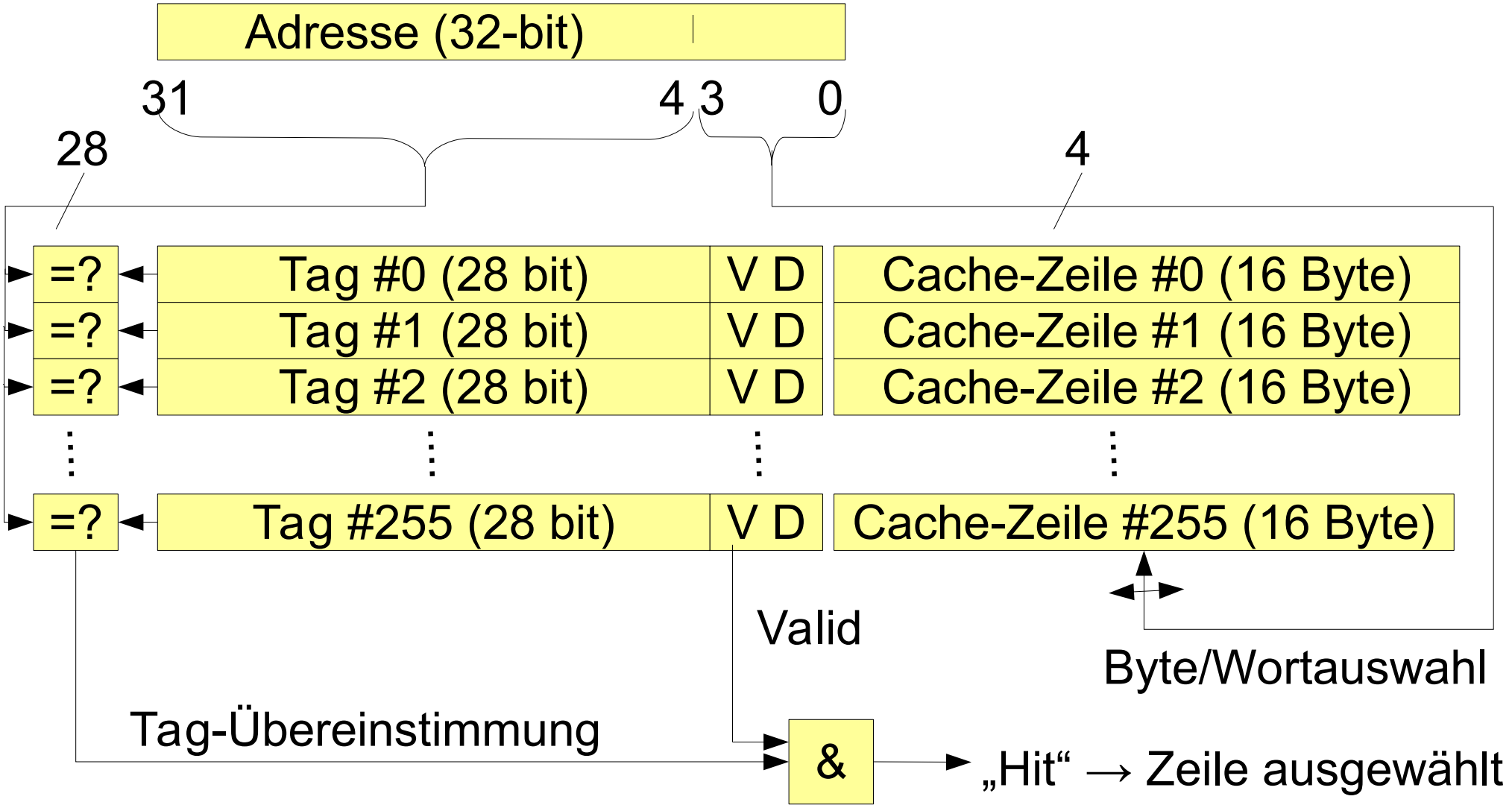
-Verdrängungsstrategien (wenn alle Cache Zeilen belegt sind → „Platz schaffen“)
  - Random (einfach, überraschend gut)
  - FIFO (die im längsten im Cache gespeicherte Adresse wird ersetzt → schlecht)
  - LRU (least recently used): die im längsten nicht verwendete Zeile ersetzen
  - LFU (least frequently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

Cache-Organisationsformen

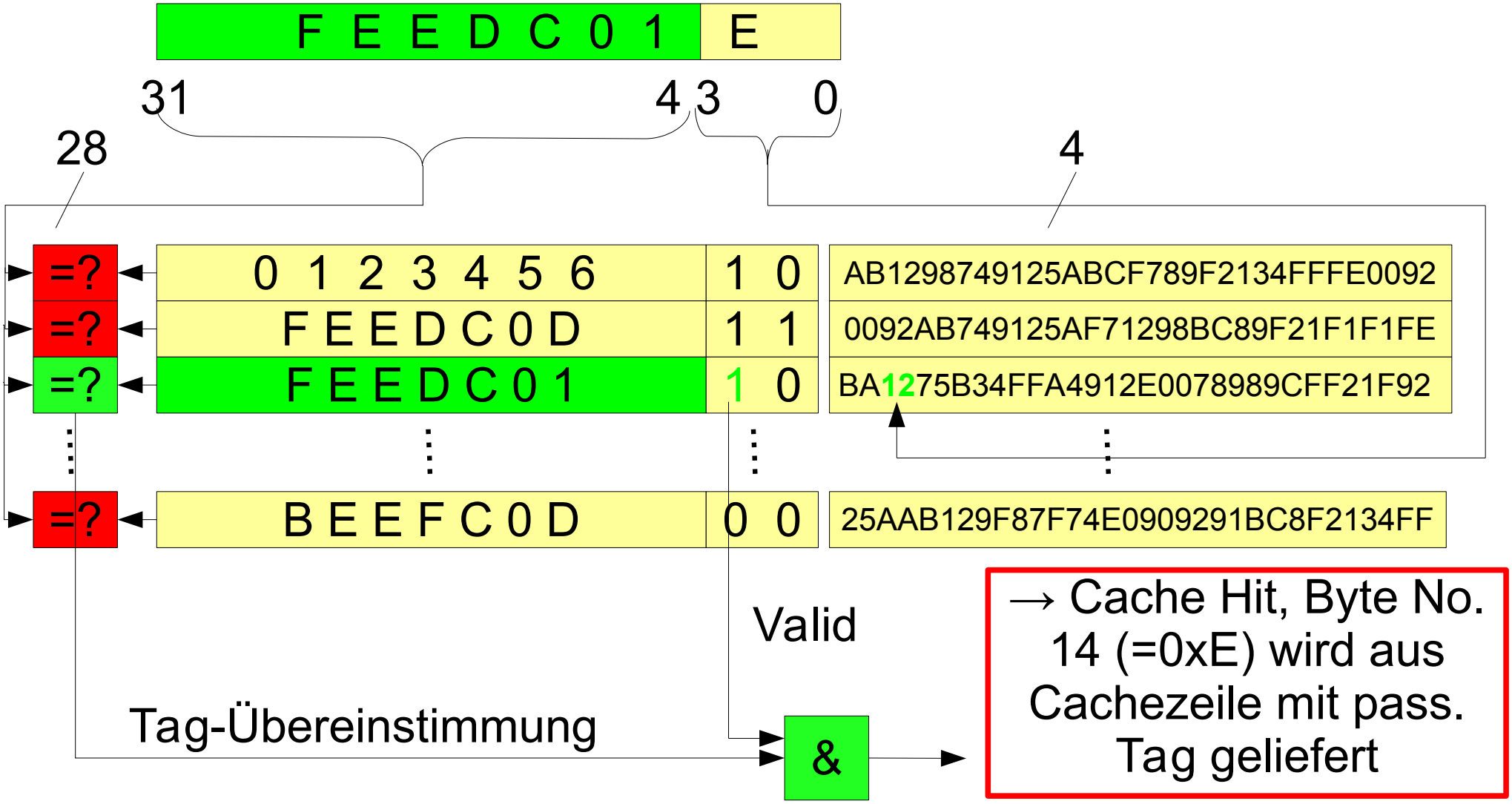
- Cache-Organisationsformen:
  1. Vollassoziativ: *fully associative*
  2. Direkt abbildend: *direct-mapped*
  3. Mehrfach assoziativ: *N-way set associative*
- Annahmen bei den folgenden Beispielen:
  - 32-bit Adressierung
  - 4K (4096) Byte Cache
  - Cache-Zeilengröße: 16 Byte
    (→ 4096 : 16 = 256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

Vollassoziativer Cache: Aufbau

Hochschule RheinMain



Vollassoziativer Cache: Beispiel

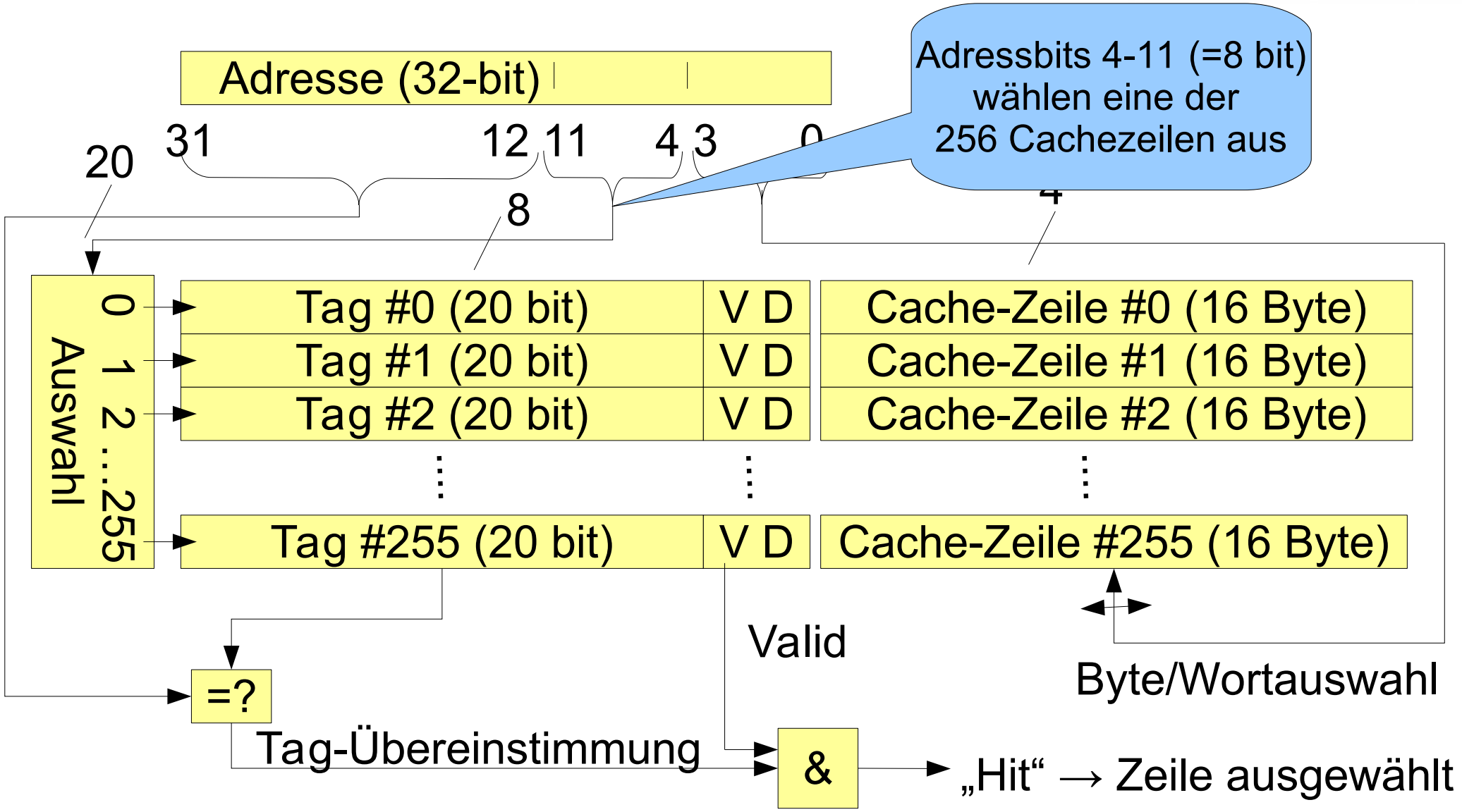


Vollassoziativer Cache

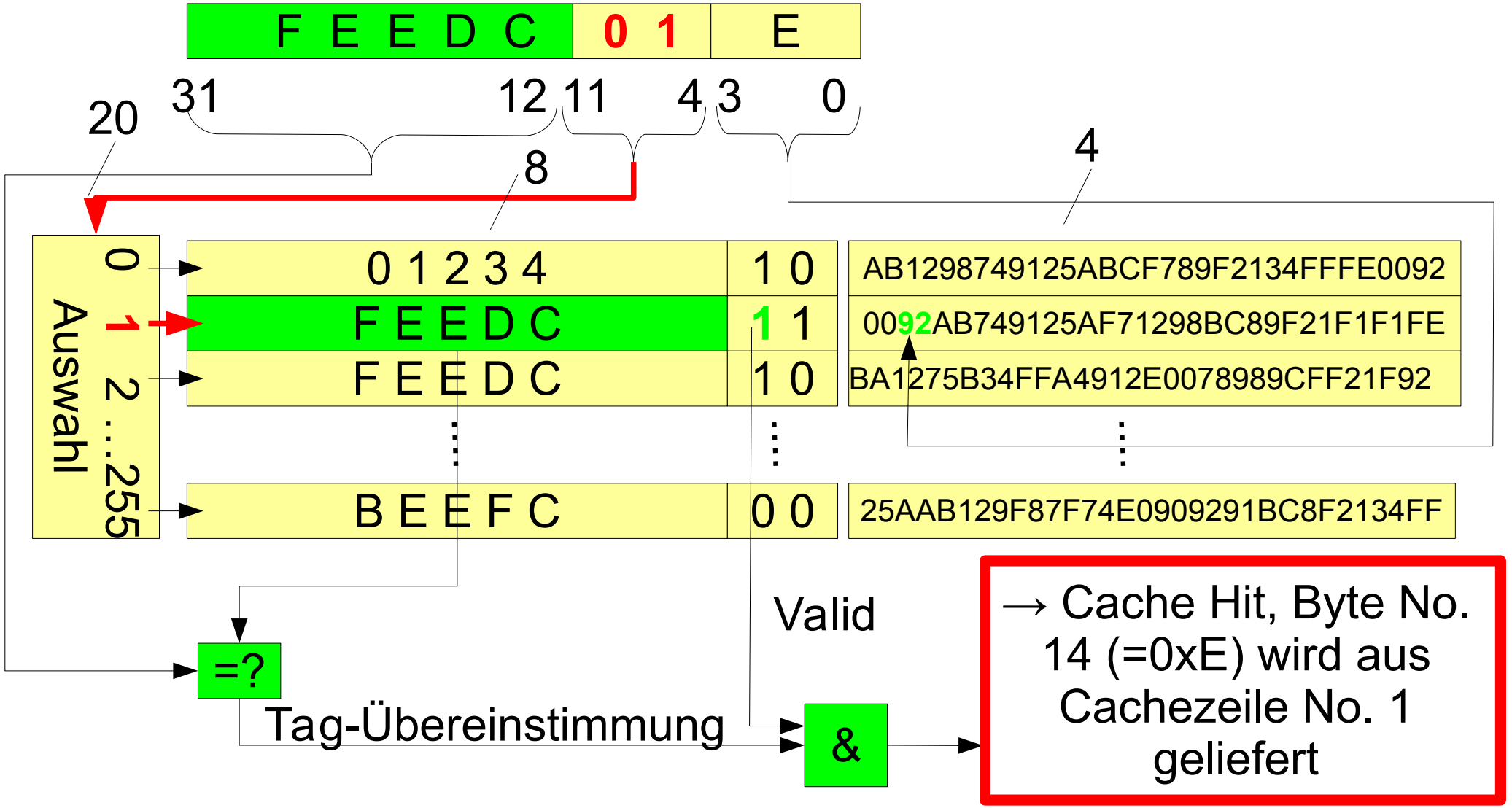
- Ein Objekt kann in eine beliebige Cache-Zeile kopiert werden
  → Freie Auswahl eines freien / zu verdrängenden Eintrags
- Identifikation der Zeile ausschließlich anhand des Tags
  → Konsequenzen:
  - Tags **aller** Zeilen müssen mit Adresse verglichen werden
  - Vergleich muss **gleichzeitig** auf 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

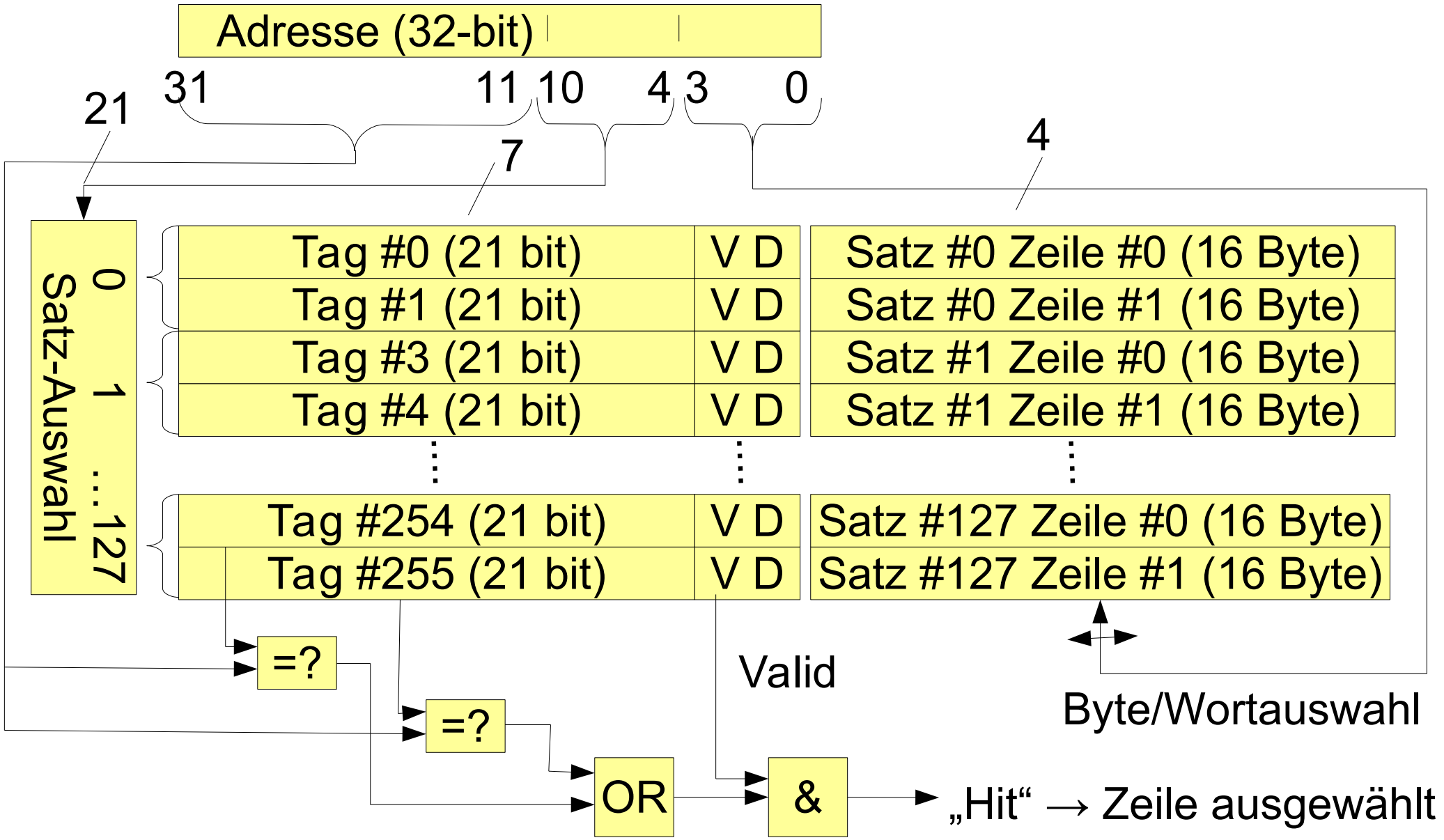
- Ein Teil der Adresse (im Beispiel: Bits 4 bis 11) wählt die Cachezeile aus
- Eindeutige Zuordnung ohne Wahlfreiheit, keine alternativen Verdrängungsstrategien möglich (aber auch keine nötig)
- Tag dient allein zur Hit/Miss Entscheidung
- **Konsequenzen**:
  - (+) Einfacher Aufbau (nur ein Vergleicher 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

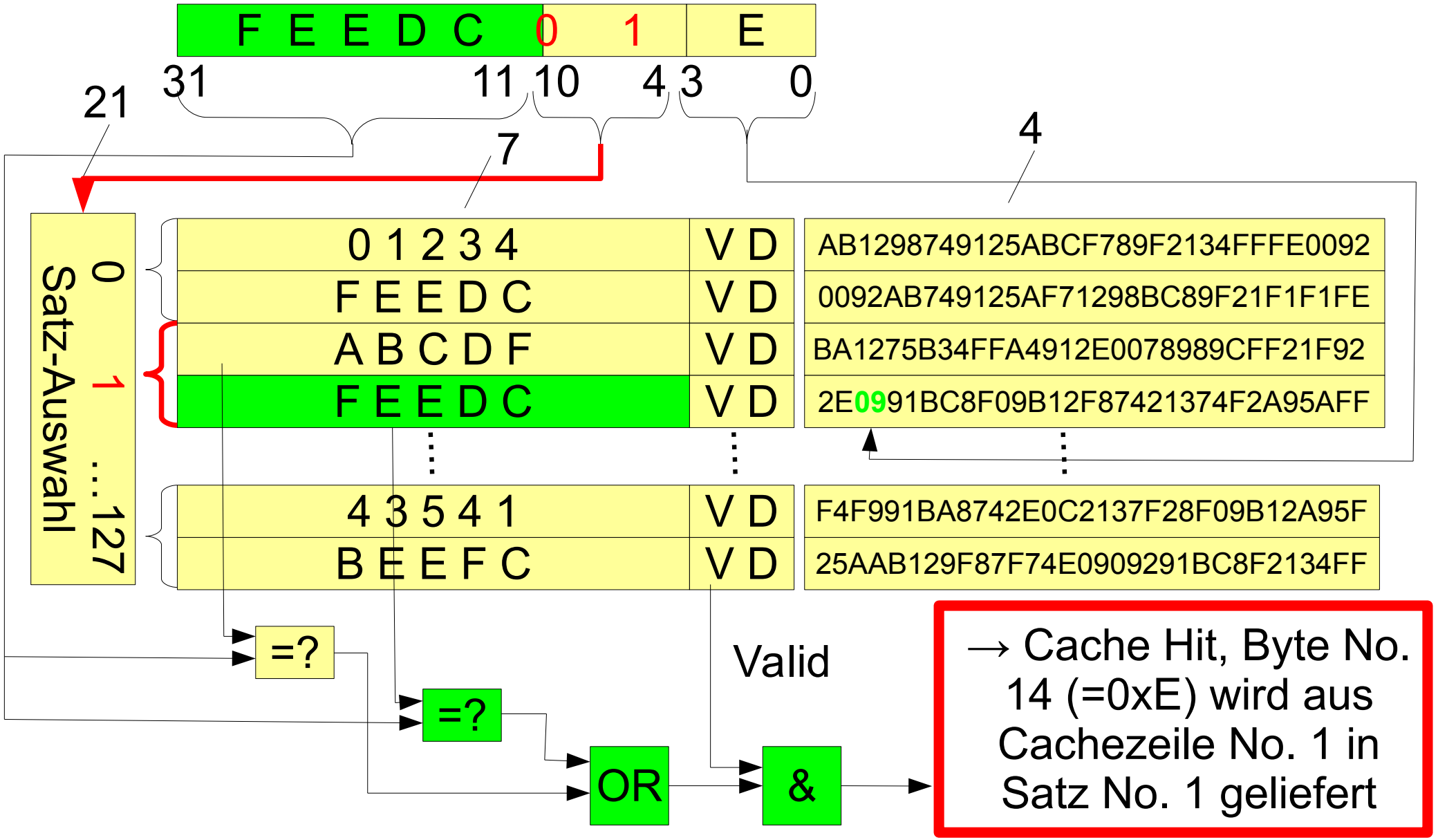
- Vollassoziativer Cache: Hohe Trefferquote, aber aufwändig
- Direkt abbildender Cache: Geringer Aufwand, aber schlechte Trefferquote (neigt zu „thrashing")
- Kompromiss: Mehrfach assoziativer Cache(*)
  - Zusammenfassen von je N (N = 2, 4, 8, ...) Cache-Zeilen zu einem „Satz" (engl. „s*et*")
  - Ein 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

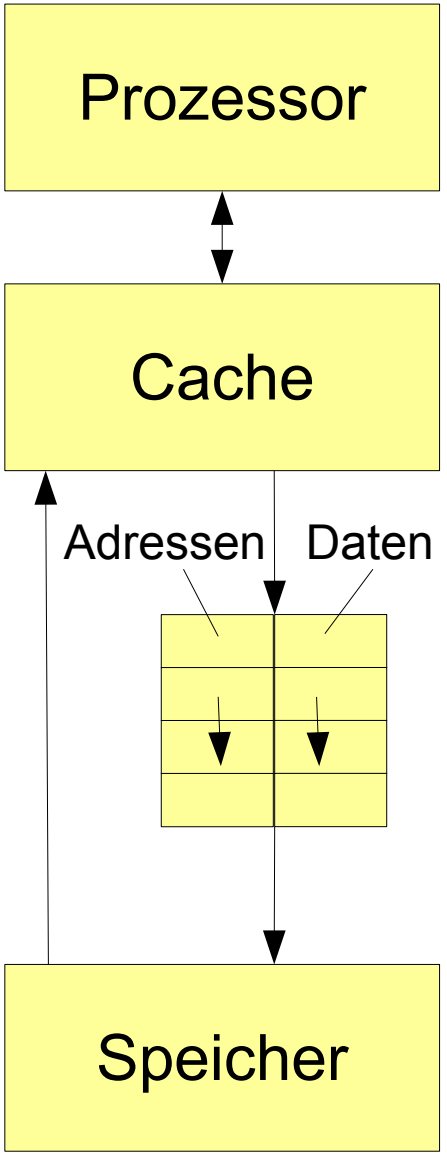
- Pentium 4:
  - L1 Datencache: 4-fach assoziativ (64 Byte Zeilengröße)
  - L1 Befehlscache: 8-fach assoziativ
  - L2 Cache: 8-fach assoziativ (64 Byte Zeilengröße)

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