# 5. Foliensatz Betriebssysteme

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences
(1971–2014: Fachhochschule Frankfurt am Main)
Fachbereich Informatik und Ingenieurwissenschaften
christianbaun@fb2.fra-uas.de

# Lernziele dieses Foliensatzes

- Am Ende dieses Foliensatzes kennen/verstehen Sie...
  - grundlegende Konzepte der **Speicherverwaltung**
    - **Statische Partitionierung**
    - **Dynamische Partitionierung**
    - **Buddy-Speicherverwaltung**
  - wie Betriebssysteme auf den **Speicher zugreifen** (ihn adressieren!)
    - **Real Mode**
    - **Protected Mode**
  - Komponenten und Konzepte um **virtuellen Speicher** zu realisieren
    - Memory Management Unit (**MMU**)
    - Seitenorientierter Speicher (**Paging**)
    - Segmentorientierter Speicher (**Segmentierung**)
- die möglichen Ergebnisse bei Anfragen an einen Speicher
  - **Hit** und **Miss**
- die Arbeitsweise und Eckdaten wichtiger **Ersetzungsstrategien**

Übungsblatt 5 wiederholt die für die Lernziele relevanten Inhalte dieses Foliensatzes

# Speicherverwaltung

- Eine wesentliche Funktion von Betriebssystemen

- Weist Programmen auf deren Anforderung hin Teile des Speichers zu
- Gibt auch Teile des Speichers frei, die Programmen zugewiesen sind, wenn diese nicht benötigt werden
- 3 Konzepte zur Speicherverwaltung:
  1. **Statische Partitionierung**
  2. **Dynamische Partitionierung**
  3. **Buddy-Speicherverwaltung**

Diese Konzepte sind schon etwas älter...


Bildquelle: unbekannt (eventuell IBM)

Eine gute Beschreibung der Konzepte zur Speicherverwaltung enthält...

- **Operating Systems – Internals and Design Principles**, *William Stallings*, 4.Auflage, Prentice Hall (2001), S.305-315
- **Moderne Betriebssysteme**, *Andrew S. Tanenbaum*, 3.Auflage, Pearson (2009), S.232-240

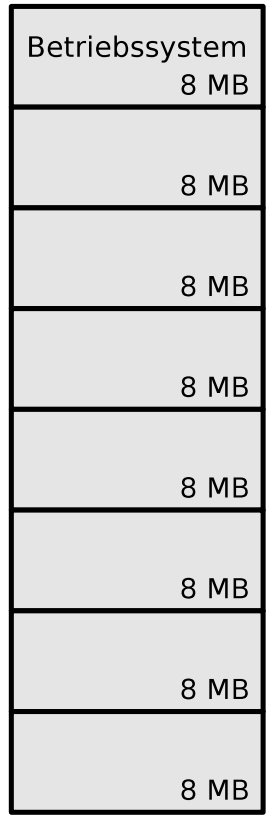
# Konzept 1: Statische Partitionierung

- Der Hauptspeicher wird in **Partitionen gleicher** oder **unterschiedlicher Größe** unterteilt
- Nachteile:
  - Es kommt zwangsläufig zu **interner Fragmentierung** $\Longrightarrow$ ineffizient
    - Das Problem wird durch Partitionen unterschiedlicher Größe abgemildert, aber nicht gelöst
  - Anzahl der Partitionen limitiert die Anzahl möglicher Prozesse
- Herausforderung: Ein Prozess benötigt mehr Speicher, als eine Partition groß ist
  - Dann muss der Prozess so implementiert sein, dass nur ein Teil des Programmcodes im Hauptspeicher liegt
    - Beim Nachladen von Programmcode (Modulen) kommt es zum *Overlay* $\Longrightarrow$ andere Module und Daten werden eventuell überschrieben

IBM OS/360 MFT in den 1960er Jahren nutzte statische Partitionierung

# Statische Partitionierung (1/2)

- Werden **Partitionen gleicher Größe** verwendet, ist es egal, welche freie Partition ein Prozess zugewiesen wird
  - Sind alle Partitionen belegt, muss ein Prozess aus dem Hauptspeicher verdrängt werden
    - Die Entscheidung, welcher Prozess verdrängt wird, hängt vom verwendeten Scheduling-Verfahren ($\Longrightarrow$ Foliensatz 8) ab

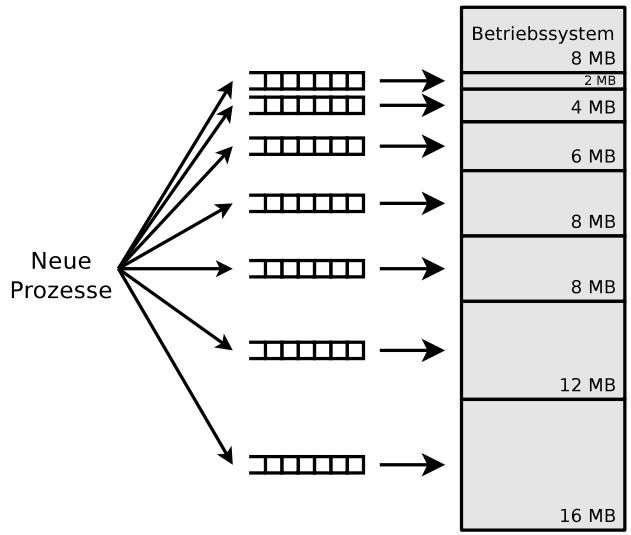


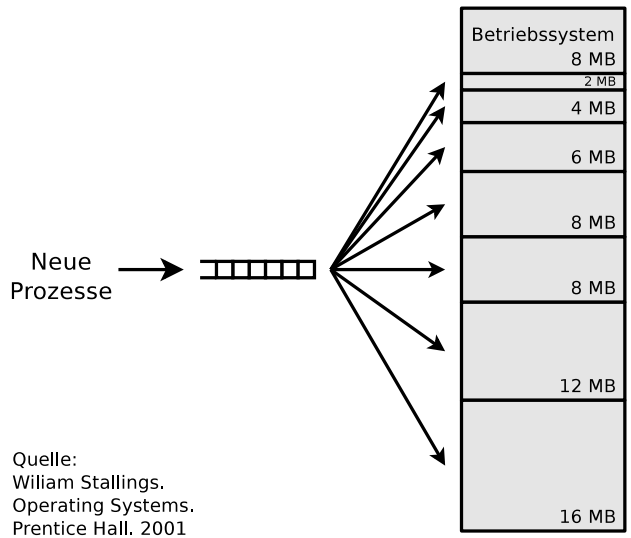

Partitionen gleicher Größe

Quelle:
Wiliam Stallings.
Operating Systems.
Prentice Hall. 2001

# Statische Partitionierung (2/2)

- Prozesse sollen eine möglichst passgenaue Partition erhalten
  - Ziel: Wenig interne Fragmentierung
- Werden **Partitionen unterschiedlicher Größe** verwendet, gibt es 2 Möglichkeiten, um Prozessen Partitionen zuzuweisen:
  1. **Eine eigene Prozess-Warteschlange für jede Partition**
     - Nachteil: Bestimmte Partitionen werden eventuell nie genutzt
  2. **Eine einzelne Warteschlange für alle Partitionen**
     - Die Zuweisung der Partitionen an Prozesse ist flexibler möglich
     - Auf veränderte Anforderungen der Prozesse kann rasch reagiert werden

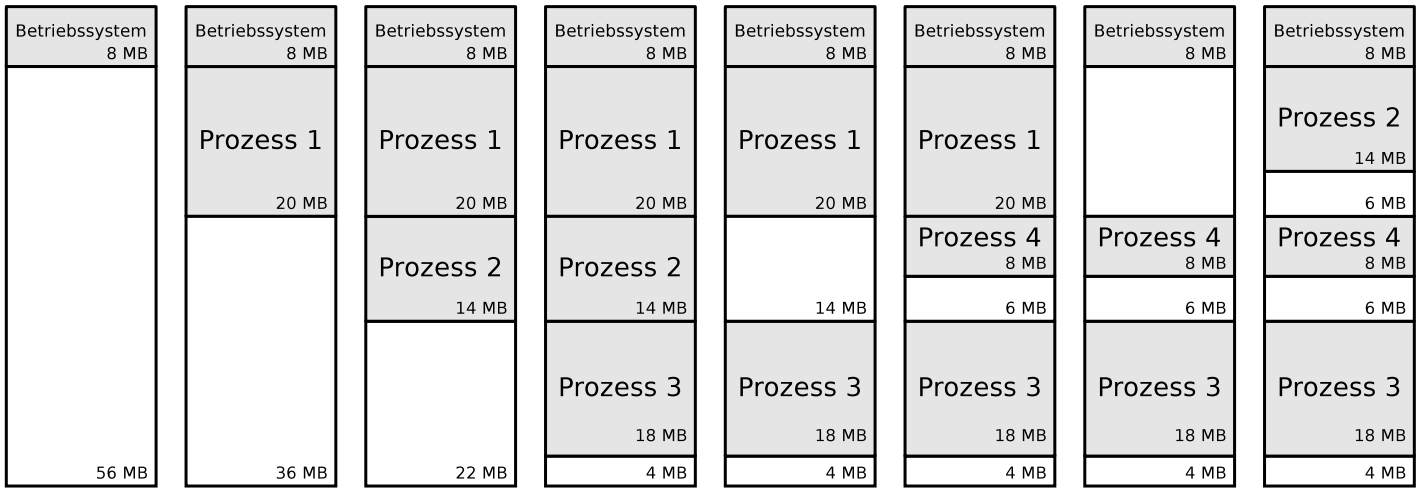



Quelle:
Wiliam Stallings.
Operating Systems.
Prentice Hall. 2001

# Konzept 2: Dynamische Partitionierung

- Jeder Prozess erhält im Hauptspeicher eine zusammenhängende Partition mit exakt der notwendigen Größe



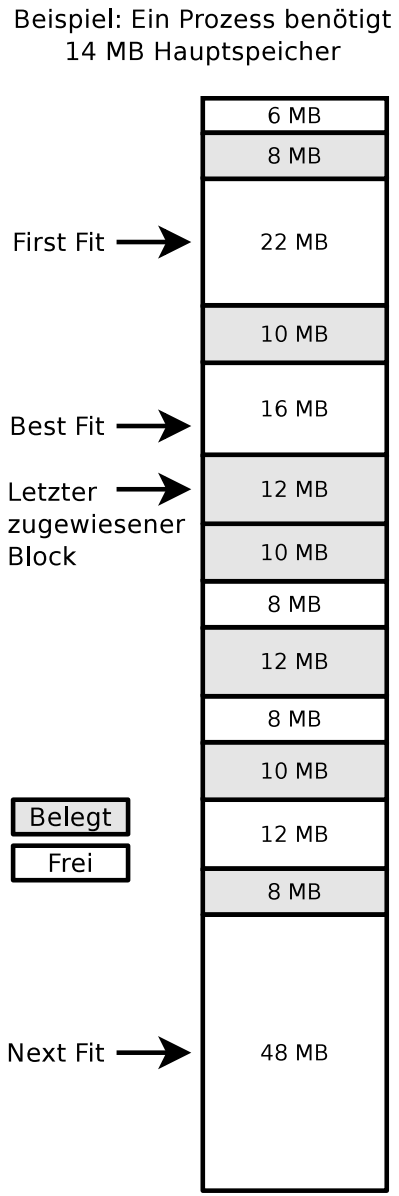

Quelle:
Wiliam Stallings.
Operating Systems.
Prentice Hall. 2001

- Es kommt zwangsläufig zu **externer Fragmentierung** $\Longrightarrow$ ineffizient
  - Mögliche Lösung: Defragmentierung
    - Voraussetzung: Verschiebbarkeit von Speicherblöcken
    - Verweise in Prozessen dürfen durch ein Verschieben von Partitionen nicht ungültig werden

IBM OS/360 MVT nutzte in den 1960er Jahren dynamische Partitionierung

# Realisierungskonzepte für dynamische Partitionierung

- **First Fit**
  - Sucht ab dem Anfang des Adressraums einen passenden freien Block
  - Schnelles Verfahren
- **Next Fit**
  - Sucht ab der letzten Zuweisung einen passenden freien Block
  - Zerstückelt schnell den großen Bereich freien Speichers am Ende des Adressraums
- **Best Fit**
  - Sucht den freien Block, der am besten passt
  - Produziert viele Minifragmente und ist langsam



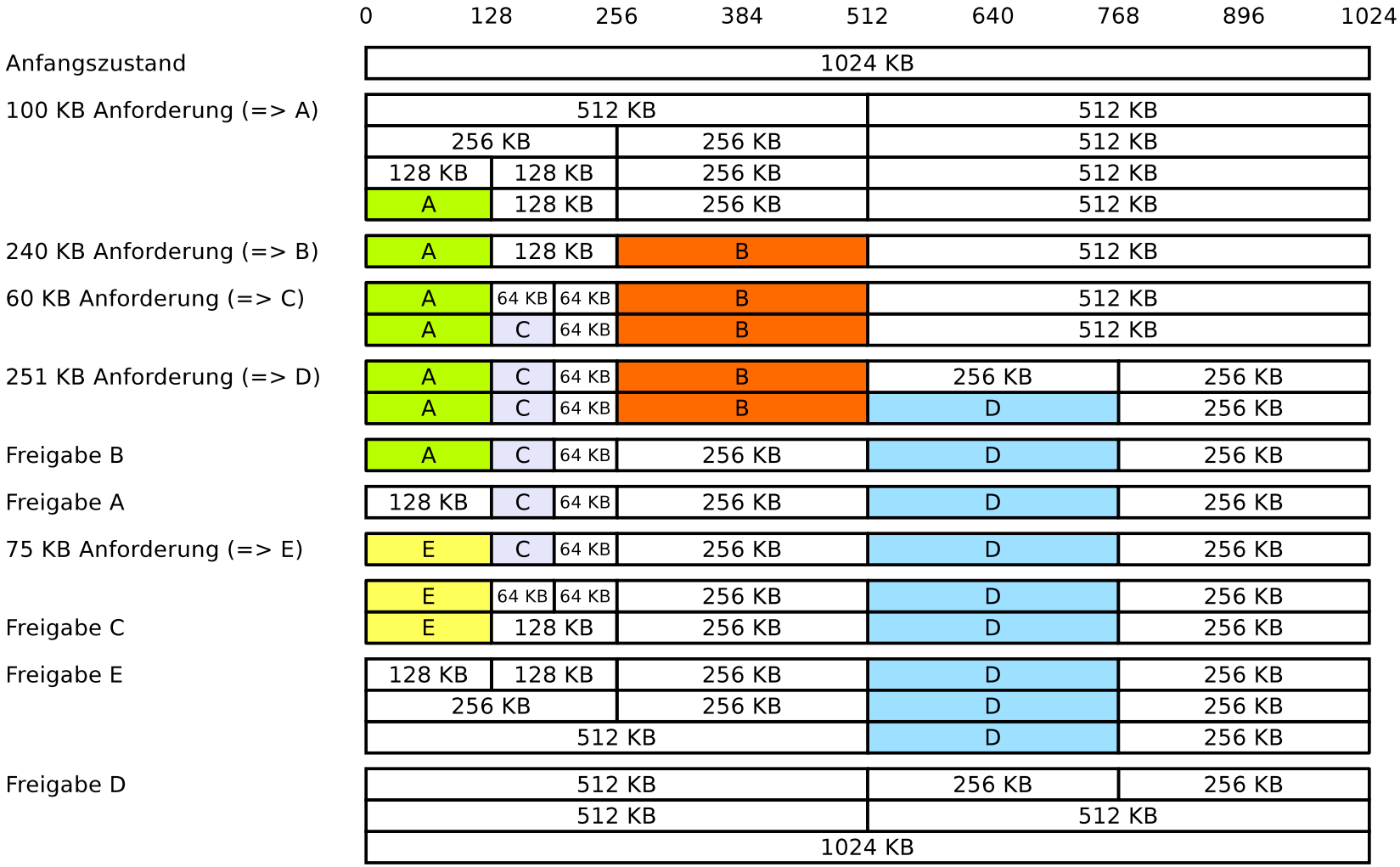
# Konzept 3: Buddy-Speicherverwaltung von Donald Knuth

- Zu Beginn gibt es nur einen Block, der den gesamten Speicher abdeckt
- Fordert ein Prozess einen Speicher an, wird zur nächsthöheren Zweierpotenz aufgerundet und ein entsprechender, freier Block gesucht
  - Existiert kein Block dieser Größe, wird nach einem Block doppelter Größe gesucht und dieser in 2 Hälften (sogenannte *Buddies*) unterteilt
    - Der erste Block wird dann dem Prozess zugewiesen
  - Existiert auch kein Block doppelter Größe, wird ein Block vierfacher Größe gesucht, usw. . .
- Wird Speicher freigegeben, wird geprüft, ob 2 Hälften gleicher Größe sich wieder zu einem größeren Block zusammenfassen lassen
  - Es werden nur zuvor vorgenommene Unterteilungen rückgängig gemacht!

**Buddy-Speicherverwaltung in der Praxis**

- Der Linux-Kernel verwendet eine Variante der Buddy-Speicherverwaltung für die Zuweisung der Seiten
- Das Betriebssystem verwaltet für jede möglich Blockgröße eine „Frei-Liste“

# Beispiel zum Buddy-Verfahren




- Nachteil: Interner und externer Verschnitt (Fragmentierung)

# Informationen zur Fragmentierung des Speichers

- Die DMA-Zeile zeigt die ersten 16 MB im System
- Die DMA32-Zeile zeigt den Speicher $>$ 16 MB und $<$ 4 GB im System
- Die Normal-Zeile zeigt den Speicher $>$ 4 GB im System

Weitere Information zu den Zeilen: `https://utcc.utoronto.ca/~cks/space/blog/linux/KernelMemoryZones`

```
# cat /proc/buddyinfo
Node 0, zone      DMA      1      1      1      0      2      1      1      0      1      1      3
Node 0, zone    DMA32    208    124   1646    566    347    116    139    115     17      4    212
Node 0, zone   Normal     43     62    747    433    273    300    254    190     20      8    287
```

- Spalte 1 $\Longrightarrow$ Anzahl freier Blöcke („*Buddies*“) der Größe $2^0 * PAGESIZE \Longrightarrow 4\,\text{kB}$
- Spalte 2 $\Longrightarrow$ Anzahl freier Blöcke („*Buddies*“) der Größe $2^1 * PAGESIZE \Longrightarrow 8\,\text{kB}$
- Spalte 3 $\Longrightarrow$ Anzahl freier Blöcke („*Buddies*“) der Größe $2^2 * PAGESIZE \Longrightarrow 16\,\text{kB}$
- ...
- Spalte 11 $\Longrightarrow$ Anzahl freier Blöcke („*Buddies*“) der Größe $2^{10} * PAGESIZE \Longrightarrow 4096\,\text{kB} = 4\,\text{MB}$

PAGESIZE $= 4096\,\text{Bytes} = 4\,\text{kB}$
Die Seitengröße eines Linux-Systems gibt folgendes Kommando aus: `$ getconf PAGESIZE`

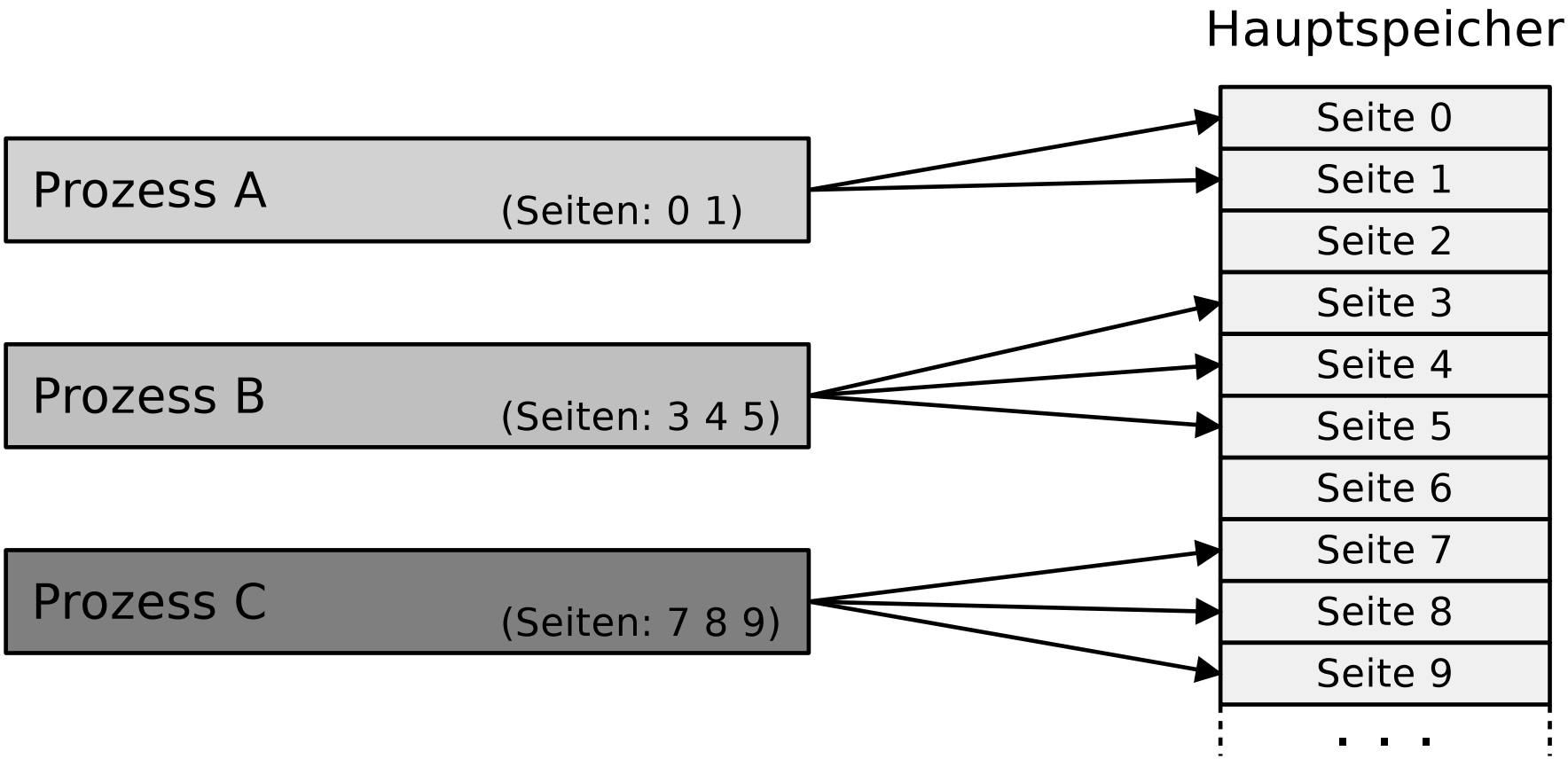
# Speicheradressierung

- Auf 16 Bit-Architekturen sind $2^{16}$ Speicheradressen und damit bis zu 65.536 Byte, also **64 kB** adressierbar
- Auf 32 Bit-Architekturen sind $2^{32}$ Speicheradressen und damit bis zu 4.294.967.296 Byte, also **4 GB** adressierbar
- Auf 64 Bit-Architekturen sind $2^{64}$ Speicheradressen und damit bis zu 18.446.744.073.709.551.616 Byte, also **16 Exabyte** adressierbar

**! ! ! Frage ! ! !**

Wie greifen Prozesse auf den Speicher zu?

# Idee: Direkter Zugriff auf den Speicher




- Naheliegende Idee: Direkter Speicherzugriff durch die Prozesse
  $\Longrightarrow$ **Real Mode**
- Leider unmöglich in Multitasking-Systemen

# Real Mode (Real Address Mode)

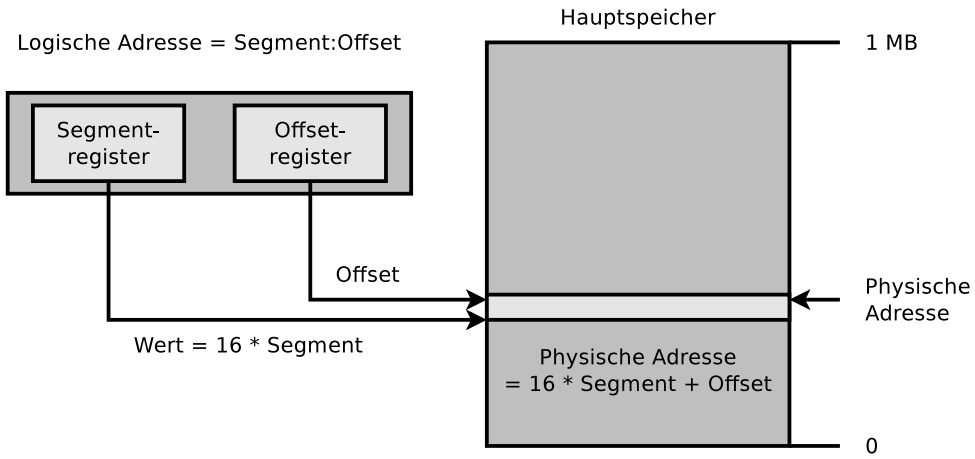
- Betriebsart x86-kompatibler Prozessoren
- **Kein Zugriffsschutz**
  - Jeder Prozess kann auf den gesamten adressierbaren Speicher zugreifen
    - Inakzeptabel für Multitasking-Betriebssysteme
- **Maximal 1 MB Hauptspeicher adressierbar**
  - Maximaler Speicherausbau eines Intel 8086
  - Grund: Der Adressbus des 8088 verfügt nur über 20 Adressleitungen
    - 20 Busleitungen $\Longrightarrow$ 20 Bits lange Speicheradressen $\Longrightarrow$ Die CPU kann $2^{20}$ = ca. 1 MB Speicher adressieren
  - Nur die ersten 640 kB (*unterer Speicher*) für das Betriebssystem (MS-DOS) und die Programme zur Verfügung
    - Die restlichen 384 kB (*oberer Speicher*) enthalten das BIOS der Grafikkarte, das Speicherfenster zum Grafikkartenspeicher und das BIOS ROM des Mainboards
- Die Bezeichnung „Real Mode“ wurde mit dem Intel 80286 eingeführt
  - Im Real Mode greift die CPU wie ein 8086 auf den Hauptspeicher zu
  - Jede x86-kompatible CPU startet im Real Mode

Interessante Quelle: `http://www.internals.com/articles/protmode/realmode.htm`

# Real Mode – Adressierung

- Der Hauptspeicher ist in 65.536 Segmente unterteilt
  - Die Speicheradressen sind 16 Bits lang
  - Jedes Segment ist 64 Bytes ($= 2^{16} = 65.536$ Bits) groß
- Adressierung des Hauptspeichers via Segment und Offset
  - Zwei 16 Bits lange Werte, die durch einen Doppelpunkt getrennt sind `Segment:Offset`
  - Segment und Offset werden in den zwei 16-Bits großen Registern **Segmentregister** (= **Basisadressregister**) und **Offsetregister** (= **Indexregister**) gespeichert

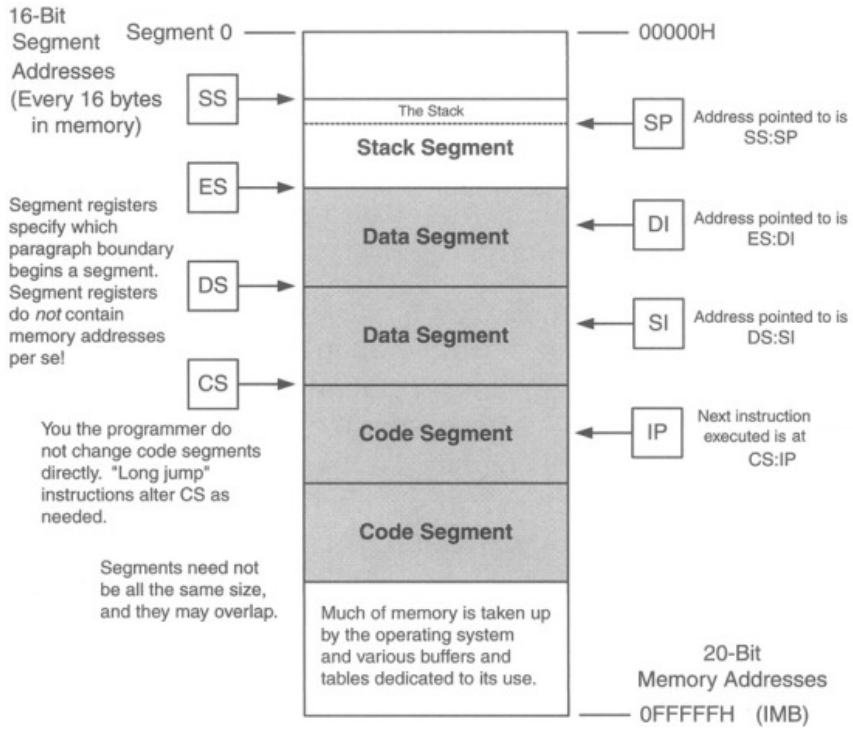
- Das **Segmentregister** speichert die Nummer des Segments
- Das **Offsetregister** zeigt auf eine Adresse zwischen 0 und $2^{16}$ (=65.536) relativ zur Adresse im Segmentregister



# Real Mode – Segmentregister seit 8086

Bildquelle: `http://www.c-jump.com`

- Beim 8086 existieren 4 **Segmentregister**
- `CS` (Code Segment)
  - Segment mit dem Quelltext des Programms
- `DS` (Data Segment)
  - Segment mit globalen Daten, des aktuellen Programms
- `SS` (Stack Segment)
  - Segment mit dem Stack für die lokalen Daten des Programms
- `ES` (Extra Segment)
  - Segment für weitere Daten

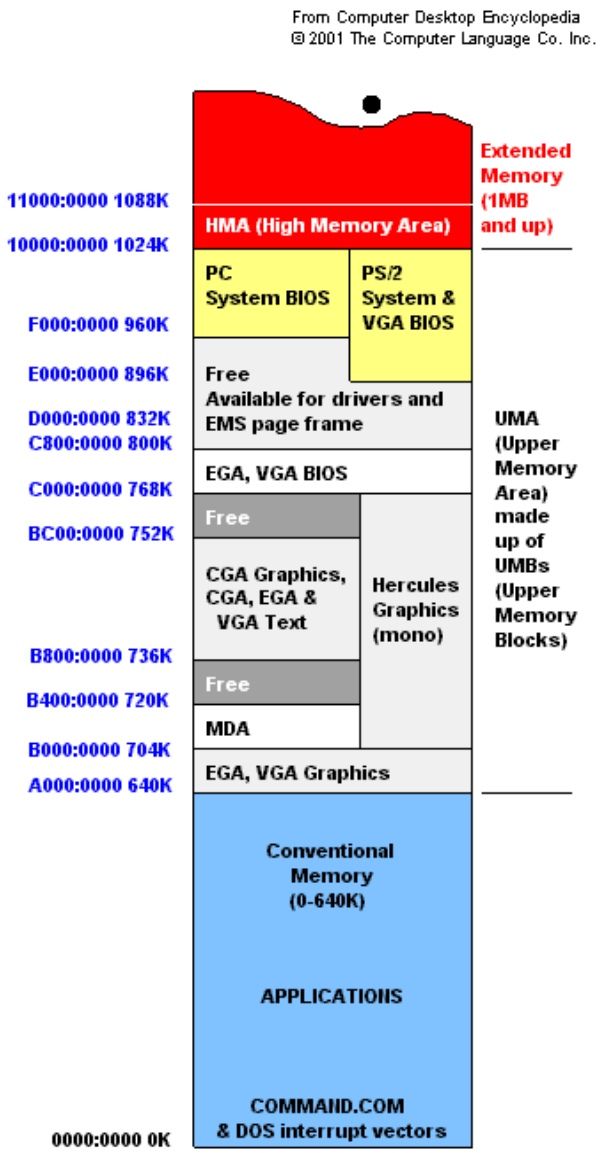



- Ab dem Intel 80386 existieren 2 weitere Segmentregister (`FS` und `GS`) für zusätzliche Extra-Segmente
- Die Segmentbereiche realisieren einen einfachen **Speicherschutz**

# Real Mode bei MS-DOS

Bildquelle: Google Bildersuche

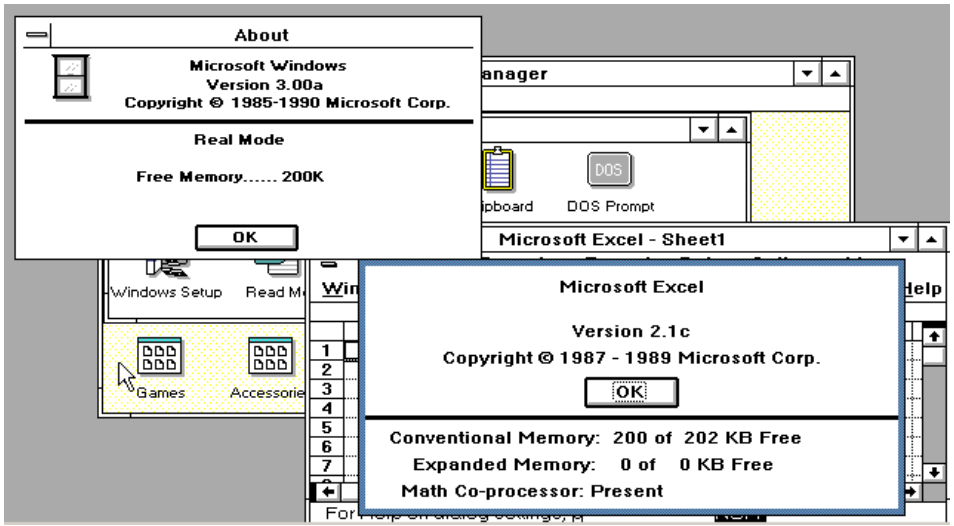
```
A:\>mem

Memory Type        Total       Used       Free
---------------  --------   --------   --------
Conventional         640K        92K       548K
Upper                  0K         0K         0K
Reserved             384K       384K         0K
Extended (XMS)   742,400K        64K   742,336K
---------------  --------   --------   --------
Total memory     743,424K       540K   742,884K

Total under 1 MB     640K        92K       548K

Largest executable program size       548K (561,552 bytes)
Largest free upper memory block         0K       (0 bytes)
MS-DOS is resident in the high memory area.
```

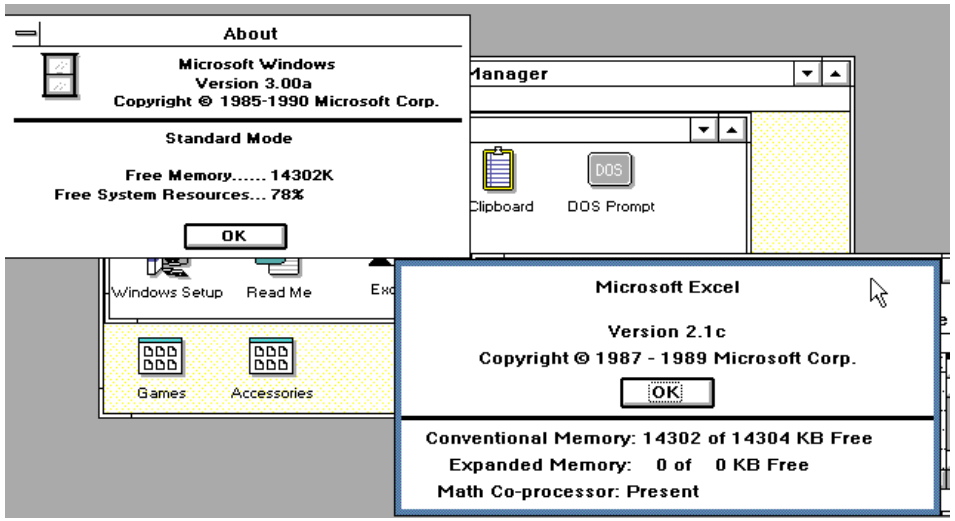



- Real Mode ist der Standardmodus für MS-DOS und dazu kompatible Betriebssysteme (u.a. PC-DOS, DR-DOS und FreeDOS)

# Real Mode bei Microsoft Windows

- Neuere Betriebssysteme verwenden ihn nur noch während der Startphase und schalten dann in den **Protected Mode** um




- Windows 2.0 läuft nur im Real Mode
- Windows 2.1 und 3.0 können entweder im Real Mode oder im Protected Mode laufen
- Windows 3.1 und spätere Versionen laufen nur im Protected Mode

Bildquelle: `http://virtuallyfun.superglobalmegacorp.com`

# Anforderungen an die Speicherverwaltung

- **Relokation**
  - Werden Prozesse aus dem Hauptspeicher verdrängt, ist nicht bekannt, an welcher Stelle sie später wieder in den Hauptspeicher geladen werden
  - Erkenntnis: Prozesse dürfen keine Referenzen auf physische Speicheradressen enthalten
- **Schutz**
  - Speicherbereiche müssen geschützt werden vor unbeabsichtigtem oder unzulässigem Zugriff durch anderen Prozesse
  - Erkenntnis: Zugriffe müssen (durch die CPU) überprüft werden
- **Gemeinsame Nutzung**
  - Trotz Speicherschutz muss eine Kooperation der Prozesse mit gemeinsamem Speicher (Shared Memory) möglich sein $\Longrightarrow$ Foliensatz 10
- **Vergrößerte Kapazität**
  - 1 MB ist nicht genug
  - Es soll mehr Speicher verwendet werden können, als physisch existiert
  - Erkenntnis: Ist der Hauptspeicher voll, können Daten ausgelagert werden
- Lösung: **Protected mode** und **virtueller Speicher**

# Protected Mode (Schutzmodus)

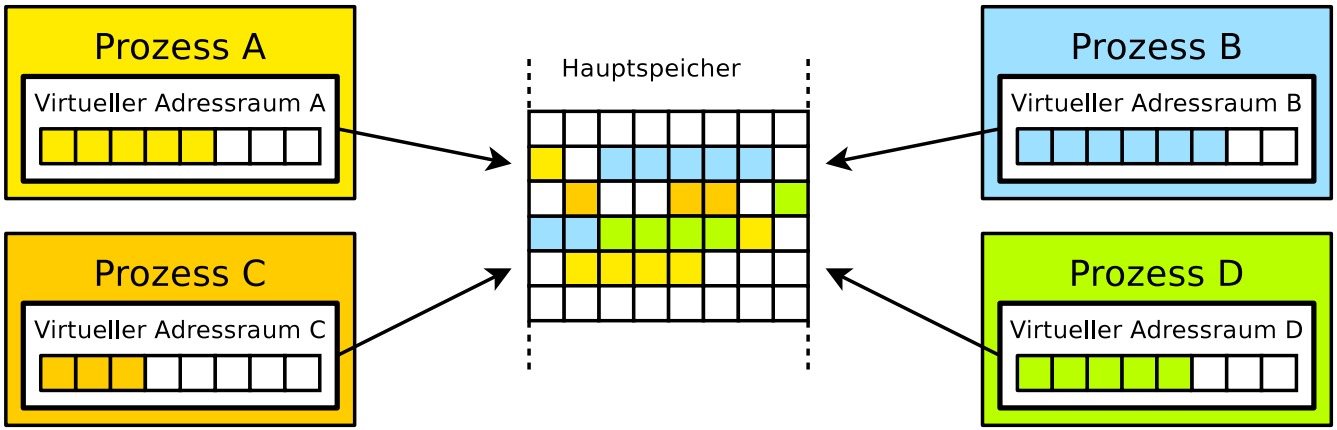
- Betriebsart x86-kompatibler Prozessoren
  - Eingeführt mit dem Intel 80286
- Erhöht die Menge des adressierbaren Speichers
  - 16-Bit Protected Mode beim 80286 $\Longrightarrow$ 16 MB Hauptspeicher
  - 32-Bit Protected Mode beim 80386 $\Longrightarrow$ 4 GB Hauptspeicher
- Realisiert **virtuellen Speicher**
  - Jeder Prozess läuft in seiner eigenen, von anderen Prozessen abgeschotteten Kopie des physischen Adressraums
    - Jeder Prozess darf nur auf seinen eigenen virtuellen Speicher zugreifen
  - Mit der Memory Management Unit (MMU) bekommen Prozesse Adressrbereiche wie im Real Mode bereitgestellt
    - Virtuelle Speicheradressen übersetzt die CPU mit Hilfe der MMU in physische Speicheradressen
- x86-CPUs enthalten 4 Privilegienstufen ($\Longrightarrow$ Foliensatz 7) für Prozesse
  - Ziel: Speicherschutz realisieren um Stabilität und Sicherheit zu erhöhen

# Virtueller Speicher (1/3)

- Moderne Betriebssysteme arbeiten im Protected Mode (Schutzmodus)
- Im Protected Mode unterstützt die CPU 2 Methoden zur Speicherverwaltung
  - **Segmentierung** existiert ab dem 80286
  - **Paging** existiert ab dem 80386
  - Beide Verfahren sind Implementierungsvarianten des **virtuellen Speichers**
- Prozesse verwenden keine physischen Hauptspeicheradressen
  - Das würde bei Multitasking-Systemen zu Problemen führen
- Stattdessen besitzt jeder Prozess einen eigenen **Adressraum**
  - Der Adressraum ist eine Abstraktion des physischen Speichers
    - Es handelt sich dabei um **virtuellen Speicher**
    - Er ist unabhängig von der verwendeten Speichertechnologie und den gegebenen Ausbaumöglichkeiten
    - Er besteht aus logischen Speicheradressen, die von der Adresse 0 aufwärts durchnummeriert sind

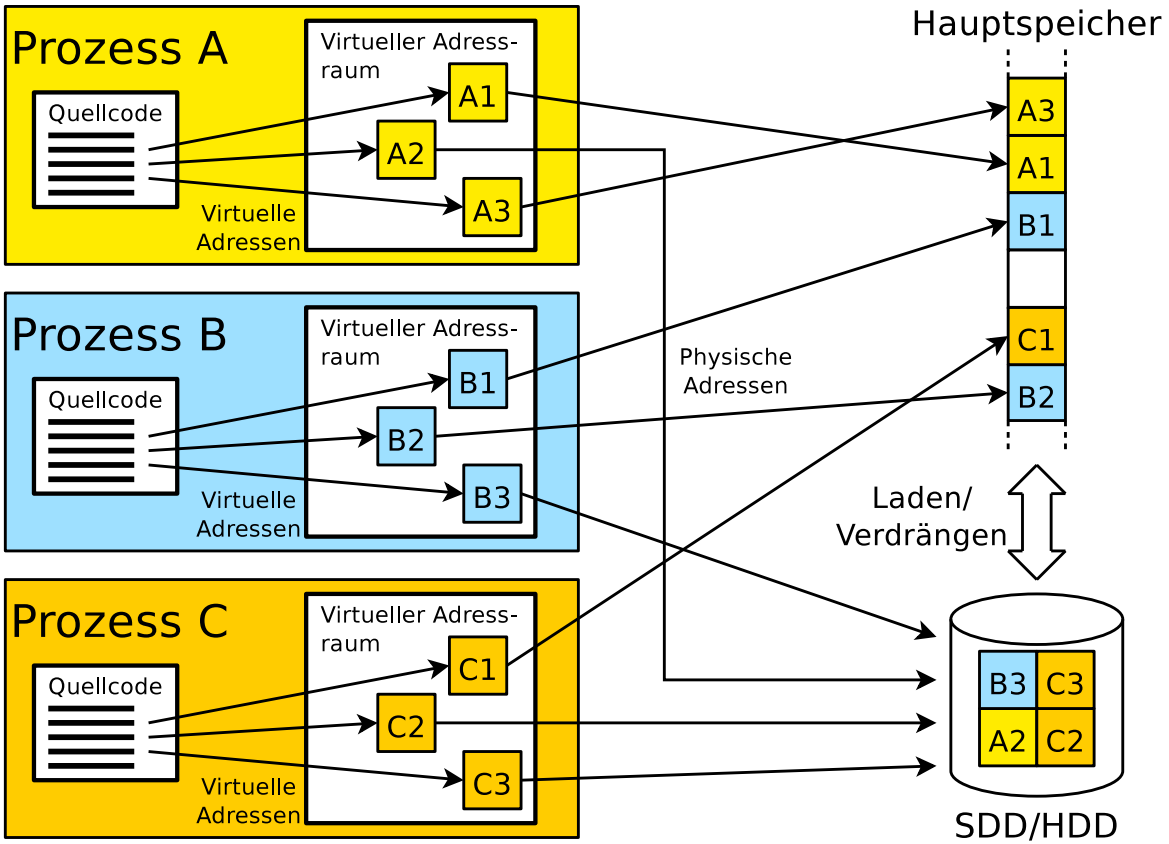
# Virtueller Speicher (2/3)

- Adressräume können nach Bedarf erzeugt oder gelöscht werden und sie sind geschützt
  - Kein Prozess kann ohne vorherige Vereinbarung auf den Adressraum eines anderen Prozesses zugreifen
- **Mapping** = Virtuellen Speicher auf physischen Speicher abbilden




- Dank virtuellem Speicher wird der Hauptspeicher besser ausgenutzt
  - Prozesse müssen nicht am Stück im Hauptspeicher liegen
  - Darum ist die Fragmentierung des Hauptspeichers kein Problem

# Virtueller Speicher (3/3)




- Durch virtuellen Speicher kann mehr Speicher angesprochen und verwendet werden, als physisch im System vorhanden ist
- **Auslagern** (**Swapping**) geschieht für Benutzer und Prozesse transparent

**Das Thema Virtueller Speicher ist anschaulich erklärt bei...**

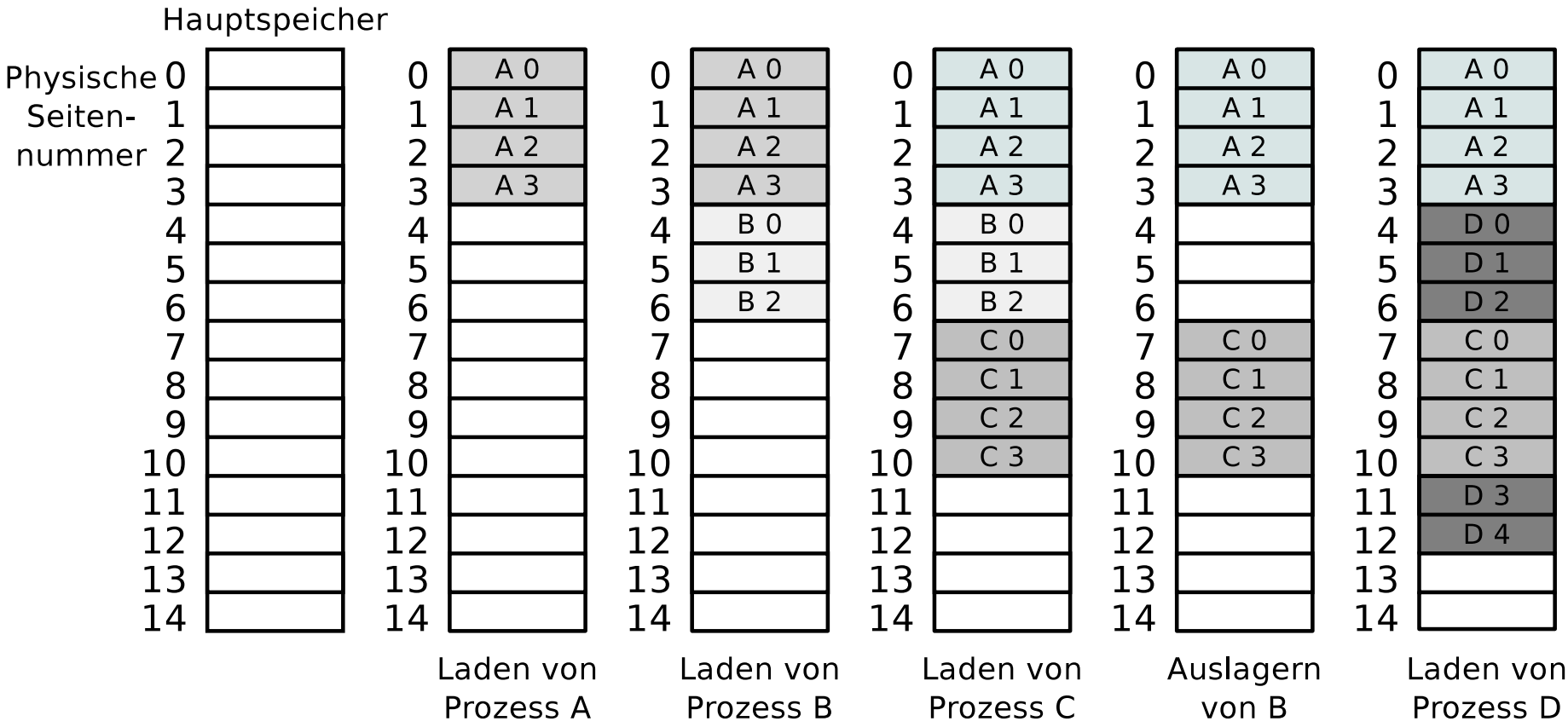
- **Betriebssysteme**, *Carsten Vogt*, 1. Auflage, Spektrum Akademischer Verlag (2001), S. 152

# Paging: Seitenorientierter Speicher

- **Virtuelle Seiten** der Prozesse werden auf **physische Seiten** im Hauptspeicher abgebildet
  - Alle Seiten haben die gleiche Länge
    - Die Seitenlänge ist üblicherweise 4 kb (bei der Alpha-Architektur: 8 kB)
- Vorteile:
  - Keine externe Fragmentierung
  - Interne Fragmentierung kann nur in der letzten Seite jedes Prozesses auftreten
- Das Betriebssystemen verwaltet **für jeden Prozess eine Seitentabelle**
  - In dieser steht, wo sich die einzelnen Seiten des Prozesses befinden
- Prozesse arbeiten nur mit **virtuellen Speicheradressen**
  - Virtuelle Speicheradressen bestehen aus 2 Teilen
    - Der werthöhere Teil enthält die Seitennummer
    - Der wertniedrigere Teil enthält den Offset (Adresse innerhalb einer Seite)
  - Die Länge der virtuellen Adressen ist architekturabhängig und darum 16, 32 oder 64 Bits

# Zuweisung von Prozessseiten zu freien physischen Seiten

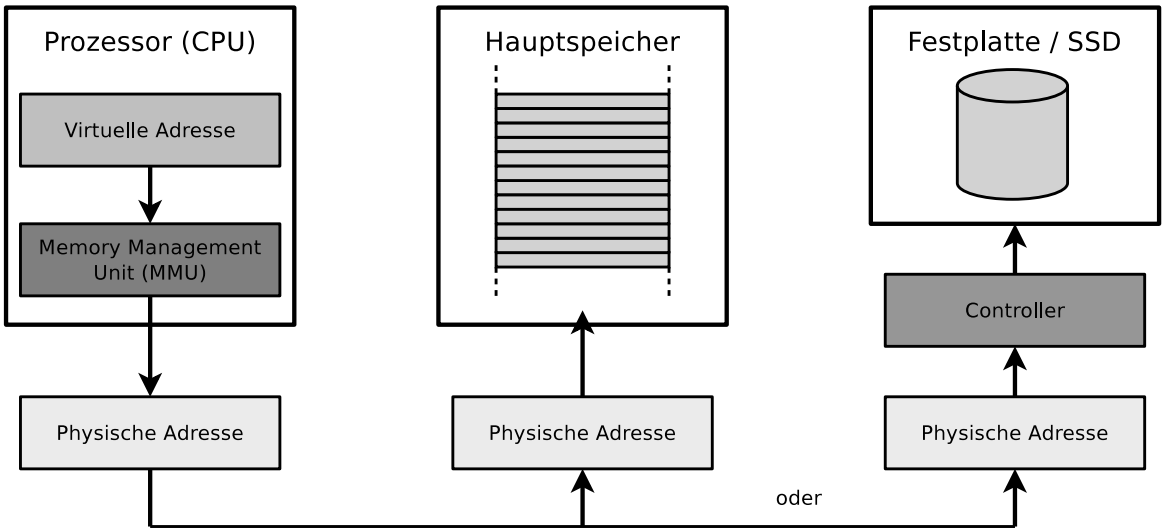
- Prozesse müssen nicht am Stück im Hauptspeicher liegen
  $\Longrightarrow$ Keine externe Fragmentierung




Bildquelle: **Operating Systems**, *William Stallings*, 4.Auflage, Prentice Hall (2001)

# Adressumwandlung durch die Memory Management Unit

- Virtuelle Speicheradressen übersetzt die CPU mit der **MMU** und der Seitentabelle in physische Adressen
  - Das Betriebssystem prüft dann, ob sich die physische Adresse im Hauptspeicher, oder auf der SSD/HDD befindet




- Befinden sich die Daten auf der SSD/HDD, muss das Betriebssystem die Daten in den Hauptspeicher einlesen
- Ist der Hauptspeicher voll, muss das Betriebssystem andere Daten aus dem Hauptspeicher auf die SDD/HDD verdrängen (*swappen*)

**Das Thema MMU ist anschaulich erklärt bei...**

- **Betriebssysteme**, *Carsten Vogt*, 1. Auflage, Spektrum Akademischer Verlag (2001), S. 152-153
- **Moderne Betriebssysteme**, *Andrew S. Tanenbaum*, 2. Auflage, Pearson (2009), S. 223-226

# Implementierung der Seitentabelle

- Die Länge der Seiten hat Auswirkungen:
  - **Kurze Seiten**: Weniger interner Verschnitt, aber längere Seitentabelle
  - **Lange Seiten**: Kürzere Seitentabelle, aber mehr interner Verschnitt
- Seitentabellen liegen im Hauptspeicher

$$\text{Maximale Größe der Seitentabelle} = \frac{\text{Virtueller Adressraum}}{\text{Seitengröße}} * \text{Größe der Seitentabelleneinträge}$$

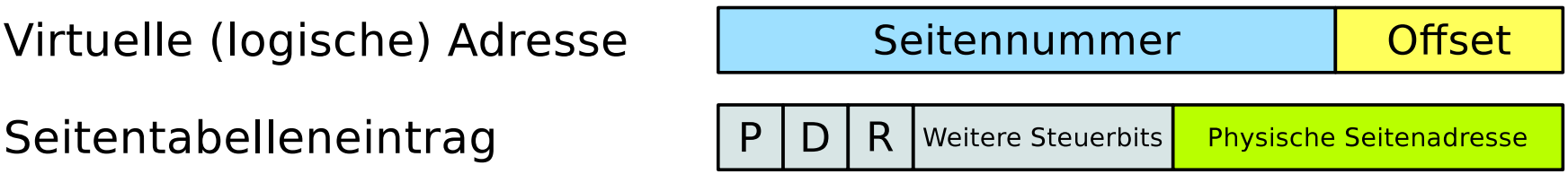
- Maximale Größe der Seitentabellen bei 32 Bit-Betriebssystemen:

$$\frac{4\,\text{GB}}{4\,\text{kB}} * 4\,\text{Bytes} = \frac{2^{32}\,\text{Bytes}}{2^{12}\,\text{Bytes}} * 2^2\,\text{Bytes} = 2^{22}\,\text{Bytes} = 4\,\text{MB}$$

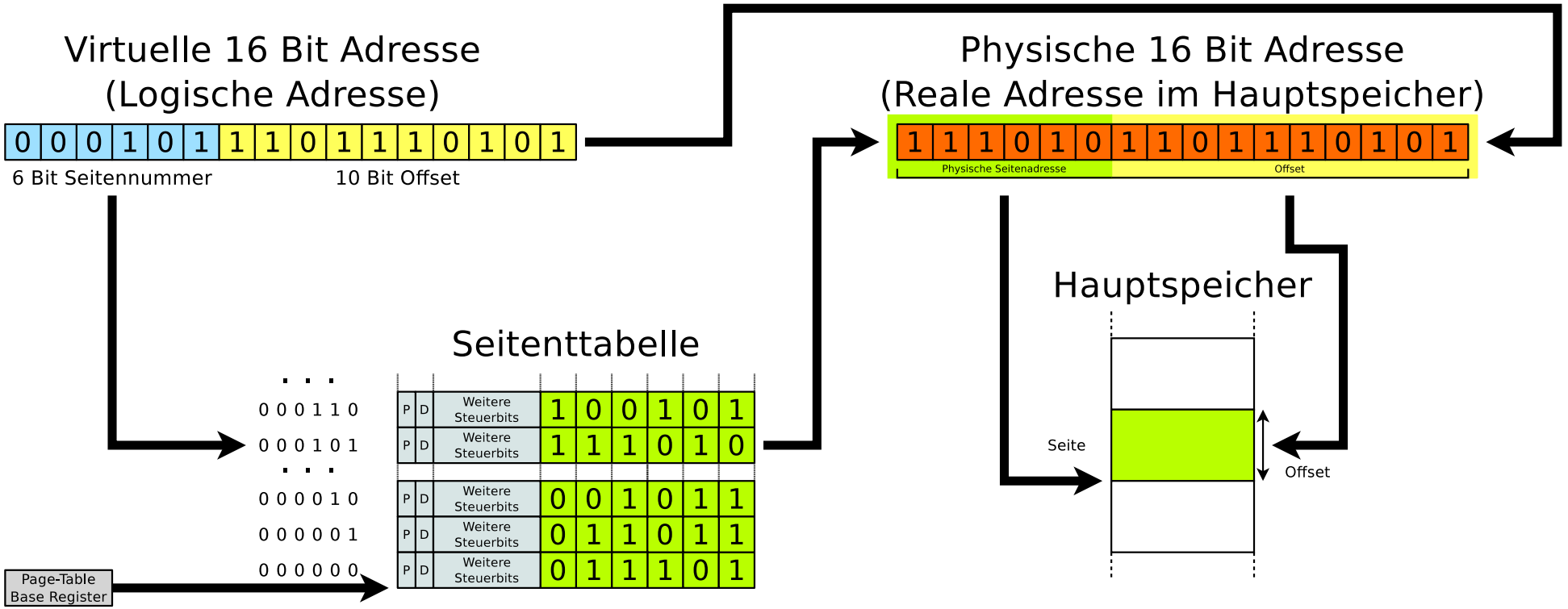
- Jeder Prozess in einem Multitasking-Betriebssystem braucht eine Seitentabelle

# Struktur der Seitentabellen (Page Table)

- Jeder Eintrag in der Seitentabelle enthält u.a.:
  - **Present-Bit**: Gibt an, ob die Seite im Hauptspeicher liegt
  - **Dirty-Bit** (*Modified-Bit*): Gibt an, ob die Seite verändert wurde
  - **Reference-Bit**: Gibt an, ob es einen (auch lesenden!) Zugriff auf die Seite gab $\Longrightarrow$ das ist evtl. wichtig für die verwendete Seitenersetzungsstrategie
  - **Weitere Steuerbits**: Hier ist u.a. festgelegt, ob...
    - Prozesse im Benutzermodus nur lesend oder auch schreibend auf die Seite zugreifen dürfen (**Read/Write-Bit**)
    - Prozesse im Benutzermodus auf die Seite zugreifen dürfen (**User/Supervisor-Bit**)
    - Änderungen sofort (*Write-Through*) oder erst beim verdrängen (*Write-Back*) durchgeschrieben werden (**Write-Through-Bit**)
    - Die Seite in den Cache geladen darf oder nicht (**Cache-Disable-Bit**)
  - **Physische Seitenadresse**: Wird mit dem Offset der virtuellen Adresse verknüpft



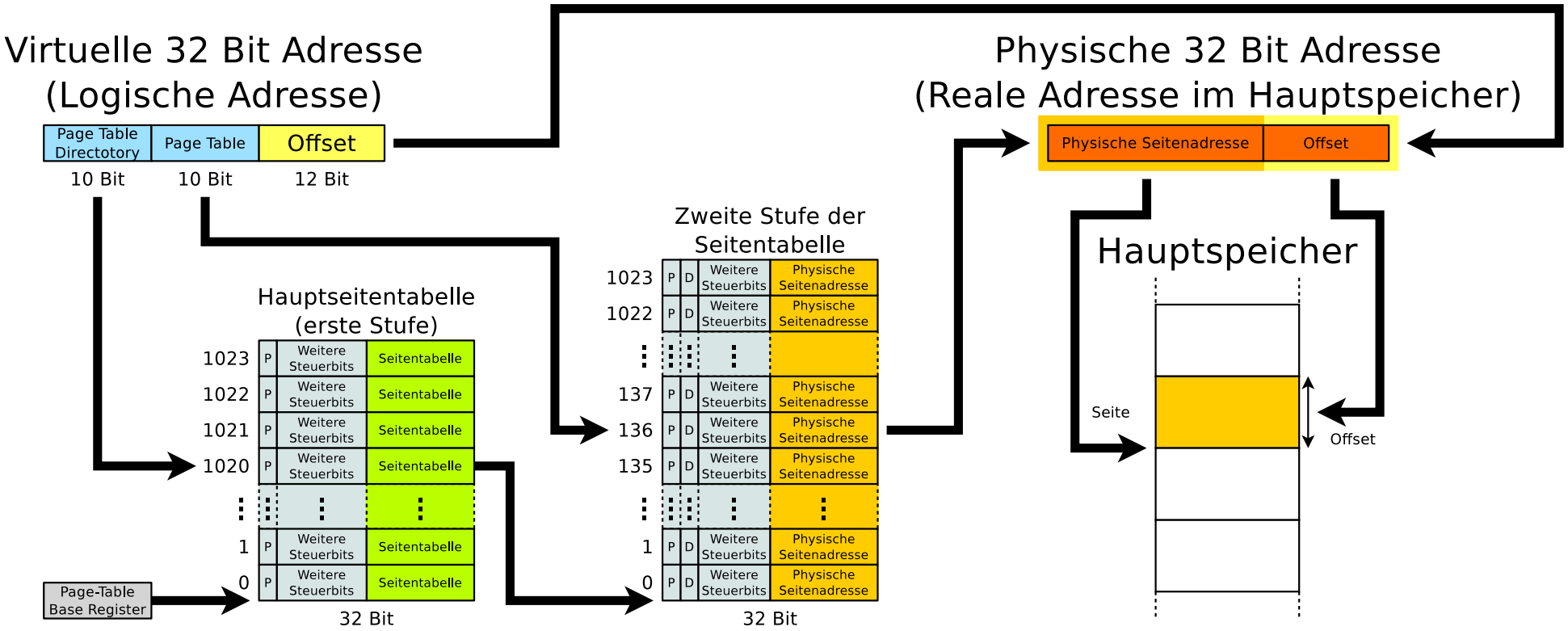
# Adressumwandlung beim Paging (einstufig)




**2 Register ermöglichen der MMU den Zugriff auf die Seitenabelle**

- **Page-Table Base Register** (PTBR): Adresse wo die Seitentabelle des laufenden Prozesses anfängt
- **Page-Table Length Register** (PTLR): Länge der Seitentabelle des laufenden Prozesses

# Adressumwandlung beim Paging (zweistufig)

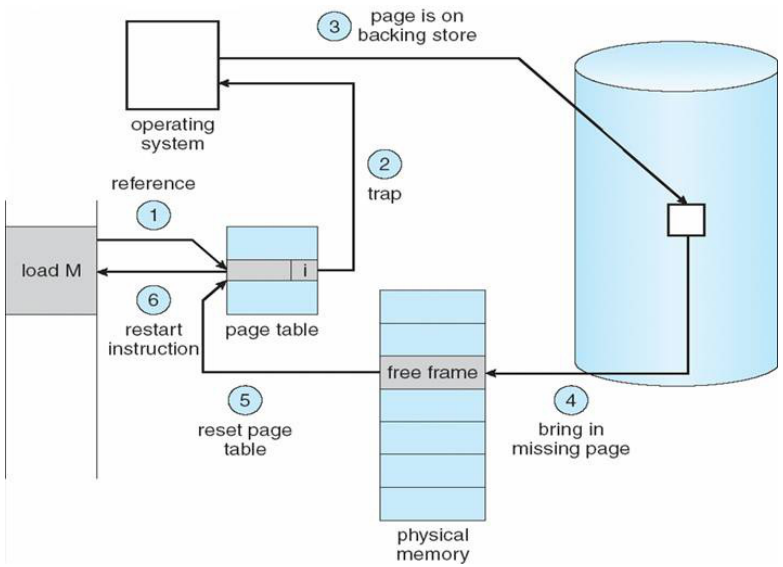



Das Thema Paging ist anschaulich erklärt bei...

- **Betriebssysteme**, *Eduard Glatz*, 2.Auflage, dpunkt (2010), S.450-457
- **Betriebssysteme**, *William Stallings*, 4.Auflage, Pearson (2003), S.394-399
- `http://wiki.osdev.org/Paging`

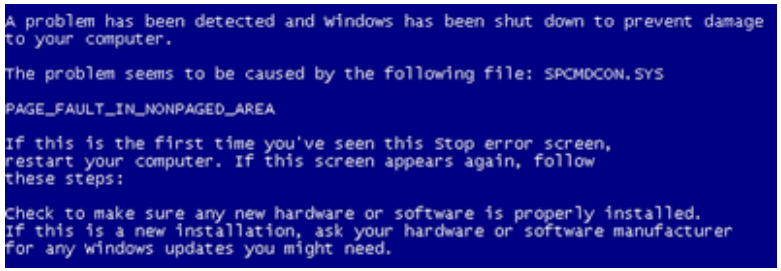
# Page Fault Ausnahme (Exception)

- Ein Programm versucht auf eine Seite zuzugreifen, die nicht im physischen Hauptspeicher ist
  - Das **Present-Bit** in jedem Eintrag der Seitentabelle gibt an, ob die Seite im Hauptspeicher ist oder nicht
- Das Betriebssystem behandelt die Ausnahme mit folgenden Schritten:
  - Daten auf dem Sekundärspeicher (SSD/HDD) lokalisieren
  - Freie Seiten im Hauptspeicher lokalisieren
  - Die Daten in die Seiten laden
  - Seitentabelle aktualisieren
  - Kontrolle an das Programm zurückgeben
    - Dieses führt die Anweisung, die zum Page Fault führte, erneut aus

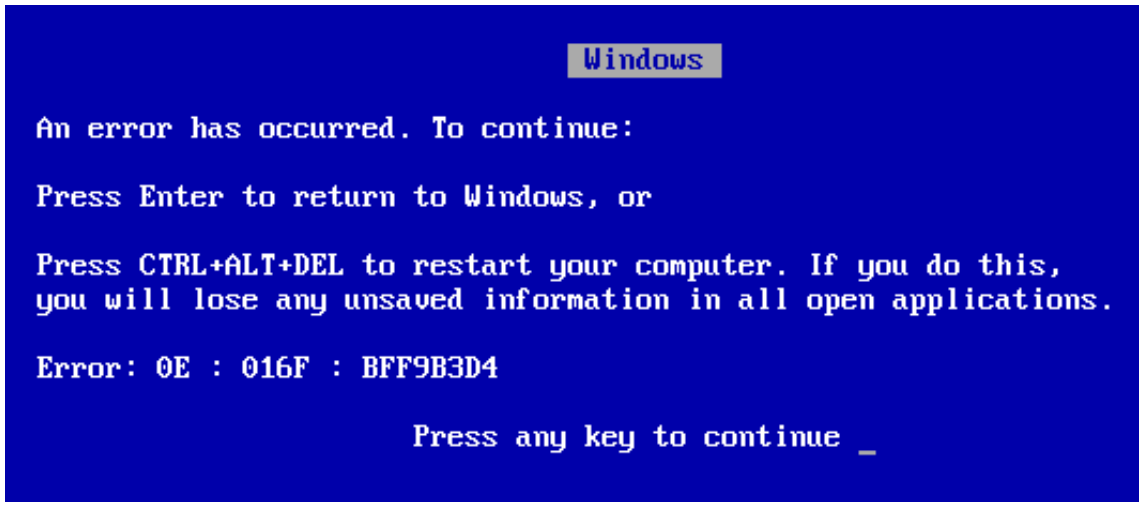



Bildquelle: `http://www.cs.odu.edu/~cs471w/spring10/lectures/virtualmemory.htm`

## Access Violation Ausnahme (Exception) oder General Protection Fault Ausnahme (Exception)




- Heißt auch **Segmentation fault** oder **Segmentation violation**
  - Ein Paging-Problem, das nichts mit Segmentierung zu tun hat!
- Ein Prozess versucht auf eine virtuelle Speicheradresse zuzugreifen, auf die er nicht zugreifen darf $\Longrightarrow$ Crash
  - Beispiel: Ein Prozess versucht in eine Seite zu schreiben, auf die er nur lesend zugreifen darf



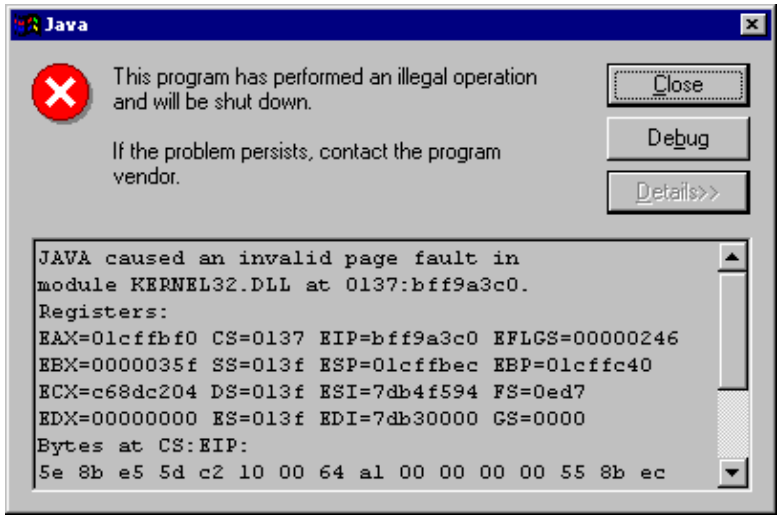




Image source: Wikipedia, `http://telcontar.net/store/archive/CrashGallery/images/crash/m/crash13.png` and `http://www.dtec-computers.com/images/jpg/computer_repair/blue-screen-of-death.gif`

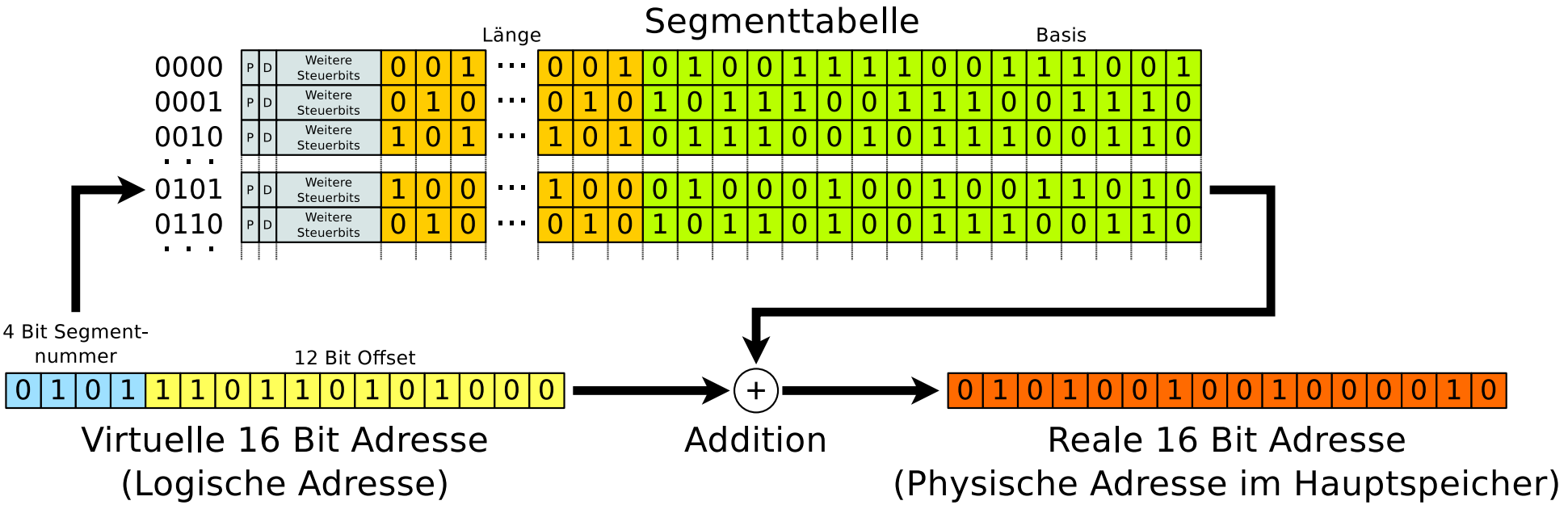
# Segmentierung

- Weitere Methode um virtuellen Speicher zu verwalten
- Der virtuelle Speicher der Prozesse besteht aus **Segmenten** unterschiedlicher Länge

Die maximale Segmentlänge beschreibt das Beispiel auf der nächsten Folie

- Das Betriebssystemen verwaltet **für jeden Prozess eine Segmenttabelle**
  - Jeder Eintrag der Segmenttabelle enthält die Länge des Segments und seine Startadresse im Hauptspeicher
  - Virtuelle Adressen der Prozesse werden mit Hilfe der Segmenttabellen in physische Adressen umgerechnet
- Keine interne Fragmentierung
- Externe Fragmentierung entsteht wie bei dynamischer Partitionierung
  - Diese ist aber nicht so ausgeprägt

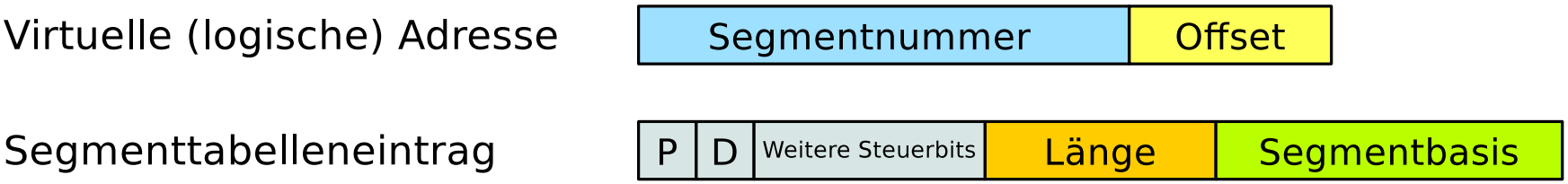
# Adressumwandlung bei Segmentierung




- Die **maximale Segmentlänge** ist festgelegt durch die Länge des Offsets der virtuellen Adressen
  - Im Beispiel ist der Offset 12 Bits lang
    $\Longrightarrow$ maximale Segmentlänge $= 2^{12} = 4.096$ Bits $= 512$ Bytes

# Segmenttabellenstruktur

- Jeder Eintrag in der Segmenttabelle enthält u.a.:
  - **Present-Bit**: Gibt an, ob das Segment im Hauptspeicher ist
  - **Modify-Bit** (*Accessed-Bit* oder *Modified-Bit*): Gibt an, ob das Segment verändert wurde
  - **Weitere Steuerbits**: u.a. sind hier die Zugriffsrechte festgelegt
  - **Länge**: Länge des Segments
  - **Segmentbasis**: Wird mit dem Offset der virtuellen Adresse verknüpft



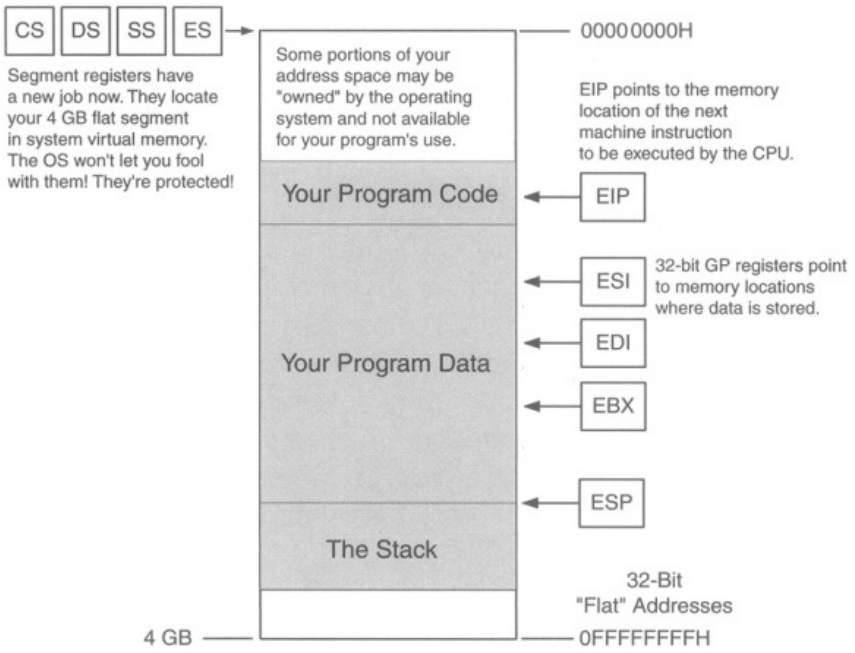

- Versucht ein Programm auf ein Segment zuzugreifen, das nicht im Hauptspeicher liegt, löst das eine **Segment not present**-Ausnahme aus
  - Das **Present-Bit** in jedem Eintrag der Segmenttabelle sagt aus, ob das Segment im Hauptspeicher liegt, oder nicht

# Wiederholung: Real Mode und Protected Mode

- **Real Mode**
  - Betriebsart x86-kompatibler Prozessoren
  - Die CPU greift wie ein Intel 8086 auf den Hauptspeicher zu
  - Kein Zugriffsschutz
    - Jeder Prozess kann auf den gesamten Hauptspeicher zugreifen
- **Protected Mode (Schutzmodus)**
  - Betriebsart x86-kompatibler Prozessoren
  - Realisiert **virtuellen Speicher**
  - **16-Bit Protected Mode** (Wurde mit dem 80286 eingeführt)
    - Ausschließlich **Segmentierung**
    - Über den 24 Bit Adressbus können maximal $2^{24}$ Bytes = 16 MB physischer Hauptspeicher angesprochen werden
    - Aufteilung des Speichers in Segmente von je maximal $2^{16}$ Bytes = 64 kB
  - **32-Bit Protected Mode** (Wurde mit dem 80386 eingeführt)
    - Aufteilung des Speichers in **Segmente** von je maximal $2^{32}$ Bytes = 4 GB
    - **Paging** kann der Segmentierung nachgeschaltet werden
    - Maximal 4 GB physischer Hauptspeicher können angesprochen werden

# Aktueller Stand beim virtuellen Speicher (1/2)

- Moderne Betriebssysteme (für x86) arbeiten im Protected Mode und verwenden ausschließlich Paging
  - Segmentierung wird nicht mehr verwendet
- Diese Arbeitsweise heißt **Flat Memory-Modell**
  - Daten-, Code-, Extra- und Stacksegment decken den gesamten Adressraum ab
  - Damit ist der komplette Adressraum jedes Prozesses über den Offset adressierbar
    - Segmentierung bietet somit keinen Speicherschutz mehr
    - Das ist aber kein Problem wegen des nachgeschalteten Pagings




Bildquelle: `http://www.c-jump.com`

- Vorteil: Betriebssysteme können leichter auf andere CPU-Architekturen ohne Segmentierung portiert werden

# Aktueller Stand beim virtuellen Speicher (2/2)

- Einige Architekturen:
  - IA32 (siehe Folie 30)
    - Zweistufige Seitentabelle
    - Länge virtueller Adressen: 32 Bits
    - 10+10+12 $\Longrightarrow$ 10 Bits für die 2 Seitentabellen plus 12 Bits Offset
  - IA32 mit Physical Address Extension (PAE) $\Longrightarrow$ Pentium Pro
    - Dreistufige Seitentabelle
    - Länge virtueller Adressen: 32 Bits
    - 2+9+9+12 $\Longrightarrow$ 2 Bits für die erste Seitentabelle und je 9 Bits für die 2 weiteren Seitentabellen plus 12 Bits Offset
  - PPC64
    - Dreistufige Seitentabelle
    - Länge virtueller Adressen: 41 Bits
    - 10+10+9+12 $\Longrightarrow$ Je 10 Bits für die ersten beiden Seitentabellen, 9 Bits für die dritte Seitentabelle plus 12 Bits Offset
  - AMD64 (x86-64)
    - Vierstufige Seitentabelle
    - Länge virtueller Adressen: 48 Bits
    - 9+9+9+9+12 $\Longrightarrow$ Je 9 Bits für die 4 Seitentabellen plus 12 Bits Offset

# Hitrate und Missrate

- Bei einer Anfrage an einen Speicher sind 2 Ergebnisse möglich:
  - **Hit**: Angefragte Daten sind vorhanden (Treffer)
  - **Miss**: Angefragte Daten sind nicht vorhanden (verfehlt)
- 2 Kennzahlen bewerten die Effizienz eines Speichers:
  - **Hitrate**: Anzahl der Anfragen an den Speicher mit Ergebnis Hit, geteilt durch die Gesamtanzahl der Anfragen
    - Ergebnis liegt zwischen 0 und 1
    - Je höher der Wert, desto höher ist die Effizienz des Speichers
  - **Missrate**: Anzahl der Anfragen an den Speicher mit Ergebnis Miss, geteilt durch die Gesamtanzahl der Anfragen
    - Missrate $= 1 -$ Hitrate

# Seiten-Ersetzungsstrategien

- Es ist sinnvoll, die Daten ($\Longrightarrow$ **Seiten**) im Speicher zu halten, auf die häufig zugegriffen wird
- Einige **Ersetzungsstrategien**:
  - **OPT** (Optimale Strategie)
  - **LRU** (Least Recently Used)
  - **LFU** (Least Frequently Used)
  - **FIFO** (First In First Out)
  - **Clock / Second Chance**
  - **TTL** (Time To Live)
  - **Random**

**Eine gute Beschreibung der Seitenersetzungsstrategien...**

- OPT, FIFO, LRU und Clock enthält **Operating Systems**, *William Stallings*, 4.Auflage, Prentice Hall (2001), S.355-363
- FIFO, LRU, LFU und Clock enthält **Betriebssysteme**, *Carsten Vogt*, 1.Auflage, Spektrum Verlag (2001), S.162-163
- FIFO, LRU und Clock enthält **Moderne Betriebssysteme**, *Andrew S. Tanenbaum*, 2.Auflage, Pearson (2009), S.237-242
- FIFO, LRU, LFU und Clock enthält **Betriebssysteme**, *Eduard Glatz*, 2.Auflage, dpunkt (2010), S.471-476

# Optimale Strategie (OPT)

Bildquelle: Lukasfilm Games

- Verdrängt die Seite, auf die **am längsten in der Zukunft nicht zugegriffen** wird
- Unmöglich zu implementieren
  - Grund: Niemand kann in die Zukunft sehen
    - Darum muss das Betriebssystem die Vergangenheit berücksichtigen
- Mit OPT bewertet man die Effizienz anderer Ersetzungsstrategien

| Anfragen: | **1** | **2** | **3** | **4** | **1** | **2** | **5** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. Seite: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 3 | 3 |
| 2. Seite: | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 4 | 4 |
| 3. Seite: | | | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |

→ 7 Miss

Die **Anfragen** sind Anforderungen an Seiten im virtuellen Adressraum eines Prozesses. Wenn eine angefragte Seite nicht schon im Cache ist, wird sie aus dem Hauptspeicher oder dem Auslagerungsspeicher (Swap) nachgeladen

# Least Recently Used (LRU)

- Verdrängt die Seite, auf die **am längsten nicht zugegriffen** wurde
- Alle Seiten werden in einer Warteschlange eingereiht
- Wird eine Seite in den Speicher geladen oder referenziert, wird sie am Anfang der Warteschlange eingereiht
- Ist der Speicher voll und es kommt zum Miss, wird die Seite am Ende der Warteschlange ausgelagert
- Nachteil: Berücksichtigt nicht die Zugriffshäufigkeit

| Anfragen: | **1** | **2** | **3** | **4** | **1** | **2** | **5** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. Seite: | 1 | 1 | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 |
| 2. Seite: | | 2 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 |
| 3. Seite: | | | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |

→ 10 Miss

# Least Frequently Used (LFU)

- Verdrängt die Seite, auf die **am wenigsten zugegriffen** wurde
- Für jede Seite im Speicher wird in der Seitentabelle ein Referenzzähler geführt, der die Anzahl der Zugriffe speichert
- Ist der Speicher voll und kommt es zum Miss, wird die Seite entfernt, deren Referenzzähler den niedrigsten Wert hat
- Vorteil: Berücksichtigt die Zugriffshäufigkeit
- Nachteil: Seiten, auf die in der Vergangenheit häufig zugegriffen wurde, können den Speicher blockieren

| Anfragen: | **1** | **2** | **3** | **4** | **1** | **2** | **5** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. Seite: | $_1$1 | $_1$1 | $_1$1 | $_1$4 | $_1$4 | $_1$4 | $_1$5 | $_1$5 | $_1$5 | $_1$3 | $_1$4 | $_1$5 |
| 2. Seite: | | $_1$2 | $_1$2 | $_1$2 | $_1$1 | $_1$1 | $_1$1 | $_2$1 | $_2$1 | $_2$1 | $_2$1 | $_2$1 |
| 3. Seite: | | | $_1$3 | $_1$3 | $_1$3 | $_1$2 | $_1$2 | $_1$2 | $_2$2 | $_2$2 | $_2$2 | $_2$2 |

→ 10 Miss

# First In First Out (FIFO)

- Verdrängt die Seite, die sich **am längsten im Speicher** befindet
- Annahme: Eine Vergrößerung des Speichers führt zu weniger oder schlechtestenfalls gleich vielen Miss
- Problem: Laszlo Belady zeigte 1969, dass bei bestimmten Zugriffsmustern FIFO bei einem vergrößerten Speicher zu mehr Miss führt ($\Longrightarrow$ **Belady's Anomalie**)
  - Bis zur Entdeckung von Belady's Anomalie galt FIFO als gute Ersetzungsstrategie

# Belady's Anomalie (1969)

| Anfragen: | **1** | **2** | **3** | **4** | **1** | **2** | **5** | **1** | **2** | **3** | **4** | **5** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. Seite: | 1 | 1 | 1 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | |
| 2. Seite: | | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 3 | 3 | |
| 3. Seite: | | | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 4 | 4 | → 9 Miss |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. Seite: | 1 | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 4 | 4 | |
| 2. Seite: | | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 5 | |
| 3. Seite: | | | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | |
| 4. Seite: | | | | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 | → 10 Miss |

**Weitere Informationen zu Belady's Anomalie**

Belady, Nelson and Shedler. *An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine*. Communications of the ACM. Volume 12 Issue 6. June 1969

# Clock / Second Chance

- Dieses Verfahren verwendet das *Reference-Bit* (siehe Folie 28), das das Betriebssystem für jede Seite in der Seitentabelle führt
  - Wird eine Seite in den Speicher geladen $\Longrightarrow$ Reference-Bit $= 0$
  - Wird auf eine Seite zugegriffen $\Longrightarrow$ Reference-Bit $= 1$
- Ein Zeiger zeigt auf die zuletzt zugegriffene Seite
- Beim Miss wird der Speicher ab dem Zeiger nach der ersten Seite durchsucht, deren Reference-Bit den Wert 0 hat
  - Diese Seite wird ersetzt
  - Bei allen bei der Suche durchgesehenen Seiten, bei denen das Reference-Bit den Wert 1 hat, wird es auf 0 gesetzt

| Anfragen: | **1** | **2** | **3** | **4** | **1** | **2** | **5** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. Seite: | $_0 1^X$ | $_0 1$ | $_0 1$ | $_0 4^X$ | $_0 4$ | $_0 4$ | $_0 5^X$ | $_0 5$ | $_0 5$ | $_0 3^X$ | $_0 4^X$ | $_0 4$ |
| 2. Seite: | | $_0 2^X$ | $_0 2$ | $_0 2$ | $_0 1^X$ | $_0 1$ | $_0 1$ | $_1 1^X$ | $_1 1$ | $_1 1$ | $_0 1$ | $_0 5^X$ |
| 3. Seite: | | | $_0 3^X$ | $_0 3$ | $_0 3$ | $_0 2^X$ | $_0 2$ | $_0 2$ | $_1 2^X$ | $_1 2$ | $_0 2$ | $_0 2$ |

$\longrightarrow$ 10 Miss

Linux, BSD-UNIX, VAX/VMS (ursprünglich von Digital Equipment Corporation) und Microsoft Windows NT 4.0 auf Uniprozessor-Systemen verwenden Clock oder Varianten davon

# Weitere Ersetzungsstrategien

- **TTL** (Time To Live): Jede Seite bekommt beim Laden in den Speicher eine Lebenszeit zugeordnet
  - Ist die TTL überschritten, kann die Seite verdrängt werden
- **Random**: Zufällige Seiten werden verdrängt
  - Vorteile: Simple und ressourcenschonende Ersetzungsstrategie
    - Grund: Es müssen keine Informationen über das Zugriffsverhalten gespeichert werden

**Die Ersetzungsstrategie Random wird (wurde) in der Praxis eingesetzt**

- Die Betriebssysteme IBM OS/390 und Windows NT 4.0 auf SMP-Systemen verwenden die Ersetzungsstrategie Random
- Die Intel i860 RISC-CPU verwendet die Ersetzungsstrategie Random für den Cache