Betriebssysteme und Netzwerke Vorlesung 12

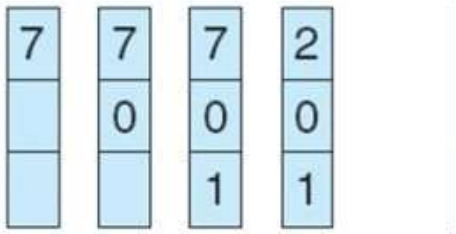
Artur Andrzejak

Seitenersetzungsalgorithmen: LRU und verwandte Strategien

Least-Recently-Used (**LRU**)-Algorithmus

- Ersetze Seite, die am längsten nicht benutzt wurde
  - Annahme: Das historische Verhalten ist eine Annäherung des Zukunftsverhaltens - wie bei Aktienfonds! ☺
- Für jede Seite wird der Zeitpunkt des letzten Zugriffs notiert; *ist das einfach?*
- Das ist ein sehr guter Algorithmus, aber wie kann man ihn umsetzen?
  - **A**: Datenstruktur, **B**: Speichern des letzten Zugriffs pro Seite

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



B: Speichern der Zeit eines Zugriffs

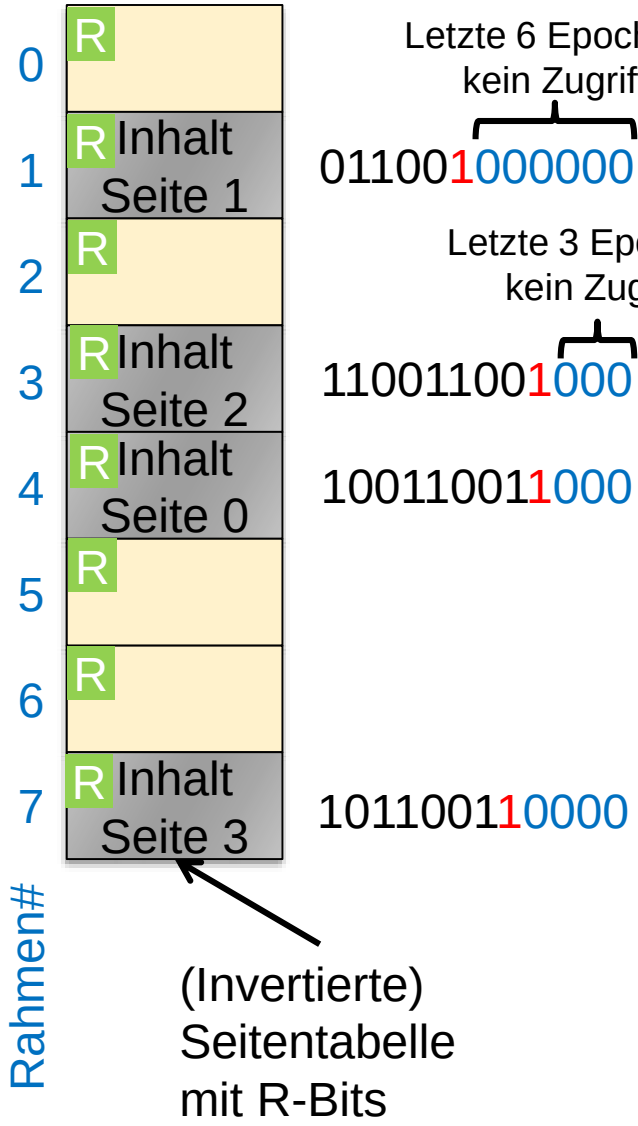
- Erste Vereinfachung
  - CPU hat eine „logische Uhr“ oder einen Zähler
  - Mit jedem Zugriff auf eine Seite wird der Inhalt des Zählers in ein spezielles Feld X der Seite kopiert
  - Der kleinste Wert X unter aller Seiten gibt uns eine Seite S mit dem ältesten Zugriff (S am längsten nicht benutzt)
- Das Speichern der normalen / logischen Zeit des letzten Zugriffs auf eine Seite ist sehr aufwändig
  - Wir brauchen einen Zeitstempel / Zähler pro Seite => Speicherbedarf wie bei einer Seitentabelle => im RAM
  - Bei jedem CPU-Befehl mit RAM-Zugriff müssten wir (nochmals) ins RAM schreiben – unmöglich!
- Problem: Kaum ein Rechner hat genug Hardware-Unterstützung, um das zu implementieren

B: LRU Approximation mit R-Bits /1

- Die meisten Systeme haben aber die **R-Bits**
  - Für jede Speicherseite ein R-Bit
  - Wird bei jedem Lesezugriff auf die Seite auf 1 gesetzt

- Anfangs setzt das BS alle R-Bits auf 0
- Nach einem Zeitintervall (**Epoche**) scannt das BS alle R-Bits
  - Nur Seiten mit R-Bit = 1 wurden in dieser Epoche benutzt!
  - Die Zeit des letzten Zugriffs innerhalb der Epoche ist unbekannt
- Am Ende der Epoche werden alle R-Bits wieder gelöscht, und eine neue Messung (Epoche) startet
- Man bekommt pro Seite eine Folge von 0/1en; Für jede Epoche bedeutet entsprechende Stelle in der Folge:
  - 0=kein Zugriff, 1=mind. ein Zugriff auf die Seite in der Epoche

B: LRU Approximation mit R-Bits /2



- Die Zeit des letzten Zugriffs auf eine Seite wird durch die Anzahl der letzten (aufeinanderfolgenden) **0**en in der Folge abgeschätzt
- Man speichert nur die Daten der letzten Epochen
  - z.B. Letzte 8, 16 oder 32 Epochen

Zählende Algorithmen

- Man zählt die Anzahl der Zugriffe auf jede Seite
  - Z.B. bei einem Interrupt werden alle R-Bits durchgegangen, und ein Zähler („Z-Zahl“) zu jeder Seite wird erhöht
- **Least-frequently-used** (**LFU**) Algorithmus
  - Ersetzt die Seite mit einer kleinsten Zugriffszahl
  - Eine wichtige Seite wird dagegen eine hohe Z-Zahl haben
- **Most-frequently-used** (**MFU**) Algorithmus
  - Das genaue Gegenteil: ersetzt Seite mit größter Zugriffszahl
  - Begründung: eine Seite mit kleiner Zugriffszahl wurde ggf. gerade geladen, und wird erst benutzt
- Problem: diese Algorithmen „vergessen“ nie
  - Eine verbesserte Version von LFU heißt **Aging**-Algorithmus

Paging und Virtueller Speicher: Anwendungen

Demand Paging – Einlagern „auf Anforderung“

- Prinzip des **Demand Pagings**:
  - *Hole eine Seite in den Speicher erst dann, wenn sie unmittelbar benötigt wird*
  - Annahme: Seiten haben stets eine Kopie auf der Disk
- Vorteile?

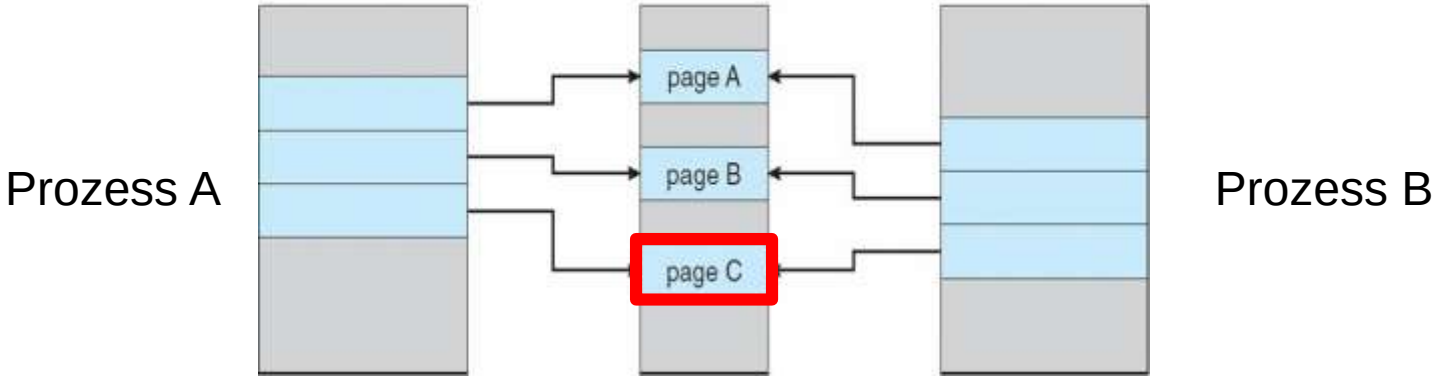
- Weniger Ein-/Ausgabeoperationen
- Weniger Speicher wird gebraucht
- Mehr Prozesse / Benutzer

Virtueller Speicher bei Prozesserzeugung

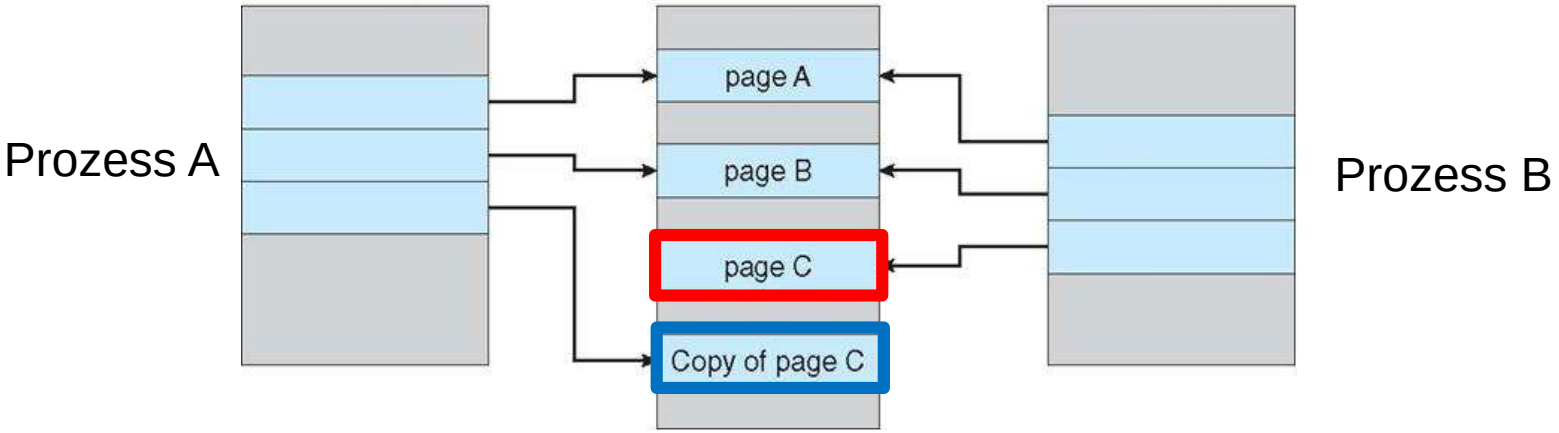
- Bei Prozesserzeugung: Copy-on-Write
- Bei **Copy-on-Write** (**COW**) teilen sich nach fork() die Eltern- und die Kindprozesse zunächst denselben Speicher
- Nur wenn einer der Beteiligten eine Seite modifiziert, wird diese kopiert
- Erlaubt es, den fork()-Befehl sehr schnell auszuführen
  - Kein unnötiges Kopieren, wenn anschließend exec() ausgeführt wird

Beispiel Copy-On-Write

- Nach fork(), vor einem Schreibzugriff

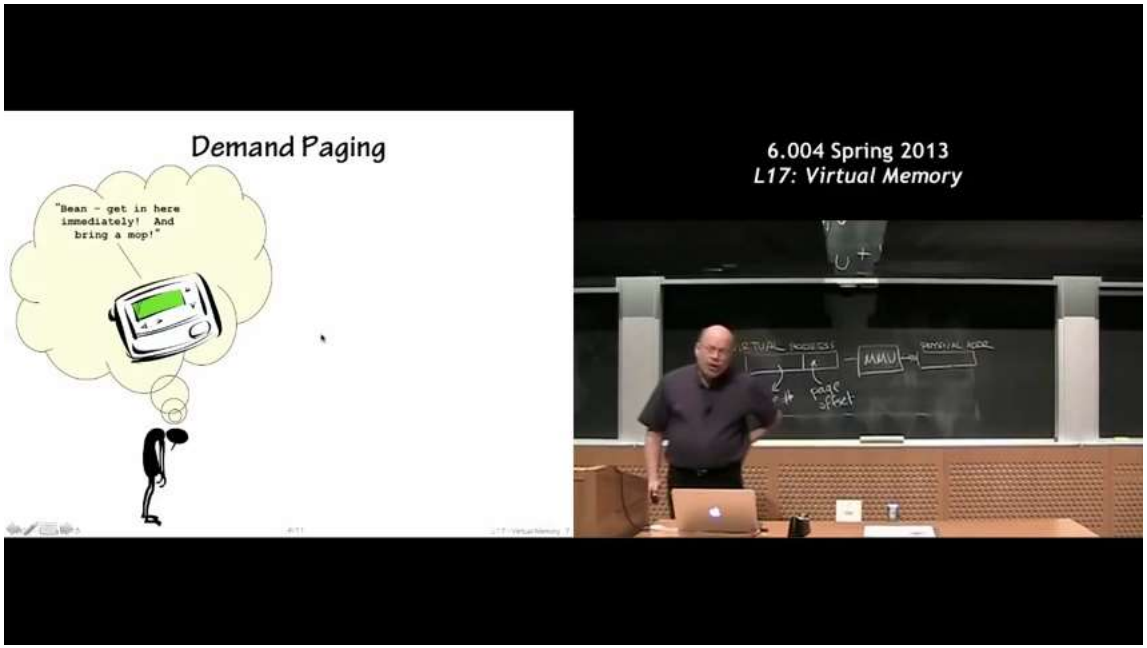


- Nach einem Schreibzugriff auf **Seite C**



Video: Demand Paging - Demonstration

- Video: Virtual Memory [11a]
  - **MIT 6.004 L17: Virtual Memory**
  - https://www.youtube.com/watch?v=3akTtCu_F_k
  - Ab 17:15 bis ca. 19:29 (min:sec)

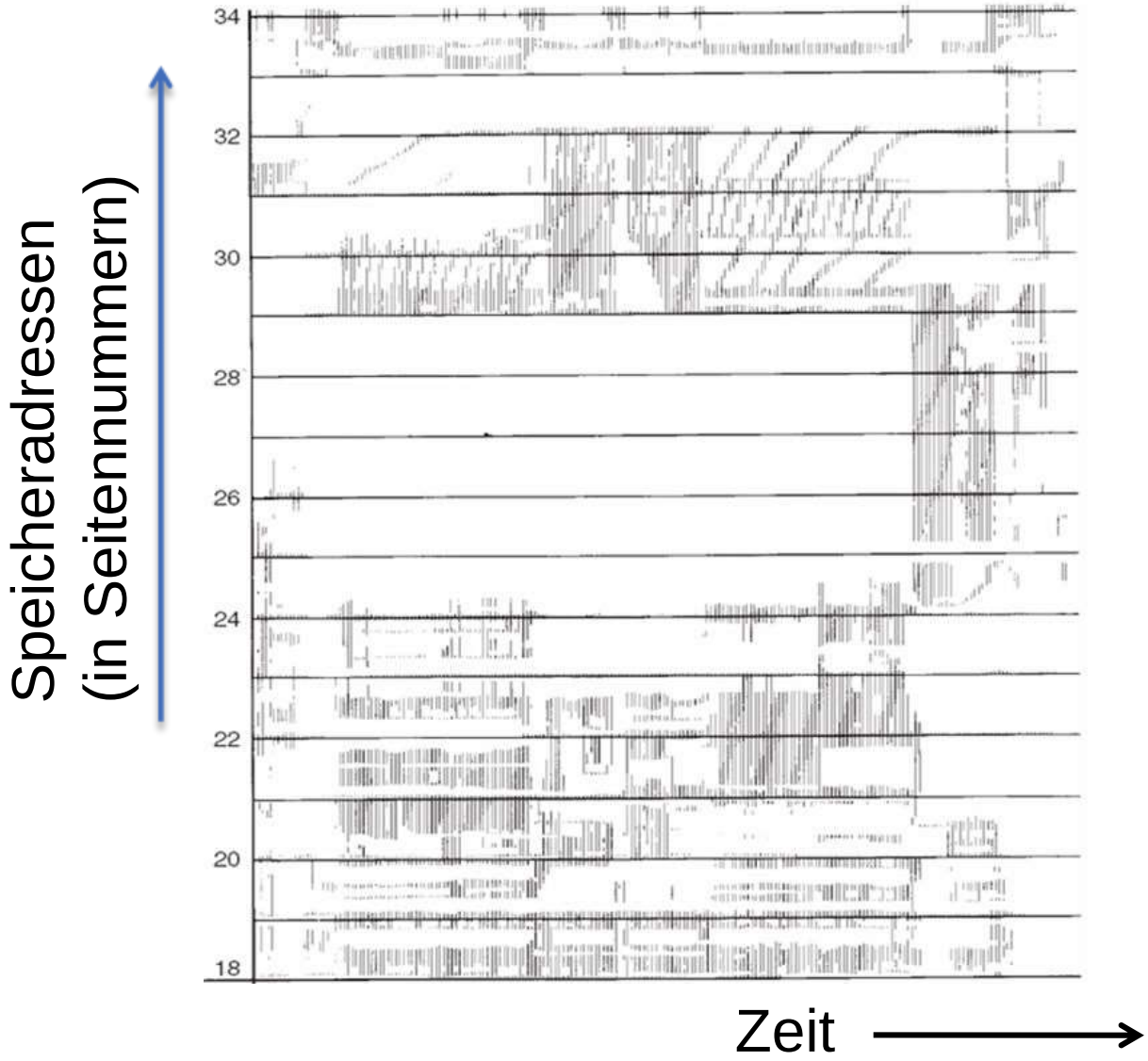


Arbeitsbereich eines Prozesses

Lokalitätseigenschaft von Programmen

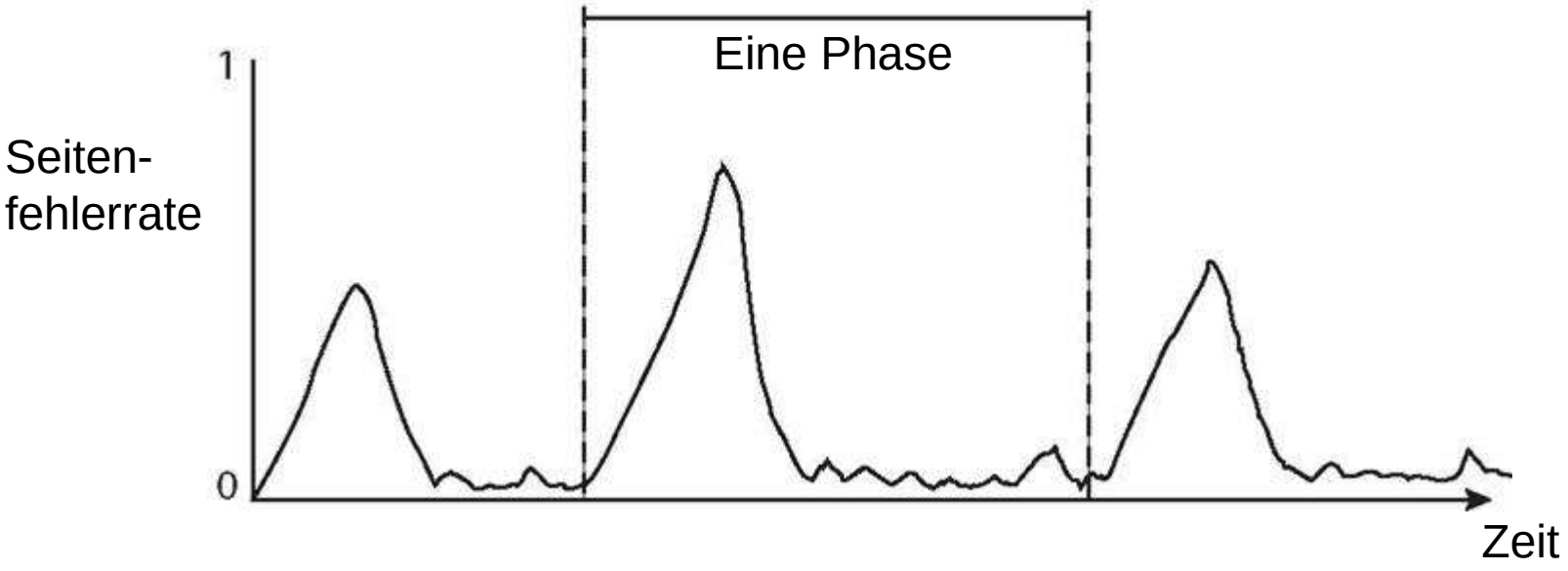
- Wir haben das **Demand Paging** bzw. **Einlagern bei Bedarf** diskutiert
  - Ein Prozess startet ohne geladene Seiten im Speicher, und lagert jede Seite erst beim 1. Zugriff ein
- Warum funktioniert diese Strategie überhaupt?
- Programme weisen oft die **Lokalitätseigenschaft** auf (**locality of reference**)
  - Sie beschränken Zugriffe (in jeder Phase der Ausführung) auf einen relativ kleinen Teil ihrer Seiten
- Die Menge von Seiten, in jeder solchen Phase benutzt wird, heißt der **Arbeitsbereich** (**working set, WS**) – Definition kommt später

Working Set - Beispiel



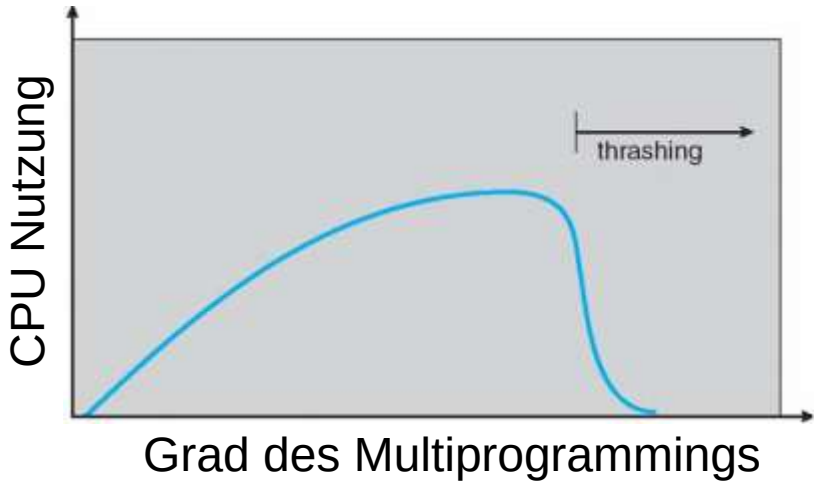
Working Set und Seitenfehlerrate

- Wenn ein Prozess von einer Phase zur anderen wechselt, so muss es (insbesondere bei Demand Paging) viele neue Seiten laden
  - Beispiel: Multi-Pass Compiler
- Das ergibt einen Anstieg der Seitenfehlerrate nach einem Phasenwechsel, bis der neue Working Set komplett geladen ist



Thrashing

- Wenn ein Prozess nicht genügend Seiten hat (weniger als WS), ist die Seitenfehlerrate sehr hoch
  - Folge: geringe CPU Nutzung (**CPU utilization**), da Prozess ständig auf Seitenaus- und Einlagerung wartet
- BS glaubt evtl., dass der Grad des Multiprogrammings erhöht werden kann, und aktiviert weitere Prozesse!
- => **Thrashing** (**Flattern**, **Überlastung**)
- Ein Prozess ist fast nur noch damit beschäftigt, die Seiten ein- und auszulagern

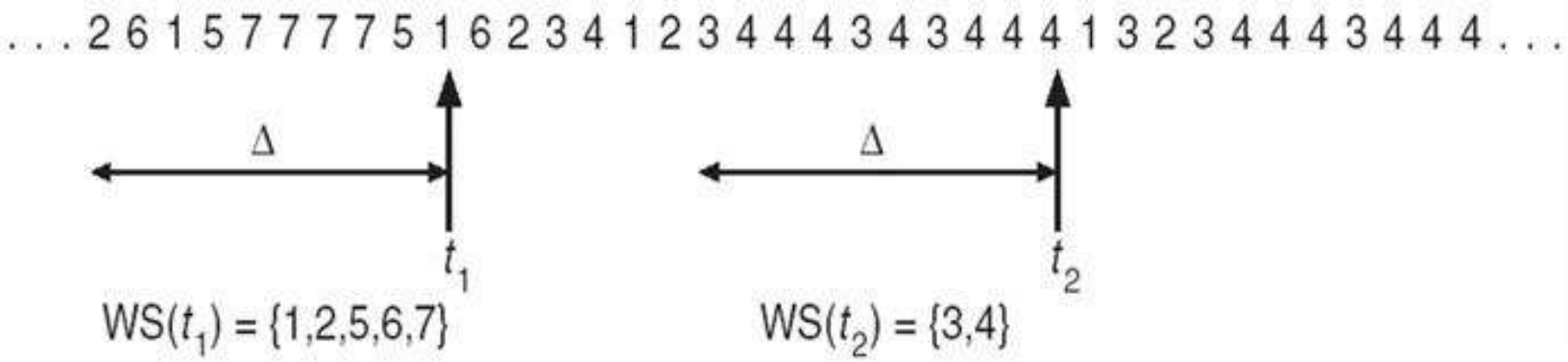


Beispiel: Trashing

- Was ist die ungünstigste Sequenz von Speicherzugriffen eines Prozesses in Hinblick auf die Seitenfehlerrate?
  - D.h. Eine Zugriffsfolge mit den meisten Seitenfehlern?
- Video: Computer Science 61C - Lecture 35: Virtual Memory [12a]
  - https://www.youtube.com/watch?v=45aZGDvzG98
  - Ab 42:45 (min:sec) - ungünstige Zugriffssequenz
  - Ab 46:50 bis 50:30 (min:sec) - Thrashing

Working Set Definieren und Messen

- Definition: „**WS** := Menge der Seiten, die in den letzten Δ Speicherzugriffen benutzt wurden“



- Wie können wir aber die WS wirklich bestimmen?
  - Wir nähern Δ durch die Ausführungszeit eines Prozesses
  - Alle T Mikrosekunden (Epoche) wird ein Interrupt alle R-Bits aufzeichnen und anschließend löschen
  - Eine Seite ist in dem WS, wenn sie innerhalb der letzten k Epochen (k*T entspricht Δ) benutzt wurde

Nutzung der Working Sets

1. Bei laden eines komplett ausgelagerten oder eines neuen Prozesses könnte man gleich den WS laden (statt dem „reinen“ Demand Paging)
   - Ein Spezialfall von **Prepaging** = Strategien, bei dem Seiten geladen werden, noch bevor sie benutzt werden
2. Wenn die kumulative Größe der WS aller Prozesse die Größe des physischen Speichers überschreitet, müssen Prozesse vollständig ausgelagert werden
   - Das BS kann dadurch Bedarf für Swapping erkennen
3. (!) Ein neuer Seitenersetzungsalgorithmus:
   - „Bei einem Seitenfehler lagere eine Seite aus, die nicht zum WS gehört“ – das ist **Working-Set-Algorithmus**

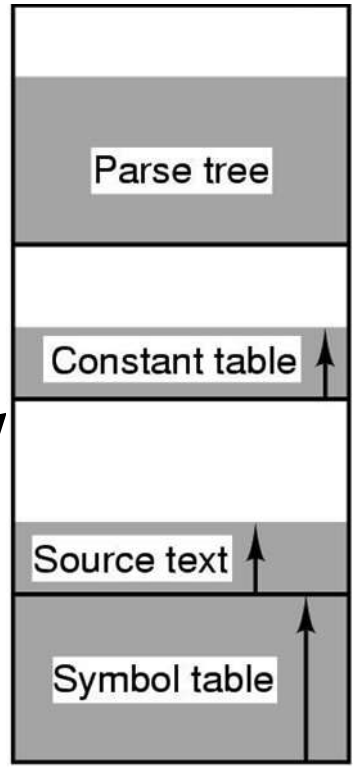
Vergleich Seitenersetzungsalgorithmen

Algorithmus	Bewertung
Optimal	Nicht realisierbar, aber nützlich als Maßstab
FIFO	Entfernt evtl. auch wichtige Seiten
Second Chance	Enorme Verbesserung gegenüber FIFO
Clock / Erw. Clock	Effizientere Version von Second Chance
LRU	Exzellent, aber schwierig zu implementieren
LFU bzw. MFU	Ziemlich große Annäherung an LRU
Aging	Verbesserungen von LFU bzw. MFU
Working Set	Wird (in modifizierter Form) wirklich genutzt

Segmentierung

Linearer Adressraum - Probleme

- Bisherige Annahme: **linearer logischer Adressraum**
  - Eine Folge von aufeinanderfolgenden Adressen
- Ein Prozess hat aber mehrere funktionelle Abschnitte im Speicher
  - Code (Text), globale Variablen, Heap (Halde), Shared Memory, Shared Libraries
  - Dazu Stack(s) - ggf. zwei pro Thread (!)
- Auch innerhalb der Halde kann es viele Abschnitte geben
  - Beispiel: Tabellen eines Compilers
- Die Verwaltung dieser Speicherabschnitte ist eine (unnötige) Belastung für den Programmierer

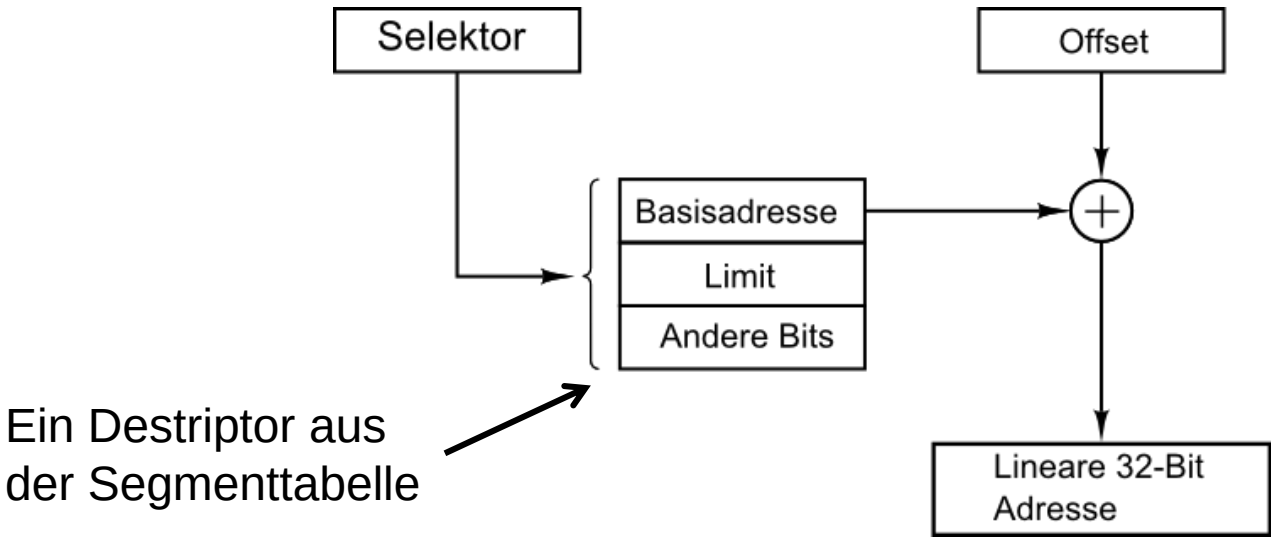


Segmentierung

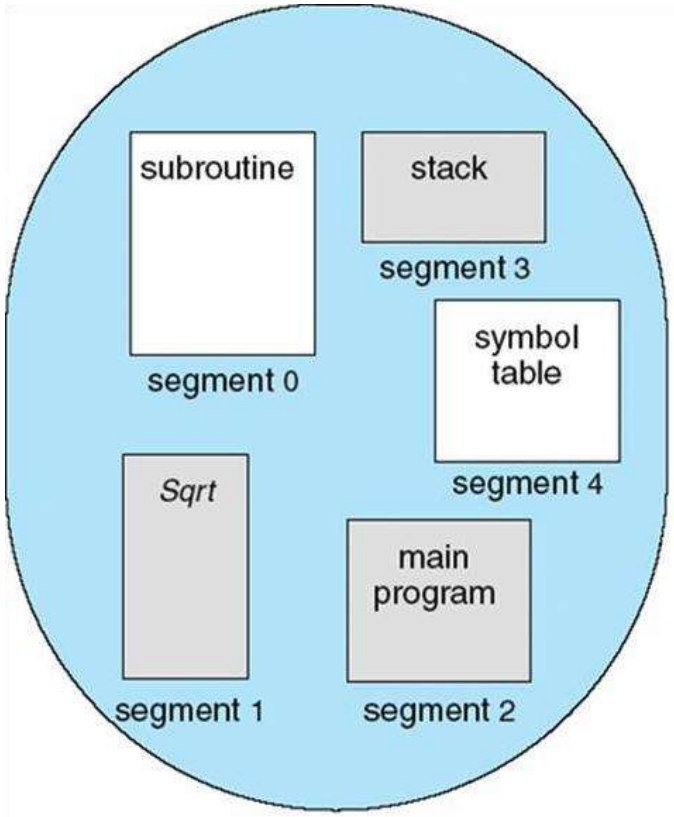
- Lösung: wir führen viele völlig unabhängige (logische) Adressräume ein, die **Segmenten**
  - Jedes Segment besteht aus einer Folge von Adressen, von 0 bis zu einem (frei wählbaren) Maximum
- Im Prinzip ist das eine Generalisierung der einfachsten MMU-Implementierung
  - D.h. MMU aus einem **Basis-** und einem **Limitregister**
- Unterschiede sind:
  - Wir können sehr viele solcher „MMU“s per Prozess haben
  - Nutzerprozesse können (ein Teil davon) selbst verwalten

Architektur der Segmentierung

- Eine logische Adresse ist nun ein Paar: <**Selektor**, **Offset**>
- **Selektor** ist der Index in die **Segmenttabelle** (**ST**)
- Jeder Eintrag der ST heißt (Segment-)**Deskriptor**, und hat
  - **Basisadresse** (**base**): Startadresse eines Segments
  - **Limit** (**limit**): die Länge eines Segments
  - Einige Bits für Zugriffsberechtigung, Einstellungen und Flags



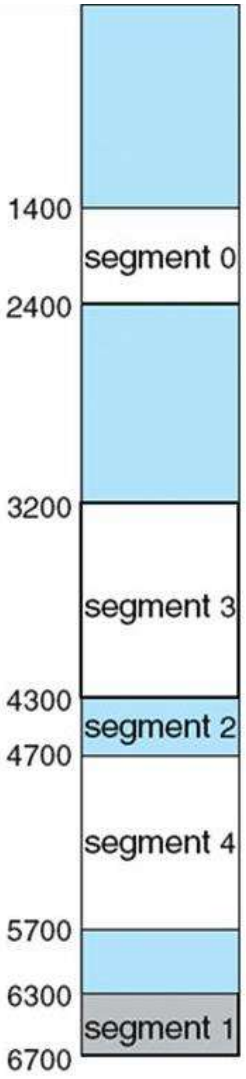
Beispiel: Verwendung der Segmentierung



Adressraum mit Segmenten

Segment-tabelle

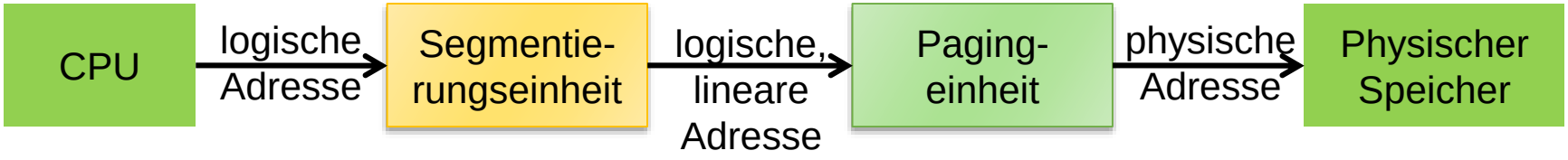
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

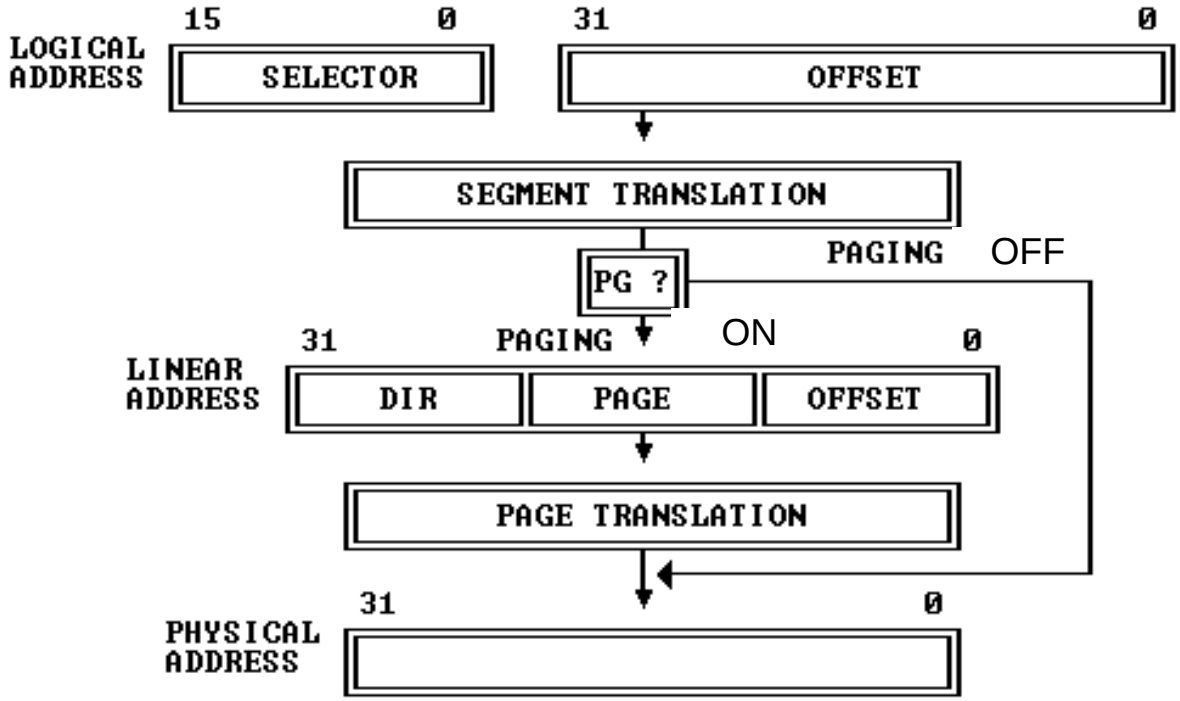


Linearer Adressraum

Komplette Adressenübersetzung

- IA32-Prozessoren haben sowohl Segmentierung (immer an) als auch Paging (kann abgeschaltet werden)





Paging vs. Segmentierung

- Zweck des Pagings
  - Um einen großen zusammenhängenden (logischen) Adressraum zu haben, während die Speicherung in RAM ggf. nicht-zusammenhängend, „wild“ stattfindet
- Zweck der Segmentierung
  - Um Programme und Daten in unabhängige logische Adressräume aufzuspalten
  - Um gemeinsame Nutzung des Speichers und Schutz zu unterstützen

Paging (P) vs. Segmentierung (S) /2

Aspekt	P	S
Ist die Technik transparent für den Programmierer?		
Wie viele lineare Adressräume gibt es?		
Kann logischer Adressraum größer als physischer sein?		
Können Code und Daten unterschieden und getrennt voneinander geschützt werden?		
Können Tabellen mit schwankender Größe verwaltet werden?		

Die Segmentierung wird kaum benutzt - warum?

Zusatzfolien: Segmentierung bei IA32 Prozessoren

Segmentregister bei 8086

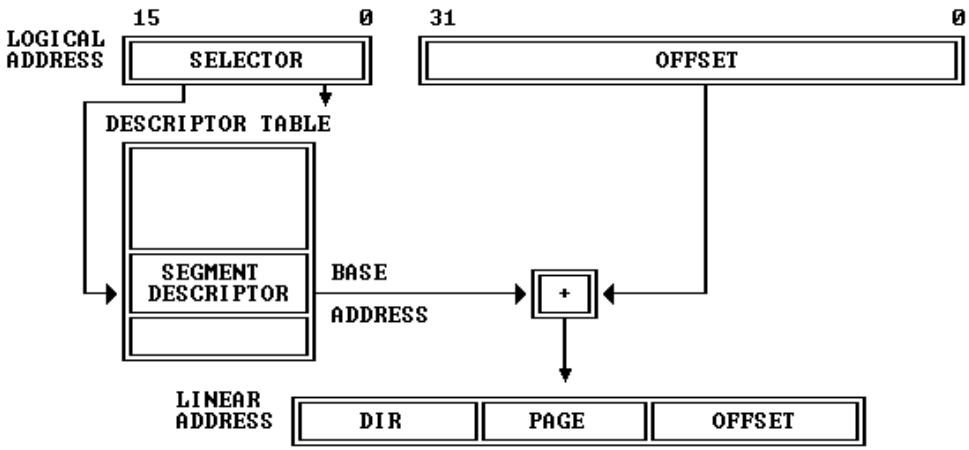
- Bei den 8086 hatte man 16-Bit Register (auch zum Adressieren) aber 20 Bit Bus (bis 1 MB Speicher)
- Wie konnte CPU trotzdem alles adressieren?
- Man hatte die **Segmentregister** (**SR**) eingeführt
- Die physische Adresse war die Summe aus: (i) Inhalt eines SR und (ii) „normale“ Adressangabe
  - **CS-Register**: für das Codesegment
  - **SS-Register**: für den Stack
  - **DS, ES** : Daten („default“), String-Befehle
  - **FS, GS**: spätere Register für die Datensegmente
- Syntax: <SR>:<adr | register>
  - D.h. effektive Adresse ist z.B. <Inhalt von SR> + adr

Segmentregister ab 386

- Ab dem 386 reichten 20 Bits nicht mehr – man hatte einen 32 Bit Adressraum
- *"All problems in computer science can be solved by another level of indirection." (David Wheeler, wiki)*

=>

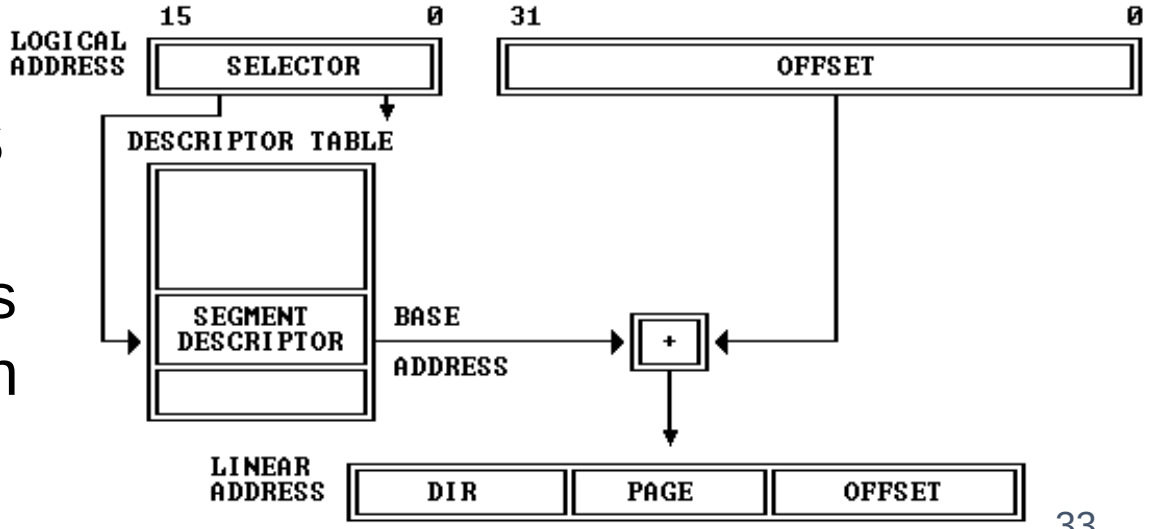
- Die Segmentregister wurden zu **Selektoren**
- Die Selektoren enthalten ein Index in eine Tabelle mit (Segment-)**Deskriptoren**
- Die **Deskriptoren** enthalten (u.a.) den tatsächlichen Adressen-Summanden („Base-Address“)



Deskriptortabellen

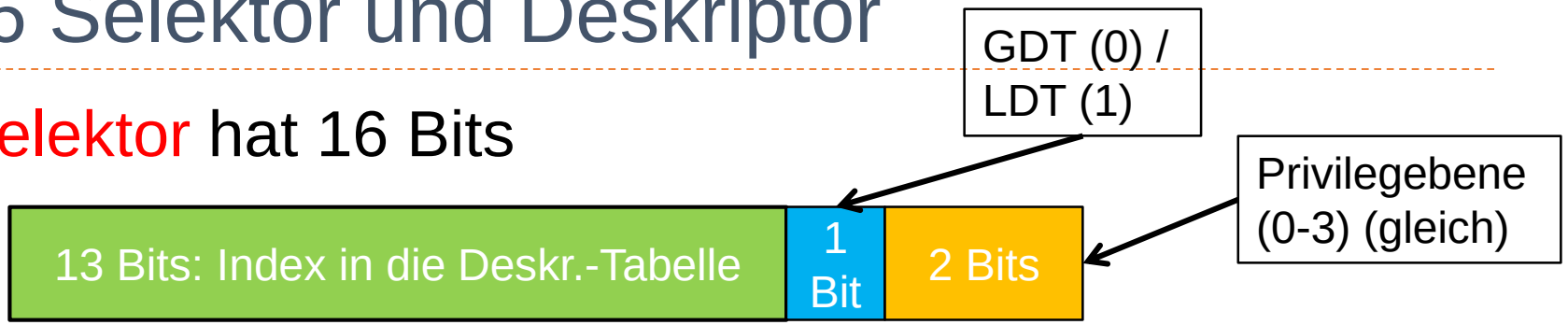
- IA32-Prozessoren haben zwei Deskriptortabellen, mit jeweils 8192 (-1) an Deskriptoren
  - **Local Descriptor Table (LDT)**
    - Separat für jeden Prozess
    - Beschreibt u.a. Code-, Daten- und Stacksegment
  - **Global Descriptor Table (GDT)**
    - Gemeinsam für alle Prozesse und für das BS

- Die Selektoren CS, DS, SS, ES, FS, GS wählen (jeweils) einen Deskriptor aus einer dieser Tabellen aus

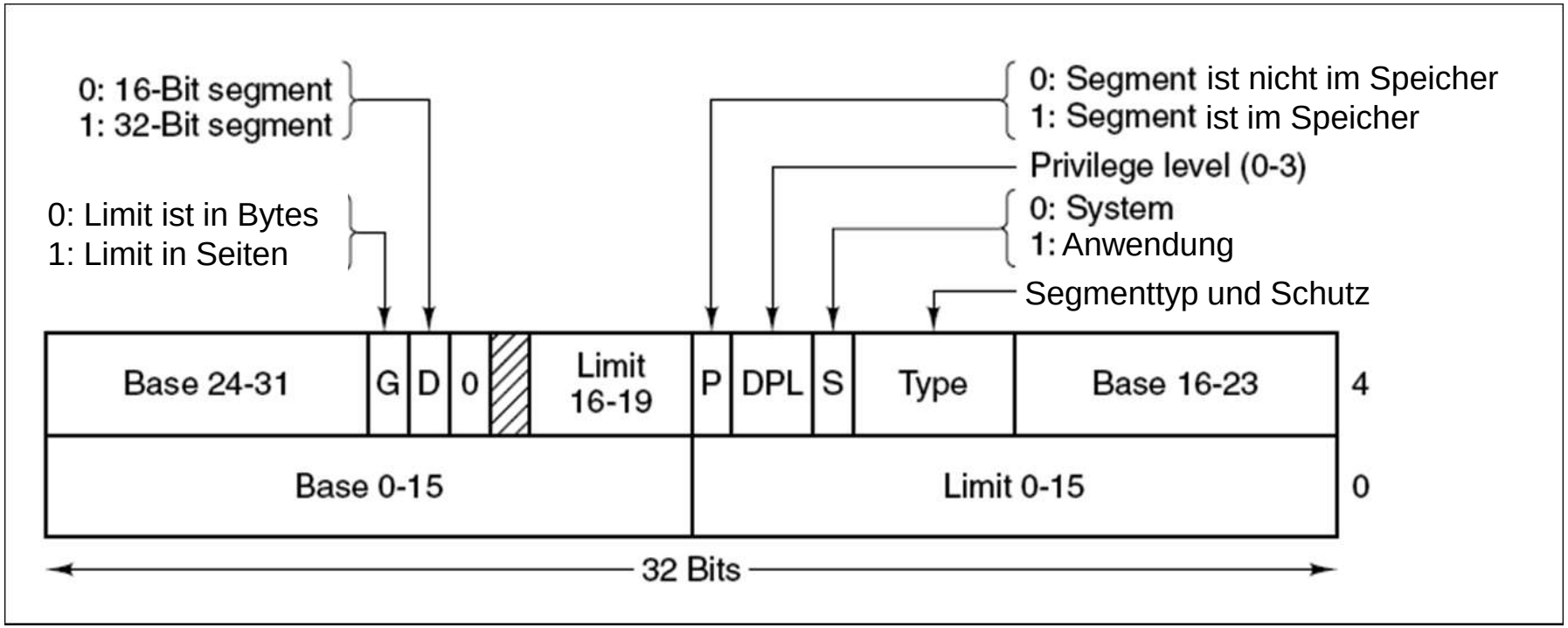


x86 Selektor und Deskriptor

- Selektor hat 16 Bits



- 1 Deskriptor hat 8 Bytes (für ein Codesegment)



Details x86 Segmentierung

- Selektoren werden i.A. implizit benutzt
  - IP instruction pointer (Befehlszähler) nutzt immer **CS**
  - Stack-Ops (push, pop, …) nutzen immer **SS**
  - Die meisten Datenzugriffe nutzen implizit den **DS**
    - Wenn nicht anders explizit durch <SR>: … angegeben
- Die Segmentierung kann man auf den CPUs der Typen x86-32 und x86-64 nicht abschalen!
  - Man muss sie auf 0 setzen – z.B. in Linux-Kernel wird ein „flat memory model“ simuliert (Segmente starten bei 0)
  - * **__KERNEL_CS** (Kernel code segment, base=**0**, limit=4GB, DPL=0)
  - * **__KERNEL_DS** (Kernel data segment, base=**0**, limit=4GB, DPL=0)
  - * **__USER_CS** (User code segment, base=**0**, limit=4GB, DPL=3)
  - * **__USER_DS** (User data segment, base=**0**, limit=4GB, DPL=3)

Speicherschutz

- Die x86-Prozessoren kennen 4 **Privilegebenen**
  - 0: die höchst privilegierte Ebene (für Kernel)
  - 3: die am wenigsten privilegierte Ebene (für Benutzer)
- Der Deskriptor enthält den **Descriptor Priviledge Level** (**DLP**)
- Die untersten 2 Bits des CS-Registers bestimmen den **Current Privilege Level** (**CPL**)
- Der Selektor (d.h. einer der Register SS, DS, ES, FS, GS) hat den **Requested Priviledge Level** (**RPL**)
- Nur wenn **DPL >= max(CPL, RPL)**, ist der Zugriff auf Inhalt des Segmentes erlaubt, sonst wird ein *general protection fault* (GP) erzeugt

Zusammenfassung

- Working Set
  - Anwendungen: Prepaging; Erkennen des Bedarfs von Swapping; Seitenersetzungsalgorithmus; Thrashing
- Segmentierung
  - Adressenübersetzung für lineare Speicherbereiche
- Quellen ():
  - Speicherv. : Silberschatz Kap. 9; Tanenbaum Kap. 3