

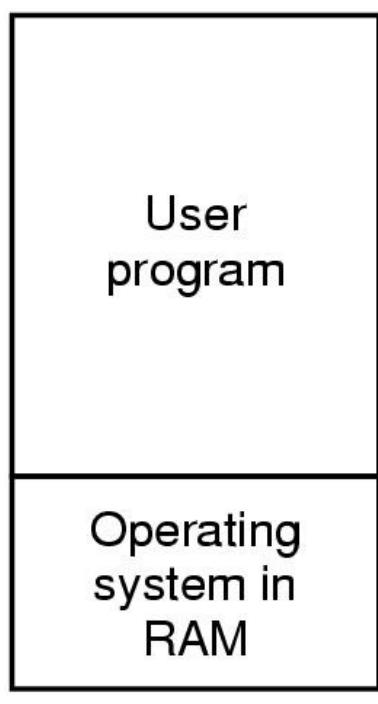
# Betriebssysteme

## Speicherverwaltung

---

Prof. Dr.-Ing. Torben Weis  
Universität Duisburg-Essen

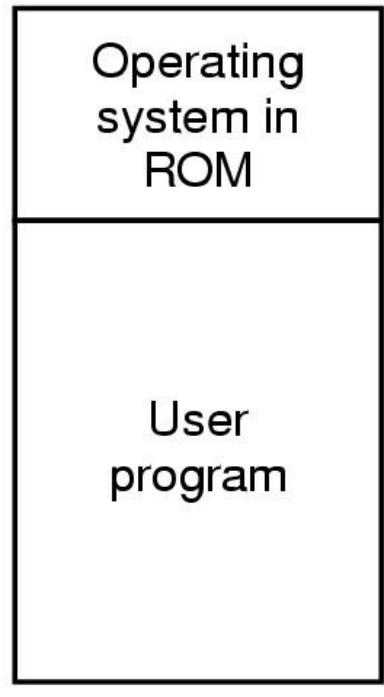
# Speicherorganisation



(a)

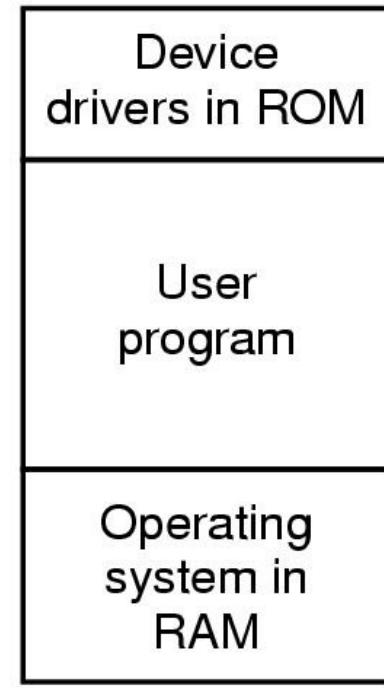
0xFFFF ...

0



(b)

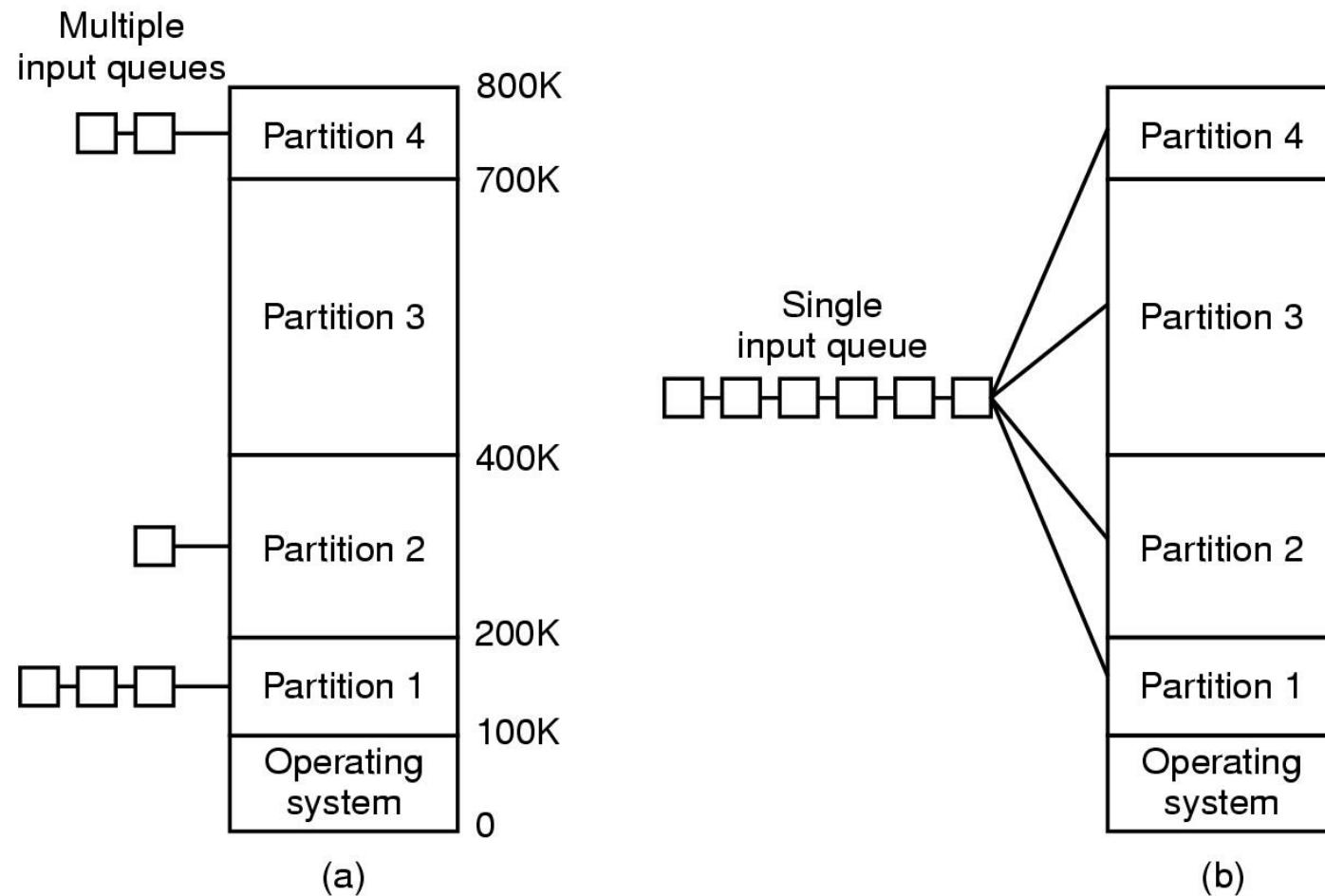
0



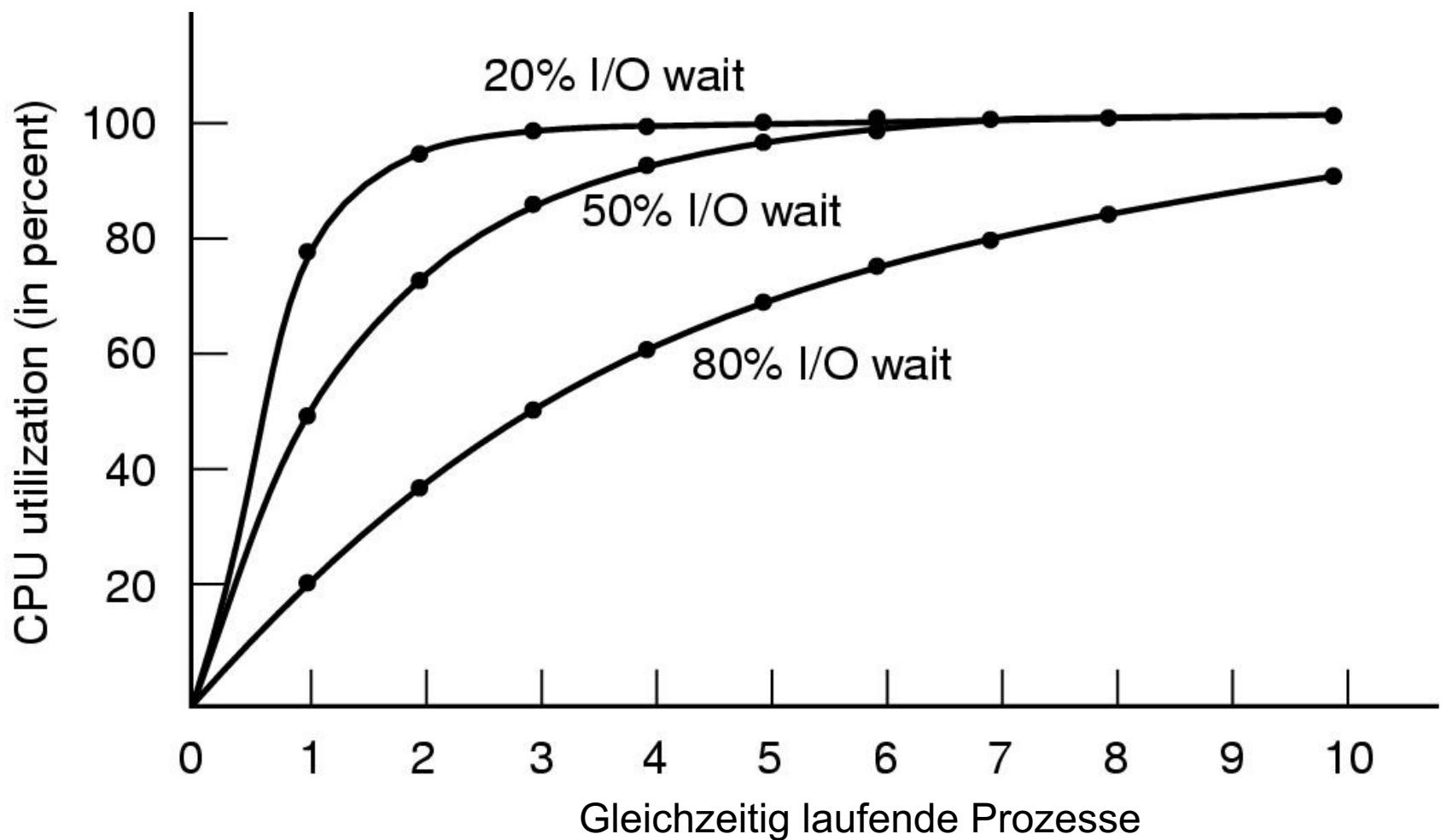
(c)

0

# Speicherorganisation bei Multiprogramming



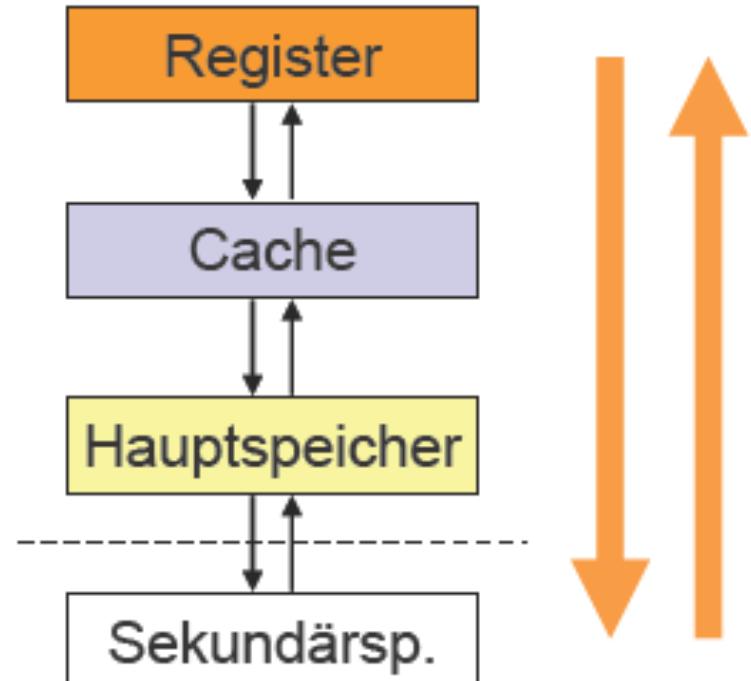
# CPU-Auslastung bei Multiprogramming



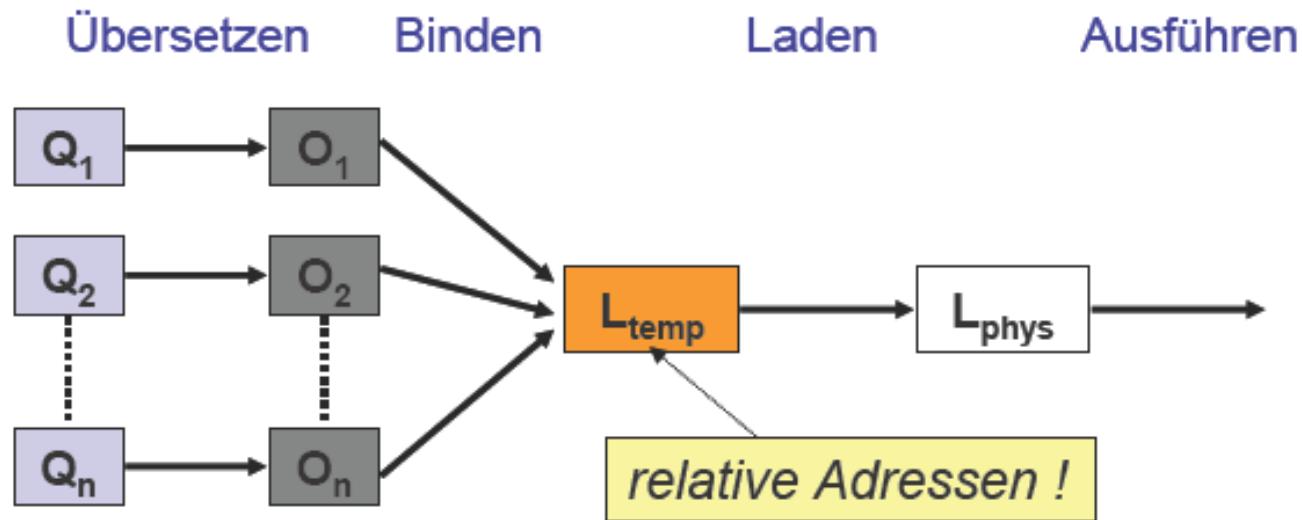
# Speicherverwaltung

- Speicherverteilung zwischen Prozessen
  - Zuteilung geschieht durch das Betriebssystem
  - Zugeteilter Speicher sieht wie linearer Adressraum aus
- Speicherverteilung innerhalb eines Prozesses
  - Übernimmt das Laufzeitsystem
  - UNIX + C: libc
  - Java/.NET: Virtuelle Maschine
  - Zugeteilter Speicher sind viele einzelne Speicherbereiche

*Speicherhierarchie*



# Laden eines Programms



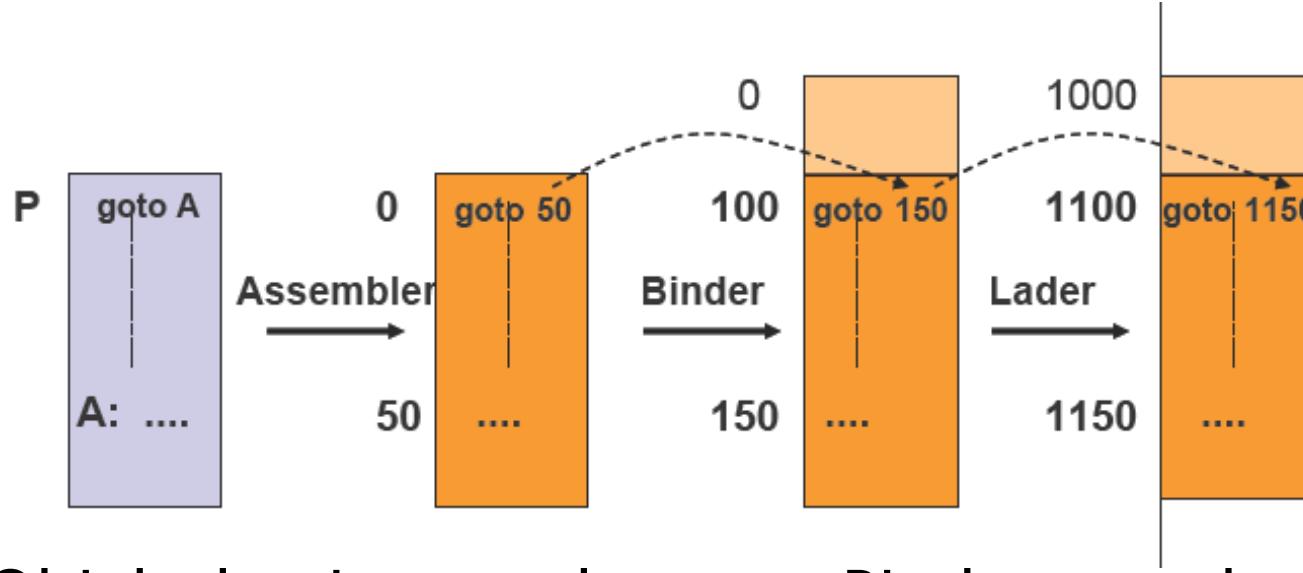
$Q_i$  Quellprogramm

$O_i$  Objektmodul

$L_{\text{temp}}$  verschiebbares Lademodul

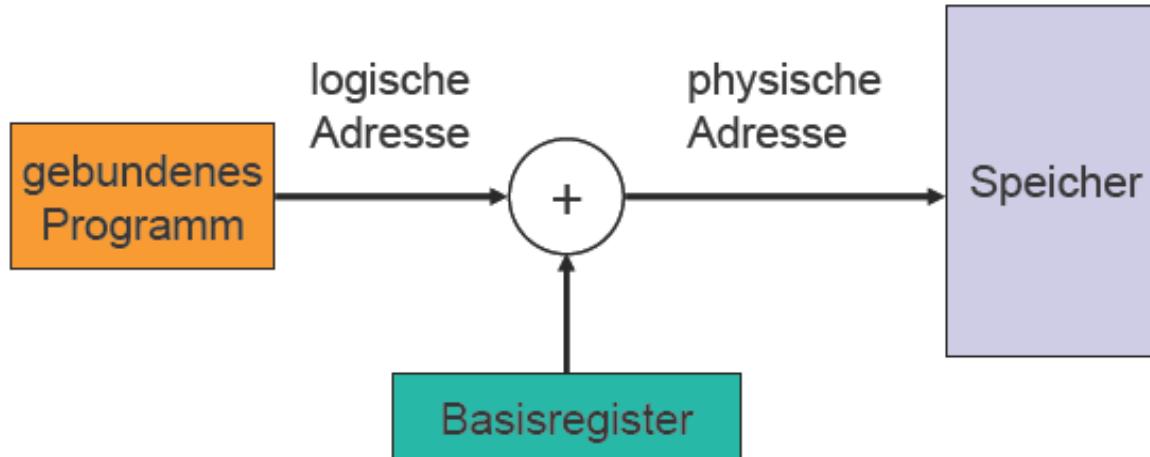
$L_{\text{phys}}$  Lademodul

# Statische Relokation



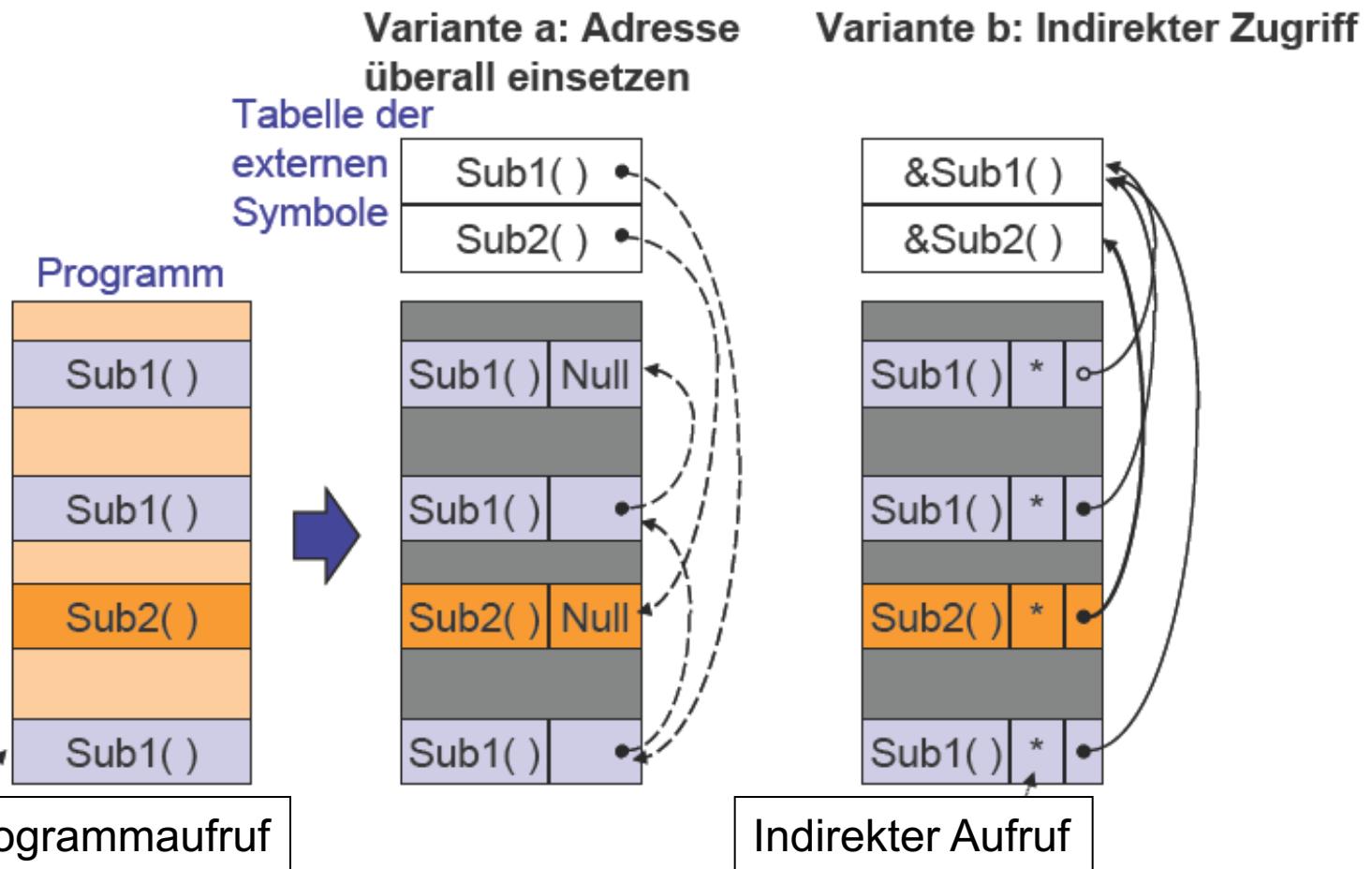
- Objektdateien werden vom Binder verschoben
  - Alle Codestücke müssen sich einen linearen Speicherbereich teilen
- Programm wird vor dem Start verschoben (relocated)
  - Verzögert den Programmstart
  - Speicherverschwendung, denn man muss festhalten, welche Bytes Adressen sind und welche nicht

# Dynamische Relokation



- Verschiebung findet bei jedem Speicherzugriff statt
- Schneller Programmstart (mangels Relokation)
- Dafür extra Aufwand bei jedem Speicherzugriff
  - Den werden wir aber sowieso haben: für virtuellen Speicher

# Binden mit Externen Symbolen



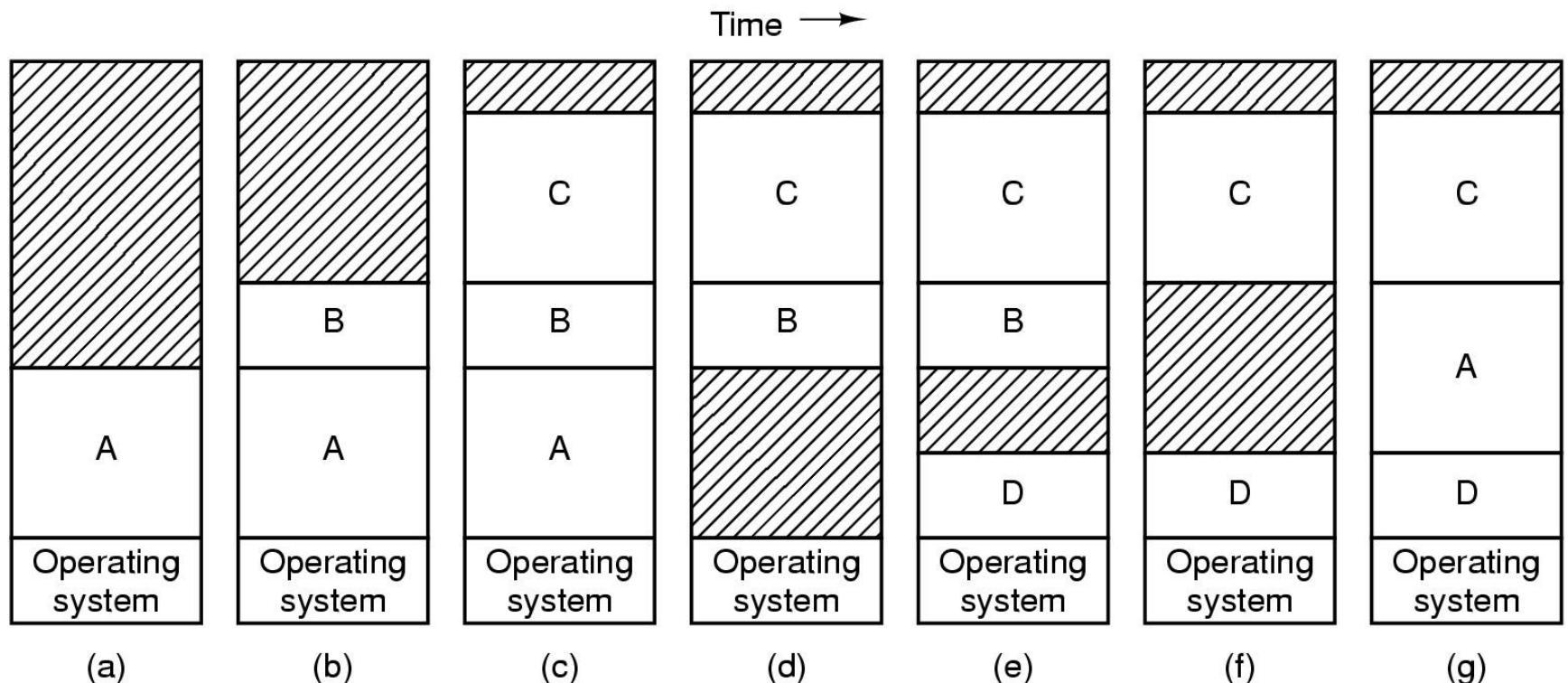
# Laden von Executables & Bibliotheken

- Maschinen-Code soll zwischen Prozessen geteilt werden
  - (Für Java/.NET ergeben sich andere Probleme)
- Statische Relokation
  - Die Bibliothek wird beim ersten Laden auch relokiert
  - Weitere Prozesse müssen die Bibliothek an der selben Stelle anspringen können
- Dynamische Relokation + Virtueller Speicher
  - Jeder Prozess hat eigenen „virtuellen“ linearen Adressraum
  - Bibliothek wird in jedem Prozess an derselben Stelle eingeblendet
  - Oder: Bibliothek wird in jedem Prozess an unterschiedlicher Stelle eingeblendet. Probleme ?

# Swapping

- Mehr Speicher benötigt als verfügbar
- Aber jeder Prozess für sich passt in den Speicher
  - Wichtiger Unterschied zu virtuellem Speicher!
- Swapping
  - Auslagern ganzer Prozesse
  - Speicher freigeben
  - Ein anderes Programm in den freigewordenen Speicher einlagern
- Nach dem Swapping muss ein Programm eventuell relokiert werden
  - Grund: Der freie Speicherbereich liegt an einer anderen physikalischen Adresse
  - Nur bei statische Relokation notwendig
  - Bei dynamischer Relokation ändert man das Basisregister

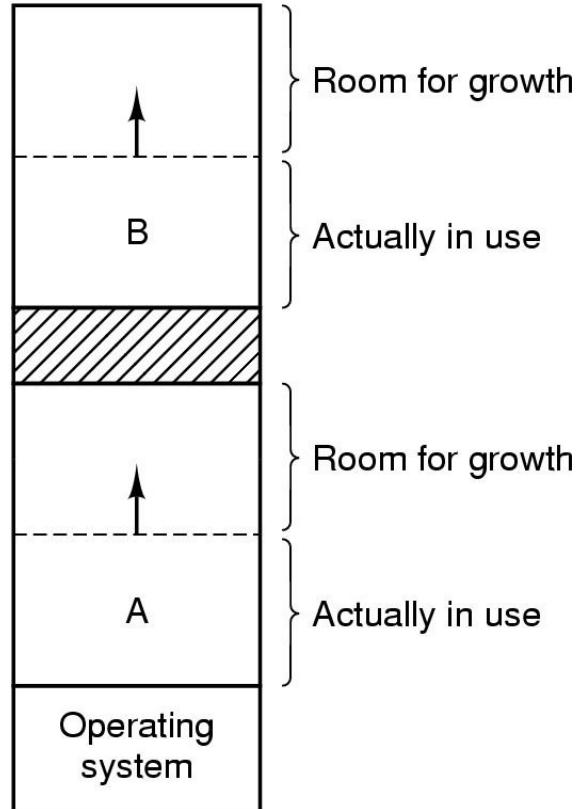
# Swapping



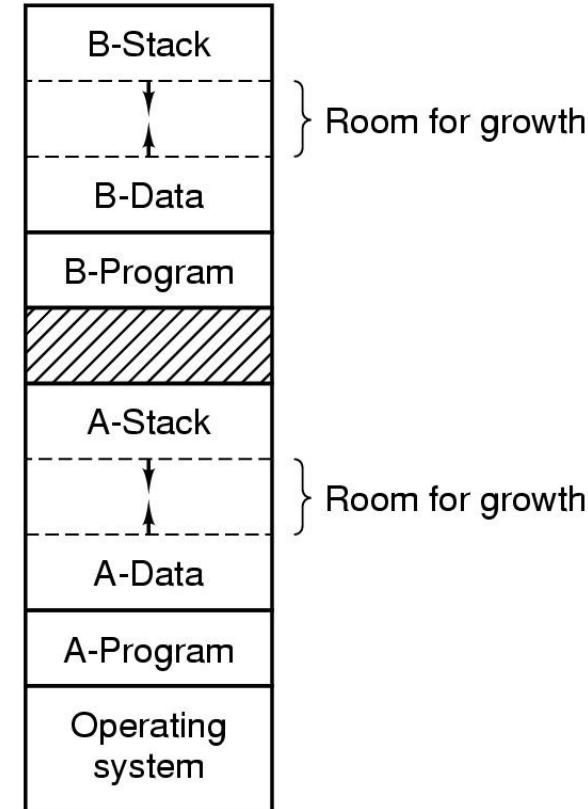
# Fragmentierung (Verschnitt)

- Speicher wird in Vielfachen von festen Blockgrößen vergeben
- Anforderungen werden auf das nächste Vielfache gerundet
- Dadurch entsteht Speicherplatz, der als belegt gekennzeichnet ist, aber nicht benutzt wird
- Man nennt solchen Speicherplatz **internen Verschnitt** (internal fragmentation)
- Durch die Dynamik des Belegens und Freigebens kann es vorkommen, dass eine Anforderung zwar von der Gesamtmenge des freien Speichers her erfüllbar wäre, durch die Zerstückelung jedoch kein hinreichend großes Stück gefunden werden kann
- Dadurch entsteht Speicherplatz, der frei, aber (momentan) nicht belegbar ist
- Er wird als **externer Verschnitt** (external fragmentation) bezeichnet

# Interne Fragmentierung (Verschnitt)

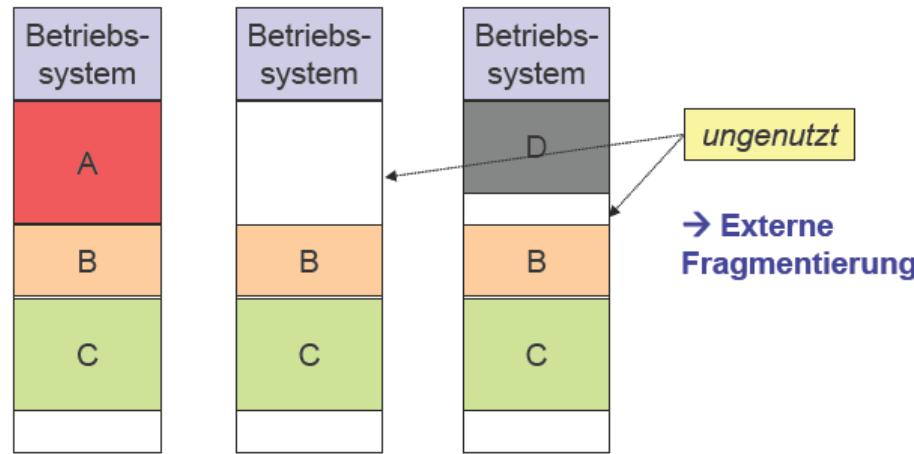


(a)



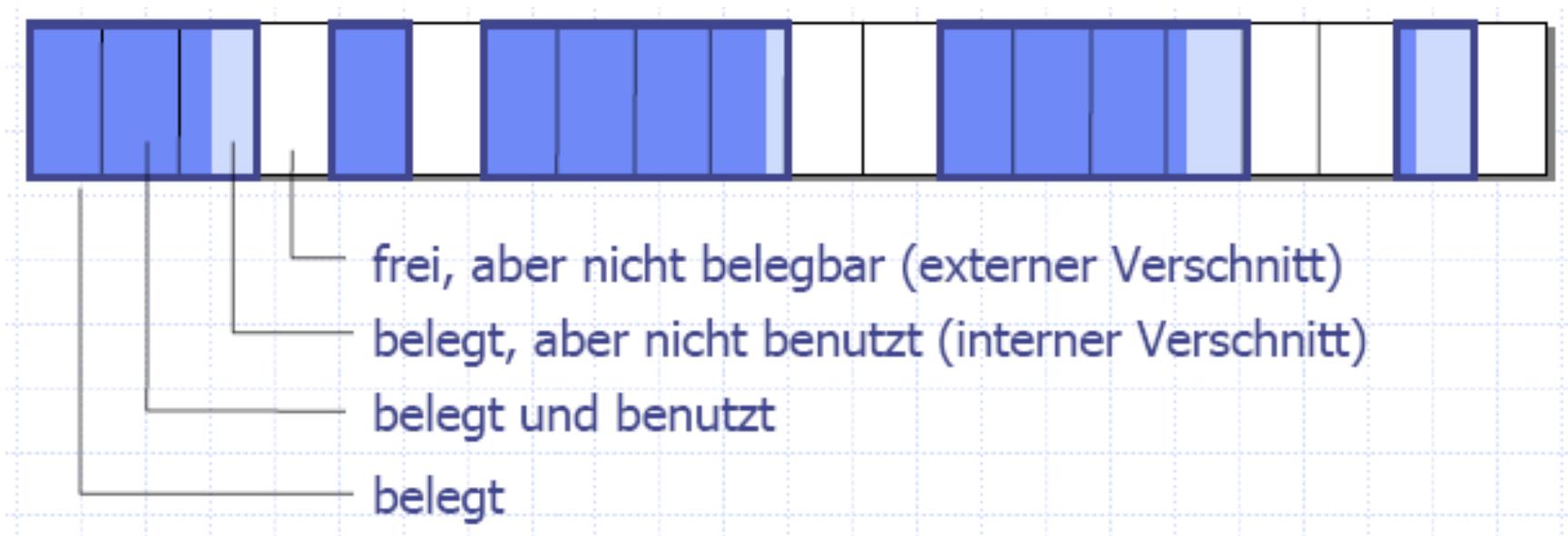
(b)

# Externe Fragmentierung (Verschnitt)



- Memory-Compaction beseitigt externe Fragmentierung
  - Relokation der Prozesse
  - Dadurch wird der freie Speicher zusammengeführt
  - Leider sehr aufwendig
- Virtuelle Speicher
  - Einzelne Fragmente lassen sich leicht und effizient zu einem virtuellen linearen Adressraum zusammenfassen

# Fragmentierung (Verschnitt)



# Freispeicherverwaltung

- Speicherverwaltungsverfahren unterscheiden sich bezüglich der folgenden Aspekte
  - Reihenfolge der Belegung/Freigabe (FIFO/LIFO)
  - Granularität
  - Belegungsdarstellung (Bitmaps, Listen)
  - Verschnitt
  - Auswahlstrategie (bei den freien Stücken)
  - Wiedereingliederung

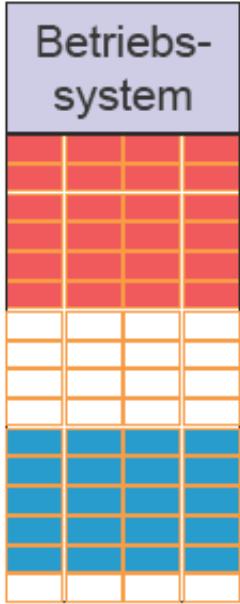
# Freispeicherverwaltung: Reihenfolge

- Belegungen und Freigaben
- In gleicher Reihenfolge
  - (Schlangenverfahren, FIFO = First In First Out)
- In umgekehrter Reihenfolge
  - (Stapelverfahren, LIFO = Last In First Out)
- In beliebiger Reihenfolge
  - (allgemeiner Fall)

# Freispeicherverwaltung: Granularität

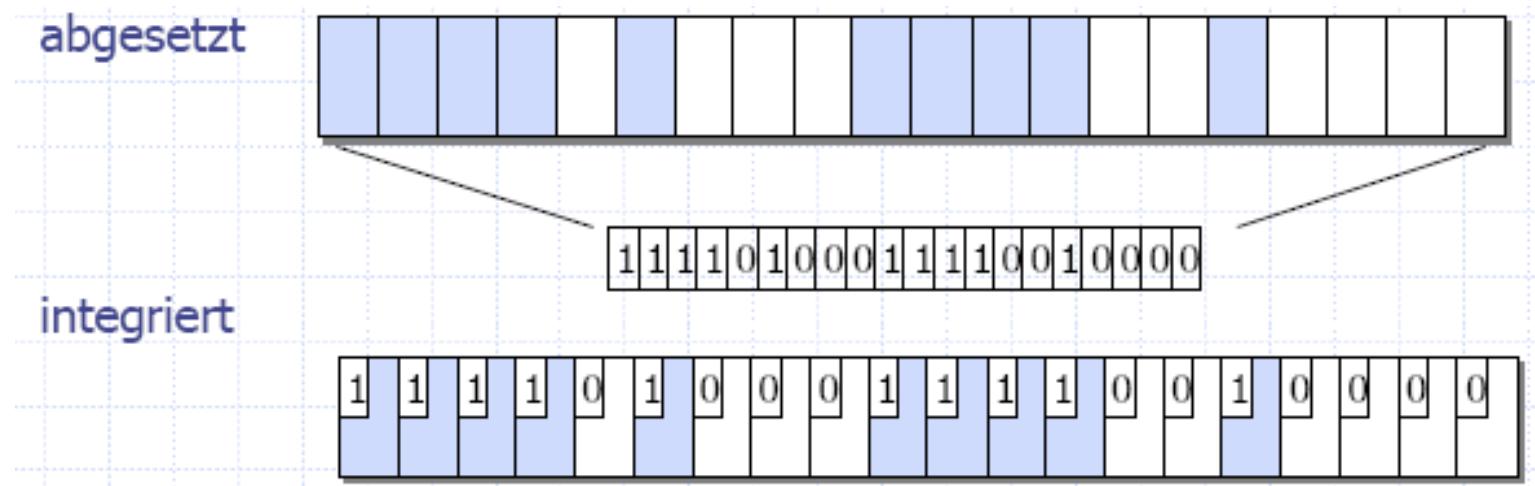
- Konstante Einheiten
  - $\text{NUM} = 1$  (Grundeinheit)
- Vielfache gleicher Grundeinheiten
  - $\text{NUM} = k$  (Grundeinheit)
- Bestimmte Portionsgrößen
  - $\text{NUM} = k_1, k_2, k_3, \dots$
- Beliebige Größen
  - $\text{NUM} = x$

# Freispeicherverwaltung mit Bitmaps



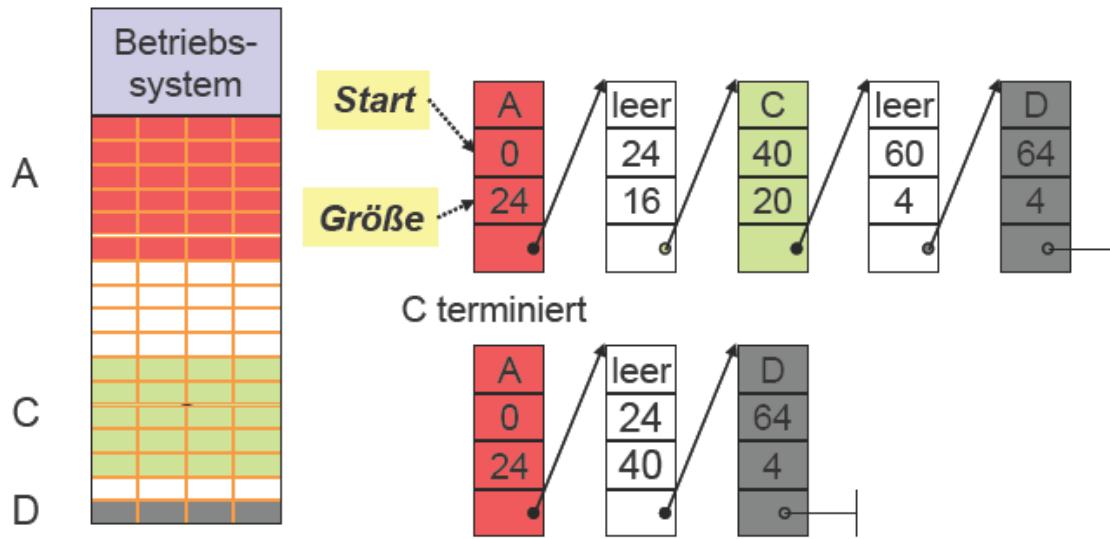
- Speicher wird in Blöcke gleicher Größe eingeteilt
- Jeder Block hat ein Bit in der Bitmap
  - 1 = belegt
  - 0 = frei
- Je größer der Block ...
  - ... desto kleiner die Bitmap
  - ... desto größer der interne Verschnitt
- Suchen freier Bereiche ist leider aufwendig:  $O(n)$

# Freispeicherverwaltung mit Bitmaps



- **Beispiel**
  - Hauptspeicher 512 MByte ( $2^{29}$  Byte)
  - Grundeinheit 2048 Byte ( $2^{11}$  Byte)
  - Ergibt 262144 Blöcke ( $2^{18}$ )
  - Belegungsdarstellung gespeichert in 8192 Worten zu je 32 Bit

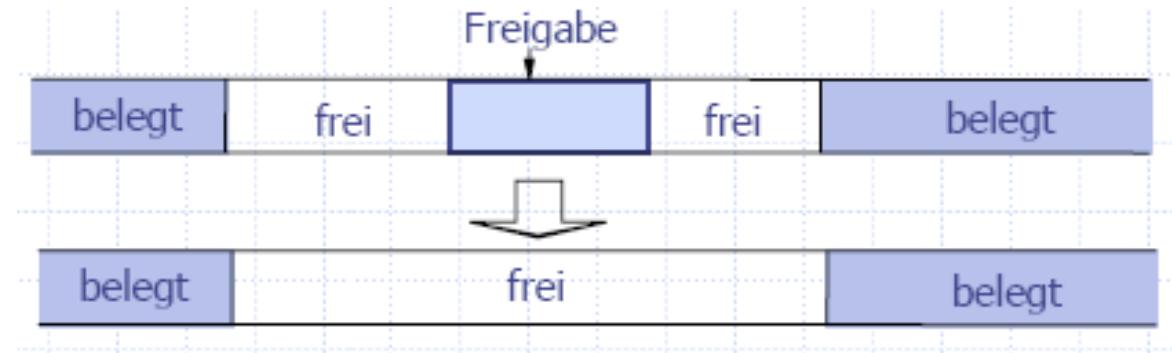
# Freispeicherverwaltung mit Listen



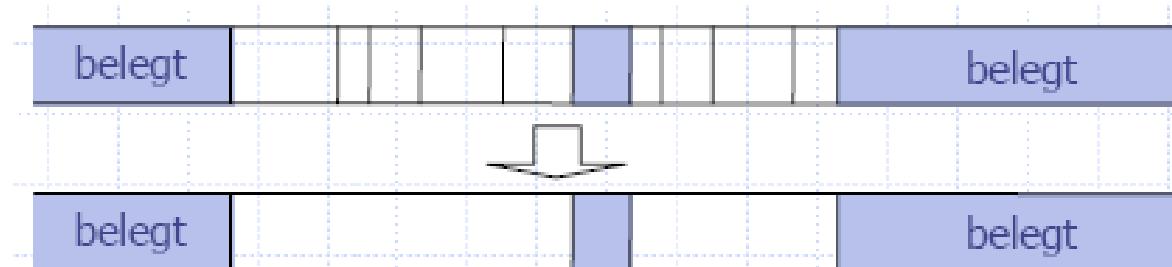
- Belegte und freie Speicherbereiche werden mit linearer Liste verkettet
- Speicherallokation
  - Finden eines „geeigneten“ freien Bereichs
- Speicherfreigabe
  - Zusammenführen angrenzender freier Bereiche

# Freispeicherverwaltung mit Listen

- Sofortiges Zusammenführen

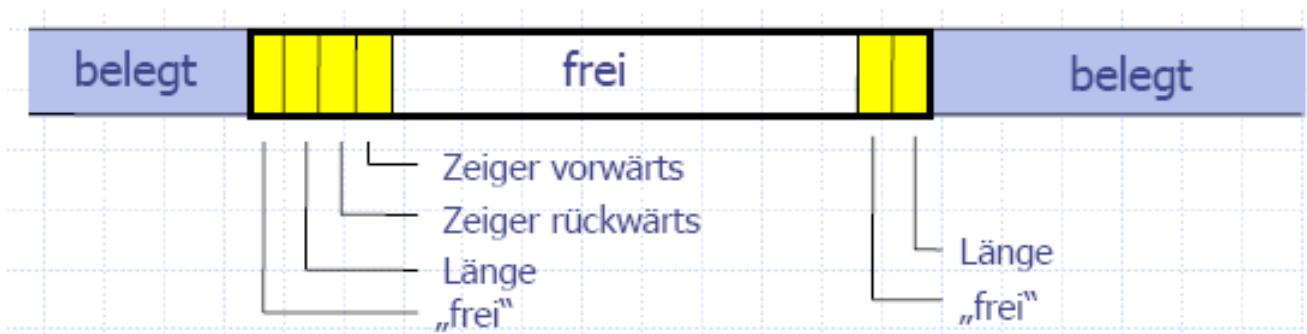


- Verspätetes Zusammenführen

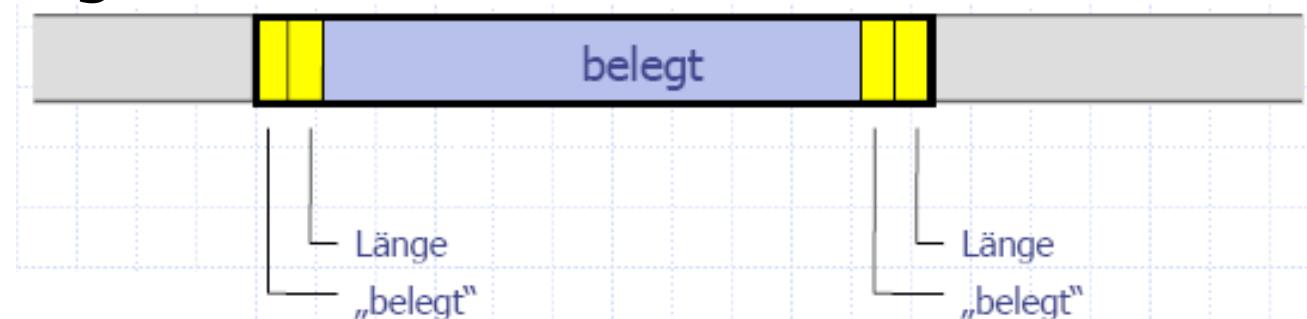


# Freispeicherverwaltung mit Listen

- Randkennzeichnungsverfahren
  - Es müssen keine extra Listen geführt werden
- Freies Stück



- Belegtes Stück

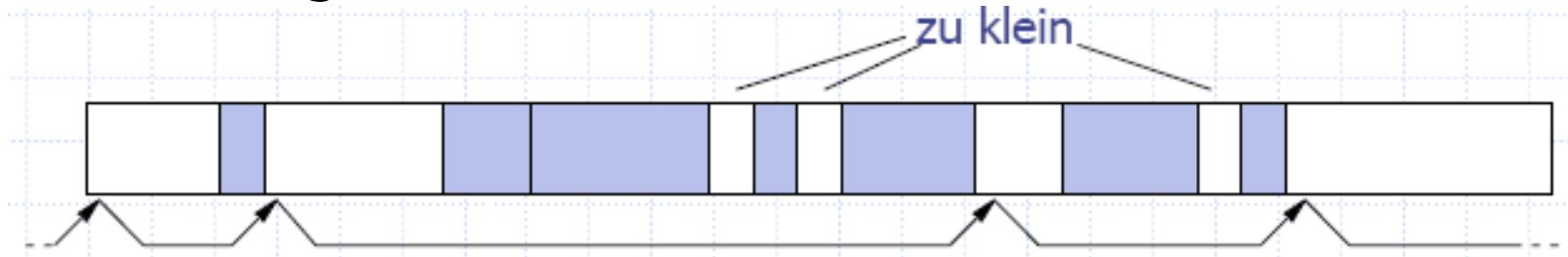


# Freispeicherverwaltung mit Listen

- Verwaltungsaufwand durch kleine Reststücke
  - Kleine Reststücke der Anforderung zuschlagen  
(Umwandlung von externen in internen Verschnitt)



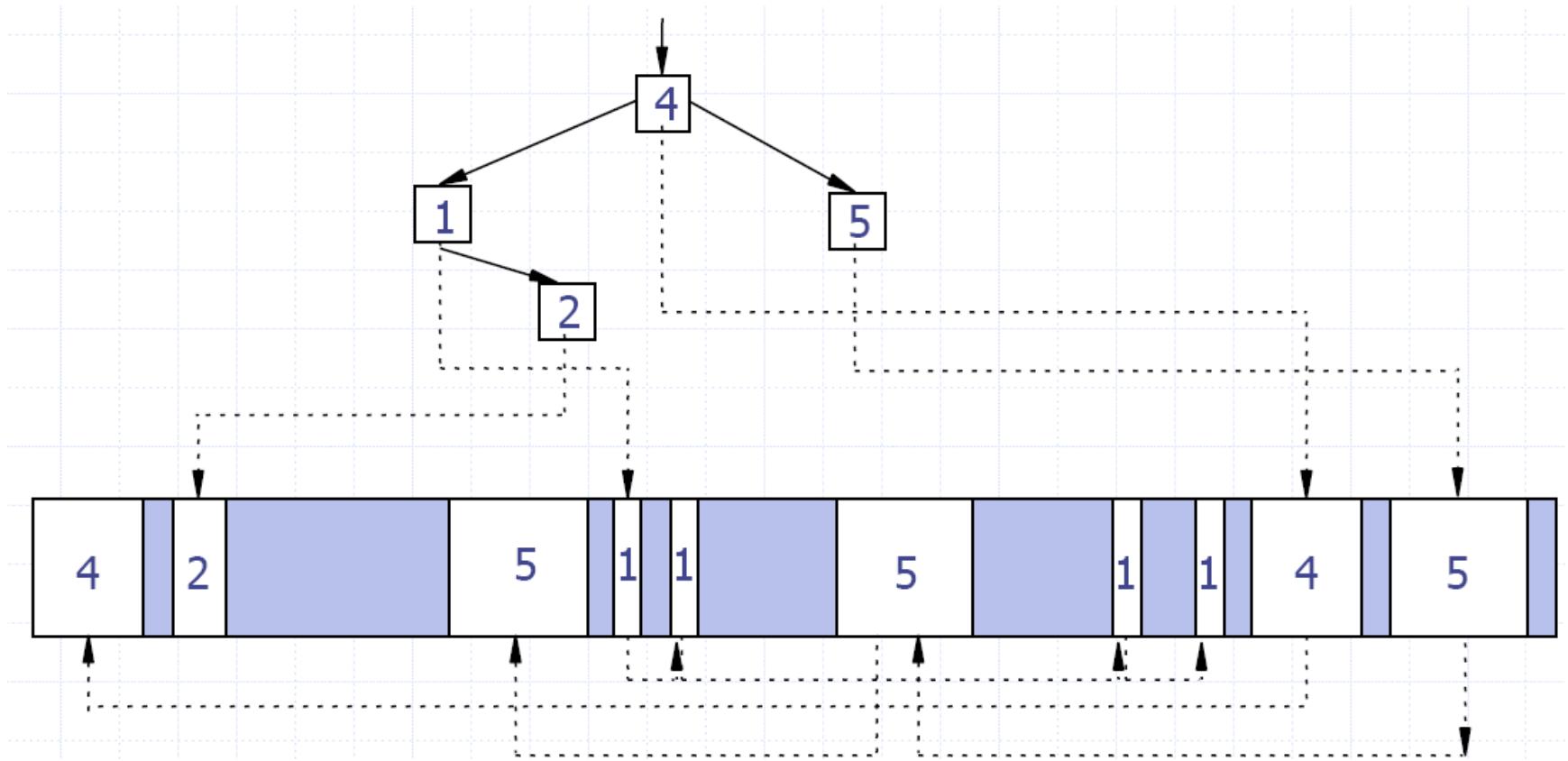
- Kleine Reststücke nicht in Freistückliste aufnehmen, aber bei Freigaben mit freigewordenem Nachbarn vereinigen



# Freispeicherverwaltung mit Listen

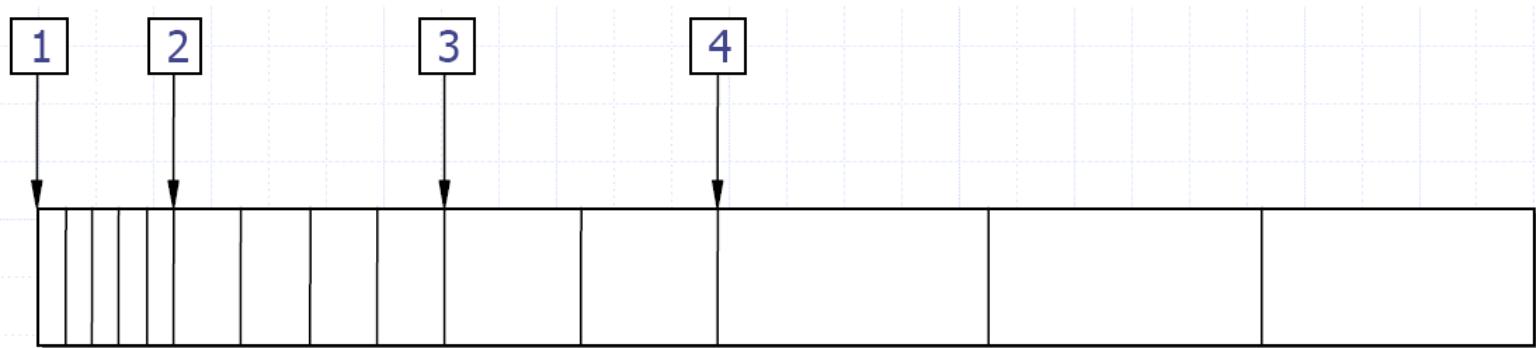
- Nachteil der Listen
  - Suchen nach freiem Speicherplatz dauert lange
- Idee
  - Leeren und belegten Speicher getrennt verwalten
  - Man muss nur noch die Liste mit leeren Speicherbereichen durchsuchen
- Nachteil der Idee
  - Theoretisch immer noch  $O(n)$
  - Speicherzusammenführung wird schwierig
  - Suchen benachbarter freier Bereiche ist  $O(n)$ 
    - Die Lösung mit nur einer Liste schafft es in  $O(1)$

# Freispeicherverwaltung mit Bäumen



# Freispeicherverwaltung mit „Konfektionsware“

- Die Größe angeforderter Speicherbereiche ist nicht statistisch zufällig
  - Bestimmte Größen kommen oft vor, andere fast nie
- Idee
  - Halte Speicherbereiche in beliebten Größen vorrätig
  - Das reduziert die interne und externe Fragmentierung
- Problem
  - Woher kennen wir die beliebten Größen?



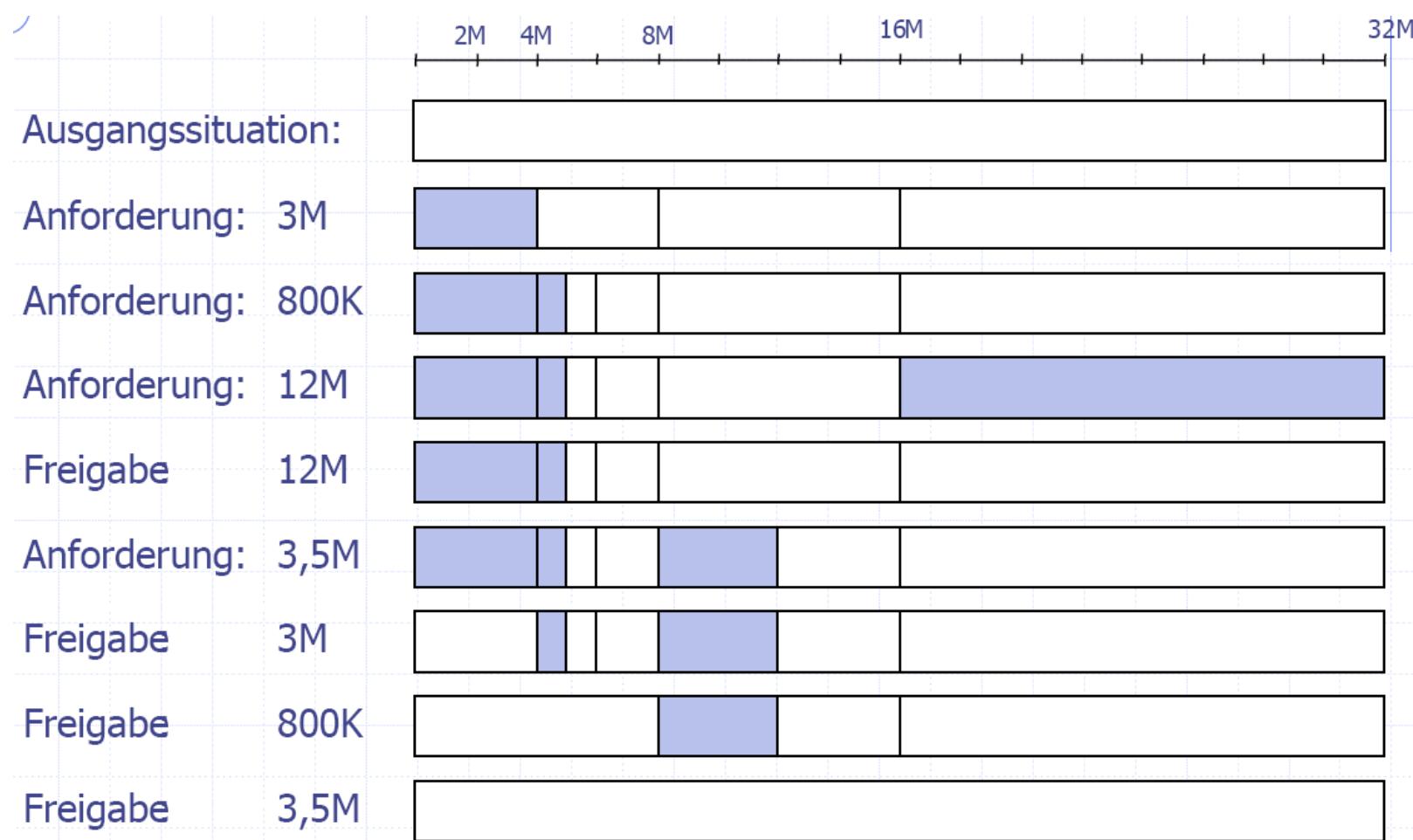
# Objekt-Orientierte Programme

- OO-Programme stellen besondere Herausforderungen an die Speicherverwaltung des Laufzeitsystems
  - Sehr viele Speicheranforderungen (jedes `new`)
  - Meistens kleine Speicherbereiche (<1KB)
  - Massenweise Freigabe durch den Garbage Collector
- Optimierungen
  - Beliebte Größen sind bekannt, weil das Laufzeitsystem die Klassen kennt
  - Es macht daher Sinn, „Konfektionsware“ vorrätig zu haben
  - Verspätetes Zusammenführen kann hilfreich sein

# Buddy System

- Alias „Halbierungsverfahren“
- Speicher besteht aus  $2^{k_{\max}}$  Bytes
  - Speicher wird in den Größen  $2^{\min}$ ,  $2^{\min+1}$ ,  $2^{\min+2}$ , ... vergeben
- Bei einer Anforderung wird das kleinste passende Stück ausgewählt
  - Angefordert wird  $2^x < A < 2^{x+1}$
  - Das passendste Stück ist  $2^{x+3}$  Bytes groß
  - Das wird halbiert in  $2^{x+2}$  und  $2^{x+2}$
  - Das erste Stück wird wieder halbiert in 2 mal  $2^{x+1}$
  - Das erste Stück der Größe  $2^{x+1}$  wird zugeteilt

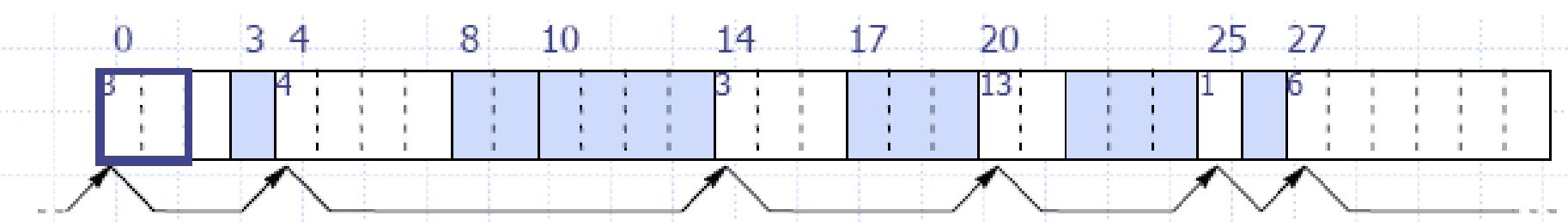
# Buddy System



# Freispeicherverwaltung

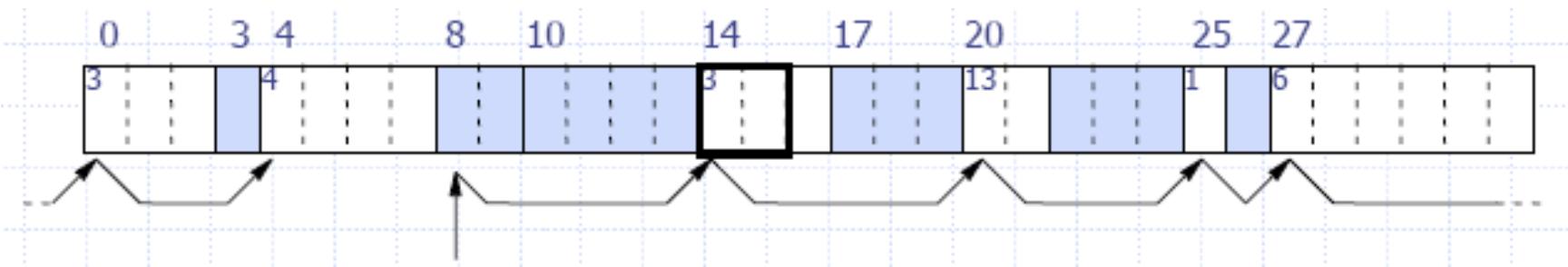
- Auswählen eines freien Blocks
- First Fit
  - Nimm den ersten freien Block, der groß genug ist
- Next Fit
  - Nimm von der aktuellen Position in der Freispeicherliste aus den nächsten, der groß genug ist
- Best Fit
  - Nimm den Block mit dem geringsten externen Verschnitt
- Worst Fit
  - Nimm den Block, der den größten externen Verschnitt hat
- Nearest Fit (eher für Speichermedien)
  - Wir geben eine Zieladresse vor und hätten den Speicherbereich gerne in der Nähe

# First Fit



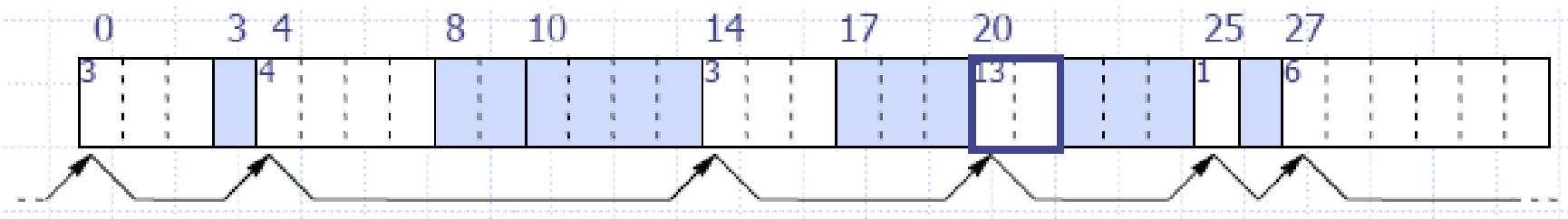
- Die (nach Adressen sortierte) Liste wird von vorne durchlaufen
- Das erste hinreichend große freie Stück wird genommen
- Eigenschaften
  - Geringer durchschnittlicher Suchaufwand
  - Im schlimmsten Fall aber immer noch  $O(n)$
  - Zerstückelung des Speichers (externer Verschnitt)
  - Konzentration der belegten Stücke am Anfang und dadurch Erhöhung des Suchaufwands

# Next Fit



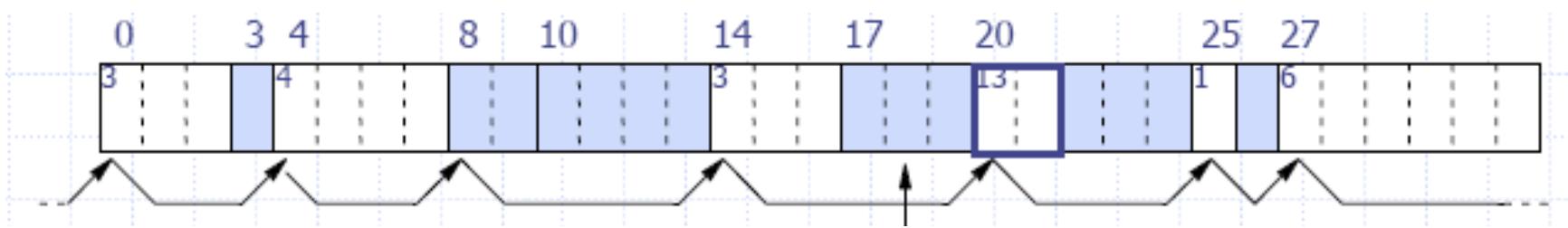
- Liste wird zyklisch durchlaufen
- Suche beginnt an der Stelle, wo die letzte Belegung stattgefunden hat
- Eigenschaften
  - Wie First-Fit, vermeidet aber den Nachteil der Konzentration der Belegungen am Anfang
  - Dadurch bessere Suchzeiten

# Best Fit



- Das kleinste hinreichend große Stück wird genommen
- Eigenschaften
  - Bei Sortierung nach Adresse muss die gesamte Freistückliste durchsucht werden. Es empfiehlt sich daher Sortierung nach Größe
  - Im Prinzip bessere Speicherausnutzung, da kleinere Anforderungen auch durch kleinere freie Bereiche erfüllt werden und nicht große Stücke „angeknabbert“ werden
  - Neigt allerdings dazu, sehr kleine freie Stücke zu erzeugen, mit denen man gar nichts mehr anfangen kann

# Nearest Fit

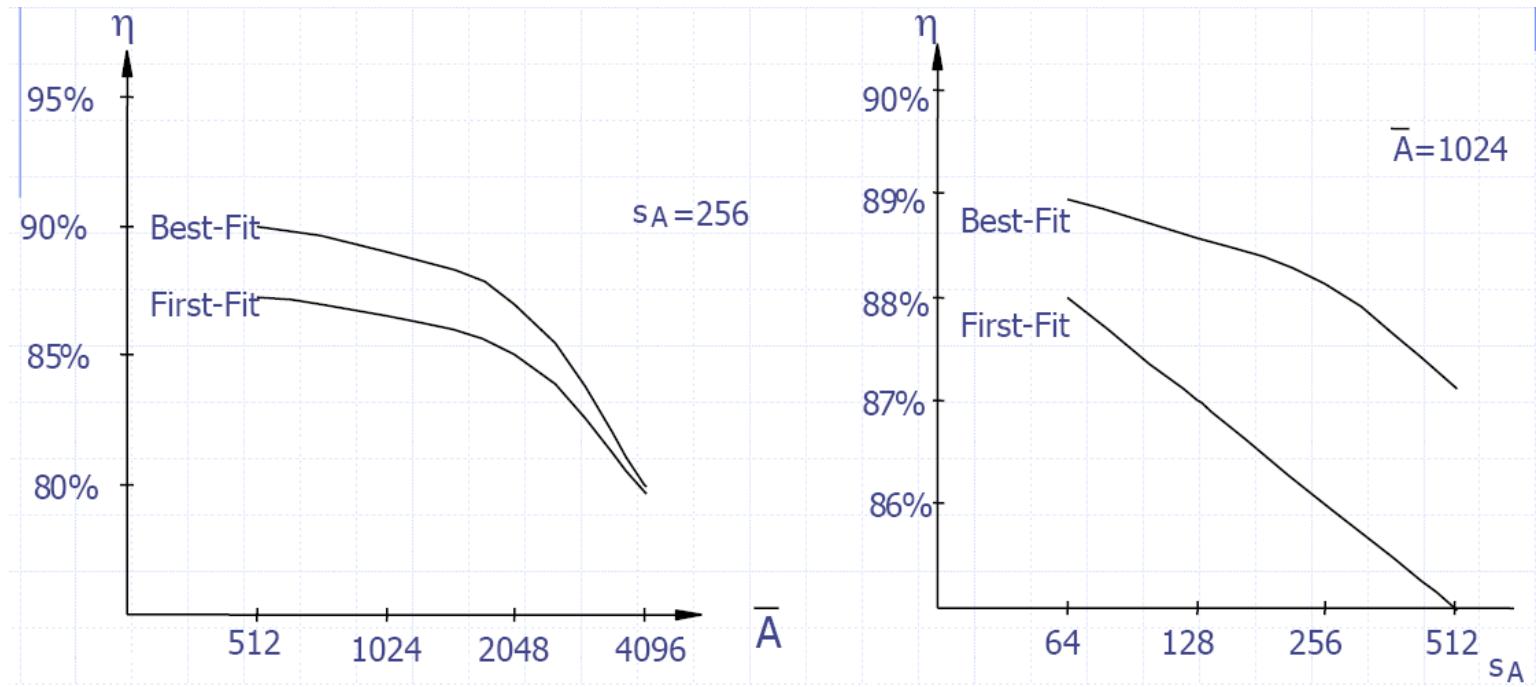


- Eine „Wunschadresse“ für das angeforderte Stück wird übergeben.
- Der Algorithmus beginnt dann eine First-Fit-Suche von der angegebenen Adresse an
- Beim Plattspeicher kann man sehen, dass es vorteilhaft ist, Armbewegungen zu minimieren
- Kennt man die Zugriffsreihenfolge oder Zugriffshäufigkeit, so kann man durch die Belegung die Armbewegung beeinflussen.
- Dateikataloge können auf die mittleren Zylinder gelegt werden.
- Bei der Erweiterung sequentieller Dateien sollten die neuen Blöcke in der Nähe der bisher belegten liegen

# Worst Fit

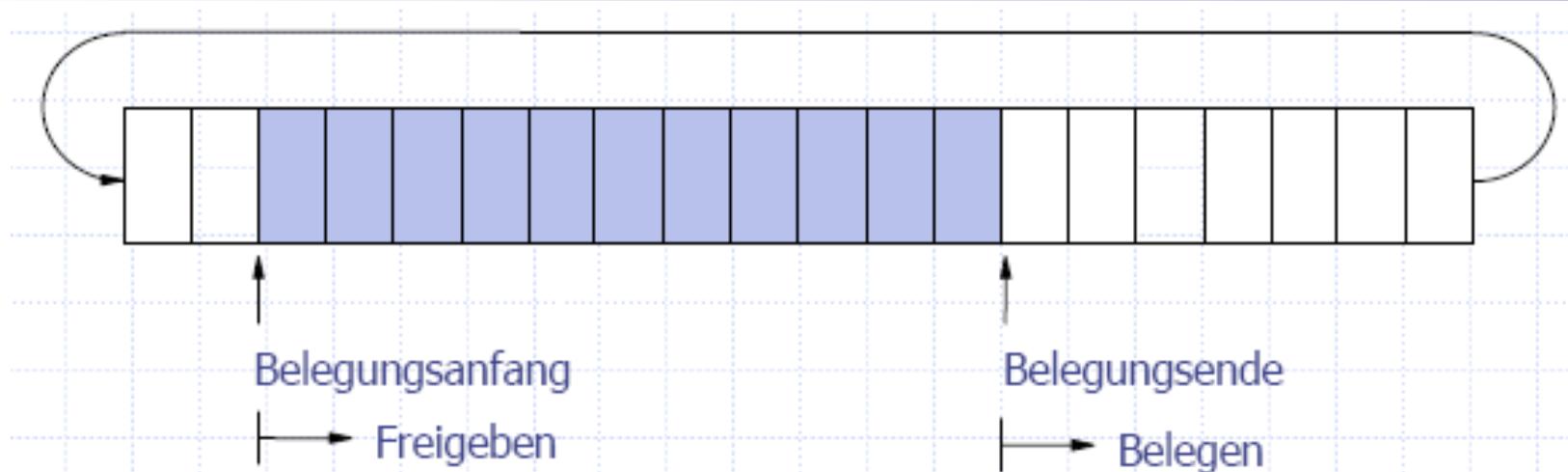
- Der Algorithmus liefert das Stück, welches am schlechtesten passt
  - Normalerweise das größte Stück
- Idee
  - Wir verhindern so das Entstehen kleiner Restschnipsel (externer Verschnitt)
  - Diese Restschnipsel machen Arbeit bei der Verwaltung und sind für nichts zu gebrauchen
  - Wenn wir immer vom größten Stück abschneiden, bleiben auch große Stücke übrig
- Leider keine wirklich gute Idee ☹

# Vergleich der \*-Fit Algorithmen



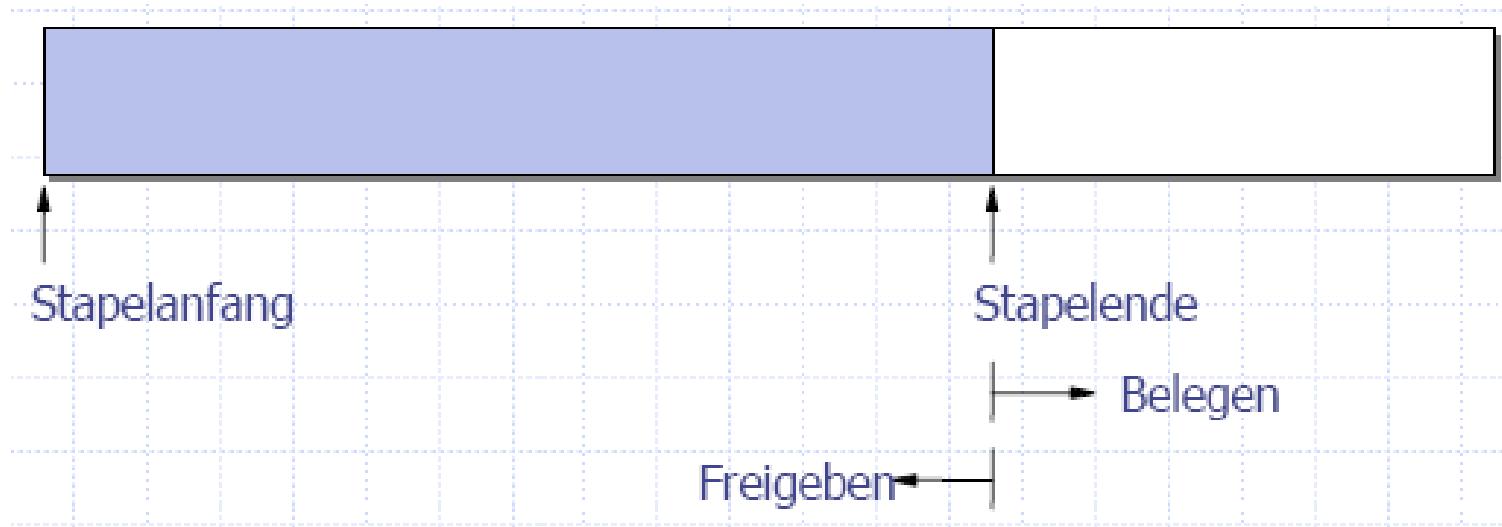
- $\bar{A}$  ist der Mittelwert der angeforderten Speicherbereiche
- $s_a$  ist die Standardabweichung
- Ergebnis
  - Je größer die angeforderten Bereiche und je größer die Standardabweichung, desto höher ist die externe Fragmentierung

# Spezialisierte Verfahren: Ringpuffer



- Belegen und Freigeben in gleicher Reihenfolge (FIFO)
- Gleich lange Stücke
- Kein Suchen
- Kein externer Verschnitt
- Automatische sofortige Wiedereingliederung

# Spezialisierte Verfahren: Stack, Kellerspeicher



- Belegen und Freigeben in umgekehrter Reihenfolge (LIFO)
- Beliebig lange Stücke
- Kein Suchen
- Wenig externer Verschnitt
- Automatische sofortige Wiedereingliederung

# Overlays

- Der komplette Prozess passt nicht in den Speicher
  - Swapping ist hier keine Hilfe
  - Typisches Problem zu Zeiten von MS-DOS
  - Moderne Systeme nutzen lieber virtuellen Speicher
- Idee
  - Teile des Programms mit einem anderen überschreiben (dem Overlay)
  - Das überschriebene Programm muss nicht ausgelagert werden
    - Es kann von Platte/Diskette wieder geladen werden
    - Daten hingegen müssten ausgelagert werden
  - Das Aufteilen eines Programms in Stücke, die sich nur selten gegenseitig aufrufen ist kompliziert

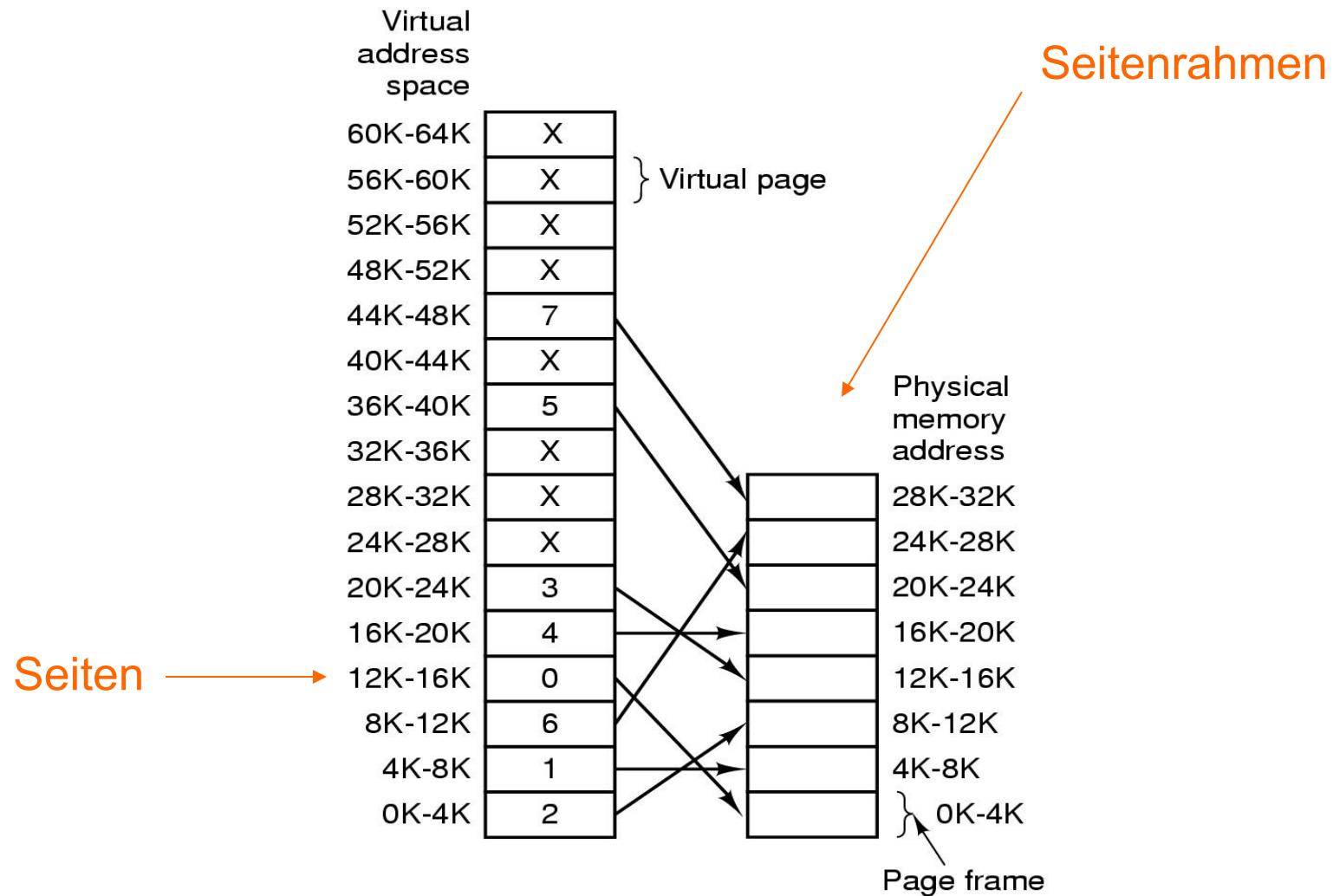
# Virtueller Speicher

- Problem 1
  - Ein Prozess passt nicht mehr komplett in den Speicher
  - Swapping hilft nicht, denn das lagert komplett Prozesse aus
  - Overlays sind unpraktisch
- Problem 2
  - Ein Prozess „wächst“ zur Laufzeit
  - Wo bekommt man einen größeren zusammenhängenden Speicherbereich her?
- Problem 3
  - Externe Fragmentierung verschwendet Speicher
  - Memory Compaction ist teuer

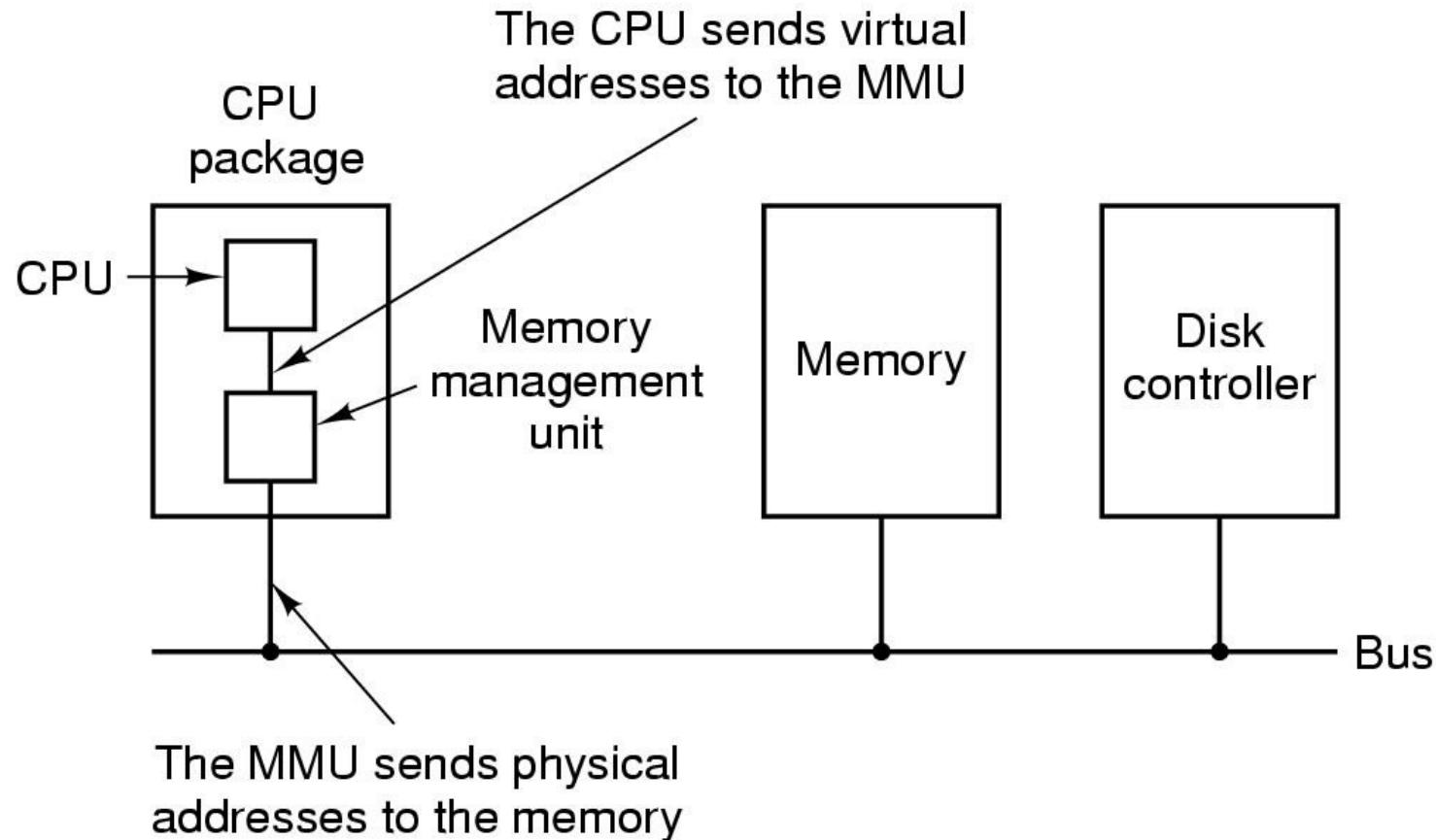
# Virtueller Speicher

- „There is no problem that cannot be solved with another level of indirection“
- Jeder Prozess sieht einen linearen zusammenhängenden (aber virtuellen) Speicherbereich
- Physikalisch setzt dieser Speicherbereich sich aus Blöcken zusammen
  - Die Blöcke können beliebig im physikalischen Speicher verstreut sein
- Wir brauchen eine Abbildung:  
Virtuelle Adresse → Physikalische Adresse
  - Diese Umrechnung fällt bei jedem Speicherzugriff an

# Paging



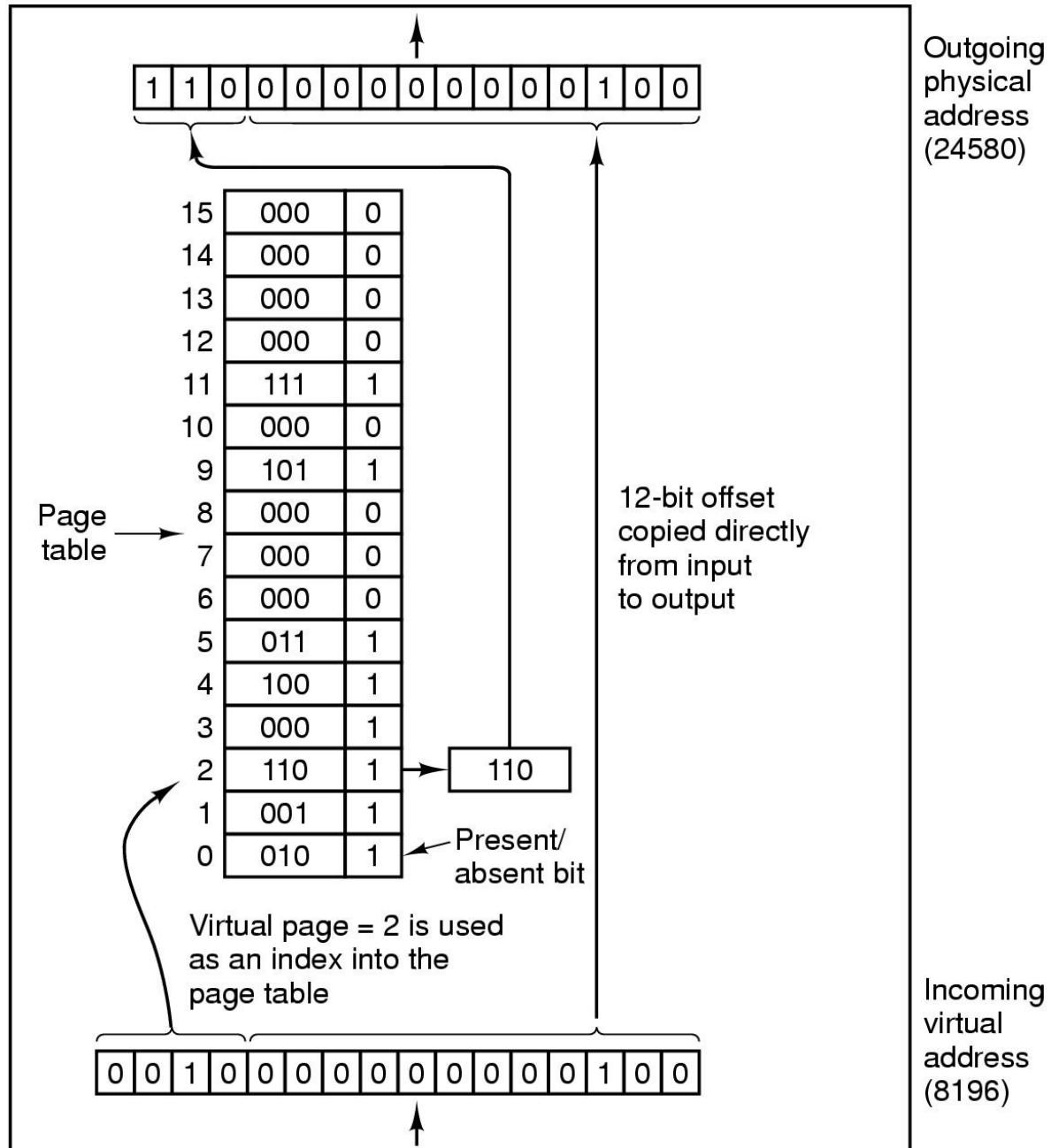
# Memory Management Unit



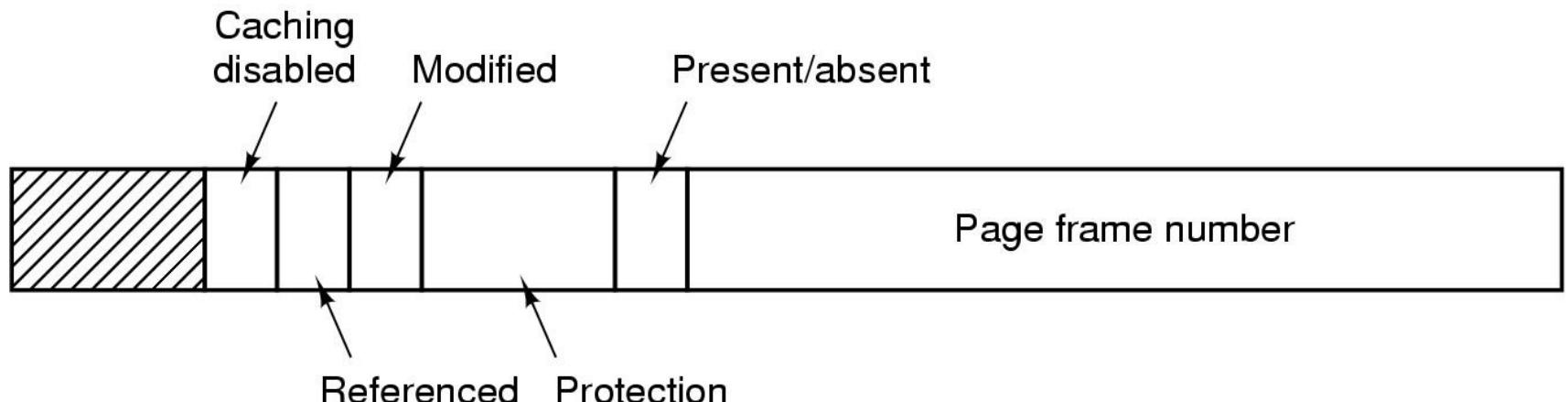
# Seitentabelle

- Die Seitentabelle hat eine Zeile für jeden virtuellen Speicherblock (Seite)
  - Wird von der MMU benutzt
- Jede Zeile enthält mindestens
  - Nummer des physikalischen Speicherblocks (Seitenrahmen)
  - Bit, das anzeigt, ob der Speicherblock ausgelagert ist
- Zu erwartende Probleme
  - 1) Seitentabelle kann sehr groß werden
  - 2) Umrechnung von virtueller Adresse in physikalische Adresse muss sehr schnell gehen

# Seitentabelle

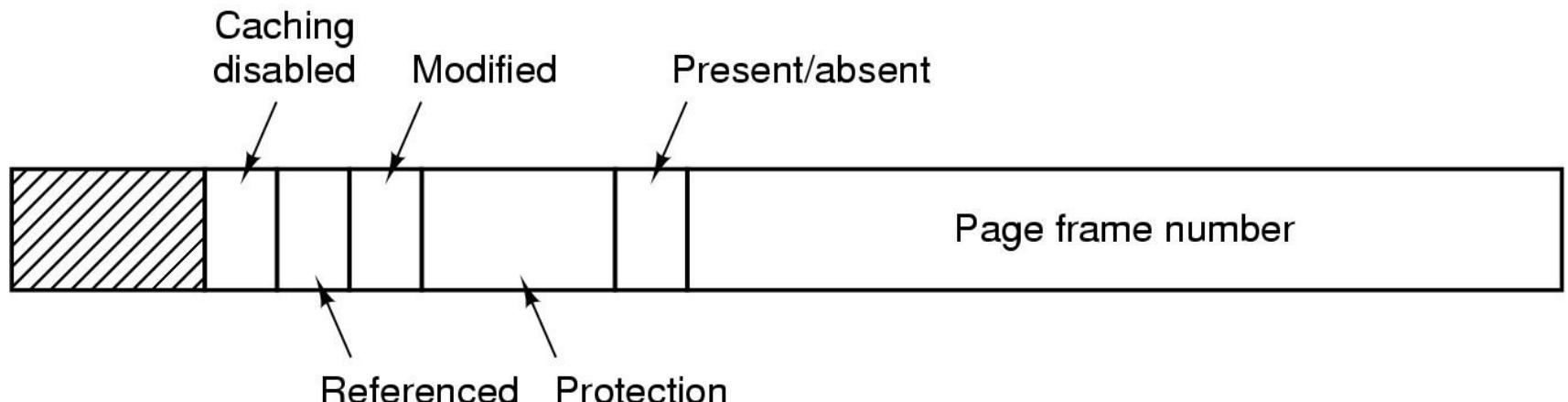


# Zeile einer Seitentabelle



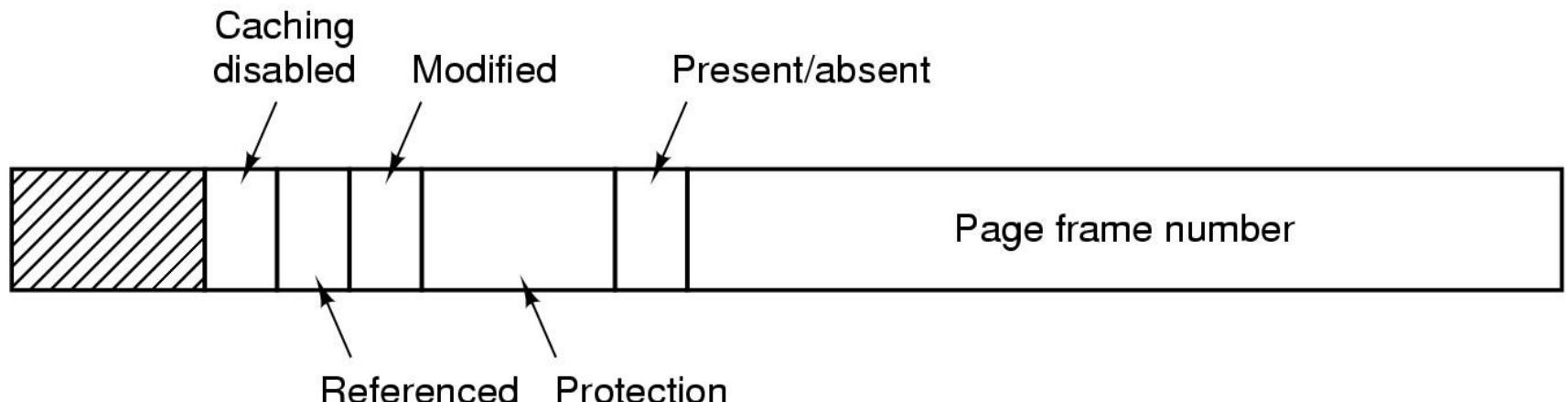
- **Caching Bit**
  - Manche Rechner blenden IO Ports in den Speicher ein
  - Diese IO Ports dürfen nicht im Cache vorgehalten werden
    - Sie können ihren Wert jederzeit ändern
    - Im Cache wäre dann der falsche Wert
  - Daher schaltet man das Caching für Speicherbereiche ab, auf die IO Ports abgebildet werden

# Zeile einer Seitentabelle



- **Modified Bit**
  - Wird nach Schreibzugriff gesetzt
  - Diese Seite muss beim Auslagern wieder auf die Festplatte geschrieben werden
  - Ansonsten kann man die alte Kopie auf der Festplatte benutzen (insofern die noch erhalten ist)
- **Referenced Bit**
  - Wird nach jeder Art von Zugriff gesetzt
  - Benötigt für Seitenersetzungsalgorithmen

# Zeile einer Seitentabelle



- Protection Bit (nur ein Bit)
  - 0 = Schreiben ist verboten
- Protection Bits (3 Bit)
  - R Bit: Lesen erlaubt?
  - W Bit: Schreiben erlaubt?
  - X Bit: Ausführen erlaubt?

# Page Miss

- Wenn das Present Bit 0 ist ...
  - Die MMU erzeugt einen Interrupt
  - Die CPU führt einen Handler des BS aus
  - Das BS muss die fehlende Seite von der Festplatte laden
    - Das kann aus Sicht der CPU lange dauern
  - Daher wird der Prozess schlafen gelegt (Waiting)
  - Ein anderer Prozess bekommt die CPU
- Wenn die fehlende Seite geladen wurde ...
  - Der Prozess kann weiterrechnen (Ready)
  - Der letzte Befehl (der den Interrupt auslöste) wird wiederholt

# Page Miss

- Die Seitentabelle sagt nicht, wo auf der Festplatte eine ausgelagerte Seite liegt
  - Das ist Sache des Betriebssystems
  - Dafür existieren eigene Datenstrukturen
  - Die Seitentabelle ist nur für die MMU gedacht
- Eine Seite kann nicht present und gleichzeitig auch nicht ausgelagert sein
  - Im virtuellen Adressraum gibt es große Löcher, d.h. viele Seiten denen nie physikalischer Speicher zugeordnet wurde
  - Ein Prozess muss das BS auffordern, Seiten physikalischen Speicher zuzuordnen
  - Versucht ein Prozess, auf ein solches Loch zuzugreifen, löst er einen Interrupt aus

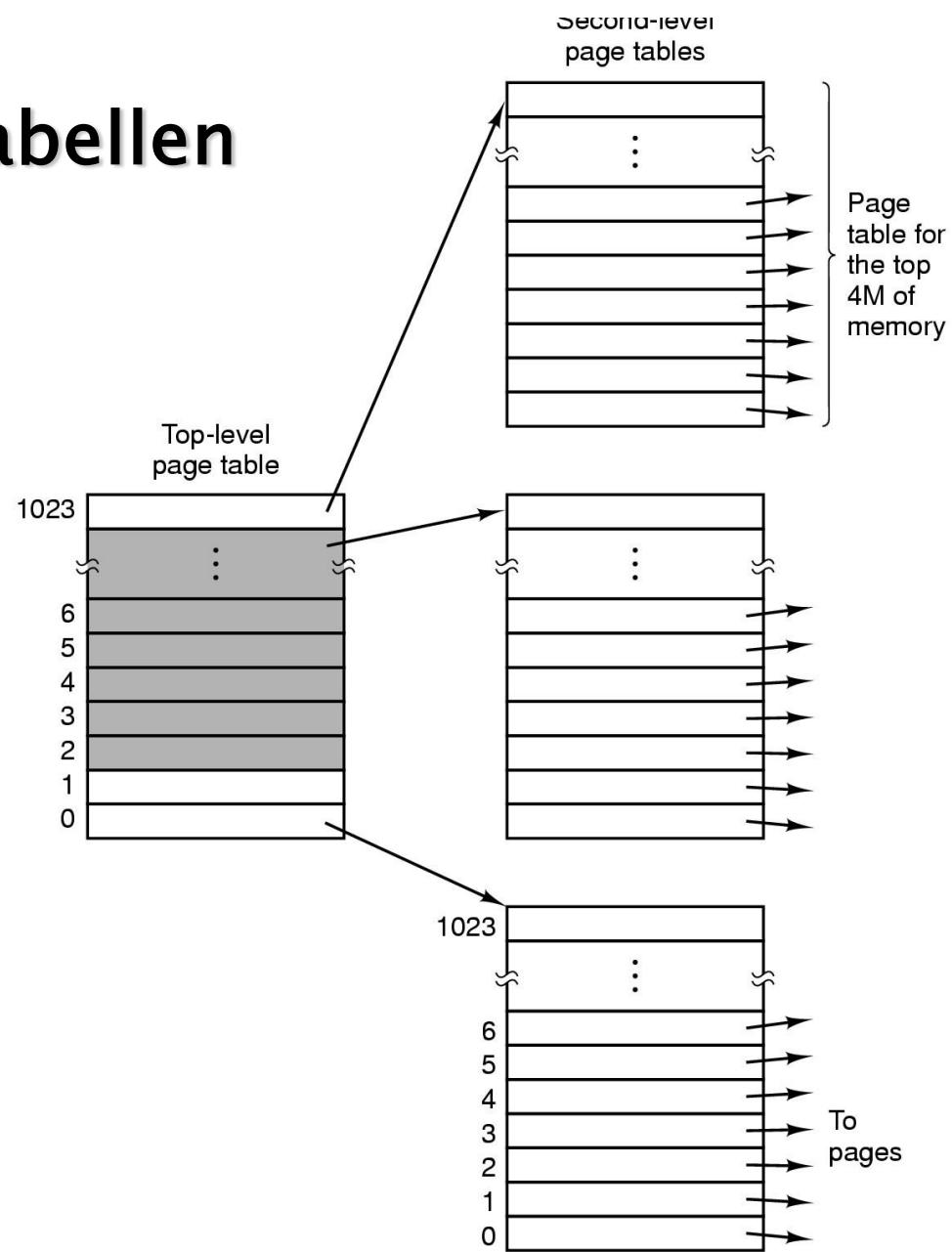
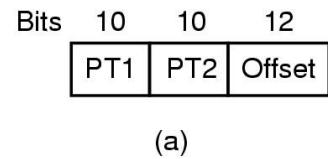
# Hierarchische Seitentabelle

- Wir brauchen viele sehr große Tabellen
  - Jeder Prozess hat eine Tabelle → viele Tabellen
  - Jede Tabelle bildet den kompletten 32 Bit Adressraum ab → große Tabellen
- Seitentabellen sind zum Großteil leer
  - Jede Tabelle deckt 4 GByte Speicher ab
  - Bei 100 Tabellen sind das insgesamt 400 GByte virtueller Speicher
  - Die Rechner haben aber nur 1 GByte Speicher plus 1 GByte Auslagerungsdatei
  - → Von 400 Tabelleneinträgen sind 398 leer!

# Hierarchische Seitentabelle

- Leere Bereiche sind (im virtuellen Adressraum) meist zusammenhängend
  - Typisch ist eine Belegung am oberen und unteren Rand
  - Dazwischen gibt es riesige (virtuelle) Speicherlöcher
  - Für diese „virtuellen Löcher“ verschwenden die Seitentabellen physikalischen Speicher
- Hierarchische Seitentabellen
  - Verbrauchen keinen Speicher um riesige Löcher zu verwalten
  - Dafür sind sie langsamer als einfache Tabellen

# Hierarchische Seitentabellen



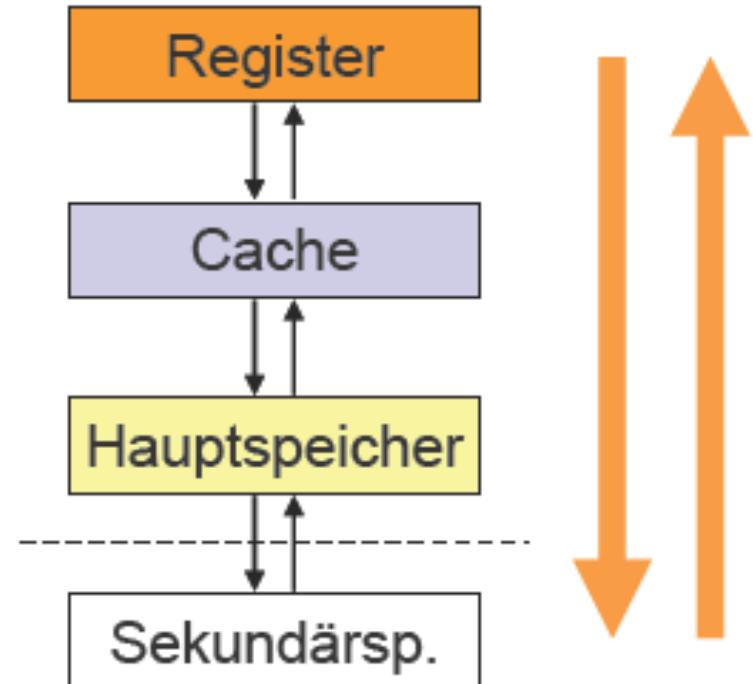
# Größe der Seitentabelle (32 Bit Maschinen)

- Berechnung der Tabellengröße
  - 32 Bit Maschine
  - 4 GByte Adressraum
  - 4 KByte Speicherblöcke
  - Etwa 1 Millionen Speicherblöcke
  - → etwa 1 Millionen Zeilen in der Tabelle
  - 32 Bit = 4 Byte pro Zeile
  - → 4 MByte für die Tabelle
- Die Tabelle ist zu groß für die MMU
- 4 MByte Tabelle pro Prozess ist Verschwendungen
- Jeder Prozesswechsel müsste 4 MByte Tabelle in die MMU kopieren → viel zu langsam

# Speicherort für Seitentabellen

- Register der MMU
  - Maximale Performance bei Speicherzugriffen
  - MMU wäre sehr teuer
  - Prozesswechsel wären extrem langsam
- Hauptspeicher
  - Schlechte Performance bei Speicherzugriffen
  - MMU wäre billig
  - Prozesswechsel wären sehr schnell
- Beide Lösungen sind inakzeptabel

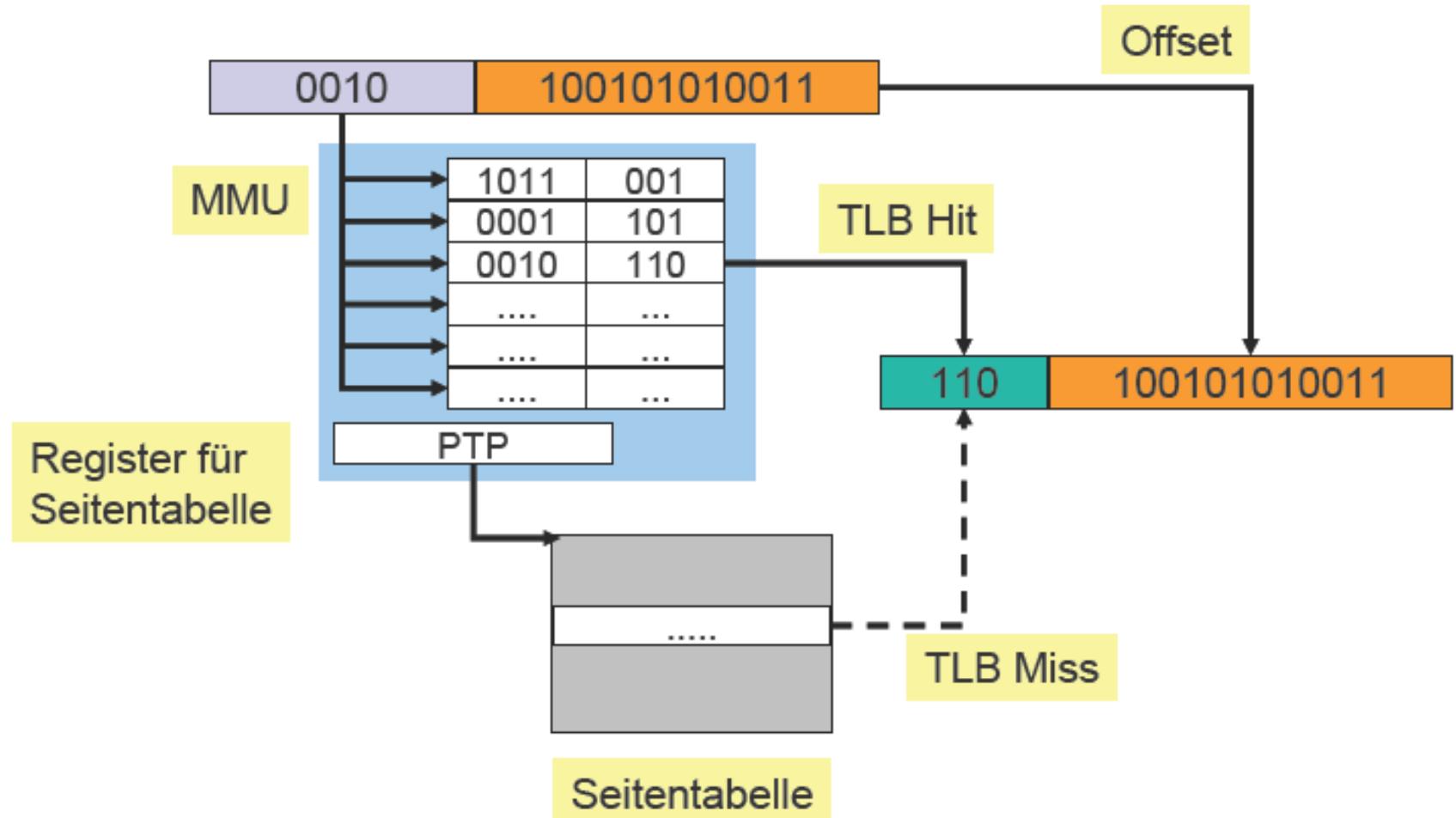
*Speicherhierarchie*



# Assoziativspeicher

- Kompromiss zwischen Tabelle im Hauptspeicher und Tabelle in der MMU
  - Die zuletzt genutzten Tabelleneinträge sind direkt in der MMU gespeichert
  - Die komplette Tabelle liegt im Hauptspeicher
  - Analog zum Cache auf der CPU
- Annahme
  - Prozesse haben eine starke „Lokalität“
  - D.h. sie greifen oft auf die selben Seiten zu

# Assoziativspeicher



# Assoziativspeicher

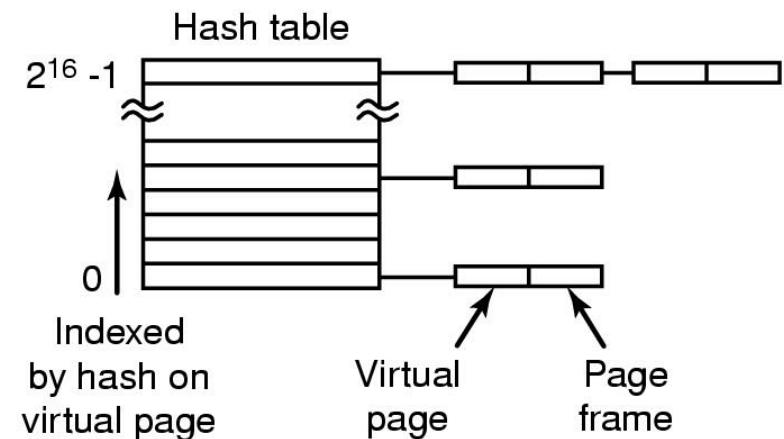
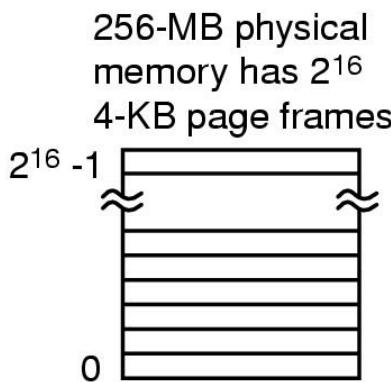
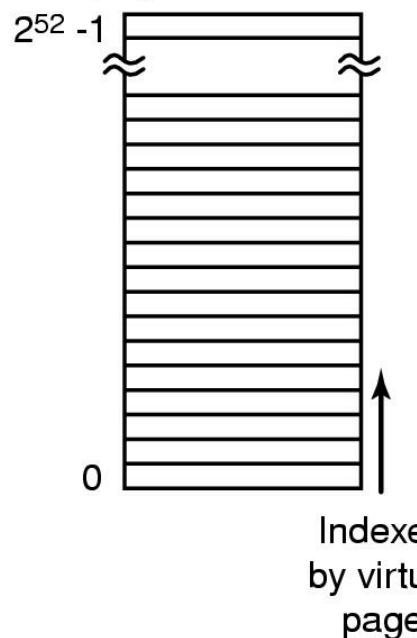
<b>Valid</b>	<b>Virtual page</b>	<b>Modified</b>	<b>Protection</b>	<b>Page frame</b>
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

# 64 Bit CPUs

- Seitentabellen wären viel zu groß
  - Auch hierarchische Tabellen helfen nicht mehr
- Beobachtung
  - Der virtuelle Adressraum ist riesig groß
  - Der physikalische Adressraum vergleichsweise klein
- Invertierte Seitentabellen
  - Normalerweise eine Zeile pro Seite (virtuell)
  - Statt dessen eine Zeile pro Seitenrahmen (physikalisch)
- Problem invertierter Seitentabellen
  - Abbildung von virtuell auf physikalisch ist langsam
  - Man müsste die ganze Tabelle absuchen

# Invertierte Seitentabellen

Traditional page table with an entry for each of the  $2^{52}$  pages



# Seitenersetzung

- Page Miss & Alle Seitenrahmen besetzt
  - Eine Seite muss ausgelagert werden
  - Die fehlende Seite kommt in den freien Rahmen
- Optimierung
  - Welche Seite auslagern?
  - Auslagern einer Seite, die bald wieder gebraucht wird ist eine schlechte Idee
  - Aber wir können nicht in die Zukunft blicken
  - Zukünftige Zugriffe müssen aus vergangenen Zugriffen extrapoliert werden
    - Verschiedene Strategien
    - Keine ist immer optimal

# Optimale Seitenersetzung

- Gute Nachricht
  - Es gibt einen optimalen Algorithmus
- Schlechte Nachricht
  - Er ist nicht praktikabel, denn er schaut in die Zukunft
  - Sinnvoll, um andere Algorithmen zu vergleichen
- Algorithmus
  - Seiten  $s_1, \dots, s_n$
  - Seitenzugriffe  $Z = \{s_{t1}, s_{t2}, \dots, s_{tk}\}$  zu Zeiten  $t_i$   
(Hier liegt die unrealistische Annahme)
  - Lagere die Seite aus, die gemäß Z am längsten nicht mehr gebraucht wird

# Beispiel: Optimale Seitenersetzung

Seiten:  $n = 8$

Seitenrahmen:  $m = 3$



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



Insgesamt 9 Seitenfehler, davon sind 6 unvermeidbar, denn jede Seite muss irgendwann zum ersten mal geladen werden

# Benchmarks für Seitenersetzung

- Für Seiten  $s_1, \dots, s_n$ , Seitenrahmen  $r_1, \dots, r_m$  mit  $m < n$  und Zugriffszeiten Z
  - Berechne die Anzahl der Seitenersetzungen: #Opt
  - Damit erhalten wir die optimale Anzahl Ersetzungen
- Der zu vergleichende Algorithmus sieht nun dieselben Seitenzugriffe
  - Er kennt aber die Zugriffszeiten Z nicht
  - Wir zählen die Seitenersetzungen: #Alg
- Die Effizienz berechnet sich durch Division  
 $\#Opt / \#Alg \leq 1$ 
  - Je größer der Wert, desto besser

# Not Recently Used (NRU) Algorithmus 1/3

- „Entferne eine Seite, die in letzter Zeit nicht mehr benutzt wurde“
- Wie stellt man fest, dass eine Seite benutzt wurde?
  - Referenced und Modified Bits aus der Seitentabelle lesen
- Wie bestimmt man „in letzter Zeit“
  - Periodisch werden die Referenced Bits gelöscht
  - Wenn das Referenced Bit gesetzt ist, dann wurde nach der letzten Löschung zugegriffen
  - Periodisches Löschen der M Bits ist nicht möglich
    - Wird beim Paging benötigt

# Not Recently Used (NRU) Algorithmus 2/3

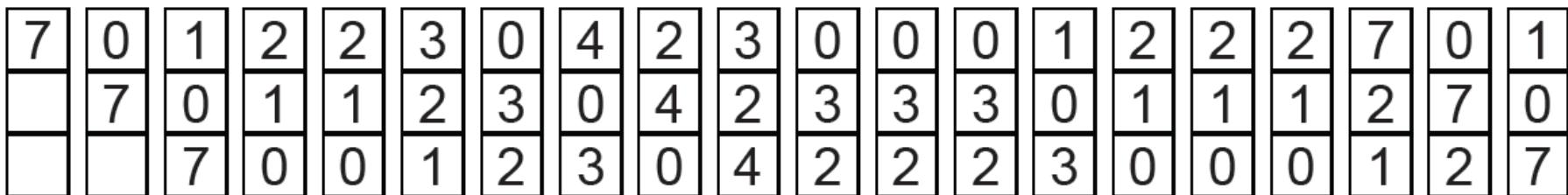
- Klassifizierung der Seiten
  1. Nicht gelesen, nicht geschrieben ( $R=0, M=0$ )
  2. Nicht gelesen, aber geschrieben ( $R=0, M=1$ )
  3. Gelesen, aber nicht geschrieben ( $R=1, M=0$ )
  4. Gelesen und geschrieben ( $R=1, M=1$ )
    - Der zweite Fall entsteht durch Schreiben nach periodischem Löschen
- Geschriebene Seiten sind teurer als gelesene
  - Wenn  $R=1$  und  $M=0$ , dann kann man die Seite eventuell einfach überschreiben
  - Wenn  $M=1$ , dann muss man die Seite auf die Festplatte zurückschreiben → teuer
  - Eventuell die Klassen 2 und 3 vertauschen?

# Not Recently Used (NRU) Algorithmus 3/3

- Algorithmus
  - Initial werden alle R Bits gelöscht
  - Periodisches Löschen der R Bits
    - Beispielsweise während eines Kontextwechsels
  - Wenn eine Seite ersetzt werden soll
    - Klassifiziere die Seiten
    - Entferne eine Seite aus der niedrigsten Klasse

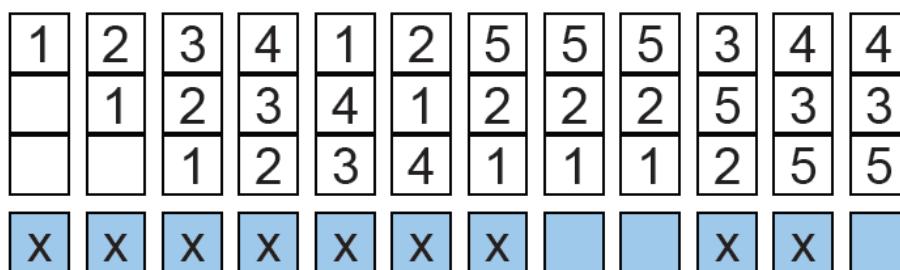
# FIFO Algorithmus

- „Ersetze die Seite, die am längsten im Speicher ist“
  - Die erste, die reinkommt ist auch die erste, die wieder raus muss (first in, first out)

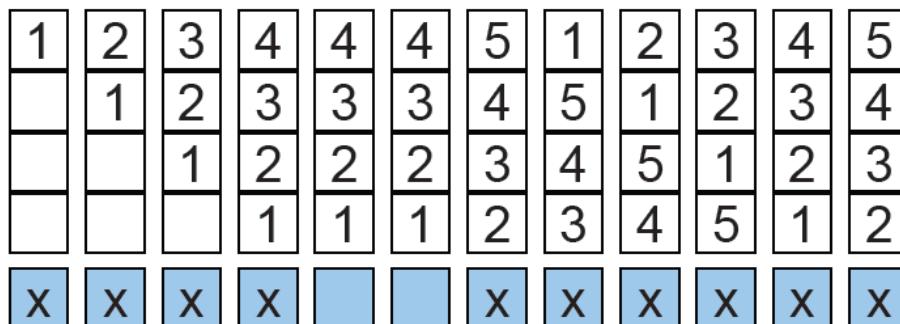


# FIFO Anomalie

- Mehr Seitenrahmen sorgen nicht automatisch für weniger Seitenfehler. Im Gegenteil!



9 Seitenfehler



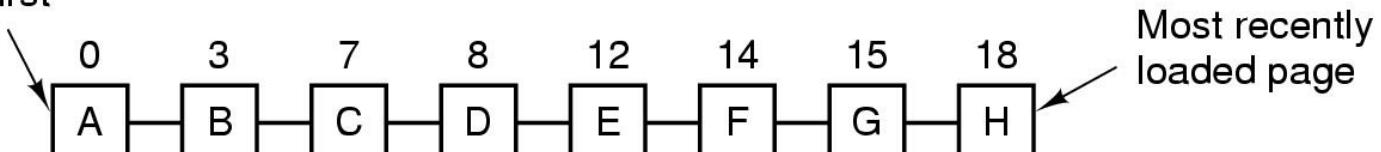
10 Seitenfehler

# Second Chance Algorithmus

- „Entferne die älteste Seite, auf die zuletzt nicht mehr zugegriffen wurde“
- Kombination aus FIFO und NRU
  - Wir halten eine lineare Liste aller Seiten
  - Alte Seiten vorne, neuere Seiten weiter hinten
  - Jede Seite hat ein R Bit in der Seitentabelle
- Algorithmus
  - Nimm älteste Seite aus der Liste
  - $R=0?$  → rauswerfen
  - $R=1?$  → R löschen und Seite hinten an die Liste anhängen

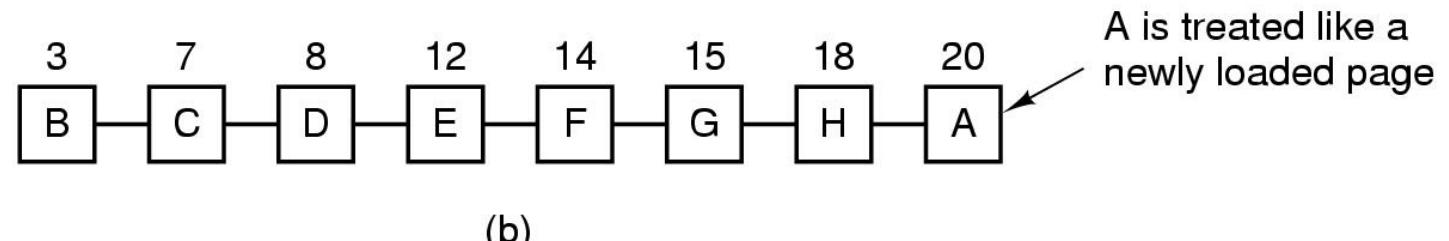
# Second Chance Algorithmus

Page loaded first



(a)

Most recently loaded page

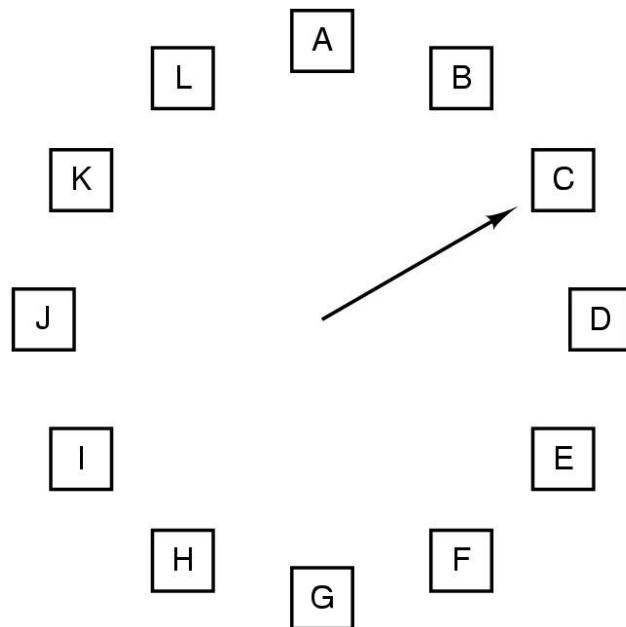


(b)

- (a) Seitenliste in FIFO Ordnung
  - Zahlen geben Zeitpunkt der Seiten-Einlagerung an
- (b) Seitenliste zum Zeitpunkt 20
  - A hatte R=1 und bekam daher eine “second chance”

# Clock Algorithmus

- Verhält sich genau wie Second Chance Algorithmus
- Vorteil liegt in der besseren Datenstruktur



When a page fault occurs,  
the page the hand is  
pointing to is inspected.  
The action taken depends  
on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

# Least Recently Used (LRU) Algorithmus

- Beobachtung
  - Seiten, auf die lange nicht mehr zugegriffen wurde, werden auch in naher Zukunft wenig gefragt sein
- LRU: “Verwerfe die Seite, die am längsten unbenutzt ist“
- Problem
  - Wie soll man sich merken, welche Seite wann zu letzt benutzt wurde?
  - Eine sortierte lineare Liste ist unrealistisch
    - Jeder Seitenzugriff würde zu einer Änderung in der Liste führen
    - Nicht effizient in Hardware/Software realisierbar

# Least Recently Used (LRU) Algorithmus

- Teure Hardware-Lösung
  - 64-Bit Zähler wird nach jedem Assembler-Befehl erhöht
    - 4 Milliarden Operationen pro Sekunde
    - 64 Bit Zähler hält für  $2^{32}$  Sekunden
  - Bei jedem Seitenzugriff speichere den Zähler in der Seitentabelle
    - Verdreifachung der Größe der Seitentabellen (32bit CPU)
    - Große Seitentabelle muss aus dem Hauptspeicher gelesen werden (von der MMU)
    - Die MMU muss die große Tabelle wieder in den Hauptspeicher zurückschreiben
  - Bei Seitenfehler, such die Seite mit dem kleinsten Zähler in der Seitentabelle

# LRU mit Matrix

- Für n Seiten, führen wir eine Seitenmatrix mit  $n \times n$  Bits
  - Das ist leider Speicherverschwendung
- Die Bits in Zeile i geben an, wie lange auf Seite i nicht mehr zugegriffen wurde
- Wenn auf eine Seite zugegriffen wurde ...
  - Setze alle Bits in Zeile i auf 1
  - Setze alle Bits in Spalte i auf 0
- Die Seite mit der kleinsten Zeile wurde am längsten nicht mehr benutzt

# LRU mit Matrix

	Page			
	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

	Page			
	0	1	2	3
0	0	0	1	1
1	1	0	1	1
2	0	0	0	0
3	0	0	0	0

(b)

	Page			
	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	1	1	0	1
3	0	0	0	0

(c)

	Page			
	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0

(d)

	Page			
	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	1
3	1	1	0	0

(e)

	Page			
	0	1	2	3
0	0	0	0	0
1	1	0	1	1
2	1	0	0	1
3	1	0	0	0

(f)

	Page			
	0	1	2	3
0	0	1	1	1
1	0	0	1	1
2	0	0	0	1
3	0	0	0	0

(g)

	Page			
	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	0
3	1	1	1	0

(h)

	Page			
	0	1	2	3
0	0	1	0	0
1	0	0	0	0
2	1	1	0	1
3	1	1	0	0

(i)

	Page			
	0	1	2	3
0	0	1	0	0
1	0	0	0	0
2	1	1	0	0
3	1	1	1	0

(j)

Zugriffsreihenfolge: 0 1 2 3 2 1 0 3 2 3

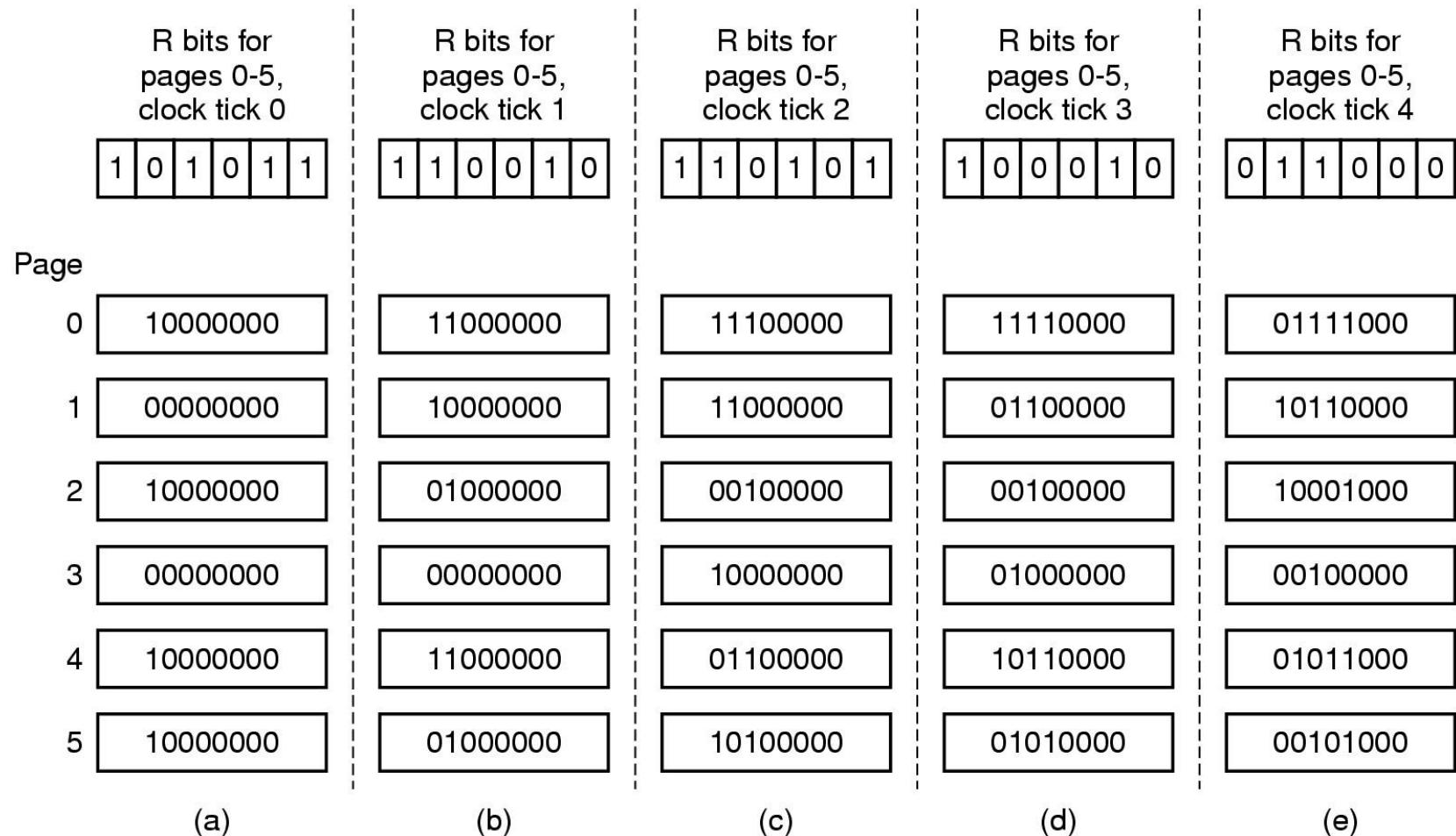
# Not Frequently Used (NFU)

- Not Frequently Used (NFU)
  - Versuch einer LRU-Approximation
- Idee
  - Jede Seite hat einen Zähler
  - Dieser ist anfangs 0
  - Periodisch wird das R Bit auf den Zähler aufaddiert
- Problem
  - Dieser Algorithmus „vergisst“ nie
  - Wenn eine Seite mal sehr gefragt war und dann lange nicht, bleibt der Zähler dennoch hoch

# Aging

- Eine weitere LRU Approximation
  - Ähnlich wie NFU, aber die Addition der R Bits ist geschickter gelöst
- Idee
  - Jede Zeile hat einen Zähler
  - Periodisch wird ...
    - 1) der Zähler nach rechts geschoben  
(d.h. durch 2 dividiert)
    - 2) das R Bit wird in das höchste Bit des Zählers geschrieben
- Vorteile
  - Der Zähler hat ein kürzeres Gedächtnis als bei NFU
  - Das R Bit bestimmt das höchste Bit des Zählers und nicht das niedrigste wie bei NFU

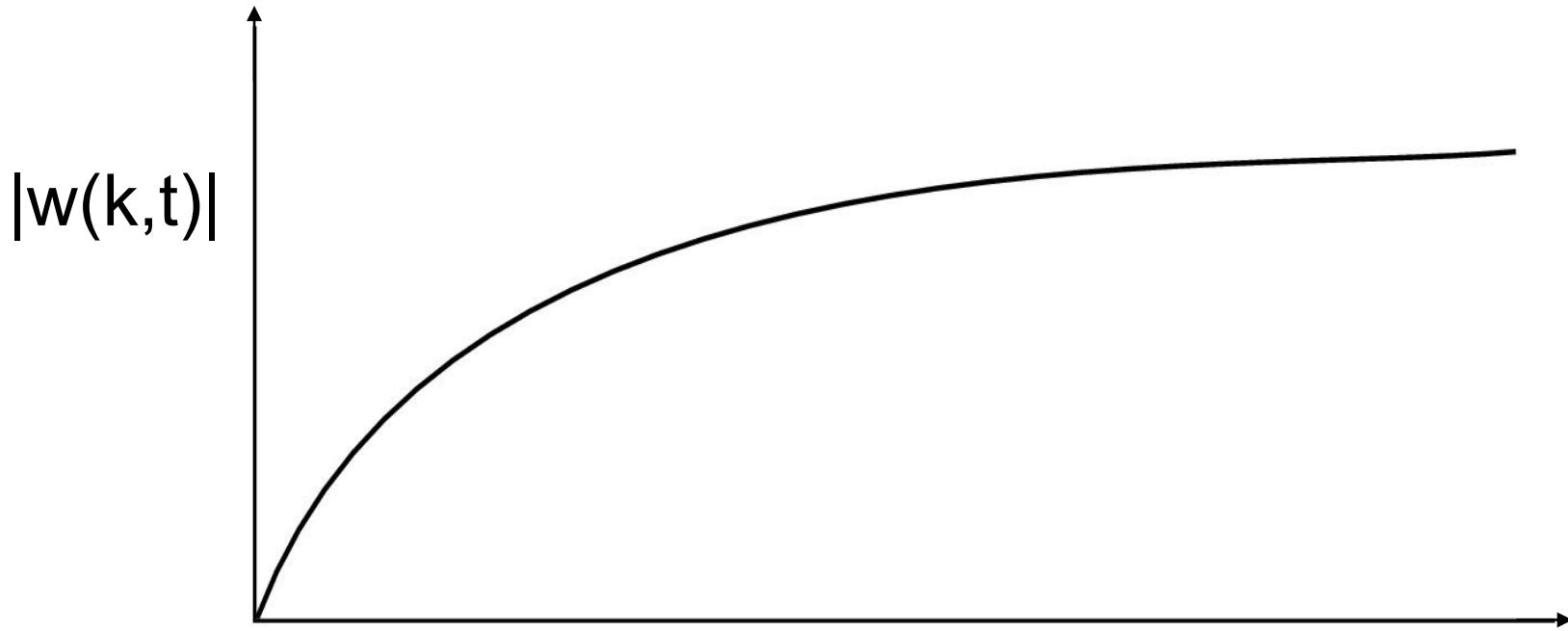
# Beispiel: Aging



# Working-Set

- Beobachtung
  - Speicherzugriffe sind nicht zufällig verteilt
  - In einem Zeitintervall wird auf wenige oft und auf die meisten gar nicht zugegriffen
  - Die oft benutzten heißen Working-Set und sind interessant für Paging und Swapping
- Definition
  - Für Zeitpunkt  $t$  und Integer  $k$  ist  $w(k,t)$  die Menge der Seiten, die in den letzten  $k$  Speicherzugriffen vor dem Zeitpunkt  $t$  benutzt wurden
  - Der Grenzwert für große  $k$  ist endlich, weil jeder Prozess nur endlich viele Seiten hat

# Working-Set



- Es gibt ein  $k$ , so dass  $W(k,t)$  sich für größere  $k$  nicht signifikant ändert
- Man muss nur endlich lange in die Vergangenheit schauen, um ein gutes Working-Set zu bestimmen

# Working-Set und Swapping

- Drei Möglichkeiten zum Einlagern eines Prozesses
  - 1) Alle Seiten eines Prozesses laden und ihn dann ausführen
    - Das dauert lange
  - 2) Keine Seite laden und warten, bis per Seitenfehler alle benötigten Seiten geladen wurden
    - Die Ausführung wird dauernd unterbrochen
    - Die CPU wird lange untätig sein
  - 3) Nur die Seiten des Working-Set werden geladen
    - Das Swapping geht schneller von statten
    - Es sind nur wenige Seitenfehler zu erwarten

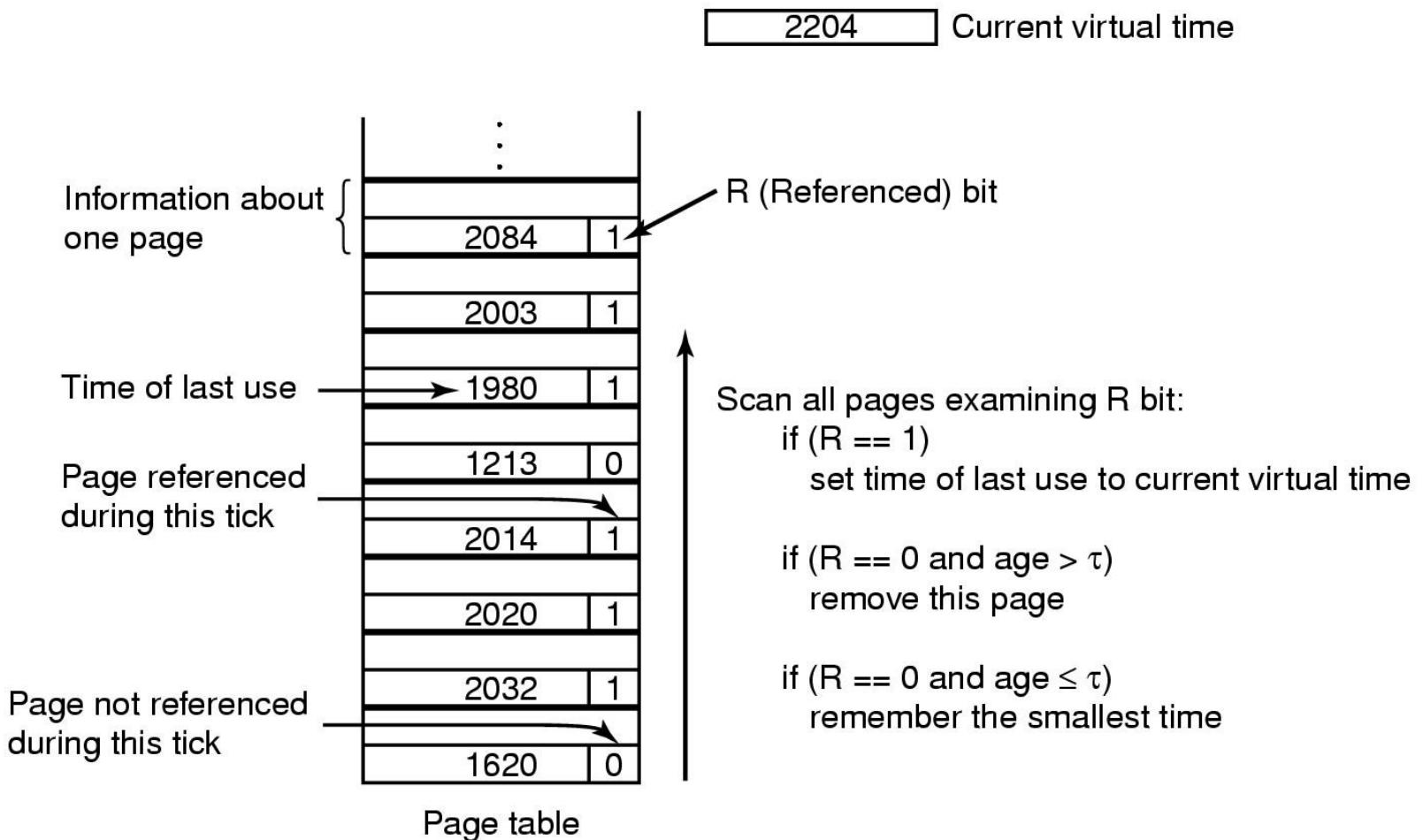
# Working-Set und Paging

- Seitenersetzungs–Algorithmus
  - Entferne eine Seite, die nicht im Working–Set ist
  - Wenn alle im Working–Set sind, dann entferne eine, auf die schon lange nicht mehr zugegriffen wurde
- Ganz langsame Implementierung
  - Schieberegister mit  $k$  Einträgen
  - Wenn auf Seite  $i$  zugegriffen wurde, dann schiebe  $i$  links in das Register
  - Rechts fällt eine „alte“ Seitennummer raus
  - Zur Zeit  $t$  enthält das Schieberegister genau  $w(k,t)$
  - Leider viel zu langsam

# Working-Set und Paging

- Weniger langsame Implementierung
  - R Bits werden periodisch gelöscht
  - Seitentabelle speichert eine „virtuelle Zeit“ pro Seite
  - Wir wählen eine „virtuelle Zeitspanne“  $\tau$ , welche vom Parameter k abgeleitet wird
    - Seiten, die länger als  $\tau$  nicht mehr benutzt wurden, gehören nicht zum Working-Set
  - Wenn eine Seite ausgelagert werden muss ...
    - Wenn  $R=1$ , dann update des Zeitstempels
    - Wenn  $R=0$  und Zeitstempel  $>\tau$ , dann auslagern
    - Wenn das nicht klappt, dann Seite mit dem ältesten Zeitstempel auslagern

# Working-Set und Paging

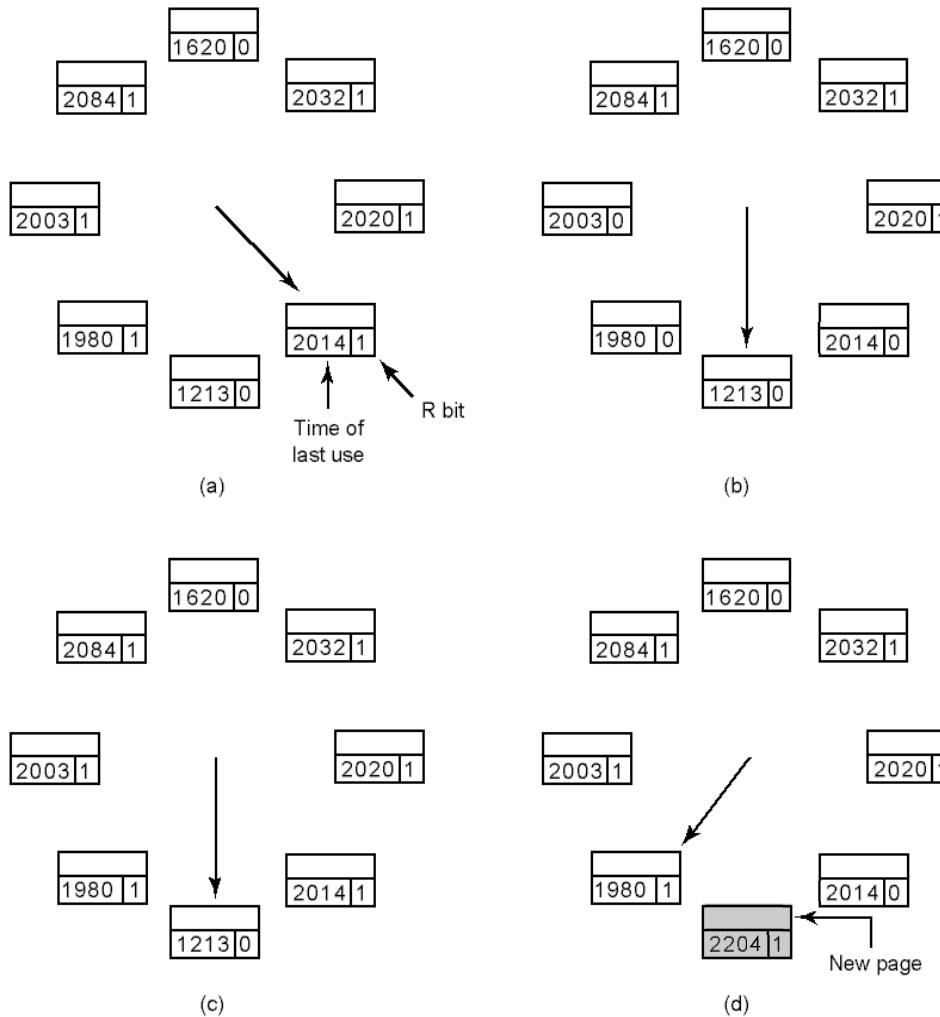


# WS-Clock Algorithmus

- Eine Optimierung des bereits vorgestellten Working-Set Algorithmus
  - Das Durchsuchen der Seitentabelle dauert zu lange
  - Das M-Bit wird nicht berücksichtigt
- Unverändert
  - Zeitstempel und Behandlung des R-Bits
- Verbesserung
  - Verwendung eines Rings (siehe Clock Algorithmus)
    - Zeiger auf ein Element des Rings
  - M-Bit bestimmt, ob eine Seite „sauber“ ist
    - $M=0 \rightarrow$  Seite leer oder unverändert auf der Festplatte
    - $M=1 \rightarrow$  Seite muss erst ausgelagert werden

# WS-Clock Algorithmus

- Suche beginnt beim Ringzeiger
- Wenn  $R=1$ 
  - Setze den Zeitstempel und lösche das R-Bit
- Wenn  $R=0$  und Zeitstempel älter als  $\tau$ 
  - Wenn Seite sauber → überschreiben
  - Ansonsten Seite für auslagern vormerken und weitersuchen
- Nach einem erfolglosen Durchlauf
  - Mindestens eine Seite vorgemerkt?
    - Warten, bis eine auf die Festplatte geschrieben wurde
  - Keine Seite vorgemerkt
    - Alle gehören zum Working-Set
    - Eine saubere Seite des Working-Set auslagern (wenn möglich)
    - Ansonsten irgendeine Seite auslagern



- (a) und (b) zeigen den Fall  $R=1$
- (c) und (d) zeigen den Fall  $R=0$

# Seitenersetzung im Überblick

Optimal	Unrealisierbar, aber gut für Vergleiche
NRU (Not Recently Used)	Sehr primitiv
FIFO (First In First Out)	Auch oft benötigte Seiten können ausgelagert werden
Second Chance	Verbesserung gegenüber FIFO
Clock	Praxistauglich
LRU (Least Recently Used)	Exzellent, aber schwierig zu implementieren
NFU (Not Frequently Used)	Der Algorithmus hat ein zu langes Gedächtnis
Aging	Fast so gut wie LRU
Working Set	Gute Idee, langsame Implementierung
WSClock	Praxistauglich

# Übersicht 1

- Binden und Laden eines Programms
  - Relokation
- Freispeicherverwaltung
  - Interner/Externer Verschnitt
- Swapping
  - Auslagern von Prozessen
  - Nützlich für Multiprogramming
- Virtueller Speicher
  - Virtuelle zusammenhängende Speicherbereiche
  - Vermeidet externen Verschnitt

# Übersicht 2

- Seitenersetzung (Paging)
  - Auslagern einzelner Seiten
  - Auswählen „nicht mehr gebrauchter“ Seiten
- Modellierung von Seitenersetzung
  - Keller-Algorithmen
  - Berechnen der Anzahl Seitenfehler
- Design-Kriterien
  - Feinheiten und kleinere Probleme, die man vor der Implementierung berücksichtigen muss
- Segmentierung
  - Erleichtert die Freispeicherverwaltung eines Prozesses

# Design-Kriterien

- Lokale Paging-Strategie
  - Jeder Prozess hat m Seitenrahmen
    - Wie bestimmt man m geeignet?
  - Bei Seitenfehlern wird ein Seitenrahmen des Prozesses freigemacht, d.h. eine Seite des Prozesses ausgelagert
- Globale Paging-Strategie
  - Alle Prozesse teilen sich m Seitenrahmen
    - Man muss nicht jedem Prozess eine fixe Anzahl zuweisen
  - Bei Seitenfehlern wird ein Seitenrahmen irgendeines Prozesses freigemacht
    - D.h. ein Prozess bekommt Seitenrahmen auf Kosten eines anderen

# Design-Kriterien

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

(c)

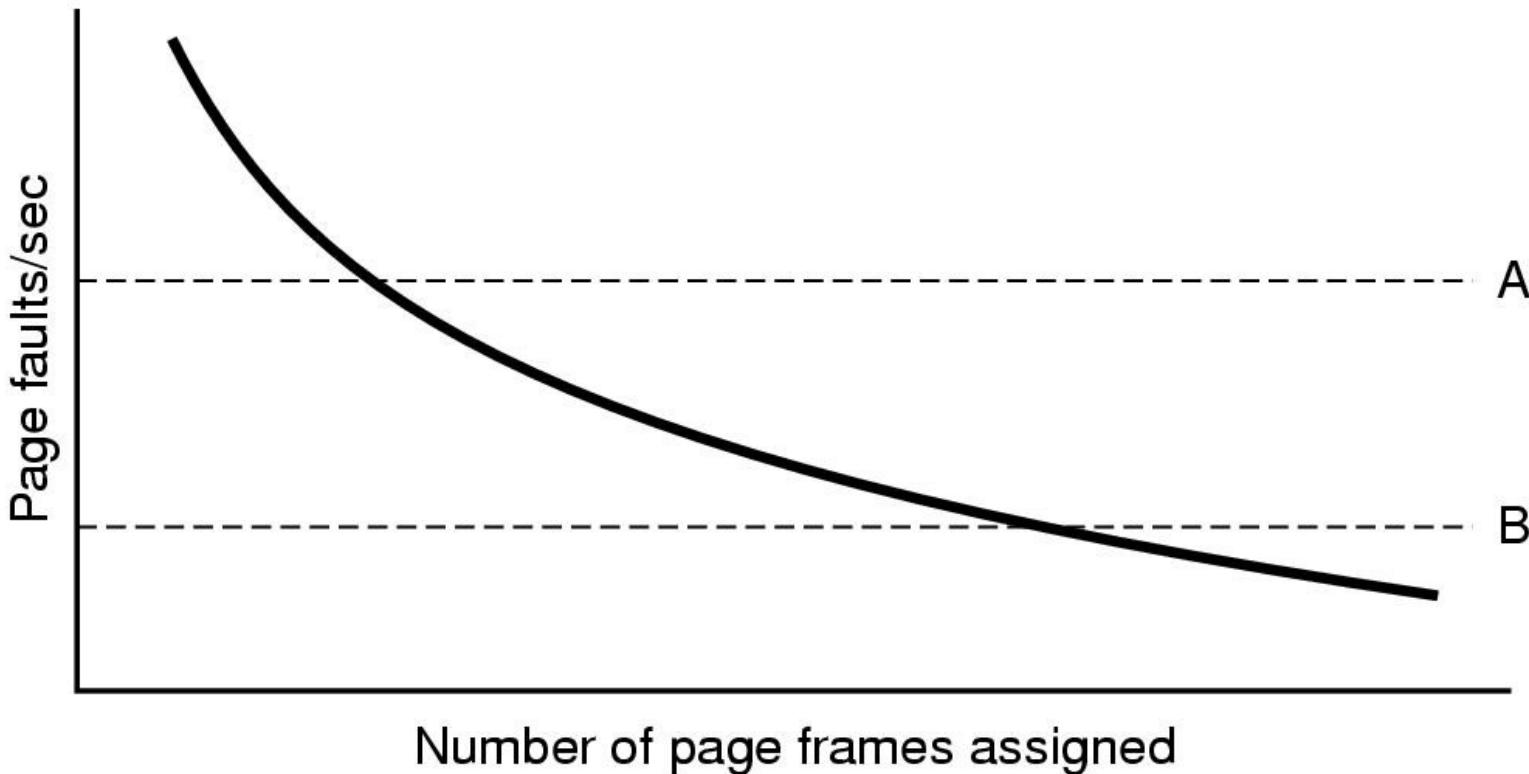
- (a) Ausgangssituation
- (b) Lokale Paging Strategie
- (c) Globale Paging Strategie

# Design-Kriterien

- Globale Paging Strategie
  - Wesentlich flexibler und daher praxistauglich
- Offenes Problem
  - Wie viele Seitenrahmen darf ein Prozess maximal bekommen?
  - Wie verhindert man, dass Prozesse sich in Überlast gegenseitig Seitenrahmen „klauen“?
- Lösung 1: Page Fault Frequency (PFF)
  - Wenn ein Prozess sehr viele Seitenfehler hat und einer wenige, dann muss man umverteilen

# Design-Kriterien

- Page Fault Frequency



# Design-Kriterien

- Lösung 2: Swapping
  - Wenn alle Prozesse sehr viele Seitenfehler produzieren, dann spricht man von Überlast
  - Entspannung durch Swapping
  - Einige Prozesse werden komplett ausgelagert
    - Keine Seitenrahmen
    - Keine CPU-Zeit
  - Die verbleibenden Prozesse können mit wenigen Seitenfehlern weiterlaufen
  - Sinkt die Seitenfehlerrate stark, werden ausgelagerte Prozesse wieder eingelagert
  - Verbleibendes Problem
    - Welchen Prozess auslagern?

# Design-Kriterien

- Seitengröße
  - Zum Teil durch die Hardware vorgegeben
    - Pentium arbeitet mit 4KB
  - Gängige Größen zwischen 512 Byte und 16KB
- Widersprüchliche Ziele
  - Große Seiten → viel interner Verschnitt
  - Kleine Seiten → große Tabellen
- Weitere Kriterien
  - Zeit für Auslagern/Einlagern
  - Auslagern vieler kleiner Seiten ist teurer als das weniger großer Seiten
    - Grund: Kopfbewegung der Festplatten

# Design-Kriterien

- Die „optimale“ Seitengröße
  - Ein Prozess fordert  $s$  Byte an
  - Das resultiert in  $s/p$  Seiten
    - Die brauchen wiederum  $s^*e/p$  Byte in der Seitentabelle
    - $e$  ist die Größe eines Tabelleneintrags
  - Interne Fragmentierung
    - Statistisch gesehen ist die Fragmentierung  $p/2$
  - Overhead insgesamt =  $s^*e/p + p/2$
  - Bestimmung des Minimum durch Ableitung und Null-Setzen:  $0 = -s^*e/p^2 + \frac{1}{2}$
  - Das Minimum liegt bei  $p = \sqrt{2 * s * e}$
  - Weitere Kriterien sind hier nicht berücksichtigt
    - Daher „optimal“ in Anführungsstrichen

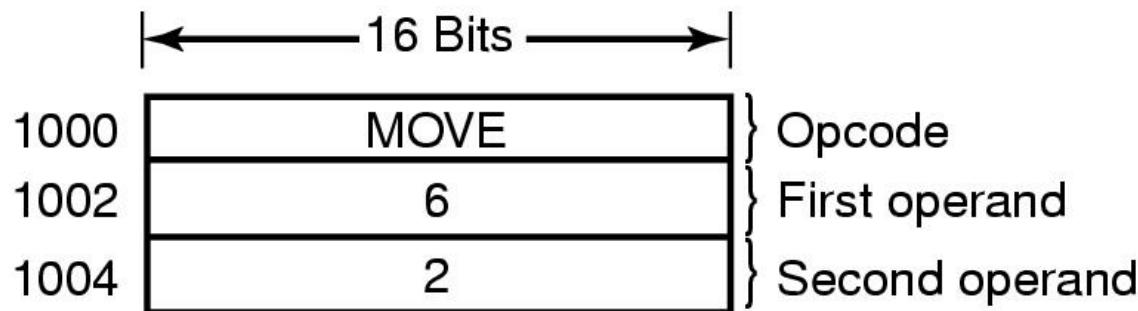
# Design-Kriterien

- Gemeinsam genutzte Pages
  - Nach einem fork (unter UNIX) teilen sich zwei Prozesse dieselben Pages
  - Teure Lösung: Kopieren
  - Effizientere Lösung: Copy on Write
- Copy on Write
  - Seite wird nur für Lesen freigegeben (Permission Bits)
  - Schreiben führt zu Fehler → BS-Handler
  - Das BS kopiert die Seite und setzt das Write-Bit in der Seitentabelle des schreibenden Prozesses
  - So muss man nur das nötige Minimum an Seiten kopieren

# Design-Kriterien

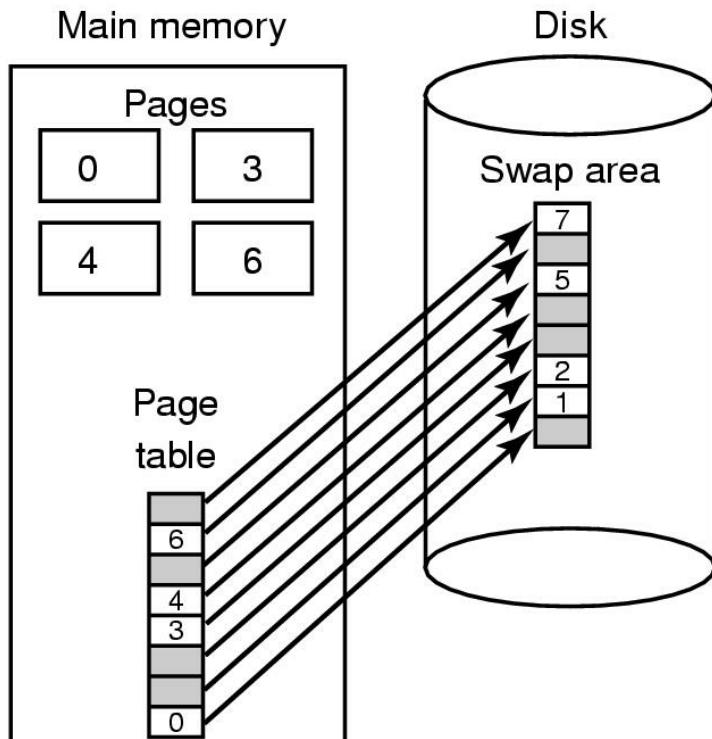
- Wiederholen des unterbrochenen Befehls
  - Der PC zeigt nicht zwangsläufig auf den Anfang des aktuellen Befehls (Opcode)
  - Der PC kann auch auf einen der Operanden zeigen
- Rollback des unterbrochenen Befehls
  - Ein Befehl kann partiell zur Ausführung kommen
  - Beispielsweise wegen eines Auto-Inkремents

MOVE.L #6(A1), 2(A0)

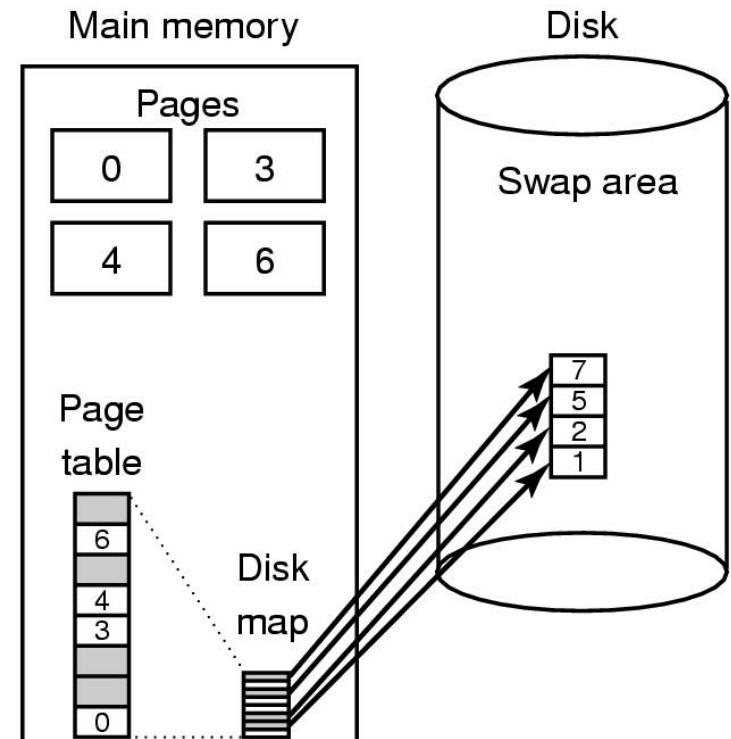


# Design-Kriterien

- Hintergrundspeicher auf der Festplatte



(a)



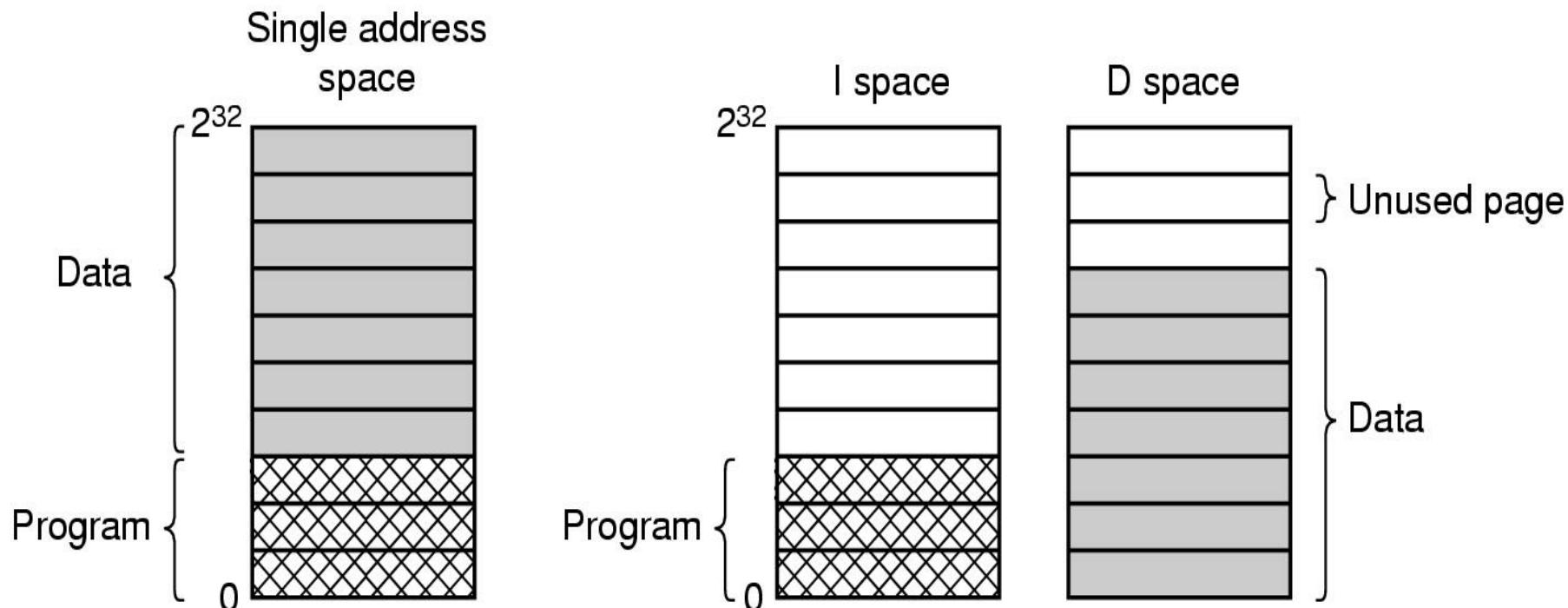
(b)

# Segmentierung

- Virtueller Speicher + Paging
  - Stellt Prozess virtuellen linearen Adressraum zur Verfügung
  - Separiert Prozesse voneinander (da jeder seine eigene Seitentabelle hat)
  - Verwendung von gemeinsamem Programm-Code ist schwierig
    - Er muss in jedem Prozess in denselben Speicherbereich eingeblendet werden
- Segmentierung
  - Stellt jedem Prozesse mehrere lineare Adressräume zur Verfügung

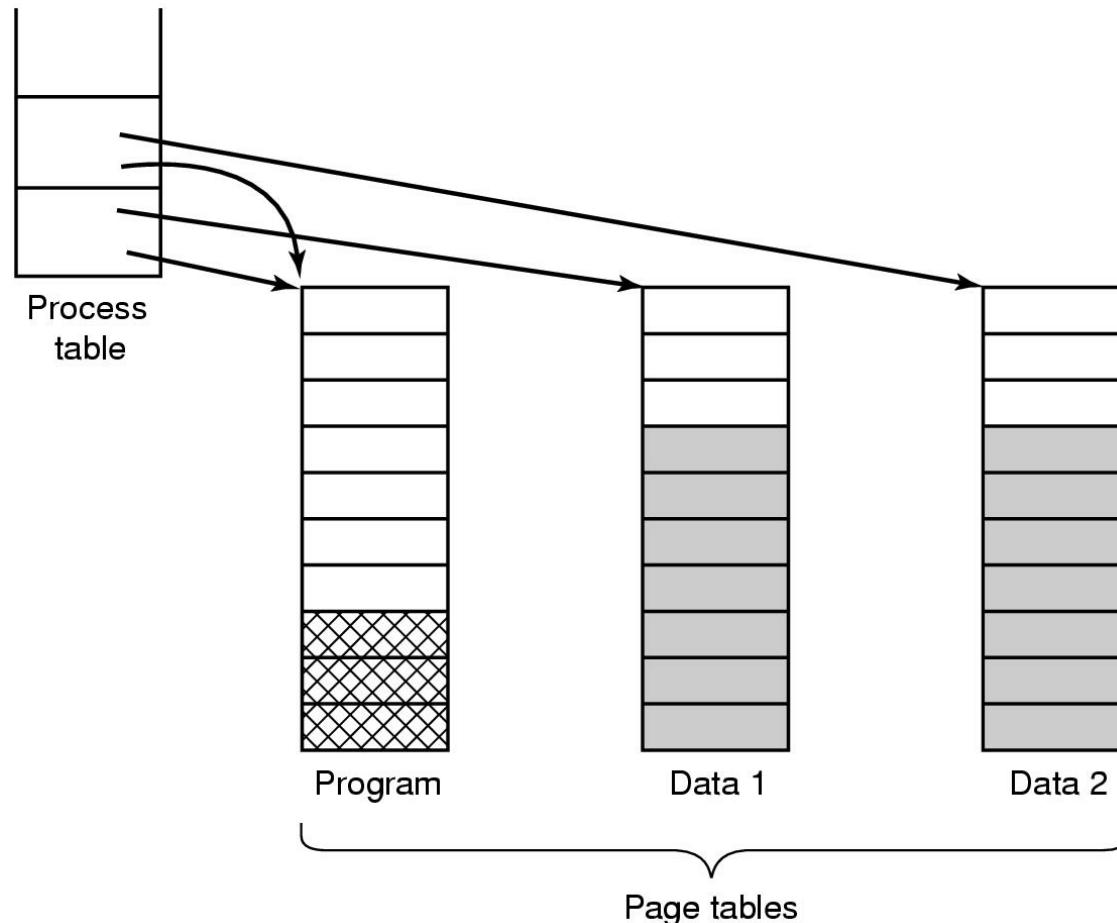
# Segmentierung

- Getrennte Programm- und Datensegmente
  - Spezialfall der Segmentierung



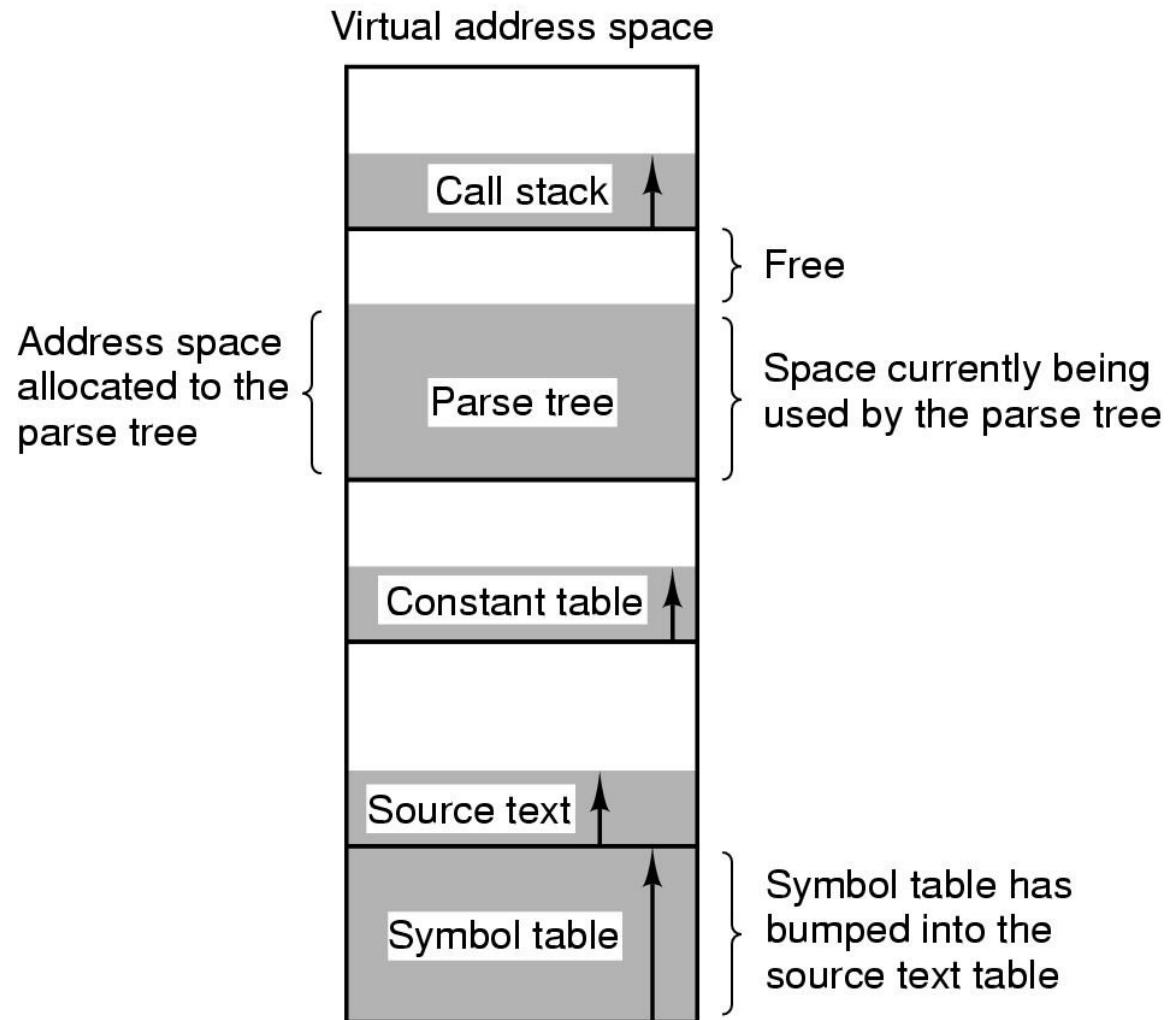
# Segmentierung

- Gemeinsam genutzte Programm-Segmente

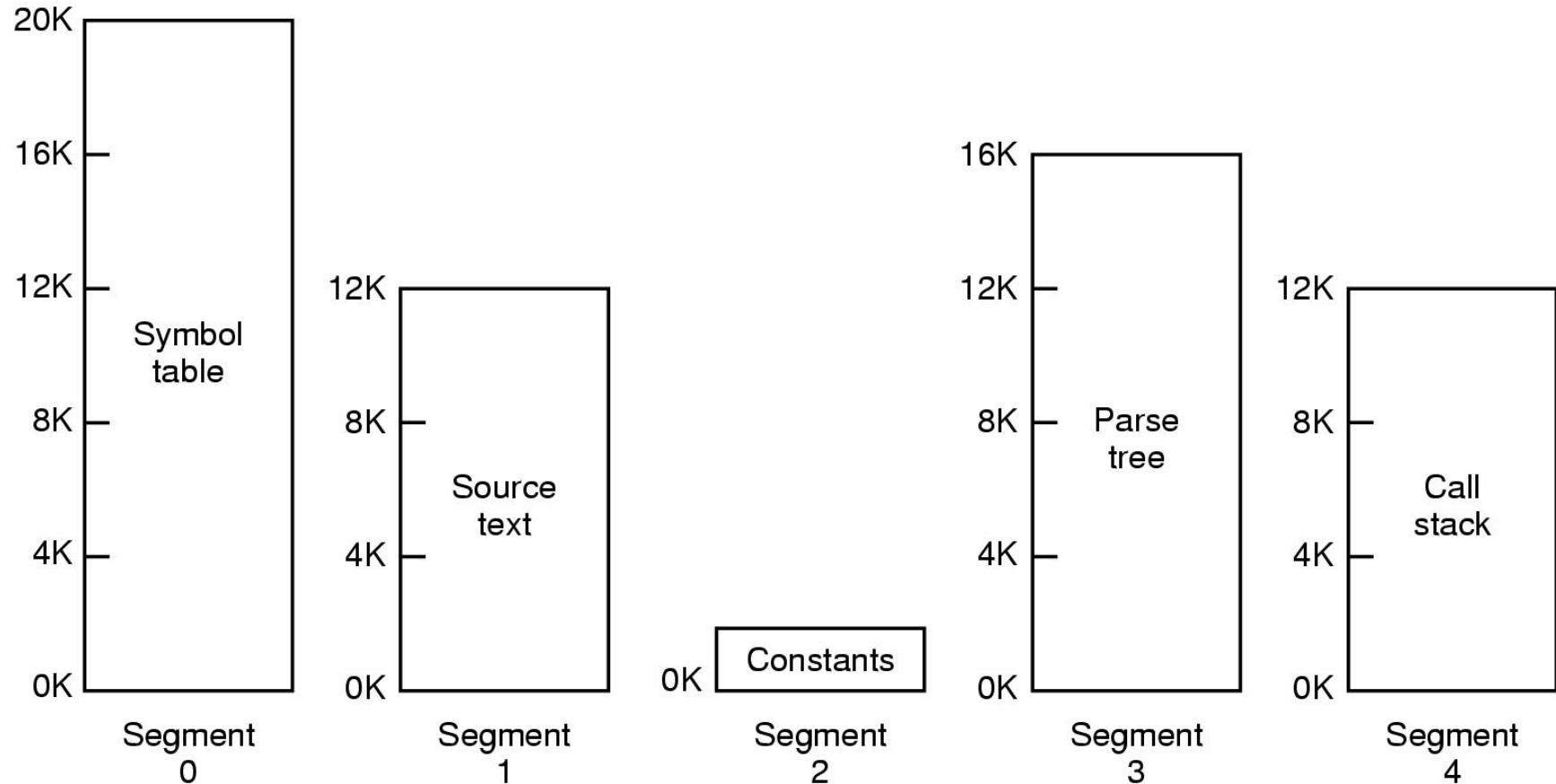


# Segmentierung

- Speicherorganisation mit nur einem linearen Adressraum



# Segmentierung

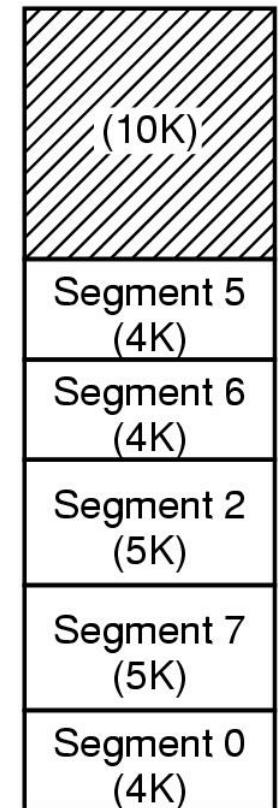
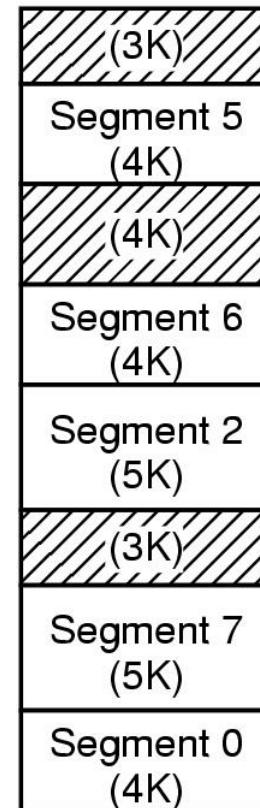
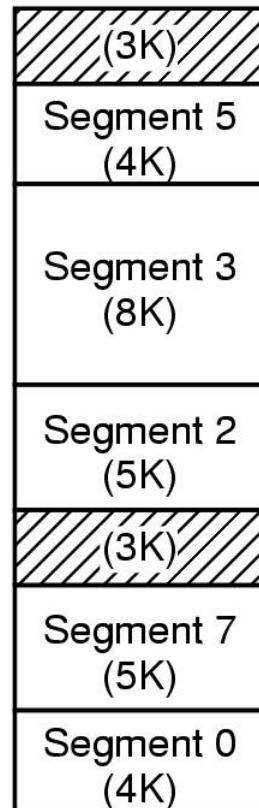
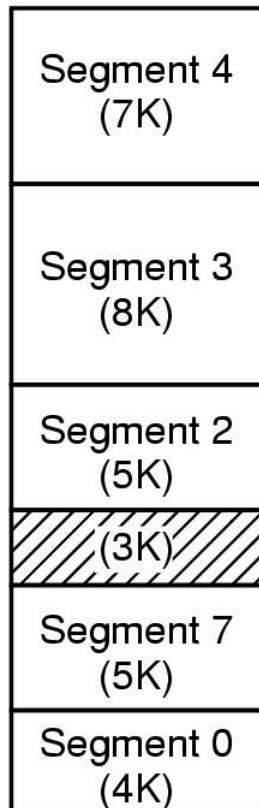
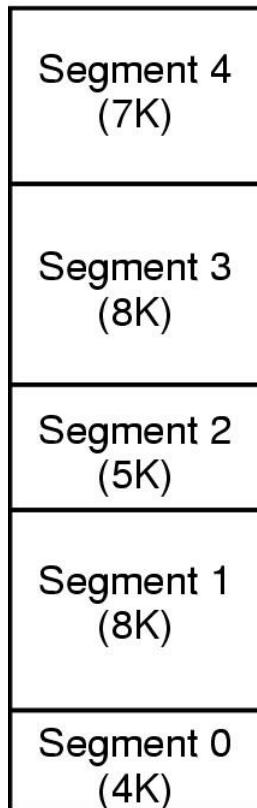


Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	Yes	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

# Segmente, Fragmentierung und Verdichtung

- Paging
  - Keine externe Fragmentierung
  - Gemeinsam genutzte Speicherbereiche sind kompliziert
    - Sie müssen in allen Prozessen an der selben virtuellen Adresse eingeblendet werden
- Segmente
  - Es gibt wieder externe Fragmentierung
  - Segmente können im Speicher verschoben werden
    - Kostet aber leider viel Zeit für das Verschieben
    - Relokation des Codes ist nicht notwendig
    - Nur das Segment-Register benötigt ein Update
  - Gemeinsam genutzte Speicherbereiche sind einfach
    - Jedes Segment hat einen eigenen Adressraum

# Segmente, Fragmentierung und Verdichtung



(a)

(b)

(c)

(d)

(e)

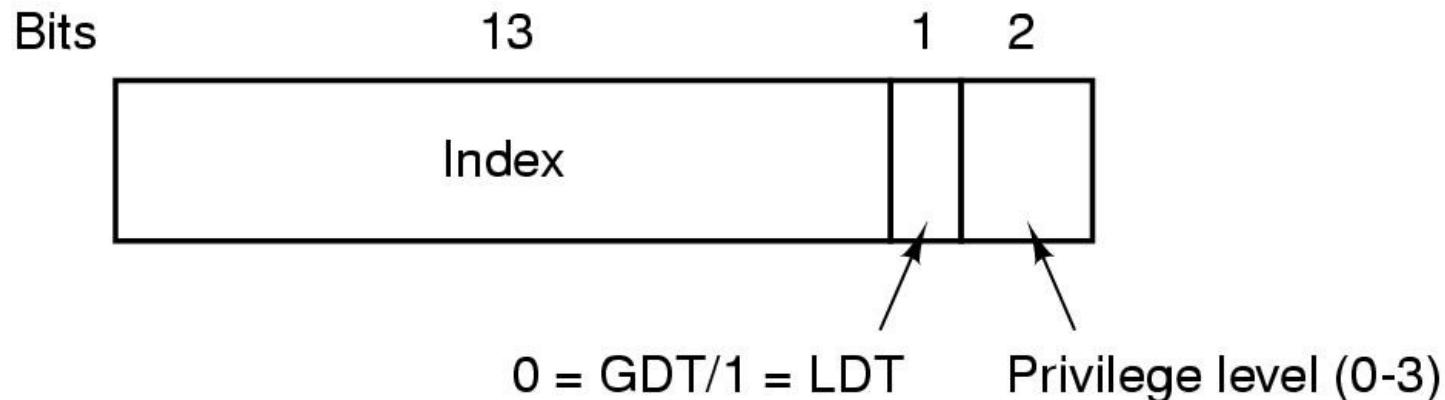
# Segmente & Paging

- Vereint die Vorteile beider Systeme
  - Gemeinsam genutzter Speicher
    - Dank Segmentierung
  - Keine externe Fragmentierung
    - Dank Paging
- Vollständige Adresse
  - Segment + Seite + Offset
- Reihenfolge
  - Zuerst wird der Segment–Deskriptor ausgelesen
  - Die Seitentabelle für das Segment wird gesucht
    - Sie ist eventuell ausgelagert
  - Die physikalische Speicheradresse wird bestimmt
    - Funktioniert wie bei normalem Paging

# Segmente bei Intel Pentium CPUs

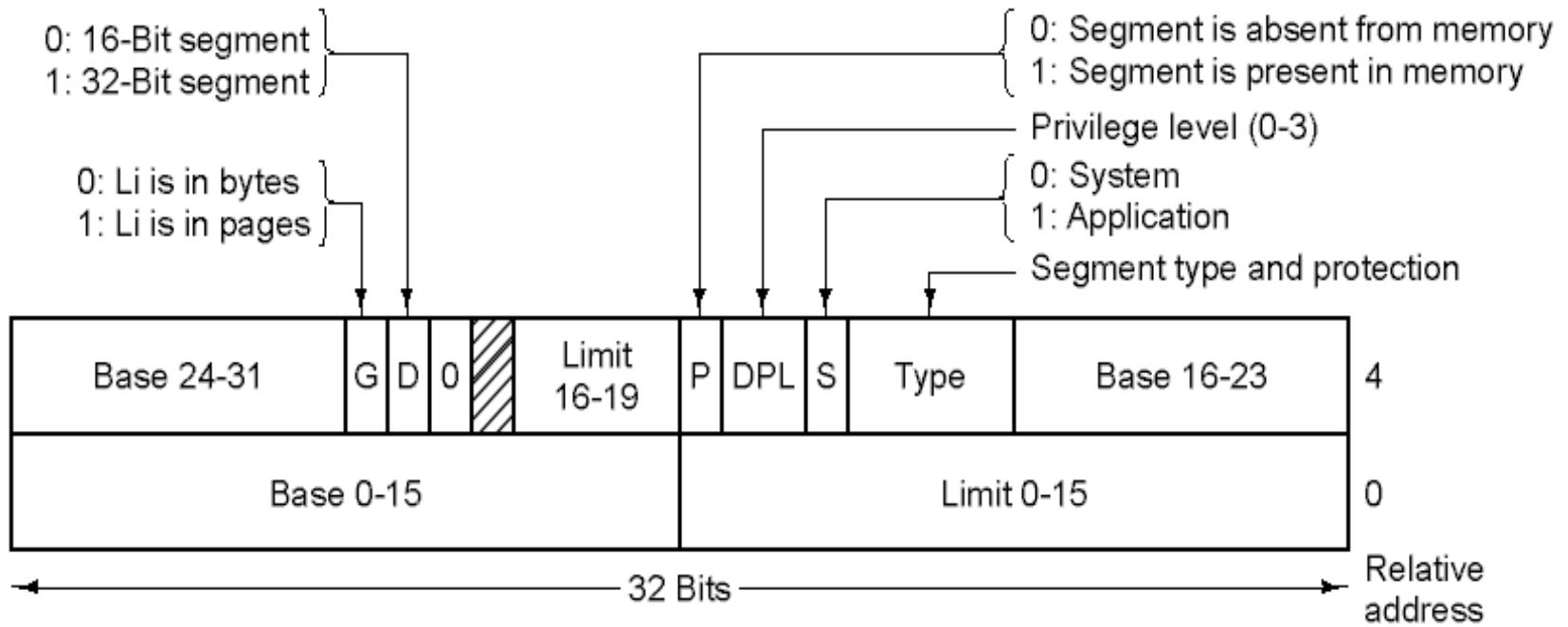
- Intel Pentium CPUs haben 2 Segmenttabellen
- GDT
  - Global Descriptor Table
  - Segmente, die von mehreren Prozessen gemeinsam benutzt werden
    - Bibliotheken
    - Betriebssystem
- LDT
  - Local Descriptor Table
  - Segmente, die nur für einen Prozess relevant sind
    - Jeder Prozess hat eine eigene Tabelle (wenn das OS es so will)

# Segmente bei Intel Pentium CPUs



- Ein Pentium Segment-Selektor
- Index
  - Index in der Segment-Deskriptor-Tabelle
- GDT/LDT
  - Bestimmt die Segment-Deskriptor-Tabelle
- Privilege Level
  - Zugriffsrechte auf Segment Ebene

# Segmente bei Intel Pentium CPUs



- Pentium Segment–Deskriptor

# Auflösen einer Segment-Adresse

