

Folien zur Veranstaltung Betriebssysteme in der TI Sommersemester 2019 (Teil 6)

Prof. Dr. Franz Korf

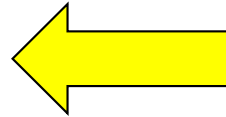
Franz-Josef.Korf@haw-hamburg.de

Basierend auf den BS Vorlesungen von M. Hübner & D. Westhoff & W. Fohl

Kapitel 6: Speicherverwaltung

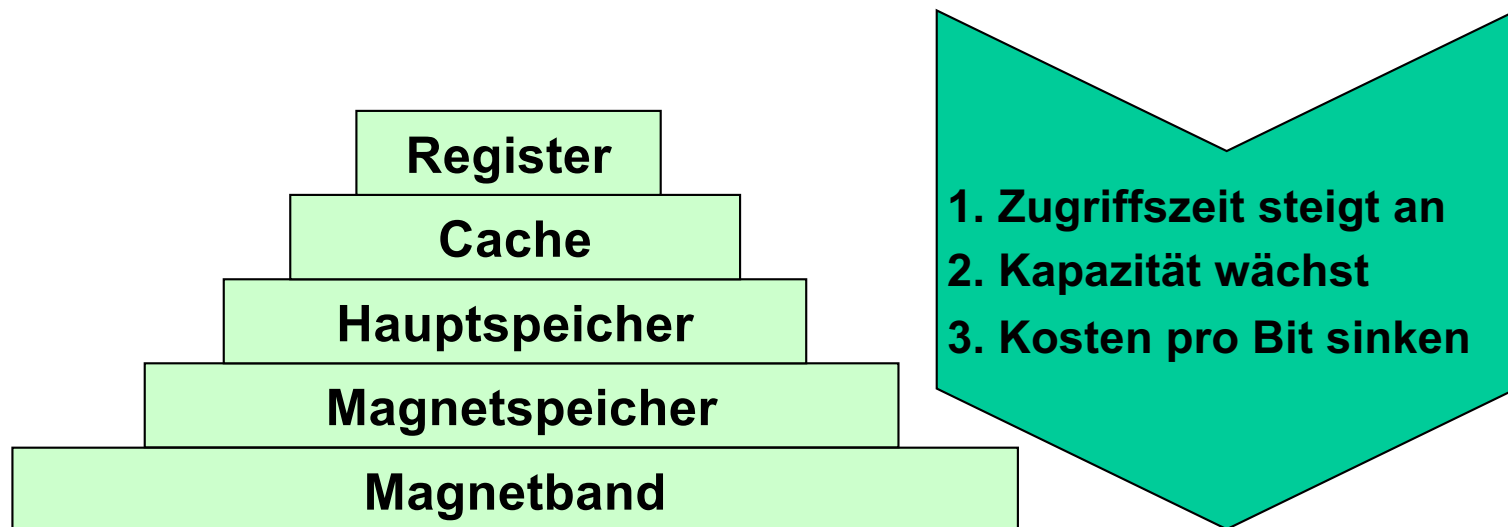
Gliederung

- Einführung und Grundlagen
- Swapping
- Virtuelle Speicherverwaltung
- Seitenersetzungsverfahren
- Entwurfsaspekte
- Unix / Windows



Wiederholung: Speicherhierarchie

- Die Speicherverwaltung des BS organisiert die Vergabe des Hauptspeichers im Rahmen der Speicherhierarchie
- Aufgaben:
 - verfolgt, welche Speicherbereiche gerade benutzt werden
 - teilt Prozessen Speicher zu (und gibt ihn wieder frei)
 - Adressumrechnung
 - Auslagerung von Speicher auf die Festplatte



Anforderungen

- Verschiebbarkeit (Relokation) von Programmen (→ Adressberechnung)
- Kapselung jedes Prozesses (Schutz und Zugriffskontrolle)
- Automatische Zuweisung und Freigabe von Hauptspeicher
- Automatische Auslagerung von Prozessen (Scheduling!)
- Gemeinsame Nutzung

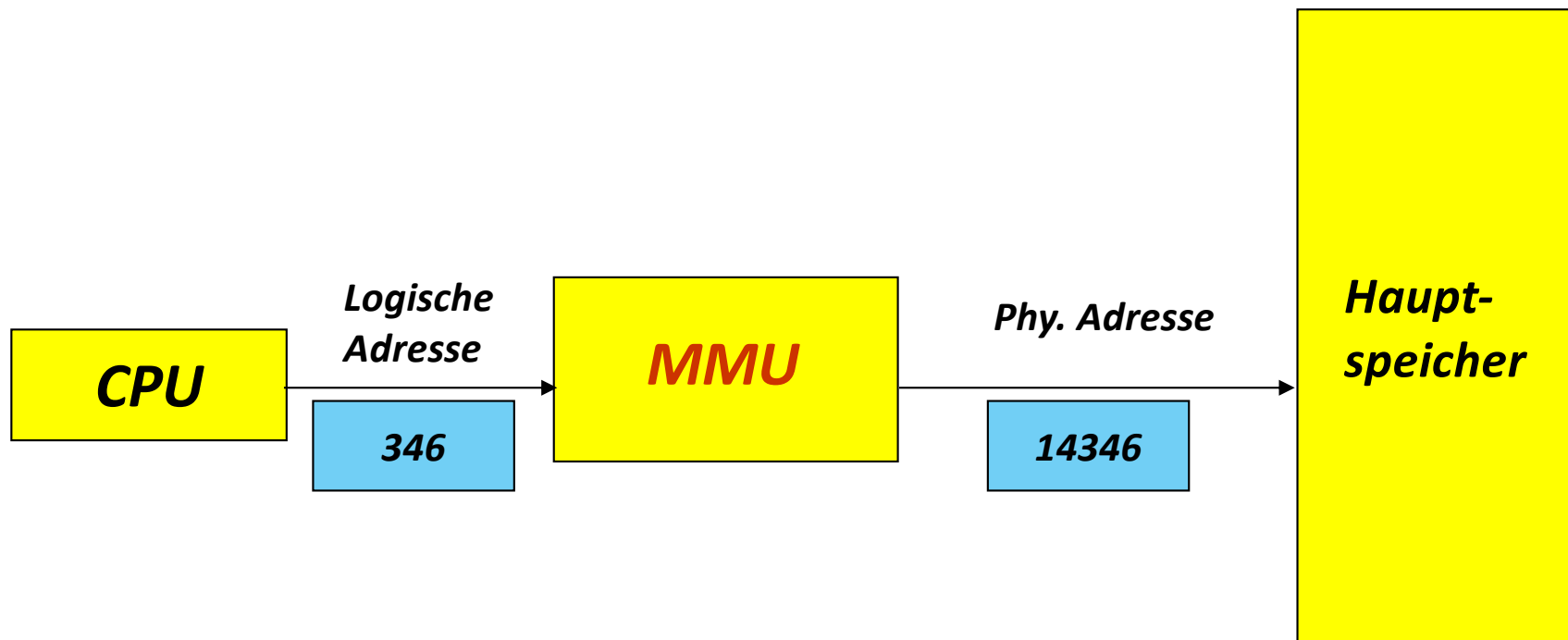
Adressierung

Zwei Arten von Adressen

- **Physikalische Adressen** (reale Adressen)
 - Die absoluten Adressen im physikalischen Hauptspeicher
- **Logische Adressen** (virtuelle Adressen)
 - Logische Adressen werden in den Prozessen verwendet.
 - Referenz auf eine Speicheradresse, ohne dass die reale (absolute) Hauptspeicheradresse bekannt ist.
 - Eine „Übersetzung“ muss vom System (Betriebssystem oder Hardware) vorgenommen werden.
- **Diskussion** des Einsatz von logischen Adressen - Relokation

Dynamische Adressberechnung zur Relokation von Programmen

- **MMU** (Memory Management Unit): Abbildung einer logischen Adresse auf die physikalische Adresse (reale Adresse) durch Hardware (erste Näherung)
- **Beispiel:**



Laden eines Programms in den Hauptspeicher

- Compiler/Assembler verknüpfen üblicherweise einzelne Programmelemente mit logischen Adressen. Bezüge zwischen Modulen werden vorerst offen gelassen.
- Linker / Binder fügen die einzelnen Module zusammen, indem sie
 - die einzelnen Teil-Adressräume gegeneinander verschieben, so dass sie sich nicht überdecken,
 - Querbezüge zwischen den Modulen auflösen.
- Es entsteht ein einheitlicher Adressraum, dessen Adressen aber noch nicht den physikalischen Adressen entsprechen müssen.
- Die Adressen dieses Adressraumes werden entweder statisch durch den Linker/Loader oder dynamisch während der Ausführung des Programms auf die physikalischen Adressen abgebildet.
- Hierdurch wird eine Unabhängigkeit der Programme von ihrer Platzierung im Hauptspeicher erreicht (**Relokation**).

Einfache Adressumsetzungsmethode (Relokation)

- **Aufgabe:** Abbildung der logischen Adressen eines Prozesses auf physikalische Adressen
- **Basis-Register** („Relocation“-Register): Enthält die Startadresse (phy. Adresse) des Prozesses

- **Berechnungsverfahren:**

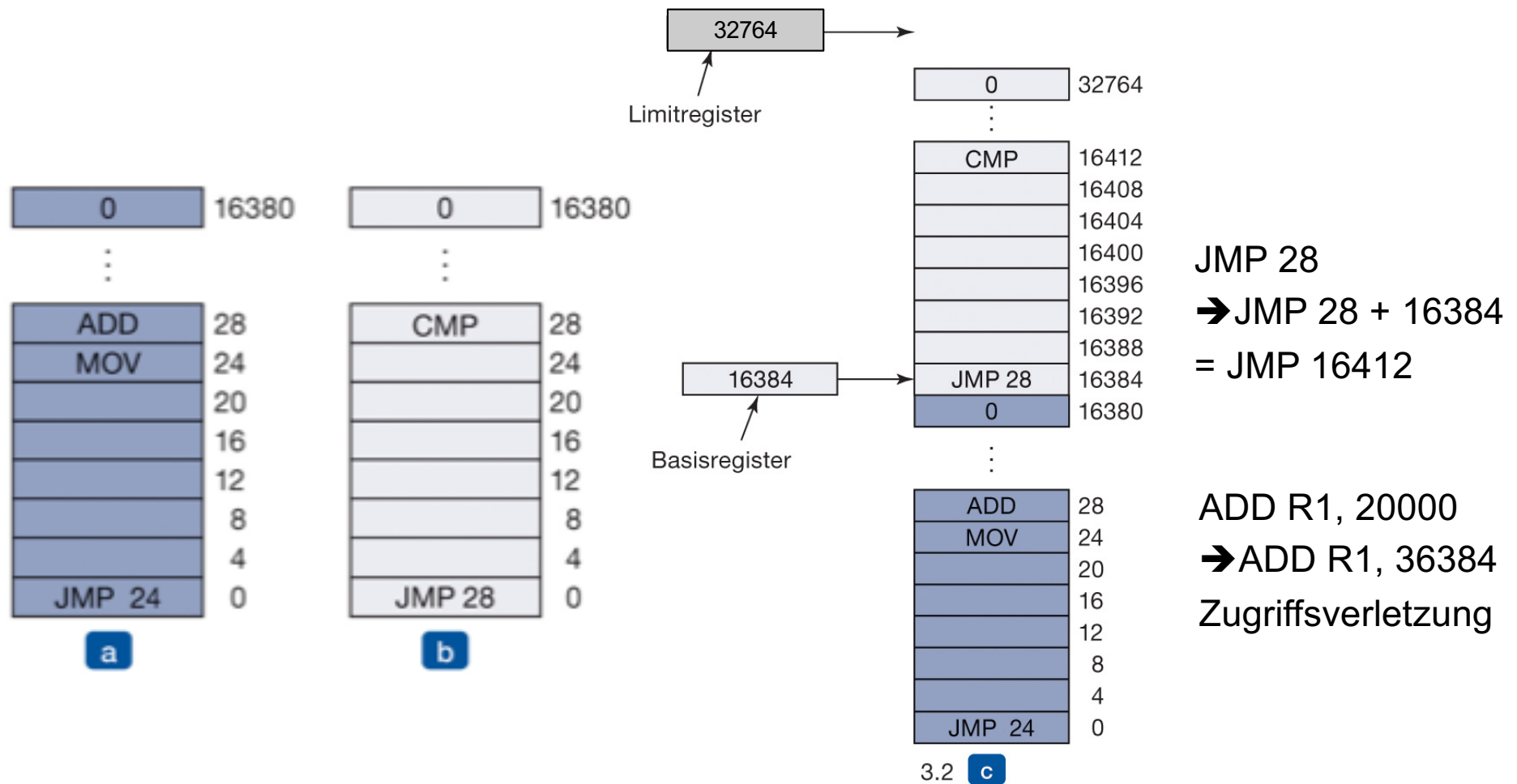
Physikalische Adresse = Logische Adresse + Wert des Basis-Registers

- **Limit-Register:** Endadresse (phy. Adresse) des Prozesses

Wenn Phy. Adresse > Limit-Register → Fehler! (Schutzverletzung)

- Diese Register werden gesetzt, wenn der Prozess geladen oder verschoben wird!
- Umrechnung bei jedem Speicherzugriff (Sprungadressen, holen von Befehlen, ...)

Beispiel

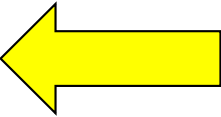


[AT]

Abbildung 3.3: Basis- und Limitregister können benutzt werden, um jedem Prozess einen separaten Adressraum zu geben.

Kapitel 6: Speicherverwaltung

Gliederung

- Einführung und Grundlagen
- Swapping 
- Virtuelle Speicherverwaltung
- Seitenersetzungsverfahren
- Entwurfsaspekte
- Unix / Windows

Swapping – Virtueller Speicher

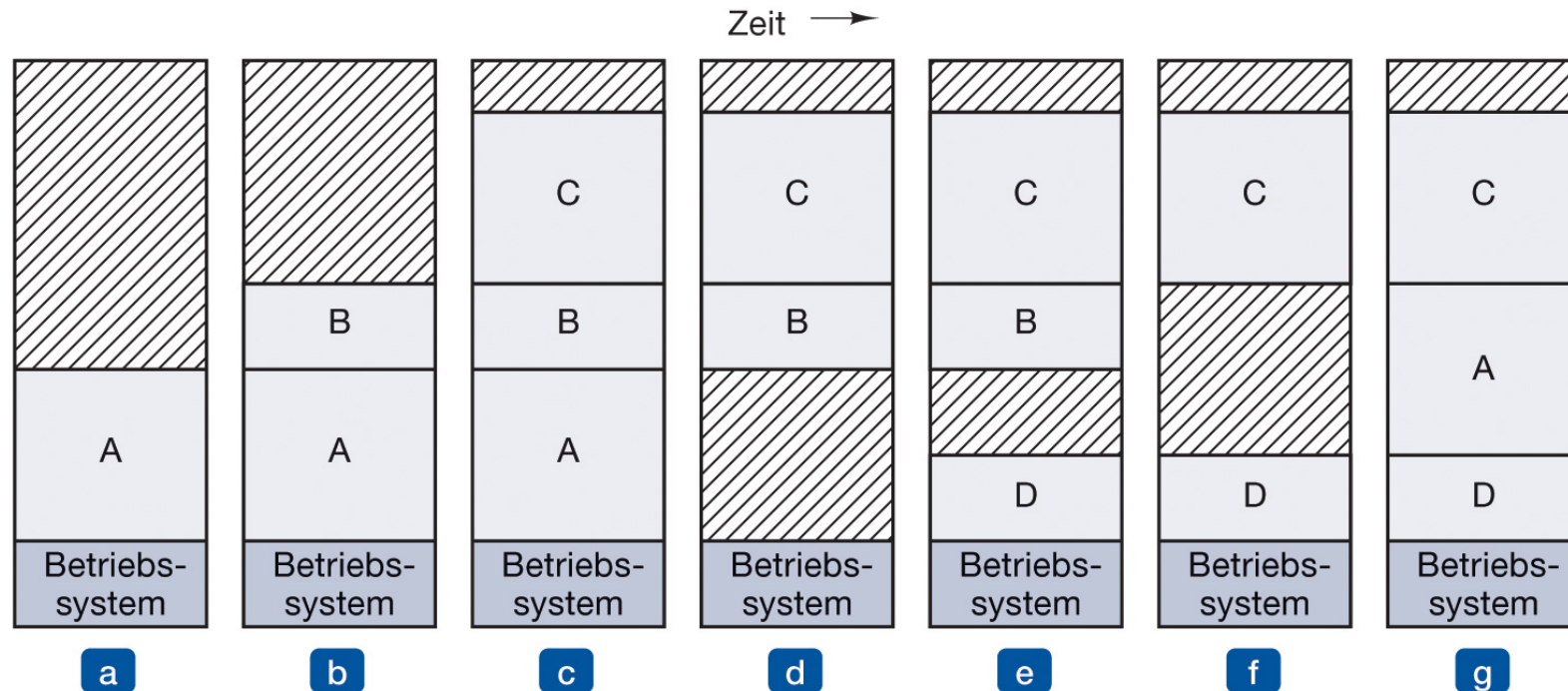
Motivation

- Ohne Speicherverwaltung: Der Hauptspeicher deckt den Speicherbedarf aller (quasi-)parallel laufenden Prozesse ab.
- Aber: Oftmals nicht genug Hauptspeicher für alle aktiven Prozesse → einige müssen auf Festplatte ausgelagert und bei Bedarf dynamisch in den Hauptspeicher geladen werden.

Zwei Ansätze

- **Swapping:** Jeder Prozess wird **komplett** in einen **zusammenhängenden** Speicherbereich geladen. Wenn ein neuer Prozess geladen werden muss und nicht genug Hauptspeicher (**am Stück**) zur Verfügung steht, dann werden ein oder mehr Prozesse komplett vom Hauptspeicher auf Platte ausgelagert. Wenn ein ausgelagerter Prozess wieder zur Verarbeitung ansteht, dann wird er wieder komplett geladen.
- **Virtueller Speicher** - Prozesse laufen auch dann, wenn sich nur **ein Teil** von ihnen im Hauptspeicher befindet. Dieser Teil muss **nicht** **zuhängenden** sein und kann über **nicht zusammenhängende** Hauptspeicherbereiche verteilt sein.

Beispiel Swapping



[AT]

Abbildung 3.4: Mit der Ein- und Auslagerung von Prozessen ändert sich die Speicherbelegung. Die schraffierten Bereiche sind ungenutzt.

- + Einfache Relokation über Basis- und Limit Register
- Fragmentierung
- Für alle Prozesse Hauptspeicher \geq Speicherbedarf des Prozesses:
Aufwendige und fehleranfällige Overlay Technik
- Working Set nicht genutzt

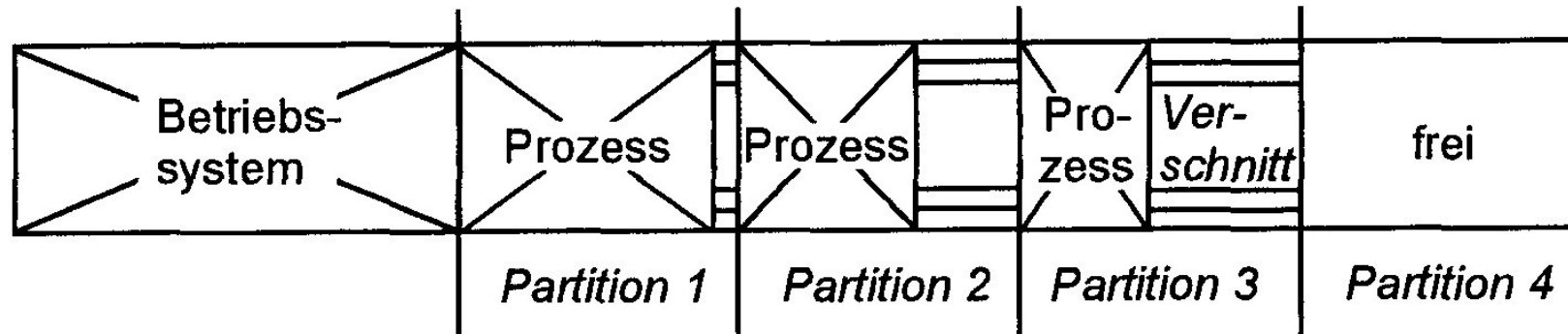
Hauptspeicheraufteilung bei Multiprogramming: Feste Partitionierung

Aufteilung in feste Anzahl Partitionen

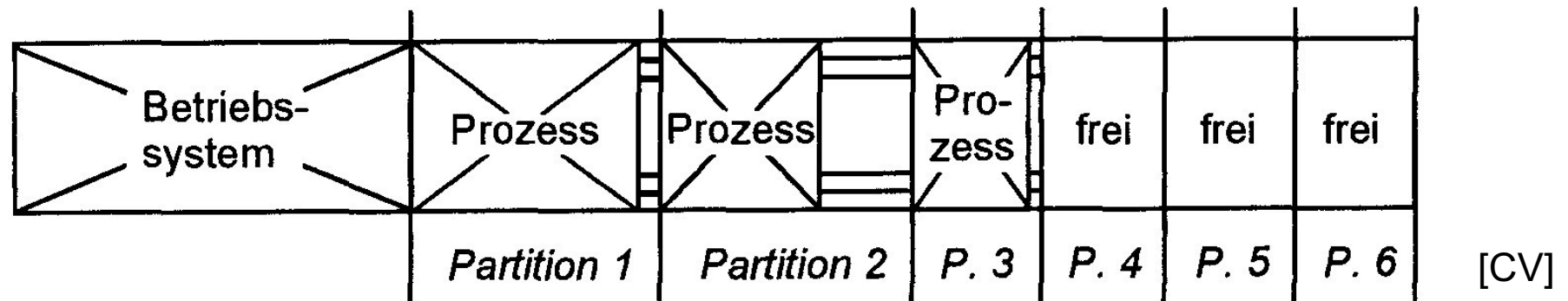
- Jeder Prozess, dessen Platzbedarf nicht über der Größe einer freien Partition ist, kann geladen werden
- Wenn alle Partitionen voll sind, kann das Betriebssystem einzelne Prozesse leicht aus-/ einlagern
- Varianten bzgl. der Partitionsgröße:
 - Alle Partitionen haben eine einheitliche Größe
 - Es gibt unterschiedliche Partitionsgrößen (→ Verringerung des „Verschnitts“)
- **Nachteil:** Ein Programm kann zu groß sein für die Partition: Der Programmierer muss dann sein Programm aufteilen: „Overlay“-Technik
- **Nachteil:** Der Hauptspeicher wird nicht effizient genutzt, jedes Programm belegt eine komplette Partition → „Verschnitt“ → ungenutzter freier Speicher („Interne Fragmentierung“)

Beispiel

b.1.) feste Hauptspeicher-Partitionen einheitlicher Größe



b.2.) feste Hauptspeicher-Partitionen unterschiedlicher Größen



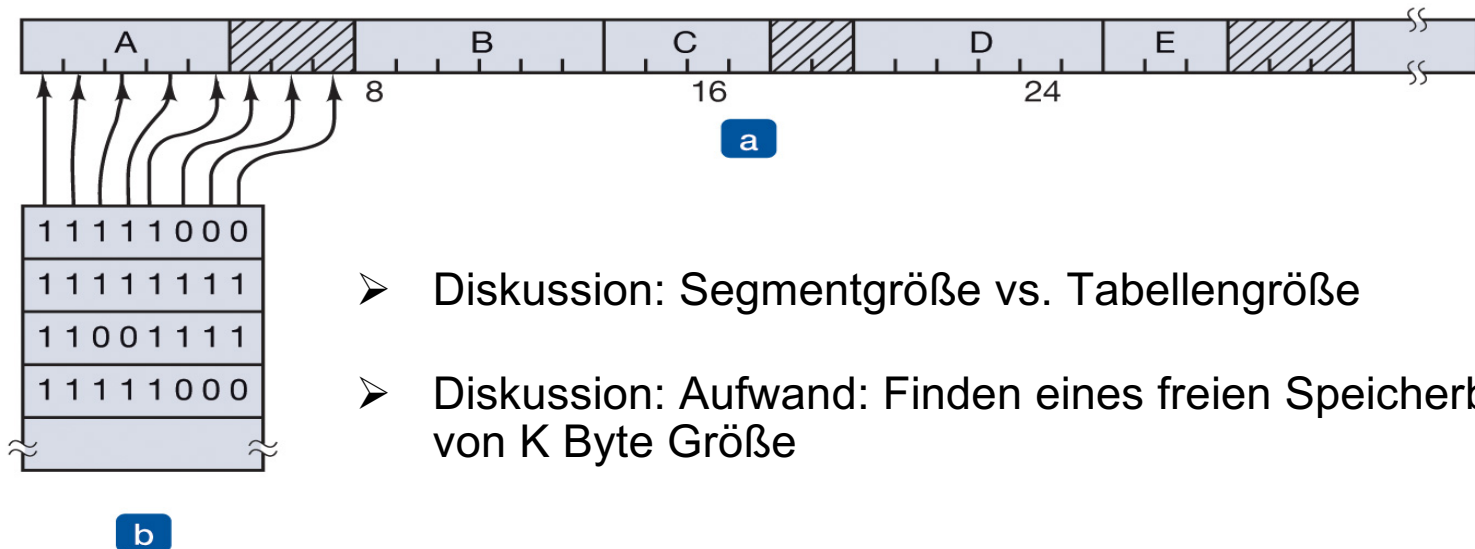
Hauptspeicheraufteilung bei Multiprogramming: Dynamische Partitionierung

- Variable Anzahl von Partitionen unterschiedlicher Größe
- Die Partitionen werden an die Prozessgröße angepasst
- Nach Zuweisung einer Partition zu einem Prozess wird der restliche freie Platz eine neue Partition
- Zusammenfassen von freien Partitionen ist möglich
- **Nachteil:** Der Hauptspeicher wird nicht effizient genutzt: Es entstehen „Löcher“ im Speicher durch kleine Partitionen („externe Fragmentierung“)
Abhilfe: Memory compaction (Speicherverdichtung): Das Betriebssystem könnte die Partitionen umkopieren (ist aber sehr zeitaufwändig)
- **Nachteil:** Ein Programm kann zu groß sein für die Partition: Der Programmierer muss dann sein Programm aufteilen: „Overlay“-Technik

Verwaltung des freien Speichers

Bitmaps

- Teile den Speicher in Belegungseinheiten (Segmentgröße) fester Größe (einige Kilobyte)
- Standardverwaltungstechnik: Wird nicht nur für dyn. Partitionen eingesetzt.
- Eine Tabelle speichert für jedes Segment in einem Bit, ob das Segment belegt (Bit = 1) oder frei (Bit = 0) ist.



[AT]

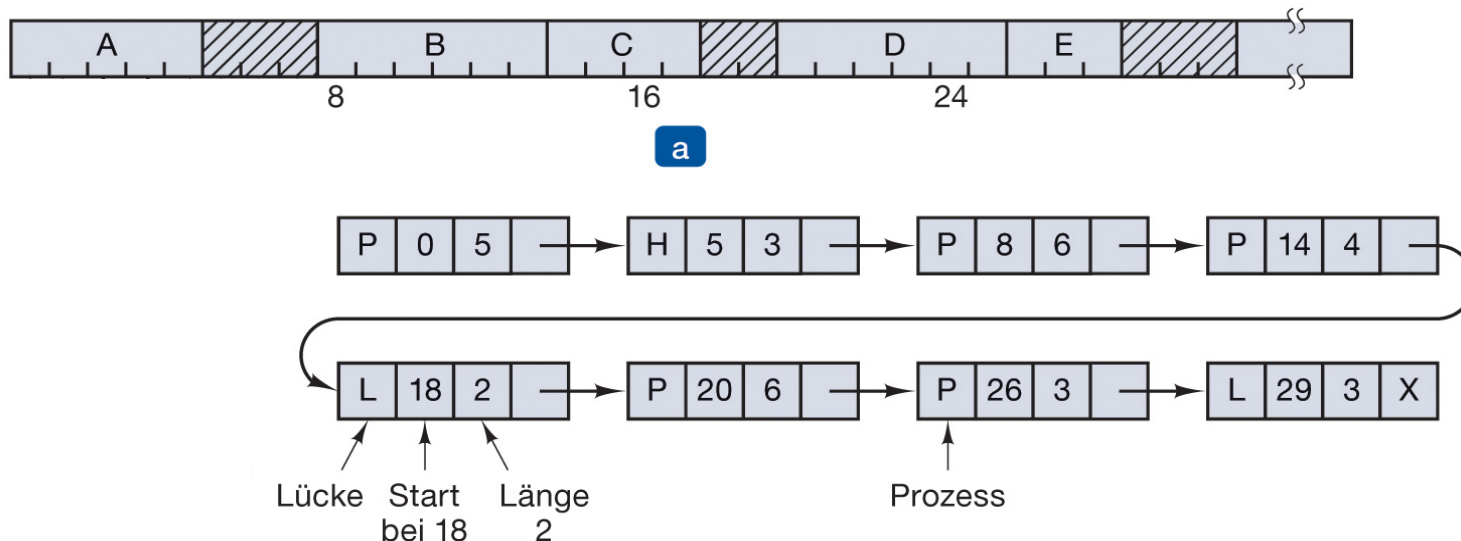
- Diskussion: Segmentgröße vs. Tabellengröße
- Diskussion: Aufwand: Finden eines freien Speicherbereichs von K Byte Größe

Abbildung 3.6: (a) Ein Teil eines Speichers mit fünf Prozessen und drei Lücken. Die Teilstriche markieren die Grenzen der Belegungseinheiten. Die schraffierten Bereiche sind frei (0 in der Bitmap). (b) Die zugehörige Bitmap (c) Dieselbe Information als Liste

Verwaltung des freien Speichers

Verkettete Liste

- Die Liste enthält folgende Informationen: Prozess / frei ; Startadresse des Segments, Länge des Segments
- Standardverwaltungstechnik: Wird nicht nur für dyn. Partitionen eingesetzt.
- Sortierte Liste nach Speicheradressen
- Alternative: Mehrere Listen für unterschiedliche Größen von freiem Speicher



- Diskussion: Aufwand: Finden eines freien Speicherbereichs von K Byte Größe

Dynamische Partitionierung: Platzierungsstrategien

Aufgabe: Das Betriebssystem muss entscheiden, welche freie Partition welchem Prozess zugewiesen wird

- Algorithmus **First-Fit**: Sucht von vorne die nächste freie Partition, die passt
- Algorithmus **Next-Fit**: Sucht ab der zuletzt belegten Partition die nächste freie Partition, die passt
- Algorithmus **Best-Fit**: Auswahl der freien Partition, bei der am wenigsten Platz verschwendet wird
- Algorithmus **Quick Fit**: Getrennte Listen für Löcher gebräuchlicher Größe

Bemerkung: Analyse der Algorithmen auf Basis von charakteristischen Szenarien ist entscheidend.

Diskussion Platzierungsstrategien

Best-Fit:

- Schlechtestes Ergebnis!
- Aufwendigste Suche
- Weil immer kleine Speicherreste bleiben, muss das Betriebssystem häufig umsortieren

**Ergebnisse hängen vom
Anwendungsszenario ab**

First-Fit:

- Schnellstes Verfahren!
- Viele Prozesse im vorderen Speicherbereich, meist hinten noch Platz für große Prozesse

Next-Fit:

- Belegt Speicher gleichmäßiger als First-Fit, nachteilig für große Prozesse

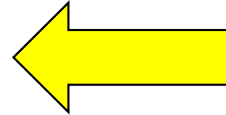
Quick-Fit:

- Sehr schnell

Kapitel 6: Speicherverwaltung

Gliederung

- Einführung und Grundlagen
- Swapping
- Virtuelle Speicherverwaltung
- Seitenersetzungsverfahren
- Entwurfsaspekte
- Unix / Windows



Motivation

Probleme Swapping

- Keine Nutzung des Working Sets
- Fragmentierung
- Speicherbedarf Prozess > Hauptspeicher: aufwendige Overlay Technik
- Heute: Virtueller Speichers: BS lädt automatisch Working Set nach

Anforderungen an einen optimalen Speicher

- Unbeschränkte Größe, so dass jedes beliebig große Programm ohne zusätzlichen Aufwand geladen und verarbeitet werden kann
- Einheitliches Adressierungsschema für alle Speicherzugriffe (keine Unterscheidung von Speichermedien)
- Direkter Zugriff auf den Speicher (ohne Zwischentransporte)
- Schutz vor fremden Zugriffen in den eigenen Speicherbereich

Lösung: Virtuelle Speicherverwaltung wurde 1961 entwickelt

Virtuelle Speicherverwaltung erreicht diese
Ziele sehr gut unter Ausnutzung der
Memory Hierarchie und des Working Sets

Paging

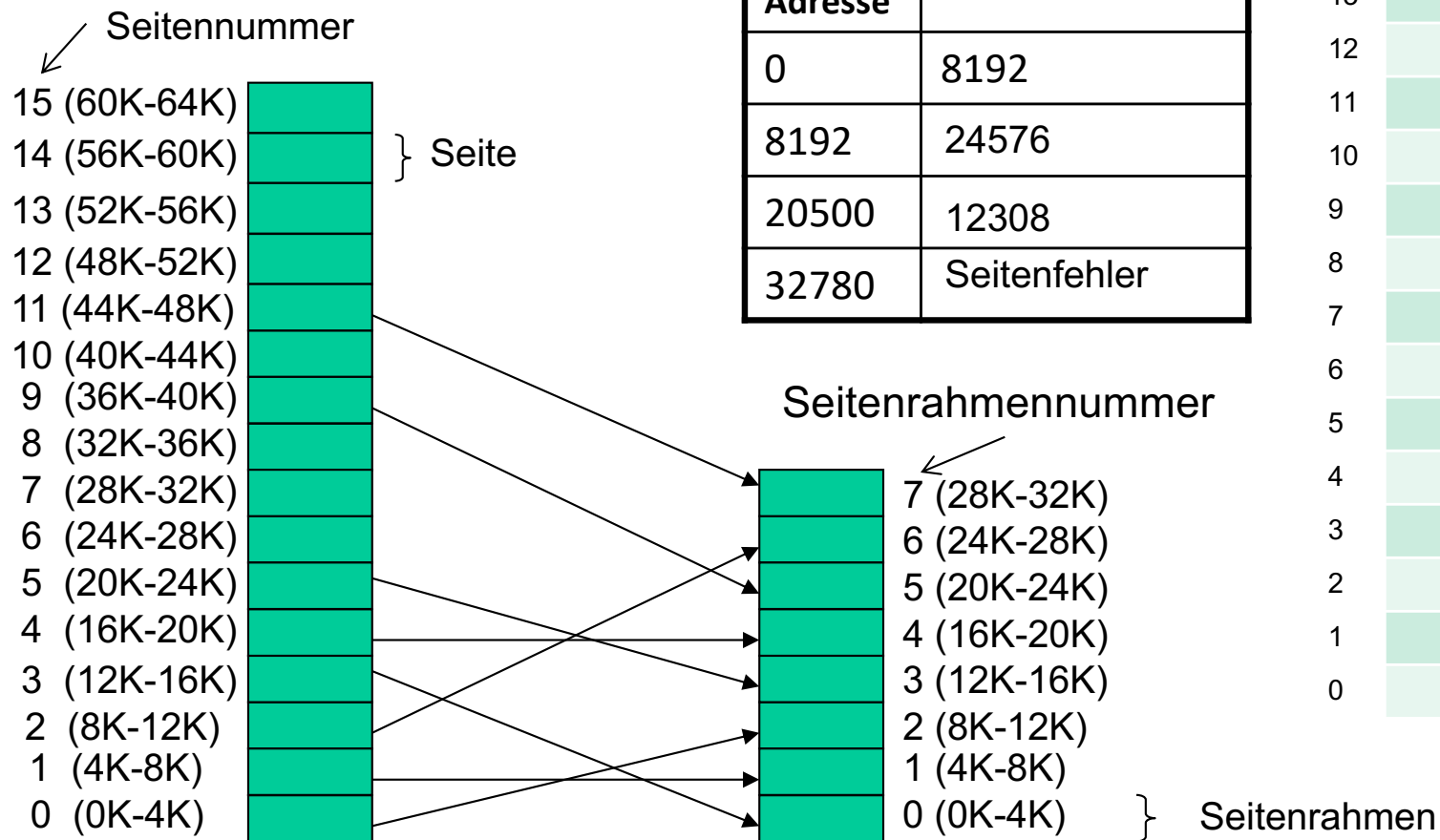
- Ein Prozess hat einen eigenen Adressraum – **virtueller Adressraum**
- Eine vom Programm generierte Adresse ist eine **virtuelle Adresse** aus dem virtuellen Adressraum
- Der virtuelle Adressraum ist in **Seiten (pages)** eingeteilt
- Die Seiten sind in der Regel auf der Festplatte gespeichert.
- Der physikalische Speicher ist ein **Seitenrahmen (page frames)** eingeteilt.
- Die MMU/TLB bildet über die **Seitentabelle** Seiten auf Seitenrahmen ab. Eine Seitentabelle pro Prozess.
- Nur die gerade benutzten / relevanten Seiten sind auf Seitenrahmen abgebildet – stehen im Hauptspeicher.
- I.a. sind Seiten und Seitenrahmen gleich groß (zwischen 512 Byte und 64KB)

Abbildung des virtuellen Speichers auf den physikalischen Speicher

Platte: Virtueller Adressraum (64K)
 Hauptspeicher: 32K
 4096 Byte pro Frame bzw. Page

Beispiele:

Virtuelle Adresse	Physikalische Adresse
0	8192
8192	24576
20500	12308
32780	Seitenfehler



Seitentabelle

	page frame	present bit
15		0
14		0
13		0
12		0
11	7	1
10		0
9	5	1
8		0
7		0
6		0
5	3	1
4	4	1
3	0	1
2	6	1
1	1	1
0	2	1

Abbildung: virtuelle Adresse → physikalische Adresse

- **Standardtechnik:** Zweiteilung der virtuellen Adresse:

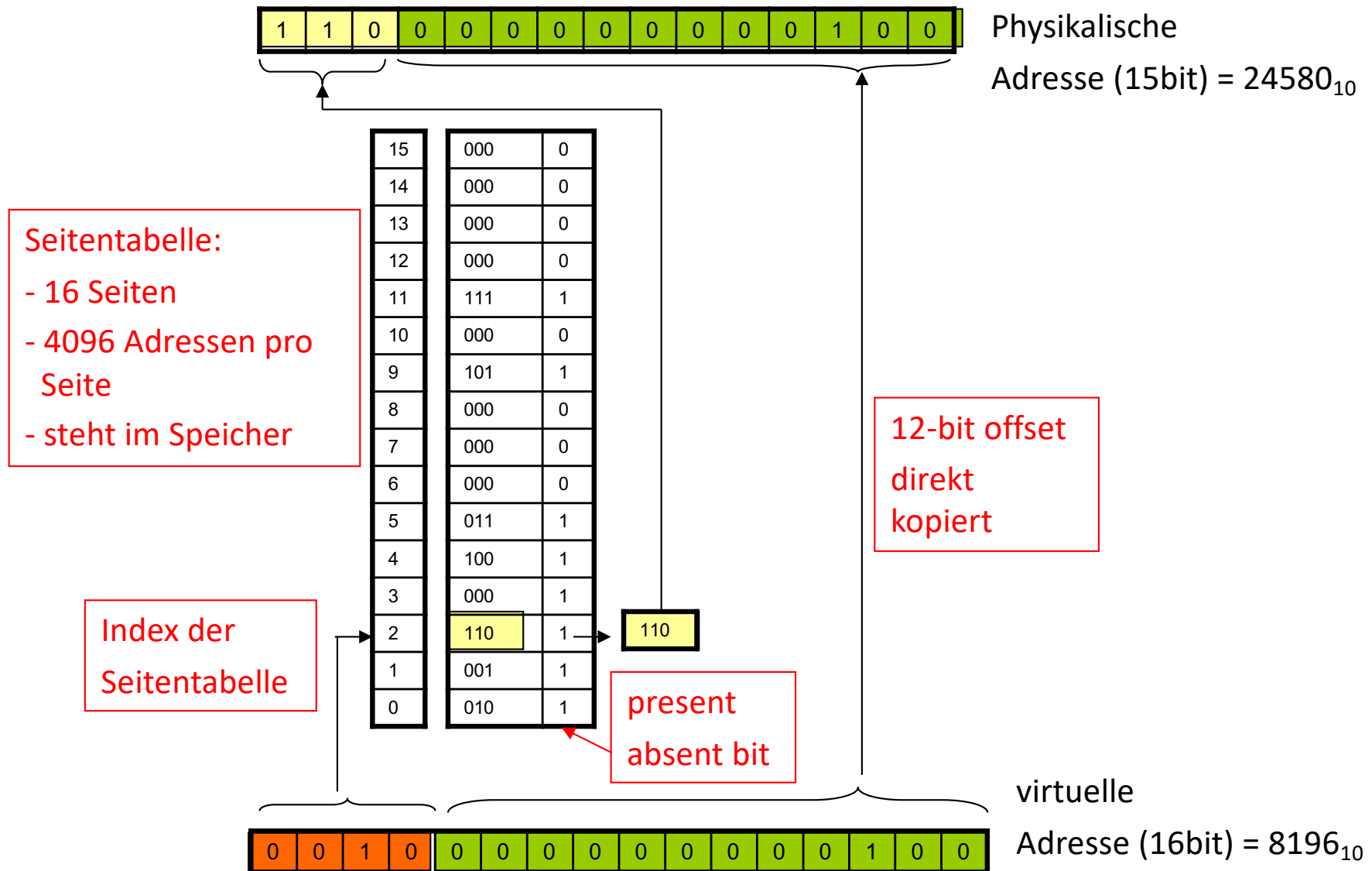
virtuelle Adresse



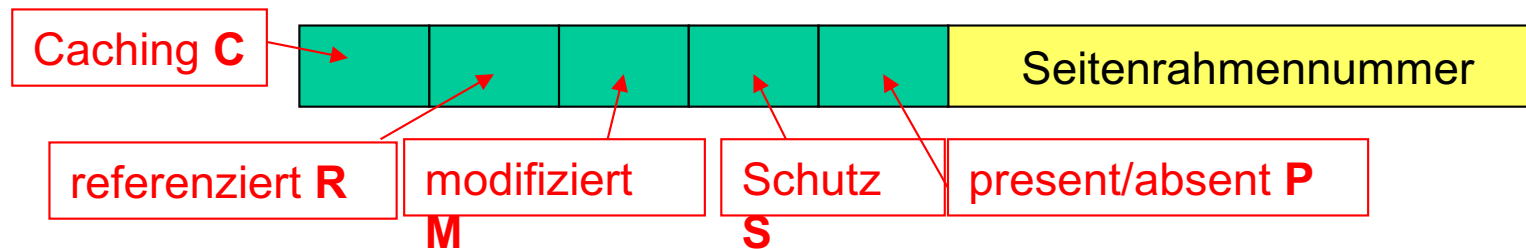
- Seitennummer dient als Index in der Seitentabelle
- Offset: physikalische Adresse im Seitenrahmen

$$\text{adr}_{\text{physikalisch}} = f_{\text{Seitentabelle}}(\text{adr}_{\text{logisch}})$$

Beispiel: 64K virtueller AR, 32K physikalischer AR



Seitentabelleneintrag



- P Bit: zeigt an, ob die Seite im Speicher steht

P = 0 : Seite ist nicht im Hauptspeicher: → Seitenfehler: die Seite muss in den Hauptspeicher geladen werden.

- S Bit: Schreib/Lese Schutz
- M Bit: Die Seitenrahmen hat andere Werte als die Seite auf der Festplatte
M = 1 : Beim Auslagern muss die Seite auf Festplatte geschrieben werden
- R-Bit: auf die Seite wurde zugegriffen (lesend oder schreibend)
- C Bit: Bei Eingabe per I/O-Gerät
C = 0 : Seite darf nicht gecached werden

Praktische Überlegungen

Beispiel:

- 32 Bit breite virtuelle Adressen
- 4 KB Seitengröße

Größe der Seitentabelle

- $2^{32} / 2^{12} = 2^{20} \approx 1$ Million Einträge
- Bei 4 Byte pro Eintrag: Größe der Seitentabelle : 4 MB

Zeit zur Umrechnung einer virtuellen Adr. in eine physikalische Adr.

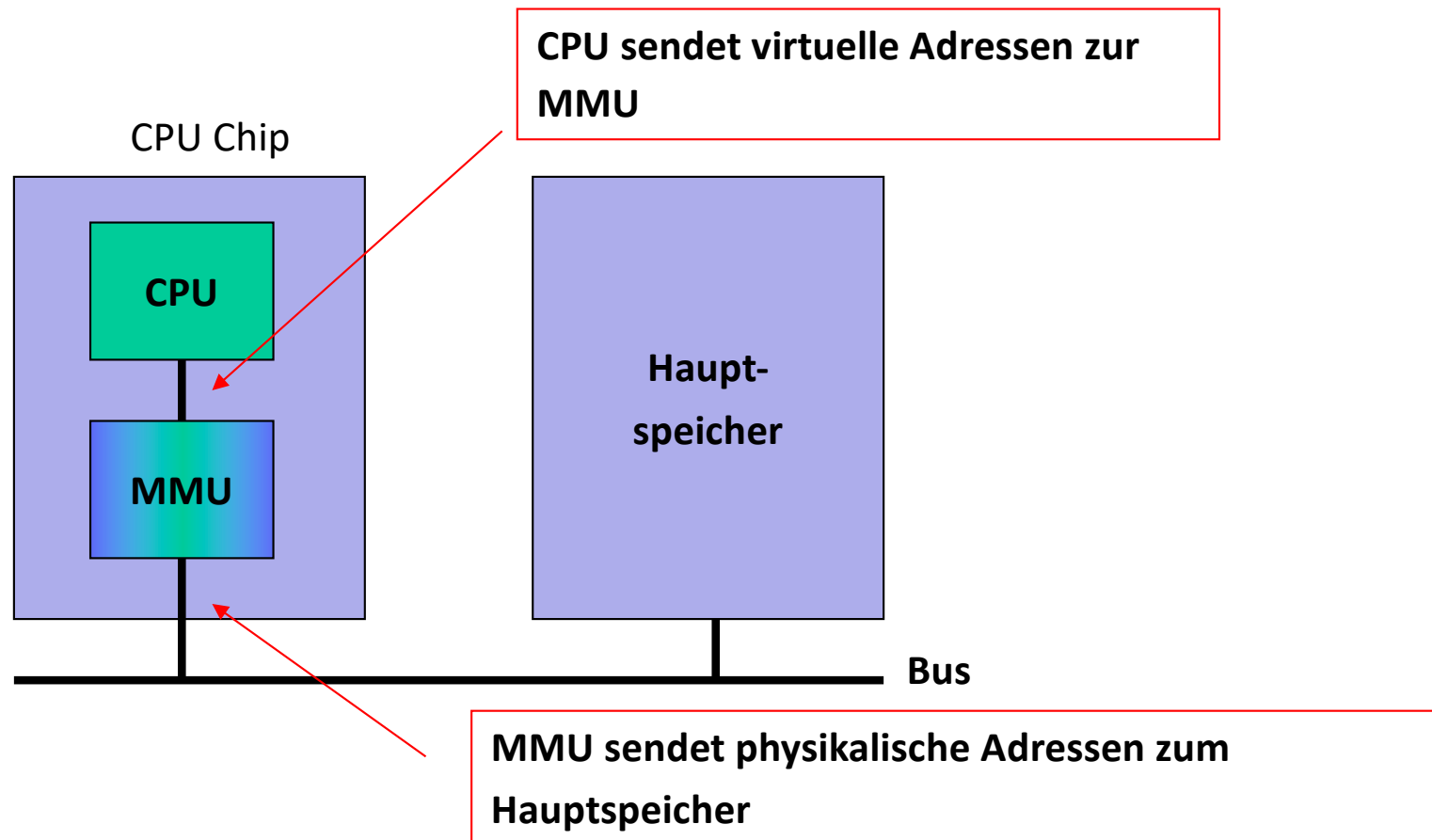
- Zugriff auf die Seitentabelle darf nicht länger als 1ns dauern, ansonsten werden Tabellenzugriffe zum Engpass

Bedarf: Schnelles Abbilden - auch für sehr große Seitentabellen

Vorteil: Workingset

MMU

Die MMU (Memory Management Unit) bildet virtuelle Adressen auf physikalische Adressen ab.



TLB – Translation Lookaside Buffer

TLB: Komponente der MMU, ist ein Assoziativspeicher

- Für eine kleine Zahl (32 bis 1024) häufig genutzter Seiten, bildet der TLB virtuelle Adressen auf physikalische Adressen ab - unter Umgehung der Seitentabelle
- Trefferquoten in der Praxis: 80% - 98%
- Diskussion: Wer behandelt TLB Miss (HW oder BS)?

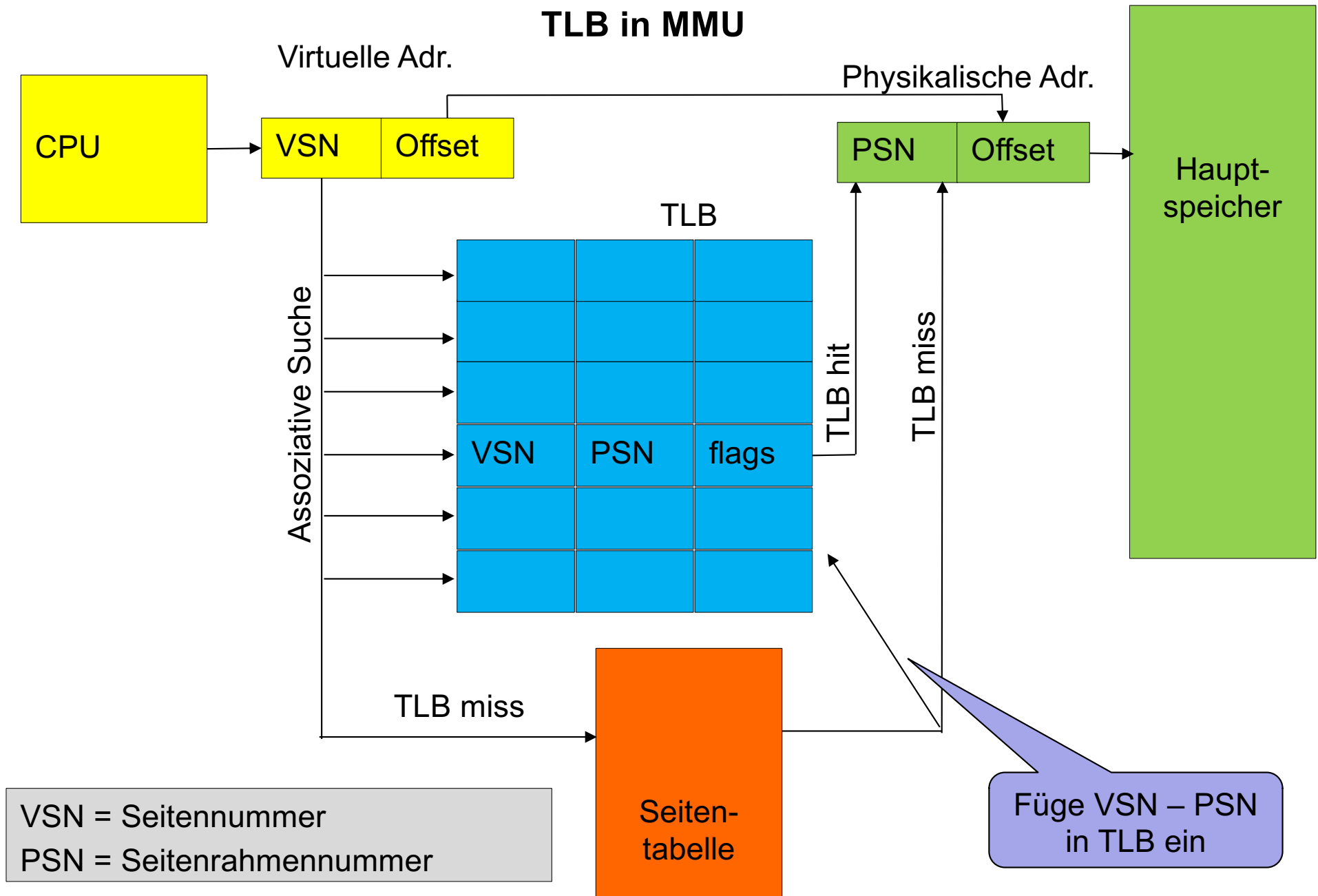
TLB zur Beschleunigung

P Bit	Virtuelle Seite	M Bit	S Bit	Seitenrahmen
1	140	1	RW	31
1	20	0	R	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R	50
1	21	0	R	45
1	860	1	RW	14
1	861	1	RW	75

Arbeitsweise:

- Falls Seitennummer (virtuell) im TLB (suche parallel)
Überprüfe Schreibschutz auf Basis des S-Bits
(ggf. löse Schutzverletzung aus)
Falls Schreibschutz o.k., nehme Seitenrahmennummer aus dem TLB
- Falls Seitenadresse nicht im TLB
 - MMU nimmt Seitenrahmennummer aus Seitentabelle (wie gewöhnlich)
 - Überschreibe „ältesten“ Eintrag im TLB mit dieser Seitenadresse + restliche Info, schreibe vorher M-Bit des ältesten Eintrag in die Seitentabelle zurück.

TLB in MMU

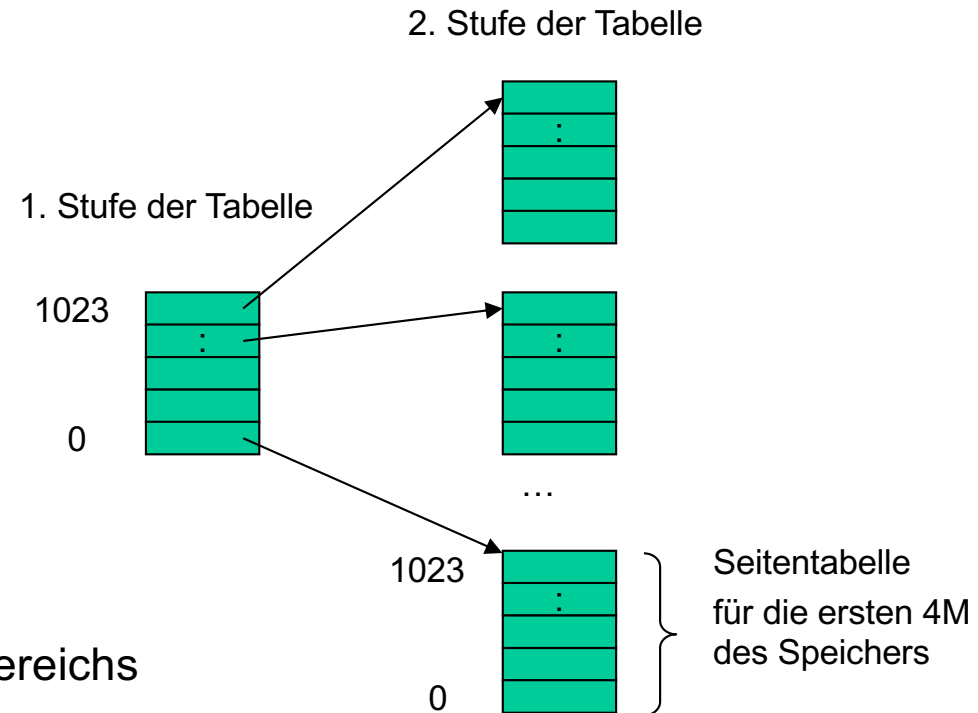
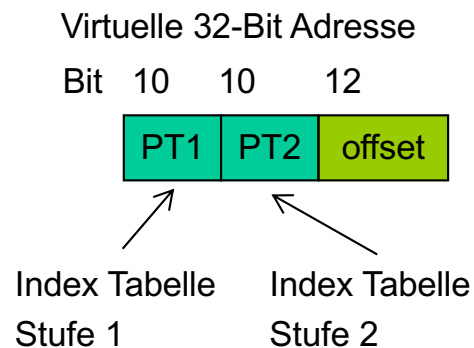


Mehrstufige Seitentabellen

Problem: Große virtuelle Adressräume führen zu sehr großen Seitentabellen.

Idee: Seitentabelle wird in 2 oder mehr Stufen aufgebaut. Nur Teile der Seitentabelle werden zeitgleich im Speicher gehalten (müssen existieren)

Beispiel: 2 stufige Seitentabelle, 32-bit Adresse mit 2x10-Bit Adressen für die Seitennummern und 12-Bit Offset: zweistufige Seitentabelle



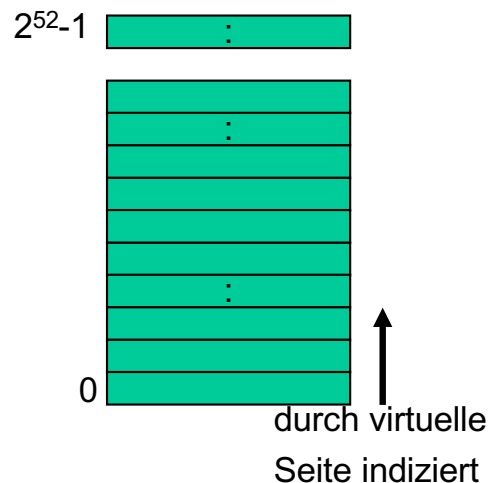
Diskussion

- Zugriff außerhalb des Speicherbereichs eines Prozesses

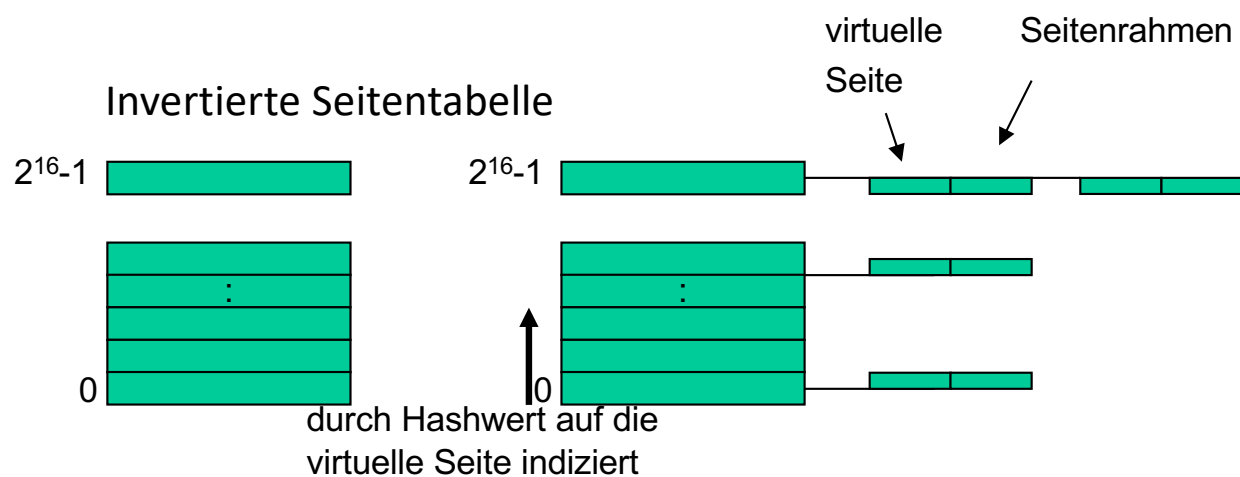
Invertierte Seitentabellen

- **Beispiel:** 64 Bit breite virtuelle Adressen, Seitengröße 4 KB, 256 MB Speicher. Größe der Seitentabelle?
- **Ansatz der invertierten Seitentabelle:** Seitentabelle hält Einträge für jeden physischen Seitenrahmen (und nicht für die Seiten selbst).
Eintrag: Prozess + Seitennummer – also auch nur eine Tabelle für alle Prozesse.
- **Vorteil:** spart enorme Menge an Speicherplatz
Beispiel: $256 \text{ MB} / 4 \text{ K} = 65536$ Einträge
- **Nachteil:** Abbildungsfunktion aufwendig (langsam)
→ praktikabel: Nutze Hashtabelle mit Hashwerten $H(\text{ProcessId}, \text{virtuelle_Adresse})$
→ virtuelle Seiten im Speicher mit gleichen Hashwert sind verkettet

Normale Seitentabelle



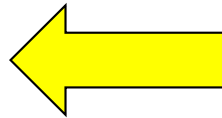
Invertierte Seitentabelle



Kapitel 6: Speicherverwaltung

Gliederung

- Einführung und Grundlagen
- Swapping
- Virtuelle Speicherverwaltung
- Seitenersetzungsverfahren
- Entwurfsaspekte
- Unix / Windows



Motivation

- Bei Seitenfehler wählt BS eine Seite aus, welche aus dem Speicher entfernt wird, um einer neuen Seite Platz zu machen.
- Prinzipielle Schritte:
 - Wähle auszulagernde Seite aus.
 - Wurde die Seite modifiziert, so wird die korrespondierende Seite auf der Platte aktualisiert.
 - Ersetze die alte Seite durch eine neue Seite.
- Ziel: Geringe Anzahl von Seitenfehlern

Ladestrategien

Demand Paging liest Seiten erst ein, wenn auf sie zugegriffen wird

- Einlesen der Seite bei Page-Fault
- Viele Page-Faults bei Prozess-Start

Prepaging liest neben der angeforderten Seite einige weitere Seiten mit ein

- z.B. die nächsten Seiten des Programmcodes oder beim Prozesswechsel die „zuletzt“ genutzten Seiten des Prozesses.
- Entspricht der Charakteristik der Platte – d.h. ist effektiv – wenn Seiten auf der Platte physikalisch hintereinander liegen
- Aber: Werden die Seiten wirklich benötigt?

Optimaler Algorithmus

Verfahren

1. Markiere jede Seite mit Anzahl der Instruktionen, die zur Ausführung gelangen, bevor auf diese Seite das nächste mal referenziert wird.
2. Entferne die Seite mit größter Markierung.

Beurteilung

- "optimal", da die aktuell am wenigsten genutzte Seite ausgelagert wird.
- nicht implementierbar, da Markierungen nicht ermittelbar sind – Blick in die Zukunft.
- dennoch sinnvoll, da per Simulation und "Zweifachdurchlauf" für konkretes Programm Vergleichsmöglichkeit mit anderen Algorithmen besteht.

Not-Recently-Used Algorithmus (NRU)

Verfahren

1. BS ordnet die Seiten in **“vier” Kategorien** ein
2. Entferne eine **zufällige Seite aus der niedrigsten Kategorie**, welche zumindest eine Seite enthält

Kategorie	Referenziert (R Bit)	Modifiziert(M Bit)
0	0	0
1	0	1
2	1	0
3	1	1

Wann wird welches Bit gesetzt?

Zyklisches Rücksetzen des R Bits
(typisch alle 20 ms)

Beurteilung

- Grundlage: Working Set, wobei R-Bit stärker gewichtet ist als M-Bit
- leicht zu verstehen und effizient implementierbar
- bessere Leistung wünschenswert, jedoch oftmals ausreichend

NRU Beispiel

Seite	Lade-zeitpunkt	Letzter Referenz-zeitpunkt	R	M
A	126	259	0	0
B	230	260	1	0
C	120	272	1	1
D	160	280	1	1

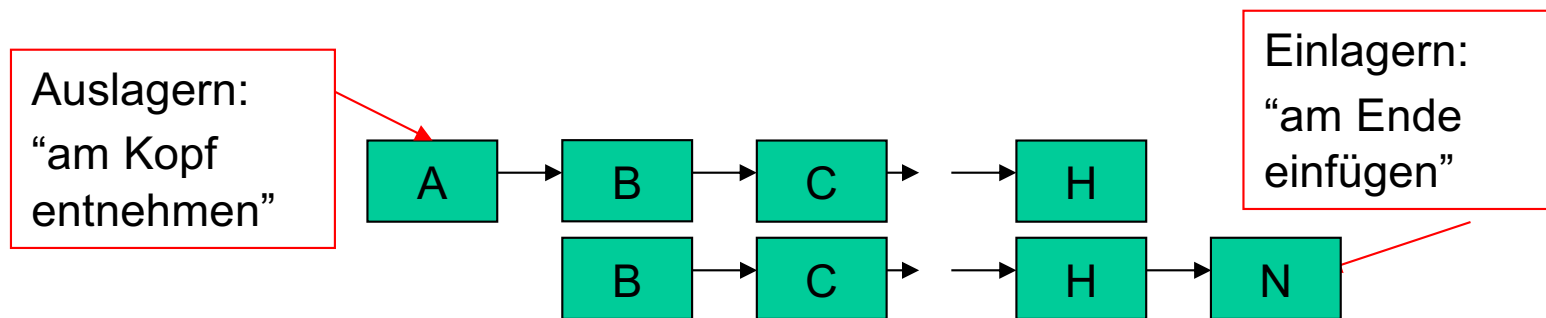
Welche Seite wird entfernt? Seite A

Diskussion: Will man „Recently“ exakt implementieren, dann muss jede Seite mit einem Zeitstempel für den letzten Zeitpunkt der Benutzung versehen werden. Das ist in der Realität aber extrem aufwendig! R-Bit als Näherung

First In First Out (FIFO)

Verfahren

1. Die Seiten stehen als verkettete Liste im Speicher
2. Bei Seitenfehler wird Seite am Listenkopf entfernt
3. Neue Seite wird an das Ende gesetzt



Beurteilung

- Die älteste Seite wird ausgelagert
- Keine Unterscheidung zwischen intensiv genutzten Seiten und wenig genutzten Seiten
- FIFO ist für den praktischen Einsatz ungeeignet

FIFO Beispiel

Seite	Lade-zeitpunkt	Letzter Referenz-zeitpunkt	R	M
A	126	259	0	0
B	230	260	1	0
C	120	272	1	1
D	160	280	1	1

Welche Seite entfernt NRU? Seite A

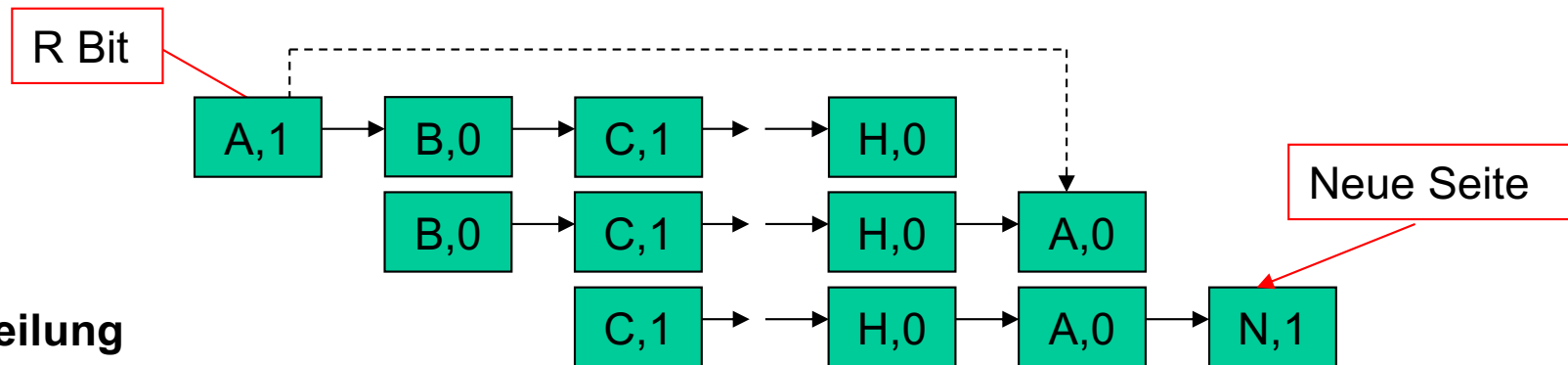
Welche Seite entfernt FIFO? Seite C

Diskussion: Working Set
Wird R-Bit benötigt?

Zweite Chance (2C-FIFO)

Verfahren (Variante von FIFO, die aktuelle Zugriffe beachtet)

1. Die Seiten stehen als verkettete Liste im Speicher
2. Bei Seitenfehler untersuche Listenkopf:
If $R=0$, lösche Kopfseite und füge neue Seite mit $R:=1$ an das Ende der Liste
If $R=1$, verschiebe Kopfseite an das Ende der Liste und setze $R:=0$
3. Wiederhole Schritt 1 solange bis eine Seite ersetzt wurde



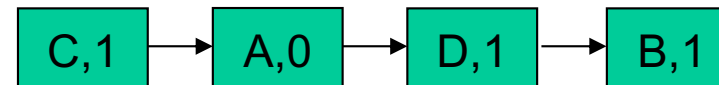
Beurteilung

- Was passiert, wenn bei allen Seiten das R-Bit gesetzt ist?
- Relativ einfach realisierbar
- Relativ hoher Verwaltungsaufwand → Verschieben von Listenelementen

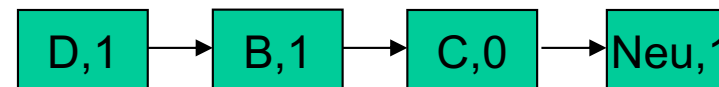
2C-FIFO Beispiel

Seite	Ladezeitpunkt	Letzter Referenzzeitpunkt	R	M
A	126	259	0	0
B	230	260	1	0
C	120	272	1	1
D	160	280	1	1

Welche Seite entfernt NRU? Seite A



Welche Seite entfernt FIFO? Seite C

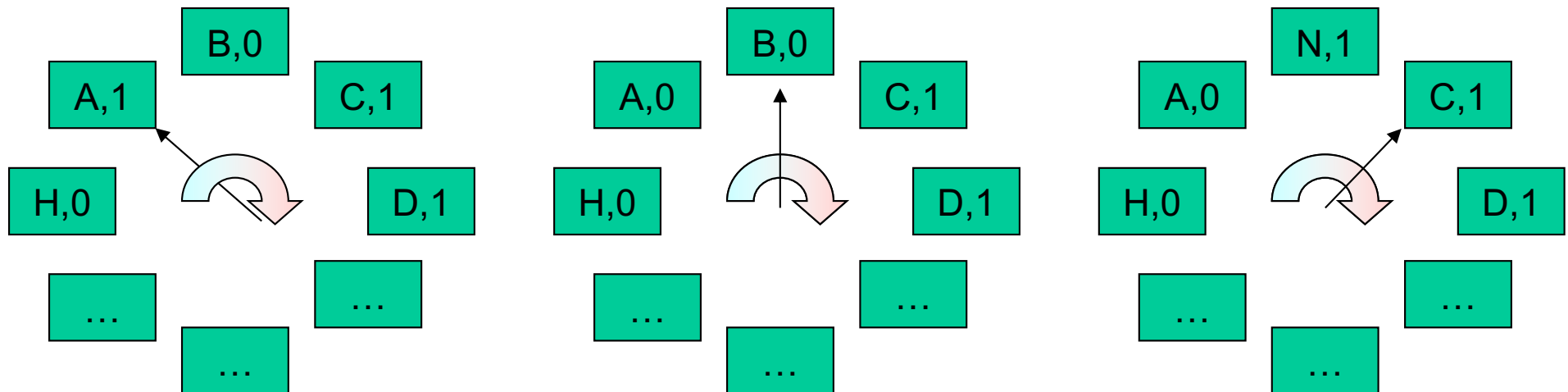


Welche Seite entfernt 2C-FIFO? Seite A

Clock

Verfahren (Spare Ein/Ausketten von 2C-FIFO durch zyklische Liste)

1. Die Seiten stehen als zyklisch verkettete Liste im Speicher
2. Bei Seitenfehler untersuche zyklische Liste:
If $R==0$, lösche Seite und setze neue Seite mit $R:=1$ ein
else if $R==1$, setze $R:=0$ fi
Gehe zum nächsten Element
3. Wiederhole Schritt 2 bis Fall $R==0$ eingetreten ist



Beurteilung: Gleiches Ersetzungsverhalten wie 2C-FIFO, aber niedrigerer Verwaltungsaufwand als 2C-FIFO

Least Recently Used (LRU)

Beobachtung: Seiten, welche für die letzten Befehle oft genutzt wurden, werden mit hoher Wahrscheinlichkeit auch für die kommenden Befehle genutzt
Bei Seitenfehler: entferne die am längsten ungenutzte Seite

Verfahren 1

1. Alle Seiten stehen in einer verketteten Liste
2. **Beim Zugriff auf eine Seite**, wird Sie an den Anfang der Liste gesetzt
3. Die Seite am Ende der Liste ist die am längsten ungenutzte Seite

Verfahren 2 (Hardware)

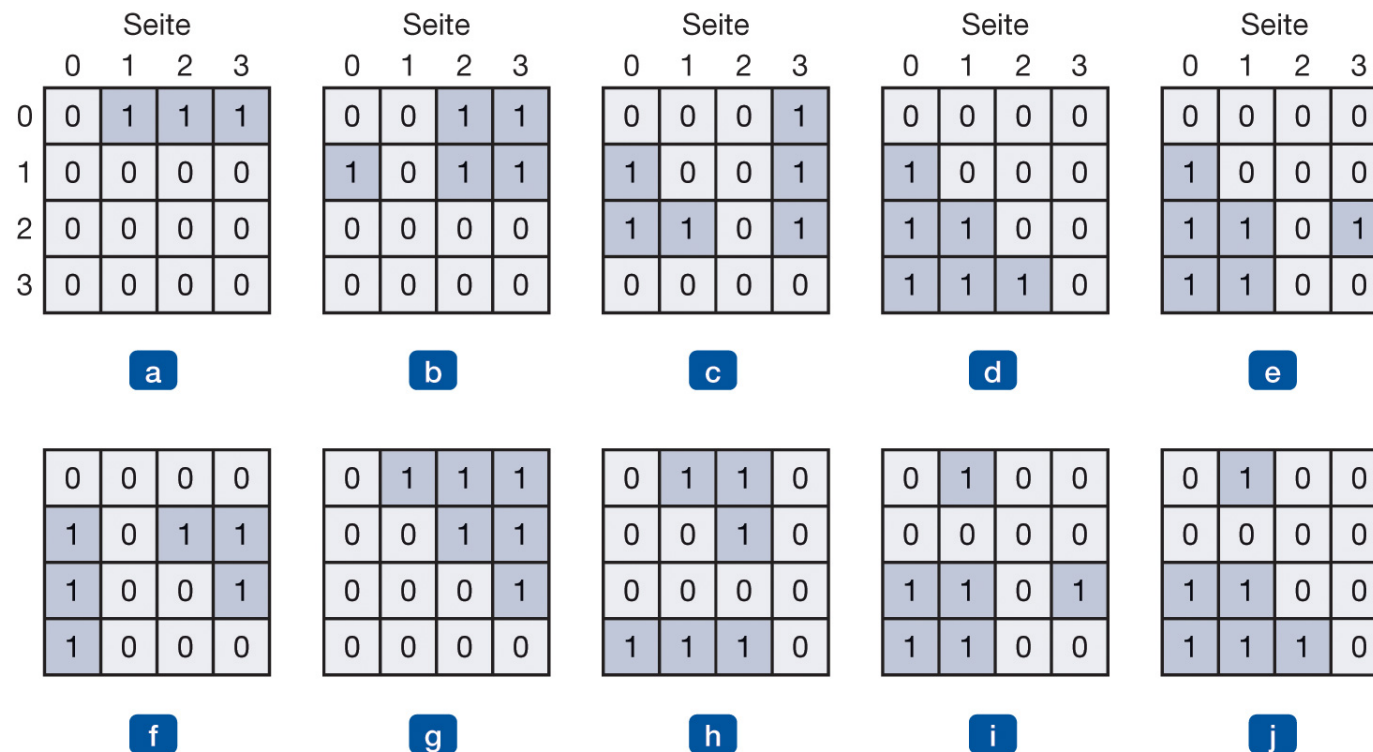
1. 64 Bit Register, das bei jedem CPU Takt erhöht wird
2. Pro Seite eine 64 Bit Eintrag, der bei Zugriff auf die Seite mit den aktuellen Zählerwert belegt wird
3. Die Seite mit dem niedrigsten Eintrag ist die am längsten ungenutzte Seite

Beurteilung: Aufwendig, auch in Hardware. Kommt dem optimalen Algorithmus sehr nahe.

Least Recently Used (LRU)

Verfahren 3 (Hardware)

1. Bei n Seitenrahmen wird eine $n \times n$ Bitmatrix benötigt, die mit 0 initialisiert ist.
2. Zugriff auf Seitenrahmen n : n -te Zeile wird auf 1 gesetzt, n -te Spalte wird auf 0 gesetzt.
3. Interpretiere die Zeilen als Binärzahlen (unsigned): Die Zeile mit dem niedrigsten Wert wurde am längsten nicht benutzt.



[AT]

Abbildung 3.17: LRU mit einer Matrix. Auf die Seiten wird in der Reihenfolge 0 1 2 3 2 1 0 3 2 3 zugegriffen.

LRU Beispiel

Seite	Lade-zeitpunkt	Letzter Referenz-zeitpunkt	R	M
A	126	259	0	0
B	230	260	1	0
C	120	272	1	1
D	160	280	1	1

Welche Seite entfernt NRU? Seite A

Welche Seite entfernt FIFO? Seite C

Welche Seite entfernt 2C-FIFO? Seite A

Welche Seite entfernt LRU? Seite A

Not Frequently Used (NFU)

Beobachtung: LRU sehr aufwendig, Spezialhardware oftmals nicht vorhanden. Bilde LRU näherungsweise in SW nach.

Verfahren

1. Jede Seite hat einen SW Zähler, der mit 0 initialisiert ist.
2. Zyklisch (z.B.: Timer Intervall alle 20 ms) werden die R-Bit zu den Zählern addiert.

Beurteilung:

- NFU vergisst keine alten Zugriffe (Problem): Alte, oft referenzierte Seiten bleiben im Speicher „kleben“, auch wenn sie nicht mehr benutzt werden.
- Unschärfe: Es wird nicht gespeichert, wie oft eine Seite in einem Intervall referenziert wird.

Aging

Verfahren: Das Problem, das NFU nicht vergisst, wird gelöst.

1. Jede Seite hat einen SW Zähler, der mit 0 initialisiert ist.
2. Zyklisch (z.B.: Timer Intervall alle 20 ms) werden die R-Bit zu dem Zähler wie folgt addiert:
 - Shifte den Zähler im 1 nach rechts (Division durch 2)
 - Setze das R Bit an die linke Position (addiere $R\text{-Bit} * 2^{\text{hochwertiges Bit-Position}}$)
 - Setze das R Bit zurück

Beurteilung:

- Unschärfe wie bei NFU
- Da 2-er Potenzen addiert werden, ist die Breite der Zähler „schnell verbraucht“. Beispiel: n Bit breiter Zähler: Kein Unterscheidung, ob auf eine Seite in den letzten n oder n+x Zyklen nicht referenziert wurde.
- Was passiert in folgenden Fall: Seite wird eingelagert, page fault tritt auf, Aging Time ist in der Zeit nicht abgelaufen. Die frisch eingelagerte Seite wird wieder ausgelagert, da age noch 0 ist. Lösung: Setze age beim Einlagern der Seite auf 0x80 (bei 8 Bit Breite von age)

Beispiel Aging

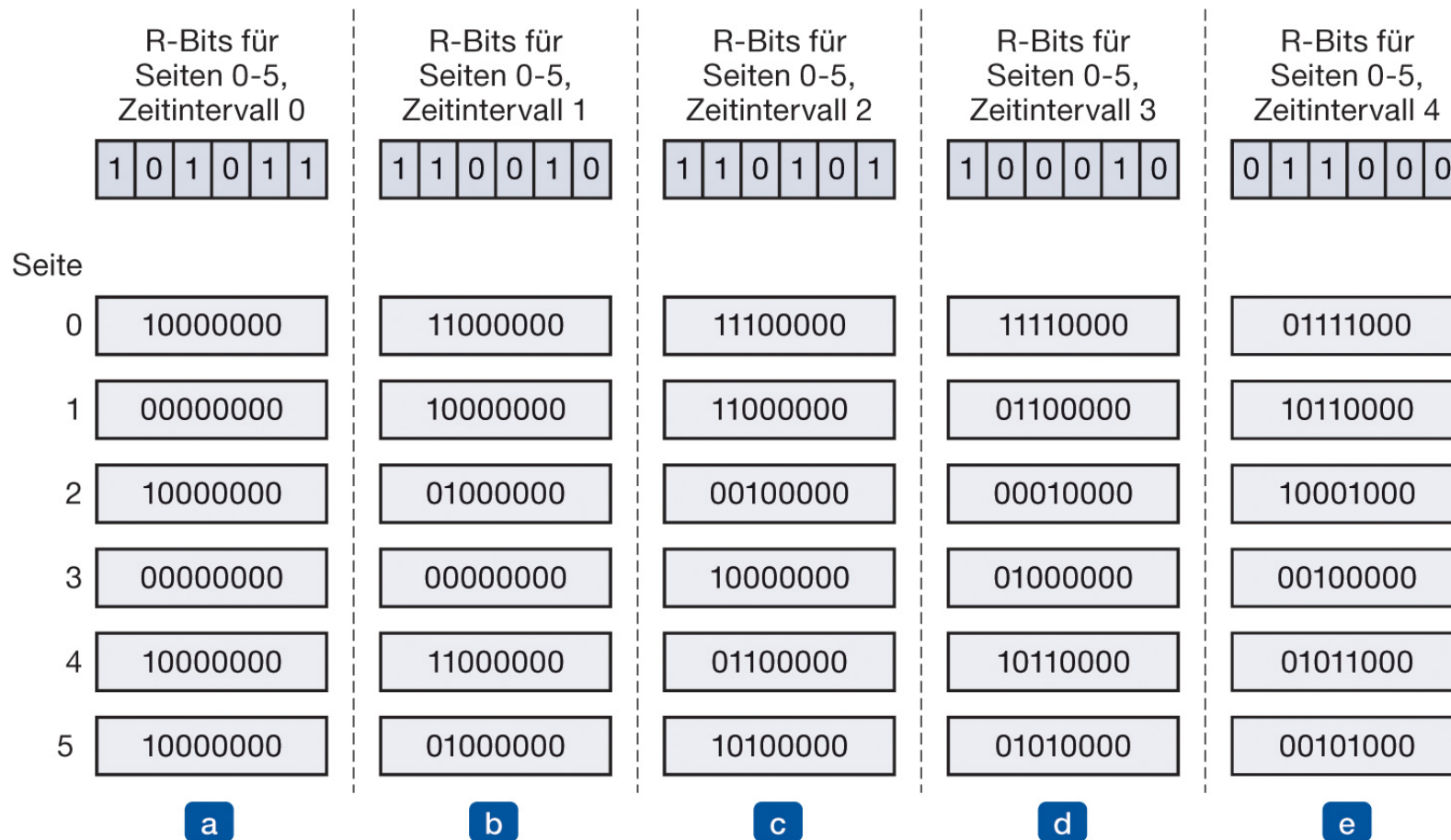
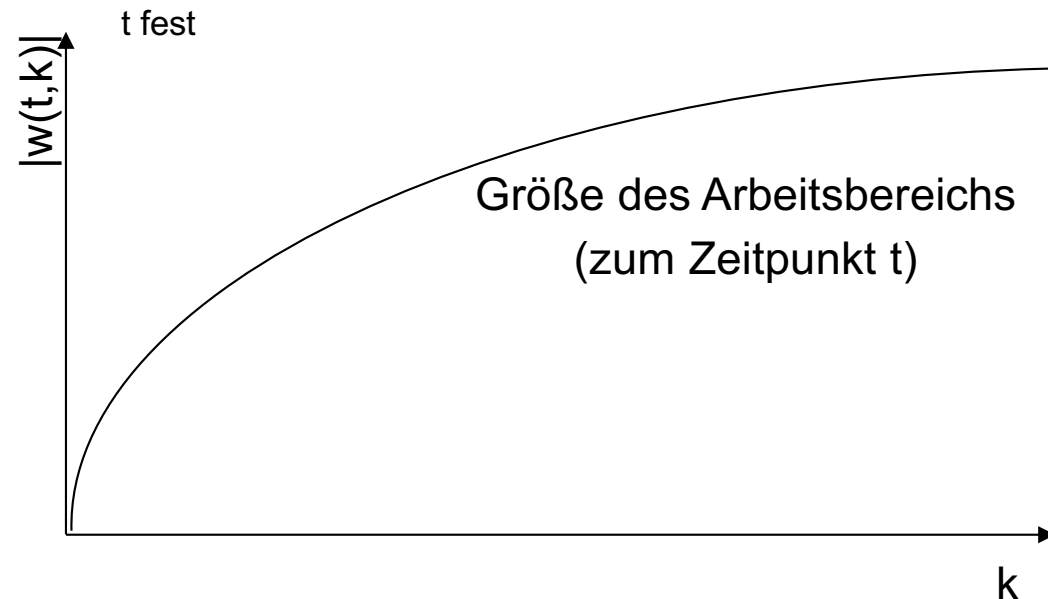


Abbildung 3.18: Der Aging-Algorithmus ist eine Software-Simulation von LRU. Dargestellt werden die Zähler von sechs Seiten für fünf Intervalle. (a) bis (e) zeigen die Zustände nach den Intervallen 1–5.

Working Set Algorithmus

Arbeitsbereich Working Set eines Prozesses

- Lokalitätsprinzip
- zu jedem Zeitpunkt t gibt es eine Menge von Seiten, die in den letzten k Speicherzugriffen genutzt wurden
- diese Menge $w(t,k)$ ist der Arbeitsbereich



Viele BS merken sich den (**Working Set**) eines Prozesses wenn sie ihn auslagern

Grund: Prepaging - letzter aktueller Arbeitsbereich wird später wieder geladen bevor Prozess weiter ausgeführt wird

Ziel: Verhindern von **Seitenflattern** (trashing) durch Seitenfehler (nur einige Nanosekunden zur Befehlsausführung, aber ca. 10ms eine Seite von Platte zu lesen)

Näherung: $w(k,t)$ aufwendig zu berechnen: Statt der letzten k Speicherzugriffe wird die Ausführungszeit des Prozesses verwendet.
 Multiprocessing: virtuelle Zeit (current virtual time)
 Arbeitsbereich eines Prozesses zum Zeitpunkt t : Die Seiten, im virtuellen Zeitintervall $[t-\tau, t]$

Working Set Algorithmus

Verfahren: Idee: Lagere bei einem Seitenfehler eine Seite aus, die nicht im Working Set liegt.

1. Betrachte nur die eingelagerten Seiten. Info pro Seite: R-Bit, M-Bit, der letzte Zugriff auf die Seite (ungefähr)
2. R-Bit, M-Bit wird von der HW gesetzt. Zyklisch wird das R-Bit zurückgesetzt
3. Durchlaufe die Tabelle, bis eine Seite ersetzt wurde.
 - (3.1) R-Bit == 1: Setze Zugriffszeit auf aktuelle virtuelle Zeit, R-Bit = 0
 - (3.2) R-Bit == 0: if (aktuelle virtuelle Zeit – τ) > letzte Zugriffszeit : Seite liegt nicht im Working Set, wird ausgelagert und durch die neue Seite ersetzt
4. Für die restlichen Seite : 3.1 – Zur Aktualisierung der Zugriffszeiten
5. Falls keine Seite mit 3.2 ausgelagert wurde, lagere die älteste Seite aus.

Beurteilung:

- Verfahren pro Prozess
- Aufwendig: alle Seitenrahmen werden bei einen Seitenfehler bearbeitet.

Beispiel

virtuelle Zeit	T
2204	200

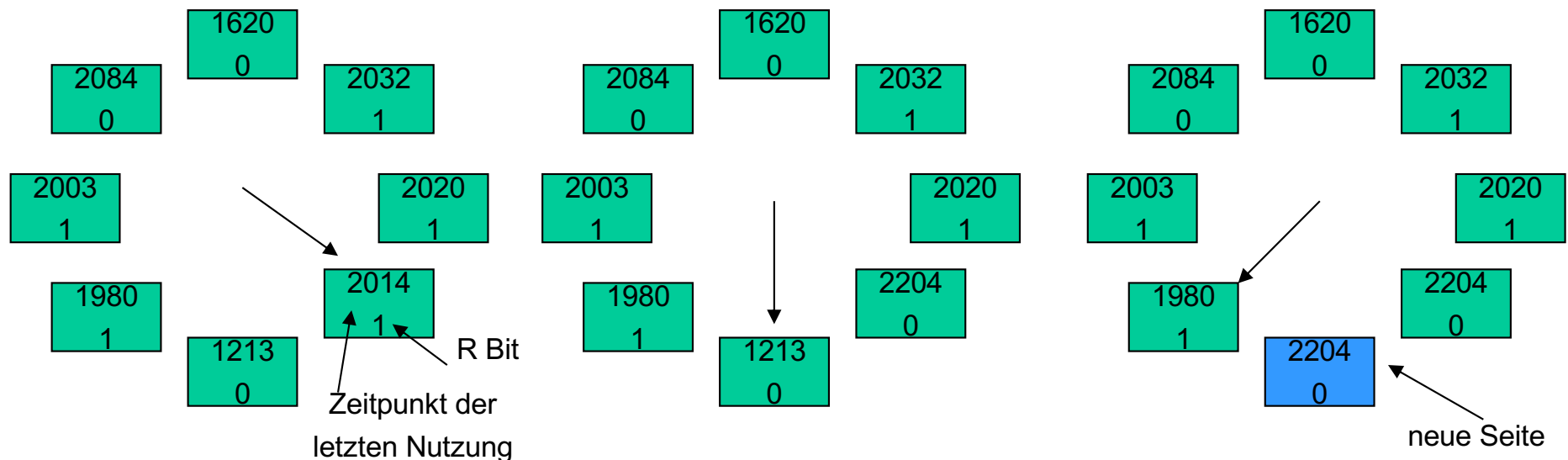
...	Zugriffszeit	R-Bit
	2084	1
	2005	0
	1980	1
	2000	0
	1903	0
	2020	1
	2032	0
	1620	0

...	Zugriffszeit	R-Bit
	2204	0
	2005	0
	2204	0
Neue Seite	2204	0
	1903	0
	2204	0
	2032	0
	1620	0

Working Set Clock Algorithmus (WSClock)

Verfahren:

1. Die eingelagerten Seiten stehen als zyklisch verkettete Liste im Speicher
Pro Seite: R-Bit, M-Bit, der letzte Zugriff auf die Seite (gemäß WS Algorithmus)
2. R-Bit, M-Bit wird von der HW gesetzt. Zyklisch wird das R-Bit zurückgesetzt
3. Bei Seitenfehler untersuche zunächst Seite auf die der Zeiger zeigt
if (R-Bit == 1) then R-Bit = 0, Zugriffszeit = aktuelle virtuelle Zeit, schiebe den Zeiger eine Seite weiter
if (R-Bit == 0) && (Seitenalter $\geq \tau$) then
Seite auslagern, neue Seite einlagern, Zugriffszeit = aktuelle virtuelle Zeit
4. Laufe noch den Rest der Liste durch: Update von Zugriffszeit und R, wenn R == 1



Working Set Clock Algorithmus (WSClock)

Optimierung:

3. ...
 if (R-Bit==0) && (Seitenalter $\geq \tau$) then
 Seite auslagern, neue Seite einlagern, Zugriffszeit = aktuelle virtuelle Zeit

Optimierung:

- Lage die Seite nur aus, wenn das M Bit nicht gesetzt ist.
- Wenn das M Bit gesetzt ist: Beauftrage das System zur Auslagerung der Seite und mache weiter mit dem Algorithmus.
Dies reduziert in der Regel die Zeit zur Behandlung eines Page Faults.

Beurteilung:

- Wird aufgrund der guten Leistung und einfachen Umsetzbarkeit eingesetzt.

Zusammenfassung Ersetzungsstrategien

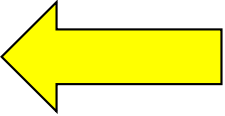
Algorithmus	Kommentar
Optimal	Nicht realisierbar, aber nützlich als Maßstab
NRU (Not Recently Used)	Sehr grobe Annäherung an LRU
FIFO (First In First Out)	Entfernt evtl. auch wichtige Seiten
Second Chance	Enorme Verbesserung gegenüber FIFO
Clock	Realistisch
LRU (Least Recently Used)	Exzellent, aber schwierig zu implementieren
NFU (Not Frequently Used)	Ziemlich grobe Annäherung an LRU
Aging	Effizienter Algorithmus, gute Annäherung an LRU
Working Set	Etwas aufwändig zu implementieren
WSClock	Guter und effizienter Algorithmus

Abbildung 3.22: Die behandelten Seitenersetzungsalgorithmen

[AT]

Kapitel 6: Speicherverwaltung

Gliederung

- Einführung und Grundlagen
- Swapping
- Virtuelle Speicherverwaltung
- Seitenersetzungsverfahren
- Entwurfsaspekte 
- Unix / Windows

Speicherzuteilungsstrategien: Grundlegende Überlegungen

Ein Ansatz: Feste Anzahl von Seitenrahmen pro Prozess

- Je weniger Platz für den einzelnen Prozess zur Verfügung steht, desto mehr Prozesse können im Hauptspeicher resident sein
(➔ Anzahl Seitenrahmen pro Prozess minimieren!)
- Stehen einem Prozess zu wenig Seitenrahmen zur Verfügung, dann wird die Seitenfehlerrate trotz Lokaliätsprinzip sehr hoch sein
(➔ Anzahl Seitenrahmen pro Prozess maximieren!)
- Über eine bestimmte Größe hinaus wird sich zusätzlicher Speicher nur geringfügig auf die Seitenfehlerrate auswirken (Working Set)
(➔ Anzahl Seitenrahmen pro Prozess optimieren!)

Speicherzuteilungsstrategien: Grundlegende Überlegungen

- Verteilung der freien Hauptspeicherseitenrahmen auf die existierenden Prozesse:
 - Einem Prozess wird zum Start eine **feste Anzahl von Seitenrahmen** zugebilligt.
 - Einem Prozess werden während seiner Ausführung abhängig von seinem Verhalten (Seitenfehlerrate, I/O Verhalten etc.) eine **variable Anzahl von Seitenrahmen** zur Verfügung gestellt.
- Strategien nach Auftreten von Seitenfehlern :
 - Bei **lokalen Strategien** muss eine Seite des aktiven Prozesses ersetzt werden.
 - Bei **globalen Strategien** sind alle Seitenrahmen im Hauptspeicher Kandidaten für die Ersetzung.

Speicherzuteilung: Feste Seitenrahmenanzahl und lokale Strategie

- Einem Prozess steht für seine Ausführung eine feste Anzahl von Seitenrahmen zur Verfügung.
- Tritt ein Seitenfehler auf, dann muss das Betriebssystem entscheiden, welche andere Seite dieses Prozesses ersetzt werden muss.
- Das wesentliche Problem liegt in der Festlegung der Seitenrahmenanzahl (x % der gesamten Prozessgröße?)
- Der Bedarf kann sich über die Zeit ändern.

Speicherzuteilung: Variable Seitenrahmenanzahl und globale Strategie

- Den im Hauptspeicher befindlichen Prozessen stehen eine variable Anzahl von Seitenrahmen zur Verfügung.
- Seitenfehler → Zuweisung eines freien Seitenrahmen, falls verfügbar (von beliebigem Prozess)
- Kein freier Seitenrahmen mehr verfügbar → Seite zur Ersetzung auswählen (beliebiger Prozess!)
- Zu viele Prozesse im Hauptspeicher
 - u.U. sind die zur Verfügung stehenden Speicherbereiche nicht mehr ausreichend und die Seitenfehlerrate wird sehr groß
→ ggf. Prozesse vollständig auf Platte auslagern
 - Jedes Programm, das zusätzliche Seitenrahmen benötigt, erhält diese auf Kosten von Seiten anderer Programme. Da diese aber ebenfalls noch benötigt werden, werden in immer schnellerer Folge weitere Seitenfehler erzeugt (Thrashing). Die Prozesse verbrauchen dann für das Seitenwechseln mehr Zeit als für ihre Ausführung.
 - Prozesse vollständig auf Platte auslagern.

Variable Seitenrahmenanzahl und lokale Strategie: Working Set-Strategie

Erinnerung: Je größer der Working Set, umso geringer die Seitenfehlerrate, aber die Kurve konvergiert.

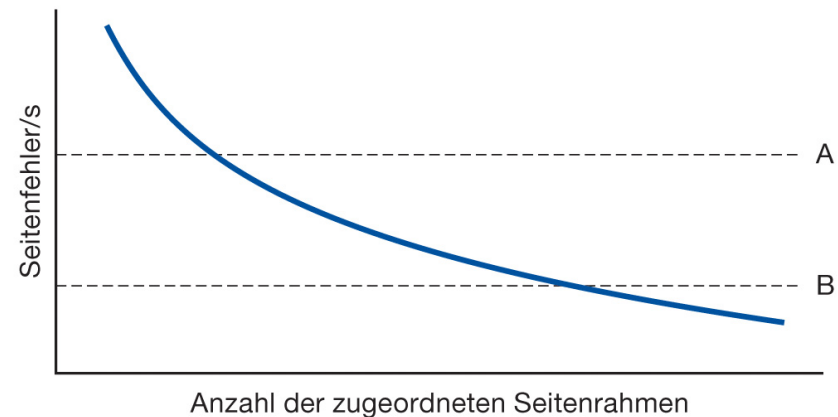
Idee:

- Das Working Set jedes Prozesses wird beobachtet.
- Seitenfehler → Zuweisung eines freien Seitenrahmens, falls verfügbar
- Periodisch wird die zugewiesene Seitenrahmenanzahl an den Working Set angepasst: Diejenigen Seiten werden aus dem Speicher entfernt, die nicht mehr zum Working Set gehören
- Ein Prozess darf nur dann ausgeführt werden, wenn sein Working Set im Hauptspeicher resident ist

Mögliche Implementierung (Annäherung): WSClock anpassen

Alternative / Ergänzung: Page-Fault Frequency (PFF) - Strategie

- Für die Seitenfehlerrate (Anzahl Seitenfehler pro s) werden untere (B) und obere (A) Grenzen definiert.



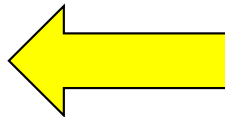
- Seitenfehler → Ersetzung einer eigenen Seite
- Wird die untere Grenze B erreicht, dann werden dem Prozess Seitenrahmen weggenommen
- Erreicht ein Prozess in Ausführung die obere Grenze, dann werden ihm – wenn möglich – neue Seitenrahmen gegeben.
- Sind keine weiteren Seitenrahmen verfügbar, so wird der Prozess suspendiert und ausgelagert.

Abbildung 3.24: Die Seitenfehlerrate in Abhängigkeit von der Anzahl der zugewiesenen Seitenrahmen

Kapitel 6: Speicherverwaltung

Gliederung

- Einführung und Grundlagen
- Swapping
- Virtuelle Speicherverwaltung
- Seitenersetzungsverfahren
- Entwurfsaspekte
- Unix / Windows



Virtueller Speicher in UNIX: Paging

- **Speicherzuteilung:** Variable Seitenrahmenanzahl und globale Strategie
- **Seitenfehler** → Zuweisung eines freien Seitenrahmen
- **Page Daemon** (Prozess-ID 2)
 - Wacht alle 250 ms auf
 - Falls Anzahl freier Seitenrahmen zu klein: Verwendet modifizierten Clock-Algorithmus, um beliebigen Prozessen Seitenrahmen zu entziehen
- **Swapper**
 - lagert Prozesse aus, falls zuviele Seitenfehler auftreten
 - lagert erst wieder ein, wenn genügend freie Seitenrahmen zur Verfügung stehen (Scheduling!)

Tanenbaum Kapitel 3 Aufgabe 31, Aufgabe 18

Zusammenfassung

