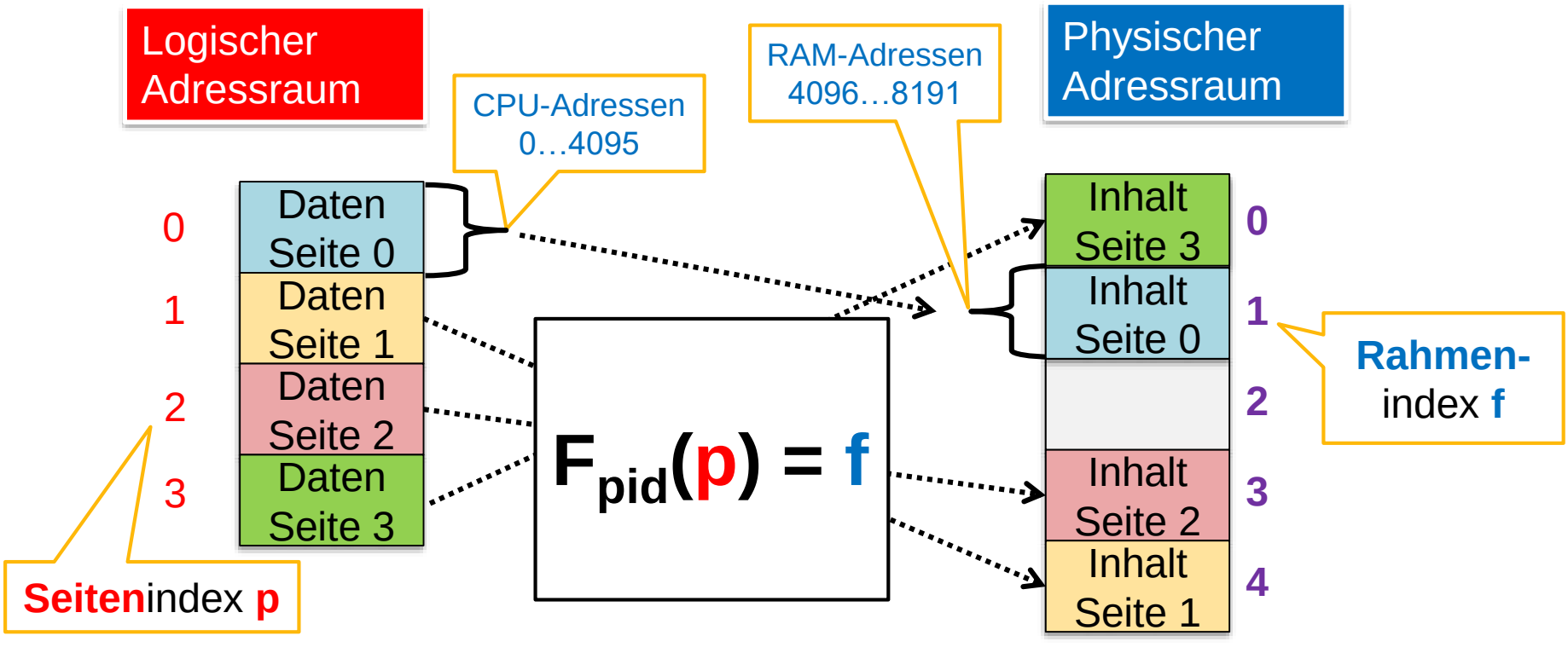
Betriebssysteme und Netzwerke Vorlesung 11

Artur Andrzejak

Zusammenfassung: Paging und Seitentabellen

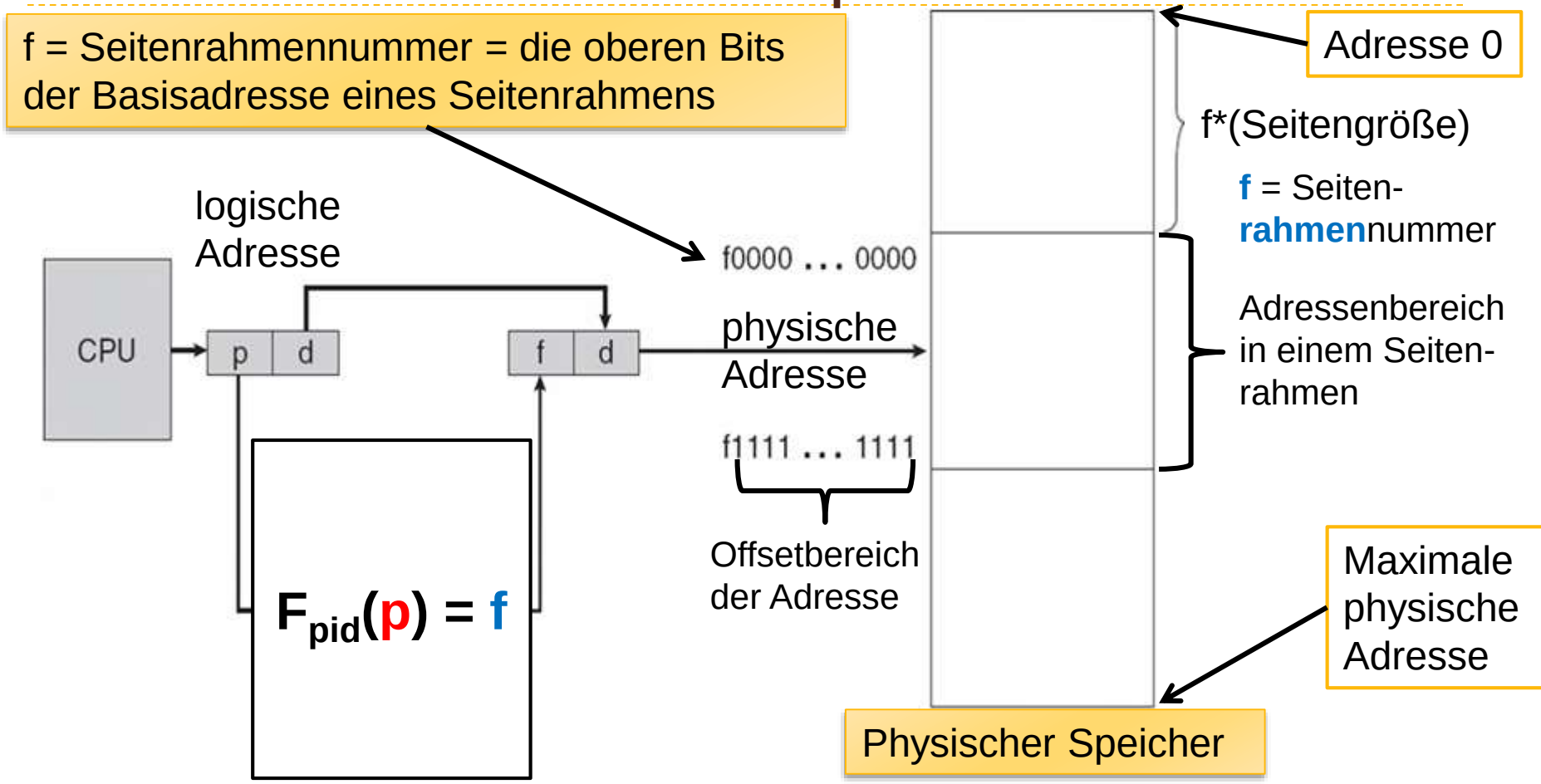
Paging: Adressenübersetzung in „Kacheln“



- **Paging** übersetzt “Kacheln” der logischen Adressen auf “Kacheln” der physischen Adressen
- Der Kern ist eine effiziente Funktion $F_{pid}(p) = f$, die Seitenindex **p** auf Rahmenindex **f** abbildet

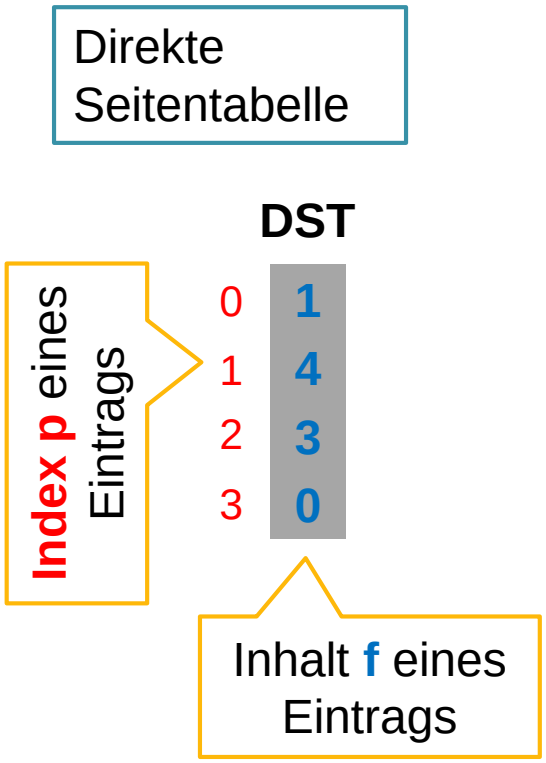
pid = Prozess ID des aktuellen Prozesses

Adressübersetzung mit $F_{pid}(p) = f$



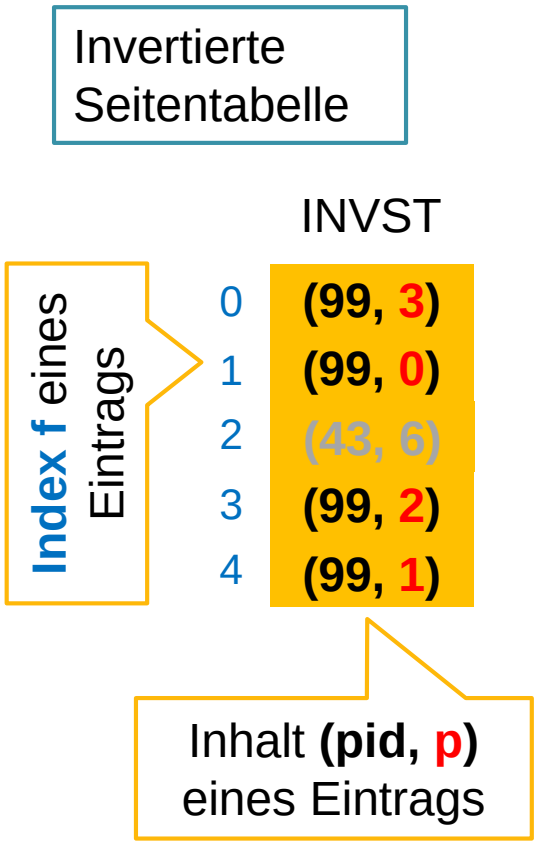
- Nachdem $F_{pid}(p) = f$ berechnet ist, ist die Übersetzung sehr einfach: obere Bits (= Wert **p**) einer logischen Adresse werden durch Bits mit Wert **f** ersetzt

Direkte (d.h. “Normale”) Seitentabellen



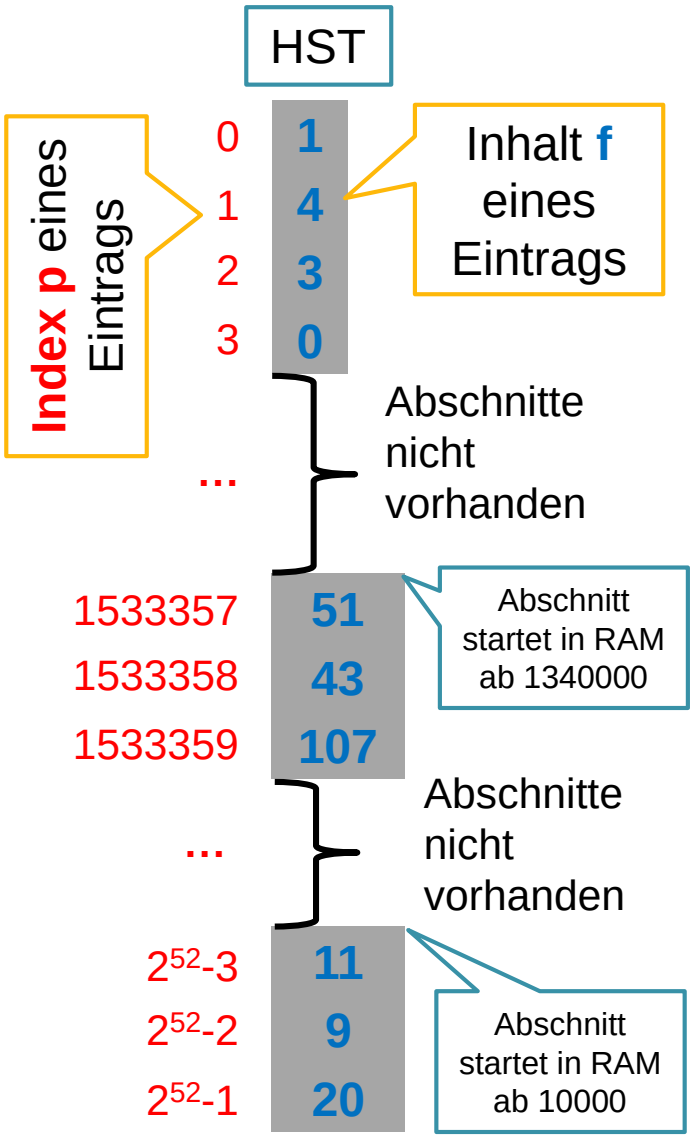
- Wie berechnet man $F_{pid}()$ mit einer direkten Seitentabelle?
- $F_{pid}()$ ist implementiert als ein Tabellen-**look-up**: sehr schnell
- In **DST**, Eintrag mit Index **p** enthält Wert **f** mit: $\mathbf{F_{pid}(p) = f}$
  - Index **p** = **Seiten**nummer
  - Inhalt **DST[p]** = **Rahmen**nummer
- Index **p** wird in DST nicht gespeichert, das wäre redundant
- Jeder Prozess (identifiziert via pid) braucht eine eigene Tabelle

Invertierte Seitentabellen



- In **INVST,** Eintrag mit Index **f** enthält Wert **(**pid,**p)** mit: $\mathbf{F_{pid}(p) = f}$
  - Index **f** = **Rahmen**nummer
  - Inhalt **INVST[f]** = (Prozess-ID **pid**, **Seiten**nummer **p**)
- D.h. um $F_{pid}()$ zu berechnen, muss man suchen: ggf. aufwändig
- Es gibt eine einzige Tabelle für alle Prozesse

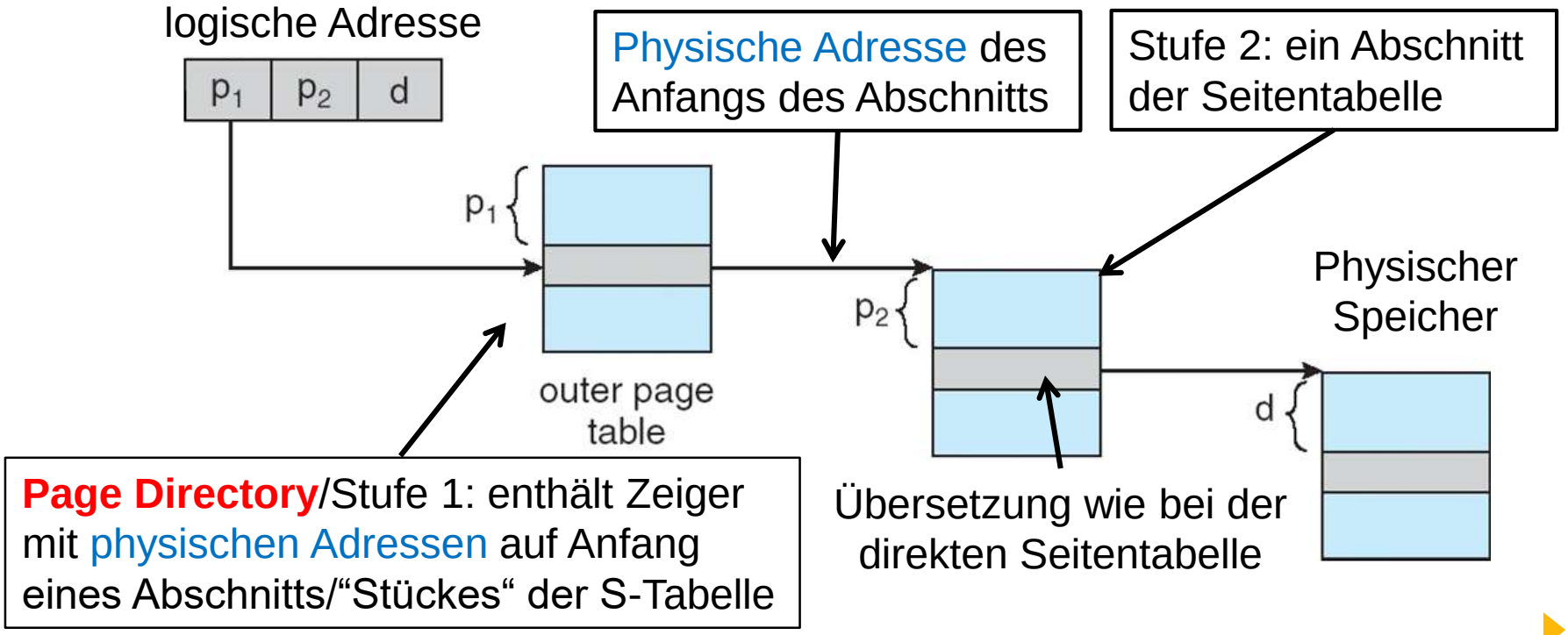
Hierarchische Seitentabellen



- Eine hierarchische Tabelle **HST** benutzt look-up (wie eine direkte Tabelle), d.h.: $\mathbf{F_{pid}(p) = HST_{pid}[p]}$
- Aber HST ist zerteilt in Abschnitte mit je 2^k Einträgen (z.B. k=10), und nur manche dieser Abschnitte existieren
- Jeder Abschnitt kann „irgendwo“ im Speicher anfangen
  - Es werden nur solche Abschnitte angelegt, die die tatsächlich verwendeten logischen Adressenbereiche abdecken

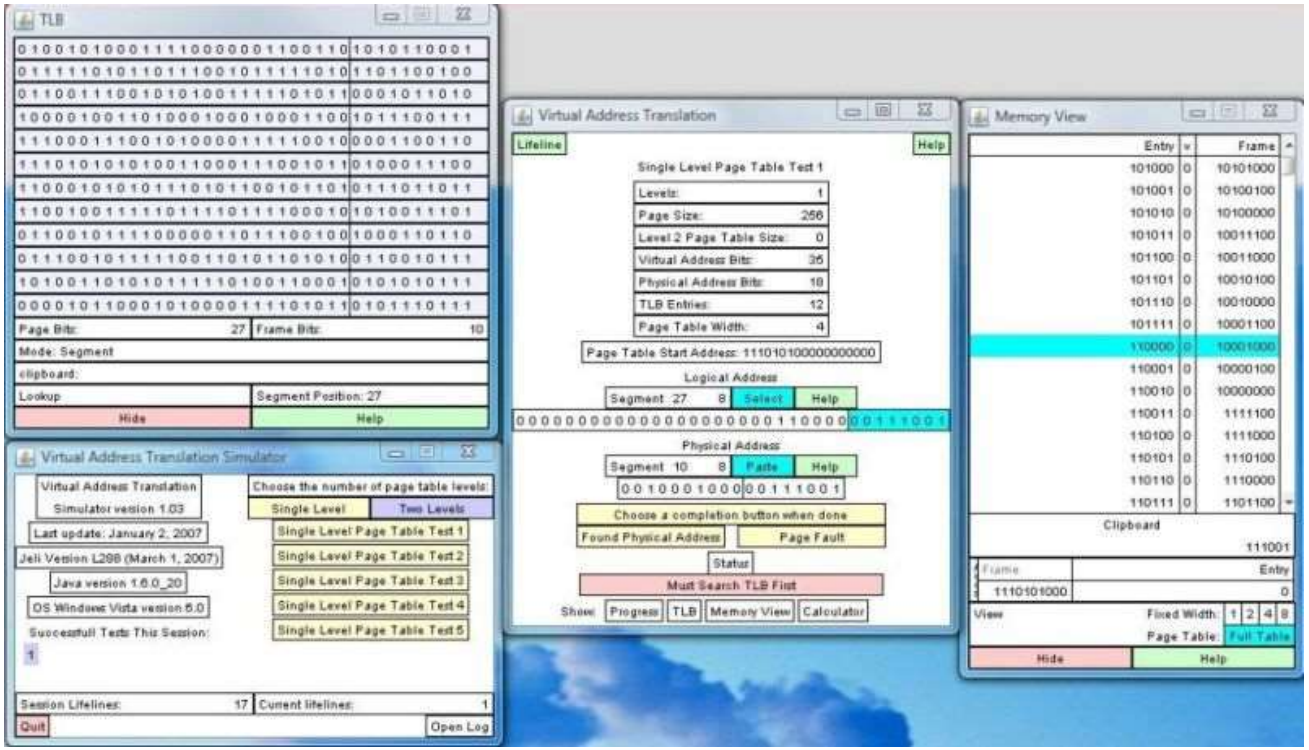
Hierarchische Seitentabellen bei IA32

- Oberste Bits (hier p_1) einer logischen Adresse indexieren die **Page Directory**, die Zeiger auf die Anfänge der einzelner Abschnitte enthält
  - Bei IA32e gibt es 3 Levels dieser Directories



Address Translation Simulator

- Simulator für Seitentabelle mit 1 oder 2 Stufen
- Download (address.zip)+ Doku:
  - https://heibox.uni-heidelberg.de/d/3f13bbd5d1/
- Originalquelle (Link kaputt):
  - http://vip.cs.utsa.edu/simulators/



Bitzählung: Nützliche Zusammenhänge /1

- Die Adressübersetzung funktioniert durch das Ersetzen (in der Adresse) der Seitennummer **p** durch eine (eigentlich beliebige) Seiten**rahmen**nummer **f**
- Die Bits des Seitenoffsets **d** werden direkt übernommen

- Sei **S** die Seitengröße (in Bytes), **N** die Busbreite (in Bits, z.B. N=32), **B**(..) die Anzahlen der Bits von p, f, d
- Was sind die Zusammenhänge zwischen S, N, B(p), B(d), B(f) usw.?

Bitzählung: Nützliche Zusammenhänge /2

- Da d die Offset-Adresse innerhalb einer Seite ist, muss $2^{B(d)}$ (=Anzahl dieser Adressen) mindestens S sein
  - => Annahme: **$S = 2^{B(d)}$**, bzw. B(d) = Zweierlogarithmus(S)
- Wie groß sind B(p) (und damit auch B(f))?
  - In einer N-Bit Adresse haben wir (neben B(d) Bits für Offset) noch genau N-B(d) Bits übrig, also **B(p) = N-B(f)**
- Wie viele Einträge hat eine vollständige direkte Seitentabelle?
  - Genau **$2^{B(p)}$** = Maximale Anzahl der Seitennummern

- [Wie viele Einträge hat eine invertierte Seitentabelle?
  - Maximalanzahl der Rahmen = **(Größe des RAMs) / S**]

Bitzählung: Beispiele

- Sei N=32 (z.B. IA32) und Seitengröße S = 4096 Bytes
  - Was sind B(d), B(p)=B(f) und die Anzahl der Einträge der direkten Seitentabelle?
  - Es ist 2^{12} = S, also B(d) = 12 (Bits)
  - Damit ist B(p) = 32 – 12 = 20 (Bits)
  - Anzahl der Einträge in Seitentabelle = $2^{B(p)} = 2^{20}$

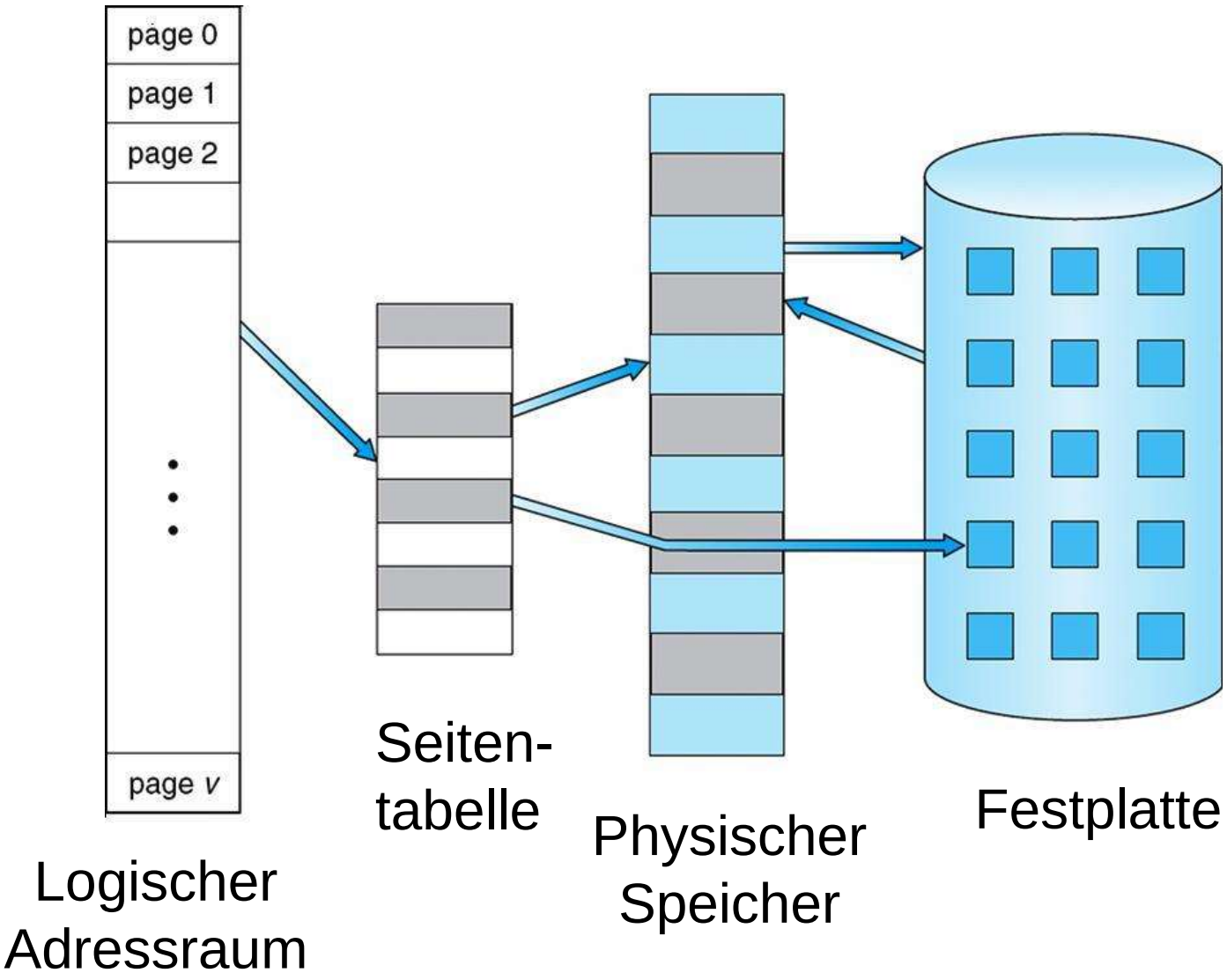
- Sei N=64, Seitengröße=1024 Bytes, RAM-Größe=1GB und eine invertierte Seitentabelle
  - Was ist B(d), B(p), die Anzahl der Einträge der Seitentab.?
  - => B(d) = $\log_2(1024)$ = 10, B(p) = 64-10 = 54
  - Wir haben als Anzahl der Rahmen: 1 GB / 1 kB = 2^{20}, also genauso viele Einträge der Seitentabelle

Virtueller Speicher

Virtuelle Speicherverwaltung

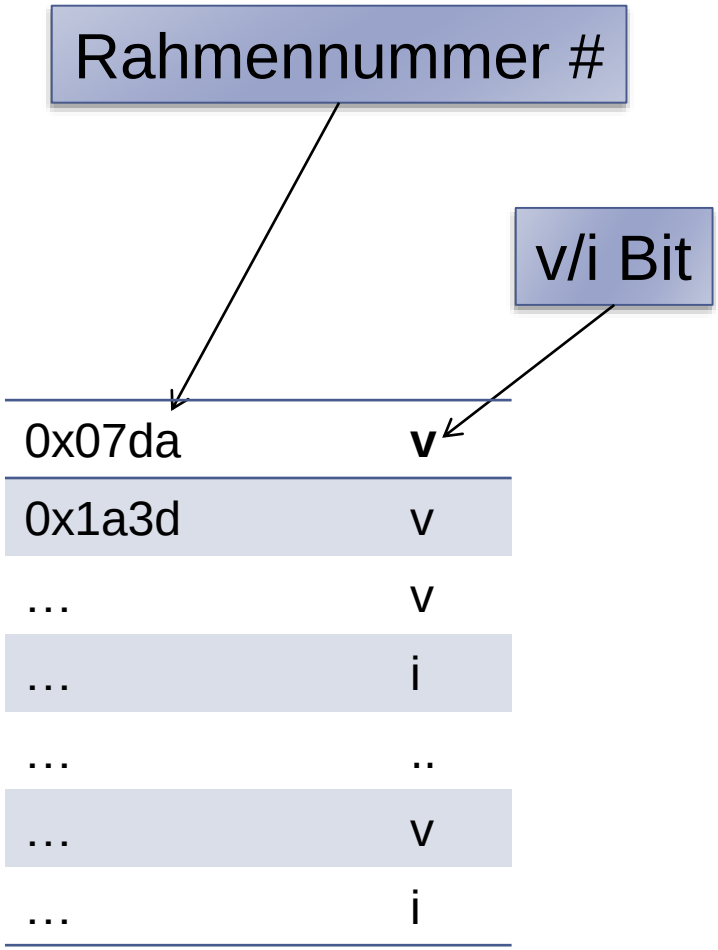
- **Virtuelle Speicherverwaltung** (**virtual memory management**) bzw. virtueller Speicher (**VS**) ist …
- Verbindung von Paging **und** dem Auslagern einzelner Seiten (bzw. Seitenrahmen) auf die Festplatte (**FP**)
  - In manchen Texten wird der Begriff Paging für die virtuelle Speicherverwaltung verwendet – wir trennen beide!
- D.h. wir erzeugen "virtuelle Rahmen": Speicher-Kacheln, die nicht (immer) im RAM liegen, sondern ggf. auf der FP
- Vorteile?
- Nur ein Teil eines Prozesses muss im Speicher sein
- Erhöhter Grad des Multiprogrammings: mehr Prozesse können zugleich ausgeführt werden
- Erlaubt eine effizientere Prozesserzeugung (später)

Prinzip des Virtuellen Speichers



Hardware-Unterstützung für VS /1

- Zu jedem Eintrag in einer Seitentabelle speichert man das **„gültig - ungültig“ Bit**
  - **v**alid - **i**nvalid Bit, **v**/**i**-Bit
  - Bzw. present – absent Bit
- Dieses Bit hat die Bedeutung:
  - **v** (= valid): Seite ist im Speicher
  - **i** (= invalid): die Seite liegt nicht im Speicher

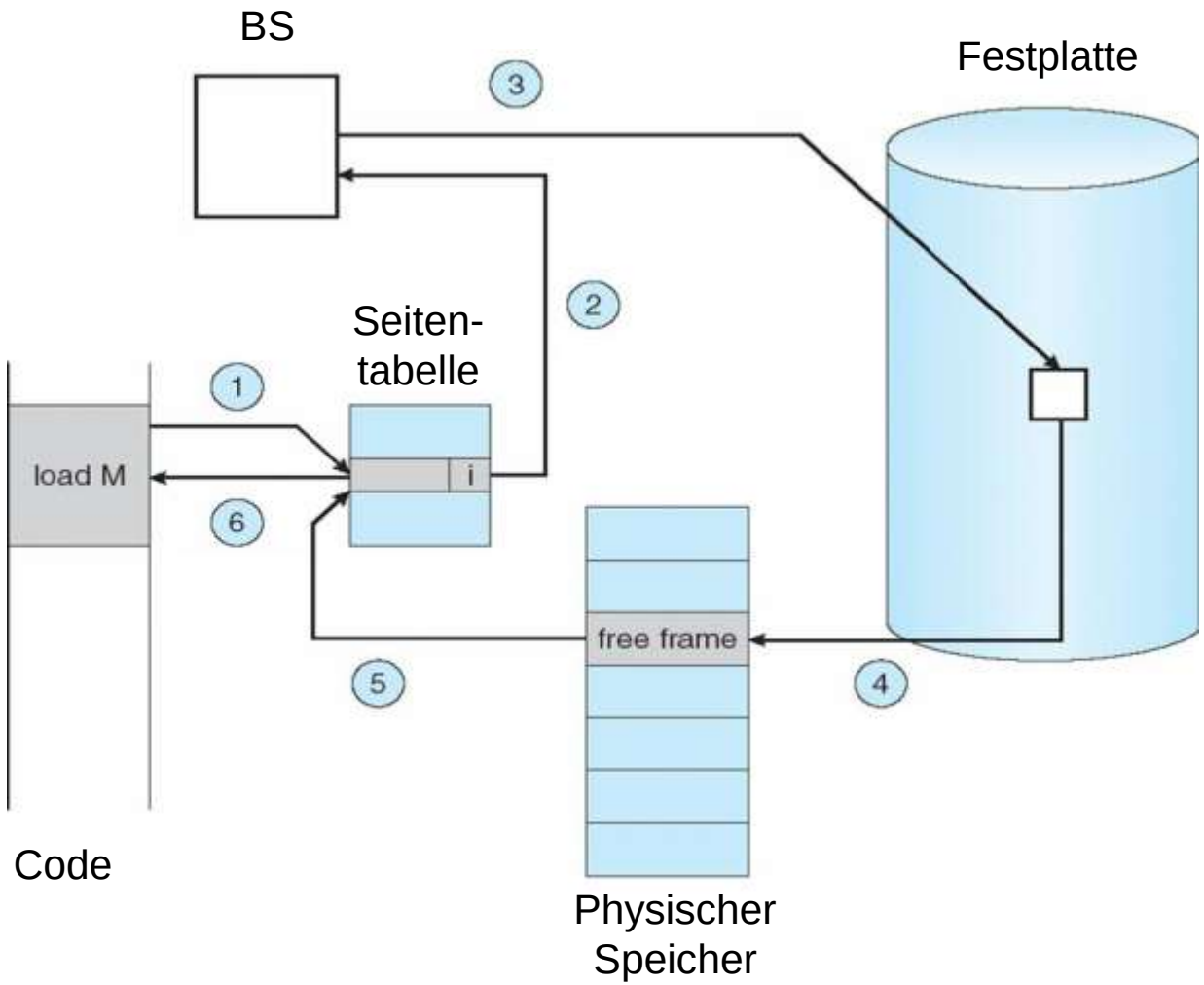


Seitentabelle

Hardware Unterstützung für VS /2

- Bei dem Zugriff auf eine Seite wird zunächst geprüft:
- Ist v/i-Bit == valid?
  - Dann normale Adressübersetzung wie bei Paging
- Ist v/i-Bit == invalid?
  - Dann hat man einen sog. **Seitenfehler** (**page fault**)

- Das führt zu einer **Seitenfehler-Behandlung**
  - Es wird eine Ausnahmebehandlung eingeleitet, um die Seite aus der Festplatte ins RAM zu holen
  - Danach erfolgt die normale Adressübersetzung



Behandlung von Seitenfehlern

1. Speicher-referenz
2. page fault = Seitenfehler
3. Seite ist ausgelagert
4. Hole den Inhalt der Seite in den Speicher
5. Aktualisiere Seitentabelle und Hilfstabelle
6. Starte den CPU-Befehl neu

Behandlung von Seitenfehlern

Ein i-Bit erzeugt einen **Seitenfehler** (**page fault**) …

- 1. BS schaut in einer Hilfstabelle **T** nach
  - Ist das eine ungültige Speicherreferenz?
    - Ja, leite Fehlerbehandlung ein (Abbruch der Adressübersetzung)
  - Nein, alles OK, nur die Seite ist ausgelagert auf die FP
- 2. BS ermittelt einen freien Seitenrahmen f
- 3. Die fehlende Seite wird von der Disk „in“ f gebracht
- 4. BS aktualisiert die Seitentabelle und Hilfstabelle T
- 5. BS setzt den v/i-Bit der fehlenden Seite auf v
- 6. BS startet den CPU-Befehl neu, der den Seitenfehler ausgelöst hat

Informationen über Ausgelagerte Seiten

- Bei einem Seitenfehler schaut das BS in einer Hilfstabelle T nach – was kann diese enthalten?
- T enthält die Informationen, wo sich die ausgelagerte Seite mit dem (logischem) Seitenindex p eines Prozesses pid (pid, p) auf der Disk befindet
- Als Datenstruktur für T eignet sich am besten eine **Hash-Map** (**Dictionary**): (pid, p) -> (Diskadresse)
- Man kann hier auch den „ungenutzten“ Eintrag (Index p) der normalen Seitentabelle nehmen
  - „Ungenutzt“ – da die Seitenrahmennummer bei v/i == i sowieso ungültig ist (bei dem Eintrag mit Index p)
- Für einer invertierten Seitentabelle ist das so nicht möglich

Leistung /1

- Sei **p** die **page fault** (PF) **Rate**, d.h. Anteil der Seitenfehler pro Speicherzugriff, $0 \leq p \leq 1$
  - p = 0: keine Seitenfehler
  - p = 1: jeder Speicherzugriff wäre ein Fehler
- Die **effektive Zugriffszeit** (**effective access time** (**EAT**)) ist dann: EAT =
- (1 – ***p***) * (Dauer des Speicherzugriffs)
- + ***p*** * (

  Dauer der Fehlerbehandlung (d.h. Interrupt-Behandlung)

  + Zeit, um eine Seite auszulagern

  + Zeit, um die gesuchte Seite zu holen

  + Zeit, die Tabellen zu aktualisieren

  + Overhead, den Befehl neu zu starten)

Leistung /2

- Es sei die Dauer des Speicherzugriffs = 200 ns
- Es sei die durchschnittliche Dauer, einen Seitenfehler komplett zu behandeln: 8 ms
- Wenn 1 von 1000 Speicherzugriffen einen Fehler verursacht, was ist die EAT?

- EAT = (1–p)*(200 ns) + p*(8 ms)
  - = (1 – p)*200 + p*8000000
  - = 200 + p*7999800
- Bei p = 1/1000 erhält man EAT = 8199.8 ns
- Faktor 41 langsamer als normaler Speicherzugriff!

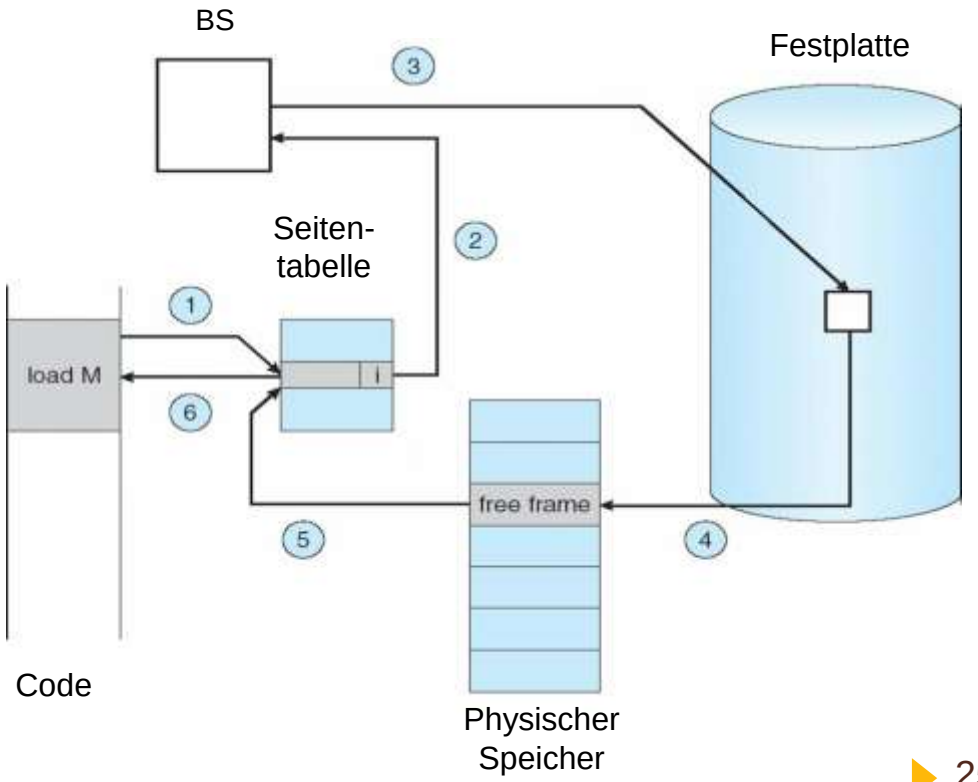
Video

- Page faults
  - **MIT 6.004 L17: Virtual Memory**
  - https://www.youtube.com/watch?v=3akTtCu_F_k
  - Ab 19:30 bis ca. 21:15 (min:sec)

Seitenersetzungsstrategien

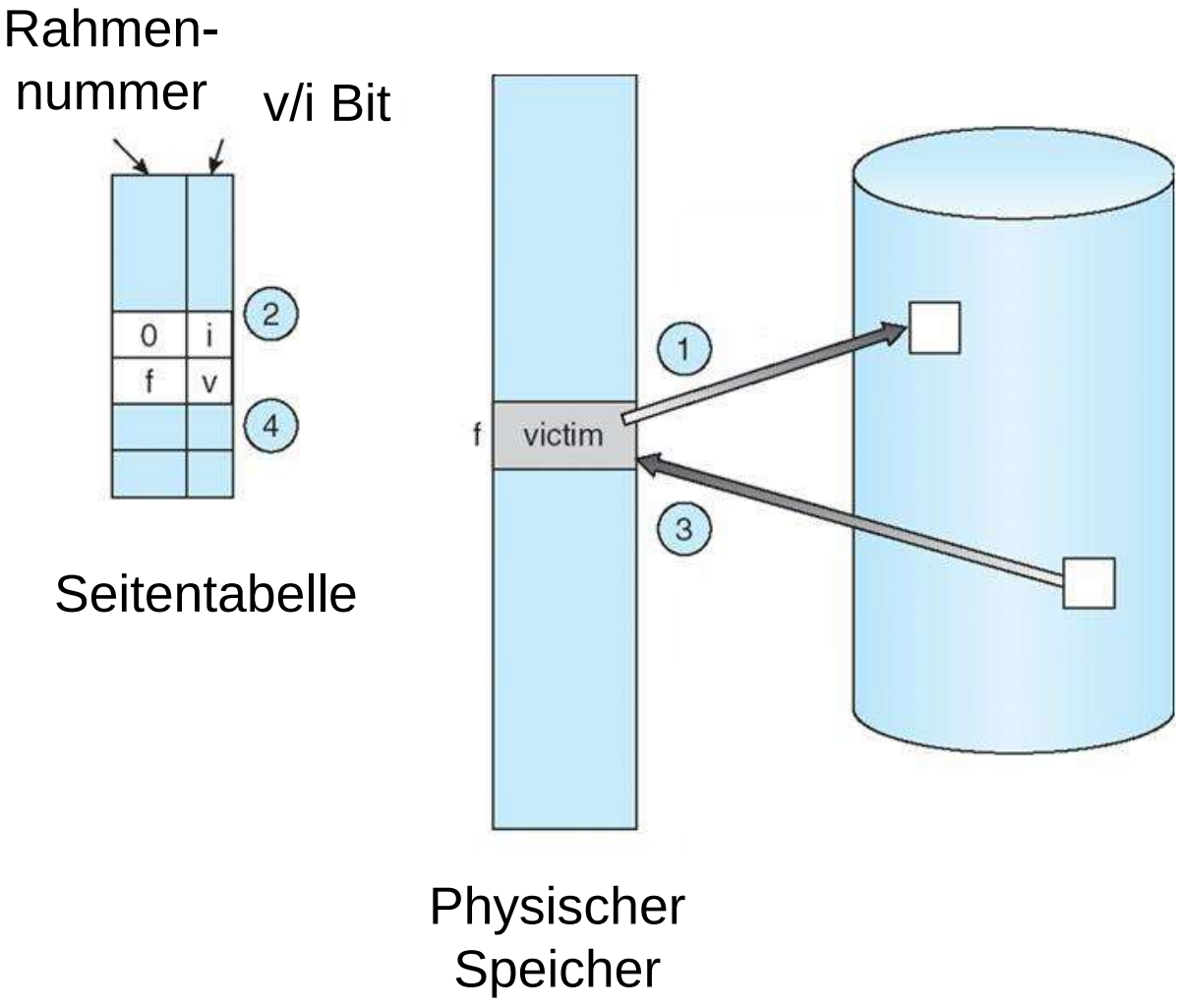
Was tun, wenn kein Seitenrahmen frei ist?

- Man betreibt die **Seitenersetzung**
  1. Finde einen Seitenrahmen **f**, der im Speicher ist, aber nicht viel gebraucht wird - das „**Opfer**“ (**victim**)
  2. Kopiere alten Inhalt von **f** auf die Festplatte und ersetze durch neuen
- Was passiert dabei genau mit der Seitentabelle?
- Welche Vorgänge müssen „atomar“ ablaufen?



Seitenersetzung

Rote Klammer
= „atomar“



1. Lagere das Opfer auf die Festplatte aus
2. Ändere v/i-Bit zu „invalid“
3. Bringe die benötigte Seite in den Speicher
4. Aktualisiere den Eintrag für die benötigte Seite in der Seitentabelle (d.h. schreibe Rahmennr. und setze auf „valid“)
5. Starte den CPU-Befehl neu (nicht gezeigt)

Gesucht: **Seitenersetzungsalgorithmen**

- Man optimiert Algorithmen nach dem Kriterium: **Minimiere die Seitenfehler-Rate**
  - **Seitenfehler-Rate** = (# page faults)**/**(# Speicherzugriffe)
  - Diese hat einen sehr großen Einfluss auf die durchschnittliche Zeit des Speicherzugriffs!
- Man vergleicht die Algorithmen anhand von fiktiven oder echten Folgen von Speicher- bzw. Seitenzugriffen
- Ein Beispiel-Log von Speicherzugriffen
  - (0x)0100, 0432, 0101, 0612, 0102, 0103, 0104, 0601, …

Bewertung der Algorithmen

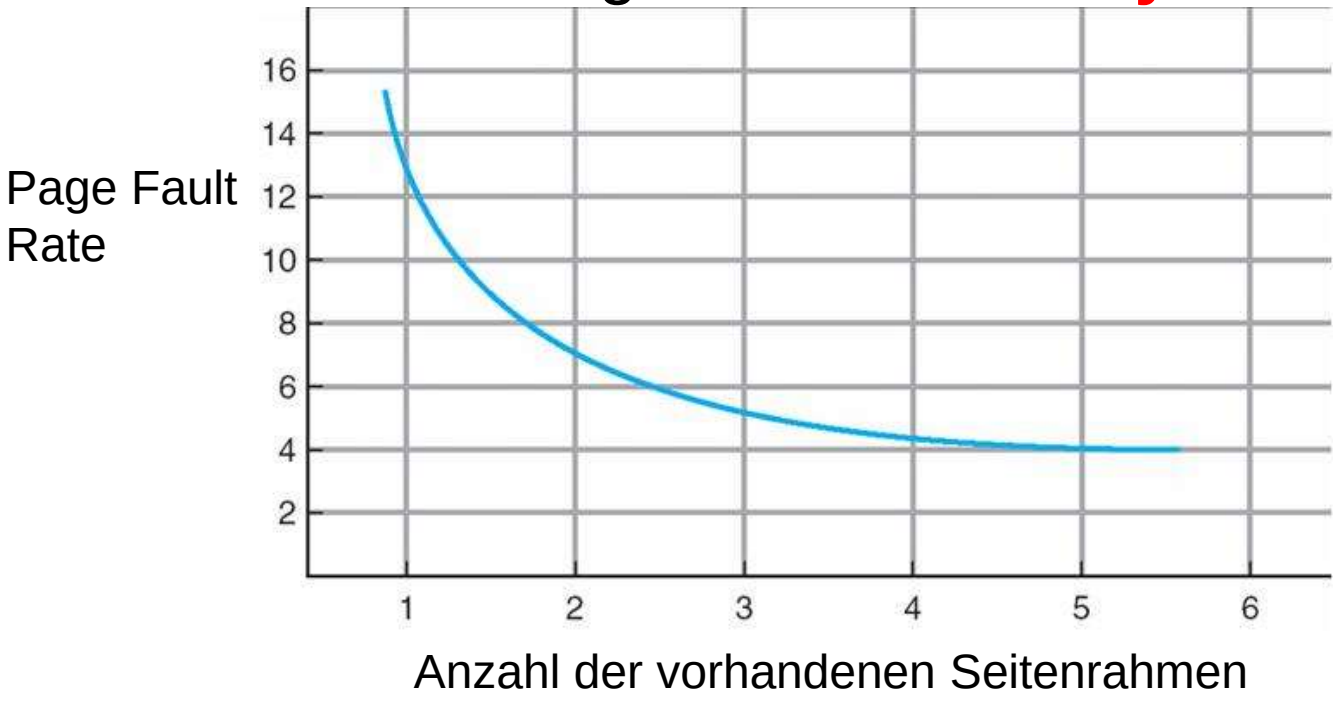
- Problem: Logs der Zugriffe können sehr groß werden
  - Wie kann sie reduzieren?
- Was müssen wir uns aus dem folgenden Log wirklich merken?
- Seitengröße = 0x100 (Hexadezimal)
- (0x)0100, 0432, 0101, 0612, 0102, 0103, 0104, 0601, …

1. Es werden nur die Seitennummern aufgezeichnet
2. Man braucht keine aufeinanderfolgende Seitennummern, da keine Seitenfehler möglich

=> 1, 4, 1, 6, 1, 6

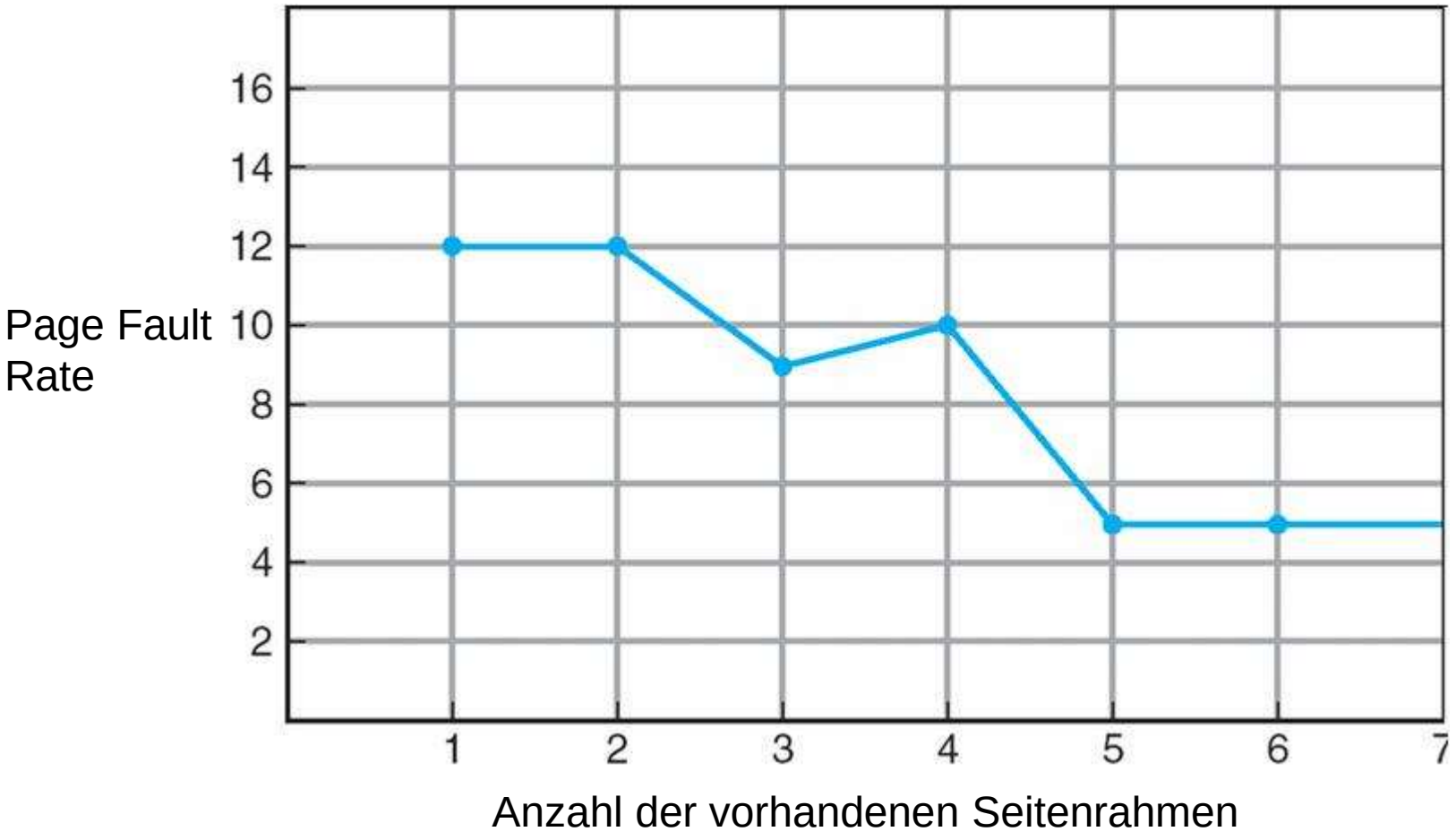
Abhängigkeiten der Page Fault Rate

- Selbstverständlich hängt die Seitenfehlerrate von der Anzahl der vorhandenen Seitenrahmen – wie?
- Mehr Seitenrahmen (RAM) => weniger Seitenfehler
- Aber: Bei schlechten Algorithmen kann sie sogar mit mehr Rahmen steigen => die **Belady-Anomalie**



Optimaler Seitenersetzungsalgorithmus

- Gibt es hier die Belady-Anomalie? Wo?



Seitenersetzungsalgorithmen: OPT, FIFO und FIFO-Verbesserungen

Seitenersetzungsalgorithmen

- Die Forschung ist inzwischen nicht mehr aktiv, kaum Verbesserungen möglich
- Aber: Diese Algorithmen sind überall wichtig, wo man Caches benutzt
  - Z.B. TLB, Netzwerke, …
- Wir lernen jetzt einige Seitenersetzungsalg. kennen
  - Der Optimale Algorithmus
  - FIFO
  - Second-Chance-Algorithmus
  - Clock-Algorithmus
  - Least-Recently-Used-Algorithmus
  - Working-Set-Algorithmus

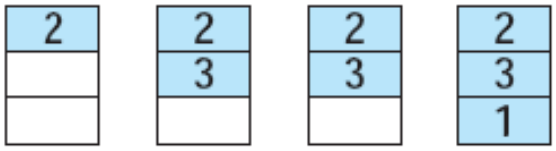
Seitenersetzungsalgorithmen

- Zum Illustrieren der Verfahren werden wir Referenzfolgen von Seitenzugriffen benutzen
  - Erinnerung: Nur bei Wechsel einer Seite in der Zugriffsfolge könnte ein Seitenfehler auftreten
- Wir benutzen zwei Referenzfolgen:
  - **A**: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
  - **B**: 2,3,2,1,5,2,4,5,3,2,5,2
- Es stehen immer 3 Seitenrahmen (= 3x eine Seite RAM) zur Verfügung

Der Optimale Algorithmus

- Gibt es einen Algorithmus, der garantiert die kleinste Rate der Seitenfehler hat
  - … und nicht unter der Belady-Anomalie leidet?
- Ja: der **Optimale Seitenersetzungsalgorithmus OPT** oder **MIN**:
  - „Ersetze eine Seite, die die längste Zeit in der Zukunft nicht verwendet wird“
- Einfach, genial, …
  - … nutzlos*: benötigt perfektes Wissen über künftige Ereignisse

2	3	2	1	5	2	4	5	3	2	5	2

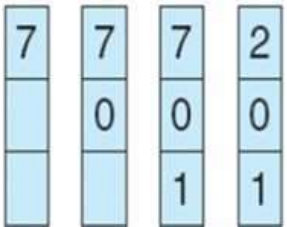


FIFO (**First-In, First-Out**)-Algorithmus

- Die „älteste“ Seite im RAM wird zum Opfer
  - Für jede Seite wird die Zeit notiert, wann sie in den Speicher geladen wurde
- Optimierung: Wir brauchen nur die Reihenfolge des Ladens in RAM, nicht die Zeit
  - Implementiert als eine **FIFO-Schlange**: Neueste Seite wird ans Ende angehängt, die älteste vom Anfang entfernt

• Problem: die *Relevanz* der Seiten wird nicht berücksichtigt (z.B. häufig benötigte Seiten können auch sehr früh in den Speicher geholt worden sein)

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



M-Bit und R-Bit der Seiten

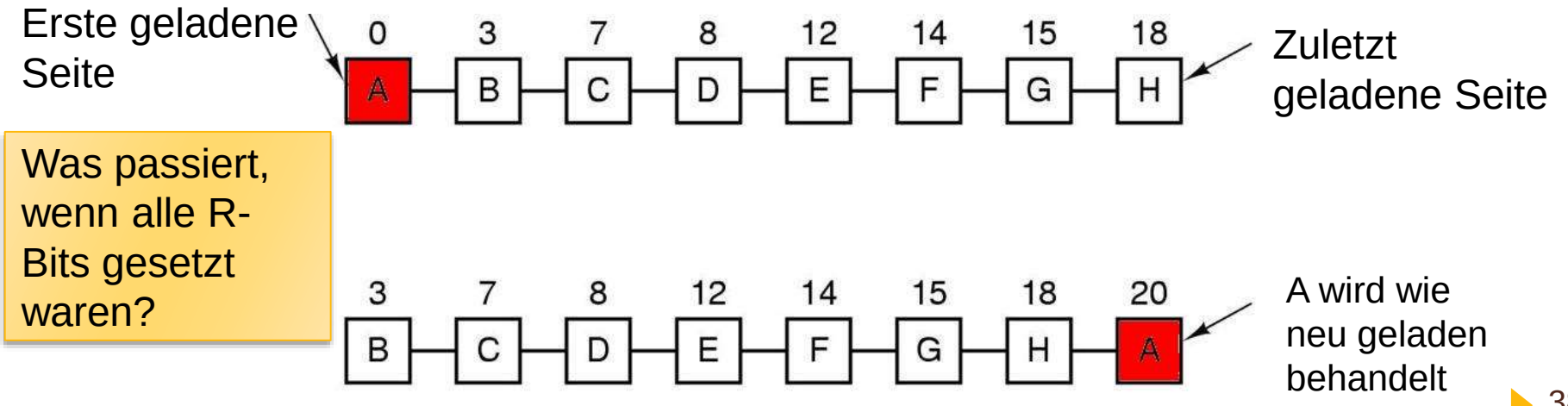
- Jeder Eintrag der Seitentabelle (direkte oder hierarchische) hat u.a. ein **M-Bit** und ein **R-Bit**
- Der **M-Bit**, bzw. **Modify Bit** wird automatisch bei einem Schreibzugriff auf die Seite gesetzt
- Der **R-Bit** (**Referenz-Bit**) wird gesetzt, wenn eine Seite gelesen wurde
- Diese Bits werden automatisch durch die Hardware gesetzt
  - Müssen aber durch Software gelöscht werden

Optimierungen durch die M-Bits und R-Bits

- Wie könnten wir den M-Bit/R-Bit nutzen, um die Seitenersetzung zu optimieren?
- Optimierung mit dem **Modify** („**dirty**“) **Bit**
  - Wenn eine Seite nicht modifiziert wurde, kann sie einfach überschrieben werden!
  - Annahme: Anfangs waren alle Seiten schon auf der Festplatte (Voraussetzung des Demand Pagings)
- Optimierung mit dem **Modify** („**dirty**“) **Bit**
  - Hilfreich für die Seitenersetzungsalgorithmen

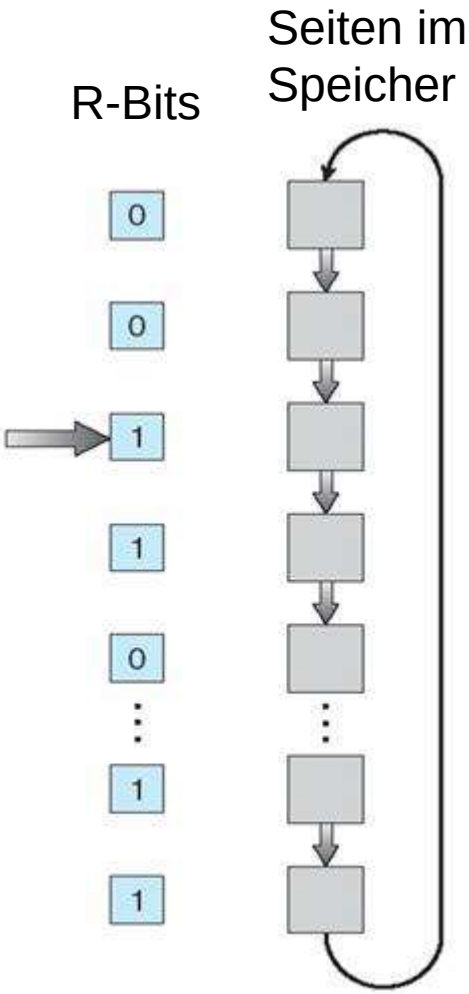
Second-Chance-Algorithmus

- Zunächst funktioniert dieser wie FIFO
- Wenn die „älteste“ Seite A gefunden wurde, wird aber zunächst geprüft, ob ihr R-Bit gesetzt ist
  - R-Bit == 0: Opfer gefunden, da Seite A alt und unbenutzt
- R-Bit == 1: „**second chance**“
  - Der R-Bit der Seite wird gelöscht, aber die „Ladezeit“ von A wird so gesetzt, als ob die erst gerade geladen worden wäre
  - Dann wird weiter gesucht



Clock-Algorithmus: Bessere Datenstruktur

- Letzer Ansatz hat eine ineffiziente Datenstruktur
  - Viele Listeneinträge müssen von Listen-Anfang an das Listen-Ende verschoben werden
- Besser: **zirkuläre Liste**
  - Ein Zeiger markiert den L-Anfang
  - Bei der Suche wird nur der Zeiger „inkrementiert“, und das R-Bit gelöscht
  - Was passiert, wenn ein Opfer gefunden wurde?



Der Eintrag wird durch die Nummer der neu geladenen Seite ersetzt, Zeiger wandert um 1 Position weiter

Erweiterter Clock-Algorithmus

- Man nutzt neben dem R-Bit auch den M-Bit („Seite wurde modifiziert“)
- Hierarchie von Kombinationen (R-Bit, M-Bit)
  - (0,0): weder benutzt noch modifiziert - bestes Opfer
  - (0,1): nicht benutzt, man muss zurückschreiben
  - (1,0): neulich benutzt, man muss nicht zurückschreiben
  - (1,1): ein richtig mieses Opfer, pfui!
- Man geht die zirkuläre Liste durch, und teilt jede Seite in eine der vier obigen Klassen
- Die zunächst gefundene Seite der ersten nicht-leeren Klasse wird zum Opfer

Zusammenfassung

- Virtueller Speicher
  - Paging + Auslagern von Seiten auf die Festplatte
  - Leistung
  - Demand Paging, Copy-On-Write
- Seitenersetzungsalgorithmen
  - Kriterium K: minimiere Rate der Seitenfehler
  - Algorithmen für die Seitenersetzung
  - Trade-off: Realistisch umsetzbar vs. gut in Bezug auf R
- Quellen: Silberschatz Kap. 9; Tanenbaum Kap. 3

Zusätzliche Folien

A: LRU Datenstruktur als „Stapel“ (stack)

- Eine doppelt-verkettete Liste mit Seitennummern, die aktuell im Speicher sind
- Bei jedem Zugriff auf eine Seite mit Index p wird ihre Nr. auf das Ende der Liste („top of stack“) gebracht
- Das potentielle Opfer ist immer am Anfang der Liste
- Muss man hier (irgend etwas) suchen?
- Nein:
  - Falls wir zu jedem Seiteneintrag einen Zeiger auf „seine Nummer“ in dem Stapel aufrechterhalten
  - Weiterhin wird Zeiger auf den Anfang und Ende der Liste aktuell gehalten

A: LRU Datenstruktur als „Stapel“ /2