Betriebssysteme Vorlesung 6

Memory Management II

virtueller Speicher,

Auswirkungen auf Paging, Segmentation

Literatur: Stallings Kapitel 8

Übersicht

- Paging und Segmentierung
  - grobe Vorgehensweise
  - Adreßtransformation, i.w. hardwareunterstützt
- Strategien zur Gestaltung eines virtuellen Speichers, die in OS Software umgesetzt werden.

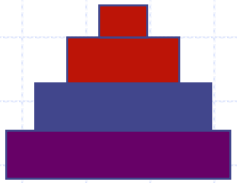
  Wir fokussieren auf Paging und betrachten
  - das Laden von Seiten
  - das Plazieren von Seiten
  - das Ersetzen von Seiten
  - das Verwalten des Resident Sets
  - das Sichern von Seiten

Nutzen

- Sie lernen mit dem virtuellen Speicher ein wesentliches Konzept aus dem Design moderner Betriebssysteme kennen.
- Sie können neben der Funktionsfähigkeit auch Rahmenbedingungen einschätzen, die für die Nützlichkeit des virtuellen Speichers wesentlich sind:
  - ausreichender Plattenplatz
  - ausreichende räumliche und zeitliche Lokalität in Programmen
- Sie lernen Stellgrößen zur Parametrierung des Speichermanagements kennen.

Wiederholung: Speicherverwaltung bei Multiprogramming

- Anforderungen an Speicherverwaltung bei n Prozessen:
  - jeder Prozeß muß ausreichend Speicher erhalten
  - Versorgung der CPU (schnellste Komponente im System!) mit Instruktionen und Daten möglichst verzögerungsfrei
  - die Größe aller Prozeßimages darf den Hauptspeicherplatz überschreiten
- dafür verfügbare Ressourcen: Speicherhierarchie



  Level 1 & Level 2 Caches, Hauptspeicher, Plattenspeicher, ...

  zunehmendes Fassungsvermögen, zunehmende Zugriffszeiten
- Lösungen
  - Paging: Aufteilen des Hauptspeichers in gleichgroße Partitionen (Seitenrahmen),

    ein Prozeß belegt eine variable Anzahl von Seitenrahmen
  - Segmentierung: Aufteilen des Hauptspeichers in variabel große Partitionen je nach Bedarf eines Prozesses

Grundlagen und Grundidee des virtuellen Speichers

- Trennung von logischem und physikalischem Adreßraum
- Prozeßimage enthält logische Adressen
  - erfordert Umrechung in physikalische Adressen zur Laufzeit
    - mit Seiten / Segmenttabellen
    - mittels Aneinanderhängen / Addition
    - mit Bereichsbegrenzung (Schutzfunktion)
  - erlaubt beliebige Position des Images im Hauptspeicher
- Paging/Segmentierung stellt Prozeß durch Menge von Speicherbereichen dar.

Virtueller Speicher
Nur ein Teil der Seiten/Segmente eines Prozesse belegen Platz im Hauptspeicher, fehlende Bereiche liegen auf der Festplatte (im Swap Space). Bereiche werden ggfs geladen, bei Platzmangel im Hauptspeicher auch im Wechsel.

Vorteile des virtuellen Speichers

- Prozessimages dürfen größer als der Hauptspeicher sein
  - aber nicht größer als der Festplattenplatz
- größere Anzahl von Prozessen „simultan“ bearbeitbar
  - es werden von vielen Prozessen nur relativ kleine Teile geladen
  - viele bearbeitbare Prozesse erhöhen die CPU Auslastung
- Nachteile ?
  - Verwaltungsaufwand, zusätzl. Prozeßwechsel, Adressierung

  Woher resultieren zusätzliche Prozeßwechsel ???

im folgenden betrachten wir im wesentlichen Paging

zusätzliche Begriffe

- resident set: Menge der Seiten eines Prozesses, die im Hauptspeicher residieren/verfügbar sind
- working set: Menge der Seiten eines Prozesses, die über einen Zeitraum zur Bearbeitung benötigt werden

Prozeßbearbeitung durch OS bei Paging

1. OS lädt einige Seiten des Prozesses in den Hauptspeicher (PCB bleibt stets präsent)
2. Prozeß wird bearbeitet, bis bei der Adreßtransformation auf eine Seite zugegriffen wird, die sich nicht im Hauptspeicher befindet (page fault).
3. Page fault bewirkt Interrupt, OS versetzt den Prozeß in den Zustand „blockiert“
4. Seite mit referenzierter Adresse wird geladen
   - OS veranlaßt I/O Leseanforderung
   - Dispatcher bewirkt Prozeßwechsel und zwischenzeitliche Bearbeitung von anderen, wartenden Prozessen (ready)
   - Wenn die Seite geladen wurde (zB durch DMA), signalisiert ein Interrupt der CPU, daß der Prozeß weiterbearbeitet werden kann. Der Prozeß wechselt in den Ready Zustand.

Wann kann virtueller Speicher funktionieren?

- Klar: viele Prozesse, große Prozesse, ...
- Aber auch: wenig Page Faults !

-> Hoffnung: Referenzen innerhalb einer Berechnung zeigen Lokalität

  - räumlich: Referenzen aufeinanderfolgender Instruktionen liegen eng beieinander, idealerweise auf derselben Seite
  - zeitlich: Prozeßbearbeitung weist Phasen aus, in denen bestimmte Seiten häufig genutzt werden. Idealerweise sind diese Phasen relativ lang und verändern sich inkrementell.

- Strategien zur Festlegung der Anzahl von Seiten und zur Auswahl von Seiten für den Resident Set basieren auf einer Lokalitätsannahme.

- In welchen Situation funktioniert das Konzept nicht ?

Die Katastrophe: Thrashing

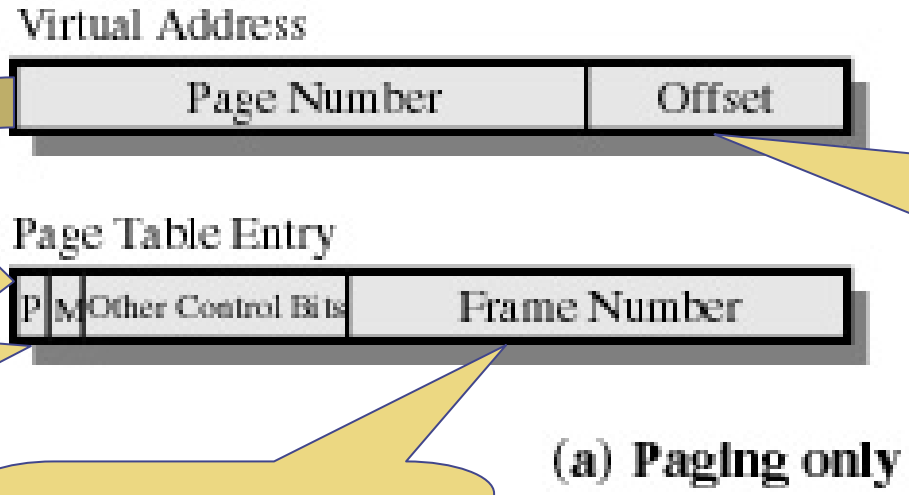
- Szenario I: es werden so viele Prozesse bearbeitet, dass
  - jeder einzelne zuwenige Seiten verfügbar hat und bei Bearbeitung fast unmittelbar auf fehlende Seiten zugreift und blockiert
  - während der Seitenanforderung wird ein anderer Prozeß bearbeitet, dem es genauso ergeht
  - neu geladene Seiten verdrängen andere, die jedoch ebenfalls in naher Zukunft angefordert werden
- Szenario II: ein Prozeß bearbeitet extrem große Datenstrukturen, Zugriffe streuen jedoch sehr stark
- Effekt: CPU arbeitet fast nur noch für Prozeßwechsel und Seitenwechsel -> Thrashing („Zappeln“)
  - Anzeichen: geringe CPU Auslastung, viel I/O Aktivität, wenig freier Speicherplatz (Unix: vmstat 5)
  - Achtung bei „top“ u.ä. Befehlen wird der Speicherbedarf einzelner Prozesse ohne Shared Libraries angegeben. Der Gesamtbedarf ist größer als die Summe der einzelnen Prozesse!

Design Fragen

- Wieviele Prozesse sollen bearbeitet werden ?
  - wg CPU Auslastung viele, wg Thrashing nicht zu viele
- Welche Größe soll eine Seite haben ?
  - wg Anpassung an Working Set klein, wg Verwaltung nicht zu klein
- Wieviele Rahmen soll ein Prozeß erhalten ?
  - wg Thrashing nicht zu wenige, wg Prozeßanzahl nicht zu viele
- Wann sollen Seiten geladen werden ?
  - bei Bedarf oder auf Verdacht
- Welche Seiten sollen ggfs überschrieben/ersetzt werden ?
  - Prozeßeigene/fremde Seiten, lange nicht genutzte/alte Seiten
- Wann sollen Seiten zurückgeschrieben werden ?
  - Sofort und einzeln, verzögert und in Gruppen (Batches)
- Wie werden Seiten verwaltet ?

Verwaltung von Seiten in Seitentabellen, je Eintrag:

Seitenzahl
liefert Zeile
der Seitentabelle



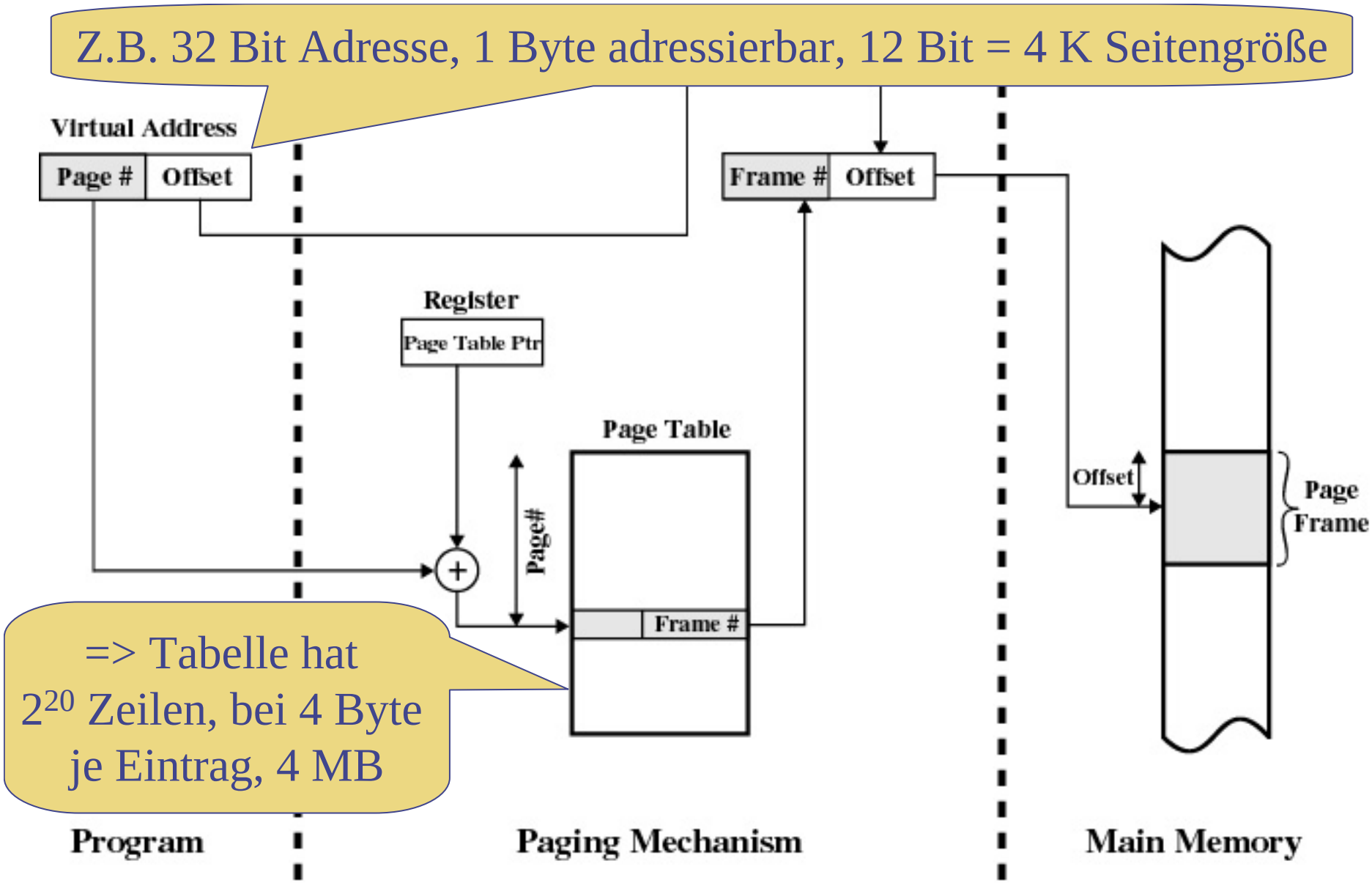
(a) Paging only

Position
innerhalb
der Seite

Kontrollbits
- Präsent
- Modifiziert
- letztens genutzt
- rw oder read-only

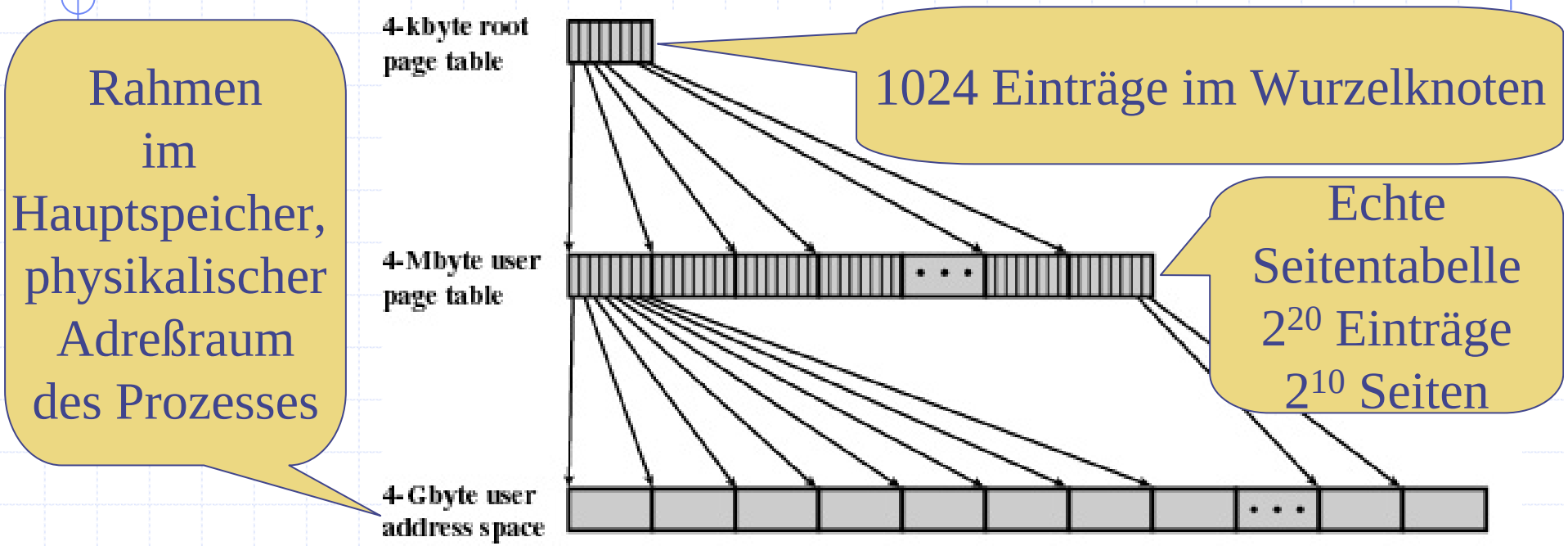
Gesuchte
Rahmennummer

- Jeder Prozeß hat eine eigene Seitentabelle mit Rahmennummern je Seite.
- Kontrollbits signalisieren, ob eine Seite verfügbar ist oder modifiziert wurde.
- Modifizierte Seiten müssen vor dem Überschreiben auf den Plattenspeicher (in den Swap Space) übertragen werden.



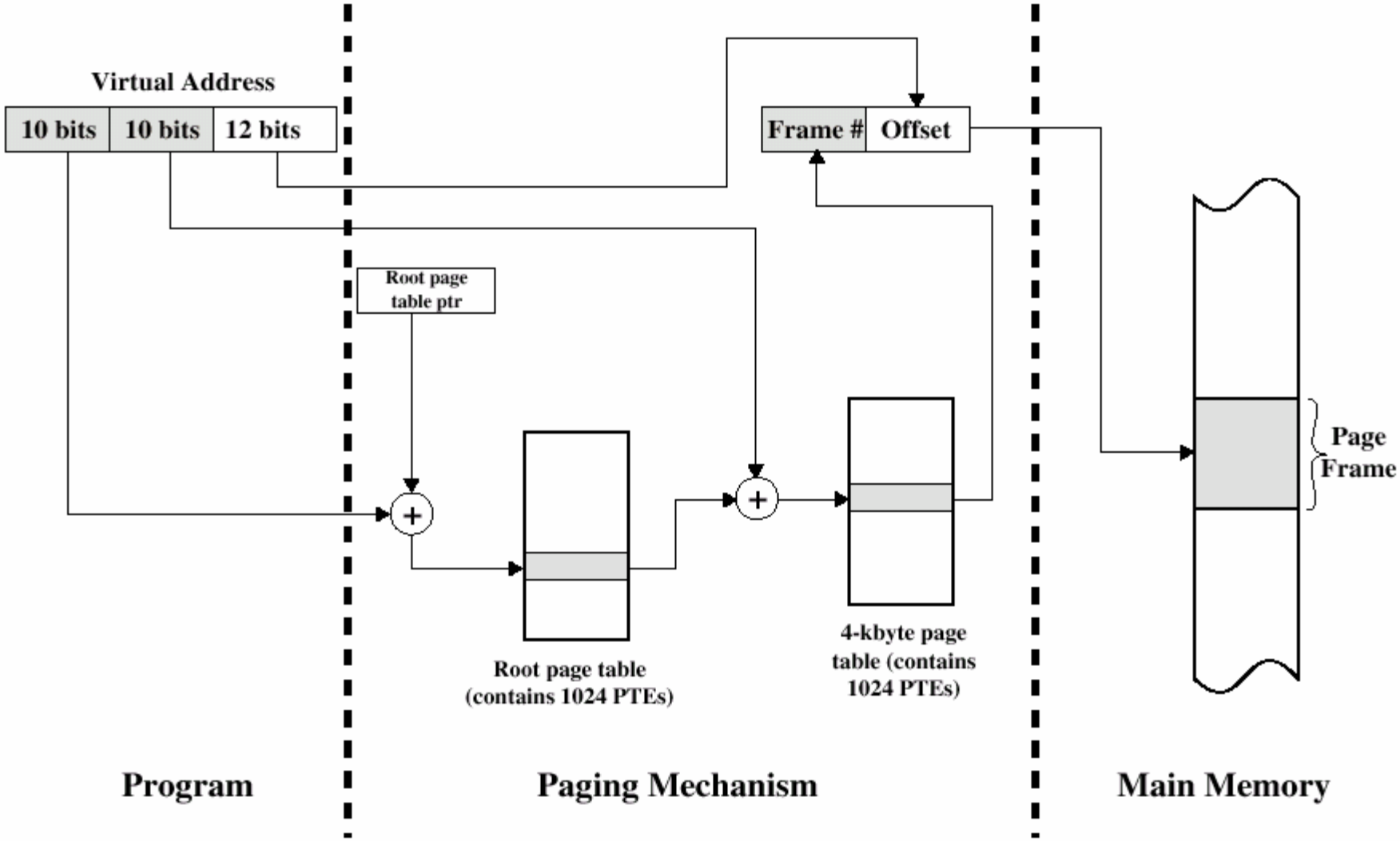
Adreßübersetzung in einem Paging System, 2 Seitenzugriffe je Adresse
wg Zeitaufwand: Hardwareunterstützung innerhalb der CPU
wg Platzbedarf, hier z.B. 4 MB je Prozeß: Seitentabelle hierarchisch

Variante 1: Schema mit 2 Ebenen für 32 Bit Adressen



Idee: Auch die Seitentabelle wird mit Paging verwaltet.
Der Wurzelknoten bleibt stets im Hauptspeicher.
Dies bewirkt 3 Speicherzugriffe je Adreßrechnung,
bei 64 Bit Adressen sogar 3-4 Ebenen,
Läßt sich damit eine akteptable Rechengeschwindigkeit erzielen ?
=> Hardwareunterstützung => Cacheing !

Adreßübersetzung bei 2-stufigem Paging



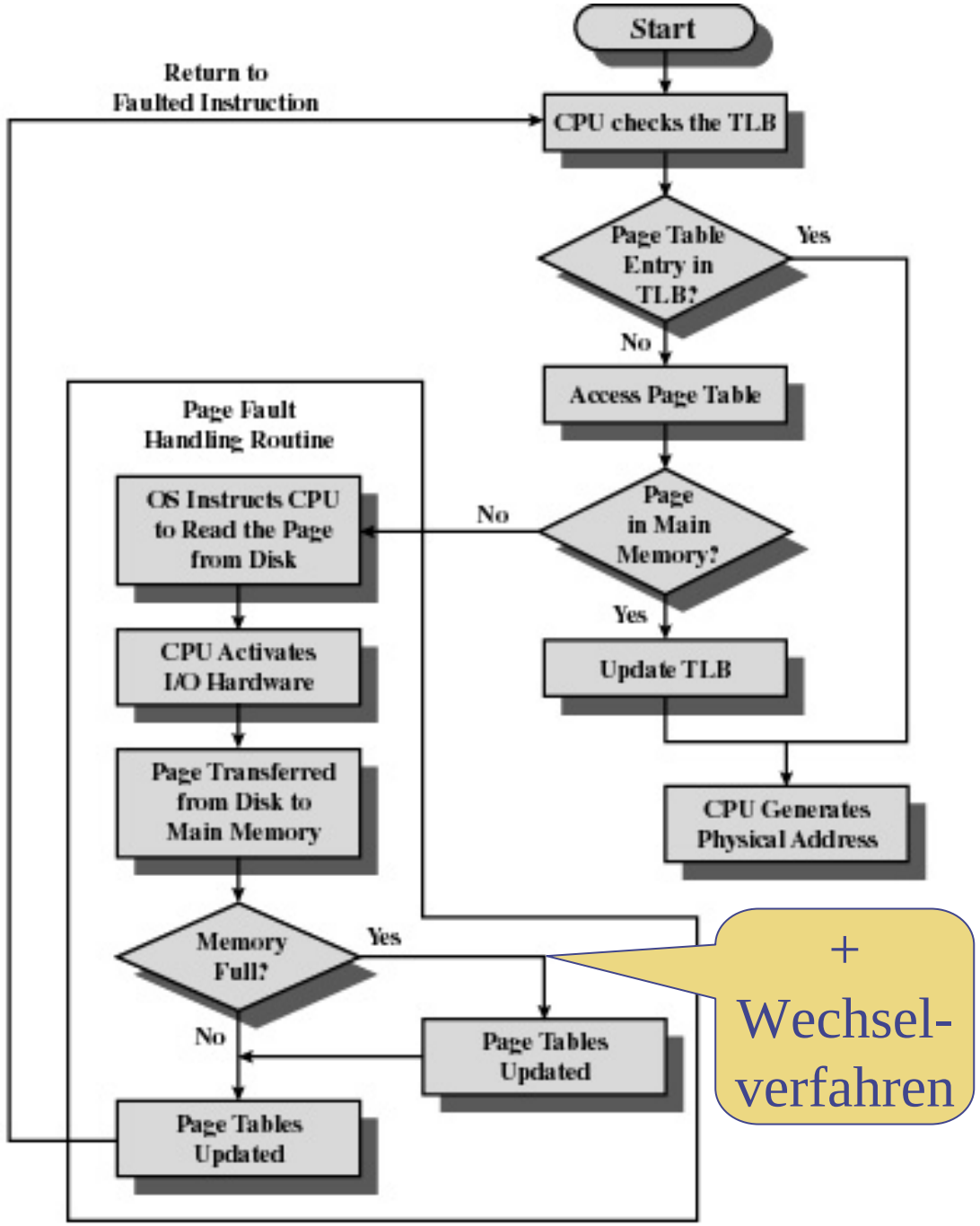
Cacheing Idee: merke Antworten auf häufige Anfragen

Hardware:

Translation Lookaside Buffer (TLB) = Assoziativspeicher speichert Zeilen der Seitentabelle als Cache.

Der TLB wird zusätzlich abgefragt, ggfs aktualisiert.

Der Ablauf muß noch um den Seitenwechselmechanismus bei vollem Speicher erweitert werden.



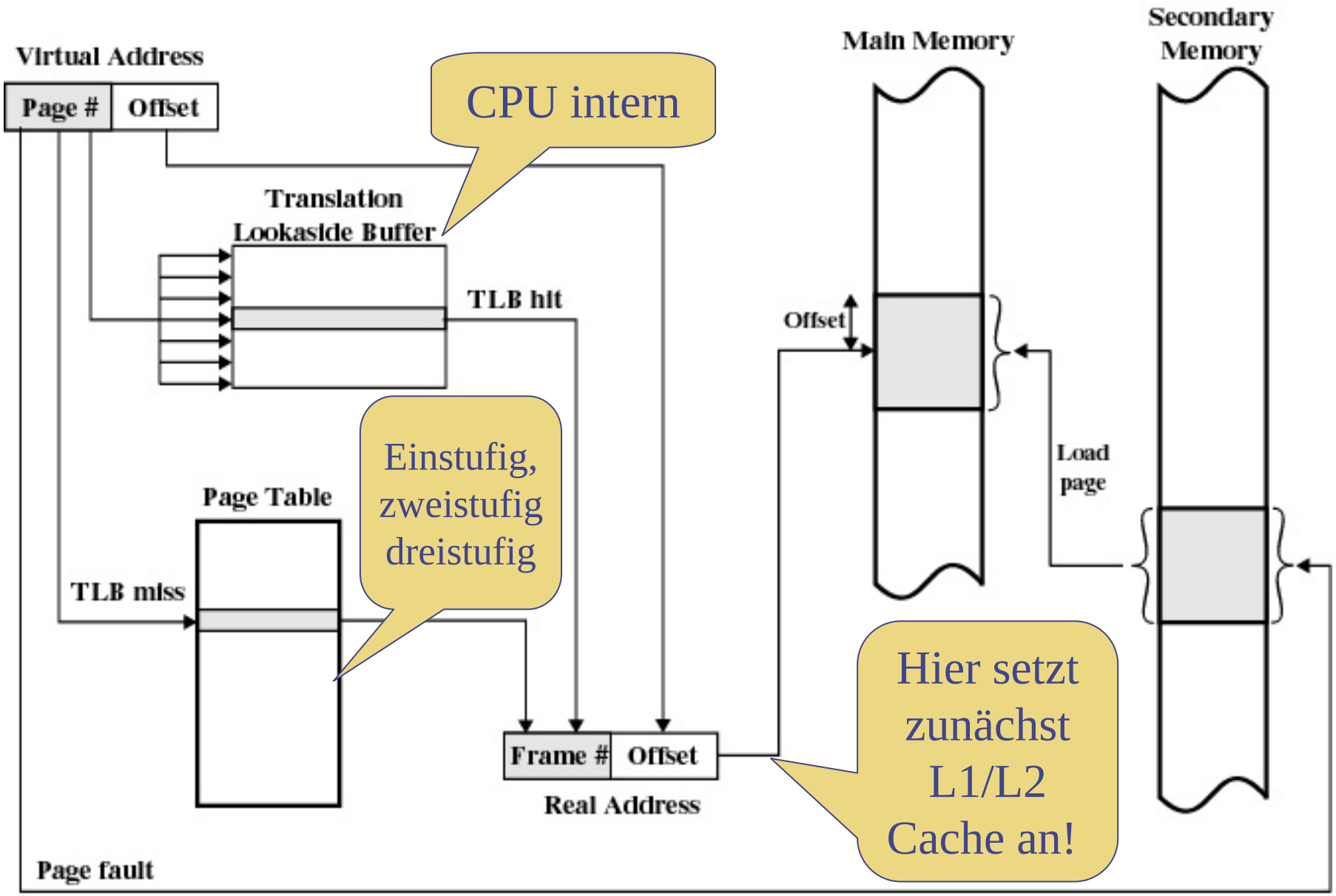


Abb. 8.7: Verwendung des TLBs
Achtung: bei Prozeßwechsel muß TLB gelöscht werden!

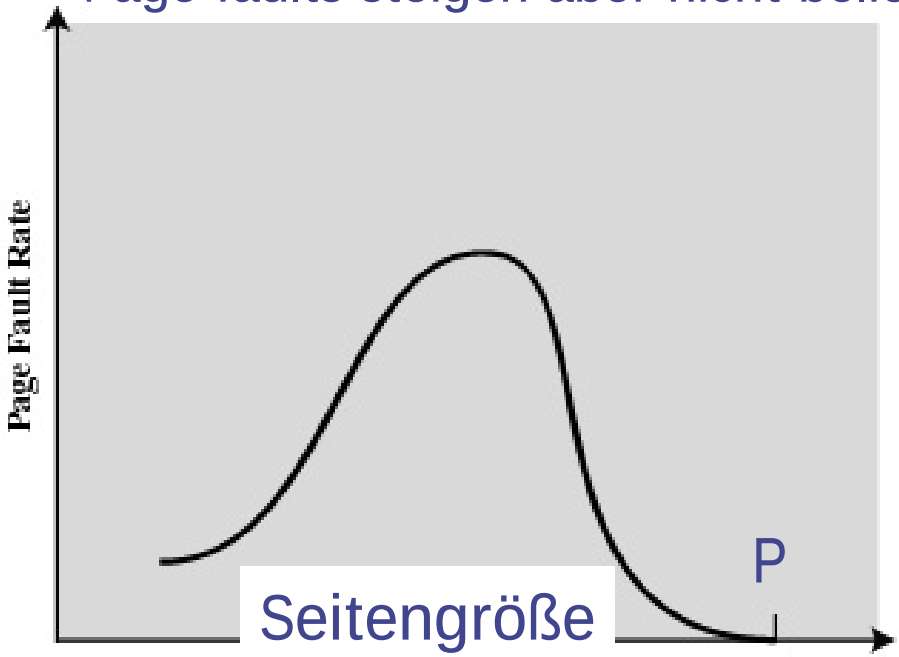
Variante 2: Invertierte Seitentabelle

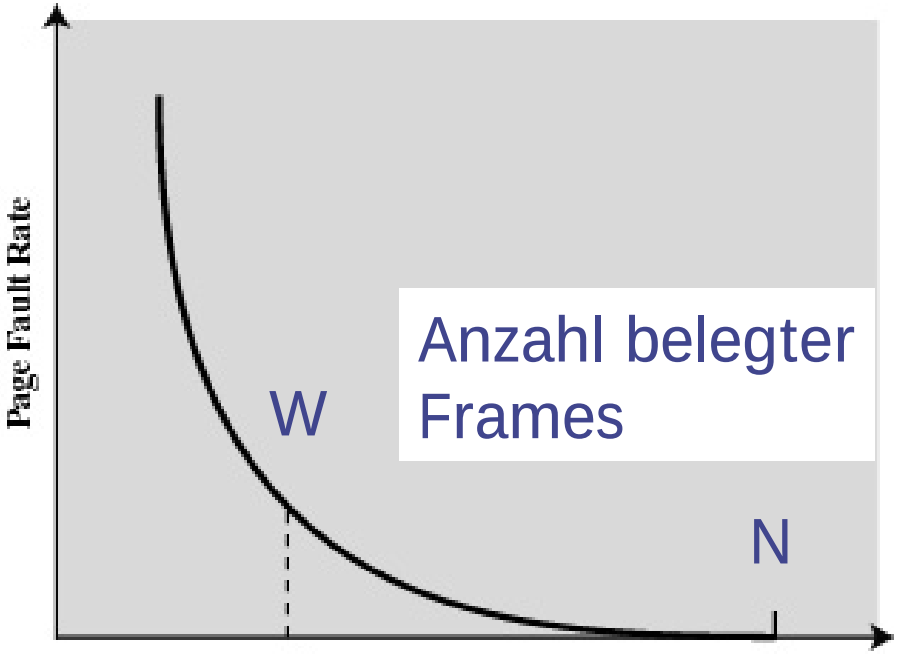
Idee: Tabelle gibt an, wer welchen Rahmen belegt

- i-ter Eintrag für i-ten Rahmen
- je Rahmen, ein Eintrag <prozessid,page#,chain>
- wenn Prozeß X auf Page Y zugreift, wird Tabelle durchsucht
  - falls <X,Y,?> in der i-ten Zeile der Tabelle enthalten, befindet sich Seite Y in Rahmen i
  - falls nicht, Seitenfehler!
- Suchvorgang wird durch Hashing beschleunigt, Konflikte der Hashfunktion werden durch Chaining behandelt. (Deshalb muß auch die Seitennummer in der Tabelle stehen!)
- Achtung: Frame# in Stallings, Abb. 8.6 unnötig!

Überlegungen zur Bestimmung der Seitengröße

bei kleinen Seiten, wenig interne Fragmentierung, viele Seiten im Speicher, wenig Page faults, weil zuletzt benutzte Seiten im Speicher sind
bei großen Seiten viele nicht-lokale Referenzen auf einer Seite,
Page faults steigen aber nicht beliebig, weil im Extremfall P nur 1 Seite hat.





P=Größe des Prozesses, W=Working Set, N = Gesamtzahl Seiten
Bei fester Seitengröße: Wieviele Seiten soll ein Prozeß bekommen ?
Working Set Idee: nur so viel wie aufgrund der Lokalität in der nächsten Zeit gebraucht wird.
Problem: zukünftiges Zugriffsverhalten unbekannt, daher hilfsweise Vergangenheit betrachten.

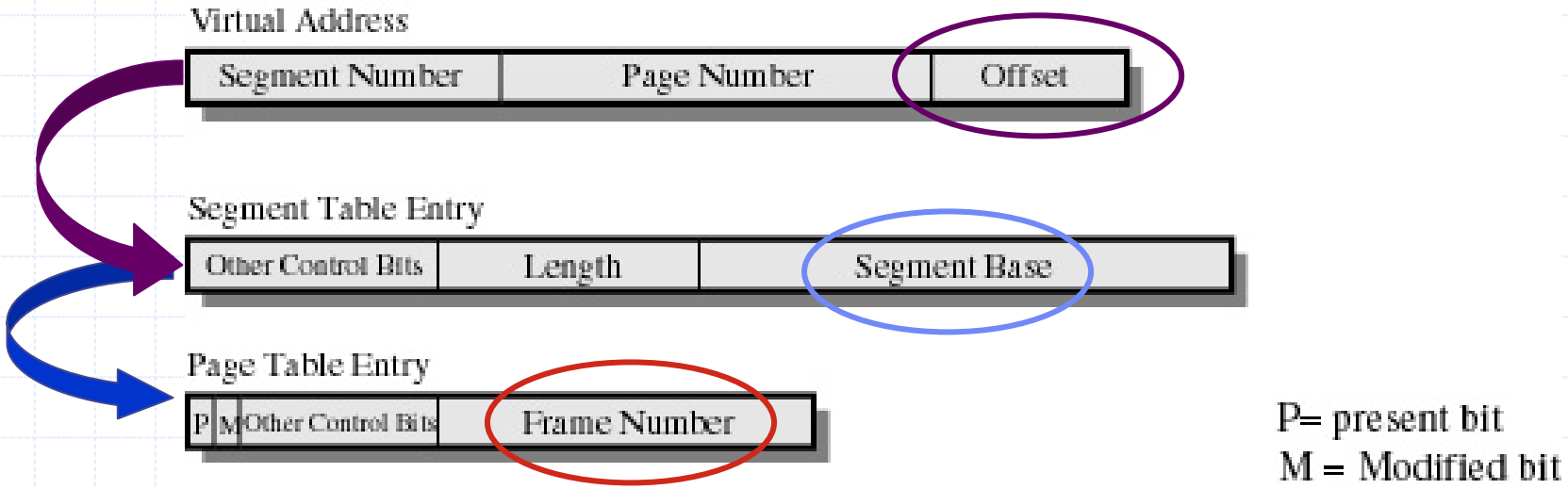
Festlegung der Seitengröße

- Seitengrößen sind meist fest gewählt, typisch: 4K, 8K
- HW Funktionalität für mehrere, unterschiedliche Größen selten genutzt
- wg Lokalität, interner Fragmentierung, Seitenfehlerrate und Hauptspeichernutzung ist die gewählte Größe relativ klein
- wg Platzbedarf für Seitentabellen und HW Eigenschaften von Plattenspeichern für schnellen Datentransfer mit DMA sind Seiten nicht beliebig klein

Segmentierung und Paging

- Segmentierung teilt Prozeß in Teile unterschiedlicher Größe auf
  - kann für verbesserten Schutzmechanismus genutzt werden
  - erlaubt einfache Unterteilung von Code und Datensegmenten
  - Adreßrechnung wg variabler Basisadressen mit Addition etwas komplizierter
  - Verwaltung mit Tabellen ähnlich
- Kombination von Segmentierung und Paging ist möglich
  - Segmente für Grobpartitionierung, jedes Segment wird durch Menge von Seiten repräsentiert
  - logische Adresse besteht aus Segmentnummer und Offset, wobei Offset in 2 Teile <Seitennummer,Seitenoffset> zerlegt wird.

Kombination aus Segmentierung und Paging



Adreßberechnung bei kombiniertem Verfahren

Segmenttabellenadresse + Segmentnummer = Zeile der Segmenttabelle

Segmentbase = Seitentabellenadresse

Seitentabellenadresse + Seitennummer = Zeile der Seitentabelle

Rahmennummer | Offset = physikalische Adresse

Zwischenfazit

- Paging und Segmentierung
  - grobe Vorgehensweise
  - Adreßtransformation, i.w. hardwareunterstützt
- jetzt: Strategien, die in OS Software umgesetzt werden
  - Laden von Seiten
  - Plazieren von Seiten
  - Ersetzen von Seiten
  - Verwalten des Resident Sets
  - Sichern von Seiten
- im wesentlichen für Paging betrachtet

Laden von Seiten (Fetch Policy)

- Wann werden welche Seiten geladen ?
- Seiten werden bei Bedarf geladen (demand paging)
  - nur Seitenfehler (Page Faults) lösen Ladevorgang aus
  - zu erwarten: bei Prozeßstart entstehen zunächst sehr viele Page Faults, bis der Prozeß über genügend Seiten seines Working Sets verfügt, die Seitenfehlerrate geht dann auf ein geringes Niveau zurück.
- Seiten werden im voraus geladen (Prepaging)
  - Seiten werden auf Vorrat geladen, z.B. bei Prozeßstart werden eine Anzahl Code Seiten geladen, bei Page Faults wird die Seite und Folgeseiten geladen.
  - wg Datentransfermöglichkeiten vom Plattenspeicher vielversprechend, falls Seiten dann auch genutzt werden
  - kaum realisiert !

Auswahl von Rahmen (Placement Policy)

- In welchen Rahmen/Hauptspeicherbereich soll eine Seite/Segment geladen werden ?
- Bei Paging und Kombination von Segmentierung mit Paging, ist die Frage irrelevant, weil Zugriffszeiten auf alle Rahmen gleich sind.
- Bei reiner Segmentierung interessant, Lösungen analog zu einfacher Segmentierung (First-Fit, Next-Fit, Best-Fit)

  Achtung: reine Segmentierung kommt in der Praxis kaum vor !!!

hier nicht weiter betrachtet!

Ersetzen von Seiten (Replacement Policy)

- Welche vorhandene Seite soll beim Laden von Seiten ersetzt/überschrieben werden ?
- Sperren von Frames wg besonderen Bedarfs (Kernel, I/O DMA Anforderung, Kontrollstrukturen wie PCBs, ...)
- Verschiedene, einfache Algorithmen
  - optimales Verfahren: ersetze Seite, deren nächster Zugriff am weitesten in der Zukunft liegt. (nur zu Vergleichszwecken)
  - Least Recently Used (LRU): ersetze Seite, deren letzter Zugriff am weitesten in der Vergangenheit liegt. (wg Lokalität gut, aber Realisierung ist zu aufwendig)
    - z.B. Jede Seite erhält bei Zugriff einen Zeitstempel
    - z.B. Verwalte Liste mit Seiten nach Zugriffen geordnet
  - First-in-first-out (FIFO): ersetze Seite, die die längste Zeit im Speicher residiert. (einfach, Implementierung mit Zeiger über zyklischen Puffer, geht jedoch am Bedarf vorbei, Beladys Anomalie)
  - LRU Approximation / FIFO mit 2. Chance / Uhren-Algorithmus

FIFO Anomalie / Beladys Anomalie

- gewünschte Eigenschaft: falls Prozeß mehr Rahmen bekommt, sollte die Anzahl Seitenfehler sinken
- FIFO hat diese Eigenschaft nicht!
- Beispiel: Sequenz 1,2,3,4,1,2,5,1,2,3,4,5 bei 5 Seiten

FIFO bei 3 Rahmen

1. lade 1,2,3 (1,2,3)
2. wechsel 1-4 (2,3,4)
3. wechsel 2-1 (3,4,1)
4. wechsel 3-2 (4,1,2)
5. wechsel 4-5 (1,2,5)
6. zugriff 1,2
7. wechsel 1-3 (2,5,3)
8. wechsel 2-4 (5,3,4)
9. zugriff 5

Page Faults: 9

FIFO bei 4 Rahmen

1. lade 1,2,3,4 (1,2,3,4)
2. zugriff 1,2
3. wechsel 1-5 (2,3,4,5)
4. wechsel 2-1 (3,4,5,1)
5. wechsel 3-2 (4,5,1,2)
6. wechsel 4-3 (5,1,2,3)
7. wechsel 5-4 (1,2,3,4)
8. wechsel 1-5 (2,3,4,5)

Page Faults: 10

Seitenersetzungsverfahren: Clock / Uhren-Algorithmus

- Einfachste Variante: 1 Bit je Rahmen (Use-Bit)
  - bei jedem Seitenzugriff wird Bit auf 1 gesetzt
  - Ersetzungsverfahren durchläuft zyklisch alle Rahmen (lokal=nur des einzelnen Prozesses, global=Hauptspeicher),

    falls Bit = 1 dann Bit auf 0 setzen, sonst Seite ersetzen
  - Verfahren beginnt jeweils beim Nachfolger der zuletzt ersetzten Seite
  - spätestens im zweiten Durchlauf wird eine Seite gefunden
- Erweiterung auf mehrere Bits möglich, z.B.

  Modified Bit kann mit genutzt werden -> Bits (U,M)

  Reihenfolge:

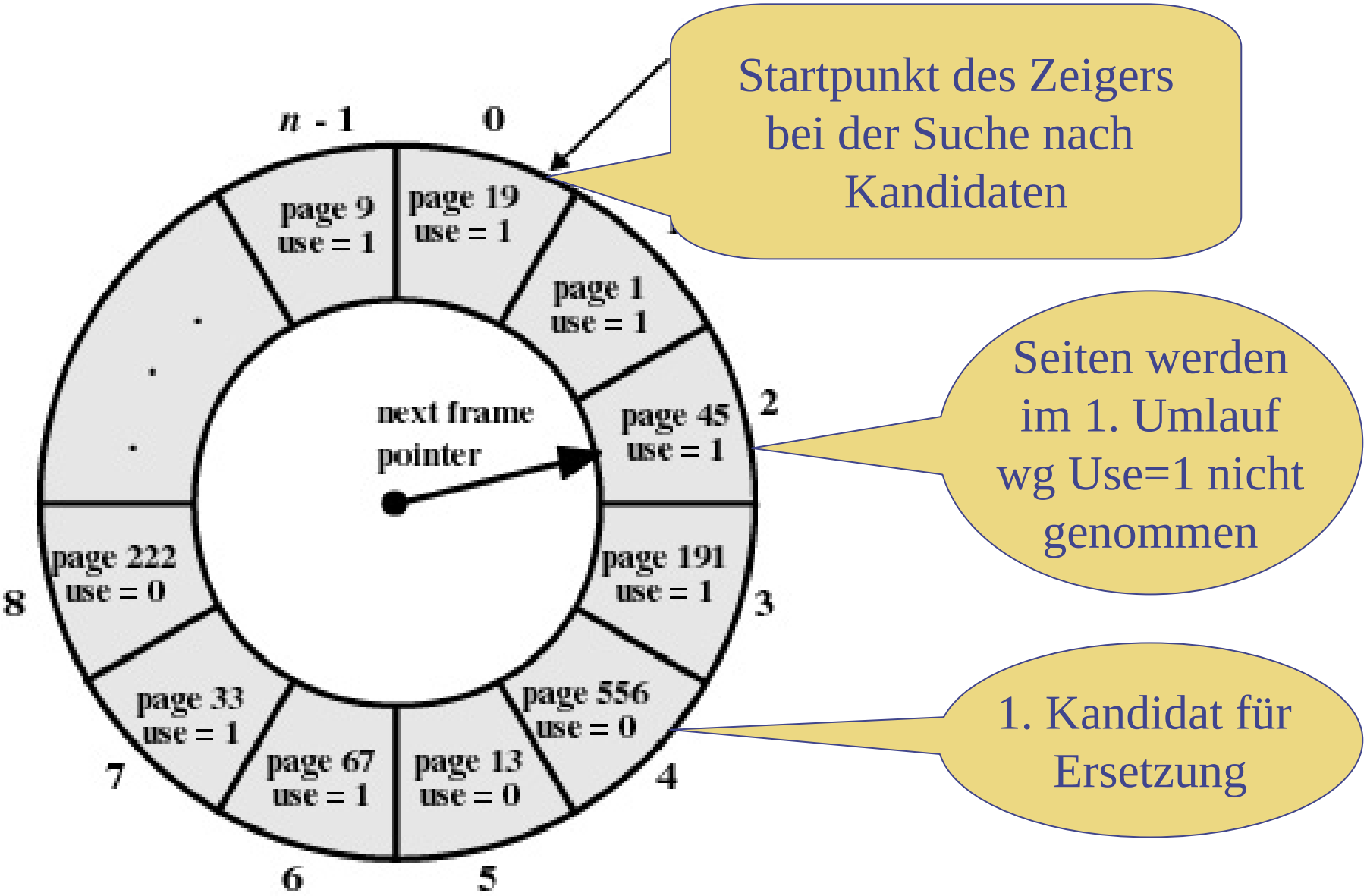
  1. (0,0): nicht genutzt, nicht modifiziert, 2. (0,1): nicht genutzt, modifiziert
  3. (1,0): genutzt, nicht modifiziert, 4. (1,1): genutzt, modifiziert

  1. Runde: wähle (0,0) Seite, 2. Runde: wähle (0,1) Seite, setze U auf 0

  3. Runde: wähle (0,0) Seite, 4. Runde: wähle (0,1) Seite

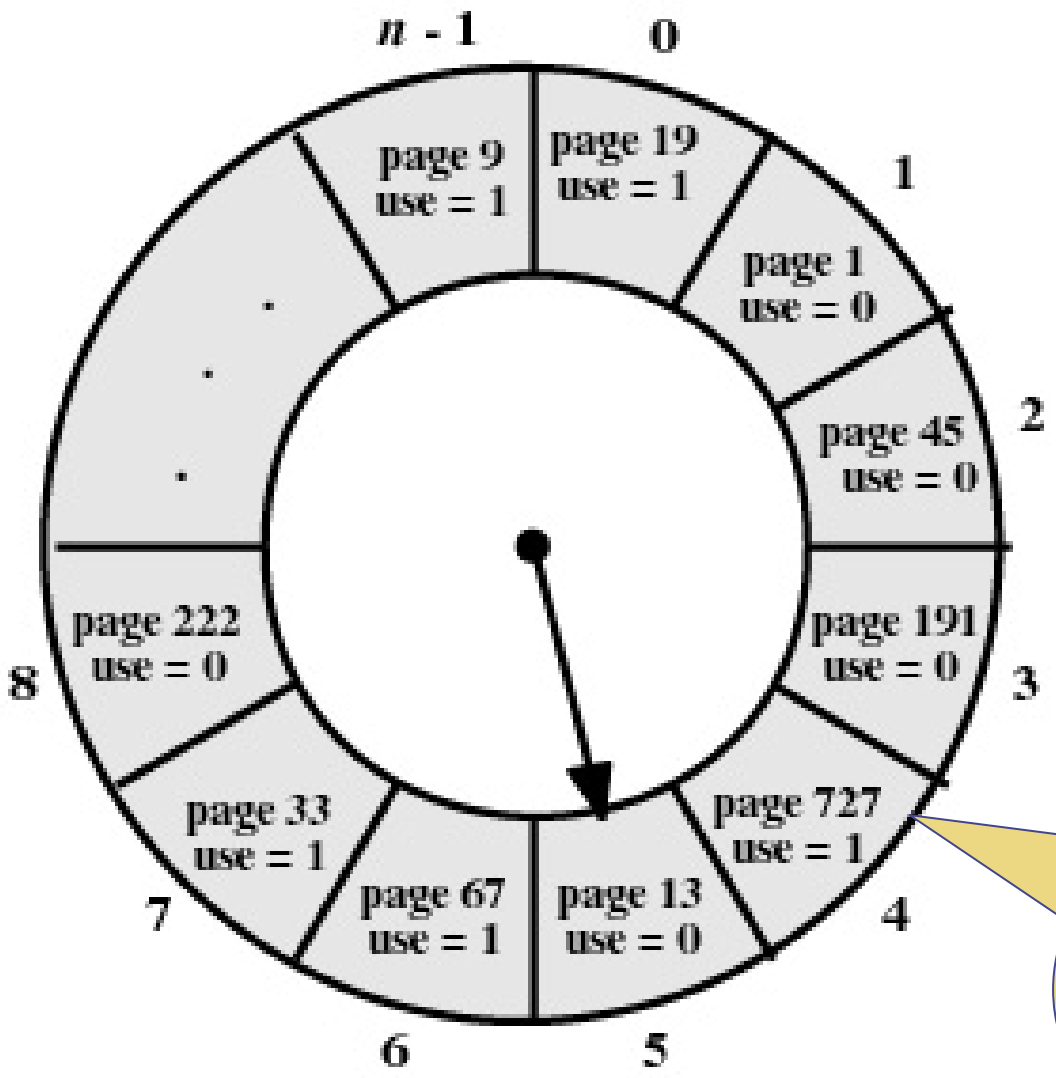
  contra: Aufwand pro: unmodifizierte Seiten ohne Write Back

Beispiel für Uhren Algorithmus



(a) State of buffer just prior to a page replacement

Beispiel für Uhren Algorithmus



(b) State of buffer just after the next page replacement

Uhren-Algorithmus wird z.B. in Mac-OS eingesetzt, verursacht jedoch Aufwand!
UNIX SVR4 Variante mit 2 Zeigern (Fronthand,Backhand)
Linux mit 8bit Zähler, daher fast LRU.

Seite ist ersetzt worden, Zeiger steht auf Nachfolger

Weiteres Verfahren: Puffern von Seiten

z.B. VAX VMS, Mach

- Idee
  - freie Seiten werden in einer Liste verwaltet, Liste ist stets gefüllt
  - zu ersetzende Seiten werden an das Ende der Liste angefügt
  - bei Anforderung eines Frames wird Seite am Anfang der Liste entnommen und dadurch zerstört
- Ziel: Liste wirkt als Puffer, weil Inhalte erst zu einem späteren Zeitpunkt zerstört werden. Unterscheidung modifizierter/unmodifizierter Seiten möglich -> 2 Listen
- Positive Effekte:
  - Seiteninhalte aus der Liste können gebündelt auf die Festplatte geschrieben werden (Sichern)
  - Seiteninhalte aus der Liste können bei Zugriffen recycelt werden, dh die Seite wird einfach wieder in die Seitentabelle eingetragen

Verwaltung des Resident Sets

- Wieviele Seiten eines Prozesses sollen im Hauptspeicher verfügbar sein ?
  - Nur wenige Seiten: viele Prozesse, aber häufig Page Faults
  - Viele Seiten: wenig Page Faults, aber Grenznutzen nimmt letztlich ab
- Größe des Resident Sets:
  - fest: bei Prozeßstart festgelegt (wie?), Ersetzung erfolgt nur lokal
  - variabel: Ersetzungsverfahren wählt aus allen (lokalen) Seiten aus, jeweilige Größe kann vom OS anhand von Page Fault Häufigkeiten und der Größe des freien Speichers gesteuert werden.
- Auswahl zu ersetzender Seiten nur im eigenen Bereich oder auch aus den Seiten anderer Prozesse:
  - lokale oder globale Auswahl.
  - Global: einfacher & weniger Overhead

Sichern von Seiten (Cleaning Policy)

- Wann sollen modifizierte Seiten auf der Festplatte gesichert/zurückgeschrieben werden ?
- Sichern bei Bedarf
  - Eine Seite wird gesichert, wenn der Rahmen durch eine andere Seite belegt wird.
  - Vorteil: I/0 nur bei dringendem Bedarf
  - Nachteil: Reaktion auf Page Fault mit Einladen einer Seite wird durch vorheriges Sichern zusätzlich verzögert
- Aufräumen im Voraus
  - Seiten werden nach Modifikation gesichert, ohne dass der Rahmen benötigt wird
  - Vorteil: Sichern für Mengen von Seiten nutzt HW Eigenschaften
  - Nachteil: Seiten werden unnötig gesichert
- Kompromiß: Puffern von Seiten mit regelmäßigem Sichern

Lastkontrolle (Load Control)

- Wieviele Prozesse sollen bearbeitet werden?
  - CPU Auslastung => viele Prozesse
  - bei zuvielen Prozessen, zuwenig Hauptspeicher => Thrashing
- Varianten:
  - bei Überwachung der Page Fault Häufigkeit zur Steuerung des Resident Sets implizit auch Anzahl aktiver Prozesse reguliert
  - empirische Regeln:
    - mittlere Zeit zwischen Page Faults

      = mittlere Zeit zur Behandlung eines Page Faults
    - 50% Kriterium: Ziel ist Auslastung der Paging Komponente bei 50% zu halten.
  - Beobachtung der Bewegungen des Zeigers im Uhren-Algorithmus mit Grenzwerten zur Erkennung von Unter- und Überlast-Situationen
- Reduktion der Prozeßzahl: Swapping

Design Fragen

- Wieviele Prozesse sollen bearbeitet werden ?
  - Situation erkennen (z.B. anhand des Uhren-Algos) und ggfs Swapping
- Welche Größe soll eine Seite haben ?
  - Relativ kleine Größen, meist 1 Größe, fest, typisch 4K, 8K
- Wieviele Seiten soll ein Prozeß haben ?
  - Resident Set möglichst gleich Working Set, Steuerung über Häufigkeiten von Page Faults
- Wann sollen Seiten geladen werden ?
  - Typisch nur bei Bedarf, ggfs initial auch auf Verdacht
- Welche Seiten sollen ggfs überschrieben/ersetzt werden ?
  - LRU Approximation mit Uhren-Algorithmus, Puffern von Seiten
- Wann sollen Seiten zurückgeschrieben werden ?
  - Puffern und periodisch in Gruppen
- Wie werden Seiten verwaltet ?
  - Seitentabellen (auch mehrstufig)

Tabellen in UNIX

- Je Prozeß eine Seitentabelle.
- Aber: Seitentabellen allein reichen nicht aus!
- Wo liegen Seiten, die nicht im Hauptspeicher sind ?
  - Zus. Tabelle „disk block descriptor“, dessen Einträge je Seite angeben, auf welchem Swap Device (mehrere möglich), in welchem Block und Format die Seite abgelegt ist.
- In welchem Zustand befindet sich ein Rahmen?
  - Zus. Tabelle „page frame data table“ zeigt an, ob Rahmen verfügbar, für DMA reserviert ist, etc. Wieviele Prozesse auf den Rahmen verweisen, welches logische Device eine Kopie der Daten hat, ...
- Was lagert auf einem Swap Device ?
  - Zus. Tabelle, mit Angaben über die Anzahl Verweise auf jede Seite im Swap Bereich und ihren Ort

Unix Tabellen zur Speicherverwaltung

Je Prozeß
eine Seitentabelle

Je Seite ein Eintrag
in zusätzlicher
Tabelle für
Swap Space
im Plattenspeicher

Page frame number	Age	Copy on write	Mod-ify	Refe-rence	Valid	Pro-tect

(a) Page table entry

Swap device number	Device block number	Type of storage

(b) Disk block descriptor

Figure 8.22 UNIX SVR4 Memory Management Formats

Unix Tabellen zur Speicherverwaltung

Rahmentabelle, je Rahmen ein Eintrag, Status: verfügbar, für DMA reserviert References: #Prozesse,...

Page state	Reference count	Logical device	Block number	Pfdata pointer

(c) Page frame data table entry

Reference count	Page/storage unit number

(d) Swap-use table entry

Je Swap Device 1 Tabelle, jeweils 1 Eintrag je Seite

Figure 8.22 UNIX SVR4 Memory Management Formats

Zusammenfassung

- Paging und Segmentierung
  - grobe Vorgehensweise
  - Adreßtransformation, i.w. hardwareunterstützt
- Strategien, die in OS Software umgesetzt werden, im wesentlichen für Paging betrachtet
  - Laden von Seiten
  - Plazieren von Seiten
  - Ersetzen von Seiten
  - Verwalten des Resident Sets
  - Sichern von Seiten