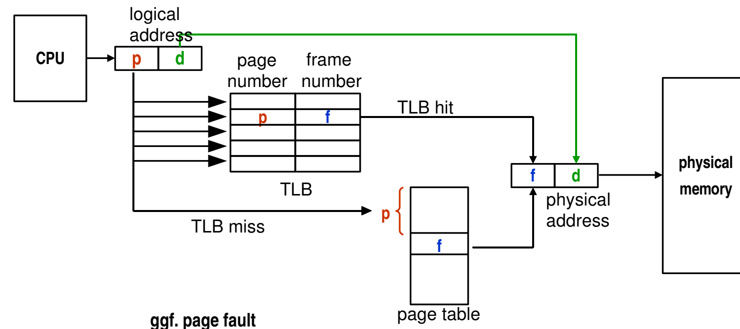
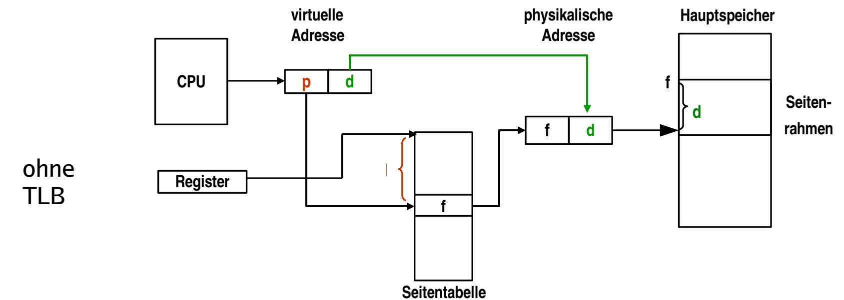
Speicherverwaltung (3)

Translation Look-Aside Buffer (1)

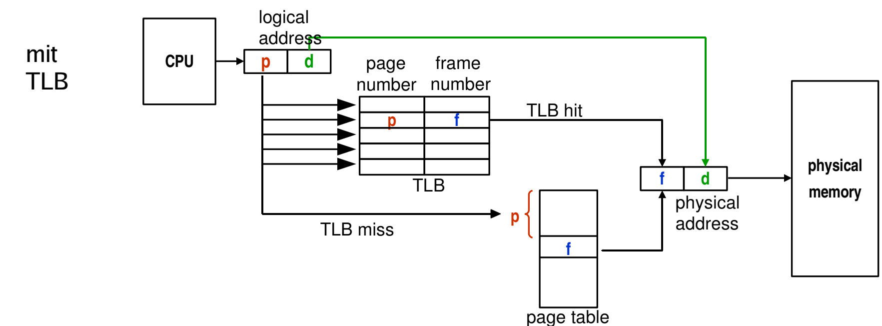
- Translation Lookaside Buffer (TLB): schneller Hardware-Cache, mit den zuletzt benutzten Seitentabelleneinträgen
- Assoziativ-Speicher: bei Übersetzung einer Adresse wird deren Seitennummer gleichzeitig mit allen Einträgen des TLB verglichen.



ohne TLB



mit TLB



Translation Look-Aside Buffer (2)

- Treffer im TLB
  -> Speicherzugriff auf Seitentabelle unnötig
- Fehltreffer
  -> Zugriff auf die Seitentabelle
  Alten Eintrag im TLB durch neuen ersetzen
- Trefferquote (hit ratio) beeinflusst die durchschnittliche Zeit einer Adressübersetzung.
- Lokalitätsprinzip: Programme greifen meist auf benachbarte Adressen zu
  -> auch bei kleinen TLBs hohe Trefferquoten (typisch: 80-98%).

Lokalitätsprinzip

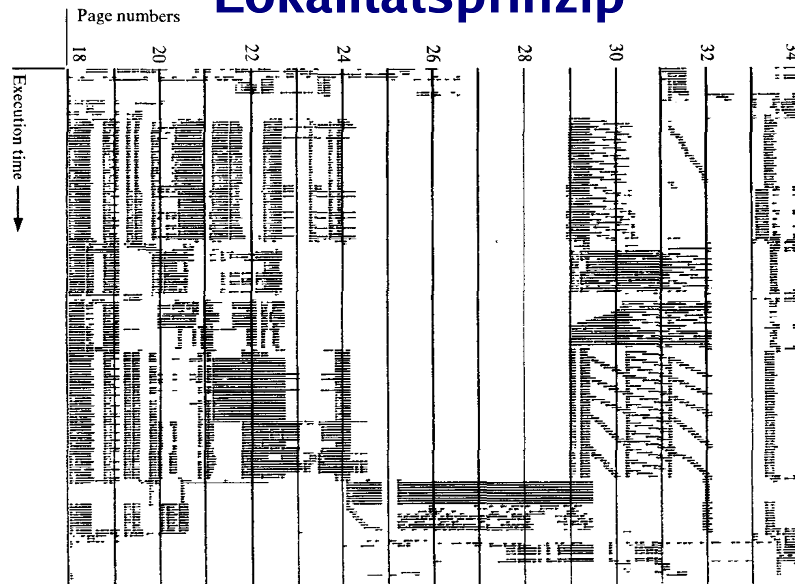


Bild: Hatfield (1972)

Translation Look-Aside Buffer (3)

- Inhalt des TLB ist prozessspezifisch!
  Zwei Möglichkeiten:
  - Jeder Eintrag im TLB enthält ein „valid bit“. Bei Prozesswechsel (Context Switch) wird der gesamte Inhalt des TLB invalidiert.
  - Jeder Eintrag im TLB enthält Prozessidentifikation (PID), die mit der PID des zugreifenden Prozesses verglichen wird.
- Beispiele für TLB-Größen:
  - Intel 80486: 32 Einträge.
  - Pentium-4, PowerPC-604: 128 Einträge für jeweils Code und Daten.

Translation Look-Aside Buffer (4) – mit Speicher-Cache

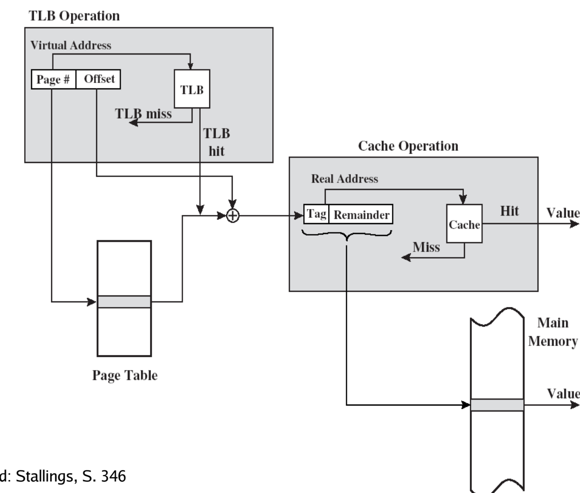


Bild: Stallings, S. 346

Translation Look-Aside Buffer (5)

Was macht hier eigentlich das Betriebssystem?

- Page-Table-Register laden
- Im Falle eines Page Fault: Fehlende Seite aus dem Swap holen und Seitentabelle aktualisieren
- Evtl. vorher: Seitenverdrängung – welche Seite aus dem Hauptspeicher entfernen? (-> später)

Alles andere: Hardware

- Zugriff auf TLB und ggf. auf Seitentabelle
- Wenn Seite im Speicher: Berechnung der phys. Adresse
- Inhalt aus Cache oder ggf. aus Hauptspeicher holen

Software-TLB

- Alternative zum TLB-Handling durch CPU/MMU
- Steht Seite nicht im TLB, erzeugt die MMU einen TLB-Fehler
- Betriebssystem führt dann Fehlerbehandlungs-routine für TLB-Fehler aus:
  - Suche Seite
  - Wähle einen TLB-Eintrag aus, der verdrängt werden kann
  - Überschreibe (verdrängten) TLB-Eintrag mit Inhalt der neuen Seite
  - Nimm Befehlsausführung wieder auf (an der Stelle, an der TLB-Fehler auftrat)

Invertierte Seitentabellen (1)

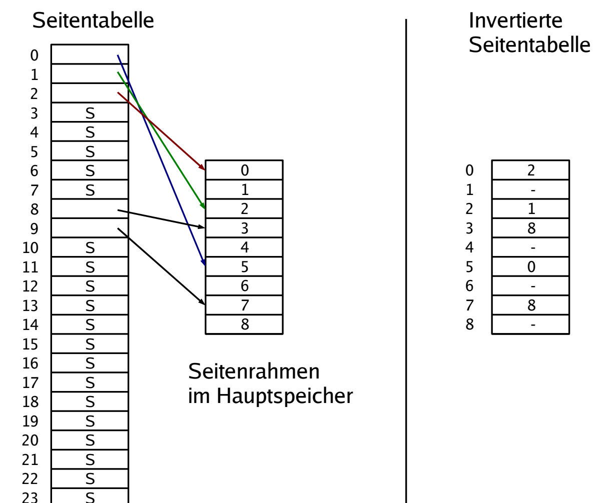
- Bei großem virtuellen Speicher sehr viele Einträge in der Seitentabelle nötig,
  z.B. 2^{32} Byte Adressraum, 4 KByte/Seite
  -> über 1 Millionen Seiteneinträge,
  also Seitentabelle > 4 MByte (pro Prozess!)
- Platz sparen durch invertierte Seitentabellen:
  - normal: ein Eintrag pro (virtueller) Seite mit Verweis auf den Seitenrahmen (im Hauptspeicher)
  - invertiert: ein Eintrag pro Seitenrahmen mit Verweis auf Tupel (Prozess-ID, virtuelle Seite)

Invertierte Seitentabellen (2)

- Problem: Suche zu Prozess *p* und seiner Seite *n* nach dem Eintrag (*p*,*n*) in der invertierten Tabelle -> langwierig
- Auch hier TLB einsetzen, um auf „meist genutzte“ Seiten schnell zugreifen zu können
- Bei TLB-Miss hilft aber nichts: Suchen...

- Andere Lösung für Problem der großen Seitentabellen: Mehrstufiges Paging (-> gleich)

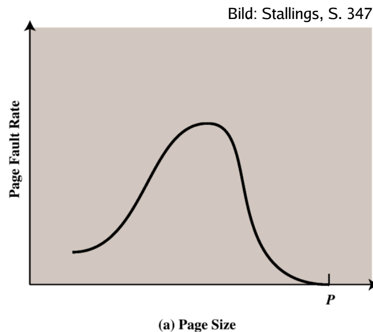
Invertierte Seitentabellen (3)



Auswirkungen der Seitengröße

- Interne Fragmentierung: Je kleiner die Seiten, desto geringer die Fragmentierung
- Kleine Seiten -> große Tabellen evtl. Teil der Tabelle ausgelagert -> doppelte Page Fault beim Zugriff auf eine Seite, deren Tabelleneintrag ausgelagert ist
- Lokalitätsprinzip: Kleine Seiten: lokal, wenig Faults. Größere Seiten, nicht mehr lokal. Annäherung der Seitengröße an Gesamtgröße *P* des Prozessspeichers

Bild: Stallings, S. 347



(a) Page Size

Mehrstufiges Paging (1)

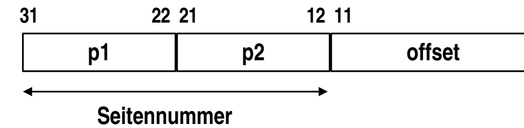
Die Seitentabelle kann sehr groß werden.

Beispiel:
- 32-Bit-Adressen,
- 4 KByte Seitengröße,
- 4 Byte pro Eintrag

Seitentabelle:
>1 Million Einträge,
4 MByte Größe (pro Prozess!)

Mehrstufiges Paging (2)

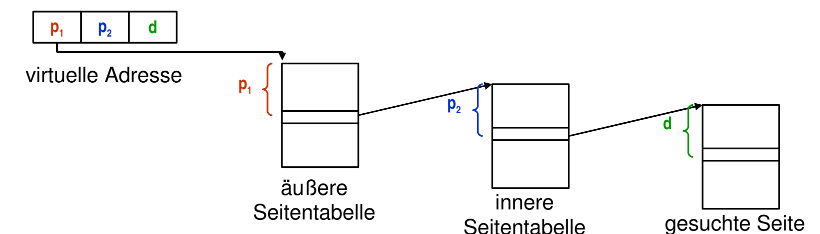
- Zweistufiges Paging:
  - Seitennummer noch einmal unterteilen, z. B.:



  - p_1: Index in äußere Seitentabelle, deren Einträge jeweils auf eine innere Seitentabelle zeigen
  - p_2: Index in eine der inneren Seitentabellen, deren Einträge auf Seitenrahmen im Speicher zeigen
  - Die inneren Seitentabellen müssen nicht alle speicherresident sein
- Analog dreistufiges Paging etc. implementieren

Adressübersetzung bei mehrstufigem Paging (1)

Zweistufiges Paging:



Adressübersetzung bei mehrstufigem Paging (2)

Dreistufiges Paging bei SPARC-Prozessoren:

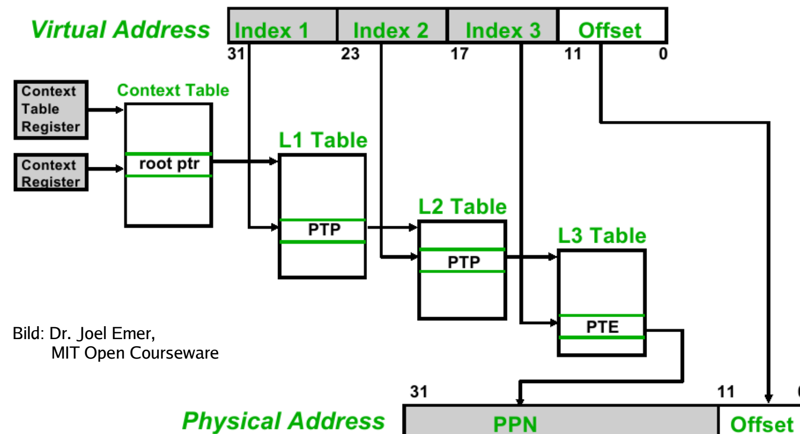


Bild: Dr. Joel Emer,
MIT Open Courseware

Mehrstufiges Paging (4)

- Größe der Seitentabellen:

Beispiel:

p_1	p_2	offset
10	10	12

  - Die äußere Seitentabelle hat 1024 Einträge, die auf (potentiell) 1024 innere Seitentabellen zeigen, die wiederum je 1024 Einträge enthalten.
  - Bei einer Länge von 4 Byte pro Seitentabelleneintrag ist also jede Seitentabelle genau eine 4-KByte-Seite groß.
  - Es werden nur so viele innere Seitentabellen verwendet, wie nötig.

Mehrstufiges Paging (5)

- Jede Adressübersetzung benötigt noch mehr Speicherzugriffe, deshalb ist der Einsatz von TLBs noch wichtiger.
- Als Schlüssel für den TLB werden alle Teile der Seitennummer zusammen verwendet (p_1,p_2,...).

Speicherschutz beim Paging (1)

- Schutz vor Zugriff durch andere Prozesse:
  - Da jeder Prozess eine eigene Seitentabelle hat, ist Zugriff auf Speicherbereiche anderer Prozesse nicht möglich. (Dies macht andererseits die Implementierung von gemeinsam benutzten Speicherbereichen aufwendiger.)
- Schutz vor (z. B.) unberechtigtem Schreiben:
  - Die Einträge der Seitentabellen enthalten zusätzlich einen Schutzcode, der z. B. angibt, ob die Seite gelesen und/oder geschrieben werden darf (evtl. auch noch abhängig davon, ob der Zugriff im User- oder im Kernel-Mode erfolgt).

Speicherschutz beim Paging (2)

- Die Seiteneinteilung ist transparent für Programmierer !
- Festlegen des Schutzcodes durch Compiler und/oder Linker:
  - Das Programm wird in Abschnitte eingeteilt, deren Größe ein Vielfaches der Seitengröße ist.
  - Pro Abschnitt wird ein Schutzcode für alle Seiten dieses Abschnitts festgelegt und im Kopf der Programmdatei vermerkt.
  - Der Loader setzt die Schutzcodes in den Seitentabelleneinträgen.

Seiten-Sharing beim Paging (1)

- *Theoretisch* könnten Einträge verschiedener Seitentabellen auf den gleichen Seitenrahmen zeigen.

  Probleme:
  - Wie stellt man fest, ob eine Seite bereits von einem anderen Prozess benutzt wird, und in welchem Seitenrahmen sich diese befindet?
  - Bei Änderungen (z. B. des verwendeten Seitenrahmens) wären viele Seitentabellen anzupassen.

Seiten-Sharing beim Paging (2)

- *Praktisch* wird der gemeinsam zu benutzende Teil des Adressraums
  - entweder als gemeinsam benutzbares Segment mit eigener Seitentabelle implementiert (Kombination von Segmentierung und Paging, z. B. bei Unix) oder
  - es werden die gemeinsam zu nutzenden Teile als eine Art Pseudo-Prozess-Adressbereich implementiert, für den es eine eigene (globale) Seitentabelle gibt (z. B. bei Windows).

Segmentierung mit Paging (1)

- Programme werden in Segmente unterteilt, die aber nicht zusammenhängend, sondern mit Hilfe von Paging im Speicher abgelegt werden.
- Der Programmierer gibt die Segment-Nummer und eine relative Adresse in diesem Segment an. Das Paging wird - transparent für das Programm - von der Hardware durchgeführt.

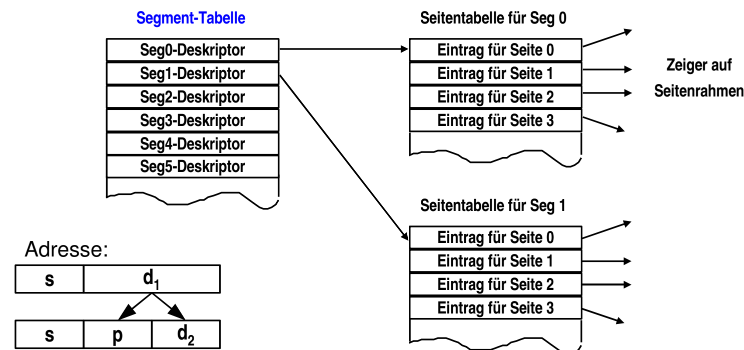
Segmentierung mit Paging (2)

Zwei Realisierungsmöglichkeiten:

- Eine Segmenttabelle und pro Segment (pro Prozess) eine eigene Seitentabelle.
  Die Einträge in der Segmenttabelle zeigen auf die Seitentabelle des jeweiligen Segments (z. B. Unix).
- Es gibt eine Segmenttabelle und eine einzige Seitentabelle (pro Prozess) (z. B. Intel-CPUs).
  - Die Einträge in der Segmenttabelle enthalten die Basisadresse des Segments in einem linearen (virtuellen) Adressraum.
  - Die relative Adresse im Segment wird zu dieser Basisadresse addiert. Die resultierende lineare (virtuelle) Adresse wird mittels der Seitentabelle in eine physikalische Adresse übersetzt.

Segmentierung mit Paging (3)

- Eine eigene Seitentabelle pro Segment:



Segmentierung mit Paging (4)

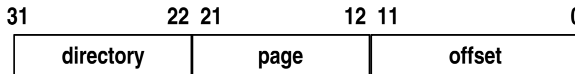
- Eine Adressangabe besteht aus Segmentnummer und linearer Adresse im Segment.
  Die Aufteilung der linearen Adresse in Seitennummer und Offset (für das Paging) ist transparent für das Programm.

Segmentierung mit Paging: Intel 80386

- Segmente:
  - Bis zu 8 K prozessprivate Segmente, beschrieben durch Einträge in der local descriptor table (LDT).
  - Bis zu 8 K von allen Prozessen gemeinsam benutzte (shared) Segmente, beschrieben durch Einträge in der global descriptor table (GDT).
  - Jedes Segment kann bis zu 4 GByte groß sein.
- Sechs Segmentregister, so dass zu jeder Zeit sechs Segmente gleichzeitig von einem Prozess benutzt werden können.

Segmentierung mit Paging: Intel 80386

- Sechs interne Register, die die entsprechenden Segment-Deskriptoren enthalten.
- Eine Adressangabe ist ein Paar
  (Segment-Selektor, Offset).
- Der Segment-Deskriptor ist die Basisadresse des Segments in einem 32-Bit-Adressraum, zu der der Offset addiert wird, um die so genannte lineare Adresse zu erhalten.
- Die lineare Adresse hat das Format (two-level paging):



Adressübersetzung beim Intel 80386