Betriebssysteme I – Sommersemester 2009
Kapitel 6: Speicherverwaltung und Dateisysteme

Hans-Georg Eßer
Hochschule München

Teil 5: Nicht-zusammenhängende Speicherzuordnung (2/3)

06/2009

Nicht-zusammenhängende Speicherzuordnung: Paging

- Einführung
- Paging

Ausgangslage

- Speicher zu knapp für große Programme
  → Overlay-Programmierung
- Programmteile dynamisch nachladen, wenn sie benötigt werden
- Programmierer muss sich um Aufteilung in Overlays kümmern

Overlay-Programmierung

Turbo Pascal, um 1985-90:

```
program grossesprojekt;

overlay procedure kundendaten;
...
overlay procedure lagerbestand;
...

{ Hauptprogramm }
begin
  while input <> "exit" do begin
    case input of
      "kunden": kundendaten;
      "lager": lagerbestand;
    end;
  end;
end.
```

Lösung des Problems

- Virtueller Speicher, der das gesamte Programm aufnehmen kann
- Programm sieht Speicherbereich, der ihm zur Verfügung gestellt wurde – wie viel wirklich vorhanden ist, spielt (für das Programm) keine Rolle

Virtuelle Speicherverwaltung (Paging)

- Aufteilung des Adressraums in Seiten (pages) fester Größe und des Hauptspeichers in Seitenrahmen (page frames) gleicher Größe.

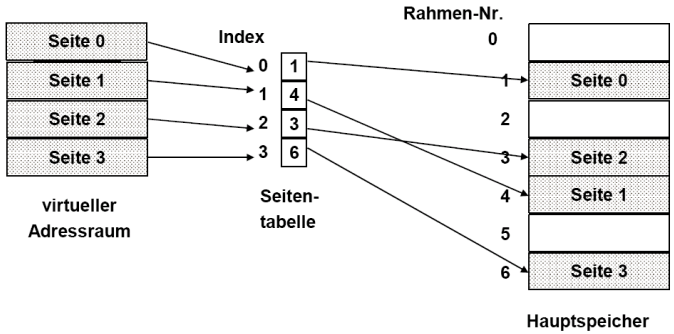
  Typische Seitengrößen: 512 – 8192 Byte (immer Zweierpotenz).
- Der lineare, zusammenhängende Adressraum eines Prozesses („virtueller“ Adressraum) wird auf beliebige, nicht zusammenhängende Seitenrahmen abgebildet.
- Eine einzige Liste freier Seitenrahmen wird vom Betriebssystem verwaltet.

Virtuelle Speicherverwaltung (Paging)

- Die Berechnung der physikalischen Speicheradresse aus der vom Programm angegebenen virtuellen Adresse
  - geschieht zur Laufzeit des Programms,
  - ist transparent für das Programm,
  - muss von der Hardware unterstützt werden.
- Vorteile der virtuellen Speicherverwaltung:
  - Einfache Zuteilung von Hauptspeicher.
  - Keine externe Fragmentierung, geringe interne Fragmentierung.
  - Kein Aufwand für den Programmierer.

Virtueller Adressraum (1)

- Beim Paging wird der Zusammenhang zwischen Programmadresse und physikalischer Hauptspeicheradresse erst zur Laufzeit mit Hilfe der Seitentabellen hergestellt.



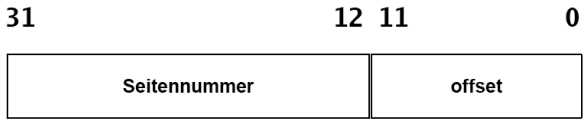
Virtueller Adressraum (2)

- Die vom Programm verwendeten Adressen werden deshalb auch virtuelle Adressen genannt.
- Der virtuelle Adressraum eines Programms ist der lineare, zusammenhängende Adressraum, der dem Programm zur Verfügung steht.

Adressübersetzung beim Paging (1)

- Die Programmadresse wird in zwei Teile aufgeteilt:
  - eine Seitennummer
  - eine relative Adresse (offset) in der Seite

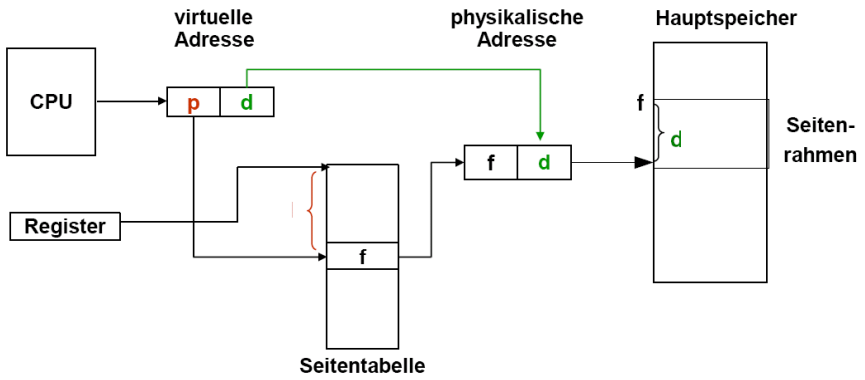
Beispiel: 32-bit-Adresse bei einer Seitengröße von 4096 ($=2^{12}$) Byte:



Adressübersetzung beim Paging (2)

- Für jeden Prozess gibt es eine Seitentabelle (page table). Diese enthält für jede Prozessseite
  - eine Angabe, ob die Seite im Speicher ist,
  - die Nummer des Seitenrahmens im Hauptspeicher, der die Seite enthält.
- Ein spezielles Register enthält die Anfangsadresse der Seitentabelle für den aktuellen Prozess.
- Die Seitennummer wird als Index in die Seitentabelle verwendet.

Adressübersetzung beim Paging (3)



Adressübersetzung beim Paging (4)

- Für jeden Hauptspeicherzugriff wird ein zusätzlicher Hauptspeicherzugriff auf die Seitentabelle benötigt. Dies muss durch Caches in der Hardware beschleunigt werden!
- Seite nicht im Speicher → spezielle Exception, einen sog. page fault (Seitenfehler) auslösen.

Virtueller Speicher allgemein (1)

- Mehr Prozesse können effektiv im Speicher gehalten werden → bessere Systemauslastung
- Ein Prozess kann viel mehr Speicher anfordern als physikalisch verfügbar

Virtueller Speicher allgemein (2)

allgemeiner Vorgang:

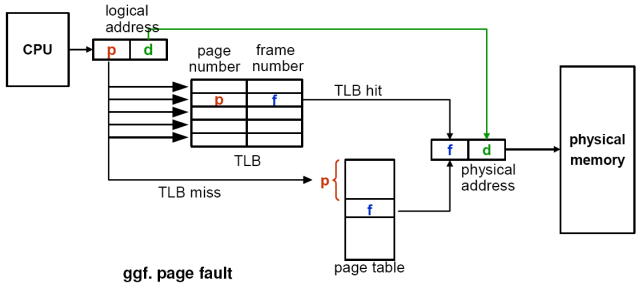
- Nur Teile des Prozesses befinden sich im physikalischen Speicher
- falls Zugriff auf eine Adresse, die ausgelagert ist:
  - BS setzt den Prozess auf blockiert
  - BS setzt eine Disk-I/O-Leseanfrage ab
  - Nach Laden des fehlenden Stücks (Seite oder Segment) wird ein I/O-Interrupt abgesetzt
  - das BS setzt Prozess zuletzt wieder in den Bereit- (Ready-) Zustand

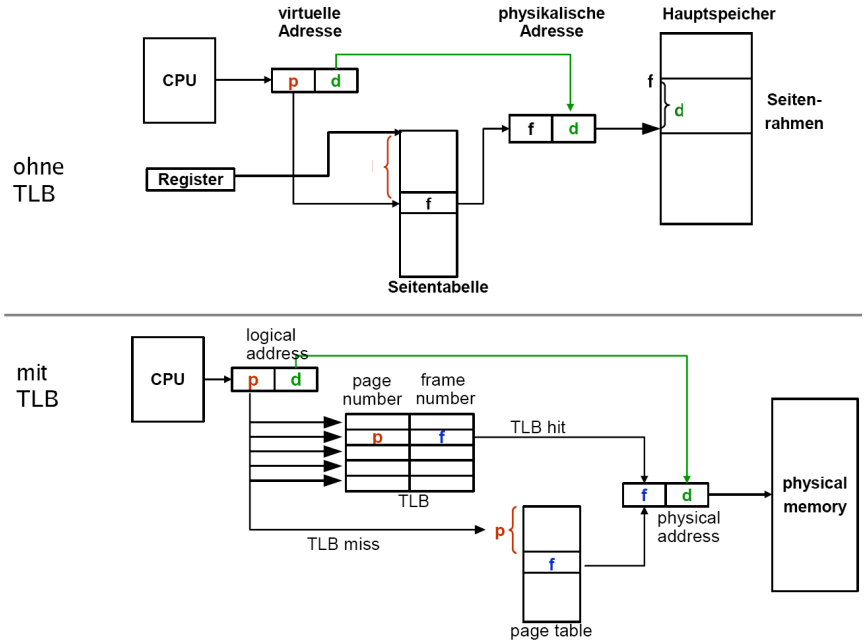
Virtueller Speicher allgemein (3)

- „thrashing“ (siehe später): Prozessor verbringt die meiste Zeit mit Ein- und Auslagern von Prozessteilen statt mit der Ausführung von Prozessanweisungen
- **Lokalitätsprinzip:**
  Zugriffe auf Daten und Programmcode häufig lokal gruppiert; → Annahme gerechtfertigt, dass nur wenige Prozessstücke während einer kurzen zeitlichen Periode gleichzeitig vorgehalten werden müssen

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.





Translation Look-Aside Buffer (3)

- 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

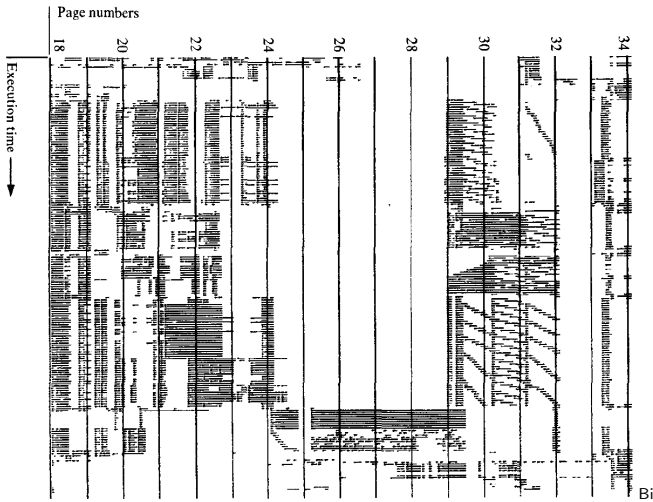


Bild: Hatfield (1972)

Translation Look-Aside Buffer (4)

- 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 (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

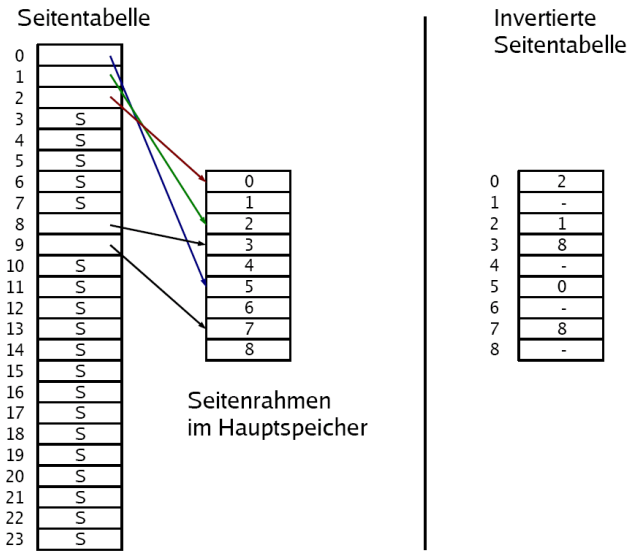
Invertierte Seitentabellen (1)

- 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)

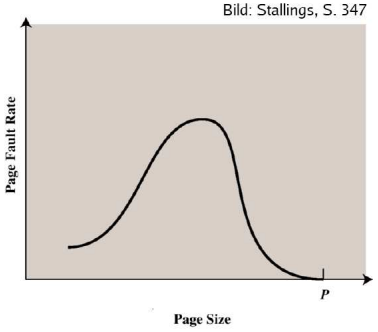
Invertierte Seitentabellen (3)



Auswirkungen der Seitengröße

- Interne Fragmentierung: Je kleiner die Seiten, desto geringer die Fragmentierung
- Kleine Seiten → große Tabellen evtl. Teil der Tabelle ausgelagert → doppelter 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



Mehrstufiges Paging (1)

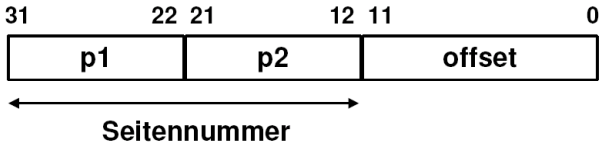
Die Seitentabelle kann sehr groß werden.
Beispiel:

- 32-Bit-Adressen
- 4 KByte Seitengröße
- 4 Byte pro Eintrag

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

Mehrstufiges Paging (2)

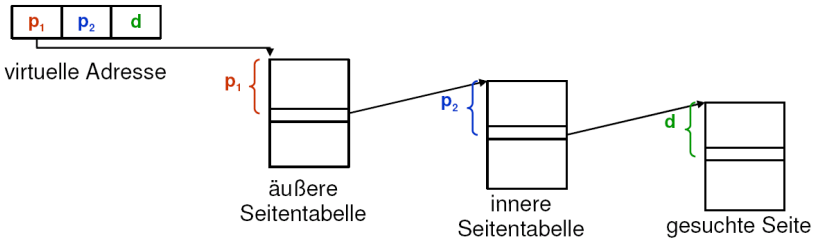
- Zweistufiges Paging:
  - Seitennummer noch einmal unterteilen, z. B.:



  - 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

Mehrstufiges Paging (3)

Adressübersetzung bei zweistufigem Paging:



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).