6. Speicherverwaltung

6. Speicherverwaltung

6.1 Zusammenhängende Speicherzuteilung

6.2 Nicht-zusammenhängende Speicherzuteilung

Speicherverwaltung: Gliederung

- **6.1 Zusammenhängende Speicherzuteilung**
  - Partitionen fester / variabler Größe
  - Methoden zur Verwaltung des freien Speichers
- **6.2 Nicht zusammenhängende Speicherzuteilung**
  - Virtuelle Speicherverwaltung (Paging)
  - Mehrstufiges Paging
  - Demand Paging
  - Page Faults und deren Behandlung
  - Strategien für die Seitenersetzung

Arten der Speicherverwaltung

Zwei prinzipielle Arten der Speicherverwaltung:

- Zusammenhängende Speicherzuteilung
  - Jede Anforderung eines Prozesses nach einer bestimmten Menge Speicher muss das BS durch zusammenhängenden (contiguous) Speicher erfüllen.
- Nicht-zusammenhängende Speicherzuteilung
  - BS kann Speicheranforderung durch Zuweisung mehrerer kleiner Speicherbereiche erfüllen, die zusammen die geforderte Menge Speicher ergeben.
  - Wiederauffinden der verstreuten Speicherbereiche ist eine Aufgabe der Speicherverwaltung.

Arten der Speicherverwaltung

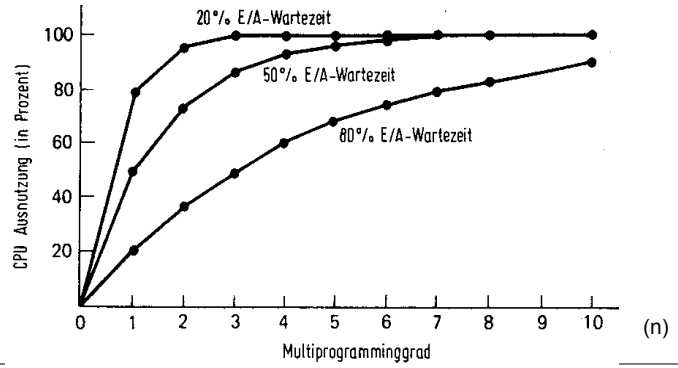
- Heute: Hauptspeicher fast immer nicht-zusammenhängend (virtuelle Speicherverwaltung)
- Zusammenhängende Speicherverwaltung bei
  - Verwaltung von Plattenplatz,
  - Verwaltung des Platzes in Page- und Swap-Dateien / -Partitionen

Multiprogramming

- Programme verbringen einen großen Teil ihrer Ausführungszeit mit Warten (auf I/O etc.).
- Auslagern (Swapping) auf Platte bei jedem Warten ist ineffizient.
- Lösung: mehrere Programme gleichzeitig im Speicher.
- Voraussetzung: verschiebbarer Code und Speicherschutz.

Multiprogramming

- Wenn *n* Programme den Bruchteil *p* ihrer Zeit mit Warten verbringen, ist die CPU-Ausnutzung = $1-p^n$.



Code-Verschiebung und Speicherschutz (1/2)

- Ein Programm muss an verschiedenen Stellen im Speicher laufen können.

  Zwei Möglichkeiten:
  - Linker vermerkt, welche Code-Stellen absolute Adressen sind. Beim Laden des Programms werden diese Stellen entsprechend abgeändert.
  - Der Rechner hat ein spezielles Hardware-Register, ein sog. **Basisregister**:
    Bei jeder Adressberechnung (zur Laufzeit) wird die Adresse im Basisregister zu der Adresse im Programm addiert.

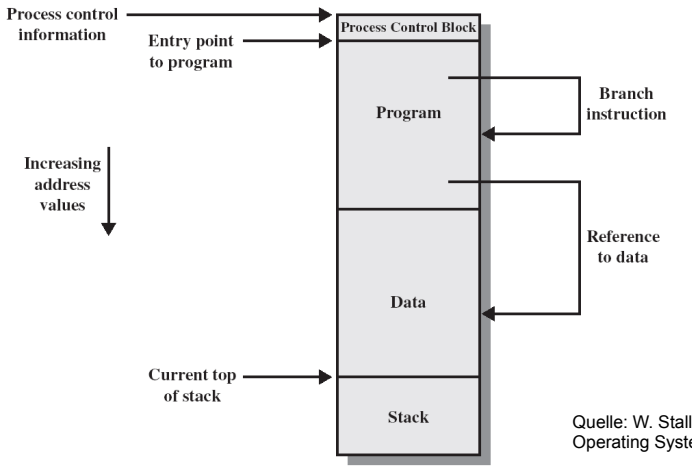
Code-Verschiebung und Speicherschutz (2/2)

- Ein Programm darf nicht auf den Speicherbereich eines anderen Programms zugreifen.

  Zwei Möglichkeiten, dies zu erreichen:
  - Schutzcode
  - Der Rechner hat ein spezielles Hardware-Register, ein sog. Längenregister

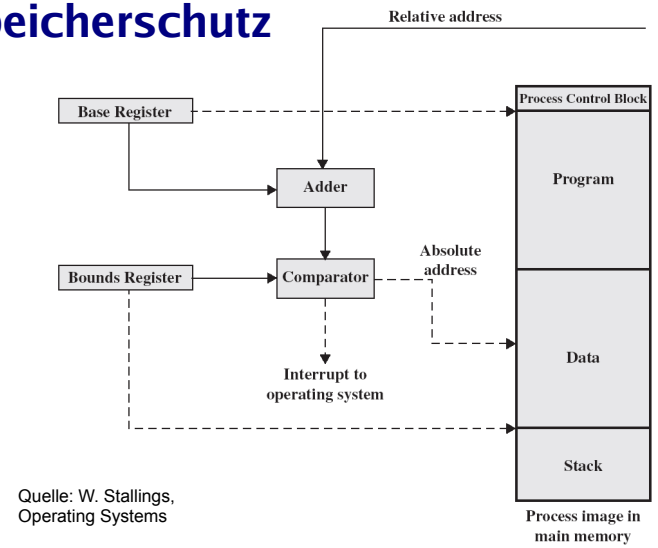
    Durch Überprüfen des Längenregisters wird festgestellt, ob die zugegriffene Adresse in der dem Programm zugewiesenen Partition liegt.

Code-Verschiebung: Prozess - Adressierungsanforderungen



Quelle: W. Stallings, Operating Systems

Unterstützung für Code-Verschiebung und Speicherschutz



Quelle: W. Stallings, Operating Systems

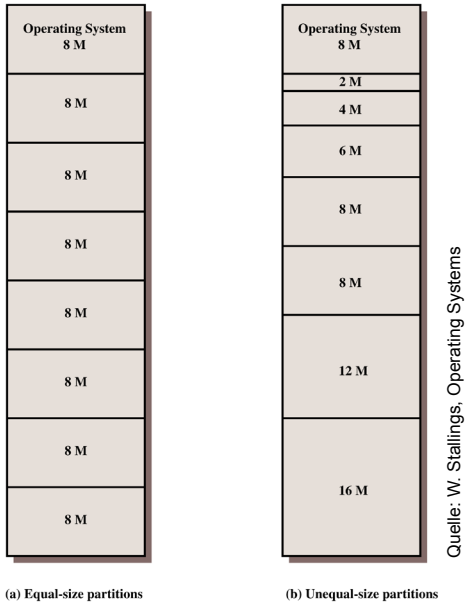
6.1 Zusammenhängende Speicheraufteilung

Aufteilung in feste Partitionen

- Aufteilung des Speichers in Partitionen fester (gleicher oder ungleicher) Größe.
- Zuweisung eines Programms zu einer freien Partition. Alternativen:
  - Erstes Programm, das in die freie Partition passt (eine Warteschlange)
  - FIFO für jede einzelne Partition (mehrere Warteschlangen)
  - Größtes Programm, das in die freie Partition passt
    - Nachteil: kleine Programme werden zurückgestellt
    - Lösung: kleines Programm nur k-mal überspringen

Aufteilung in feste Partitionen

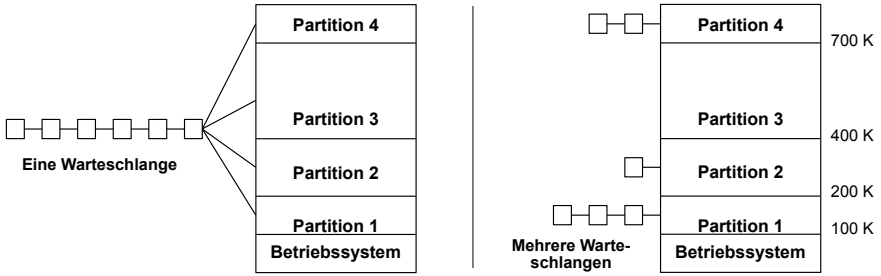
- **Gleiche Größe**
  - Verschwendung bei kleinen Programmen
  - Programme passen in jede freie Partitition
- **Ungleiche Größe**
  - Bessere Speicher-ausnutzung
  - Evtl. ungeschickte Belegung



Quelle: W. Stallings, Operating Systems

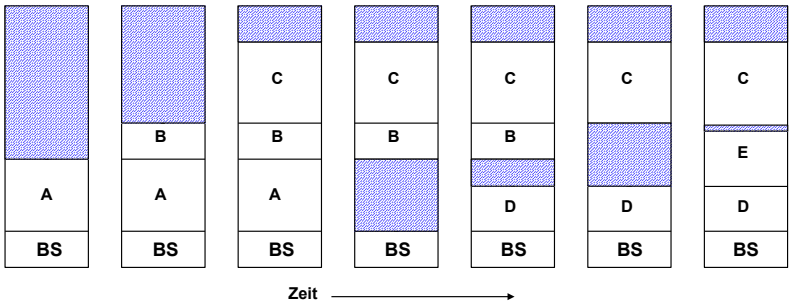
Aufteilung in feste Partitionen

- Evtl. große freie Bereiche in der Partition: **interne Fragmentierung**



Aufteilung in variable Partitionen (1)

- Anzahl und Größe der Partitionen werden dynamisch festgelegt.



Aufteilung in variable Partitionen (2)

- Es bleiben evtl. viele kleine freie Bereiche im Hauptspeicher (Löcher)
  → **externe Fragmentierung**
- Evtl. müssen diese Löcher durch Verschieben der Partitionen entfernt werden
  → **memory compaction**

Aufteilung in variable Partitionen (3)

Was ist, wenn der Speicherbedarf eines Prozesses wächst?

- Wenn neben der Partition ein freier Speicherbereich liegt, kann die Partition vergrößert werden.
- Es kann eine neue, ausreichend große Partition reserviert und der Inhalt dorthin verschoben werden.

Aufteilung in variable Partitionen (4)

- Wenn es keine ausreichend große Partition gibt, müssen ein oder mehrere Prozesse auf Platte ausgelagert werden (**Swapping**)
- Es kann auch dem Prozess von vornherein ein größerer Speicherbereich zugewiesen werden, als angefordert
  - interne Fragmentierung
  - Wenn auch dieser größere Bereich nicht ausreicht, muss wieder eines der vorher beschriebenen Verfahren angewandt werden

Zuteilung freien Speichers

Verschiedene Strategien für Speicherzuteilung:

- **First-Fit**
  - ersten ausreichend großen freien Speicherbereich zuordnen
- **Best-Fit**
  - kleinsten ausreichend großen freien Speicherbereich zuordnen
  - aufwendiger, da man die gesamte Liste durchsuchen muss
  - starke Fragmentierung des freien Speichers in viele kleine Bereiche

Zuteilung freien Speichers

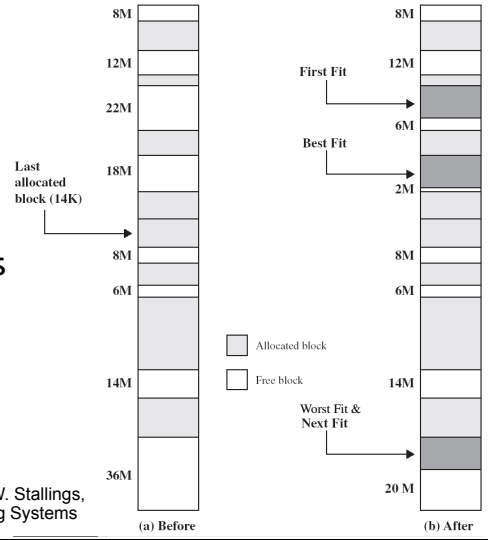
- **Worst-Fit**
  - größten freien Speicherbereich zuteilen
  - Es bleiben verhältnismäßig große freie Bereiche übrig
- **Quick-Fit**
  - Verwalten mehrerer Listen freier Speicherbereiche für bestimmte Standardgrößen (z. B. 4 KByte, 8 KByte, 12 KByte etc.) führt zu schneller Zuteilung
  - Bei Freigabe eines Bereiches muss dieser mit evtl. benachbarten freien Bereichen zusammengelegt werden. Dies bedeutet bei mehreren Listen einen erhöhten Aufwand

Zuteilung freien Speichers: Beispiel

Beispiel für eine Speicherbelegung

- vor und
- nach

der Allozierung eines 16-MByte-Blocks



Quelle: W. Stallings, Operating Systems

Betriebssysteme I – Sommersemester 2011
Kapitel 6: Speicherverwaltung

Hans-Georg Eßer
Hochschule München

6.2 Nicht-zusammenhängende Speicherzuordnung

06/2011

Ausgangslage

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

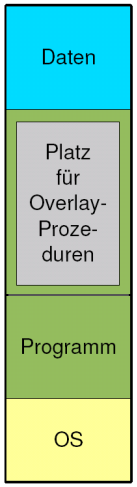
Overlay-Programmierung

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.

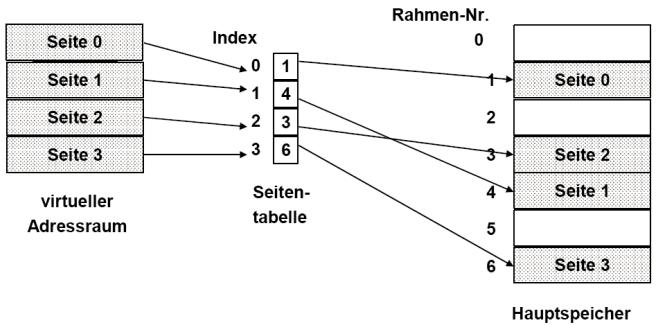
  Typische Seitengrößen: 512 – 8192 Byte (immer Zweierpotenz).
- Der lineare, zusammenhängende Adressraum eines Prozesses („virtueller“ Adressraum) wird auf beliebige, nicht zusammenhängende Seitenrahmen abgebildet.
- Eine einzige Liste freier Seitenrahmen wird vom Betriebssystem verwaltet.

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.



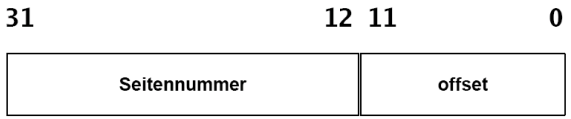
Virtueller Adressraum (2)

- 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

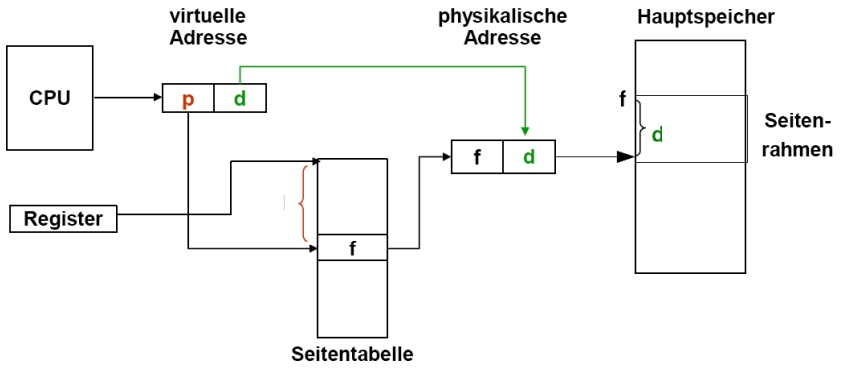
Beispiel: 32-bit-Adresse bei einer Seitengröße von 4096 ($=2^{12}$) Byte:



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:

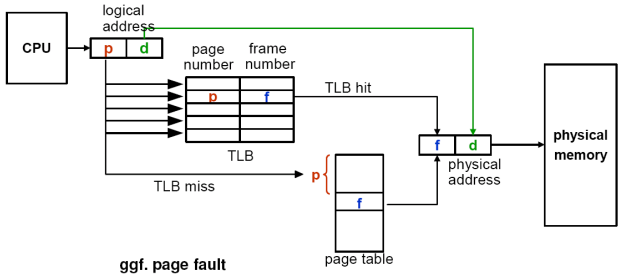
- Nur Teile des Prozesses befinden sich im physikalischen Speicher
- falls Zugriff auf eine Adresse, die ausgelagert ist:
  - BS setzt den Prozess auf blockiert
  - BS setzt eine Disk-I/O-Leseanfrage ab
  - Nach Laden des fehlenden Stücks (Seite oder Segment) wird ein I/O-Interrupt abgesetzt
  - das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

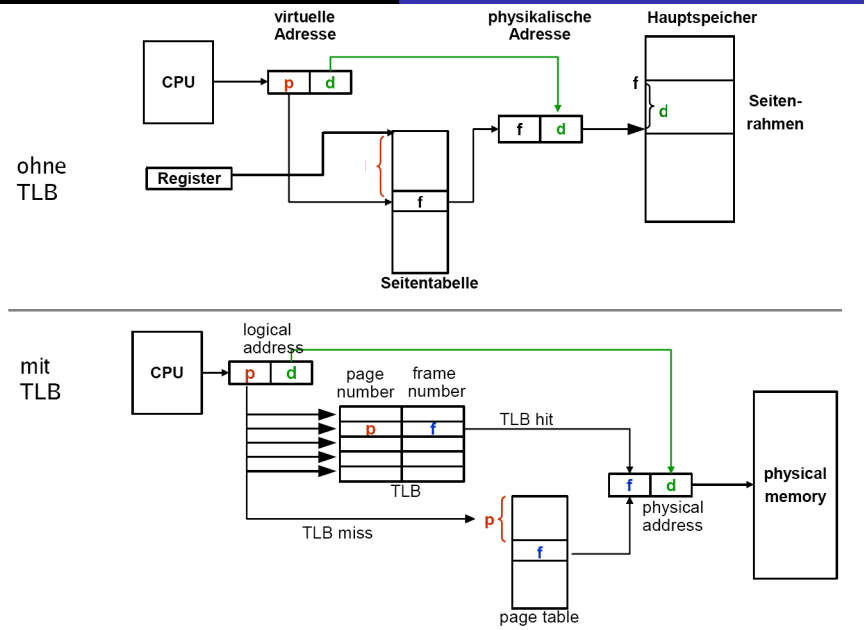
Virtueller Speicher allgemein (3)

- „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

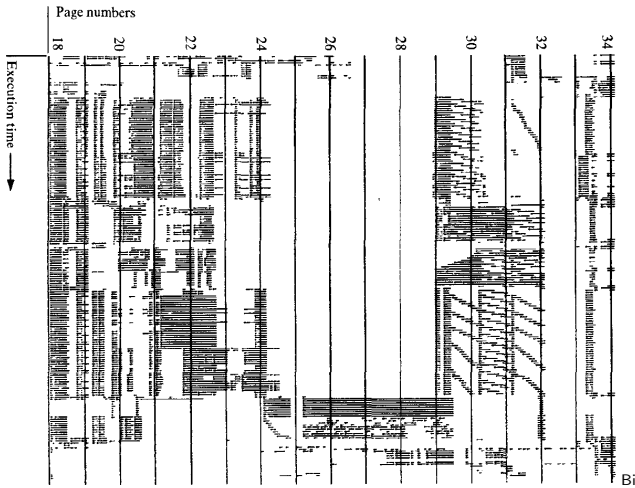


Bild: Hatfield (1972)

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? ($\rightarrow$ 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

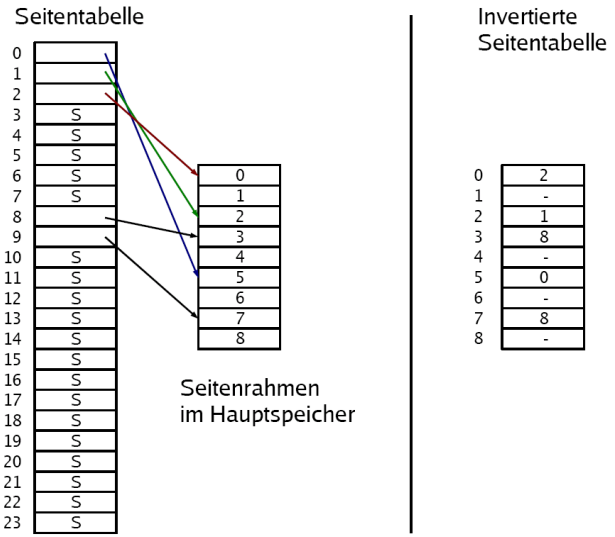
Invertierte Seitentabellen (1)

- Bei großem virtuellen Speicher sehr viele Einträge in der Seitentabelle nötig, z.B. 2^{32} Byte Adressraum, 4 KByte/Seite
  $\rightarrow$ 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 $\rightarrow$ 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 ($\rightarrow$ gleich)

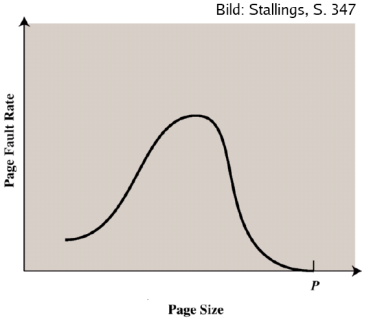
Invertierte Seitentabellen (3)



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)

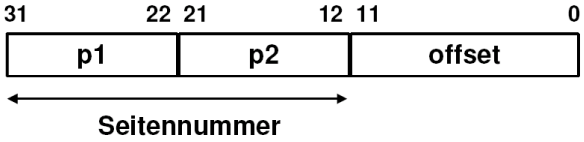
Die Seitentabelle kann sehr groß werden.
Beispiel:

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

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

Mehrstufiges Paging (2)

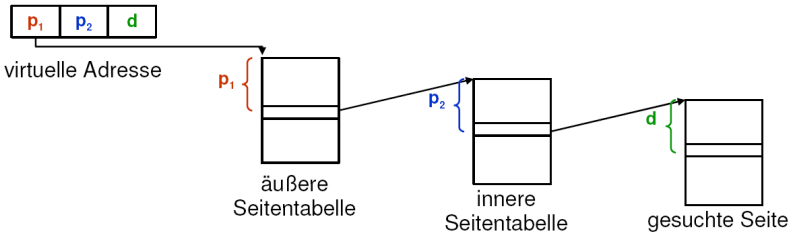
- Zweistufiges Paging:
  - Seitennummer noch einmal unterteilen, z. B.:



  - 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).

Demand Paging (1)

- Der Adressbereich eines Prozesses muss nicht vollständig im Hauptspeicher sein.
  - Das Lokalitätsprinzip besagt, dass ein Prozess in einer Zeitspanne nur relativ wenige, nahe beieinanderliegende Adressen anspricht.
  - Teile des Programms werden bei einem bestimmten Ablauf möglicherweise gar nicht benötigt (Spezialfälle, Fehlerbehandlungsroutinen etc.).

Demand Paging (2)

- Demand Paging bedeutet:
  - dass eine Seite nur dann in den Speicher geladen wird, wenn der Prozess sie anspricht,
  - dass eine Seite auch wieder aus dem Speicher entfernt werden kann.
- Vorteile von Demand Paging:
  - Der Adressbereich eines Prozesses kann größer sein als der physikalische Hauptspeicher.
  - Prozesse belegen weniger Platz im Hauptspeicher, somit können mehr Prozesse gleichzeitig aktiv sein.

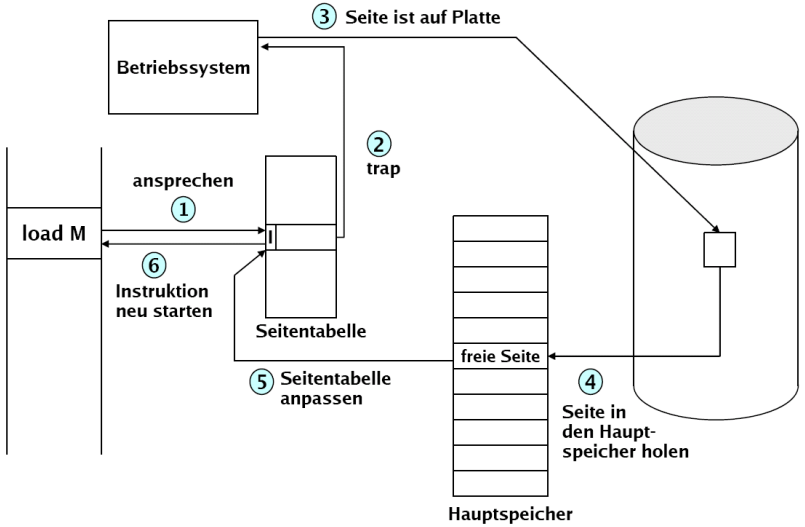
Voraussetzungen für Demand Paging (1)

- Jeder Eintrag in der Seitentabelle enthält ein valid bit, das angibt, ob die Seite im Speicher ist oder nicht.
- Wenn ein Prozess eine Seite anspricht, die nicht im Speicher ist, wird eine spezielle Exception ausgelöst, ein sog. page fault.
- Eine Betriebssystem-Routine, der page fault handler, lädt bei einem page fault die benötigte Seite in den Speicher.

Voraussetzungen für Demand Paging (2)

- Falls kein freier Seitenrahmen im Speicher vorhanden ist, muss eine andere Seite ersetzt werden. Für die Auswahl der zu ersetzenden Seite muss eine Strategie implementiert werden.
- Die durch den page fault unterbrochene Instruktion muss erneut ausgeführt werden (können).

Page-Fault-Behandlung



Seitenersetzung (1)

- Wenn bei einem Page Fault kein freier Seitenrahmen zur Verfügung steht, muss das Betriebssystem einen frei machen.
- Ein Algorithmus wählt nach einer bestimmten Strategie diesen Seitenrahmen aus.

Seitenersetzung (2)

- Falls die zu ersetzende Seite, seit sie zuletzt in den Speicher geholt wurde, verändert wurde, muss ihr aktueller Inhalt gesichert werden:
  - Ein modify bit (oder dirty bit) im Seitentabelleneintrag vermerkt, ob die Seite verändert wurde.
  - Eine veränderte Seite wird auf Platte gesichert (im sog. Page- oder Swap-Bereich).

Seitenersetzung (3)

- Eine unveränderte Seite kann später – bei Bedarf – wieder von der alten Stelle auf der Platte geladen werden.
- Im Seitentabelleneintrag für die ersetzte Seite wird
  - das valid bit gelöscht,
  - vermerkt, von wo die Seite wieder geladen werden kann.

Seitenersetzungsstrategien (1)

- Ziel: Es sollen so wenig Page Faults wie möglich auftreten.
- Zwei prinzipielle Arten von Seitenersetzungsstrategien:
  - lokale Ersetzung
  - globale Ersetzung

Seitenersetzungsstrategien (2)

Lokale Ersetzung: Es wird immer eine Seite desjenigen Prozesses ersetzt, der eine neue Seite anfordert.

- Die Zahl der Seiten, die ein Prozess im Speicher belegen kann, ist nach oben beschränkt. Die maximale Anzahl wird pro Prozess festgelegt (z. B. vom Systemverwalter) und kann die Laufzeit eines Prozesses stark beeinflussen.
- Ein Prozess, der viele Page Faults verursacht, z. B. weil er sich nicht an das Lokalitätsprinzip hält, beeinträchtigt nur sich selbst, nicht aber das Gesamtsystem.

Seitenersetzungsstrategien (3)

Globale Ersetzung: Es wird eine beliebige Seite im Speicher ersetzt.

- Prozesse nehmen sich gegenseitig Seiten weg.
- Ein Prozess, der viele Page Faults verursacht, erhält automatisch mehr Speicher. (Dies kann sowohl ein Vorteil als auch ein Nachteil sein.)

Optimale Strategie

Diejenige Seite ersetzen, auf die in Zukunft am längsten nicht zugegriffen wird.

- Vorteil: Diese Strategie verursacht die kleinste Zahl an Page Faults.
- Nachteil: Diese Strategie ist nicht implementierbar. (vgl. idealer Scheduler)

Die optimale Strategie kann modellhaft zur Bewertung anderer Strategien benutzt werden.

First In First Out (FIFO)

Die Seite ersetzen, die schon am längsten im Speicher ist.

- Vorteil: Sehr einfach zu implementieren:
  - Es wird eine verkettete Liste der Seiten im Speicher (globale Strategie) bzw. der Seiten eines Prozesses (lokale Strategie) unterhalten.
  - Bei einem Page Fault wird die erste Seite der Liste ersetzt und die neue Seite ans Ende der Liste angefügt.
- Nachteil: Die ersetzte Seite kann in dauernder Benutzung sein und gleich wieder angefordert werden.

Least Recently Used (LRU) (1)

Die Seite ersetzen, die am längsten nicht benutzt worden ist.

- Vorteil: In der Regel weniger Page Faults als FIFO.
- Nachteil: Aufwändige Implementierung.

Zwei mögliche Implementierungen:

- mit Zähler
- mit verketteter Liste

Least Recently Used (LRU) (2)

Implementierung mit Zähler:

- Systemweiten Zähler bei jedem Speicherzugriff inkrementieren.
- Aktuellen Wert des Zählers in einem Feld in der Datenstruktur vermerken, welche die angesprochene Seite beschreibt.
- Seite mit dem kleinsten Zählerwert ersetzen.

Least Recently Used (LRU) (3)

Implementierung mit verketteter Liste:

- Eine verkettete Liste enthält alle Seiten.
- Bei jedem Speicherzugriff wird die angesprochene Seite an den Anfang der Liste gebracht. (Liste durchsuchen und Reihenfolge ändern, also Zeiger umsetzen!)
- Die Seite am Ende der Liste wird ersetzt.

Benutzen eines Referenz-Bits (1)

- Jeder Seitentabelleneintrag kann ein Referenz-Bit enthalten
  - das bei einem Zugriff auf die Seite gesetzt wird (Hardware),
  - das nach bestimmten Kriterien wieder gelöscht wird (Software).
- Ein Referenz-Bit
  - liefert die Information, ob auf eine Seite seit dem letzten Löschen des Bits zugegriffen wurde,
  - sagt nichts über den Zeitpunkt des Zugriffs auf eine Seite aus,
  - sagt nichts über die Reihenfolge der Zugriffe auf mehrere Seiten aus.

Benutzen eines Referenz-Bits (2)

- Mit Referenz-Bits kann man weitere Seitenersetzungsstrategien implementieren, z. B.
  - Modifikationen von LRU, die weniger aufwändig sind.
  - Second-Chance-Algorithmus, eine Verbesserung der FIFO-Strategie.

Modifikation von LRU (1)

- Es wird ein binärer Zähler für jede Seite unterhalten.
- In regelmäßigen Abständen wird
  - jeder Zähler eine Position nach rechts geschoben („Aging“),
  - das Referenzbit in das höchste Bit des Zählers kopiert,
  - das Referenzbit gelöscht.

Modifikation von LRU (2)

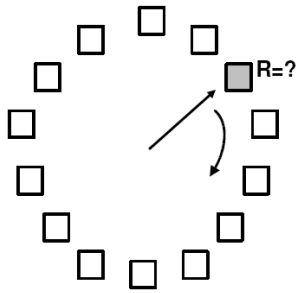
- Es wird eine der Seiten ersetzt, die den kleinsten Zählerwert enthalten.
  - Bei gleichem Zählerwert ist nicht bekannt, auf welche Seite zuletzt zugegriffen wurde.
  - Länger zurückliegende Zugriffe werden zunächst weniger stark gewichtet und schließlich „vergessen“ (aus dem Zähler hinausgeschoben).

Second-Chance-Algorithmus (1)

- Modifikation des FIFO-Algorithmus: Ist bei der Seitenersetzung das Referenz-Bit der ältesten Seite gesetzt, so wird
  - das Referenz-Bit gelöscht und die Seite am Ende der Liste eingereiht,
  - die gleiche Prüfung für die nächstälteste Seite durchgeführt.
- Es wird also
  - die älteste Seite ersetzt, deren Referenz-Bit gelöscht ist,
  - einer kürzlich benutzten Seite zunächst eine „zweite Chance“ gegeben.

Second-Chance-Algorithmus (2)

- Einfachere Implementierung: „Uhrzeiger“
  - Anordnung der Seiten in einer Ringliste, und Verschieben eines Zeigers statt Umpositionieren eines Listenelements.



Überprüfen der Seite, auf die der Zeiger zeigt:

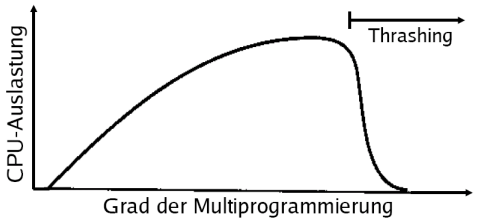
- Wenn $R = 0$: Seite ersetzen, Zeiger weiter bewegen
- Wenn $R = 1$: R löschen, Zeiger weiter bewegen, nächste Seite prüfen

Seitenersetzung, Beispiele

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2 3	2 3	2 3 1	2 3 5 F	2 3 5	4 3 5 F	4 3 5	4 3 5	2 3 5 F	2 3 5	2 3 5
LRU	2	2 3	2 3	2 3 1	2 5 1 F	2 5 1	2 5 4 F	2 5 4	3 5 4 F	3 5 2 F	3 5 2	3 5 2
FIFO	2	2 3	2 3	2 3 1	5 3 1 F	5 2 1 F	5 2 4 F	5 2 4	3 2 4 F	3 2 4	3 5 4 F	3 5 2 F
CLOCK	2*	2* 3*	2* 3*	2* 3* 1*	5* 3 1 F	5* 2* 1 F	5* 2* 4* F	5* 2* 4*	3* 2 4 F	3* 2* 4	3* 2 5* F	3* 2* 5*

Thrashing (1)

- Thrashing bedeutet, dass ein Prozess exzessiv viele Page Faults macht (alle paar tausend Instruktionen).
- Thrashing entsteht, wenn ein Prozess mehr Seiten aktiv benutzt, als ihm Seitenrahmen zur Verfügung stehen.
- Thrashing mehrerer Prozesse führt zu niedriger CPU-Auslastung:



Thrashing (2)

Lösungen

- Falls noch freier Speicher vorhanden: Zuteilung weiterer Seitenrahmen an den betreffenden Prozess (z. B. durch globale Ersetzungsstrategie).
- Falls kein freier Speicher mehr: Auslagern (Swapping) von Prozessen.