Speicherverwaltung

Gliederung

Speicherverwaltung

Übersicht:

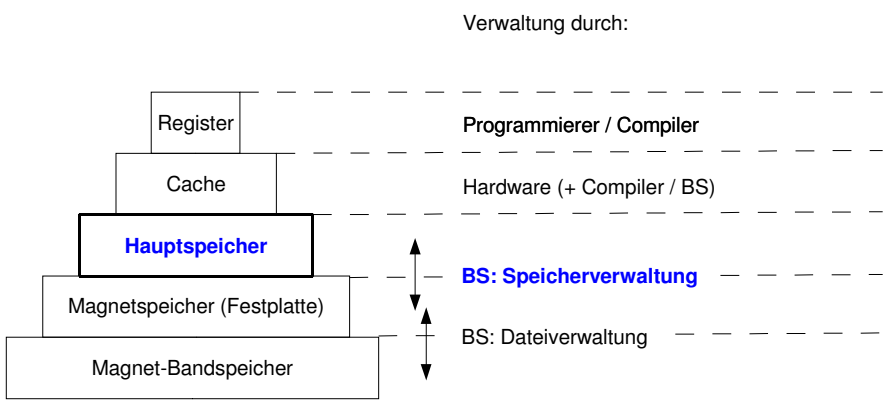
- **Aufgaben der Speicherverwaltung**
- **Code-Verschiebung, Speicherschutz**
- **Zusammenhängende Speicherzuteilung**
  - Partitionen fester / variabler Größe
  - Swapping
  - Methoden zur Verwaltung freien Speichers
  - Segmentierung
- **Nicht zusammenhängende Speicherzuteilung**
  - Virtuelle Speicherverwaltung (Paging)
  - Mehrstufiges Paging
  - Demand Paging
  - Page Faults und deren Behandlung
  - Strategien für die Seitenersetzung

Motivation

Motivierende Fragen:

- Wie setzt sich eine Speicheradresse zusammen ?
- Was geschieht bei einem Adreßzugriff ?
- Wie kann man als Administrator oder Softwareentwickler Nutzen aus Kentnissen der Speicherverwaltung ziehen ?
  - wie funktioniert „Shared Memory“ ?
  - was sind „Memory-mapped Files“ ?
- Wie entsteht ein „Segmentation Fault“ ?
- Wie erreicht man einen „virtuellen Adreßraum“ ?

Speicherhierarchie



Aufgaben

Welche Aufgaben übernimmt im BS die Speicherverwaltung ?

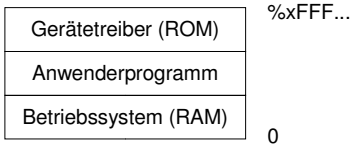
- **Finden und Zuteilung freier Speicherbereiche**
  z.B.
  - bei Erzeugung eines neuen Prozesses
  - bei Speicheranforderung durch bestehenden Prozess
- **Effiziente Nutzung des Speichers:** verschiedene Aspekte
  - gesamter freier Speicher sollte nutzbar sein
  - Prozess benötigt nicht immer alle Teile seines Adreßraumes
  - Adreßräume aller Prozesse evtl. größer als verfügbarer Hauptspeicher
- **Speicherschutz**
  - Threads eines Prozesses dürfen nur auf Daten dieses Prozesses zugreifen

Einfache historische Ansätze

Monoprogramming

Monoprogramming

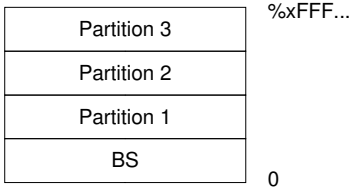
- **Aufteilung des (RAM + ROM) in Bereiche für**
  - Betriebssystem
  - Anwenderprogramm
- Beim Beenden eines Programms:
  - Überlagern des Programmbereichs durch neues Programm



Multiprogramming (1)

Multiprogramming, feste Partitionen

- Programme warten vergleichsweise häufig (I/O, etc.)
- Lösung: mehrere Programme gleichzeitig im Speicher
- Voraussetzung:
  - verschiebbarer Code und
  - Speicherschutz



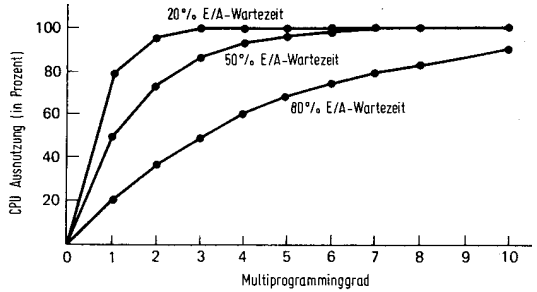
- Aufteilung des Hauptspeichers in Partitionen fester, ggf. unterschiedlicher Größe
- Zuweisung neuer Programme über Warteschlange für Speicherzuteilung
  - gemeinsam oder je Partition

Multiprogramming (2)

Multiprogramming, feste Partitionen

- Nachteile:
  - ineffiziente Speichernutzung
  - unflexibel, Grad des Multiprogramming starr vorgegeben
- wenn n Programme mit Wahrscheinlichkeit p zu einem Zeitpunkt warten:

CPU-Nutzung = $1 - p^n$

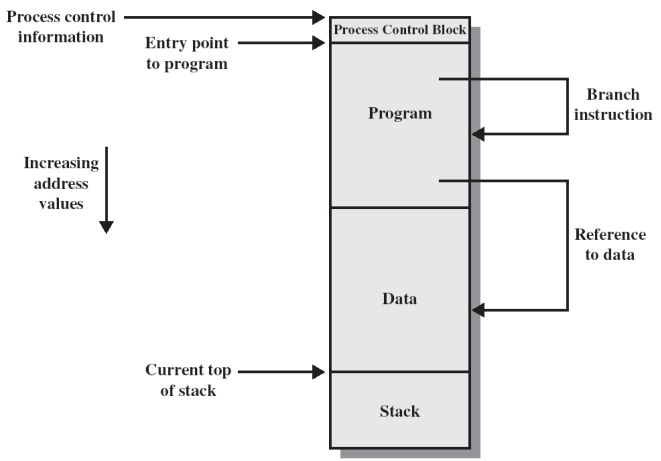


Code-Verschiebung

- Code-Verschiebung (Relokation)
  - 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.
    - 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: Adressierungsanforderungen

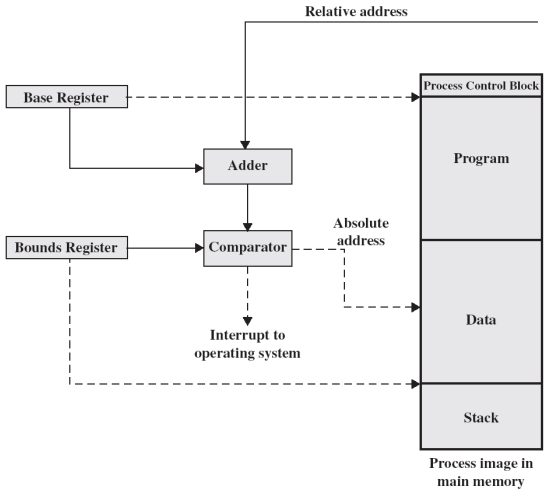


Speicherschutz

- Speicherschutz:
  - Programm darf nicht auf Speicherbereich anderer Programme zugreifen.
  - Zwei Möglichkeiten:
    - Schutzcode
    - Rechner nutzt spezielles Hardware - Längenregister

      Überprüfung, ob zugegriffene Adresse noch im Bereich der dem Programm zugewiesenen Partition liegt.

Unterstützung für Code-Verschiebung und Speicherschutz



Arten der Speicherverwaltung (1)

Zwei prinzipielle Arten der Speicherverwaltung:

- **Zusammenhängende Speicherzuteilung**
  - jede Anforderung nach einer Menge Speicher muss das BS durch zusammenhängenden (contiguous) Speicher erfüllen
- **Nicht-zusammenhängende Speicherzuteilung**
  - Speicheranforderung kann erfüllt werden durch
    - Zuweisung mehrerer kleiner Speicherbereiche,
    - die zusammen die geforderte Menge Speicher ergeben
  - Wiederauffinden der verstreuten Speicherbereiche muss Speicherverwaltung transparent leisten

Arten der Speicherverwaltung (2)

Heutzutage:

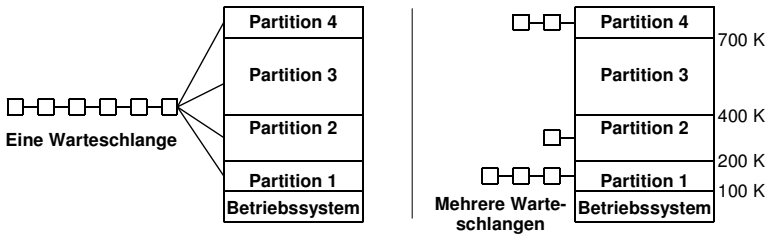
- Hauptspeicher fast immer nicht-zusammenhängend (virtuelle Speicherverwaltung)
- Zusammenhängende Speicherverwaltung bei
  - Verwaltung von Plattenplatz
  - Verwaltung des Platzes in Page- und Swap-Dateien / -Partitionen

Zusammenhängende

Speicheraufteilung

Aufteilung in feste Partitionen (1)

- Aufteilung des Speichers in Partitionen fester, ggf. unterschiedlicher Größe
- Zuweisung eines Programms zu einer freien Partition:
  Alternativen:
  - eine Warteschlange: erstes Programm, das in die freie Partition passt
  - FIFO je Partition (mehrere WS)

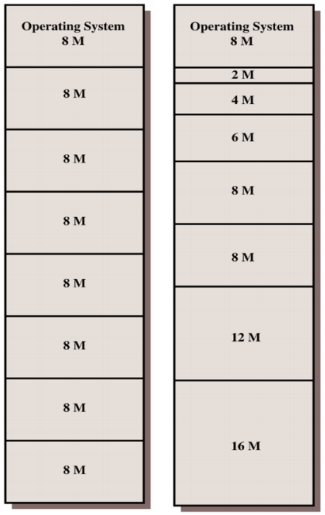


Aufteilung in feste Partitionen (2)

- gleiche Größe:
  - Verschwendung bei kleinen Programmen
  - Programme passen in jede freie Partition
- ungleiche Größe:
  - bessere Speicherausnutzung
  - evtl. ungeschickte Belegung
- Generell:

  evtl. große freie Bereiche in der Partition:

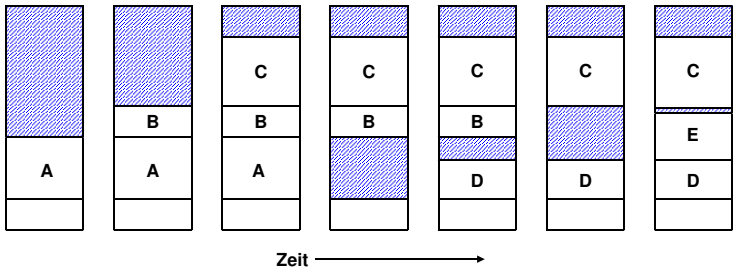
  Interne Fragmentierung



Quelle: W. Stallings, Operating Systems

Aufteilung in variable Partitionen (1)

- Anzahl und Größe der Partitionen werden dynamisch festgelegt:



- es bleiben evtl. viele kleine freie Bereiche (Löcher) im Hauptspeicher

  --> externe Fragmentierung
- evtl. müssen diese Löcher durch Verschieben der Partitionen entfernt werden

  --> memory compaction

Aufteilung in variable Partitionen (2)

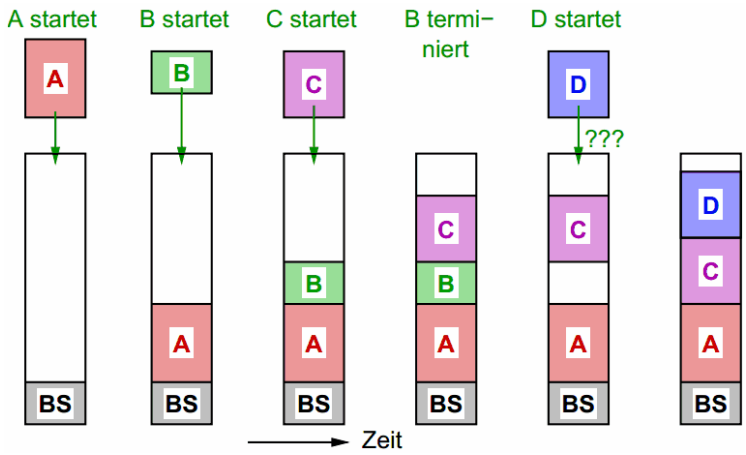
- Was, wenn der Speicherbedarf eines Prozesses wächst?
  - ggf. Partition über benachbarten freien Speicherbereich vergrößern
  - ggf. Reservierung und Verschiebung der Inhalte in neue, ausreichend große Partition (Relokation)
- einem Prozess kann auch ein größerer Speicherbereich als angefordert zugewiesen werden --> Vermeidung einer Verlagerung
- wenn keine ausreichend große Partition verfügbar: Swapping
  - Auslagerung einer oder mehrerer Prozesse auf Platte

Swapping (1)

- **Swapping:**
  - zeitweise Auslagerung aller Speicherbereiche eines Prozesses (oder zumindest kompletter Segmente) auf Festplatte
  - anschließende Suspendierung des Prozesses
  - später Wiedereinlagerung, ggf. an anderer Stelle im Speicher
- Bei zusammenhängender Speicherzuteilung
  - einzige Möglichkeit, mehr Programme gleichzeitig auszuführen, als Platz im Hauptspeicher vorhanden,
  - müssen Speicherbereiche eines Prozesses, die dynamisch wachsen, u.U. ausgelagert und an anderer Stelle eingelagert werden --> Relokation i.d.R. über Basisregister

Swapping (2)

Beispiel:



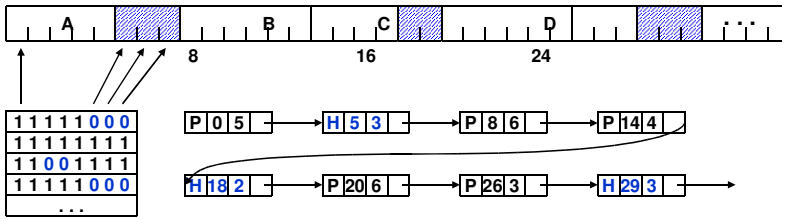
Swapping (3)

- Kriterien für die Auslagerung, z.B.:
  - Prozeßzustand
  - Prozeßpriorität
  - Prozeßgröße (im Hauptspeicher)
  - Zeit, die der Prozeß im Hauptspeicher war
- innerhalb des Swapbereichs auf Platte:
  - Zuteilung und Verwaltung des Platzes mit einem der Verfahren der zusammenhängenden Speicherverwaltung.
- Probleme:
  - Ein- und Auslagerung sehr aufwendig
  - Prozess kann zur Laufzeit mehr Speicher anfordern
    - benötigt evtl. Verschiebung oder Auslagerung mehrerer Prozesse
  - schlechte Speichernutzung: Prozess benötigt i.d.R. nicht seinen gesamten Adreßraum

besseres Verfahren: --> Paging

Dynamische Speicherverwaltung

- BS muss bei Prozesserzeugung oder Einlagerung passenden Speicherbereich finden
- --> BS benötigt Informationen. typisch: Liste aller belegten und freien Speicherbereiche
  - belegt oder frei (P/H=Hole)
  - Anfangsadresse
  - Länge
  - Zeiger auf nächsten Eintrag
- bei Speicherfreigabe: Verschmelzung der Einträge aneinandergrenzender freier Bereiche
  - --> doppelt verkettete Liste, sortiert nach Adressen



Anm.: Problem/Lösung identisch zur Verwaltung des Heaps

Zuteilung freien Speichers (1)

Suchverfahren / Strategien:

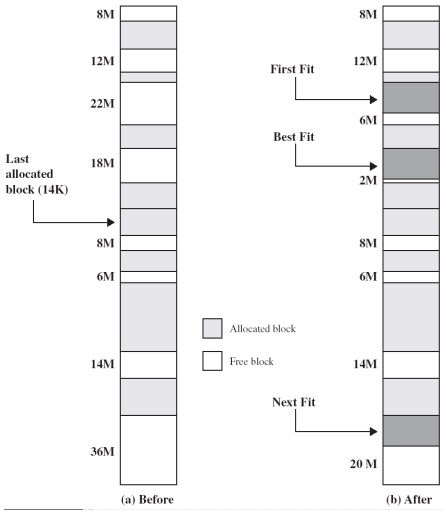
- **First Fit:**
  - verwende ersten freien Speicherbereich (einfach, relativ gut)
- **Best Fit:**
  - kleinsten ausreichenden freien Bereich zuordnen
  - aufwendiger, da gesamte Liste zu durchsuchen
  - starke Fragmentierung des freien Speichers in viele kleine Bereiche
- **Worst Fit:**
  - größten freien Bereich zuteilen
  - es bleiben verhältnismäßig große freie Bereiche übrig
- **Quick Fit:**
  - mehrere Listen für freie Bereiche mit gebräuchlichen Größen, z.B. 4kB, 8kB, 12kB, ...
  - schnelle Suche und Zuteilung möglich
  - Problem: Verschmelzen freier Bereiche aufwendig
  - Variante: Buddy-System

Zuteilung freien Speichers: Beispiel

Speicherbelegung

- **vor und**
- **nach**

der Allokation eines 16-MByte Bereiches



Zuteilung freien Speichers (2)

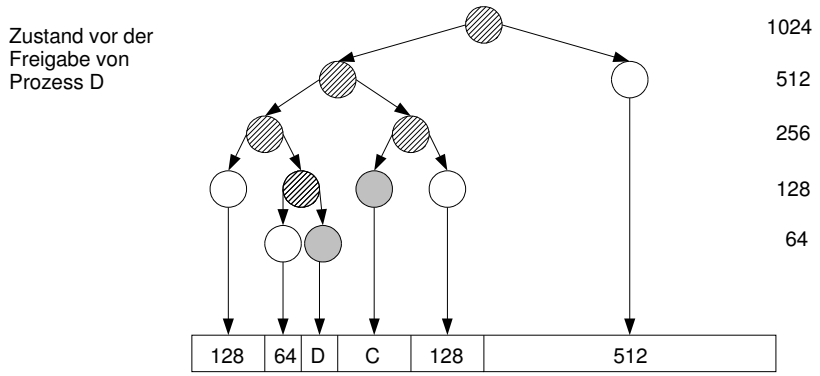
Buddy-System

- separate Listen freier Bereiche in 2er-Potenzen
- bei Freigabe eines Bereiches nur eine Liste zu durchsuchen, um Möglichkeit einer Verschmelzung freier Bereiche zu prüfen
- bei *Splits* und *Merges* umhängen in andere Liste

Anforderungen:	0	128K	256K	384K	512K	640K	768K	896K	1M
anfangs	1024								
70	A	128	256		512				
35	A	B	64	256		512			
80	A	B	64	C	128	512			
Freigabe A	128	B	64	C	128	512			
60	128	B	D	C	128	512			
Freigabe B	128	64	D	C	128	512			
Freigabe D	256			C	128	512			
Freigabe C	1024								

Zuteilung freien Speichers (3)

Buddy-System: Zustandsbaum



Segmentierung (1)

Ausgangslage:

- große Programme ein-/auslagern zu aufwendig
- ggf. zuwenig Speicher für große Programme vorhanden
  - Zerlegung von Programmen in **Segmente**
  - Nachladen von Segmenten (Programmteilen) bei Bedarf
    - Bsp.: Overlay-Programmierung (noch in Verantwortung des Programmierers)
    - Bsp.: Aufteilung in Code-, Daten- und Stacksegment(e)
- Adressberechnung von Segmenten über Segmentadresstabelle und Register in Hardware
- Segmente noch zu groß / unflexibel --> heutzutage: Paging

Segmentierung (2)

Overlay-Programmierung:

- Turbo-Pascal (1985-1990):

```
program GrossProjekt;

overlay procedure kundendaten;
overlay procedure lagerbestand;
...

begin
    while input <> “exit“ do
begin
        case input of
            1: kundendaten();
            2: lagerbestand();
        end;
    end;
end.
```

Nicht - zusammenhängende Speicheraufteilung

Virtuelle Speicherverwaltung

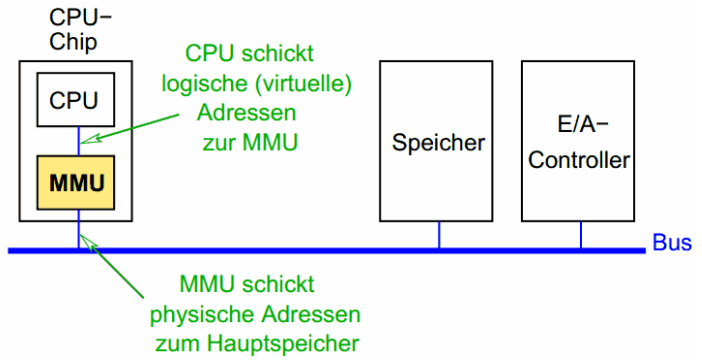
Virtueller Adressraum (1)

- **Virtueller Adreßraum:**
  - strikte Trennung zwischen
    - logischen (virtuellen) Adressen, die der Prozess sieht / verwendet,
    - physischen Adressen, die der Hauptspeicher sieht
  - Idee:
    - **bei jedem Speicherzugriff** wird die vom Prozess erzeugte logische Adresse auf eine physische Adresse abgebildet
    - durch Hardware: **MMU** (Memory Management Unit)
  - Vorteile:
    - keine Code-Verschiebung beim Laden eines Prozesses erforderlich
    - Prozess „sieht“ transparent seinen linearen zusammenhängenden Speicherbereich, unabhängig davon, wie dieser auf Hardware abgebildet wird

Anm.: auch mit Segmenten realisierbar

Virtueller Adressraum (2)

Ort und Funktion der MMU im Rechner:



Paging

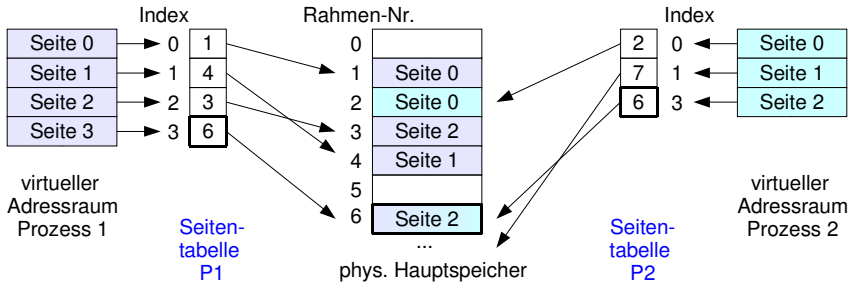
Paging: Seitenbasierte Speicherabbildung

- Aufteilung der Adreßräume in Blöcke fester Größe:
  - **Seite** (***Page***): Block im virtuellen Adressraum
  - **Kachel** (**Seitenrahmen** / ***page frame***): Block im physischen Adressraum
  - typische Seitengrößen: 512-8192 Byte
- Abbildung
  - des linearen zusammenhängenden (virtuellen) Adressraums auf beliebige, nicht zusammenhängende Seitenrahmen
- Dadurch
  - nur eine Liste freier Seitenrahmen vom BS zu verwalten
  - einfache Zuteilung von Hauptspeicher (alle Blöcke gleich groß)
  - keine externe, geringe interne Fragmentierung

Paging: Adressumsetzung (1)

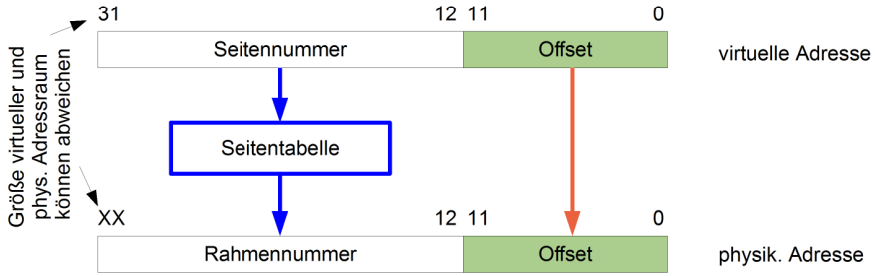
Paging: Adressumsetzung

- Berechnung virtuelle Programmadresse --> physikalische Hauptspeicheradresse
  - transparent (kein Aufwand für Programmierer)
  - zur Laufzeit
  - muss von Hardware unterstützt werden
  - mittels Umsetzungstabelle (**Seitentabelle** / ***page table***) je Prozess



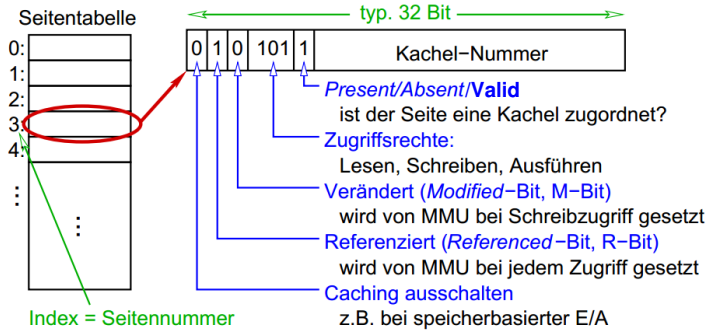
Paging: Adressumsetzung (2)

- Programmadresse besteht aus zwei Teilen:
  - Seitennummer
  - relative Adresse (Offset) innerhalb der Seite
- Beispiel: 32-bit-Adresse bei einer Seitengröße von 4096 (=2^{12}) Byte

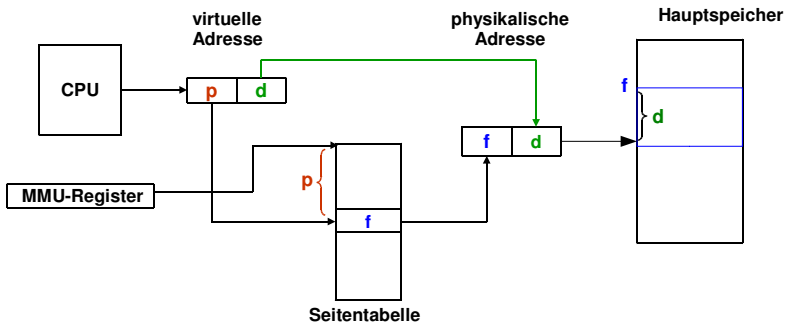


Paging: Adressumsetzung (3)

- Seitennummer ist Index in Seitentabelle
- typischer Aufbau der Einträge:



Paging: Adressumsetzung (4)



- spezielles Register (MMU) enthält Anfangsadresse der Seitentabelle für aktuellen Prozess
- je Hauptspeicherzugriff ein zusätzlicher Hauptspeicherzugriff auf Seitentabelle notwendig
- dies muss durch Hardware-Caches beschleunigt werden

Paging: Adressumsetzung (5)

Adressumsetzung: Zusammenspiel zwischen BS und MMU

- **BS** setzt für jeden Prozess eine Seitentabelle auf
  - Startadresse wird bei Prozesswechsel im Register gesetzt
- **MMU** realisiert Adressumsetzung
  - falls Seite nicht im Speicher steht, ihr also kein Rahmen zugeordnet ist, ist das *Present*-Bit gelöscht:
    - --> MMU erzeugt Ausnahmebehandlung/Exception (--> **Seitenfehler / Page Fault**)
    - --> ausgelagerte Seite wird vor Zugriff eingelagert
  - Speicherschutz automatisch gegeben, da Seitentabelle nur auf prozesseigene Rahmen verweist
  - zusätzlich: Zugriffsrechte für einzelne Seiten
    - z.B. Schreibschutz für Programmcode, bei Verletzung: Ausnahmebehandlung

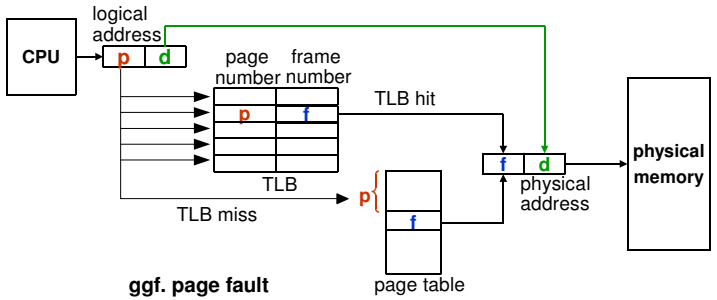
Translation Look-Aside Buffer (1)

- **Translation Lookaside Buffer (TLB)**:

  schneller Hardware-Cache, mit den zuletzt benutzten Einträgen der Seitentabelle

- Assoziativ-Speicher:

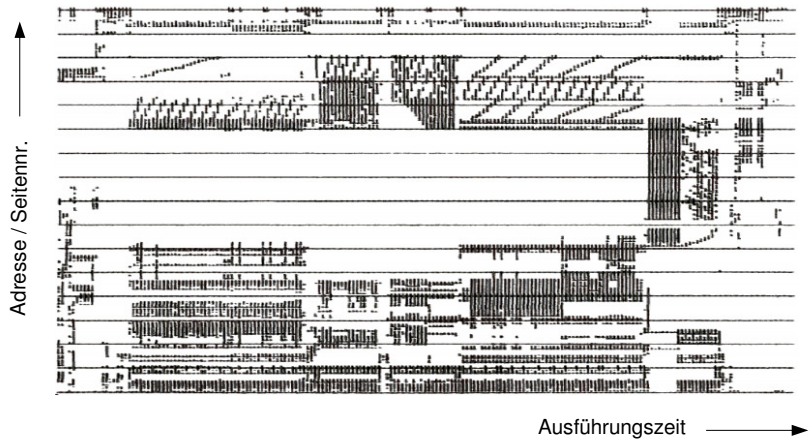
  bei Übersetzung einer Adresse wird deren Seitennummer (Index) gleichzeitig mit allen Einträgen des TLB vergleichen



Translation Look-Aside Buffer (2)

- Treffer im TLB --> Speicherzugriff auf Seitentabelle unnötig
- Fehltreffer --> Zugriff auf Seitentabelle und alten Eintrag im TLB aktualisieren
- Trefferquote (*hit ratio*) beeinflusst durchschnittliche Zeit einer Adreßübersetzung
- **Lokalitätsprinzip**: Programme greifen meist auf benachbarte Adressen (in gleicher Seite) zu --> auch bei kleinen TLBs hohe Trefferquoten (typisch 80-98 %)
- Inhalt des TLB prozessspezifisch! Zwei Möglichkeiten:
  - jeder Eintrag im TLB enthält „valid bit“.
    Bei Prozesswechsel (Context Switch) wird gesamter Inhalt des TLB invalidiert
  - jeder Eintrag im TLB enthält PID, die mit PID des zugreifenden Prozesses verglichen wird

Lokalitätsprinzip



Translation Look-Aside Buffer (3)

- Beispiel: AMD-Opteron:
  - Speicher-Hierarchie:
    - 64 KB L1 Instruction Cache, 64 KB L1 Data-Cache
    - 512 KM L2 Cache, 2 MB L3 Cache
    - mittlere L2 miss-latency: 100 Cycles
  - Data TLB:
    - 64-entry, fully associative L1 D-TLB
    - 512-entry, 4-way L2 D-TLB
  - Instruction TLB:
    - 32-entry, fully associative L1 I-TLB
    - 512-entry, 4-way L2 I-TLB

Paging: Adressumsetzung (6)

Was macht das BS ?

- Page-Table-Register laden (bei Prozesswechsel)
- bei Page-Fault: Seite einlagern und Seitentabelle aktualisieren
- evtl. vorher: Seitenverdrängung: welche Seite aus dem Hauptspeicher verdrängen ? (--> später)

Alles andere in Hardware:

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

Mehrstufiges Paging (1)

Problem: zu große Seitentabellengrößen

- Beispiel 1:
  - 32-bit virtuelle Adressen mit
  - 4 kByte Seitengröße (== 12 Bit)
  - 4 Byte je Tabelleneintrag
  - --> 2^{20} Seiten
  - --> 4 MByte Tabellengröße (pro Prozess !)
- Beispiel 2:
  - bei 64-Bit virtuellen Adressen:
  - --> 2^{52} Seiten
  - --> 16 PByte (PetaByte)

Aber ...

- Prozess nutzt virtuellen Adressraum i.a. nicht vollständig

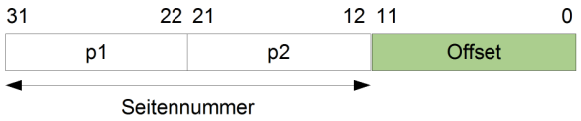
  --> mehrstufige Seitentabellen sinnvoll:

  z.B. für 2^{20} Seiten:
  - 1. Stufe: Hauptseitentabelle mit 1024 Einträgen
  - 2. Stufe: <= 1024 Seitentabellen mit je 1024 Einträgen

Mehrstufiges Paging (2)

Zweistufiges Paging:

- Seitennummer in virtueller Adresse nochmals unterteilen, z.B.

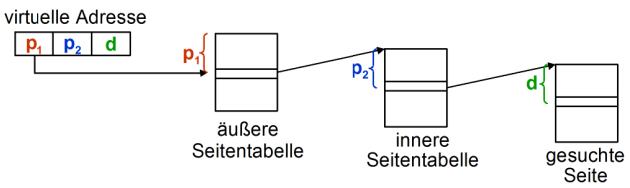


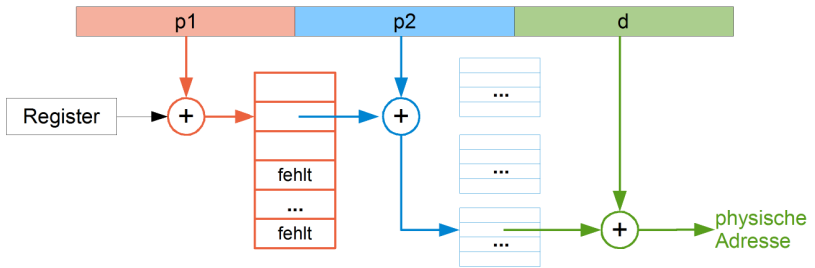
- p1: Index in äußere Seitentabelle, deren Einträge zeigen auf jeweils eine innere Seitentabelle
- p2: Index in eine der inneren Seitentabellen, deren Einträge auf Seitenrahmen im Speicher zeigen
- die inneren Seitentabellen
  - brauchen nicht speicherresident zu sein
  - sind nur vorhanden, falls notwendig !

Analog dreistufiges Paging etc. zu implementieren

Mehrstufiges Paging (3)

Zweistufiges Paging:





Mehrstufiges Paging (4)

- **Größe der Seitentabellen:**
  - Beispiel 32-Bit-Adresse: p1 (10 Bit), p2 (10 Bit), d (12 Bit):
    - bei jeweils 1024 Einträgen und 4 Byte/Eintrag:
    - --> Seitentabellen sind 4 kByte groß, genau passend zur Größe eines Seitenrahmens
- **Zahl der Speicherzugriffe:**
  - jede Adressübersetzung benötigt noch mehr Speicherzugriffe:
    - --> der Einsatz von TLBs noch wichtiger
    - --> als Schlüssel für TLB dienen alle Teile der Seitennummer (p1, p2, ...)

Speicherschutz

- **Schutz vor Zugriff anderer Prozesse:**
  - jeder Prozess hat eigene Seitentabelle
    - --> andere Prozesse können nicht zugreifen
    - --> Implementierung gemeinsamer Speicherbereiche aufwendiger
- **Schutz vor (z.B.) unberechtigtem Schreiben:**
  - Einträge der Seitentabellen enthalten u.a. Zugriffsrechte (vgl. S.38)
  - Festlegung des Schutzcode durch Compiler/Linker:
    - Einteilung des Programms in Abschnitte, Größe ist Vielfaches der Seitenrahmengröße
    - pro Abschnitt ein Schutzcode in Programmdatei vermerkt
    - Loader setzt Schutzcode in Seitentabelleneinträgen

Seiten-Sharing

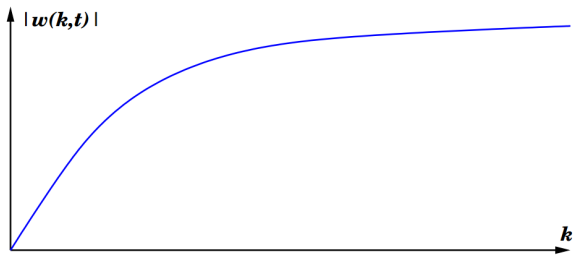
- **Theoretisch:**
  - Einträge verschiedener Seitentabellen zeigen auf gleichen Seitenrahmen
  - Problem:
    - wie stellt man fest, ob Seite bereits von anderen Prozessen verwendet wird ?
    - bei Änderungen (z.B. des verwendeten Seitenrahmens) wären viele Seitentabellen anzupassen
- **Praktisch:**
  - gemeinsamer Teil des Adreßraumes
    - entweder als gemeinsames Segment mit eigener Seitentabelle (Kombination von Segmentierung und Paging z.B. bei Unix) oder
    - als eine Art Pseudo-Prozess-Adressbereich mit eigener globaler Seitentabelle (z.B. Windows)

Demand-Paging (1)

- **Virtueller Speicher:**
  - Teile von Programmen werden nicht ständig benötigt
  - Lokalitätsprinzip: Zugriffe auf Daten und Programmcode häufig lokal gruppiert
- **Idee:** nur Teilmenge an Seiten eines Prozesses werden im Hauptspeicher gehalten: **Resident Set**
- Dadurch
  - --> bessere Systemauslastung: mehr Prozesse gleichzeitig im System
  - ein Prozess kann viel mehr Speicher anfordern als physikalisch verfügbar
- **Demand Paging:**
  - eine Seite
    - wird in den Speicher geladen, wenn auf sie zugegriffen wird
    - kann auch wieder aus dem Speicher entfernt (verdrängt) werden

Demand-Paging (2)

- **Working Set** eines Prozesses P: ***w(k,t)***
  - zur Zeit t: Menge an Seiten, die P bei den letzten k Speicherzugriffen verwendet hat
  - Abhängigkeit von k:

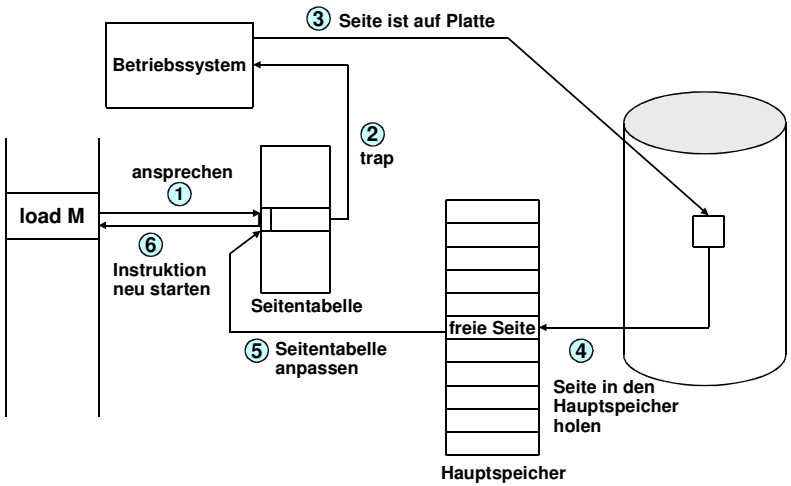


  - für genügend großes k:
    - |w(k,t)| nahezu konstant
    - meist deutlich kleiner als Prozessadressraum

Voraussetzungen

- **Present / Valid-Bit:**
  - in Einträgen der Seitentabelle, gibt an, ob Seite im Speicher
- **page-fault Exeption:**
  - Ausnahmebehandlung, wenn Prozess ausgelagerte Seite anspricht
- **page-fault Handler:**
  - BS-Routine, die benötigte Seite in den Speicher lädt
- **Seitenersetzungsstrategie:**
  - wenn kein freier Seitenrahmen für Einlagerung verfügbar
  - Strategie für Auswahl einer zu ersetzenden Seite

Page-Fault Behandlung



Seitenersetzung

Ablauf eines Seitenwechsels:

- MMU löst page-fault-Exception (Seitenfehler-Ausnahme) aus
- BS ermittelt virtuelle Adresse bzw. die benötigte Seite ***S***
- Falls kein freier Seitenrahmen verfügbar:
  - bestimme zu verdrängende Seite ***S'***
  - falls ***S'*** modifiziert (Modified-Bit = 1): Inhalt von ***S'*** sichern
    - ***S'*** auf Platte schreiben (im Page- oder Swap-Bereich)
    - Seitentabelleneintrag für ***S'*** aktualisieren (u.a. Present-Bit = 0)
- Seite ***S*** von Platte in freien Seitenrahmen (ggf. den von S') laden
- Seitentabelleneintrag für ***S*** aktualisieren (u.a. Present-Bit = 1)
- abgebrochenen Befehl weiterführen oder wiederholen

Kosten von Seitenwechseln

- **Kosten von Seitenwechseln:**
  - Mittlere Speicherzugriffszeit bei Wahrscheinlichkeit ***p*** für Seitenfehler:

$$t_Z = t_{HS} + p \cdot t_{PF}$$

  - t_{HS}: Zugriffszeit auf Hauptspeicher (ca. 10-100 ns)
  - t_{PF}: Zeit für Behandlung eines Seitenfehlers (domoniert durch Plattenzugriff (ca. 10 ms)
  - --> ***p*** muss klein sein
  - --> max. ein Seitenfehler bei mehreren Millionen Zugriffen
- <-> *„thrashing“ (Seitenflattern)* (--> später)

Strategien zur Seitenersetzung

- **Ziel:** möglichst wenige Page-Faults
- **Abrufstrategien** (Wann? --> später)
- **Austauschstrategien**: Wo wird nach Verdängungskandidaten für Seiten gesucht ?
  - lokale Strategie:
    - verdränge nur Seiten des Prozesses, der neue Seite anfordert
      - Adressraumgröße fest oder variabel
      - Größe sollte *Working-Set* entsprechen
      - Einstellung z.B. aufgrund der Seitenfehlerfrequenz
    - Prozess mit vielen Page-Faults beeinträchtigt nicht das Gesamtsystem
  - globale Strategie:
    - suche Verdrängungskandidaten unter den Seiten aller Prozesse
      - Prozesse nehmen sich gegenseitig Seiten weg
      - Prozess mit vielen Page-Faults erhält mehr Seiten (kann vor- und nachteilig sein)

Optimale Strategie und FIFO

Optimale Strategie

- Verdränge die Seite, die künftig am längsten nicht mehr benötigt wird
- verursacht kleinste Zahl an Page-Faults
- ist nicht implementierbar
- kann modellhaft zur Bewertung anderer Strategien dienen

First In First Out (FIFO)

- Verdränge die Seite, die am längsten im Hauptspeicher ist
- einfach umzusetzen, aber schlechte Strategie
  - Zugriffsverhalten (Working-Set) wird ignoriert
  - bspw. kann ersetzte Seite dauernd in Verwendung sein -> gleich wieder einzulagern

Nutzung eines Referenz-Bits

- Rerefenz-Bit (R-Bit) kann in Seitentabelleneinträgen stehen
  - wird von MMU (Hardware) bei Zugriff auf Seite gesetzt
  - wird nach bestimmten Kriterien vom BS (Software), z.B. regelmäßig, wieder gelöscht
- liefert Information, ob auf Seite nach letztem Löschen zugegriffen wurde
- sagt nichts
  - über Zeitpunkt des Zugriffs,
  - über Reihenfolge der Zugriffe auf mehrere Seiten
- R-Bit und weitere Bits in Einträgen der Seitentabelle sind Basis für weitere Strategien:
  - NRU (Not Recently Used)
  - Second-Chance
  - LRU (Least Recently Used)

Not Recently Used (NRU)

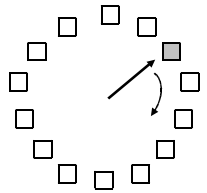
Not Recently Used (NRU)

- Basis: von MMU gesetzte Statusbits in Seitentabelle:
  - **R**-Bit: Seite wurde referenziert
  - **M**-Bit: Seite wurde modifiziert
- R-Bit wird vom BS in regelmäßigen Abständen gelöscht
- Bei Verdrängung: vier Prioritätsklassen
  - 0: nicht referenziert, nicht modifiziert
  - 1: nicht referenziert, modifiziert
  - 2: referenziert, nicht modifiziert
  - 3: referenziert, modifiziert
- Auswahl innerhalb einer Klasse zufällig
- nicht besonders gut, aber einfach

Second-Chance Algorithmus

Second Chance

- Erweiterung von FIFO
- Idee: verdränge älteste Seite, auf die seit dem letzten Seitenwechsel nicht zugegriffen wurde
- Implementierung:
  - Seiten im Hauptspeicher werden nach Alter sortiert in Ringliste angeordnet
  - Zeiger zeigt auf älteste Seite



- **bei Seitenfehler:**

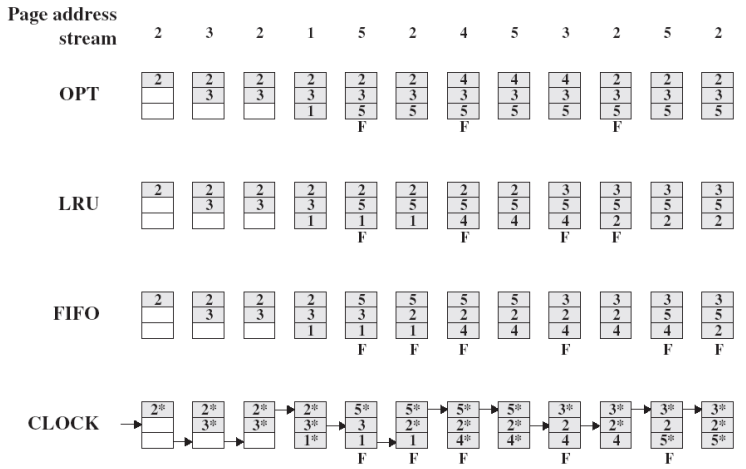
  betrachte Seite, auf die der Zeiger zeigt
  - R = 0: verdränge Seite, fertig
  - R = 1: lösche R-Bit,
    setze Zeiger eins weiter
    wiederhole von vorne
  - --> kürzlich referenzierte Seite erhält „zweite Chance“

Least Recently Used (LRU)

Least Recently Used (LRU)

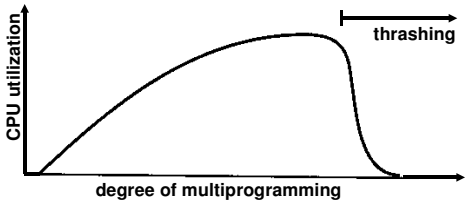
- Verdränge die Seite, die am längsten nicht benutzt wurde
  - Vermutung: wird auch in Zukunft am längsten nicht mehr benutzt
- nahezu optimal, wenn Lokalität gegeben
- Problem: (Software-) Implementierung
  - bei jedem Zugriff muss ein Zeitstempel aktualisiert werden
  - --> Näherungen, z.B. Aging über einen Zähler

Beispiele



Thrashing

- **Thrashing:**
  - Prozess erzeugt exzessiv viele Page-Faults (alle paar tausend Instruktionen)
    - --> Prozess verbringt meiste Zeit mit Ein- und Auslagern von Speicherseiten
    - --> Systemleistung sinkt dramatisch
  - entsteht, wenn Prozess mehr Seiten aktiv verwendet als Seitenrahmen für ihn verfügbar



- **Lösungen:**
  - falls freier Speicher vorhanden:
    Zuteilung weiterer Seitenrahmen:
    - Working-Set anpassen
    - globale Ersetzungsstrategie
  - falls kein freier Speicher:
    - Swapping ganzer Prozesse

Weitere Design-Möglichkeiten (1)

- Austauschstrategien: --> siehe zuvor
- **Abrufstrategien:** Wann werden Seiten ein-/ausgelagert ?
  - erst bei Bedarf, also bei Seitenfehlern: **Demand Paging**
  - im Voraus: **Prepaging**, z.B,
    - bei Programmstart: gewisse Anzahl Seiten laden, noch bevor sie angesprochen werden
    - z.B. lade Folgeseiten auf Platte gleich mit
  - **asynchron:**
    - modifizierte Seiten vorab auslagern, wenn
      - I/O-Last gering oder
      - Vorrat freier Seiten zu klein

Weitere Design-Möglichkeiten (2)

- **Abrufstrategien:** ...
  - **Clustering:**
    - bei Page-Fault nicht einzelne Seiten, sondern Einlagern ganzer Cluster
    - modifizierte Seiten als Cluster auf Platte schreiben
  - **Locking von Seiten** im Speicher:
    - einzelne Seiten eines Prozesses werden vom Paging ausgenommen (implementiert über privilegierten System-Call)

Auch das Design der Programme hat Auswirkungen auf die Zahl der Page Faults und somit auf die Laufzeit des Programms selbst wie auch auf die Leistung des gesamten Systems!

Praxisbeispiele

Praxis: Übersicht

- Shared-Memory --> Kapitel IPC
- mmap

Praxis: File Mapping (1)

- Grundidee eines file mapping ist es, den Inhalt einer Datei in den virtuellen Adreßraum abzubilden und dann ohne den Overhead des Dateisystems zuzugreifen.
- Ein file mapping kann prozeßprivat oder gemeinsam genutzt (shared) sein.
- Um eine Datei in den Adreßraum abzubilden verwendet man den System Call

  `paddr = mmap( addr_hint, len, prot, flags, fd, offset )`

  wodurch der Byte-Bereich `[offset,offset+len)` in der durch den file descriptor `fd` angegebenen Datei auf den Adreßbereich `[paddr,paddr+len)` abgebildet wird. Für das Argument `flag` kann entweder MAP_SHARED oder MAP_PRIVATE angegeben werden.
- File Mappings können von Applikationen und von der Speicherverwaltung selbst genutzt werden.

Praxis: File Mapping (2)

```
#include <fcntl.h>                    //O_RDWR
#include <sys/mman.h>                 //mmap, munmap
#include <unistd.h>                   //open, close
#include <new>                        //new() []
...
    fildes = open( "test.data", O_CREAT | O_RDWR | O_TRUNC );

    void * address = mmap( (caddr_t) 0, len, (PROT_READ | PROT_WRITE),
                           /*MAP_PRIVATE*/ MAP_SHARED, fildes, offset );

    Object * ptr = new (address) Objekt[len] ();
    ptr[...] = ...;

    msync( address, len, MS_SYNC );      //flush

    munmap( address, len );

    close( fildes );
...
```


Zusammenfassung

Zusammenfassung (1)

- Ziele der Speicherverwaltung:
  - effiziente Speicherzuteilung, Speicherschutz
- Wichtig: Unterscheidung logischer / physischer Adressraum
- Swapping: Ein-/Auslagern kompletter Adressräume
  - Suche nach freiem Speicher: *First Fit, Quick Fit*
- Virtuelle Speicherverwaltung: *Paging*
  - Einteilung
    - des logischen Adressraums in Seiten (*Pages*)
    - des physischen Adressraums in Seitenrahmen (*Page Frames*)
  - jede Seite kann
    - auf beliebige Seitenrahmen im Speicher abgebildet werden
    - auf Festplatte ausgelagert werden

Zusammenfassung (2)

- Hardware (MMU) bildet bei jedem Speicherzugriff logische auf physische Adresse ab
- Beschreibung der Abbildung in Seitentabelle pro Prozess
  - pro Seite ein Eintrag , enthält u.a.
    - Nummer des Seitenrahmens
    - Present-Bit: ist der Seite ein Seitenrahmen zugewiesen ?
- mehrstufige Seitentabellen
  - Tabellen tieferer Stufen nur vorhanden, falls nötig
- TLB: Cache in der MMU
  - speichert zuletzt verwendete Tabelleneinträge

Zusammenfassung (3)

- Seitenersetzungsstrategien
  - bestimmen, welche Seite wann verdrängt wird
  - Optimale Strategie
  - NRU: vier Klassen gemäß R und M-Bit
  - FIFO: die Seite, die am längsten im Speicher ist
  - *Second Chance*:
    - die älteste Seite, die seit letztem Seitenwechsel nicht benutzt wurde
    - Clock-Algorithmus: effiziente Implementierung
  - LRU: die Seite, die am längsten nicht benutzt wurde

Zusammenfassung (3)

- Dynamische Seitenersetzung (*Demand-Paging*)
  - nur aktuell benötigte Seiten (*Working Set*) werden im Hauptspeicher gehalten
  - Rest auf Plattenspeicher verdrängt
  - bei Zugriff auf ausgelagerte Seiten: Seitenfehler (*Page Fault*)
    - BS lädt Seite in Hauptspeicher,
    - BS muss ggf. andere Seite verdrängen (Seitenersetzung)
-