

I. VIRTUELLE SPEICHERVERWALTUNG

Notwendigkeit

Immer größere Programme

Immer mehr Programme „gleichzeitig“

↪ verfügbarer Arbeitsspeicher schnell zu klein

Lösung: Nur gerade benötigte Teile der aktiven Programme im Arbeitsspeicher, Rest bei Bedarf aus Hauptspeicher nachladen (*swapping, paging*)

Umsetzung: MMU (*memory management unit*) setzt virtuelle Adressen in physikalische um

Virtueller Speicher

Speicherkapazität größer als effektive Hauptspeicherkapazität

Betriebssystem lagert nach Bedarf Speicherbereiche ein/aus

MMU-Adressberechnung hardwaremäßig eindeutig

Abbildungsinformation in Übersetzungstabellen gespeichert

↪ Abbildungsinformation für zusammenhängende Adressbereiche, um Übersetzungstabellen klein zu halten

Virtueller Speicher – Verwaltung (Segmentierung)

Virtueller Adressraum wird in Segmente verschiedener Länge zerteilt

Mehrere Segmente pro Programm (zB für Programmcode, Daten)

Segmente enthalten logisch zusammenhängende Informationen, relativ groß

Vorteile:

- spiegelt logische Programmstruktur wieder
- große Segmente ↪ relativ seltener Datentransfer

Nachteile:

- Datentransfer umfangreich falls notwendig
- Programm aus nur einem Code- und Datensegment
- ↪ muss vollständig eingelagert werden

Virtueller Speicher – Verwaltung (Seiten)

logischer und physikalischer Adressraum in Teile fester Länge (Pages) zerteilt

Pages relativ klein (256-4k Byte)

Viele Seiten pro Prozess, keine logischen Zusammenhänge

Vorteile:

- kleine Seiten ↪ nur wirklich benötigter Programmteil wird eingelagert
- geringerer Verwaltungsaufwand als Segmentierung

Nachteile:

- häufigerer Datentransfer als bei Segmentierung

Segmentbasierte Speicherverwaltung – Implementierung

virtuelle Adresse wird in **Segmentnummer** (n höherwertigste Bits der virtuellen Adresse) als Segmentkennung und in **Byte-nummer** (verbleibende m Bits der virtuellen Adresse) als Abstand zum Segmentanfang unterteilt

max. virtuelle Segmentanzahl 2^n , max. Segmentgröße 2^m

Adressabbildung über Segmenttabelle (im MMU-Registerspeicher)

reale Adresse = Segmentbasisadresse + virtuelle Byte-Nummer

Segmentlängenangaben um segmentüberschreitende Zugriffe feststellen/verhindern zu können

Verschnitt: ungenutzter Raum bei Segmenten kleiner als 2^m

↪ gute Hauptspeicherausnutzung, wenn Segmentgrenzen an jeder Byteadresse zulässig sind

Realität: Segmentgrenzen an Vielfachen von Blöcken (hier 256 Bytes)

Segmente im virtuellen physikalischen Adressraum in 256-Byte-Blöcke unterteilt

↪ m -Bit-Bytenummer wird aufgeteilt in kürzere Bytenummer für Byteadressierung in Block und Blocknummer

Adressumsetzung: virtuelle Segmentnummer wird auf reale 24-Bit-Blocknummer als Segmentbasis abgebildet

Virtuelle Bytenummer für Adressierung innerhalb des Blocks wird unverändert übernommen

Virtuelle Speicherverwaltung – Probleme

Einlagerungszeitpunkt: Wann werden Segmente/Seiten in Hauptspeicher eingelagert?

Zuweisungsproblem: Wo in Hauptspeicher werden Seiten/Segmente eingelagert?

Ersetzungsproblem: Welche Segmente/Seiten auslagern um Platz für neu benötigte Daten zu schaffen?

Probleme – Einlagerungszeitpunkt

Gängiges Verfahren: Einlagerung auf Anforderung (*demand paging* bei Seitenverfahren)

Daten werden eingelagert, sobald auf sie zugegriffen wird, aber nicht in Hauptspeicher liegen

Segment-/Seitenfehler (*segment/page fault*): Zugriff auf nicht in Hauptspeicher vorhandene(s) Segment/Seite

Probleme – Zuweisungsproblem (Segmentierung)

ausreichend große Lücke in Hauptspeicher muss gefunden werden
Strategien:

1. **first-fit:** erste passende Lücke wird genommen
2. **best-fit:** kleinste passende Lücke wird genommen
3. **worst-fit:** größte passende Lücke wird genommen

Problem: Speicher zerfällt in belegte und unbelegte Speicherbereiche

↪ **externe Fragmentierung**

unbelegte Speicherbereiche oft zu klein um weitere Segmente aufnehmen zu können

Probleme – Zuweisungsproblem (Seiten)

Problem taucht nicht auf, da alle Seiten gleich groß

↪ es entstehen immer passende Lücken

↪ **keine externe Fragmentierung**

Interne Fragmentierung: Einheitliche Seitengröße ↪ auf letzter Seite des Programm-Moduls wahrscheinlich ungenutzter Leer-raum

Probleme – Ersetzungsproblem (Segmentierung)

Limitierung der Anzahl gleichzeitig von einem Prozess benutzten Segmente

↪ bei Einlagerung eines neuen Segments wird anderes Segment des Prozesses ausgelagert

Probleme – Ersetzungsproblem (Seiten)

Seitenersetzungsstrategien:

1. **FIFO** (*first in first out*): älteste Seite wird ersetzt
2. **LIFO** (*last in first out*): jüngste Seite wird ersetzt
3. **LRU** (*last recently used*): am längsten unbenutzte Seite wird ersetzt
4. **LFU** (*least frequently used*): am seltensten benutzte Seite wird ersetzt
5. **LRD** (*least reference density*): Seite mit geringster Zugriffsichte wird ersetzt

Virtueller vs. physikalischer Cache

virtueller Cache: zwischen CPU und MMU, Tags = höherwertigste Bits der logischen Adressen

Vorteile:

- bei Treffer keine MMU nötig

physikalischer Cache: zwischen MMU und Speicher, Tags = höherwertigste Bits der physikalischen Adressen

Vorteile:

- kleinerer Tag, da physikalische Adresse idR viel kleiner als logische Adresse ist

Segmentorientierte Speicherverwaltung – Beispiel x86

Segment-Deskriptor: beschreibt sein Segment durch Attribute:

- **Segment-Basisadresse** (*base address*)
- **Segment-Größe** (*limit*) in Bytes
- **Zugriffsrechte** (*access rights*) auf das Segment

Speicherwort wird durch **Segment-Selektor** (kennzeichnet Segmentanfang) und **Offset** innerhalb des Segments adressiert

Segment-Selektoren wechseln selten, da durch Lokalitätseigenschaften von Programmen nicht bei jedem Hauptspeicherzugriff neues Segment benutzt wird

Segment-Selektoren in **Segment-Registern** gespeichert (wegen Geschwindigkeit)

Segment-Register:

- **Code-Segment** (CS)
- **Stack-Segment** (SS)
- **Daten-Segment** (DS)
- **Extra-Segment** (ES, FS, GS,...)

↪ Speicherwortadressierung durch Segment-Register + Offset

Segment-Deskriptor-Cache: Speichert wichtige Informationen über ausgewähltes Segment

↪ keine Speicherzugriffe beim Lesen von Segmenteigenschaften

Jedes Segment mit Anfangsadresse im Segmentregister ist physikalisch im Hauptspeicher

Arbeitsmenge (*working set*): durch Segmentregister spezifizierte Segmente eines Prozesses

Adressierungsmodi – Beispiel x86

Verschiedene Adressierungsmodi möglich (für Kompatibilität)

Real (Address) Mode (Kompatibilitätsmode zu 8086):

- physikalische Adressen: 20 Bit
- max. adressierbare Hauptspeicherkapazität: 1 MB
- max. Segmentlänge: 64 KB
- Abbildung virtueller Adressraum (mehrere TB) auf physikalischen Adressraum (4 GB)
- Segment-Selektor spezifiziert nicht Basisadresse des Segments, sondern verweist auf Segment-Deskriptor in Segment-Deskriptor-Tabelle

Protected (Virtual Address) Mode:

- vergrößerter Adressraum mit erweiterten Speicherschutzfunktionen, Multitasking

Seitenorientierte Speicherverwaltung:

- existiert zur Kompatibilität mit 80386

Fehlererkennung

Segmentfehler: Bit in Segment-Deskriptor zeigt, ob Segment in Hauptspeicher ist oder nicht

Seitenfehler: Seite/Seitentabelle befindet sich nicht in Hauptspeicher ↪ spezielles Kennungsbit in Seitentabellen-Verzeichnis

Beschleunigung der Adressberechnung

schneller, vlassoziativer Cache (*translation lookaside buffer*, TBL) speichert automatisch zuletzt benutzte Einträge aus Seitentabellenverzeichnis und Seitentabelle

↪ im Trefferfall (ca. 90%) muss **nicht** auf im Hauptspeicher liegende Seitentabelle zugegriffen werden

Schutzmechanismen

Schutz vor unerlaubten Speicherzugriffen:

- Trennung von Systemsoftware von Anwendungsprozessen
- Trennung der Anwendungsprozesse voneinander

Schutzebenen

= *privilege level* (PL)

wichtiges Mittel zur Realisierung von Schutzmechanismen

Varianten:

1. zwei Schutzebenen (bei x86-Seitenverwaltung):
 - **OS-Modus** (*supervisor mode*)
 - **Benutzermodus** (*user mode*)
 - Auftrag in Benutzermodus darf keine Daten des OS-Modus benutzen
2. vier Schutzebenen (Segmentverwaltung):
 - PL0 (am vertrauenswürdigsten) bis PL3

Zugriffsschutz – Regeln

Prozess darf nur auf Daten zugreifen, die höchstens so vertrauenswürdig sind wie er selbst

Prozess darf nur Code benutzen, der mindestens so vertrauenswürdig ist wie er selbst

Zugriffsrechte ermöglicht, dass nur unter bestimmten Voraussetzungen auf Informationen zugegriffen werden darf