**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

# 7 Zentralspeicherverwaltung

## *7.1 Grundlagen*

Erinnerung: jedes Programm verwendet den Hauptspeicher. Die CPU lädt Programmcode und Daten aus dem Hauptspeicher, führt diese aus bzw. verarbeitet die Daten und schreibt sie in den Hauptspeicher zurück. Eine CPU-Anweisung kann mehrere Speicherzugriffe enthalten. Beispiele:

```
MOV R1, 4711      // lade den Inhalt von Speicherzelle 4711 nach R1
TSL 0815          // teste 0815 und setzte auf eins
```

### 7.1.1 Bindung der Speicheradressen

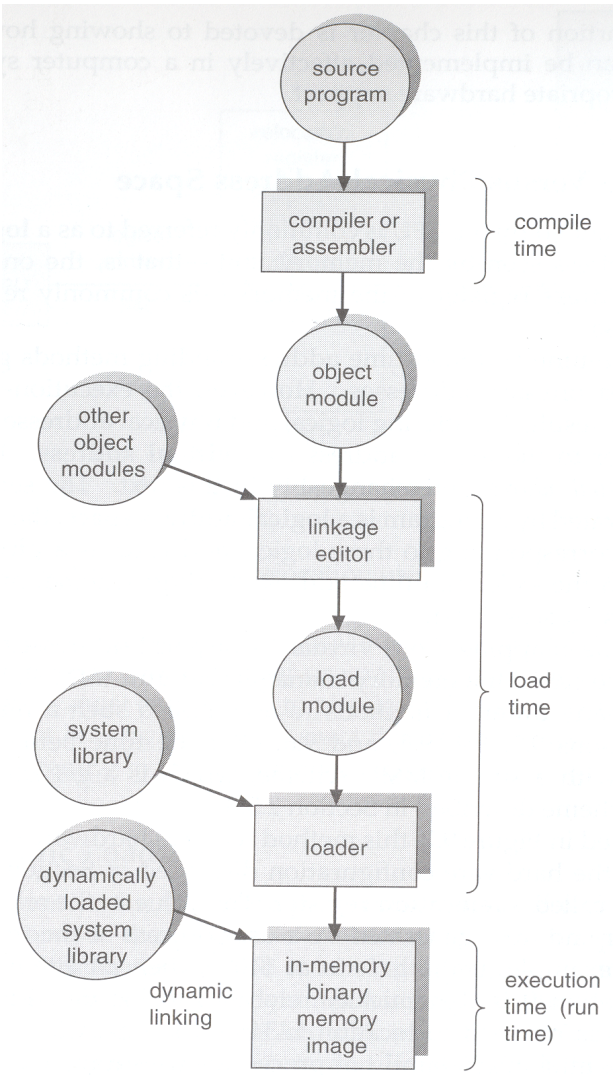



**Abbildung 62: Übersicht Adressenbindung (nach [2])**

- **Compile-Time:** Beim Compilieren werden die physikalischer Adressen im Programm festgelegt. Wurde bei MS-DOS im COM-Format verwendet.

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Nachteil: Es kann nur ein Programm gleichzeitig im Speicher sein! Würde z.B. ein Programm zweimal gestartet, so würden beide Instanzen dieselben Adressen verwenden!

- **Load-Time:** Beim Laden des Programmes werden die physikalischen Adressen ins Programm geschrieben. Der Linker erzeugt *relative* Adressangaben, z.B. „Variable X belegt 4 Byte an Stelle 14 gezählt vom Modul-Start", oder „Funktion Y() befindet sich an Position 4711 im Code-Modul". Der Program Loader ersetzt vor Programmstart alle relativen Adressbezüge durch die physikalischen Adressen; dasselbe Programm kann zweimal gestartet werden, da beide Instanzen unterschiedliche Adressen bekommen.
- **Execution-Time:** Beim Adresszugriff wird die relative Adresse in eine physikalische umgesetzt->nur mit Spezialhardware möglich

### 7.1.2 Logischer und physikalischer Speicher

**Logische (virtuelle) Adresse:**
bezeichnet die Adresse, wie sie von der CPU erzeugt und gesehen wird.

**Physikalische Adresse:**
Die Adresse, mit der die Speicherstelle im Hauptspeicher angesprochen wird.

Bei Compile-Time und Load-Time Adressbindung sind logische und physikalische Adresse identisch.
Umsetzung von logischer auf physikalische Adresse durch Spezial-Hardware Memory Management Unit (MMU):

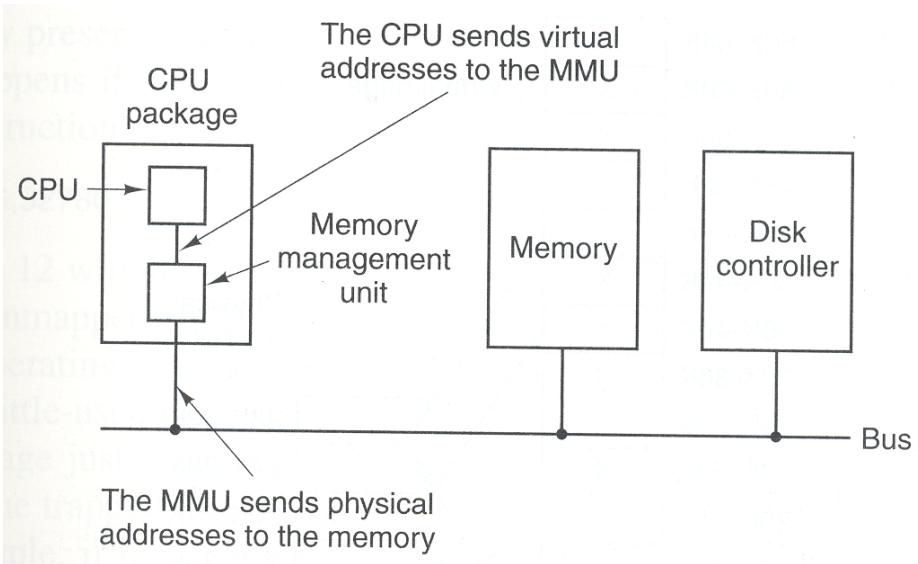
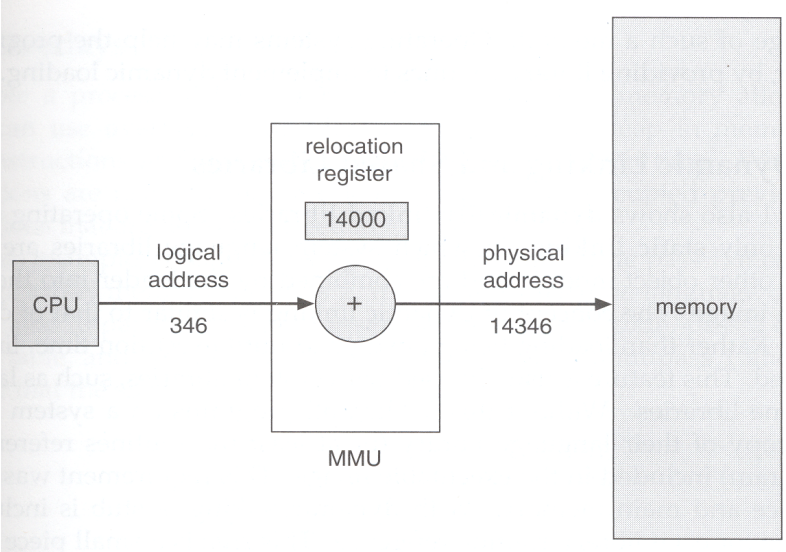



**Abbildung 63: MMU Architektur**

Einfache MMU zur Adress-Verschiebung mittels Relocation Register:

- Die CPU adressiert die Adresse 346
- Das Relocation Register enthält 14000; der Adresszugriff wird umgesetzt auf 14346 und die entsprechenden Daten aus dem Speicher geholt.
- Logische Adressen: 0 bis zu einem Maximum *max*
- Physikalische Adressen: 14000 bis 14000+*max*

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de




Das Programm sieht niemals die physikalische Adresse, sondern arbeitet ausschliesslich mit den logischen Adressen!

### 7.1.3 Dynamisches Laden / Overlay Technik

Nicht der gesamte Programmcode befindet sich im Hauptspeicher, sondern nur Teile davon. Wird ein Teil benötigt, der nicht im Hauptspeicher ist, so wird dieser dynamisch nachgeladen.
->Overlay-Technik

### 7.1.4 Dynamisches Linken und gemeinsame Bibiliotheken (Shared Libraries)

**Statisches Linken:**
Alle Programmbibliotheken werden vom Linker zum Programm hinzugefügt.

**Dynamisches Linken:**
Der Linker fügt anstelle der Bibliothek einen *Stub* hinzu; wird die erste Funktion aus der Bibliothek aufgerufen, so wird diese *dynamisch* hinzulinkt.

- Befindet sich die Bibliothek noch nicht im Speicher, so wird sie geladen
- Alle Referenzen im Programm werden an die geladene Bibiliothek angepasst.
- Befindet sich die Bibliothek bereits im Speicher, da sie z.B. von einem anderen Programm genutzt wird, so muss sie nicht mehr geladen werden; ggfs. sind Initialisierungen durchzuführen
- Wird hauptsächlich für Systembibliotheken verwendet.

**Vorteil:**

- DLLs können mehrfach genutzt werden (geringerer Speicherverbrauch, kürzere Ladezeit).
- Bugfixes: wird eine DLL verbessert, so profitieren automatisch alle Programme davon!

**ABER:**

- "DLL-Hell" (DLL-Hölle) bei Windows:

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

    - viele Programme verwenden z.B. comctl32.dll (enthält graphische Elemente von Windows, z.B. Buttons, Drop-Down-Lists, etc).
    - Irgendwann hat Microsoft die DLL erweitert, um neue Buttons etc. zu implementieren. Kein Problem, die alten Programme funktionieren auch mit den neuen DLL.
    - Jahre gehen ins Land und jedes Programm nutzt das neue Feature.
    - Jetzt installiert der Benutzer ein Uralt-Programm "grauer-opa.exe" von 1993. Dieses benötigt die comctl32.dll, und um sicher zu gehen, dass sie auch da ist, kopiert es diese beim installieren ins Windows-System32 Verzeichnis. "grauer-opa.exe" funktioniert dann auch einwandfrei, leider stürzen alle anderen Programme ab! Sie nutzen die neuen Features in der comctl32.dll, die in der alten DLL nicht drin sind!
- Dies führte zu bizarren Fehlern, die kaum beseitigt werden konnten. Nach der Installation eines neuen Programmes **X** funktierten Teile von **Y** auf einmal nicht mehr!
- Es gab sogar Situationen, wo *entweder* **X** *oder* **Y** auf einem System installiert werden konnten, jedoch nicht beide gemeinsam!
- Das Problem liegt in der mangelhaften Versionierung von DLLs. .NET enthält Mechanismen, um alle DLLs für jedes Programm getrennt zu verwalten. Jedes Programm erhält dann "seine" DLLs in der gewünschten / benötigten Version (was natürlich den Gedanken der "gemeinsamen Nutzung" etwas untergräbt)

### 7.1.5 Echtzeitsysteme

Viele einfache Echtzeitsysteme haben keine Festplatte. Das Programm ist unveränderlich im ROM (Read-Only Memory) gespeichert, während die Daten im RAM gespeichert sind. ROM kann z.B. sein:
- EPROM (erasable programmable read-only memory): durch UV-Licht löschbarer Speicher
- Flash: elektrisch löschbarer Speicher
- Maske: es wird eine eigene Prozessormaske entwickelt, die das Programm fest enthält.

Für ROM und RAM werden jeweils eigene Adressbereiche festgelegt. Nach dem Compilieren und dem Linken wird ein weiterer Schritt durchgeführt, in dem die absoluten Adressen für RAM und Programmcode eingefügt werden. Anschließend kann das Progamm auf die Target-Hardware ausgeführt werden.

**Beispiel: Aufzugsteuerung, 16-Bit Prozessor (NEC V25) mit 20 Adressleitungen**
20 Adressleitungen, möglicher Adressraum $2^{20}$ = 0x100000, 128 K Programm, 6 K Speicher:
- Programmcode (ROM) 0xE0000 – 0xFFF00
- Hauptspeicher (RAM) 0x01000 – 0x02800
- 256 Spezialregister des Prozessors (z.B. serielle Schnittstelle, Timer, etc.) 0xFFF00 – 0xFFFFD
- Reset-Vektor (16 Bit) 0xFFFFE – 0xFFFFF
- Interrupt-Vektortabelle 256 Bytes 0x00000 – 0x00100

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

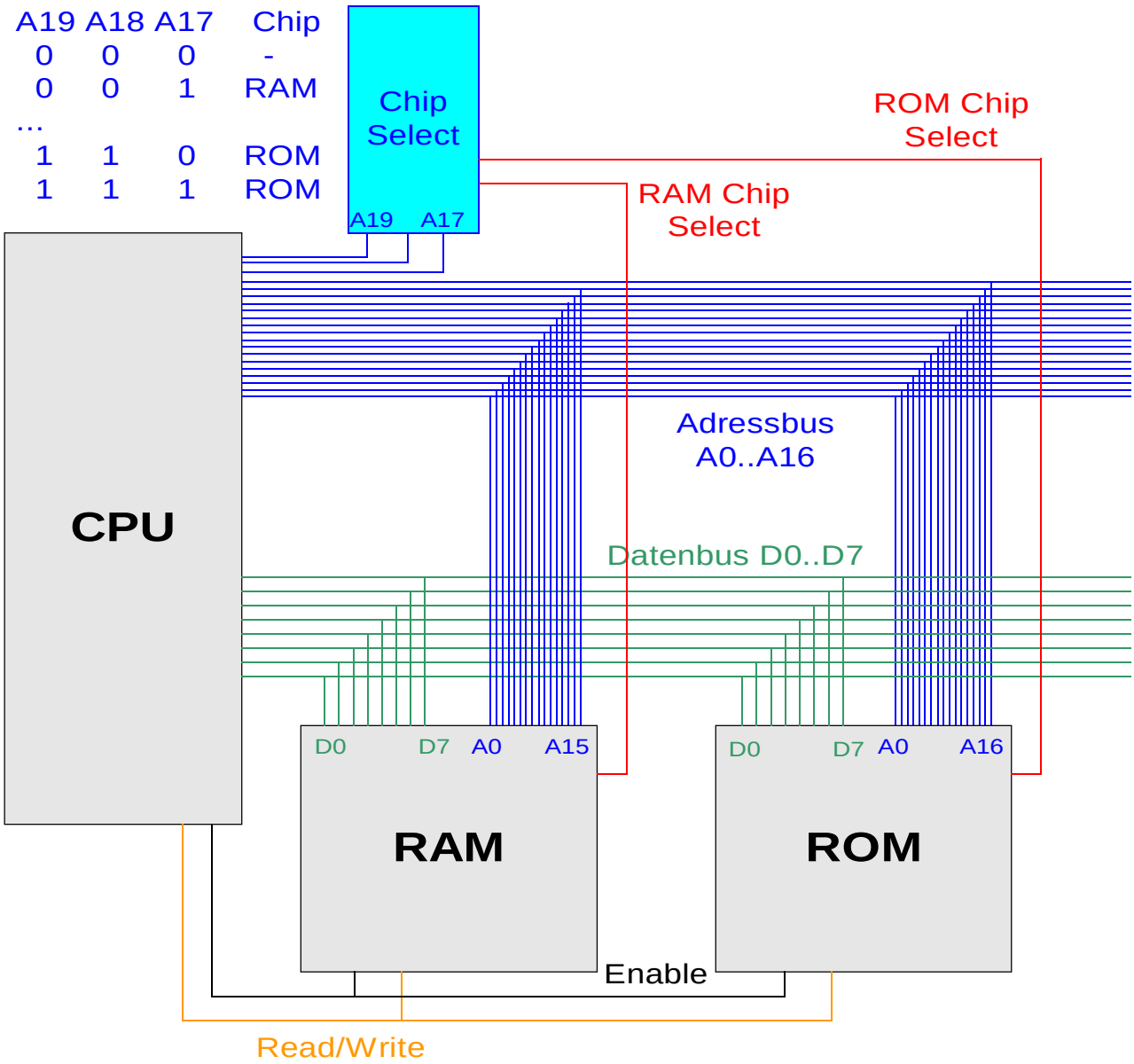



**Abbildung 66: Adress- und Datenbus für ein Echtzeitsystem**

**Hardware-Aufbau:**
- Adress-Leitung 20 Bits
- RAM 64kB (16 Adressleitungen)
- ROM 128 kB (17 Adressleitungen)
- Adressleitungen A17-A19 werden zur Auswahl des Chips herangezogen

**Daten lesen:**
- CPU legt Adresse an den Adressbus an, und gibt Read/Write aus (für Einlesen: Read-aktiv)
- Chip-Select Chip wählt entsprechenden Chip aus (RAM, ROM)
- CPU wartet, bis Adressbus stabilisiert ist

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Dann wird mit **Enable** der Chip aktiviert; die entsprechenden Daten werden an den Datenbus gelegt
- Wiederum wartet die CPU kurz; dann liest sie die Daten von D0..D7 ein.

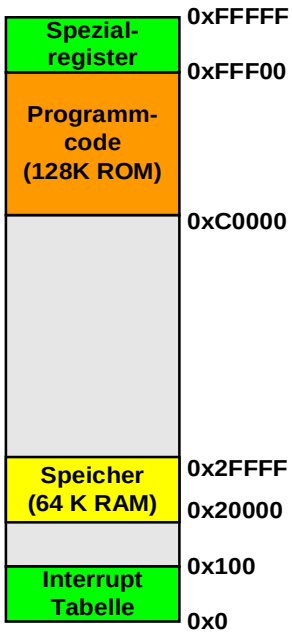



**Abbildung 67: Speicherlayout Echtzeitsystem**

## *7.2 Monoprogrammierung*

Es läuft maximal ein Programm, z.B. MS-DOS.

- Das Betriebssystem belegt ein oder mehrere Bereiche im Speicher, typischerweise am unteren und / oder oberen Rand. Der Rest steht für das Programm zur Verfügung.
- Programm verwendet physikalische Adressen, keine Adressumsetzung nötig

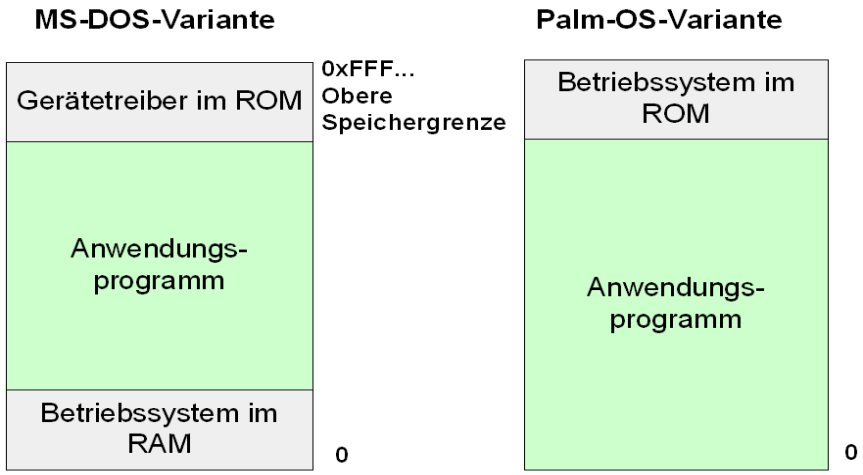



**Abbildung 68: Speicherverwaltung bei Monoprogrammierung (nach [1])**

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

## *7.3 Multiprogrammierung-Betrieb mit festen Partitionen*

Wurde früher für Großrechnersysteme, z.B. IBM OS/360 verwendet. Der verfügbare Speicher wird in Partitionen mit fester Größe aufgeteilt. In jeder Partition wird genau ein Job ausgeführt. Die Jobs warten in einer Warteschlange auf die Zuteilung zu einer Partition. Die Partitionen werden vom Operator fest konfiguriert und können nicht geändert werden.

- Adressumsetzung durch Loader (bei Programmstart) oder durch einfache Verschiebung (z.B. Relocation Register)

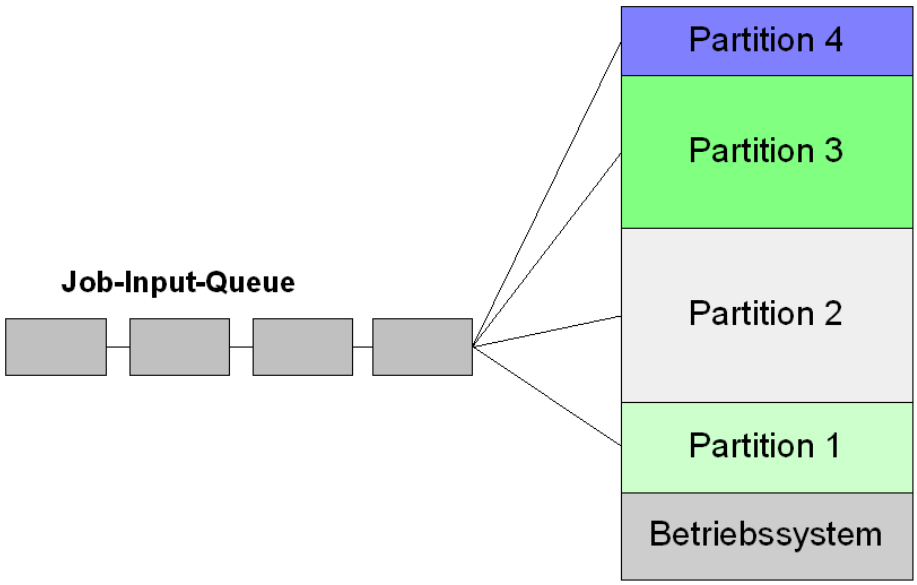



**Abbildung 69: Speicherverwaltung mit festen Partitionen (nach [1])**

## *7.4 Swapping*

- Ein Prozess kann angehalten und aus dem Speicher entfernt werden. Dabei wird sein gesamter Speicher auf Festplatte geschrieben.
- Der Prozess kann später jederzeit wieder fortgeführt werden; der Speicherinhalt wird wieder geladen. Dies muss nicht notwendigerweise an derselben physikalischen Adresse geschehen.
- Erfordert Adressbindung zur Laufzeit (Execution-Time), da der Prozess ja wieder an anderer Stelle geladen werden kann.
- Kontextwechsel dauert sehr lange, da gesamter Speicher auf Festplatte geschrieben werden muss (bzw. wieder geladen werden muss)
- Wenn der Prozess auf I/O wartet, so kann er nicht so einfach geswappt werden
- Kann mit einfachen Hardware-Mitteln (z.B. relocation register, RR) verwirklicht werden: Das Relocation Register wird im PCB gespeichert; bei einem Prozesswechsel wird das entsprechende RR wieder hergestellt.
- Wird nur noch „im Notfall" verwendet, z.B. wenn Linux der physialische Speicher ausgeht

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

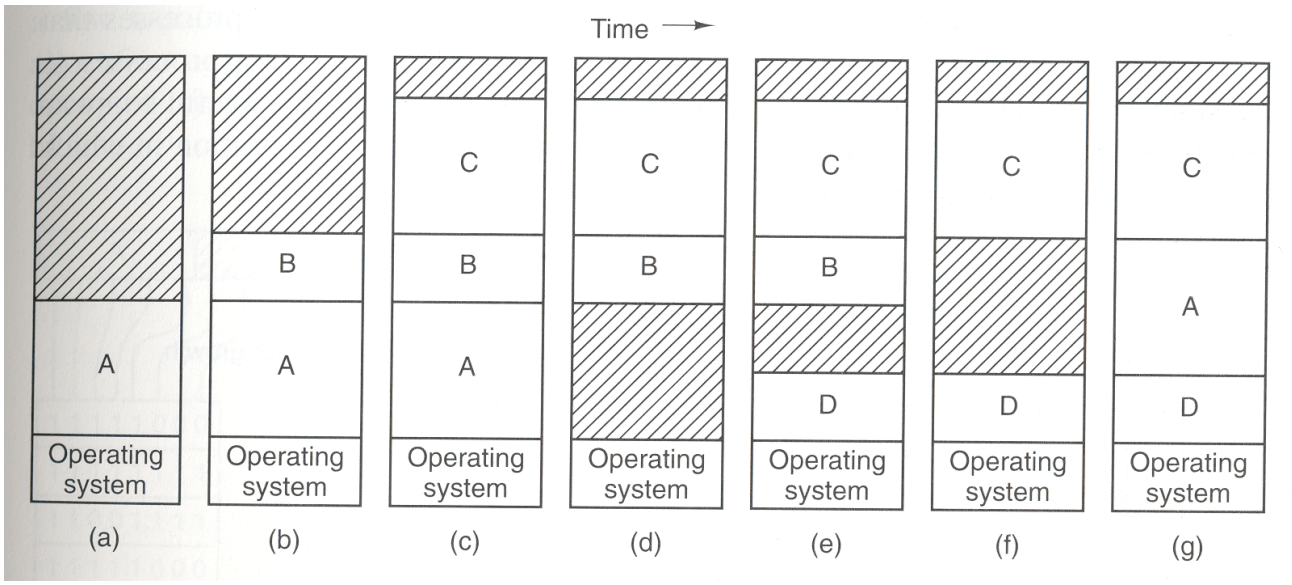



**Abbildung 70: Ablauf beim Swapping**

### 7.4.1 Speicherverteilung

- typischerweise OS unten oder oben (z.B. Windows: obere 2 GB sind Betriebssystem)
- Platz zum Wachsen bereitstellen: typischerweise wird für Stack und Daten (Heap) Platz bereitgestellt (siehe folgende Abbildung (b) )

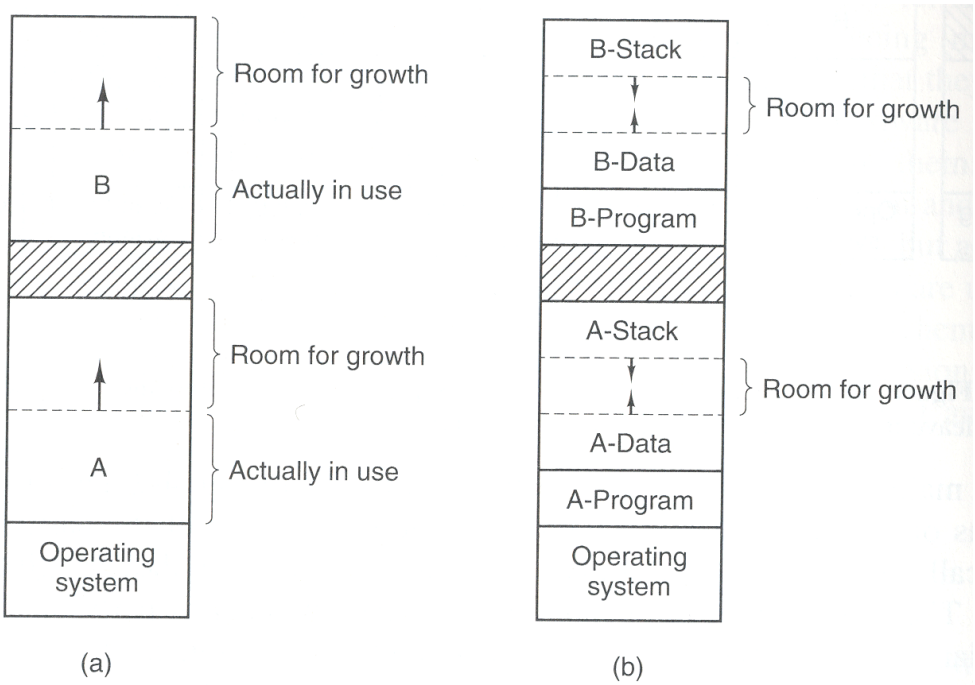



**Abbildung 71: Speicherverteilung im Prozess (nach [1])**

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

### 7.4.2 Speicherschutz

Einfache Möglichkeit: Relocation Register mit Limit Register

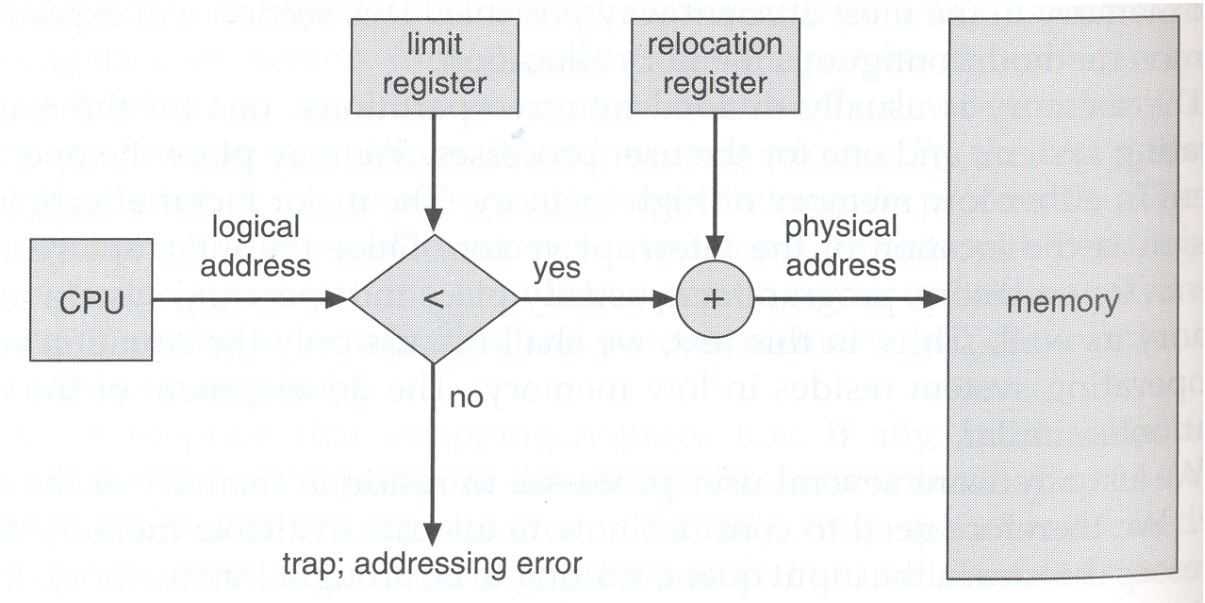



**Abbildung 72: Adressverschiebung mit Begrenzung**

### 7.4.3 Problem: Fragmentierung und Verschnitt

**Interne Verschnitt:**
Das Betriebssystem teilt Speicher in festen Blockgrößen zu, z.B. immer 4 kB Blöcke. Benötigt ein Prozess z.B. 10 kB, so bekommt er 12 kB zugeteilt -> interner Verschnitt von 2 kB.

**Externe Verschnitt:**
Das Betriebssystem teilt Speicher in variabler Größe zu, dadurch wird der verfügbare freie Speicher zerstückelt.
Externer Verschnitt: Verhältnis von größtem freiem Block zu freiem Speicher
Durch Relocation (Verschieben von Prozessen) kann der externe Verschnitt beseitigt werden; wird jedoch nicht gemacht, da es sehr zeitaufwendig ist

### 7.4.4 Verwaltung von freien Speicherbereichen

#### *7.4.4.1 Bitmaps: Bitketten für Speicher*

Der Speicher wird in gleich große Blöcke aufgeteilt. Eine Bitmap enthält für jeden Block den Zustand frei / belegt: ist das Bit 0, so ist der Block frei, ist das Bit 1, so ist der Block belegt.
**Algorithmus:**

- Anfangszustand: alle Bits auf Null (Speicher ist frei)
- Neue Anforderung für *n* Blöcke: suche eine Unterkette der Länge *n* aus Null-Bits, belege diese Unterkette mit Einsen
- Freigabe: setzte alle *n* Bits auf Null

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

**Beispiel:** Blockgröße 4K
Blockbelegung:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| Block: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Belegt: Block 0, Block 2, Block 7
Anfoderung von 10 K RAM, d.h. 3 Blöcke: z.B. Block 3 – 5

**Analyse:**
Speicherbedarf:
- Blockgröße 4 Byte (32 Bit): 1/32 ≈ 3%
- Blockgröße 512 Bytes: 1/(512*8) = 1/4096 ≈ 0,024%
- Bei 1 MB ($2^{20}$ Bytes) Speichergröße und einer Blockgröße von 512 Bytes: 2048 Blöcke, 256 Bytes Speicher werden benötigt
- Bei 1 GB ($2^{30}$ Bytes) Speichergröße und einer Blockgröße von 512 Bytes: 2 Millionen Blöcke, 260 kB werden benötigt

Laufzeit:
- Bei 8086: infolge fehlender Befehle nur aufwändig möglich -> für MS-DOS nicht geeignet trotz Verwendung im Vorgänger CP/M
- Bei 386 oder 68000: kein Problem

*7.4.4.2 Speicherverwaltung über lineare Listen (Freikette)*

In einer verketteten Liste (Freikette) werden alle freien Blöcke gespeichert.
**Algorithmus:**
- Anfangszustand: gesamter Speicher ist frei, d.h. die Freikette enthält 1 Listenelement mit dem gesamten (freien) Speicher
- Neue Anforderung von n Bytes:
    - suche einen Element in der Freikette mit mehr als n Bytes
    - Teile diesen Bereich in zwei Teile: *n* Bytes zur Erfüllung der Anforderung, Rest wird neues Element in der Freikette
- Freigabe von Speicherbereich: füge neuen freien Speicher in die Freikette ein, anschließend eventuelle Verschmelzung von benachbarten Freibereichen. Dabei sind 4 Fälle zu unterscheiden:
    - keine Verschmelzung möglich: der freigegebene Speicherbereich ist links und rechts von belegtem Speicher umgeben
    - Verschmelzung mit linkem (oder rechtem) Speicherbereich: der linke (oder rechte) Speicherbereich ist frei und wird mit dem neu freigegebenem Speicher verschmolzen.
    - Verschmelzung mit linkem und rechtem Speicherbereich: links und rechts von dem freigegebenen Speicher sind Freibereiche. Dann können alle 3 Bereiche (links, rechts und neu freigegeben) zu einem Element in der Freikette verschmolzen werden.

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

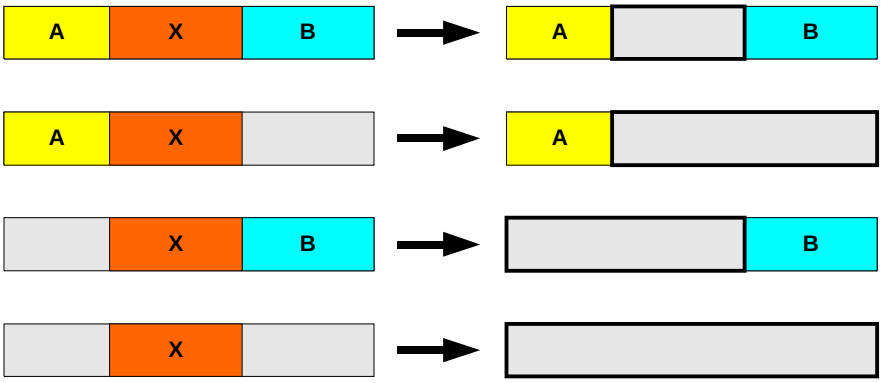



**Abbildung 73: Verschmelzung von freien Bereichen (nachdem Prozess X terminiert)**

### 7.4.5 Algorithmen zur Auswahl von Speicher aus der Freikette

Wenn eine Speicheranforderung an das BT gestellt wird, so muss aus den verfügbaren freien Speicherbereichen ein Bereich ausgewählt werden. Dazu werden verschiedene Algorithmen verwendet.

#### *7.4.5.1 First Fit*

**Algorithmus:**
Die Liste der verfügbaren Speicherbereiche wird durchsucht; der erste, passende Speicherbereich wird verwendet

- Die Suche geht immer vom Anfang des Speichers aus, der niedrige Speicherbereich wird stark fragmentiert. Am Ende sammeln sich größere freie Bereiche an.
- Einfachster, und sehr effektiver Algorithmus.

#### *7.4.5.2 Best Fit*

**Algorithmus:**
Die gesamte Liste wird durchsucht; der kleinste freie Speicherbereich, der die Anforderung erfüllen kann, wird verwendet
Auswirkung:
- Die Freikette muss vollständig durchsucht werden. Dies kann durch eine Anordnung verbessert werden: die Freikette wird nach Größe sortiert. Durch binäre Suche kann schnell der passende freie Speicherbereich gefunden werden.
- Nachteil: es entstehen viele kleine (nicht weiter verwendbare) Blöcke

#### *7.4.5.3 Worst Fit:*

**Algorithmus:**
Der größte freie Speicherbereich wird verwendet.
Auswirkung:
- Nach einiger Zeit keine großen freien Bereiche mehr vorhanden.

#### *7.4.5.4 Problem: Externer Verschnitt*

In allen Fällen tritt externer Verschnitt auf!

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

**Beispiel:**
Speicher: 1 MB, Algorithmus: First Fit
Anforderungen:
1. 200K: Bereich 0 – 200K
2. 100K: Bereich 200K – 300K
3. 200 K: Bereich: 300K – 500K
4. 100K: Bereich 500K – 600K
5. Freigabe von 1.
6. Freigabe von 3.
7. 500K: geht nicht, obwohl nur 800K frei sind!

**Speicherabbild (nach Schritt 6):**

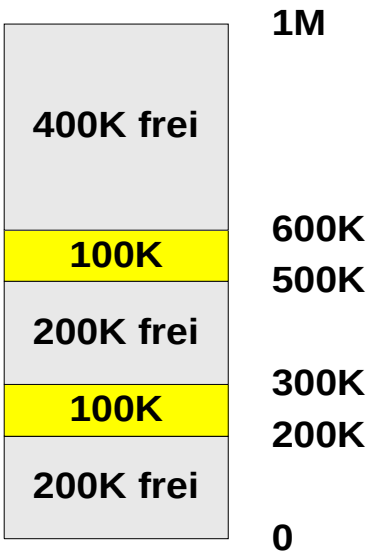



**Abbildung 74: Speicherfragmentierung**

*7.4.5.5 Buddy-System nach Knuth*

**Ziel:**
- freien Speicher schnell finden
- effizientes Verschmelzen
- Fragmentierung begrenzen

Algorithmus:
- Der Speicher wird in Einheiten von $2^n$ vergeben (Anforderung von 100 Bytes -> Zuteilung von 128 Bytes)
- Für jeden Topf mit Elementen der Länge $2^i$ gibt es eine Freikette ($i = 1..n$)
- Anfangszustand: alle Freiketten sind leer, bis auf die mit den höchsten Topf
- Anforderung von x bytes:
    - x wird auf die nächst-höhere Zweierpotenz $2^i$, $i < n$, aufgerundet. Anschließend wird die Speicheranforderung aus der i-ten Freikette erfüllt.

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

  - Ist die i-te Freikette leer, so wird aus der nächst-höheren Freikette i+1ein freier Speicherbereich entnommen und in zwei Hälften geteilt. Ein Teil wird zur Befriedigung der Anforderung verwendet, der andere wird in der i-ten Freikette als frei gespeichert. Falls die Freikette i+1 leer ist, so wird eine höhere, nicht-leere Freikette j>i gesucht und diese heruntergebrochen.
- Freigabe: wird ein Speicherbereich freigegeben, so wird er wieder in die i-te Freikette zurückgegeben.
  - Nur wenn die andere Hälfte, aus der er ursprünglich durch Teilung entstanden ist, auch in der Freikette vorhanden ist, werden beide Teile wieder verschmolzen und an die nächst-höheren Freikette i+1 zurückgegeben.
  - Zwei Bereiche der Freikette i können dann verschmolzen werden, wenn sich Anfangsadressen nur in einem Bit unterscheiden: die Bits 0..i-1 sind Null, Bit i ist unterschiedlich, und alle höheren Bits müssen gleich sein.
  - Dies ist in Abbildung 75: Speicheraufteilung nach dem Buddy-System dargestellt: die Zahlen bezeichnen die Anfangsadresse der jeweiligen Speicherbereiche (die unteren 16 Bits wurden weggelassen).

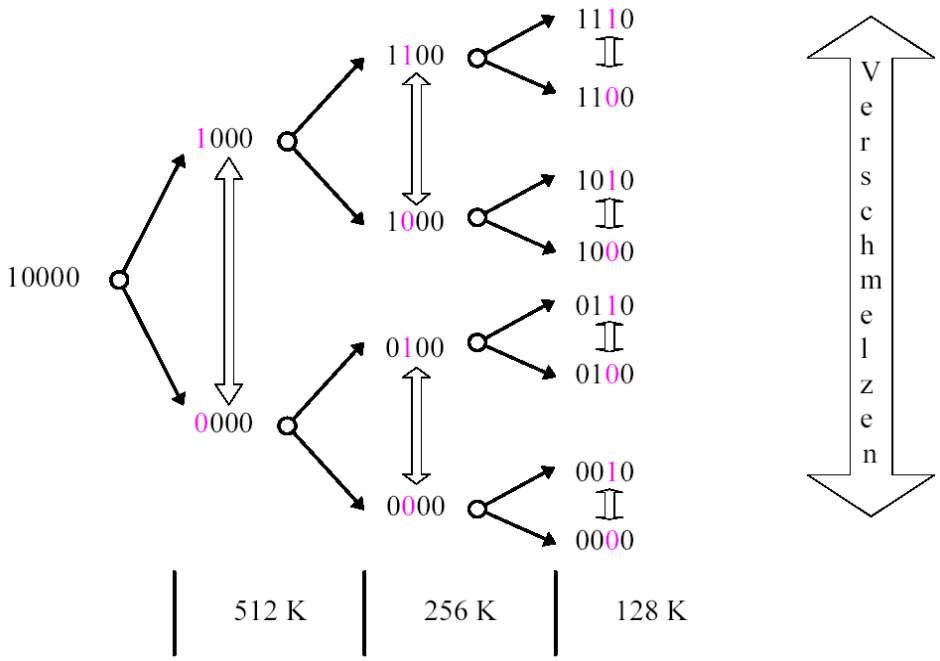



**Abbildung 75: Speicheraufteilung nach dem Buddy-System**

Vorteil: Speicher wird nicht so stark fragmentiert wie bei anderen Verfahren (externe Fragmentierung).
Jedoch interner Verschnitt, da immer auf Zweierpotenzen aufgerundet wird; durchschnittliche Fragmentierung liegt etwa 25 %.

**Beispiel:**
Ablauf: Anforderung 100K (1), 200K (2), 30K (3), 20K (4); Freigabe Anf. 2 (5); Anforderung 80K (6); Freigabe Anf. 3 (7); Anforderung 90 K (8); Freigabe Anf. 6 (9), Anf. 4 (10)
Bemerkungen:

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- In Zeile 1-3 wird jeweils ein freier Speicherbereich halbiert, ebenso in Zeile 6 und 7 sowie in Zeile 11
- In Zeile 21 und 22 werden jeweils benachtbarte Bereiche verschmolzen
- Die beiden 128K Bereiche in Zeile 22 können nicht verschmolzen werden, da sie nicht auseinander hervorgegangen sind.

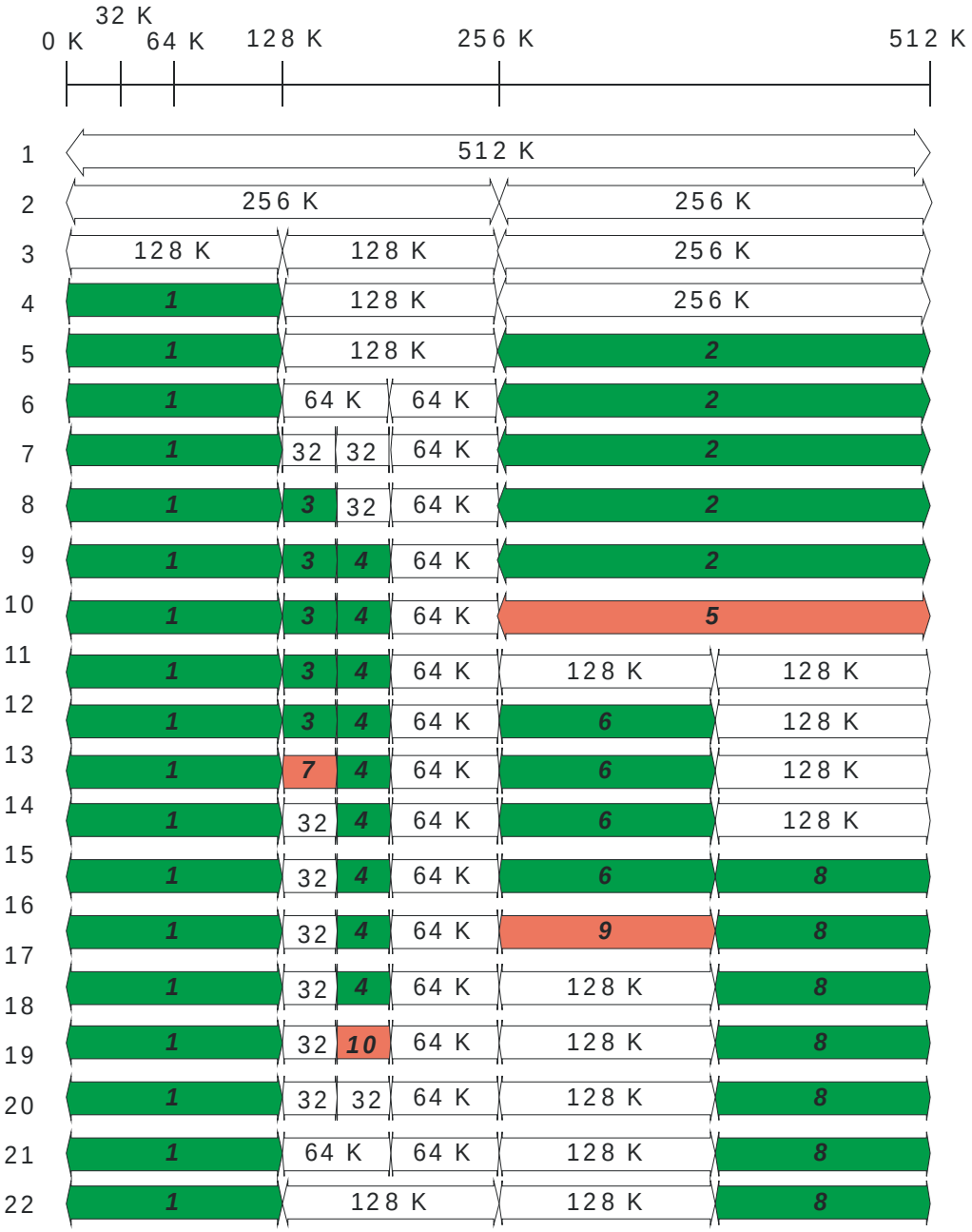



**Abbildung 76: Buddy-System (Beispiel)**

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

## *7.5 Virtueller Speicher: Paging*

**Virtueller Speicher:**

- Trennung zwischen logischer Adresse und physikalischer Adresse.
- Mehr logischer Speicher als physikalisch verfügbarer Speicher
- Mehreer Prozesse nutzen denselben physikalischen Speicher

### 7.5.1 Übersicht Paging

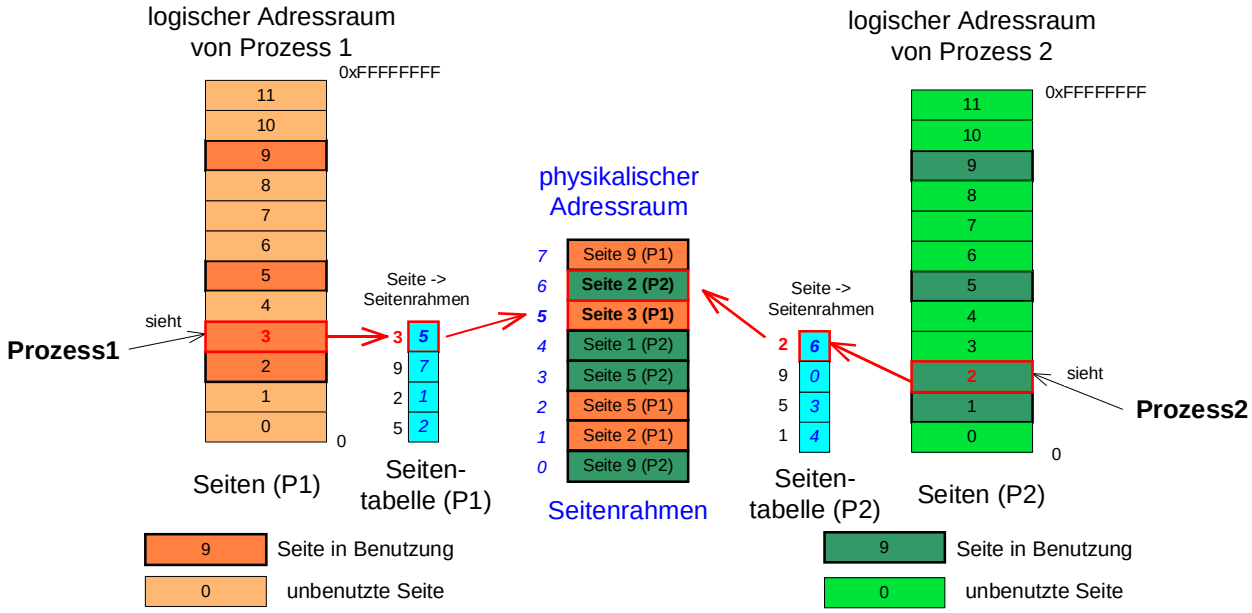



**Abbildung 77: Übersicht Paging**

- Der logische Speicher wird in gleich große Blöcke aufgeteilt (z.B. 4 kB)
- Der Prozess sieht einen zusammenhängenden Speicherbereich, der größer sein kann als der verfügbare physikalische Speicher
- Jeder Zugriff auf eine logische Adresse wird auf die entsprechende physikalische Adresse umgesetzt: Jeder Block wird über eine Seitentabelle einem Bereich im physikalischen Adressraum abgebildet
- Ist eine Seite nicht im Speicher, so wird ein System-Trap ausgelöst.

### 7.5.2 Umsetzung mittels Seitentabellen

- Logische Adresse wird geteilt:
  - obere Bits geben die Seitennummer an
  - untere Bits den Offset in dieser Seite

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

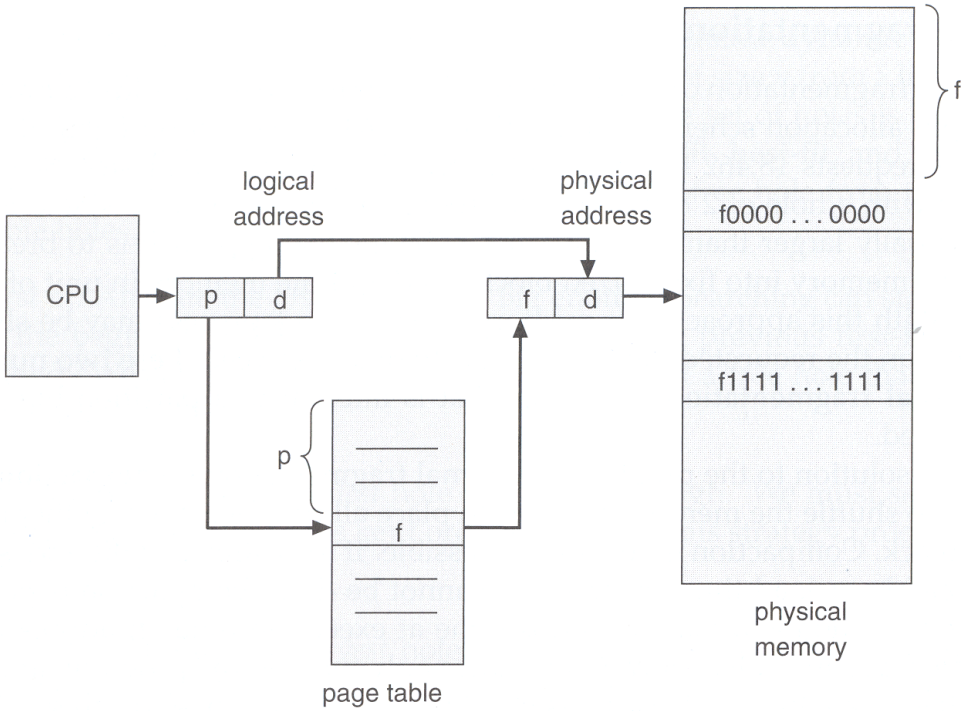



**Abbildung 78: Übersicht Paging MMU**

- In einer Tabelle werden zu jeder Seite der physikalische Seitenrahmen gespeichert
  - ist eine Seite nicht im physikalischen Speicher, so wird dies durch ein entsprechendes Bit vermerkt.
- Anhand der Seitennummer wird die zugehörige Seitennummer herausgesucht
  - Ist die Seite nicht im Speicher, so wird ein System-Trap ausgelöst
- Durch Addition der physikalischen Adresse der Seite und des Offsets wird die physikalische Adresse gebildet.

**Beispiel:**

- Seitengröße 4K (der Offset beträgt 12 bit, von 0 .. 4095 )
- logischer Speicher: 16 Seiten a 4K = 64K
- physikalischer Speicher: 32 K
- Seitentabelle siehe Abbildung 79: Beispiel Seitentabelle
- Die 16-bit Adresse 8196 = 0010.0000.0000.0100 wird wie folgt umgesetzt:
  - Offset (die unteren 12 Bit) gleich 4
  - Seitennummer (die oberen 4 Bits) gleich 2
- Heraussuchen der Seitenrahmennummer zur Seite 2: Seitenrahmennummer 6
- Berechnung der physikalischen Adresse:
  - Seitenrahmennummer 6 * 4096 (Seitengröße) ergibt 24576
  - plus Offset 4 = 24580
- Die logische Adresse 8196 wird umgesetzt auf die physikalische Adresse 24580

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

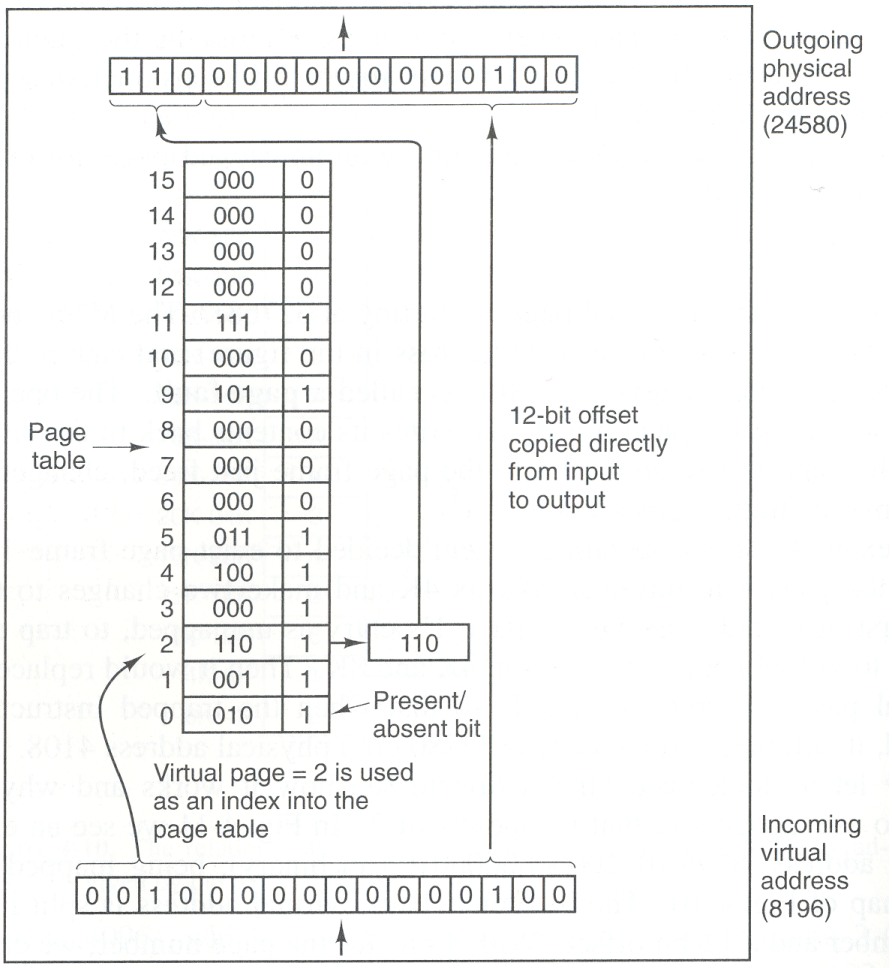



**Abbildung 79: Beispiel Seitentabelle**

Bemerkungen:

- Die Seitentabelle kann sehr groß werden:
  - 32-Bit Adressraum bei 4K Seitengröße: Seitentabelle hat 1 Million Einträge
  - Für jeden Prozess ist eine eigene Seitentabelle nötig!
  - 64-Bit Adressraum: sehr sehr viele Einträge im Adressraum...
- Die Umsetzung muss sehr schnell sein.
  - Für jeden Speicherzugriff ist eine Umsetzung nötig! Viele Prozessoranweisungen greifen ein oder mehrmals auf den Speicher zu, so das bei fast jedem Prozessorschritt auf die Seitentabelle zugegriffen werden muss.
  - Bei 1 GHz wird 1 Befehl pro Nanosekunde abgearbeitet, d.h. der Befehl dauert 1ns. Wenn ein Zugriff auf die Seitentabelle auch 1 ns dauert, so wird der Rechner nur halb so schnell sein!
- Problem: wo wird die Seitentabelle gespeichert?
  - In Prozessorregistern:
    - Vorteil: schneller Zugriff
    - Nachteil: nicht genügend Register vorhanden
  - Im Hauptspeicher:
    - Vorteil: große Seitentabellen möglich

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Nachteil: Zugriff auf die Seitentabelle bedeutet, dass die MMU zuerst die Seitentabelle aus dem Hauptspeicher holen muss, um dann den Seitenrahmen zu ermitteln; d.h. es sind zwei Speicherzugriffe nötig

### 7.5.3 Hierarchische Seitentabellen

- Um keine großen Seitentabellen im Speicher zu halten, wird eine Hierarchie von Seitentabellen verwendet.
- Die logische Adresse wird in 3 Bereiche aufgeteilt:
  - Die unteren 12 Bit sind wieder der Offset
  - Die oberen 20 Bit werden in zwei Felder aufgeteilt: 10 Bit Seitentabelle ST1, 10 Bit Seitentabelle ST2. Jede Tabelle enthält 1024 Einträge
  - Die Felder von ST1 (obere 10 Bit einer logischen Adresse) geben die Nummer einer Seitentabelle ST2 an
  - In dieser Seitentabelle ST2 wird nun mit den nächsten 10 Bit der Seitenrahmen ermittelt; zusammen mit dem Offset kann nun die physikalische Adresse gebildet werden
  - Trick dabei: Man muss nur ein paar der Seitentabellen ST2 im Speicher halten. Ein Prozess benutzt z.B. 4 MB am unteren Ende des Adressbereiches für den Programmcode, die nächsten 4 MB für Daten und die obersten 4 MB für den Stack. Dann müssen nur 4 Seitentabellen im Speicher gehalten werden (ST1 und 3 mal ST2), zusammen also 4*1024 gleich 4096 Einträge (anstelle 1 Million!)
- Auch 3 oder mehr Hierarchie-Ebenen möglich

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

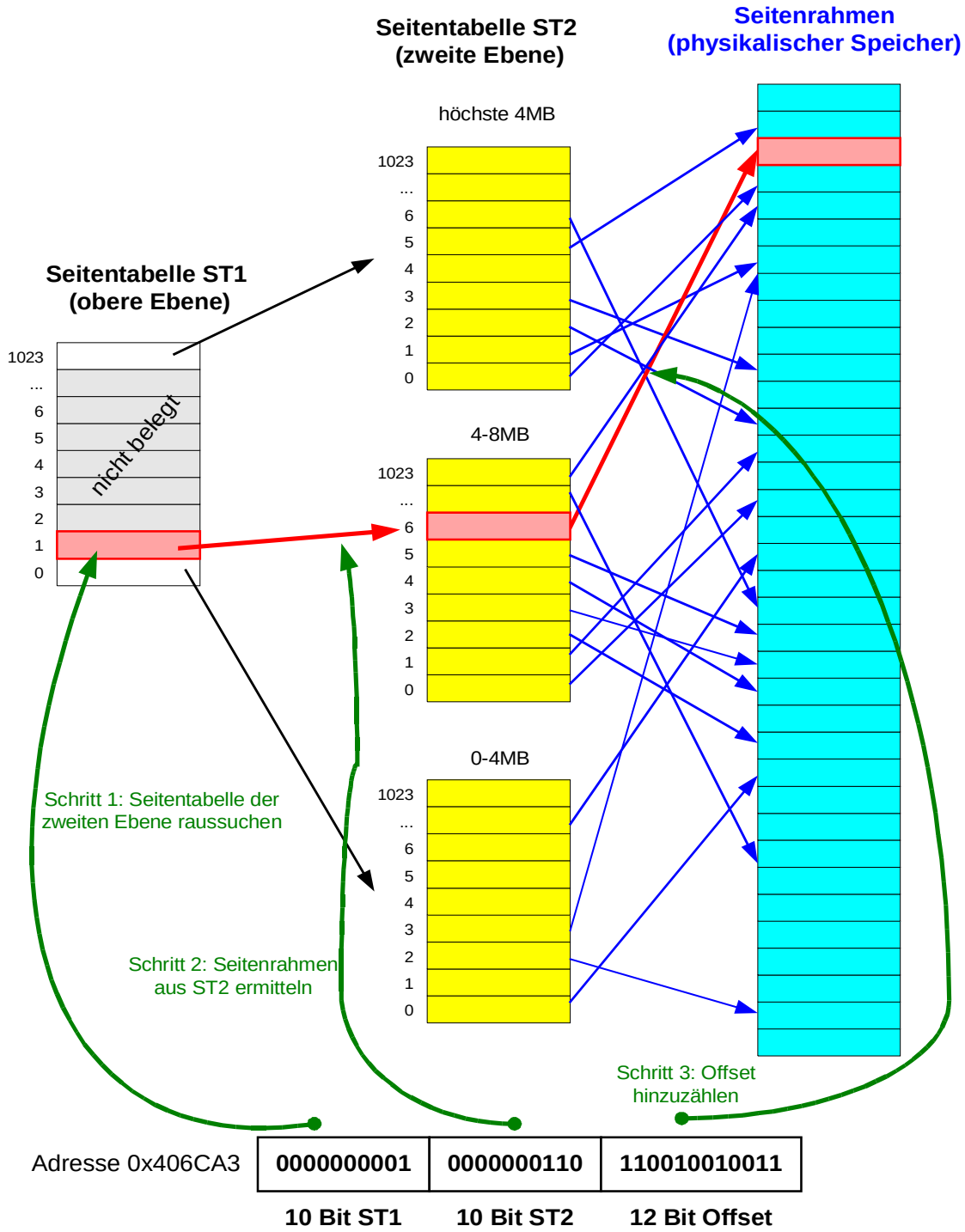



**Abbildung 80: Zwei-stufige Seitentabelle**

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

### 7.5.4 Einträge in einer Seitentabelle

Ein Eintrag in einer Seitentabelle enthält im Allgemeinen die folgenden Informationen (manche Einträge sind abhängig von der speziellen Hardware des Rechners):

- Wichtigster Eintrag ist natürlich die **Nummer des Seitenrahmens**
- **P-Bit (Present-Bit):** Ist die Seite überhaupt vorhanden? Falls nicht -> Seitenfehler -> MMU löst Interrupt aus, wird vom Betriebssystem bearbeitet
- **Seitenschutz (Protection):** Zugriffsrechte, z.B. Read-Only, Read-Write, Execute-Recht, etc. Kann 1-Bit oder mehrere Bits sein.
- **M-Bit (Modified-Bit):** Wurde die Seite geändert? Wird nach Schreibzugriffen auf die Seite gesetzt.
- **R-Bit (Referenced-Bit):** Wurde auf die Seite schon einmal zugegriffen? Dieses Bit wird nach einem Zugriff (lesend oder schreibend) auf die Seite gesetzt; wird für Seitentauschstrategien benötigt
- **Caching-Disabled Bit:** Diese Seite darf nicht gecached werden (nötig z.B. wenn die Seite auf ein Hardware-Gerät gemapped wird).

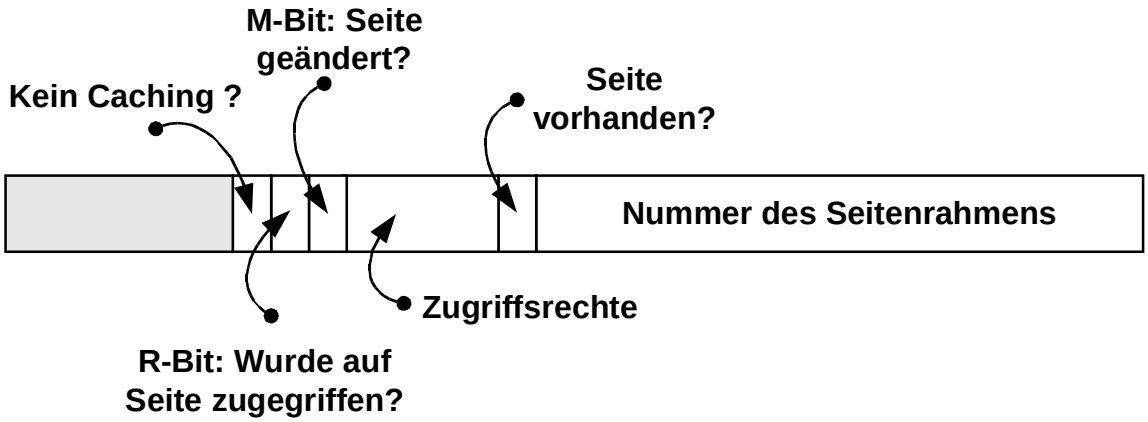



**Abbildung 81: Typischer Eintrage in einer Seitentabelle**

### 7.5.5 TLBs – Translation Lookaside Buffers

Heutezutage befinden sich die Seitentabellen meistens im Speicher. Bei einer zweistufigen Seitentabelle sind anstelle eines Speicherzugriffes drei nötig, d.h. das System ist um 2/3 langsamer!
Um dies zu vermeiden, wird eine spezielle Hardware verwendet, die *Translation Lookaside Buffers*. Diese speichern einige Seitentabelleneinträge in Spezialregistern. Durch die Verwendung von assoziativem Speicher können alle Einträge in einem Prozessortakt untersucht werden.
Ein Programm benötigt typischerweise nur eine geringe Anzahl von Seiten zu einem bestimmten Zeitpunkt. Deshalb reichen auch geringe Größen (selten mehr als 64) der TLBs aus, um eine drastische Performance-Steigerung zu erreichen.

**Ablauf:**

- Zugriff auf eine Speicherstelle
- Die logische Adresse wird in zwei Teile zerlegt

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Im TLB wird die Seitennummer gesucht
  - Falls im TLB vorhanden: bingo! (TLB hit)
  - Andernfalls in der Seitentabelle die Seite suchen und Seitenrahmennummer entnehmen (TLB miss)
  - Gegebenfalls die Seite ins TLB laden
- Aus Seitenrahmennummer und Offset physikalische Adresse berechnen

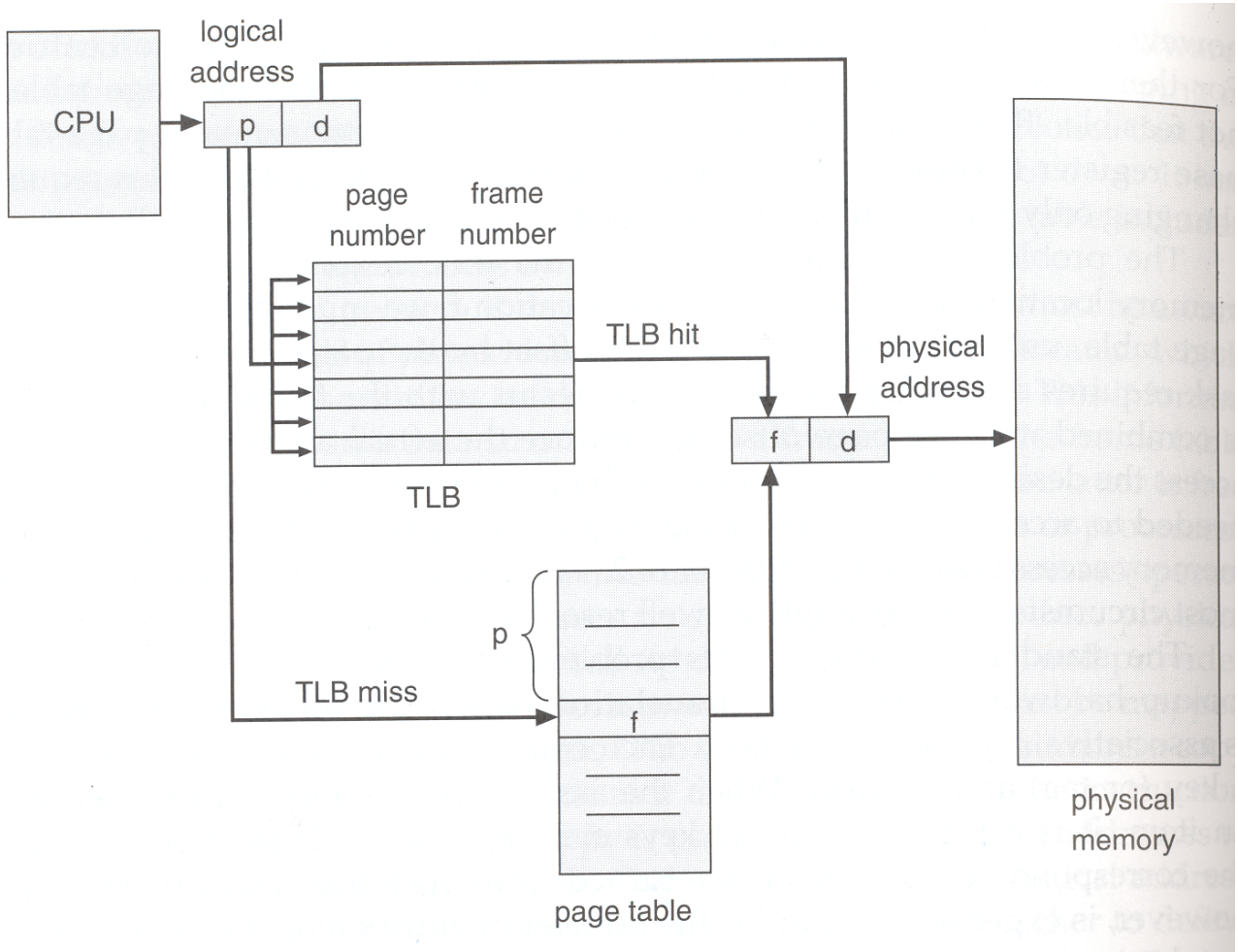



**Abbildung 82: Schema Translation Lookaside Buffer**

**Behandlung eines TLB-Fehlers in Software**:
Wenn ein TLB-Miss stattfindet, so wurde dies früher in Hardware behandelt (laden der Seite über die Seitentabelle).
Moderne Risc-Maschinen (SPARC, MIPS, Alpha, HP PA) implementieren einen TLB-miss jedoch in Software; eine Spezialhardware ist nicht mehr nötig. Wird ein Eintrag nicht im TLB gefunden, so wird ein TLB-Miss-Interrupt erzeugt, und das Betriebssystem kümmert sich darum:

- es lädt den entsprechenden Eintrag aus der Seitentabelle
- führt ggfs. ein Update der TLB durch

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- komplexe Algorithmen sind hier möglich, z.B. das vorausschauende Lader der TLB mit Seiten, die *vermutlich* als nächstes gebraucht werden.

Problem:
Bei einem TLB-miss muss die Seitentabelle befragt werden. Diese kann jedoch ebenfalls zu einem TLB-miss führen, was zu einer verschlechterten Performance führt.
Abhilfe:

- Es wird ein Cache verwendet, z.B. 1024 Einträge, der in eine 4KB-Seite reinpasst. In diesem Cache können TLB-Einträge zwischengespeichert werden, auf die dann schnell zugegriffen werden kann. Diese Seite wird nie aus dem TLB entfernt, kann als immer schnell zugegriffen werden.

### 7.5.6 Invertierte Seitentabellen

Bisher: Einträge in der Seitentabelle werden über die virtuelle Adresse indiziert, d.h. für jeden Block im virtuellen Speicherraum gibt es einen Eintrag. Auf einem 32-bit System mit maximal $2^{32}$ = 4.294.967.296 Bytes und einer Blockgröße von 4096 Bytes sind 1 Million Einträge in der Seitentabelle nötig, d.h. 4 MB Speicher.
Für 64-Bit Rechner ist das nicht so einfach. Bei $2^{64}$ Bytes und 4K Blockgröße sind $2^{52}$ Einträge in der Seitentabelle nötig; bei 8 Byte / Eintrag ergeben sich über *30 Millionen GB*! Das ist nicht so einfach zu realisieren.

Ein anderer Ansatz ist die invertierte Seitentabelle. Hier wird ein Eintrag *pro physikalischem Seitenrahmen* verwendet, anstelle einem Eintrag pro virtueller Seite. Die Größe des virtuellen Speicher spielt dann keine Rolle mehr; entscheidend ist der physikalische Speicher. Bei einer Seitenrahmengröße von 4K und 12 Bytes pro Eintrag müssen 0.3 % des physikalischen Speichers für die Seitentabelle verwendet werden.

**Beispiel:**
Ein Rechner mit 64-Bit virtuellem Adressraum, 4 KB Seitengröße und 1 GB RAM benötigt 262144 Einträge; bei 12 Byte / Eintrag sind das 3 MB.

**Nachteil:**

- Umsetzung von virtueller auf physikalische Adresse ist nicht mehr direkt möglich, sondern sie erfordert eine Suche in der invertierten Tabelle. Bei jedem Speicherzugriff ist dies nicht möglich; steht jedoch ein ausreichender TLB zur Verfügung steht, so ist ein schneller Speicherzugriff wieder möglich.
- Die Suche kann durch Hash-Tabellen oder andere Algorithmen weiter verbessert werden (siehe folgende Abbildung c) )

**Verwendung:**
Inverierte Seitentabellen werden gegenwärtig in einigen IBM und HP Rechnern verwendet; bei größerer Verbreitung von 64-Bit Rechnern werden sie häufiger eingesetzt werden.

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

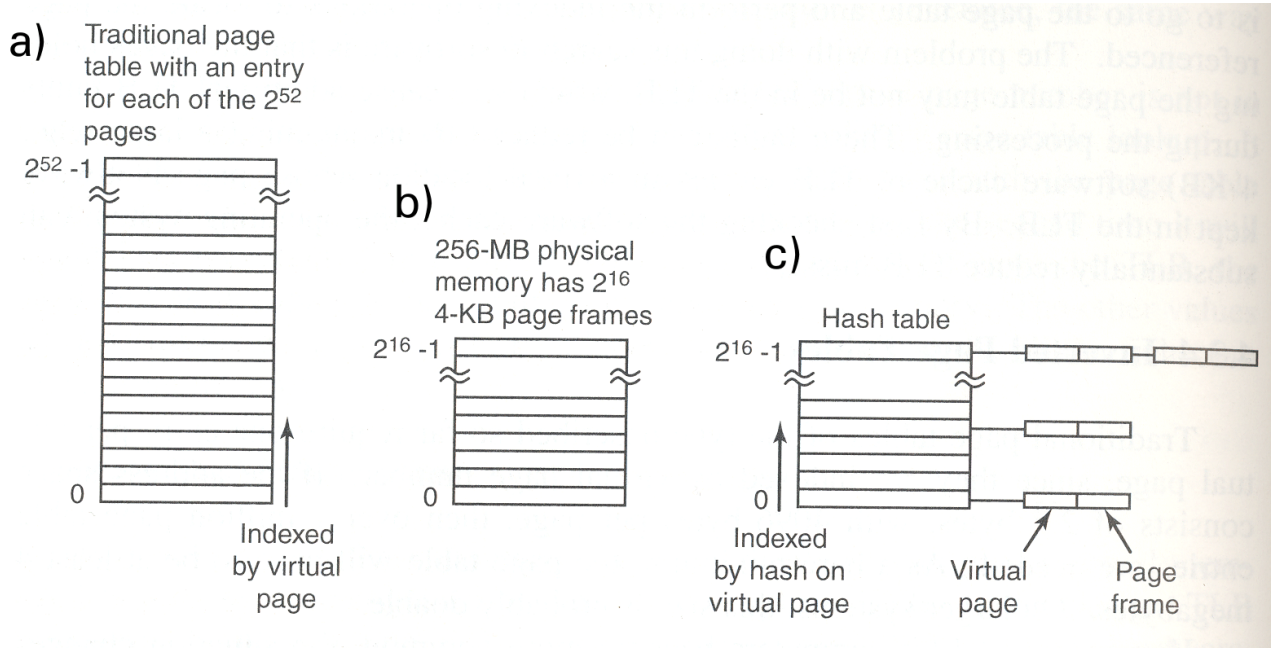



**Abbildung 83: Seitentabelle a) normal b) invertiert c) mit Hash-Tabelle (nach [1])**

### 7.5.7 Speicherschutz

Durch Paging kann Speicherschutz einfach realisiert werden:

- Jeder Prozess sieht einen eigenen Adressraum. Entweder er greift auf seine eigene Seite zu (das darf er), oder eine Seite ist in der Seitentabelle als nicht vorhanden markiert. Dann wird durch den Zugriff ein Seitenfehler ausgelöst & der Zugriff verhindert.
- Zudem können bestimmte Zugriffsrechte für jede Seite definiert werden, z.B. Read-Only, Execute-Only.

**Buffer-Overflow:**

Der Buffer-Overflow ist eine der häufigsten und gefährlichsten Angriffsarten von Viren, Würmern und Hackern.

In der Programmiersprache C werden lokale Daten (die nur innerhalb einer Funktion gebraucht werden), auf dem Programm-Stack abgelegt. Dort liegt jedoch auch die Rücksprungadresse, die nach Beendigung der Funktion aufgerufen wird.

Ein Buffer-Overflow-Angriff geht nun wie folgt:

Der Angreifer entdeckt eine Server- oder Betriebssystem-Funktion, die fehlerhaft programmiert ist: sie legt ein Datenfeld (z.B. String) auf dem Stack ab, überwacht jedoch nicht (oder nicht korrekt) die Länge des Datenfeldes. Der Angreifer manipuliert nun die Eingangsdaten in diese Funktion in der Art, dass der Stack überläuft. Legt die Funktion z.B. ein Datenfeld mit 100 Zeichen auf dem Stack an, so "füttert" der Angreifer die Funktion mit 200 Zeichen. Die ersten 100 Zeichen passen in das Datenfeld; die zweiten 100 Zeichen überschreiben jedoch andere Daten auf dem Stack. Gelingt es dem Angreifer, die Rücksprungadresse zu überschreiben, so kann er den Rücksprung der Funktion kontrollieren. Hat der Angreifer z.B. ausführbaren Code in den 200 Zeichen Eingangsdaten untergebracht, so kann der Angreifer dadurch die Kontrolle über das System übernehmen!

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Beispiele für einen Buffer-Overflow: **Code Red** (August 2001), **Nimda** (August 2001), **SQLSlammer** (Februar 2003). Alle diese Würmer haben sich über das Internet verschickt: die ersten beiden haben einen Buffer-Overflow im MS Internet Information Server verursacht, der SQLSlammer im MS SQL Server.

Einfache Schutzmassnahme durch Trennung von Daten und Code: Dadurch kann Code, der auf dem Stack gelegt wird, nicht ausgeführt werden.

Wird z.B. in OpenBSD verwendet (www.openbsd.org)

### 7.5.8 Gemeinsame Seiten

Mittels Paging kann gemeinsamer Speicher leicht realisiert werden:
Dieselbe Seite wird in mehrere Adressräume (von verschiedenen Prozessen) eingeblendet.
Änderungen in der Seite sind sofort in allen Prozessen sichtbar!
Achtung: bei Zugriff auf gemeinsamen Speicher sind Prozesssynchronisierungen nötig!

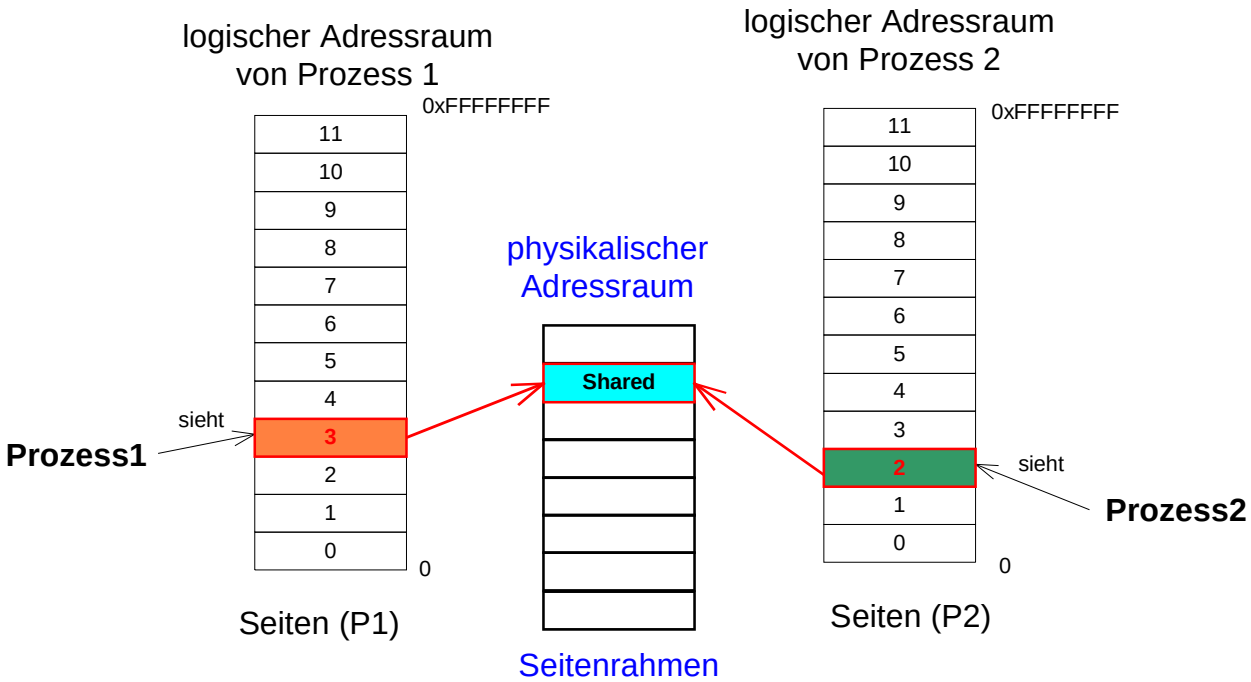



**Abbildung 84: Gemeinsamer Speicher durch Paging**

### 7.5.9 Demand Paging, Seitentauschstrategien, Seitenflattern

-> Siehe Powerpoint-Folien!

### 7.5.10Anforderungen an den Prozessor

Bei Seitenfehler: Unterbrechung der Prozessor-Anweisung -> muss später wiederholt werden!
Problem: nicht immer möglich!

**Beispiel:** Spezialbefehle die ganze Speicherbereiche kopieren, eg.

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

```
STRMOV Zieladresse, Quelladresse, Anzahl Bytes
```

Der Befehl kppiert einen ganzen Speicherbereich von der Quell- zur Zieladresse. Tritt dabei ein Seitenfehler auf, so muss der Befehl wieder neu gestartet werden; dies ist jedoch nicht immer möglich, da z.B. Quell-und Zielbereich überlappen können und Daten schon kopiert wurden.

**Beispiel:** indiziertes Lesen: lade den Speicherinhalt, auf den R1 zeigt, an die Speicherstelle, auf die R2 zeigt; dabei wird R1 nach der Anweisung um eins erhöht, und R2 vor der Anweisung um eins erniedrigt:

```
MOV (R1)+, -(R2)
```

Die Anweisung enthält 3 Speicherzugriffe; nach einer Unterbrechung ist unklar, welche Register verändert wurden

-> Prozessor muss für virtuelles Speichermanagement geeignet sein!
**Abhilfe:**
- Schattenregister speichern den Prozessorzustand; dann kann der Zustand nach einem Seitenfehler wieder hergestellt werden

### 7.5.11Nebeneffekte des Paging

- Paging ist transparent für den Benutzer. Aber: Nebeneffekte sind möglich!
- Paging kann zu Verzögerungen des Programms führen -> ungeeignet für Echtzeitsysteme

Beispiel: Seitengröße 4kByte; Programm benötigt Feld mit 1024 mal 1024 Integer-Zahlen

```
int i, j, a[1024][1024];
// schnelle Schleife:
for(i=0; i<1024; i++)
  for(j=0; j<1024; j++)
    a[i][j] = 0;
```

```
// langsame Schleife:
for(i=0; i<1024; i++)
  for(j=0; j<1024; j++)
    a[j][i] = 0;
```

Links 1024 Seitenanforderungen, rechts 1024*1024 !

## *7.6 Segmentierung*

Kein linearer Speicher, sondern verschiedene Segmente. Jede Adresse wird durch zwei Angaben gebildet: Segment und Offset im Segment. Die CPU kennt und arbeitet mit verschiedenen Segmenten.
Typische Segmente:
- globale Daten
- Stack
- Programmcode für jede Funktion
- lokale Variablen jeder Funktion

**Dr. Ulrich Margull**
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

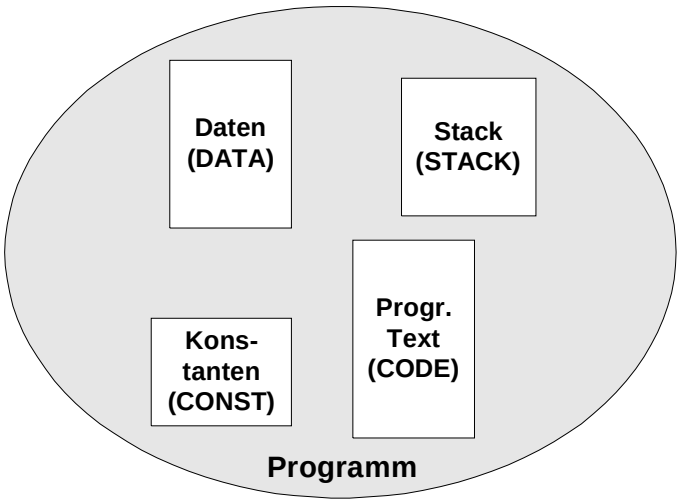



**Abbildung 85: Verschiedene Segmente eines Programmes**

Abbildung der <Segment, Offset> Adresse auf die physikalische Adresse:

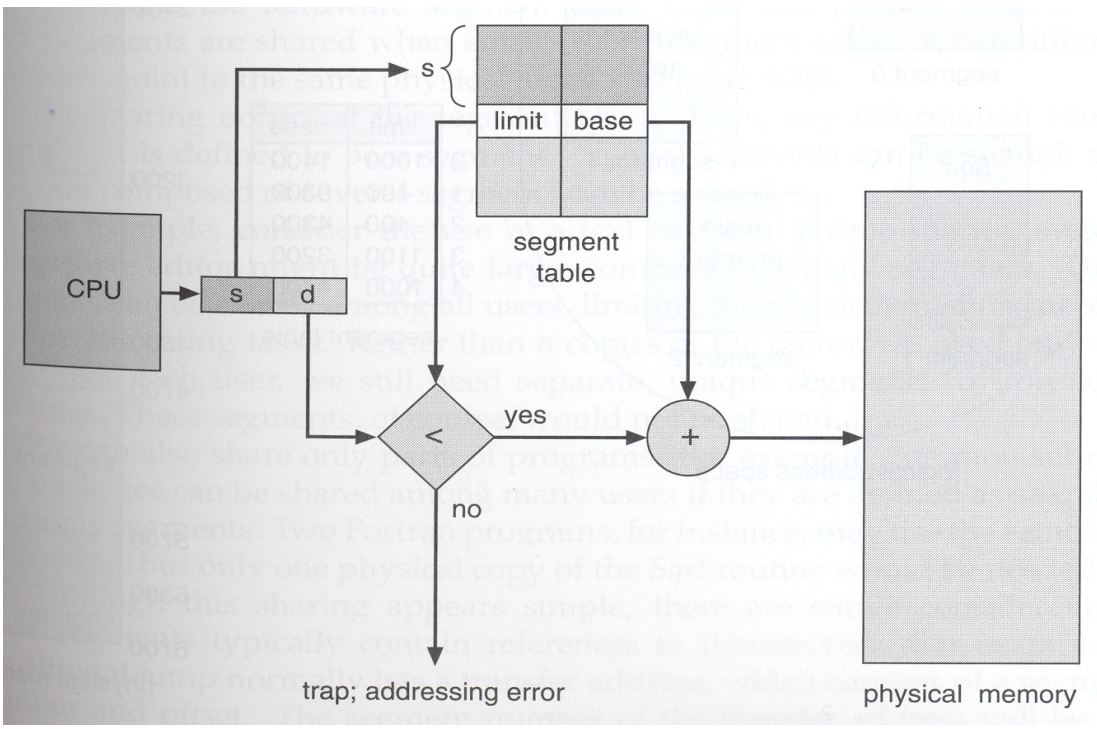



**Abbildung 86: Segemente**

Auch Kombinationen von Segmentierung und Paging möglich.