4 Speicherverwaltung

4.1 Grundlagen

4.2 Zusammenhängende Speicherung

4.3 Segmentierung

4.4 Seitenadressierung, gestreute Speicherung (*paging*)

4.5 Beispiel: Wechselpuffer

sp1/96/folien/3: 18.12.1996

4.1 Grundlagen

- Bereiche aufeinanderfolgender Adressen werden als Segmente bezeichnet.

- Während der Ausführung eines Aktivitätsträgers müssen wenigstens die folgenden drei Informationsklassen im Arbeitspeicher bereit stehen:

  - Ein Befehlssegment, das die auszuführenden Befehle enthält.

  - Ein Datensegment, das die zu bearbeitenden Daten enthält.

  - Ein Stapelsegment (*stack*), das zu organisatorischen Zwecken herangezogen wird, z. B. zur Organisation von Prozedur- und Methodenaufrufen.

- Programme werden häufig aus Gründen der Übersichtlichkeit und der Aufteilbarkeit bei der Implementierung aus mehreren Teilprogrammen (Modulen, Objekten) zusammengesetzt.
  - Wie und wann werden die Speicheradressen in den Befehlen ermittelt, ein Vorgang der als **Binden** bezeichnet wird?
    Drei wesentliche Möglichkeiten:
    - Während des Compilierens: **absolutes Binden**

      Setzt voraus, daß bereits während des Compilierens die endgültige Lage im Arbeitsspeicher bekannt ist.

      Wird im wesentlichen nur bei eingebetteten Systemen (*embedded systems*) verwendet.

    - Während des Ladens in den Arbeitsspeicher:
      statisches Binden

      Setzt voraus, daß Compiler und Assembler Informationen mitliefern, die eine Anpassung an die endgültige Lage im Arbeitsspeicher erlauben (*relocation information*).

    - Während des Programmablaufs:
      dynamisches Binden

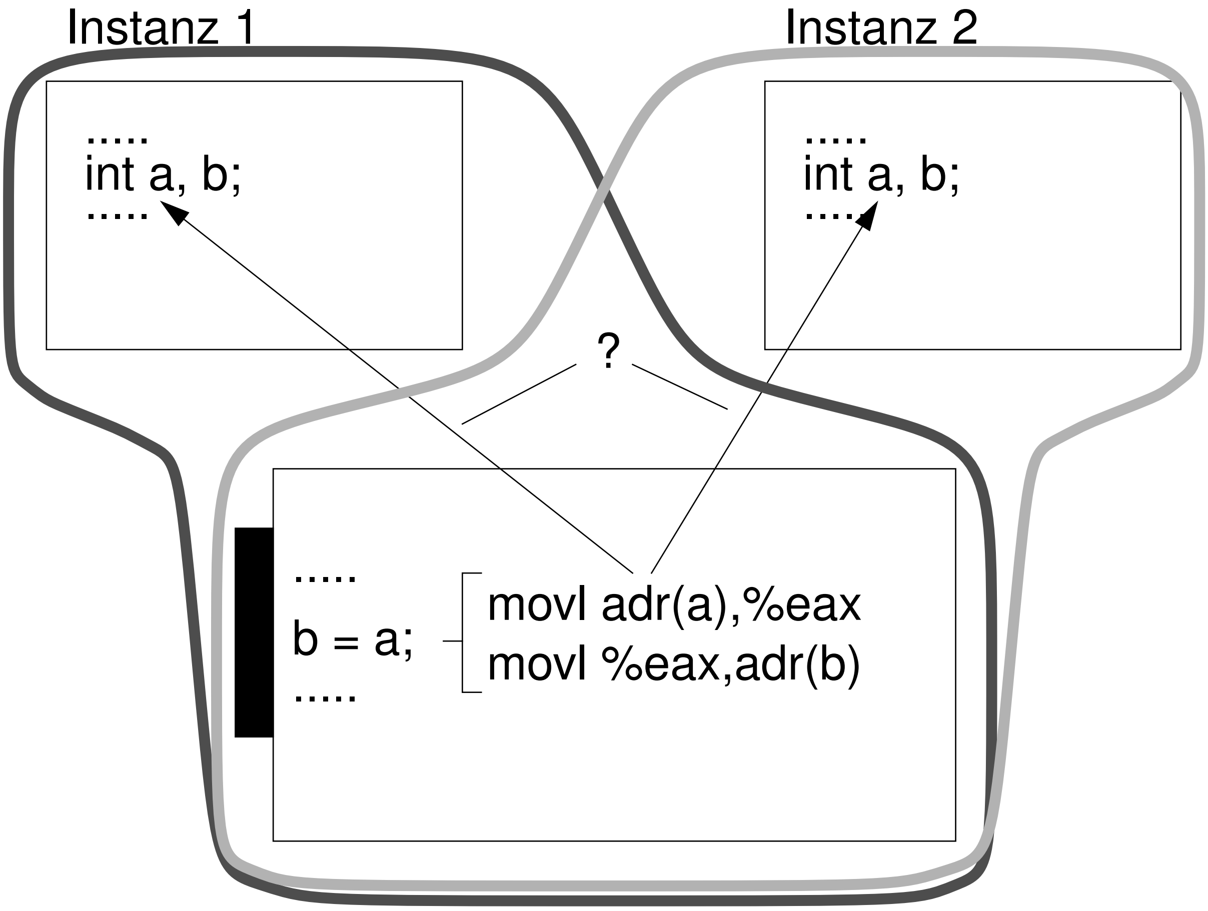
      Erlaubt mehrfache Nutzung von Befehlssegmenten.

      Realisierbar durch Verwendung:

      indizierter Adressierung

      geeigneter, noch zu besprechender Hardwaremechanismen, die auf der Assemblerebene transparent sind.

Bindung Befehlsadressen - Datensegment mittels Indizierung bei objektorientierter Programmierung



Mögliche (wenn auch unbefriedigende) Lösung:

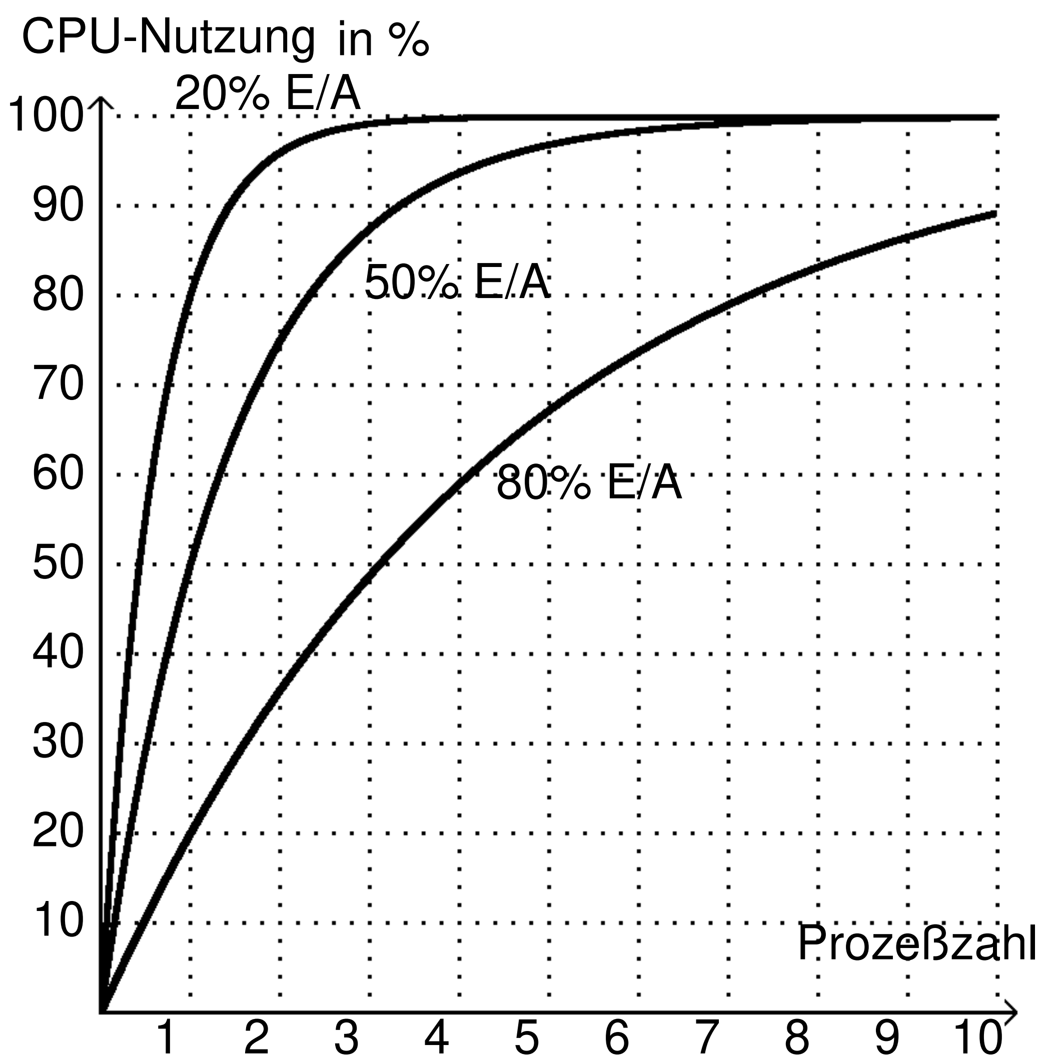
```
                          basis:
  0000 00000000           a:         .long   0
  0004 00000000           b:         .long   0

                          start:
  0008 8D 2D 00000000      leal basis,%ebp
  000e 8B 85 00000000      movl a-basis(%ebp),%eax
  0014 89 85 04000000      movl %eax,b-basis(%ebp)
```

4.2 Zusammenhängende Speicherung

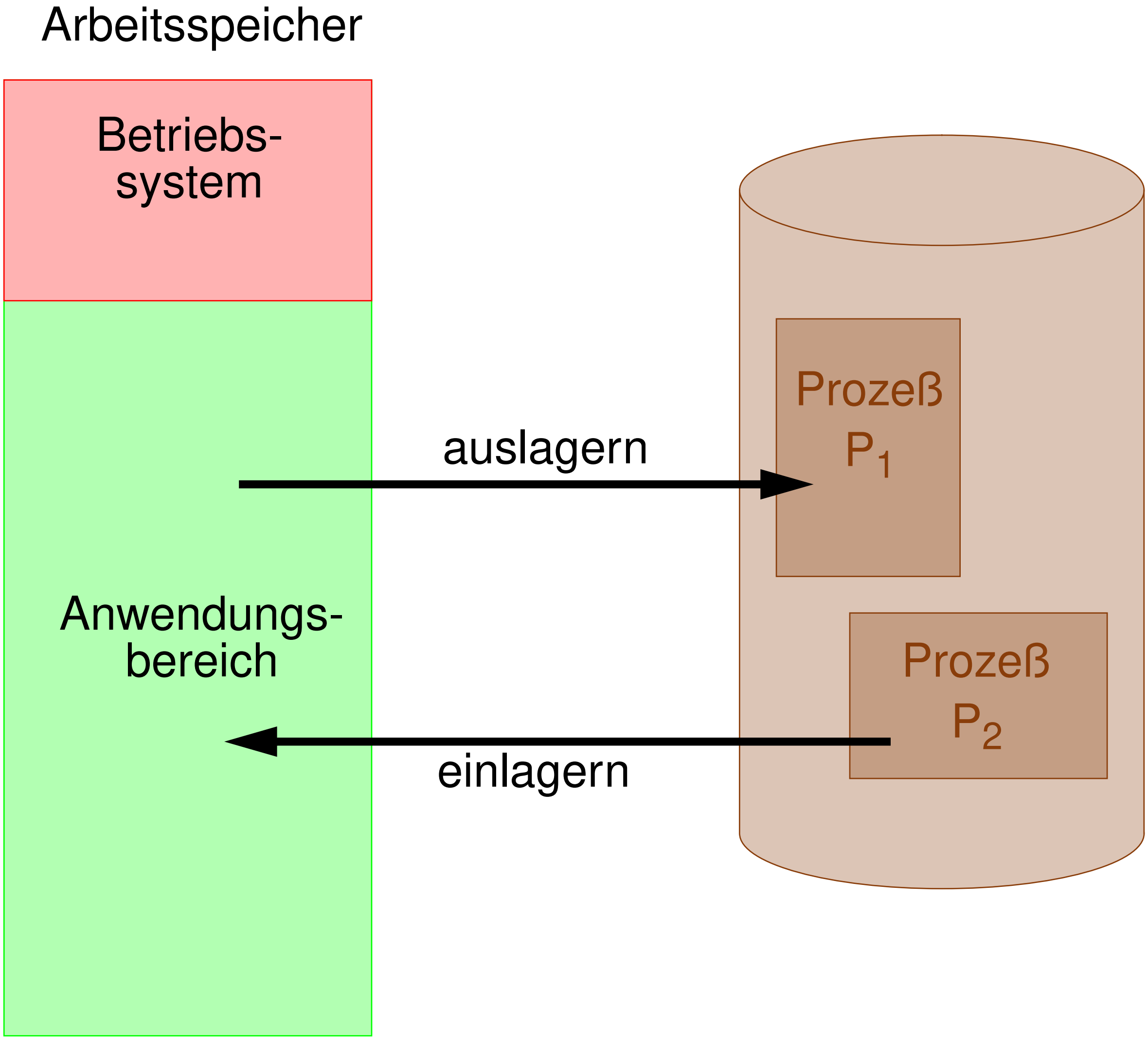
- Ein-Programmbetrieb

  Ein Programm im Arbeitsspeicher von Beginn bis Ende seiner Abarbeitung

- Verbesserung der Speichernutzung durch Mehrprogramm-betrieb

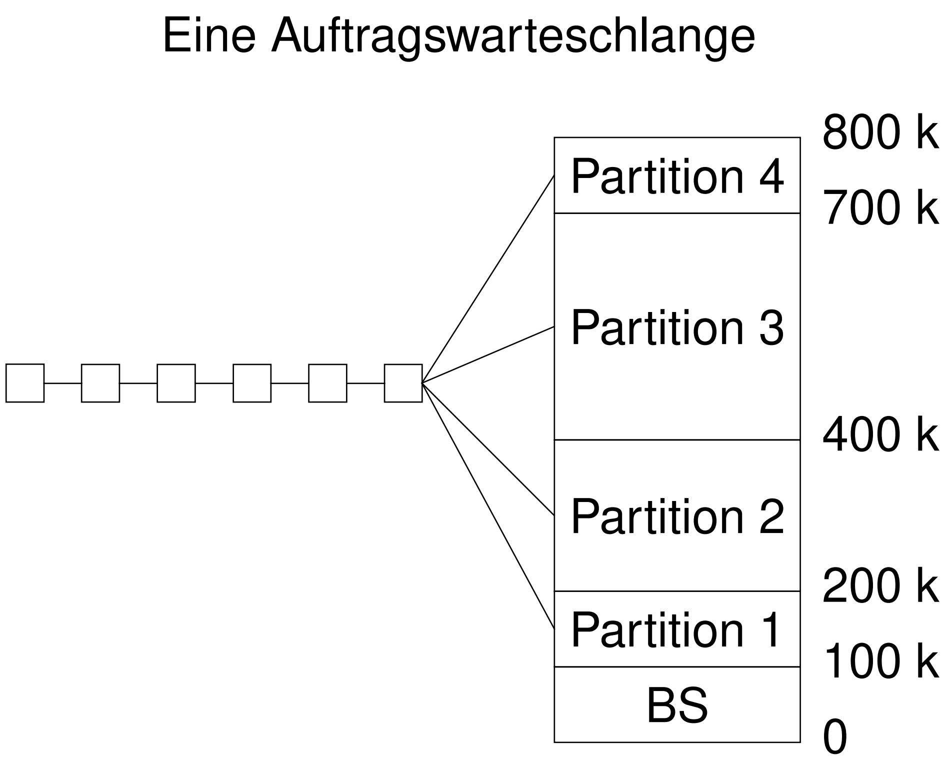


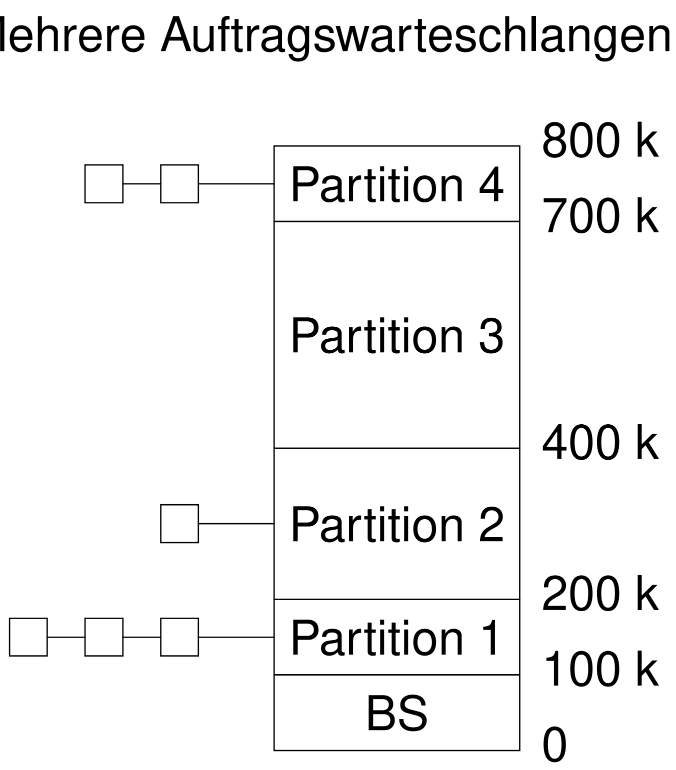
- swapping



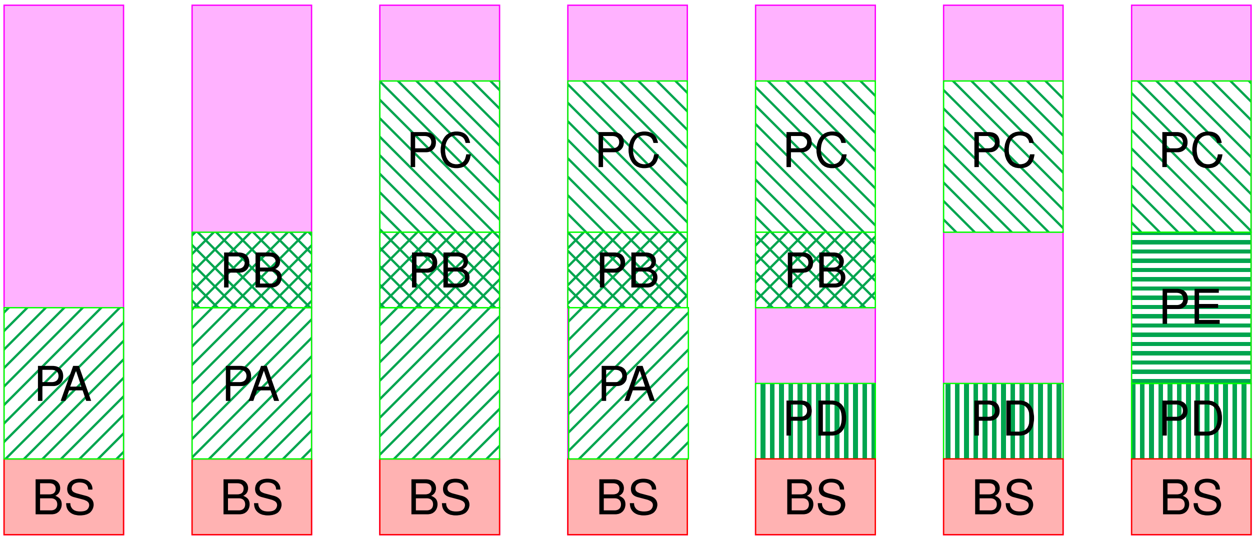
Prozeßwechsel wegen der Datentransporte sehr aufwendig.

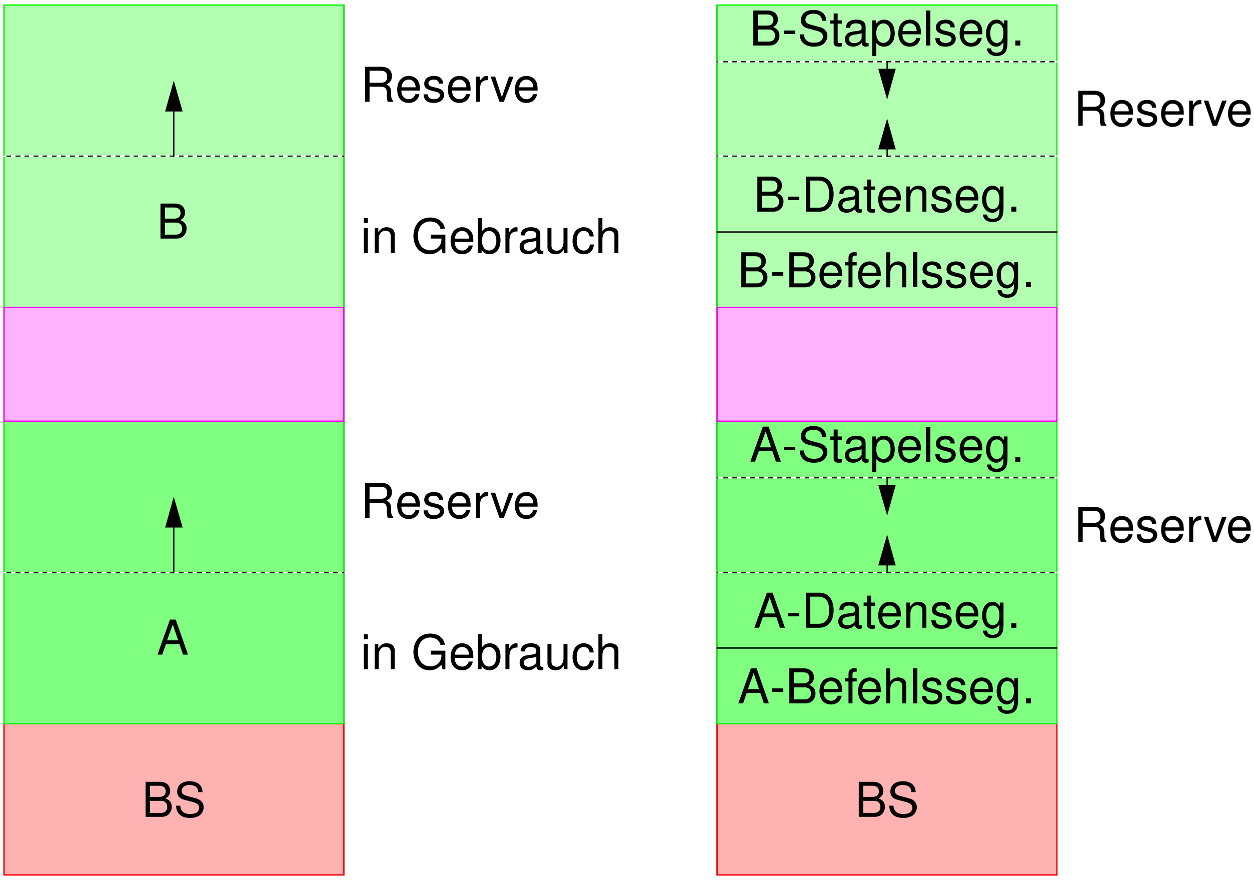
- Statische Partitionierung

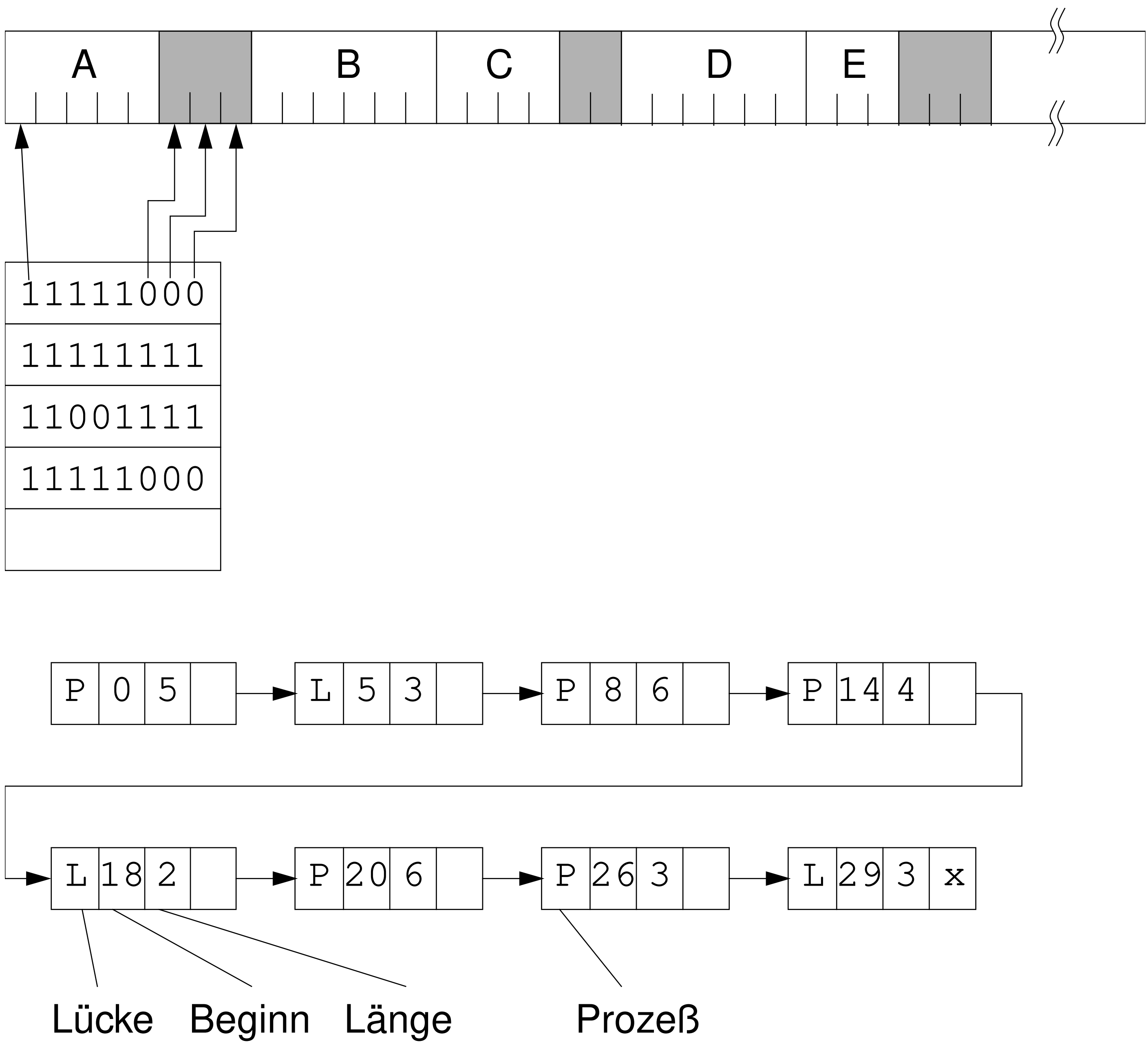




- Dynamische Partitionierung

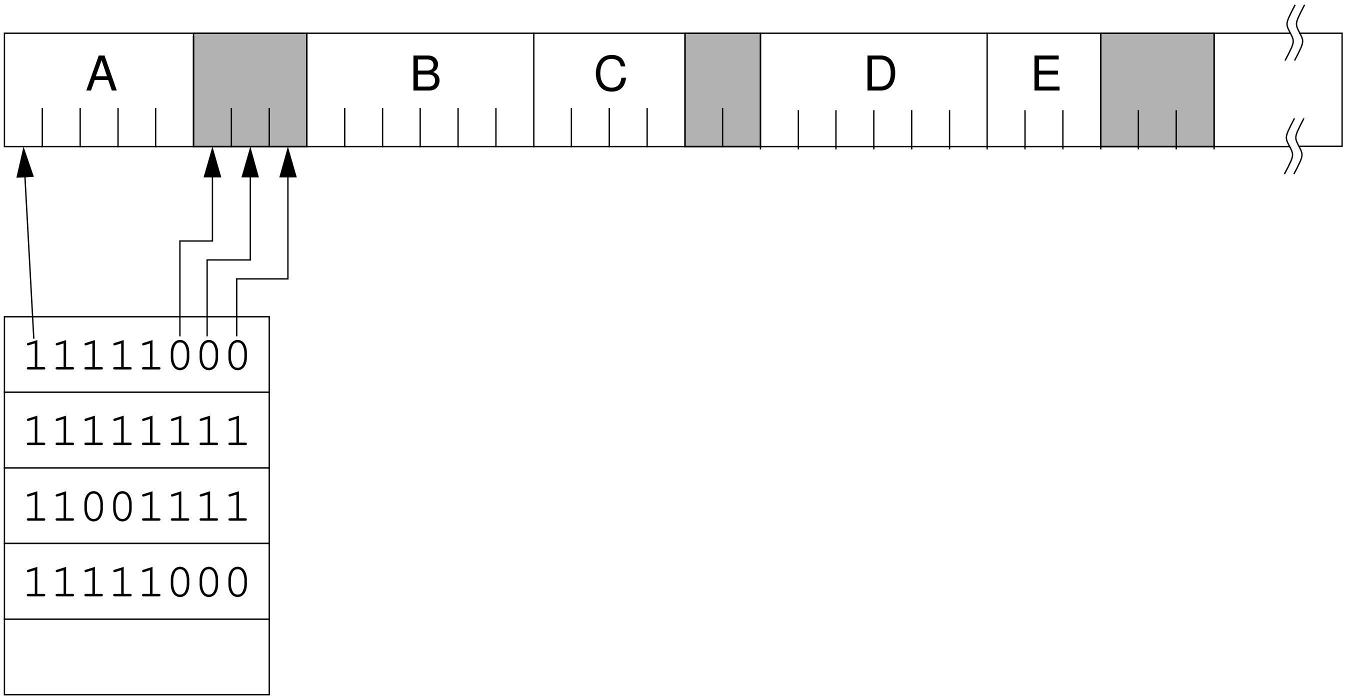




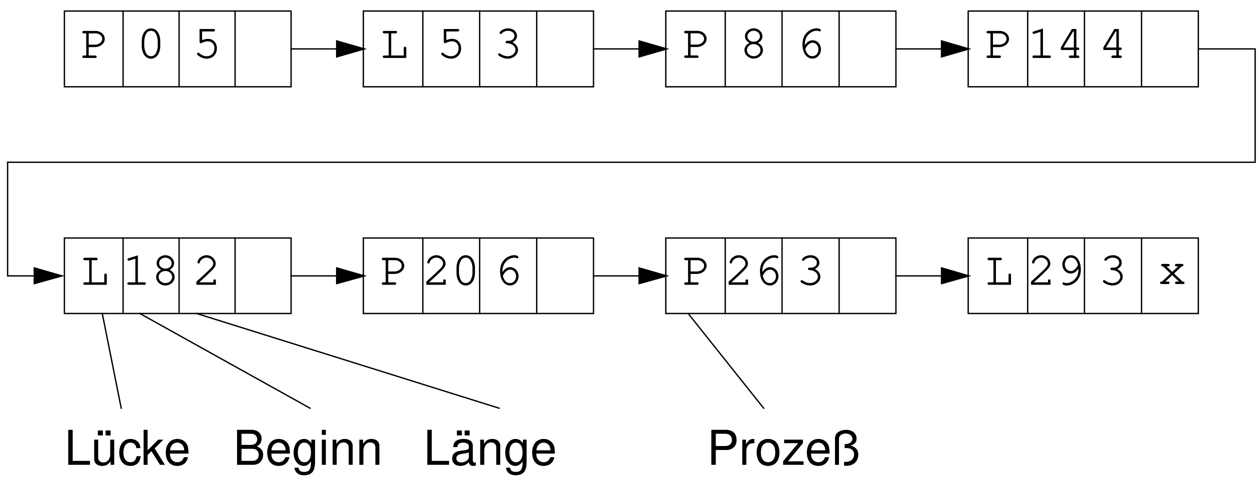


- Datenstrukturen zur Lückenverwaltung

Bitlisten



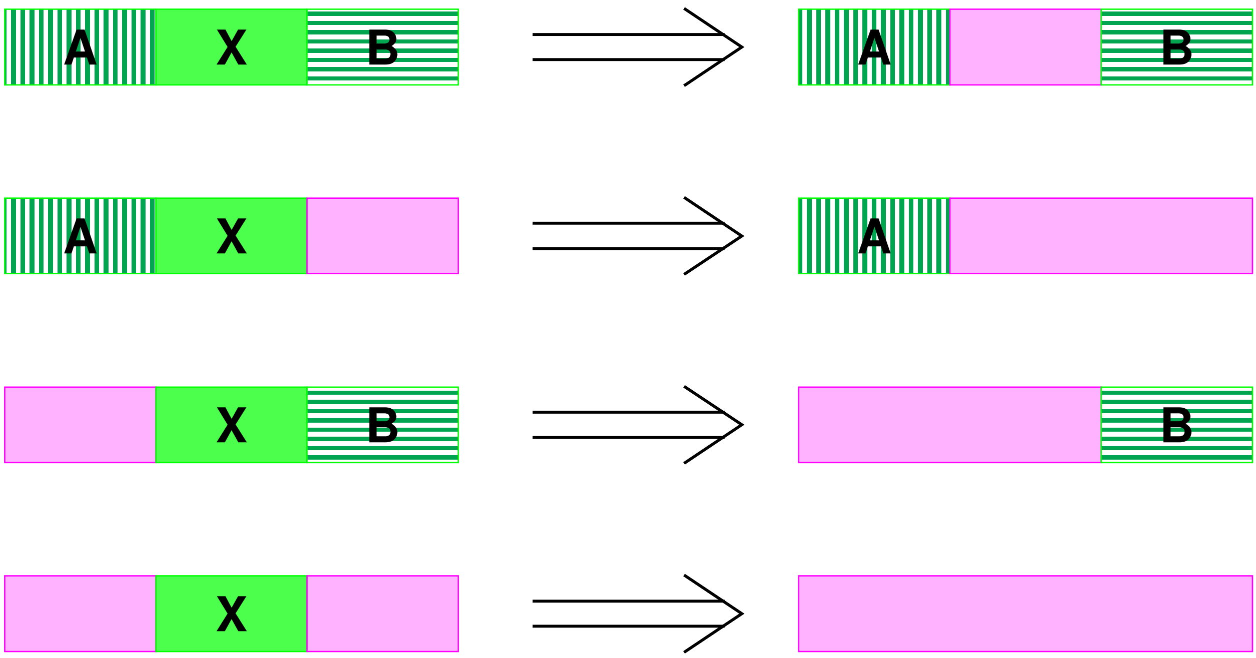
Verkettete Listen



- Plazierungsstrategien
  - Bei Freigabe Lücken verschmelzen

vor **nach**

Freigabe von X



  - Vergabe von Lückenrand aus

    First Fit

    Best Fit

    Worst Fit

    Rotating First Fit

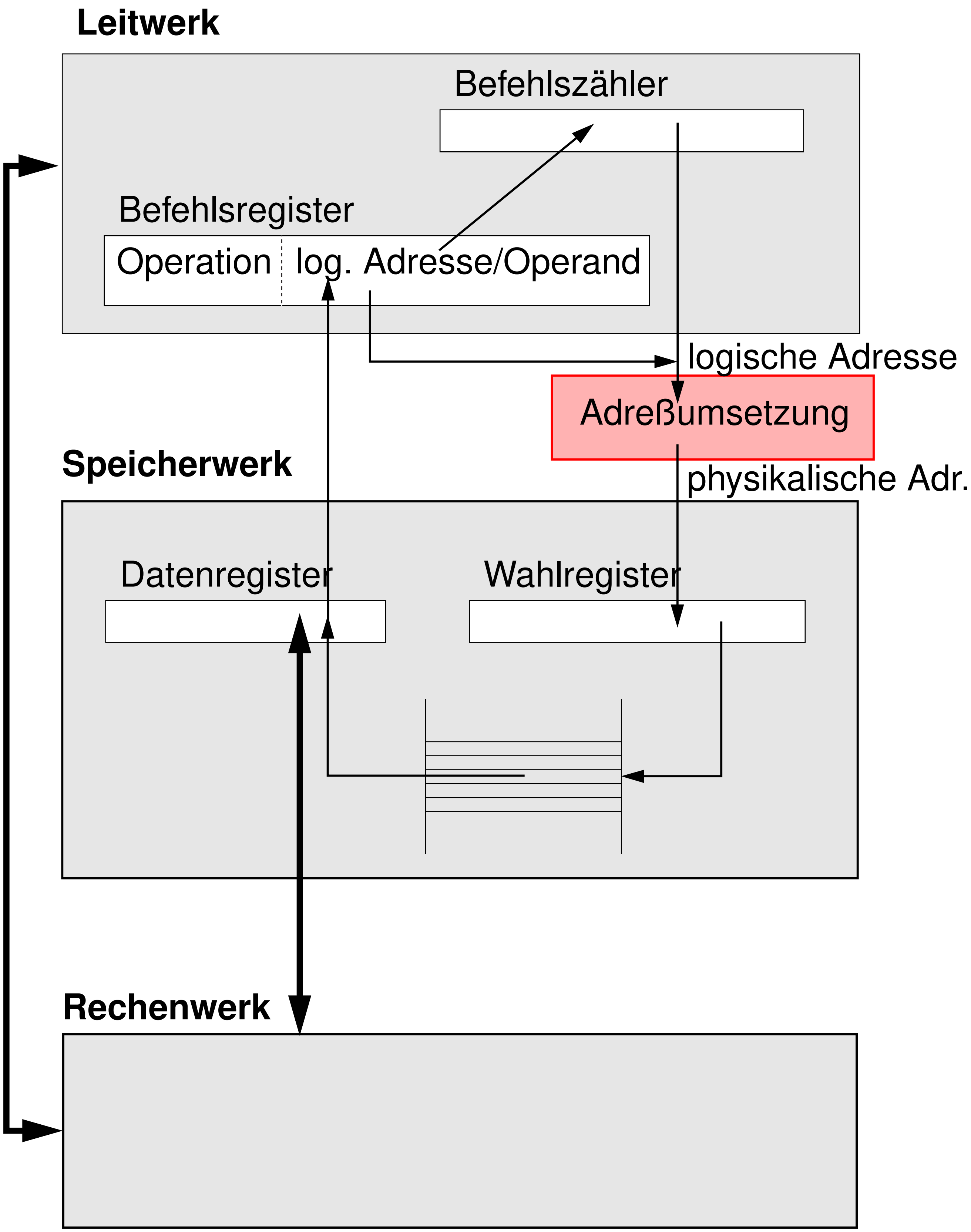
◆ Buddy-Systeme

0 128 K 256 K 384 K 512 K 640 K 768 K 896 K 1 M

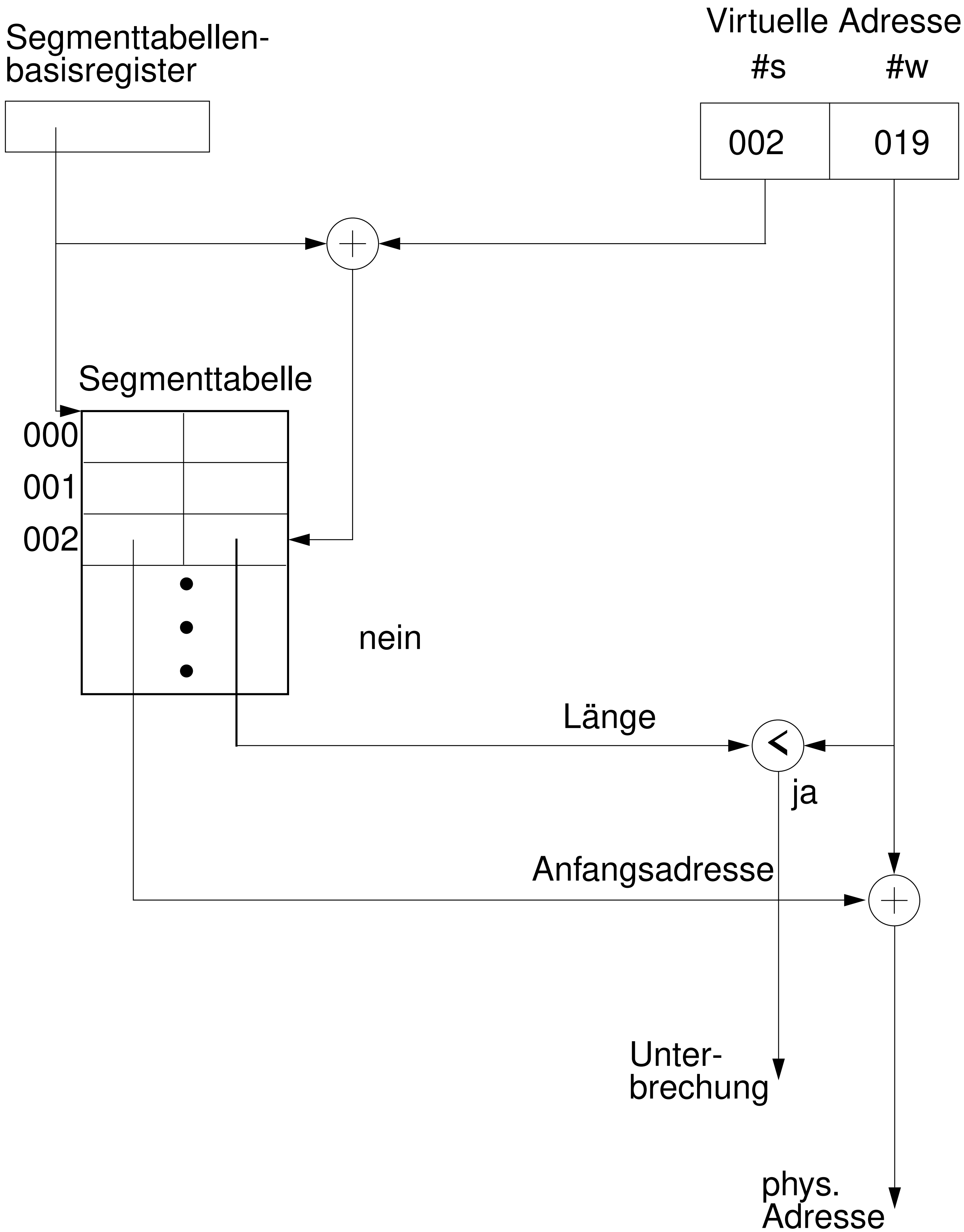
<table>
<tr><td>A 70</td><td>A</td><td colspan="2">128</td><td colspan="2">256</td><td>512</td></tr>
<tr><td>A 35</td><td>A</td><td>B</td><td>64</td><td colspan="2">256</td><td>512</td></tr>
<tr><td>A 80</td><td>A</td><td>B</td><td>64</td><td>C</td><td>128</td><td>512</td></tr>
<tr><td>F A</td><td>128</td><td>B</td><td>64</td><td>C</td><td>128</td><td>512</td></tr>
<tr><td>A 60</td><td>128</td><td>B</td><td>D</td><td>C</td><td>128</td><td>512</td></tr>
<tr><td>F B</td><td>128</td><td>64</td><td>D</td><td>C</td><td>128</td><td>512</td></tr>
<tr><td>F D</td><td colspan="3">256</td><td>C</td><td>128</td><td>512</td></tr>
<tr><td>F C</td><td colspan="6">1024</td></tr>
</table>

4.3 Segmentierung

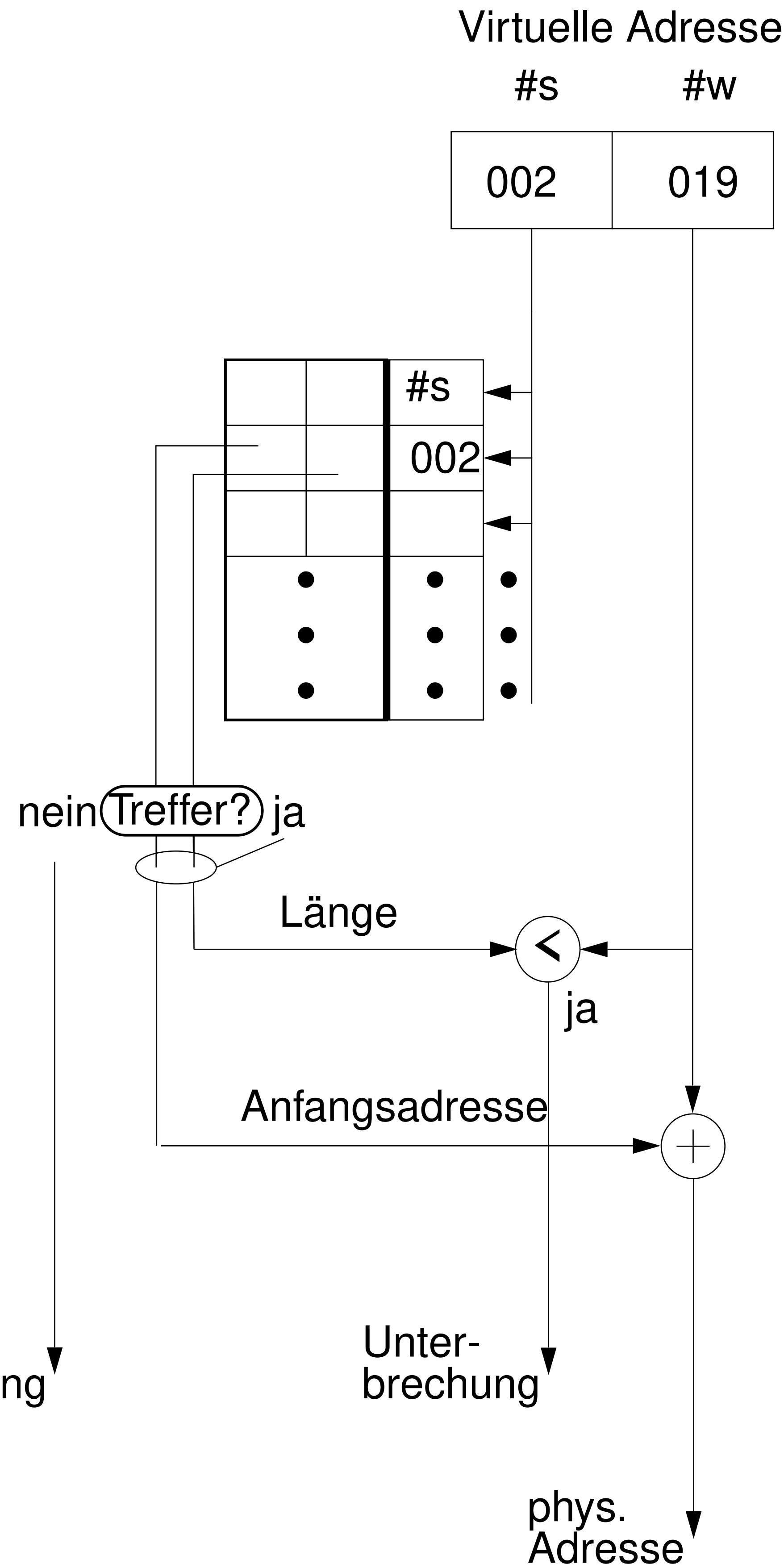
- Das Prinzip



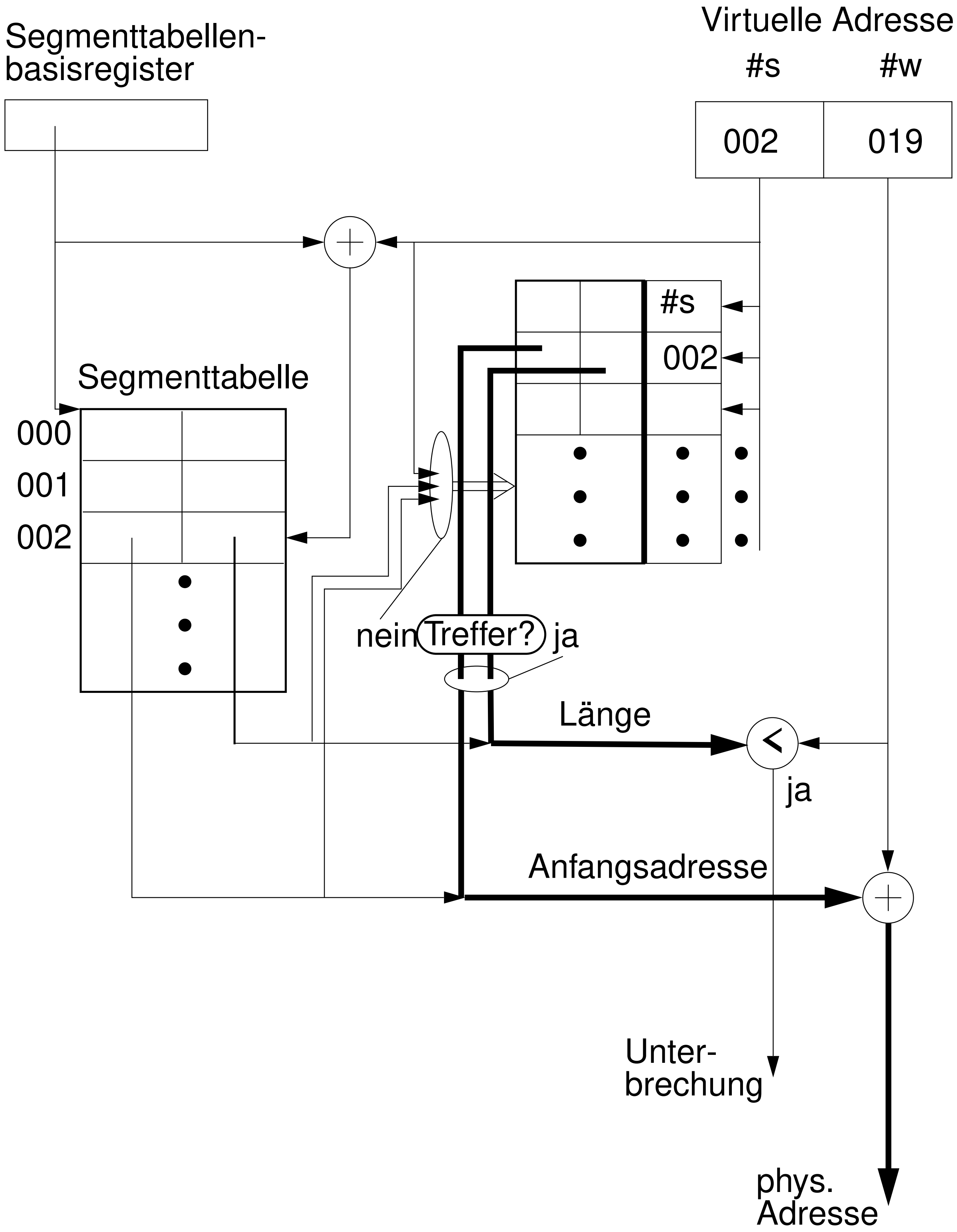
◆ Segmentierung mit Übersetzungstabelle



- Segmentierung mit Assoziativregistern

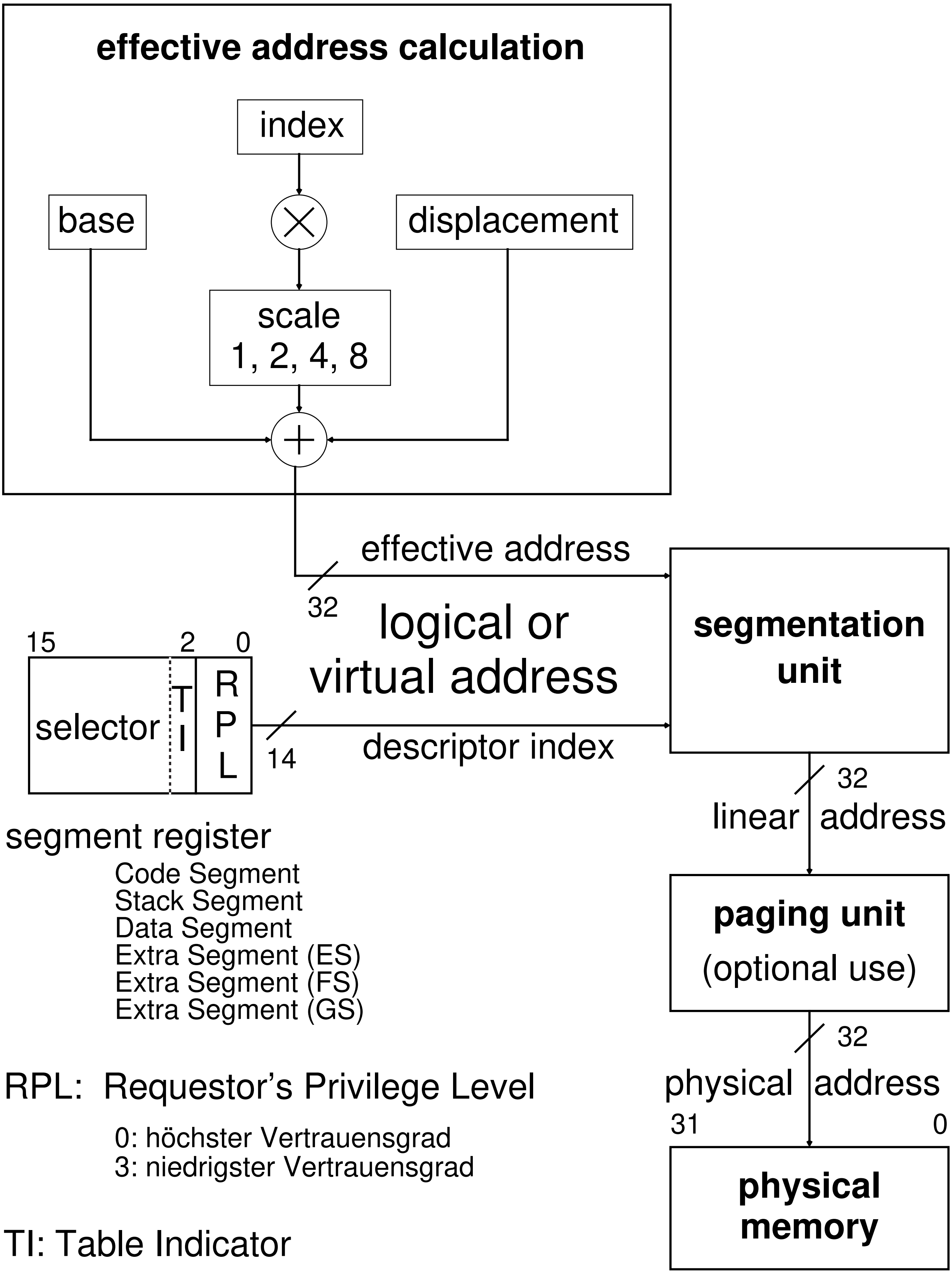


- Mischung beider Methoden



sp1/96/folien/3: 18.12.1996

- Beispiel: Intel 80486



segment register

Code Segment
Stack Segment
Data Segment
Extra Segment (ES)
Extra Segment (FS)
Extra Segment (GS)

RPL: Requestor's Privilege Level

0: höchster Vertrauensgrad
3: niedrigster Vertrauensgrad

TI: Table Indicator

0: Global Descriptor Table
1: Local Descriptor Table

- Vorteile der Segmentierung
  - Segmentierung ermöglicht in einfacher Weise Kapselung und dynamische Verlagerbarkeit
  - Flexiblere Wiedereinlagerung bei dynamischer Partitionierung
  - Ermöglicht Kompaktifizierung
  - Ermöglicht in einfacher Weise gemeinsame Segmente
    - Gemeinsame Befehlssegmente

      Platzsparende Realisierung von Instanzen einer Klasse

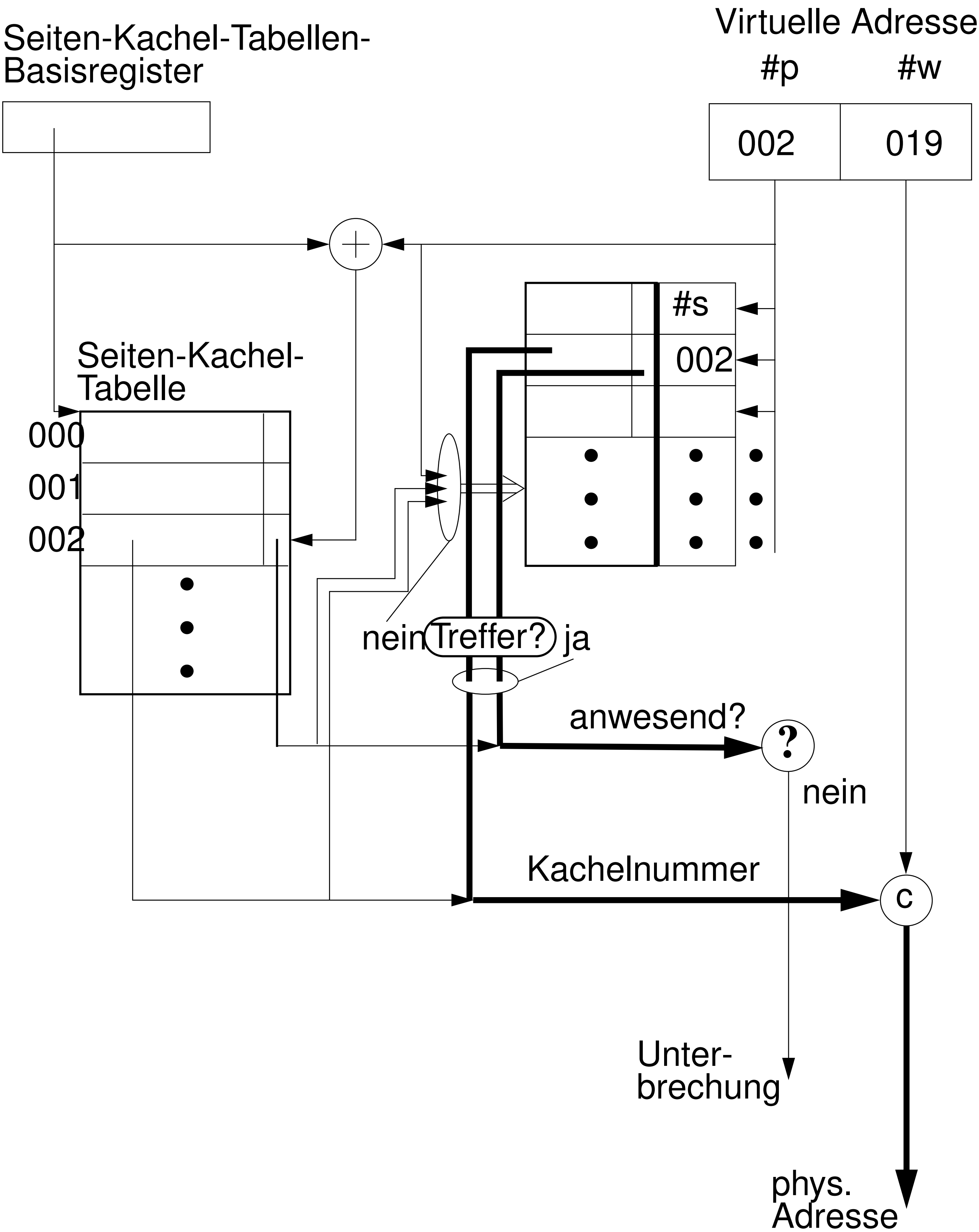
      Gemeinsame Bibliotheken
    - Gemeinsame Daten

      Effiziente Kooperation
  - In einfacher Weise erweiterbar um Basismechanismen für Zugriffsschutz
    - Übliche Zugriffsattribute
      - Lesender Zugriff erlaubt
      - Lesender und modifizierender Zugriff erlaubt
      - Ausführender Zugriff erlaubt
      - Im Anwendungsmodus zugreifbar

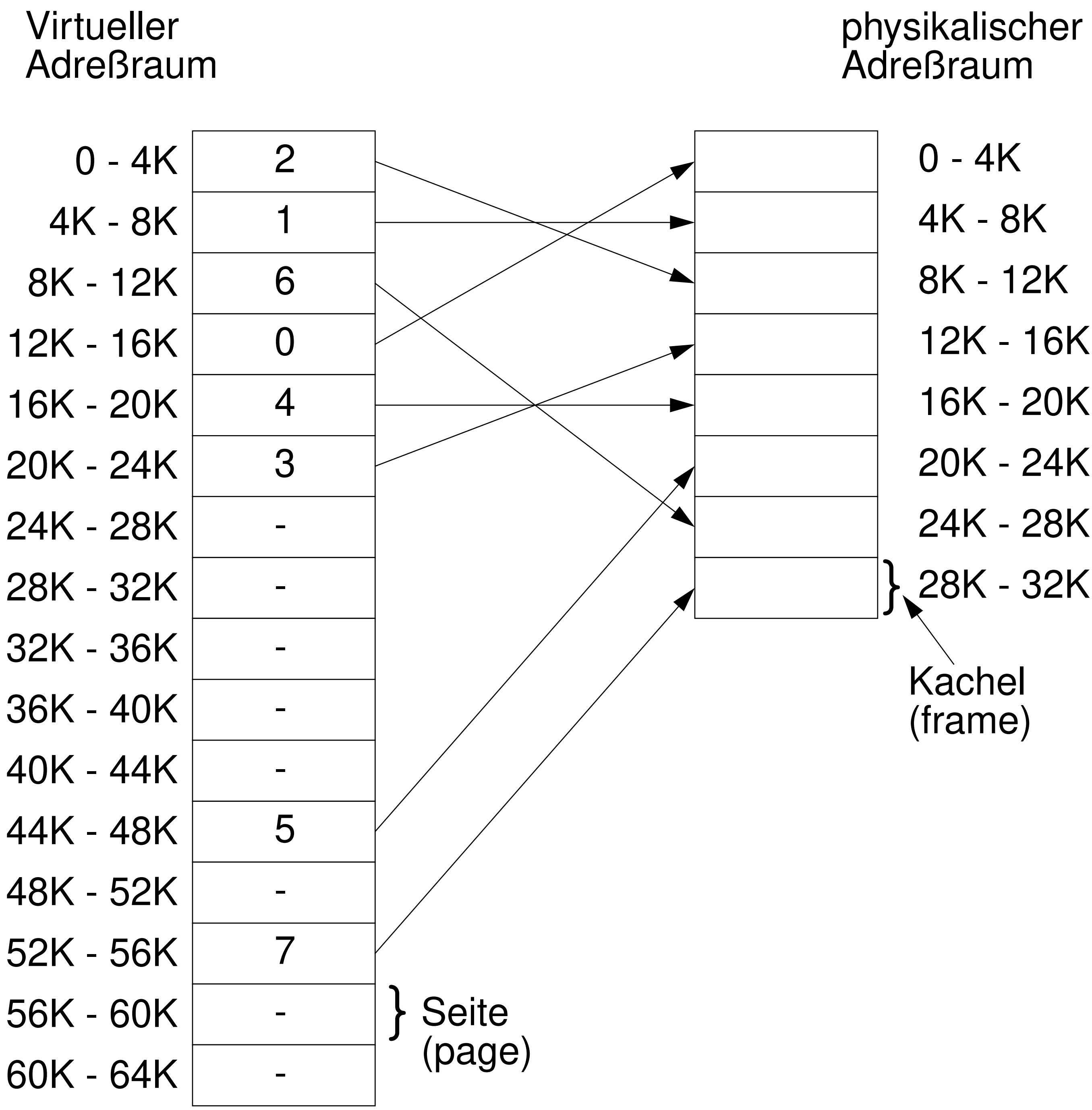
sp1/96/folien/3: 18.12.1996

4.4 Seitenadressierung, gestreute Speicherung (paging)

- Mechanismus

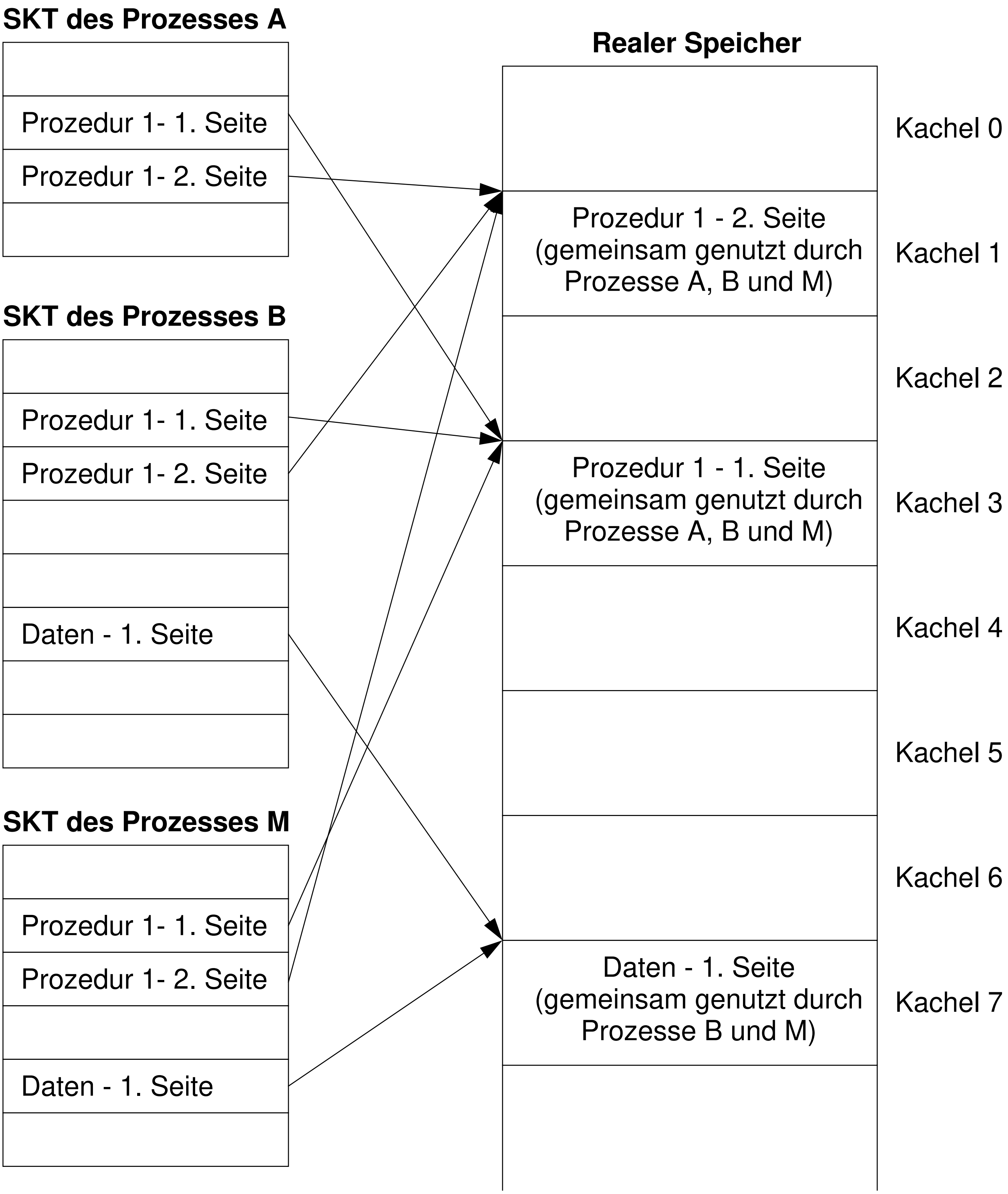


- Nutzung zur gestreuten Speicherung

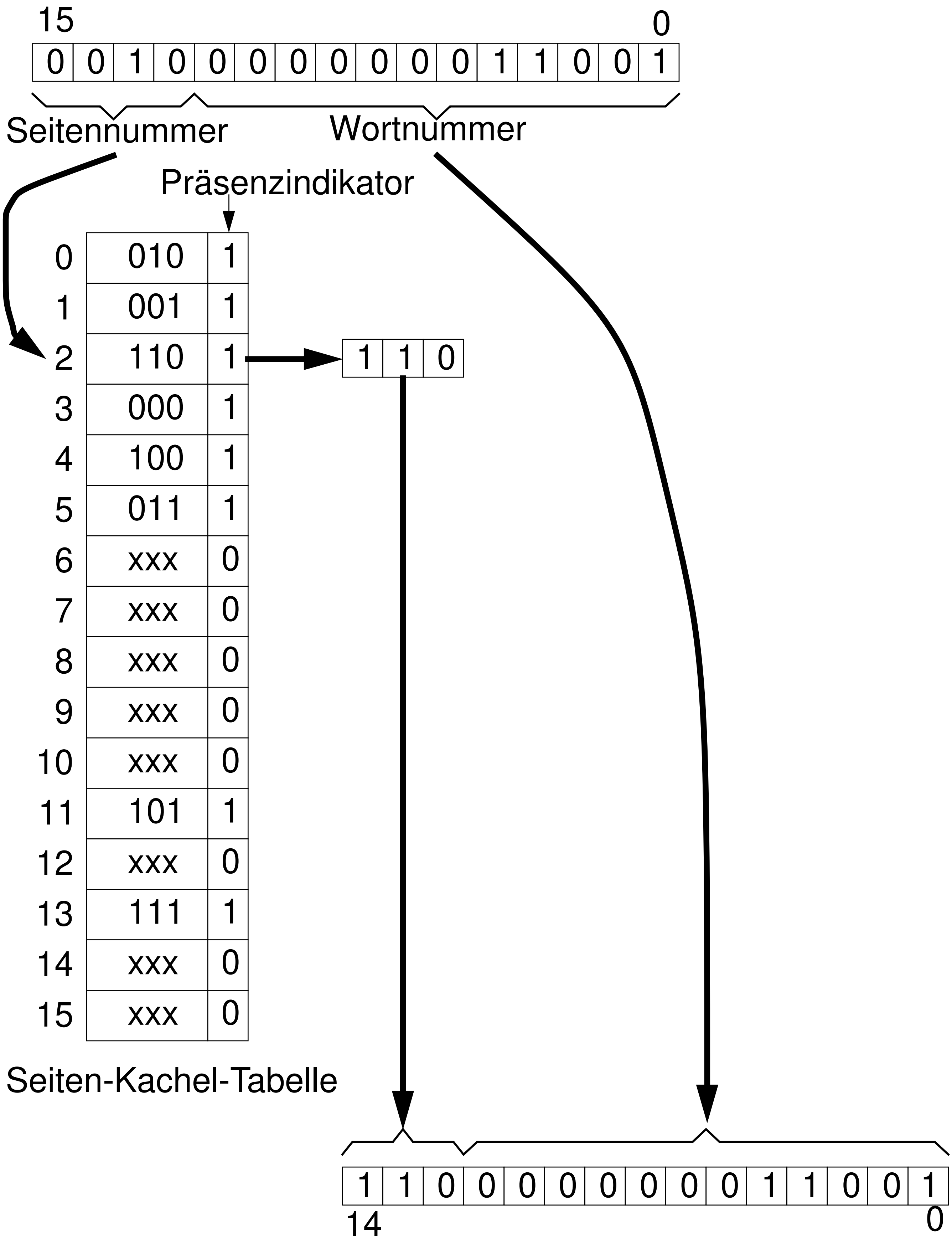


Vereinfacht das Plazierungsproblem bei Speicherbelegung

- Realisierung gemeinsamer Speicherbereiche

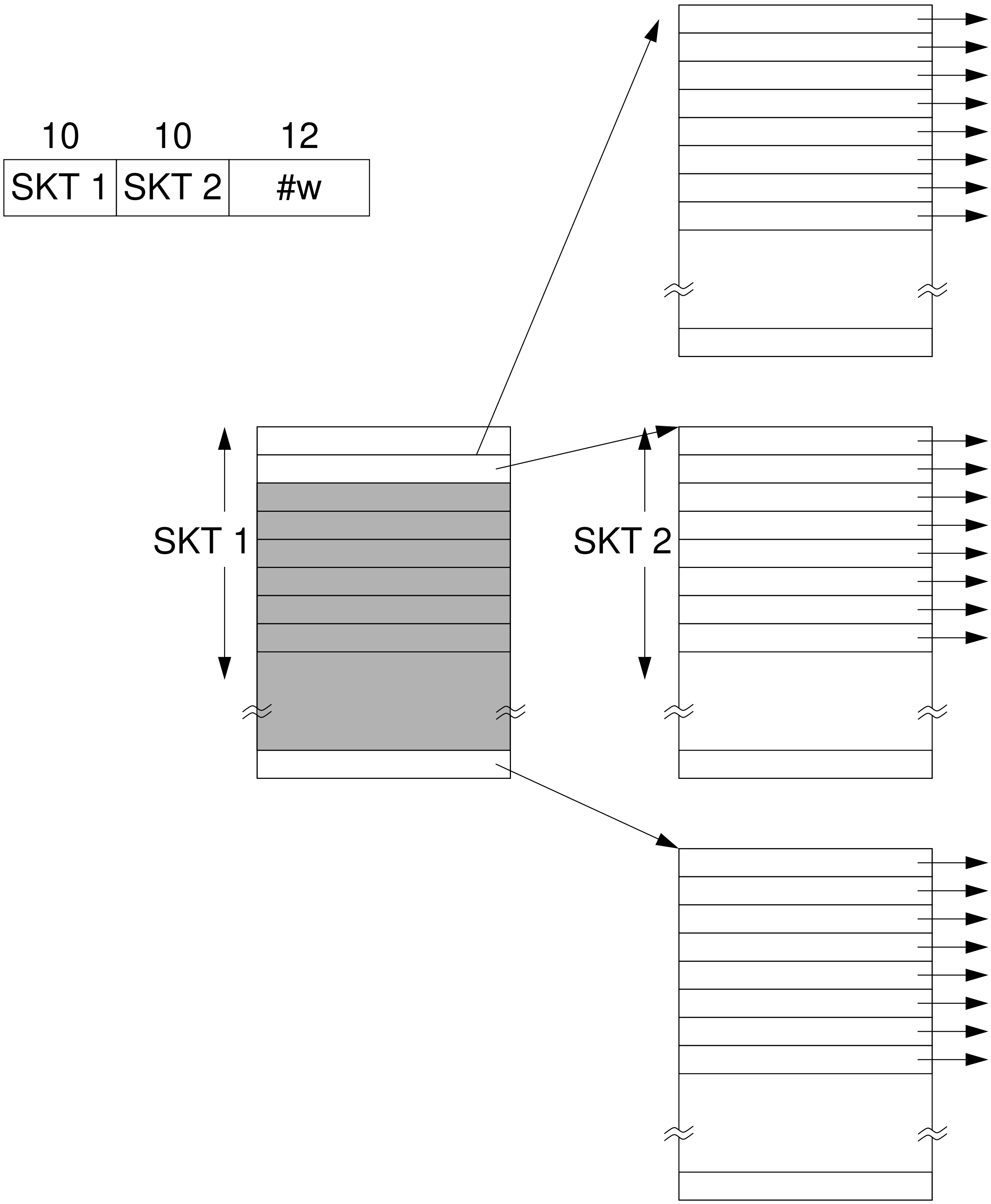


- Unvollständig abgebildete (virtuelle) Adreßräume



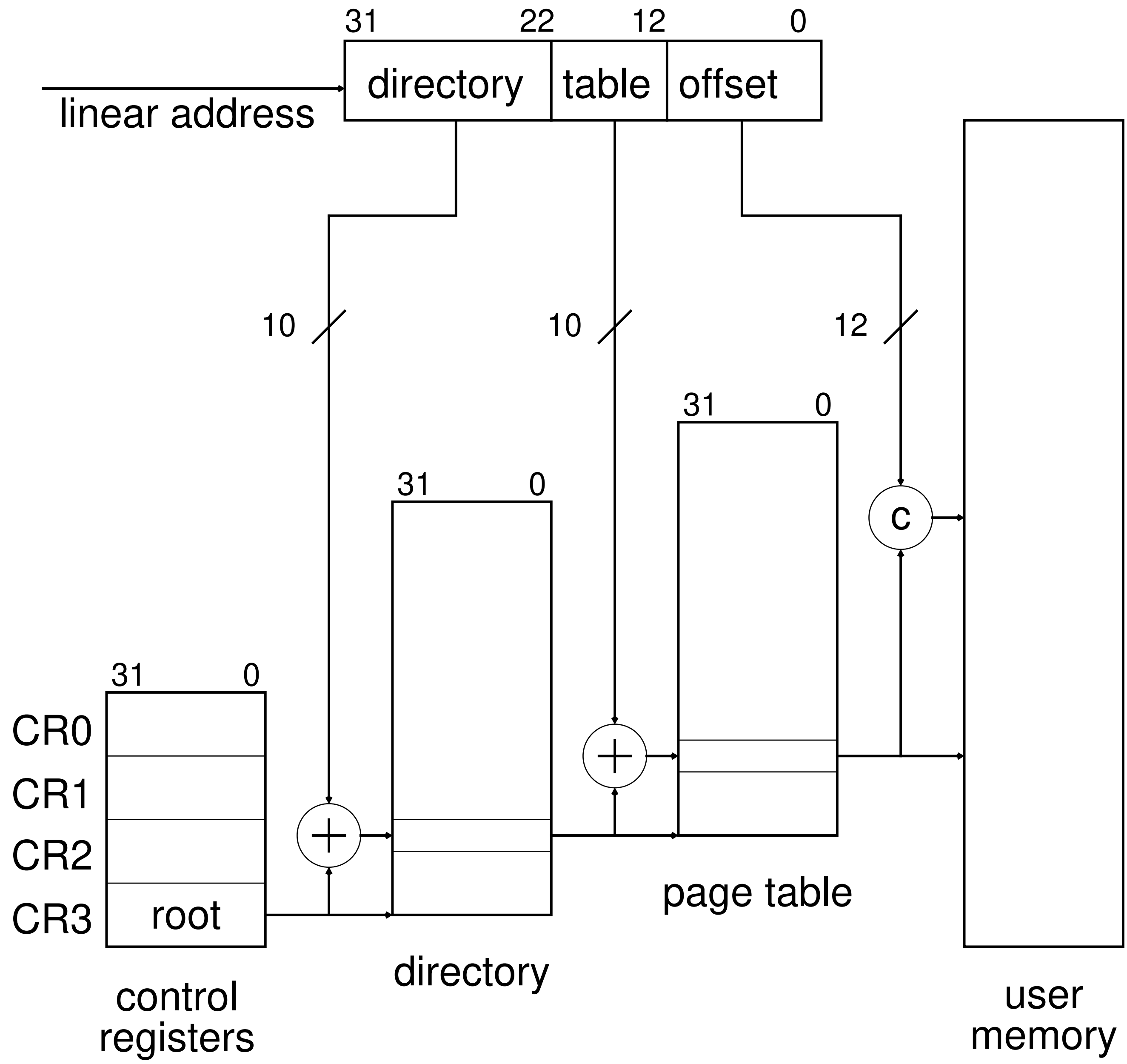
sp1/96/folien/3: 18.12.1996

- Mehrstufige Seitenadressierung



sp1/96/folien/3: 18.12.1996

- Beispiel: Intel 80486



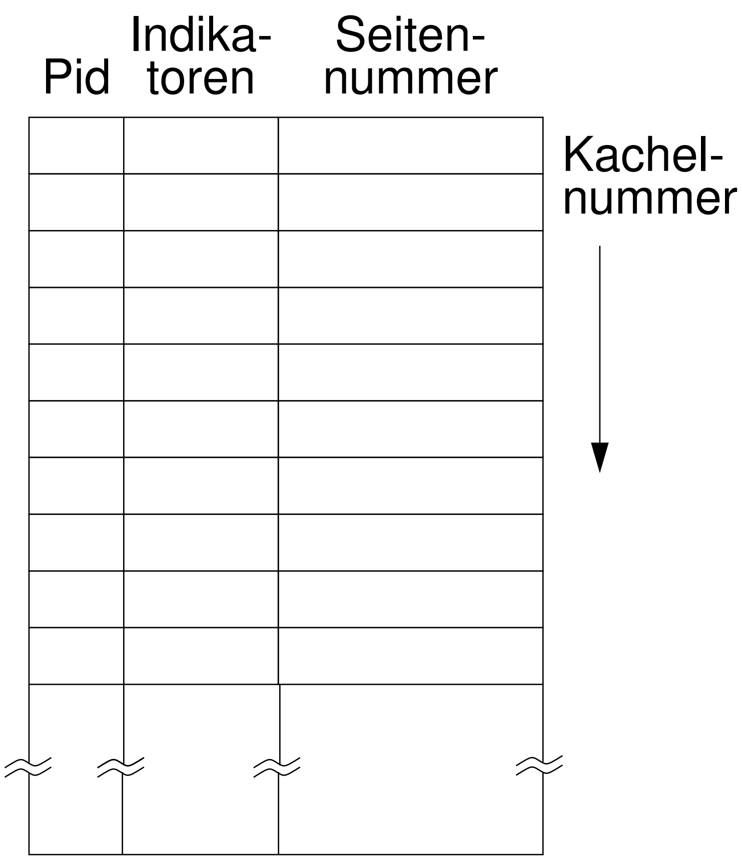
- Invertierte Seiten-Kachel-Tabellen (KST)

  z. B. MIPS 2000

Bits	20	6	6	20	1	1	1	1	8
	Seitennr.	Pid		Kachelnr.	N	D	V	G	

N: 0 Use cache, 1 no cache
D: 0 Entry is clean, 1 entry is dirty
V: 0 Invalid, 1 valid
G: 0 Check pid, 1 do not check pid

Systemweite KST



sp1/96/folien/3: 18.12.1996

Einschlägige Systemaufrufe bei SunOS (Auswahl)

shared memory operations

```
char *shmat(int shmid, char *shmaddr, int shmflg)
int shmdt(char *shmaddr)
```

shared memory control operations

```
int shmctl (int shmid, int cmd, struct shmid_ds *buf)
```

get shared memory segment identifier

```
int shmget(key_t key, int size, int shmflg)
```

- Anlegen eines gemeinsamen Segments

```
// Tilgen eines eventuell unter SHM_KEY
// existierenden gemeinsam nutzbaren Segments
shmid = shmget(SHM_KEY, 1, 0600 | IPC_CREAT);
if ((status = shmctl(shmid, IPC_RMID, 0)) == -1) {
  cout << "***** Fehler beim Tilgen des"
       << " gmeinsamen Sgementes: "
       << sys_errlist[errno] << "\n";
  exit(0);
}

// Deskriptor fuer - eventuell erst neu angelegtes -
// Segment beschaffen
if ((shmid = shmget(SHM_KEY, LAENGE_SHM,
                    0600 | IPC_CREAT)) < 0) {
  cerr << "\n\nError in shmget for shared memory: "
       << sys_errlist[errno]
       << "\n" << flush;
  exit(0);
}

// Segment in den eigenen Adressraum einblenden
if (local_shm = shmat(shmid, (void *)0, 0)) == -1) {
  cerr << "\n\nError: " << sys_errlist[errno]
       << "\n" << flush;
  exit(0);
}
```

- Initialisierung gemeinsamer Informationen

```
pv empty0(1), full0(0), empty1(1), full1(0);

shared_char buffer0[BUFFERLENGTH],
            buffer1[BUFFERLENGTH];
shared_int buffercount(BUFFERCOUNT),
           bufferlength(BUFFERLENGTH);
```

❑ Puffer füllen

```
void fuellen()
{ int i, k, l, length;
  float x = 1, y = 3;

  length = buffercount * bufferlength;

  for (i = 1; i <= length; i) {
    empty0.P();
    l = 0;
    do {
      x = 1;
      for (k = 0; k < load; k++)
        x = x / y; // Rechenlast erzeugen
      buffer0[l++] = i++ & 255;
    } while (l < bufferlength);
    full0.V();

    empty1.P();
    l = 0;
    do {
      x = 1;
      for (k = 0; k < load; k++)
        x = x / y; // Rechenlast erzeugen
      buffer1[l++] = i++ & 255;
    } while (l < bufferlength);
    full1.V();
  }
}
```

- ❑ Puffer in Datei schreiben

```
void sichern()
{ int i, file, result;
  float x = 0, y = 3;

  file = open("hilfsdatei",
              O_WRONLY | O_CREAT | O_TRUNC | O_SYNC);

  for (i = 0; i < buffercount; i +=2) {
    full0.P();
    result = write(file, buffer0, bufferlength);
    empty0.V();

    full1.P();
    result = write(file, buffer1, bufferlength);
    empty1.V();
  }
}
```