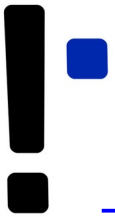


Betriebssysteme 1

SS 2020

Prof. Dr.-Ing. Hans-Georg Eßer
Fachhochschule Südwestfalen

Foliensatz F:

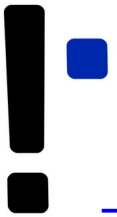
v2.0, 2020/06/18

- Speicherverwaltung, Paging
- Speichernutzung unter Linux

Classics: Overlay-Programmierung

Turbo Pascal, um 1985-90:

```
program grossesprojekt;

overlay procedure kundendaten;
...

overlay procedure lagerbestand;
...

{ Hauptprogramm }
begin
  while input <> "exit" do begin
    case input of
      "kunden": kundendaten;
      "lager":  lagerbestand;
    end;
  end;
end.
```



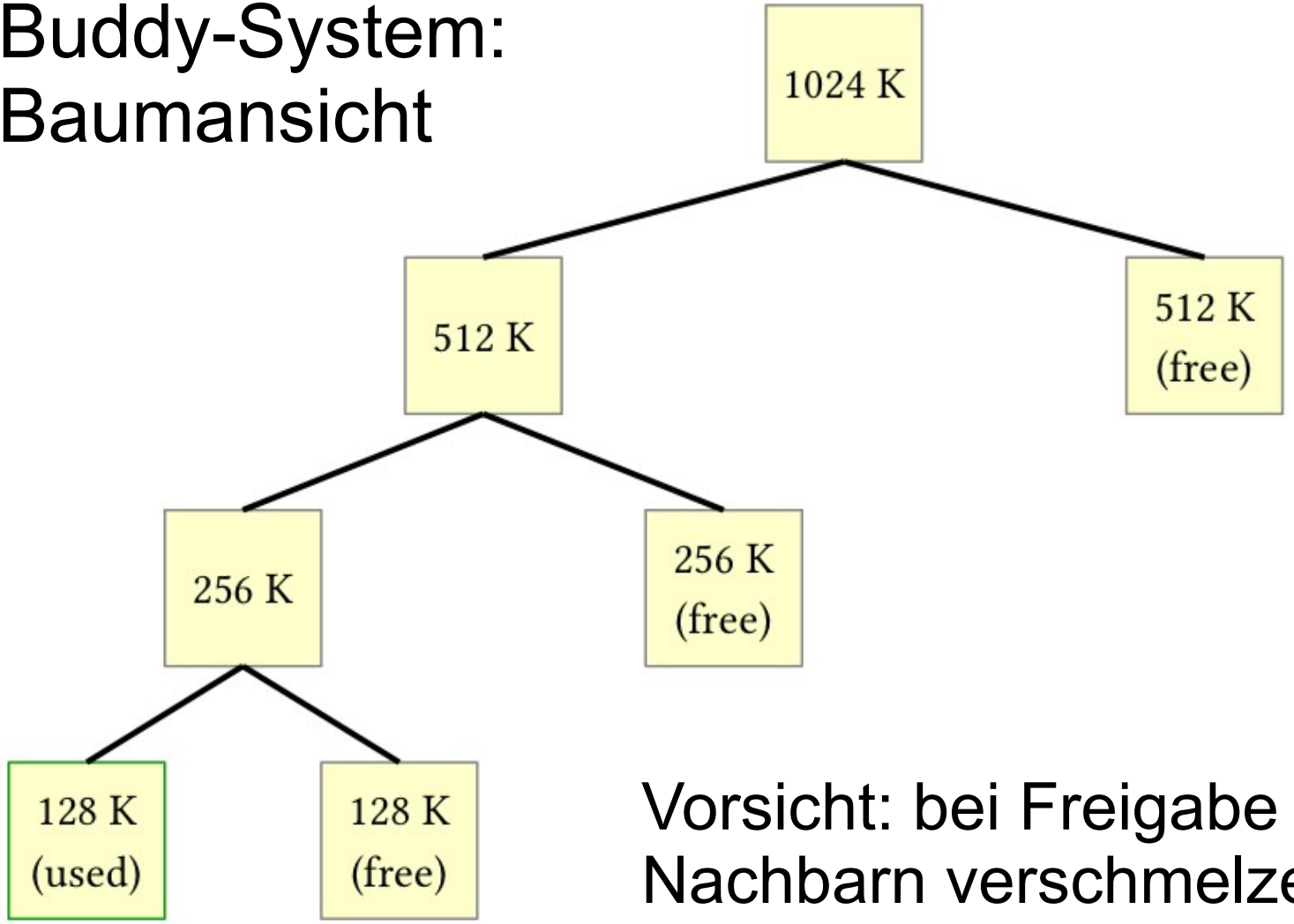
Buddy-System (1)

- Buddy-System (dynamische Zuteilung)
  - Speichergröße ist 2^n (für ein n)
  - Bei Anforderung schrittweise freien Speicherbereich halbieren, bis gerade noch passender Bereich verfügbar ist
  - Bei Rückgabe von Speicher diesen ggf. mit freiem Nachbarn verschmelzen (→ Rekursion?)
  - Beispiel: 1 MByte, alles frei, Anforderung 90 KByte

<table>
<tr><td colspan="4">1024 KB</td></tr>
<tr><td colspan="2">512 KB</td><td colspan="2">512 KB</td></tr>
<tr><td colspan="2">256 KB</td><td>256 KB</td><td>512 KB</td></tr>
<tr><td>128 KB</td><td>128 KB</td><td>256 KB</td><td>512 KB</td></tr>
</table>

Buddy-System (2)

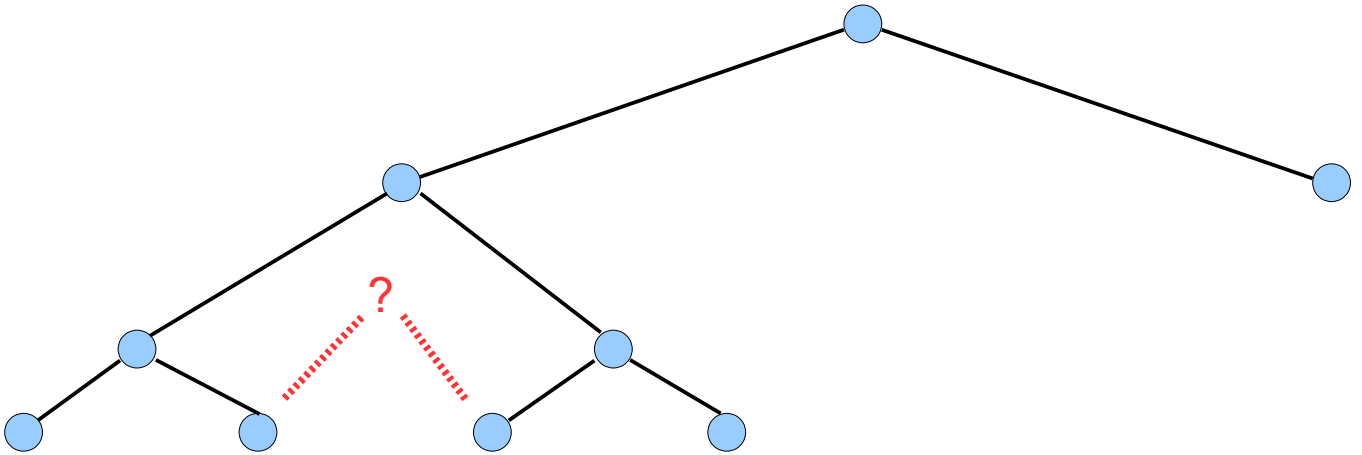
- Buddy-System: Baumansicht



Vorsicht: bei Freigabe nur **direkte** Nachbarn verschmelzen!

Buddy-System (3)

- Buddy-System: unmögliche Verschmelzung

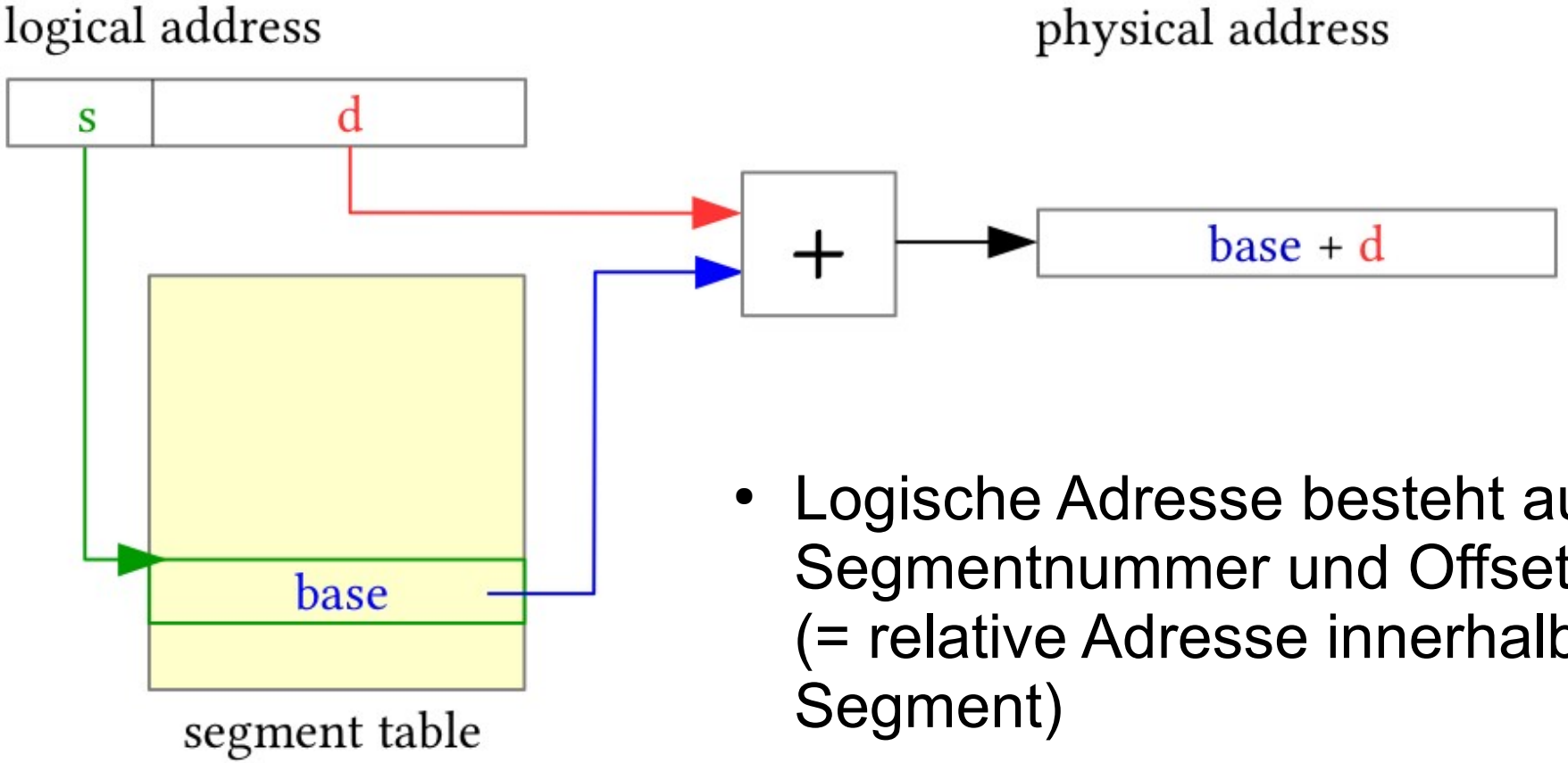


128 KB	128 KB	128 KB	128 KB	512 KB
128 KB	256 KB		128 KB	512 KB

!

Segmentierung (1)

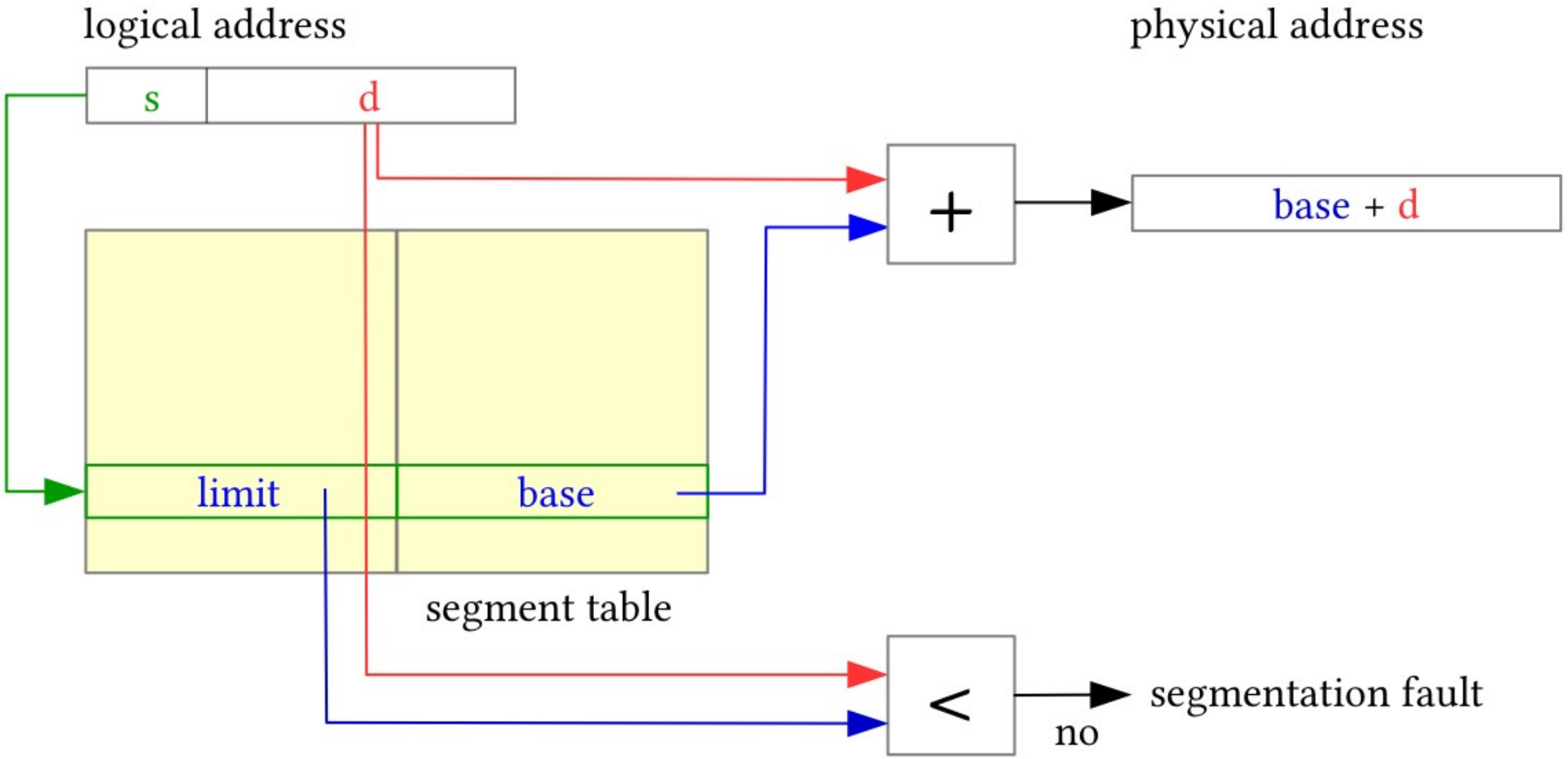
- Über eine Segment-Tabelle wird Speicher in Segmente (zusammenhängende Bereiche) aufgeteilt



- Logische Adresse besteht aus Segmentnummer und Offset (= relative Adresse innerhalb Segment)

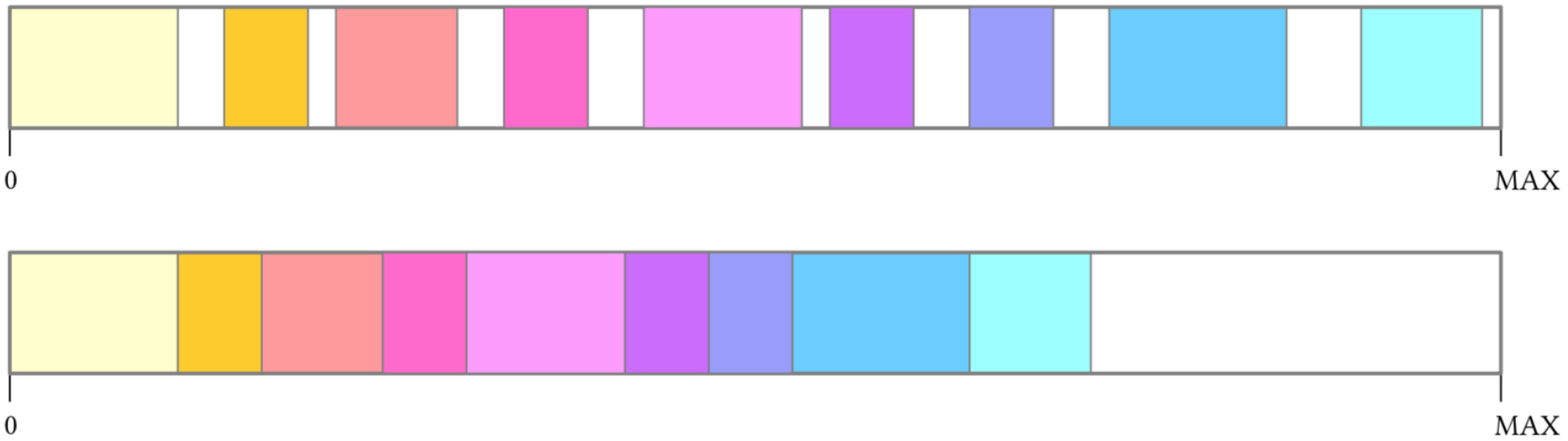
Segmentierung (2)

- Angabe einer Segmentgröße → Prüfung bei Zugriff



Segmentierung (3)

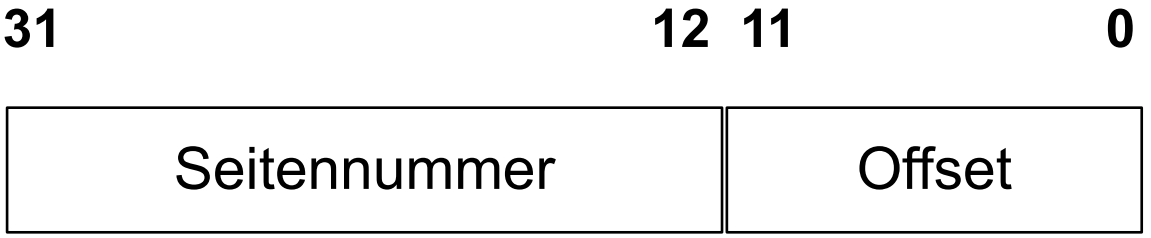
- Problem bei dynamischem Erzeugen und Löschen von Segmenten: Lücken
- Regelmäßig aufräumen („Kompaktierung“)



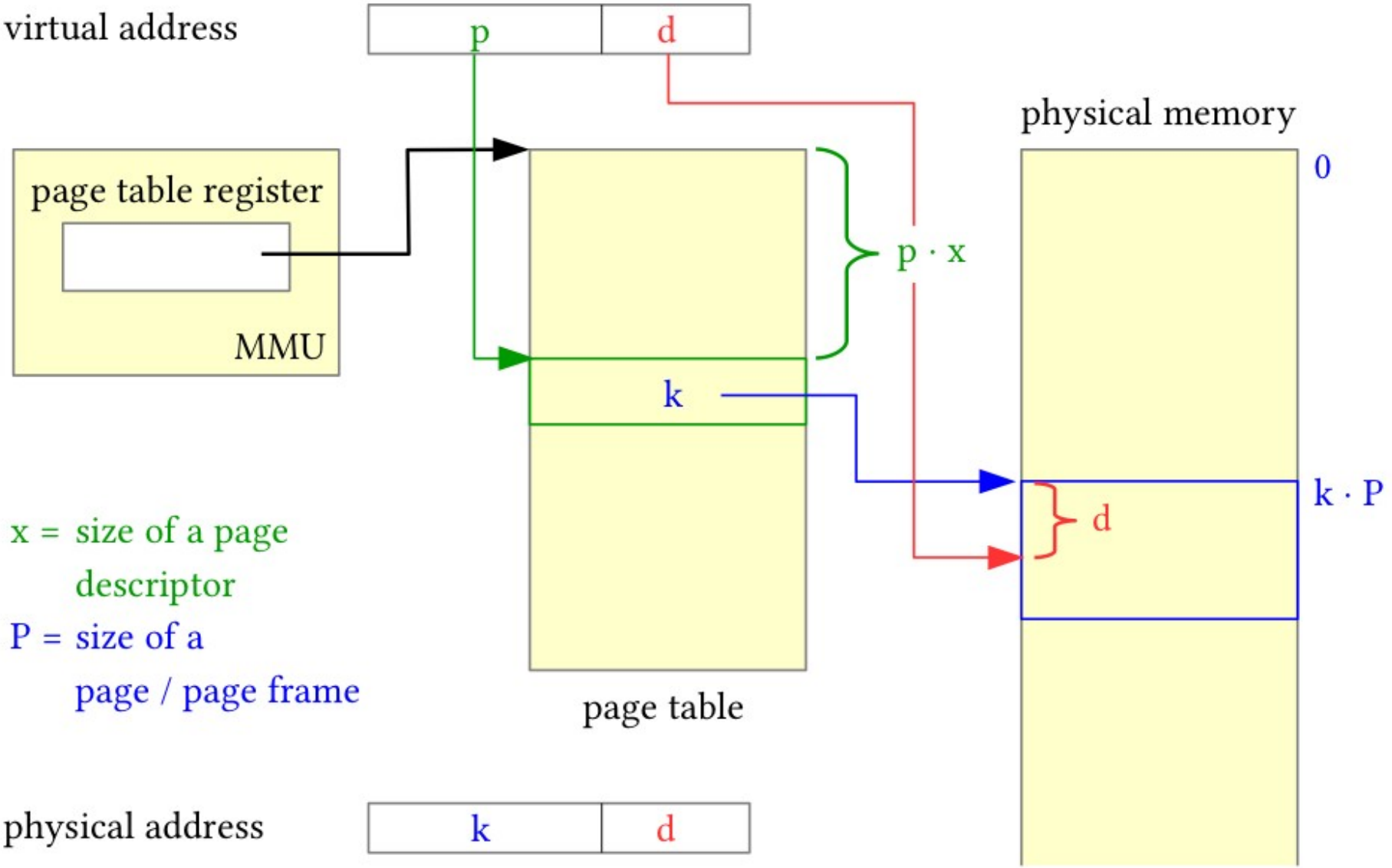
Adressübersetzung beim Paging (1)

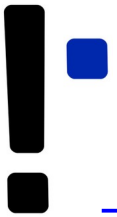
- Die Programmadresse wird in zwei Teile aufgeteilt:
  - eine Seitennummer
  - eine relative Adresse (offset) in der Seite

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



Adressübersetzung beim Paging (2)





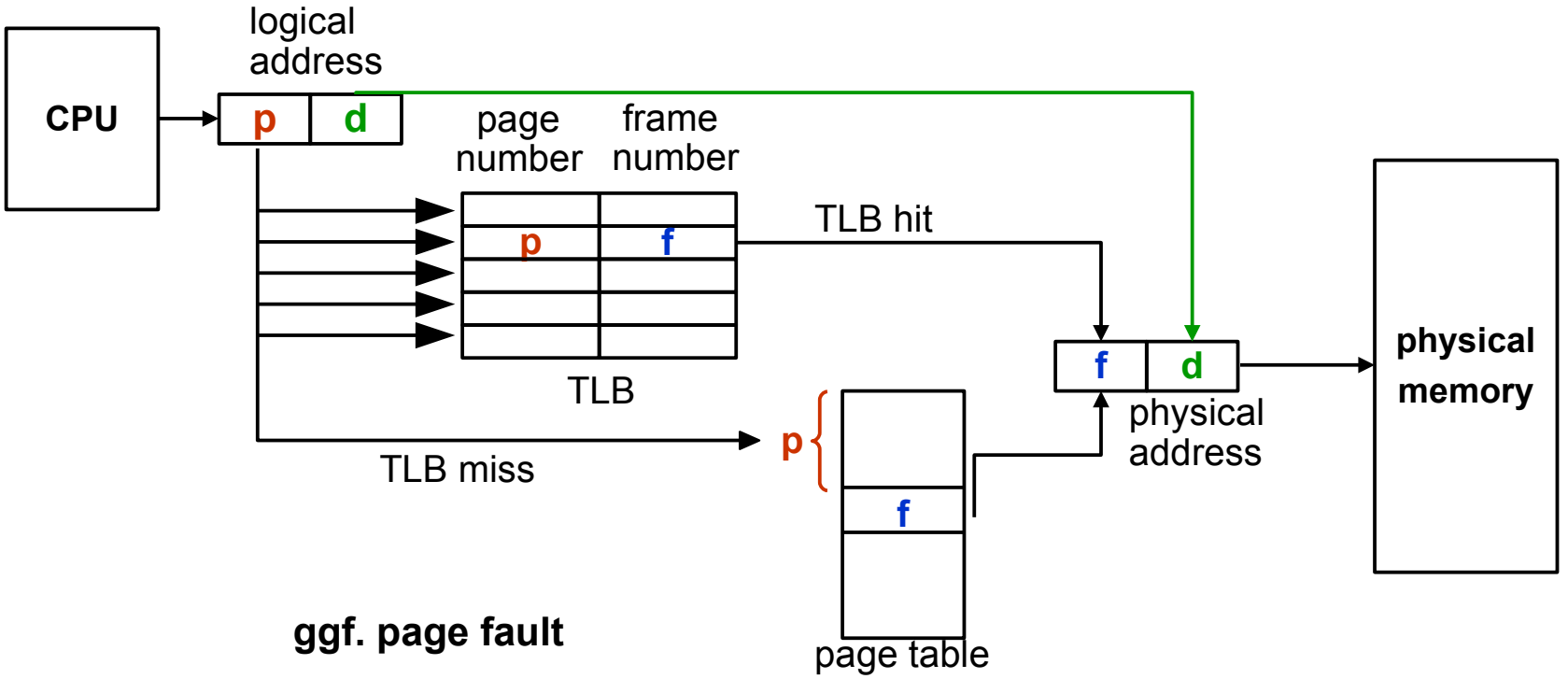
Thrashing / Lokalität / TLB (1)

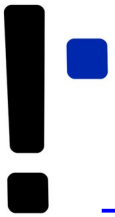
- **Thrashing:** Prozessor verbringt die meiste Zeit mit Ein- und Auslagern von Prozessteilen statt mit der Ausführung von Prozessanweisungen
- **Lokalitätsprinzip:**
  - Zugriffe auf Daten und Programmcode häufig lokal gruppiert;

    → Annahme gerechtfertigt, dass nur wenige Prozessstücke während einer kurzen zeitlichen Periode gleichzeitig vorgehalten werden müssen

Thrashing / Lokalität / TLB (2)

- **Translation Look-Aside Buffer** (**TLB**): schneller Hardware-Cache für zuletzt benutzte Seitentabelleneinträge
- Assoziativ-Speicher: bei Übersetzung einer Adresse wird deren Seitennummer gleichzeitig mit allen Einträgen des TLB verglichen.





Thrashing / Lokalität / TLB (3)

- Treffer im TLB → Speicherzugriff auf Seitentabelle unnötig
- Fehltreffer → Zugriff auf die Seitentabelle; alten Eintrag im TLB durch neuen ersetzen
- Trefferquote (hit ratio) beeinflusst die durchschnittliche Zeit einer Adressübersetzung.
- Lokalitätsprinzip: Programme greifen meist auf benachbarte Adressen zu → auch bei kleinen TLBs hohe Trefferquoten (typisch: 80-98%).

Thrashing / Lokalität / TLB (4)

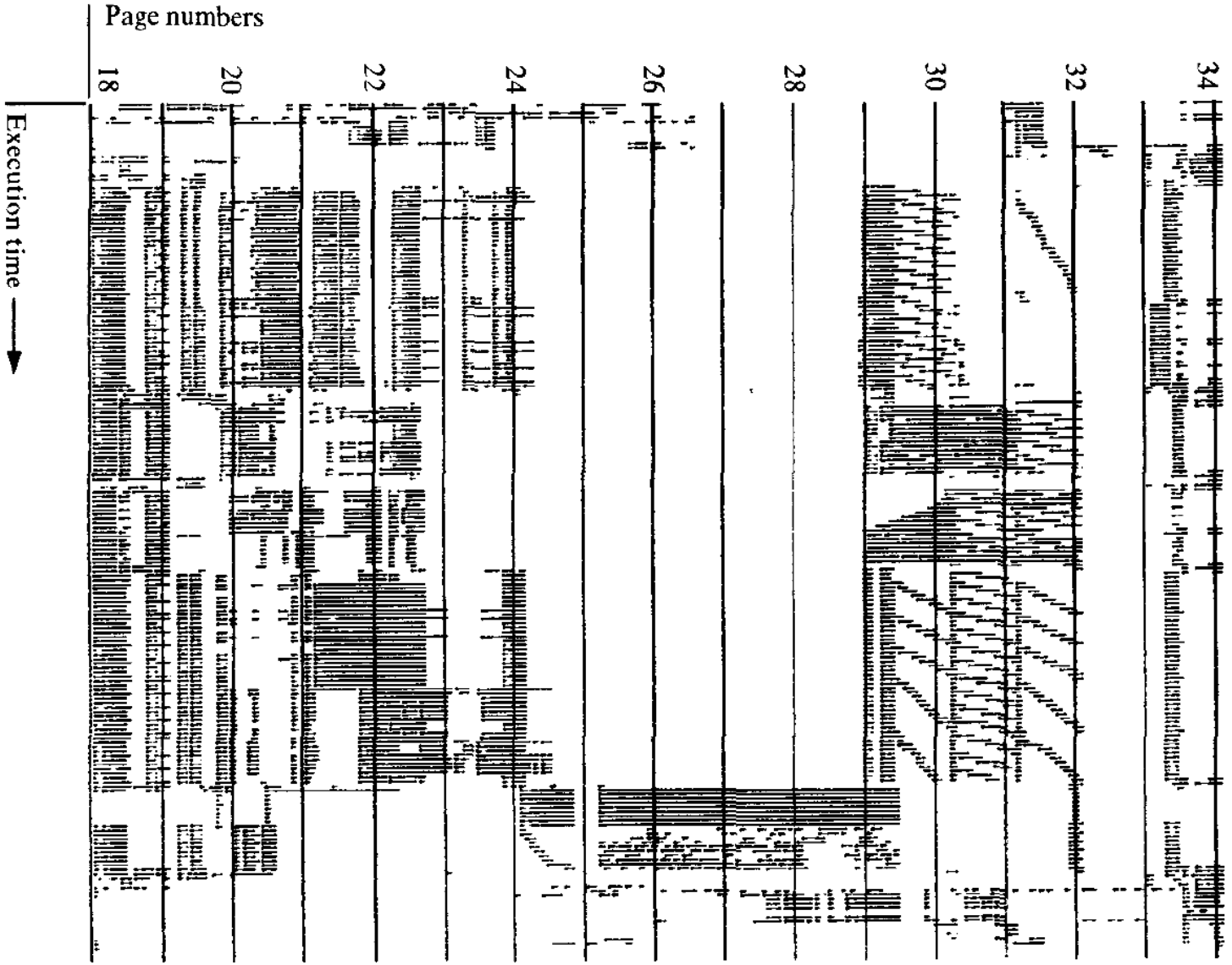
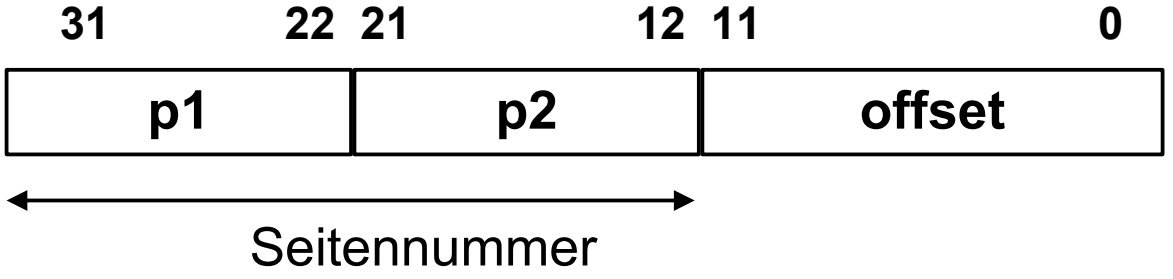


Bild: Hatfield (1972)

Mehrstufiges Paging (1)

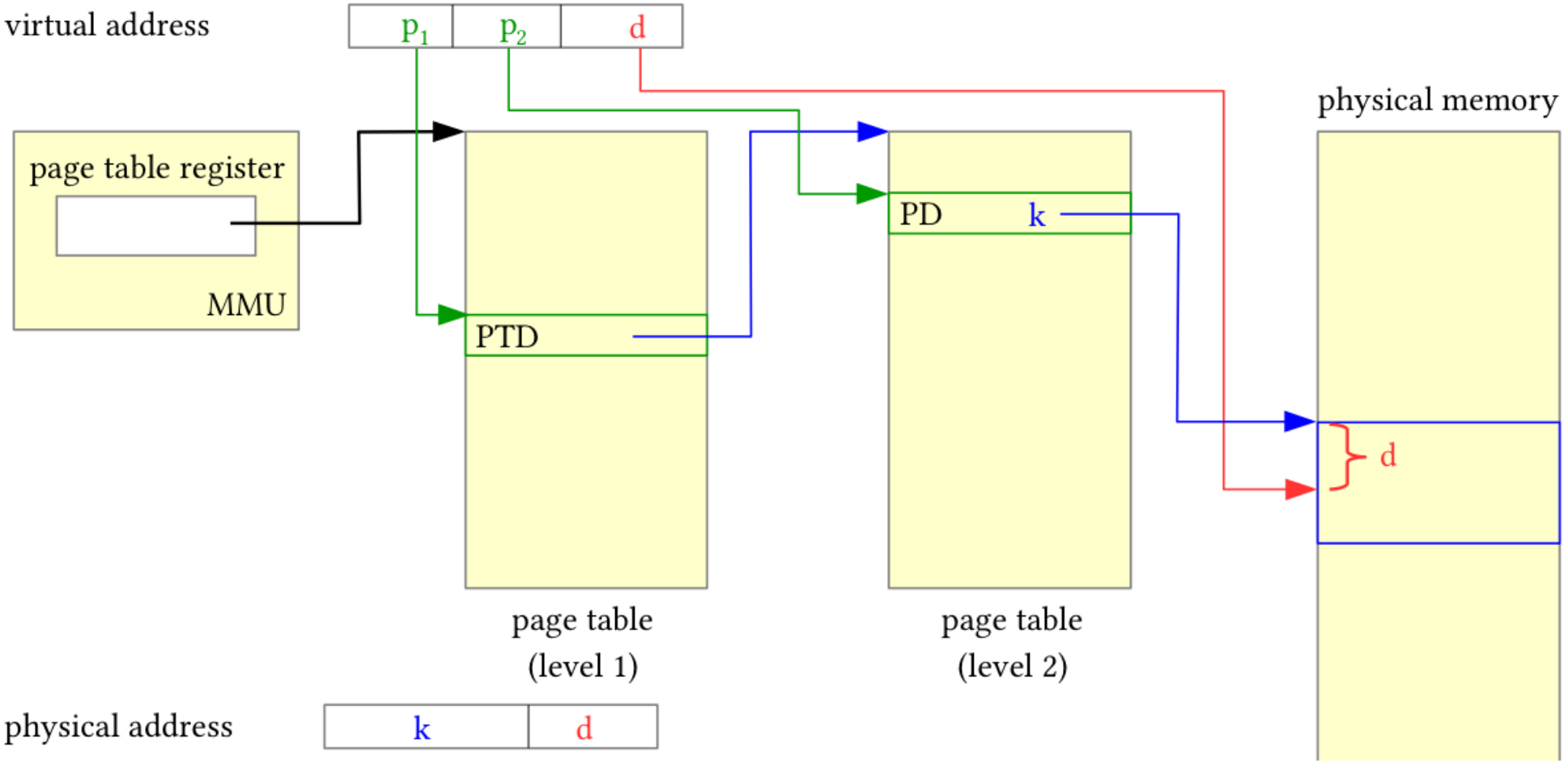
- Zweistufiges Paging:
  - Seitennummer noch einmal unterteilen, z. B.:



  - p_1: Index in **äußere Seitentabelle**, deren Einträge jeweils auf eine **innere Seitentabelle** zeigen
  - p_2: Index in eine der inneren Seitentabellen, deren Einträge auf Seitenrahmen im Speicher zeigen
  - Die inneren Seitentabellen müssen nicht alle speicherresident sein
- Analog dreistufiges Paging etc. implementieren

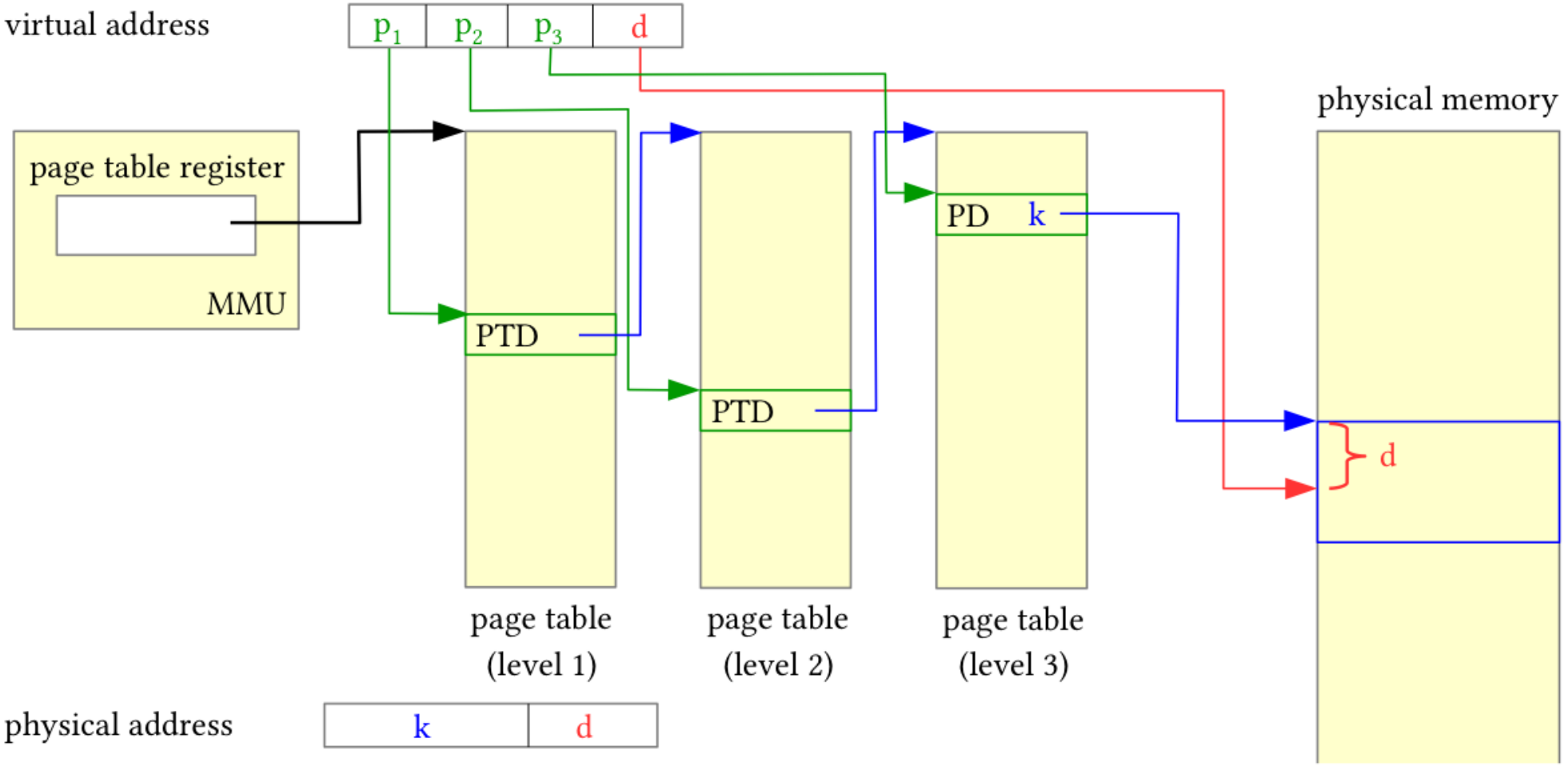
Mehrstufiges Paging (2)

Adressübersetzung bei zweistufigem Paging:



Mehrstufiges Paging (3)

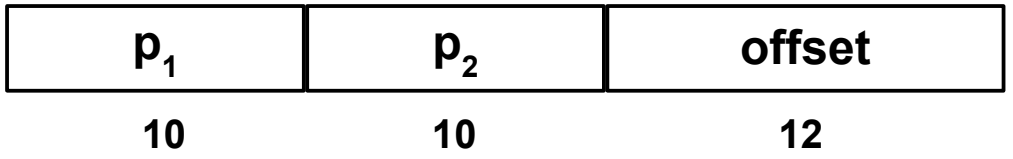
Adressübersetzung bei dreistufigem Paging:



Mehrstufiges Paging (4)

- Größe der Seitentabellen:

Beispiel:



  - Die äußere Seitentabelle hat 1024 Einträge, die auf (potentiell) 1024 innere Seitentabellen zeigen, die wiederum je 1024 Einträge enthalten.
  - Bei einer Länge von 4 Byte pro Seitentabelleneintrag ist also jede Seitentabelle genau eine 4-KByte-Seite groß.
  - Es werden nur so viele innere Seitentabellen verwendet, wie nötig.

Paging: Beispiel (1)

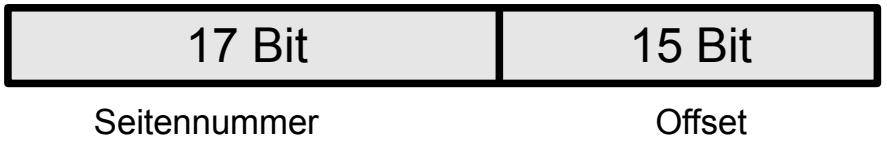
Paging mit folgenden Parametern:
- 32-Bit-Adressbus
- 32 KB Seitengröße
- 64 MB RAM
- 1-stufiges Paging

Zu berechnen:

a) maximale Anzahl der adressierbaren virtuellen Seiten

b) Größe der erforderlichen Seitentabelle (in KB)

a) 32 KB (Seitengröße) = 2^5 x 2^{10} Byte = 2^{15} Byte
d.h.: Offset ist 15 Bit lang



Also gibt es 2^{17} virtuelle Seiten

b) Zur Seitentabelle:
In 64 MB RAM passen 64 M / 32 K = 2 K = 2048 (2^{11}) Seitenrahmen
Ein Eintrag in der Seitentabelle benötigt darum 11 Bit, in der Praxis 2 Byte.

→ Platzbedarf:
#(virt. Seiten) x Größe(Eintrag)
= 2^{17} x 2 Byte = 2^{18} Byte = 256 KB

Paging: Beispiel (2)

Paging mit folgenden Parametern:
- 32-Bit-Adressbus
- 16 KB Seitengröße
- 2 GB RAM
- 3-stufiges Paging

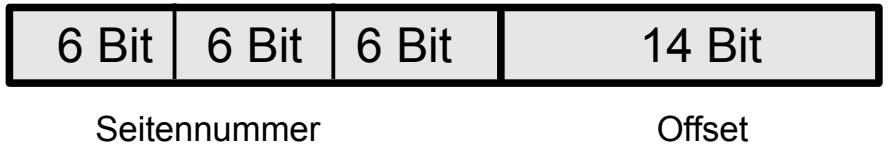
Zu berechnen:

a) maximale Anzahl der adressierbaren virtuellen Seiten

b) Größe der Seitentabelle(n)

c) Anzahl der Tabellen

a) 16 KB (Seitengröße) = 2^4 x 2^{10} Byte
= 2^{14} Byte,
d.h.: Offset ist 14 Bit lang

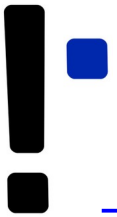


Also gibt es 2^{18} virtuelle Seiten

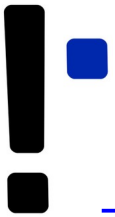
b) Zur Seitentabelle:
In 2 GB RAM passen 2 G / 16 K
= 128 K = 2^{17} Seitenrahmen
Ein Eintrag in der Seitentabelle benötigt darum 17 Bit, in der Praxis 4 Byte.

→ Platzbedarf **einer** Tabelle:
#(Einträge) x Größe(Eintrag)
= 2^6 x 4 Byte = 2^8 Byte = 256 Byte

Es gibt 1 äußere, 2^6 mittlere und 2^{12} innere Seitentabellen

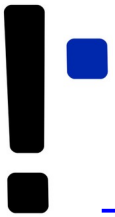


Praxis: Linux



Speicherverwaltung unter Linux

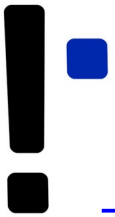
- `malloc()`, `calloc()`, `realloc()`
- `free()`
- `memset()`
- `memcpy()`, `memcmp()`
- Anonymous Memory Mapping mit `mmap()`



malloc()

- dynamisch Speicher allozieren
- verwendet Heap oder Anon-Map
- bei Nutzung des Heap: nicht initialisiert!

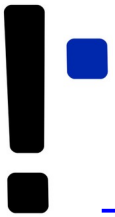
```
char *p;
p = malloc (2000)   // 2000 Bytes anfordern
if (!p) {
  // Fehler
  perror ("malloc");
}

...

free (p);
```



xmalloc()

- Test auf `malloc()`-Fehler wird oft in Wrapper `xmalloc()` integriert:

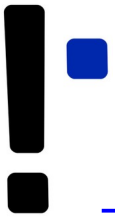
```
void *xmalloc (size_t size) {
  void *p;
  p = malloc (size);
  if (!p) {
    perror ("xmalloc");
    exit (EXIT_FAILURE);    // returns 1
  };
  return p;
};
```



calloc()

- Ähnlich `malloc()`, aber für **Arrays**
- Angabe von Anzahl und Elementgröße
- Speicher immer initialisiert (0)

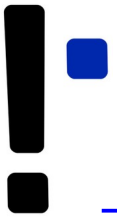
```
struct mystruct { ... };
struct mystruct *p;
p = calloc (200, sizeof(struct mystruct));  // 200 Einträge

if (!p) {
  // Fehler
  perror ("calloc");
};
```



realloc()

- ändert die Größe von Speicher, der mit `malloc()` bzw. `calloc()` angefordert wurde
- verkleinern oder vergrößern
- Vorsicht: Rückgabewert ist Pointer für neuen Speicherbereich, der jetzt an anderer Stelle anfangen kann!

```
p = malloc (10*sizeof(struct xy));
...
r = realloc (p, 20*sizeof(struct xy));
if (!r) {
  // Fehler, p noch intakt!
}
...                     // evtl. r != p

free (r);               // nicht: free (p) !
```



free()

- gibt einen dynamisch reservierten Speicherbereich wieder frei
- darf nur für Rückgabewerte von `malloc()` oder `calloc()` aufgerufen werden!
- keine Freigabe von „Teilen“ möglich (→ `realloc`)
- Nach Freigabe Speicher nicht mehr nutzen!
- Doppelter `free()`-Aufruf schlägt fehl (Prog.-Abbruch)
- `free(NULL)` geht immer, ohne Wirkung

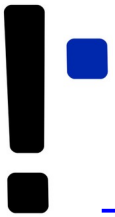
```
p = malloc(...); free (p);   // auch ok, wenn p==0
```

memset()

- Speicher, der mit `malloc()` alloziert wurde, ist (evtl.) nicht **initialisiert**
- Das kann man mit `memset()` nachholen
- benötigt `#include <string.h>`

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>  // memset

int main () {
  int SIZE = 200;
  char* p = malloc (SIZE); strcpy (p, "FH-SWF"); free (p);
  p = malloc (SIZE);
  printf ("%2x %2x %2x %2x %2x %2x %2x -- %s\n",
    p[0], p[1], p[2], p[3], p[4], p[5], p[6], p);
  memset (p, 0, SIZE);  // oder statt 0 beliebiges Füll-Byte
  printf ("%2x %2x %2x %2x %2x %2x %2x -- %s\n",
    p[0], p[1], p[2], p[3], p[4], p[5], p[6], p);
};
```

```
esser@ubu64:~$ ./memset
46 48 2d 53 57 46  0 -- FH-SWF
 0  0  0  0  0  0  0 --
```



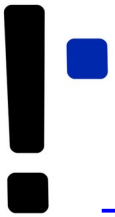
memcpy()

- kopiert einen Speicherbereich:

  `memcpy (ziel, quelle, laenge)`

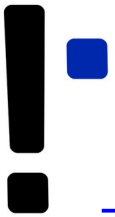
- Rückgabewert: Zeiger auf `ziel`

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>  // memcpy, memset

int main () {
  int SIZE = 200;
  char* p = malloc (SIZE); strcpy (p, "FH-SWF");
  char* q = malloc (SIZE);
  memcpy (q, p, SIZE);
  printf ("q: %2x %2x %2x %2x %2x %2x %2x -- %s\n",
    q[0], q[1], q[2], q[3], q[4], q[5], q[6], q);
};
```



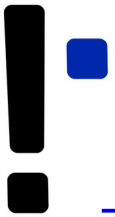
`memcpy()` vs. `strncpy()`

- Zum Unterschied
  - `memcpy (ziel, quelle, laenge)`
  - `strncpy (ziel, quelle, laenge)`
- `strncpy()` achtet auf Besonderheiten von **Strings**:
  - Ist Länge(`quelle`) < `laenge`, wird `ziel` mit Null-Bytes aufgefüllt
  - Inhalt in `quelle` nach erstem Null-Byte wird ignoriert
  - terminierendes Null-Byte bei `laenge` berücksichtigen
  - Aber: Ist `quelle` zu lang, entsteht ein *nicht*-0-terminierter String!



memcpy() vs. strncpy()

```
esser@ubu64:~$ cat strncpy.c
#include <string.h>
#include <stdio.h>

int main () {
  char quelle[] = "Vier";
  char ziel[]   = "ZZZZZZZZZ";
  strncpy (ziel, quelle, 4);   // kopiert 4 Bytes, ohne \0
  printf ("1. Versuch: %s\n", ziel);
  strncpy (ziel, quelle, 5);   // kopiert ganzen String mit \0
  printf ("2. Versuch: %s\n", ziel);
};

esser@ubu64:~$ ./strncpy
1. Versuch: VierZZZZZ
2. Versuch: Vier
```



memcmp()

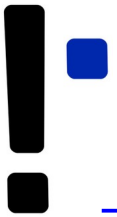
- `memcmp (a, b, len)` vergleicht Speicherbereiche

```
int main () {
  int SIZE = 200;
  char* p = malloc (SIZE); strcpy (p, "Ohm-HS");
  char* q = malloc (SIZE); char* s = malloc (SIZE);
  memcpy (q, p, SIZE);     memcpy (s, p, SIZE);
  s[0] = 'a';
  if (memcmp(p, q, SIZE) != 0) printf ("p, q verschieden\n");
  if (memcmp(p, s, SIZE) != 0) printf ("p, s verschieden\n");
};
```

- nicht (!) zum Vergleich von structs verwenden:

```
struct xy *a, *b;
memcmp (a, b, sizeof(struct xy))
```

sagt nicht unbedingt, ob a und b gleich sind

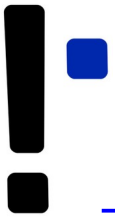


Anonymous Memory Mapping

- Alternative zur Nutzung des Heaps
- Jedes Anon-Mapping wie ein separater Heap …

```
void *p;
p = mmap (NULL,                              // do not care where
          512 * 1024,                        // 512 KB
          PROT_READ | PROT_WRITE,            // read/write
          MAP_ANONYMOUS | MAP_PRIVATE,       // anonymous, private
          -1,                                // fd (ignored)
          0);                                // offset (ignored)

if (p == MAP_FAILED)
  perror ("mmap");
else
  // 'p' points at 512 KB of anonymous memory...
```

Quelle: Robert Love, Linux System Programming



Alternative: /dev/zero mappen

```
void *p; int fd;
fd = open (”/dev/zero”, O_RDWR);   // open /dev/zero for reading/writing
if (fd < 0) { perror (”open”); return -1; }

// map [0,page size) of /dev/zero
p = mmap (NULL,                    // do not care where
          getpagesize ( ),         // map one page
          PROT_READ | PROT_WRITE,  // map read/write
          MAP_PRIVATE,             // private mapping
          fd,                      // map /dev/zero
          0);                      // no offset

if (p == MAP_FAILED) {
          perror (”mmap”);
          if (close (fd)) perror (”close”);
          return -1;
}

if (close (fd)) perror (”close”);     // close /dev/zero, no longer needed
// ’p’ points at one page of memory, use it...
```

Quelle: Robert Love, Linux System Programming