Betriebssysteme 1

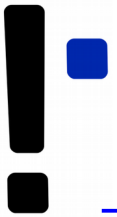
SS 2018

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

Foliensatz F:

v1.0, 2016/06/15

- Speicherverwaltung, Paging
- Speichernutzung unter Linux



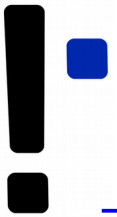
Ausgangslage

- Speicher zu knapp für große Programme
  → Overlay-Programmierung
- Programmteile dynamisch nachladen, wenn sie benötigt werden
- Programmierer muss sich um Aufteilung in Overlays kümmern

Overlay-Programmierung

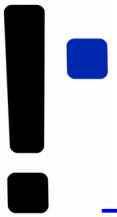
Turbo Pascal, um 1985-90:

```
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.
```



Verwaltung des Speichers durch das BS

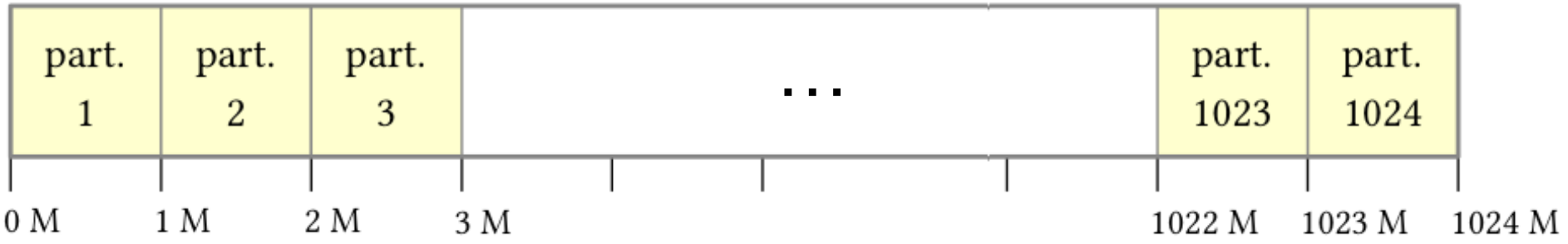
- klassische Verfahren:
  zusammenhängende Zuteilung
  - feste Partitionierung (mit gleicher Größe)
  - feste Partitionierung (mit untersch. Größe)
  - dynamische Partitionierung (z. B. Buddy-System)
  - Segmentierung
- modernes Verfahren:
  nicht-zusammenhängende Zuteilung
  - Paging (Seiten-basiert)



Klassische Verfahren (zusammenhängende Zuteilung)

Zusammenhängende Zuteilung (1)

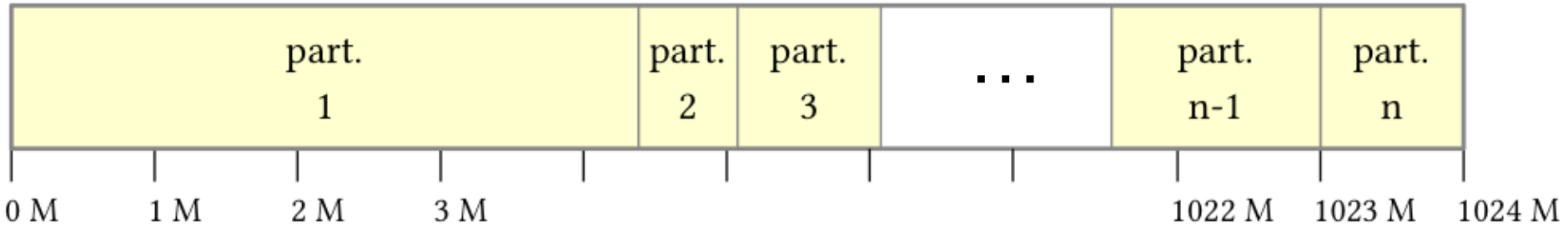
- Partitionen fester, gleicher Größe



  - unflexibel: Anzahl und Größe fix

Zusammenhängende Zuteilung (2)

- Partitionen fester, verschiedener Größe



  - nur minimal flexibler
  - Aufteilung wird bei System-Initialisierung festgelegt

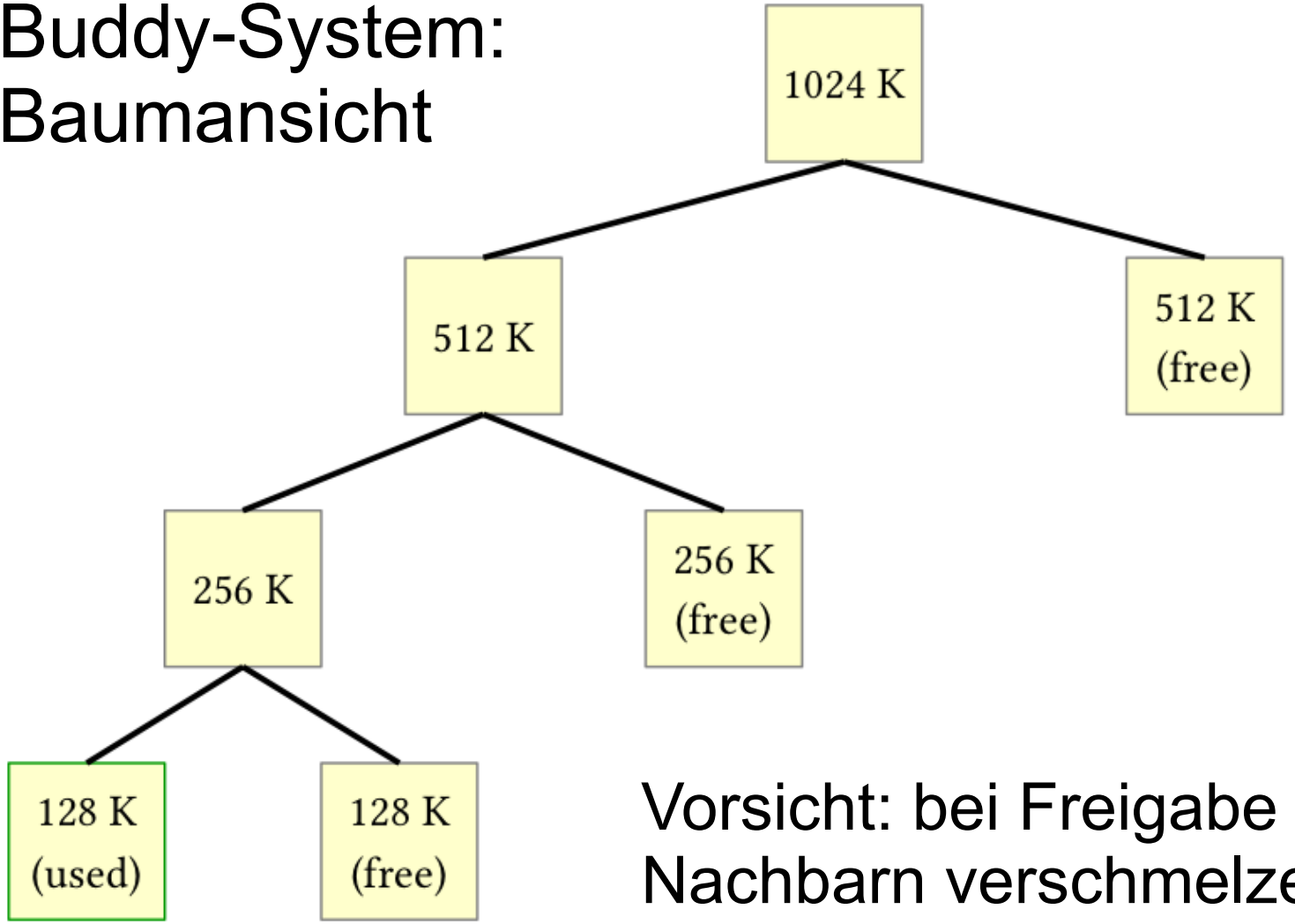
Zusammenhängende Zuteilung (3)

- 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>

Zusammenhängende Zuteilung (4)

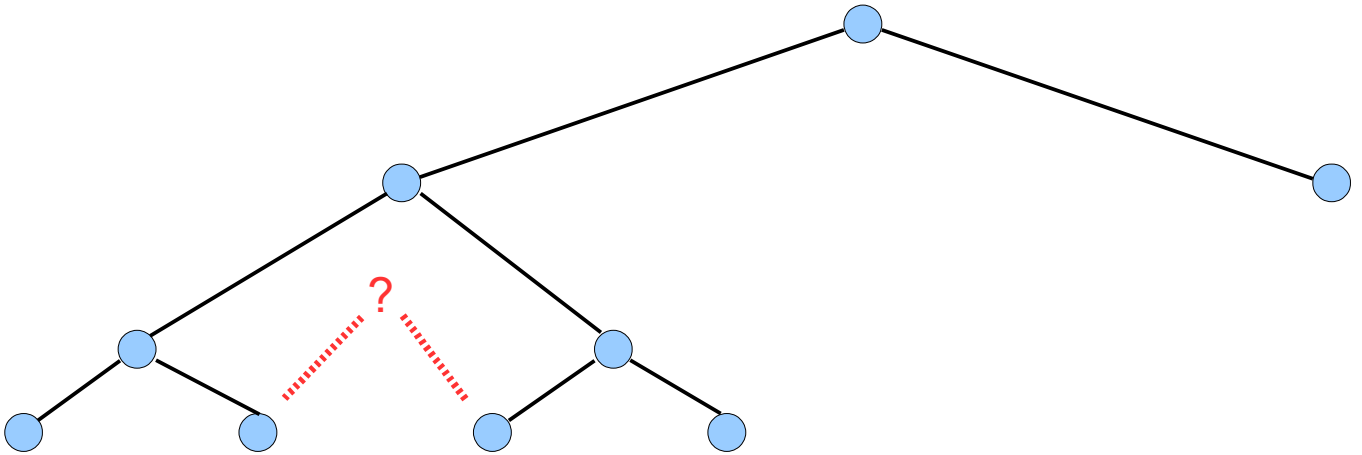
- Buddy-System: Baumansicht



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

Zusammenhängende Zuteilung (5)

- Buddy-System: unmögliche Verschmelzung

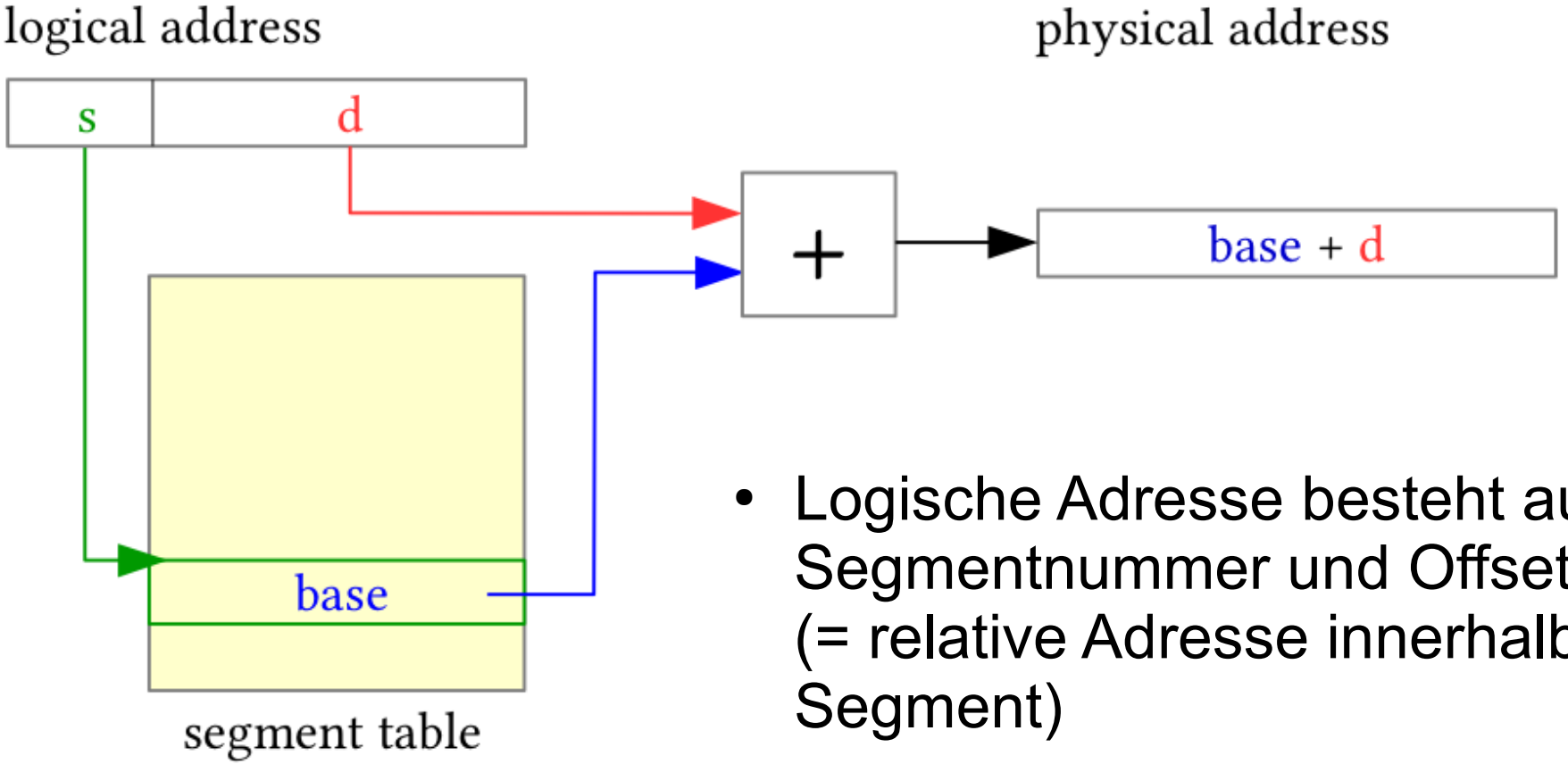


128 KB	128 KB	128 KB	128 KB	512 KB
128 KB	256 KB		128 KB	512 KB

!

Segmentierung (1)

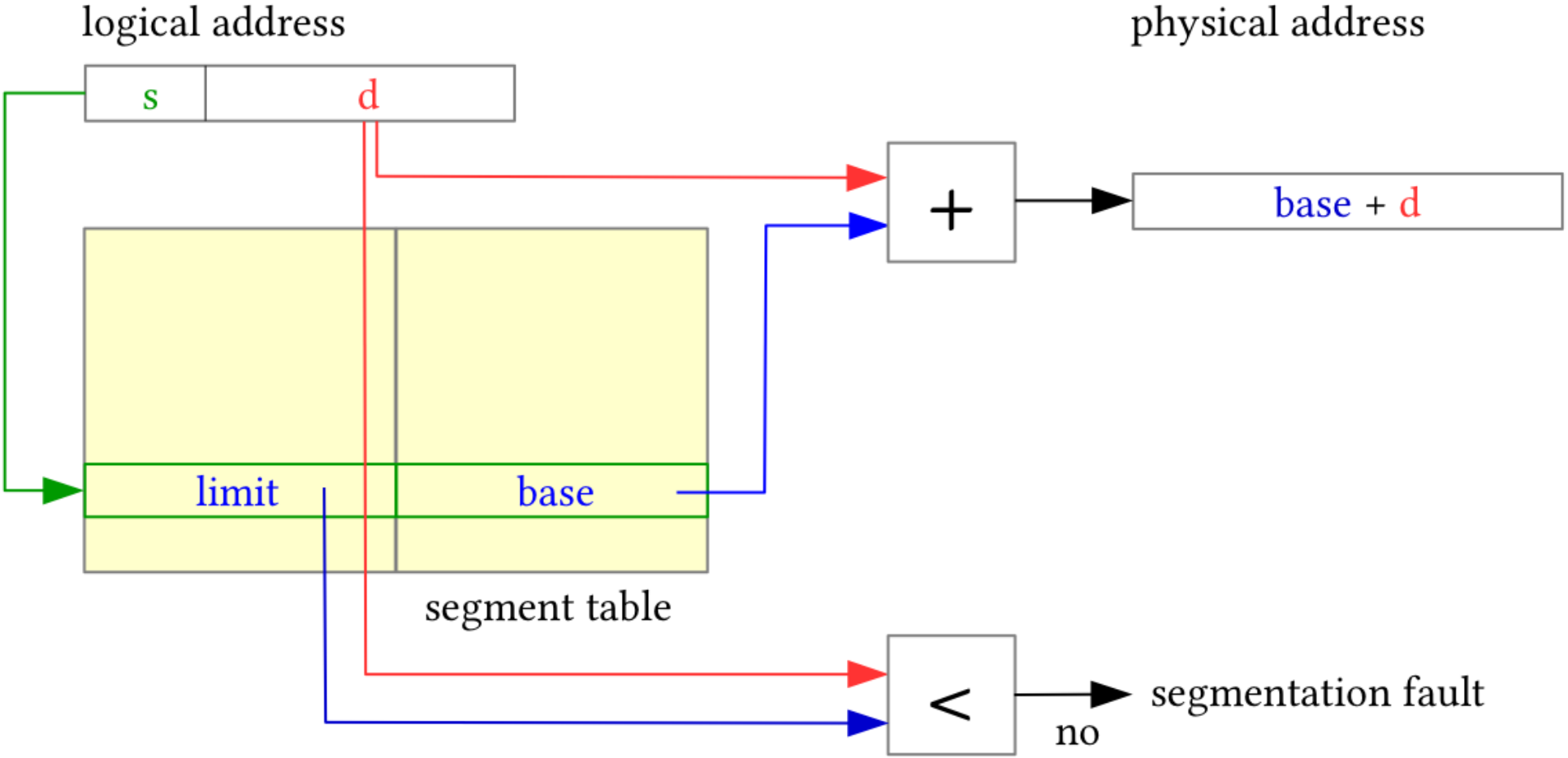
- Über eine Segment-Tabelle wird Speicher in Segmente (zusammenhängende Bereiche) aufgeteilt

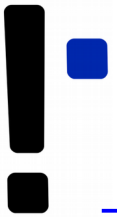


- Logische Adresse besteht aus Segmentnummer und Offset (= relative Adresse innerhalb Segment)

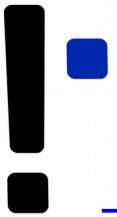
Segmentierung (2)

- Angabe einer Segmentgröße → Prüfung bei Zugriff



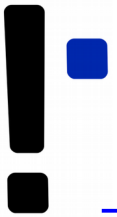


Modernes Verfahren (Paging)



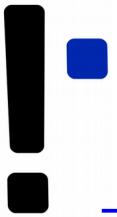
Moderne Lösung des Problems

- Virtueller Speicher, der das gesamte Programm aufnehmen kann
- Programm sieht Speicherbereich, der ihm zur Verfügung gestellt wurde – wie viel wirklich vorhanden ist, spielt (für das Programm) keine Rolle



Virtuelle Speicherverwaltung (Paging)

- Aufteilung des Adressraums in **Seiten** (**pages**) fester Größe und des Hauptspeichers in **Seitenrahmen** (**page frames**) gleicher Größe.
  - Typische Seitengrößen: 512 Byte bis 8192 Byte (immer Zweierpotenz).
- Der lineare, zusammenhängende Adressraum eines Prozesses (**„virtueller“ Adressraum**) wird auf beliebige, nicht zusammenhängende Seitenrahmen abgebildet.
- BS verwaltet eine einzige Liste freier Seitenrahmen

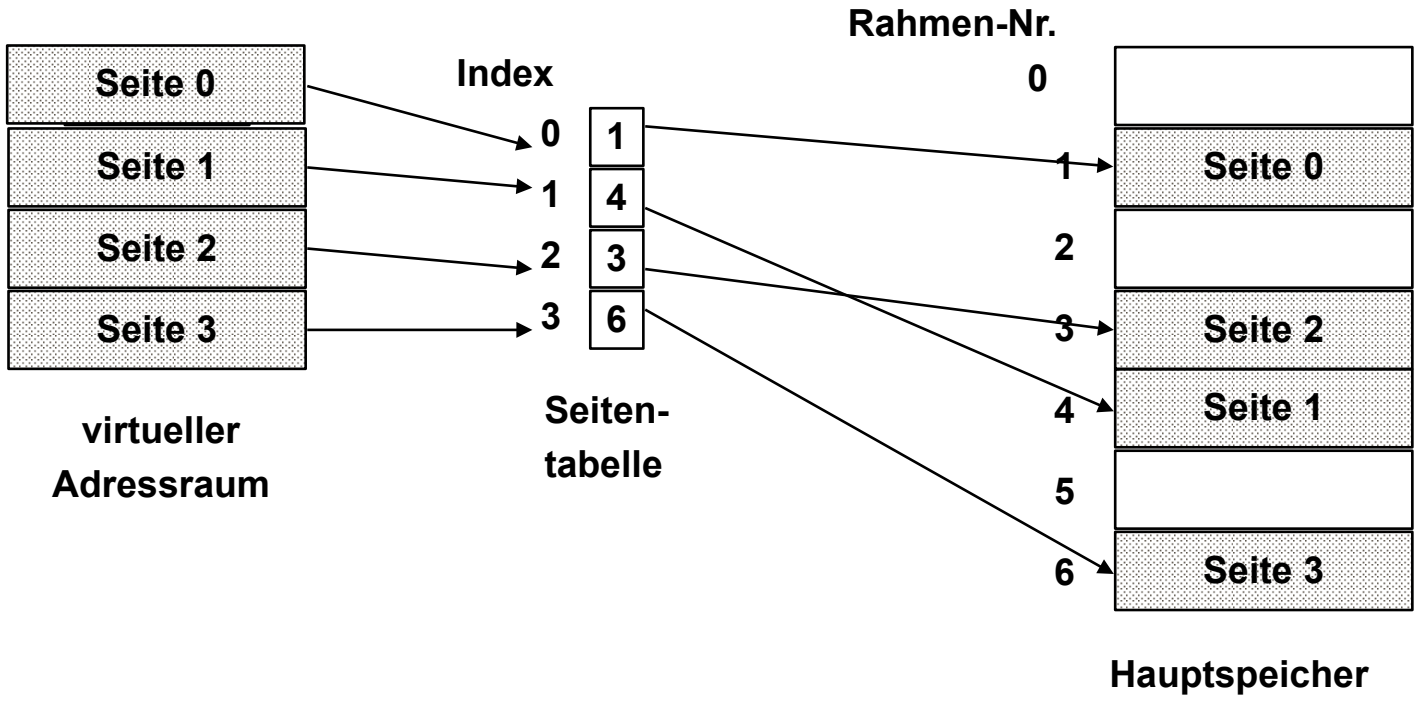


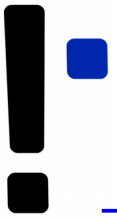
Virtuelle Speicherverwaltung (Paging)

- Die Berechnung der **physikalischen Speicheradresse** aus der vom Programm angegebenen **virtuellen Adresse**
  - geschieht zur Laufzeit des Programms,
  - ist transparent für das Programm,
  - muss von der Hardware unterstützt werden.
- Vorteile der virtuellen Speicherverwaltung:
  - Einfache Zuteilung von Hauptspeicher.
  - Kein Aufwand für den Programmierer.

Virtueller Adressraum (1)

- Paging stellt Zusammenhang zwischen Programmadresse und physikalischer Hauptspeicheradresse erst zur Laufzeit über Seitentabellen her.





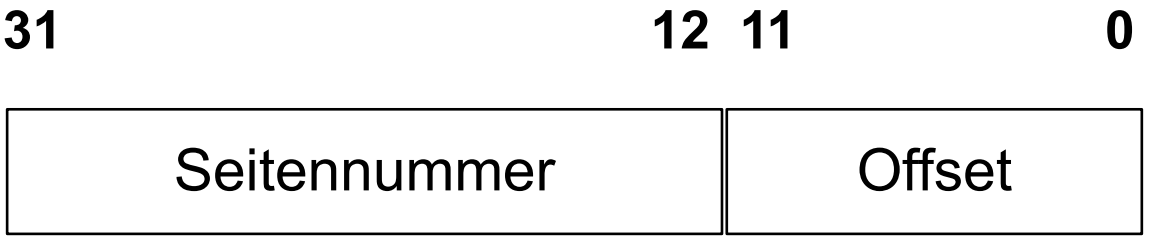
Virtueller Adressraum (2)

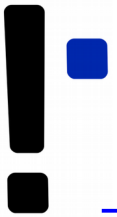
- Die vom Programm verwendeten Adressen werden deshalb auch **virtuelle Adressen** genannt.
- Der **virtuelle Adressraum** eines Programms ist der lineare, zusammenhängende Adressraum, der dem Programm zur Verfügung steht.

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:

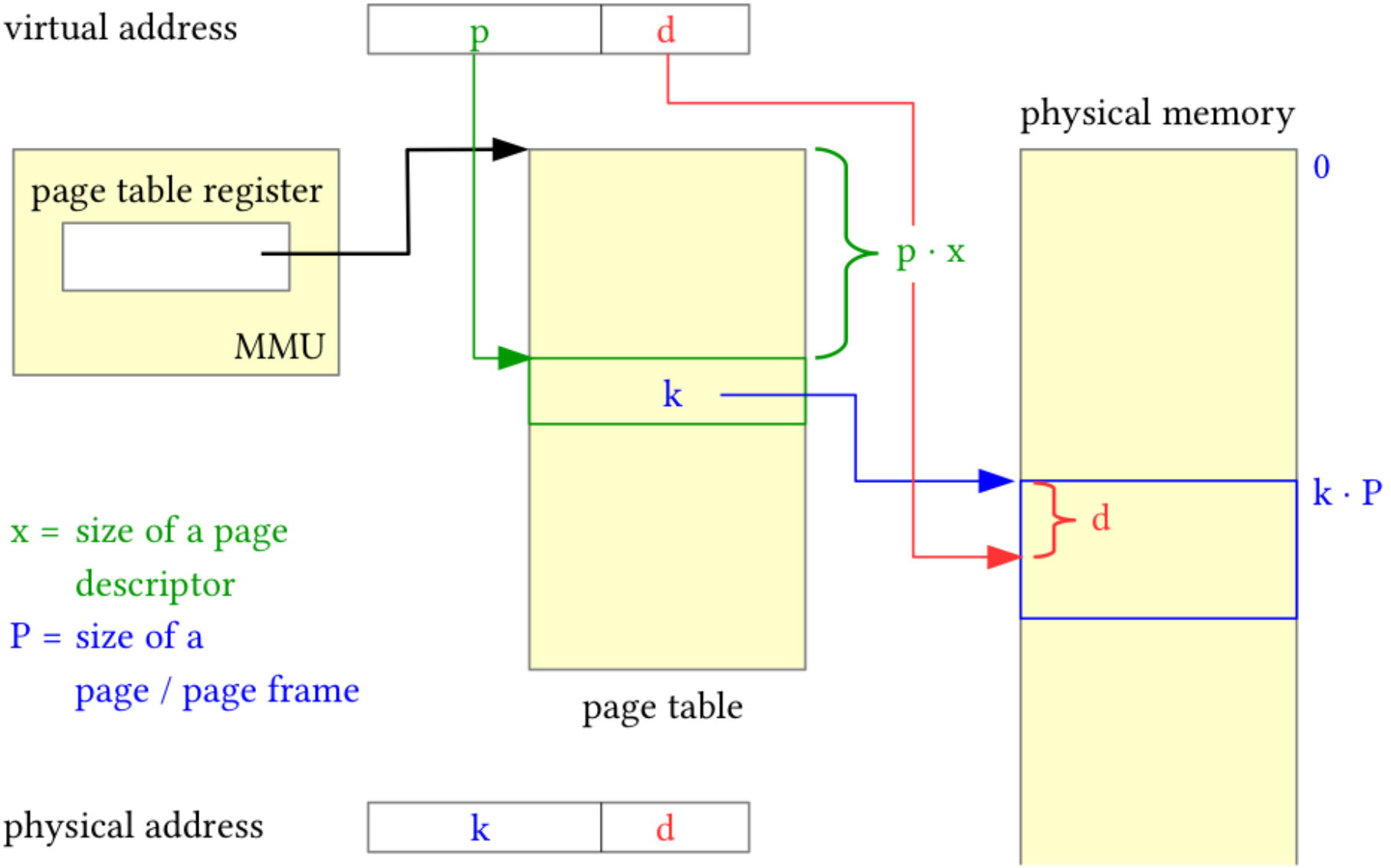


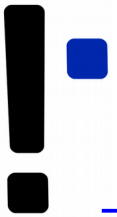


Adressübersetzung beim Paging (2)

- Für jeden Prozess gibt es eine **Seitentabelle** (**page table**). Diese enthält für jede Prozessseite
  - eine Angabe, ob die Seite im Speicher ist,
  - die Nummer des Seitenrahmens im Hauptspeicher, der die Seite enthält.
- Ein spezielles Register (PTR) enthält die Anfangsadresse der Seitentabelle für den aktuellen Prozess.
- Seitennummer dient als Index in die Tabelle.

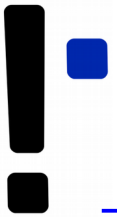
Adressübersetzung beim Paging (3)





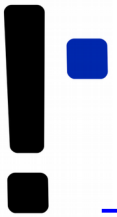
Adressübersetzung beim Paging (4)

- Für jeden Hauptspeicherzugriff wird ein zusätzlicher Hauptspeicherzugriff auf die Seitentabelle benötigt. Dies muss durch Caches in der Hardware beschleunigt werden.
- Seite nicht im Speicher → spezielle Exception, einen sog. page fault (Seitenfehler) auslösen.



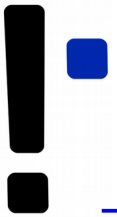
Virtueller Speicher allgemein (1)

- Mehr Prozesse können effektiv im Speicher gehalten werden
  → bessere Systemauslastung
- Ein Prozess kann viel mehr Speicher anfordern als physikalisch verfügbar



Virtueller Speicher allgemein (2)

- allgemeiner Vorgang:
  - Nur Teile des Prozesses befinden sich im physikalischen Speicher
  - falls Zugriff auf eine Adresse, die ausgelagert ist:
    - BS setzt den Prozess auf blockiert
    - BS setzt eine Disk-I/O-Leseanfrage ab
    - Nach Laden der fehlenden Seite wird ein I/O-Interrupt erzeugt
    - das BS setzt Prozess zuletzt wieder in den Bereit- (Ready-) Zustand



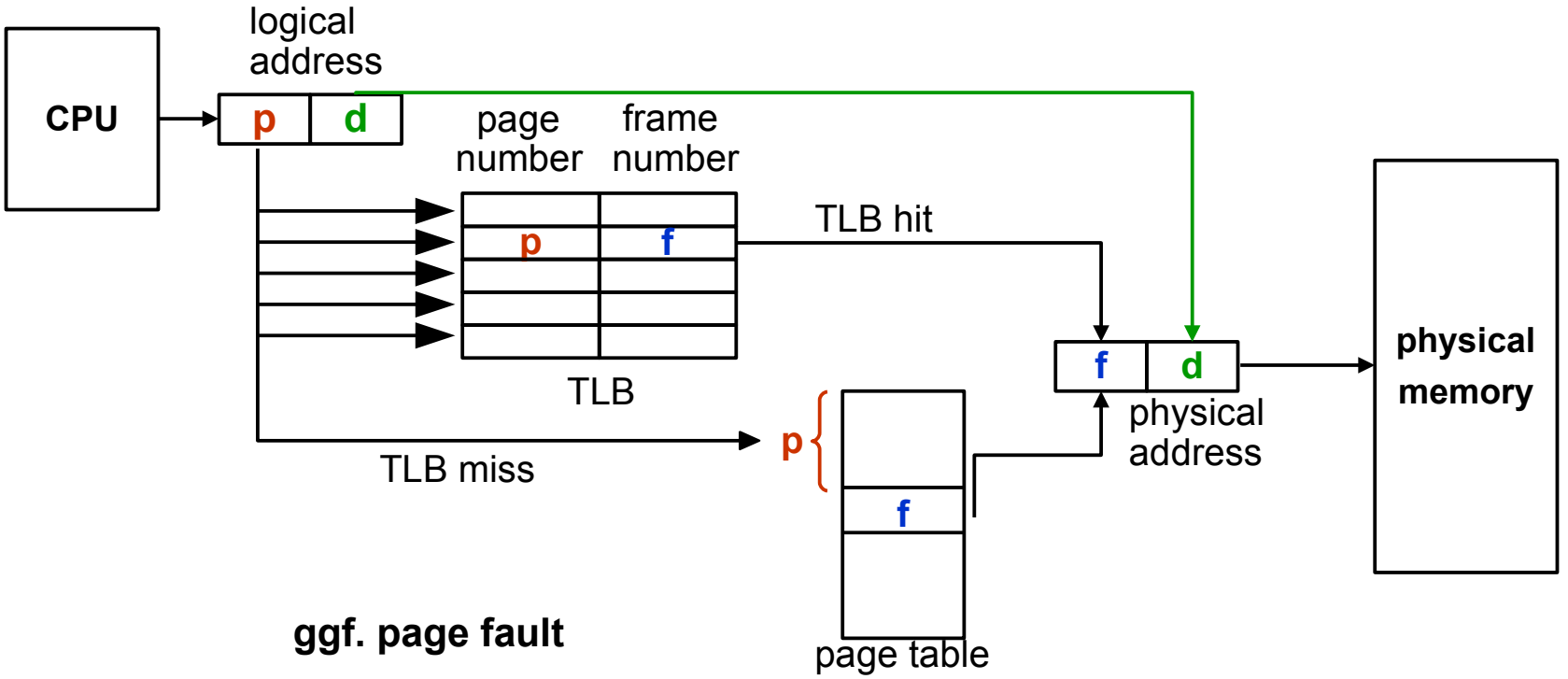
Virtueller Speicher allgemein (3)

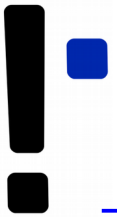
- **Thrashing:** Prozessor verbringt die meiste Zeit mit Ein- und Auslagern von Prozessteilen statt mit der Ausführung von Prozessanweisungen
- **Lokalitätsprinzip:**
  - Zugriffe auf Daten und Programmcode häufig lokal gruppiert;

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

Translation Look-Aside Buffer (1)

- **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.





Translation Look-Aside Buffer (2)

- 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%).

Lokalitätsprinzip

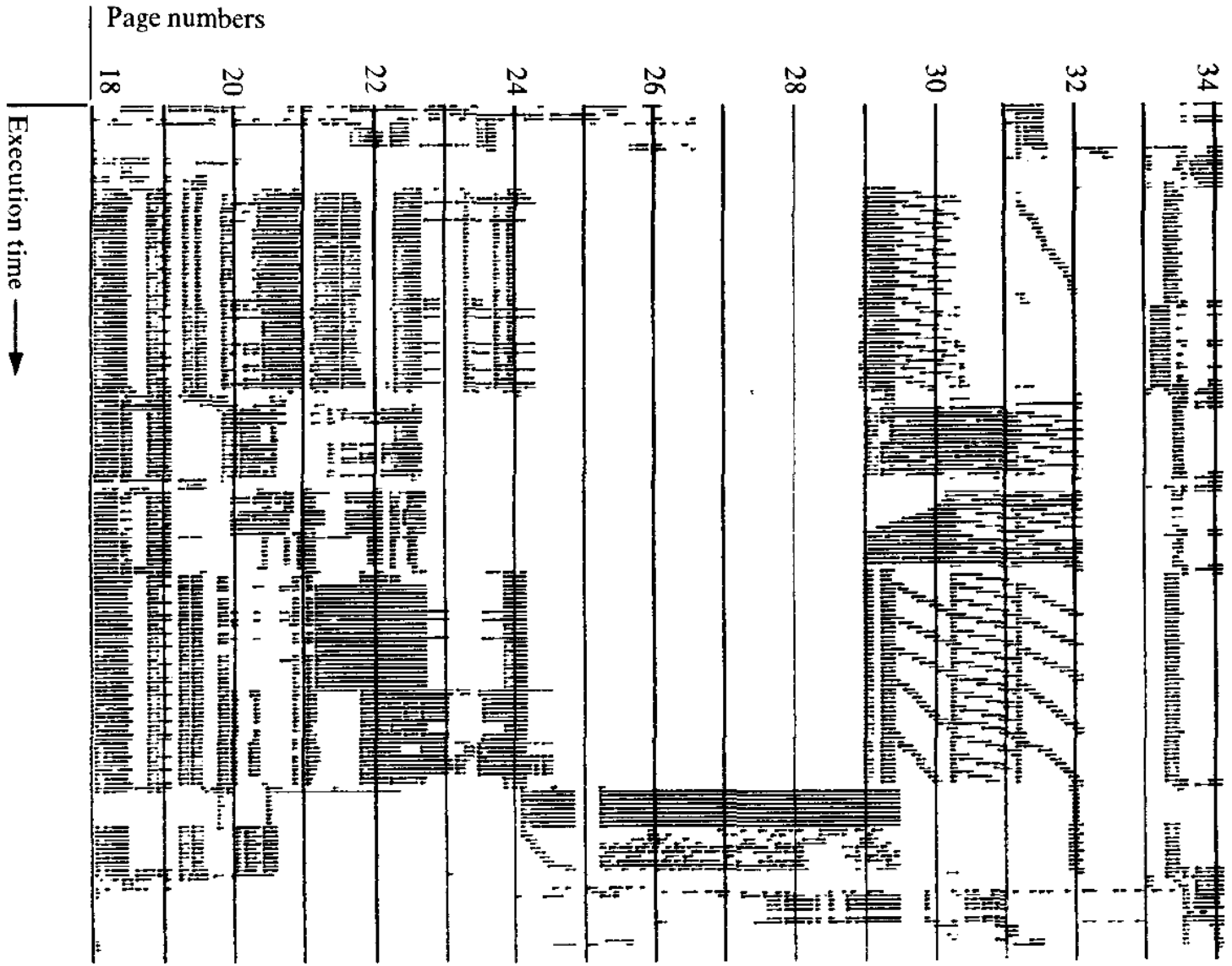
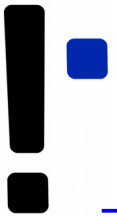
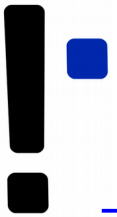


Bild: Hatfield (1972)



Translation Look-Aside Buffer (3)

- Inhalt des TLB ist prozess-spezifisch!
  Zwei Möglichkeiten:
  - Jeder Eintrag enthält ein „valid bit“. Bei Prozess-wechsel (Context Switch) ganzen TLB invalidieren.
  - Jeder Eintrag im TLB enthält Prozessidentifikation (PID), die mit der PID des zugreifenden Prozesses verglichen wird.
- Beispiele für TLB-Größen:
  - Intel 80486: 32 Einträge.
  - Pentium-4, PowerPC-604: 128 Einträge für jeweils Code und Daten.



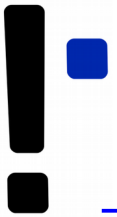
Translation Look-Aside Buffer (4)

Was macht hier eigentlich das Betriebssystem?

- Page-Table-Register laden
- Im Falle eines Page Fault: Fehlende Seite aus dem Swap holen und Seitentabelle aktualisieren
- Evtl. vorher: Seitenverdrängung – welche Seite aus dem Hauptspeicher entfernen? (→ später)

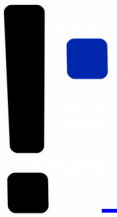
Alles andere: Hardware

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



Invertierte Seitentabellen (1)

- Bei großem virtuellen Speicher sehr viele Einträge in der Seitentabelle nötig,
  z.B. 2^{32} Byte Adressraum, 4 KByte/Seite
  → über 1 Millionen Seiteneinträge,
  also Seitentabelle > 4 MByte (pro Prozess!)
- Platz sparen durch invertierte Seitentabellen:
  - normal: ein Eintrag pro (virtueller) Seite mit Verweis auf den Seitenrahmen (im Hauptspeicher)
  - invertiert: ein Eintrag pro Seitenrahmen mit Verweis auf Tupel (Prozess-ID, virtuelle Seite)

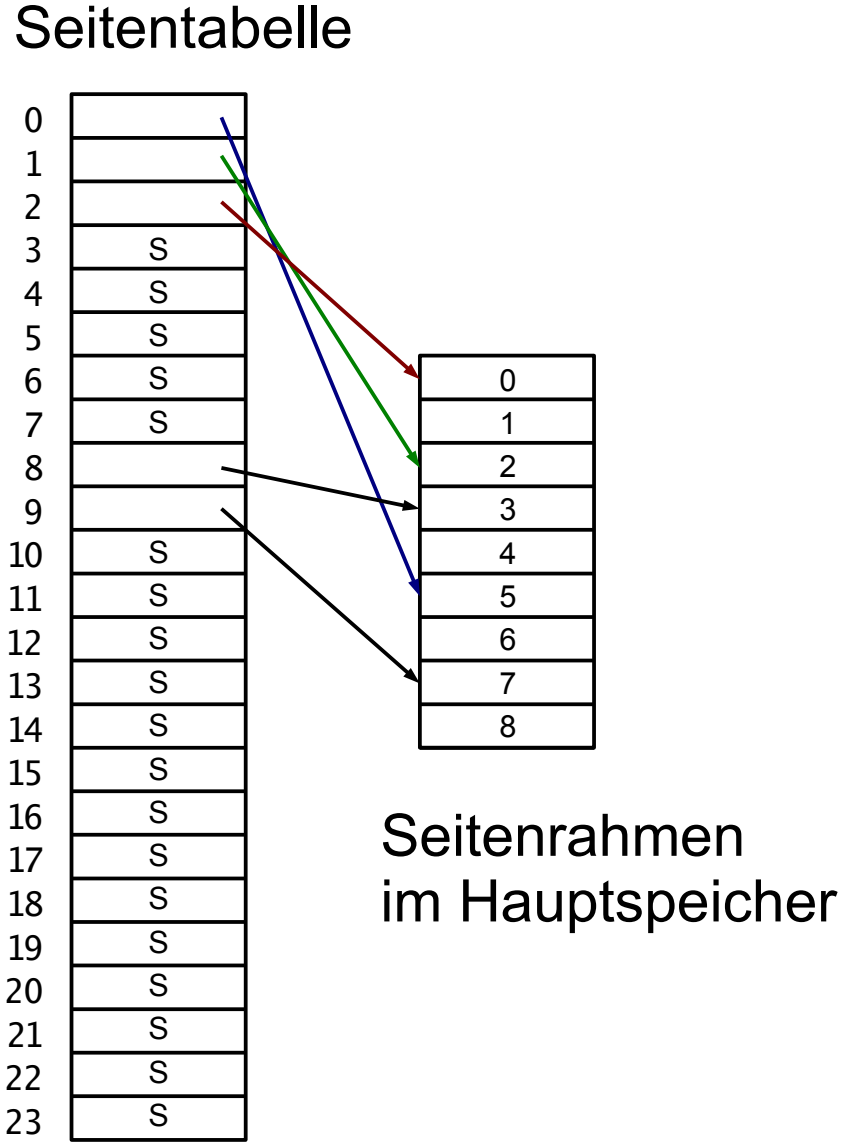


Invertierte Seitentabellen (2)

- Problem: Suche zu Prozess *p* und seiner Seite *n* nach dem Eintrag (*p*,*n*) in der invertierten Tabelle → langwierig
- Auch hier TLB einsetzen, um auf „meist genutzte“ Seiten schnell zugreifen zu können
- Bei TLB-Miss hilft aber nichts: Suchen...

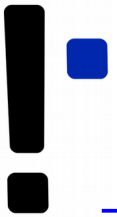
- Andere Lösung für Problem der großen Seitentabellen: Mehrstufiges Paging (→ gleich)

Invertierte Seitentabellen (3)



Invertierte Seitentabelle

0	2
1	–
2	1
3	8
4	–
5	0
6	–
7	9
8	–



Mehrstufiges Paging (1)

Die Seitentabelle kann sehr groß werden.

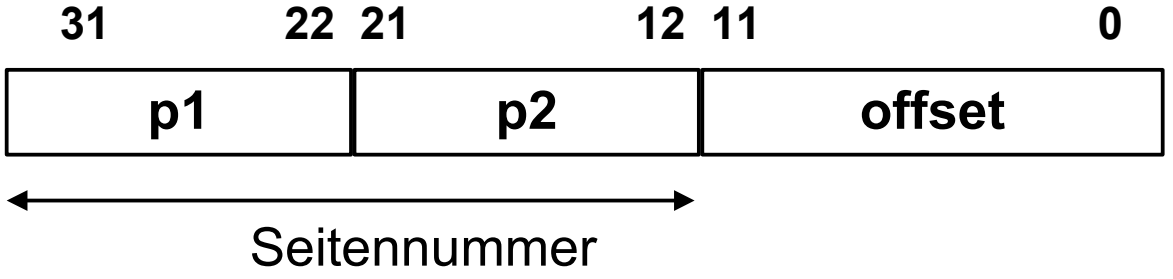
Schon gesehen: typisches Beispiel

- 32-Bit-Adressen,
- 4 KByte Seitengröße,
- 4 Byte pro Eintrag

Seitentabelle: >1 Million Einträge,
4 MByte Größe (pro Prozess!)

Mehrstufiges Paging (2)

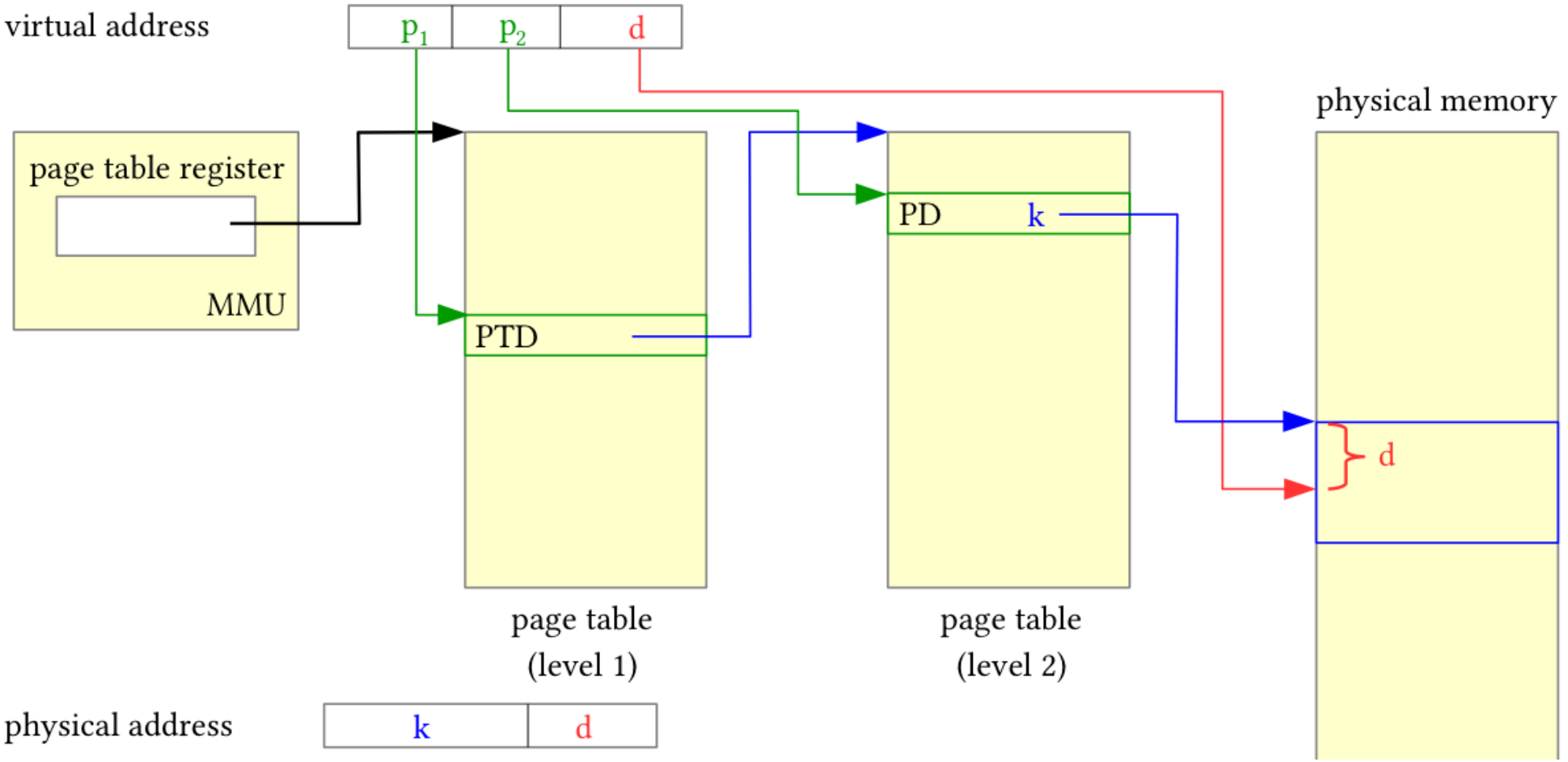
- Zweistufiges Paging:
  - Seitennummer noch einmal unterteilen, z. B.:



  - 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

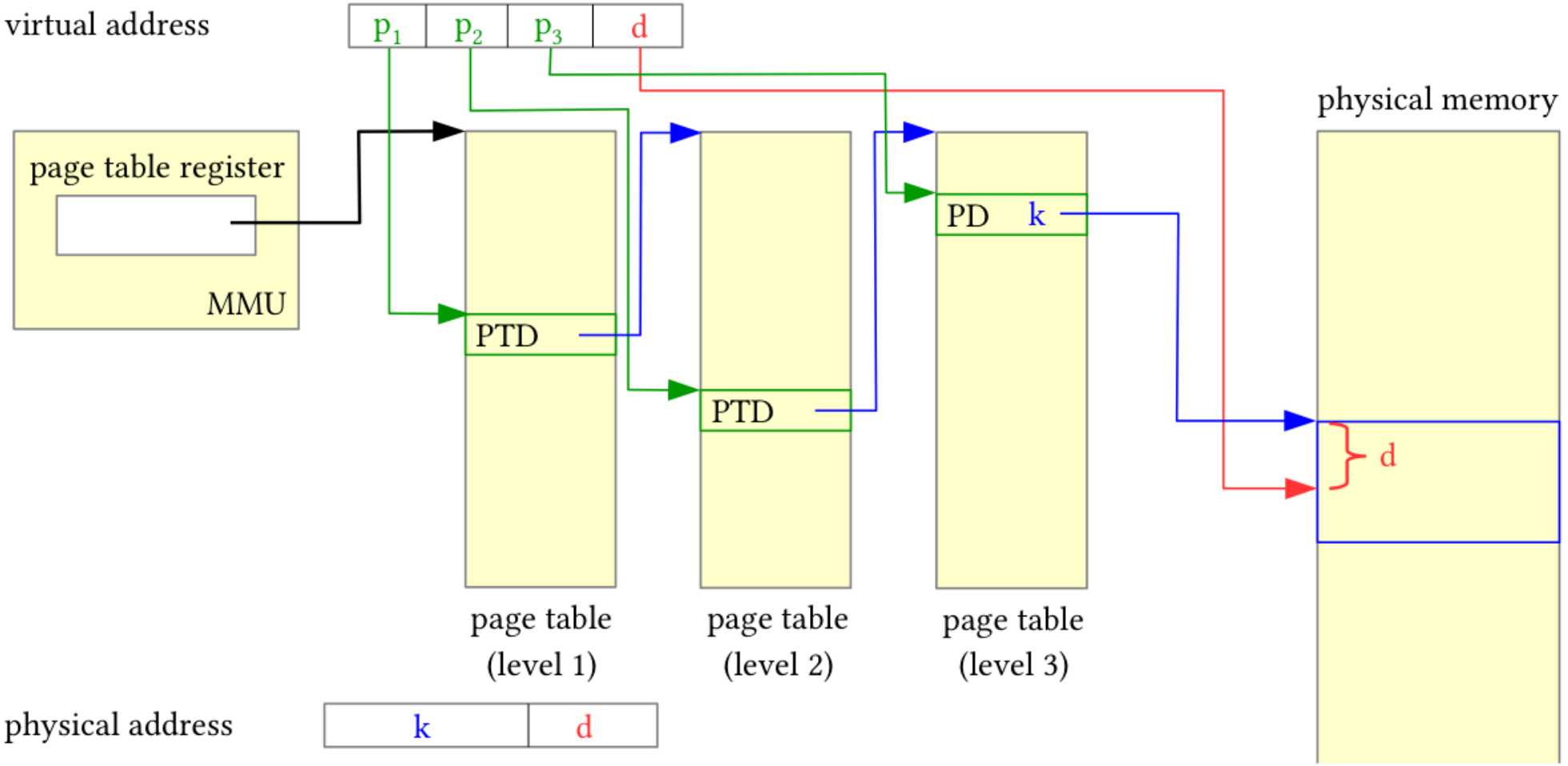
Mehrstufiges Paging (3)

Adressübersetzung bei zweistufigem Paging:



Mehrstufiges Paging (4)

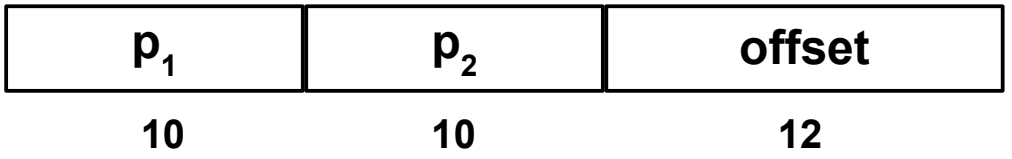
Adressübersetzung bei dreistufigem Paging:



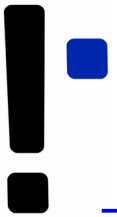
Mehrstufiges Paging (5)

- 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.



Mehrstufiges Paging (6)

- Jede Adressübersetzung benötigt noch mehr Speicherzugriffe, deshalb ist der Einsatz von TLBs noch wichtiger.
- Als Schlüssel für den TLB werden alle Teile der Seitennummer zusammen verwendet (p_1,p_2,...).

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

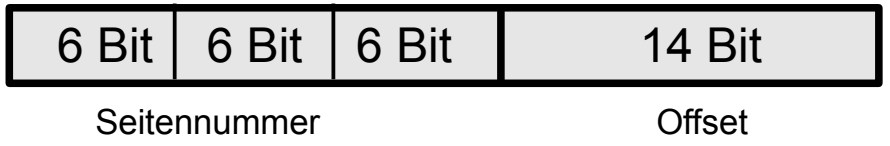
Zu berechnen:

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

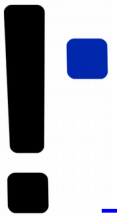


Also gibt es 2^{18} virtuelle Seiten

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.

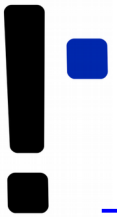
→ Platzbedarf **einer** Tabelle:
#(Einträge) x Größe(Eintrag)
= 2^6 x 4 Byte = 2^8 Byte = 256 Byte

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



Demand Paging (1)

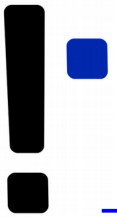
- Der Adressbereich eines Prozesses muss nicht vollständig im Hauptspeicher sein.
  - Das **Lokalitätsprinzip** besagt, dass ein Prozess in einer Zeitspanne nur relativ wenige, nahe beieinanderliegende Adressen anspricht.
  - Teile des Programms werden bei einem bestimmten Ablauf möglicherweise gar nicht benötigt (Spezialfälle, Fehlerbehandlungsroutinen etc.).



Demand Paging (2)

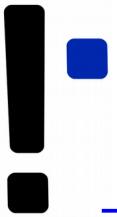
- **Demand Paging** bedeutet
  - dass eine Seite nur dann in den Speicher geladen wird, wenn der Prozess sie anspricht,
  - dass eine Seite auch wieder aus dem Speicher entfernt werden kann.

- Vorteile von Demand Paging:
  - Der Adressbereich eines Prozesses kann größer sein als der physikalische Hauptspeicher.
  - Prozesse belegen weniger Platz im RAM, somit können mehr Prozesse gleichzeitig aktiv sein.



Voraussetzungen für Demand Paging (1)

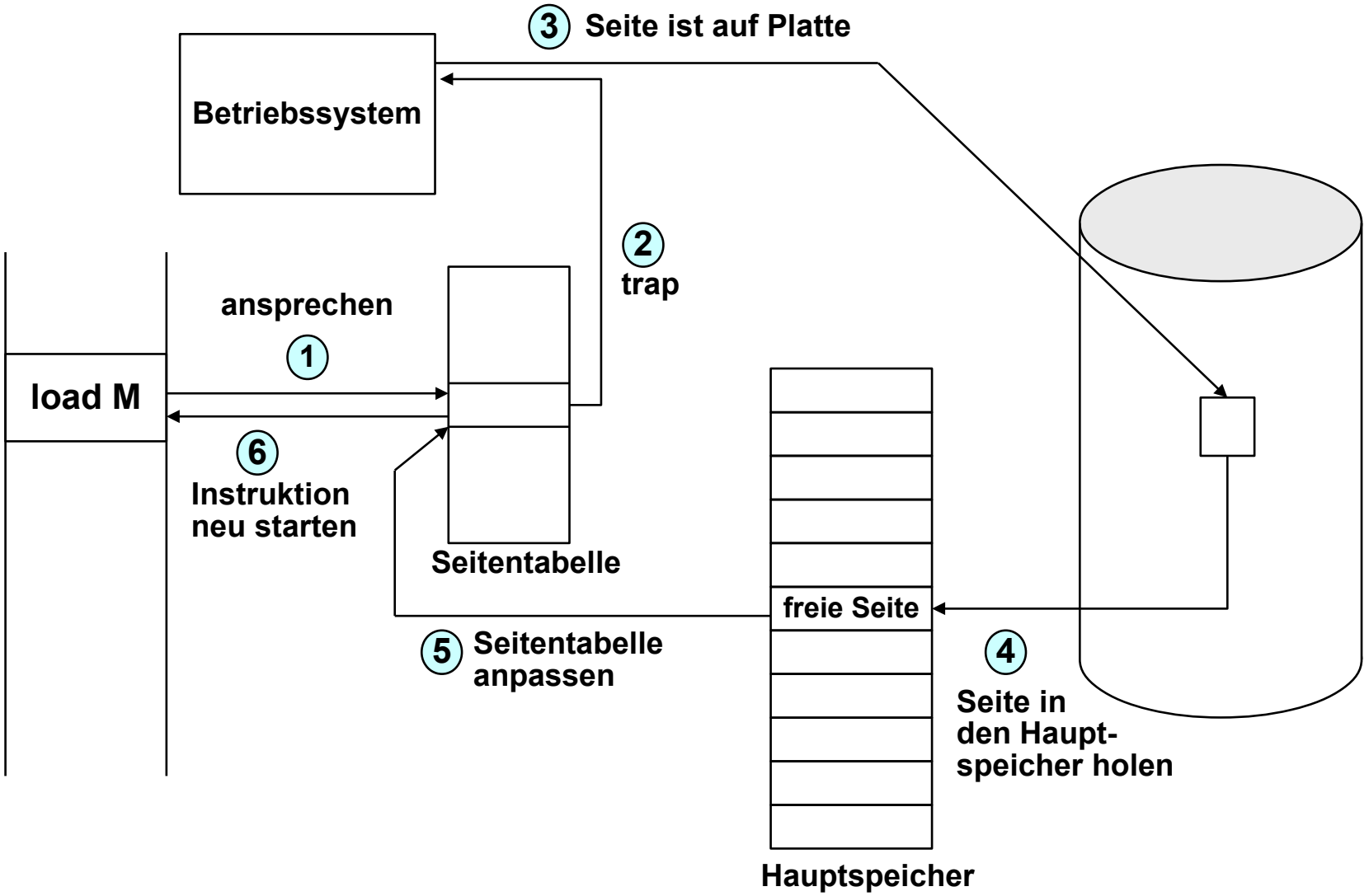
- Jeder Eintrag in der Seitentabelle enthält ein **valid bit** (auch: **present bit**), das angibt, ob die Seite im Speicher ist oder nicht.
- Wenn ein Prozess eine Seite anspricht, die nicht im Speicher ist, wird eine spezielle Exception ausgelöst, ein sog. **page fault**.
- Eine Betriebssystem-Routine, der **page fault handler**, lädt bei einem page fault die benötigte Seite in den Speicher.

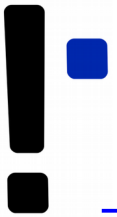


Voraussetzungen für Demand Paging (2)

- Falls kein freier Seitenrahmen im Speicher vorhanden ist, muss eine andere Seite ersetzt werden. Für die Auswahl der zu ersetzenden Seite muss eine Strategie implementiert werden.
- Die durch den page fault unterbrochene Instruktion muss erneut ausgeführt werden (können).

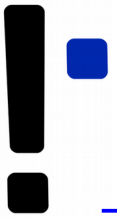
Page-Fault-Behandlung





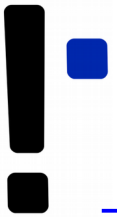
Seitenersetzung (1)

- Wenn bei einem Page Fault kein freier Seitenrahmen zur Verfügung steht, muss das Betriebssystem einen frei machen.
- Ein Algorithmus wählt nach einer bestimmten Strategie diesen Seitenrahmen aus.



Seitenersetzung (2)

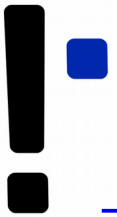
- Falls die zu ersetzende Seite, seit sie zuletzt in den Speicher geholt wurde, verändert wurde, muss ihr aktueller Inhalt gesichert werden:
  - Ein **modify bit** (oder **dirty bit**) im Seitentabelleneintrag vermerkt, ob die Seite verändert wurde.
  - Eine veränderte Seite wird auf Platte gesichert (im sog. Page- oder Swap-Bereich).



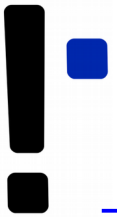
Seitenersetzung (3)

- Eine unveränderte Seite kann später – bei Bedarf – wieder von der alten Stelle auf der Platte geladen werden.

- Im Seitentabelleneintrag für die ersetzte Seite wird
  - das **valid bit** gelöscht,
  - vermerkt, von wo die Seite wieder geladen werden kann.

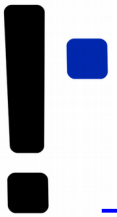


Praxis: Linux



Speicherverwaltung unter Linux

- Paging (Linux)
- Aufbau des Prozess-Speichers unter Linux
- `malloc()`, `calloc()`, `realloc()` und `free()`
- `memset()`
- `memcpy()`, `memcmp()`
- Alignment
- Anonymous Memory Mapping mit `mmap()`

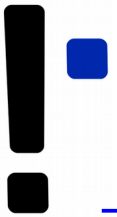


Paging unter Linux

- Linux verwendet **Paging** für die Speicherverwaltung
  - Seitengröße 4 KByte (Intel 32/64 bit)
    oder variabel: 4, 16, 64, 256 KByte (PPC32)

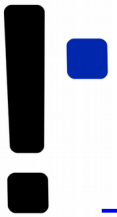
```
esser@ubu64:~$ cat getpagesize.c
#include <unistd.h>
#include <stdio.h>
int main () { printf ("page size: %d\n", getpagesize()); };
esser@ubu64:~$ gcc -o getpagesize getpagesize.c ; ./getpagesize
page size: 4096
esser@ubu64:~$ uname -m
x86_64
```

  - nutzt **Page Sharing** (z. B. für Bibliotheken) und **Copy-on-Write** (z. B. für `fork`)



Prozess-Speicher

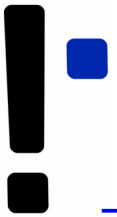
- **Text-Segment**
  - Programm-Code, String-Literale, Konstanten
  - read-only, direkt auf Programmdatei gemappt
- **Stack**
  - lokale Variablen, Funktionsrückgabewerte, Rücksprungadressen
  - wächst und schrumpft nach Bedarf
- **Data-Segment**
  - globale, initialisierte Variablen
  - **Heap**, dynamischer Prozess-Speicher
  - verwaltet Prozess über `malloc()`, `free()` etc.



Prozess-Speicher

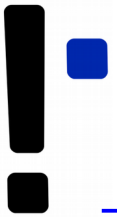
- **BSS-Segment**
  - nicht initialisierte globale Variablen
  - werden bei Prozess-Start auf 0 initialisiert

    (landen nicht in der Objektdatei eines Programms)
  - implementiert über Copy-on-Write-Mapping auf Seite mit Nullen
- **Mapped Files** (`mmap`)



Prozess-Speicher

- **Anonymous Memory Mappings**
  - für große Speicher-Anforderungen (`malloc`), die nicht auf dem Heap landen
  - `glibc` entscheidet abhängig von Größe, ob Anon-Mapping oder Heap (bis 128 K) verwendet wird
  - vermeidet Fragmentierung des Heap
  - sind schon mit 0 gefüllt



Beispiel für Speicher-Aufteilung

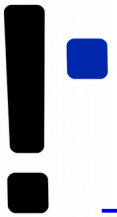
```
// memtest.c
// Hans-Georg Eßer, Betriebssysteme 1
#include <stdio.h>    // printf
#include <unistd.h>   // sleep
#include <stdlib.h>   // malloc

char chararray[1024];                 // BSS-Segment  (nicht init.)
const int i = 7;                      // Text-Segment (konstant)
int j = 4095;                         // Data-Segment (initialisiert)

void testfunc () {
  int array[4096];                    // Stack
  array[i] = 3; array[j] = 5;
  chararray[1] = 'B';
  printf ("Test: %d, %d\n", array[i], array[j]);
};

int main () {
  chararray[0] = 'A'; chararray[2] = '\0';
  testfunc ();
  printf ("Test: %s\n", chararray);
  char* s = malloc (40*1024*1024);  // 40 MByte, Heap
  sleep (20);
  return 0;
};
```

Linux, 32 Bit

```
[esser@quadamd:tmp]$ pmap $(pidof memtest)
30387:   ./memtest
08048000      4K r-x--  /tmp/memtest
08049000      4K r----  /tmp/memtest
0804a000      4K rw---  /tmp/memtest
b4f22000  40968K rw---    [ anon ]
b7724000   1496K r-x--  /lib/i386-linux-gnu/libc-2.13.so
b789a000      8K r----  /lib/i386-linux-gnu/libc-2.13.so
b789c000      4K rw---  /lib/i386-linux-gnu/libc-2.13.so
b789d000     12K rw---    [ anon ]
b78c5000     12K rw---    [ anon ]
b78c8000      4K r-x--    [ anon ]
b78c9000    120K r-x--  /lib/i386-linux-gnu/ld-2.13.so
b78e7000      4K r----  /lib/i386-linux-gnu/ld-2.13.so
b78e8000      4K rw---  /lib/i386-linux-gnu/ld-2.13.so
bf99a000    132K rw---    [ stack ]
 total    42776K

[esser@quadamd:tmp]$ cat /proc/$(pidof memtest)/maps
08048000-08049000 r-xp 00000000 08:04 1156248    /tmp/memtest
08049000-0804a000 r--p 00000000 08:04 1156248    /tmp/memtest
0804a000-0804b000 rw-p 00001000 08:04 1156248    /tmp/memtest
b4f22000-b7724000 rw-p 00000000 00:00 0
b7724000-b789a000 r-xp 00000000 08:04 1966089    /lib/i386-linux-gnu/libc-2.13.so
b789a000-b789c000 r--p 00176000 08:04 1966089    /lib/i386-linux-gnu/libc-2.13.so
b789c000-b789d000 rw-p 00178000 08:04 1966089    /lib/i386-linux-gnu/libc-2.13.so
b789d000-b78a0000 rw-p 00000000 00:00 0
b78c5000-b78c8000 rw-p 00000000 00:00 0
b78c8000-b78c9000 r-xp 00000000 00:00 0          [vdso]
b78c9000-b78e7000 r-xp 00000000 08:04 9569185    /lib/i386-linux-gnu/ld-2.13.so
b78e7000-b78e8000 r--p 0001d000 08:04 9569185    /lib/i386-linux-gnu/ld-2.13.so
b78e8000-b78e9000 rw-p 0001e000 08:04 9569185    /lib/i386-linux-gnu/ld-2.13.so
bf99a000-bf9bb000 rw-p 00000000 00:00 0          [stack]
```

```
esser@ubu64:~$ pmap $(pidof memtest)
1839:   ./memtest
0000000000400000      4K r-x--  /home/esser/memtest
0000000000600000      4K r----  /home/esser/memtest
0000000000601000      4K rw---  /home/esser/memtest
00007f06b1887000  40964K rw---    [ anon ]
00007f06b4088000   1512K r-x--  /lib/libc-2.11.1.so
00007f06b4202000   2044K -----  /lib/libc-2.11.1.so
00007f06b4401000     16K r----  /lib/libc-2.11.1.so
00007f06b4405000      4K rw---  /lib/libc-2.11.1.so
00007f06b4406000     20K rw---    [ anon ]
00007f06b440b000    128K r-x--  /lib/ld-2.11.1.so
00007f06b460d000     12K rw---    [ anon ]
00007f06b4627000     12K rw---    [ anon ]
00007f06b462a000      4K r----  /lib/ld-2.11.1.so
00007f06b462b000      4K rw---  /lib/ld-2.11.1.so
00007f06b462c000      4K rw---    [ anon ]
00007fff5fc9b000     84K rw---    [ stack ]
00007fff5fdff000      4K r-x--    [ anon ]
ffffffffff600000      4K r-x--    [ anon ]
 total            44828K
esser@ubu64:~$ cat /proc/$(pidof memtest)/maps
00400000-00401000                 r-xp 00000000 08:01 100804             /home/esser/memtest
00600000-00601000                 r--p 00000000 08:01 100804             /home/esser/memtest
00601000-00602000                 rw-p 00001000 08:01 100804             /home/esser/memtest
7f06b1887000-7f06b4088000         rw-p 00000000 00:00 0
7f06b4088000-7f06b4202000         r-xp 00000000 08:01 8765               /lib/libc-2.11.1.so
7f06b4202000-7f06b4401000         ---p 0017a000 08:01 8765               /lib/libc-2.11.1.so
7f06b4401000-7f06b4405000         r--p 00179000 08:01 8765               /lib/libc-2.11.1.so
7f06b4405000-7f06b4406000         rw-p 0017d000 08:01 8765               /lib/libc-2.11.1.so
7f06b4406000-7f06b440b000         rw-p 00000000 00:00 0
7f06b440b000-7f06b442b000         r-xp 00000000 08:01 8718               /lib/ld-2.11.1.so
7f06b460d000-7f06b4610000         rw-p 00000000 00:00 0
7f06b4627000-7f06b462a000         rw-p 00000000 00:00 0
7f06b462a000-7f06b462b000         r--p 0001f000 08:01 8718               /lib/ld-2.11.1.so
7f06b462b000-7f06b462c000         rw-p 00020000 08:01 8718               /lib/ld-2.11.1.so
7f06b462c000-7f06b462d000         rw-p 00000000 00:00 0
7fff5fc9b000-7fff5fcb0000         rw-p 00000000 00:00 0                  [stack]
7fff5fdff000-7fff5fe00000         r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

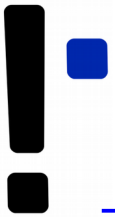
```
esser@ubu64:~$ size memtest
 text    data     bss     dec     hex   filename
 1538     536    1056    3130    c3a   memtest
```



malloc()

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

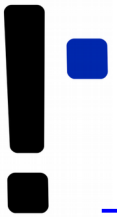
```
char *p;
p = malloc (2000)   // 2000 Bytes anfordern
if (!p) {
  // Fehler
  perror ("malloc");
}

...

free (p);
```



xmalloc()

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

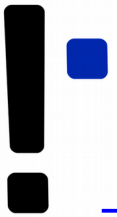
```
void *xmalloc (size_t size) {
  void *p;
  p = malloc (size);
  if (!p) {
    perror ("xmalloc");
    exit (EXIT_FAILURE);    // returns 1
  };
  return p;
};
```



calloc()

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

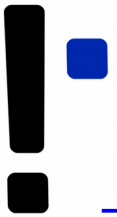
```
struct mystruct { ... };
struct mystruct *p;
p = calloc (200, sizeof(struct mystruct));  // 200 Einträge

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 Speicher-bereich, der jetzt an anderer Stelle anfangen kann!

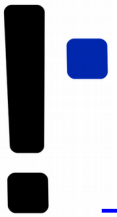
```
p = malloc (10*sizeof(struct xy));
...
r = realloc (p, 20*sizeof(struct xy));
if (!r) {
  // Fehler, p noch intakt!
}
...                     // evtl. r != p

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
```

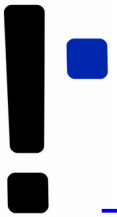


free()

- Shell-Variable `MALLOC_CHECK_` erlaubt Einsatz einer alternativen `malloc()`-Implementierung, die z. B. doppelte `free()`s erkennt

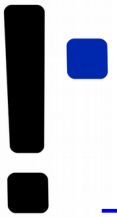
```
// free2.c
#include <stdlib.h>
#include <stdio.h>

int main () {
  int* p = malloc (200);
  free (p);
  free (p);  // Fehler!!
  printf ("nach 2x free\n");
};
```

```
esser@ubu64:~$ ./free2
(Programm bricht ab, Backtrace etc.)

esser@ubu64:~$ MALLOC_CHECK_=0 ./free2
nach 2x free

esser@ubu64:~$ MALLOC_CHECK_=1 ./free2
*** glibc detected *** ./free2: free():
invalid pointer: 0x0000000001780010 ***
nach 2x free

esser@ubu64:~$ MALLOC_CHECK_=2 ./free2
Abgebrochen
```



Verwaltung des Heap

- Es gibt verschiedene Speicher-Allokations-Routinen
- Linux-Programme (mit `glibc`) nutzen Variante von `dlmalloc` („Doug Lea's malloc“), http://g.oswego.edu/dl/html/malloc.html
  - Best-Fit (für Anforderungen >= 256 Byte, < 256 KByte)
  - nutzt OS-Features ab 256 KByte (→ Anon-Mapping)
  - Sonderbehandlung für kleine Anforderungen
- Beschreibung von `dlmalloc`:
  → ftp://g.oswego.edu/pub/misc/malloc.c
  → http://www.securecoding.cert.org/confluence/download/attachments/3524/04+Dynamic+Memory+v6.pdf (Folie 58-65)
- Tutorial: http://www.inf.udec.cl/~leo/Malloc_tutorial.pdf

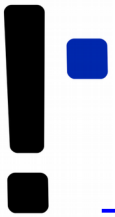


memset()

- 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 --
```



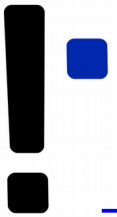
memcpy()

- kopiert einen Speicherbereich:

  `memcpy (ziel, quelle, laenge)`

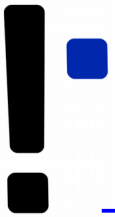
- Rückgabewert: Zeiger auf `ziel`

```
#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);
};
```



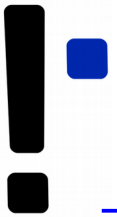
`memcpy()` vs. `strncpy()`

- 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()

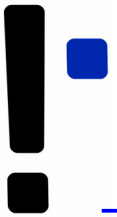
- `memcmp (a, b, len)` vergleicht Speicherbereiche

```
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



Alignment

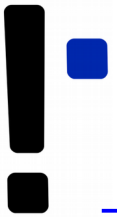
- `malloc()` & Co. lassen allozierte Speicherbereiche immer an Adressen anfangen, die ein Vielfaches von 8 (32 Bit) bzw. 16 (64 Bit) sind

  → Variablen aller Typen sind „**naturally aligned**“

```
// malloc-align-test.c
#include <stdlib.h>
#include <stdio.h>
int main () {
  int* a = malloc(1);
  int* b = malloc(1);
  int* c = malloc(1);
  printf ("a: %p\n", a);
  // %p: Addresse eines Pointers
  printf ("b: %p\n", b);
  printf ("c: %p\n", c);
}
```

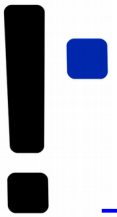
```
esser@ubu64:~$ uname -m
x86_64
esser@ubu64:~$ ./malloc-align-test
a: 0x6cc010
b: 0x6cc030
c: 0x6cc050
```

(Abstand: 0x20 = 32, Extraplatz für Verwaltungsdaten von `malloc`, → nächste Folie)



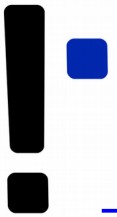
Extra-Platz bei `malloc()`

```
// malloc-test.c

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

void dump (char *filename, char *buf, int len) {
  int fd = open (filename, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
  write (fd, buf, len);
  close (fd);
}

int main () {
  char *p1 = malloc(4);       // gibt: p1 = 0x23e6010
  strncpy (p1, "ABC", 4);
  dump ("out1", p1-16, 96);

  char *p2 = malloc(4);       // gibt: p2 = 0x23e6030,  Differenz 0x20
  strncpy (p2, "XYZ", 4);
  dump ("out2", p1-16, 96);

  free(p2);
  dump ("out3", p1-16, 96);
}
```



Extra-Platz bei `malloc()`

```
[esser@ohmvm:~]$ ./malloc-test

[esser@ohmvm:~]$ hexdump -C out1    # nach dem 1. malloc() / strncpy()
00000000  00 00 00 00 00 00 00 00  21 00 00 00 00 00 00 00  |........!.......|
00000010  41 42 43 00 00 00 00 00  00 00 00 00 00 00 00 00  |ABC.............|
00000020  00 00 00 00 00 00 00 00  e1 0f 02 00 00 00 00 00  |................|
...

[esser@ohmvm:~]$ hexdump -C out2    # nach dem 2. malloc() / strncpy()
00000000  00 00 00 00 00 00 00 00  21 00 00 00 00 00 00 00  |........!.......|
00000010  41 42 43 00 00 00 00 00  00 00 00 00 00 00 00 00  |ABC.............|
00000020  00 00 00 00 00 00 00 00  21 00 00 00 00 00 00 00  |........!.......|
00000030  58 59 5a 00 00 00 00 00  00 00 00 00 00 00 00 00  |XYZ.............|
00000040  00 00 00 00 00 00 00 00  c1 0f 02 00 00 00 00 00  |................|
...

[esser@ohmvm:~]$ hexdump -C out3    # nach dem free()
00000000  00 00 00 00 00 00 00 00  21 00 00 00 00 00 00 00  |........!.......|
00000010  41 42 43 00 00 00 00 00  00 00 00 00 00 00 00 00  |ABC.............|
00000020  00 00 00 00 00 00 00 00  21 00 00 00 00 00 00 00  |........!.......|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  c1 0f 02 00 00 00 00 00  |................|
...
```



Alignment

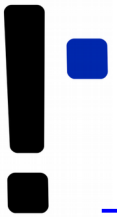
- Durch **Cast-Operationen** können Variablen entstehen, die nicht aligned sind:

```
// wrong-alignment.c
#include <stdio.h>

int main () {
  char *str = "ABCDEFGHIJK";
  char *c = str + 1;
  putchar (*c); printf ("\n");
  unsigned long *u1, *u2;
  u1 = (unsigned long *) str;
  u2 = (unsigned long *) c;
  printf ("Pointer c:  %p\n",  c);
  printf ("Pointer u1: %p\n",  u1);
  printf ("Inhalt u1:  %lx\n", *u1);
  printf ("Pointer u2: %p\n",  u2);
  printf ("Inhalt u2:  %lx\n", *u2);
};
```

```
esser@ubu64:~$ ./wrong-alignment
B
Pointer c:  0x40072d
Pointer u1: 0x40072c
Inhalt u1:  4847464544434241
Pointer u2: 0x40072d  <-- 0xd = 13
Inhalt u2:  494847464544434 2
```

ASCII-Tabelle:

A	0x41	E	0x45
B	0x42	F	0x46
C	0x43	G	0x47
D	0x44	H	0x48

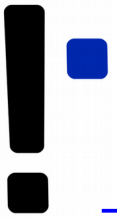


Anonymous Memory Mapping

- 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