

# Betriebssysteme

SS 2015

**Hans-Georg Eßer**

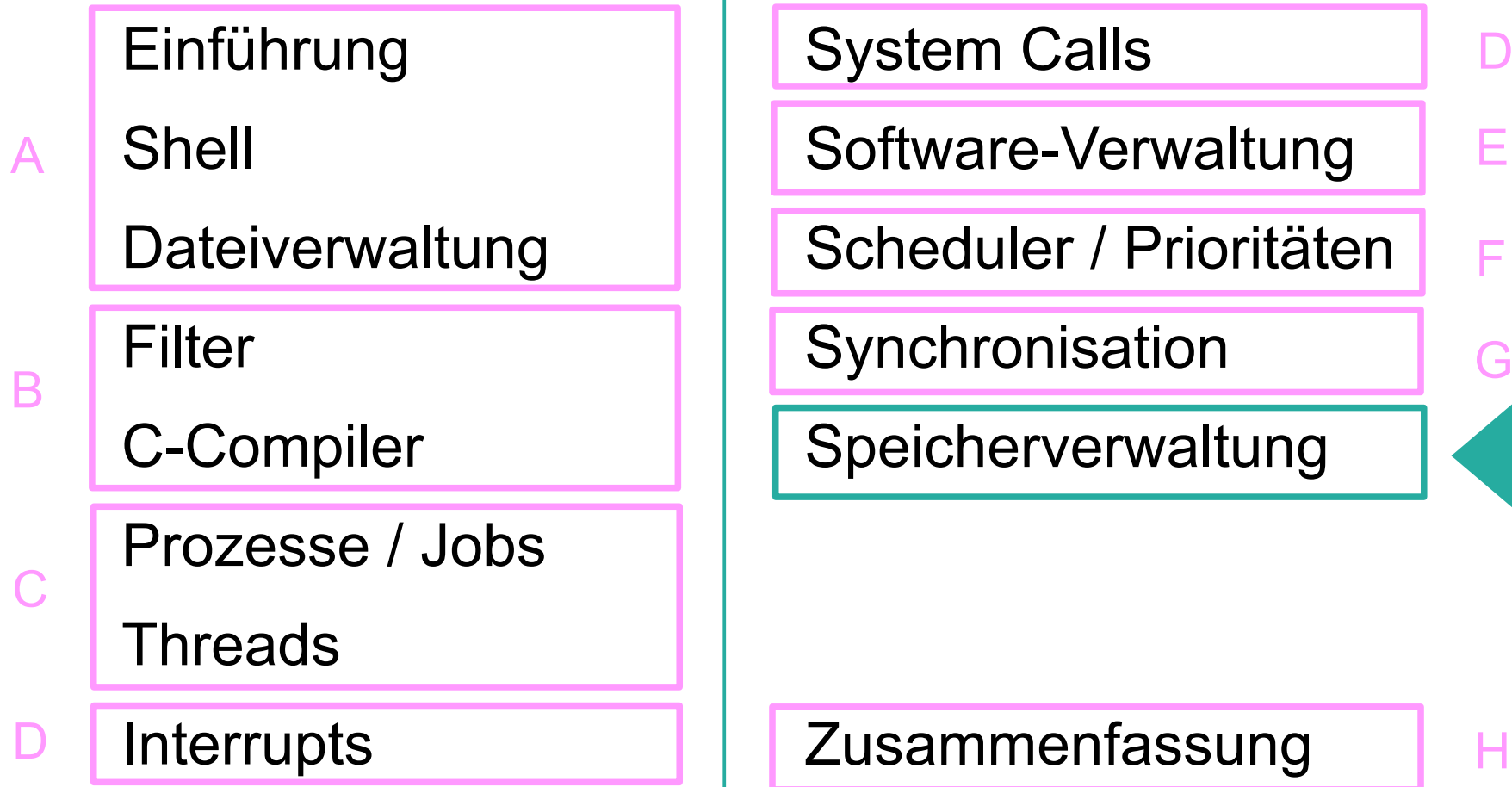
Dipl.-Math., Dipl.-Inform.

**Foliensatz S:**

- Speicherverwaltung

v1.0, 2015/04/14

# Übersicht: BS Praxis und BS Theorie



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

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.

```



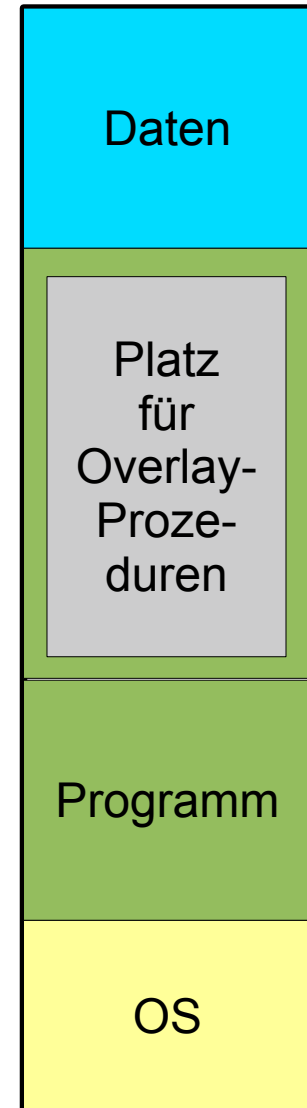
projekt.001



projekt.002



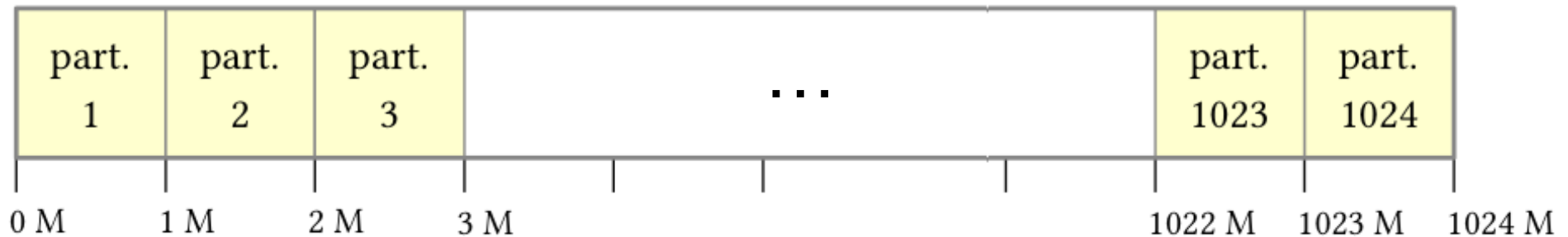
projekt.com



- 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)

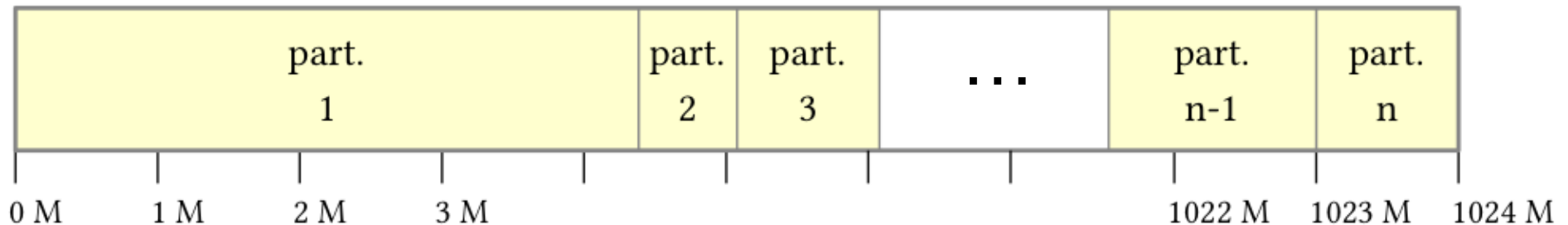
# **Klassische Verfahren (zusammenhängende Zuteilung)**

- Partitionen fester, gleicher Größe



- unflexibel: Anzahl und Größe fix

- Partitionen fester, verschiedener Größe



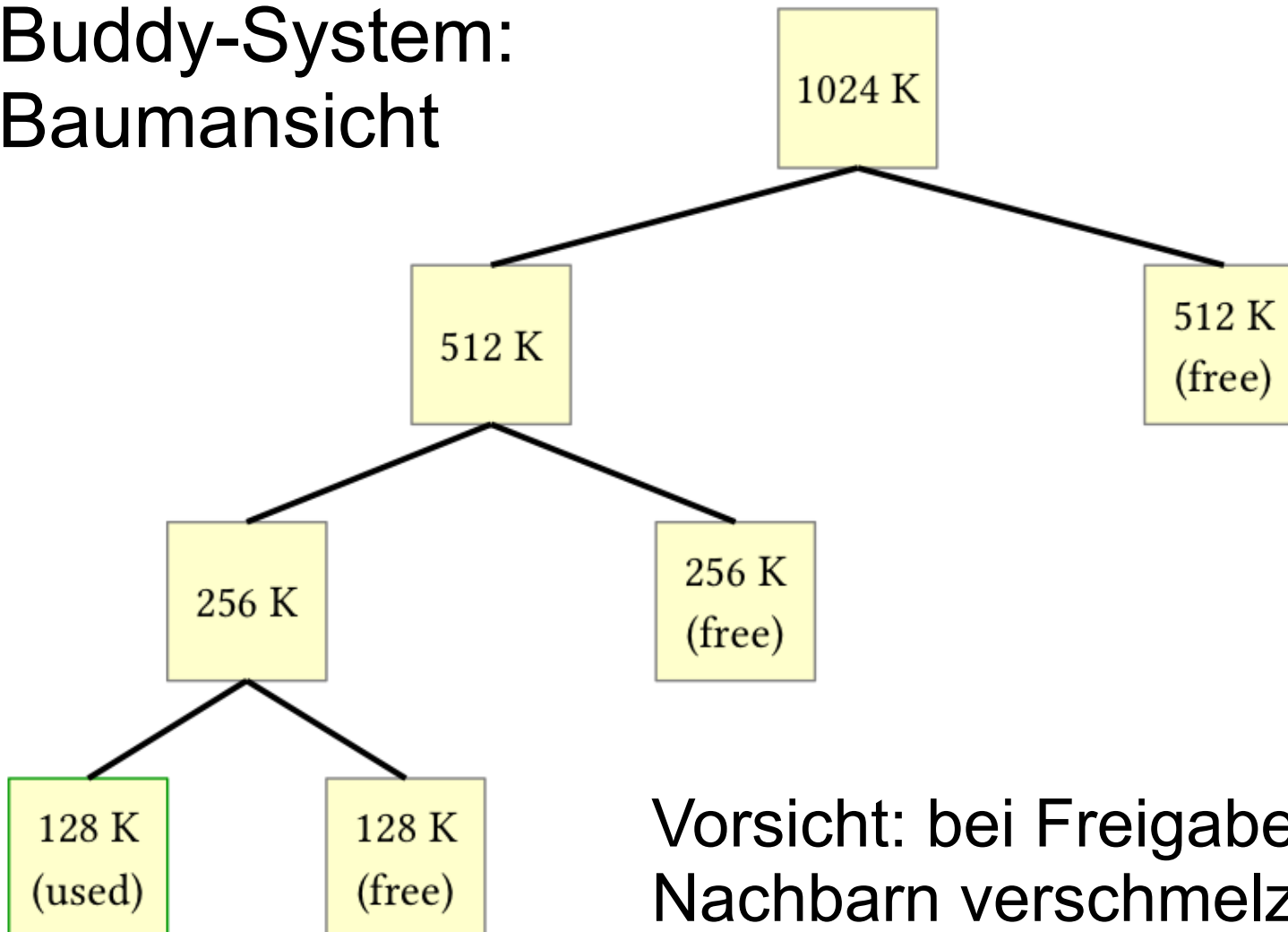
- nur minimal flexibler
- Aufteilung wird bei System-Initialisierung festgelegt



- 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 ( $\rightarrow$  Rekursion?)
  - Beispiel: 1 MByte, alles frei, Anforderung 90 KByte

1024 KB			
512 KB		512 KB	
256 KB	256 KB	512 KB	
<b>128 KB</b>	128 KB	256 KB	512 KB

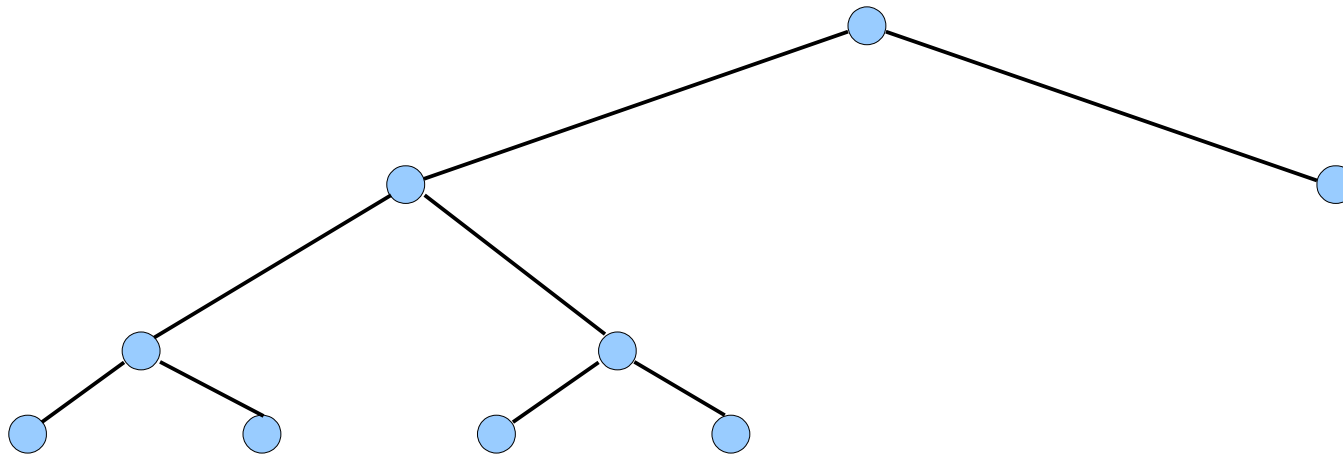
- Buddy-System:  
Baumansicht



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

# Zusammenhängende Zuteilung (5)

- Buddy-System: unmögliche Verschmelzung

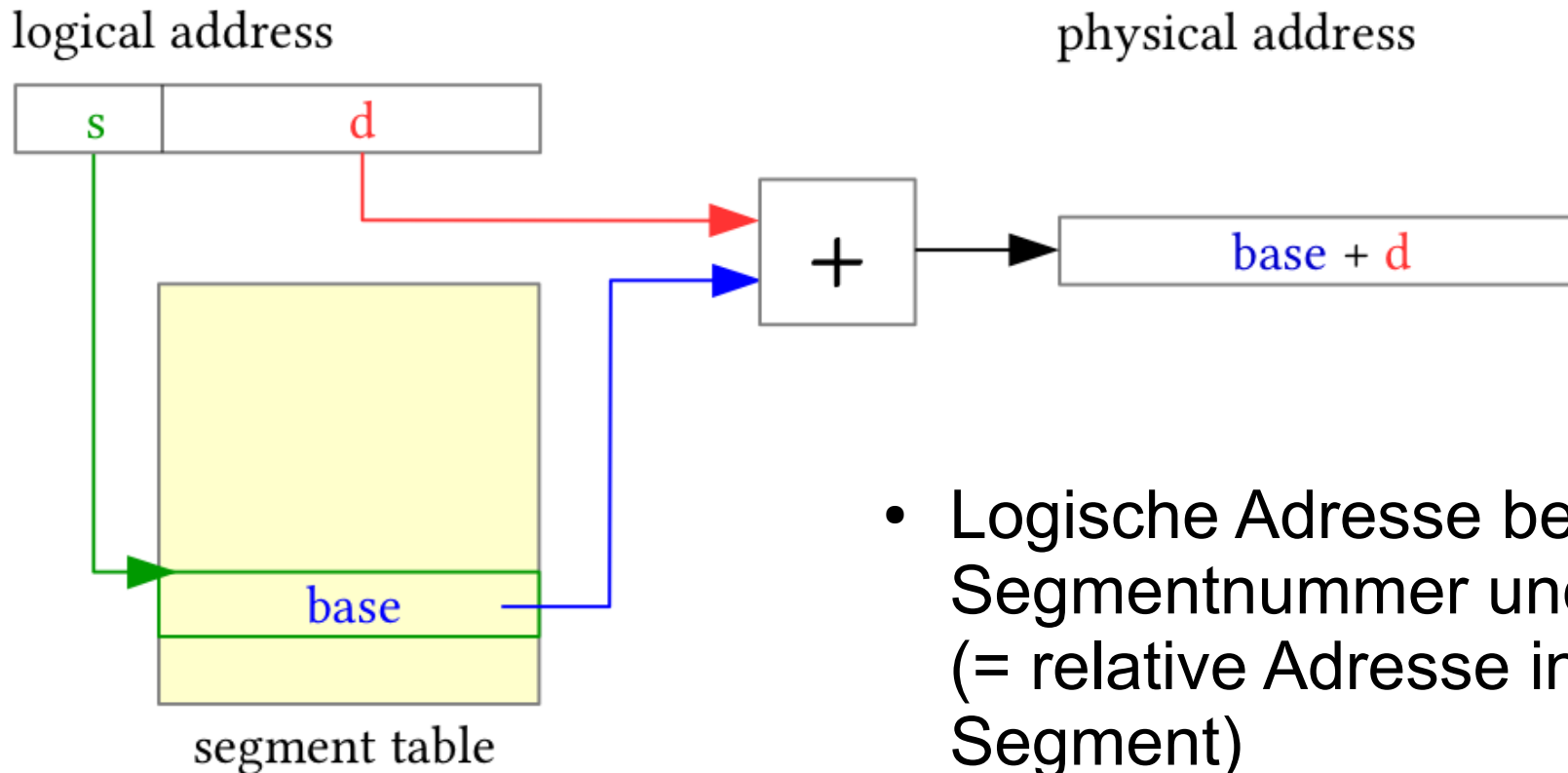


128 KB	128 KB	128 KB	128 KB	512 KB
128 KB	<del>256 KB</del>		128 KB	512 KB

!

# Segmentierung (1)

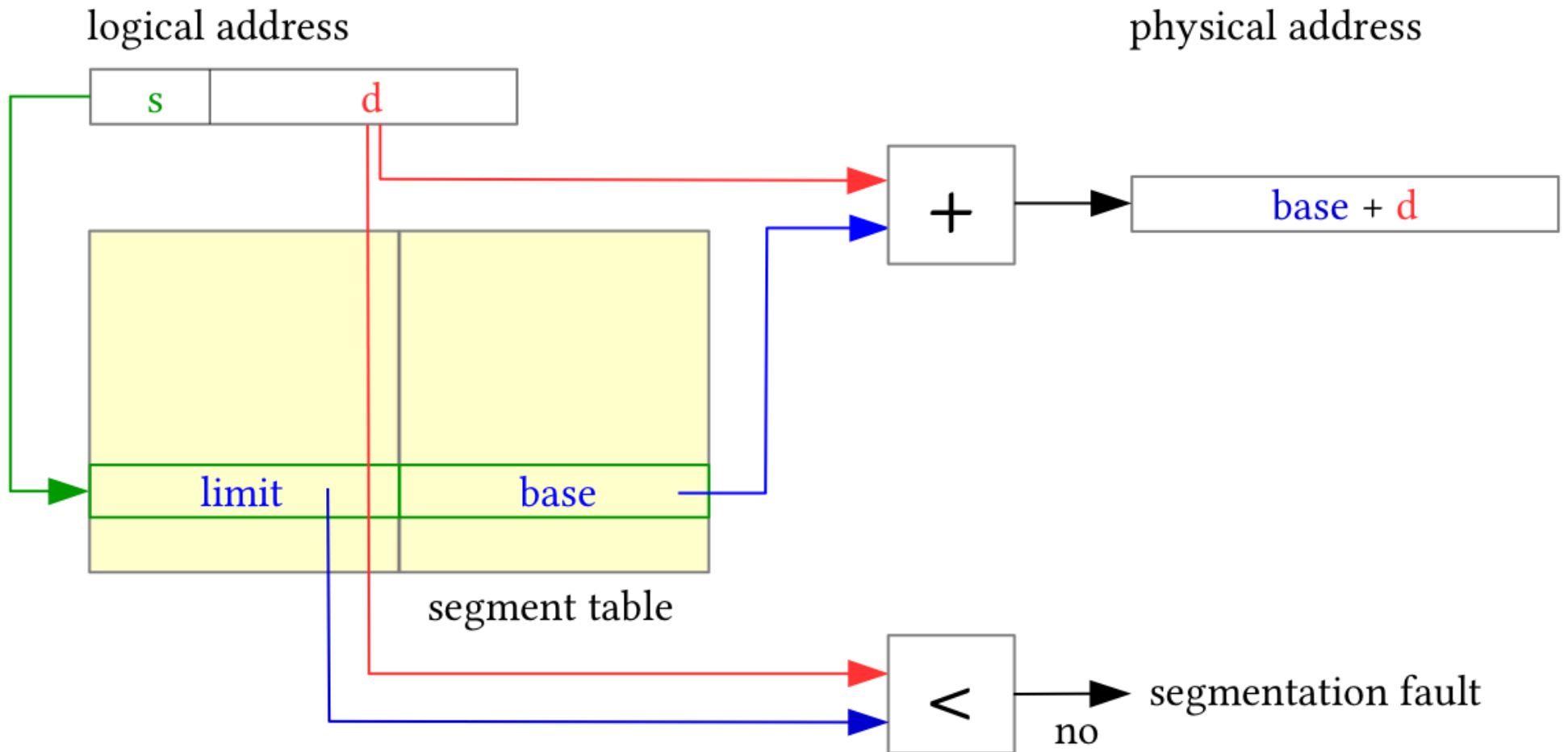
- Über eine Segment-Tabelle wird Speicher in Segmente (zusammenhängende Bereiche) aufgeteilt



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

# Segmentierung (2)

- Angabe einer Segmentgröße → Prüfung bei Zugriff



# Modernes Verfahren (Paging)

- 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

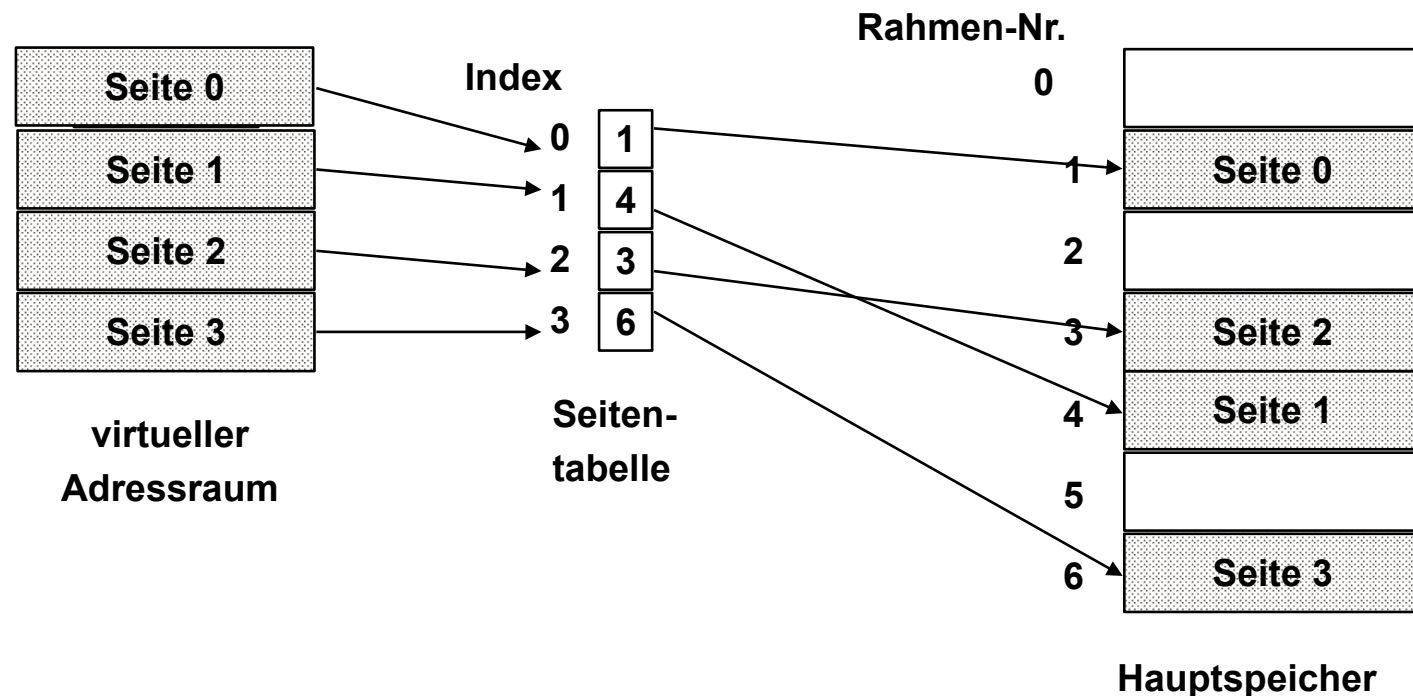
- Aufteilung des Adressraums in **Seiten (pages)** fester Größe und des Hauptspeichers in **Seitenrahmen (page frames)** gleicher Größe.
  - Typische Seitengrößen: 512 Byte bis 8192 Byte (immer Zweierpotenz).
- Der lineare, zusammenhängende Adressraum eines Prozesses („**virtueller**“ **Adressraum**)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



- 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)

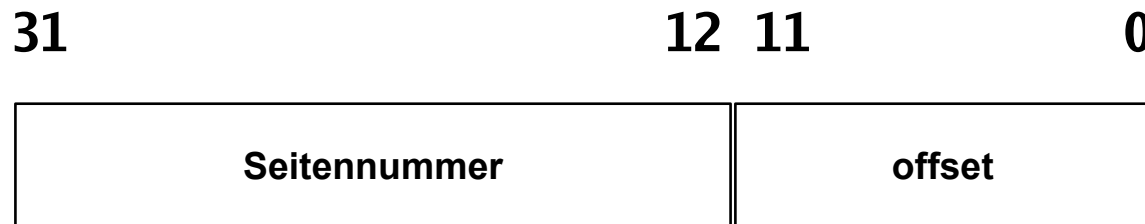
- Paging stellt den Zusammenhang zwischen Programmadresse und physikalischer Hauptspeicheradresse erst zur Laufzeit mit Hilfe der Seitentabellen her.



- 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.

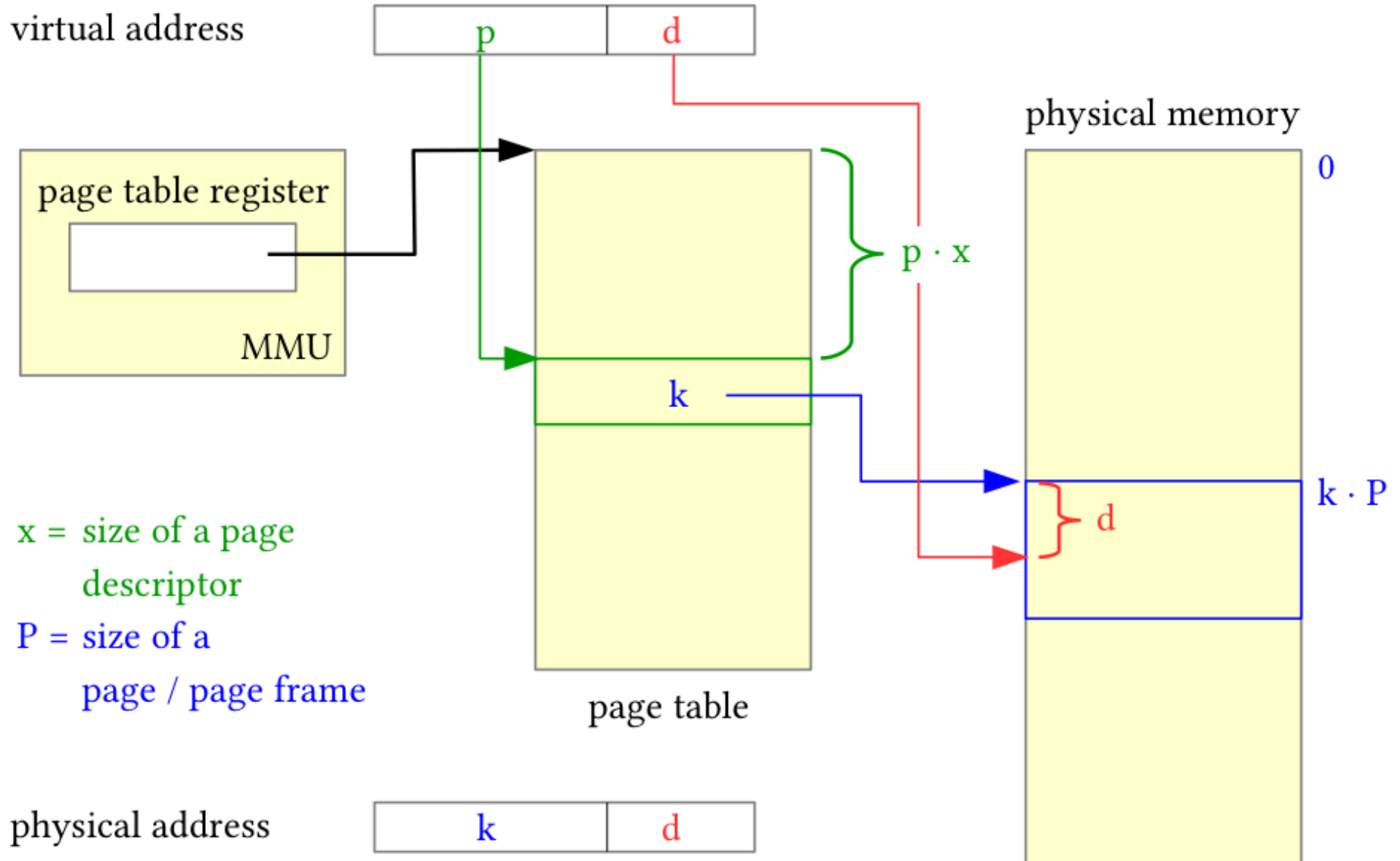
- Die Programmadresse wird in zwei Teile aufgeteilt:
  - eine Seitennummer
  - eine relative Adresse (offset) in der Seite

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



- 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.
- Die Seitennummer wird als Index in die Seitentabelle verwendet.

# Adressübersetzung beim Paging (3)



- 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.

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

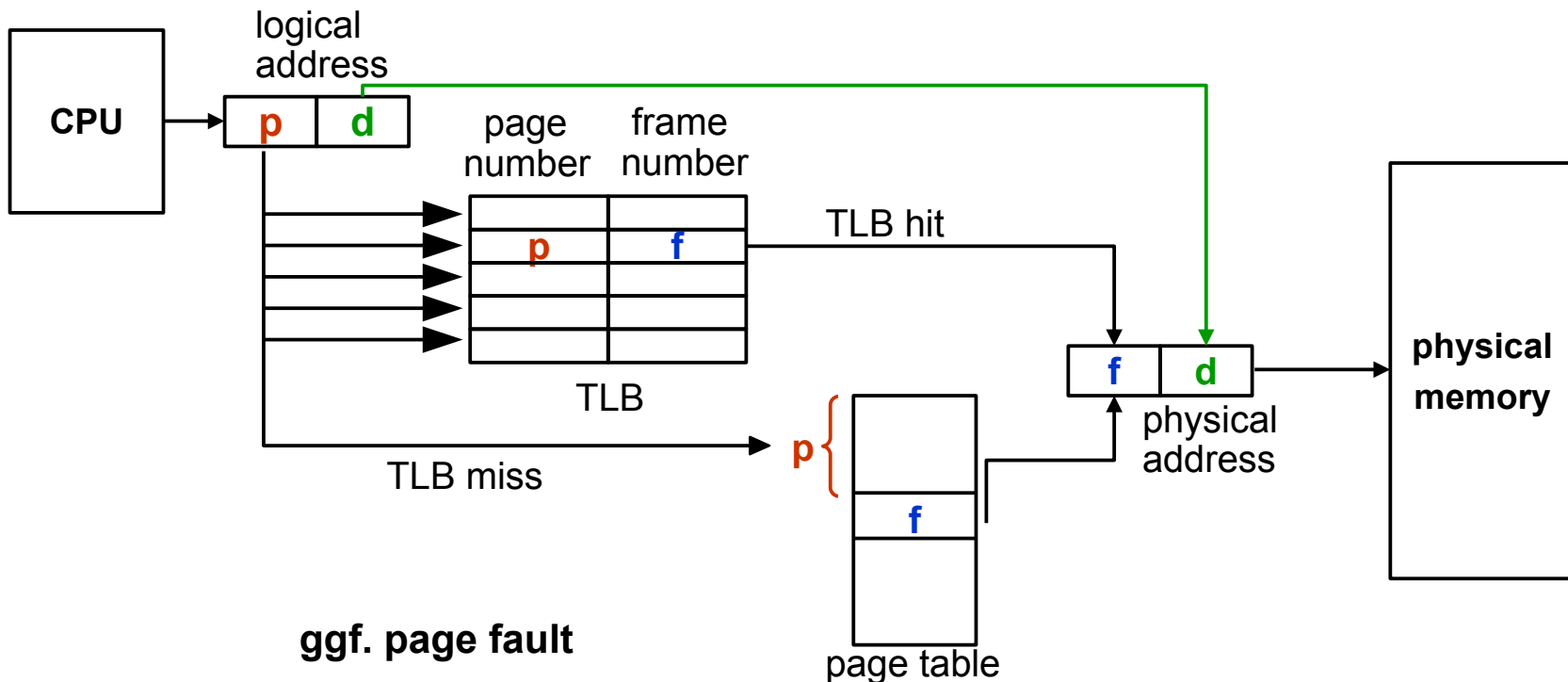


- allgemeiner Vorgang:
  - Nur Teile des Prozesses befinden sich im physikalischen Speicher
  - falls Zugriff auf eine Adresse, die ausgelagert ist:
    - BS setzt den Prozess auf blockiert
    - BS setzt eine Disk-I/O-Leseanfrage ab
    - Nach Laden der fehlenden Seite wird ein I/O-Interrupt erzeugt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

- **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.



- 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

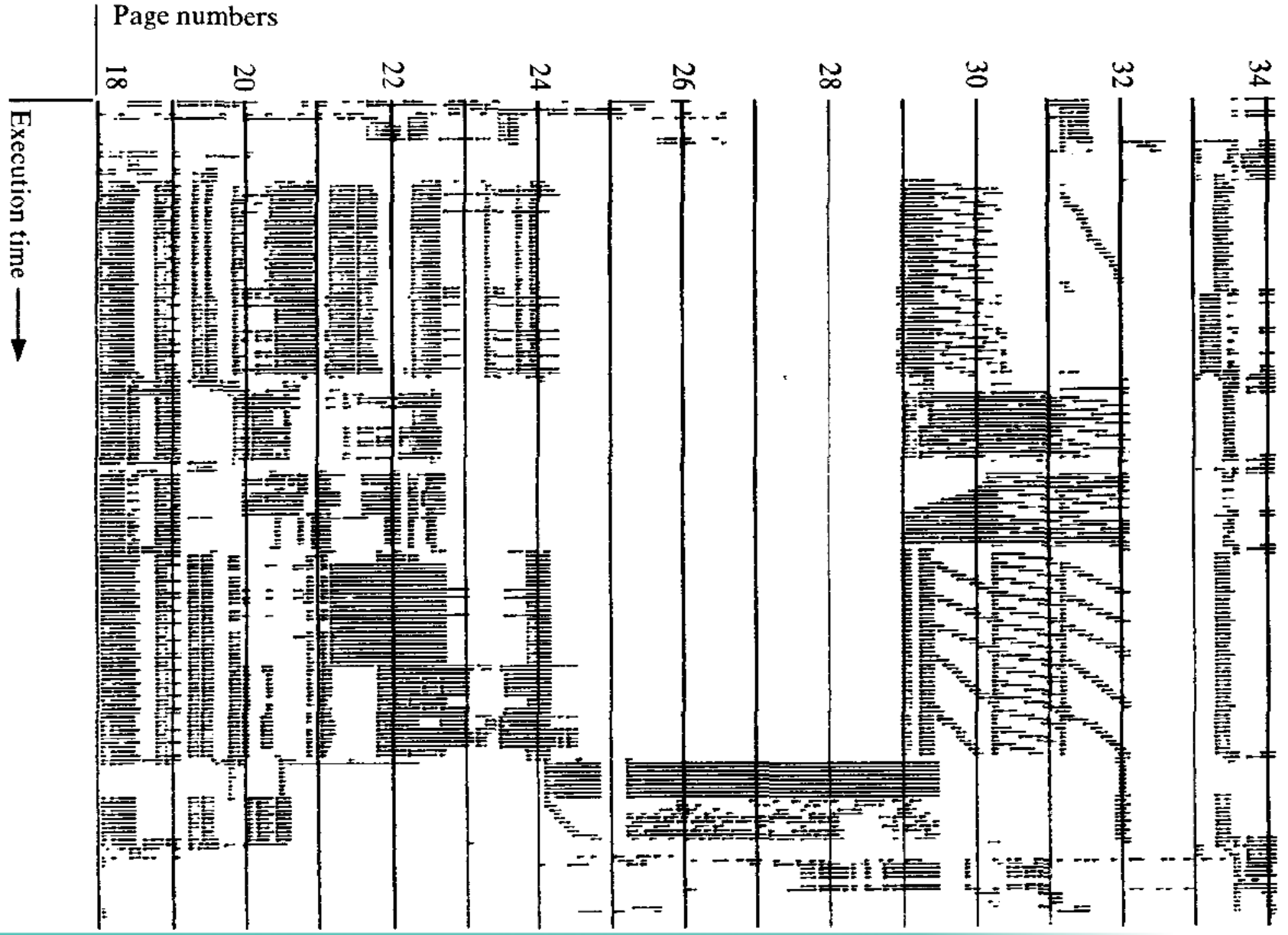


Bild: Hatfield (1972)

- Inhalt des TLB ist prozess-spezifisch!  
Zwei Möglichkeiten:
  - Jeder Eintrag enthält ein „valid bit“. Bei Prozesswechsel (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.

## 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

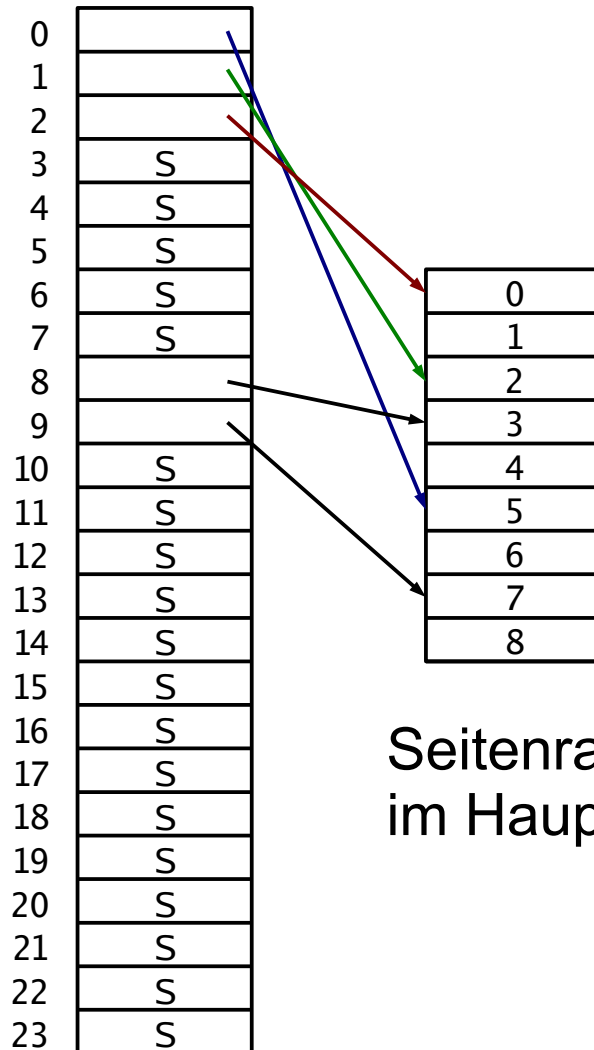
- Bei großem virtuellen Speicher sehr viele Einträge in der Seitentabelle nötig, z.B.  $2^{32}$  Byte Adressraum, 4 KByte/Seite → über 1 Millionen Seiteneinträge, also Seitentabelle > 4 MByte (pro Prozess!)
- Platz sparen durch invertierte Seitentabellen:
  - normal: ein Eintrag pro (virtueller) Seite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



- Problem: Suche zu Prozess  $p$  und seiner Seite  $n$  nach dem Eintrag  $(p,n)$  in der invertierten Tabelle  $\rightarrow$  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 ( $\rightarrow$  gleich)

# Invertierte Seitentabellen (3)

Seitentabelle



Invertierte  
Seitentabelle

0	2
1	-
2	1
3	8
4	-
5	0
6	-
7	9
8	-

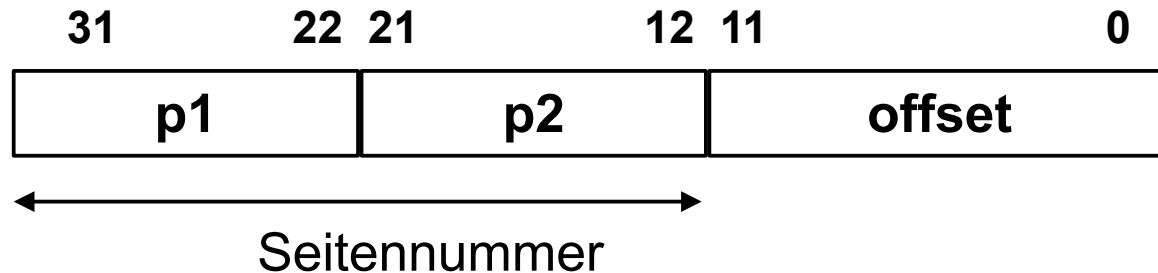
Die Seitentabelle kann sehr groß werden.

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

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

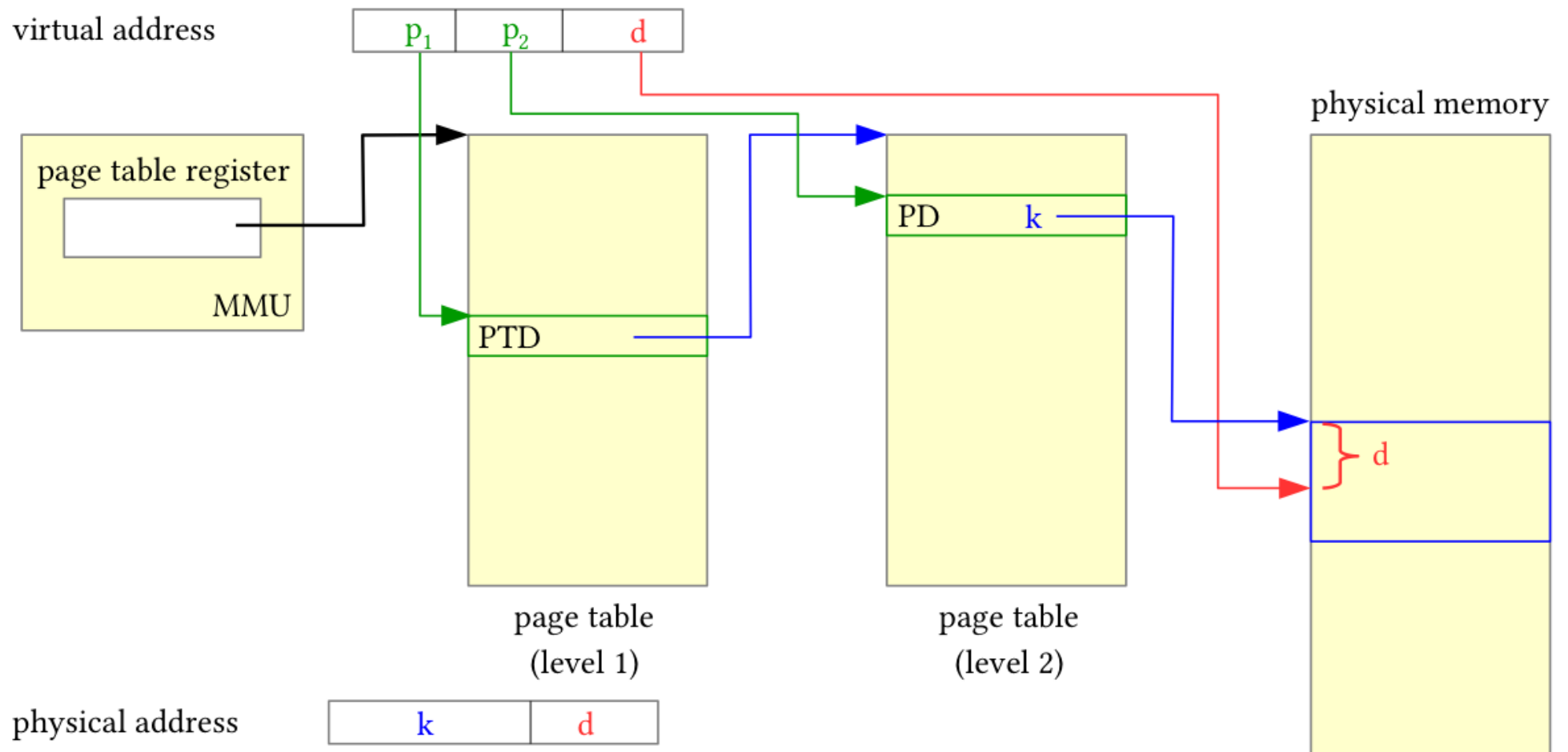
# Mehrstufiges Paging (2)

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

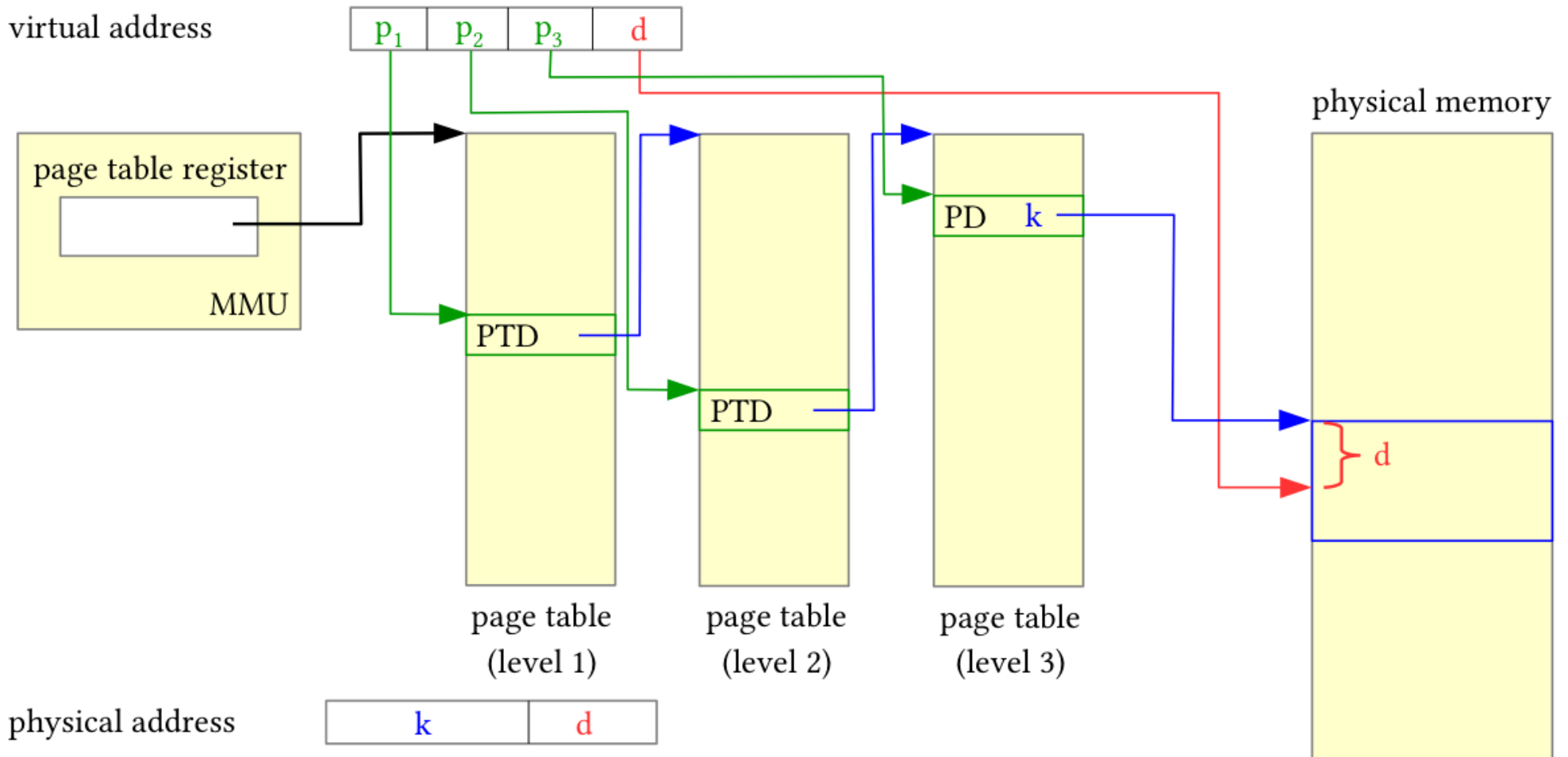


- $p_1$ : Index in **äußere Seitentabelle**, deren Einträge jeweils auf eine **innere Seitentabelle** zeigen
  - $p_2$ : Index in eine der inneren Seitentabellen, deren Einträge auf Seitenrahmen im Speicher zeigen
  - Die inneren Seitentabellen müssen nicht alle speicherresident sein
- Analog dreistufiges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

## Adressübersetzung bei zweistufigem Paging:



## Adressübersetzung bei dreistufigem Paging:



- Größe der Seitentabellen:

Beispiel:

$p_1$	$p_2$	offset
10	10	12

- 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.

- 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 ( $p_1, p_2, \dots$ ).



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) =  $2^5 \times 2^{10}$  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  $64 \text{ M} / 32 \text{ K} = 2 \text{ K} = 2048$  ( $2^{11}$ ) Seitenrahmen  
Ein Eintrag in der Seitentabelle benötigt darum 11 Bit, in der Praxis 2 Byte.

→ Platzbedarf:

$$\begin{aligned} & \#(\text{virt. Seiten}) \times \text{Größe}(\text{Eintrag}) \\ &= 2^{17} \times 2 \text{ Byte} = 2^{18} \text{ Byte} = \underline{256 \text{ KB}} \end{aligned}$$

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) =  $2^4 \times 2^{10}$  Byte  
=  $2^{14}$  Byte,  
d.h.: Offset ist 14 Bit lang



Seitennummer

Offset

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:

$$\begin{aligned} & \#(\text{Einträge}) \times \text{Größe}(\text{Eintrag}) \\ & = 2^6 \times 4 \text{ Byte} = 2^8 \text{ Byte} = 256 \text{ Byte} \end{aligned}$$

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

# Demand Paging

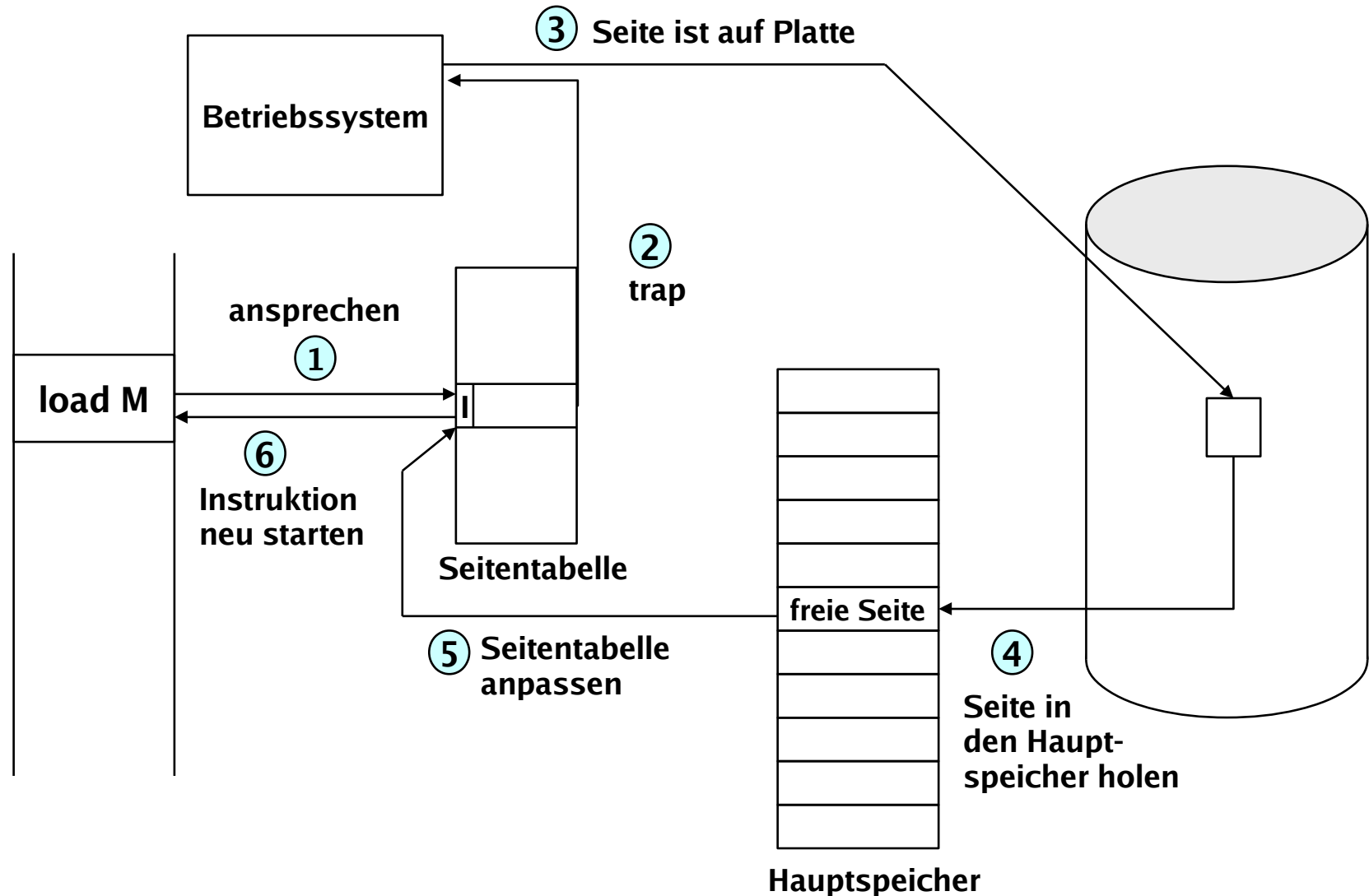
- Der Adressbereich eines Prozesses muss nicht vollständig im Hauptspeicher sein.
  - Das Lokalitätsprinzip besagt, dass ein Prozess in einer Zeitspanne nur relativ wenige, nahe beieinanderliegende Adressen anspricht.
  - Teile des Programms werden bei einem bestimmten Ablauf möglicherweise gar nicht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

- **Demand Paging** 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 Hauptspeicher, somit können mehr Prozesse gleichzeitig aktiv sein.

- Jeder Eintrag in der Seitentabelle enthält ein **valid 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.

- 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





- 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.

- 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**).

- 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.