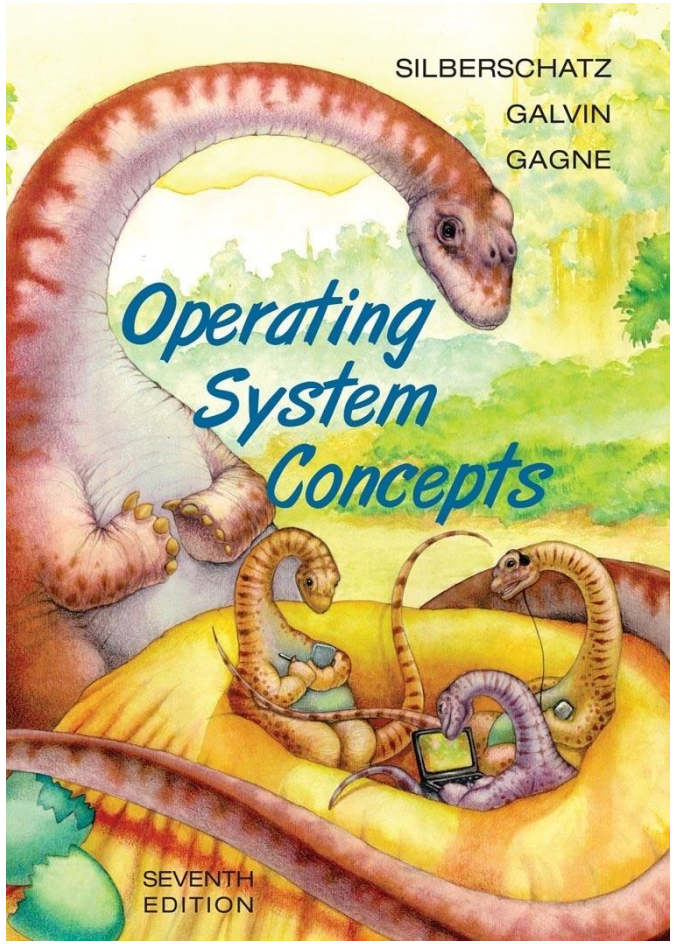
4. Speicherverwaltung und Dateisystem

Eine ausgereifte Speicherverwaltung ist von zentraler Bedeutung für die **effiziente Kooperation** von Prozessen. Darüber hinaus dient sie dem **Schutz gegenüber konkurrierenden Prozessen**.

Literaturhinweis

Dieses Kapitel basiert stark auf dem Buch „Operating Systems Concepts“ von Silberschatz et al.

Eigenständiges „Studieren“ dieses Buches, das auch in anderen Bereichen hervorragend ist, wird empfohlen, ist aber nicht prüfungsrelevant.



4.1. Grundlegende Betrachtung

Die wichtigsten Aufgaben der **Speicherverwaltung** sind:

- **Bereitstellung von Information** über die aktuelle Speicherbelegung
- **Vergabe von Speicherplatz** an Prozesse
- **Entzug von** nicht mehr benötigtem **Speicherplatz**
- **Verwaltung der Speicherhierarchie**
  Ein-/Auslagerung von Daten zwischen Hauptspeicher und Sekundärspeicher (Festplatte, Band, ...): „**Swapping**“ bzw. „**Paging**“

Anmerkungen:

- Die Speicherverwaltung ist eine der zentralen Aufgaben des **Betriebssystems**. Sie wird in **enger Kooperation mit der Hardware** wahrgenommen.
- Verwaltung ist **für alle Arten von Speichermedien** erforderlich (RAM, Festplatte, ...).
- Die Mehrzahl der nachfolgenden Betrachtungen zu Prozessen gilt **analog auch für Threads**.

Das **Dateisystem** ist dagegen zuständig für die **langfristige Datenhaltung**:

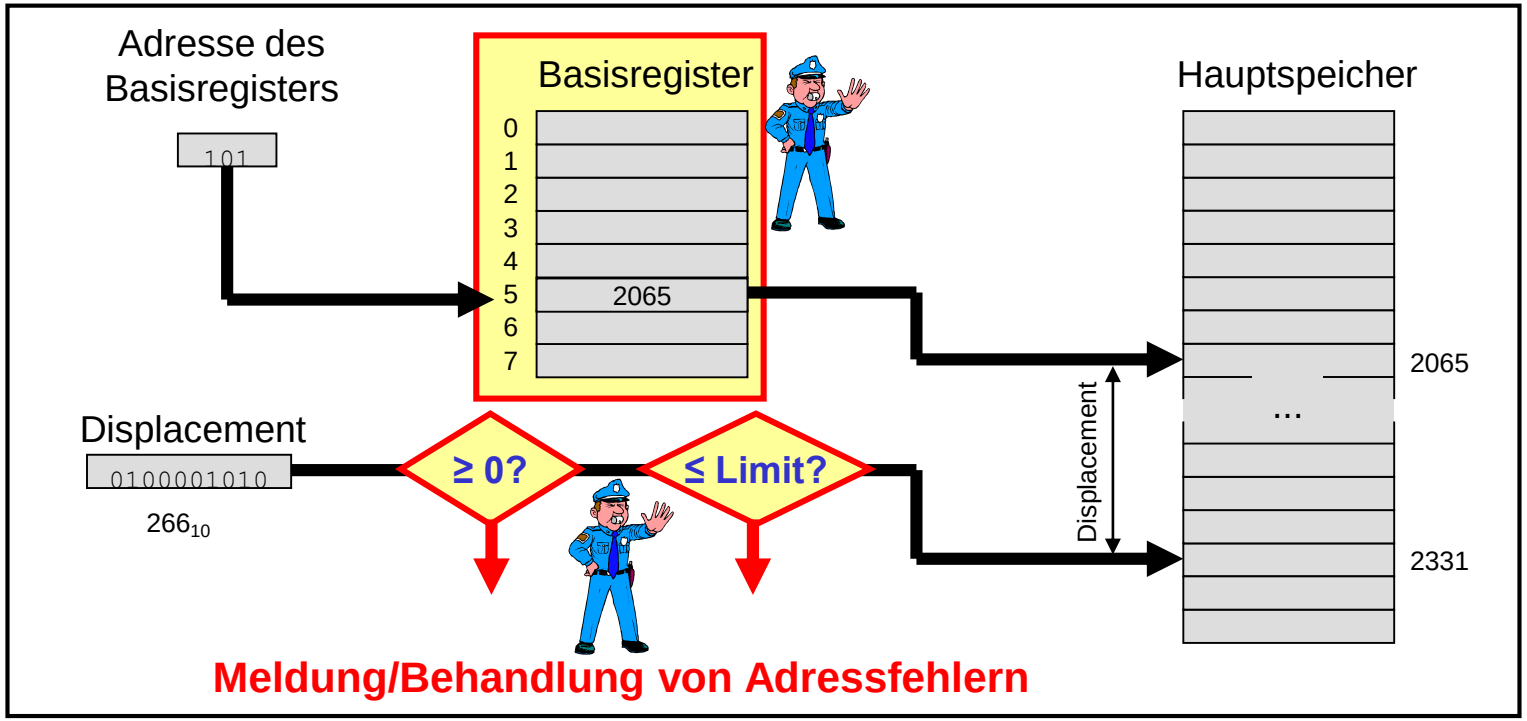
- **Speicherung sehr großer Mengen von Information**
- **Sicherstellung von Persistenz** (Datenhaltung über die Lebensdauer der Prozesse hinaus)
- **Gleichzeitiger / Koordinierter Zugriff auf Daten / Dateien**

Schutz vor Zugriff auf fremde Speicherbereiche

Wir hatten bereits gesehen, dass ein lauffähiges Programm vom Lader in den Speicher platziert wird.

Ein Schutz vor (beabsichtigten oder unbeabsichtigten) Zugriffen auf fremde Speicherbereiche kann gewährleistet werden, wenn

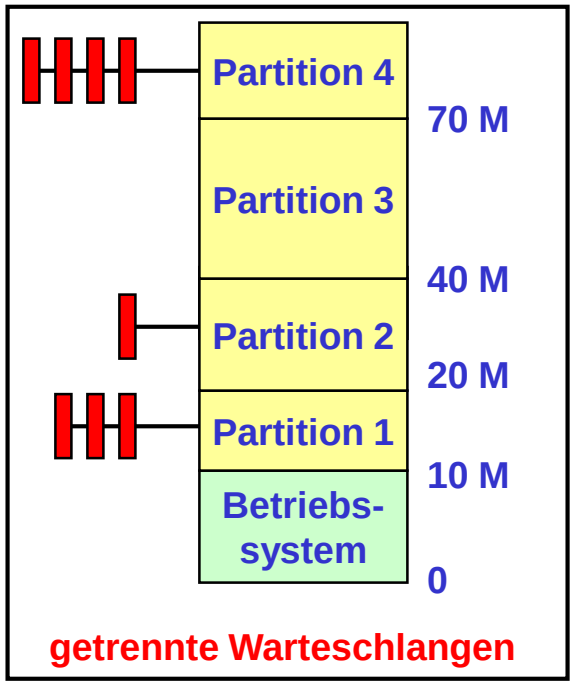
- **das Basisregister vor Modifikation geschützt** und
- **das Displacement überprüft** wird (≥ 0, ≤ „Limit“ mit geeignetem „Limit“)

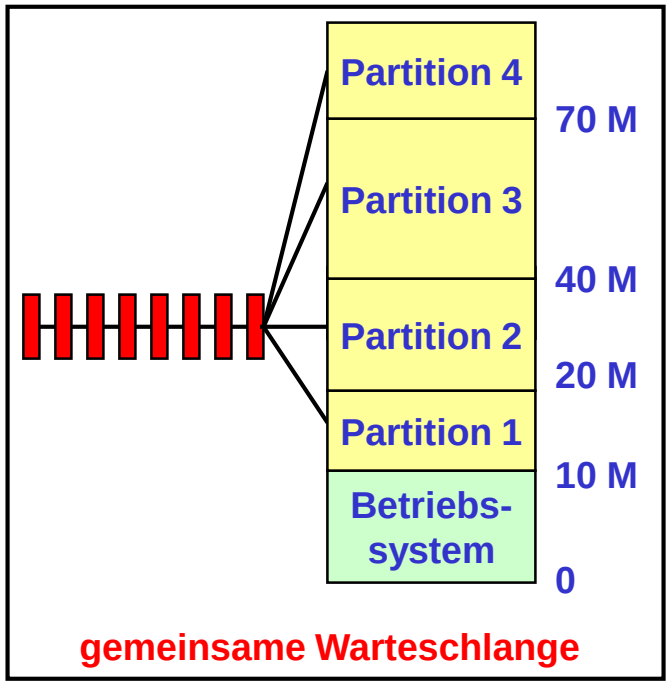


4.2. Multiprogramming mit festen Partitionen

Sollen mehrere Prozesse gleichzeitig bearbeitbar sein, dann kann der Systemadministrator den Speicher in Bereiche („**Partitionen**“) aufteilen.

➡ **Lauffähigen Prozessen** werden (nach Bedarf) geeignete Partitionen **zugeordnet**.





Anmerkungen:

- Die Festlegung geeigneter **Größen** für die Partitionen ist **nicht trivial** !
- Bei getrennten Warteschlangen können **Wartezeiten** auftreten, **obwohl** eigentlich hinreichend viel **Speicherplatz** für einen weiteren Prozess **verfügbar** ist !
- **Zugriffe auf „fremde“ Partitionen** müssen wirksam **verhindert** werden: Schutz vor Fehlern und Attacken.

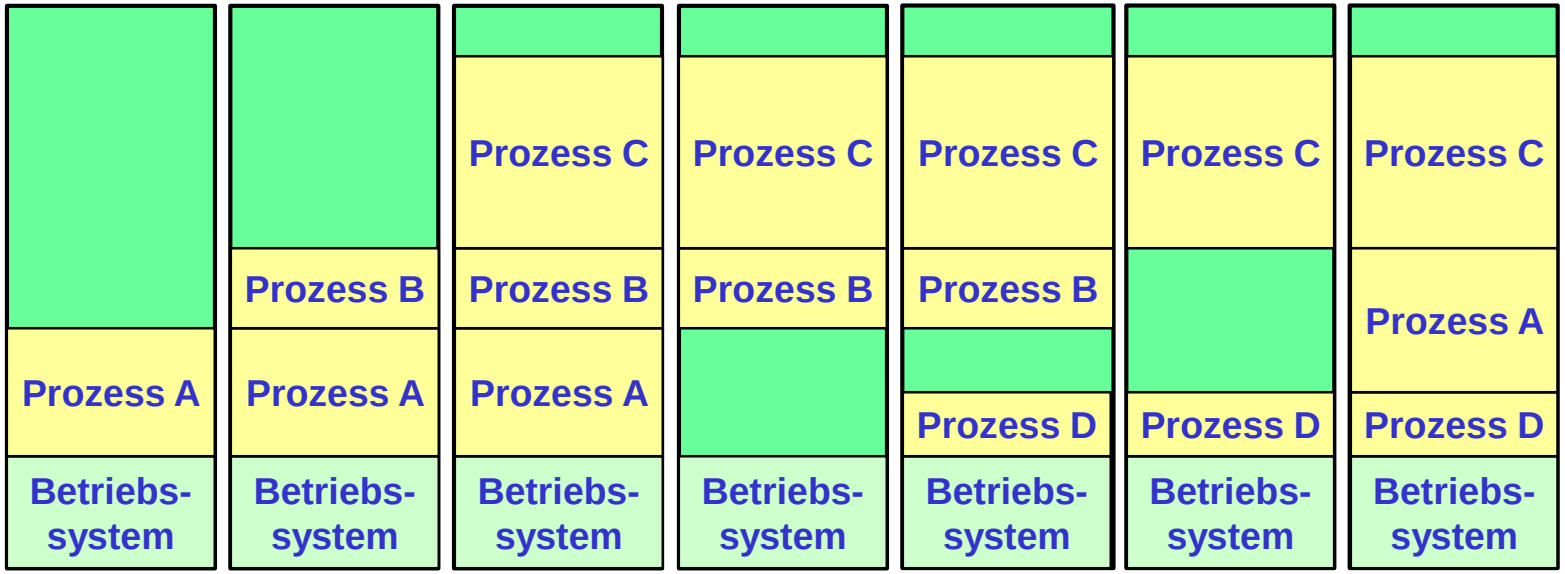
Mögliche Varianten bei gemeinsamer Schlange:

- Auswahl der Anforderung gemäß **FIFO**,
- Auswahl der **am besten passenden Anforderung**,
- Auswahl nach **Prioritäten der Prozesse**,
- ...

4.3. Multiprogramming mit variablen Partitionen

Variable Partitionen verursachen einen deutlich **höheren Verwaltungsaufwand**.
Sie gestatten aber häufig eine **bessere Speicherplatzauslastung**.

Beispiel:



Problem bei häufigem Prozesswechsel:

➡ Zerstückelung des Arbeitsspeichers (**Fragmentation**)

Bei fragmentiertem Speicher kann ggf. „umgepackt“ (= verschoben) werden.

Achtung:

- Die Bearbeitung der betroffenen Prozesse muss für diese Zeit unterbrochen werden.
- Verschiebung im Speicher kostet relativ viel Zeit (während der Prozess aktiv sein sollte!).
- Verschieben wird durch Adressierung mit Basisregister und Displacement erleichtert.

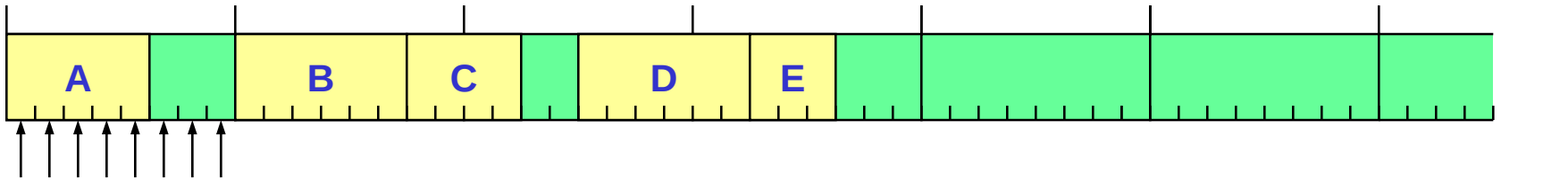
Speicherplatzverwaltung: Bitmaps und Verkettete Listen

Bitmaps:

- Teile den Speicher in **Bereiche gleicher Länge** (wie lang ?)
- Kennzeichne diese Teile jeweils mit **0 bzw. 1 als „frei“ bzw. „belegt“.**
- Update: trivial

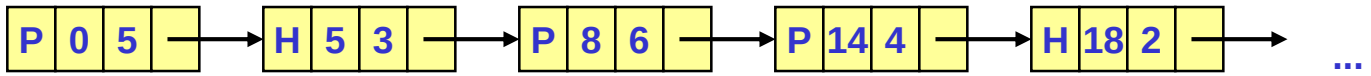
Verkettete Listen:

- Verwalte die Speicherbelegung als verkettete Liste mit
  - **H** (= Hole, Lücke) bzw.
  - **P** (= Process)
- Update: **Einfügen/Entfernen von Listenelementen**



1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 1
1 1 0 0 1 1 1 1
...

Bitmap



Verkettete Liste

Algorithmen zur Vergabe von Speicherplatz

1. First Fit:

- **Beginne** die Suche **mit Adresse 0.**
- **Wähle den ersten Bereich**, der groß genug ist.

2. Next Fit:

- **Beginne** die Suche **bei der aktuellen Position**.
- **Wähle den ersten Bereich**, der groß genug ist.

3. Best Fit:

- Wähle unter allen freien Bereichen den aus, in den der Prozess **am besten passt.**
- Vorteil: **Nutzung des kleinst möglichen Bereiches**
- Nachteil: Lange Suchzeit, **schlechte Speichernutzung** (kleine, nutzlose Lücken)

4. Worst Fit:

- **Wähle** unter allen freien Bereichen **den größten**.
- Vorteil: **Vermeidung** kleiner, **nutzloser Bereiche**
- Nachteil: **Konsequente Vernichtung großer Bereiche**
  (Problematisch für Prozesse mit großem Bedarf an Speicherplatz)

4.4. Swapping

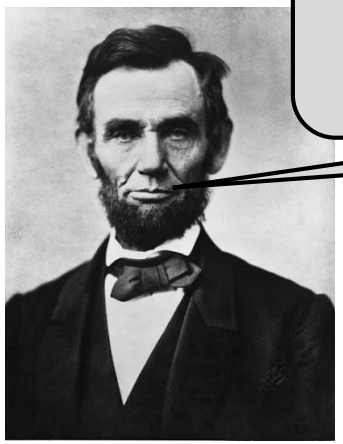
Ein Prozess ist nur lauffähig, wenn ihm Platz im Hauptspeicher zugeordnet ist.

Wenn aber

- der **Hauptspeicher zu klein** ist (… um alle lauffähigen Prozesse dort zu halten) und/oder
- **Scheduling mit Unterbrechung** (Preemption) eingesetzt wird und/oder
- **längere Zeit auf E/A-Operationen** wartet wird und während dieser Zeit der Hauptspeicher für andere Zwecke genutzt werden soll

dann kann das sog. „Swapping“ eingesetzt werden:

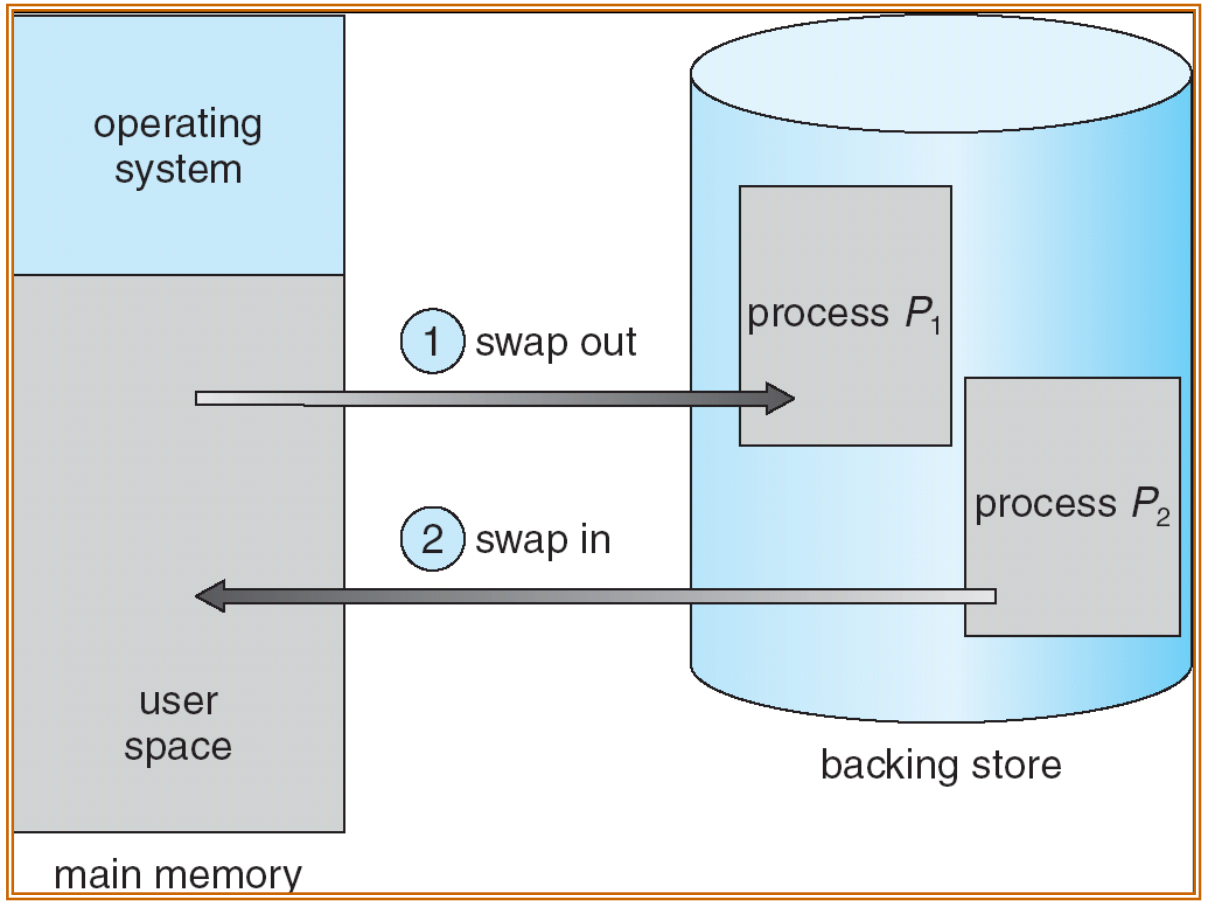
Beim **Swapping** werden **Prozesse komplett in den Hintergrundspeicher** verlagert und nach einer gewissen Zeit in den Hauptspeicher zurück geholt.



"I have not permitted myself, gentlemen, to conclude that I am the best man in the country; but I am reminded in this connexion of a story of an old Dutch farmer, who remarked to a companion once that it was not best to swap horses when crossing a stream.“
Abraham Lincoln, 1864

Foto:
http://hdl.loc.gov/loc.pnp/cph.3a53289
No known restrictions on publication.

Swap Out – Swap In



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 282

4.5. Virtueller Speicher

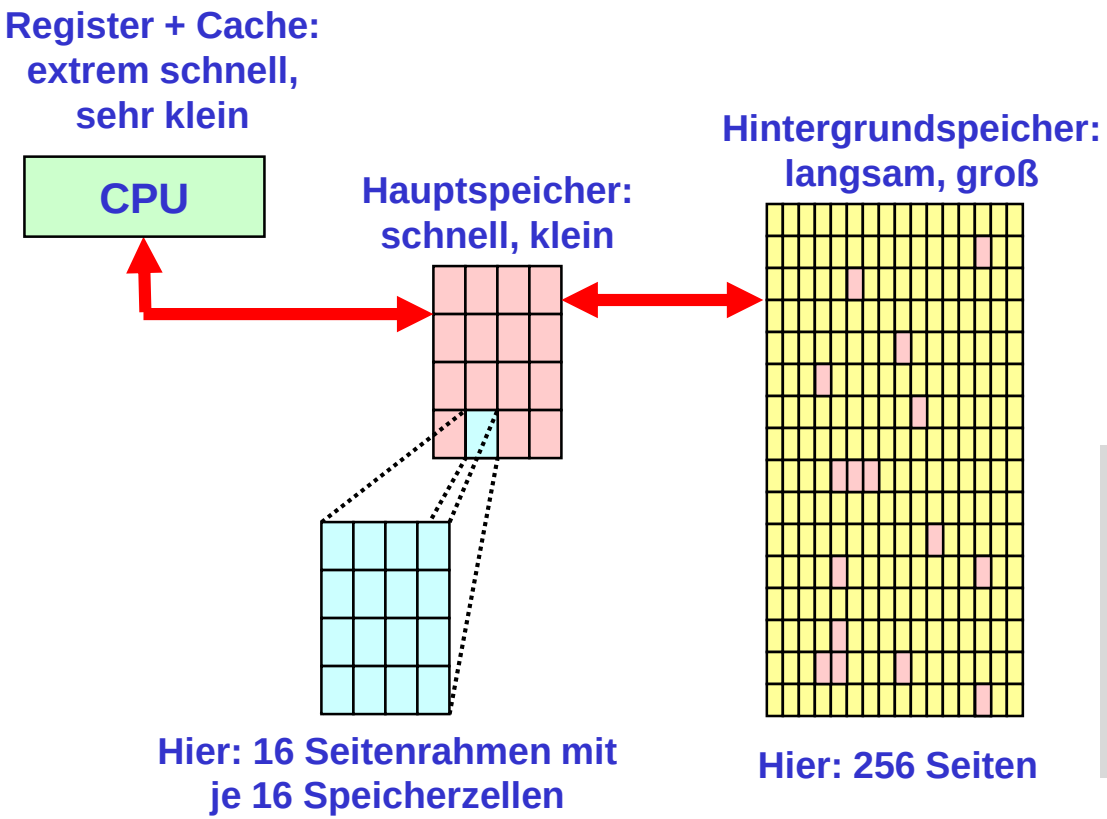
Häufig ist **gar nicht genug Hauptspeicher** vorhanden, um das gesamte Programm und alle Daten aufzunehmen.

Daher wird meist eine **Speicherhierarchie** eingesetzt, mit der die **Illusion eines (fast) beliebig großen Speichers** mit direktem Zugriff erzeugt werden kann.

4.5.1. Grundlegende Betrachtung

Heute ist der sog. „**virtuelle Speicher**“ Bestandteil jedes größeren Computers:

- Der **virtuelle Adressraum** entspricht der logischen (virtuellen) Sicht auf den Speicher.
- Der **Hauptspeicher** enthält aber **sehr viel weniger Speicherworte**.
- **Jeweils aktuell benötigte Daten werden ausgetauscht** zwischen Hauptspeicher und Hintergrundspeicher(n). In der Praxis wird mit Informationseinheiten fester Größe gearbeitet: „**Seiten**“ (Pages). Man spricht von „**Paging**“



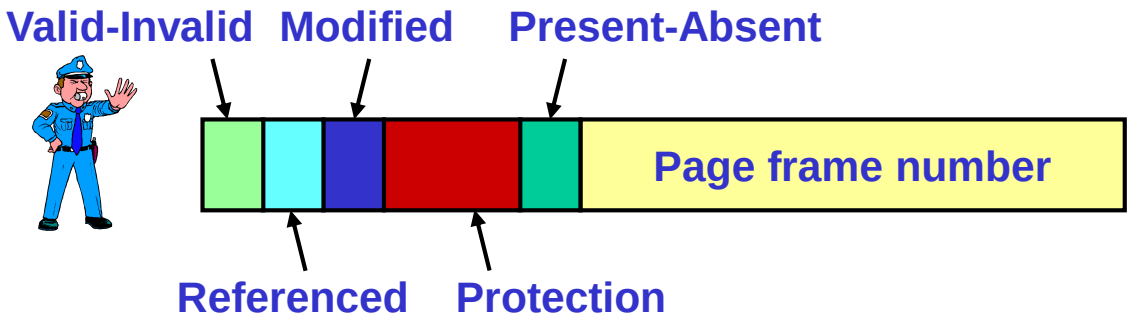
Anmerkungen:

- Die praktische Realisierung wird massiv vereinfacht, wenn die Größen als **Zweierpotenzen** gewählt werden.
- Der logische/virtuelle Adressraum wird meist nicht vollständig auf den Hintergrundspeicher abgebildet.
- Real ist die Speicherhierarchie höher bzw. tiefer als hier dargestellt.

Ein typischer Eintrag in einer Seitentabelle

Die Struktur der Seitentabelle (bzw. ihrer Einträge) ist **stark maschinenabhängig**.

Wir betrachten ein Beispiel* mit Einträgen, die fast immer verwendet werden:



Page frame number (z.B. 32 Bit)
Nummer des Seitenrahmens

Present-Absent (1 Bit)
Ist die Seite aktuell im Hauptspeicher?

Protection (z.B. 1 Bit oder 3 Bits)
Zugriff zulässig für
- Read only / Read-Write (1 Bit reicht aus)
- Lesen/Schreiben/Ausführen (Meist 3 Bit)

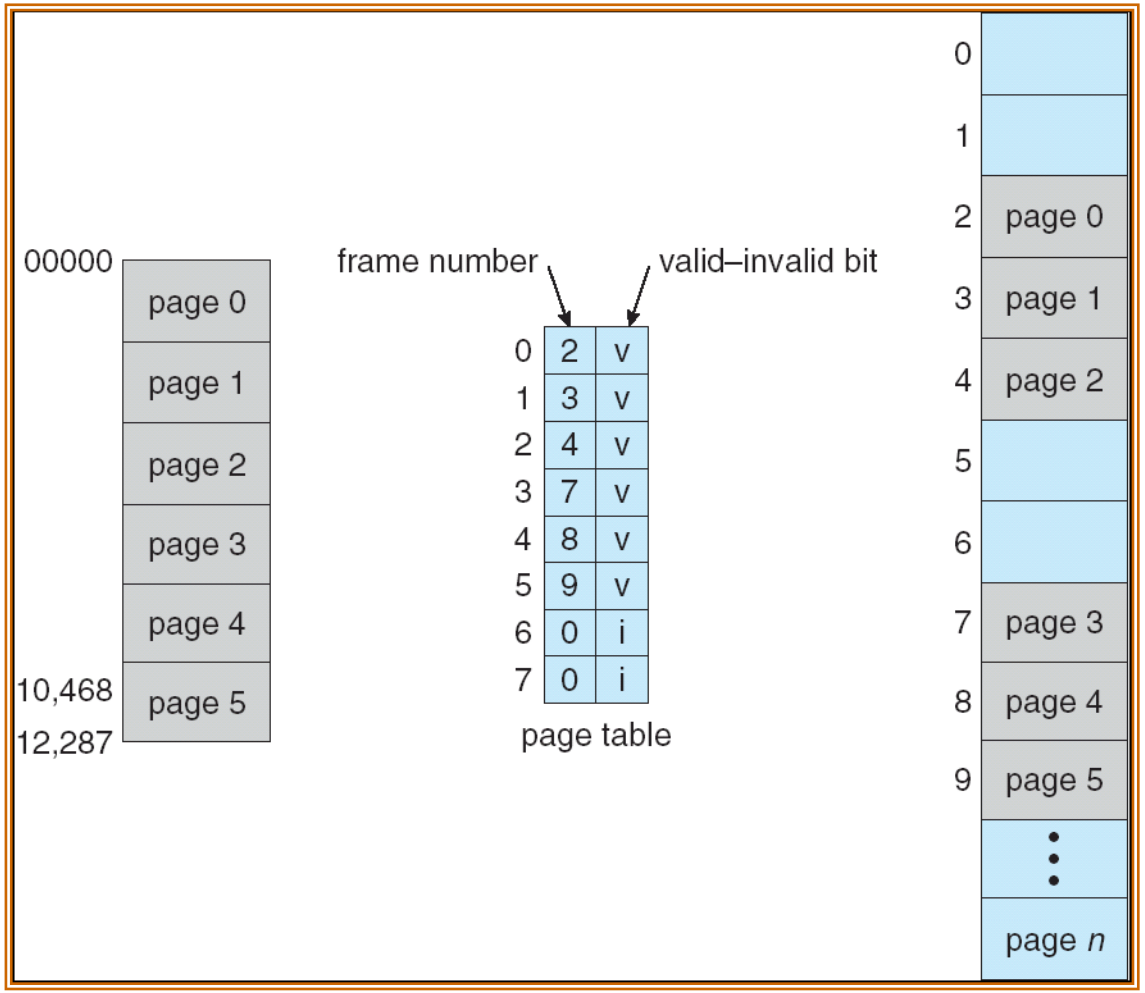
Modified (1 Bit, sog. „dirty bit“)
Wurde die Seite im Hauptspeicher verändert? Falls ja: Nicht einfach verdrängen, sondern im Hintergrundspeicher sichern.

Referenced (1 Bit)
Wurde seit dem Rücksetzen dieses Bits auf die Seite zugegriffen?

Valid-Invalid (1 Bit)
Ist dies überhaupt eine gültige Seitennummer des aktuellen Prozesses ?

*A.S. Tanenbaum, Modern Operating Systems, 3rd Edition, Pearson 2009, p. 191

Beispiel: Valid-Invalid



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 295

4.5.2. Paging

Bei Paging muss die von der CPU gelieferte **logische** (= virtuelle) Adresse auf eine **physische** (= reale) Adresse **umgesetzt** werden, die den Seitenrahmen angibt, in dem sich die Seite aktuell befindet.

Ist die Seite aktuell **nicht im Hauptspeicher, dann muss sie geladen werden**, bevor die Nummer des hierfür genutzten Seitenrahmens geliefert werden kann.

Achtung: **Die Adressumsetzung erfolgt während der Programmausführung !**

Die CPU wartet nicht gerne !

4.5.2.1. Adressumsetzung … formal betrachtet

Der virtuelle Speicher kann beschrieben werden durch ein **Tripel (N, M, f_t)** mit

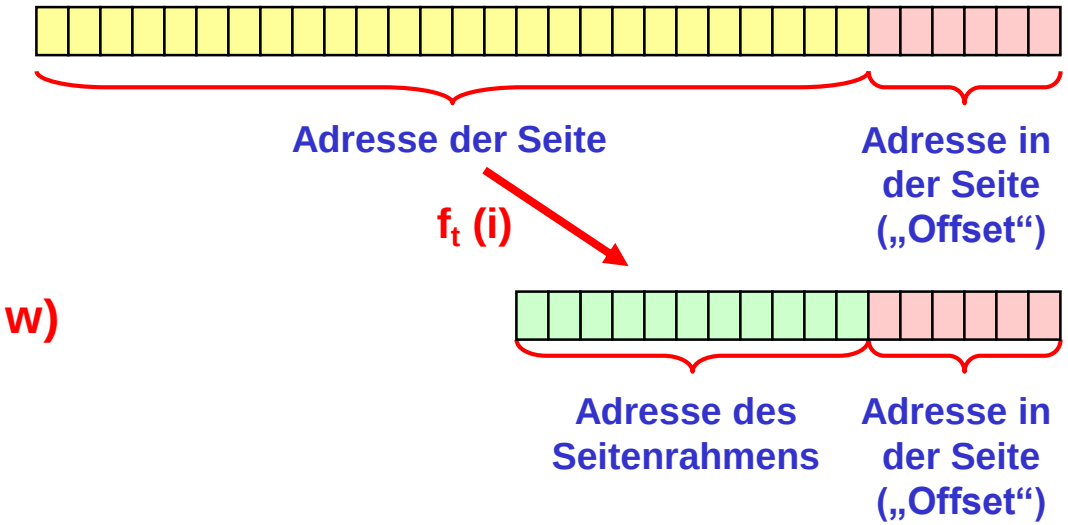
- **$N = \{0, 1, \ldots n-1\}$:** Menge der adressierbaren Seiten (logischer Adressraum)
- **$M = \{0, 1, \ldots m-1\}$:** Menge der Seitenrahmen (physischer Adressraum)
- **$f_t: N \rightarrow M \cup \{\perp\}$:** Funktion, die den zur Zeit t aktuellen Seitenzustand angibt.

  $f_t(i) = j$ falls Seite $i \in N$ zur Zeit t im Seitenrahmen $j \in M$ ist

  $f_t(i) = \perp$ sonst („Seitenfehler“, Nachladen erforderlich)

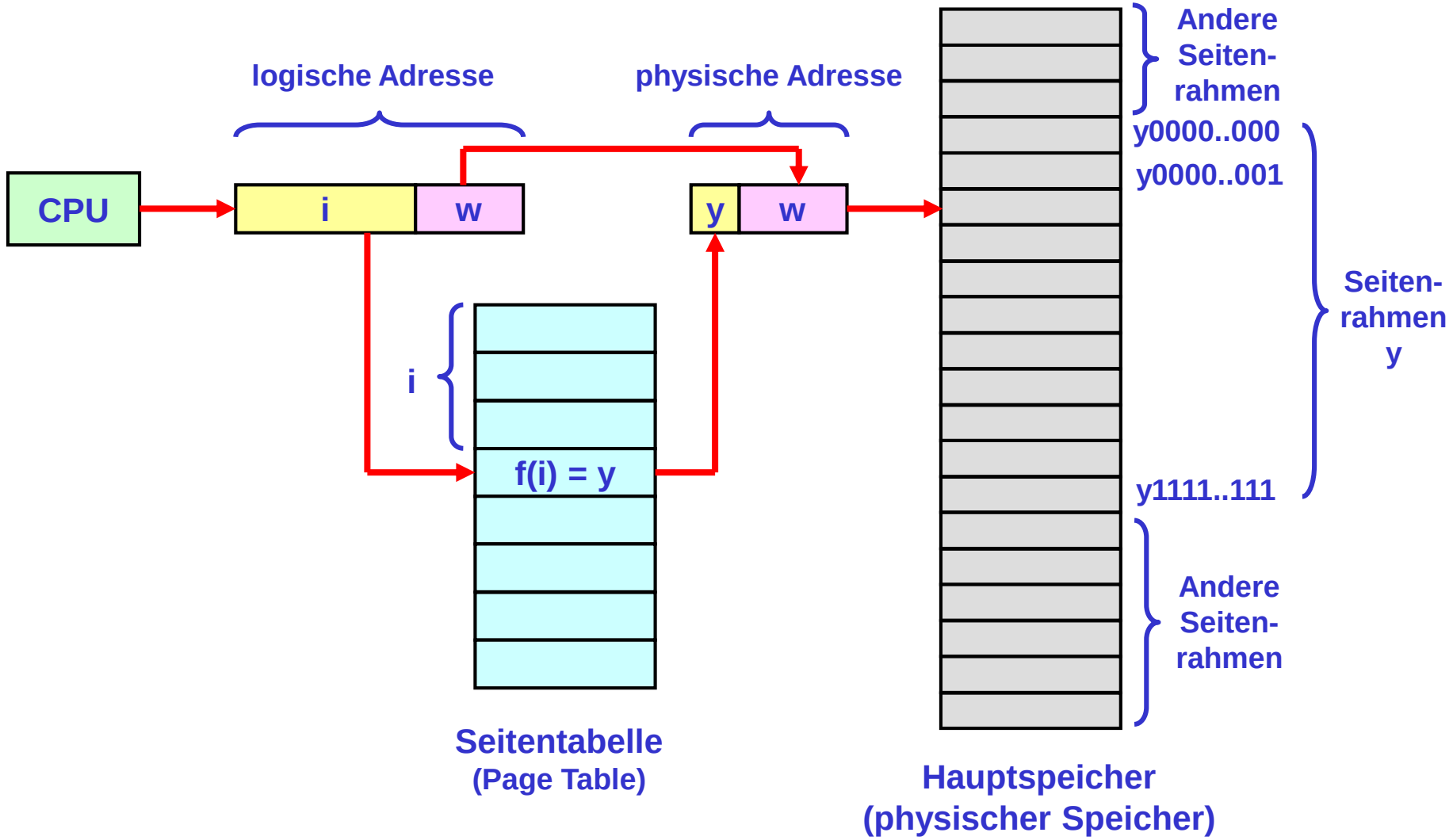
Befindet sich die gewünschte Seite aktuell in einem Seitenrahmen, dann ist die **Adressumsetzung bei geeigneter Wahl der Seitengröße trivial:**

Logische Adresse: (i, w)



Adresse im Hauptspeicher: ($f_t(i)$, w)

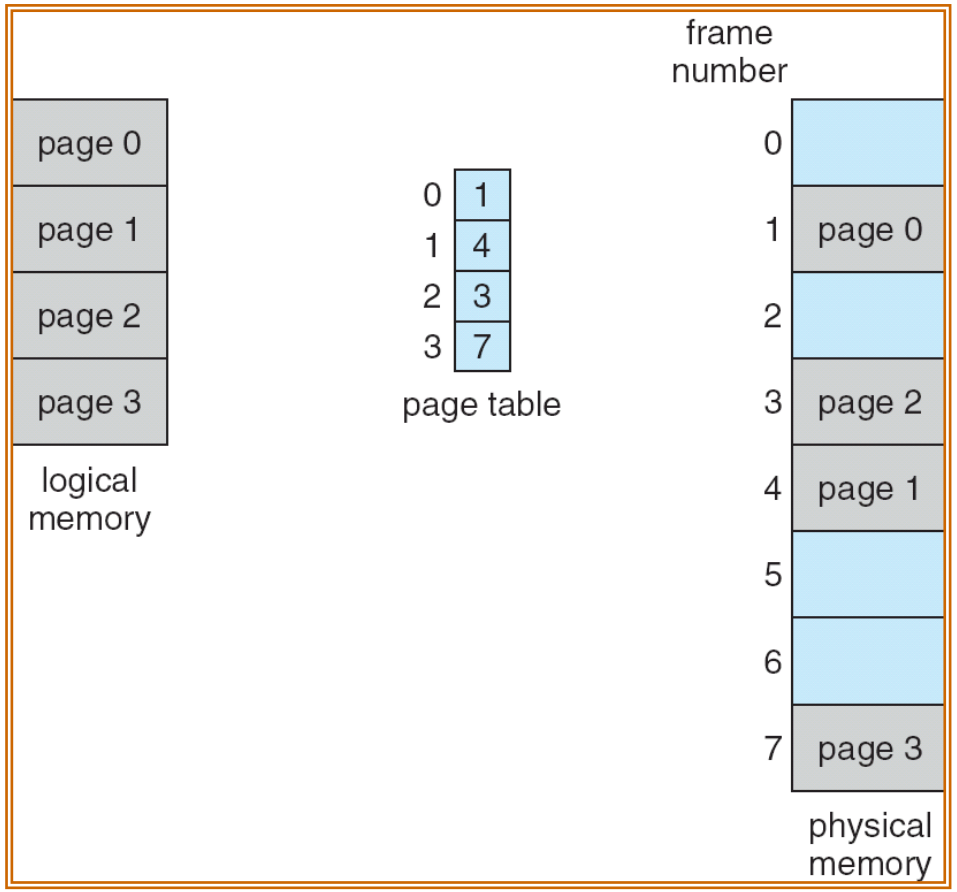
4.5.2.2. Logische Adressen und physische Adressen



i: Seitennummer
y: Nummer des Seitenrahmens
w: Adresse innerhalb der Seite (Offset/Displacement)

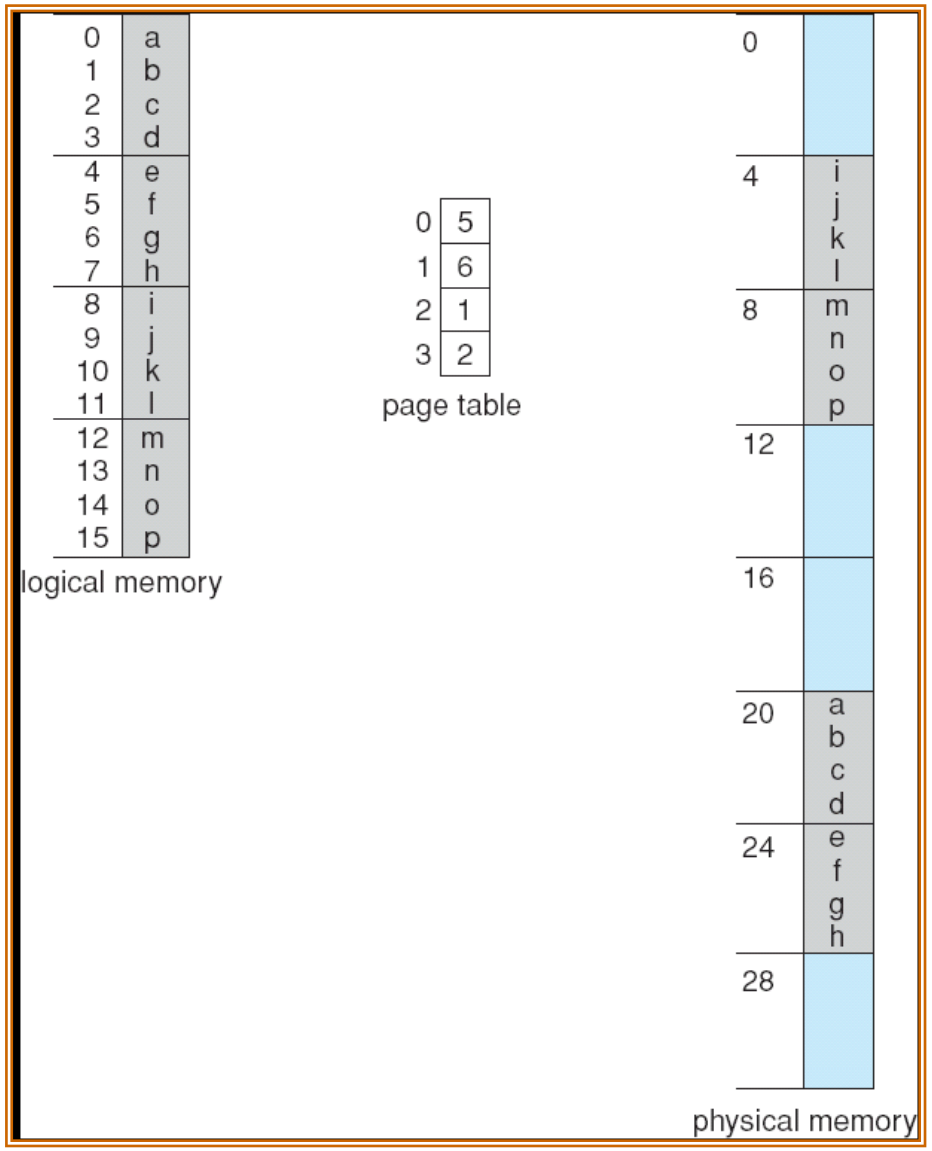
Beispiel: Logischer und physischer Speicher

Die nachfolgende Graphik zeigt ein Beispiel, in dem die Seiten 0, 1, 2 und 3 in die Seitenrahmen 1 (Seite 0), 3 (Seite 2), 4 (Seite 1) und 7 (Seite 3) eingelagert sind.



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 289

Beispiel: Paging mit 32 Byte Speicher (Seitengröße 4 Byte)



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 289

Die Graphik zeigt ein Beispiel, in dem aktuell insgesamt 16 Byte des (insgesamt sehr großen) logischen Adressraums genutzt werden.

Da als Seitengröße 4 Byte gewählt wurde, verteilen sich diese Daten auf 4 Seiten, die hier in den Seitenrahmen 1, 2, 5 und 6 gespeichert sind:

Seitenrahmen 1 (ab Byte 1 x 4 = 4) enthält die Seite 2 mit Offset/Displacement
- 0 für „i“
- 1 für „j“
- 2 für „k“
- 3 für „l“.

Seitenrahmen 5 (ab Byte 5 x 4 = 20) enthält die Seite 0 mit Offset/Displacement
- 0 für „a“
- 1 für „b“
- 2 für „c“
- 3 für „d“.

4.5.2.3. Zur Wahl der Seitengröße

Die Seiten sollten groß gewählt werden (viele Speicherworte enthalten),
… damit die **Seitentabelle klein** wird.

⇨ Ein großer Adressraum enthält sonst zu viele Seiten.

Die Seiten sollten klein gewählt werden (wenige Speicherworte enthalten),
… um den **Verschnitt innerhalb von Seiten** klein zu halten.

⇨ Seiten werden den Prozessen stets ganz oder gar nicht zugeordnet.
Hieraus resultiert „interne Fragmentierung“.

Aus offensichtlichen Gründen sollte die Seitengröße als Zweierpotenz gewählt werden.
Eine „optimale“ konkrete Wahl dieser Zweierpotenz ist aber stark anwendungsabhängig.

Manche Rechnerarchitekturen unterstützen mehrere Seitengrößen gleichzeitig.

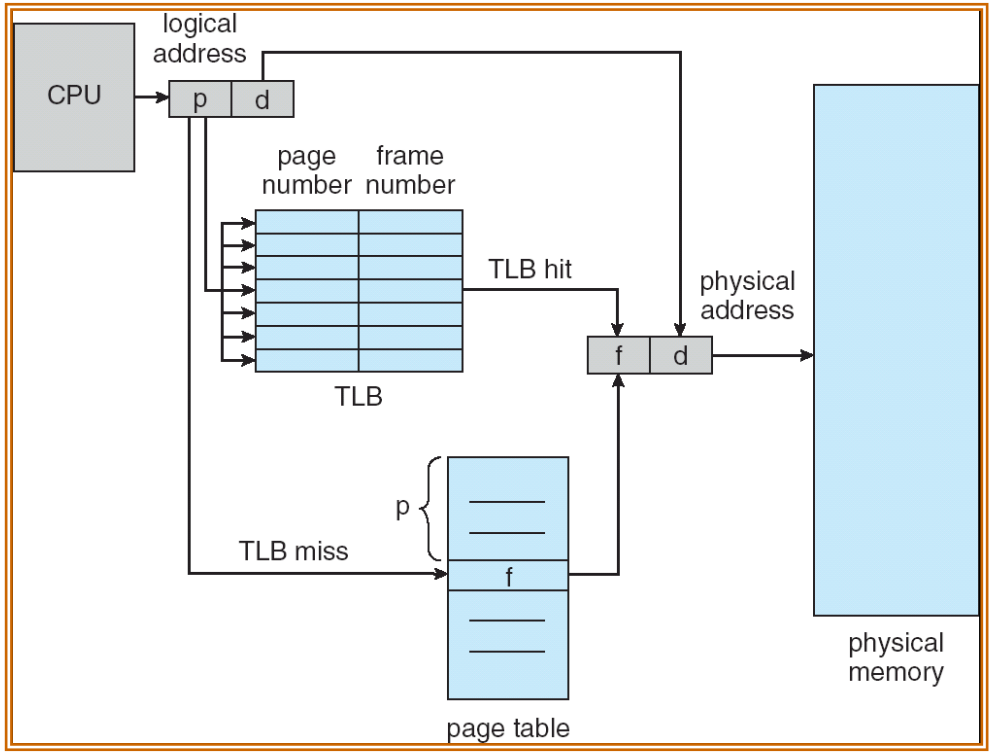
Insgesamt sind die Seitengrößen in den vergangenen Jahren gewachsen. In vielen Fällen ist die Seitengröße vom Nutzer einstellbar – sofern er hierzu berechtigt ist …. ;-)

4.5.2.4. Der Translation Look-aside Buffer (TLB)

Heutige Betriebssysteme verwalten meist **für jeden Prozess eine eigene Seitentabelle mit ggf. Millionen von Einträgen**.

Somit kann im Regelfall die Seitentabelle nicht mehr vollständig in schnellen Registern gehalten werden, sondern es muss auf andere Speicher (insbes. den Hauptspeicher) ausgewichen werden.

Die Standard-Lösung hierzu heißt **Translation Look-aside Buffer (TLB).**



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 294

Erläuterungen:

TLBs sind sehr schnelle Assoziativspeicher,
ca. 10 x schneller als Hauptspeicher-Zugriffe.

Zugriffe auf Assoziativspeicher erfolgen durch (parallelen) Vergleich mit dem Inhalt der Zellen; also ***nicht*** durch Adressierung!

Da TLBs nicht nur schnell sind, sondern auch teuer, gestatten sie **meist nur zwischen 64 und 1.024 Einträge.**

Wenn der Rahmen zur gesuchten Seite nicht im TLB enthalten ist, dann muss auf die „normale“ Seitentabelle zugegriffen und die Wartezeit in Kauf genommen werden.

4.5.2.5. Strukturierung der Seitentabelle

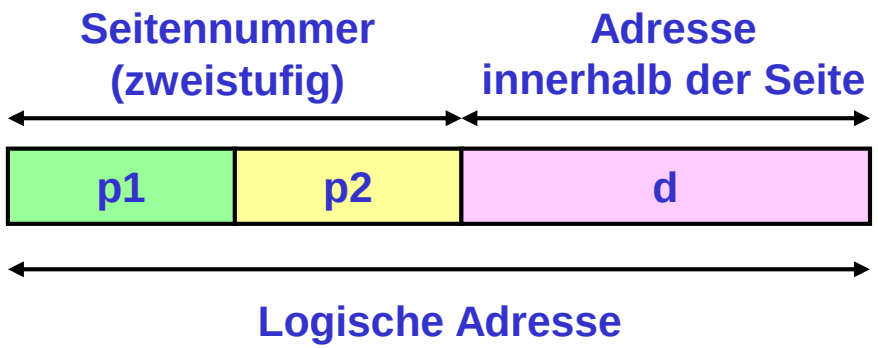
Bei großem logischem Adressraum (z.B. 64-Bit-Adressen) wird die Seitentabelle riesig.

Beispiel:	Logische Adresse:	32 Bit
	Seitengröße:	4 KByte (= 2^{12} Byte)
	Größe der Seitentabelle:	**1 Million Einträge** ($2^{32}/2^{12}$)
		→ **4 MByte** bei 4 Byte pro Eintrag
		… pro Prozess !!!

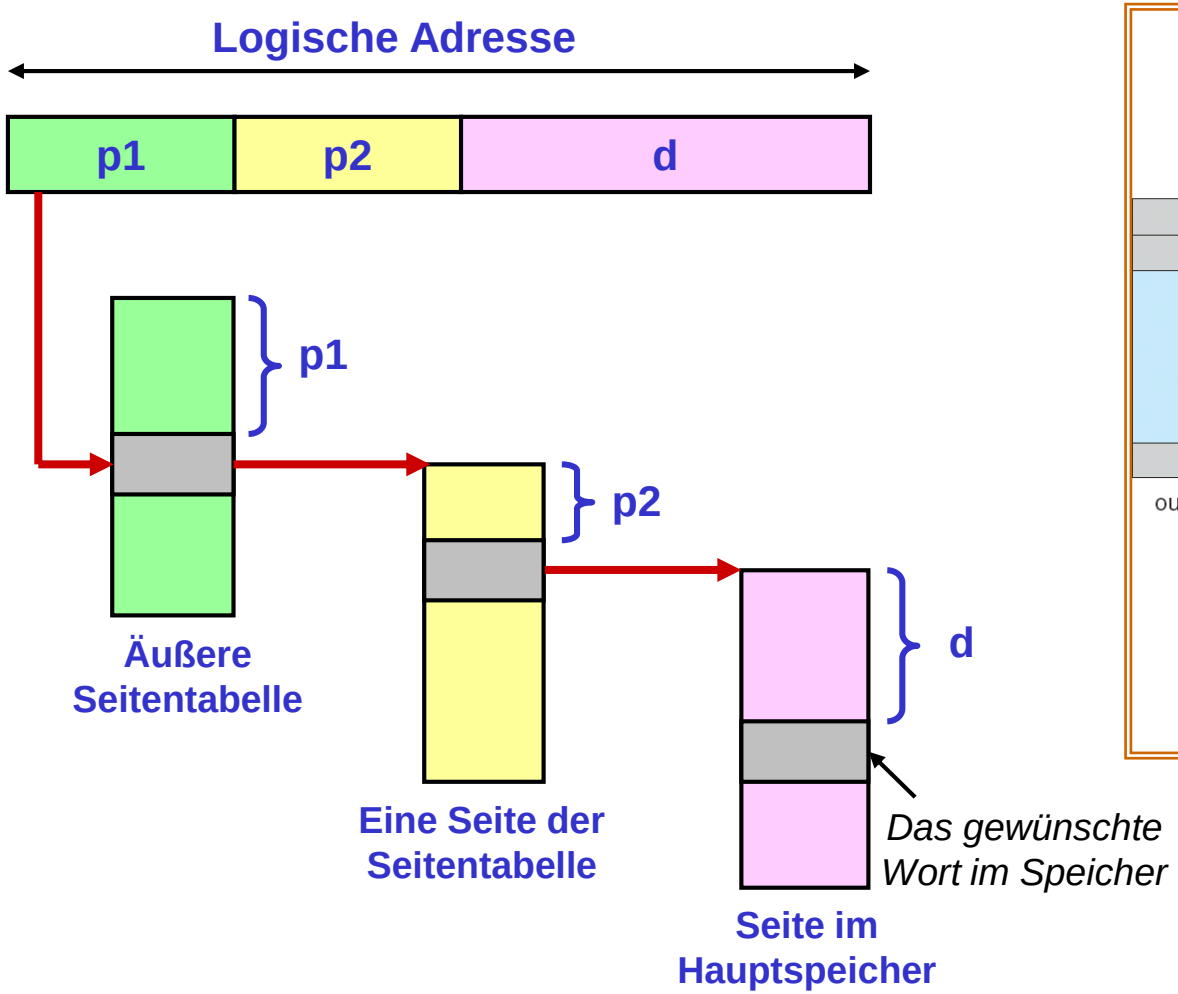
Das hier skizzierte Problem kann durch **Paging für die Seitentabelle** gelöst werden:

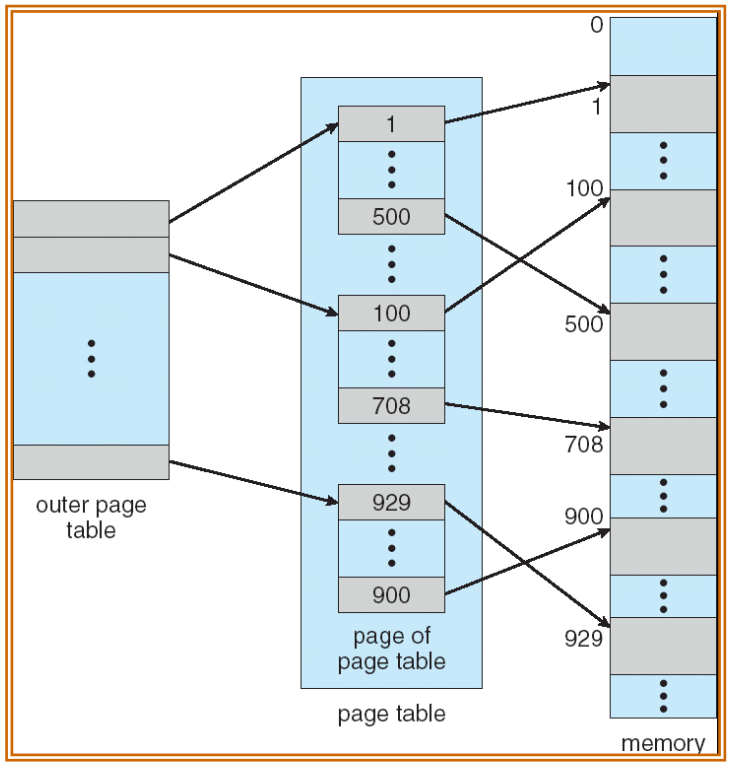
Die logische Adresse wird aufgeteilt in

- **Seite innerhalb der Seitentabelle** (im Beispiel: p1)
- **Offset/Displacement innerhalb dieser Seite der Seitentabelle** (im Beispiel: p2).
- **Offset/Displacement innerhalb der eigentlichen Seite** (im Beispiel: d)



Beispiel: Zweistufige Seitentabelle





Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 298

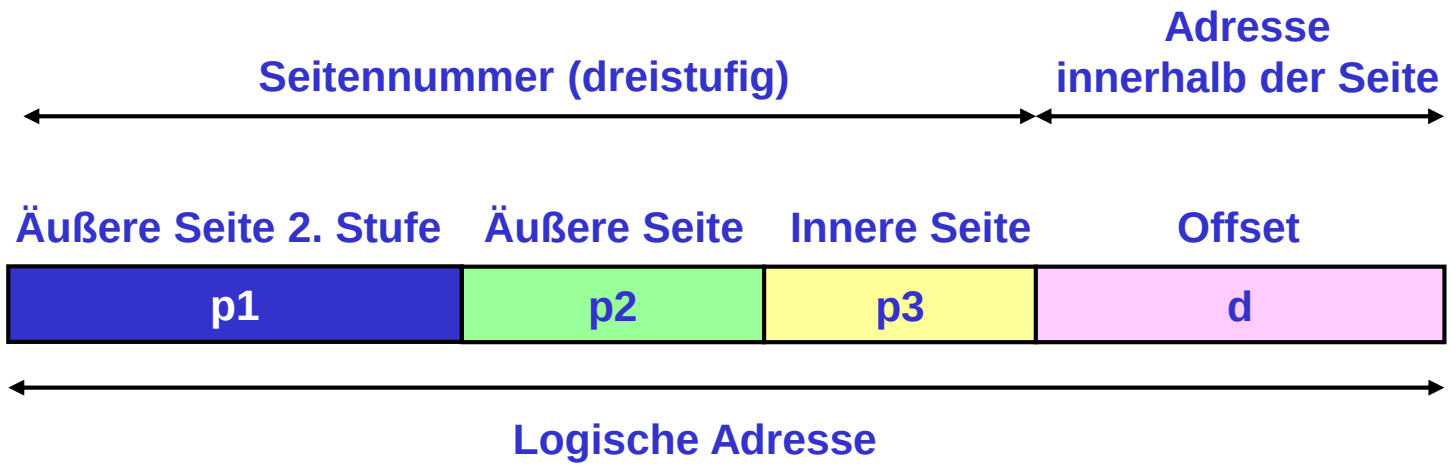
Erläuterungen:

- Die äußere Seitentabelle (grün) gibt an, in welchen Rahmen der Seitentabelle sich der Eintrag zur gewünschten Seite befindet.
- Innerhalb dieser Seite der Seitentabelle findet sich die Angabe, in welchem Rahmen des Hauptspeichers sich die gewünschte Seite befindet – sofern bzw. sobald sie dort eingelagert ist bzw. wurde

Weitere Hierarchiestufen

Bei **Adressierung mit 64 Bit** ist die **Erhöhung der Hierarchie** zweckmäßig, z.B. eine dreistufige Seitentabelle mit

- äußerer Seite der 2. Stufe,
- äußerer Seite
- innerer Seite
- Offset/Displacement



Seitentabellen mit Hashing

Eine **„Hash-Funktion“** ist eine Abbildung von einer

- **Eingabe, die Element einer großen Quellmenge** ist, auf einen
- **Hash-Wert, der Element einer kleineren Zielmenge** (meist sehr viel kleiner) ist.

Hash-Funktionen erweisen sich in weiten Bereichen der praktischen Informatik als außerordentlich nützlich, u.a.

- beim Suchen in Datenbanken
- im Bereich Compilerbau (… Suchen in Symboltabellen, …)
- in Rechnernetzen (digitale Signaturen)
- … in Betriebssystemen

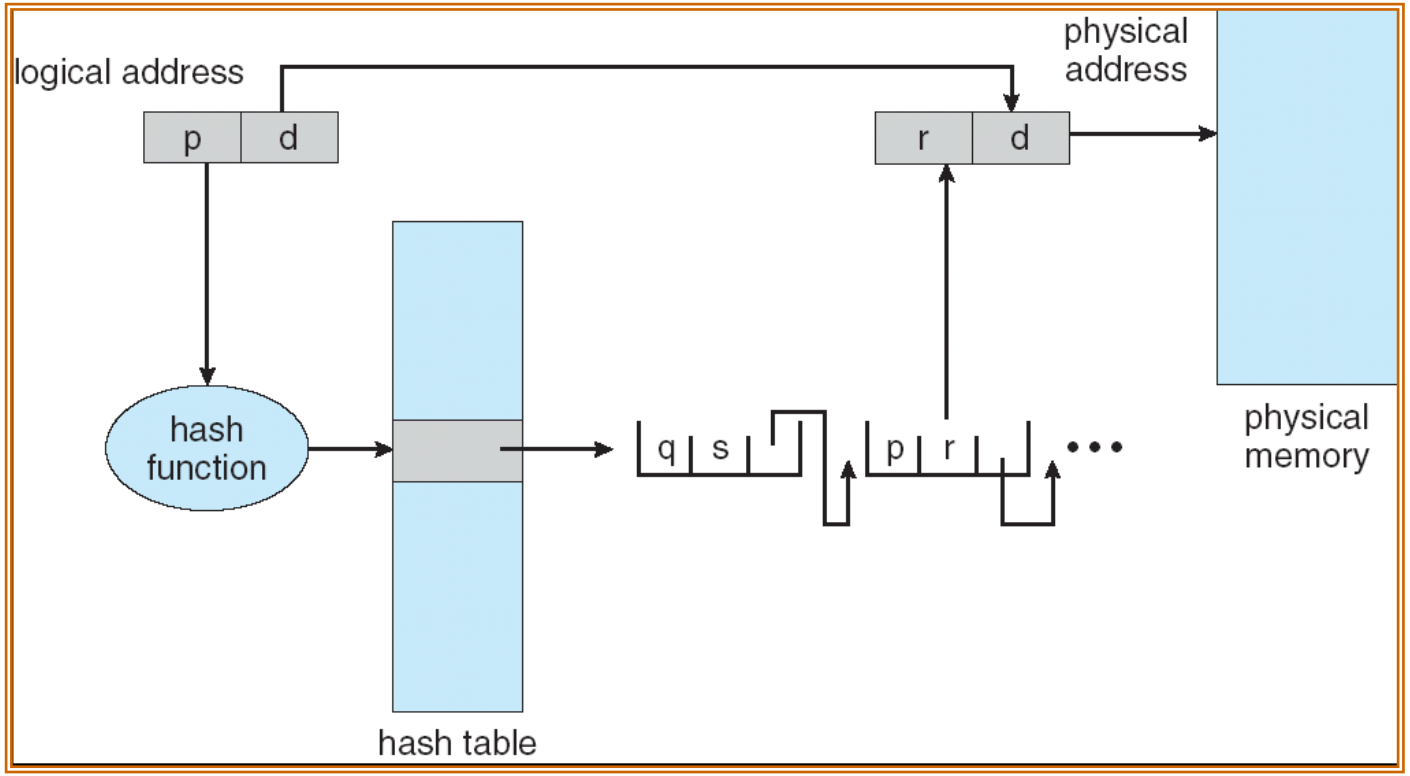
Beim Paging kann die (virtuelle/logische) Seitennummer **durch eine Hash-Funktion** abgebildet werden **auf den Eintrag in einer Hash-Tabelle**.

Dieser Eintrag kann eine **verkettete Liste mit Rahmenangaben** zu allen Seitennummern enthalten, die auf diesen Hash-Wert abgebildet werden.

Beispiel: Seitentabellen mit Hashing

Im Beispiel liefert die Hash-Funktion zu der Seitennummer p eine verkettete Liste* mit den Seitenrahmen zu

- **Seite q** (aktuell in Seitenrahmen s),
- **Seite p** (aktuell in Seitenrahmen r),
- ggf. weiteren Seiten.



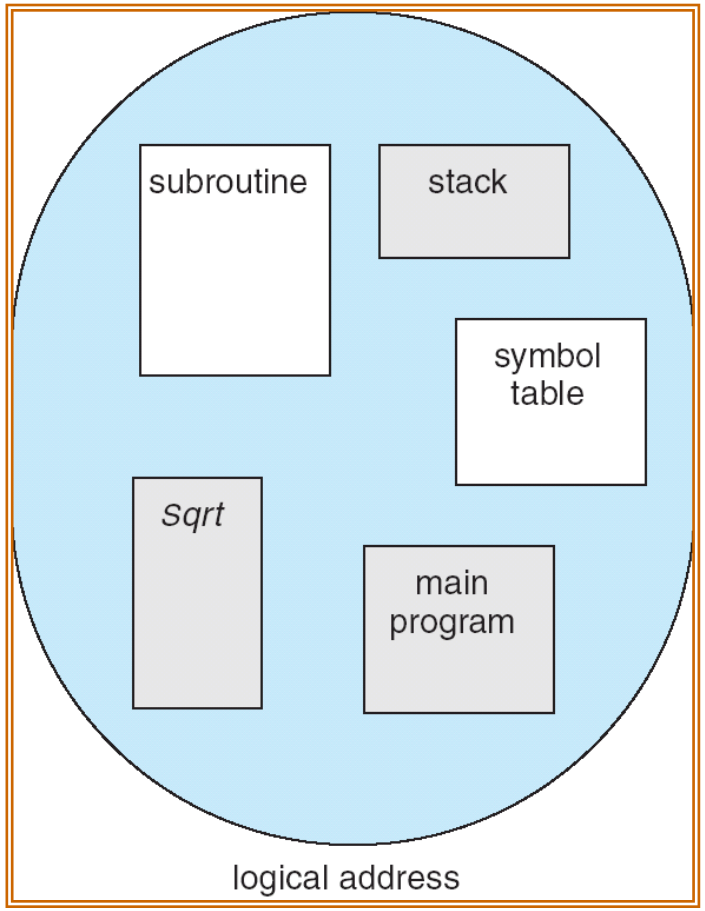
Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 300

*Verkettete Liste: Jedes Element enthält einen Pointer zum Nachfolger in der Liste

4.5.3. Segmentierung

Der Nutzer/Programmierer/Anwender sieht im Speicher meist

- **nicht** ein **lineares Feld von adressierbaren Speicherworten**,
- **sondern** eine **Kollektion von logischen Bereichen** (Segmenten)



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 303

Das Konzept der „Segmentierung“ setzt diese Nutzer-Sicht auch zur tatsächlichen Organisation des Speichers ein:

Logische Adressen geben jetzt an

- **Segmentnummer** (bzw. Segmentname) und
- **Offset innerhalb dieses Segments**.

Die hier angesprochenen Segmente werden vom Compiler automatisch erstellt.

Die Zuordnung zum physischen Adressraum erfolgt anschließend durch die **Segment-Tabelle**:

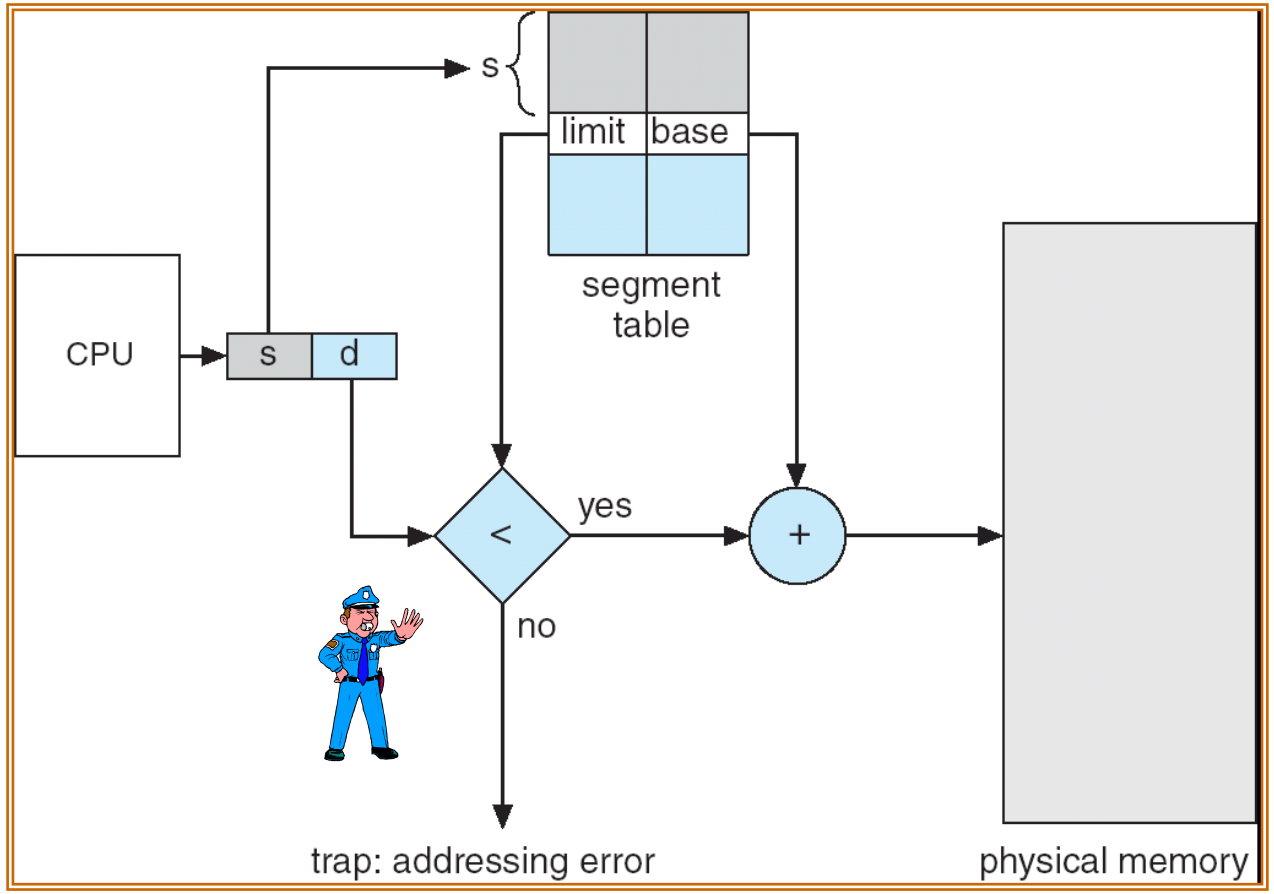
- Anfangsadresse des Segments (Base)
- maximale Segment-Größe (Limit)

Schutzmechanismen bei Segmentierung

Im hier gezeigten Beispiel wird beim Speicherzugriff überprüft, ob

- der **Offset d**
- innerhalb des Segments **s**
- die zugehörige maximale Segmentgröße **limit**

nicht erreicht bzw. überschreitet.



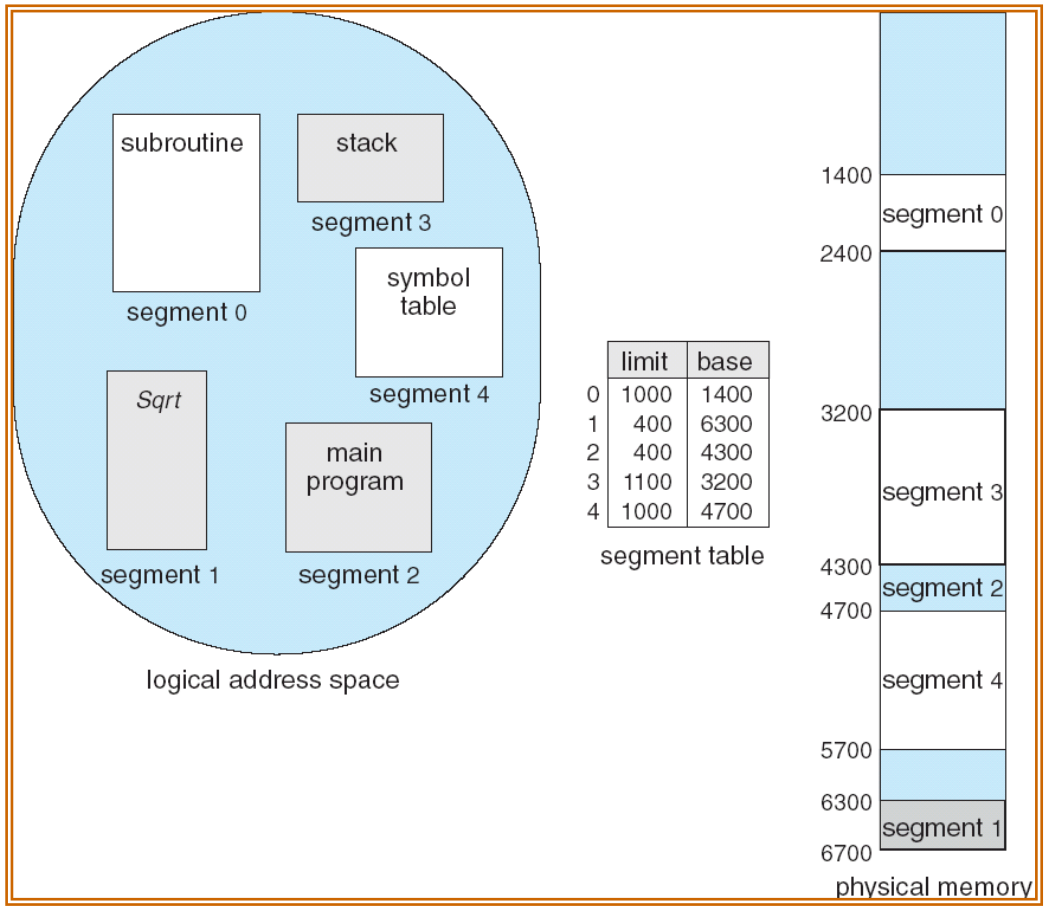
Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 304

Ein Beispiel für Segmentierung

Im Beispiel würde etwa ein Zugriff auf Byte 53 in Segment 2 abgebildet auf Speicherzelle 4353 (= 4300 + 53)

Basisadresse von Segment 2

Offset innerhalb von Segment 2



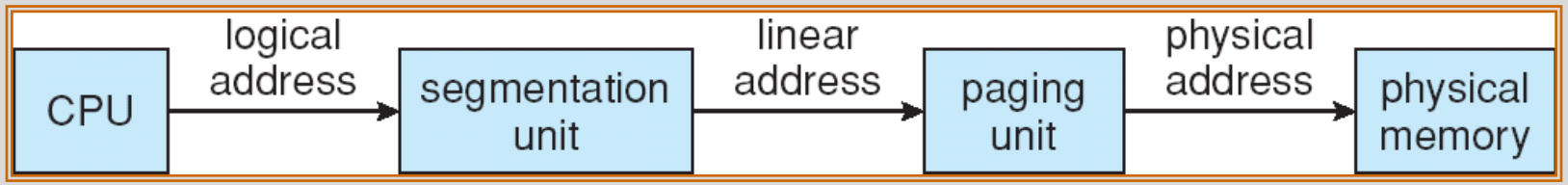
Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 305

Segmentierung in Kombination mit Paging

Wenn das Konzept der Segmentierung in Kombination mit Paging eingesetzt werden soll, dann ist zunächst eine

- **Umsetzung der logischen Adresse** (= Segment + Offset innerhalb des Segments)
- **in eine Adresse des linearen Adressraums** erforderlich.

Beispiel: Segmentierung und Paging beim Intel Pentium:



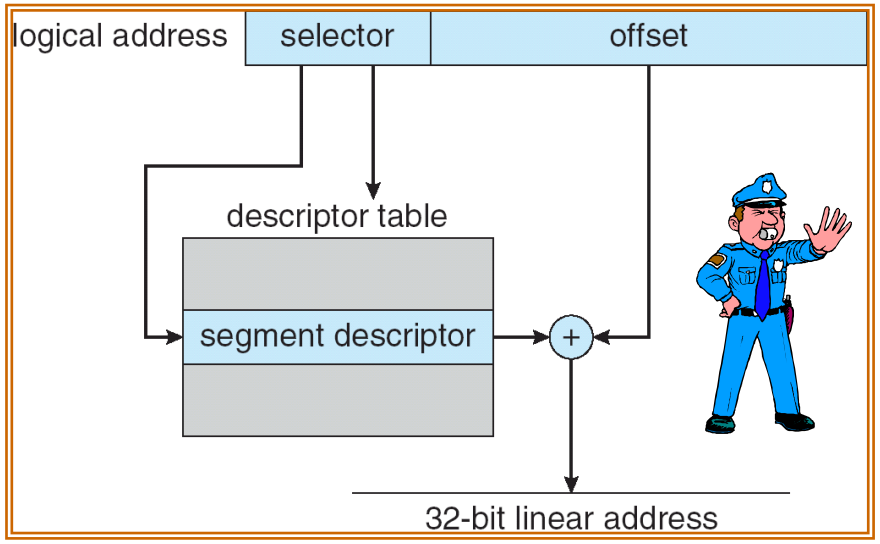
Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 306

Beispiel: Segmentierung/Paging beim Intel Pentium

Beim Pentium umfasst die **Segmentnummer („Selector“) 16 Bit**.

Innerhalb des Segments wird mit **32 Bit Offset** adressiert.

Die 32-Bit-Adresse für den Zugriff auf den linearen Speicher wird unter Rückgriff auf die sog. „descriptor table“ ermittelt.



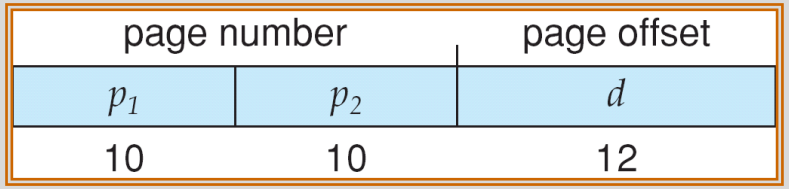
Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 307

Bei der Adressumsetzung wird überprüft, ob die entstehende lineare Adresse im zulässigen Bereich liegt.

Beispiel: Segmentierung/Paging beim Intel Pentium (2)

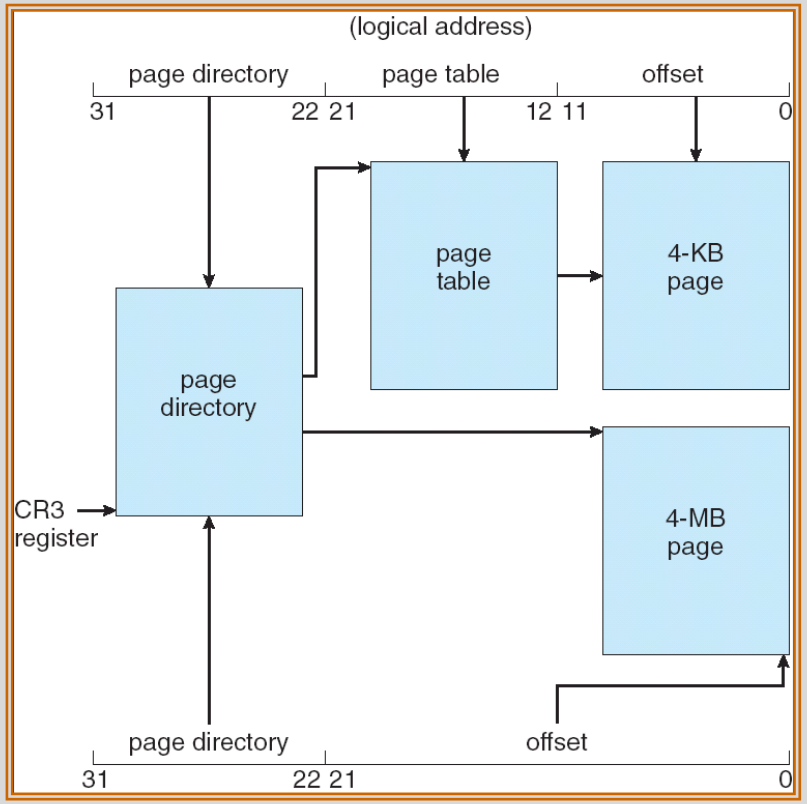
Die Pentium-Architektur unterstützt Seiten der Größen **4 KByte** und **4 MByte**.

Bei Seitengröße 4 KByte wird eine zweistufige Seitentabelle eingesetzt.



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 306

Abschließend eine Graphik, welche die Umsetzung der linearen 32-Bit-Adresse in die physische Adresse zeigt.



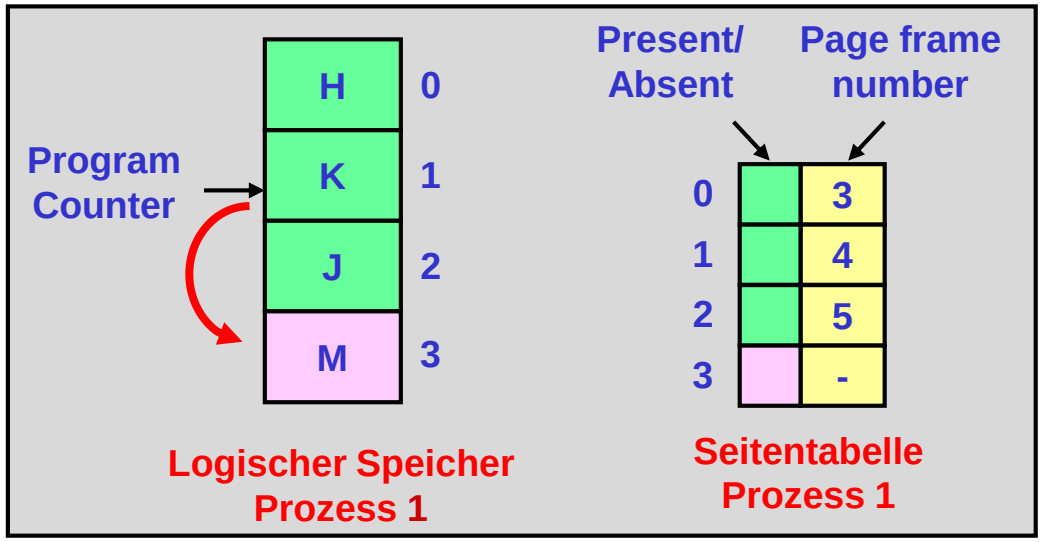
Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 308

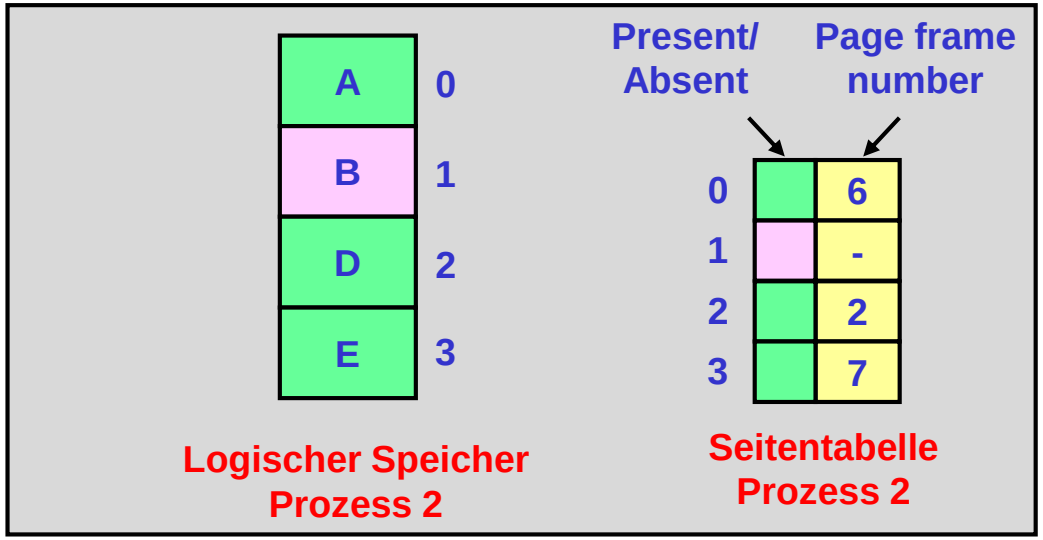
4.5.4. Behandlung von Seitenfehlern

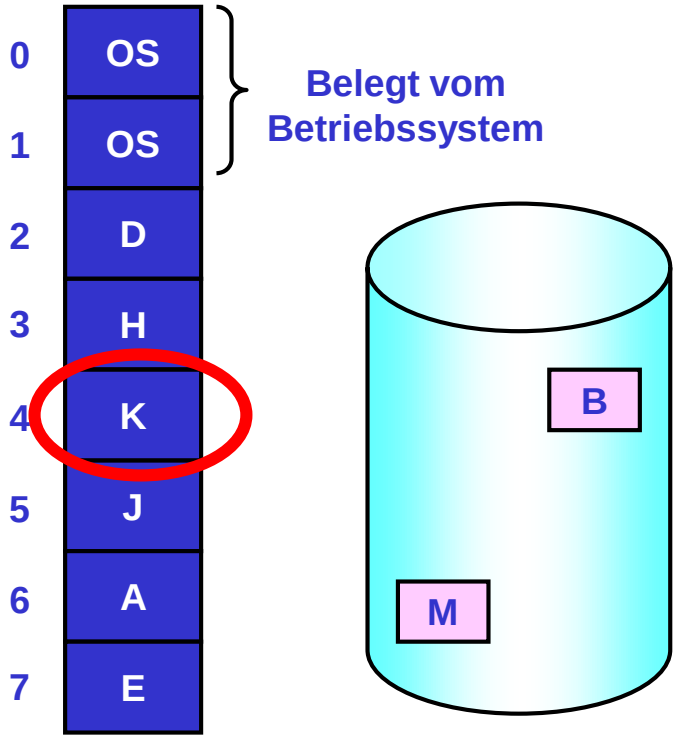
Ein „**Seitenfehler**“ tritt auf, wenn auf eine Seite zugegriffen werden soll, die sich aktuell nicht im physischen Speicher befindet.

4.5.4.1. Grundlegende Betrachtung

Betrachten wir nun als Beispiel ein System, in dem 2 Prozesse laufen, die insgesamt auf 8 Seiten zugreifen. Zwei Seiten werden vom Betriebssystem belegt.







Erläuterungen zur vorangehenden Folie

- Prozess 1 arbeitet auf den Seiten H, K, J, M (mit logischen Nummern 0, 1, 2, 3).
  - Die Seiten H, K und J sind aktuell im physischen Speicher eingelagert (Rahmen 3, 4, 5).
  - Die Seite M liegt im Hintergrundspeicher.
  - Aktuell wird Programm-Code in Seite K (Seite 1) bearbeitet.
  - Als Nächstes ist **Zugriff auf ein Wort in Seite M** erforderlich.
  - Die Seite M ist aber aktuell nicht verfügbar.

- Prozess 2 arbeitet auf den Seiten A, B, D, E (mit logischen Nummern 0, 1, 2, 3).
  - Die Seiten A, D und E sind aktuell im physischen Speicher eingelagert (Rahmen 6, 2, 7).
  - Die Seite B liegt im Hintergrundspeicher.

- Damit Prozess 1 weiterarbeiten kann, muss die Seite M in den physischen Speicher eingelagert werden, „**Swap in**“. Hierfür ist es aber erforderlich, eine andere Seite auszulagern, „**Swap out**“.

Offene Fragen:

- Welche Seite soll ausgelagert werden (Auswahl eines Opfers) ?
  - Eine eigene Seite, d.h. eine Seite von Prozess 1 ?
  - Eine Seite von Prozess 2 ?
- Sollte vielleicht einer der Prozesse komplett stillgelegt und ausgelagert werden?

4.5.4.2. Einlagern und Verdrängen von Seiten

Wenn ein Seitenfehler auftritt, dann können wir

a) den zugehörigen **Prozess zunächst stilllegen und**

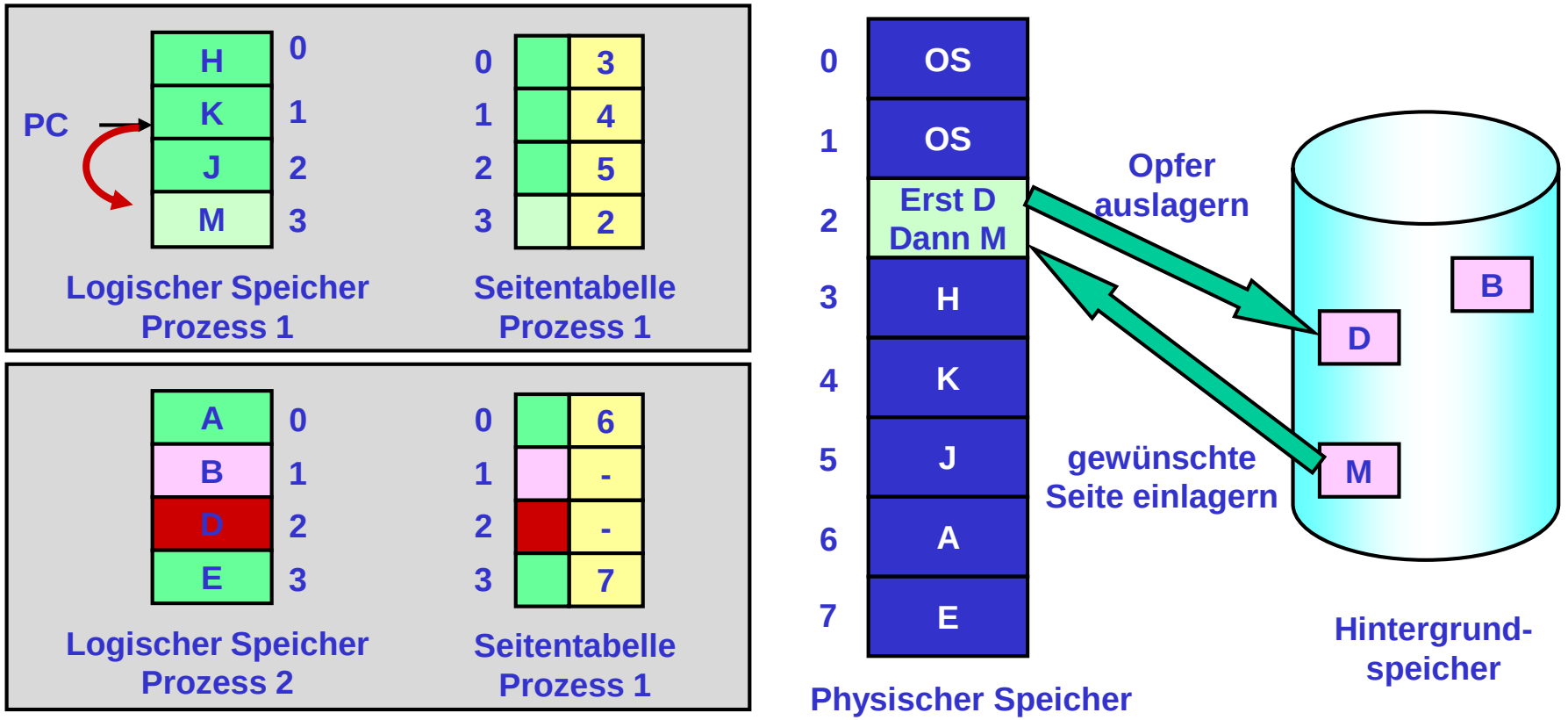
b) die fehlende **Seite nachladen**.

Nachladen von Seiten:

1. Suche/Finde die **fehlende Seite im Hintergrundspeicher**
2. Suche/Finde einen **freien Seitenrahmen**.
   a) Ist ein Rahmen frei, dann nutze diesen.
   b) Ist kein Rahmen frei, dann wähle ein „Opfer“ aus. Lagere dieses Opfer in den Hintergrundspeicher aus und ändere die Seitentabelle.
3. **Lade eine Kopie der fehlenden Seite** in den physischen Speicher und ändere die Seitentabelle entsprechend.
4. **Starte erneut den Prozess**, bei dem der Seitenfehler aufgetreten war.

Beispiel: Einlagern und Verdrängen von Seiten

Für das Beispiel aus der grundlegenden Betrachtung würde sich folgendes Bild ergeben, wenn die Seite D in Seitenrahmen 2 als „Opfer“ gewählt würde:



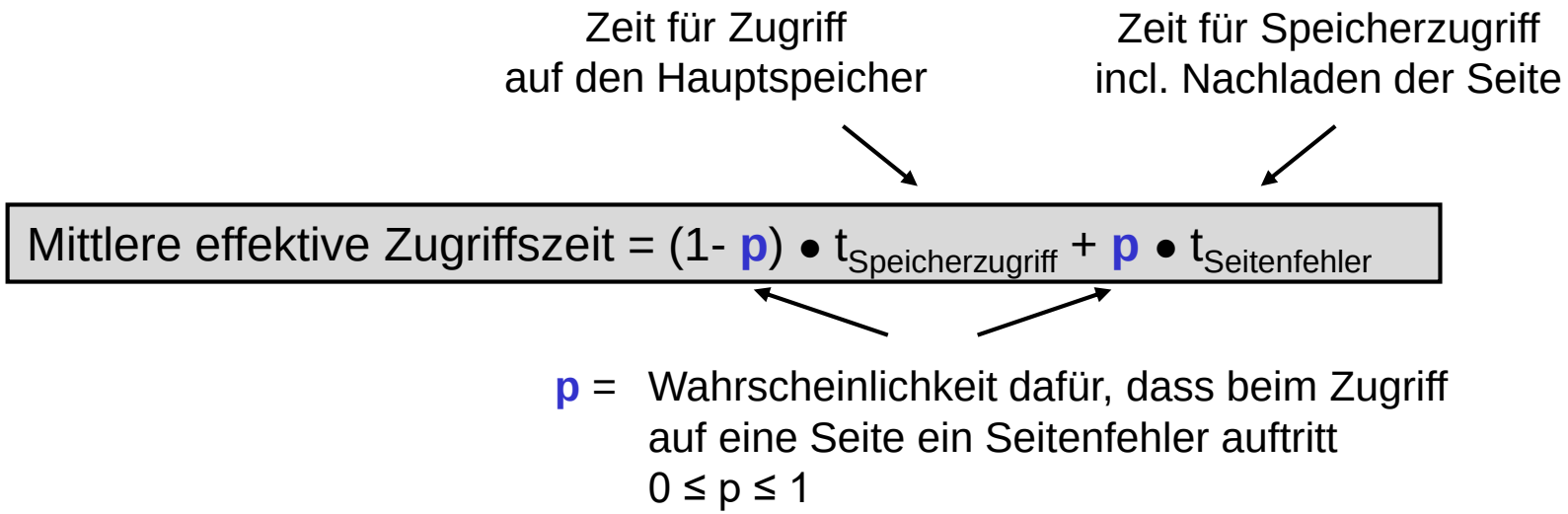
Wenn die auszulagernde Seite nicht verändert wurde, vgl. „Modified-Bit“ („Dirty Bit“), dann kann auf das Ablegen im Hintergrundspeicher verzichtet und die Seite einfach überschrieben werden.

4.5.4.3. Kosten von Seitenfehlern

Seitenfehler sind teuer: Sie kosten viel Zeit!

Eine hohe Leistungsfähigkeit des Gesamtsystems kann nur erzielt werden, wenn Seitenfehler selten auftreten. Zur Veranschaulichung folgen wir einer Berechnung, die Silberschatz in Kapitel 9.2.2 seines Buches „Operating System Concepts“ anstellt:

Bestimmung der „mittleren effektiven Zugriffszeit“:



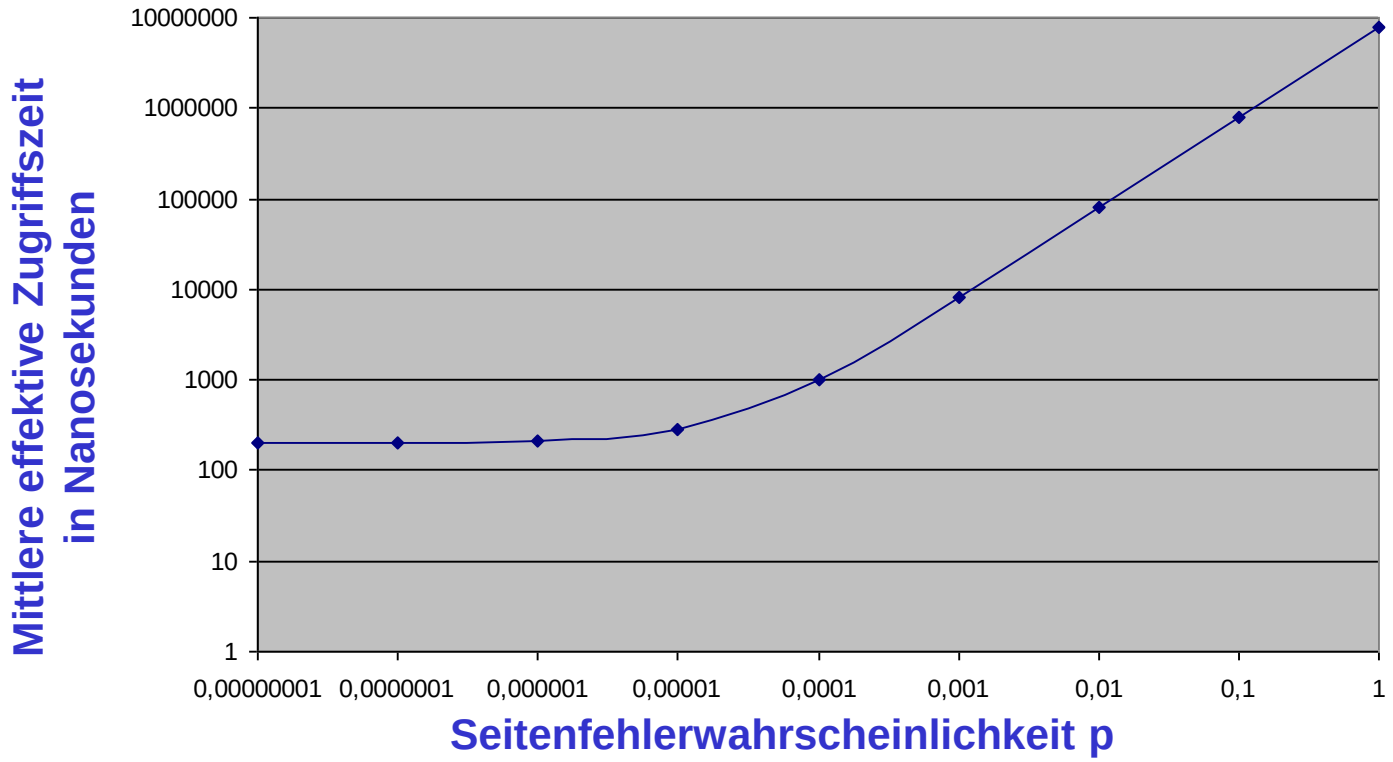
Für Zugriffe auf den **Hauptspeicher** werden lediglich Zeiten im Bereich von **10 bis 200 Nanosekunden** benötigt.

Werden als **Hintergrundspeicher** Festplatten eingesetzt, dann liegen die Zugriffszeiten meist im Bereich **einiger Millisekunden.**

Quantitative Betrachtung

Wir betrachten den Verlauf der mittleren effektiven Zugriffszeit für

$t_{Speicherzugriff}$ = 200 ns; $t_{Seitenfehler}$ = 8 ms

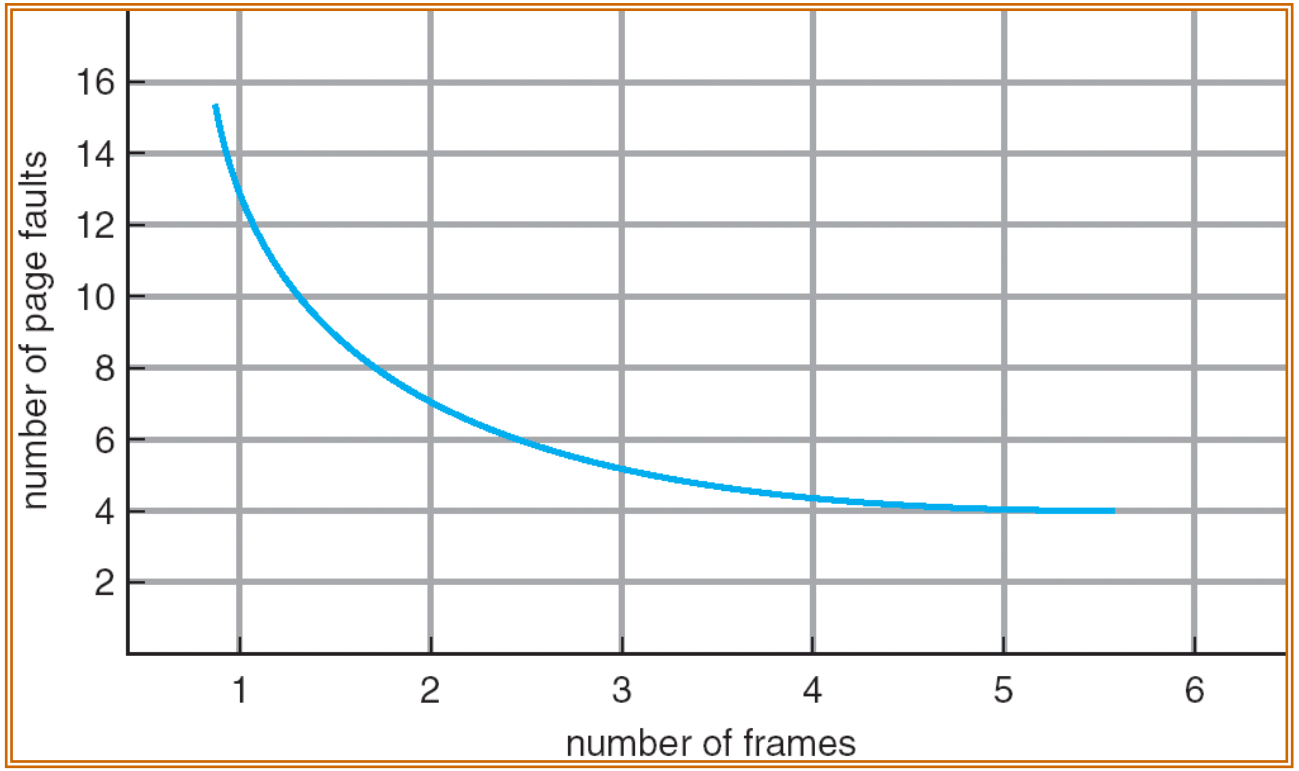


Tritt bei einem von 1000 Speicherzugriffen (p = 0,001) ein Seitenfehler auf, dann liegt die mittlere effektive Zugriffszeit schon bei ca. 8.200 ns. Das Paging verlangsamt das System dann um den Faktor 40.

Soll die Leistungseinbuße unterhalb von 10 % bleiben, dann darf nur weniger als einer von 400.000 Speicherzugriffen ein Seitenfehler sein !

Mehr Rahmen, weniger Seitenfehler ?

Wir würden erwarten, dass die Zahl der Seitenfehler sinkt, wenn wir die Zahl der Seitenrahmen erhöhen, d.h. mehr physischen Speicher bereit stellen.



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 331

In der Realität wird diese Erwartungshaltung nicht von allen Seitenersetzungs-strategien erfüllt.

4.5.4.4. Seitenersetzungsalgorithmen

Seitenfehler können häufig vermieden werden, indem die „richtigen“ Seiten im physischen Speicher gehalten – also nicht verdrängt – werden.

a) Welche Seite soll verdrängt werden, „Opfer“ werden ?

b) Auf welche Seite wurde lange nicht zugegriffen ?

c) Welche Seite wird lange nicht benötigt ?

Die Frage b) ist durch „**Buchhaltung**“ leicht zu beantworten.

Der sog. „**Reference String**“ $r = r_0\ r_1\ \ldots\ r_{T-1}$ erfasst, auf welche unterschiedlichen Seiten hintereinander zugegriffen wurde bzw. wird. Mehrfache Zugriffe auf dieselbe Seite unmittelbar hintereinander werden im Reference String also nur einfach erfasst.

Beispiel:

Bei Seitengröße 100 Bytes werde auf die folgenden Bytes in folgender Reihenfolge zugegriffen:

0100, 0423, 0201, 0202, 0203, 0204, 0484, 0205, 0483, 0402, 0304, 0238, 0453

Zugehöriger Reference String: 1, 4, 2, 4, 2, 4, 3, 2, 4

„Demand Paging“

Sei nun:

$r = r_0\ r_1\ \ldots\ r_{T-1}$ ein „reference string“ (= Folge der Seitenzugriffe eines Prozesses mit r_i: Seitennummer)

S_i: die Menge der Seiten, die nach r_i in Seitenrahmen sind,

X_i: die Menge der Seiten, die bei r_i nachgeladen werden,

Y_i: die Menge der Seiten, die bei r_i verdrängt werden.

Dann gilt offenbar: **$S_i = (S_{i-1} \cup X_i) \setminus Y_i$** „alt + neu - verdrängt“

Von **„Demand Paging“** sprechen wir, wenn gilt:

$0 \leq |Y_i| \leq |X_i| \leq 1$, mit $|Y_i| = |X_i|$ genau dann, wenn vor r_i alle Rahmen belegt waren.

Kann ein größerer Speicher zu mehr Seitenfehlern führen ?

Speicher mit mehr Seitenrahmen

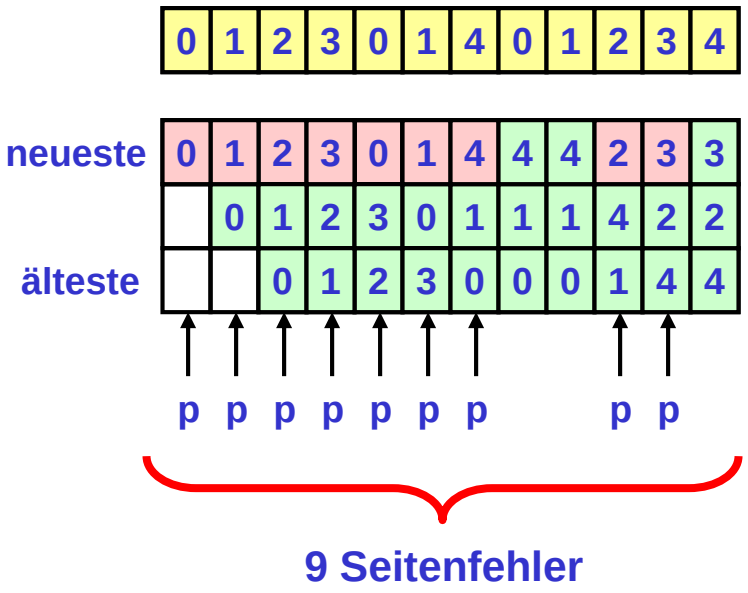
Die Belady-Anomalie (FIFO-Anomalie)

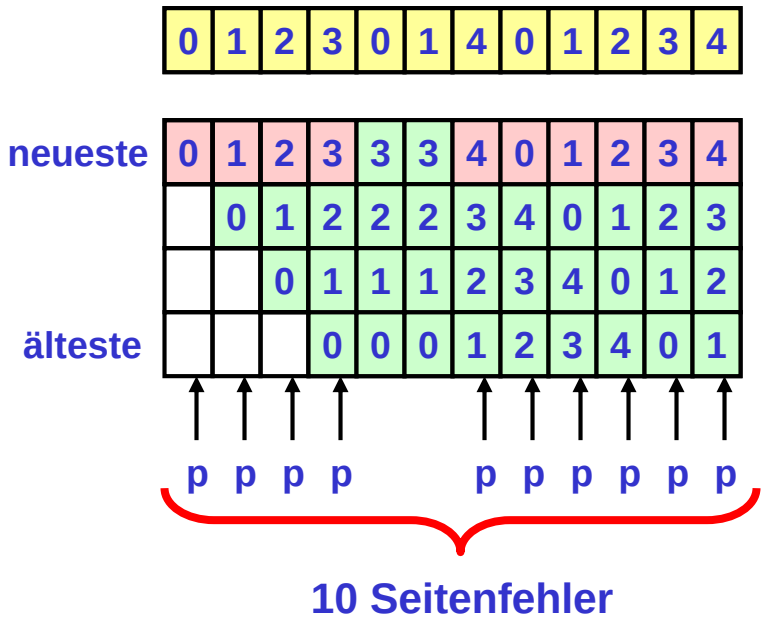
Sei n = 5 die Zahl der Seiten

m = 3 bzw. m = 4 die Zahl der Seitenrahmen

r = 0 1 2 3 0 1 4 0 1 2 3 4 ein Referenzstring.

Dann ergibt sich bei Seitenaustausch gemäß FIFO (Verdrängung der ältesten Seite):





Die hier im Beispiel gezeigte Eigenart ist als „**Belady-Anomalie**“ bekannt.

Sie kann vermieden werden, wenn für den Seitenaustausch eine bestimmte Klasse von Algorithmen eingesetzt wird: „**Stack-Algorithmen**“

„Stack-Algorithmen“

Definition: „Stack-Algorithmus“

Sei **A:** ein **Demand Paging Algorithmus,**
n: die Anzahl der **Seiten,**
m: die Anzahl der **Seitenrahmen** und
r: ein **Referenzstring** der Länge T ($r = r_0 \ldots r_{T-1}$).

Ferner bezeichne **S(m, r) die Menge der Seiten**, die nach r_{T-1} (d.h. **nach Abarbeitung von r**) in Seitenrahmen eingelagert sind.

A heißt „Stack-Algorithmus“ genau dann, wenn

- für **alle m ∈ IN** (d.h. für alle Speichergrößen) und
- für **alle r ∈ {0, 1, ... n-1}^{T}** (d.h. alle Referenzstrings der Länge T) gilt: **S(m+1, r) ⊃ S(m, r)**

aktive Seiten bei viel Speicher

aktive Seiten bei wenig Speicher

Anschaulich:

- Ein größerer Speicher enthält ggf. zusätzliche Seiten. Er enthält aber stets auch alle Seiten, die bei kleinerem Hauptspeicher vorhanden sind.
- Dies gilt für alle reference strings.

„Least Recently Used“ (LRU) ist ein Stack-Algorithmus !

(Verdränge stets die Seite, auf die am längsten nicht mehr zugegriffen wurde.)

4.6. Speicherverwaltung bei Multiprogramming

Bei Multiprogramming sind u.a. die folgenden Fragen zu beantworten:

a) Wie viele Seitenrahmen sollen pro Prozess eingesetzt werden ?

Idee: Wähle den sog. „Aktivitätsbereich“, d.h. alle „aktuell“ benötigten Seiten.

b) Wann ändert sich die Anzahl der Seitenrahmen eines Prozesses ?

Idee: Bei Änderung des Aktivitätsbereiches (z.B. bei Verlassen einer Schleife)

c) Wie viele Prozesse sollen gleichzeitig aktiv sein ?

viele: häufige Seitenfehler („Thrashing“)
wenige: ggf. schlechte Auslastung des Computer-Systems

d) Wann muss mindestens ein Prozess suspendiert werden ?

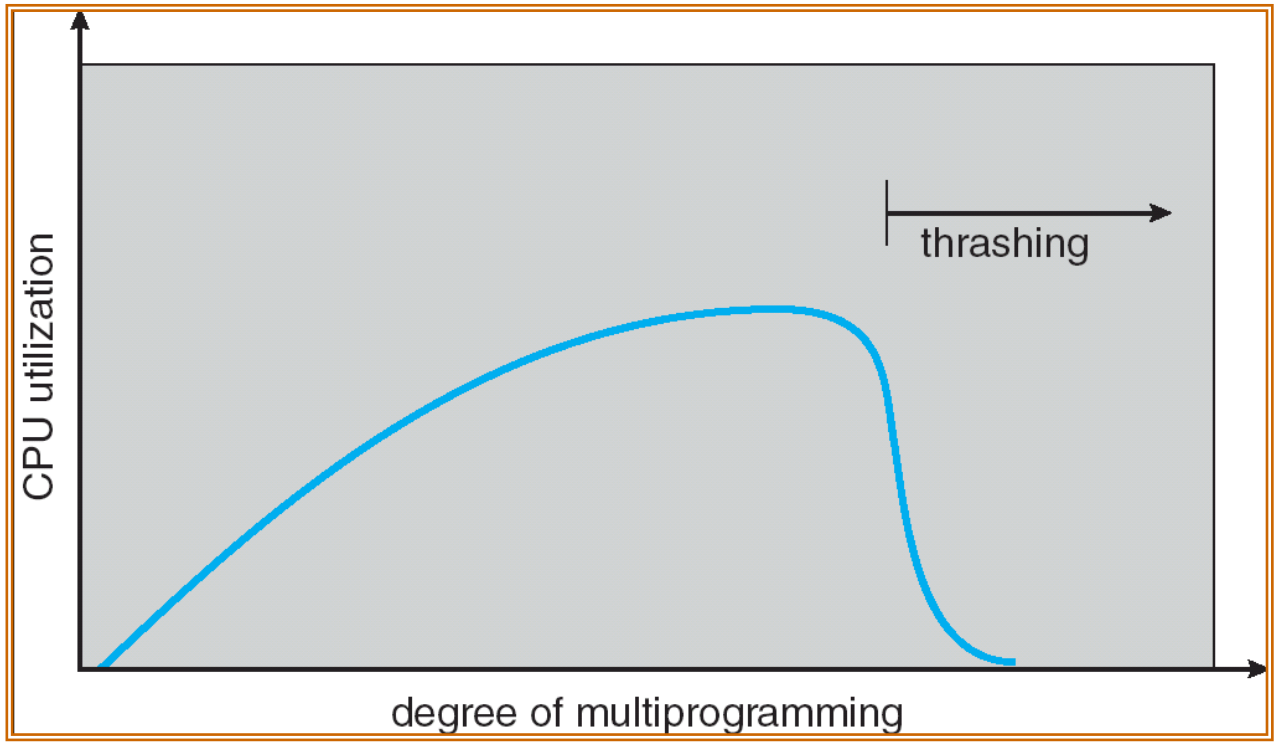
Wenn zu viele Seitenfehler auftreten und keine Vergrößerung des Speicherbereiches möglich ist.

e) Wann kann ein weiterer Prozess gestartet werden ?

Sobald die Aktivitätsbereiche aller z.Z. aktiven Prozesse klein genug geworden sind, um einen weiteren Aktivitätsbereich zuzulassen.

„Thrashing“

Wir sprechen von „Thrashing“ (Dreschen, auch: Seitenflattern), wenn mehr Zeit für das Austauschen von Seiten (Paging) aufgewendet wird als für das eigentliche Bearbeiten von Prozessen.



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 344

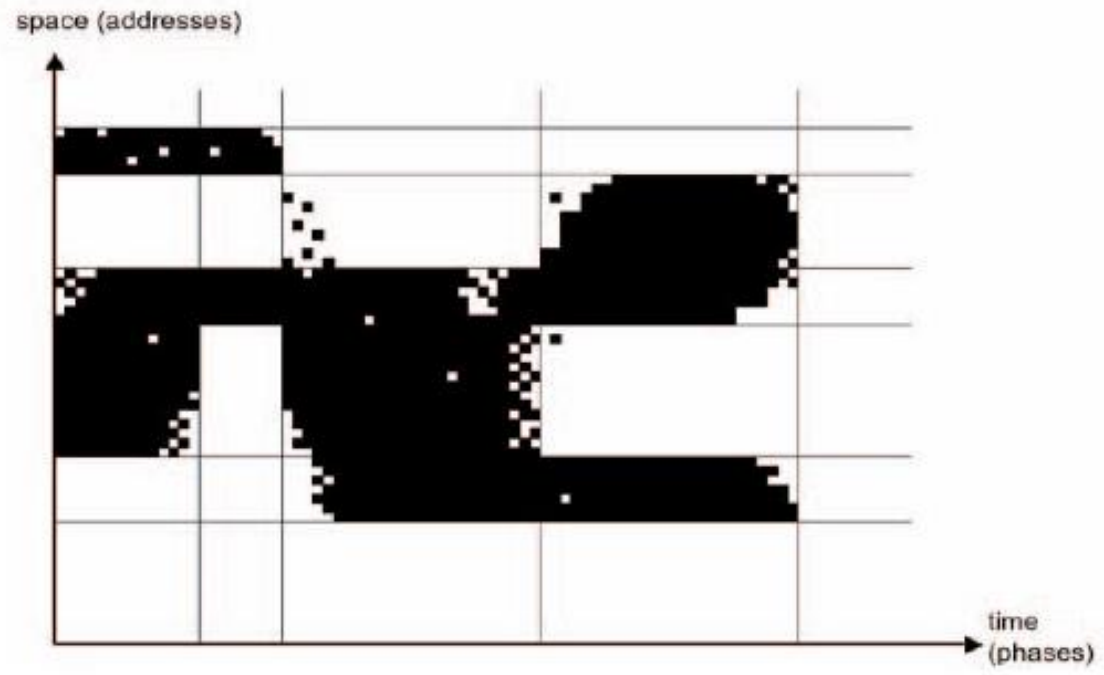
Wenn Thrashing auftritt, dann steht für einen oder mehrere Prozesse aktuell zu wenig physischer Speicherplatz zur Verfügung.

Frage: Wie viele Rahmen braucht ein Prozess ?

Das „Lokalitätsprinzip“

Auf Peter J. Denning geht die Idee zurück, jedem Prozess so viele Seitenrahmen zuzuordnen, wie seinem aktuellen „**Working Set**“ entsprechen. Das Working Set ist die Menge der **Seiten, auf die „in jüngster Vergangenheit“ zugegriffen wurde**.

Die Grundlage bildet das sog. „**Lokalitätsprinzip**“, d.h. die Beobachtung, dass aktive Prozesse **in bestimmten Phasen auf bestimmte Speicherbereiche zugreifen**.



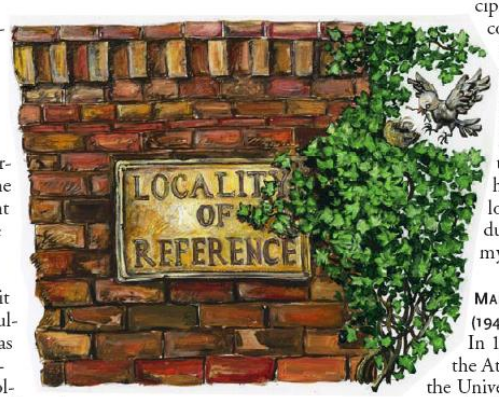
Source: **P.J. Denning, „The Locality Principle“, Communications of the ACM, July 2007, Vol. 48, No. 7, p. 23**

Das „Lokalitätsprinzip“ (2)

The Locality Principle

Locality of reference is a fundamental principle of computing with many applications. Here is its story.

Locality of reference is one of the cornerstones of computer science. It was born from efforts to make virtual memory systems work well. Virtual memory was first developed in 1959 on the Atlas System at the University of Manchester. Its superior programming environment doubled or tripled programmer productivity. But it was finicky, its performance sensitive to the choice of replacement algorithm and to the ways compilers grouped code onto pages. Worse, when it was coupled with multiprogramming, it was prone to thrashing—the near-complete collapse of system throughput due to heavy paging. The locality principle guided us in designing robust replacement algorithms, compiler code generators, and thrashing-proof systems. It transformed virtual memory from an unpredictable to a robust, self-regulating technology that optimized throughput without user intervention. Virtual memory became such an engineering triumph that it faded into the background of every operating system, where it performs so well at managing memory with multithreading and multitasking that no one notices.



The locality principle found application well beyond virtual memory. Today it directly influences the design of processor caches, disk controller caches, storage hierarchies, network interfaces, database systems, graphics display systems, human-computer interfaces, individual application programs, search engines, Web browsers, edge caches for Web-based environments, and computer forensics. Tomorrow it may help us overcome our problems with brittle, unreliable software.

I will tell the story of this principle, starting with its discovery to solve a multimillion-dollar performance problem, through its evolution as an idea, to its widespread adoption today. My telling is highly personal because locality was my focus during the first part of my career.

Manifestation of a Need (1949–1965)

In 1949, the builders of the Atlas computer system at the University of Manchester recognized that computing systems would always have storage hierarchies consisting of at least main memory (RAM) and secondary memory (disk, drum). To simplify management of these hierarchies, they introduced the page as the unit of storage and transfer. Even with this simplification, programmers spent well over half their time planning and

(MIT) and Les Belady (IBM), I started using the term “locality” for the observed tendency of programs to cluster references to small subsets of their pages for extended intervals. We could represent a program’s memory demand as a sequence of locality sets and their holding times:

(L1,T1), (L2,T2), ... , (Li,Ti), ...

This seemed natural because we knew that programmers planned overlays using diagrams that showed subsets and their time phases (see Figure 1). But what was surprisingly interesting was that programs showed the locality behavior even when it was not explicitly preplanned. When measuring actual page use, we repeatedly observed many long phases with relatively small locality sets (see Figure 2). Each program had its own distinctive pattern, like a voiceprint.

We saw two reasons that this would happen: temporal clustering due to looping and executing within modules with private data; and spatial clustering due to related values being grouped into arrays, sequences, modules, and other data structures. Both these reasons seemed related to the human practice of “divide and conquer”—breaking a large problem into parts and working separately on each. The locality bit maps captured someone’s problem-solving method in action. These underlying phenomena gave us confidence to claim that programs have natural sequences of locality sets. The working set sequence is a measurable approximation of a program’s intrinsic locality sequence.

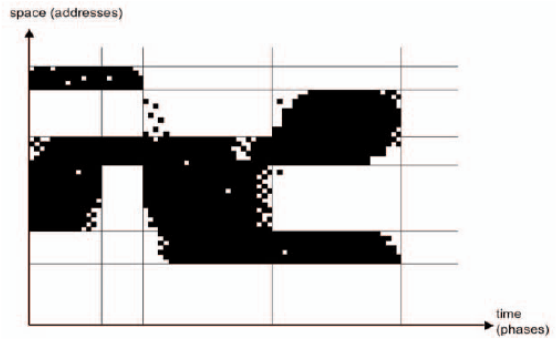


Figure 2. Locality-sequence behavior observed by sampling use bits during program execution.

During the 1970s, I continued to refine the locality idea and develop it into a behavioral theory of computational processes interacting with storage systems within a network of servers. (Table 1 lists key milestones.) By 1980, we defined locality much the same as it is defined today [4], in terms of a distance from a processor to an object x at time t, denoted $D(x,t)$. Object x is in the locality set at time t if the distance is less than a threshold: $D(x,t) \leq T$. Distance can take on several meanings: (1) Temporal: A time distance, such as the time since last reference, time until next reference, or even an access time within the storage system or network. (2) Spatial: A space distance, such as the number of hops in a network or number of addresses away in a sequence. (3) Cost: Any nondecreasing function of time since prior reference.

Adoption of Locality Principle (1967–present)

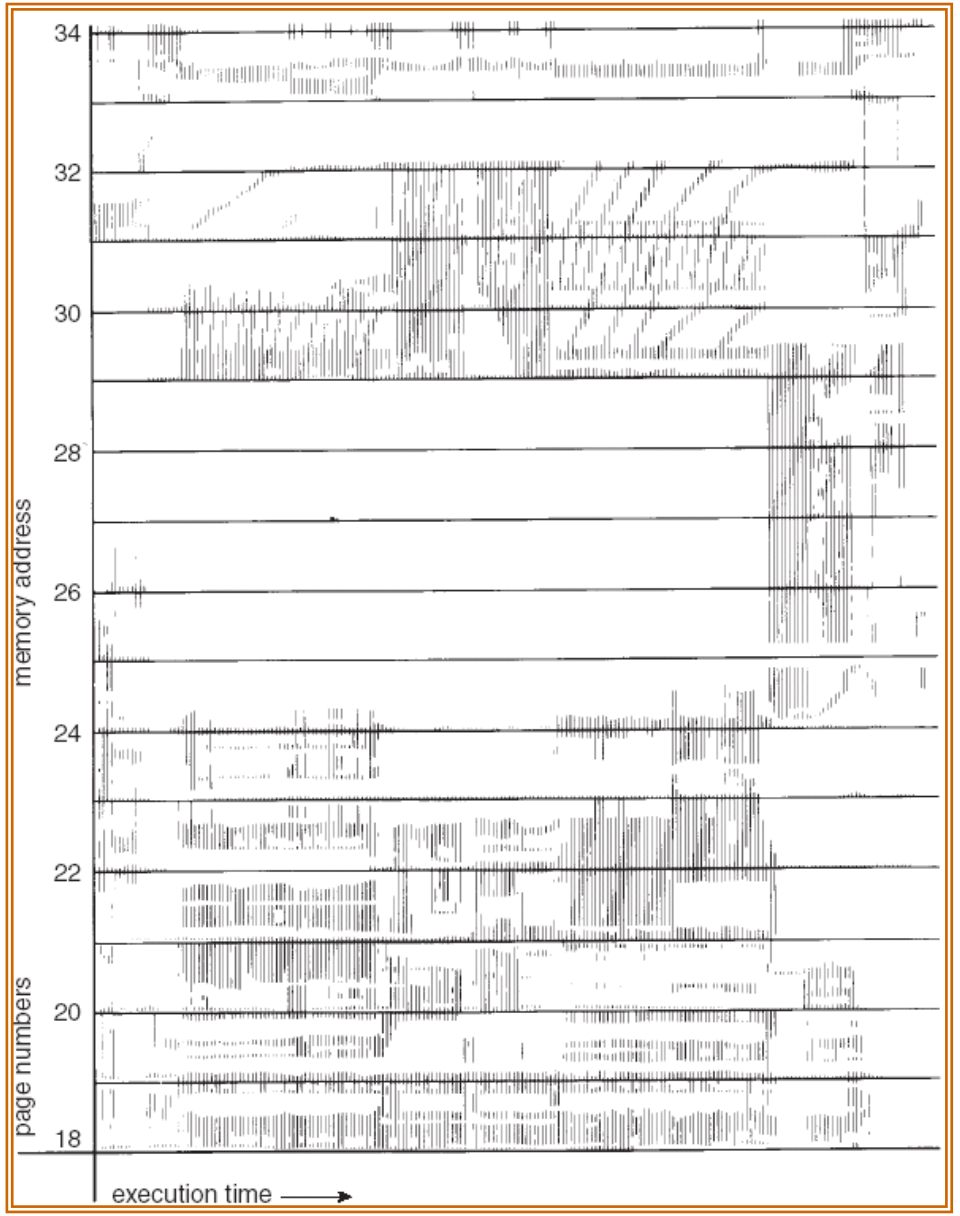
The locality principle was adopted as an idea almost immediately by operating systems, database, and hardware architects. It was applied in ever-widening circles:

- In virtual memory to organize caches for address translation and to design the replacement algorithms.
- In data caches for CPUs, originally as mainframes and now as microchips.
- In buffers between main memory and secondary memory devices.
- In buffers between computers and networks.
- In video boards to accelerate graphics displays.

Source: P.J. Denning, „The Locality Principle“, Communications of the ACM, July 2007, Vol. 48, No. 7

Das „Lokalitätsprinzip“ (3)

Silberschatz verwendet zur Illustration des Lokalitätsprinzips die folgende Graphik:



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p.345

Das Working-Set-Konzept

Wir definieren das „**Working Set**“ eines Prozesses mit reference string $r = r_0\, r_1 \ldots r_{T-1}$ als

$$W(k,h,r) := \bigcup_{\substack{i=1 \\ i+k-h \geq 0}}^{h} r_{i+k-h}$$

Der **Parameter h** muss

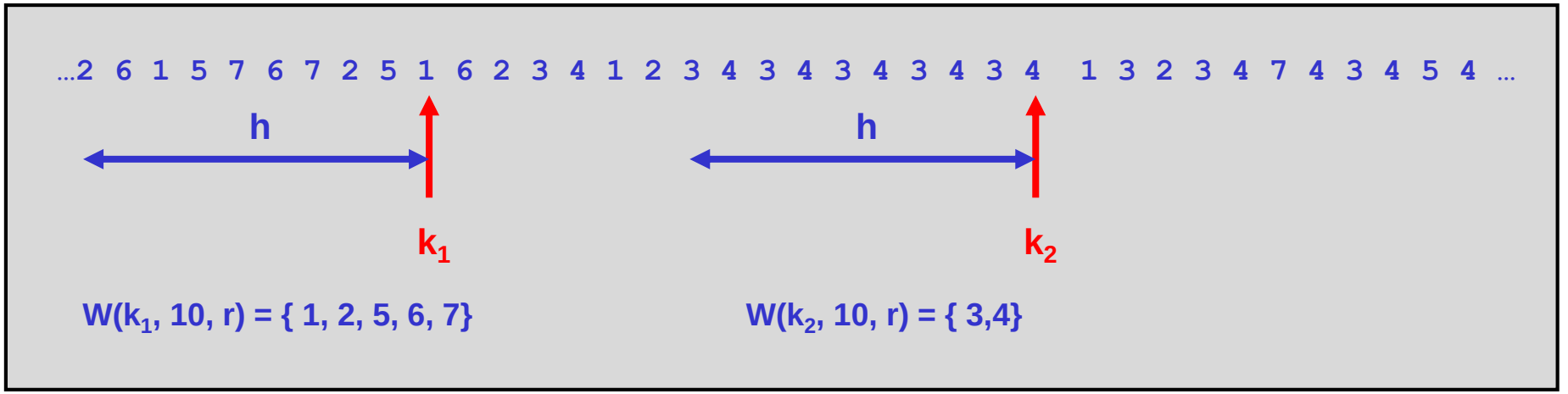
- so **groß gewählt** werden, dass die jeweils aktuell **nächste Seite** mit hoher Wahrscheinlichkeit (noch) im Working Set **enthalten** ist,
- so **klein gewählt** werden, dass **nicht mehr benötigte Seiten schnell aus dem Working Set fallen**.

Das **Working-Set-Konzept** lautet nun:

- Ein Prozess kann **nur dann aktiv werden, wenn** im Hauptspeicher **Platz genug für sein Working Set** ist.
- **Tauschkandidaten** bei Seitenfehlern sind genau die Seiten, die **aktuell zu keinem Working Set** gehören.
- Wenn **kein Tauschkandidat verfügbar** ist, dann wird der **Prozess mit dem aktuellen Seitenfehler stillgelegt.**

Das Working-Set-Konzept (2)

Sei das „Rückwärtsfenster“ h = 10 gewählt, dann ergeben sich im nachfolgenden Beispiel die angegebenen Working Sets.



In der Praxis wird das Working-Set-Konzept in unterschiedlichen Ausprägungen eingesetzt.

Eine besondere Herausforderung besteht in der **Protokollierung von Seitenzugriffen**, die nur mit **Hardware-Unterstützung** effizient möglich ist.

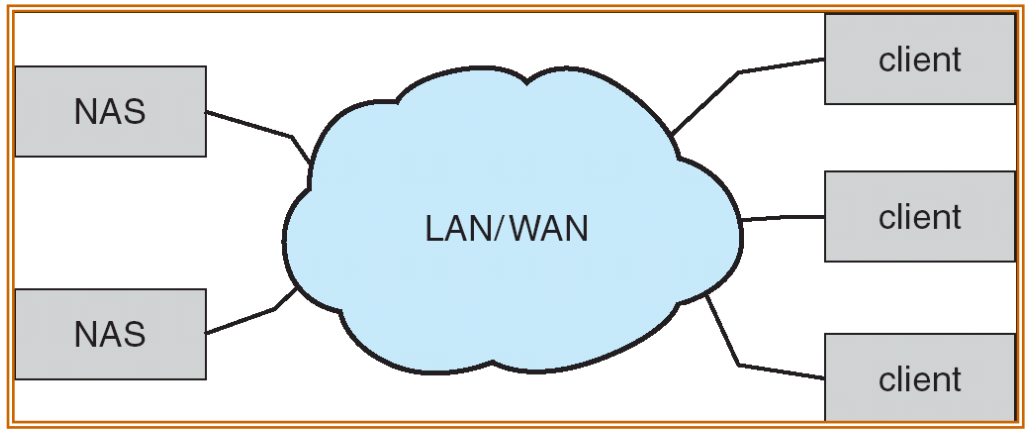
4.7. „Network-Attached Storage“ und Speichernetze

Bei kleinen Systemen wird der Hintergrundspeicher (z.B.: Festplatte) an den einzelnen Computer **direkt angeschlossen** bzw. ist in diesen integriert.

In größeren Gesamtsystemen ist es aber häufig sinnvoll, **über Netzwerke** (Local Area Networks, im Einzelfall auch Wide Area Networks) auf den/die Hintergrundspeicher zuzugreifen. Auf diese Weise sind möglich:

- **gemeinsam genutzte Speicherbereiche** (unkritisch bei ausschließlich lesenden Zugriffen)
- **dynamische Zuordnung von Speicherplatz**
  Wer aktuell viel Speicher benötigt, der kann diesen bekommen und später wieder freigeben, womit dieser Speicher für andere Computer verfügbar wird.

Eine besonders einfache Möglichkeit besteht darin, den Speicher als „**Network Attached Storage**“ (NAS) einfach an das oft ohnehin vorhandene **Local Area Network (LAN)** anzuschließen und so für die „Klienten“ verfügbar zu machen.

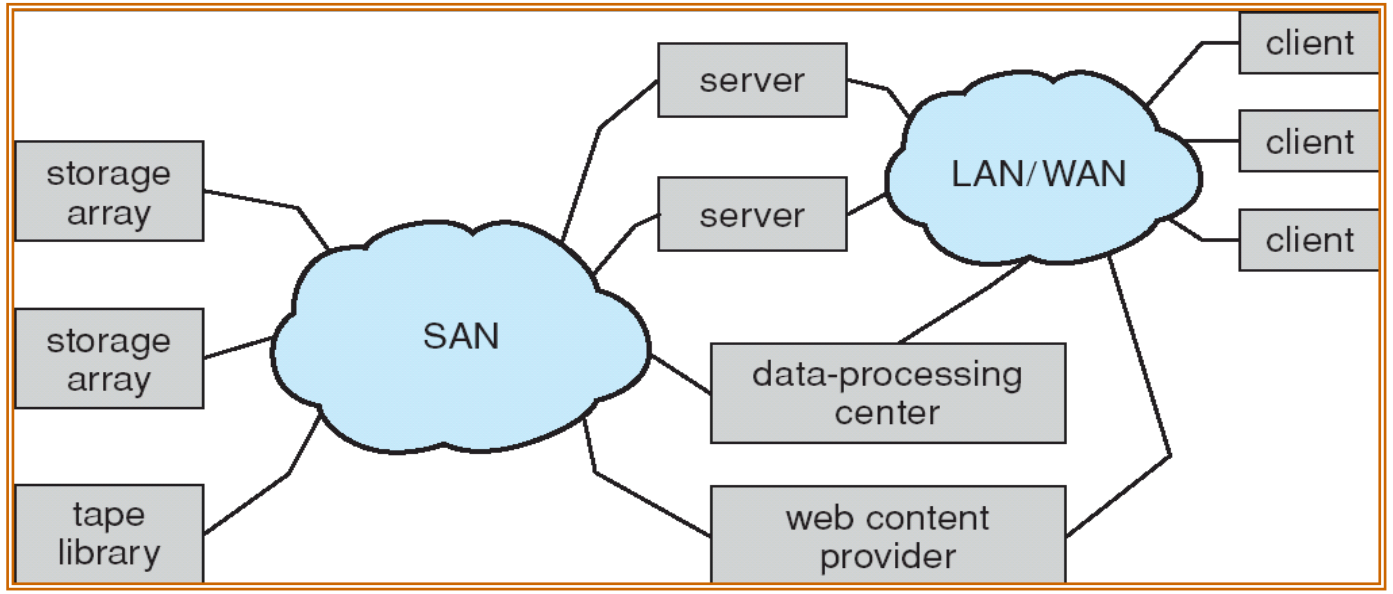


Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 456

Storage Area Network (SAN)

Weitere Verbesserungen ergeben sich durch die heute weit verbreitete Realisierung von „**Storage Area Networks**“ (SANs).

- **Lastreduktion** im LAN/WAN (bzw. Last-Entzerrung)
- **Dedizierte Technologien im SAN** für extrem hohe Datenraten, d.h. Verwendung anderer (und teurerer) Netztechnik (z.B. „Fiber Channel“) als im LAN bzw. WAN üblich.



Source: Silberschatz et al., „Operating System Concepts“, 7th Edition, p. 344

4.8. Dateisystem und Dateiverwaltung

Eine ***Datei*** ist eine Sammlung von Daten, die auf einem **Permanentspeicher** gehalten wird.

Ein ***Dateisystem*** ist nun dafür verantwortlich, Daten auf einem Speichermedium in geeigneter Form **zugänglich zu machen**, ohne dass der Benutzer sich um die Details der internen Datenorganisation kümmern muss.

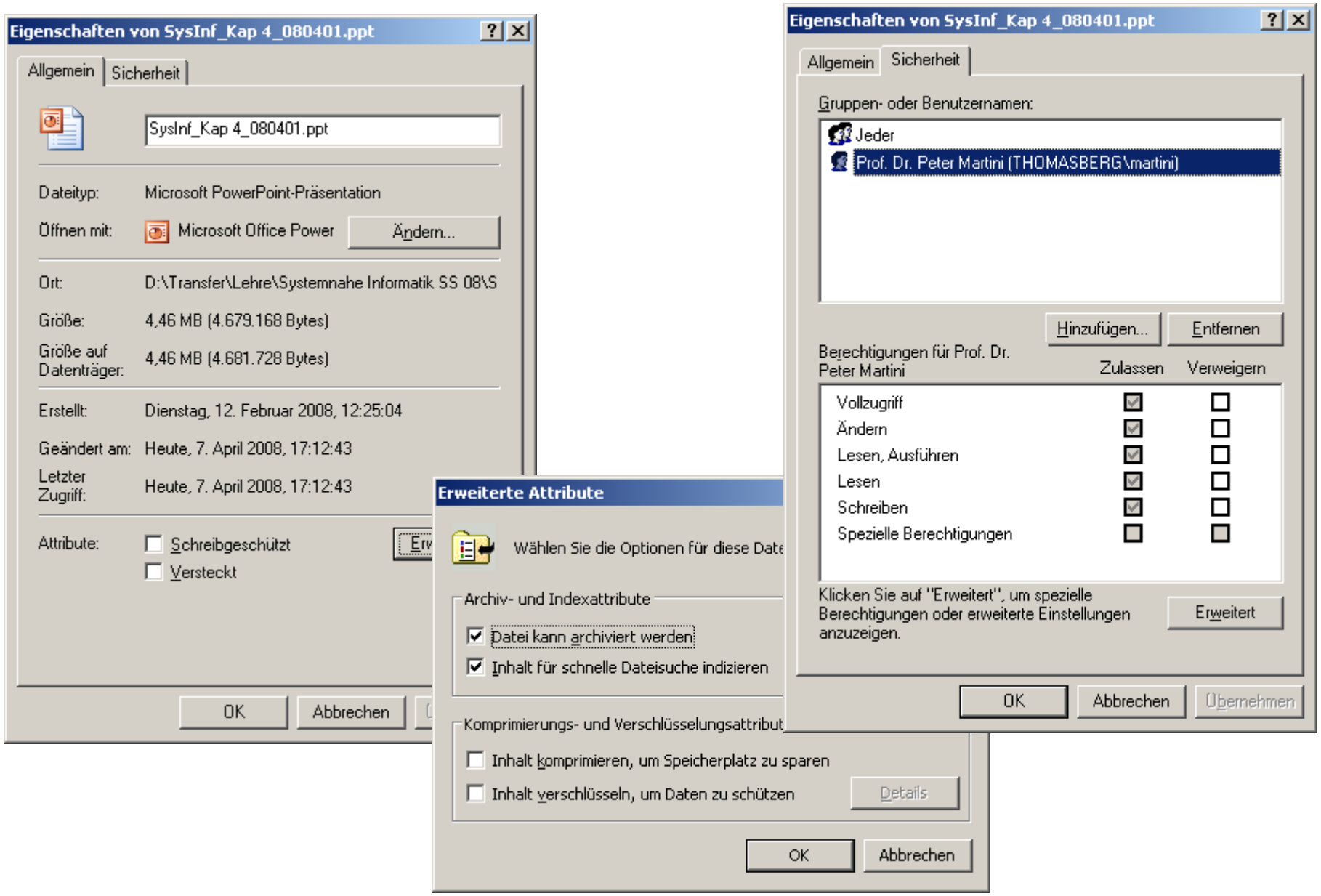
Herold u.a., „Grundlagen der Informatik“, Pearson 2006, S. 383

Im einfachsten (und am häufigsten verwendeten) Fall enthält eine Datei (engl.: „File“) lediglich eine **lineare Folge von Bytes** bzw. „Worten“ des Computersystems.

Zu jeder Datei pflegt das Betriebssystem **umfangreiche Verwaltungsdaten**, viel mehr, als den meisten Benutzern bewusst ist:

- Name der Datei
- Besitzer der Datei
- Art der Datei (z.B. ppt)
- Zugriffsrechte
- Größe der Datei
- Letzte Änderung
- Ort der Speicherung
- Letzter Zugriff auf die Datei
- Zuletzt gedruckt von …
- Zahl der Absätze, Worte, Zeichen, Zeilen in der Datei
- Ursprünglicher Ersteller der Datei
- …

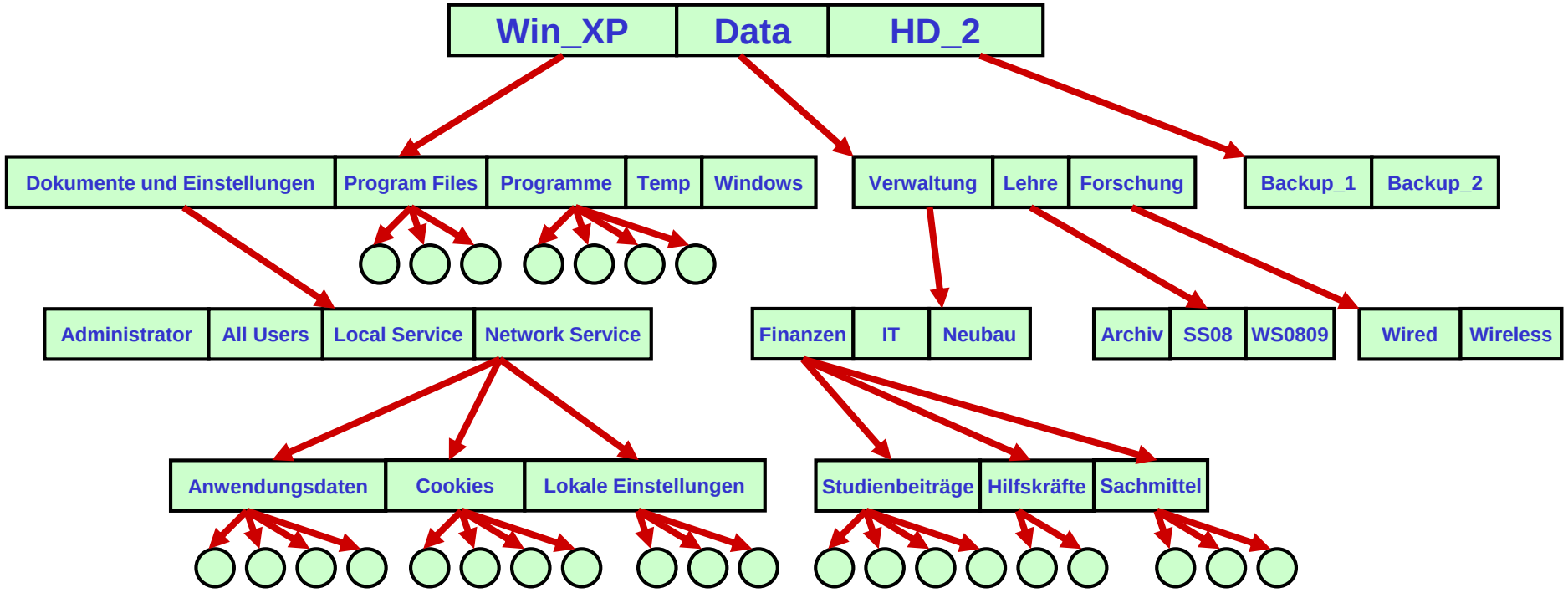
Beispiel: Attribute der Datei SysInf_Kap 4_080401.ppt



Das Verzeichnis (Directory)

Um den Überblick und den Zugriff auf Dateien zu erleichtern, werden die Dateien meist in Unterverzeichnissen bzw. „**Ordnern**“ (engl. Folder) strukturiert.

Häufig werden **Baumstrukturen** verwendet, die dadurch entstehen, dass unterhalb eines Wurzelverzeichnisses (root directory) Teilverzeichnisse angelegt werden, in denen wiederum Teilverzeichnisse liegen können. Die reine Baumstruktur kann oft mit „**Verknüpfungen**“ durchbrochen werden.



„Öffnen“ von Dateien, „Mounten“ von Verzeichnissen

Öffnen von Dateien:

Dateien müssen meist vor dem Zugriff durch einen Systemaufruf der Art `open()` verfügbar gemacht werden. Auf diese Art kann das Betriebssystem u.a. die **Zugriffsrechte prüfen und Zugriffskonflikte erkennen** bzw. verhindern.

Mounten von Verzeichnissen:

In ähnlicher Weise müssen ggf. Teilverzeichnisse zuerst „ge-mounted“ werden, bevor sie verwendet werden können. Konkret handelt es sich hierbei um das **Einfügen „fremder“ Verzeichnisstrukturen** (insbes. Verzeichnisse auf anderen Computern) in das eigene Verzeichnis/Directory.

Auch hierbei sind **Sicherheitsbelange** massiv berührt.

4.9. Zusammenfassung (Kapitel 4)

- Die Speicherverwaltung hält Informationen zur aktuellen Speicherbelegung bereit, vergibt und entzieht Speicherplatz.
- Der Speicher ist hierarchisch aufgebaut: Extrem schnelle Speicher (ihrerseits hierarchisch) innerhalb der CPU, Hauptspeicher, Hintergrundspeicher, ggf. externe Server.
- Nach dem Lokalitätsprinzip liegt die Vermutung nahe, das kürzlich verwendete Daten bald wieder benötigt werden – und somit in schnellen Speichern gehalten werden sollten.
- Virtuelle Speicher befreien den logischen Adressraum von den Einschränkungen aufgrund der real vorhandenen Hardware. Nur die jeweils aktuellen Seiten werden in den Hauptspeicher eingelagert (in sog. „Seitenrahmen“).
- Die Seitentabellen können sehr groß werden, was Strukturierung erforderlich machen kann.
- Translation Look-aside Buffer sind sehr schnelle Assoziativspeicher, die aktuelle Seiteneinträge schneller bereit stellen, als die „normale“ Seitentabelle.
- Seitenfehler kosten extrem viel Zeit. Sie dürfen daher nur extrem selten auftreten.
- Als „Demand Paging“ bezeichnet man eine Strategie, welche neue Seiten erst dann in den Hauptspeicher einlagert, wenn dies aufgrund eines Seitenfehlers unvermeidlich ist.
- Bei der Speicherverwaltung muss ein Kompromiss gefunden werden zwischen „viele Prozesse mit jeweils wenig Speicher“ und „wenige Prozesse mit jeweils viel Speicher“.
- Das Dateisystem ist zuständig für die Organisation der langfristigen Datenhaltung.