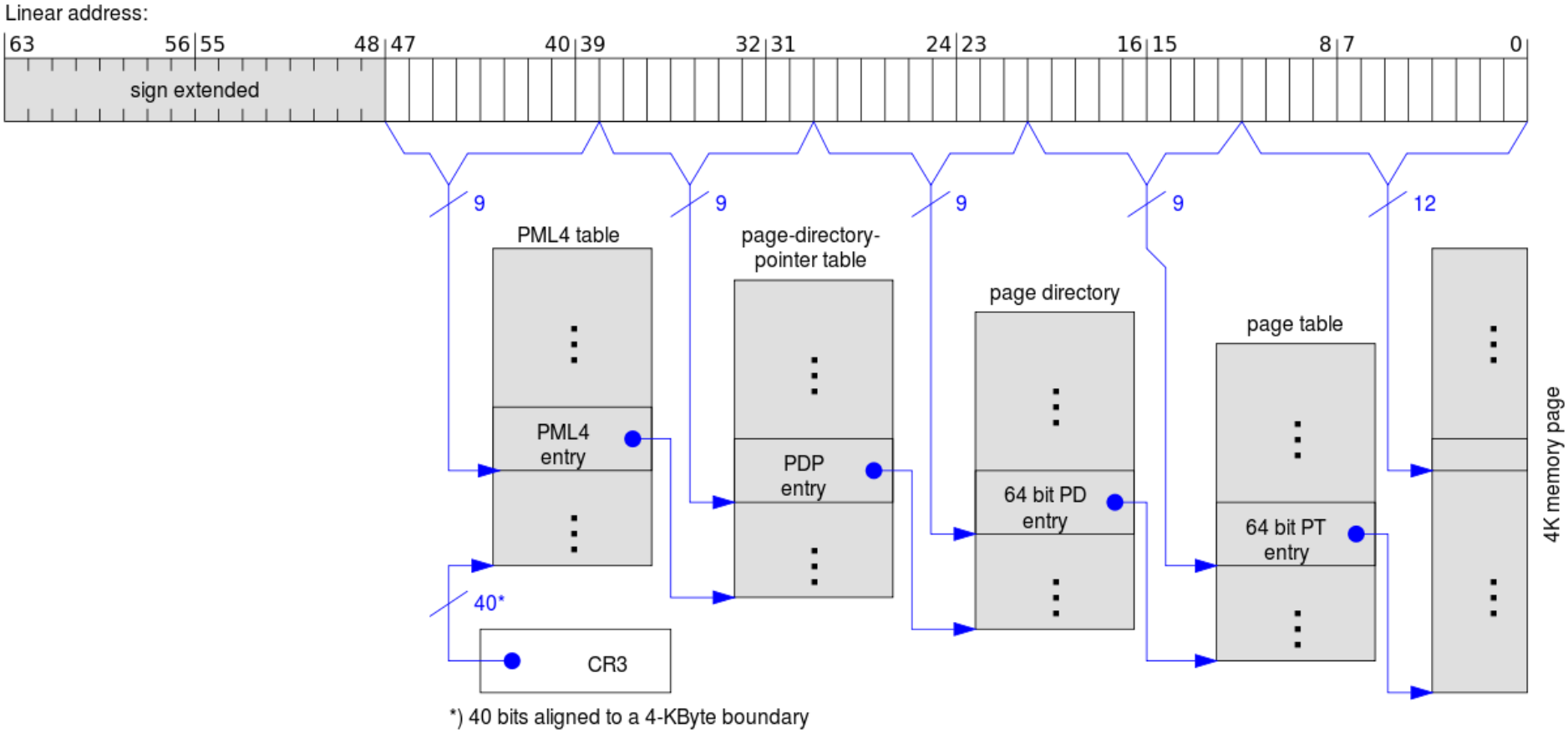



# Speicherverwaltung

**Grafik: Wikipedia**

# Grundlagen

Speicher ist ein unbedingt notwendiges **Betriebsmittel** für die Ausführung von Prozessen. Er dient

- der Bereitstellung von Programmcode und Laufzeitdaten für die **Ausführung** durch die CPU (Primärspeicher)
- zur **Ablage** und zeitnahen **Verfügbarmachung** des Programmcodes und seiner persistenten Daten auch ohne permanente **Energieversorgung** (Sekundärspeicher),
- zur **Langzeitarchivierung** und Datensicherung (Tertiärspeicher)

# Grundlagen

Speicher sind adressierbar, d.h. sie sind in Einheiten fester Größe unterteilt, von denen jede dieser Einheiten einzeln durch eine fortlaufende Zahl (Adresse) identifiziert ist.

- Bei **Massenspeichern** werden **Blöcke** adressiert. Ihre Größe bewegt sich zwischen 512 Byte (aktuell) und 4096 Byte (zukünftig).

- Bei **kleineren Speichern** (wie auch dem Hauptspeicher eines Rechners) werden **Worte** adressiert. Deren Größe richtet sich nach der der Größe der durch die CPU in einem Befehl verarbeitbaren Daten (Wortbreite). Heutzutage beträgt diese Wortbreite 64 Bit, also 8 Byte.

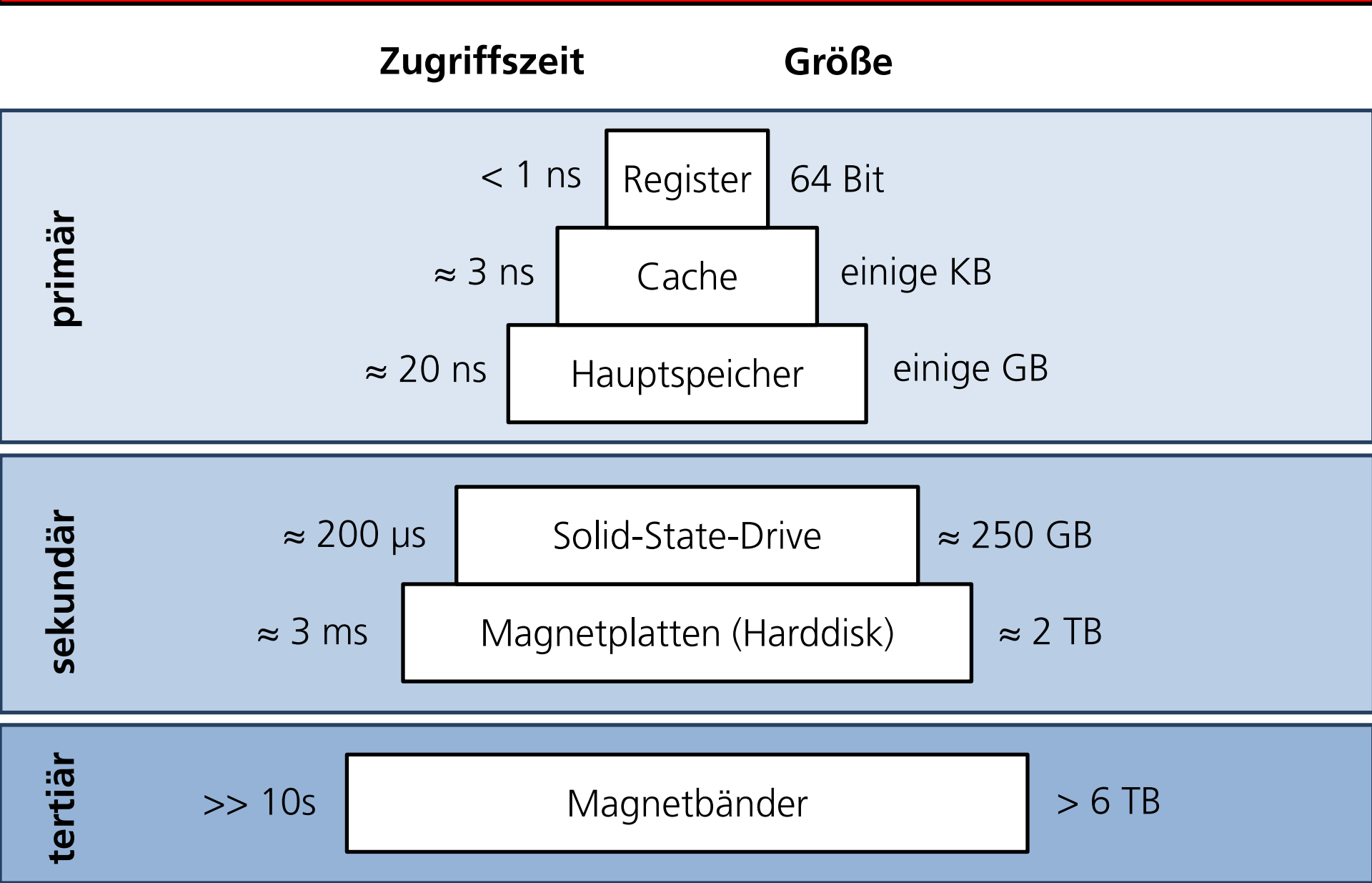
# Speicherhierarchie

Der ideale Speicher hätte sehr kleine **Zugriffszeiten**, sehr große **Kapazität**, könnte **persistent** speichern und wäre sehr **kostengünstig** verfügbar. Einen solchen Speicher gibt es bisher nicht.

Stattdessen ist schneller Speicher meist teuer und klein. Daher wird in typischen Rechnersystemen eine **Speicherhierarchie** aus unterschiedlichen Speichertechniken aufgebaut, die mit wachsendem Abstand zur CPU immer größere Kapazität und immer langsamere Zugriffszeiten besitzen.

**Merkregel:** Primärspeicher sind klein, schnell und teuer, Sekundärspeicher und Tertiärspeicher sind groß, langsam und günstig.

# Speicherhierarchie



# Speicherverwaltung – Fokus dieser Kurseinheit

- Die Verwaltung von **Sekundär-** und **Tertiärspeicher** übernimmt das **Dateisystem** (eigene Kurseinheit).
- **Register** bedürfen keiner gesonderten Verwaltung, sie sind in den Anweisungen an die CPU fest codiert.
- In dieser Kurseinheit fokussieren wir daher auf die Verwaltung des **Hauptspeichers** (*Random-Access-Memory*, RAM), also desjenigen Primärspeichers, der Programmcode und –daten für die Ausführung durch die CPU bereithält.

# Speicherverwaltung

Solange nur ein einziger Prozess zur Zeit auf dem betrachteten System läuft, dessen Speicherbedarf durch den maximal verfügbaren Hauptspeicher abgedeckt ist, braucht es im Prinzip keine gesonderte Verwaltung des Speichers durch das Betriebssystem.

# Speicherverwaltung

Diese Situation taucht heute in der Praxis selten auf, Ausnahmen bilden u.A. Batchsysteme oder eingebettete Systeme. Stattdessen

- konkurrieren mehrere, nebenläufige Prozesse um das Betriebsmittel Hauptspeicher,
- kann der Gesamtbedarf an Speicher durch ebendiese Prozesse den zur Verfügung stehenden Hauptspeicher leicht überschreiten.

Es wird also eine Verwaltung der aktuell belegten Speichers und der verbleibenden freien Speicherbereiche benötigt.

# Speicherverwaltung – Aufgaben

Die **Aufgaben** der Speicherverwaltung sind daher:

1. freien und belegten Speicher verwalten,
2. Speicherbereiche den Prozessen dynamisch zuteilen,
3. den gegenseitigen Zugriff auf die den Prozessen zugewiesenen Speicherbereiche einschränken oder verhindern,
4. logische und virtuelle Adressräume verwalten,
5. Speicherbereiche auf Sekundärspeicher auslagern und von dort wieder einlagern.

# Adressierungsvarianten

Um in einem Kommando darzustellen, auf welche Speicherstelle(n) zugegriffen werden soll, gibt es unterschiedliche Varianten. Für diesen Kurs sind vier Ansätze*) von Bedeutung:

1. **Direkte Adressierung** – Die Adresse wird explizit angegeben.

2. **Indirekte Adressierung** – Die Adresse, auf die zugegriffen werden soll, steht an anderer Stelle. Dies kann entweder eine weitere Speicherstelle oder aber ein Register(-paar) sein.

*) Tatsächlich gibt es eine Vielzahl von Adressierungsvarianten! Wir gehen in diesem Kurs nur auf einen Ausschnitt ein.

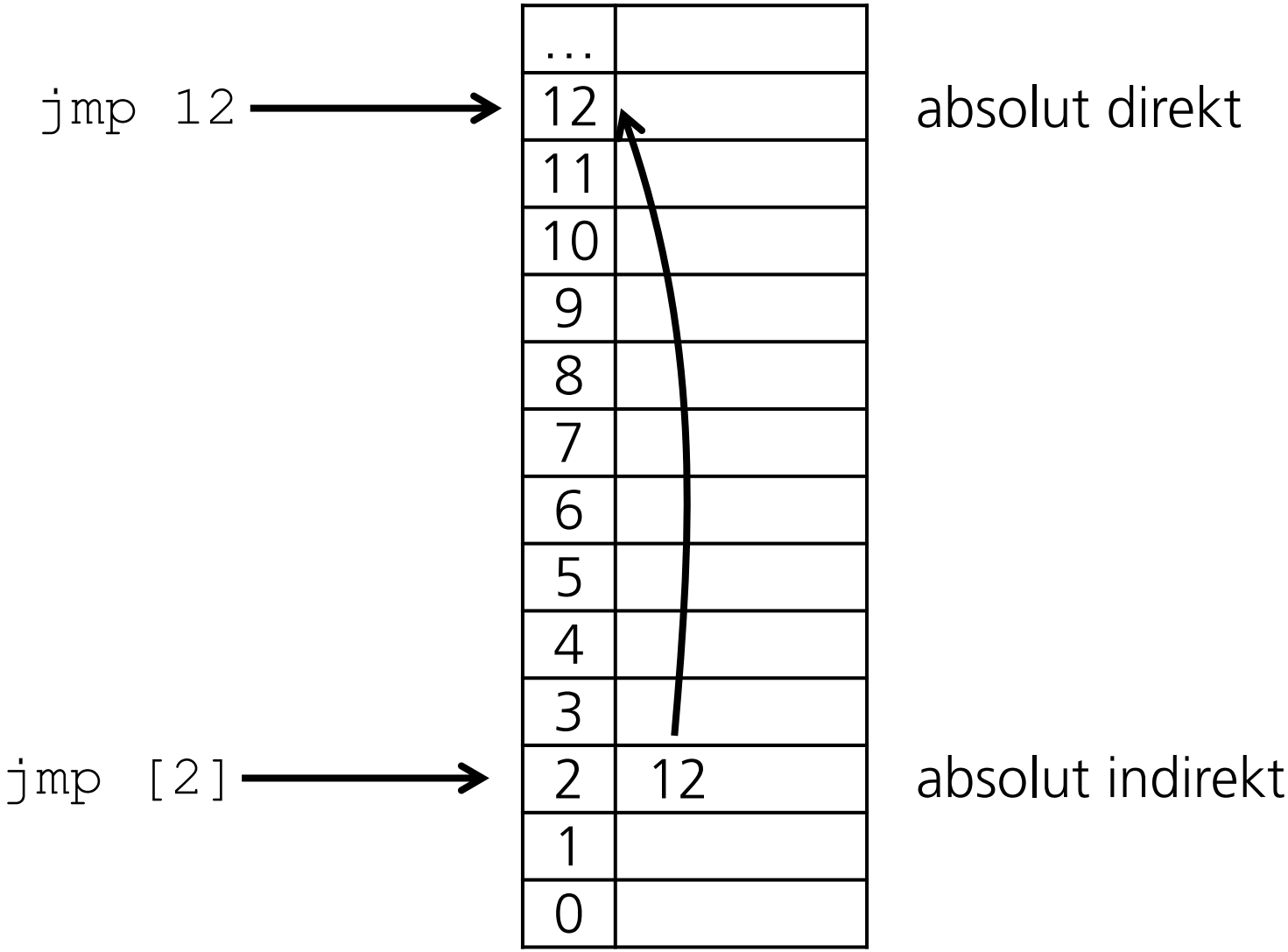
# Adressierungsvarianten

3. **Absolute Adressierung** – Die Adresse wird vollständig und ohne weitere Bezugsadressen angegeben.

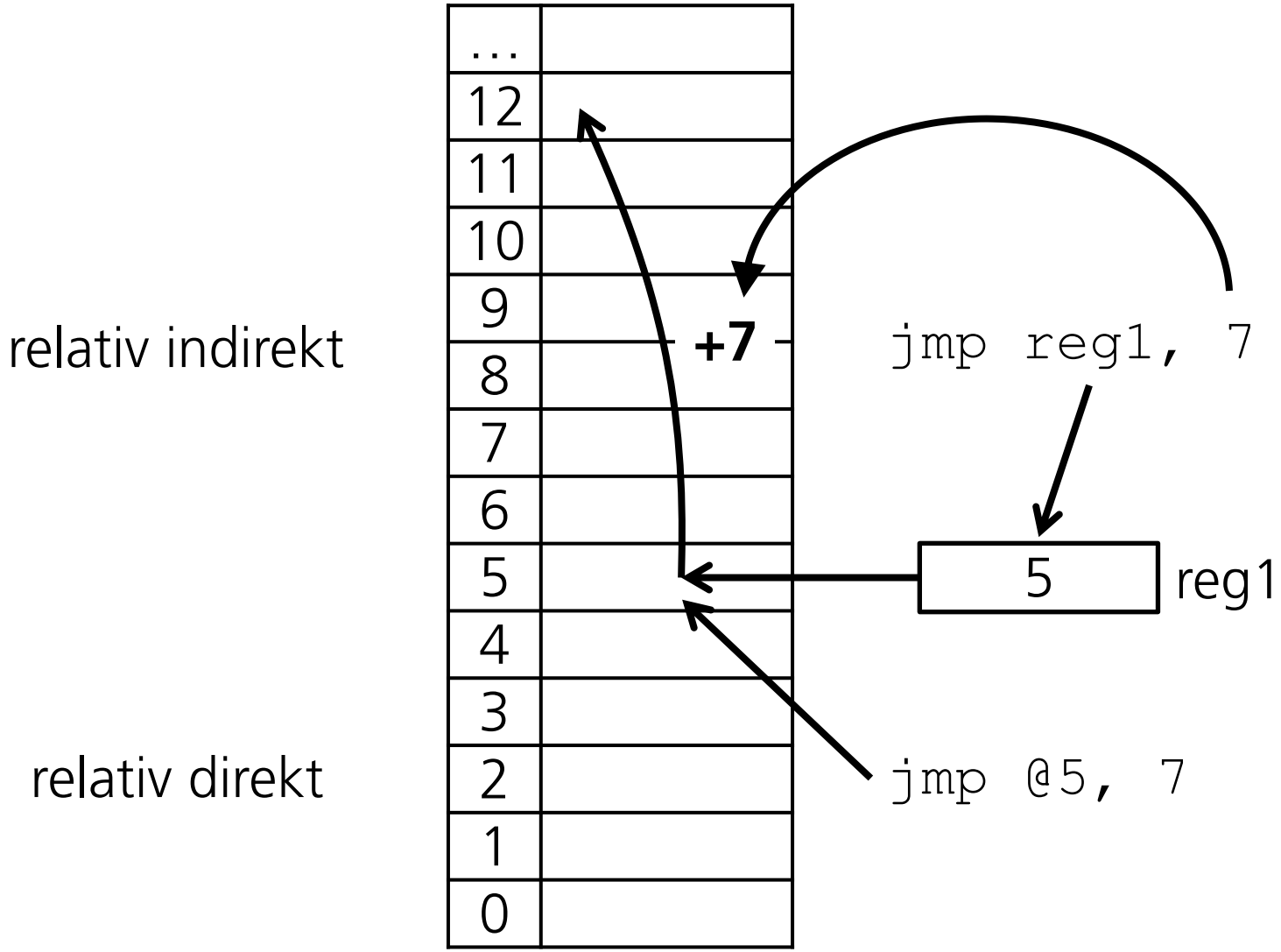
4. **Relative Adressierung** – Die Adresse wird als Distanz zu einer Referenzadresse angegeben.

Direkte/indirekte und absolute/relative Adressierung können paarweise kombiniert werden.

# Adressierungsvarianten – Beispiele



# Adressierungsvarianten – Beispiele



# Adressierung

- Adressen haben eine **feste Länge**, die durch die Rechnerarchitektur (Anzahl der Leitungen im Adressbus) vorgegeben ist.
  - 32 Bit Adressen (z.B. Intel 80486 bis 2007) erlaubten maximal 4GiB adressierbaren Speicher,
  - 64 Bit Adressen (aktuell) adressieren 16 EiB.
- Der physikalisch **vorhandene** Speicher ist typischerweise deutlich kleiner als der theoretisch **adressierbare** Raum. Dies ermöglicht u.a. **Virtualisierung** (siehe weiterer Kurstext).

# Segmentierung

- Segmentierung ist ein Speicherverwaltungsansatz, bei dem Speicherbereiche (Segmente), die Prozessen zugeordnet sind, durch ihre **Startadresse (Basis)** und ihre **Länge** festgelegt sind.
- Durch entsprechende Hardwareunterstützung ist es möglich, jedes dieser Segmente mit **Zugriffsrechten** und **Priviligierungsstufen** zu versehen.
- Die Segmentinformationen werden in einer entsprechenden **Segmenttabelle** durch das Betriebssystem verwaltet und von der CPU (genauer: deren **Speicherverwaltungseinheit**) ausgewertet.

# Segmentierung

- Eine **Adresse** besteht bei der Segmentierung aus einem Segmentteil und einem Offset (Position innerhalb des Segmentes). Typische Aufteilungen sind
  - bei 32-Bit-Adressen: 16 Bit Segment, 16 Bit Offset
  - bei 64-Bit-Adressen: 32 Bit Segment, 32 Bit Offset
- Der **Segmentteil** beinhaltet die **Segmentnummer**, mit deren Hilfe in der Segmenttabelle die **Startadresse** ermittelt werden kann.
- Die **Segmentlänge** ist durch das maximal mögliche Offset begrenzt. Bei 16 Bit sind maximal 64KiB große Segmente zulässig, bei 32 Bit 4GiB.

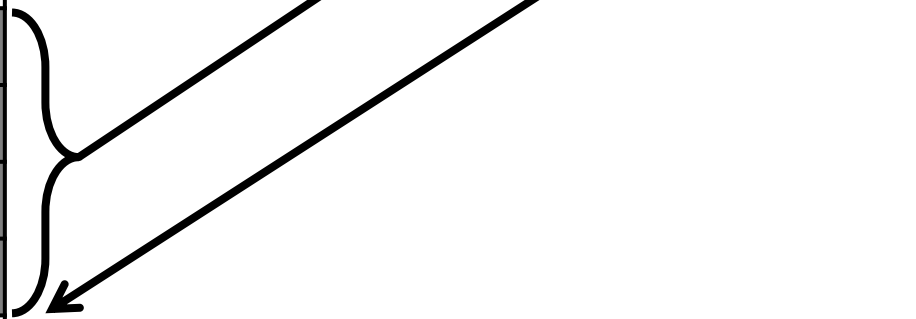
# Segmenttabelle

## Hauptspeicher

| | |
|---|---|
| … | |
| 0xD000 | |
| 0xC000 | |
| 0xB000 | |
| 0xA000 | |
| 0x9000 | |
| 0x8000 | |
| 0x7000 | |
| 0x6000 | |
| 0x5000 | |
| 0x4000 | |
| 0x3000 | |
| 0x1000 | |
| 0x0000 | |

## Segmenttabelle

| Nr. | Eigenschaften | Basis |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | RW, LEN=16K | 0x1000 |
| 4 | | |
| … | | |



# Logische Adressen

Adressen werden bei der Segmentierung als **logische Adressen** bezeichnet. Sie werden durch die Speicherverwaltungseinheit der CPU (*Memory Management Unit*, *MMU*) in physikalische Adressen übersetzt.

Dazu verwendet die MMU den Segmentteil, um die **Startadresse** in der Segmenttabelle zu ermitteln und addiert auf diese den Offset, um die physikalische Adresse zu erhalten.

# Logische Adressen – Beispiel

**Beispiel:** Es werden 32-Bit-Adressen verwendet, die Segmenttabelle habe die Einträge (Ausschnitt)

| Nr. | Eigenschaften | Basis |
|---|---|---|
| … | | |
| 0x0018 | RW, LEN=16K | 0x00100000 |
| 0x0019 | R, LEN=32K | 0x0F800000 |
| … | | |

Die Adresse 0x0018A100 (entspricht $1614080_{10}$) wird in einen 16 Bit Segmentteil von 0x0018 und einen 16 Bit Offset von 0xA100 zerlegt:

# Logische Adressen – Beispiel

Die **Basis** des Segmentes Nummer 0x0018 ist laut Segmenttabelle bei 0x00100000 (entspricht $1048576_{10}$).

| Nr. | Eigenschaften | Basis |
|---|---|---|
| … | | |
| **0x0018** | RW, LEN=16K | **0x00100000** |
| 0x0019 | R, LEN=32K | 0x0F800000 |
| … | | |

Die **physikalische** Adresse lautet also:

$$0x0010000 + 0xA100 = 0x0010A100 = 1089792_{10}$$

# Zugriffsbeschränkungen bei Segmentierung

- Zugriffe auf den Speicher sind Prozessen im Regelfall nur **innerhalb** ihrer Segmente erlaubt. Zugriffe außerhalb der Segmente werden durch die MMU erkannt und lösen einen **Softwareinterrupt** aus.
- Ebenso lösen Zugriffe, die nicht den Zugriffsrechten oder der Priviligierungsstufe des Segments entsprechen, einen Softwareinterrupt aus.
  - **Beispiel:** Ein Prozess versucht, schreibend auf ein Segment zuzugreifen, das er nur lesen darf.
- Die bekannteste Meldung bei unerlaubten Zugriffen auf Speicher ist die „**allgemeine Schutzverletzung**“ von Windows.

# Segmentierung und indirekte Adressierung

Segmentierung unterstützt in besonderem Maße die **indirekte** Adressierung:

- Die **Segmentnummer** wird nicht in der Adresse selbst, sondern in einem **Register** abgelegt. Intel-CPUs bieten dafür ein Register `CS` für Codesegmente und ein Register `DS` für Datensegmente.

- Der Prozess verwendet für alle Zugriffe innerhalb des Segmentes **indirekte relative** Adressierung, muss also nur das Offset explizit angeben. Es muss also nur die **Hälfte** der Adresse explizit angegeben werden, der jeweilige Befehl nimmt weniger Speicher ein.

# Hinweis

**Hinweis:** Die praktische Umsetzung der Segmentierung unterscheidet sich bei realen Prozessoren in einigen Details von der hier gewählten Darstellung. Aus Gründen der Überschaubarkeit haben wir uns hier für eine etwas modellhaftere Erläuterung entschieden, die Ihnen das grundlegende Verständnis der Segmentierung ermöglichen soll.

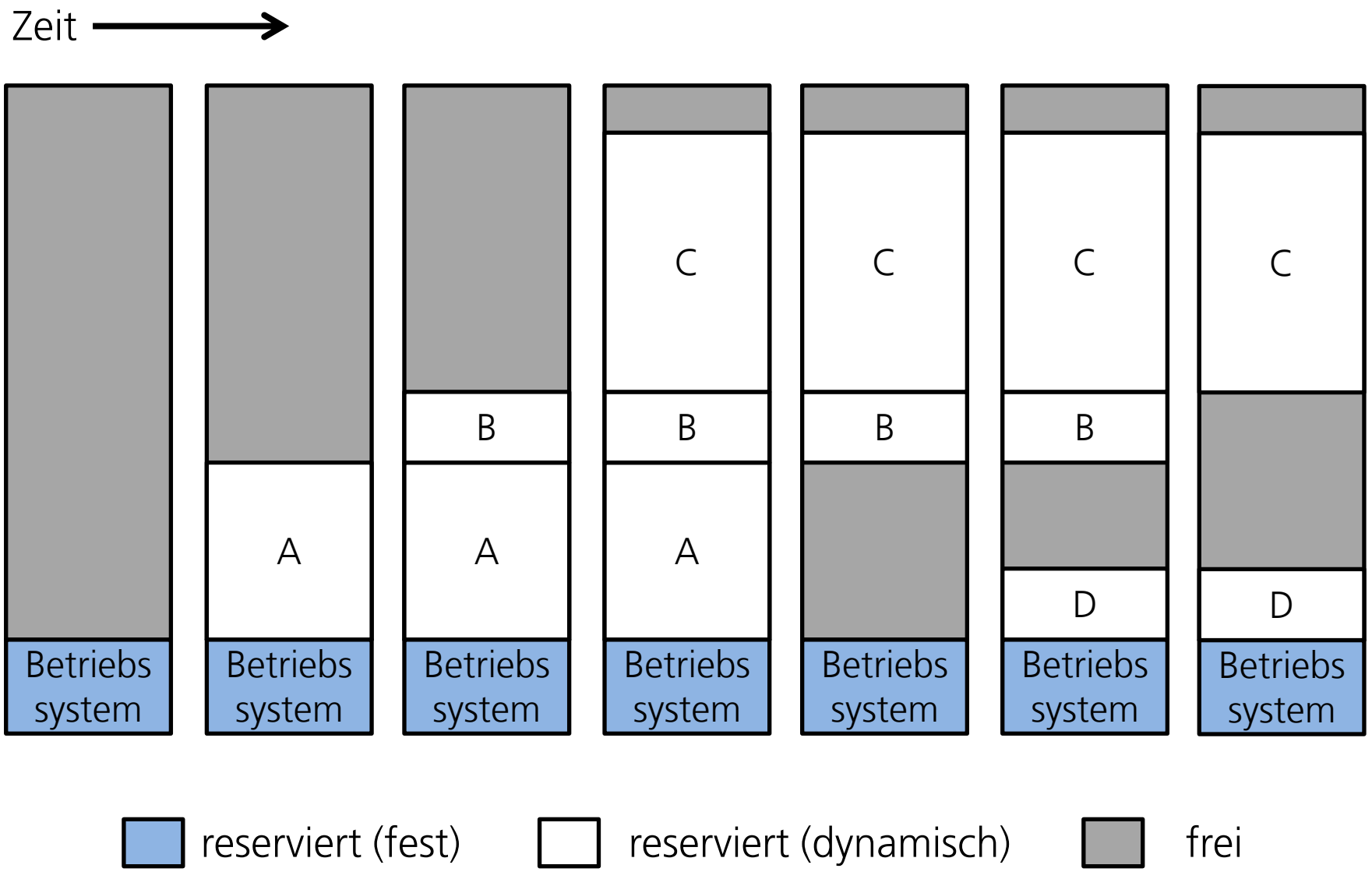
# Verschnitt und Fragmentierung

Durch dynamische Speicherzuweisung (wie sie bei der Segmentierung möglich ist) können Speicherbereiche ungenutzt bleiben. Man unterscheidet hier zwei Effekte:

- **Verschnitt** entsteht, wenn ein Prozess den ihm zugewiesenen Speicherbereich nicht voll ausschöpft. Dies geschieht bspw. dann, wenn Speicher nur in festen Größen vergeben wird (siehe *Paging* im weiteren Verlauf des Kurstexts).
- Als **Fragmentierung** wird die durch dynamische Belegung entstehende Zerteilung des freien Speichers in mehrere, kleinere Teilbereiche verstanden.

# Fragmentierung

Grafik nach Tanenbaum



# Verschnitt und Fragmentierung

- **Verschnitt** lässt sich reduzieren, indem die verwendete **Blockgröße** verkleinert wird. Dies führt allerdings zu erhöhtem Speicheraufwand für die Verwaltung. In der Praxis werden hier Kompromisse eingegangen.

- **Fragmentierung** lässt sich durch **Kompaktifizierung** erreichen, also durch Verschieben der Prozesse, sodass ein einzelner freier Speicherbereich verbleibt.
  - Allerdings müssen dazu die Prozesse zur **Laufzeit verschiebbar** (relozierbar) sein!
  - Die Kompaktifizierung ist **aufwändig** (kostet Zeit). Daher wird sie möglichst vermieden und durch geeignete **Belegungsstrategien** kompensiert.

# Belegungsstrategien

Fragmentierung kann durch den Einsatz geeigneter Strategien beim Belegen freien Speichers gemildert werden

**Ziele:**

1. **Anzahl** der freien Bereiche minimieren,
2. **Größe** der freien Bereiche maximieren

# Belegungsstrategien

Benötigt ein Prozess Speicher, kann das Betriebssystem nach verschiedenen Strategien ein passendes Stück auswählen.

- **FirstFit** – Gehe die Liste der freien Speicherbereiche durch und wähle das erste ausreichend große Stück.

  **Vorteil:** Sehr schnell

  **Nachteil:** Es entstehen ggf. unbrauchbare Reststücke

- **NextFit** – Wie FirstFit, aber beginne die Suche hinter dem Bereich, der als letztes zugeteilt wurde.

  **Vorteil:** Bessere Ausnutzung des gesamten Speichers

  **Nachteil:** Immer noch großes Risiko für Reststücke

# Belegungsstrategien

- **BestFit** – Führe eine Liste über alle freien Stücke und wähle bei einer Anfrage das kleinste passende freie Stück.

  **Vorteil:** Restlücken werden minimiert

  **Nachteil:** Aufwand bei der Verwaltung

- **WorstFit** – Wie Bestfit, aber wähle das größte verbleibende freie Stück, um große Reststücke zu erreichen, die weiteren Prozessen zugeordnet werden können.

  **Vorteil:** Restlücken werden minimiert

  **Nachteil:** Prozesse mit großem Speicherbedarf werden schlechter bedient, weil große Lücken systematisch verkleinert werden.

# Belegungsstrategien

- **QuickFit:** Führe für verschiedene potentielle Anfragegrößen jeweils eine Liste der am besten geeigneten verbleibenden Freispeicherstücke.

  **Vorteil:** Restlücken werden minimiert, Anfragen können schnell beantwortet werden

  **Nachteil:** Aufwand bei der Verwaltung

**Anaylse:** In Sachen Ausnutzung des Speichers und Verringerung von Fragmentierung ergibt sich die Rangfolge

FirstFit > NextFit > QuickFit > BestFit

BestFit schneidet interessanterweise besonders schlecht ab. Die Ursache liegt darin, dass dieser Ansatz viele kleine **Reststücke** produziert, die nicht weiter verwendbar sind.

# Belegungsstrategien – Buddy-Prinzip

**Ansatz:** Teile den Speicher gedanklich in möglichst große Teilstücken der Größe $2^n$. Bei einer Anfrage wird stets versucht, ein Speicherstück bereitzustellen, dessen Größe eine Zweierpotenz entspricht und das die Anfrage gerade abdecken kann.

- Existiert aktuell kein solches (kleinstmögliches) Stück, wird ein **doppelt** so großes Stück gesucht und in zwei gleiche Teile geteilt. Existiert ein solches ebenfalls nicht, wird ein vierfach so großes Stück gesucht, usw.
- Wird ein Speicherstück wieder frei, kann es mit einem **benachbart** (*buddy* = *Kumpel*) liegenden Speicherstück gleicher Größe wieder **zusammengefasst** werden.

**Vorteile:** Schnell, keine Hardwareunterstützung nötig

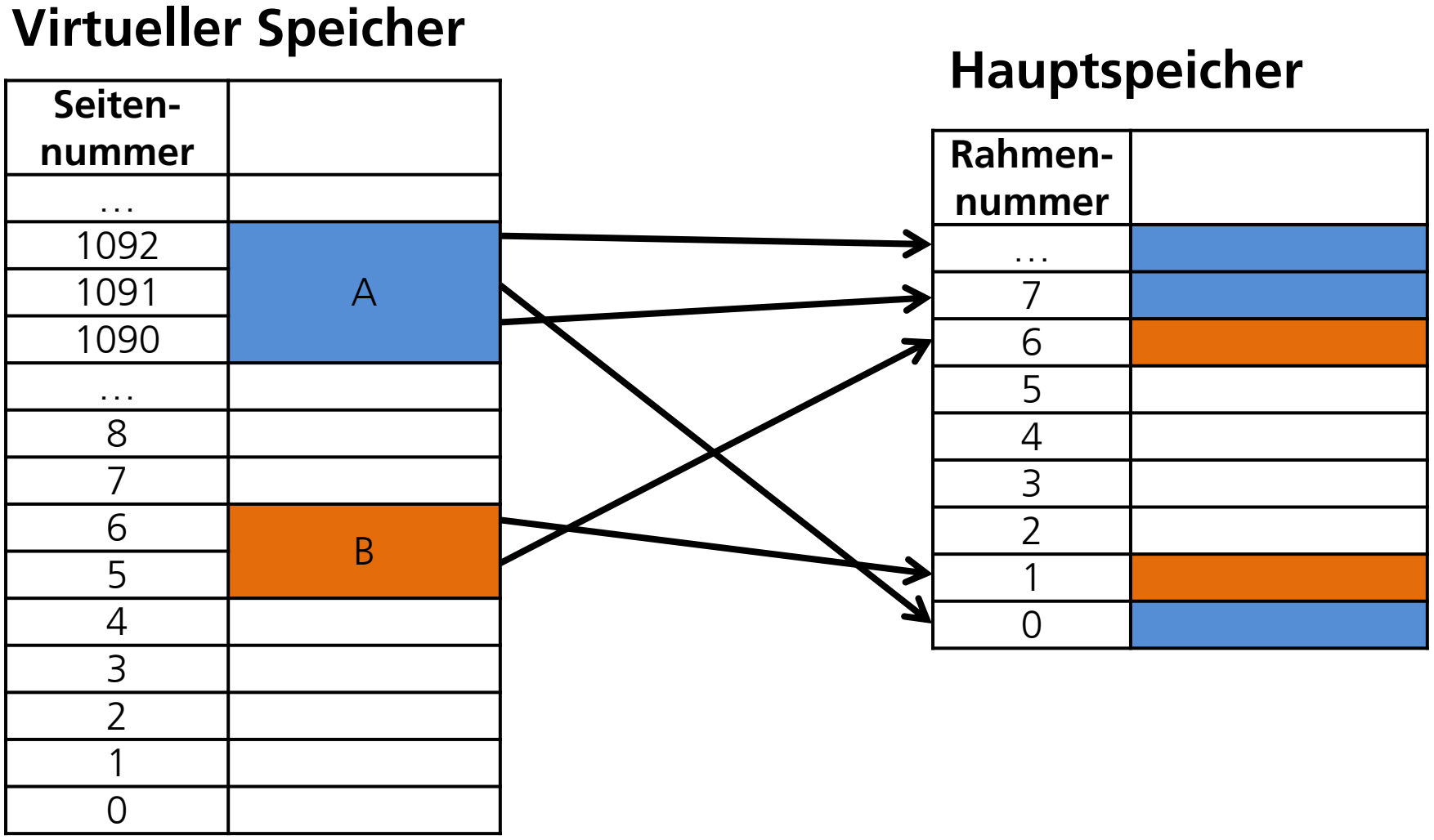
**Nachteil:** Anfällig für beide Arten der Fragmentierung

# Virtueller Speicher durch Paging

Als **virtueller Speicher** wird ein vom Hauptspeicher unabhängiger linearer Adressraum bezeichnet, der meist wesentlich größer als dieser ist. Prozessen werden im virtuellen Adressraum **zusammenhängende** Bereiche zugeordnet.

Die Abbildung vom virtuellen auf den physikalischen Speicher erfolgt durch die Speicherverwaltungseinheit der CPU (*MMU*). Als Grundlage hierfür wird der virtuelle Speicher in **Seiten** (Pages) identischer Größe unterteilt, der physikalische Speicher in **Seitenrahmen**. Die Aufgabe der Speicherverwaltung besteht nun darin, die Seiten des virtuellen Speichers freien Rahmen des physikalischen Speichers zuzuordnen. Diese Zuordnung nennt man **Paging**.

# Virtueller Speicher durch Paging



# Seitentabelle und Seitengröße

- Die Zuordnung von Seiten im virtuellen Speicher zu Seitenrahmen im physikalischen Speicher wird in einer **Seitentabelle** abgelegt. Diese hat maximal so viele Einträge, wie Seiten insgesamt im virtuellen Speicher existieren. Sie muss **vollständig** im **physikalischen** Speicher residieren!

- Die **Seitengröße** entspricht einer Zweierpotenz. Sie ist ein **Kompromiss** zwischen Vermeidung interner Fragmentierung (bevorzugt kleine Seiten) und Verringerung der Größe der Seitentabelle (große Seiten).

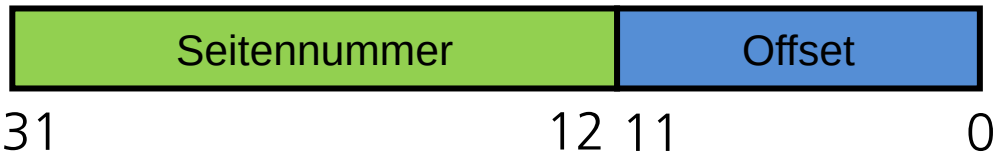
- Bei 32-Bit-Systemen wird meist eine Seitengröße von $2^{12}\,\mathrm{Bytes} = 4\,\mathrm{KiB}$ verwendet, bei 64-Bit-Systemen ist die Seitengröße dynamisch und reicht bis $2^{30}\,\mathrm{Bytes} = 1\,\mathrm{GiB}$.

# Paging und Adressen

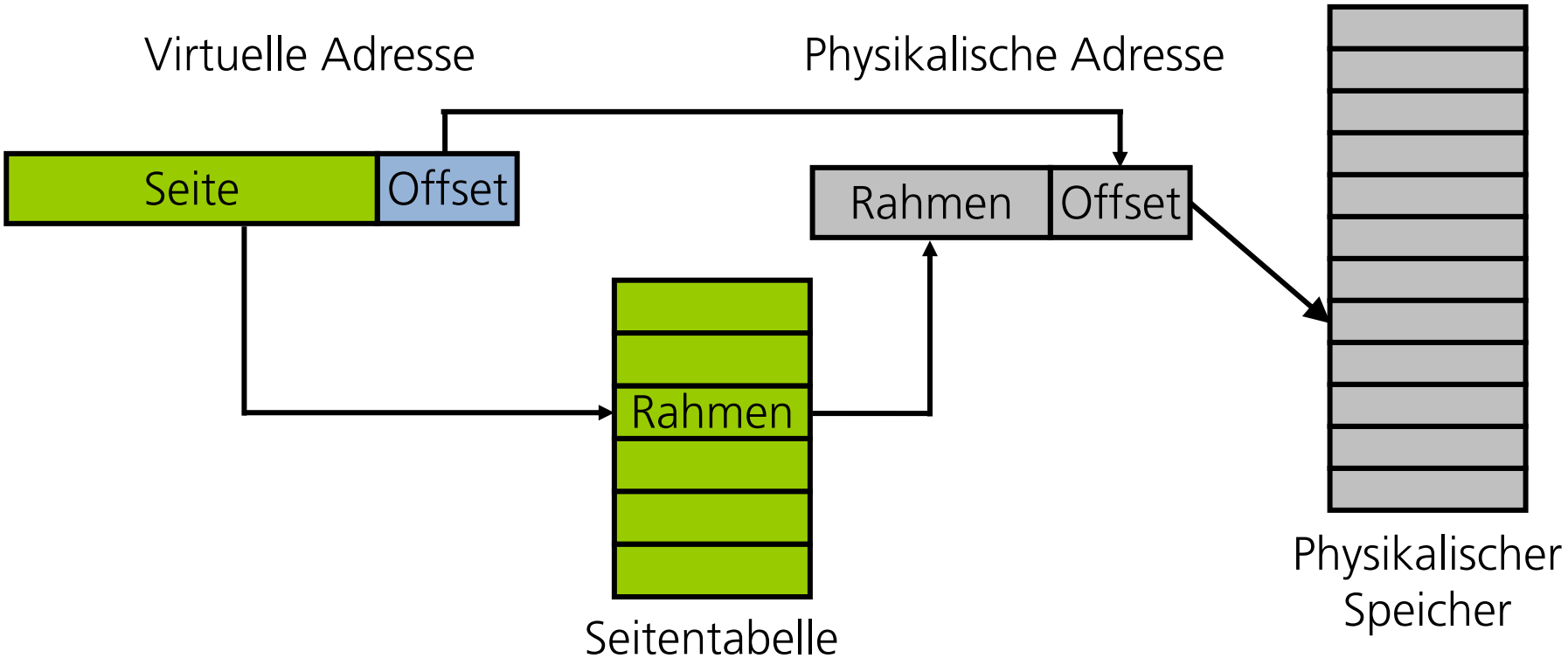
- Ähnlich wie bei der Segmentierung kann man auch beim Paging eine Adresse fester Breite gedanklich in Seitennummer und Offset innerhalb der Seite trennen.

- Durch die gewählte Seitengröße ist fest determiniert, welcher Anteil der Adresse zur Seitennummer und welcher zum Offset gehört.

  **Beispiel:** Bei 32-Bit Systemen und 4 KiB Seitengröße ($= 2^{12}$ Byte) sind die letzten 12 Bit der Adresse das Offset, die vorderen 20 Bit identifizieren die Seitennummer.



# Umsetzung von virtueller in physikalische Adresse



# Umsetzung von virtueller in physikalische Adresse

**Rechenweg:**

1. Teile die virtuelle Adresse durch die Seitengröße. Das ganzzahlige Ergebnis ist die Seitennummer, der ganzzahlige Rest das Offset innerhalb der Seite.

2. Ermittle die Seitenrahmennummer der Seite in der Seitentabelle.

3. Multipliziere die Seitenrahmennummer mit der Seitengröße und addiere den Offset darauf. Das Ergebnis ist die physikalische Adresse, die der virtuellen Adresse zugeordnet ist.

# Paging – Eigenschaften

**Vorteile:**

- Ein einzelner Prozess kann einen **größeren** Speicher adressieren als physikalisch vorhanden ist.
- Alle Prozesse erhalten **linearen** Speicherbereich, externe **Fragmentierung** stellt praktisch kein Problem dar.
- Durch entsprechende Markierungen der Einträge in der Seitentabelle sind **geschützte** Speicherbereiche für exklusiven Zugriff aber auch **gemeinsam genutzte** Speicherbereiche möglich.

**Nachteile:**

- Der **Hardwareaufwand** ist größer, weil Paging durch die CPU unterstützt werden muss.

# Größe der Seitentabelle

Da für **jede verwendete Seite** im virtuellen Speicher ein Eintrag in der Seitentabelle vorliegt, ist deren maximale **Größe** durch die des virtuellen Speichers und die einer einzelnen Seite fest determiniert:

Die Seitentabelle eines virtuellen Speichers der Größe $2^n$ Bytes hat bei einer Seitengröße von $2^k$ Bytes maximal $2^{n-k}$ Einträge.

**Beispiel:** Bei einer Seitengröße von $4\,\mathrm{KiB} = 2^{12}$ Byte ergeben sich:

| Adress-breite | Größe des virtuellen Speichers | Maximale Größe der Seitentabelle |
|---|---|---|
| 16 Bit | $2^{16}$ Bytes $= 64$ KiB | $2^{16-12}$ Einträge $= 16$ Einträge |
| 32 Bit | $2^{32}$ Bytes $= 4$ GiB | $2^{32-12}$ Einträge $\approx 10^6$ Einträge |
| 64 Bit | $2^{64}$Bytes $= 16$ EiB | $2^{64-12}$ Einträge $\approx 4{,}5 \cdot 10^{15}$ Einträge |

# Lösungsalternativen

1. **Adressbreite verkleinern**

   Vorteil: Weiterhin kleine Seiten

   Nachteil: Speichergrenze mit Blick auf externe Fragmentierung schneller erreicht

2. **Seitengröße vergrößern**

   Vorteil: Seitentabelle wird kleiner

   Nachteil: Interne Fragmentierung nimmt zu.

3. **Mehrstufige Seitenverwaltung**

   Vorteile: Feine Granularität der Seiten bleibt erhalten, Seitentabellen bleiben trotzdem klein

   Nachteil: Komplexere Hardwareunterstützung notwendig

# Mehrstufige Seitenverwaltung

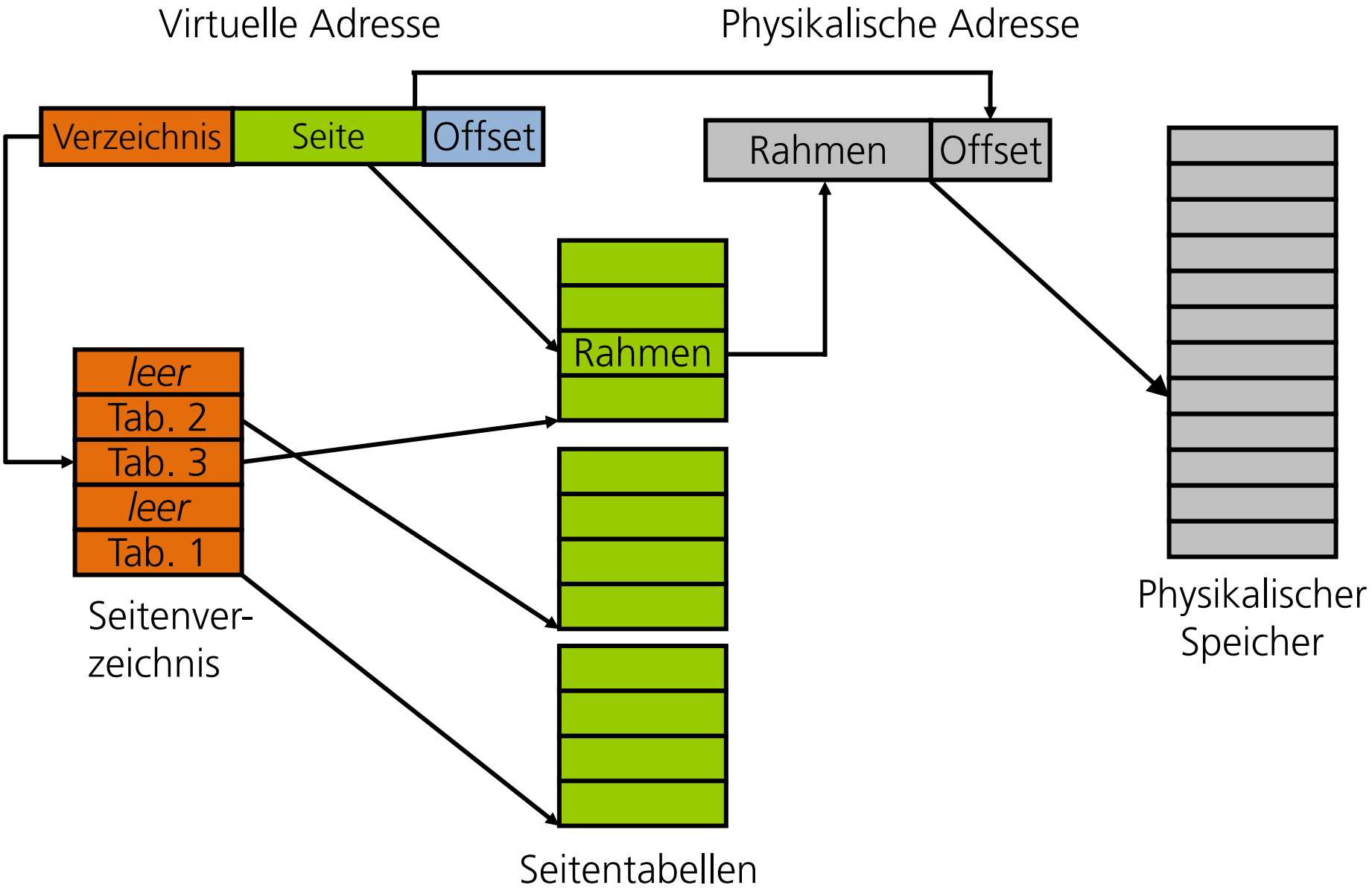
Bei einer mehrstufigen Verwaltung werden **mehrere Seitentabellen** verwendet, die disjunkte Bereiche des virtuellen Speichers abdecken. Ist ein Teil des virtuellen Speichers ungenutzt, wird auch keine Seitentabelle für diesen Teil angelegt.

Alle Seitentabellen werden in einem **Seitenverzeichnis** referenziert. Dieses ist also der **Einstiegspunkt** bei der Umsetzung einer virtuellen in eine reale Adresse.

Der Seitenanteil der virtuellen Adresse wird also unterteilt:

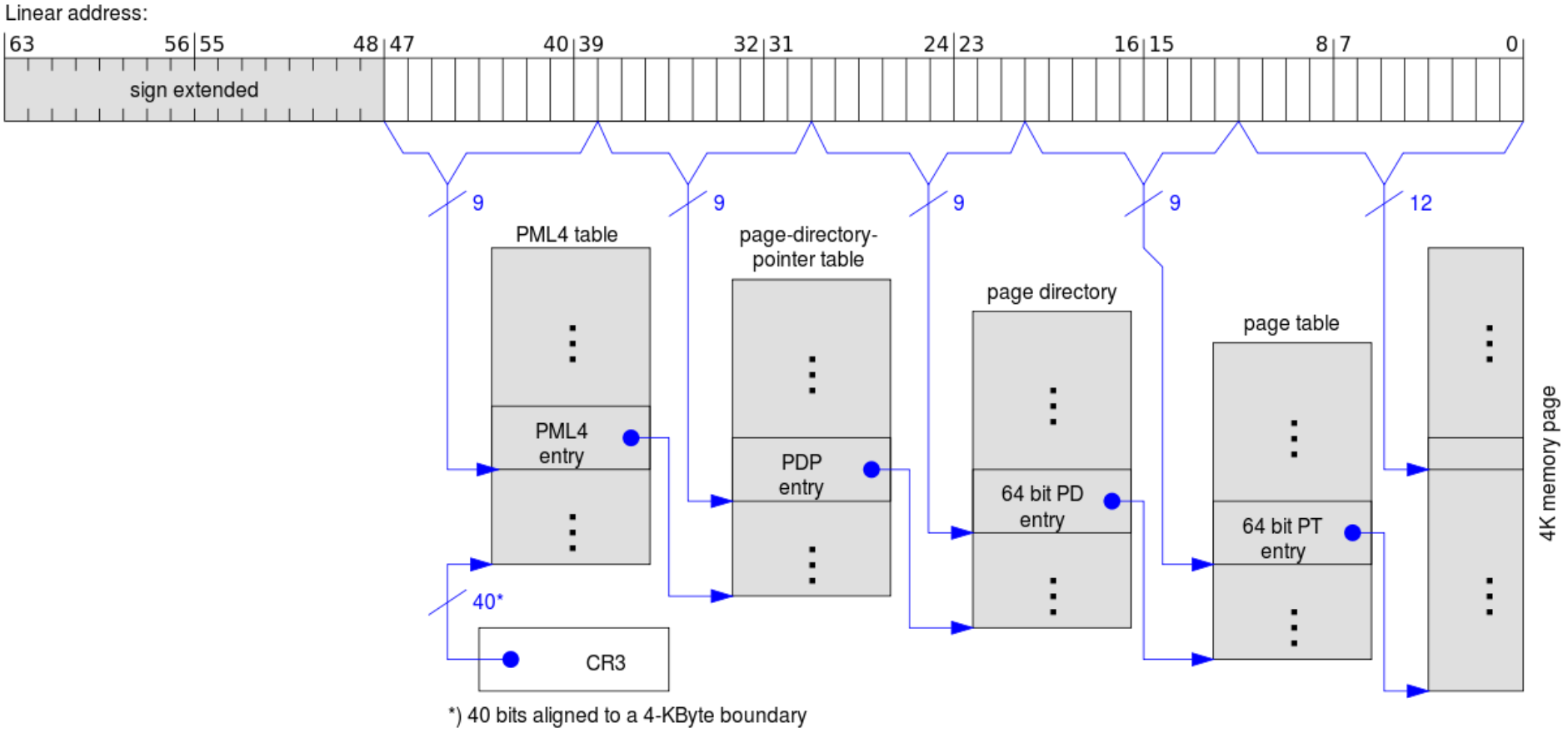
- Der höherwertige Teil verweist auf einen Eintrag im Seitenverzeichnis (wählt die zu verwendende Seitentabelle aus)
- Der niederwertige Teil adressiert eine Seite in der Seitentabelle.

# Mehrstufige Seitentabellen – Schema



# Mehrstufige Seitenverwaltung – Beispiel AMD64

**Grafik: Wikipedia**




- Virtueller Speicher auf $2^{48}$ Byte $= 256$ TiB begrenzt
- Seitengröße dynamisch bis 1GiB möglich

# Paging und Segmentierung

- In früheren Prozessorarchitekturen wurden **Paging** und **Segmentierung** miteinander verknüpft, d.h. die logischen Adressen (Segmentierung) wurden zunächst auf virtuelle Adressen (Paging) abgebildet. Aus diesen wurden dann physische Adressen ermittelt.
- Seit der **AMD64-Architektur** wird dieses Verfahren im Bereich der Heimcomputer nicht mehr eingesetzt, es wird nur noch Paging verwendet.

# Auslagerung von Seiten – Swapping

- Da der virtuelle Speicher wesentlich größer ist als der tatsächlich existierende, können Prozesse insgesamt mehr Speicher anfordern als zur Verfügung steht.
- Da wiederum die Prozesse nur **virtuellen** Speicher adressieren, ist es jedoch ohne weiteres möglich, Seiten auf sekundäre Datenträger **auszulagern** und bei Bedarf wieder einzulagern. Dieses Vorgehen nennt man **Swapping** (Austauschung).
- Hieraus ergibt sich auch die Terminologie der Seitenrahmen: Sie sind temporäre Behältnisse für Seiten, die nach Bedarf in diese hineingelegt und wieder aus diesen entnommen werden können.

# Auslagerung von Seiten – Swapping

- Um Swapping zu realisieren, enthält jede Eintrag in der Seitentabelle eine **zusätzliches Information**, ob diese **eingelagert** ist oder nicht.
- Soll auf eine Seite zugegriffen werden, die aktuell nicht eingelagert ist, wird ein **Softwareinterrupt** ausgelöst. Dieser startet einen Betriebssystemprozess, der die Einlagerung der Seite veranlasst. Im Anschluss kann der unterbrochene Prozess auf die Seite zugreifen.
- Ist kein freier Seitenrahmen mehr für die Einlagerung vorhanden, muss (mindestens) eine Seite ausgelagert werden. Hierfür existieren verschiedene **Seitenersetzungsstrategien**.

# Seitenersetzungsstrategien

- Die verschiedenen Algorithmen zur Ermittlung der nächsten zu ersetzenden Seite arbeiten meist zyklisch.
- Sie greifen für ihre Entscheidungsfindung auf Zugriffsstatistiken für jede einzelne Seite zurück. Daher gibt es in der Seitentabelle zwei zusätzliche Flags*) je Seite:
  1. Das **R-Flag**, das angibt, ob die Seite seit dem letzten Analysezyklus gelesen wurde (*read*),
  2. Das **M-Flag**, das analog dazu angibt, ob die Seite verändert wurde (*modify*).

*) Flags (Flaggen) sind 1-Bit Werte, sind also entweder gesetzt (=1) oder gelöscht (=0).

# Seitenersetzungsstrategien – LRU

**Least Recently Used (LRU):**

- Ermittle für jede potentiell zu ersetzende Seite, wie oft sie in der **Vergangenheit** genutzt wurde. Dies ist (laut Algorithmus) ein Indiz dafür, wie oft sie in Zukunft verwendet werden wird.
- Ersetze nun die Seite mit der **kleinsten** Benutzungswahrscheinlichkeit.

| Angeforderte Seiten: | 1 | 2 | 3 | 1 | 4 | 1 | 3 | 2 | 3 |
|---|---|---|---|---|---|---|---|---|---|
| Rahmen 1 | (1) | 1 | 1 | (1) | 1 | (1) | 1 | 1 | 1 |
| Rahmen 2 | - | (2) | 2 | 2 | (4) | 4 | 4 | (2) | 2 |
| Rahmen 3 | - | - | (3) | 3 | 3 | 3 | (3) | 3 | (3) |

Ersetzung Ersetzung

# Seitenersetzungsstrategien – LFU und FIFO

**Least Frequently Used (LFU):**

- Ermittle für jede potentiell zu ersetzende Seite ihre Nutzungsfrequenz. Ersetze die Seite mit der **kleinsten** Frequenz.

**First-In-First-Out (FIFO):**

- Die **älteste** Seite wird zuerst ersetzt.

**Random:**

- Eine **zufällige** Seite wird ersetzt.

# Lokalität und Working-Set

- Bei der Beobachtung realer Prozesse stellt man fest, dass sich deren Zugriffe über die Zeit meistens innerhalb **derselben Datenmenge** bewegen (Lokalität).
- Diese Erkenntnis rechtfertigt nicht nur den Einsatz von **Caches** sondern kann auch als Information für **Seitenersetzung** genutzt werden.
- Das **Working-Set** eines Prozesses sind diejenigen Seiten, die er in den letzten $T$ Schritten verwendet hat.
- **Ersetzungsstrategie:** Tausche möglichst nur Seiten aus, die nicht zum Working-Set eines laufenden Prozesses gehören.

# Zusammenfassung

- Die Verwaltung des zur Verfügung stehenden Hauptspeichers ist eine wesentliche Aufgabe des Betriebssystems.
- Das Konzept der Segmentierung ermöglicht geschützte Speicherbereiche für jeden Prozess und reduziert außerdem die Länge des Programmcodes durch indirekt-relative Adressierung.
- Beim Paging wird virtueller Speicher adressiert, sodass Prozessen wesentlich mehr Freiraum zur Verfügung steht. Zugriffsschutz und Seitenauslagerung werden von der Hardware unterstützt.