Speicherverwaltung (1)

Ute Bormann, TI2

2023-10-13

Betriebssysteme

- Prozessverwaltung
- ⇒ Speicherverwaltung
- Dateiverwaltung
- Geräteverwaltung
- Prozessverwaltung
  ⇒ Nebenläufigkeit ⇒ Kommunikation

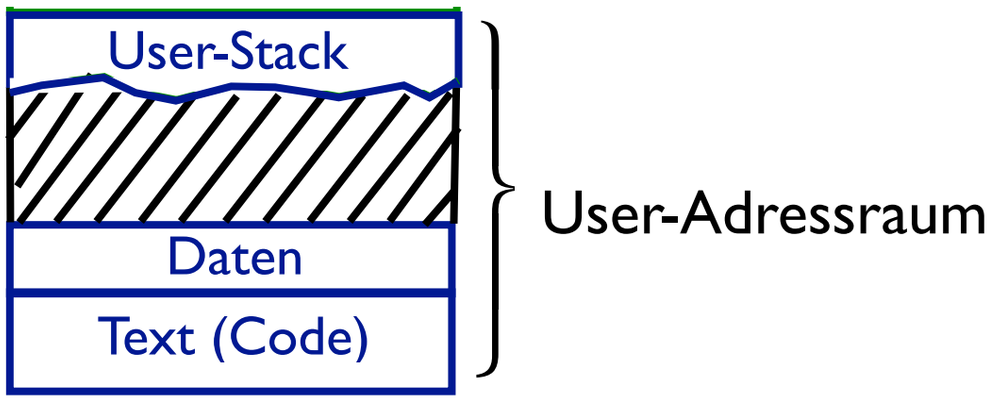
Inhalt

1. Grundlegendes
2. Prozessinterne Speicherwaltung
3. Abbildung auf den Hauptspeicher
4. Segmentierung vs. Paging

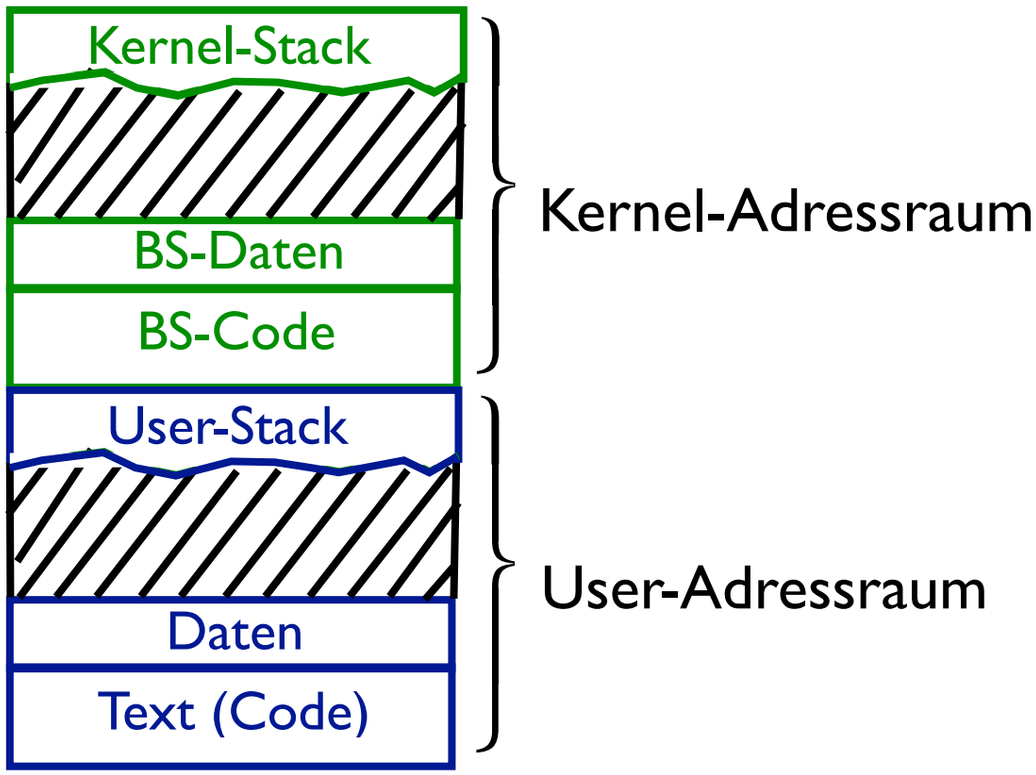
Teil 1:
Grundlegendes

08 Speicherverwaltung (1), Ute Bormann

Adressraum eines Prozesses (Wdh.)

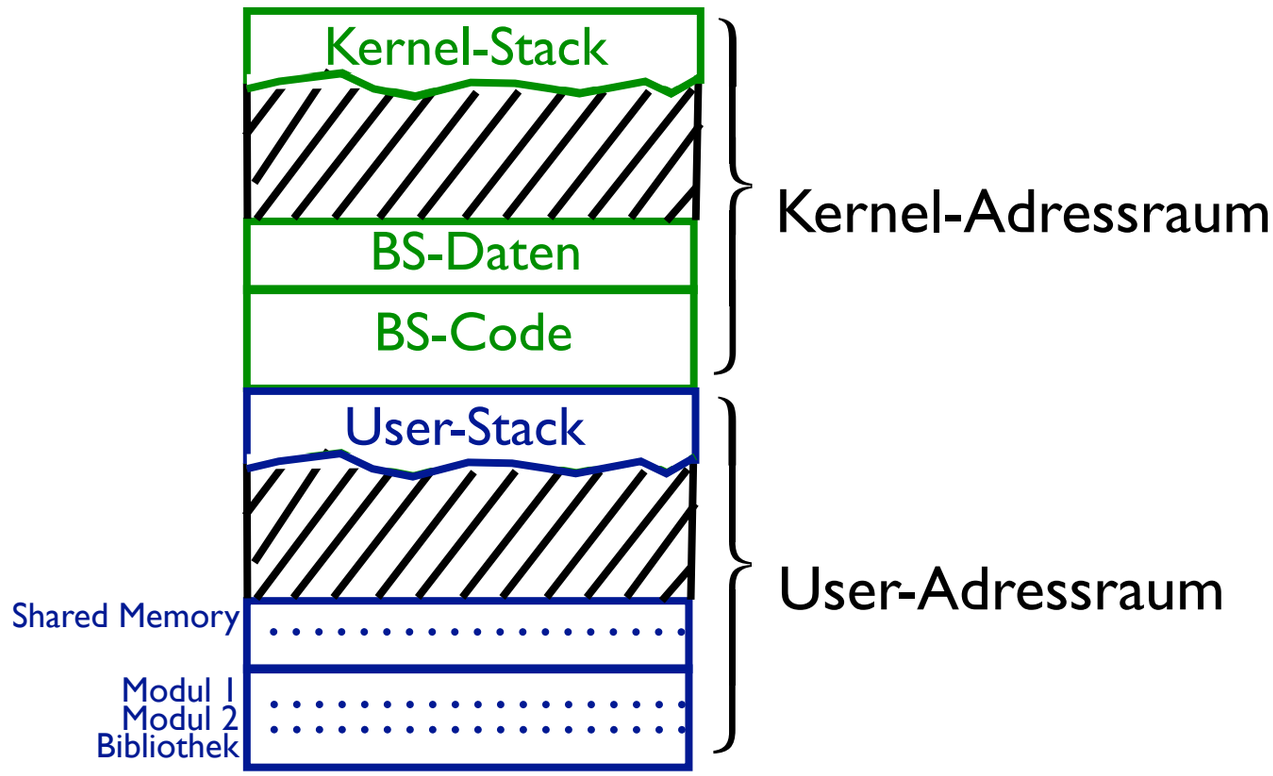


Adressraum eines Prozesses (Wdh.)



- Im Kernel-Mode auch Kern-Adressraum zugreifbar
- Verschiedene Segmente für verschiedene Zwecke
  - Read-only vs. Read-Write
  - „Lokale Daten“ (Stack) vs. globale Daten
  - User vs. Kernel

Adressraum eines Prozesses (Wdh.)



- Im Kernel-Mode auch Kern-Adressraum zugreifbar
- Verschiedene Segmente für verschiedene Zwecke
  - Read-only vs. Read-Write
  - „Lokale Daten“ (Stack) vs. globale Daten
  - User vs. Kernel

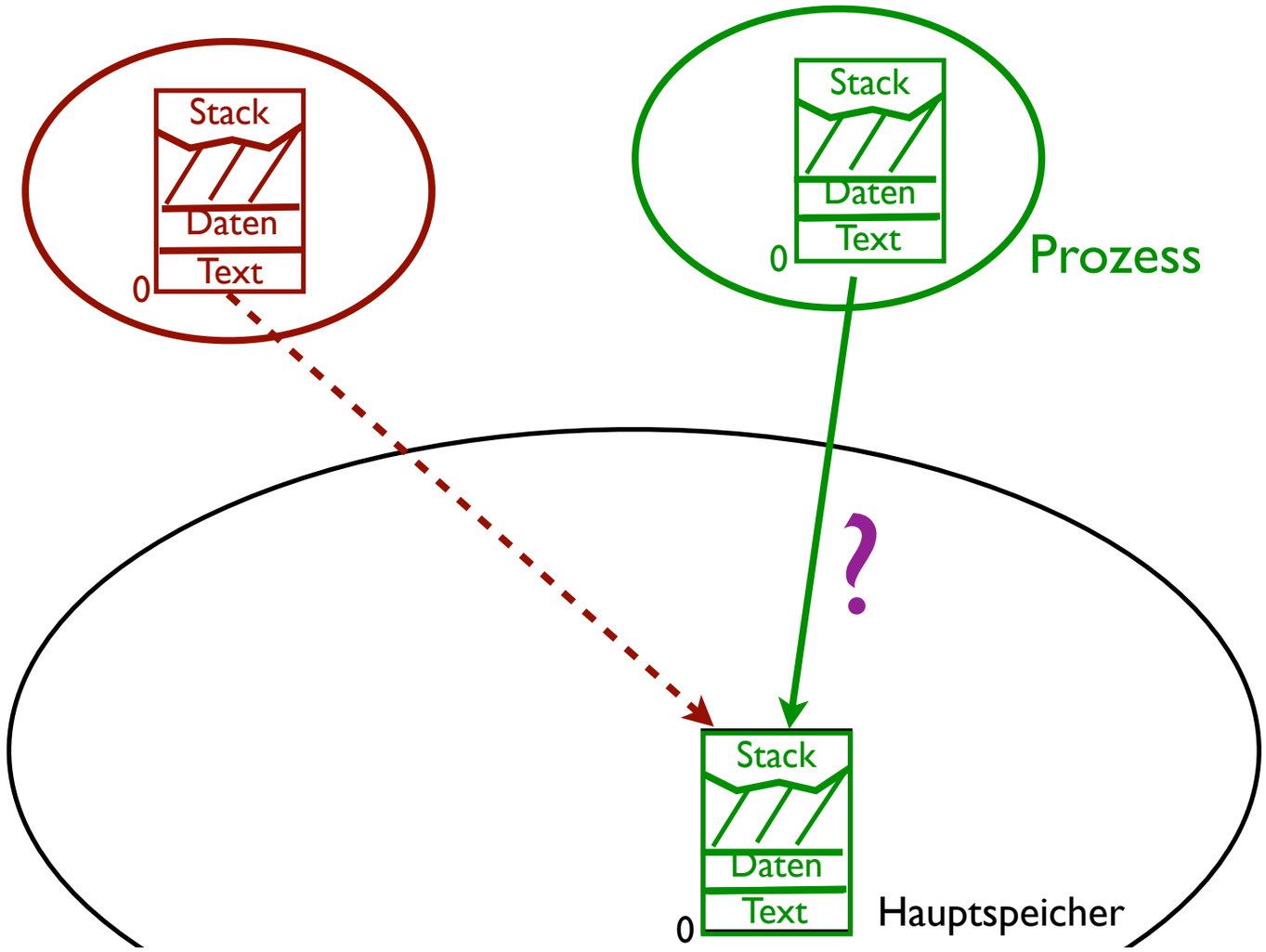
⇒ auch feinere Unterteilung denkbar
(Module, Bibliotheken, Shared Memory, Dateien,...)

⇒ „logische“ Gliederung des Adressraums

- Virtueller Adressraum nur Modellvorstellung

  ⇒ Abbildung auf realen Speicher erforderlich

  ⇒ Verwaltung realen Speichers durch Betriebssystem



- Prozessstart (fork) + Programmstart (exec)

  ⇒ initialer virtueller Adressraum durch a.out-Datei gegeben

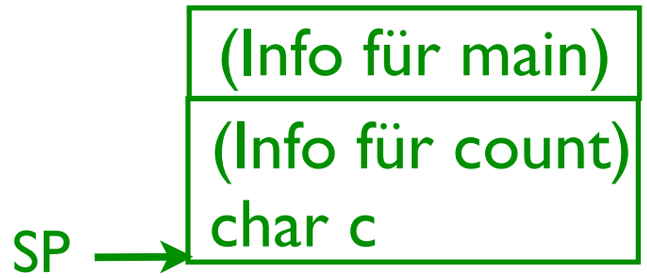
  ⇒ initial gültige virtuelle Adressen

Während des Programmlaufs:

- Textsegment bleibt unverändert
  (Alternativ: dynamisches Linken)

- Stack wächst/schrumpft nach Bedarf (streng linear)

  ⇒ implizit maximale Größe von Betriebssystem bereitgestellt



Während des Programmlaufs:

- Textsegment bleibt unverändert
  (Alternativ: dynamisches Linken)

- Stack wächst/schrumpft nach Bedarf (streng linear)

  ⇒ implizit maximale Größe von Betriebssystem bereitgestellt

- Datensegment kann Größe nach Bedarf verändern

  ⇒ explizit anfordern/freigeben bei Betriebssystem

  ⇒ brk („erste ungültige Adresse“)

  ⇒ ist neue Obergrenze des Datensegments

  ⇒ ebenfalls streng linear

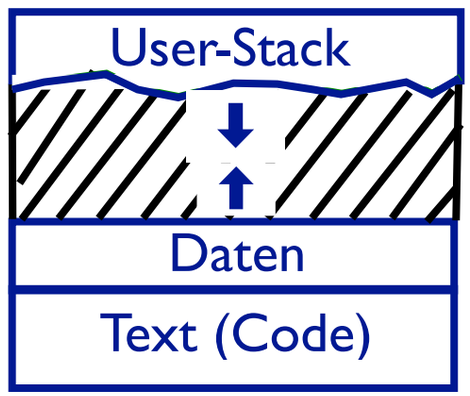
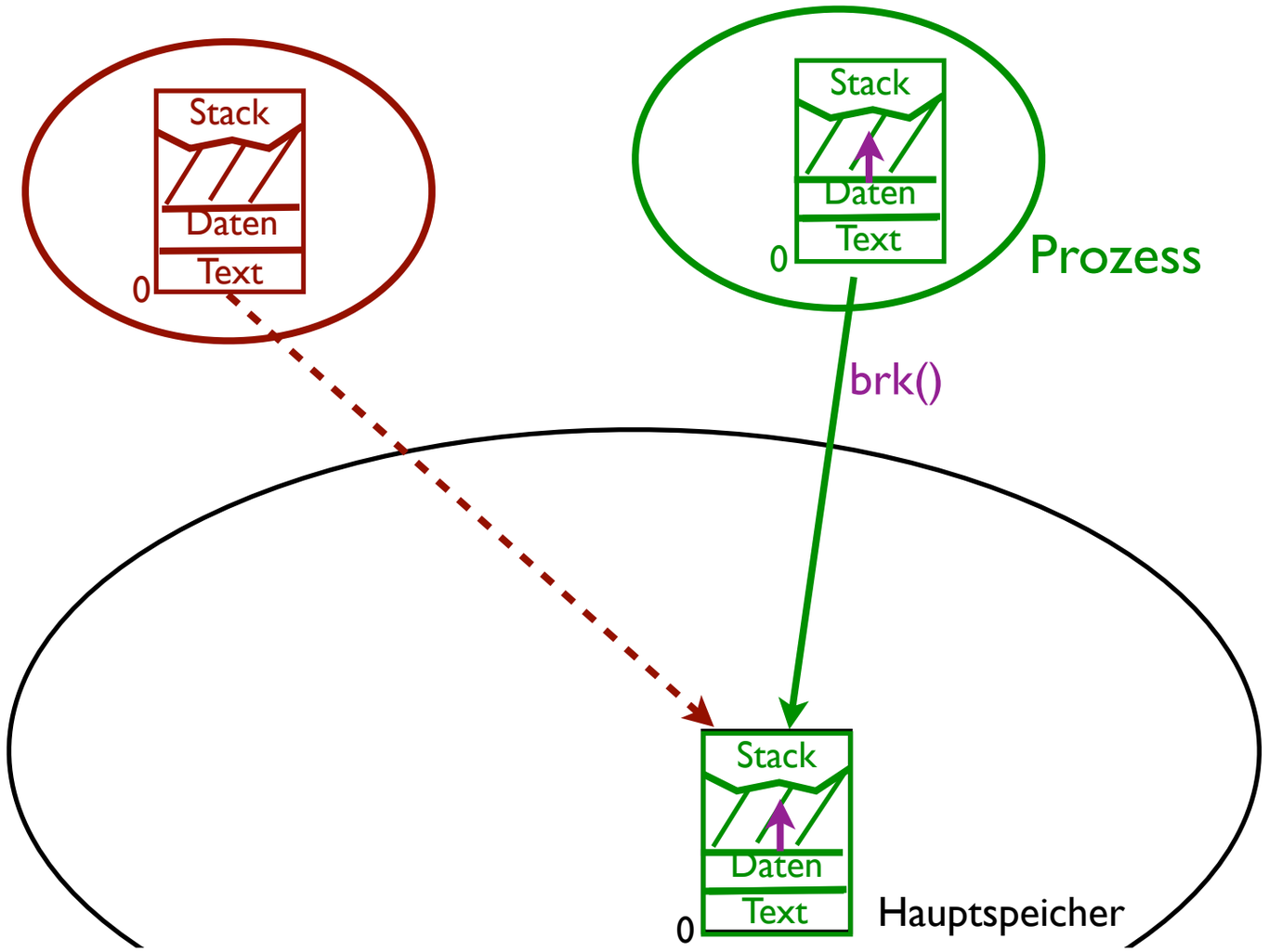


Abbildung virtueller Adressraum auf Hauptspeicher



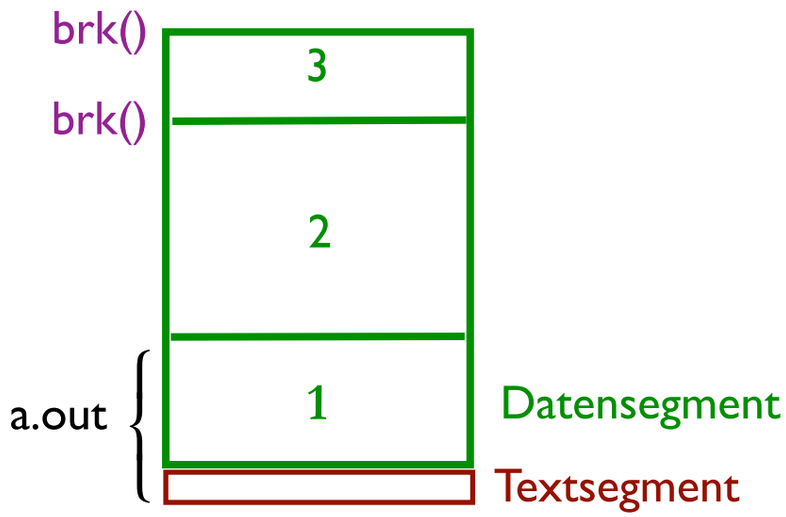
Fragen – Teil 1

- Wie verändert sich die Größe der verschiedenen Segmente des Prozess-Adressraums beim Programmablauf?

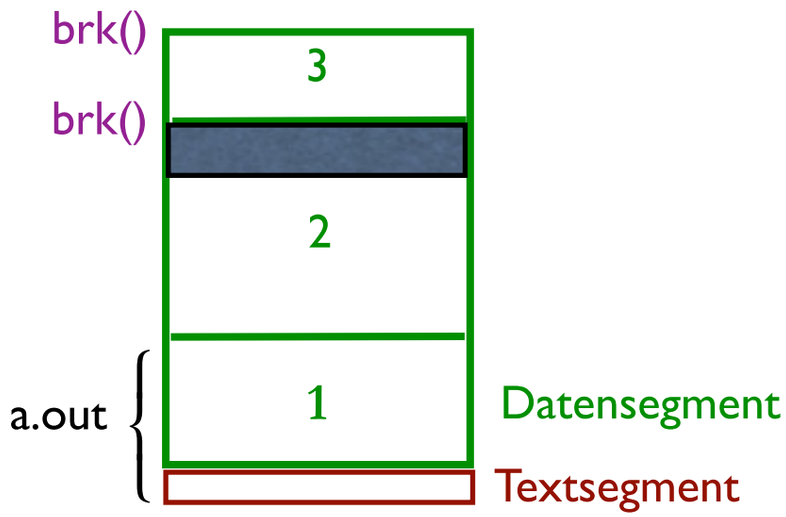
Teil 2:
Prozessinterne Speicherwaltung

08 Speicherverwaltung (1), Ute Bormann

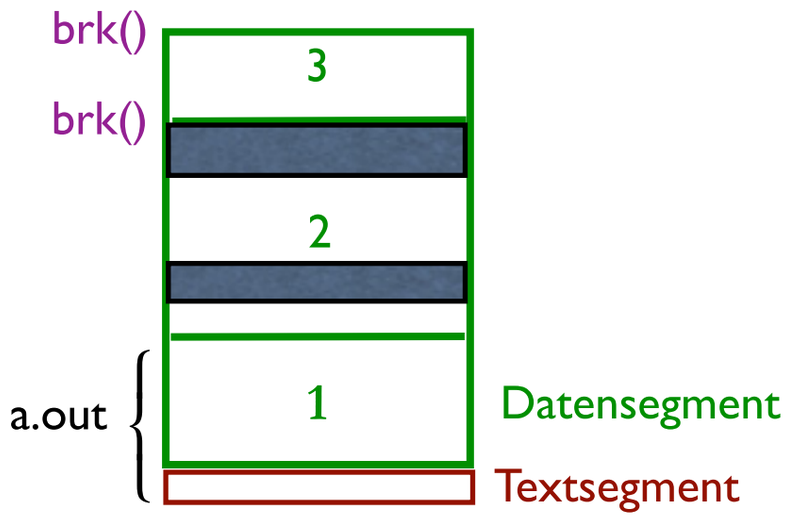
Speicherverwaltung innerhalb eines Prozesses



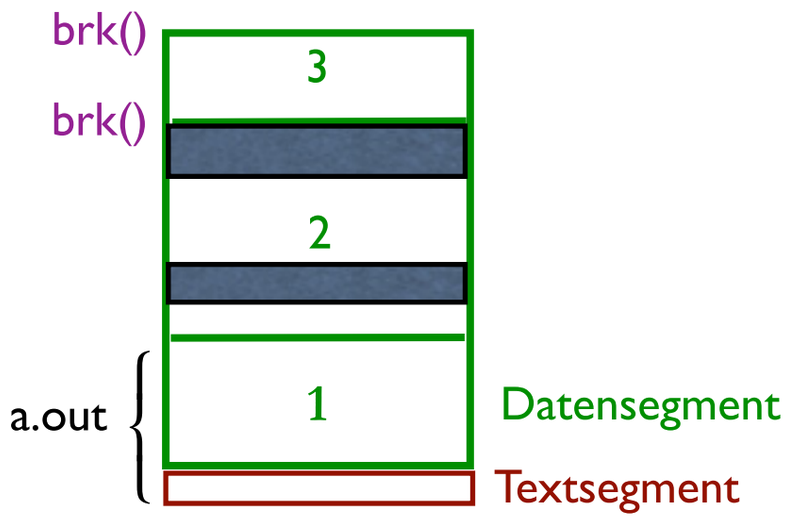
Speicherverwaltung innerhalb eines Prozesses



Speicherverwaltung innerhalb eines Prozesses

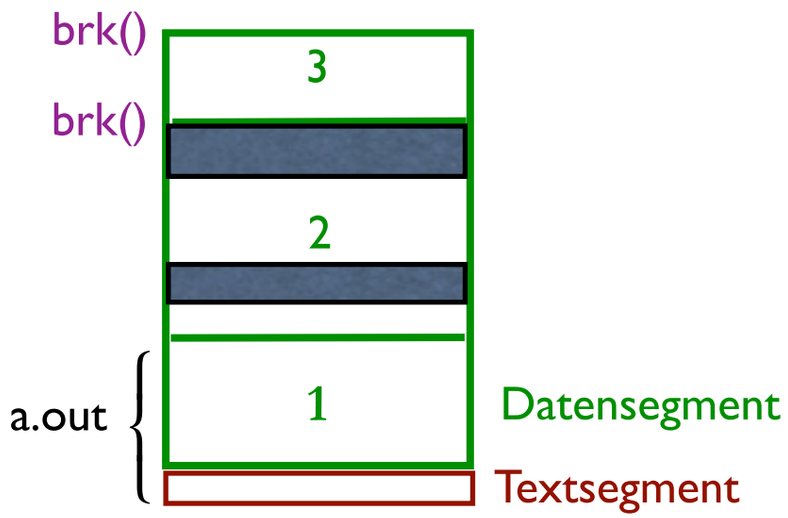


Speicherverwaltung innerhalb eines Prozesses



- Belegte Bereiche im Datensegment werden nicht unbedingt in umgekehrter Reihenfolge wieder frei

Speicherverwaltung innerhalb eines Prozesses



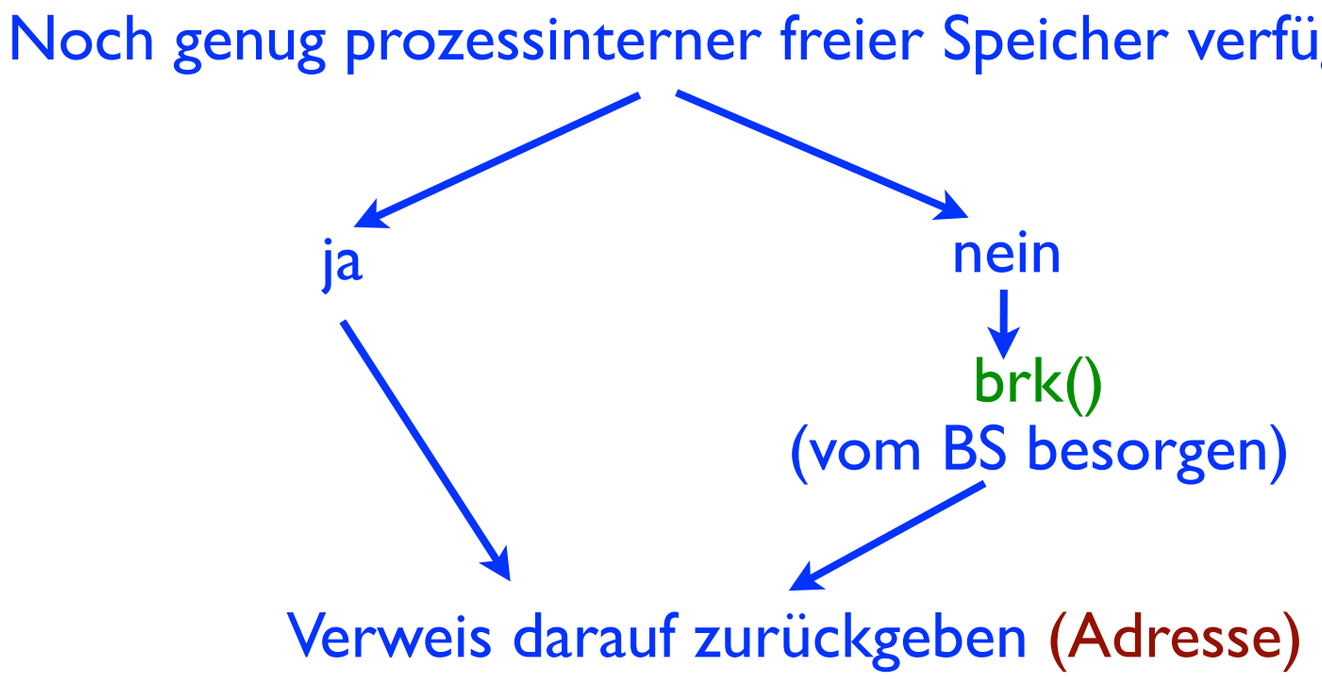
- Belegte Bereiche im Datensegment werden nicht unbedingt in umgekehrter Reihenfolge wieder frei
- Können (noch) nicht an Betriebssystem zurückgegeben werden
- Können aber für neue Speicheranforderungen genutzt werden

  ⇒ prozessinterne Freispeicherverwaltung

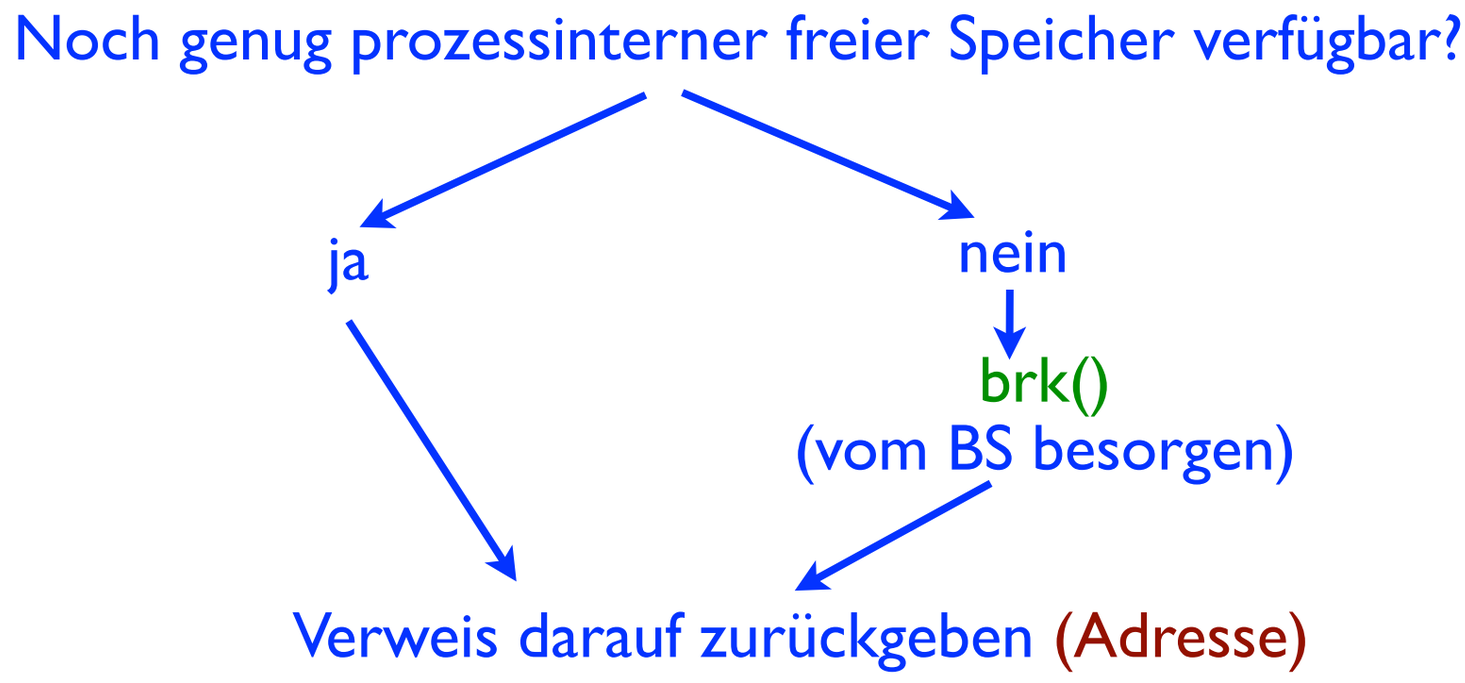
  z.B. über Bibliotheksroutinen (oder selbst geschrieben)

  - malloc() Anfordern von zusammenhängendem Speicherbereich
  - free() Freigeben von Speicherbereich
  - (In C++ wird prozessinterne Speicherverwaltung über Operatoren new/ delete angesprochen)

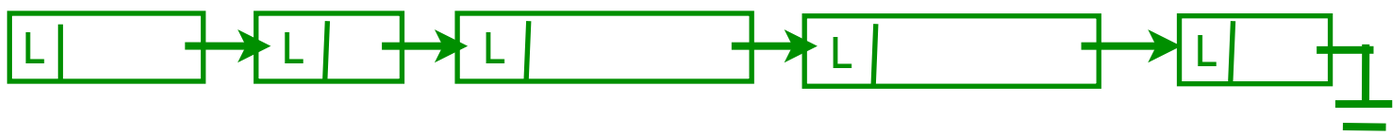
- Anfordern von Freispeicher: malloc (Anzahl in Bytes)



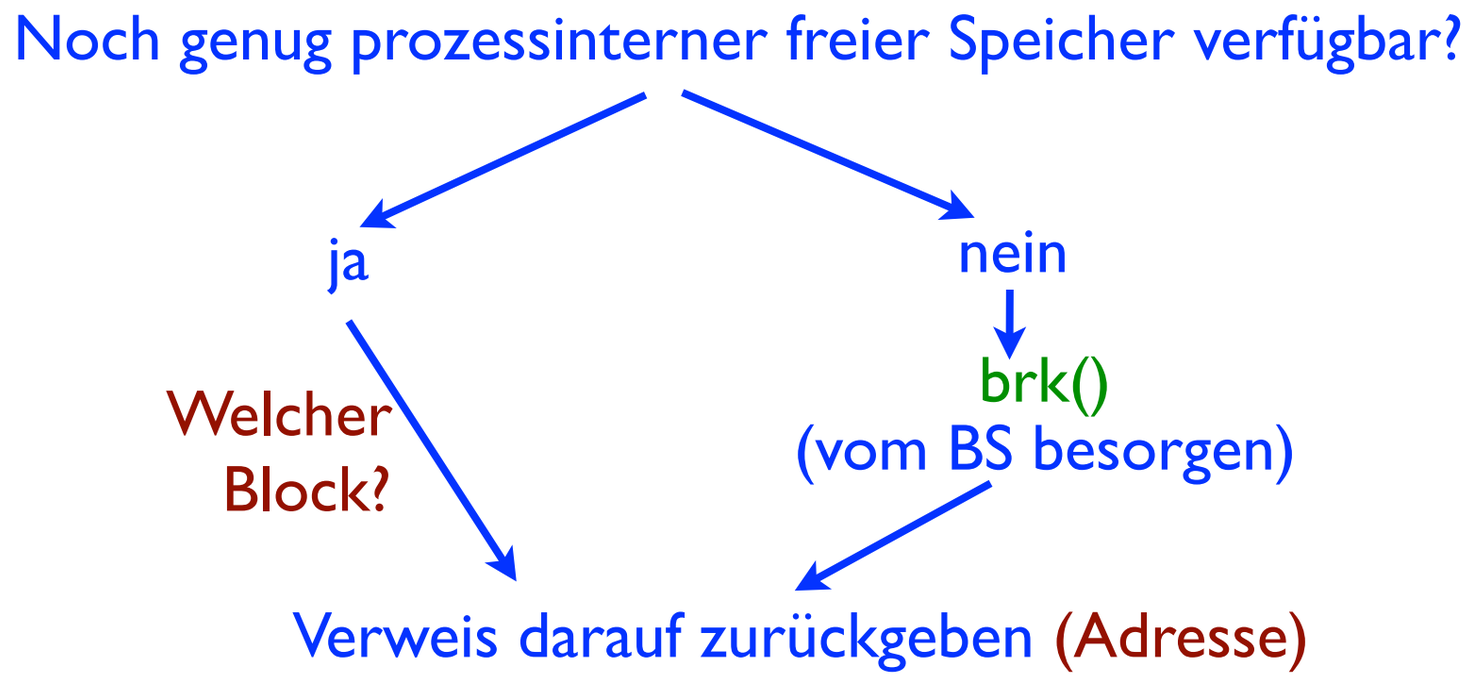
- Anfordern von Freispeicher: malloc (Anzahl in Bytes)



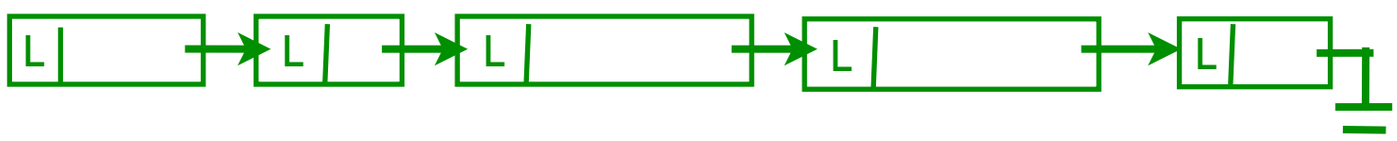
- Freigeben von prozessinternem Speicher: free (Adresse)
  (nur Block als ganzes zurückgeben)
  - wird i.d.R. nicht an Betriebssystem zurückgegeben
  - sondern in Freispeicherliste eingehängt



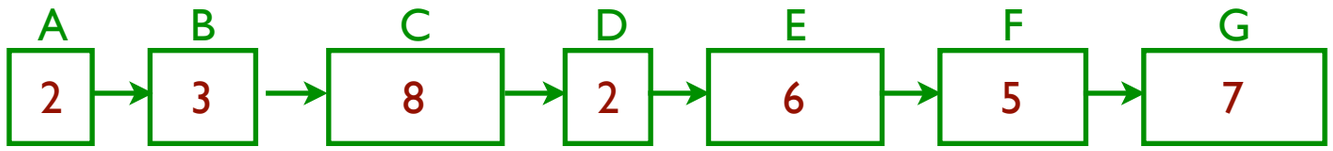
- Anfordern von Freispeicher: malloc (Anzahl in Bytes)



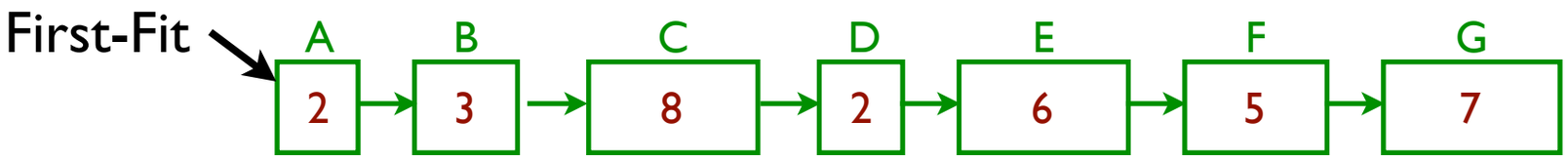
- Freigeben von prozessinternem Speicher: free (Adresse)
  (nur Block als ganzes zurückgeben)
  - wird i.d.R. nicht an Betriebssystem zurückgegeben
  - sondern in Freispeicherliste eingehängt



- Zurück zu malloc():
  - Block muss groß genug sein
  - welchen nehmen? ⇒ verschiedene Algorithmen



Beispiel: Anforderung von 4 Einheiten



Beispiel: Anforderung von 4 Einheiten

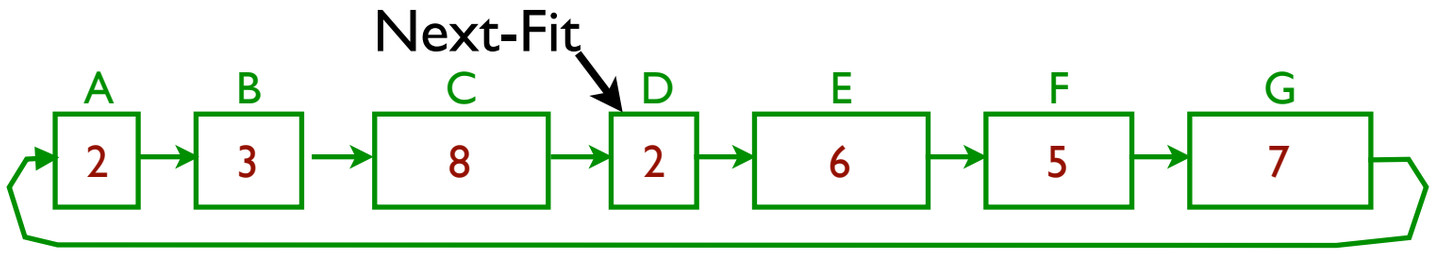
- First-Fit
  - Ersten Block nehmen, der groß genug

    ⇒ hier: Block C
  - Aufspalten in zwei Teile:
    - Belegt (4)
    - Frei (4)

    ⇒ hätte man besser nutzen können

⇒ Folgen:

- Ansammlung kleiner Blöcke am Anfang der Liste
- Suche dauert im Mittel immer länger
- Aber kleine Anforderungen schnell erfüllt



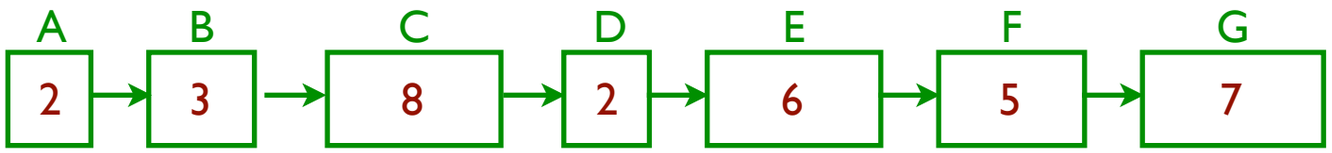
Beispiel: Anforderung von 4 Einheiten

- Next-Fit
  - Ab vorheriger Position zyklisch weitersuchen

    ⇒ hier: falls bei D aufgehört: Block E

⇒ Folgen:

  - Zunächst schnelle Ergebnisse
  - Aber: Große Blöcke werden schnell zerhackt



Beispiel:
Anforderung von 4 Einheiten

- Best-Fit
  - Liste ganz durchsuchen (sofern kein Block genau passt)
  - Block mit geringstem Verschnitt nehmen

    ⇒ hier: Block F

  ⇒ Folgen:
    - Große Blöcke noch lange verfügbar
    - Mehr Anforderungen erfüllbar
    - Aber: Suche dauert i.d.R. lange
    - Außerdem: Viele extrem kleine Blöcke bleiben übrig (u.U. gar nicht nutzbar)
- Mögliche Optimierung:
  - Liste nach Größe sortieren
  - Dann dauert aber Freigeben länger

- Alle diese Algorithmen zerlegen Freispeicher i.d.R. in kleinere Blöcke:
  - Freispeicherblock auswählen
  - gewünschten Anteil belegen
  - Rest verbleibt in Freispeicherliste

⇒ Datensegment wird mehr und mehr fragmentiert

⇒ externe Fragmentierung (Freiräume zwischen belegten Blöcken)

- Alle diese Algorithmen zerlegen Freispeicher i.d.R. in kleinere Blöcke:
  - Freispeicherblock auswählen
  - gewünschten Anteil belegen
  - Rest verbleibt in Freispeicherliste

  ⇒ Datensegment wird mehr und mehr fragmentiert

  ⇒ externe Fragmentierung (Freiräume zwischen belegten Blöcken)

- Fragmentierung reduzieren/beseitigen?
  - Bei Freigabe Verschmelzbarkeit prüfen
  - Dazu evtl. Freispeicherliste nach Adressen sortieren
  - Oder: Belegte Bereiche zusammenschieben (umkopieren)

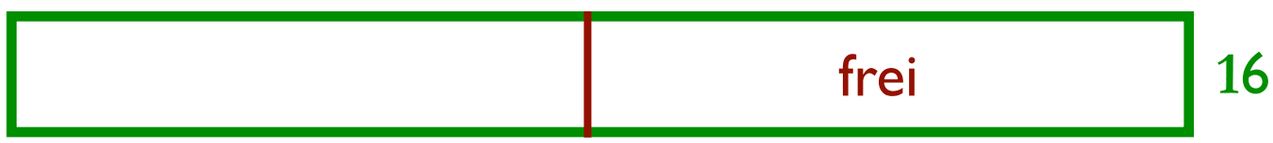
    ⇒ alles sehr aufwendig!

Der Buddy-Algorithmus

- Ziel: Externe Fragmentierung reduzieren und Verschmelzen vereinfachen
- Prinzip: Nur Blöcke der Größe 2^k anbieten
- Beispiel: Anforderung von 3 Einheiten

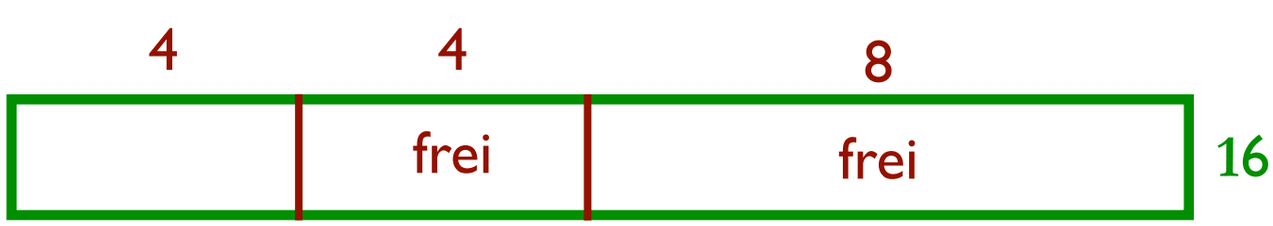
Der Buddy-Algorithmus

- Ziel: Externe Fragmentierung reduzieren und Verschmelzen vereinfachen
- Prinzip: Nur Blöcke der Größe 2^k anbieten
- Beispiel: Anforderung von 3 Einheiten



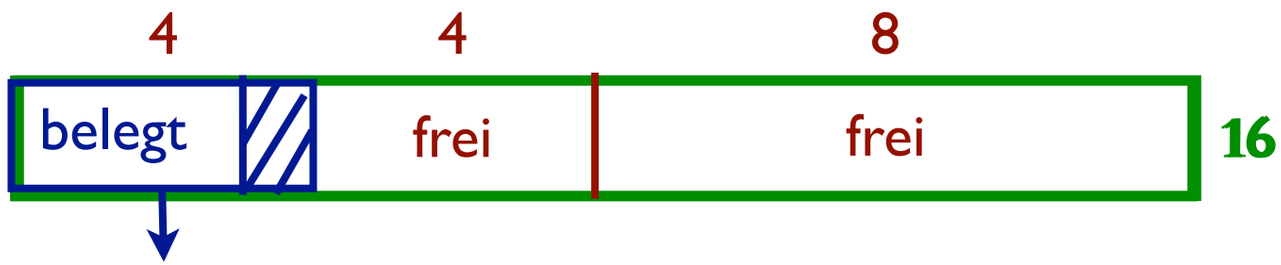
Der Buddy-Algorithmus

- Ziel: Externe Fragmentierung reduzieren und Verschmelzen vereinfachen
- Prinzip: Nur Blöcke der Größe 2^k anbieten
- Beispiel: Anforderung von 3 Einheiten



Der Buddy-Algorithmus

- Ziel: Externe Fragmentierung reduzieren und Verschmelzen vereinfachen
- Prinzip: Nur Blöcke der Größe 2^k anbieten
- Beispiel: Anforderung von 3 Einheiten

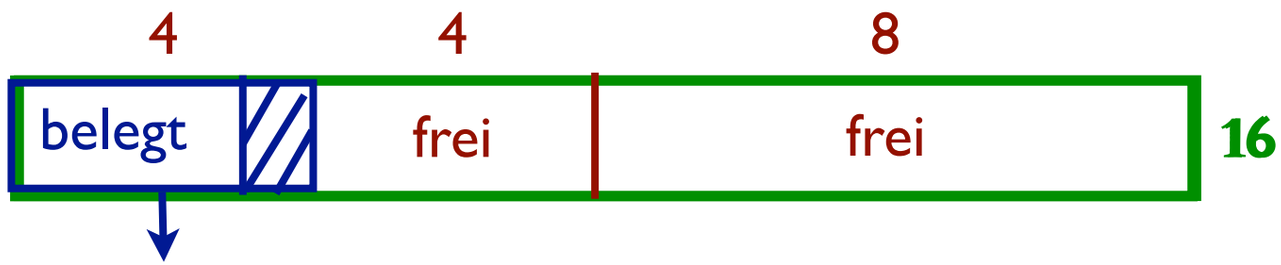


- Block der Größe 4 belegen
- Nur 3 benutzt ⇒ Rest ungenutzt

  ⇒ interne Fragmentierung

Der Buddy-Algorithmus

- Ziel: Externe Fragmentierung reduzieren und Verschmelzen vereinfachen
- Prinzip: Nur Blöcke der Größe 2^k anbieten
- Beispiel: Anforderung von 3 Einheiten



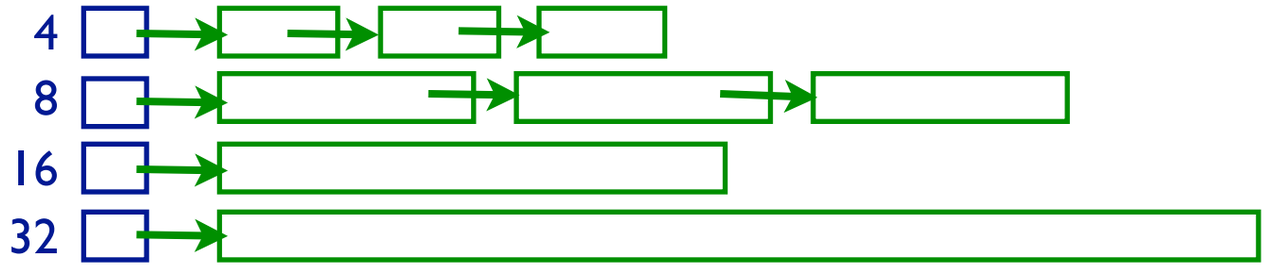
  - Block der Größe 4 belegen
  - Nur 3 benutzt ⇒ Rest ungenutzt

    ⇒ interne Fragmentierung

- Nächste Anforderungen nutzen verbleibende freie Blöcke bzw. splitten sie weiter

  ⇒ Splitten erzeugt jeweils zwei „Buddies“ (Nachbarn)

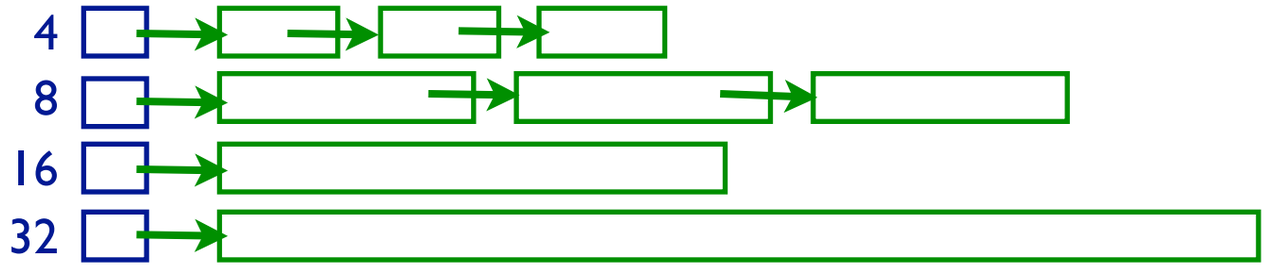
- Optimierung: Vorsehen von eigener Liste pro Blockgröße



⇒ sofortiges Auffinden eines „passenden“ Blocks

⇒ falls keiner da: größeren Block splitten

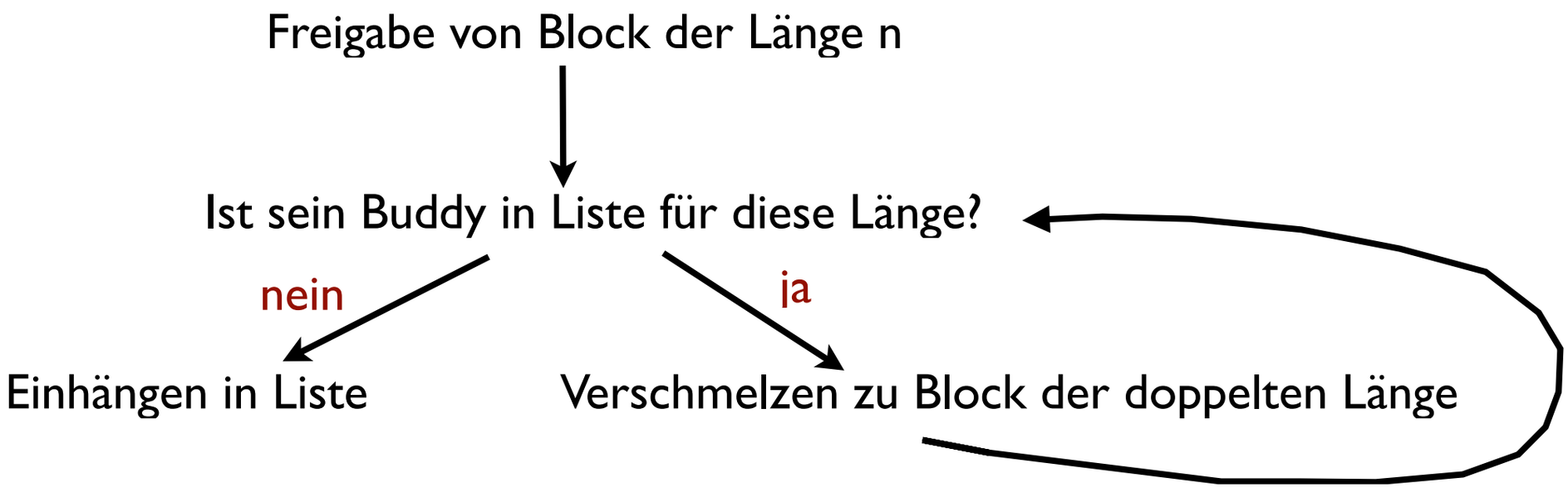
- Optimierung: Vorsehen von eigener Liste pro Blockgröße



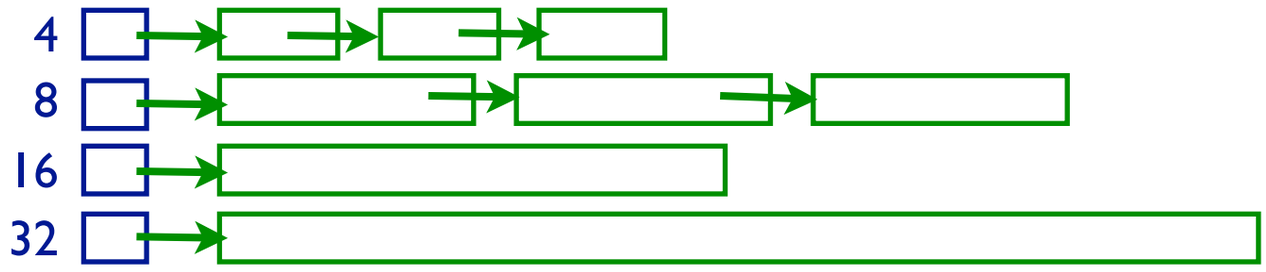
⇒ sofortiges Auffinden eines „passenden“ Blocks

⇒ falls keiner da: größeren Block splitten

- Verschmelzen bei Freigabe, falls möglich:



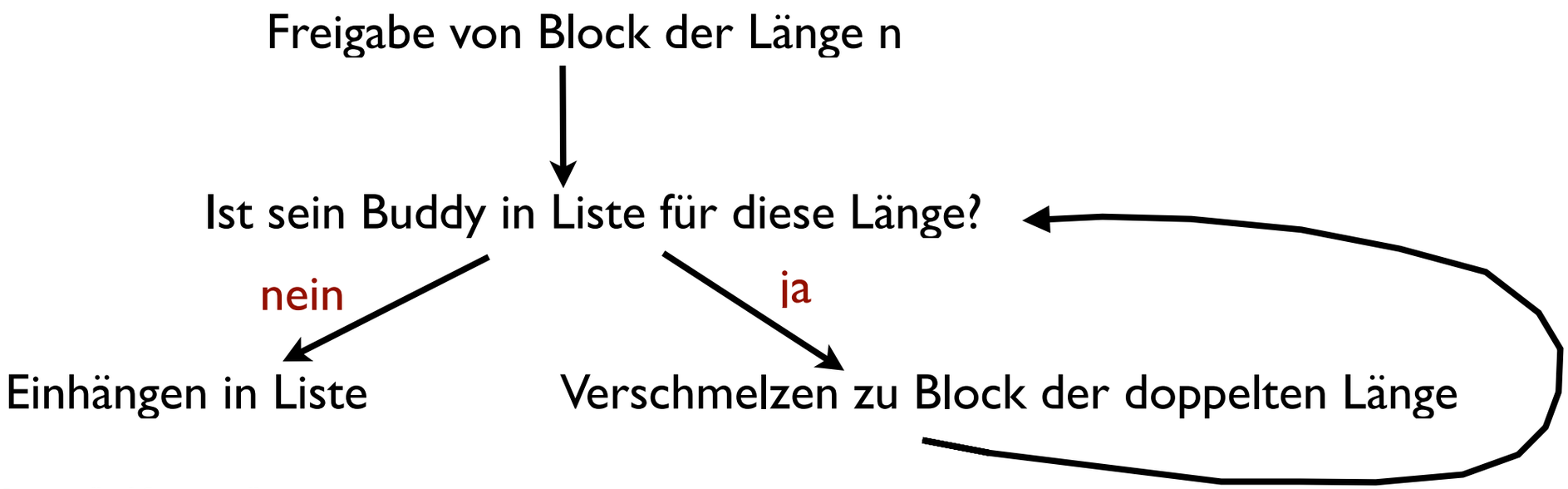
- Optimierung: Vorsehen von eigener Liste pro Blockgröße



⇒ sofortiges Auffinden eines „passenden“ Blocks

⇒ falls keiner da: größeren Block splitten

- Verschmelzen bei Freigabe, falls möglich:



- malloc()/free()
  - Große Blöcke: Ähnlich zu First-Fit/Next-Fit
  - Kleine Blöcke: Wenige, feste Blockgrößen, kein Splitten/Verschmelzen

Kleine Aufgabe

Ein Prozess verwendet den Buddy-Algorithmus für die prozessinterne Speicherverwaltung.

a) Initial ist ein Block der Länge 64 verfügbar.

b) Es gibt folgende Speicheranfragen: 13, 7, 3, 5, 2, 9, 6, 15. Sind sie erfüllbar?

c) Ist nach Freigabe der ersten fünf Blöcke eine Anfrage der Länge 18 erfüllbar?

Fragen – Teil 2

- Welche Vor- und Nachteile hat der *First-Fit*- bzw. der *Best-Fit*-Algorithmus zur Speicherverwaltung?
- Wie arbeitet der *Buddy-Algorithmus* in etwa?
- Wo tritt *interne Fragmentierung*, wo *externe Fragmentierung* auf?

Was ist das?

Teil 3:
Abbildung auf den Hauptspeicher

Abbildung virtueller Adressraum auf Hauptspeicher

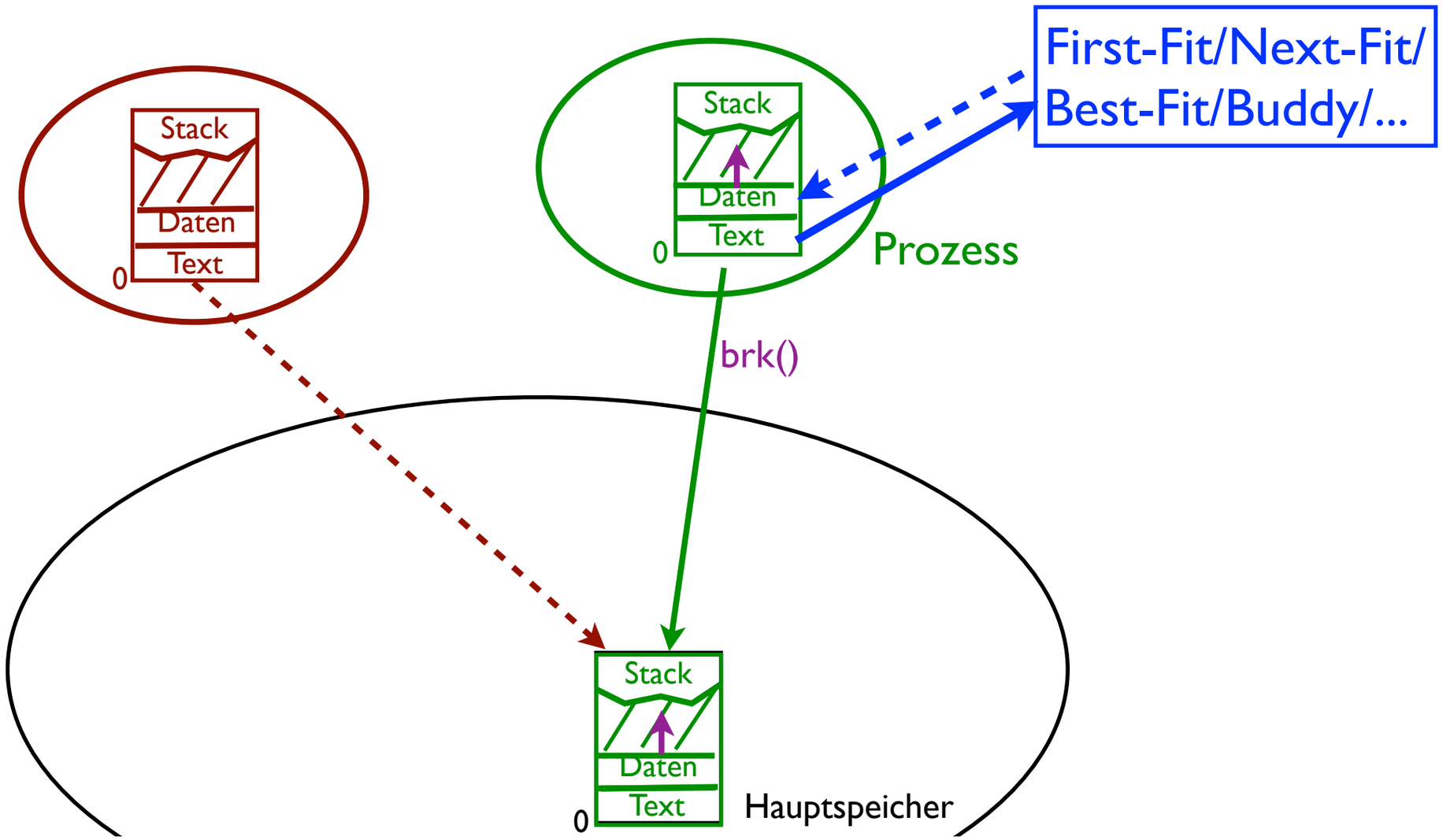


Abbildung virtueller Adressraum auf Hauptspeicher

Historischer Abriss:

- 1:1-Abbildung: Direkte Verwendung von Hauptspeicheradressen

  ⇒ zwei potentielle Probleme:

  1) Hauptspeicher zu teuer ⇒ zu klein
  2) Mehrprozessbetrieb

Abbildung virtueller Adressraum auf Hauptspeicher

Historischer Abriss:

- 1:1-Abbildung: Direkte Verwendung von Hauptspeicheradressen

  ⇒ zwei potentielle Probleme:

  1) Hauptspeicher zu teuer ⇒ zu klein
  2) Mehrprozessbetrieb

Ad 1) Overlays (ca. 1960)

- Programmierer bestimmt, welche Programmstücke dieselben Adressen bekommen

  ⇒ unterschiedliche „Phasen“ des Programms/Lochkartenstapels (ca. 1960)

  ⇒ Setzt verschiedenartige Speichermedien voraus (Speicherhierarchie)

Abbildung virtueller Adressraum → realer Speicher

• Abarbeitung von Code sowie Zugriff auf Daten durch CPU ⇒ Informationen müssen im Hauptspeicher sein	

Abbildung virtueller Adressraum → realer Speicher

• Hauptspeicher zu klein/teuer, um alle Informationen aller (potentiellen) Prozesse aufzunehmen ⇒ langfristige Ablage auf Hintergrundspeicher (i.d.R. Platte) ⇒ nur aktuell wichtige Informationen im Hauptspeicher	
• Abarbeitung von Code sowie Zugriff auf Daten durch CPU ⇒ Informationen müssen im Hauptspeicher sein	

Abbildung virtueller Adressraum → realer Speicher

• Hauptspeicher zu klein/teuer, um alle Informationen aller (potentiellen) Prozesse aufzunehmen ⇒ langfristige Ablage auf Hintergrundspeicher (i.d.R. Platte) ⇒ nur aktuell wichtige Informationen im Hauptspeicher	Platte: Zugriffszeit: ca. 10ms/512–4096 Bytes Preis: ca. 20 €/TByte Größe: z.B. 12 TByte
• Abarbeitung von Code sowie Zugriff auf Daten durch CPU ⇒ Informationen müssen im Hauptspeicher sein	Hauptspeicher: Zugriffszeit: ca. n*10 ns/4–100 Bytes Preis: ca. 4 €/GByte Größe: ca. 8–64 GByte

Abbildung virtueller Adressraum ➔ realer Speicher

• Hauptspeicher zu klein/teuer, um alle Informationen aller (potentiellen) Prozesse aufzunehmen ⇒ langfristige Ablage auf Hintergrundspeicher (i.d.R. Platte) ⇒ nur aktuell wichtige Informationen im Hauptspeicher	Platte: Zugriffszeit: ca. 10ms/512–4096 Bytes Preis: ca. 20 €/TByte Größe: z.B. 12 TByte SSD: Zugriffszeit: < 1ms Preis: ca. 100 €/TByte
• Abarbeitung von Code sowie Zugriff auf Daten durch CPU ⇒ Informationen müssen im Hauptspeicher sein	Hauptspeicher: Zugriffszeit: ca. n*10 ns/4–100 Bytes Preis: ca. 4 €/GByte Größe: ca. 8–64 GByte

Abbildung virtueller Adressraum ➔ realer Speicher

• Hauptspeicher zu klein/teuer, um alle Informationen aller (potentiellen) Prozesse aufzunehmen ⇒ langfristige Ablage auf Hintergrundspeicher (i.d.R. Platte) ⇒ nur aktuell wichtige Informationen im Hauptspeicher	Platte: Zugriffszeit: ca. 10ms/512–4096 Bytes Preis: ca. 20 €/TByte Größe: z.B. 12 TByte SSD: Zugriffszeit: < 1ms Preis: ca. 100 €/TByte
• Abarbeitung von Code sowie Zugriff auf Daten durch CPU ⇒ Informationen müssen im Hauptspeicher sein	Hauptspeicher: Zugriffszeit: ca. n*10 ns/4–100 Bytes Preis: ca. 4 €/GByte Größe: ca. 8–64 GByte
Genauer: Direkter CPU-Zugriff erfolgt i.d.R. auf Cache (Hardwareunterstützung, keine Betriebssystem-Aktivitäten)	Cache (Level1/Level2/Level3-Cache): Zugriffszeit: < 2ns/1–8 Byte Preis: (in CPU integriert) Größe: ca. 64/256/8192 KByte

Abbildung virtueller Adressraum → realer Speicher

• Hauptspeicher zu klein/teuer, um alle Informationen aller (potentiellen) Prozesse aufzunehmen ⇒ langfristige Ablage auf Hintergrundspeicher (i.d.R. Platte) ⇒ nur aktuell wichtige Informationen im Hauptspeicher	Platte: Zugriffszeit: ca. 10ms/512–4096 Bytes Preis: ca. 20 €/TByte Größe: z.B. 12 TByte SSD: Zugriffszeit: < 1ms Preis: ca. 100 €/TByte ?
• Abarbeitung von Code sowie Zugriff auf Daten durch CPU ⇒ Informationen müssen im Hauptspeicher sein	Hauptspeicher: Zugriffszeit: ca. n*10 ns/4–100 Bytes Preis: ca. 4 €/GByte Größe: ca. 8–64 GByte
Genauer: Direkter CPU-Zugriff erfolgt i.d.R. auf Cache (Hardwareunterstützung, keine Betriebssystem-Aktivitäten)	Cache (Level1/Level2/Level3-Cache): Zugriffszeit: < 2ns/1–8 Byte Preis: (in CPU integriert) Größe: ca. 64/256/8192 KByte

Exkurs: Maßeinheiten

- Hauptspeicher: oft in 2er-Potenzen von Bytes

$$1024 = 2^{10} \text{ Bytes} = 1 \text{ KByte } (\neq 10^3 = 1 \text{ kByte})$$
$$1048576 = 2^{20} \text{ Bytes} = 1 \text{ MByte } (\neq 10^6 = 1 \text{ MByte})$$
$$1073741824 = 2^{30} \text{ Bytes} = 1 \text{ GByte } (\neq 10^9 = 1 \text{ GByte})$$
$$\vdots$$

⇒ allerdings oft auch einige Sprachverwirrung
und z.T. Mischung von Einheiten

$$10^3 * 2^{10} \Rightarrow 1\text{M}\ldots$$
$$10^3 * 10^3 * 2^{10} \neq 10^3 * 2^{10} * 2^{10} \Rightarrow 1\text{G}\ldots$$

Exkurs: Maßeinheiten

- Hauptspeicher: oft in 2er-Potenzen von Bytes

$1024 = 2^{10}$ Bytes = 1 KByte ($\neq 10^3$ = 1 kByte)

$1048576 = 2^{20}$ Bytes = 1 MByte ($\neq 10^6$ = 1 MByte)

$1073741824 = 2^{30}$ Bytes = 1 GByte ($\neq 10^9$ = 1 GByte)

⋮

⇒ allerdings oft auch einige Sprachverwirrung
und z.T. Mischung von Einheiten

$10^3 * 2^{10} \Rightarrow$ 1M…

$10^3 * 10^3 * 2^{10} \neq 10^3 * 2^{10} * 2^{10} \Rightarrow$ 1G…

- Hauptspeicher: „1G = 2^{30}“
- Platten: „1T = 10^{12}“ (bei Disketten gab's auch Mischformen)
- Exkurs Rechnernetze: 1 Gbit/s = 10^9 bit/s

Das Problem mit den Maßeinheiten

$$1\ \mathrm{K} = 2^{10} \neq 10^{3} = 1\ \mathrm{k}$$

$$1\ \mathrm{M} = 2^{20} \neq 10^{6} = 1\ \mathrm{M}$$

$$1\ \mathrm{G} = 2^{30} \neq 10^{9} = 1\ \mathrm{G}$$

Das Problem mit den Maßeinheiten

1 K = 2^{10} ≠ 10^3 = 1 k

1 M = 2^{20} ≠ 10^6 = 1 M

1 G = 2^{30} ≠ 10^9 = 1 G

⇒ Ein Terminologie-Standard schafft Abhilfe
(IEEE 60027-2, Am2 → IEC 80000–13):

2^{10}	kilobinary	Kibi	Ki
2^{20}	megabinary	Mebi	Mi
2^{30}	gigabinary	Gibi	Gi
2^{40}	terabinary	Tebi	Ti

Das Problem mit den Maßeinheiten

1 K = 2^{10} ≠ 10^3 = 1 k

1 M = 2^{20} ≠ 10^6 = 1 M

1 G = 2^{30} ≠ 10^9 = 1 G

⇒ Ein Terminologie-Standard schafft Abhilfe
(IEEE 60027-2, Am2 → IEC 80000–13):

2^{10}	kilobinary	Kibi	Ki
2^{20}	megabinary	Mebi	Mi
2^{30}	gigabinary	Gibi	Gi
2^{40}	terabinary	Tebi	Ti

Außerdem: Byte üblicherweise als B abgekürzt

Abbildung virtueller Adressraum → realer Speicher

• Hauptspeicher zu klein/teuer, um alle Informationen aller (potentiellen) Prozesse aufzunehmen ⇒ langfristige Ablage auf Hintergrundspeicher (i.d.R. Platte) ⇒ nur aktuell wichtige Informationen im Hauptspeicher	Platte: Zugriffszeit: ca. 10ms/512–4096 Bytes Preis: ca. 20 €/TByte Größe: z.B. 12 TByte SSD: Zugriffszeit: < 1ms Preis: ca. 100 €/TByte
• Abarbeitung von Code sowie Zugriff auf Daten durch CPU ⇒ Informationen müssen im Hauptspeicher sein	Hauptspeicher: Zugriffszeit: ca. n*10 ns/4–100 Bytes Preis: ca. 4 €/GByte Größe: ca. 8–64 GByte
Genauer: Direkter CPU-Zugriff erfolgt i.d.R. auf Cache (Hardwareunterstützung, keine Betriebssystem-Aktivitäten)	Cache (Level1/Level2/Level3-Cache): Zugriffszeit: < 2ns/1–8 Byte Preis: (in CPU integriert) Größe: ca. 64/256/8192 KByte

Abbildung virtueller Adressraum ➔ realer Speicher

• Hauptspeicher zu klein/teuer, um alle Informationen aller (potentiellen) Prozesse aufzunehmen ⇒ langfristige Ablage auf Hintergrundspeicher (i.d.R. Platte) ⇒ nur aktuell wichtige Informationen im Hauptspeicher	Platte: Zugriffszeit: ca. 10ms/512–4096 Bytes Preis: ca. 20 €/TB Größe: z.B. 12 TB SSD: Zugriffszeit: < 1ms Preis: ca. 100 €/TB
• Abarbeitung von Code sowie Zugriff auf Daten durch CPU ⇒ Informationen müssen im Hauptspeicher sein	Hauptspeicher: Zugriffszeit: ca. n*10 ns/4–100 Bytes Preis: ca. 4 €/GiB Größe: ca. 8–64 GiB
Genauer: Direkter CPU-Zugriff erfolgt i.d.R. auf Cache (Hardwareunterstützung, keine Betriebssystem-Aktivitäten)	Cache (Level1/Level2/Level3-Cache): Zugriffszeit: < 2ns/1–8 Byte Preis: (in CPU integriert) Größe: ca. 64/256/8192 KiB

Ziel: Optimale Nutzung der Speicherhierarchie

- 85–95% der Zugriffe direkt auf Cache
- > 99,99% der Zugriffe auf Cache oder Hauptspeicher
- < 0,01% der Zugriffe: Infos erst von Platte laden

- Wie erzielen?

Ziel: Optimale Nutzung der Speicherhierarchie

- 85–95% der Zugriffe direkt auf Cache
- > 99,99% der Zugriffe auf Cache oder Hauptspeicher
- < 0,01% der Zugriffe: Infos erst von Platte laden

- Wie erzielen?

  a) Nur laufende/lauffähige Prozesse im Hauptspeicher (ggf. schnelle Umschaltung ermöglichen)

  b) Lokalitätsprinzip von Prozessen berücksichtigen

Lokalitätsprinzip von Prozessen

- Reale Programme springen i.d.R. nicht wild hin und her
- Arbeiten Codestücke sequentiell ab
- Bleiben lange in (kleinen) Schleifen
- Wechseln hin und wieder in neuen Kontext
  (Phasen der Abarbeitung: Prozeduren, Module)
- Darüber hinaus Zugriff auf Daten/Stack

⇒ Innerhalb eines bestimmten Zeitintervalls wird nur (kleine) Teilmenge der Infos benutzt

⇒ Working Set (sollte im Hauptspeicher sein)

- Working Set ändert sich von Zeit zu Zeit

Abbildung virtueller Adressraum auf Hauptspeicher

Historischer Abriss:

- 1:1-Abbildung: Direkte Verwendung von Hauptspeicheradressen

  ⇒ zwei potentielle Probleme:

  1) Hauptspeicher zu teuer ⇒ zu klein
  2) Mehrprozessbetrieb

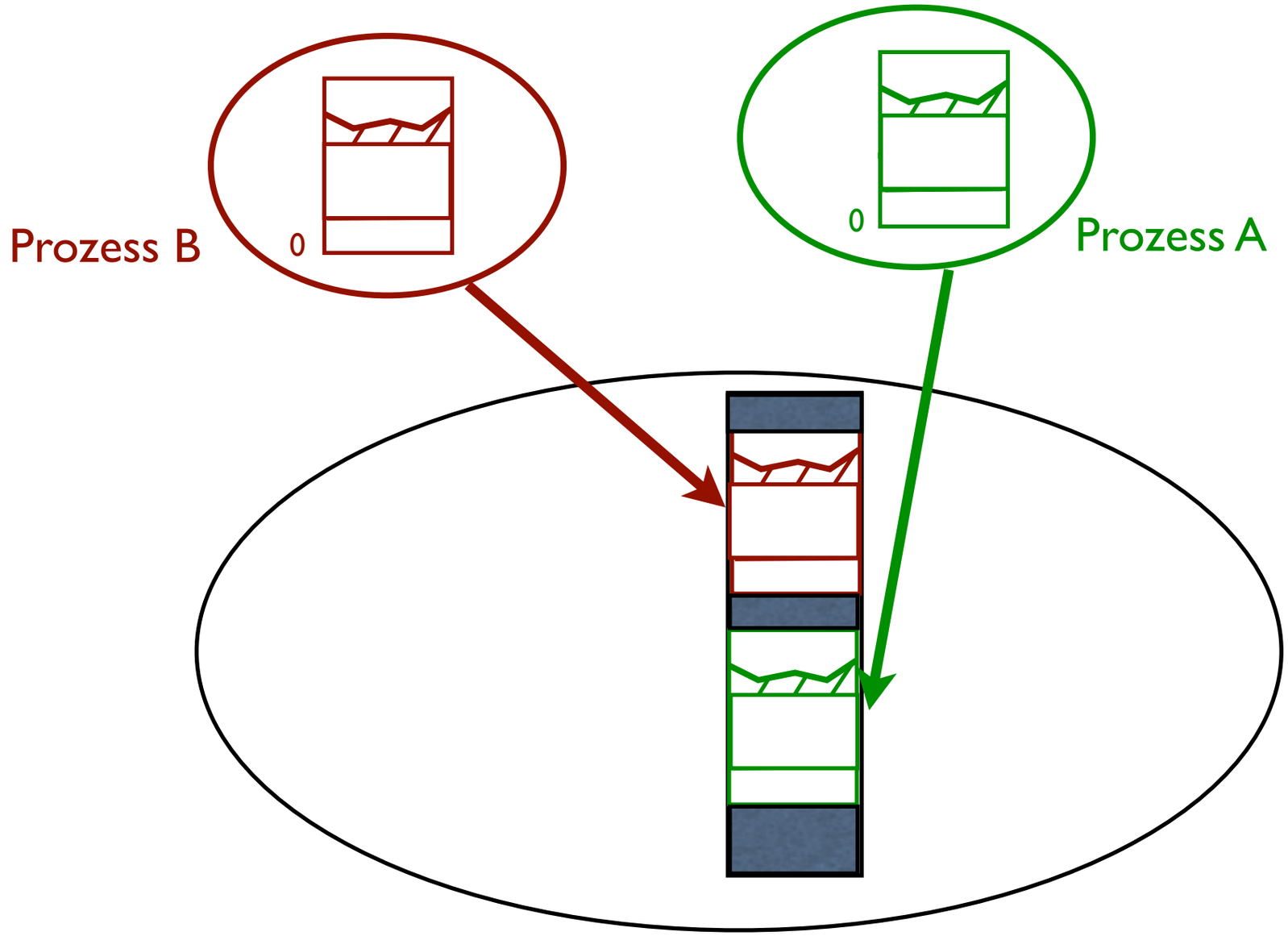
Ad 1) Overlays (ca. 1960)

…..

Ad 2) Relocation (ca. 1970)

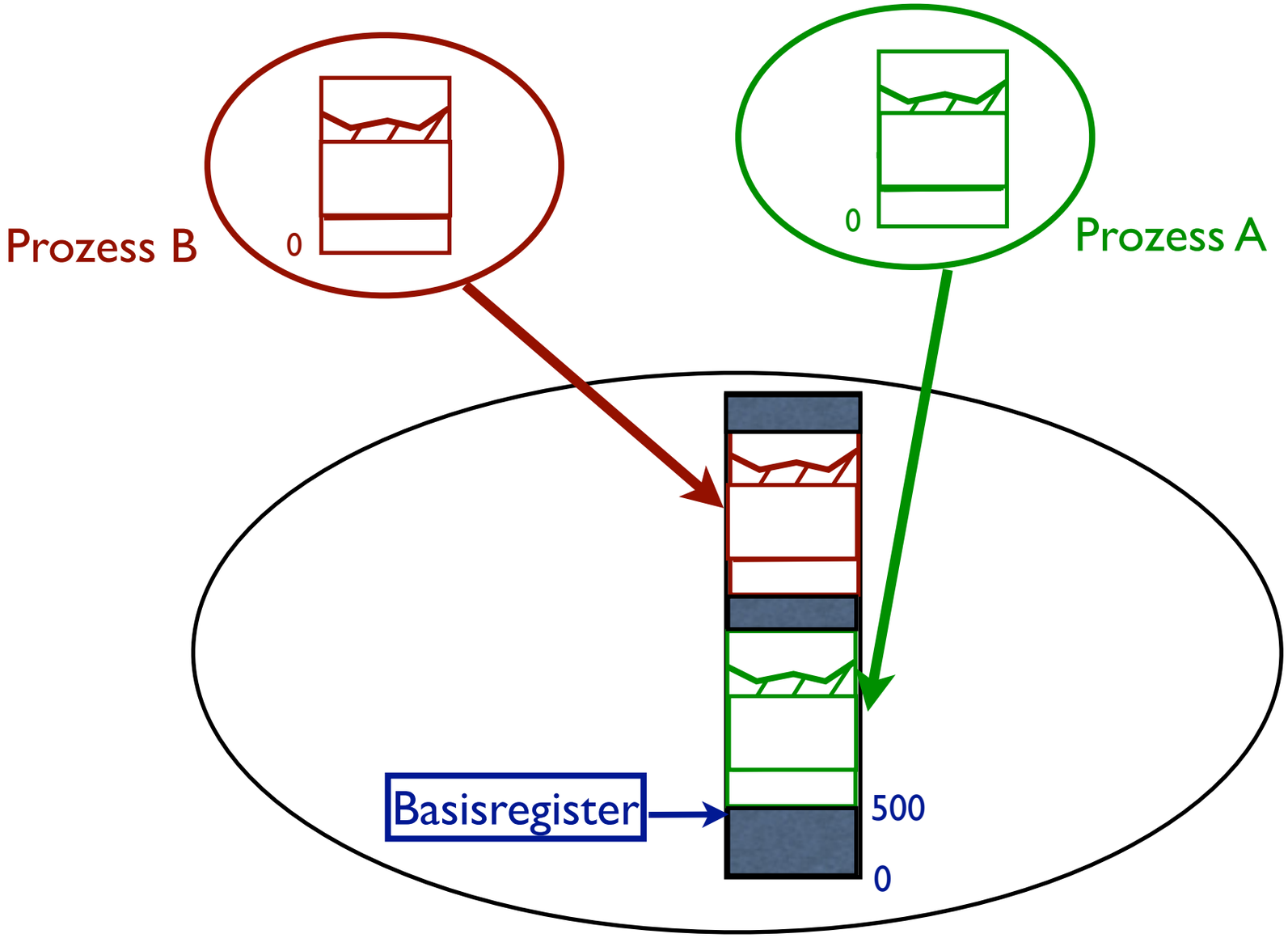
Ad 2) Relocation (ca. 1970)

- Programme werden in unterschiedliche Hauptspeicherbereiche geschoben



Ad 2) Relocation (ca. 1970)

- Programme werden in unterschiedliche Hauptspeicherbereiche geschoben

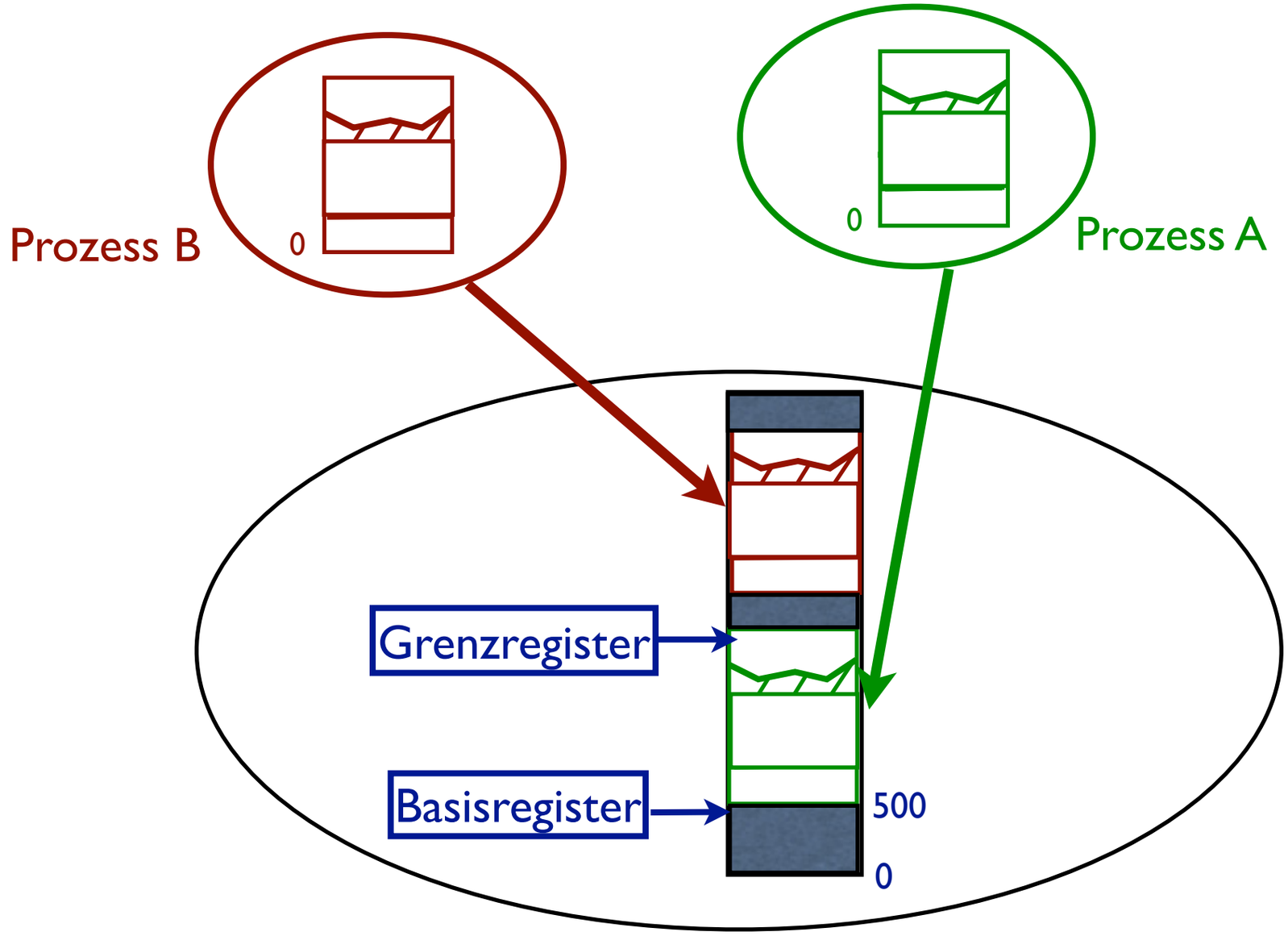


- Alle im Programm verwendeten virtuellen Adressen müssen auf reale Anfangsadresse umgestellt werden

  ⇒ Basisregister enthält Anfangsadresse des aktuellen Prozesses

Ad 2) Relocation (ca. 1970)

- Programme werden in unterschiedliche Hauptspeicherbereiche geschoben



- Adressraum der Prozesse gegeneinander schützen

  ⇒ Abfangen von ungültigen Adressen durch Grenzregister
  (letzte Adresse oder Länge)

⇒ Hauptspeicherverwaltung hat alle Probleme der externen Fragmentierung

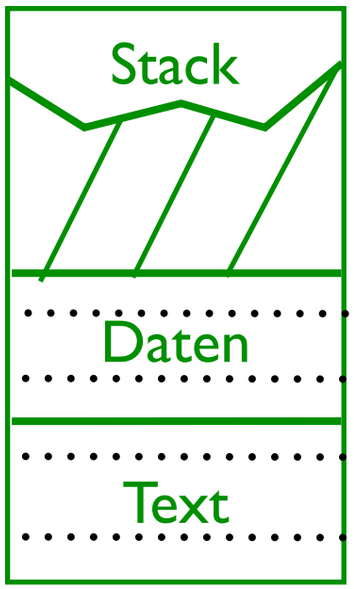
Fragen – Teil 3

- Wozu bieten Systeme eine *Speicherhierarchie* an? Welche Beobachtung über den Speicherzugriff realer Programme liegt dem zugrunde? Welche verschiedenen Arten von Speicher werden typischerweise bereitgestellt?
- Warum ist es in der Regel nicht sinnvoll, den Adressraum eines Prozesses in einem Stück im Hauptspeicher abzulegen?

Teil 4:
Segmentierung vs. Paging

Segmentierung

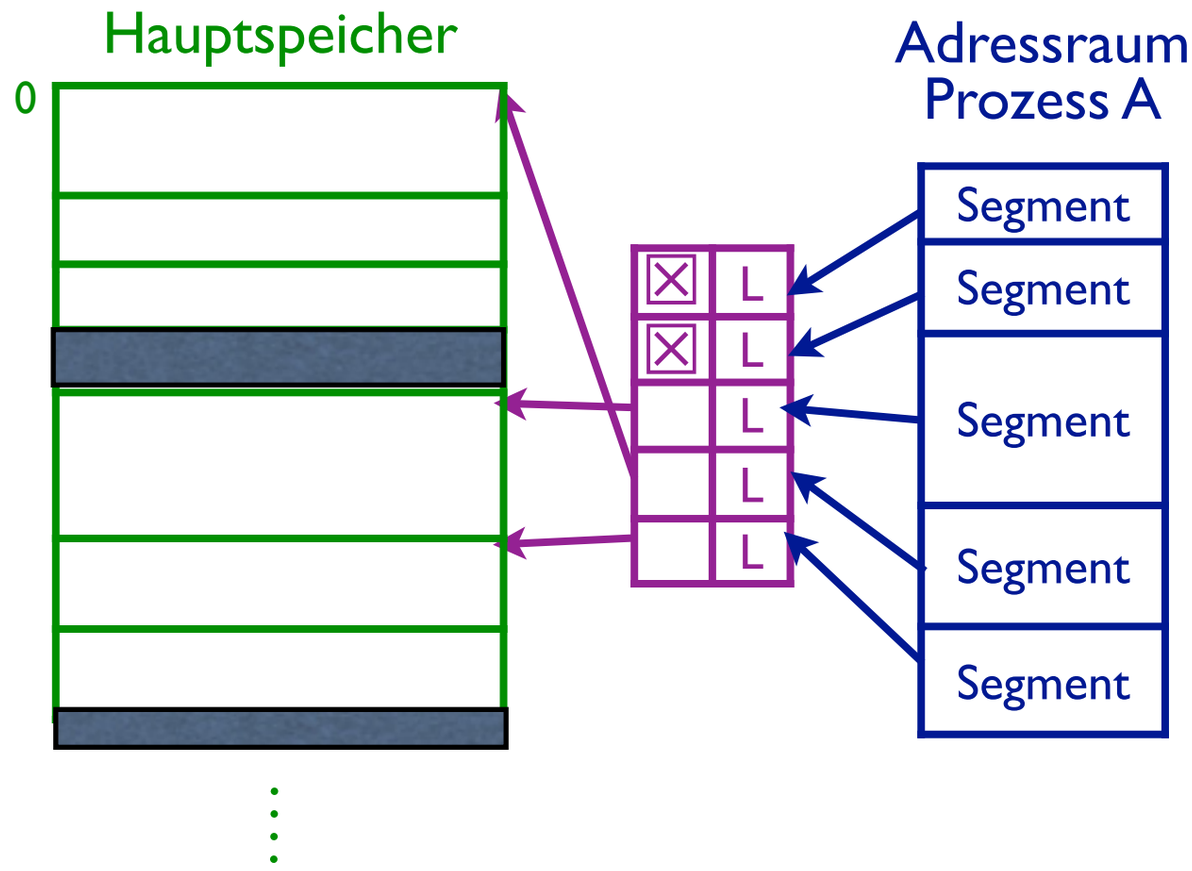
- Verfeinerung: Segmente des Adressraums bei Speicherzuteilung einzeln betrachten:



- Nutzung von kleineren Freispeicherbereichen
- Weitergehender Schutz möglich: z.B. read-only vs. read-write
- Gemeinsames Nutzen von Segmenten durch mehrere Prozesse („Shared Memory“)
- Zur Zeit nicht benutzte Segmente können aus Hauptspeicher ausgelagert werden (Lokalitätsprinzip)

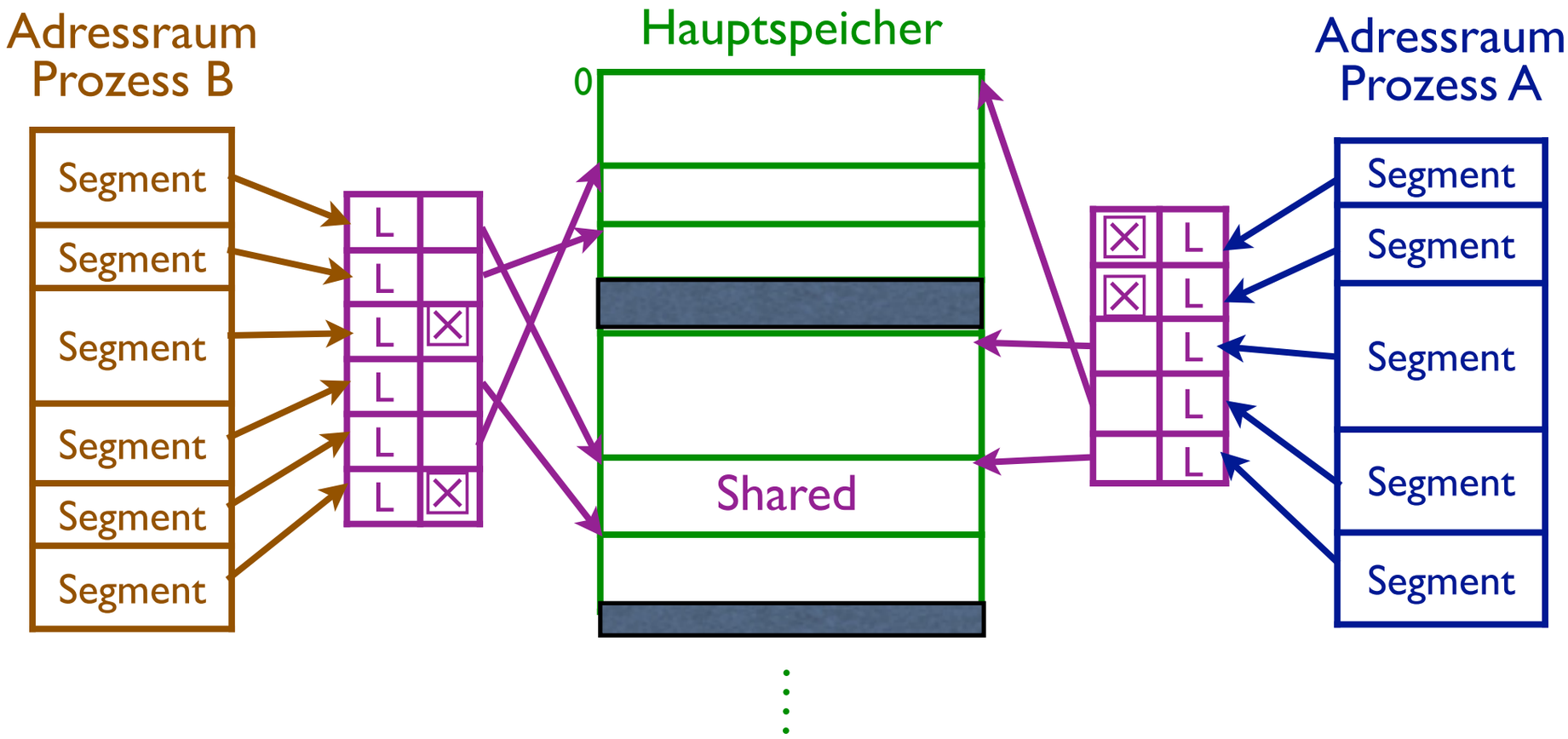
- Aber: Erhöhter Verwaltungsaufwand

  ⇒ Adressumsetzungstabellen pro Prozess



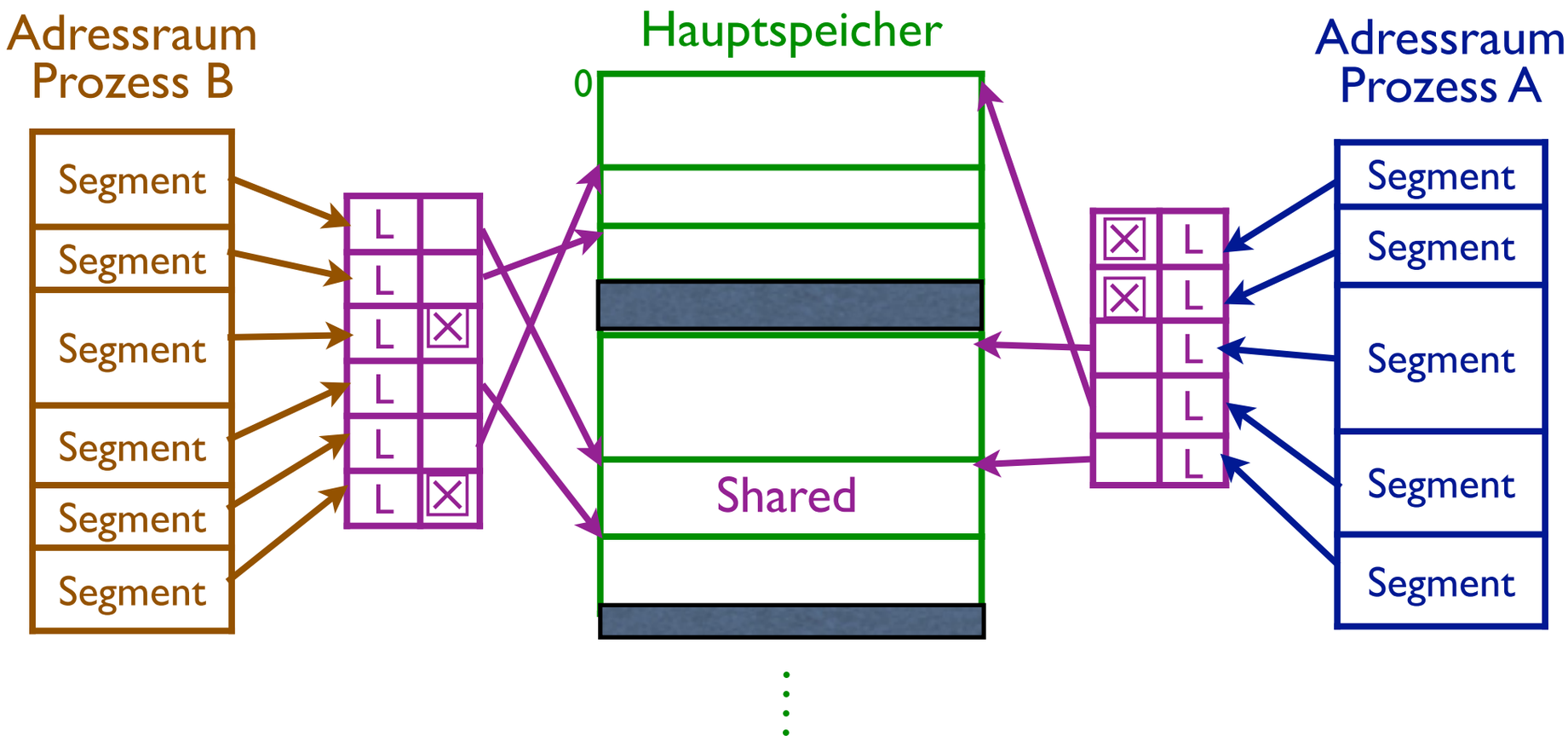
- Aber: Erhöhter Verwaltungsaufwand

  ⇒ Adressumsetzungstabellen pro Prozess



- Aber: Erhöhter Verwaltungsaufwand

  ⇒ Adressumsetzungstabellen pro Prozess



- Speicherverwaltung durch Segmentierung relativ aufwendig:
  - Unterschiedliche Länge von Segmenten
  - Probleme der externen Fragmentierung

Paging

- Ähnliche Grundprinzipien (flexible Zuteilung von Hauptspeicherbereichen, Sharing, ...)
- Aber:
  - Hauptspeicher in Kacheln (Page Frames) fester Größe eingeteilt (z.B. 2^{12} Bytes = 4096 Bytes = 4 KiB)
  - Prozessadressräume in Seiten (Pages) derselben Größe eingeteilt
  - Jede Page passt 1:1 in beliebigen Page Frame

    ⇒ keine externe Fragmentierung

    ⇒ Jedoch: „Letzte“ Seite muss nicht voll sein

    ⇒ interne Fragmentierung

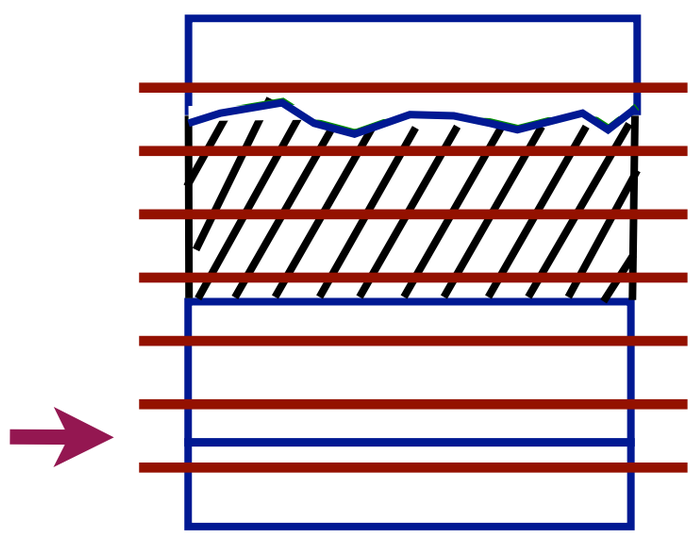
Paging

- Ähnliche Grundprinzipien (flexible Zuteilung von Hauptspeicherbereichen, Sharing, ...)
- Aber:
  - Hauptspeicher in Kacheln (Page Frames) fester Größe eingeteilt (z.B. 2^{12} Bytes = 4096 Bytes = 4 KiB)
  - Prozessadressräume in Seiten (Pages) derselben Größe eingeteilt
  - Jede Page passt 1:1 in beliebigen Page Frame

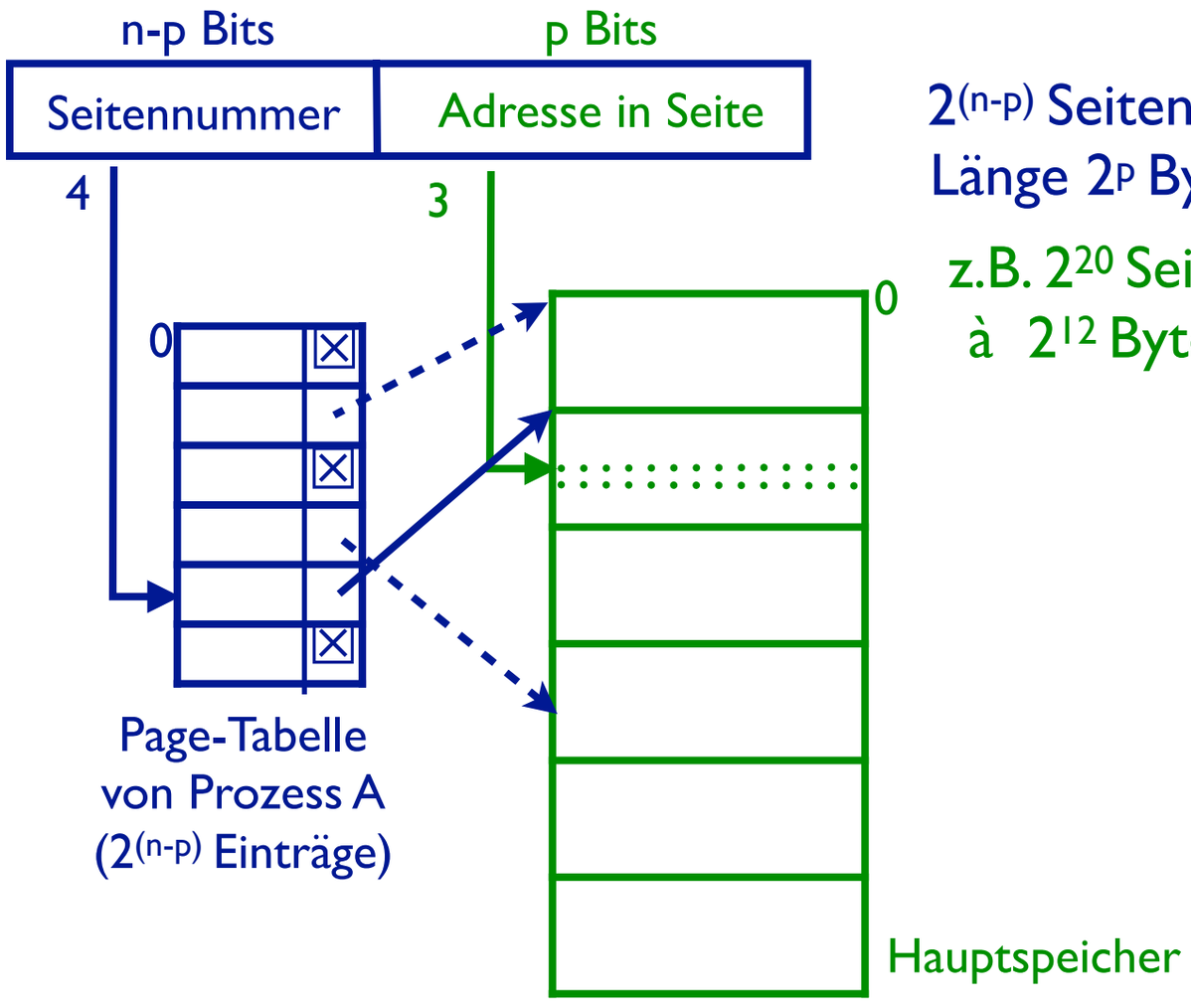
    ⇒ keine externe Fragmentierung

    ⇒ Jedoch: „Letzte“ Seite muss nicht voll sein

    ⇒ interne Fragmentierung
- Durch starre Seitenaufteilung zunächst Verlust der „logischen“ Aufteilung des Adressraums (Abhilfe: später)



- Seitengröße: 2^p (z.B. 4KiB bei p=12)
- Virtuelle Adresse: n Bits (z.B. n=32) aufteilbar in zwei Komponenten

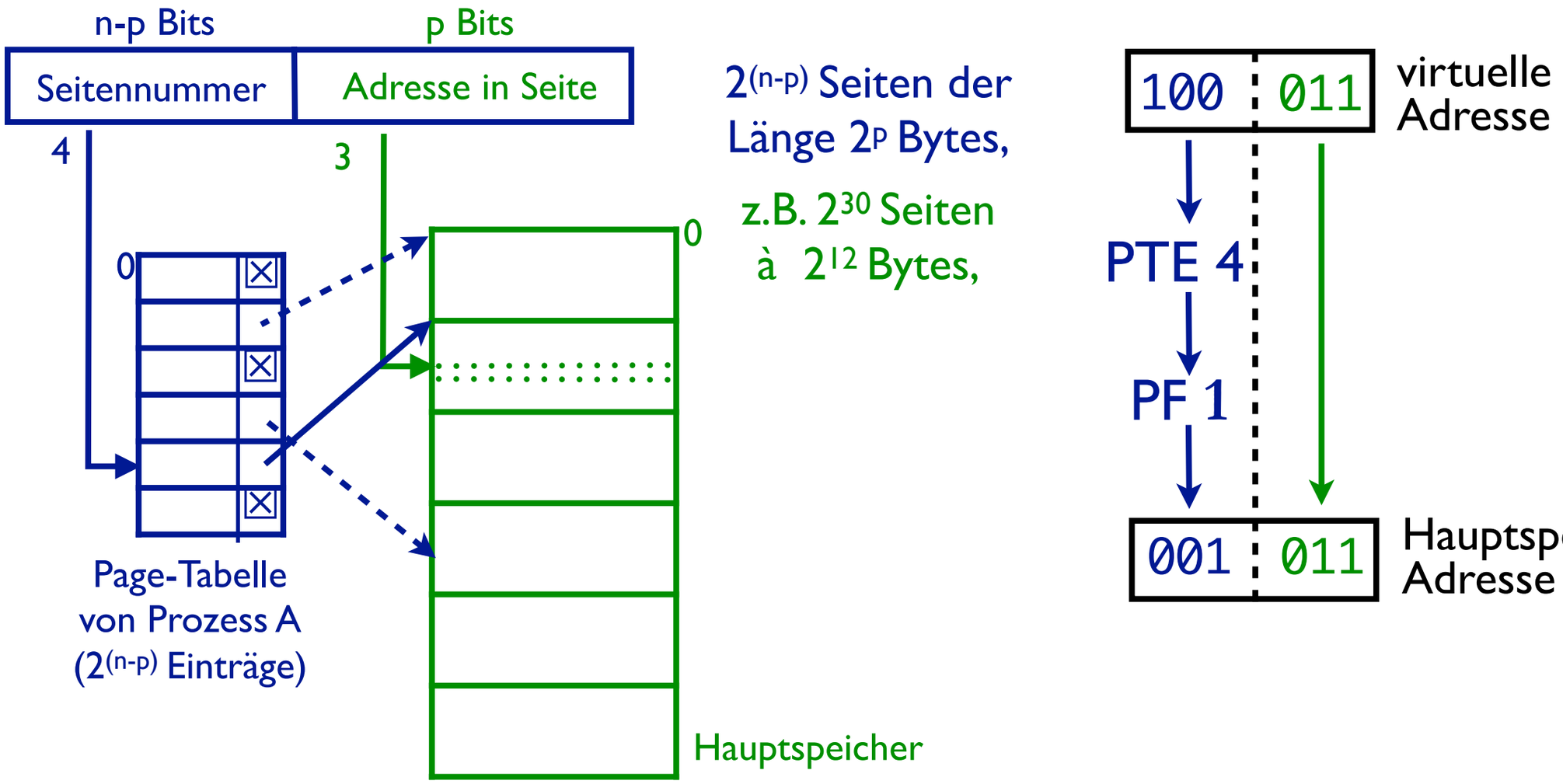


2(n-p) Seiten der Länge 2^p Bytes,

z.B. 2^{20} Seiten à 2^{12} Bytes,

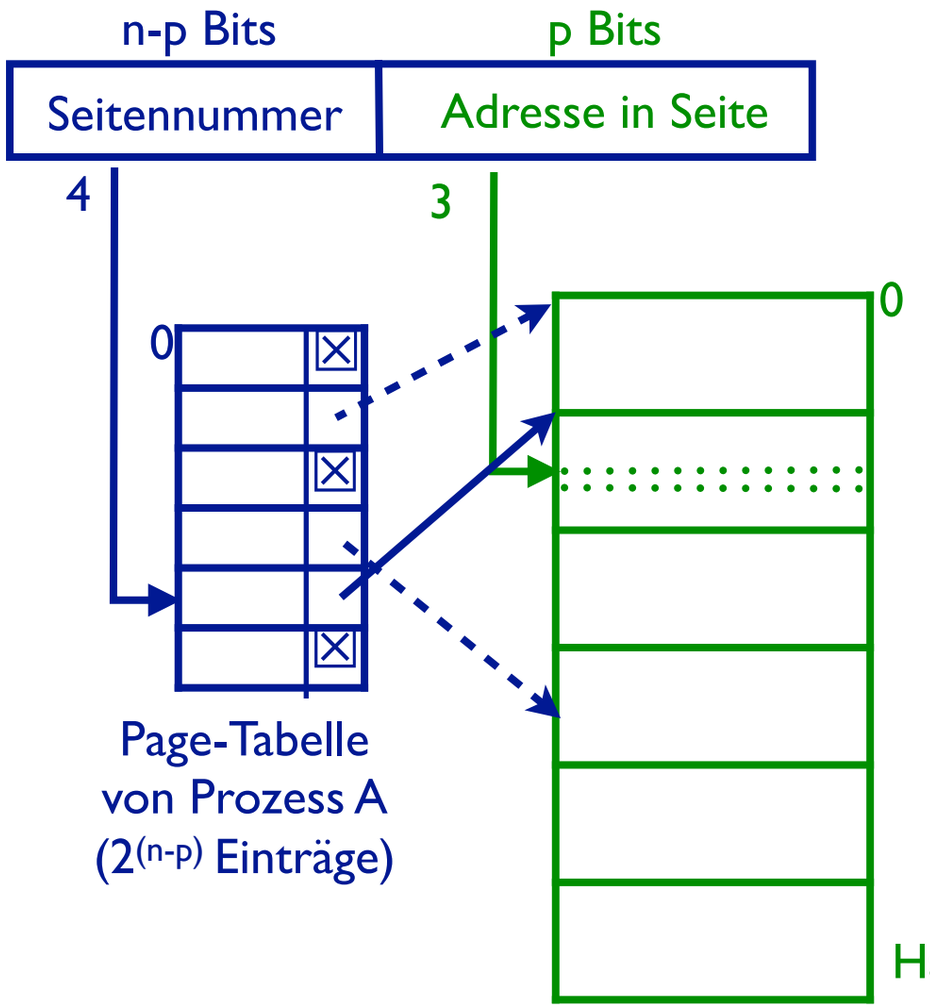
- Adressumsetzung über prozesseigene Page-Tabelle

- Seitengröße: 2^p (z.B. 4KiB bei p=12)
- Virtuelle Adresse: n Bits (z.B. n=32) aufteilbar in zwei Komponenten



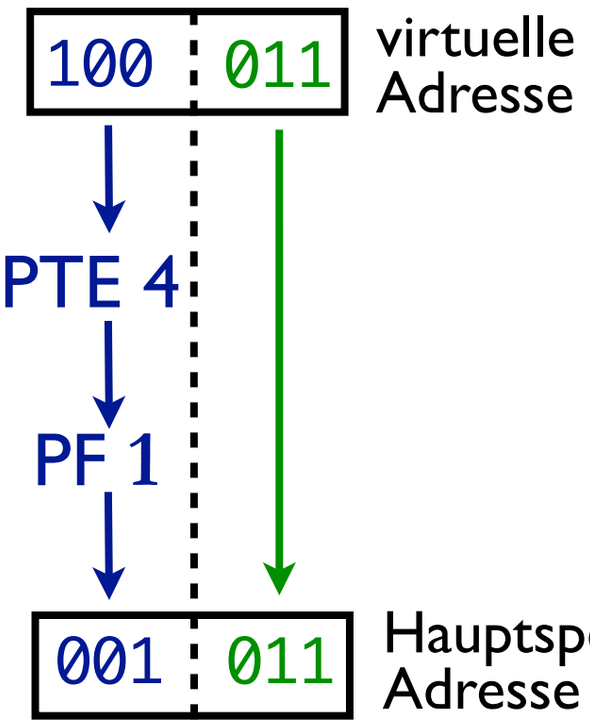
- Adressumsetzung über prozesseigene Page-Tabelle

- Seitengröße: 2^p (z.B. 4KiB bei p=12)
- Virtuelle Adresse: n Bits (z.B. n=32) aufteilbar in zwei Komponenten



$2^{(n-p)}$ Seiten der Länge 2^p Bytes,

z.B. 2^{30} Seiten à 2^{12} Bytes,



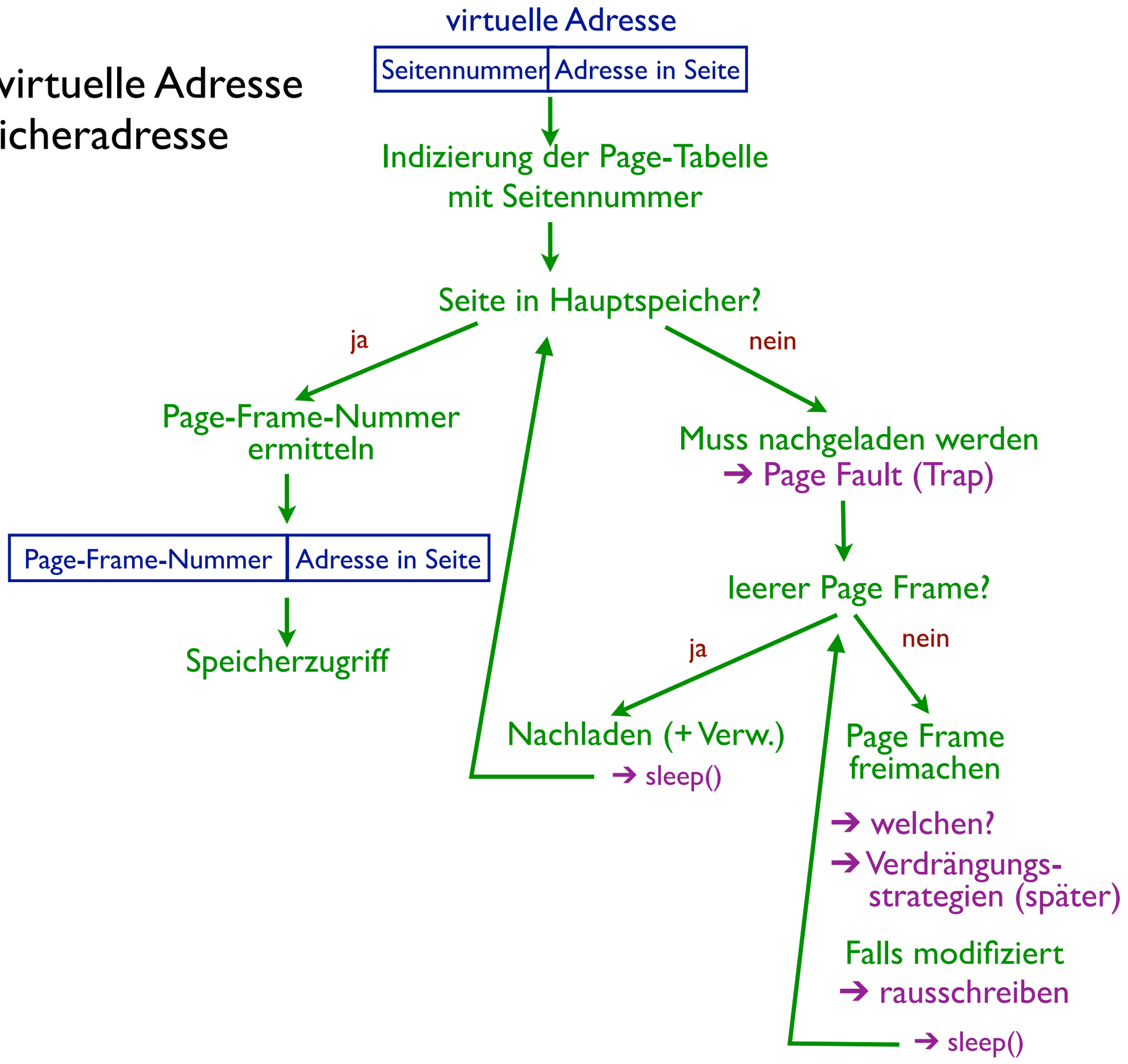
- Adressumsetzung über prozesseigene Page-Tabelle
- Unterstützt durch Hardware
  ⇒ Memory Management Unit (MMU)

- Page-Tabellen-Einträge (PTEs) enthalten: (vereinfacht)
  - Nummer des Page Frames im Hauptspeicher (oder Hintergrundspeicher)
  - Schutzbits (lesen, schreiben, ...)
  - Weitere Zustandsbits (gültig, modifiziert, ...)

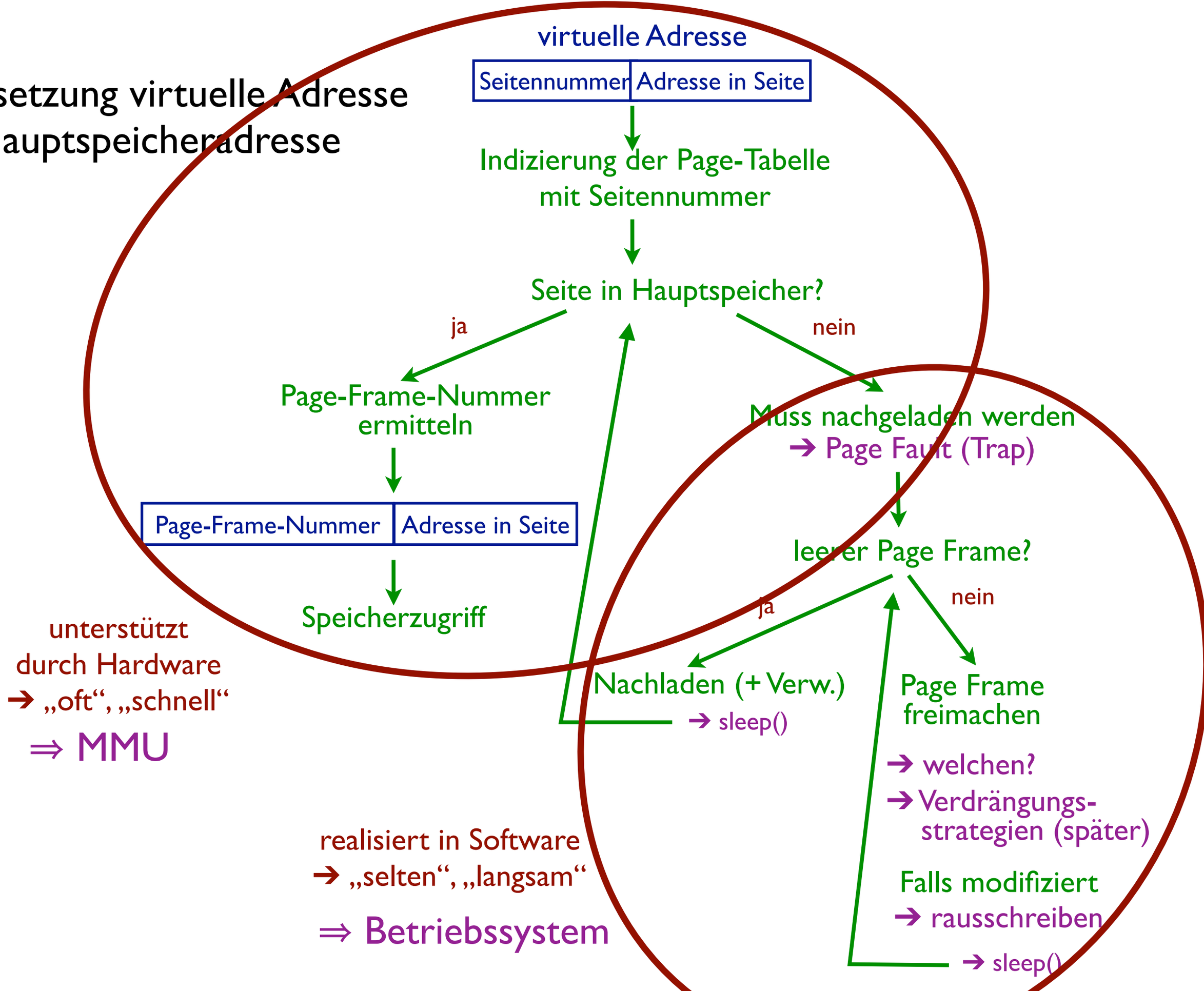
- Schutz: Prozess kann nur auf eigene Page-Tabelle und damit auf Seiten des eigenen Adressraums zugreifen

- Gemeinsame Informationen (Shared Memory):
  PTEs verschiedener Prozesse weisen auf denselben Page Frame (wie bei Segmentierung)

Umsetzung virtuelle Adresse in Hauptspeicheradresse



Umsetzung virtuelle Adresse in Hauptspeicheradresse



unterstützt durch Hardware
→ „oft“, „schnell“
⇒ MMU

realisiert in Software
→ „selten“, „langsam“
⇒ Betriebssystem

Fragen – Teil 4

- Was versteht man unter *Paging*, was unter *Segmentierung*?
- Aus welchen Teilen besteht eine *virtuelle Adresse* zumeist? Wie ermittelt sich daraus die entsprechende Hauptspeicheradresse, d.h. wie läuft die Adressverwaltung in etwa ab?
- Wie können mehrere Prozesse mit Hilfe virtueller Adressierung auf dieselben Programmstücke (oder auch Datenbereiche) zugreifen?

Zusammenfassung

- Speicherverwaltung innerhalb eines Prozesses:
  - First-Fit, Next-Fit, Best-Fit, Buddy
  - externe vs. interne Fragmentierung
- Abbildung virtueller Adressraum → Hauptspeicher
  - Speicherhierarchie (+ Exkurs Maßeinheiten)
  - Lokalitätsprinzip, Working Set
  - Relocation, Segmentierung (Shared Memory)
  - Paging
    - Pages vs. Page Frames
    - Adressabbildung über MMU (Page Tabelle, Page Faults)

Speicherverwaltung 1 – Fragen

1. Wie verändert sich die Größe der verschiedenen Segmente des Prozess-Adressraums beim Programmablauf?
2. Welche Vor- und Nachteile hat der *First-Fit*- bzw. der *Best-Fit*-Algorithmus zur Speicherverwaltung?
3. Wie arbeitet der *Buddy-Algorithmus* in etwa?
4. Wo tritt *interne Fragmentierung*, wo *externe Fragmentierung* auf? Was ist das?
5. Wozu bieten Systeme eine *Speicherhierarchie* an? Welche Beobachtung über den Speicherzugriff realer Programme liegt dem zugrunde? Welche verschiedenen Arten von Speicher werden typischerweise bereitgestellt?
6. Warum ist es in der Regel nicht sinnvoll, den Adressraum eines Prozesses in einem Stück im Hauptspeicher abzulegen?
7. Was versteht man unter *Paging*, was unter *Segmentierung*?
8. Aus welchen Teilen besteht eine *virtuelle Adresse* zumeist? Wie ermittelt sich daraus die entsprechende Hauptspeicheradresse, d.h. wie läuft die Adressverwaltung in etwa ab?
9. Wie können mehrere Prozesse mit Hilfe virtueller Adressierung auf dieselben Programmstücke (oder auch Datenbereiche) zugreifen?