

Systeme I: Betriebssysteme

Kapitel 8 **Speicherverwaltung**

Maren Bennewitz



Inhalt Vorlesung

- Aufbau einfacher Rechner
- Überblick: Aufgabe, Historische Entwicklung, unterschiedliche Arten von Betriebssystemen
- Verschiedene Komponenten / Konzepte von Betriebssystemen
 - Dateisysteme
 - Prozesse
 - Nebenläufigkeit und wechselseitiger Ausschluss
 - Deadlocks
 - Scheduling
 - **Speicherverwaltung**

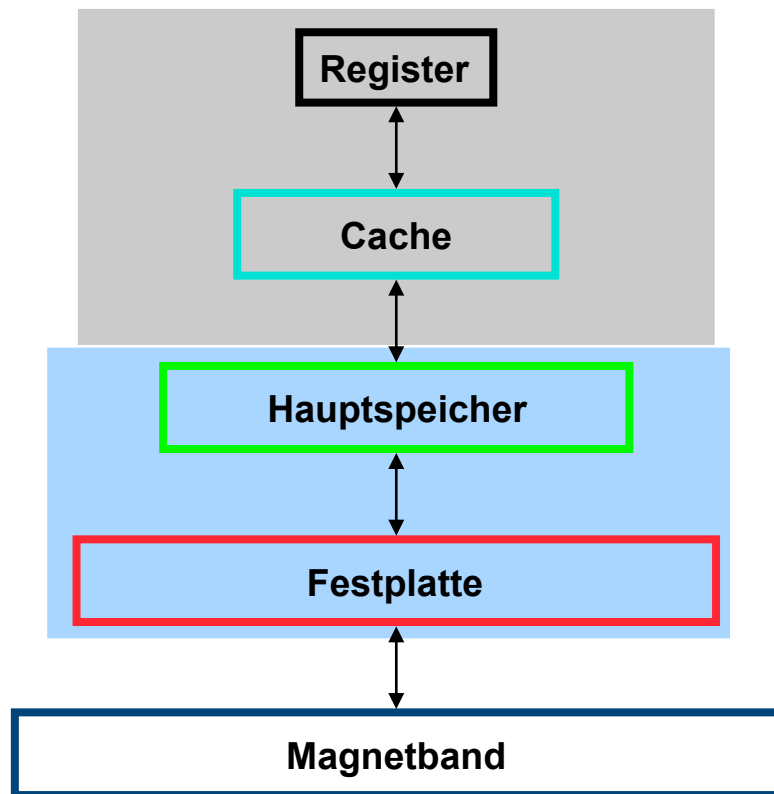
Einführung

- Hauptspeicher ist in verschiedene Bereiche aufgeteilt
 - Bereich für das Betriebssystem
 - Bereich für verschiedene Prozesse
- Speicherverwaltung: Dynamische Aufteilung entsprechend aktueller Prozesse
- Speicher muss effizient aufgeteilt werden, damit möglichst viele Prozesse Platz haben

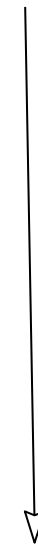
Adressraum

- Speicherzellen im Hauptspeicher haben eindeutige Adresse
- Abstraktion vom physikalischen Speicher
- Adressraum: Menge von Adressen, die ein Prozess benutzen darf (lesen / schreiben)
- Jeder Prozess hat seinen eigenen Adressraum

Speicherhierarchie



Kleine Kapazität, kurze Zugriffszeit,
hohe Kosten pro Bit



Große Kapazität, längere Zugriffszeit,
niedrige Kosten pro Bit

Anforderungen an Speicherverwaltung

- Bereitstellung von Platz im Hauptspeicher für Betriebssystem und Prozesse
- Ziel aus Betriebssystemersicht: Möglichst viele Prozesse im Speicher
- Fünf wichtige Anforderungen:
 - Relokation
 - Schutz
 - Gemeinsame Nutzung
 - Logische Organisation
 - Physikalische Organisation

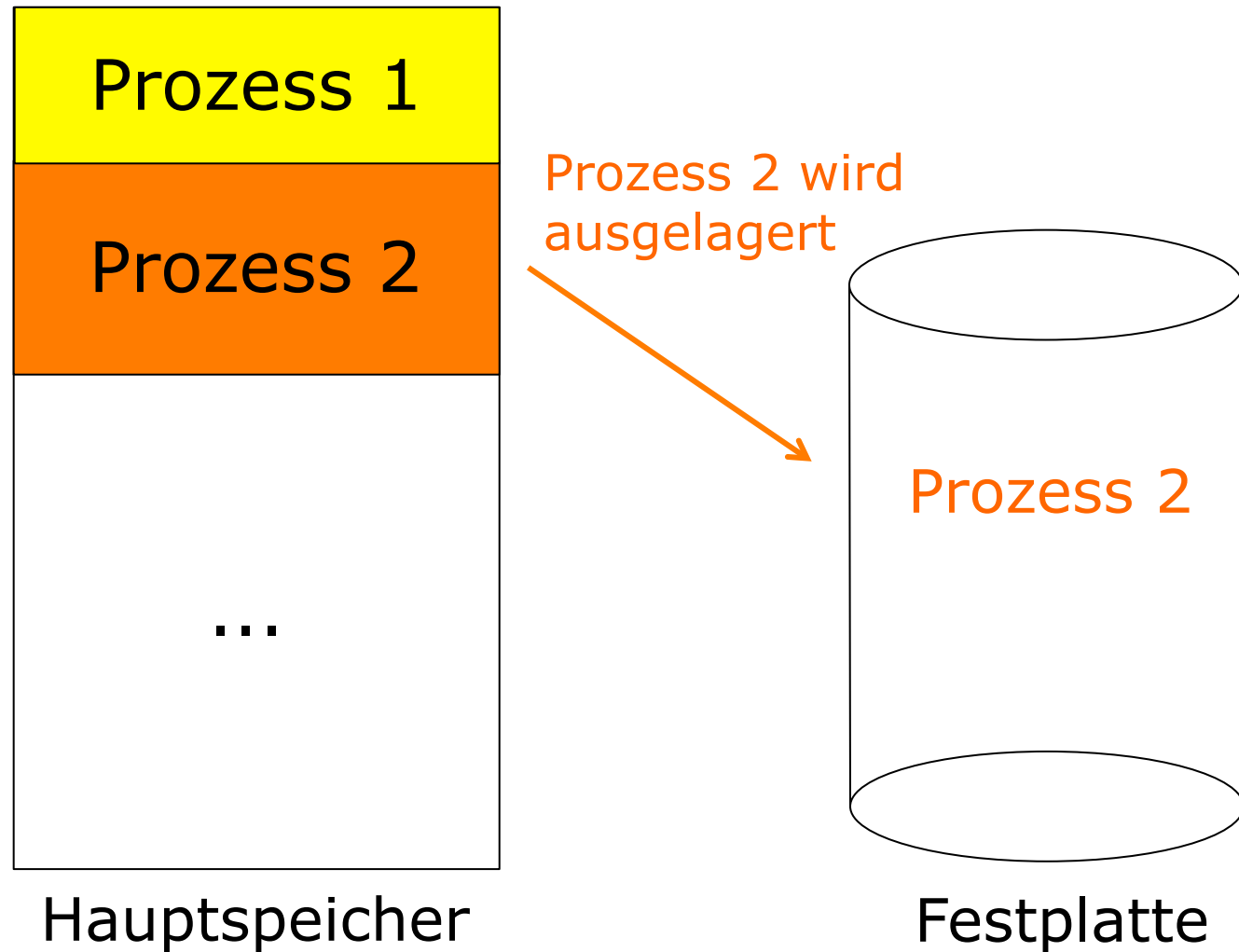
Relokation (1)

- Relokation = Verlagerung
- Mehrere Prozesse gleichzeitig im System
- Auslagern und Wiedereinlagern von Prozessen aus dem Hauptspeicher
- Ort der Einlagerung im Voraus **unbekannt**

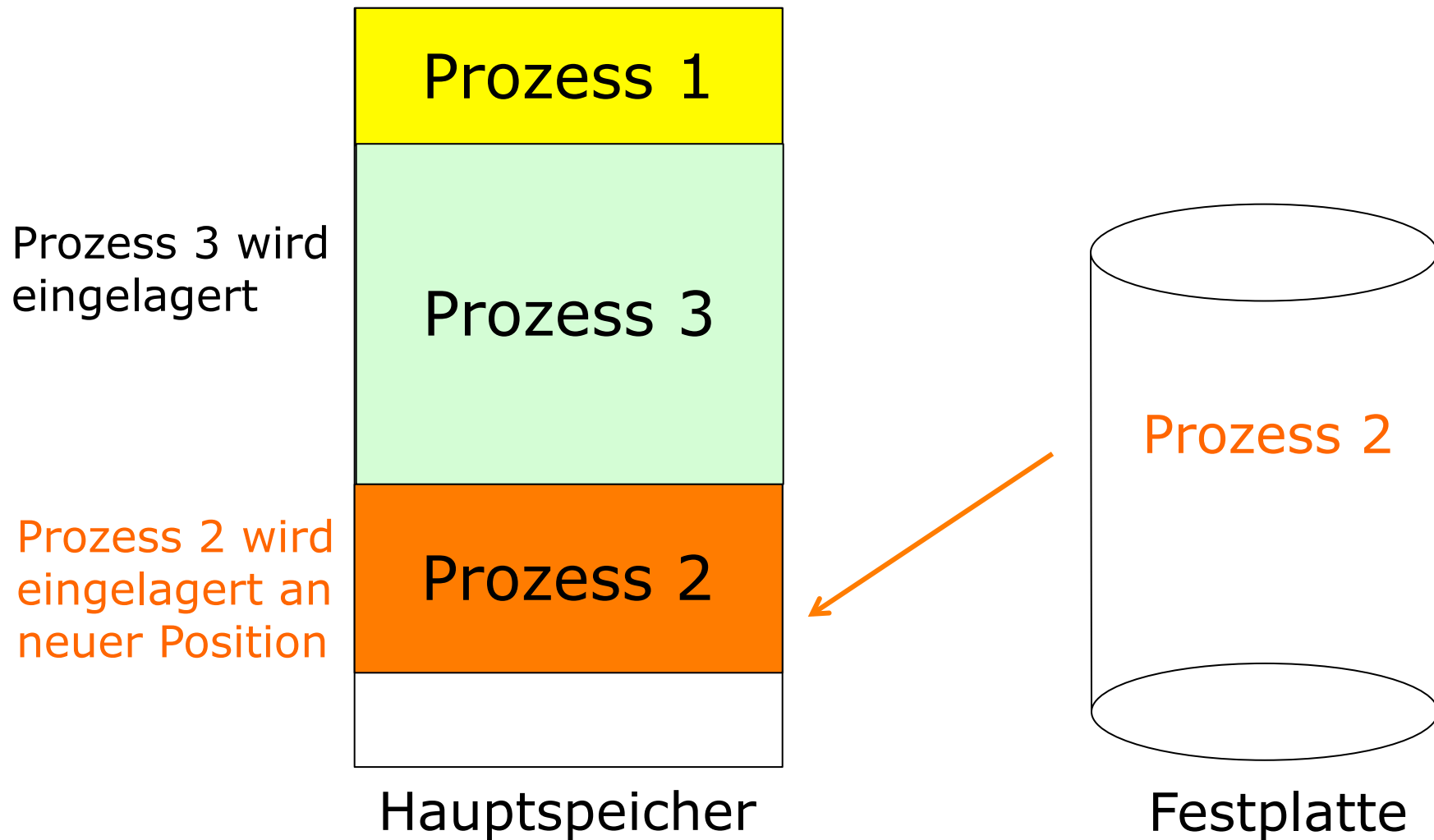
Relokation (2)

- Problem: Speicherreferenzen innerhalb des Programms
- Absolute Sprungbefehle: Adresse auf den nächsten auszuführenden Befehl
- Datenzugriffsbefehle: Adresse des Bytes, das referenziert wird
- Prozessorhardware und Betriebssystem müssen die Speicherreferenzen in physikalische Speicheradressen übersetzen

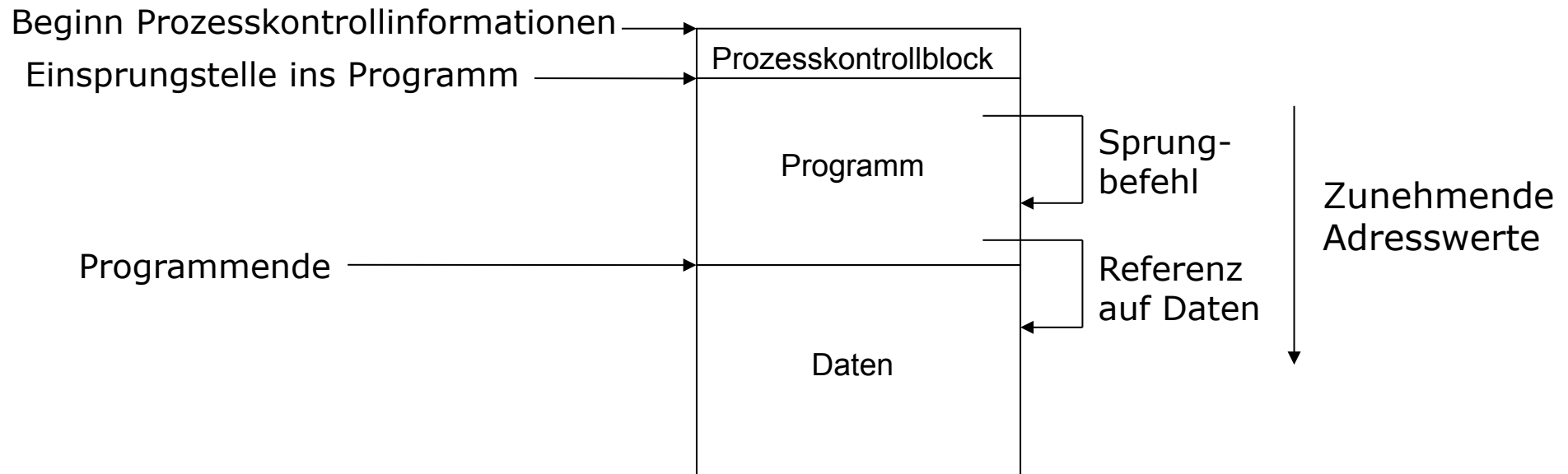
Relokation (3)



Relokation (4)



Relokation (5)



- Übersetzung der Speicherreferenzen im Programmcode in physikalische Speicheradressen
- Um z.B. Maschinenbefehle wie JUMP und LOAD zu realisieren

Schutz (1)

- Schutz von Prozessen gegen Störungen durch andere Prozesse
- Überprüfung aller Speicherzugriffe notwendig
- Schwierigkeit: Nicht zur Übersetzungszeit eines Programms überprüfbar
- Grund: Dynamisch berechnete Adressen während der Laufzeit, absolute Adressen nicht bekannt

Schutz (2)

- Dynamische Überprüfung zur Laufzeit
- Ggf. Abbruch von Befehlen bei Zugriff auf Datenbereich anderer Prozesse

Gemeinsame Nutzung

- Kontrollierter Zugriff mehrerer Prozesse auf gemeinsam genutzte Bereiche des Speichers
- Anwendungsbeispiele:
 - Ausführung des gleichen Programms durch eine Reihe von Prozessen, Code nur einmal im Speicher
 - Zugriff auf dieselbe Datenstruktur bei Zusammenarbeit von Prozessen
 - Kooperation von Prozessen über gemeinsam genutzten Datenspeicher („Shared Memory“)

Logische Organisation

- Logischer Aufbau großer Programme:
 - Verschiedene Module
 - Unabhängig übersetzt; Referenzen auf Funktionen in anderen Modulen werden erst zur Laufzeit aufgelöst
 - Verschiedene Module können unterschiedliche Grade von Schutz besitzen (z.B. nur lesen / ausführen)
 - Gemeinsame Nutzung von Modulen durch verschiedene Prozesse
- Betriebssystem muss mit Modulen umgehen können

Physikalische Organisation

- Betrachte zwei Ebenen
 - Hauptspeicher (schnell, teuer, flüchtig)
 - Festplatte (langsam, billig, nicht flüchtig)
- Grundproblem: Daten zwischen Haupt- und Sekundärspeicher verschieben
 - Aufwändig, erschwert durch Multiprogramming
 - Verwaltung durch das Betriebssystem

Grundlegende Methoden der Speicherverwaltung

Partitionierung

- Speicheraufteilung zwischen verschiedenen Prozessen (Partitionierung mit festen Grenzen)

Paging

- Einfaches Paging / kombiniert mit Konzept des virtuellen Speichers

Segmentierung

- Einfache Segmentierung / kombiniert mit Konzept des virtuellen Speichers

Partitionierung

- Aufteilung des Speichers in Bereiche mit festen Grenzen
- Fester, zusammenhängender Teil des Hauptspeichers für Betriebssystem
- Pro Prozess ein zusammenhängender Teil des Speichers
- Verschiedene Varianten:
 - Statische Partitionierung
 - Dynamische Partitionierung
 - Buddy-Verfahren

Statische Partitionierung (1)

- Einteilung des Speichers in **fixe Anzahl von Partitionen**
- Zwei Varianten

Alle Partitionen mit gleicher Länge

Betriebssystem 8 MB
8 MB
8 MB
8 MB
8 MB
8 MB
8 MB
8 MB

Alle Partitionen mit unterschiedlicher Länge

Betriebssystem 8 MB
2 MB
2 MB
4 MB
8 MB
12 MB
16 MB

Statische Partitionierung (2)

Zuweisung von Partitionen an Prozesse:

- Bei Bereichen mit gleicher Länge: trivial
- Bei Bereichen mit variabler Länge: Kleinste verfügbare Partition, die gerade noch ausreicht (Verwaltung nicht trivial)
- Oft Speicherbedarf nicht im Voraus feststellbar (dafür Verfahren des virtueller Speichers, siehe später)

Statische Partitionierung (3)

Probleme bei gleich großen Partitionen:

- Programm zu groß für Partition
- Interne Fragmentierung:
Platzverschwendung, wenn Programm kleiner als Größe der zugeordneten Partition
- Fest vorgegebene Anzahl von Prozessen im Speicher

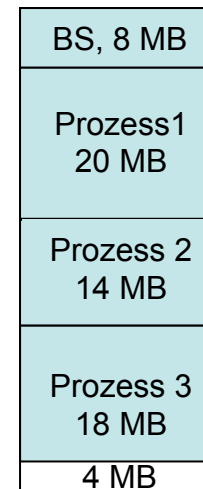
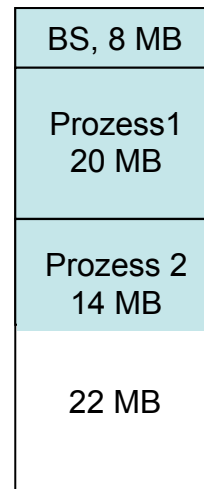
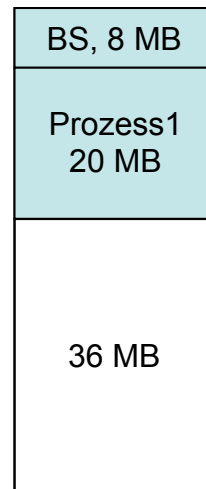
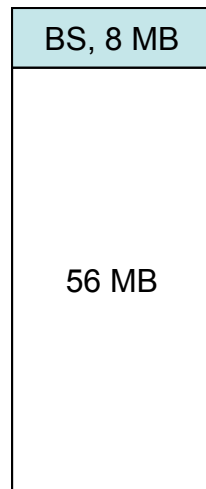
Partitionen variabler Länge

- Größere Programme können untergebracht werden
- Kleinere Programme führen zu geringerer interner Fragmentierung

Dynamische Partitionierung (1)

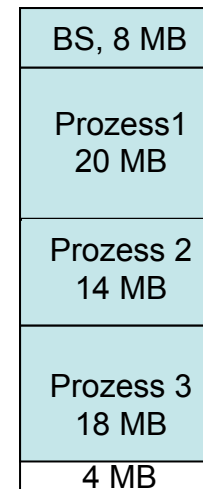
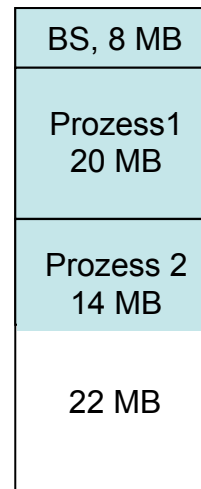
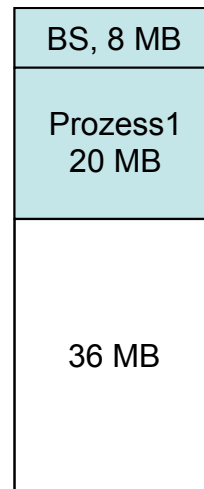
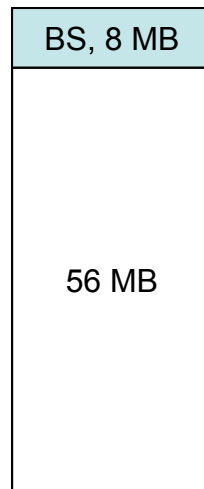
- Einteilung des Speichers in Partitionen
 - variable Länge
 - variable Anzahl
- Prozesse erhalten exakt passende Speicherbereiche (keine interne Fragmentierung)
- Aber: Ein- und Auslagern führt zu **externer Fragmentierung**, Vielzahl kleiner Lücken, Speicherauslastung nimmt ab

Dynamische Partitionierung (2)

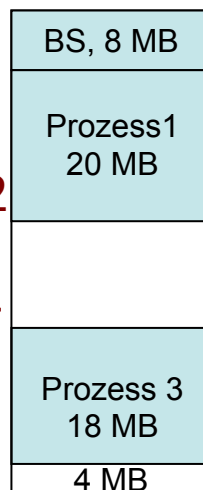


Anforderung:
Prozess 4
braucht 8 MB

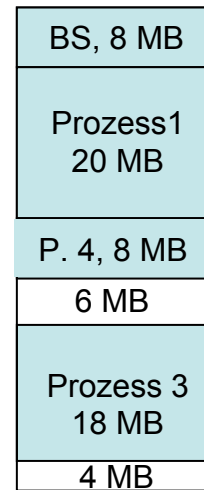
Dynamische Partitionierung (2)



Anforderung:
Prozess 4
braucht 8 MB



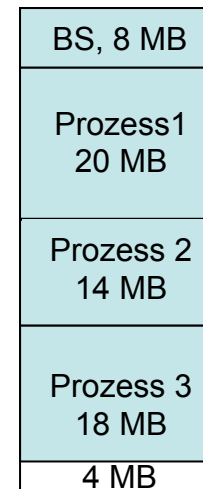
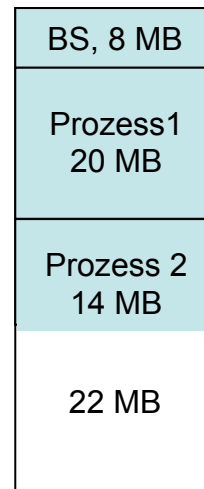
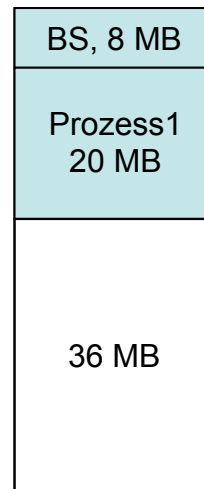
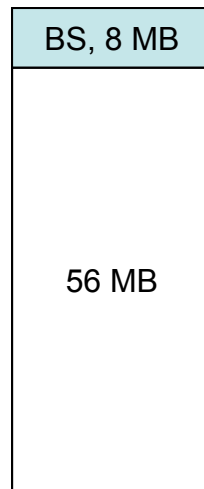
Proz. 2
wird
ausge-
lagert



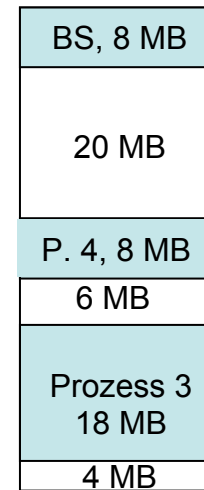
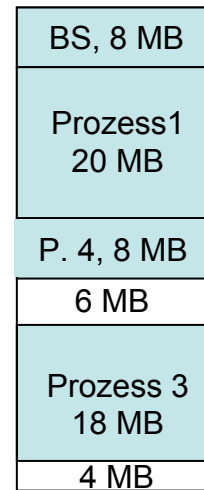
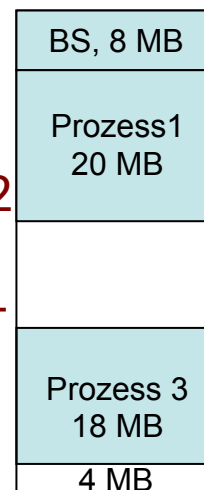
genügend Platz für
Prozess 4, aber
Lücke entsteht

Annahme: Kein
Prozess im
Hauptspeicher
bereit, aber
ausgelagerter
Prozess 2 bereit

Dynamische Partitionierung (2)



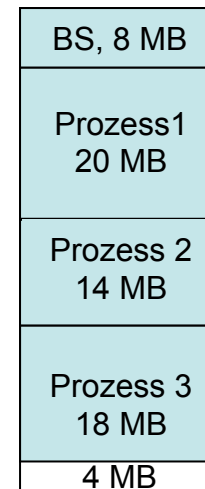
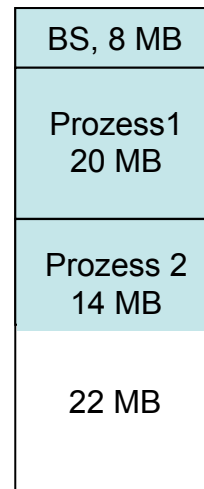
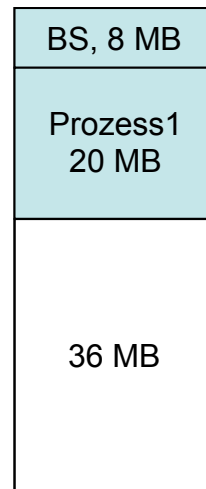
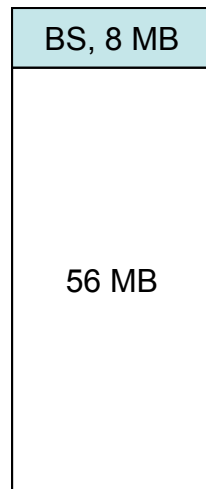
Anforderung:
Prozess 4
braucht 8 MB



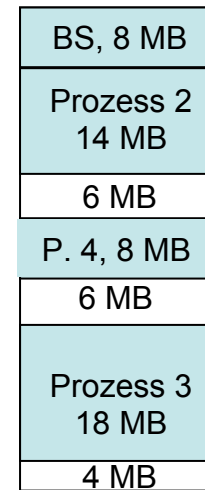
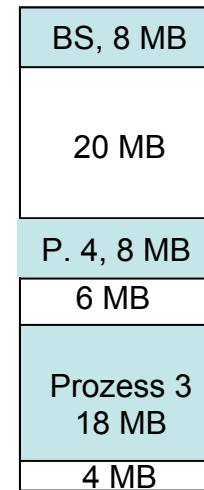
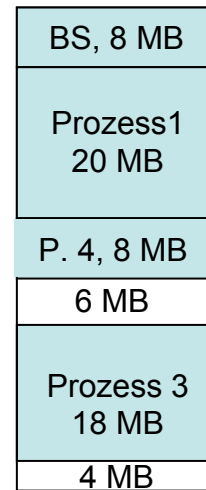
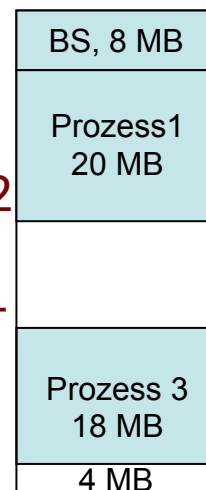
Proz. 2
wird
ausge-
lagert

da nicht genügend
Platz für Prozess 2:
Prozess 1 wird
ausgelagert

Dynamische Partitionierung (2)



Anforderung:
Prozess 4
braucht 8 MB



Prozess 2 wird
wieder eingelagert

Proz. 2
wird
ausge-
lagert

Dynamische Partitionierung (3)

Defragmentierung möglich, aber

- Verschiebung aufwändig:
Speicherzuteilungsstrategie wichtig
- Speicherverdichtung nur erfolgreich, wenn
dynamische Relokation möglich

Dynamische Partitionierung (4)

Speicherzuteilungsalgorithmen:

- **Best Fit:** Suche kleinsten Block, der ausreicht
- **First Fit:** Suche beginnend mit Speicheranfang bis ausreichend großer Block gefunden
- **Next Fit:** Suche beginnend mit der Stelle der letzten Speicherzuweisung

Dynamische Partitionierung (5)

Analyse der Speicherzuteilungsalgorithmen:

- Im Schnitt ist First Fit am besten!
- Next Fit: Etwas schlechter
 - Typischer Effekt: Schnelle Fragmentierung des größten freien Speicherblocks am Ende des Speichers
- Best Fit: Am schlechtesten
 - Schnell eine Reihe von sehr kleinen Fragmenten, Defragmentierung nötig
 - Außerdem: Suche braucht Zeit

Nachteile Partitionierung

- **Statische** Partitionierung:
 - Anzahl von Prozessen im Speicher beschränkt
 - Interne Fragmentierung
- **Dynamische** Partitionierung:
 - Schwierigere Verwaltung
 - Externe Fragmentierung
- **Buddy-System** (Halbierungsverfahren):
Kompromiss zwischen statischer und dynamischer Partitionierung

Buddy System (1)

- Dynamische Anzahl nicht-ausgelagerter Prozesse
- Interne Fragmentierung **beschränkt**
- **Keine** explizite Defragmentierung
- **Effiziente Suche** nach „passendem Block“

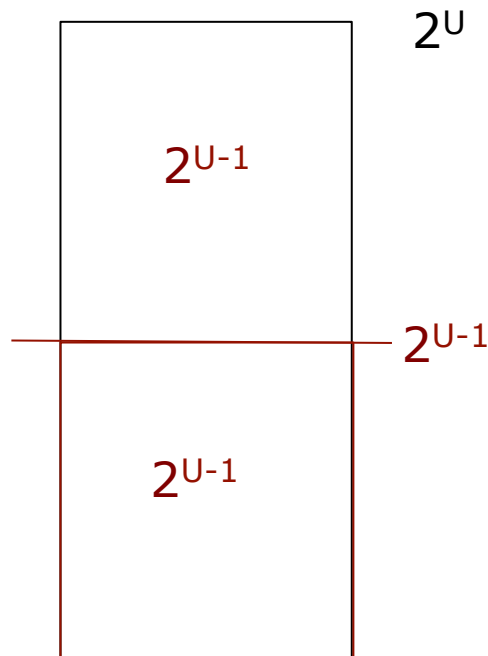
Buddy System (2)

- Verwalte Speicherblöcke der Größe 2^K
 - mit $L \leq K \leq U$, wobei
 - 2^L = Größe des kleinsten zuteilbaren Blocks
 - 2^U = Größe des größten zuteilbaren Blocks
(in der Regel Gesamtgröße des verfügbaren Speichers)
- Zu Beginn: Es existiert genau ein Block der Größe 2^U

Buddy System (3)

Anforderung eines Blocks der Größe s :

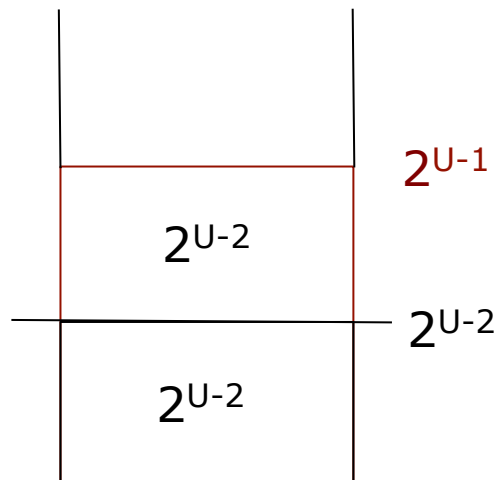
- Wenn $2^{U-1} < s \leq 2^U$: Weise gesamten Speicher zu
- Sonst: Teile auf in zwei Blöcke der Größe 2^{U-1}



Buddy System (4)

Anforderung eines Blocks der Größe s :

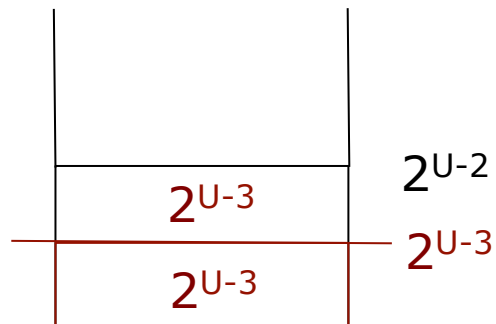
- Wenn $2^{U-1} < s \leq 2^U$: Weise gesamten Speicher zu
- Sonst: Teile auf in zwei Blöcke der Größe 2^{U-1}
- Wenn $2^{U-2} < s \leq 2^{U-1}$: Weise Block der Größe 2^{U-1} zu
- Sonst: Wähle Block der Größe 2^{U-1} und halbiere



Buddy System (4)

Anforderung eines Blocks der Größe s :

- Wenn $2^{U-1} < s \leq 2^U$: Weise gesamten Speicher zu
- Sonst: Teile auf in zwei Blöcke der Größe 2^{U-1}
- Wenn $2^{U-2} < s \leq 2^{U-1}$: Weise Block der Größe 2^{U-1} zu
- Sonst: Wähle Block der Größe 2^{U-1} und halbiere
- ...



Buddy System (4)

Anforderung eines Blocks der Größe s :

- Wenn $2^{U-1} < s \leq 2^U$: Weise gesamten Speicher zu
- Sonst: Teile auf in zwei Blöcke der Größe 2^{U-1}
- Wenn $2^{U-2} < s \leq 2^{U-1}$: Weise Block der Größe 2^{U-1} zu
- Sonst: Wähle Block der Größe 2^{U-1} und halbiere
- ...
- Fahre fort bis zu Blöcken der Größe 2^K mit $2^{K-1} < s \leq 2^K$
- Weise einen der beiden Blöcke zu (bzw. weise Block mit Minimalgröße zu)

Vorteil: Bei resultierendem Block ist der Verschnitt kleiner als die halbe Blockgröße

Buddy System (5)

- Verwalte für alle $L \leq K \leq U$ Listen mit freien Blöcken der Größe 2^K
- Allgemeiner Fall: Anforderung eines Blocks der Größe $2^{i-1} < s \leq 2^i$:
 - Vergib Block aus Liste i , wenn vorhanden
 - Sonst: Wähle Block aus nächstgrößerer nichtleerer Liste
 - Teile diesen rekursiv auf, bis ein Block der Größe 2^i vorhanden, weise diesen zu (bzw. weise Block mit Minimalgröße zu)

Buddy System (6)

- Wenn nach Freigabe eines Blocks der Größe 2^k der entsprechende **Partnerblock** der Größe 2^k ebenfalls frei ist:
 - Fasse die Blöcke zu einem Block der Größe 2^{k+1} zusammen
 - Mache ggf. rekursiv weiter
- Binärbaumdarstellung der Blockzuteilung

Buddy-System (7)

Beispiel: Speicher der Größe 1 GiB

Folge von Anforderungen und Freigaben:

- A fordert 100 MiB an
- B fordert 240 MiB an
- C fordert 64 MiB an
- D fordert 256 MiB an
- Freigabe B
- Freigabe A
- E fordert 75 MiB an
- Freigabe C
- Freigabe E
- Freigabe D

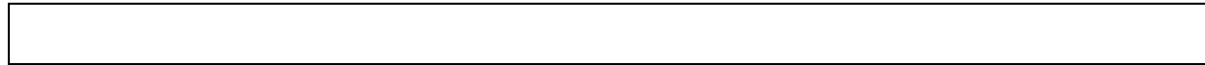
Annahme:

- Obergrenze der Blockgröße: 1 GiB
- Untergrenze: 64 MiB

Buddy-System (8)

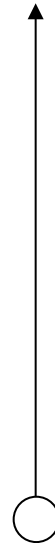
Zunächst verfügbar:

1 GiB



Freie Blöcke:

1 GiB: 1
512 MiB: 0
256 MiB: 0
128 MiB: 0
64 MiB: 0



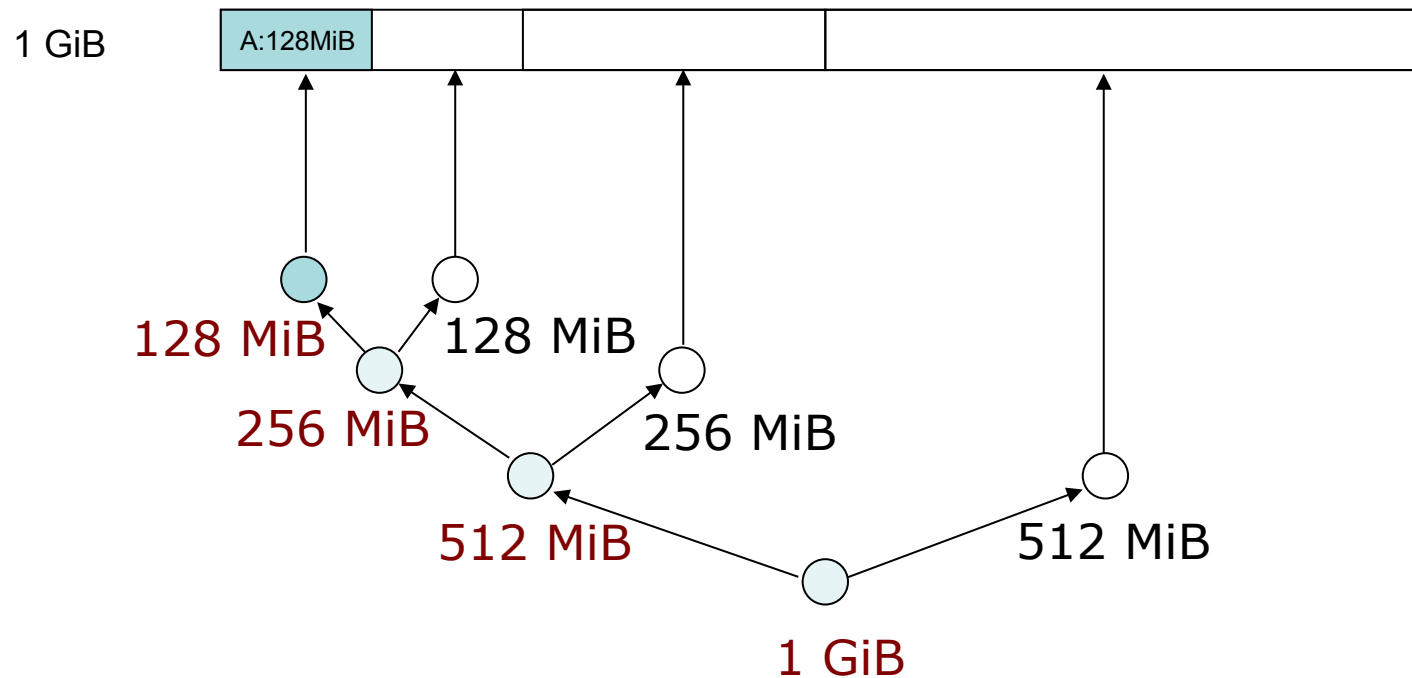
Gesamter Speicher als 1 Block verfügbar

Es folgt Anforderung A: 100 MiB, d.h. Block der Größe 128 MiB

Buddy-System (9)

Nach Anforderung A: 100 MiB, d.h. Block der Größe 128 MiB:

Freie Blöcke:



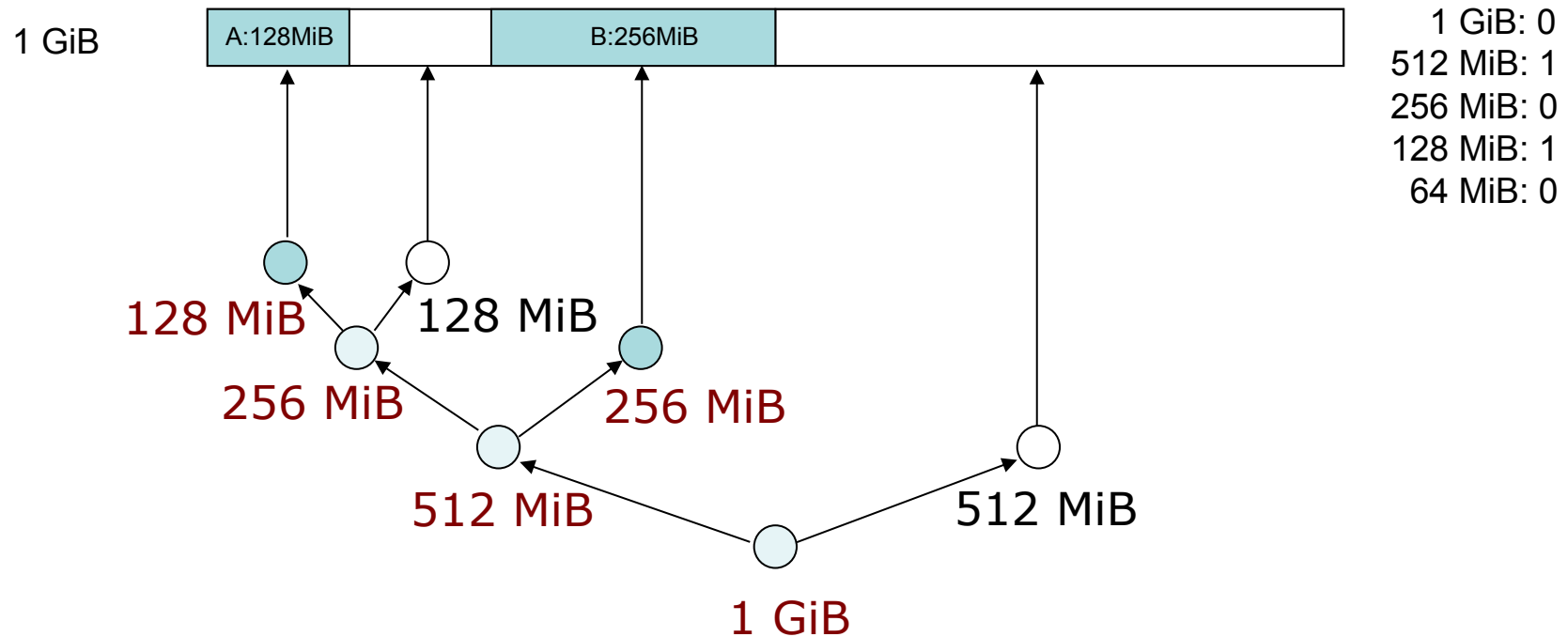
1 GiB: 0
512 MiB: 1
256 MiB: 1
128 MiB: 1
64 MiB: 0

Es folgt Anforderung B: 240 MiB, d.h. Block der Größe 256 MiB

Buddy-System (10)

Nach Anforderung B: 240 MiB, d.h. Block der Größe 256 MiB.

Freie Blöcke:

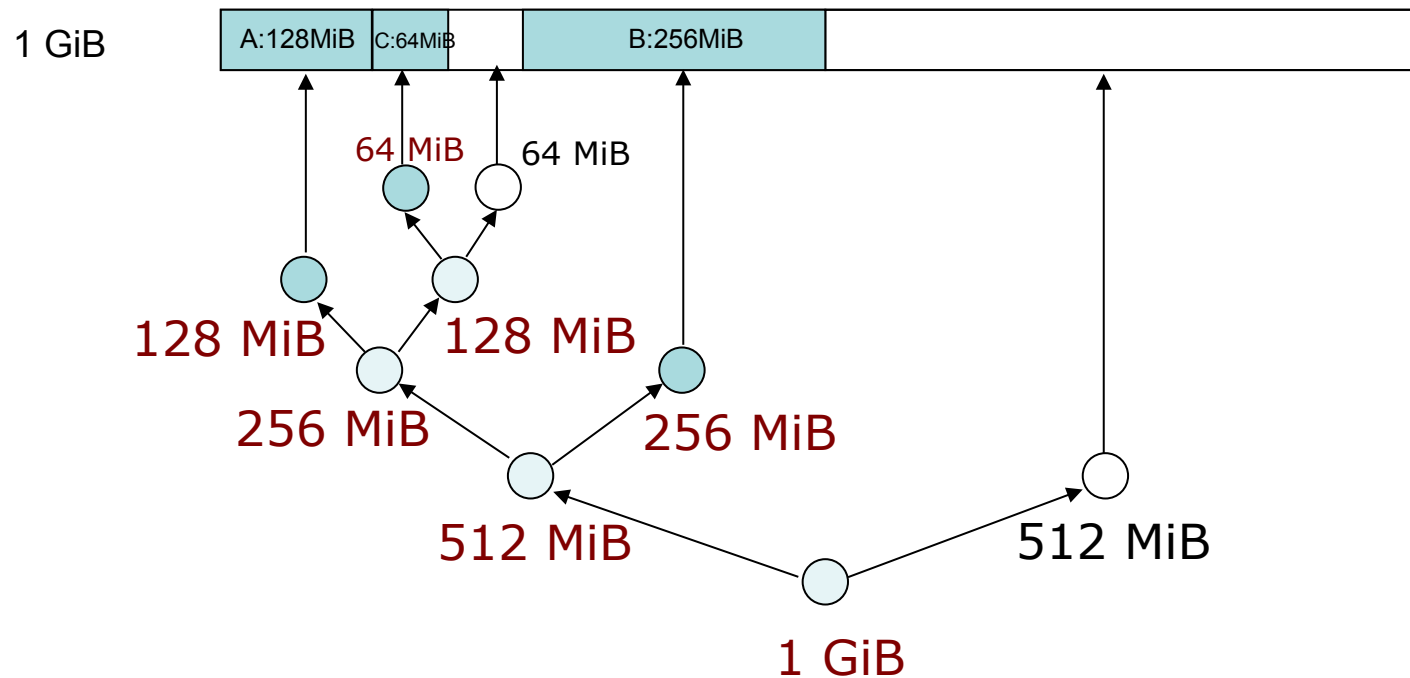


Es folgt Anforderung C: 64 MiB, d.h. Block der Größe 64 MiB

Buddy-System (11)

Nach Anforderung C: 64 MiB, d.h. Block der Größe 64 MiB.

Freie Blöcke:



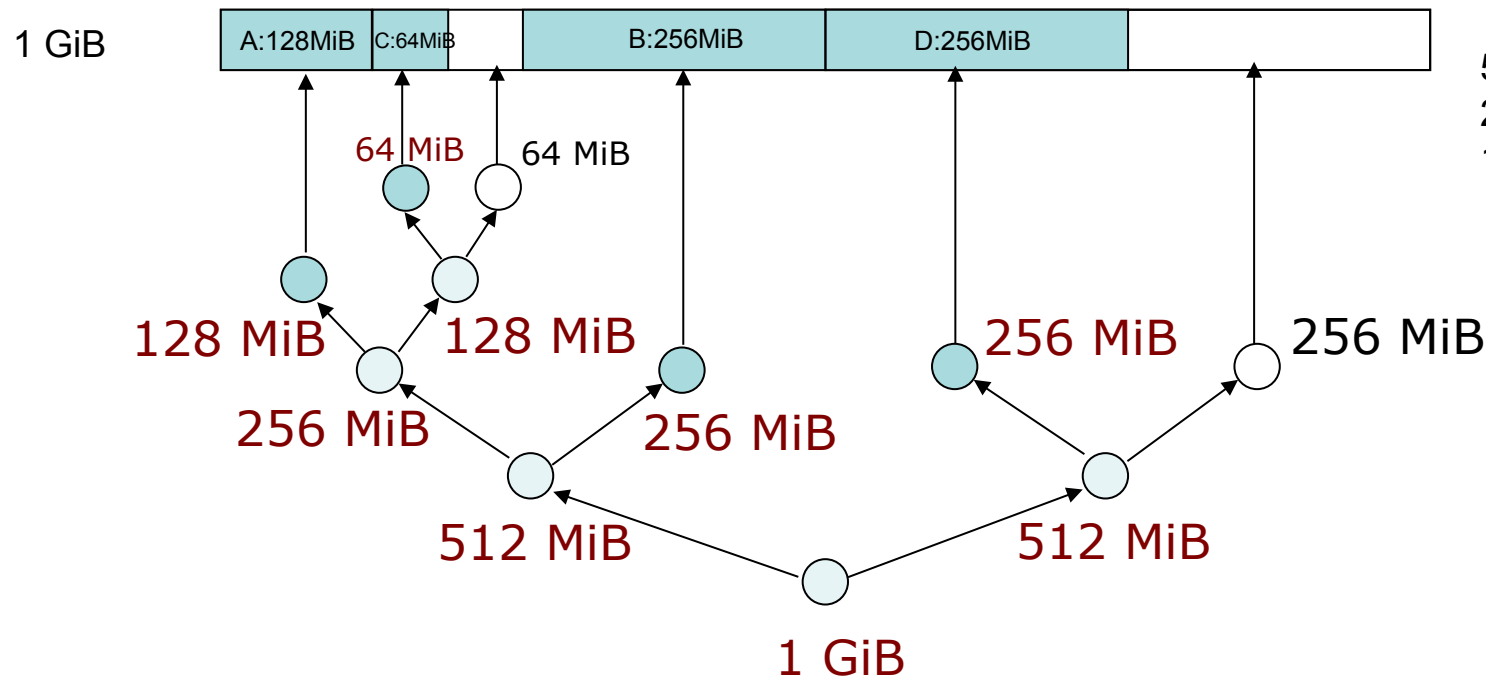
1 GiB: 0
512 MiB: 1
256 MiB: 0
128 MiB: 0
64 MiB: 1

Es folgt Anforderung D: 256 MiB, d.h. Block der Größe 256 MiB

Buddy-System (12)

Nach Anforderung D: 256 MiB, d.h. Block der Größe 256 MiB.

Freie Blöcke:



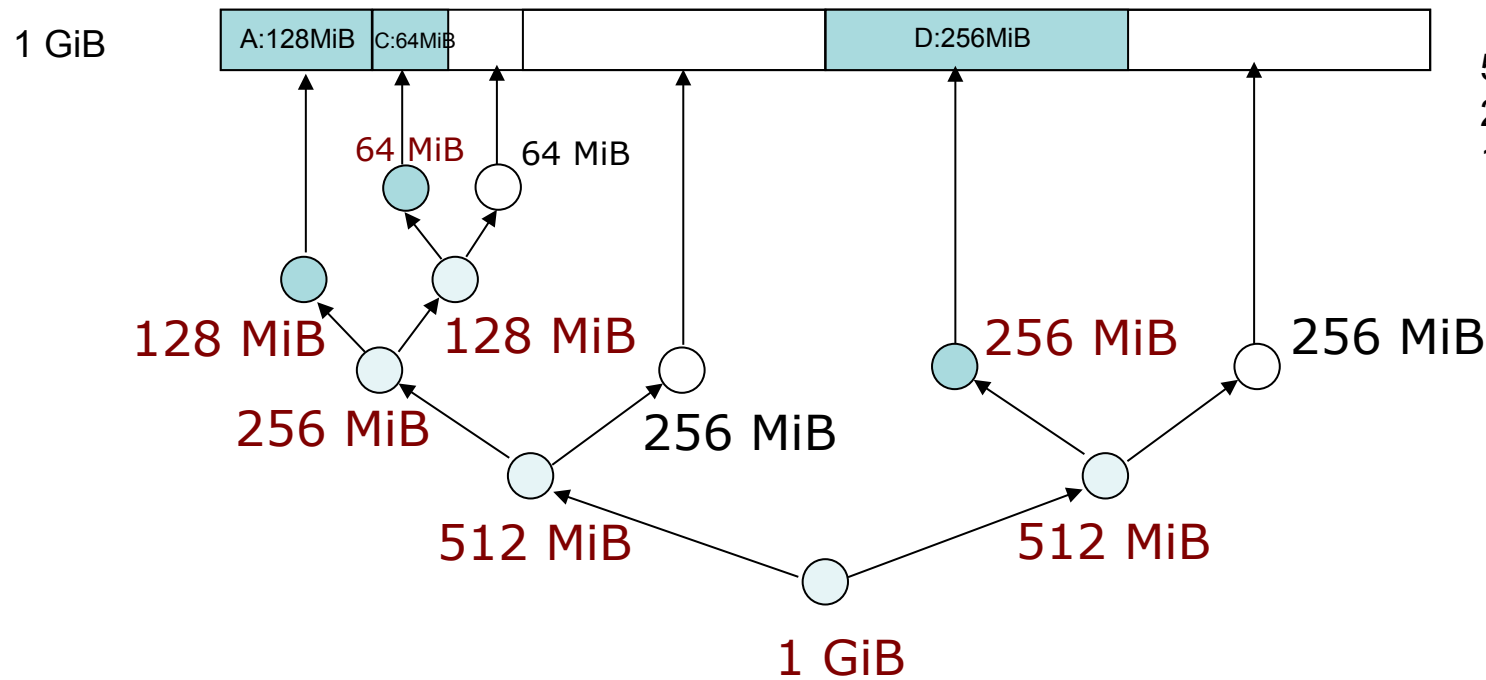
1 GiB: 0
512 MiB: 0
256 MiB: 1
128 MiB: 0
64 MiB: 1

Es folgt Freigabe B

Buddy-System (13)

Nach Freigabe B:

Freie Blöcke:

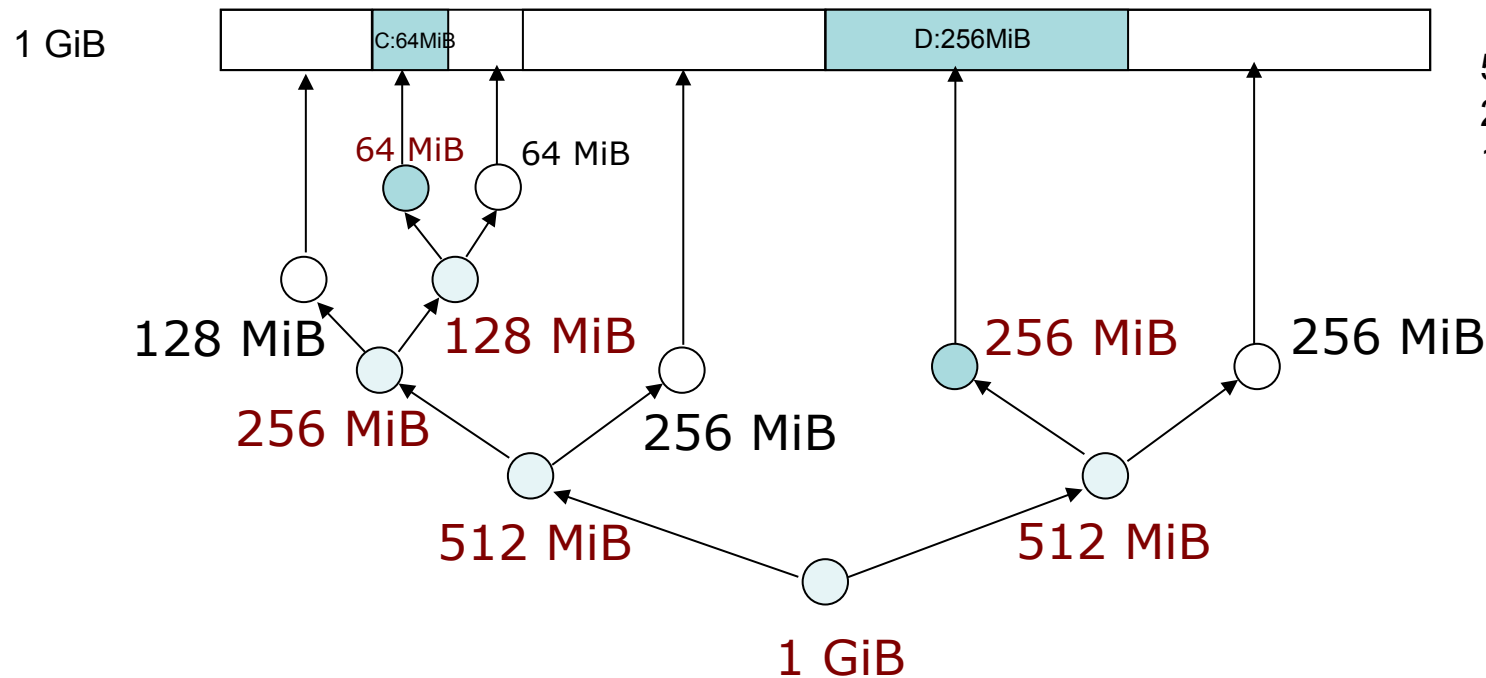


1 GiB: 0
512 MiB: 0
256 MiB: 2
128 MiB: 0
64 MiB: 1

Es folgt Freigabe A

Buddy-System (14)

Nach Freigabe A:



Freie Blöcke:

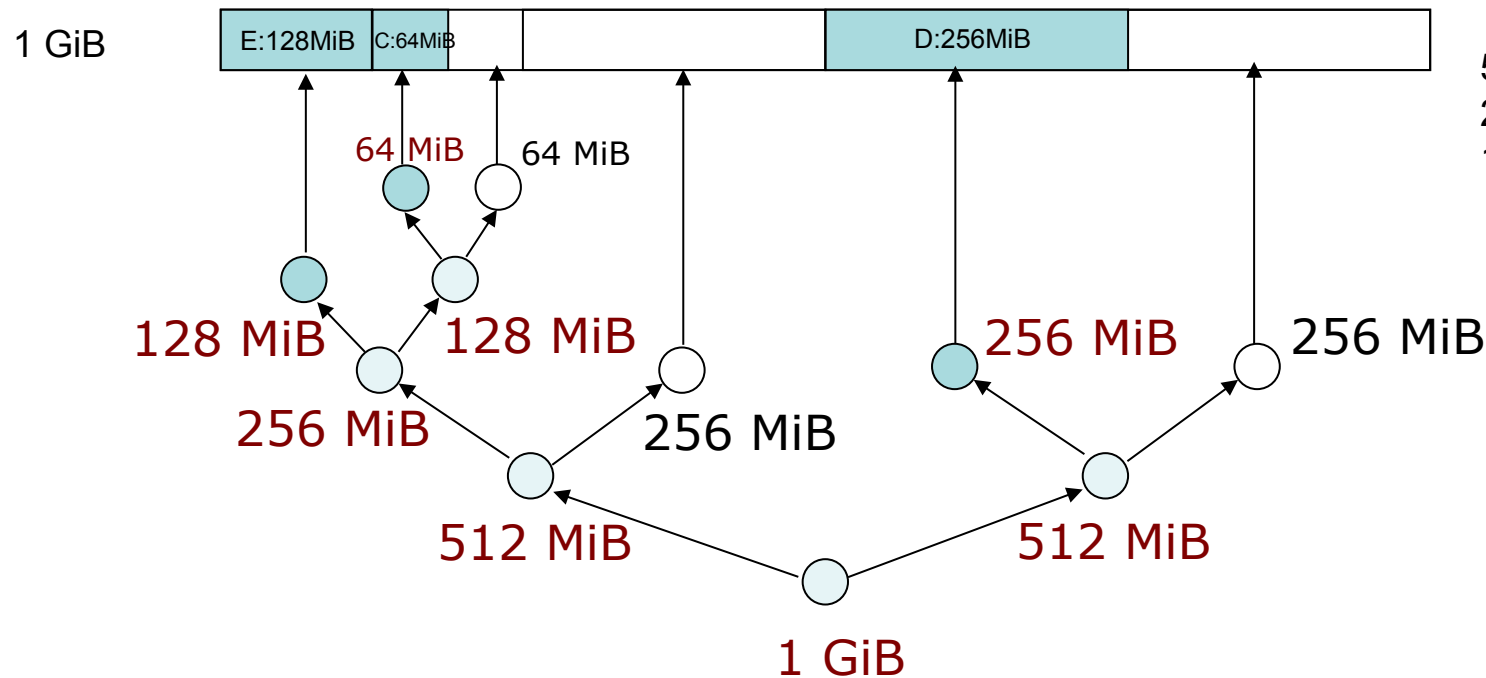
1 GiB: 0
512 MiB: 0
256 MiB: 2
128 MiB: 1
64 MiB: 1

Es folgt Anforderung E: 75 MiB, d.h. Block der Größe 128 MiB

Buddy-System (15)

Nach Anforderung E: 75 MiB, d.h. Block der Größe 128 MiB:

Freie Blöcke:

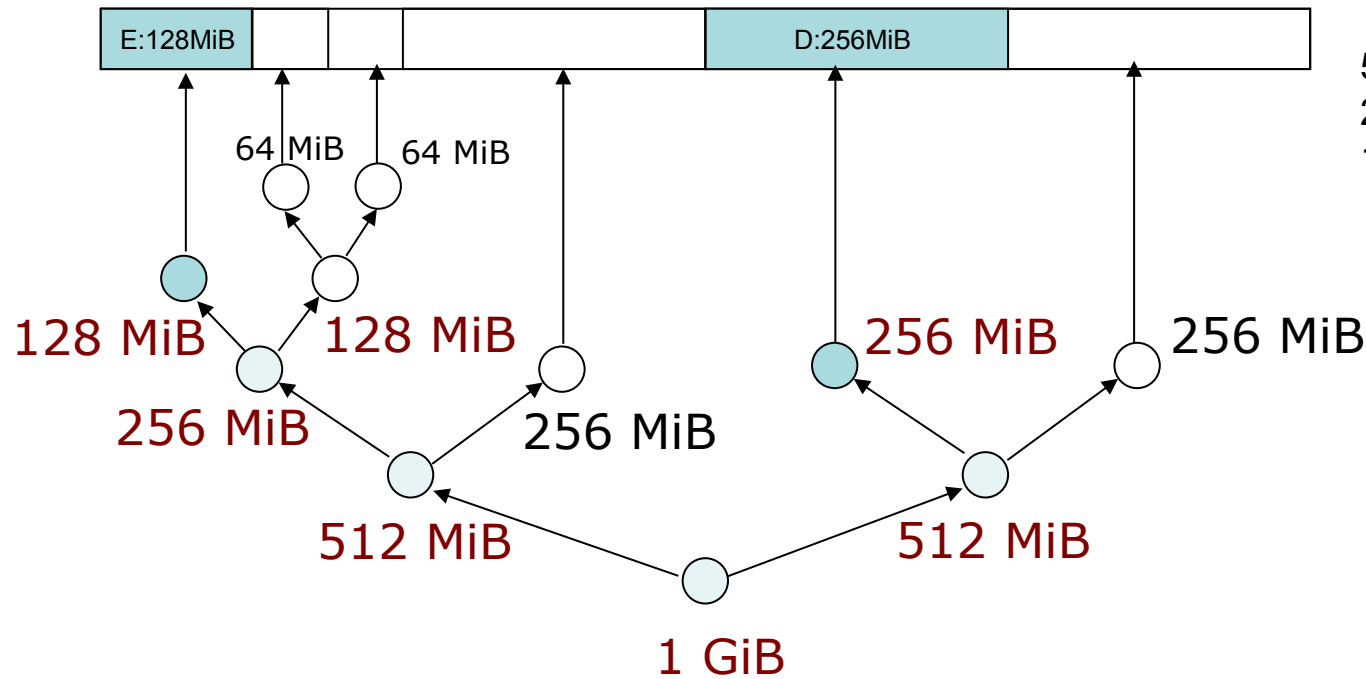


Es folgt Freigabe C

Buddy-System (16)

Bei Freigabe C:

1 GiB



Freie Blöcke:

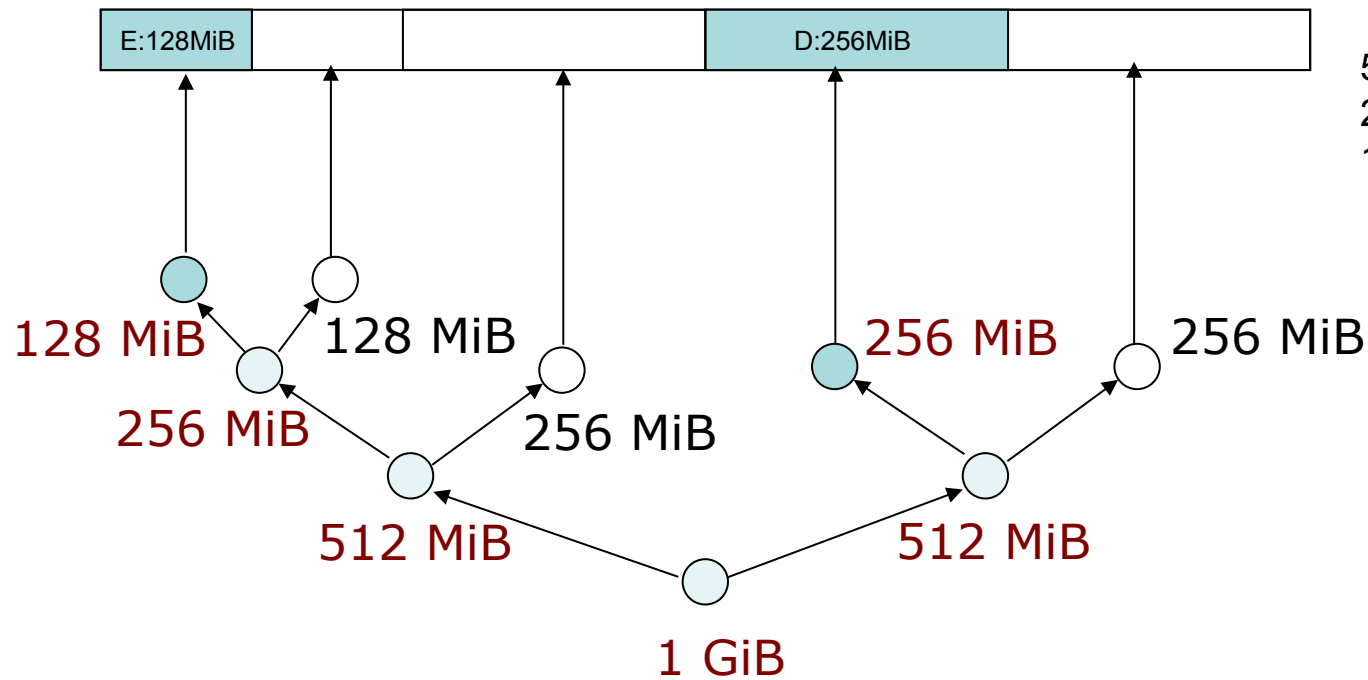
1 GiB: 0
512 MiB: 0
256 MiB: 2
128 MiB: 0
64 MiB: 2

Buddy-System (17)

Nach Freigabe C:

Freie Blöcke:

1 GiB



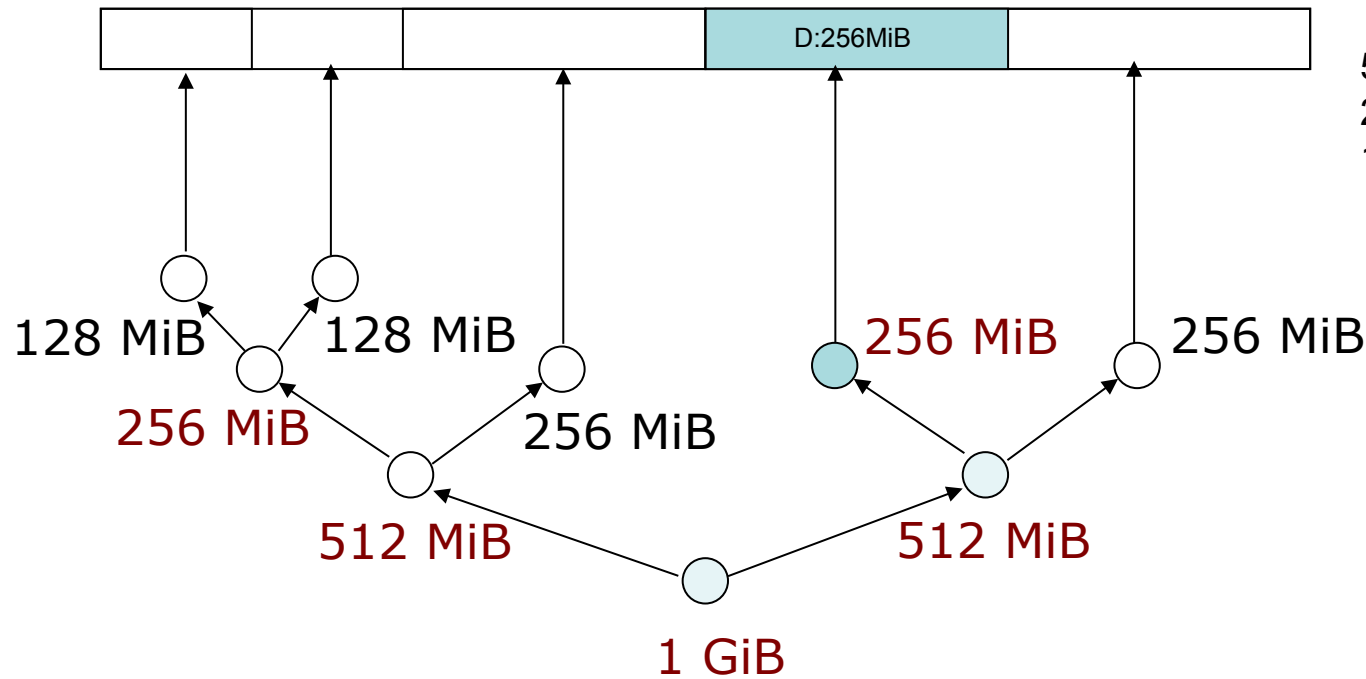
1 GiB: 0
512 MiB: 0
256 MiB: 2
128 MiB: 1
64 MiB: 0

Es folgt Freigabe E

Buddy-System (18)

Bei Freigabe E:

1 GiB

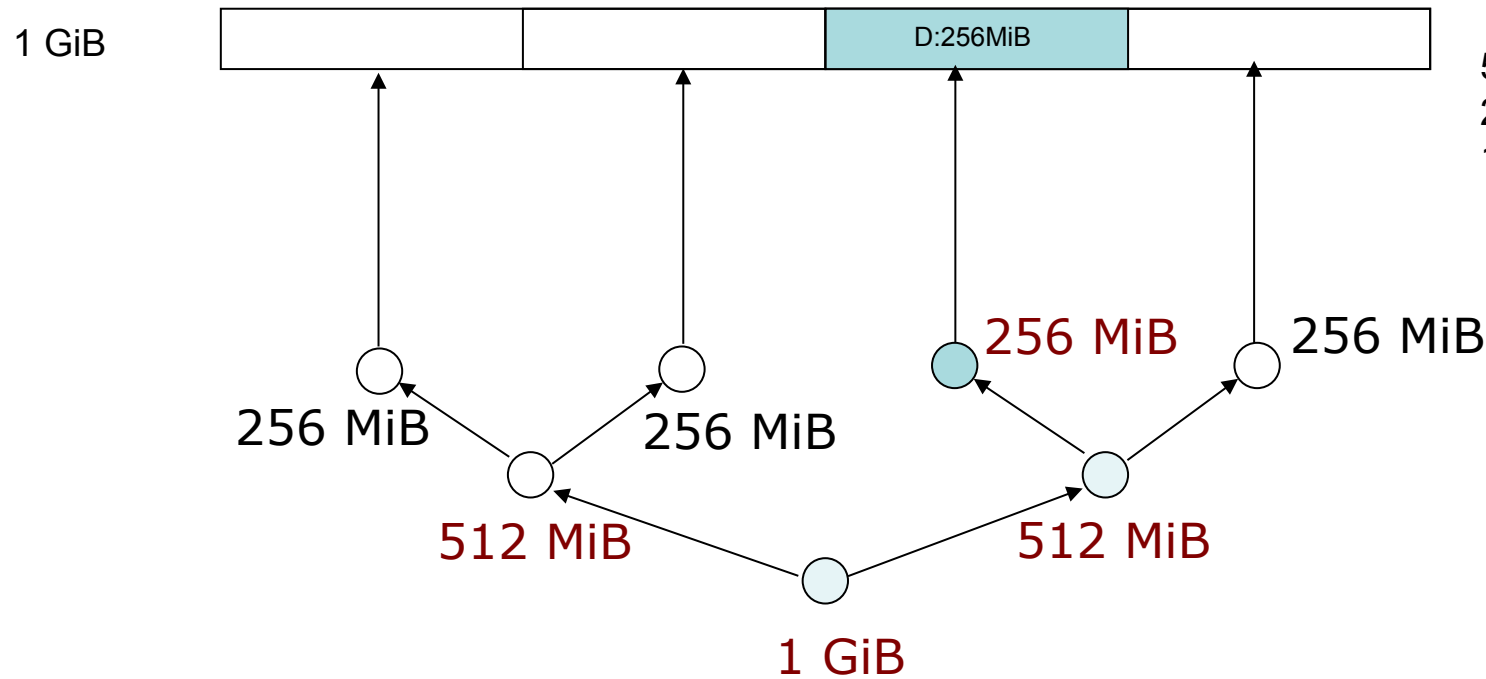


Freie Blöcke:

1 GiB: 0
512 MiB: 0
256 MiB: 2
128 MiB: 2
64 MiB: 0

Buddy-System (19)

Bei Freigabe E:

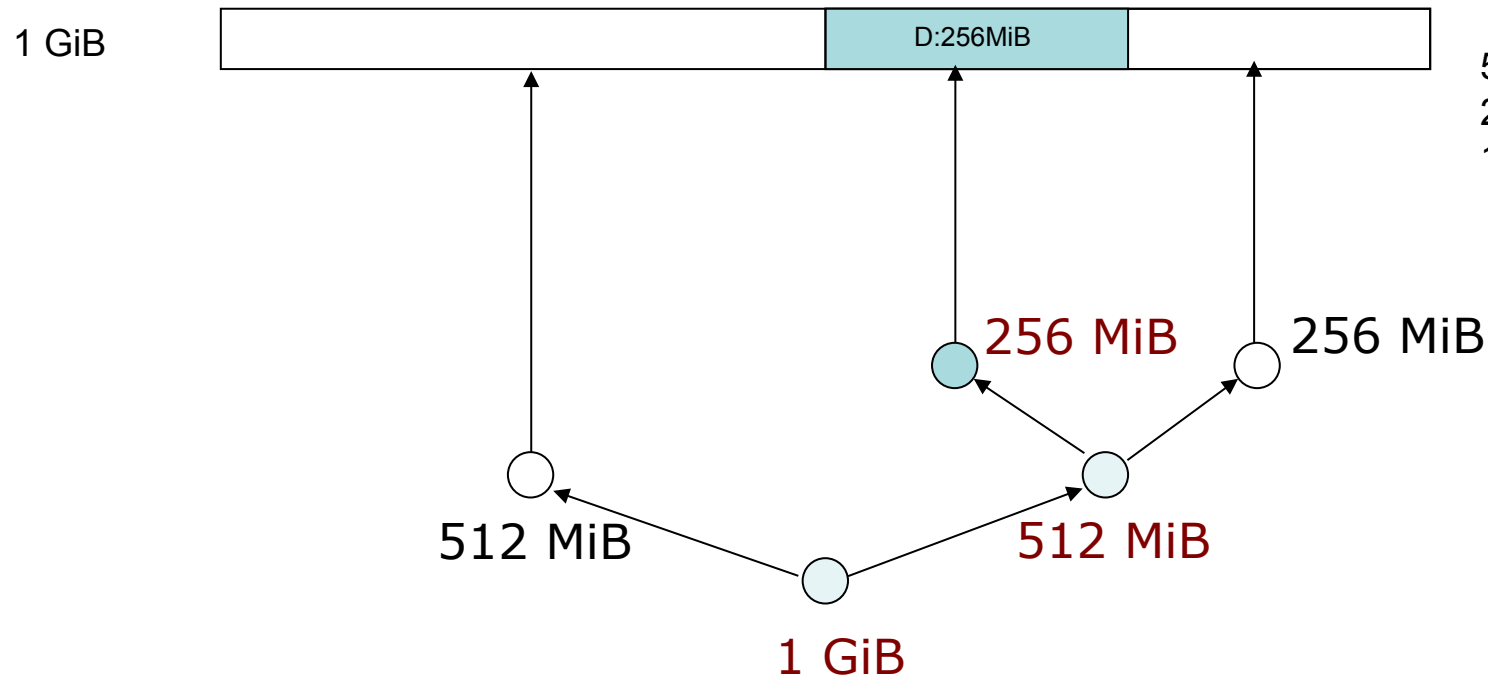


Freie Blöcke:

1 GiB: 0
512 MiB: 0
256 MiB: 3
128 MiB: 0
64 MiB: 0

Buddy-System (20)

Nach Freigabe E:



Freie Blöcke:

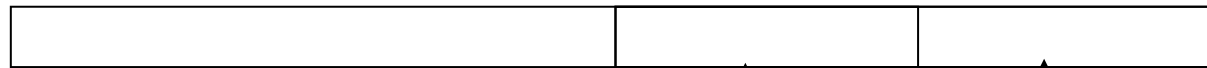
1 GiB: 0
512 MiB: 1
256 MiB: 1
128 MiB: 0
64 MiB: 0

Es folgt Freigabe D

Buddy-System (21)

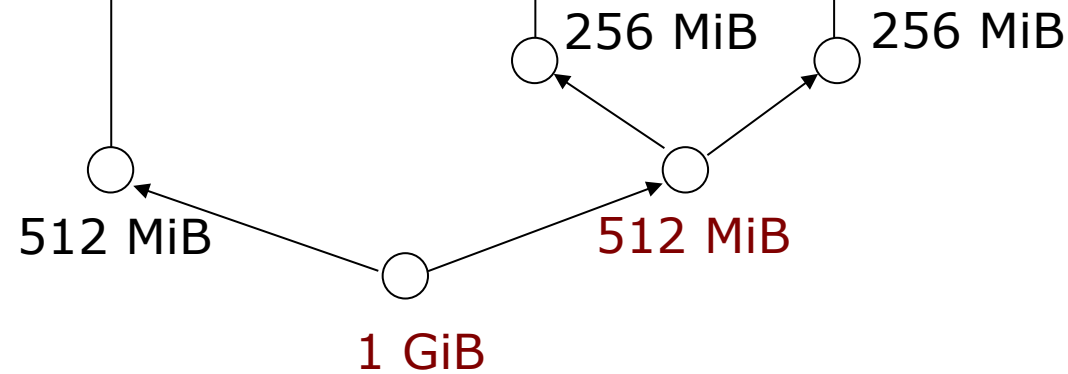
Bei Freigabe D:

1 GiB



Freie Blöcke:

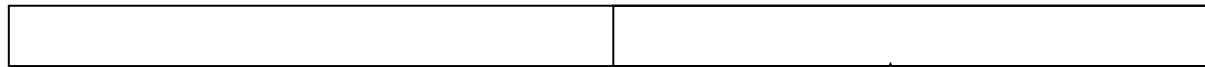
1 GiB: 0
512 MiB: 1
256 MiB: 2
128 MiB: 0
64 MiB: 0



Buddy-System (22)

Bei Freigabe D:

1 GiB



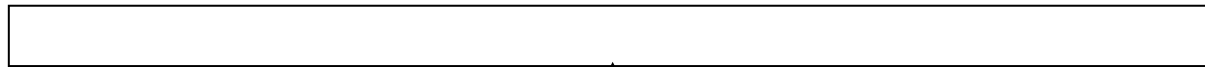
Freie Blöcke:

1 GiB: 0
512 MiB: 2
256 MiB: 0
128 MiB: 0
64 MiB: 0

Buddy-System (23)

Nach Freigabe D:

1 GiB



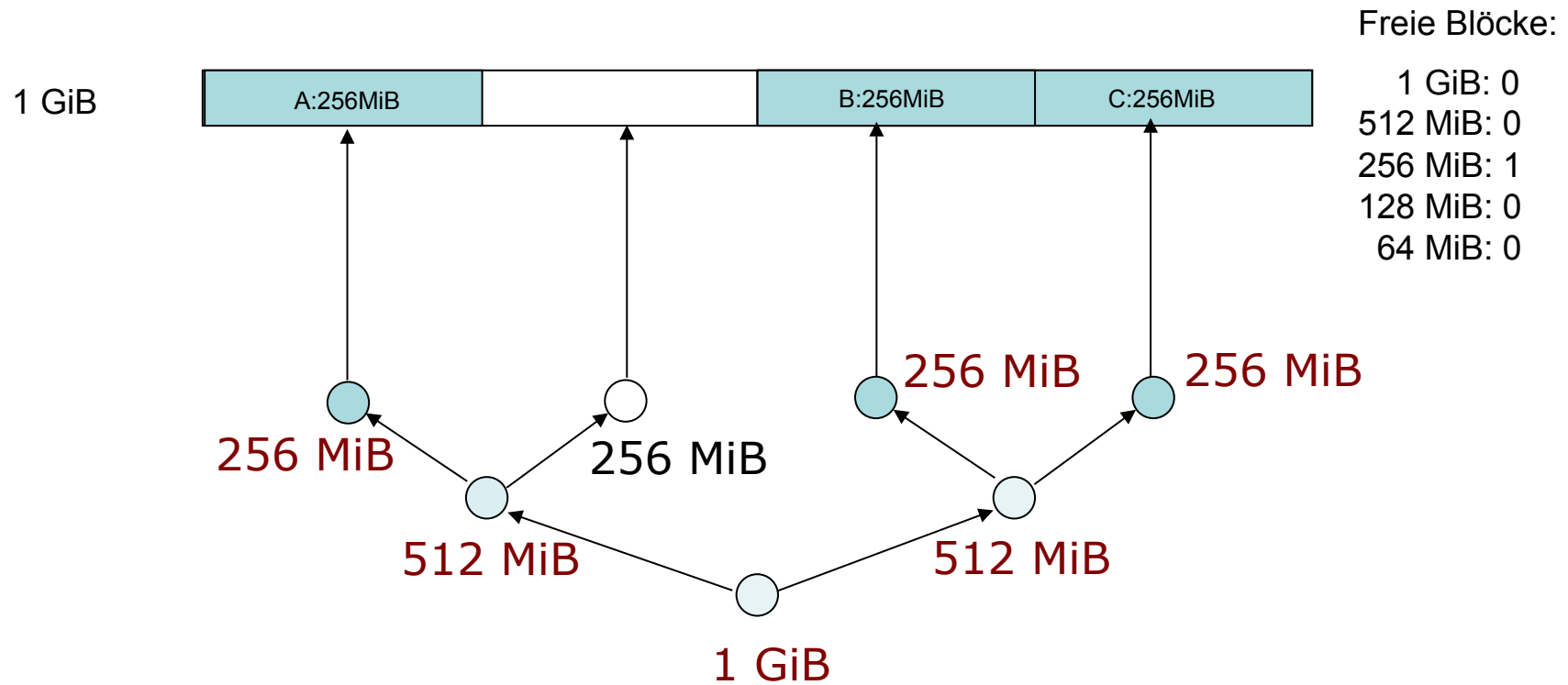
1 GiB

Freie Blöcke:

1 GiB: 1
512 MiB: 0
256 MiB: 0
128 MiB: 0
64 MiB: 0

Gesamter Speicher wieder als 1 Block verfügbar

Buddy-System (24)



Was passiert bei Freigabe von B?

Buddy-System (25)

- Effiziente Suche nach freiem Block
- Dynamische Anzahl nicht-ausgelagerter Prozesse
- Beschränkte interne Fragmentierung:
Kleiner als die halbe Größe des gewählten freien Blockes
- Wenig externe Fragmentierung, schnelle Zusammenfassung von freien Blöcken

Relokation (1)

- Nach Aus- und Wiedereinlagern von Prozess: Speicherbereich ändert sich
- Absolute Sprungbefehle und Datenzugriffsbefehle müssen weiterhin funktionieren

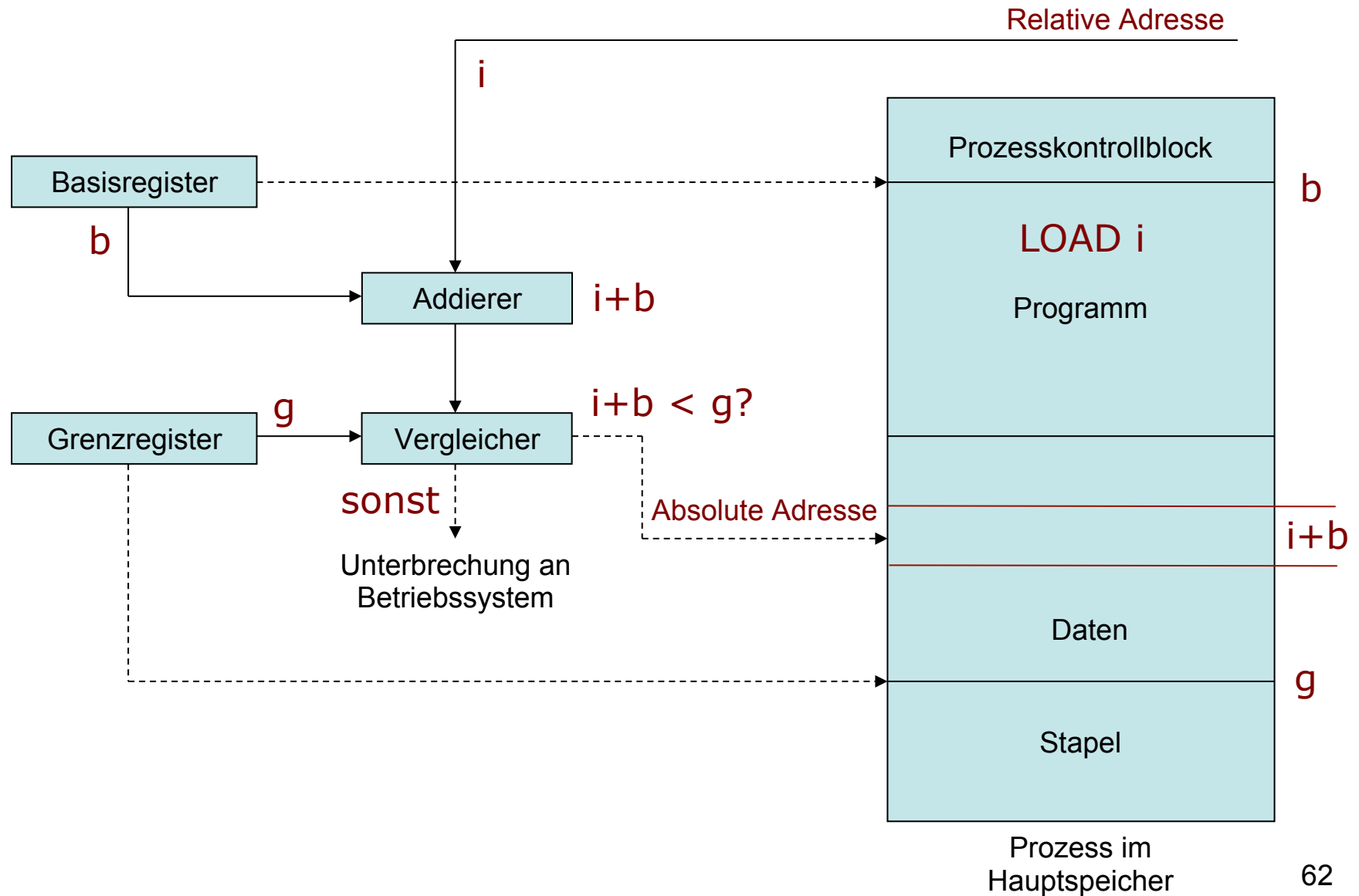
Relokation (2)

- **Physikalische bzw. absolute Adresse:** konkrete Stelle im Hauptspeicher
- **Logische Adresse:** Bezug auf eine Speicherstelle, unabhängig von der aktuellen Zuteilung im Speicher
- **Relative Adresse:**
 - Spezialfall einer logischen Adresse
 - Adresse relativ zu einem bekannten Punkt (in der Regel Programmmanfang)

Relokation (3)

- Dynamisches Laden zur Laufzeit: Berechnung von absoluten Adressen aus relativen Adressen durch Hardware
- Beim Einlagern: Adresse des Programm-anfangs wird im **Basisregister** gespeichert
- **Grenzregister**: Enthält höchste erlaubte Speicheradresse (Ende des Programms)

Relokation (4)



Relokation (5)

- Absolute Adresse: Relative Adresse wird um den Wert erhöht, der sich im Basisregister befindet
- Resultierende Adresse wird mit dem Wert im Grenzregister verglichen
- Befehlsausführung nur, wenn die Adresse innerhalb der Grenzen liegt, sonst Interrupt

Einfaches Paging (1)

- Zuerst wie bisher: Prozesse sind entweder ganz im Speicher oder komplett ausgelagert
- Prozessen werden Speicherbereiche zugeordnet, die **nicht** notwendigerweise zusammenhängend sind
- Hauptspeicher aufgeteilt in viele **gleichgroße Seitenrahmen**
- Prozesse aufgeteilt in **Seiten derselben Größe**

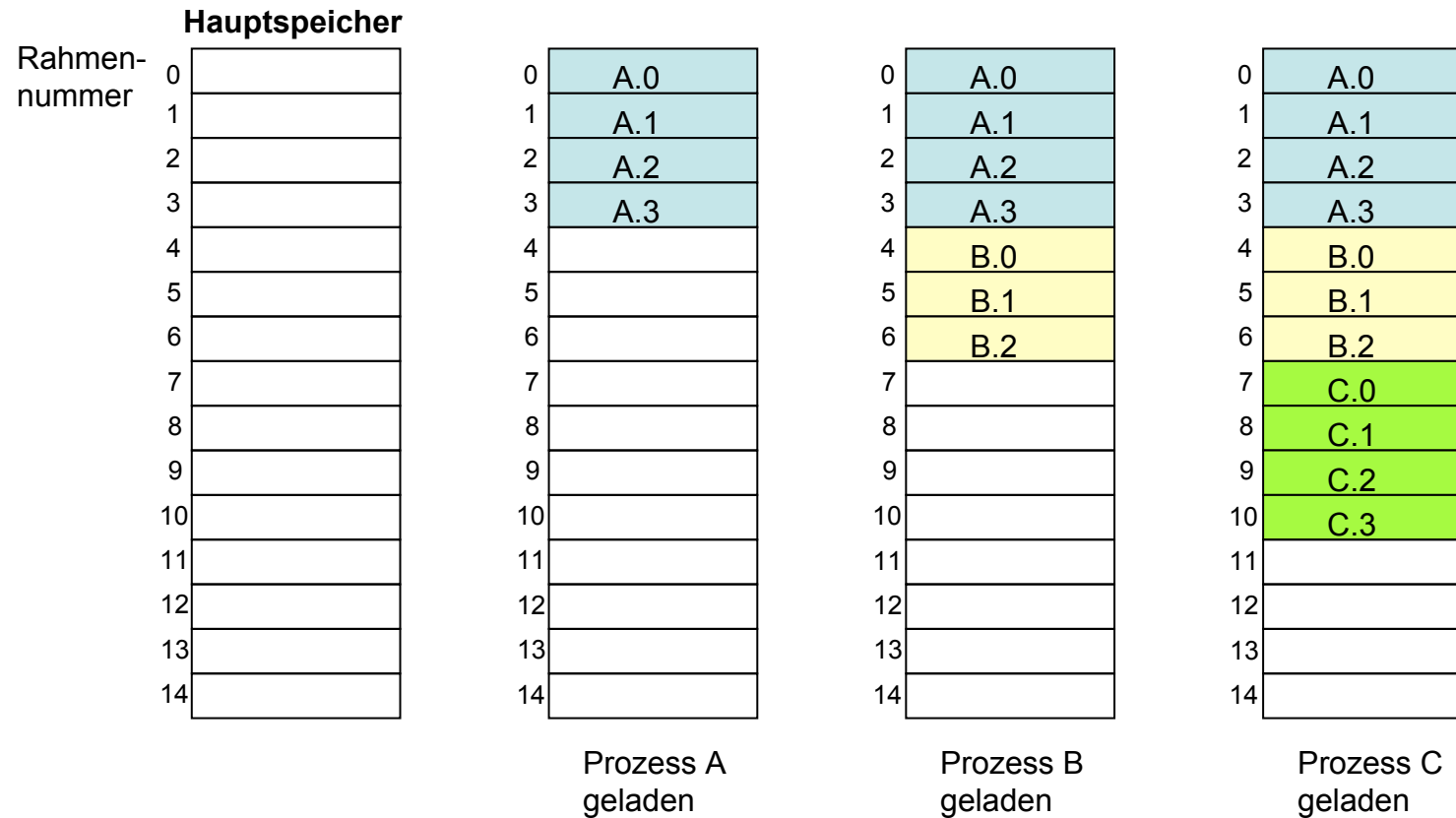
Einfaches Paging (2)

- BS verwaltet **Seitentabelle** für jeden Prozess
- Zuordnung von Seiten zu Seitenrahmen bei der Ausführung von Prozessen
- Innerhalb Programm: Logische Adresse der Form „**Seitennummer, Offset**“
- Durch Seitentabelle: Übersetzung der logischen Adressen in physikalische Adressen „**Rahmennummer, Offset**“

Einfaches Paging (3)

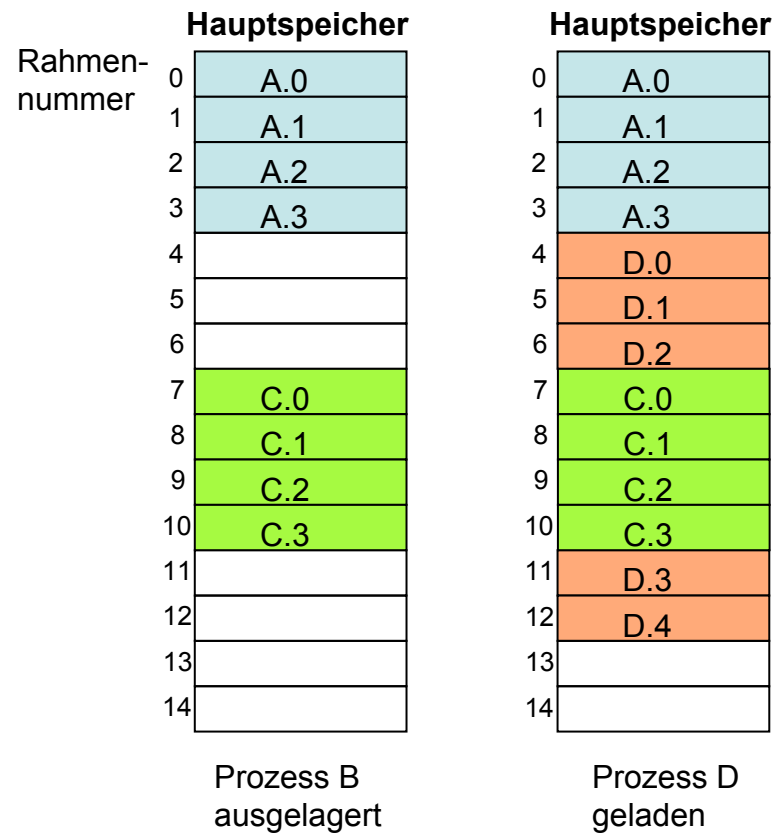
- Prozessorhardware übersetzt logische Adresse in physikalische Adresse
- Interne Fragmentierung nur bei letzter Seite eines Prozesses
- Keine externe Fragmentierung

Einfaches Paging (4)

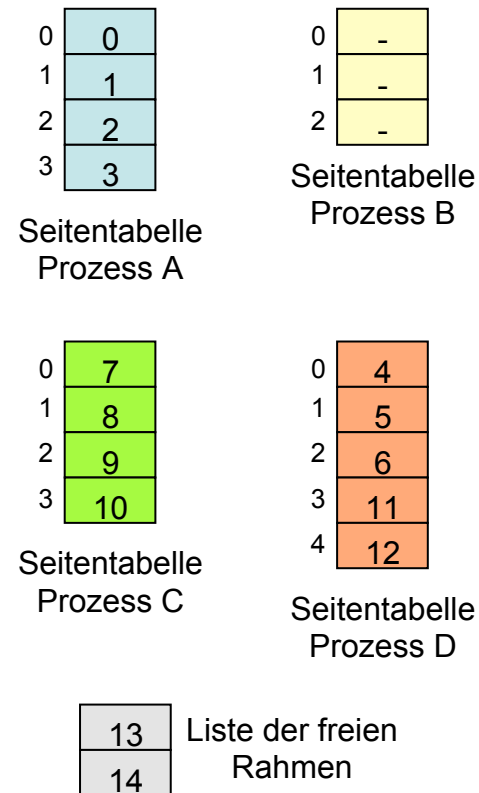


Prozess D
mit 5 Seiten
soll jetzt
geladen
werden!

Einfaches Paging (5)



Datenstrukturen zum aktuellen Zeitpunkt:



Einfaches Paging (6)

- Einfaches Paging ähnlich zum Konzept des statischen Partitionierens
- Aber: Beim Paging sind die Partitionen relativ klein
- Programm kann mehrere Partitionen/Rahmen belegen, die nicht aneinander angrenzen

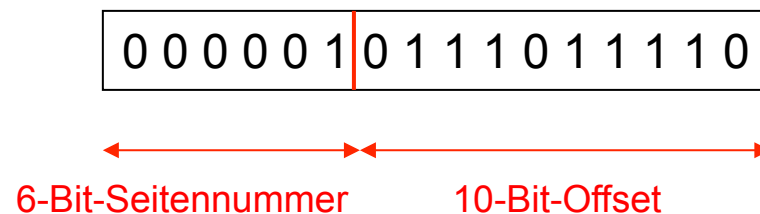
Einfaches Paging (7)

Berechnung von physikalischen Adressen aus logischen Adressen:

- Seitengröße (und Rahmengröße) ist eine Zweierpotenz
- Logische Adresse im Programm besteht aus Seitennummer und Offset
- Physikalische Adresse besteht aus Rahmennummer und Offset

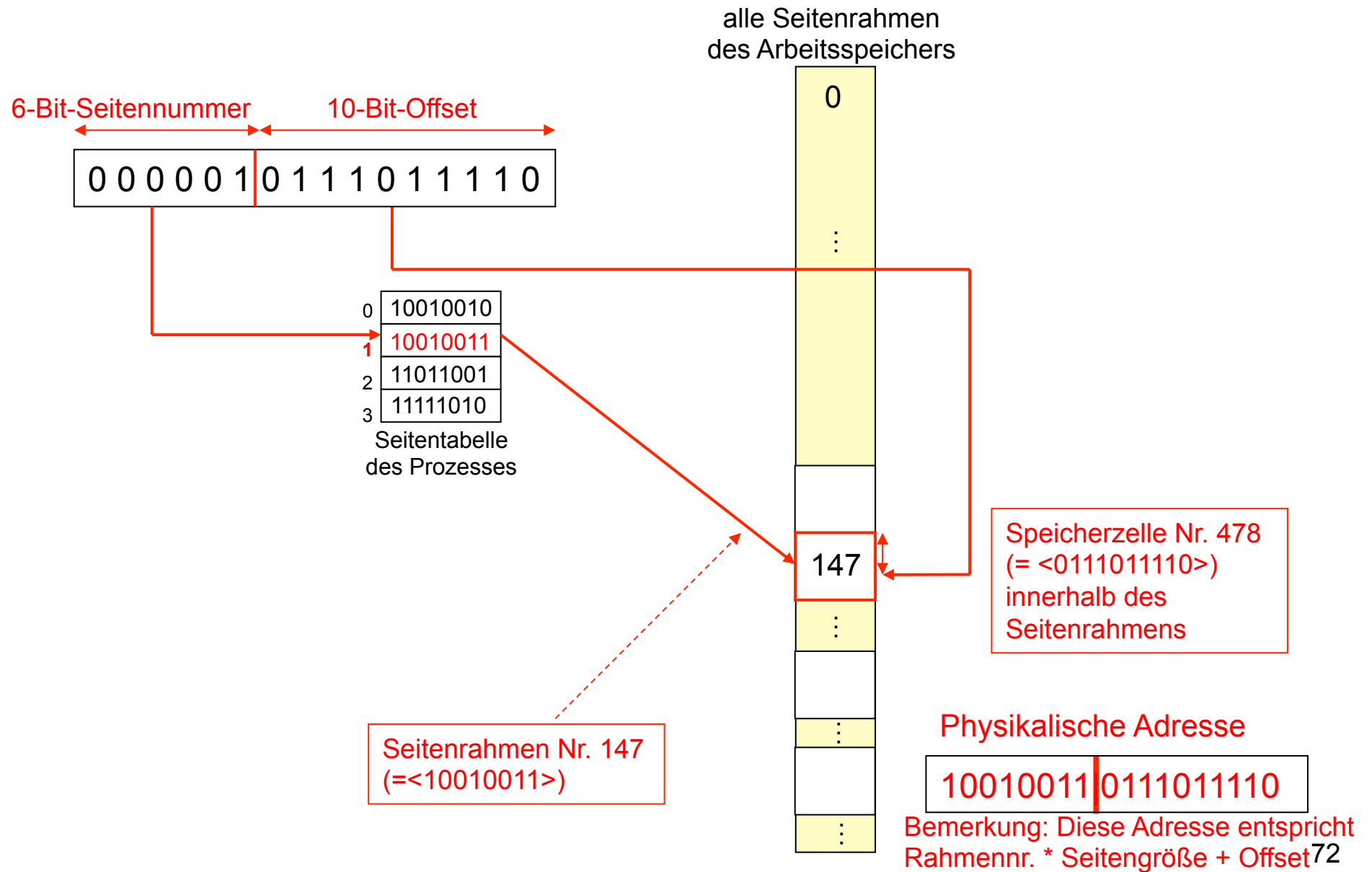
Einfaches Paging (8)

- Beispiel: Logische Adresse der Länge 16 Bit
- Seitengröße 1 KiB = $2^{10} = 1024$ Bytes
- Offset-Feld von 10 Bit wird benötigt, um alle Bytes referenzieren zu können



- Der Prozess kann bis zu $2^6=64$ verschiedene Seiten haben, die über die Seitentabelle des Prozesses auf Seitenrahmen im Hauptspeicher abgebildet werden

Einfaches Paging (9)



Einfaches Paging (10)

- Hauptspeicher wird in viele kleine Rahmen gleicher Größe unterteilt
- Jeder Prozess wird in Seiten geteilt, deren Größe der der Rahmen entspricht
- Seitentabelle enthält Zuordnung von Prozessseiten an Seitenrahmen des Speichers
- Interne Fragmentierung nur bei letzter Seite eines Prozesses

Einfaches Paging (11)

Entfernen eines Prozesses aus dem Speicher:

- Seitentabelle enthält Information, welche Seitenrahmen dem Prozess gehören
- Füge diese Rahmen zur Liste der freien Rahmen hinzu
- Keine zusätzlichen Datenstrukturen des Betriebssystems benötigt

Paging mit virtuellem Speicher (1)

Grundidee:

- Lagere **Teile** von Prozessen ein- bzw. aus anstelle kompletter Prozesse
- Programm kann auch weiter ausgeführt werden, auch wenn **nur die aktuell benötigten** Informationen (Code und Daten) im Speicher sind
- Bei Zugriff auf Informationen, die ausgelagert sind: Nachladen von Seiten

Paging mit virtuellem Speicher (2)

- Hauptspeicher = realer Speicher
- Hauptspeicher + Hintergrundspeicher =
virtueller Speicher
- Vorteile:
 - Platz für mehr bereite Prozesse
 - Tatsächlicher Speicherplatzbedarf eines Prozesses muss nicht im Voraus feststehen
 - Adressraum eines Prozesses kann größer sein als verfügbarer Hauptspeicher

Paging mit virtuellem Speicher (3)

Nachteile:

- Nachladen von Seiten
- Notwendiges Auslagern von anderen Seiten
- System wird langsamer

Lokalität (1)

Effizienz von virtuellem Speicher

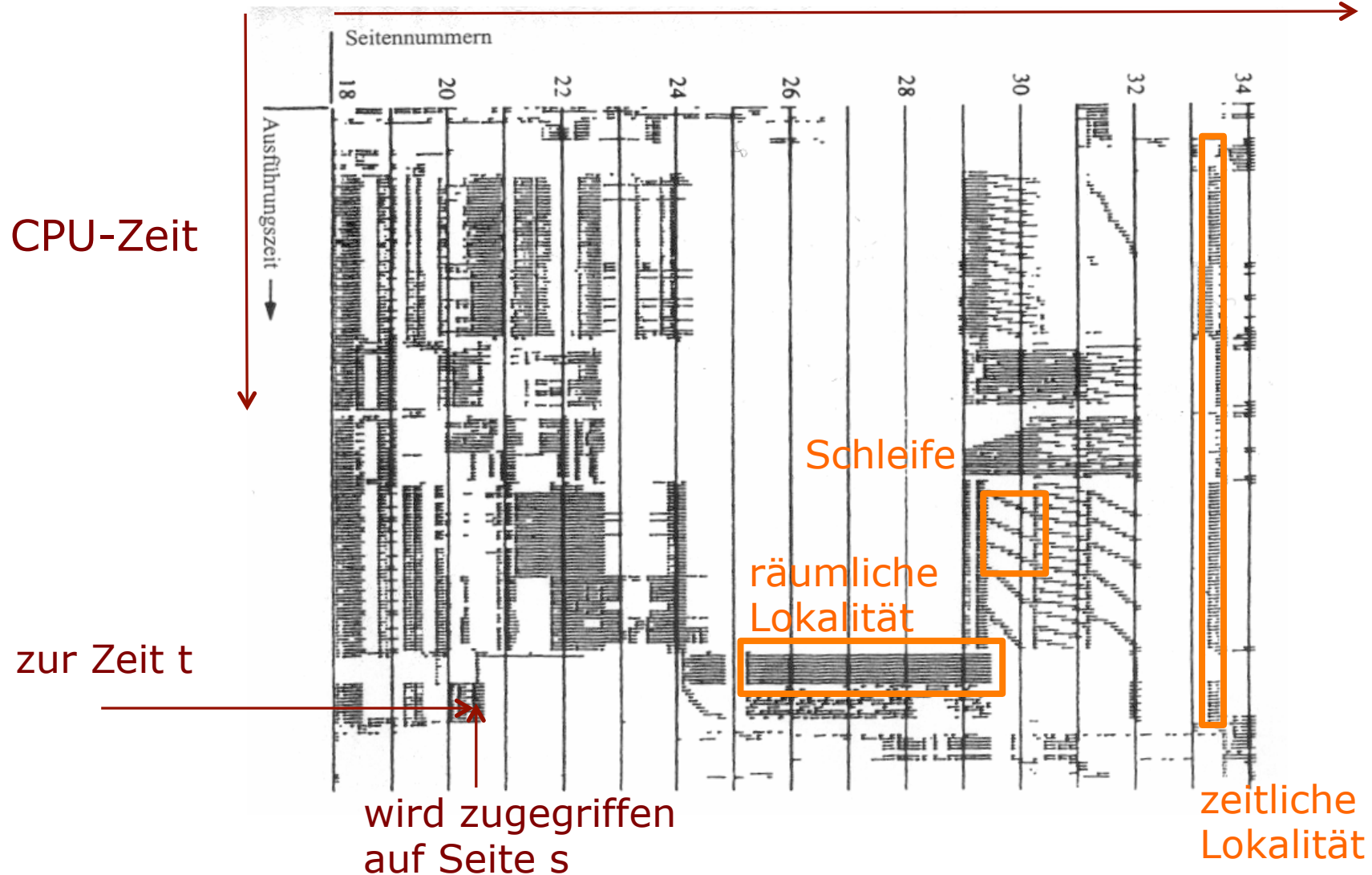
- Typischerweise räumliche und zeitliche Lokalität von Programmen
- **Zeitliche Lokalität:** Nach Zugriff auf eine Speicherzelle ist die Wahrscheinlichkeit hoch, dass in naher Zukunft noch einmal darauf zugegriffen wird
- **Räumliche Lokalität:** Nach Zugriff auf eine bestimmte Speicherzelle gehen die Zugriffe in naher Zukunft auf Speicheradressen in der Nähe

Lokalität (2)

- Die Abarbeitung während kürzerer Zeit bewegt sich häufig in engen Adressbereichen
- Zeitliche Lokalität:
 - Abarbeitung von **Schleifen**
 - In zeitlich engem Abstand Zugriff auf **gleiche Daten**
- Räumliche Lokalität:
 - Sequentieller Abarbeitung von Programmen: Zugriffe auf **benachbarte** Daten
 - Lage von zusammenhängenden Daten

Lokalität (3)

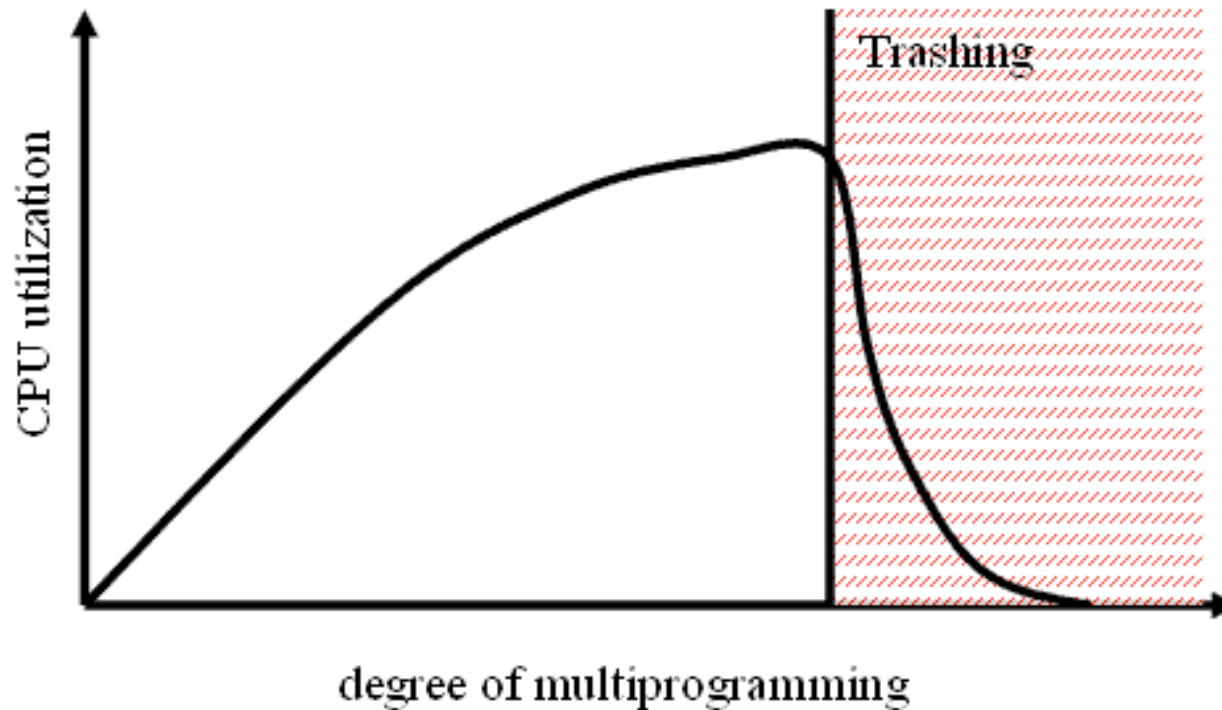
zunehmende Seitennummern eines Prozesses



Lokalität (4)

- Paging mit virtuellem Speicher ist nur dann effizient, wenn Lokalität gegeben ist
- Ansonsten Gefahr von „Thrashing“:
 - Ständiges Aus- und Einlagern von Seiten zwischen Hauptspeicher und Festplatte
 - Der Prozessor ist mehr mit Ein- und Auslagern anstatt Ausführen von Befehlen beschäftigt

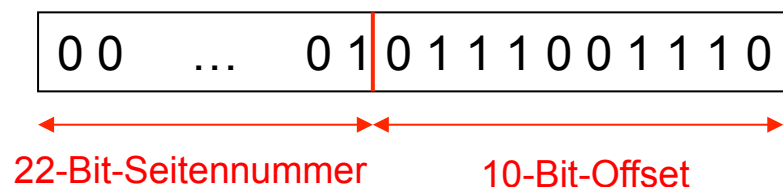
Thrashing



- Um Thrashing zu vermeiden, versucht das BS zu vorherzusagen, welche Seiten in naher Zukunft nicht benötigt werden

Technische Realisierung (1)

- Einfachstes Modell: Prozess (Daten+Code) befindet sich im Hintergrundspeicher
- Bei teilweise eingelagerten Prozessen: Zusätzlich Teile im Hauptspeicher
- Logische Adressen überdecken **kompletten virtuellen Adressraum**
- Wie bei einfachem Paging: Trennung der logischen Adressen in Seitennummer und Offset

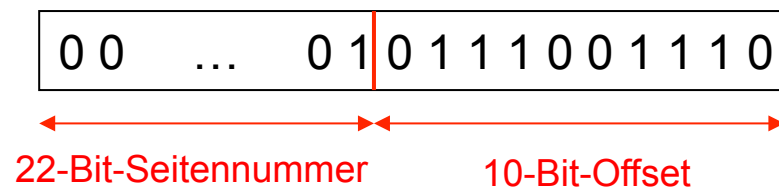


Bsp.: 32-Bit-Adresse
1 KiB Seitengröße

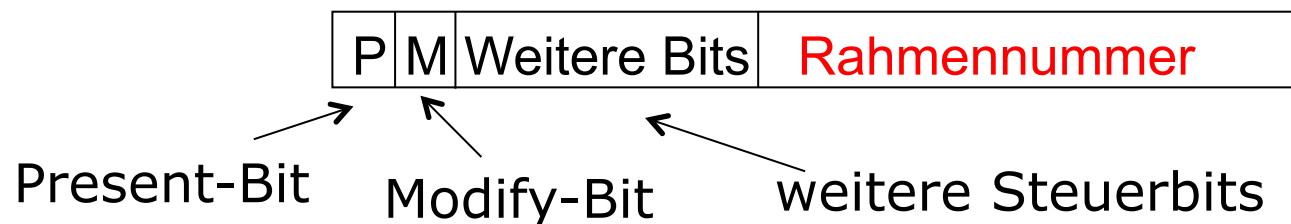
Technische Realisierung (2)

- Zusätzliche Informationen in Seitentabelle:
 - Ist die Seite im Hauptspeicher präsent?
 - Wurde der Inhalt der Seite seit letztem Laden in den Hauptspeicher verändert?

- Logische Adresse:



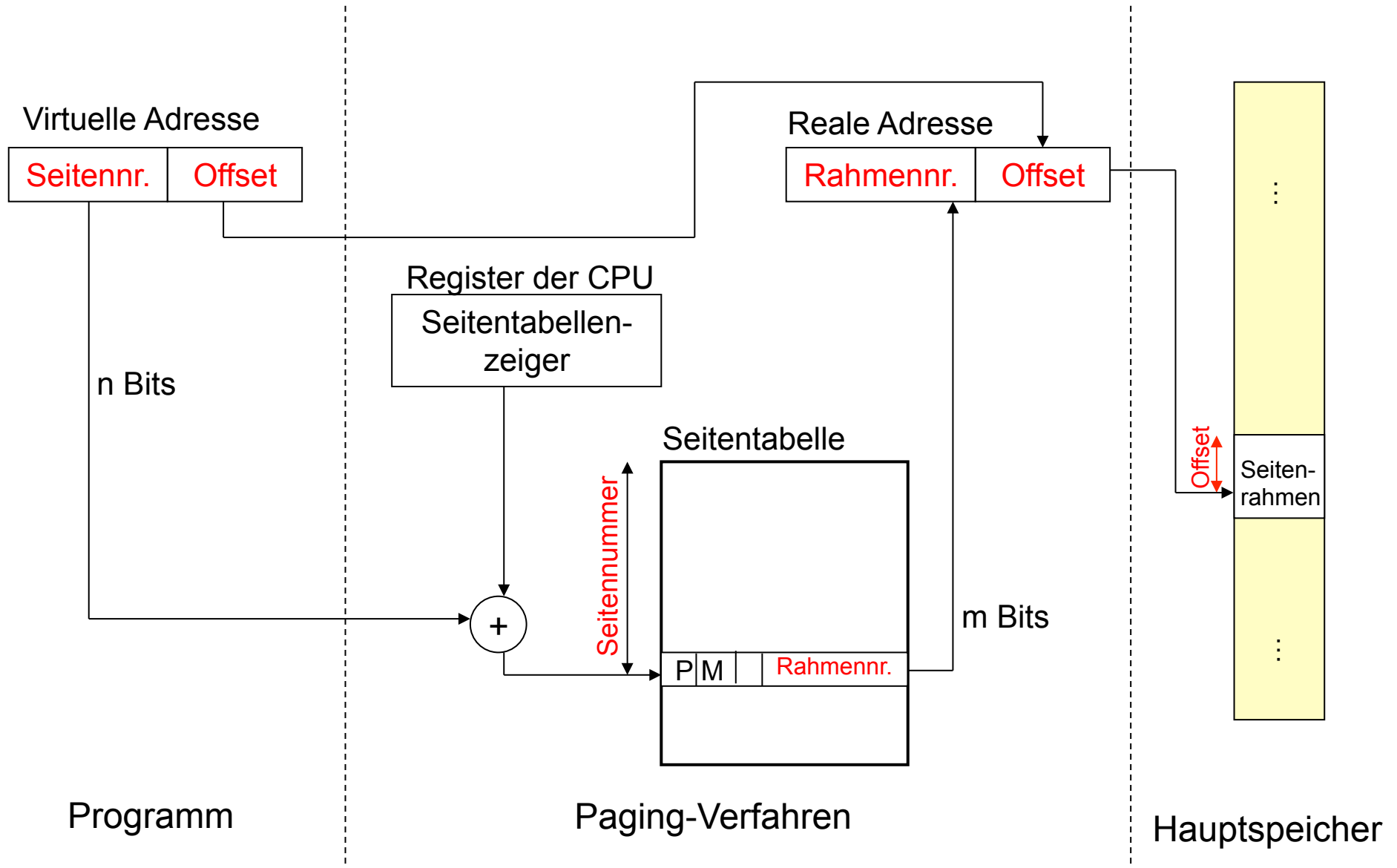
- Seitentabelleneintrag:



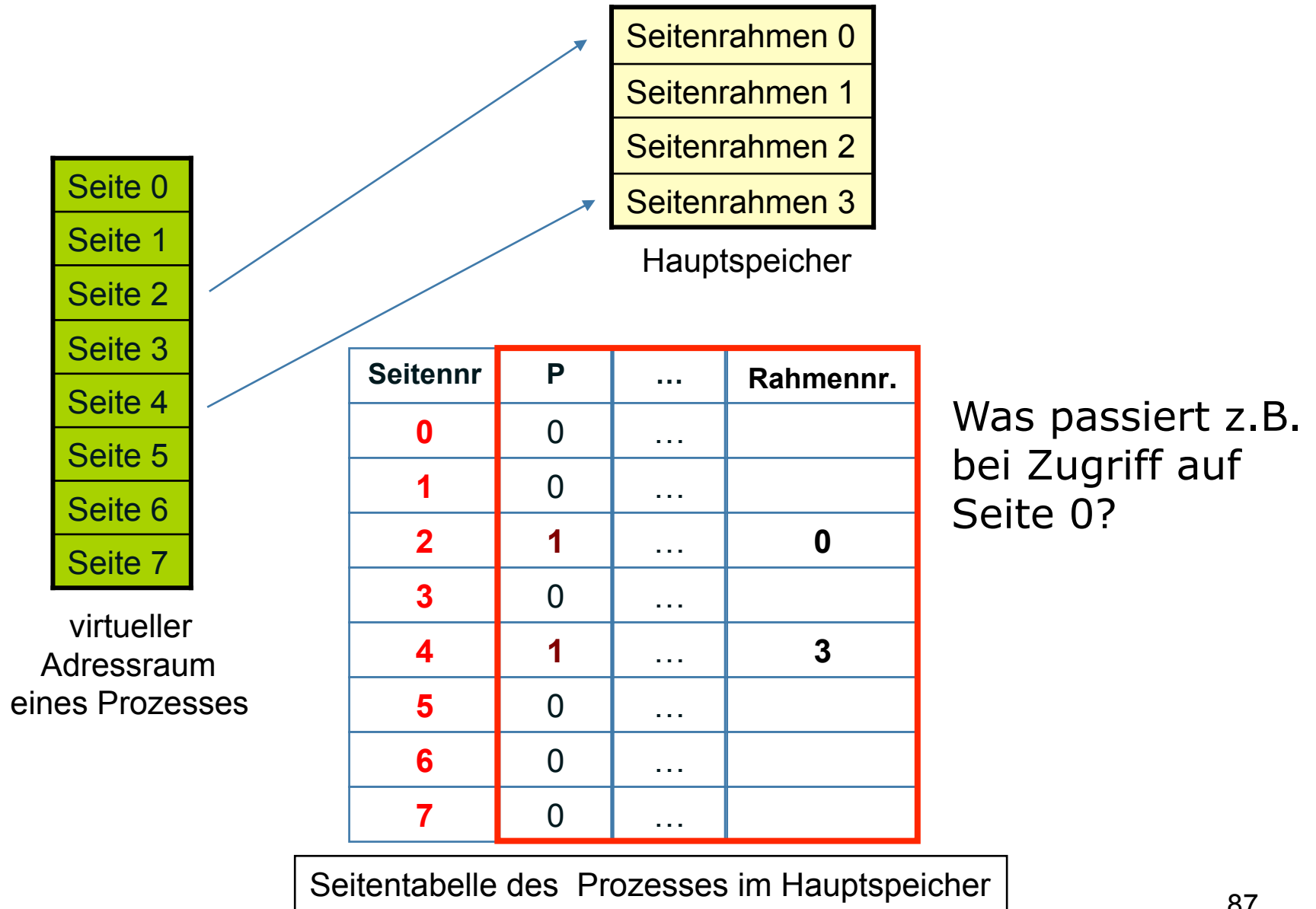
Technische Realisierung (3)

- Seitentabelle liegt im Hauptspeicher
- Umsetzung der virtuellen Adressen in reale Adressen mit Hardwareunterstützung
- Memory Management Unit (MMU) des Prozessors führt Berechnung durch

Adressumsetzung



Seitentabelle



Seitenfehler (1)

- Bei Zugriff auf Seite, die nicht im Hauptspeicher vorhanden
- Hardware (MMU) hat durch Present-Bit die Information, dass angefragte Seite nicht im Hauptspeicher ist
- Laufendes Programm wird unterbrochen, aktueller Programmzustand gesichert

Seitenfehler (2)

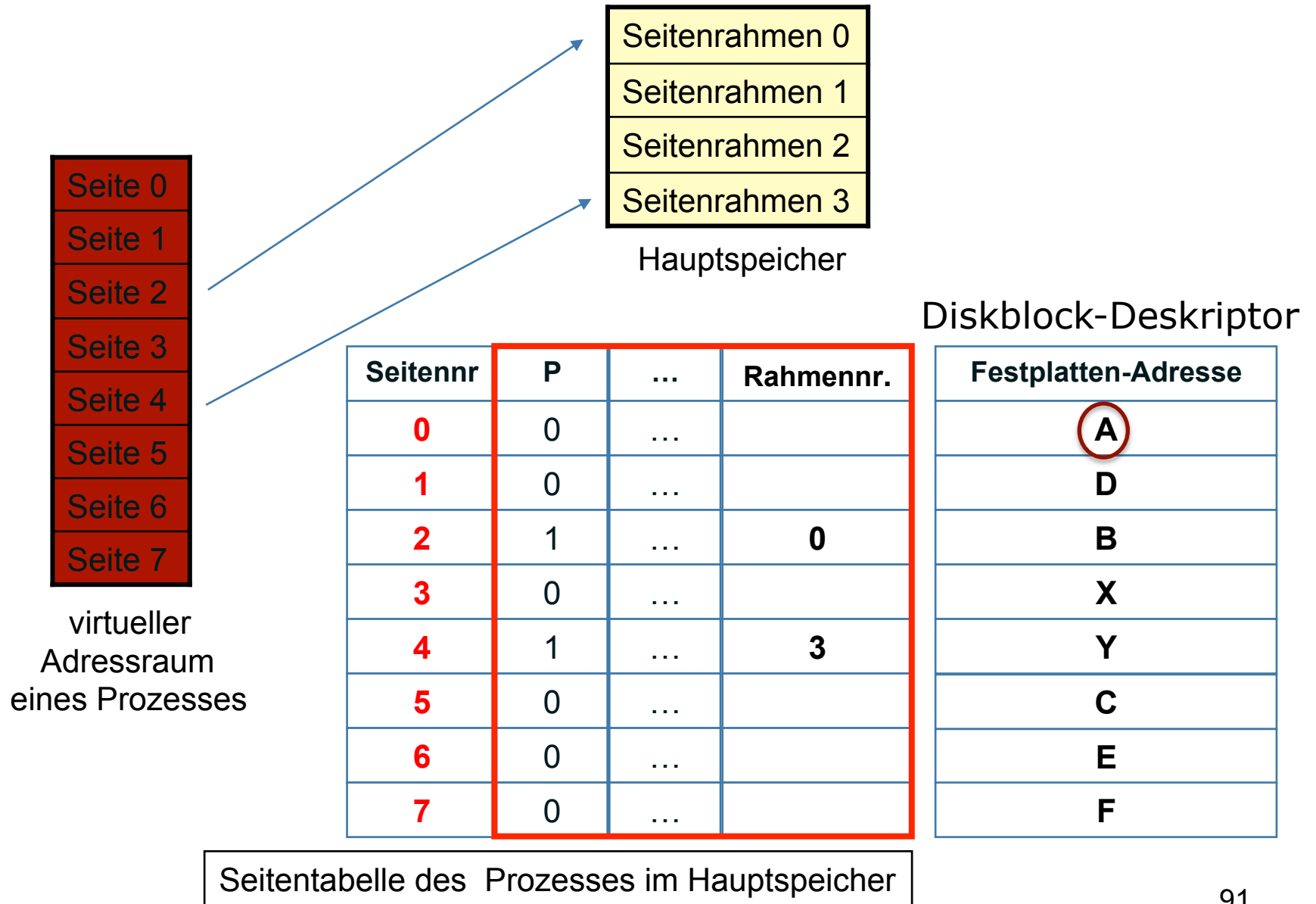
- Betriebssystem lädt die entsprechende Seite von der Festplatte in einen freien Rahmen
- Falls kein Rahmen frei: Vorheriges Verdrängen der Daten eines belegten Rahmens (beachte dabei Modify-Bit)
- Aktualisierung der Seitentabelleneinträge (Present-Bit und Rahmennummer)
- Danach kann unterbrochenes Programm fortgesetzt werden, Prozess ist rechenbereit

Seitenfehler (3)

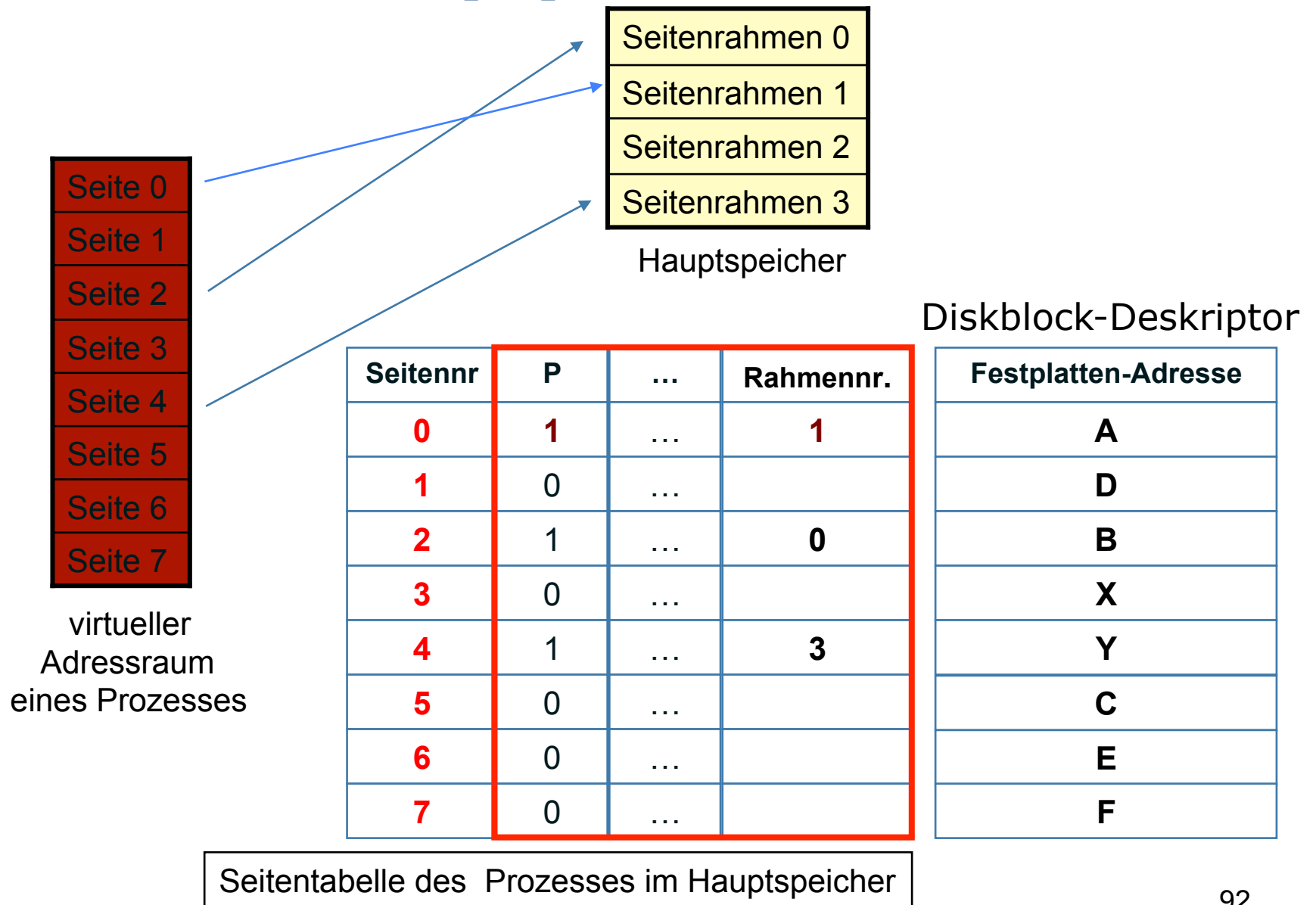
Welche Informationen benötigt das Betriebssystem zum Einlagern von Seiten?

- Abbildung Seitennummer auf Festplattenadresse
- Liste freier Seitenrahmen

Seitenfehler (4)



Seitenfehler (4)



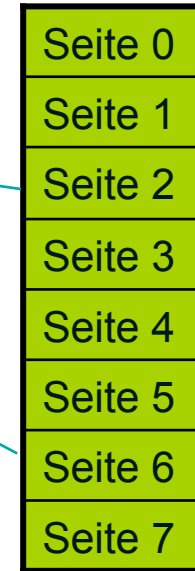
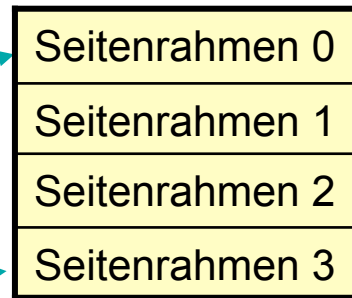
Verdrängung (1)

- Wenn kein freier Rahmen vorhanden:
Verdrängen von Seitenrahmen auf die Festplatte
- Je nach Betriebssystem:
 - Alle Seitenrahmen sind Kandidaten für Verdrängung oder
 - Nur Seitenrahmen des eigenen Prozesses
- Entscheidung unter diesen Kandidaten gemäß Verdrängungsstrategie
- Ziel: Gute Ausnutzung von Lokalität

Verdrängung (2)

- Modify-Bit gesetzt: Schreibe Seite im entsprechenden Rahmen auf Festplatte zurück
- Aktualisiere Seitentabelle (P-Bit, Rahmennummer)
- Generell: Suche von bestimmten Seitenrahmen in Seitentabelle von Prozessen ineffizient
- Weitere Tabelle: Abbildung von Seitenrahmennummer auf $\langle \text{Prozessnummer, Seitennummer} \rangle$

Verdrängung (3)



Seite	P	...	Rahmen
0	0	...	
1	0	...	
2	1	...	0
3	0	...	
4	1	...	3
5	0	...	
6	0	...	
7	0	...	

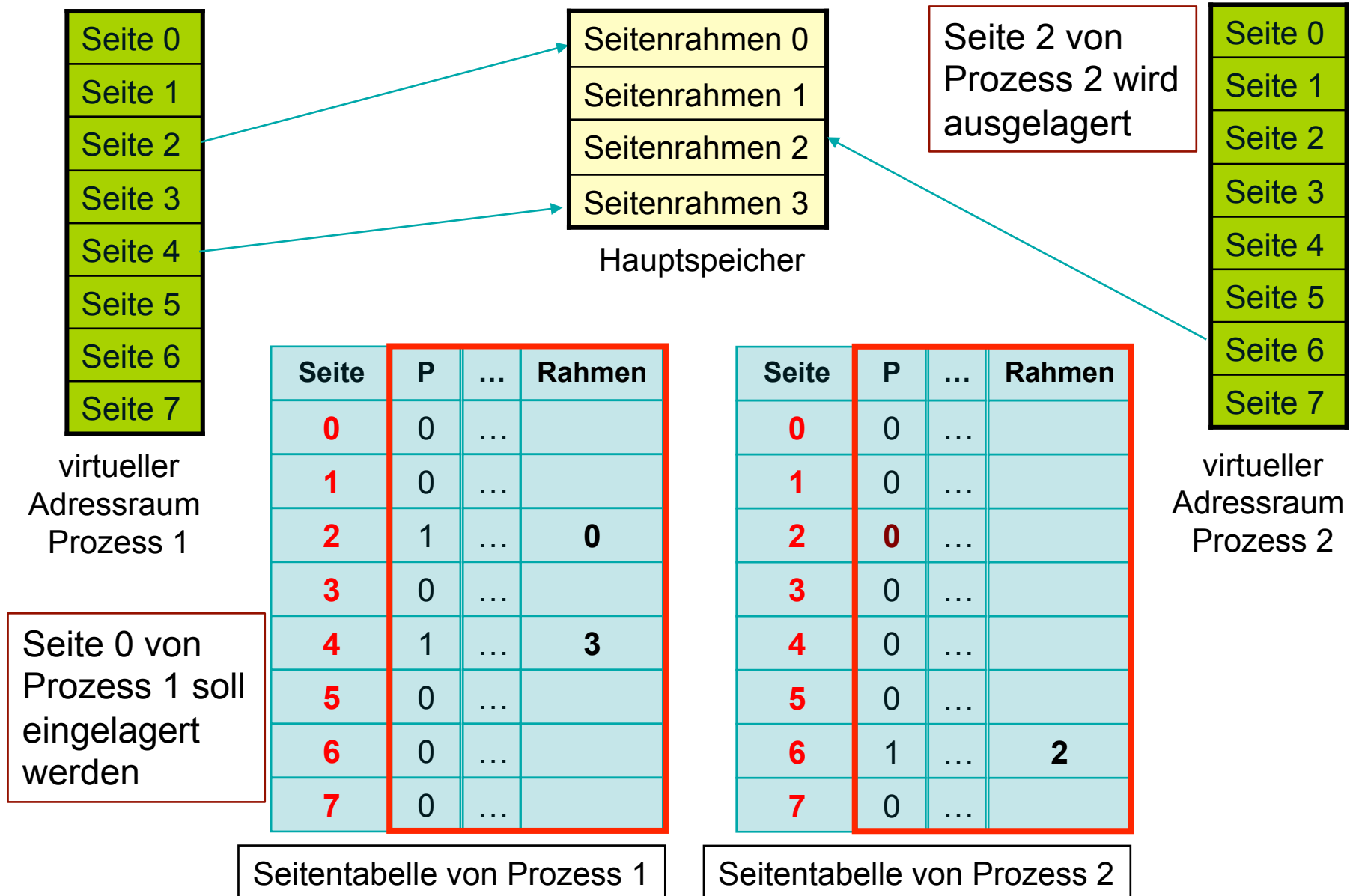
Seitentabelle von Prozess 1

Seite	P	...	Rahmen
0	0	...	
1	0	...	
2	1	...	1
3	0	...	
4	0	...	
5	0	...	
6	1	...	2
7	0	...	

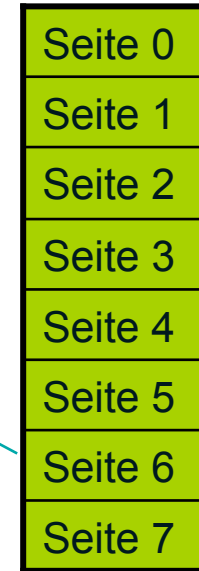
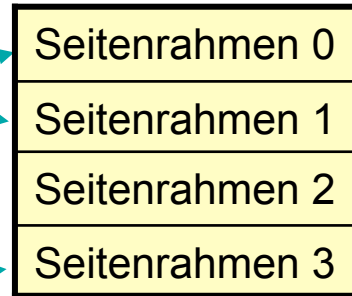
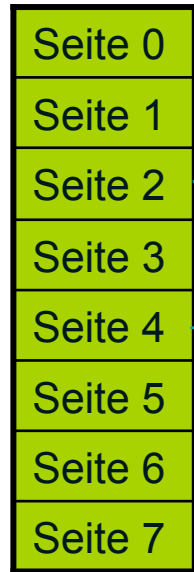
Seitentabelle von Prozess 2

Seite 0 von Prozess 1 soll eingelagert werden

Verdrängung (3)



Verdrängung (3)



Seite	P	...	Rahmen
0	1	...	1
1	0	...	
2	1	...	0
3	0	...	
4	1	...	3
5	0	...	
6	0	...	
7	0	...	

Seitentabelle von Prozess 1

Seite	P	...	Rahmen
0	0	...	
1	0	...	
2	0	...	
3	0	...	
4	0	...	
5	0	...	
6	1	...	2
7	0	...	

Seitentabelle von Prozess 2

Seite 0 von Prozess 1 wird eingelagert

Verdrängung (4)

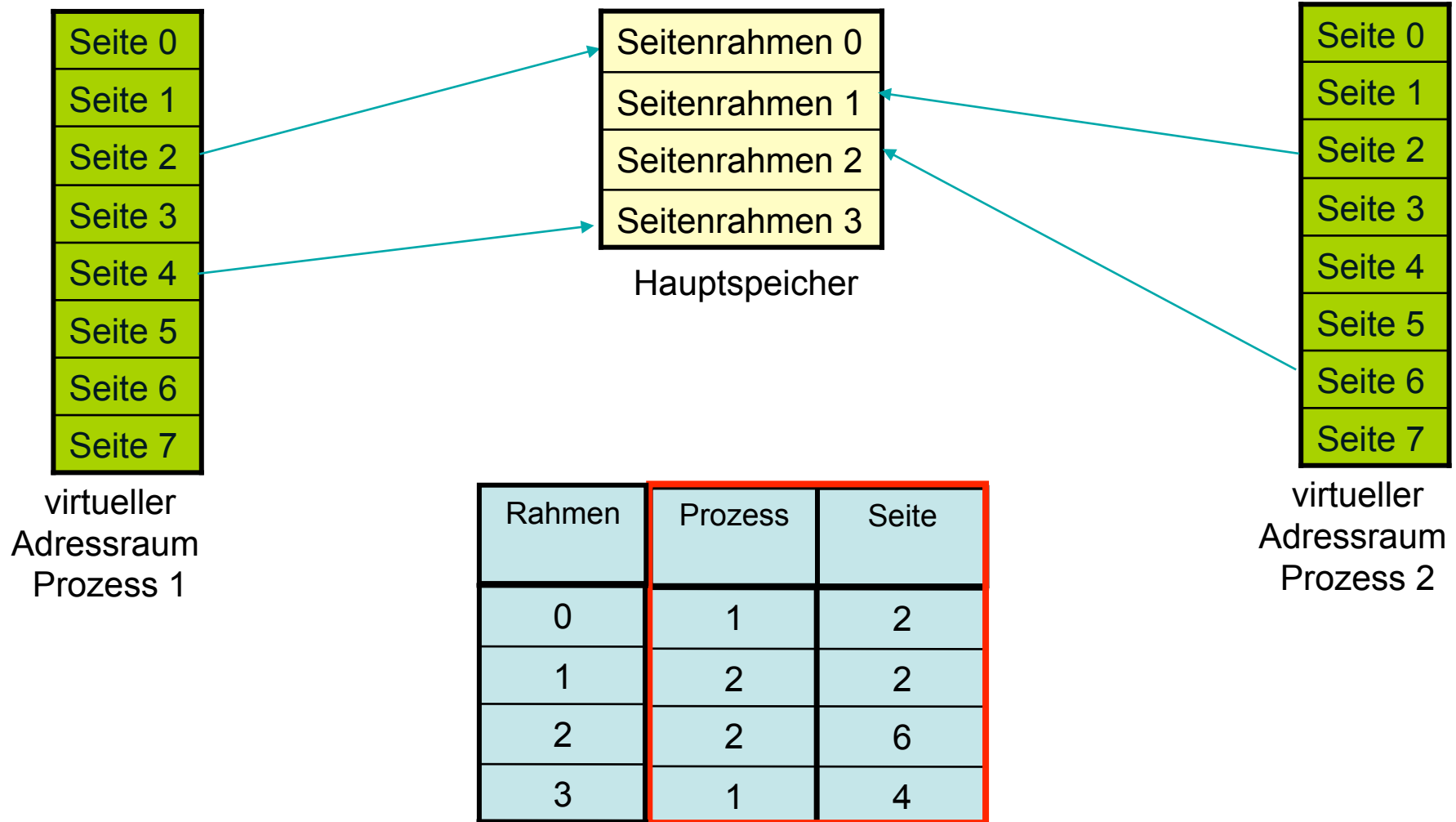


Abbildung Seitenrahmennummer auf <Prozessnummer, Seitennummer>

Größe von Seitentabellen

- Problem: Größe der Seitentabelle bei großem virtuellen Adressraum
- Beispiel: 32-Bit-Adressraum, 4 KiB Seiten
 - 20-Bit-Seitennummer, 12-Bit-Offset
 - Also: 2^{20} Seiten der Größe 2^{12} Byte, Seitentabelle mit 2^{20} Zeilen!
 - Annahme: 4 Byte pro Zeile
 - Also: 2^{22} Byte für Seitentabelle, d.h. 2^{10} Rahmen für Seitentabelle eines Prozesses im Hauptspeicher benötigt

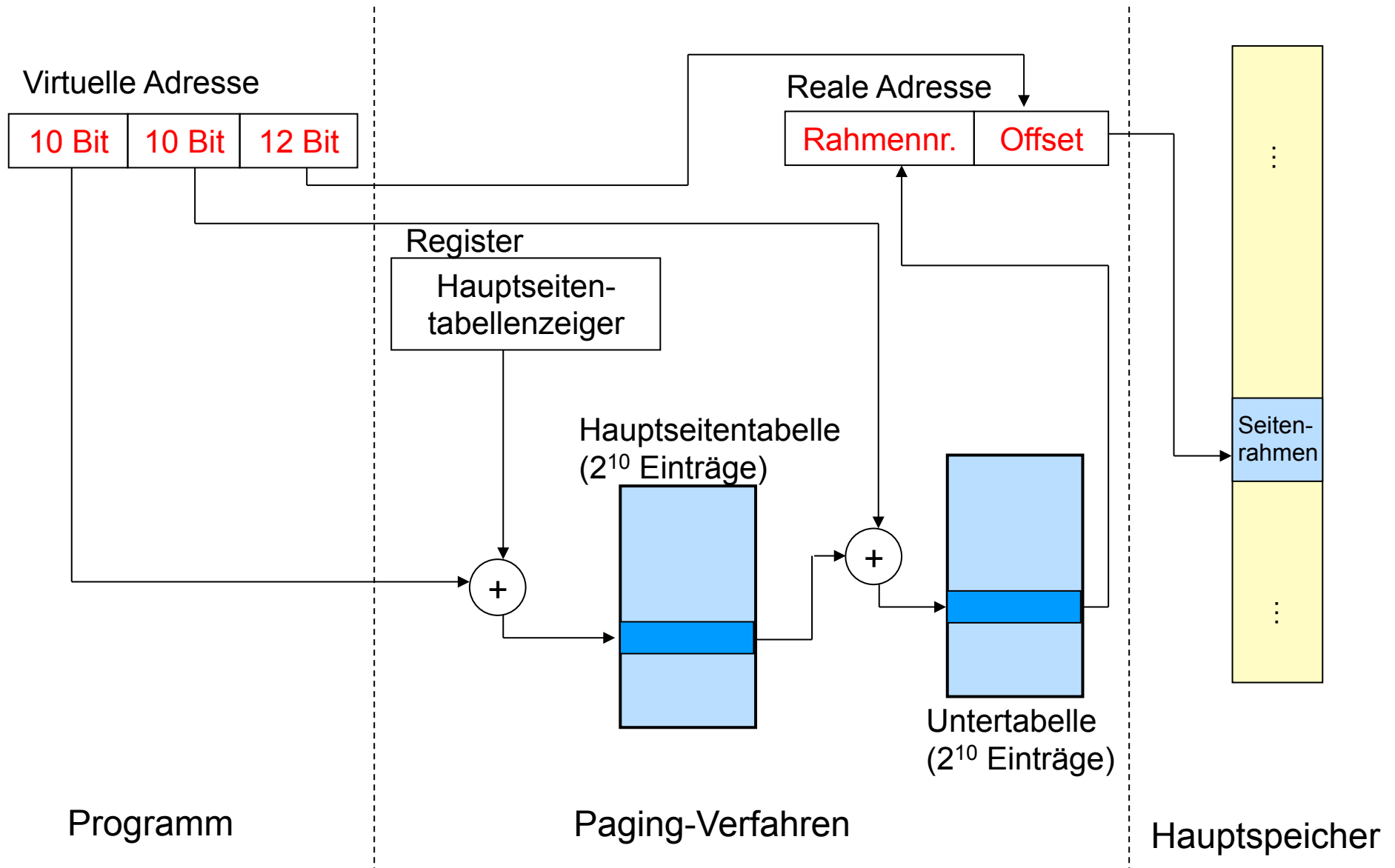
Zweistufige Seitentabellen (1)

- Hierarchische Seitentabelle
- Idee: Speichere auch Seitentabelle im virtuellen Speicher
- Im Beispiel: 2^{20} Seiten müssen angesprochen werden, also werden 2^{10} Rahmen für Seitentabelle benötigt
- Idee: Benutze eine Hauptseitentabelle, die immer im Speicher liegt
- Diese enthält 2^{10} Verweise auf Untertabellen

Zweistufige Seitentabellen (2)

- Erste 10 Bits einer virtuellen 32-Bit-Adresse: Index für die Hauptseite, um die benötigte Untertabelle zu finden
- Wenn entsprechende Seite nicht im Speicher: Lade in freien Seitenrahmen
- Mittlere 10 Bits der Adresse: Index von Seitenrahmen in Untertabelle
- Also Referenzen auf 2^{20} Seiten möglich
- Restliche 12 Bit der virtuellen Adresse: wie vorher Offset innerhalb des Seitenrahmens

Adressumsetzung



Invertierte Seitentabellen (1)

- Alternative für noch größere Adressräume
- Viel größere Anzahl von Seiten des virtuellen Adressraumes als zugeordnete Rahmen von Prozessen
- Seitentabellen meist nur sehr dünn besetzt
- Seitentabellen zur Abbildung Seitennummer auf Rahmennummer verschwenden Speicherplatz

Invertierte Seitentabellen (2)

- Nicht für jede virtuelle Seite einen Eintrag, sondern für jeden **physischen Seitenrahmen**
- Speichere zu Seitenrahmen die zugehörige Seitennummer
- Unabhängig von Gesamtanzahl der Seiten der Prozesse: Ein **fester Teil des realen Speichers** für die Tabellen benötigt

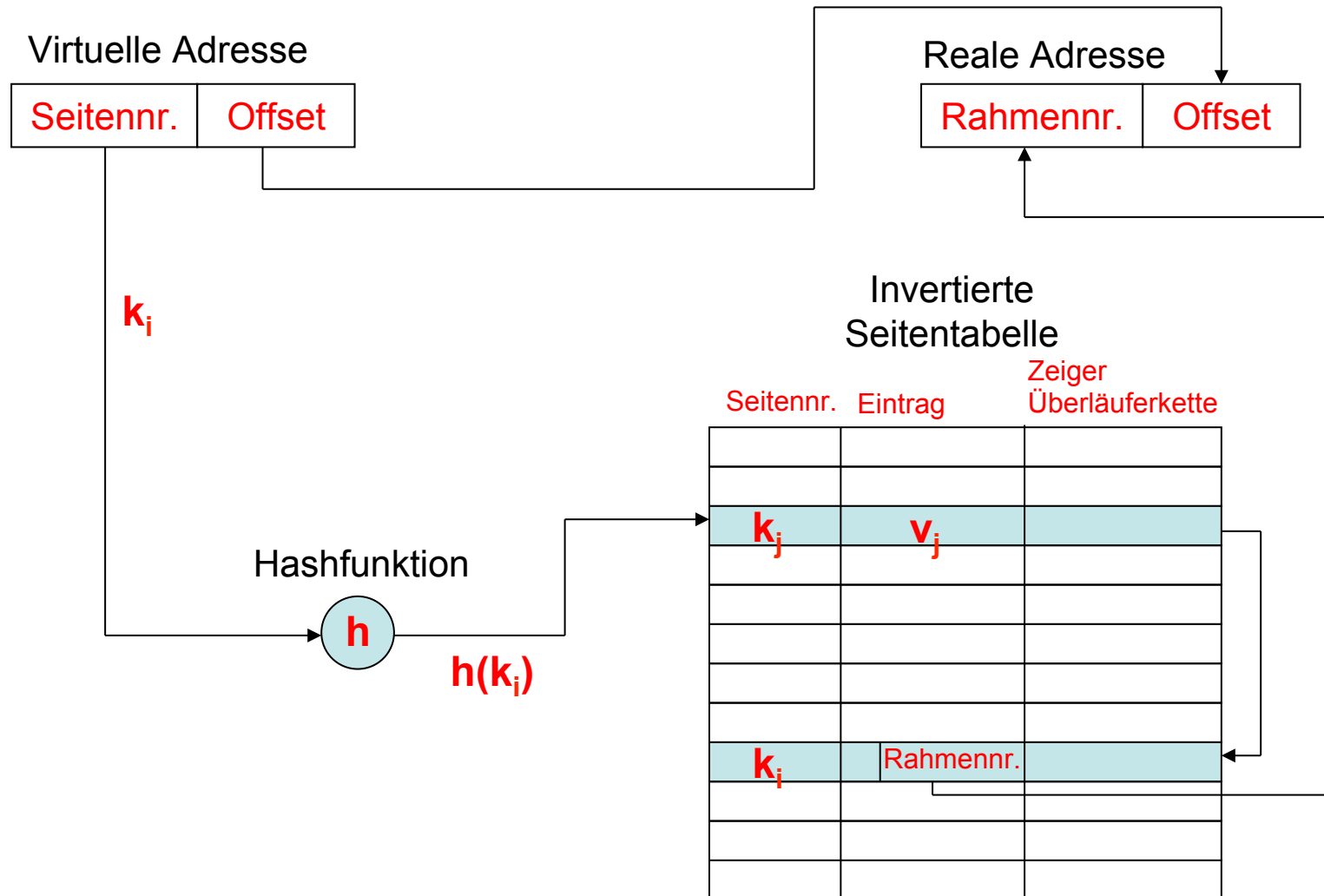
Invertierte Seitentabellen (3)

- Nachteil: Aufwändiger, eine virtuelle Adresse auf eine physische abzubilden
- Benutze eine Hashtabelle um Seitennummern mit Rahmen zu speichern
- $n = \# \text{Seiten}$, $m = \# \text{Rahmen}$, Hashfunktion:
$$h : \{0, \dots, n-1\} \rightarrow \{0, \dots, m-1\}$$
- Sei k_i Seitennummer, einfaches Beispiel:
$$h(k_i) = k_i \bmod m$$
- Bei Vergabe eines neuen Seitenrahmens v_i :
Speichere an Platz $h(k_i)$ das Paar (k_i, v_i) ab

Invertierte Seitentabellen (4)

- Problem: Hashkollisionen (mehr als eine virtuelle Seitennummer wird derselbe Hashwert zugewiesen)
- Verkettete Liste zur Verwaltung des Überlaufs
- Suche mit Schlüssel k_i : Nachschauen an Stelle $h(k_i)$
- Wenn Stelle belegt: Überprüfe, ob Schlüssel k_i übereinstimmt
- Wenn nicht: Verfolge Überläuferkette

Invertierte Seitentabellen (5)



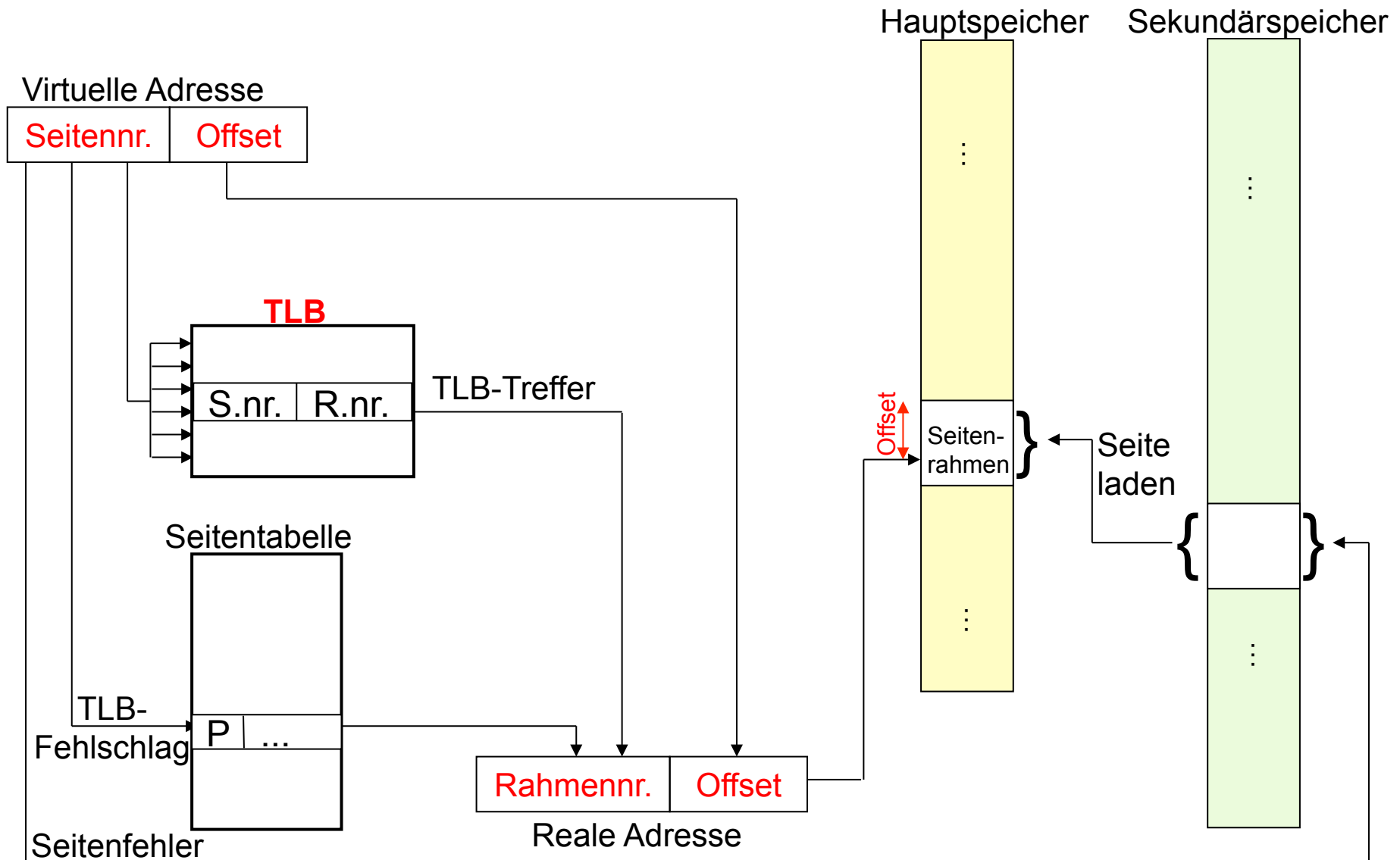
Translation Lookaside Buffer (TLB)

- Bei Speicherzugriff mit Paging: mindestens ein zusätzlicher Zugriff auf die Seitentabelle
- Hardwaremäßige Beschleunigung durch zusätzlichen Cache für Adressübersetzung (Adressumsetzungspuffer, TLB)
- TLB enthält Seitentabelleneinträge, auf die zuletzt zugegriffen wurde (Lokalitätsprinzip)

TLB: Suche nach virtueller Adresse

- Sieh nach, ob Eintrag zu virtueller Adresse in TLB
- Wenn ja: Lies Rahmennummer und bilde reale Adresse
- Sonst:
 - Sieh nach in Seitentabelle, ob P-Bit gesetzt
 - Wenn ja: Aktualisiere TLB durch Einfügen des neuen Seitentabelleneintrags, bilde Adresse
- Wenn Seite nicht im Hauptspeicher: Lade Seite von Festplatte nach und aktualisiere Seitentabelle

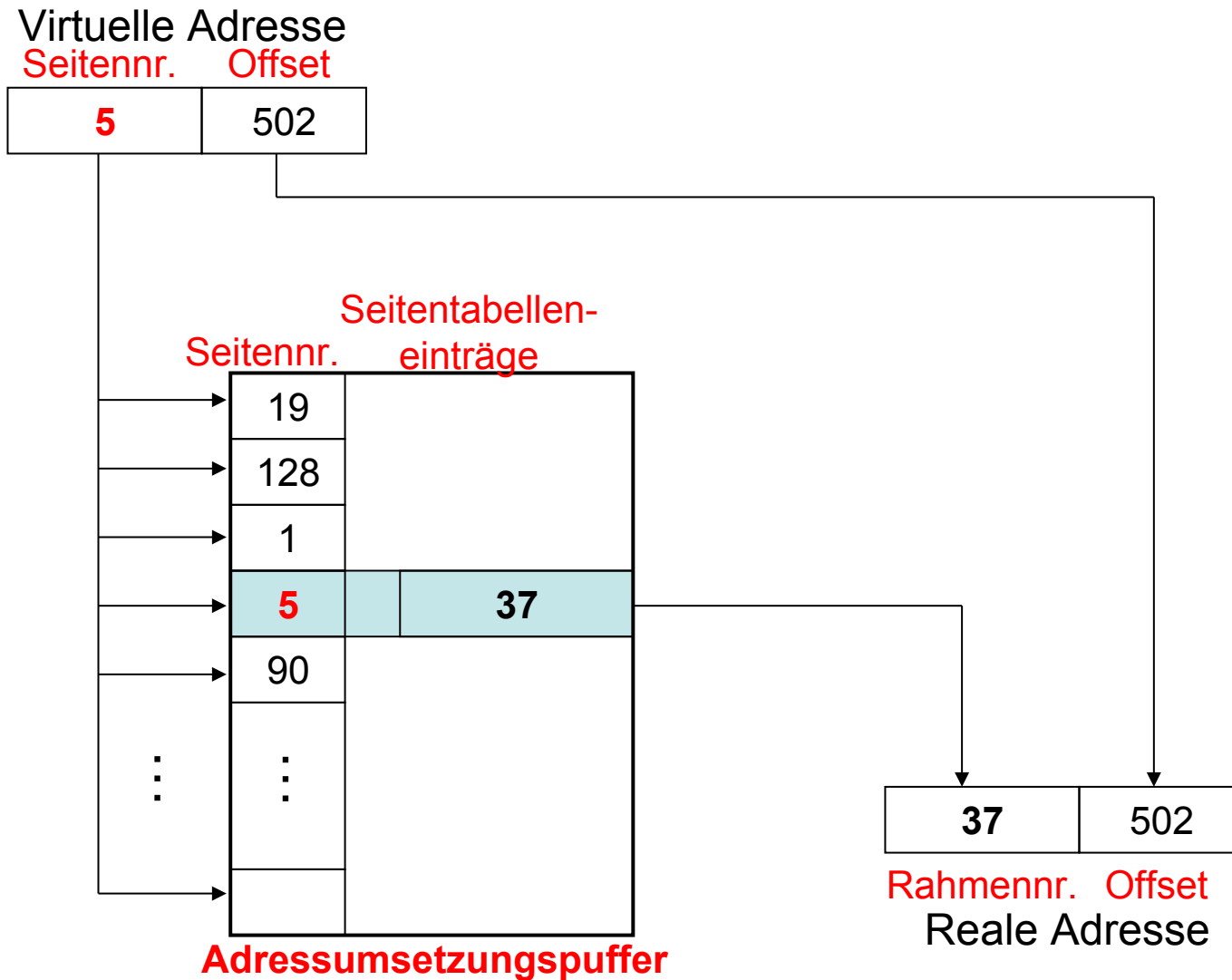
Verwendung des TLB



TLB: Assoziative Zuordnung (1)

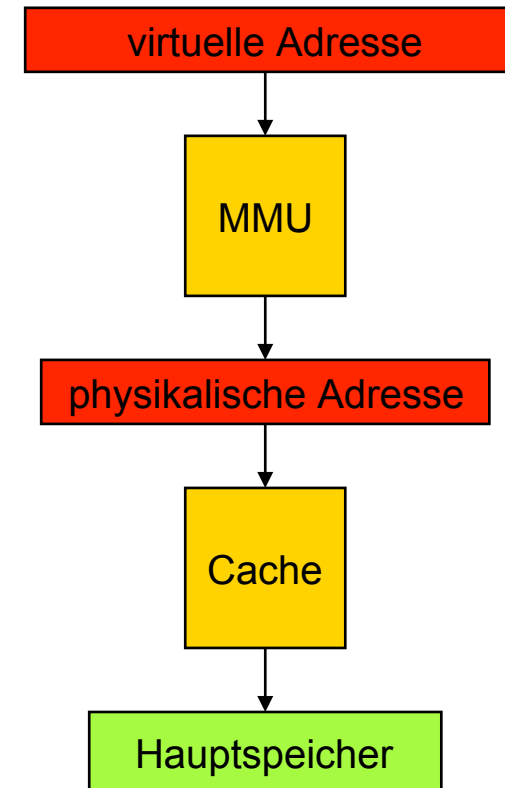
- TLB enthält nur einige Einträge
- Seitennummer kann nicht als Index dienen
- Angefragte Seitennummer wird durch Hardware **parallel** mit allen Einträgen in TLB verglichen
- Bei neuem Eintrag: Verdrängungsstrategie notwendig

TLB: Assoziative Zuordnung (2)



TLB und Caches

- Zusätzlich noch Caches für Programme und Daten
- Verwenden physikalische Adressen



Seitengröße

- Wahl der Seitengröße wichtig für Effizienz
- Kleine Seiten: Wenig Verschnitt (interne Fragmentierung)
- Große Seiten: Kleinere Seitentabellen, weniger Verwaltungsaufwand
- In Realität: Seitengrößen 4 KiB bis 1 GiB

Segmentierung (1)

- Virtueller Adressraum eines Prozesses aufgeteilt in Segmente mit verschiedener Größe (z.B. Code, Daten)
- Größe der Segmente unterschiedlich und dynamisch, können wachsen
- Nicht notwendigerweise zusammenhängende Speicherbereiche
- Nicht alle Segmente eines Prozesses müssen im Arbeitsspeicher vorhanden sein

Segmentierung (2)

- Keine interne Fragmentierung, aber externe Fragmentierung
- Zuteilungsalgorithmen benötigt (z.B. First fit oder Best fit)
- Schutz und gemeinsame Nutzung auf Segmentbasis einfach zu regeln
- Segmenttabelleneintrag:

P	M	Weitere Bits	Länge	Basisadr.
---	---	--------------	-------	-----------

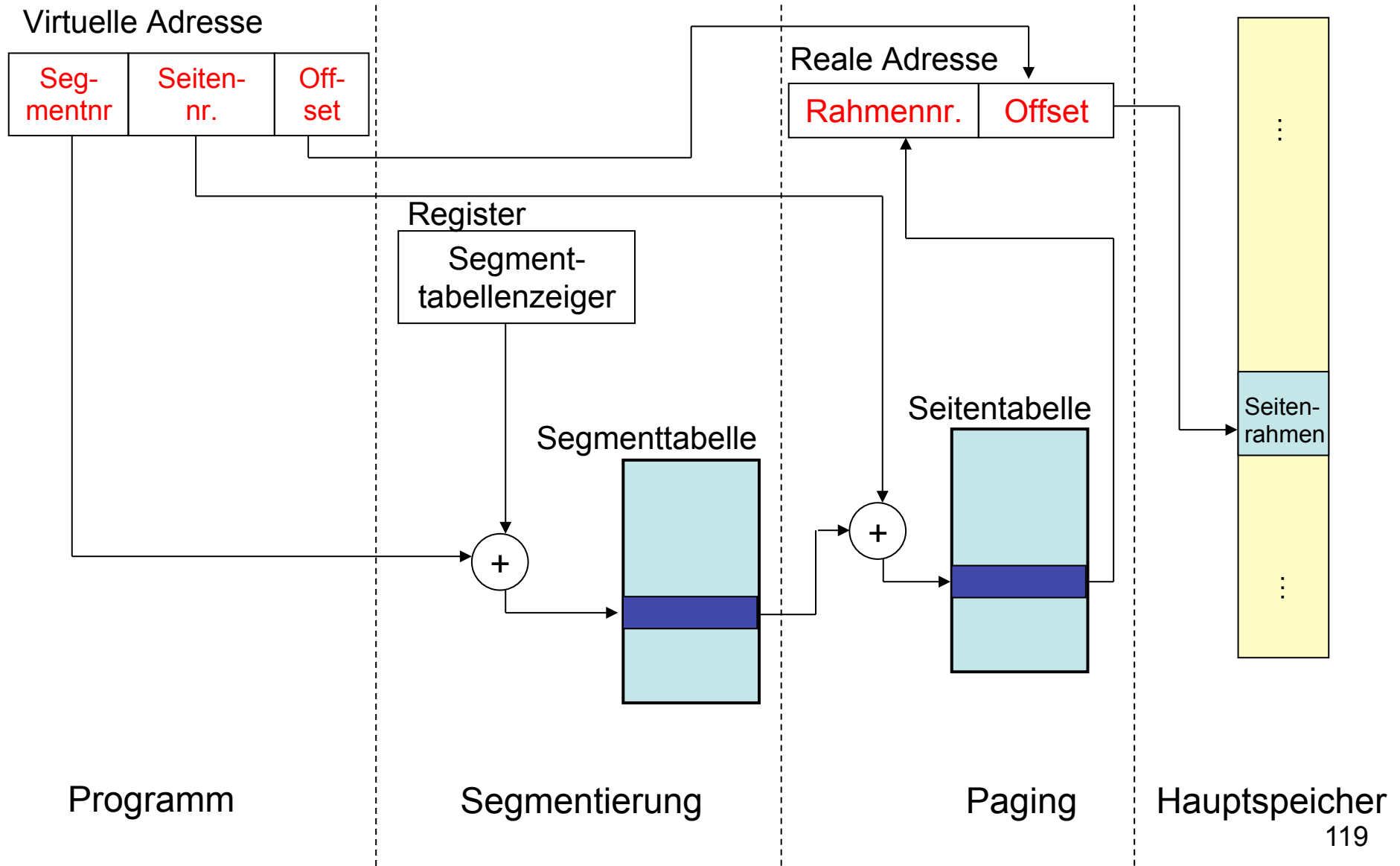
Segmentierung (3)

- Adressumrechnung komplexer: Virtuelle Adresse besteht aus $\langle \text{Segmentnummer}, \text{Offset} \rangle$, reale Adresse: **Basisadresse+Offset**
- Wenn Prozesse wachsen, Anpassung an dynamischen Speicherbedarf
- Segmentvergrößerung:
 - Allokieren von nachfolgendem Speicher oder
 - Verschiebung in einen größeren freien Bereich (kann sehr aufwändig sein)

Segmentierung (4)

- Um große Segmente effizient zu verwalten:
Kombination Segmentierung mit Paging
- Prozesse aufgeteilt in Segmente, **pro Prozess eine Segmenttabelle**
- Segmente aufgeteilt in Seiten fester Größe,
pro Segment eine Seitentabelle
- Aufbau einer Adresse:
 - Virtuelle Adresse:
 $\langle \text{Segmentnummer}, \text{Offset}_{\text{Segmentierung}} \rangle$
 - $\text{Offset}_{\text{Segmentierung}}$ beim Paging interpretiert als:
 $\langle \text{Seitennummer}, \text{Offset}_{\text{Paging}} \rangle$

Segmentierung und Paging kombiniert: Adressumsetzung



Aspekte der Speicherverwaltung

Wichtige Entscheidungen:

- Wann werden Seiten in den Hauptspeicher geladen?
- Welche Seiten werden ausgelagert?
- Wie viele Rahmen darf ein Prozess belegen?

Abrufstrategie

- **Paging on Demand:**
 - Lade Seite erst in den Hauptspeicher, wenn sie benötigt wird
 - Anfangs: Viele Seitenfehler, danach Verhalten entsprechend Lokalitätsprinzip
- **Prepaging:**
 - Neben einer angeforderten Seite werden noch weitere in der Umgebung geladen
 - Nicht besonders effizient, wenn die meisten Seiten nicht benötigt werden
 - In Praxis nicht eingesetzt

Austauschstrategie (1)

- Auswahl einer Seite, die ausgelagert werden soll, wenn der Speicher voll ist
- Nach Möglichkeit: Seite, auf die in nächster Zeit wahrscheinlich nicht zugegriffen wird
- Vorhersage anhand der vergangenen Speicherreferenzen

Austauschstrategie (2)

- **LRU** (Least Recently Used): Lagere Seite aus, auf die am längsten nicht zugegriffen wurde
- **FIFO** (First In First Out): Lagere Seite aus, die zuerst eingelagert wurde
- **Clock-Algorithmus:**
 - Rahmen/Seiten haben Use Bit: Anfangs 0
 - Setze auf 1, wenn nach Seitenzugriffsfehler referenziert
 - Suche im Uhrzeigersinn Seite mit Use Bit 0
 - Setze Use Bit auf 0 bei überquerten Seiten
 - Geringer Overhead, gute Leistung

Größe des Resident Set (1)

- Entscheidung, wie viel Hauptspeicher einem Prozess bei Laden zur Verfügung gestellt wird
- Je kleiner der Teil ist, der geladen wird, desto mehr Prozesse können sich im Hauptspeicher befinden
- Desto größer ist aber die Seitenfehlerrate

Größe des Resident Set (2)

- **Feste Zuteilung:**
 - Feste Zahl von Rahmen im Hauptspeicher (abhängig von Art des Prozesses)
 - Bei Seitenfehler: Eine der Seiten dieses Prozesses muss gegen die benötigte Seite ausgetauscht werden
- **Variable Zuteilung:**
 - Anzahl der zugewiesenen Seitenrahmen kann während Lebensdauer variieren
 - Bei ständig hohen Seitenfehlerraten: Zuweisung von zusätzlichen Rahmen
 - Bei niedrigen Seitenfehlerraten: Versuche Anzahl der zugewiesenen Rahmen zu reduzieren

Cleaning Strategie

- **Demand Cleaning:**
 - Seite wird nur dann in Sekundärspeicher übertragen, wenn sie ersetzt wird
 - Bei Seitenfehlern müssen 2 Seitenübertragungen stattfinden
- **Precleaning:**
 - Seiten werden auch geschrieben, bevor ihre Rahmen benötigt werden
 - Kann ineffizient sein

Multiprogramming-Grad

- Entscheidung, wie viele Prozesse sich im Hauptspeicher befinden sollen
- Zu geringe Anzahl: Häufig alle blockiert, Ein- und Auslagern ganzer Prozesse notwendig
- Zu große Anzahl: Größe des Resident Set nicht mehr ausreichend, häufig Seitenfehler
- Wenn geringe Anzahl Seitenfehler, kann Multiprogramming-Grad erhöht werden

Auslagern von Prozessen

- Reduzierung des Multiprogramming-Grads
- Mögliche Strategien:
 - Prozess mit geringster Priorität
 - Prozess, der Seitenfehler verursacht hat
 - Prozess mit kleinstem Resident Set
 - Prozess mit größtem Resident Set
 - Prozess mit größter verbleibender Ausführungszeit

Zusammenfassung (1)

- Speicherverwaltungsstrategien sind extrem wichtig für die Effizienz des Gesamtsystems
- Paging unterteilt den Speicher in viele gleich große Teile
- Betriebssysteme arbeiten mit virtuellem Speicher
- Lokalität ist die Grundvoraussetzung für das effiziente Funktionieren virtueller Speicher-konzepte

Zusammenfassung (2)

- Komplexe Hardware/Software nötig für einen einfachen Speicherzugriff
- Seitentabelleneintrag kann im TLB, Hauptspeicher oder auch auf Festplatte sein
- Seite kann sich im Cache, im Hauptspeicher, oder auf Festplatte befinden
- Bei Bedarf:
 - Nachladen und Auslagern von Seiten und Aktualisieren von Seitentabelleneinträgen
 - Auslagern kompletter Prozesse

Lehrevaluation

- Evaluation läuft bis Sonntag, 09.02.2014
- Bitte online ausfüllen