Systeme I: Betriebssysteme

Kapitel 8 Speicherverwaltung

Maren Bennewitz

Version 13.02.2013

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

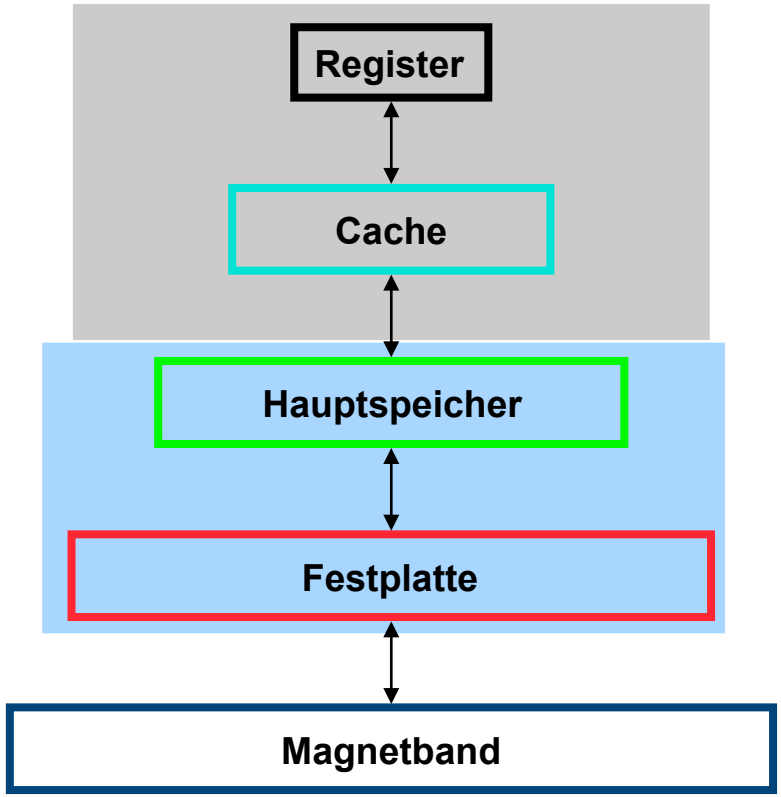
Einführung

- 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 so viele Prozesse wie möglich Platz haben

Adressraum

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

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 Betriebssystemsicht: Möglichst viele Prozesse können im Speicher vorhanden sein
- Es existieren fünf wichtige Anforderungen:
  - Relokation
  - Schutz
  - Gemeinsame Nutzung
  - Logische Organisation
  - Physikalische Organisation

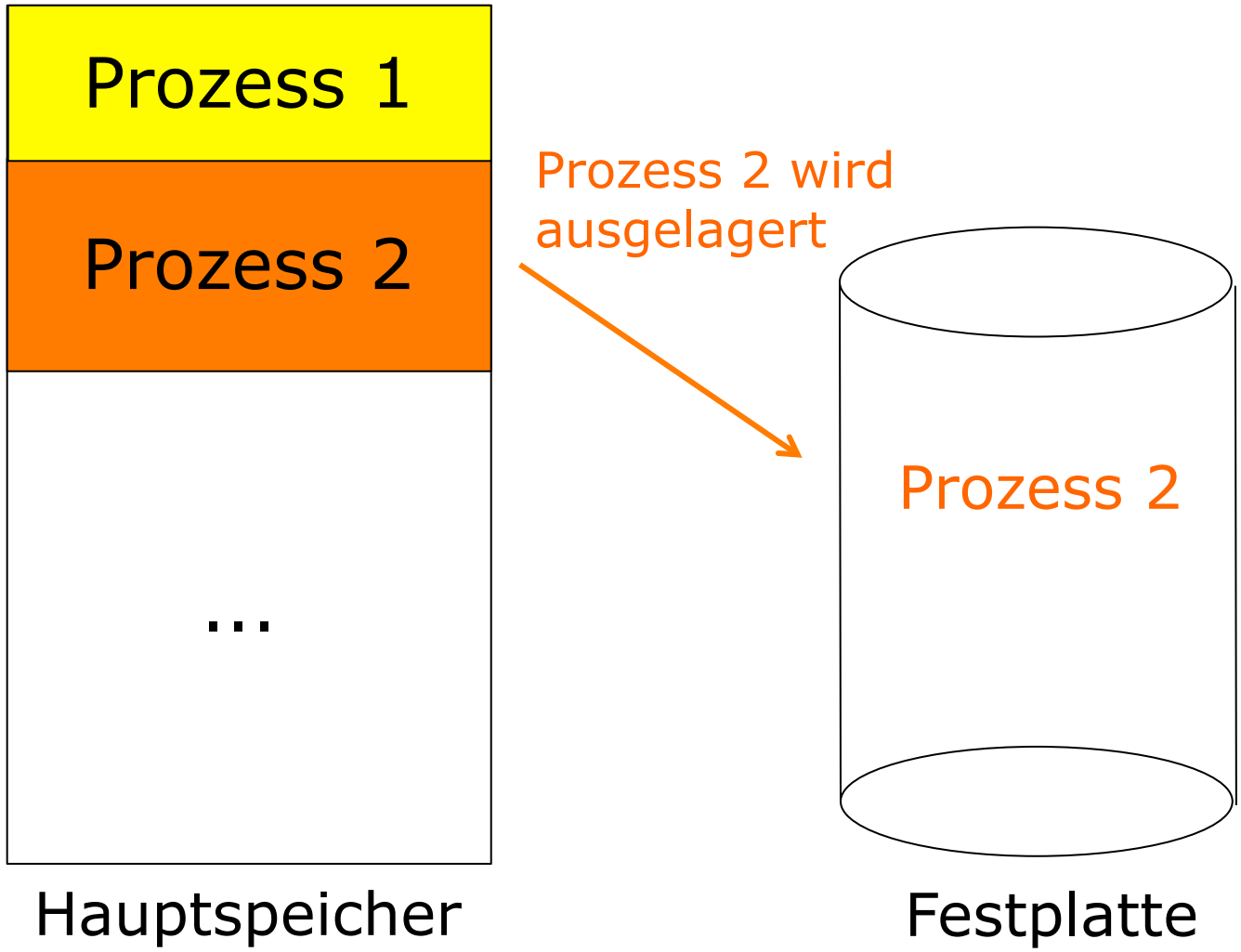
Relokation (1)

- Relokation = Verlagerung
- Motivation:
  - Mehrere Prozesse gleichzeitig im System
  - Auslagern und Wiedereinlagern ganzer Prozesse aus dem Hauptspeicher soll möglich sein
  - Ort der Einlagerung im Voraus unbekannt
  - Bindung an alten Ort beim Wiedereinlagern sehr ungünstig (vermutlich andere Prozesse vorhanden)

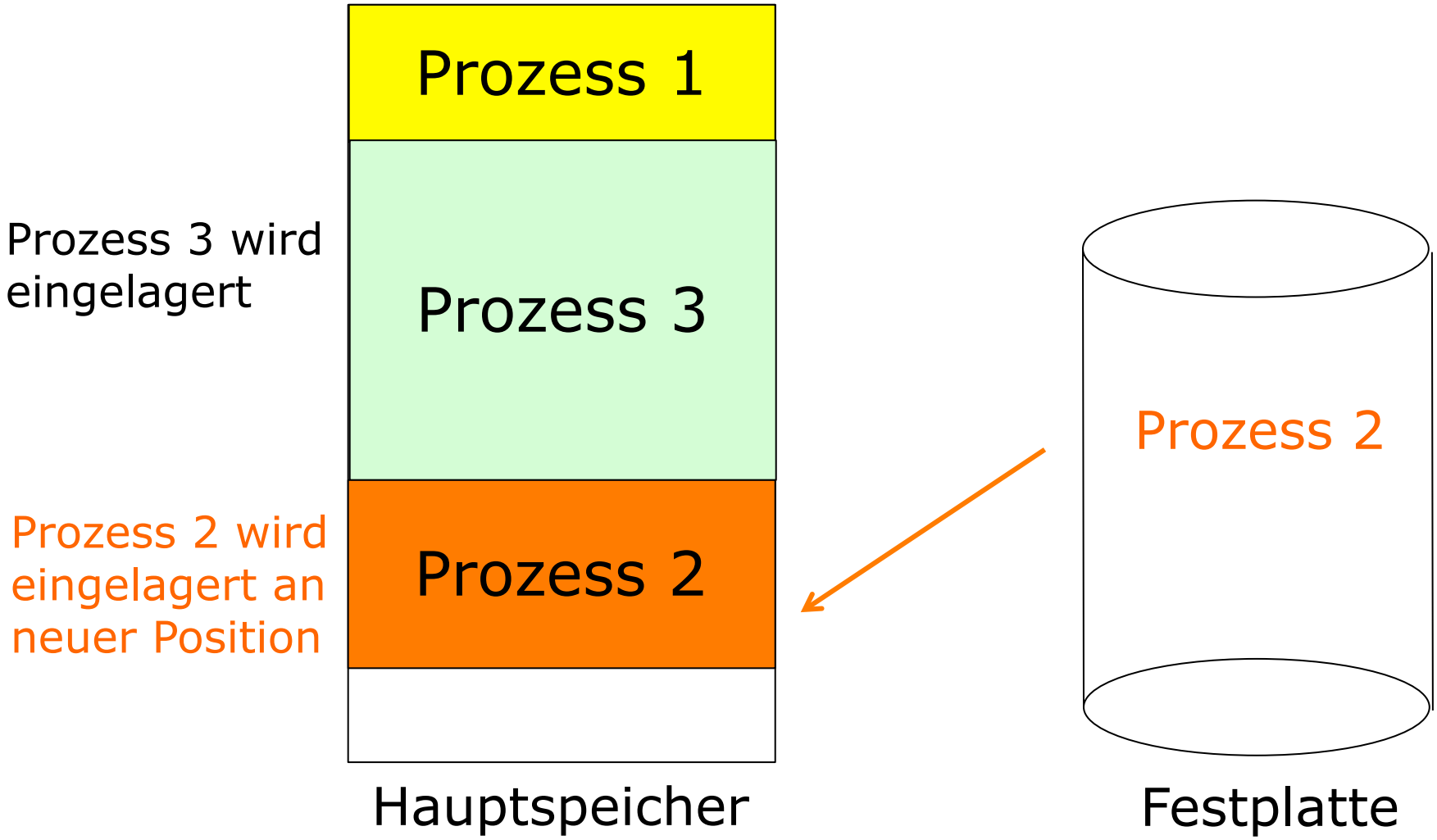
Relokation (2)

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

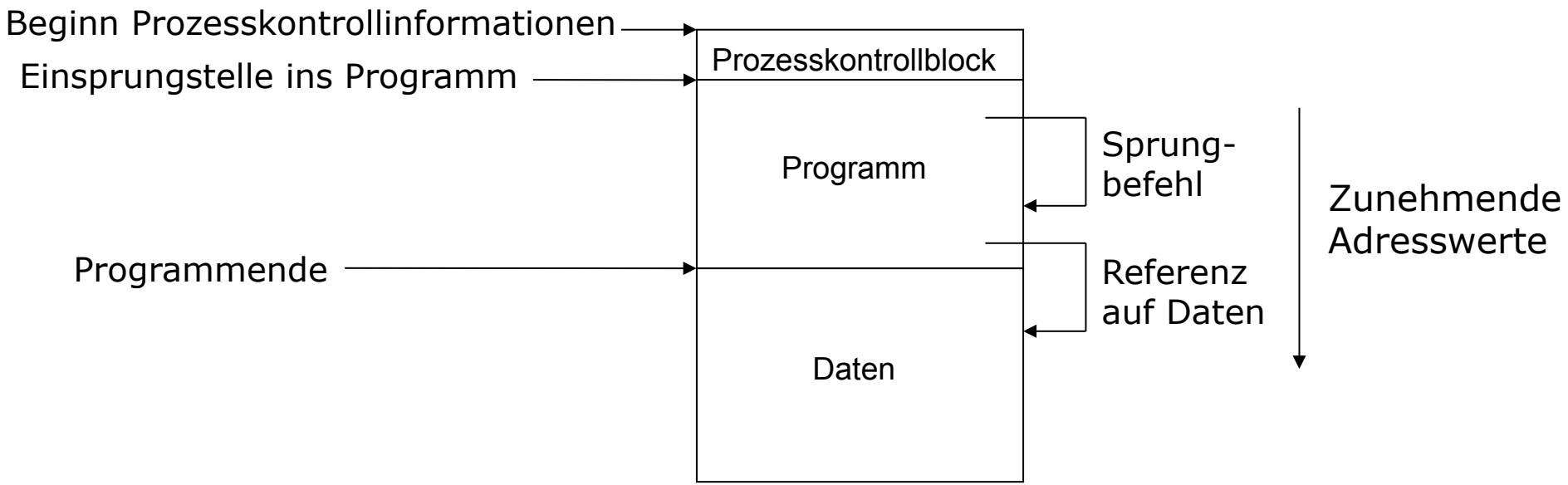
Relokation (3)



Relokation (4)



Relokation (5)



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

Schutz (1)

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

Schutz (2)

- Dynamische Überprüfung zur Laufzeit nötig, um sicherzustellen, dass Prozess nur auf seinen Adressraum zugreift
- Hardwareunterstützung nötig, BS kann nicht alle Speicherreferenzen im Voraus berechnen
- Prozessor muss in der Lage sein, Befehle wie Zugriff auf Datenbereich eines anderen Prozesses bei ihrer Ausführung abzubrechen

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 wird erst zur Laufzeit aufgelöst
  - Verschiedenen 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: Organisation des Informations-flusses zwischen Haupt- und Sekundärspeicher
  - Prinzipiell möglich: in Verantwortung des Programmierers
  - Aufwändig, erschwert durch Multiprogramming
  - Deshalb: Verwaltung durch das Betriebssystem

Grundlegende Methoden der Speicherverwaltung

Partitionierung

- Für Speicheraufteilung zwischen verschiedenen Prozessen (wird häufig bei eingebetteten Systemen benutzt, sonst veraltet); auch intern von BS genutzt

Paging

- Einfaches Paging, kombiniert mit Konzept des virtuellen Speichers

Segmentierung

- Einfache Segmentierung, kombiniert mit Konzept des virtuellen Speichers

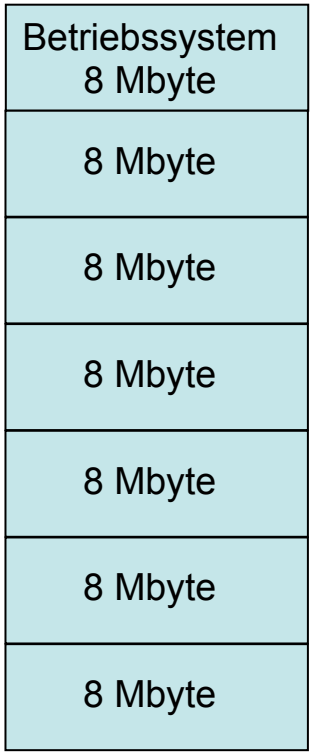
Partitionierung

- 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

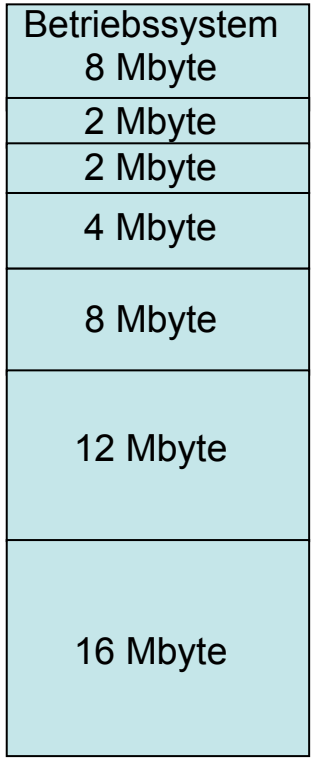
Statische Partitionierung (1)

- Einteilung des Speichers in feste Anzahl von Partitionen
- Zwei Varianten

Alle Partitionen mit gleicher Länge



Alle Partitionen mit unterschiedlicher Länge



Statische Partitionierung (2)

Probleme bei gleich großen Partitionen:

- Programm zu groß für Partition
- Programmerstellung durch Überlagerung nötig (aufwändig!)
- Interne Fragmentierung: Platzverschwendung, wenn Programm kleiner als Größe der zugeordneten Partition
- Fest vorgegebene Anzahl von Prozessen im Speicher

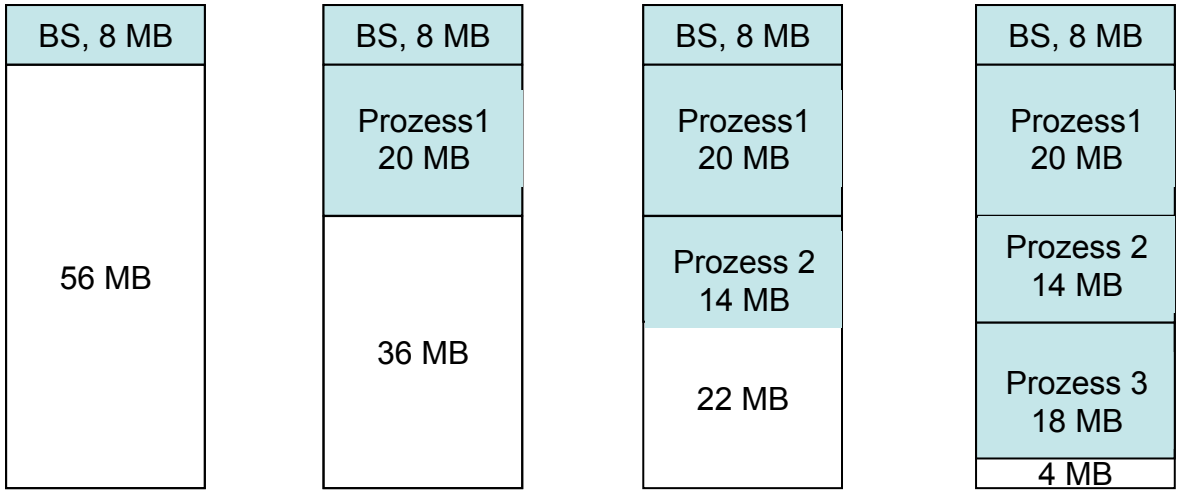
Statische Partitionierung (3)

- Partitionen variabler Länge:
  - Größere Programme können untergebracht werden
  - Kleinere Programme führen zu geringerer interner Fragmentierung
- Zuweisung von Partitionen an Prozesse:
  - Bei Bereichen mit gleicher Länge: trivial
  - Bei Bereichen mit variabler Länge: z.B. kleinste verfügbare Partition, die gerade noch ausreicht
    (Verwaltung nicht ganz trivial)
  - Oft Speicherbedarf nicht im Voraus feststellbar
    (dafür Verfahren des virtueller Speichers, siehe später)

Dynamische Partitionierung (1)

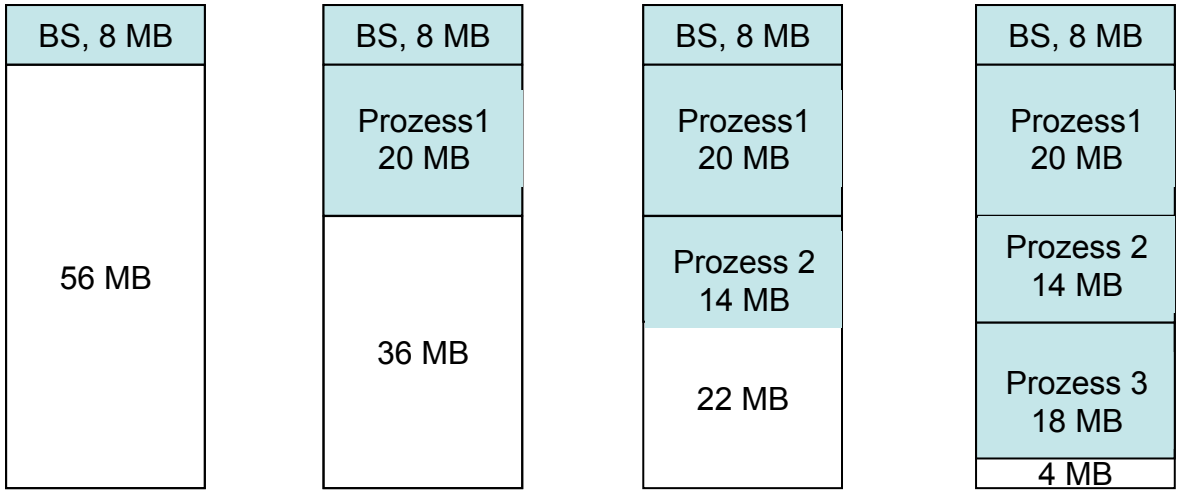
- Einteilung des Speichers in Partitionen
  - variabler Länge und
  - variabler Anzahl
- Prozesse erhalten exakt passende Speicher-bereiche (keine interne Fragmentierung wie vorher)
- Aber: Ein- und Auslagern führt zu externer Fragmentierung: Vielzahl kleiner Lücken, Speicherauslastung nimmt ab
  (vgl. auch Kapitel 3 Dateisysteme)

Dynamische Partitionierung (2)

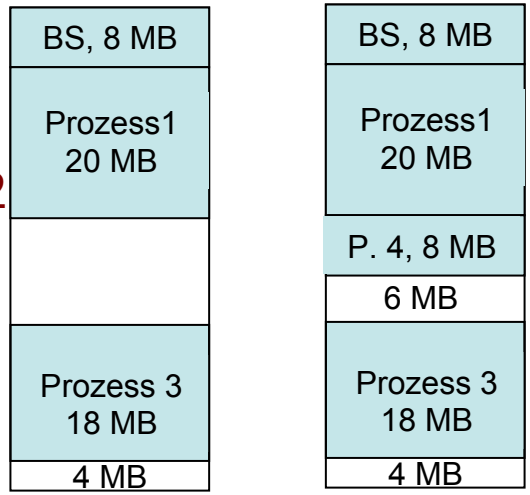


Anforderung:
Prozess 4
braucht 8 MB

Dynamische Partitionierung (2)



Anforderung: Prozess 4 braucht 8 MB

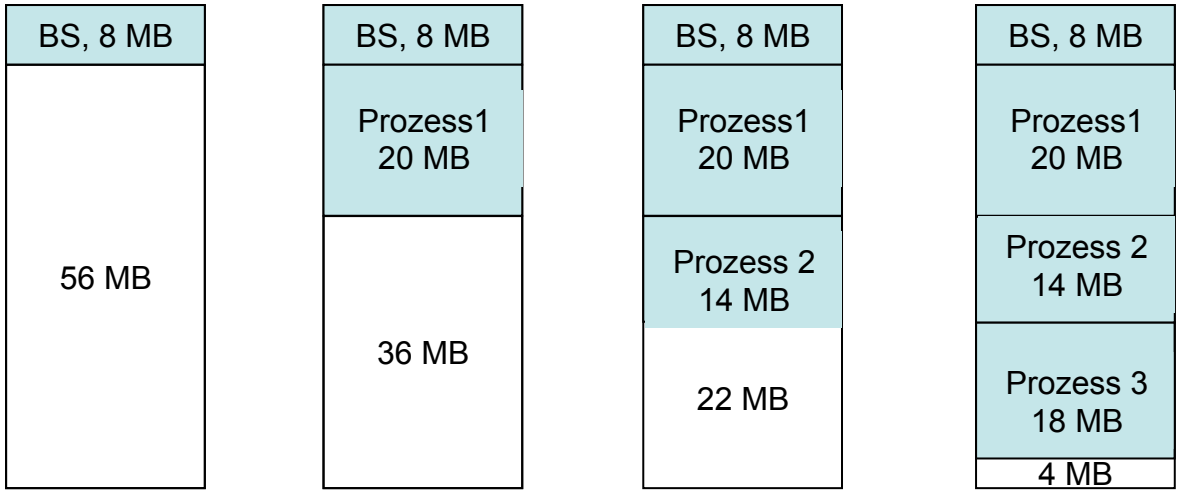


Proz. 2 wird ausge-lagert

genügend Platz für Prozess 4, aber Lücke entsteht

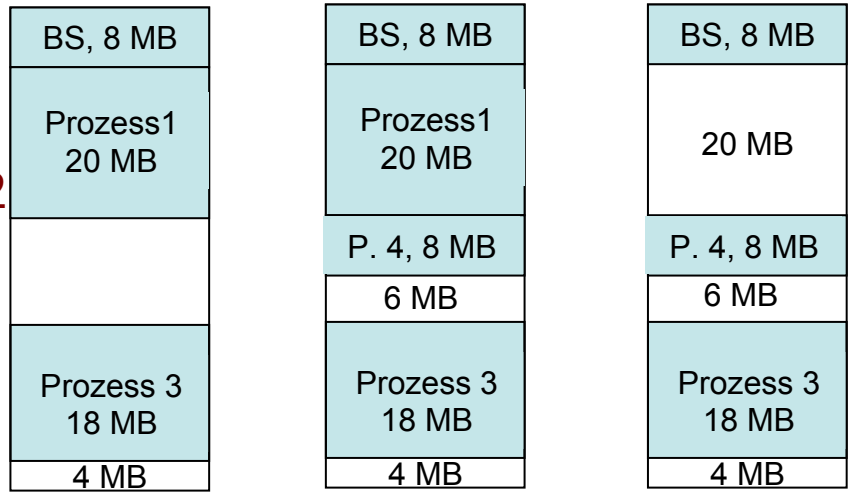
Annahme: Kein Prozess im Hauptspeicher bereit, aber ausgelagerter Prozess 2 bereit

Dynamische Partitionierung (2)



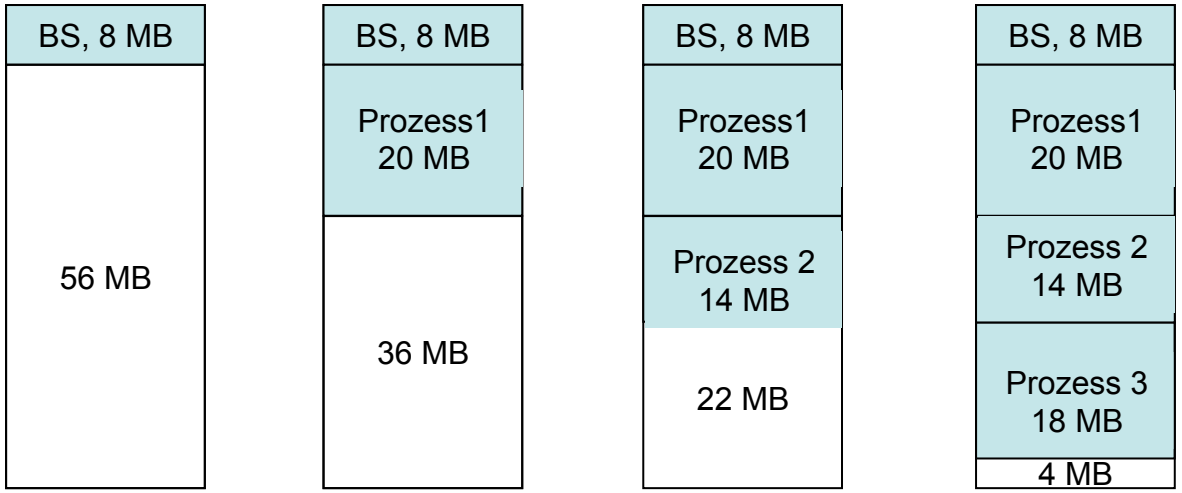
Anforderung:
Prozess 4
braucht 8 MB

Proz. 2
wird
ausge-
lagert



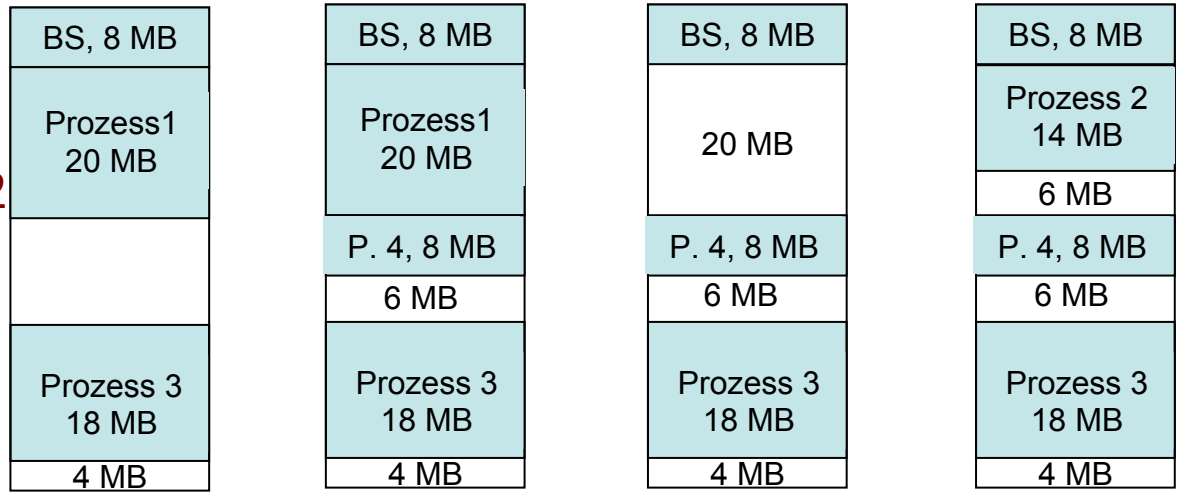
da nicht genügend
Platz für Prozess 2:
Prozess 1 wird
ausgelagert

Dynamische Partitionierung (2)



Anforderung:
Prozess 4
braucht 8 MB

Proz. 2
wird
ausge-
lagert



Prozess 2 wird
wieder eingelagert

Dynamische Partitionierung (3)

- Defragmentierung erforderlich!
  - Verschiebung aufwändig
  - Speicherverdichtung nur erfolgreich, wenn dynamische Relokation möglich (damit Speicherreferenzen nicht ungültig werden)
- Speicherzuteilungsalgorithmen:
  - Best Fit: Suche kleinsten Block, der ausreicht
  - First Fit: Suche beginnend mit Speicheranfang bis ausreichender Block gefunden
  - Next Fit: Suche beginnend mit der Stelle der letzten Speicherzuweisung

Dynamische Partitionierung (4)

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
  - Produziert schnell eine Reihe von sehr kleinen Fragmenten, die ohne Defragmentierung nie mehr benutzt werden können
  - Zudem: langsames Verfahren

Buddy System (1)

- Nachteile statischer Partitionierung:
  - Beschränkte Anzahl nicht-ausgelagerter Prozesse
  - Interne Fragmentierung
- Nachteile dynamischer Partitionierung:
  - Schwierigere Verwaltung
  - Defragmentierung nötig wegen externer Fragmentierung
- Buddy-System (Halbierungsverfahren): Kompromiss zwischen statischer und dynamischer Partitionierung
- Buddy = Partner

Buddy System (2)

Eigenschaften:

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

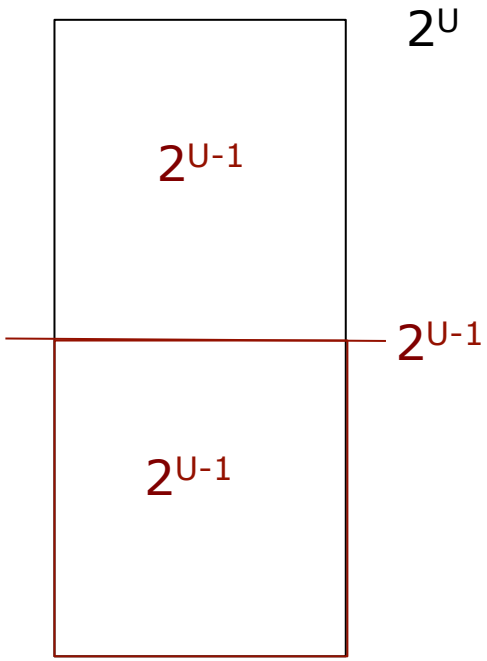
Buddy System (3)

- 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 (4)

Anforderung eines Blocks der Größe s:

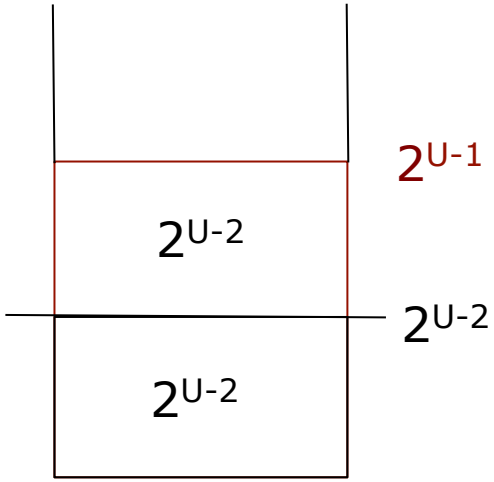
- Wenn $2^{U-1} < s \leq 2^{U}$: Weise gesamten Speicher zu
- Sonst: Teile auf in zwei Blöcke der Größe 2^{U-1}



Buddy System (4)

Anforderung eines Blocks der Größe s:

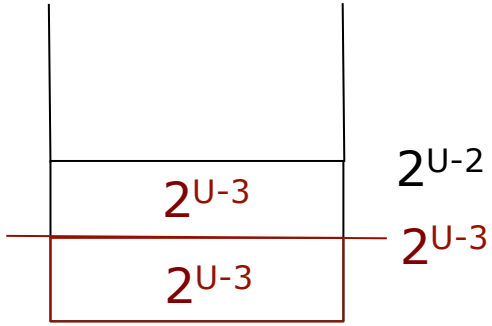
- Wenn $2^{U-1} < s \leq 2^{U}$: Weise gesamten Speicher zu
- Sonst: Teile auf in zwei Blöcke der Größe 2^{U-1}
- Wenn $2^{U-2} < s \leq 2^{U-1}$: Weise einen der beiden Blöcke zu
- Sonst: Wähle einen der beiden Blöcke aus und teile



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 einen der beiden Blöcke zu
- Sonst: Wähle einen der beiden Blöcke aus und teile
- …



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 einen der beiden Blöcke zu
- Sonst: Wähle einen der beiden Blöcke aus und teile
- …
- Fahre fort bis zu Blöcken der Größe 2^{K} mit $2^{K-1} < s \leq 2^{K}$
- Weise einen der beiden Blöcke 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

Buddy System (6)

- Wenn nach Freigabe eines Blocks der Größe 2^K der entsprechende Partnerblock der Größe 2^K ebenfalls frei war:
  - Verschmelze die Blöcke zu einem Block der Größe 2^{K+1}
  - Mache rekursiv mit Verschmelzen weiter

Buddy-System (7)

Beispiel: Speicher der Größe 1 GB

Folge von Anforderungen und Freigaben:

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

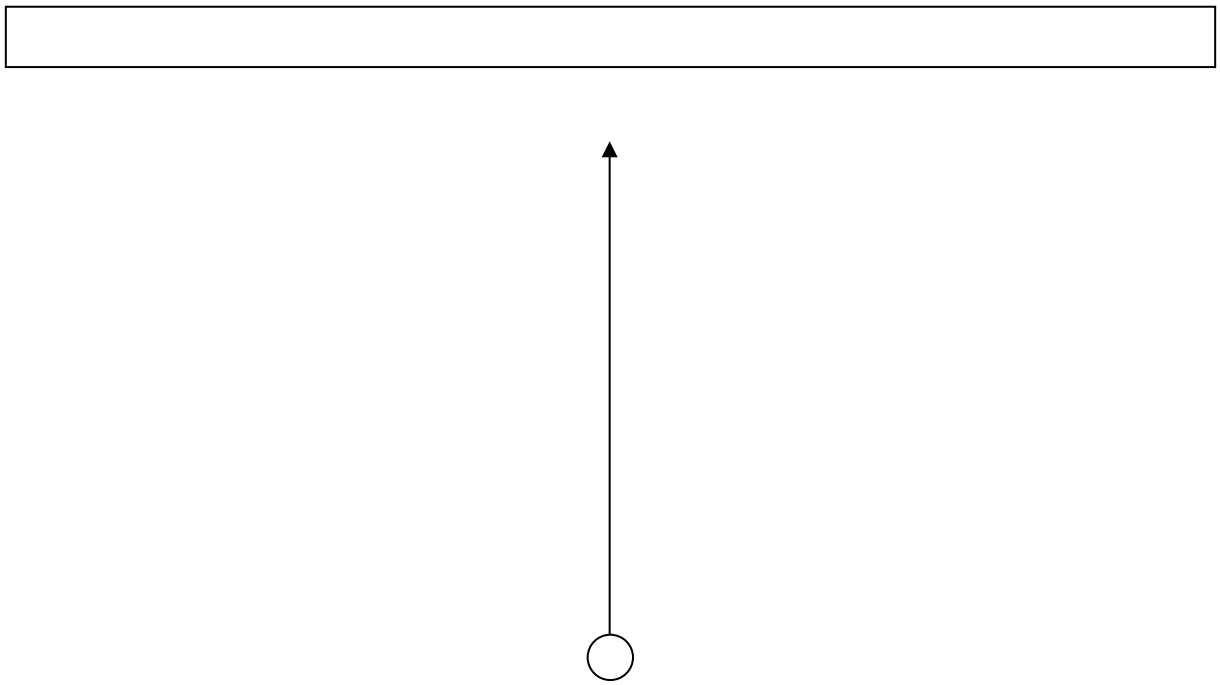
Annahme:

- Obergrenze der Blockgröße: 1 GB
- Untergrenze: 64 MB

Buddy-System (8)

Zunächst verfügbar:

1 GB



Freie Blöcke:

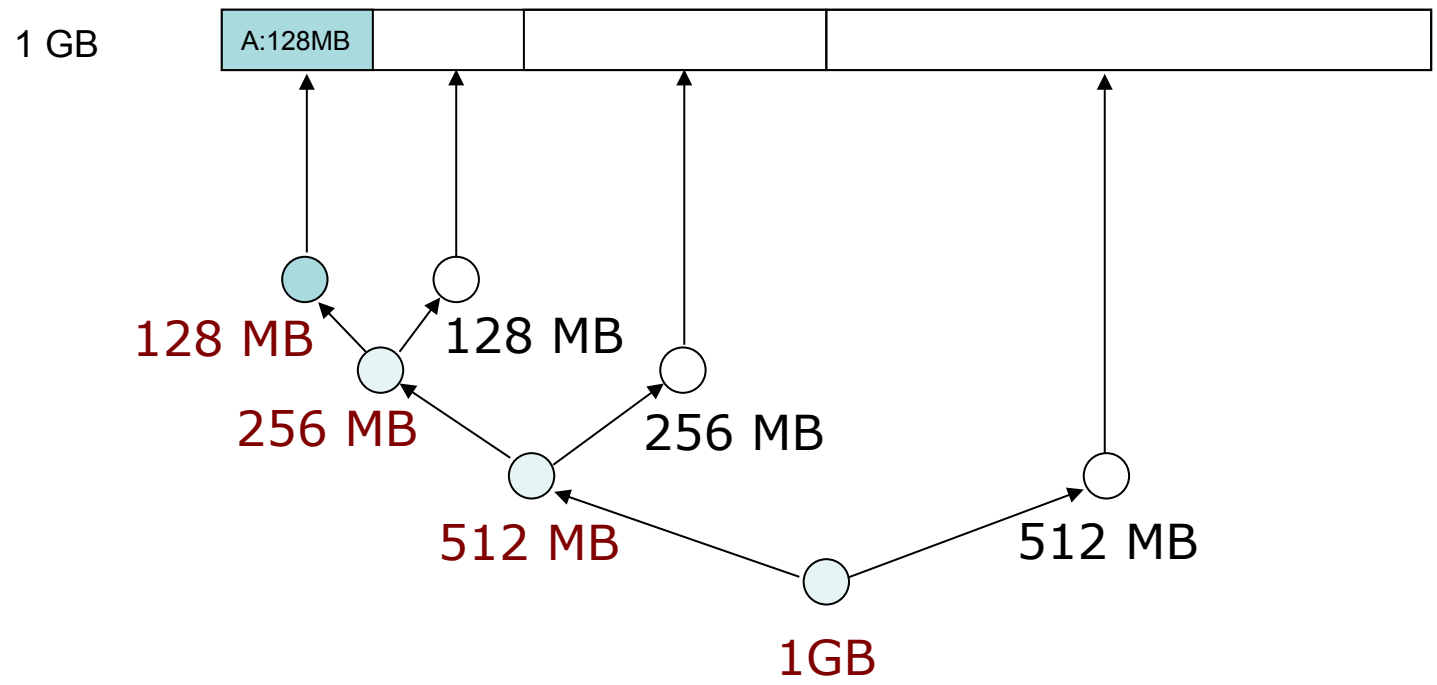
1 GB: 1
512 MB: 0
256 MB: 0
128 MB: 0
64 MB: 0

Gesamter Speicher als 1 Block verfügbar

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

Buddy-System (9)

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



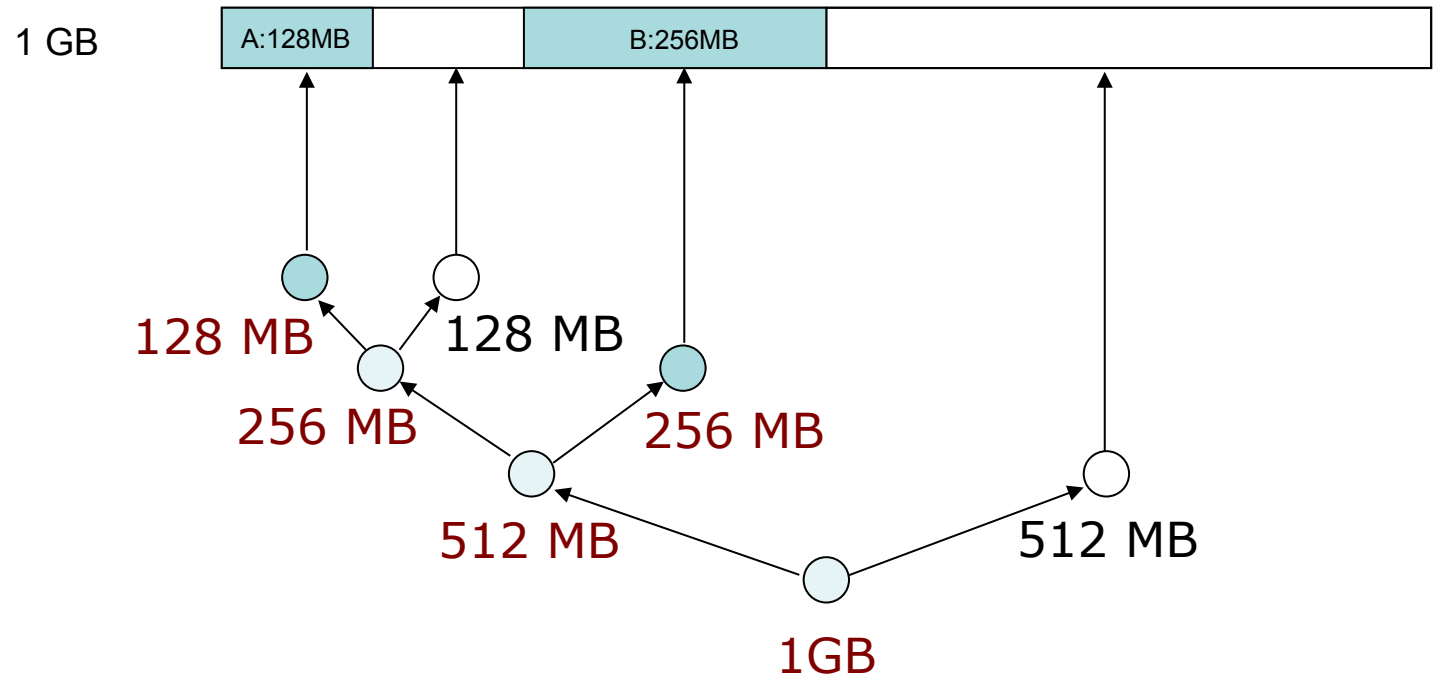
Freie Blöcke:

1 GB: 0
512 MB: 1
256 MB: 1
128 MB: 1
64 MB: 0

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

Buddy-System (10)

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



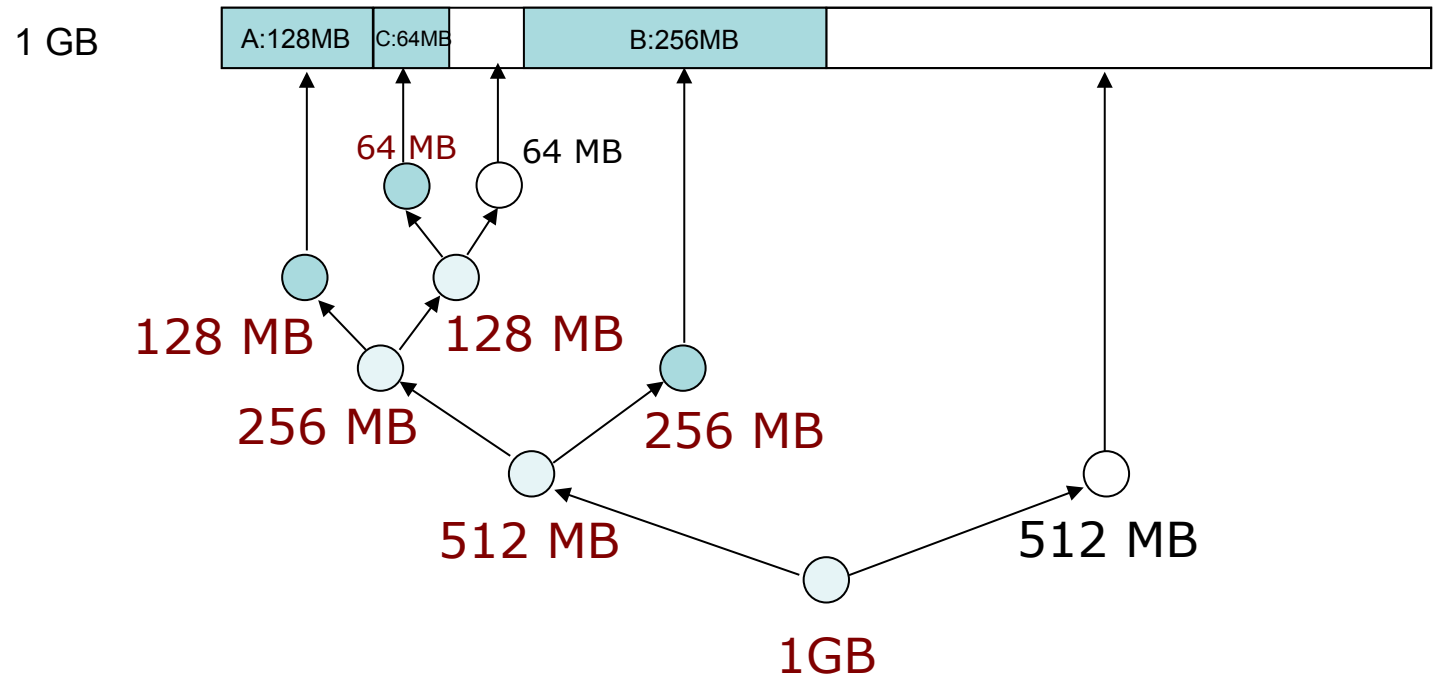
Freie Blöcke:

1 GB: 0
512 MB: 1
256 MB: 0
128 MB: 1
64 MB: 0

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

Buddy-System (11)

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



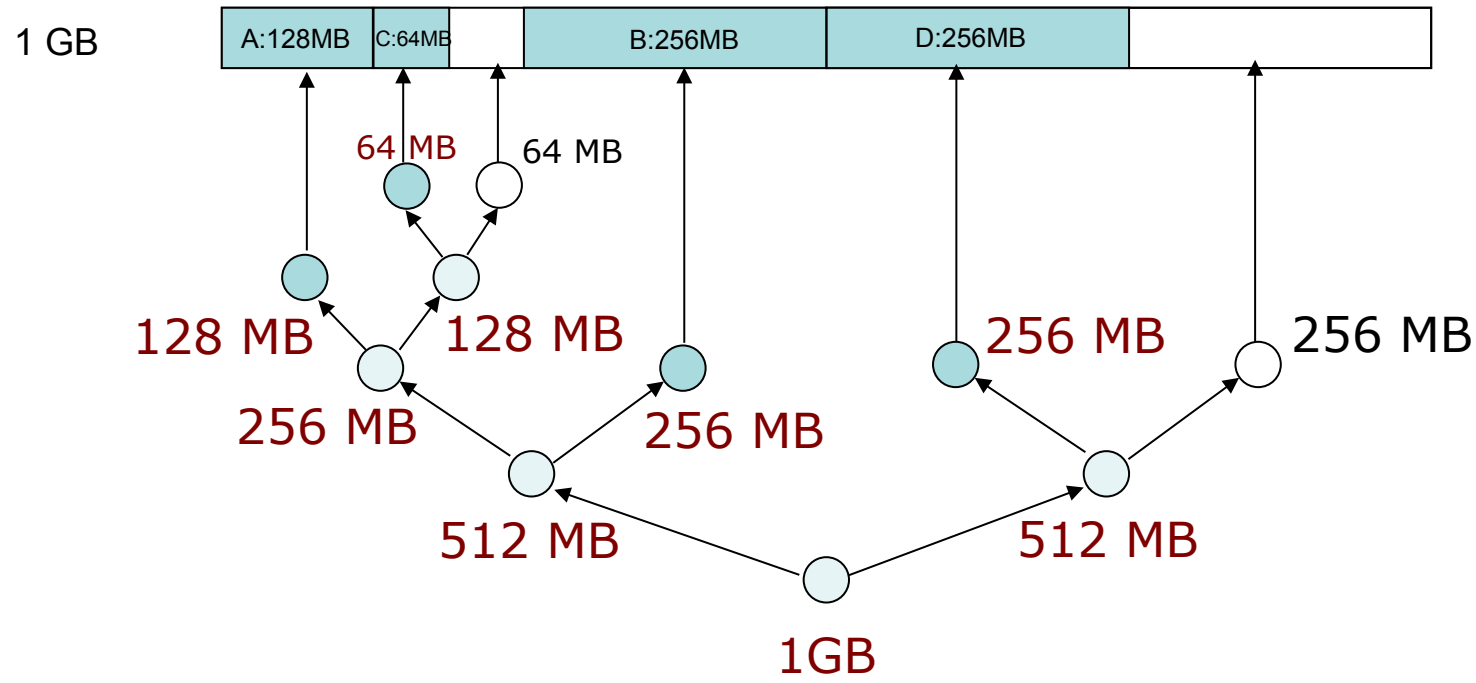
Freie Blöcke:

1 GB: 0
512 MB: 1
256 MB: 0
128 MB: 0
64 MB: 1

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

Buddy-System (12)

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



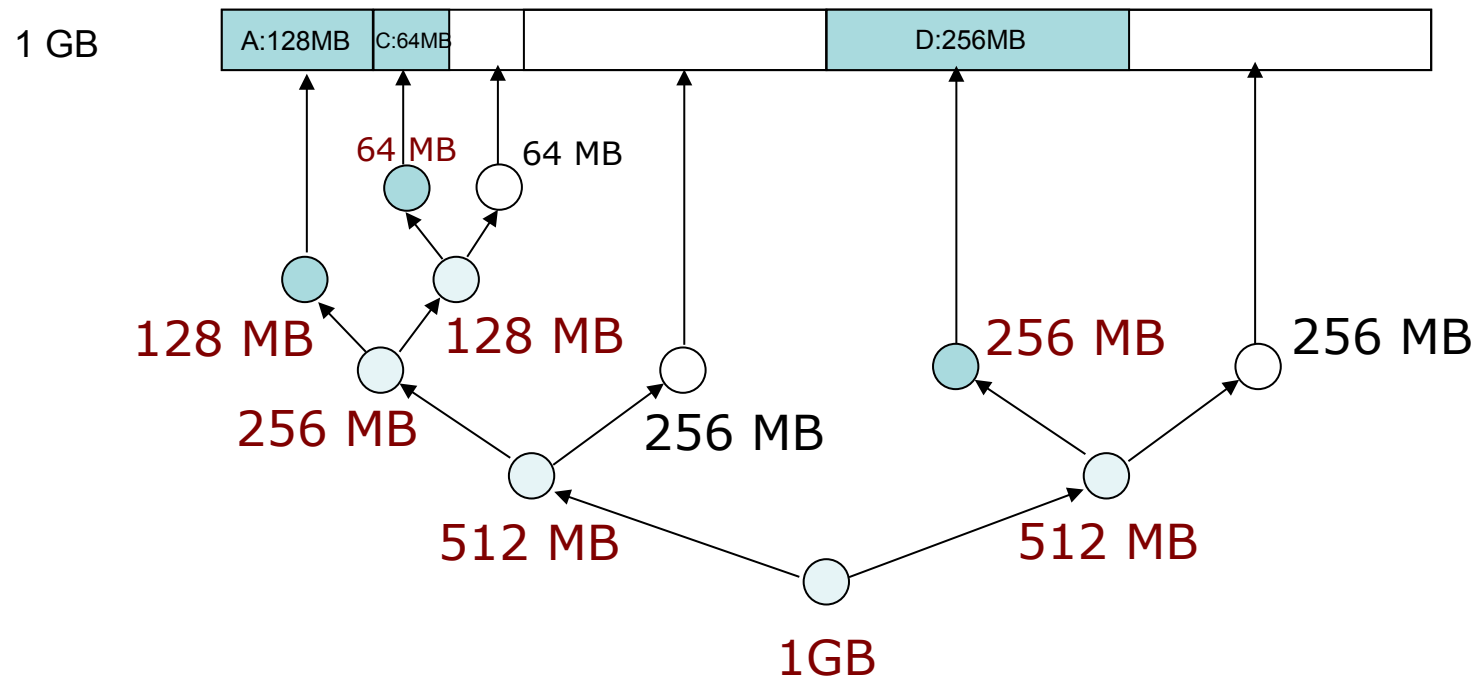
Freie Blöcke:

1 GB: 0
512 MB: 0
256 MB: 1
128 MB: 0
64 MB: 1

Es folgt Freigabe B

Buddy-System (13)

Nach Freigabe B:



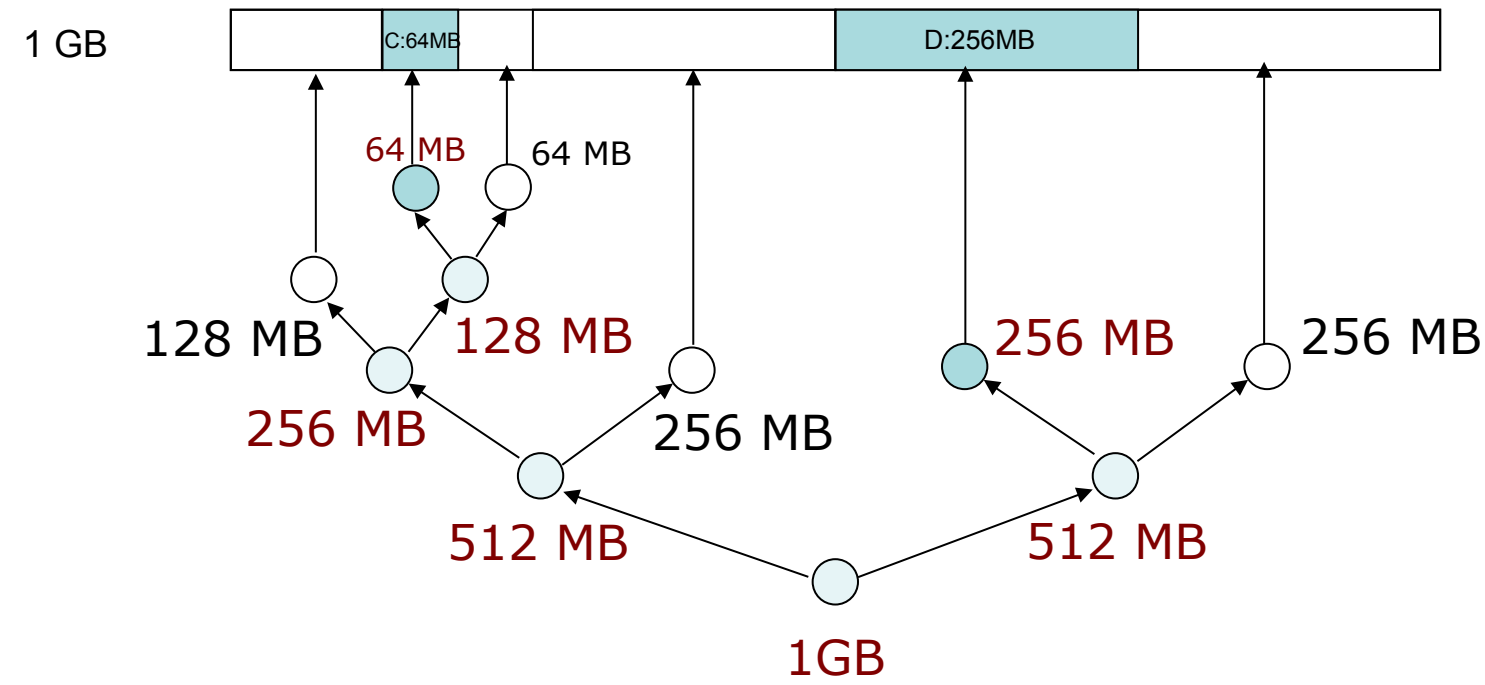
Freie Blöcke:

1 GB: 0
512 MB: 0
256 MB: 2
128 MB: 0
64 MB: 1

Es folgt Freigabe A

Buddy-System (14)

Nach Freigabe A:



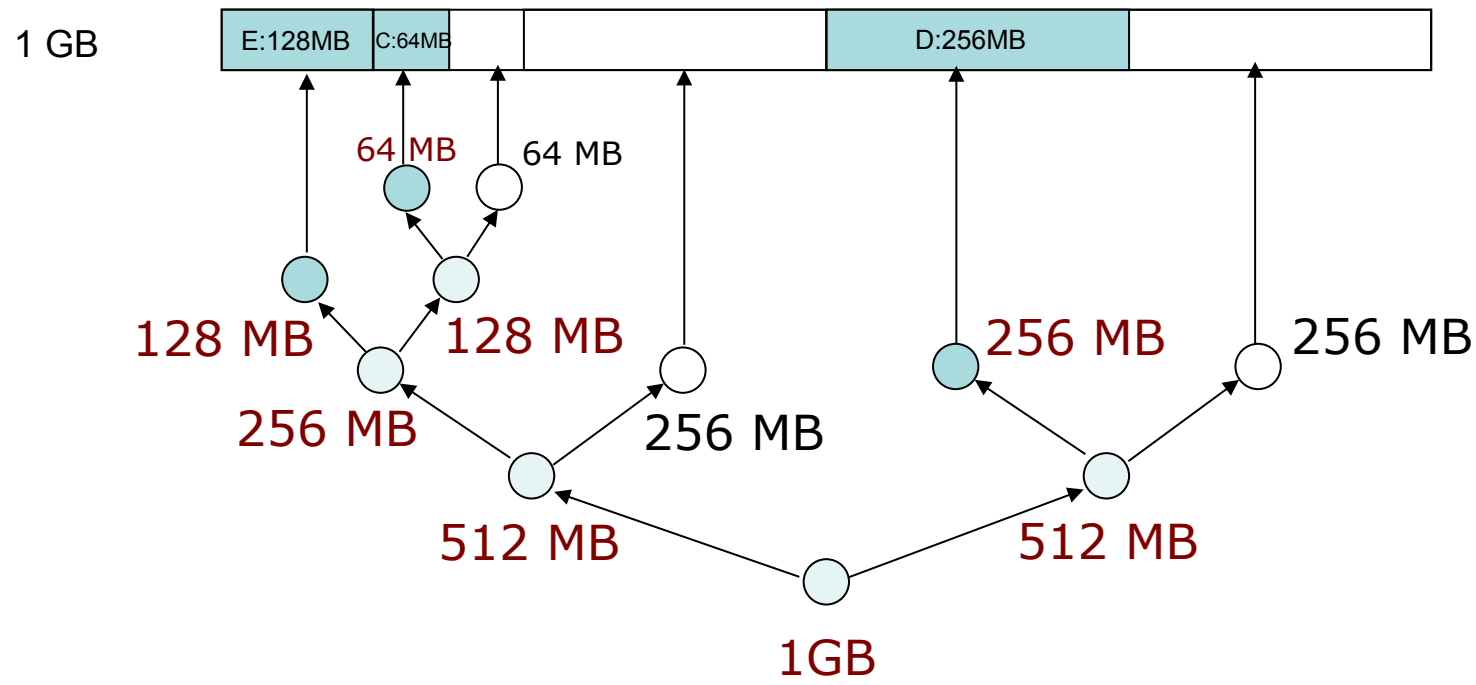
Freie Blöcke:

1 GB: 0
512 MB: 0
256 MB: 2
128 MB: 1
64 MB: 1

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

Buddy-System (15)

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



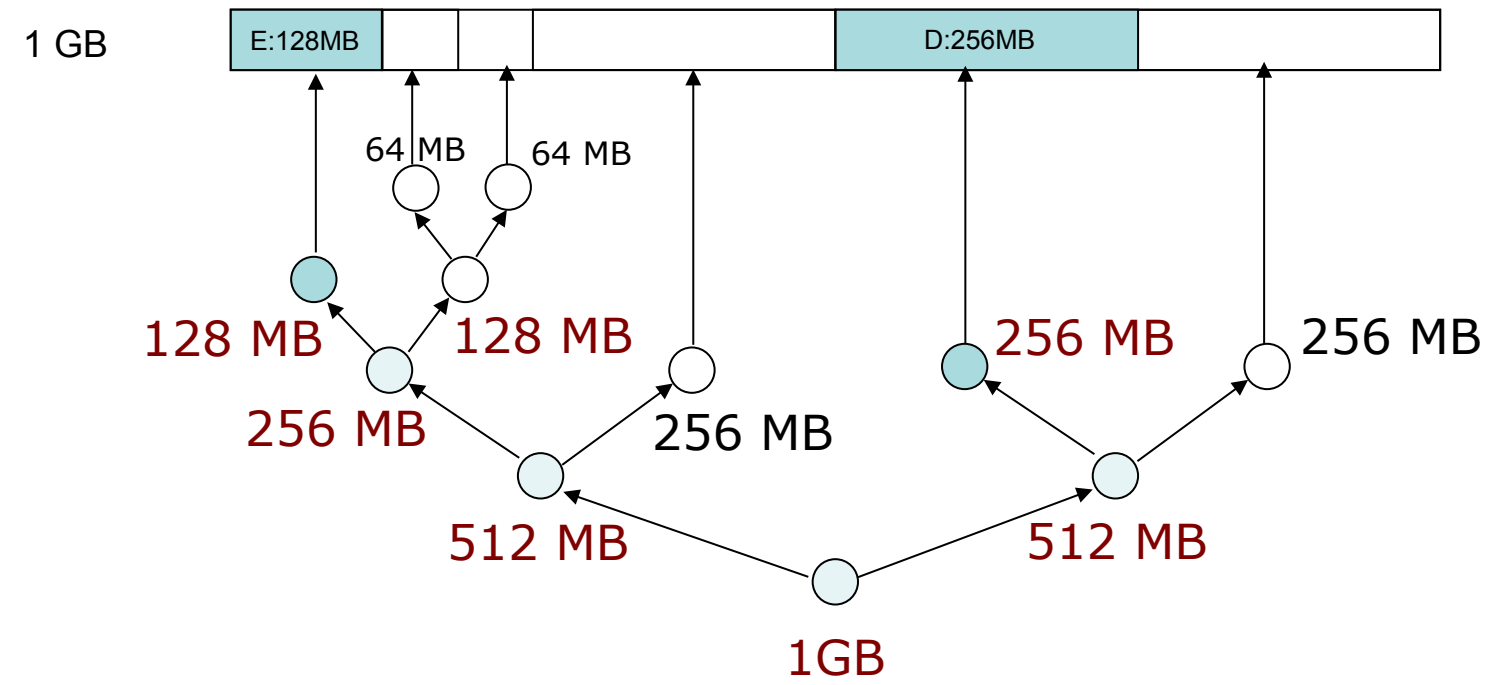
Freie Blöcke:

1 GB: 0
512 MB: 0
256 MB: 2
128 MB: 0
64 MB: 1

Es folgt Freigabe C

Buddy-System (16)

Bei Freigabe C:

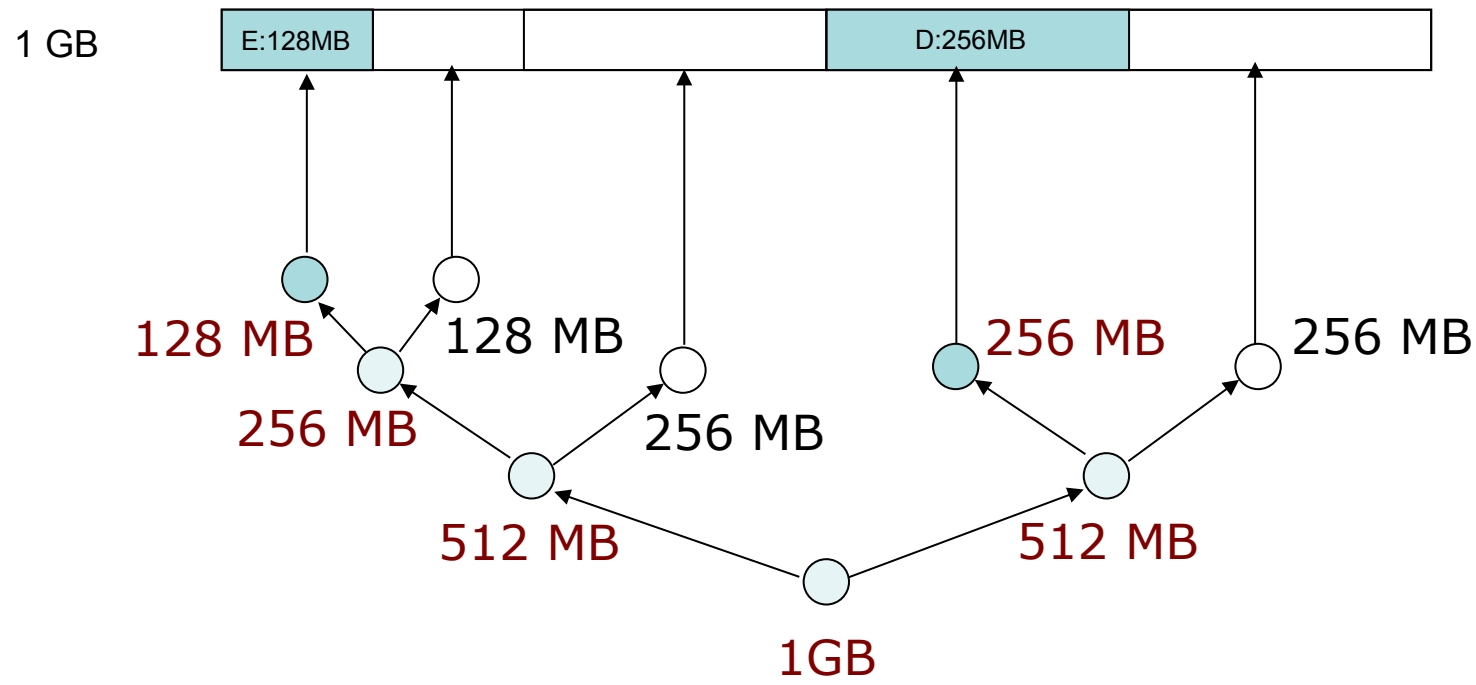


Freie Blöcke:

1 GB: 0
512 MB: 0
256 MB: 2
128 MB: 0
64 MB: 2

Buddy-System (17)

Nach Freigabe C:



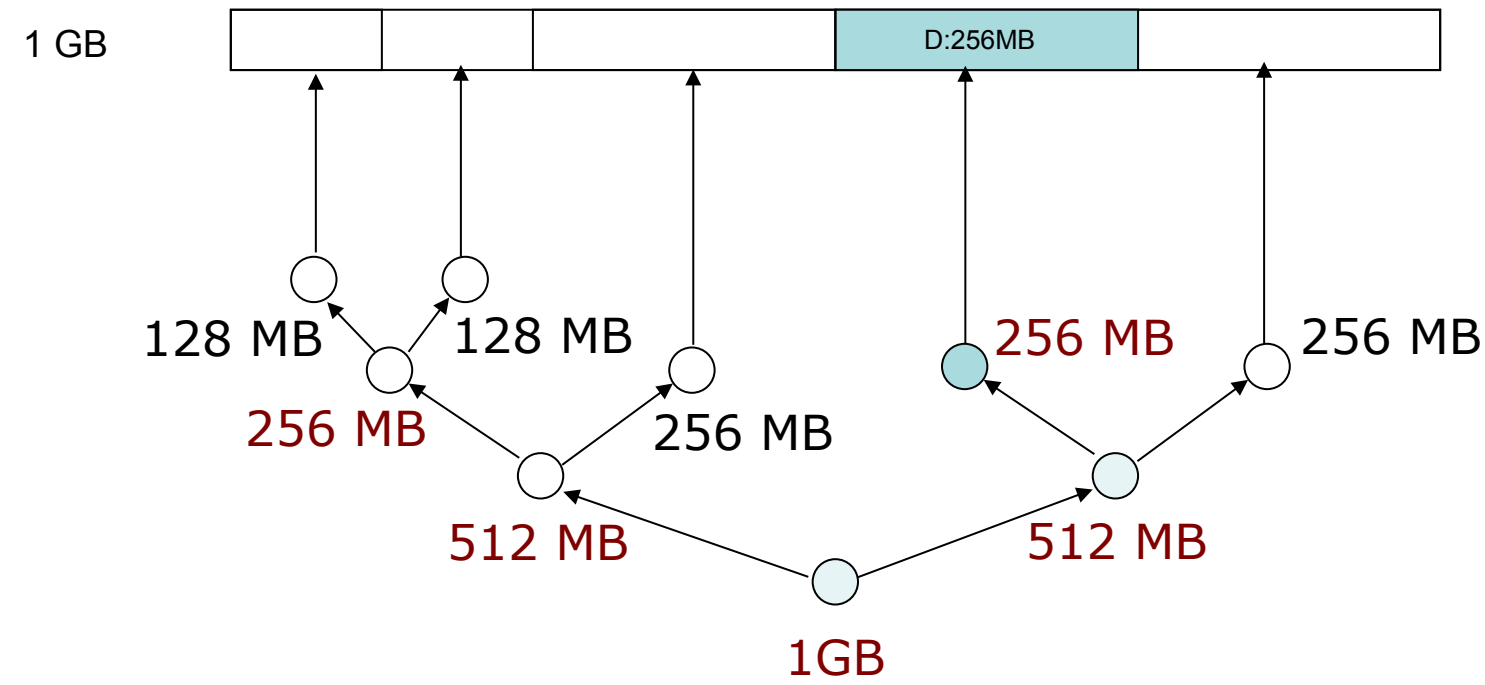
Freie Blöcke:

1 GB: 0
512 MB: 0
256 MB: 2
128 MB: 1
64 MB: 0

Es folgt Freigabe E

Buddy-System (18)

Bei Freigabe E:

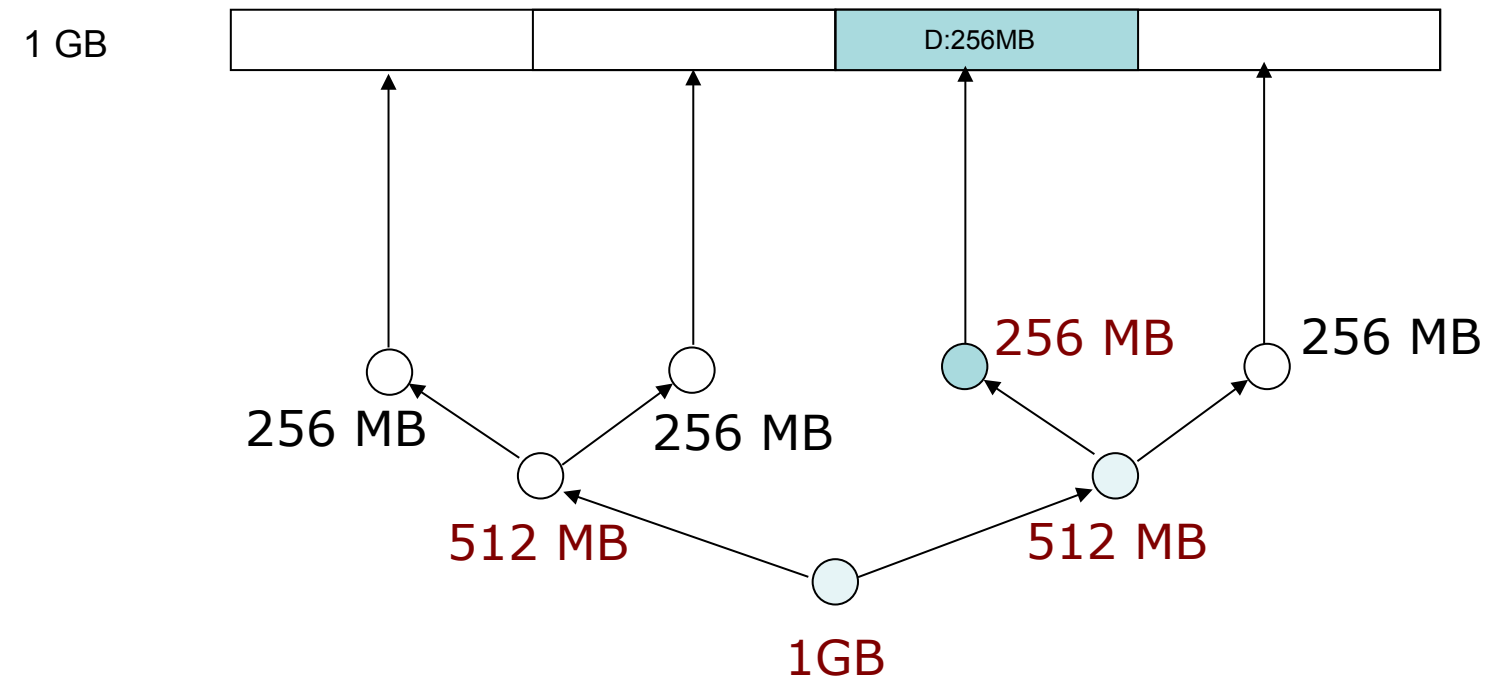


Freie Blöcke:

1 GB: 0
512 MB: 0
256 MB: 2
128 MB: 2
64 MB: 0

Buddy-System (19)

Bei Freigabe E:

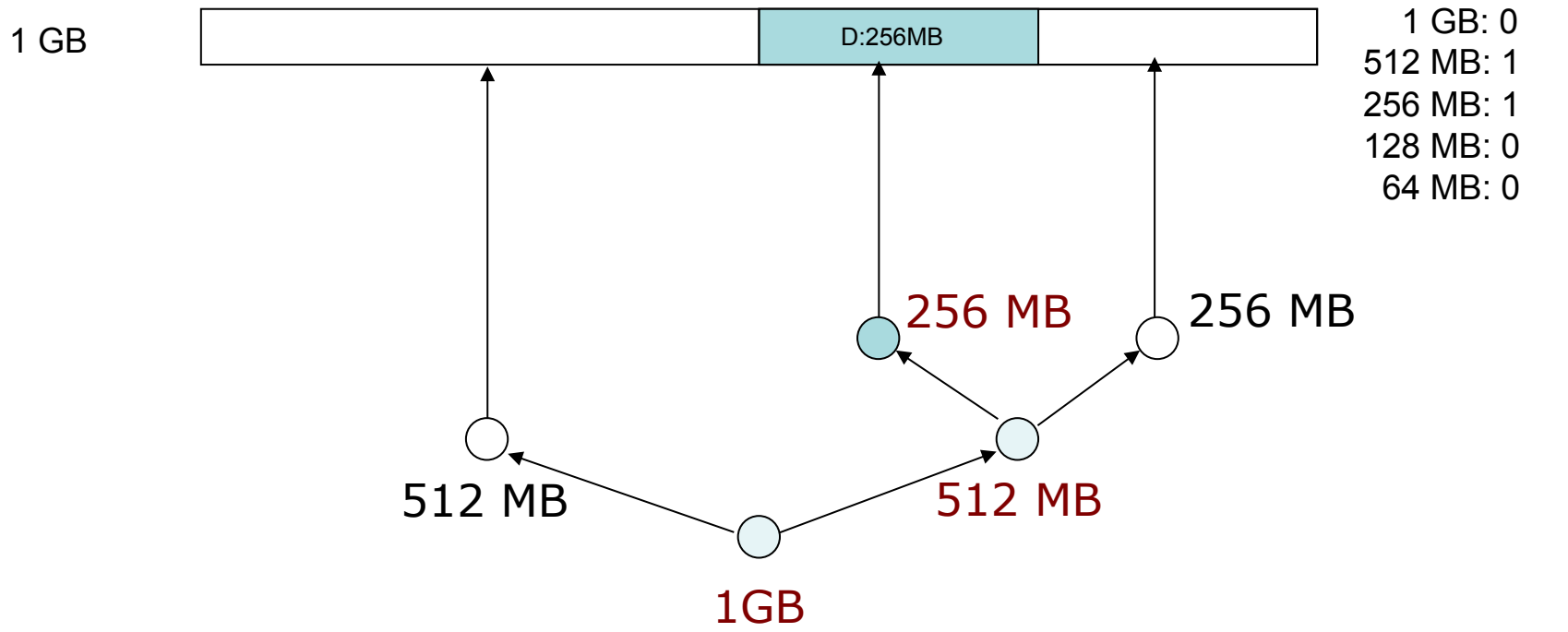


Freie Blöcke:

1 GB: 0
512 MB: 0
256 MB: 3
128 MB: 0
64 MB: 0

Buddy-System (20)

Nach Freigabe E:



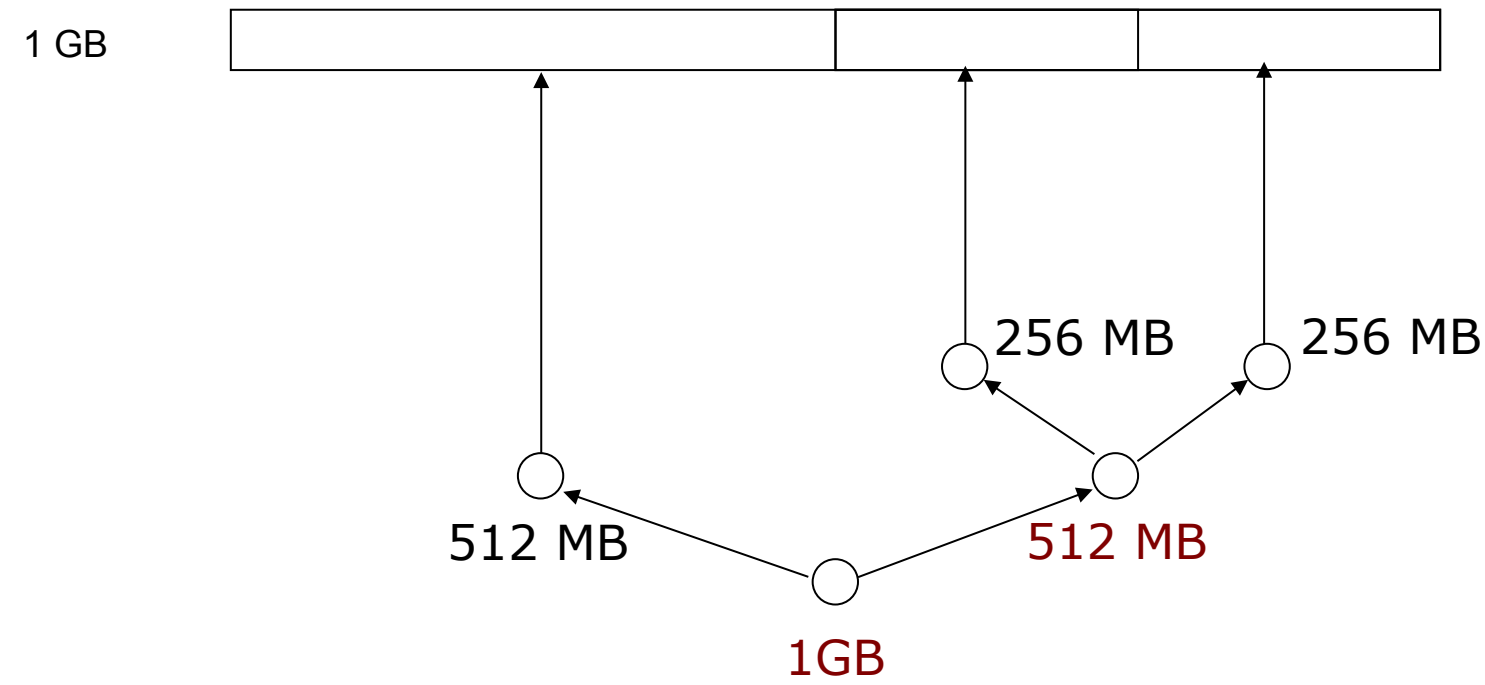
Freie Blöcke:

1 GB: 0
512 MB: 1
256 MB: 1
128 MB: 0
64 MB: 0

Es folgt Freigabe D

Buddy-System (21)

Bei Freigabe D:

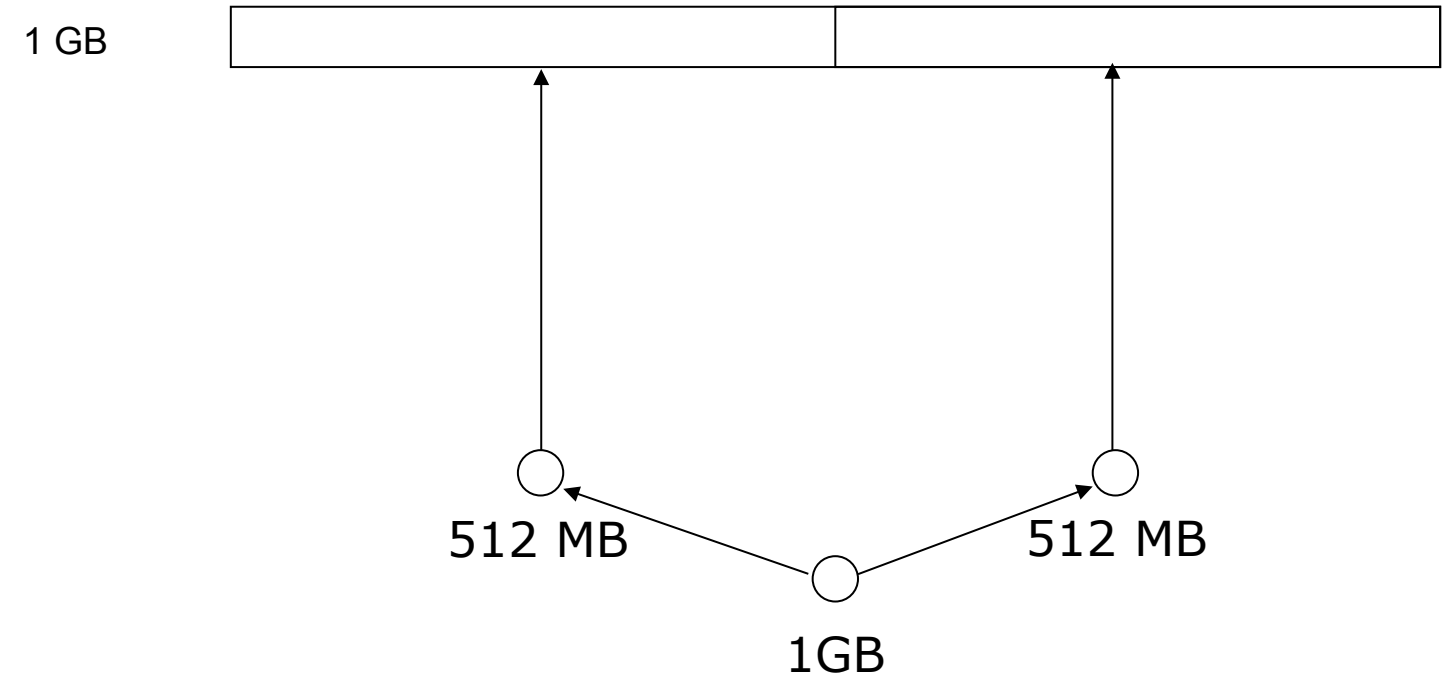


Freie Blöcke:

1 GB: 0
512 MB: 1
256 MB: 2
128 MB: 0
64 MB: 0

Buddy-System (22)

Bei Freigabe D:

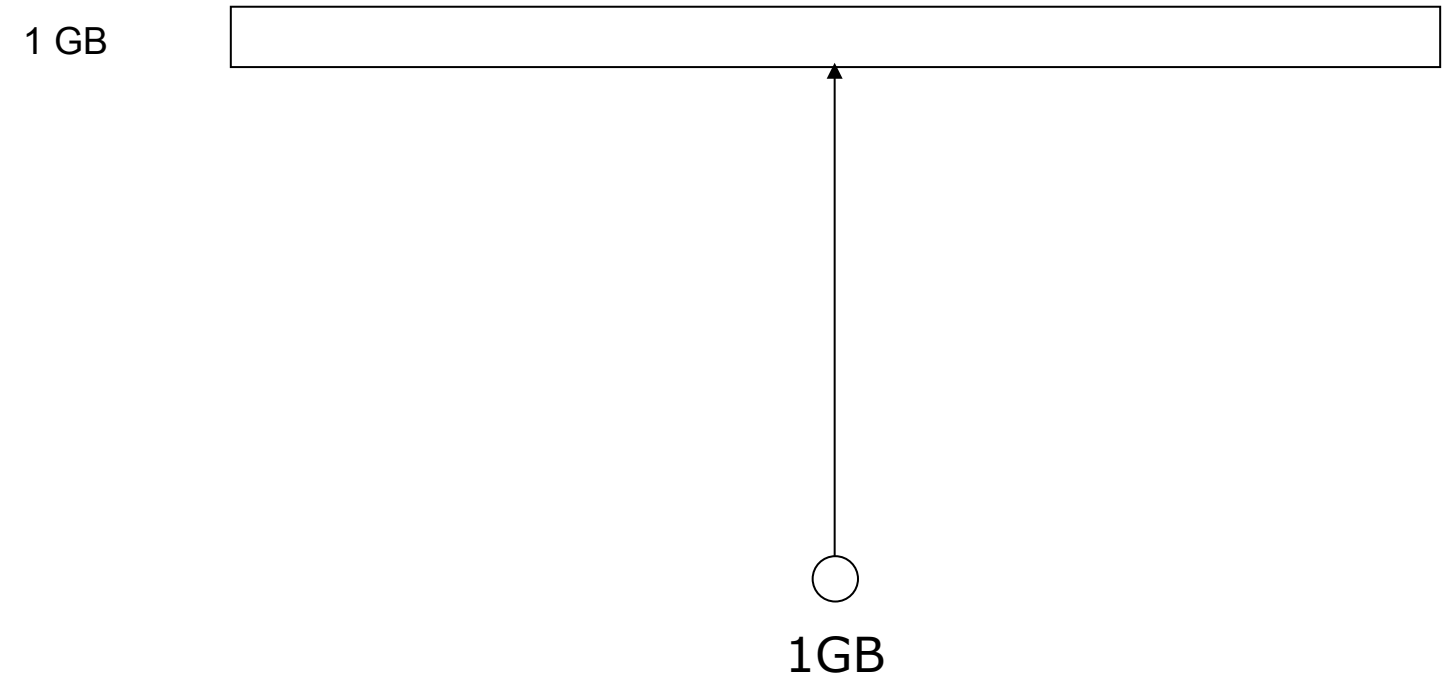


Freie Blöcke:

1 GB: 0
512 MB: 2
256 MB: 0
128 MB: 0
64 MB: 0

Buddy-System (23)

Nach Freigabe D:

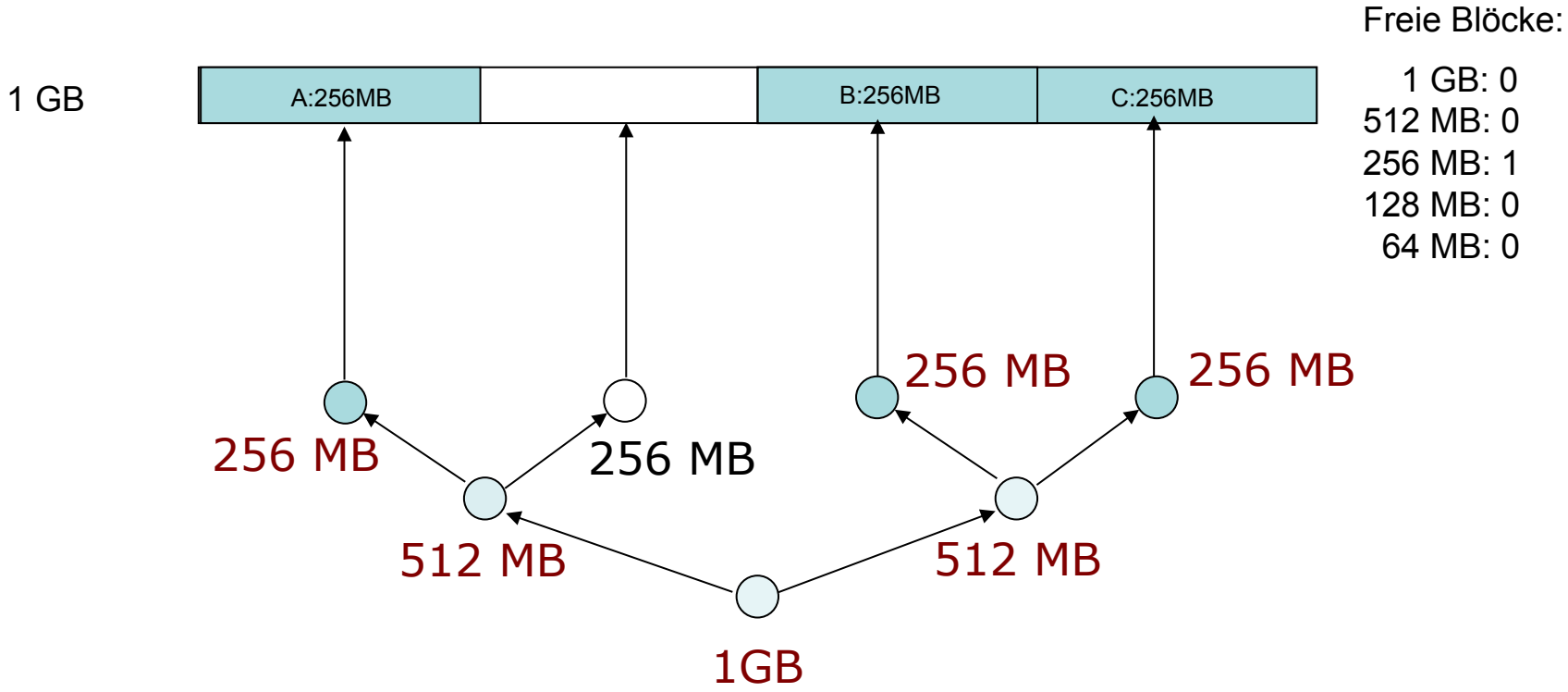


Freie Blöcke:

1 GB: 1
512 MB: 0
256 MB: 0
128 MB: 0
64 MB: 0

Gesamter Speicher wieder als 1 Block verfügbar

Buddy-System (24)



Was passiert bei Freigabe von B?

Buddy-System (25)

- Dynamische Anzahl nicht-ausgelagerter Prozesse
- Effiziente Suche nach freiem Block
- Beschränkte interne Fragmentierung: kleiner als die halbe Größe des resultierenden freien Blockes
- Kommt in parallelen Systemen zum Einsatz und in modifizierter Form bei der Speicherzuteilung an die Hauptprozesse des Betriebssystems

Relokation (1)

- Nach Aus- und Wiedereinlagern von Prozessen werden ihnen andere Speicherbereiche zugewiesen
- Ebenfalls ändert sich der Speicherbereich, wenn Prozesse im Rahmen von Defragmentierung verschoben werden
- Absolute Sprungbefehle und Datenzugriffs-befehle müssen weiterhin funktionieren

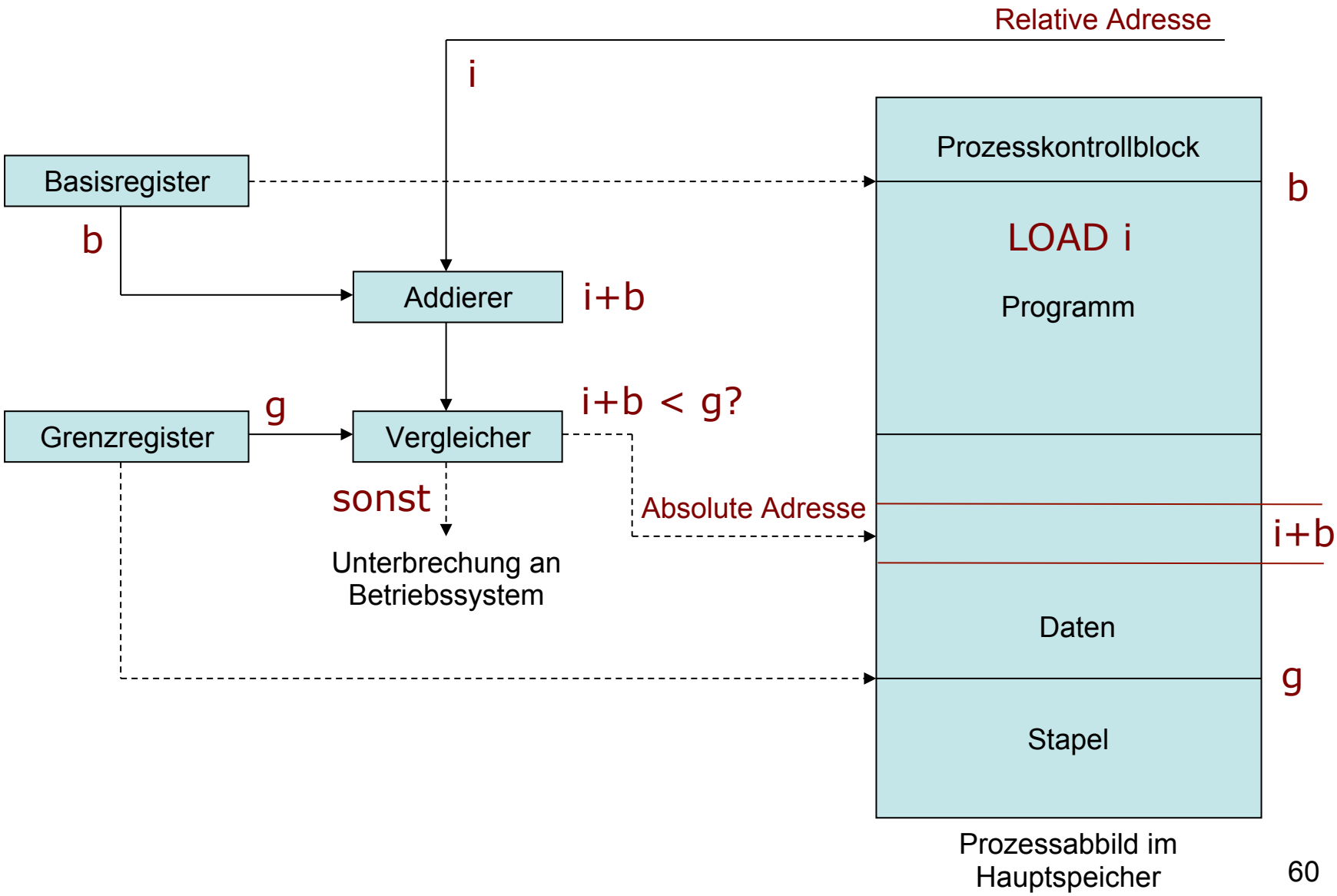
Relokation (2)

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

Relokation (3)

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

Relokation (4)



Relokation (5)

- Relative Adresse wird um den Wert erhöht, der sich im Basisregister befindet (dadurch: absolute Adresse)
- Resultierende Adresse wird mit dem Wert im Grenzregister verglichen
- Befehlsausführung nur, wenn die Adresse innerhalb der Grenzen liegt, sonst Interrupt
- Schutzmechanismus: Adressräume sind sicher vor Zugriffen anderer Prozesse

Einfaches Paging (1)

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

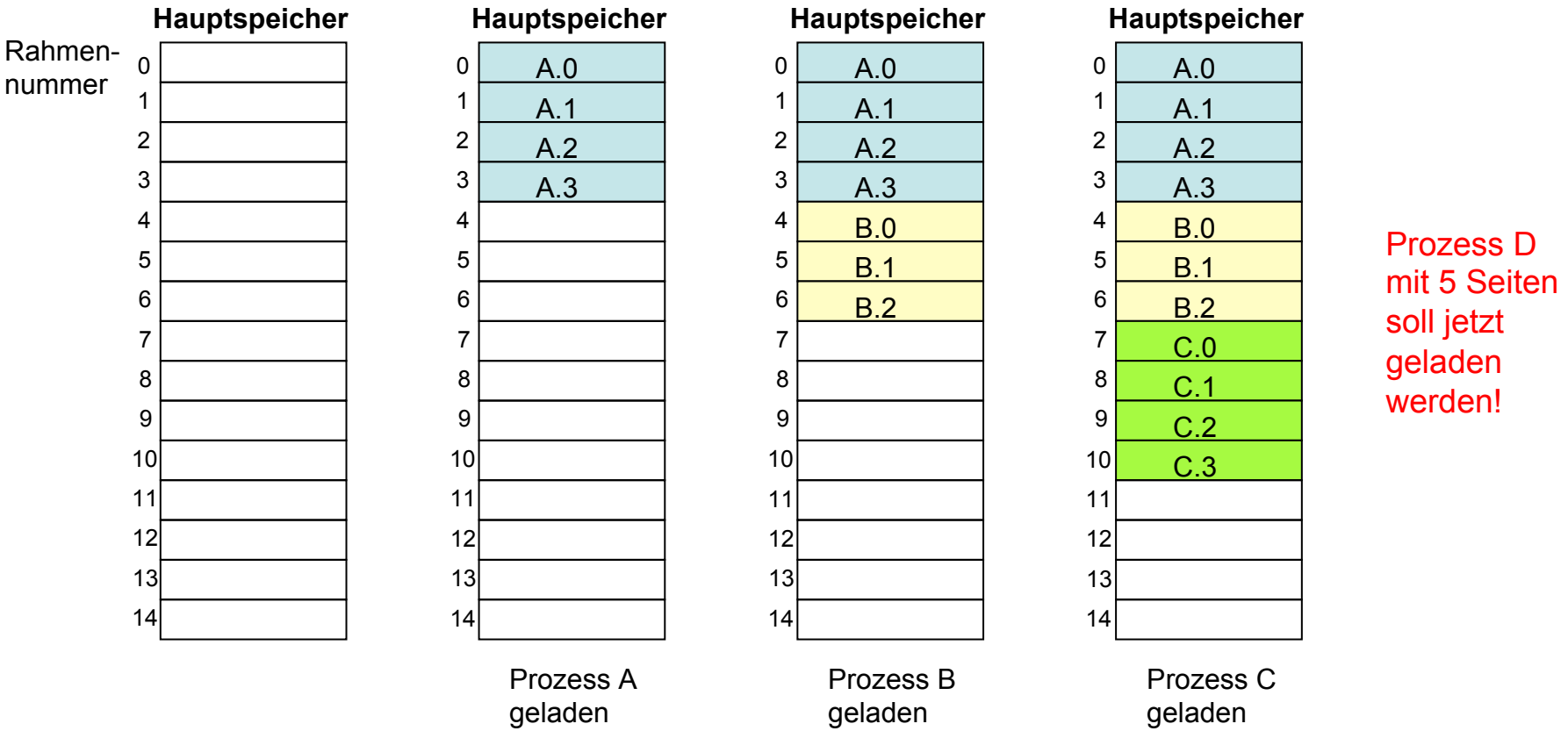
Einfaches Paging (2)

- BS verwaltet Seitentabelle für jeden Prozess: Anzeige, wo sich die einzelnen Seiten befinden (in welchem Seitenrahmen)
- Zuordnung von Seiten zu Seitenrahmen beim Laden von Prozessen
- Innerhalb Programm: Logische Adresse der Form „Seitennummer, Offset“
- Durch Seitentabelle: Übersetzung der logischen Adressen in Nummern von Seitenrahmen im physikalischen Speicher „Rahmennummer, Offset“

Einfaches Paging (3)

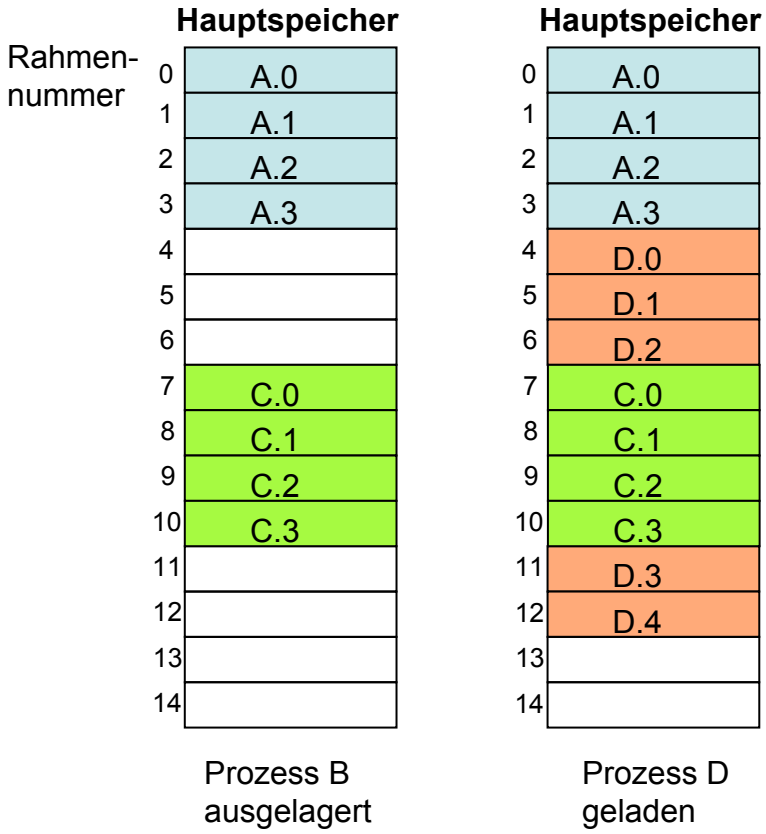
- Prozessorhardware übersetzt logische Adresse in physikalische Adresse
- Interne Fragmentierung nur bei letzter Seite eines Prozesses
- Keine externe Fragmentierung

Einfaches Paging (4)

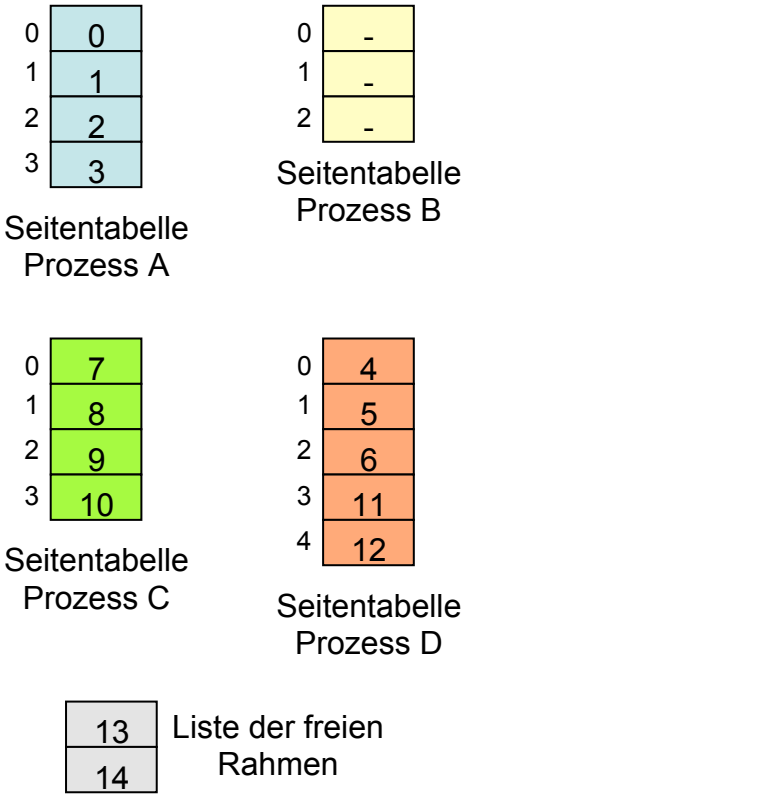


Prozess D mit 5 Seiten soll jetzt geladen werden!

Einfaches Paging (5)



Datenstrukturen zum aktuellen Zeitpunkt:



Einfaches Paging (6)

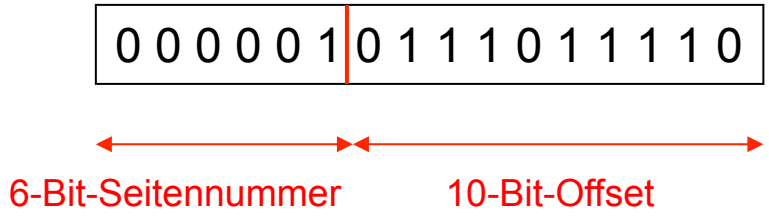
- Einfaches Paging sehr ähnlich dem Konzept des statischen Partitionierens
- Beim Paging sind jedoch die Partitionen relativ klein
- Ein Programm kann mehrere Partitionen belegen, die nicht aneinander angrenzen müssen

Einfaches Paging (7)

- Berechnung von physikalischen Adressen aus logischen Adressen:
  - Länge der Seitengröße (und Rahmengröße) ist eine Zweierpotenz
  - Logische Adresse im Programm besteht aus Seitennummer und Offset
- Absolute Adresse wird durch Hardware auf Grundlage der Seitentabelle des Prozesses berechnet

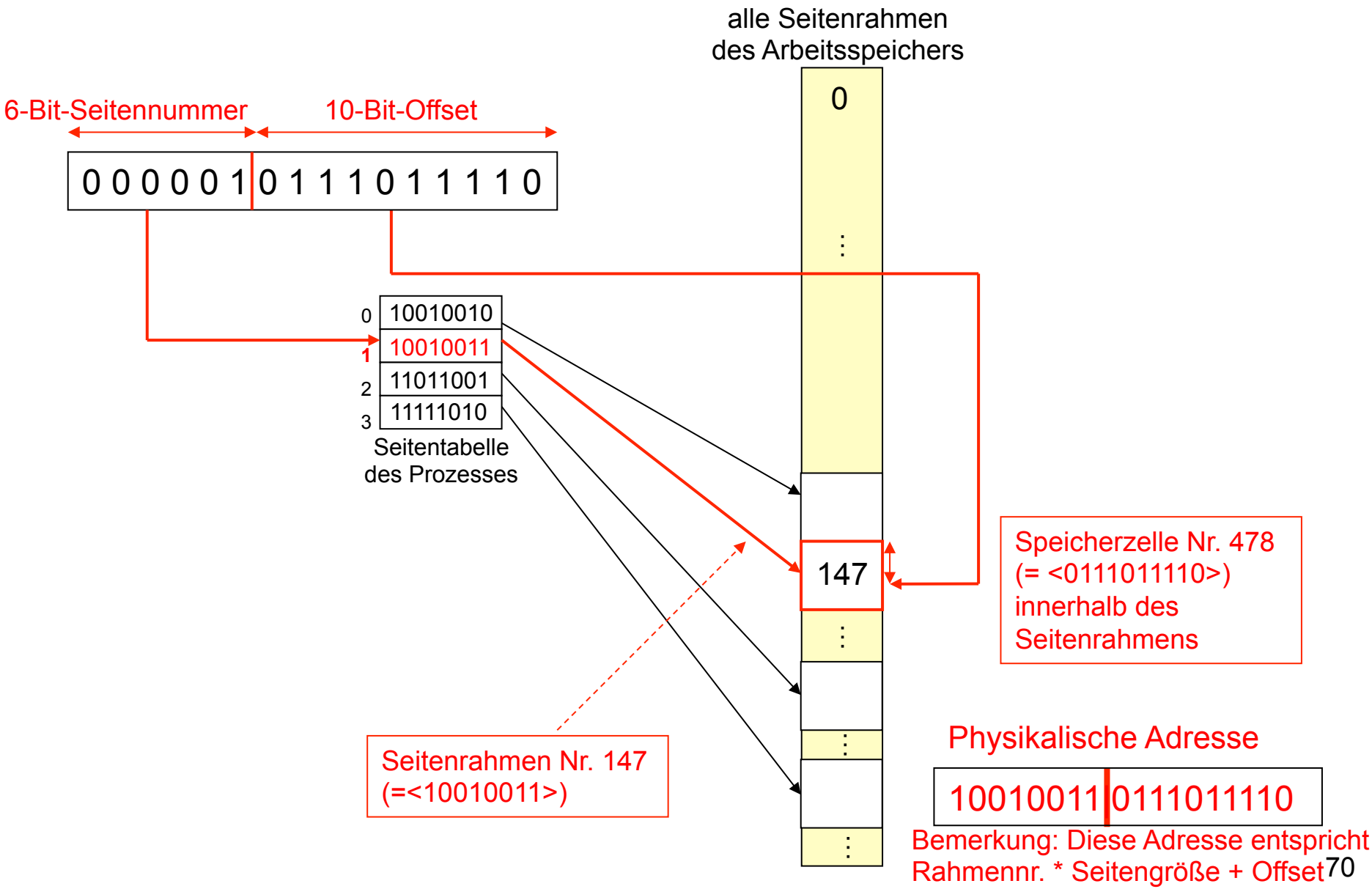
Einfaches Paging (8)

- Beispiel: Logische Adresse der Länge 16 Bit
- Seitengröße 1 Kbyte = 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 Haupt-speicher 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 (vorher freie) Seiten-rahmen des Speichers
- Interne Fragmentierung nur bei letzter Seite eines Prozesses

Einfaches Paging (11)

- Bei Entfernen eines Prozesses aus dem Speicher: Über Seitentabelle hat man die Information, welche Seitenrahmen dem Prozess gehören
- Füge diese Rahmen zur Liste der freien Rahmen hinzu
- Keine zusätzlichen Datenstrukturen des Betriebssystems werden benötigt

Paging mit virtuellem Speicher (1)

Grundidee:

- Lagere Teile der Daten von Prozessen ein- bzw. aus anstelle kompletter Prozess
- Programm kann auch weiter ausgeführt werden, auch wenn nur die aktuell benötigten Informationen (Code und Daten) im Speicher sind
- Wird auf Informationen zugegriffen, die ausgelagert sind, so müssen diese nachgeladen werden

Paging mit virtuellem Speicher (2)

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

Paging mit virtuellem Speicher (3)

- Nachteile:
  - Bei Zugriff auf Code/Daten, die nicht im Hauptspeicher vorhanden sind, muss das Betriebssystem die entsprechenden Seiten nachladen
  - Dabei müssen evtl. andere Seiten ausgelagert werden, um Platz zu schaffen
  - Dadurch wird das System langsamer

Lokalität (1)

Kann das überhaupt effizient funktionieren?

- Ja, Grund: 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

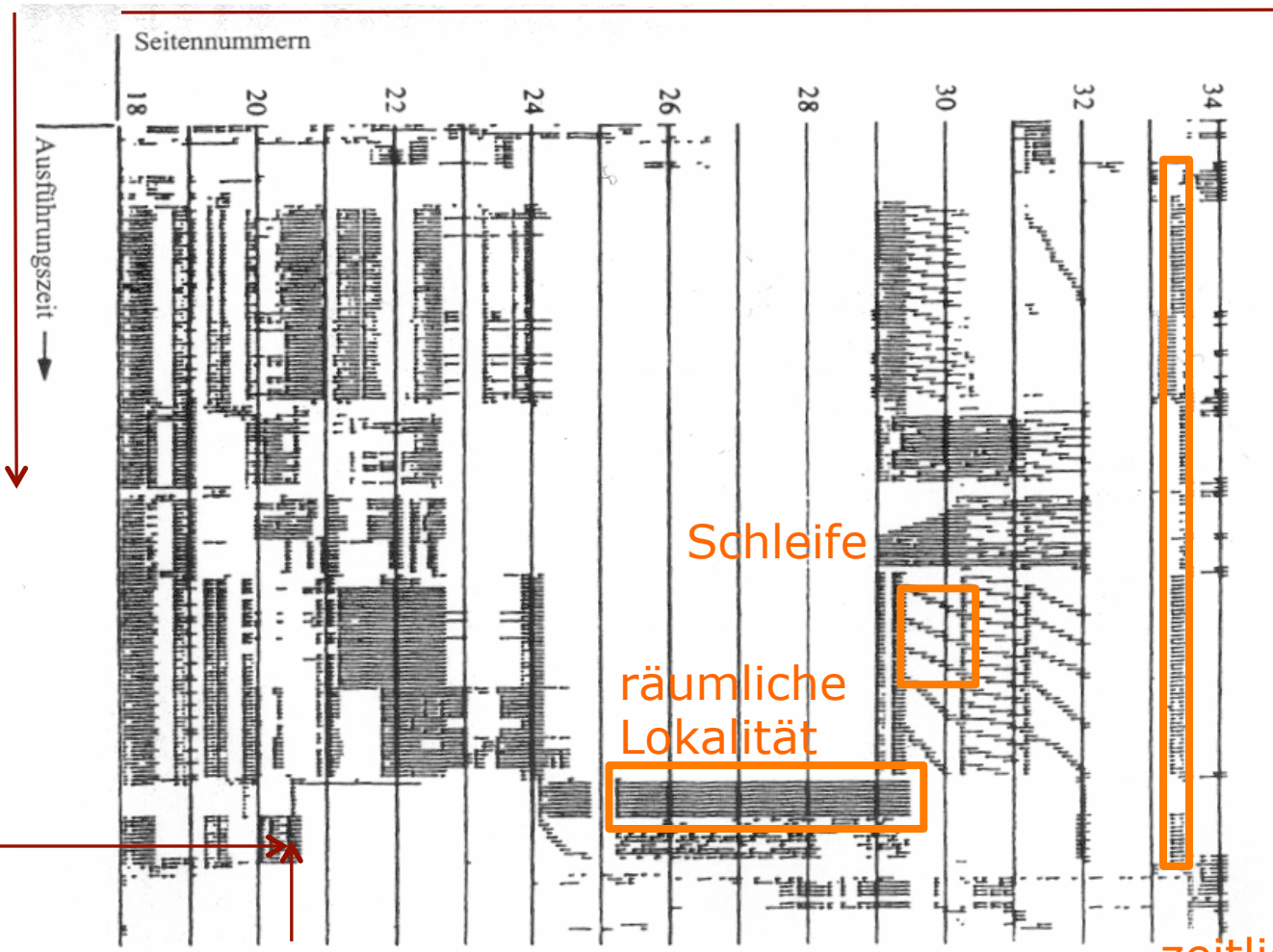
Lokalität (2)

- Die Abarbeitung während kürzerer Zeit bewegt sich häufig in engen Adress-bereichen
- Zeitliche Lokalität:
  - Abarbeitung von Schleifen: gleiche Befehle
  - In zeitlich engem Abstand Zugriff auf gleiche Daten
- Räumliche Lokalität:
  - Zugriffe auf benachbarte Daten bei sequentieller Abarbeitung von Programmen
  - Lage von zusammenhängenden Daten

Lokalität (3)

zunehmende Seitennummern eines Prozesses

CPU-Zeit



zur Zeit t

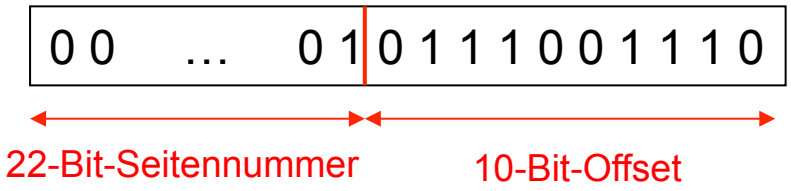
wird zugegriffen
auf Seite s

Lokalität (4)

- Paging mit virtuellem Speicher ist nur dann effizient, wenn Lokalität gegeben ist
- Falls dies nicht der Fall ist, resultieren:
  - Ständiges Aus- und Einlagern von Seiten zwischen Hauptspeicher und Festplatte („Thrashing“)
  - Der Prozessor ist mehr mit Ein- und Auslagern anstatt Ausführen von Befehlen beschäftigt
- Um Thrashing zu vermeiden, versucht das BS zu vorherzusagen, welche Seiten in naher Zukunft nicht benötigt werden

Technische Realisierung (1)

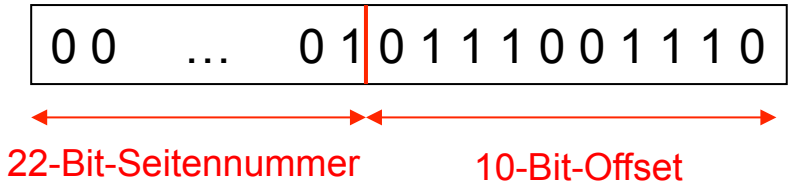
- Einfachstes Modell: Prozess (Daten+Code) befindet sich im Hintergrundspeicher
- Bei teilweise eingelagerten Prozessen: Teile im Hauptspeicher (zusätzlich)
- Logische Adressen überdecken kompletten virtuellen Adressraum
- Wie bei einfachem Paging: Trennung der logischen Adressen in Seitennummer und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



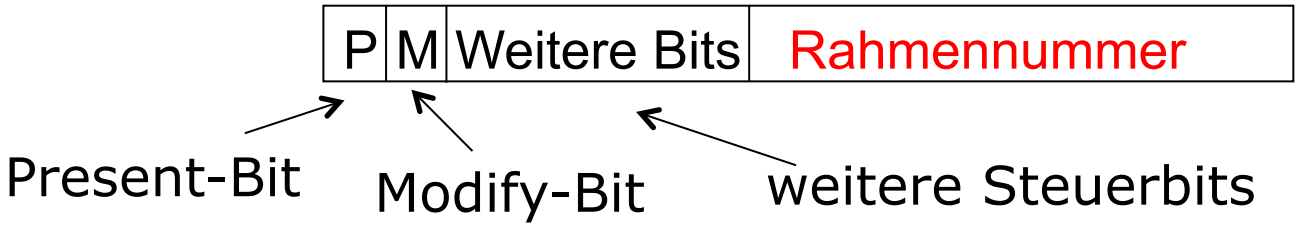
Bsp.: 32-Bit-Adresse
1 KB Seitengröße

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:



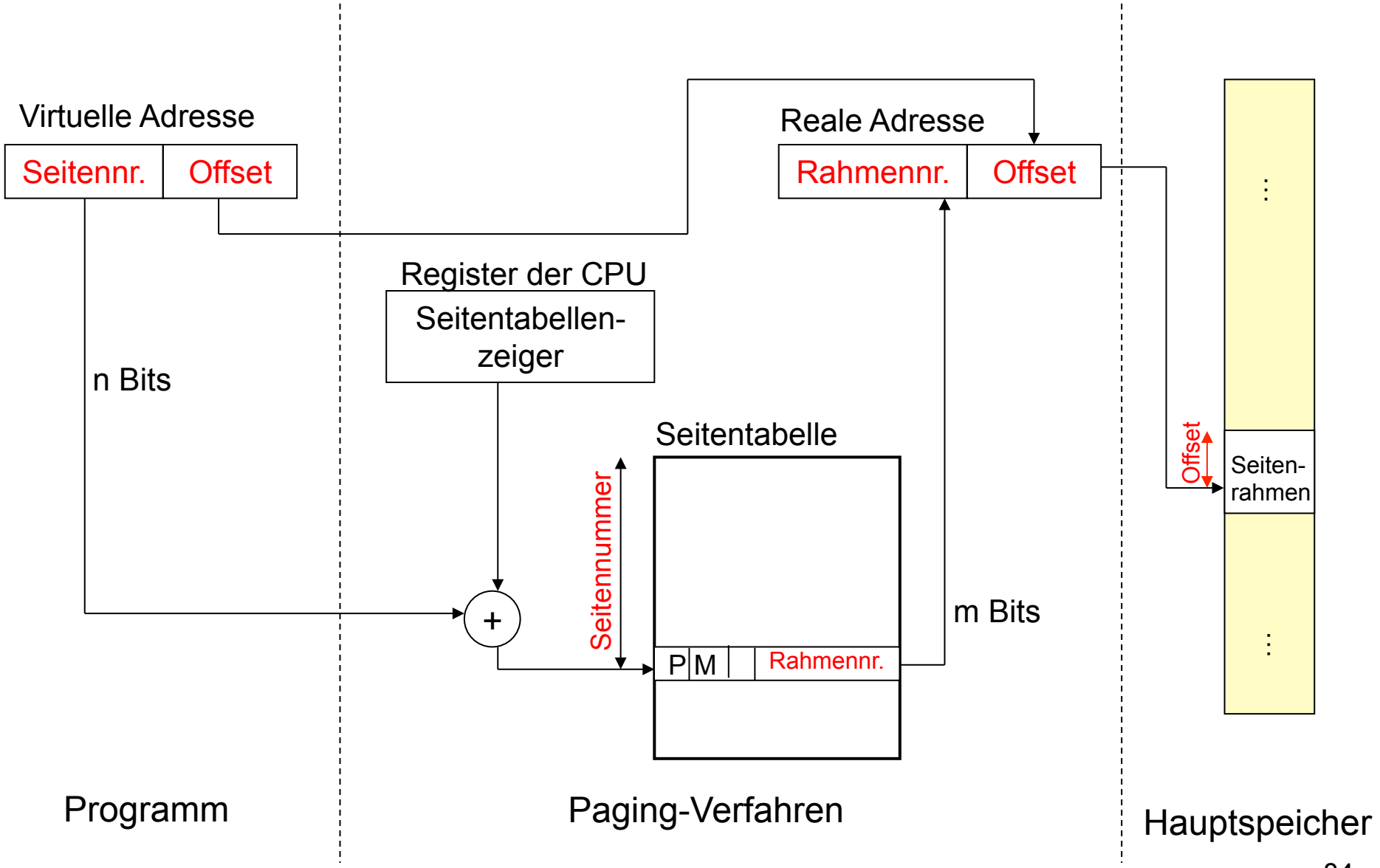
- Seitentabelleneintrag:



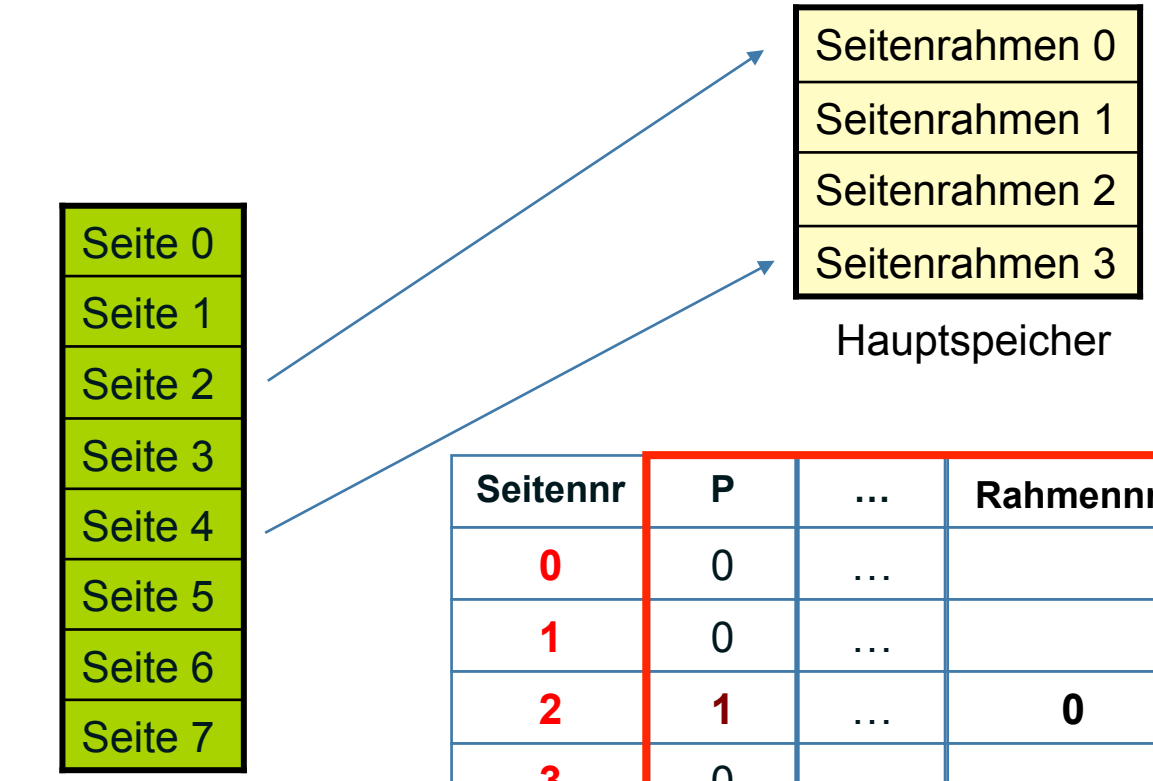
Technische Realisierung (3)

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

Adressumsetzung



Seitentabelle



Seitennr	P	…	Rahmennr.
0	0	…	
1	0	…	
2	1	…	0
3	0	…	
4	1	…	3
5	0	…	
6	0	…	
7	0	…	

Seitentabelle des Prozesses im Hauptspeicher

Was passiert z.B. bei Zugriff auf Seite 0?

Seitenfehler (1)

- Bei Zugriff auf Seite, die nicht im Haupt-speicher 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)

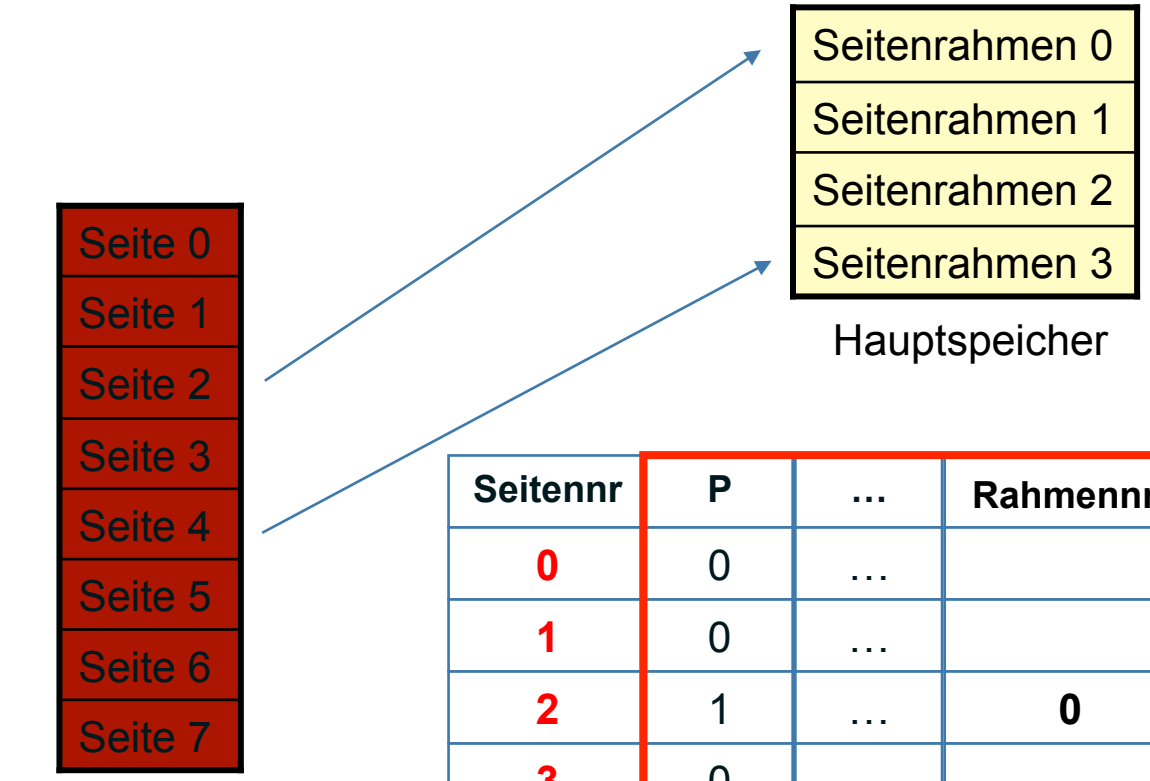
- Betriebssystem lädt die entsprechende Seite von der Festplatte in einen freien Rahmen
- Falls kein Rahmen frei: Vorheriges Verdrängen der Daten eines belegten Rahmens (beachte dabei Modify-Bit)
- Aktualisierung der Seitentabelleneinträge (Present-Bit und Rahmennummer)
- Danach wird unterbrochenes Programm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

Seitenfehler (3)

Welche Informationen benötigt das Betriebssystem zum Einlagern von Seiten?

- Abbildung Seitennummer auf Festplatten-adresse, um die gesuchte Seite auf der Festplatte zu finden
- Liste freier Seitenrahmen

Seitenfehler (4)



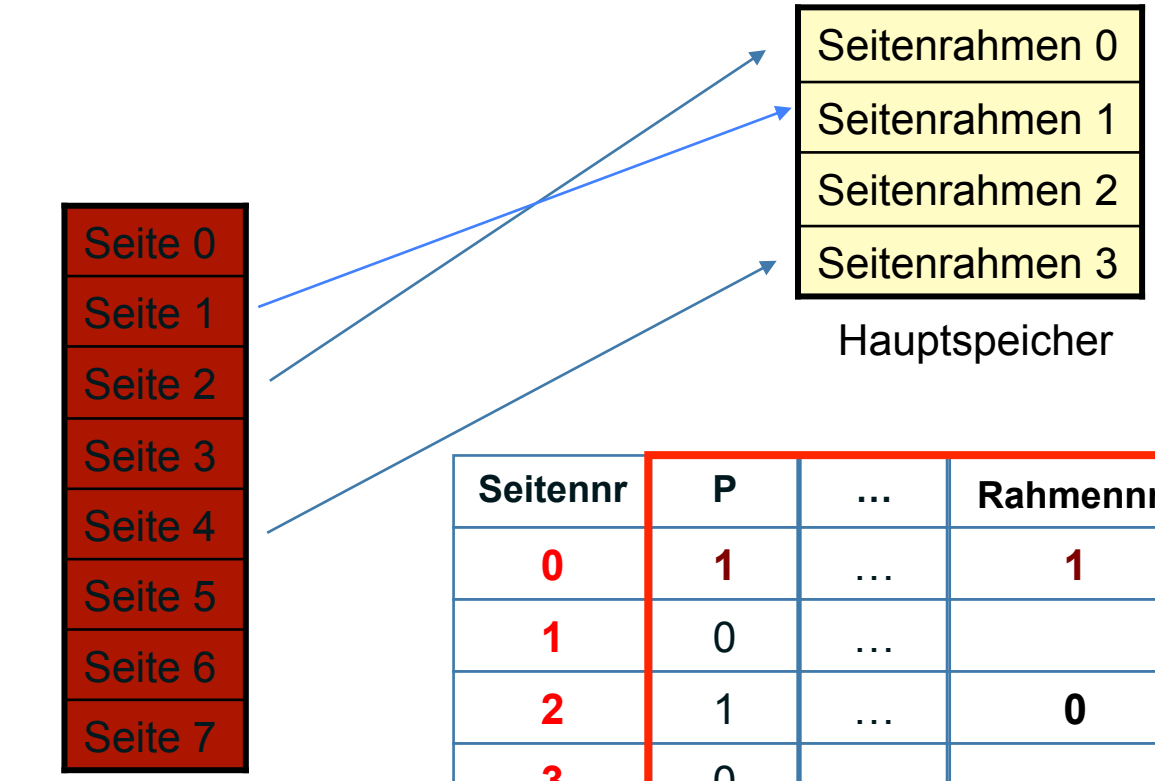
Diskblock-Deskriptor

Seitennr	P	…	Rahmennr.
0	0	…	
1	0	…	
2	1	…	0
3	0	…	
4	1	…	3
5	0	…	
6	0	…	
7	0	…	

Festplatten-Adresse
A
D
B
X
Y
C
E
F

Seitentabelle des Prozesses im Hauptspeicher

Seitenfehler (4)



Diskblock-Deskriptor

Seitennr	P	…	Rahmennr.
0	1	…	1
1	0	…	
2	1	…	0
3	0	…	
4	1	…	3
5	0	…	
6	0	…	
7	0	…	

Festplatten-Adresse
A
D
B
X
Y
C
E
F

Seitentabelle des Prozesses im Hauptspeicher

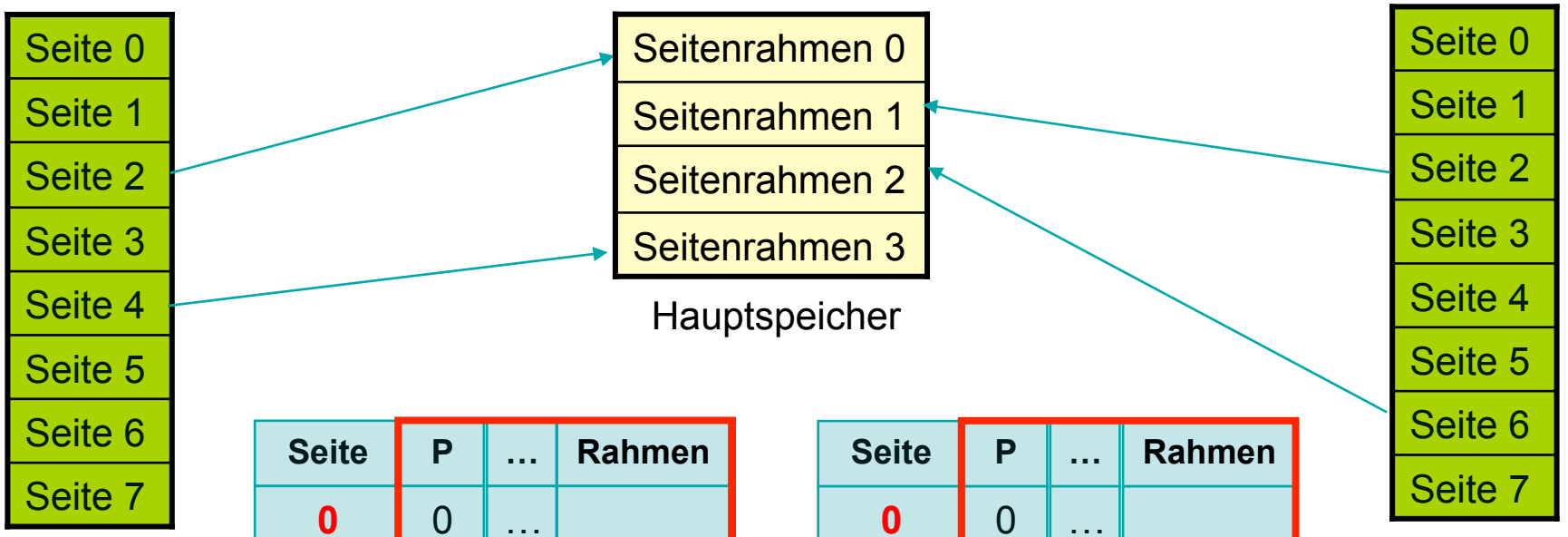
Verdrängung (1)

- 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)

- Ist das Modify-Bit gesetzt, dann muss Seite im entsprechenden Rahmen auf Festplatte zurück geschrieben werden
- Seitentabelle muss aktualisiert werden
- Generell: Suche von bestimmten Seiten-rahmen in Seitentabelle von Prozessen ineffizient
- Weitere Tabelle: Abbildung von Seiten-rahmennummer auf <Prozessnummer, Seitennummer>

Verdrängung (3)



Seite	P	…	Rahmen
0	0	…	
1	0	…	
2	1	…	**0**
3	0	…	
4	1	…	**3**
5	0	…	
6	0	…	
7	0	…	

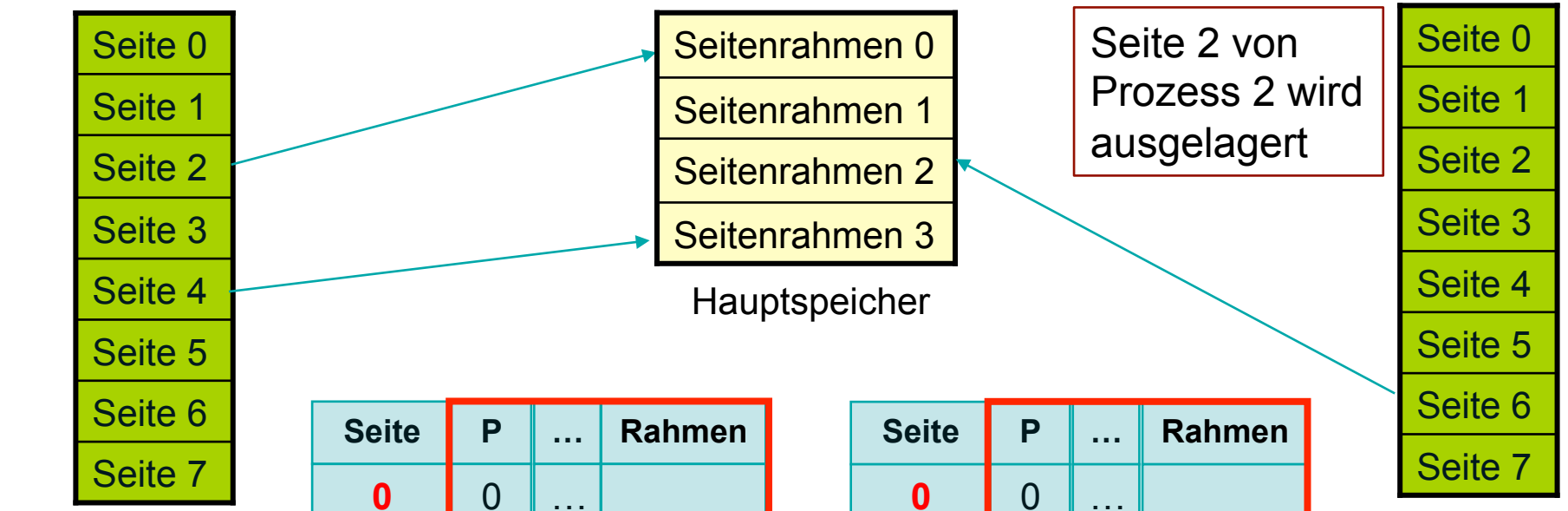
Seitentabelle von Prozess 1

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)



Seite 0 von Prozess 1 soll eingelagert werden

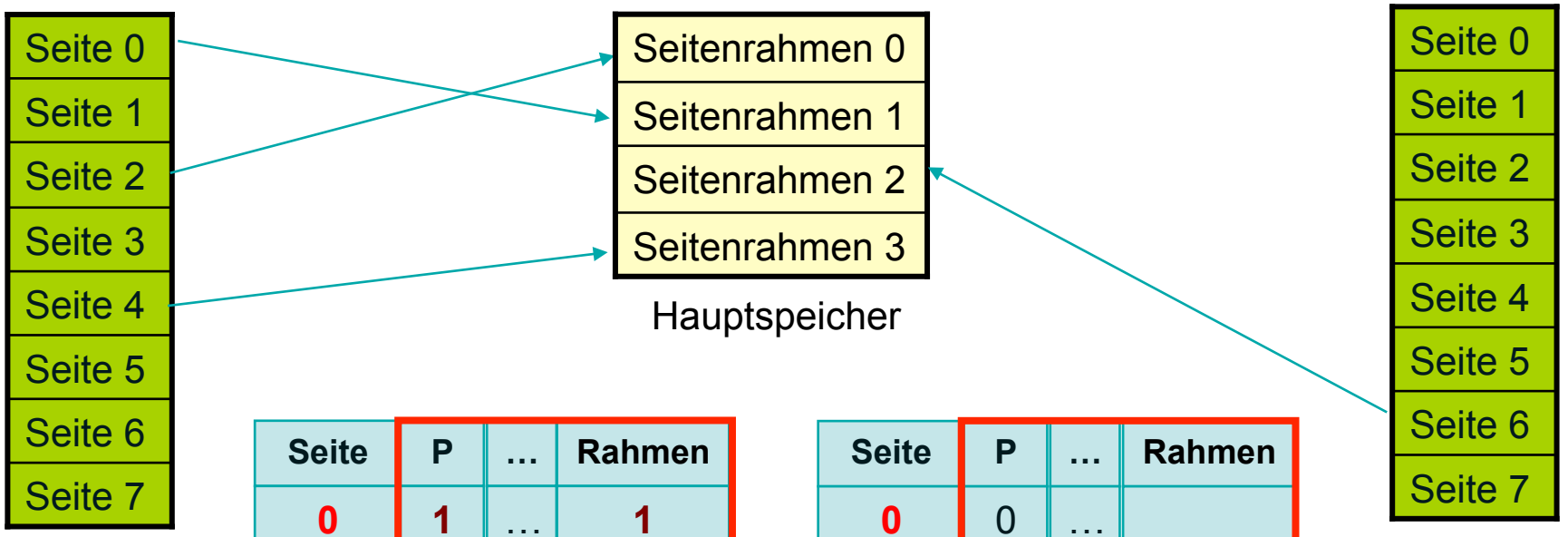
Seite	P	…	Rahmen
0	0	…	
1	0	…	
2	1	…	**0**
3	0	…	
4	1	…	**3**
5	0	…	
6	0	…	
7	0	…	

Seitentabelle von Prozess 1

Seite	P	…	Rahmen
0	0	…	
1	0	…	
2	**0**	…	
3	0	…	
4	0	…	
5	0	…	
6	1	…	**2**
7	0	…	

Seitentabelle von Prozess 2

Verdrängung (3)



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

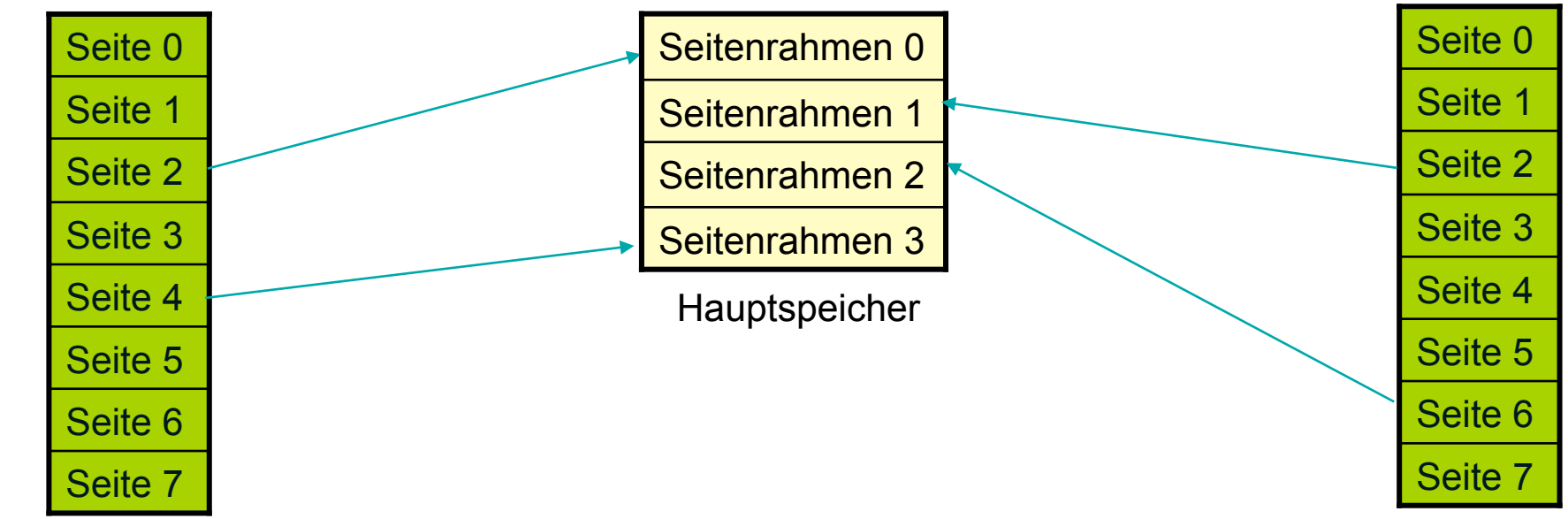
Seitentabelle von Prozess 1

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)



Rahmen	Prozess	Seite
0	1	2
1	2	2
2	2	6
3	1	4

Abbildung Seitenrahmennummer auf <Prozessnummer, Seitennummer>

Größe von Seitentabellen

- Problem: Größe der Seitentabelle bei großem virtuellen Adressraum
- Beispiel: 32-Bit-Adressraum
  - Annahme: 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 (Seitengröße 2^{12} Byte)

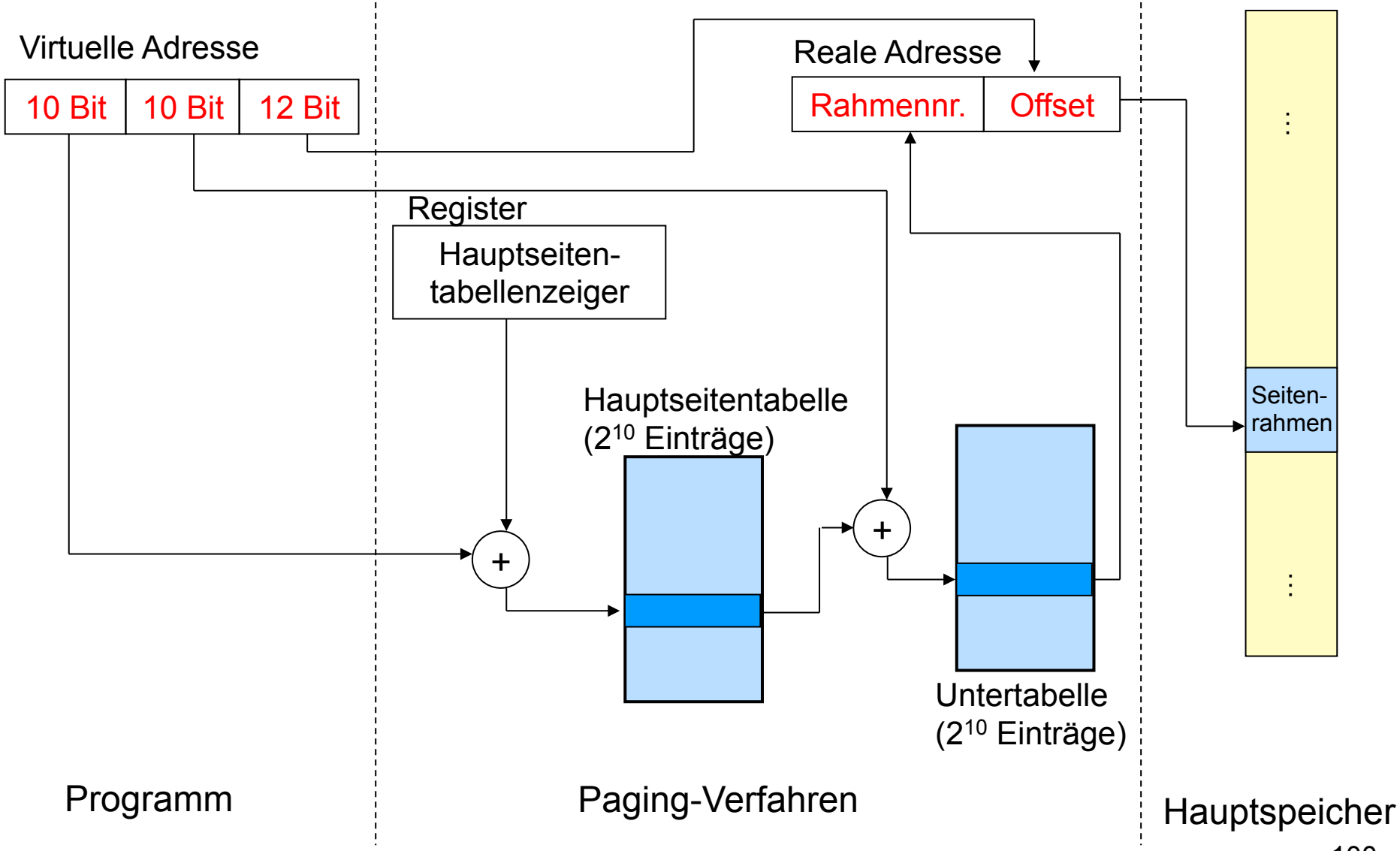
Zweistufige Seitentabellen (1)

- Hierarchische Seitentabelle (Pentium)
- 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 Hauptseite, die immer im Speicher liegt (2^{12} Byte, entsprechend Rahmengröße)
- 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 einen freien Seitenrahmen
- Mittlere 10 Bits der Adresse: Index von Seitenrahmen in Untertabelle
- Damit bis zu 2^{20} Seiten referenziert
- Restliche 12 Bit der virtuellen Adresse: wie vorher Offset innerhalb des Seitenrahmens

Adressumsetzung



Invertierte Seitentabellen (1)

- Für noch größere Adressräume (64-Bit-Systeme)
- 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
- Vorteil: Platzersparnis!

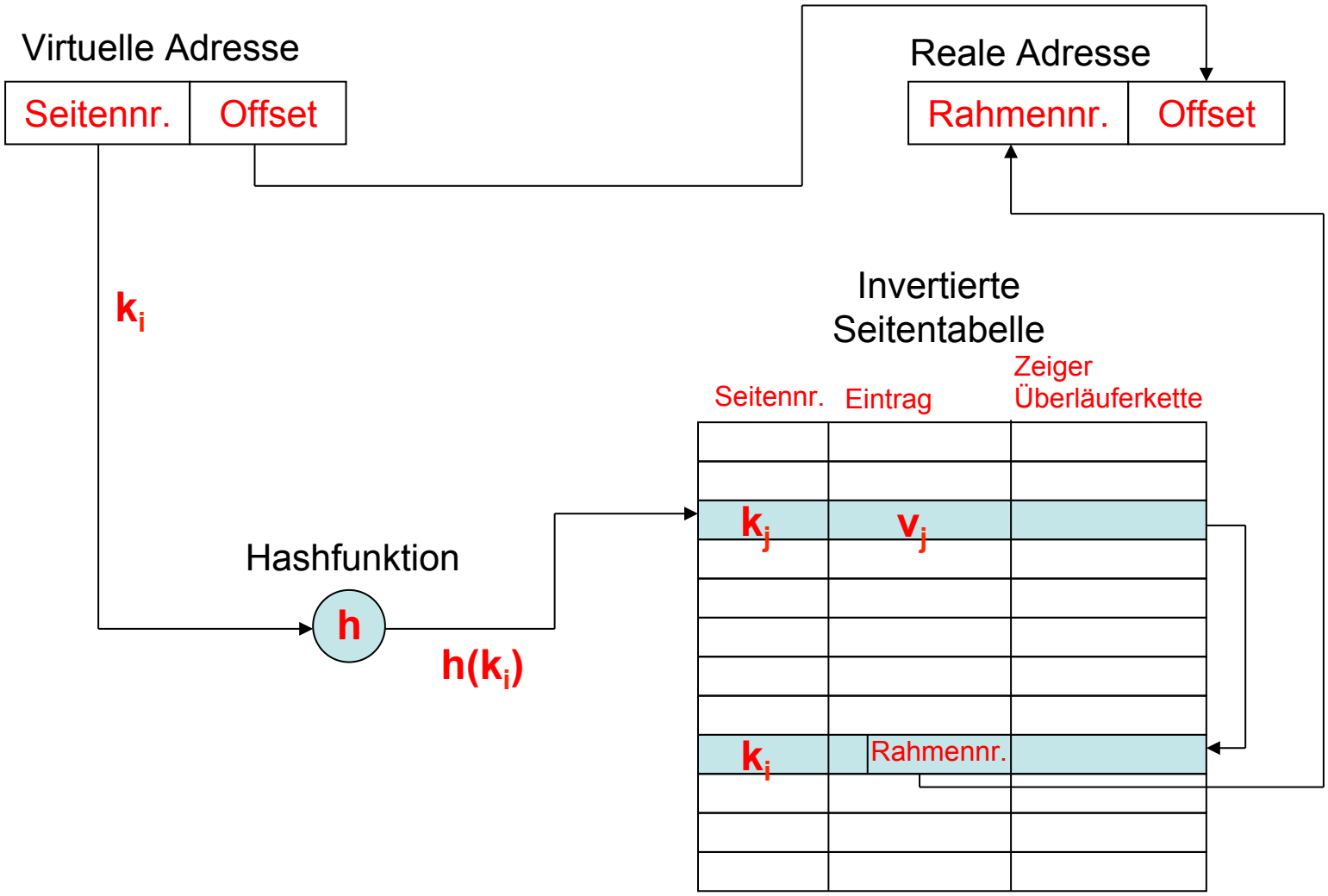
Invertierte Seitentabellen (3)

- Nachteil: Aufwändiger, eine virtuelle Adresse auf eine physische abzubilden
- Benutze eine Hashtabelle (s. Info II) um Seitennummern mit Rahmen zu speichern
- n = #Seiten, m = #Rahmen, Hashfunktion:
  $h : \{0, \ldots, n\text{-}1\} \rightarrow \{0, \ldots, m\text{-}1\}$
- 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 demselben 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)



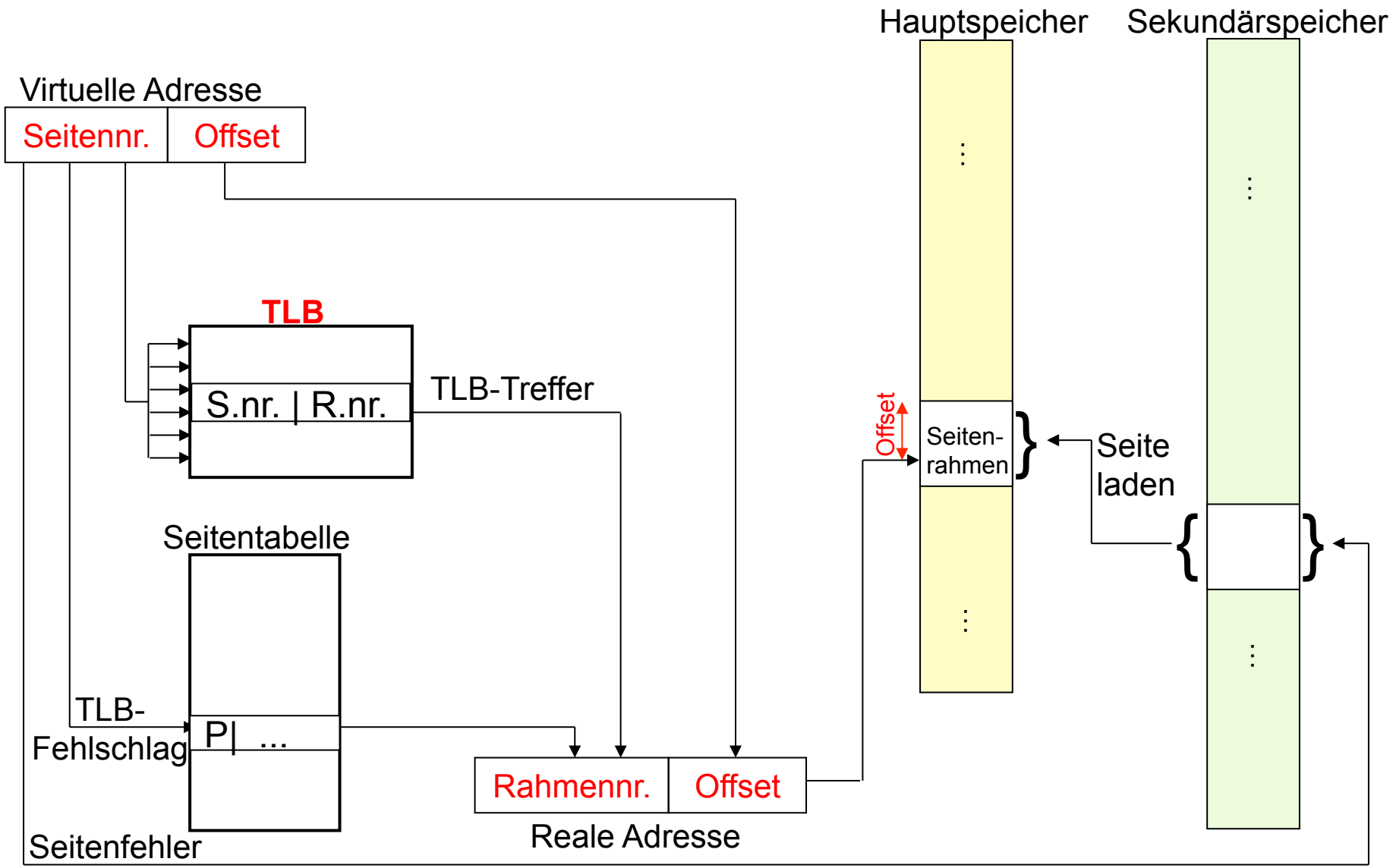
Translation Lookaside Buffer (TLB)

- Bei Speicherzugriff mit Paging: mind. ein zusätzlicher Zugriff auf die Seitentabelle
- Mindestens doppelte Zugriffszeit
- Hardwaremäßige Beschleunigung durch einen zusätzlichen Cache für Adress-übersetzung (Adressumsetzungspuffer, TLB)
- TLB enthält Seitentabelleneinträge, auf die zuletzt zugegriffen wurde (entsprechend 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
- Wenn Seite nicht im Hauptspeicher: BS muss Seite von Festplatte nachladen und Seitentabelle aktualisieren

Verwendung des TLB

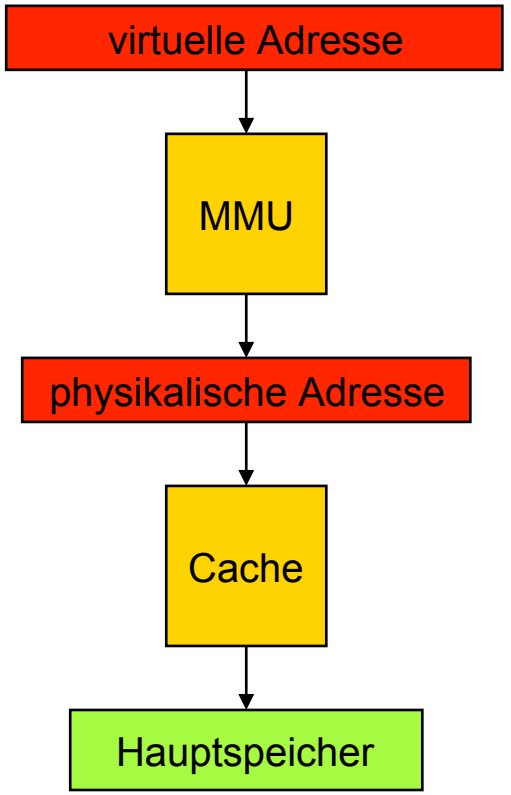


TLB: Assoziative Zuordnung

- 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: Verdrängungsstrategien notwendig

TLB und Caches

- Zusätzlich noch Caches für Programme und Daten
- Caches sind TLB / MMU nachgeschaltet
- 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 KB bis 4 MB (Pentium), oder auch 4 KB bis 1 GB (Intel core i7)

Segmentierung (1)

- Speicher eines Prozesses aufgeteilt in Segmente mit verschiedener Größe
- Größe der Segmente unterschiedlich und dynamisch festlegbar
- Nicht notwendigerweise zusammen-hängende Speicherbereiche
- Nicht alle Segmente eines Prozesses müssen im Speicher vorhanden sein

Segmentierung (2)

- Keine interne Fragmentierung, aber (geringe) externe Fragmentierung
- Aufteilung in Codesegmente, Daten-segmente, … (durch Programmierer, Compiler)
- Schutz und gemeinsame Nutzung auf Segmentbasis einfach zu regeln
- Segmenttabelleneintrag:

P	M	Weitere Bits	Länge	Basisadr.

Segmentierung (3)

- Komplexere Adressumrechnung (notwendig: Startadresse und Segmentlänge)
- Zuteilungsalgorithmen benötigt (z.B. First fit oder Best fit)
- 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
- Wird in Praxis nicht mehr benutzt

Aspekte der Speicherverwaltung

Wichtige Entscheidungen:

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

Abrufstrategie

- Demand Paging:
  - 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-Algoritmus:
  - Use Bit: Anfangs 0; wird hochgesetzt, wenn Seite nach Seitenzugriffsfehler referenziert wird; wieder auf 0, wenn leerer Rahmen gesucht wird
  - Geringer Overhead, gute Leistung
  - Linux: Altersvariable, die bei Zugriff erhöht wird

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 zugeteilten 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
  - Regelmäßiges Schreiben in Gruppen

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
  - Prozesse mit dem kleinsten Resident Set
  - Größter Prozess
  - Prozess mit größter verbleibender Ausführungszeit

Zusammenfassung (1)

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

Zusammenfassung (2)

- Komplexe Hardware/Software nötig für einen einfachen Speicherzugriff
- Verweis auf Seitentabelleneintrag kann im im TLB, RAM 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